gta

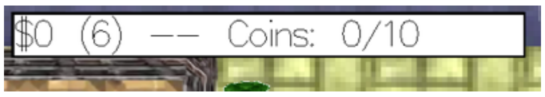
GTA implemented in Haskell

Grand Theft Auto I (originally released in 1997, for MS-DOS/Windows).

You'll spawn in a beautifull city where you can walk and drive around. Next to you, other people are doing theire daily stuff and will move along you. Drivers are busy people, so watch out, they are able to kill you when they drive over you. It's your mission to find all the coins that are around to earn as much cash as possible. Cash can be earned as well by killing people by driving over them.

When you start the game, a begin-screen will pop up. This should explain the game once again. Everything is controlled by keys; S: Start game P: Pause game (brings you to the start-screen again) R: Restart level (resets player state, scores and location) C: Enter/exit nearby car (you need to be next to a car to enter it) Arrow keys: movement (controls your player in 4 directions)

Find all the coins before you're out of time, otherwise you're wasted as well. Indicators of the amount of cash you already earned, coins found, and time that's left can be found in the top of your screen while playing.



```
$0 (6) -- Coins: 0/10
```

Top left of your screen: earned cash, cash highscore, found coins, total coins.

Our documentation should explain itself and our code. The documentation is devided in a few parts, as is our code. Each of them has a little description in it with additional information.

Modules

- Helpers
- Main
- ⊟ Models
  - Models.Block
  - Models.Car
  - Models.Color
  - Models.Game
  - Models.Person
  - Models.Player
  - Models.Position

Minimal requirements

Player

The player can move using the arrow keys (left, right, up, down) and enter/leave cars. While being in a car you're able to kill people by driving over them and earn cash with it.

Enemies

The cars are your enemies, they'll drive over you when they can while you're looking for all the coins. Another enemy of yours; time. You should find all the coins before you ran out of time.

Randomness

People and cars walk and drive around randomly; when a person/car needs to choose if it should go left, right or continue in its direction this is being done randomly.

Animation

Player, people, cars and coins all show their movements and states. The player and people walk, turn, and are able to die. Cars turn as well.

Pause

The player can pause the game using the P-key, which will bring you back to the startscreen.

Interaction with the file system

The world/city is loaded from a json file (configs/world.json). All the sprites are listed in a config file as well (config/sprites.txt), and the highscore is saved in another config file.

Optional requirements

Custom levels

Custom levels can be made by replacing the 'config/world.json' file with another JSON. These JSON's exist out of a few lists of objects e.g. cars, people and blocks. Each of these should furfill the required attributes as they are defined in the data-types. Good to know is that we've made our version of the in Adobe Illustrator and exported to JSON (https://github.com/bronzehedwick/Adobe-Export-Scripts). This one is converted to the exact JSON we want by a little script (ruby, recalc.rb). The AI-file of our world can be found in the config-files as well.

Complex graphics.

Graphics are loaded with the file-system from a list of sprites (configurable).

Use JSON to save information.

JSON is used for the format of our world as described above. It's implemented with the Aeson library.

Helpers

Safe Haskell	None
Language	Haskell2010

Documentation

Helpers' include several functions which are used to check if movables' can move around on a given list of blocks. To determine if moves are possible, the points of an object (given amount) should fall in a shape of the given list of objects. In this module, only 'canMove' is exported (the other functions are added here for documentation purposes).

```
canMove :: (Movable a, Movable b) => Int -> a -> [b] -> Bool                #
```

Private functions:

```
canMove' :: Movable a => Direction -> [(Float, Float)] -> a -> [Bool]       #
```

```
inObject :: Direction -> [(Float, Float)] -> [(Float, Float)] -> [Bool]     #
```

```
checkObject :: Direction -> [Float] -> (Float, Float) -> Bool               #
```

Main

Safe Haskell None
Language Haskell2010

Documentation

Main' includes all functions to draw and build the game.

