Benjamin Canou, Emmanuel Chailloux and Jérôme Vouillon
Laboratoire d'Informatique de Paris 6 (LIP6)
Laboratoire Preuves Programmes et Systèmes (PPS)

How to Run your Favorite Language on Browsers

The Revenge of Virtual Machines

WWW 2012, Lyon, France

Introduction

What ?

- You have a favorite language
- You have just designed or extended one
- You want to run it on a Web browser

Why ?

- To program a new Web app
- To program your client with the same language than your server
- To run an online demo of an existing app

How ?

- Use applets
- Write an interpreter in JavaScript
- Write a compiler to JavaScript

Or as we present in this talk:

- Reuse the language bytecode compiler
- Write a bytecode interpreter in JavaScript
- Write a bytecode to JavaScript expander

An experiment report:

- Project Ocsigen: use OCaml to code entire Web apps
- OBrowser: an OCaml bytecode interpreter
- js_of_ocaml: an OCaml bytecode expander

Retrospectively, a good approach:

- Reasonable time to obtain a first platform
- Good performance achievable
- Fidelity to language/concurrency models

Core techniques

Main method:

1. Make the bytecode file network compliant (ex. JSON array)
2. Choose/implement the representation of values
3. Write a minimal runtime and standard library
4. Write the main interpretation loop
5. Run tests and extend the library as needed

Possible improvements:

- Use core, well supported/optimized JavaScript control structures
- Use simple, array based memory representation
- Preliminary bytecode cleanup pass

Pros:

- Fairly simple architecture
- Debug/adjustments using step-by-step execution
- The original VM can be used as a reference
- Semantics and performance scheme preservation
- Acceptable performance

Cons:

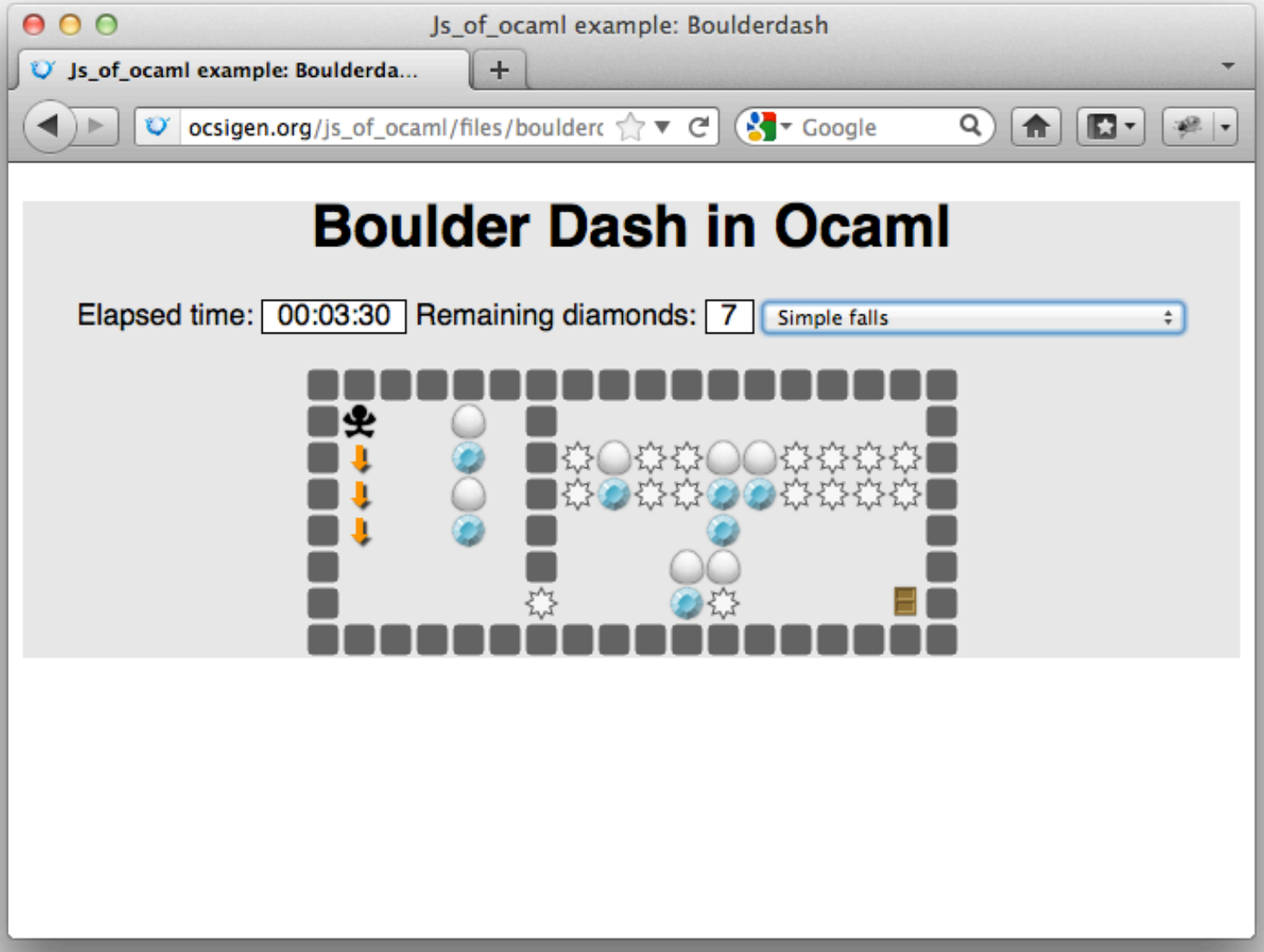
- Impossible to obtain great performance

