Writing a 3-D Multiplayer Game with Kawa and JMonkeyEngine

Per Bothner (Kawa) <per@bothner.com>

mikel evins (the Fabric) <mevins@me.com>

http://www.gnu.org/software/kawa/

JavaOne October 2015; San Francisco [CON2111]

Who are we?

- **Per Bothner** is the lead and main implementer of the Kawa project.
- Previously: JavaFX Script language and compiler, Java/JavaScript bridge for JavaFX WebEngine (at Sun/Oracle); Gcc steering committee; GCJ (AOT for Java using Gcc), libg++ (GNU/Cygnus); DomTerm; emacs term mode; Qexo XQuery implementation; ...
- **mikel evins** is developing The Fabric game. He also writes science fiction.
- Previously: Dylan language; Bard language; Delectus personal database; folio library for Lisp; educational iOS games (Habilis and LearningTouch); Mousechief Games; embedded Lisp system software for Secure Outcomes; AllegroGraph; HyperCard; AppleScript; Newton OS; ...

Introduction

- "The Fabric" is a far-future MMORPG [Massively multiplayer online role-playing game]
- A MMORPG is typically a major multi-million dollar project
- Typically developed using a traditional edit/compile/debug cycle.
- The Fabric is basically developed just by mikel
- It is written 100% in the Kawa dialect of Scheme
- Uses the JMonkeyEngine gaming engine, which is written in Java
- This talk looks at how this is possible, and experience gained

DEMO

Demo Fabric features

Factors in language choice and design

- Interactive and incremental development
- Syntax: composability, extensibility, avoiding boiler-plate
- Language specification, standards, documentation
- Java integration
- Performance: execution speed, start-up, compilaton, memory use
- Pragmatics: tools, building, deployment

Interactive and incremental program development

- No explicit required compile step
- Avoid needless boilerplate; types not required
- While the program is running, new code can be added.
  Existing code can be replaced.
- dynamic: eval, repl, create functions/classes at run-time
- Values can be modified and functions called from a REPL.
- Seamless transition: exploration, prototyping, testing, development, optimization

The Lisp family of Languages

- Includes Common Lisp, Scheme, Dylan, Clojure and variations
- Decades of history, experience, research (from 1958)
- Originally tied to AI and “symbolic processing”
  Pioneered REPLs, GC, lambda, more
  (now being “re-discovered” by others)
- Emphasized interactive development
  Progressively eliminate boilerplate - types optional; no main program
  Designed to make the programmer fast - easy to go from vague idea to running code

Expression languages

- Expression-oriented: A statement is just an expression whose value is ignored.
  Where Java has statements (loops, `if`, `switch`, `try`) — Lisp uses expressions.
  Equivalent of Java's `break` takes a return value.
- Expressions can composed more easily.
  Can also be moved around more easily.

Lisp prefix notation

- Syntax consistently uses pathenthesised prefix notation:
  `(OP ARG1 ARG2 ... ARGn)`
  `OP` can be:
  - a procedure (`+`, `sqrt`, `length`)
  - an expression that evaluates to a procedure
  - control structure (`if`, `lambda`, `do`, `let`)
  - user-defined macro

  Kawa also allows `OP` to be:
  - a class or type name (in a constructor expression)
  - an array, list, vector, or string (indexing)

Benefits of Lisp syntax

- Using expressions and a simple regular prefix syntax simplifies:
  - Treating programs as data (I/O)
  - Constructing, manipulating, and analyzing programs
  - Macros and syntactic extension
  - Language extensions (DSLs)
  - No reserved identifiers
- If you notice repetitive code, abstract it out with macros

Lisps for the JVM

- For Fabric, mikel tried out 3 Lisp-family languages for the JVM:
  - Kawa (an implementation of Scheme, with extensions)
  - ABCL (an implementation of Common Lisp)
  - Clojure

What is ABCL?

