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, Febrero 2018

Resumen

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 promoviendo la reutilización del código y una menor 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 común en todo videojuego, que en el lenguaje *Haskell*, el cual presenta sus propios retos y complicaciones diferentes a los encontrados en lenguajes imperativos tradicionales como C.

Palabras clave Programación funcional. Haskell. Motores de juegos. Videojuegos. FRP.

Índice general

Índice de figuras

Índice de cuadros

Nomenclature

Roman Symbols

API Interfaces de programación de aplicación.

CPU Unidad de procesamiento central.

ECS Patrón Sistema Entidad Componente.

FRP Programación Funcional Reactiva - Functional reactive programming

GPU Unidad de procesamiento de gráficos.

NPC Personaje no jugable – Non player character

CAPÍTULO 1

Introducción

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.

Conceptualmente, alejarse de un lenguaje imperativo (Java, C#, C++, etc.) mejorara 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. Ello puede sonar como un pequeño beneficio por el esfuerzo que implica cambiar la forma en que abordamos el código, sin embargo, tener una mejor comprensión del código lleva a una gran mejora en la depuración y mantenimiento, traduciéndose en ahorros de tiempo y dinero que podrán afectar 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.

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.

1.1. Planteamiento del problema

Ya que la industria moderna de videojuegos requiere tiempos de producción cortos y la capacidad de generar programas grandes que se requiere que corran en tiempo real, es necesario el uso de una herramienta que permita reducir los errores en el proceso de producción y que de un acceso más transparente a los recursos del ordenador al mismo tiempo que esta herramienta produce programas eficientes. Por esta razón este proyecto presenta como propuesta la implementación de un motor de juego que haga uso de lenguajes funcionales, en este caso el lenguaje *Haskell*, ya que este lenguaje es conocido por tener la capacidad de detectar muchos errores comunes a tiempo de compilación y producir código eficiente que puede fácilmente ser corrido en forma concurrente.

1.2. Justificación e importancia

Usando el lenguaje *Haskell* como base para la producción de juegos, puede permitir la reducción de los errores del programa gracias a su sistema de tipos estrictos, reduciendo el mantenimiento que el juego pueda requerir después de su lanzamiento inicial para la corrección de errores. *Haskell* también es conocido por generar código que puede ser fácilmente paralelizable lo que puede llevar a un mejor consumo de los recursos del ordenador.

La aplicación de este proyecto presenta beneficios a diferentes areas:

- 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.

1.3. Objetivo general de la investigación

Crear una herramienta que facilite la producción de juegos mediante el uso del lenguaje *Haskell*.

1.3.1. Objetivos específicos

1. Estudiar las estructuras de datos y algoritmos necesarios para el funcionamiento de juegos desde el punto de vista de la programación funcional.
2. Estudiar las diferencias que un motor de juegos funcional presenta ante uno imperativo.

3. Crear un sistema que permita a los juegos producidos con la herramienta hacer uso de las ventajas de la programación Funcional.

4. Crear programas de prueba usando la herramienta creada que sirvan de prueba de la funcionalidad de la misma.

1.4. Alcance de la investigación

Ya que un motor de juegos contiene varios subsistemas diferentes, cada uno con problemas y requerimientos específicos, este trabajo se enfocará en facilitar la creación juegos en las areas de la lógica de los juegos y los gráficos.

1.5. Estructura del trabajo

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, los detalles de implementación y uso del motor de juego 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 y recomendaciones 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 juego 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 videojuegos es a menudo economizado, en gran parte, mediante la reutilización/adaptación del mismo motor de juego para crear diferentes videojuegos, o para facilitar la portabilidad del videojuego a múltiples plataformas [1].

Algunos de los motores de juego más usados en la actualidad, como lo son Source, Unity, Unreal Engine, GameMaker: Studio y CryEngine, hacen uso de lenguajes 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 videojuego está en que el motor está diseñado con una arquitectura enfocada en los datos. Un videojuego contiene una lógica o reglas del juego hard-coded, o emplea un código de caso especial para representar tipos específicos de objetos del videojuego, lo que hace difícil o imposible reutilizar ese software para hacer un videojuego diferente. El motor de juego permite reutilizar de gran parte del código para varios videojuego diferentes en una forma modular, donde un programador solo se encarga de programar la lógica de su juego. Esta característica hace al motor de juego la opción ideal para facilitar la creación de juegos en *Haskell*.

Todo videojuego 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 videojuego 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].

2.1.1. Programa principal

La lógica del videojuego a crear debe de ser implementada en algún algoritmo, de forma independiente del audio u otros componentes. Los motores de juego modernos usan un patrón de arquitectura conocido como sistema entidad-componente (ECS) en donde una entidad consiste en uno o más componentes y todos los objetos en el videojuego estan representados como una entidad [1].

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.

Un ejemplo común del uso del patrón ECS es el de un sistema encargado de dibujar en pantalla, este interactúa con todos las entidades que tenga, por ejemplo, las componentes de visibilidad y física, estos componentes se encargarían de indicar al sistema cómo y dónde 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 actualizará el componente cada cierto tiempo. En estos sistemas la comunicación entre componentes y sistemas es compleja y varia dependiendo de la implementación.

2.1.2. 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) [1].

2.1.3. Motor de audio

El motor de audio es el componente que consiste en algoritmos relacionados con el sonido encargado de la reproducción de sonidos generados por los diversos elementos del videojuego. Puede realizar

cálculos 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 [1].

2.1.4. 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 [1].

2.1.5. 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 [3].

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 [4].

2.2.1. 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 [4].

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 permite 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.

2.2.2. 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 [4].

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.

2.2.3. 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 [4].

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 [5].

2.2.4. 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 [4].

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 [6], robótica [7], visión por computadora, interfaces de usuario [8] 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 [9]. 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* [9]. Adicionalmente las librerías de *FRP* en *Haskell* usan una extensión del lenguaje propuesta por Ross Paterson [10] 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” [8] 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 [11]. *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.

2.4.1. 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 [11] [12].

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

2.4.2. 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 [11].

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

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.

2.4.3. 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*.

2.4.4. 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 [11].

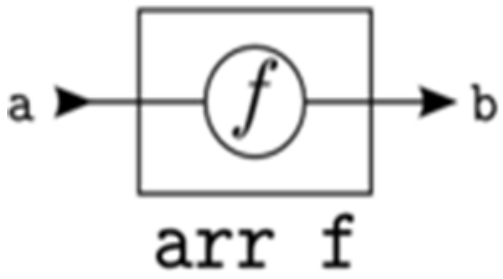

Figura 2.1 Torna una función en SF.

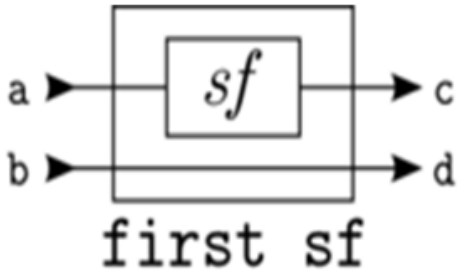

Figura 2.2 Aplica un SF al primer elemento.

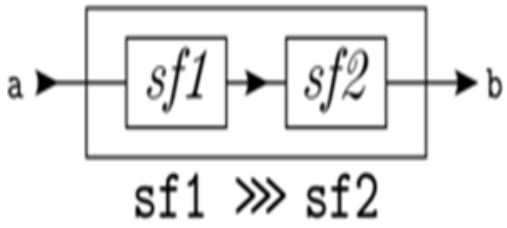

Figura 2.3 Compone dos SF.

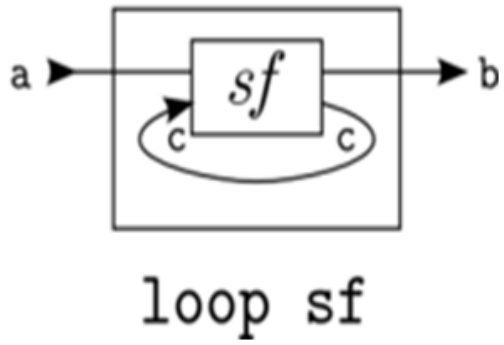

Figura 2.4 Yampa loop

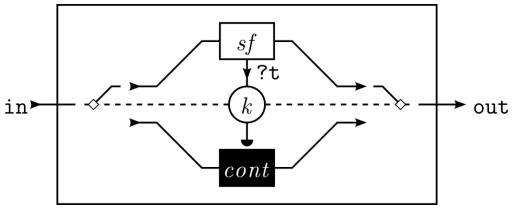

Figura 2.5 Una SF cambia a una continuación al dispararse una condición k.

2.4.5. 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" [13] 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 [14]. 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. Requerimientos del sistema

Ya que el objetivo de este proyecto es la creación de un motor de juego, este debe, al igual que otros motores 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.

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.

3.1.1. Requerimientos funcionales

- Generar videojuegos funcionales, el motor debe de ofrecer una interfaz mediante la cual los programadores que usen el motor puedan implementar la lógica de sus juegos y el motor debe de encargarse de enlazar los sistemas necesarios para generar un juego funcional.
- Administrar la entrada del videojuego, el motor debe de garantizar que la entrada generada por el jugador pueda ser correctamente procesada.
- Permitir la carga de recursos multimedia al juego, el motor debe de proveer facilidades para ingresar contenido de los artistas dentro del juego.

3.1.2. Requerimientos no funcionales

- Administrar los recursos del dispositivo, el motor debe hacer un uso eficiente de los recursos del dispositivo para proveer una mejor experiencia de juego.
- Concurrencia, el motor debe de poder hacer uso de la concurrencia para ofrecer un mejor desempeño, aprovechando las ventajas de lenguaje Haskell para este fin.

3.2. Limitaciones del sistema

Ya que crear un motor de juego con todas las características que poseen los motores profesionales es una labor de gran envergadura, 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. 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 [15], estos se utilizaran para la construcción de la herramienta.

Los videojuegos creados usaran una arquitectura en pipeline, que consiste en la transformación contínua del flujo de datos (entrada del jugador) en un resultado (gráficos del juego). Este modelo surge naturalmente en la programación funcional por su equivalencia a la composición de funciones y al flujo de trabajo encontrado en las librerías de FRP.

3.3.1. Modelado de requerimientos del sistema

La *Figura 3.2* muestra el diagrama de casos de uso para la herramienta creada. En este diagrama se observan tres actores principales para esta herramienta, los primeros siendo el artista y el programador que hacen uso directo de la misma, y el jugador, que si bien no se encuentra en contacto directo con la herramienta, interactúa en forma indirecta con la misma ya que será esta la que se encargue de la funcionalidad el juego.

La labor principal del artista es la carga de contenido multimedia al juego, labor que el programador también podrá realizar en caso de ser necesario. Para ello la herramienta ofrecerá la capacidad de cargar elementos en múltiples formatos comunes en varios otros programas de diseño.

El programador por su parte tiene la responsabilidad de diseñar los niveles/mapas del juego haciendo uso del contenido multimedia disponible. También se requiere que este implemente la lógica que todo juego requiera. Finalmente el programador tiene siempre la opción de generar un ejecutable del juego para pruebas o como versión de lanzamiento.

El jugador usar los ejecutables generados por el programador.

Otros requerimientos del sistema como la administración de la entrada y salida del juego serán manejada de forma transparente por la herramienta sin la necesidad de la intervención de los actores.

3.3.2. Componentes del sistema

Todo juego creado por la herramienta tendrá dependencias de código como visto en el diagrama de componentes *Figura 3.3*. Donde:

- OpenGL es la librería encargada del despliegue de gráficos.
- GLUT es la librería encargada de la creación de un entorno de OpenGL y el manejo de la ventana.

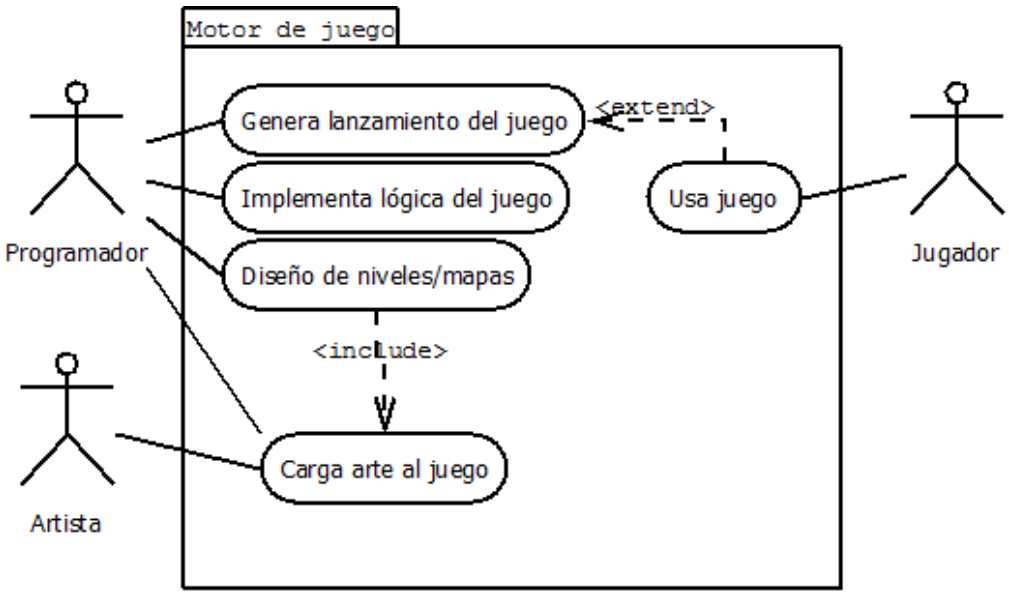

Figura 3.2 Diagrama de casos de uso

- EasyGL es una librería creada para generar una abstracción en las llamadas de las librerias OpenGL y GLUT, haciendo el uso de estas librerías más fácil en Haskell.
- El motor de juego, que se encarga de la administración de las diferentes entidades del juegousando la libreria Yampa, y actualiza los gráficos en pantalla en función del resultado de actualizar las entidades en cada frame.
- Yampa, que provee la infraestructura para la programación y actualización de las entidades del juego.
- Finalmente el código del programador proveer el comportamiento específico que cada entidad requiera para la creación del juego.

Ya que la herramienta solo genera un ejecutable que corre en forma independiente, un diagrama de despliegue no se muestra en este documento.

3.4. 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.

3.5. 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

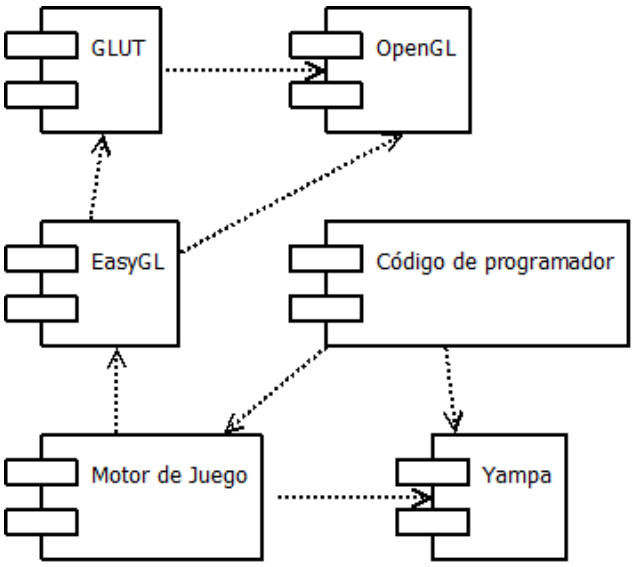

Figura 3.3 Diagrama de componentes

por el motor, se usara el estándar de comentarios de *Haskell* para generar documentación. Usando el programa ¨stack¨ sera posible generar la documentación en html usando el comando ¨stack haddock¨ en el directorio raíz del código provisto junto a este documento.

3.6. Pruebas funcionales

Para realizar pruebas continuas en cada iteración de la contrucción de la herramienta se crearon cuatro juegos de prueba usando la misma herramienta para prevenir que algún componente se viese afectado en cualquiera de las iteraciones.

1. Programa de prueba 1: juego en que cubos y esferas se mueven sobre un mapa, los objetos iguales rebotan entre sí y los diferentes se eliminan al chocar. Adicionalmente el jugador puede controlar la cámara con el ratón y teclado. Véase *Figura A.1*.

2. Programa de prueba 2: juego en que una esfera rebota sobre un plano, el jugador puede controlar la cámara con el ratón y teclado. Véase *Figura A.2*.

3. Programa de prueba 3: juego en que 200 esferas rebotan entre sí. Esta resultó la prueba más importante para medir rendimiento por la cantidad de elementos en escena. Véase *Figura A.3*.

4. Programa de prueba 4: juego en que un cono persigue a una esfera controlada por el jugador. Adicionalmente el jugador puede controlar la cámara con el ratón y teclado. Véase *Figura A.4*.

Estos programas de prueba pueden ser compilados y ejecutados corriendo los siguientes comandos desde el directorio raíz del código provisto junto a este documento:

```
stack build
stack exec test1
```

```
stack exec test3
stack exec test2
stack exec gameia
```

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.

