Multiplayer Game from Scratch

Nickolay Kudasov

Haskell eXchange 2017, London

Disclaimer

- I am CTO at GETSHOP.TV TV Commerce technology powered by Haskell
- I teach Haskell at Moscow State University
- I contribute a bit (swagger2, http-api-data, servant)

Disclaimer

- I am CTO at GETSHOP.TV TV Commerce technology powered by Haskell
- I teach Haskell at Moscow State University
- I contribute a bit (swagger2, http-api-data, servant)

I am NOT a game developer!

Why make games?

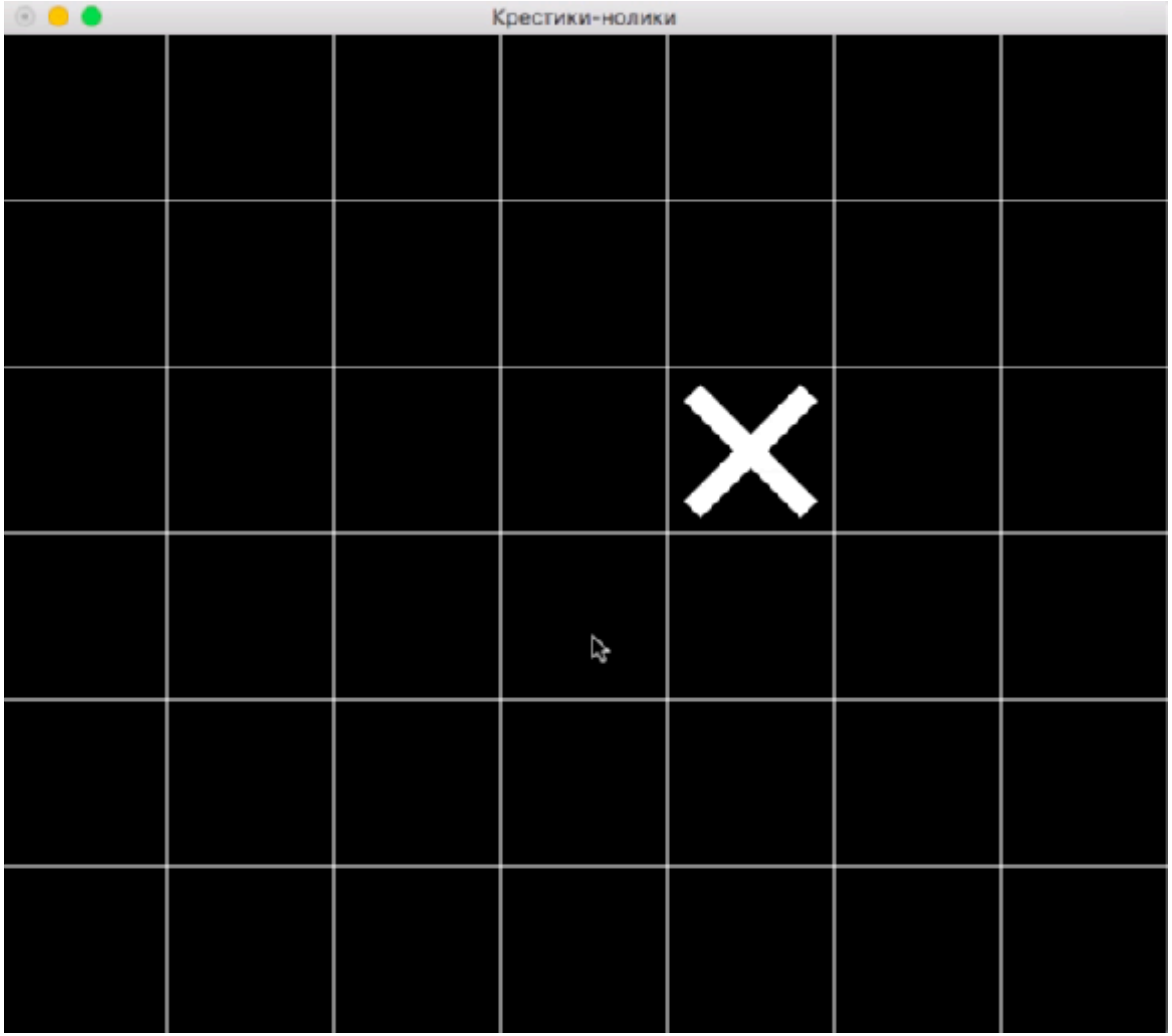
Games are fun

- Games are fun **to play**
- Games are fun **to make**
- Games are fun **to share**

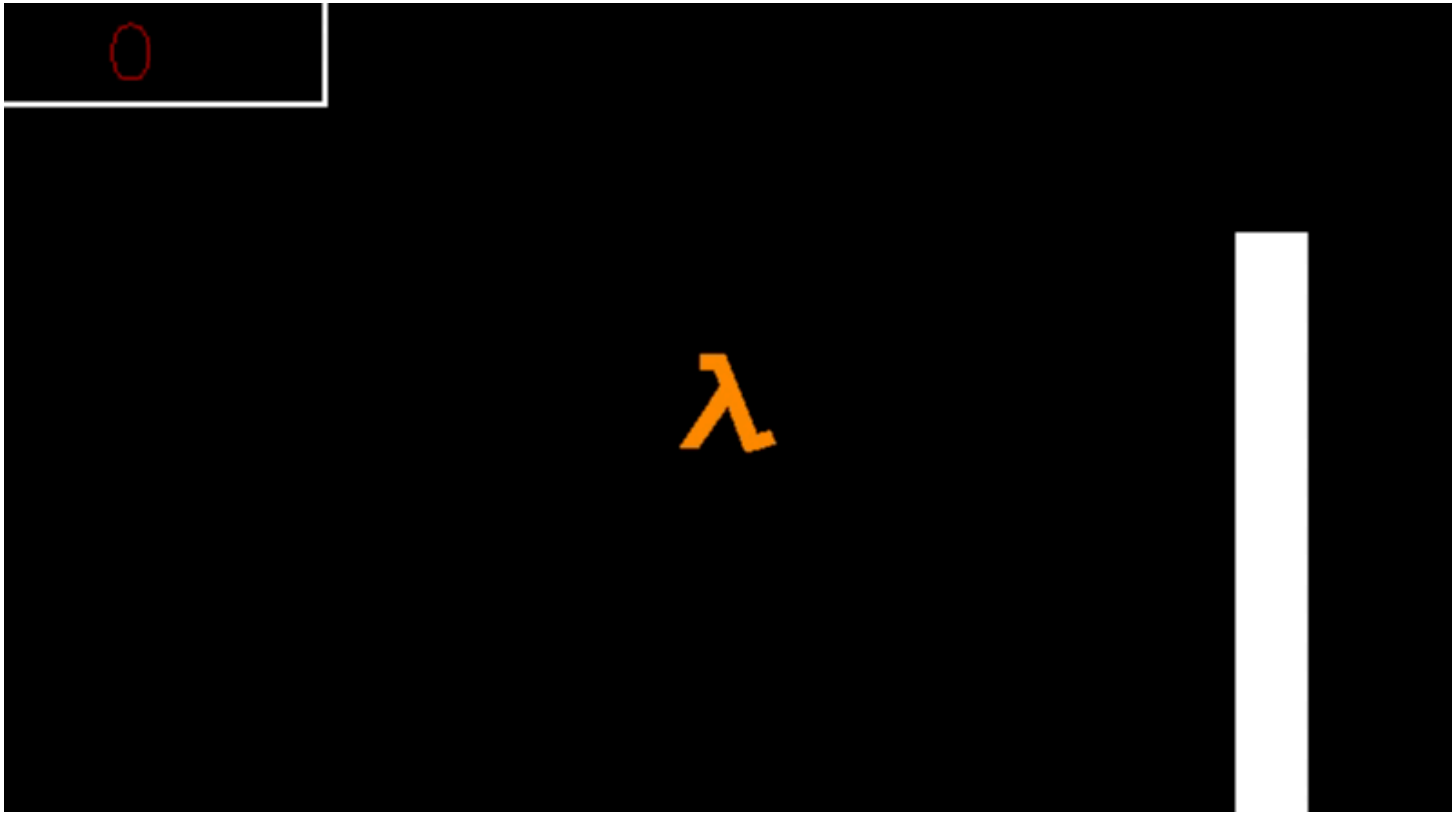
Making games is a way to learn

- 2D and 3D graphics
- Animation
- Physics
- Logic
- Game AI
- Network
- and more

