THE UNIVERSITY OF CHICAGO

DECLARATIVE COMPUTER GRAPHICS USING FUNCTIONAL REACTIVE PROGRAMMING

A DISSERTATION SUBMITTED TO

THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES

IN CANDIDACY FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

LAMONT KENNETH SAMUELS

CHICAGO, ILLINOIS

AUGUST 2016

ABSTRACT

Most graphics applications are highly interactive and event-driven. These applications must continuously interact and respond to events firing from the outside environment, such as mouse and keyboard events, while also updating their internal state in response to these events. As an application grows in complexity, the process of handling these external events and maintaining a valid application state becomes more challenging.

Functional Reactive Programming (FRP) is another approach to programming interactive applications. FRP languages define abstractions, called signals, to express time-varying values. Signals can either be continuous (always changing) or discrete (changing at specific points in time). These abstractions allow for modeling external events or internal changes in a more direct and declarative way by making their behavior more explicit. The FRP system automatically manages dependencies between signals, which allows programmers to express their logic at a higher-level instead of dealing with the low-level implementation details of mutable state and callbacks that are required in the event-driven paradigm.

In this dissertation, we continue to advance the abstractions provided by prior FRP languages by presenting a new language called Tesel, which provides a new abstraction (*i.e.*, signal groups) for distinctly propagating discrete and continuous signal changes efficiently through the program. This dissertation presents a detailed description of the language and its implementation, a formal operational semantics and the implementations for the Tesel compiler and runtime system. We also show how Tesel can be used to write practical and efficient graphics applications.

To my family

ACKNOWLEDGMENTS

First and foremost, I would like to thank my loving family. My parents, Salonn and Derryl, and my sister, Tiffany, for their endless love and support in my success. I dedicate this dissertation to you all, thank you.

Special thanks to my advisor, John Reppy, for his guidance and instruction throughout the course of my graduate career. It has been a great honor to work with him. The knowledge he has given me on various aspects of computer science has made me a better computer scientist. Thank you, John.

I would like to thank my committee members, Ravi Chugh, and Stuart Kurtz, for their invaluable feedback on many aspects of my dissertation. Without your guidance and support, especially in the the final weeks leading up to the dissertation deadline, this work would not have been possible.

Many thanks to my fellow graduate students and friends for their moral support throughout the past six years. In particular, I would like to thank my fellow PL peers, Lars Bergstrom, Charisee Chiw, Kavon Farvardin, Joe Wingerter, and Brian Hempel for the invaluable and countless hours spent talking about all aspects of life. Thank you to all the friends I have made in Chicago, especially Erik Bodzsar, Bruno Betat, Chris Bun, and Jeff Strayer for making my time here an absolute pleasure. Many thanks to Michael Wallerich for his unwavering support and advice in my personal and professional success. You all truly kept my spirit lifted and made my graduate studies a time in my life that I will never forget.

Finally, I would like to thank the diligent administrative and technical staff and faculty members of the computer science department. Whenever I needed advice or help with completing a task, you all were always there to help me. Special thanks to Anne Rogers for her encouragement and advice, especially during my darkest hours as a graduate student. Anne, words cannot describe how much your confidence in me truly pushed me through my graduate studies. I thank you dearly.

Portions of this research were supported by the National Science Foundation under award CCF-

1446412. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Most graphics applications we write today are highly interactive and event-driven. Events originate from within the application or from an outside source. For example, an application must react to and handle multiple input events (*e.g.*, mouse clicks, key presses, multi-touch gestures, *etc.*) that asynchronously arise from the graphical user interface (GUI). Thus, applications must continuously maintain interaction with the outside environment, handle the processing of these events and execute tasks in response to an event such as updating application state or displaying data [64, 66]. To further illustrate these application responsibilities, Figure 1.1 shows an example of a dependency graph between typical entities (orange boxes) in graphics applications and events sources (blue clouds).

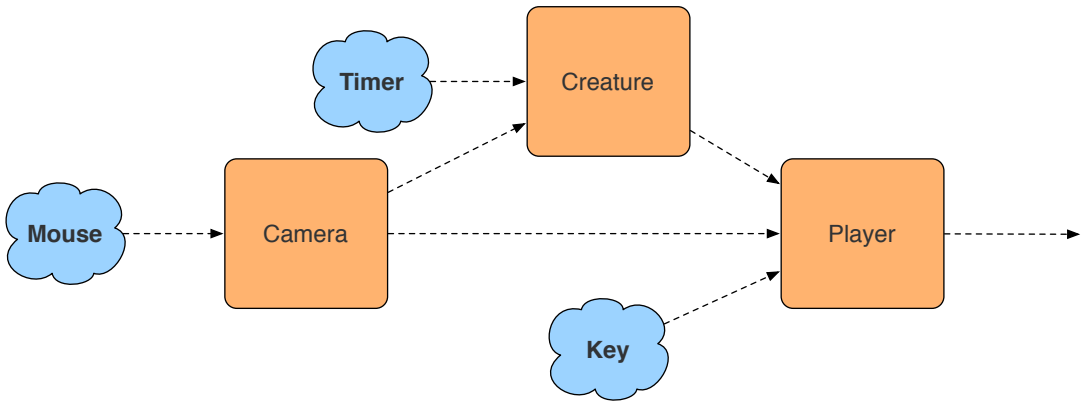


Figure 1.1: An example of showing the dependencies between entities (orange boxes) and events (clouds) in an application. The arrows indicate that an entity depends on the values of other entities and event sources in order to update its value.

The dashed arrows model dependencies: the output of an entity is computed from certain input entities or event sources. In other words, the updating of an entity requires retrieving the current value of its inputs whether they be from other entities or by the firing of new event values from the outside environment. As shown in Figure 1.2, the firing of a key event causes an update to the player entity. Since the outside environment did not fire any other events at the same time, the

other application entities do not require an update. Thus, the player entity uses the current values of the camera and creature entities along with the new key event to update itself.

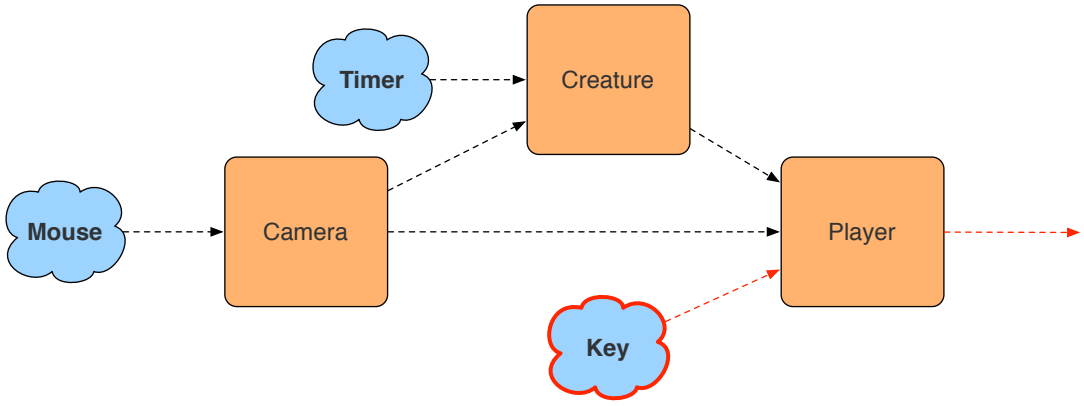


Figure 1.2: The firing of an event source causes its dependents to update their values. The dependent needs to ensure it receives the most current value of each input before updating its value.

The complexity of handling events and maintaining a valid application state comes when the outside environment fires multiple external events. These events may cause a race condition of updating multiple internal entities of the application. As Figure 1.3 illustrates, the firing of a timer event (*e.g.*, firing after every few seconds) happens at the same time as a key event firing.

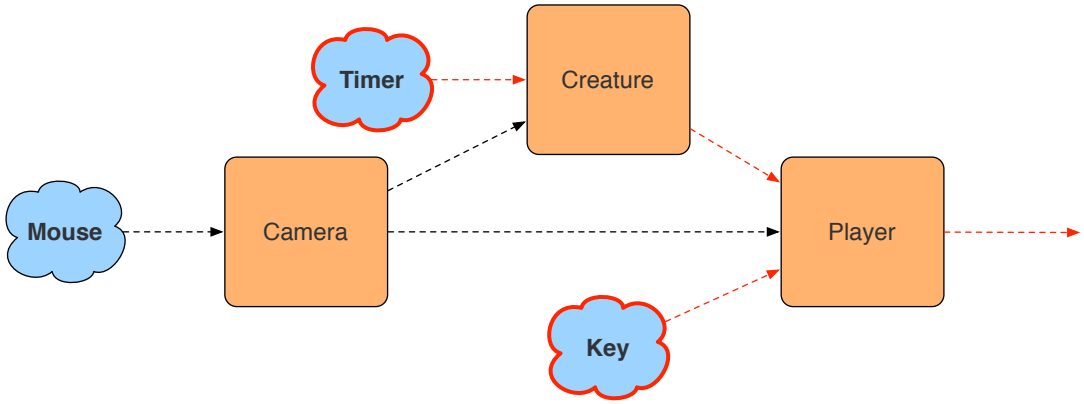


Figure 1.3: This illustration shows an example of the need to process events correctly. The timer event should be processed before the key event to ensure the player entity uses the updated creature value and not the old value.

Since these events are happening asynchronously, the application needs to ensure all dependen-

cies are updated correctly when processing the events. If the application updates the player entity first before updating the creature entity, then it needs to update the player entity again after updating the creature entity. Otherwise, not updating the player again after updating the creature leads to an invalid application state because the player entity uses the creature's old state to update its value. Thus, the correct ordering of processing events is important to avoid needless recomputations of entities or an invalid application state. As Figure 1.4 depicts, the complexity of handling these updates only grows as more external and internal changes are happening within the application.

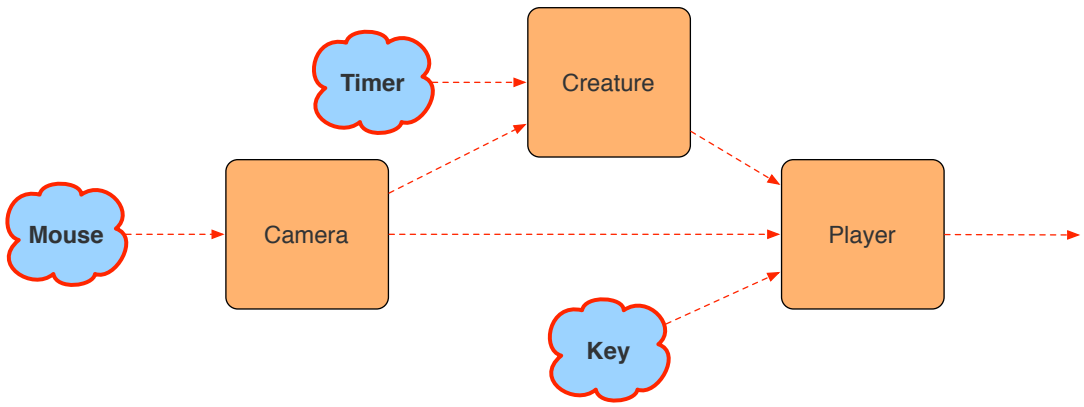


Figure 1.4: Multiple events firing causes multiple updates to the entities, which adds on to the complexity of managing updates.

Handling all the tasks mentioned above makes event-driven applications quite difficult to maintain and implement, since the arrival of external events is unpredictable and impractical to mange. As a result of programming within an event-driven programming model, a programmer must deal with inverted control flow, which requires a programmer to handle the external events of a program within functions that are invoked by the outside environment. This requirement forces the application to maintain a global state that is accessible within these functions such that as the state of the application changes the programmer can ensure that all dependent components are updated correctly. Such a task of managing state changes and data dependencies is complex and error prone owing to potentially handling state changes at the wrong time or in the wrong order [17, 66].

Many interactive applications use the mechanism of asynchronous callbacks (*i.e.*, event han-

dlers) to handle reacting to external events. But typically managing callbacks is a burdensome task and results in the problem infamously known as Callback Hell [28]. The difficulty comes from having to deal with unpredictable execution orders owing to sections of code that can potentially manipulate the same data. Additionally, callbacks do not always have a return value; therefore, they must perform side effects to the application state [18]. This issue is further illustrated in real-world applications such as Adobe Photoshop where the company reported in 2008 that a third of Photoshop's code is devoted to event handling, and half of the bugs reported during a product cycle exists in the event-handling code [60]. These numbers show how the number of bugs is disproportional to the amount of code that event handling represents and further illustrates that there is a need for a better option to reduce these bugs.

Another example of real-world applications that are implemented in event-driven paradigm are video games [21]. Many games can contain various interconnected components or entities such as creatures in the game interacting with each other. Figure 1.5 shows an example of one way to implement a modern video game level. Games can have various subsystems that largely depend on each other. The complexity of implementing the game grows even further as more levels are applied since it would require additional code to manage the interactions between them. Additionally, the game needs to handle the external events from the outside world (e.g., key presses) and update the internal components of the application, which change when these new events occur. Most games have their event queue as the backbone for handling the interactions between their components [58]. But this burden requires the programmer to manage the low-details of creating the necessary hooks for the system and maintaining this throughout the application. These event queues are usually implemented using concurrent constructs, which entails the additional work of limiting overhead and maintaining correctness.

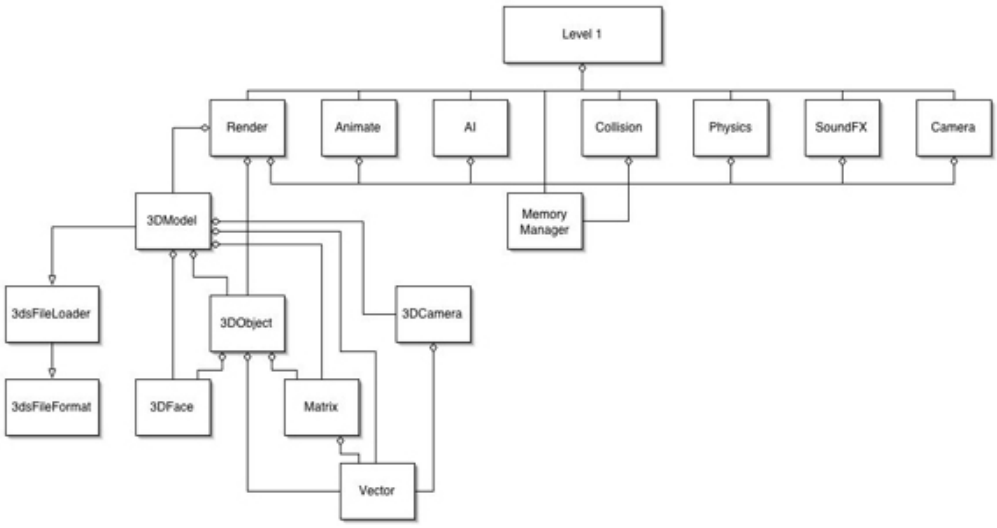


Figure 1.5: Various components that can make up a game level [48].

1.1 Implementing graphics applications

In computer graphics, programmers also have to deal with the complexities from the interactions between the hardware that performs the rendering of entities and the application itself. A graphics processing unit (GPU) is the hardware primarily responsible for rendering the meshes[1] in graphics applications. These highly efficient and parallel devices accelerate the process of transforming the meshes into an image to be shown on screen. To interact with a GPU, a programmer usually uses a low-level (*e.g.*, OpenGL[35]) or a high-level (*e.g.*, SceneKit[43]) API. Low-level APIs provide more fine-grained control over the GPU. The programmer explicitly handles the process of stating to the API how to create, transfer, and render mesh data. This control allows developers to tune their applications to achieve the best performance. Although low-level APIs can give the programmer a greater deal of performance power and flexibility, fixing errors when they occur can become a difficult and tedious task. GPUs are notorious for being complicated devices to debug. The debugging features provided by these APIs may not always provide enough information about how or why an error occurred (*e.g.*, forgetting to reset a uniform variable) [70].

1. A mesh is a collection of vertices, edges and faces that define a polyhedral shape. The data that represents a mesh is transformed by the GPU to generate an image for the screen.

High-level APIs abstract away many of the low-level details for the programmer. Consequently, one is losing the explicit control of fine tuning an application at the expense of greater expressibility and flexibility. Many of these APIs are built on top of low-level APIs and provide additional features for loading meshes, textures, and standard mathematical types and functions used in computer graphics. These features decrease the burden on the programmer to handle the nuisances associated with rendering and many low-level coding costs. But understanding the low-level details and the rendering pipeline is beneficial to gaining better performance when working at a higher-level. Thus, there is a conflict between wanting to have both expressive higher-level features and low-level tuning power without the tedious programming costs, which not many (if any) high-level APIs provide. For example, Figure 1.6 describes the steps (in pseudocode) to animate an object across the screen. These steps are carried out with lots of tedious low-level code where most of the code describes how to animate the object, instead of what the animation should do (*e.g.*, linearly interpolating the object back and forth). The issue persists in a similar way in SpriteKit as shown in Figure 1.7. The framework hides the low-level details of frame update cycle and uses method callbacks to allow the program to update their meshes every frame. Hence, an animation is updated either manually by a callback to the update method or using SKActions [43].

```
Allocate and initialize windowing and graphics APIs;
Setup sprites (i.e., textures);
while window is not closed do
    t = get current time;
    clear back buffer;
    forall the sprite (back to front) do
        compute position, scale, etc. at time t;
        draw to back buffer;
    end
    swap back and front buffers;
end
Deallocate textures, graphics and window contexts;
```

Figure 1.6: Pseudocode for animating an object each frame.

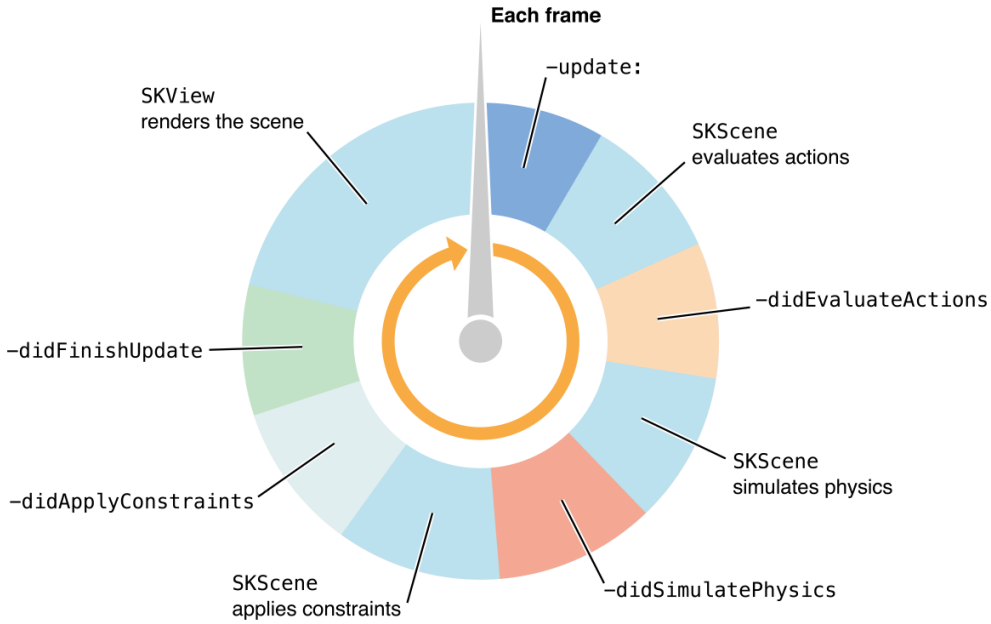


Figure 1.7: The per-frame update cycle in SpriteKit

With all the intricacies that come from the event-driven paradigm, an alternative approach is needed to lessen the burden of developing graphics applications. An alternative strategy to handling the render loop and managing dependencies between entities and events is by using a *declarative* approach. This method places the responsibility of managing the irrelevant details on the compiler while allowing the programmers to think at a higher-level about "what" should be presented versus "how" a computer should present it.

1.2 Thesis

Reactive programming [8] is an alternative approach to developing event-driven applications. The paradigm provides two key abstractions: *behaviors* (also termed *signals* in various works), which are continuous time-varying values, and *events*, which are sequences of discrete time-value pairs. With these abstractions, reactive programming provides programmers with a higher-level of abstraction to express programs in terms of *what to do*, while having the implementation automatically handle *how to do* it. The paradigm's underlying execution model automatically propagates

changes through a static or dynamic network of dependent computations. The usual example researchers provide for explaining reactive programming is a spreadsheet application. A spreadsheet contains cells that are either literal values (*e.g.*, "A1 = 3","B1 = 4"), or formulas that use the current values of other cells in its computation (*e.g.*, "C1 = A1 + B1"). If the value of a cell used in the computation of a formula cell changes then the formula cell automatically changes.

Functional Reactive Programming (FRP) [30] is reactive programming embedded in a purely functional language. Along with the benefits of the reactive programming paradigm, FRP also includes additional benefits from functional programming. A functional language treats computation as the evaluation of mathematical functions. The result of a function only depends on the inputs to it; therefore, calling a function with the same input multiple times will produce the same output each time. Furthermore, many functional languages avoid side-effects (*i.e.*, changes in state that do not depend on function inputs). This property leads to programs being more maintainable and predictable in behavior. Thus, the combination of the reactive and functional features allows for a truly pure and declarative approach to programming.

In this work, we show how the FRP paradigm decreases the burdensome responsibilities of managing mutability and events associated with implementing graphics applications. In particular, we propose an embedded language inside the Swift programming language [44] called *Tesel* that allows programmers to develop low-level graphics applications quickly and efficiently using the declarative nature of the FRP paradigm.

1.3 Contributions

This dissertation makes contributions in both the areas of continuous-based FRP and low-level computer graphics in the following way.

- We develop a new embedded FRP language that makes writing event-driven graphics applications less cumbersome.

- Within Tesel, we define the notion of *signal groups* as objects that encapsulate and manage the execution of a collection of signals efficiently. In particular, signal groups allow the programmer to separate the dependency graph into sections that can be executed in parallel. Thus, independent and time consuming signal groups are executed in parallel to achieve better performance.

- We formalize the evaluation model of Tesel by providing a formal semantics.

- We develop a runtime system that efficiently handles various event-systems that makeup a graphics application such as the windowing, event queue, and rendering systems. Furthermore, the runtime system manages the execution of the signal groups and signals and propagates any events from these event-systems to the signals and signal groups.

- We provide a compiler that creates a standalone application based on the code written using Tesel.

- Within Tesel, we define intuitive rendering constructs that are intertwined with our FRP abstractions to remove the burden of dealing with the render-loop and the management of rendering low-level mesh data.

- We show through developing real-world applications the practicality of our language and demonstrate through these applications how well our parallel execution of signal groups scale in comparison to their sequential execution.

1.4 Dissertation overview

The remaining chapters of this dissertation elaborate on the design, implementation, and evaluation of the Tesel language. The outline of this dissertation is as follows.

- Chapter 2 provides additional background information on FRP, and how Tesel relates to it.

- Chapter 3 gives an overview of the core language features of Tesel and compares Tesel to other continuous-based FRP languages.
- Chapter 4 describes the rendering constructs provided by Tesel.
- Chapter 5 provides an operational semantics for a core subset of Tesel that focuses primarily on its FRP combinators.
- Chapter 6 describes the implementation of the Tesel language, compiler and runtime.
- Chapter 7 describes the practically of Tesel by developing real-world applications.
- Chapter 8 surveys related work.
- Chapter 9 concludes.

CHAPTER 2

BACKGROUND

The FRP paradigm brings with it powerful mechanisms that allow for composability, abstraction and clarity [30, 41]. Typical applications implemented in FRP languages model physical systems such as handling the movement of entities in video games [21]. The declarative nature of the FRP paradigm makes it much easier to implement many of these physical laws [55, 74]. In this chapter, we discuss the history of FRP and how it has evolved to its current form. We conclude by providing background on embedded-domain specific languages and why we chose Swift as our host language for Tesel.

2.1 Synchronous Data-flow Principle

Most FRP languages support the notion of the *synchronous data-flow principle* [9, 36]. The principle models reactions in the system as being instantaneous. Reactive languages also model reactions in the system as being instantaneous but also allow reactions to be continuous. Thus, these languages are closely related to earlier work in synchronous data-flow languages such as Signal [34], Lustre [15], or Lucid Synchrone [63]. In these languages, programs are formed by using wiring primitives to compose processing elements (*i.e.*, a set of primitive operations) into a static hierarchical network. This model provides a natural way of expressing program modularity, since larger programs are composed hierarchically of smaller processing elements, which are themselves reactive programs. But FRP languages provide further benefits such as supporting *dynamism* within the structure of the system. A system providing dynamism allows its structure to be reconfigured over time, as the time varying values change within the system. FRP systems provide constructs that describe dynamism in a declarative way, which results in these systems being highly suitable for reactive systems in areas of computer graphics such as video games [16, 21] and virtual reality [11].

2.2 Evaluation Models of FRP

The evaluation model of an FRP language determines how changes propagate through the data-flow network of the reactive program (*i.e.*, are behaviors continuous in nature, discretely updated based on an event occurrence or both). The evaluation system is required to automatically propagate changes from nodes in the graph and recompute any dependent node affected by any changes from its input nodes. In the FRP literature, two types of evaluation models have been developed to support this automatic propagation:

Pull-based: A pull-based model requires the dependent nodes to obtain the new values from its input nodes. This model is commonly referred to as a *demand-driven* model because propagation is only initiated when a node needs the new piece of data to complete its computation. This property allows the pull-based model more flexibility because nodes that need this new value can pull it at their discretion.

As presented in the vast majority of the FRP literature, the pull-based model can produce a major performance issue in regards to having significant latency between an event occurrence and the propagation of that reaction to the other nodes in the network. This delay produces what Paul Hudak *et al.* describe as *time* and *space* leaks:

> "a time-leak in a real-time system occurs whenever a time-dependent computation falls behind the current time because its value or effect is not needed yet, but then requires catching up at a later point in time. This catching up can take an arbitrarily long time (time-leak) and may or may not consume space as well (space-leak)" [40].

This issue mostly stems from FRP languages that are embedded within functional languages that use a lazy evaluation strategy. Fixing this problem has been a significant portion of the research done in the FRP area and many languages have produced solutions that limit the expressive power of the language and restrict behaviors to be non-first class citizens [40].

Push-based: In a push-based model, if an input node has a new value then it pushes that value

to its dependent nodes. This model is commonly referred to as a *data-driven* since propagation is driven by when an event happens instead of on demand. This model is used by many FRP languages implemented in eager languages that use callback methods[72]. A significant problem a push-based system needs to handle is how to avoid excessive and wasteful recomputations. This issue occurs since recomputations happen every time the input node changes.

2.3 FRP Languages

FRP languages provide reactive constructs to work with continuous time-varying values (*i.e.*, behaviors) and streams of timed values (*i.e.*, events). Just like the fundamental constructs of an imperative language consists of primitive operators (*e.g.*, assignments) and values (*e.g.*, numbers), FRP languages at the reactive level use reactive constructs to operate on behaviors and events to facilitate the construction of synchronous data-flow networks. The two general conceptual models used to implement FRP systems are known as *Classical FRP (CFRP)* and *Arrowized FRP (AFRP)*. In CFRP, behaviors and events are first- class citizens, whereas in AFRP transformers of behaviors and events are considered first-class citizens.

2.3.1 Classic FRP

The original FRP semantics were formulated by Paul Hudak and Conal Elliot for functional reactive animation (Fran) [30]. The embedded-language, Fran, was implemented in Haskell and provided a way to specify declaratively interactive multimedia animations. In the FRP literature, their initial model for FRP is referred to as "Classic FRP" and presented two key primitive type constructors: *Behavior* and *Event*. In CFRP, behaviors are modeled semantically as functions from time to a value:

$$Behavior\ \alpha\ =\ Time\ \mapsto\ \alpha$$

Time is considered continuous and is represented as a non-negative real number. The type parameter α specifies the values carried by the behavior. Behaviors are also a good way to model equations of motion in physics such as position and velocity. For example, if *Vector2* represents a 2-dimensional vector then velocity can be represented aa time-varying value with the type *Behavior Vector2*.

Events represent a sequence of time-stamped discrete values:

$$Event\ \alpha\ =\ \{\ (Time,\ \alpha)\}$$

Events can model any discrete value such as user input or time running out in a game. Originally in Fran, events were presented as inputs such as key presses and mouse clicks (e.g., *lbp* for left mouse clicks and *rbp* for right mouse clicks).

Behaviors and events are first-class values in CFRP and can be composed and built using reactive combinators. For example, Figure 2.1 shows the core type definitions for the reactive combinators that allow behaviors to switch to new behaviors when an event occurs. The *untilB* combinator allows for switching to a new behavior only on the first occurrence of the event. The first argument is the initial behavior followed by the event that carries a new behavior. The resulting behavior for the combinator acts as the initial behavior until the event occurs, then switches to the event's behavior. Similarly, The stepper combinator is similar to the untilB combinator with the exception that it repeatedly switches to the behavior provided by the event on every occurrence. Figure 2.1 also contains two operators for merging and tagging events. The operator *(-=>)* is a function that tags every event occurrence with a particular value, whereas the (.|.) operator produces events when either of the input events has an occurrence. The lifting functions (*liftB1*, *liftB2*) lift a pure function to a behavior level function. The *integral* integrates a value with respect to time. Finally, A *constB* function is defined for lighting values to behaviors. The functions are defined as follows.

```
-- Single-event reactivity
untilB :: Behavior a -> Event (Behavior a) -> Behavior a

-- Continous-event reactivity
stepper :: Behavior a -> Event (Behavior a) -> Behavior a

-- merge operator
(.|.)  :: Event a -> Event a -> Event a

-- tag operator
(-=>) :: Event a -> b -> Event b

-- Lifting
liftB1:: (a -> b) -> Behavior a ->  Behavior b
liftB2:: (a -> b -> c) -> Behavior a -> Behavior b -> Behavior c

-- Integration
integral:: Behavior a -> Behavior a

 -- Constant Behavior
constB  :: a -> Behavior a
```

Figure 2.1: The type definitions for the *switching* combinators in Classic FRP.

More concretely, Figure 2.2 shows a behavior that switches between the colors red or blue depending on a mouse click.

```
redOrBlue :: Behavior Vector4
redOrBlue = color
            where
            red =  constB(vector4XYZW 1 0 0 1)
            blue = constB(vector4XYZW 0 0 1 1)
            color = stepper red  (lbp  -=> blue  .|. rbp -=> red)
```

Figure 2.2: A code snippet that represents a color that is either red or blue depending on the mouse key clicked.

Lines 4–5 define two constant behaviors that represent the colors red and blue. Line 6 defines the *color* behavior using the tag and event operators along with the stepper combinator. The color is initially the red behavior until a mouse event occurs and then the stepper combinator changes the color to either red or blue.

Another CFRP example that models physical phenomena is shown in Figure 2.3. The figure provides an example of free falling ball. Figure 2.3a provides the mathematical equations and code to produce the behaviors. A ball is modeled as having a height and velocity. With this information, behaviors can easily be implemented by integrating the acceleration (*i.e.*, gravity) to compute the

velocity and integrating the velocity to compute the height. We assume the ball is modeled as a point mass, with an initial velocity of zero, and falling in one dimension (*i.e.*, the height of the ball above the ground). In this example, we start the ball at a height of 10 and an initial velocity of 0.

$$y = y_0 + \int v \, \mathrm{d}t$$
$$v = v_0 + \int -9.81 \, \mathrm{d}t$$

(a) Mathematical equations for a freely falling ball.

```
type Pos = Double
type Vel = Double

fallingBall :: Behavior (Pos,Vel)
fallingBall = liftB2 (,)  x y
               where
               y = y0 +  integral v
               y0 = 10
               v = v0 + integral a
               v0 = 0
               a =  -9.8
```

(b) A code snippet of representing a free-falling ball due to a gravitational force.

Figure 2.3: Free falling ball example in CFRP

Although CFRP provides a highly expressive language, which allows for writing declarative code easily, this expressive power came at a performance cost. CFRP uses the pull-based evaluation model. By using this model, early variants of Fran suffered from the latency issues associated with pull-based evaluation, such as difficult to determine time and space leaks. Relying on past values causes memory usage to grow significantly in proportion to the program running time. Although the focus of this dissertation is not to thoroughly examine or fix these issues, many FRP researchers have investigated these limitations and present possible solutions to limit their burden on program performance [68, 6, 31].

2.3.2 *Arrowized FRP*

Later iterations of CFRP focused on largely improving the performance of its evaluation model. One proposed system, Yampa [40], introduced a library based on an abstraction called *signal functions*. These functions were derived from the Arrow framework [42], which provides a generalization of functions. In the AFRP paradigm, arrows are abstract type constructors with input and output type parameters that can be used to express and compose a reactive program using special routing combinators.

CFRP assumed an fixed, implicit input system. The input sources (*e.g.*, mouse and key presses) were predefined by the system. This property means that behaviors and events can only produce output. In AFRP, inputs are explicit. Signal functions not only provide output but also take in input. Hence, signal functions are conceptually functions on signals (*i.e.*, behaviors in CFRP):

$$SF\ \alpha\ \beta\ =\ Signal\ \alpha\ \mapsto\ Signal\ \beta$$

Signal functions are the first class citizens in AFRP. Behaviors and events are second class citizens and only exist through signal functions. The motivations behind this decision are twofold. In CFRP, behaviors incurred performance problems because it could depend on the past and present values for its current value. In AFRP, a signal function can only be defined based on its input and specific primitives. Signal functions cannot implicitly preserve past data, and they cannot evaluate other signal functions at arbitrary times. This property eliminates a large class of space and time leaks in the implementation of the AFRP language. Secondly, this property increases modularity, since combinators can transform both input and output. CFRP only the output of a behavior was able to be transformed.

In AFRP, events are not a sequence of time-stamped discrete values. If the model did allow this type then variants of signal functions would need to be created for each possible input/output combination[68]. Instead, an event abstract type is defined:

```
data Event a = EvOcc a   -- an event occurrence
             | NoEvent   -- a non-occurence
```

Events are then embedded within signals, such as $Signal\ (Event\ a)$ to define a discrete-time signal of some type a. At time T, a discrete-time signal will have a value of **EvOcc a** if there is an event occurrence at time T and a value of **NoEvent** if there is no event occurrence at time T.

The primitive combinators for composing signal functions are shown in Figure 2.4. These primitives allow the construction of the program's data-flow structure. Additionally, a set of switching constructs is provided to support dynamically reconfiguring the network.

```
-- Lifting
arr :: (a -> b) -> SF a b

-- Constant signal function
constant :: a -> SF a a

-- Identity signal function
identity :: SF a a

-- Time signal function
time :: SF a Time

-- Integration (rectangle rule)
integral :: (Floating a) => SF a a
```

(a) Primitive combinators

```
-- Signal function composition
(>>) ::  SF a b -> SF b c -> SF a c

-- Splitting
(&&&) :: SF a b -> SF a c -> SF a (b,c)

-- Parallel pass-through
first :: SF a b -> SF (a, c) (b,c)

-- Single-event reactivity
switch SF a (b, event c) -> ( c -> SF a b) -> SF a b

-- Continuing reactivity
rSwitch :: SF a b ->  SF (a, Event (SF a b)) b
```

(b) Routing and Switiching signal functions combinators

Figure 2.4: The core primitives for AFRP

To demonstrate the use of these primitives, we can recreate the freely falling ball examples from section 2.3.1 in Figure 2.5.

```
type Pos = Double
type Vel = Double

fallingBall :: Pos -> SF () (Pos, Vel)
fallingBall y0 = (constant (-9.81) >>> integral )
                 >>> ((integral >>^  (+ y0))
                 &&& identity)
```

Figure 2.5: Free falling ball example in AFRP

As shown Figure 2.5, signal functions can become quite complicated and cumbersome to read when just using the arrows operators [57]. Most AFRP implementations also provide Paterson's

arrow notation syntax [62], which is a more convenient notation for programming using arrows. The notation allows for intermediate signals to be named within a block defined using the *proc* keyword. The falling ball example section could be described using this notational style as follows:

```
type Pos = Double
type Vel = Double

fallingBall :: Pos -> Vel -> SF () (Pos,Vel)
fallingBall y0 v0 = proc input -> do
    v <- integral >>^ (+ v0) -< (-9.81)
    y <- integral >>^ (+ y0) -< v
    returnA -< (y,v)
```

The *proc* keyword is analogous to λ in λ-calculus with the exception that it defines a signal function instead of a function. The overall structure begins with the input signal to the signal function appearing after the *proc* keyword. The lines after the input line have the basic structure where inputs signals are placed on the right, signal functions appear in the middle, and output signals are bound to identifiers on the left. The *-¡* operator is pushing the value into the signal function to be computed. Thus, we feed the gravity (*i.e.*, -9.81) to the integral function and the output is applied to the initial velocity using the $>>\hat{}$ operator to produce the velocity value that is assigned to *v*. This operator creates a signal function that takes the output of one signal function (*i.e.*, integral) and then applies a pure function (*i.e.*, (+ v0)) to produce the overall output for the signal function. This process is repeated to produce the position value by feeding in the velocity value into the signal function that integrates and adds the initial position to its output. The overall output of the fallingBall signal function is produced by the final line in the block, which is pairing the position and velocity together.

2.3.3 Problems with the Pull-based Model

Many of the continuous-based FRP languages use a pull-based evaluation model [31, 30, 73, 20, 40]. As stated earlier, this model fits well with the nature of continuous time semantics because

sampling is done on demand. Although AFRP fixed many of the performance issues (*i.e.*, space and time leaks) from prior FRP languages [30], AFRP and other continuous-based languages still suffer from two known problems that result from using this demand-driven model: *wasteful re-computations*, and *global delays* [31].

