UNR - FCEIA

Análisis de Lenguajes de Programación

Un eDSL para animación esqueletal en 3D

Lautaro Garavano

1. Introducción

Describiremos un lenguaje de programación embebido en Haskell diseñado para crear animaciones esqueletales de objetos 3D.

La animación esqueletal es el método más utilizado hoy en día para describir animaciones 3D en la industria del cine y de los videojuegos. A cada modelo 3D se le asocia una jerarquía de huesos (joints) con los cuales se controla el movimiento de los distintos vértices que lo componen. Una animación es luego una serie de traslaciones, rotaciones, y escalados de esos huesos, los cuales actuarán sobre sus vértices asociados. Los vértices se asocian a uno o más huesos manualmente, en un proceso llamado *skinning*, en el cual se especifica para cada vértice qué huesos afectan su posición, rotación y escala respectivamente.

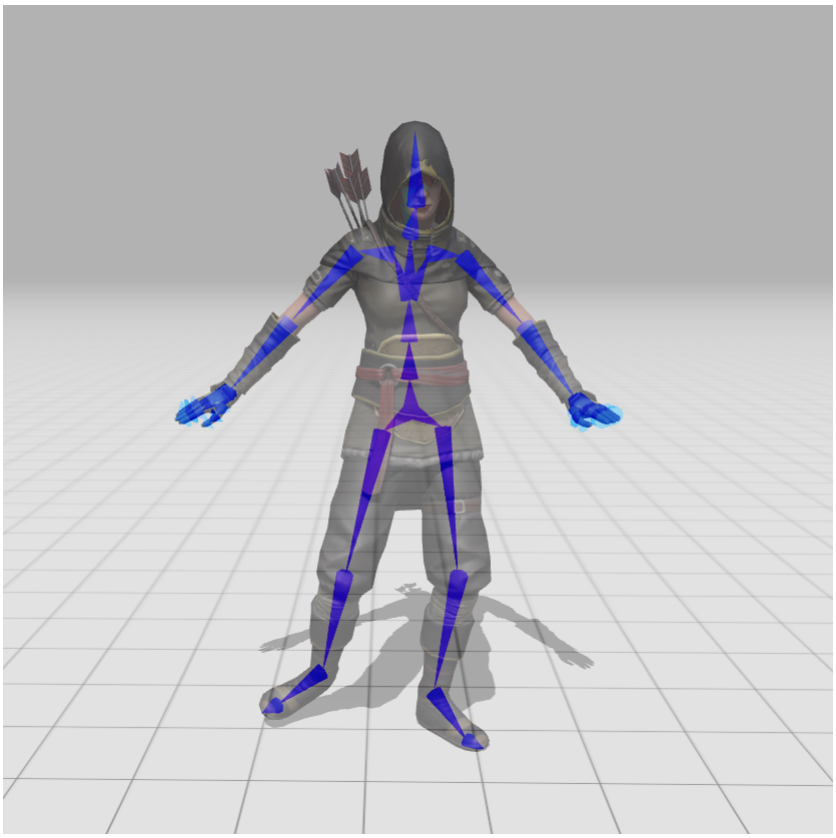

Figura 1: Ejemplo de un modelo "riggeado" (con un esqueleto asociado)

Tradicionalmente, estas animaciones se almacenan en archivos como una lista de fotogramas clave (en inglés *keyframes*), donde para cada uno se guarda las posiciones, rotaciones, y escala de cada uno de los huesos, en cada momento del tiempo. Esta representación de las animaciones es esencialmente sólo una lista de poses para el modelo en cuestión. Estas poses luego se reproducen en secuencia a alta velocidad (30 o 60 fotogramas por segundo) para dar una ilusión de movimiento.

Se puede formular una analogía con la forma en la que las poses discretas se secuencian a alta velocidad para dar la ilusión de movimiento con la forma en la que las imágenes que solemos ver

por computadora están compuestas de píxeles que, al ser lo suficientemente pequeños, le dan una buena fidelidad a la imagen. Sin embargo, en el caso de las imágenes existe otra representación que también es bastante utilizada y cuenta con sus propias ventajas y desventajas: las imágenes vectoriales.

