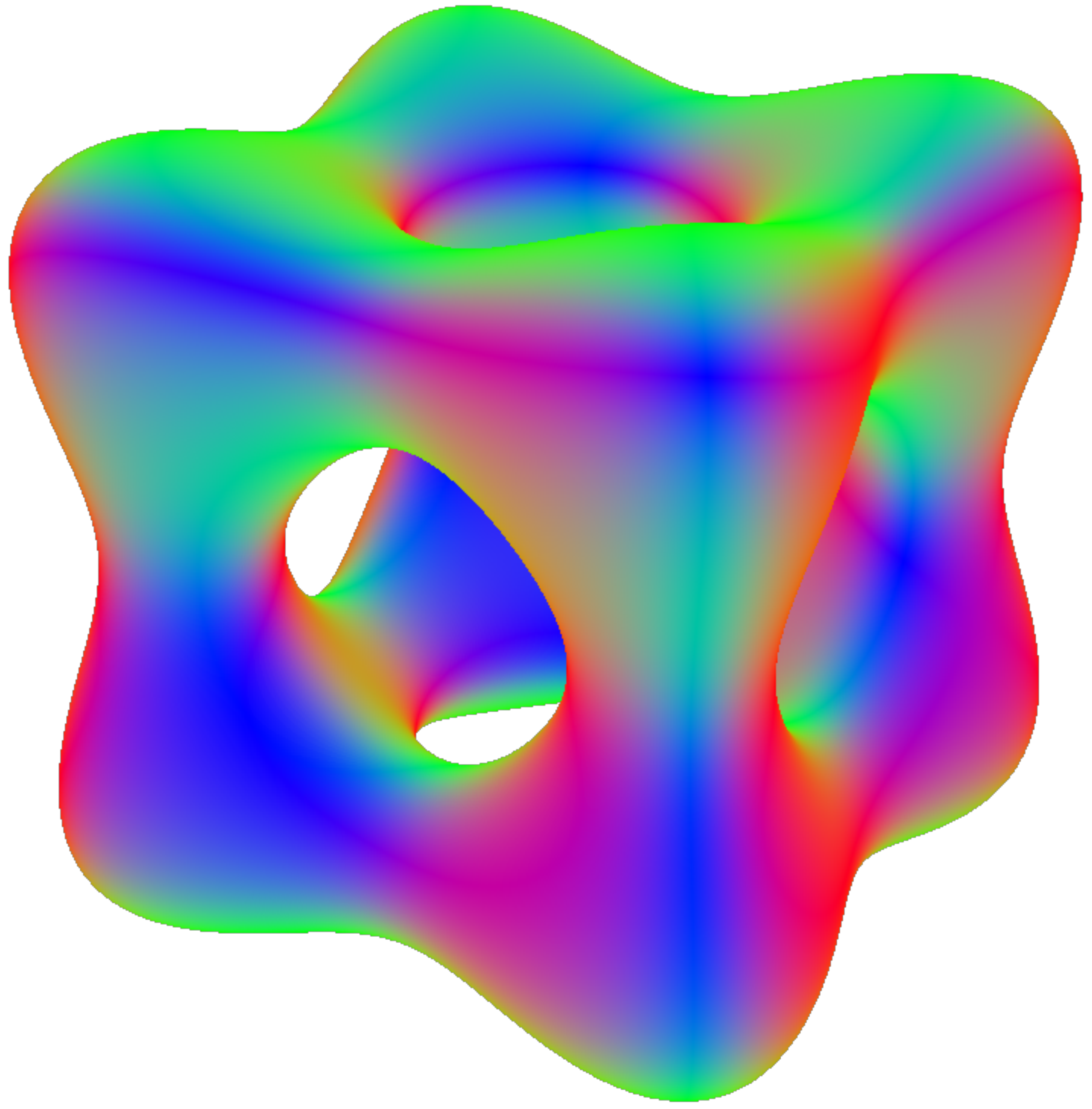
Implicit Surfaces & Independent Research

Matt Keeter
March 7, 2025



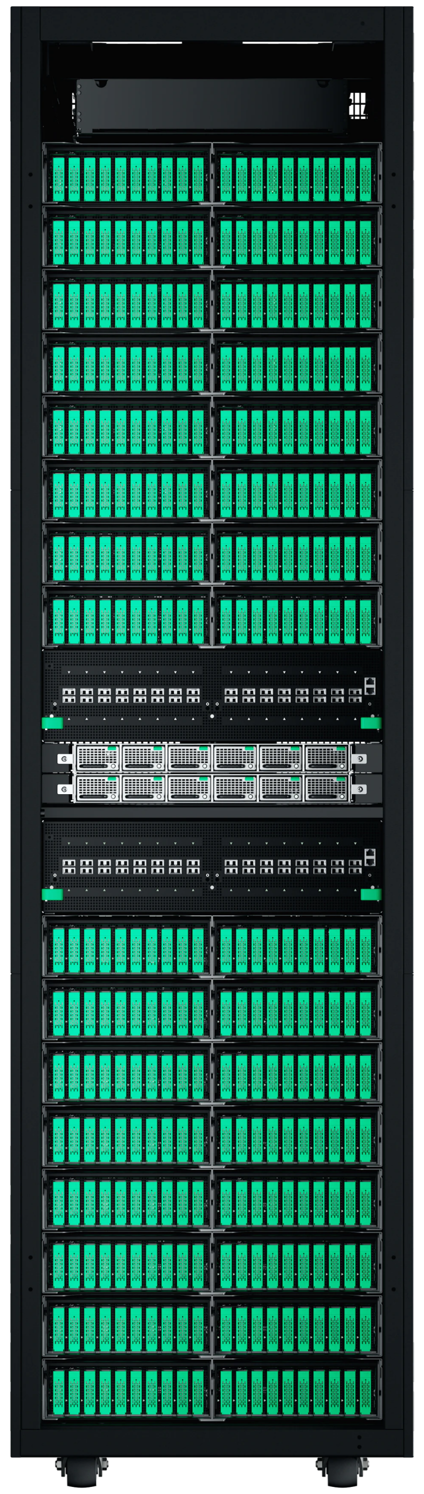
A brief timeline

2025

A brief timeline

2021

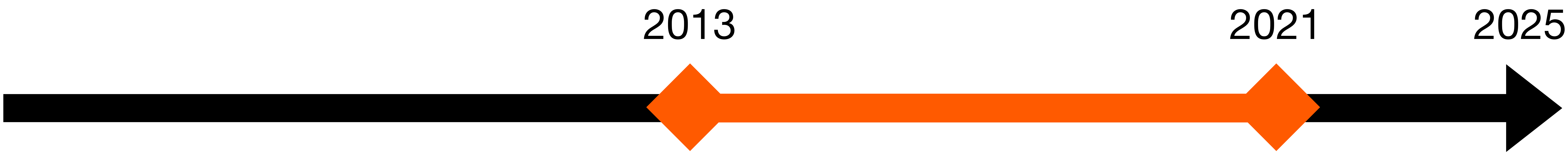
2025



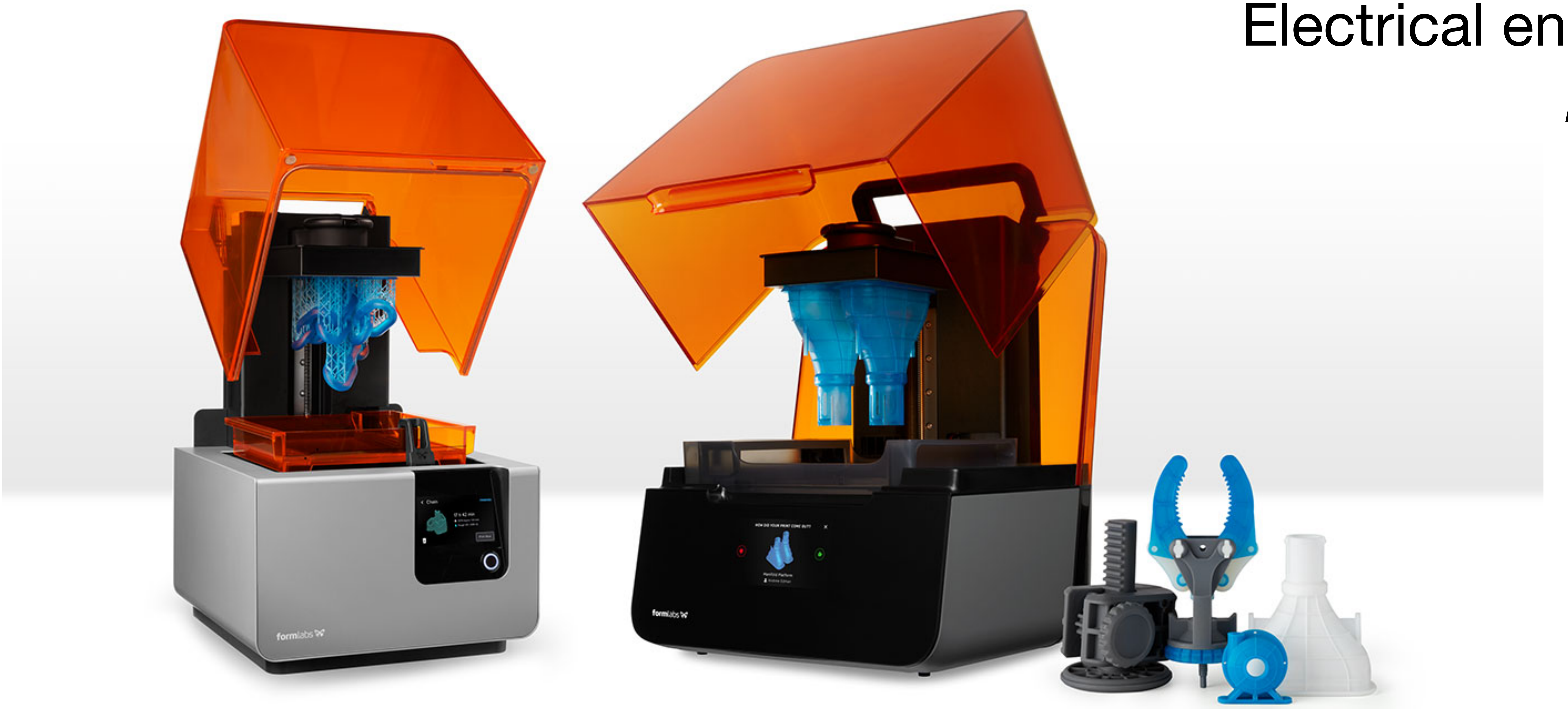
Embedded & systems software

Oxide Computer Company

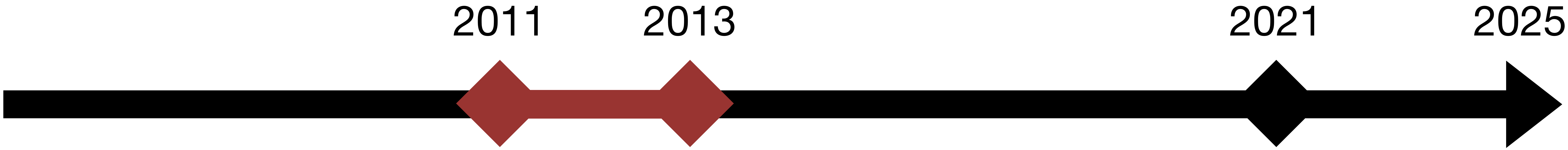
A brief timeline



Electrical engineering
Formlabs

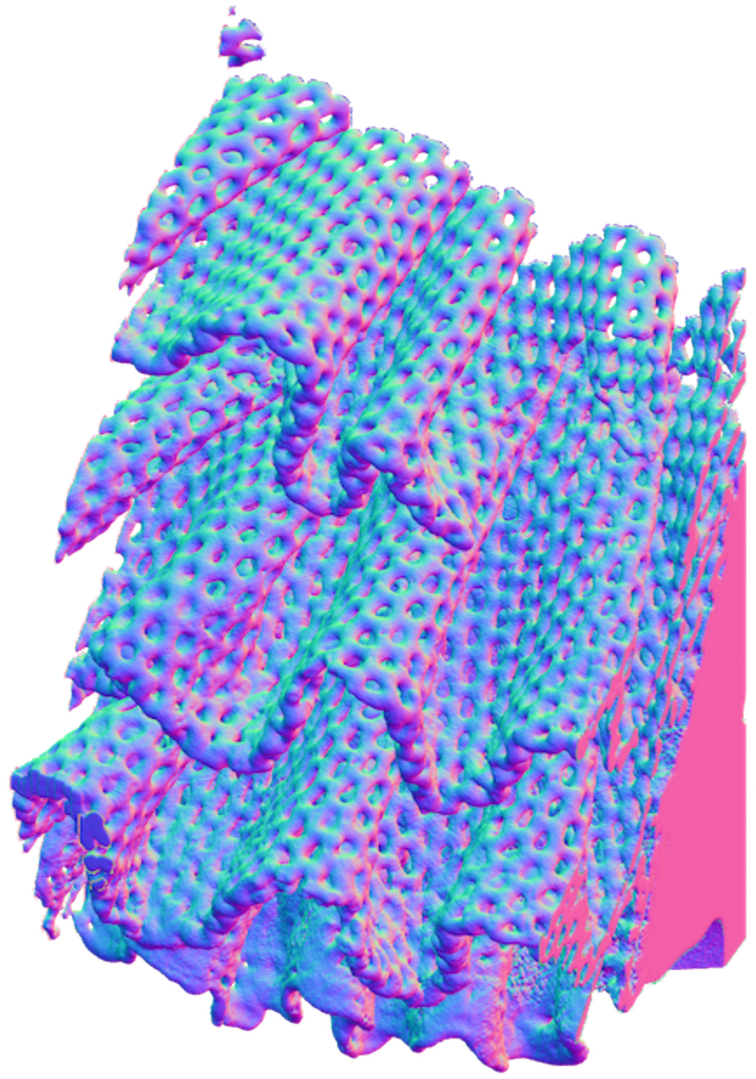


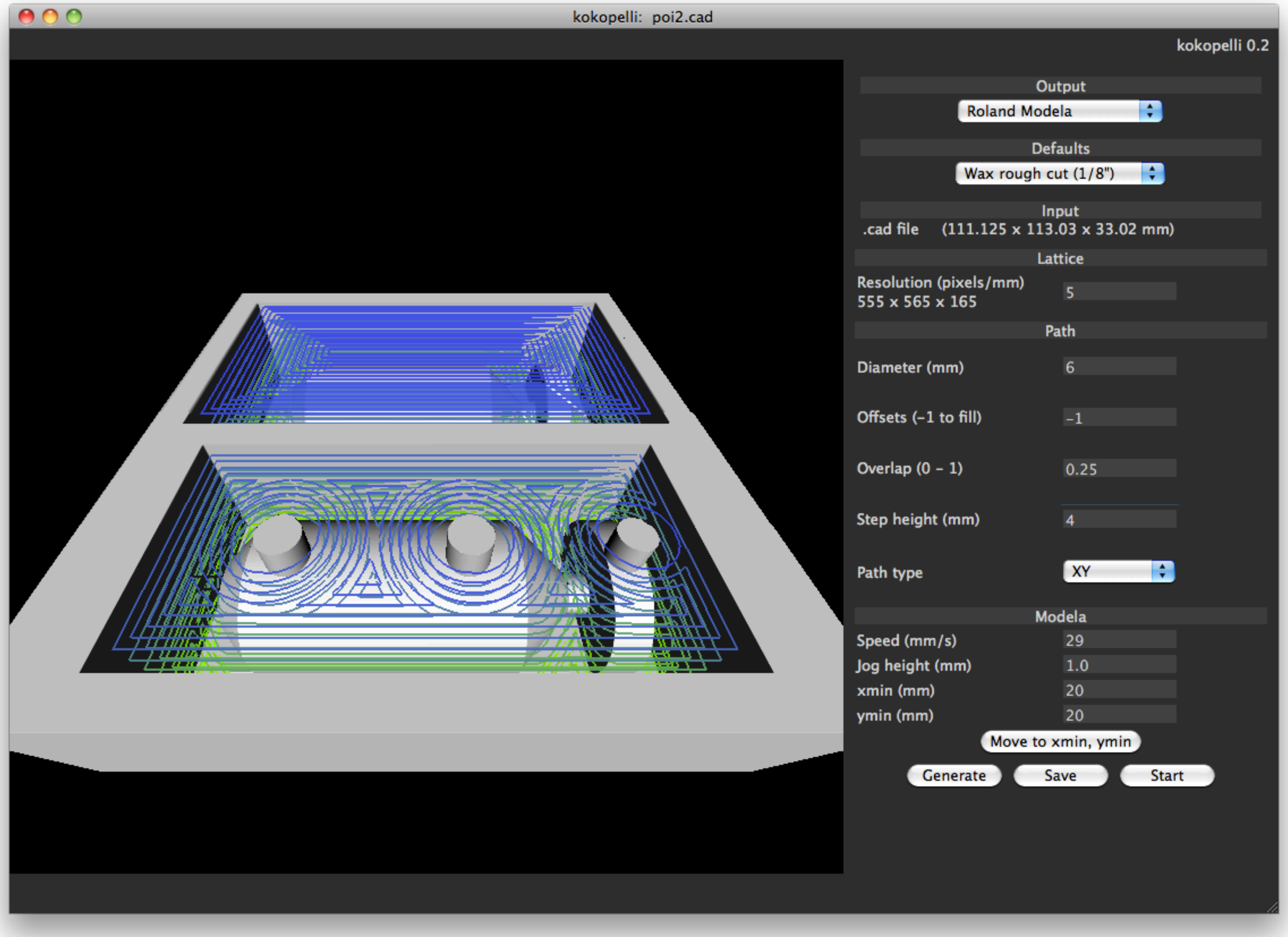
A brief timeline



Grad school

MIT Center for Bits & Atoms



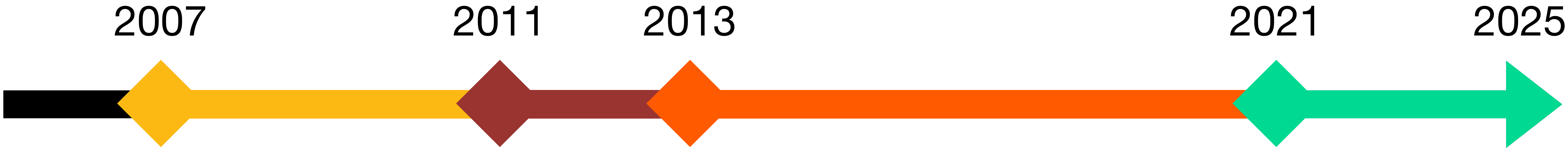


A brief timeline

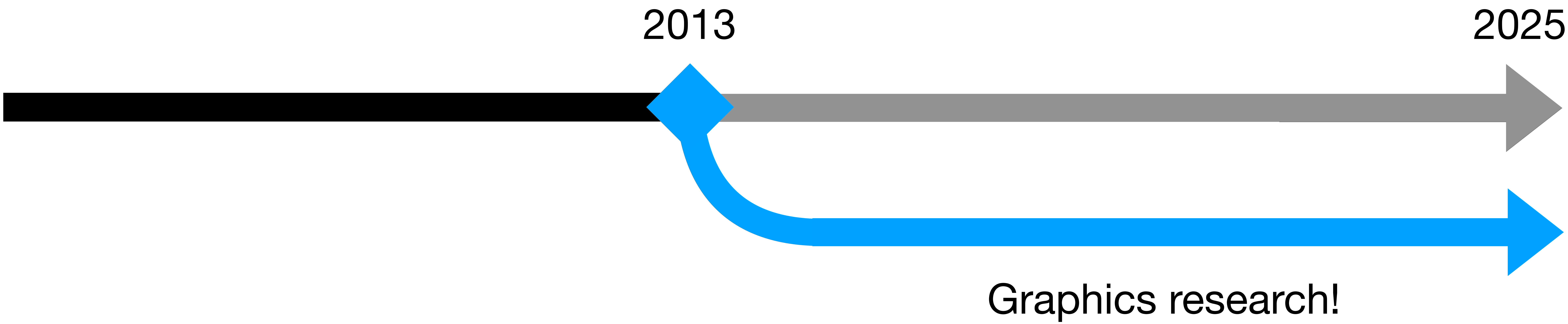
2007 — 2011 — 2013 — 2021 — 2025

Undergrad
Harvey Mudd College

A brief timeline

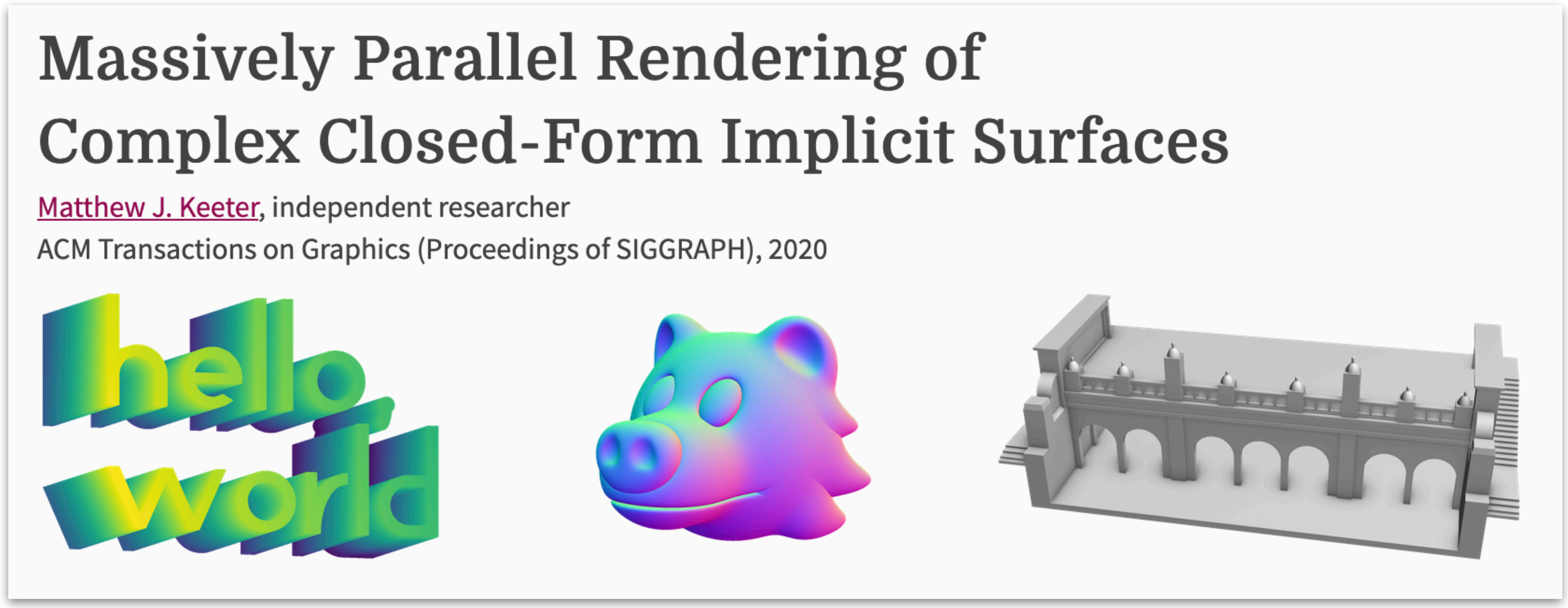


A brief timeline



libfive

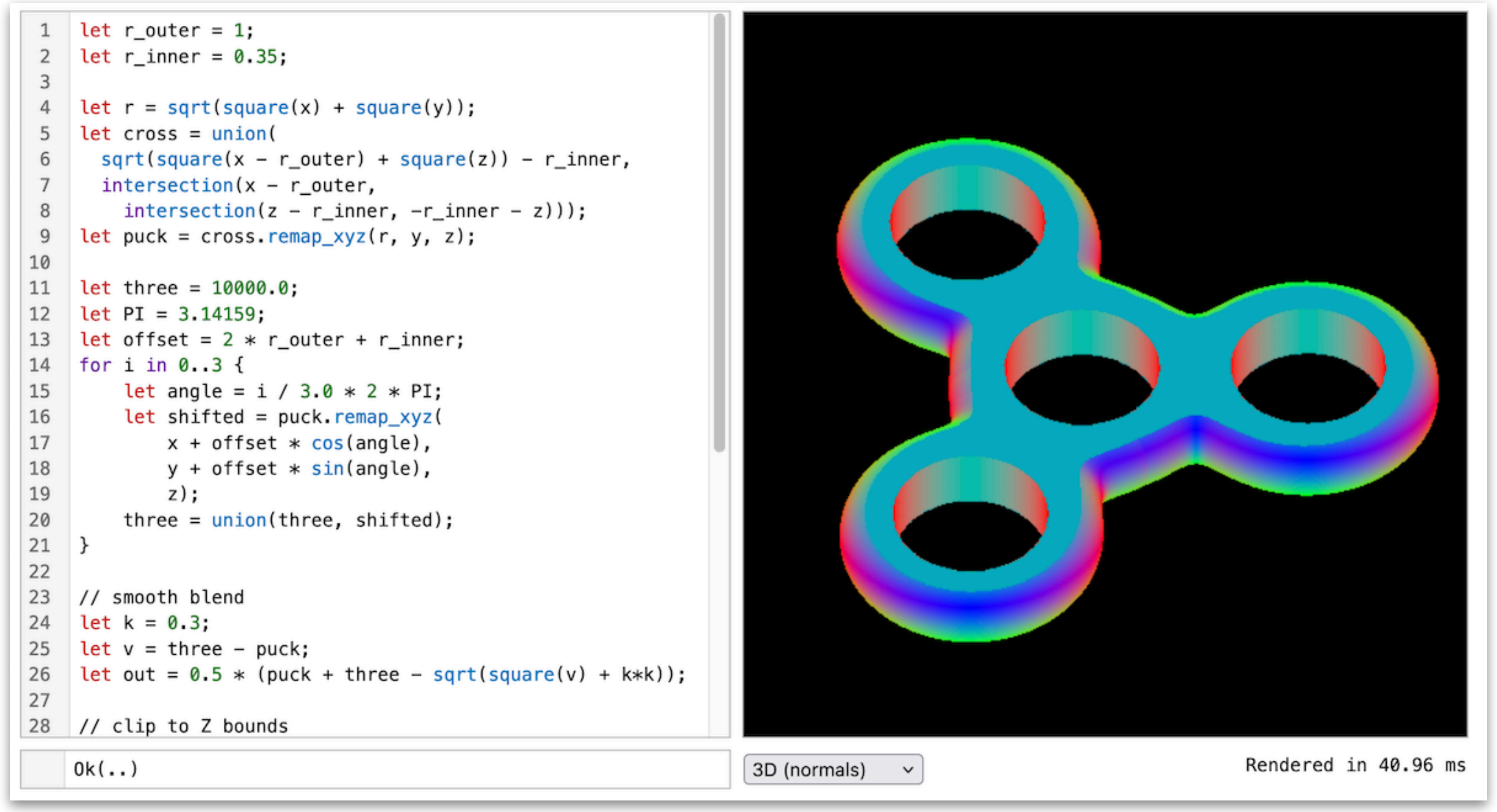
Studio



Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces

Matthew J. Keeter, independent researcher

ACM Transactions on Graphics (Proceedings of SIGGRAPH), 2020



What’s so interesting about implicit surfaces?

What does independent research look like?

What are implicit surfaces?

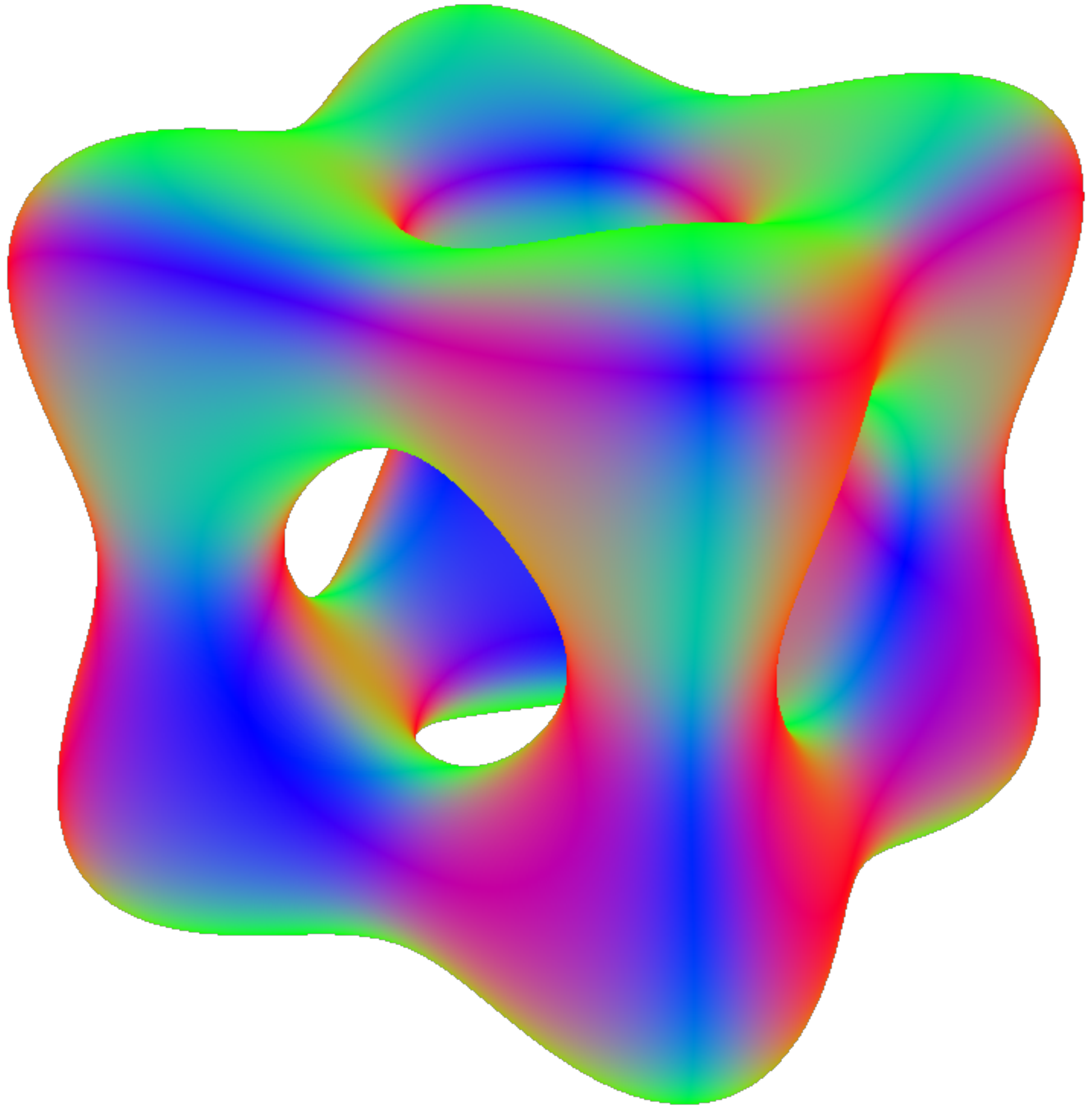
$$f(x, y, z) \to \mathbb{R}$$

$f(x, y, z) < 0$ inside the shape

$f(x, y, z) > 0$ outside the shape

$f(x, y, z) = 0$ at the surface

$$f(x, y, z) = x^4 - 5x^2 + y^4 - 5y^2 + z^4 - 5z^2 + 10$$

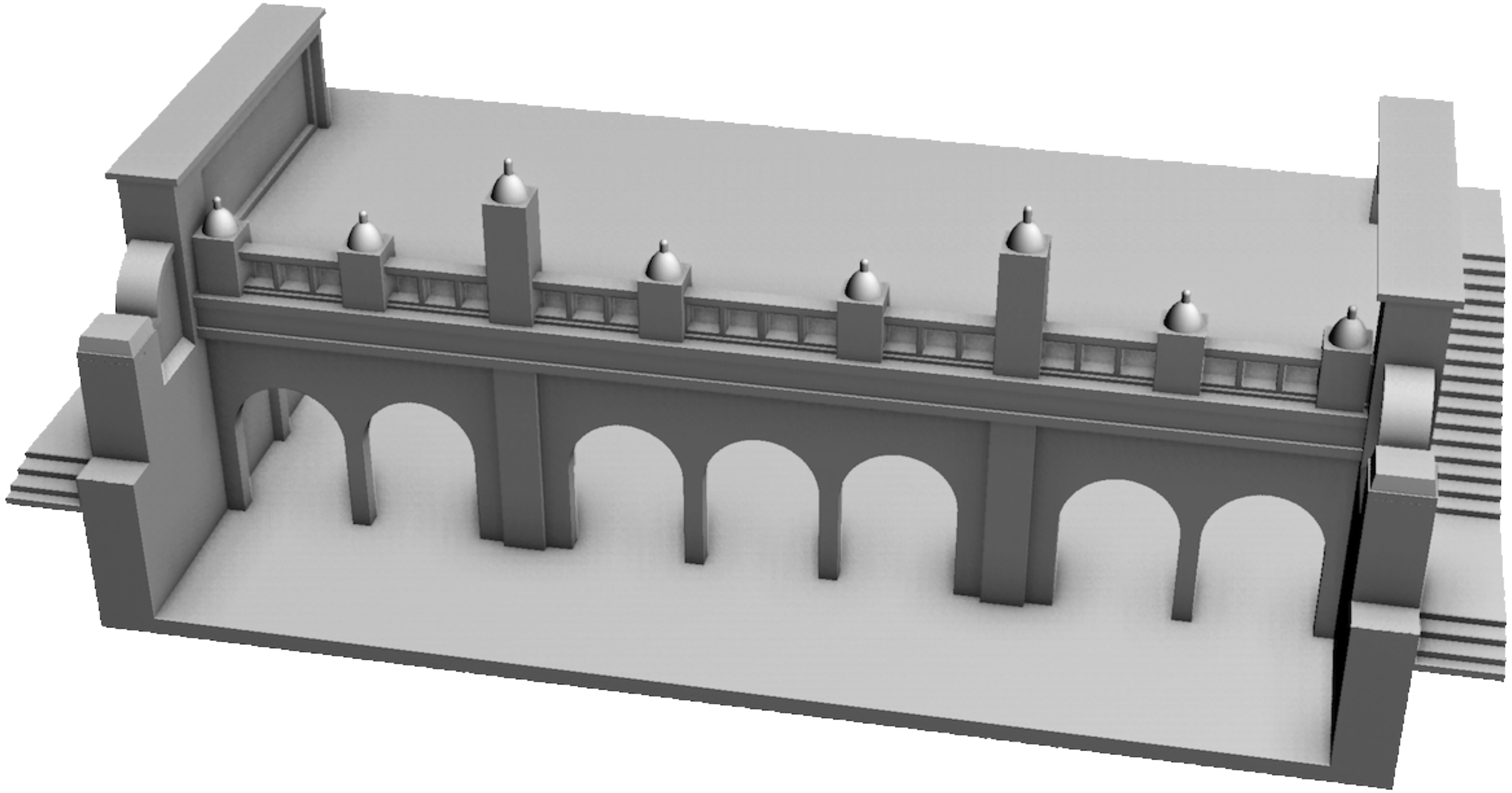


Complex closed-form implicit surfaces

Complex
closed-form
implicit surfaces

Complex
closed-form
implicit surfaces

Compact representation
…with arbitrary resolution

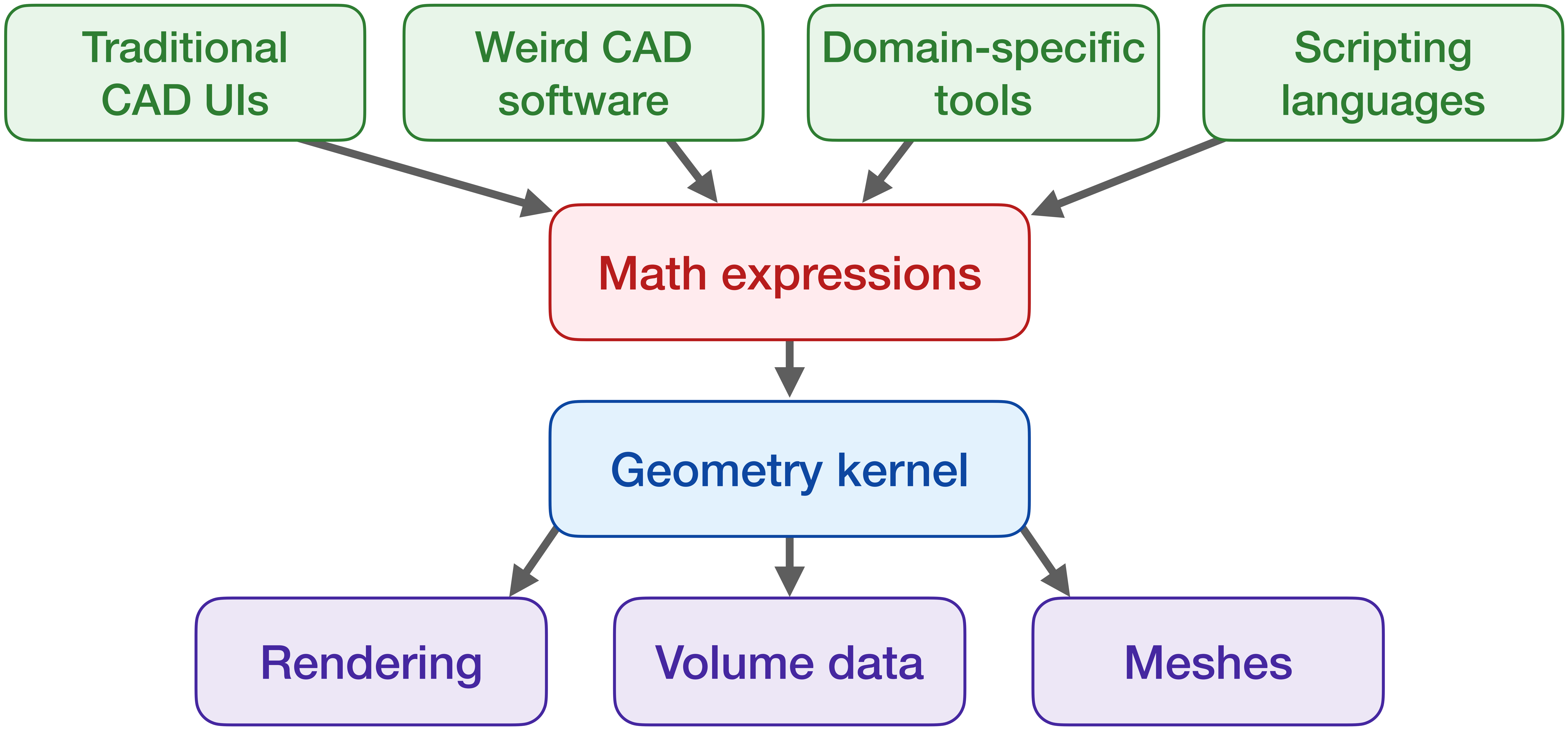


680 math operations

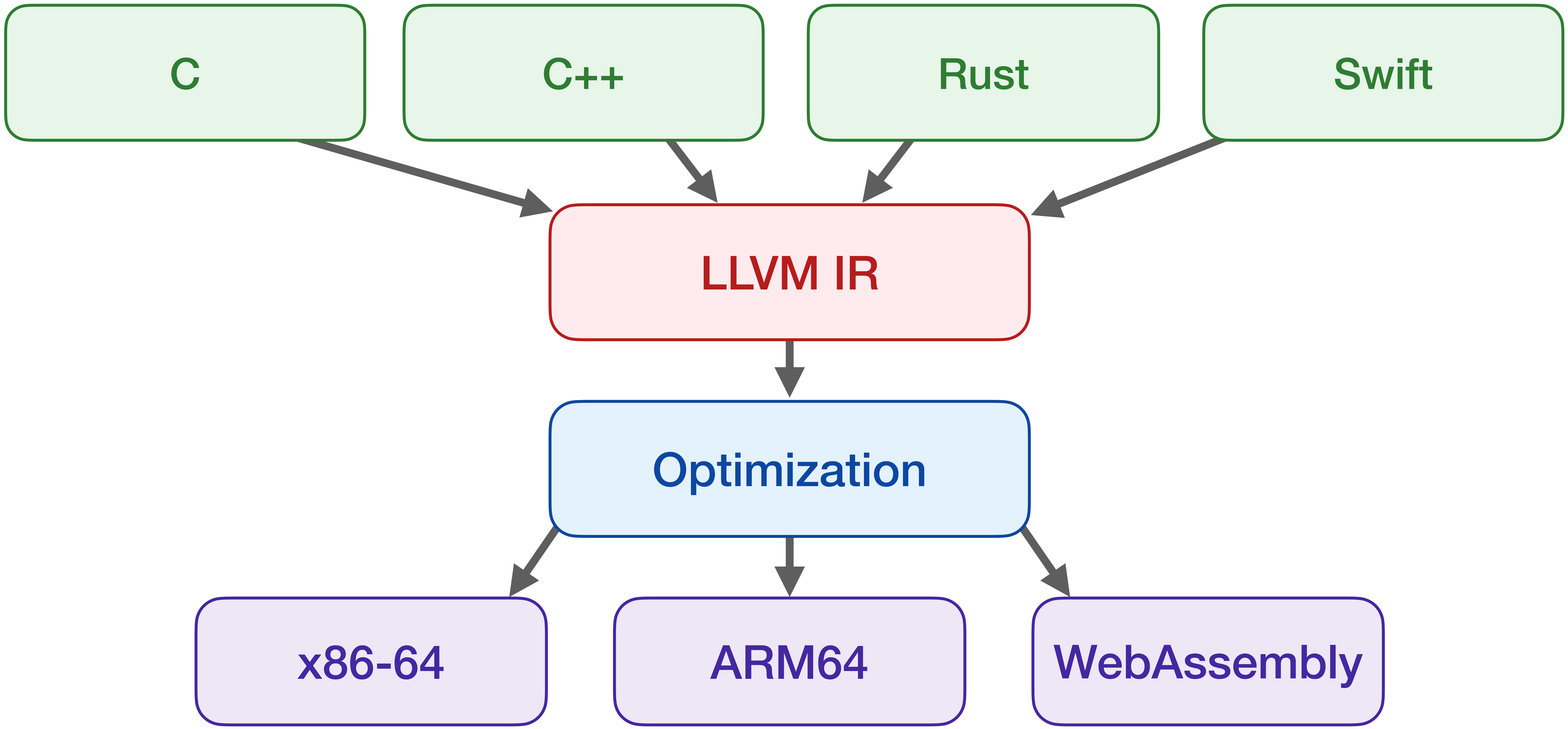
3612 bytes

(this image is 250633 bytes, 69× larger)

