

Haskell Snake

Projekt Haskell Snake predstavlja implementaciju popularne igre Snake u programskom jeziku Haskell. Vektorska grafika je relizirana pomoću biblioteke Gloss, dok je igra implementirana funkcionalnih programskih konstrukcija jezika Haskell (uz nekoliko monada :).

Interakcija između igrača i igre se vrši pomoću tipkovnice (tipke W, A, S, D) za kretanje zmiје. Cilj igre je sakupiti što više jabuka, a igrač gubi kada zmiја udari u zid ili u samu sebe.

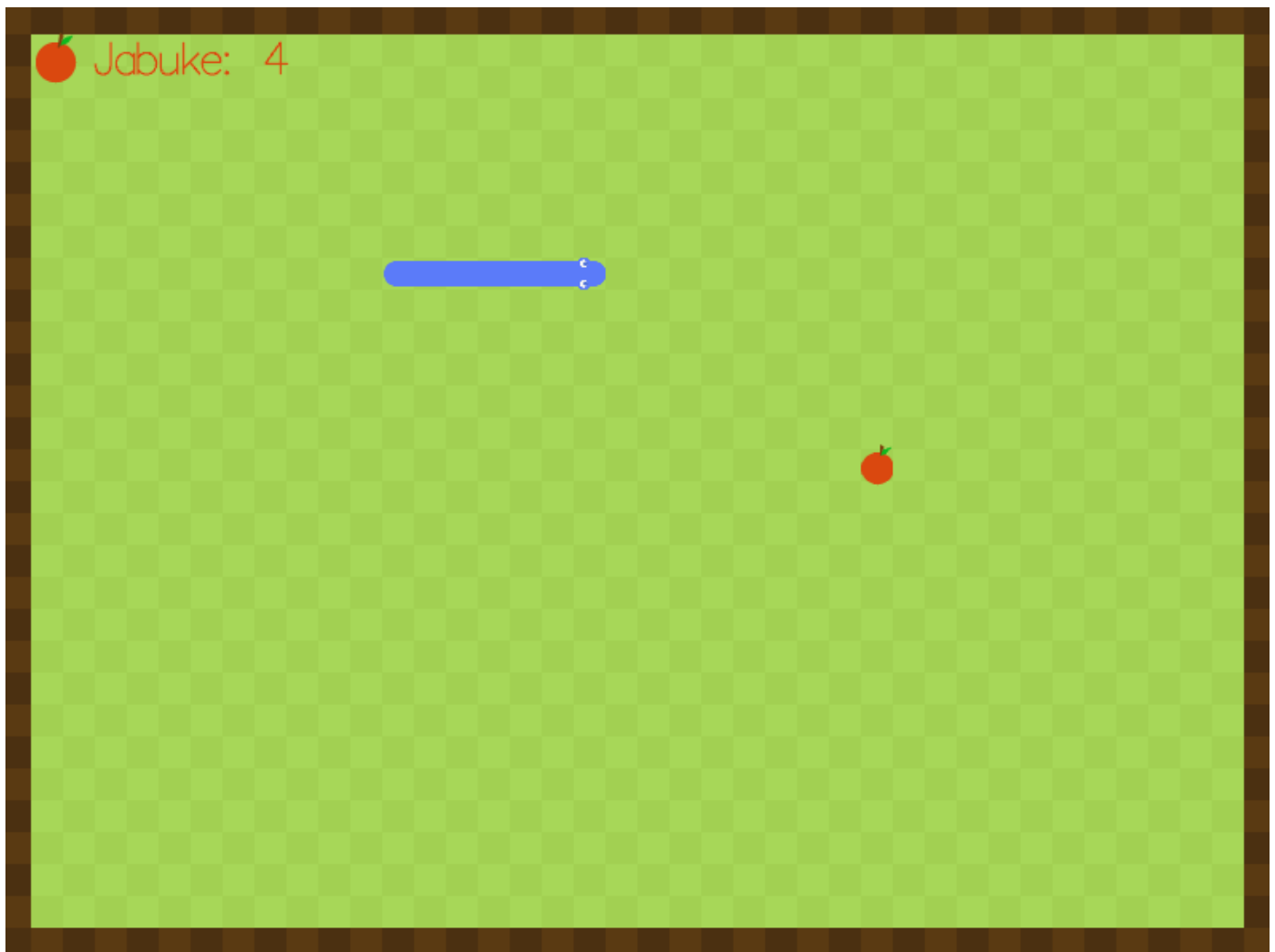
Implementacija je podijeljena u nekoliko modula (njih 8 ukupno) koji upravljaju stanjem igre, iscrtavanjem grafičkih elemenata, obradom korisničkih akcija i logikom igre.

Projekt izradio: Luka Blašković

Ustanova: Sveučilište Jurja Dobrile u Puli, Fakultet informatike

Kolegij: Funkcijsko programiranje, ak. god. 2023/2024, **nositelj:** [doc. dr. sc. Siniša Miličić](#)

[Source kod projekta ovdje](#)



Screenshot igre Haskell Snake

Sadržaj

- [Haskell Snake](#) 🐍
 - [Sadržaj](#)
- [Kako pokrenuti igru](#)
- [Moduli](#) 🛠️
 - [1. Main modul 1/2](#)
 - [main](#) funkcija 🎮
 - [2. GameState modul](#)
 - [GameState](#) tip podatka 🛑
 - [initialState](#) funkcija NEW
 - [moveSnake](#) funkcija 🐍 ⬆️ ⬆️ ⬆️ ⬆️ ⬆️
 - [snakeEatsApple](#) funkcija 🐍 🍏
 - [checkCollision](#) funkcija 🐍 🚗
 - [growSnake](#) funkcija 🐍 🐍 🐍
 - [3. Main modul 2/2 \(nastavak\)](#)
 - [update](#) funkcija ↺ ↺ ↺
 - [resetGame](#) funkcija ⬅️ BACK
 - [handleSnakeMovement](#) i [updateGameStateAfterMovement](#) funkcije za kretanje 🐍 ↺
 - [handleAppleEaten](#) i [handleAppleRespawn](#) funkcije 🍏 🍏 🍏
 - [4. Apple modul](#)
 - [loadAppleSprite](#) funkcija 🍏 🖼️
 - [newApple](#) funkcija NEW 🍏
 - [renderApple](#) funkcija 🎮 🍏
 - [5. Input modul](#)
 - [handleEvent](#) funkcija 🏠
 - [6. AppleCounter modul](#)
 - Definiranje boje teksta i pomoćna [boldText](#) funkcija 🌈
 - [renderAppleCounter](#) funkcija 🎮 🍏 12 34
 - [7. SnakeRender modul](#)
 - [loadSnakeSprites](#) funkcija 🐍 🖼️
 - [directions](#) i njene pomoćne funkcije 🐍 ⬆️ ⬆️ ⬆️ ⬆️ ⬆️
 - [renderSnake](#) funkcija 🐍 🎮
 - [8. Render modul](#)
 - [render](#) funkcija 🎮 🖥️

Kako pokrenuti igru

1. Instalacija Haskell-a, može se preuzeti sa [službene stranice](#), preporuka je instalirati GHC (Glasgow Haskell Compiler)
2. Instalacija `stack` alata za upravljanje Haskell bibliotekama i ukupnim projektom, upute su na sljedećem [linku](#)
3. Jednom kad su alati instalirani, otvorite terminal i u direktoriju projekta `snake` pokrenite sljedeće naredbe:
 - `stack build` - build cijelog projekta
 - `stack ghc -- -o snake main.hs` - kompilacija `Main.hs` modula
 - `stack.exe` - pokretanje igre

Moduli

[1] Main modul 1/2

`main` funkcija

Glavna `main` funkcija koja pokreće igru.