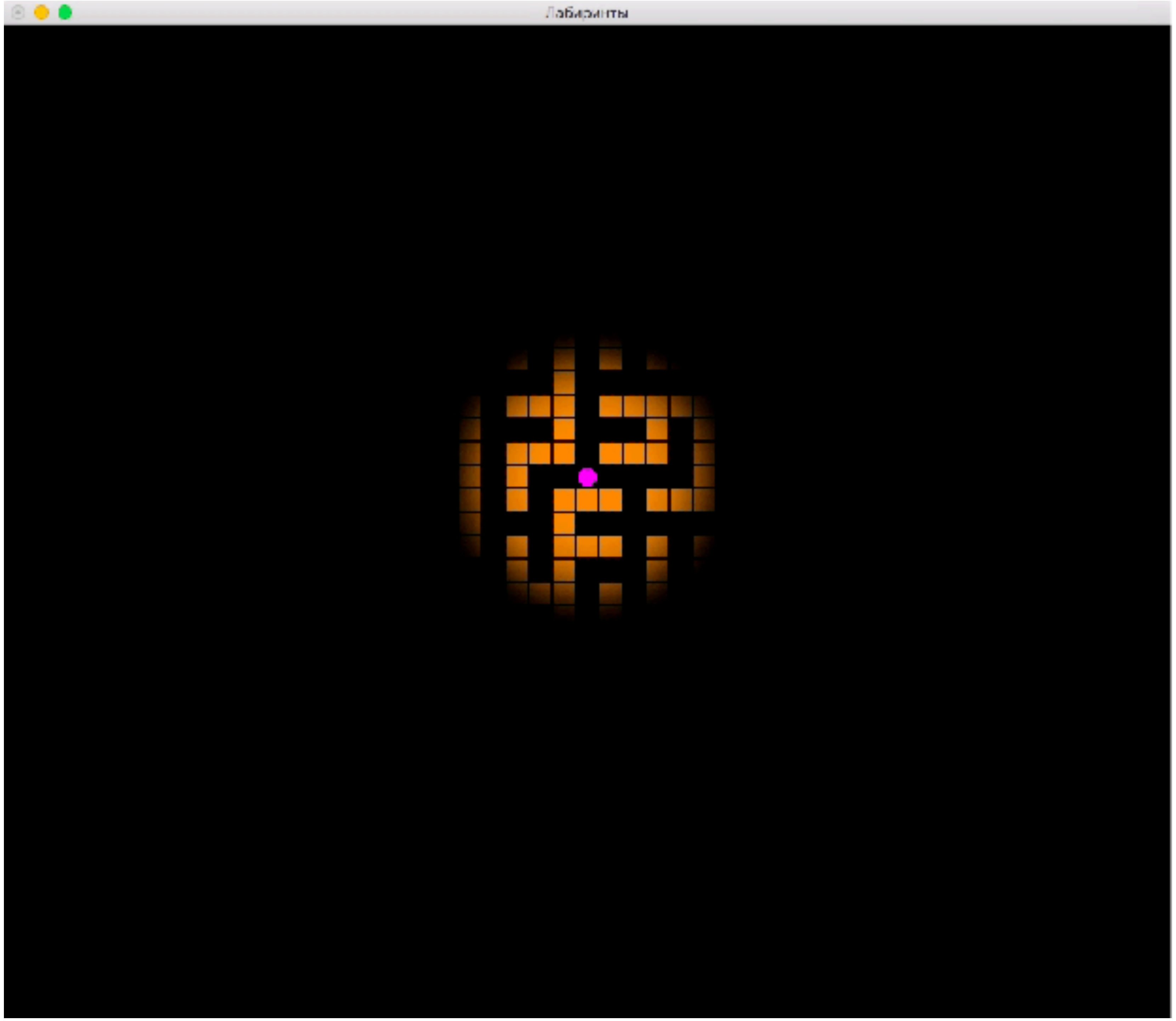
Simple Games: Tic-Tac-Toe



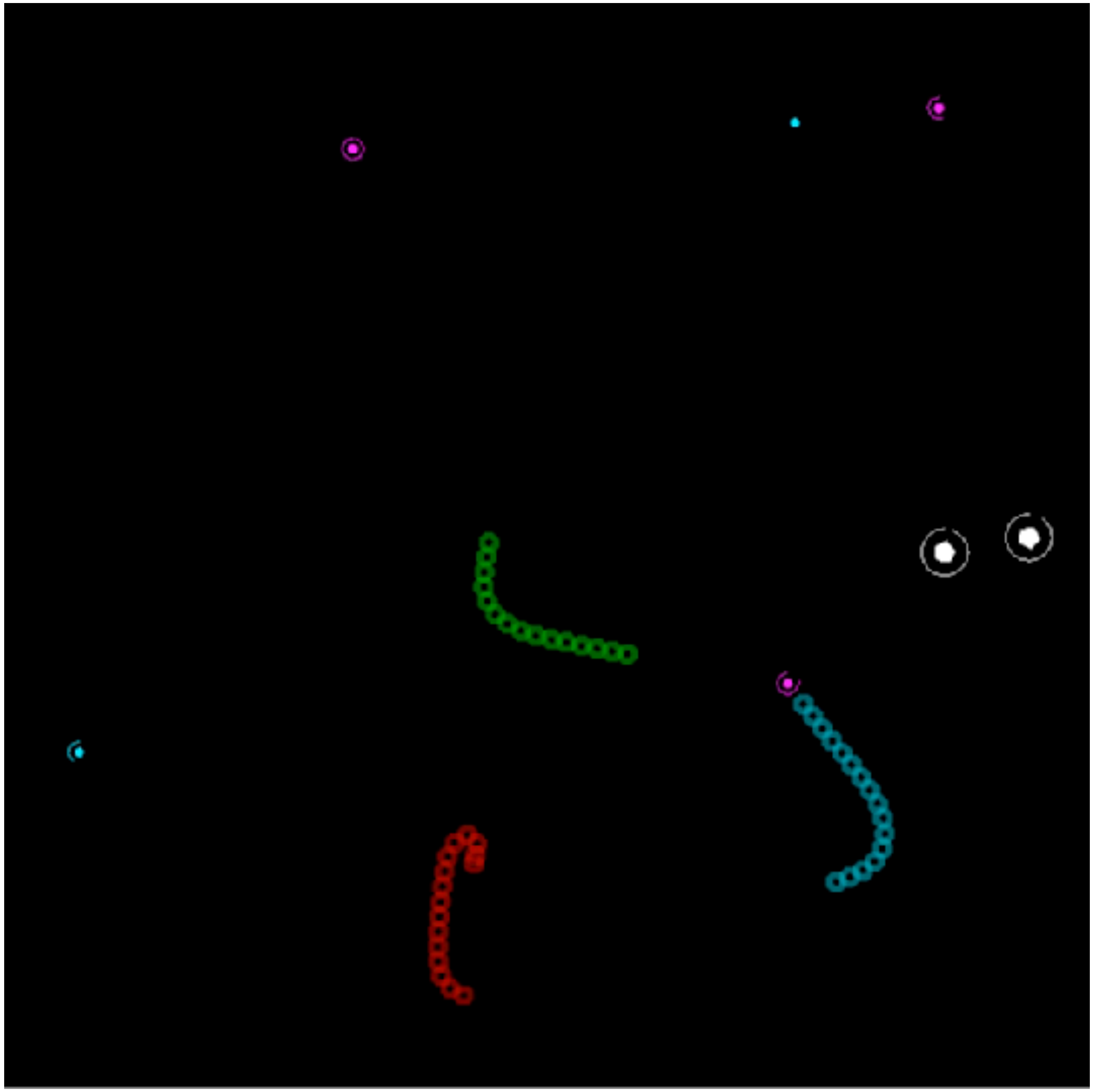
Simple Games: Flappy Lambda



Simple Games: Maze



Simple Games: Snakes

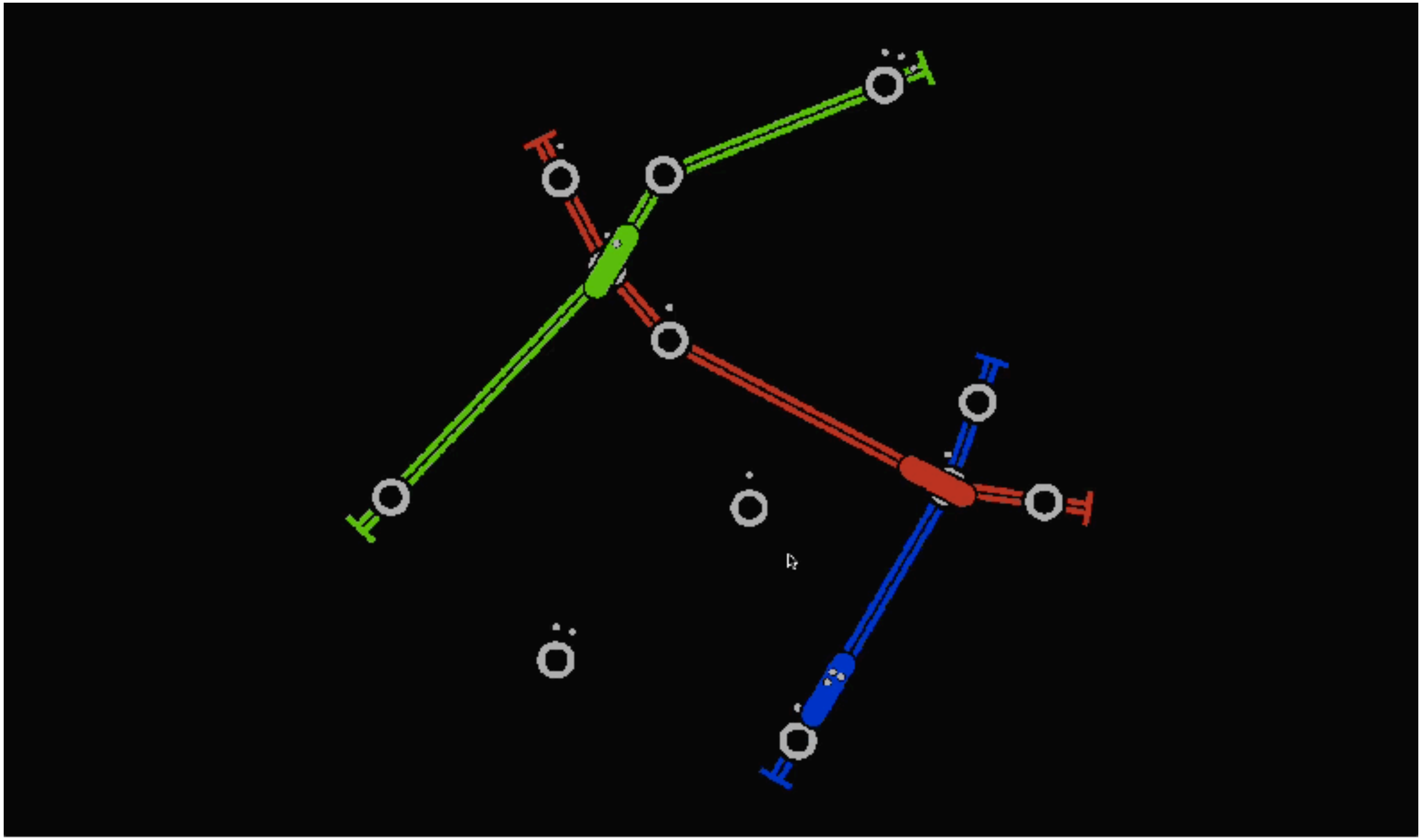


Trains

~~Trains~~

Tubes

Tubes



Empty project

Initialising a project

```
$ stack new tubes new-template --resolver nightly-2017-09-29
Downloading template "new-template" to create project "tubes" in tubes/ ...
...
All done.
```

Initialising a project

```
$ stack new tubes new-template --resolver nightly-2017-09-29
Downloading template "new-template" to create project "tubes" in tubes/ ...
...
All done.

$ tree tubes
snakes
├── LICENSE
├── Setup.hs
├── app
│   └── Main.hs
├── tubes.cabal
├── src
│   └── Lib.hs
├── stack.yaml
└── test
    └── Spec.hs

3 directories, 7 files
```

Initialising a project

```
$ stack new tubes new-template --resolver nightly-2017-09-29
Downloading template "new-template" to create project "tubes" in tubes/ ...
...
All done.

$ tree tubes
snakes
├── LICENSE
├── Setup.hs
├── app
│   └── Main.hs
├── tubes.cabal
├── src
│   └── Lib.hs
├── stack.yaml
└── test
    └── Spec.hs

3 directories, 7 files
```

`src/`

Library source code.
Later — common client and server logic.

Initialising a project

```
$ stack new tubes new-template --resolver nightly-2017-09-29
Downloading template "new-template" to create project "tubes" in tubes/ ...
...
All done.

$ tree tubes
snakes
├── LICENSE
├── Setup.hs
├── app
│   └── Main.hs
├── tubes.cabal
├── src
│   └── Lib.hs
├── stack.yaml
└── test
    └── Spec.hs

3 directories, 7 files
```

app/

Executable sources.
Client, server and single player
will be different executables.

