GO PROFILING

ERIK LUPANDER

GO WEST 2020-02-19 | CALLISTAENTERPRISE.SE

CALLISTA

— ENTERPRISE —

AGENDA

- Introduction
- Go profiling with pprof
- Case study
- Summary

INTRODUCTION

ABOUT THE SPEAKER

- Erik Lupander
- Architect & Developer at Callista
- 15+ years of Java EE & Spring
- Started coding Go in 2015
  - Full time Go projects for the last year
    - » And it's my language of choice!

WHAT'S PROFILING ANYWAY?

- Dynamic program analysis
- Runtime analysis
  - Memory use / allocations / gc
  - Freq / duration of calls
  - On a very fine-granular level
- Used for optimization and troubleshooting
  - Waiting for IO ;)

PROFILING THROUGH CODE INSTRUMENTATION

- Compiles or runtime-injects measurement code into your application
- Allows fine-grained study of code-paths, allocations etc.
  - May have performance impact or require agents on servers etc.
- Exists for many languages

GO TOOL PPROF

PPROF

- Tool authored by Google for visualization and analysis of profiling data
- Based around profiling samples stored in a protobuf format
  - A sample ”describes a program call stack and a number or weight of samples collected at a location”

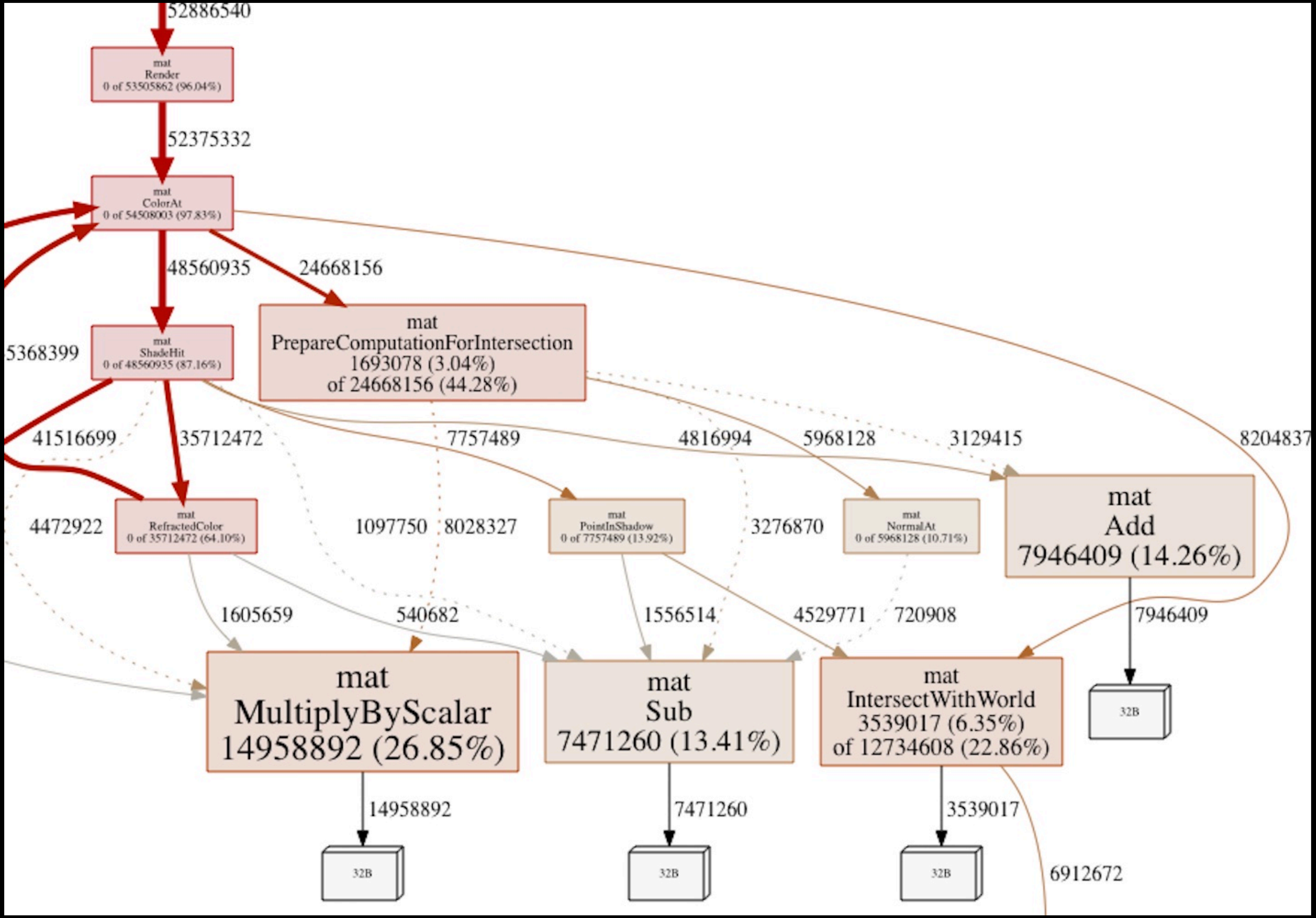
PPROF - VISUALIZATIONS

- What:
  - Interactive console UI

```
~/pprof> go tool pprof pprof.samples.cpu.021.pb.gz
Type: cpu
Time: Jan 30, 2020 at 2:55pm (CET)
Duration: 30s, Total samples = 6.61s (22.03%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 6310ms, 95.46% of 6610ms total
Dropped 26 nodes (cum <= 33.05ms)
Showing top 10 nodes out of 35
      flat  flat%   sum%        cum   cum%
    2830ms 42.81% 42.81%     2830ms 42.81%  crypto/sha256.block
    1660ms 25.11% 67.93%     1660ms 25.11%  crypto/md5.block
     570ms  8.62% 76.55%      570ms  8.62%  runtime.memmove
     500ms  7.56% 84.11%      500ms  7.56%  runtime.nanotime
     240ms  3.63% 87.75%     3160ms 47.81%  crypto/sha256.(*digest).Write
     140ms  2.12% 89.86%      140ms  2.12%  runtime.usleep
     110ms  1.66% 91.53%      110ms  1.66%  runtime.memclrNoHeapPointers
     100ms  1.51% 93.04%     1440ms 21.79%  crypto/sha256.(*digest).checkSum
      80ms  1.21% 94.25%     1740ms 26.32%  crypto/md5.(*digest).Write
      80ms  1.21% 95.46%     3370ms 50.98%  crypto/sha256.Sum256
(pprof) 
```

PPROF - VISUALIZATIONS

- What:
  - Interactive console UI
  - Viz-based visualizations



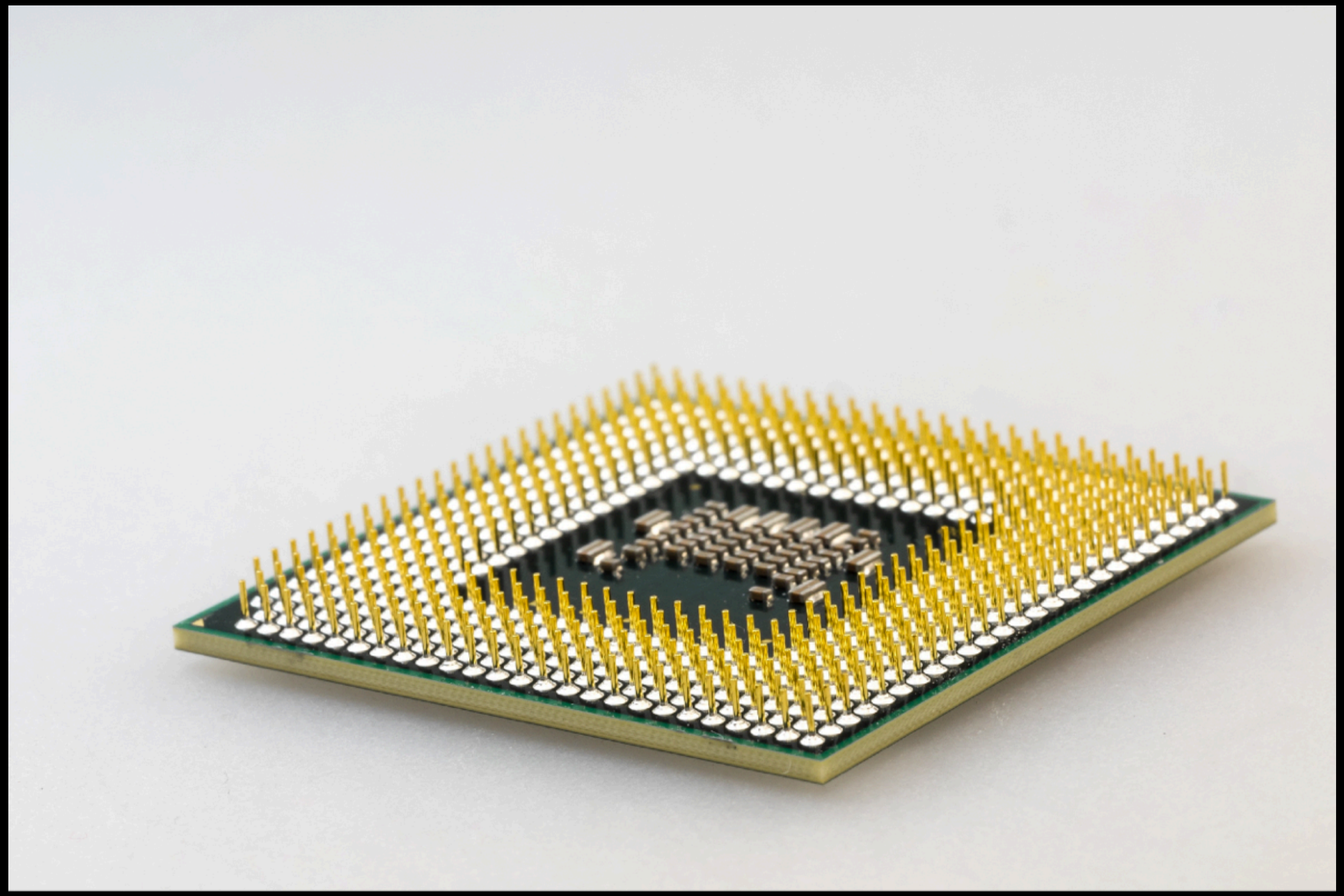
PPROF - VISUALIZATIONS

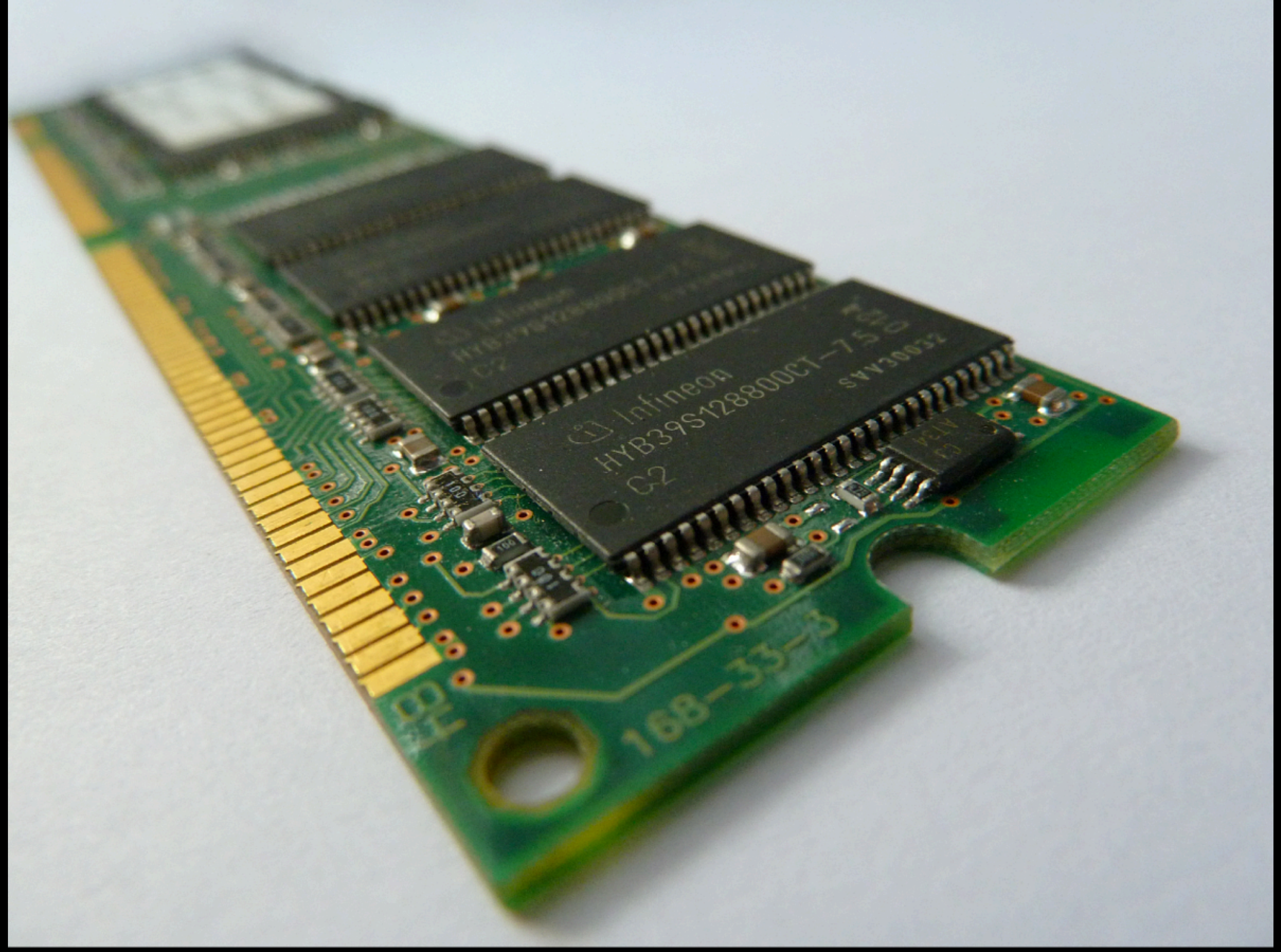
- What:
  - Interactive console UI
  - Viz-based visualizations
  - Listings (text / web)

```
(pprof) list CPULoader
Total: 6.54s
ROUTINE ======================== github.com/eriklupander/profiling/cpu.CPULoade
oad.go
     810ms         3s (flat, cum) 45.87% of Total
         .          .     23:func CPULoader(times int) []int {
         .          .     24:   // Keeps it to a 32 bit int
         .          .     25:   //num := 40
         .          .     26:   var r []int
         .          .     27:   result := false
      50ms       50ms     28:   for n := 0; n < times*200; n++ {
         .          .     29:           if n%1 != 0 {
         .          .     30:                   result = false
         .          .     31:           } else if n <= 1 {
         .          .     32:                   result = false
         .          .     33:           } else if n <= 3 {
         .          .     34:                   result = true
         .          .     35:           } else if n%2 == 0 {
         .          .     36:                   result = false
         .          .     37:           }
     290ms      290ms     38:           dl := int(math.Sqrt(float64(n)))
     330ms      330ms     39:           for d := 3; d <= dl; d += 2 {
     140ms      140ms     40:                   if n%d == 0 {
         .          .     41:                           result = false
         .          .     42:                   }
         .          .     43:           }
         .          .     44:           result = true
         .          .     45:   }
         .      2.19s     46:   sum := SumRoots(result)
         .          .     47:   if int(sum) % 1000  == 0 {
         .          .     48:           fmt.Print(".")
         .          .     49:   }
         .          .     50:   return r
         .          .     51:}
```

PROFILING TYPES

- /debug/pprof/profile
  - CPU, time spent in file/func/line
    - » But not invocations counts!
- /debug/pprof/heap
  - in-use objects and space
  - allocs objects and space
- /debug/pprof/block
  - Blocked goroutines
- /debug/pprof/mutex
  - Holders of contended mutexes
- /debug/pprof/trace





WIKIPEDIA

HOW TO ADD PROFILING TO YOUR GO CODE?

- Very easy:
  - The HTTP way or the Programmatic way

HOW TO ADD PROFILING TO YOUR GO CODE?

```go
package main

import (
    "log"
    "net/http"
    _ "net/http/pprof"
)


func main() {

    go func() {
        log.Println(http.ListenAndServe("localhost:6060", nil))
    }()

    // rest of your program
}
```

CAPTURE A PROFILE - GO TOOL

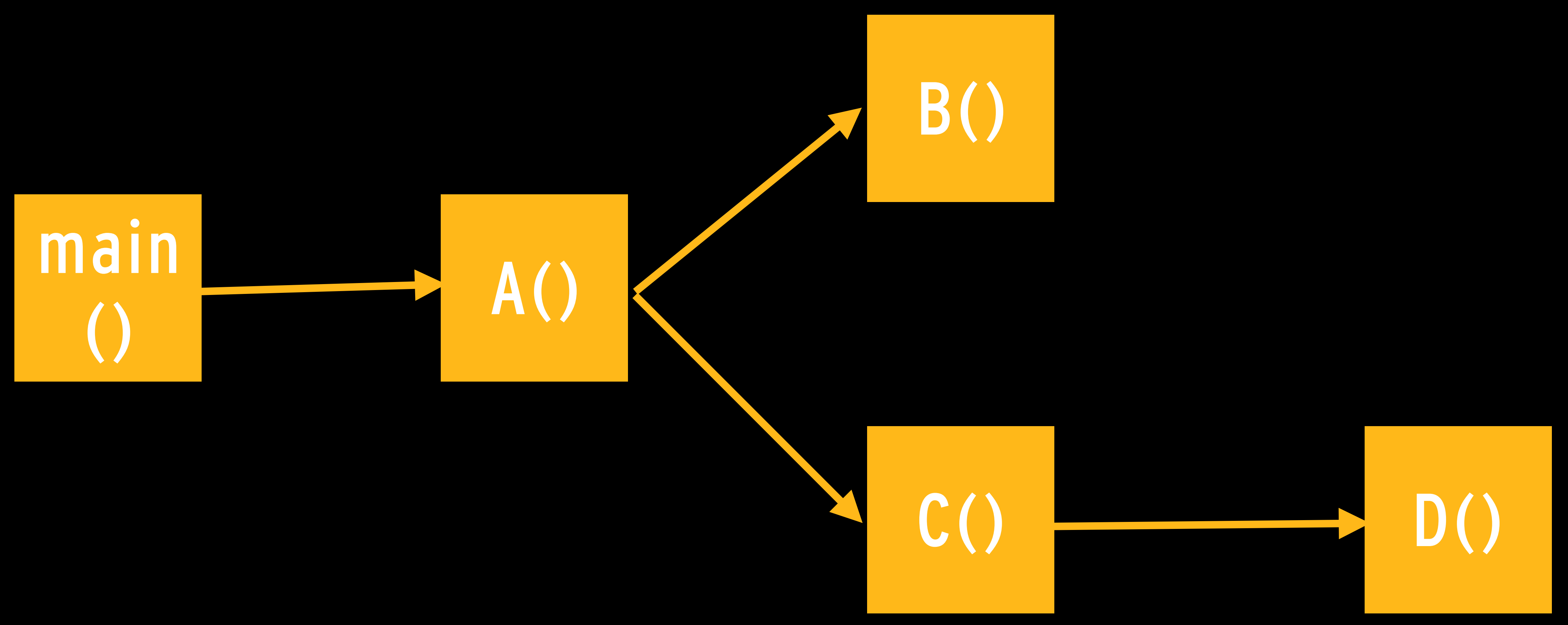
- While your program is executing, run:
  - go tool pprof http://localhost:6060/debug/pprof/profile
    - » Opens the pprof console with the produced profile loaded
  - go tool pprof -png http://localhost:6060/debug/pprof/profile > profile.png
    - » Produces a viz graph in PNG format
  - Both saves a .pprof file to disk into ~/pprof (on my mac)
- One can also do a HTTP GET from curl / web browser and the result will be downloaded

