SIGGRAPH THINK BEYOND

2020 S2020.SIGGRAPH.ORG

MASSIVELY PARALLEL RENDERING OF COMPLEX CLOSED-FORM IMPLICIT SURFACES

Matthew J. Keeter

Independent researcher

DEFINING IMPLICIT SURFACES

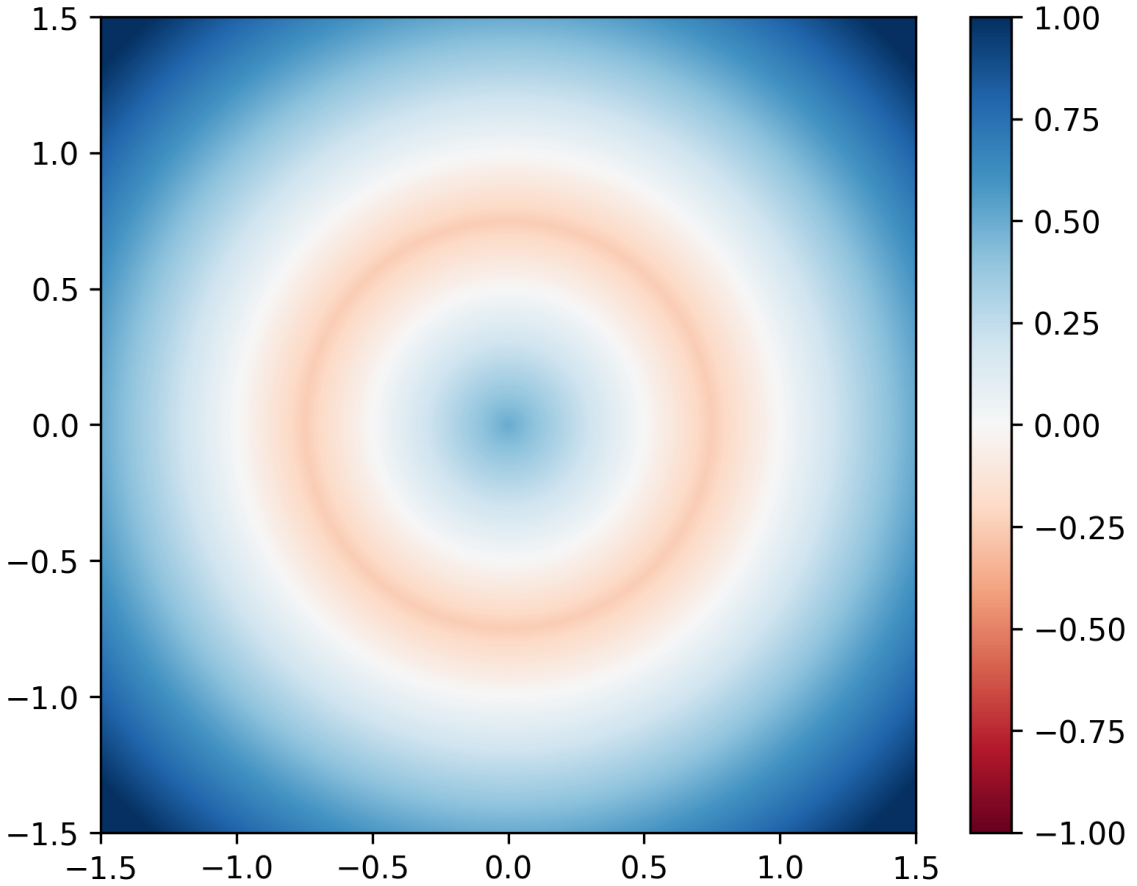
`f(x, y, z) < 0` → inside the shape (“filled”)

`f(x, y, z) > 0` → outside the shape (“empty”)

`f(x, y, z) = 0` → surface of the shape

CLOSED-FORM IMPLICIT SURFACES

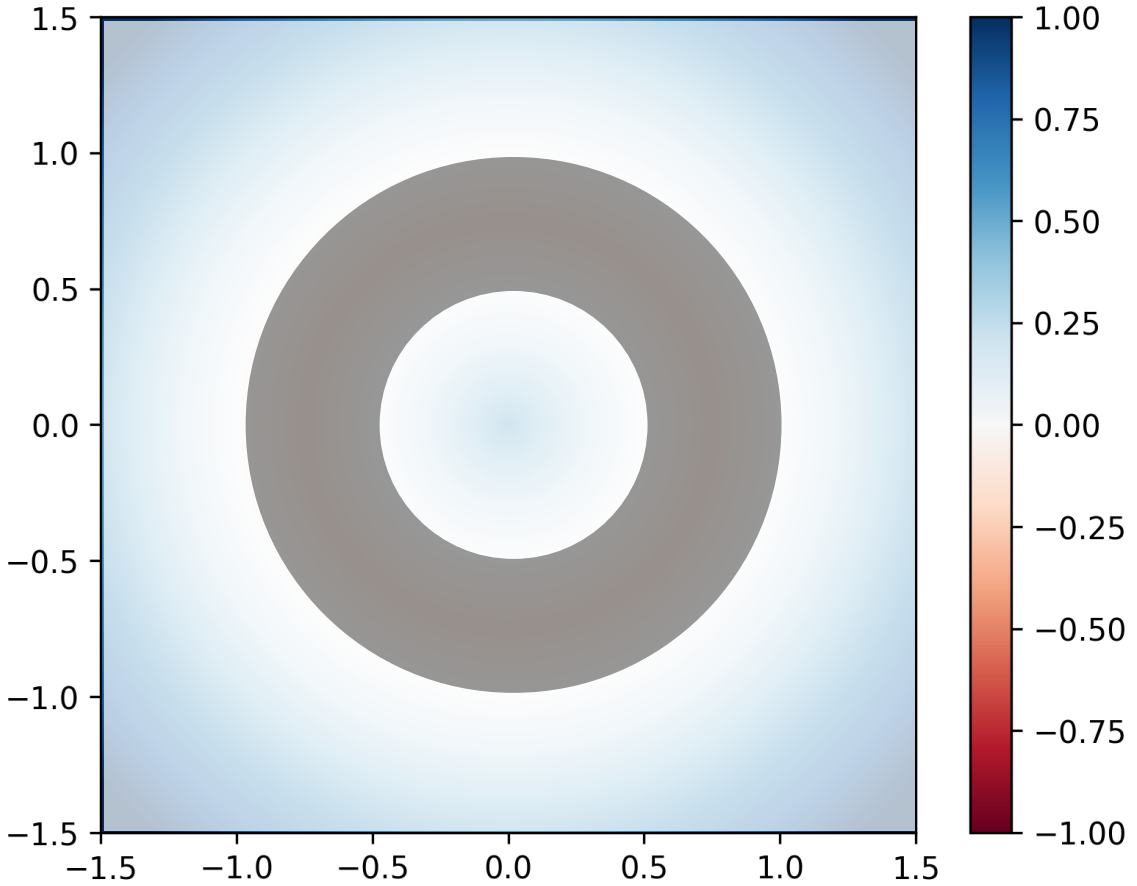
- `f(x, y, z)` is some sequence of mathematical operations
- Our reference implementation supports `+`, `−`, `×`, `/`, `sqrt`, `sin`, `cos`, `asin`, `acos`, `atan`, `exp`, `log`, `abs`, `min`, `max`
- CSG is easy:
  - Intersection → `max`
  - Union → `min`
  - Inverse → `negation`



$$\max\left(0.5 - \sqrt{x^2 + y^2}, \sqrt{x^2 + y^2} - 1\right) < 0$$

CLOSED-FORM IMPLICIT SURFACES

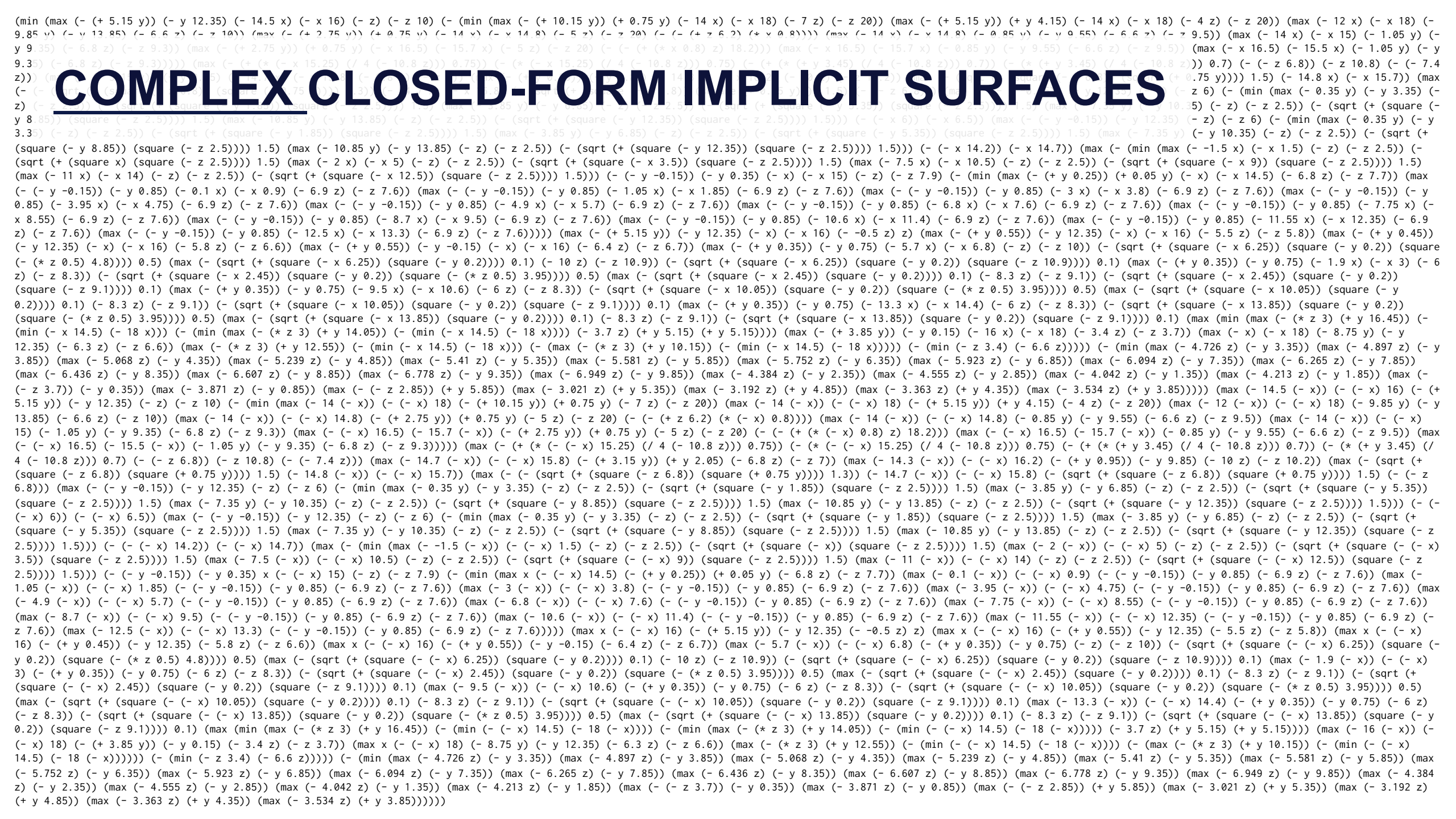
- `f(x, y, z)` is some sequence of mathematical operations
- Our reference implementation supports `+`, `−`, `×`, `/`, `sqrt`, `sin`, `cos`, `asin`, `acos`, `atan`, `exp`, `log`, `abs`, `min`, `max`
- CSG is easy:
  - Intersection → `max`
  - Union → `min`
  - Inverse → `negation`



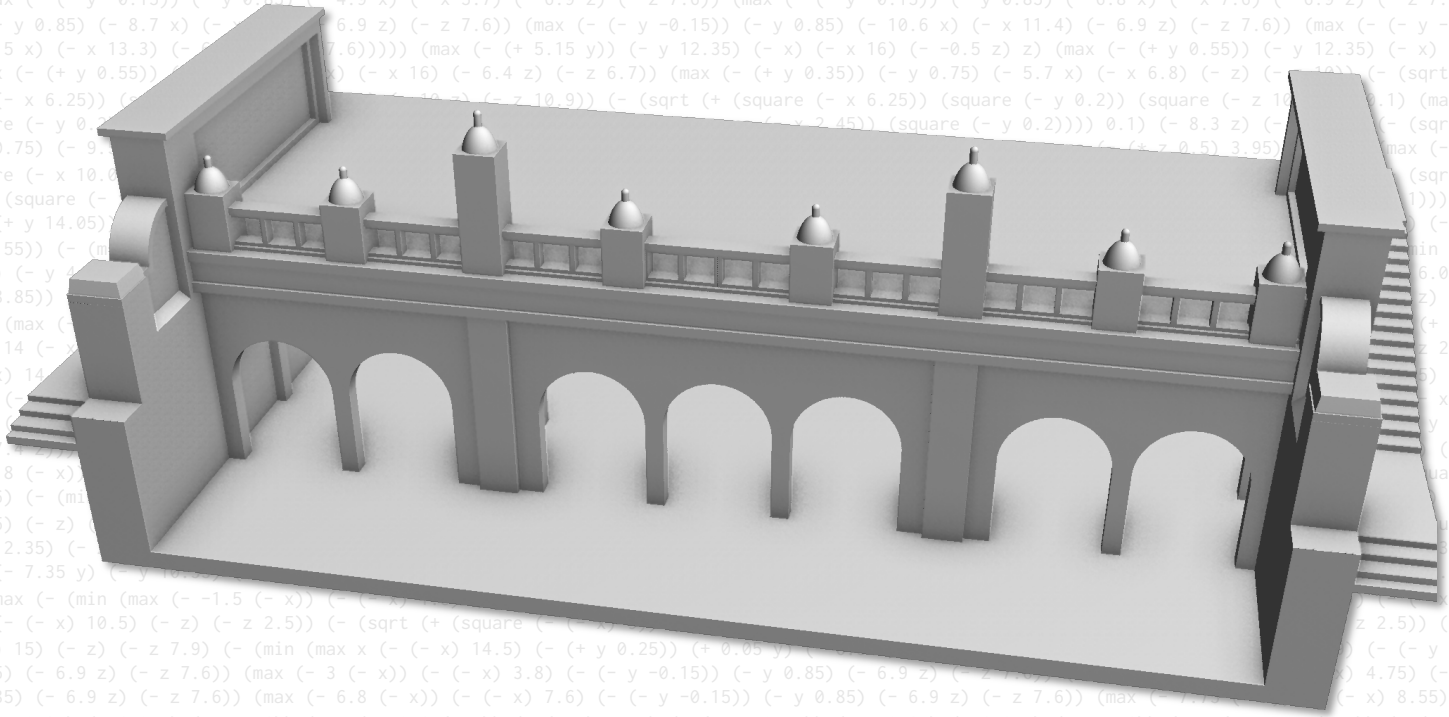
$$\max\left(0.5 - \sqrt{x^2 + y^2}, \sqrt{x^2 + y^2} - 1\right) < 0$$

