Tetris mit Gloss

Sven Eric Panitz

22. März 2022

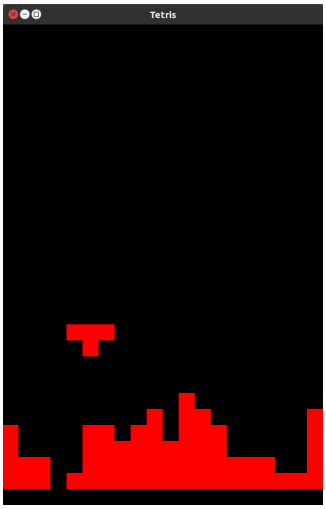
Inhaltsverzeichnis

1 Gloss

In diesem Lehrbrief verwenden wir die Bibliothek Gloss (`gloss.ouroborus.net`), die den Anspruch hat, auf einfache Art und Weise animierte Grafiken mit Tastatureingaben umsetzen zu können. Implementiert ist Gloss mit OpenGL and GLUT.

Es soll der Anfang für ein kleines Tetris-Spiel in diesem Lehrbrief gemacht werden.

Zunächst ein kleiner Screenshot des fertigen Spiels:



Zunächst werden die notwendigen Gloss Module importiert.

Haskell: GlossTetris

```
> module GlossTetris where

> import Graphics.Gloss
> import Graphics.Gloss.Data.ViewPort
> import Graphics.Gloss.Interface.Pure.Game
```

Wir werden zusätzlich Zufallszahlen benötigen, über Linsen ein einfacher Update auf Strukturen machen und ein paar Standardlistenfunktionen.

Somit hier auch die dazu nötigen Importanweisungen.

Haskell: GlossTetris

```
> import System.Random
> import Control.Lens
> import Data.List
```

Die entscheidene Einstiegsfunktion für Gloss-Spiele ist die Funktion `play` aus dem Modul `Graphics.Gloss.Interface.Pure.Game`

Hier sind die typischen Argumente für ein animiertes Spiel zu übergeben. Es gibt einen globalen Tick, in dem das Spiel weiterschaltet und so die einzelnen nacheinander abgespielten Rahmen der Animation erzeut werden.

Kernargument ist ein Spielzustand, der hier mit der Typvariable `world` bezeichnet wird. Dieses ist ein beliebiger Datentyp, der den kompletten Spielzustand zu einem bestimmten Zeitpunkt darstellt.

Für diesen Zustand sind drei Funktionen zu übergeben:

- Eine Funktion, die den Spielzustand visualisiert.
- Eine Funktion, die aufgrund eines Ereignisses zum Beispiel einer Tastatureingabe den Folgezustand berechnet.
- Eine Funktion, die den Folgezustand für den nächsten Tick berechnet.

Zusätzlich wird noch ein Fenster zum Darstellen des Spiels, eine Hintergrundfarbe und die Anzahl der Rahmen pro Sekunde benötigt.

Hier die Dokumentation der Funktion `play`:

Haskell: GlossTetris

```
play::
     Display  Display mode.
  -> Color    Background color.
  -> Int
     Number of simulation steps to take for each second of real time.
  -> world    The initial world.
  -> (world -> Picture)
     A function to convert the world a picture.
  -> (Event -> world -> world)
     A function to handle input events.
  -> (Float -> world -> world)
     A function to step the world one iteration.
     It is passed the period of time (in seconds) needing to be advanced.
  -> IO ()
```

Wie man sieht, finden sich im Prinzip alle Komponenten, die auch aus der Subato Game Library des ersten Semesters in Java bekannt sind.

Wir können ein kleines Beispiel eines hüpfenden Balls machen.

Der Spielzustand sind dabei ein Tripel bestehend aus der y-Position des Balls, der Anfangsgeschwindigkeit und der Ticks, seit dem der Ball gestartet ist. Der initiale Spielzustand ist `(480.0,0.4,0)`.

Dargestellt wird das Spiel in einem Fenster der Höhe von 500 und Breite von 200 Pixel.

Die Hintergrundfarbe ist schwarz und es gibt 33 Rahmen in der Sekunde.

Es wird beim Visualisieren ein roter Kreis auf der aktuellen Höhe gezeichnet. Hier ist zu berücksichtigen, dass der Ursprung des Koordinatensystems in der Mitte des Fensters liegt.

Bei jedem Tastendruck soll wieder der initiale Zustand gesetzt werden.

Am komplexesten ist die Funktion für den Folgezustand. Hier wird die aktuelle Bewegungsgeschwindigkeit berechnet und beim Auftreffen auf dem Boden, springt der Ball mit leicht geringerer Geschwindigkeit wieder nach oben.