“Assembly language for shapes”

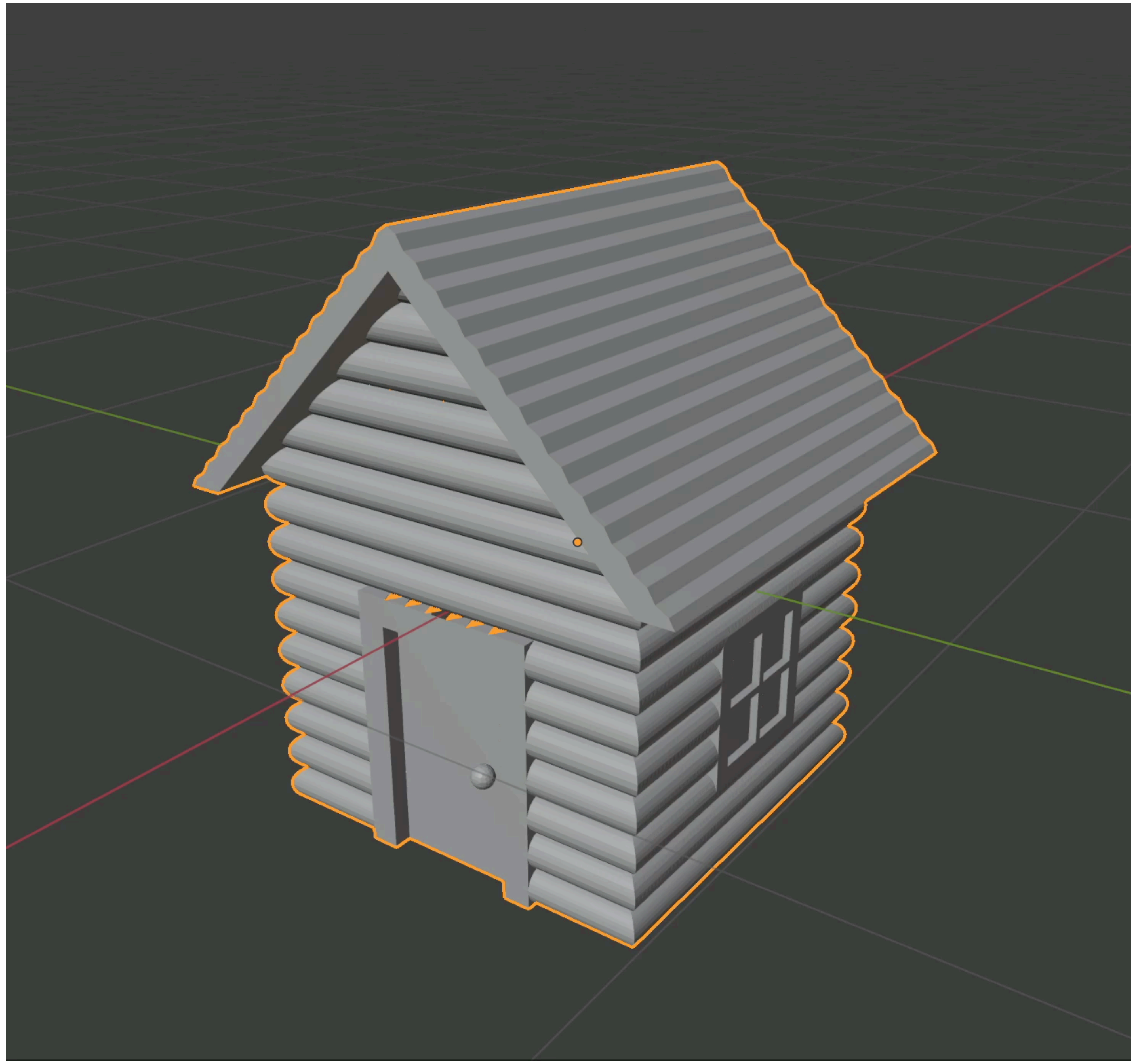


By analogy to LLVM IR



Solid modeling and CSG

(that's Constructive Solid Geometry)



Solid modeling and CSG

(that's Constructive Solid Geometry)

A microcosm of CS topics

Graphics programming

Data structures

Algorithms

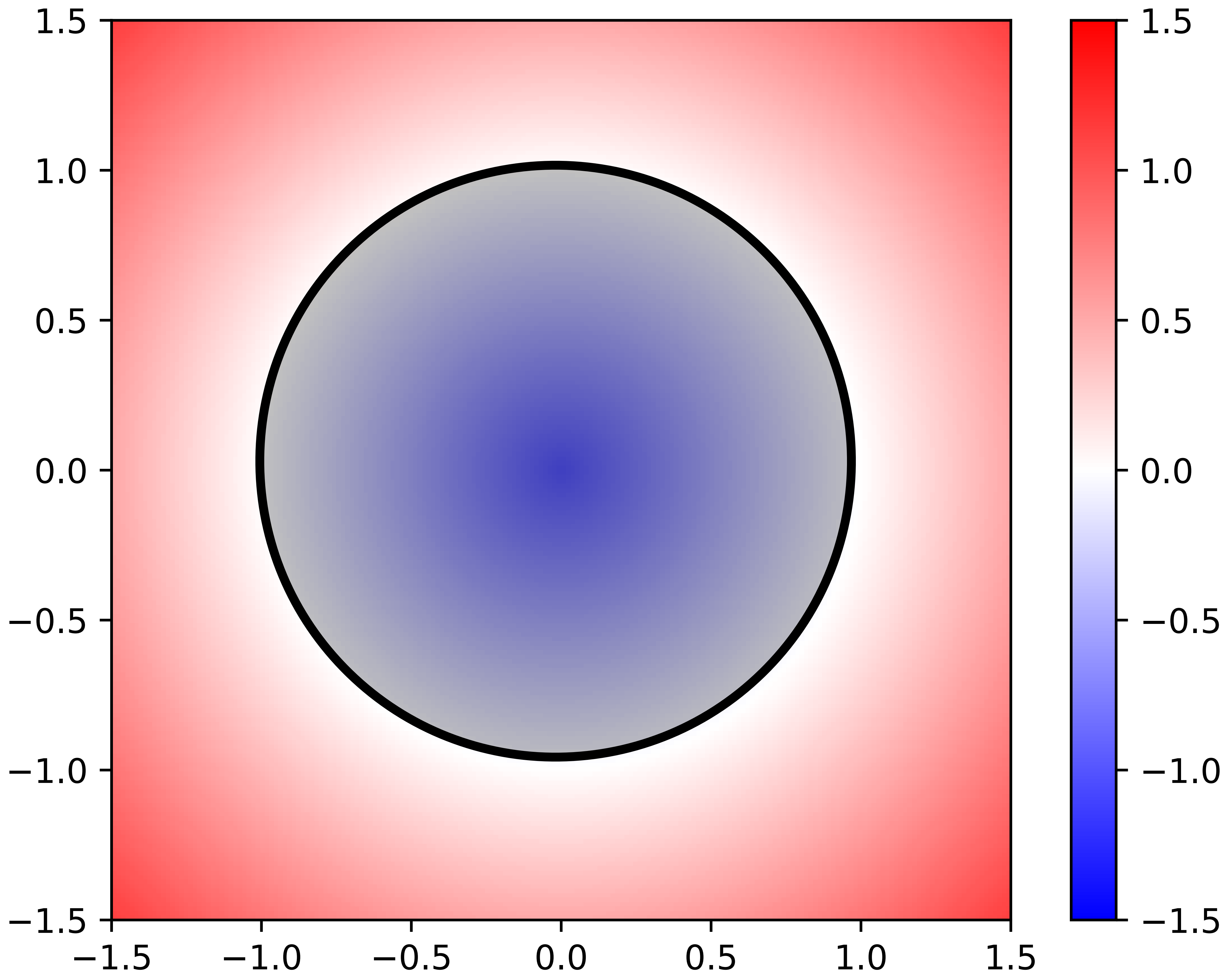
Numerical programming

Compilers

GPU programming

A simple example

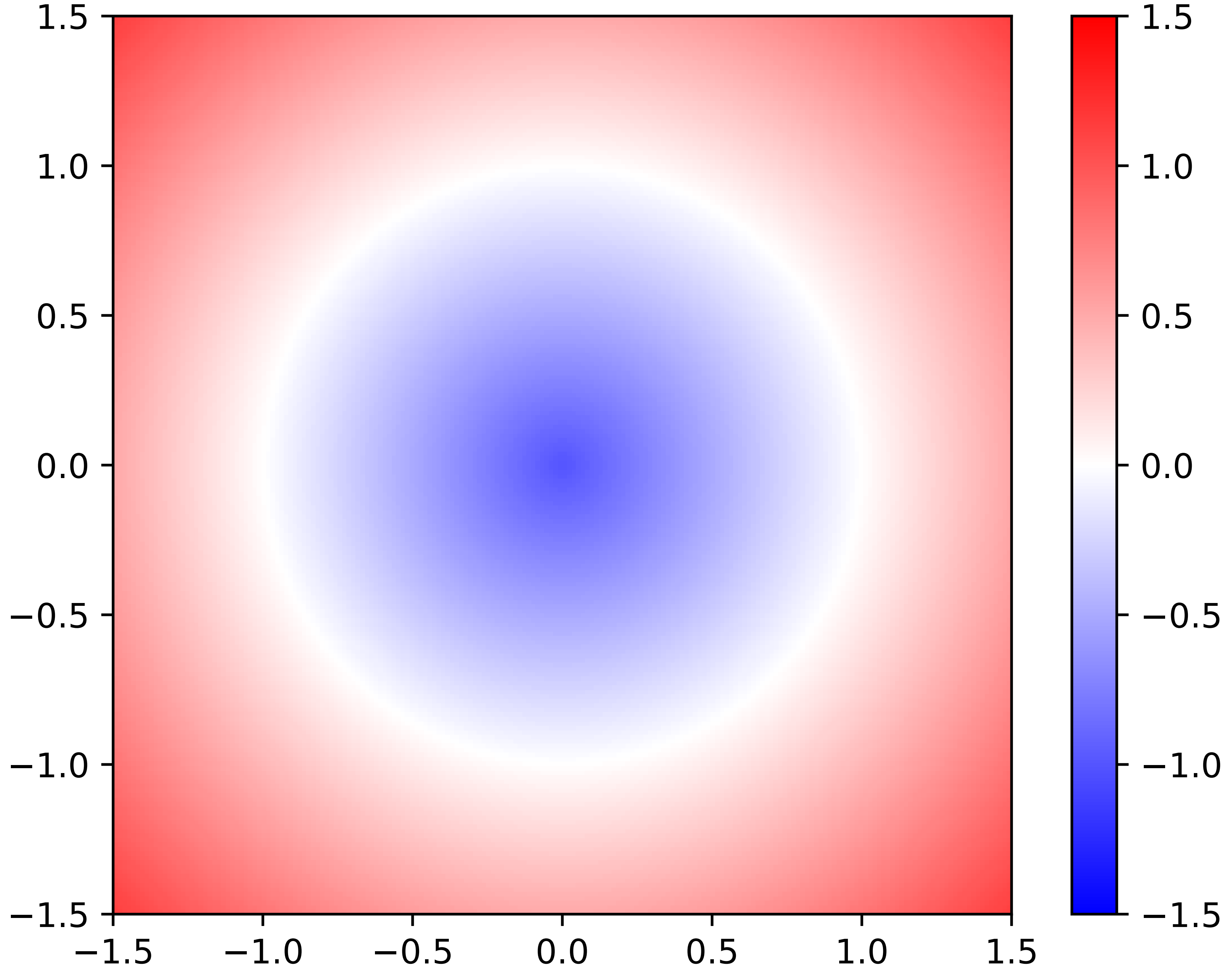
$$\sqrt{x^2 + y^2} - 1$$



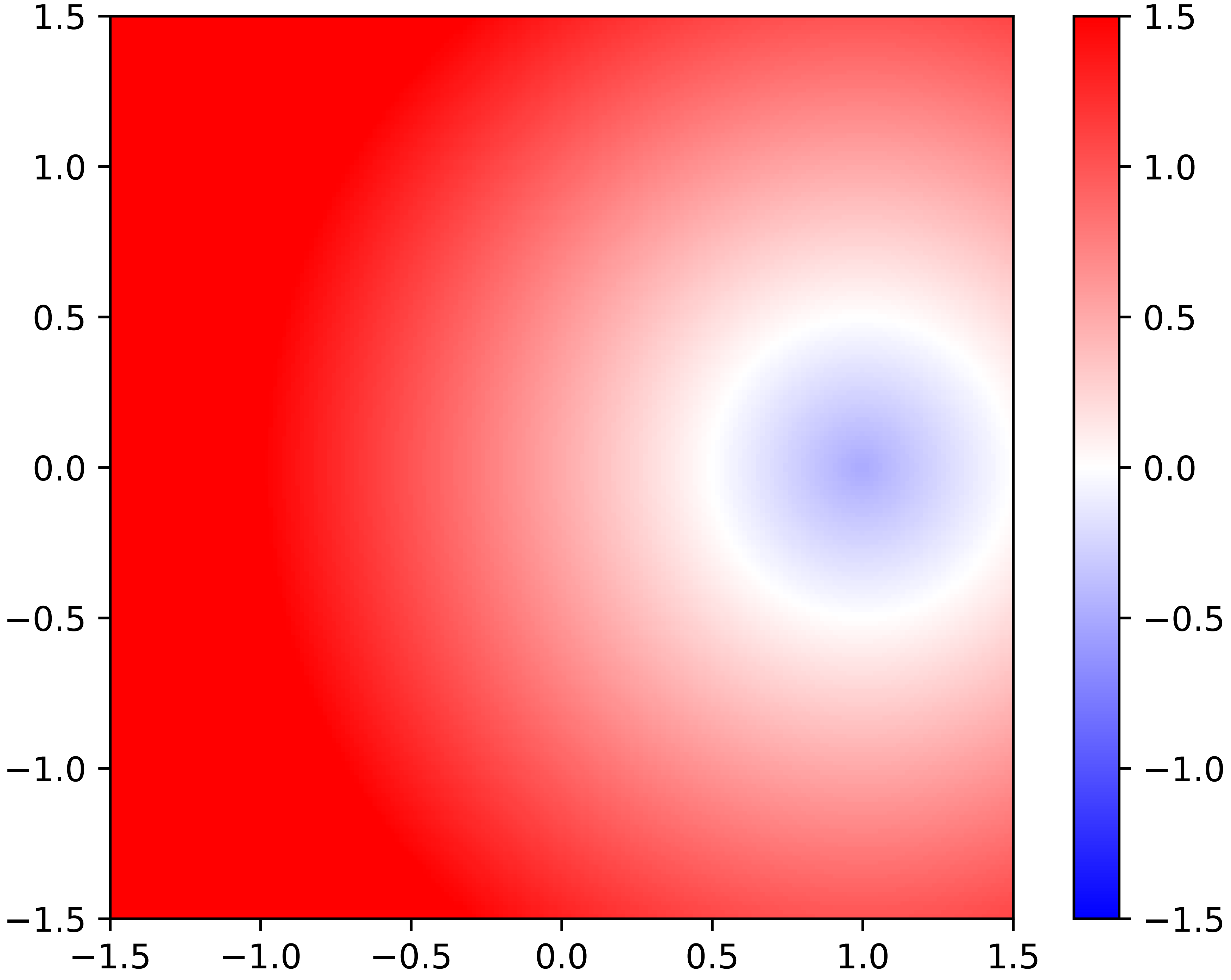
Constructive Solid Geometry

A small example

$$f_1(x, y) = \sqrt{x^2 + y^2} - 1$$



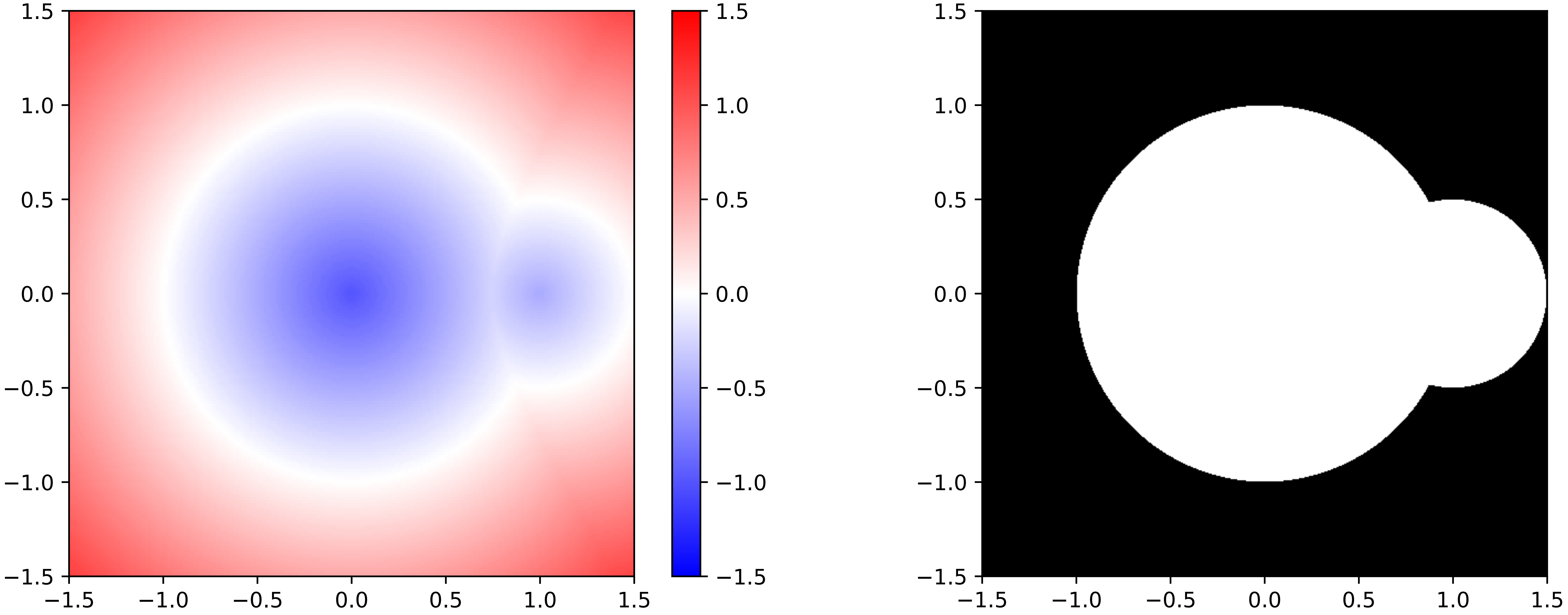
$$f_2(x, y) = \sqrt{(x-1)^2 + y^2} - 0.5$$



Constructive Solid Geometry

Union

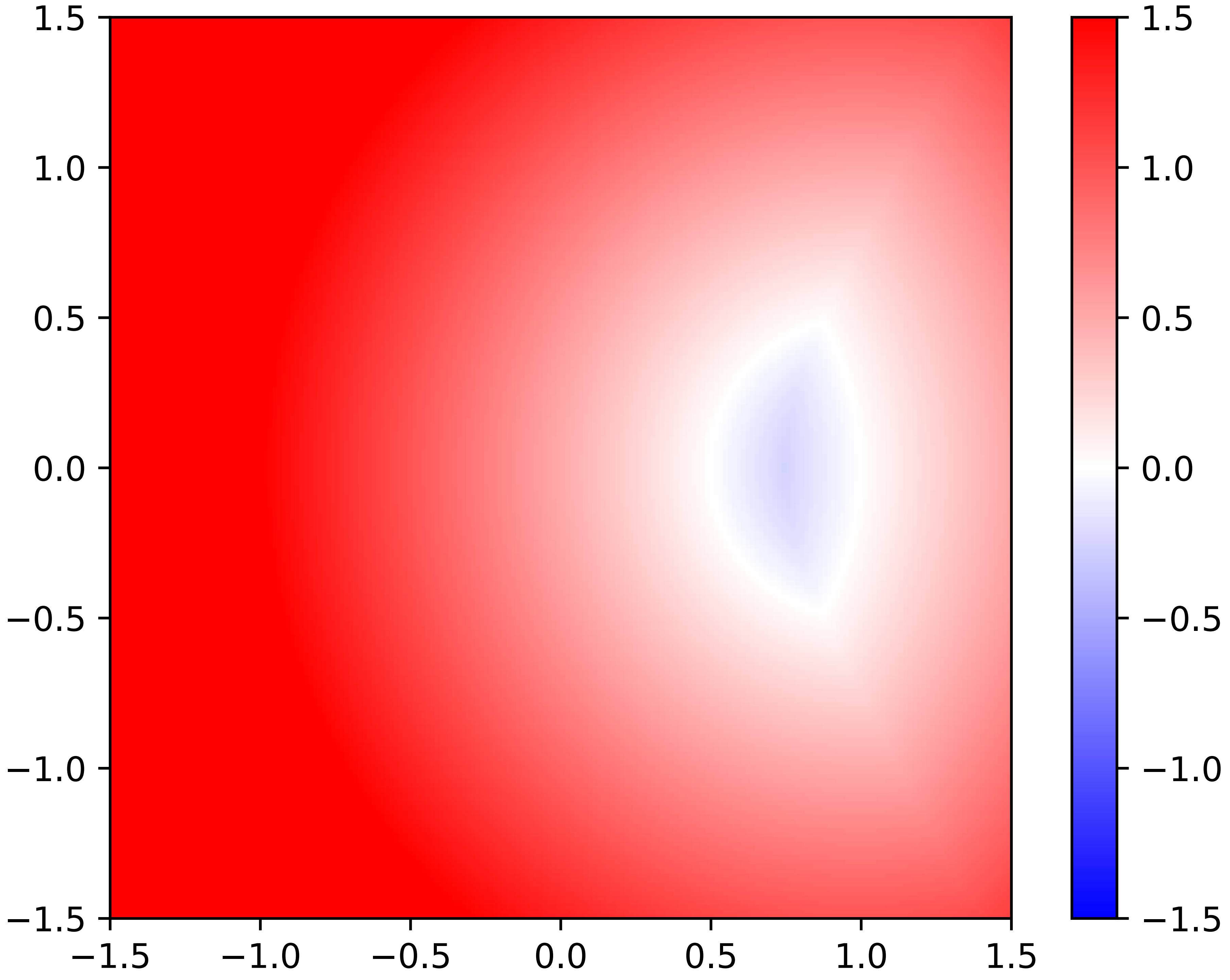
$$\min\left(f_1(x,\ y), f_2(x,\ y)\right)$$

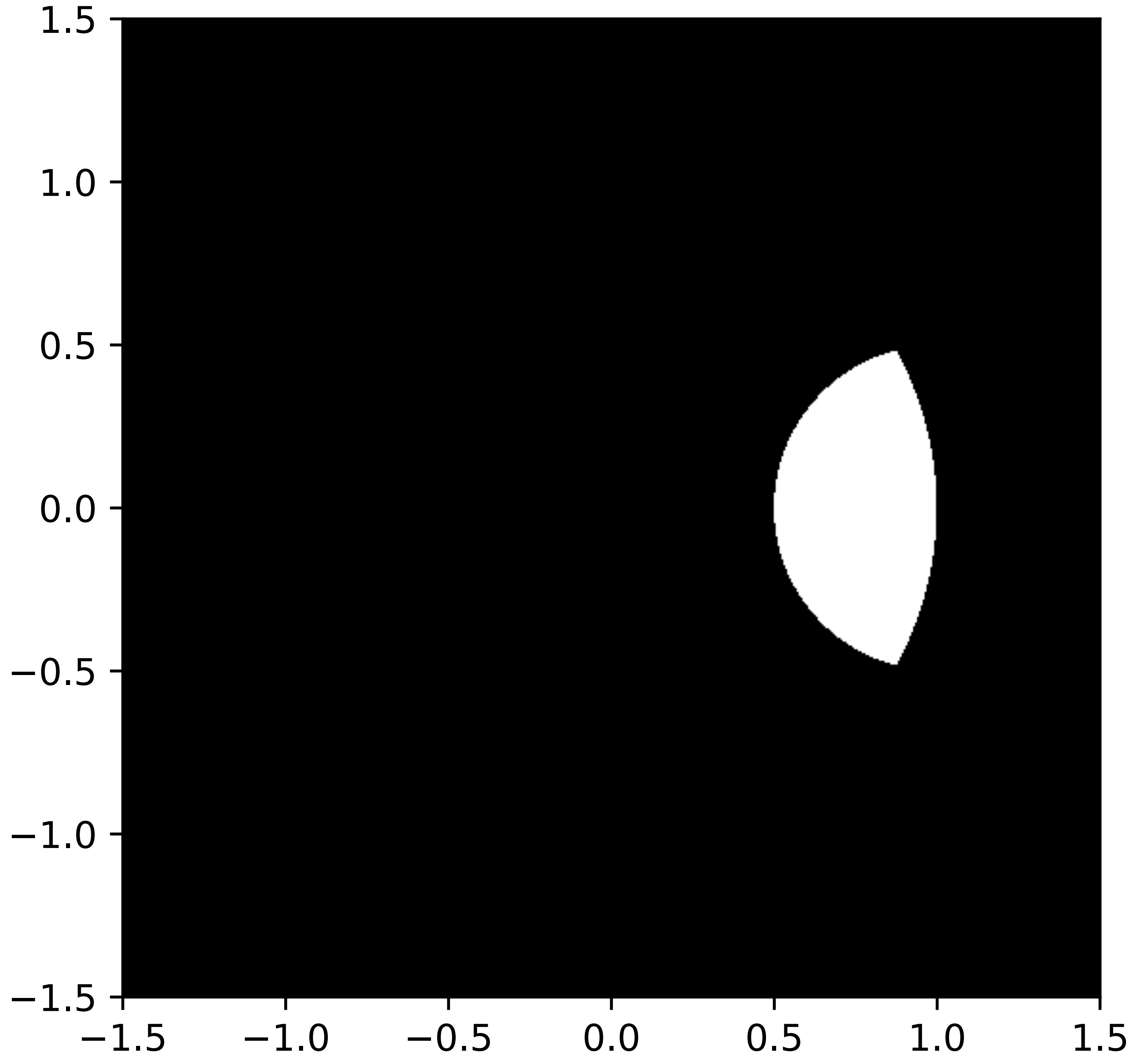


Constructive Solid Geometry

Intersection

$$\max\left(f_1(x, y), f_2(x, y)\right)$$

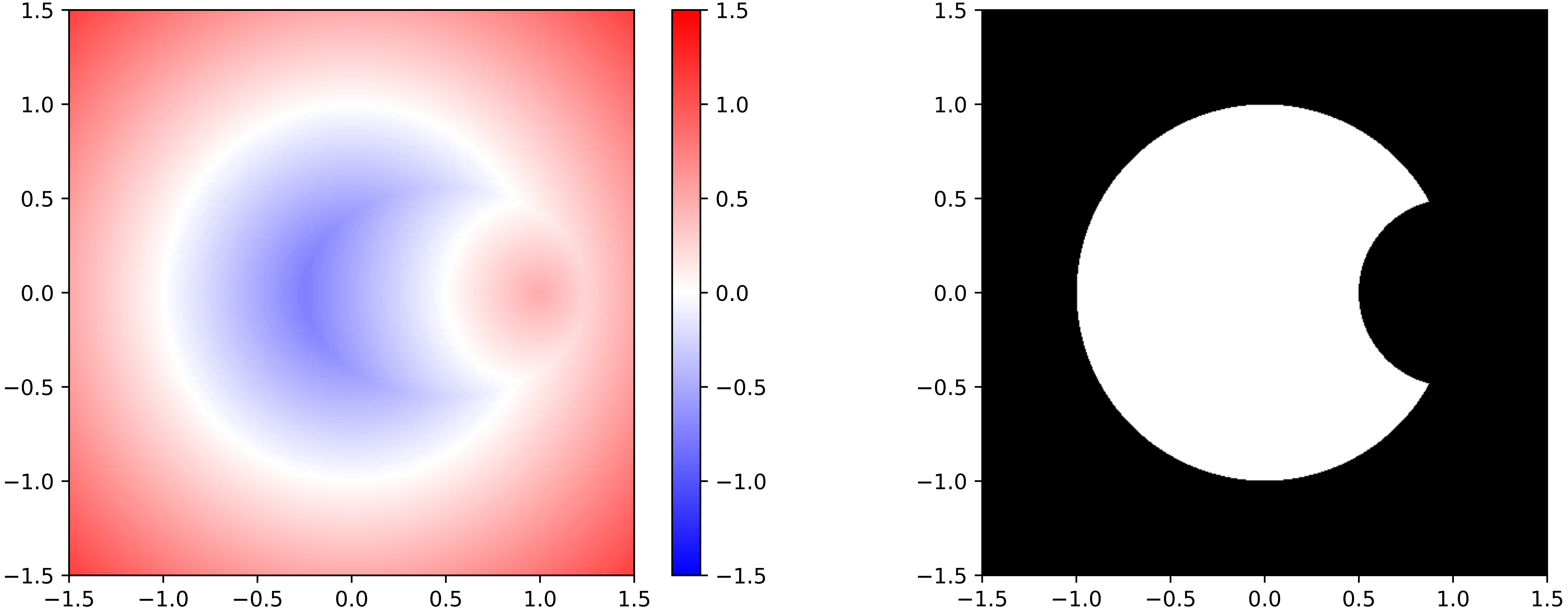




Constructive Solid Geometry

Difference

$$\max\left(f_1(x,\,y),\ -f_2(x,\,y)\right)$$



How do we render these shapes?

Rendering

The naive strategy

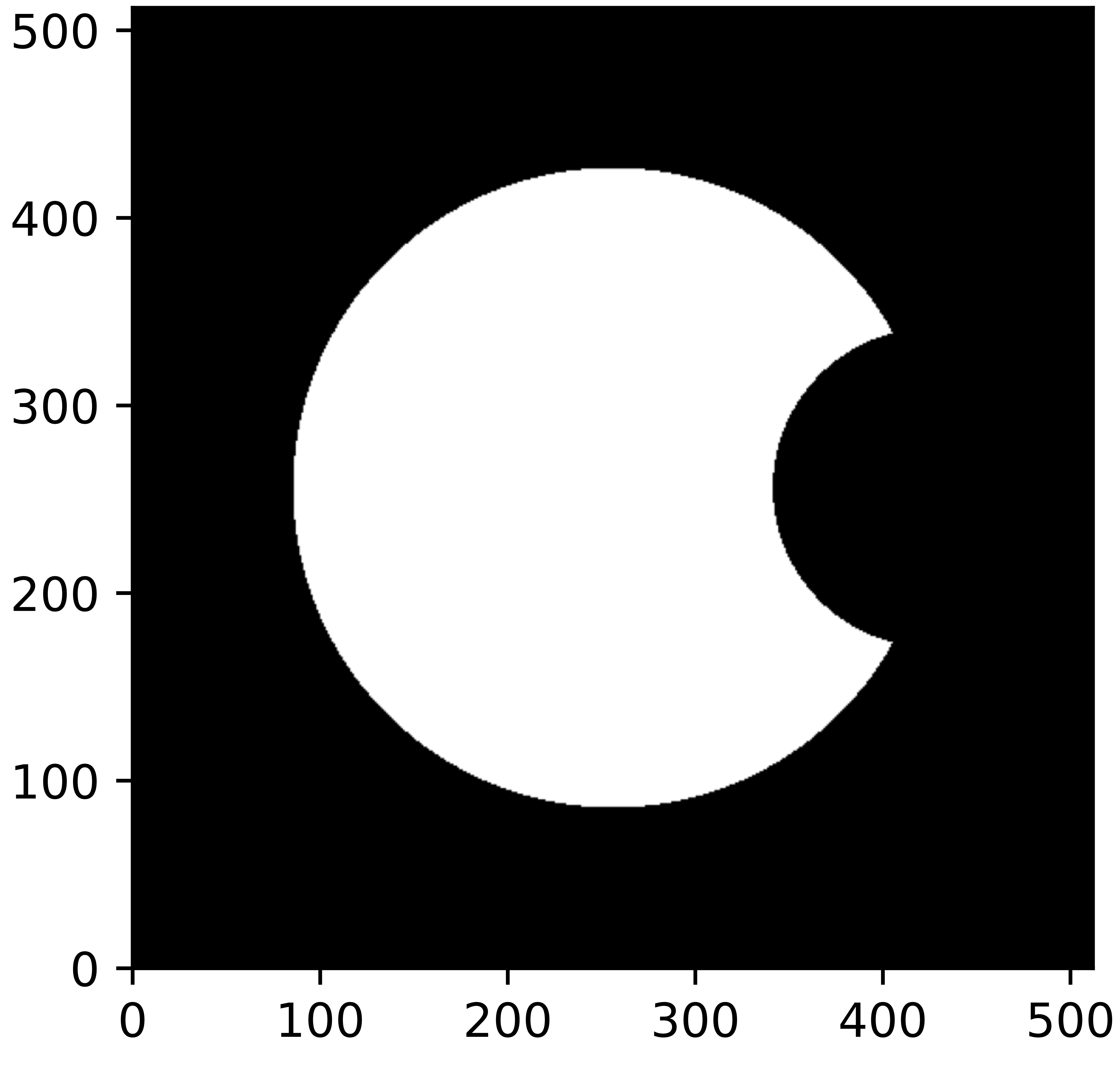
$$\max\left(f_1(x,\ y),\ \ -f_2(x,\ y)\right)$$

Evaluate at every pixel!

$$O(N^2 \times E)$$

N is image size

E is expression size



Rendering in 3D

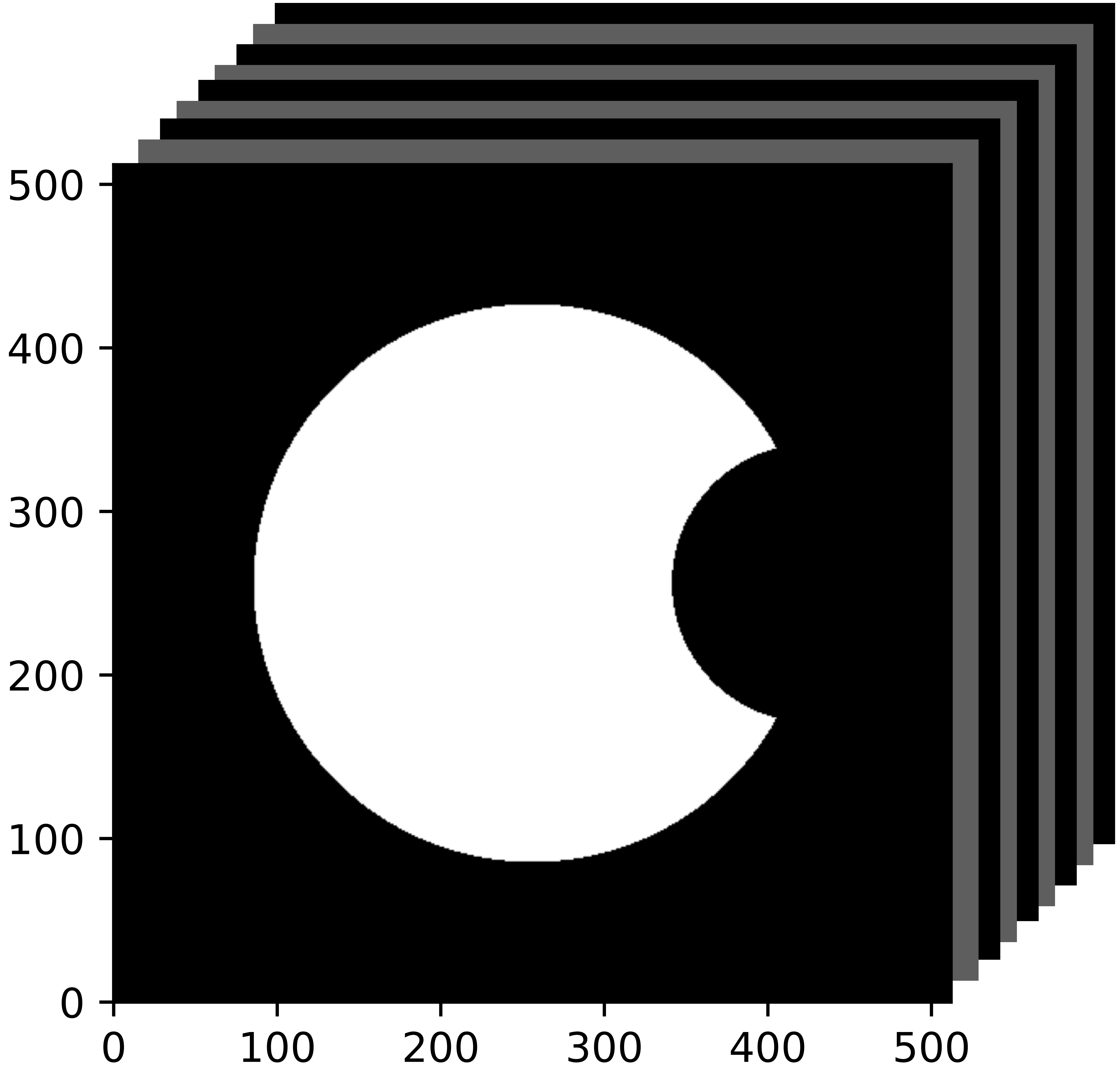
The naive strategy

$$O(N^3 \times E)$$

N is image size

E is expression size

This is not feasible!