COMPLEX CLOSED-FORM IMPLICIT SURFACES

COMPLEX CLOSED-FORM IMPLICIT SURFACES



COMPLEX CLOSED-FORM IMPLICIT SURFACES



“ASSEMBLY LANGUAGE FOR SHAPES”

Benefits

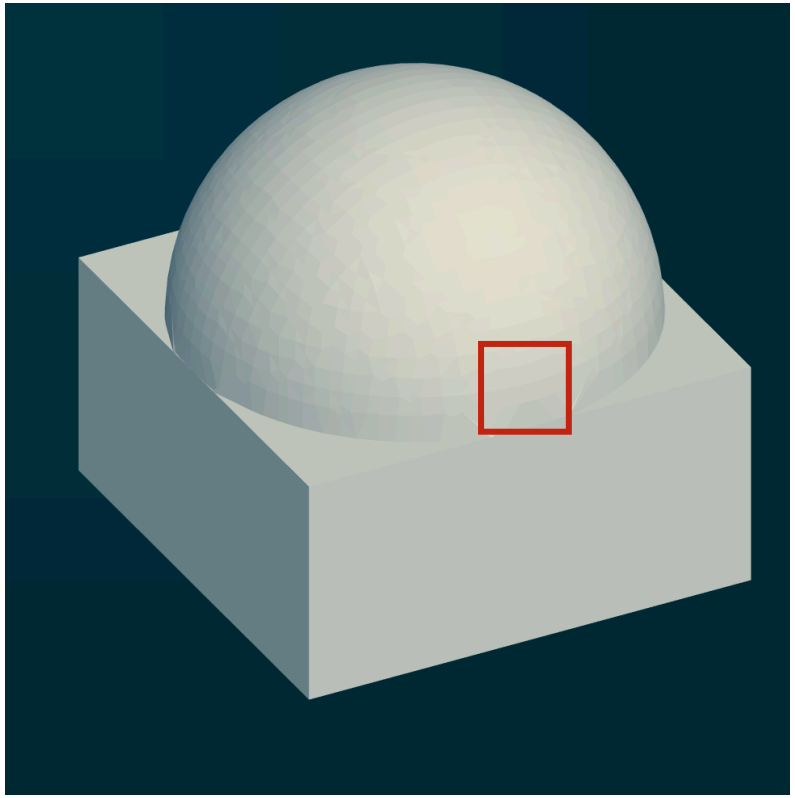
- Easy CSG and solid modeling!
- Unambiguous inside-outside checking
- Supports arbitrary resolution
- Compact representation

Applications

- CAD/CAM
- Demoscene graphics
- Art and exploration
- User-generated content

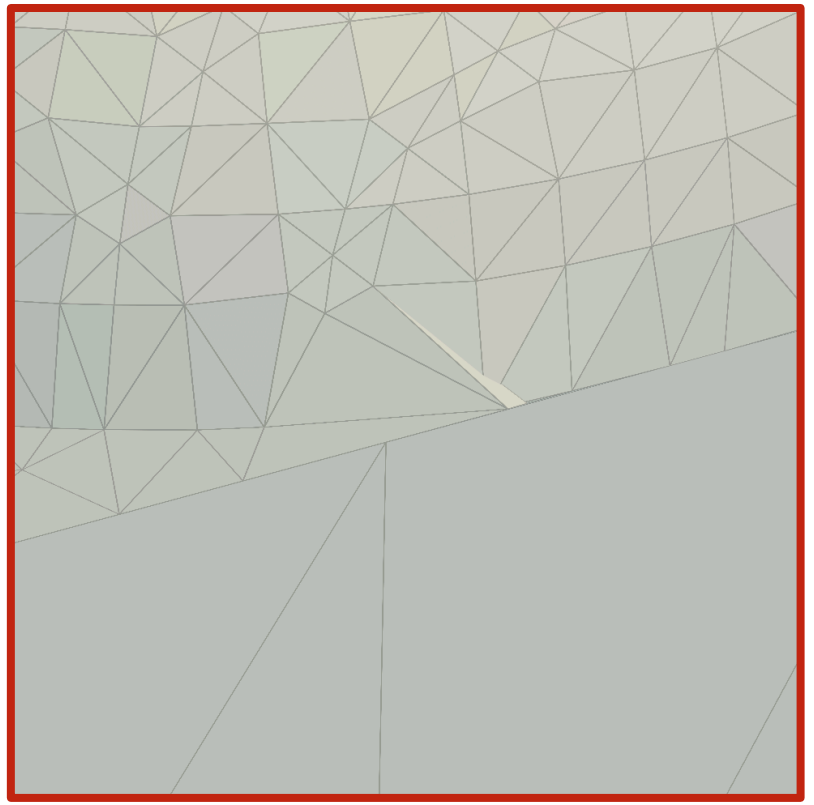
RENDERING IMPLICIT SURFACES IS HARD

- Meshing
  - Hard to get right for arbitrary shapes
  - Philosophically unsatisfying



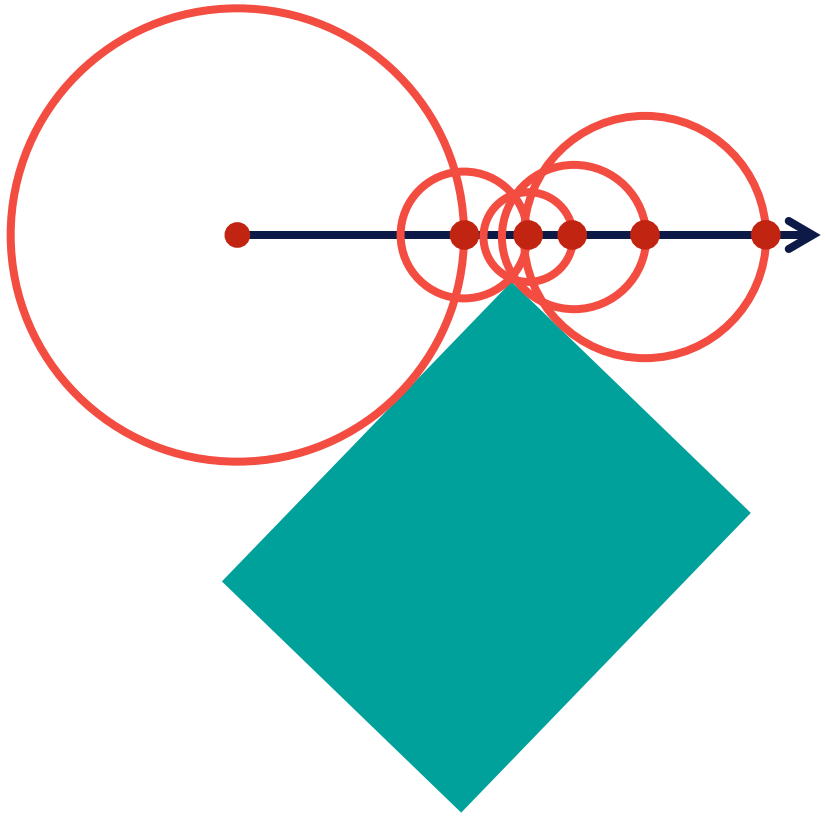
RENDERING IMPLICIT SURFACES IS HARD

- Meshing
  - Hard to get right for arbitrary shapes
  - Philosophically unsatisfying



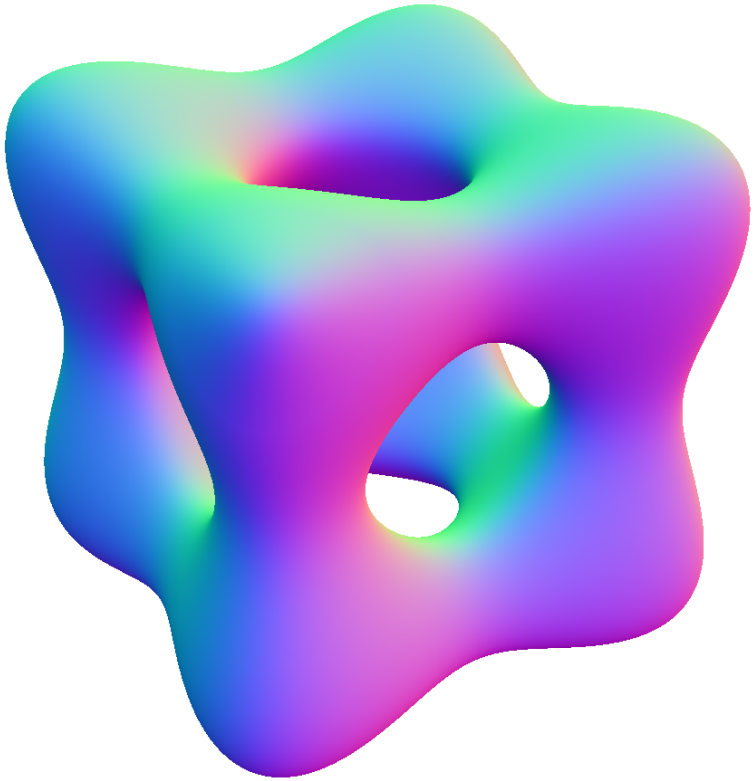
RENDERING IMPLICIT SURFACES IS HARD

- Meshing
  - Hard to get right for arbitrary shapes
  - Philosophically unsatisfying
- Raytracing
  - Often requires C^1 or Lipshitz continuity
  - Limits the kind of transforms that you can apply

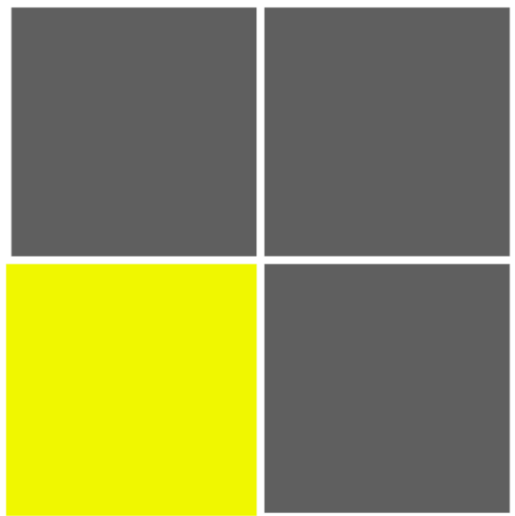


RENDERING IMPLICIT SURFACES IS HARD

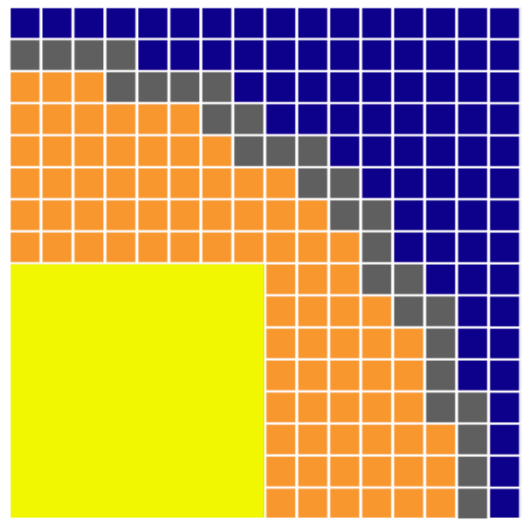
- Meshing
  - Hard to get right for arbitrary shapes
  - Philosophically unsatisfying
- Raytracing
  - Often requires C^1 or Lipshitz continuity
  - Limits the kind of transforms that you can apply
- Our strategy
  - Interval arithmetic + subdivision + recursion (Duff ’92)
  - Per-tile command lists on the GPU (Ganacim et al. ’14)
  - Only requires C0 continuity, and scales to large models
  - Extremely philosophically satisfying