Initialising a project

```
$ stack new tubes new-template --resolver nightly-2017-09-29
Downloading template "new-template" to create project "tubes" in tubes/ ...
...
All done.

$ tree tubes
snakes
├── LICENSE
├── Setup.hs
├── app
│   └── Main.hs
├── tubes.cabal
├── src
│   └── Lib.hs
├── stack.yaml
└── test
    └── Spec.hs

3 directories, 7 files
```

`test/`

Tests.
Important stuff.
But we are going to ignore it today.

Initialising a project

```
$ stack new tubes new-template --resolver nightly-2017-09-29
Downloading template "new-template" to create project "tubes" in tubes/ ...
...
All done.

$ tree tubes
snakes
├── LICENSE
├── Setup.hs
├── app
│   └── Main.hs
├── tubes.cabal
├── src
│   └── Lib.hs
├── stack.yaml
└── test
    └── Spec.hs

3 directories, 7 files
```

`Setup.hs`

Build instructions.
Default one is fine for us.

Initialising a project

```
$ stack new tubes new-template --resolver nightly-2017-09-29
Downloading template "new-template" to create project "tubes" in tubes/ ...
...
All done.

$ tree tubes
snakes
├── LICENSE
├── Setup.hs
├── app
│   └── Main.hs
├── tubes.cabal
├── src
│   └── Lib.hs
├── stack.yaml
└── test
    └── Spec.hs

3 directories, 7 files
```

`tubes.cabal`

Cabal package configuration
(library, executables, dependencies, etc.).

Initialising a project

```
$ stack new tubes new-template --resolver nightly-2017-09-29
Downloading template "new-template" to create project "tubes" in tubes/ ...
...
All done.

$ tree tubes
snakes
├── LICENSE
├── Setup.hs
├── app
│   └── Main.hs
├── tubes.cabal
├── src
│   └── Lib.hs
├── stack.yaml
└── test
    └── Spec.hs

3 directories, 7 files
```

`stack.yaml`

Stack project configuration.
Specifies a snapshot with compiler
and dependency versions.

Build and run

```
$ cd tubes
$ stack build
tubes-0.1.0.0: configure
Configuring tubes-0.1.0.0...
tubes-0.1.0.0: build
...
Registering tubes-0.1.0.0…

$ stack exec tubes-exe
someFunc
```

gloss

gloss: Painless 2D vector graphics

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

gloss: Painless 2D vector graphics

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

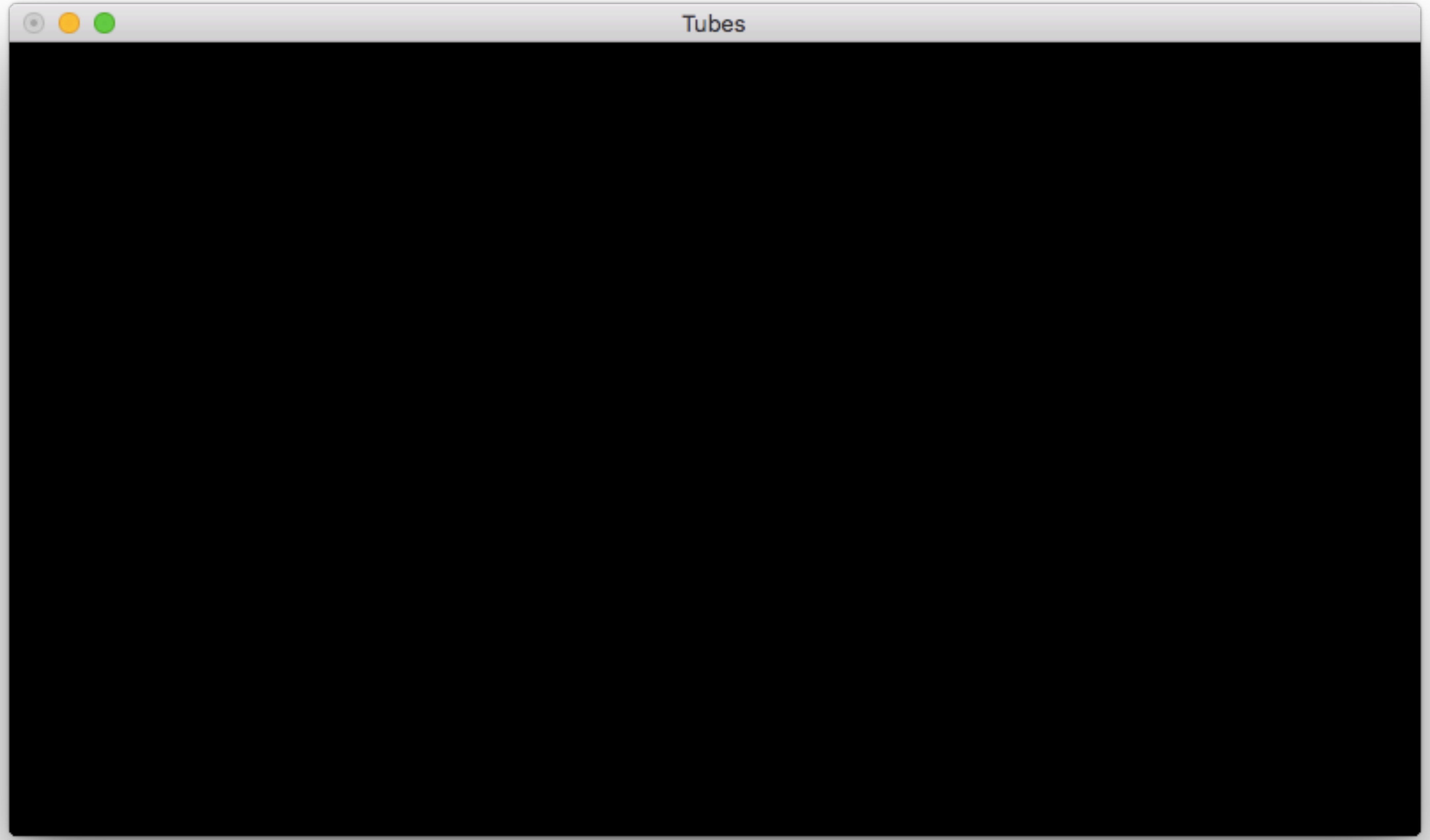
gloss: Painless 2D vector graphics

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

Black Screen: Main.hs

```
module Main where

import Graphics.Gloss.Interface.Pure.Game

main :: IO ()
main =
  play display bgColor fps initialWorld renderWorld handleWorld updateWorld
  where
    display = InWindow "Tubes" winSize winOffset
    bgColor = black
    fps     = 60

    initialWorld = ()
    renderWorld w = blank
    handleWorld _ w = w
    updateWorld _ w = w

    winSize = (800, 450)
    winOffset = (100, 100)
```

Black Screen



Lonely Train

Project structure

```
.
├── app
│   └── Main.hs
└── src
    ├── Tubes
    │   ├── Config.hs
    │   ├── Control.hs
    │   ├── Model.hs
    │   └── Render.hs
    └── Tubes.hs
```

Tubes.Model

Core types and functions.

Project structure

```
.
├── app
│   └── Main.hs
└── src
    ├── Tubes
    │   ├── Config.hs
    │   ├── Control.hs
    │   ├── Model.hs
    │   └── Render.hs
    └── Tubes.hs
```

`Tubes.Render`

Rendering functions.

Project structure

```
.
├── app
│   └── Main.hs
└── src
    ├── Tubes
    │   ├── Config.hs
    │   ├── Control.hs
    │   ├── Model.hs
    │   └── Render.hs
    └── Tubes.hs
```

`Tubes.Control`

Handling user input
(keyboard, mouse, etc.).

Project structure

```
.
├── app
│   └── Main.hs
└── src
    ├── Tubes
    │   ├── Config.hs
    │   ├── Control.hs
    │   ├── Model.hs
    │   └── Render.hs
    └── Tubes.hs
```

`Tubes.Config`

Game constants and parameters.
Like train size, speed and color.

Model: Types

```haskell
-- | A segment is a straight track that connects two points.
data Segment = Segment
  { segmentStart  :: Point    -- ^ Segment starting point.
  , segmentEnd    :: Point    -- ^ Segment end point.
  }
```

Model: Types

```haskell
-- | A segment is a straight track that connects two points.
data Segment = Segment
  { segmentStart  :: Point    -- ^ Segment starting point.
  , segmentEnd    :: Point    -- ^ Segment end point.
  }

-- | A train moving along a segment.
data Train = Train
  { -- | Rail segment the train is on.
    trainSegment      :: Segment
  , -- | Time spent on this segment (in seconds).
    trainProgress     :: Float
  , -- | Train location on the current segment (from start).
    trainLocation     :: Float
  }
```

Model: Functions

```haskell
-- | Move a train back and forth along its segment.
moveTrainBackAndForth :: Float -> Train -> Train
```

Model: Functions

```
-- | Move a train back and forth along its segment.
moveTrainBackAndForth :: Float -> Train -> Train

-- | Move a train along its segment.
-- Leftover time is returned as a second result.
moveTrain :: Float -> Train -> (Train, Maybe Float)
```

Model: Functions

```
-- | Move a train back and forth along its segment.
moveTrainBackAndForth :: Float -> Train -> Train

-- | Move a train along its segment.
-- Leftover time is returned as a second result.
moveTrain :: Float -> Train -> (Train, Maybe Float)

-- | Compute train location on a linear track.
-- Leftover time is returned as a second result.
trainTrackLocation
  :: Float                  -- ^ Linear track length.
  -> Float                  -- ^ Time since start (in seconds).
  -> (Float, Maybe Float)
```

Model: Functions

```
-- | Move a train back and forth along its segment.
moveTrainBackAndForth :: Float -> Train -> Train

-- | Move a train along its segment.
-- Leftover time is returned as a second result.
moveTrain :: Float -> Train -> (Train, Maybe Float)

-- | Compute train location on a linear track.
-- Leftover time is returned as a second result.
trainTrackLocation
  :: Float                  -- ^ Linear track length.
  -> Float                  -- ^ Time since start (in seconds).
  -> (Float, Maybe Float)

-- | Initialise a train at the start of a segment.
initTrain :: Segment -> Train
```

Render: Train

```haskell
-- | Render a train.
renderTrain :: Color -> Train -> Picture
```

Render: Train

```haskell
-- | Render a train.
renderTrain :: Color -> Train -> Picture
renderTrain trainColor train
  = renderLocomotive trainColor
  & rotate (- theta * 180 / pi)
  & translate x y
  where
    (x, y) = trainPosition train
    theta = trainOrientation train
```

