Computer Games'

Carl Joachim Svenn
(karamellpelle@hotmail.com)

March 29, 2012

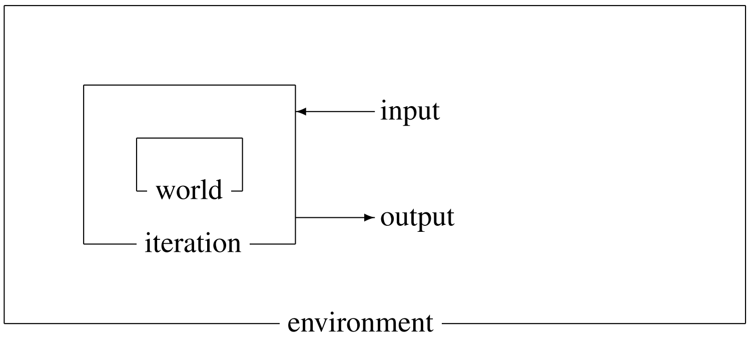
Abstract

The reason I got interested in programming is because I wanted to create computer games. I tried to find out how computer games typically should be constructed, but I could not find any information about that. So I had to make my own construction for computer games, and this is the (current) result. I would be glad if somebody can learn me more game programming.

This paper is an updated and better version of the previous paper. We make an attempt on defining computer games, and construct such a thing in Haskell, with focus on implementing the game flow of a computer game.

Definition

A computer game is the sideeffects of the following system:



At each frame t there is an iteration i_t which iterate the current world. This gives sequences

$$\begin{tikzcd}\mathtt{i}_0 \arrow[r] & \mathtt{i}_1 \arrow[r] & \mathtt{i}_2 \arrow[r] & \cdots\end{tikzcd}$$

$$\begin{tikzcd}\mathtt{w}_0 \arrow[r] & \mathtt{w}_1 \arrow[r] & \mathtt{w}_2 \arrow[r] & \cdots\end{tikzcd}$$

$$\begin{tikzcd}\mathtt{o}_0 \arrow[r] & \mathtt{o}_1 \arrow[r] & \mathtt{o}_2 \arrow[r] & \cdots\end{tikzcd}$$

where $\mathtt{i}_t$ is the input state, $\mathtt{w}_t$ is the game state, $\mathtt{o}_t$ is the output state, at frame t.

Below we construct such a system.

The environment

The environment typically consist of input, output, time, random and resources used in game (time, random and resources are not considered as input/output). The input may typically be

- Keys (keyboard, mouse, joystick, etc.)
- Network

The output may typically be

- Screen
- Network
- Sound

The world

The world data is game specific. But the world data may typically contain the following fields:

```
data World =
    World
    {
        worldTick :: TickT,
        worldEvents :: [WorldEvent],
        -- ...
    }
```

where `WorldEvent` is game specific.

Computer games inside computer games

It might be reasonable to consider a computer game as computer games inside computer games. For example, a game may consist of a set of levels, making a story. Hence we can think of `LevelWorld` as contained inside `StoryWorld`. Then we can let an iteration of `StoryWorld` iterate `LevelWorld` in order to play that level of the story. `StoryWorld` may again be contained in `RunWorld`, in order to run the whole game.

Construction

We model computations inside an environment as a monad `MEnv`. If `a` is a world contained inside a world `b`, we let an iteration of `a` work on both `a` and `b`. So `a` is typically the main world. We will use a stack of iterations, and at each frame we pop the top element of the current stack (if any) and perform that iteration on the worlds `a` and `b`, modifying the worlds and giving a new top on the stack.

```
data Iteration a b =
    Iteration
    {
        iteration :: a -> b -> MEnv (a, b, IterationStack a b)
    }


type IterationStack a b =
    [Iteration a b]


-- | running a stack on 'a' and 'b'
```

```haskell
runABStack :: a -> b -> IterationStack a b -> MEnv (a, b)
runABStack a b stack =
    case stack of
        []          -> return (a, b)
        (i:is)      -> do
            -- (MEnv iteration begin)

            (a', b', top) <- (iteration i) a b

            -- (MEnv iteration end)

            runABStack a' b' (top ++ is)


-- | iterating a stack on 'a' and 'b'
--   (typically to be done inside another iteration)
iterateABStack :: a -> b -> IterationStack a b ->
                  MEnv (a, b, IterationStack a b)
iterateABStack a b stack =
    case stack of
        []          -> return (a, b, [])
        (i:is)      -> do
            (a', b', top) <- (iteration i) a b
            return (a', b', top ++ is)
```

Iteration

In our construction we want to, at each frame, output the current world and then step it. Hence we have the following function:

```haskell
defaultIteration ::
                s ->
                (s -> a -> b -> MEnv s) ->
                (s -> a -> b -> MEnv (a, b, IterationStack a b)) ->
                Iteration a b
defaultIteration s output step =
    Iteration $ \a b -> do
        s' <- output s a b
        step s' a b
```

The parameter `s` makes it possible for an iteration to work with a state.

Output

The output is game and environment specific. Examples here are screen drawing, send data on network, playing sounds.