Certain values change continuously over time, while others change at discrete points in time such as the location of the mouse after a mouse click or object collision. With the exception of the purely continuous case, languages using a pull-based model of evaluation waste a considerable amount of resources recomputing values even when they do not change [31]. To further illustrate this problem, we will use the simple example shown in Figure 2.6. The primary signal function (*sfMain*) is composed of four signal functions (solid blue boxes) and is fed as input a tuple of three values: *a*, *b*, *c*. These input values are then fed to their appropriate signal function that we assume exists outside of the primary signal function. The colorful spinning wheel represents that a signal takes a significant amount of time to compute its output value. The output values of the three beginning signal functions (*sfA*, *sfB*, *sfC*) are fed to the final signal function (*sfD*) to compute the final output value for the primary signal function. The wasteful recomputations problem arises when a signal function only needs to be updated if its input changes (*i.e.*, at discrete points in time). For example, let's assume that we only want to update *sfA* if its input changes. If there are new input values for *b* and *c* but not *a* then *sfA* should not be recomputed. But AFRP languages, and in this case Yampa, have no way of determining that only one of the tupled inputs has changed. Thus, the long-running signal function (*sfA*) is recomputed, even though its input has not changed. Furthermore, AFRP and many other continuous-based languages think of signals as only be continuous (*i.e.*, always changing); therefore, the whole program is continuously changing and needs to be recomputed.

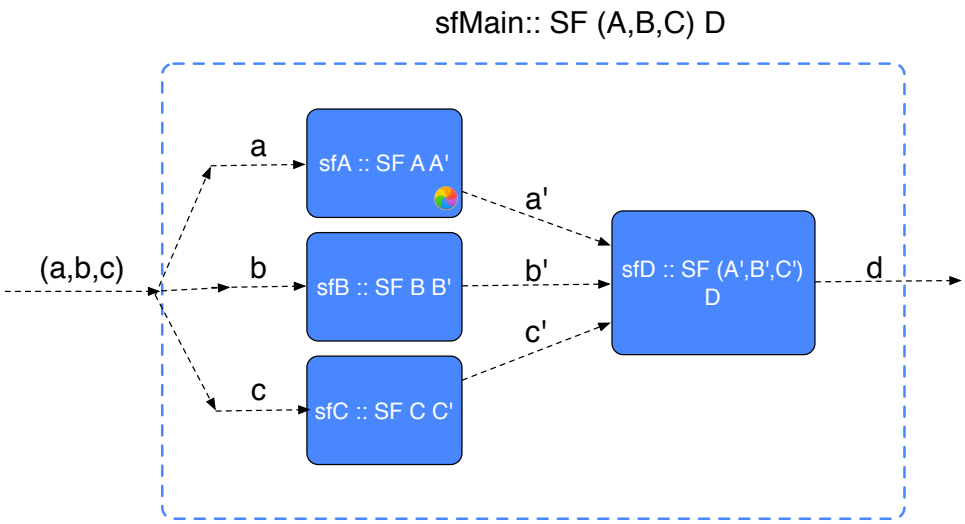


Figure 2.6: An example of a signal function that takes in a tuple of values as its input. The whole signal function is recomputed even if only one input has changed. Thus, the long running signal (sfA) is always ran (even a is not updated), which can cause performance problems.

The other problem with the pull-based approach is that it introduces significant latency into the program. There can be a lengthy delay between the occurrence of an event and the visible result of its reaction. These global delays are caused by the sequential execution of events. All continuous-based FRP systems enforce a strict ordering to events and require them to be processed in the exact order of occurrence [23]. For example, the signal functions in Figure 2.6 are evaluated in the order of the inputs given to *sfMain*. Thus, the long running signal function (*sfA*) blocks the execution of *sfB* and *sfC*, even though all three signal functions can be executed independently. This requirement of sequential ordering events results in incurring significant latencies in the program.

In contrast to the pull-based model used in many continuous-based languages, Tesel uses a hybrid-model that uses concepts from both the push-based and pull-based evaluation models. As shown in Figure 2.7, Tesel makes a distinction between continuous signals (blue boxes) and discrete signals (orange boxes). Continuous signals are always evaluated during each sampling period, whereas discrete signals are only evaluated when one of their input signals changes. Thus, the long-running signal $S_{A'}$ is only evaluated if the value of the signal S_A changes. We discuss our hybrid model in further details in Chapter 3.

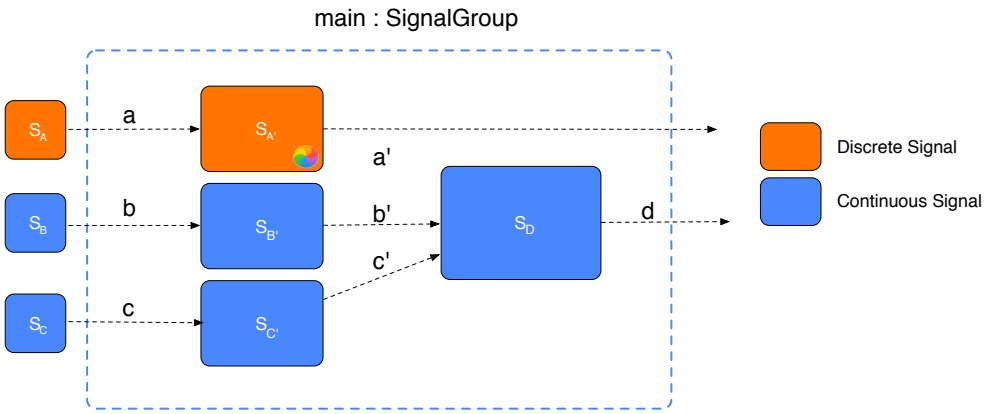


Figure 2.7: Tesel makes a distinction between discrete and continuous signals. This distinction allows for only changing signals when needed, which avoids the global delays and wasteful computations experienced by prior FRP languages.

2.4 Embedded Domain-Specific Languages

An *embedded domain-specific* languages (EDSLs) is a domain-specific language that is implemented as a library in a more powerful general-purpose host language. EDSLs inherit the features of its host language (*e.g.*, types, and constructs), while also providing domain-specific constructs that allow programmers to develop applications at a higher-level of abstraction. Many FRP variants are implemented as EDSLs in Haskell such as Fran [30], Yampa [57], Reactive [31], GrapeFruit [47], Elerea [61]. Implementing an EDSL has many benefits. In addition to allowing the EDSL to take advantage of the host language's features, embedding a language with a general-purpose language also provides access to pre-existing libraries, and tools (*e.g.*, IDEs, compilers and debuggers) without needing to develop them from scratch. if the end-users are already familiar with the host language then it makes it easier for them to learn the EDSL [38].

When developing an EDSL there are two common embedding approaches: shallow-embedding and deep-embedding [12]. Shallow-embedding is one where terms of the DSL directly translate to values of the host language. The representation used by the shadow-embedding directly encode their desired semantics. A shallow-embedding is useful for languages that continuously add new constructs, as long as the new constructs can easily be implemented by host-level operations. On

the other hand, a deep-embedding constructs a data structure (*e.g.*, AST) to represent the domain of the DSL. That data structure must the be executed to produce a value. A major advantage for deep-embedding is that its possible to transform and optimize the representation, but adding new languages constructs can be difficult task since it may require modifying the deep implementation and it transformation functions.

EDSLs do come with some disadvantages. The host language has no knowledge about the domain of the EDSL; therefore, domain-specific optimizations, and error messages [37] are not provided upfront. Although there is a possibility for runtime optimizations, these optimizations are usually less efficient than compile-time optimizations [56]. This issue is a well-known problem with EDSLs [38] and the usual fix to improve performance within these languages is to develop their own pre-processor [51, 69] or domain specific compiler [50].

2.4.1 Swift

One of the major goals in the near future for Tesel is implementing a backend for mobile devices. Currently, not many functional languages have adequate infrastructure or programmer support for developing mobile applications. However, the host language for this research, Swift, provides a number of developer tools such as an IDE (*i.e.*Xcode) designed for easily implementing, simulating, and testing mobile applications. Furthermore, the language provides large number of libraries that make it easy to implement mobile applications. Additionally, Swift is a multi-paradigm[1] programming language that includes many modern and useful features from the various programming paradigms that it supports. In particular, it contains aspects of functional programming that make it easier to implement FRP systems, such as higher-order functions. The language also supports a modern graphics API, Metal [45], which is used by many graphics programmers. This API is used by the Tesel runtime environment to gain the best performance on modern hardware instead

1. The language supports multiple programming paradigms that include: object-oriented, functional, imperative, and block structured.

of using older low-level APIs (*e.g.*, OpenGL) as described earlier. Thus, we choose Swift over a well-known FRP host language such as Haskell, with these important long-term design goals in mind.

We chose the Metal API for its ability to efficiently manage the flow of data to and from the GPU. As mentioned earlier, specialized APIs have been created to ease the process of rendering graphics. The most commonly used low-level APIs are OpenGL [35], and Direct3D [54]. OpenGL is a cross-platform API that is implemented on many operating systems, while Direct3D is a proprietary library for Microsoft's Windows family of operating systems. These APIs provide a way for programmers to access and interact with the GPU in an abstract way and to achieve hardware-accelerated rendering. The APIs can be thought of as abstraction layers between the application and the underlying graphics hardware. Programmers do not need to know the inner workings or the performance of the graphics hardware. As the application submits commands to the API, the API will handle interacting with the driver of the hardware to quickly and efficiently produce the desired result. The technology in graphics hardware is advancing rapidly and these older APIs are starting to develop bottlenecks in regards to performance. Thus, research in the area of low-level computer graphics has focused on creating new APIs related to fixing performance issues in older APIs such as state validation, resource management, and the lack of parallelism. The Vulkan [49], and Metal [45] APIs were created to enable lower overhead, more predictable performance, and better programmability. In particular, Tesel incorporates the Metal API into its runtime system, where the Metal API automatically and efficiently manages the data sent to GPU that is specified by programmer using our declarative constructs.

CHAPTER 3

TESEL AND FUNCTIONAL REACTIVE PROGRAMMING

This chapter gives an informal introduction to Tesel. We provide an overview of our hybrid model that also contains a new abstraction (*i.e.*, signal groups) that can further improve the implementation and programmability of the model. Lastly, this chapter discusses how we interact with the outside environment. Chapter 5 presents a more detailed and formal semantics of the language.

3.1 The Conceptual Model of Tesel

As mentioned in section 2.3.3, the performance issues owing to the implementations used by many CFRP and AFRP languages have led to the development of a hybrid model. This hybrid model combines the push and pull evaluation models together to efficiently implement its abstractions. A push-based evaluation model works well for discrete signals, whereas continuous signals benefit from using a pull-based implementation. Thus, the key idea of this hybrid model is to have a clear distinction between discrete and continuous signals such that the implementation can use the most efficient evaluation model for a given type of signal. As with other hybrid FRP systems [31, 47, 68], Tesel makes this clear distinction by providing three types of signals:

- **Event Signal** (`ESignal<A>`): Signals that are defined *only* at discrete points in time. For example, mouse presses only happen at specific points in time; therefore, we can represent them as event signals: `ESignal<MouseEvent>`.

- **Discrete Signal** (`DSignal<A>`): A signal that is given an initial value where this value is updated at discrete points in time. In a 2D video game, a player's movement that is controlled by the user can be represented with the type `DSignal<Float2>`, where its value represents its 2D position. The player will have an initial position, and this position will be updated as the user causes event occurrences (*e.g.*, key presses) in order to move the player.

- **Continuous Signal** (`CSignal<A>`): A signal that is always defined (*i.e.*, the same type of continuous signal defined in CFRP and AFRP). For example, we can represent the velocity of a 3D moving ball as a continuous signal with the type `CSignal<Float3>`.

The combinators described in section 3.2 define these three types of signals and allow dependencies to be created between them.

3.1.1 Signal Groups

Tesel also provides a novel abstraction to the hybrid-model, namely signal groups, which provide additional implementation and programmability benefits. The primitive functions and combinators used in FRP languages construct dependency graphs between signals as shown in Figure 3.1 (to keep the motivation for signal graphs simple, we show a dependency graphs that only contain discrete and event signals).

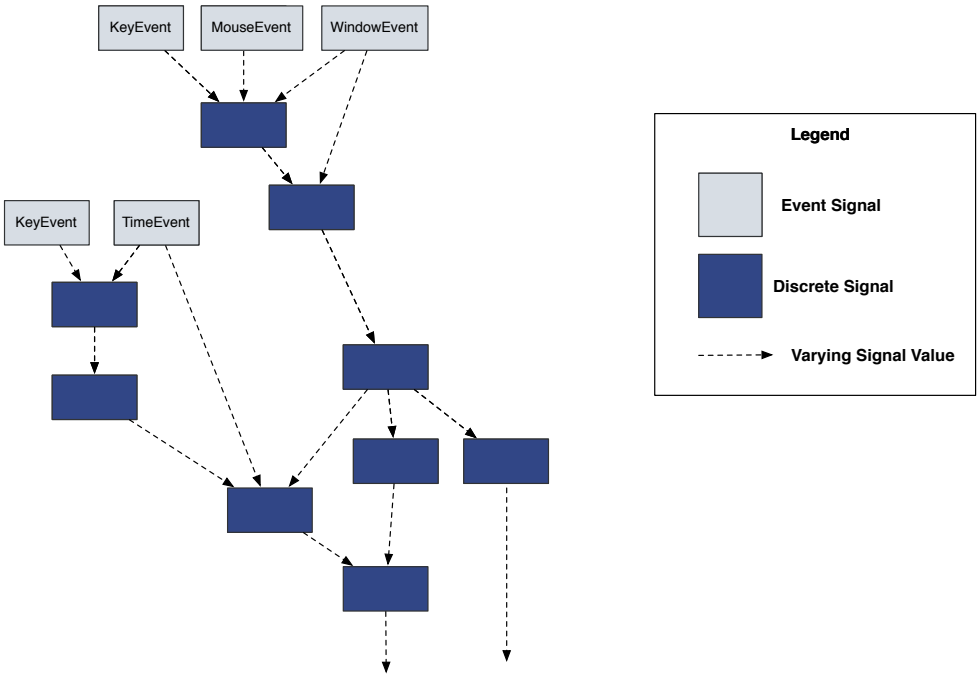


Figure 3.1: An example of a static dependency graph with only discrete signal types (*i.e.*, event and discrete signals).

Event occurrences from the outside environment (or from internal event signals) may cause other discrete signals to update. As signal values are updated, they flow to other dependent signals that use these values to update their values. But what if computing the updated value of a signal requires a significant amount of time? Furthermore, what if updating *portions* of the graph is a time-consuming process? These delays may cause a global delay for updating the entire dependency graph. Thus, if the system can identify parts of the graph that take a considerable amount of time to update then these parts can potentially be evaluated in parallel to improve performance.

Signal groups can act as a way to allow the programmer to separate time-consuming code such that the execution of the code within the signal group can be efficiently handled by the *implementation*. Figure 3.2 updates the prior dependency graph with signal groups that define local dependency graphs. Tesel contains combinators to connect these local signal groups together in order to flow data between groups. If groups A and B in Figure 3.2 are long running groups and a key press fires then since both groups are independent of each other the implementation can run them in parallel. After updating both groups, group C is evaluated since it depends on the values from both A and B. The programmer has full control on whether the groups are updated sequentially or in parallel by either enabling or disabling concurrency at compile time. Programmers declaratively write their application using signal groups and the details of identifying independent groups and executing them in parallel on an event occurrence is left to the system implementation.

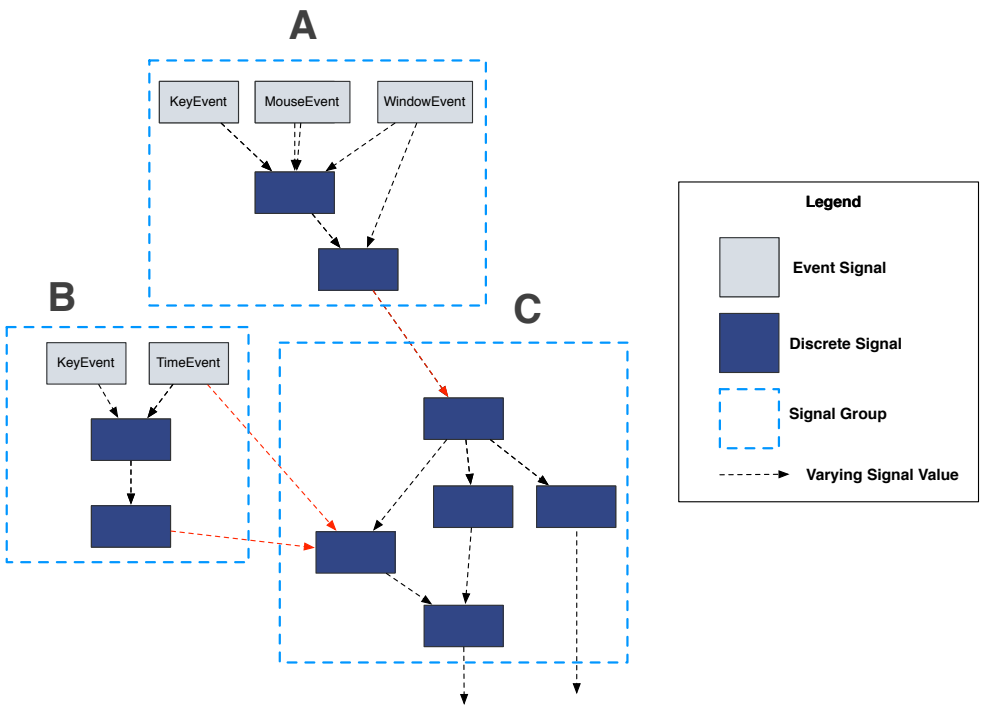


Figure 3.2: An example of a dependency graph that is separated into signal groups. Tesel contains combinators for connecting signal groups (indicated by the red lines). The implementation identifies groups that can be ran in parallel. Thus, since A and B are independent of each other then they can be ran in parallel on an event occurrence.

The embedding of Tesel inside the object-oriented language, Swift, allows signal groups to benefit from the programmability benefits provided by objects. Signal groups are defined by implementing class definitions that inherit from the signal group base class. Thus, signal groups are objects and with that comes the major benefits of object-oriented programming such as data-encapsulation, and modularity. Data-encapsulation hides the details of the object's internal implementation from the outside world and forces outside objects to retrieve information about the object through its properties and methods. This property makes it easy to change the internal implementation of the object without changing the interface seen by any outside object. For example, Figure 3.3 illustrates a high-level overview of the construction a signal group dependency graph for an application that performs image convolution on large-sized images. If the image size is too small then the overhead of performing the convolution in parallel hinders performance. We discuss

this potential issue in further details in Chapter 6. A programmer defines an `ImageWorker` group that performs the convolution effect for a section of the image. Once implemented, multiple instances are defined and assigned sections of the image. On an event occurrence that causes the image to be modified, the system identifies that these are independent groups and runs each worker group in parallel. After each worker completes, the `Viewer` signal group is evaluated and can use the updated image in some way (*e.g.*, displaying the filtered image in a window). Additionally, if another programmer at a future point in time wants to use a better image processing algorithm but maintains the API used by outside classes then the programmer only has to refactor the `ImageWorker` class and not other portions of the application.

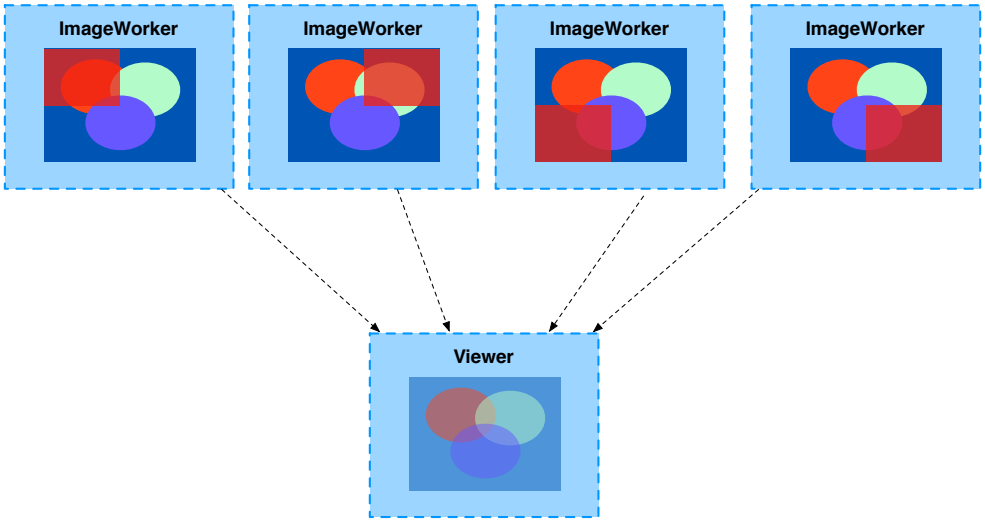


Figure 3.3: A high-level overview of an image convolution application. We omit the internal signal configuration for simplicity purposes. A programmer can define an ImageWorker signal group that performs a filtering effect on a predetermined section of an image. Multiple instances of the signal group can be defined, which will allow the FRP system to run these groups in parallel to improve performance. Once each group is completed, the Viewer signal group can push the completed image to the display.

3.1.2 Time in Tesel

Time is continuous and non-negative real number in Tesel. Discrete signals and internal event signals are updated at discrete points in time (*i.e.*, when event occurs from the outside environment).

Continuous signals are updated over a series of discrete points in time known as *sample periods*. For each sample period, every continuous signals within a signal group is sampled (*i.e.*, updated). We define *local-time* properties that represent the time when the signal group was last-sampled, the current sampling time, and the delta-time (*i.e.*, the current sampling time minus the last sampling time):

```
typealias Time = Double
var lastTime    : Time
var deltaTime   : Time
var currentTime : Time
```

Every signal contains a reference to its parent signal group. Thus, continuous signals that use time as parameter when updating their value have access to these time properties. For example, Tesel provides an integral primitive function that produces a continuous signal that integrates its value over time using the delta-time property of its signal group.

A few primitive functions require time values as parameters to their functions. We remove the need for the programmer to understand the system's units of time by providing a time utilities class:

```
public struct TimeUtils {
    public static let millisecond : Time
    public static let second : Time
    public static let minute : Time
    public static let hour : Time
}
```

A programmer can produce a `Time` value easily by using the `TimeUtils` properties:

```
let halfSecond =  500 * TimeUtils.millisecond
```

3.2 Core Construction Primitives

The core construction primitives define discrete and continuous signals in Tesel. These combinators allow for building complex signal networks within signal groups. At signal group initialization, these combinators are used to initialize the internal signal network. Many of the lift functions are overloaded to work with the various signal types in the language. A single example is given for each combinator but the semantics is the same for each primitive regardless of the type of signal it produces.

3.2.1 Lifted Functions

The following primitives represent a collection of primitives that lift pure functions to the level of signals.

- **constant**: As its name implies, the constant function:

```
func constant<O>(initVal : O) -> CSignal<O>
func constant<O>(initVal : O) -> DSignal<O>
```

  creates a signal whose value never changes. For example, the acceleration due to Gravity from the falling ball example never change with respect to time. We can represent this constant property as a constant signal:

```
let gravity : CSignal<Float> = constant(9.81)
```

- **lift**: Lifting of signals into a different signal is done by the lift combinator:

```
func lift<I1,O> (fn : I1 -> O, in1 : DSignal<I1>) -> DSignal<O>
func lift<I1,O> (fn : I1 -> O, in1 : CSignal<I1>) -> CSignal<O>
func lift<I1,O> (fn : I1 -> O, in1 : ESignal<I1>) -> ESignal<O>
```

  A signal created with the lift combinator provides the given pure function (`fn`) with an input

signal (`in1`), which the function can then transform the input value into the value of the signal. For example, graphics applications need to know when the user changes the window size in order to reshape the entities in the application. This transformation is usually done by using the window dimensions to create a projection matrix, which is used to reshape the graphics entities. We can easily accomplish this by using a signal created by the lift function to change the updated window dimensions into a new projection matrix:

```
1  let projectionSignal : ESignal<WinEvent> = lift(reshape, in1:TeselExternal.windowSize)
2
3  func reshape(winSize : WinEvent) ->  float4x4 {
4      let projMat : float4x4 = Transform.ortho(left:0,
5                                    right:Float(winSize.width),
6                                    bottom:Float(winSize.height),
7                                    top:0,
8                                    nearZ:-1.0,
9                                    farZ: 1.0)
10     return projMat
11 }
```

Line 1 creates a projection signal using the lift function. The `TeselExternal` class defines all the external signals (*e.g.*, key events, window size changes, and mouse events) that connects the outside environment to the program. The signal will use the window event signal as an input to the `reshape` function (Line 3–11) to convert the window size into a projection matrix. Lines 4–9 create a new orthographic projection matrix using the window size. The matrix is returned as the new value for the signal on Line 10.

- **merge**: The lift functions contain overload versions that allow for up to five parameters of signals to be lift into another signal with the exception of event signals. Event signals only contain values at discrete points in time. Thus, there can be ambiguity when two or more event signals occur simultaneously or when one occurs and the others don't. The merge function fixes this problem by combining multiple event signals into a single signal:

```
func merge<O> (signals : [ESignal<O>]) -> ESignal<O>
```

If one of the input signals changes then the value of the merge signal becomes the value of the changed input signal. The combinator has a left-biased signal priority to handle the situation when more than one input signals occur simultaneously. The left-most signal input "wins" and its value becomes the value of the merge signal. Below is an example of determining the offset for a moving camera by merging the arrows keys and then producing the offset using the lift function. The full implementation of creating a moving camera within a signal group is discussed in section 3.3.

```
1  let front  = float3(0,0,-1)
2  let up     = float3(0,1,0))
3
4  let inputMerged = merge([TeselExternal.leftArrow,
5                           TeselExternal.rightArrow,
6                           TeselExternal.upArrow,
7                           TeselExternal.downArrow])
8
9  let offset : ESignal<KeyEvent> = lift(computeOffset, in1:inputMerged)
10
11 func computeOffset(arrowKey : KeyEvent) -> float3 {
12         let speed : Float = 0.5
13         var offset : float3!
14         switch(arrowKey.key)
15         {
16         case .W:
17             offset = (speed * front)
18         case .S:
19             offset = -(speed * front)
20         case .A:
21             offset = -normalize(cross(front, up)) * speed
22         case .D:
23             offset = normalize(cross(front, up)) * speed
24         default:
25             offset = float3(0.0)
26         }
```

```
27          return offset
28  }
```

Lines 1–2 define constants that represent the front direction of the camera, and its up direction. We define a signal for merging the incoming arrow keys signals (Lines 4–7), and an offset signal that is updated if the merged signal is updated owing to an event occurrence by one of its event signals (Line 9). The `computeOffset` function computes the offset value based on which arrow key was clicked. Line 12 defines a constant speed for the camera movement, while Line 13 defines an offset variable that holds the amount to move the camera by. Lines 14-26 determine the offset amount based on the arrow key input, which is then returned on Line 27.

3.2.2 History-Sensitive functions

The history-sensitive functions allow for signals to accumulate and transform their values from the past.

- **foldp**: Updating a signal value based on its previous value is important in many situations. In Tesel, we provide a *history-sensitive* combinator:

```
func foldp<I1,O> (initVal : O, fn : (O,I1) -> O, in1 : DSignal<I1>) -> DSignal<O>
func foldp<I1,O> (initVal : O, fn : (O,I1) -> O, in1 : ESignal<I1>) -> DSignal<O>
```

  The foldp combinators stands for "*fold from the past*" and is similar to the foldp combinator provided by Elm [23]. Similar to the higher-order fold functions (*e.g.*, foldl and foldr), the foldp combinators allows a signal value to accumulate over time. The combinator defines an accumulator, which is initially given a value of the `initVal` parameter. As the signal receives new values from the input signal (`in1`), it combines them with its accumulator using the provided update function (`fn`). The signal value is then updated to the value of the accumulator. For example, remembering the sequence of the last mouse presses:

```
1  let mouseHistSig = foldp("", fn:concatMouseLoc, in1:TeselExternal.mouse)
2
3  func concatMouseLoc (accum : String, mouseEvent : MouseEvent) -> String {
4      let mouseLoc = mouseEvent.location
5      let newAccum = "\(accum), \(mouseLoc))"
6      return newAccum
7  }
```

Line 1 defines a mouse history signal, which is given an empty string as its initial value. As the user clicks on the mouse, the signal will update its value by calling the `concatMouseLoc` function(Lines 3–6). The function takes in the previous value of the signal (*i.e.*, the old mouse locations) and the mouse event, which holds the new mouse location. Line 4 retrieves the new location of the mouse and Line 5 uses string interpolation to create a new string with the old locations and new location. Finally, Line 6 returns the newly updated mouse history.

- **filter**: Stopping event occurrences is done with the filter function:

```
func filter<O> (fn : (O -> Bool), in1 : ESignal<O>) -> ESignal<O>
```

The given function (`fn`) decides whether the event occurrence from (`in1`) should continue (*i.e.*, the function returning true) or end (*i.e.*, the function returning false). The following example only changes the filter signal value on even key presses:

```
1  let inputMerged : ESignal<KeyEvent> = merge ([TeselExternal.one,
2                          TeselExternal.two, TeselExternal.three,
3                          TeselExternal.four, TeselExternal.five,
4                          TeselExternal.six, TeselExternal.seven,
5                          TeselExternal.eight, TeselExternal.nine])
6
7  let evenSignal : ESignal<KeyEvent> = filter(isEven, in1:inputMerged)
8
9  func isEven(key : KeyEvent) -> Bool {
10     var filterThrough : Bool!
11     switch(key.key)
```

```
12      {
13      case .Two, .Four, .Six, .Eight:
14          filterThrough = true
15      default:
16          filterThrough = false
17      }
18      return filterThrough
19  }
```

Lines 1–5 define a signal that merges all the number key events. Line 7 creates a signal that uses the `isEven` function (Lines 9-19) to filter out only the even number key events. The `isEven` determines whether the number key pressed was even (Lines 11-17), and returns the result on Line 18. In this example, the signal will only cause event occurrences for the values: `KeyEvent(key:.Two)`, `KeyEvent(key:.Four)`, `KeyEvent(key:.Six)`, or `KeyEvent(key:.Eight)`.

- **integral**: The integral combinator integrates the value of a signal over time:

```
func integral <O : IntegralType>(val : CSignal<O>) -> CSignal<O>
```

The type of the integrating value needs to conform to `IntegralType` protocol, which are numerical types that can be mathematically integrated (*e.g.* `Float`, `Double`, `float2`, *etc.*). As stated earlier, the integral function is useful for developing programs that use systems of ordinary differential equations. In particular, we can recreate the falling ball example from before. First, we define two constants that represent the initial position and velocity:

```
let y0 : Float = 10.0
let v0 : Float = 0.0
```

Next, we create the the signals for determining the velocity of the ball by integrating the gravitational constant with respect to time and adding the initial velocity ($v = v_0 + \int -9.81 \text{ dt}$):

```
let gravity : CSignal<Float> =  constant(-9.81)
let acceleration : CSignal<Float> = integral(gravity)
let v : CSignal<Float> = lift(computeV, in1:acceleration)

func computeV(acceleration: Float) ->  Float {
    return v0 + acceleration.value
}
```

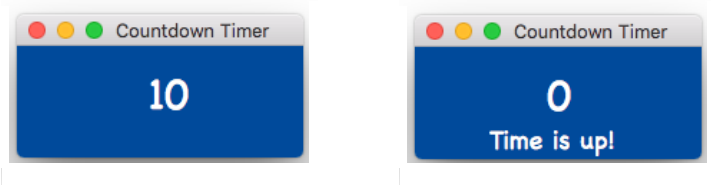
Finally, the position signal is created by integrating the velocity value and adding its initial position ($y = y_0 + \int v \, \mathrm{dt}$):

```
let vIntegral  : CSignal<Float> = integral(v)
let y : CSignal<Float> = lift(computeY, in1:vIntegral)

func computeY(vel: Float) -> Float {
    return  y0 + vel.value
}
```

- **after**: The after combinator produces an event after a certain amount of time has passed:

```
func after(time : Time, recurring : Bool = false) -> ESignal<UnitValue>
```

The function takes in a parameter that represents how much time needs to pass before sending a `UnitValue`. Swift does not contain a unit type that is accessible to the programmer; therefore, we provide one for signals that do not produce a value. The second parameter to the after function allows for after signals to be recurring indefinitely. By default the after signals do not reoccur after the first event. Figure 6.5 shows an example of implementing a countdown timer in Tesel.



(a) Start (b) End

Figure 3.4: A Tesel implementation of a timer countdown

The code below shows how the after combinator is used to update the text seen in Figure 6.5:

```
1  let timerStart = 10
2  let timeEventSig = after(TimeUtils.second, recurring:true)
3  let time = foldp((timerStart,""),fn:updateTime, in1:timeEventSig)
4
5  func updateTime(time : (Int,String), _ : UnitValue ) -> (Int,String) {
6      let seconds = time.0
7
8      if seconds - 1 == 0 || seconds == 0 {
9          return (0,"0\nTime is up")
10     }else {
11         return (seconds - 1, "\(seconds - 1)\n")
12     }
13 }
```

Line 1 defines a start time for the timer, while Line 2 creates an after signal that fires repeatedly every second. To keep track of the seconds that have passed after each firing, a foldp signal is created (Line 3) that uses the `updateTime` function (Lines 5–13) to decrement its value each time `timeEventSig` fires. The `updateTime` function retrieves the current time (Line 6) and then determines whether the time is up (Lines 8–11). If time has not reached zero then it decrements the number of seconds (Line 11) ; otherwise, the function stops decrementing the time and states to the user that the time is up (Line 9).

3.2.3 Additional Combinators

The list of combinators before represent additional combinators that are beneficial when working with the core combinators.

- **hold**: Takes in an initial value and updates its value to the value of the input event signal on an event occurrence:

```
func hold<O>(initVal: O, in1:ESignal<O>) -> DSignal<O>
```

- **sampleWithC**: Merges an event signal with a continuous signal. An event occurrence from the input signal (`in2`) causes the update function to run with the current value of the continuous signal. The function produces an output event occurrence based on its return value.

```
func sampleWithC<I1,I2,O>(fn : (I1,I2) -> O, in1:CSignal<I1>, in2:ESignal<I2>)
                          -> ESignal<O>
```

- **when**: Invokes the given function with the continuous signal's value. An event occurrence happens whenever the function changes from false to true.

```
func when <O>(fn: O -> Bool, in1:CSignal<O>) -> ESignal<O>
```

- **join**: Gathers all the values within a collection of discrete signals together. Whenever a signal within the collection changes then the join signal updates its collection of values.

```
func join<O> (signals : [DSignal<O>]) -> DSignal<TeselCollection<O>>
```

3.2.4 Dynamic Collections

Tesel provides a the `collect` combinator that is used to gather information about signals that collectively model a system of common entities. The collect combinator produces a continuous signal

that represents a *dynamic collection*, which allows for elements to be added and removed from the collection. This combinator is similar to the dynamic collections combinators provided in Yampa [21] but dynamic collections in Tesel can update the elements in parallel. Thus, this combinator is useful if the collection elements that are performing complex tasks. For example, Figure 3.5 is a Tesel program that models a particle system. Each particle contains a world position, an energy property, a color that is either red or orange, and counters that represent the number of neighboring particles. Using the collect combinator, the particles are grouped together into a "system" where information can be gathered about the system such as the number of particles, the mean energy, and the number of particles of each color.

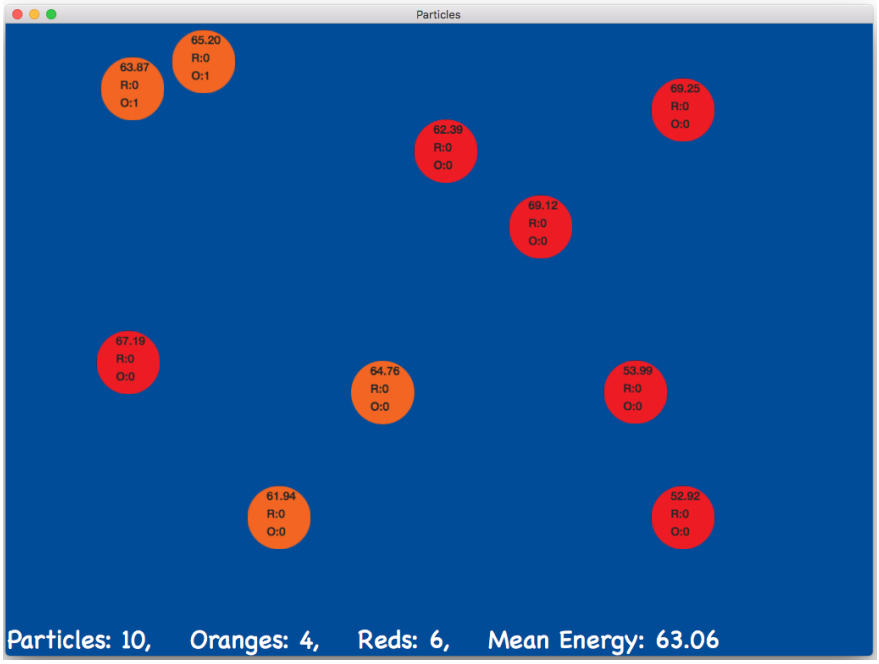


Figure 3.5: A Tesel program that illustrates an example of implementing a 2D particle system. Each particle has a position in the world, a color, an energy value, and neighbors.