4.1.1. 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 B.2*. 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. El 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)
```

4.1.1.1. 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.

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 [16] [17] [18], 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.

4.1.2. 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.

4.1.2.1. 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 B.8*.

4.1.2.2. Imágenes

Para la carga de imágenes se usa la librería JuicyPixels [19] 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 B.7* al crear un material.

4.1.3. Shaders

El módulo *EasyGL.Shader B.13* 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.

4.1.4. Materiales

En el módulo *EasyGL.Material B.7* 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.

4.1.5. 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 B.3* 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.

4.1.6. 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 B.6* 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 B.5*.

Para cargar la información de los archivos Obj a OpenGL, se implementa el módulo *EasyGL.Obj.Obj2IM B.10*, 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 [2], 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.

4.2.1. Tipos de datos

Siguiendo el ejemplo de visto en "The Yampa Arcade" [12], 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 B.19* 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 B.19* define múltiples funciones de consulta y modificación con *IL*, además de incluir a *IL* en las clases *Functor*, *Foldable*, *Traversable* y *NFData*.

4.2.2. 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 B.20* 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 B.20* 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 B.20* 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.

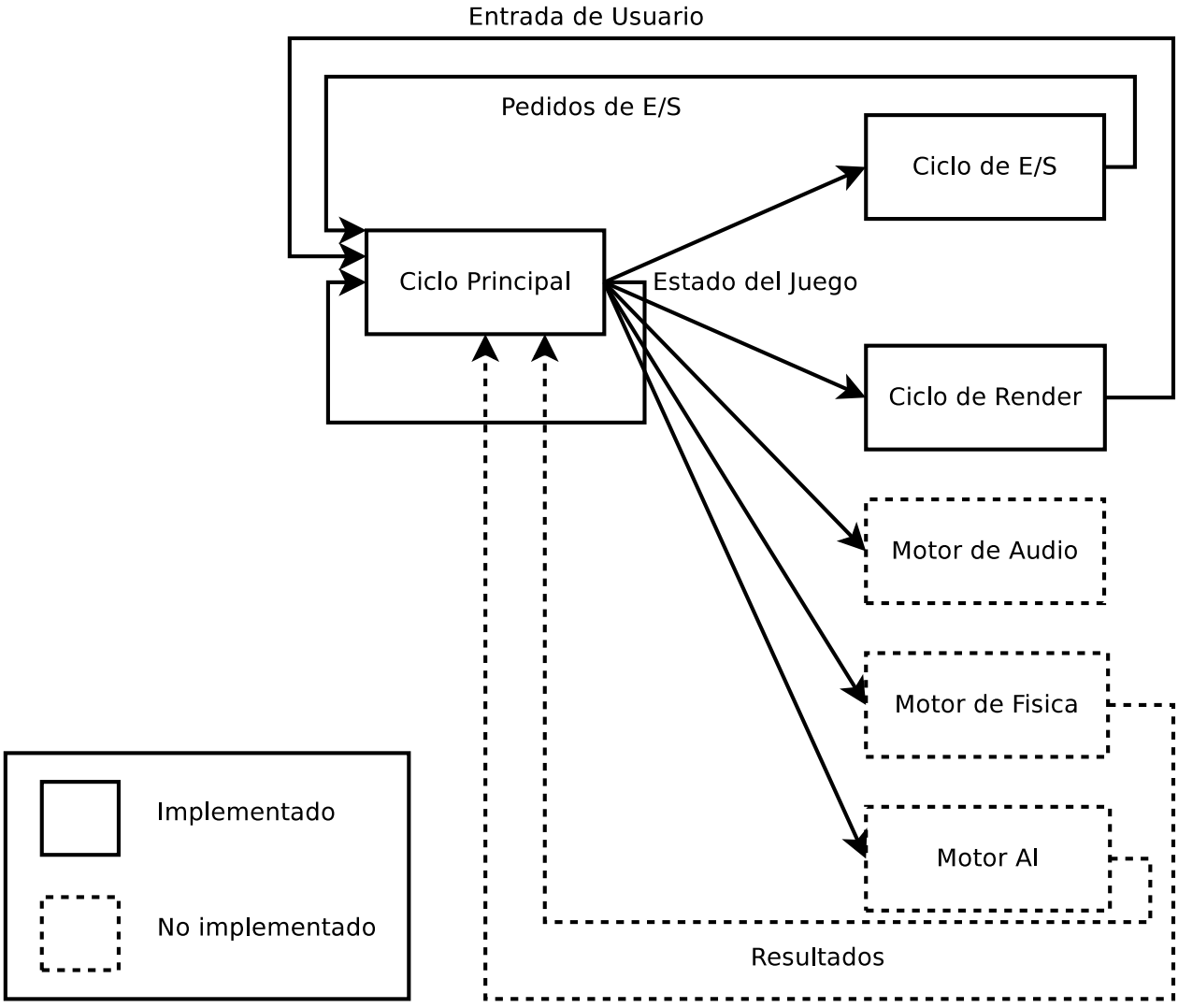

Figura 4.1 Esta figura muestra los diferentes hilos que corren en una escena de juego y a donde se envían los resultados de cada unos de ellos.

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 B.24*, el código requerido para este programa es sencillo y corto gracias al motor. Otro ejemplo, *Figura 5.2*, muestra una escena más compleja usando 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 B.25*. Véase también *Figura A.3*.

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, pero hacer esto causa el problema de forzar al programador a implementar la clase NFData en sus tipos de datos de salida.

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, esta corra en un hilo separado que solo dependa del estado del juego.

CAPÍTULO 6

Conclusiones y recomendaciones

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 gráfico 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 gráficar) 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” [3], FRP resulta mucho más modular y reutilizable 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.

Para mejorar la interacción con la herramienta se recomienda crear la creación de una interfaz gráfica para la herramienta. Con una interfaz gráfica el programador y el artista del juego podrían ahorrar tiempo y esfuerzo en ciertas labores que en la versión actual de la herramienta son tediosas e inevitables. Otra recomendación que puede hacer la herramienta mejor es la creación de una librería de FRP personalizada para los requerimientos de la herramienta.

Bibliografía

[1] J. Gregory, *Game Engine Architecture*, 2nd ed., June 2014.

[2] J. Andrews, "Designing the framework of a parallel game engine," *Articles on Intel Software Network*, vol. 109, p. 127, 2009.

[3] I. Millington and J. Funge, *Artificial intelligence for games*. CRC Press, 2016.

[4] (2014, December) Functional programming. [Online]. Available: https://wiki.haskell.org/Functional_programming

[5] J. Hughes, "Why functional programming matters," *The computer journal*, vol. 32, no. 2, pp. 98–107, 1989.

[6] C. Elliott and P. Hudak, "Functional reactive animation," in *International Conference on Functional Programming*, June 1997, pp. 163–173.

[7] I. Pembeci, H. Nilsson, and G. Hager, "Functional reactive robotics: An exercise in principled integration of domain-specific languages," in *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ser. PPDP '02. New York, NY, USA: ACM, 2002, pp. 168–179. [Online]. Available: http://doi.acm.org/10.1145/571157.571174

[8] E. Czaplicki, "Elm: Concurrent frp for functional guis," *Senior thesis, Harvard University*, 2012.

[9] J. Hughes, "Generalising monads to arrows," *Science of computer programming*, vol. 37, no. 1-3, pp. 67–111, 2000.

[10] R. Paterson, "A new notation for arrows," *ACM SIGPLAN Notices*, vol. 36, no. 10, pp. 229–240, 2001.

[11] (2017) Yampa. [Online]. Available: https://wiki.haskell.org/Yampa

[12] A. Courtney, H. Nilsson, and J. Peterson, "The Yampa arcade," in *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*. Uppsala, Sweden: ACM Press, Aug. 2003, pp. 7–18.

[13] S. Thomasson, *Haskell High Performance Programming*. Packt Publishing, 2016.

[14] S. McConnell, *Code complete*. Pearson Education, 2004.

[15] (2016, December) How to write a haskell program. [Online]. Available: https://wiki.haskell.org/How_to_write_a_Haskell_program

[16] E. Moggi, "Notions of computation and monads," *Information and computation*, vol. 93, no. 1, pp. 55–92, 1991.

[17] (2011, December) Monads as computation. [Online]. Available: https://wiki.haskell.org/Monads_as_computation

[18] (2011, December) Monads as containers. [Online]. Available: https://wiki.haskell.org/Monads_as_containers

[19] (2017, November) Juicypixels. [Online]. Available: https://hackage.haskell.org/package/JuicyPixels-3.2.9.1

[heading=bibintoc, title=References]

APÉNDICE A

Imágenes

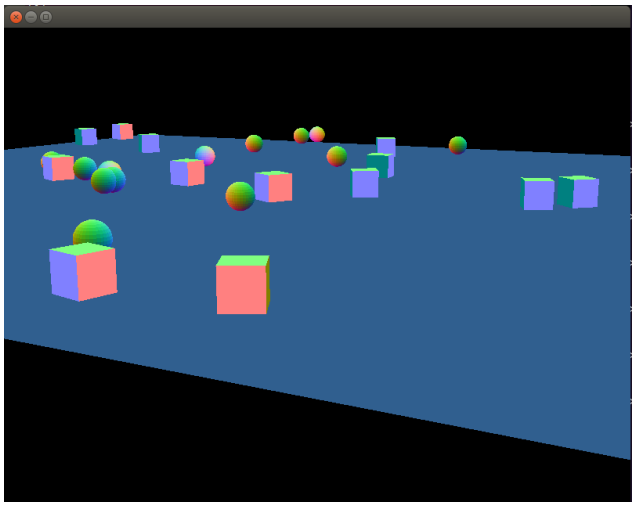

Figura A.1 Juego que contiene esferas y cubos que aparecen random y se eliminan al chocar.

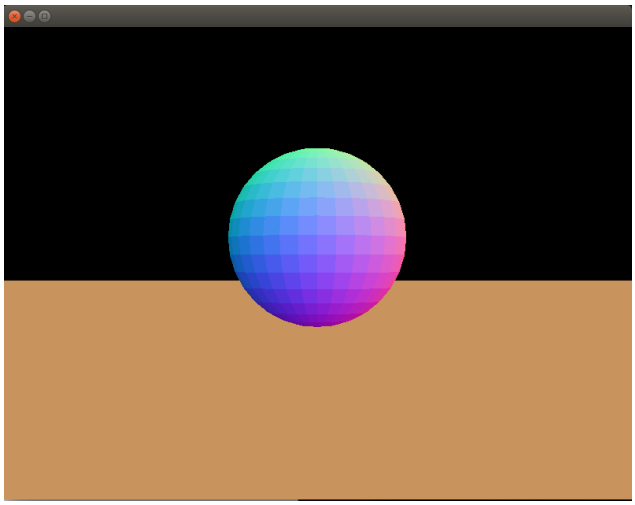

Figura A.2 Juego de una pelota rebotando contra un piso.

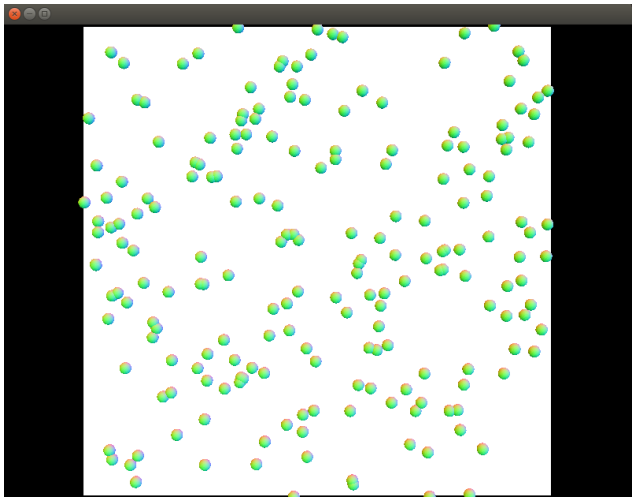

Figura A.3 Juego en el que 200 esferas chocan y rebotan entre sí.

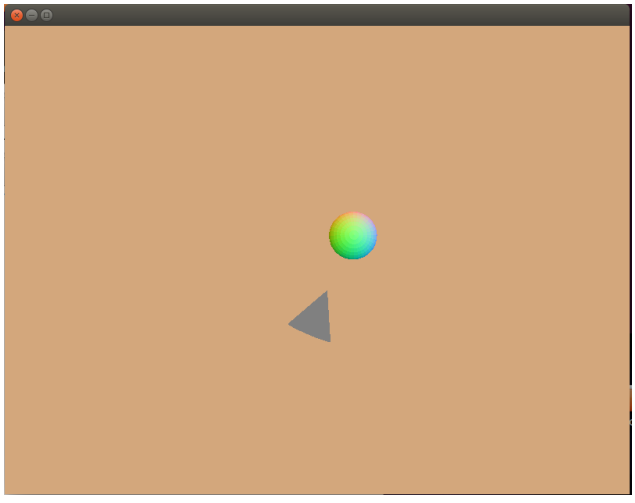

Figura A.4 Juego en el que un cono persigue a una esfera controlada por un jugador.

APÉNDICE B

Código

Listing B.1 EasyGL.hs

```
------------------------------------------------------------------------------
-- |
-- Module : EasyGL
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- A module that makes OpenGL easier in haskell.
--
------------------------------------------------------------------------------

module EasyGL (
  module EasyGL.Camera,
  module EasyGL.Entity,
  module EasyGL.IndexedModel,
  module EasyGL.Material,
  module EasyGL.Obj,
  module EasyGL.Shader,
  module EasyGL.Texture,
  module EasyGL.Util
) where

import EasyGL.Camera
import EasyGL.Entity
import EasyGL.IndexedModel
import EasyGL.Material
import EasyGL.Obj
import EasyGL.Shader
import EasyGL.Texture
import EasyGL.Util
```

Listing B.2 EasyGLUT.hs

```
------------------------------------------------------------------------------
-- |
-- Module : EasyGLUT
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
```

```
-- Stability : stable
-- Portability : portable
--
-- Provides and easy to use interface to GLUT, performs all the heavy lifting required to render OpenGL to a single window.
--
-----------------------------------------------------------------------------

module EasyGLUT (
  MouseState(..),
  GLUT.MouseButton(..),
  GLUT.SpecialKey(..),
  GLUT.Key(..),
  KeyState(..),
  GLUT,
  changeMainLoop,
  getMouseInfo,
  getKeysInfo,
  fixMouseAt,
  freeMouse,
  currentAspect,
  getWindowSize,
  setWindowSize,
  getWindowPosition,
  setWindowPosition,
  fullScreen,
  hideCursor,
  showCursor,
  maybeDown,
  maybePressed,
  maybeUp,
  maybeReleased,
  initOpenGLEnvironment,
  initGL)
where

import Control.Monad.Reader
import Control.Monad.State.Strict hiding (get)
import qualified Control.Monad.State.Strict as State (get)
import Data.IORef (IORef, newIORef)
import Data.Map.Strict
import qualified Data.Map.Strict as Map
import Data.StateVar (get, ($=), ($=!), ($~!))
import qualified Graphics.Rendering.OpenGL as GL
import qualified Graphics.UI.GLUT as GLUT
import Control.Seq

-----------------------------------------------------------------------------
-- $GLUT Monad
-- The GLUT monad takes care of input (mouse and keyboard) as well as other common GLUT callbacks,
-- state of the window can be access using the monad functions.
--
-- GLUT will run double buffered and the buffer change will be perform automatically at the end of each frame.
-- The user of this monad is expected to not import the GLUT library, and communicate with GLUT by the exposed functions in this library.
-- Inside GLUT monad the user should perform an IOlift and call a function that would only make OpenGL calls to draw the current frame.
--

-- | State of mouse.
data MouseState = FreeMouse GL.GLint GL.GLint | FixMouse GL.GLint GL.GLint
```

```
-- | Keeps info of mouse and keyboard keys state.
type MouseKey = (Map GLUT.Key KeyState,MouseState)

type GLUTState = DisplayData
type GLUTRead = MouseKey

-- | Monad that describes behavior of GLUT.
type GLUT a = StateT GLUTState (ReaderT GLUTRead IO) a

-- | Changes GLUT main callback.
changeMainLoop :: GLUT () -> GLUT ()
changeMainLoop f = modify' (\s -> s{ioLoop=f})

-- | Returns current frame mouse info.
getMouseInfo :: GLUT MouseState
getMouseInfo = fmap snd ask

-- | Returns current frame keys info.
getKeysInfo :: GLUT (Map GLUT.Key KeyState)
getKeysInfo = fmap fst ask

-- | Fix mouse at a given point of window.
fixMouseAt :: GL.GLint -> GL.GLint -> GLUT ()
fixMouseAt x y = do
  modify' (\s -> s{mouseIsFixed=True})
  bottons <- fmap bottons State.get
  GLUT.pointerPosition $=! GL.Position x y
  GLUT.passiveMotionCallback $=! Just (fixedMouse x y bottons)
  GLUT.motionCallback $=! Just (fixedMouse x y bottons)

-- | Allow free mouse movement.
freeMouse :: GLUT ()
freeMouse = do
  modify' (\s -> s{mouseIsFixed=False})
  bottons <- fmap bottons State.get
  GLUT.passiveMotionCallback $=! Just (defaultMouse bottons)
  GLUT.motionCallback $=! Just (defaultMouse bottons)

-- | Returns current aspect ratio of active glut windows.
currentAspect :: GLUT GL.GLdouble
currentAspect = do
  (GLUT.Size w h) <- GLUT.get GLUT.windowSize
  return (fromIntegral w / fromIntegral h)

-- | Returns size of active glut windows.
getWindowSize :: GLUT (GL.GLsizei,GL.GLsizei)
getWindowSize = do
  (GLUT.Size w h) <- GLUT.get GLUT.windowSize
  return (w,h)

-- | Sets size of active glut windows.
setWindowSize :: GL.GLsizei -> GL.GLsizei -> GLUT ()
setWindowSize w h = GLUT.windowSize $=! GLUT.Size w h

-- | Returns position of active glut windows.
getWindowPosition :: GLUT (GL.GLint,GL.GLint)
getWindowPosition = do
  (GLUT.Position x y) <- GLUT.get GLUT.windowPosition
  return (x,y)
```

```
-- | Sets position of active glut windows.
setWindowPosition :: GL.GLint -> GL.GLint -> GLUT ()
setWindowPosition x y = GLUT.windowPosition $=! GLUT.Position x y

-- | Makes glut window fullscreen.
fullScreen :: GLUT ()
fullScreen = GLUT.fullScreen

-- | Hides cursor.
hideCursor :: GLUT ()
hideCursor = GLUT.cursor $=! GLUT.None

-- | Show cursor.
showCursor :: GLUT ()
showCursor = GLUT.cursor $=! GLUT.RightArrow

-- | States in witch a key can be found.
data KeyState = Down | Up | Pressed | Released deriving Show

change :: KeyState -> KeyState
change Down = Down
change Up = Up
change Released = Up
change Pressed = Down

filterDown :: t -> t -> KeyState -> t
filterDown x _ Down = x
filterDown x _ Pressed = x
filterDown _ x _ = x
filterPressed :: t -> t -> KeyState -> t
filterPressed x _ Pressed = x
filterPressed _ x _ = x
filterUp :: t -> t -> KeyState -> t
filterUp x _ Up = x
filterUp x _ Released = x
filterUp _ x _ = x
filterReleased :: t -> t -> KeyState -> t
filterReleased x _ Released = x
filterReleased _ x _ = x

maybeDown :: t -> t -> Maybe KeyState -> t
maybeDown x y = maybe x (filterDown y x)
maybePressed :: t -> t -> Maybe KeyState -> t
maybePressed x y = maybe x (filterPressed y x)
maybeUp :: t -> t -> Maybe KeyState -> t
maybeUp x y = maybe x (filterUp y x)
maybeReleased :: t -> t -> Maybe KeyState -> t
maybeReleased x y = maybe x (filterReleased y x)

idle :: GLUT.IdleCallback
idle = GLUT.postRedisplay Nothing

reshape :: GLUT.ReshapeCallback
reshape size@(GL.Size w h) = do
  GL.viewport $= (GL.Position 0 0, size)
  GL.matrixMode $= GL.Projection
  GL.loadIdentity
  --GL.perspective 30 (fromIntegral w / fromIntegral h) 1 200
```

```
  --GL.matrixMode $= GL.Modelview 0
  GLUT.postRedisplay Nothing

data DisplayData = DisplayData {
    ioLoop :: !(GLUT ()),
    mouseIsFixed :: !Bool,
    bottons :: IORef MouseKey
  }
display :: IORef DisplayData -> GLUT.DisplayCallback
display ref = do
  mydata <- get ref
  GL.clear [GL.ColorBuffer,GL.DepthBuffer]
  GL.loadIdentity
  keys <- get (bottons mydata)
  s <- runReaderT (execStateT (ioLoop mydata) mydata) keys
  bottons mydata $~! \(myMap,x) -> (Map.map change myMap,if mouseIsFixed mydata then FixMouse 0 0 else x)
  GLUT.swapBuffers
  ref $=! s

keyboard :: IORef MouseKey -> GLUT.KeyboardCallback
keyboard ref c _ = do
  (myMap,delta) <- get ref
  let state = Map.lookup (GLUT.Char c) myMap
  ref $=! maybe (insert (GLUT.Char c) Pressed myMap,delta)
    (\s -> case s of
             Pressed -> (insert (GLUT.Char c) Down myMap,delta)
             Down -> (myMap,delta)
             _ -> (insert (GLUT.Char c) Pressed myMap,delta)
    )
    state

keyboardUp :: IORef MouseKey -> GLUT.KeyboardCallback
keyboardUp ref c _ = ref $~! \(myMap,delta) -> (insert (GLUT.Char c) Released myMap,delta)

special :: IORef MouseKey -> GLUT.SpecialCallback
special ref key _ = ref $~! \(myMap,delta) -> (insert (GLUT.SpecialKey key) Pressed myMap,delta)

specialUp :: IORef MouseKey -> GLUT.SpecialCallback
specialUp ref key _ = ref $~! \(myMap,delta) -> (insert (GLUT.SpecialKey key) Released myMap,delta)

mouseCall :: IORef MouseKey -> GLUT.MouseCallback
mouseCall ref key GLUT.Up _ = ref $~! \(myMap,delta) -> (insert (GLUT.MouseButton key) Released myMap,delta)
mouseCall ref key GLUT.Down _ = ref $~! \(myMap,delta) -> (insert (GLUT.MouseButton key) Pressed myMap,delta)

defaultMouse :: IORef MouseKey -> GLUT.Position -> IO ()
defaultMouse ref (GLUT.Position x y) = ref $~! \(myMap,_) -> x `seq` y `seq` myMap `seq` (myMap,FreeMouse x y)

fixedMouse :: GL.GLint -> GL.GLint -> IORef MouseKey -> GLUT.Position -> IO ()
fixedMouse fixx fixy bottons (GLUT.Position x y) = do
  bottons $~! \(myMap,_) -> (myMap,FixMouse (x-fixx) (y-fixy))
  GLUT.passiveMotionCallback $=! Just help
  GLUT.motionCallback $=! Just help
  GLUT.pointerPosition $=! GL.Position fixx fixy
  where
    help _ = do
      GLUT.passiveMotionCallback $=! Just (fixedMouse fixx fixy bottons)
      GLUT.motionCallback $=! Just (fixedMouse fixx fixy bottons)

