My History with Papers

Papers We Love 2017

@jlongster

James Long

- Web/iOS contractor
- Mozilla
- PL nerd, attraction to Lisps, more recently ML languages
- Recent project: Prettier

Real Time Rendering of Light Scattering For Outdoor Scenes

Abstract

The field of computer graphics has given scientists, mathematicians, and others an opportunity to simulate realistic environments and scenarios without the cost or limitations of physical models. As technological power increases rapidly, researchers are able to model more complex and realistic scenes, such as atmospheres that include light scattering and aerial perspective. This paper focuses on the real-time rendering of atmospheric scattering and aerial perspective which give realistic depth cues and color. Because it is computationally expensive due to the calculation of several integrals, the calculations must be optimized and simplified to achieve results in real-time. After briefly summarizing the mathematical operations required for this effect, this paper will describe several variations and their corresponding implementations. These implementations will be analyzed thoroughly and judged according to their performance and quality.

James Long
Academic Literature Review
Information and Computer Sciences,

ATI Technologies Inc.

Rendering Outdoor Light Scattering in Real Time

Naty Hoffman
Westwood Studios
Email: naty@westwood.com

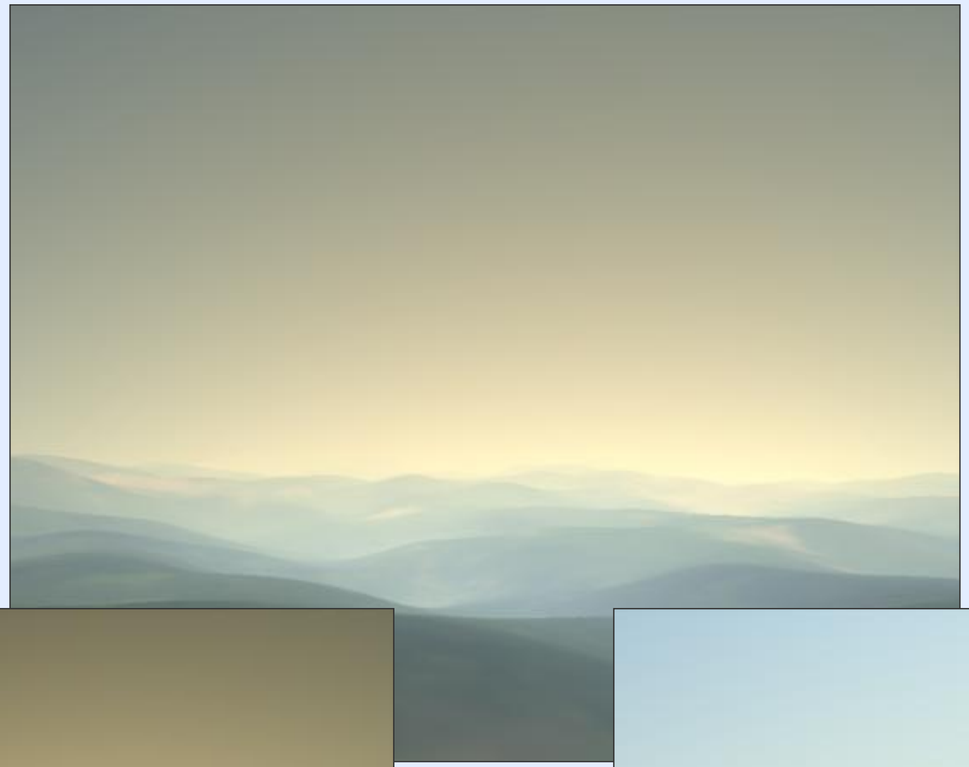
Arcot J Preetham
ATI Research
Email: preetham@ati.com

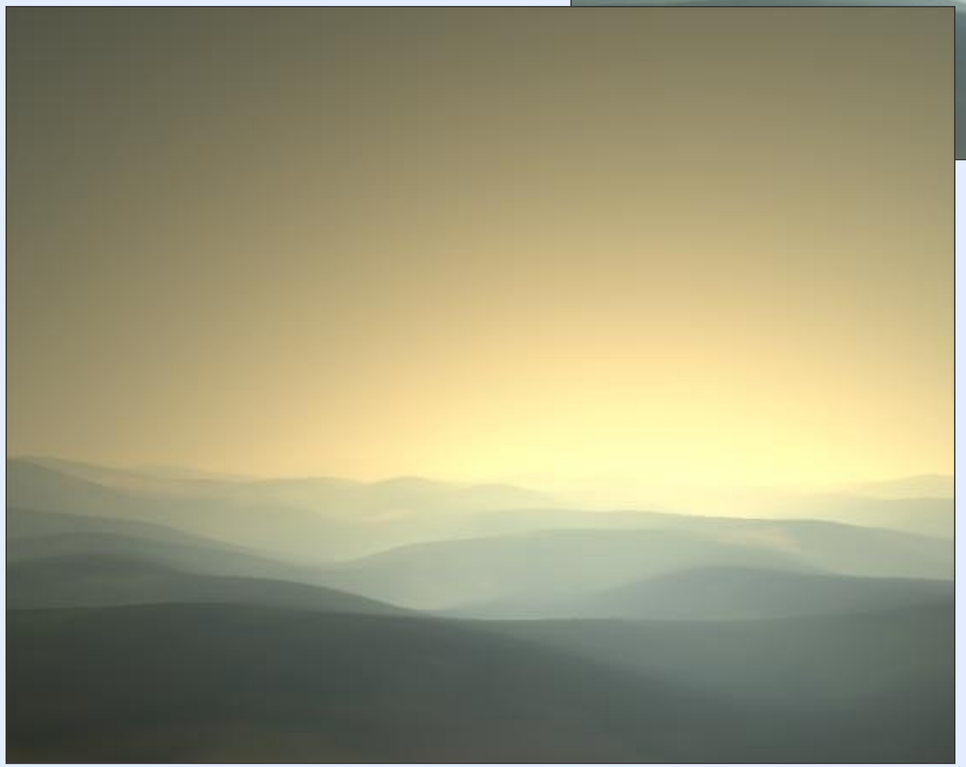
Introduction

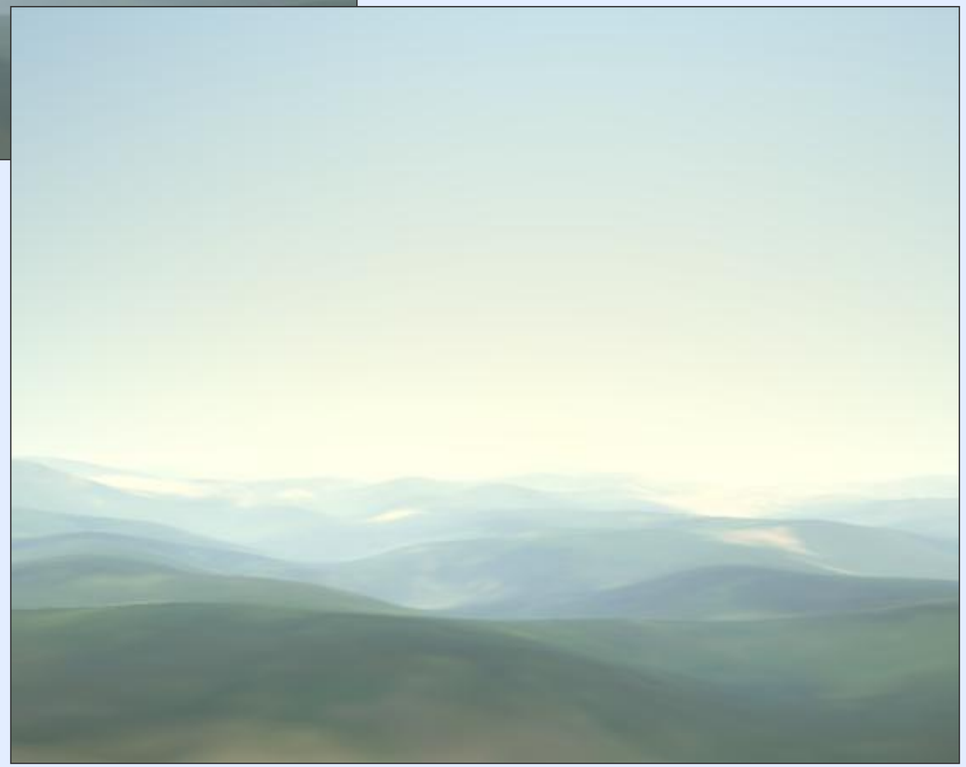
Atmospheric scattering of light is important in outdoor scenes. It changes sunlight from the pale red of dawn to the bright yellow of midday and back again. It determines the color and brightness of the sky throughout the day, and it cues us to the distance of objects by shifting their colors. All these effects vary not only based on time of day, but also depending on weather, pollution and other factors. On planets with different atmospheric compositions, these effects would differ significantly from those seen on Earth.

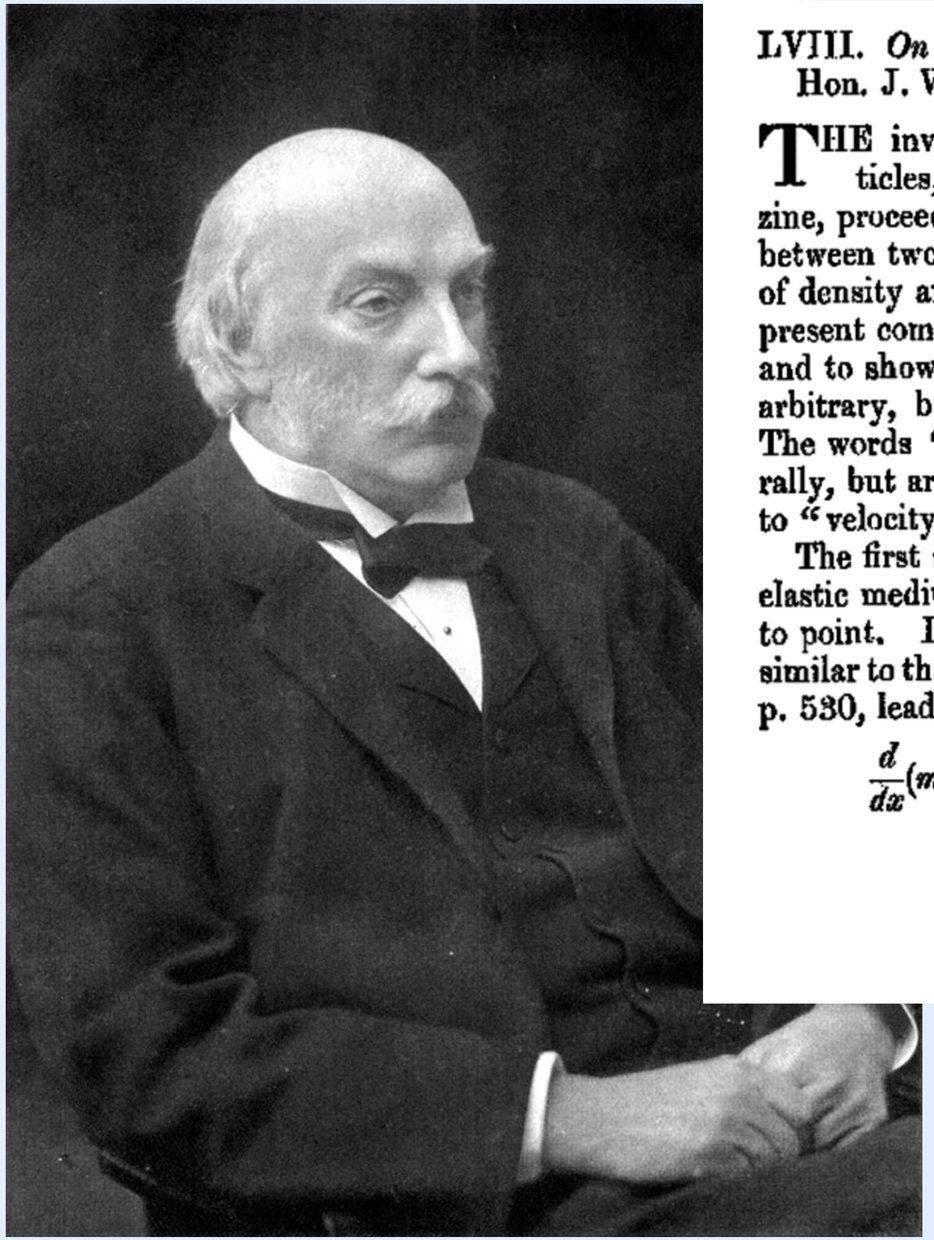
In this paper, we will explain the ways in which atmosphere affects light, including the

http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/ATI-LightScattering.pdf