One weird trick

Interval arithmetic

```
struct Interval {
    lower: f32,
    upper: f32,
}
```

$$X \in [0,\ 1]$$

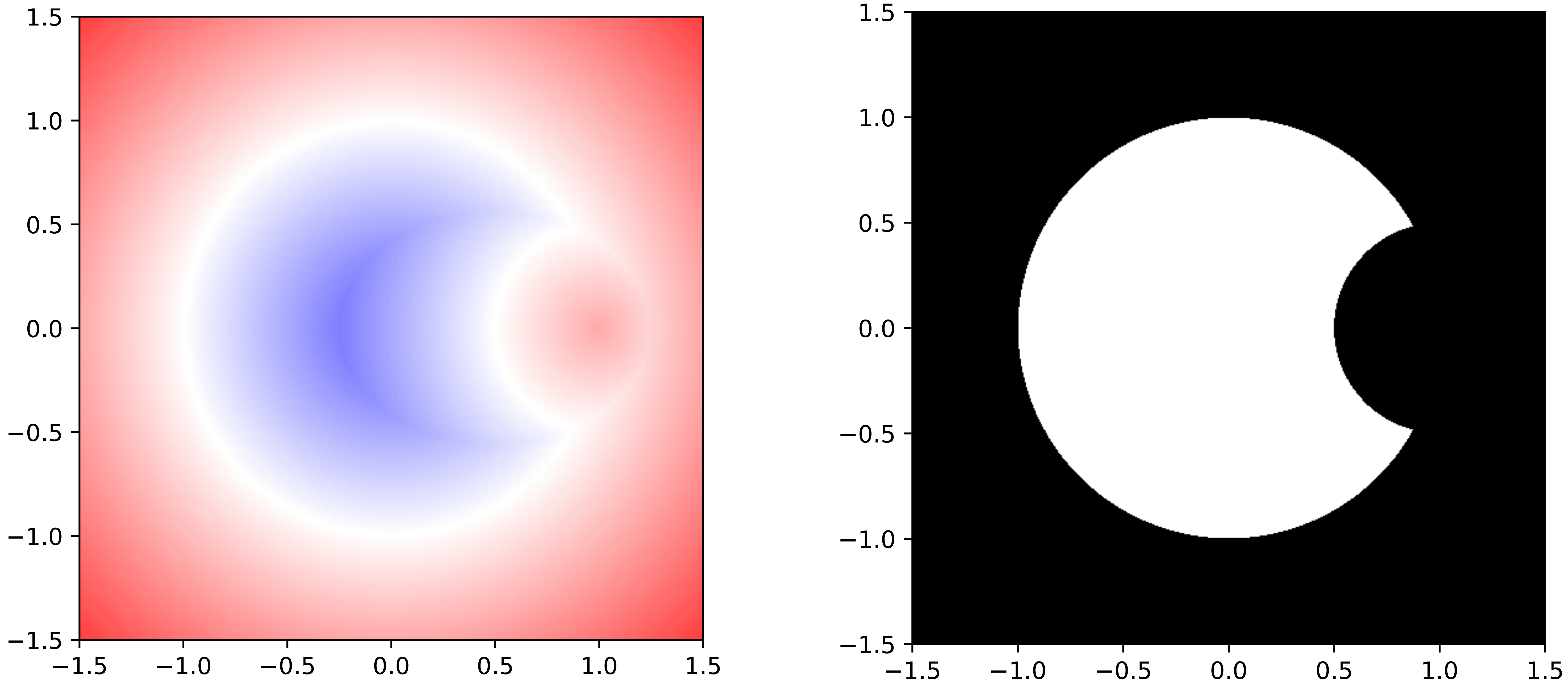
$$Y \in [2,\ 4]$$

$$X + Y \in [2,\ 5]$$

One weird trick

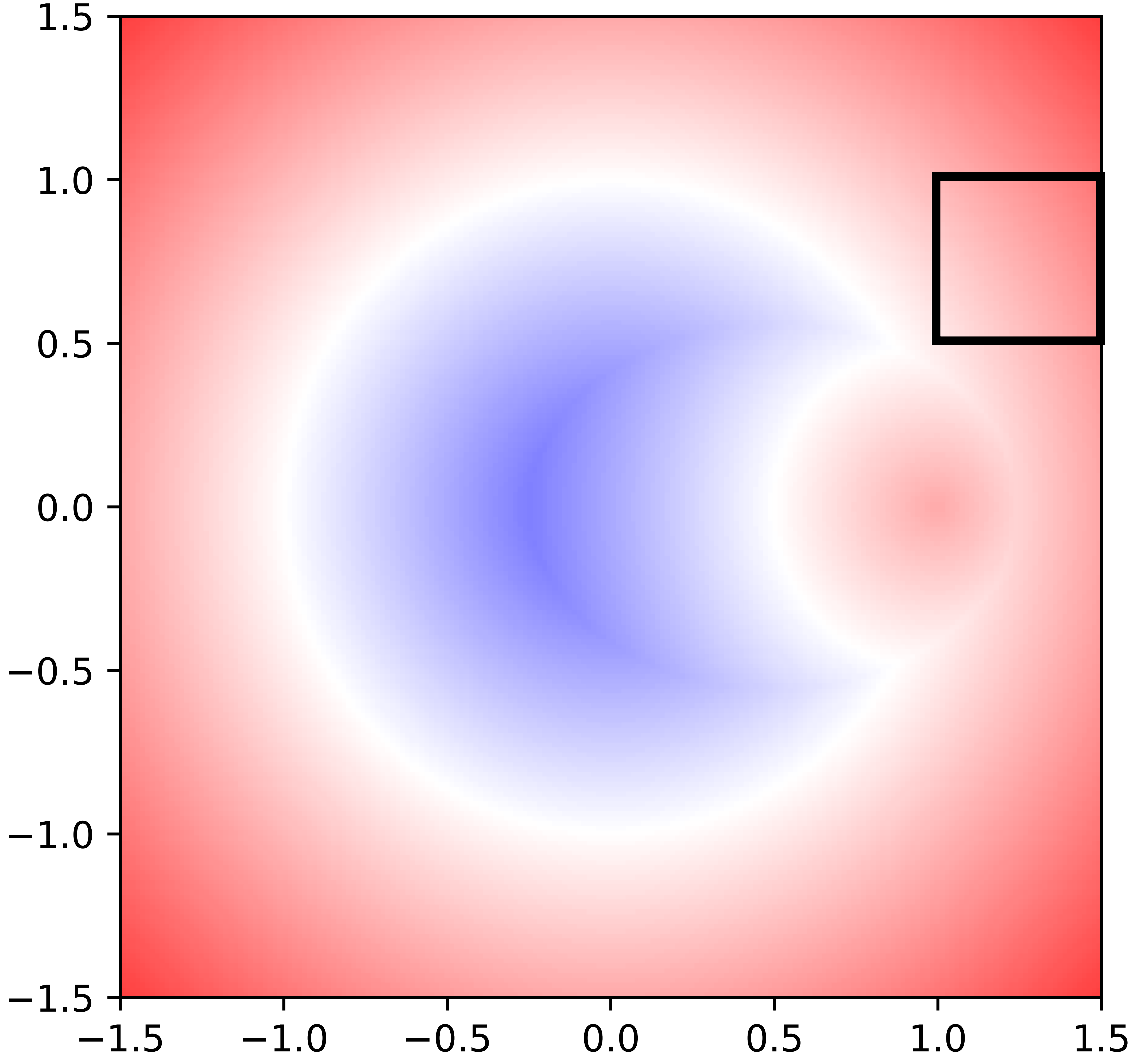
Interval arithmetic

$$f(x, y) = \max\left(\sqrt{x^2 + y^2} - 1,\ 0.5 - \sqrt{(x-1)^2 + y^2}\right)$$



One weird trick

Interval arithmetic



$X \in [1,\ 1.5]$

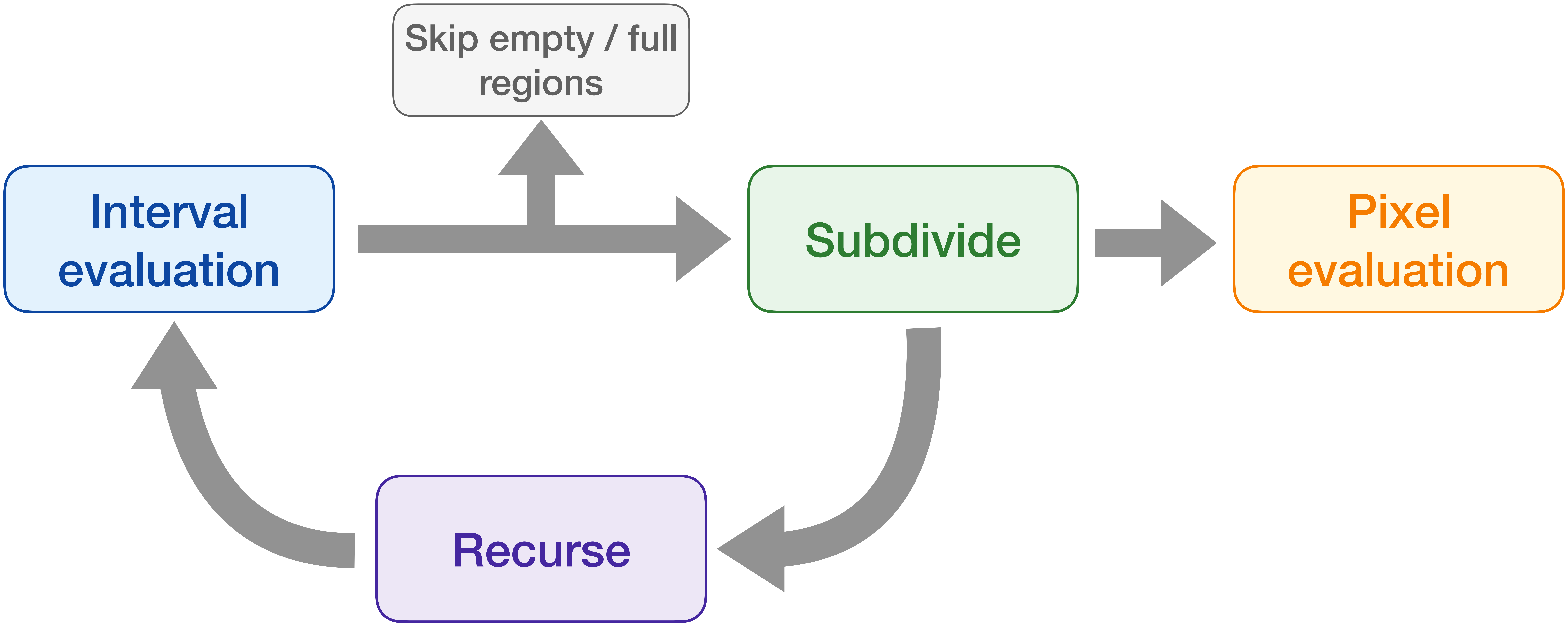
$Y \in [0.5,\ 1]$

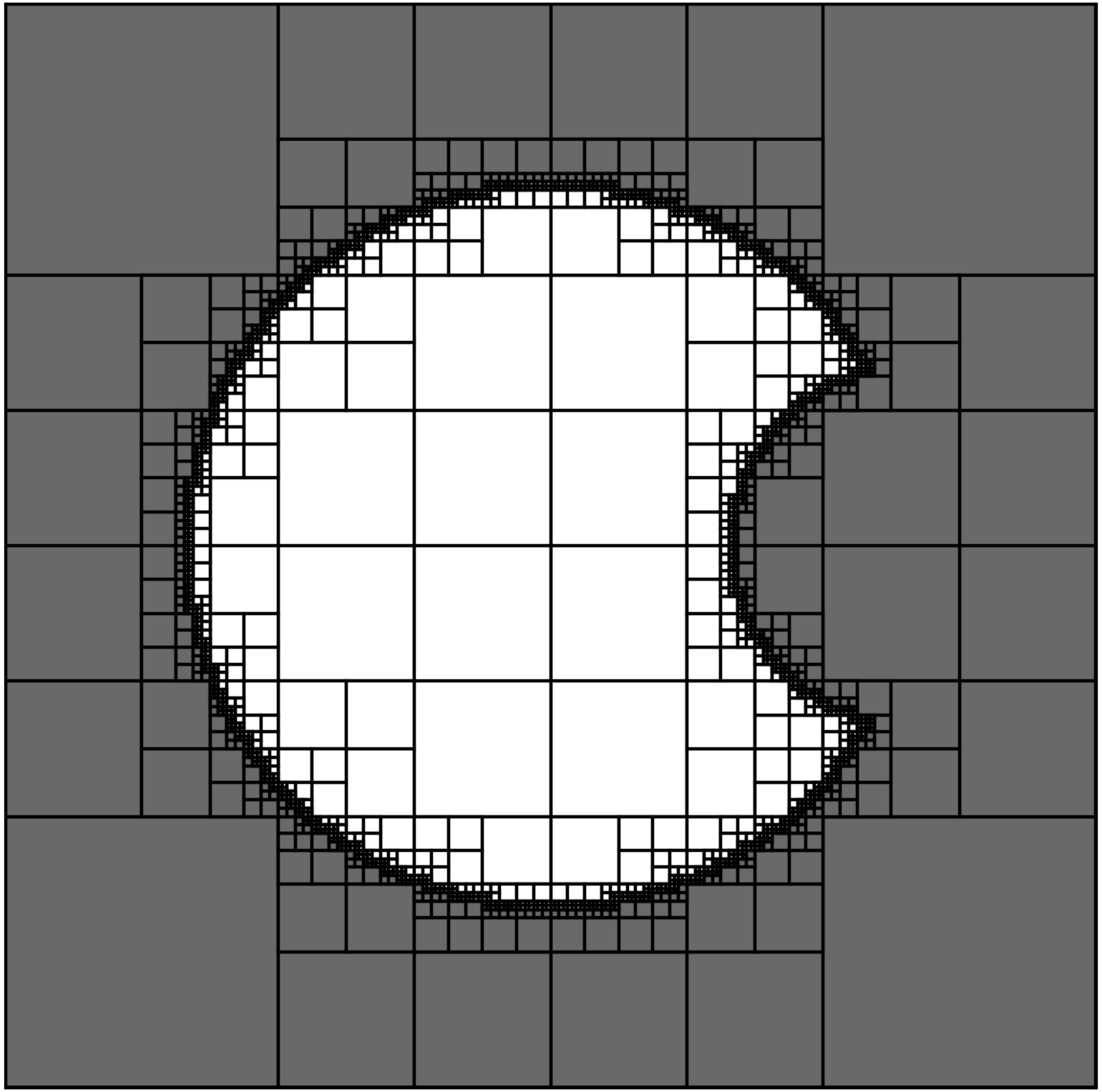
$f(X, Y) \in [0.11,\ 0.80]$

$f(X, Y) > 0$

Interval arithmetic lets us prove regions empty or full

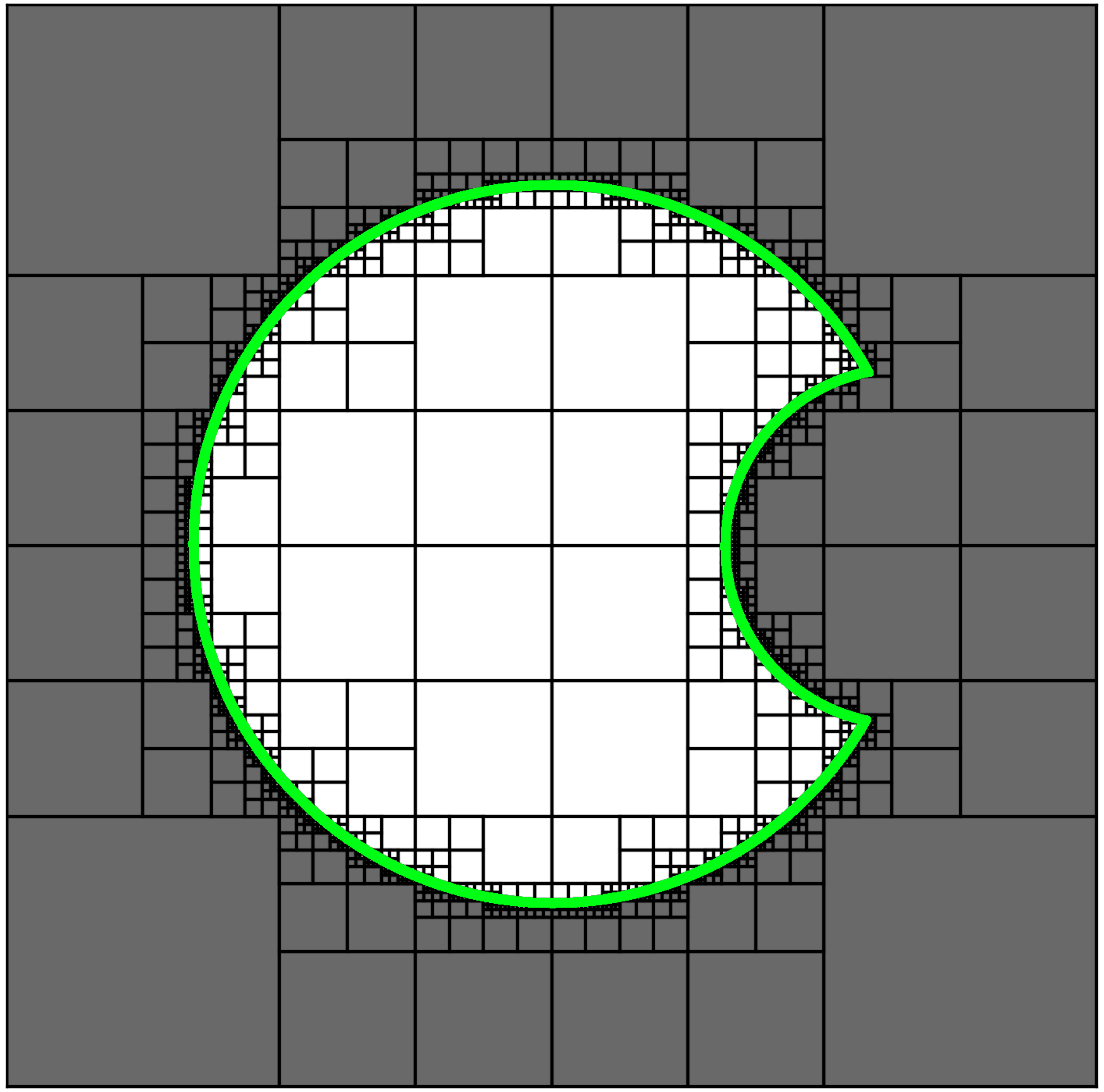
The render loop





Evaluation complexity

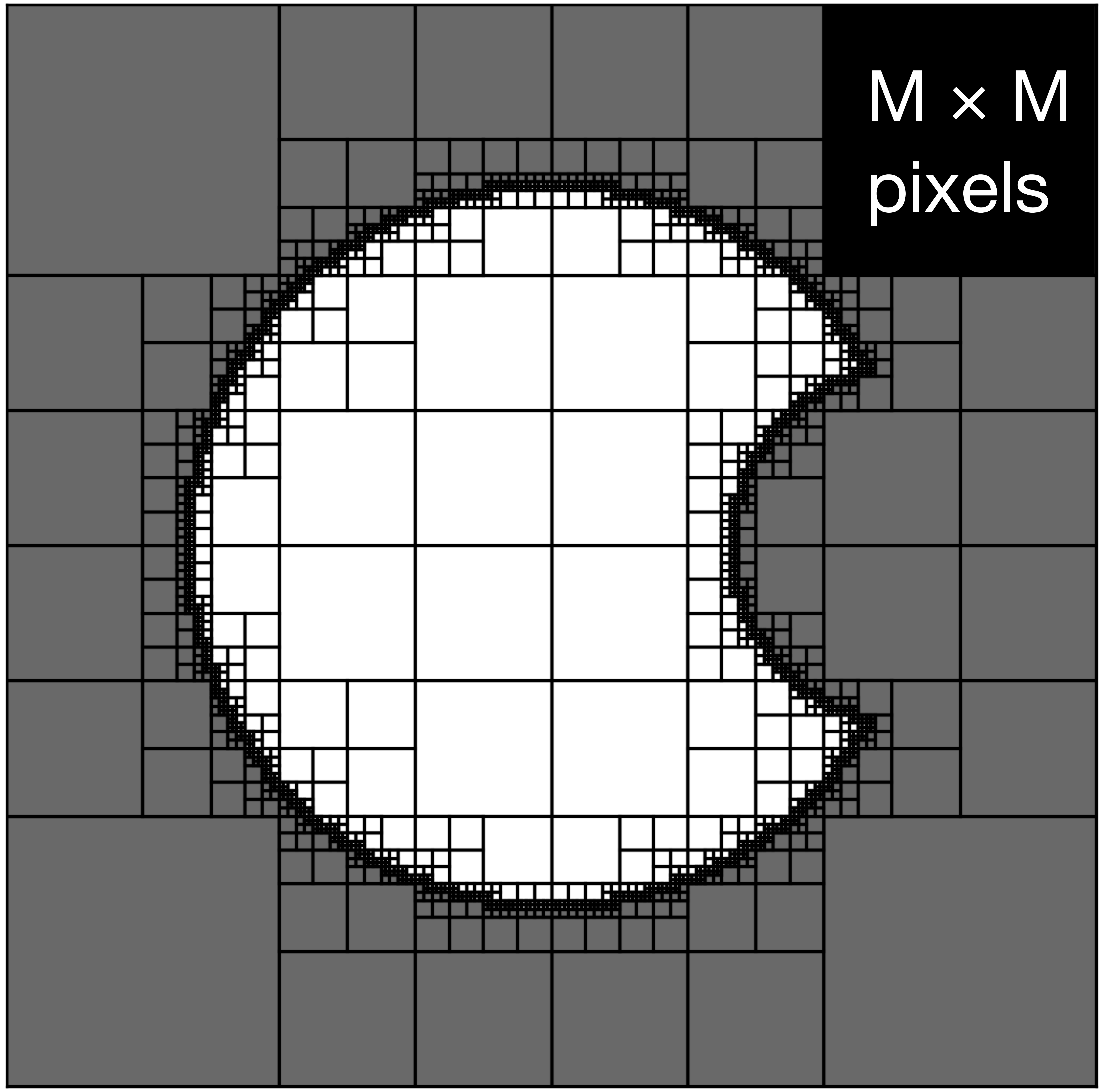
Reduced dimensionality



Work is concentrated at the model's edges

Evaluation complexity

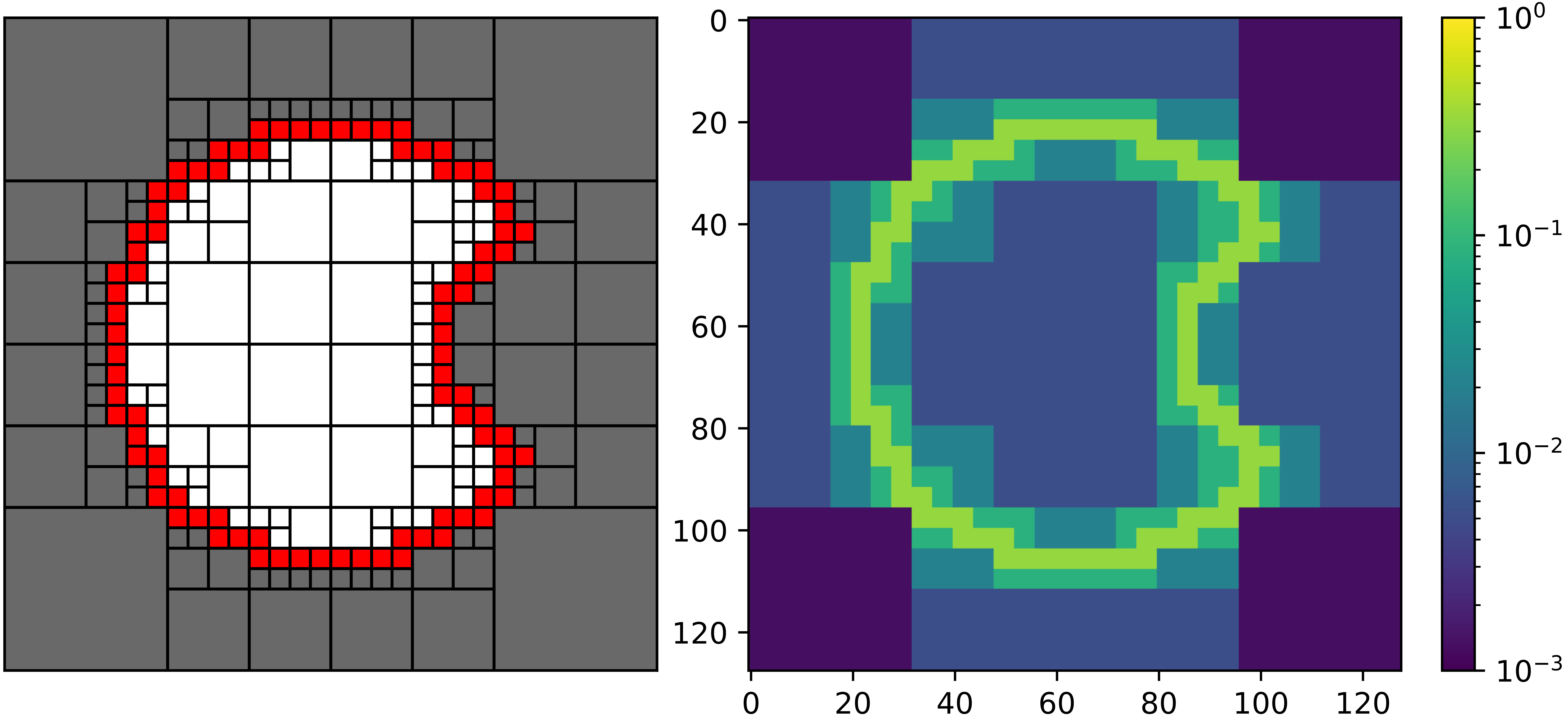
Amortization over pixels



The expression is evaluated **once** for this region

Interval evaluation cost is *amortized* over pixels

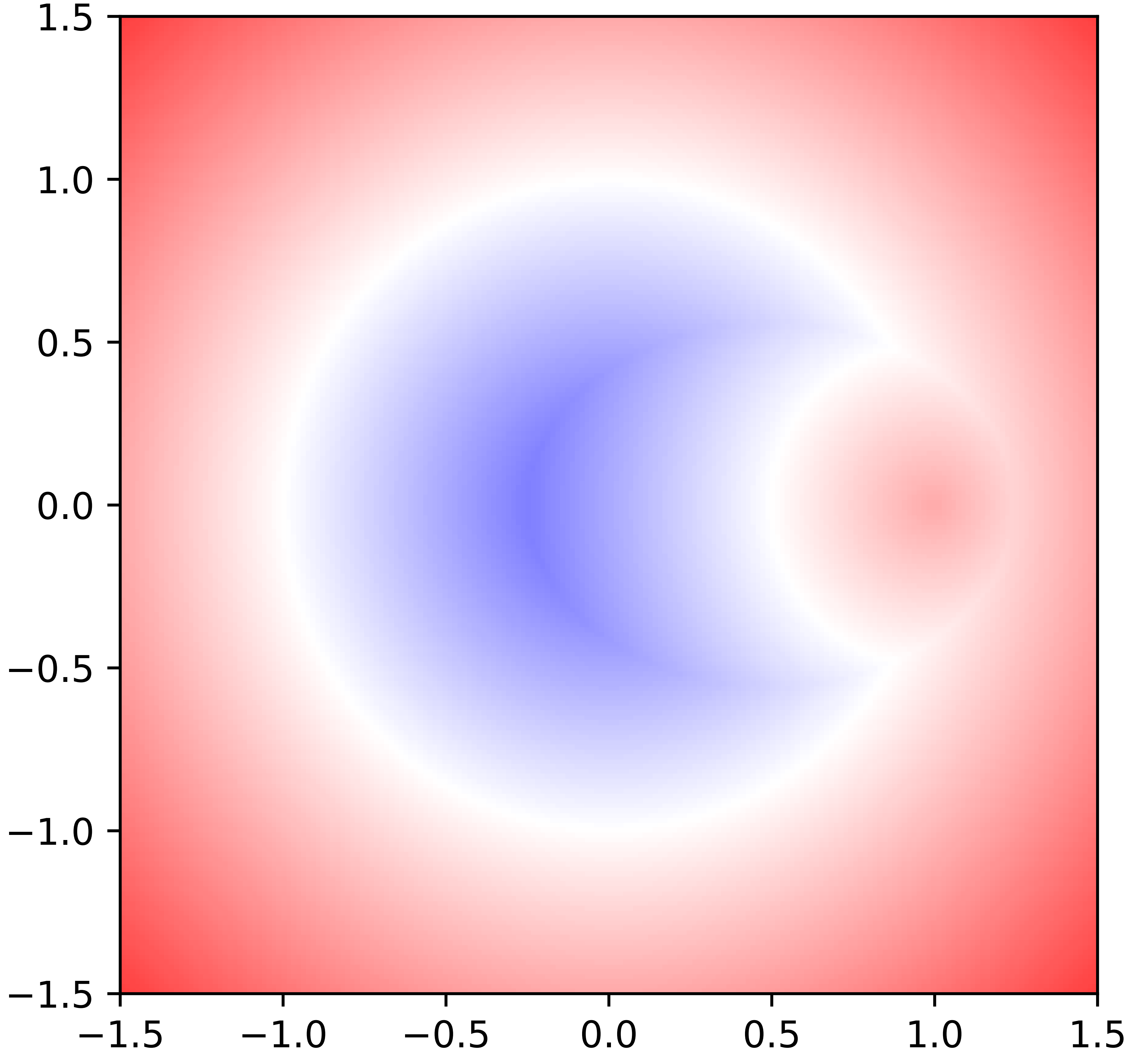
Amortized evaluation count



Expression simplification

The second weird trick

$$\max\left(\sqrt{x^2+y^2}-1,\ 0.5-\sqrt{(x-1)^2+y^2}\right)$$



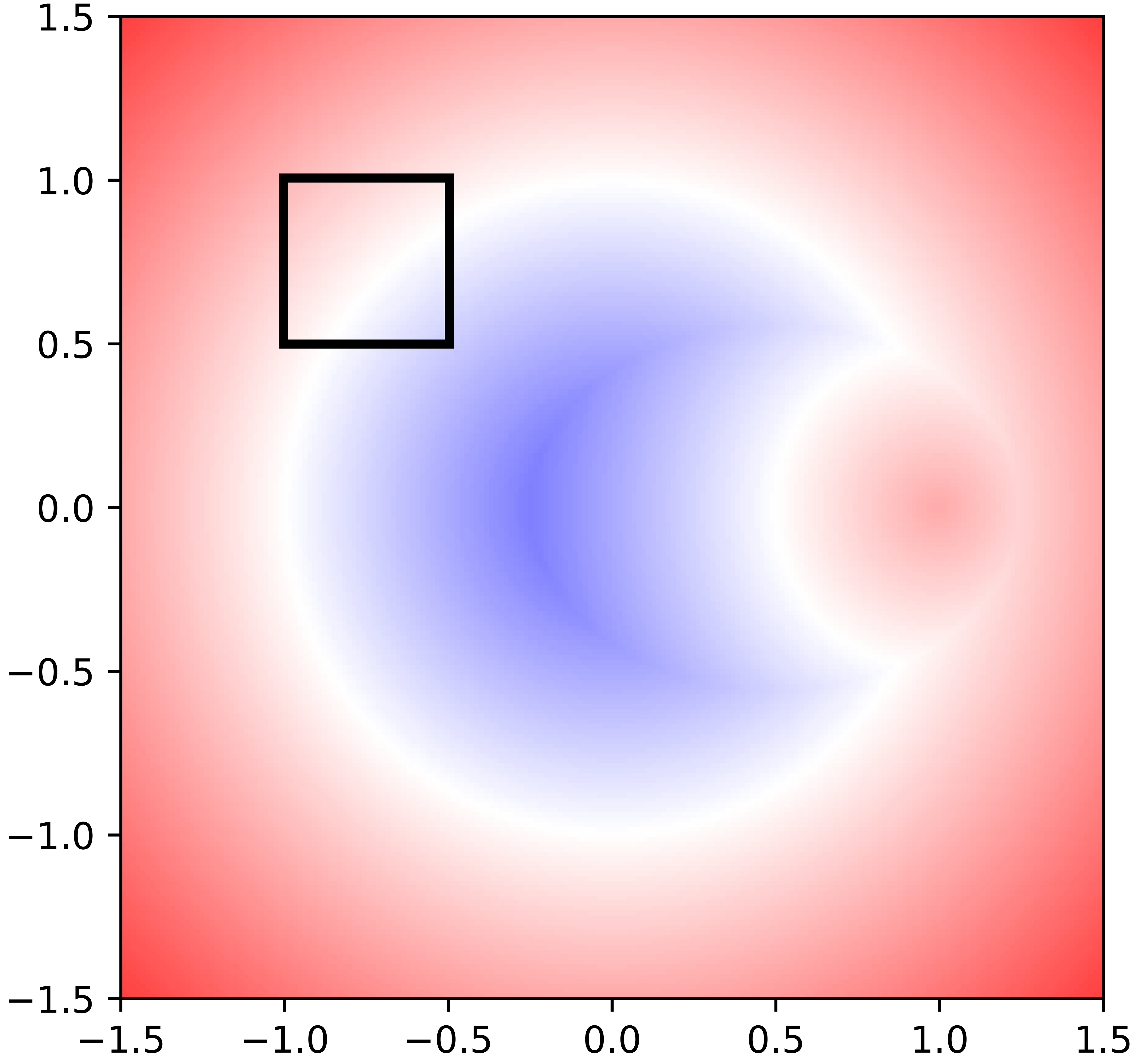
Expression simplification

The second weird trick

$$\max\left(\sqrt{x^2+y^2}-1,\ 0.5-\sqrt{(x-1)^2+y^2}\right)$$

$$X \in [-1,\ -0.5]$$
$$Y \in [0.5,\ 1]$$

$$\max\left([-0.3,\ 0.4],\ [-1.7,\ -1.1]\right)$$



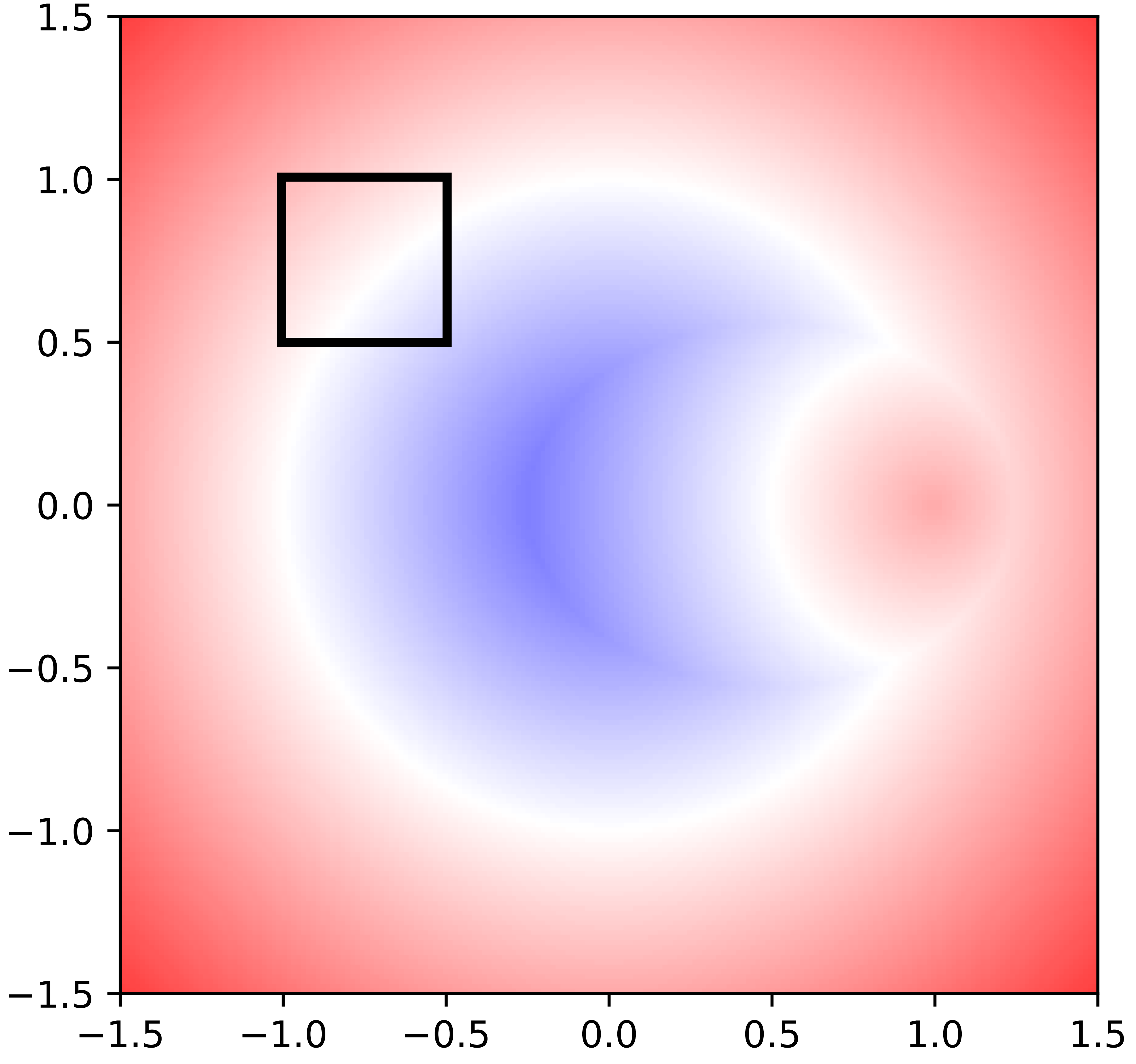
Expression simplification

The second weird trick

$$\max\left(\sqrt{x^2+y^2}-1,\ 0.5-\sqrt{(x-1)^2+y^2}\right)$$

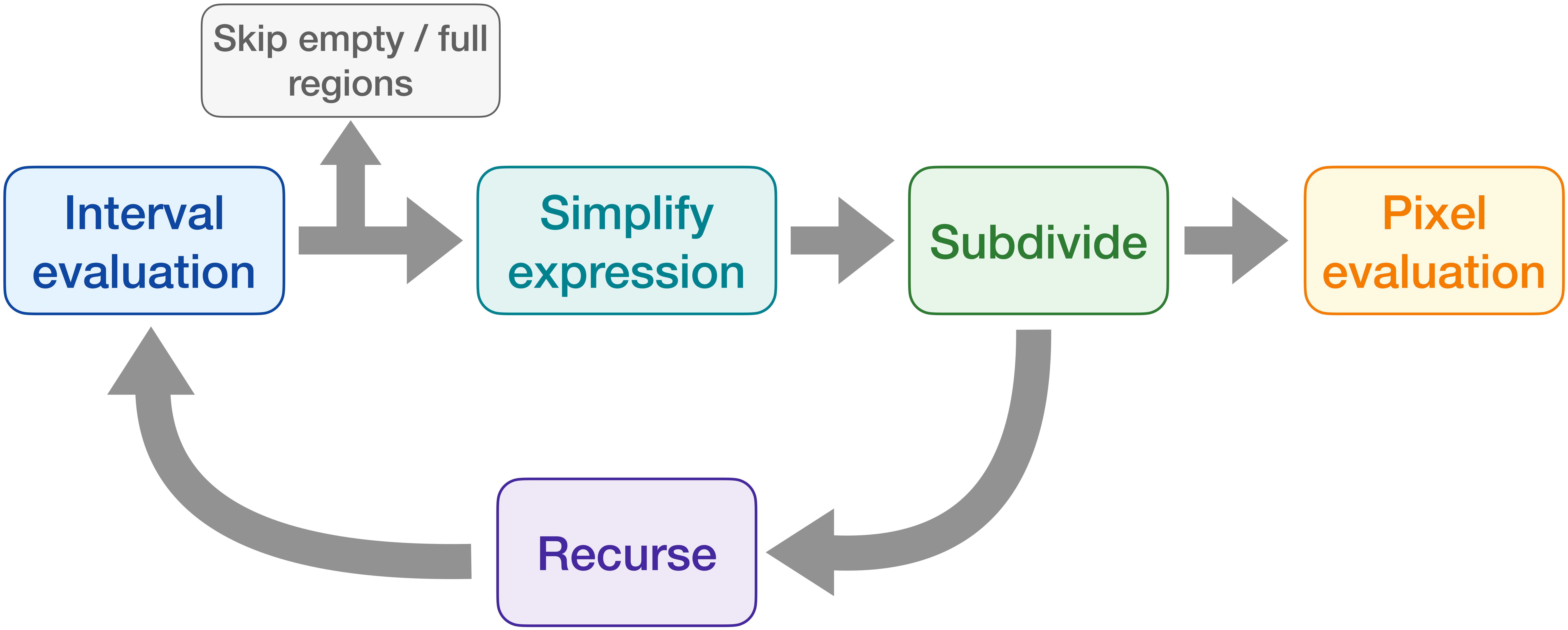
$$\max\left([-0.3,\ 0.4],\ [-1.7,\ -1.1]\right)$$