-- | Initializes Opengl and GLUT environments, OpenGL calls can be made after this function.
```

```
-- Must always be call before calling initGL.
-- Useful if you need to load data to the GPU, such as compiling shaders, sending mesh data or textures.
initOpenGLEnvironment :: GL.GLsizei -> GL.GLsizei -> String -> IO GLUT.Window
initOpenGLEnvironment sizex sizey windowName = do
  GLUT.getArgsAndInitialize
  GLUT.initialDisplayMode $= [GLUT.DoubleBuffered,GLUT.RGBAMode,GLUT.WithDepthBuffer]
  GLUT.initialWindowSize $= GL.Size sizex sizey
  GLUT.createWindow windowName

-- | Given a callback GLUT function, starts the main loop of program.
initGL :: GLUT () -> IO ()
initGL f = do

  GL.clearColor $=! GL.Color4 0 0 0 0
  GL.materialDiffuse GL.Front $=! GL.Color4 1 1 1 1
  GL.materialSpecular GL.Front $=! GL.Color4 1 1 1 1
  GL.materialShininess GL.Front $=! 100

  GL.depthFunc $=! Just GL.Less
  GL.autoNormal $=! GL.Enabled
  GL.normalize $=! GL.Enabled

  bottons <- newIORef ((empty,FreeMouse 0 0) :: MouseKey)

  GLUT.passiveMotionCallback $=! Just (defaultMouse bottons)
  GLUT.motionCallback $=! Just (defaultMouse bottons)

  GLUT.keyboardCallback $=! Just (keyboard bottons)
  GLUT.keyboardUpCallback $=! Just (keyboardUp bottons)
  GLUT.specialCallback $=! Just (special bottons)
  GLUT.specialUpCallback $=! Just (specialUp bottons)
  GLUT.mouseCallback $=! Just (mouseCall bottons)

  GLUT.reshapeCallback $=! Just reshape
  GLUT.idleCallback $=! Just idle
  ref <- newIORef $ DisplayData f False bottons
  GLUT.displayCallback $=! display ref
  GLUT.mainLoop
```

Listing B.3 Camera.hs

```
-------------------------------------------------------------------------------
-- |
-- Module : EasyGL.Camera
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Utility for easy interaction with OpenGL environment, provides a camera structure that is user friendly.
--
-------------------------------------------------------------------------------

module EasyGL.Camera (
  Camera2D,
  Camera3D,
  IsCamera,
```

```
  useCamera,
  createCamera2D,
  createCamera3D,
  yawCamera,
  picthCamera,
  rollCamera,
  setYawPitchRoll,
  setPosition,
  translateCamera
)
where
import Control.Monad.IO.Class (MonadIO,liftIO)
import qualified Graphics.GLU as GLU
import qualified Graphics.GL as GL
import Graphics.GL (GLdouble)
import qualified Data.Matrix as Mat
import Foreign.Marshal.Array

{-|
  Makes OpenGL environment use given camera.
-}
class IsCamera c where
  useCamera :: MonadIO m => c -> m ()

{-|
  Stores OpenGL data required to generate a 2d perspective.
-}
data Camera2D = Camera2D GLdouble GLdouble GLdouble GLdouble GLdouble GLdouble

instance IsCamera Camera2D where
  useCamera (Camera2D left right bottom top near far) = GL.glOrtho left right bottom top near far

{-|
  Creates 2D camera.
-}
createCamera2D :: GLdouble -- ^ Specify the coordinates for the left clipping plane.
  -> GLdouble -- ^ Specify the coordinates for the right clipping plane.
  -> GLdouble -- ^ Specify the coordinates for the bottom clipping plane.
  -> GLdouble -- ^ Specify the coordinates for the top clipping plane.
  -> GLdouble -- ^ Specify the distances to the nearer depth clipping plane.
  -> GLdouble -- ^ Specify the distances to the farther depth clipping plane.
  -> Camera2D
createCamera2D = Camera2D

{-|
  Stores OpenGL data required to generate a 3d perspective.
-}
data Camera3D = Camera3D !(Mat.Matrix GLdouble) GLdouble GLdouble GLdouble GLdouble GLdouble GLdouble

instance IsCamera Camera3D where
  useCamera (Camera3D matrix x y z fovy aspect zNear zFar) = do
    GL.glLoadIdentity
    GLU.gluPerspective fovy aspect zNear zFar
    GL.glMatrixMode GL.GL_MODELVIEW
    GL.glLoadIdentity
    liftIO $ withArray (Mat.toList matrix) $ \ptr -> GL.glMultMatrixd ptr
    GL.glTranslated (-x) (-y) (-z)

{-|
```

```
   Creates 3D camera.
-}
createCamera3D :: GLdouble -- ^ Position of camera in x.
   -> GLdouble -- ^ Position of camera in y.
   -> GLdouble -- ^ Position of camera in z.
   -> GLdouble -- ^ Camera Picth.
   -> GLdouble -- ^ Camera Roll.
   -> GLdouble -- ^ Camera Yaw.
   -> GLdouble -- ^ Specifies the field of view angle, in degrees, in the y direction.
   -> GLdouble -- ^ Specifies the aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (
      ↪ height).
   -> GLdouble -- ^ Specifies the distance from the Camera to the near clipping plane (always positive).
   -> GLdouble -- ^ Specifies the distance from the Camera to the far clipping plane (always positive).
   -> Either String Camera3D
createCamera3D x y z yaw picth roll fovy aspect zNear zFar
   | zNear > 0 && zFar > 0 = Right $ Camera3D rot x y z fovy aspect zNear zFar
   | otherwise = Left "zNear␣or␣zFar␣is␣not␣positive."
   where
      rot = yawPicthRollMatrix yaw picth roll

-- | Sets camera yaw picth and roll
setYawPitchRoll :: GLdouble -> GLdouble -> GLdouble -> Camera3D -> Camera3D
setYawPitchRoll yaw picth roll (Camera3D _ x y z fovy aspect zNear zFar) = Camera3D newMat x y z fovy aspect zNear zFar
   where
      newMat = yawPicthRollMatrix yaw picth roll

-- | Sets camera position (x,y,z)
setPosition :: GLdouble -> GLdouble -> GLdouble -> Camera3D -> Camera3D
setPosition x y z (Camera3D mat _ _ _ fovy aspect zNear zFar) = Camera3D mat x y z fovy aspect zNear zFar

-- | translates camera position (x,y,z)
translateCamera :: GLdouble -> GLdouble -> GLdouble -> Camera3D -> Camera3D
translateCamera x y z (Camera3D mat oldx oldy oldz fovy aspect zNear zFar) = Camera3D mat (x+oldx) (y+oldy) (z+oldz) fovy aspect zNear zFar

-- | Yaws Camera
yawCamera :: GLdouble -> Camera3D -> Camera3D
yawCamera yaw (Camera3D mat x y z fovy aspect zNear zFar) = Camera3D newMat x y z fovy aspect zNear zFar
   where
      newMat = Mat.multStd (yawMatrix yaw) mat

-- | Picthes Camera
picthCamera :: GLdouble -> Camera3D -> Camera3D
picthCamera picth (Camera3D mat x y z fovy aspect zNear zFar) = Camera3D newMat x y z fovy aspect zNear zFar
   where
      newMat = Mat.multStd (picthMatrix picth) mat

-- | Rolls Camera
rollCamera :: GLdouble -> Camera3D -> Camera3D
rollCamera roll (Camera3D mat x y z fovy aspect zNear zFar) = Camera3D newMat x y z fovy aspect zNear zFar
   where
      newMat = Mat.multStd (rollMatrix roll) mat

toAngle :: Floating a => a -> a
toAngle a = pi*a/180

sina :: Floating a => a -> a
sina = sin . toAngle

cosa :: Floating a => a -> a
```

```
cosa = cos . toAngle

picthMatrix :: Floating a => a -> Mat.Matrix a
picthMatrix angle = Mat.fromList 4 4 [
    1, 0, 0, 0,
    0, cosa angle, -1 * sina angle, 0,
    0, sina angle, cosa angle, 0,
    0, 0, 0, 1
  ]

yawMatrix :: Floating a => a -> Mat.Matrix a
yawMatrix angle = Mat.fromList 4 4 [
    cosa angle, 0, sina angle, 0,
    0, 1, 0, 0,
    -1 * sina angle, 0, cosa angle, 0,
    0, 0, 0, 1
  ]

rollMatrix :: Floating a => a -> Mat.Matrix a
rollMatrix angle = Mat.fromList 4 4 [
    cosa angle, -1 * sina angle, 0, 0,
    sina angle, cosa angle, 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1
  ]

rollYawPicthMatrix :: Floating a => a -> a -> a -> Mat.Matrix a
rollYawPicthMatrix yaw picth roll = Mat.fromList 4 4 [
    cosRoll * cosYaw,
    (cosRoll * sinYaw * sinPicth) - (sinRoll * cosPicth),
    (cosRoll * sinYaw * cosPicth) + (sinRoll * sinPicth),
    0,
    sinRoll * cosYaw,
    (sinRoll * sinYaw * sinPicth) + (cosRoll * cosPicth),
    (sinRoll * sinYaw * cosPicth) - (cosRoll * sinPicth),
    0,
    negate sinYaw,
    cosYaw * sinPicth,
    cosYaw * cosPicth,
    0,
    0,
    0,
    0,
    1
  ]
  where
    sinYaw = sina yaw
    cosYaw = cosa yaw
    sinPicth = sina picth
    cosPicth = cosa picth
    sinRoll = sina roll
    cosRoll = cosa roll

yawPicthRollMatrix :: Floating a => a -> a -> a -> Mat.Matrix a
yawPicthRollMatrix yaw picth roll = Mat.fromList 4 4 [
    (cosYaw * cosRoll) + (sinYaw * sinPicth * sinRoll),
    (sinYaw * sinPicth * cosRoll) - (cosYaw * sinRoll),
    sinYaw * cosPicth,
    0,
```

```
    cosPicth * sinRoll,
    cosPicth * cosRoll,
    negate sinPicth,
    0,
    (cosYaw * sinPicth * sinRoll) - (sinYaw * cosRoll),
    (sinYaw * sinRoll) + (cosYaw * sinPicth * cosRoll),
    cosYaw * cosPicth,
    0,
    0,
    0,
    0,
    1
  ]
  where
    sinYaw = sina yaw
    cosYaw = cosa yaw
    sinPicth = sina picth
    cosPicth = cosa picth
    sinRoll = sina roll
    cosRoll = cosa roll
```

Listing B.4 EasyMem.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : EasyGL.EasyMem
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Helps to manage memory easier.
-- Sometimes people do not remember to run the destructor for a given data type, leaving date in gpu memory, this module provides a way to
    ↪ make the garbage collector clean gpu memory when the object is no longer reachable.
-- Every function in this module behaves the same as the ones in the modules they originally come from, with the difference that objects create
    ↪ from function from this module will call their destructor function on becoming unreachable.
-- Destructor function will only be call when haskell GC runs, witch may never happen, it is recommended to run performMinorGC from module
    ↪ System.Mem, witch will trigger a minor GC, this will ensure that destructors are run for every unreachable object.
-- A GC is expensive, be sure not to run one every frame as you may encounter performance issues, instead run it every so may seconds or every
    ↪ time you think garbage have been generated, like loading a new scene and forgetting the previews one.
-- If you create objects with this function make sure not to call their destructor, I do not know what OpenGL may do on double call for an object
    ↪ destruction, also OpenGL may reuse the name of a destroyed object and erase an object it should not.
-- The only way to ensure correct resource management will be to create an OpenGL monad that manages all of that internally, it may be create
    ↪ in a future update.
--
-----------------------------------------------------------------------------

module EasyGL.EasyMem (
  readObj2Ent,
  obj2Ent,
  indexedModel2Ent,
  loadShadersFromFile,
  loadShadersFromBS,
  makeMaterial,
  createTexture2DStatic
) where
```

```
import Control.Monad
import Control.Monad.IO.Class (MonadIO, liftIO)
import qualified Data.ByteString as BS
import qualified EasyGL.Entity as E
import qualified EasyGL.IndexedModel as IM
import qualified EasyGL.Material as M
import qualified EasyGL.Obj as Obj
import qualified EasyGL.Shader as S
import qualified EasyGL.Texture as T
import Foreign.Ptr
import qualified Graphics.Rendering.OpenGL as GL
import System.IO (Handle)
import System.Mem.Weak

readObj2Ent :: MonadIO m => String -> m E.Entity
readObj2Ent s = do
  e <- Obj.readObj2Ent s
  liftIO $ mkWeakPtr e $ Just $ E.deleteEntity e
  return e

obj2Ent :: MonadIO m => Obj.Obj -> m E.Entity
obj2Ent o = do
  e <- Obj.obj2Ent o
  liftIO $ mkWeakPtr e $ Just $ E.deleteEntity e
  return e

indexedModel2Ent :: MonadIO m => [IM.IndexedModel] -> m E.Entity
indexedModel2Ent l = do
  e <- E.indexedModel2Ent l
  liftIO $ mkWeakPtr e $ Just $ E.deleteEntity e
  return e

loadShadersFromFile :: MonadIO m => [String] -> [GL.ShaderType] -> Maybe Handle -> m S.Shader
loadShadersFromFile ls lst mh = do
  s <- S.loadShadersFromFile ls lst mh
  liftIO $ mkWeakPtr s $ Just $ S.deleteShader s
  return s

loadShadersFromBS :: MonadIO m => [BS.ByteString] -> [GL.ShaderType] -> Maybe Handle -> m S.Shader
loadShadersFromBS lbs lst mh = do
  s <- S.loadShadersFromBS lbs lst mh
  liftIO $ mkWeakPtr s $ Just $ S.deleteShader s
  return s

makeMaterial :: (MonadIO m) => S.Shader -> [(FilePath,String)] -> m (Either String M.Material)
makeMaterial shader textures = do
  m <- M.makeMaterial shader textures
  forM_ m (\mat -> liftIO $ mkWeakPtr mat $ Just $ M.deleteMaterial mat)
  return m

createTexture2DStatic :: (Integral a,MonadIO m) => a -> a -> Ptr () -> T.ClampType -> T.FilteringType -> m T.Texture
createTexture2DStatic width height dataPtr ct ft = do
  t <- T.createTexture2DStatic width height dataPtr ct ft
  liftIO $ mkWeakPtr t $ Just $ T.deleteTexture t
  return t
```

Listing B.5 Entity.hs

```
-------------------------------------------------------------------------
```

```
-- |
-- Module : EasyGL.Entity
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Assists in the process of loading a mesh to gpu and drawing it with OpenGL.
--
-----------------------------------------------------------------------------

module EasyGL.Entity(
  Entity,
  renderEnt,
  indexedModel2Ent,
  deleteEntity
  ) where

import qualified EasyGL.IndexedModel as IM
import qualified EasyGL.Shader as S

import qualified Data.Vector.Storable as VS
import Foreign.Marshal.Array
import Foreign.Ptr
import Foreign.Storable

import Control.Monad
import Control.Monad.IO.Class (MonadIO, liftIO)
import Graphics.Rendering.OpenGL
import System.IO

bufferOffset :: Integral a => a -> Ptr b
bufferOffset = plusPtr nullPtr . fromIntegral

setVertexAttribPointer :: (GLuint,NumComponents) -> IO ()
setVertexAttribPointer (location,numComponents) = do
  vertexAttribArray (AttribLocation location) $= Enabled
  vertexAttribPointer (AttribLocation location) $=
    (ToFloat, VertexArrayDescriptor numComponents Float 0 (bufferOffset 0))

cleanVertexAttribPointer :: (GLuint,NumComponents) -> IO ()
cleanVertexAttribPointer (location,_) = vertexAttribArray (AttribLocation location) $= Disabled

makeBufferObject :: (Storable a) => BufferTarget -> VS.Vector a -> BufferUsage -> Maybe (GLuint,NumComponents) -> IO BufferObject
makeBufferObject target elems usage m = do
  --creating buffer
  buffer <- genObjectName
  bindBuffer target $= Just buffer
  --filling buffer
  VS.unsafeWith elems $ \ptr -> do
    let size = fromIntegral (VS.length elems * sizeOf (VS.head elems))
    bufferData target $= (size, ptr, usage)
  --setting vertex attrib pointer to be aware of buffer
  maybe (return ()) setVertexAttribPointer m
  return buffer

makeArrayBufferObject = do
```

```
  buffer <- genObjectName :: IO VertexArrayObject
  bindVertexArrayObject $= Just buffer
  return buffer

data SubEntity = SubEntity {
  vertexArrObject :: VertexArrayObject,
  vertexes :: BufferObject, --most always occur
  textureCoord :: Maybe BufferObject,
  normals :: Maybe BufferObject,
  indexBuffer :: BufferObject,--most always occur
  vertexNum :: !Int,
  vertexIndexNum :: !Int
} deriving (Show)

-- | Represents a mesh with possibly normals and texture coordinates, that is loaded in gpu and can be draw within an OpenGL environment.
type Entity = [SubEntity]

-- | Given an indexed model, generates an entity.
indexedModel2Ent :: MonadIO m => [IM.IndexedModel] -> m Entity
indexedModel2Ent = mapM (liftIO . fromIM)

fromIM :: IM.IndexedModel -> IO SubEntity
fromIM g = do
  mesh <- makeArrayBufferObject
  vertexBuffer <- makeBufferObject ArrayBuffer (IM.vertices g) StaticDraw $ Just (0,3)
  textureCoord <- if VS.null (IM.textureCoord g) then return Nothing else
    fmap Just $ makeBufferObject ArrayBuffer (IM.textureCoord g) StaticDraw $ Just (1,2)
  normalBuffer <- if VS.null (IM.normals g) then return Nothing else
    fmap Just $ makeBufferObject ArrayBuffer (IM.normals g) StaticDraw $ Just (2,3)
  indexBuffer <- makeBufferObject ElementArrayBuffer (IM.indexes g) StaticDraw Nothing
  bindVertexArrayObject $= Nothing
  return (SubEntity mesh vertexBuffer Nothing normalBuffer indexBuffer (VS.length (IM.vertices g)) (VS.length (IM.indexes g)))

-- | Given a shader and an io action (that should only set the shader uniform variables), draws with OpenGL the given Entity.
renderEnt :: MonadIO m => S.Shader -> Entity -> S.Uniform () -> m ()
renderEnt shader ent uniformAction = S.withShaderSafe shader uniformAction $ liftIO $ mapM_ renderSubEnt ent

renderSubEnt :: SubEntity -> IO ()
renderSubEnt e = do
  bindVertexArrayObject $= Just (vertexArrObject e)
  --drawElements Triangles (fromIntegral.vertexIndexNum $ e) UnsignedInt (vertexIndex e)
  drawElementsBaseVertex Triangles (fromIntegral.vertexIndexNum $ e) UnsignedInt nullPtr 0
  bindVertexArrayObject $= Nothing

-- | Deletes and entity and frees gpu memory.
deleteEntity :: MonadIO m => Entity -> m ()
deleteEntity = mapM_ deleteSubEntity

deleteSubEntity :: MonadIO m => SubEntity -> m ()
deleteSubEntity se = do
  deleteObjectName $ vertexArrObject se
  deleteObjectName $ vertexes se
  deleteObjectName $ indexBuffer se
  forM_ (textureCoord se) deleteObjectName
  forM_ (normals se) deleteObjectName
```

```
--
```