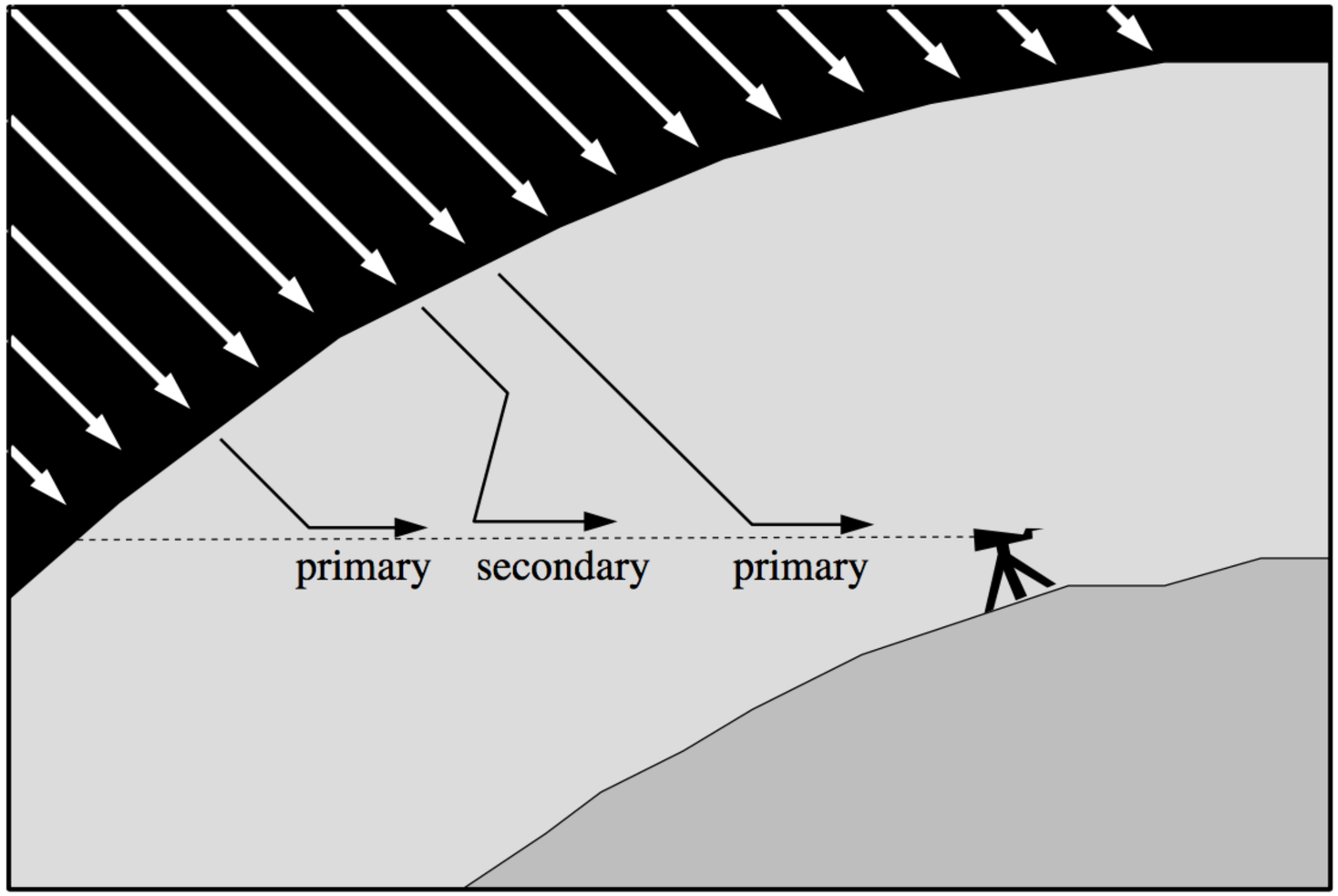
LVIII. *On the Scattering of Light by small Particles.* *By the* Hon. J. W. Strutt, *Fellow of Trinity College, Cambridge**.

THE investigation of the diffraction of light by small particles, contained in the February Number of this Magazine, proceeds throughout on the assumption that the difference between two media which differ in refractive power is a difference of density and not a difference of rigidity. My object in the present communication is to attack the problem more generally, and to show that the more special hypothesis is in no degree arbitrary, but forced upon us by the phenomena themselves. The words "density," "rigidity" need not be interpreted literally, but are used in a generalized sense analogous to that given to "velocity" and "force" in the higher mechanics.

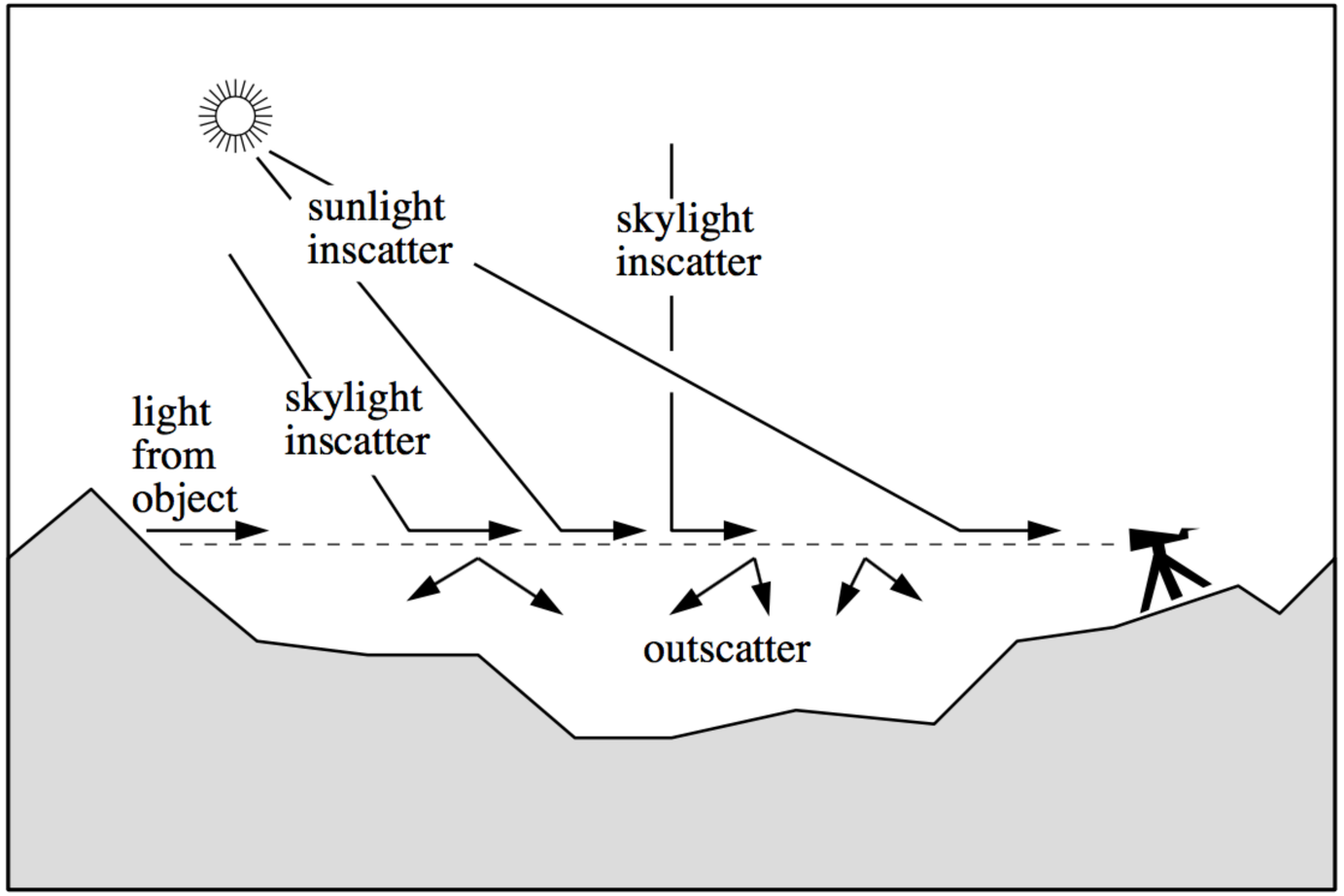
The first step is to find the equation of motion of an isotropic elastic medium whose density and rigidity may vary from point to point. If D denote the density and n the rigidity, a process similar to that used in Thomson and Tait's 'Natural Philosophy,' p. 530, leads us to the following:—

$$\frac{d}{dx}(m\delta) + \nabla_n \xi - \mathrm{D}\frac{d^2\xi}{dt^2} - \frac{dn}{dx}\frac{d\eta}{dy} - \frac{dn}{dx}\frac{d\zeta}{dz} + \frac{dn}{dy}\frac{d\eta}{dx} + \frac{dn}{dz}\frac{d\zeta}{dx} = 0, \quad . \quad . \quad (1)$$

* Communicated by the Author.