Haskell: GlossTetris

```
> ball
>   = play
>       (InWindow "Ball" (200, 500) (0, 0))
>       black
>       60
>       (480.0,0.4,0)
>       (\(h,v0,t)->translate 0 (h-250)$color red$circleSolid 10)
>       (\inp x-> (480.0,0.4,0) )
>       (\_ (h,v0,t) -> let v = v0 + 0.981 * t
>                       in if h<10 then (10,-v*0.8,0) else (h-v,v0,t+1))
```

2 Typklasse für Tetrisspiele

Wir beginnen nicht mit einer Implementierung eines Tetrispieles, sondern mit einer Typklassen, die die Funktionen sammelt, die wir von einem Tetrispiel erwarten.

Wir gehen davon aus, dass ein Tetrisspiel ein Raster von Quadraten ist. Somit hat ein Tetrisspiel eine Anzahl von Spalte und von Zeilen. Eine Zelle in diesem Raster ist mit einer Farbe belegt, die über eine Funktion `get` zu erfragen ist. Die noch fallende Figur wird in einer eigenen Liste erfragt. Die Liste beschreibt die Koordinaten, die die Figur hat.

Eine Funktion zum Erzeugen eines initialen Spiels darf nicht fehlen. Diese erzeugt das Spiel in einer IO-Monade, damit Zufallszahlen oder andere externe Abfragen möglich sind.

Somit erhalten wir die folgende Typklasse:

Haskell: GlossTetris

```
> class Tetris a where
>   newTetris :: IO a
>   rows    :: a -> Int
>   columns :: a -> Int
>   get     :: a -> Int -> Int -> Color
>   current :: a -> [(Int,Int)]
>   ended   :: a -> Bool
>   move    :: a -> a
>   prInput :: Input -> a -> a
```

In der letzten Funktion wird ein Aufzählungswert, der in Tetris gültigen Anwenderinteralktionen repräsentiert, benötigt. Die Aufzählung besteht aus fünf Werten:

Haskell: GlossTetris

```
> data Input = Left|Right|RotateLeft|RotateRight|Fall deriving Eq
```

3 Tetris mit Gloss rendern

Bevor wir die Typklassen implementieren, soll definiert sein, wie ein Spiel, das die Klasse implementiert, in Gloss gespielt werden kann.

Global definieren wir die Hintergrundfarbe in dem Spielraster.

Haskell: GlossTetris

```
> background :: Color
> background = black
```

3.1 Tastatureingaben

Wir haben die Anwenderinteraktionen auf fünf Werte abstrahiert. Die folgende Funktion bildet die Gloss-Events der Tastatur auf diese ab. Wir verwenden die Pfeiltasten sowie die Leertaste.

Haskell: GlossTetris

```
> inputHandler (EventKey (SpecialKey KeyLeft) Down _ _) game
>   = prInput GlossTetris.Left game
> inputHandler (EventKey (SpecialKey KeyRight) Down _ _) game
>   = prInput GlossTetris.Right game
> inputHandler (EventKey (SpecialKey KeyUp) Down _ _) game
>   = prInput RotateRight game
> inputHandler (EventKey (SpecialKey KeyDown) Down _ _) game
>   = prInput RotateLeft game
> inputHandler (EventKey (SpecialKey KeySpace) Down _ _) game
>   = prInput Fall game
> inputHandler _ game = game
```

3.2 Spielschritt

In Spielschritt beschreibt, dass die fallende Figur im Gitter eine Zeile tiefer gerutscht ist. Dieses wird durch die Funktion `move` der Typklasse ausgedrückt.

Haskell: GlossTetris

```
> update seconds game =  move game
```

3.3 Rendern

In der Funktion `render` wird der aktuelle Spielzustand auf dem Bildschirm visualisiert.

In der Funktion `move` bewegen sich die fallenden Steine immer um einen ganzen Block des Rasters. Das wären in einem Schritt die Pixel der Kantenlänge eines Rasterfeldes. Damit wird aber keine flüssige Bewegung möglich. Deshalb verwenden wir beim Rendern mit Gloss als Zustand nicht allein das Tetrisspiel sondern zusätzlich eine Zahl, die von 20 (also der Höhe eines Rasterquadrats) runter gezählt wird. Die Spielsteine, die noch fallen, sind noch nicht im Raster verankert, und erst auf den Weg zum nächsten Feld. Wie weit sie davon entfernt sind, zeigt die Zahl `t` an.

