UNIVERSIDAD SIMÓN BOLÍVAR
DECANATO DE ESTUDIOS PROFESIONALES
COORDINACIÓN DE INGENIERÍA DE LA COMPUTACIÓN

IMPLEMENTACIÓN DE UN MOTOR DE JUEGO EN LENGUAJES PURAMENTE FUNCIONALES

Por:
José Daniel Duran Toro

Realizado con la asesoría de:
Victor Theoktisto

PROYECTO DE GRADO
Presentado ante la Ilustre Universidad Simón Bolívar
como requisito parcial para optar al título de
Ingeniero de la Computación

Sartenejas, Enero 2018

Resumen

La industria moderna de videojuegos maneja algunos de los proyectos más grandes y complejos llevados a cabos en el área de la informática, y con los riesgos financieros que estos proyectos implican, llevan a esta industria a la búsqueda de nuevas formas de aumentar la productividad, reducir errores y adaptar nuevas y cambiantes tecnologías a los ciclos de desarrollo de juegos. Sin embargo, en la actualidad esta industria sufre de ciclos de desarrollo cortos con equipos que comprendes miles de personas, llevando a código que muchas veces contiene fallas o es lo suficientemente complicado para hacer imposible su crecimiento y expansión.

El presente trabajo propone una alternativa para la producción de videojuegos en la forma de un motor de juego que permita la producción de los mismos en un lenguaje funcional, en este caso el lenguaje Haskell, para así poder hacer uso de las ventajas que este tipo de lenguajes, como la facilidad para crear código paralelo y la seguridad de tipos, para proveer así de una mejora en la productividad y calidad de los juegos creados.

En el lenguaje funcional Haskell se ha producido diversos juegos a pequeña escala, e inclusive se ha visto motores de juego simples, como por ejemplo Helm, Bogre-Banana y actionkid, pero ninguno de estos programas explota las capacidades del lenguaje como en el caso del paralelismo y, en el caso de los motores, de la seguridad de tipos que se puede obtener al usar funciones puras.

A lo largo de este proyecto, se creó un motor de juego funcional siguiendo una metodologia de desarrollo dirigida por prototipos. El prototipo creado en el desarrollo de este trabajo usa el paradigma de programación funcional reactiva como substituto del sistema entidad-componente popular en la industria de videojuegos. Este paradigma permite a los objetos del juego ser representados como cómputos puros en el contexto de Haskell, lo que permite al motor y al compilador realizar mejoras en la corrida del juego.

El motor también permite, en forma transparente al programador, ejecutar código en forma concurrente llevando a una mejora en el rendimiento de los juegos creados y así a la experiencia de juego. En resumidas cuentas, un motor de juego puede beneficiarse de usar un lenguaje funcional como base y permitir al programador enfocarse en la jugabilidad mientras que el motor garantiza una mayor correctitud del código y a la vez que hace un mayor uso de los recursos del sistema. Futuras mejoras de este motor pueden enfocarse en agregar funcionalidades en la forma de subsistemas, como un motor de física o de inteligencia artificial, o en mejorar la usabilidad.

Índice general

Índice de figuras iv

Índice de cuadros v

1 Introducción 1

2 Marco Teórico 3

2.1 Motores de videojuego . . . 3

2.1.1 Programa principal . . . 4

2.1.2 Motor gráfico . . . 4

2.1.3 Motor de audio . . . 4

2.1.4 Motor de física . . . 5

2.1.5 Inteligencia artificial . . . 5

2.2 Programación funcional . . . 5

2.2.1 Funciones de primera clase y de orden superior . . . 6

2.2.2 Funciones puras . . . 6

2.2.3 Evaluación demorada (lazy evaluation) . . . 7

2.2.4 Sistemas de tipos . . . 7

2.3 Programación Funcional Reactiva . . . 8

2.4 Lenguaje de especificaciones para sistemas híbridos Yampa . . . 8

2.4.1 Señales . . . 8

2.4.2 Funciones de señal . . . 9

2.4.3 Combinadores de funciones de señal . . . 9

2.4.4 Interruptores . . . 9

2.4.5 Ejemplo de uso de Yampa . . . 11

3 Marco Metodológico 13

3.1 Requisitos del sistema . . . 14

3.2 Limitaciones del sistema . . . 14

3.3 Especificación de requisitos . . . 14

3.4 Arquitectura . . . 15

3.5 Desarrollo . . . 15

3.6 Documentación 15

4 Implementación **16**
4.1 Motor gráfico 16
4.1.1 Manejador de ventana 16
4.1.1.1 GLUT como monad 17
4.1.2 Carga de recursos 17
4.1.2.1 Mallado poligonal 18
4.1.2.2 Imágenes 18
4.1.3 Shaders 18
4.1.4 Materiales 18
4.1.5 Cámara 18
4.1.6 Despliegue gráfico 19
4.2 Motor lógico 19
4.2.1 Tipos de datos 20
4.2.2 Escenas 21
4.2.3 Utilidades 24

5 Resultados **25**
5.1 Ejemplos del motor gráfico 25
5.2 Ejemplo motor de juego 26

6 Conclusiones **29**

Índice de figuras

2.1 Yampa-arr 10
2.2 Yampa-first 10
2.3 Yampa-composition 10
2.4 Yampa-loop 11
2.5 Yampa-switch 11

3.1 Metodología de desarrollo por prototipos 13

4.1 Ciclo de juego 23

5.1 Ejemplo 1 - Escena simple del motor de gráfico 25
5.2 Ejemplo 2 - Escena más compleja del motor de gráfico 26
5.3 Ejemplo 3 - Escena del motor de juego 27

Índice de cuadros

5.1 FPS en corridas de Ejemplo 2. 26

CAPÍTULO 1

Introducción

Los avances recientes en la informática han visto a los lenguajes funcionales conducir a una mejor productividad en muchas industrias, y la industria de los videojuegos y medios interactivos también se pueda beneficiar de estos avances con el uso de lenguajes funcionales. Los lenguajes de programación funcionales ofrecen muchas ventajas comparadas con los lenguajes imperativos que son ampliamente utilizados en esta industria. Los lenguajes funcionales son mucho más concisos en comparación con los lenguajes imperativos, y está probado, que en ciertas ocasiones, los lenguajes funcionales muestran un mejor rendimiento que sus contrapartes imperativas. También permiten el uso de poderosas abstracciones que pueden ser utilizadas para mejorar la estructura y modularidad del código. Los lenguajes funcionales también permiten el polimorfismo que promueve la reutilización del código y menos redundancia en los programas.

Es por estas razones que este proyecto se propone la creación de un framework que permita la construcción de juegos en el lenguaje *Haskell*, a través de una interfaz que permita, de manera transparente al programador, implementar la funcionalidad requerida para su propio proyecto sin tener que preocuparse por la lógica y funcionalidad subyacente a todo videojuego, que en el lenguaje *Haskell* presenta sus propios retos y complicaciones. Algunas aplicaciones que esta herramienta puede presentar a diversos campos son:

- Industria: un entorno funcional ayudaría a reducir los errores en el código final y reducir el tiempo de producción.
- Entretenimiento: esta herramienta podría ofrecer un avance en la calidad y cantidad de la producción de medios interactivos.
- Educación: esta herramienta puede ser usada para la visualización y modelado de simulaciones, debido a su similitud con los videojuegos.

Conceptualmente, alejarse de un lenguaje imperativo (Java, C#, C++, etc.) nos ayudará a mejorar la capacidad de comprensión del código, ya que la Programación funcional se centra en el problema en sí. Es decir, la programación funcional se centra en qué hacer en lugar de cómo hacerlo. Proporcionando una mejor abstracción para la resolución de problemas. Eso puede sonar como un pequeño beneficio por el esfuerzo que implica cambiar la forma en que abordamos el código. Pero cuando nos damos cuenta de que tener una mejor comprensión del código nos lleva a una gran

mejora en la depuración y mantenimiento, podemos ver claramente cómo se traduce en ahorros de tiempo y dinero que afectarán el éxito general del proyecto. Hoy en día, varias startups dependen de cuán rápido se puede llegar a una solución de trabajo, qué tan fácil es escalar desde allí y cómo se pueden hacer esas cosas con la menor cantidad de dinero posible.

A través del paradigma de Programación Funcional ganamos elegancia y simplicidad, descomposición más fácil de los problemas y código más estrechamente relacionado con el dominio del problema. Esto también nos conduce a pruebas unitarias simples y directas, depuración más sencilla y concurrencia simple.

Además, la adopción inevitable de CPUs con docenas de núcleos, y por lo tanto la creciente importancia de la programación no secuencial, solo acelerará el aumento en la adopción de la Programación Funcional, a medida que los viejos modelos de paralelismo traen complejidad que hace imposible razonar sobre los programas.