From “A Practical Analytic Model for Daylight”, Preetham 1999



From “A Practical Analytic Model for Daylight”, Preetham 1999

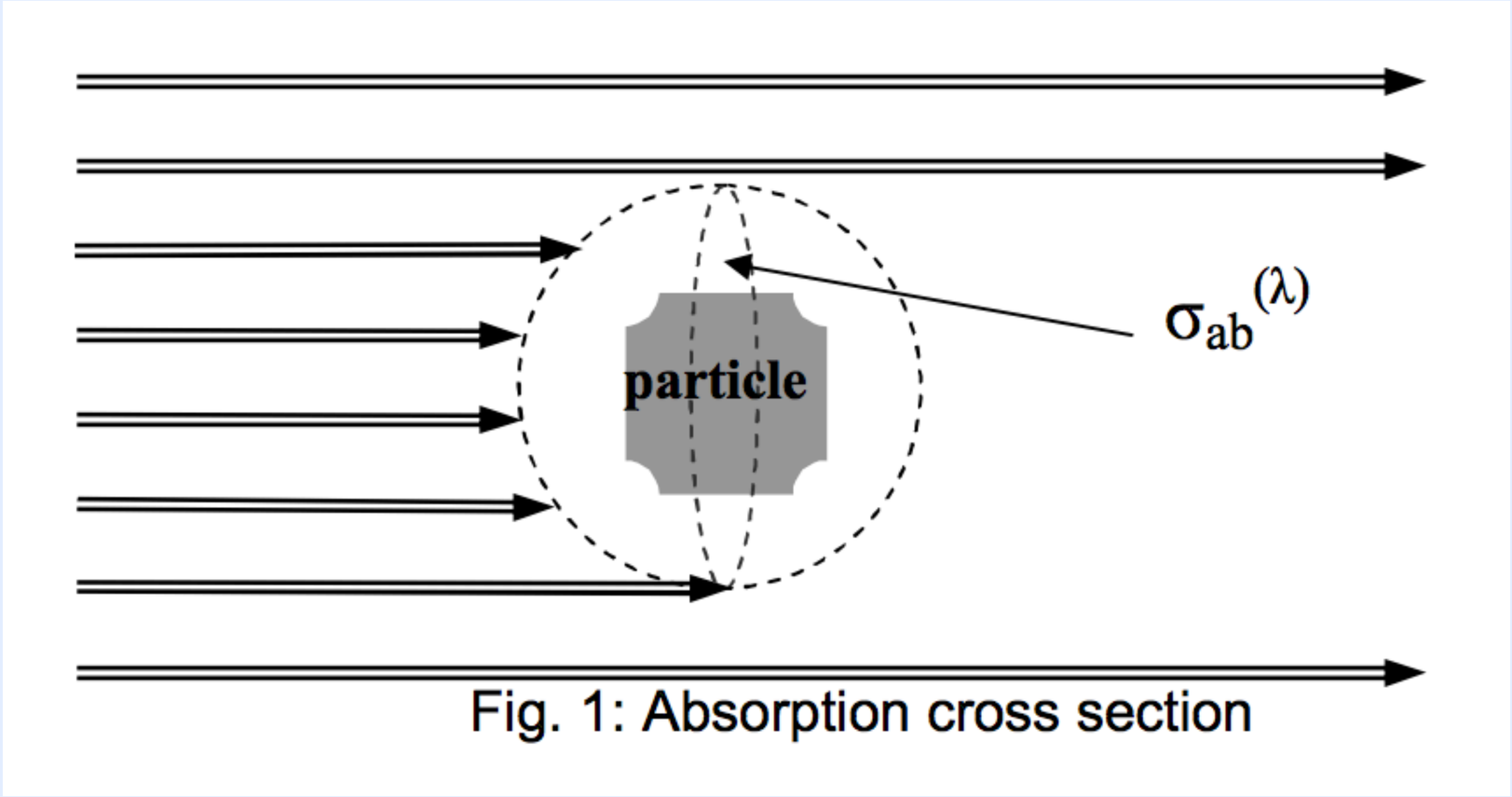


Fig. 1: Absorption cross section

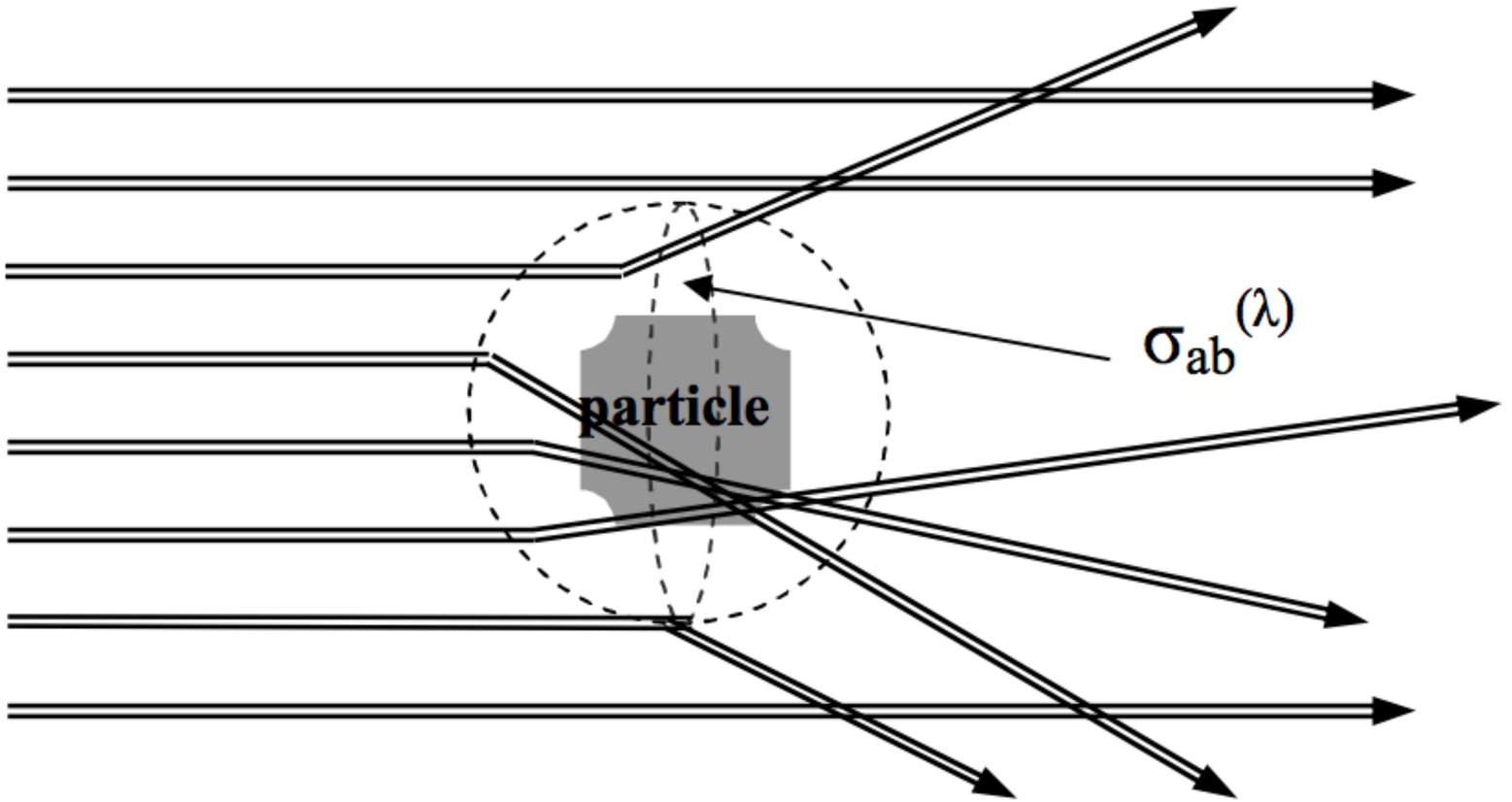


Fig. 3: Scattering cross section

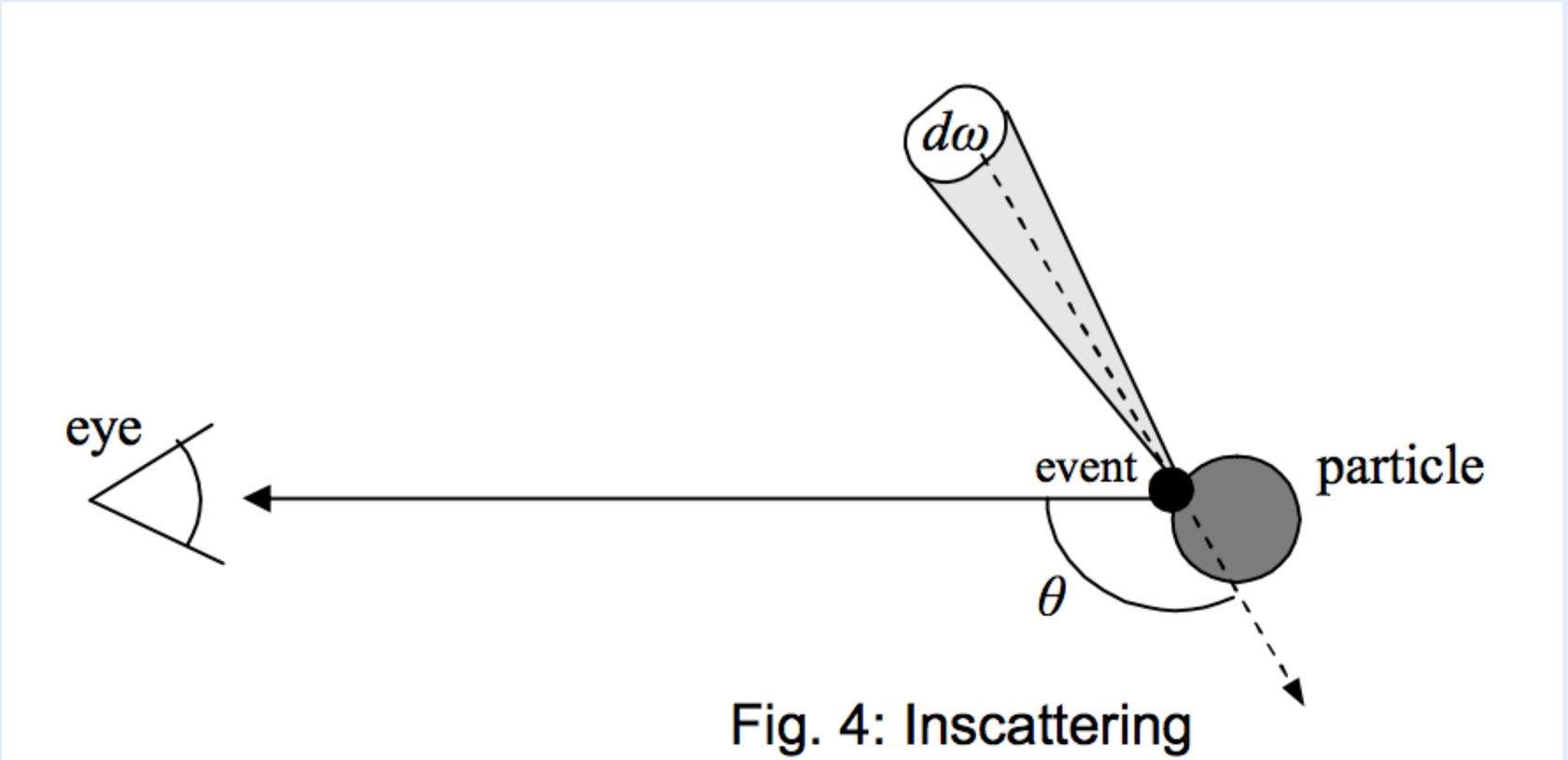
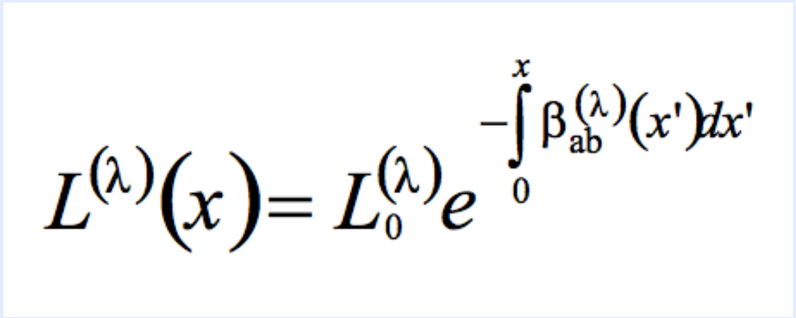
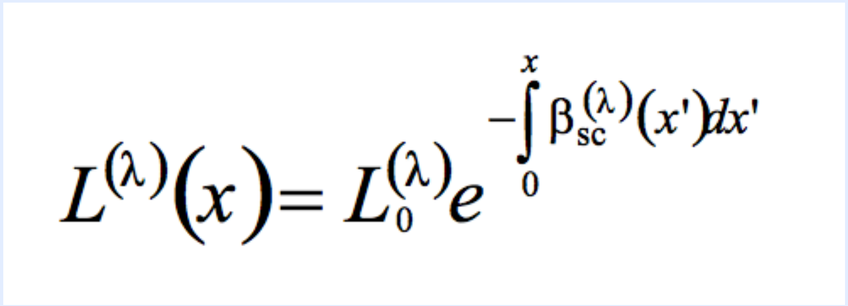