So zeichnet die Funktion `render` die Quadrate, die bereits fest im Raster gelandet sind, plus

Haskell: GlossTetris

```
> render w (t,game) =
>   pictures
>     ([ square  (x * w - width  `div` 2 + w `div` 2)
>                (-y * w + height `div` 2 + w `div` 2) c
>      | x <- [0 .. columns game-1],y <- [0 .. rows game-1]
>      , let c = get game x y
>     ]++
>     [ square  (x * w -width `div` 2+w `div` 2)
>               (-y * w +height`div` 2+w `div` 2+t) red
>     | (x,y)<-current game])
>  where
>    dXYf = fromIntegral w
>    width = w*columns game
>    height = w*rows game
>    square x y c
>      = translate (fromIntegral x) (fromIntegral y)
>                  (color c (rectangleSolid  dXYf dXYf))
```

3.4 Spielschleife starten

Der Trick, dass der Zustand in der Gloss-Schleife nicht allein das Spiel ist, sondern ein Paar, müssen wir hier berücksichtigen. Dieses Paar hat zunächst einen Zähler, für das Pixelweise fallen der Steine und dann den eigentlichen Spielzustand. Das sieht an insbesondere bei der Funktion zum Update des Spielzustands. Wenn der Zähler noch über 0 ist, wird dieser im Update nur heruntergezählt. Erst wenn dieser 0 erreicht hat, wird die eigentliche Update-Funktion auf den Spielzustand angewendet.

Haskell: GlossTetris

```
> moin :: Tetris a => Int -> a -> IO ()
> moin w game = play
>  (InWindow "Tetris" (w*columns game,w*rows game) (20,20))
>  background
>  60
>  (w,game)
>  (\g->render w g)
>  (\inp (t,g)->(t,inputHandler inp g))
>  (\f (t,g)->if t<=0 then (w,update f g) else (t-3,g))
```

4 Einstein-Tetris

Wir implementieren jetzt für ein sehr reduziertes Tetrisspiel die Typklasse. das Spiel hat nur fallende Figuren, die genau aus einem Quadrat bestehen. Es bietet sich der Name

Einstein-Tetris an.

4.1 Datenkonstruktor

Als Datenstruktur dient der Datentyp `OneTetris` in Record-Notation.

Haskell: GlossTetris

```
> data OneTetris
>   = OneTetris
>      { mcurrent::(Int,Int)
>      , randomGen1::StdGen
>      , grid1:: [[Color]]
>      , mcolumns1::Int
>      , mrows1::Int
>      }
```

Die Felder für Spalten- und Zeilenanzahl erklären sich von selbst. Im Feld `mcurrent` wird der gerade fallende Stein gespeichert. Das `grid1` ist das eigentliche Raster der Quadrate. Es ist zeilenweise angeordnet. Die inneren Listen sind die Zeilen, die von oben nach unten in der Liste gespeichert sind.

4.2 Linsen

Updates in einer Datenstruktur sind in reinen funktionalen Sprachen ein Problem, weil es sie nicht gibt. Statt zum Beispiel in einer Liste an einem bestimmten Index ein Element neu zu setzen, also eine der primitivsten Anweisungen eines Arrays in imperativen Sprachen, kann man nur eine neue Liste erzeugen. Hierzu muss manuell eine Funktion geschrieben werden, die die Liste dekonstruiert und dann wieder mit dem neuen Element verpackt.

Haskell: GlossTetris

```
> a = [12,3,4,5,5,6,7]
>
> updateAt i e [] = []
> updateAt 0 e (x:xs)  = (e:xs)
> updateAt n e (x:xs)  = (x:updateAt (n-1) e xs)
>
> a' = updateAt 3 42 a
```

Dieses kann für eigene komplexe Datenstrukturen recht aufwändig werden. Wenn man ein tief verschachteltes Datenmodell hat, in dem ein Blatt verändert werden soll, ist der komplette Code zum Dekonstruieren und wieder Konstruieren der Datenmodells zu schreiben.

Sogenannte Optiken und insbesondere Linsen versuchen diesen Boilerplate-Code zu verallgemeinern, so dass mehr oder weniger generische Getter- und Setter-Funktionen für tiefe Datenstrukturen generiert werden.

Die komplexe Haskellbibliothek `lens` vereint eine Reihe von Optiken. Für sie gibt es ein umfangreiches Wiki [len]. Einen recht guten Überblick auf Optiken finden sich in den Vorlesungsfolien von Sabel [Sab20].

Für unsere Zwecke wollen wir nur eine minimale Funktionalität aus der Lens-Bibliothek nutzen. Den Update von Listen, also die oben von Hand geschriebene Funktion `updateAt`.