Estas imágenes se describen utilizando diversos objetos geométricos representados como ecuaciones matemáticas (curvas, polígonos, etc.). La principal ventaja de este formato es que, si se utilizan funciones continuas para las representaciones, se pueden crear imágenes que no pierden fidelidad al agrandarse o achicarse. En las imágenes tradicionales compuestas de píxeles, es inevitable que si nos acercamos lo suficiente a la imagen veamos los píxeles individuales, al punto de que ya no se distingue el objeto original de la imagen. Por otro lado, la desventaja de este formato es que no es inmediatamente obvio cuál es la mejor forma de crear una imagen arbitraria sólo utilizando las primitivas típicas de las imágenes vectoriales. Para este propósito, se han creado numerosos programas que ayudan con la tarea de creación y edición de imágenes vectoriales, tales como Inkscape o Adobe Illustrator.

De manera similar, en este trabajo práctico buscamos crear un lenguaje que nos permita crear fácilmente animaciones continuas en vez de discretizadas. Es decir, buscamos representar las animaciones como una función que dado un esqueleto, y el tiempo actual, nos dé como resultado la pose del esqueleto en ese momento. Consideramos que un DSL embebido en Haskell era un buen medio para lograr esto, ya que Haskell ofrece una gran variedad de construcciones que nos permitirán facilitar la composición de animaciones, al punto tal que el código resultante para describir una animación es perfectamente legible en lenguaje natural. Como un pequeño ejemplo, una animación que consiste en un personaje levantando su brazo izquierdo en el plazo de un segundo se describe con la siguiente función

```
levantarBrazoIzquierdo = (rotate upper_Arm_L `around` xAxis) (to 90) `for` 1
```

2. Descripción del eDSL

Describiremos ahora el modelo conceptual del eDSL para poder entender cómo se utiliza. Nuestras animaciones se compondrán de distintos 'clips' de animación que representan algún movimiento, y que escribimos como funciones. En general construiremos estas funciones componiendo las primitivas más básicas `translate`, `rotate` y `scale`. En su construcción tendremos acceso a un parámetro `?t` cuyo valor será distinto en cada momento de la animación. Esta función junto con un número especificando su duración representan una animación. Estas animaciones podrán componerse mediante dos combinadores principales, `==>` y `|||`. Estos operadores refieren a la composición secuencial y en paralelo respectivamente. Otra cosa que se puede hacer con estos clips es alterar el paso del tiempo en cada uno de ellos, ventaja que obtenemos gracias al modelo de tiempo continuo que utilizamos. Con esto, podemos acelerar, ralentizar, reproducir de manera no uniforme y hasta rebobinar nuestras animaciones a voluntad.

Lo primero que hay que hacer es definir sobre qué modelo va a actuar nuestra animación. Esto lo hacemos con la función `forModel`, aclarando la ruta hacia el archivo que contiene nuestro modelo riggeado.

```
forModel "model/chabon.dae"
```

Esta función nos creará una representación de nuestro modelo `model`, y funciones que utilizaremos para referirnos a los distintos huesos que componen el esqueleto de nuestro modelo.

Estas funciones tomarán el nombre que le hayamos dado a nuestros huesos cuando los definimos en el programa de modelado 3D, un ejemplo de ellas es la función `upper_Arm_L` que vimos anteriormente.