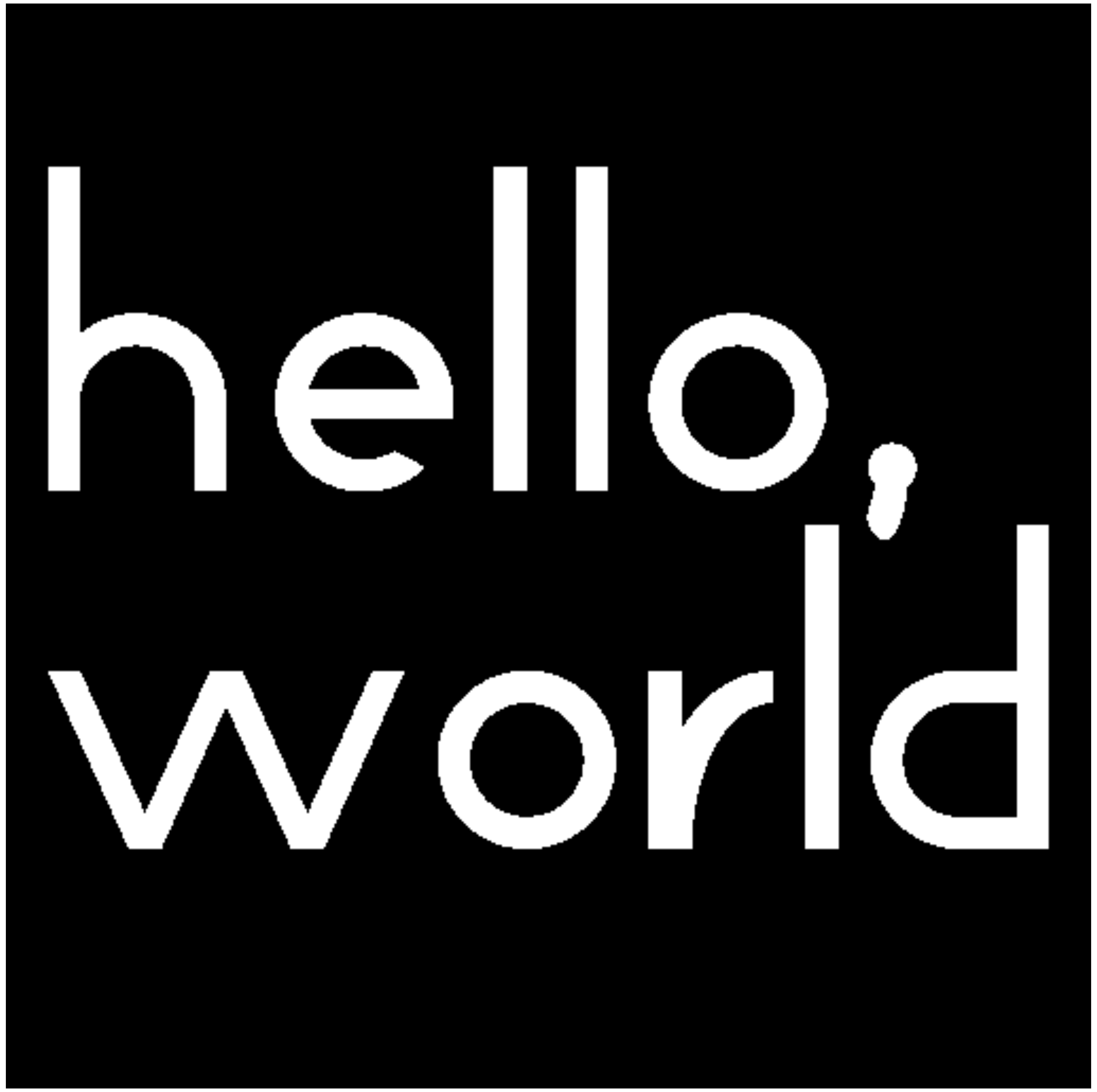
Within this region, we can simplify the expression to $\sqrt{x^2+y^2}-1$



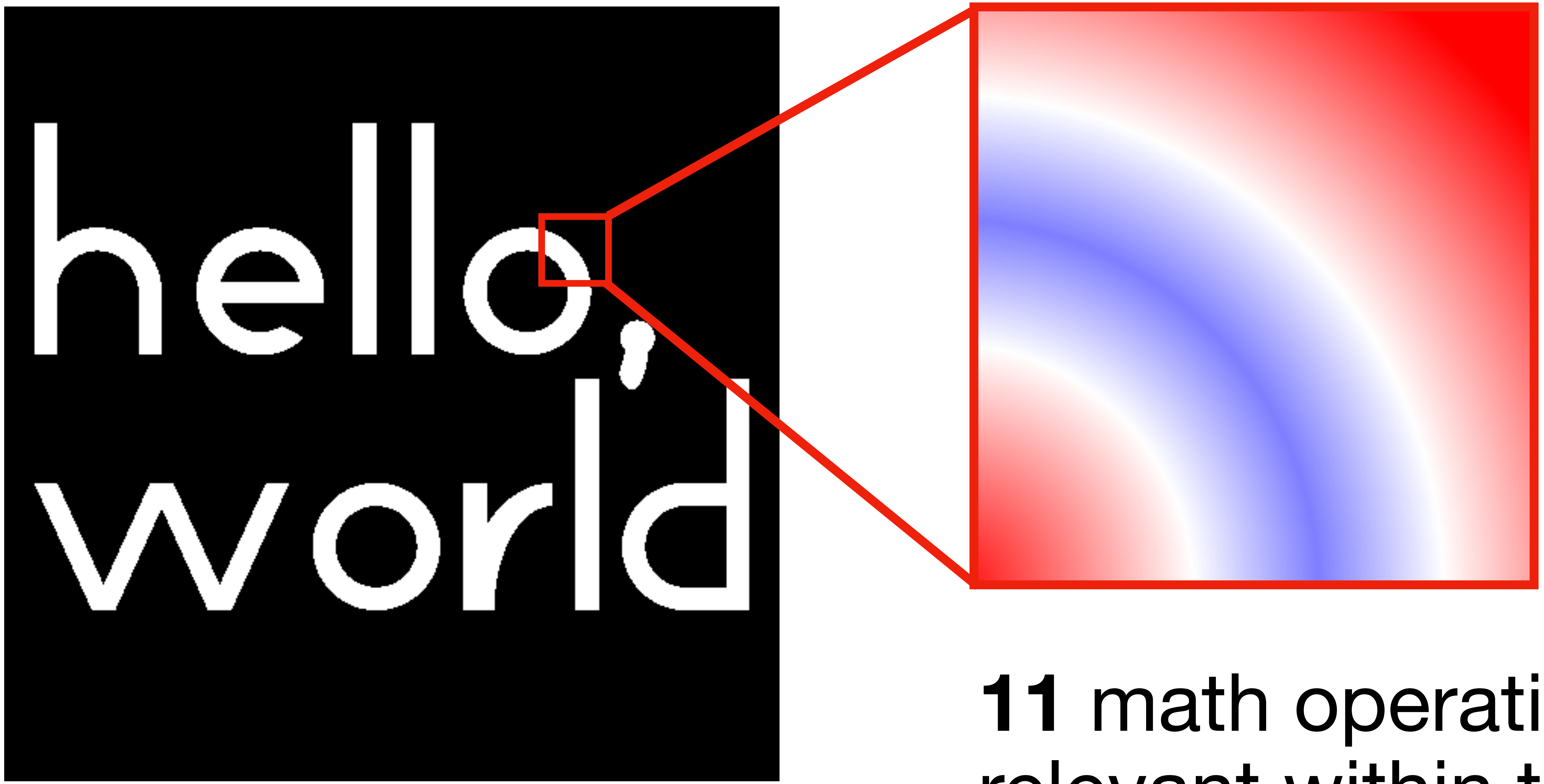
Interval arithmetic lets us skip entire chunks of computation

Modified render loop

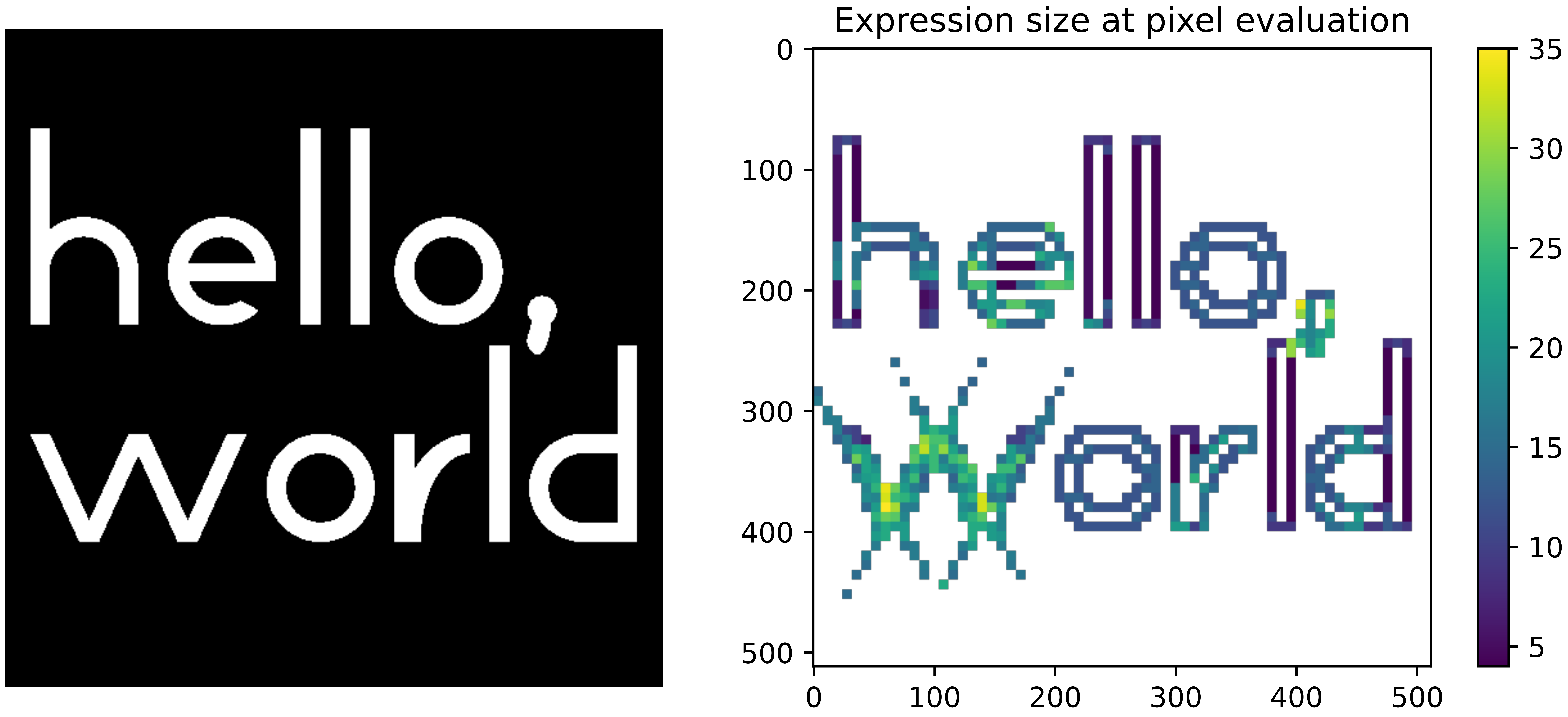


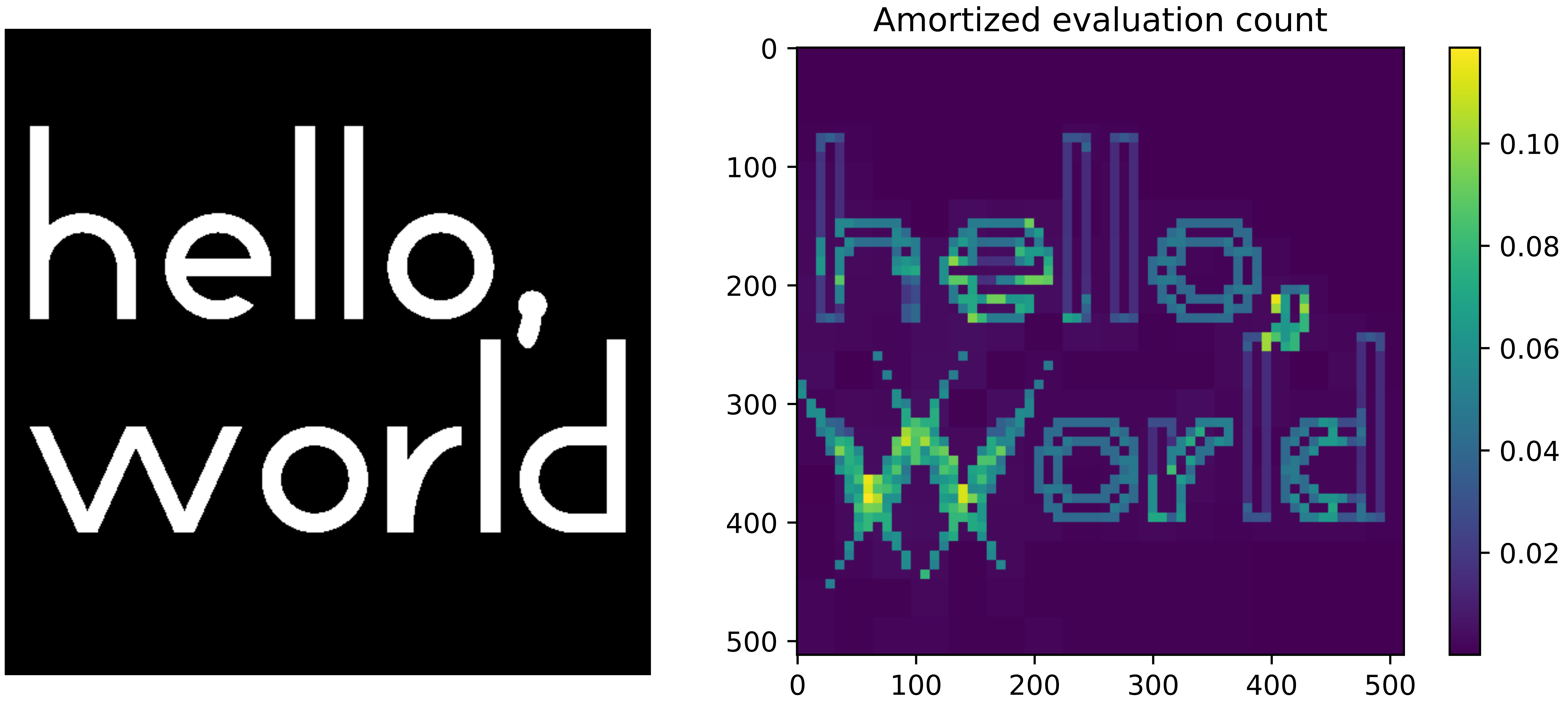


304 math operations,
129 of which are CSG



11 math operations are relevant within this region

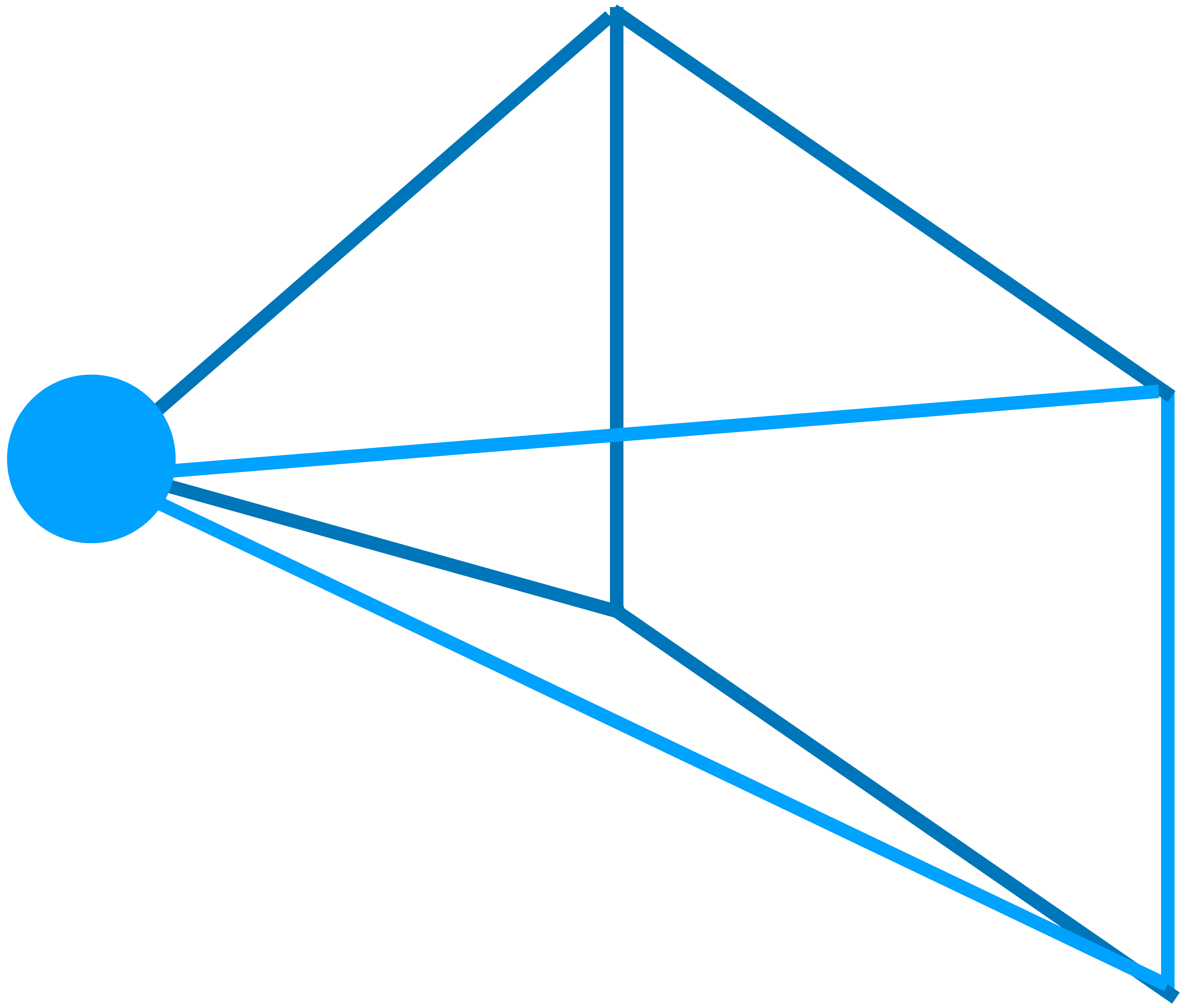

hello,
world
Expression size at pixel evaluation
0
100
200
300
400
500
0
100
200
300
400
500
35
30
25
20
15
10
5

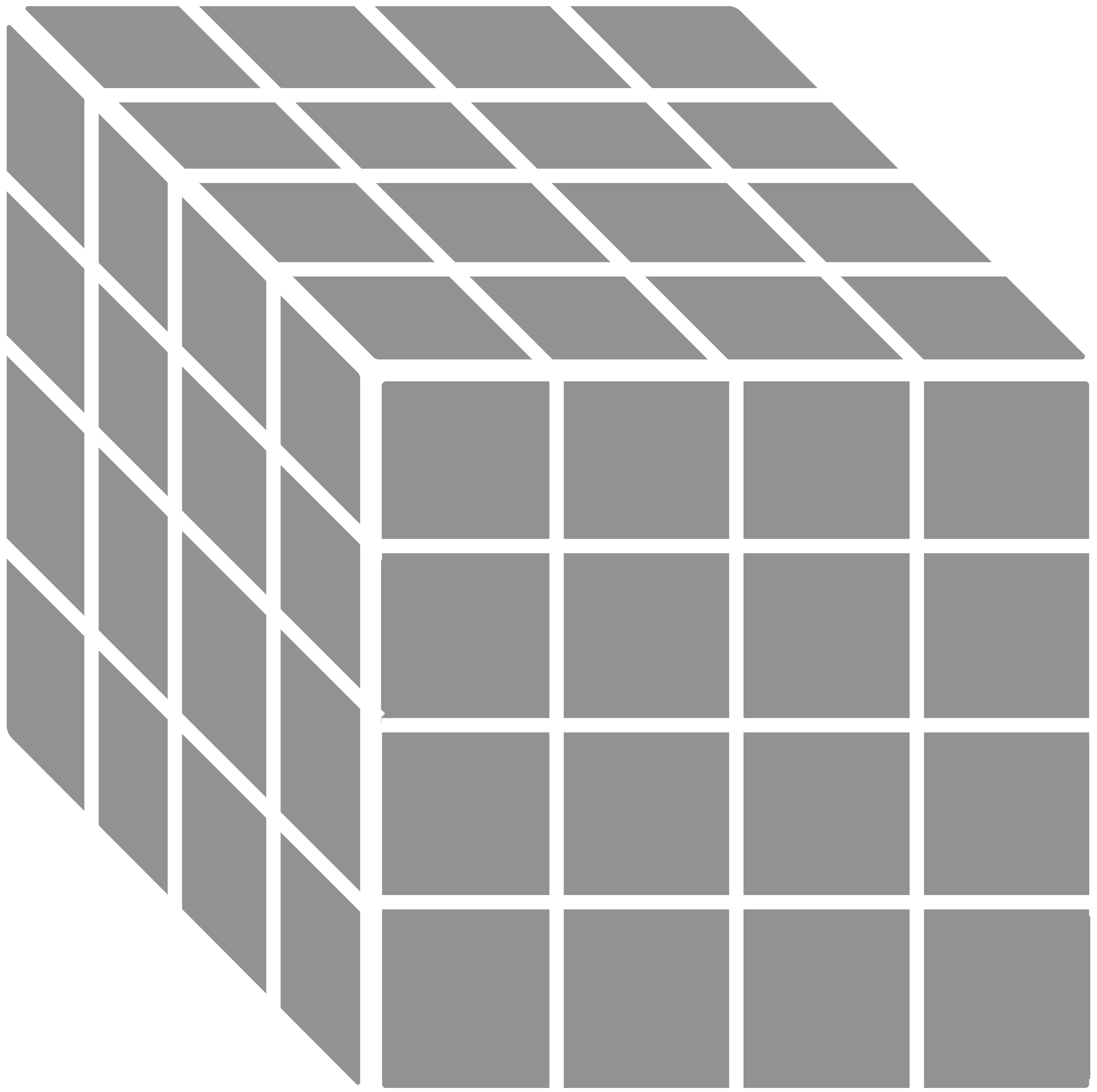

hello,
world
Amortized evaluation count
0
100
200
300
400
500
0
100
200
300
400
500
0.10
0.08
0.06
0.04
0.02

From 2D to 3D

3D rendering

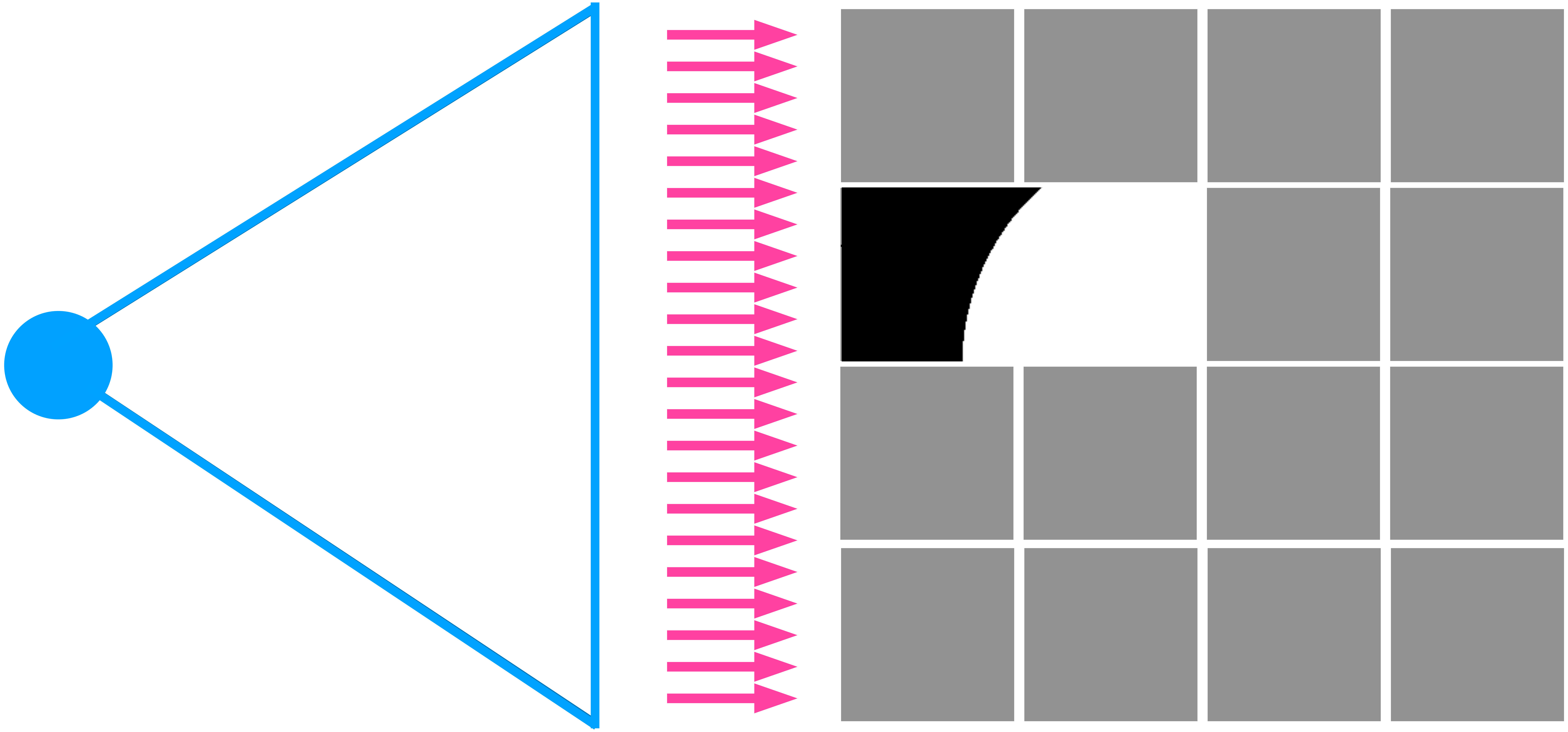
It’s basically the same!