Con la realización de este trabajo se logró mostrar que un framework de juego programado en lenguaje *Haskell* puede ser usado para la producción de videojuegos, el framework creado provee una manera transparente para aprovechar el uso de concurrencia en procesadores modernos y proveer la capacidad de expresar la lógica del juego en forma funcional.

A lo largo de este trabajo se explica con mayor detalle las características que muestran los motores de juego y los lenguajes funcionales, la metodología usada en el desarrollo del framework, los detalles de implementación y uso del framework y finalmente se procederá a analizar el desempeño del mismos mediante programas de prueba. El *Capítulo 2* de este trabajo trata los temas teóricos de importancia para la comprención de este trabajo. El *Capítulo 3* se enfoca la metodología utilizada en la producción del programa. El *Capítulo 4* expone la implementación y características finales del programa. El *Capítulo 5* muestra los resultados obtenidos del uso de juegos implementados usando el programa creado. Finalmente el *Capítulo 6* expone las conclusiones sobre usar un lenguaje funcional para motores de juego.

CAPÍTULO 2

Marco Teórico

Esta sección muestra los diferentes conceptos y ventajas de diferentes tecnologías. Ya que la creación e videojuegos y sistemas interactivos están actualmente dominados por una metodología imperativa contraria a la programación funcional, este capítulo abarca ambos temas en forma separada para luego establecer conexión en capítulos posteriores.

2.1. Motores de videojuego

Un motor de videojuego es un término que hace referencia a una serie de rutinas de programación, frameworks u otras herramientas, que permiten el diseño, la creación y la representación de un videojuego. La funcionalidad básica de un motor es proveer al videojuego de un motor de renderizado para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, scripting, animación, inteligencia artificial, redes, streaming, administración de memoria y un escenario gráfico. El proceso de desarrollo de juegos es a menudo economizado, en gran parte, mediante la reutilización/adaptación del mismo motor de juego para crear diferentes juegos, o para facilitar la portabilidad de juegos a múltiples plataformas [**?**].

Algunos de los motores de juego más usados en la actualidad, como lo son Source, Unity, Unreal Engine, GameMaker: Studio y CryEngine, son conocidos por ser imperativos, siendo sus funciones más importantes implementadas en lenguaje C++, lenguaje moderno conocido por ayudar en la ejecución de grandes proyectos.

El factor más importante que diferencia un motor de juego de un juego está en que el motor está diseñado con una arquitectura enfocada en los datos. Mientras que un juego contiene una lógica o reglas del juego hard-coded, o emplea un código de caso especial para representar tipos específicos de objetos del juego, se vuelve difícil o imposible reutilizar ese software para hacer un juego diferente. El motor de juego permite el reutilizar de gran parte del código para varios juegos diferentes en una forma modular, donde un programador solo se encarga de programar la lógica de su juego.

Todo juego es por naturaleza una aplicación multimedia, y un motor de juego es responsable de recibir y mantener todos estos recursos (assets en inglés) que vienen en la forma de mallados 3D, bitmaps de texturas, animaciones, audio y cualquier otro elemento que el juego requiera. Todo motor de juego moderno debe de ser capaz de leer recursos multimedia de los diferentes formatos

de las aplicaciones usadas por los artistas y usar esos recursos en el subsistema adecuado para su reproducción.

Por último, y no menos importante, el motor de juego debe de hacerse cargo de administrar los diferentes recursos de la máquina de la manera más eficiente posible. Debe mantener recursos y objetos en RAM, administrar el acceso a disco duro, manejar el ciclo de dibujo del GPU y administrar la ejecución del juego en el CPU, que con el auge de CPUs modernos con varios hilos de cómputo, hace necesario el uso de estructuras de datos y de una arquitectura especial para poder hacer mejor uso del CPU y brindar una mejor experiencia de juego [**?**].

2.1. Programa principal

La lógica del juego a crear debe de ser implementada en algún algoritmo, de forma independiente del audio u otro componente. Los motores modernos de juego usan un patrón conocido como sistema entidad-componente (ECS) en donde una entidad consiste en uno o más componentes y todos los objetos en el juego son una entidad [**?**].

Una entidad es un objeto de propósito general, y suele consistir de no más que un identificador único que lo diferencia a la entidad de otras. Los componentes son el conjunto de información requerida para el funcionamiento de un aspecto del objeto y su interacción con el mundo. Finalmente cada sistema corre de manera continua e interactúa con cada entidad de posea el componente correspondiente.

Con el patrón ECS, un ejemplo común de su uso sería un sistema encargado de dibujar en pantalla interactúa con todos las entidades que tenga, por ejemplo, las componentes de visibilidad y física, estos componentes se encargarían de indicar al sistema como y donde dibujar el objeto.

En los motores de juegos con ECS normalmente se implementa la lógica del juego permitiendo al usuario implementar un componente que será administrado por un sistema que actualizara el componente cada cierto tiempo. En estos sistemas la comunicación entre componentes y sistemas es compleja y varia en varias implementaciones.

2.1. Motor gráfico

El motor de gráficos genera gráficos y animaciones 2D o 3D mediante algún método de dibujo.

En lugar de programarse y compilarse para ejecutarse en la CPU o GPU directamente, la mayoría de los motores de gráficos se construyen sobre una o múltiples interfaces de programación de aplicaciones (API), como Direct3D u OpenGL siendo las más comunes, que proporcionan una abstracción de software de la unidad de procesamiento de gráficos (GPU).

2.1. Motor de audio

El motor de audio es el componente que consiste en algoritmos relacionados con el sonido. Puede calcular cosas en la CPU o en un ASIC dedicado. Las API de abstracción, como OpenAL, audio SDL, XAudio 2, Web Audio, etc. están disponibles y simplifican la implementación de estos sistemas.

2.1. Motor de física

Un motor de física es un software que proporciona una simulación aproximada de ciertos sistemas físicos, como la dinámica de cuerpos rígidos (incluida la detección de colisiones), la dinámica del cuerpo blando y la dinámica de fluidos, para ser usado en diferentes dominios como gráficos por computadora, videojuegos y películas.

En general, hay dos clases de motores de física: en tiempo real y de alta precisión. Los motores de física de alta precisión requieren más potencia de procesamiento para calcular la física de mayor precisión y, por lo general, se utilizan para fines científicos y en películas animadas por computadora. Los motores de física en tiempo real usan cálculos simplificados y precisión disminuida para dar una respuesta en tiempo real, y son usados en videojuegos y otros medios de computación interactiva.

En la mayoría de los juegos de computadora, la velocidad de los juegos son más importantes que la precisión de la simulación. Esto conduce a diseños para motores de física que producen resultados en tiempo real pero que replican la física del mundo real solo para casos simples y típicamente con alguna aproximación. Los motores de física para videojuegos generalmente tienen dos componentes principales, un sistema de detección y respuesta a colisiones y el componente de simulación dinámica responsable de resolver las fuerzas que afectan a los objetos simulados [**?**].

2.1. Inteligencia artificial

La inteligencia artificial se usa para generar comportamientos sensibles, adaptativos e inteligentes principalmente en personajes de juego (NPC), a manera de simular a la inteligencia humana y proporcionar a los jugadores un mejor entretenimiento. Dado que la inteligencia artificial para juegos se centra en aparentar un comportamiento inteligente, su enfoque es muy diferente a la inteligencia artificial tradicional [**?**].

Debido a la complejidad involucrada en estos algoritmos y la necesidad de hacer que estos provean resultados en tiempo real a medida que el jugador interactúa con los elementos del juego, y dado también el hecho que la mayoría de los juegos suelen tener los mismos o similares requerimientos en cuanto inteligencia artificial, los motores de juego modernos proveen un conjunto de estos algoritmos optimizados para ciertas circunstancias, además de permitir a estos algoritmos correr en forma paralela dentro del motor cuando su resultado no se requiere en forma urgente y poder interrumpirlo cuando se requiera realizar una tarea más importante.

2.2. Programación funcional

Es un paradigma de programación, un estilo de construcción de la estructura y elementos de programas informáticos, que trata el cálculo como la evaluación de funciones matemáticas y evita el cambio de estado y datos mutables. Es un paradigma de programación declarativa, lo que significa que la programación se hace con expresiones o declaraciones. En código funcional, el valor de salida de una función sólo depende de los valores de entrada recibidos por la función al momento

de ser llamada, por lo que llamar a una función f dos veces con el mismo valor para un argumento x producirá el mismo resultado f (x) cada vez. La eliminación de efectos secundarios, es decir, cambios en el estado que no dependen de las entradas de función, puede hacer mucho más fácil comprender y predecir el comportamiento de un programa, que es una de las motivaciones clave para el desarrollo de la programación funcional.