Fig. 4: Inscattering



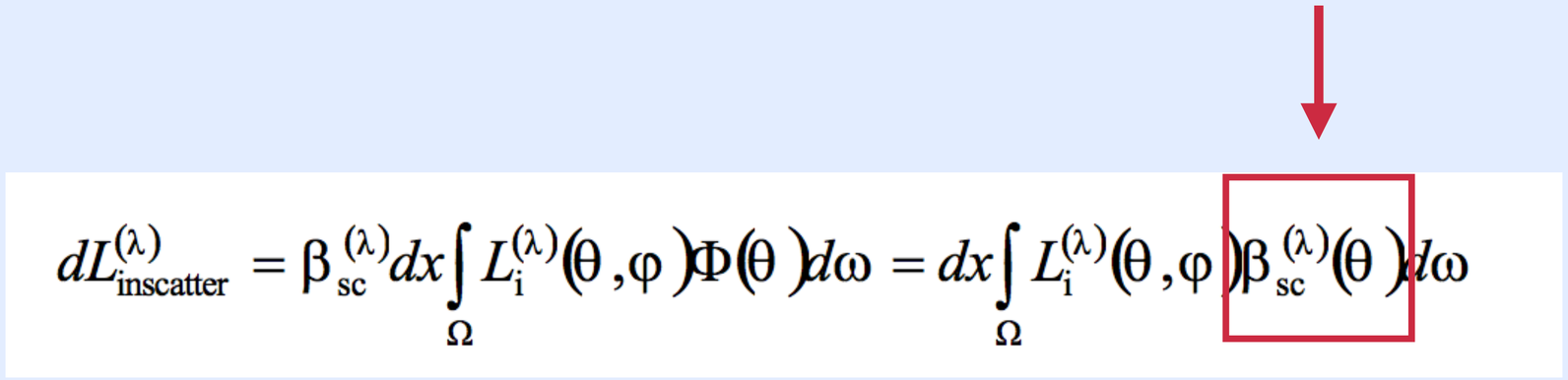
$$L^{(\lambda)}(x) = L_0^{(\lambda)} e^{-\int_0^x \beta_{ab}^{(\lambda)}(x')dx'}$$

Absorption

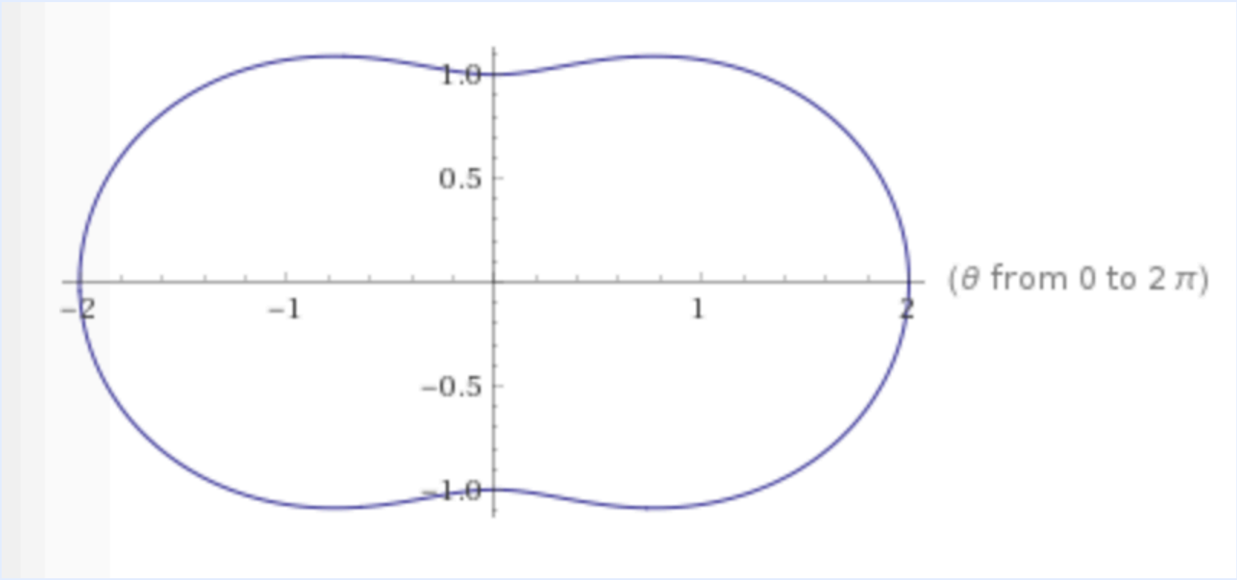


$$L^{(\lambda)}(x) = L_0^{(\lambda)} e^{-\int_0^x \beta_{sc}^{(\lambda)}(x')dx'}$$