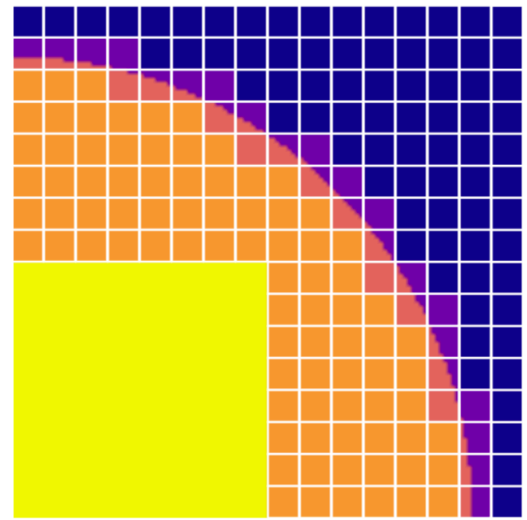
First pass



Second pass



Per-pixel evaluation



Interpreter on the GPU

Interval arithmetic

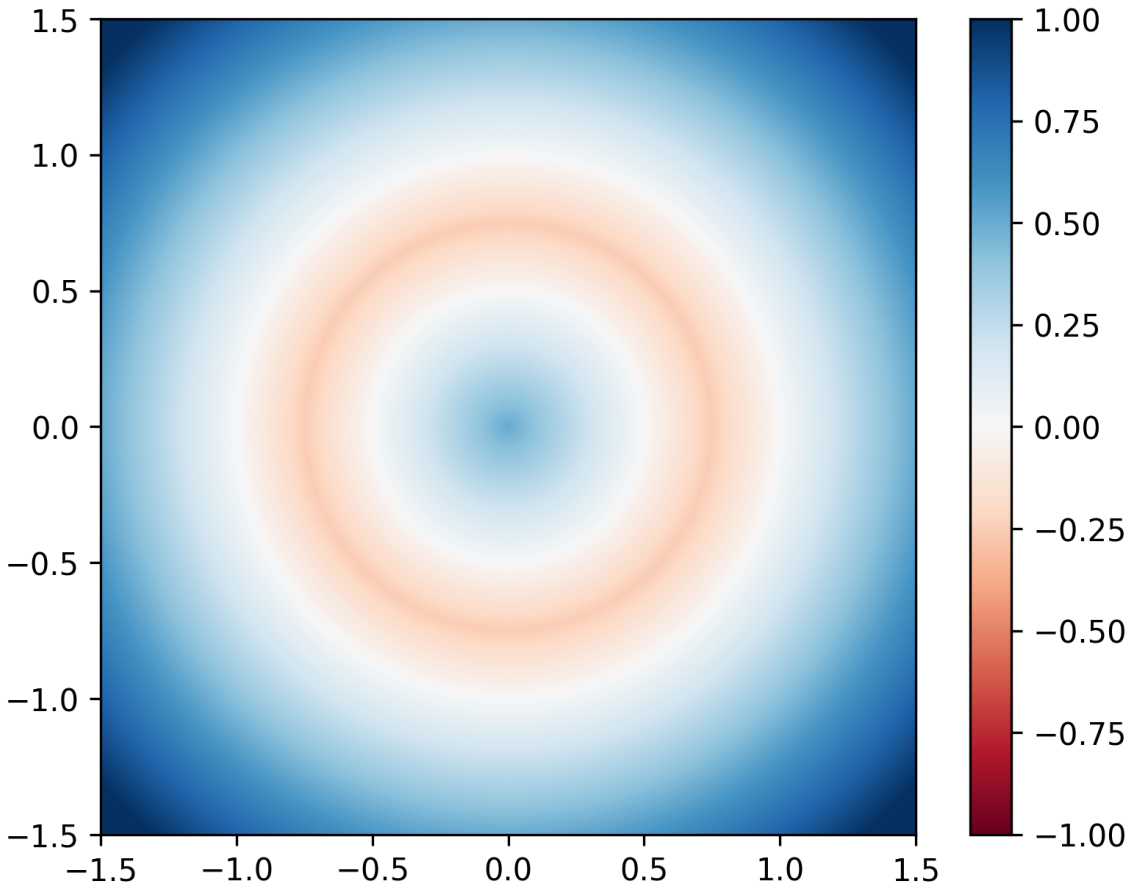
Tape shortening

Interpreter on the GPU

Interval arithmetic

Tape shortening

$$\max\left(0.5 - \sqrt{x^2 + y^2}, \sqrt{x^2 + y^2} - 1\right) < 0$$



Compiled kernel

- Uses hardware registers
- Very fast!
- Longer initial startup (must compile one shader per shape)
- Harder to specialize

Interpreted shape

- Uses soft registers (“slots”)
- Not as fast!
- Faster startup (build tape and send data to GPU)
- Easy to specialize at runtime

Compiled kernel

- Uses hardware registers
- Very fast!
- Longer initial startup (must compile one shader per shape)
- Harder to specialize

Interpreted shape

- Uses soft registers (“slots”)
- Not as fast!
- Faster startup (build tape and send data to GPU)
- **Easy to specialize at runtime**

THE INSTRUCTION TAPE

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

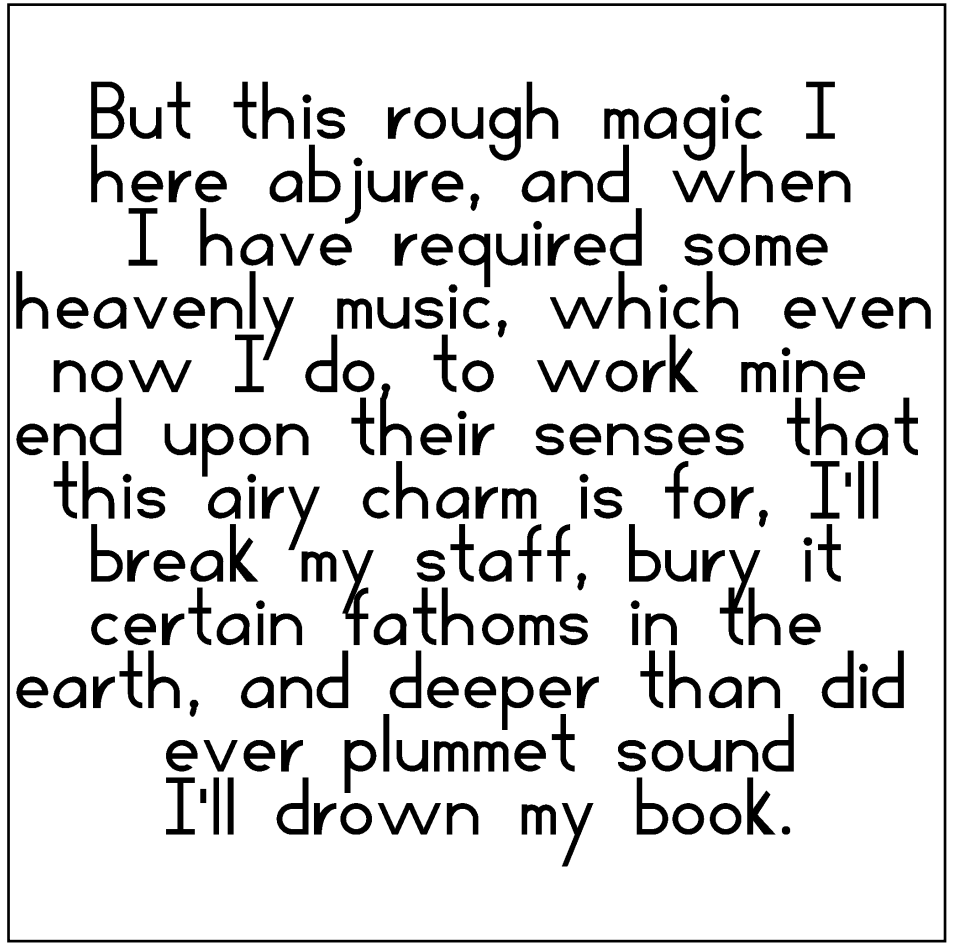
opcode	lhs	rhs	out
X	–	–	slot 0
Y	–	–	slot 1
SQUARE	slot 0	–	slot 0
SQUARE	slot 1	–	slot 1
ADD	slot 0	slot 1	slot 1
SQRT	slot 1	–	slot 1
SUB	slot 1	1.0f	slot 0
SUB	0.5f	slot 1	slot 1
MAX	slot 0	slot 1	slot 1

THE INTERPRETER LOOP

```
def run(tape):
  for (opcode, lhs, rhs, out) in tape:
    match opcode:
      ADD: slots[out] = get(lhs) + get(rhs)
      SUB: slots[out] = get(lhs) - get(rhs)
      MUL: slots[out] = get(lhs) * get(rhs)
      DIV: slots[out] = get(lhs) / get(rhs)
      ...other opcodes here...
  return slots[tape[-1].out]

def get(v):
  return v.is_constant()
    ? v.constant_value()
    : slots[v.slot_index()]
```

INTERPRETER OVERHEAD

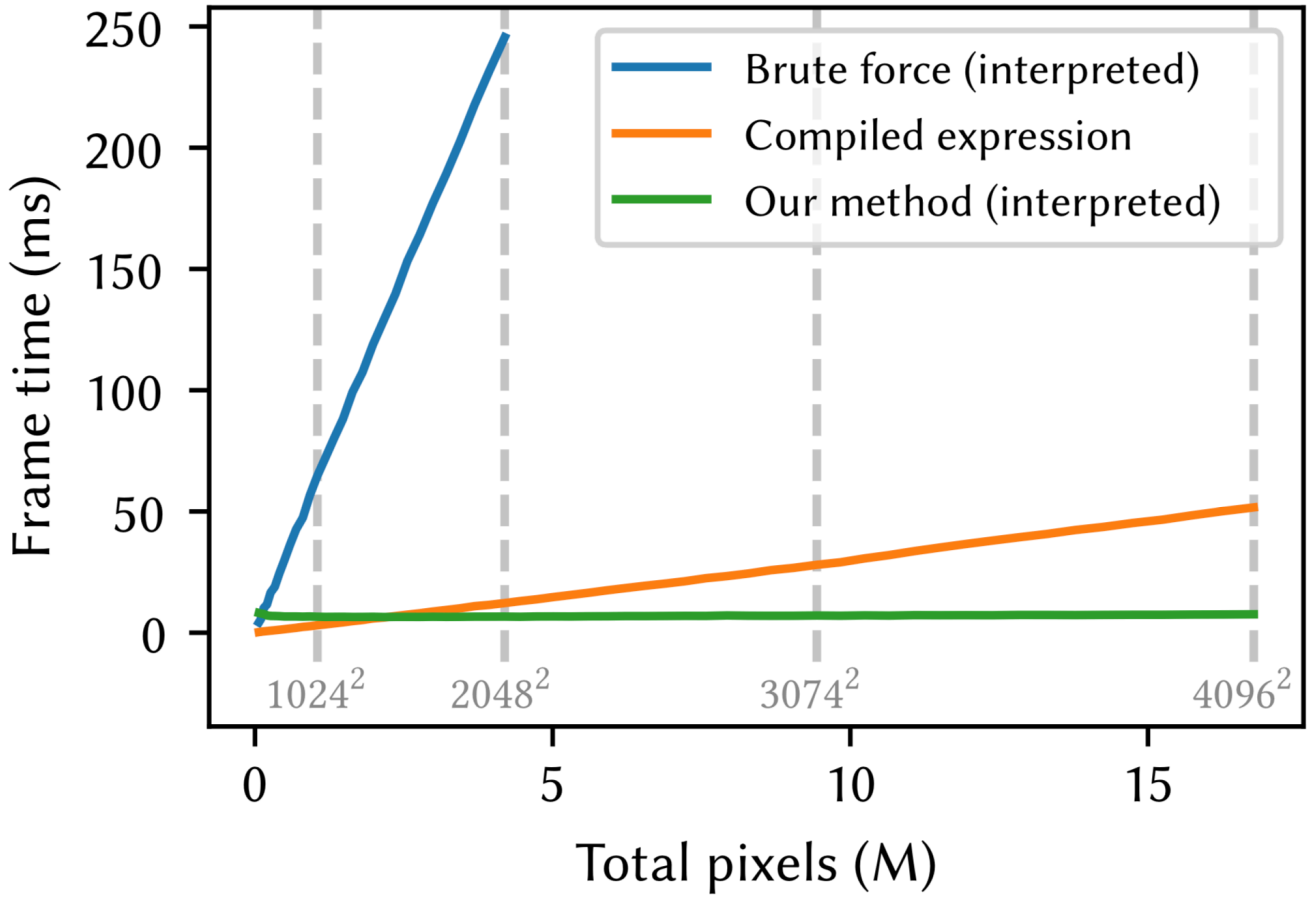


6056 clauses

INTERPRETER OVERHEAD

But this rough magic I here abjure, and when I have required some heavenly music, which even now I do, to work mine end upon their senses that this airy charm is for, I'll break my staff, bury it certain fathoms in the earth, and deeper than did ever plummet sound I'll drown my book.

6056 clauses



GPU PERFORMANCE NOTES

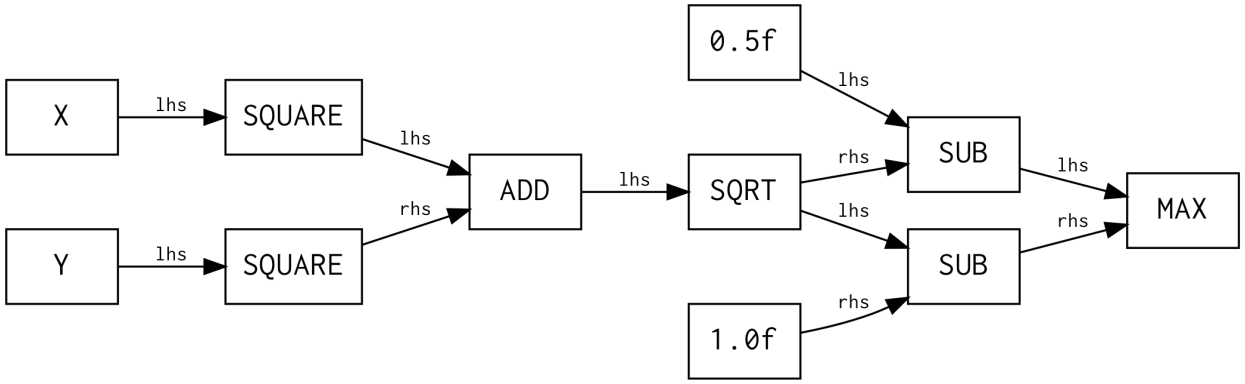
- Implemented in CUDA C/C++, compiled with `nvcc`
- Must use fewer than 32 registers to reach 100% occupancy!
  - Keep inner loop very simple
  - Consider moving setup logic to a separate kernel
  - Prefer C-style over C++-style CUDA
  - Recompile and check after even small changes; the compiler works in mysterious ways
- Avoid thread divergence within a warp
  - 32 threads per warp → 64x subdivision at each level is a convenient multiple
  - Ensure that all threads in a warp are executing the same **tape**

WHERE DOES THE TAPE COME FROM?