- funkcija se sastoji od **niza sekvencijalnih naredbi** koje rade unutar monade
- IO (Input/Output) monada se koristi za izvršavanje IO operacija

1. Postavljanje prozora za igru

- `Graphics.Gloss | InWindow String (Int, Int) (Int, Int)` - funkcija iz Gloss biblioteke koja postavlja prozor za igru
- `windowWidth` i `windowHeight` parametri predstavljaju širinu i visinu prozora (definirani u `Config` modulu)
- Posljednji argument `(Int, Int)` predstavlja poziciju prozora na ekranu

```
let window = InWindow "Snake Game" (windowWidth, windowHeight) (200, 200)
```

2. Boja pozadine i FPS

```
let backgroundColor = black
    framesPerSecond = 60
```

3. Učitavanje spriteova i inicijalnog stanja igre

- učitavanje spriteova uz pomoć funkcija `loadAppleSprite` i `loadSnakeSprites` iz modula `Apple` i `SnakeRender`

```
appleSprite <- loadAppleSprite
snakeSprites <- loadSnakeSprites
initialGameState <- initialState
```

4. Pokretanje simulacije igre

- `(Graphics.Gloss.Interface.Pure.Game) play` [funkcija iz Gloss biblioteke](#) koja simulira igru.

Funkcija prima 7 parametara:

- **window** (`Display | InWindow`),
- **backgroundColor** (`Color`),
- **framesPerSecond** (`Int`),
- **initialGameState** (`world`),
- **funkcije za iscrtavanje (render appleSprite snakeSprites)** (`world -> Picture`),
- **handleEvent** (`Event -> world -> world`) i
- **update** (`Float -> world -> world`)

```
play window backgroundColor framesPerSecond initialGameState (render appleSprite
snakeSprites) handleEvent update
```

Cijeli isječak koda:

```
main :: IO ()
main = do
    let window = InWindow "Haskell Snake Game" (windowWidth, windowHeight) (200, 200)
        backgroundColor = black
        framesPerSecond = 60
    appleSprite <- loadAppleSprite
    snakeSprites <- loadSnakeSprites
    initialGameState <- initialState
    play window backgroundColor framesPerSecond initialGameState (render appleSprite
snakeSprites) handleEvent update
```

Za razumijevanje ostatka koda `Main` modula, potrebno je proučiti `GameState` modul koji definira istoimeni tip podataka o stanju igre - `GameState`.

[2] GameState modul

GameState tip podatka 🟡

Tip podataka koji predstavlja stanje igre

Cijeli isječak koda:

```
data GameState = GameState
  { snake :: [(Float, Float)] -- Lista koordinata (tuplova) (X, Y) za svaki segment
  zmiije
  , direction :: (Float, Float) -- Trenutni smjer kretanja zmiije, tuple (X, Y)
  , apple :: (Float, Float) -- Koordinate trenutne pozicije jabuke (X, Y)
  , rng :: StdGen -- Generator slučajnih brojeva za stvaranje novih pozicija
  jabuke
  , timer :: Float -- Timer za "apple respawn"
  , appleCount :: Int -- Brojač pojedenih/sakupljenih jabuka
  , isGameOver :: Bool -- Game Over flag
  , movementTimer :: Float -- Timer za kontrolu koliko često se zmija pomiče
  } deriving Show -- Kako bi se GameState mogao ispisati u konzoli
```

initialState funkcija NEW

Funkcija koja vraća početno stanje igre (GameState)

- osim tipa `GameState`, funkcija vraća i `IO` monadu jer ima element generiranja slučajnih brojeva
- `gen <- newStdGen` - vezivanje generatora slučajnih brojeva za varijablu `gen`
- `let (applePos, newGen) = newApple gen` - generiranje nove pozicije jabuke
- `return GameState { ... }` - `IO` akcija vraća vrijednost tipa `GameState` (početno stanje igre)

Cijeli isječak koda:

```
initialState :: IO GameState -- IO operacija koja rezultira GameState tipom
initialState = do
  gen <- newStdGen
  let (applePos, newGen) = newApple gen
  return GameState
    { snake = [(0, 0), (cellSize, 0), (2 * cellSize, 0)]
    , direction = (cellSize, 0)
    , apple = applePos
    , rng = newGen
    , timer = 0
    , appleCount = 0
    , isGameOver = False
    , movementTimer = 0
    }
```

moveSnake funkcija 🐍⬆️➡️⬇️⬅️

Funkcija vraća novu poziciju zmiije na temelju trenutnog smjera kretanja (**direction**)

- `GameState` parametar je input,
- `[(Float, Float)]` je povratna vrijednost - lista koordinata `(x, y)` za svaki segment zmiije

funkcija se sastoji od **glavnog izraza** i **where klauzule**

1. `newHead` - nova pozicija glave zmiije

- izraz vraća **ново stanje tijela zmije**
- `init` funkcija vraća sve elemente liste osim zadnjeg
- `newHead` se dodaje na početak liste tijela zmije, a zatim se uklanja zadnji element `init (snake gameState)`
- `newHead` postaje nova glava zmije

```
newHead : init (snake gameState)
```

2. `where` klauzula

- `where` klauzula se koristi za definiranje dodatnih pomoćnih funkcija i varijabli unutar glavnog izraza
- `(dx, dy) = direction gameState` - ekstrakcija trenutnog smjera kretanja iz stanja igre (`dx = delta x`, `dy = delta y`) - vektor koji predstavlja **promjenu koordinata X i Y**
- `newHead = (\(x, y) -> (x + dx, y + dy)) (head $ snake gameState)` **izraz koji račune nove koordinate glave zmije**
 - `head $ snake gameState` - head funkcija vraća prvi element liste tijela zmije (dakle glavu)
 - `(\ (x, y) -> (x + dx, y + dy))` - lambda izraz koji uzima trenutno poziciju glave i dodaje promjenu koordinata (`dx`, `dy`)

Cijeli isječak koda:

```
moveSnake :: GameState -> [(Float, Float)]
moveSnake gameState = newHead : init (snake gameState)
  where
    (dx, dy) = direction gameState
    newHead = (\(x, y) -> (x + dx, y + dy)) (head $ snake gameState)
```

`snakeEatsApple` funkcija 🐍🍏

Funkcija koja provjerava je li zmija "pojela" jabuku

- `head (snake gameState)` - uzima prvi element liste tijela zmije (`snake gameState`)
- `apple gameState` - uzima trenutnu poziciju jabuke iz stanja igre
- **uspoređuje se pozicija glave zmije i jabuke** i vraća `Bool` rezultata usporedbe

Cijeli isječak koda:

```
snakeEatsApple :: GameState -> Bool
snakeEatsApple gameState = head (snake gameState) == apple gameState
```

`checkCollision` funkcija 🐍🧱

Funkcija koja provjerava je li zmija "udarila" u zid ili u samu sebe

- preciznije, provjerava se da li je nova pozicija glave zmije izvan granica prozora ili nalazi li se glava zmije u tijelu zmije

- tuple `(Float, Float)` predstavlja koordinate **glave zmije**
- lista tuplova `[(Float, Float)]` predstavlja **tijelo zmije**

prema tome možemo i podijeliti logiku za provjere sudara na dva (2) dijela:

1. Provjera sudara sa zidovima

Provjera ako je `x` koordinata glave zmije manja od lijeve granice prozora. Granica se računa kao `polovica širina prozora - veličina ćelije` gdje je: `veličina ćelije = veličina segmenta zmije`

```
x < -fromIntegral windowWidth / 2 + cellSize
```

- provjera ako je `x` koordinata glave zmije veća ili jednaka od desne granice prozora

```
x >= fromIntegral windowWidth / 2 - cellSize
```

- provjerava ako je `y` koordinata glave zmije manja od donje granice prozora

```
y < -fromIntegral windowHeight / 2 + cellSize
```

- provjera ako je `y` koordinata glave zmije veća ili jednaka od gornje granice prozora

```
y >= fromIntegral windowHeight / 2 - cellSize
```

- `fromIntegral` koristi se za pretvaranje `windowWidth` i `windowHeight` u `Float` tip podataka kako bismo mogli koristiti operacije s pomičnim zarezom

2. Provjera sudara sa tijelom zmije

Provjera je li se zmija sudarila u samu sebe

- jednostavno se provjerava je li glava zmije prisutna u listi pozicija tijela zmije `(x, y) `elem` body`
- `elem` funkcija provjerava sadrži li lista element jednak prvom argumentu (u ovom slučaju `(x, y)`)

Cijeli isječak koda:

```
checkCollision :: (Float, Float) -> [(Float, Float)] -> Bool
checkCollision (x, y) body = x < -fromIntegral windowWidth / 2 + cellSize
    || x >= fromIntegral windowWidth / 2 - cellSize
    || y < -fromIntegral windowHeight / 2 + cellSize
    || y >= fromIntegral windowHeight / 2 - cellSize
    || (x, y) `elem` body
```

growSnake funkcija 🐍🐍🐍

Rekurzivna funkcija koja povećava tijelo zmije za jedan segment svaki put kada zmija pojede jabuku

- ulazni parametar je lista koordinata (tuplova) `[(Float, Float)]` za svaki segment trenutne zmije

- povratna vrijednost je lista koordinata (tuplova) `[(Float, Float)]` za svaki segment zmije nakon povećanja

1. Korištenje **pattern matchinga**

```
growSnake (x:xs) = x : x : xs` - ako lista nije prazna, uzmi prvi element liste `x` i ostatak liste `xs`
```

`(x:xs)` - razdvaja prvu koordinatu `x` (prvi segment tijela zmije) i ostatak liste `xs`, tj. tijela.