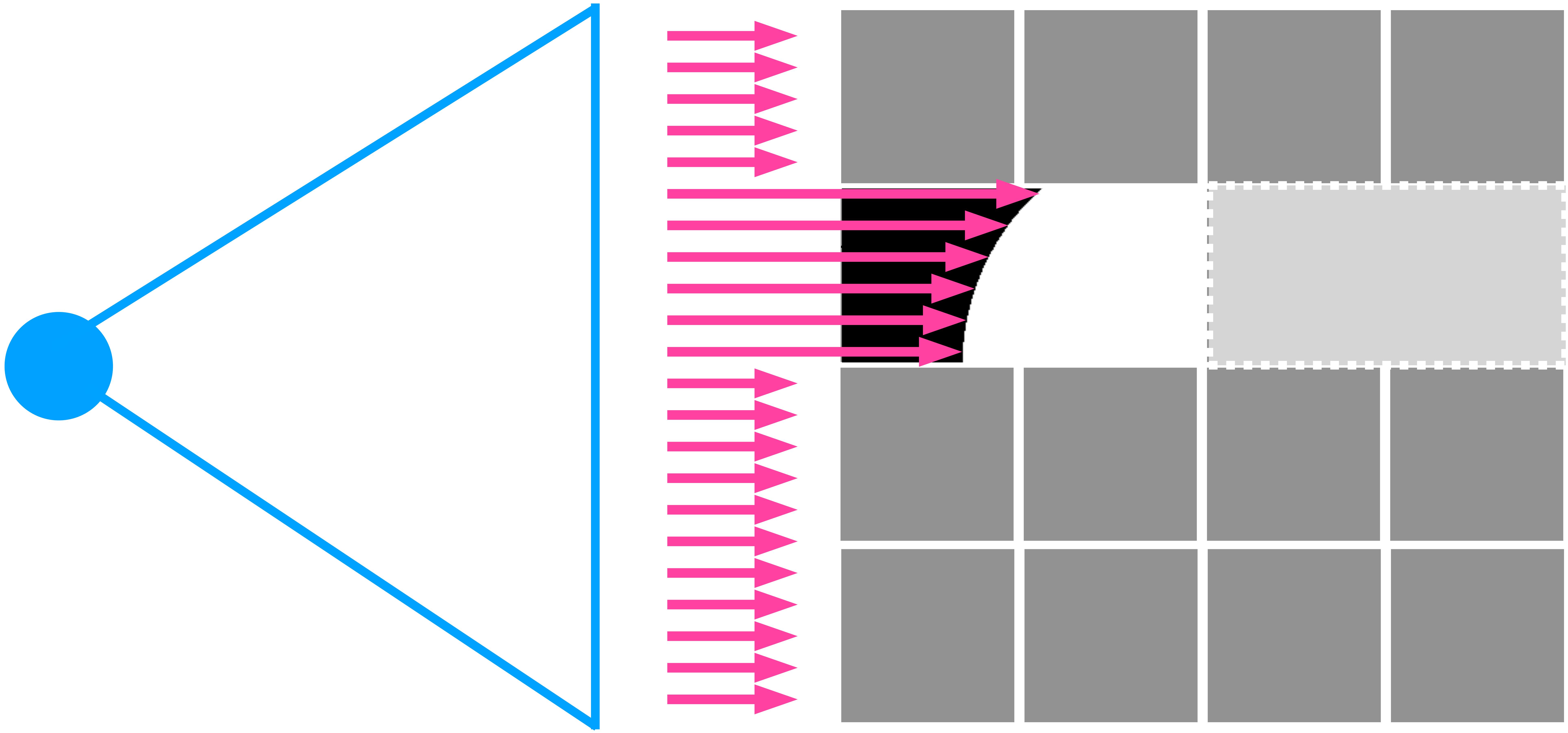
3D rendering

Side view

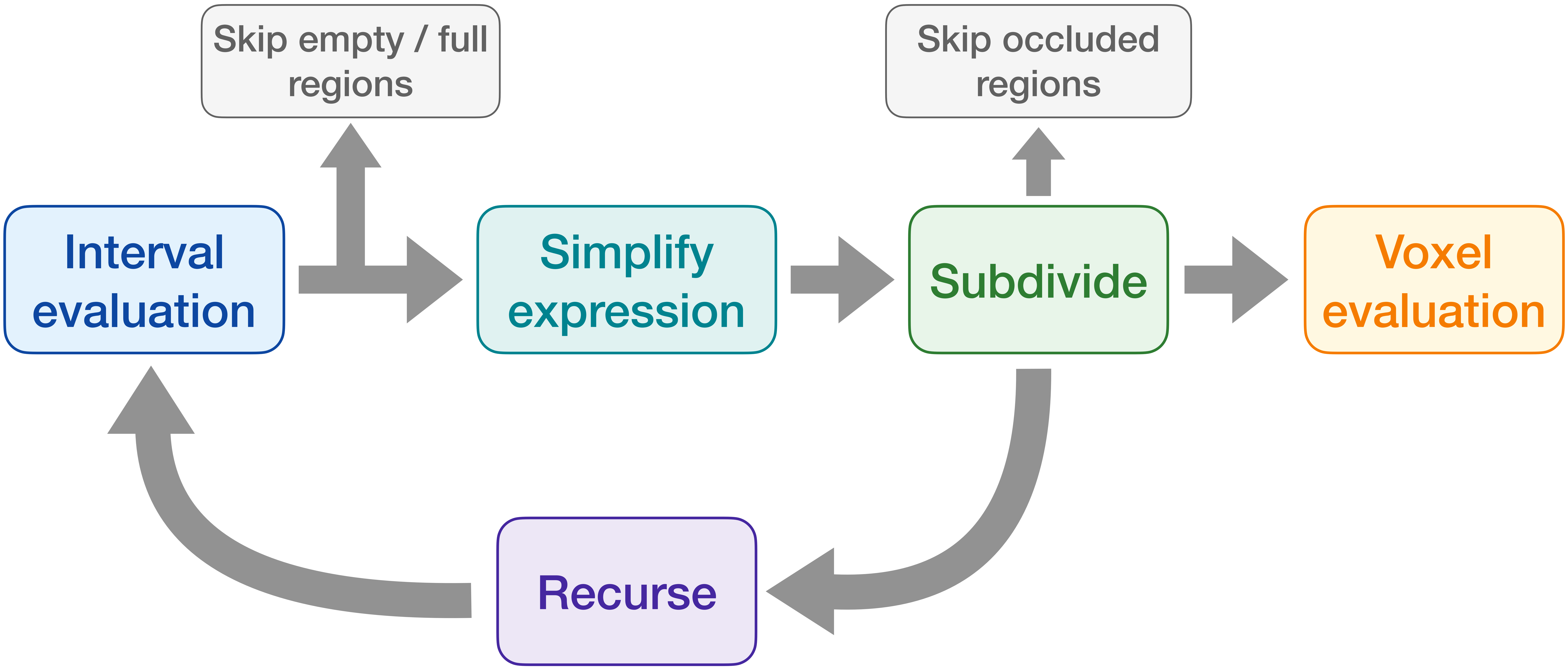


3D rendering

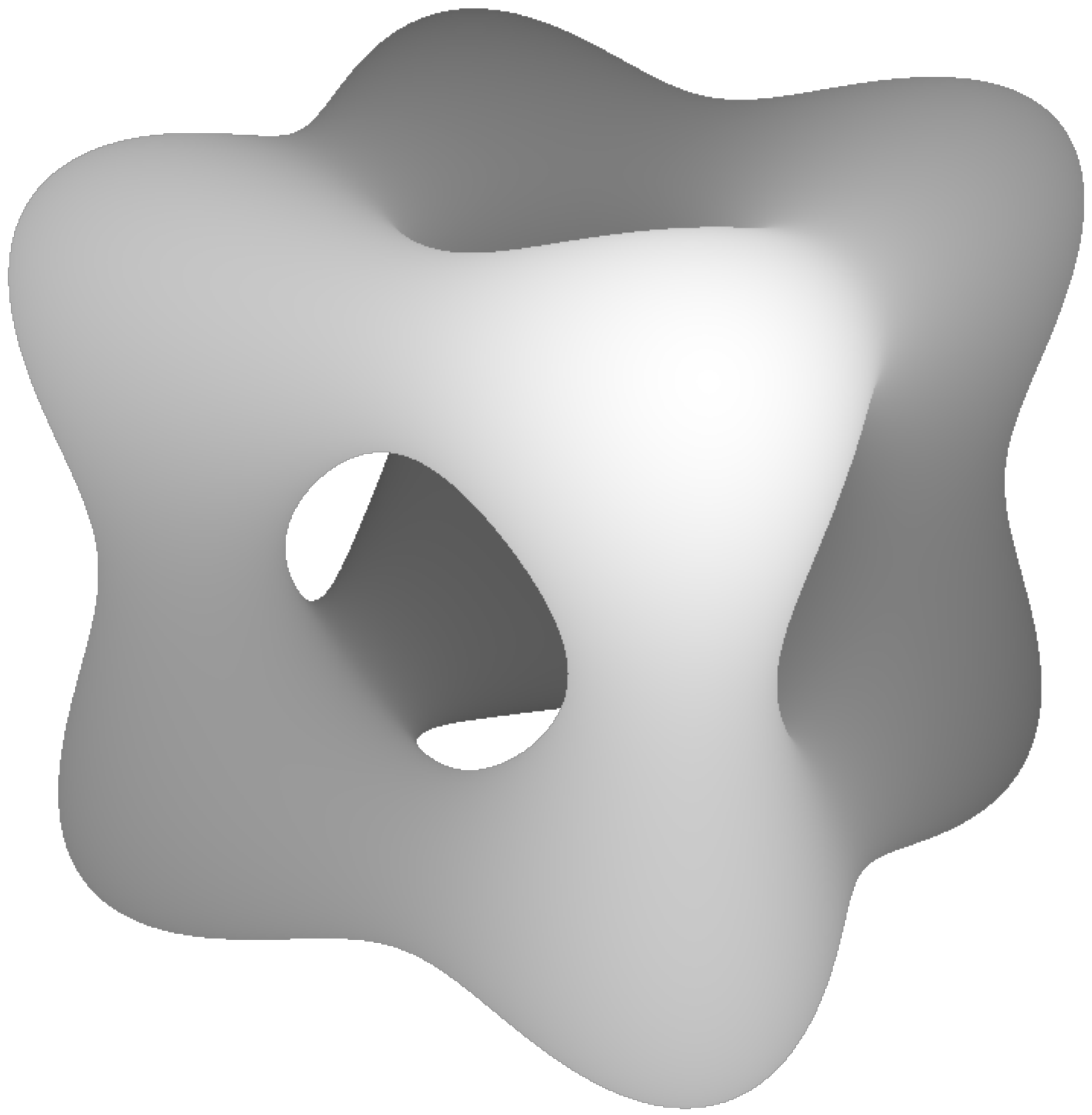
Side view



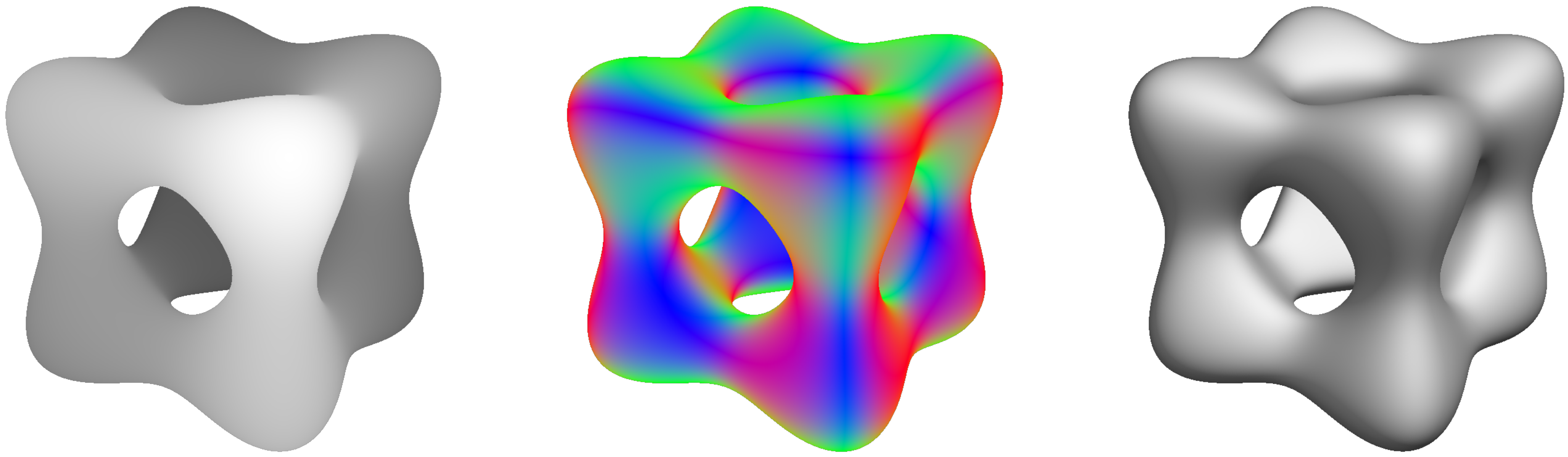
Modified render loop



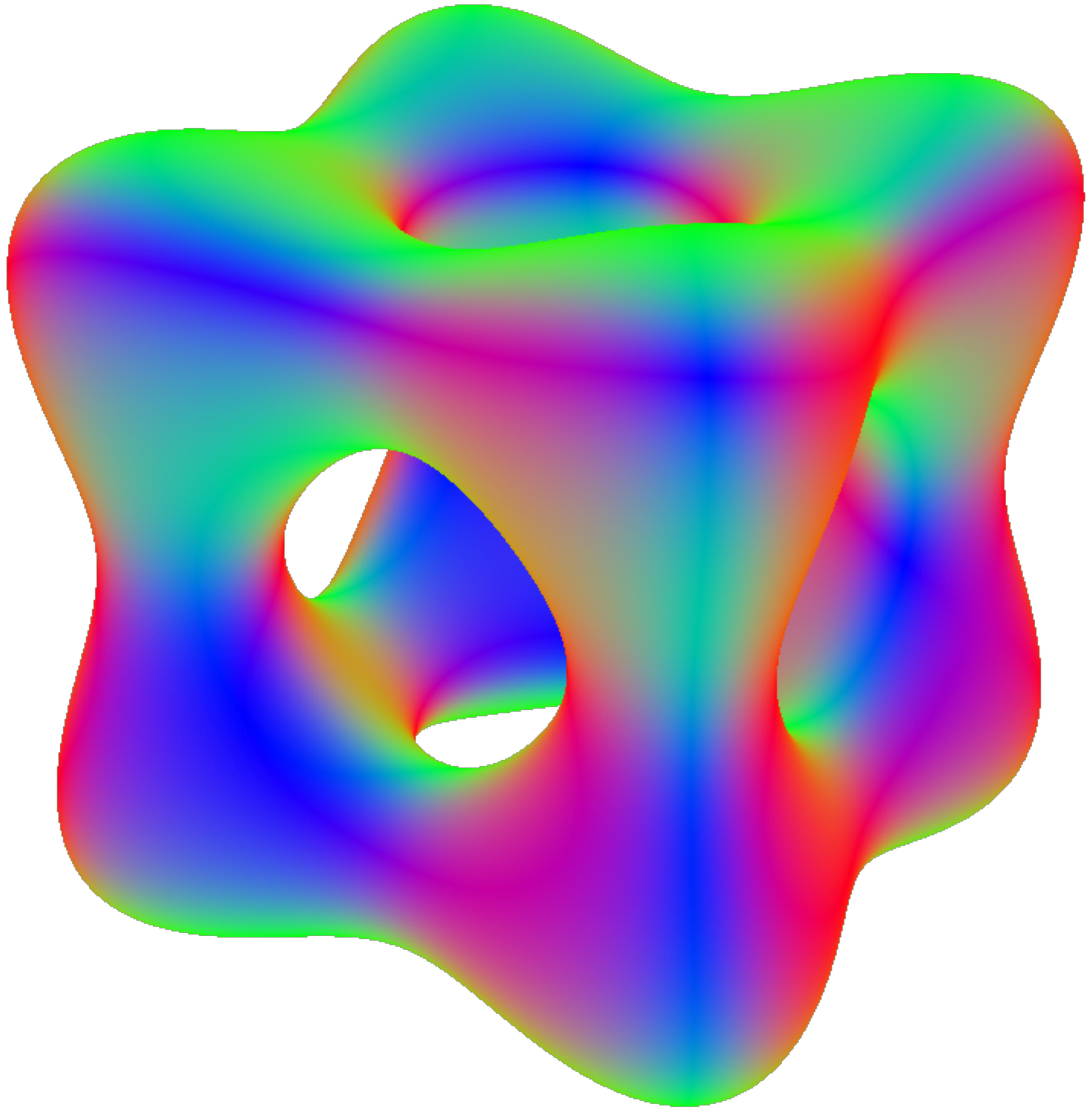
Heightmaps and shading



Heightmaps and shading



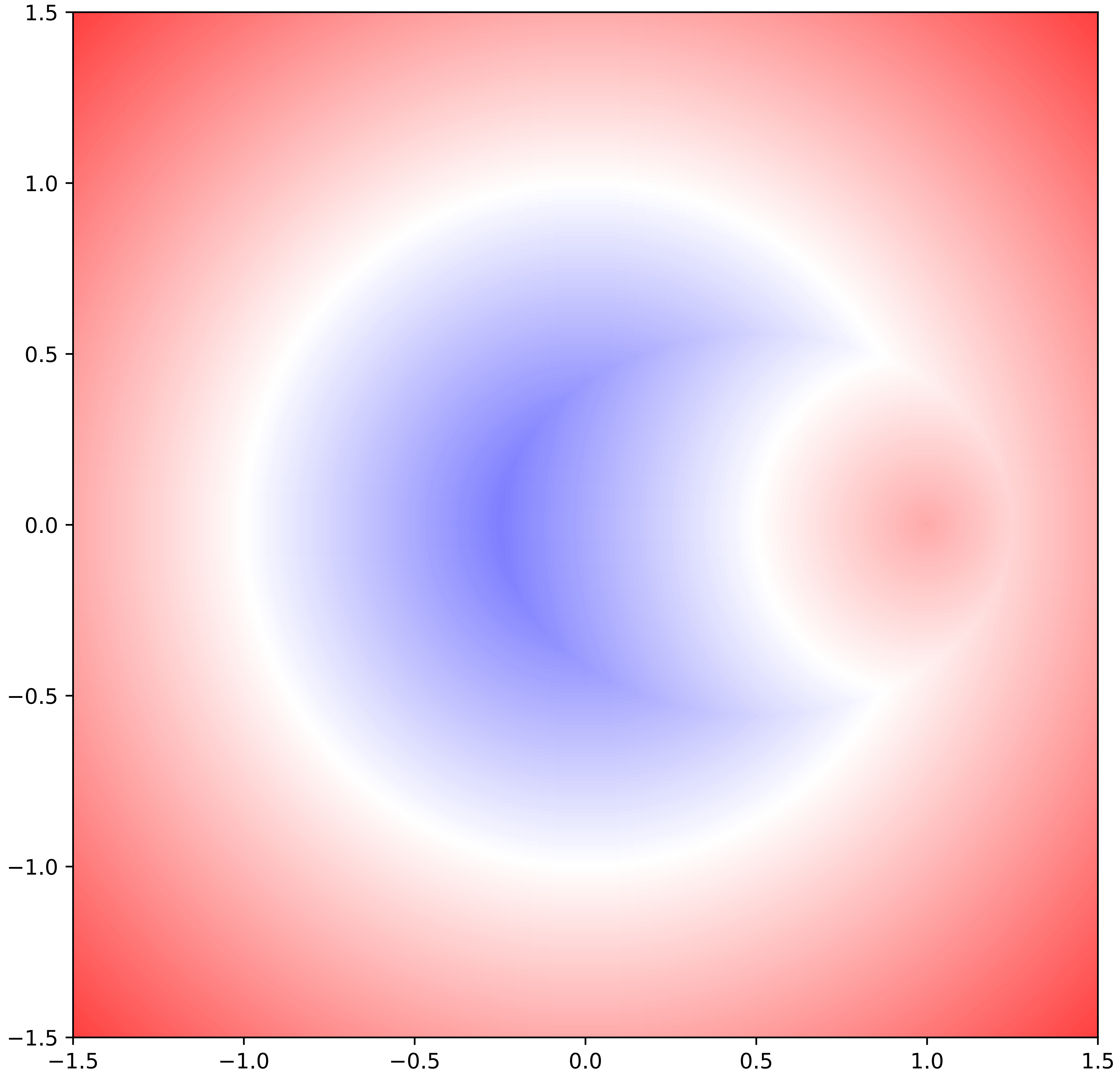
Finding surface normals



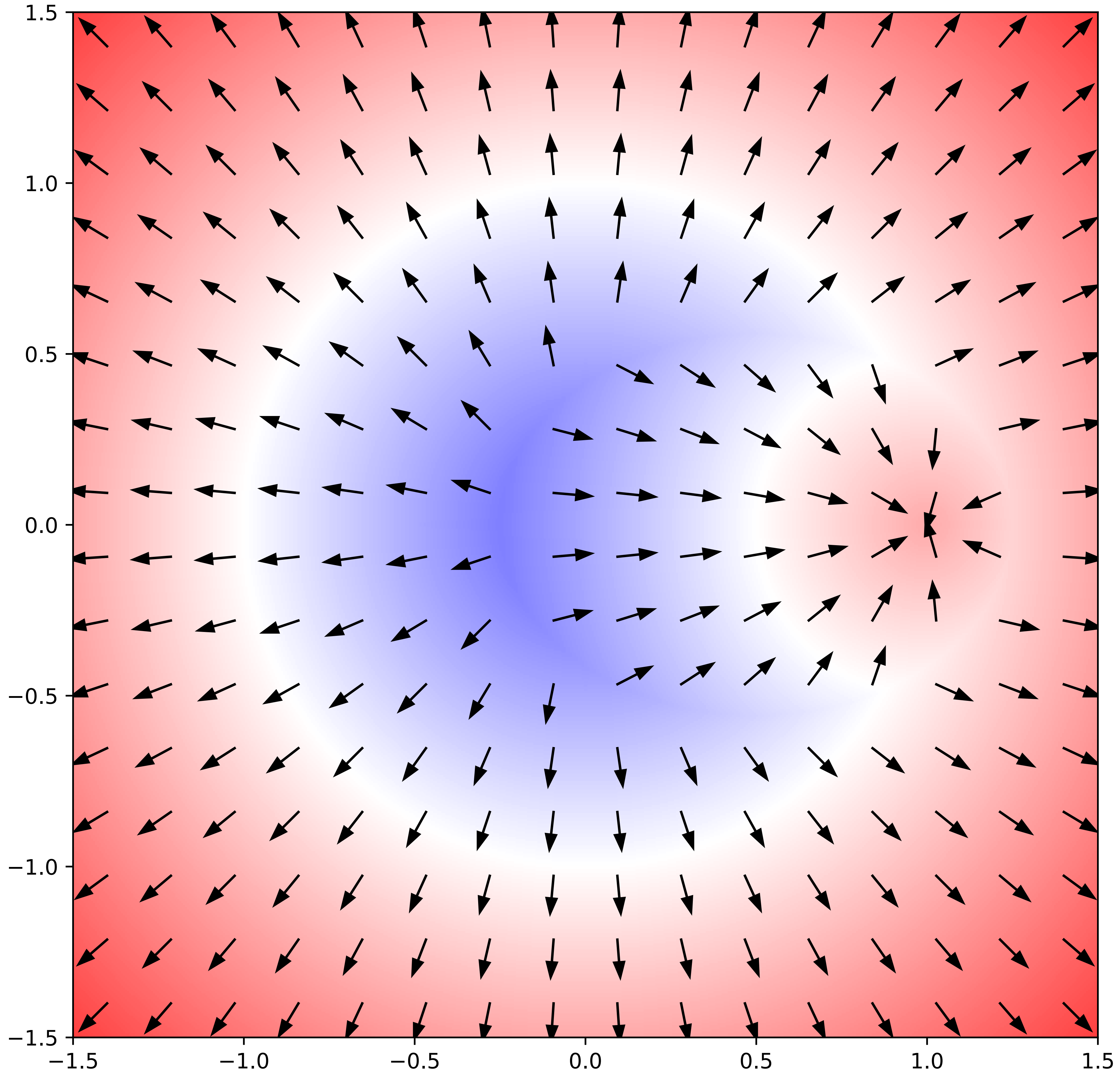
At the surface of the model, the normal is given by

$$\left(\frac{\partial f(x, y, z)}{\partial x}, \frac{\partial f(x, y, z)}{\partial y}, \frac{\partial f(x, y, z)}{\partial z}\right)$$

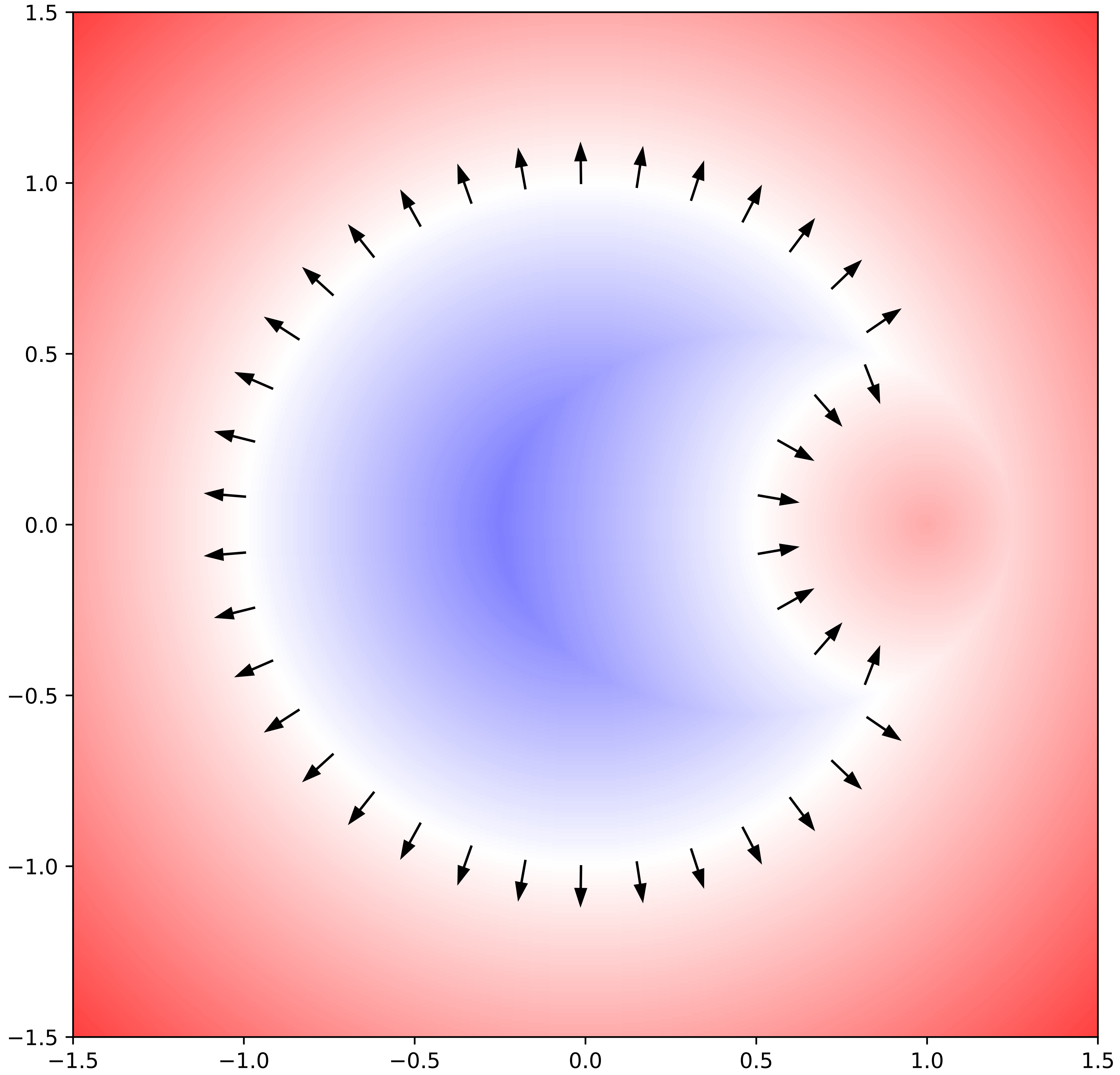
Partial derivatives in 2D



Partial derivatives in 2D



Partial derivatives in 2D



Gradient operator overloading

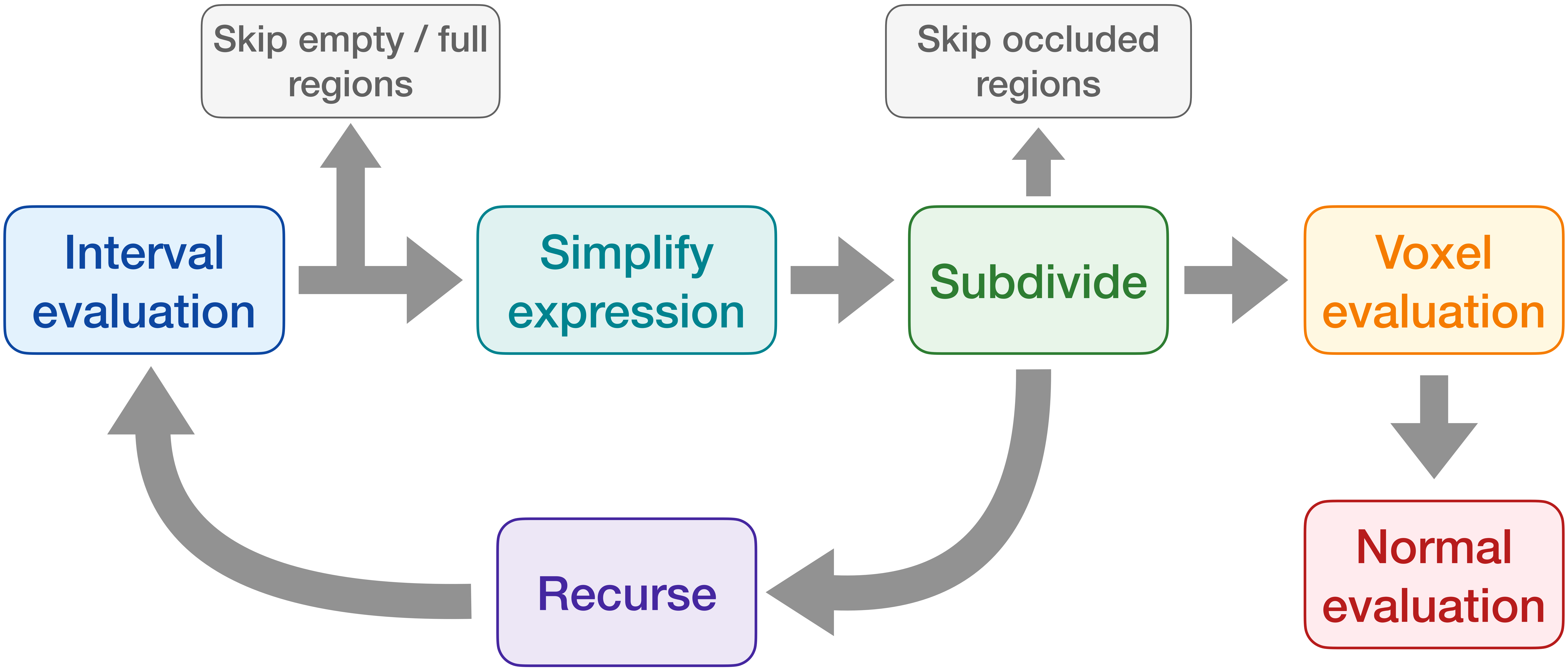
```
struct Grad {
    value: f32,
    dx: f32,
    dy: f32,
    dz: f32,
}
```

$$a = 1, \frac{\partial a}{\partial x} = 0.1$$

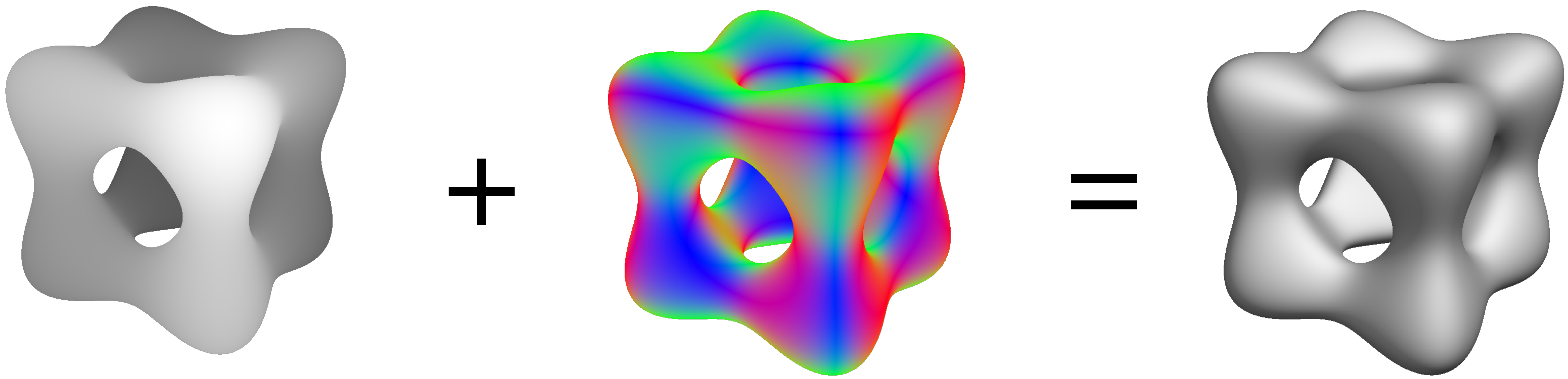
$$b = 3, \frac{\partial b}{\partial x} = 0.4$$

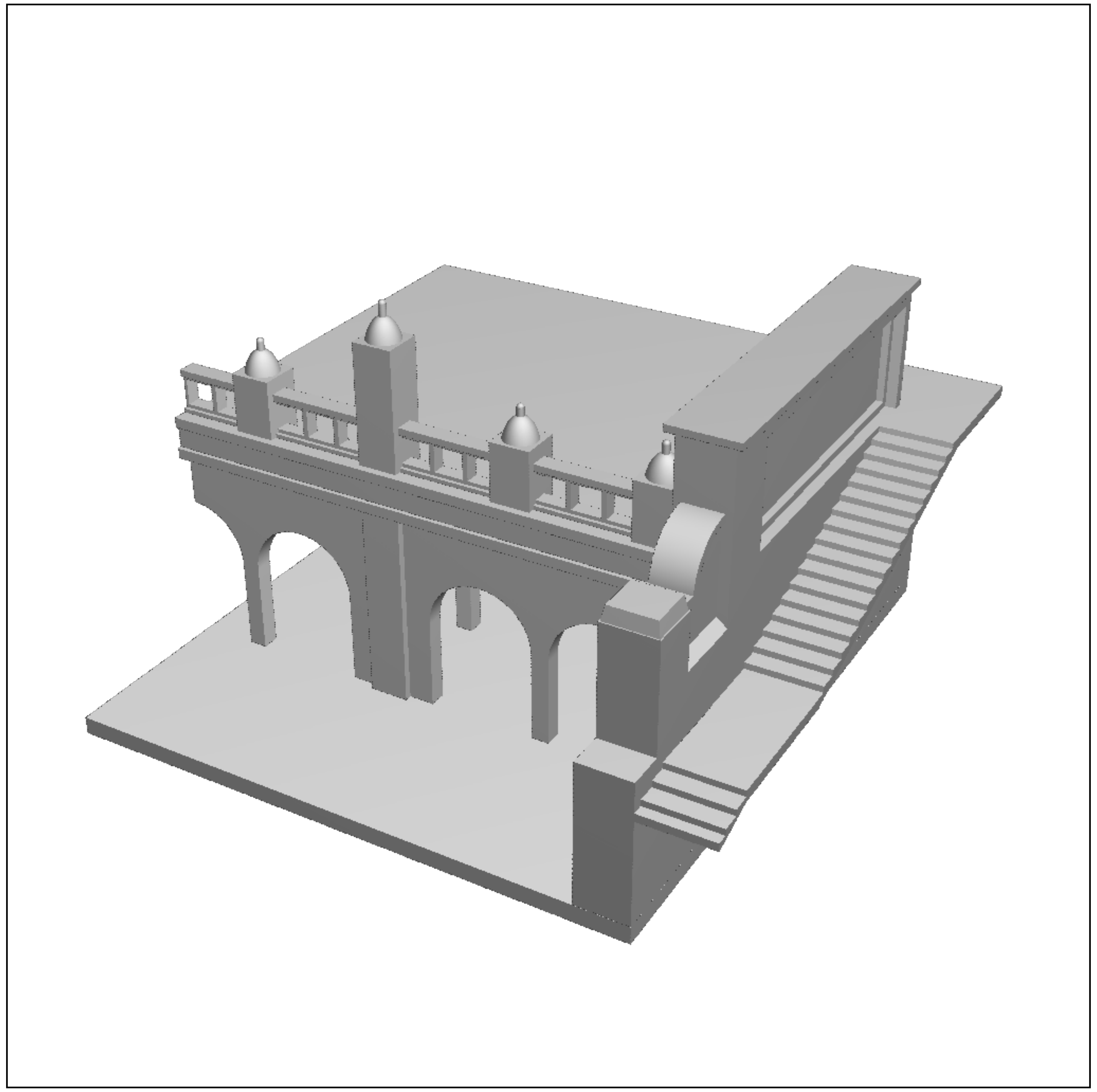
$$a + b = 4, \frac{\partial \left(a + b\right)}{\partial x} = 0.5$$

Modified render loop

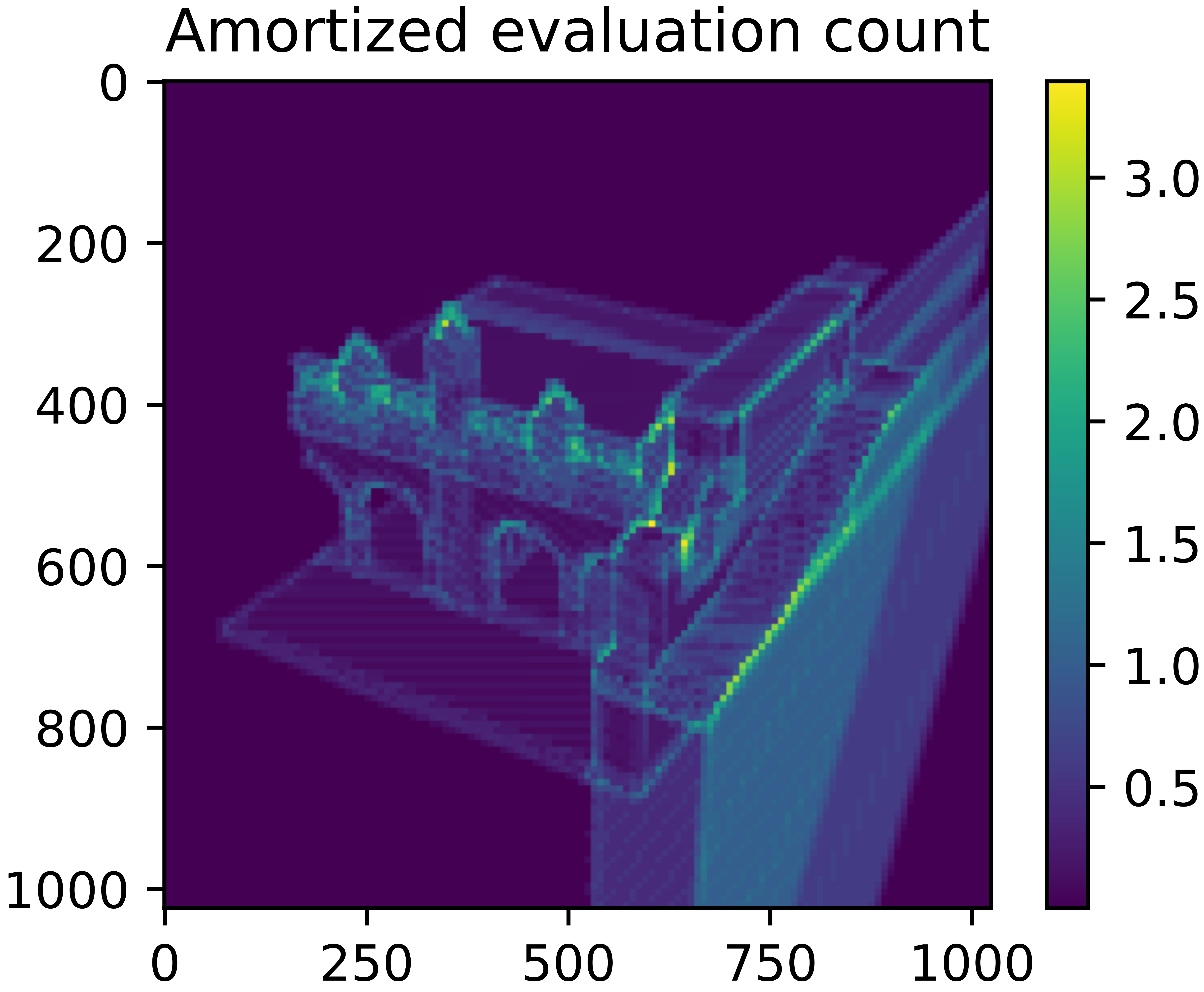


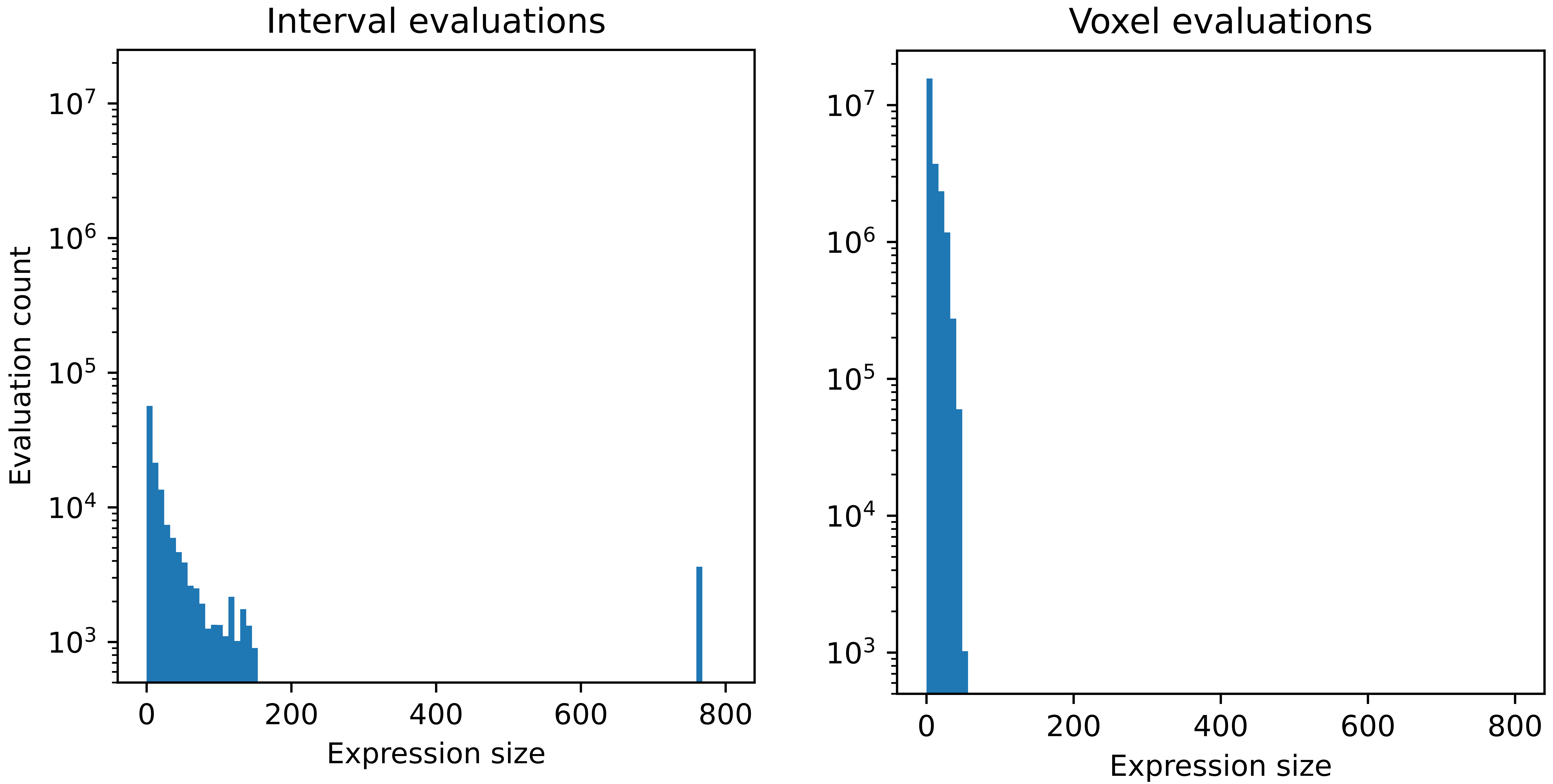
Deferred rendering





1024 × 1024 × 1024 render region

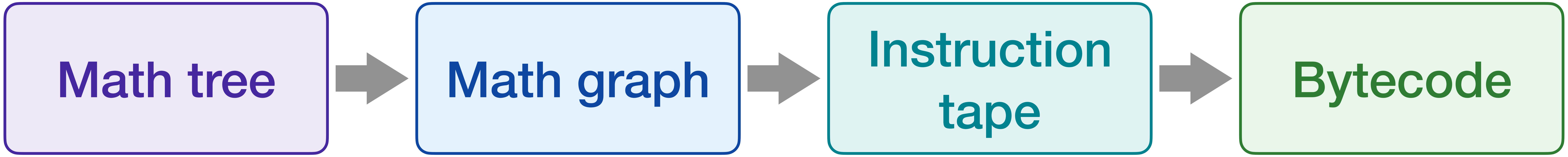




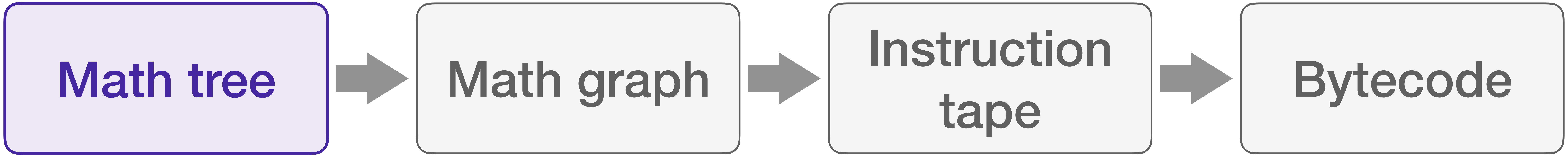
Interval evaluations
Voxel evaluations
Evaluation count
Expression size
Expression size
10^7
10^6
10^5
10^4
10^3
0
200
400
600
800

Fast evaluation of math trees

Our roadmap

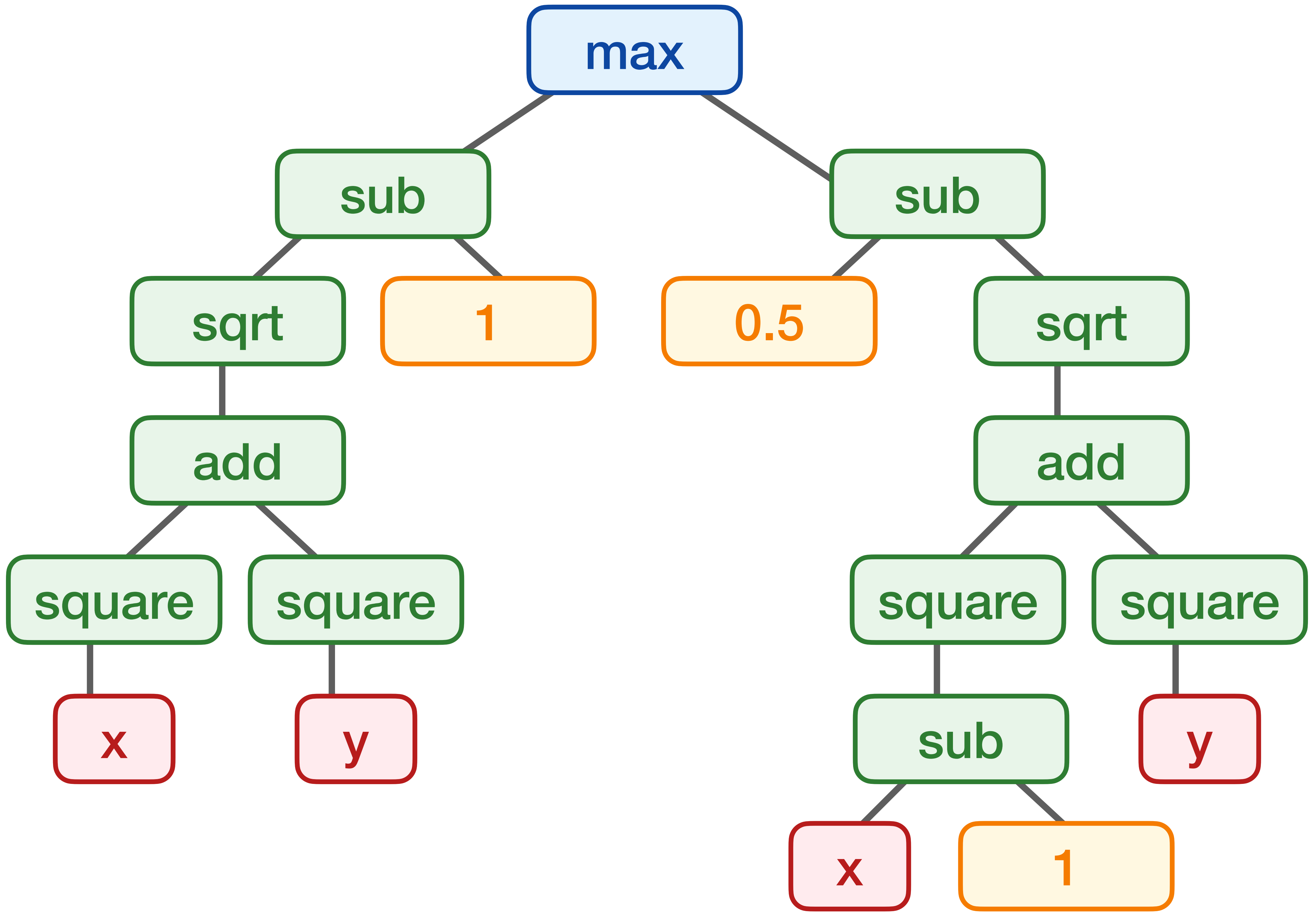


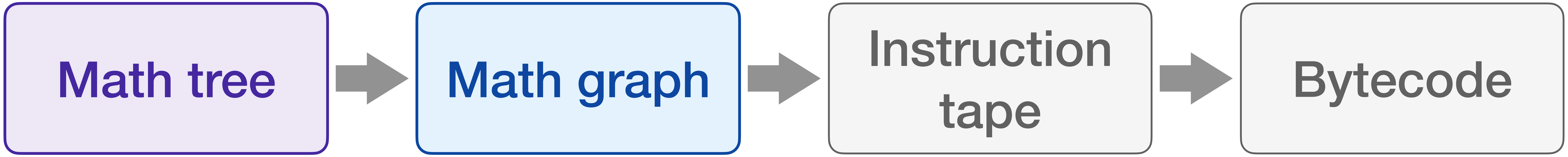
```
fn eval(ops, reg_count, vars) {
    regs = vec![0; reg_count]
    for (out, op) in ops {
        regs[out] = match op {
            Op::X => vars.x,
            Op::Y => vars.y,
            Op::Const(c) => c,
            Op::Sqrt(arg) => regs[arg].sqrt(),
            Op::Square(arg) => regs[arg].square(),
            Op::Sub(lhs, rhs) => regs[lhs] - regs[rhs],
            Op::Add(lhs, rhs) => regs[lhs] + regs[rhs],
            Op::Max(lhs, rhs) => max(regs[lhs], regs[rhs]),
        }
    }
    regs[0]
}
```


Math tree
Math graph
Instruction tape
Bytecode

Equation → Tree

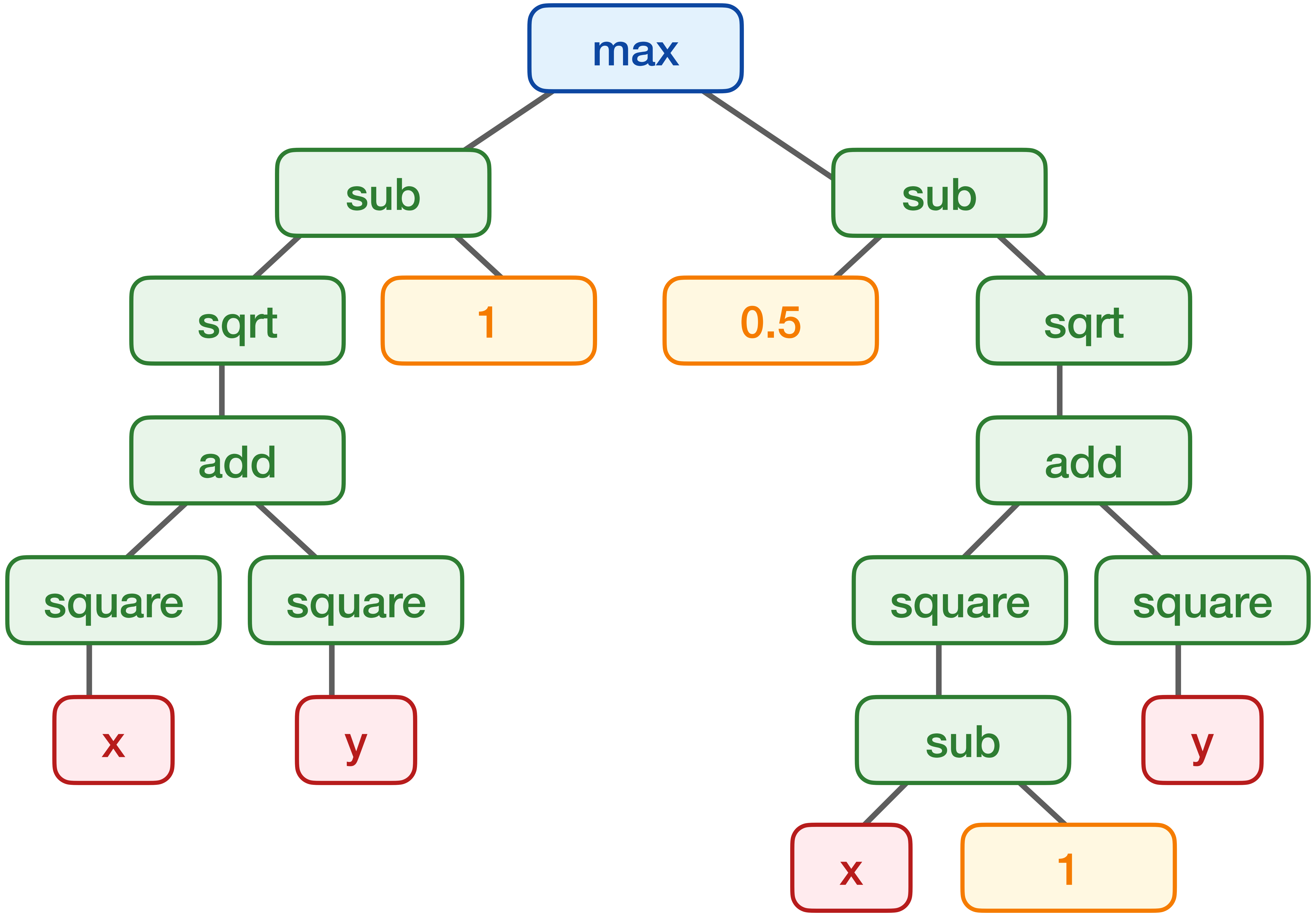
$$\max\left(\sqrt{x^2 + y^2} - 1,\ 0.5 - \sqrt{(x-1)^2 + y^2}\right)$$