Experiment: OBrowser

- Bytecode for the OCaml virtual machine
- A few weeks to develop and debug
- Performance < 10x JavaScript equivalents
- Runs existing OCaml programs, compiled with unmodified ocamlc
- Actually usable to start writing Web apps in OCaml

Demo: a *Boulder Dash* clone

- Uses the DOM and HTML elements for the interface
- Event handlers in OCaml
- Loads levels via HTTP requests
- In pretty standard OCaml style


Js_of_ocaml example: Boulderdash
Js_of_ocaml example: Boulderda...
ocsigen.org/js_of_ocaml/files/boulderc
Google
Boulder Dash in Ocaml
Elapsed time: 00:03:30 Remaining diamonds: 7
Simple falls

Bytecode expansion (1/3)

Basic method:

1. Reconstruct the control flow graph
2. Expand every basic block to a JavaScript function
3. Expand every bytecode to javascript instructions

Necessary improvements (for code size):

- Expression reconstruction
- Dead code elimination

Possible improvements:

- Finer (than function only) basic block mapping
- Inlining of run-time primitives
- Any compiler optimization

Pros:

- Potential great performance
- Easier to write than a from-source compiler
- Lower maintenance cost than a from-source compiler

Cons:

- More difficult to write than an interpreter
- Takes more time to see your first program running
- Easier to introduce bugs/more difficult to debug

Bytecode expansion (3/3)