`x : x : xs` - konstruira novu listu gdje se prvi segment `x` duplira, tako da se zmija povećava za jedan segment. **Novi segment je kopija prvog segmenta, što znači da će zmija narasti na mjestu svoje glave.**

Primjer

Ako je zmija trenutno definirana kao `[(2, 2), (1, 2), (0, 2)]`, poziv `growSnake [(2, 2), (1, 2), (0, 2)]` rezultira novom listom `[(2, 2), (2, 2), (1, 2), (0, 2)]`.

2. Obrada prazne liste

- ako nema segmenata (lista je prazna), funkcija jednostavno vraća praznu listu. Ovo je zaštitni mehanizam za slučaj kada se funkcija pozove s praznom listom, kako bi se izbjegla greška.

```
growSnake [] = []
```

Funkcija koristi **rekurziju** kroz uvjetovanje: ako je lista prazna, vraća praznu listu; inače, duplicira prvi segment (glavu) kako bi zmija "narasla".

Cijeli isječak koda:

```
growSnake :: [(Float, Float)] -> [(Float, Float)]
growSnake (x:xs) = x : x : xs
growSnake [] = []
```

Završetak modula GameState

[3] Main modul 2/2 (nastavak)

Jednom kad smo definirali `GameState` modul, možemo se vratiti na `Main` modul funkcije i objasniti preostale funkcije budući da intenzivno koriste stanje igre `GameState`.

update funkcija

Funkcija vraća novo stanje igre na temelju vremenskog koraka

- `Float` parametar predstavlja vremenski korak (delta vremena) između dva kadra (frame)
- `GameState` parametar predstavlja trenutno stanje igre
- povratna vrijednost je novo stanje igre `GameState`

1. isječak provjerava je li stanje igre gotovo, ako jest poziva funkciju `resetGame` (u nastavku)

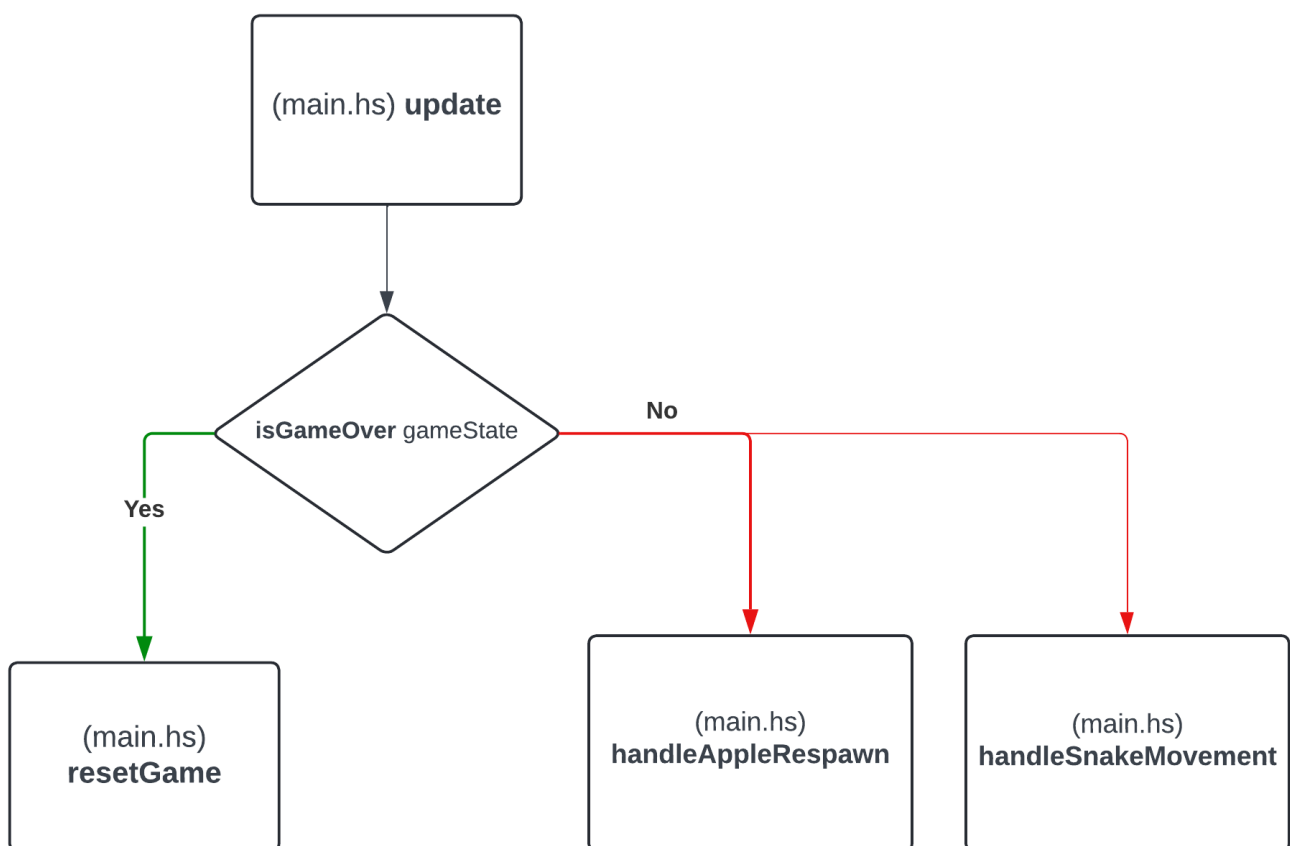
- `isGameOver gameState = resetGame`

2. isječak ažurira stanje ako igra nije gotova.

- Prvo ažuriraj `timer` povećanjem za `seconds` (`gameState { timer = timer gameState + seconds }`)
- `handleSnakeMovement seconds` ažurira poziciju zmiје (funkcija u nastavku)
- `handleAppleRespawn` ažurira poziciju jabuke (funkcija u nastavku)

Cijeli isječak koda:

```
update :: Float -> GameState -> GameState
update seconds gameState
  | isGameOver gameState = resetGame
  | otherwise = handleAppleRespawn . handleSnakeMovement seconds $ gameState { timer =
timer gameState + seconds }
```



resetGame funkcija ← BACK

- funkcija koja resetira igru na početno stanje
- ne prima argumente, a povratna vrijednost je početno stanje igre
- `apple = fst $ newApple rng` - vraća par (jabuka, novi generator slučajnih brojeva)
- `rng = mkStdGen 42` - postavlja seed za generator slučajnih brojeva

```

resetGame :: GameState
resetGame = GameState
  { snake = [(0, 0), (cellSize, 0), (2 * cellSize, 0)]
  , direction = (cellSize, 0)
  , apple = fst $ newApple rng
  , rng = snd $ newApple rng
  , timer = 0
  , appleCount = 0
  , isGameOver = False
  , movementTimer = 0
  }
where
  rng = mkStdGen 42

```

handleSnakeMovement i updateGameStateAfterMovement funkcije za kretanje 🐍🔄

Funkcije koje upravljaju kretanjem zmije i vraćanjem novog stanja igre nakon kretanja.