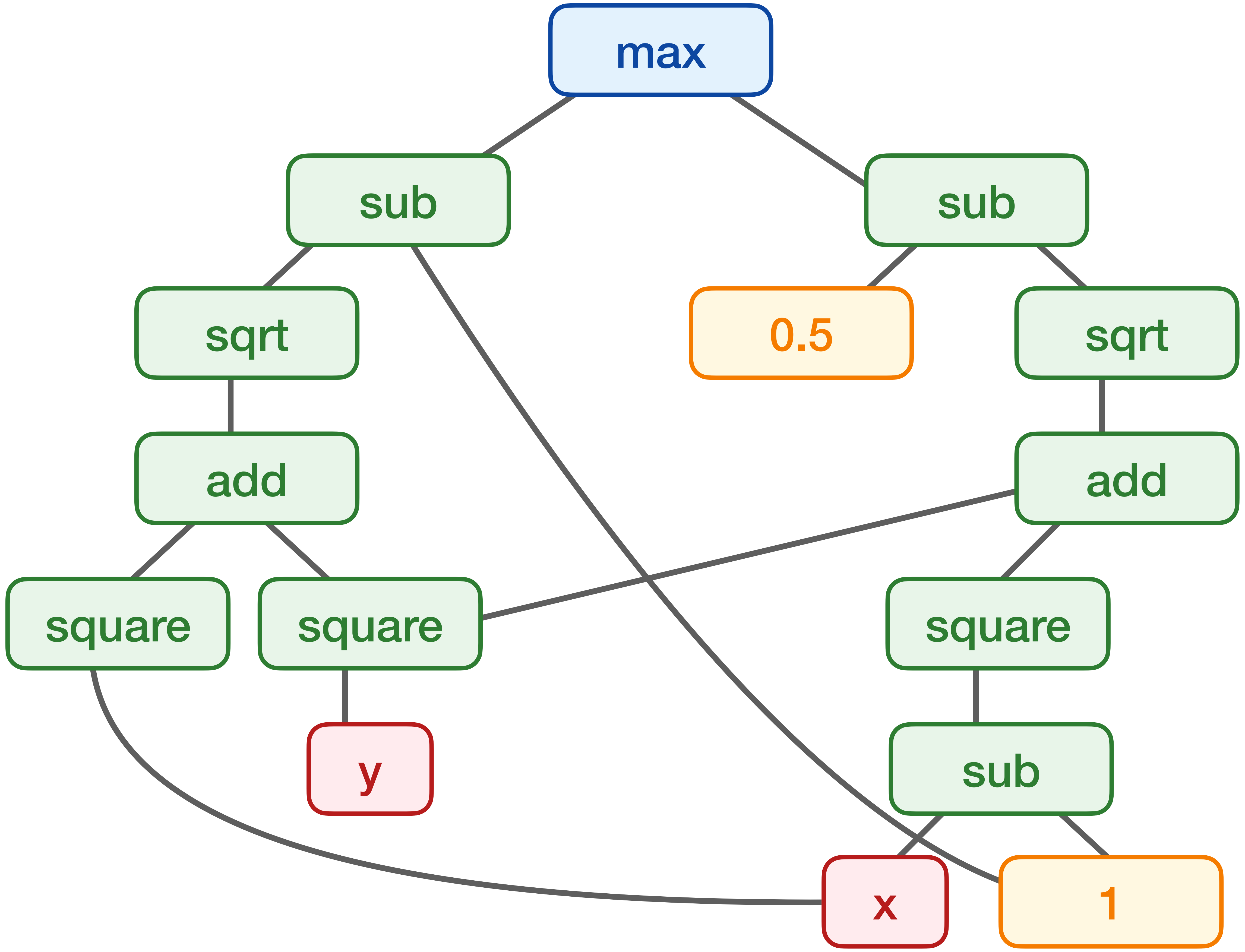
Deduplication

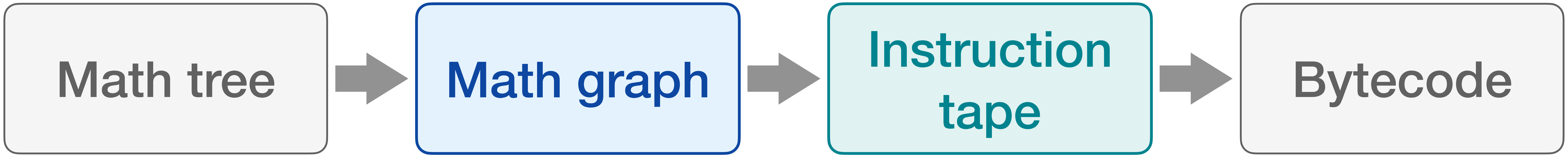
Tree → Graph



max
sub
sub
sqrt
1
0.5
sqrt
add
add
square
square
square
square
x
y
sub
y
x
1

Tree → Graph

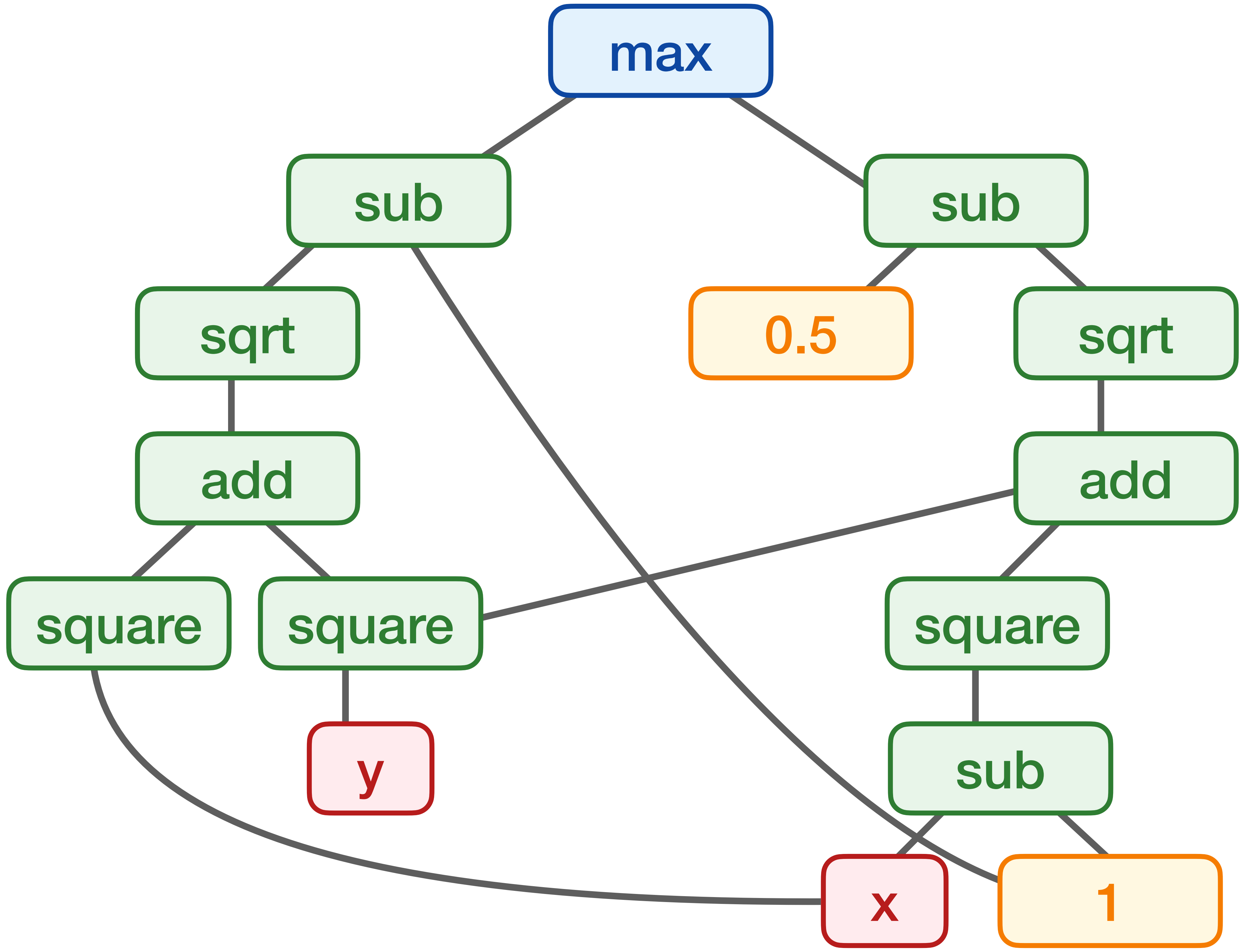




Flattening

Flattening the graph

"Postorder traversal"

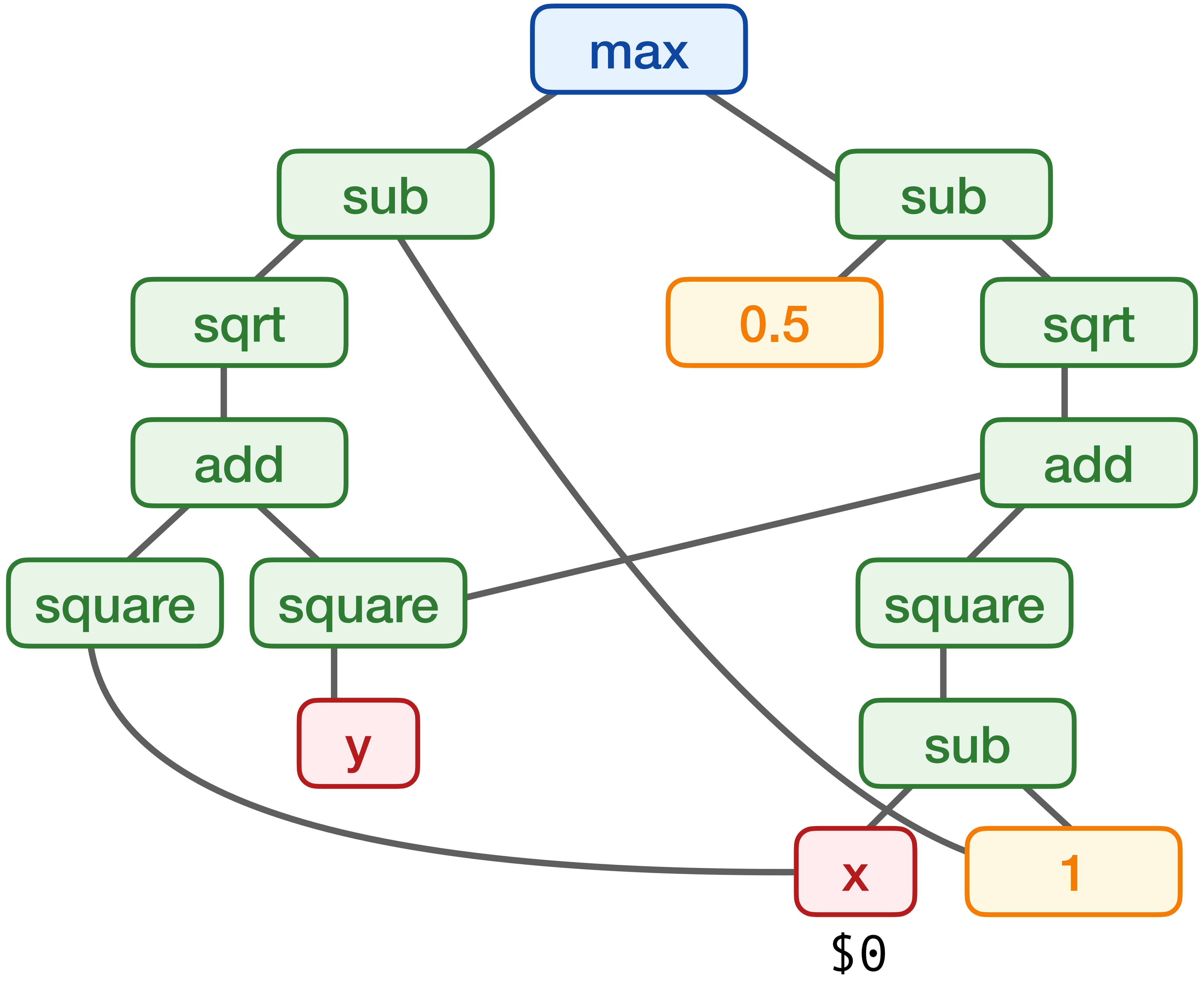


We want an instruction ordering such that arguments are defined before they are used

- DFS walk through the graph
- Emit a node once all of its arguments have been emitted

Flattening the graph

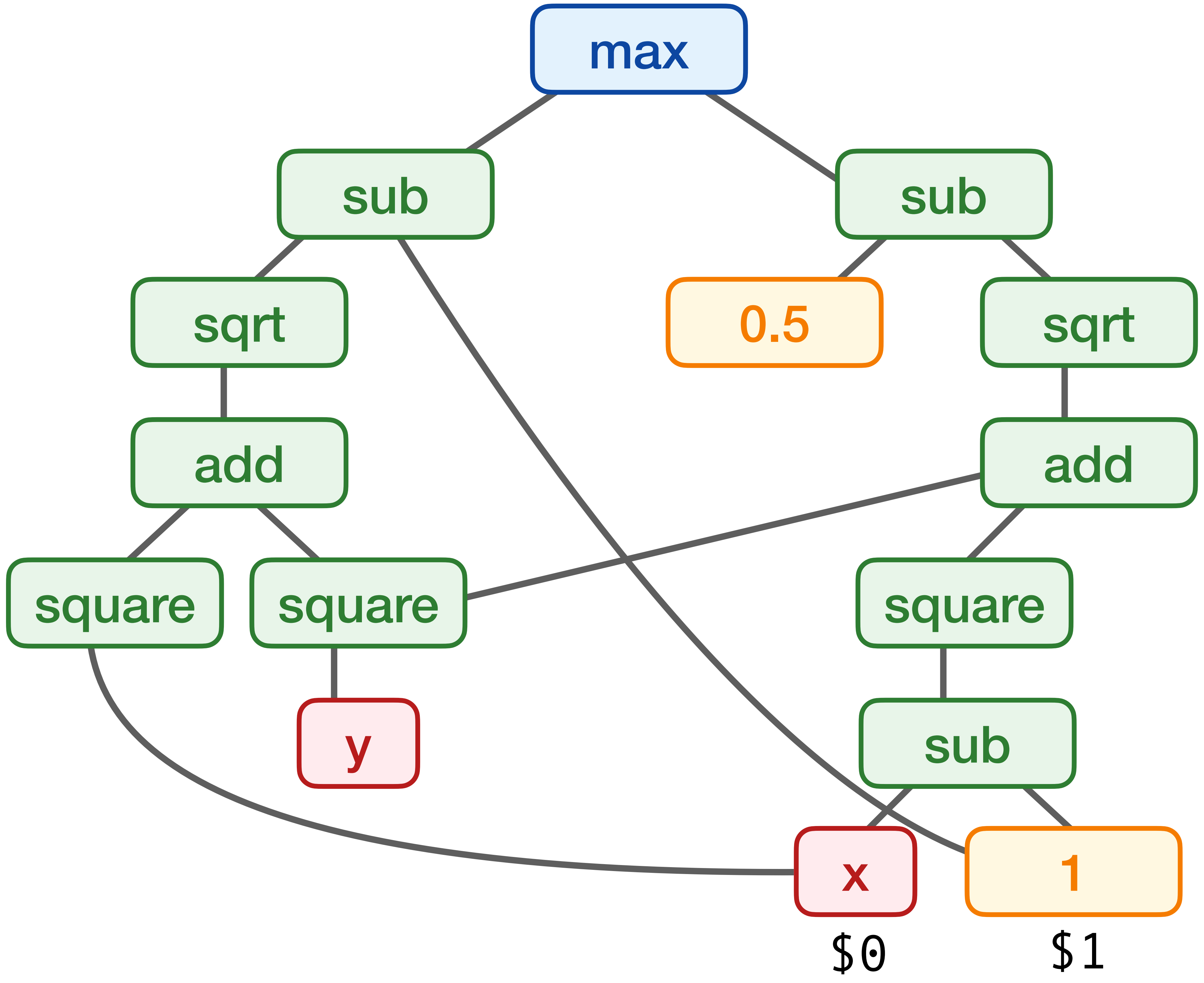
"Postorder traversal"




```
$0 = var-x
```

Flattening the graph

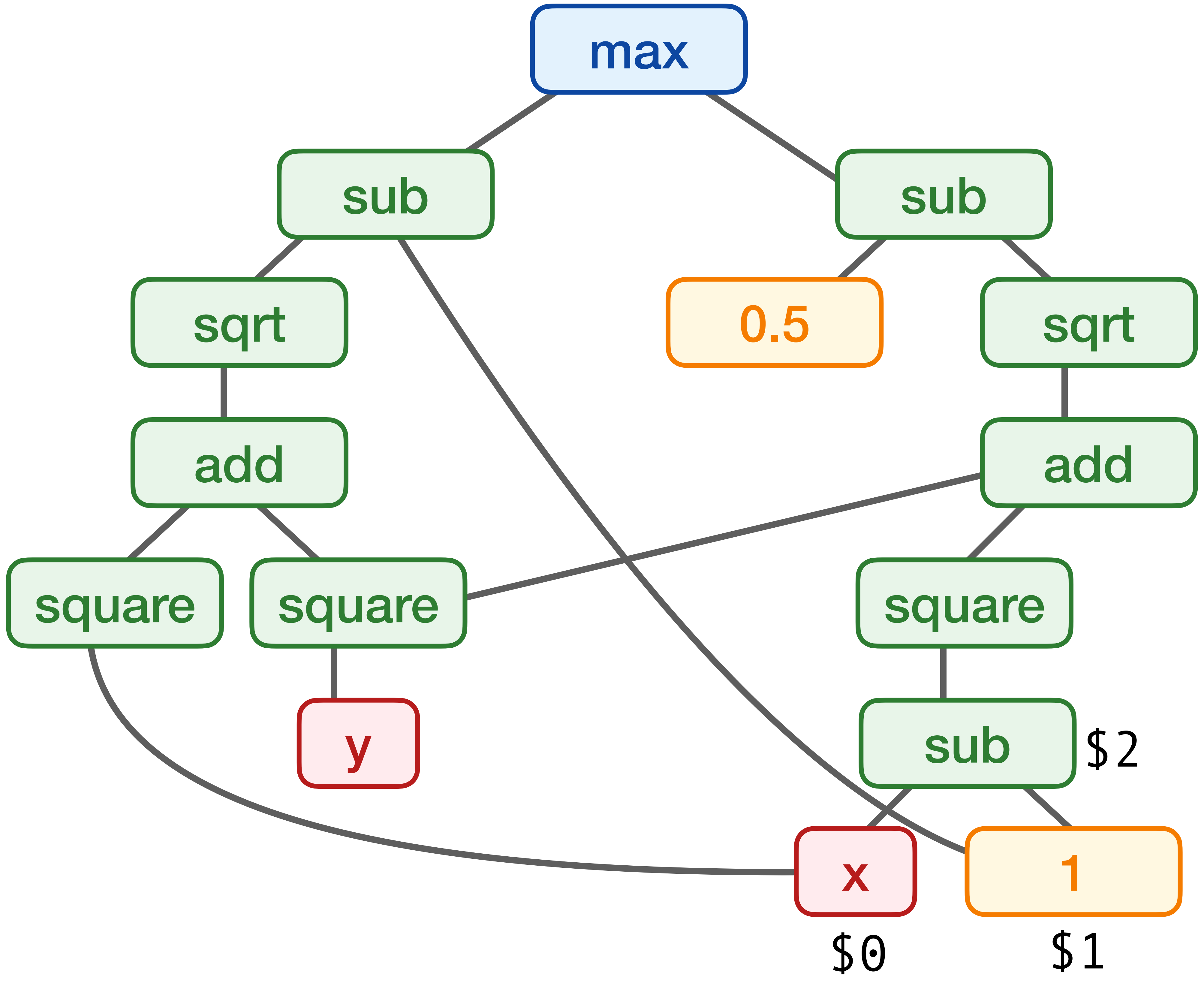
“Postorder traversal”




```
$0 = var-x
$1 = const 1
```

Flattening the graph

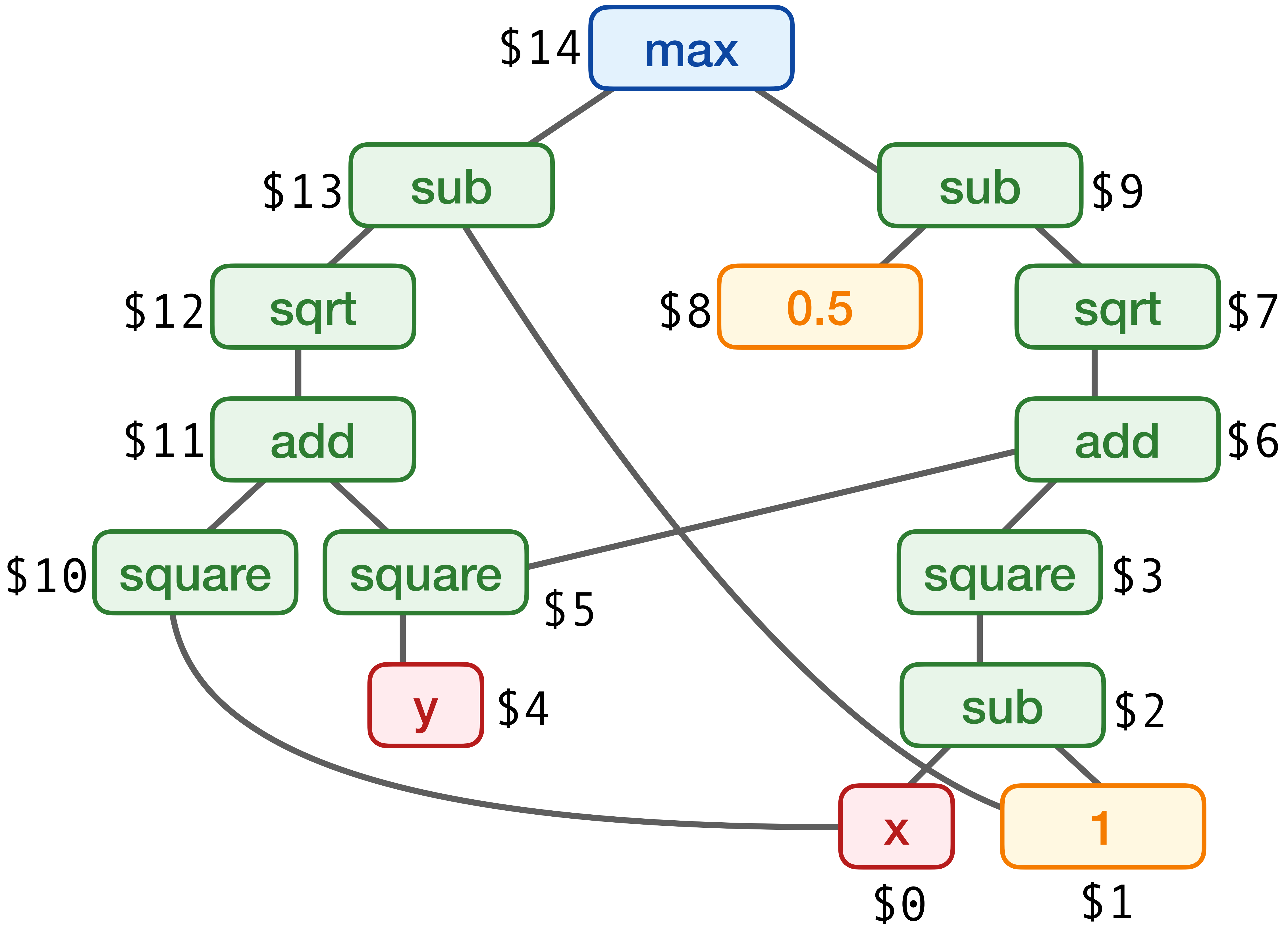
"Postorder traversal"



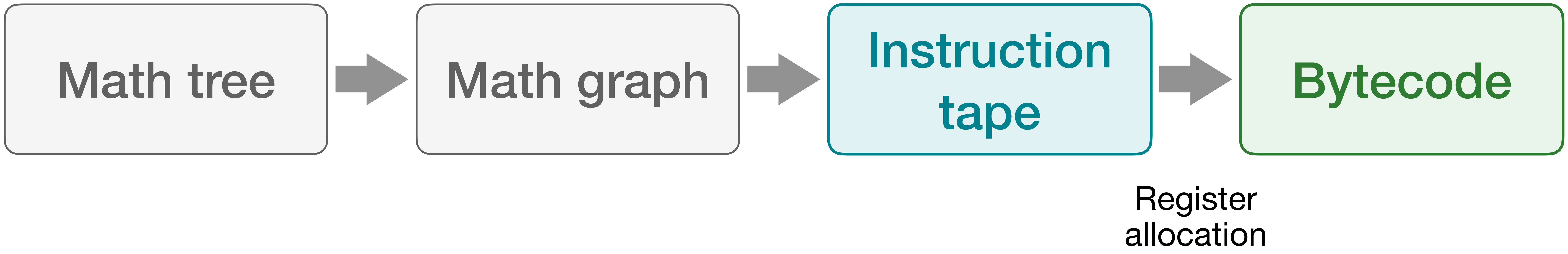

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
```

Flattening the graph

"Postorder traversal"




```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13
```


Math tree
Math graph
Instruction tape
Bytecode
Register allocation

Reducing memory usage

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13
```

Register allocation →

```
r1 = var-x
r2 = const 1
r4 = sub r1 r2
r4 = square r4
r3 = var-y
r3 = square r3
r4 = add r4 r5
r4 = sqrt r4
r0 = const 0.5
r0 = sub r0 r4
r1 = square r1
r1 = add r1 r3
r1 = sqrt r1
r1 = sub r1 r2
r0 = max r0 r1
```

Single static assignment form

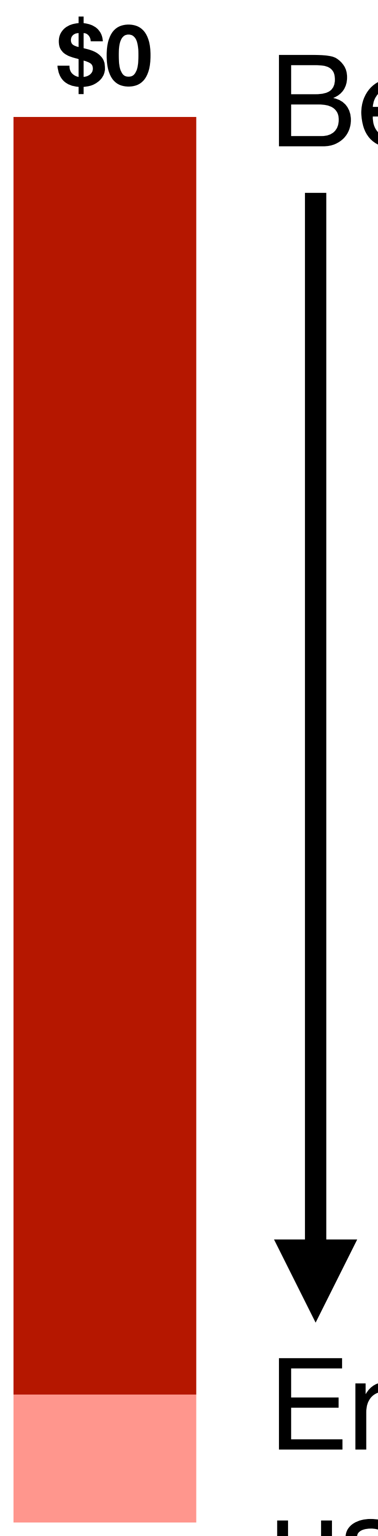
This is now a compilers talk!

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13
```

Register allocation

Liveness ranges

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13
```



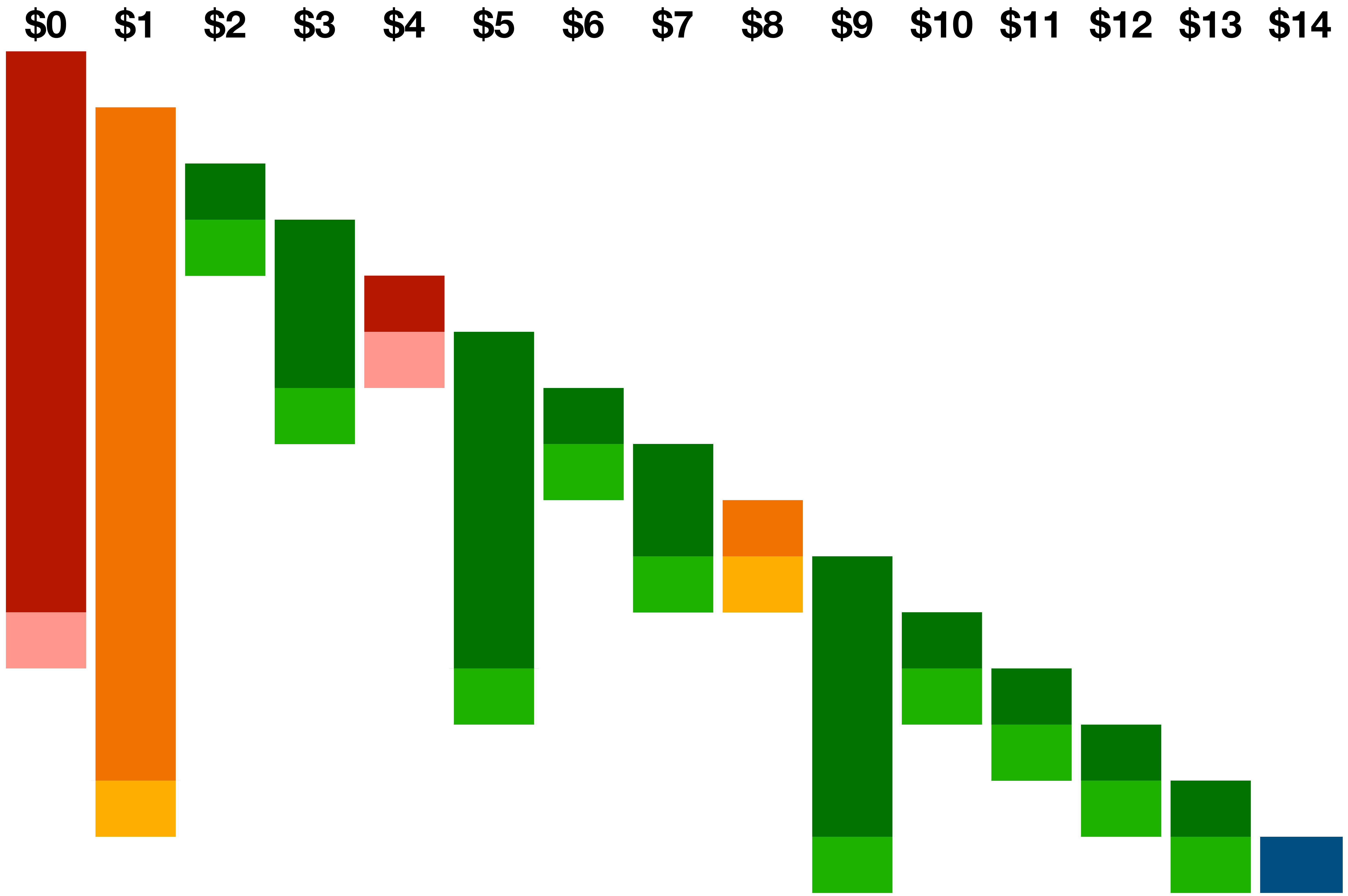
Begins when a value is written

Ends when the value is used for the last time

Register allocation

Liveness ranges

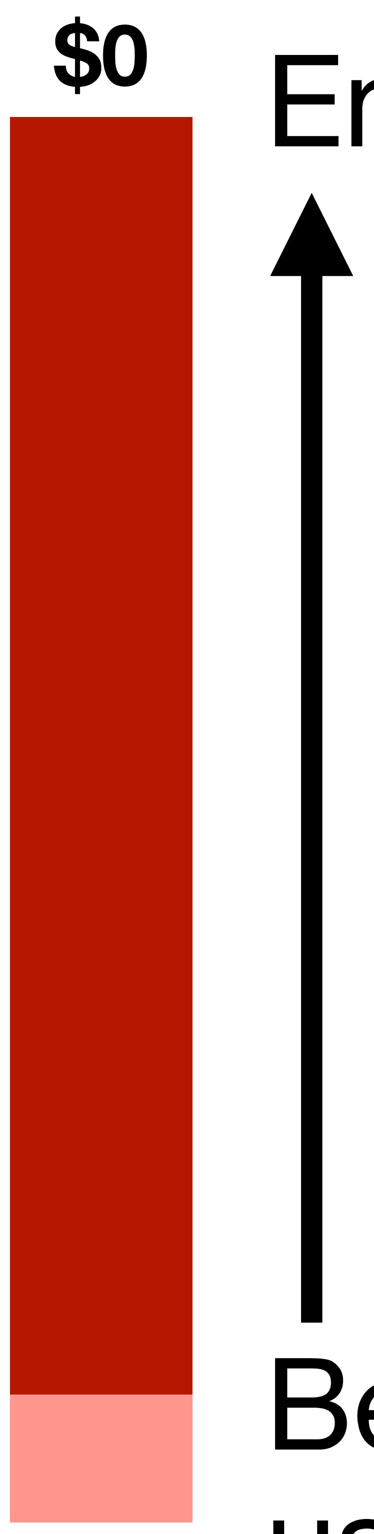
```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13
```



Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13
```



Ends when a value is written

Begins when the value is used for the first time

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13
```

- Maintain a value → register mapping
- Bind the output value to `r0`
- Walk through the instructions in reverse
  - When an value becomes live, bind it to an unused register
  - When a value is no longer live, release its register binding

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13     r0 = max r? r?
```

SSA value	Register
$14	r0
-	r1
-	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13      r0 = max r? r?
```

SSA value	Register
-	r0
-	r1
-	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13      r0 = max r0 r?
```

SSA value	Register
$9	r0
-	r1
-	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13     r0 = max r0 r1
```

SSA value	Register
$9	r0
$13	r1
-	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1     r1 = sub r? r?
$14 = max $9 $13     r0 = max r0 r1
```

SSA value	Register
$9	r0
$13	r1
-	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1     r1 = sub r? r?
$14 = max $9 $13     r0 = max r0 r1
```

SSA value	Register
$9	r0
-	r1
-	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1      r1 = sub r1 r?
$14 = max $9 $13      r0 = max r0 r1
```

SSA value	Register
$9	r0
$12	r1
-	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1      r1 = sub r1 r2
$14 = max $9 $13      r0 = max r0 r1
```

SSA value	Register
$9	r0
$12	r1
$1	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11        r1 = sqrt r?
$13 = sub $12 $1      r1 = sub r1 r2
$14 = max $9 $13      r0 = max r0 r1
```

SSA value	Register
$9	r0
$12	r1
$1	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11        r1 = sqrt r?
$13 = sub $12 $1      r1 = sub r1 r2
$14 = max $9 $13      r0 = max r0 r1
```

SSA value	Register
$9	r0
-	r1
$1	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11        r1 = sqrt r1
$13 = sub $12 $1      r1 = sub r1 r2
$14 = max $9 $13      r0 = max r0 r1
```

SSA value	Register
$9	r0
$11	r1
$1	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5      r1 = add r1 r3
$12 = sqrt $11        r1 = sqrt r1
$13 = sub $12 $1      r1 = sub r1 r2
$14 = max $9 $13      r0 = max r0 r1
```

SSA value	Register
$9	r0
$10	r1
$1	r2
$5	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0       r1 = square r1
$11 = add $10 $5      r1 = add r1 r3
$12 = sqrt $11        r1 = sqrt r1
$13 = sub $12 $1      r1 = sub r1 r2
$14 = max $9 $13      r0 = max r0 r1
```

SSA value	Register
$9	r0
$0	r1
$1	r2
$5	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7           r0 = sub r0 r4
$10 = square $0          r1 = square r1
$11 = add $10 $5         r1 = add r1 r3
$12 = sqrt $11           r1 = sqrt r1
$13 = sub $12 $1         r1 = sub r1 r2
$14 = max $9 $13         r0 = max r0 r1
```

SSA value	Register
$8	r0
$0	r1
$1	r2
$5	r3
$7	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5       r0 = const 0.5
$9 = sub $8 $7       r0 = sub r0 r4
$10 = square $0      r1 = square r1
$11 = add $10 $5     r1 = add r1 r3
$12 = sqrt $11       r1 = sqrt r1
$13 = sub $12 $1     r1 = sub r1 r2
$14 = max $9 $13     r0 = max r0 r1
```

SSA value	Register
-	r0
$0	r1
$1	r2
$5	r3
$7	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6            r4 = sqrt r4
$8 = const 0.5          r0 = const 0.5
$9 = sub $8 $7          r0 = sub r0 r4
$10 = square $0         r1 = square r1
$11 = add $10 $5        r1 = add r1 r3
$12 = sqrt $11          r1 = sqrt r1
$13 = sub $12 $1        r1 = sub r1 r2
$14 = max $9 $13        r0 = max r0 r1
```

SSA value	Register
-	r0
$0	r1
$1	r2
$5	r3
$6	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5          r4 = add r4 r5
$7 = sqrt $6            r4 = sqrt r4
$8 = const 0.5          r0 = const 0.5
$9 = sub $8 $7          r0 = sub r0 r4
$10 = square $0         r1 = square r1
$11 = add $10 $5        r1 = add r1 r3
$12 = sqrt $11          r1 = sqrt r1
$13 = sub $12 $1        r1 = sub r1 r2
$14 = max $9 $13        r0 = max r0 r1
```

SSA value	Register
-	r0
$0	r1
$1	r2
$5	r3
$3	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4          r3 = square r3
$6 = add $3 $5          r4 = add r4 r5
$7 = sqrt $6            r4 = sqrt r4
$8 = const 0.5          r0 = const 0.5
$9 = sub $8 $7          r0 = sub r0 r4
$10 = square $0         r1 = square r1
$11 = add $10 $5        r1 = add r1 r3
$12 = sqrt $11          r1 = sqrt r1
$13 = sub $12 $1        r1 = sub r1 r2
$14 = max $9 $13        r0 = max r0 r1
```

SSA value	Register
-	r0
$0	r1
$1	r2
$4	r3
$3	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y                r3 = var-y
$5 = square $4            r3 = square r3
$6 = add $3 $5            r4 = add r4 r5
$7 = sqrt $6              r4 = sqrt r4
$8 = const 0.5            r0 = const 0.5
$9 = sub $8 $7            r0 = sub r0 r4
$10 = square $0           r1 = square r1
$11 = add $10 $5          r1 = add r1 r3
$12 = sqrt $11            r1 = sqrt r1
$13 = sub $12 $1          r1 = sub r1 r2
$14 = max $9 $13          r0 = max r0 r1
```

SSA value	Register
-	r0
$0	r1
$1	r2
-	r3
$3	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2         r4 = square r4
$4 = var-y             r3 = var-y
$5 = square $4         r3 = square r3
$6 = add $3 $5         r4 = add r4 r5
$7 = sqrt $6           r4 = sqrt r4
$8 = const 0.5         r0 = const 0.5
$9 = sub $8 $7         r0 = sub r0 r4
$10 = square $0        r1 = square r1
$11 = add $10 $5       r1 = add r1 r3
$12 = sqrt $11         r1 = sqrt r1
$13 = sub $12 $1       r1 = sub r1 r2
$14 = max $9 $13       r0 = max r0 r1
```

SSA value	Register
-	r0
$0	r1
$1	r2
-	r3
$2	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1        r4 = sub r1 r2
$3 = square $2        r4 = square r4
$4 = var-y            r3 = var-y
$5 = square $4        r3 = square r3
$6 = add $3 $5        r4 = add r4 r5
$7 = sqrt $6          r4 = sqrt r4
$8 = const 0.5        r0 = const 0.5
$9 = sub $8 $7        r0 = sub r0 r4
$10 = square $0       r1 = square r1
$11 = add $10 $5      r1 = add r1 r3
$12 = sqrt $11        r1 = sqrt r1
$13 = sub $12 $1      r1 = sub r1 r2
$14 = max $9 $13      r0 = max r0 r1
```

SSA value	Register
-	r0
$0	r1
$1	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x
$1 = const 1          r2 = const 1
$2 = sub $0 $1        r4 = sub r1 r2
$3 = square $2        r4 = square r4
$4 = var-y            r3 = var-y
$5 = square $4        r3 = square r3
$6 = add $3 $5        r4 = add r4 r5
$7 = sqrt $6          r4 = sqrt r4
$8 = const 0.5        r0 = const 0.5
$9 = sub $8 $7        r0 = sub r0 r4
$10 = square $0       r1 = square r1
$11 = add $10 $5      r1 = add r1 r3
$12 = sqrt $11        r1 = sqrt r1
$13 = sub $12 $1      r1 = sub r1 r2
$14 = max $9 $13      r0 = max r0 r1
```