La programación funcional tiene su origen en el cálculo lambda, un sistema formal desarrollado en la década de 1930 para investigar la computabilidad, el problema de Entscheidung, la definición de funciones, la aplicación de funciones y la recursión. Muchos lenguajes de programación funcional pueden ser vistos como elaboraciones sobre el cálculo lambda.

En la programación funcional, los programas se ejecutan evaluando expresiones, en contraste con la programación imperativa, donde los programas se componen de operaciones que cambian el estado global cuando se ejecutan. Adicionalmente, lo que en programación imperativa se conoce como funciones difiere al significado matemático de funciones, estas poseen un comportamiento de subrutinas en donde se puede modificar el estado global y un valor de retorno no siempre es necesario [**?**].

2.2. Funciones de primera clase y de orden superior

Funciones de orden superior son funciones que pueden tomar otras funciones como argumentos o devolverlos como resultados. Las funciones de orden superior están estrechamente relacionadas con las funciones de primera clase, en las cuales las funciones de orden superior y las funciones de primera clase pueden recibir como argumentos y generar como resultados otras funciones. Las funciones de orden superior describen un concepto matemático de funciones que operan sobre otras funciones, mientras que las funciones de primera clase son un término informático que describe las entidades del lenguaje de programación que no tienen ninguna restricción de su utilización [**?**].

Las funciones de orden superior permiten la aplicación parcial, una técnica en la que se aplica una función a sus argumentos uno a la vez, con cada aplicación devolver una nueva función que acepta el siguiente argumento. Esto le permite a uno expresar, por ejemplo, la función sucesor como el operador de suma aplicada parcialmente al número natural uno.

Los ejemplos más comunes de este tipo de funciones son map, filter y fold, que son funciones que toman como argumento otras funciones.

2.2. Funciones puras

También denominadas expresiones, son funciones que no tienen ningún efecto secundario, como alterar memoria o realizar operaciones de entrada o salida (E/S). Esto significa que las funciones puras tienen varias propiedades útiles, muchas de las cuales pueden ser utilizadas para optimizar el código [**?**].

Si no se utiliza el resultado de una expresión pura, se puede eliminar sin afectar a otras expresiones. El resultado es constante con respecto a la lista de parámetros, es decir, si la función pura se

llama de nuevo con los mismos parámetros, el mismo resultado será devuelto, esto puede habilitar las optimizaciones de almacenamiento en caché. Si no hay una dependencia de datos entre dos expresiones puras, entonces su orden puede ser invertido, o pueden llevarse a cabo en paralelo.

Las funciones puras son especialmente útiles cuando se trabaja en proyectos conjuntos o colaborativos, ya que permite una metodología de desarrollo por caja negra, en donde el código desarrollado por una persona no se vea afectado por cambios en otras rutinas, disminuyendo errores y permitiendo la realización de pruebas aisladas de este tipo de funciones.

Algunos ejemplos de este tipo de función son las funciones aritméticas, como la suma y la resta, estas funciones siempre retornan el mismo valor para una misma entrada.

2.2. Evaluación demorada (lazy evaluation)

La evaluación demorada, más comúnmente conocida como evaluación perezosa, es un mecanismo de evaluación de llamada por necesidad que demora la evaluación de una expresión hasta que se necesite su valor. En los lenguajes funcionales, esto permite estructuras como listas infinitas, que normalmente no estarían disponibles en un lenguaje imperativo donde la secuencia de comandos es significativa [**?**].

Los argumentos a una función no se evalúan a menos que se utilicen realmente en la evaluación del cuerpo de la función. Ello ofrece un gran potencial para optimizaciones de parte de los compiladores y la posibilidad de evitar realizar ciertas operaciones.

Los lenguajes de evaluación demorada permiten la definición de estructuras de datos infinitas, algo que es mucho más complicado en un lenguaje de evaluación estricta. Por ejemplo, considera una lista con los números de Fibonacci. Está claro que no podemos realizar cálculos sobre una lista infinita en un tiempo razonable, o guardarla en memoria. Como el lenguaje es de evaluación demorada, solo las partes necesarias de la lista que son usadas realmente por el programa serán evaluadas. Esto permite abstraer muchos problemas y verlos desde una persepectiva de más alto nivel.

Adicionalmente, en combinación con funciones de orden superior, la evaluación demorada permite un mayor nivel de modularización del código de los programas, haciéndolos más cortos y fáciles de escribir mejorando así la productividad [**?**].

2.2. Sistemas de tipos

El uso de tipos de datos algebraicos y la coincidencia de patrones hace que la manipulación de estructuras de datos complejas convenientes y expresivos, la presencia de comprobaciones estrictas de tipos en tiempo de compilación hace que los programas sean más fiables, mientras que la inferencia de tipos libera al programador de la necesidad de declarar manualmente los tipos para el compilador al momento de declarar funciones u otras expresiones [**?**].

2.3. Programación Funcional Reactiva

La programación reactiva funcional (FRP, por sus siglas en inglés) es un enfoque elegante para especificar de forma declarativa los sistemas reactivos, que son sistemas orientados en la propagación de cambios de múltiples entidades.

FRP integra la idea de flujo de tiempo y composición de eventos en la programación puramente funcional. Al manejar el flujo de tiempo de manera uniforme y generalizada, una aplicación obtiene claridad y fiabilidad. Así como la evaluación perezosa puede eliminar la necesidad de estructuras de control complejas, una noción uniforme de flujo de tiempo soporta un estilo de programación más declarativo que oculta un complejo mecanismo subyacente. Esto proporciona una manera elegante de expresar la computación en dominios como animaciones interactivas [**?**], robótica [**?**], visión por computadora, interfaces de usuario [**?**] y simulación.

Las implementaciones más comunes de *FRP* para *Haskell* hacen uso de la notación de flechas, que son una nueva manera abstracta de visualizar los cómputos, creada por John Hughes [**?**]. Las flechas, al igual que los monads, proveen una estructura común para la implementación de librerías siendo más generales que los monads. John Hughes demostró que existen tipos de datos que no se adaptan bien a la estructura de monads causando así fugas de memoria indeseadas, que con flechas pueden ser resueltas en lenguajes funciones como *Haskell* [**?**]. Adicionalmente las librerías de *FRP* en *Haskell* usan una extensión del lenguaje propuesta por Ross Paterson [**?**] que hace el uso de flechas mucho más fácil y conveniente que la versión original creada por Hughes. Para más detalles de la evolución de *FRP* leer “Elm: Concurrent FRP for Functional GUIs” [**?**] capítulo 2.1.

En su más simple forma, *FRP* tiene dos abstracciones principales: Comportamientos, que cambian continuamente y Eventos que suceden en diferentes puntos en el tiempo. La continuidad del tiempo hace que estas abstracciones se puedan componer.

2.4. Lenguaje de especificaciones para sistemas híbridos Yampa

Yampa es un lenguaje embebido para la programación de sistemas híbridos (tiempo discreto y continuo) utilizando los conceptos de Programación Reactiva Funcional (FRP). *Yampa* está estructurado con flechas, que reducen en gran medida la posibilidad de introducir fugas de espacio y tiempo en sistemas reactivos que varían en el tiempo [**?**]. *Yampa* es un sistema diseñado en base a una noción de muestreo y cambios dirigidos por eventos, es decir, la generación de un evento (como el presionar una tecla) causa la evaluación de valores y cambios en el estado actual del programa.

2.4. Señales

En *FRP* una señal puede representar cualquier valor mutable. Estas pueden ser transformadas y combinadas, que a diferencia de otras soluciones más tradiciones, permite abstraer varios detalles menores, permitiendo al programador lograr un mismo resultado usando menos código. En *Yampa* una señal es una función que retorna un valor según el tiempo transcurrido, dicho en otras palabras,

es una función que dado un tiempo determinado regresa el valor adecuado para el objeto mutable en el momento indicado. Un ejemplo es la posición del ratón [**?**] [**?**].

$$Signal\ \alpha \approx Time \rightarrow \alpha \tag{2.1}$$

2.4. Funciones de señal

Es una función que recibe una señal y produce otra señal. Estas funciones permiten alterar o generar nuevas señales dependiendo del estado actual de la señal original, por ejemplo, puede ser deseable generar una señal nueva cuando el ratón pasa por sobre algún elemento gráfico en específico [**?**].

$$SF\ \alpha\ \beta \approx Signal\ \alpha \rightarrow Signal\ \beta \tag{2.2}$$

Las funciones de señal son evaluadas de forma discreta a medida pasa el tiempo, al ser evaluadas retornan su valor de trabajo actual. Las SF pueden ser vistas como el comportamiento de un objeto, que cambia a medida que transcurre el tiempo.

