Functional languages in games:

Plotting the coup

Sam Martin

AngloHaskell 2009

(Quick slide about me)

- Hi, I'm Sam Martin
- Once game developer
  - Kuju, Intrepid, Lionhead
- Now lighting middleware developer
  - Lead code @ Geomerics: Enlighten (RT radiosity SDK)
- And part time Haskell enthusiast

Reasons for this talk

- Games are a great playground for functional languages
  - Demanding
  - Competitive
  - Interesting!
- Multi-core going to P0WN us all !!1!
- Game developers are an adaptable sort

Talk outline

1. The Why
2. The Killer App: Multi-core
3. Haskell vs. C++
4. Games in Haskell

This is the start of the road. Not a roadmap.

Expect: hand waving, much shuffling of details under carpet till later...

Why mix games and functional languages?

- In a nutshell:
  - Games already quite functional
  - Increase productivity in many ways
  - Increase performance through parallelism

- Rest will become obvious, and...

I am utterly, utterly compelled to see if it will work.

Why now?

- To date, productivity benefits probably not enough to cover cost of transition.
- But we live in interesting times...
- Multithreading really could tip the stalemate into a landslide

Why does multi-core matter?

Looking the devil in the eye

- Now:
  - Most gaming PCs dual core.
    - http://store.steampowered.com/hwsurvey/
  - XBox360, PS3 both multi-core
    - XBox360 = GPU, 3x hyper-threaded cores. UMA. Less in practice.
    - PS3 = GPU, 1x PPU, 6x SPU, explicit DMA.

- Future:
  - Will see consumer 16-core processors this/next year
    - (-ish)
    - AMD/Intel understandably hazy about exact dates.
  - Intel's Larrabee "late-2010" (-ish)
    - Starting with 32 cores
  - GPU-CPU collision course
  - To infinity and beyond!
  - Only X years away!
    - Where X is < 2 or maybe 3. AAAAAAAAAAAAAARRRHHHHHH!

How quickly it starts to matter

- Huge difference between 2, 4, 8, 16,... cores!
- No really, it’s huge!
- (Side track)
  - Actually this a bit of a problem...
  - Not acceptable to just run 2x slower on a PC with half the cores
  - Maintaining consistent frame rates over that range is hard

How quickly it hurts

- Effective parallelism on large #cores is:
  - Increasingly hard
  - Vital

- Amdahl's law: $$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{\#\ \text{processors}} + (1 - \text{Fraction}_{\text{parallel}})}$$

The answer!

- Simple really
- Compile regular code to GPU code!

```
makeFast :: Code -> GPUCode
```

- …
- Done! Now have a cup of tea ☺.

Haskell + multi-core = gold (in theory)

- Basic premise:
  - No side effects => thread safe code
  - Sadly doesn't "just work" in practice. Damn.
- Loads of different research lines
  - Lots of EDSLs, prodding GPUs all different ways
  - Compiler extensions, libraries.
- Particularly excited by Nested Data Parallelism
  - Semi-implicit approach
  - Sounds like it's still a way off, but looking promising

What happens to games without functional languages?

- Suspect:
  - Imperative code written in more functional style (no bad thing)
  - Wide use of functional helper libraries
  - Suffer the pain

- **This is probably the default position**

- Examples
  - Intel threaded building blocks
    - Nested data parallelism
  - FC++
    - Functional language in C++ templates
  - SPURS on PS3
    - Job manager system
  - Gramps, OpenCL (arguably a language), CUDA, BrookGPU, Ct, etc, etc
  - Loads of others, including hand-rolled solutions

Task/Data/Explicit/Implicit

- Current consoles/PCs
  - Mostly explicit task scheduling
- GPUS
  - More data parallelism than task parallelism
  - “Semi-implicit”
- Always room for both, but:
  - Shades of grey in-between data/task parallelism also important
  - Just not enough tasks to schedule after a while -> data parallelism will become more important
  - Will have to be more implicit in the future
- Manual parallelism a pain in the ass, but manual data parallelism is (arguably) even more so.

A case in point

- Sparse matrix traversal code in Ct
- (Sorry Intel! Nothing personal)
- Nicked straight from Ct white paper

Sequential:

```
for (col = 0; col < col_num; col++) {
    for (elt = ColP[col]; elt < ColP[col+1];elt++)
    {
        int row = RowIdx[elt];
        …touch elements of A[row][col]…
    }
}
```

A case in point

- Promise not deliberately picking on Ct...
- The rest really aren't great either.

Data parallel:

```
TVEC<F64> sparseMatrixVectorProducSC(
  TVEC<F64> Values,
  TVEC<I32> RowIdx,
  TVEC<I32> ColP,
  TVEC<F64> v)
{
  TVEC<F64> expv = distribute(v,ColP);
  TVEC<F64> product = Values*expv;
  product = product.applyNesting(RowIdx, ctIndex) ;
  TVEC<F64> result = product.addReduce();
  return result;
}
```

Life is hard enough

(that’s enough misery for now)

Haskell or C++?

FIGHT!

Haskell – my prime candidate

- It’s a heavy weight
  - Amazing type system.
  - EDSLs / monads / laziness / streams
  - Purity + multithreading research
  - Good FFI design (implementation still needs work)
  - (You know the rest)
  - Kicks ass!

- Other language contenders
  - Haskell variants and extensions (DDC?)
  - Lisp
  - OCaml
  - Erlang
  - D, F#, ...