SSA value	Register
-	r0
$0	r1
-	r2
-	r3
-	r4

Register allocation

Reverse Linear Scan

```
$0 = var-x               r1 = var-x
$1 = const 1             r2 = const 1
$2 = sub $0 $1           r4 = sub r1 r2
$3 = square $2           r4 = square r4
$4 = var-y               r3 = var-y
$5 = square $4           r3 = square r3
$6 = add $3 $5           r4 = add r4 r5
$7 = sqrt $6             r4 = sqrt r4
$8 = const 0.5           r0 = const 0.5
$9 = sub $8 $7           r0 = sub r0 r4
$10 = square $0          r1 = square r1
$11 = add $10 $5         r1 = add r1 r3
$12 = sqrt $11           r1 = sqrt r1
$13 = sub $12 $1         r1 = sub r1 r2
$14 = max $9 $13         r0 = max r0 r1
```

SSA value	Register
-	r0
-	r1
-	r2
-	r3
-	r4

Register allocation

Liveness ranges + simplification

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13
```

A value's liveness range begins when the value is used for the first time

If `min` or `max` can be simplified, then one or the other argument is **not used**

Register allocation

Liveness ranges + simplification

```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13
```

A value's liveness range begins when the value is used for the first time

If `min` or `max` can be simplified, then one or the other argument is **not used**

Register allocation

Reverse linear scan with simplification

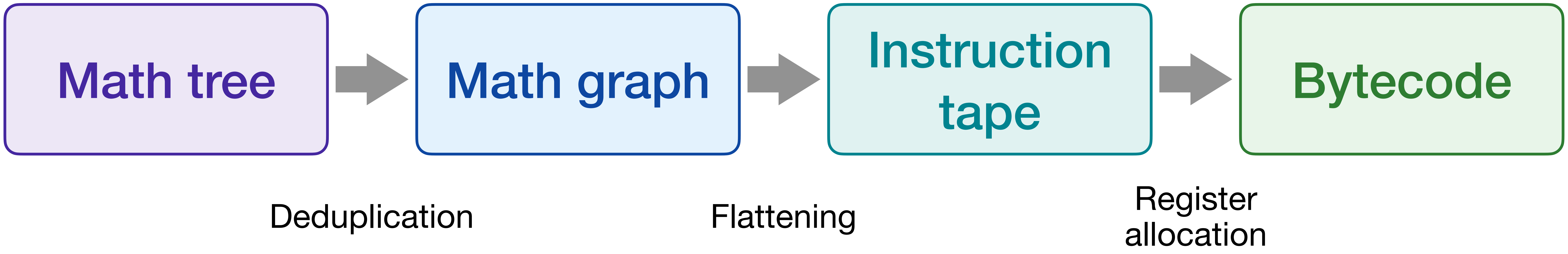
```
$0 = var-x
$1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y
$5 = square $4
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0
$11 = add $10 $5
$12 = sqrt $11
$13 = sub $12 $1
$14 = max $9 $13
```

- Maintain a value → register mapping
- Bind the output value to `r0`
- Walk through the instructions in reverse
  - **If the instruction's output value is not live, then skip it!**
  - When an value becomes live, bind it to an unused register
  - When a value is no longer live, release its register binding

Register allocation

Reverse linear scan with simplification

```
$0 = var-x              r0 = var-x
$1 = const 1            r1 = const 1
$2 = sub $0 $1
$3 = square $2
$4 = var-y              r2 = var-y
$5 = square $4          r2 = square r2
$6 = add $3 $5
$7 = sqrt $6
$8 = const 0.5
$9 = sub $8 $7
$10 = square $0         r0 = square r0
$11 = add $10 $5        r0 = add r0 r2
$12 = sqrt $11          r0 = sqrt r0
$13 = sub $12 $1        r0 = sub r0 r1
$14 = max $9 $13        r0 = copy r0
```


Math tree
Math graph
Instruction tape
Bytecode
Deduplication
Flattening
Register allocation

Bytecode interpreter

```
r1 = var-x
r2 = const 1
r4 = sub r1 r2
r4 = square r4
r3 = var-y
r3 = square r3
r4 = add r4 r5
r4 = sqrt r4
r0 = const 0.5
r0 = sub r0 r4
r1 = square r1
r1 = add r1 r3
r1 = sqrt r1
r1 = sub r1 r2
r0 = max r0 r1
```

```
fn eval(ops, reg_count, vars) {
  regs = vec![0; reg_count]
  for (out, op) in ops {
    regs[out] = match op {
      Op::X => vars.x,
      Op::Y => vars.y,
      Op::Const(c) => c,
      Op::Sqrt(arg) => regs[arg].sqrt(),
      Op::Square(arg) => regs[arg].square(),
      Op::Sub(lhs, rhs) => regs[lhs] - regs[rhs],
      Op::Add(lhs, rhs) => regs[lhs] + regs[rhs],
      Op::Max(lhs, rhs) => max(regs[lhs], regs[rhs]),
    }
  }
  regs[0]
}
```

Bytecode interpreter overhead

```
fn eval(ops, reg_count, vars) {
  regs = vec![0; reg_count]
  for (out, op) in ops {
    regs[out] = match op {
      Op::X => vars.x,
      Op::Y => vars.y,
      Op::Const(c) => c,
      Op::Sqrt(arg) => regs[arg].sqrt(),
      Op::Square(arg) => regs[arg].square(),
      Op::Sub(lhs, rhs) => regs[lhs] - regs[rhs],
      Op::Add(lhs, rhs) => regs[lhs] + regs[rhs],
      Op::Max(lhs, rhs) => max(regs[lhs], regs[rhs]),
    }
  }
  regs[0]
}
```

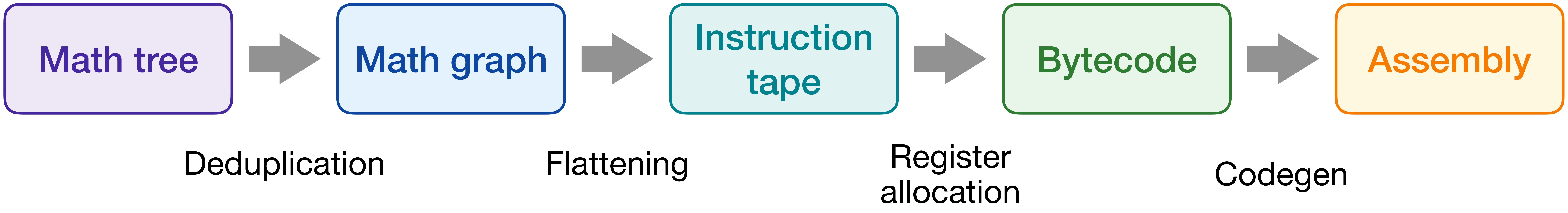
Unpredictable branch

Lots of reading and writing to RAM

Bytecode → Assembly

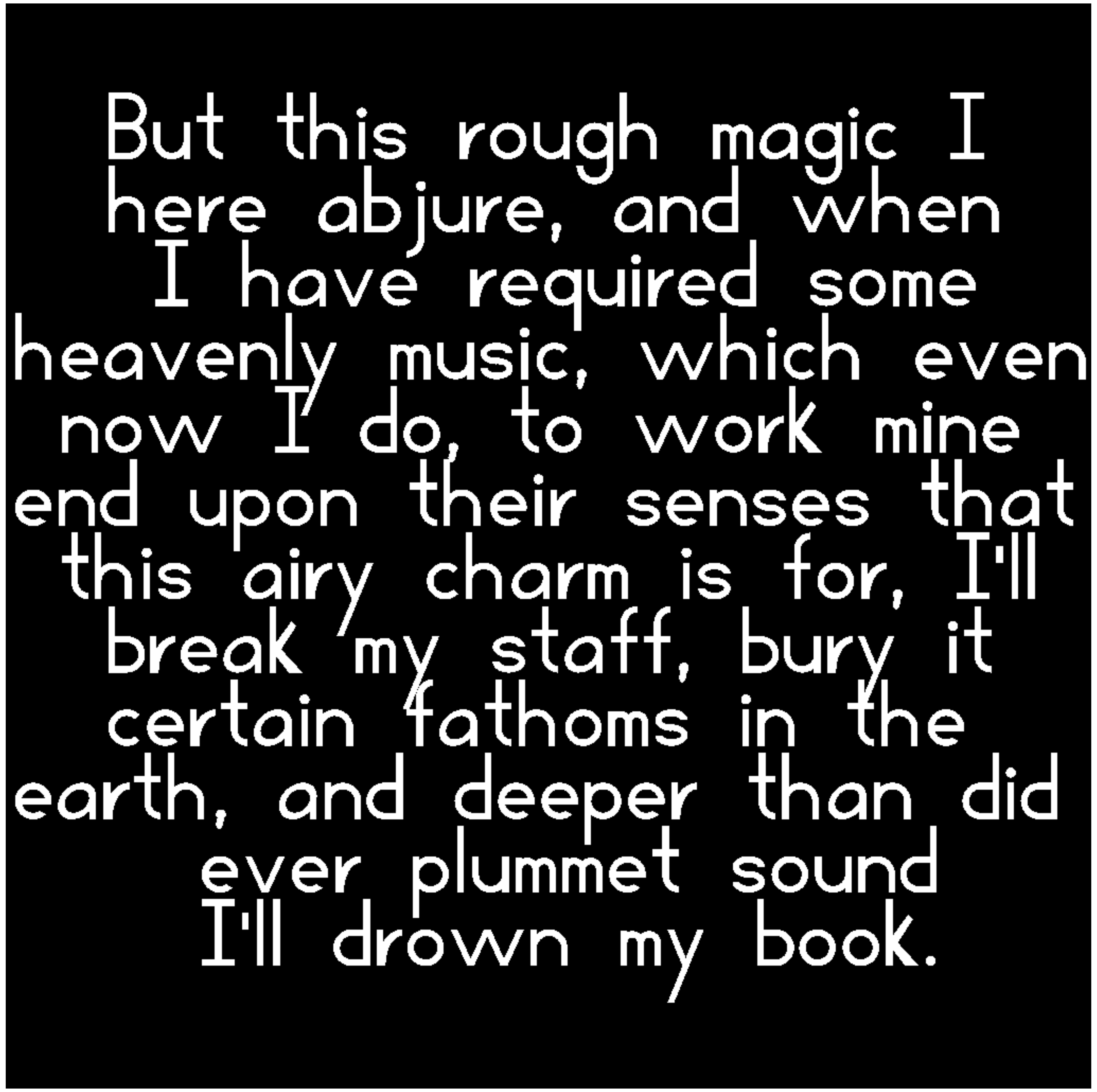
The final frontier

```
r0 = var-y              ldr s0, [x0, 4]
r0 = square r0          fmul s0, s0, s0
r1 = var-x              ldr s1, [x0, 0]
r1 = square r1          fmul s1, s1, s1
r0 = add r0 r2          fadd s0, s0, s1
r0 = sqrt r0            fsqrt s0, s0
r1 = const 1            movz w9, 0x3f80, lsl 16
                        fmov s1, w9
r0 = sub r0 r1          fsub s0, s0, s1
```


Math tree
Math graph
Instruction tape
Bytecode
Assembly
Deduplication
Flattening
Register allocation
Codegen

Performance comparison

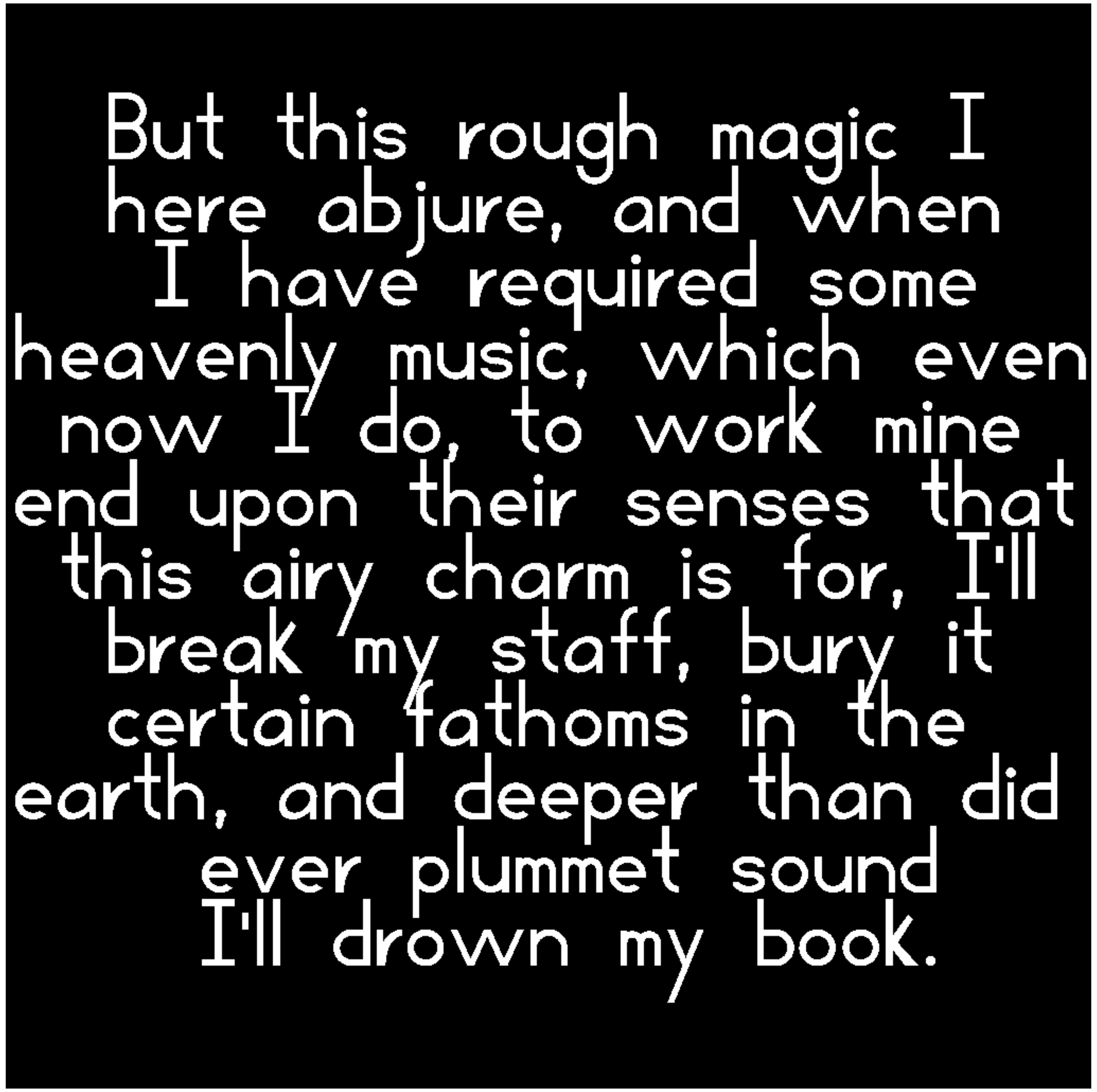
2D benchmarking

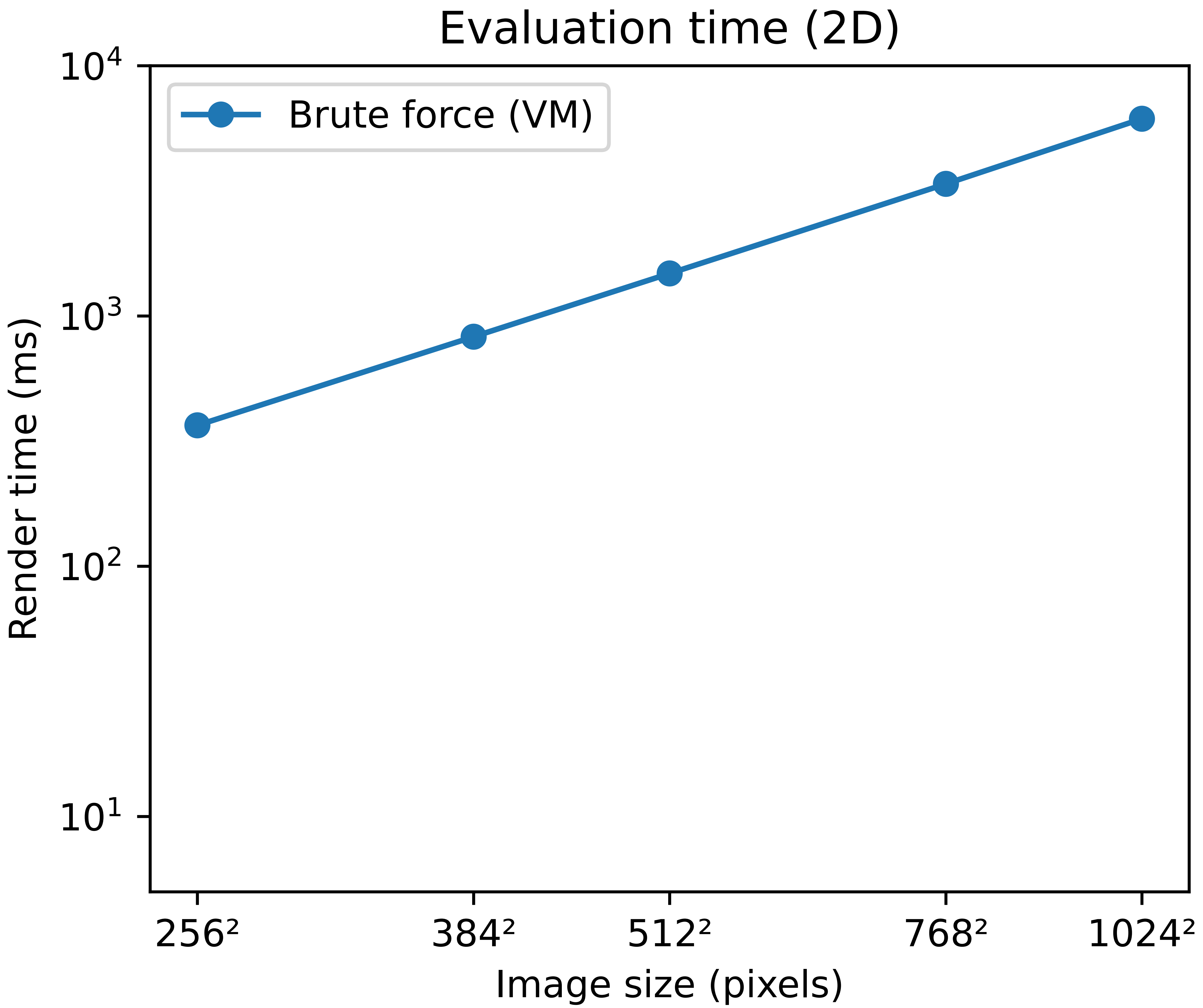


7867 math operations,
2354 of which are CSG

Performance comparison

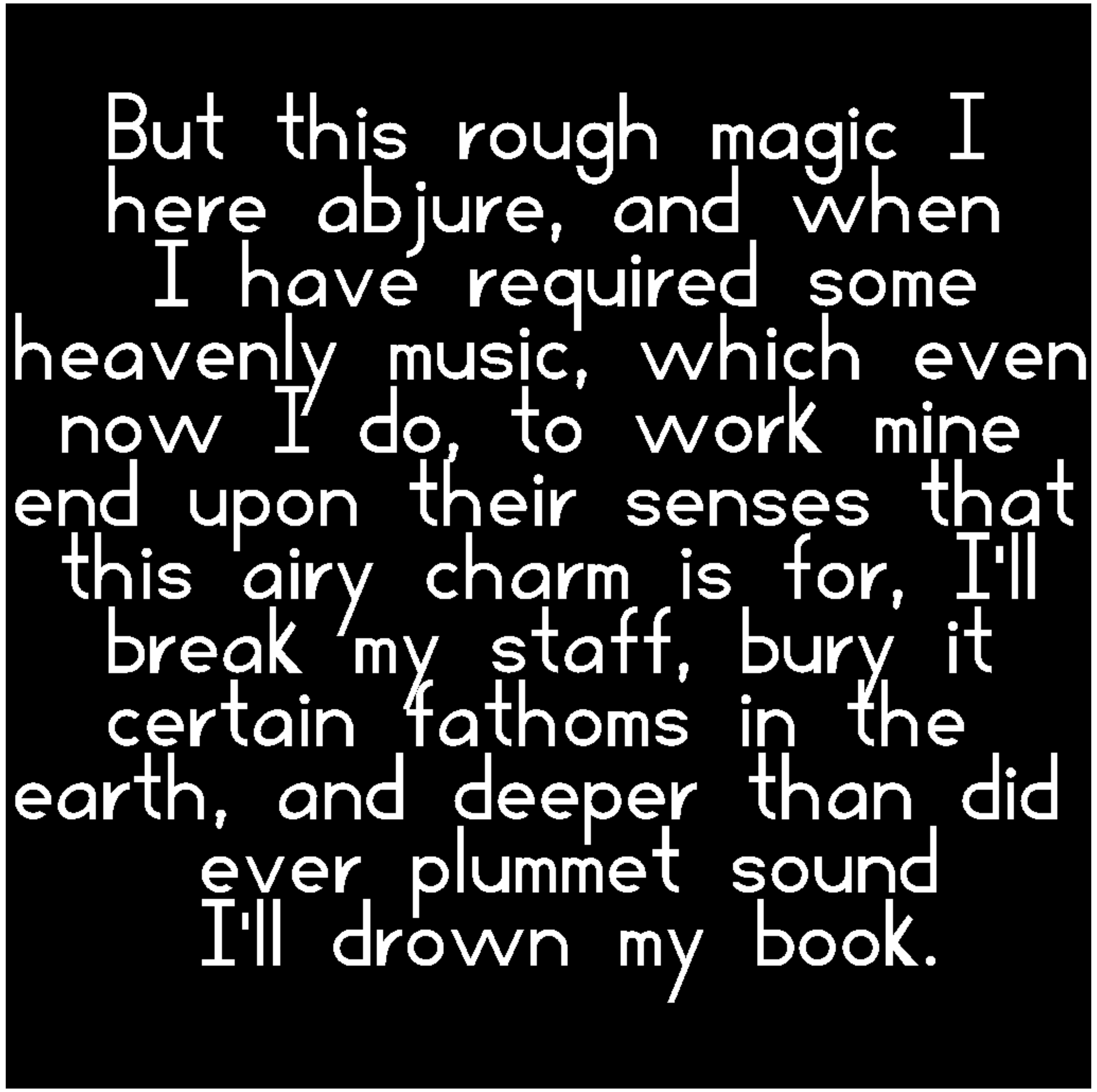
2D benchmarking

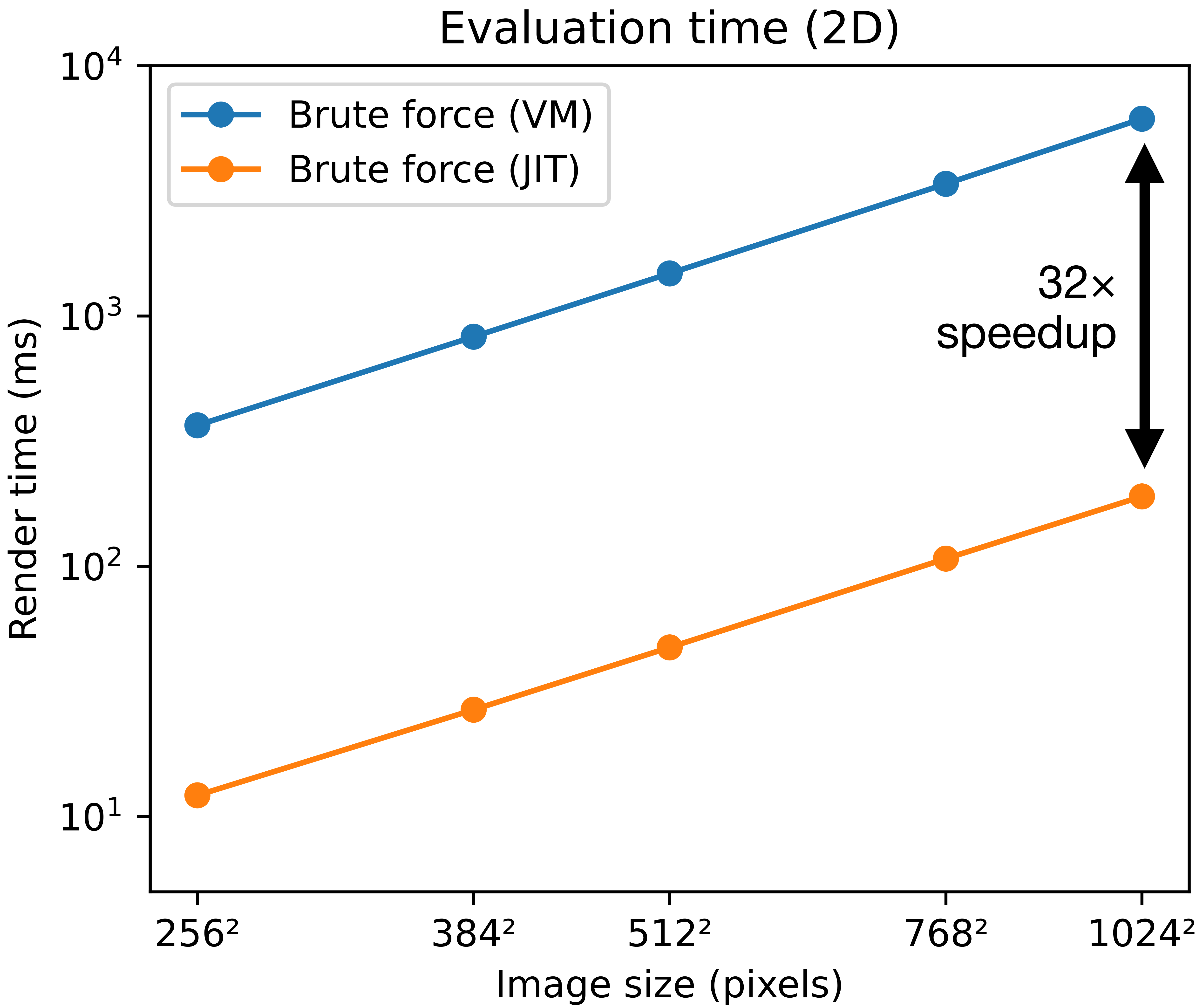




Performance comparison

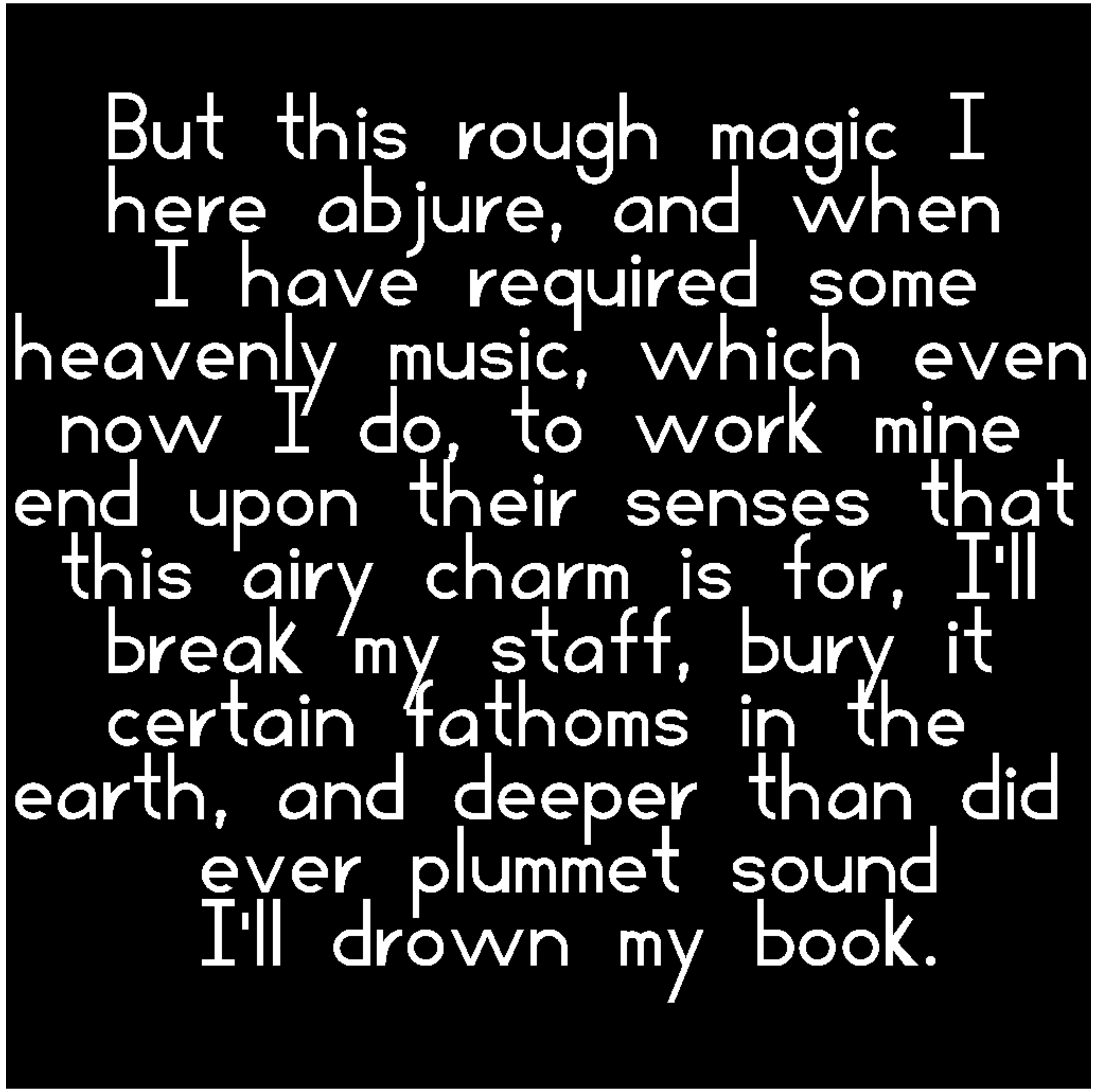
2D benchmarking

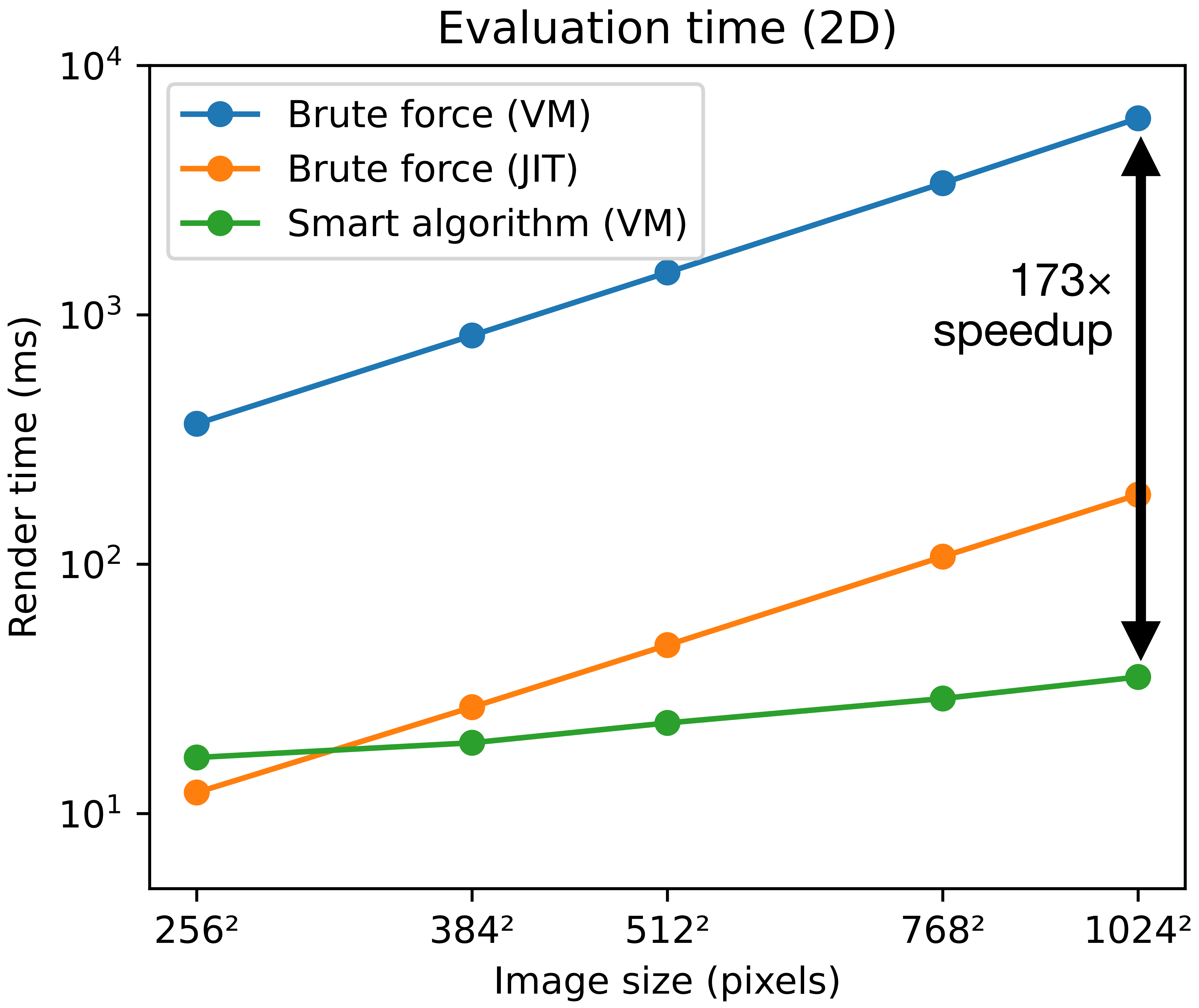




Performance comparison

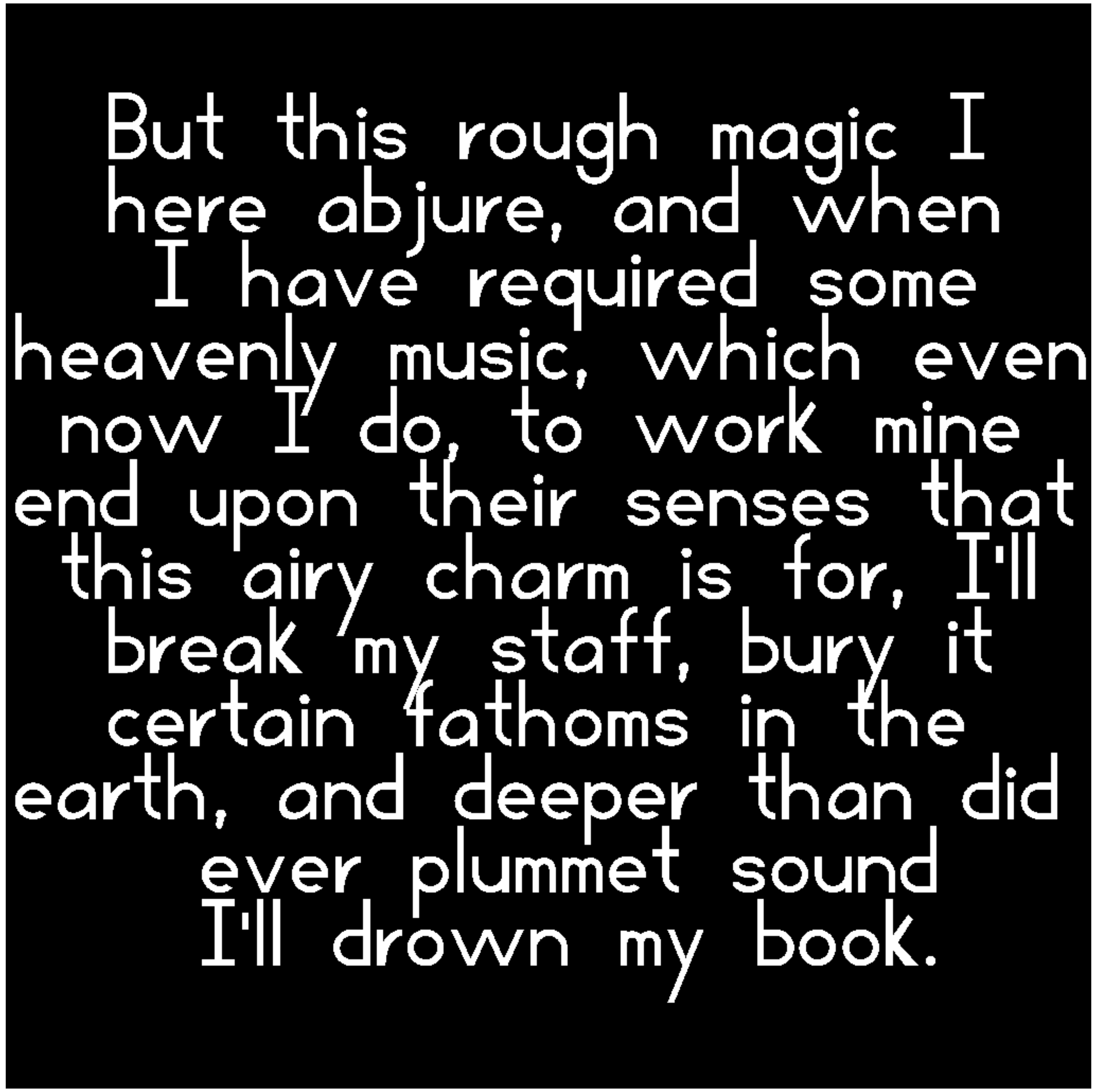
2D benchmarking

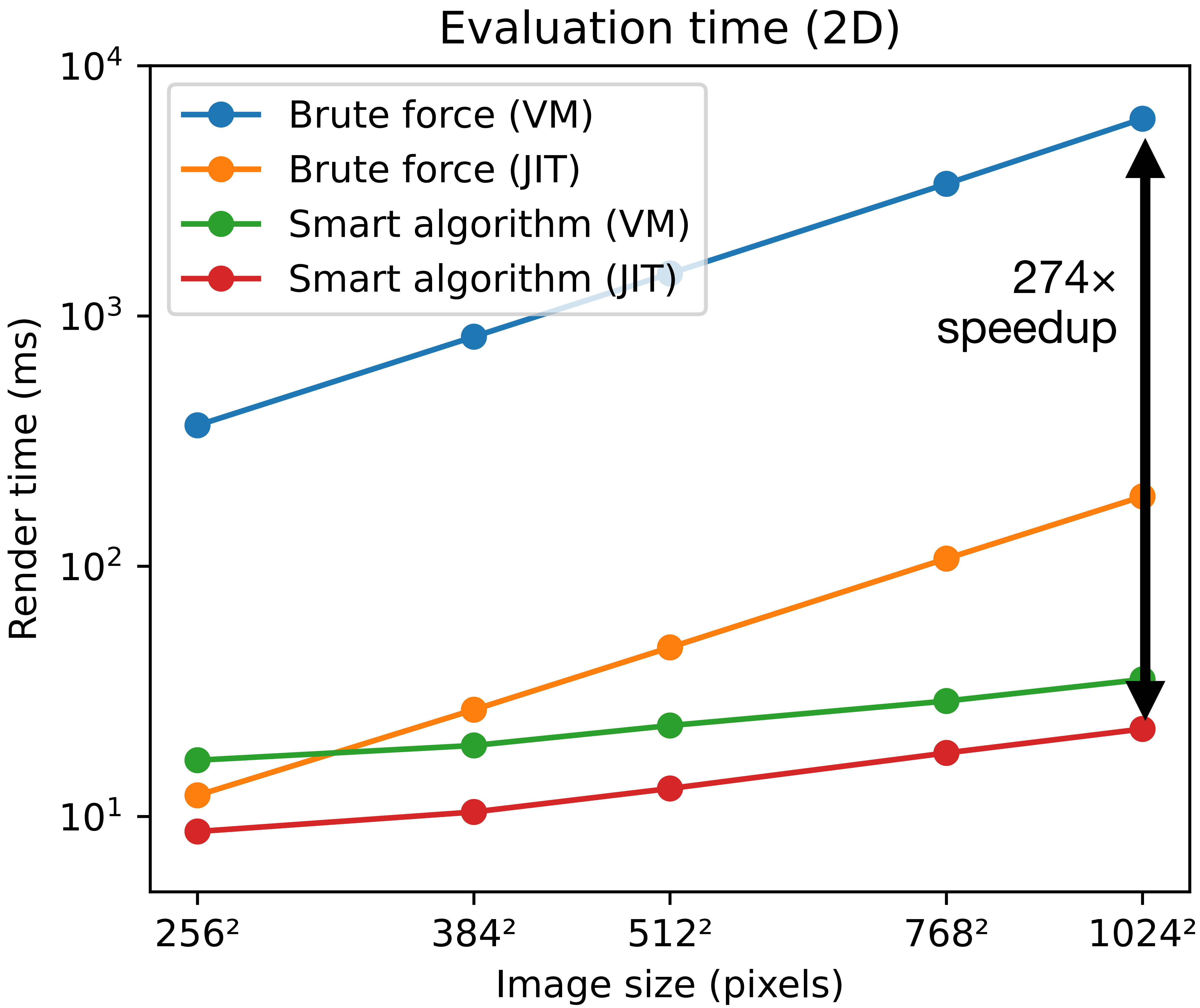




Performance comparison

2D benchmarking





Graphics programming

Data structures

Algorithms

Numerical programming

Compilers

GPU programming

What does independent research look like?

read papers

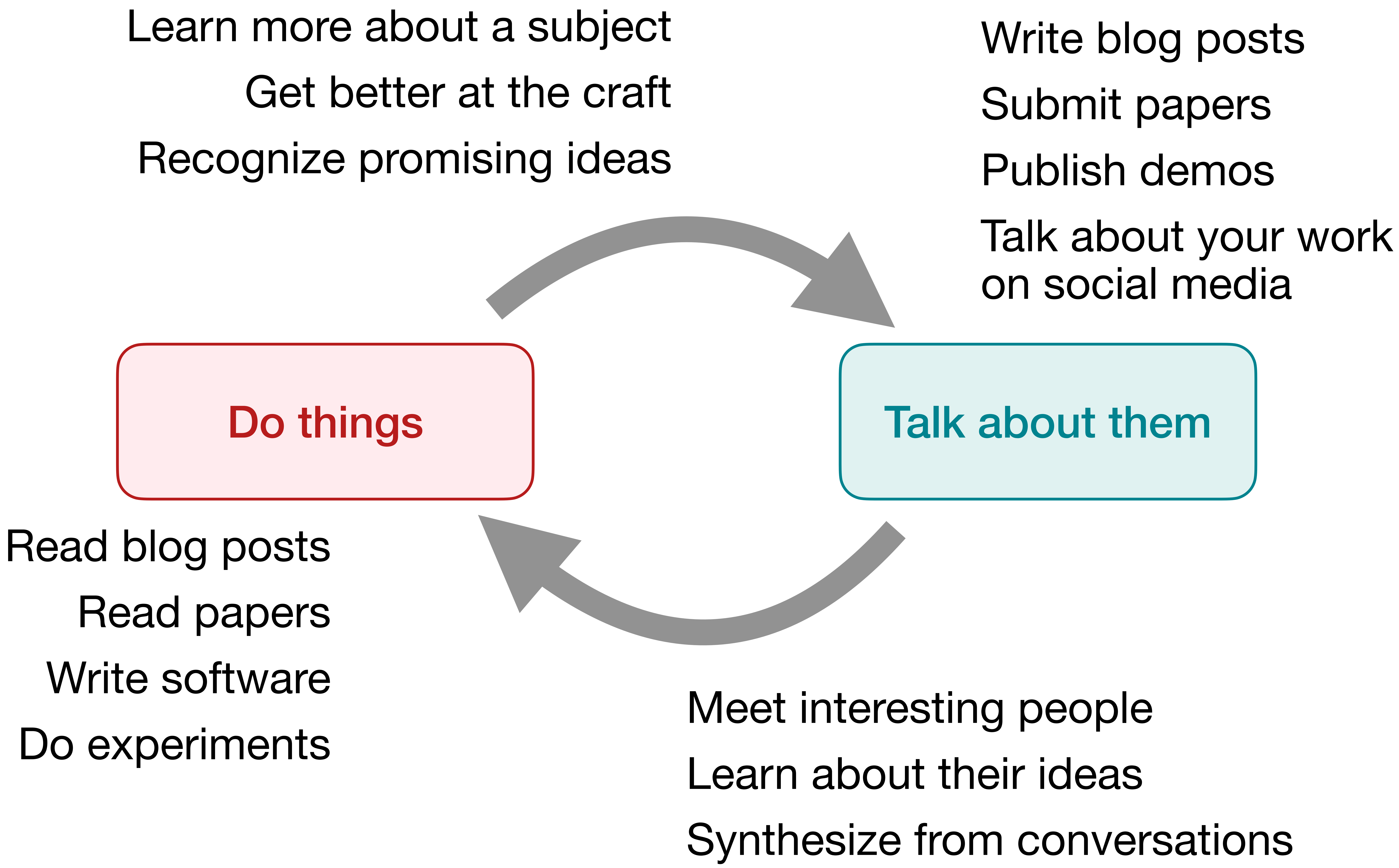
write software

email authors

You can just do things

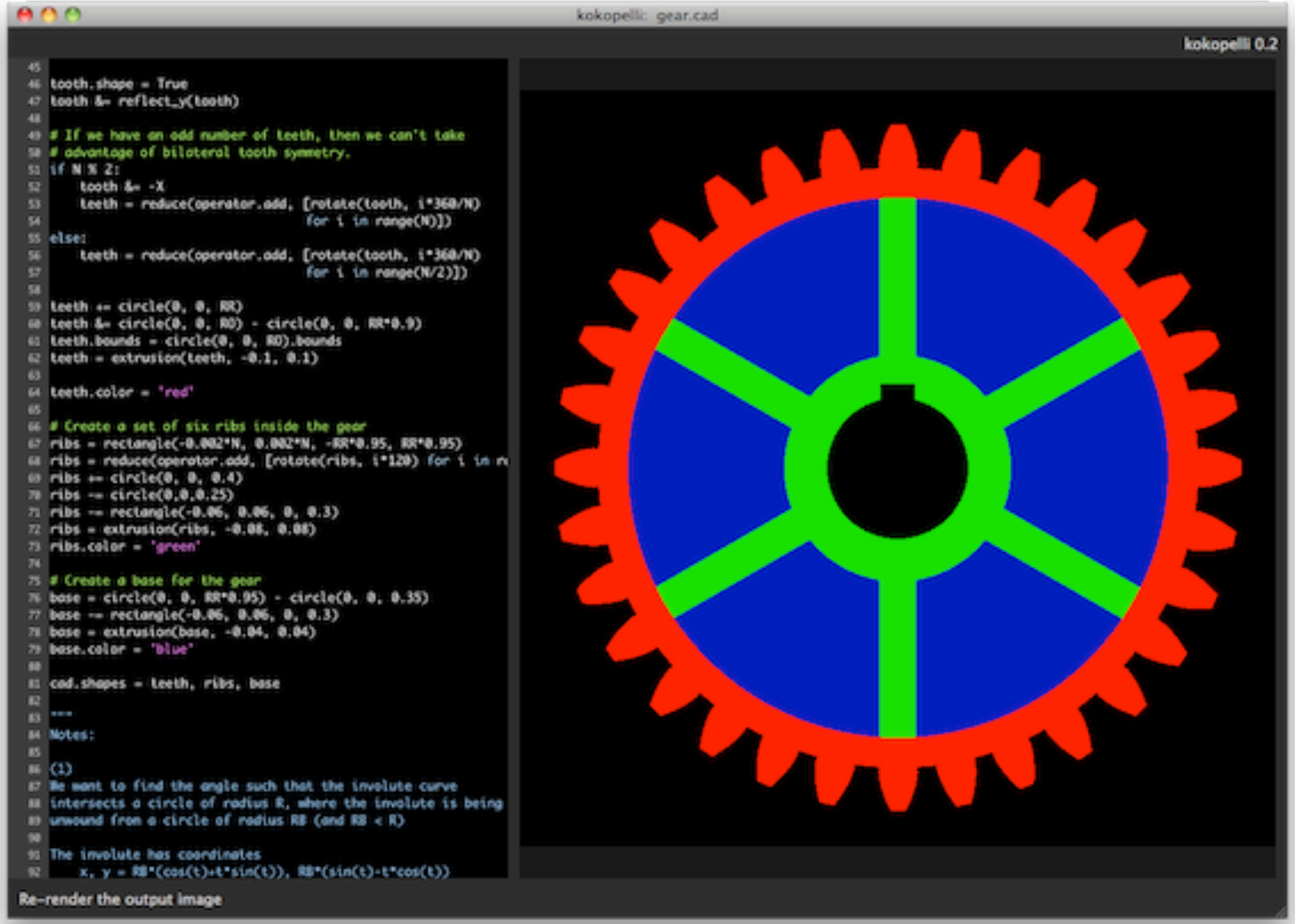
publish blog posts

submit to journals


Learn more about a subject
Get better at the craft
Recognize promising ideas
Write blog posts
Submit papers
Publish demos
Talk about your work on social media
Do things
Talk about them
Read blog posts
Read papers
Write software
Do experiments
Meet interesting people
Learn about their ideas
Synthesize from conversations

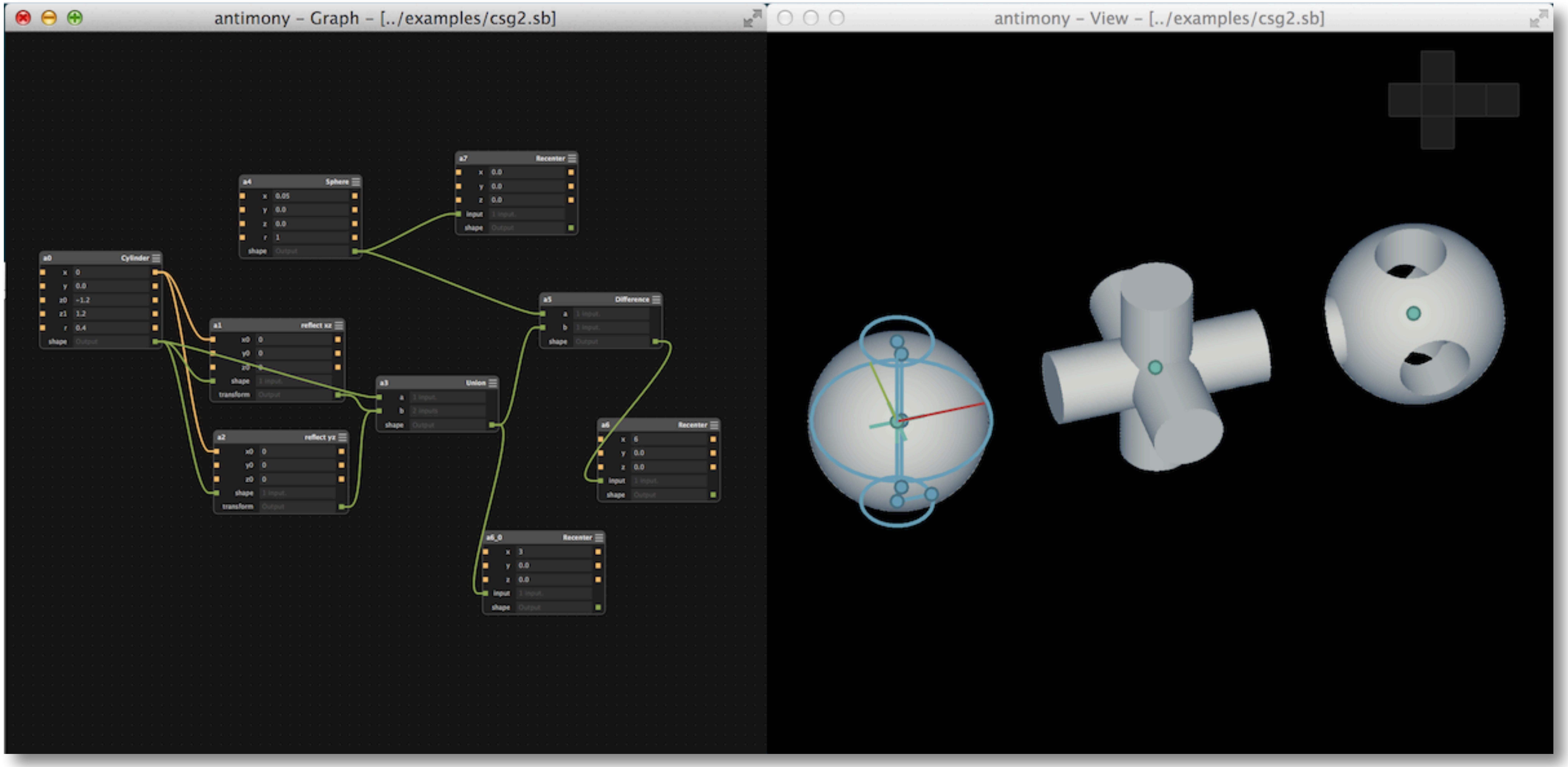
Kokopelli (2013)

C, Python, script-based UI



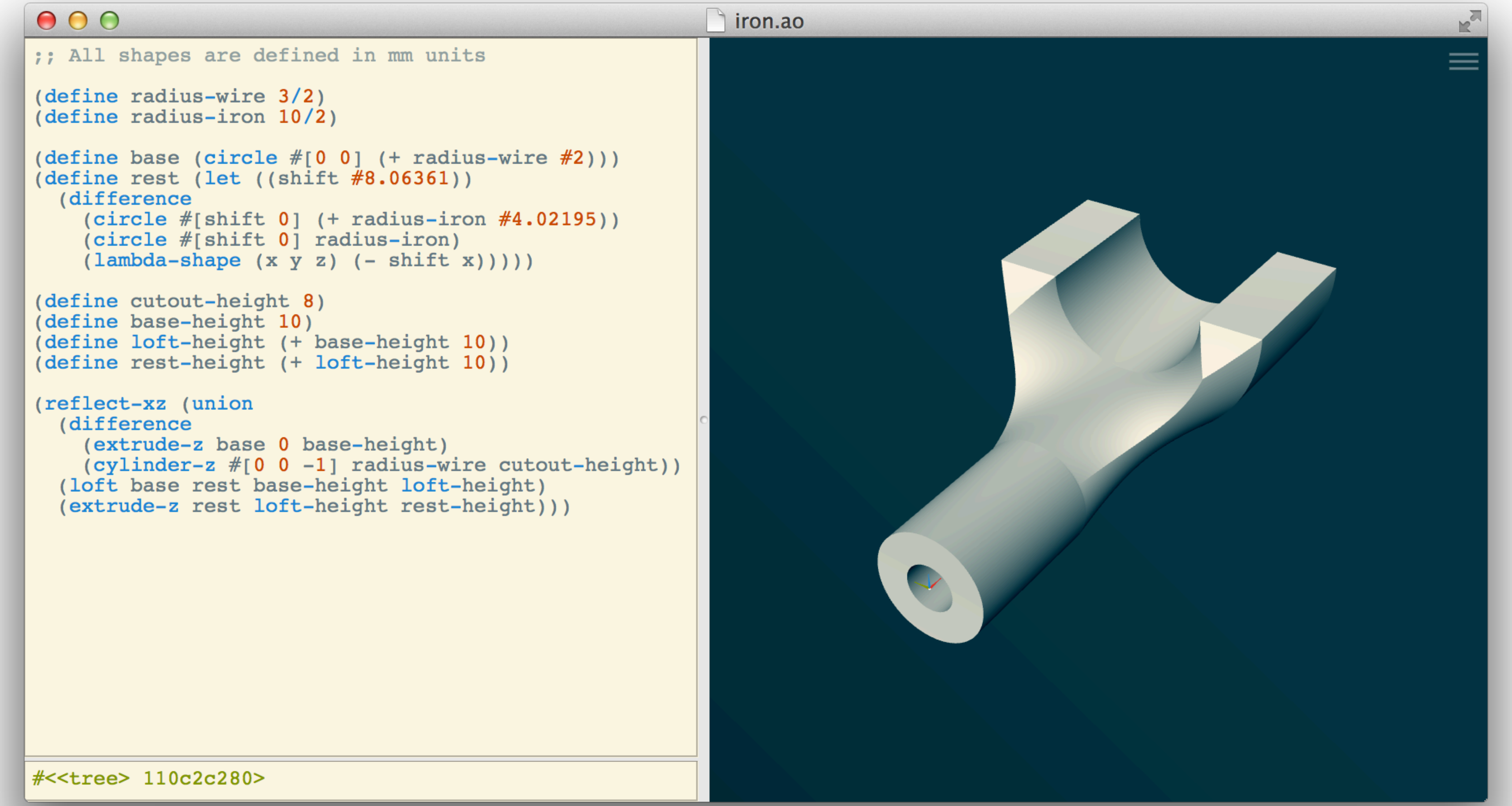
Antimony (2015)

C/C++, Python, same kernel, graph-based UI



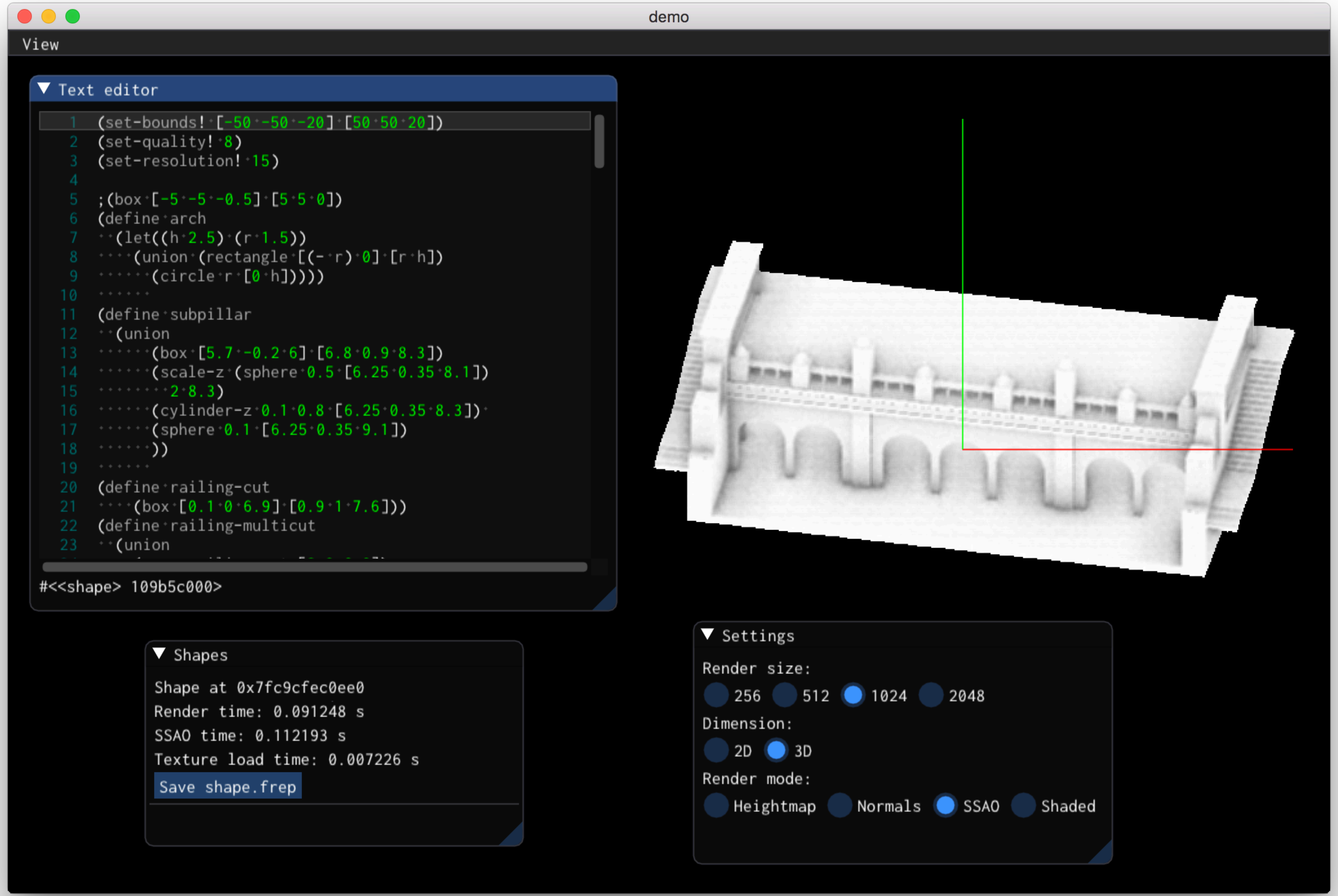
libfive + Studio (2018)

C++, Scheme, new kernel with robust meshing



Porting to CUDA (2020)

GPU-powered rendering

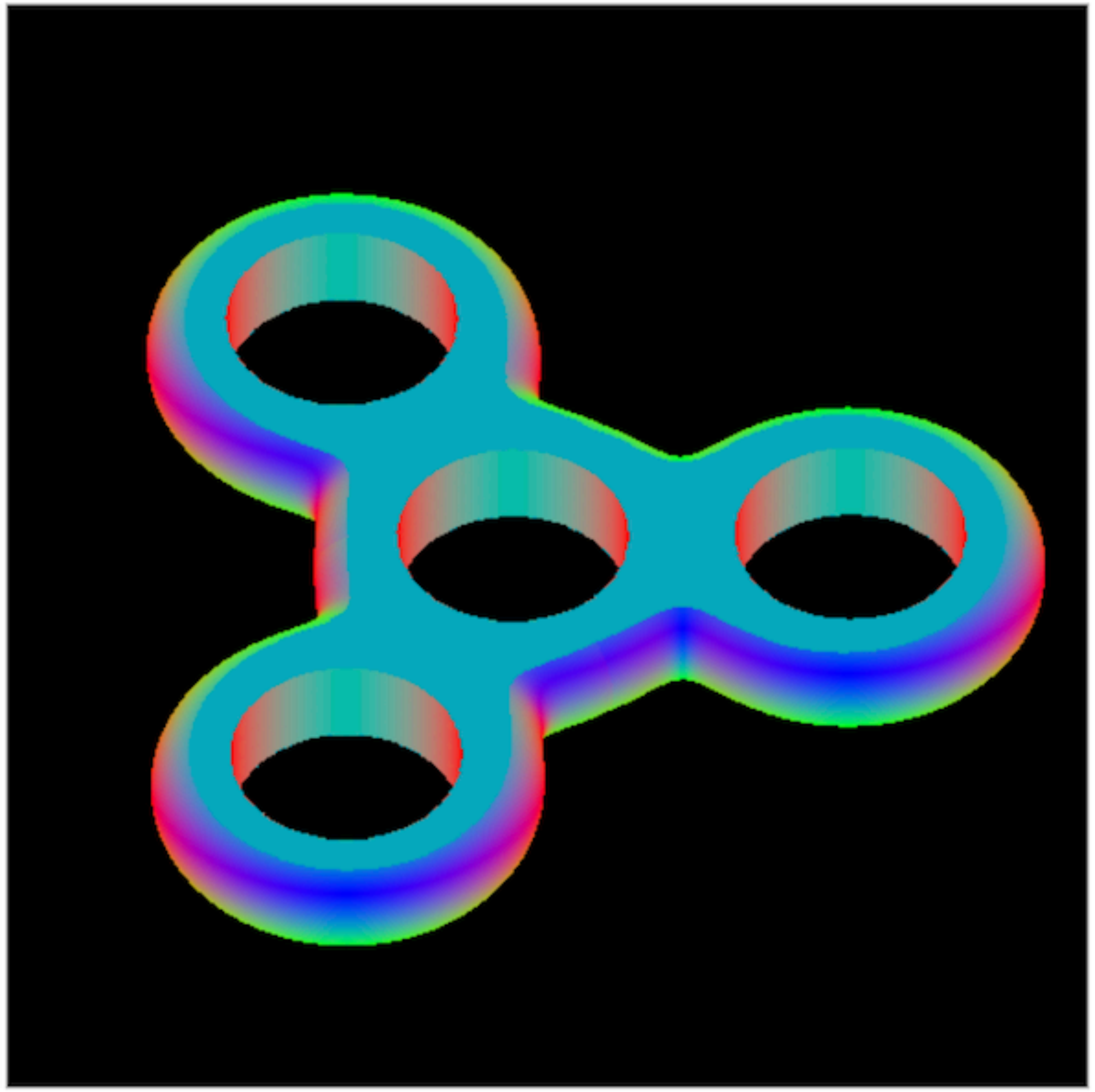


Fidget (2024)

Rust, WebAssembly, new kernel with JIT compiler