Hierzu kennt die Bibliothek zwei Operatoren. Der Operator `&`, um zu signalisieren, dass ein Upadte auf einer Liste durchzuführen ist, und der Operator `.~`, der hilft zu spezifizieren an welchen Index welches neues Element zu setzen ist.

Im Zusammenspiel ergibt das den folgenden Ausdruck, um an Index 3 der Liste den Wet 42 zu setzen:

Haskell: GlossTetris

```
> b = a & (ix 3) .~ 42
```

Das Raster des Tetrisspiels ist eine Liste von Listen von Farben. Um also einen Farbwert zu ändern, ist die obige Operation doppelt anzuwenden.

Haskell: GlossTetris

```
> setColor x y c grd = grd& (ix y) .~ ((grd!!y) & (ix  x) .~ c)
```

4.3 Tetris-Instanz

Wir können nun die Datenstruktur `OneTetris` zu einer Instanz der Typklasse `Tetris` machen.

4.3.1 Initialisierung

Zunächst eine kleine Hilfsfunktion: wir brauchen oft eine Liste von gleichen Elementen und einer festen Länge. Hierzu können wir die Funktionen `repeat` und `take` kompinieren.

Haskell: GlossTetris

```
> nof n = take n.repeat
```

Die erste Funktion ist dazu da, einen initialen Spielzustand zu erzeugen. Da wir eine Zufallskompenente integrieren wollen, liefert diese Funktion einen in einer Monade verpackten Spielzustand.

Um ein neues Spiel zu erzeugen wird zunächst ein neuer Zufallszahlengenerator erzeugt. Dieser wird gleich einmal angewendet, um die Spalte des ersten Steins zufällig zu generieren.

Die Funktion `uniformR` erzeugt ein Paar, aus einer Zufallszahl in einem bestimmten Bereich und den neuen Zufallsgenerator.

Wir geben ein `Onetetris` Objekt zurück mit 20 Spalten und 30 Zeilen, einem Raster aus 30 Zeilen mit 20 Mal der Hintergrundfarbe.

Der erste fallende Stein startet in Zeile 0 und der generierten Spalte.

Haskell: GlossTetris

```
> instance Tetris OneTetris where
>   newTetris  =  do
>     gen <- getStdGen
>     let (c,gen') = uniformR (0, 19) gen
>     return$OneTetris (c,0) gen' (nof 30$nof 20 background) 20 30
```

4.3.2 Spalten und Zeilen

Die Frage nach Spalten und Zeilen kann direkt über die Selektorfunktionen des Record-Datentyps erfolgen.

Haskell: GlossTetris

```
>   rows    = mrows1
>   columns = mcolumns1
```

4.3.3 Farbfelder

Für einen Zeilen- und Spalten-Wert kann der aktuell gesetzte Farbwert im Raster direkt über Indexzugriff auf der Liste erzeugt werden.

Haskell: GlossTetris

```
>   get g c r= grid1 g!!r!!c
```

Wir haben bisher immer nur ein Kästchen als fallende Figur. Deshalb erhalten wir dafür die einelementige Liste von dessen Koordinaten.

Haskell: GlossTetris

```
>   current g = [mcurrent g]
```

4.3.4 Ende des Spiels

Wir verzichten vorerst darauf das Spielende zu implementieren.

Haskell: GlossTetris

```
>   ended   g = False
```

4.3.5 Einzelschritt des Spiels

Die entscheidene Funktion beschreibt, wie man von einem Spielzustand in den nächsten kommt. Im einfachsten Fall, er versteckt sich in der Implementierung im `otherwise` rutscht der Stein eine Zeile tiefer.

Wenn dort bereits im Raster ein Stein liegt `get g x (y+1) /= background = g'` oder wenn dort keine Zeile mehr kommt `y+1 >= rows g`, dann ist der Stein auf dieser Position fest zu verankern: `setColor x y red (grid1 g)`.

Dann wird geschaut, ob es Zeilen gibt, die komplett gefüllt sind:
`partition (all (\c->c/=background)) grd`

Diese werden entfernt und durch neue Leerzeilen aufgefüllt:
`nof (length full)(nof 20 background)`

Schließlich wird der Zufallsgenerator bedient, um eine Spalte des neuen Steins zu generieren.