The `collect` combinator can also create a static collection along with a dynamic collection:

```
// Static collection
func collect<O> (initVal : [O],
                 efn:(O, [O])-> O)
                 -> CSignal<TeselCollection<O>>

// Dynamic collection
func collect<I1, O> (initVal : [O],
                     efn:(O, [O])-> O, upfn:([O],I1)->[O], in1:ESignal<I1>)
                     -> CSignal<TeselCollection<O>>
```

The static `collect` function takes in an array of values that makes up the static collection along with element update function, which updates the elements within a collection signal. Each element is provided with its current value and the current values of the other elements in the collection. The dynamic `collect` function takes in same parameters as the static version but with the addition of a collection update function (`upfn`) and an input event signal. On an occurrence from the event signal, the collection updates by invoking the update function. Both these functions produce an signal that contains the collection within a `TeselCollection` class. Currently, Swift does not allow the `Array` struct to conform to the `Equatable` protocol, which is required by the implementation. Thus, we provide our own wrapper class that implements the `Equatable` protocol. We discuss why this is important in Chapter 6. The following code snippets illustrate implementing a particle system using the collect combinator. First, we define an enumeration to represent a particle color followed by a struct to define a particle with properties for representing an energy value, position, color and neighbor counts:

```
enum ParticleColor : Int {
    case Orange, Red
}
struct Particle  {
    static let radius : Float = 12.5
    let energy       : Float
    let pos          : float2
    let color        : ParticleColor
    let neighbors : (red: Int, orange: Int)
```

```
    init(energy : Float, pos: float2, color: ParticleColor,
         neighbors:(Int,Int)) {
        self.energy = energy
        self.pos = pos
        self.color = color
        self.neighbors = neighbors
    }
}
```

The code below shows how to creates program seen in Figure 3.5. Lines(1–6) define a static array that represents the particle positions for the system. Lines 9–15 create and append particles to an array that represents the initial elements for the collection. Each particle is initialized to a position from the positions array, a random color, and an initial energy value of zero. The dynamic collection of particles is defined on Lines 18–21. The collection uses the `updateParticle` function to update the particles in the collection and on a mouse occurrence the function invokes the `updateSystem` function. Information is gathered about the particle system by creating the `systemInfo` signal to map over the system (Line 24).

The update function of a particle (`updateParticle`) determines the number of neighboring particles defined within a query radius. Lines 29–43 determine whether a particle in the collection is a neighbor. For each particle within the collection, we determine the distance between a potential neighbor and the querying particle. If the potential neighbor is not itself (*i.e.*, `d > 0`) and is defined within the query radius then we determine its color and update the appropriate color counter. In this example, the particles remain in at a static location. As the user clicks on the screen to add more particles, the neighbor counts eventually update. Lines 46-49 use the counters to define and return a new particle.

```
1  //particle positions
2  let positions = [float2(55.0, 77.0), float2(234.0, 45.0),
3                   float2(800,102), float2(632,240),
4                   float2(145,400), float2(800,583),
5                   float2(520.0, 150.6), float2(744,435),
```

```
                  float2(445.4,435.4), float2(323,583)]

//Creates the initial collection of Particles
var particles : [Particle] = []
for index in 0..<positions.count {
    let pos   = positions[index]
    let color = ParticleColor(rawValue: Random.generate())!
    let particle = Particle(energy:0.0, pos:pos,color:color)
    particles.append(particle)
}

//Creates the dynamic collection
let system = collect(particles,
                     efn:updateParticle,
                     upfn:updateSystem,
                     in1:TeselExternal.mouse)

//Retrieves information about the system
let systemInfo = map(computeInfo, in1system)

func updateParticle(particle : Particle,
                     particles : [Particle) -> Particle {

    let queryRadius : Float = 100.0
    var neighborCountRed = 0
    var neighborCountOrange = 0

    // Determine the number of neighbors
    for neighbor in particles {
        let d = distance(particle.pos, neighbor.pos)
        if d > 0 && d <= queryRadius {
            switch(neighbor.color)
            {
            case .Red:
                neighborCountRed += 1
            case .Orange:
                neighborCountOrange += 1
            }
        }
    }
```

```
46      return Particle(energy:particle.energy,
47                      pos: particle.pos,
48                      color:particle.color,
49                      neighbors:(neighborCountRed,neighborCountOrange))
50  }
```

The `updateSystem` function initializes an empty array that will hold the new particles (Line 5). Next, the we determine what type of mouse event action was performed. A mouse event has three potential actions: `.Pressed` (the mouse button was clicked), `.Dragging` (the mouse button is held down and the user is moving the mouse), and `.Release` (the mouse button was released). Updating the system only occurs when the mouse button is released and fired (Line 7) ; otherwise, the prior particles in the collection are the new particles (Line 26). Once the mouse is released, we begin to update the system. A particle is added to the system if the user did not click on a prior particle; therefore, we define a boolean variable to determine if the mouse click was on a particle (Line 10). Lines 11–18 iterate through the system to see if the user clicked on a particle by performing a inside circle test that uses the particle radius, mouse location, and the particle position. If the function returns false then the click was not on that particle, and the particle is added to the new particles array with an updated energy value (Line 10). Otherwise, we remove the current particle by not adding it to the new particles array and setting the flag variable to false since the click was on a particle (Lines 16). Potentially particles could overlap so we continue processing the rest of the particles (Line 17). If no particles were removed after iterating through the system then a new particle is added to the system with a random color and at the mouse location (Lines 21-23). Finally, the `newParticles` array is returned as the new collection value for the signal (Line 29). The `computeInfo` function (Lines 31–52) iterates through the particles to compute the mean energy and count the number of red and orange particles. This information is converted into a string, which can then be displayed on the window.

```
1  func isInsideCircle (radius : Float, pt1 : float2, pt2: float2) -> Bool {
2      return distance(pt1, pt2) < radius
```

```
3  }
4  func updateSystem(particles:[Particle], mouse: MouseEvent) -> [Particle] {
5      var newParticles : [Particle] = []
6
7      if mouse.action == .Release {
8          let y = Float(windowSize.height - mouse.location.y)
9          let actualPos = float2(Float(mouse.location.x),y)
10         var shouldAddParticle = true
11         for var particle in particles {
12             if !isInsideCircle(37.5, pt1:actualPos, pt2:particle.pos){
13                 particle.energy = particle.energy + Random.within(1.0...10.0)
14                 newParticles.append(particle)
15             }else {
16                 shouldAddParticle = false
17                 continue;
18             }
19         }
20         if shouldAddParticle {
21             let color = ParticleColor(rawValue: Random.generate())!
22             let particle = Particle(energy:0.0, pos:actualPos,color:color)
23             newParticles.append(particle)
24         }
25     }else {
26         newParticles = particles
27     }
28
29     return newParticles
30 }
31 func computeInfo(collection : TeselCollection<Particle>) -> ParticleSystemInfo {
32     let particles = collection.rawArray
33     var sum      : Float = 0.0
34     var nOranges : Int = 0
35     var nReds    : Int = 0
36
37     for particle in particles {
38         sum += particle.energy
39         switch(particle.color) {
40         case .Red:
41             nReds += 1
42         case .Orange:
```

```
43              nOranges += 1
44          }
45      }
46      let mean = sum / Float(particles.count)
47
48      return  ParticleSystemInfo(meanEnergy:mean,
49                                 numOfParticles:particles.count,
50                                 numOfOranges:nOranges,
51                                 numOfReds:nReds)
52  }
```

3.3 Constructing and Connecting Signal Groups

Creating a signal group is done by implementing a class that inherits from the base case, `SignalGroup`. All primitive functions discussed in section 3.2 are defined as methods of the `SignalGroup` class. The local dependency graph is constructed within the `init` methods of each group. The static graph is constructed *internally* by the signal group when calling the primitive functions. Passing the signals as inputs to other primitive functions creates the dependency connections between signals. For example, a full implementation of defining a moving camera is defined as follows.

```
1   class Camera : SignalGroup {
2
3       private let _front : float3
4       private let _up    : float3
5       private let _speed : Float
6
7       var matrix         : DSignal<float4x4>!
8
9       init(initPos : float3, front: float3, up : float3, speed: Float = 0.5) {
10          self._front = front
11          self._up = up
12          self._speed = speed
13          super.init()
14          let inputMerged = merge([TeselExternal.a,TeselExternal.d,
```

```
15                                  TeselExternal.w,TeselExternal.s])
16          let offset = lift(computeOffset, in1:inputMerged)
17          let position = foldp(initPos, fn: {pos, offset in pos + offset},
18                               in1:offset)
19          self.matrix = lift(makeMatrix, in1:position)
20      }
21      func computeOffset(arrowKey : KeyEvent ) -> float3 {
22          var offset : float3!
23          switch(arrowKey.key)
24          {
25          case .W:
26              offset = (_speed * _front)
27          case .S:
28              offset = -(_speed * _front)
29          case .A:
30              offset = -normalize(cross(_front, _up)) * _speed
31          case .D:
32              offset = normalize(cross(_front, _up)) * _speed
33          default:
34              offset = float3(0.0)
35          }
36          return offset
37      }
38      func makeMatrix(position : float3) ->  float4x4 {
39          return Transform.lookAt(position, dir: position + _front, up: _up)
40      }
41  }
```

Line 1 defines a `Camera` class that inherits from the `SignalGroup` class. Lines 3–5 define constant properties for the camera's front direction, up direction, and speed. Alternatively, these constants could also be passed as constant signals to the various primitive functions. The property defined on Line 7 represents a signal that will hold the camera's view matrix. Inside the init function (Lines 9-20), we define the static dependency graph for the signal group. First, the `init` function takes in four parameters to initialize the constant properties (Lines 12–13). The `initPos` parameter is used to initialize the position signal (Line 17). Swift requires that all properties of the subclass and super

class are initialized (Line 13) before using any method of a class as parameter to another function. Lines 14–19 define the inner signal graph, which uses methods that are part of the `Camera` (*e.g.*, `computerOffset`). Lines 14–16 is the same code from section 3.2.1 to determine the offset for the camera. Line 17 defines a history-sensitive signal that contains the initial position and is updated on each offset occurrence. Each time the `position` signal updates the `matrix` signal is updated by lifting the `makeMatrix` function (Lines 38–40), which produces the updated view matrix.

The `Camera` signal group is connected to other parts of the entire signal group for the program by using the `group` function:

```
func group <O : SignalGroup, I1> (group : O, in1:DSignal<I1>) -> DSignal<I1>
func group <O : SignalGroup, I1> (group : O, in1:ESignal<I1>) -> ESignal<I1>
func group <O : SignalGroup, I1> (group : O, in1:CSignal<I1>) -> CSignal<I1>
```

The combinator takes in an instance of a signal group and any type of signal defined *within* the signal group. Each time the input signal is updated, the returned signal's value is updated to the input signal's value. The group primitive links two signal groups together. For example, the code below connects the output of camera signal group (*i.e.*, the view matrix) to a view signal group that uses the matrix for rendering a scene.

```
1  struct Viewer : SignalGroup {
2
3      var frame     : DSignal<Frame>!
4
5      init() {
6          //Create a camera group
7          let camera = Camera(initPos:float3(0.0,0.0,3.0),
8                              front:float3(0,0,-1),
9                      up:float3(0,1,0))
10         let camMatrix = group(camera, in1:camera.matrix)
11
12         self.frame =  render(self.flatFrame, in1:camMatrix)
13     }
14     func flatFrame(camMat: float4x4) -> Frame {
15          //Make the render frame
```

```
16            // ...
17        }
```

The `Viewer` class is defined as a signal group and contains a property that represents a signal containing the view's frame (Lines 1–3). Lines 7–10 define an instance of the `Camera` signal group and uses the `group` function to connect its output to the `camMatrix` signal of the `Viewer` signal group. The `Viewer` group uses the camera matrix for creating a frame for the viewer (Lines 12–17). We discuss the rendering constructs in further detail in section 4.

3.3.1 Dynamic Signal Groups

A dynamic signal group allows for reseting and reconfiguring the initial signal network within a signal group. Implementing a dynamic signal group is done by inherting from the `DSignalGroup` class. This class defines additional properties and methods for resetting and sampling the internal state.

```
1  public class DSignalGroup<O> : SignalGroup {
2          public var sampleSignal : CSignal<O>?
3          public var switchSignal : SSignal<DSignalGroup<O>>?
4          func switchc<E1, T: DSignalGroup<O>>(fn:E1->(T,Bool),
5                                               eIn:ESignal<E1>)
6                                               -> SSignal<T> {...}
7   }
```

A dynamic signal group is a generic class where its type parameter represents the value that can be sampled outside of the class. Thus, the `sampleSignal` property holds the value that represents the output of the dynamic signal group. The internal state can be reset by assigning the `switchSignal` a signal that is produced by the `switchc` function. The `SSignal` is a special signal that is used for switching out the state for a dynamic signal group. The value the switch signal holds represents the new internal state at the time of the switching. The switching behavior is defined using the `switchc` function. The function takes in the event that causes the switch (`eln`) and a function that produces a

tuple that contains a new dynamic signal group and a boolean that represents if the dynamic signal group will switch into a constant state for future update steps.

The dynamic signal group implementation of the falling ball example is as follows.

```
1  public struct Ball : Equatable {
2      let yPos : Float
3      let yVel : Float
4      static let radius : Float =  2.0
5  }
6
7  class BouncingBall : DSignalGroup<Ball> {
8
9      init(initBall : Ball) {
10         super.init()
11
12         // Calculate the acceleration
13         let acceleration : CSignal<Float> = integral(constant(-9.81))
14
15         // Calculate the velocity
16         let vY  = lift({(v0:Float, accel:Float) in  v0 + accel} ,
17                        in1:constant(initBall.yVel), in2:acceleration)
18
19         // Integrate the velocity
20         let velIntegral = integral(vY)
21
22         // Calcualte the new position of the ball
23         let ball  = lift({(p0:Float, vel:Float) in
24                                      Ball(yPos:p0 + vel, yVel:vel)},
25                           in1:constant(initBall.yPos),
26                           in2:velIntegral)
27
28         // Determine if the ball hit the ground
29         let detectedBounce = when({(ball : Ball) in ball.yPos <= 0 },
30                                   in1:ball)
31
32         // The detected bounce will cause the group tto switch based
33         // on calling the flipBall function
34         self.switchSignal = switchc(flipBall, eIn:detectedBounce)
35
```

```
36          // The Ball will be the sampled object for the dynamic group
37          self.sampleSignal = ball
38
39      }
40      func flipBall(ball : Ball) -> (BouncingBall,Bool) {
41          if abs(ball.yVel) < 1 {
42              return (BouncingBall(initBall:ball), true)
43          } else {
44              let ball =  Ball(yPos:0, yVel:-ball.yVel)
45              let bouncingBall =  BouncingBall(initBall:newBall)
46              return (bouncingBall,false)
47          }
48      }
49  }
```

The output of the dynamic group is a `Ball` object. A `Ball` contains a velocity, an initial position (Lines 1–5). Lines 13–26 is similar to the code described in the integral section. We calculate the acceleration, integrate the velocity, and generate a new Ball with an updated position (Lines 23–26). Line 29 defines an event signal that determines whether the ball has hit the ground. We use this event signal on Line 34 to determine whether switch the group into a new state based on an occurrence from `detectedBound`. The switching of the group states is determined by the `flipBall` function. Line 37 assigns the ball signal to be equal to the sampled signal. Thus, anytime the signal group is sampled from outside the group it will be the value of the ball signal.

The `flipBall` function (Lines 40–47) first checks how fast the ball is moving. If the ball is still moving then we create a new `BouncingBall` instance with the ball's velocity inverted (Lines 44-45). The function returns the new `BouncingBall` instance and the value false, which represents the dynamic group has not reached a constant state. Otherwise, if the ball is barely moving then we return a new `BouncingBall` instance with the ball current state and the value true, which states that the dynamic group has reached a constant state (Line 42). After the group updates to the new instance value, then it will no longer be sampled.

We can connect a dynamic group to another signal group using the `group` function:

```
func group <O : Equatable, I2> (group : DSignalGroup<O>, in1:ESignal<I2>) -> ESignal<O>
```

This overloaded version of the `group` function takes in the dynamic signal group and an event signal. When the input event signal has an occurrence then the dynamic signal group will be sampled (*i.e.*, its sampled signal will be sampled) to produce an output event occurrence. For example, We update the `Viewer` class from the camera example by connecting the bouncing ball signal group. We define an instance of a `Ball` on Line 8 and a signal that is defined by the group function to retrieve the ball on every render event (Lines 9–10). Line 11 creates a discrete signal with the initial ball configuration and the bouncing ball event. Line 20 passes the ball to the frame signal for rendering.

```
1  struct Viewer : SignalGroup {
2
3      var frame     : DSignal<Frame>!
4
5      init() {
6
7          // Create the bouncing ball group
8          let initialBall = Ball(yPos:10.0, yVel:0)
9          let bouncingBall = group(BouncingBall(initBall:initialBall),
10                                   in1:TeselExternal.render)
11         let ball = hold(initialBall, in1:bouncingBall)
12
13         //Create a camera group
14         let camera = Camera(initPos:float3(0.0,0.0,3.0),
15                             front:float3(0,0,-1),
16                     up:float3(0,1,0))
17
18         let camMatrix = group(camera, in1:camera.matrix)
19
20         self.frame =  render(self.flatFrame, in1:camMatrix, in2:ball)
21     }
22     func flatFrame(camMat: float4x4, ball: Ball) -> Frame {
23          //Make the render frame
24         // ...
```

```
25      }
```

3.4 Interacting with the External Environment

Most FRP languages provide a *input-process-output* loop to connect the FRP system to the outside environment. The loop pushes external inputs from the outside environment as inputs into the FRP system to produce an output. Figure 3.6 shows a slightly simplified version of the input-process-output loop for signal functions in the Yampa language.

```
reactimate :: IO a            -- init
           -> IO (DTime, Maybe a)  -- input/sense
           -> (b -> IO Bool)        -- output/actuate
           -> SF a b                       -- process/signal function
           -> IO ()
```

Figure 3.6: The reactimate combinator in Yampa

The first argument to reactimate is the initial action (*e.g.*, printing a welcome message) that eventually returns the first sample input for the main signal function (*i.e.*, the third argument of reactimate). The second argument (sense) is an IO action that should return that contains the time passed since the last sample period (*i.e.*, "delta" time, `DTime`) and a new sample input for the main signal function. Wrapping the sample input allows it to either provide a new sample input (`Just a`) or reuse the prior sample input (`Nothing a`). The third argument is a function that uses the output from the main signal function to produce an IO action that will process the output value in some way. The IO action returns true if the processing loop should stop. Otherwise, it continues sampling the signal function. The last argument is the main signal function that is continuously sampled. The reactimate function approximates the continuous-time model by sampling the main signal function at discrete points in time by invoking the sense action at the start and the actuate at the end of each sampling period. For example, Figure 3.7 shows how the graphics render loop is handled using the reactimate function. The init action handles setting up the windowing sys-

tem and the graphics context. The sense and actuate functions handle reading in events from the window system and using the graphics API and context to render an image on the screen. The reactimate function and the various other input-proccess-loops all suffer from the same problem: the programmer is still required to write the tedious code that handles external events and the low-level code needed to render the signal function's output to the screen. Many of the FRP languages focused in the domain of computer graphics [30, 31, 20]. did provide higher-level constructs for rendering objects or hook-ins to their reactive system to handle the windowing system. But these languages require third-party libraries to be linked into the program, which resulted in many of the libraries not being updated or using older graphics pipelines (*e.g.*, the fixed function pipeline). Ultimately, programmers are forced to use to rendering loop paradigm if they want to write custom rendering algorithms via shader programs.

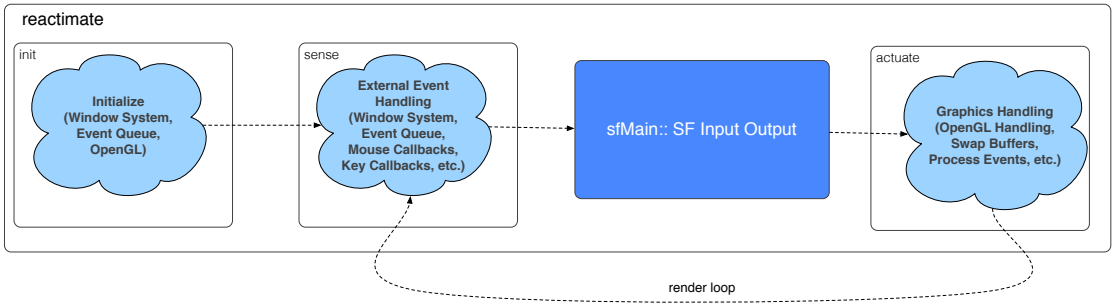


Figure 3.7: An example of how to implement the render-loop within the reactimate combinator.

As shown in Figure 3.8, Tesel alleviates this problem by having the runtime handle all external events and the low-level details of rendering objects. The programmer uses the Tesel framework, which includes all the FRP language features and rendering constructs, to define the main signal group that will be continuously sampled. Embedded within the signal group will be code that forces the runtime to render frames of objects or to quickly retrieve signals tracking external events. Tesel completely removes the explicit render-loop and instead allows the programmer to decide when to send frames to the runtime for rendering.

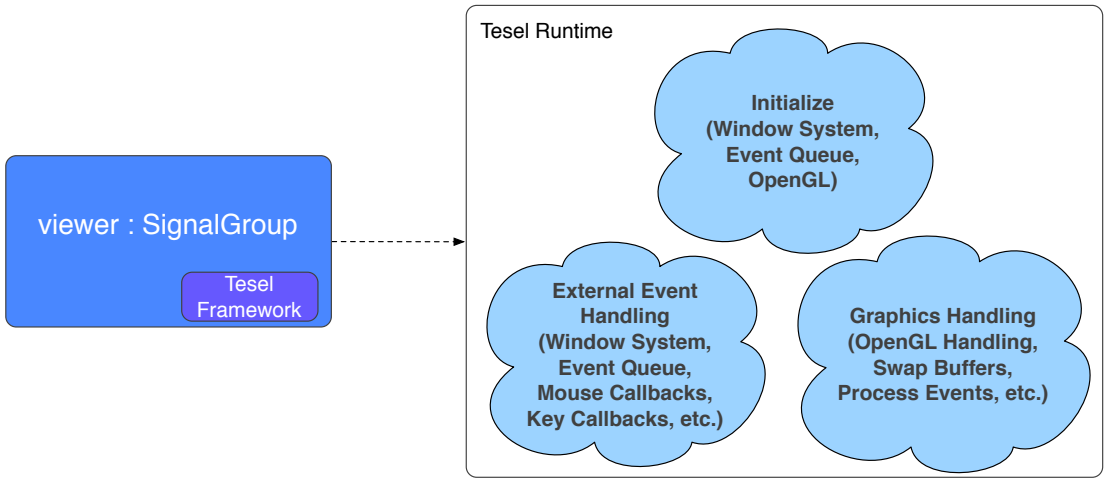


Figure 3.8: The Tesel runtime handles all interactions between the outside environment, including the GPU and the root signal group. This property frees the programmer from writing additional code needed to handle these various systems.

The only way a Tesel program interacts with the outside environment other than receiving updates from event signals is through defining a main function. Tesel programs are required to a have global function named *teselMain*, which is the designated start of the program:

```
func teselMain<O : SignalGroup> () -> (TeselConfiguration, 0)
```

The Tesel main function takes in no parameters and is required to return a `TeselConfiguration`, which provides initialization information for the window, and the root signal group, which is the group that is continuously updated by the runtime. For example, the main function for the falling ball is defined as follows:

```
1  func teselMain() -> (TeselConfiguration, Viewer) {
2
3      //The initialize size of the window.
4      let initWinSize = CGSize(width:1024, height:768)
5
6      //The signal group that contains the bouncing ball
7      let rootGroup : Viewer = Viewer(initWinSize: initWinSize)
8
9      //Configuration object for setting up the window
10     let config = TeselConfiguration(windowSize: initWinSize, windowTitle:"Falling Ball")
11
12     return (config, rootGroup)
13 }
```

CHAPTER 4

RENDERING CONSTRUCTS

Along with providing FRP abstractions for handling the dependencies and changes to entities in the application, Tesel also provides features to visually display these entities as they change through time. In keeping with our declarative programming approach, the rendering constructs only require the programmer to declare what they want to display to the screen. The Tesel runtime handles the low-level details of managing and passing the rendering data to the rendering hardware (*i.e.*, the GPU). The following sections discuss the core-component of a graphics application, mesh representations that can be used to represent entities in Tesel, and the constructs used to render those representations to the screen.

4.1 The Graphics Pipeline

It is important to first understand the core-component of graphics applications, namely the *graphics pipeline* [5], since Tesel incorporates this pipeline into its rendering process. The primary purpose of the pipeline is to generate a two-dimensional image, given a virtual camera and entities to render. Additional information can be passed to the pipeline that affects the appearance of the objects such as material properties, textures, and light sources. The locations and shapes of the entities are determined by properties of the scene, their geometry, and the placement of the virtual camera in the scene.

The graphics pipeline can be divided into two parts [25]: the first part transforms the objects positions (*i.e.*, their three-dimensional coordinates) into two-dimensional coordinates and the second part converts the two-dimensional coordinates into colored pixels on the screen. Each part contains several steps, where each step requires the output of the previous step as its input. Each step has a specific function that carries out its task and can easily be executed in parallel. Most steps are programmable, which means that the operations carried out by the GPU are specified by

small programs called *shaders*. They are defined as functions and are written by the programmer in a shading language.[1] Shaders give the developers finer-grain control over transforming the vertices or the colored pixels in the pipeline, because shaders run on the GPU in parallel, it reduces the need to process certain time-consuming tasks on the CPU. Linking these shaders together into a single object forms a *shader program*. The program is built on the CPU and passed to the GPU, which will use the shaders inside the program while running through the pipeline.

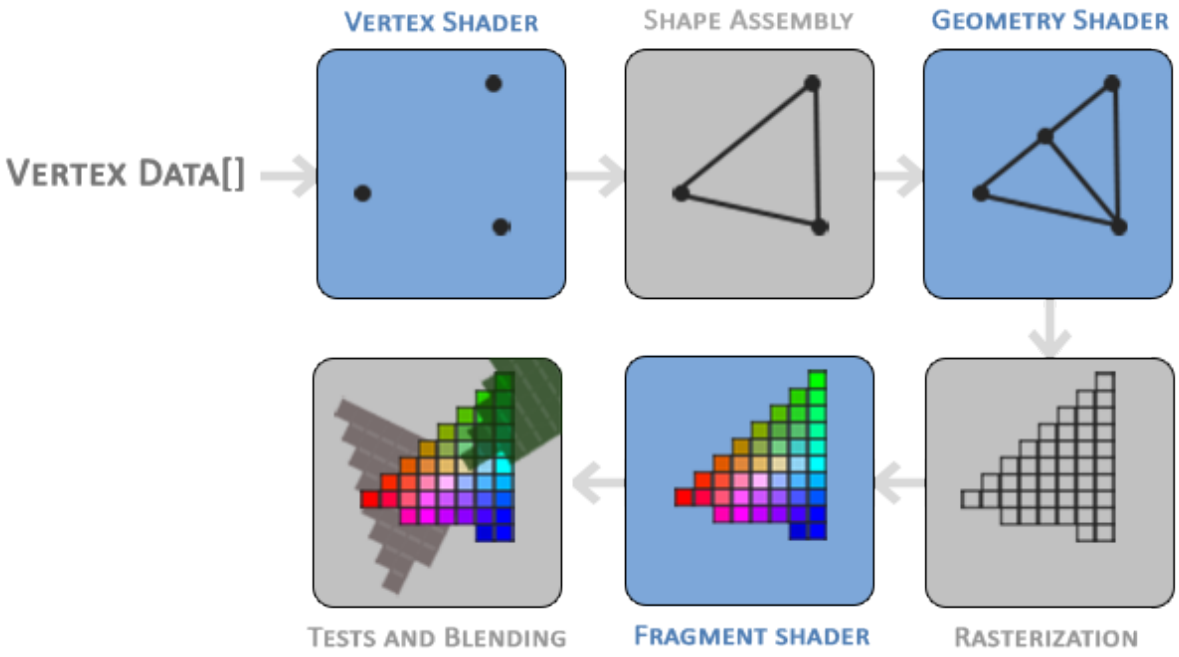


Figure 4.1: The stages of the graphics pipeline [25].

Figure 4.1 presents an overview of the main stages of the graphics pipeline. Each of these stages handles one particular part of the pipeline and are briefly explained as follows [25]:

- **Vertex Data**: A collection of three 3D coordinates are passed to the graphics pipeline as input. These three coordinates should form a triangle. Graphics hardware are optimized to operate more efficiently on triangles, since triangles tessellate well and are planar. The

1. A shading language is a graphics programming language adapted to programming shader effects on vertices and pixels.

coordinates are passed in as an array of *vertex data*, which is a collection of vertices. A vertex is essentially a collection of data per 3D coordinate. The data is represented using *vertex attributes* that can contain all essential data a vertex will need in the beginning stages such as position and color. Additional data such as uniforms and textures usually accompanies vertex data throughout the graphics pipeline. Uniforms are unique per shader program and can be accessed from any shader at any stage in the pipeline. Typical uses of uniforms include holding onto material and light data, which does not change through the pipeline and representing matrices that transform the vertices in the vertex shader.

Textures are used to bring a greater level of detail and realism to meshes. Each texture contains formatted image data that can be mapped onto an object. Specifically, each vertex is assigned a texture coordinate that specifies a point in texture space from the texture. These texels are then used to map the image on to the mesh. How texture data is interpreted while drawing is controlled by *samplers*. Samplers include various information about how to read the image data. For example, samplers include a property for texture filtering, which determines how to reconstruct the image data of a texture if the texture is drawn at a higher or lower resolution than its native size.

- **Vertex shader**: The vertex shader takes in a single vertex as input and transforms the 3D position component of the vertex data into normalized device coordinates (NDC), which are 3D coordinates where each coordinate component is bound between $[-1.0, 1.0]$. NDC values are needed by the pipeline in future steps. The shader also can perform necessary processing on the other vertex attributes (*e.g.*, normals, textures coords, *etc.*).

- **Shape assembly**: Vertices are connected together to make a primitive shape (*e.g.*, a triangle). Thus, the primitive assembly stage take as input all the vertices from the vertex shader that form a primitive and assembles all the positions in the primitive shape given (*i.e.*, it forms triangles from the vertices).

- **Geometry shader**: The geometry shader takes in a collection of vertices from the shape assembly stage. The collection of vertices form a primitive, which the geometry shader can transform to create new primitives by generating new vertices that form these new primitives. This type of shader function is not supported in Tesel.

- **Rasterization stage**: The formed primitives come into the rasterization stage where it maps these primitives to screen pixels. At this stage, pixels are not the actual screen pixels but rather they are known as fragments, which is a collection of data that is needed to render a single screen pixel. These fragments are then passed to the fragment shader, which actually assigns a pixel color .

- **Fragment shader**: Each fragment from the rasterization stage is processed by the fragment shader. The fragment shader is to compute the final color of a pixel. Most of the pixel shading algorithms are implemented in the fragment shader because the shader also contains data about the scene (*e.g.*, lights sources, and shadow information). This data becomes useful for determining the final color of each pixel.

- **Testing and blending**: Once all the color values have been identified, the object will then pass through the final stage known as alpha testing and blending. This stage has two important jobs. First, the stage checks if fragments should be discarded because they fall behind other fragments within the scene. This process is done by checking the depth values of the fragments. Lastly, the stage checks for alpha values and blends the objects together.

4.2 Mesh Representations

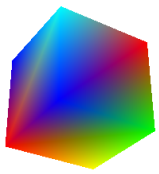
As described in 4.1, Meshes are a collection of vertex attribute data. The mesh representations in Tesel require this collection to contain values that conform to the `Attributes` protocol. For example, we can define a vertex struct that contains vertex attributes for position and color:

```
struct Vertex : Attributes {
    let pos        : float3
    let color      : float4
}
```

Tesel provides two types of mesh representations: a triangle mesh (`TriMesh`), which is a collection of vertex data where every three vertices form a triangle and an indexed face-vertex mesh (`ElementTriMesh`), which provides an array of vertex data and an integer array that specifies how triangles are formed. In particular, these integers represents indices into the vertices array that state which vertices form certain triangles. For example, Figure 4.3 and Figure 4.2 show the code to create a cube using both mesh representations and how they differ in color.

```
var cubeElementTriMesh : ElementTriMesh<Vertex> {
    let vertices = [
        Vertex(pos:float3(-1.0,  -1.0,  1.0),color:red),
        Vertex(pos:float3 (-1.0,  1.0,  1.0),color:green),
        Vertex(pos:float3 ( 1.0,  1.0,  1.0),color:blue),
        Vertex(pos:float3( 1.0,  -1.0,  1.0),color:skyBlue),
        Vertex(pos:float3 (-1.0,  -1.0, -1.0),color:pink),
        Vertex(pos:float3 (-1.0,  1.0, -1.0),color:yellow),
        Vertex(pos:float3 ( 1.0,  1.0, -1.0),color:red),
        Vertex(pos:float3 ( 1.0,  -1.0, -1.0),color:blue)]

    /* the indices that allow us to create the triangles. */
    let indices : [UInt16] = [
        0,2,1,  0,3,2,
        4,3,0,  4,7,3,
        4,1,5,  4,0,1,
        3,6,2,  3,7,6,
        1,6,5,  1,2,6,
        7,5,6,  7,4,5
    ]
    return ElementTriMesh<Vertex>(vertices, indices:indices)
}
```

(a) The code to create a indexed-triangle mesh.

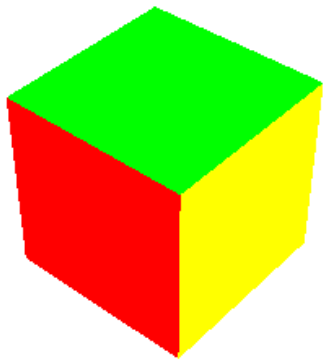


(b) A visual representation of the indexed-triangle mesh with the cube slightly rotated to see the various sides. The blending of colors comes from the interpolation of the vertices colors as it goes through the graphics pipeline.

Figure 4.2: The code and visual representation of creating a blended multi-colored cube in Tesel.

```
var cubeTriMesh : TriMesh<Vertex> {
    let vertices  = [  /* Front Side */
        Vertex(pos:float3( -1.0,  -1.0,   1.0), color:red),
        Vertex(pos:float3(  1.0,   1.0,   1.0), color:red),
        Vertex(pos:float3( -1.0,   1.0,   1.0), color:red),
        Vertex(pos:float3( -1.0,  -1.0,   1.0), color:red),
        Vertex(pos:float3(  1.0,  -1.0,   1.0), color:red),
        Vertex(pos:float3(  1.0,   1.0,   1.0), color:red),

        /* Bottom Side */
        Vertex(pos:float3( -1.0,  -1.0,  -1.0), color:green),
        Vertex(pos:float3(  1.0,  -1.0,   1.0), color:green),
        Vertex(pos:float3( -1.0,  -1.0,   1.0), color:green),
        Vertex(pos:float3( -1.0,  -1.0,  -1.0), color:green),
        Vertex(pos:float3(  1.0,  -1.0,  -1.0), color:green),
        Vertex(pos:float3(  1.0,  -1.0,   1.0), color:green),

        /* Left Side */
        Vertex(pos:float3( -1.0,  -1.0,  -1.0), color:yellow),
        Vertex(pos:float3( -1.0,   1.0,   1.0), color:yellow),
        Vertex(pos:float3( -1.0,   1.0,  -1.0), color:yellow),
        Vertex(pos:float3( -1.0,  -1.0,  -1.0), color:yellow),
        Vertex(pos:float3( -1.0,  -1.0,   1.0), color:yellow),
        Vertex(pos:float3( -1.0,   1.0,   1.0), color:yellow),

        /* Right Side */
        Vertex(pos:float3(  1.0,  -1.0,   1.0), color:blue),
        Vertex(pos:float3(  1.0,   1.0,  -1.0), color:blue),
        Vertex(pos:float3(  1.0,   1.0,   1.0), color:blue),
        Vertex(pos:float3(  1.0,  -1.0,   1.0), color:blue),
        Vertex(pos:float3(  1.0,  -1.0,  -1.0), color:blue),
        Vertex(pos:float3(  1.0,   1.0,  -1.0), color:blue),

        /* Top Side*/
        Vertex(pos:float3( -1.0,   1.0,   1.0), color:skyBlue),
        Vertex(pos:float3(  1.0,   1.0,  -1.0), color:skyBlue),
        Vertex(pos:float3( -1.0,   1.0,  -1.0), color:skyBlue),
        Vertex(pos:float3( -1.0,   1.0,   1.0), color:skyBlue),
        Vertex(pos:float3(  1.0,   1.0,   1.0), color:skyBlue),
        Vertex(pos:float3(  1.0,   1.0,  -1.0), color:skyBlue),

        /* Back Side */
        Vertex(pos:float3(  1.0,  -1.0,  -1.0), color:pink),
        Vertex(pos:float3( -1.0,   1.0,  -1.0), color:pink),
        Vertex(pos:float3(  1.0,   1.0,  -1.0), color:pink),
        Vertex(pos:float3(  1.0,  -1.0,  -1.0), color:pink),
        Vertex(pos:float3( -1.0,  -1.0,  -1.0), color:pink),
        Vertex(pos:float3( -1.0,   1.0,  -1.0), color:pink),
    ]
    return TriMesh<Vertex>(vertices)
}
```

(a) The code to create a triangle mesh.