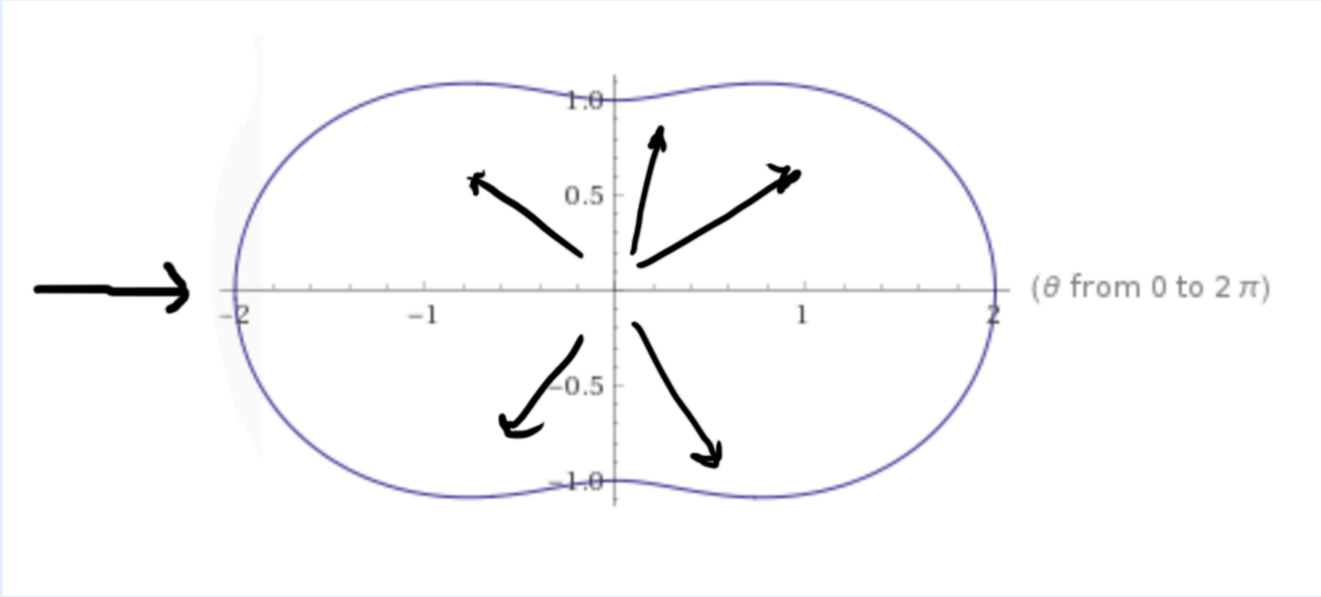
Out-scattering



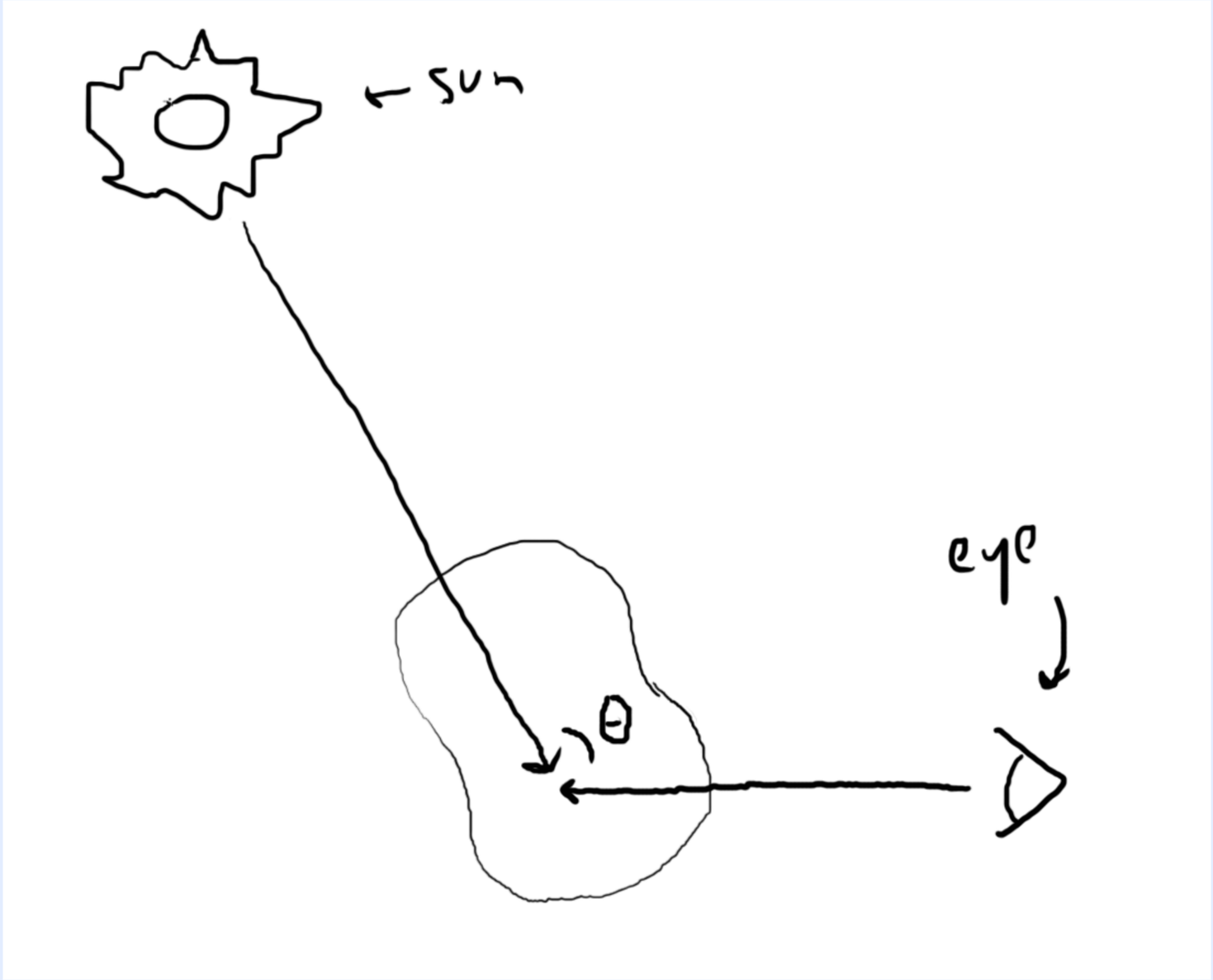
In-scattering

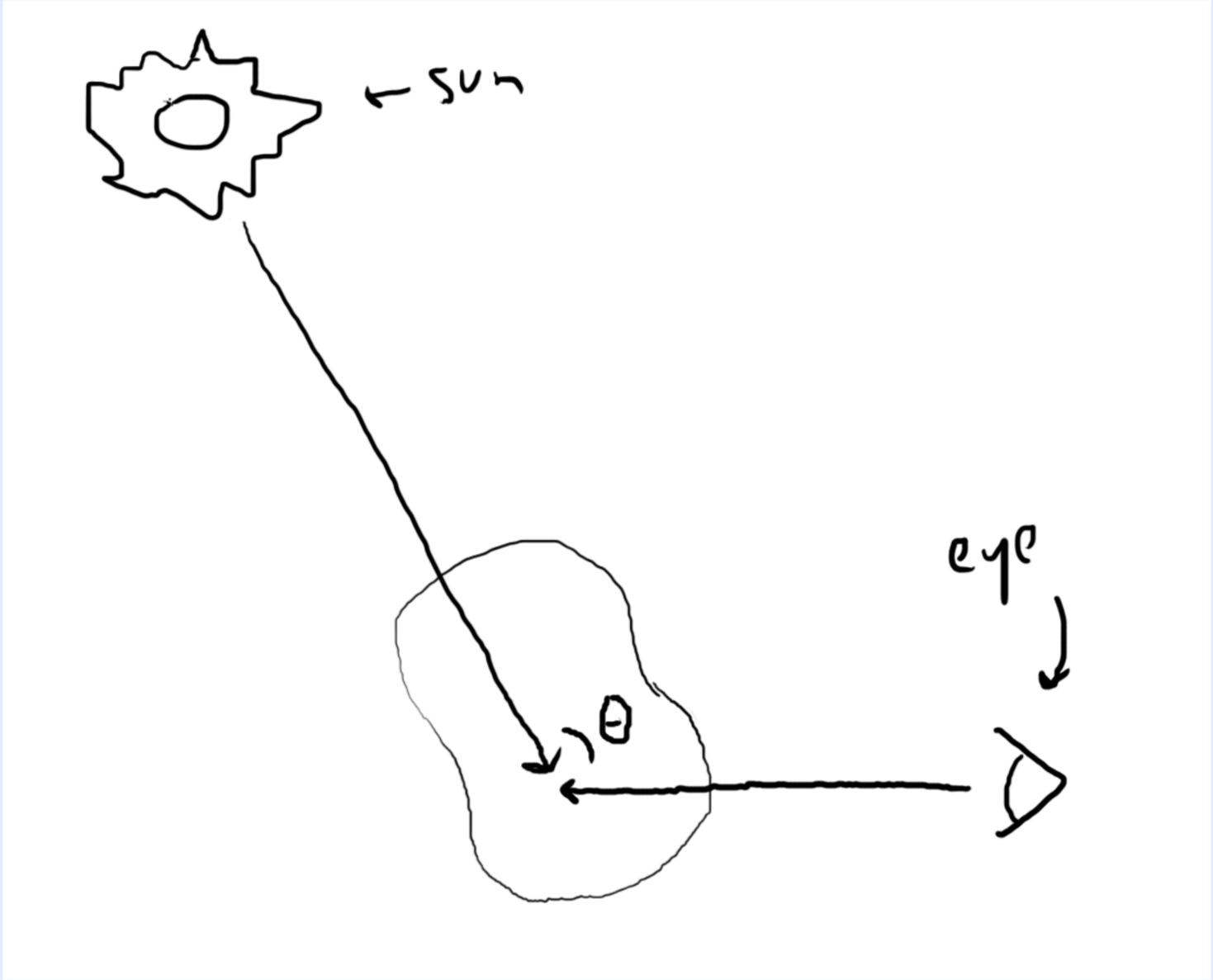


Phase function: $1 + cos^2\theta$



Phase function: $1 + cos^2\theta$


sun
θ
eye


sun
θ
eye

$$L(s,\theta) = L_0 F_{ex}(s) + L_{in}(s,\theta)$$

(19)

where L_0 is the input color, s is the distance to the camera, and θ is the angle to the sun.

In the vertex shader we compute $F_{ex}(s)$ and $L_{in}(s,\theta)$ using the following equations:

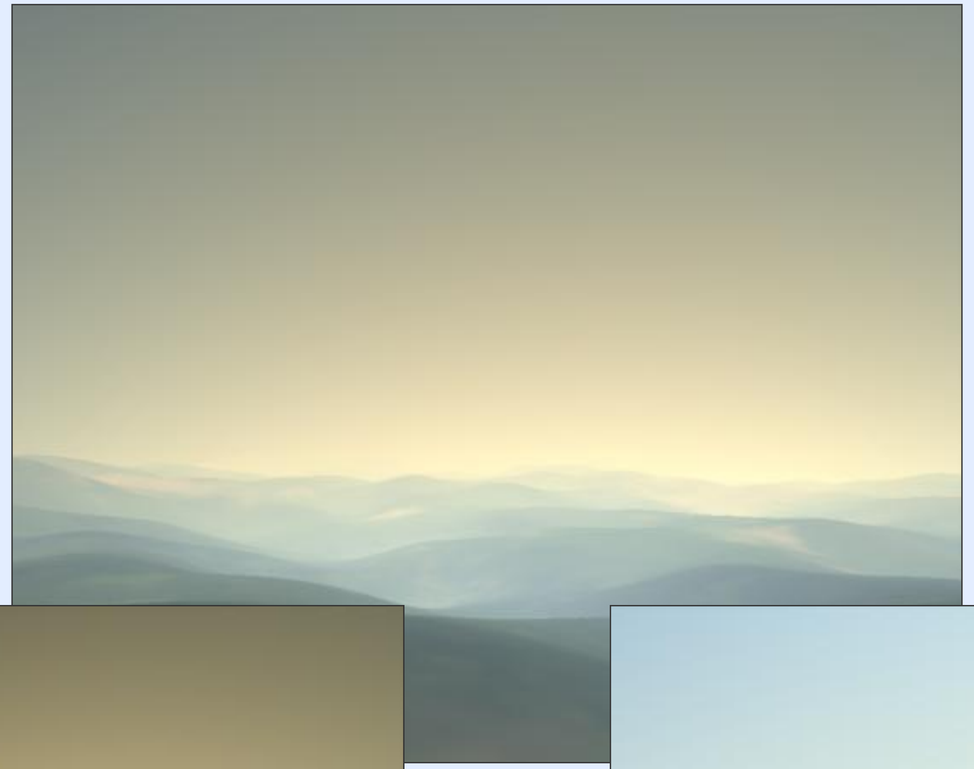
$$F_{ex}(s) = e^{-(\beta_R + \beta_M)s}$$

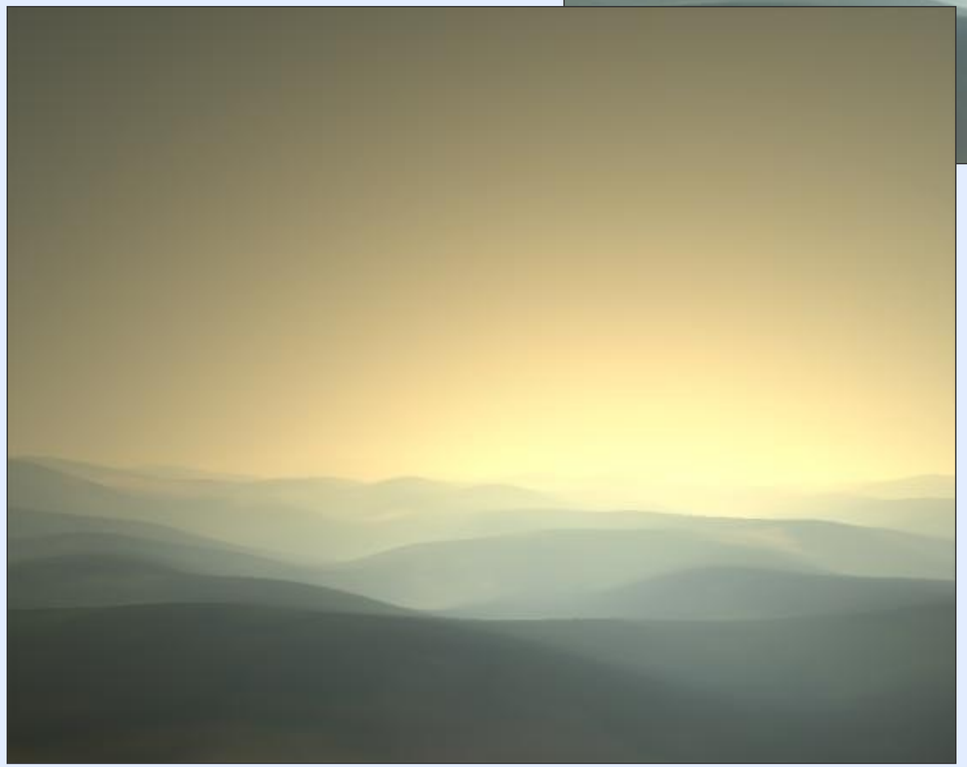
$$L_{in}(s,\theta) = \frac{\beta_R(\theta) + \beta_M(\theta)}{\beta_R + \beta_M} E_{sun}(1 - e^{-(\beta_R + \beta_M)s})$$

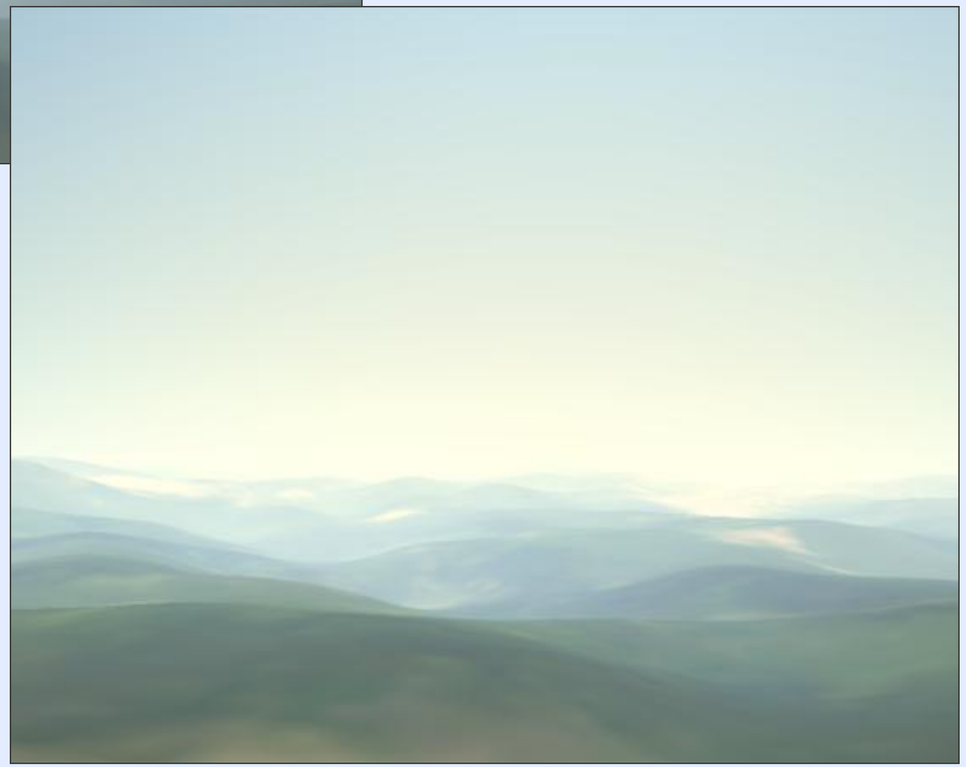
$$\beta_R(\theta) = \frac{3}{16\pi}\beta_R(1 + \cos^2\theta)$$

$$\beta_M(\theta) = \frac{1}{4\pi}\beta_M \frac{(1-g)^2}{(1 + g^2 - 2g\cos(\theta))^{3/2}}$$

where β_R is the Rayleigh coefficient, β_M is the Mie coefficient, g is Henyey/Greenstein phase function eccentricity, and E_{sun} is the irradiance of the sun. All that's left for the pixel shader is the multiplication and addition as described in Eq. 19.