Step

The action of step is to modify the worlds and modify the stack of iterations. Since we have output at each frame, `defaultStep` is "do–think" instead of "think–do".

```
defaultStep :: (s -> a -> b -> MEnv (s, a, b)) ->
               (s -> a -> b -> MEnv (a, b, IterationStack a b)) ->
               s -> a -> b -> MEnv (a, b, IterationStack a b)
defaultStep doWorld
            thinkWorld = \s a b -> do

    -- do
    (s', a', b') <- doWorld s a b

    -- think
    thinkWorld s' a' b'
```

Do

A world may consist of physical objects which are modified by time, and thus we define a function `defaultDo`:

```
defaultDo :: (s -> a -> b -> MEnv (s, a, b)) ->
             (MEnv TickT, TickT -> MEnv (), TickT, TickT) ->
             (TickT -> s -> a -> b -> MEnv (s, a, b)) ->
             (s -> a -> b -> MEnv (Maybe (s, a, b))) ->
             (s -> a -> b -> MEnv (s, a, b)) ->
             s -> a -> b -> MEnv (s, a, b)
defaultDo modify
     (getTick, setTick, dtUnit, maxElaps)
     stepDT
     breakModify
     defaultModify = \s a b -> do

        -- begin modify world
        (s', a', b') <- modify s a b

        -- ignore too long elaps
        tick <- getTick
        when ( worldTick a' + maxElaps <= tick ) $
            setTick ( worldTick a' + maxElaps )
        tick <- getTick

        -- step physics in 'dtUnit' portions
        helper tick s' a' b'
        where
          helper tick s a b =
            if worldTick a + dtUnit <= tick

              -- take a 'dtUnit'-step of physical objects
              then do
                (s', a', b') <- stepDT dtUnit s a b
                maybeSAB <- breakModify s' a' b'
                case maybeSAB of
                    -- continue
                    Nothing               -> helper tick s' a' b'
                    -- end modify world
                    Just (s'', a'', b'')  -> return (s'', a'', b'')

              -- end modify world
              else do
                defaultModify s a b
```

After a first modification of world, we step the physical part of the world by elapsed time. This is done in constant `dtUnit` timesteps. We may escape from further `stepDT`–steps and then modify the worlds with `breakModify`, or continue until reaching end, whereat we modify the worlds with `defaultModify`.

Modify

A first modification of world before the physical step may look like:

```
defaultModify :: (s -> a -> b -> MEnv (s, a, b)) ->
                 (s -> a -> b -> MEnv (s, a, b)) ->
                 (s -> a -> b -> MEnv (s, a, b)) ->
                 s -> a -> b -> MEnv (s, a, b)
```

```
defaultModify beginModify
              controlModify
              updateModify = \s a b -> do

    -- modify world at beginning of step (like clean up)
    (s', a', b') <- beginModify s a b

    -- modify world from controls (like keys, network, AI, ...)
    (s'', a'', b'') <- controlModify s' a' b'

    -- modify world by updating it (like timers, ...)
    (s''', a''', b''') <- updateModify s'' a'' b''

    return (s''', a''', b''')
```

StepDT

The `stepDT` function is world and game specific. The function typically also checks for collisions between physical objects, so an implementation might be like:

```
stepDT :: CollisionHandler ->
          TickT ->
          s -> a -> b -> MEnv (s, a, b)
stepDT handleCollision dt = \s a b -> do
    -- make a 'dt' step of physical objects. check for collisions
    -- between such, and use 'handleCollision' to make changes to
    -- world. example of such changes to world is pushing
    -- WorldEvent's to world. increment worldTick by 'dt'
    -- ...
```

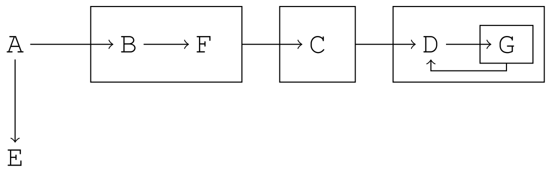
Think

At iteration end, we modify the worlds and decide the next top of the iteration stack. An iteration `A` might decide stack tops of different sorts:

`[]`	finish iteration `A`
`[B]`	finish iteration `A`, new iteration `B`
`[B, C]`	finish iteration `A`, new iteration `B`, iteration `C` on finish
`[A]`	continue iteration `A`
`[B, A]`	new iteration `B`, continue with iteration `A` on finish
`[A, B]`	continue iteration `A`, iteration `B` on finish

(this is not a complete characterization). So it is possible to construct a sequence of iterations `A` → `B` → `C` differently: iteration `A` returns `[B, C]`, and iteration `A` returns `[B]`, iteration `B` returns `[C]`. In the second case, iteration `B` needs to be aware of iteration `C`, but in the first case, it does not. Hence the second construction is more specialized than the first construction.

We should be able to draw the flow of iterations. Here is an attempt:



This should be read as:
Except for `[]` and `[A]`, iteration `A` may return either `[B, C, D]` or `[E]`.
Except for `[]` and `[B]`, iteration `B` may return `[F]`.
Iteration `C` returns `[]` or `[C]`.
Except for `[]` and `[D]`, iteration `D` may return `[G, D]`.
Iteration `E` returns `[]` or `[E]`.
Iteration `F` returns `[]` or `[F]`.
Iteration `G` returns `[]` or `[G]`.

Hence we read a box with an arrow as "on finish".

Iteration modifiers

We can make functions working as tools to manipulate the iterations, changing the flow of iterations, like:

```
-- | modify worlds before iteration
modifyBefore :: Iteration a b ->
                (a -> b -> MEnv (a, b)) ->
                Iteration a b

-- | modify worlds after iteration
modifyAfter :: Iteration a b ->
               (a -> b -> MEnv (a, b)) ->
               Iteration a b

-- | run iteration with local world, then continue with original
--   world when finished
localWorldA :: a ->
               Iteration a b ->
               Iteration a b
```

```
-- | run iteration, then continue with original world when finished
saveWorldA :: Iteration a b ->
              Iteration a b

-- | when finished, choose next iterations by looking at worlds
chooseStackAfter :: Iteration a b ->
                    (a -> b -> MEnv (IterationStack a b)) ->
                    Iteration a b
```

They can in theory be used for special effects in game as jumping back in time with `localWorldA`, jumping forward in time with `saveWorldA`, etc.