2.4. Combinadores de funciones de señal

Similar a la composición de funciones, las SF pueden ser compuestas para permitir un mejor filtrado de las señales. En *Yampa* los combinadores básicos son:

- arr: crea un SF a partir de una función *Figura 2.1*.
- first: aplica una SF solo al primer elemento de la entrada y deja el resto sin cambios *Figura 2.2*.
- composición: compone dos SF, la señal resultada de la primera SF es dada como entrada a la segunda SF *Figura 2.3*.
- loop: crea una SF que usa su propia salida para calcular su salida, que es posible gracias a la evaluación perezosa de los lenguajes funciones *Figura 2.4*.

Partiendo de estos combinadores se pueden derivar todos los combinadores más complejos usados por la librería *Yampa*.

2.4. Interruptores

Los interruptores (*Figura 2.5*) son elementos que permiten que las SF sean comportamientos capaces de cambiar en base a eventos externos además de cambiar con el tiempo. Los interruptores dan a las SF propiedades similares a la de una máquina de estados, donde los eventos causan una transición de un comportamiento a otro [**?**].

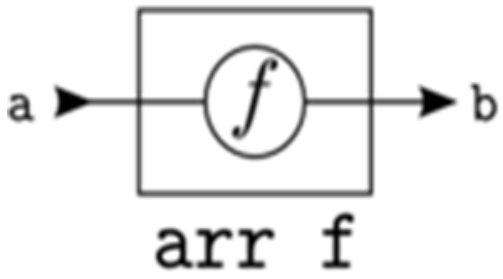

Figura 2.1 Torna una función en SF.

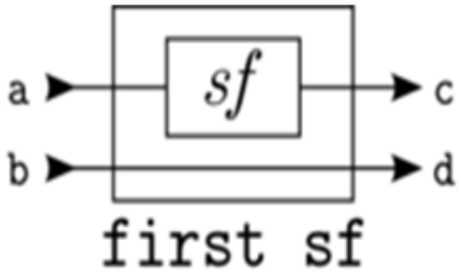

Figura 2.2 Aplica un SF al primer elemento.

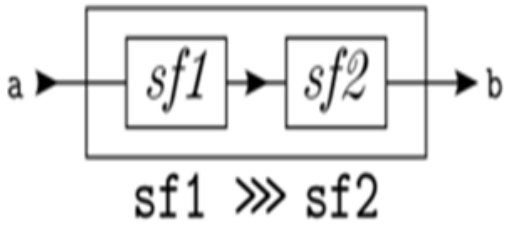

Figura 2.3 Compone dos SF.

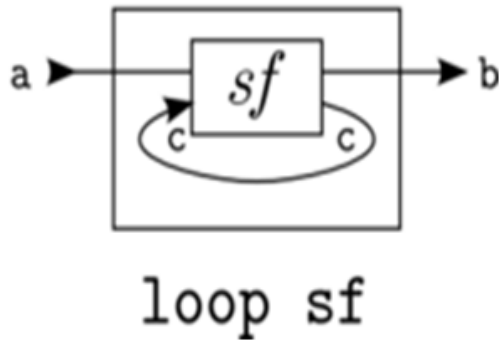

Figura 2.4 Yampa loop

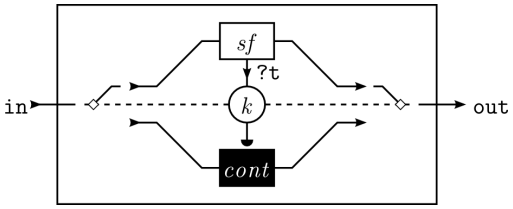

Figura 2.5 Una SF cambia a una continuación al dispararse una condición k.

2.4. Ejemplo de uso de Yampa

Las funciones de señal se pueden crear y manipular a través de la interfaz de Arrow. Por ejemplo, una transformación pura (a ->b) se convierte en una función de señal simplemente con arr de la clase Arrow. Aquí hay una función de señal que eleva a dos los valores que pasan a través de ella:

```
square :: SF Double Double
square = arr (^2)
```

La función de utilidad **embed** se puede usar para probar funciones de señal:

```
embed square (1, [(0, Just 2), (1, Just 3)])
[1.0,4.0,9.0]
```

La firma de la función **embed** se ve así:

```
embed :: SF a b −> (a, [(DTime, Maybe a)]) −> [b]
```

El primer argumento es la función de señal a muestrear. El segundo es una tupla que consiste en la entrada inicial a la función de señal y una lista de tiempos de muestra, posiblemente acompañada de un nuevo valor de entrada.

Un evento que contiene un valor de tipo a está representado por el Evento a:

```
data Event a = Event a | NoEvent
```

Una fuente de evento es una función de señal con algún tipo SF a (Evento b). Algunos ejemplos de fuentes de eventos son los siguientes:

```
never :: SF a (Event b)
now :: b -> SF a (Event b)
after :: Time -> b -> SF a (Event b)
```

Como el origen del evento es solo un SF normal, podríamos generar eventos con arr. Sin embargo, entonces debemos tener cuidado de que el evento se emita solo una vez: debido a que las uniones de señal son continuas, a menos que el evento se suprima en muestreos posteriores, podría ocurrir más de una vez.

Para reaccionar en eventos, necesitamos interruptores. El interruptor básico de una sola vez tiene el siguiente tipo:

```
switch :: SF a (b, Event c) -- SF por defecto
          -> (c -> SF a b) -- SF despues del evento
          -> SF a b
```

La siguiente señal produce la cadena foo durante los primeros 2 segundos, después de lo cual se dispara un evento y se produce bar:

```
switchFooBar, switchFooBar' :: SF () String
switchFooBar = switch (constant "foo"&&& after 2 "bar") constant
switchFooBar' = dSwitch (constant "foo"&&& after 2 "bar") constant
```

La función dSwitch es idéntica a switch, excepto que, en el momento del evento, en la segunda el cambio es inmediato, en la primera el cambio se genera al siguiente muestreo.

```
> embed switchFooBar ((), [(2, Nothing), (3, Nothing)])
["foo","bar","bar"]
> embed switchFooBar' ((), [(2, Nothing), (3, Nothing)])
["foo","foo","bar"]
```

Ejemplos mostrados en esta sección fueron creados por Samuli Thomasson, si el lector quiere profundizar en el uso de *Yampa* leer “Haskell High Performance Programming” [**?**] capítulo 13.

CAPÍTULO 3

Marco Metodológico

A fin de mejorar la productividad en el desarrollo y la calidad del motor de juego, se hace uso de la metodología de desarrollo por prototipos. Esta metodología se caracteriza por la construcción de un prototipo, el cual es evaluado y usado en un ciclo de retroalimentación, en donde se refinan los requisitos del software que se desarrollará [**?**]. Esta metodología permite probar la eficacia de diferentes algoritmos y la forma que debería tomar la interacción humano-máquina a través de diferentes prototipos, aspectos importantes para este proyecto, debido a que los algoritmos tradicionales para motores de juegos diseñados para lenguajes iterativos no son ideales para su uso en lenguajes funcionales.

Figura 3.1 Ciclo de desarrollo de una aplicación mediante la metodología de desarrollo por prototipos.

3.1. Requisitos del sistema

Ya que el objetivo de este proyecto es la creación de un motor de juego, los requisitos ideales que este debe suplir son los mismos que cualquier otro motor de juego, tener un motor gráfico para dibujar en pantalla, permitir a los programadores implementar la lógica de sus juegos, permitir a los artistas importar recursos al juego, un motor de física que resuelva colisiones entre objetos, entre otras características comunes antes mencionadas.

La diferencia que este proyecto debe necesariamente suplir, es funcionar en un lenguaje funcional, permitiendo a los programadores de los juegos trabajar con el mismo, y hacer el mayor uso posible de los beneficios que brinda la programación funcional para hacer que los juegos producidos sean lo más eficientes posible.

3.2. Limitaciones del sistema

Ya que crear un motor de juego con todas las características que poseen los motores profesionales es una labor colosal, este proyecto limitara las funcionalidades del programa final en lo mínimo requerido para poder producir juegos funcionales.

Las capacidades mínimas elegidas para poder producir juegos son, tener un motor gráfico, poder cargar recursos para el motor gráfico, en este caso imágenes, shaders y mallados poligonales, y finalmente el motor lógico donde el programador implementará la funcionalidad del juego. Están serán tomadas como los requerimientos mínimos que el programa final deberá de proveer como motor de juego, y serán usados como guía para la producción de los prototipos durante el desarrollo.

3.3. Especificación de requisitos

El programa final tendrá como usuario tres tipos de actores, que son los artistas, interesados principalmente en poder usar sus creaciones dentro del juego, los programadores, que implementan la lógica del juego usando el motor, y los jugadores, que si bien no interactúan directamente con el motor, lo hacen indirectamente a través de los juegos creados.