```
let r_outer = 1;
let r_inner = 0.35;

let r = sqrt(square(x) + square(y));
let cross = union(
  sqrt(square(x - r_outer) + square(z)) - r_inner,
  intersection(x - r_outer,
    intersection(z - r_inner, -r_inner - z)));
let puck = cross.remap_xyz(r, y, z);

let three = 10000.0;
let PI = 3.14159;
let offset = 2 * r_outer + r_inner;
for i in 0..3 {
    let angle = i / 3.0 * 2 * PI;
    let shifted = puck.remap_xyz(
        x + offset * cos(angle),
        y + offset * sin(angle),
        z);
    three = union(three, shifted);
}

// smooth blend
let k = 0.3;
let v = three - puck;
let out = 0.5 * (puck + three - sqrt(square(v) + k*k));

// clip to Z bounds
```

Ok(..)



3D (normals)

Rendered in 40.96 ms

2025 Implicit Surfaces & Independent Research
Fidget

2024 Fidget: Yet Another Implicit Kernel
Efficiently updating implicit in-order forests
A simple adversarial model for dual contouring

2022 Do Not Taunt Happy Fun Branch Predictor
The Solid-State Register Allocator
Ray tracing with M-reps
Writing a SIGGRAPH paper (for fun)
Implicit Surfaces on the GPU

2020 Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces (+talk)
Quadratic Error Function Explainer
Consulting on libfive
libfive + Studio

2018 Implicit Kernels for Solid Modeling
Consistent Ordering of N-Dimensional Neighbors

2017 QEFs, Eigenvalues, and Normals
Finding bounding boxes with interval math
Fixing a soldering iron with 3D printing
Zero-crossing logic for robust meshing
Higher-order reactive graph programming

2016 Lineage of CBA CAD tools
Abstraction and instances in graph programming
Ao: Homoiconic solid modeling
Automatic tracking of bounding boxes
Affine coordinates in Ao
Ao
Representation and JITting of math trees
2D contouring

2013 Antimony
Kokopelli

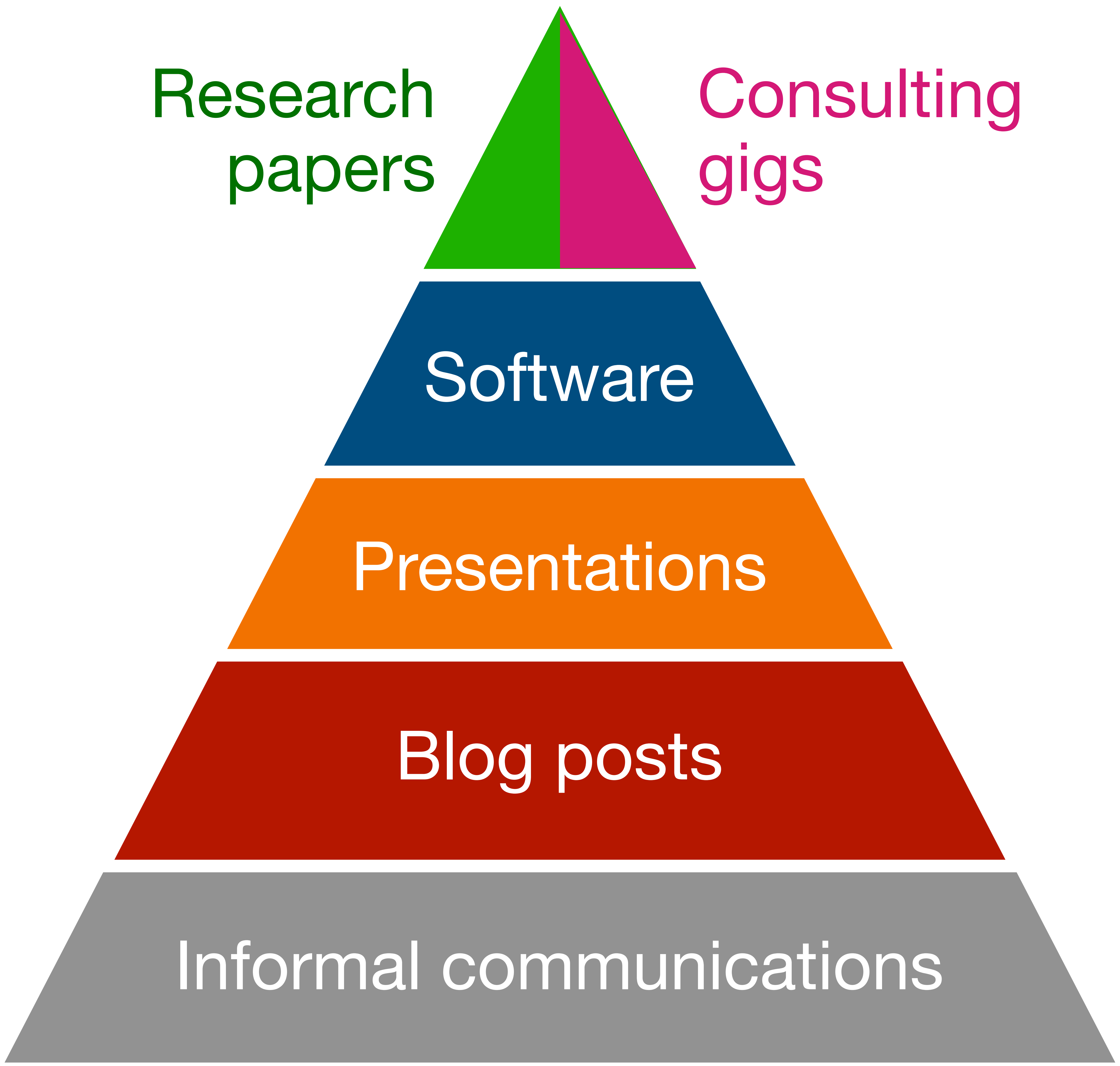
Implicit Surfaces & Independent Research
Fidget
Fidget: Yet Another Implicit Kernel
Efficiently updating implicit in-order forests
A simple adversarial model for dual contouring
Do Not Taunt Happy Fun Branch Predictor
The Solid-State Register Allocator
Ray tracing with M-reps
Writing a SIGGRAPH paper (for fun)
Implicit Surfaces on the GPU
Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces (+talk)
Quadratic Error Function Explainer
Consulting on libfive
libfive + Studio
Implicit Kernels for Solid Modeling
Consistent Ordering of N-Dimensional Neighbors
QEFs, Eigenvalues, and Normals
Finding bounding boxes with interval math
Fixing a soldering iron with 3D printing
Zero-crossing logic for robust meshing
Higher-order reactive graph programming
Lineage of CBA CAD tools
Abstraction and instances in graph programming
Ao: Homoiconic solid modeling
Automatic tracking of bounding boxes
Affine coordinates in Ao
Ao
Representation and JITting of math trees
2D contouring
Antimony
Kokopelli

19 blog posts
6 presentations
5 software packages
1 research paper
1 paid consulting gig

Not shown:

Dozens of email conversations and one-on-one interactions

Too many tweets


Research papers
Consulting gigs
Software
Presentations
Blog posts
Informal communications

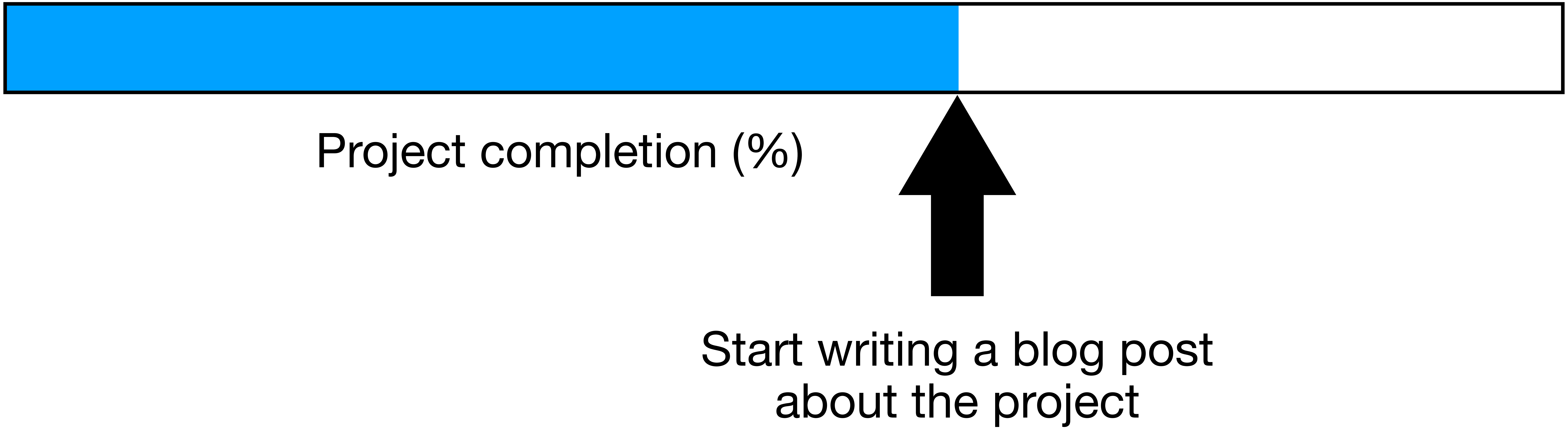
Motivation and energy

Staying motivated

Project completion (%)

Staying motivated

through the power of blog-post driven development



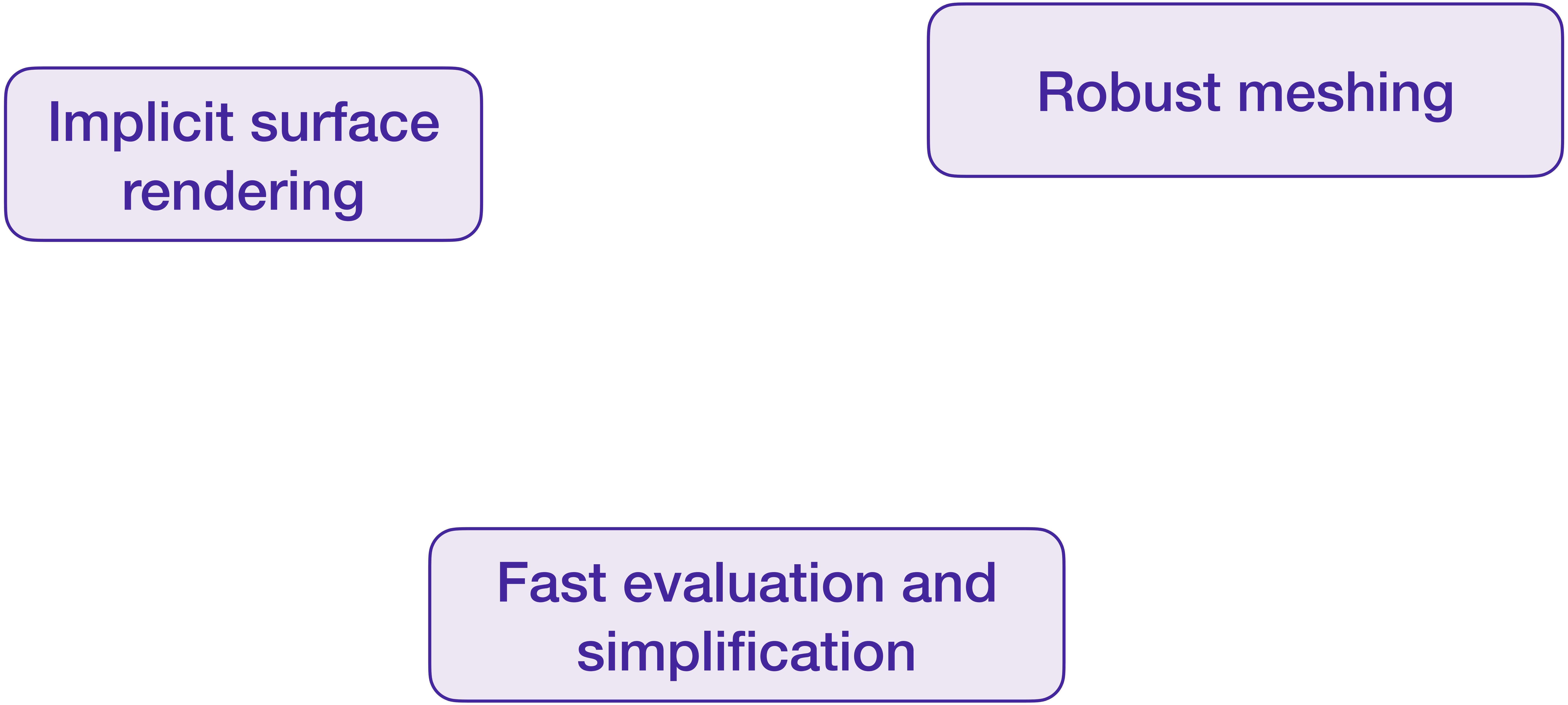
Blog-post driven development

Why does this work?

- Stop doing open-ended work!
- Narrowly focus on the writeup
- Reminds me what’s cool about the project
- Sharing work has social benefits
- You should have a website

Coming up with ideas


A lot of reading
+
Pet problems
=
Coming up with new ideas


Implicit surface rendering
Robust meshing
Fast evaluation and simplification

Implicit surface rendering

+

Raph Levien's blog

2D Graphics on Modern GPU

May 8, 2019

=

Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces

MATTHEW J. KEETER, Independent researcher

Fig. 1. An assortment of implicit surfaces rendered using our technique. Left: an extruded text string, rotated and rendered as a heightmap. Center: a bear head sculpted using smooth blending operations, with normals found by automatic differentiation. Right: a complex architectural model rendered with screen-space ambient occlusion and perspective. All models are rendered directly from their mathematical representations, without triangulation or raytracing.

We present a new method for directly rendering complex closed-form implicit surfaces on modern GPUs, taking advantage of their massive parallelism. Our model representation is unambiguously solid, can be sampled at arbitrary resolution, and supports both constructive solid geometry (CSG) and more unusual modeling operations (e.g. smooth blending of shapes). The rendering strategy scales to large-scale models with thousands of arithmetic operations in their underlying mathematical expressions. Our method only requires C^0 continuity, allowing for warping and blending operations which break Lipshitz continuity.

To render a model, its underlying expression is evaluated in a shallow hierarchy of spatial regions, using a high branching factor for efficient parallelization. Interval arithmetic is used to both skip empty regions and construct reduced versions of the expression. The latter is the optimization that makes our algorithm practical: in one benchmark, expression complexity decreases by two orders of magnitude between the original and reduced expressions. Similar algorithms exist in the literature, but tend to be deeply recursive with heterogeneous workloads in each branch, which makes them GPU-unfriendly; our evaluation and expression reduction both run efficiently as massively parallel algorithms, entirely on the GPU.

The resulting system renders complex implicit surfaces in high resolution and at interactive speeds. We examine how performance scales with computing power, presenting performance results on hardware ranging from older laptops to modern data-center GPUs, and showing significant improvements at each stage.

CCS Concepts: • **Computing methodologies** → **Rasterization**; **Volumetric models**.

Additional Key Words and Phrases: implicit surface, signed distance field, freps, octrees, rasterization, gpu, cuda

Author's address: Matthew J. Keeter, Independent researcher, matt.j.keeter@gmail.com.

0730-0301/2020/7-ART141 $15.00
https://doi.org/10.1145/3386569.3392429

ACM Reference Format:
Matthew J. Keeter. 2020. Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces. *ACM Trans. Graph.* 39, 4, Article 141 (July 2020), 10 pages. https://doi.org/10.1145/3386569.3392429

1 INTRODUCTION

Implicit surfaces and functional representations are a powerful way to represent solid models [Bloomenthal and Wyvill 1997; Gomes et al. 2009]. Compared to boundary representations (e.g. triangle meshes or NURBS surfaces), they offer unambiguous inside-outside checking, easy constructive solid geometry (CSG) operations, and arbitrary resolution. In recent years, functional representations (f-reps) have been used as the kernel of both commercial [Courter 2019] and open-source [Keeter 2019] CAD packages. They are a fundamental building block in the demoscene community [Burger et al. 2002; Quilez 2008], used as a representation for generative art [Moen 2019], and even as the underlying technology for a recent PlayStation 4 game [Evans 2015].

Unlike boundary representations, implicit surfaces cannot easily be rendered in their native forms. This paper presents a new method for rendering the family of implicit surfaces represented by arbitrary closed-form arithmetic expressions, i.e., representing a sphere as

$$f(x,y,z) < 0 \text{ where } f(x,y,z) = \sqrt{x^2+y^2+z^2} - 1$$

This representation is particularly flexible and can be treated as an "assembly language for shapes" which is targeted by higher-level tools. The space of higher-level tools spans the gamut from advanced solid modeling packages [Allen 2019] to user-friendly content generation tools [Keeter 2015].

Our rendering strategy runs in both 2D and 3D, making efficient use of modern GPU hardware and APIs. Unlike previous work, it scales to complex expressions, maintaining interactive framerates while rendering models built from hundreds or thousands of arithmetic operations. It requires no continuity higher than C^0, which allows for extremely flexible modeling and unusual spatial transformations. Finally, it scales well with GPU power; as GPU performance

Where to find pet problems?

- Adopt them
- Read a lot (papers, blog posts, etc)
  - Find interesting people through news aggregators
  - Subscribe to them via RSS
- Look for personal-scale problems
- Revisiting old ideas on modern hardware

Thank you!

Matt Keeter
mattkeeter.com