Una vez tenemos nuestro modelo, solo necesitamos definir la animación. Esto se hace creando una función de tipo `Animation`. Como dijimos antes, una `Animation` no es más que una función que toma un esqueleto, y devuelve su nuevo estado en un determinado momento del tiempo. Sin embargo, nuestro tipo `Animation` no es simplemente una función de dos argumentos, sino que utiliza una extensión del lenguaje llamada `ImplicitParams`. Esta extensión nos permite que las funciones tengan argumentos extra que no se explicitan en su definición, pero a los que podemos referirnos igualmente. Esto quiere decir que para crear una función de tipo `Animation`, solamente debemos crear una función de tipo `Skeleton` → `Skeleton`. Ahora, para que esta animación no sea simplemente una función constante que devuelva siempre la misma pose, tendríamos que utilizar en su definición el parámetro implícito `?t`, que representa el tiempo. Para cada 'pedazo' de animación que definimos (cada función de tipo `Skeleton` → `Skeleton`), este parámetro tomará valores de 0 a 1. Entonces, si queremos, por ejemplo, agrandar la cabeza de nuestro personaje hasta el doble de su tamaño, podríamos definir la siguiente función.

```
agrandarCabeza :: Animation
agrandarCabeza = scale head (1 + ?t) `for` 1
```

Nótese en el ejemplo el uso de `` `for` ``. En realidad, las funciones de tipo `Skeleton` → `Skeleton` con un `?t` implícito tampoco son exactamente el tipo de `Animation`. Para construir una `Animation` debemos también especificar la duración de esa animación, lo cual se hace con la función `` `for` ``.

Puede ser dificil de entender cómo la expresión `scale head (1 + ?t)` tiene el tipo de función que queremos. Resulta ser que `head`, y el resto de funciones generadas por `forModel` para referirse a los huesos, son lo que se conoce como 'lentes', de la libreria de Haskell `Control.Lens`. Podemos pensar en estos lentes como funciones que pueden actuar como getters y setters para un campo de un tipo determinado. En este caso, las funciones nos permiten acceder y modificar los huesos individuales de nuestro `Skeleton`. La función `scale`, entonces, es una función que toma una de estas lentes, y el número por el cual se quiere multiplicar el tamaño de ese determinado hueso, y devuelve una función. Esta función aceptará un esqueleto, y le aplicará el escalado deseado al hueso especificado.

De manera similar a `scale` se definen el resto de operaciones 'primitivas' del lenguaje. En total tenemos 3 de estas operaciones, que corresponden a trasladar, rotar y escalar un hueso.

```
translate :: BodyPart -> V3 Number -> RunAnim (Skeleton -> Skeleton)
rotate :: BodyPart -> Quaternion Number -> RunAnim (Skeleton -> Skeleton)
scale :: BodyPart -> Number -> RunAnim (Skeleton -> Skeleton)
```

Como se puede ver, `translate` y `rotate` toman vectores y cuaterniones respectivamente para especificar sus transformaciones. Estos tipos provienen de la librería `Linear`, que se utiliza en este lenguaje para introducir numerosos conceptos del álgebra lineal (utilizados intensivamente en el desarrollo de gráficos computarizados). Puede ser complicado especificar una rotación deseada en cuaterniones, por lo que `rotate` viene acompañado de otra función, `around`.

```
around :: (Quaternion Number -> a) -> V3 Number -> Number -> a
```

Como se puede intuir viendo su tipo, `around` convierte cualquier función que tome un cuaternión a una que toma un vector (para representar un eje de rotación) y un ángulo (para representar

la cantidad de rotación deseada, en grados). Una ventaja que obtenemos por usar Haskell como lenguaje anfitrión, es que podemos utilizar esta función de manera infija rodeándola con ``. Si nos remontamos al primer ejemplo mostrado del lenguaje, podemos ver el uso de esta notación infija, logrando una gran legibilidad del código resultante.

```
levantarBrazoIzquierdo = (rotate upper_Arm_L `around` xAxis) (to 90) `for` 1
```

Vemos también en este ejemplo el uso de `xAxis`. Esta es una constante de tipo `V3 Number`, es decir, un vector 3D. De la misma forma, tenemos definidos `yAxis` y `zAxis` para especificar fácilmente estos ejes usuales de rotación.