THE PPROF CONSOLE

```
~/pprof> go tool pprof pprof.samples.cpu.021.pb.gz
Type: cpu
Time: Jan 30, 2020 at 4:58pm (CET)
Duration: 30s, Total samples = 5.61s (18.70%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) help
  Commands:
    callgrind        Outputs a graph in callgrind format
    comments         Output all profile comments
    disasm           Output assembly listings annotated with samples
    dot              Outputs a graph in DOT format
    eog              Visualize graph through eog
    evince           Visualize graph through evince
    gif              Outputs a graph image in GIF format
    gv               Visualize graph through gv
    kcachegrind      Visualize report in KCachegrind
    list             Output annotated source for functions matching regexp
    pdf              Outputs a graph in PDF format
    peek             Output callers/callees of functions matching regexp
    png              Outputs a graph image in PNG format
    proto            Outputs the profile in compressed protobuf format
    ps               Outputs a graph in PS format
    raw              Outputs a text representation of the raw profile
    svg              Outputs a graph in SVG format
    tags             Outputs all tags in the profile
    text             Outputs top entries in text form
    top              Outputs top entries in text form
    topproto         Outputs top entries in compressed protobuf format
    traces           Outputs all profile samples in text form
    tree             Outputs a text rendering of call graph
    web              Visualize graph through web browser
    weblist          Display annotated source in a web browser
    o/options        List options and their current values
    quit/exit/^D     Exit pprof
```

THE DEMO PROGRAM

- Toy program that calculates prime numbers to simulate load in interdependent functions

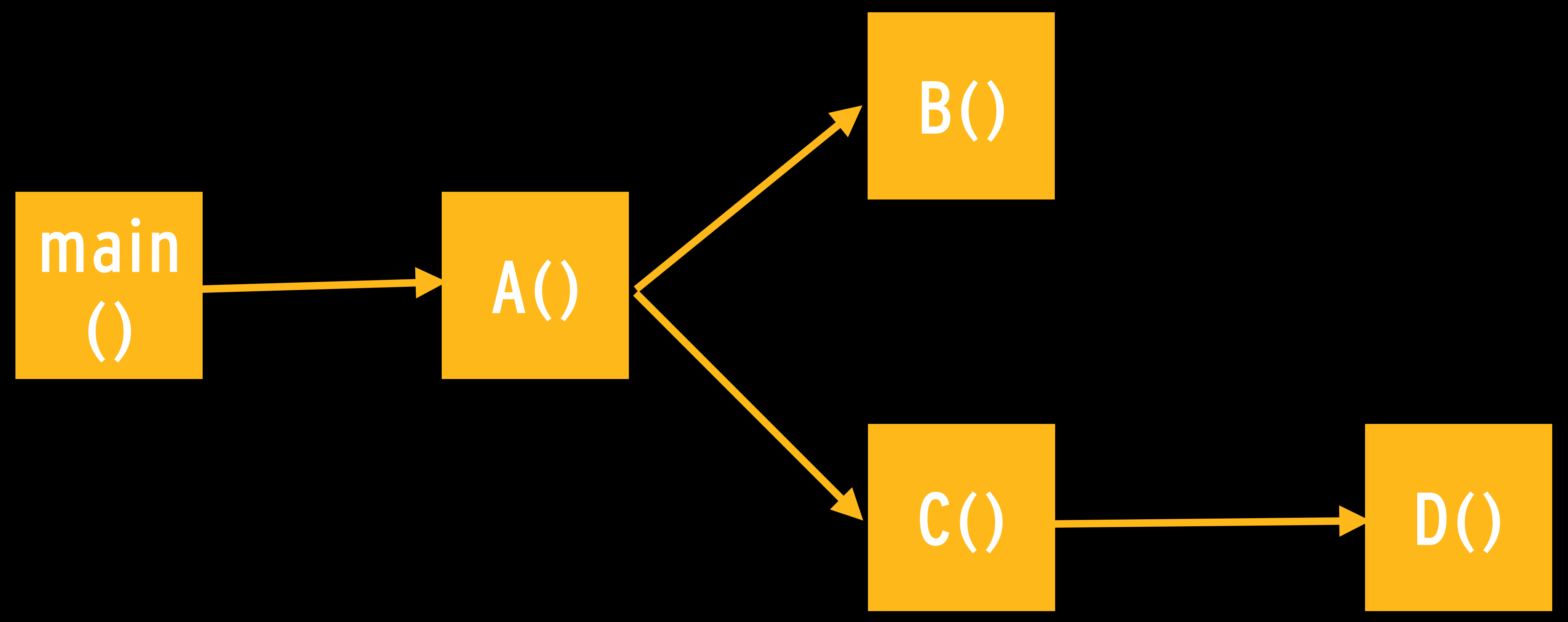


PPROF CONSOLE - TOP

```
(pprof) top
Showing nodes accounting for 6530ms, 99.85% of 6540ms total
Dropped 4 nodes (cum <= 32.70ms)
      flat  flat%   sum%        cum   cum%
    3150ms 48.17% 48.17%     3150ms 48.17%  github.com/eriklupander/profiling/cpu.prime
    2190ms 33.49% 81.65%     2190ms 33.49%  github.com/eriklupander/profiling/cpu.SumRoots
     810ms 12.39% 94.04%     3000ms 45.87%  github.com/eriklupander/profiling/cpu.CPULoader
     310ms  4.74% 98.78%      310ms  4.74%  runtime.nanotime
      70ms  1.07% 99.85%     6220ms 95.11%  github.com/eriklupander/profiling/cpu.CPU
         0     0% 99.85%     6230ms 95.26%  main.main
         0     0% 99.85%     6230ms 95.26%  runtime.main
         0     0% 99.85%      310ms  4.74%  runtime.mstart
         0     0% 99.85%      310ms  4.74%  runtime.mstart1
         0     0% 99.85%      310ms  4.74%  runtime.sysmon
```

PROFILER BASICS

- Measurements:
    - flat => time spent in own function
    - flat % => percentage of program time spent in own function
    - cum => cumulative time spent in self + all child functions
    - cum % => cumulative % spent in self + all child functions
    - sum % => sum of flat

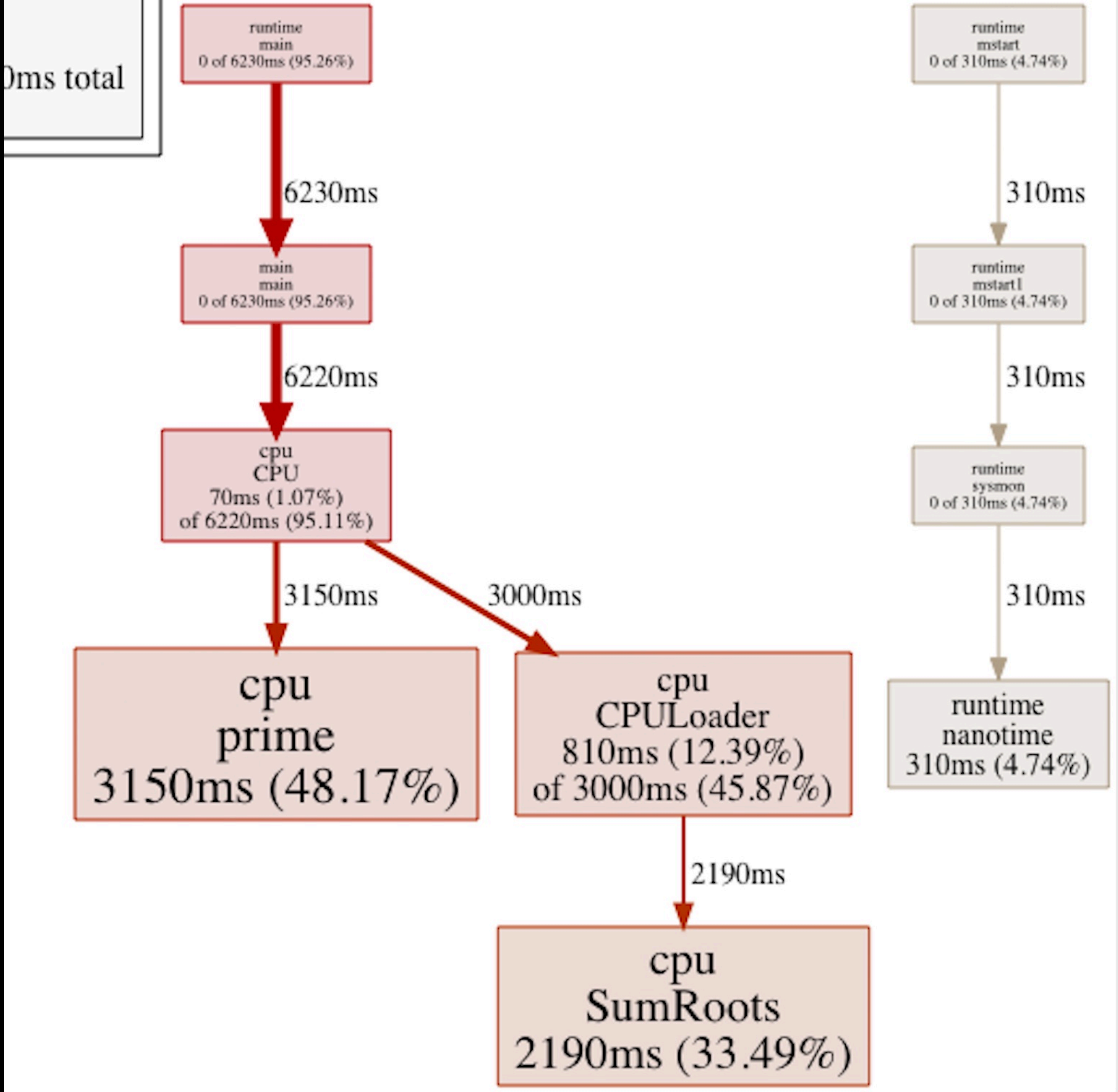


PPROF CONSOLE - TOP

```
(pprof) top
Showing nodes accounting for 6530ms, 99.85% of 6540ms total
Dropped 4 nodes (cum <= 32.70ms)
      flat  flat%   sum%        cum   cum%
    3150ms 48.17% 48.17%     3150ms 48.17%  github.com/eriklupander/profiling/cpu.prime
    2190ms 33.49% 81.65%     2190ms 33.49%  github.com/eriklupander/profiling/cpu.SumRoots
     810ms 12.39% 94.04%     3000ms 45.87%  github.com/eriklupander/profiling/cpu.CPULoader
     310ms  4.74% 98.78%      310ms  4.74%  runtime.nanotime
      70ms  1.07% 99.85%     6220ms 95.11%  github.com/eriklupander/profiling/cpu.CPU
         0     0% 99.85%     6230ms 95.26%  main.main
         0     0% 99.85%     6230ms 95.26%  runtime.main
         0     0% 99.85%      310ms  4.74%  runtime.mstart
         0     0% 99.85%      310ms  4.74%  runtime.mstart1
         0     0% 99.85%      310ms  4.74%  runtime.sysmon
```

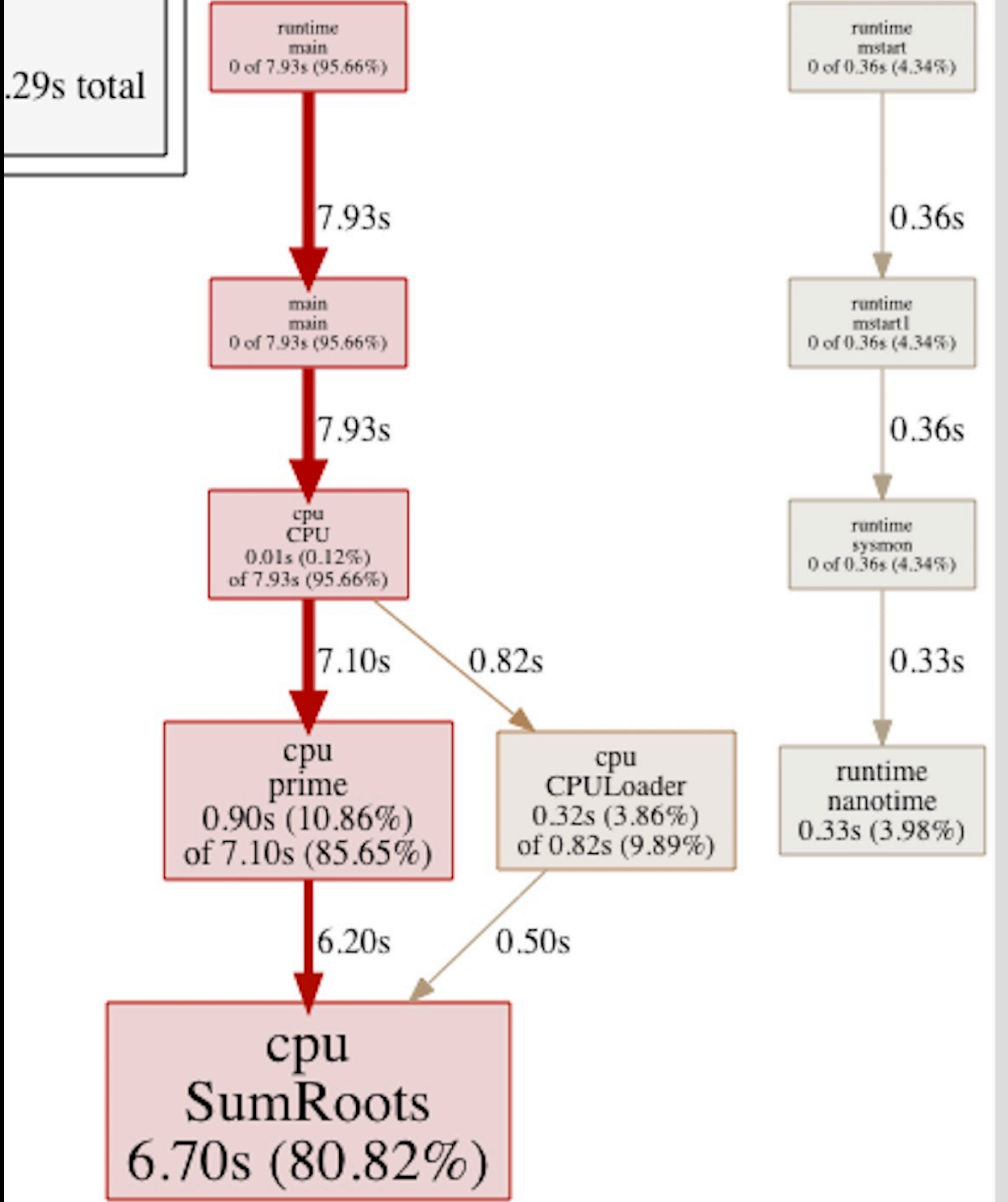
PPROF - VIZ GRAPHS

- Bigger boxes or thicker arrows means more time spent
- Shows call hierarchies
- Numbers:
  - profile / mutex / blocks: time in ms
  - heap: memory size in MB
  - heap allocs: number of allocs



PPROF - VIZ GRAPHS

- Bigger boxes or thicker arrows means more time spent
- Shows call hierarchies
- Numbers:
  - profile / mutex / blocks: time in ms
  - heap: memory size in MB
  - heap allocs: number of allocs
- Many-to-one

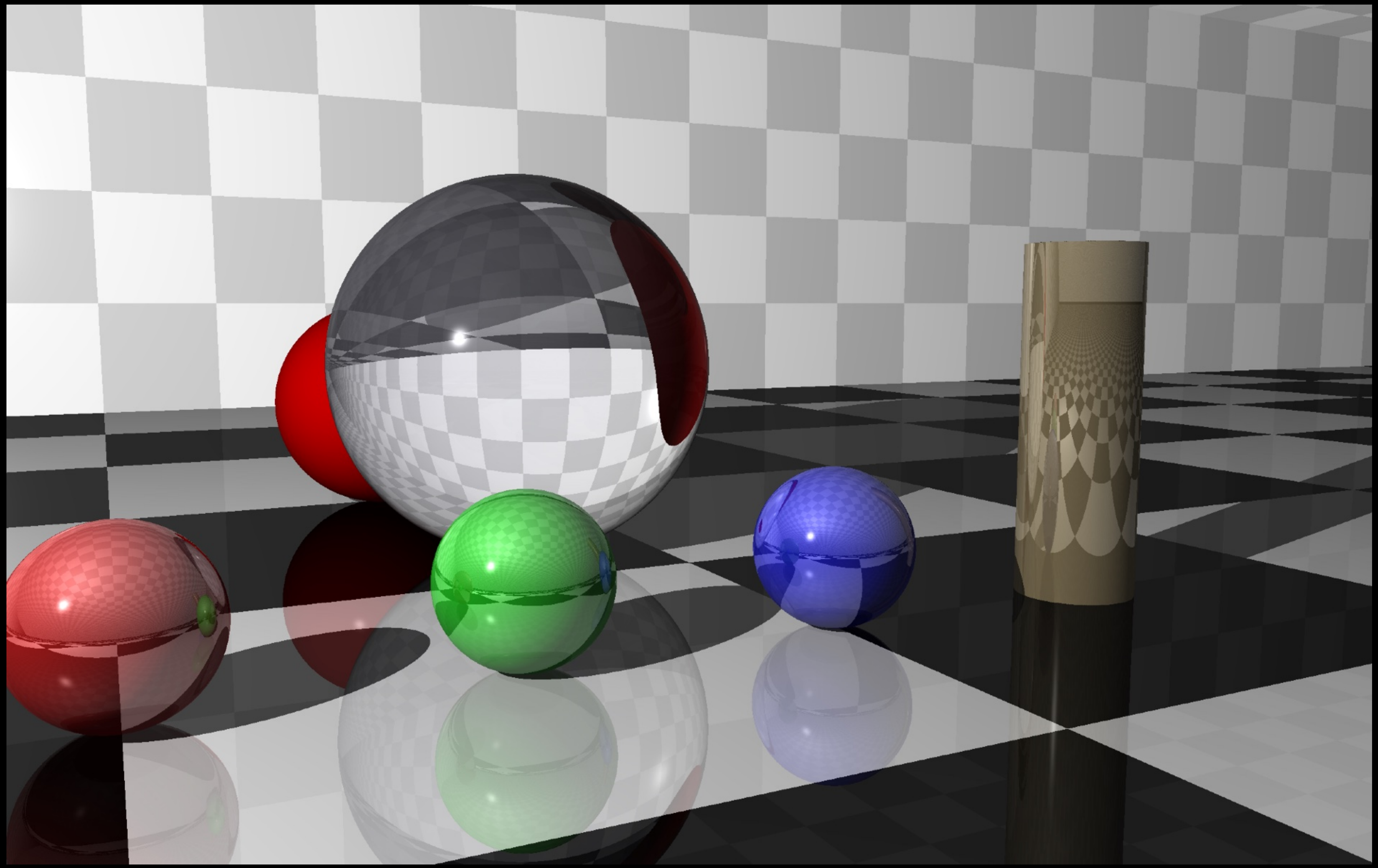


PPROF CONSOLE - LISTING

```
(pprof) list CPULoader
Total: 6.54s
ROUTINE ======================== github.com/eriklupander/profiling/cpu.CPULoader in /Users/eriklupander/privat/profiling/cpu/l
oad.go
     810ms         3s (flat, cum) 45.87% of Total
         .          .     23:func CPULoader(times int) []int {
         .          .     24:   // Keeps it to a 32 bit int
         .          .     25:   //num := 40
         .          .     26:   var r []int
         .          .     27:   result := false
      50ms       50ms     28:   for n := 0; n < times*200; n++ {
         .          .     29:           if n%1 != 0 {
         .          .     30:                   result = false
         .          .     31:           } else if n <= 1 {
         .          .     32:                   result = false
         .          .     33:           } else if n <= 3 {
         .          .     34:                   result = true
         .          .     35:           } else if n%2 == 0 {
         .          .     36:                   result = false
         .          .     37:           }
     290ms      290ms     38:           dl := int(math.Sqrt(float64(n)))
     330ms      330ms     39:           for d := 3; d <= dl; d += 2 {
     140ms      140ms     40:                   if n%d == 0 {
         .          .     41:                           result = false
         .          .     42:                   }
         .          .     43:           }
         .          .     44:           result = true
         .          .     45:   }
         .      2.19s     46:   sum := SumRoots(result)
         .          .     47:   if int(sum) % 1000  == 0 {
         .          .     48:           fmt.Print(".")
         .          .     49:   }
         .          .     50:   return r
         .          .     51:}
```