Why C++ is a tough nut to crack

- It is an **exceptional** language
  - Don’t believe the hype, C++ is alive and well
  - Has a huge monopoly in games
  - C# often used for tools
  - Lua/python/homebrew used for scripting
- Can glue anything to anything
  - Cross platform compilation
  - Can drop to assembly
  - And mix that with high level concepts
- Performance is everything
  - Overheads are always controllable

No really, **performance is everything**

- Code quality matters
  - Exception handling is uncommon
  - Focus on data: structure packing, alignment etc
  - Avoid unnecessary virtuals, branches, etc
  - Mem leaks, general instability => death
  - Hand optimised inner loops, etc
- Limited use of dynamic memory
  - malloc() is expensive
  - Fragmentation is a major problem
- **Worth significant development sacrifices**
  - No or hand-rolled garbage collection
  - No auto meta data (reflection, serialisation, ..)
  - Wide open to painful bugs
  - Large compilation and iteration time
  - Large projects become a significant burden
  - ‘Explicit everything’

Development model breaking down?

The 4 riders of the apocalypse:

- Timely development
  - Projects die under the weight of their own ambition
- Complexities of scale
  - Projects die under their own weight
- Execution speed
  - Projects die when everything else kicks their ass

- And even then, they sometimes die because they aren’t fun enough (sneaky rider #4)

Evidence of other functional rebellions

- OCaml at Definition6 – Chris Hecker
- Lisp (GOAL) at Naughty Dog
- CG / HLSL
  - “Referentially transparent” languages
  - (Note how well they optimise)
- Tim Sweeney at Epic
- Lua, (maybe it counts...)

- Not much else to my knowledge

Why aren’t productivity benefits enough?

- It is close to call, but probably no.
- Would put money on never seeing a leading game title written in C#
- Scripting systems + sensible development is enough in practice

Haskell – more things to like

- “Why functional programming matters”
  - New kinds of glue
  - Predicts functional languages will scale well (?)
- ghc-core
  - Essential to see the internals
- Mature extensible compiler
- Expect type system to be helpful
  - Plenty of suitable problems
- Haskell plug-in
  - Dynamic code!
- Bright, thriving community
  - Important! Code doesn’t just write itself.

Haskell – the road to go

- <u>Parallelism</u>
- How to deal with space leaks and inappropriate laziness?
- FFI / compilation
  - Cross platform support more than gcc backend
  - Target hard architectures: Larrabee, Cell SPUs. They will be unforgiving.
- Custom runtimes
  - Should be exposed and easy to integrate/modify.
    - People should and will customise it
  - More control over memory
    - ‘Strategies’ are good, verbose code would kill benefits
  - GC “applicable for games”
- SIMD support
- Worry more about locality of memory
- Performance compilation warnings?

- (yeah, it’s a bit wishy washy at the moment. I’m on it.)

Haskell rules the world.
So now what?

The fun bit ☺

- Apply Haskell to game coding problems!

- Wide variety to choose from:
  - Rendering: <insert huge topic>
  - Physics: rigid bodies, fluids, vehicles, ...
  - FX: particles, volumetric fx, ...
  - Animation: IK, rigging, blending, authoring, ...
  - Geometry: collision detection, authoring, ...
  - AI: <insert large topic>
  - Navigation: path finding, obstacle avoidance, ...
  - Tools: distributed build systems, asset management, ...
  - Iterative development: coding ‘live’, immediate feedback, ...
  - ...

Slimming the list down a bit…

- A quick overview of a handful
  - Really quite an arbitrary list
  - Loads of problems I'd like to explore in Haskell

- Rendering
  - Everyone likes pretty pictures.
  - Still a huge topic
- State machines in AI
- Asset conditioning pipelines

Example #1: Rendering in Haskell

- Haskell OpenGL bindings very neat but not really very functional
- OpenGL is <u>not</u> a good model of what actually happens
- Don't think side effects, think instruction buffer
  - Play/read points
  - Like audio buffer
  - Instruction *stream*
- <u>Re-think *functional* graphics API in Haskell</u>
- A starting point:
  - doFrame :: [Instruction] -> IO ()
- Model for thinking about other problems in games. Be suspicious of IO.

Rendering as...

- Series of transformations from scene description to instruction buffer
- High and low level optimisation potential
- Like a compiler!

Some common high level rendering optimisations

- Stream rendering
- Combining together similar shaders
- Shader stitching/generation
- Using common vertex declaration
- Forward vs deferred rendering

- All quite time consuming to change

The thing about optimisation is..

- People rarely factor in the lost opportunity cost!
- Time == potential for optimisations
- High level rendering optimisations time consuming but important
- Goal: make high level optimisations less time consuming without sacrificing performance.

State machines

- Basic state machines are switch statements.
- Quickly get out of hand.
- Possible next steps:
  - States as objects
  - Hierarchical state machines
  - Component oriented systems
- Lots of verbose code.
  - Pros and cons. No especially good solution.

State machines

- Haskell?
  - Don’t know!
  - Exercise for reader ☺

- Weapons at hand:
  - Monads
  - Algebraic types
  - Functional glue

Summary

- Assertions:
  - Games community would adopt Haskell immediately give demonstrable Killer App
  - Parallelism could be Killer App
  - Games would present interesting problems for Haskell community
  - Game hardware would present interesting hardware challenges for functional languages.

- Please draw own conclusions on the above.

Thank you

- Let me know your thoughts!

- Drop me a line for links to any referenced material

- sam@palgorithm.co.uk
- Homepage: http://palgorithm.co.uk

- Thanks to Chris Hecker and Pål-Kristian Engstad for their input.