El principal caso de uso de los jugadores con el motor es el uso de los juegos, en ese aspecto el motor debe de encargarse de generar juego que transmitan correctamente las intenciones del jugador al sistema del juego. Para este proyecto este caso de uso se limitara a la entrada que el jugador genere vía el teclado, el ratón o modificaciones que este realice a la ventana del juego.

En cuanto a los artistas, el motor solo requiere tener la capacidad de cargar recursos para el motor gráfico.

El actor de mayor importancia para el programa final será el programador, este debe de tener la capacidad de implementar la lógica de su juego usando programación funcional, de preferencia con funciones puras. También se requerirá de un sistema que le permita correr funciones de E/S cuando se le sea necesario sin interferir con el flujo de cómputo de la lógica del juego, de preferencia en forma asíncrona. Finalmente el programador deberá de tener una manera de recibir y hacer cálculos

con la entrada recibida del usuario y usar los recursos provistos por los artistas para dibujar en pantalla.

Adicionalmente de cubrir las necesidades de los actores, se espera que el motor haga el mayor uso posible de las facilidades ofrecidas por la programación funcional para hacer los que los juegos creados puedan hacer el mayor uso de los recursos del ordenador, especialmente en temas de paralelismo donde destaca la programación funcional.

3.4. Arquitectura

Los diferentes requerimientos del proyecto serán implementados en forma modular, para permitir que nuevos prototipos solo requieran la modificación de una sección del programa. Como el lenguaje de implementación *Haskell* posee estándares para la construcción de paquetes y librerías [**?**], estos se utilizaran para la construcción del programa.

3.5. Desarrollo

Usando la metodología de desarrollo por prototipos, se implementará módulos que busquen satisfacer los requisitos del sistema junto con prototipos que hagan uso de estos módulos. A través de estos prototipos se decidirá la satisfacción de los requisitos y se entrará en un ciclo de corrección y ajuste.

3.6. Documentación

El presente documento servirá como el principal manual para el funcionamiento interno del programa. Para documentación que indique el uso y funcionamiento de las funciones públicas implementadas por el motor, se usara el estándar de comentarios de *Haskell* para generar documentación con el programa Haddock.

CAPÍTULO 4

Implementación

Para el desarrollo de la aplicación se hizo uso del lenguaje *Haskell* versión 8.0.2 y usando Cabal versión 1.24.2.0 como manejador de paquetes. Se escoge el lenguaje *Haskell* sobre otros lenguajes funcionales por la consistencia que tiene este lenguaje a acatar de forma más estricta los conceptos asociados a la programación funcional y por ser un lenguaje compilado, lo que permite un mayor desempeño a tiempo de ejecución.

Dirigidos por la metodología de desarrollo por prototipos, se llevaron a cabo diversas iteraciones en las se expandió la funcionalidad del motor. Durante la primera iteración de trabajo se analizaron los métodos para graficar en *Haskell* y se implementó el motor gráfico. En la segunda iteración se implementó el motor lógico y se establecieron los métodos con los cuales el programador puede implementar la lógica de su juego. La tercera iteración se dedicó a la mejora del rendimiento del motor, añadiéndole la capacidad de correr los diferentes subsistemas en hilos de cómputo separados. La cuarta iteración, se añadió la capacidad para que el programador pueda correr funciones de E/S en forma asíncrona. En la quinta iteración se arreglaron desperfectos y se mejoró la interacción entre el motor y el programador según las experiencias obtenidas en los prototipos anteriores.

4.1. Motor gráfico

Este módulo debe contener las facilidades necesarias para el despliegue de gráficos en una ventana. La librería de despliegue gráfico seleccionada para el proyecto es OpenGL, que es una librería que interactúa con el GPU para brindar despliegue grafico acelerado por hardware. OpenGL tiene la ventaja de ser multiplataforma e independiente del sistema de ventanas y del sistema operativo en que corra la aplicación, permitiendo así a nuestra aplicación ser fácilmente portable entre diferentes dispositivos.

4.1. Manejador de ventana

La interfaz de OpenGL para *Haskell* hace uso de la librería escrita en lenguaje c a través de la interfaz de funciones foráneas de *Haskell*, haciendo necesario el uso de apuntadores e IO para poder interactuar con el contexto de OpenGL, siendo el objetivo de este módulo facilitar el despliegue gráfico, es importante hacer una librería que exponga una interfaz más funcional (de la misma forma en la que el monad IO oculta la naturaleza imperativa del mundo exterior) que sea más familiar a

los usuarios de lenguajes funcionales y también que la librería provea de facilidades para cargar recursos multimedia al contexto de OpenGL así como controlar el proceso de despliegue gráfico.

El primer paso para usar OpenGL es la creación del contexto, para ello se ha elegido la librería GLUT, que es una librería de utilidades para aplicaciones que utilicen OpenGL, enfocándose primariamente en E/S a nivel de sistema operativo, operaciones como la creación y control de ventanas y entrada del teclado.

Trabajos realizados con GLUT se encuentran en el módulo *EasyGLUT* **??**. La primera necesidad a la hora de usar GLUT está en definir callbacks para los diferentes eventos, entrada de mouse y teclado y cambios en el estado de la ventana. La motor gráfico provee funciones que inicializan una ventana GLUT (y consigo el contexto de OpenGL) y se definen diversos callbacks para el contexto de GLUT que procesan la entrada de la ventana y se acumula toda la entrada para ser procesada en el ciclo principal del programa.

Los callbacks definidos por el motor gráfico, permiten la detección de la entrada del usuario, que es almacenada en el tipo de dato *MouseKey 4.1.1*, que es una tupla que contiene un mapa del estado de las teclas y la posición del ratón. Usando esta información, el programador puede consultar en cualquier momento dado la entrada del jugador.

```
data KeyState = Down | Up | Pressed | Released deriving Show
data MouseState = FreeMouse GL.GLint GL.GLint | FixMouse GL.GLint GL.GLint
type MouseKey = (Map GLUT.Key KeyState,MouseState)
```

4.1. GLUT como monad

El motor gráfico también tiene la labor de manejar el ciclo principal de ejecución, y normalmente, al usar GLUT de forma convencional en *Haskell*, este ciclo corre dentro del monad IO, permitiendo al usuario hacer cualquier cosa. Como la idea de un motor gráfico es que este se encargue de todos estos aspectos de E/S, se implementó el monad GLUT para sustituir al monad IO del ciclo principal del motor gráfico, y darle acceso al programador solo de las cosas que este pueda necesitar e impidiéndole alterar cosas que el motor gráfico controle.

Visto desde un punto de vista funcional, se puede considerar a GLUT como un conjunto de cómputos que definen y controlan el estado de la ventana GLUT, así, estos cómputos pueden ser visto como un monad [**?**] [**?**] [**?**], que dentro de lenguajes funcionales facilitaría la interacción del usuario con la librería, ocultando apuntadores y demás detalles de comunicación con librería GLUT. De esta forma se introduce el monad GLUT que tiene las propiedades de poder ser consultado por eventos externos y entrada de dispositivos además de poder controlar una ventana de despliegue y un contexto de OpenGL.

4.1. Carga de recursos

Este módulo debe de proveer de herramientas para cargar recursos multimedia de formatos comunes. Esta sección tendrá un énfasis en imágenes y mallas poligonal que son los dos recursos multimedia con los cuales se puede hacer prototipos de juegos más rápidamente.

4.1. Mallado poligonal

El formato de archivos de mallado poligonal elegido para este proyecto es Wavefront .obj, que es un formato que guarda en forma simple información de geometría 3D en texto plano. Este es uno de los formatos más antiguos y es soportado por la gran mayoría de las herramientas de creación usadas por los artistas.

Ya que la información en un archivo Obj se guarda en texto plano, se crea un lexer y parser que puedan leer la información de un archivo. La información es guardada en el mismo formato en el tipo de dato Obj implementado en el módulo *EasyGL.Obj* **??**.

4.1. Imágenes

Para la carga de imágenes se usa la librería JuicyPixels [**?**] que soporta la lectura de imágenes en los formatos PNG, Bitmap, Jpeg, Radiance, Tiff y Gif. La carga de las imágenes se realiza en el módulo *EasyGL.Material* **??** al crear un material.

4.1. Shaders

El módulo *EasyGL.Shader* **??** provee facilidades para cargar y compilar archivos de shaders a OpenGL. Todos los shaders compilados con esta librería poseerán atributos adicionales para recibir la posición del vértice, junto con su normal y coordenada de textura.

Este módulo ofrece funciones para cargar shaders de archivos y compilarlos. También implementa la función withShader que permite usar un shader en OpenGL de una manera funcional.

