HGamer 3D

a toolset for developing games with haskell

Peter Althainz

HAL 2016, 14.9., Leipzig
Agenda

Part I – Intro
  History
  Shortcomings
  New Approach
  API Technology

Part II – What can I do with
  API Structure
  Showcase Game
    Demo & Features
    Architecture
    HGamer3D API examples
  Feature Coverage of HGamer3D
Part I - History
History

Started with Irrlicht bindings – cumbersome
Then more serious with Ogre3D bindings
Got into trouble
- Spent months with API changes, GUI integration, shader libs
- Fragmented build process, not working on other computers
- No binary distribution
- No media tooling
- External C-libs are a nightmare for others to build

New Approach
New Approach – Fixed It

Integrated Engine: Urho3D
- All parts integrated, shaders as well, build tooling much better
- Media tooling available

Binary Components, Installer Technology
- Fixed distribution of programs, C-library problem

Coarse Grained API Strategy
- Fixes API stability issues
API Technology

Zero Install Tool – Arriccio (Go)
- Dependency injection and resolution
- Web download for your platform

Component Runtime – Intonaco (Rust)
- Lock free data-structures for thread abstraction
- Intermediate format: messagepack
- Entity-Component-System

Data Description Tool – Sinopia (Haskell)
- To describe ADT for interface language independent
10 min to install in 5 easy steps:

- download aio for your platform
- aio Stack setup –resolver lts-5.8
- aio CreateProject
- ./build
- ./run

The same Haskell code on: Windows / Linux / Mac!
Part II
What can I do with it?
Entities

- Composable from components
- Reference style
- CRUD within IO Monad
- Components are regular ADT’s
- Reading & writing fully thread-safe
- One or more of components are kind-of objects, the other attributes
- Creation with „newE“ and component list
- Attributes modify all objects in Entity
- Threading – behind the scenes
- All you need to know: Data Types

```haskell
-- create minimum elements, like a camera
cCam <- newE hg3d [  
  ctCamera #: FullViewCamera,  
  ctPosition #: Vec3 1 1 (-30.0),  
  ctLight #: Light PointLight 1.0 1000.0 1.0  
]

-- do something interesting here, in this example case, it is a text and  
-- a rotating cube
cText <- newE hg3d [  
  ctText #: "Rotating Cube Example",  
  ctSCREENRect #: Rectangle 10 10 100 25  
]

cGeo <- newE hg3d [  
  ctGeometry #: ShapeGeometry Cube,  
  ctMaterial #: matBlue,  
  ctScale #: Vec3 10.0 10.0 10.0,  
  ctPosition #: Vec3 0.0 0.0 0.0,  
  ctOrientation #: unitU  
]

let rotateCube = do  
  forever $ do  
    updateC cGeo ctOrientation (\u -> (rotU vec3Z 0.02) .* u)  
    updateC cGeo ctOrientation (\u -> (rotU vec3X 0.015) .* u)  
    sleepFor (msecT 12)

forkIO rotateCube  
return ()
```
Only Data Types to memorize

 HGamer3D.Graphics3D.Material
 Module providing the Material type

 HGamer3D.Graphics3D.Light
 Module providing the Light type

 HGamer3D.GUIGUI
 H Gamer3D.GUICheckBox
 H Gamer3D.GUIDropDropDownList
 H Gamer3D.GUIEditText
 H Gamer3D.GUISlider
 H Gamer3D.GUIText
 H Gamer3D.UIElement

 HGamer3D.Graphics3D
 HGamer3D.Graphics3D.Camera
 HGamer3D.Graphics3D.Geometry
 HGamer3D.Graphics3D.Graphics3DCommand
 HGamer3D.Graphics3D.Graphics3DConfig
 HGamer3D.Graphics3D.Light
 HGamer3D.Graphics3D.Material
 HGamer3D.Graphics3D.Window

 HGamer3D.Input
 HGamer3D.InputInputEventHandler
 HGamer3D.InputJoystick
 HGamer3D.InputKeyboard
 HGamer3D.InputMouse

 HGamer3D.Util
 HGamer3D.Util.FileLocation
 HGamer3D.Util.UniqueName
 HGamer3D.Util.Variable
API Three
Examples for Sound and Event Handling
Vector Arithmetics with Vect package from Balázs Kőműves

- Vector substraction, addition, scaling, rotation ...
- Quaternion arithmetics
- Examples:
  - Rotation around an axis: “updateC eGeo ctOrientation (\u -> (rotU vec3Z 0.02) .* u)”
  - Implementation of yaw, roll, pitch:

```
-- yaw, roll, pitch functions
-- functions, to rotate on axis, relative to object
rotRelativeToObjectAxis :: Orientation -> Vec3 -> Float -> Orientation
rotRelativeToObjectAxis ori axis val = let
  odir = actU ori axis
  qrot = rotU odir val
  nrot = qrot .* ori
in nrot
```
Showcase Game