1. handleSnakeMovement funkcija

Funkcija prima 2 argumenta:

- `Float` - proteklo vrijeme u sekundama između dva kadra
- `GameState` - trenutno stanje igre

Uvjet provjerava je li prošlo dovoljno vremena da se zmije pomakne. Ako je `movementTimer` veći ili jednak `movementThreshold`, poziva se funkcija `updateGameStateAfterMovement` kako bi se ažuriralo stanje igre nakon pokreta zmije.

```

movementTimer gameState >= movementThreshold =
  updateGameStateAfterMovement gameState

```

- Ako nije prošlo dovoljno vremena, ažurira se `movementTimer` dodavanjem proteklog vremena `seconds`.

```

otherwise = gameState { movementTimer = movementTimer gameState + seconds }

```

- `movementThreshold = 0.07` - prag za kretanje zmije. Ako je `movementTimer` veći ili jednak ovom pragu, zmija se pomakne - **ujedno je parametar za kontrolu brzine kretanja zmije** (manji prag = brže kretanje).

Cijeli isječak koda:

```

handleSnakeMovement :: Float -> GameState -> GameState
handleSnakeMovement seconds gameState
  | movementTimer gameState >= movementThreshold = updateGameStateAfterMovement gameState
  | otherwise = gameState { movementTimer = movementTimer gameState + seconds }
where
  movementThreshold = 0.07

```

2. updateGameStateAfterMovement funkcija

Funkcija koja vraća novo stanje igre nakon svake kretnje zmije

- provjerava je li zmija pojela jabuku ili imala sudar
- `GameState` parametar je trenutno stanje igre, a povratna vrijednost je novo stanje igre (`GameState`)

Ako je zmija pojela jabuku (`snakeEatsApple gameState` vraća `True`), poziva se `handleAppleEaten` funkcija (u nastavku) koja vraća novo stanje igre. `snakeEatsApple` funkcija iz `GameState` modula provjerava je li glava zmije na istoj poziciji kao jabuka.

```
snakeEatsApple gameState = handleAppleEaten gameState
```

Ako zmija napravi koliziju (rekli smo da se provjerava sudar glave zmije s tijelom ili zidovima koristeći `checkCollision` funkciju iz `GameState` modula), postavlja se `isGameOver` flag na `True` i `movementTimer` na `0`.

- `head` funkcija uzima prvi element liste tijela zmije (glavu) i provjerava je li došlo do sudara (*no pun intended*)
- `tail` funkcija uzima sve elemente liste osim prvog (preostalo tijelo zmije)

Ako ni ne jede jabuku, niti je u koliziji, ažurira se stanje igre tako da se zmija pomakne na novu poziciju (`newSnake`), a `movementTimer` se resetira.

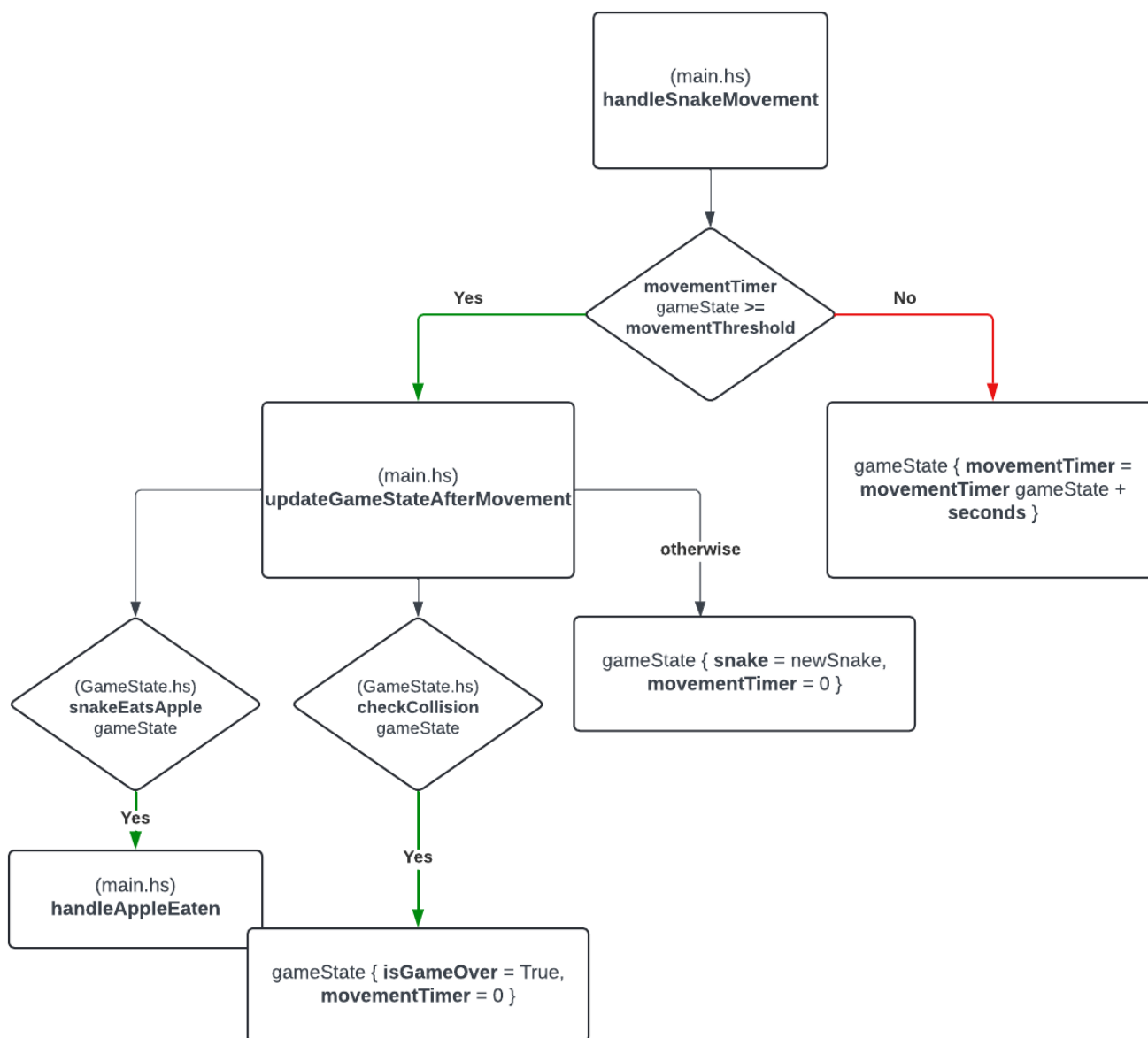
- izračun nove pozicije zmije `newSnake = GS.moveSnake gameState`. `GS` je alias za `GameState` modul (funkcija `moveSnake` iz `GameState` modula vraća novu poziciju zmije)

Cijeli isječak koda:

```

updateGameStateAfterMovement :: GameState -> GameState
updateGameStateAfterMovement gameState
  | snakeEatsApple gameState = handleAppleEaten gameState
  | checkCollision (head newSnake) (tail newSnake) = gameState { isGameOver = True,
movementTimer = 0 }
  | otherwise = gameState { snake = newSnake, movementTimer = 0 }
where
  newSnake = GS.moveSnake gameState

```



Ilustracija funkcija za kretanje zmije

handleAppleEaten i handleAppleRespawn funkcije 🍏🍏🍏

Funkcije koje upravljaju scenarijem kada zmija pojede jabuku i kada jabuka treba biti ponovo stvorena

1. handleAppleEaten funkcija

Funkcija koja vraća novo stanje igre nakon što zmija pojede jabuku

- `GameState` parametar je trenutno stanje igre
- povratna vrijednost je novo stanje igre (`GameState`)

Funkcija radi sljedeće:

- (`GameState.hs`) `growSnake` funkcija povećava tijelo zmije za jedan segment, argument je trenutno tijelo zmije koje se dobiva iz `snake gameState`
- (`Apple.hs`) `newApple` funkcija vraća novu nasumičnu poziciju jabuke i novi generator slučajnih brojeva koji se pohranjuju u `newApplePos` i `newRng`

- ažurira se stanje igre s novom pozicijom jabuke, novim generatorom i resetiranim timerom za jabuku (`timer`)
- broj jabuka se povećava za 1 (`appleCount`)

Cijeli isječak koda:

```
handleAppleEaten :: GameState -> GameState
handleAppleEaten gameState = gameState
  { snake = growSnake (snake gameState)
  , apple = newApplePos
  , rng = newRng
  , timer = 0
  , appleCount = appleCount gameState + 1
  , movementTimer = 0
  }
where
  (newApplePos, newRng) = newApple (rng gameState)
```

2. `handleAppleRespawn` funkcija

Funkcija koja vraća novo stanje igre nakon što prođe određeno vrijeme i novom jabukom