(b) A visual representation of the triangle mesh with the cube slightly rotated to see the various sides.

Figure 4.3: The code and visual representation of creating a solid multi-colored cube in Tesel.

A cube has six faces, and two triangles represent each face; therefore, to create a cube mesh we need 12 triangles. Using a triangle mesh requires specifying all vertices to make up each triangle, which results in 36 vertices. A triangle mesh wastes memory because many of vertices are repeated. On the other hand, an indexed-triangle mesh only specifies the vertices of the mesh and uses an integer array to determine which vertices form the triangles. This representation saves on memory since the memory footprint of an integer is much smaller than a struct of attributes. The difference in the visual appearance of the cubes comes from the pipeline interpolating the color values. With the triangle mesh, each vertex of a face contain the same color. Thus, the interpolation of those colors will produce the same color. On the other hand, the indexed-triangle mesh reuses the same vertices for the different faces; therefore, the vertices of a face may have different colors, which the pipeline blends together.

Tesel also provides a class that already defines mesh representations for commonly used mesh primitives in graphics (*e.g.*, cubes, spheres, *etc.*). These representations are defined in the `SolidShapes` object and use a predefined attributes struct (`TeselDefaultAtrributes`), that provide commonly needed attribute data (*e.g.*, positions, normals, texture coordinates, *etc.*). For example, creating a sphere mesh is easily done using the `SolidShapes` object:

```
let sphere = SolidShapes.sphere()
```

4.3 Rendering Frames

A *frame* contains all the necessary data needed by the graphics pipeline (*e.g.*, vertex data, shader programs, uniforms, textures, *etc.*). In Tesel, a frame encompasses the data that will be sent to the GPU for rendering. The components that are used to define and render a frame in Tesel are described below.

- **Mesh entities**: A `MeshEntity` embeds all meshes that a particular frame will render:

```
MeshEntity<A : Attributes>(mesh: Mesh<A>,
                           uniforms:Uniforms? = nil,
                           textures: Textures? = nil,
                           samplers:Samplers? = nil)
```

Creating a mesh entity requires providing the mesh's vertex attribute data. The type `Mesh` is a protocol that both mesh representations implement, which allows a mesh entity to be defined in terms of either representation. We can create the cube mesh entity with only the vertex data as such:

```
let cubeEntity = MeshEntity<Vertex>(mesh:cubeElementTriMesh)
```

The other properties of the mesh entity are optional. The uniforms allow for specific mesh uniforms, such as the model-view-projection matrix to be specified. The texture parameter allows for passing an object to the shader program that holds all the texture data for the mesh. For example, multiple 2D textures for our cube mesh can be imported by defining `Texture2D` instances:

```
let cubeTex1 = Texture2D(fileToLoad:"cubetexture1.png")
let cubeTex2 = Texture2D(fileToLoad:"cubetexture2.png")
let cubeTex3 = Texture2D(fileToLoad:"cubetexture3.png")
```

We define a struct that contains properties for all the cube textures and then create an instance of this struct to pass to the mesh entity:

```
struct CubeTextures : Textures {
        let texture1 : Texture2D
        let texture2 : Texture2D
        let texture3 : Texture2D
}
let cubeTexs = CubeTextures(texture1:cubeTex1, texture2:cubeTex2, texture3:cubeTex3)
```

Finally, we can specify any number of samplers for sampling the mesh's textures. Similar to

textures, we first create a `Sampler` instances,

```
// Using the default init function creates
// linear filtering properties and clamps
// the wrapping modes to zero.
let cubeSampler = Sampler()
```

followed by a struct definition that conforms to the `Samplers` protocol and then creates an instance of this struct to pass to the mesh entity.

```
struct CubeSamplers : Samplers  {
        let sampler : Sampler
}
let cubeSamplers = CubeSamplers(sampler:cubeSampler)
```

- **Shader programs**: Graphics programmers write their shader programs in a shading language. For example, the OpenGL Shading Language (GLSL) [70] is a C-like shading language that is used for programmer shader programs and contains useful features such as vector and matrix types and operators. Additionally, the shading language provides the programmer with mechanisms to retrieve the inputs and outputs for each shader stage and the uniform data used throughout the pipeline. Unlike other low-level APIs, shader programs in Tesel are written in Swift. Although this feature makes it easier on the programmers since they do not have to learn two languages, shader programs are only allowed to use a subset of Swift's features. Specifically, the subset is limited to the following mechanism:
  - variable and function declarations with explicit type annotations
  - primitive types (*e.g.*Float, Double, Int) and the arithmetic operations associated with these types
  - plain-old struct definitions that contain no init or methods with the exception of the struct that conforms to the shader program protocol
  - the SIMD library that provides all many of the need vector and matrix manipulation

functions

- control-flow constructs with the exception of for-each loop, which we plan to provide at later time

Although this subset may seem limiting, it is sufficient to write many useful algorithms that are written in other shading languages such as lighting calculations.

Defining a shader program requires defining a class, or struct that conforms to the `ShaderProgram` protocol:

```
public protocol ShaderProgram {
    var vertFuncName : String {get}
    var fragFuncName : String {get}
}
```

Each shader program is required to define two string properties that provide the names of the vertex and fragment functions in the program, specifically these need to be literal strings (see Section 6 for further details). An example of defining a flat shader program (*i.e.*, the primitive triangles are rendered in the object's color without any lighting calculations) is shown in Figure 4.4. Lines 1–8 define the uniforms and attributes used in this shader program. Lines 9–34 defines the definition of the flat shader program. The required properties of the `ShaderProgram` are defined on Lines 11–12 and are given the string literals of the vertex and fragment functions (*i.e.*, shaders) respectively. As stated earlier, the vertex shader is responsible for passing out the NDC coordinates for a vertex. But the shader can pass out additional information along with the NDC coordinates as outputs. Defining a struct that conforms to the `FragInput` protocol:

```
public protocol FragInput {
    var position : float4 {get}
}
```

This protocol only requires that a position property is defined, which represents the NDC

coordinate for a vertex. Additionally, the struct conforming to this protocol can define additional properties to pass out of the vertex shader (*e.g.*normalized normals, texture coordinates, etc). The data is interpolated and converted into fragments that will be inputs to the fragment by the graphics pipeline. We define this struct on Lines 14–17. In Tesel, the vertex function can only take in five types of parameters: attributes, uniforms, textures, samplers, or a vertex information object. The vertex information object contains data about the current vertex being executed (*i.e.*, vertex's instance id and vertex id). The flat shader's vertex function takes in the vertex attribute data (`vert`) and the uniforms for the mesh (`uniforms`) and returns an instance of the projected vertex. Inside the vertex function, we convert the vertex's position into its NDC coordinates (Lines 21–24) and return an instance of the user-define `ProjectedVertex` type that contains the converted coordinates[2] and the color of the vertex. The fragment shader can only take in five types of parameters: `FragInput` objects, fragment information objects, uniforms, textures, and samplers. The fragment information object contains information about the each fragment such as the relative window position for the fragment and sample id for texturing. The output of the fragment shader is always a `float4` that represents the color assigned to the fragment. The projected vertex object is passed into the fragment shader (Line 28) and we use it to assign each fragment its interpolated color (Line 30).

2. The vertex function can also be defined to return a `float4` if only the NDC coordinates is passed through to the next stage.

```
1  struct FrameUniforms : Uniforms {
2      let model          : float4x4;
3      let projectionView : float4x4;
4  }
5  struct Vertex : Attributes {
6      let pos    :  float3;
7      let color  :  float4;
8  }
9  public struct FlatShaders : ShaderProgram {
10
11     public var vertFuncName   : String  = "vertexFunc";
12     public var fragFuncName   : String  = "fragFunc";
13
14     struct ProjectedVertex : FragInput
15     {
16         var position : float4;
17         var color    : float4;
18     }
19     func vertexFunc(vert: Vertex,
20                     uniforms: FrameUniforms) -> ProjectedVertex
21     {
22         let ndCoord: float4 = uniforms.projectionView *
23                               uniforms.model *
24                               float4(vert.pos, 1.0);
25
26         return  ProjectedVertex(position:ndCoord, color:vert.color);
27     }
28     func fragFunc(pVertex: ProjectedVertex) -> float4
29     {
30         return pVertex.color;
31     }
32 }
```

Figure 4.4: An example of a flat shader program implemented in Tesel.

- **Render entities**: A mesh render entity:

```
MeshRenderEntity<A : Attributes, T : ShaderProgram> (meshes:[MeshEntity<A>],
                                                     program: T,
                                                     info : ShaderInfo)
```

combines all the meshes into a single entity that the shader program will render. The shader information parameter is needed by the runtime to perform additional optimizations, and every shader program by default has a type property called info. A flat cube can be rendered as follows:

```
let shader = FlatShaders()
let renderEntity  = MeshRenderEntity<Vertex,FlatShaders>(meshes:[cubeEntity],
                                                         program:shader,
                                                         info:FlatShaders.info)
```

- **Frames**: A frame:

```
Frame (clearColor:float4, entities:[MeshRenderEntity])
```

defines a clear color for the frame (*i.e.*the background color of the window), and all the mesh render entities that need to be rendered. We define a frame that renders a flat cube:

```
let frame = Frame(clearColor:float4(0.0,0.29803,0.6,1.0), entities:[renderEntity])
```

Pushing a frame to be run on the GPU requires calling the render function:

```
func render<I>(fn: I -> Frame, in1: DSignal<I>) -> Signal<Frame>
```

The render function creates a signal frame that sends the frame returned by its update function to the GPU. As with the other signal function primitives, render function can take in more than one discrete input to its function. For example, we can define a frame signal that sends a frame to the GPU after executing its update function:

```
// projection : DSignal<float4x4> = ...
// camMat : DSignal<float4x4>   = ...

//Create and tag the frame signal to automatically send
its updated frame to the GPU for rendering.
let frameSig = render(self.makeFrame, in1:projection, in2:camMat)

func makeFrame(projection: float4x4, camera : float4x4) -> Frame {
        // Make the frame
       // ...
}
```

4.4 Rendering a Bouncing Ball

In section 3.2.4, we implemented the bouncing ball example as a dynamic signal group in Tesel. We also discussed how to connect the ball group to a `Viewer` signal group. Using the rendering constructs discussed in this chapter, we will render the bouncing ball along with a flat plane representing the floor.

First, we define the uniforms struct and shader program that is used to render the floor and bouncing ball:

```
1  struct FrameUniforms : Uniforms {
2      var model          : float4x4;
3      var projectionView : float4x4;
4      var color          : float4;
5  }
6  public struct FlatShaders : ShaderProgram {
7
8      public var vertFuncName   : String  = "vertexFunc";
9      public var fragFuncName   : String  = "fragFunc";
10
11     struct ProjectedVertex : FragInput
12     {
13         var position : float4;
14     }
15     func vertexFunc(vInfo: VertexInfo,
16                 vert: TeselDefaultAtrributes,
17                 uniforms: FrameUniforms) -> ProjectedVertex {
18         let clipCoord : float4 = uniforms.projectionView *
19                                  uniforms.model *
20                                  float4(vert.pos, 1.0);
21         return  ProjectedVertex(position:clipCoord);
22     }
23     func fragFunc(fInfo: FragInfo,
24                 pVertex: ProjectedVertex,
25                 uniforms: FrameUniforms) -> float4
26     {
27         return uniforms.color;
28     }
```

```
29  }
```

The uniforms definition is defined on Lines 1–5. It contains common properties needed to move vertices into the various coordinate spaces such as the model, and the combined projection and view matrices. It also contains a color for the mesh. We then define the vertex (Lines 15–22) and fragment (Lines 23–28) functions for the shader program. The vertex function uses Tesel's default attributes type to represent its vertex attributes. The vertex function converts the vertex position into its NDC coordinates and returns the user-defined struct `ProjectedVertex` as the function result. The fragment function returns the uniform's color to assign as its pixel color.

The updated `Viewer` class that uses the `FlatShaders` definition as its shader program is defined as follows.

```
1  struct Viewer : SignalGroup {
2
3      var frame     : DSignal<Frame>!
4
5      init(initWinSize : CGSize) {
6
7          // Create the bouncing ball group
8          let initialBall = Ball(yPos:10.0, yVel:0)
9          let bouncingBall = group(BouncingBall(initBall:initialBall),
10                                   in1:TeselExternal.render)
11         let ball = hold(initialBall, in1:bouncingBall)
12
13         // Create a camera group
14         let camera = Camera(initPos:float3(0.0,0.0,3.0),
15                             front:float3(0,0,-1),
16                             up:float3(0,1,0))
17
18         let camMatrix = group(camera, in1:camera.matrix)
19
20
21         // Create a projection signal to know when the
22         // viewer is being resized
23         let projection = hold(mkProjectInfo(initWinSize),
```

```
                                in1:lift(self.reshape,
                                         in1:TeselExternal.windowSize))

        // Render a frame using the project, camera matrix, and ball
        self.frame =  render(self.flatFrame,in1:projection,
                                in2:camMatrix, in3:ball)
    }
    func flatFrame(projInfo: ProjectionInfo,
                camMat : float4x4, ball:Ball) -> Frame {
        // Define and retrieve the ball mesh entity
        let ballEnt = ballEntity(projInfo.matrix, camera: camMat, ball: ball)

        // Define and retrieve the floor mesh entity
        let floorEntity = floorMeshEntity(projInfo.matrix, camera: camMat)

       // Define an instance of a flat-shader program
        let shader = FlatShaders()

        // Define a render entity that contains the ball and floor
        // mesh entities and uses the flat shaders program to
        // render them.
        let entity = MeshRenderEntity<TeselDefaultAtrributes,
                                      FlatShaders>(
                                      meshes:[floorEntity, ballEnt],
                                                  program:shader,
                                                  info:FlatShaders.info)

       // Create a frame that uses the flat render entity
       var frame = Frame(clearColor:float4(0.0,0.29803,0.6,1.0),
                         entities:[entity])

        return frame
    }
   func mkProjectInfo(winSize : CGSize) -> ProjectionInfo {
        let aspect = Float(winSize.width / winSize.height)
        let projection = Transform.perspective(60, aspect:aspect,
                                               nearZ:0.1e-1, farZ:100)
        return ProjectionInfo(matrix:projection, winSize:winSize)
    }
    func reshape(window : WinEvent) -> ProjectionInfo {
```

```
64          let winSize = window.size
65          return mkProjectInfo(winSize)
66      }
67      func floorMeshEntity(projection : float4x4,
68                  camera : float4x4)  -> MeshEntity<TeselDefaultAtrributes> {
69          let mesh = SolidShapes.plane()!
70          let transMat = Transform.translate(x:0.0, y:-Ball.radius/2.0, z: 0.0)
71          let rotateMat = Transform.rotate(transMat, x: 90.0, y: 0.0, z: 0.0)
72          let model = Transform.scale(rotateMat,
73                                  scaleVector:float3(x:10.0, y: 10.0, z: 1.0))
74          let uniforms = FrameUniforms(model:model,
75                                       projectionView:(projection * camera),
76                                       color: float4(1.0,1.0,0.0,1.0))
77          return MeshEntity<TeselDefaultAtrributes>(mesh:mesh,
78                                                    uniforms:uniforms,
79                                                    textures:nil,
80                                                    samplers:nil)
81      }
82      func ballEntity(projection : float4x4,
83                  camera : float4x4,
84                  ball: Ball) -> MeshEntity<TeselDefaultAtrributes> {
85          let bPos = float3(0.0, ball.yPos, 0.0)
86          let radius : Float = Ball.radius
87          let mesh = SolidShapes.sphere()!
88          let transMat = Transform.translate(x:bPos.x, y:bPos.y , z: bPos.z)
89          let model = Transform.scale(transMat,sx:radius, sy:radius, sz:radius)
90          let uniforms = FrameUniforms(model:model,
91                                       projectionView:(projection * camera),
92                                       color: float4(1.0,0.0,0.0,1.0))
93          return MeshEntity<TeselDefaultAtrributes>(mesh:mesh,
94                                                    uniforms:uniforms,
95                                                    textures:nil,
96                                                    samplers:nil)
97      }
```

The code on lines 7–18 is the same as the code described in section 3.2.4, which defines and connects the bouncing ball and camera signals groups to the viewer signal group. We define a signal to handle the reshaping of the event occurrence of the window resizing (lines 23–25). The

projection creates a `ProjectInfo` object that contains the window size and the projection matrix using. We use these signals to create a frame for rendering the ball (lines 28–29). The `flatFrame` function (lines 31–51) creates a frame that contains all the mesh entities that are rendered using the flat-shader program.

The `flatFrame` function uses two functions we define for retrieving the mesh entities for the ball and floor the ball bounces on. The `ballEntity` function (lines 78–92), we first define the 3D position for the ball (line 81) and retrieve the ball radius (line 83). The mesh for the ball is a sphere mesh from the `SolidShapes` library. The model matrix for the mesh is translated and scaled up by the properties of the ball (*i.e.*, its position, and the size of its radius)(lines 84–85). The uniforms for the ball mesh contains its model matrix, a concatenation of its projection and camera matrix, and a red color (lines 86–88). We define and return the `MeshEntity` for the ball with its mesh and uniform data on line 92. The `floorMeshEntity` (lines 63–77) uses the `SoldSphapes` plane mesh as the floor's mesh. The ball bounces when its y-value is equal to zero, but if we place the floor at the origin then it appears as if the ball was going through the floor since its position is centered at its radius. Visually, this setup appears as if half the ball is above the floor and the other half below it. Thus, we translate the ball down by half its radius (line 69). Initially, the plane appears standing up on the y-axis; therefore, we rotate it by 90 degrees to lay it flat on the x-axis (line 70). We also scale the ball up on the x and y axis (line 71). As with the ball mesh entity, we define the uniforms containing the floor's model matrix, the projection-view matrix, and a yellow color. A `MeshEntity` with the floor's mesh and uniforms is returned on line 76.

Inside the `flatFrame` function, We retrieve the mesh entities for the ball and floor by calling the `ballEntity` function (line 34) and `floorMeshEntity` function (lines 37). We combine these entities together inside a `MeshRenderEntity` instance we create on lines 43-46 along with a `FlatShaders` instance (line 40). Finally, we define and return the frame with a background color and the flat-shading render entity (lines 51–52).

CHAPTER 5

THE FORMAL SEMANTICS OF TESEL

Defining a formal semantics for a programming language allows implementors and users to fully understand the meanings of the language's constructs in an unambiguous manner. In this chapter, we present the formal semantics of a small language that represents the core features of Tesel, which we call "Mini-**Tesel**". The following sections provide the syntax for Mini-**Tesel** and its dynamic and static semantics. Mini-**Tesel** also provides a simple expression syntax (*e.g.*, let variables, functions, constants, *etc.*) along with the syntax for the core constructs of Tesel.

5.1 Notation

Before discussing the semantics of Mini-**Tesel**, we define the notation that we use throughout this chapter. If A and B are two sets, then $A \cup B$ is their union, and $A \cap B$ is their intersection, and $A \setminus B$ is their difference. We use the notation A $\xrightarrow{\text{fin}}$ B to denote the set of finite maps from A to B and A $\cup$ B as there union. We write $\{a_1 \mapsto b_1, ..., a_n \mapsto b_n\}$ for the finite map that maps a_1 to b_1, etc. For a finite map, f, we write $dom(f)$ for its domain and $rng(f)$ for its range. If f and g are finite maps, we write $f \pm g$ for the finite map

$$\{x \mapsto g(x) \mid x \in dom(g)\} \cup \{x \mapsto f(x) \mid x \in dom(f) \setminus dom(g)\}$$

and we write $f[x \mapsto v]$ to represent a binding of a variable to a value that is defined as:

$$f[x \mapsto v](y) = \begin{cases} v & \text{if } x = y \\ f(y) & \text{if } x \neq y. \end{cases}$$

If X and Y are two tuples then $X \oplus Y$ is the concatenation of their components. Finally, If X is a tuple then $\{X[n_i] | n_i \in \mathbb{N}\}$ means the component at n_i in X and $\{X[n_i : n_j] \mid n_i,\ n_j \in \mathbb{N}\}$ means concatenating the components from n_i to n_j (inclusive) of X.

5.2 Syntax

We first present the *surface syntax* of Mini-**Tesel**.[1] We assume the existence of countable sets of *variables*, *constants*, *base constants*, *function constants* and *external signal identifiers*:

x	$\in$	VAR	variables
c	$\in$	CONST $=$ BCONST $\cup$ FCONST	constants
		BCONST $= \{(), true, false, 0, 1, ...\}$	base constants
		FCONST $= \{+, -, *, ...\}$	function constants
i	$\in$	EXTERNALSIGIDENTIFIER $= \{keyEvt,\ mouseEvt, ...\}$	external signal identifiers

The set of FCONST also includes the following core Mini-**Tesel** primitives and signal group constructors:

foldpD	**foldpE**	**merge**	**integral**
collectS	**collectD**	**after**	**filter**
liftD	**liftE**	liftC	**constantC**
constantD	**groupC**	**groupD**	**groupS**
external	**mkGroup**		

The following grammar defines of the surface syntax for expressions and function abstraction in Mini-**Tesel**:

1. The syntax used by programmers and accepted by compilers.

f	::=	**fun**(x) => e	function abstraction
e	::=	x	variables
	|	c	constants
	|	**None**	none optional
	|	**Some**(e)	some optional
	|	**let** x = e_1 **in** e_2	let binding
	|	$(e_1, \ ..., \ e_n)$	tuples
	|	$e.c$	tuple selection
	|	**if** e_1 **then** e_2 **else** e_3	if expression
	|	f	function abstraction
	|	$(e_1 \ e_2)$	function application
	|	i	external signal identifier

The dynamic semantics of Mini-**Tesel** are defined in terms of an *internal syntax*, which is the surface syntax extended with special syntactic forms used to represent semantic objects. First, we distinguish one subset of expressions: *values* (denoted v), which are irreducible terms in the dynamic semantics. Values are considered as the least subsets of expressions and they also include the terms for our semantic signal objects.

We define three sets of semantic objects for *discrete* (denoted ω), *continuous* (denoted γ), and *event* signals (denoted κ) that are defined as internal representations of the core primitives. We also define a *universal signal* set (denoted α) that is the union of the continuous, discrete, and event signal sets. We also define sets to represent *signals groups* (denoted β). Finally, we define a set of *time values* (denoted T), which are non-negative real numbers.

Finally, we define a set of *programs* (denoted p), which is a tuple containing an expression (denoted e), and a tuple of *update periods* (denoted ψ). Each update period is composed of a tuple of updated external event identifiers with their value (denoted $((i_1, v_1), \ ..., \ (i_n, v_1))$), a boolean value (denoted c) that represents whether the continuous signals should be sampled at this time, and the current time (denoted T). Thus, the time values within the update period tuples ($\psi_0, \ ..., \ \psi_n$) are monotonically increasing since they represent absolute time (*i.e.*, $T_0 \ < \ T_1, \ < \ ... \ < \ T_N$). For example, a update period could contain a time T, a **true** value, and an event identifier that

represents the mouse events. Thus, this update period signifies that at time T a mouse event occurred and the continuous signals are sampled. It is important to have this distinction between sampling and event occurrences to allow for updates to the program (*i.e.*, event occurrences) in between sampling periods. We also define a set of update functions (denoted χ) that update the signal and signal group objects at each update period.

We extend the surface language with the following grammar productions:

$$
\begin{array}{rcll}
p & ::= & (e, (\psi_1,\ ...,\ \psi_n)) & \\
\psi & ::= & (((i_1,\ v_1),\ ...,\ (i_n,\ v_n)),\ c,\ T) & \\
v & ::= & c & \\
 & | & (v_1,\ ...,\ v_n) & \\
 & | & \alpha & \\
 & | & \omega & \\
 & | & T & \\
 & | & \kappa & \\
 & | & \gamma & \\
 & | & \texttt{None} & \\
 & | & \texttt{Some}(v) & \\
 & | & f & \\
 & | & i & \\
 & | & \beta & \\
T & ::= & c & \\
\chi & ::= & \text{upPeriod}(\beta,\ \psi) &
\end{array}
$$

$$
\begin{array}{rcll}
\alpha & ::= & \kappa \mid \omega \mid \gamma & \text{signals} \\
\kappa & ::= & \textsc{Lifte}(v_1,\ v_2) & \text{lift event signal} \\
 & | & \textsc{Merge}((\kappa_1,\ ...,\ \kappa_n)) & \text{merge signal} \\
 & | & \textsc{After}(T,\ c_1)c_2 & \text{after signal} \\
 & | & \textsc{Filter}(f,\ \kappa) & \text{filter signal} \\
 & | & \textsc{Groupe}(\beta,\ \kappa) & \text{group event signal} \\
 & | & \textsc{External}(i) & \text{extermal event signal} \\
\omega & ::= & \textsc{Foldpd}(v,\ f,\ \omega) & \text{foldp discrete signal} \\
 & | & \textsc{Foldpe}(v,\ f,\ \kappa) & \text{foldp discrete-event signal} \\
 & | & \textsc{Liftd}(f,\ v) & \text{lift discrete signal} \\
 & | & \textsc{Constantd}(v) & \text{constant discrete signal} \\
 & | & \textsc{Groupd}(\beta,\ \omega) & \text{group discrete signal} \\
\gamma & ::= & \textsc{Integral}(v,\ \gamma) & \text{integral signal} \\
 & | & \textsc{Collects}(v,\ f) & \text{static collection signal} \\
 & | & \textsc{Collectd}(v,\ f_1,\ f_2,\ \kappa) & \text{dynamic collection signal} \\
 & | & \textsc{Liftc}(f,\ v) & \text{constant continuous signal} \\
 & | & \textsc{Constantc}(v) & \text{constant continuous signal} \\
 & | & \textsc{Groupd}(\beta,\ \gamma) & \text{group continuous signal} \\
\beta & ::= & (\alpha_1,\ ...,\ \alpha_n) & \text{signal group}
\end{array}
$$

The dynamic semantics uses substitution-based semantics, instead of environment-based semantics. Thus, variables are not included in the set of values. Additionally, the update period function for updating signal groups is an internal function of the evaluation system and is not accessible in the surface language. Thus, there are eleven syntactic classes of terms in the surface language of Mini-**Tesel**:

p	$\in$	PROGRAMS	programs
ψ	$\in$	UPDATEPERIOD	update periods
e	$\in$	EXPR	expressions
v	$\in$	VAL $\subset$ EXPR	values
f	$\in$	FUNABS $\subset$ VAL	function abstraction
T	$\in$	TIME $\subset$ EXPR	time values
α	$\in$	SIGNAL $\subset$ VAL	signals
κ	$\in$	EVENTSIGNAL $\subset$ SIGNAL	event signals
ω	$\in$	DISCRETESIGNAL $\subset$ SIGNAL	discrete signals
γ	$\in$	CONTINUOUSSIGNAL $\subset$ SIGNAL	continuous signals
β	$\in$	SIGNALGROUP $\subset$ VAL	signal groups

5.3 Dynamic Semantics

Specifying the dynamic semantics requires defining additional environments and finite maps. We use Ψ to denote a set of tuples that contain a tuple of the changed external identifiers with their values for a update period, the current time (*i.e.*, the time at the beginning of the update period), and the time from the last update period (*i.e.*, the delta time) and a boolean value indicating whether this update needs to sample the continuous signals.

$$\Psi \;=\; (((i_1, v_1),\ ...,\ (i_n, v_n)),\ T_1,\ T_2,\ c) \;\in\; \text{UPDATEENVS} \quad \text{update period environment}$$

We also define a finite map (σ) that maps variables to values:

$$\sigma \;=\; \text{VAR} \xrightarrow{\text{fin}} \text{VALUE} \quad \text{local variable environment}$$

The following are the evaluation relations for the dynamic semantics:

$$
\begin{array}{lll}
(1) & \sigma \vdash e \Downarrow v & \textit{expression relation} \\
(2) & \Psi \vdash \alpha \Downarrow (v, \alpha') & \textit{signal relation} \\
(3) & \Psi \vdash \beta \Downarrow \beta' & \textit{signal group relation} \\
(4) & (\Psi, \chi) \Downarrow (\Psi', \beta) & \textit{update period relation} \\
(5) & p \Downarrow v & \textit{program relation}
\end{array}
$$

5.3.1 Expression Evaluation

Before showing the expression rules, we assume their exists a partial function (δ) to evaluate function constants:

$$\delta \;:\; \text{FCONST} \times \text{VAL} \mapsto \text{VAL}$$

The standard function constants adhere to the their mathematical meanings as expected. For example:

$$
\begin{aligned}
\delta(*, (3,\ 2)) &\equiv_{def} 6 \\
\delta(>, (3,\ 2)) &\equiv_{def} \mathit{true} \\
\delta(\mathit{fst}, (v_1,\ v_2)) &\equiv_{def} v_1
\end{aligned}
$$

The δ function also contains translations from the surface-syntax of Mini-**Tesel** signal and signal group objects to their internal representation. The translation rules are as follows:

$$
\begin{aligned}
\delta(\mathbf{integral}, \gamma) &\equiv_{def} \textsc{Integral}(0,\ \gamma) \\
\delta(\mathbf{collectS}, (v, f)) &\equiv_{def} \textsc{CollectS}(v,\ f) \\
\delta(\mathbf{collectD}, (v,\ f_1,\ f_2,\ \kappa)) &\equiv_{def} \textsc{CollectD}(v,\ f_1,\ f_2,\ \kappa) \\
\delta(\mathbf{liftC}, (f, \gamma)) &\equiv_{def} \textsc{LiftC}(f,\ \gamma) \\
\delta(\mathbf{constantC}, v) &\equiv_{def} \textsc{ConstantC}(v) \\
\delta(\mathbf{groupC}, (\beta,\ \gamma)) &\equiv_{def} \textsc{GroupC}(\beta,\ \gamma) \\
\delta(\mathbf{foldpD}, (v,\ f,\ \omega)) &\equiv_{def} \textsc{FoldpD}(v,\ f,\ \omega) \\
\delta(\mathbf{foldpE}, (v,\ f,\ \kappa)) &\equiv_{def} \textsc{FoldpE}(v,\ f,\ \kappa) \\
\delta(\mathbf{liftD}, (f, \omega)) &\equiv_{def} \textsc{LiftD}(f,\ \omega) \\
\delta(\mathbf{constantD}, v) &\equiv_{def} \textsc{ConstantD}(v) \\
\delta(\mathbf{groupD}, (\beta,\ \omega)) &\equiv_{def} \textsc{GroupD}(\beta,\ \omega) \\
\delta(\mathbf{merge}, (\kappa_1,\ ...,\ \kappa_n)) &\equiv_{def} \textsc{Merge}((\kappa_1,\ ...,\ \kappa_n)) \\
\delta(\mathbf{after}, (T)) &\equiv_{def} \textsc{After}(T,\ false) \\
\delta(\mathbf{liftE}, (f, \kappa)) &\equiv_{def} \textsc{LiftE}(f,\ \kappa) \\
\delta(\mathbf{filter}, (f, \kappa)) &\equiv_{def} \textsc{Filter}(f,\ \kappa) \\
\delta(\mathbf{groupE}, (\beta,\ \kappa)) &\equiv_{def} \textsc{GroupD}(\beta,\ \kappa) \\
\delta(\mathbf{mkGroup}, (\alpha_1,\ ...,\ \alpha_n)) &\equiv_{def} (\alpha_1,\ ...,\ \alpha_n) \\
\delta(\mathbf{external}, i) &\equiv_{def} \textsc{External}(i)
\end{aligned}
$$

The evaluation rules for the expressions in the surface-language are defined as follows.

(e-**ifTrue**):

$$\frac{\sigma \vdash\ e_1\ \Downarrow\ \texttt{true} \qquad \sigma \vdash\ e_2\ \Downarrow\ v}{\sigma \vdash\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \Downarrow\ v}$$

(e-**ifFalse**):

$$\frac{\sigma \vdash\ e_1\ \Downarrow\ \texttt{false} \qquad \sigma \vdash\ e_3\ \Downarrow\ v}{\sigma \vdash\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \Downarrow\ v}$$

(e-**tuple**):

$$\frac{\sigma \vdash\ e_1\ \Downarrow\ v_1 \qquad ... \qquad \sigma \vdash\ e_n\ \Downarrow\ v_n}{\sigma \vdash\ (e_1,\ ...,\ e_n)\ \Downarrow (v_1,\ ...,\ v_n)}$$

(e-**variable**):

$$\frac{\sigma(x)\ =\ v}{\sigma \vdash\ x\ \Downarrow\ v}$$

(e-**optSome**):

$$\frac{\sigma \vdash e \Downarrow\ v}{\sigma \vdash\ \texttt{Some}(e)\ \Downarrow \texttt{Some}(v)}$$

(e-**let**):

$$\frac{\sigma \vdash\ e_1\ \Downarrow\ v \qquad \sigma \pm \{x\ \mapsto\ v\} \vdash\ e_2\ \Downarrow\ v'}{\sigma \vdash\ \texttt{let}\ x\ =\ e_1\ \texttt{in}\ e_2\ \Downarrow v'}$$

(e-**tupleSelection**):

$$\frac{\sigma \vdash\ e\ \Downarrow\ (v_1,\ ...,\ v_n) \qquad c\ \in\ \mathbb{Z} \qquad v_c\ \in\ (v_1,\ ...,\ v_n)}{\sigma \vdash\ e.c\ \Downarrow\ v_c}$$

(e-**functionApp**):

$$\frac{\sigma \vdash e_1 \Downarrow\ \ \texttt{fun}(x) => e \qquad \sigma \vdash e_2 \Downarrow\ \ v \qquad \sigma \pm \{x\ \mapsto\ v\} \vdash e \Downarrow\ \ v'}{\sigma \vdash\ (e_1\ e_2)\ \Downarrow\ v'}$$

(e-**funcConstant**):

$$\frac{\sigma \vdash e_1 \Downarrow\ c \qquad c\ \in\ \text{FCONST} \qquad \sigma \vdash e_2 \Downarrow\ \ v}{\sigma \vdash\ (e_1\ e_2)\ \Downarrow\ \delta(c, v)}$$

5.3.2 Program Evaluation

This section provides the evaluation rules for signals, signal groups, and programs. Owing to margin constraints, a few of the rule's predicates are provided on two lines. The predicates should be read from left to right and starting at the top line to the bottom line.

Continuous Signal Evaluation: The evaluation rules for updating continuous signals is as follows:

(γ-**integral**):

$$\frac{\Psi[4] \;=\; \texttt{true} \qquad \Psi \vdash \gamma \Downarrow (v_1,\ \gamma') \qquad v' \;=\; v \;+\; (\Psi[3] \;*\; v_1)}{\Psi \vdash \textsc{Integral}(v,\ \gamma) \;\Downarrow (v',\ \textsc{Integral}(v',\ \gamma'))}$$

(γ-**collects**):

$$\frac{\Psi[4] \;=\; \texttt{true} \qquad \Psi \vdash (f\ v_1) \Downarrow v_1' \qquad ... \qquad \Psi \vdash (f\ v_n) \Downarrow v_n'}{\Psi \vdash \textsc{Collects}((v_1,\ ...,\ v_n),\ f) \;\Downarrow ((v_0',\ ...,\ v_n'),\ \textsc{Collects}((v_0',\ ...,\ v_n'),\ f))}$$

(γ-**collectd-Event**):

$$\frac{\begin{array}{c}\Psi[4] \;=\; \texttt{true} \qquad \Psi \vdash \kappa \Downarrow (v, \kappa') \\ \Psi \vdash (f_1\ v_1) \Downarrow v_0' \qquad ... \qquad \Psi \vdash (f_1\ v_n) \Downarrow v_n' \qquad \Psi \vdash (f_2\ (v_1',\ ...,\ v_n')) \Downarrow (v_1'',\ ...,\ v_n'')\end{array}}{\Psi \vdash \textsc{Collectd}((v_1,\ ...,\ v_n),\ f_1,\ f_2,\ k) \;\Downarrow ((v_1'',\ ...,\ v_n''),\ \textsc{Collectd}((v_1'',\ ...,\ v_n''),\ f_1,\ f_2,\ \kappa'))}$$

(γ-**collectd-NoEvent**):

$$\frac{\begin{array}{c}\Psi[4] \;=\; \texttt{true} \qquad \Psi \vdash \kappa \Downarrow (\texttt{None}, \kappa') \\ \Psi \vdash (f_1\ v_1) \Downarrow v_1' \qquad ... \qquad \Psi \vdash (f_1\ v_n) \Downarrow v_n'\end{array}}{\Psi \vdash \textsc{Collectd}((v_1,\ ...,\ v_n),\ f_1,\ f_2,\ k) \;\Downarrow ((v_1',\ ...,\ v_n'),\ \textsc{Collectd}((v_1',\ ...,\ v_n'),\ f_1,\ f_2,\ \kappa'))}$$

(γ-**liftc**):

$$\frac{\Psi[4] \;=\; \texttt{true} \qquad \Psi \vdash \gamma \Downarrow (v_1,\ \gamma') \qquad \Psi \vdash (f\ v_1) \Downarrow v_1'}{\Psi \vdash \textsc{Liftc}(f,\ \gamma) \;\Downarrow (v_1',\ \textsc{Liftc}(f,\ \gamma'))}$$

(γ-**constantc**):

$$\frac{\Psi[4] \;=\; \texttt{true}}{\Psi \vdash \textsc{Constantc}(v) \;\Downarrow (v,\ \textsc{Constantc}(v))}$$

(γ-**groupc**):

$$\frac{\Psi[4] \;=\; \texttt{true} \qquad \Psi \vdash \beta \;\Downarrow \beta' \qquad \Psi \vdash \gamma \;\Downarrow (v,\ \gamma') \qquad \gamma' \;\in\; \beta'}{\Psi \vdash \textsc{Groupc}(\beta,\ \gamma) \;\Downarrow (v,\ \textsc{Groupc}(\beta',\ \gamma'))}$$

(γ-**nonSamplePeriod**):

$$\frac{\Psi[4] \;=\; \texttt{false}}{\Psi \vdash \gamma \;\Downarrow (\texttt{None}, \gamma)}$$

Continuous signals are updated during sample periods. Thus, all the rules require that the fourth component of the update environment is set to true ($\Psi[4] \;=\; \texttt{true}$). If the current update period is not also a sampling period then rule (γ-**nonSamplePeriod**) applies.