What I learned

How it influenced me

An Incremental Approach to Compiler Construction

Abdulaziz Ghuloum
Department of Computer Science, Indiana University, Bloomington, IN 47408
aghuloum@cs.indiana.edu

Abstract

Compilers are perceived to be magical artifacts, carefully crafted by the wizards, and unfathomable by the mere mortals. Books on compilers are better described as wizard-talk: written by and for a clique of all-knowing practitioners. Real-life compilers are too complex to serve as an educational tool. And the gap between real-life compilers and the educational toy compilers is too wide. The novice compiler writer stands puzzled facing an impenetrable barrier, "better write an interpreter instead."

The goal of this paper is to break that barrier. We show that building a compiler can be as easy as building an interpreter. The compiler we construct accepts a large subset of the Scheme programming language and produces assembly code for the Intel-x86 architecture, the dominant architecture of personal computing. The development of the compiler is broken into many small incremental steps. Every step yields a fully working compiler for a progressively expanding subset of Scheme. Every compiler step produces real assembly code that can be assembled then executed directly by the hardware. We assume that the reader is familiar with the basic computer architecture: its components and execution model. Detailed knowledge of the Intel-x86 architecture is not required.

The development of the compiler is described in detail in an extended tutorial. Supporting material for the tutorial such as an

too simplistic and do not prepare the novice compiler writer to construct a useful compiler. The source language of these compilers often lacks depth and the target machine is often fictitious. Niklaus Wirth states that "to keep both the resulting compiler reasonably simple and the development clear of details that are of relevance only for a specific machine and its idiosyncrasies, we postulate an architecture according to our own choice"[20]. On the other extreme, advanced books focus mainly on optimization techniques, and thus target people who are already well-versed in the topic. There is no gradual progress into the field of compiler writing.

The usual approach to introducing compilers is by describing the structure and organization of a finalized and polished compiler. The sequencing of the material as presented in these books mirrors the passes of the compilers. Many of the issues that a compiler writer has to be aware of are solved beforehand and only the final solution is presented. The reader is not engaged in the process of developing the compiler.

In these books, the sequential presentation of compiler implementation leads to loss of focus on the big picture. Too much focus is placed on the individual passes of the compiler; thus the reader is not actively aware of the relevance of a single pass to the other passes and where it fits in the whole picture. Andrew Appel states that "a student who implements all the phases described in Part I of

An Incremental Approach to Compiler Construction

Abdulaziz Ghuloum
Department of Computer Science, Indiana University, Bloomington, IN 47408
aghuloum@cs.indiana.edu

Abstract

Compilers are perceived to be magical artifacts, carefully crafted by the wizards, and unfathomable by the mere mortals. Books on compilers are better described as wizard-talk: written by and for a clique of all-knowing practitioners. Real-life compilers are too complex to serve as an educational tool. And the gap between real-life compilers and the educational toy compilers is too wide. The novice compiler writer stands puzzled facing an impenetrable barrier, "better write an interpreter instead."

The goal of this paper is to break that barrier. We show that building a compiler can be as easy as building an interpreter. The compiler we construct accepts a large subset of the Scheme programming language and produces assembly code for the Intel-x86 architecture, the dominant architecture of personal computing. The development of the compiler is broken into many small incremental steps. Every step yields a fully working compiler for a progressively expanding subset of Scheme. Every compiler step produces real assembly code that can be assembled then executed directly by the hardware. We assume that the reader is familiar with the basic computer architecture: its components and execution model. Detailed knowledge of the Intel-x86 architecture is not required.

The development of the compiler is described in detail in an extended tutorial. Supporting material for the tutorial such as an

too simplistic and do not prepare the novice compiler writer to construct a useful compiler. The source language of these compilers often lacks depth and the target machine is often fictitious. Niklaus Wirth states that "to keep both the resulting compiler reasonably simple and the development clear of details that are of relevance only for a specific machine and its idiosyncrasies, we postulate an architecture according to our own choice"[20]. On the other extreme, advanced books focus mainly on optimization techniques, and thus target people who are already well-versed in the topic. There is no gradual progress into the field of compiler writing.

The usual approach to introducing compilers is by describing the structure and organization of a finalized and polished compiler. The sequencing of the material as presented in these books mirrors the passes of the compilers. Many of the issues that a compiler writer has to be aware of are solved beforehand and only the final solution is presented. The reader is not engaged in the process of developing the compiler.

In these books, the sequential presentation of compiler implementation leads to loss of focus on the big picture. Too much focus is placed on the individual passes of the compiler; thus the reader is not actively aware of the relevance of a single pass to the other passes and where it fits in the whole picture. Andrew Appel states that "a student who implements all the phases described in Part I of

Growing a Language, by Guy Steele

85,976 views

1K 14 SHARE

https://www.youtube.com/watch?v=_ahvzDzKdB0

1. **Write a compiler for a language that returns a fixed-sized integer.**

Let's write a small C function that returns an integer:

```
int scheme_entry(){
  return 42;
}
```

Let's compile it using `gcc -O3 --omit-frame-pointer -S test.c` and see the output. The most relevant lines of the output file are the following:

```
1.          .text
2.          .p2align 4,,15
3.  .globl scheme_entry
4.          .type   scheme_entry, @function
5.  scheme_entry:
6.          movl    $42, %eax
7.          ret
```

```
(define (compile-program x)
  (emit "movl $~a, %eax" x)
  (emit "ret"))
```

```
(read "(+ 1 2 (* foo 1000)")
;; '(+ 1 2 (* foo 1000))

(read "(define (foo x y) (+ x y))")
;; '(define (foo x y) (+ x y))
```

Immediate Constants

```
(define (compile-program x)
  (define (immediate-rep x)
    (cond
     ((integer? x) (shift x fixnum-shift))
     ...))
  (emit "movl $~a, %eax" (immediate-rep x))
  (emit "ret"))
```

Unary Primitives

```
(define (emit-expr x)
  (cond
   ((immediate? x)
    (emit "movl $~a, %eax" (immediate-rep x)))
   ((primcall? x)
    (case (primcall-op x)
      ((add1)
       (emit-expr (primcall-operand1 x))
       (emit "addl $~a, %eax" (immediate-rep 1)))
      ...))
   (else ...)))
```

integer->char
char->integer
null?
zero?
not

Binary Primitives

So far, single registry for return value: %eax

```
(+ (add1 10) (add1 11))
```

Need a stack!

Binary Primitives

- Contiguous array of memory locations
- Pointer to base of stack in `%esp` register
- Return point stored at the base: `0(%esp)`
- Free to use locations above it for intermediate values: `-4(%esp)` etc
- Compiler tracks current index into stack

Binary Primitives

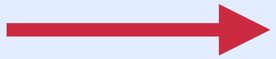
```
(define (emit-primitive-call x si)
  (case (primcall-op x)
    ((add1) ...)
    ((+)
     (emit-expr (primcall-operand2 x) si)
     (emit "movl %eax, ~a(%esp)" si)
     (emit-expr
      (primcall-operand1 x)
      (- si wordsize))
     (emit "addl ~a(%esp), %eax" si))
    ...))
```

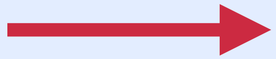
Local Variables

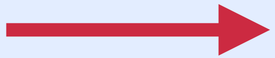
- Compiler passes around an environment in addition to stack index
- When new variable is introduced:
  - compile expression
  - push it onto the stack
  - add variable to environment with stack location

Local Variables

- When a variable is used:
  - get stack location from environment
  - read it from the stack







```
(define (emit-expr x si env)
  (cond
   ((immediate? x) ...)
   ((variable? x)
    (emit "movl ~a(%esp), %eax" (lookup x env)))
   ((let? x)
    (emit-let (bindings x) (body x) si env))
   ((primcall? x) ...)
   ...))

(define (emit-let bindings body si env)
  (let f ((b* bindings) (new-env env) (si si))
       (cond
        ((null? b*) (emit-expr body si new-env))
        (else
         (let ((b (car b*)))
           (emit-expr (rhs b) si env)
           (emit "movl %eax, ~a(%esp)" si)
           (f (cdr b*)
              (extend-env (lhs b) si new-env)
              (- si wordsize)))))))
```

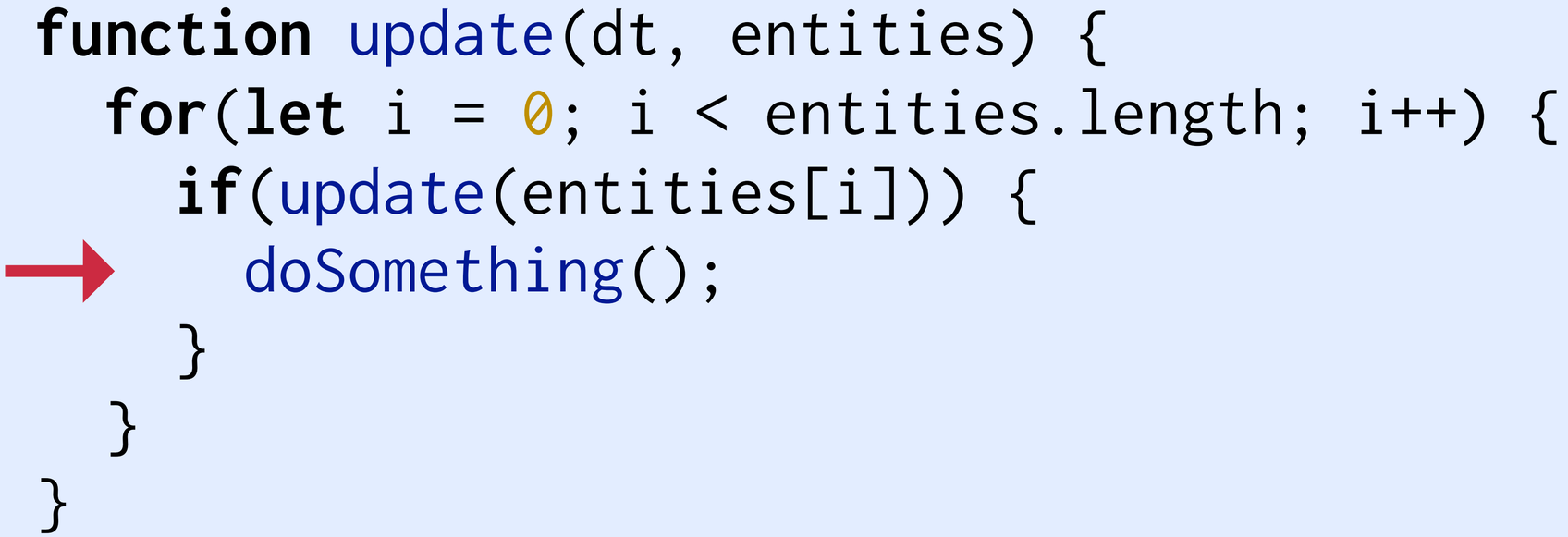
On and on...