- “Armed Bear Common Lisp” is a full implementation of Common Lisp on the JVM.
- Has both an interpreter and a compiler.
- The Common Lisp language was standardized by ANSI in 1994.
- Includes CLOS (CL Object System), a very flexible and dynamic object system.

What is Clojure?

- Not compatible with other Lisp-like languages.
- Strong support for parallel and side-effect-free programming using immutable data structures.
- Strong eco-system and specialized tools.
- Was released in 2007.

What is Kawa?

- An implementation of the Scheme language
  Implements the latest Scheme standard (R7RS from 2013)
  (- except for full continuations - which are in-progress)
  Many extensions and conveniences for JVM users
- The oldest still-active compiler-based language for JVM (beside Java): 1996
- A toolkit for language implementation, including a compiler that produces efficient JVM bytecode.
- An interactive programming system

Some Kawa language features

- the repl
- comprehensive Java interop
- JavaFX support
- separate compilation
- supports Android
- shell programming features
- implements many semi-standard extensions (SRFIs)

Java/JVM integration

- Any “serious” JVM-based language lets you define and access JVM classes, members, and plain-old-Java-objects.
- Though sometimes there are limitations, complications, or inefficiencies

  Example: JMonkeyEngine3 requires you to subclass its library classes.

  No straight-forward way to do this in either Clojure or ABCL. (There are work-arounds.)

  With Kawa, it's easy.
- ABCL's CLOS object system system is very powerful and dynamic

  Hence you can't directly map CLOS methods and fields to JVM members
- Clojure has multiple ways to define types and interfaces; most flexible is gen-class

Kawa class definition

- A Kawa "simple" class compiles very directly to a plain Java class or interface.
- Syntax is based on Common Lisp.

```
(define-simple-client FabricClient
    (SimpleApplication ActionListener) ;; super-types
  (@Serializable) ;; annotations are supported
  ;; fields
  (username::String init: #!null)
  ;; init method - calls init-client procedure
  ((simpleInitApp) (init-client (this)))
...)
```

- The non-simple `define-class` supports true multiple inheritance.

Property and method references

- Kawa doesn't distinguish Kawa object from Java objects.
- “colon operator” `X:N` gets property named `N` from object `X`.

  `doc:buffer` — *get field*

  `("abab":indexOf "ab" 1)` — *call method*

  `Color:GREEN` — *get static field*

  `(BigDecimal:valueOf 123456 2)` — *call static method*
- Hides field vs getter method difference:
  `uri:raw-authority` — *same as*
  `(uri:getRawAuthority)`
- All of these can compile to same bytecode as Java, assuming types are known to compiler.

object creation

- Type name does double duty as constructor function:

  `(T x y)  ;; Java: new T(x, y)` *or* `T.valueOf(x, y)`
- Keyword arguments are translated to setting of fields or set methods:

```
(RadioButton screen "CannonButton"
             (compute-cannon-button-origin screen)
             (compute-cannon-button-size screen)
             text: "Cannon"
             fontSize: 20
             textAlign: Align:Center
             textVAlign: VAlign:Bottom)
```

arrays and objects with children

- Arrays, lists, vectors are created with the pattern:

  ```
  (TYPE x1 x2 ... xN)
  ```

  For example:

  ```
  (int[] 3 4 5 6)
  (vector 3 4 5 6)
  (java.util.ArrayList 3 4 5 6)
  ```

  Generalized to tree nodes with "child" values:

  ```
  (WeaponButtonGroup screen "WeaponGroup" state: state
                     (RadioButton screen "CannonButton" ...)
                     (RadioButton screen "ImpulseButton" ...)))
  ```

- Calls `add` method (or in this case `addButton`)

Kawa Modules

- Each source file defines a namespace aka a “module”.
- A module contains definitions (named classes, functions, macros, variables, aliases) and top-level actions.
- Some definitions are exported.
- Another module (or the REPL) can import a module.

  This creates aliases for the module's exported definitions.
- Imported definitions can be re-exported.
- A variable can only be assigned to in its defining module.
- Easy for compiler to map name to definition and assignments.