Listing B.6 IndexedModel.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : EasyGL.IndexedModel
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Middle representation of a mesh data (mesh,texture coordinates and normals) in a format usable to OpenGL. This middle form allows to alter
    ↪ the mesh by, for example, adding normal in meshes that lacks them.
--
-----------------------------------------------------------------------------

module EasyGL.IndexedModel (
  IndexedModel(..),
  emptyIndexedModel,
  renderNormals,
  generateNormalsSoft,
  generateNormalsHard
)
where

import Control.Monad.Reader
import Control.Monad.ST
import Control.Monad.State.Lazy
import Data.Foldable (toList)
import qualified Data.Map.Strict as Map
import qualified Data.Sequence as Seq
import qualified Data.Vector as V
import qualified Data.Vector.Mutable as VM
import Data.Vector.Storable ((!))
import qualified Data.Vector.Storable as VS
import qualified Data.Vector.Storable.Mutable as VSM
import EasyGL.Util
import Graphics.Rendering.OpenGL hiding (get)

-- | Data representing a mesh.
data IndexedModel = IndexedModel {
```

```
  vertices :: !(VS.Vector (Vertex3 GLfloat)),
  normals :: !(VS.Vector (Vector3 GLfloat)),
  textureCoord :: !(VS.Vector (Vector2 GLfloat)),
  indexes :: !(VS.Vector GLuint)
} deriving (Show)

-- | An empty mesh.
emptyIndexedModel :: IndexedModel
emptyIndexedModel = IndexedModel VS.empty VS.empty VS.empty VS.empty

toVector :: (Num a) => Vertex3 a -> Vertex3 a -> Vector3 a
toVector (Vertex3 x1 y1 z1) (Vertex3 x2 y2 z2) = Vector3 (x1-x2) (y1-y2) (z1-z2)

toVertex :: (Num a) => Vertex3 a -> Vector3 a -> Vertex3 a
toVertex (Vertex3 x1 y1 z1) (Vector3 x2 y2 z2) = Vertex3 (x1+x2) (y1+y2) (z1+z2)

renderLine :: Vertex3 GLfloat -> Vertex3 GLfloat -> IO ()
renderLine a b = do
  vertex a
  vertex b

-- | An utility function that draw with OpenGL the normals of a given mesh as lines originating in the corresponding vertex of each normal.
-- The lines will be drawn with the color set in OpenGL environment.
renderNormals :: IndexedModel -> IO ()
renderNormals g = renderPrimitive Lines $ VS.zipWithM_ renderLine verts newVerts
  where
    verts = vertices g
    newVerts = VS.zipWith toVertex verts (normals g)


-- | Genarate normal for a given mesh, see hardness (0%) in: https://help.thefoundry.co.uk/modo/content/help/pages/uving/vertex_normals.html
generateNormalsSoft :: IndexedModel -> IndexedModel
generateNormalsSoft g = g{normals=runST $ do
    acc <- VM.replicate len Nothing
    runReaderT (evalStateT (generateNormalsSoftAux g) 0) acc
    fmap (V.convert . V.map maybeNormalize) $ V.freeze acc
  }
  where
    len = VS.length . vertices $ g
    maybeNormalize Nothing = Vector3 0 0 0
    maybeNormalize (Just v) = normalizeVec3 v

(<+>) :: (Num a) => a -> Maybe a -> Maybe a
e <+> Nothing = Just e
e <+> (Just a) = Just $ a+e

generateNormalsSoftAux :: IndexedModel -> StateT Int (ReaderT (VM.MVector s (Maybe (Vector3 GLfloat))) (ST s)) ()
generateNormalsSoftAux g = do
  acc <- ask
  current <- get
  unless (current == len) $ do
    let x = fromIntegral $ index ! current
        y = fromIntegral $ index ! (current+1)
        z = fromIntegral $ index ! (current+2)
        v1 = toVector (verts ! y) (verts ! x)
        v2 = toVector (verts ! z) (verts ! x)
        normal = normalizeVec3 $ crossVec3 v1 v2
    VM.modify acc (normal <+>) x
    VM.modify acc (normal <+>) y
```

```
    VM.modify acc (normal <+>) z
    put $ current+3
    generateNormalsSoftAux g
  where
    len = VS.length . indexes $ g
    verts = vertices g
    index = indexes g

-- | Genarate normal for a given mesh, see hardness (100%) in: https://help.thefoundry.co.uk/modo/content/help/pages/uving/vertex_normals.
    ↪ html
generateNormalsHard :: IndexedModel -> IndexedModel
generateNormalsHard g = runST $ do
  inde <- VS.thaw $ indexes g
  acc <- VM.replicate len Nothing
  if VS.null texts then do
    (extraVerts,extraNorms) <- runReaderT (evalStateT generateNormalsHardAux1 (Hard1 0 len Map.empty Seq.empty Seq.empty)) (verts,inde,
        ↪ acc)
    let newVerts = verts VS.++ (VS.fromList . toList $ extraVerts)
    newInde <- VS.freeze inde
    newNorms0 <- fmap (V.convert . V.map toVer) $ V.freeze acc
    let newNorms = newNorms0 VS.++ (VS.fromList . toList $ extraNorms)
    return $ IndexedModel newVerts newNorms VS.empty newInde
  else do
    (extraVerts,extraNorms,extraText) <- runReaderT (evalStateT generateNormalsHardAux2 (Hard2 0 len Map.empty Seq.empty Seq.empty
        ↪ Seq.empty)) (verts,texts,inde,acc)
    let newVerts = verts VS.++ (VS.fromList . toList $ extraVerts)
    newInde <- VS.freeze inde
    let newText = texts VS.++ (VS.fromList . toList $ extraText)
    newNorms0 <- fmap (V.convert . V.map toVer) $ V.freeze acc
    let newNorms = newNorms0 VS.++ (VS.fromList . toList $ extraNorms)
    return $ IndexedModel newVerts newNorms newText newInde
  where
    toVer Nothing = Vector3 0 0 0
    toVer (Just v) = v
    verts = vertices g
    texts = textureCoord g
    len = VS.length verts

data Hard1 = Hard1 {
    current1 :: Int,
    next1 :: Int,
    extraMap1 :: Map.Map (Vertex3 GLfloat,Vector3 GLfloat) GLuint,
    extraVert1 :: Seq.Seq (Vertex3 GLfloat),
    extraNorm1 :: Seq.Seq (Vector3 GLfloat)
  }

generateNormalsHardAux1 :: StateT Hard1 (ReaderT (VS.Vector (Vertex3 GLfloat),VSM.MVector s GLuint,VM.MVector s (Maybe (Vector3
    ↪ GLfloat))) (ST s)) (Seq.Seq (Vertex3 GLfloat),Seq.Seq (Vector3 GLfloat))
generateNormalsHardAux1 = do
  (verts,index,normacc) <- ask
  data1 <- get
  let current = current1 data1
      emap = extraMap1 data1
  if current == VSM.length index then return (extraVert1 data1,extraNorm1 data1)
  else do
    x <- fmap fromIntegral $ VSM.read index current
    y <- fmap fromIntegral $ VSM.read index (current+1)
    z <- fmap fromIntegral $ VSM.read index (current+2)
    currentNorm1 <- VM.read normacc x
```

```
    currentNorm2 <- VM.read normacc y
    currentNorm3 <- VM.read normacc z
    let vert1 = verts ! x
        vert2 = verts ! y
        vert3 = verts ! z
        v1 = toVector vert2 vert1
        v2 = toVector vert3 vert1
        normal = normalizeVec3 $ crossVec3 v1 v2
        mextra1 = Map.lookup (vert1,normal) emap
        mextra2 = Map.lookup (vert2,normal) emap
        mextra3 = Map.lookup (vert3,normal) emap
    decider1 0 x normal currentNorm1 mextra1
    decider1 1 y normal currentNorm2 mextra2
    decider1 2 z normal currentNorm3 mextra3
    modify $ \data2 -> data2{current1=current+3}
    generateNormalsHardAux1

decider1 :: Int -> Int -> Vector3 GLfloat -> Maybe (Vector3 GLfloat) -> Maybe GLuint -> StateT Hard1 (ReaderT (VS.Vector (Vertex3
    ↪ GLfloat),VSM.MVector s GLuint,VM.MVector s (Maybe (Vector3 GLfloat))) (ST s)) ()
decider1 offset pos norm currentNorm mextra = do
  (verts,index,normacc) <- ask
  data1 <- get
  let current = current1 data1
      next = next1 data1
      emap = extraMap1 data1
      extraV = extraVert1 data1
      extraN = extraNorm1 data1
      vert = verts ! pos
  maybe (do
        VM.write normacc pos (Just norm)
        modify $ \data2 -> data2{extraMap1=Map.insert (vert,norm) (fromIntegral pos) emap}
    )
    (const $ maybe (do
        VSM.write index (current+offset) (fromIntegral next)
        modify $ \data2 -> data2{next1=next + 1,
          extraVert1=extraV Seq.|> vert,
          extraNorm1=extraN Seq.|> norm,
          extraMap1=Map.insert (vert,norm) (fromIntegral next) emap}
      )
      (\i->VSM.write index (current+offset) (fromIntegral i))
      mextra
    )
    currentNorm

data Hard2 = Hard2 {
    current2 :: Int,
    next2 :: Int,
    extraMap2 :: Map.Map (Vertex3 GLfloat,Vector2 GLfloat,Vector3 GLfloat) GLuint,
    extraVert2 :: Seq.Seq (Vertex3 GLfloat),
    extraNorm2 :: Seq.Seq (Vector3 GLfloat),
    extraText2 :: Seq.Seq (Vector2 GLfloat)
  }

generateNormalsHardAux2 :: StateT Hard2 (ReaderT (VS.Vector (Vertex3 GLfloat),VS.Vector (Vector2 GLfloat),VSM.MVector s GLuint,VM.
    ↪ MVector s (Maybe (Vector3 GLfloat))) (ST s)) (Seq.Seq (Vertex3 GLfloat),Seq.Seq (Vector3 GLfloat),Seq.Seq (Vector2 GLfloat))
generateNormalsHardAux2 = do
  (verts,texts,index,normacc) <- ask
  data1 <- get
  let current = current2 data1
```

```
      emap = extraMap2 data1
  if (current == VSM.length index) then return (extraVert2 data1,extraNorm2 data1,extraText2 data1)
  else do
    x <- fmap fromIntegral $ VSM.read index current
    y <- fmap fromIntegral $ VSM.read index (current+1)
    z <- fmap fromIntegral $ VSM.read index (current+2)
    currentNorm1 <- VM.read normacc x
    currentNorm2 <- VM.read normacc y
    currentNorm3 <- VM.read normacc z
    let vert1 = verts ! x
        vert2 = verts ! y
        vert3 = verts ! z
        text1 = texts ! x
        text2 = texts ! y
        text3 = texts ! z
        v1 = toVector vert2 vert1
        v2 = toVector vert3 vert1
        normal = normalizeVec3 $ crossVec3 v1 v2
        mextra1 = Map.lookup (vert1,text1,normal) emap
        mextra2 = Map.lookup (vert2,text2,normal) emap
        mextra3 = Map.lookup (vert3,text3,normal) emap
    decider2 0 x normal currentNorm1 mextra1
    decider2 1 y normal currentNorm2 mextra2
    decider2 2 z normal currentNorm3 mextra3
    modify $ \data2 -> data2{current2=current+3}
    generateNormalsHardAux2

decider2 :: Int -> Int -> Vector3 GLfloat -> Maybe (Vector3 GLfloat) -> Maybe GLuint -> StateT Hard2 (ReaderT (VS.Vector (Vertex3
    ↪ GLfloat),VS.Vector (Vector2 GLfloat),VSM.MVector s GLuint,VM.MVector s (Maybe (Vector3 GLfloat))) (ST s)) ()
decider2 offset pos norm currentNorm mextra = do
  (verts,texts,index,normacc) <- ask
  data1 <- get
  let current = current2 data1
      next = next2 data1
      emap = extraMap2 data1
      extraV = extraVert2 data1
      extraN = extraNorm2 data1
      extraT = extraText2 data1
      vert = verts ! pos
      text = texts ! pos
  maybe (do
        VM.write normacc pos (Just norm)
        modify $ \data2 -> data2{extraMap2=Map.insert (vert,text,norm) (fromIntegral pos) emap}
    )
    (const $ maybe (do
        VSM.write index (current+offset) (fromIntegral next)
        modify $ \data2 -> data2{next2=next + 1,
          extraVert2=extraV Seq.|> vert,
          extraNorm2=extraN Seq.|> norm,
          extraText2=extraT Seq.|> text,
          extraMap2=Map.insert (vert,text,norm) (fromIntegral next) emap}
      )
      (\i->VSM.write index (current+offset) (fromIntegral i))
      mextra
    )
    currentNorm
```

Listing B.7 Material.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : EasyGL.Material
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Material allows loading textures to gpu and easily using then within a shader, uniform variable must be set with the uniform monad.
--
-----------------------------------------------------------------------------

module EasyGL.Material(
makeMaterial,
drawWithMat,
deleteMaterial,
Material
) where

import Codec.Picture (Image (..), Pixel, PixelBaseComponent,
                                                PixelRGBA8, convertRGBA8, readImage)
import Control.Monad
import Control.Monad.IO.Class (MonadIO, liftIO)
import Data.Either
import qualified Data.Map as Map
import Data.Vector.Storable (unsafeWith)
import Data.Word
import qualified EasyGL.Entity as Ent
import qualified EasyGL.Shader as S
import qualified EasyGL.Texture as T
import Foreign.Marshal.Array
import Foreign.Marshal.Utils
import Foreign.Ptr
import Foreign.Storable
import qualified Graphics.GL as GL
import System.IO

texturesEnums = [GL.GL_TEXTURE0,GL.GL_TEXTURE1,GL.GL_TEXTURE2,GL.GL_TEXTURE3,GL.GL_TEXTURE4,GL.
    ↪ GL_TEXTURE5,GL.GL_TEXTURE6,GL.GL_TEXTURE7,GL.GL_TEXTURE8,GL.GL_TEXTURE9,GL.GL_TEXTURE10,GL.
    ↪ GL_TEXTURE11,GL.GL_TEXTURE12,GL.GL_TEXTURE13,GL.GL_TEXTURE14,GL.GL_TEXTURE15,GL.GL_TEXTURE16,
    ↪ GL.GL_TEXTURE17,GL.GL_TEXTURE18,GL.GL_TEXTURE19,GL.GL_TEXTURE20,GL.GL_TEXTURE21,GL.
    ↪ GL_TEXTURE22,GL.GL_TEXTURE23,GL.GL_TEXTURE24,GL.GL_TEXTURE25,GL.GL_TEXTURE26,GL.GL_TEXTURE27,
    ↪ GL.GL_TEXTURE28,GL.GL_TEXTURE29,GL.GL_TEXTURE30,GL.GL_TEXTURE31]

-- | Data for keeping materials
data Material = Material {
  shader :: S.Shader,
  textures :: [(T.Texture,String,Word32)]
}

allocTexture :: Image PixelRGBA8 -> IO GL.GLuint
allocTexture image = unsafeWith (imageData image) $ \ptr ->
    T.createTexture2DStatic (fromIntegral . imageWidth $ image) (fromIntegral . imageHeight $ image)
      (castPtr ptr) T.REPEAT T.NEAREST

-- | Creates a material.
makeMaterial :: (MonadIO m) => S.Shader -- ^ Shader for the material.
```

```
  -> [(FilePath,String)] -- ^ List of Path to and name of uniform sampler2D for each texture.
  -> m (Either String Material)
makeMaterial shader textures = do
  imgDynamic <- mapM (liftIO . readImage . fst) textures
  if null (lefts imgDynamic) then do
    let images = map convertRGBA8 $ rights imgDynamic
    textureNames <- mapM (liftIO . allocTexture) images
    return . Right $ Material shader $ zip3 textureNames textureUniforms [0..]
  else return $ Left $ unlines $ lefts imgDynamic
  where
    textureUniforms = map snd textures

setTexture :: (GL.GLuint,String,Word32) -> IO (S.Uniform ())
setTexture (name,uniform,number) = do
  GL.glActiveTexture number
  GL.glBindTexture GL.GL_TEXTURE_2D name
  return $ S.set uniform number

-- | Draws an entity using a material.
drawWithMat :: (MonadIO m) => Material -> Ent.Entity -> S.Uniform () -> m ()
drawWithMat m e userUniforms = do
  uni <- liftIO $ mapM setTexture $ textures m
  Ent.renderEnt (shader m) e $ do
    sequence_ uni
    userUniforms

-- | Deletes a material. It only frees the textures from gpu memory, the shader is not free as it may be shared with other materials.
deleteMaterial :: (MonadIO m) => Material -> m ()
deleteMaterial m = mapM_ (\(tex,_,_) -> T.deleteTexture tex) $ textures m
```

Listing B.8 Obj.hs

```
-----------------------------------------------------------------------------
--
-- Module : EasyGL.Obj
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Loading and Parsing of Obj files.
--
-----------------------------------------------------------------------------

module EasyGL.Obj (
  EasyGL.Obj.ObjData.Obj(..),
  EasyGL.Obj.ObjData.Group(..),
  EasyGL.Obj.ObjData.FaceNode(..),
  EasyGL.Obj.ObjData.Face(..),
  EasyGL.Obj.ObjData.Shading(..),
  EasyGL.Obj.ObjData.Usemtl(..),
  EasyGL.Obj.ObjData.Mtllib(..),
  EasyGL.Obj.ObjData.IndexBlock(..),
  readObj,
  readObj2Ent,
  obj2Ent,
  toIndexedModel,
```

```
  groupToIndexedModel,

  faceTrianges,
  indexBlockTrianges,
  groupTrianges,
  vertIndexes,
  textureCoordIndex,
  normalsIndex,
  allIndexes,

  faceTriangesS,
  indexBlockTriangesS,
  groupTriangesS,
  vertIndexesS,
  textureCoordIndexS,
  normalsIndexS,
  allIndexesS,

  groupTriangesV,
  vertIndexesV,
  textureCoordIndexV,
  normalsIndexV,
  allIndexesV
  )
where

import Control.Monad.IO.Class (MonadIO, liftIO)
import qualified Data.ByteString.Lazy as BS
import EasyGL.Entity
import EasyGL.Obj.Grammar
import EasyGL.Obj.Obj2IM
import EasyGL.Obj.ObjData
import EasyGL.Obj.Tokens

-- | Parse a String into an Obj.
readObj :: BS.ByteString -> Obj
readObj = parseObj.alexScanTokens

-- | Given the .obj text, generates an entity.
readObj2Ent :: MonadIO m => String -> m Entity
readObj2Ent s = liftIO $ (readObj <$> BS.readFile s) >>= indexedModel2Ent . toIndexedModel

-- | Given an .obj mesh, generates an entity.
obj2Ent :: MonadIO m => Obj -> m Entity
obj2Ent = indexedModel2Ent . toIndexedModel
```