Render: Train

```
-- | Render a train.
renderTrain :: Color -> Train -> Picture
renderTrain trainColor train
  = renderLocomotive trainColor
  & rotate (- theta * 180 / pi)
  & translate x y
  where
    (x, y) = trainPosition train
    theta = trainOrientation train

-- | Render train's locomotive at the origin.
renderLocomotive :: Color -> Picture
```

Render: Train

```haskell
-- | Render a train.
renderTrain :: Color -> Train -> Picture
renderTrain trainColor train
  = renderLocomotive trainColor
  & rotate (- theta * 180 / pi)
  & translate x y
  where
    (x, y) = trainPosition train
    theta = trainOrientation train

-- | Render train's locomotive at the origin.
renderLocomotive :: Color -> Picture

-- | Compute actual train position.
trainPosition :: Train -> Point

-- | Compute train orientation (angle in radians).
trainOrientation :: Train -> Float
```

Render: Tracks

```
-- | Render tracks for a segment.
renderSegment :: Color -> Segment -> Picture
```

Render: Tracks

```haskell
-- | Render tracks for a segment.
renderSegment :: Color -> Segment -> Picture
renderSegment segmentColor s
  = foldMap polygon [ leftRail, rightRail , start, end ]
  & rotate (- theta * 180 / pi)
  & translate x y
  & color segmentColor
  where
    ...
```

Render: Tracks

```haskell
-- | Render tracks for a segment.
renderSegment :: Color -> Segment -> Picture
renderSegment segmentColor s
  = foldMap polygon [ leftRail, rightRail , start, end ]
  & rotate (- theta * 180 / pi)
  & translate x y
  & color segmentColor
  where
    ...

-- | Render a train together with a segment
-- it is moving along.
renderTrainWithSegment :: Train -> Picture
renderTrainWithSegment train
  =  renderTrain defaultTrainColor train
  <> renderSegment defaultSegmentColor (trainSegment train)
```

Control

```
-- | Set the end of a train segment.
setTrainSegmentEnd :: Point -> Train -> Train
setTrainSegmentEnd point train = train
  { trainSegment = Segment from point }
  where
    Segment from _ = trainSegment train
```

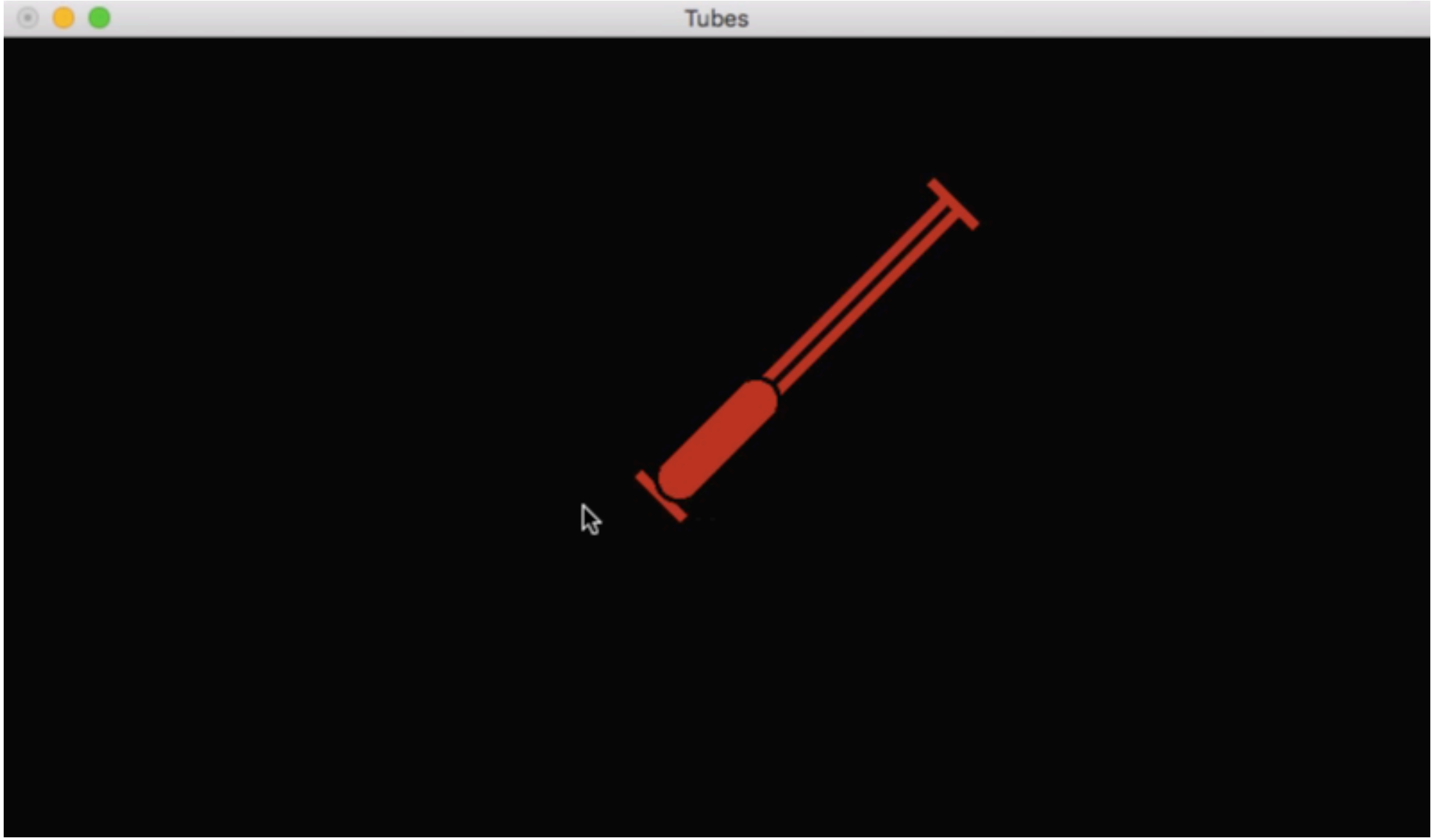
Config

```haskell
-- | Train acceleration (pixels per second squared).
trainAcceleration :: Float

-- | Train maximum speed (pixels per second).
trainMaxSpeed :: Float

-- | Train width.
trainWidth :: Float

-- | Train length (without bumps).
trainLength :: Float

-- | Track width (two rails with a gap).
trackWidth :: Float

-- | Single rail width.
railWidth :: Float

-- | An extra part of a segment to give a train more space.
endTrackLength :: Float

-- | Background color. Also used for outlining.
backgroundColor :: Color
```

Bringing it all together

```
initialWorld = initTrain (Segment (0, 0) (100, 100))
renderWorld  = renderTrainWithSegment
updateWorld  = moveTrainBackAndForth

handleWorld (EventMotion mouse)
  = setTrainSegmentEnd mouse
handleWorld _ = id
```

Lonely Train

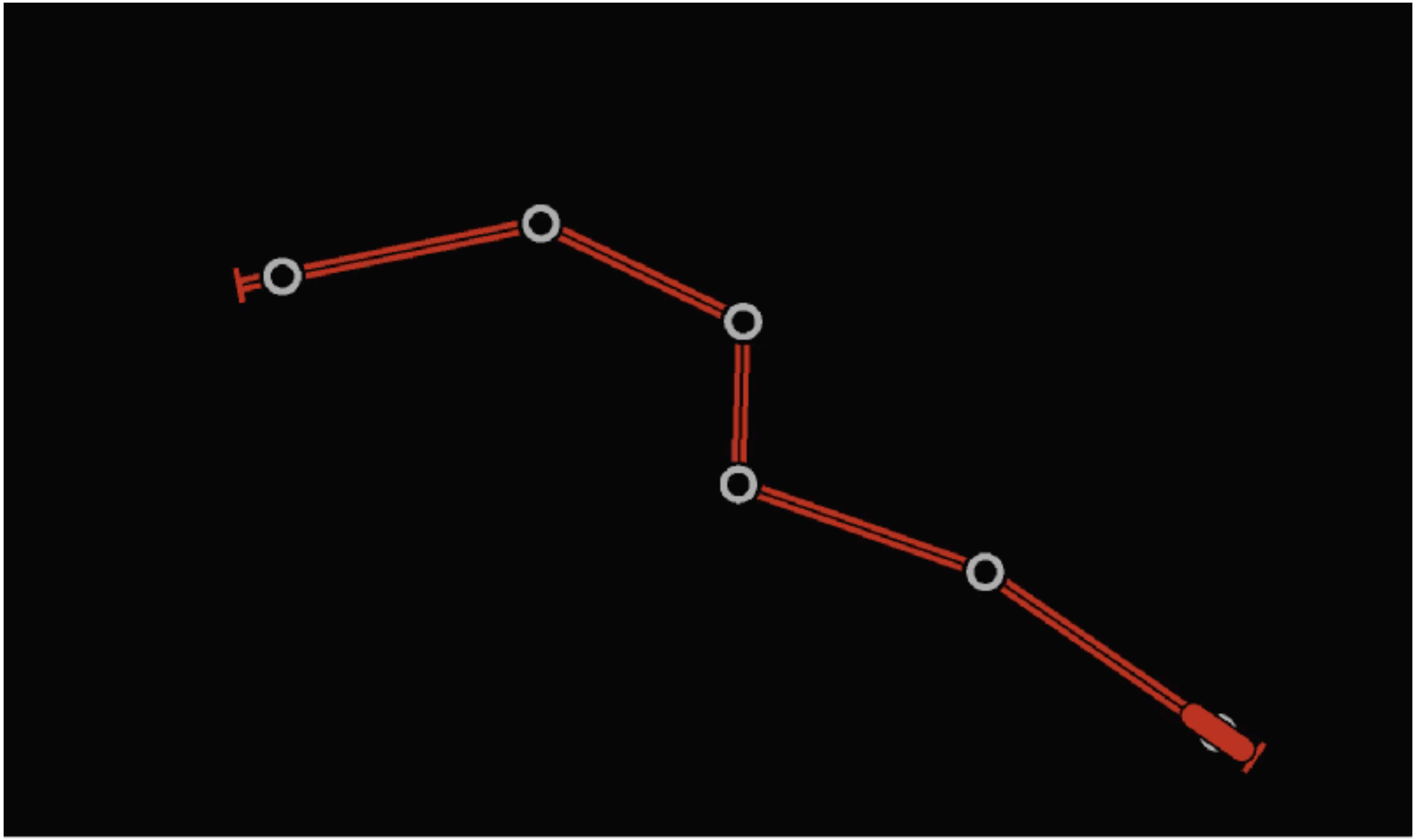


Tube Line System

Tube Line

```
-- | A line with tracks and trains.
data TubeLine = TubeLine
  { -- | Segments of which a line consists.
    tubeLineSegments  :: [Segment]
  , -- | Trains on the line.
    tubeLineTrains    :: [Train]
  }

-- | Update all trains on the line.
updateTubeLineTrains :: Float -> TubeLine -> TubeLine

-- | Append a new segment to the end of the line.
appendTubeLineSegment :: Point -> TubeLine -> TubeLine

-- | Prepend a new segment to the beginning of the line.
prependTubeLineSegment :: Point -> TubeLine -> TubeLine
```