The integral rule (γ-**integral**) integrates the value of a continuous signal with respect to time. Informally, it works as follows:

Given the last state x, the delta time dt, and the speed dx : **`return`** x + (dt * dx)

Formally, the speed is given by a single input signal (γ), the past state is stored in its current value (v). The rule uses the equation above to calculate the integral given the delta time for a update period ($\Psi[2]$).

Signal collections can be either static or dynamic. A static collections is evaluated by rule (γ-**collects**). Each element within the collection is evaluated by the update function ($(fv1)...(fv_n)$) and then the updated values are returned along with the updated collection signal. Dynamic collections are evaluated using rules: (γ-**collectd-Event**) and (γ-**collectd-NoEvent**). For both rules, each element in the collection is evaluated by the element update function (f_1) . If there is an event occurrence from the input signal (κ) then (γ-**collectd-Event**) is used to evaluate the dynamic collection, which also calls the collection update function (f_2) to remove or add elements. Otherwise, rule (γ-**collectd-NoEvent**) is used to evaluate the collection, which does not perform a collection update.

Evaluating a continuous signal value with a pure function ($(f\ v_1)$) is handled by using the (γ-**liftc**) rule. The value of the input signal (γ) is applied to the function (f) to produce a value (v_1'), which is returned with the updated signal. Constants are evaluated using the (γ-**constantc**) which returns the signal value (v). Finally, the (γ-**groupc**) evaluates a continuous signal by assigning its value to a signal that is part of another signal group. The key idea behind the group primitive is to connect signal groups together by flowing their output values into signals inside another group. Thus, (γ-**groupc**) evaluates the signal group (β) and input signal (γ). The evaluation rule requires that the updated input signal (γ') is part of the updated signal group (β'). Thus, only signals defined inside the input signal group are allowed as arguments to the group primitive. The value of the input signal is then returned as the signal's value.

Discrete Signal Evaluation: The evaluation rules for evaluating discrete signals is as follows:

(ω-**foldpd**):

$$\frac{\Psi \vdash \omega \Downarrow (v_1,\ \omega') \qquad (f\ (v, v_1)) \Downarrow v'}{\Psi \vdash \textsc{Foldpd}(v,\ f,\ \omega)\ \Downarrow (v',\ \textsc{Foldpd}(v',\ f,\ \omega'))}$$

(ω-**folde-Event**):

$$\frac{\Psi \vdash \kappa \Downarrow (v_1,\ \kappa') \qquad (f\ (v, v_1)) \Downarrow v'}{\Psi \vdash \textsc{Foldpe}(v,\ f,\ \kappa)\ \Downarrow (v',\ \textsc{Foldpd}(v',\ f,\ \kappa'))}$$

(ω-**folde-NoEvent**):

$$\frac{\Psi \vdash \kappa \Downarrow (\texttt{None},\ \kappa')}{\Psi \vdash \textsc{Foldpe}(v,\ f,\ \kappa)\ \Downarrow (v,\ \textsc{Foldpd}(v,\ f,\ \kappa'))}$$

(ω-**liftd**):

$$\frac{\Psi \vdash \omega \Downarrow (v_1,\ \omega') \qquad (f\ v_1) \Downarrow v'}{\Psi \vdash \textsc{Liftd}(f,\ \omega)\ \Downarrow (v',\ \textsc{Liftd}(f,\ \omega'))}$$

(ω-**constantd**):

$$\frac{}{\Psi \vdash \textsc{Constantd}(v)\ \Downarrow (v,\ \textsc{Constantd}(v))}$$

(ω-**groupd**):

$$\frac{\Psi \vdash \beta\ \Downarrow \beta' \qquad \Psi \vdash \omega\ \Downarrow (v,\ \omega') \qquad \omega'\ \in\ \beta'}{\Psi \vdash \textsc{Groupd}(\beta,\ \omega)\ \Downarrow (v,\ \textsc{Groupd}(\beta',\ \omega'))}$$

The history-sensitive signals that take in a discrete signal are evaluated using the (ω-**foldpd**) rule. First, the input signal is evaluated and produces a value (v_1) and updated state (ω'). This value is used to compute the updated value for the signal by calling the update function (f) with the current value of the signal (v) and the input value (v_1) to produce a new value (v'). The updated value is returned along with the updated internal representation for the discrete signal. The two other history-sensitive rules: (ω-**folde-Event**) and (ω-**folde-NoEvent**) are similar to the first rule with the exception that its input is an event signal. Thus, if there is an event occurrence then rule (ω-**folde-Event**) is used to evaluate the signal. The update function takes in the event value along with the current value of the signal to produce the updated value of the signal. Otherwise,

the (ω-**folde-NoEvent**) rule does not update the signal value. The current value of the signal is returned along with the unchanged internal representation. As with continuous signals, the lifting functions evaluate the discrete signal value using a pure function by using the (ω-**liftd**) rule or the value of the discrete signal remains constant by using the (ω-**constantd**) rule. The evaluation rule (ω-**groupd**) is the same as the (γ-**groupc**) rule discussed above with the difference being that it handles discrete input signals.

Event signal evaluation: The evaluation rules for evaluating event signals is as follows:

(κ-**merge-Event**):

$$\frac{\Psi \vdash \kappa_1 \Downarrow (v_1',\ \kappa_1') \qquad ... \qquad \Psi \vdash \kappa_n \Downarrow (v_n',\ \kappa_n') \qquad (v',\ \kappa') \in ((v_1', \kappa_1'),\ ...,\ (v_n', \kappa_n'))}{\Psi \vdash \textsc{Merge}((\kappa_1,\ ...,\ \kappa_n)) \Downarrow (v',\ \textsc{Merge}((\kappa_1',\ ...,\ \kappa_n')))}$$

(κ-**merge-NoEvent**):

$$\frac{\Psi \vdash \kappa_1 \Downarrow (\texttt{None},\ \kappa_1') \qquad ... \qquad \Psi \vdash \kappa_n \Downarrow (\texttt{None},\ \kappa_n')}{\Psi \vdash \textsc{Merge}((\kappa_1,\ ...,\ \kappa_n)) \Downarrow (\texttt{None},\ \textsc{Merge}((\kappa_1',\ ...,\ \kappa_n')))}$$

(κ-**after - Event**):

$$\frac{\Psi[2] \geq T}{\Psi \vdash \textsc{After}(T,\ \texttt{true}) \Downarrow ((),\ \textsc{After}(T,\ \texttt{false}))}$$

(κ-**after - NoEvent**):

$$\frac{\Psi[2] < T}{\Psi \vdash \textsc{After}(T,\ \texttt{true}) \Downarrow (\texttt{None},\ \textsc{After}(T,\ true))}$$

(κ-**after - Complete**):

$$\frac{}{\Psi \vdash \textsc{After}(T,\ \texttt{false}) \Downarrow (\texttt{None},\ \textsc{After}(T,\ \texttt{false}))}$$

(κ-**lifteEvent**):

$$\frac{\Psi \vdash \kappa \Downarrow (v_1,\ \kappa') \qquad (f\ v_1) \Downarrow v'}{\Psi \vdash \textsc{Lifte}(f,\ \kappa) \Downarrow (v',\ \textsc{Lifte}(f,\ \kappa'))}$$

(κ-**lifteNoEvent**):

$$\frac{\Psi \vdash \kappa \ \Downarrow (\texttt{None},\ \kappa')}{\Psi \vdash \textsc{Lifte}(f,\ \kappa) \ \Downarrow (\texttt{None},\ \textsc{Lifte}(f,\ \kappa'))}$$

(κ-**filterNoEvent**):

$$\frac{\Psi \vdash \kappa \ \Downarrow (\texttt{None},\ \kappa')}{\Psi \vdash \textsc{Filter}(f,\ \kappa) \ \Downarrow (\texttt{None},\ \textsc{Filter}(f,\ \kappa'))}$$

(κ-**filterFalseEvent**):

$$\frac{\Psi \vdash \kappa \ \Downarrow (v_1,\ \kappa') \qquad (f\ v_1) \Downarrow \texttt{false}}{\Psi \vdash \textsc{Filter}(f,\ \kappa) \ \Downarrow (\texttt{None},\ \textsc{Filter}(f,\ \kappa'))}$$

(κ-**filterTrueEvent**):

$$\frac{\Psi \vdash \kappa \ \Downarrow (v_1,\ \kappa') \qquad (f\ v_1) \Downarrow \texttt{true}}{\Psi \vdash \textsc{Filter}(f,\ \kappa) \ \Downarrow (v_1,\ \textsc{Filter}(f,\ \kappa'))}$$

(κ-**groupe**):

$$\frac{\Psi \vdash \beta \ \Downarrow \beta' \qquad \Psi \vdash \kappa \ \Downarrow (v,\ \kappa') \qquad \kappa' \in \beta'}{\Psi \vdash \textsc{Groupe}(\beta,\ \kappa) \ \Downarrow (v,\ \textsc{Groupe}(\beta',\ \kappa'))}$$

(κ-**external-Event**):

$$\frac{(i, v) \in \Psi[1]}{\Psi \vdash \textsc{External}(i) \ \Downarrow (v,\ \textsc{External}(i))}$$

(κ-**external-NoEvent**):

$$\frac{(i, v) \notin \Psi[1]}{\Psi \vdash \textsc{External}(i) \ \Downarrow (\texttt{None},\ \textsc{External}(i))}$$

Each of the event signals have two evaluation rules. One rule evaluates the signals when its input has an event occurrence. The other rule returns the value of **`None`** when its input signal has not changed (*i.e.*, no event occurrence). The (κ-**merge-Event**) rule evaluates all of its input signals. If there is an event occurrence by one of the inputs then it returns that input's value. Otherwise, the evaluation uses the (κ-**merge-NoEvent**) rule, which returns **`None`** if none of the inputs had an event occurrence.

The after primitive fires an event occurrence after a specified amount of time (T) has passed. Thus, we look at the absolute time within the update environment ($\Psi[2]$) to check if enough time

has passed. The internal representation of after contains a boolean property that represents if the event occurrence has happened. We use the (κ-**after - Event**) rule to evaluate the signal when the desired amount of time passed has been reached. We return a unit value as the even occurrence's value and set the boolean property of the internal representation to **`false`**. Any future updating of the signal are handled by the (κ-**after - Complete**) rule, since the core after primitive is not a recurring time event. If the desired amount of time has not been reached then the (κ-**after - NoEvent**) rule is used to evaluate the signal.

As with the other lifting functions, the (κ-**lifteEvent**) rule applies the pure function on an input event occurrence. Again, the evaluation rule (ω-**groupe**) is the same as the other versions discussed with the difference being that it handles event input signals. The (κ-**filterTrueEvent**) rule passes along the the value of a filter signal's input signal (κ) if it has an event occurrence and the provided function (f) returns **`true`** when given the input's value (v_1). Otherwise, the (κ-**filterNoEvent**) rule no value is passed along if the input signal has no occurrence or the update function returns **`false`** using the (κ-**filterFalseEvent**) rule. Finally, the (κ-**external-Event**) and (κ-**external-NoEvent**) rules evaluate external event signals. The (κ-**external-Event**) rule is applied if the external identifier (i) is within the tuple of event occurrences for the current update period ($Psi[1]$). Otherwise, the (κ-**external-NoEvent**) rule is used to signify no event occurrence for the external event signal.

Signal Group Evaluation: The evaluation of signal groups is simply evaluating the all signals within the group. The updated signals are then returned to represent the updated state for the signal group as shown in rule (β-**eval**).

(β-**eval**):

$$\frac{\beta \;=\; (\alpha_0,\; ...,\; \alpha_n) \qquad \Psi \vdash \alpha_1 \Downarrow (v_1',\; \alpha_1') \qquad ... \qquad \Psi \vdash \alpha_n \Downarrow (v_n',\; \alpha_n')}{\Psi \vdash \beta \;\Downarrow (\alpha_1',\; ...,\; \alpha_n')}$$

Program evaluation: The evaluation of a program involves two rules: (χ-**updatePeriod**) and (p-**eval**):

(χ-**updatePeriod**):

$$\frac{dt \;=\; T - \Psi[2] \qquad \Psi' \;=\; (((i_1,\; v_1),\; ...,\; (i_n,\; v_n)),\; T,\; dt, c) \qquad \Psi' \vdash \beta \Downarrow \beta'}{(\Psi, \text{upPeriod}(\beta,\; (((i_1,\; v_1),\; ...,\; (i_n,\; v_n)),\; c,\; T))) \Downarrow (\Psi', \beta')}$$

(p-**eval**):

$$\frac{\begin{array}{c} e \Downarrow \beta_0 \qquad ((\emptyset,\; 0,\; 0,\; \texttt{true}),\; \text{upPeriod}(\beta_1,\; ((),\; \texttt{true}, 0))) \Downarrow (\Psi_1,\; \beta_1) \\ ((\Psi_1,\; \text{upPeriod}(\beta_1,\; \psi_1)) \Downarrow (\Psi_2, \beta_2) \qquad ... \qquad (\Psi_{n-1},\; \text{upPeriod}(\beta_{n-1},\; \psi_n)) \Downarrow (\Psi'_n,\; \beta_n) \end{array}}{\Psi \vdash (e,\; (\psi_1,\; ...,\; \psi_n)) \Downarrow ()}$$

In the (p-**eval**) rule, the first step is to evaluate the program expression (e), which produces the initial signal group (β_0). The next step is the *initialization* step that initializes the signal group's signals at $T = 0$. The evaluation of a program involves evaluating the update periods by iteratively passing along the updated signal group and update period environment into the next update period. This process is done until there are no more update periods (*i.e.*, the evaluation system has reached ψ_n). The evaluation of the (χ-**updatePeriod**) rule produces the new signal group and update period environment after each update period. The evaluation of an update period requires passing in the prior update period environment (Ψ) and using the environment to calculate the delta-time ($dt \;=\; T - \Psi[2]$, where $\Psi[2]$ is the last update period time). A new update environment is created (Ψ') that is used to evaluate the signal group (β) to produce the update signal group (β'). The environment and signal group are returned so that they can be used in the next update period.

5.4 Static Semantics

In this section, we present the type system for Mini-**Tesel**. We present the following sets for type constants and types.

$$\begin{array}{rcll} \iota & \in & \textsc{TypeCcon} \;=\; \{\texttt{Int},\; \texttt{Bool},\; \texttt{Time},\; ...\} & \text{type constants} \\ \tau & \in & \textsc{Type} & \text{types} \\ \sigma & \in & \textsc{SignalType} =\subset \textsc{Type} & \text{signal types} \end{array}$$

and with the set of types defined by the following grammar:

$$
\begin{array}{rcll}
\tau & ::= & \iota & \text{type constants} \\
 & | & \varphi & \text{type variables} \\
 & | & \tau_1 \ \rightarrow \ \tau_2 & \text{function types} \\
 & | & (\tau_1, \ ..., \ \tau_n) & \text{tuple types} \\
 & | & \sigma & \text{signal type} \\
 & | & \texttt{SignalGroup} & \text{signal group type} \\
 & & & \\
\sigma & ::= & \texttt{DSignal} \ \langle \tau \rangle & \text{discrete signal type} \\
 & | & \texttt{CSignal} \ \langle \tau \rangle & \text{continuous signal type} \\
 & | & \texttt{ESignal} \ \langle \tau \rangle & \text{event signal type}
\end{array}
$$

The Tesel typing judgments are written with respect to a static environment Γ, which represents a variable environment:

$$\Gamma \ \in \ \textsc{Env} = \textsc{Var} \xrightarrow{\text{fin}} \textsc{Type} \quad \text{typing environment}$$

We assume this environment is filled by the semantics of Swift's declaration syntax and instead focus on the specific semantics for Tesel types. The judgement forms and their meanings are defined in Table 5.1.

Table 5.1: Judgement forms and their meanings

Judgement Form	Meaning
$\Gamma \vdash \ p : \ \Gamma$	Program p has type τ.
$\Gamma \vdash \ e : \ \tau$	Expression e has type τ.

5.4.1 Typing Rules

We assume the existence of functions that associate types with constants and external signal identifiers:

$$TypeOf \quad : \quad \textsc{Const} \mapsto \textsc{Type}$$

$$TypeOfExtern \quad : \quad \textsc{ExternalSigIdentifier} \mapsto \textsc{Type}$$

The typing rules for surface language expressions are given in Figure 5.1 and Figure 5.2. We separate out the rules for the primitive functions based on the type of signal the function produces. Figure 5.3 shows the typing rules for event primitives, Figure 5.4 displays the discrete primitives, and Figure 5.5 shows the continuous primitives.

(τ-**variable**):

$$\frac{x \in dom(\Gamma) \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

(τ-**let**):

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \pm \{x \mapsto \tau\} \vdash e_2 : \tau'}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau'}$$

(τ-**constant**):

$$\frac{TypeOf(c) = \tau}{\Gamma \vdash c : \tau}$$

(τ-**optSome**):

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{Some}(e) : \texttt{Optional}\langle\tau\rangle}$$

(τ-**optNone**):

$$\frac{}{\Gamma \vdash \texttt{None} : \texttt{Optional}\langle\varphi\rangle}$$

(τ-**tupleSelection**):

$$\frac{\Gamma \vdash e : (\tau, ..., \tau_n)) \qquad TypeOf(c) = \tau = \texttt{Int} \qquad \tau_c \in (\tau_1, ..., \tau_n)}{\Gamma \vdash e.c : \tau_c}$$

(τ-**externalIdentifier**):

$$\frac{TypeOfExtern(n) = \tau}{\Gamma \vdash n : \tau}$$

(τ-**tuple**):

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad ... \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash (e_1, ..., e_n) : (\tau_1, ..., \tau_n)}$$

Figure 5.1: The typing rules for surface expressions in Mini-**Tesel**

(τ-**if**):

$$\frac{\Gamma \vdash e_1 \ : \ \texttt{Bool} \qquad \Gamma \vdash e_2 \ : \ \tau \qquad \Gamma \vdash e_3 \ : \ \tau}{\Gamma \vdash \ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \ : \ \tau}$$

(τ-**functionApp**):

$$\frac{\Gamma \vdash e_1 \ : \ \tau \rightarrow \ \tau' \qquad \Gamma \vdash e_2 \ : \ \tau'}{\Gamma \vdash \ (e_1\ e_2) \ : \tau'}$$

(τ-**functionAbs**):

$$\frac{\Gamma \pm \{x \ \mapsto \ \tau\} \vdash e \ : \ \tau'}{\Gamma \vdash \ \texttt{fun}(x) => e \ : \ \tau \ \rightarrow \ \tau'}$$

(τ-**program**):

$$\frac{\Gamma \vdash e \ : \ \tau \qquad \Gamma \vdash (\psi_1, \ ..., \ \psi_n) \ : \ (\tau_1, \ ..., \ \tau_n)}{\Gamma \vdash (e, (\psi_1, \ ..., \psi_n)) : \texttt{Unit}}$$

Figure 5.2: The typing rules for surface expressions in Mini-**Tesel**(cont'd.)

(τ-**after**):

$$\frac{\Gamma \vdash\ e\ :\ \texttt{Time}}{\Gamma \vdash \text{after}(e)\ :\ \texttt{ESignal}\langle\texttt{Time}\rangle}$$

(τ-**merge**):

$$\frac{\Gamma \vdash\ e\ :\ (\texttt{ESignal}\langle\tau\rangle_0,\ ...,\ \texttt{ESignal}\langle\tau\rangle_n)}{\Gamma \vdash \text{merge}(e)\ :\ \texttt{ESignal}\langle\tau\rangle}$$

(τ-**lifte**):

$$\frac{\Gamma \vdash\ e_1\ :\ \tau \to \tau' \qquad \Gamma \vdash\ e_2\ :\ \texttt{ESignal}\langle\tau\rangle}{\Gamma \vdash \text{lifte}(e_1, e_2)\ :\ \texttt{ESignal}\langle\tau'\rangle}$$

(τ-**filter**):

$$\frac{\Gamma \vdash\ e_1\ :\ \tau \to \texttt{Bool} \qquad \Gamma \vdash\ e_2\ :\ \texttt{ESignal}\langle\tau\rangle}{\Gamma \vdash \text{filter}(e_1, e_2)\ :\ \texttt{ESignal}\langle\tau\rangle}$$

(τ-**groupe**):

$$\frac{\Gamma \vdash\ e_1\ :\ \texttt{SignalGroup} \qquad \Gamma \vdash\ e_2\ :\ \texttt{ESignal}\langle\tau\rangle}{\Gamma \vdash \text{groupe}(e_1, e_2)\ :\ \texttt{ESignal}\langle\tau\rangle}$$

(τ-**external**):

$$\frac{TypeOfExtern(n)\ =\ \tau}{\Gamma \vdash\ \text{external}(n)\ :\ \texttt{ESignal}\langle\tau\rangle}$$

Figure 5.3: The typing rules for event primitives in Mini-**Tesel**

(τ-**foldpD**):

$$\frac{\Gamma \vdash\ e_1\ :\ \tau \qquad \Gamma \vdash\ e_2\ :\ \tau \to \tau' \to \tau \qquad \Gamma \vdash\ e_3\ :\ \texttt{DSignal}\langle\tau'\rangle}{\Gamma \vdash \text{foldpD}(e_1, e_2, e_3)\ :\ \texttt{DSignal}\langle\tau\rangle}$$

(τ-**foldpE**):

$$\frac{\Gamma \vdash\ e_1\ :\ \tau \qquad \Gamma \vdash\ e_2\ :\ \tau \to \tau' \to \tau \qquad \Gamma \vdash\ e_3\ :\ \texttt{ESignal}\langle\tau'\rangle}{\Gamma \vdash \text{foldpE}(e_1, e_2, e_3)\ :\ \texttt{DSignal}\langle\tau\rangle}$$

(τ-**liftd**):

$$\frac{\Gamma \vdash\ e_1\ :\ \tau \to \tau' \qquad \Gamma \vdash\ e_2\ :\ \texttt{DSignal}\langle\tau\rangle}{\Gamma \vdash \text{liftd}(e_1, e_2)\ :\ \texttt{DSignal}\langle\tau'\rangle}$$

(τ-**constantd**):

$$\frac{\Gamma \vdash\ e_1\ :\ \tau}{\Gamma \vdash \text{constantd}(e_1)\ :\ \texttt{DSignal}\langle\tau\rangle}$$

(τ-**groupd**):

$$\frac{\Gamma \vdash\ e_1\ :\ \texttt{SignalGroup} \qquad \Gamma \vdash\ e_2\ :\ \texttt{DSignal}\langle\tau\rangle}{\Gamma \vdash \text{groupd}(e_1, e_2)\ :\ \texttt{DSignal}\langle\tau\rangle}$$

Figure 5.4: The typing rules for discrete primitives in Mini-**Tesel**

(τ-**integral**):

$$\frac{\Gamma \vdash\ e\ :\ \texttt{CSignal}\langle\tau\rangle}{\Gamma \vdash \text{integral}(e)\ :\ \texttt{CSignal}\langle\tau\rangle}$$

(τ-**collects**):

$$\frac{\Gamma \vdash\ e_1\ :\ (\tau_0,\ ...,\ \tau_n) \qquad \Gamma \vdash\ e_2\ :\ \tau \to (\tau_0,\ ...,\ \tau_n) \to \tau}{\Gamma \vdash \text{collects}(e_1, e_2)\ :\ \texttt{CSignal}\langle(\tau_0,\ ...,\ \tau_n)\rangle}$$

(τ-**collectd**):

$$\frac{\begin{array}{c}\Gamma \vdash\ e_1\ :\ (\tau_0,\ ...,\ \tau_n) \qquad \Gamma \vdash\ e_2\ :\ \tau \to (\tau_0,\ ...,\ \tau_n) \to \tau \\ \Gamma \vdash\ e_3\ :\ (\tau_0,\ ...,\ \tau_n) \to \tau' \to (\tau_0,\ ...,\ \tau_n) \qquad \Gamma \vdash\ e_4\ :\ \texttt{ESignal}\langle\tau'\rangle\end{array}}{\Gamma \vdash \text{collectd}(e_1, e_2, e_3, e_4)\ :\ \texttt{CSignal}\langle(\tau_0,\ ...,\ \tau_n)\rangle}$$

(τ-**liftc**):

$$\frac{\Gamma \vdash\ e_1\ :\ \tau \to \tau' \qquad \Gamma \vdash\ e_2\ :\ \texttt{CSignal}\langle\tau\rangle}{\Gamma \vdash \text{liftc}(e_1, e_2)\ :\ \texttt{CSignal}\langle\tau'\rangle}$$

(τ-**constantc**):

$$\frac{\Gamma \vdash\ e_1\ :\ \tau}{\Gamma \vdash \text{constantc}(e_1)\ :\ \texttt{CSignal}\langle\tau\rangle}$$

(τ-**groupc**):

$$\frac{\Gamma \vdash\ e_1\ :\ \texttt{SignalGroup} \qquad \Gamma \vdash\ e_2\ :\ \texttt{DSignal}\langle\tau\rangle}{\Gamma \vdash \text{groupc}(e_1, e_2)\ :\ \texttt{CSignal}\langle\tau\rangle}$$

Figure 5.5: The typing rules for continuous primitives in Mini-**Tesel**

CHAPTER 6

IMPLEMENTATION

In this chapter, we discuss the prior FRP implementations and their influence on Tesel's implementation. We also describe our compiler and the process of building a standalone Tesel application. Finally, we provide a description of our runtime system and its various subsystems.

6.1 FRP Implementations

CFRP and AFRP languages primarily use either a push-based or pull-based implementation [31]. Pull-based approaches sampled signals over a series of discrete time-steps. This evaluation model works well for evaluating continuous signals because they change often. Elliot [29] describes several approaches that was used to evaluate continuous signals in the original CFRP language of Fran [30] and later variants of CFRP [74]. Each of these approaches represented events as lazy lists of time-stamped values (*i.e.*, occurrences of the event). One approach represented signals as a function from time to a value and sampled signals at increasing time steps. This approach suffered from the space-time leaks discussed in section 2.2. Two other approaches presented by Elliot made use of the observation that sampling times were monotonically increasing. Thus, one approach represented signals functions from a time to a value at that time and a "residual behavior" (*i.e.*, a continuation) that contained information from prior sampling periods. Anther approach used synchronized streaming processors to represent behaviors as a function from time-streams to value streams, where a stream can be represented a lazy list [39]. Both these representations suffered from long sampling periods owing to the amount of time needed to construct signals within the representation. All approaches presented by Elliot suffered from a more significant problem of redundant sampling. All the representations presented there is no notion of sharing signals within an expression; therefore, if a signal is used multiple times in an expression then it sampled multiple times. One technique of fixing this problem is to use some form of memoization.

Many CFRP languages have adopted a pure push-based implementation [75, 18, 47, 19, 52]. In these languages, evaluation is performed at every event occurrence instead of repeatedly sampling output. Thus, signals are constructed using various combinators provided by the language, which results in constructing a dependency graphs. At every event occurrence, the value of the event is pushed as input into the dependency graph. Signals are then recomputed as updated values flow-through the graph to produce an output.

An important aspect of a push-based implementation is to avoid wasteful recomputations of signal values. If a signal is executed before its dependents then this leads to an invalid program state and requires a recomputation of the signal after the dependents execute. Push-based implementations try to avoid this problem by topologically sorting the dependency graph such that a signal is always evaluated after all of its dependents have executed [18]. Additionally, a change propagation [4] optimization technique should be used to avoid recomputations of signal values that do not change. In other words, if the input values to the signal do not change then the signal value does not need to be recomputed. In particular, FrTime [18] uses change propagation to check at run-time if updated value is the same as the old value and if so then notifies dependents of the unchanged value.

In AFRP, signal functions are represented as *transition functions* within the implementation of the system. For each sampling period, the transition function is given the time since the previous time step (*i.e.*, the delta time) and an input value. The result of the transition function is the output for the signal function at the current time step and a continuation representing the new state of signal function that is used in the next time step. This representation suffers from wasteful recomputations since all signal functions are recomputed on every new event occurrence, even if the event occurrence has no affect on a signal function.

6.2 Implementation Overview

As with other implementations of the FRP hybrid model [31, 47, 68], Tesel uses a push-based implementation for evaluating our discrete signal types (`ESignal` and `DSignal`) and pull-based implementation for evaluating continuous signals (`CSignal`). Both evaluation implementations update the signals by pushing sampling events (*i.e.*, requesting the sampling of the continuous signals) or external events through a dependency graph, which automatically updates the signals as values propagate through the graph. Tesel constructs two types of dependency graphs: a local signal graph that is internal to a signal group, and a signal group graph, where the local signal graphs are connected to each other. The primitive functions defined within the `SignalGroup` class internally construct the local signal graphs. The `group` functions connect the internal local graphs together. The signal group returned by the Tesel main function represents the root of the signal group graph, which is used to determine the executing order when updating the entire graph.

Figure 6.1 shows the overall process of updating signals as external events arise from the outside environment. The `TeselViewController` is the user interface view controller that receives callbacks from its view when the user clicks on the mouse or a key. For sampling continuous signals, a display callback function is used to synchronize the sampling rate to the refresh rate of the display. On each event occurrence, the callback functions notify the `TeselDispatcher` to enqueue the callback function's event into its event queue. This occurrence causes the dispatcher to enqueue an event-queue process task into its *dispatch queue*. The dispatcher's dispatch queue is an Grand Central Dispatch (GCD) queue [7]. GCD is a thread management system that handles executing asynchronous tasks using threads. The programmer specifies a task and submits it to a dispatch queue. GCD is in charge of creating the threads and scheduling the execution of tasks within the dispatch queue. Since execution is handled by the operating system, it can efficiently and quickly execute tasks based on the current workload of the system. Thus, GCD will responds to an event-queue process task by launching a thread and dequeuing all the events within the event queue. Thus, both sample events and external events can be processed if they occur at the same time. All signals are

notified which external event signals has an occurrence and thenthe `SignalGroupController`'s `update` function is called to update all the signal groups within the graph. Only one thread can update the signal group at a time; therefore, all other threads needing to process events wait until the current thread updating the signal graphs has completed.

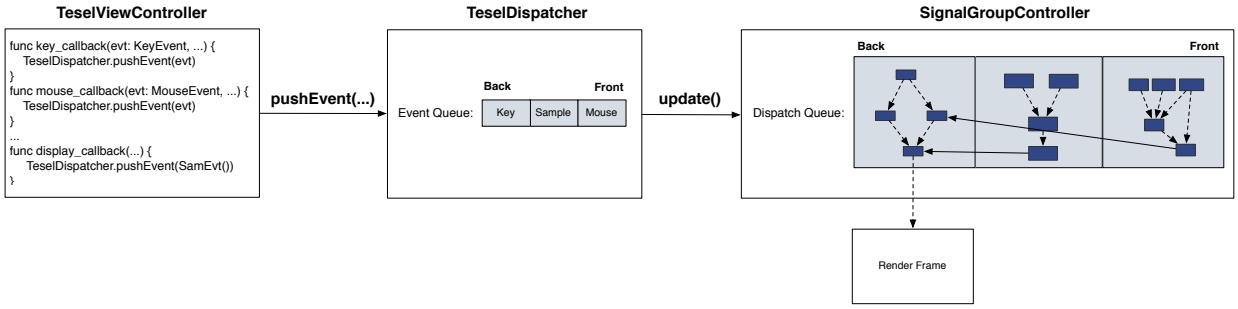
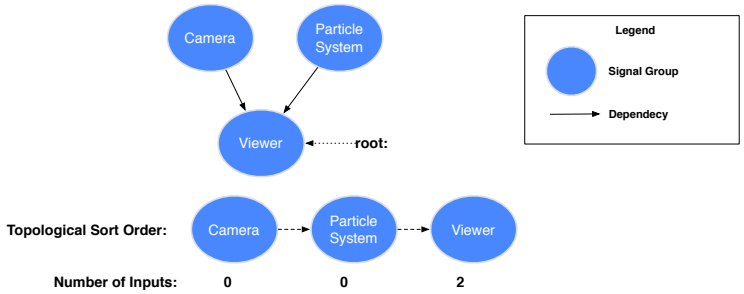


Figure 6.1: An overview of the implementation of Tesel.

The `SignalGroupController` updates the signal groups by first determining the update order by topologically sorting the graph. For example, the dependency group below shows the dependencies between three signal groups:



Nodes in the graph are represented as signal groups. Each child node represents a signal group that is embedded within its parent node and creates a dependency requirement: each child node is updated before its parent. In this case, updating is done on the Camera and ParticleSystem groups before the View group. The topological determines this order by using a modified version of a depth-first search. If concurrent execution is enabled then the sorted signal groups are enqueued into a concurrent dispatch queue. Otherwise, the sorted signal groups are updated in sequential order. Updating a signal group means updating the local signal network within the group. As with

other implementations, the local network is also topologically sorted and executed sequentially. Usually the output of the last signal group that is updated is a frame that is sent to the rendering system to be displayed in the window.

6.3 The Tesel Framework

Every Tesel application is required to import the Tesel framework:

```
import Tesel
```

The framework contains the implementations of the language abstractions (*i.e.*, signals and signal groups) and constructs. In this section, we describe the language implementations for signals and signal groups. We also describe how the primitive functions are used to define instances of signals and signal groups.

6.3.1 Implementing Signals

All signal types in Tesel inherit from the base class, `Signal`:

```
1  public ESignal<O> : Signal<O> { ... }
2  public DSignal<O> : Signal<O> {...}
3  public CSignal<O> : Signal<O>{...}
4  // etc...
```

Within the framework, the `Signal` class conforms to the internal `SignalType` protocol, which defines a group of properties and methods for internally working with signals of any type (*e.g.*, `DSignal<Float>`, `CSignal<Int>`, `ESignal<String>`, *etc.*). In particular, it defines a method for adding dependents (*i.e.*, signals needing the value of another signal to update their value) to a signal. The `SignalType` protocol also contains additional properties and methods for updating a signal, which section 6.5.2 discusses in further detail.

```
internal protocol SignalType  {
    //Adds a signal to the dependents collection
    func addDependency(dependent : SignalType)
    //...
}
```

Along with implementing the required properties and methods of the `SignalType` protocol, the `Signal` also defines a set of core properties as follows:

```
public class Signal<O> : SignalType  {

        // Represents the current internal value of a signal
        private var inVal : O?

        // Represents the number of input signals for this signal's update function
        internal var numOfInputs : Int  = 0

        // Represents the  collection of signals that are notified
        // when this signal has completed iits sampling process
        internal var dependents : [SignalType] = []

        // The update function for this signal
        private var _upFunc : (val : Signal<O>?) ->  O?

        //...
}
```

The signal value is an optional value since some signals (*e.g.*, `ESignal`) do not always contain a value. The class also contains properties for the number of input signals (`numOfInputs`) that an instance depends on before updating its value and a collection of dependent signals (`dependents`) that are notified once the signal has completed its updating process. Lastly, a signal contains a property (`_upFunc`) that handles running the signal's update function.

The primtive functions (*i.e.*, `integral`, `foldp`, `map`, *etc.*.) are used to return instances of the signal types. The primary purpose for each combinator is to initialize the signal with an *auxiliary* update function and initial value (if applicable). The primitive functions also add dependency connections

between the inputs to the update function and the signal. For example, the implementation of the lift primitive for event signals is as follows:

```
1  public func lift<I1,O> (fn : I1 -> O, in1 : ESignal<I1>) -> ESignal<O> {
2
3      func updateFun(val:O?) -> O? {
4          return fn(in1.value)
5      }
6      let signal = ESignal<O>(upFunc:updateFun, initVal:nil, inputs:1)
7      in1.addDependency(signal)
8      return signal
9  }
```

Lines 3–5 creates an auxiliary update function that handles the running of the update function provided to the combinator. Lines 6 creates an instance of a signal given the update function and the initial value of the signal. In this case, the map primitive does not provide an initial value. Finally, Line 7 calls the `addDependency` function to notify the input signal that the new signal depends on its updated values and Line 8 returns the new signal.

6.3.2 Implementing Signal Groups

A signal group is added to the runtime system for updating either through the group primitive or by being the root signal group of the program (*i.e.*, the signal group returned by the Tesel main function). As with the `Signal` class, the signal group class conforms to an internal protocol, namely `SignalGroupType`, which requires each signal group to provide methods and properties to add signals and other signal groups to the signal group.

```
internal protocol SignalGroupType {

    // Adds a dependency between the internal signal group and the provided signal group
    func addDependency(dependent : SignalGroupType)

    //Represents the collection of signals within the group
    var signals : [SignalType] {get}

    // Adds an internal signal group
    func addGroup  (group : InternalSignalGroupType)

    //...
}
```

The core properties of the internal signal group, `SignalGroup`, is defined as follows:

```
internal class InternalSignalGroup<T : SignalGroup> : InternalSignalGroupType {

    //The actual signal group that is associated with this internal signal group
    internal var group : T

    //Represents the collection of signals within the group
    internal var signals : [SignalType] = []

    //Represents all the embedded signal groups within this signal group
    internal var groups  : [InternalSignalGroupType] = []

    //Represents all signals that need to be notified when this signal group
    //is done with its sampling process.
    internal var dependents : [SignalType] = []

    //...
}
```

A `SignalGroup` maintains a collection of the signals (`signals`) and the embedded groups (`groups`) within the group. The runtime system uses the groups property during an update step to determine the execution order. In particular, the runtime system uses a topological sort to ensure independent signal groups are executed before dependent signal groups. This requirement avoids the possibility

of deadlock. The `dependents` property represents the signal groups that depend on being notified once the signal group has completed its update step. This notification is important in the parallel execution of the groups to allow the thread that is updating a signal group to continue its execution.

