SHIRAKUMO

Nicolas Hafner
@Shinmera

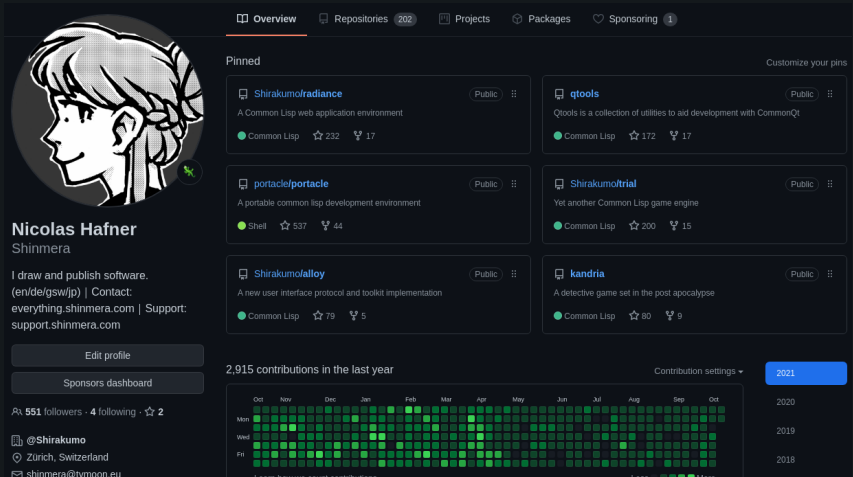
https://shinmera.com https://shirakumo.org

About me

- Founder of Shirakumo

About me

- Founder of Shirakumo
- Open source maintainer



About me

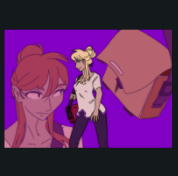
- Founder of Shirakumo
- Common Lisp library maintainer
- Artist, etc.

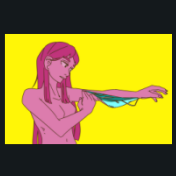






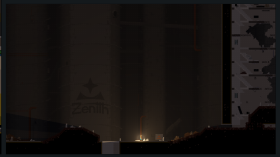
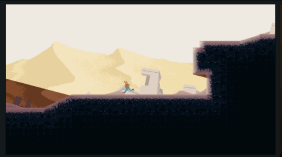




Background

- Trial game engine and Kandria
- Full-stack lisp development
- Shipped Eternia: Pet Whisperer on PC


Why Common Lisp

- Dynamic nature attractive for game iteration
- CLOS protocol design very comfortable
- Restarts great for interactive development
- Can change game entirely *while it's running*

Why Common Lisp

- Dynamic nature attractive for game iteration
- CLOS protocol design very comfortable
- Restarts great for interactive development
- Can change game entirely *while it's running*

We'll look at:

- Mixins / CLOS
- Restarts
- Optimization
- Garbage Collection

Mixins / CLOS

- Encapsulate behaviours into small classes
- Generic functions + mixins comparable to ECS
- Fully runtime redefinable
- Can even change class of an existing instance (!)

Mixins / CLOS

```
(defclass event () ())
(defclass tick (event) ())

(defclass entity () ())
(defclass player (entity) ())
(defclass enemy (entity) ())

(defgeneric handle (event object))
```

Mixins / CLOS

```
;; Catch-all
(defmethod handle ((event tick) (object entity)))

;; Player input handling
(defmethod handle ((event tick) (object player))
  (print :player)
  (when (pressed :left) ...)
  (when (pressed :right) ...))

;; Enemy AI logic
(defmethod handle ((event tick) (object enemy))
  (print :enemy)
  (when (see 'player) ...))

;; Sample call
(handle tick enemy)
=> :enemy
```

Mixins / CLOS

```
;; A new emitter class with a flickering light
(defclass emitter () ())

(defmethod handle :after ((event tick) (object emitter))
  (print :emitter)
  (update-intensity ...))
```

Mixins / CLOS

```
;; A new emitter class with a flickering light
(defclass emitter () ())

(defmethod handle :after ((event tick) (object emitter))
  (print :emitter)
  (update-intensity ...))


;; Update the class ↓
(defclass enemy (emitter entity) ())

;; Sample call
(handle tick enemy)
=> :enemy :emitter
```

Mixins / CLOS

An example from our actual code-base

```
(defclass player (alloy:observable stats-entity
                  paletted-entity animatable
                  profile ephemeral inventory)
  (...))
```

Mixins / CLOS

An example from our actual code-base

```
(defclass player (alloy:observable stats-entity
                  paletted-entity animatable
                  profile ephemeral inventory)
  (...))


(length (compute-class-precedence-list 'player))
=> 32
```

Problems

- Class order can have surprising consequences
- Protocols need to be carefully designed
- Dispatch overhead can be significant