- `GameState` parametar je trenutno stanje igre
- povratna vrijednost je novo stanje igre (`GameState`)

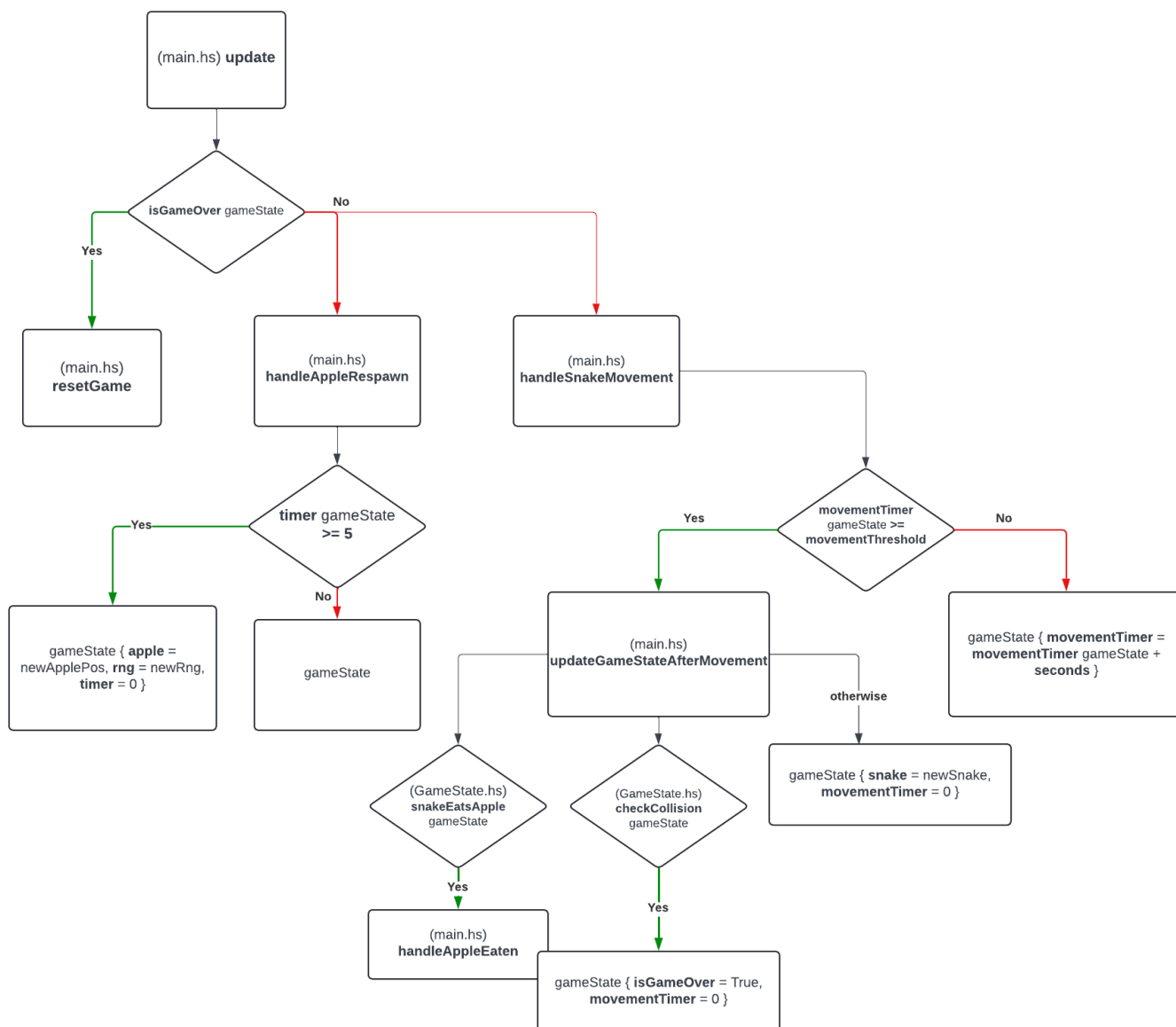
Ako je prošlo 5 sekundi (`timer gameState >= 5`), jabuka se ponovno stvara na novoj nasumičnoj poziciji. Nova pozicija jabuke je `newApplePos`, a novi generator slučajnih brojeva je `newRng`. Timer se resetira na 0.

Ako nije prošlo 5 sekundi, stanje igre ostaje nepromijenjeno.

`where` klauzula je ista kao i u `handleAppleEaten` funkciji.

Cijeli isječak koda:

```
handleAppleRespawn :: GameState -> GameState
handleAppleRespawn gameState
  | timer gameState >= 5 = gameState { apple = newApplePos, rng = newRng, timer = 0 }
  | otherwise = gameState
where
  (newApplePos, newRng) = newApple (rng gameState)
```



Ilustracija svih funkcija u `Main` modulu

[4] Apple modul

loadAppleSprite funkcija 🍏 🖼️

Funkcija koja učitava sprite jabuke

- funkcija vraća `IO Picture` monadu jer vraća akciju koja daje objekt tipa `Picture`
- za učitavanje slika koristi se funkcija `loadJuicy` iz [gloss-juicy](#) biblioteke
- učitana slika je `Maybe Picture` tipa i koristi se pattern matching za izvlačenje slike iz `Maybe` konteksta
- `do` notacija zbog sekvencijalnih operacija unutar monade

Cijeli isječak koda:

```
loadAppleSprite :: IO Picture
loadAppleSprite = do
  Just apple <- loadJuicy "assets/apple.png"
  return apple
```

newApple funkcija

Funkcija koja vraća novu poziciju jabuke i novi generator slučajnih brojeva

- funkcija prima generator slučajnih brojeva `StdGen`
- vraća tuple `(Float, Float), StdGen` - prva vrijednost je nova pozicija jabuke (tuple), a druga je novi generator slučajnih brojeva

Funkcija generira novu poziciju za jabuku `StdGen` generator.

- `borderOffset = 2 * cellSize` - offset za izbjegavanje rubnog područja. Definira se kao dvostruka veličina ćelije, što znači da se jabuka neće postaviti preblizu ruba prozora.
- `randomR` funkcija generira nasumičan broj unutar određenog raspona. `x` i `y` koordinate se generiraju odvojeno, svaka s vlastitim generatorom (`gen` za `x` i `gen1` za `y`).
- `alignToGrid` je pomoćna funkcija koja se koristi za poravnanje koordinata na mrežu. Funkcija poravnana koordinate na mrežu veličina `cellSize`. Prvo podijeli koordinatu s veličinom ćelije, zatim zaokružuje rezultat na najbliži cijeli broj i množi ga nazad s veličinom ćelije.

Cijeli isječak koda:

```
newApple :: StdGen -> ((Float, Float), StdGen)
newApple gen = ((alignToGrid x, alignToGrid y), newGen)
  where
    borderOffset = 2 * cellSize -- Offset to avoid the border area
    (x, gen1) = randomR (-fromIntegral windowWidth / 2 + borderOffset, fromIntegral
windowWidth / 2 - borderOffset) gen
    (y, newGen) = randomR (-fromIntegral windowHeight / 2 + borderOffset, fromIntegral
windowHeight / 2 - borderOffset) gen1

alignToGrid :: Float -> Float
alignToGrid coord = fromIntegral (round (coord / cellSize)) * cellSize
```

renderApple funkcija

Funkcija koja iscrtava jabuku na ekranu

Koriste se pomoćne funkcije `Gloss` biblioteke za iscrtavanje slike na ekranu:

- `translate :: Float -> Float -> Picture -> Picture` - funkcija koja postavlja sliku na određene koordinate
- `scale :: Float -> Float -> Picture -> Picture` - funkcija koja skalira sliku s danim faktorima na X i Y osi

Slika jabuke se skalira na veličinu `(cellSize / 25)` i postavlja na iste koordinate kao i pozicija jabuke.

```
scale (cellSize / 25) (cellSize / 25) appleSprite
```

Postavljanje skalirane slike na određene koordinate `x` i `y`:

```
translate x y $ scale (cellSize / 25) (cellSize / 25) appleSprite
```

Cijeli isječak koda:

```
renderApple :: Picture -> (Float, Float) -> Picture
renderApple appleSprite (x, y) = translate x y $ scale (cellSize / 25) (cellSize / 25)
appleSprite
```