  Simplifies data-flow analysis, type inference, error checking.

Module implementation

- Module name is a fully-qualified class name
- Importing a module searches for the class or a corresponding source file
- Can optionally specify a source file, or generated from module name
- Each exported definition gets a static field, possibly with annotations
- To import a module, the compiler scans the static fields
- No “module database” or “namespace database” needed
- Simple, powerful, and efficient

DEMO - update in place

Performance - execution speed

- Kawa prioritizes run-time performance and low overhead
- Speed is similar to Java or Scala
- Performance helped by type inference, data-flow analysis, and optional type specifiers
- Kawa compiler does custom analysis and code generation for many builtin functions
- ABCL and Clojure are slower then Kawa, though faster than many other “dynamic” languages

multiple threads and side effects

- Clojure provides sophisticated side-effect-free data structures (collections and more)

  Useful for multi-threaded programs; avoids synchronization and races

  Also avoids some bugs
- However, they do require more time *and* more space.
- Kawa and ABCL also encourage pure side-effect-free programs.

  But not enforced by language - or data structures
- In practice, easier to get good performance with Kawa.
- SIMD parallelism (Java 8 streams or APL-like arrays) may be an easier path to multi-threaded performance.

Performance - start-up speed

- Clojure and ABCL have notorously bad start-up times.
- Starting Kawa and loading Fabric from jar 0.5s.
- Starting Kawa and loading Fabric from source 1.7s.
- Compiling source (7500 lines) to jar with ant 5s.

  Whole shebang (Fabric, Kawa, JMonkeyEngine, assets) is 35MB.

Startup issue - number of classes

- Kawa compiles each function to a separate method.
  (Except nested functions, which may require “frame” classes.)
- Closure compiles each function to a separate class.
  Extra classes means bigger `jar` files, slower startup.

Startup issue - loading each function at startup

- Clojure initializes the data structure for each accessible function at load time.
- Kawa only enters a (classname, fieldname) entry in the initial symbol table.
  The actual function object and its class are loaded lazily as needed.

Performance - memory use

- Lazy code loading: saves memory of unused classes and functions
- Clojure no-side-effect data structures have higher overhead.
  Side-effect-free vectors require more memory than plain arrays.
- Because CLOS classes don't map so directly to JVM classes, expect ABCL object and classes to have more overhead.

Re-loading code in Kawa

- Kawa does a lot of type propagation and inlining.
  Great for performance, but inconsistencies possible when reloading a function or a module.
- Clojure is better in this respect; ABCL is very robust.
- Solution: define an “interactive mode”, where we do less inlining and more indirection.
- Future: tracking of dependencies and automatic recompilation.
  Hardest for class changes, but indirection can handle most changes (except `extends` or `implements`).
  (The reason it's hard: co-existing of new and old instances.)
- This is a work-in-progress.

Building and tooling

- Most Clojure projects use the Leiningen tool
  Leiningen can create, compile, test, run projects. It can fetch dependencies
  Other powerful tools also available
- Kawa lacks anything similar - but doesn't need it
  `kawa` commmand is a simple wrapper for `java -jar kawa.jar`
  For building use whatever tools you like: `ant`, `make`, `gradle`
  For deploying use `jar`, `zip`, `tar`
  Nothing new to learn

mikel on clojure and its ecosystem

“Clojure is embedded in a large and complex ecosystem, and choosing to use Clojure essentially means committing to that entire ecosystem.
The majority of learning Clojure is not learning the language;
it’s learning clojars and leiningen and nrepl and boot and ring and om and maven and datomic and ...”

The Fabric development environment

- `kawa.jar`
- The JMonkeyEngine jars
- `ant`
- Emacs with the `quack.el` package

No main function

- With Java or Clojure you create a separate `main` function.
- Kawa top-level actions are compiled to a `run` method.
  When a module is loaded, its `run` method is invoked.
- Compile option `--main` generates `main` method which calls `run`.

