Если вы видите что-то необычное, просто сообщите мне.

Rendering with Gloss

Welcome to the final part of our Open AI Gym series! Throughout this series, we've explored some of the ideas in the Open AI Gym framework. We made a couple games, generalized them, and applied some machine learning techniques. When it comes to rendering our games though, we're still relying on a very basic command line text format.

But if we want to design agents for more visually appealing games, we'll need a better solution! On Monday Moring Haskell, we've spent quite a bit of time with the Gloss library. This library makes it easy to create simple games and render them using OpenGL. Take a look at this article for a summary of our work there and some links to the basics. You can also see the code for this article on the gloss branch of our Github repository.

In this article, we'll explore how we can draw some connections between Gloss and our Open AI Gym work. We'll see how we can take the functions we've already written and use them within Gloss!

GLOSS BASICS

The key entrypoint for a Gloss game is the play function. At its core is the world type parameter, which we'll define for ourselves later.

```
play :: Display -> Color -> Int
  -> world
  -> (world -> Picture)
  -> (Event -> world -> world)
  -> (Float -> world -> world)
  -> IO ()
```

We won't go into the first three parameters. But the rest are important. The first is our initial world state. The second is our rendering function. It creates a Picture for the current state. Then comes

an "event handler". This takes user input events and updates the world based on the actions. Finally there is the update function. This changes the world based on the passage of time, rather than specific user inputs.

This structure should sound familiar, because it's a lot like our Open AI environments! The initial world is like the "reset" function. Then both systems have a "render" function. And the update functions are like our stepEnv function.

The main difference we'll see is that Gloss's functions work in a pure way. Recall our "environment" functions use the "State" monad. Let's explore this some more.

RE-WRITING ENVIRONMENT FUNCTIONS

Let's take a look at the basic form of these environment functions, in the Frozen Lake context:

```
resetEnv :: (Monad m) => StateT FrozenLakeEnvironment m Observation
stepEnv :: (Monad m) =>
  Action -> StateT FrozenLakeEnvironment m (Observation, Double, Bool)
renderEnv :: (MonadIO m) => StateT FrozenLakeEnvironment m ()
```

These all use State. This makes it easy to chain them together. But if we look at the implementations, a lot of them don't really need to use State. They tend to unwrap the environment at the start with get, calculate new results, and then have a final put call.

This means we can rewrite them to fit more within Gloss's pure structure! We'll ignore rendering, since that will be very different. But here are some alternate type signatures:

resetEnv' :: FrozenLakeEnvironment -> FrozenLakeEnvironment stepEnv' :: Action -> FrozenLakeEnvironment -> (FrozenLakeEnvironment, Double, Bool) We'll exclude Observation as an output, since the environment contains that through currentObservation. The implementation for each of these looks like the original. Here's what resetting looks like:

```
resetEnv' :: FrozenLakeEnvironment -> FrozenLakeEnvironment
resetEnv' fle = fle
  { currentObservation = 0
  , previousAction = Nothing
  }
```

Now for stepping our environment forward:

```
stepEnv' :: Action -> FrozenLakeEnvironment -> (FrozenLakeEnvironment, Double, Bool)
stepEnv' act fle = (finalEnv, reward, done)
  where
    currentObs = currentObservation fle
    (slipRoll, gen') = randomR (0.0, 1.0) (randomGenerator fle)
    allLegalMoves = legalMoves currentObs (dimens fle)
    numMoves = length allLegalMoves - 1
    (randomMoveIndex, finalGen) = randomR (0, numMoves) gen'
    newObservation = ... -- Random move, or apply the action
    (done, reward) = case (grid fle) A.! newObservation of
      Goal -> (True, 1.0)
      Hole -> (True, 0.0)
      _ -> (False, 0.0)
    finalEnv = fle
      { currentObservation = newObservation
      , randomGenerator = finalGen
      , previousAction = Just act
      }
```

What's even better is that we can now rewrite our original State functions using these!

```
resetEnv :: (Monad m) => StateT FrozenLakeEnvironment m Observation
resetEnv = do
  modify resetEnv'
  gets currentObservation

stepEnv :: (Monad m) =>
  Action -> StateT FrozenLakeEnvironment m (Observation, Double, Bool)
stepEnv act = do
  fle <- get
  let (finalEnv, reward, done) = stepEnv' act fle
```

```
  put finalEnv
  return (currentObservation finalEnv, reward, done)
```

IMPLEMENTING GLOSS

Now let's see how this ties in with Gloss. It might be tempting to use our Environment as the world type. But it can be useful to attach other information as well. For one example, we can also include the current GameResult, telling us if we've won, lost, or if the game is still going.

```
data GameResult =
  GameInProgress |
  GameWon |
  GameLost
  deriving (Show, Eq)

data World = World
  { environment :: FrozenLakeEnvironment
  , gameResult :: GameResult
  }
```