[5] Input modul

`handleEvent` funkcija

Funkcija omogućava upravljanje korisniku smjerom kretanja zmije

- Koriste se tipke `w`, `a`, `s`, `d` za kretanje zmije: `gore`, `lijevo`, `dolje` i `desno`

Funkcija prima dva argumenta:

- `(Graphics.Gloss.Interface.Pure.Game) Event` - događaj koji se dogodio (npr. pritisak tipke)
- `GameState` - trenutno stanje igre

Svaka grana funkcije provjerava koja je tipka pritisnuta i ažurira smjer kretanja zmije u skladu s tim. Na primjer, za kretanje "gore" tipkom `w`, koristi se sljedeći izraz:

```
handleEvent (EventKey (Char 'w') Down _ _) gameState
  | direction gameState /= (0, -cellSize) = gameState { direction = (0, cellSize) }
```

- `EventKey (Char 'w') Down _ _` - odgovara događaju kada je pritisnuta tipka `w` i stanje tipke je `Down` (pritisnuta)
- Provjerav se trenutni smjer kretanja zmije `direction gameState` i ažurira se samo ako trenutni smjer nije suprotan odabranom smjeru. Na primjer, ako je zmija kretala prema dolje, ne može se kretati prema gore. Preciznije, ako zmija ne ide dolje `(0, -cellSize)``, smjer se može promijeniti i mijenja se na gore `(0, cellSize)``.

Ako se dogodi bilo koji drugi događaj, stanje igre ostaje nepromijenjeno (`handleEvent _ gameState = gameState`).

Cijeli isječak koda:


```

handleEvent :: Event -> GameState -> GameState
handleEvent (EventKey (Char 'w') Down _ _) gameState
  | direction gameState /= (0, -cellSize) = gameState { direction = (0, cellSize) }
handleEvent (EventKey (Char 'a') Down _ _) gameState
  | direction gameState /= (cellSize, 0) = gameState { direction = (-cellSize, 0) }
handleEvent (EventKey (Char 's') Down _ _) gameState
  | direction gameState /= (0, cellSize) = gameState { direction = (0, -cellSize) }
handleEvent (EventKey (Char 'd') Down _ _) gameState
  | direction gameState /= (-cellSize, 0) = gameState { direction = (cellSize, 0) }
handleEvent _ gameState = gameState

```

[6] AppleCounter modul

Modul sadrži funkciju za iscrtavanje broja jabuka koje je zmija pojela na ekranu.

Definiranje boje teksta i pomoćna `boldText` funkcija 🎨

Na početku je definirana crvena boja teksta pomoću funkcije `makeColorI` iz modula `Graphics.Gloss.Color` koja prima RGBA vrijednosti.

```

Graphics.Gloss.Color
```haskell
textColor :: Color
textColor = makeColorI 218 72 15 255 -- #DA480F

```

Funkcija `boldText` crta podebljani tekst

- funkcija je implementirana budući da `Gloss` nema ugrađenu podršku za crtanje boldanog teksta
- funkcija prima `string` odnosno tekst koji se želi prikazati
- vraća `Picture` - sliku koja sadrži tekst

Funkcionalnost se postiže crtanjem teksta više puta s malim pomacima (offsets) u svim smjerovima.

- `offsets = [0, 1, -1]` - lista pomaka za svaki smjer
- `pictures` funkcija kombinira više slika (`Pictures`) u jednu
- `translate x y $ text str` crta tekst `str` s pomakom `(x, y)`

Cijeli isječak koda:

```

boldText :: String -> Picture
boldText str = pictures [translate x y $ text str | x <- offsets, y <- offsets]
 where
 offsets = [0, 1, -1]

```

### `renderAppleCounter` funkcija 🎮🍏📦

Funkcija koja iscrtava GUI za broj jabuka na ekranu

Funkcija prima 2 argumenta:

- `Picture` - slika jabuke tj. `appleSprite`
- `Int` - broj jabuka koje je zmija pojela
- povratna vrijednost je `Picture` - objekt koji prikazuje ikonu jabuke i broj jabuka

#### 1. Izraz za `apple counter` tekst:

- `translate 50 (-20)` - pomak teksta na koordinate `(50, -20)`
- `scale 0.2 0.2` - skalira tekst na 20% originalne veličine
- `color textColor` - postavlja boju teksta na `textColor`
- `boldText $ "Jabuke: " ++ show count` - crta podebljani tekst koji prikazuje broj jabuka (funkcija `show` pretvara argument u `String`)

```
translate 50 (-20) $ scale 0.2 0.2 $ color textColor . boldText $ "Jabuke: " ++ show count
```

#### 2. `where` klauzula:

- `x = -fromIntegral windowWidth / 2` - određivanje lijeve strane ekrana
- `y = fromIntegral windowHeight / 2 - 35` - vrh ekrana, pomaknut za 35 piksela prema dolje
- `appleIcon` - slika jabuke postavljena na koordinate `(25, -10)` i skalirana na originalnoj veličini

```
where
 x = -fromIntegral windowWidth / 2
 y = fromIntegral windowHeight / 2 - 35
 appleIcon = translate 25 (-10) $ scale 1 1 appleSprite
```

Cijeli isječak koda:

```
renderAppleCounter :: Picture -> Int -> Picture
renderAppleCounter appleSprite count = translate x y $ pictures [appleIcon, translate 50
(-20) $ scale 0.2 0.2 $ color textColor . boldText $ "Jabuke: " ++ show count]
where
 x = -fromIntegral windowWidth / 2
 y = fromIntegral windowHeight / 2 - 35
 appleIcon = translate 25 (-10) $ scale 1 1 appleSprite
```

## [7] SnakeRender modul

Ovaj modul za cilj ima iscrtavanje ispravnih slika (sprite-ova) za svaki segment zmije.

### `loadSnakeSprites` funkcija 🐍🖼️

IO funkcija koja učitava sprite-ove za svaki segment zmije

- korištenje `loadJuicy` funkcije za učitavanje slika iz datoteka (isto kao kod jabuke)

- nakon što su sve slike učitane, vraća se lista parova [(String, Picture) ...] gdje je prvi element ime segmenta zmiје, a drugi slika segmenta zmiје

Imena elemenata jednaka su nazivima datoteka slika.

Cijeli isječak koda:

```
loadSnakeSprites :: IO [(String, Picture)]
loadSnakeSprites = do
 headLeft <- fromJust <$> loadJuicy "assets/head_left.png"
 headRight <- fromJust <$> loadJuicy "assets/head_right.png"
 headUp <- fromJust <$> loadJuicy "assets/head_up.png"
 headDown <- fromJust <$> loadJuicy "assets/head_down.png"

 bodyHorizontal <- fromJust <$> loadJuicy "assets/body_horizontal.png"
 bodyVertical <- fromJust <$> loadJuicy "assets/body_vertical.png"

 tailUp <- fromJust <$> loadJuicy "assets/tail_up.png"
 tailDown <- fromJust <$> loadJuicy "assets/tail_down.png"
 tailLeft <- fromJust <$> loadJuicy "assets/tail_left.png"
 tailRight <- fromJust <$> loadJuicy "assets/tail_right.png"

 return [("head_left", headLeft)
 , ("head_right", headRight)
 , ("head_up", headUp)
 , ("head_down", headDown)
 , ("body_horizontal", bodyHorizontal)
 , ("body_vertical", bodyVertical)
 , ("tail_up", tailUp)
 , ("tail_down", tailDown)
 , ("tail_left", tailLeft)
 , ("tail_right", tailRight)
]
```



Slike sprite-ova za zmiју, [source](#)

## directions i njene pomoćne funkcije 🐍⬆️➡️⬇️⬅️

Funkcija i pomoćne funkcije za određivanje smjera kretanja segmenata zmiје

Uključene su sljedeće 3 pomoćne funkcije:

- `getBodyDirection` - funkcija određuje smjer segmenata između glave i repa.
- `getHeadDirection` - funkcija uzima dvije pozicije (prvi i drugi segment zmiје) i vraća string koji predstavlja smjer glave te boolean vrijednosti za zrcaljenje.

- `getTailDirection` - funkcija radi slično kao `getHeadDirection`, ali za posljednja dva segmenta zmije.

Funkcija `directions` uzima listu koordinata segmenata `[(Float, Float)]` zmije i vraća **listu smjerova za svaki segment** zmije `[(String, Bool, Bool)]`.

- `String` predstavlja tip segmenta (glava, tijelo, rep) i njegov smjer (*npr. "head\_left"*)
- `Bool` vrijednosti predstavljaju treba li sliku segmenta zrcaliti horizontalno ili vertikalno

Ako je lista koordinata prazna, funkcija vraća praznu listu jer nema segmenata za renderiranje.