Restarts

- Specify ways to recover from errors
- Debugger can then use restarts to continue
- Surrounding dynamic context can do this, too

Restarts

- Specify ways to recover from errors
- Debugger can then use restarts to continue
- Surrounding dynamic context can do this, too

```
Restarts:
 0: [ABORT] Don't handle #<TEXT-ENTERED {1002C11053}> in #<UI-PASS  UI-PASS {1021B2BE43}>.
 1: [SKIP-EVENT] Skip handling the event entirely.
 2: [ABORT] Don't handle #<TEXT-ENTERED {1002C11053}> in #<WORLD 280 units {1021356CD3}>.
 3: [DISCARD-EVENTS] Discard all events.
 4: [RESET-RENDER-LOOP] Reset the render loop timing, not catching up with lost frames.
 5: [EXIT-RENDER-LOOP] Exit the render loop entirely.
 --more--

Backtrace:
  0: ((LAMBDA NIL :IN "/run/media/data/Projects/cl/kandria/cheats.lisp"))
  1: (PROCESS-CHEATS "e")
  2: ((:METHOD HANDLE :AROUND (EVENT T)) #<TEXT-ENTERED {1002C11053}> #<UI-PASS  UI-PASS {1021B2BE43
  3: ((:METHOD HANDLE :AROUND (EVENT UI-PASS)) #<TEXT-ENTERED {1002C11053}> #<UI-PASS  UI-PASS {1021
  4: ((:METHOD HANDLE (EVENT EVENT-LOOP)) #<TEXT-ENTERED {1002C11053}> #<WORLD 280 units {1021356CD3
  5: ((SB-PCL::EMF HANDLE) #<unused argument> #<unused argument> #<TEXT-ENTERED {1002C11053}> #<WORLI
```

Restarts

```
(defmethod render :around (object target)
  (restart-case (call-next-method)
    (abort ())
    (retry ()
      (render object target))))
```

Restarts

```
(defmethod render :around (object target)
  (restart-case (call-next-method)
    (abort ())
    (retry ()
      (render object target))))


(defun main ()
  #+release
  (handler-bind ((error (invoke-restart 'abort)))
    (start-game))
  #-release
  (start-game))
```

Restarts

- Debugger pauses affected thread
- Can fix underlying problem while running
- Recompile anything, change any variable!
- Then resume from a fitting restart

Optimisation

- Standard dynamic language issues apply
- Dynamic by default means indirection and checks
- Generic function dispatch overhead significant

Optimisation

- Standard dynamic language issues apply
- Dynamic by default means indirection and checks
- Generic function dispatch overhead significant

but...

- SBCL can infer a lot of type information
- Programmer can declare missing types
- Assembly of generated functions can be inspected
- Compiler is customisable from within Lisp
- New work being done to speed up dispatch (Strandh et al.)

Optimisation

```
(disassemble
  (lambda (x)
    (* x x)))
; disassembly for (LAMBDA (X))
; Size: 33 bytes. Origin: #x54A24624
; 24:       498B5D10         MOV RBX, [R13+16]
; 28:       48895DF8         MOV [RBP-8], RBX
; 2C:       488BD6           MOV RDX, RSI
; 2F:       488BFE           MOV RDI, RSI
; 32:       FF14251801A052   CALL QWORD PTR [#x52A00118]
; 39:       488B75F0         MOV RSI, [RBP-16]
; 3D:       488BE5           MOV RSP, RBP
; 40:       F8               CLC
; 41:       5D               POP RBP
; 42:       C3               RET
; 43:       CC10             INT3 16
```

Optimisation

```
(disassemble
  (lambda (x)
    (declare (type (unsigned-byte 16) x))
    (declare (optimize speed (safety 0)))
    (* x x)))
; disassembly for (LAMBDA (X))
; Size: 16 bytes. Origin: #x5365AB36
; 36:       48D1FA           SAR RDX, 1
; 39:       480FAFD2         IMUL RDX, RDX
; 3D:       48D1E2           SHL RDX, 1
; 40:       488BE5           MOV RSP, RBP
; 43:       F8               CLC
; 44:       5D               POP RBP
; 45:       C3               RET
```

Garbage collection

- Dynamic languages need a GC
- SBCL provides a generational compacting GC
- GC pauses *are* a problem
- Pauses are stop-the-world, single-threaded

Garbage collection

- Dynamic languages need a GC
- SBCL provides a generational compacting GC
- GC pauses *are* a problem
- Pauses are stop-the-world, single-threaded

but...

- Problem can be mitigated with standard tech:
- Object pooling, static allocation, immutability
- Thanks to macros, boilerplate can be hidden!