Listing B.9 Grammar.y

```
{

module EasyGL.Obj.Grammar where
import qualified EasyGL.Obj.Tokens as L
import EasyGL.Obj.ObjData
import Graphics.Rendering.OpenGL
import Data.Sequence
import Data.Foldable (toList)
}
```

```
-- %monad { StateT ParseState IO } { (>>=) } { return }
%name parseObj All
%tokentype { L.Token }
%error { parseError }

%token
      Object {L.Object $$}
      Group {L.Group $$}
      Vertex {L.Vertex}
      VertexTexture {L.VertexTexture}
      VertexNormal {L.VertexNormal}
      Face {L.Face}
      TFloat {L.TFloat $$}
      TInt {L.TInt $$}
      TUInt {L.TUInt $$}
      '/' { L.TDiv }
      Usemtl {L.Usemtl $$}
      Mtllib {L.Mtllib $$}
      Shadingoff {L.ShadingOff}
      Shading{L.Shading $$}
      GroupV{L.GroupV $$}

%%

All : Mtllib Groups { Obj Nothing (Just $1) (toList $2) }
   | Groups { Obj Nothing Nothing (toList $1) }

toFloat : TFloat { $1 }
   | TInt { fromIntegral $1 :: GLfloat}
   | TUInt { fromIntegral $1 :: GLfloat}


Groups : GroupBlock { singleton $1 }
   | Groups GroupBlock { $1 |> $2 }


GroupBlock : Object Geometry Faces { (\(v,tc,n) -> Group (Just $1) v tc n $3 ) $2 }
   | GroupV toFloat toFloat toFloat Geometry Faces { (\(v,tc,n) -> Group (Just $1) ((Vertex3 $2 $3 $4) <| v) tc n $6 ) $5 }
   | Geometry Faces { (\(v,tc,n) -> Group Nothing v tc n $2 ) $1 }


Faces : FaceLineBlock { singleton (IndexBlock Nothing Nothing Nothing $1) }
   | FaceBlock { $1 }


FaceBlock : FaceBlockHeader FaceLineBlock { singleton ((\(name,shad,mat) -> IndexBlock name shad mat $2 ) $1) }
   | FaceBlock FaceBlockHeader FaceLineBlock { $1 |> ((\(name,shad,mat) -> IndexBlock name shad mat $3 ) $2) }


FaceBlockHeader : ShadingLine { (Nothing,$1,Nothing) }
   | Usemtl { (Nothing,Nothing,Just $1) }
   | Group { (Just $1,Nothing,Nothing) }
   | Group ShadingLine { (Just $1,$2,Nothing) }
   | Group Usemtl { (Just $1,Nothing,Just $2) }
   | ShadingLine Group { (Just $2,$1,Nothing) }
   | Usemtl Group { (Just $2,Nothing,Just $1) }
   | ShadingLine Usemtl { (Nothing,$1,Just $2) }
   | Usemtl ShadingLine { (Nothing,$2,Just $1) }
   | Group ShadingLine Usemtl { (Just $1,$2,Just $3) }
```

```
  | Group Usemtl ShadingLine { (Just $1,$3,Just $2) }
  | ShadingLine Group Usemtl { (Just $2,$1,Just $3) }
  | Usemtl Group ShadingLine { (Just $2,$3,Just $1) }
  | ShadingLine Usemtl Group { (Just $3,$1,Just $2) }
  | Usemtl ShadingLine Group { (Just $3,$2,Just $1) }


Geometry : VertexBlock { ($1,Nothing,Nothing) }
  | VertexBlock VertexTextureBlock { ($1,Just $2,Nothing) }
  | VertexBlock VertexNormalBlock { ($1,Nothing,Just $2) }
  | VertexBlock VertexTextureBlock VertexNormalBlock { ($1,Just $2,Just $3) }
  | VertexBlock VertexNormalBlock VertexTextureBlock { ($1,Just $3,Just $2) }


ShadingLine : Shadingoff { Nothing }
  | Shading { Just $1 }


VertexLine : Vertex toFloat toFloat toFloat { Vertex3 $2 $3 $4 }
  | Vertex toFloat toFloat toFloat toFloat { Vertex3 $2 $3 $4 } -- some implementations use an optional w (x,y,z[,w]), we ignore it.

VertexBlock : VertexLine { singleton $1 }
  | VertexBlock VertexLine { $1 |> $2 }


VertexTextureLine : VertexTexture toFloat toFloat { Vector2 $2 $3 }
  | VertexTexture toFloat toFloat toFloat { Vector2 $2 $3 } -- some implementations use an optional w (u, v [,w]), we ignore it.

VertexTextureBlock : VertexTextureLine { singleton $1 }
  | VertexTextureBlock VertexTextureLine { $1 |> $2 }


VertexNormalLine : VertexNormal toFloat toFloat toFloat { Vector3 $2 $3 $4 }

VertexNormalBlock : VertexNormalLine { singleton $1 }
  | VertexNormalBlock VertexNormalLine { $1 |> $2 }


FaceNode : TUInt { FaceNode ($1-1) Nothing Nothing }
  | TUInt '/' TUInt { FaceNode ($1-1) (Just ($3-1)) Nothing }
  | TUInt '/' TUInt '/' TUInt { FaceNode ($1-1) (Just ($3-1)) (Just ($5-1)) }
  | TUInt '/' '/' TUInt { FaceNode ($1-1) Nothing (Just ($4-1)) }

FaceLine : Face FaceNodeList { $2 }

FaceNodeList : FaceNode { singleton $1 }
  | FaceNodeList FaceNode { $1 |> $2 }

FaceLineBlock : FaceLine { singleton $1 }
  | FaceLineBlock FaceLine { $1 |> $2 }

{

--errStrPut = hPutStrLn stderr

--printError = lift.errStrPut

--sprint = lift.putStrLn
```

```
parseError :: [L.Token] -> b
parseError t = error ("Parse␣error:␣" ++ show t)

}
```

Listing B.10 Obj2IM.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : EasyGL.Obj.Obj2IM
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Functions required to transform a Obj to an EasyGL Indexed Model.
--
-----------------------------------------------------------------------------

module EasyGL.Obj.Obj2IM (
toIndexedModel,
groupToIndexedModel
  )
where

import Control.Monad.Reader
import Control.Monad.ST
import Control.Monad.State.Lazy
import Data.Foldable (toList)
import qualified Data.Map.Strict as Map
import qualified Data.Maybe as M
import qualified Data.Sequence as Seq
import qualified Data.Vector as V
import qualified Data.Vector.Mutable as VM
import qualified Data.Vector.Storable as VS
import qualified EasyGL.IndexedModel as IM
import EasyGL.Obj.ObjData
import Graphics.Rendering.OpenGL hiding (get)

defaultNormal :: Maybe (Vector3 GLfloat) -> Vector3 GLfloat
defaultNormal (Just x) = x
defaultNormal Nothing = Vector3 0 0 0

defaultTexture :: Maybe (Vector2 GLfloat) -> Vector2 GLfloat
defaultTexture (Just x) = x
defaultTexture Nothing = Vector2 0 0

-- | A needed correction to all textureCoord, as image loading library loads everything upside down, and it is easier to correct here.
easyGLVector2Correction :: Vector2 GLfloat -> Vector2 GLfloat
easyGLVector2Correction (Vector2 x y) = Vector2 x (1-y)

-- | Turns every group within an Obj into an IndexedModel.
toIndexedModel :: Obj -> [IM.IndexedModel]
toIndexedModel = map groupToIndexedModel . groups

groupToIndexedModel :: Group -> IM.IndexedModel
groupToIndexedModel g
```

```
    | M.isNothing norm0 && M.isNothing tex0 = IM.IndexedModel (V.convert vert0) VS.empty VS.empty (V.convert index0)
    | M.isJust norm0 && M.isNothing tex0 = runST $ do
      (newVert,newNorms,newIndex) <- rearrangeCaller defaultNormal vert0 index0 (V.fromList . toList . M.fromJust $ norm0) normIndex0
      return $ IM.IndexedModel (V.convert newVert) (V.convert newNorms) VS.empty (V.convert newIndex)
    | M.isNothing norm0 && M.isJust tex0 = runST $ do
      (newVert,newText,newIndex) <- rearrangeCaller defaultTexture vert0 index0 (V.fromList . toList . M.fromJust $ tex0) textIndex0
      return $ IM.IndexedModel (V.convert newVert) VS.empty (V.convert . V.map easyGLVector2Correction $ newText) (V.convert newIndex)
    | M.isJust norm0 && M.isJust tex0 = runST $ do
      (newVert,newNorms,newIndex) <- rearrangeCaller defaultNormal vert0 index0 (V.fromList . toList . M.fromJust $ norm0) normIndex0
      let newThing = V.zip newVert newNorms
      (newThing,newText,newIndex) <- rearrangeCaller defaultTexture newThing newIndex (V.fromList . toList . M.fromJust $ tex0)
          ↪ textIndex0
      let (newVert,newNorms) = V.unzip newThing
      return $ IM.IndexedModel (V.convert newVert) (V.convert newNorms) (V.convert . V.map easyGLVector2Correction $ newText) (V.
          ↪ convert newIndex)
    | otherwise = IM.emptyIndexedModel
  where
    vert0 = V.fromList . toList $ groupVertices g
    norm0 = groupNormals g
    tex0 = groupTextureCoord g
    (index0,textIndex0,normIndex0) = allIndexesV g

rearrangeCaller :: (Ord a,Ord b,Integral c,Num c) => (Maybe b -> b) -> V.Vector a -> V.Vector c -> V.Vector b -> V.Vector c -> ST s (V.
    ↪ Vector a,V.Vector b,V.Vector c)
rearrangeCaller f vectorA indexA vectorB indexB = do
  mutableIndex <- V.thaw indexA
  acc <- VM.replicate (V.length vectorA) Nothing
  let reader = RearrangeRead vectorA vectorB acc mutableIndex indexB
  (RearrangeState _ extraA extraB _) <- runReaderT (execStateT rearrange defaultRearrangeState) reader
  freezed <- V.freeze acc
  let returnA = vectorA V.++ (V.fromList . toList $ extraA)
      returnB = V.map f freezed V.++ (V.fromList . toList $ extraB)
  returnC <- V.freeze mutableIndex
  return (returnA,returnB,returnC)

data RearrangeState a b c = RearrangeState {
  current :: Int,
  extraA :: Seq.Seq a,
  extraB :: Seq.Seq b,
  locator :: Map.Map (a,b) c
}

defaultRearrangeState :: RearrangeState a b c
defaultRearrangeState = RearrangeState 0 Seq.empty Seq.empty Map.empty

data RearrangeRead a b c s = RearrangeRead {
  readVetorA :: V.Vector a,
  readVetorB :: V.Vector b,
  resulVector :: VM.MVector s (Maybe b),
  indexA :: VM.MVector s c,
  indexB :: V.Vector c
}

type RearrangeM a b c s = StateT (RearrangeState a b c) (ReaderT (RearrangeRead a b c s) (ST s)) ()

rearrange :: (Ord a,Ord b,Integral c,Num c) => RearrangeM a b c s
rearrange = do
  readdata <- ask
  state <- get
```

```
  unless (current state == VM.length (indexA readdata)) $ do
    currentIndexA <- VM.read (indexA readdata) (fromIntegral $ current state)
    let currentIndexB = indexB readdata V.! fromIntegral (current state)
        currentA = readVetorA readdata V.! fromIntegral currentIndexA
        currentB = readVetorB readdata V.! fromIntegral currentIndexB
    currentMaybeB <- VM.read (resulVector readdata) (fromIntegral currentIndexA)
    if M.isNothing currentMaybeB then
      VM.write (resulVector readdata) (fromIntegral currentIndexA) (Just currentB)
    else
      unless (currentB == M.fromJust currentMaybeB) $ do
        let search = Map.lookup (currentA,currentB) $ locator state
        case search of
          Just x -> VM.write (indexA readdata) (fromIntegral $ current state) x
          Nothing -> do
            let newIndex = fromIntegral $ V.length (readVetorA readdata) + Seq.length (extraA state)
            VM.write (indexA readdata) (fromIntegral $ current state) newIndex
            modify' (\s-> s{
                    extraA=extraA s Seq.|> currentA,
                    extraB=extraB s Seq.|> currentB,
                    locator=Map.insert (currentA,currentB) newIndex (locator s)
                  }
                )
    modify' (\s->s{current=current state + 1})
    rearrange
```

Listing B.11 ObjData.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : EasyGL.Obj.ObjData
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Data types for storing a parsed Obj file.
--
-----------------------------------------------------------------------------

module EasyGL.Obj.ObjData (
  Obj(..),
  Group(..),
  FaceNode(..),
  Face(..),
  Shading(..),
  Usemtl(..),
  Mtllib(..),
  IndexBlock(..),
  faceTrianges,
  indexBlockTrianges,
  groupTrianges,
  vertIndexes,
  textureCoordIndex,
  normalsIndex,
  allIndexes,

  faceTriangesS,
```

```
    indexBlockTriangesS,
    groupTriangesS,
    vertIndexesS,
    textureCoordIndexS,
    normalsIndexS,
    allIndexesS,

    groupTriangesV,
    vertIndexesV,
    textureCoordIndexV,
    normalsIndexV,
    allIndexesV
)
where

import Control.Monad.ST
import qualified Data.ByteString.Lazy as BS
import Data.Foldable (toList)
import Data.Maybe (fromJust, mapMaybe)
import Data.Sequence hiding (index)
import qualified Data.Sequence as S
import qualified Data.Vector as V
import qualified Data.Vector.Mutable as VM
import Graphics.Rendering.OpenGL (GLfloat, GLuint, Vector2, Vector3,
                                  Vertex3)

-- | Some Obj's often have more than one mesh, a Group is one such mesh within an Obj.
data Group = Group {
    groupName :: !(Maybe BS.ByteString),
    groupVertices :: !(Seq (Vertex3 GLfloat)),
    groupTextureCoord :: !(Maybe (Seq (Vector2 GLfloat))),
    groupNormals :: !(Maybe (Seq (Vector3 GLfloat))),
    index :: !(Seq IndexBlock)
} deriving (Show)

-- | Represents data of an Obj file.
data Obj = Obj {
    objName :: !(Maybe BS.ByteString),
    mtllib :: Mtllib,
    groups :: [Group]
} deriving (Show)

-- | Some Obj's formats support materials, that are applied to some faces within a mesh,
-- an IndexBlock represents a set of faces with an applied material.
data IndexBlock = IndexBlock {
    indexBlockame :: Maybe BS.ByteString,
    shading :: Shading,
    usemtl :: Usemtl,
    faces :: Seq Face
} deriving (Show)

data FaceNode = FaceNode {
    nodeIndex :: GLuint,
    nodeTextureCoords :: Maybe GLuint,
    nodeNormal :: Maybe GLuint
} deriving (Show)

-- | A face within a mesh, might be composed of more than 3 points.
type Face = Seq FaceNode
```

```
type Shading = Maybe Int

type Usemtl = Maybe BS.ByteString

type Mtllib = Maybe BS.ByteString


-------------------------------------------------------------------------------
-- Utility Functions
-------------------------------------------------------------------------------

-- | Obj faces can contain more that 3 index, but for a face to be use in computer graphics,
-- it needs to be in the form of triangles, this function turns faces into triangles,
-- every 3 elements of the returned list will be a triangle.
faceTrianges :: Face -> [FaceNode]
faceTrianges f = if S.null f then [] else aux asList
  where
    (pivot :< rest) = viewl f
    asList = toList rest
    aux [x] = []
    aux (x:y:xs) = pivot:x:y:aux (y:xs)

indexBlockTrianges :: IndexBlock -> [FaceNode]
indexBlockTrianges = concatMap faceTrianges . faces

groupTrianges :: Group -> [FaceNode]
groupTrianges = concatMap indexBlockTrianges . index

-- | Returns a list of the vertex indexes as triangles.
vertIndexes :: Group -> [GLuint]
vertIndexes = map nodeIndex . groupTrianges

-- | Returns a list of the texture coordinates indexes as triangles.
textureCoordIndex :: Group -> [GLuint]
textureCoordIndex = mapMaybe nodeTextureCoords . groupTrianges

-- | Returns a list of the normal indexes as triangles.
normalsIndex :: Group -> [GLuint]
normalsIndex = mapMaybe nodeNormal . groupTrianges

-- | Returns the vertex indexes, texture coordinates indexes and normal indexes of a group.
allIndexes :: Group -> ([GLuint],[GLuint],[GLuint])
allIndexes g = (map nodeIndex triangles,
    mapMaybe nodeTextureCoords triangles,
    mapMaybe nodeNormal triangles)
  where
    triangles = groupTrianges g

-------------------------------------------------------------------------------
-- Same as above but with Seq.
-------------------------------------------------------------------------------

faceTriangesS :: Face -> S.Seq FaceNode
faceTriangesS f = if S.null f then S.empty else aux $ viewl rest
  where
    (pivot :< rest) = viewl f
    aux (x :< sx) = case viewl sx of
      S.EmptyL -> S.empty
```

```
      (y :< _) −> (S.singleton pivot |> x |> y) S.>< aux (viewl sx)

indexBlockTriangesS :: IndexBlock −> S.Seq FaceNode
indexBlockTriangesS = foldl (S.><) S.empty . fmap faceTriangesS . faces

groupTriangesS :: Group −> S.Seq FaceNode
groupTriangesS = foldl (S.><) S.empty . fmap indexBlockTriangesS . index

vertIndexesS :: Group −> S.Seq GLuint
vertIndexesS = fmap nodeIndex . groupTriangesS

textureCoordIndexS :: Group −> S.Seq GLuint
textureCoordIndexS = fmap (fromJust . nodeTextureCoords) . groupTriangesS

normalsIndexS :: Group −> S.Seq GLuint
normalsIndexS = fmap (fromJust . nodeNormal) . groupTriangesS

allIndexesS :: Group −> (S.Seq GLuint,S.Seq GLuint,S.Seq GLuint)
allIndexesS g = (fmap nodeIndex triangles,
    fmap (fromJust . nodeTextureCoords) triangles,
    fmap (fromJust . nodeNormal) triangles)
  where
    triangles = groupTriangesS g

−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
−− Same as above but with vectors.
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

faceTriangesNum :: Face −> Int
faceTriangesNum = subtract 2 . S.length

indexBlockTriangesNum :: IndexBlock −> Int
indexBlockTriangesNum = sum . fmap faceTriangesNum . faces

groupTriangesNum :: Group −> Int
groupTriangesNum = sum . fmap indexBlockTriangesNum . index

faceTriangesV :: VM.MVector s FaceNode −> Face −> Int −> ST s Int
faceTriangesV arr f i = aux i $ viewl rest
  where
    (pivot :< rest) = viewl f
    aux index (x :< sx) = case viewl sx of
      S.EmptyL −> return index
      (y :< _) −> do
        VM.write arr index pivot
        VM.write arr (index+1) x
        VM.write arr (index+2) y
        aux (index + 3) (viewl sx)

indexBlockTriangesV :: VM.MVector s FaceNode −> IndexBlock −> Int −> ST s Int
indexBlockTriangesV arr ib i = foldr ((=<<) . faceTriangesV arr) (return i) $ faces ib

groupTriangesV :: Group −> V.Vector FaceNode
groupTriangesV g = V.create $ do
  let size = groupTriangesNum g ∗ 3
  arr <− VM.unsafeNew size −− i don't care about initialization.
  foldr ((=<<) . indexBlockTriangesV arr) (return 0) $ index g
  return arr
```

```
-- | Returns a list of the vertex indexes as triangles.
vertIndexesV :: Group -> V.Vector GLuint
vertIndexesV = V.map nodeIndex . groupTriangesV

-- | Returns a list of the texture coordinates indexes as triangles.
textureCoordIndexV :: Group -> V.Vector GLuint
textureCoordIndexV = V.map (fromJust . nodeTextureCoords) . groupTriangesV

-- | Returns a list of the normal indexes as triangles.
normalsIndexV :: Group -> V.Vector GLuint
normalsIndexV = V.map (fromJust . nodeNormal) . groupTriangesV

-- | Returns the vertex indexes, texture coordinates indexes and normal indexes of a group.
allIndexesV :: Group -> (V.Vector GLuint,V.Vector GLuint,V.Vector GLuint)
allIndexesV g = (V.map nodeIndex triangles,
    V.map (fromJust . nodeTextureCoords) triangles,
    V.map (fromJust . nodeNormal) triangles)
  where
    triangles = groupTriangesV g
```