Deployment

- With Kawa it's simple:
  - You ship the `kawa.jar`
  - You ship your application, as either jars or source files
  - Use same tools as Java - for example `ant` or JavaFX packager

standards and specifications

- A specification enables multiple implementations
- Encourages text-books, classes, research
- Encourages stabilty and compatibility over time
- Helps separate bug from feature :-)
- However, standards can be very political and committees can slow progress
- Clojure follows no separate standard
- ABCL implements the Common Lisp standard.
- Kawa implements the 2013 R7RS Scheme sepecification

mikel's summary of Scheme and Kawa

- Kawa is a particularly convenient implementation of a particularly convenient standard.
- Scheme is a convenient standard because it’s small and mature.

  Its standards are well understood, flexible, and adaptable.

  It’s community, though small, is evergreen. Its partisans continue to contribute to the advancement of language design.
- Kawa is a convenient implementation because it makes the cost of using a non-Java language about as low as it possibly can be.

  A single jar on your classpath is all you need to incorporate Kawa into your project.

Bottom line - why mikel chose Kawa

- Kawa's startup time was far better
- Kawa’s performance was generally better
- Kawa made it much easier to work with JMonkeyEngine than either Clojure or ABCL:
  - Inheriting from Java classes is much simpler
  - Working with a traditional imperative, object-oriented library is easy in Kawa or ABCL, but more awkward in Clojure
- Kawa toolchain is much simpler than Clojure’s

Bottom line - other considerations

- ABCL wins on interactivity, expressiveness, and a robust REPL
- Cjojure wins on size of community
- Cjojure wins on elegant side-effect-free programming
- ABCL and Kawa win in being standards-based
- Clojure wins in tool support

Questions and answers

(More slides about Kawa after, if there is time.)

Soon: patterns

- Variable names generalized to patterns in parameter lists and definitions:
  ```
  (! [x y] (make-a-list))
  ```
  Succeeds if `(make-a-list)` returns a list of size 2.
- Conditional patterns use '?':
  `(if (? pattern value) action-if-match action-if-nonmatch)`
- Common use case:
  `(if (? x::T val) (use-as-T x) (not-a-T))`
  This simplifies `instanceof` tests.

process literals

- Simple syntax for creating and running a process:
  `(define p1 &`{date --utc})`
- When you convert a process to a string, you gets its standard output:
  `(->string p1)` ⇒ `"Mon Oct 26 18:54:55 UTC 2015"`
- Simple process substitution:
  `&{echo The time is &`{date --utc}}`
- The `in:` specifies standard input as a string (“here document”):
  `&`[in: "Foo\n"]{tr a-zA-Z A-Za-z}` ⇒ `"fOO"`
- A pipe is just a combination of these ideas:
  `&`[in: &`{date --utc}]{tr a-zA-Z A-Za-z}`

xml literals

- An “XML” literal is a '#' followed by an XML element:
  `#<p>The result is <b>final</b>!<p>`
- This evaluates to a DOM `Element` value.
- You can substitue the value of an expression (a string or a `Node`):
  `#<em>The result is &{result}.</em>`

APL-style arrays

```
(define v1 [2 3 5 7 11 13])
(v1 2)  ⇒ 5
(v1 [3 1])  ⇒ [7 3]
(v1 [4 >=: 2])  ⇒ [11 7 5]
```

- Supports assign/replacment of slices.
  Can change the side (insert/delete)
  Working on generalizing this to APL-style multi-dimensional arrays.

splices

- The syntax `@lst` is used in a call.
  It evaluates `lst` to a sequence (list or array).
  Each element of `lst` becomes a separate argument
- If `lst` is `[3 4 5]`:
  `(+ @lst)` ⇒ `(+ 3 4 5)` — i.e. reduction.
- Works well with array/list constructor:
  `(int[] @lst 9 @lst)` ⇒ `[3 4 5 9 3 4 5]`