Adicionalmente, de la misma forma en que se realizó con GLUT, se creó el monad Uniform que representa las acciones de cargar datos a los atributos de los shaders, que combinado con una clase Uniform, ayuda a garantizar que solo datos que puedan ser aceptados por el GPU sean cargados a este. Este monad premite la creación de la función withShaderSafe que permite usar un shader en forma segura ya que todo el pasaje de información al shader se hace por el monad Uniform.

4.1. Materiales

En el módulo *EasyGL.Material* **??** define el tipo de dato Material que contiene referencia a un shader ya compilado y a texturas que ya han sido cargadas a OpenGL. Este módulo solo ofrece la facilidad de cargar automáticamente texturas al shader contenido en el material.

4.1. Cámara

Se necesitan facilidades para el controlar los elementos a dibujar así como la perspectiva a usarse en los dibujos, este módulo debe ser capaz de utilizar los recursos cargados en otros módulos y producir imágenes en la ventana del juego. El módulo *EasyGL.Camera* **??** ofrece una abstracción de cámara fácil de usar, que oculta las dificultades de configurar la visión en OpenGL así como la manipulación de matrices de transformación que ellos implican.

4.1. Despliegue gráfico

Para poder hacer uso de los mallados poligonales se debe primero cargar la información de los mismos al GPU. OpenGL hace uso de objetos conocidos como Vertex Array Object (VAO) y Vertex Buffer Object (VBO) como medios para almacenar información en el GPU y dibujar un objeto en pantalla de forma rápida y eficiente. Estos objetos pueden ser usados para cargar información de la posición de los vértices, la coordenada de las texturas y las normales del objeto, que es información que podemos obtener de los archivos Obj.

Ya que la información que se almacena en VAO's y en VBO's se ordena en forma diferente a la información almacenada en los archivos Obj, se implementó en el módulo *EasyGL.IndexedModel* **??** el tipo de dato IndexedModel, que almacena la información de un mallado (vertices, coordenadas de texturas y normales) en el orden y formato que se requiere para crear VAO's y VBO's. Uan gran ventaja que provee el dato IndexedModel, es que sirve como un intermediario para cargar información al GPU, y cualquier módulo que se agregue al proyecto para cargar algún formato de objeto poligonal al sistema, solo debe de implementar un método que convierta la información en un IndexedModel. Para cargar al GPU y dibujar en pantalla un IndexedModel, se usan las funciones implementadas en el modulo *EasyGL.Entity* **??**.

Para cargar la información de los archivos Obj a OpenGL, se implementa el módulo *EasyGL.Obj.Obj2IM* **??**, que lee el archivo y transforma la información en un IndexedModel.

4.2. Motor lógico

Esta sección del programa se encargara de mantener y actualizar a los diferentes objetos del juego a medida que transcurre el tiempo. Este módulo debe de proveer al usuario las herramientas necesarias para implementar la lógica de su juego e inicializarlo al ser invocado. Para aprovechar las ventajas de la programación funcional la actualización de objetos debe de ejecutarse de una forma pura.

Permitir que el usuario implemente la lógica de su juego es de vital importancia para todo motor de juego, y como ya se mencionó en la *sección 2.1.1*, los motores de juegos modernos emplean el patrón sistema entidad-componente, este patrón de diseño en su implementación depende en gran medida de la existencia de un estado mutable, la lógica creada por el usuario inevitablemente tiene que modificar componentes de una o más entidades para poder así generar cambios en el estado del juego. Sistemas implementados usando este patrón además requieren complejos sistemas para llevar a cabo la comunicación entre diversos subsistemas, y además, como muestra Jeff Andrews [**?**], la complejidad se hace mucho mayor cuando se quiere que estos sistemas corran de forma concurrente para hacer mejor use de procesadores modernos.

La implementación de un sistema que haga uso del patrón sistema entidad-componente en un lenguaje funcional como *Haskell* es posible si se hace que todas las entidades se referencien con apuntadores y solo sean accesibles dentro del monad IO, semejante implementación podría dar los mismos resultados que en lenguajes imperativos, pero se incurriría en los mismos problemas sufridos

en estos lenguajes además de no aprovechar ninguna de las ventajas que ofrece la programación funcional como las funciones puras.

La programación funcional reactiva es un paradigma de programación que nos permite sustituir al modelo sistema entidad-componente. Para este proyecto se ha decidido usar la librería *Yampa*, que implementa *FRP* y puede ser usado para mantener y actualizar una colección de entidades en forma pura.

4.2. Tipos de datos

Siguiendo el ejemplo de visto en "The Yampa Arcade" [**?**], podemos representar los objetos del juego como una SF que recibe como entrada el estado previo del juego y la entrada del jugador y retorna el nuevo estado del objeto así como la información necesaria para interactuar con los subsistemas del motor. Así un objeto tiene la forma:

```
type Object = SF ObjInput ObjOutput
```

Esta abstracción nos permite distinguir los objetos de nuestro juego de otras SF. *ObjInput* y *ObjOutput* deben de ser definidos tomando en consideración que el usuario debe de poder usar su propio tipo de dato para representar el estado del juego y otro tipo de dato que represente los eventos que este quiera generar por su cuenta.

```
data ObjInput state eventType = ObjInput {
  oiEvents :: Event eventType,
  oiGameInput :: GameInput,
  oiPastFrame :: IL state
}
```

ObjInput es definido como un tipo de dato que contiene la entrada del juego representada en el tipo GameInput, eventos que este objeto reciba y la colección de los objetos en el frame anterior del juego.

```
data ObjOutput state eventType = ObjOutput {
  ooObjState :: !state,
  ooRenderer :: Maybe (ResourceIdentifier,Transform,Uniform ()),
  ooKillReq :: Event (),
  ooSpawnReq :: Event [Object state eventType],
  ooWorldReq :: [IOReq eventType],
  ooWorldSpawn :: [IO ()],
  ooUIReq :: [UIActions]
}
```

Cada registro del tipo *ObjOutput* almacena:

- ooObjState: el estado de salida del objeto. Contiene la información relevante a la lógica del juego y es el único campo visible por otros objetos. Es estricto para garantizar que no existan fugas de memoria.
- ooRenderer: este campo permite al objeto indicar al motor gráfico que dibujar. Si este campo contiene un Nothing entonces nada será dibujado en pantalla, de ser un Just, este contendrá una tupla con el id del mesh a usar, la posición en el espacio y los datos a enviar al shader.

- ooKillReq: indica al sistema si se debe o no destruir el objeto.
- ooSpawnReq: indica al sistema los nuevos objetos que se quiere crear.
- ooWorldReq: hace al sistema iniciar una nueva operación de E/S cuyo resultado será de tipo *eventType*. Este resultado será luego devuelto al objeto vía *ObjInput* cuando esté disponible.
- ooWorldSpawn: crea una nueva operación de E/S cuyo resultado no importa.
- ooUIReq: permite interactuar con el manejador de ventana (GLUT en este caso).

Con la entrada y salida de nuestros objetos definidos, la nueva firma del tipo objeto pasa a ser:

```
type Object outState eventType = SF
    (ObjInput outState eventType)
    (ObjOutput outState eventType)
```

Ahora que se ha definido el tipo de datos para los objetos del juego, es necesario poder crear una colección de estos objetos, esta colección requiere que cada objeto pueda ser identificado de manera única. En el módulo *Val.Strict.IL* **??** se implementa el tipo de dato *IL* (Identity List), *IL* se define como:

```
type ILKey = Integer
data IL a = IL {
    ilNext :: ILKey,
    ilAssocs :: Map ILKey a
  }
```

El tipo *IL* almacena los objetos en un mapa de la librería *Data.Map*, que este posee muy buen tiempo para insertar, eliminar, consultar y recorrer, de usar un arreglo el tiempo para la inserción, la eliminación y la consulta serían mayores por requerir que los objetos sean identificables. Cada objeto es identificado con un *ILKey* al momento de ser insertado.

Para hacer su uso lo más conveniente posible, el modulo *Val.Strict.IL* **??** define múltiples funciones de consulta y modificación con *IL*, además de incluir a *IL* en las clases *Functor*, *Foldable*, *Traversable* y *NFData*.

4.2. Escenas

Para crear una escena de juego, se requiere primero una forma de actualizar la colección de objetos, para ellos se hace uso del dpSwitch de la librería *Yampa*. Este suiche se caracteriza por manejar colecciones dinámicas de objetos, en el módulo *Val.Strict.Scene* **??** se hace uso de este suiche para crear la SF "sceneSF" que tiene la firma:

```
SF
  (GameInput, IL (ObjOutput s et), IL (Event et))
  (IL (ObjOutput s et))
```

La SF sceneSF recibe como entrada el estado de la ventana y entrada del usuario, el estado de los objetos en el frame anterior y eventos que deban ser enviados a objetos específicos, y retorna como resultado los objetos actualizados.