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

opcode	lhs	rhs	out
X	–	–	slot 0
Y	–	–	slot 1
SQUARE	slot 0	–	slot 0
SQUARE	slot 1	–	slot 1
ADD	slot 0	slot 1	slot 1
SQRT	slot 1	–	slot 1
SUB	slot 1	1.0f	slot 0
SUB	0.5f	slot 1	slot 1
MAX	slot 0	slot 1	slot 1

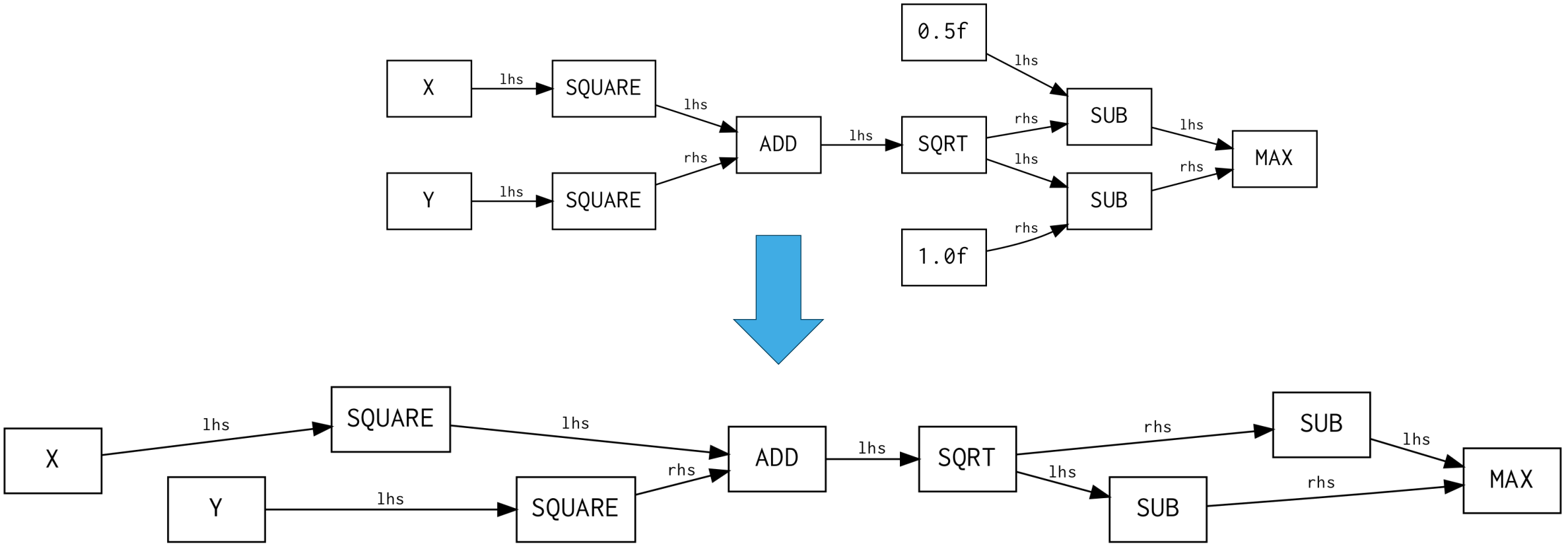
DAG → INSTRUCTION TAPE



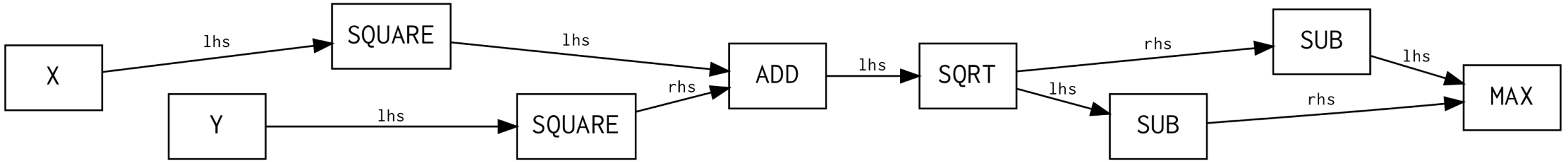
- The equation is encoded as a directed acyclic graph
- Common subexpressions are merged into one node

opcode	lhs	rhs	out
X	–	–	slot 0
Y	–	–	slot 1
SQUARE	slot 0	–	slot 0
SQUARE	slot 1	–	slot 1
ADD	slot 0	slot 1	slot 1
SQRT	slot 1	–	slot 1
SUB	slot 1	1.0f	slot 0
SUB	0.5f	slot 1	slot 1
MAX	slot 0	slot 1	slot 1

TOPOLOGICAL SORT



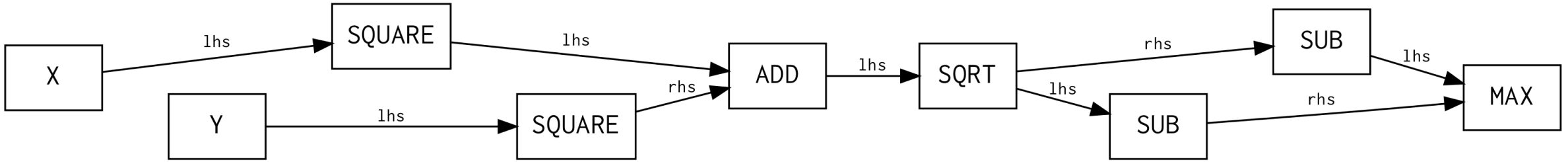
SLOT ALLOCATION



- Same problem as **register allocation** in compiler design
- Simple greedy algorithm:
  - Maintain a pool of available slots
  - After a slot is used for the last time, release it to the pool
  - Choose the top slot in the pool
  - If pool is empty, create a new slot

opcode	lhs	rhs	out
X	–	–	$1
Y	–	–	$2
SQUARE	$1	–	$3
SQUARE	$2	–	$4
ADD	$3	$4	$5
SQRT	$5	–	$6
SUB	$6	1.0f	$7
SUB	0.5f	$6	$8
MAX	$7	$8	$9

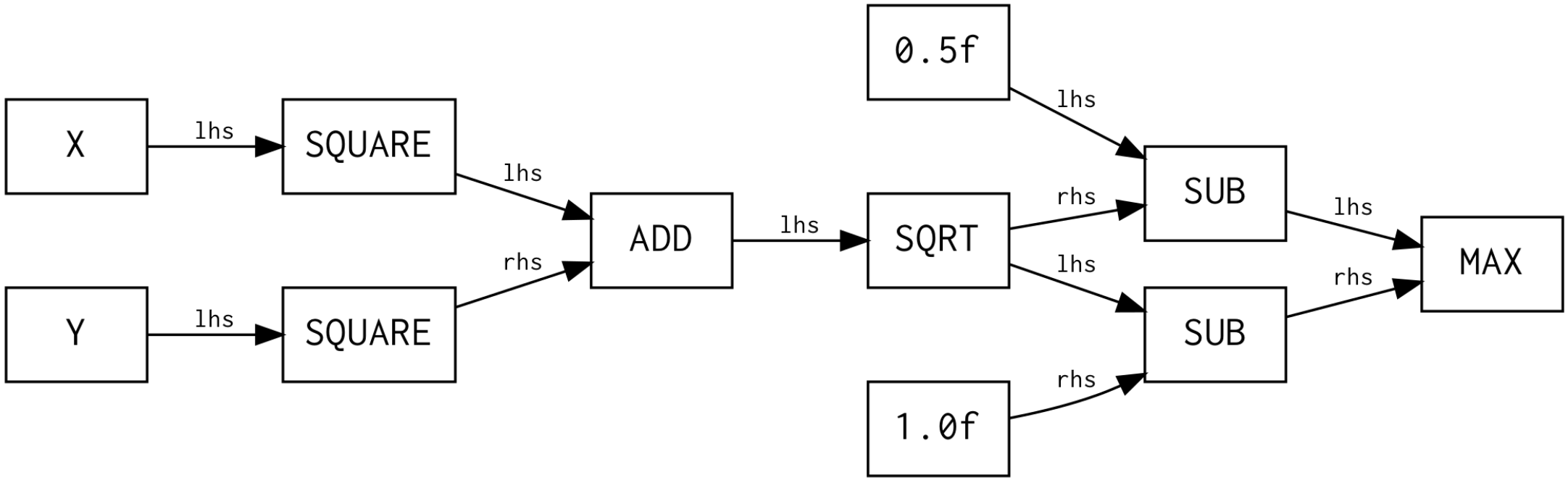
SLOT ALLOCATION



- Same problem as **register allocation** in compiler design
- Simple greedy algorithm:
  - Maintain a pool of available slots
  - After a slot is used for the last time, release it to the pool
  - Choose the top slot in the pool
  - If pool is empty, create a new slot

opcode	lhs	rhs	out
X	–	–	slot 0
Y	–	–	slot 1
SQUARE	slot 0	–	slot 0
SQUARE	slot 1	–	slot 1
ADD	slot 0	slot 1	slot 1
SQRT	slot 1	–	slot 1
SUB	slot 1	1.0f	slot 0
SUB	0.5f	slot 1	slot 1
MAX	slot 0	slot 1	slot 1

WHERE DOES THE DAG COME FROM?



Studio
Desktop design tool

Standard library
Shapes and geometric operations

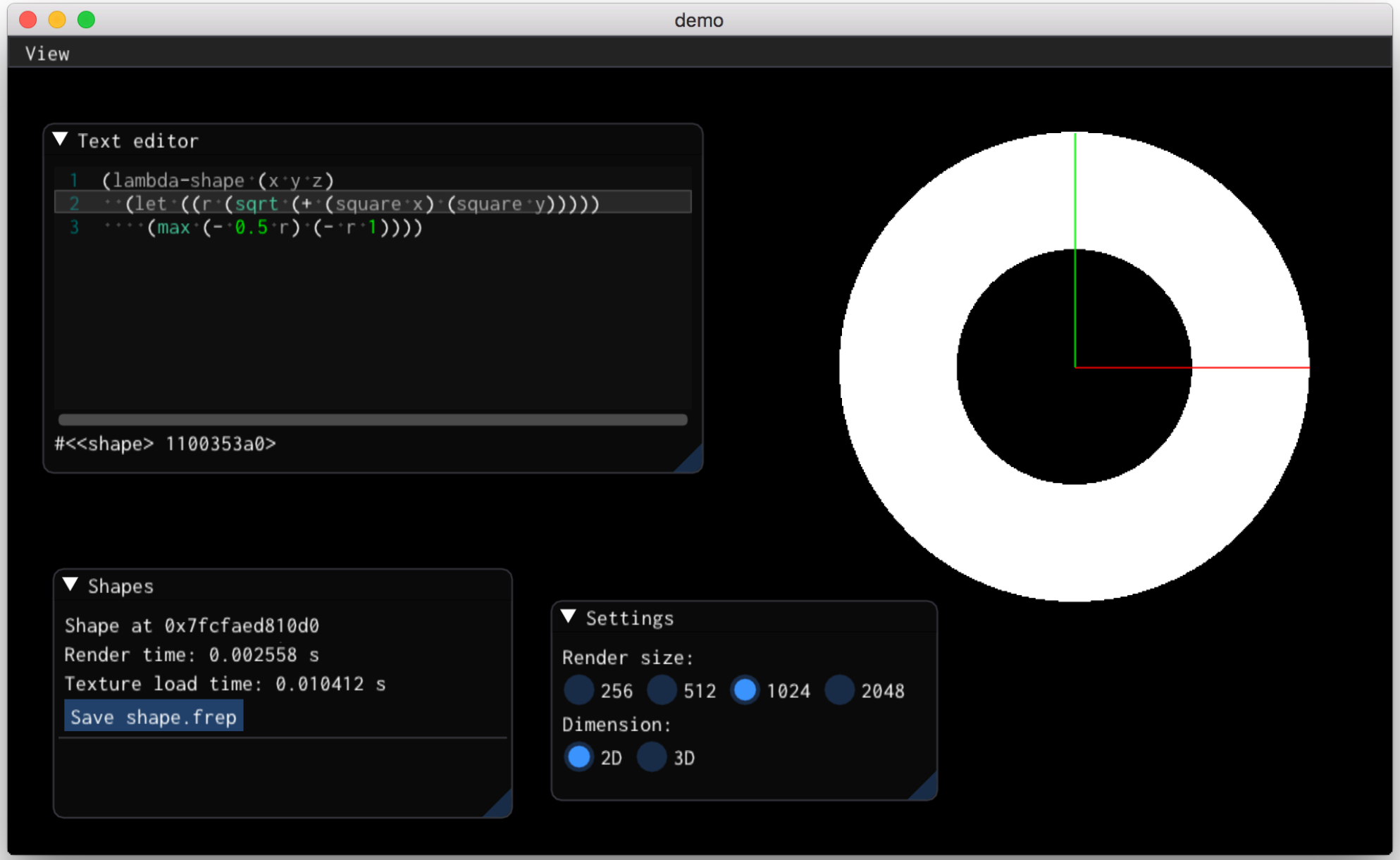
Scheme bindings
`(use-modules (libfive kernel))`