```
directions [] = []
```

Ako lista koordinata sadrži samo jedan segment (nepotpuna zmija), funkcija također vraća praznu listu jer nije moguće odrediti smjer segmenata.

```
directions snake = headDirection : bodyDirections ++ [tailDirection]
```

Ako lista koordinata sadrži više od jednog segmenta (valjana zmija) određuju se smjerovi za svaki segment.

- prvo se određuje smjer glave pomoću `headDirection`
- zatim se određuju smjerovi tijela pomoću `bodyDirections`
- na kraju se određuje smjer repa pomoću `tailDirection`

Koriste se operatori `:` (konstrukcija liste) i `++` (konkatenacija listi) kako bismo sastavili konačnu listu smjerova.

Cijeli isječak koda:

```
directions :: [(Float, Float)] -> [(String, Bool, Bool)]
directions [] = []
directions [x] = []
directions snake = headDirection : bodyDirections ++ [tailDirection]
 where
 headDirection = getHeadDirection (snake !! 0) (snake !! 1)
 bodyDirections = zipWith3 getBodyDirection (init (tail snake)) (tail (init snake))
 (drop 2 snake)
 tailDirection = getTailDirection (last (init snake)) (last snake)
```

- `getHeadDirection` uzima prve dvije koordinate zmije `snake !! 0` (glava) i `snake !! 1` (prvi segment tijela) te vraća tuple `(String, Bool, Bool)` koji predstavlja smjer glave i boolean vrijednosti za zrcaljenje.
- `bodyDirections` određujemo pomoću funkcije `zipWith3 getBodyDirection`:
  - `init (tail snake)` - uzima sve segmente osim prvog i posljednjeg (glava i rep)
  - `tail (init snake)` - uzima sve segmente zmije osim prvog i posljednjeg, ali pomaknuto za jedno mjesto unaprijed
  - `drop 2 snake` - uzima sve segmente zmije osim prva dva (glava i prvog segmenta tijela)

- funkcija `zipWith3` primjenjuje `getBodyDirection` na svaku trojku koordinata (prethodni, trenutni i sljedeći segment) i vraća listu smjerova tijela.
- `getTailDirection` uzima zadnje dvije koordinate zmije:
  - `last (init snake)` - predzadnji segment zmije (zadnji segment tijela)
  - `last snake` - zadnji segment zmije (rep)
  - funkcija `getTailDirection` vraća tuple `(String, Bool, Bool)` koji predstavlja smjer repa

```

getHeadDirection :: (Float, Float) -> (Float, Float) -> (String, Bool, Bool)
getHeadDirection (x1, y1) (x2, y2)
 | x1 == x2 && y1 < y2 = ("head_down", False, False)
 | x1 == x2 && y1 > y2 = ("head_up", False, False)
 | x1 < x2 && y1 == y2 = ("head_left", False, False)
 | x1 > x2 && y1 == y2 = ("head_right", False, False)
 | otherwise = ("body_horizontal", False, False)

getTailDirection :: (Float, Float) -> (Float, Float) -> (String, Bool, Bool)
getTailDirection (x1, y1) (x2, y2)
 | x1 == x2 && y1 < y2 = ("tail_up", False, False)
 | x1 == x2 && y1 > y2 = ("tail_down", False, False)
 | x1 < x2 && y1 == y2 = ("tail_right", False, False)
 | x1 > x2 && y1 == y2 = ("tail_left", False, False)
 | otherwise = ("tail_right", False, False)

getBodyDirection :: (Float, Float) -> (Float, Float) -> (Float, Float) -> (String, Bool, Bool)
getBodyDirection (x1, y1) (x2, y2) (x3, y3)
 -- Straight horizontal and vertical segments
 | x1 == x2 && x2 == x3 = ("body_vertical", False, False)
 | y1 == y2 && y2 == y3 = ("body_horizontal", False, False)
 | otherwise = ("body_horizontal", False, False)

```

## renderSnake funkcija 🐍

Funkcija primjenjuje odgovarajući sprite za svaki segment zmije

- funkcija iscrtava zmiju koristeći pomoćne funkcije za određivanje smjera

Funkcija prima 2 argumenta:

- `sprites` lista parova `(String, Picture)` gdje je prvi element ime segmenta zmije, a drugi slika segmenta zmije
- `snake` lista koordinata segmenata zmije `[(Float, Float)]`

### 1. Tijelo funkcije:

Ovdje se koristi funkcija `Graphics.Gloss pictures` funkcija koja spaja više slika u jednu. `zipWith` kombinira dvije liste (`snake` i `directions snake`) koristeći pomoćnu funkciju `renderSegment`. Rezultat je lista segmenata zmije kao slika, koja se zatim spaja u jednu sliku.

```
renderSnake sprites snake = pictures $ zipWith renderSegment snake (directions snake)
```

## 2. `renderSegment` i `applyFlip` funkcije:

Funkcija `renderSegment` iscrtava jedan segment slike i prima 2 argumenta:

- `(x, y)` - koordinate segmenta
- `(dir, flipX, flipY)` - smjer i informacije o zrcaljenju segmenta

Što se tiče implementacije, funkcija prvo primjenjuje `translate` i `scale` funkcije za postavljanje segmenta na odgovarajuće koordinate i skaliranje na odgovarajuću veličinu (da odgovara veličini ćelije). Zatim se primjenjuje `applyFlip` funkcija za zrcaljenje segmenta smao ako je potrebno.

Odgovarajući sprite se dohvaća iz liste `sprites` koristeći `fromJust` i `lookup` funkcije.

```
where
 renderSegment (x, y) (dir, flipX, flipY) = translate x y $ scale (cellSize / 40)
 (cellSize / 40) $ applyFlip flipX flipY $ fromJust (lookup dir sprites)
```

Cijeli isječak koda:

```
renderSnake :: [(String, Picture)] -> [(Float, Float)] -> Picture
renderSnake sprites snake = pictures $ zipWith renderSegment snake (directions snake)
 where
 renderSegment (x, y) (dir, flipX, flipY) = translate x y $ scale (cellSize / 40)
 (cellSize / 40) $ applyFlip flipX flipY $ fromJust (lookup dir sprites)
 applyFlip True False = scale (-1) 1
 applyFlip False True = scale 1 (-1)
 applyFlip True True = scale (-1) (-1)
 applyFlip False False = id
```

## [8] Render modul

Posljednji modul `render` sadrži istoimenu funkciju za iscrtavanje stanja igre (`GameState`) na ekranu.

Na početku su definirane **boje**. Boje `color1` i `color2` su za crtanje mreže ćelija (grid), dok su boje `edgeColor1` i `edgeColor2` za rubne ćelije mreže.

```
color1, color2, edgeColor1, edgeColor2 :: Color
color1 = makeColorI 167 215 88 255 -- #A7D758
color2 = makeColorI 162 208 82 255 -- #A2D052
edgeColor1 = makeColorI 90 58 18 255 -- #693A12
edgeColor2 = makeColorI 75 48 17 255 -- #5E3511
```

### `render` funkcija 🎮🖥️

Funkcija koja iscrtava stanje igre na ekranu

Funkcija prima 3 argumenta:

- `appleSprite` - slika jabuke
- `snakeSprites` - lista sprite-ova za segmente zmije
- `gameState` - trenutno stanje igre

Funkcija vraća `Picture` koja predstavlja cijelu scenu igre. Koristi `pictures` za spajanje više slika u jednu:

- `gridPicture` - slika mreže ćelija
- `snakePicture` - slika zmije
- `applePicture` - slika jabuke
- `appleCounterPicture` - slika broja pojedenih jabuka

## 1. Iscrtavanje zmije

- Funkcija kreira sliku zmije pomoću funkcije `renderSnake` iz modula `SnakeRender`, koristeći sprite-ove zmije i koordinate segmenata zmije iz stanja igre (`gameState`).