Features

- Sound, GUI, Key-Input
- Different game modes
  - Intro Screen
  - Flying / Playing
- Animated Invaders
- Fast Key-Input & Shooting
- Collision Detection by Haskell
Haskell – Game Architecture

- Actors to partition code and use multi-threading scalable
- Reader – State – Monad for Actor functions
- Persistent data types, used a tree for all moving game elements
- Traversable, to operate on tree
- Collision detection, send messages of current state to detection actor
- Not one big loop but multiple small ones, with confined state for each
Actors: looping function within Reader-State-Monad

```haskell
newtype Actor = Actor (MVar Message)

newActor :: IO Actor
newActor = do
    mv <- newEmptyMVar
    return (Actor mv)

type ReaderStateIO r s a = StateT s (ReaderT r IO) a

runActor :: Actor -> (Message -> ReaderStateIO r s () ) -> r -> s -> IO ()
runActor a@(Actor mv) f r1 s1 = do
    let loop mv s = do
            msg <- takeMVar mv
            (r', s') <- runReaderT (runStateT (f msg) s) r1
            loop mv s'
        forkIO $ loop mv s1
        sendMsg a InitActor

stopActor a = sendMsg a StopActor

sendMsg :: Actor -> Message -> IO ()
sendMsg (Actor mv) m = putMVar mv m
```

---

Actors

https://www.youtube.com/watch?v=VczbbiRmDik
Structure of program

- Screen Actor: Distributes Events based on Mode
- Move Actor: Invader movement
- Canon Actor: Canon movement
- Collision Actor: Collision detection
- Sound Actor
- Status Actor
- Flying Actor
Actors are clearly separated pieces

```
-- create actors

-- interconnect and run them
runActor statusBarA statusBarActorF hg3d (undefined, undefined, undefined)
runActor flyingA flyingActorF (hg3d, cam) (undefined, undefined)
runActor musicA musicActorF hg3d (undefined, undefined, undefined, undefined)
runActor collA collisionActorF (moveA, canonA, statusBarA, keys) (Nothing, Nothing, undefined)
runActor canonA canonActorF (hg3d, screenA, musicA, collA, keys) (undefined, undefined, undefined)
runActor moveA movementActorF (hg3d, screenA, musicA, collA, keys) (0, undefined, [])
runActor screenA gameScreenActorF (hg3d, moveA, canonA, collA, musicA, flyingA, statusBarA) (undefined, undefined, ProgramInitializing)
runActor keyA keyInputActorF (hg3d, screenA) (undefined, [])

let cycleLoop n m = do
  if n == 0
    then sendMessage screenA SlowCycle
    else return ()
  sendMessage keyA PollKeys
  sendMessage screenA FastCycle
  sleepFor (msecT 30)
  cycleLoop (if n == 0 then m else n - 1) m

forkIO $ cycleLoop 0 3

-- start with game logic by starting first screen
sendMessage screenA StartProgram
```
Screen Actor

gameState is stored in state monad

Big switch on gameState

Incoming messages are handled depending on current gameState

In this actor they are just re-distributed to next actors

Example: In pause mode keys are not forwarded, other keys are valid

State Machine for mode handling

type GsR = (HG3D, Actor, Actor, Actor, Actor, Actor, Actor)
type GsS = ((Entity, Entity, Entity, Entity, Entity), T.Text, GameState)

gameScreenActorF :: Message -> ReaderStateIO GsR GsS ()
gameScreenActorF msg = do

  (hg3d, moveA, canonA, collA, musicA, flyingA, statusBarA) <- lift ask
  (screenText, name, gameState) <- get

  let returnStay = return ()

  let returnMoveTo state = put (screenText, name, state) >> return ()

  case gameState of

    ProgramInitializing ->
      case msg of
        StartProgram -> do
          (eT1, eT2, eT3, eT4, eName) <- liftIO $ showInitScreen hg3d
          liftIO $ sendMessageA StartMusic
          put ((eT1, eT2, eT3, eT4, eName), name, InitScreen) >> return ()
          _ -> returnStay
          P1
Depending on mode, messages are distributed to next actor.

```haskell
    BuildField1 ->
        case msg of
            BuildDone -> do
                liftIO $ sendMsg statusBarA (SetName name)
                liftIO $ sendMsg statusBarA (SetCount 0)
                liftIO $ sendMsg statusBarA (SetMode "playing")
                liftIO $ sendMsg statusBarA (DisplayStatus)
                returnMoveTo PlayGame
            _    -> returnStay

    PlayGame ->
        case msg of
            FastCycle -> do
                liftIO $ sendMsg canonA FastCycle
                liftIO $ sendMsg collA FastCycle
                returnStay
            SlowCycle -> do
                liftIO $ sendMsg moveA SlowCycle
                liftIO $ sendMsg canonA SlowCycle
                returnStay
```

Screen Actor II

There are two beats, a fast and a slow cycle beat.