Garbage collection

```
(defun color (r g b)
  (make-instance 'color r g b))

(define-compiler-macro color (r g b &whole form)
  (if (and (constantp r)
           (constantp g)
           (constantp b))
      `(load-time-value (make-instance 'color ,r ,g ,b))
      whole))
```

Garbage collection

```
(defun color (r g b)
  (make-instance 'color r g b))

(define-compiler-macro color (r g b &whole form)
  (if (and (constantp r)
           (constantp g)
           (constantp b))
      `(load-time-value (make-instance 'color ,r ,g ,b))
      whole))


(color 1.0 1.0 1.0)
=> (load-time-value (make-instance 'color 1.0 1.0 1.0))
```

Conclusions

- Significant challenges exist
- Effort has to be put in to optimise...
- especially regarding vectorisation and garbage
- Proper design is important and takes time

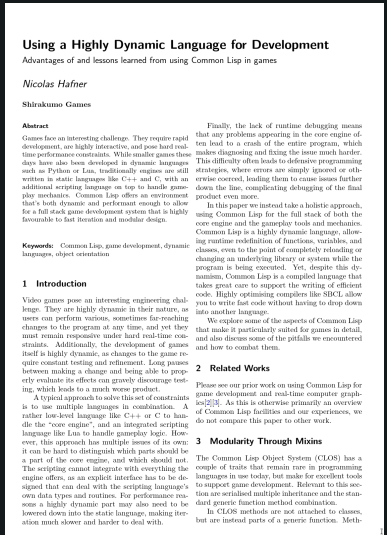
Conclusions

- Significant challenges exist
- Effort has to be put in to optimise...
- especially regarding vectorisation and garbage
- Proper design is important and takes time

but...

- Often the development speed benefits outweigh
- Iteration much faster with runtime recompilation
- The capabilities are all there

Paper available



Using a Highly Dynamic Language for Development

Advantages of and lessons learned from using Common Lisp in games

Nicolas Hafner

Shirakumo Games

Abstract

Games face an interesting challenge. They require rapid development, are highly interactive, and pose hard real-time performance constraints. While smaller games these days have also been developed in dynamic languages such as Python or Lua, traditionally engines are still written in static languages like C++ and C, with an additional scripting language on top to handle gameplay mechanics. Common Lisp offers an environment that's both dynamic and performant enough to allow for a full-stack game development system that is highly favourable to fast iteration and modular design.

Keywords: Common Lisp, game development, dynamic languages, object orientation

1 Introduction

Video games pose an interesting engineering challenge. They are highly dynamic in their nature, as users can perform various, sometimes far-reaching changes to the program at any time, and yet they must remain responsive under hard real-time constraints. Additionally, the development of games itself is highly dynamic, as changes to the game require constant testing and refinement. Long pauses between making a change and being able to properly evaluate its effects can gravely discourage testing, which leads to a much worse product.

A typical approach to solve this set of constraints is to use multiple languages in combination. A rather low-level language like C++ or C to handle the "core engine", and an integrated scripting language like Lua to handle gameplay logic. However, this approach has multiple issues of its own: it can be hard to distinguish which parts should be a part of the core engine, and which should not. The scripting cannot integrate with everything the engine offers, as an explicit interface has to be designed that can deal with the scripting language's own data types and routines. For performance reasons a highly dynamic part may also need to be lowered down into the static language, making iteration much slower and harder to deal with.

Finally, the lack of runtime debugging means that any problems appearing in the core engine often lead to a crash of the entire program, which makes diagnosing and fixing the issue much harder. This difficulty often leads to defensive programming strategies, where errors are simply ignored or otherwise coerced, leading them to cause issues further down the line, complicating debugging of the final product even more.

In this paper we instead take a holistic approach, using Common Lisp for the full stack of both the core engine and the gameplay tools and mechanics. Common Lisp is a highly dynamic language, allowing runtime redefinition of functions, variables, and classes, even to the point of completely reloading or changing an underlying library or system while the program is being executed. Yet, despite this dynamism, Common Lisp is a compiled language that takes great care to support the writing of efficient code. Highly optimising compilers like SBCL allow you to write fast code without having to drop down into another language.

We explore some of the aspects of Common Lisp that make it particularly suited for games in detail, and also discuss some of the pitfalls we encountered and how to combat them.

2 Related Works

Please see our prior work on using Common Lisp for game development and real-time computer graphics[2][3]. As this is otherwise primarily an overview of Common Lisp facilities and our experiences, we do not compare this paper to other work.

3 Modularity Through Mixins

The Common Lisp Object System (CLOS) has a couple of traits that remain rare in programming languages in use today, but make for excellent tools to support game development. Relevant to this section are serialised multiple inheritance and the standard generic function method combination.

In CLOS methods are not attached to classes, but are instead parts of a generic function. Meth-

shinmera.com/paper/gic21.pdf

Thanks for listening!

https://kandria.com