Listing B.12 Tokens.x

```
{
module EasyGL.Obj.Tokens where
import Graphics.Rendering.OpenGL
import qualified Data.Maybe as M
import qualified Data.ByteString.Lazy as BS
import qualified Data.ByteString.Lazy.Char8 as BS8
import qualified Data.ByteString.Conversion as BSC
}

%wrapper "posn-bytestring"
$graphic = [\.\;\,\$\|\*\+\?\#\~\-\{\}\(\)\[\]\^\/\\\ 0-9a-zA-Z]

@entero10 = [0-9][0-9]*
@float1 = [\-]? @entero10 "." @entero10 [eE] [\-\+] @entero10 | @entero10 [eE] [\-\+] @entero10
@float2 = [\-]? @entero10 "." @entero10
@signedInteger = [\-]? @entero10

@vertex = "v␣"

@vertexTexture = "vt␣"

@vertexNormal = "vn␣"

@face = "f␣"

@object = "o␣" .*

@groupVertex = "g␣" .* [\n] "v␣"

@group = "g␣" .*

@div = "/"

@usemtl = "usemtl␣" .*

@mtllib = "mtllib␣" .*
```

```
@shadingoff = "s␣off"

@shading = "s␣" @entero10

tokens :−

$white+ ; −−espacios
"#".* ; −−comentarios de resto de linea
@usemtl{ \(AlexPn _ l c) s −> Usemtl (BS8.unwords . tail . BS8.words $ s)
}
@mtllib{ \(AlexPn _ l c) s −> Mtllib (BS8.unwords . tail . BS8.words $ s)
}
@shadingoff{ \(AlexPn _ l c) s −> ShadingOff
}
@shading{ \(AlexPn _ l c) s −> Shading (M.fromJust . BSC.fromByteString' . BS.drop 2 $ s)
}

@groupVertex{ \(AlexPn _ l c) s −> GroupV (BS8.unwords . tail . BS8.words . head . BS8.lines $ s)
}
@group{ \(AlexPn _ l c) s −> Group (BS8.unwords . tail . BS8.words $ s)
}
@object{ \(AlexPn _ l c) s −> Object (BS8.unwords . tail . BS8.words $ s)
}

@div { \(AlexPn _ l c) s −> TDiv
}

@vertex { \(AlexPn _ l c) s −> Vertex
}

@vertexTexture { \(AlexPn _ l c) s −> VertexTexture
}

@vertexNormal { \(AlexPn _ l c) s −> VertexNormal
}

@face { \(AlexPn _ l c) s −> Face
}

@float1 { \(AlexPn _ l c) s −> TFloat (realToFrac (M.fromJust (BSC.fromByteString' s) :: Double))
}

@float2 { \(AlexPn _ l c) s −> TFloat (realToFrac (M.fromJust (BSC.fromByteString' s) :: Double))
}

@entero10 { \(AlexPn _ l c) s −> TUInt (M.fromJust $ BSC.fromByteString' s)
}

@signedInteger { \(AlexPn _ l c) s −> TInt (M.fromJust $ BSC.fromByteString' s)
}


{

data Token = Vertex | VertexTexture | VertexNormal | Face | Group BS.ByteString | GroupV BS.ByteString | Object BS.ByteString | TDiv |
  TFloat GLfloat | TUInt GLuint | TInt GLint | Usemtl BS.ByteString | Mtllib BS.ByteString | Shading Int | ShadingOff deriving (Show)

}
```

Listing B.13 Shader.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : EasyGL.Shader
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Provides an easy to use interface to OpenGL shaders and programs, from creation to usage.
--
-----------------------------------------------------------------------------

module EasyGL.Shader (
  GL.ShaderType(..),
  Shader,
  loadShadersFromFile,
  loadShadersFromBS,
  deleteShader,
  withShader,
  putActiveUniforms,
  setVar,
  setArr,
  setArrLen,
  setArrPtr,
  Uniform,
  withShaderSafe,
  set,
  setF,
  setP,
  uniforms
)
where

import Control.Monad
import Control.Monad.IO.Class (MonadIO, liftIO)
import qualified Data.ByteString as BS
import Data.Foldable (Foldable, toList)
import Data.StateVar (get, ($=!))
import Foreign.Marshal.Array (withArray, withArrayLen)
import Foreign.Ptr (Ptr)
import Foreign.Storable (Storable)
import qualified Graphics.Rendering.OpenGL as GL
import System.IO (Handle, hPutStr)

{-|
  Defines an simple shader, that contains all compiled code gpu-ready.
-}
newtype Shader = Shader {
  program :: GL.Program}

empty :: MonadIO m => m Shader
empty = liftIO $ fmap Shader GL.createProgram

mensajes :: Maybe Handle -> GL.GettableStateVar String -> IO ()
mensajes mhandle var =
  maybe (return ())
```

```
        (\h -> do
          s <- get var
          unless (s == "\NUL" || s == "No errors.\n\NUL") $ hPutStr h s
        )
        mhandle

loadShaderFromFile :: String -> GL.ShaderType -> IO GL.Shader
loadShaderFromFile s shadertype = do
  out <- GL.createShader shadertype
  BS.readFile s >>= (GL.shaderSourceBS out $=!)
  GL.compileShader out
  return out

loadShaderFromBS :: BS.ByteString -> GL.ShaderType -> IO GL.Shader
loadShaderFromBS bs shadertype = do
    out <- GL.createShader shadertype
    GL.shaderSourceBS out $=! bs
    GL.compileShader out
    return out

{-|
  Creates a simple shader from files.
-}
loadShadersFromFile :: MonadIO m => [String] -- ^ FilePaths to each shader to load.
  -> [GL.ShaderType] -- ^ Shader type to each shader to load. See http://hackage.haskell.org/package/OpenGL-3.0.1.0/docs/Graphics-
      ↪ Rendering-OpenGL-GL-Shaders-ShaderObjects.html#t:ShaderType
  -> Maybe Handle -- ^ If given a file handle, will print all compile errors to given handle
  -> m Shader
loadShadersFromFile s st mhandle = liftIO $ zipWithM loadShaderFromFile s st >>= linkShaders mhandle

{-|
  Creates a simple shader from byte strings.
-}
loadShadersFromBS :: MonadIO m => [BS.ByteString] -> [GL.ShaderType] -> Maybe Handle -> m Shader
loadShadersFromBS s st mhandle = liftIO $ zipWithM loadShaderFromBS s st >>= linkShaders mhandle

bindEasyGLShaderAttrib :: GL.Program -> IO ()
bindEasyGLShaderAttrib shaderProgram = do
  GL.attribLocation shaderProgram "position" $=! GL.AttribLocation 0
  GL.attribLocation shaderProgram "textCoord" $=! GL.AttribLocation 1
  GL.attribLocation shaderProgram "normal" $=! GL.AttribLocation 2

linkShaders :: Maybe Handle -> [GL.Shader] -> IO Shader
linkShaders mhandle shaders = do
  active <- empty
  mapM_ (GL.attachShader (program active)) shaders
  bindEasyGLShaderAttrib (program active)
  GL.linkProgram (program active)
  mapM_ (mensajes mhandle . GL.shaderInfoLog) shaders
  mapM_ GL.deleteObjectName shaders -- flags shaders for deletion
  mensajes mhandle $ GL.programInfoLog (program active)
  return active

{-|
  Frees the memory and invalidates the name associated with the shader
-}
deleteShader :: MonadIO m => Shader -> m ()
deleteShader = GL.deleteObjectName . program
-- GL.Shader objects where mark for deletion on linkShaders, no other action is required.
```

```
{-|
  Specifies an action to be made using a given shader.
-}
withShader :: MonadIO m => Shader -> m a -> m a
withShader s action = do
  pastProgram <- liftIO $ GL.get GL.currentProgram
  liftIO $ GL.currentProgram $=! Just (program s)
  ret <- action
  liftIO $ GL.currentProgram $=! pastProgram
  return ret

{-|
  Prints to stdout all active uniforms variables of currently selected shader.
-}
putActiveUniforms :: MonadIO m => m ()
putActiveUniforms = liftIO $ do
  pro <- get GL.currentProgram
  maybe (return ())
        (GL.activeUniforms >=> print)
        pro

{-|
  Sets an uniform variable of currently selected shader to a given value.
  Warning: does not checks types of provide value against variable.
-}
setVar :: (MonadIO m,GL.Uniform a) => a -> String -> m ()
setVar val str = liftIO $ do
  pro <- get GL.currentProgram
  maybe (return ())
        (\x->do
          loc <- GL.uniformLocation x str
          GL.uniform loc $=! val
        )
        pro

{-|
Sets an uniform variable of currently selected shader to a given array of values.
Warning: does not checks types of provide value against variable.
-}
setArr :: (MonadIO m,Storable a, GL.Uniform a, Integral b) => [a] -> b -> String -> m ()
setArr vals len str = liftIO $ do
  pro <- get GL.currentProgram
  maybe (return ())
        (\x->do
          loc <- GL.uniformLocation x str
          withArray vals $ \ptr -> GL.uniformv loc (fromIntegral len) ptr
        )
        pro

{-|
Same as setArr but the number of elements is calculated.
-}
setArrLen :: (MonadIO m,Storable a, GL.Uniform a) => [a] -> String -> m ()
setArrLen vals str = liftIO $ do
  pro <- get GL.currentProgram
  maybe (return ())
        (\x->do
          loc <- GL.uniformLocation x str
```

```
            withArrayLen vals $ \tam ptr -> GL.uniformv loc (fromIntegral tam) ptr
        )
        pro

{-|
Same as setArr but the user provides the pointer to data and then number of elements.
-}
setArrPtr :: (MonadIO m, Integral b, GL.Uniform a) => Ptr a -> b -> String -> m ()
setArrPtr ptr len str = liftIO $ do
  pro <- get GL.currentProgram
  maybe (return ())
        (\x->do
          loc <- GL.uniformLocation x str
          GL.uniformv loc (fromIntegral len) ptr
        )
        pro

------------------------------------------------------------------------------
-- Next section describes uniform variables as a monad to increase safety.
------------------------------------------------------------------------------

-- | Monad that describes uniform variables operations.
newtype Uniform a = Uniform { runUniform :: IO a }

instance Functor Uniform where
    fmap f (Uniform v) = Uniform (fmap f v)

instance Applicative Uniform where
    pure = Uniform . pure
    Uniform f <*> Uniform v = Uniform (f <*> v)

instance Monad Uniform where
    return = pure
    (>>=) (Uniform a) f = Uniform $ fmap f a >>= runUniform
    (>>) (Uniform a) (Uniform b) = Uniform (a >> b)

{-|
  Specifies an action to be made using a given shader.
  Safety comes from designating a monad to perform only the uniform variables operations.
-}
withShaderSafe :: MonadIO m => Shader -- ^ Shader to use.
  -> Uniform () -- ^ Uniform variables setting actions.
  -> m a -- ^ OpenGL function to render an object.
  -> m a
withShaderSafe s u action = do
  pastProgram <- liftIO $ GL.get GL.currentProgram
  liftIO $ GL.currentProgram $=! Just (program s)
  liftIO $ runUniform u
  ret <- action
  liftIO $ GL.currentProgram $=! pastProgram
  return ret

-- | Sets an uniform variable to a value.
set :: GL.Uniform a => String -> a -> Uniform ()
set str a = Uniform $ setVar a str

-- | Sets an uniform array to the values in a foldable container.
setF :: (Foldable t,Storable a, GL.Uniform a) => String -> t a -> Uniform ()
setF str container = Uniform $ setArrLen (toList container) str
```

```
-- | Given a pointer to data and the number of elements, sets an uniform array to data in pointer.
setP :: (Integral b, GL.Uniform a) => String -> Ptr a -> b -> Uniform ()
setP str ptr len = Uniform $ setArrPtr ptr len str

-- | Prints the uniform variables of a shader.
uniforms :: Uniform ()
uniforms = Uniform putActiveUniforms
```

Listing B.14 Texture.hs

```
--------------------------------------------------------------------------------
-- |
-- Module : EasyGL.Texture
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Helps to load and manage textures in the gpu.
--
--------------------------------------------------------------------------------

module EasyGL.Texture (
  Texture,
  ClampType(..),
  FilteringType(..),
  createTexture2DStatic,
  deleteTexture
)
where
import Control.Monad.IO.Class
import Foreign.Marshal.Utils
import Foreign.Ptr
import Foreign.Storable
import qualified Graphics.GL as GL

-- | Textue clamping alternatives.
data ClampType = REPEAT | MIRRORED_REPEAT | CLAMP2EDGE | CLAMP2BORDER
clampType2Number :: ClampType -> GL.GLint
clampType2Number REPEAT = fromIntegral GL.GL_REPEAT
clampType2Number MIRRORED_REPEAT = fromIntegral GL.GL_MIRRORED_REPEAT
clampType2Number CLAMP2EDGE = fromIntegral GL.GL_CLAMP_TO_EDGE
clampType2Number CLAMP2BORDER = fromIntegral GL.GL_CLAMP_TO_BORDER

-- | Texture filtering alternatives.
data FilteringType = NEAREST | LINEAR
filteringType2Number :: FilteringType -> GL.GLint
filteringType2Number NEAREST = fromIntegral GL.GL_NEAREST
filteringType2Number LINEAR = fromIntegral GL.GL_LINEAR

type Texture = GL.GLuint

-- | Creates and loads to gpu a texture.
createTexture2DStatic :: (Integral a,MonadIO m) => a -> a -> Ptr () -> ClampType -> FilteringType -> m Texture
createTexture2DStatic width height dataPtr ct ft = liftIO $ with 0 $ \name -> do
  GL.glGenTextures 1 name
```

```
  val <- peek name
  GL.glBindTexture GL.GL_TEXTURE_2D val
  GL.glTexParameteri GL.GL_TEXTURE_2D GL.GL_TEXTURE_WRAP_S $ clampType2Number ct
  GL.glTexParameteri GL.GL_TEXTURE_2D GL.GL_TEXTURE_WRAP_T $ clampType2Number ct
  GL.glTexParameteri GL.GL_TEXTURE_2D GL.GL_TEXTURE_MIN_FILTER $ filteringType2Number ft
  GL.glTexParameteri GL.GL_TEXTURE_2D GL.GL_TEXTURE_MAG_FILTER $ filteringType2Number ft
  GL.glTexImage2D GL.GL_TEXTURE_2D 0 (fromIntegral GL.GL_RGBA) (fromIntegral width) (fromIntegral height) 0 GL.GL_RGBA GL
      ↪ .GL_UNSIGNED_BYTE dataPtr
  GL.glGenerateMipmap GL.GL_TEXTURE_2D
  return val

-- | Deletes from gpu a texture.
deleteTexture :: (MonadIO m) => Texture -> m ()
deleteTexture val = liftIO $ with val $ \ptr -> GL.glDeleteTextures 1 ptr
```

Listing B.15 Util.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : EasyGL.Util
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Math functions for Graphics.Rendering.OpenGL types.
--
-----------------------------------------------------------------------------

module EasyGL.Util (
  normVec2,
  normalizeVec2,
  dotVec2,
  normVec3,
  normalizeVec3,
  dotVec3,
  crossVec3,
  normVec4,
  normalizeVec4,
  dotVec4)
where

import qualified Graphics.Rendering.OpenGL as GL

normVec2 :: (Floating a) => GL.Vector2 a -> a
normVec2 (GL.Vector2 x y) = sqrt $ x**2 + y**2

normalizeVec2 :: (Floating a) => GL.Vector2 a -> GL.Vector2 a
normalizeVec2 v@(GL.Vector2 x y) = GL.Vector2 (x/norm) (y/norm)
  where
    norm = normVec2 v

dotVec2 :: (Num a) => GL.Vector2 a -> GL.Vector2 a -> a
dotVec2 (GL.Vector2 x1 y1) (GL.Vector2 x2 y2) = x1*x2 + y1*y2

normVec3 :: (Floating a) => GL.Vector3 a -> a
normVec3 (GL.Vector3 x y z) = sqrt $ x**2 + y**2 + z**2
```

```
normalizeVec3 :: (Floating a) => GL.Vector3 a -> GL.Vector3 a
normalizeVec3 v@(GL.Vector3 x y z) = GL.Vector3 (x/norm) (y/norm) (z/norm)
  where
    norm = normVec3 v

dotVec3 :: (Num a) => GL.Vector3 a -> GL.Vector3 a -> a
dotVec3 (GL.Vector3 x1 y1 z1) (GL.Vector3 x2 y2 z2) = x1*x2 + y1*y2 + z1*z2

crossVec3 :: (Num a) => GL.Vector3 a -> GL.Vector3 a -> GL.Vector3 a
crossVec3 (GL.Vector3 x1 y1 z1) (GL.Vector3 x2 y2 z2) =
  GL.Vector3 (y1*z2-z1*y2) (z1*x2-x1*z2) (x1*y2-y1*x2)

normVec4 :: (Floating a) => GL.Vector4 a -> a
normVec4 (GL.Vector4 x y z w) = sqrt $ x**2 + y**2 + z**2 + w**2

normalizeVec4 :: (Floating a) => GL.Vector4 a -> GL.Vector4 a
normalizeVec4 v@(GL.Vector4 x y z w) = GL.Vector4 (x/norm) (y/norm) (z/norm) (w/norm)
  where
    norm = normVec4 v

dotVec4 :: (Num a) => GL.Vector4 a -> GL.Vector4 a -> a
dotVec4 (GL.Vector4 x1 y1 z1 w1) (GL.Vector4 x2 y2 z2 w2) = x1*x2 + y1*y2 + z1*z2 + w1*w2

instance (Num a) => Num (GL.Vector2 a) where
  (+) (GL.Vector2 x1 y1) (GL.Vector2 x2 y2) = GL.Vector2 (x1+x2) (y1+y2)
  (-) (GL.Vector2 x1 y1) (GL.Vector2 x2 y2) = GL.Vector2 (x1-x2) (y1-y2)
  (*) (GL.Vector2 x1 y1) (GL.Vector2 x2 y2) = GL.Vector2 (x1*x2) (y1*y2) --This makes no sense, do not use.
  negate (GL.Vector2 x y) = GL.Vector2 (negate x) (negate y)
  abs (GL.Vector2 x y) = GL.Vector2 (abs x) (abs y)
  signum (GL.Vector2 x y) = GL.Vector2 (signum x) (signum y)
  fromInteger x = GL.Vector2 (fromInteger x) (fromInteger x)

instance (Num a) => Num (GL.Vector3 a) where
  (+) (GL.Vector3 x1 y1 z1) (GL.Vector3 x2 y2 z2) = GL.Vector3 (x1+x2) (y1+y2) (z1+z2)
  (-) (GL.Vector3 x1 y1 z1) (GL.Vector3 x2 y2 z2) = GL.Vector3 (x1-x2) (y1-y2) (z1-z2)
  (*) = crossVec3
  negate (GL.Vector3 x y z) = GL.Vector3 (negate x) (negate y) (negate z)
  abs (GL.Vector3 x y z) = GL.Vector3 (abs x) (abs y) (abs z)
  signum (GL.Vector3 x y z) = GL.Vector3 (signum x) (signum y) (signum z)
  fromInteger x = GL.Vector3 (fromInteger x) (fromInteger x) (fromInteger x)
```