Nos queda por ver del ejemplo anterior una sola función, `to`. Esta función no es más que `(?t *)`. Su propósito es facilitar la escritura de código más legible, y permitir la construcción de animaciones sin utilizar la extensión de parámetros implícitos de no ser deseada, ya que su uso cuenta con algunas particularidades y puede ser confuso.

En general, cuando construimos estos clips de animación, deseamos realizar más de una acción sobre los huesos al mismo tiempo. Ya que estamos trabajando con meras funciones, podemos componer los resultados de realizar varias operaciones en un mismo clip de animación, utilizando nada más la composición de funciones provista por Haskell. Por ejemplo, para levantar ambos brazos:

```
levantarBrazos = (((rotate upper_Arm_L `around` xAxis) (to 90))
                . ((rotate upper_Arm_R `around` xAxis) (to 90))) `for` 1
```

El eDSL también provee una función llamada `andAlso`, que no es más que el operador de composición `(.)`, para permitir mejor legibilidad.

Veremos ahora los combinadores de animaciones, que nos permiten crear animaciones más complejas a partir de otras ya definidas. Contamos con 2 combinadores, uno para secuenciar clips y otro para ejecutarlos en paralelo.

La composición secuencial `x ==> y` reproduce la animación `x`, y al terminar su duración reproduce la animación `y`. Algo importante a saber es que `y` efectuará sus traslaciones, rotaciones y escalados a partir del estado final del esqueleto después de reproducir `x`. De esta manera, la siguiente animación

```
levantarYBajar = (rotate upper_Arm_L `around` xAxis) (to 90) `for` 1
            ==> (rotate upper_Arm_L `around` xAxis) (to (-90)) `for` 1
```

tendrá el efecto de levantar el brazo y luego regresarlo a su posición original.

El otro operador, la composición en paralelo `x ||| y` tendrá el efecto de crear una nueva animación, cuya duración será el máximo de las dos duraciones, y que efectuará las modificaciones al esqueleto correspondientes a ambos clips al mismo tiempo. De esta forma, la función

```
inutil = (rotate upper_Arm_L `around` xAxis) (to 90) `for` 1
     ||| (rotate upper_Arm_L `around` xAxis) (to (-90)) `for` 1
```

no hará absolutamente nada por 1 segundo, a diferencia del ejemplo usando `==>`. En este ejemplo vemos también que podríamos haber usado una simple composición de funciones para lograr el mismo efecto, pero aún así este combinador es útil cuando ya se tienen animaciones definidas.

Además de los combinadores, tenemos otras 2 operaciones que operan sobre animaciones ya definidas: `loop` y `|>>|`. La primera convierte cualquier animación en una animación que al

terminar, vuelve a repetirse. La duración de esta nueva animación pasa a ser infinita. La segunda función nos sirve para aumentar o reducir la velocidad de reproducción de un clip.

Por último, veremos las funciones de tipo `Curve`. Estas funciones nos servirán para describir animaciones más complejas. En todos los ejemplos que vimos, utilizamos `to` para interpolar linealmente hasta la cantidad de rotación deseada para nuestra animación. Resulta ser que esto es todo lo que necesitamos, puesto que podemos utilizar las distintas funciones `Curve` para alterar la forma en la que el parámetro `?t` es pasado a nuestro clip de animación a lo largo de su duración. Por ejemplo, `easeOut` le aplica la siguiente función de transformación al parámetro `?t`.