The most important function, main, has a PlayIO function running which triggers everything.

```
main :: IO ()
```

Display mode for the game; fullscreen

```
window :: Display
```

Has initial values for player, time and some scores in it. The rest of the initial game state is loaded from JSON.

```
initialState :: GTA
```

Key updates:

Notices all the key changes and triggers actions.

```
handleKeys :: Event -> GTA -> IO GTA
```

Sets keystates to state, in order to know if a key is being pressed (holded).

```
updateKeyState :: (KeyState, KeyState, KeyState, KeyState, Direction) -> GTA ->
IO GTA
```

Pictures, rendering etc:

Base function from playIO to render the game (picture). Is mostly a list of other pictures, and defines the scal of the game + the focus around the player in the middle of the screen.

```
render :: GTA -> IO Picture
```

Shows the main (start) screen with information about the game.

```
mainScreen :: IO Picture
```

Function to show texts in the right scale and color.

```
text' :: (Float, Float) -> String -> Picture
```

Draws pictures that hold game state information.

```
statePicture :: GameState -> IO Picture
```

Holds the Picture with points (cash) and objectives in the top of the screen.

```
drawPoints :: GTA -> (Int, Int) -> Picture                                    #
```

Creates the timer in the top of the screen.

```
drawTimer :: GTA -> (Int, Int) -> Picture                                     #
```

Creates the text next to the timer.

```
timeLeftText :: GTA -> Picture                                                #
```

Game updates:

Base update function for game changes. The Running state triggers a few other update functions.

```
update :: Float -> GTA -> IO GTA                                              #
```

Checks if there's time left.

```
timeUp :: GTA -> GTA                                                          #
```

Decides if a car can be entered or should be left.

```
enterOrLeaveCar :: GTA -> IO GTA                                              #
```

Varies gamestates, based on current gamestates.

```
changeGameState :: Char -> GTA -> IO GTA                                      #
```

State that's being used on startup to set everything in place.

```
loading :: GTA -> IO GTA                                                      #
```

Generates a random int value, used in several decisions.

```
randomNr :: IO Int                                                            #
```

Triggers all people and car updates (movements).

```
updateTraffic :: Int -> GTA -> GTA                                            #
```

Updates coins if a coin is reached.

```
updateCoins :: Float -> GTA -> GTA                                            #
```

Player, car and people, game(state) related updates

Triggers

```
updatePlayerPosition :: GTA -> GTA                                                  #
```

Gives a subset of all blocks; all walkable/drivable for the player.

```
blocks' :: GTA -> [Block]                                                           #
```

Gives the coins' subset of blocks.

```
coins :: GTA -> [Block]                                                             #
```

Gives the subset of people, which are alive.

```
livingPeople :: GTA -> [Person]                                                     #
```

If a person is driven over, it should be killed; changes the state to a dead person.

```
killPerson :: GTA -> GTA                                                            #
```

If a coin is found, it should be removed from the coins list.

```
removeCoin :: GTA -> GTA                                                            #
```

Makes it possible to leave a car, and change back to a player.

```
leaveCar :: GTA -> GTA                                                              #
```

Enter a car.

```
enterCar :: GTA -> GTA                                                              #
```

Update all the cars (movements).

```
updateCars :: [Car] -> Int -> GTA -> GTA                                            #
```

Update single car (movement, with checks for driving over player).

```
updateCar :: GTA -> Int -> Car -> (GameState, Car)                                  #
```

Update all the people.

```
updatePeople :: [Person] -> Int -> GTA -> GTA                                       #
```

Update single person (and checks for player/cars in their way)

```
updatePerson :: GTA -> Int -> Person -> Person
```
#

```
close :: (Movable a, Movable b) => a -> [b] -> [Bool]
```
#

Models.Block

Safe Haskell	None
Language	Haskell2010

Documentation

All information needed for a block.

```
data Block                                                    #
```

Constructors

```
Block
    blockPosition :: Position
    blockWidth :: Float
    blockHeight :: Float
    blockType :: BlockType
    blockSprite :: Sprite
```

Instances

```
Show Block         #
Generic Block      #
ToJSON Block       #
FromJSON Block     #
Movable Block      #
type Rep Block     #
```

All possible block types.

```
data BlockType                                                #
```