Listing B.16 EasyGL.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : Val.Strict
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Val implementation for strict game scenes.
--
-----------------------------------------------------------------------------
```

```
module Val.Strict (
  -- From UI
  UIRead(..),
  UIActions(..),

  -- From Scene
  initScene,
  initScenePar,

  -- From Util
  makeSF,
  makeCamSF,
  deltaTime,
  uiState,

  -- From Events
  keyPress,
  keyDown,
  keyUp,
  keyReleased,
  mouseMoved,
  mouseMovedX,
  mouseMovedY,
  mousePosition,
  mousePositionMoved,
  getObjOut,
  getObjects,
  inputEvent,

  -- From Data
  ResourceIdentifier,
  Resource,
  ResourceMap,

  MergeableEvent(..),
  GameInput(keysGI,mouseGI),
  Object,
  ObjInput,
  IOReq(..),
  ObjOutput(..),
  ILKey,
  IL,

  emptyGameInput,
  emptyObjInput,
  newObjOutput,
  emptyIL,
  lookupIL,
  insertIL,
  insertILWithKey,
  fromList,
  elemsIL,
  assocsIL,
  deleteIL,
  mapIL,
  mapILKeys,
  mapILWithKey,
  modifyIL,
  memberIL,
```

```
  Rotation(..),
  Transform(..),

  -- From Resources.
  loadResouces
)
where

import Val.Strict.Scene
import Val.Strict.Util
import Val.Strict.Events
import Val.Strict.Data
import Val.Strict.UI
import Val.Strict.Scene.Resources
```

Listing B.17 EasyGL.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : Val.Strict.Data
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Data types for Val strict scenes.
--
-----------------------------------------------------------------------------

module Val.Strict.Data (
  ResourceIdentifier,
  Resource,
  ResourceMap,

  MergeableEvent(..),
  GameInput(..),
  Object,
  ObjInput(..),
  IOReq(..),
  IOExec(..),
  ObjOutput(..),

  emptyGameInput,
  toExec,
  emptyObjInput,
  newObjOutput,

  -- From Transform
  Rotation(..),
  valRotate,
  Transform(..),
  useTransform,

  -- Modules
  module Val.Strict.IL
)
```

```
where

import qualified Graphics.Rendering.OpenGL as GL
import FRP.Yampa
import Data.Map.Strict hiding (fromList,union,unions,toList)
import qualified Data.Map.Strict as Map
import Control.DeepSeq
import Control.Seq
import EasyGL
import EasyGLUT
import Data.List (foldl1')
import Data.Foldable (toList)
import qualified EasyGLUT as GLUT
import Control.Exception
import Control.Concurrent.Async
import Val.Strict.UI
import Val.Strict.IL

------------------------------------------------------------------------------
-- About resource generation.
------------------------------------------------------------------------------

-- | A value that identifies a resource.
type ResourceIdentifier = String

-- | Info to load a Resource.
type Resource = (
    ResourceIdentifier, -- name of Resource
    String, -- Path to Obj to load.
    Material -- material to render obj
  )

-- | A map from a ResourceIdentifier to a loaded mesh and material.
type ResourceMap = Map ResourceIdentifier (Entity,Material)

------------------------------------------------------------------------------
-- Game stuff.
------------------------------------------------------------------------------

-- | Describes the type of objects that are event containers and can combine the events.
class MergeableEvent a where
  union :: a -> a -> a

-- | Data about the world sent to game objects.
data GameInput = GameInput {
    keysGI :: Map Key KeyState,
    mouseGI :: MouseState,
    timeGI :: Time,
    uiGI :: UIRead
  }

-- | Creates an empty GameInput.
emptyGameInput :: GameInput
emptyGameInput = GameInput Map.empty (GLUT.FreeMouse 0 0) 0 undefined

-- | Type for game objects.
type Object outState eventType = SF (ObjInput outState eventType) (ObjOutput outState eventType)

-- | Input of a game objects.
```

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

instance (NFData eventType) => NFData (ObjInput state eventType) where
  rnf ObjInput{oiEvents=events} = rnf events -- the game input from the world will normally be normal form because it comes from IO.

-- | Request from a game objects to perform an IO action.
data IOReq a = IOReq {
    ioReqIO :: IO a,
    ioReqError :: SomeException -> a,
    ioBloq :: Bool -- whether to wait or not the io. If True the result will be deliver next frame, otherwise, when result is available.
  }

-- | Asynchronous IO action in execution.
data IOExec a = IOExec {
    ioExecIO :: Async a,
    ioExecError :: SomeException -> a,
    ioExecBloq :: Bool -- whether to wait or not the io. If True the result will be deliver next frame, otherwise, when result is available.
  }

toExec :: IOReq a -> IO (IOExec a)
toExec req = do
  a <- async $ ioReqIO req
  return IOExec{
      ioExecIO=a,
      ioExecError=ioReqError req,
      ioExecBloq=ioBloq req
    }

-- | Output of a game object.
data ObjOutput state eventType = ObjOutput {
  ooObjState :: !state,
  ooRenderer :: Maybe (ResourceIdentifier,Transform,Uniform ()),
  ooKillReq :: Event (),
  ooSpawnReq :: Event [Object state eventType],
  ooWorldReq :: [IOReq eventType], -- request to perform IO
  ooWorldSpawn :: [IO ()], -- request to perform IO, result is not needed, action will be spawn in a new thread and system will forget about it.
  ooUIReq :: [UIActions]
}

emptyObjInput :: ObjInput s a
emptyObjInput = ObjInput undefined emptyGameInput emptyIL

newObjOutput :: a -> ObjOutput a b
newObjOutput state = ObjOutput{ooObjState=state,
  ooRenderer=Nothing,
  ooKillReq=noEvent,
  ooSpawnReq=noEvent,
  ooWorldReq=[],
  ooWorldSpawn=[],
  ooUIReq=[]}

-- Event generators:
-- broadcast - recorre, acumula en un solo map y une al final
-- pair
-- scan and shot, evalua todo por evento y luego decide a quien darselos.
```

```
-----------------------------------------------------------------------------
-- Object Transformation in OpenGL.
-----------------------------------------------------------------------------

-- | Rotation for a given object.
data Rotation = Euler {
        yaw :: GL.GLdouble,
        picth :: GL.GLdouble,
        roll :: GL.GLdouble
      }
  | Quaternion {
        angle :: GL.GLdouble,
        vector :: GL.Vector3 GL.GLdouble
      } deriving Show

-- | Resolves rotations.
valRotate :: Rotation -> IO ()
valRotate (Quaternion angle vector) = GL.rotate angle vector
valRotate (Euler yaw picth roll) = do
  GL.rotate roll (GL.Vector3 0 0 1 )
  GL.rotate picth (GL.Vector3 1 0 0 )
  GL.rotate yaw (GL.Vector3 0 1 0 )

-- | Transformations for a given object.
data Transform = Transform {
     translation :: GL.Vector3 GL.GLdouble,
     rotation :: Rotation,
     scaleX :: GL.GLdouble,
     scaleY :: GL.GLdouble,
     scaleZ :: GL.GLdouble
  } deriving Show

-- | Sets OpenGL matrix to the given transformation.
useTransform :: Transform -> IO ()
useTransform (Transform translation rotation scaleX scaleY scaleZ) = do
  GL.scale scaleX scaleY scaleZ
  --valRotate rotation
  GL.translate translation
  valRotate rotation
```

Listing B.18 EasyGL.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : Val.Strict.Events
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Provides event for the ObjInput that uses GLUT. This module is specially useful if you implement objects using arrows and signal functions, is
      ↪ you are using makeSF you are better off importing EasyGLUT and Val.Data and constructing your conditions yourself.
--
-----------------------------------------------------------------------------
{-# LANGUAGE Arrows #-}
```

```
module Val.Strict.Events (
  --Key Events
  keyPress,
  keyDown,
  keyUp,
  keyReleased,

  --Mouse Fix Events
  mouseMoved,
  mouseMovedX,
  mouseMovedY,

  --Mouse Free Events
  mousePosition,
  mousePositionMoved,

  getObjOut,
  getObjects,
  inputEvent
) where

import EasyGLUT
import FRP.Yampa
import Control.Arrow
import Val.Strict.Data
import Val.Strict.IL
import qualified Data.Map.Strict as Map

-- | Generates an Event if the key have been press this frame.
keyPress :: Key -> ObjInput s a -> Event ()
keyPress k oi = maybePressed NoEvent (Event ()) $ Map.lookup k . keysGI . oiGameInput $ oi

-- | Generates an Event if the key is down.
keyDown :: Key -> ObjInput s a -> Event ()
keyDown k oi = maybeDown NoEvent (Event ()) $ Map.lookup k . keysGI . oiGameInput $ oi

-- | Generates an Event if the key is up.
keyUp :: Key -> ObjInput s a -> Event ()
keyUp k oi = maybeUp NoEvent (Event ()) $ Map.lookup k . keysGI . oiGameInput $ oi

-- | Generates an Event if the key have been released this frame.
keyReleased :: Key -> ObjInput s a -> Event ()
keyReleased k oi = maybeReleased NoEvent (Event ()) $ Map.lookup k . keysGI . oiGameInput $ oi


-- | Generates an Event, containing the x and y movement of the mouse, if the mouse if fixed to a position (see EasyGLUT) and have moved in
    ↪ this frame.
mouseMoved :: ObjInput s a -> Event (Int,Int)
mouseMoved ObjInput{oiGameInput=GameInput{mouseGI=FreeMouse _ _}} = NoEvent
mouseMoved ObjInput{oiGameInput=GameInput{mouseGI=FixMouse x y}} =
  if x /= 0 || y /= 0 then Event (fromIntegral x,fromIntegral y) else NoEvent

-- | Same as mouseMoved but only on the x axis.
mouseMovedX :: ObjInput s a -> Event Int
mouseMovedX ObjInput{oiGameInput=GameInput{mouseGI=FreeMouse _ _}} = NoEvent
mouseMovedX ObjInput{oiGameInput=GameInput{mouseGI=FixMouse x _}} =
  if x /= 0 then Event (fromIntegral x) else NoEvent

-- | Same as mouseMoved but only on the y axis.
```

```
mouseMovedY :: ObjInput s a -> Event Int
mouseMovedY ObjInput{oiGameInput=GameInput{mouseGI=FreeMouse _ _}} = NoEvent
mouseMovedY ObjInput{oiGameInput=GameInput{mouseGI=FixMouse _ y}} =
  if y /= 0 then Event (fromIntegral y) else NoEvent


-- | Generates and event when the mouse have free movement containing the actual mouse position.
mousePosition :: ObjInput s a -> Event (Int,Int)
mousePosition ObjInput{oiGameInput=GameInput{mouseGI=FixMouse _ _}} = NoEvent
mousePosition ObjInput{oiGameInput=GameInput{mouseGI=FreeMouse x y}} =
  Event (fromIntegral x,fromIntegral y)

-- | Generates and event when the mouse have free movement and is in a different position than last time, containing the actual mouse position.
mousePositionMoved :: (Int,Int) -> SF (ObjInput s a) (Event (Int,Int))
mousePositionMoved initialPosition = proc oi -> do
  rec
    mouseThisFrame <- hold initialPosition -< mousePosition oi
    mouseLastFrame <- hold initialPosition -< tag (mousePosition oi) mouseThisFrame
  returnA -< aux mouseThisFrame mouseLastFrame
  where
    aux (x1,y1) (x2,y2) = if x1 /= x2 || y1 /= y2 then Event (x1-x2,y1-y2) else NoEvent


-- | Returns the output state from last frame of the object with the given key if it existed last frame.
getObjOut :: ObjInput state eventType -> ILKey -> Maybe state
getObjOut oi key = lookupIL key $ oiPastFrame oi

-- | Retruns the objects from last frame along with its key.
getObjects :: ObjInput state eventType -> [(ILKey,state)]
getObjects = assocsIL . oiPastFrame

-- | Returns the event combinator.
inputEvent :: ObjInput s a -> Event a
inputEvent = oiEvents
```

Listing B.19 EasyGL.hs

```
-----------------------------------------------------------------------------
--  |
--  Module : Val.Strict.IL
--  Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
--  License : BSD3
--
--  Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
--  Stability : stable
--  Portability : portable
--
--  A module that makes OpenGL easier in haskell.
--
-----------------------------------------------------------------------------

module Val.Strict.IL (
  ILKey,
  IL(..),
  emptyIL,
  lookupIL,
  insertIL,
  insertILWithKey,
  fromList,
```

```
  elemsIL,
  assocsIL,
  deleteIL,
  mapIL,
  mapILKeys,
  mapILWithKey,
  modifyIL,
  memberIL
) where

import Data.Map.Strict hiding (fromList)
import qualified Data.Map.Strict as Map
import Control.DeepSeq
import Control.Seq

------------------------------------------------------------------------
-- Container for objects.
------------------------------------------------------------------------

-- | Identity List id.
type ILKey = Integer

-- | Identity List, inspired by Yampa Arcade paper, contains objects that can be identifiable.
data IL a = IL {
    ilNext :: ILKey,
    ilAssocs :: Map ILKey a
  }

-- | Creates an empty IL.
emptyIL :: IL a
emptyIL = IL 0 Map.empty

-- | Looks up an element by key.
lookupIL :: ILKey -> IL a -> Maybe a
lookupIL key = Map.lookup key . ilAssocs

-- | Inserts an object into the IL.
insertIL :: a -> IL a -> IL a
insertIL a il = IL (ilNext il + 1) (Map.insert (ilNext il) a (ilAssocs il))

-- | Inserts an object into the IL with a specific key.
insertILWithKey :: a -> ILKey -> IL a -> IL a
insertILWithKey a k il@IL{ilAssocs=m} = il{ilAssocs=Map.insert k a m}

-- | Turns a list into an IL.
fromList :: [a] -> IL a
fromList l = IL (fromIntegral $ length l) (Map.fromList $ zip [0..] l)

-- | Returns elements in IL.
elemsIL :: IL a -> [a]
elemsIL = Map.elems . ilAssocs

-- | Returns all associations of keys and objects.
assocsIL :: IL a -> [(ILKey,a)]
assocsIL = Map.assocs . ilAssocs

-- | Deletes objects with given id.
deleteIL :: ILKey -> IL a -> IL a
deleteIL k il = il{ilAssocs=Map.delete k $ ilAssocs il}
```

```
-- | Maps a function to all elements in IL.
mapIL :: (a->b) -> IL a -> IL b
mapIL f il = il{ilAssocs=Map.map f $ ilAssocs il}

-- | Maps a function to all elements in IL.
mapILKeys :: (ILKey->b) -> IL a -> IL b
mapILKeys f il = il{ilAssocs=Map.mapWithKey (\k _ -> f k) $ ilAssocs il}

-- | Maps a function to all elements in IL.
mapILWithKey :: (ILKey -> a -> b) -> IL a -> IL b
mapILWithKey f il = il{ilAssocs=Map.mapWithKey f $ ilAssocs il}

-- | Modifies a single element with given function if it exist.
modifyIL :: ILKey -> (a -> a) -> IL a -> IL a
modifyIL key f il@IL{ilAssocs=m} = maybe il (\a -> il{ilAssocs=Map.insert key (f a) m} ) $ Map.lookup key m

-- | Returns whether an element is part of an IL.
memberIL :: ILKey -> IL a -> Bool
memberIL key IL{ilAssocs=m} = Map.member key m

instance Functor IL where
  fmap = mapIL

instance Foldable IL where
  foldMap f il = foldMap f (ilAssocs il)
  foldr f b il = Map.foldr f b (ilAssocs il)

instance Traversable IL where
  traverse f il = IL (ilNext il) <$> traverse f (ilAssocs il)

instance (NFData a) => NFData (IL a) where
  rnf (IL n assoc) = rnf n `seq` rnf assoc
```

Listing B.20 EasyGL.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : Val.Strict.Scene
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Val implementation of a strict game scene.
--
-----------------------------------------------------------------------------

{-# LANGUAGE Arrows #-}
module Val.Strict.Scene (
initScene,
initScenePar
)
where

import Val.Strict.Data
import qualified Val.Strict.IL as IL
```

```
import Val.Strict.UI hiding (FreeMouse)
import Val.Strict.Scene.Resources

import Control.Concurrent
import Control.Concurrent.Async
import Control.DeepSeq
import Control.Exception
import Control.Monad
import Control.Monad.IO.Class (MonadIO, liftIO)
import Control.Parallel.Strategies
import Data.IORef (IORef, newIORef, readIORef,
                                                writeIORef)
import Data.Map.Strict hiding (foldl, map, foldr, foldl', union, unions, null)
import qualified Data.Map.Strict as Map
import Data.Foldable (foldl')
import Data.Maybe
import DeltaClock
import EasyGL
import EasyGLUT
import FRP.Yampa
import qualified Graphics.Rendering.OpenGL as GL
import qualified Graphics.UI.GLUT as GLUT
import System.Exit
import System.Mem

-- | Reads outputs values of objects to generate a killOrSpawn event.
killOrSpawn :: (a,IL (ObjOutput s et)) -> Event (IL (Object s et) -> IL (Object s et))
killOrSpawn (_,oo) = foldl (mergeBy (.)) noEvent events
  where
    events = map toEvent $ assocsIL oo
    toEvent (k,v) = mergeBy (.) kill spawns
      where
        spawns = fmap (foldl (.) id . map insertIL) $ ooSpawnReq v
        kill = tag (ooKillReq v) (deleteIL k)

feedbackSF :: IsCamera c => SF (GameInput,IL s) c
  -> SF (GameInput,IL (ObjOutput s et),IL (Event et)) (IL (ObjOutput s et))
  -> SF (GameInput,IL (Event et)) (IL (ObjOutput s et),c)
feedbackSF camera sf = proc (gi,c) -> do
  rec
    b <- sf -< (gi,b,c)
  cam <- camera -< (gi,mapIL ooObjState b)
  returnA -< (b,cam)

unions :: (MergeableEvent a) => [Event a] -> Event a
unions = foldl' (mergeBy union) noEvent

------------------------------------------------------------------------------
-- Game single scene.
------------------------------------------------------------------------------

-- | Generates the input for each object based on world input and provided event generators.
route :: MergeableEvent et => [ IL s -> IL (Event et) ] -- event generators
  -> (GameInput, IL (ObjOutput s et), IL (Event et)) -- input from world and oo from last iteration
  -> IL sf -- objects to route
  -> IL (ObjInput s et,sf)
route eventGenerators (gi,ooil,worldEvents) objs = mapILWithKey aux objs
  where
    defaultObjInput = ObjInput noEvent gi states
```

```
    states = mapIL ooObjState ooil
    events = map (\f -> f states) eventGenerators
    aux key o = (defaultObjInput{oiEvents= unions e },o)
      where
        worldEvent = lookupIL key worldEvents
        thisEvent = map (lookupIL key) events
        e = catMaybes $ worldEvent:thisEvent

-- | Generates the input for each object based on world input and provided event generators.
routePar :: MergeableEvent et => [ IL s -> IL (Event et) ] -- event generators
  -> (GameInput, IL (ObjOutput s et), IL (Event et)) -- input from world and oo from last iteration
  -> IL sf -- objects to route
  -> IL (ObjInput s et,sf)
routePar eventGenerators (gi,ooil,worldEvents) objs =
  withStrategy (parTraversable (evalTuple2 rpar r0)) $ mapILWithKey aux objs
  where
    defaultObjInput = ObjInput noEvent gi states
    states = mapIL ooObjState ooil
    events = map (\f -> f states) eventGenerators
    aux key o = (defaultObjInput{oiEvents= unions e },o)
      where
        worldEvent = lookupIL key worldEvents
        thisEvent = map (lookupIL key) events
        e = catMaybes $ worldEvent:thisEvent

type SceneSF s et = [ IL s -> IL (Event et) ]
  -> IL (Object s et)
  -> SF (GameInput, IL (ObjOutput s et), IL (Event et)) (IL (ObjOutput s et))

-- | A val scene.
sceneSF :: MergeableEvent et => SceneSF s et
sceneSF eventGenerators objs = dpSwitch
  (route eventGenerators)
  objs
  (arr killOrSpawn >>> notYet)
  (\objects f -> sceneSF eventGenerators (f objects))

-- | A val scene, allows concurrency.
sceneSFPar :: MergeableEvent et => SceneSF s et
sceneSFPar eventGenerators objs = dpSwitch
  (routePar eventGenerators)
  objs
  (arr killOrSpawn >>> notYet)
  (\objects f -> sceneSFPar eventGenerators (f objects))

-- | Inicialices a scene.
initScene :: (IsCamera c,MergeableEvent et) => SF (GameInput,IL s) c
  -> IO ResourceMap
  -> [ IL s -> IL (Event et) ]
  -> [Object s et]
  -> IO ()
initScene = initSceneAux sceneSF

-- | Inicialices a scene, allows concurrency.
initScenePar :: (IsCamera c,MergeableEvent et) => SF (GameInput,IL s) c
  -> IO ResourceMap
  -> [ IL s -> IL (Event et) ]
  -> [Object s et]
  -> IO ()
```

```
initScenePar = initSceneAux sceneSFPar

initSceneAux :: (IsCamera c,MergeableEvent et) => SceneSF s et
  -> SF (GameInput,IL s) c
  -> IO ResourceMap
  -> [ IL s -> IL (Event et) ]
  -> [Object s et]
  -> IO ()
initSceneAux sf camSF resources eventGenerators objsList = do
  let objs = IL.fromList objsList

  glfeedMVar <- newEmptyMVar
  glinputMVar <- newMVar $ GameInput Map.empty (FreeMouse 0 0) 0 (UIRead (800,600) (0,0))
  ioreqfeedMVar <- newEmptyMVar
  ioreqResponseMVar <- newEmptyMVar

  waitLoad <- newEmptyMVar -- Force Yampa to wait until resources have been loaded.