Con sceneSF es posible ahora implementar el ciclo principal del juego, el módulo *Val.Strict.Scene* **??** implementa la función initScene, que dado la información de los recursos a usar, una SF que controle una cámara y una colección inicial de objetos, carga todos los recursos al sistema e inicia el ciclo principal.

La función initScene inicia la escena del juego creando tres hilos de cómputo encargados de tareas específicas, en la *Figura 4.1* se puede apreciar el flujo de los datos entre los hilos. Cada hilo esta e encargado de:

- Hilo 1, hilo lógico: Este hilo se caracteriza por actualizar la colección de objetos cada frame, este hilo inicia recibiendo la entrada del usuario del hilo 2 y el retorno de las funciones asíncronas generadas por los objetos manejados por el hilo 3. Con estos dos datos se actualizan los objetos en base al tiempo transcurrido y la salida de ellos es transmitida a los otros hilos.

- Hilo 2, hilo de despliegue: Este hilo maneja el contexto de OpenGL y GLUT, inicia enviando la entrada del usuario al hilo 1 y recibe la salida de los objetos generados en el hilo 1. Con esta salida se puede generar el despliegue de gráficos.

- Hilo 3, hilo de E/S: Este hilo existe para que los objetos del juego puedan ejecutar funciones de E/S de manera asíncrona. Este hilo mantiene una lista de todas las funciones asíncronas en corrida. Este hilo inicia evaluando la culminación de alguna función y enviando los resultados al hilo 1, luego recibe la salida del hilo 1 y corre cualquier nueva función asíncrona que los objetos soliciten.

Como cada hilo solo requiere de una referencia a los resultados de los objetos del juego para correr, que son valores inmutables y por lo tanto no pueden afectar a otros hilos, se puede añadir otros hilos nuevos con otro tipo de funcionalidad como sonido, networking, inteligencia artificial, entre otros, sin alterar el funcionamiento ya implementado y sin modificar la lógica actual.

La comunicación entre los diferentes hilos se logra con el uso de *MVar*, que son apuntadores especiales de *Haskell* que tienen la propiedad de funcionar como semáforos, haciendo que los hilos solo puedan enviar información a otros una vez estos estén listos para recibirla. El uso de estas *MVar* causa, en la implementación actual, que todos los hilos se sincronicen al más lento.

El módulo *Val.Strict.Scene* **??** ademas aprovecha el hecho de que el tipo *IL* implemente la clase *Traversable*, con esta clase se puede usar la librería *Control.Parallel.Strategies* de *Haskell* para que el calculo de los diferentes objetos del juego se realice en forma concurrente manejado por el runtime environment de *Haskell*, permitiendo que los cálculos realizados en el hilo lógico puedan realizare en más de un procesador. Para poder usar esta versión del hilo lógico, el usuario solo debe cambiar la llamada de la función initScene por la función initScenePar implementada en el mismo módulo.

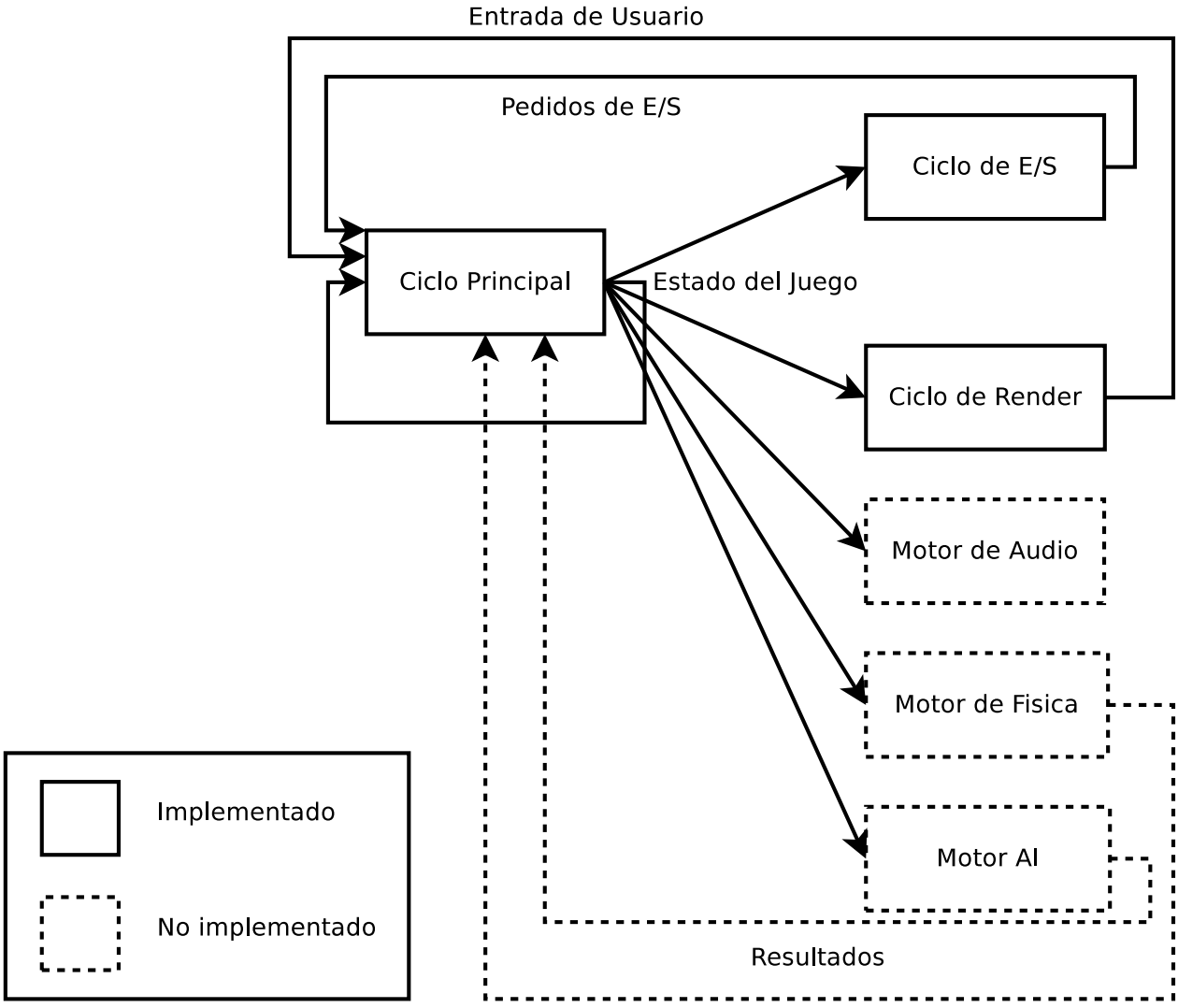

Figura 4.1 Esta figura muestra los diferentes hilos que corren en una escena de juego y a donde se envían los resultados de cada unos de ellos.

4.2. Utilidades

Ya que *FRP* es un concepto complicado (de aprender a usar), puede resultar ser una barrera para usar el motor de juego, por ello se ha creado en el módulo *Val.Strict.Util* **??** la función makeSF que tiene la firma:

```
makeSF :: a
  −> (a −> ObjInput b c −> (a,ObjOutput b c))
  −> Object b c
```

Con la función makeSF el usuario puede crear un objeto de juego sin usar *FRP* o la librería *Yampa*. Esta función toma como primer argumento un estado inicial y de segundo una función que toma ese estado, una entrada de objeto y retorna el estado actualizado y una salida de objetos.

Sin embargo esta forma de crear objetos de juego carece de la capacidad de cambiar comportamiento a medida que corre el programa que posee *FRP*, un objeto creado con esta función solo podrá tener el comportamiento de la función dada en el segundo argumento. La única manera de cambiar el comportamiento es creando un objeto nuevo y destruir el viejo.

CAPÍTULO 5

Resultados

El proyecto logró producir una herramienta que permite la creación de juegos en lenguaje *Haskell*. Para poder comprobar su funcionalidad y desempeño se hace uso de programas de ejemplo creados usando la herramienta.

5.1. Ejemplos del motor gráfico

Para la prueba del motor gráfico se crearon varios programas de prueba con escenas diferentes donde se pudiese apreciar la funcionalidad del motor gráfico. Uno de los ejemplos más sencillos es un programa que dibuja en pantalla el armadillo de Stanford usando su normal como textura, ver *Figura 5.1*. Este programa no tiene ninguna clase de interacción. Se puede ver el programa en la sección de código *Ejemplo1* **??**, el código requerido para este programa es sencillo y corto gracias al motor. Otro ejemplo, *Figura 5.2*, muestra una escena más compleja usand el motor gráfico.