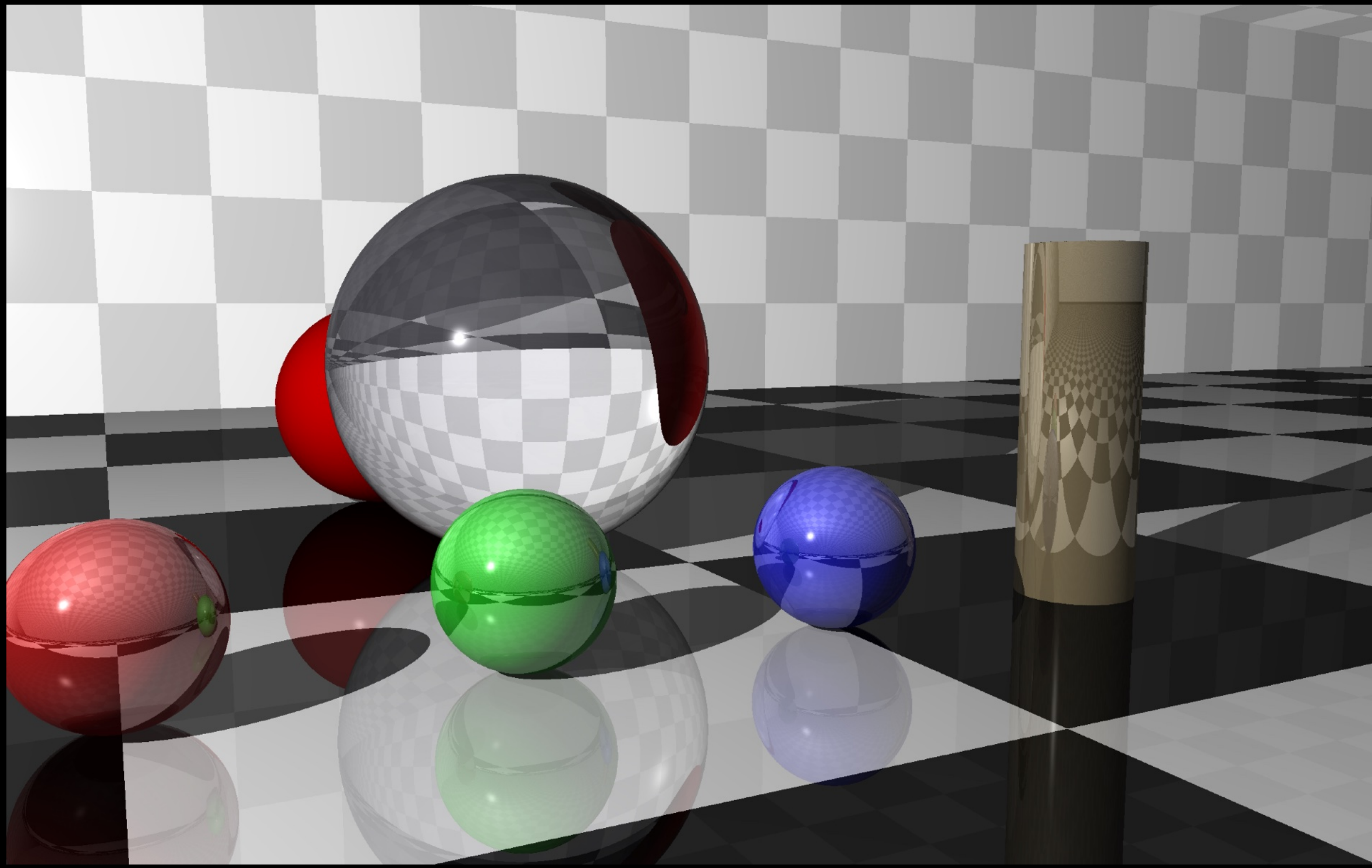
MY PET-PROJECT PPROF USE-CASE

FIND THE DIFFERENCE



RENDER #1

3 min 14sec

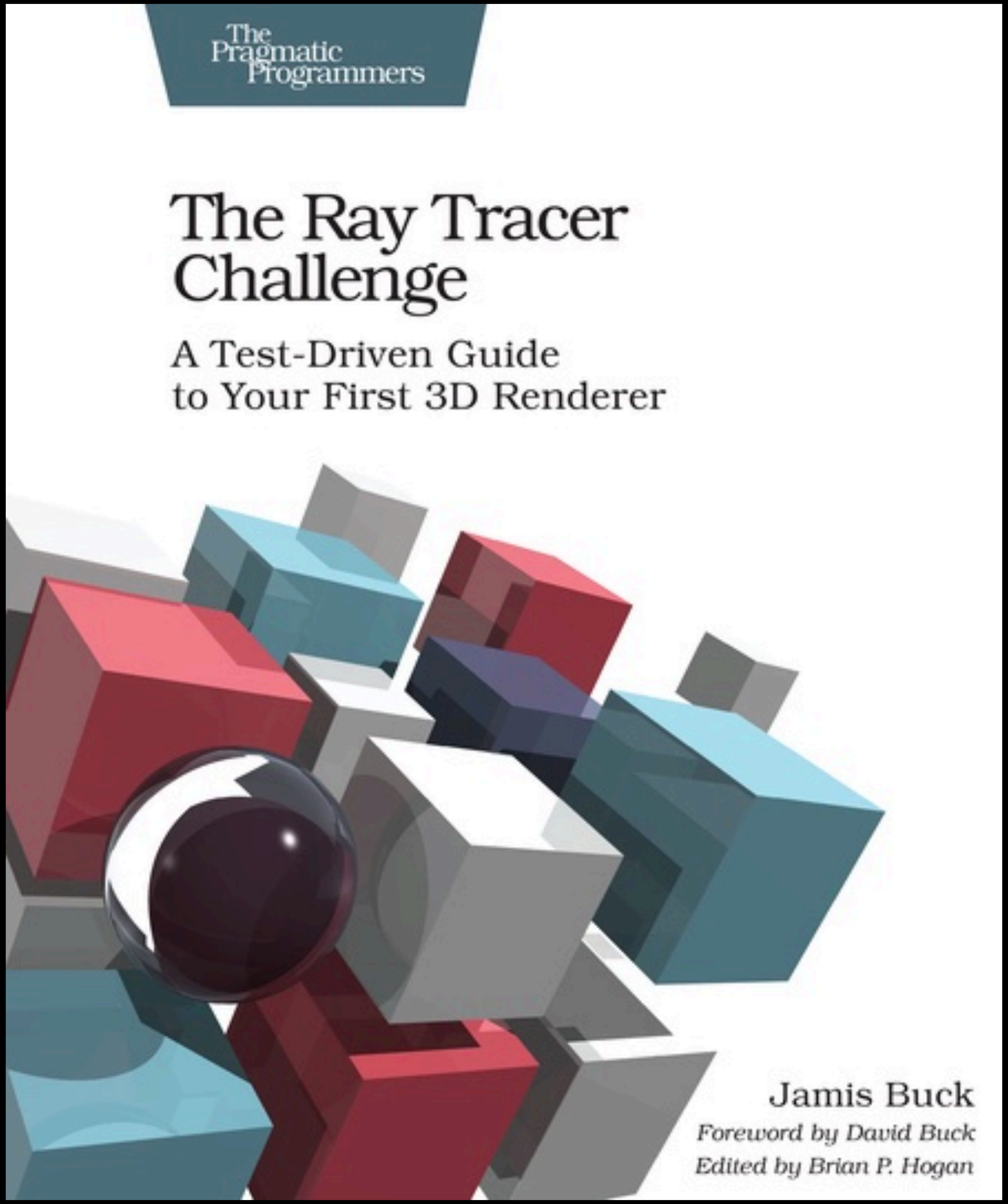


RENDER #2

1.6 sec

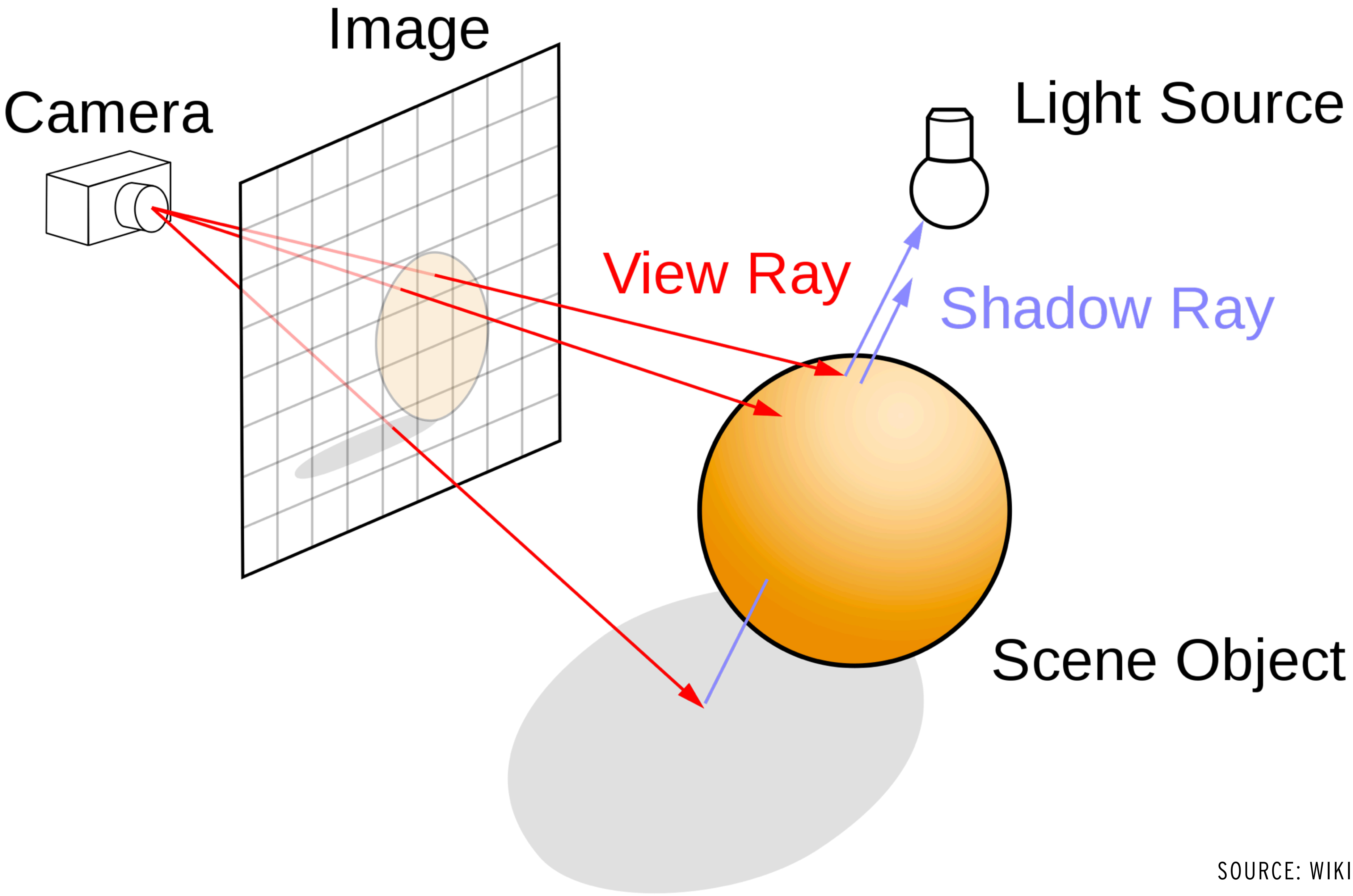
WHY RAY-TRACING?

- Just for fun!
- Book: "The Ray Tracer challenge"
- Relatively simple renderer
- CPU intensive task, good fit for profiling