```
snakePicture = renderSnake snakeSprites (snake gameState)
```

## 2. Iscrtavanje jabuke

- Funkcija kreira sliku jabuke pomoću funkcije `renderApple` iz modula `Apple`, koristeći sliku jabuke i koordinate jabuke iz stanja igre.

```
applePicture = renderApple appleSprite (apple gameState)
```

## 3. Iscrtavanje broja jabuka

- Funkcija kreira sliku broja pojedenih jabuka pomoću funkcije `renderAppleCounter` iz modula `AppleCounter`, koristeći sliku jabuke i broj pojedenih jabuka iz stanja igre.

```
appleCounterPicture = renderAppleCounter appleSprite (appleCount gameState)
```

## 4. Iscrtavanje mreže

- Funkcija kreira sliku mreže koristeći listu koordinata za svaki kvadrat u mreži. Za svaku koordinatu (`x`, `y`), kreira se kvadrat `rectangleSolid` veličine `cellSize` i boje određene funkcijom `cellColor`.

```
-- Create a picture for the grid
gridPicture = pictures [translate x y $ color (cellColor x y) $ rectangleSolid
cellSize cellSize
 | x <- [-fromIntegral windowHeight / 2, -fromIntegral
windowWidth / 2 + cellSize .. fromIntegral windowHeight / 2 - cellSize]
 , y <- [-fromIntegral windowHeight / 2, -fromIntegral
windowHeight / 2 + cellSize .. fromIntegral windowHeight / 2 - cellSize]
]
```

## 5. Funkcija `cellColor`

Funkcija `cellColor` određuje boju ćelije na poziciji (`x`, `y`):

```

cellColor x y
 | isEdge x y = if isEdgeColor1 x y then edgeColor1 else edgeColor2
 | even (floor (x / cellSize) + floor (y / cellSize)) = color1
 | otherwise = color2

```

## 6. Funkcija `isEdge`

Funkcija `isEdge` provjerava je li ćelija na poziciji `(x, y)` rubna ćelija:

```

isEdge :: Float -> Float -> Bool
isEdge x y = x == -fromIntegral windowWidth / 2
 || x == fromIntegral windowWidth / 2 - cellSize
 || y == -fromIntegral windowHeight / 2
 || y == fromIntegral windowHeight / 2 - cellSize

```

## 7. Funkcija `isEdgeColor1`

Funkcija `isEdgeColor1` određuje treba li rubna ćelija koristiti `edgeColor1` ili `edgeColor2` (boje idu naizmjenično iz estetskih razloga). Koristi isti kriterij pariteta kao i za unutarnje ćelije (`even (floor (x / cellSize) + floor (y / cellSize))`).

Cijeli isječak koda:

```

render :: Picture -> [(String, Picture)] -> GameState -> Picture
render appleSprite snakeSprites gameState = pictures [gridPicture, snakePicture,
applePicture, appleCounterPicture]
 where
 -- Create a picture for the snake
 snakePicture = renderSnake snakeSprites (snake gameState)

 -- Create a picture for the apple
 applePicture = renderApple appleSprite (apple gameState)

 -- Create a picture for the apple counter
 appleCounterPicture = renderAppleCounter appleSprite (appleCount gameState)

 -- Create a picture for the grid
 gridPicture = pictures [translate x y $ color (cellColor x y) $ rectangleSolid
cellSize cellSize
 | x <- [-fromIntegral windowWidth / 2, -fromIntegral
windowWidth / 2 + cellSize .. fromIntegral windowWidth / 2 - cellSize]
 , y <- [-fromIntegral windowHeight / 2, -fromIntegral
windowHeight / 2 + cellSize .. fromIntegral windowHeight / 2 - cellSize]
]

 -- Determine the color of the cell
 cellColor x y
 | isEdge x y = if isEdgeColor1 x y then edgeColor1 else edgeColor2
 | even (floor (x / cellSize) + floor (y / cellSize)) = color1
 | otherwise = color2

```

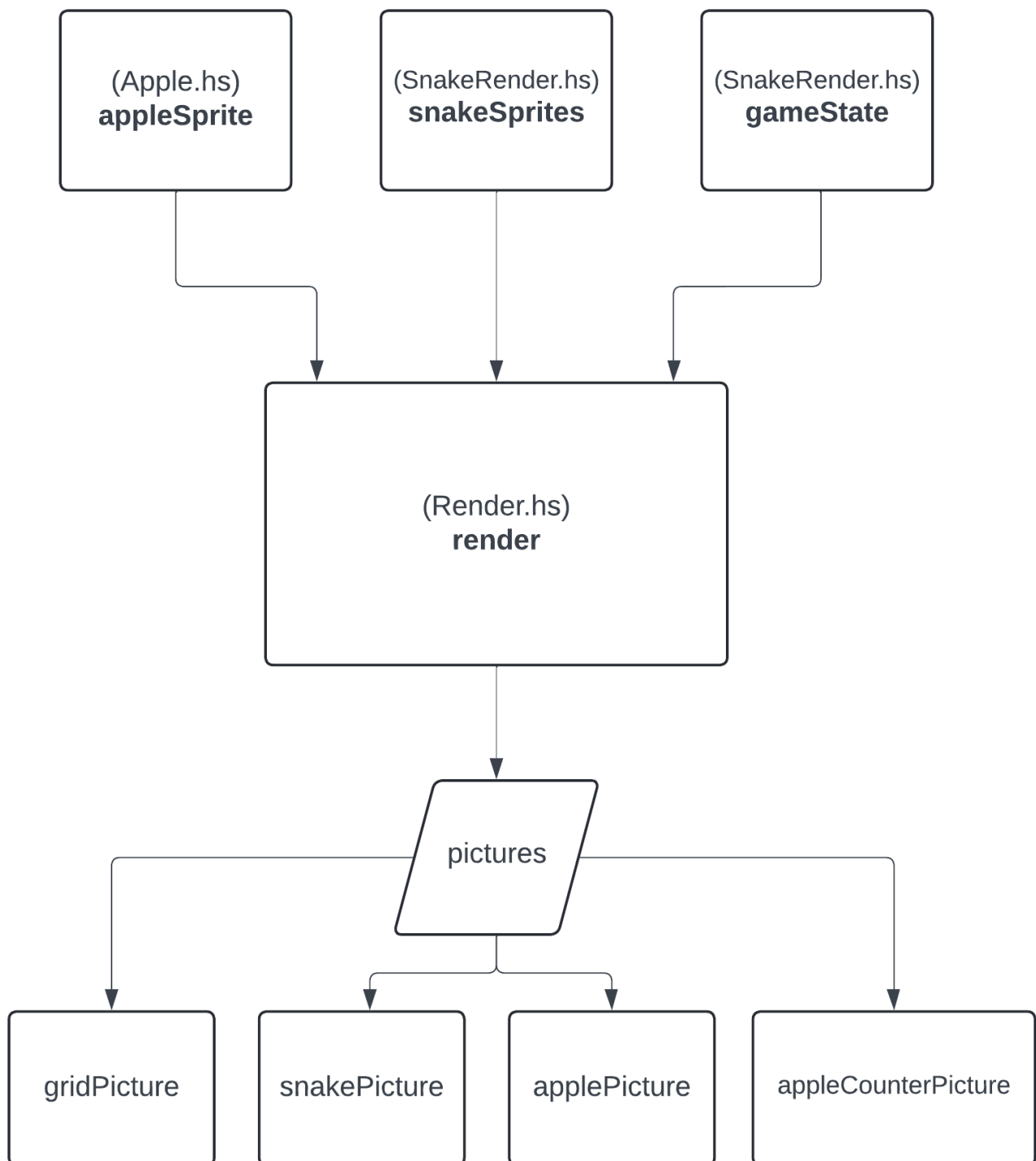


```

isEdge :: Float -> Float -> Bool
isEdge x y = x == -fromIntegral windowWidth / 2
 || x == fromIntegral windowWidth / 2 - cellSize
 || y == -fromIntegral windowHeight / 2
 || y == fromIntegral windowHeight / 2 - cellSize

-- Determine if the edge cell should use edgeColor1 or edgeColor2
isEdgeColor1 x y = even (floor (x / cellSize) + floor (y / cellSize))

```



Prikaz funkcije `render`, njenih ulaznih parametara i povratne `pictures` vrijednosti