C API
`#include <libfive.h>`

libfive
C++ library

libfive

https://libfive.com


demo
View
Text editor
1 (lambda-shape (x y z)
2 (let ((r (sqrt (+ (square x) (square y)))))
3 (max (- 0.5 r) (- r 1))))
#<<shape> 1100353a0>
Shapes
Shape at 0x7fcfaed810d0
Render time: 0.002558 s
Texture load time: 0.010412 s
Save shape.frep
Settings
Render size:
256
512
1024
2048
Dimension:
2D
3D

Interpreter on the GPU

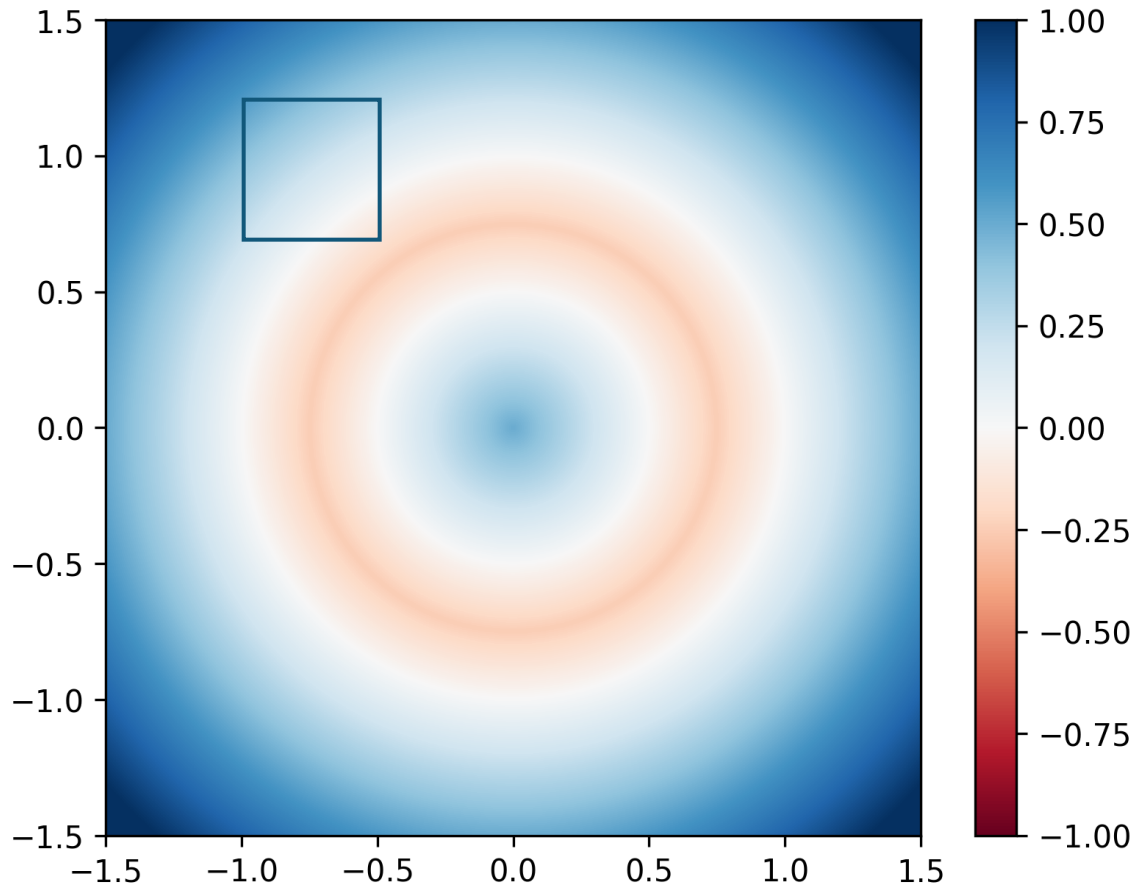
Interval arithmetic

Tape shortening

Interpreter on the GPU

Interval arithmetic

Tape shortening


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

INTERVAL ARITHMETIC

opcode	lhs	rhs	out
X	–	–	[-1.0, -0.5]
Y	–	–	[0.7, 1.2]
SQUARE		–	
SQUARE		–	
ADD			
SQRT		–	
SUB		1.0f	
SUB	0.5f		
MAX			

INTERVAL ARITHMETIC

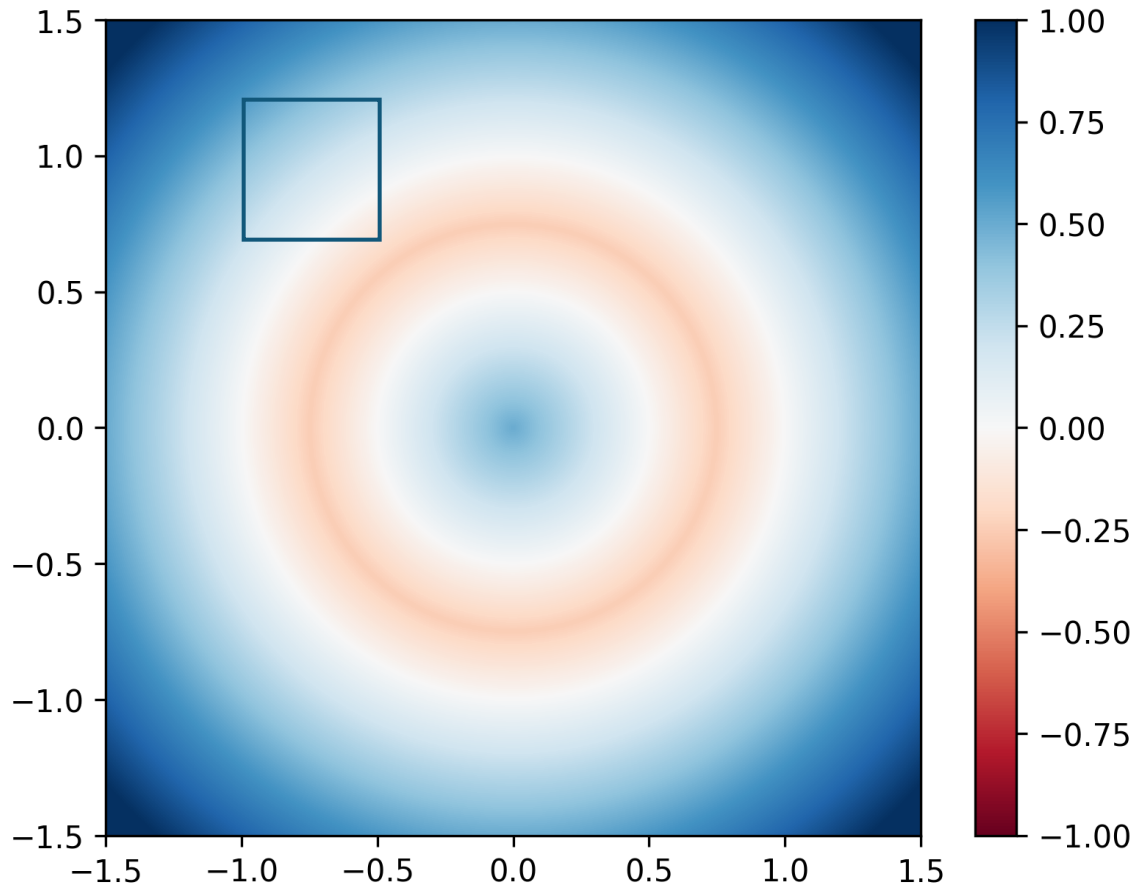
opcode	lhs	rhs	out
X	–	–	[-1.0, -0.5]
Y	–	–	[0.7, 1.2]
SQUARE	[-1.0, -0.5]	–	
SQUARE		–	
ADD			
SQRT		–	
SUB		1.0f	
SUB	0.5f		
MAX			

INTERVAL ARITHMETIC

opcode	lhs	rhs	out
X	–	–	[-1.0, -0.5]
Y	–	–	[0.7, 1.2]
SQUARE	[-1.0, -0.5]	–	[0.25, 1.0]
SQUARE		–	
ADD			
SQRT		–	
SUB		1.0f	
SUB	0.5f		
MAX			

INTERVAL ARITHMETIC

opcode	lhs	rhs	out
X	–	–	[-1.0, -0.5]
Y	–	–	[0.7, 1.2]
SQUARE	[-1.0, -0.5]	–	[0.25, 1.0]
SQUARE	[0.7, 1.2]	–	[0.49, 1.44]
ADD	[0.25, 1.0]	[0.49, 1.44]	[0.74, 2.44]
SQRT	[0.74, 2.44]	–	[0.86, 1.56]
SUB	[0.86, 1.56]	1.0f	[-0.14, 0.56]
SUB	0.5f	[0.86, 1.56]	[-1.06, -0.36]
MAX	[-0.14, 0.56]	[-1.06, -0.36]	[-0.14, 0.56]


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00

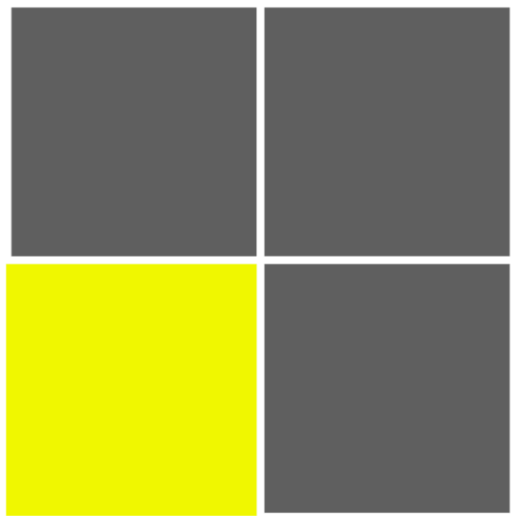
INTERVAL ARITHMETIC RESULTS

`f(X, Y, Z).upper < 0` → inside the shape (“filled”)

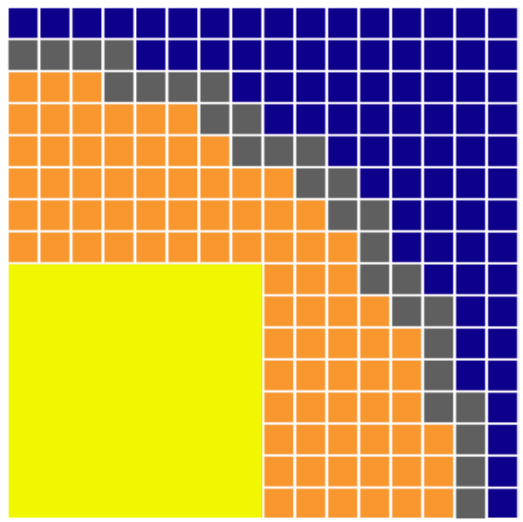
`f(X, Y, Z).lower > 0` → outside the shape (“empty”)

`otherwise` → ambiguous

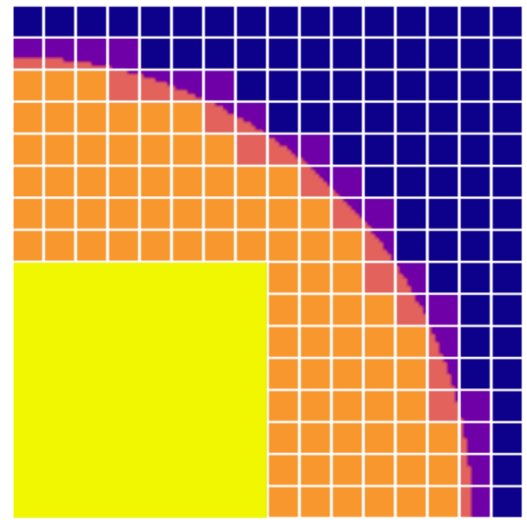
First pass



Second pass



Per-pixel evaluation



GPU IMPLEMENTATION NOTES

- Implemented a GPU interval class based on `boost::interval`
- Using compiler intrinsics to ensure proper rounding modes

```
__device__ inline Interval operator+(const Interval& x,
                                     const Interval& y)
{
    return {__fadd_rd(x.lower(), y.lower()),
            __fadd_ru(x.upper(), y.upper())};
}
```

- Intrinsics are not available on all GPU compute platforms (but it may not matter)