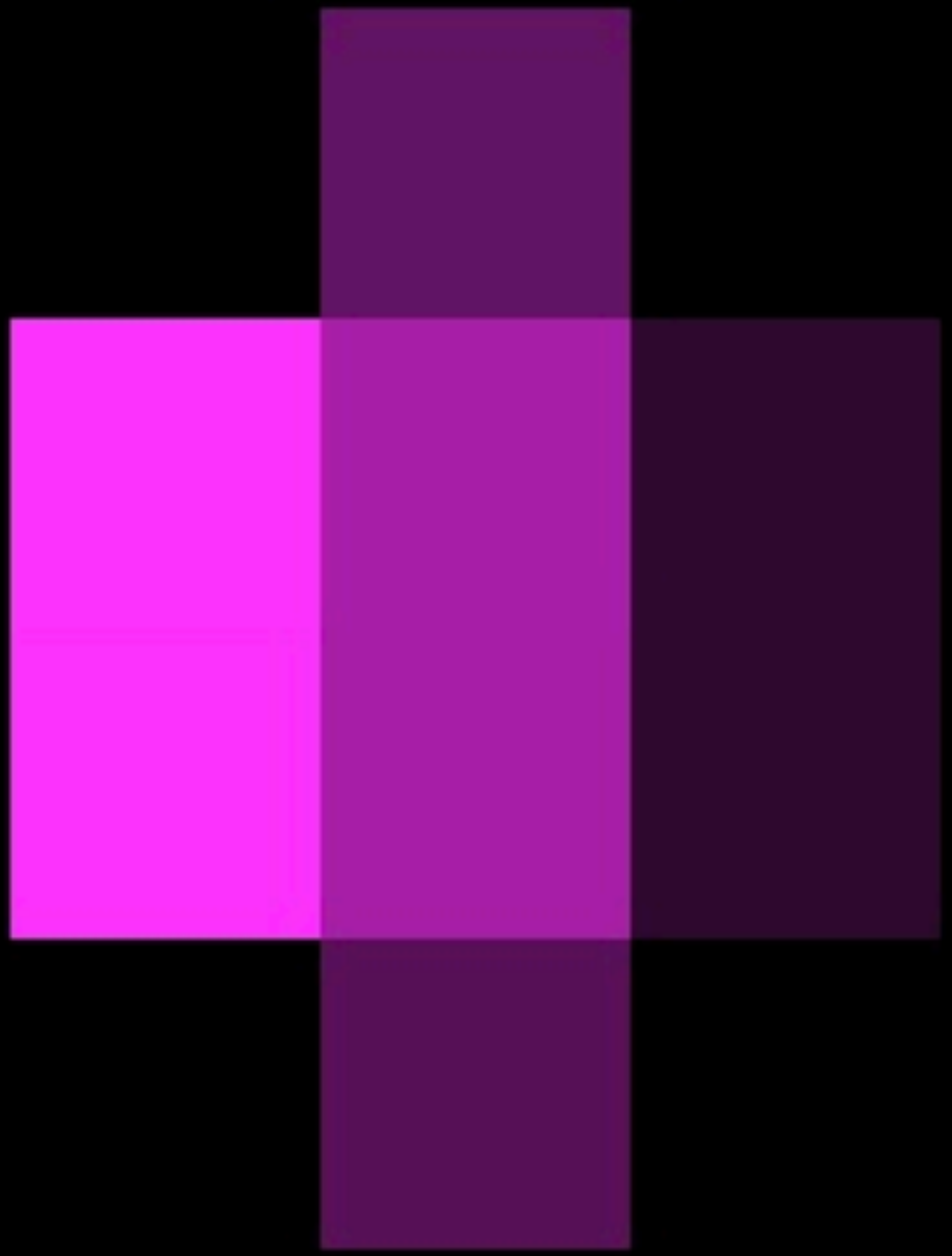


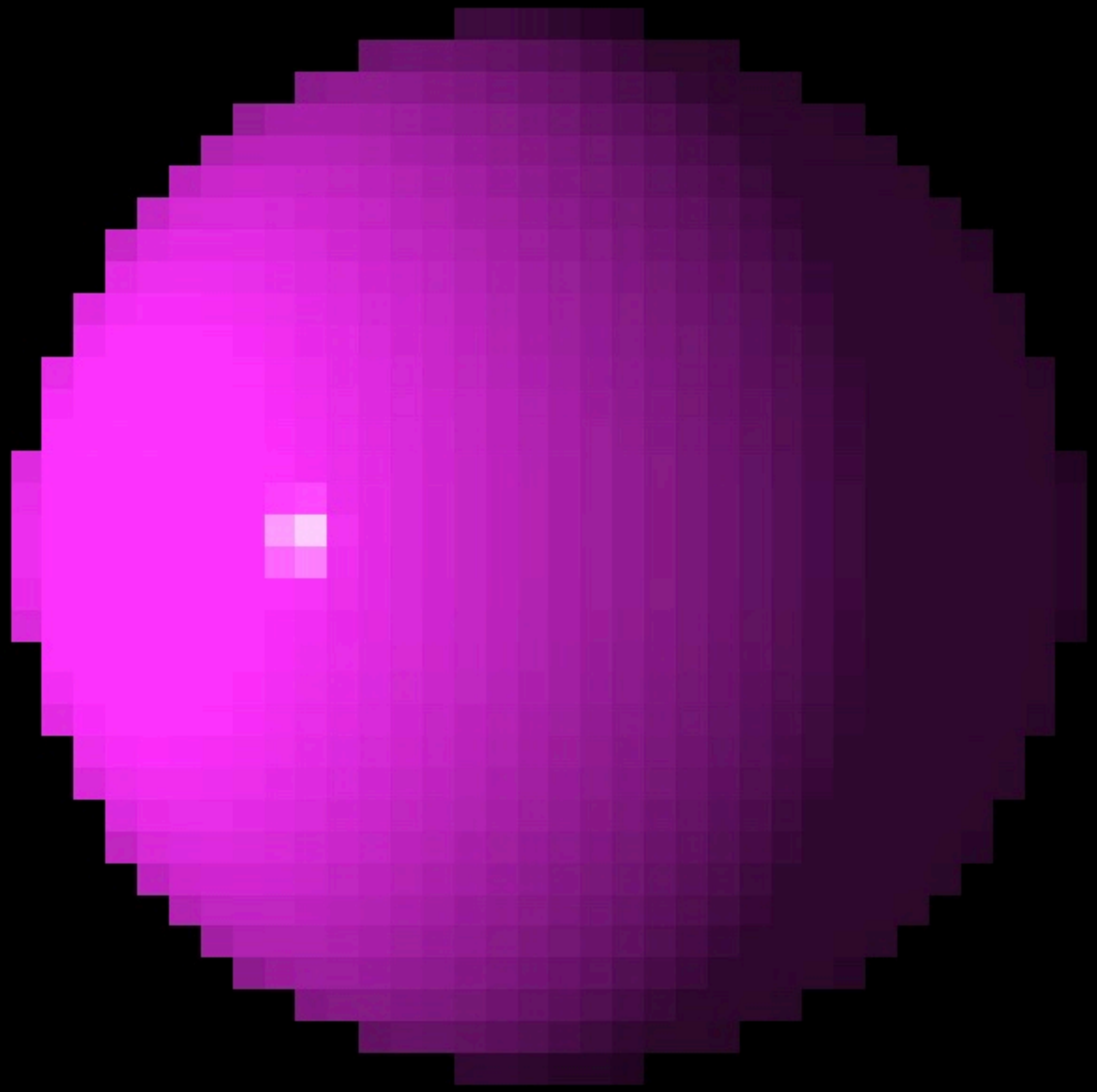
PRAGPROG.ORG

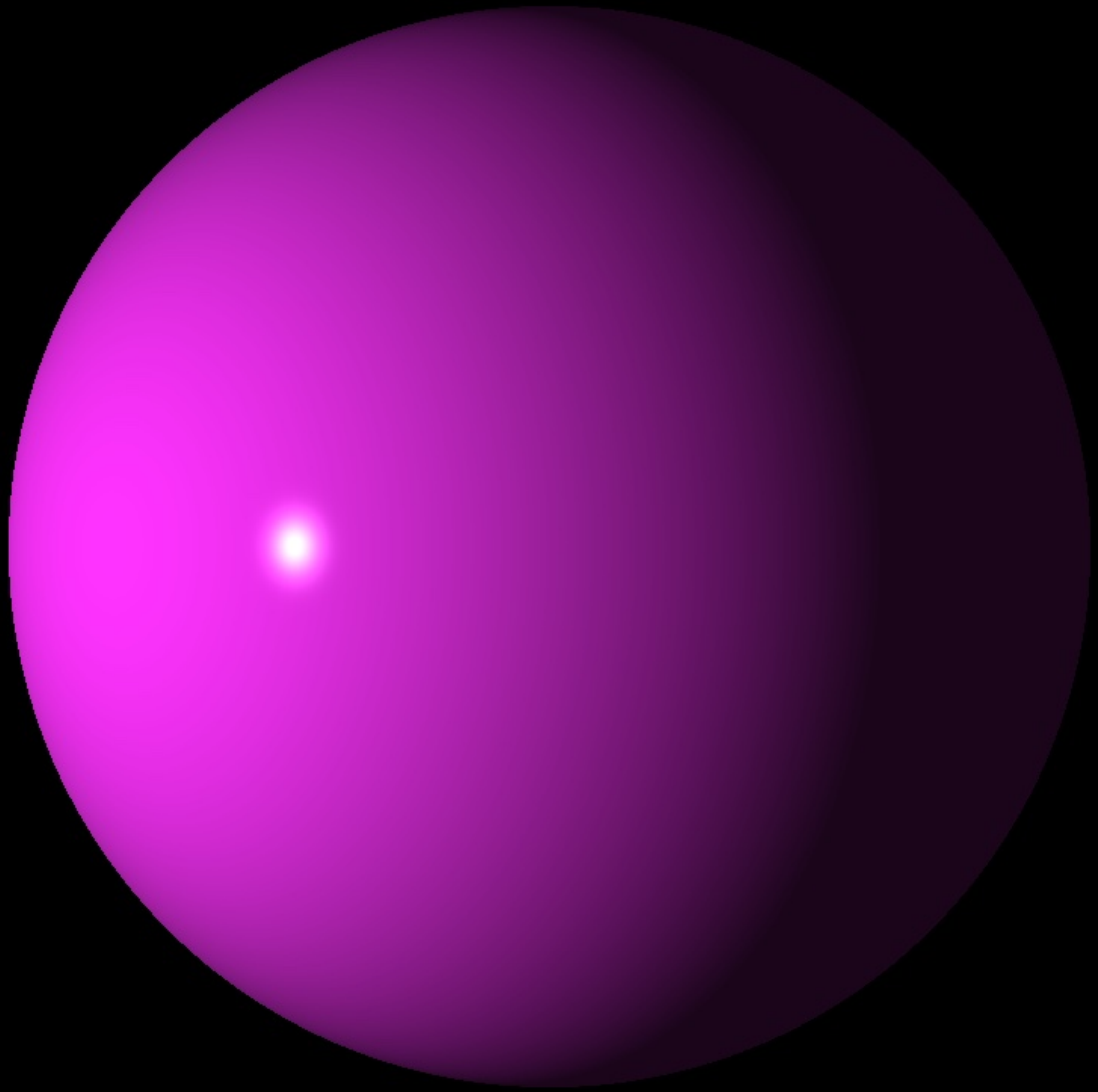
RAY-TRACING IN 3 MINUTES

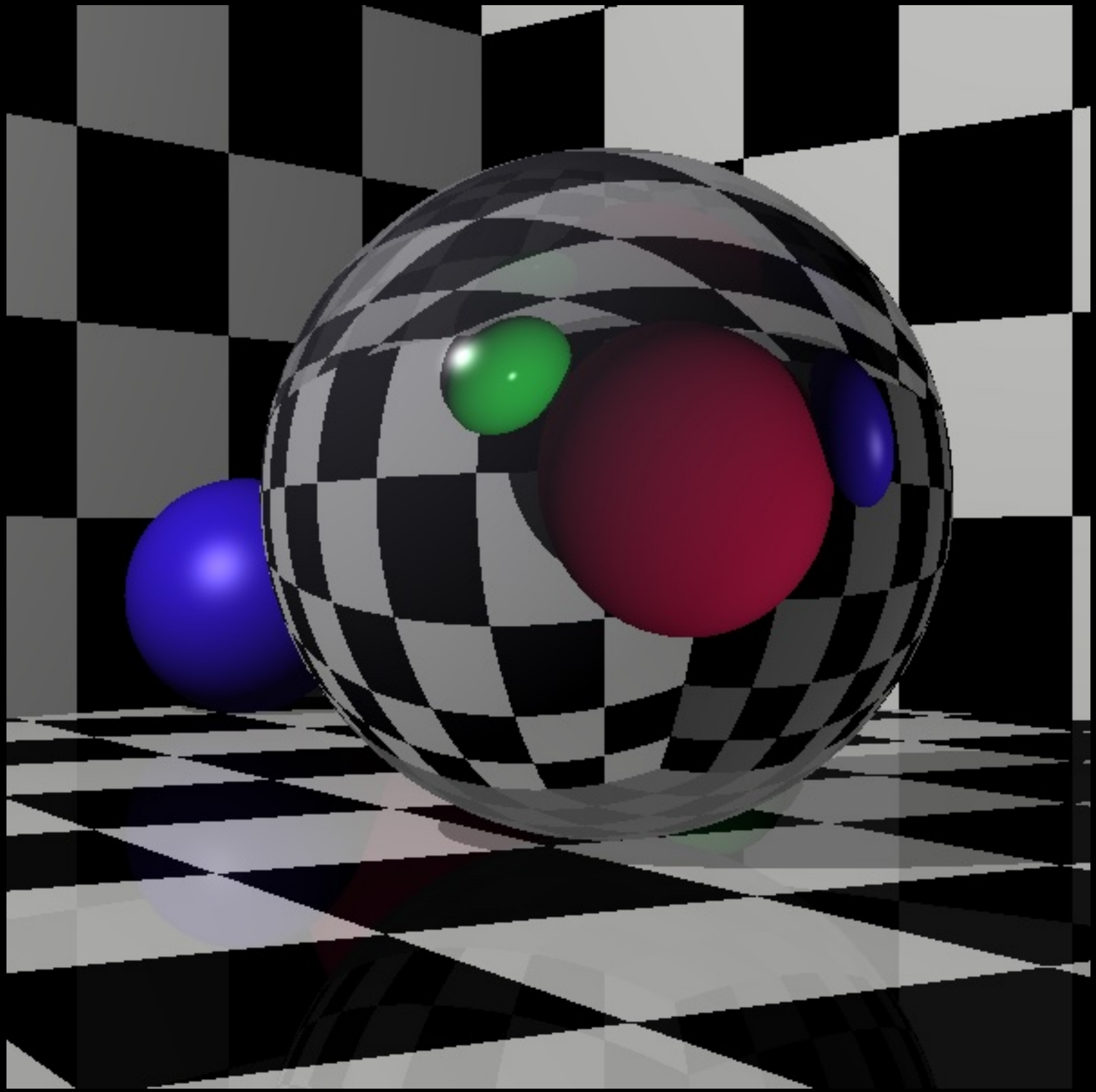


SOURCE: WIKIPEDIA







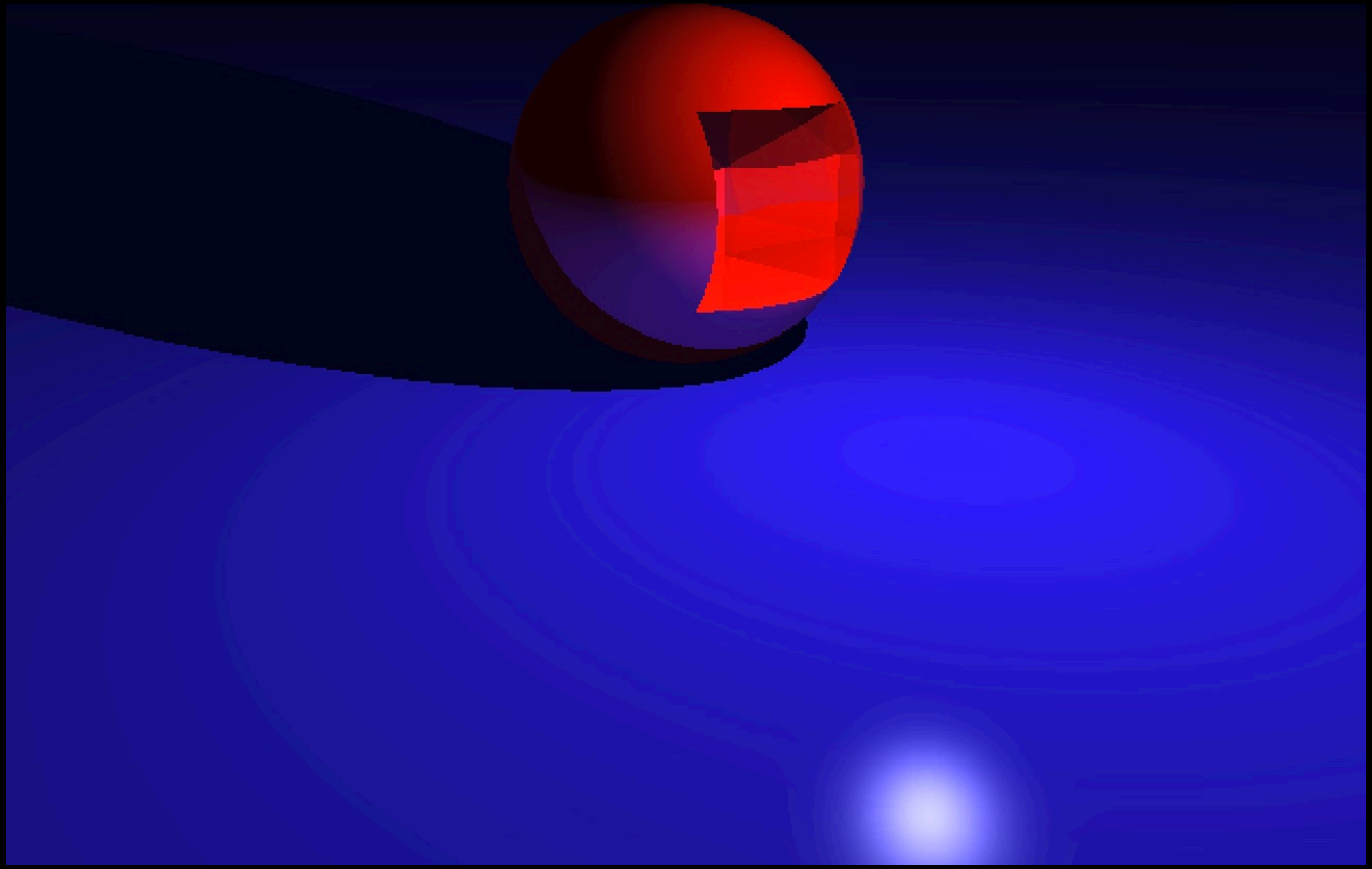


IT'S ALL ABOUT THE COLOR

OF EVERY SINGLE PIXEL

WHY PROFILING?

- Once the book was finished, rendering was rather slow.

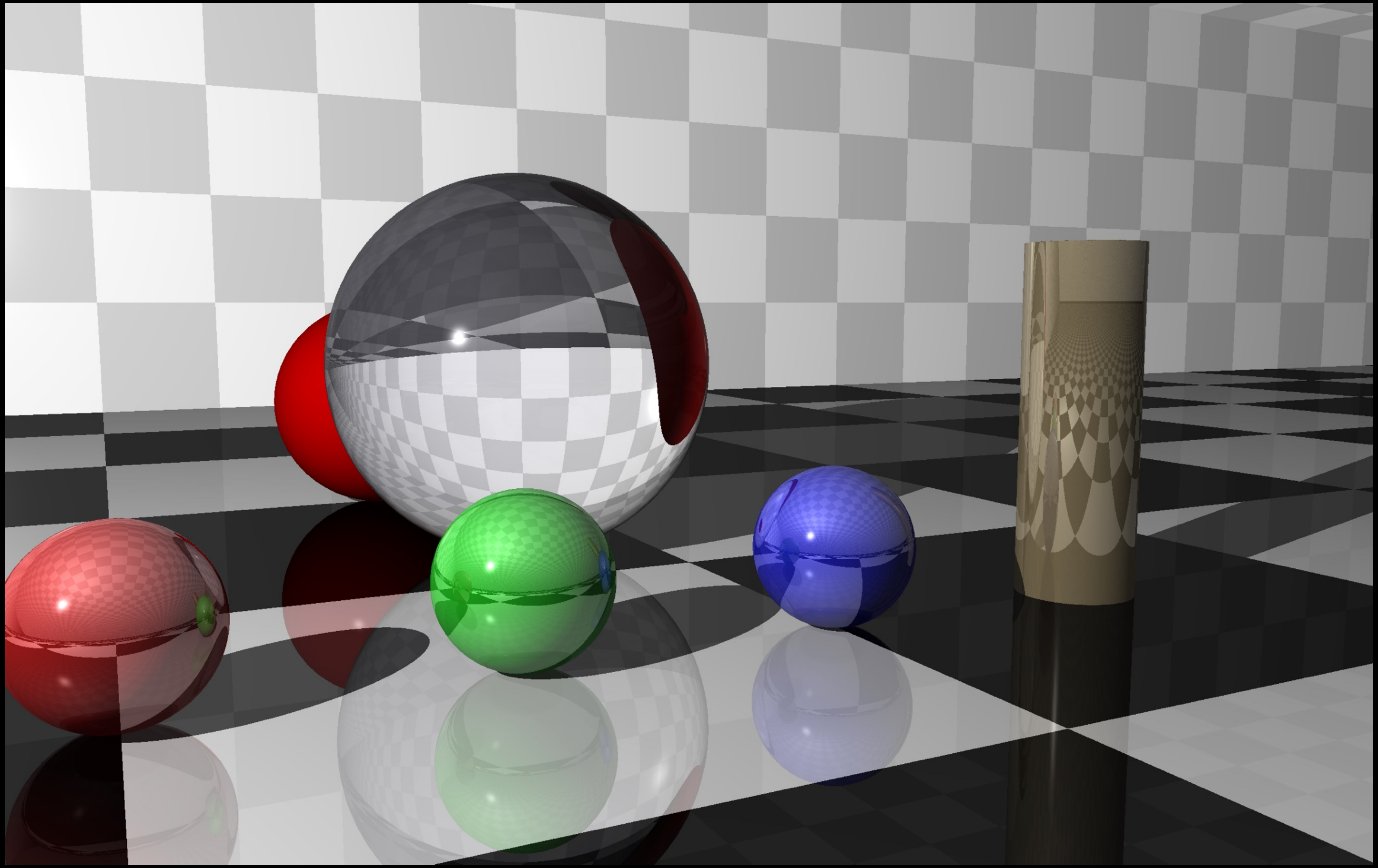


NAIVE IMPLEMENTATION

- Single-threaded
- Plain Go code
  - No 3rd party libraries for math etc.
- Correctness over premature optimization
  - No caching, prefer immutability
  - ”… never-ending series of headaches…”

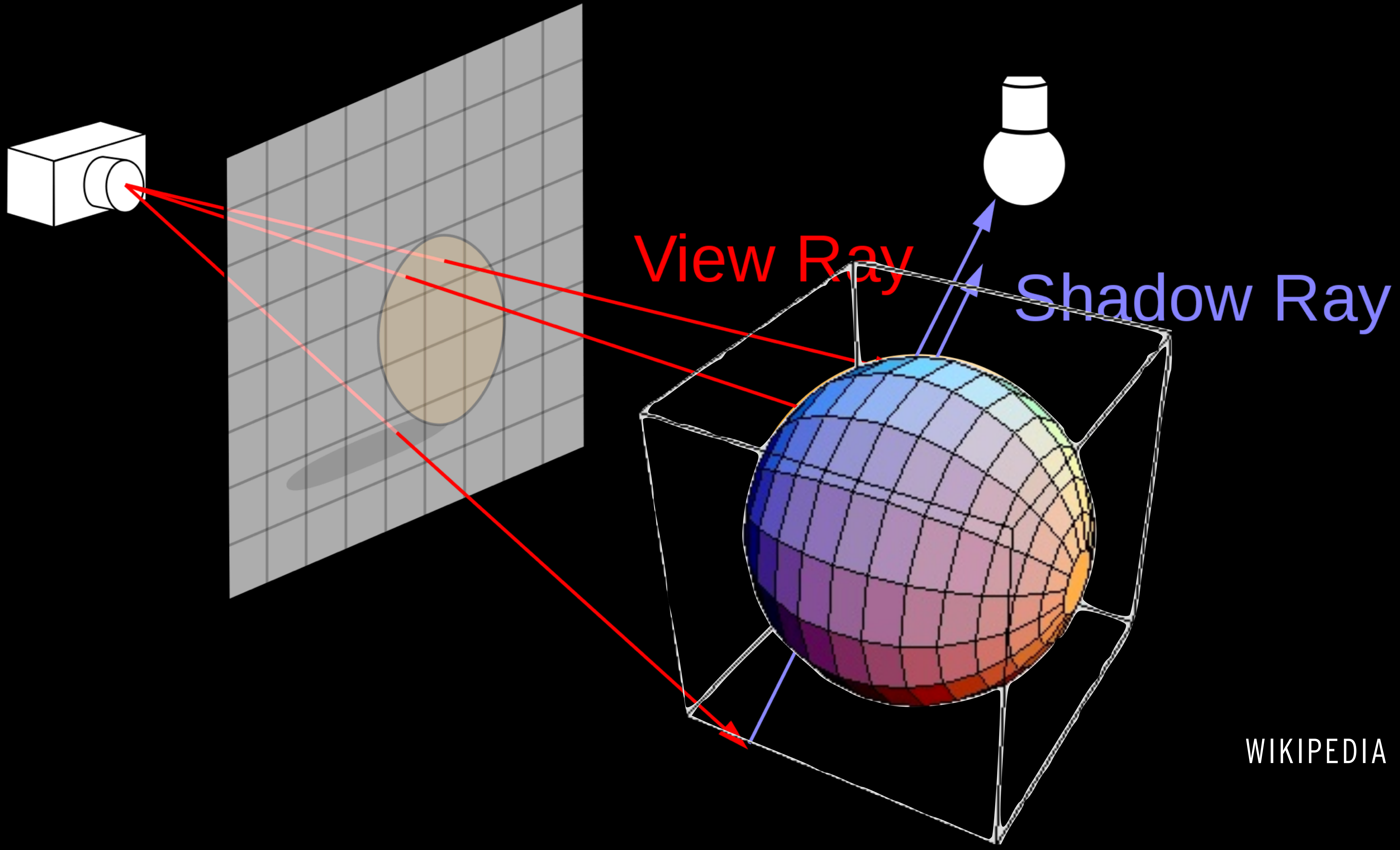
REFERENCE IMAGE

- Reference image
  - 9 primitives
  - Reflection and refraction
- At 640x480:
  - 307 200 pixels
  - Limited recursion depth
    - » Max 5 reflections and 5 refractions per ray



TWO OPTIMIZATION ROUTES

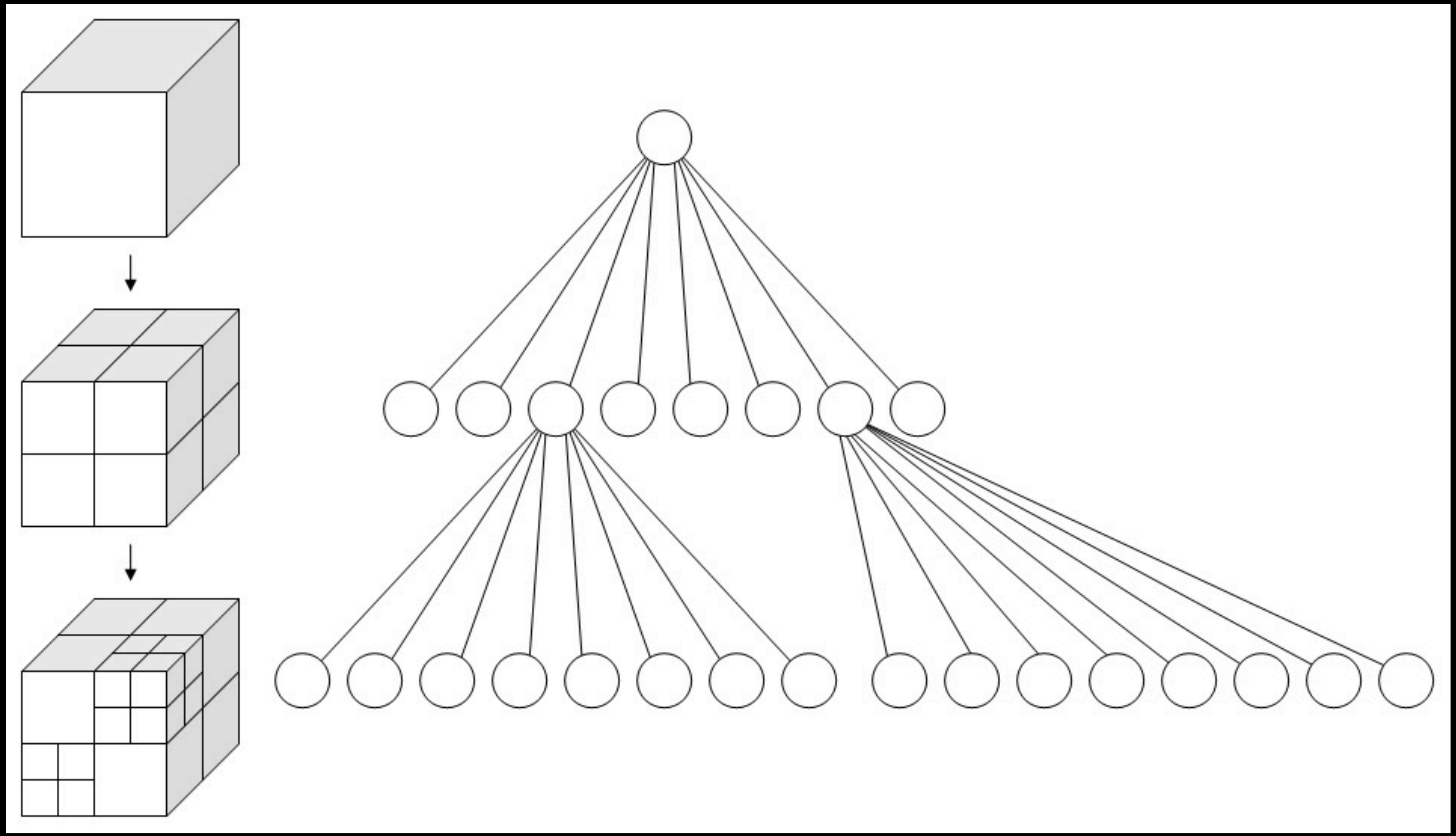
- Algorithm specific:
  - ”Do Less Work”
    - » Bounding boxes