6.4 Compiler Overview

As shown in Figure 6.2, the Tesel compiler is a multi-pass compiler written in Standard ML. The compiler's main jobs are to type-check the source files, generate code needed by the runtime system's rendering system, and link together the runtime system and source files to build an executable application. In this section, we describe our compilation process and how a standalone application is generated.

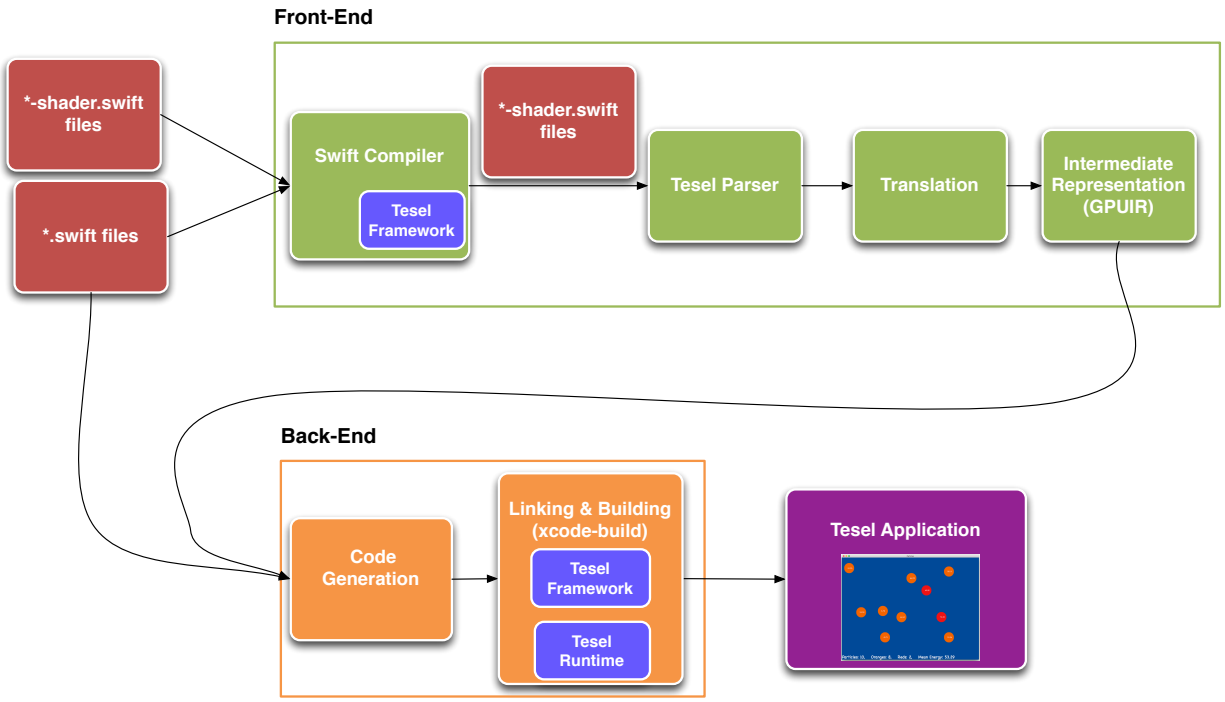


Figure 6.2: An overview of the Tesel compiler.

6.4.1 Front-End Phases

A Tesel program is required to be separated into two types of files: regular Swift files ("*.swift") and GPU-based files ("*-shader.swift"). The GPU-based files consists of the shader program definitions and the struct definitions that are used to represent data used in the shader functions (*e.g.*`Attributes`, `Uniforms`, `Textures`, *etc.*). The front-end consists of type checking, parsing the GPU-based files, and translating the parse-tree into our intermediate representation. The Swift compiler is used to perform syntax and type checking. The compiler feeds the source files along with our framework to the Swift compiler. After syntax and type checking, the GPU-based files are parsed into a parse tree. We parse the GPU-based files for two reasons. First, we need to know about the sizes of the data structures that the GPU needs for rendering (*e.g.*, mesh data, uniforms, textures). Second, the shader programs are transformed from their Swift representations to the shading language used by the rendering system (*i.e.*, the Metal shading language). The final phase in the front-end is translation, where we convert the parse-tree into the intermediate representation as shown below. The IR represents a collection of shader programs with additional collections for struct definitions that are used on the GPU. Although the compiler does not perform any lowering techniques on the IR, we potentially have the opportunity for performing them in the future. For example, converting distinct Swift mechanisms into a form recognizable by the shading language such as transforming the Swift for-each statement into an iterative for-loop. Thus, programmers can use Swift-specific features and allow the compiler to handle the conversion easily.

```
structure GPUIR : sig

    type var = Atom.atom
    datatype stmt = datatype ParseTree.stmt
    datatype unary = datatype ParseTree.unary
    datatype expr = datatype ParseTree.expr
    datatype ty = datatype Types.ty

    datatype gpuIR
       = IR of {
```

```
      program    : shader list,
      vertexInfo : struct_decl,
      fragInfo   : struct_decl,
      uniforms   : struct_decl AtomTable.hash_table,
      attributes : struct_decl AtomTable.hash_table,
      textures   : struct_decl AtomTable.hash_table,
      samplers   : struct_decl AtomTable.hash_table,
      structs    : struct_decl AtomTable.hash_table,
      globals    : var_decl AtomTable.hash_table,
      funcs      : func AtomTable.hash_table
   }
and  shader =
   P_Shader of {
     name    : var,
     structs : struct_decl list,
     globals : var_decl list,
     funcs   : func list,
     vFunc   : func,
     fFunc   : func
  }

and func
  = F_Func of {     (* function decl *)
      name: var,
      params: (ty * var) list,
      retTy : ty,
      body: stmt
  }

and var_decl
  = VD_Decl of {
      isConstant : bool,
      name : ParseTree.var,
      ty : ty,
      expr : expr option
    }
and struct_decl
  = SD_Struct of {
     name : var,
     properties : var_decl list
```

```
        }
    end
```

6.4.2 Back-End Phases

The first phase of the back-end is code generation. The main purpose of the code generation phase generate code for both Metal API and the Swift from the IR. Each shader program is contained within a single Metal file along with the structs needed by that shader program as shown in the Figure 6.3 and Figure 6.4. The main differences between the representations are their syntax and the requirement to add additional qualifiers to the properties within a struct definition and parameters to the shader functions. The Metal shading language is based on C++ [45]. Thus, the main job of this phase is converting the Swift syntax for structs and functions into their C++ versions. The phase also needs to transform the names of the functions embedded within the shader program struct definition. Struct definitions in the Metal shading language can only contain fields; therefore, we must extract out the functions defined within the shader program structs. But an error could occur if the names of the functions are the same. For example, defining two shader program structs (*e.g.*, FlatShaders, and TexShaders) with both naming the vertex shader:"vertexFunc". We resolve this issue by appending on the name of the struct definition to the function names (Line 18 and Line 26 of Figure 6.4). The Metal shading language also requires that parameters to the shader functions and structs that represent vertex attributes append qualifiers, which state the location of where the data is stored. For example, the uniform data is stored in an array of buffer objects on the GPU and requires a qualifier to state which buffer object holds the data. We append this qualifier to the uniform data with the buffer index of where the data is stored on Line 18 of Figure 6.4.

```
1  struct FrameUniforms : Uniforms {
2      var model          : float4x4;
3      var projectionView : float4x4;
4      var color          : float4;
5  }
6  struct Vertex : Attributes  {
7      var pos : float3;
8  };
9  public struct FlatShaders : ShaderProgram {
10
11     public var vertFuncName   : String  = "vertexFunc";
12     public var fragFuncName   : String  = "fragFunc";
13
14     struct ProjectedVertex : FragInput
15     {
16         var position : float4;
17     }
18     func vertexFunc(vInfo: VertexInfo,
19                     vert: Vertex,
20                     uniforms: FrameUniforms) -> ProjectedVertex {
21
22         let clipCoord : float4 = uniforms.projectionView *
23                                  uniforms.model * float4(vert.pos, 1.0);
24         return  ProjectedVertex(position:clipCoord);
25     }
26     func fragFunc(fInfo: FragInfo,
27                   pVertex: ProjectedVertex,
28                   uniforms: FrameUniforms) -> float4
29     {
30         return uniforms.color;
31     }
32 }
```

Figure 6.3: A flat shading program implemented in Tesel.

```
#include <metal_stdlib>

using namespace metal;

struct FrameUniforms {

    float4x4 model;
    float4x4 projectionView;
    float4 color;

};
struct Vertex {
    float3 pos[[attribute(0)]];
};
struct ProjectedVertex {
    float4 position[[position]];
};
vertex ProjectedVertex FlatShaders_vertexFunc(constant FrameUniforms &uniforms [[buffer(1)]],
                                              TeselDefaultAtrributes vert [[stage_in]],
                                              uint vInfo_instance_id [[instance_id]],
                                              uint vInfo_vertex_id [[vertex_id]])
{
    float4 clipCoord = uniforms.projectionView*uniforms.model*float4(vert.pos,  0.1e1);
    return ProjectedVertex{clipCoord};
}
fragment float4 FlatShaders_fragFunc(constant FrameUniforms &uniforms [[buffer(1)]],
                                     ProjectedVertex pVertex [[stage_in]])
{
    return uniforms.color;
}
```

Figure 6.4: The equivalent Metal code of the shader program in Figure 6.3.

The code-generation phase also generates one Swift file that contains `extension` declarations for all the GPU-based struct definitions. Each extension includes functions and properties that provide additional information needed by the low-level API to move the struct data from the CPU to GPU. For example, the code generation phase generates an extension for an attributes struct as follows:

```
struct FlatAttributes : Attributes {
    var pos           : float3;
    var tex           : float2;
}
```

(a) Attributes definition defined in within a Swift file.

```
extension FlatAttributes {

    var alignments : [Int]{
        return [alignment(pos), alignment(tex)]
    }
    static var formatWithSizes : [(MTLVertexFormat,Int)]{
        return [(.Float3, sizeof(float3)),
                (.Float2, sizeof(float2))]
    }
    static var size : Int{
        return sizeof(float3)+(sizeof(float2)+0)
    }

    func loadBuffer(var bufferPointer:UnsafeMutablePointer<Void>)
                        -> UnsafeMutablePointer<Void>
    {
        memcpy(bufferPointer, self.pos.rawValue, sizeof(float3))
        bufferPointer += sizeof(float3)
        memcpy(bufferPointer, self.tex.rawValue, sizeof(float2))
        bufferPointer += sizeof(float2)
        return bufferPointer
    }
}
```

(b) Generated attributes extension that includes information about its structure and a function for loading the data into a buffer.

Figure 6.5: An overview of generating the size information for Attributes.

The extension includes alignment sizes for fields, struct sizes, and field sizes. The extension also contains a function for loading the data of the struct into a Metal buffer to be used on the GPU. The rendering system calls these properties and the loadBuffer function to correctly initialize GPU bound buffers that contain attributes and to load the data from these attributes into the buffer.

The final phase of the back-end builds the executable. The building of the executable involves linking together three parts: the runtime system, which includes the Tesel framework, the original

program source files, the generated Swift files with the extensions, and the Metal files. We use the *xcode-build* command line tool to build and link all these parts together to produce the final runnable application.

6.5 Runtime Support

The Tesel runtime contains code that creates and manages a windowing system for displaying frames, handles external events (*e.g.*, propagating new key and mouse events to signals), periodically updating the signal groups and signals, and implements a rendering a system that handles the low-level graphics details for rendering frames. The system contains four main parts:

- **TeselDipsatcher**: The handles receiving event occurrences from the `TeselViewController` and `TeselViewRenderer`. Event occurrences are stored inside an event queue where the thread dispatching system processes them by dispatching a thread to dequeue an event occurrence and notify the `SignalGroupController` to begin an update cycle based off this event occurrence.

- **TeselViewController**: All interactions between the user and application is handled by the `TeselViewController`. This class contains callback methods for handling event occurrences that are initiated by the user (*e.g.*, clicking on a key or the mouse). The events are then dispatched to the `TeselDispatcher` for further processing.

- **SignalGroupController**: Updating the signals and signal groups in the application is managed by the `SignalGroupController`. The `TeselDipsatcher` notifies the controller on each an event occurrence. This notification begins an update cycle, which updates all the signals within the signal groups. The controller also handles executing the elements of the dynamic collections and signal groups in parallel if concurrency is enabled.

- **TeselViewRenderer**: The `TeselViewRenderer` handles all the rendering of frames that are requested by the application. The renderer maintains a thread-safe frame queue that allows

frames to be inserted for rendering. Each time a frame is inserted, the renderer notifies the main user interface queue to process the frame for rendering. All aspects of setting up the Metal pipeline state for rendering a frame is handled within this controller.

Figure 6.6 shows the runtime lifecycle of a Tesel program. In this section, we provide details about how a program transitions through this lifecycle and how the various components of the runtime help with program execution.

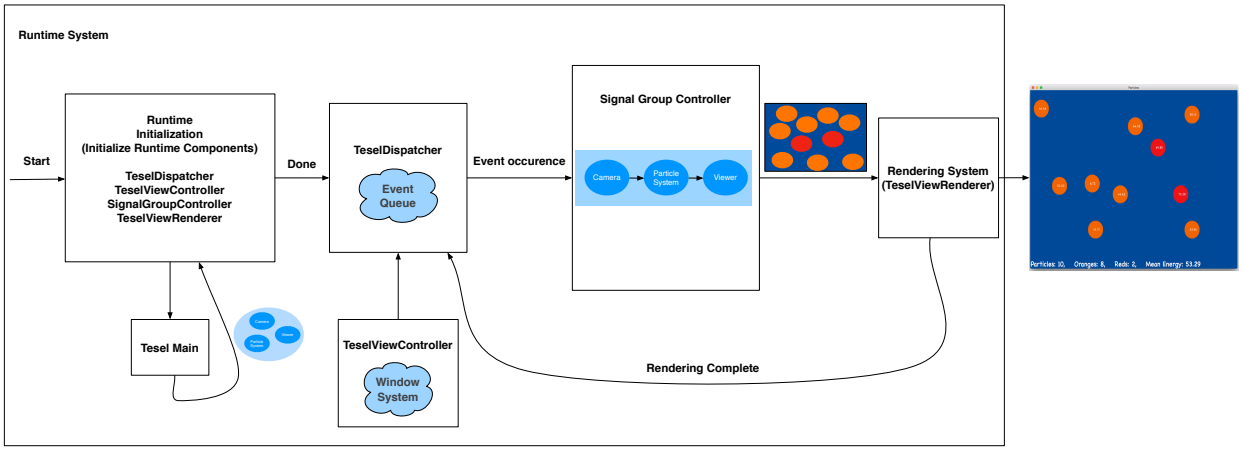


Figure 6.6: An overview of the runtime lifecycle for a Tesel program.

6.5.1 Runtime Initialization

The first phase of the runtime lifecycle handles initializing the inner components in the runtime. First, the Tesel main function of the application is invoked to receive the main signal group and the configuration object. The main signal group is given to the `SignalGroupController` and the runtime initializes the `TeselViewController` by providing it with the configuration object. The view controller uses the configuration object to initialize the window's size and title properties. The runtime notifies to the `TeselDipsatcher` to define and initialize its event queue for event processing. The last step of this phase initializes the rendering system (*i.e.*, `TeselViewRenderer`). The system uses the Metal framework as its low-level graphics API. Initializing the rendering system means retrieving and creating the Metal components needed to submit work to the GPU such as retrieving the main

graphics device and creating a command queue to submit rendering tasks. The object that loads textures from the system and the default library that holds the program's metal shaders programs are also created and initialized.

6.5.2 Updating the Application

After the initialization step, the runtime notifies the `TeselDispatcher` that it can begin receiving events to update the application. Events occurrences arrive from two sources: the `TeselViewController` and the `TeselViewRenderer`. Each callback inside the `TeselViewController` notify the dispatcher about user event occurrences. Additionally, sampling events, which cause the `SignalGroupController` to sample the continuous signals is initiated by the view controller. Currently, the sampling rate is tied to the display's refresh rate, which is common to the update rate of Metal-based applications in the event-driven paradigm. In the future, we will allow the user to specify a fixed-rate or variable rate for sampling.

The `TeselDispatcher`'s also maintains a thread dispatch queue, where each dispatch task handles processing the events within the event queue. A thread is spawned by the dispatch managing system (*i.e.*, GCD) begins an update cycle. An update cycle entails processing the events currently in the event queue and notifying the `SignalGroupController` to update the dependency graph. Only one thread can perform an update cycle at a time. All other threads will wait to process future events until the current thread finishes its update cycle. Each type of outside external event (*i.e.*, mouse events, and key events) is assigned an `ESignal` signal that holds the event's value (*e.g.*, `WinEvent`, `KeyEvent`). During event processing, each event inside the queue is dequeued and its corresponding `ESignal` signal is updated with the new event value. Once all events are processed then the updated `ESignal` signals are broadcast to all of their dependent signals within the program (*i.e.*, signals using these event signals as input signals). The last step of the update cycle is calling the `update` function of the `SignalGroupController` to update the dependency graph. The `update` function is defined as follows.

```
1  /** Executes a update cycle for the signal groups **/
2  internal static func update(shouldSample : Bool) {
3      self.isSamplePeriod = shouldSample
4      if var rGroup = rootGroup {
5
6          //Check if the program's dependency graph has changed
7          // (ie., the rootGroup's tree has been modified)
8          if SignalGroupController.shouldSort {
9              //Retrieve the new update order and the collection
10             //signals in the graph
11             let (rGroups, cSignals) = rGroup.computeExecutionOrder()
12             self.runGroups = rGroups
13             self.collectSigs = cSignals
14             shouldSort = false
15         }
16         //Retrive the current time of the update period
17         if let currTime = _timer.time {
18             self.currentTime = currTime
19             //Determine if the signal groups should be executed in parallel
20             if SignalGroupController.isParallel {
21
22                 let global = dispatch_get_global_queue(
23                                                 DISPATCH_QUEUE_PRIORITY_HIGH,
24                                                 0)
25                 let groupQueue = dispatch_group_create()
26
27                 if self.isSamplePeriod {
28                     for signal in self.collectSigs {
29                         signal.runElementFuncGroup(global, group:groupQueue)
30                     }
31                     launchGroupsInPar(global,groupQueue:groupQueue)
32
33                 } else {
34                     launchGroupsInPar(global,groupQueue:groupQueue)
35                 }
36             } else {
37                 //Updates the signal groups in based on sequentially
38                 // iterating over the runGroups collection.
39                 for group in runGroups {
40                     if !group.remainConstant {
```

```
41                              //update the time properties
42                              updateGroupTime(group)
43
44                              //Update the signal group
45                              group.update()
46
47                              //Notify any signal dependents that
48                              //the group has finished updating.
49                              group.notifyDependents()
50                          }
51                      }
52                  }
53              }
54          }
55      }
```

The `update` function takes in a parameter that represents whether this current update should sample the continuous signals. We assign that parameter to the `isSamplePeriod` property (Line 3), which is accessible to the continuous signals when they try to update their value. On Line 4 , we retrieve the main signal group and beings the update process. We determine whether the groups need to be sorted in topological order (Lines 8–15). This sorting process is done initially and only when a dynamic signal group resets itself, since this reset could introduce new signals and signal groups to the graph. The sorting is performed by the `computeExecutionOrder` of the main signal group (Line 11). The `computeExecutionOrder` function returns the sorted groups and all dynamic collections within the groups. We assign these to properties that are needed when launching signal groups in parallel, which is performed outside this function (Lines 12–13).

We retrieve the current time from the global absolute timer in the controller (Lines 8–15). This time is used for updating the local time properties of the the signal groups. whether the signal groups and elements within the dynamic collections should be executed in parallel is determined on Lines 20. If the groups should be executed in parallel then we retrieve a predefined global queue from the dispatch system and create a dispatch group context (Lines 22–25). The dispatch group

context makes the thread that is executing the signal groups and collection elements in parallel wait until they all have finished updating before continuing its execution. Collection signals are continuous signals; therefore, we only executed them during a sample period (Line 27). If it is a sampling period then we execute the elements in the dynamic collection in parallel (Lines 28–30); otherwise, we only dispatch the signals groups (Line 34). The `runElementFuncGroup` function dispatches tasks to process the element function of each dynamic collection. For example in the particle example in section 3.2.4, the `updateParticle` function is ran in parallel given the particle state and the collection of particles. After dispatching the collection elements, the signal groups are dispatched by the `launchGroupsInPar` (Line 31). The handling of the sequential execution of signal groups is done on Lines 39–49. The group's `remainConstant` property is checked such that only active signal groups are running. Dynamic groups state can become constant after a switching event; thus, we do not run those signal groups. The group's time properties are updated (Line 42) and the group is updated by calling its `update` function (Lines 45). After updating its state, the group notifies its dependents that is has completed its updating process (Lines 49).

The `launchGroupsInPar` function is listed below. The function uses `dispatch_group_async` to launch the groups concurrently, where each task launched updates a group's state. The thread handling the launch of the signal groups waits for all tasks to finish before continuing its execution.

```
1  internal static func launchGroupsInPar(queue : dispatch_queue_t,
2                                         groupQueue: dispatch_group_t) {
3
4      for group in runGroups {
5          if !group.remainConstant {
6              dispatch_group_async(groupQueue,queue) {
7                  //update the time properties
8                  updateGroupTime(group)
9
10                 //Sample the signal group
11                 group.update()
12
13                 //Notify any signal dependents that the
```

```
14                  // group has finished sampling.
15                  group.notifyDependents()
16              }
17          }
18      }
19      // When you cannot make any more forward progress,
20      // wait on the group, which blocks the current thread.
21      dispatch_group_wait(groupQueue, DISPATCH_TIME_FOREVER);
22  }
```

The `update` function of the signal group (shown below) updates all the signals within the group. The `tryReset` function is required to reset a dynamic signal group when its internal state changes (Lines 5). This function reassigns the groups and signals in the switched signal to the group and signal properties of the dynamic signal group. The `waitForInputs` function call (Lines 8) ensures the signal group waits for any prior groups (*i.e.*, input signal groups) it depends on finishes their updating process first. Inside the `waitForInputs` (Lines 19–31), the thread performing the updating process acquire the signal group's lock and checking if all inputs have completed. If necessary, the thread running the update function waits until all inputs are completed. The last input signal group is responsible for notifying the waiting thread that all inputs have completed. Once awoken, the thread begins continues updating the signal group. It is important to note that locking and unlocking are only necessary during parallel runtime execution. If the sequential runtime is running then these methods immediately return and execution continues. The signals within the graph are topologically sorted and returned by the `computeSignalsSampleOrder()` (Line 11). All signals in the group are updated on Lines 14–16.

```
1  /** Begins the updat process for this signal group instance. **/
2  internal func update() {
3
4       //Try to reset of a dynamic signal group
5       tryReset()
6
7      // Wait for all dependent signals to complete
8      waitForInputs()
```

```
9
10      // Compute the signal sampling order
11      self.signalsExecutionOrder = computeSignalsSampleOrder()
12
13      //Use grand central dispatch if the parallel target is enabled
14      for signal in signalsExecutionOrder {
15          signal.runUpFunc()
16      }
17  }
18  /** Waits for any inputs to finish updating **/
19  internal func waitForInputs() {
20      //Acquire the signal group "lock"
21      tryWait(self.grpSema)
22      //Check if all inputs have completed
23      if (self.doneInputs != self.numOfInputs){
24          //If not then release the signal group "lock"
25          trySignal(self.grpSema)
26          //Try to wait for the all inputs to complete
27          tryWait(self._waitSema)
28      }else {
29          //all inputs have completed so release the lock
30          trySignal(self.grpSema)
31      }
32  }
```

The `runUpFunc` function for the discrete signal types (*i.e.*, `DSignal`, `ESignal`) are optimized such that they are only reevaluated when one of their inputs have changed (Line 5). This is determined by the `shouldRun` property, which is set to true by the input signal when it calls its `notifyDependents` function. If the discrete signal has an update function then it is called producing a updated value (Lines 10–12). We then check to see if the old value is equal to the new value (Lines 15–20). If the values are the same then we notify all the dependents that the value has not changed (Line 16); otherwise, we update the signals value to the new value (Line 18) and notify the dependents of the change (Line 19).

```
1  /** Runs the update function of the signal **/
2  private func runUpFunc() {
```

```
3       // Only run if the shouldRun property is set to true
4       // (ie, one or more of its inputs has changed)
5       if shouldRun {
6           let oldVal = self.inVal
7           var newVal  = oldVal
8           /If the signal has an update function then run it to
9           //get the new value and status.
10          if let fn = self._upFunc {
11              newVal = fn(self)
12          }
13          //Notify all dependents this
14          // signal has completed its updating process
15          if self.inVal == oldVal {
16              notifyDependents(false)
17          }else {
18              self.inVal = newVal
19              notifyDependents(true)
20          }
21      }else {
22          notifyDependents(false)
23      }
24  }
```

Currently the continuous signals do not have change propagation because we do not have a mechanism for determining if a signal depends on time, which is always changing. Thus, if the current update cycle is a sample period then we update the signal value and notify its dependents.

```
1   /** Runs the update function of the signal **/
2   func runUpFunc() {
3           if SignalGroupController.isSamplePeriod  {
4               //If the signal has an update function then run it to
5               //get the new value and status.
6               if let fn = self._upFunc {
7                   self.inVal = fn(self)
8               }else {
9               notifyDependents(true)
10          }else {
11              notifyDependents(false)
12          }
```

```
13            }
14    }
```

6.5.3 Rendering System

The rendering system contains a frame queue that holds all submitted frames during a sample period. The rendering system renders each frame. The images generated by each rendered frame is blended to produce the final image that are shown on the screen. The process of rendering a frame is provided in Figure 6.7.

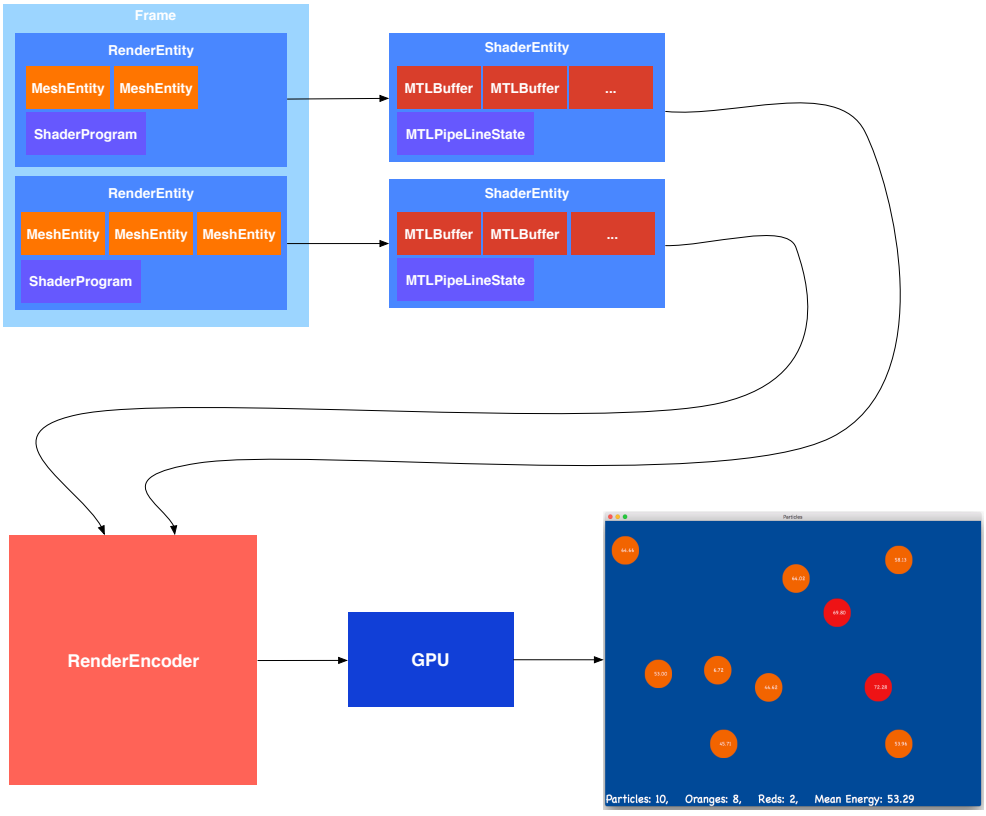


Figure 6.7: The process of rendering a frame using the render system.

A frame consists of render entities. These entities contain a shader program and a collection of meshes that are rendered using the shader program within the entity. The render entities are mapped to an internal object known as a `ShaderEntity`, which prepares the mesh data for rendering

by converting the mesh entities and shader programs into their equivalent Metal forms. For example, a Tesel `MeshEntity`'s data is loaded into a Metal vertex buffer (*i.e.*, `MTLBuffer`). The shader entity creates Metal buffer objects and loads the vertex attribute data and additional mesh data (*i.e.*, uniforms, textures, *etc.*) into these buffer objects. The `MTLRenderPipelineState` objects define the graphics state, including vertex and fragment shader functions for a shader program. The shader entity creates a pipeline state and uses the shader program object to initialize its state. Creating a pipeline state is very expensive and should only be done once during a application's lifecycle. Thus, the rendering system caches all shader entities for future usage. If a frame uses the shader program again, then the rendering system will retrieve the shader entity associated with that shader program and reload all necessary mesh data.

The `RenderEnconder` object is responsible for submitting draw calls to the GPU. The rendering system creates a `RenderEnconder` object for each frame. The render encoder submits draw calls by activating each shader entity's pipeline state (*i.e.*, making a shader program active), and assigning the shader entity's buffer objects to memory locations on the GPU. Finally, the encoder calls the appropriate draw call based on whether the mesh is a triangle mesh or element-indexed triangle mesh. Once the GPU finishes executing all draw calls submitted by the render encoder, the final image is passed to the display for drawing.

CHAPTER 7

APPLICATIONS

The examples provided throughout this dissertation show the usage of the language mechanisms are trivial. They do not fully explain the usefulness and practically of implementing applications using Tesel. In this chapter, present three more substantial examples, which demostrate the efficiency and scalability of our implementation. In particular, we show that using signal groups and parallel collections provide additional performance benefits as the application's workload increases. We also demonstrate the feasibility of writing Tesel applications compared to their event-driven versions.

We begin the chapter with an application that uses the boids algorithm [67] to demonstrates the efficiency and scalability of dynamic collections. Next, we demonstrate the performance gains achieved by using signal groups by dividing up computationally expensive algorithms into smaller tasks. Finally, we discuss the benefits of implementing the applications discussed in this chapters versus their even-drive implementations.

Our test machine is an Apple Mac Pro with a 3.57 GHz 6-Core Intel Xeon E5 processor with 6 cores, 16Gb of memory that is running Mac OS X 10.11.3. For each benchmark, execution time is given in seconds and is the arithmetic mean of 30 individual runs. The launching of threads to execute the tasks that are assigned to the dynamic collections and signal groups is controlled by the thread management system (Grand Central Dispatch). Thus, we have no way of determining how the system schedules tasks, but allowing the system to mange threads gives it the advantage to optimize the scheduling of tasks based on its workload.

7.1 Dynamic Collections: Boids

The first application demonstrates the scalability of our dynamic collections implementation. We implement Craig Reynold's boids algorithm [67]. Reynolds designed this algorithm to simulate the

flocking behavior of various species. Flocking comes from the practice of birds flying or foraging together in a group. A flock is similar to groups of other animals, such as shoaling of fish, or swarming of insects. Reynolds developed a program called Boids that simulates local agents (*i.e.*, boids) that move according to three simple rules: separation, alignment, and cohesion. Reynolds describes each rule as follows.

- Separation: boids steer away from their flockmates to avoid over crowding within the group.
- Alignment: boids head in the same direction as their local flockmates.
- Cohesion: boids try to stay together by moving towards the average position of their local flockmates.

With information about neighboring boids, along with these three steering rules, a reasonably accurate representation of how flocking works in real-life can be simulated. We discuss the Tesel specific portions of the application and leave the reader to look over the full implementation of the Boids application that is provided in Appendix B.

The Tesel specific code is split into two signal groups: `Flock`, which handles moving the boids, and `Viewer`, which handles the rendering aspects of the application. The code sample below represents the heart of the application, which is in the `Flock`'s initialization method and the `move` method of the `Boid` struct.

```
1  class Flock : SignalGroup {
2
3      //a collection signal for all the boids
4      var boids : DSignal<TeselCollection<Boid>>!
5
6      init(numOfBoids : Int, initWinSize : CGSize) {
7          // Initialize the signal group
8          super.init()
9
10         // Initialize the initial set of boids
11         var initialBoids : [Boid] = []
```

```
12
13          for _ in 1...numOfBoids {
14              // Define and add a boid to the initial
15              // collection
16              let boid = Boid(winSize:initWinSize)
17              initialBoids.append(boid)
18          }
19
20          //Give the collection signal the initial set of boids
21          let boidsCollect = collect(initialBoids,
22                                     efn: moveBoid)
23
24          // Generate an event occurence that will contain the
25          // new collection of the boids every time the application
26          // is able to render.
27          let boidsEvent = sampleWithC({(boids : TeselCollection<Boid>,
28                                         _ : NoEventVal) in
29              return boids}, in1: boidsCollect, in2: TeselExternal.render)
30
31          // Update the output of the flock (i.e., the collection
32          // of boids) based on the prior event signal
33          self.boids = hold(TeselCollection(rawArray:initialBoids),
34                            in1:boidsEvent)
35      }
36      func moveBoid (boid : Boid, flock : [Boid]) -> Boid {
37          return boid.move(flock)
38      }
39  }
40  struct Boid {
41       //...
42      public func move(flock : [Boid]) -> Boid {
43
44          //Separation
45          var sep : float2  = seperate(flock)
46
47          //Alignment
48          var ali : float2 = align(flock)
49
50          //Cohesion
51          var coh : float2 = cohesion(flock)
```

```
52
53          // Arbitrarily weight the steering forces
54          sep *= 2
55          ali *= 1.5
56          coh *= 1.5
57
58          // Determine the accleration based
59          // off the three steering calculations
60          let acceleration = sep + ali + coh
61
62          //Calculate new velocity
63          var newVel = self.vel + acceleration
64          newVel = limit(newVel, maxVal:maxSpeed) //Limit speed
65
66          //Calculate new position
67          var newPos += newVel
68
69          // Wrap around the boid if it trails off the screen
70          newPos = clampToScreen(self.pos)
71
72          // Return the new boid
73          return Boid(pos:newPos, vel:newVel, cxt:self.context)
74      }
```

Line 4 defines a discrete signal that holds the collection of boids[1] that are rendered each frame. Boids wrap around the window as they move off the view area. Thus, Lines 13-18 initialize the collection of boids by giving each boid the current window size. Line 21 defines the dynamic collection of boids where each boid is updated by the `move` function. Line 27 samples the collection each time we are able to render a frame (`TeselExternal.render`), which causes an update to the boids discrete signal on Line 33. The `move` function (Lines 36–38) redirect the moving of a boid to the internal `move` method of the `Boid` struct. The `move` function within the `Boid` struct handles the actual moving the boid based on the three rules[2] described earlier.

1. As mentioned earlier, Swift arrays cannot implement `Equatable`, therefore we provide a wrapper class for collections.

2. Full implementations of each rule is supplied in Appendix B.

We ran the boids application to test the efficiency of the dynamic collection that represents the boids. In particular, we time how long the implementation takes to complete an update cycle (*i.e.*, a sample period in the boids case). For an experimental run, we time 1000 update cycles. We averaged these 1000 timed cycles to represent the average update time for a run. We perform both sequential and parallel runs of the dynamic collection to compare their update cycle's execution times. Table 7.1 reports the execution times when increasing the number of boids in the collection. As the application's workload increases, the amount of time to update the dependency graph increases. The results demonstrate that the parallel execution of the collection performs better as the number of boids increase, whereas performance degrades in the sequential execution. Figure 7.1 visually clarifies these results by showing the speedup graph of the sequential execution versus the parallel version. We attribute the performance gains of the parallel execution to the implementation efficiently handling the dispatch queue that executes the collection in parallel.