Interpreter on the GPU

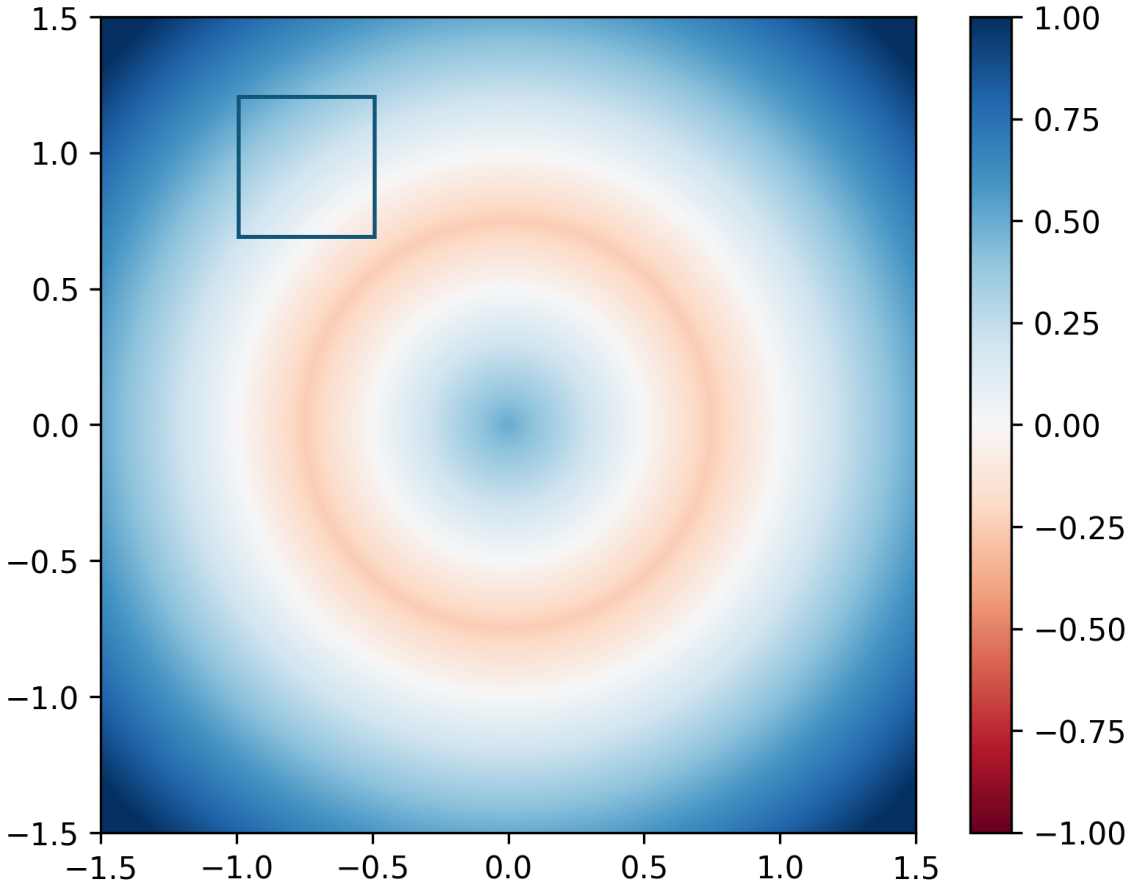
Interval arithmetic

Tape shortening

Interpreter on the GPU

Interval arithmetic

Tape shortening



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

MIN AND MAX

opcode	lhs	rhs	out
X	–	–	[-1.0, -0.5]
Y	–	–	[0.7, 1.2]
SQUARE	[-1.0, -0.5]	–	[0.25, 1.0]
SQUARE	[0.7, 1.2]	–	[0.49, 1.44]
ADD	[0.25, 1.0]	[0.49, 1.44]	[0.74, 2.44]
SQRT	[0.74, 2.44]	–	[0.86, 1.56]
SUB	[0.86, 1.56]	1.0f	[-0.14, 0.56]
SUB	0.5f	[0.86, 1.56]	[-1.06, -0.36]
MAX	[-0.14, 0.56]	[-1.06, -0.36]	[-0.14, 0.56]

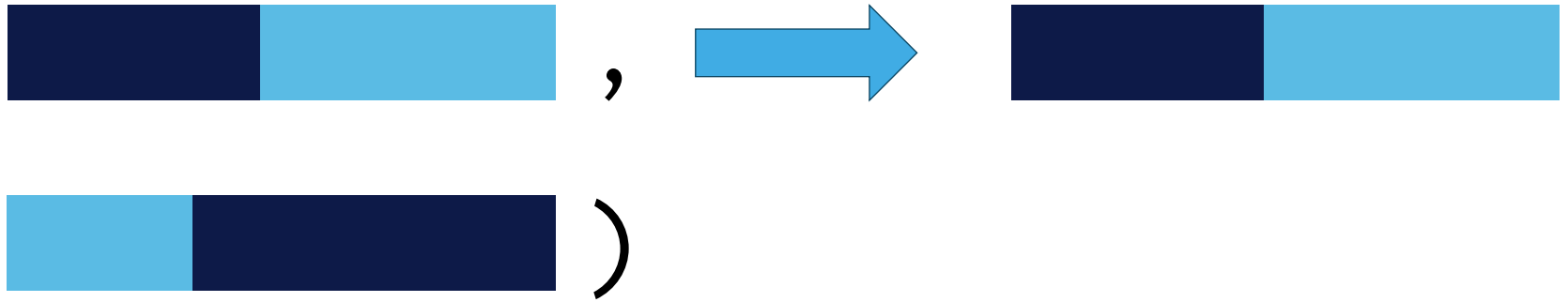
MIN AND MAX

max( ,

)

MIN AND MAX

max(,)



REPLACING WITH COPY

opcode	lhs	rhs	out
X	–	–	slot 0
Y	–	–	slot 1
SQUARE	slot 0	–	slot 0
SQUARE	slot 1	–	slot 1
ADD	slot 0	slot 1	slot 1
SQRT	slot 1	–	slot 1
SUB	slot 1	1.0f	slot 0
SUB	0.5f	slot 1	slot 1
COPY	slot 0	–	slot 1

NOTICING INACTIVE CLAUSES

opcode	lhs	rhs	out
X	–	–	slot 0
Y	–	–	slot 1
SQUARE	slot 0	–	slot 0
SQUARE	slot 1	–	slot 1
ADD	slot 0	slot 1	slot 1
SQRT	slot 1	–	slot 1
SUB	slot 1	1.0f	slot 0
~~SUB~~	~~0.5f~~	~~slot 1~~	~~slot 1~~
COPY	slot 0	–	slot 1

NOTICING INACTIVE CLAUSES

opcode	lhs	rhs	out
X	–	–	slot 0
Y	–	–	slot 1
SQUARE	slot 0	–	slot 0
SQUARE	slot 1	–	slot 1
ADD	slot 0	slot 1	slot 1
SQRT	slot 1	–	slot 1
SUB	slot 1	1.0f	slot 0
COPY	slot 0	–	slot 1

TAPE SHORTENING

- Analogous to *mark-and-sweep* garbage collection
- Mark output slot as **active**, all other slots as **inactive**
- Walk backwards through the tape, skipping clauses with **inactive** output slots
- For all clauses **except** `MIN` and `MAX`
  - Mark the output slot as **inactive**
  - Mark *all* input slots as **active**
- For `MIN` and `MAX` clauses:
  - Mark the output slot as **inactive**
  - If unambiguous, only mark *one* input output slot as **active**, and replace `MIN/MAX` with `COPY`
  - Otherwise, mark *all* input slots as **active**

TAPE SHORTENING

- Use interval results at `min`/`max` clauses to detect clauses that are inactive in a particular region
- Construct a shortened tape without those clauses
- That tape is valid for all subregions contained within the parent region

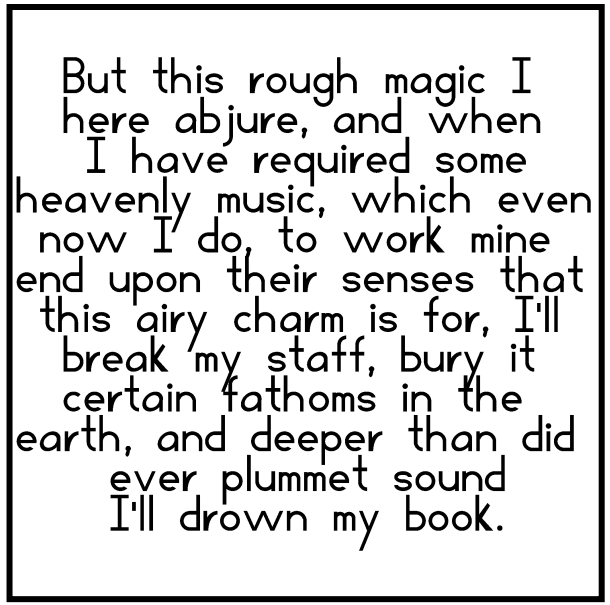
Algorithm 1: Evaluate a tape, recording which branch is taken at every `min` and `max` node

```
choices ← an empty stack
foreach clause in tape do
    lhs ← getValue(clause.lhs)
    rhs ← getValue(clause.rhs)
    switch clause.opcode do
        case OP_MIN
            if lhs.upper < rhs.lower then
                choices.push(CHOICE_LHS)
            else if rhs.upper < lhs.lower then
                choices.push(CHOICE_RHS)
            else
                choices.push(CHOICE_BOTH)
            slots[clause.out] ← min(lhs, rhs)
        case OP_MAX
            Similar logic to push a choice slots[clause.out]
            ← max(lhs, rhs)
        case OP_ADD
            slots[clause.out] ← lhs + rhs
        case OP_SUB
            ...and so on for other opcodes
clause ← the last clause in the tape
return (slots[clause.out], choices)
```

Algorithm 2: Use the choice data from Alg. 1 to construct a shorter tape which only contains active clauses

```
output ← an empty tape
active ← an array of all false
active[final output slot] ← true
foreach clause in tape.reversed() do
    if clause.opcode ∈ [OP_MIN, OP_MAX] then
        choice ← choices.pop()
    else
        choice ← CHOICE_BOTH
    if active[clause.out] then
        active[clause.out] ← false
        if choice == CHOICE_LHS then
            active[clause.lhs] ← true
            clause.rhs ← clause.lhs
        else if choice == CHOICE_RHS then
            active[clause.rhs] ← true
            clause.lhs ← clause.rhs
        else
            active[clause.lhs] ← true
            active[clause.rhs] ← true
        output.push_back(clause)
return output
```

HOW WELL DOES THIS WORK?

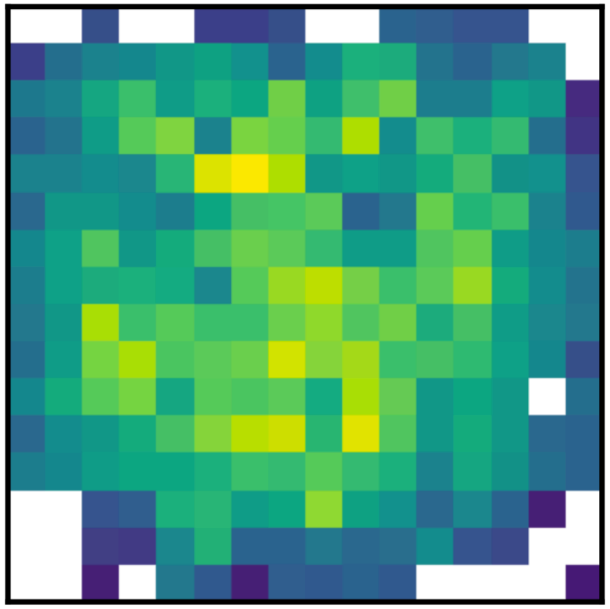


Original: 6056 clauses
(rendered at 1024x1024)

HOW WELL DOES THIS WORK?

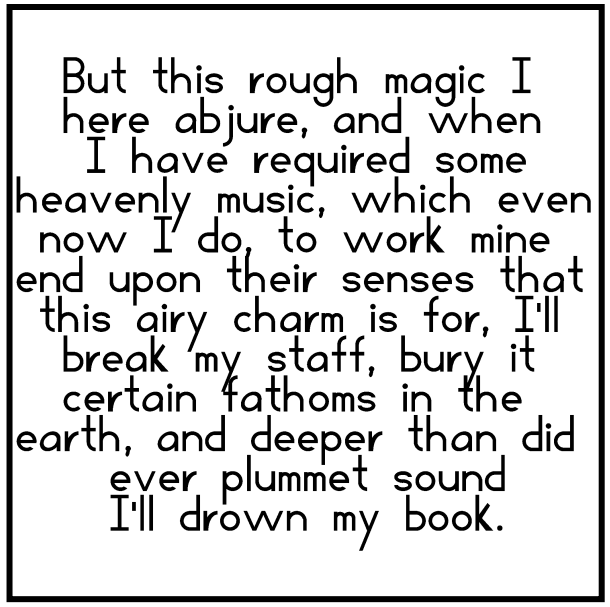
But this rough magic I here abjure, and when I have required some heavenly music, which even now I do, to work mine end upon their senses that this airy charm is for, I'll break my staff, bury it certain fathoms in the earth, and deeper than did ever plummet sound I'll drown my book.

Original: 6056 clauses
(rendered at 1024x1024)

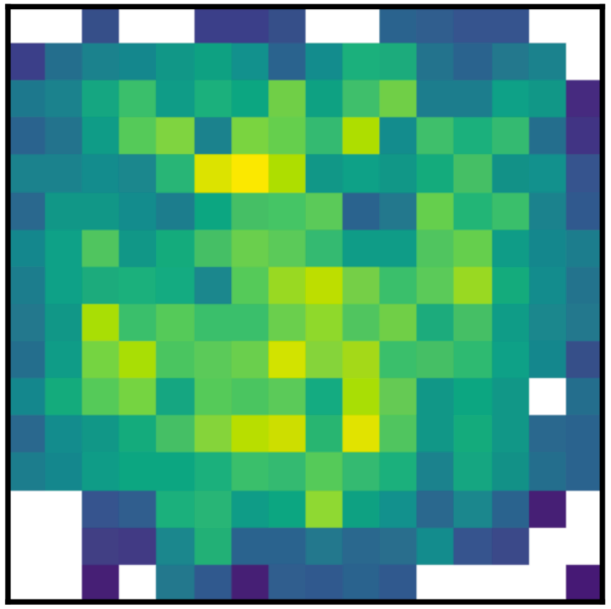


356 ± 125 clauses

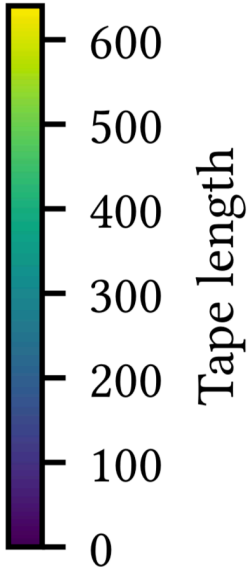
HOW WELL DOES THIS WORK?