WIKIPEDIA

TWO OPTIMIZATION ROUTES

- Algorithm specific:
  - "Do Less Work"
    - Bounding boxes
      - BVH
  - Reduce number of intersection checks
- Implementation specific:
  - Use Go profiling tools to find bottlenecks and optimize accordingly



WIKIPEDIA

STEP 1 - MULTI-THREADING

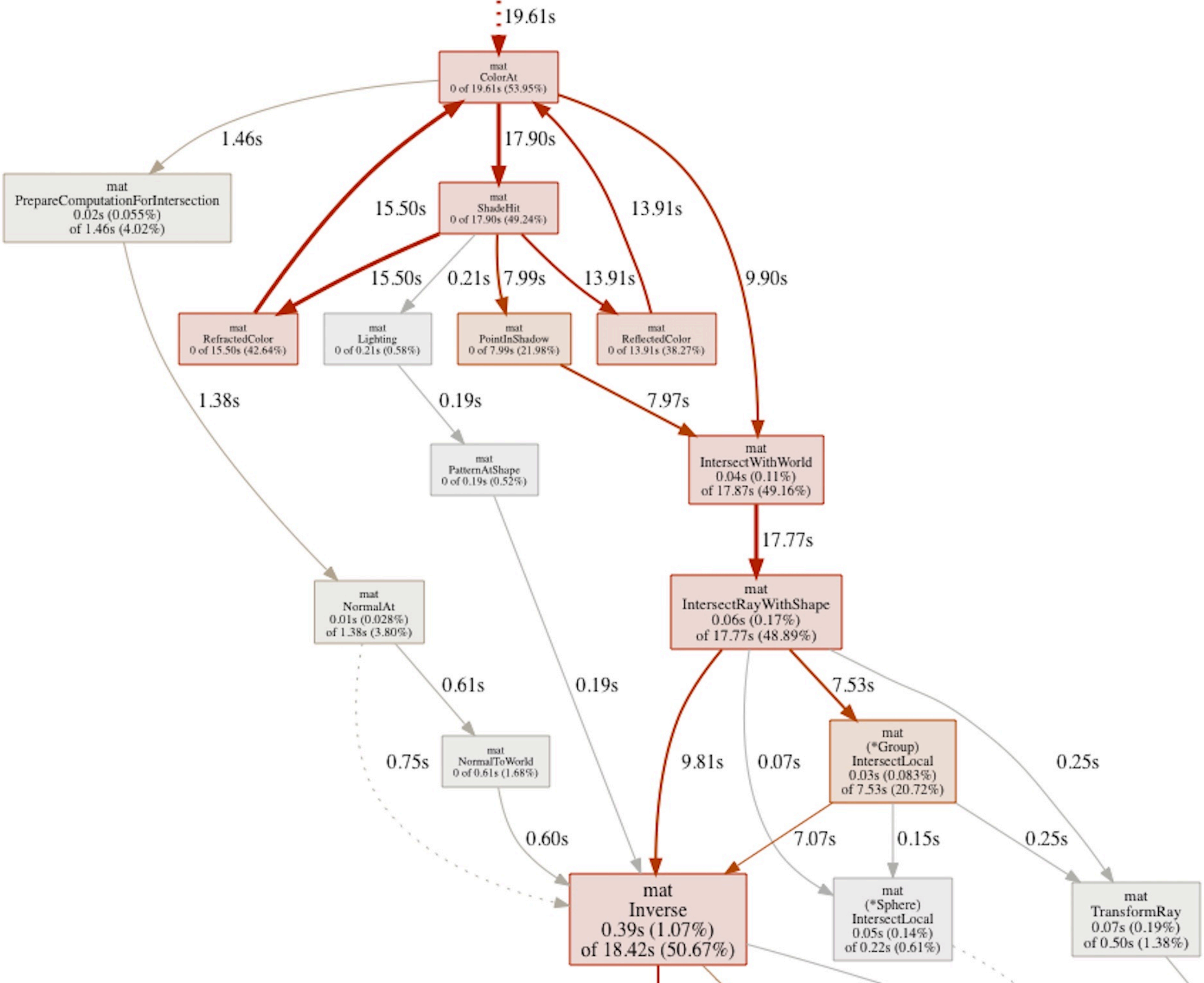
- ”Embarrassingly parallell problem”
- Worker-pool implementation
- 1 -> 8 threads
- Performance improved performance by:

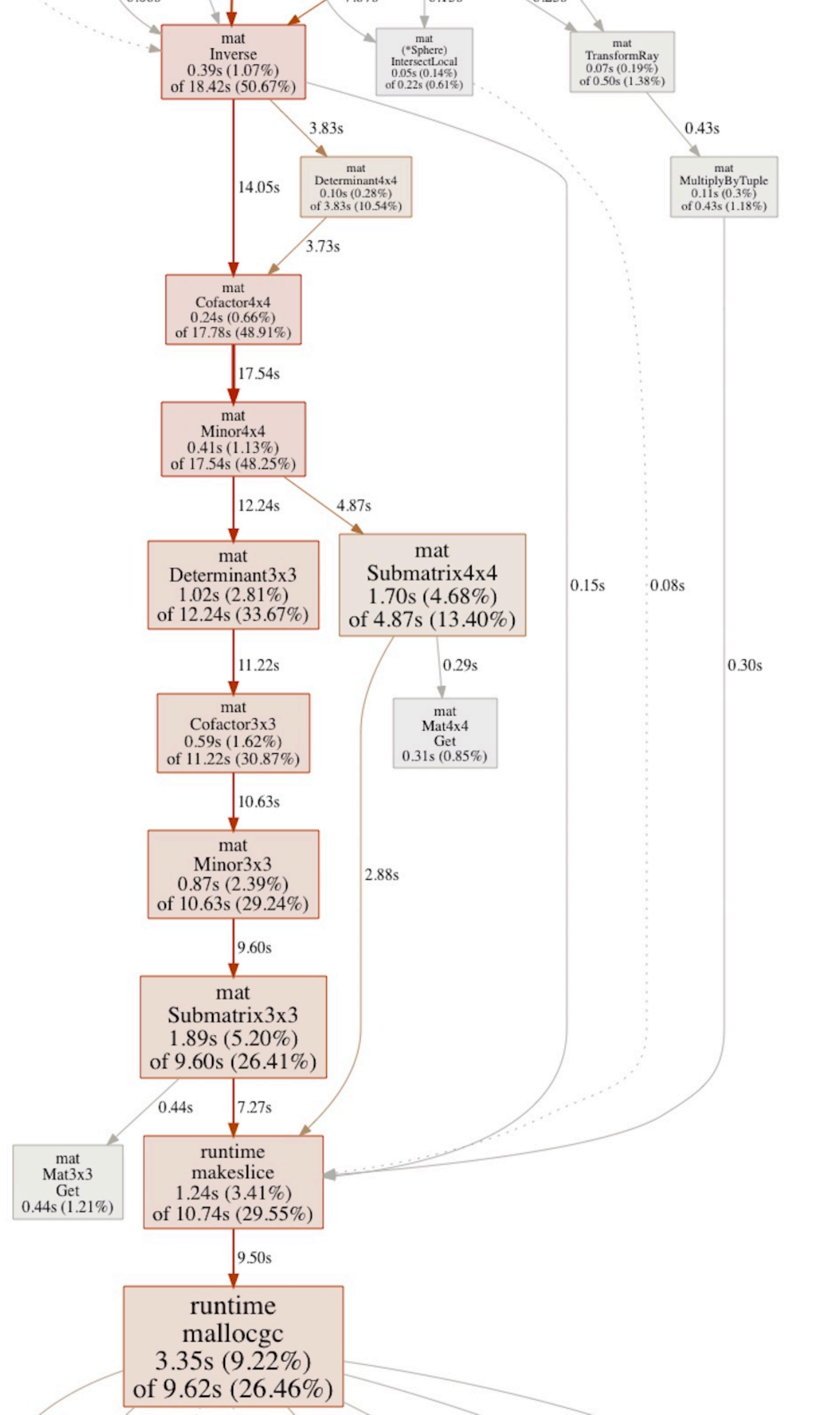
2.25x

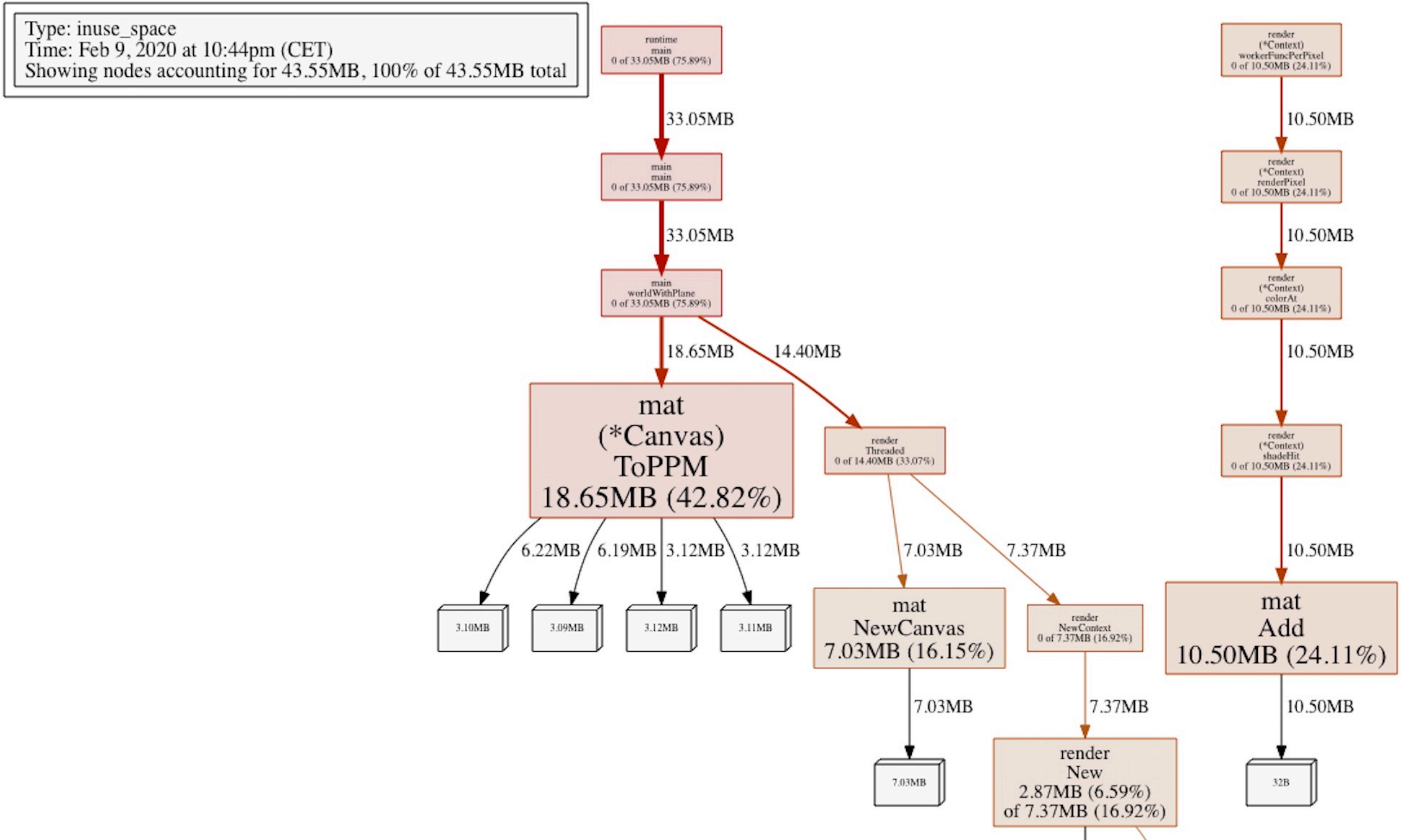
~1 min 30sec

FIRST RUN OF PPROF - CPU PROFILING

- I added the pprof HTTP boilerplate code and then captured a 30-second time window using */debug/pprof/profile* with .PNG export


19.61s
mat
ColorAt
0 of 19.61s (53.95%)
1.46s
17.90s
mat
PrepareComputationForIntersection
0.02s (0.055%)
of 1.46s (4.02%)
15.50s
mat
ShadeHit
0 of 17.90s (49.24%)
13.91s
15.50s
0.21s
7.99s
13.91s
9.90s
mat
RefractedColor
0 of 15.50s (42.64%)
mat
Lighting
0 of 0.21s (0.58%)
mat
PointInShadow
0 of 7.99s (21.98%)
mat
ReflectedColor
0 of 13.91s (38.27%)
1.38s
0.19s
7.97s
mat
PatternAtShape
0 of 0.19s (0.52%)
mat
IntersectWithWorld
0.04s (0.11%)
of 17.87s (49.16%)
17.77s
mat
NormalAt
0.01s (0.028%)
of 1.38s (3.80%)
mat
IntersectRayWithShape
0.06s (0.17%)
of 17.77s (48.89%)
0.61s
0.19s
7.53s
0.75s
mat
NormalToWorld
0 of 0.61s (1.68%)
9.81s
0.07s
mat
(*Group)
IntersectLocal
0.03s (0.083%)
of 7.53s (20.72%)
0.25s
0.60s
7.07s
0.15s
0.25s
mat
Inverse
0.39s (1.07%)
of 18.42s (50.67%)
mat
(*Sphere)
IntersectLocal
0.05s (0.14%)
of 0.22s (0.61%)
mat
TransformRay
0.07s (0.19%)
of 0.50s (1.38%)


mat
Inverse
0.39s (1.07%)
of 18.42s (50.67%)
mat
(*Sphere)
IntersectLocal
0.05s (0.14%)
of 0.22s (0.61%)
mat
TransformRay
0.07s (0.19%)
of 0.50s (1.38%)
3.83s
0.43s
mat
Determinant4x4
0.10s (0.28%)
of 3.83s (10.54%)
mat
MultiplyByTuple
0.11s (0.3%)
of 0.43s (1.18%)
14.05s
3.73s
mat
Cofactor4x4
0.24s (0.66%)
of 17.78s (48.91%)
17.54s
mat
Minor4x4
0.41s (1.13%)
of 17.54s (48.25%)
12.24s
4.87s
mat
Determinant3x3
1.02s (2.81%)
of 12.24s (33.67%)
mat
Submatrix4x4
1.70s (4.68%)
of 4.87s (13.40%)
0.15s
0.08s
0.30s
11.22s
0.29s
mat
Cofactor3x3
0.59s (1.62%)
of 11.22s (30.87%)
mat
Mat4x4
Get
0.31s (0.85%)
10.63s
mat
Minor3x3
0.87s (2.39%)
of 10.63s (29.24%)
2.88s
9.60s
mat
Submatrix3x3
1.89s (5.20%)
of 9.60s (26.41%)
0.44s
7.27s
mat
Mat3x3
Get
0.44s (1.21%)
runtime
makeslice
1.24s (3.41%)
of 10.74s (29.55%)
9.50s
runtime
mallocgc
3.35s (9.22%)
of 9.62s (26.46%)

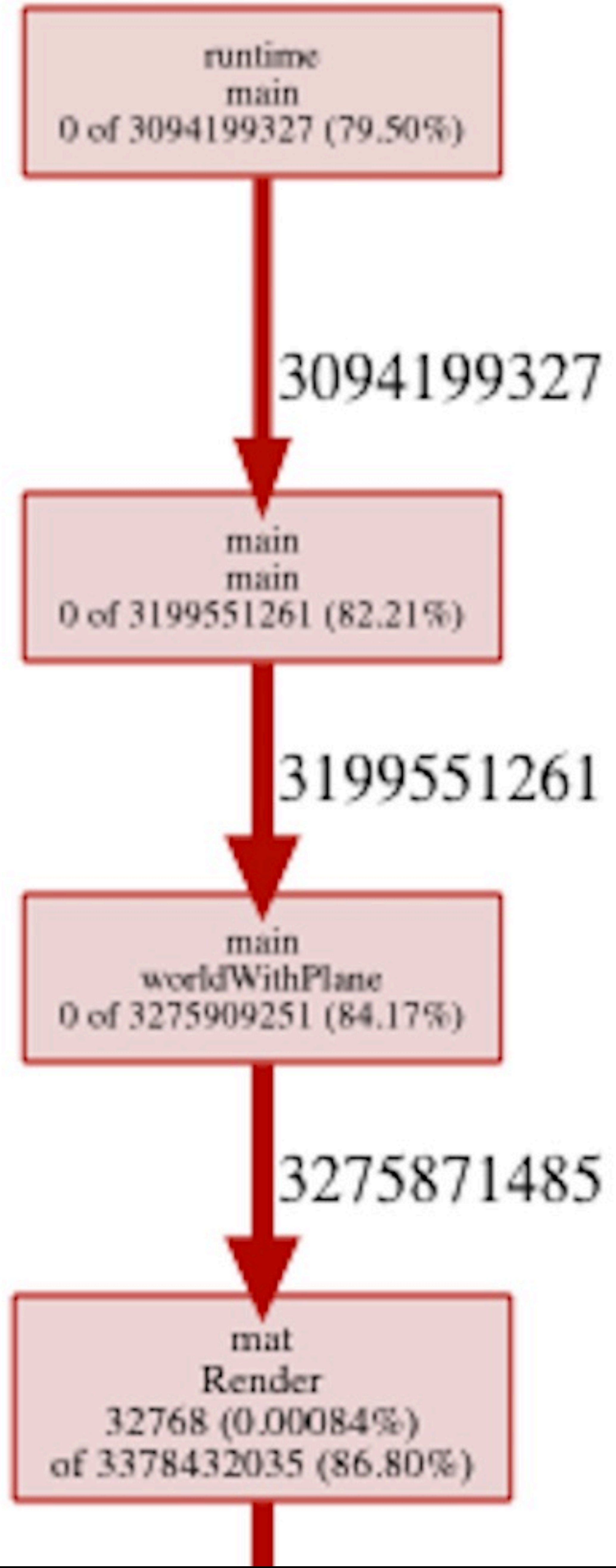


Type: inuse_space
Time: Feb 9, 2020 at 10:44pm (CET)
Showing nodes accounting for 43.55MB, 100% of 43.55MB total
runtime main 0 of 33.05MB (75.89%)
33.05MB
main main 0 of 33.05MB (75.89%)
33.05MB
main worldWithPlane 0 of 33.05MB (75.89%)
18.65MB
14.40MB
mat (*Canvas) ToPPM 18.65MB (42.82%)
6.22MB
6.19MB
3.12MB
3.12MB
3.10MB
3.09MB
3.12MB
3.11MB
render Threaded 0 of 14.40MB (33.07%)
7.03MB
7.37MB
mat NewCanvas 7.03MB (16.15%)
7.03MB
7.03MB
render NewContext 0 of 7.37MB (16.92%)
7.37MB
render New 2.87MB (6.59%) of 7.37MB (16.92%)
render (*Context) workerFuncPerPixel 0 of 10.50MB (24.11%)
10.50MB
render (*Context) renderPixel 0 of 10.50MB (24.11%)
10.50MB
render (*Context) colorAt 0 of 10.50MB (24.11%)
10.50MB
render (*Context) shadeHit 0 of 10.50MB (24.11%)
10.50MB
mat Add 10.50MB (24.11%)
10.50MB
32B

HEAP - MEMORY USE / ALLOCATIONS?