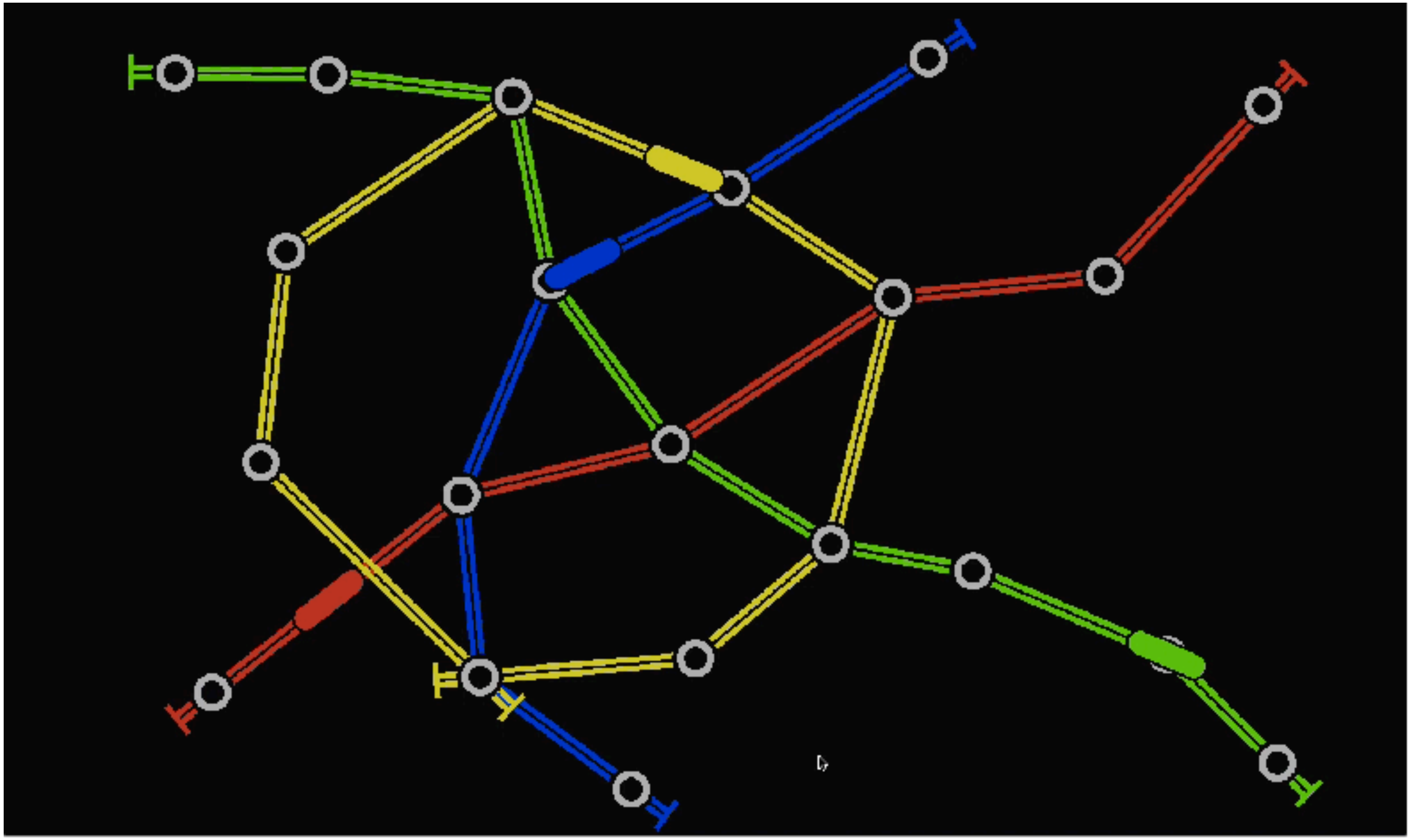
Tube Line



Tube Line System

```
-- | Tube system consisting of multiple lines and stations.
data Tube = Tube
  { tubeLines     :: [TubeLine]   -- ^ Lines.
  , tubeStations  :: [Station]    -- ^ Stations.
  }

-- | A station.
data Station = Station
  { stationLocation :: Point  -- ^ Location of the station.
  }

-- | Update everything in a tube system.
updateTube :: Float -> Tube -> Tube
```

Tube Line System



Tube Line Construction

```
-- | A tube line construction action.
data Action f
  = StartNewLine Point (f Point)
  | ContinueLine TubeLineId TubeLineDirection Point (f Point)

data Missing a = Missing

newtype Present a = Present a

-- | An incomplete action (drag).
type IncompleteAction = Action Missing

-- | A complete action (drop).
type CompleteAction = Action Present
```

Tube Line Construction

```
-- | Start a construction action at a given point
-- (start a drag).
startAction :: Point -> Tube -> Maybe IncompleteAction

-- | Complete a construction action at a given point
-- (perform a drop).
completeAction
  :: Point -> IncompleteAction -> Tube -> Maybe CompleteAction

-- | Apply a construction action to a tube line system.
applyCompleteAction :: CompleteAction -> Tube -> Tube
```

Tube Line Construction

```haskell
-- | Handle user input to construct the tube line system.
handleTubeConstruction
  :: Event
  -> (Tube, Maybe IncompleteAction)
  -> (Tube, Maybe IncompleteAction)
handleTubeConstruction
  (EventKey (MouseButton LeftButton) Down _ point)
  (tube, _)
  = (tube, startAction point tube)
handleTubeConstruction
  (EventKey (MouseButton LeftButton) Up   _ point)
  (tube, Just ia)
  = case completeAction point ia tube of
      Just ca -> (applyCompleteAction ca tube, Nothing)
      _ -> (tube, Nothing)
handleTubeConstruction _ t = t
```

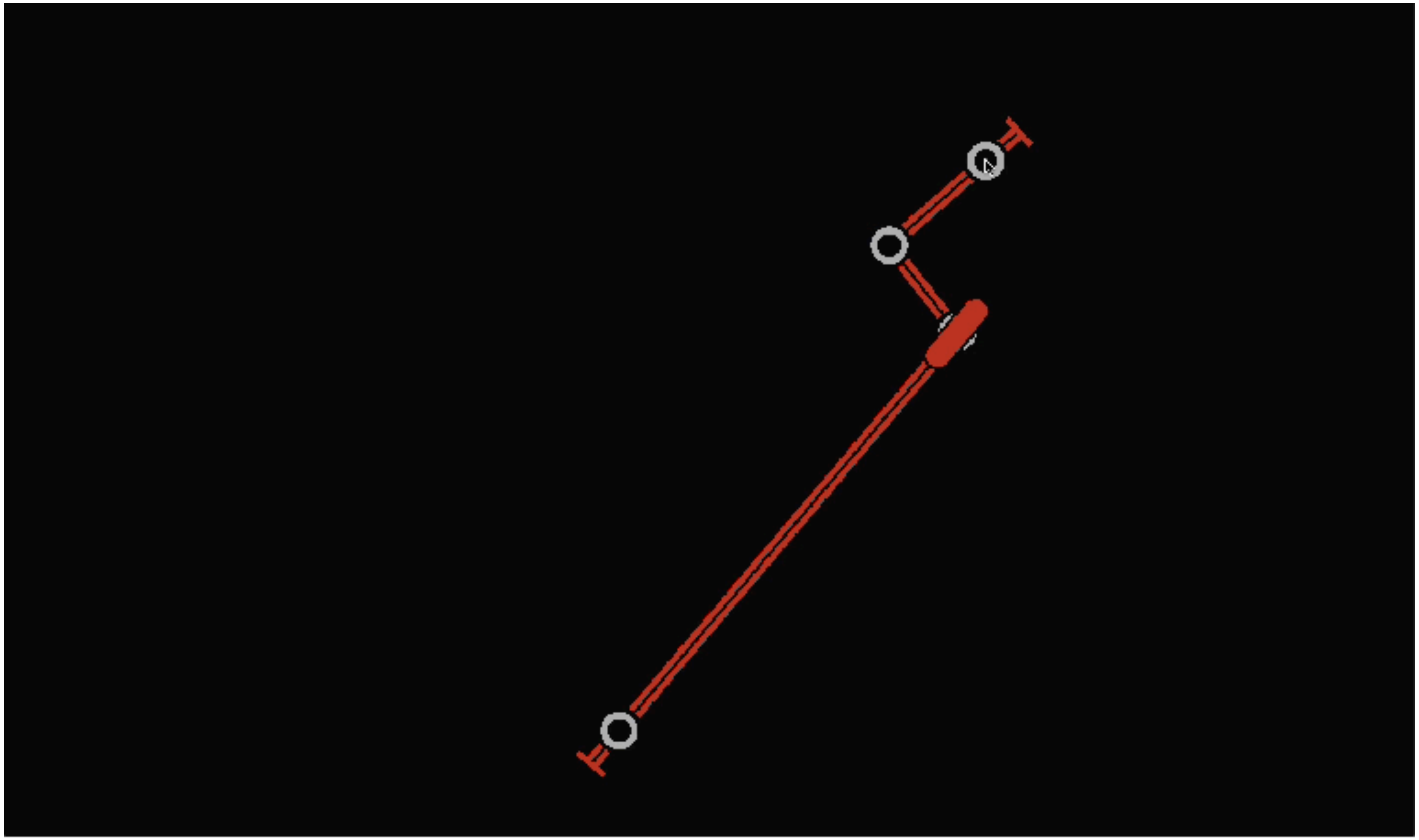
Tube Line Construction

```
-- | Handle user input to construct the tube line system.
handleTubeConstruction
  :: Event
  -> (Tube, Maybe IncompleteAction)
  -> (Tube, Maybe IncompleteAction)
handleTubeConstruction
  (EventKey (MouseButton LeftButton) Down _ point)
  (tube, _)
  = (tube, startAction point tube)
handleTubeConstruction
  (EventKey (MouseButton LeftButton) Up   _ point)
  (tube, Just ia)
  = case completeAction point ia tube of
      Just ca -> (applyCompleteAction ca tube, Nothing)
      _ -> (tube, Nothing)
handleTubeConstruction _ t = t
```