Table 7.1: Boids: update cycle execution times (seconds), per number of boids, cycle iterations:1000

Boid Sizes	Sequential Execution	Parallel Execution
25	0.0006	0.0007
50	0.0022	0.0010
75	0.0044	0.0016
100	0.0073	0.0023
300	0.0599	0.0136
600	0.2277	0.0519
900	0.5045	0.1007
1200	0.8921	0.1780
1500	1.3847	0.2699

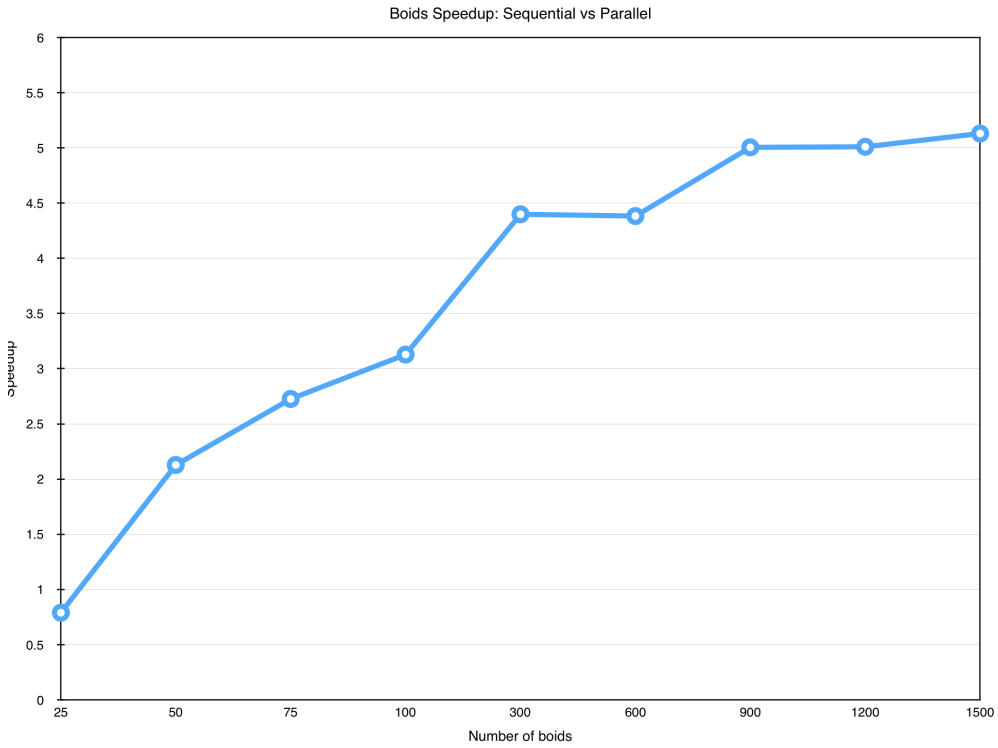


Figure 7.1: Boids speedup of the parallel vs sequential execution.

7.2 Signal Groups: Procedural Generation

The method of generating application content algorithmically is known as procedural generation. For example, this method is used in video games to generate game content (*e.g.*, textures) randomly on the fly. Generating fractals to represent textures and meshes is an example of procedural generation. Fractals can also be used to generate aesthetically pleasing mathematical visualizations. Fractal generation is used to demonstrate the scalability of signal groups. The application computes the Mandelbrot set [53] and displays it as a texture to the display. This algorithm produces computationally intense tasks that can be evaluated independently. Thus, this benchmark can benefit from using signal groups. The main idea of the implementation is to have a signal group represent a worker that is performing a computation to generate a piece of the output data for the benchmark. Once all signal groups have completed, their generated output is displayed to the user.

The core of the Mandlebrot application is implemented in the initialization steps of the `Viewer` and `MandleBotWorker` signal groups:

```
1  class Viewer : SignalGroup {
2
3      // ...
4
5      init(initWinSize : CGSize) {
6          // ...
7          let iterations = 1000
8          // Produces the complex numbers for the mandlebrot set
9          self.generator = MandlebrotGenerator(imgSize:initWinSize,
10                                                 iterations:iterations)
11
12         // A complex number is a given a worker (ie a signal group)
13         // to determine whether or not the complex number
14         // is part of the set.
15         var workersData : [DSignal<MandlePoint>] = []
16         for complex in generator.generateComplexNumbers() {
17             let worker = MandleBrotWorker(c :complex, iterations:iterations)
18             let mPoint = group(worker, in1:worker.mandlePoint)
19             workersData.append(mPoint)
20         }
21         // Collect all the worker's mandle points
22         // together
23         let mandlePoints = join(workersData)
24
25         // Create the projection information containing
26         // the window size and projection matrix
27         let projection = hold(makeProjInfo(initWinSize),
28             in1:lift(reshape, in1: TeselExternal.windowSize))
29
30         let render = render(flatFrame, in1:projection, in2:mandlePoints)
31          //...
32     }
33     //...
34 }
35 public struct MandlePoint : Equatable {
36     let c         : Complex
```

```
37      let iteration : Int
38  }
39  class MandleBrotWorker : SignalGroup {
40
41      let escape     : Double = 2.0
42      let c          : Complex
43      let iterations : Int
44      var mandlePoint : DSignal<MandlePoint>!
45
46      init(c : Complex, iterations : Int) {
47          self.c = c
48          self.iterations = iterations
49          super.init()
50          //Compute the set of points everytime we need to render
51          let renderEvent = lift(computeMandlePoint, in1:TeselExternal.render)
52
53          //By default the point is not in the set but renderEvent will contain
54          // the updated point
55          let initPoint = MandlePoint(c:c, iteration:-1)
56          self.mandlePoint = hold(initPoint,in1:renderEvent)
57      }
58      func computeMandlePoint(_ : NoEventVal) -> MandlePoint {
59          var iteration = 0
60          var z = Complex(real:0, imaginary:0)
61          repeat {
62              z = z * z + c
63              iteration++
64          } while (z.normal() < escape && iteration < iterations)
65
66          return iteration < iterations ? MandlePoint(c:c, iteration:iteration)
67                                        : MandlePoint(c:c, iteration:-1)
68      }
69  }
```

Line 9 defines the generator that will produce the complex numbers to test whether one is inside the Mandelbrot set. We assign each complex number the generator produces to a `MandleBrotWorker`, which determines if the number is inside the set. The worker data is joined together on Line 23

and render the points in the `flatFrame` function after all workers have completed (Line 30). Inside the `MandleBrotWorker` class, we assign the complex number and the number of iterations to constant variables (Lines 47–48). We determine whether the complex number is inside the set by calling the `computeMandlePoint` function each time we render (Line 51). The updated point is then assigned to `mandlePoint` (Line 56). The `computeMandlePoint` performs the set membership test. A complex number is part of the set if it does not diverge before reaching a certain number of iterations. A number diverges outside the set by having a modulus greater than an escape value (we use 2). The function returns a `MandlePoint` where the `iteration` property states the iteration at which the point escapes or is assigned -1 to mean that the number is not in the set.

As with the boids benchmark, we measure the time for an update cycles to generate the Mandelbrot set. In the case of the Mandelbrot benchmark, we use an upper limit of 1000 iterations. To test the scalability of signal groups, we fluctuate a stride property that is defined within the `MandlebrotGenerator` class. This stride value determines how many complex numbers to test. The value ranges from $(0.25, 5.0)$. The lower the value the more complex numbers that are generated for testing. Thirty runs are performed on a sequential execution of the signals groups and a parallel execution.

Table 7.2 reports the execution times as we increase the size stride value, whereas Figure 7.2 shows the speedup graph. As with the boids benchmark, we see that the parallel execution performs better than the sequential version, at both high and lower stride values (*i.e.*, a higher work-load). Again, we attribute this speedup to processing the signal groups in parallel. We believe we can achieve even better performance gains if the work is batched together. Lower stride values produce larger sets of complex numbers. Assigning each worker to one complex number forces the implementation to execute a large number of tasks, which can degrade performance. Another approach is to have a small number of signal groups and have each group test more than one complex number. Thus, we are spawning fewer tasks, which increases the opportunity for better performance.

Although we show that performance gains are possible with parallel execution, one limitation

of Tesel is that it requires the programmer to compile his application as either *always* sequential or parallel. As the results show, smaller workloads benefit from sequential execution, whereas larger workloads benefit from parallel execution. As future work, we want to research ways to switch between execution modes depending on the application's workload. One area that focuses on managing this task is auto-tuning [10, 76]. With auto-tuning, the implementation can optimize the performance of the application by allowing the programmer to specify parameters that tells the system to switch between the execution types. Thus, the programmer would not need to choose at compile-time the execution target. This area of research is still in its early stages but shows promise for providing this switching ability.

Table 7.2: Mandlebrot: update cycle execution times (seconds), per stride value, testing iterations:1000.

Stride	Memory Size	Sequential Execution	Parallel Execution
0.9	3gb	16.9727	8.3512
1.0	1.78gb	13.6728	6.7082
1.5	1.03gb	6.1217	3.1765
2.0	609mb	3.4487	1.6516
2.5	404mb	2.2766	1.0585
3.0	315mb	1.5377	0.7265
3.5	255mb	1.1365	0.5421
4.0	190mb	0.8684	0.4184
5.0	146mb	0.5597	0.2612
6.0	131mb	0.3838	0.1900

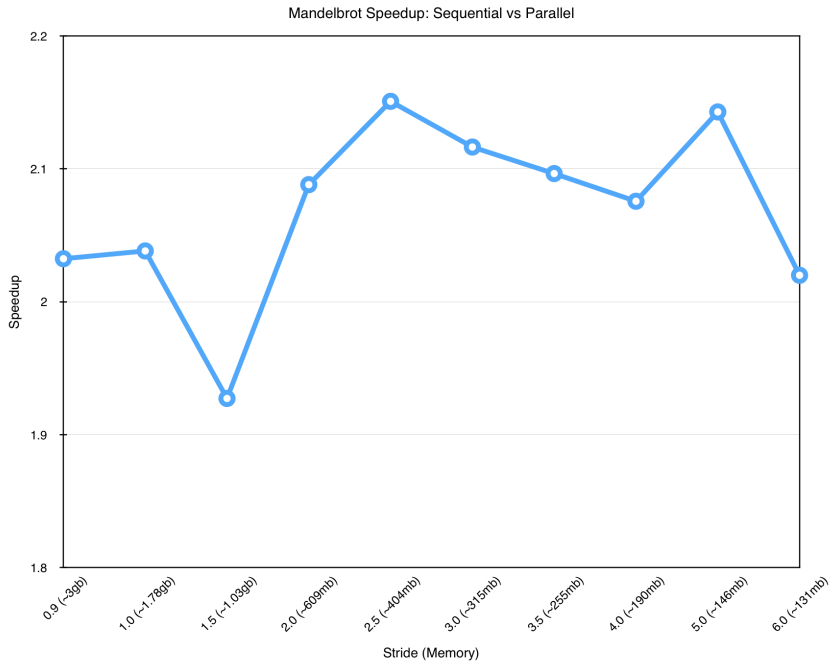


Figure 7.2: Mandlebrot speedup of the parallel vs sequential execution.

7.3 Programmability

Table 7.3: Lines of Code: Tesel vs Event-Driven (Swift) using cloc[24]

Benchmark	Tesel	Event-Driven
Boids	342	591
Mandlebrot	326	544
Heightmap (Terrain Generation)	301	595

Table 7.3 shows the lines of code of both the Tesel version and event-driven version of the benchmarks. The lines of code numbers were generated using cloc [24] and do not include comments, blank lines, or timing code. As the table indicates, the Tesel benchmarks require less code than their event-driven versions. We attribute this programmability benefit from Tesel implementing and handling all of the event-driven features for the programmer. Event-driven applications require the programmer to explicitly manage these events. Additionally, event-driven applications have to deal with rendering aspects of the application such as managing the Metal context, rendering looping, and implementing code to pass data from the CPU and GPU. From a programmability

perspective, a major design goal of Tesel is to remove these nuisances from programmers and allow them to quickly and efficiently develop applications without worrying about the complexities of the low-details.

The table also shows another application (*i.e.*, Heightmap) that generates a natural looking fractal landscape using the midpoint displacement algorithm [33]. The algorithm repeatedly divides up a square equally into four smaller squares. The center of each square is vertically adjusted by a random amount. This subdivision process is continued until a desired level of detail is acceptable. Although this algorithm is computationally expensive and can benefit from signal groups, it quickly becomes memory bound as the size of the landscape increases. Thus, this algorithm is useful for smaller sized terrains. Nonetheless, we still implement this application as another example of a potential application that can be easily be implemented in Tesel to generate smaller terrains.

Tesel also strives for providing mechanisms that bring clarity to a programmer's code. For example, the event-driven application that generates a terrain implements a camera class to represent the virtual camera in the scene. The user clicks on the w-s-a-d keys to move the camera in various directions. But the handling of these event occurrences requires going through multiple redirections before updating the camera. First, event occurrences can only be handled by user interface view classes. Thus, a view class has to override a key callback method to receive key events. For each event occurrence, it has to delegate the handling of the event to its view controller. The view controller contains the camera object, which then has to notify the camera to update. In Tesel, this event handling is much clearer. The camera object directly handles the event occurrence within its local signal network, which it can then easily propagate as input to another signal that updates the camera position. Thus, this approach benefits from code organization since all code is defined within a single signal group instead of through multiple files and code redirections.

As discussed in sections 7.1 and 7.2, parallelism can increase the performance of an application, but managing tasks that are running concurrently is not always easy. A programmer has to efficiently manage threads and provide them with enough work such that the overhead of managing

threads does not further degrade performance. Tesel does not require the programmer to implement their own thread management system but implement tasks in a parallelistic style. These tasks are handled efficiently by the implementation rather than putting the burden on the programmer.

CHAPTER 8

RELATED WORK

In this chapter, we discuss the conceptual models and abstractions used by other FRP and reactive languages that have influenced the design and implementation of Tesel. We also give an overview of optimization techniques that have affected FRP research and Tesel.

8.1 FRP Abstractions and Languages

As discussed in Section 2.3, the two main approaches of continuous-based FRP are CFRP and AFRP. Many FRP researchers have used the conceptual models and abstractions defined by these approaches to implement languages that help further advance the practicality of FRP paradigm to develop reactive programs. In this section, we discuss the FRP languages implemented in these approaches and how their semantics and evaluation models are related to those in Tesel.

The push-pull model was first presented by Conal Elliot in his 2009 paper *Push-Pull Functional Reactive Programming* [31]. The semantics and implementation of the model was used as a basis for implementing the language *Reactive* [31]. Elliot suggests having distinct types for discrete and continuous signals to help improve performance and reaction latency. Discrete signals are defined by two types: *Event* (*i.e.*, external events such as mouse and key presses) and *Reactive* (*i.e.*, a discrete signal with an initial value that is updated periodically). Reactive evaluates the discrete signals using a push-based implementation, since these signals are recomputed only when necessary. This property eliminates the problem of wasteful program recomputations. Continuous signals are defined with the type *Fun* (*i.e.*, timed functions). The evaluation of continuous signals such as integration is handled using a pull-based implementation that samples the signals over a series of discrete time steps. Similar to Reactive, Jeltsch developed *Grapefruit* [47], which also defines three signal types: *DSignal* (*i.e.*, event signals), *SSignal* (*i.e.*, discrete signals with a value), and *CSignal* (*i.e.*, continuous signals) but a significant difference between the two languages is

how they handle switching continuous signals. Reactive's continuous signals can still suffer from time leaks since switching may cause a a long-delay owing to computing the value of the signal being switched in. Whereas in Grapefruit, continuous signals that switch in the future are forced by the type system to incrementally sample the signal before the switch occurs.

Sculthorpe *et al.* presented N-Ary FRP [68], which improves the semantics and evaluation of the abstractions (*i.e.*, signal functions and signals) provided by AFRP by adopting a push-pull model. In N-Ary FRP, signal functions are conceptually functions from signal vectors to signal vectors. A signal vector groups a collection of distinct signals together. The distinction comes from tagging signals as continuous or discrete using signal descriptors, which tag a signal is either an event signal, step signal (a discrete signal with an initial value), or continuous signal. Sculthorpe's dissertation provides the semantics and an unoptimized implementation for N-Ary but also discussed optimizations to improve the implementation. Sculthorpe stated that the primary purpose of this work was to provide a semantics for N-Ary and allow programmers to develop an efficient implementation using these semantics and the optimization techniques discussed.

Tesel is significantly influenced by the push-pull languages Reactive, and Grapefruit, and by the signal function push-pull model used in *N-Ary FRP*. Tesel's signal types directly correspond to the signal types used in these languages. The semantics and implementation of our abstractions, and combinators are similar to the ones provided in these push-pull languages, but with the additional benefits of providing parallelism support via signal groups and dynamic collections. N-Ary FRP and other AFRP languages use arrows as their basis for constructing signal functions, but this feature results in a nonintuitive style of programming that can be hard to understand [21]. Signals groups use a direct style of programming that is familiar to the majority of programmers, therefore making it programmability better for programmers wanting to use an FRP-based language. Tesel also provides 3D rendering constructs that intertwine nicely with the FRP constructs. Thus, programmers to more declarative 3D graphics applications without worrying about the low-level details.

Using signal functions as first class citizens was first introduced in by Courtney and Elliot Fruit [20], which is a FRP language for declaratively specifying graphics user interfaces. As mentioned earlier, signal functions were developed to avoid the large class of time and space leaks that occurred in prior CFRP languages such as Fran [30]. Unlike CFRP, where only its output can be transformed, signal functions also allow for an increase in modularity since transformations are allowed on both input and output. An optimized version of AFRP was presented by Courtney and Hudak in Yampa [22]. Signal functions in Yampa are represented as Generalized Algebraic DataTypes (GADTs) embedded within continuations. With representing signal functions as GADTs, Yampa optimized signal functions by dynamically combining them when switching between signal functions. Although this caused additional overhead when a new continuation was produced, the optimization still provided significant speedups in relation to other AFRP implementations. Unfortunately, both of these languages still suffered from performance latencies owing to representing signals being continuous. Tesel eliminates these problems by using a hybrid-model that makes a distinction between discrete and continuous signals such that the implementation can efficiently handle the evaluation of these distinct types separately.

Languages that use a push-based model are primarily split into categories that are defined by whether reactions are considered to be instantaneous (synchronous) or reactions that cause a delayed response (asynchronous). The synchronous push-based languages have been embedded within many languages such as Scheme [32], Java [19], and Scala [52] and web-based languages such as Elm [23]. These push-based languages construct static data-flow graphs that represent the dependencies between discrete signals. As events come in from the outside environment they are push through the graph to produce output. The main difference between these languages and Tesel is that Tesel also contains the notion of continuous signals that use a pull-based evaluation system to update their values.

Asynchronous push-based languages operate with signals that defined as push-based streams. A signal is a representation of an asynchronous flow where tasks (*i.e.*, events) are push into stream

that can be transformed using higher-order primitives (*e.g.*, map, filter *etc.*) or by a transforming entity (*e.g.*, a server) and the result is delivered to subscribers (*i.e.*, other signals or objects) of that stream. Many of these asynchronous push-based languages are used to develop client-server applications where a task can be asynchronously sent to the server and the result can be handled efficiently by the client. Thus, many of these languages are developed in web-based languages such as Javascript [59] but also other languages such as Swift [1] and Haskell [14]. Tesel's current push-based implementation does not have an asynchronous signal but it might be useful for long-running tasks that can be asynchronously computed in the background. We plan on adding this feature in the future.

8.2 Optimization Techniques

FRP systems have used various optimizations, such as self-adjusting and incremental computation [2, 3, 65], and change propagation [4, 26] to gain better performance. A self-adjusting computation (SAC) is a computation that automatically responds when its data changes. For example, Acar expressed that a self-adjusting program can compute a property that dynamically changes a set of moving objects [2]. This automatic change property is similar to how Tesel and other FRP systems can propagate changes across its dependency graph. The main difference between Tesel (and other FRP systems) and SAC programs is that SAC is not history sensitive, whereas Tesel contains history-sensitive signals (*e.g.*, the discrete signal created using the `foldp` combinator). In a SAC program, only the current state of the program can be used.

Change propagation is an optimization technique that automatically updates the output of a program when its input changes [4, 26]. Similar to the static data-flow graphs constructed for signals in FRP systems, change propagation requires the same data-flow graph to keep track of dependencies between the program's code and data. A dependency is created between a memory location and function when a function accesses the data at that location. If a computation inside a function uses any of these tracked memory locations (*e.g.*, the memory locations of the function's

inputs) then the computation is automatically recomputed if these memory locations change. Tesel uses change propagation in its push-based implementation to avoid recomputing discrete signals when their input do not change. Continuous signals can also use change propagation, but only if it does not depend on time, since time is continuously changing.

CHAPTER 9

CONCLUSION

Functional Reactive Programming provides a clear, concise, and modular way of developing highly interactive and event-driving applications. In particular, using a FRP language is well-suited for developing graphics applications because it provides abstractions (i.e., continuous signals) that nicely model physical phenomena. Many of the current continuous-based FRP languages still suffer from performance issues owing to using a pull-based evaluation model. The FRP system continuously resamples the entire program even if only one portion of the program requires updating. Additionally, graphics programmers still need to handle and manage external events and the rendering of frames to the display.

In this dissertation, we present a new language that uses a push-pull evaluation that makes a distinction between discrete signals and continuous signals. This evaluation model allows for only propagating changes through the program when necessary. We also introduced the notion of signal groups that allow for signals representing a single entity to be grouped together and sampled in parallel for better performance. Our operational semantics precisely describes the meanings of each signal function primitives and how external events and sampling changes the signals and signal groups within a program. Tesel also provides 3D rendering constructs that intertwine nicely for quickly and efficiently developing graphics applications. Our compiler and runtime allow for developing practical graphics applications using signals and signal groups to write more concise and declarative applications without the need to handle external systems or interactions with the graphics hardware. In particular, our rendering system handles the interactions between CPU and GPU that is burdensome task for many 3D graphics programmers. Lastly, we illustrated through experiments that as the application is given more work that our parallel target scales efficiently.

9.1 Future Work

This section outlines further avenues of research to better improve the benefits that Tesel brings to developing graphics application.

- Modern graphics hardware is capable of executing tasks other than rendering graphics. The hardware can perform fast computations in data-parallel applications, which are programs that allow the hardware to execute the same operation on different data elements simultaneously. For example, an image processing application that detects the edges within an image. Tesel could explore extending the `Frame` object to also include data-parallel tasks to run on the GPU and provide mechanisms to allow a signal to know when the result of a task has completed.

- We want to further extend the practicality and usefulness of Tesel by using the language within an educational environment. Teaching students about Tesel and having them use it to develop graphics applications is a great way of determining how easy is it to develop a Tesel application. We can also use this opportunity to make changes or add new mechanisms to the language based on feedback from the students.

- Many of the signal combinators are based on combinators defined in other FRP languages. Currently, Tesel does not contain combinators that could be beneficial in graphics applications. For example, linear interpolation is used in graphic applications to produce smooth animation of objects. Tesel could provide a "lerp" combinator to allow values to be linearly interpolated with respect to time. Additional combinators could include interpolating various curves such as bezier curves or cubic spline curves.

APPENDIX A

SWIFT OVERVIEW

As discussed in section 2.4.1, Tesel is an embedded language within the Swift programming language. The runtime system and the language's core constructs and abstractions are all implemented within Swift. In this chapter, we provide a basic introduction to Swift that allows the reader to follow the examples presented throughout the rest of this thesis. We focus solely on the Swift syntax and constructs used in the examples and the implementation of Tesel. A more in depth explanation of the Swift language is provided in The Swift Programming Language Guide[44].

A.1 Basics

The Swift's language design is highly influenced by long established and popular imperative languages such as C and Objective-C. Thus, it contains syntax familiar to many programmers. In this section, we give an overview of the basic values, standard types, assignment operations, and control-flow mechanisms provided by the language.

A.1.1 Basic Values and Operations

The Swift language provides the common primitive values and types seen in many imperative languages, which are shown in Table A.1. Swift also includes the basic operators for working with these values such as the arithmetic operators (*e.g.*, $+, -, *$, *etc.*) for working with real and integer number values, relational operators to compare values and logical operators to combine boolean values and expressions.

The core mechanism used to associate values with variables is the assignment operator (=). Variables in Swift are either mutable or immutable. Both constants (immutable variables) and variables (mutable variables) must be declared before they are used. Constants are declared with the `let` keyword and variables with the `var` keyword. For example,

Table A.1: The core primitive types of Swift

Type	Sample literal values
Int	..., -3, -2, -1, 0, 1, 2, 3, ...
Float, Double	23.36
String	“hello”, “world”
Bool	true, false

```
let maxNumOfParticles = 20
var meanParticleEnergy = 76.97
```

defines a constant named `maxNumOfParticles` with the value of `20`, and a variable named `meanParticleEnergy` with the value of `76.97`. The types of both the constant and variable is automatically inferred by the compiler, which results in assigning the type `Int` to `maxNumOfParticles` and `Double` to `meanParticleEnergy`. On occasion, the compiler cannot infer the type of the variable; therefore, an explicit *type annotation* is required to specifically state the type:

```
let maxNumOfParticles : Int = 20
var meanParticleEnergy : Double = 23.36
```

By using a colon, followed by a type, we have explicitly defined the type of the variable and constant. Type annotation are also useful for making it clear about the kind of values a constant or variable can store.

A.1.2 Tuples and Arrays

Swift contains many different types to group individual values together. We explain two of those types: tuples and arrays. A tuple groups multiple values of various types into a single compound value. For example, `("Kavon", 43948)` is a tuple that represents a student and their id number. The tuple contains the value `"Kavon"` to represent the student name and `43948` to represent his id number. The resulting type combines a `String` and `Int` together and is described as a tuple of type `(String,Int)`. To access the individual components of the tuple, Swift uses index numbers starting at zero:

```
let student = ("Kavon", 43948)
let name = student.0  // name = "Kavon"
let id = student.1   // id = 43948
```

An array is one of Swift's collection types that stores values of the same type in an ordered list. The type is denoted `[Element]` where `Element` is the type of values the array stores. For example, we can create an array that will hold a list of integers in two ways:

```
//Both lines define an empty integer array
var intArray = [Int]()
var intArray2 : [Int] = []
```

The syntax `[]` is the literal form to create an empty array. Another way to create an array is by using the array literal syntax to define the initial elements within the array:

```
//Both lines define an empty integer array
var names : [String] = ["Kavon", "Joe", "Charisee"]
```

Accessing and modifying the array contents is done by methods and subscript syntax. In particular, the `append` method adds items to the array, and the subscript syntax uses an integer index value to retrieve or change the element at that index:

```
names.append("Brian") // names = ["Kavon", "Joe", "Charisee", "Brian"]
let student = names[0] //  student = "Kavon"
names[1] = "Lamont" // names = ["Kavon", "Lamont", "Charisee", "Brian"]
```

Finally, arrays are concatenated together by using the addition assignment operator (+=):

```
names  + = ["Peter", "Bob"]
// names =  ["Kavon", "Lamont", "Charisee", "Brian", "Peter", "Bob"]
```

A.1.3 Control-flow

Swift provides various types of control flow statements. We focus solely on three statements: *for-in*, *if*, and *switch*. The for-in statement allows for iterating over items within a sequence or

collection of values. For example, the following sums of the values from 1 to 10:

```
var sum : Int = 0
for index in 1...10 {
   sum += index
}
// After iterating through the loop, sum = 55
```

The closed range operator (...) is used to create a range of numbers between 1 to 10, inclusive. the iterator variable (`index`) is assigned to each number within the range and is used to calculate the sum of those numbers in that range. The half-open range operator (..< is anther range operator commonly used with the for-in loop that defines a range that runs from a to b, but does not include b. The prior example can be modified to only sum the numbers from 1 to 9 using the half-open range operator as such:

```
var sum : Int = 0
for index in 1..< 10 {
   sum += index
}
// Now after iterating through the loop, sum = 45
```

Iterating over the items within an array is also done using the for-in loops:

```
var numbers : [Int] = [34, 23, 19]
for element in numbers {
   sum += element
}
// sum = 76
```

Each element is assigned to the iterator variable (`element`), which then can be used within the body of the loop. In this case, we again calculate the sum of the items within the array. If the value of the iterator varible is not needed within the body of the loop then it can be replaced by the underscore character (_). For example, we can compute the multiplication of $5 * 3$ as follows:

```
let num = 5
var factor : Int = 3
var mult   : Int = 0
for _ in 1...factor {
   mult += num
}
// mult = 5 * 3 = 15
```

The if-statement is the same syntax as many other imperative languages with the exception that parenthesis around the condition is optional and braces are required at the beginning and end of an if-case. For example, checking if a variable represents the hours in the day is valid:

```
let hourInMilitary : Int  = 4
var message : String = ""
if hourInMilitary >= 1 && hourInMilitary <= 24 {
   message = "hours is valid"
} else {
   message = "hours is invalid"
}
```

Finally, the switch statement compares a value against several possible matching cases. If it finds a matching case it then executes the statements inside that case. Every switch statement must be *exhaustive*, which means every possible value of the type being considered must be handled by a case within the switch statement. Typically, the `default` case handles any remaining matches that are not addressed explicitly. Lastly, cases can be compounded together by separating each matching value by a comma on the same line of the case. For example, determining the number of days of each month:

```
let month : Int = 10
var daysString : String = ""
switch month {
case 1, 3, 5, 7, 9, 10, 12:
    daysString = "Number of days = 31"
case 4, 6, 9, 11:
    daysString = "Number of days = 30"
case 2:
    daysString = "Number of days = 28 or 29 (leap year)"
default:
    daysString = "Invalid month"
}
```

The first switch case is a compound case, which executes its body if month matches to any of the values for the case (*i.e.*, 1,3,5,7,9,10,12), similar to the second case. The third case handles a singular value and the last case handles all remaining values (*i.e.*invalid integers assigned to the month constant). It is important to note that after executing the body of a matching case the execution of the case statement is completed. In contrast to other imperative languages such as C, where it requires an explicit break statement at the end of every switch case to prevent the execution from falling through to the next switch case.

A.1.4 Optionals

Optionals provide a way to handle situations where the value may be absent. An optional is either assigned an actual value or `nil`, which means the there is not a value assigned to it. An optional is defined by specifiying the type followed by a "?" (*e.g.*, `Int?` is an optional integer) Optionals are similar to using nil with pointers in Objective-C, but they can be used for any type and not just classes. For example, using an optional to represent an error code:

```
var errorCode: Int? = 37
// errorCode contains an actual Int value of 37
errorCode = nil
// errorCode now contains no value
```

Checking to see whether an optional contains a value is done using an if-statement. For example, checking to see if the error code contains a value:

```
var errorCode: Int? = 37
var message : String = ""
if errorCode != nil {
  message = "The errorCode contains a value"
} else {
  message = "The errorCode does not contain a value"
}
```

Comparing the optional against `nil` determines whether it contains a value. Another way to check if an optional holds a value is done using *optional binding*, which assigns the value of an optional (if it has one) to a temporary constant or variable within the condition expression of an if-statement. Optional binding has the form:

```
if (var or let) temp = optional {
    // statements
}
```

If the optional holds a value then the if-body is evaluated by first assigning a temporary constant or variable to the optional value and then executing the body's statements. The temporary is only accessible within the scope of the if-body. For example, using optional binding to retrieve the error code from the prior example:

```
var errorCode: Int? = 37
var message : String = ""
if let actualErrorCode = errorCode {
    message = "The error code contains the value : \(actualErrorCode)"
} else {
    message = "The error code contains no value"
}
```

In this case, the error code optional does have a value; therefore, its value is assigned to the temporary constant, `actualErrorCode` and the body of the if-clause is executed. The temporary is

then used to create a message that contains the actual value of the error code.

Lastly, optionals can be explicitly unwrapped by using the (!) operator to retrieve its value; although, this operator should only be used if the optional is guaranteed to have a value. A runtime error is trigged if nil is assigned to an optional that is being explicitly unwrapped. If the optional is guaranteed to always have a value after that value is first set then Swift recommends using the *implicitly unwrapped optional* type. This type is defined by using a exclamation point (*e.g.*`Int!`) instead of a question mark (*e.g.*`Int?`), which is used to define a normal optional type. An example used from the Swift Programming Guide is shown below, which illustrates the difference in behavior between an optional string and an implicitly unwrapped optional when accessing their underlying value:

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! // requires an exclamation mark

let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString // no need for an exclamation mark
```

As the authors of the guide write:

> "You can think of an implicitly unwrapped optional as giving permission for the optional to be unwrapped automatically whenever it is used. Rather than placing an exclamation mark after the optional's name each time you use it, you place an exclamation mark after the optional's type when you declare it."

A.1.5 Enumerations

An enumeration is a data type that groups a set of named values together. Enumerations have similar syntax to enumerations used in C and Objective C where the definition is defined by using the keyword `enum` and each value is defined as an enumeration case that is introduced using the keyword `case`. For example, defining an enumeration for the days of the week:

```
enum Weekdays {
        case Monday
        case Tuesday
        case Wednesday
        case Thursday
        case Friday
        case Saturday
        case Sunday
}
```

Variables and constants can be declared as an enumeration type by stating the name of the enumeration, followed by a period, and then an enumeration case value.