Original: 6056 clauses
(rendered at 1024x1024)



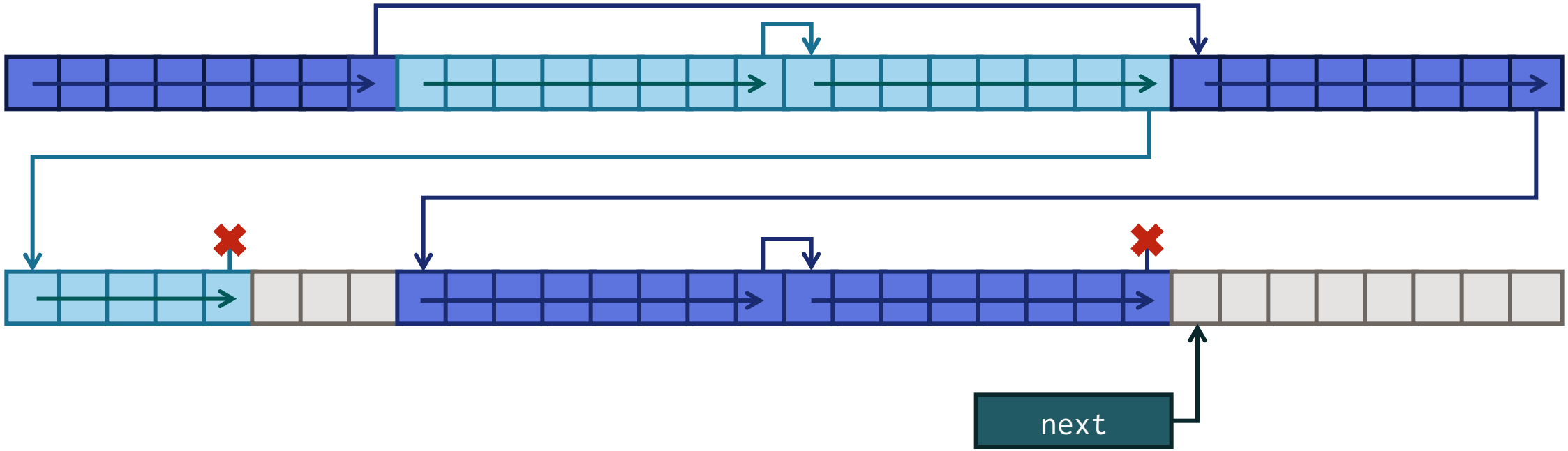
356 ± 125 clauses



28 ± 13 clauses

STORING TAPES ON THE GPU

- Unrolled linked lists (64 clauses per chunk)
- Stored in a large (> 1 GB) pre-allocated array in **global** memory
- Threads claim memory by bumping an atomic integer



Interpreter on the GPU

Interval arithmetic

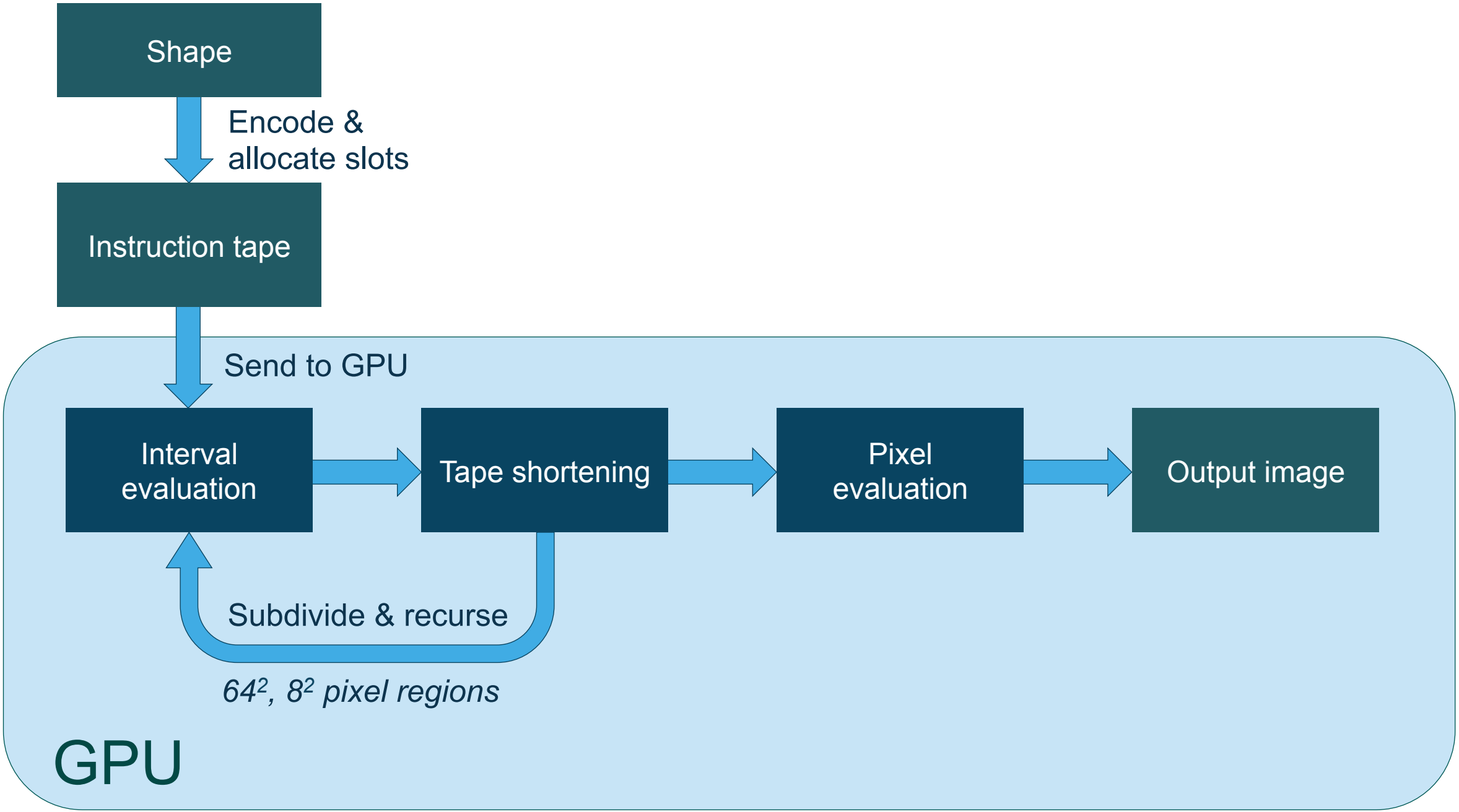
Tape shortening

Interpreter on the GPU

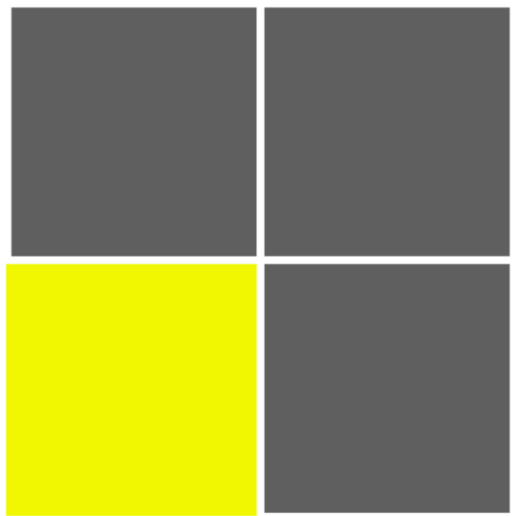
Interval arithmetic

Tape shortening

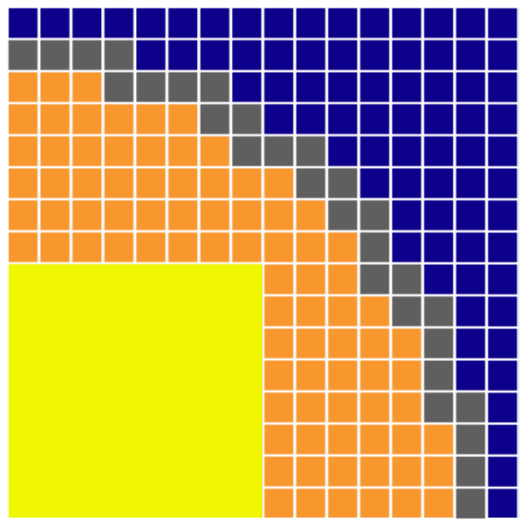
PUTTING IT ALL TOGETHER!


Shape
Encode &
allocate slots
Instruction tape
Send to GPU
Interval
evaluation
Tape shortening
Pixel
evaluation
Output image
Subdivide & recurse
64², 8² pixel regions
GPU

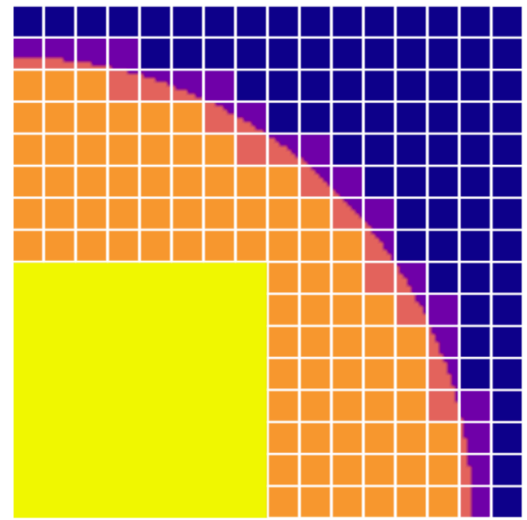
First pass



Second pass



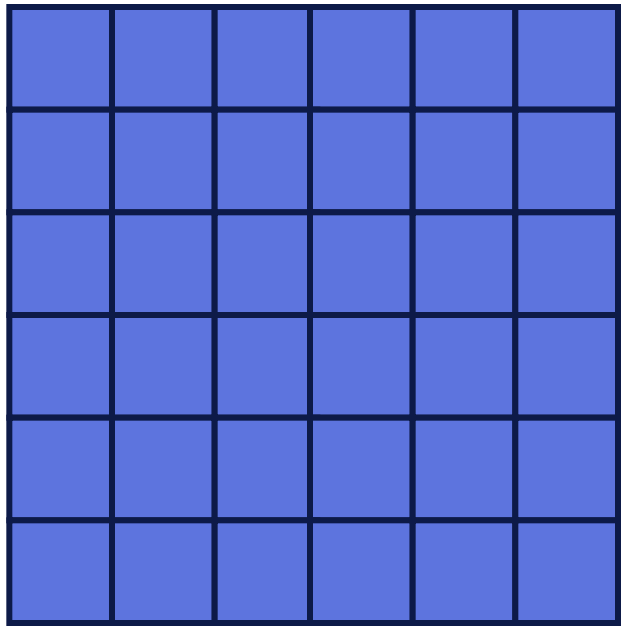
Per-pixel evaluation



2D RENDERING PROCEDURE

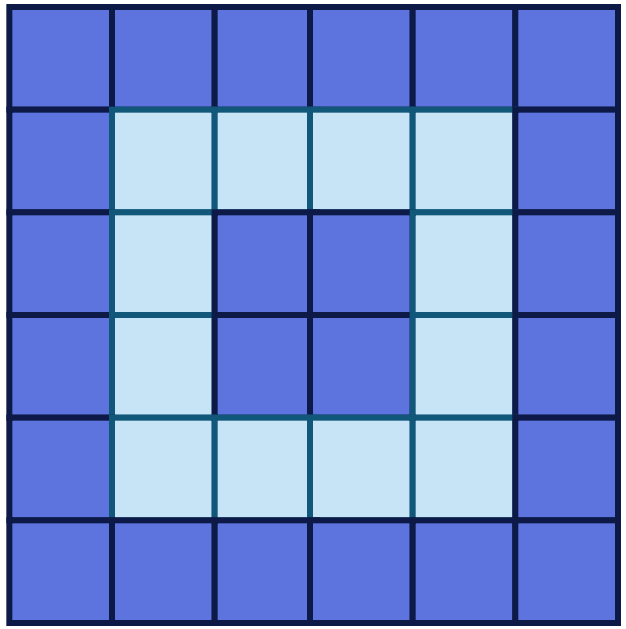
- Interval evaluation of 64x64-pixel **tiles**
  - Record empty and filled tiles, but do nothing more with them
  - For ambiguous tiles, calculate shortened tape
  - Collect ambiguous tiles into a dense list for further evaluation
- Split ambiguous tiles into 64 **subtiles** (8x8-pixels) each
- Interval evaluation of subtiles
  - Record empty and filled subtiles, but do nothing more with them
  - For ambiguous subtiles, calculate shortened tape
  - Collect ambiguous subtiles into a dense list for pixel evaluation
- Per-**pixel** evaluations of ambiguous subtiles
  - Same interpreter loop as interval evaluation, but using single values instead