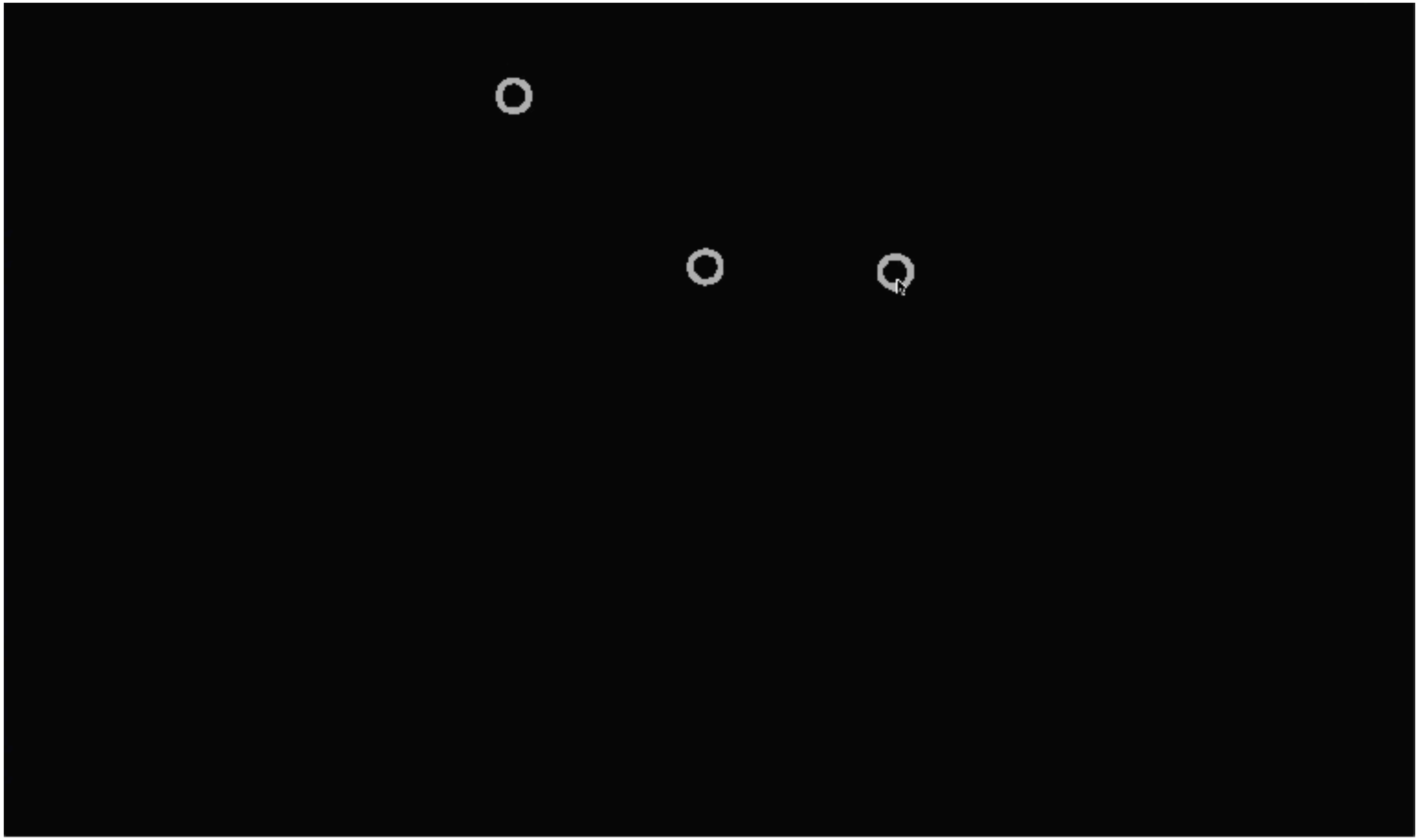
Tube Line Construction

```haskell
-- | Handle user input to construct the tube line system.
handleTubeConstruction
  :: Event
  -> (Tube, Maybe IncompleteAction)
  -> (Tube, Maybe IncompleteAction)
handleTubeConstruction
  (EventKey (MouseButton LeftButton) Down _ point)
  (tube, _)
  = (tube, startAction point tube)
handleTubeConstruction
  (EventKey (MouseButton LeftButton) Up   _ point)
  (tube, Just ia)
  = case completeAction point ia tube of
      Just ca -> (applyCompleteAction ca tube, Nothing)
      _ -> (tube, Nothing)
handleTubeConstruction _ t = t
```

Tube Line Construction



Tube Line Construction



Passengers

Passengers

```
-- | A passenger is a user of a tube system.
data Passenger = Passenger
  { passengerDestination :: StationId
  }

-- | A station.
data Station = Station
  { stationLocation   :: Point
  , stationPassengers :: [Passenger]
  }

-- | A moving train with passengers on board.
data Train = Train
  { ...
  , trainPassengers :: [Passenger]
  }
```

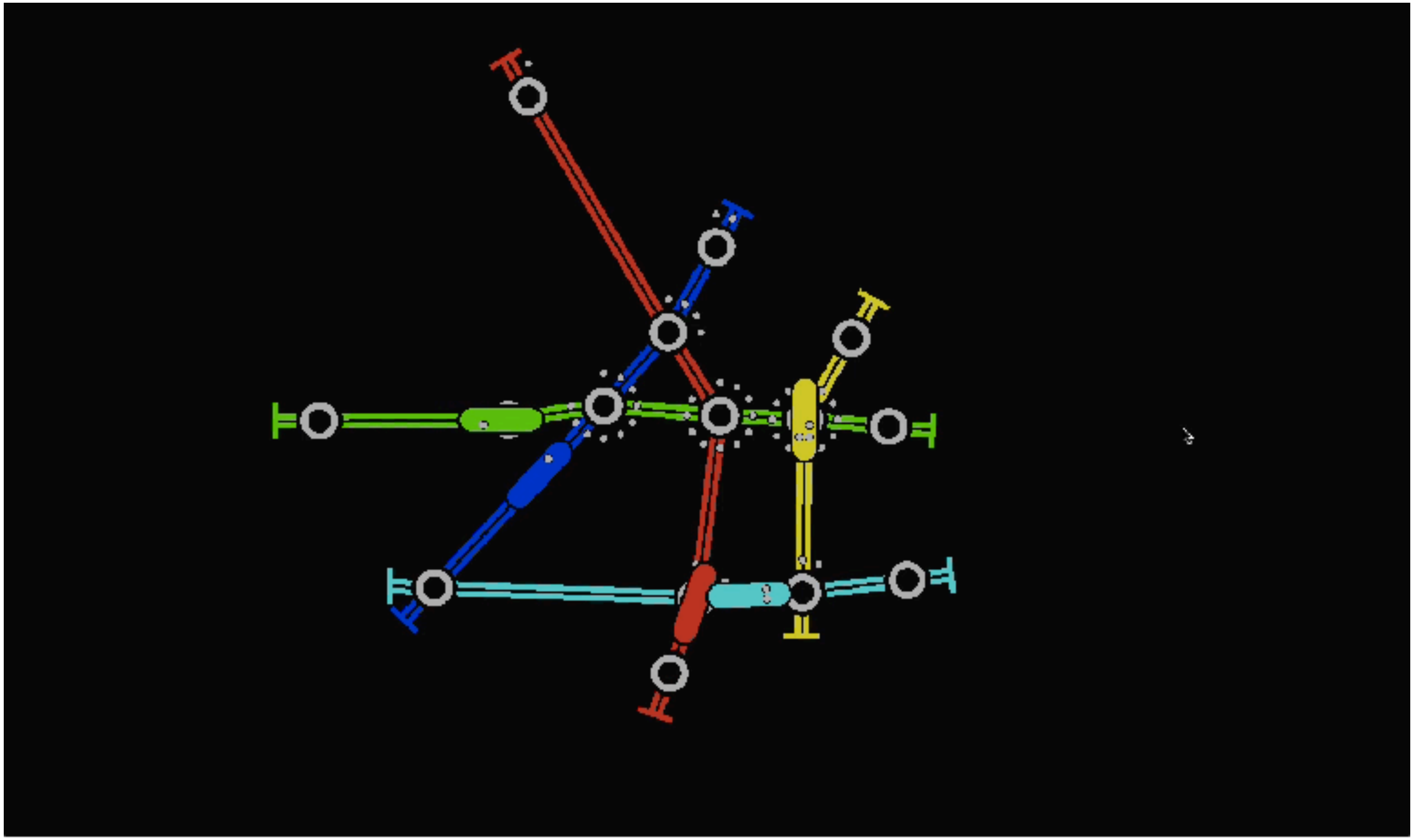
Train Stops

```
-- | Details for a train stop.
data TrainStopEvent = TrainStopEvent
  { trainStopTrain      :: TrainId
  , trainStopStation    :: StationId
  , trainStopTubeLine   :: TubeLineId
  , trainStopDirection  :: TubeLineDirection
  }

-- | Update all trains and produce TrainStopEvents
-- to update passengers later.
updateTubeLineTrains
  :: Float
  -> TubeLineId
  -> TubeLine
  -> ([TrainStopEvent], TubeLine)
```

Boarding and Interchanges

```
-- | Move passengers around when a train stops somewhere.
handleTrainStopEvent :: TrainStopEvent -> Tube -> Tube

-- | Move passengers around when a train stops.
-- Passengers on a station can stay on it or board.
-- Passengers on a train can stay on it or take off.
updatePassengersAt
  :: Station            -- ^ Station where the train has stopped.
  -> Train              -- ^ A train with passengers.
  -> TubeLineId         -- ^ Which line the train belongs to.
  -> TubeLineDirection  -- ^ Where the train is going on the line.
  -> Tube               -- ^ The whole tube line system.
  -> (Station, Train)

-- | Make a decision for a single passenger.
handlePassenger
  :: Passenger  -- ^ The passenger.
  -> StationId  -- ^ Which station is the passenger on.
  -> Tube       -- ^ The whole tube system map.
  -> Maybe (TubeLineId, TubeLineDirection)
```

Passengers



The Uncertain

Random Passengers

```haskell
-- | Pick some existing station at random.
selectRandomStation :: MonadRandom m => Tube -> m StationId
selectRandomStation tube = getRandomR (0, n - 1)
  where
    n = length (tubeStations tube)

-- | Spawn a random passenger by choosing
-- a random start and destination stations.
spawnRandomPassenger :: MonadRandom m => Tube -> m Tube
spawnRandomPassenger tube = do
  from <- randomStation tube
  to   <- randomStation tube
  return (addPassenger from to tube)
```

Random Passenger Flow

```
-- | Randomly choose how many events happen
-- during a unit time interval via Poisson distribution.
poisson
  :: MonadRandom m
  => Float  -- ^ Average rate.
  -> m Int

-- | Spawn a random number of random passengers.
spawnRandomPassengers
  :: MonadRandom m
  => Float  -- ^ Time passed since last frame.
  -> Tube
  -> m Tube
spawnRandomPassengers dt tube = do
  k <- poisson (dt * newPassengerRate)
  applyTimesM k spawnRandomPassenger tube
```