```
var day = Weekdays.Thursday
```

Swift also provides a short-hand syntax uses the dot syntax:

```
day = .Thursday
```

The type for day is inferred when it is initialized with one of the possible values of `Weekdays`. Thus, the explicit writing of the enumeration name is not required.

A switch or if statement can be used to determine the assigned enumeration value of a variable or constant. Furthermore, the dot syntax can be used; therefore, not requiring the enumeration name before hand. For example, using a switch statement to determine which day of the week the day variable holds:

```
day = .Tuesday
dayAsString = ""
switch day {
case .Monday:
    dayAsString = "The day is Monday"
case .Tuesday:
    dayAsString = "The day is Tuesday"
case .Wednesday:
    dayAsString = "The day is Wednesday"
case .Thursday:
    dayAsString = "The day is Thursday"
case .Friday:
    dayAsString = "The day is Friday"
case .Saturday:
    dayAsString = "The day is Saturday"
case .Sunday:
    dayAsString = "The day is Sunday"
}
// dayAsString = "The day is Tuesday"
```

A.2 Functions

Functions within Swift can be written in a variety of different ways, which results in a great flexibility on accomplishing specific tasks. The basic form of a Swift function,

```
func name (parameterName : type, ...) -> type {
    //statements
}
```

contains the function name, a list of parameters and its types, a return type and a body that contains a list of statements. The function is invoked by simply providing the name of the function followed by any arguments it needs for its parameters. For example, the length of a 2D vector function can be defined and called as:

```
func lenOfVec2 (p1 : float2) -> Float {
    return sqrt(p1.x * p1.x + p1.y * p1.y)
}
let vector = float2(3.0,2.0)
let vecLen =  lenOfVec2(vector) // vecLen = 3.61...
```

Functions are not required to return or take in any parameters. For example, the main job of the below function is to update a global counter:

```
var counter : = 0

func updateGlobalCounter ()  {
    counter += 1
}
updateGlobalCounter() // counter = 1
updateGlobalCounter() // counter = 2
```

Similar to functional languages, functions also are higher-order and have their own *function type*, which consist of a specification of its parameter types and return type. For example, We can assign a variable to be of a function type and assign it an appropriate function and call the function using that same variable:

```
//Assign vecFunction to be equal to the length of a 2D vector function
var vecFunction : (float2) -> Float = lenOfVec2

//We can call it using the same variable
let len = vecFunction(float2(3.0,2.0)) // len = 3.61
```

A.2.1 Generic Functions

Many of the signal generating functions in this thesis use generic functions, which is useful for writing functions that can work with any type. For example, here's an example of finding the maximum between two integers:

```
func maxInt( a : Int, b : Int) -> Int {
   if a > b  {
        return a
   } else {
        return b
   }
}
```

But this function **only** works for integers. If we wanted to find the maximum for Doubles, or Floats then we would need to write redundant functions where the types would only be changed. Instead, the function can become a generic function as such:

```
func max<T>( a : T, b : T) -> T {
   if a > b  {
        return a
   } else {
        return b
   }
}
```

We define the type parameter (`T`) between the angle brackets right after the function name. The type parameter can be used to define the types of the parameters, the return type, or local constants or variables within the function body. The type parameter is replaced with the actual type whenever the function is called (*e.g.*, `Int`, `Float`, *etc.*.). A function can have more than one type parameter by writing more type parameters names within the angle brackets, separated by commas. For example, `func someFunc <I,O> (in1 : I, in2: I) -> O` defines three type parameters that are used to define specific types for parameters and return type.

A.3 Classes and Structs

Swift provides to ways to represent objects in the language: classes, and structs. The two representations provide nearly the identical features with the exception that classes provide additional benefits such as inheritance, type casting and deinitializers. We do not explain the details about

these additional class features in this chapter, but instead refer you to the programming guide for a more comprehensive explanation. Another significant difference between classes and sturcts is that classes are reference types and structs are value types. A value type is a type whose value is *copied* when assigned to a variable, constant, or function parameter, whereas reference types is not copied. Swift strongly encourages using value types because the compiler provides various optimizations for working with these types. Thus, we will stick to talking about structs in this section but classes can be interchanged with the code examples with the understanding that they are reference types.

Structs and classes are defined with the keywords `struct` and `class` followed by their name. For example, we can define the struct to represent a 2D vector as follows:

```
struct Vec2 {
    // struct definition
}
```

Both classes and structs can define a properties (*i.e.*, fields in other languages), initialization methods (*i.e.*, constructors in other languages), and methods. We can augment the vector struct from above to contain properties for x and y, an initialization method to initialize the properties, and a method to return a string representation for the properties:

```
struct Vec2 {
    let x : Float
    let y : Float

    init(x : Float, y : Float) {
         self.x = x
         self.y  = y
    }
    func toString ()  -> String  {
         return "(\(x),\(y))"
    }
}
```

Similar to other object-oriented languages, Swift also contains an implicit property, namly `self`,

which is the property pertaining to the instance itself. An instance of the Point class is created by stating its name and providing any arguments needed by its constructor. After creating an instance, properties and methods are accessed and invoked by using dot syntax as such:

```
let pt = Vec2(x:23.0, y:93.34)
let ptX = pt.x  // ptX = 23.0
let description = pt.toString() // description = "(23.0,93.34)"
```

A.4 Protocols

A protocol defines an *interface*, which is a group of related methods, and properties that suit a particular task. A protocol is adapted by a struct or class, which provides the actual implementation. All methods and properties specified by the protocol must be implemented by the *confomring* type. For example, we can define the `VectorType` protocol that defines a list of method that represent a vector:

```
protocol VectorType {
    func length()  -> Float
}
```

We can then make the `Vec2` struct adapt the `VectorType` and provide the implementation of the protocol's length method:

```
struct Vec2 : VectorType {
   let x : Float
   let y : Float

   init(x : Float, y : Float) {
        self.x = x
        self.y  = y
   }
   func toString ()  -> String  {
        return "(\(x),\(y))"
   }
   func length()  -> Float {
        return sqrt(x * x + y * y)
   }
}
```

A struct adapts to a protocol by placing a colon after its name followed by the protocol name. After the struct implements all requirements by the protocol it is said it has conformed to that protocol.

APPENDIX B

SELECTED CODE SOLUTIONS

B.1 Boids

The code defined below is a port application where the boid algorithm is largely based off the Coffescript code [13] provided by Harry Brundage and Processing code [71] provided by Daniel Shiffman with modifications for Tesel and Swift.

B.1.1 Main.swift

```
func teselMain() -> (TeselConfiguration, Viewer) {

    //The initialize size of the window.
    let initWinSize = CGSize(width:1024, height:768)

    //Configuration object for setting up the window
    let config = TeselConfiguration(windowSize: initWinSize,
                                    windowTitle:"Boids Simulation")

    let viewer = Viewer(initWinSize: initWinSize)

    return (config,viewer)
}
```

B.1.2 Viewer.swift

```
func ==(lhs: ProjectionInfo, rhs: ProjectionInfo) -> Bool {
    return lhs.matrix == rhs.matrix &&
        lhs.winSize == rhs.winSize
}

//Stores information about the projection matrix and
```

```
// the display size.
struct ProjectionInfo : Equatable {
    let matrix  : float4x4
    let winSize : CGSize
}

class Viewer : SignalGroup {

    // The texture shader program
    let shaderProg   : FlatShaders

    init(initWinSize : CGSize) {
        //Represents the shader program
        self.shaderProg = FlatShaders()
        super.init()

        let projection = hold(makeProjInfo(initWinSize),
            in1:lift(reshape, in1: TeselExternal.windowSize))
        let flock = Flock(initWinSize:initWinSize)
        let boids = group(flock, in1:flock.boids)

        let _ = render(flatFrame, in1:projection, in2:boids)
    }
    /*** Viewer Signal Functions ***/
    func makeProjInfo(winSize : CGSize) -> ProjectionInfo {
        let projection = Transform.ortho(left:0,
            right:Float(winSize.width),
            bottom:Float(winSize.height),
            top:0, nearZ:-1.0, farZ: 1.0)
        return ProjectionInfo(matrix:projection, winSize:winSize)
    }
    func reshape(winEvt : WinEvent) -> ProjectionInfo {
        return makeProjInfo(winEvt.size)
    }
    func flatFrame(projInfo : ProjectionInfo,
                   bCollection : TeselCollection<Boid>) -> Frame {

        //Stores all the mesh entities for the boids
        var meshes : [MeshEntity<Vertex>] = []
        let boids = bCollection.rawArray
```

```
    //Retrieve all the meshes for the boids (i.e., the triangles)
    for boid in boids {
        let entity = boidMeshEntity(boid,info:projInfo)
        meshes.append(entity)
    }

    //Create the MeshRenderEntity that will use the texture
    // shader program for rendering
    let flatEntity = MeshRenderEntity<Vertex,FlatShaders>(
        meshes:meshes,
        program:shaderProg, info:FlatShaders.info)

    //Create a 2D text mesh to display on the screen
    var textMeshes : [Text2DMesh] = []
    var stats = Text2DMesh(location:CGPoint(x:0, y:0),
        size:CGSize(width:2048, height:45),
        text:"Number of boids = \(boids.count)")
    stats.font = NSFont(name:"Chalkboard", size:30)
    stats.color = NSColor.whiteColor()
    textMeshes.append(stats)

    //Provide all the render entities for frame along with the
    // clear color and a custom color attachment.
    var frame = Frame(clearColor:float4(0.157,0.157,0.157,1.0),
        entities:[flatEntity])
    frame.textMeshes = textMeshes

    return frame
}
func boidMeshEntity(boid: Boid, info : ProjectionInfo)
        -> MeshEntity<Vertex> {

    //Get the direction the boid is heading in
    let angle = (-1 * atan2(-1 * boid.vel.y, boid.vel.x))
                + Float(M_PI/2.0)
    let direction : Float = Transform.degrees(radians:angle)
    //The boids position is based on its center but anchor location
    // of the
    // triangle that reprsents the boid is
```

```
        //at the upper right cornder.
        let actualPos = float2(boid.pos.x - Boid.radius,
                               boid.pos.y - Boid.radius)

        //Translation matrix
        let transMat = Transform.translate(x:actualPos.x,
                                           y:actualPos.y, z:0.0)

        //Rotation matrix based on the direction of the boid
        let rotModel = Transform.rotate(transMat, x:0.0, y: 0.0,
                                        z:direction)

        //Scale the matrix by doubling it radius
        let model    = Transform.scale(rotModel,
            scaleVector:float3(x:Float(Boid.radius * 2.0),
                y:Float(Boid.radius * 2.0),
                z:0.0))

        //Creates the uniforms for this mesh
        let uniforms = FrameUniforms(model:model,
            projectionView:info.matrix,
            color:float4(1.0,1.0,1.0,1.0))

        //Returns a mesh entity
        return MeshEntity<Vertex>(mesh:Boid.mesh, uniforms:uniforms,
                                  textures:nil, samplers:nil)
    }
}
```

B.1.3 Flock.swift

```
class Flock : SignalGroup {

    //a collection signal for all the boids
    var boids : DSignal<TeselCollection<Boid>>!

    init(numOfBoids : Int = 126, initWinSize : CGSize) {
        super.init()
        // Initialize the initial set of boids
        var initialBoids : [Boid] = []
```

```
        let cxt = BoidContext(winSize:initWinSize)
        for _ in 1...numOfBoids {
            // Add the boid to the array only the initial collection
            // of boids and pass the collection signal to all the boids
            // so they have a reference to all boids in the simulation
            let boid = Boid(cxt:cxt)
            initialBoids.append(boid)
        }

        //Given the collection signal the initial set of boids
        let boidsCollect = collect(initialBoids, efn: moveBoid)
        let boidsEvent = sampleWithC({(boids : TeselCollection<Boid>,
             _ : NoEventVal) in
            return boids}, in1: boidsCollect, in2: TeselExternal.render)

        self.boids = hold(TeselCollection(rawArray:initialBoids),
                          in1:boidsEvent)
    }
    func moveBoid (boid : Boid, flock : [Boid]) -> Boid {
        return boid.move(flock)
    }
}
```

B.1.4 Boid.swift

```
// ? constant
let ? = Float(M_PI)

// Maxium steering force
let maxForce : Float = 0.05

// Maximum speed
let maxSpeed : Float  = 2

//Limit the speed to a maxium value
public func limit (vec : float2, maxVal : Float) -> float2 {
    if length(vec) > maxVal {
        return normalize(vec) * maxVal
    }
```

```
    return vec
}
// An extension for keeping the mesh information seperate from the
// boid structure definition
extension Boid {

    static private var _mesh  : TriMesh<Vertex>?
    static var mesh : TriMesh<Vertex> {
        if let triMesh = _mesh {
            return triMesh
        }else {
            let vertices = [Vertex(pos:float3(0.75,1,0)),
                Vertex(pos:float3(0.5,0,0)),
                Vertex(pos:float3(0.25,1,0))]
            return TriMesh<Vertex>(vertices)
        }
    }
}
public func ==(lhs:Boid, rhs:Boid) -> Bool {
    return lhs.pos == rhs.pos && lhs.vel == rhs.vel
}
public struct BoidContext {
    let winSize : CGSize
}
public struct Boid : CustomStringConvertible,Equatable  {

    // Represents the current position of the boid
    let pos     : float2

    // Represents the current velocity of the boid
    let vel     : float2

    // Represents the context information for the boid
    let context : BoidContext

    static var radius : Float = 11.0

    public var description: String {
        return "Boid{pos: \(pos), vel: \(vel)"
    }
```

```
init(pos: float2, vel:float2, cxt: BoidContext) {
    self.pos = pos
    self.vel = vel
    self.context = cxt
}
init(cxt: BoidContext) {

    //Place the boid initially in the center of the screen
    let center = float2(Float(cxt.winSize.width/2),
                        Float(cxt.winSize.height/2))
    self.pos = center

    //Decide a random velocity for the boid
    self.vel = float2(Random.within(-1...1),Random.within(-1.0...1))

    //Assign the context
    self.context = cxt
}
public func move(flock : [Boid]) -> Boid {

    //Separation
    var sep = seperate(flock)

    //Alignment
    var ali = align(flock)

    //Cohesion
    var coh = cohesion(flock)

    // Arbitrarily weight these forces
    sep *= 2
    ali *= 1.5
    coh *= 1.5

    let acceleration = sep + ali + coh

    var newPos = clampToScreen(self.pos)

    //Calculate new velocity
```

```
    var newVel = self.vel + acceleration
    newVel = limit(newVel, maxVal:maxSpeed) //Limit speed

    //Calculate new position
    newPos += newVel

    return Boid(pos:newPos, vel:newVel, cxt:self.context)
}
//Clamp the boids to around the screen.
private func clampToScreen (pos : float2) -> float2 {
    var newPos   = pos
    let  winSize = context.winSize
    if pos.x < -Boid.radius {
        newPos.x = Float(winSize.width) + Boid.radius
    }
    if pos.y < -Boid.radius {
        newPos.y = Float(winSize.height) + Boid.radius
    }
    if pos.x > Float(winSize.width) + Boid.radius {
        newPos.x = -Boid.radius
    }
    if pos.y > Float(winSize.height) + Boid.radius {
        newPos.y = -Boid.radius
    }

    return newPos
}
// A method that calculates and applies a steering force towards
// a target
private func seek (target : float2) -> float2 {
    // A vector pointing from the location to the target
    var desired = target - self.pos

    let d = length(desired)

    if d > 0
    {
        // Scale to maximum speed
        desired = normalize(desired)
        //desired *= maxSpeed
```

```
        if d < 100.0 {
            desired *= (maxSpeed * d / 100.0)
        } else {
            desired *= maxSpeed
        }

        // Steering = Desired minus Velocity
        var steer = desired - self.vel

        // Limit to maximum steering force
        steer = limit(steer, maxVal: maxForce)

        return steer;
    }

    return float2(0,0)
}

// Separation
private func seperate (boids : [Boid]) -> float2 {
    let desiredSeparation : Float = Boid.radius * 6.0
    var steer = float2(0.0)
    var neighbors = 0

    // For every boid in the system, check if it's too close
    for neighborBoid in boids {
        let dist = distance(self.pos,neighborBoid.pos)

        // If the distance is greater than 0 and less than an
        // arbitrary amount (0 when you are yourself)
        if (dist > 0 && (dist < desiredSeparation)) {
            // Calculate vector pointing away from neighbor
            var diff = self.pos - neighborBoid.pos
            diff = normalize(diff)
            // Weight by distance
            diff = float2 (diff.x / dist, diff.y / dist)
            steer += diff
            // Keep track of how many
            neighbors += 1
```

```
        }
    }

    // Average -- divide by how many
    if neighbors > 0 {
        steer = float2 (steer.x / Float(neighbors),
                        steer.y / Float(neighbors))
    }

    return steer
}
// Alignment
public func align (boids : [Boid]) -> float2  {
    let neighbordist : Float = Boid.radius * 16.0
    var sum = float2(0.0)
    var neighbors = 0

    // For every boid in the system, check if it's too close
    for neighborBoid in boids {
        let dist = distance(self.pos,neighborBoid.pos)

        // If the distance is greater than 0 and less than an
        // arbitrary amount (0 when you are yourself)
        if (dist > 0 && (dist < neighbordist)) {
            sum += neighborBoid.vel
            neighbors += 1
        }
    }

    // Average -- divide by how many
    if neighbors > 0 {
        var steer = float2 (sum.x / Float(neighbors),
                            sum.y / Float(neighbors))

        steer = limit(steer, maxVal: maxForce)
        return steer
    }

    return float2(0.0)
}
```

```
    // Cohesion
    public func cohesion (boids : [Boid]) -> float2  {
        let neighbordist : Float = Boid.radius * 2
        var sum = float2(0.0)
        var neighbors = 0
        let winSize = context.winSize
        // For every boid in the system, check if it's too close
        for neighborBoid in boids {
            let dist = distance(self.pos,neighborBoid.pos)

            // If the distance is greater than 0 and less than an
            // arbitrary amount (0 when you are yourself)
            if (dist > 0 && (dist < neighbordist)) {
                sum += (neighborBoid.pos.wrapRelativeTo(self.pos,
                        dimensions:winSize))
                neighbors += 1
            }
        }

        // Average -- divide by how many
        if neighbors > 0 {
            sum = float2 (sum.x / Float(neighbors),
                          sum.y / Float(neighbors))
        }else {
            sum = self.pos
        }

        // Steer towards the locaiton
        return seek(sum)

    }
}
```

B.2 Mandlebrot

The code for the Mandlebrot algorithm with its additional support code was provided by a tutorials written by Colin Eberhardt [27] and Apple inc. [46]. These algorithms were combined with Tesel

code to produce an application that displays the Mandlebrot set as a texture.

B.2.1 main.swift

```
func teselMain() -> (TeselConfiguration, Viewer) {

    //The initialize size of the window.
    let initWinSize = CGSize(width:1024, height:768)

    //Configuration object for setting up the window
    let config = TeselConfiguration(windowSize: initWinSize,
        windowTitle:"MandleBrot")

    let viewer = Viewer(initWinSize: initWinSize)

    return (config,viewer)
}
```

B.2.2 Utils.swift

```
struct ImageTexture {

    private static var _mesh : TriMesh<FlatAttributes>? = nil

    static var mesh : TriMesh<FlatAttributes> {
        if let mesh = _mesh {
            return mesh
        }else {
            let attribs : [FlatAttributes] = [
                FlatAttributes(pos:float3(1,1,0), tex:float2(1,0)),
                FlatAttributes(pos:float3(0,1,0), tex:float2(0,0)),
                FlatAttributes(pos:float3(0,0,0), tex:float2(0,1)),
                FlatAttributes(pos:float3(0,0,0), tex:float2(0,1)),
                FlatAttributes(pos:float3(1,0,0), tex:float2(1,1)),
                FlatAttributes(pos:float3(1,1,0), tex:float2(1,0))
            ]
            _mesh = TriMesh<FlatAttributes>(attribs)
            return _mesh!
        }
```

```
    }
}
```

B.2.3 Flat-shader.swift

```
//The Attributes representing this shader program
struct FlatAttributes : Attributes {
    var pos           : float3;
    var tex           : float2;
}

//The Textures used by the shader program
struct FlatTextures   : Textures {
    var flatTex       : Texture2D;
}

//The Samplers used by the shader program
struct FlatSampler    : Samplers {
    var flatSampler   : Sampler;
}

//The Uniforms needed for the shader program
struct FrameUniforms : Uniforms {
    var model          : float4x4;
    var projectionView : float4x4;
    var color          : float4;
}

public struct FlatTexShaders : ShaderProgram {

    public var vertFuncName   : String  = "vertexFunc";
    public var fragFuncName   : String  = "fragFunc";

    struct ProjectedVertex : FragInput
    {
        var position : float4;
        var texCoord : float2;
    }
    func vertexFunc(vInfo: VertexInfo, vert: FlatAttributes,
                    uniforms: FrameUniforms) -> ProjectedVertex {
```

```
        let clipCoord : float4 = uniforms.projectionView * uniforms.model
            * float4(vert.pos.x, vert.pos.y, vert.pos.z, 1.0);

        return  ProjectedVertex(position:clipCoord, texCoord:vert.tex);
    }
    func fragFunc(fInfo: FragInfo, pVertex: ProjectedVertex,
                  uniforms: FrameUniforms, texture : FlatTextures,
                  samplers : FlatSampler) -> float4
    {
        return texture.flatTex.sample(samplers.flatSampler,
                                      coord:pVertex.texCoord);
    }
}
```

B.2.4 MandleBrotWorker.swift

```
/* Copyright 2013 Colin Eberhardt
Linq to Objective-C

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/
//  Modifications by Lamont Samuels on 7/26/16.
```

```
//  Copyright   2016 Lamont Samuels. All rights reserved.
//


import Foundation
import simd
import Tesel

public func ==(lhs : MandlePoint, rhs: MandlePoint) -> Bool
{
    return lhs.c == rhs.c && lhs.iteration == rhs.iteration
}

public struct MandlePoint : Equatable {
    let c         : Complex
    let iteration : Int
}

class MandleBrotWorker : SignalGroup {

    let escape     : Double = 2.0
    let c          : Complex
    let iterations : Int
    var mandlePoint : DSignal<MandlePoint>!

    init(c : Complex, iterations : Int) {
        self.c = c
        self.iterations = iterations
        super.init()
        let renderEvent = lift(computeIterations, in1:TeselExternal.render)
        let initPoint = MandlePoint(c:c, iteration:-1)
        self.mandlePoint = hold(initPoint,in1:renderEvent)
    }
    func computeIterations(_ : NoEventVal) -> MandlePoint {
        var iteration = 0
        var z = Complex(real:0, imaginary:0)
        repeat {
            z = z * z + c
            iteration++
        } while (z.normal() < escape && iteration < iterations)
```

```
        return iteration < iterations ? MandlePoint(c:c, iteration:iteration)
            : MandlePoint(c:c, iteration:-1)
    }
}
```

B.2.5 ComplexNumber.swift

```
/* Copyright 2013 Colin Eberhardt
Linq to Objective-C

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
**/
//  Modifications by Lamont Samuels on 7/26/16.
//  Copyright   2016 Lamont Samuels. All rights reserved.
//


public func ==(lhs: Complex, rhs: Complex) -> Bool {
    return lhs.real == rhs.real && lhs.imaginary == rhs.imaginary
}
public func +(lhs: Complex, rhs: Complex) -> Complex {
    return Complex(real:lhs.real + rhs.real,
```

```
                    imaginary:lhs.imaginary + rhs.imaginary)
}
public func *(lhs: Complex, rhs: Complex) -> Complex {
    return Complex(real: lhs.real * rhs.real - lhs.imaginary * rhs.imaginary,
        imaginary: lhs.real * rhs.imaginary + lhs.imaginary * rhs.real)
}
public struct Complex: Equatable {

    public var real: Double
    public var imaginary: Double

    func normal() -> Double {
        return real * real + imaginary * imaginary
    }
}
public struct ComplexRect {

    public var topLeft: Complex
    public var bottomRight: Complex
    public var bottomLeft: Complex
    public var topRight: Complex

    public init(c1: Complex, c2: Complex) {
        let tlr = min(c1.real, c2.real)
        let tli = max(c1.imaginary, c2.imaginary)
        let brr = max(c1.real, c2.real)
        let bri = min(c1.imaginary, c2.imaginary)
        self.topLeft = Complex(real: tlr, imaginary: tli)
        self.bottomRight = Complex(real: brr, imaginary: bri)
        self.bottomLeft = Complex(real: tlr, imaginary: bri)
        self.topRight = Complex(real: brr, imaginary: tli)
    }
}
```

B.2.6 Viewer.swift

```
//Stores information about the projection matrix and
// the display size.
struct ProjectionInfo : Equatable {
    let matrix  : float4x4
```

```
    let winSize : CGSize
}

class Viewer : SignalGroup {

    // The texture shader program
    let shaderProg   : FlatTexShaders
    let generator    : MandlebrotGenerator

    init(initWinSize : CGSize) {
        //Represents the shader program
        self.shaderProg = FlatTexShaders()
        let iterations = 1000
        self.generator = MandlebrotGenerator(imgSize:initWinSize,
                                             iterations:iterations)
        super.init()
        var workersData : [DSignal<MandlePoint>] = []
        for complex in generator.generateComplexNumbers() {
            let worker = MandleBrotWorker(c :complex,
                iterations:iterations)
            let mPoint = group(worker, in1:worker.mandlePoint)
            workersData.append(mPoint)
        }
        let mandlePoints = join(workersData)
        // Create the projection information containing
        // the window size and projection matrix
        let projection = hold(makeProjInfo(initWinSize),
            in1:lift(reshape, in1: TeselExternal.windowSize))

        let _ = render(flatFrame, in1:projection, in2:mandlePoints)
    }
    /*** Viewer Signal Functions ***/
    func makeProjInfo(winSize : CGSize) -> ProjectionInfo {
        let projection = Transform.ortho(left:0,
            right:Float(winSize.width),
            bottom:Float(winSize.height),
            top:0, nearZ:-1.0, farZ: 1.0)
        return ProjectionInfo(matrix:projection, winSize:winSize)
    }
    func reshape(winEvt : WinEvent) -> ProjectionInfo {
```

```
    return makeProjInfo(winEvt.size)
}
func flatFrame(projInfo : ProjectionInfo,
               points:TeselCollection<MandlePoint>) -> Frame {

    //Stores all the mesh entities for the boids
    var meshes : [MeshEntity<FlatAttributes>] = []

    guard let image = generator.renderPoints(points.rawArray) else {
        print("Could not create mandlebrot image")
        exit(EXIT_FAILURE)
    }

    //Retrieve the mesheEntity for the heightmap
    meshes.append(brotMeshEntity(projInfo, image:image))

    //Create the MeshRenderEntity that will use the texture
    // shader program for rendering
    let flatEntity = MeshRenderEntity<FlatAttributes,
                                      FlatTexShaders>(meshes:meshes,
        program:shaderProg, info:FlatTexShaders.info)

    //Provide all the render entities for frame along with
    // the clear color
    let frame = Frame(clearColor:float4(0.157,0.157,0.157,1.0),
        entities:[flatEntity])

    return frame
}
func brotMeshEntity(info : ProjectionInfo, image : CGImage)
    -> MeshEntity<FlatAttributes> {

    let transMat = Transform.translate(x:0.0, y:0.0, z:-0.7889)
    let model = Transform.scale(transMat, scaleVector:
        float3(x:Float(info.winSize.width),
            y:Float(info.winSize.height),
            z:0.0))
    let uniforms = FrameUniforms(model:model,
        projectionView:info.matrix,
        color:float4(1.0,0.0,0.0,1.0))
```

```
        let texture = FlatTextures(flatTex: Texture2D(image:image))
        let sampler = FlatSampler(flatSampler:Sampler())
        return MeshEntity<FlatAttributes>(mesh:ImageTexture.mesh,
                                          uniforms:uniforms,
                                          textures:texture,
                                          samplers:sampler)

    }
}
```

B.2.7 MandlebrotGenerator

```
public class MandlebrotGenerator
{
    let mandelRect = ComplexRect(c1:Complex(real:-2.1, imaginary:1.5),
                                 c2:Complex(real:1.0, imaginary:-1.5))
    let imgSize    : CGSize
    let blockiness : Double = 1.0
    let iterations : Int
    var colors     : [NSColor] = []
    let escape     : Double = 2.0

    init(imgSize:CGSize, iterations:Int) {
        self.imgSize    = imgSize
        self.iterations = iterations

        for iter in  0...(iterations-1) {
            let c : CGFloat = CGFloat(iter)
            colors.append(NSColor(hue: CGFloat(abs(sin(c/30.0))),
                saturation: 1.0, brightness: c/100.0 + 0.8, alpha: 1.0))
        }
    }
    func generateComplexNumbers() -> [Complex] {
        var numbers : [Complex] = []
        let width  : Double = Double(imgSize.width)
        let height : Double = Double(imgSize.height)
        let rect = NSMakeRect(0, 0, imgSize.width, imgSize.height)
        for x in 0.0.stride(through:width, by: blockiness) {
            for y in  0.0.stride(through:height, by: blockiness) {
```

```
            let complex = cvtViewCoordToComplexCoord(x: x, y: y,
                                                     rect: rect)
            numbers.append(complex)
        }
    }
    return numbers
}
func renderPoints(points : [MandlePoint]) -> CGImage? {
    let width  : Double = Double(imgSize.width)
    let height : Double = Double(imgSize.height)
    let size = NSMakeSize(imgSize.width, imgSize.height);
    let im = NSImage(size: size)
    var index = 0

    let rep = NSBitmapImageRep.init(bitmapDataPlanes: nil,
        pixelsWide: Int(size.width),
        pixelsHigh: Int(size.height),
        bitsPerSample: 8,
        samplesPerPixel: 4,
        hasAlpha: true,
        isPlanar: false,
        colorSpaceName: NSCalibratedRGBColorSpace,
        bytesPerRow: 0,
        bitsPerPixel: 0)

    im.addRepresentation(rep!)
    im.lockFocus()

    let rect = NSMakeRect(0, 0, size.width, size.height)
    let ctx = NSGraphicsContext.currentContext()?.CGContext
    CGContextClearRect(ctx, rect)
    CGContextSetFillColorWithColor(ctx, NSColor.blackColor().CGColor)
    CGContextFillRect(ctx, rect)

    for x in 0.0.stride(through:width, by: blockiness) {
        for y in  0.0.stride(through:height, by: blockiness) {
            let point = points[index]
            index += 1
            if point.iteration == -1 {
                NSColor.blackColor().set()
```

```
            }else {
                colors[point.iteration].set()
            }
            NSBezierPath.fillRect(CGRect(x: x, y: y, width: blockiness,
                height: blockiness))
        }
    }

    im.unlockFocus()

    var imageRect:CGRect = CGRectMake(0, 0, im.size.width,
                                      im.size.height)
    return im.CGImageForProposedRect(&imageRect, context: nil,
                                     hints: nil)
}
func cvtViewCoordToComplexCoord(x x: Double, y: Double,
                                rect: CGRect) -> Complex {
    let tl = mandelRect.topLeft
    let br = mandelRect.bottomRight
    let r = tl.real + (x/Double(rect.size.width)) *
        (br.real - tl.real)
    let i = tl.imaginary + (y/Double(rect.size.height)) *
        (br.imaginary - tl.imaginary)
    return Complex(real:r, imaginary:i)
}
func computeMandelbrotPoint(c : Complex) -> MandlePoint {
    var iteration = 0
    var z = Complex(real:0, imaginary:0)
    repeat {
        z = z * z + c
        iteration++
    } while (z.normal() < escape && iteration < iterations)

    return iteration < iterations ? MandlePoint(c:c,
                                                iteration:iteration)
        : MandlePoint(c:c, iteration:-1)
}
func renderMandelbrot() -> CGImage? {
    let width  : Double = Double(imgSize.width)
    let height : Double = Double(imgSize.height)
```

```
let size = NSMakeSize(imgSize.width, imgSize.height);
let im = NSImage(size: size)

let rep = NSBitmapImageRep.init(bitmapDataPlanes: nil,
    pixelsWide: Int(size.width),
    pixelsHigh: Int(size.height),
    bitsPerSample: 8,
    samplesPerPixel: 4,
    hasAlpha: true,
    isPlanar: false,
    colorSpaceName: NSCalibratedRGBColorSpace,
    bytesPerRow: 0,
    bitsPerPixel: 0)

im.addRepresentation(rep!)
im.lockFocus()

let rect = NSMakeRect(0, 0, size.width, size.height)
let ctx = NSGraphicsContext.currentContext()?.CGContext
CGContextClearRect(ctx, rect)
CGContextSetFillColorWithColor(ctx, NSColor.blackColor().CGColor)
CGContextFillRect(ctx, rect)

for x in 0.0.stride(through:width, by: blockiness) {
    for y in  0.0.stride(through:height, by: blockiness) {
        let cc = cvtViewCoordToComplexCoord(x: x, y: y, rect: rect)
        let point = computeMandelbrotPoint(cc)
        if point.iteration == -1 {
            NSColor.blackColor().set()
        }else {
            colors[point.iteration].set()
        }
        NSBezierPath.fillRect(CGRect(x: x, y: y,
            width: blockiness,
            height: blockiness))
    }
}

im.unlockFocus()
```

```
        var imageRect:CGRect = CGRectMake(0, 0, im.size.width,
                                          im.size.height)
        return im.CGImageForProposedRect(&imageRect, context: nil,
                                         hints: nil)
    }
}
```

REFERENCES

[1] Josh Abernathy and Justin Spahr-Summers. Reactivecocoa. `https://github.com/ReactiveCocoa/ReactiveCocoa`. Accessed: 2016-06-28.

[2] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, May 2005.

[3] Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative Self-adjusting Computation. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 309–322, New York, NY, USA, 2008. ACM.

[4] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28(6):990–1034, November 2006.

[5] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[6] Edward Amsden. Timeflies: Push-pull signal-function functional reactive programming. Master's thesis, Rochester, NY, USA, 2013.

[7] Apple. Grand Central Dispatch (gcd) reference, 2015. Available from `https://developer.apple.com/library/prerelease/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html`.

[8] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, August 2013.

[9] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and real-time systems. In *Proceedings of the IEEE*, pages 1270–1282, 1991.

[10] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM.

[11] Kristopher J. Blom and Steffi Beckhaus. The design space of dynamic interactive virtual environments. *Virtual Reality*, 18(2):101–116, 2013.

[12] Richard J. Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with Embedding Hardware Description Languages in HOL. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co.

[13] Harry Brundage. Neat Alogirthms - Flocking. Avaliable from url=http://harry.me/blog/2011/02/17/neat-algorithms-flocking/.

[14] Magnus Carlsson and Thomas Hallgren. *Fudgets Purely Functional Processes with applications to Graphical User Interfaces*. PhD thesis, Gteborg, Sweden, 1998.

[15] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[16] Mun Hon Cheong and Assessor Ken Robinson. Functional programming and 3d games, 2005.

[17] Gregory H. Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-value Language. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 294–308, Berlin, Heidelberg, 2006. Springer-Verlag.

[18] Gregory Harold Cooper. *Integrating Dataflow Evaluation into a Practical Higher-order Call-by-value Language*. PhD thesis, Providence, RI, USA, 2008. AAI3335643.

[19] Antony Courtney. Frappé: Functional Reactive Programming in Java. In *Third International Symposium on Pratical Aspects of Declarative Languages (PADL)*, March 2001.

[20] Antony Courtney and Conal Elliott. Genuinely Functional User Interfaces. In *Proceedings of the 2001 Haskell Workshop*, September 2001.

[21] Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18, Uppsala, Sweden, August 2003. ACM Press.

[22] Antony Alexander Courtney. *Modeling User Interfaces in a Functional Language*. PhD thesis, New Haven, CT, USA, 2004. AAI3125177.

[23] Evan Czaplicki and Stephen Chong. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 411–422, New York, NY, USA, June 2013. ACM Press.

[24] Al Danial. Cloc. `https://github.com/aldanial/cloc`. Accessed: 2016-07-25.

[25] Joey de Vries. Learn opengl. `http://learnopengl.com/`. Accessed: 2016-06-28.

[26] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental Evaluation for Attribute Grammars with Application to Syntax-directed Editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 105–116, New York, NY, USA, 1981. ACM.

[27] Colin Eberhardt. Mandelbrot Generation With Concurrent Functional Swift. `http://blog.scottlogic.com/2014/10/29/concurrent-functional-swift.html`, Oct 2014. Accessed: 2016-06-28.

[28] Jonathan Edwards. Coherent Reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 925–932, New York, NY, USA, 2009. ACM.

[29] Conal Elliott. Functional Implementations of Continuous Modeled Animation. In *Proceedings of PLILP/ALP*, 1998.

[30] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.

[31] Conal M. Elliott. Push-pull Functional Reactive Programming. In *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.

[32] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A Programming Environment for Scheme. *J. Funct. Program.*, 12(2):159–182, March 2002.

[33] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, June 1982.

[34] Thierry Gautier, Paul Le Guernic, and Löic Besnard. SIGNAL: A Declarative Language for Synchronous Programming of Real-time Systems. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 257–277, London, UK, UK, 1987. Springer-Verlag.

[35] Khronos Group. OpenGL Core Profile Specification, May 2015. Available from `https://www.opengl.org/registry`.

[36] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

[37] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the Type Inference Process. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 3–13, New York, NY, USA, 2003. ACM.

[38] P. Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 134–, Washington, DC, USA, 1998. IEEE Computer Society.

[39] Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[40] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures*, chapter Arrows, Robots, and Functional Reactive Programming, pages 159–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[41] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, April 1989.

[42] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, May 2000.

[43] Apple Inc. SceneKit Framework Guide, September 2014. Available from `https://developer.apple.com/library/ios//documentation/SceneKit/Reference/SceneKit_Framework/index.html`.

[44] Apple Inc. *The Swift Programming Language*. Apple Inc., Cupertino, CA, 2014.

[45] Apple Inc. Metal Programming Guide, December 2015. Available from `https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Introduction/Introduction.html`.

[46] Apple Inc. New playgrounds part 2 - sources - swift blog. `https://developer.apple.com/swift/blog/?id=26`, Mar 2015. Accessed: 2016-06-20.

[47] Wolfgang Jeltsch. Signals, Not Generators! In *Proceedings of the Tenth Symposium on Trends in Functional Programming*, pages 145–160, 2009.

[48] Robert W Kerbs. Gamasutra - Faculty Postmortem: Cal Poly Pomona's Game Development Course, Jul 2006.

[49] Khronos Group. Khronos Vulkan Registry, March 2015. Available from `https://www.khronos.org/registry/vulkan/#apispecs`.

[50] Daan Leijen and Erik Meijer. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-specific Languages*, DSL '99, pages 109–122, New York, NY, USA, 1999. ACM.

[51] Paul Liu, Eric Cheng, and Paul Hudak. Causal Commutative Arrows and Their Optimization. In *Proc. International Conference on Functional Programming*. ACM SIGPLAN, 2009.

[52] Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.React. Technical Report EPFL-REPORT-176887, École Polytechnique Fédérale de Lausanne, May 2012.

[53] Benoit Mandelbrot. How long is the coast of britain? statistical self-similarity and fractional dimension. *Science*, 156(3775):636–638, 1967.

[54] Microsoft. Direct3D 12 Programming Guide, May 2015. Available from `https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121(v=vs.85).aspx`.

[55] Henrik Nilsson. Functional Automatic Differentiation with Dirac Impulses. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 153–164, New York, NY, USA, 2003. ACM.

[56] Henrik Nilsson. Dynamic Optimization for Functional Reactive Programming Using Generalized Algebraic Data Types. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, ICFP '05, pages 54–65, New York, NY, USA, 2005. ACM.

[57] Henrik Nilsson, Antony Courtney, and John Peterson. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell'02)*, pages 51–64, Pittsburgh, Pennsylvania, USA, October 2002. ACM Press.

[58] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[59] Juha Paananen. What is bacon.js? `https://baconjs.github.io/`. Accessed: 2016-06-28.

[60] Sean Parent. A possible future of software development. `https://stlab.adobe.com/wiki/images/2/20/2008_07_25_google.pdf`, Jul 2008.

[61] Gergely Patai. *Functional and Constraint Logic Programming: 19th International Workshop, WFLP 2010, Madrid, Spain, January 17, 2010. Revised Selected Papers*, chapter Efficient and Compositional Higher-Order Streams, pages 137–154. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[62] Ross Paterson. A New Notation for Arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 229–240, New York, NY, USA, 2001. ACM.

[63] Marc Pouzet. Lucid Synchrone release, version 3.0 tutorial and reference manual, April 2006. Available from `http://www.di.ens.fr/~pouzet/lucid-synchrone/lucid-synchrone-3.0-manual.pdf`.

[64] Riccardo R. Pucella. Reactive Programming in Standard ML. In *IN PROCEEDINGS OF THE 1998 INTERNATIONAL CONFERENCE ON COMPUTER LANGUAGES*, pages 48–57. IEEE Computer Society Press, 1998.

[65] G. Ramalingam and Thomas Reps. A Categorized Bibliography on Incremental Computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 502–510, New York, NY, USA, 1993. ACM.

[66] John H. Reppy. *Higher-order Concurrency*. PhD thesis, Cornell University, 1992. Available as Computer Science Technical Report 92-1285.

[67] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Comput. Graph.*, 21(4):25–34, August 1987.

[68] Neil Sculthorpe. *Towards Safe and Efficient Functional Reactive Programming*. PhD thesis, Nottingham, England, UK, 2011.

[69] Sean Seefried, Manuel Chakravarty, and Gabriele Keller. *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, Vancouver, Canada, October 24-28, 2004. Proceedings*, chapter Optimising Embedded DSLs Using Template Haskell, pages 186–205. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[70] Graham Sellers, Richard S. Wright, and Nicholas Haemel. *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Addison-Wesley Professional, 6th edition, 2013.

[71] Daniel Shiffman. Flocking. `https://processing.org/examples/flocking.html`. Accessed: 2016-06-28.

[72] Michael Sperber. *Computer-Assisted Lighting Design and Control*. PhD thesis, Tbingen, Germany, 2001.

[73] Ertugrul Sylemez. Netwire. `http://hub.darcs.net/ertes/netwire`. Accessed: 2016-06-28.

[74] Zhanyong Wan and Paul Hudak. Functional Reactive Programming from First Principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 242–252, New York, NY, USA, 2000. ACM.

[75] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-Driven FRP. In *Practical Aspects of Declarative Languages (PADL'02)*, January 2002.

[76] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the atlas project. *PARALLEL COMPUTING*, 27:2001, 2000.