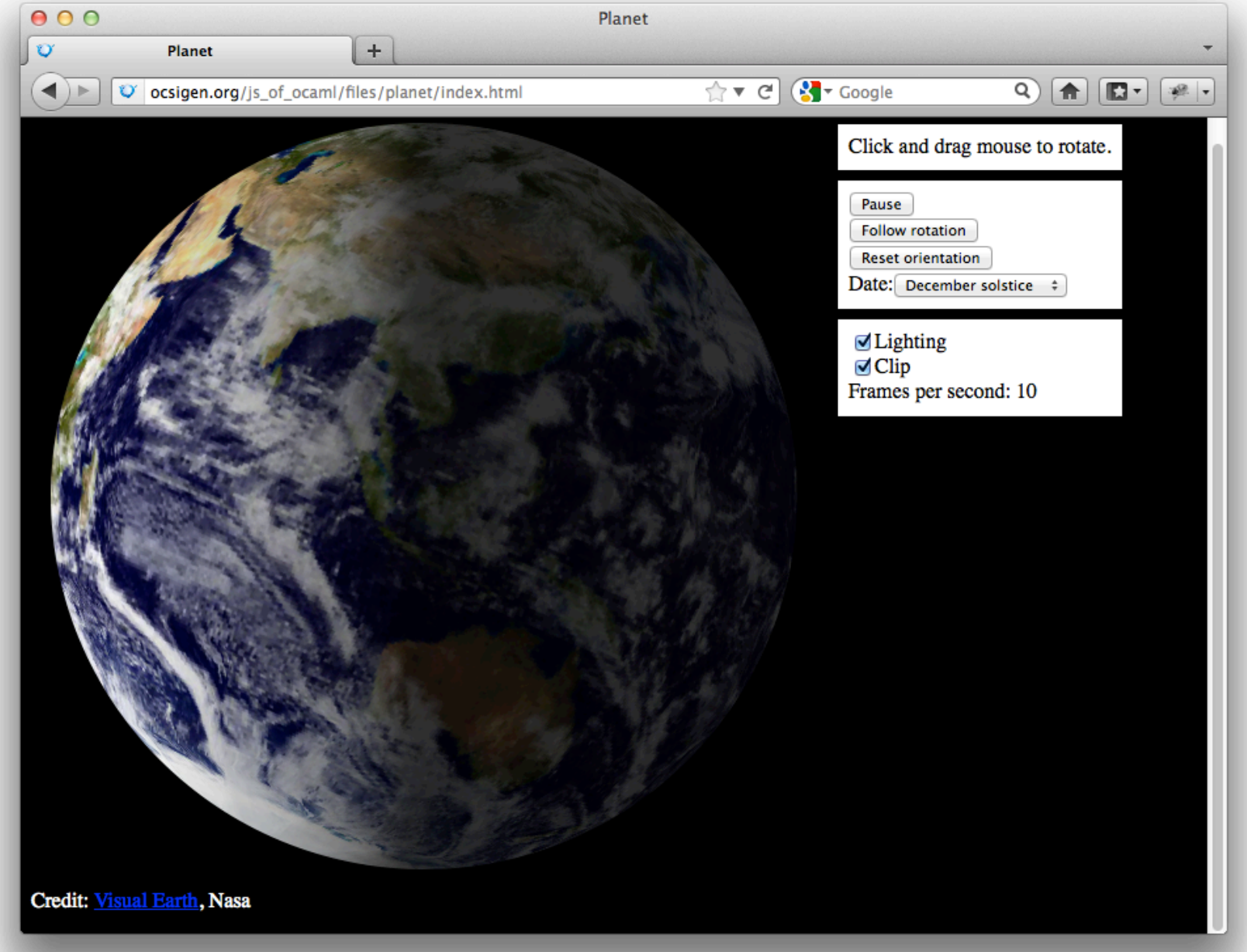
Experiment: js_of_ocaml

- Compiles OCaml bytecode to JavaScript
- Runs existing OCaml programs, compiled with unmodified ocamlc
- Excellent performance (as permitted by JavaScript)
- A few concessions to semantics preservation

Demos:

- Real time 3D software rendering
- OCaml compiler and interactive prompt
- An SMT solver in the browser !

Bytecode expansion (demo)



1. Write a bytecode interpreter
2. Start writing a bytecode expander if performance is required
3. When the interpreter is ready, start developing your Web app
4. Use the expander in production
5. Use the interpreter for debugging

Advanced topics

Breaking news: there is more to concurrency than the event loop !

Why ?

- Maybe the event loop is not ideal for your task
- To respect the original language semantics
- To be consistent with the server
- To increase modularity (plugging components without surprise)

Some examples:

- Preemptive threads: scheduling bytecode interpreter
- Background tasks: quota of bytecodes at each event loop turn
- Functional cooperative concurrency: language closures mapped to JavaScript event handlers

Simplified (untyped, low level) interoperability:

- Use the FFI of the language in a minimal way
- Write a set of primitives to operate on generic JavaScript objects
- Compose the primitives to simulate equivalent JavaScript code
  Example:

```
let getElementById id =
    call_method
      (eval "document")
      "getElementById"
      [| id |]
```

Compared to classical methods:

- No JavaScript to write
- Typing possibilities
- Optimizable by detecting calls to the primitives

Conclusion

- Successful approach for us (Ocsigen project)
  - We were able to lead client side experiments since 2006
  - Had the time to write a better backend in parallel
  - Now have a convincing solution with very good performance
- Probably the best approach for existing languages
  - Easier/more maintainable than a from-source compiler
  - Semantics preservation
  - Possibility to keep the concurrency model

`http://www.ocsigen.org/js_of_ocaml`

`http://www-apr.lip6.fr/~canou/obrowser/examples.html`

Ocsigen booth present at WWW 2012