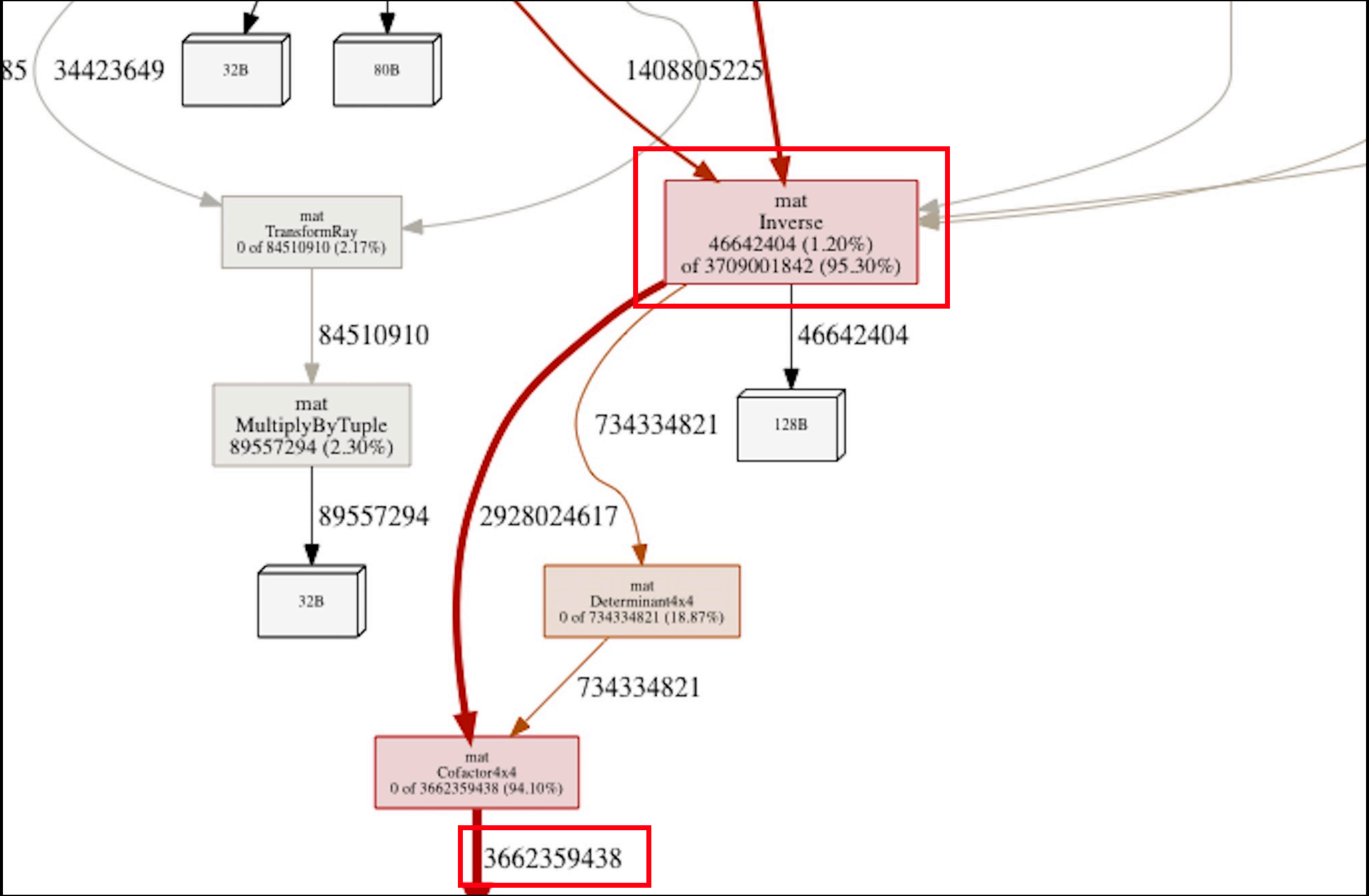
- Heap size seemed OK at 44 mb
- Could we be performing an excessive number of memory allocations?
- pprof does that too with the -alloc_objects flag!
  - go tool pprof **-alloc_objects** -png http://localhost:6060/debug/pprof/heap

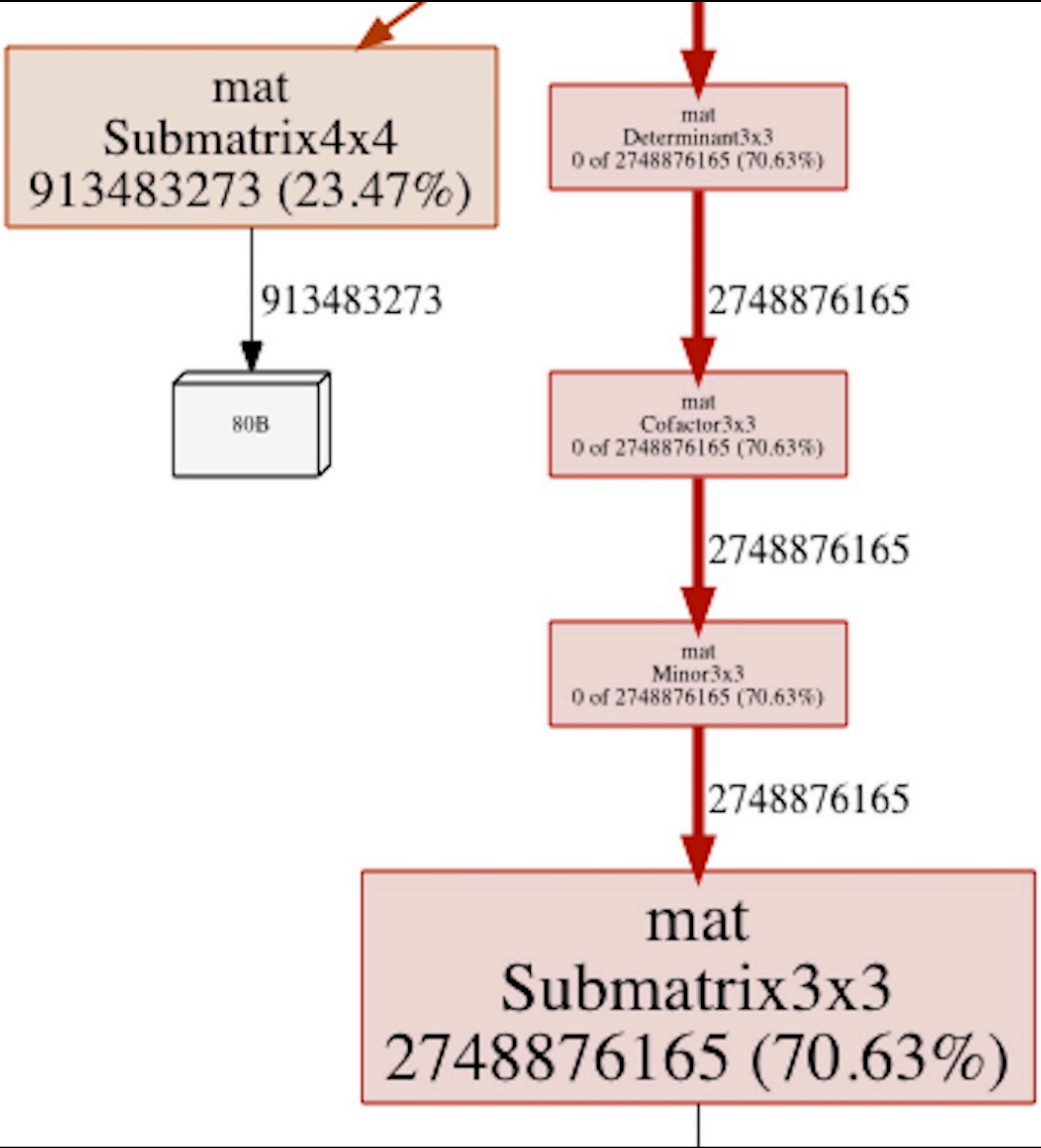
Type: alloc_objects
Time: Dec 22, 2019 at 5:29pm (CET)
Showing nodes accounting for 3842689091, 98.73% of 3892008614 total
Dropped 44 nodes (cum <= 19460043)



~12 700 allocations per pixel!

154 GB of RAM allocated


85
34423649
32B
80B
1408805225
mat
TransformRay
0 of 84510910 (2.17%)
mat
Inverse
46642404 (1.20%)
of 3709001842 (95.30%)
84510910
46642404
mat
MultiplyByTuple
89557294 (2.30%)
734334821
128B
89557294
2928024617
32B
mat
Determinant4x4
0 of 734334821 (18.87%)
734334821
mat
Cofactor4x4
0 of 3662359438 (94.10%)
3662359438


mat
Submatrix4x4
913483273 (23.47%)
913483273
80B
mat
Determinant3x3
0 of 2748876165 (70.63%)
2748876165
mat
Cofactor3x3
0 of 2748876165 (70.63%)
2748876165
mat
Minor3x3
0 of 2748876165 (70.63%)
2748876165
mat
Submatrix3x3
2748876165 (70.63%)

WHAT ARE THE INVERSE() AND SUBMATRIX() FUNCTIONS DOING!?!?

STEP 2 - FIX INVERSE() AND SUBMATRIX

```
func Submatrix4x4(m1 Mat4x4, deleteRow, deleteCol int) Mat3x3 {
    m3 := NewMat3x3(make([]float64, 9))
    idx := 0
    for row := 0; row < 4; row++ {
        if row == deleteRow {
            continue
        }
        for col := 0; col < 4; col++ {
            if col == deleteCol {
                continue
            }
            m3.Elems[idx] = m1.Get(row, col)
            idx++
        }
    }
    return m3
}
```

CACHING THE INVERSE

- The Inverse transformation matrix of each primitive is used in every ray / object intersection test
- Since our geometry and camera is static per frame rendered, it turns out we can pre-compute and store the Inverse matrix for each primitive once during scene setup.

```
middle = mat.NewSphere()
middle.SetTransform(mat.Translate(-0.5, 0.75, 0.5))

glassMtrl := mat.NewMaterial(mat.NewColor(0.8, 0.8, 0.9), 0, 0.2, 0.9, 300)
glassMtrl.Transparency = 1.0
glassMtrl.RefractiveIndex = 1.57

func (s *Sphere) SetTransform(translation Mat4x4) {
    s.Transform = Multiply(s.Transform, translation)
    s.Inverse = Inverse(s.Transform)
}
```

CACHING THE INVERSE

- The Inverse transformation matrix of each primitive is used in every ray / object intersection test
- Since our geometry and camera is static per frame rendered, it turns out we can pre-compute and store the Inverse matrix for each primitive once during scene setup.

```
type Sphere struct {
    Id        int64
    Transform Mat4x4
    Inverse   Mat4x4
    Material  Material
}

func (s *Sphere) SetTransform(translation Mat4x4) {
    s.Transform = Multiply(s.Transform, translation)
    s.Inverse = Inverse(s.Transform)
}
```

BEST OPTIMIZATION EVER!

INVERSE CACHING OUTCOME

- Single-threaded went from 3m 14s to 10.9s
- Multi-threaded went from 1m30s to 4.2s
- Allocations went from 3.9 billion to 180 million!
- From 154 GB to 5.9 GB

NOT DONE YET!

STEP 3 - ELIMINATE ALLOCATIONS

- Still room for improvement
- Time for a new pprof check of allocs to the heap


mat
IntersectWithWorld
7820925 (4.39%)
of 126678722 (71.11%)
mat
Reflect
0 of 7438504 (4.18%)
mat
Position
0 of 2785334 (1.56%)
mat
Negate
3260482 (1.83%)
mat
NormalAt
0 of 8552777 (4.80%)
mat
MultiplyByScalar
16105798 (9.04%)
mat
Add
9240773 (5.19%)
mat
IntersectRayWithShape
0 of 116465651 (65.38%)
mat
(*Group)
IntersectLocal
3714339 (2.09%)
of 52032028 (29.21%)
mat
(*Plane)
IntersectLocal
6826934 (3.83%)
mat
(*Sphere)
IntersectLocal
2267687 (1.27%)
of 24075371 (13.52%)
sort
Slice
0 of 2949220 (1.66%)
mat
TransformRay
0 of 81152116 (45.56%)
reflectlite
Swapper
2949220 (1.66%)
mat
MultiplyByTuple
85969117 (48.26%)
mat
Sub
30049014 (16.87%)
786445
1589277
3096636
1802279
376840
4767848
1376283
1409051
3260482
1232156
3522633
2676152
819217
1769508
116465651
16105798
9240773
1523742
52032028
6826934
2392146
589835
1015832
1490973
2670656
9804510
688140
3457095
6826934
47662362
14270861
557074
33489754
2267687
2949220
81152116
21807684
2949220
85969117
32B
80B
48B

ELIMINATE ALLOCATIONS

- Start pre-allocating memory whereever possible and re-use:
  - Vectors and matrices being used in intermediate calculations
  - Intersection lists (slices)
  - …
- Sometimes not trivial

RENDER CONTEXT PER WORKER

- Each ”render context” needs to have it’s own copy of world objects and pre-allocated lists and storage for recurring computations

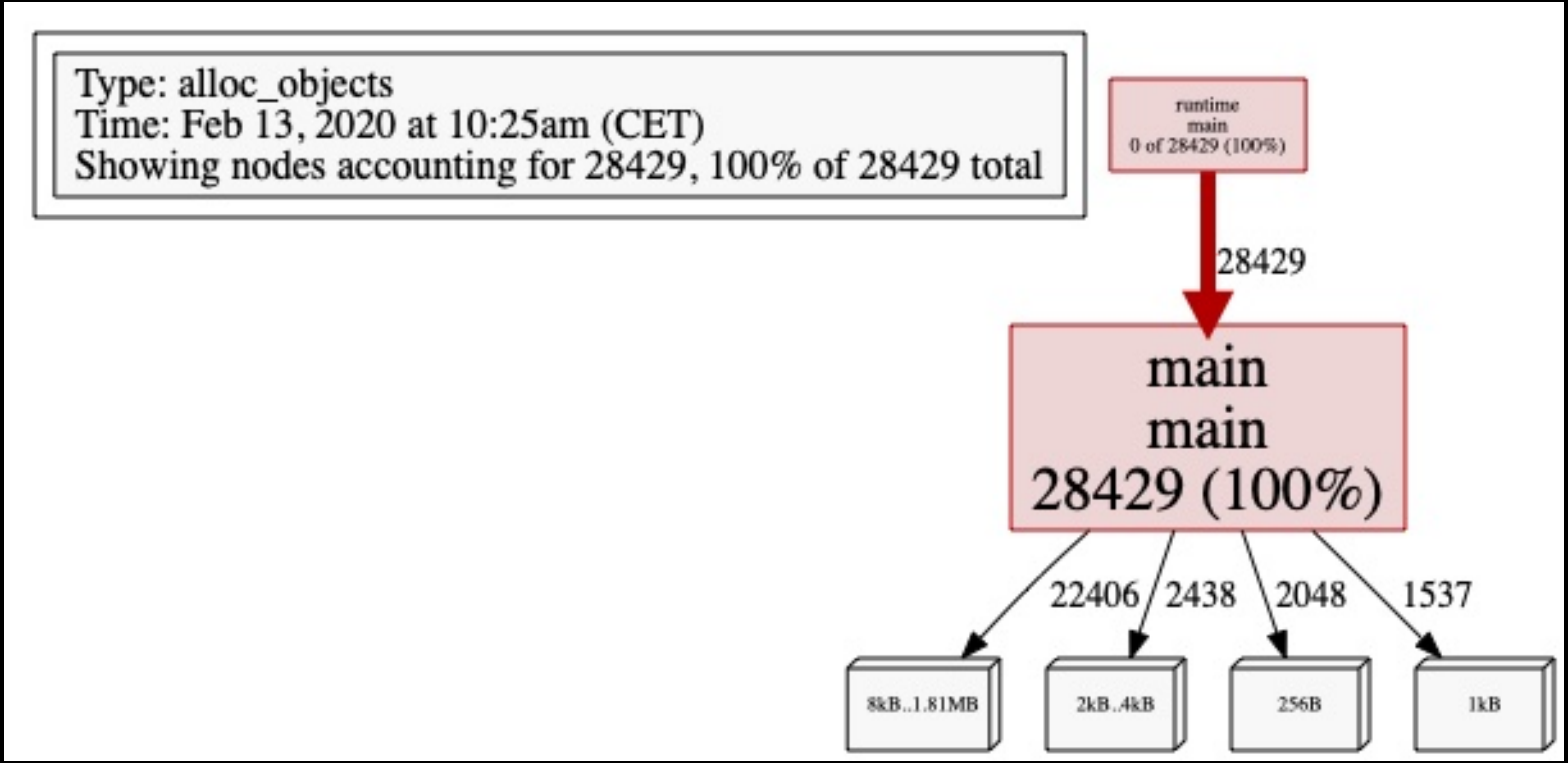
```
return Context{
    world: world,
    total: 0,

    // allocate memory
    pointInView: mat.NewPoint(0, 0, -1.0),
    pixel:       mat.NewColor(0, 0, 0),
    origin:      mat.NewPoint(0, 0, 0),
    direction:   mat.NewVector(0, 0, 0),
    subVec:      mat.NewVector(0, 0, 0),

    // allocate ray
    firstRay: mat.NewRay(mat.NewPoint(0, 0, 0),
                         mat.NewVector(0, 0, 0)),

    // stack for shading
    cStack: cStack,
}
```

RE-USE SLICE MEMORY

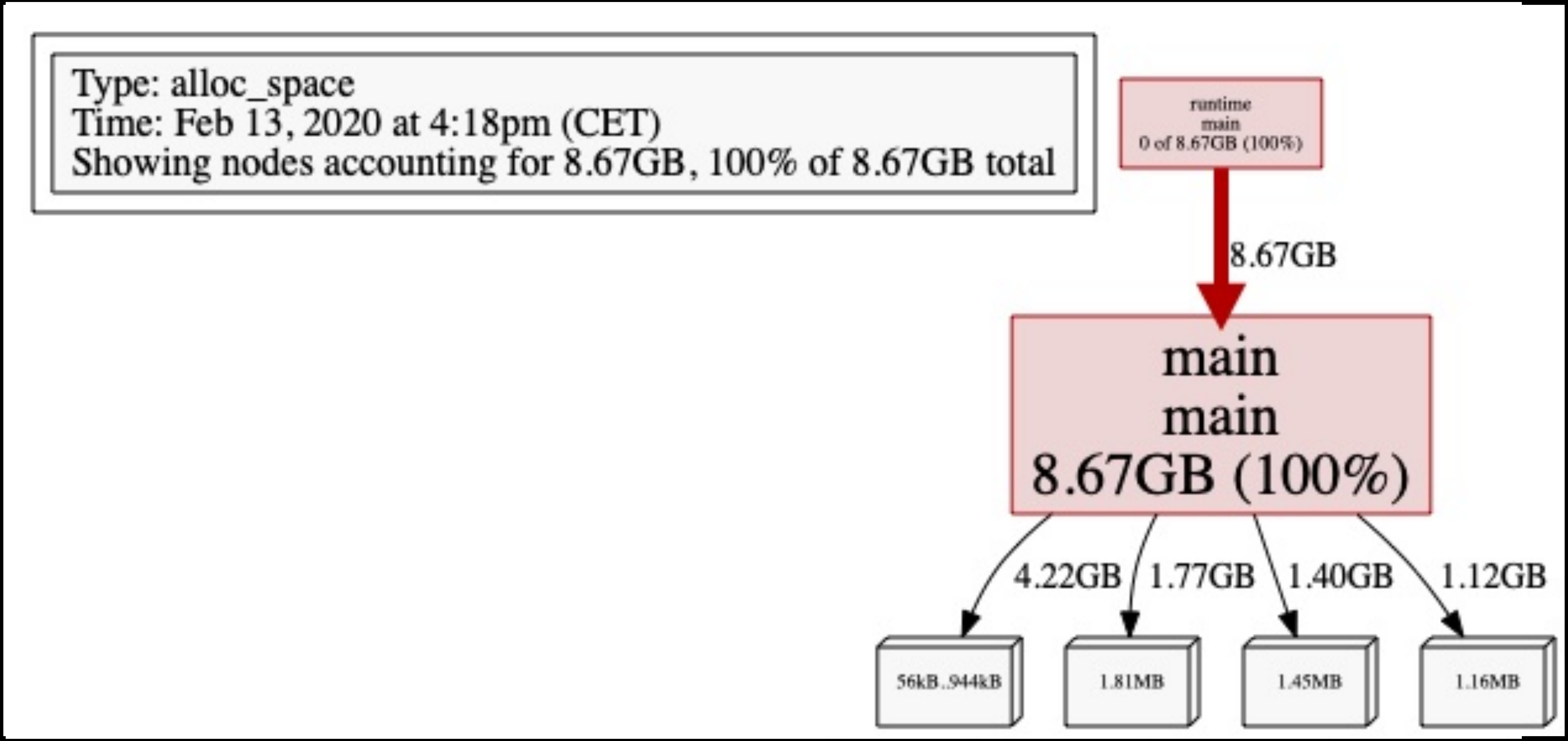
- Re-slice used slices rather than setting them to nil
- Preserves memory

```
bigslice := make([]int, 200000)
for i := 0; i < 1000; i++ {
    bigslice = nil
    for i := 0; i < 200000; i++ {
        bigslice = append(bigslice, rand.Intn(1000000))
    }
    if i % 1000 == 0 {
        fmt.Printf("Data len: %v\n", len(bigslice))
    }
}
```