Random City Growth

```haskell
-- | Randomly choose how many events happen
-- during a unit time interval via Poisson distribution.
poisson
  :: MonadRandom m
  => Float  -- ^ Average rate.
  -> m Int

-- | Spawn a random number of random stations.
spawnRandomStations
  :: MonadRandom m
  => Float  -- ^ Time passed since last frame.
  -> Tube
  -> m Tube
spawnRandomStations dt tube = do
  k <- poisson (dt * newStationRate)
  applyTimesM k spawnRandomStation tube

-- | Spawn one station at a random location in a city.
spawnRandomStation :: MonadRandom m => Tube -> m Tube
```

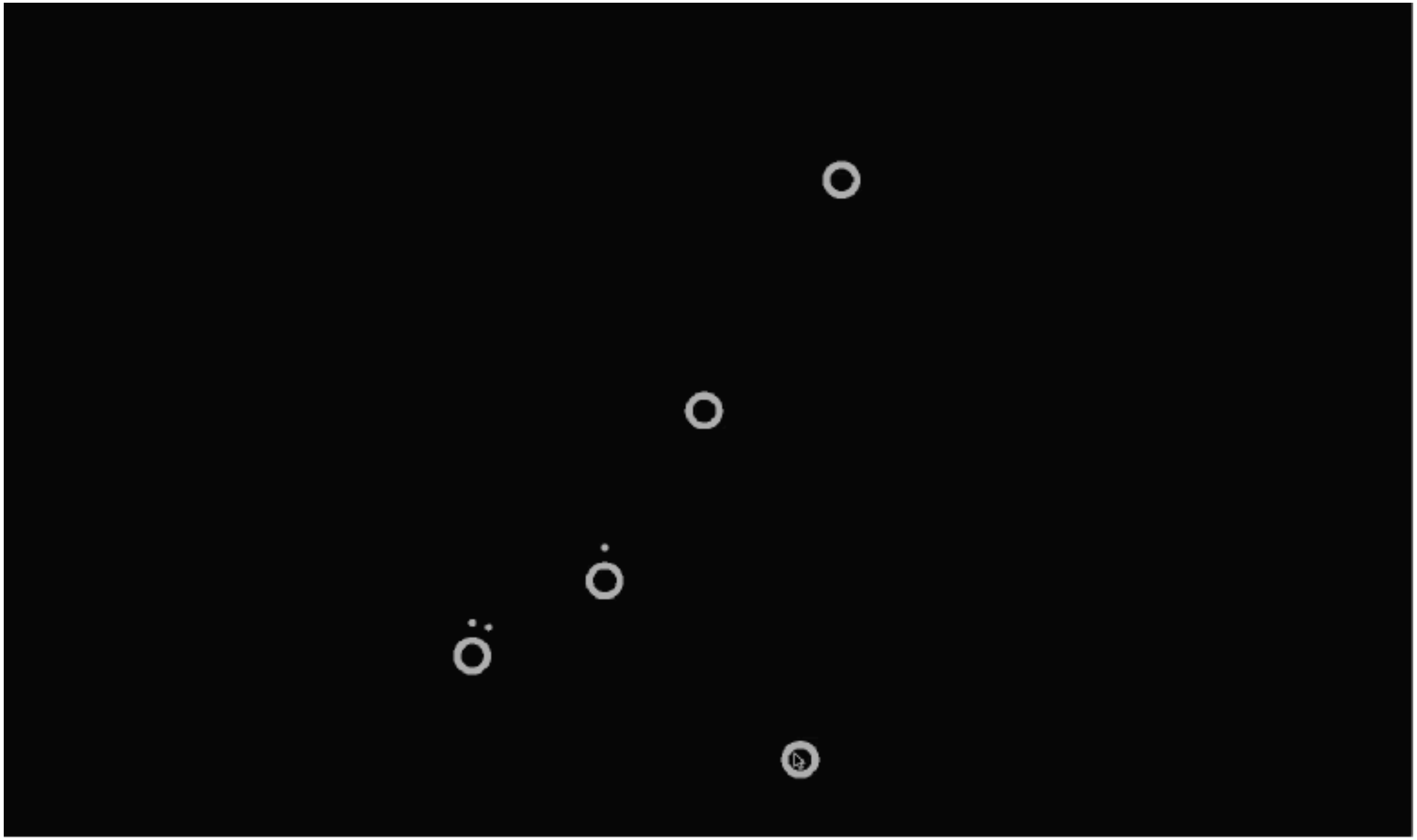
Random Updates

```
– | Update a world using a random generator.
updateWorld :: Float -> (Tube, StdGen) -> (Tube, StdGen)
updateWorld dt (tube, g) = runRand (updateTube dt tube) g

main :: IO ()
main = do
  g <- newStdGen
  play display bgColor fps
    initialWorld renderWorld handleWorld updateWorld
  where
    initialWorld = (initTube, g)
    ...
```

Random Updates

```
-- | Update a world using a random generator.
updateWorld :: Float -> (Tube, StdGen) -> (Tube, StdGen)
updateWorld dt (tube, g) = runRand (updateTube dt tube) g

main :: IO ()
main = do
  g <- newStdGen
  play display bgColor fps
    initialWorld renderWorld handleWorld updateWorld
  where
    initialWorld = (initTube, g)
    ...
```

Random Generation



Game Universe

Universe

```
-- | Everything in a game universe.
data Universe = Universe
  { universeTube   :: Tube
  , universePlayer :: Player
  , universeGen    :: StdGen
  }

-- | Player’s drag-n-drop state.
data Player = Player
  { playerAction  :: Maybe IncompleteAction
  , playerPointer :: Maybe Point
  }
```

Universe

```haskell
-- | Initialise a random universe with some starter stations.
newRandomUniverse :: Int -> IO Universe

-- | Update everything in a game universe.
updateUniverse :: Float -> Universe -> Universe

-- | Handle user input in a game universe.
handleUniverse :: Event -> Universe -> Universe

-- | Render the whole game universe.
renderUniverse :: Universe -> Picture
```

Local Multiplayer

Multiple Players

```haskell
-- | Everything in a game universe.
data Universe = Universe
  { universeTube   :: Tube
  , universePlayer :: Map PlayerId Player
  , universeGen    :: StdGen
  }

-- | Add a new player.
addPlayer :: PlayerId -> Universe -> Universe

-- | Update player’s drag-n-drop state.
updatePlayer
  :: PlayerId -> (Player -> Player) -> Universe -> Universe

-- | Handle user input for a given player.
handleUniverse :: PlayerId -> Event -> Universe -> Universe
```

Bots

```haskell
-- | A bot is just a function that makes
-- a decision based on the current state of the universe.
type Bot = PlayerId -> Universe -> Maybe CompleteAction

-- | A bot that always starts a new line.
newLineBot :: Bot

-- | A bot that always extends an existing line.
extendLineBot :: Bot
```

Concurrency

```haskell
data STM a

-- | Perform a series of STM actions atomically.
atomically :: STM a -> IO a
```

Concurrency

```haskell
data STM a

-- | Perform a series of STM actions atomically.
atomically :: STM a -> IO a

data TVar a

-- | Create a new TVar holding a value supplied.
newTVarIO :: a -> IO (TVar a)

-- | Return the current value stored in a TVar.
readTVar :: TVar a -> STM a

-- | Write the supplied value into a TVar.
writeTVar :: TVar a -> a -> STM ()

-- | Mutate the contents of a TVar.
modifyTVar :: TVar a -> (a -> a) -> STM ()
```

Spawn a Bot

```haskell
-- | A bot is just a function that makes
-- a decision based on the current state of the universe.
type Bot = PlayerId -> Universe -> Maybe CompleteAction

-- | Add a bot to the 'Universe'.
spawnBot :: PlayerId -> Bot -> TVar Universe -> IO ()
spawnBot botId bot w = do
  atomically $ modifyTVar w (addPlayer botId)
  void $ forkIO $ forever $ do
    threadDelay sec
    atomically $ do
      u <- readTVar w
      case bot botId u of
        Just action -> writeTVar w
          (applyPlayerCompleteAction botId action u)
        Nothing -> return ()
  where
    sec = 10^6  -- one second in milliseconds
```

IO with gloss

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

IO with gloss

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

Adding IO

```
updateUniverse
  :: Float -> Universe -> Universe
```

```
handleUniverse
  :: PlayerId -> Event -> Universe -> Universe
```

```
renderUniverse :: Universe -> Picture
```

Adding IO

```
updateUniverseIO
  :: Float -> TVar Universe -> IO (TVar Universe)
```

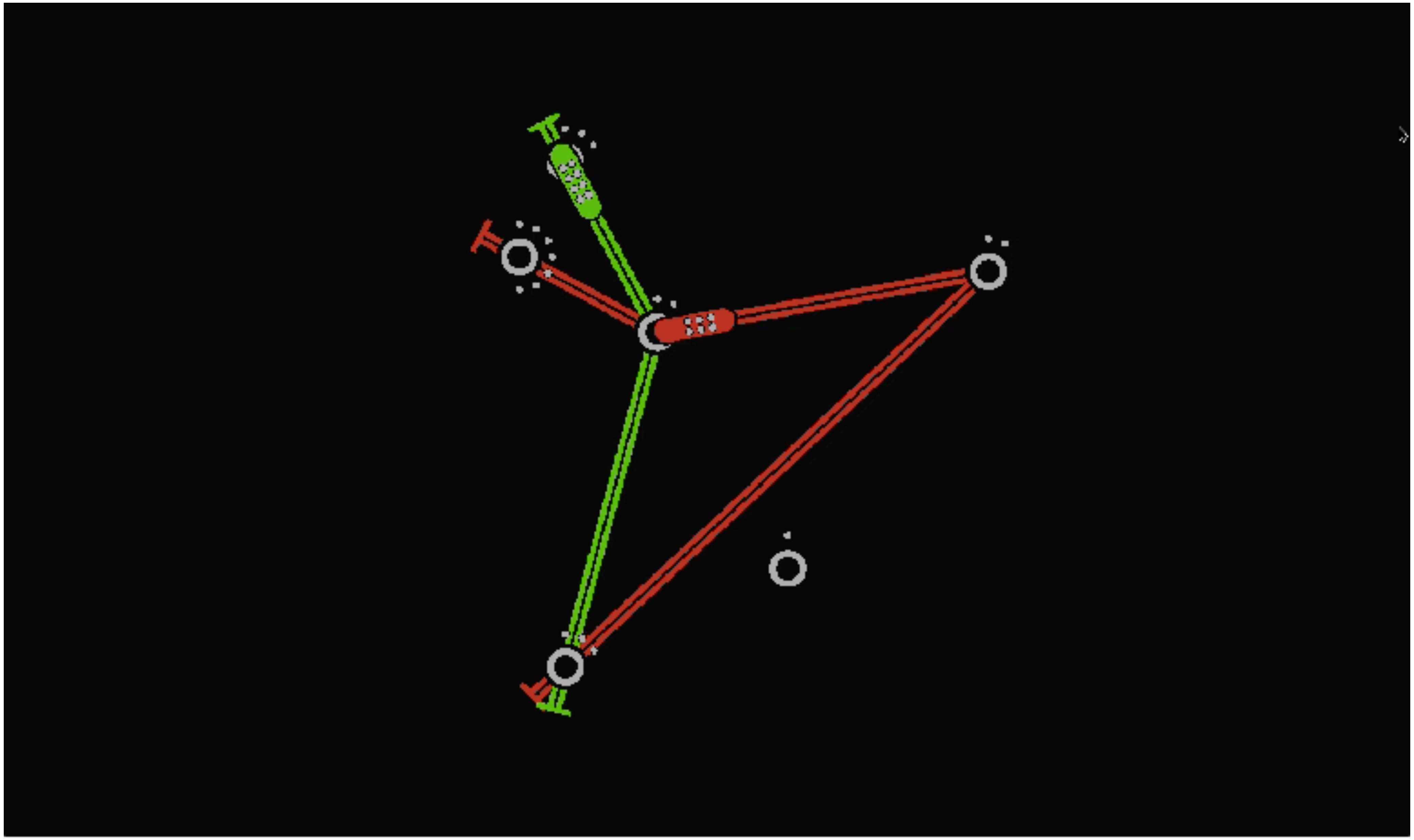
```
handleUniverseIO
  :: PlayerId -> Event -> TVar Universe -> IO (TVar Universe)
```

```
renderUniverseIO :: TVar Universe -> IO Picture
```