  _ <- forkIO $ ioReqThread ioreqfeedMVar ioreqResponseMVar []
  _ <- forkIO $ do
    rh <- reactInit
      (return (emptyGameInput,emptyIL))
      (reacFun [glfeedMVar,ioreqfeedMVar])
      (feedbackSF camSF (sf eventGenerators objs))

    _ <- takeMVar waitLoad
    clock <- initClock >>= newIORef
    forever $ do
      gameInput <- takeMVar glinputMVar
      events <- takeMVar ioreqResponseMVar
      time <- getDelta clock
      react rh (time,Just (gameInput{timeGI=time},events))

  glThread waitLoad glfeedMVar glinputMVar resources
  where
    reacFun l _ bool out = do
      mapConcurrently_ (`putMVar` out) l
      return bool



-------------------------------------------------------------------------------
-- Threads.
-------------------------------------------------------------------------------

-- | Thread that run io request from the objects output.
ioReqThread :: MergeableEvent et => MVar (IL (ObjOutput s et),c)
  -> MVar (IL (Event et))
  -> [(ILKey,IOExec et)]
  -> IO ()
ioReqThread inMvar outMvar l = do
  (e,l2) <- foldM eventAux (emptyIL,[]) l
  putMVar outMvar e
  (out,_) <- takeMVar inMvar
  l3 <- foldM newAsynAux l2 $ assocsIL out
  ioReqThread inMvar outMvar l3
  where
    newAsynAux l (key,a) = do
      mapM_ async $ ooWorldSpawn a
      let ioReqs = ooWorldReq a
```

```
        execs <- mapM toExec ioReqs
        let ll = zip (repeat key) execs
        return $ ll++l

-- | process a ioreq.
eventAux :: MergeableEvent et => (IL (Event et),[(ILKey,IOExec et)])
  -> (ILKey,IOExec et)
  -> IO (IL (Event et),[(ILKey,IOExec et)])
eventAux (il,l) (key,ioreq) =
  if ioExecBloq ioreq then do
    out <- waitCatch (ioExecIO ioreq)
    return $ either (\e -> (myinsert il key (ioExecError ioreq e),l) )
      (\et -> (myinsert il key et,l) )
      out
  else do
    mio <- poll (ioExecIO ioreq)
    case mio of
      Nothing -> return (il,(key,ioreq):l)
      Just e -> return $ either (\e -> (myinsert il key (ioExecError ioreq e),l) )
        (\et -> (myinsert il key et,l) )
        e
  where
    myinsert il k e = if memberIL k il then modifyIL k (mergeBy union $ Event e) il
      else insertILWithKey (Event e) k il

-- | A thread that render the output of each tick.
glThread :: (IsCamera c) => MVar Bool
  -> MVar (IL (ObjOutput s et),c)
  -> MVar GameInput
  -> IO ResourceMap
  -> IO ()
glThread waitLoad mvar gimvar resources = do
  initOpenGLEnvironment 800 600 ""
  rm <- resources
  performGC
  putMVar waitLoad True
  initGL $ aux rm
  where
    aux rm = do
      keys <- getKeysInfo
      mouse <- getMouseInfo
      ui <- getData
      (out,cam) <- liftIO $ takeMVar mvar
      liftIO $ putMVar gimvar $ GameInput keys mouse undefined ui
      mapM_ (mapM_ execUIActions . ooUIReq) out
      let assocs = assocsIL out
          renderComponents = foldl' (\l (_,oo) -> maybe l (:l) $ ooRenderer oo) [] assocs
      useCamera cam
      liftIO $ render rm renderComponents
```

Listing B.21 EasyGL.hs

```
-------------------------------------------------------------------------------
-- |
-- Module : Val.Strict.UI
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
```

```
-- Stability : stable
-- Portability : portable
--
-- Allows game objects to communicate with window manager.
--
-----------------------------------------------------------------------------

module Val.Strict.UI (
  UIRead(..),
  UIActions(..),
  getData,
  execUIActions
) where

import FRP.Yampa
import EasyGL
import EasyGLUT hiding (FreeMouse)

-- | Current state of windows.
data ScreenState = Windowed {
      heigth :: Double,
      width :: Double
  } | FullScreen {
      heigth :: Double,
      width :: Double
  }

data UIRead = UIRead {
      uiScreenSize :: (Int,Int),
      uiScreenPosition :: (Int,Int)
  }

getData :: GLUT UIRead
getData = do
  (sizex,sizey) <- getWindowSize
  (posx,posy) <- getWindowPosition
  return $ UIRead
    (fromIntegral sizex,fromIntegral sizey)
    (fromIntegral posx,fromIntegral posy)

-- | Actions that game objects can perform over window manager.
data UIActions = FixMouseAt Int Int
  | FreeMouse
  | HideCursor
  | ShowCursor
  | SetScreenSize Int Int
  | SetScreenPosition Int Int

execUIActions :: UIActions -> GLUT ()
execUIActions (FixMouseAt x y) = fixMouseAt (fromIntegral x) (fromIntegral y)
execUIActions FreeMouse = freeMouse
execUIActions HideCursor = hideCursor
execUIActions ShowCursor = showCursor
execUIActions (SetScreenSize x y) = setWindowSize (fromIntegral x) (fromIntegral y)
execUIActions (SetScreenPosition x y) = setWindowPosition (fromIntegral x) (fromIntegral y)
```

Listing B.22 EasyGL.hs

```
-----------------------------------------------------------------------------
```

```
-- |
-- Module : Val.Strict.Util
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Utilities.
--
-----------------------------------------------------------------------------

{-# LANGUAGE Arrows #-}

module Val.Strict.Util (
  makeSF,
  makeCamSF,
  deltaTime,
  uiState
)
where

import FRP.Yampa
import Val.Strict.Data
import Val.Strict.UI
import EasyGL

-- | Returns time since last frame from ObjInput.
deltaTime :: ObjInput b c -> Time
deltaTime = timeGI . oiGameInput

-- | Returns screen state from ObjInput.
uiState :: ObjInput b c -> UIRead
uiState = uiGI . oiGameInput

-- | Creates a game object from an initial state and an update function.
makeSF :: a -> (a -> ObjInput b c -> (a,ObjOutput b c)) -> Object b c
makeSF a0 f = proc oi -> do
  rec
    --(a1,oo) <- iPre (f a0 emptyObjInput) <<< arr (uncurry f) -< (a1,oi)
    --(a1,oo) <- iPre (a0,undefined) <<< arr (uncurry f) -< (a1,oi)
    (a2,oo) <- arr (uncurry f) -< (a1,oi)
    a1 <- iPre a0 -< a2
  returnA -< oo

-- | Creates a camera from an initial state and an update function.
makeCamSF :: IsCamera c => a
  -> (a -> Time -> (GameInput,IL b) -> (a,c))
  -> SF (GameInput,IL b) c
makeCamSF a0 f = proc input -> do
  rec
    t2 <- iPre 0 -< t
    t <- time -< ()
  let delta = t - t2
  rec
    (a1,cam) <- iPre (f a0 0 (emptyGameInput,emptyIL)) <<< arr (\(state,time,oi) -> f state time oi) -< (a1,delta,input)
  returnA -< cam
```

Listing B.23 EasyGL.hs

```
-----------------------------------------------------------------------------
-- |
-- Module : Val.Strict.Scene.Resources
-- Copyright : Copyright (c) 2017, Jose Daniel Duran Toro
-- License : BSD3
--
-- Maintainer : Jose Daniel Duran Toro <jose_daniel_d@hotmail.com>
-- Stability : stable
-- Portability : portable
--
-- Functions to load resources to the game system.
--
-----------------------------------------------------------------------------

module Val.Strict.Scene.Resources (
loadResouces,
render
) where

import EasyGL

import Control.Concurrent.Async
import Data.Map.Strict
import qualified Data.Map.Strict as Map
import qualified Graphics.Rendering.OpenGL as GL
import qualified Data.ByteString.Lazy as BS
import Val.Strict.Data

loadResouce :: (ResourceIdentifier,[IndexedModel],Material) -> IO (ResourceIdentifier,(Entity,Material))
loadResouce (identifier,im,mat) = do
  ent <- indexedModel2Ent im
  return (identifier,(ent,mat))

loadResouce1 :: Resource -> IO (ResourceIdentifier,[IndexedModel],Material)
loadResouce1 (identifier,path,mat) = do
  im <- (toIndexedModel . readObj) <$> BS.readFile path
  return (identifier,im,mat)

-- | Loads resources from files.
loadResouces :: [Resource] -> IO ResourceMap
loadResouces resouces = do
  resouces0 <- mapConcurrently loadResouce1 resouces
  loadedResources <- mapM loadResouce resouces0
  return $ Map.fromList loadedResources

-- | Renders resources with transform.
render :: ResourceMap -> [(ResourceIdentifier,Transform,Uniform ())] -> IO ()
render resouces = mapM_ aux
  where
    aux (identifier,transform,uni) = maybe
      (putStrLn $ "No resouce: "++identifier)
      (\(ent,mat) -> GL.preservingMatrix (useTransform transform >> drawWithMat mat ent uni))
      (Map.lookup identifier resouces)
```

Listing B.24 Ejemplo - Mostrando armadillo

```
import EasyGL
import EasyGLUT
```

```
import System.Exit
import System.IO (stderr)
import Control.Monad.IO.Class (MonadIO,liftIO)
import qualified Graphics.Rendering.OpenGL as GL

armadillo :: Shader -> IO (Material,Entity)
armadillo myShader = do
  m <- makeMaterial myShader []
  case m of
    Left s -> putStrLn s >> exitFailure
    Right mat -> do
      e <- readObj2Ent "./armadillo.obj"
      return (mat,e)

main = do
        -- se inicializa el contexto de OpenGL y GLUT.
  initOpenGLEnvironment 800 600 "test"
        -- se carga los shaders.
  myShader <-
            loadShadersFromFile
                ["./vertex.shader","./frag.shader"]
                [VertexShader,FragmentShader]
                (Just stderr)
        -- se carga el mayado.
  assets <- armadillo myShader
  initGL $ myfun assets

-- funcion para el ciclo principal de la aplicacion.
myfun :: (Material,Entity) -> GLUT ()
myfun (mat,ent) = do
        liftIO $ GL.preservingMatrix $ do
            useCamera cam
            drawWithMat mat ent $
                set "color" $ GL.Color4 1 1 0 (1 :: GL.GLfloat)
        where
            (Right cam) =
                createCamera3D 0.0 0.0 10.0 0 0 0 30 (800/600) 0.3 200
```

Listing B.25 Ejemplo - Objetos que chocan

```
{-# LANGUAGE Arrows #-}

import Val.Strict hiding (yaw)
import EasyGL
import EasyGLUT
import Data.Either
import System.IO (stderr)
import FRP.Yampa hiding (RandomGen,randomR)
import Data.List (find)
import System.Random
import System.Environment
import qualified Graphics.Rendering.OpenGL as GL
import qualified Data.Map as Map
import qualified System.Random.TF as TF

-- Se cargan los recursos a utilizar en el ejemplo
load :: IO ResourceMap
load = do
  myShader <- loadShadersFromFile
```

```
      ["./assets/3Dshaders/vertex.shader",
      "./assets/3Dshaders/ColorShader.shader"]
      [VertexShader,FragmentShader]
      (Just stderr)
    (Right mat) <- makeMaterial myShader []
    let plane = ("plane","./assets/plane.obj",mat)

    myShader <- loadShadersFromFile
      ["./assets/3Dshaders/vertex.shader",
      "./assets/3Dshaders/NormalShader.shader"]
      [VertexShader,FragmentShader]
      (Just stderr)
    (Right mat) <- makeMaterial myShader []
    let sphere = ("sphere","./assets/cube.obj",mat)

    loadResouces [plane,sphere]

-- Informacion de los objetos.
data GameState = Null
  | Sphere {
      x :: Double,
      y :: Double,
      z :: Double,
      size :: Double
    }

collition :: GameState -> GameState -> Bool
collition Sphere{x=x1,z=y1} Sphere{x=x2,z=y2} =
  ( (x1-x2)^2 + (y1-y2)^2 ) < 4
collition _ _ = False

data EventTypes = Collition GameState
  | NoCollition

instance MergeableEvent EventTypes where
  union NoCollition NoCollition = NoCollition
  union a NoCollition = a
  union NoCollition a = a
  union a b = a

-- Funcion que detecta colisiones.
collitionGen :: IL GameState -> IL (Event EventTypes)
collitionGen inObjs = mapILWithKey aux inObjs
  where
    assocs = assocsIL inObjs
    aux key obj = case valid of
      (x:_) -> Event $ Collition . snd $ x
      [] -> noEvent
      where
        valid = filter
          (\(key2,obj2) -> key /= key2 && collition obj obj2 )
          assocs

cam :: SF (GameInput,IL GameState) Camera3D
cam = proc _ -> do
    returnA -< c
    where
    (Right c) = createCamera3D 0 150 0 0 (-90) 0 30 (800/600) 0.3 200
```

```
plane :: Object GameState EventTypes
plane = proc _ −> do
  let ret = newObjOutput Null
      trans = Transform
        (GL.Vector3 0 (−1) 0)
        (Quaternion 0 (GL.Vector3 0 1 0))
        2 1 2
      uni = do
        set "color" $ GL.Color4 1 1 1 (1 :: GL.GLfloat)
  returnA −< ret{ooRenderer=Just("plane",trans,uni)}

moveSF :: GL.GLdouble
  −> GL.GLdouble
  −> (GL.GLdouble,GL.GLdouble)
  −> SF () (GL.GLdouble,GL.GLdouble)
moveSF limMax limMin d@(dir,_) =
  if (dir > 0) then aux2 (> limMax) (< limMin) d else aux2 (< limMin) (> limMax) d
  where
    aux2 f1 f2 (dir,initx) = switch (aux f1 dir initx) (aux2 f2 f1)
    aux f dir initx = proc _ −> do
      xnew <− (+initx) ^<< integral −< dir
      e <− edge −< f xnew
      returnA −< ((xnew,dir),tag e (−dir,xnew))

moveXZSF :: GL.GLdouble −> GL.GLdouble −> (GL.GLdouble,GL.GLdouble) −>
  GL.GLdouble −> GL.GLdouble −> (GL.GLdouble,GL.GLdouble) −>
  SF a (GL.GLdouble,GL.GLdouble,GL.GLdouble,GL.GLdouble)
moveXZSF limMaxX limMinX initX limMaxZ limMinZ initZ = proc _ −> do
  (x,dirx) <− moveSF limMaxX limMinX initX −< ()
  (z,dirz) <− moveSF limMaxZ limMinZ initZ −< ()
  returnA −< (x,z,dirx,dirz)

type Info = (GL.GLdouble,GL.GLdouble,GL.GLdouble,GL.GLdouble)

getCollition :: SF (ObjInput GameState EventTypes,Info) (Event Info)
getCollition = proc (oi,salida) −> do
  let myEvent = event noEvent toEvent $ inputEvent oi
  returnA −< tag myEvent salida
  where
    toEvent NoCollition = noEvent
    toEvent a = Event a

sphere :: GL.GLdouble −> GL.GLdouble
  −> GL.GLdouble −> GL.GLdouble
  −> Object GameState EventTypes
sphere initx initz velx velz = proc gi −> do
  rec
    rot <− impulseIntegral −< (180,tag e (−360))
    e <− iPre noEvent <<< edge −< rot > 360
    (x,z,dirx,dirz) <− moveArr (initx,initz,velx,velz) −< gi

  let ret = newObjOutput $ Sphere (realToFrac x) 0 (realToFrac z) 1
      trans = Transform
        (GL.Vector3 x 0 z)
        (Quaternion rot (GL.Vector3 0 1 0))
        1 1 1
  returnA −< ret{ooRenderer=Just("sphere",trans,return ())}
  where
    moveArr (x,z,velx,velz) = dkSwitch
```

```
        (moveXZSF 40 (−40) (velx,x) 40 (−40) (velz,z))
        (getCollition >>> notYet)
        (\sf (x,z,velx,velz) −> moveArr (x,z,−velx,−velz) )

randomSphere :: RandomGen g => g −> (Object GameState EventTypes, g)
randomSphere g = (sphere initx initz velx velz,g4)
  where
    (initx,g1) = randomR (−39,39) g
    (initz,g2) = randomR (−39,39) g1
    (velx,g3) = randomR (−4,4) g2
    (velz,g4) = randomR (−4,4) g3

randomSphereList :: RandomGen g => g −> [Object GameState EventTypes]
randomSphereList g = obj:(randomSphereList g1)
          where
                (obj,g1) = randomSphere g

main :: IO ()
main = do
  num <− fmap (read . head) getArgs
  gen <− TF.mkSeedTime >>= return . TF.seedTFGen
  let il = [plane] ++ (take num $ randomSphereList gen)
  initScenePar cam load [collitionGen] il
```