Now we can start building the other pieces of our game. There aren't really any "time" updates in our game, except to update the result based on our location:

```
updateWorldTime :: Float -> World -> World
updateWorldTime _ w = case tile of
  Goal -> World fle GameWon
  Hole -> World fle GameLost
  _ -> w
  where
    fle = environment w
    obs = currentObservation fle
    tile = grid fle A.! obs
```

When it comes to handling inputs, we need to start with the case of restarting the game. When the game isn't InProgress, only the "enter" button matters. This resets everything, using resetEnv':

```
handleInputs :: Event -> World -> World
handleInputs event w
  | gameResult w /= GameInProgress = case event of
      (EventKey (SpecialKey KeyEnter) Down _ _) ->
        World (resetEnv' fle) GameInProgress
      _ -> w
  ...
```

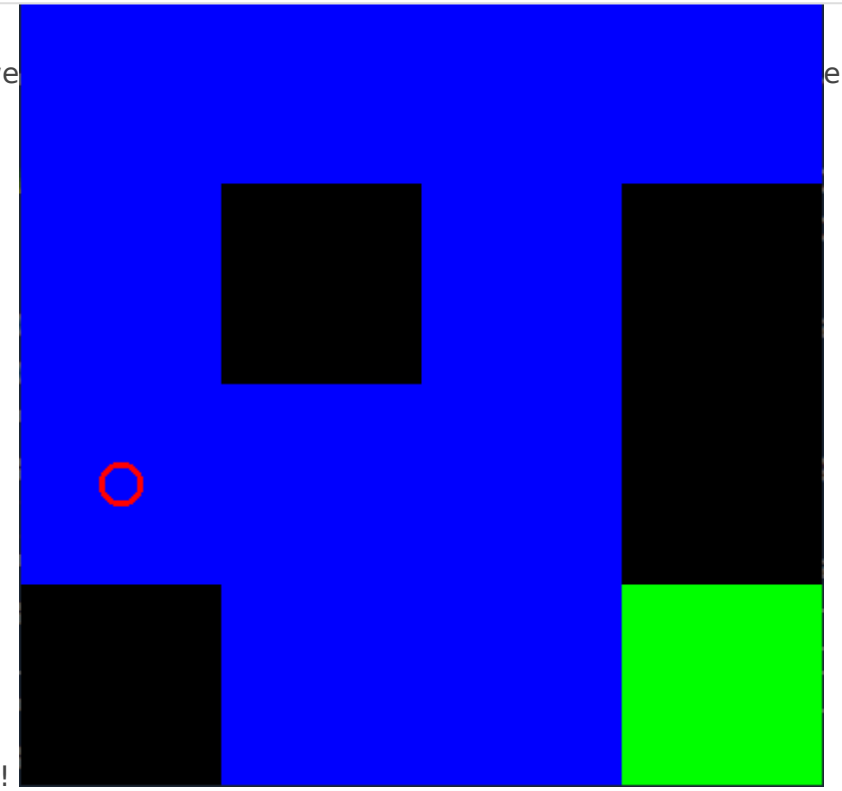
Now we handle each directional input key. We'll make a helper function at the bottom that does the business of calling stepEnv'.

```
handleInputs :: Event -> World -> World
handleInputs event w
  | gameResult w /= GameInProgress = case event of
      (EventKey (SpecialKey KeyEnter) Down _ _) ->
        World (resetEnv' fle) GameInProgress
  | otherwise = case event of
      (EventKey (SpecialKey KeyUp) Down _ _) ->
        w {environment = finalEnv MoveUp }
      (EventKey (SpecialKey KeyRight) Down _ _) ->
        w {environment = finalEnv MoveRight }
      (EventKey (SpecialKey KeyDown) Down _ _) ->
        w {environment = finalEnv MoveDown }
      (EventKey (SpecialKey KeyLeft) Down _ _) ->
        w {environment = finalEnv MoveLeft }
      _ -> w
  where
    fle = environment w
    finalEnv action =
      let (fe, _, _) = stepEnv' action fle
      in  fe
```

The last step is rendering the environment with a draw function. This just requires a working knowledge of constructing the Picture type in Gloss. It's a little tedious, so I won't belabor the details. Check the Github links at the bottom if you're curious!

We can then combine all these pieces like so:

```
main :: IO ()
main = do
  env <- basicEnv
  play windowDisplay white 20
    (World env GameInProgress)
    drawEnvironment
    handleInputs
    updateWorldTime
```

After we have all these pieces, we  e green tile while avoiding the black tiles!

CONCLUSION

With a little more plumbing, it would be possible to combine this with the rest of our "Environment" work. There are some definite challenges. Our current environment setup doesn't have a "time update" function. Combining machine learning with Gloss rendering would also be interesting.

This is the end of the Open AI Gym series, but take a look at our Github repository to see all the code we wrote in this series! The code for this article is on the gloss branch, particularly in FrozenLakeGloss.hs, with some modifications to FrozenLakeBasic.hs. If you liked this series, don't

forget to Subscribe to Monday Morning Haskell to get our monthly newsletter!

Revision #1

Created 2022-03-11 06:56:13 UTC by gasick

Updated 2022-03-11 17:11:16 UTC by gasick