Adding IO

```
updateUniverseIO
  :: Float -> TVar Universe -> IO (TVar Universe)
updateUniverseIO dt w = do
  atomically $ modifyTVar w (updateUniverse dt)
  return w

handleUniverseIO
  :: PlayerId -> Event -> TVar Universe -> IO (TVar Universe)
handleUniverseIO _ (EventKey (SpecialKey KeyEsc) Down _ _) _
  = exitSuccess -- exit on ESC
handleUniverseIO playerId event w = do
  atomically $ modifyTVar w (handleUniverse playerId event)
  return w

renderUniverseIO :: TVar Universe -> IO Picture
renderUniverseIO = fmap renderUniverse . readTVarIO
```

Playing with Bots



Multiplayer over Web

WebSockets

```
-- | Receive data from a WebSocket connection.
receiveData :: WebSocketsData a => Connection -> IO a


– | Send data over a WebSocket connection.
sendBinaryData :: WebSocketsData a => Connection -> a -> IO ()
```

Binary Serialization

```
deriving (Generic, Binary)
```

Server

```
-- | Server config.
data Config = Config
  { configUniverse  :: TVar Universe
  , configClients   :: TVar (Map PlayerId Client)
  , configPlayerIds :: TVar [PlayerId]
  }


-- | Default server config with empty universe and no clients.
newConfig :: IO Config
newConfig = do
  universe <- newRandomUniverse 3
  cfg <- atomically $ Config
          <$> newTVar universe
          <*> newTVar Map.empty
          <*> newTVar (map show [1..])
  spawnBot "Bot 1" newLineBot     (configUniverse cfg)
  spawnBot "Bot 2" extendLineBot  (configUniverse cfg)
  return cfg
```

Server

```
-- | The Game of Tubes server.
server :: Config -> Server TubesAPI
server config = Tagged (websocketsOr defaultConnectionOptions wsApp backupApp)
  where
    wsApp :: ServerApp
    wsApp pending_conn = do
        conn <- acceptRequest pending_conn
        playerId <- addClient conn config
        putStrLn $ playerId ++ " joined!"
        handleActions playerId conn config

    -- this application will be used for non-websocket requests
    backupApp _ respond = respond $ responseLBS status400 [] "Not a WebSocket request”

-- | Add a new client to the server state.
-- This will update 'configClients' and add
-- a new player to the 'configUniverse'.
addClient :: Client -> Config -> IO PlayerId
addClient client Config{..} =
  atomically $ do
    playerId:playerIds <- readTVar configPlayerIds
    writeTVar configPlayerIds playerIds
    modifyTVar configClients (Map.insert playerId client)
    modifyTVar configUniverse (addPlayer playerId)
    return playerId
```

Server

```haskell
-- | An infinite loop, receiving data from the 'Client'
-- and handling its actions via 'handlePlayerAction'.
handleActions :: PlayerId -> Connection -> Config -> IO ()
handleActions playerId conn cfg@Config{..} = forever $ do
  action <- decode <$> receiveData conn
  atomically $ do
    modifyTVar configUniverse (applyPlayerAction action playerId)

-- | Periodically update the 'Universe' and send updates to all the clients.
periodicUpdates :: Int -> Config -> IO ()
periodicUpdates ms cfg@Config{..} = forever $ do
  threadDelay ms -- wait ms milliseconds
  clients <- readTVarIO configClients
  when (not (null clients)) $ do
    universe <- atomically $ do
      universe <- updateUniverse dt <$> readTVar configUniverse
      writeTVar configUniverse universe
      return universe
    broadcastUpdate universe cfg
  where
   dt = fromIntegral ms / 1000000
```

Server

```
main :: IO ()
main = do
  config <- newConfig
  forkIO $ periodicUpdates 10000 config
  run 8000 $ unTagged $ server config
```

Client

```
-- | Game state on client.
data ClientState = ClientState
  { clientUniverse    :: TVar Universe
  , clientConnection  :: Connection
  }

-- | Handle user input.
handleClient :: Event -> ClientState -> IO ClientState
handleClient (EventKey (SpecialKey KeyEsc) Down _ _) _
  = exitSuccess -- exit on ESC
handleClient event cs@ClientState{..} =
  case eventToPlayerAction event of
    Just action -> do
      -- fork to avoid interface freezing
      _ <- forkIO $
        sendBinaryData clientConnection (encode action)
      return cs
    _ -> return cs
```

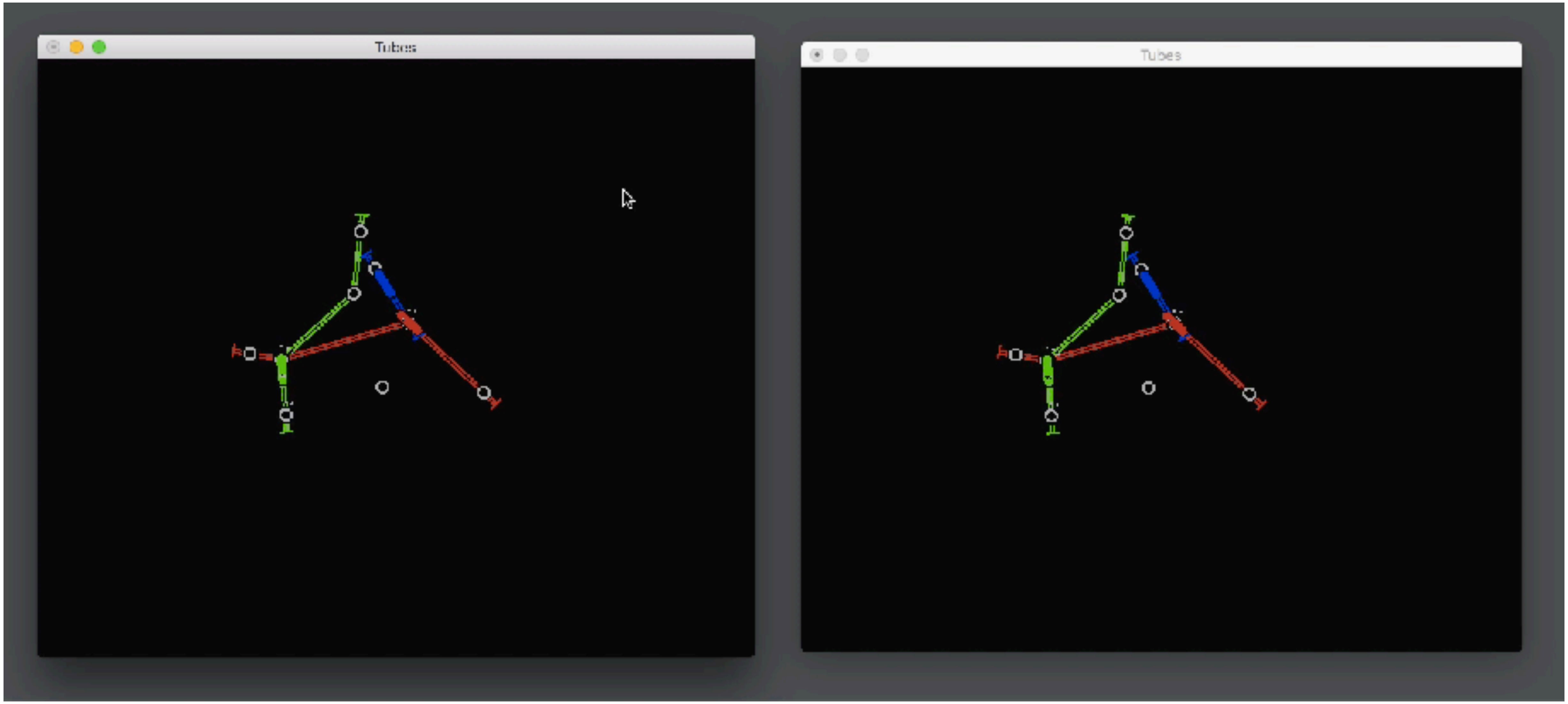
Client

```
-- | Game state on client.
data ClientState = ClientState
  { clientUniverse    :: TVar Universe
  , clientConnection  :: Connection
  }

-- | Handle 'Universe' updates coming from server.
handleUpdates :: ClientState -> IO ()
handleUpdates ClientState{..} = forever $ do
  (players, tube) <- decode <$> receiveData clientConnection
  atomically $ modifyTVar clientUniverse $ \universe ->
    universe
      { universeTube = tube
      , universePlayers = players
      }
```

Client

```
-- | Game state on client.
data ClientState = ClientState
  { clientUniverse    :: TVar Universe
  , clientConnection  :: Connection
  }

-- | Draw the current state of the 'Universe'.
renderClient :: ClientState -> IO Picture
renderClient ClientState{..} = renderUniverseIO clientUniverse

-- | This does nothing since updates come from server.
updateClient :: Float -> ClientState -> IO ClientState
updateClient _dt cs = return cs
```

Multiplayer



What’s next?

- Tubes at https://github.com/fizruk/tubes
- Snakes at https://github.com/fizruk/snakes-demo
- Join me on the **HaskellX Community Weekend**!
    - I’m going to challenge myself with a WebVR/WebAR
    - I can help you make a simple game of your own