RE-USE SLICE MEMORY



RE-USE SLICE MEMORY

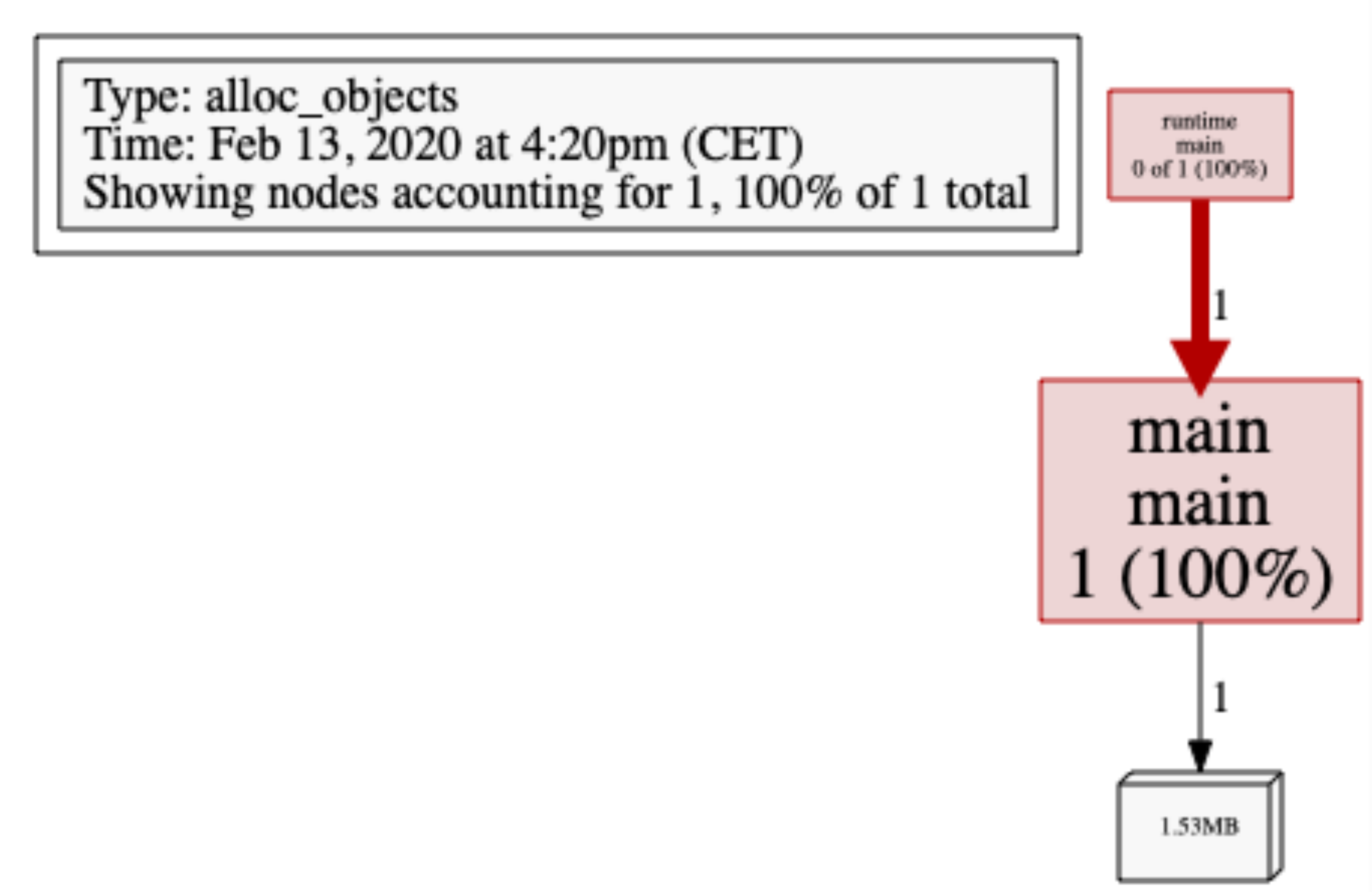


RE-USE SLICE MEMORY BY [:0]

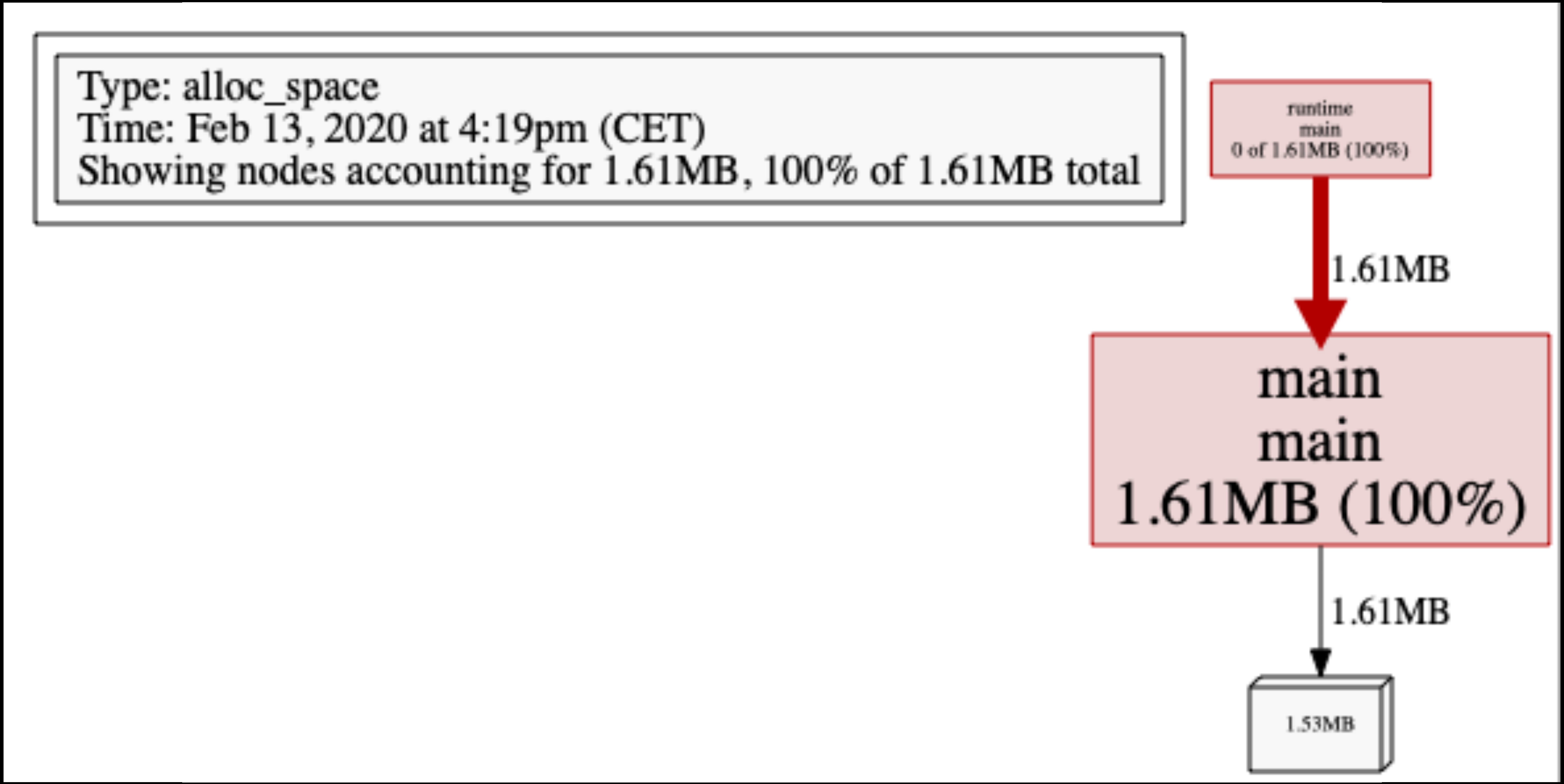
- reslice by slice = slice[:0]

```go
bigslice := make([]int, 200000)
for i := 0; i < 1000; i++ {
    bigslice = bigslice[:0]
    for i := 0; i < 200000; i++ {
        bigslice = append(bigslice, rand.Intn(1000000))
    }
    if i % 1000 == 0 {
        fmt.Printf("Data len: %v\n", len(bigslice))
    }
}
```

RE-SLICE MEMORY USING [:0]



RE-SLICE MEMORY USING [:0]



MORE EFFICIENT C-STYLE RETURNS

- Pre-allocate memory and pass function results through a parameter passed as pointer instead of allocating locally

```
return Context{ rixByTuple(m1 Matrix4x4, t Tuple4) Tuple4 { e4) {
    // omitted for brevity float64, 4))
    for row := 0; row < 4; row++ { , 0) * t.Get(0)) +
    // allocate memory (m1.Get(row, 0) * t.Get(0)) +
    pointInView: mat.NewPoint(0, 0, -1.0),
    pixel:       mat.NewColor(0, 0, 0),
    origin:      mat.NewPoint(0, 0, 0),

}   return t1
}

// Somewhere else
mat.MultiplyMatrixByTuple(rc.camera.Inverse, rc.pointInView, &rc.pixel)
mat.MultiplyMatrixByTuple(rc.camera.Inverse, originPoint, &rc.origin)
```

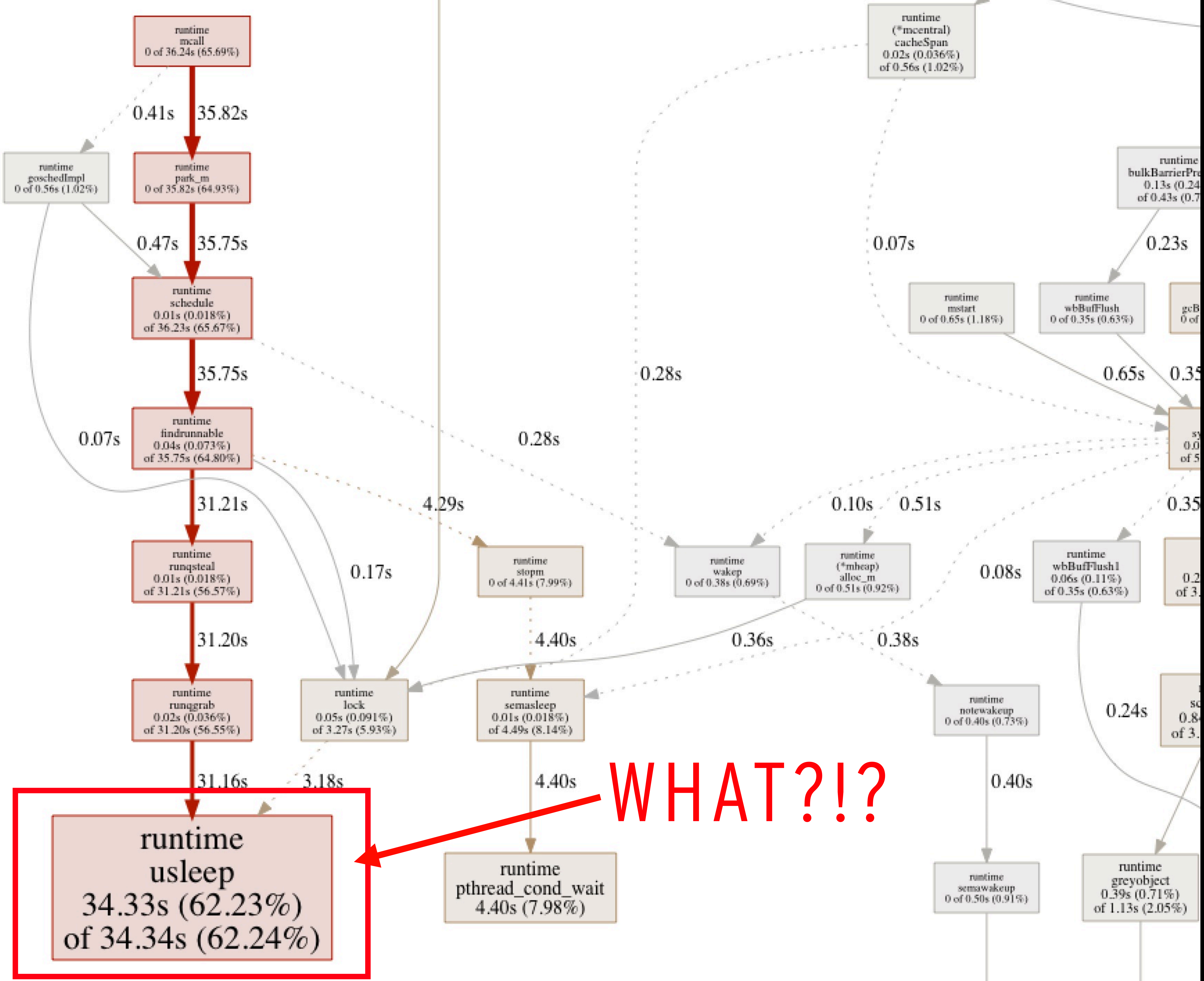
REFACTORING OUTCOME

- Continued refactoring and optimization resulted in:
  - Multi-threaded render: 4.2 -> 1.9 seconds
  - Allocation number: 180 million -> 33 million
    - » 12700 -> 100 allocs per pixel
  - Memory allocated: 5.9 GB -> 1.31 GB
- However…
  - More complex code base
  - Multi-threading requires careful access to shared data or context-exclusive copies

ONE MORE OPTIMIZATION…

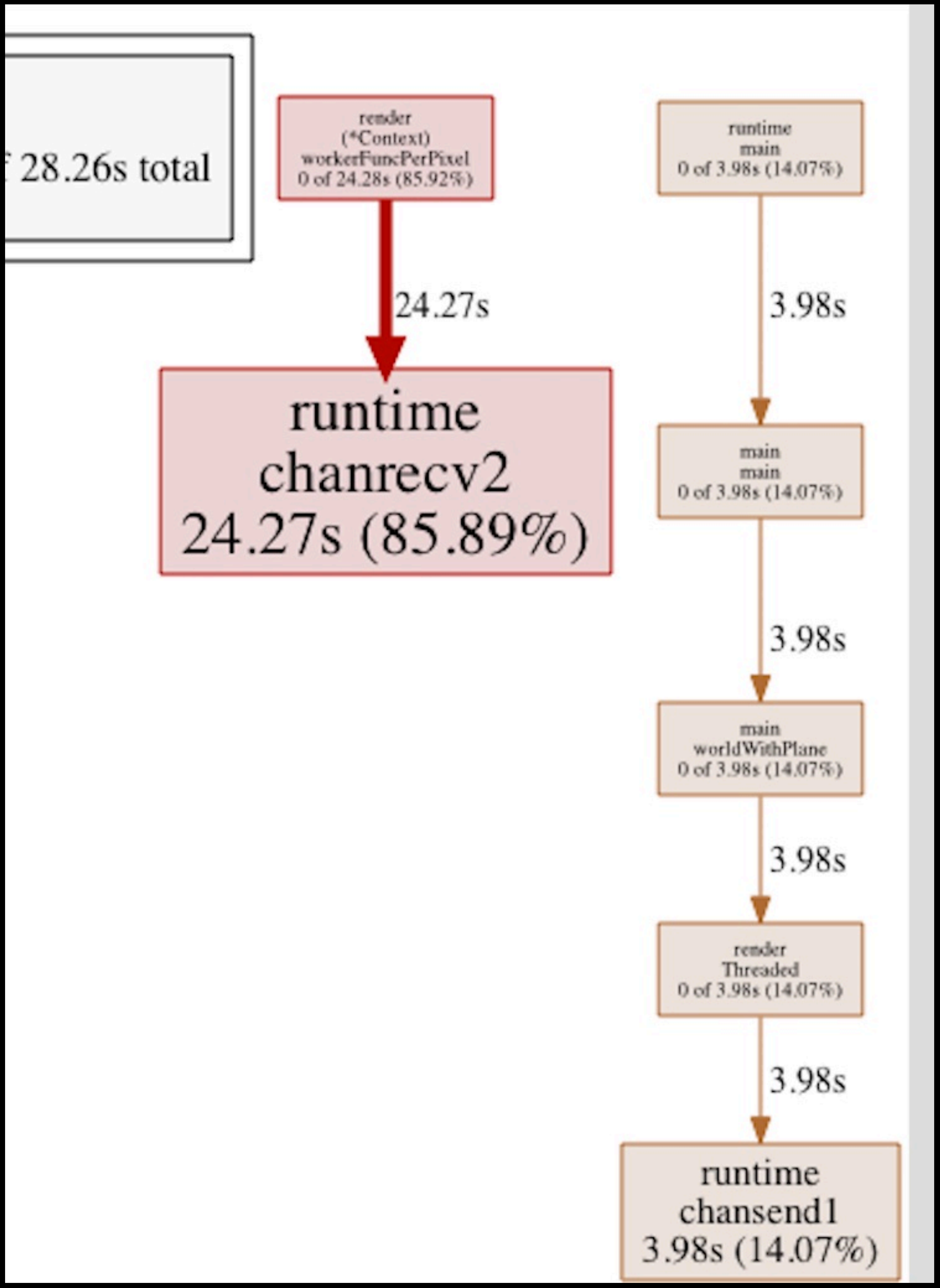
STEP 4 - LAST OPTIMIZATION

- After adding caching and reducing allocations significantly performance was quite good.
- Time to CPU profile again…


runtime mcall 0 of 36.24s (65.69%)
0.41s
35.82s
runtime goschedImpl 0 of 0.56s (1.02%)
runtime park_m 0 of 35.82s (64.93%)
0.47s
35.75s
runtime schedule 0.01s (0.018%) of 36.23s (65.67%)
35.75s
runtime findrunnable 0.04s (0.073%) of 35.75s (64.80%)
0.07s
0.28s
4.29s
0.17s
31.21s
runtime runqsteal 0.01s (0.018%) of 31.21s (56.57%)
31.20s
runtime runqgrab 0.02s (0.036%) of 31.20s (56.55%)
runtime lock 0.05s (0.091%) of 3.27s (5.93%)
runtime stopm 0 of 4.41s (7.99%)
4.40s
runtime semasleep 0.01s (0.018%) of 4.49s (8.14%)
31.16s
3.18s
4.40s
runtime usleep 34.33s (62.23%) of 34.34s (62.24%)
runtime pthread_cond_wait 4.40s (7.98%)
WHAT?!?
runtime (*mcentral) cacheSpan 0.02s (0.036%) of 0.56s (1.02%)
0.07s
0.28s
runtime wakep 0 of 0.38s (0.69%)
runtime (*mheap) alloc_m 0 of 0.51s (0.92%)
0.10s
0.51s
0.36s
0.38s
0.08s
runtime notewakeup 0 of 0.40s (0.73%)
0.40s
runtime semawakeup 0 of 0.50s (0.91%)
runtime mstart 0 of 0.65s (1.18%)
runtime wbBufFlush 0 of 0.35s (0.63%)
0.23s
0.65s
runtime wbBufFlush1 0.06s (0.11%) of 0.35s (0.63%)
0.24s
runtime greyobject 0.39s (0.71%) of 1.13s (2.05%)