During gameplay canon movement and collision detection are done more often then movement of invaders.
Data Structure (Persistent)

- Tree structure to be flexible
- Sub-elements move with parent
- Each element is an Hmap
- More: canon, shot, ...

**Properties**:
- Position
- HG3D Entity

**Diagram**:
- Root
- Invader Row
  - Invader I
  - Invader II
- Pixel A
  - Cube I
  - Cube II
- Pixel B
  - Cube I
  - Cube II
  - ...
  - ...
  - ...
  - ...
  - ...
  - ...

- properties: Position
- Animation
Animation - Traverse over tree

-- pixel animation
(gameData'', _) <- mapAccumLM \((\text{nodeType}, \text{nodeData})\) \(\text{nt, nd}\) -> case \text{nodeType} of
   \((\text{Invader} \_\_\_)\) -> do
     let \(\text{ai} = \text{nodeData} ! \text{kanim}\)
     let \(\text{ai'} = \text{getCurrentAnimation ai moves}\)
     let \(\text{nodeData'} = \text{setData kanim ai'} \text{nodeData}\)
     return ((\text{nodeType}, \text{nodeData'}), (\text{nodeType}, \text{nodeData'}))  -- set acc to nodeData of Invader

\(\text{PixelA} \rightarrow \text{do}\)
   let \(\text{ai} = \text{nd} ! \text{kanim}\)  -- accu, this is the previous Invader node info!
   if \(\text{aiSwapNow} \text{ai}\)
     then do
       let \((\text{x, y}) = \text{nodeData} ! \text{kpos}\)
       case \text{aiType ai} of
         \(\text{PixelA} \rightarrow \text{setC (nodeData ! kent) ctPosition $ relativePosFromPixelPos (nd ! kdim) (x, y)}\)
         \(\text{PixelB} \rightarrow \text{setC (nodeData ! kent) ctPosition (Vec3 (-1000.0) 0 0)}\)
         \(_ \rightarrow \text{return ()}\)
       else \text{return ()}\n     return ((\text{nodeType}, \text{nodeData}), (\text{nt, nd}))
collisionActorF :: Message -> ReaderStateIO CoaR CoaS ()
collisionActorF msg = do

  (moveA, canonA, statusA, keys) <- lift ask
  (invaderData, canonData, busyFlag) <- get

  case msg of

    InitActor -> do
      mv <- liftIO $ newMVar ()
      put (invaderData, canonData, mv)

    FastCycle -> case (invaderData, canonData) of
      (Just invaderData', Just canonData') -> do
        mBF <- liftIO $ tryTakeMVar busyFlag
        case mBF of
          Just () -> do
            liftIO $ forkIO $ runCollisionDetection keys invaderData' canonData' busyFlag moveA canonA statusA
            return ()
          Nothing -> return ()
          _ -> return ()

    ActualInvaderData invaderData' -> put (Just invaderData', canonData, busyFlag)
    ActualCanpData canonData' -> put (invaderData, Just canonData', busyFlag)
runCollisionDetection :: Keys -> GameData -> GameData -> MVar () -> Actor -> Actor -> Actor -> IO ()
runCollisionDetection keys invaderData canonData busyFlag moveA canonA statusA = do

   let (kent, kdim, kpos, khits, kanim, kuni) = keys

   let invs = filter (\(nt, nd) -> case nt of
                    (Invader _) -> let (x, y) = nt ! kpos in if x < (-500) then False else True
                    _ -> False
                ) (flatten invaderData)

   let shots = filter (\(nt, nd) -> nt == Shot) (flatten canonData)

   let cols = [(s ! kuni, i ! kuni) | (_, s) <- shots, (_, i) <- invs, isCollision keys i s]

   if length cols > 0
       then do
           (mapM (\(sid, iid) -> do
                          sendMsg canonA $ Collision sid
                          sendMsg moveA $ Collision iid
                     ) cols) >>= return ()

       else return ()
- modular by actors
- fully multi-threaded
- simple tree data structure
- HMap for properties
- beginner/intermediate Haskell
- API enables this kind of structure

Combines persistent data structure with threading, still being modular

No complex Haskell magic needed, uses the basics of FP
Feature Coverage of HGamer3D

very basic feature coverage:
- 3d geometry, GUI, sound, device input, light, material
- wish-list: particles, effects, animation, network, physics, ...
- targeting today: education, fun programming, ...

included:
- easy to use
- beginner friendly API
- fully multi-threading capable
let future = fmap (createNewVersions .
    addFeatures .
    evolveGame) (Maybe neededTime)

www.hgamer3d.org

Thank You For Your Time!