Figura 5.1 El armadillo de Stanford dibujado usando su normal como color.

Figura 5.2 Una escena mas compleja usando el motor gráfico.

FPS	1 hilo	2 hilos	3 hilos	4 hilos
5 esferas	1550	1650	1820	1780
10 esferas	1350	1480	1550	1520
50 esferas	430	520	650	620
100 esferas	212	270	350	340
200 esferas	100	150	160	160
400 esferas	42	68	72	72
600 esferas	27	40	44	44

Cuadro 5.1 FPS en corridas de Ejemplo 2.

5.2. Ejemplo motor de juego

El motor de juego se prueba mediante un programa que crea un número de esferas que se mueven en alguna dirección e invierten la dirección en la que se mueven cuando colisionan con otra esfera o llegan al límite de un área predefinida. Este ejemplo se puede encontrar en la sección de código *Ejemplo2* **??**.

Para este ejemplo se observa los FPS (frames per second) del programa con diferente cantidad de esferas y permitiendo al RTE de *Haskell* correr con una cantidad diferente de hilos máximos. Esta prueba permitirá observar el rendimiento y el uso de paralelismo del motor de juego en diversas situaciones.

En la *Tabla 5.1* se observa, como era de esperarse, una disminución de los FPS a medida que se aumenta el número de entidades en la escena. Pero gracias al uso de paralelismo del motor, el uso de hilos adicionales permite aumentar el número de FPS por cantidad de entidades. Los datos muestran que el programa corre a mayor velocidad cuando se utilizan tres hilos de cómputo, en especial en los casos con pocas entidades.

En los casos con pocas entidades, la caída en FPS puede atribuirse al costo incurrido en crear y destruir hilos siendo mayor a la ganancia de procesar las entidades en hilos separados. Sin embargo, cuando se aumenta el número de entidades, los FPS con tres y cuatro hilos se hacen iguales en lugar de aumentar, en la *Sección 4.2.2* se establece que el hilo lógico usa la librería Control.Parallel.Strategies para realizar la actualización de las entidades en paralelo, el motor de juego posee, en la implementación actual, tres hilos que siempre se encuentran activos, cualquier otro hilo es creado en base a demanda. Ello significa que al usar tres hilos el RTE de *Haskell* corra la actualización de las entidades en el hilo asignado al hilo lógico, pero al haber cuatro, el RTE puede estar incurriendo en un problema en el que administrar que tareas asignar a este hilo disponible sea más caro que la tarea siendo asignada al hilo. La otra posible causa de no aprovecharse el cuarto hilo yace en que la estrategia usada de la librería Control.Parallel.Strategies no esté llevando los objetos a forma normal, en *Haskell* eso significa que un cómputo este totalmente computado y no sea una promesa, que podría ser resuelto cambiando de estrategia e implementando la clase NFData para el tipo de dato ObjOutput.

Este ejemplo muestra que el uso de hilos en el motor provee de mayor rendimiento a los juegos creados sin aumentar la complejidad del motor. Es recomendable para cualquier funcionalidad adicional que se agregue al motor, el implementarse un hilo separado que solo dependa del estado del juego.

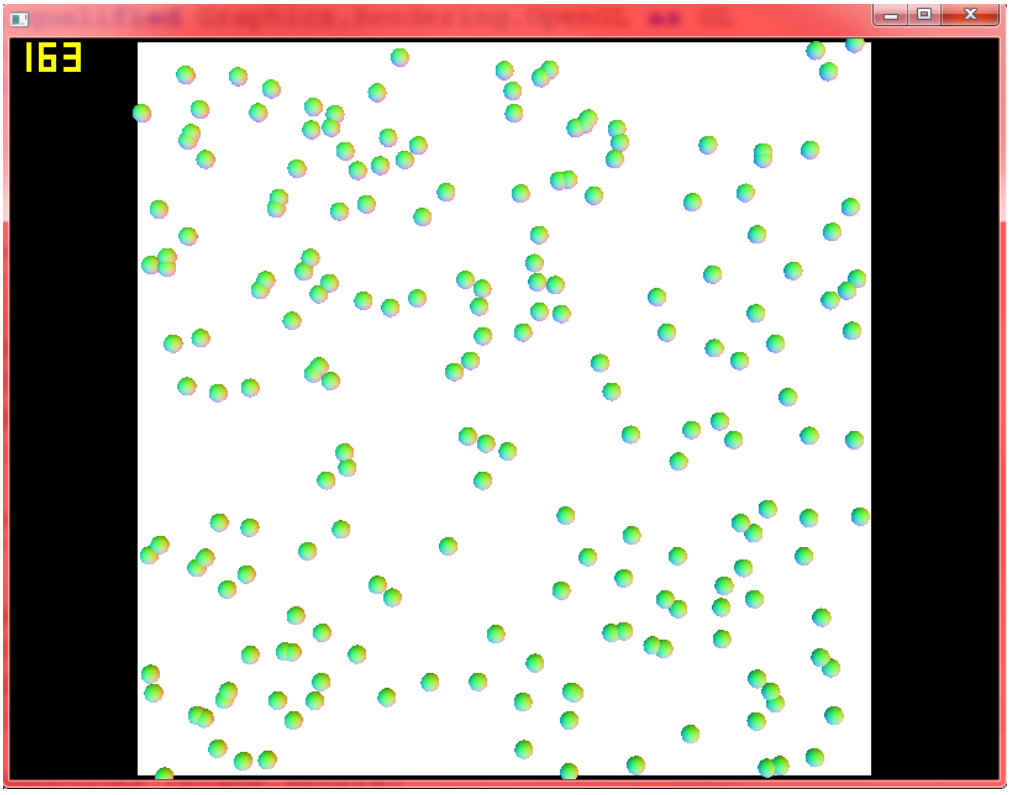

Figura 5.3 200 esferas.

Otros ejemplos fueron creados para probar el motor de juego, algunos de esos ejemplos usan algoritmos de inteligencia artificial para tener entidades capaces de interactuar con el jugador y donde el jugador puede controlar una entidad, otro ejemplo permite simplemente navegar por un espacio virtual. Sin embargo ya que el rendimiento observado en los otros ejemplos es similar al *Ejemplo2* **??** y que el código de estos ejemplos es considerablemente largo, no se hace mención detallada de estos otros ejemplos.

CAPÍTULO 6

Conclusiones

Con la realización de este trabajo se logró la creación de una herramienta que simplifica la creación de juegos en un entorno de programación funcional. La herramienta creada permite el despliegue grafico de objetos, el manejo de la entrada del usuario, pedidos de E/S y la actualización, de forma pura, entidades de juego que interactúan entre sí que conforman la escena del juego.

Un avance importante que provee la herramienta creada al conocimiento de motores de juego está en la habilidad de optimizar para el paralelismo. Como la herramienta requiere que el usuario programe las entidades del juego en forma pura, es posible ejecutar el código del usuario en un orden arbitrario sabiendo que este no interactúa con otros elementos y que su resultado no será alterado. Esta flexibilidad permite procesar la actualización de cada entidad en hilos de cómputo separados. Adicionalmente, como la información de las entidades es inmutable, se puede compartir con otros hilos de cómputo que usen la información para operaciones de entrada y salida (como graficar) de forma segura.

Esta misma pureza en la programación de las entidades, combinada con un sistema de tipos estricto como el de *Haskell*, permite que la herramienta sea útil para detectar errores de forma temprana en el código de nuestros juegos.

La estructura del motor permite que nueva funcionalidad sea fácilmente añadida, solo se tiene que pedir a las entidades que su salida provea la interfaz requerida por la nueva funcionalidad. Cualquier nueva funcionalidad, como por ejemplo un motor de física, podría fácilmente ser añadido a la herramienta para que corra en un hilo de computo independiente, ya que esta solo requeriría un apuntador a la información de las entidades y esta información es inmutable, permitiendo al motor de física realizar su función sin tener que cambiar otros sistemas para poder añadirlo.

La experiencia con el motor muestra la conveniencia de usar FRP como máquinas de estado, comparando con una implementación imperativa, como la propuesta en el libro “Artificial intelligence for games” [**?**], FRP resulta mucho más modular y re utilizable mientras que esconde la transiciones entre los estados.

También es de notar que durante la creación de juego de prueba hechos usando la herramienta, se realizaron múltiples cambios en la interfaz que permite la interacción de las entidades del juego con los diferentes sistemas de la herramienta, esto es debido a que en ciertas circunstancias ciertas formas de hacer las cosas resulta más cómodo, y con una herramienta tan joven como esta, solo su uso continuo proveerá una mejor visión de los cambios requeridos para hacer su uso más placentero

a cualquier posible usuario en un futuro, esta es una herramienta que todavía posee espacio para mejorar y ser expandida.