Constructors

```
Road
Sidewalk
Building
Wall
Tree
Coin
```

Instances

```
Eq BlockType          #
Show BlockType        #
Generic BlockType     #
ToJSON BlockType      #
```

```
⊞ FromJSON BlockType | #
```

```
⊞ type Rep BlockType | #
```

Filtering all [BlockType] and returning [Block]

```
moveBlocks :: [Block] -> [BlockType] -> [Block]    | #
```

Models.Car

Safe Haskell	None
Language	Haskell2010

Documentation

All information needed for a car.

```
data Car
```

Constructors

```
Car
    carPosition :: Position
    carSprite :: Sprite
    carDirection :: Direction
    velocity :: Int
```

Instances

```
Eq Car
Show Car
Generic Car
ToJSON Car
FromJSON Car
Movable Car
type Rep Car
```

Based on a given car, the output will have a new position.

```
newCarPosition :: Car -> Car
```

Models.Color

Safe Haskell	None
Language	Haskell2010

Documentation

Convert a given string to a color.

```
stringToColor :: (IsString a, Eq a) => a -> Maybe Color
```
#

Orphan instances

```
FromJSON Color
```
#

Models.Game

Safe Haskell	None
Language	Haskell2010

Documentation

Main data-type containing all necessary constructors for the game.

```
data GTA
```

Constructors

```
Game
    player :: Player
    cars :: [Car]
    people :: [Person]
    blocks :: [Block]
    gameState :: GameState
    elapsedTime :: Float
    highscore :: Int
    timeLeft :: Float
    coinCount :: (Int, Int)
```

Instances

```
Show GTA
Generic GTA
type Rep GTA
```

Almost the same as the previous data-type, except this one is used for reading a JSON file.

```
data GTAJSON
```

Constructors

```
GameJSON
    blocksJSON :: [Block]
    peopleJSON :: [Person]
    carsJSON :: [Car]
```

Instances

```
Show GTAJSON
Generic GTAJSON
ToJSON GTAJSON
FromJSON GTAJSON
type Rep GTAJSON
```

Defines all possible states of the game. The types should be self explaining.

```
data GameState
```

Constructors

```
Loading
Running
Paused
Init
Dead
GameOver
Completed
```

Instances

```
Eq GameState
Show GameState
Generic GameState
FromJSON GameState
type Rep GameState
```

Containing a fixed path to the default JSON file.

```
jsonFile :: FilePath
```

Read a given JSON file, and retrieve its contents.

```
getJSON :: IO ByteString
```

Decode JSON data and format it into the GTAJSON data.

```
readJSON :: IO GTAJSON
```

Read the world, using previous functions, from a JSON file.

```
readWorld :: IO GTA
```

Get the highscore of a text file.

```
readHighscore :: Handle -> IO Int
```

Only write the new highscore if there is a difference, to the highscore file.

```
writeHighscore :: IO GTA -> IO GTA
```

Convert a given String to a GameState.

```
stringToGameState :: (IsString a, Eq a) => a -> Maybe GameState
```

Updates player points and also checks if there is a new highscore.

```
updatePoints :: GTA -> GTA                                                  #
```

Models.Person

Safe Haskell	None
Language	Haskell2010

Documentation

All information needed for a person.

```
data Person                                   #
```

Constructors

```
Person
    personPosition :: Position
    personSprite :: Sprite
    personDirection :: Direction
    personVelocity :: Int
```

⊟ Instances

```
⊞ Eq Person              #
⊞ Show Person            #
⊞ Generic Person         #
⊞ ToJSON Person          #
⊞ FromJSON Person        #
⊞ Movable Person         #
⊞ type Rep Person        #
```

Based on a given person, the output will have a new position.

```
newPersonPosition :: Person -> Person         #
```

Models.Player

Safe Haskell	None
Language	Haskell2010

Documentation

All information needed for the player.

```
data Player
```
#

Constructors

```
Player
    playerPosition :: Position
    keys :: Keys
    playerDirection :: Direction
    playerHeight :: Float
    playerWidth :: Float
    playerSprite :: Sprite
    playerVelocity :: Float
    playerState :: PlayerState
    points :: Int
```