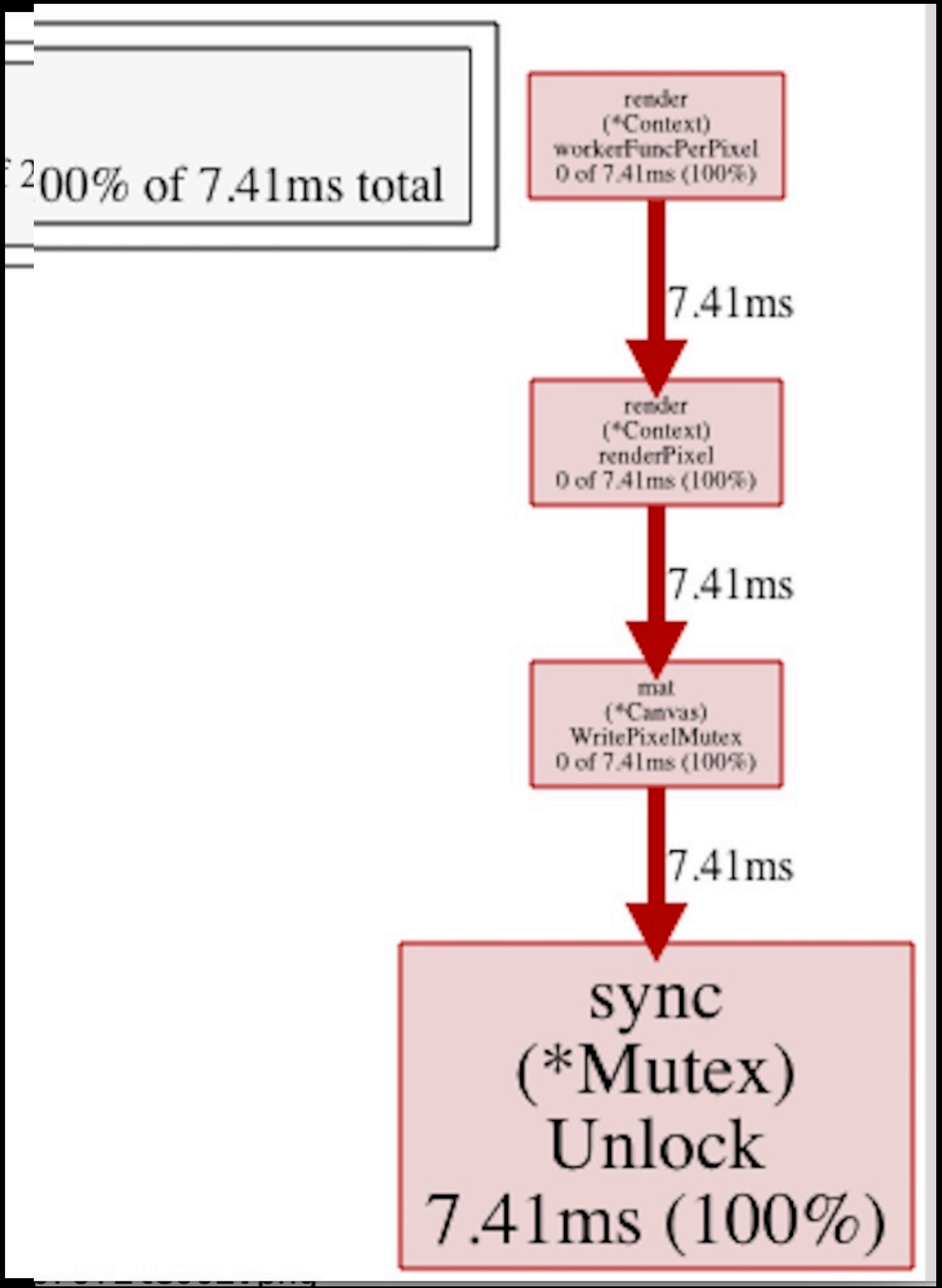
CONGESTION?

- Something slightly weird…
  - runtime usleep: 62.3% CPU time?!?!
    - » /debug/pprof/block



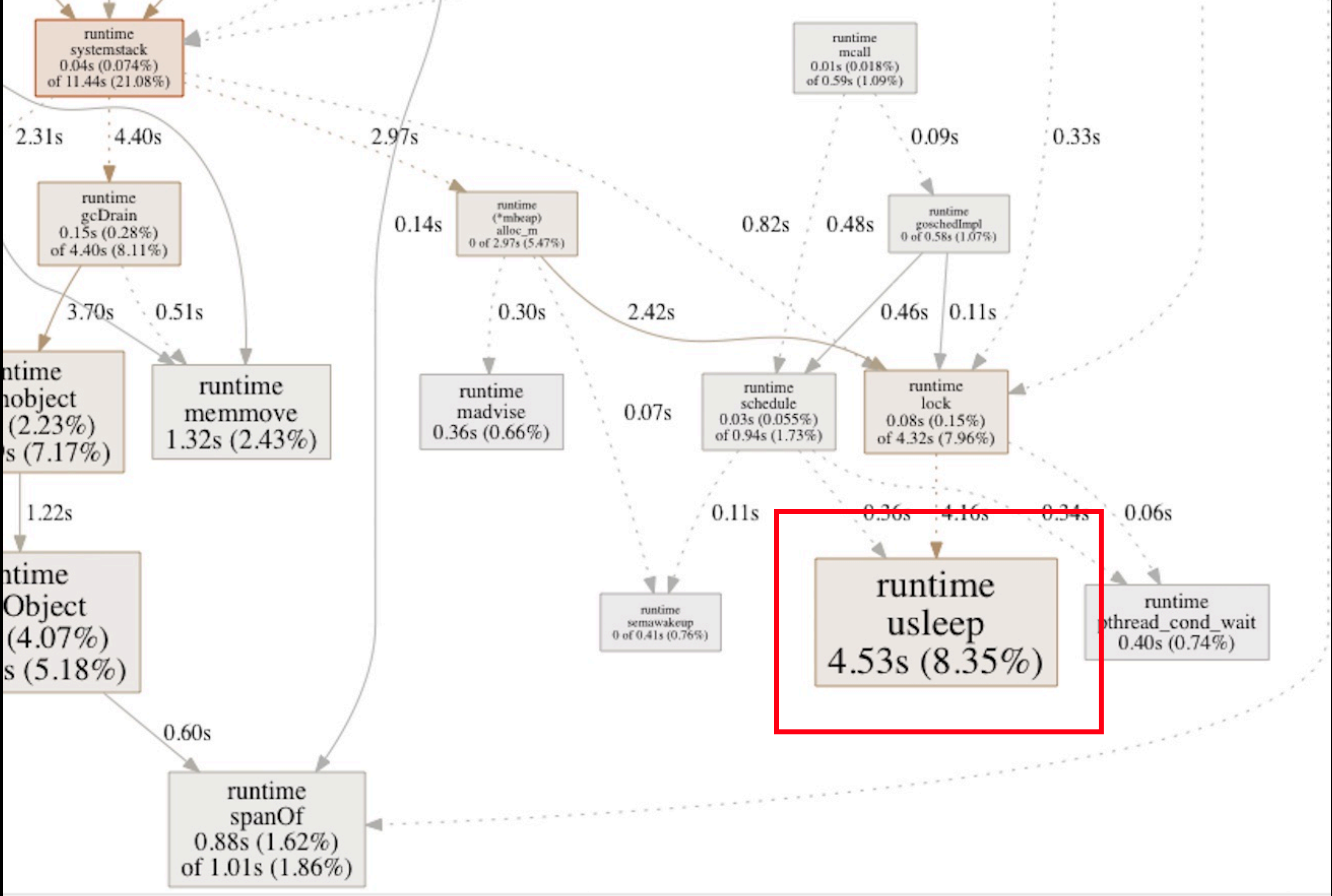
CONGESTION?

- Something slightly weird…
  - runtime usleep: 62.3% CPU time?!?!
    - » /debug/pprof/block
    - » /debug/pprof/mutex



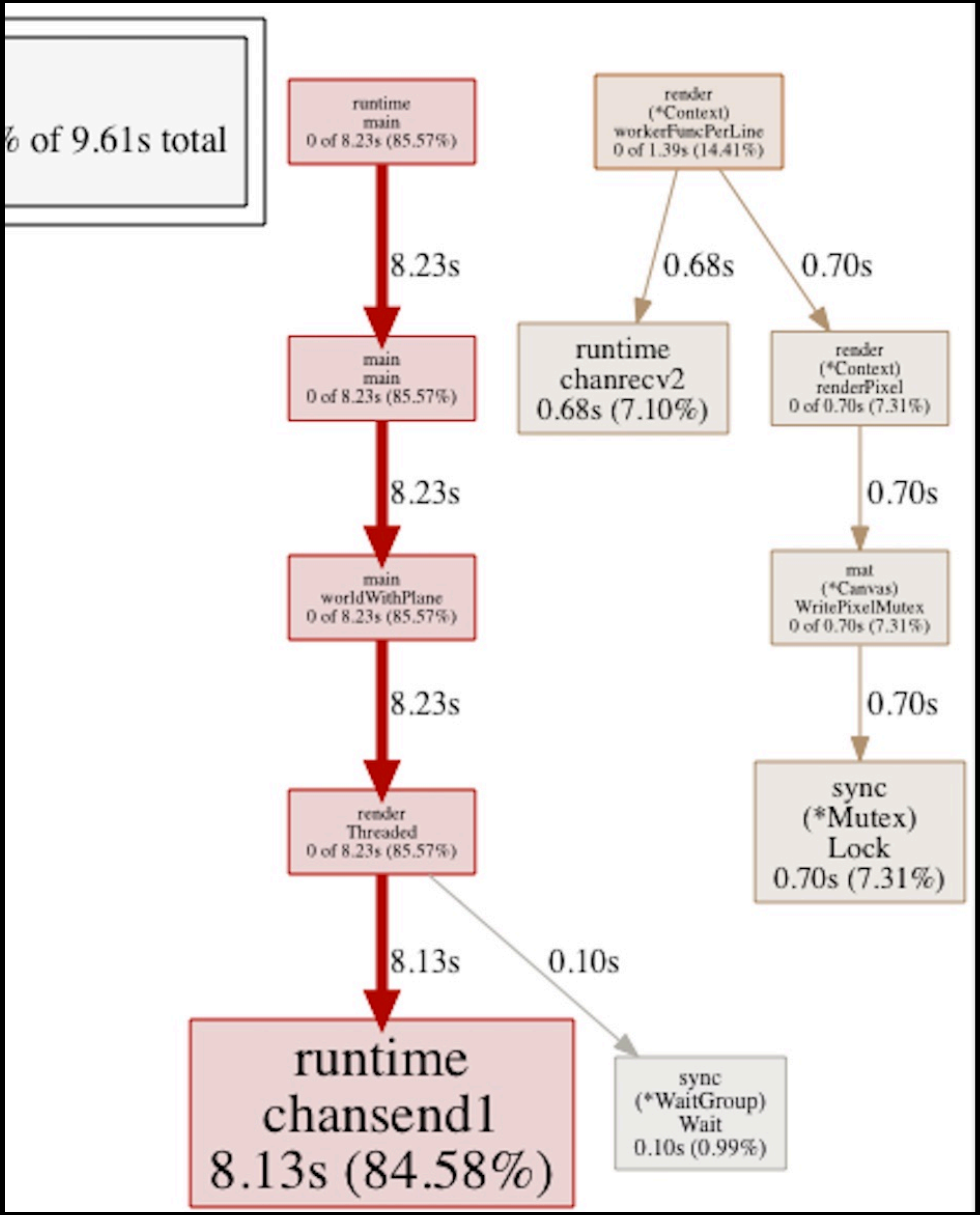
CONGESTION

- Renderer is based on the worker-pool pattern
  - 8 workers having their own rendering context / memory
- One job per pixel
  - 1920x1080 -> ~2 million jobs passed to either of the 8 workers through an unbuffered channel
  - 16.4 seconds
- One job per line
  - 1920x1080 -> 1080 jobs passed
  - 14.7 seconds



runtime
systemstack
0.04s (0.074%)
of 11.44s (21.08%)
runtime
mcall
0.01s (0.018%)
of 0.59s (1.09%)
2.31s
4.40s
2.97s
0.09s
0.33s
runtime
gcDrain
0.15s (0.28%)
of 4.40s (8.11%)
0.14s
runtime
(*mheap)
alloc_m
0 of 2.97s (5.47%)
0.82s
0.48s
runtime
goschedImpl
0 of 0.58s (1.07%)
3.70s
0.51s
0.30s
2.42s
0.46s
0.11s
runtime
memmove
1.32s (2.43%)
runtime
madvise
0.36s (0.66%)
0.07s
runtime
schedule
0.03s (0.055%)
of 0.94s (1.73%)
runtime
lock
0.08s (0.15%)
of 4.32s (7.96%)
1.22s
0.11s
0.36s
4.16s
0.34s
0.06s
runtime
semawakeup
0 of 0.41s (0.76%)
runtime
usleep
4.53s (8.35%)
runtime
pthread_cond_wait
0.40s (0.74%)
0.60s
runtime
spanOf
0.88s (1.62%)
of 1.01s (1.86%)

BLOCKS - PASS BY LINE



REFACTORING CONGESTION OUTCOME

- Multi-threaded render: 1.9 -> 1.6 seconds
- However, 8 threads vs 1 thread is still just 2.7x faster, so there's definitely a lot more bottlenecks to be found
  - slice.Sort is run on every intersection which allocates memory internally
  - A lot of basic vector / matrix ops still allocating memory
  - Experiment with GOGC to run GC less often
  - I'm considering a total rewrite ;)

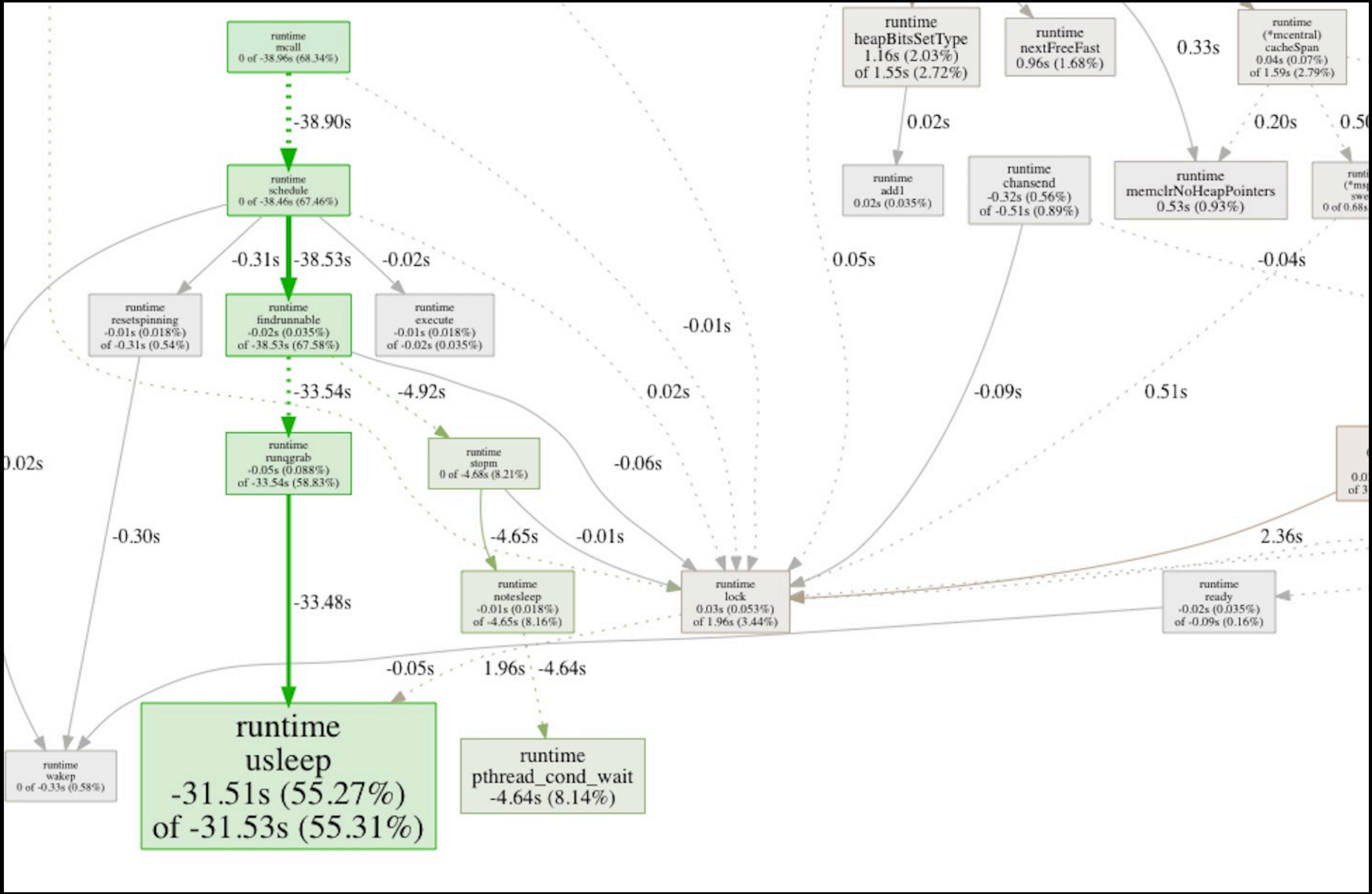
A FINAL TRICK -
LIVE CODING -
WITH DEMO!

COMPARING PPROFS

- Pprof supports loading two pprof files in order to compare them
- go tool pprof -diff_base pixel-render.pb.gz line-render.pb.gz

```
~/diffprofiles> go tool pprof -diff_base pixel-render.pb.gz line-render.pb.gz
Type: cpu
Time: Feb 13, 2020 at 9:44pm (CET)
Duration: 20.31s, Total samples = 57.01s (280.72%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for -23.71s, 41.59% of 57.01s total
Dropped 55 nodes (cum <= 0.29s)
Showing top 10 nodes out of 288
      flat  flat%   sum%        cum   cum%
   -31.51s 55.27% 55.27%    -31.53s 55.31%  runtime.usleep
    -4.64s  8.14% 63.41%     -4.64s  8.14%  runtime.pthread_cond_wait
     2.63s  4.61% 58.80%      9.28s 16.28%  runtime.mallocgc
     2.22s  3.89% 54.90%      4.05s  7.10%  github.com/eriklupander/rt/internal/pkg/mat.MultiplyByTuplePtr
     1.51s  2.65% 52.25%      1.72s  3.02%  runtime.findObject
     1.41s  2.47% 49.78%      1.41s  2.47%  github.com/eriklupander/rt/internal/pkg/mat.Tuple4.Get
     1.29s  2.26% 47.52%      1.29s  2.26%  github.com/eriklupander/rt/internal/pkg/mat.Mat4x4.Get
     1.16s  2.03% 45.48%      1.55s  2.72%  runtime.heapBitsSetType
     1.15s  2.02% 43.47%      3.68s  6.46%  github.com/eriklupander/rt/internal/pkg/mat.(*Sphere).IntersectLocal
     1.07s  1.88% 41.59%      1.32s  2.32%  github.com/eriklupander/rt/internal/pkg/mat.Dot
```

COMPARING PPROFS



SUMMARY

LESSONS LEARNED

- Even though Go is garbage collected, you need to think on how and when you're allocating memory if the code you're writing is performance critical.
  - That said:
    - Avoid premature optimization!!
- Always try to allocate memory that does not escape to the heap
  - *go build -gcflags '-m' <path>* to perform escape analysis (another topic…)
- Goroutines and channels are cheap…
  - But not free!
- … and so on …

SUMMARY

- Writing a ray-tracer is great fun
  - (with a good book to hold your hand while doing it!)
- Optimizing it was maybe **even** more fun!
  - (since I got to dive into go profiling in greater depth than ever before)
- Renderer could use a whole bunch of improvements
  - Or just use Blender… ;)
- The journey is the reward!

SUMMARY

- Do not give in to premature optimization!!!
  - (unless <insert reason>)
- go pprof is an invaluable tool for profiling running go code without having to manually log/measure/summarize or polluting the code base
  - With a quite low (1-3%) performance hit, some people even run it on their production servers!
- Powerful, but can be quite difficult to decipher the semantics of the output
  - I personally prefer the viz graphs

THANKS!