- Heap allocation
- User-defined functions and closures
- Macros
- Foreign Functions
- I/O
- Reader

Self-hosting 👑

What I learned

How it influenced me


function update(dt, entities) {
for(let i = 0; i < entities.length; i++) {
if(update(entities[i])) {
doSomething();
}
}
}

jlongster / YPS

Watch 0 | Star 3 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Settings | Insights

Yield Passing Style http://jlongster.github.com/YPS/ Edit

Add topics

7 commits | 2 branches | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

jlongster update README — Latest commit 454ef64 on Nov 14, 2012

node_modules	CLI interface	5 years ago
Makefile	CLI interface	5 years ago
README	update README	5 years ago
index.html	attempt to optimize & benchmarks	5 years ago
machine.js	attempt to optimize & benchmarks	5 years ago
test-output.js	CLI interface	5 years ago

```
function bar(x) {
    return foo(x - 1) + 1;
}

function foo(x) {
    if(x > 0) {
        return 3 + bar(x);
    }
    return 0;
}

module.exports = { foo: foo };
```

```
function* bar(x) {
    yield ["return", (yield foo(x - (yield 1))) + (yield 1)]
}

function* foo(x) {
    if(x > (yield 0)) {
        yield ["return", (yield 3) + (yield bar(x))]
    }
    yield ["return", (yield 0)]
}

module.exports = { foo: foo };
```

Exceptional Continuations in JavaScript

Florian Loitsch

Inria Sophia Antipolis
2004 route des Lucioles - BP 93
F-06902 Sophia Antipolis, Cedex,
France
http://www.inria.fr/mimosa/Florian.Loitsch

ABSTRACT

JavaScript, the main language for web-site development, does not feature continuation. However, as part of client-server communication they would be useful as a means to suspend the currently running execution.
In this paper we present our adaption of exception-based continuations to JavaScript. The enhanced technique deals with closures and features improvements that reduce the cost of the work-around for the missing `goto`-instruction. We furthermore propose a practical way of dealing with exception-based continuations in the context of non-linear executions, which frequently happen due to callbacks.
Our benchmarks show that under certain conditions continuations are still expensive, but are often viable. Especially compilers translating to JavaScript could benefit from static control flow analyses to make continuations less costly.

1. Introduction

Xml-http-requests are an integral part of Ajax and the now called JavaScript does not feature any construct similar to continuations, though. Most interpreters carry the necessary information to efficiently implement them, but as far as we know only Rhino (Mozilla Foundation) gives access to its continuations. In general continuations need to be implemented on top of JavaScript's high level constructs.
Continuation Passing Style (CPS) lends itself for this task and with sufficiently high level features (in particular closures) CPS can be implemented as a simple source code transformation (see for example (Steele 1976)). A program in CPS form, as the name suggests, passes the current continuation directly as a parameter to every function and the continuation is hence always available. This technique does indeed work in JavaScript, and some systems such as Links (Cooper et al. 2006) actually use it. In Links the continuations are not exposed to the developers either, but are used internally for threading and transparent asynchronous `xml-http-requests`.
CPS's efficiency is however largely dependent on the speed of closure creation and tail call handling. Neither are fast in current main-

```
function sleep(ms) {
  suspend(function(resume) {
    setTimeout(resume, ms);
  });
}
```

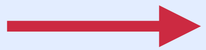
```
function suspend(handler) {
  const continuation = new ContinuationException();
  continuation.handler = handler;
  throw continuation;
}
```

```
function sequence(f, g) {
  print('1: ' + f());
  return g();
}
```

```
function sequence(f, g) {
  var tmp1;
  var index = 0;
  try {
    index = 1; tmp1 = f();
    print('1: ' + tmp1);
    index = 2; return g();
  } catch(e) {
    if (e instanceof ContinuationException) {
      var frame = new Object();
      frame.index = index; // save position
      frame.f = f; frame.g = g;
      frame.tmp1 = tmp1;
      e.pushFrame(frame);
    }
    throw e;
  }
}
```

```
function sequence(f, g) {
  var tmp1, goto = false;
  if (RESTORE.doRestore) {
    var frame = RESTORE.popFrame();
    index = frame.index;
    f = frame.f; g = frame.g;
    tmp1 = frame.tmp1;
    goto = index; // emulate a jump
  }
  // ... <suspension-code omitted> ...
  switch (goto) {
    case false:
    case 1: goto = false;
      tmp1 = f();
      print('1: ' + tmp1);
      // fall-through
    case 2: goto = false;
      return g();
  }
  // ... <suspension-code omitted> ...
}
```

```
function sequence(f, g) {
  print('1: ' + f());
  return g();
}
```

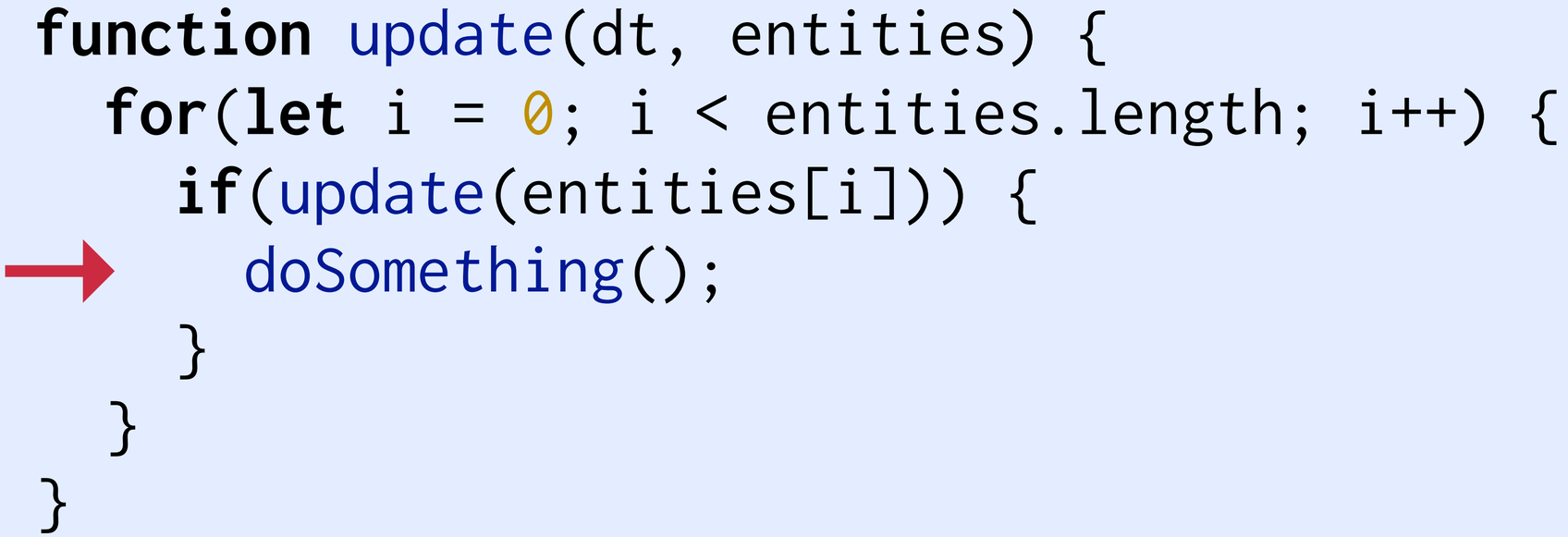


```
function sequence(f, g) {
  var tmp1;
  var index = 0;
  var goto = false;
  if (RESTORE.doRestore) {
    var frame = RESTORE.popFrame();
    index = frame.index;
    f = frame.f; g = frame.g;
    tmp1 = frame.tmp1;
    goto = index;
  }
  try {
    switch (goto) {
      case false:
      case 1: goto = false;
        index = 1; tmp1 = f();
        print('1: ' + tmp1);
      case 2: goto = false;
        index = 2; return g();
    }
  } catch(e) {
    if (e instanceof ContinuationException) {
      var frame = new Object();
      frame.index = index; // save position
      frame.f = f; frame.g = g;
      frame.tmp1 = tmp1;
      e.pushFrame(frame);
    }
    throw e;
  }
}
```

```
function f() {
  var x = 1;
  var y = 2;
  var g = function() { print(x, y); };
  suspendCall();
  x = 3;
  g(); // should print 3, 2
}
```

call/cc

But wait...


function update(dt, entities) {
for(let i = 0; i < entities.length; i++) {
if(update(entities[i])) {
doSomething();
}
}
}

facebook / **regenerator**

Watch 85 | Unstar 2,084 | Fork 214

Code | Issues 35 | Pull requests 12 | Projects 0 | Wiki | Insights

Source transformer enabling ECMAScript 6 generator functions in JavaScript-of-today.
http://facebook.github.io/regenerator/

808 commits | **13** branches | **132** releases | **49** contributors | BSD-2-Clause

Branch: **master** | New pull request | Create new file | Upload files | Find file | Clone or download