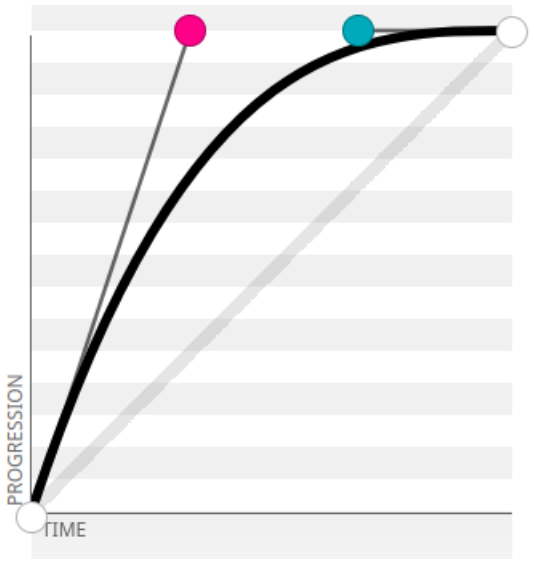

Figura 2: Curva de la función easeOut

Como podemos ver, esto acelera la animación al principio y la ralentiza sobre el final. De manera análoga, definimos `easeIn` y `easeInOut`. Actualmente en el eDSL hay implementadas algunas de las principales curvas de animación frecuentemente utilizadas en el campo de gráficos 3D. Sus definiciones pueden verse en la página web easings.net.

Como último ejemplo combinando todas las funciones vistas, declaremos una animación que ejecute un movimiento de patada con la pierna derecha.

```
kick :: Animation
kick =
  easeInOut
    ( (rotate upper_Leg_R `around` yAxis) (to (-30))
        `andAlso` (rotate lower_Leg_R `around` xAxis) (to 45)
    )
    `for` 1
    ==> easeOutStrong
      ( (rotate upper_Leg_R `around` yAxis) (to 90)
          `andAlso` (rotate lower_Leg_R `around` xAxis) (to (-45))
      )
      `for` 1
```

Esta animación primero retrae la pierna lentamente desde la pose inicial, y luego la extiende rápidamente hacia adelante. Puede ser que el movimiento de la pierna por sí sóla no sea lo suficientemente convincente, por lo que podemos definir animaciones para el resto del cuerpo y luego combinarlas para tener una animación con movimiento más realista.

```
turnArms :: Animation
turnArms =
  (rotate upper_Arm_L `around` yAxis) (to (-30))
    `andAlso` (rotate upper_Arm_R `around` yAxis) (to (-30))
      `for` 1
      ==> easeOutStrong
        ( (rotate upper_Arm_L `around` yAxis) (to 80)
            `andAlso` (rotate upper_Arm_R `around` yAxis) (to 80)
        )
        `for` 1

turnHips :: Animation
turnHips =
  easeInOut ((rotate spine `around` zAxis) (to 30)) `for` 1
    ==> easeOutBack ((rotate spine `around` zAxis) (to (-30))) `for` 1

idlePose :: Animation
idlePose =
  ( (rotate upper_Arm_R `around` xAxis) (-30)
      `andAlso` (rotate lower_Arm_R `around` xAxis) (-30)
      `andAlso` (rotate upper_Arm_L `around` xAxis) (-30)
      `andAlso` (rotate lower_Arm_L `around` xAxis) (-30)
  )
    `for` 0

main = idlePose ==> kick ||| turnHips ||| turnArms
```

Efectivamente esta animación se asemeja más a un movimiento realista. Un truco interesante utilizado en esta animación es la animación `idlePose`. Por defecto, nuestro modelo empieza en una pose por defecto establecida en el programa de modelado 3D (usualmente en 'pose T'). Para empezar desde una pose más natural, podemos simplemente definir una animación de duración 0 que 'inicialice' la pose del modelo. Aquí lo que hacemos es bajar los brazos respecto de la pose T en la que se crea el modelo.

Vemos que es posible crear animaciones más complejas a partir del uso exclusivo de las primitivas del eDSL. Por supuesto, esto es bastante inconveniente. Idealmente una implementación completa del eDSL tendría una 'librería estándar' de animaciones básicas que componer para realizar la mayoría de tareas deseadas.

3. Implementación

La implementación del eDSL hace uso de 3 componentes importantes externos: La librería `Control.Lens`, la extensión `ImplicitParams`, y un muy rudimentario motor de gráficos 3D creado usando la librería `OpenGL`, cuyo código puede explorarse en detalle en la carpeta `Engine` del proyecto. Allí se definen las funciones para leer los modelos 3D de archivos (que para simplificar

el alcance del proyecto solo funciona con el archivo `chabon.dae` provisto, aunque puede llegar a funcionar con otros modelos en formato `.dae` creados en Blender), y un renderer básico el cual podemos ejecutar para visualizar nuestras animaciones.