Haskell: GlossTetris

```
>   move g
>    |y+1 >= rows g || get g x (y+1) /= background = g'
>    |otherwise = g{mcurrent=(x,y+1)}
>      where
>       (x, y) = mcurrent g
>       (c,gen') = uniformR (0, columns g -1) (randomGen1 g)
>       g' = g{ mcurrent=(c, 0)
>             , randomGen1=gen'
>             , grid1 = removeFull$setColor x y red (grid1 g)
>             }
>       removeFull grd = (nof (length full)(nof 20 background))++notfull
>         where
>           (full, notfull) = partition (all (\c->c/=background)) grd
```

4.3.6 Tastaturverarbeitung

Auch das kleine Einstein-Tetris soll auf die Tastatur reagieren. Der fallende Stein soll mit den Pfeiltasten einen Rasterplatz nach links oder rechts verschoben werden.

Damit ist die x Position (die Spalte) des fallenden Steins neu zu setzen, je nachdem, ob rechts oder links gedrückt wurde.

Das geht natürlich nur, wenn die neue Spalte nicht außerhalb des Rasters liegt und wenn dort noch kein Stein liegt.

Haskell: GlossTetris

```
>   prInput key g
>     |x'<0 || x'>=columns g || get g x' y /= background = g
>     |otherwise = g{mcurrent=(x',y)}
>      where
>       (x,y) = mcurrent g
>       x'
>        |key==GlossTetris.Left = x-1
>        |key==GlossTetris.Right = x+1
>        |otherwise = x
```

4.3.7 Spielstart

Damit ist die Implementierung der Typklasse `Tetris` für das Einstein-Tetris abgeschlossen. Wir können es mit Hilfe von Gloss spielen:

Haskell: GlossTetris

```
> play1 = do
>   tetris <- newTetris::IO OneTetris
>   moin 25 tetris
```

5 Mehrstein-Tetris

Jetzt soll das Tetris mit Ihrer Hilfe zu einem vollen Tetris erweitert werden. Hierzu machen wir eine neue Implementierung der Typklasse `Tetris`, um auch dieses mit Gloss zu spielen.

Haskell: GlossTetris

```
> play2 = do
>   tetris <- newTetris::IO MoreTetris
>   moin 25 tetris
```

5.1 Datenkonstruktor

Der einzige Unterschied im Datentyp zum Einstein-Tetris ist, dass nun nicht nur die Koordinate für einen Stein gespeichert wird, sondern für eine Figur aus mehreren Steinen.

Haskell: GlossTetris

```
> data MoreTetris
>   = MoreTetris
>      { mcurrent2::[(Int,Int)]
>      , randomGen2::StdGen
>      , grid2:: [[Color]]
>      , mcolumns2::Int
>      , mrows2::Int
>      }
```

5.2 Tetris-Instanz

Es ist wieder die Instanz zu implementieren.

Die Basisfunktion sind kaum abzuändern. Lediglich die erste fallende Figur besteht jetzt aus vier Quadraten.

Haskell: GlossTetris

```
> instance Tetris MoreTetris where
>   newTetris  =  do
>     gen <- getStdGen
>     let (c,gen') = uniformR (1, 17) gen
>     return$MoreTetris
>       [(c-1,0),(c,0),(c+1,0),(c+2,0)]
>       gen'
>       (nof 30$nof 20 background) 20 30

>   rows    = mrows2
>   columns = mcolumns2

>   get g c r= grid2 g!!r!!c
>   current  = mcurrent2

>   ended   g = False
```

6 Aufgaben

Aufgabe 1 Schreiben Sie die Funktion `move`, die die ganze Figur eine Zeile tiefer fallen lässt, wenn unter ihr noch frei ist und das Spielfeld noch nicht beendet.

Sehen Sie eine Reihe von Figuren vor, aus denen zufällig gewählt wird, wenn eine neue Figur fallen gelassen wird.

Haskell: GlossTetris

```
> move g = g
```

Aufgabe 2 Implementieren Sie jetzt die Funktion zur Tastatursteuerung. Die fallende Figur soll nach links und rechts bewegt werden können und sie soll mit und gegen den Uhrzeigersinn gedreht werden können.

Haskell: GlossTetris

```
> prInput key g = g
```

Aufgabe 3 Enwickeln Sie weitere Varianten des Tetrispiels.

Aufgabe 4 Nutzen Sie die Gloss Bibliotek, um weitere Klassiker der Computerspiele zu entwickeln oder eine EIgenkreation umzusetzen.

7 Lernzuwachs

- Pseudo-Randomisierung
- Verwendung der Gloss Bibliothek
- Linsen

Literatur

[len] Lens wiki. `https://github.com/ekmett/lens/wiki`. [Online; accessed 22-March-2022].

[Sab20] Sabel, David. Funktionale Optiken. `www.tcs.ifi.lmu.de/lehre/ss-2020/fun/material/folien/folien-09-druckversion`, 2020. [Online; accessed 2-March-2022].