Instances

```
Show Player
Movable Player
```

Defines the possible states for the player.

```
data PlayerState
```
#

Constructors

```
Walking
Driving
```

Instances

```
Eq PlayerState
Show PlayerState
Generic PlayerState
type Rep PlayerState
```

Defines the possible states.

```
data Keys
```
#

Constructors

```
Keys
    left :: KeyState
    right :: KeyState
    up :: KeyState
    down :: KeyState
```

⊟ Instances

```
⊞ Show Keys       | #
⊞ Generic Keys    | #
⊞ FromJSON Keys   | #
⊞ type Rep Keys   | #
```

Convert a given String to a KeyState.

```
stringToKeyState :: (IsString a, Eq a) => a -> Maybe KeyState    | #
```

Updates the player position and spite based on a given player and time.

```
updatePlayerPosition' :: Player -> Float -> Player    | #
```

Determines which sprite it has to display, based on a given player and time.

```
sprite :: Player -> Float -> Sprite    | #
```

Based on pressed keys, the next position of the player will be determined.

```
newPosition :: Keys -> Position -> (Position, Float)    | #
```

Converts a player back from a car to a player.

```
carToPlayer :: Player -> Player    | #
```

Converts a player to a car.

```
playerToCar :: Player -> Car -> Player    | #
```

Change the sprite for player when the player died.

```
killPlayer :: Player -> Player    | #
```

Orphan instances

```
⊞ FromJSON KeyState    | #
```

Models.Position

Safe Haskell	None
Language	Haskell2010

Documentation

Default data-type for a position.

```
data Position #
```

Constructors

```
Position
    x :: Float
    y :: Float
```

Instances

```
Eq Position #
Show Position #
Generic Position #
ToJSON Position #
FromJSON Position #
type Rep Position #
```

Data-type for the Sprite, where spriteType links an object and spriteState to the state.

```
data Sprite #
```

Constructors

```
Sprite
    spriteType :: String
    spriteState :: Int
```

Instances

```
Eq Sprite #
Show Sprite #
Generic Sprite #
ToJSON Sprite #
FromJSON Sprite #
type Rep Sprite #
```

Data-type for all possible directions.

```
data Direction #
```

Constructors

```
North
East
South
West
```

⊟ Instances

```
⊞ Enum Direction        #
⊞ Eq Direction          #
⊞ Show Direction        #
⊞ Generic Direction     #
⊞ ToJSON Direction      #
⊞ FromJSON Direction    #
⊞ type Rep Direction    #
```

Class Movable for Player, Person and Car to do something with the position and/or sprite.

```
class Movable a where                                    #
```

Minimal complete definition

```
getPos, setPos, getDir, setDir, width, height, getSprite
```

Methods

```
getPos :: a -> Position                                  #
```

```
setPos :: Position -> a -> a                             #
```

```
getDir :: a -> Direction                                 #
```

```
setDir :: Direction -> a -> a                            #
```

```
width :: a -> Float                                      #
```

```
height :: a -> Float                                     #
```

```
getSprite :: a -> Sprite                                 #
```

⊟ Instances

```
⊞ Movable Person    #
⊞ Movable Car       #
⊞ Movable Block     #
⊞ Movable Player    #
```

Draws the picture based on given arguments.

```
draw :: Movable a => [(String, Picture)] -> a -> Picture             #
```

Based on a given positions, moves an 'a'.

```
move :: Movable a => Position -> a -> a
```
#

Retrieves the coordinates of a given 'a'.

```
coordinates :: Movable a => a -> [(Float, Float)]
```
#

Gives the next direction, based on a given direction.

```
next :: Direction -> Direction
```
#

Gives the previous direction, based on a given direction.

```
prev :: Direction -> Direction
```
#

Changes the directions based on the given arguments.

```
changeDir :: Movable a => Int -> a -> a
```
#

Self defined function to round numbers.

```
roundDecimals :: (Fractional a2, RealFrac a1, Integral b) => a1 -> b -> a2
```
#

Gives the next sprite state, based on a given sprite.

```
nextSprite :: Sprite -> Int
```
#

Changes the direction randomly.

```
changeDirR :: Movable a => Int -> a -> a
```
#