Cuando escribimos la sentencia `forModel`, en realidad estamos llamando a una función definida con la extensión `TemplateHaskell` (es decir, que funciona en tiempo de compilación), la cual se encarga de crear los lentes para nuestro modelo a partir del archivo. Para esto, lee el archivo y carga su esqueleto como un árbol de `Transforms`, que especifican el estado de cada hueso. Los árboles ya cuentan con lentes para traversar sus distintas ramas. Lo que hace `forModel` es, para cada hueso, componer estas lentes de manera que refieran a un hueso determinado, y declarar una función cuyo nombre es el nombre del hueso especificado en el programa de modelado 3D.

El hecho de que tengamos que leer el modelo 3D del archivo en tiempo de compilación suele ralentizar mucho la compilación, así como el Haskell Language Server, por lo que es conveniente extraer la llamada a `forModel` a otro módulo, para evitar su recompilación lo más posible. En el código provisto, la llamada a `forModel` está en el módulo `Model`, mientras que las animaciones están en el módulo `Anims`.

En el módulo `Types`, podemos ver los tipos principales utilizados en el eDSL, así como algunos utilizados por el motor y la representación de los modelos 3D leídos de archivos. Destacamos aquí el tipo `RunAnim`, que en cierto modo es la columna vertebral del eDSL. Un RunAnim es cualquier cosa que acepte un parámetro implícito `?t` de tipo `Time` (`Time` es simplemente el tipo `Number`, que a su vez es el tipo `GLdouble` definido por la librería `OpenGL`). Nuestras `Animation` entonces, contienen un `RunAnim (Skeleton → Skeleton)`, y su duración, que es también del tipo `Time`.

```
type RunAnim a = (?t :: Time) => a

data Animation = Anim
  { _duration :: Time,
    _runAnim :: RunAnim (Skeleton -> Skeleton)
  }
```

Otro tipo a destacar es el tipo de `Curve`. Este transforma cualquier `RunAnim` a otro `RunAnim`. La naturaleza polimórfica de esta definición hace que lo único que pueda modificar una `Curve` sea el parámetro implícito `?t`.

```
type Curve = forall a. RunAnim a -> RunAnim a
```

Por último, en el módulo `APrelude`, podemos ver la definición de cada una de las primitivas del eDSL. Este es el módulo principal que debe importarse si se quiere trabajar con el eDSL.

4. Utilización del código

El proyecto puede ejecutarse utilizando la herramienta Stack. La animación a visualizar deberá escribirse en el archivo `Anims.hs` con el nombre de `main`. Luego, el comando `stack run` nos abrirá una ventana donde podremos ver nuestra animación. Por defecto, las animaciones no se repiten, así que es conveniente envolver nuestra animación deseada con un `loop`. Una vez dentro del visualizador, podemos movernos con las teclas WASD y mirar con el mouse.

Bibliografía y Software de Terceros

Como indicamos antes, el eDSL utiliza extensivamente las librerias Linear, Control.Lens y OpenGL. Además, se utilizan las librerías Attoparsec y Text.XML para la lectura de los archivos 3D a partir de modelos en formato COLLADA (No había ninguna librería que leyera modelos 3D con esqueleto así que hice el lector a mano). El renderer 3D fue hecho con ayuda de numerosos tutoriales, en particular este que utiliza los bindings de Haskell igual que en este proyecto. El lector de modelos también se hizo con bastante ayuda, la principal referencia fue esta serie de videotutoriales del canal ThinMatrix. Para aprender a usar las distintas extensiones del lenguaje, sirvieron este tutorial de TemplateHaskell, y este tutorial de la libreria Control.Lens. Por último, para el diseño del lenguaje sirvió de inspiración este dsl para animación 2D.