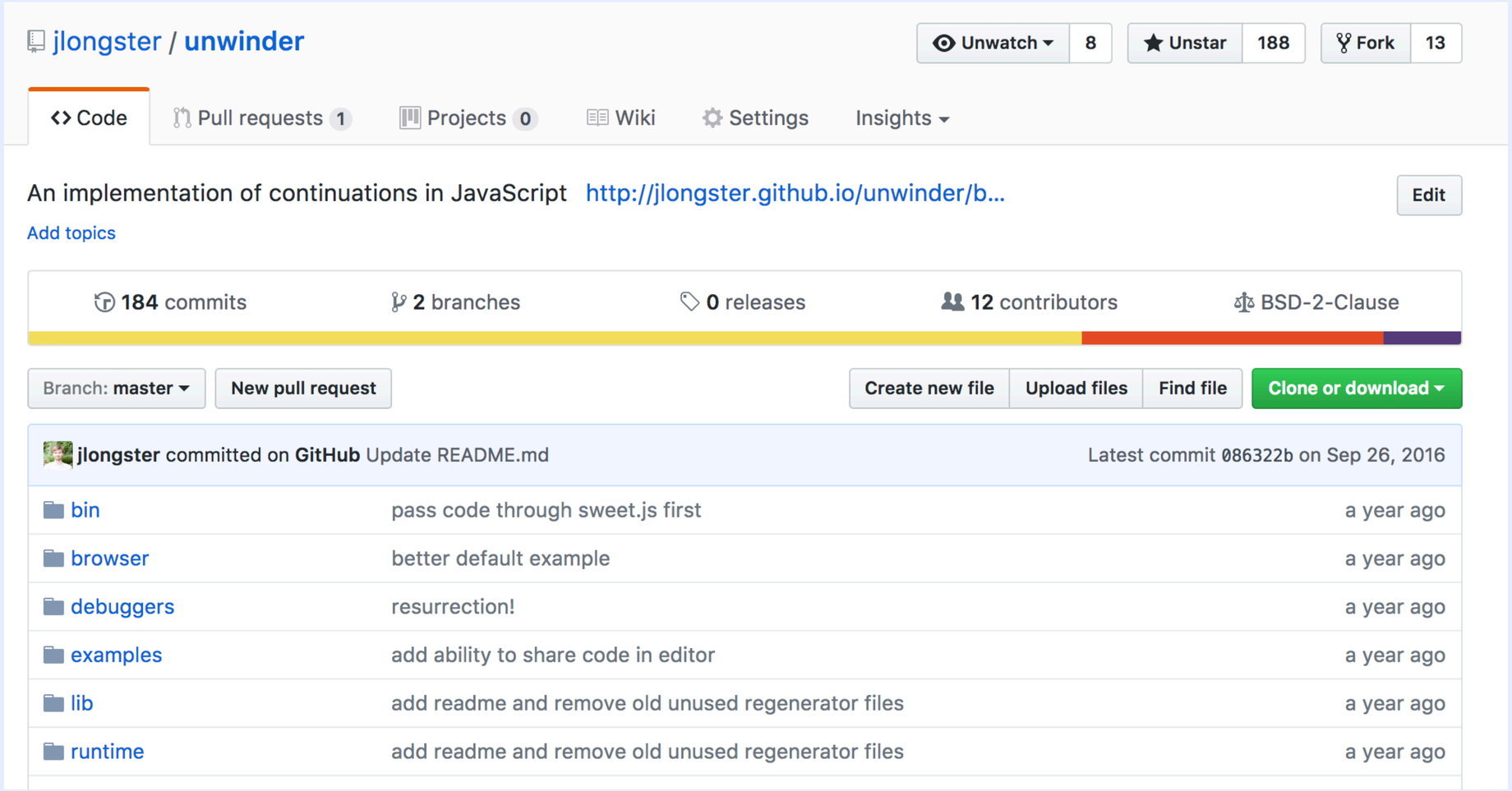
benjamn Publish ... Latest commit `3860530` 8 days ago

bin	Add basic tests of bin/regenerator options.	3 years ago
docs	Regenerate docs/bundle.js with latest implementation.	a month ago
lib	More cleanup of dependencies and main.js.	10 months ago
packages	Publish	8 days ago
test	Move uglify-js to devDependencies and other minor tweaks.	a month ago
.gitignore	Move sloppy tests into their own file	10 months ago

```
function run() {
  while(!stop) {
    update(entities);
  }
}
```

```
while (1) {
  if (VM.hasBreakpoint(funcId, $__next)) {
    throw new $ContinuationExc();
  }

  switch ($__next) {
    case 0:
      if (!!stop) {
        $__next = 9;
        break;
      }

      $__next = 3;
      break;
    case 3:
      $__next = 7;
      $__tmpid = 1;
      $__t1 = update(entities);
      break;
    case 7:
      $__next = 0;
      break;
    default:
    case 9:
      return;
  }
}
```

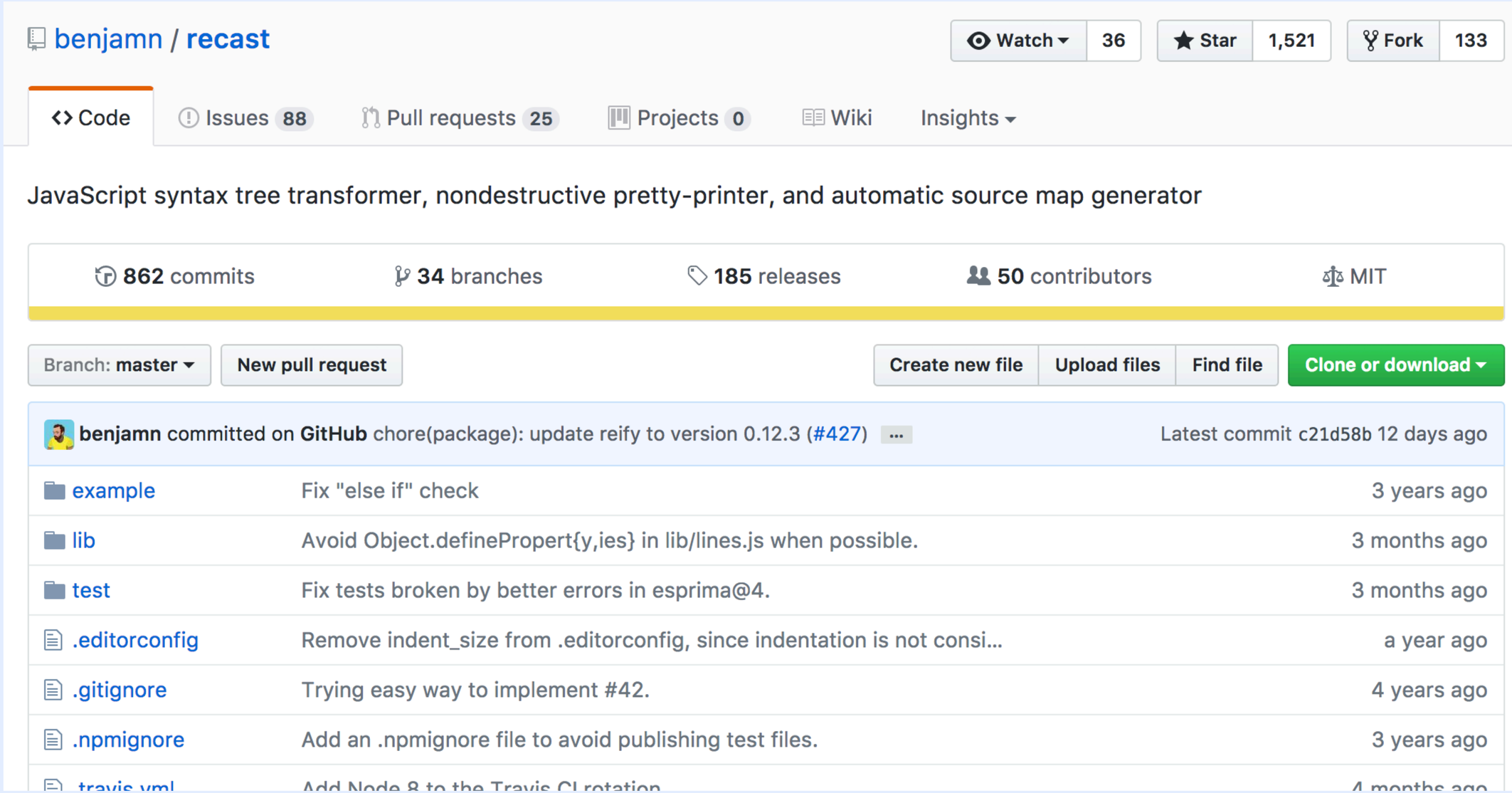
```
function foo() {
  var x = callCC(cont => cont);
  if(isContinuation(x)) {
    x(5);
  }
  return x;
}

console.log(foo()); // -> 5
```



https://github.com/jlongster/unwinder

What I learned

How it influenced me


benjamn / recast
Watch 36
Star 1,521
Fork 133
Code
Issues 88
Pull requests 25
Projects 0
Wiki
Insights
JavaScript syntax tree transformer, nondestructive pretty-printer, and automatic source map generator
862 commits
34 branches
185 releases
50 contributors
MIT
Branch: master
New pull request
Create new file
Upload files
Find file
Clone or download
benjamn committed on GitHub chore(package): update reify to version 0.12.3 (#427)
Latest commit c21d58b 12 days ago
example
Fix "else if" check
3 years ago
lib
Avoid Object.definePropert{y,ies} in lib/lines.js when possible.
3 months ago
test
Fix tests broken by better errors in esprima@4.
3 months ago
.editorconfig
Remove indent_size from .editorconfig, since indentation is not consi...
a year ago
.gitignore
Trying easy way to implement #42.
4 years ago
.npmignore
Add an .npmignore file to avoid publishing test files.
3 years ago

A prettier printer

Philip Wadler

Joyce Kilmer and most computer scientists agree: there is no poem as lovely as a tree. In our love affair with the tree it is parsed, pattern matched, pruned — and printed. A pretty printer is a tool, often a library of routines, that aids in converting a tree into text. The text should occupy a minimal number of lines while retaining indentation that reflects the underlying tree. A good pretty printer must strike a balance between ease of use, flexibility of format, and optimality of output.

Over the years, Richard Bird and others have developed the algebra of programming to a fine art. John Hughes (1995) describes the evolution of a pretty printer library, where both the design and implementation have been honed by an appealing application of algebra. This library has become a standard package, widely used in the field. A variant of it was implemented for use in the Glasgow


Prettier

```
(<>) :: Doc -> Doc -> Doc
nil :: Doc
text :: String -> Doc
line :: Doc
nest :: Int -> Doc -> Doc
layout :: Doc -> String
```

```
showTree (Node s ts) = text s <> showBracket ts

showBracket [] = nil
showBracket ts = text "[" <>
                 nest 2 (line <> showTrees ts) <>
                 line <> text "]"

showTrees [t] = showTree t
showTrees (t:ts) = showTree t <> text "," <> line <> showTrees ts
```

```
aaa[
  bbbbb[
    ccc,
    dd
  ],
  eee,
  ffff[
    gg,
    hhh,
    ii
  ]
]
```

```
group :: Doc -> Doc
```

```
group (text "foo" <> line <> text "bar")
```

```
foo bar

foo
bar
```

```
group (text "foo" <> line <> group (text "bar" <> line <> text "baz"))
```

```
foo bar baz

foo bar
baz

foo
bar
baz
```

```
pretty :: Int -> Doc -> String
```

```
showTree (Node s ts) = group (text s <> nest (length s) (showBracket ts))
```

```
aaa[bbbbb[ccc, dd],
    eee,
    ffff[gg, hhh, ii]]
```

group
concat
indent
conditionalGroup
fill
ifBreak
line
softline
hardline
literalline

```
group(
  concat([
    "[",
    indent(
      concat([
        softline,
        printArrayItems(path, options, "elements", print)
      ])
    ),
    needsForcedTrailingComma ? "," : "",
    ifBreak(
      shouldPrintComma(options)
        ? ","
        : ""
    ),
    ...
  ])
)
```

What I learned

How it influenced me

Thanks!

@jlongster

References

1. Rendering Outdoor Light Scattering in Real Time: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/ATI-LightScattering.pdf
2. A Practical Analytic Model for Daylight: https://www.cs.utah.edu/~shirley/papers/sunsky/sunsky.pdf
3. Growing a Language: https://www.youtube.com/watch?v=_ahvzDzKdB0
4. An Incremental Approach to Compiler Construction: http://scheme2006.cs.uchicago.edu/11-ghuloum.pdf
5. YPS: https://github.com/jlongster/yps
6. regenerator: https://github.com/facebook/regenerator
7. Exceptional Continuations in JavaScript: http://www.schemeworkshop.org/2007/procPaper4.pdf
8. Unwinder: https://github.com/jlongster/unwinder
9. A Prettier Printer: https://homepages.inf.ed.ac.uk/wadler/papers/prettier/prettier.pdf
10. recast: https://github.com/benjamn/recast