MEMORY LAYOUT



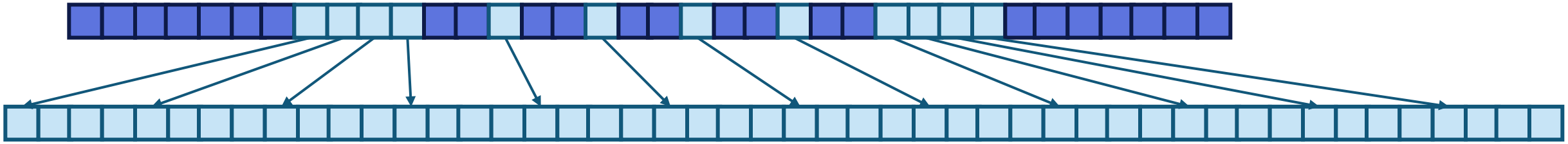
- Largest tiles are **densely packed**
  - $(N / 64)^D$ tiles, where N is image size and D is dimension
- Each tile contains three values
  - Tile position (packed 32-bit int, with 10 bits each for x/y/z)
  - Index of specialized tape for this tile (-1 by default)
  - Index of subdivided region for this tile (-1 by default)

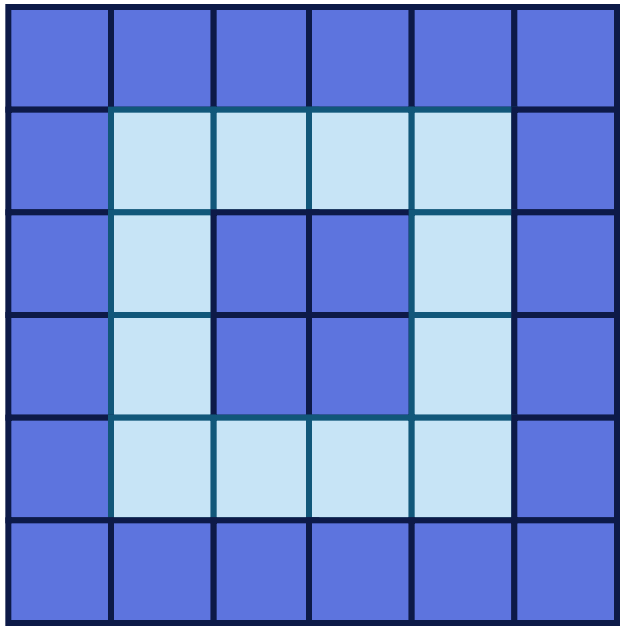
MEMORY LAYOUT

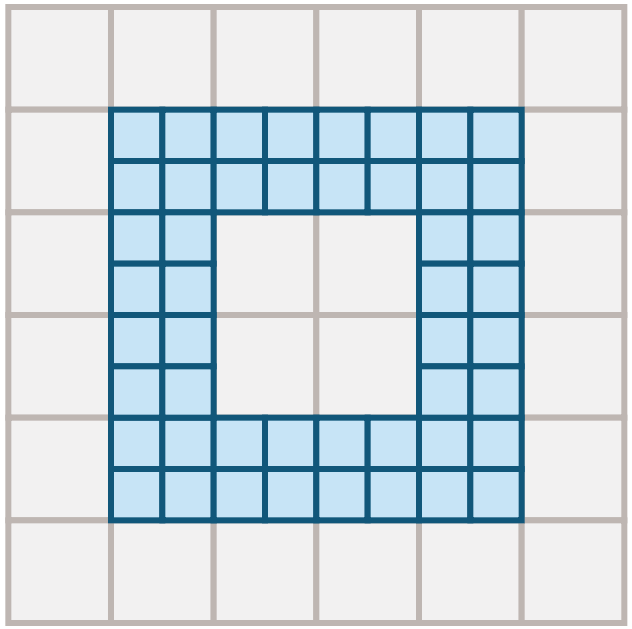


- Interval evaluation picks out ambiguous tiles
- These tiles will be subdivided and evaluated again
- Subdivided tiles are **sparsely distributed** in space but **densely packed** in GPU RAM

MEMORY LAYOUT

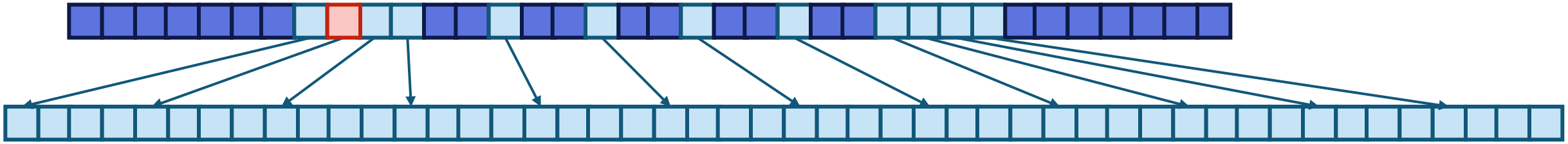


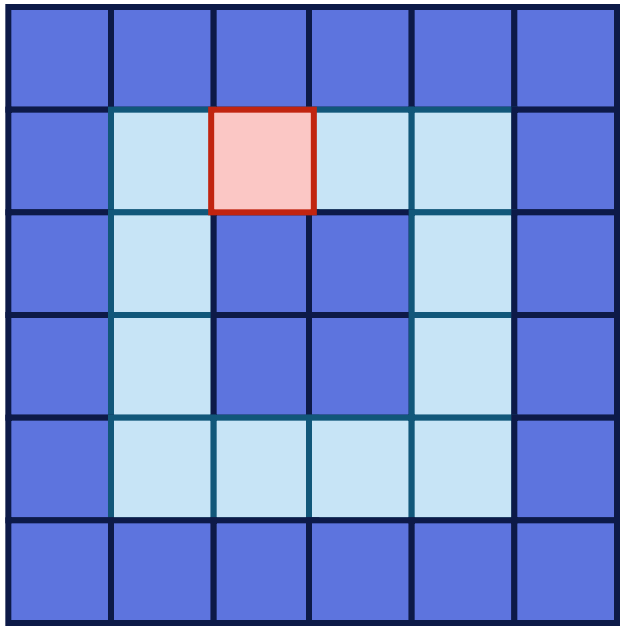


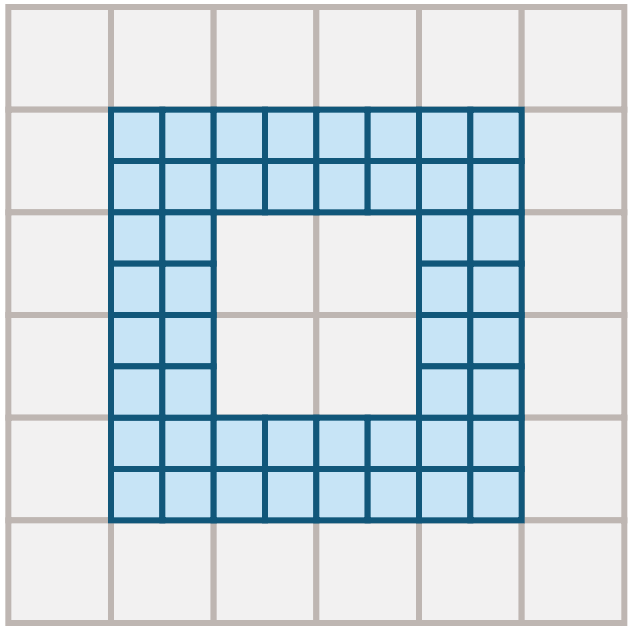


(Diagram shows 4x subdivision,
but we use 64x in practice)

STORING THE HIERARCHY

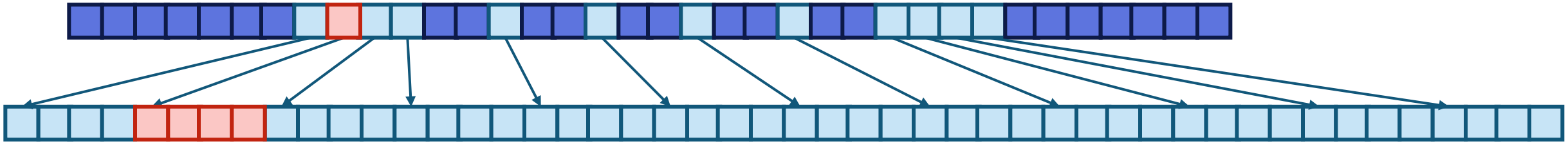


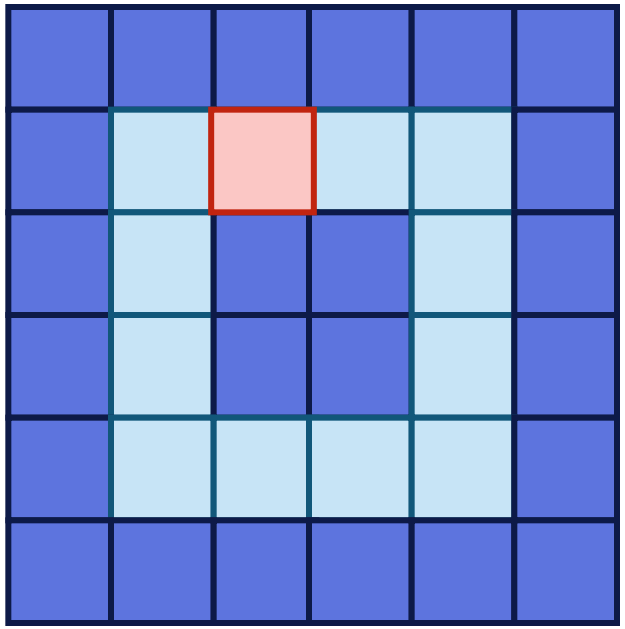


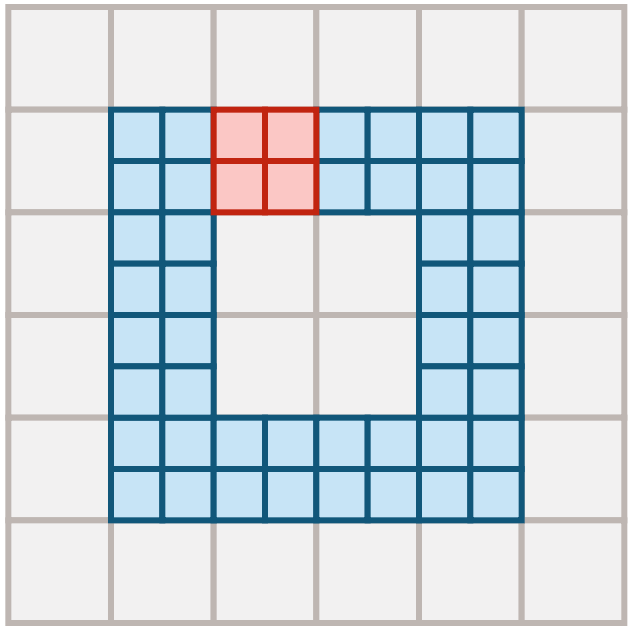


(Diagram shows 4x subdivision,
but we use 64x in practice)

MEMORY LAYOUT

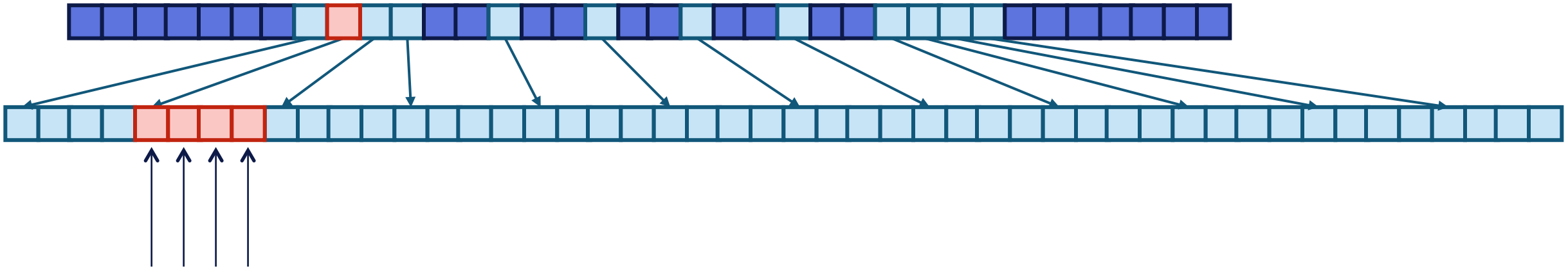






(Diagram shows 4x subdivision,
but we use 64x in practice)

MEMORY LAYOUT



64 subtiles = 64 threads = 2 warps

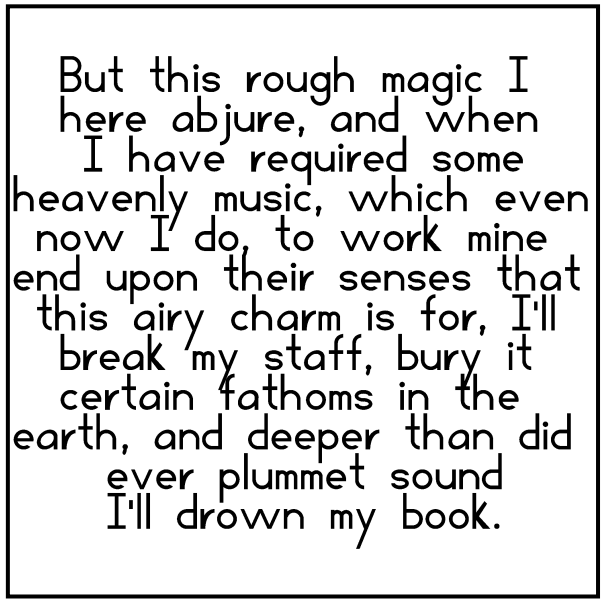
All threads within these warps use the same shortened tape (from parent tile), to minimize divergence

BENCHMARKING MACHINES

GeForce GT 750M
2013 MacBook Pro

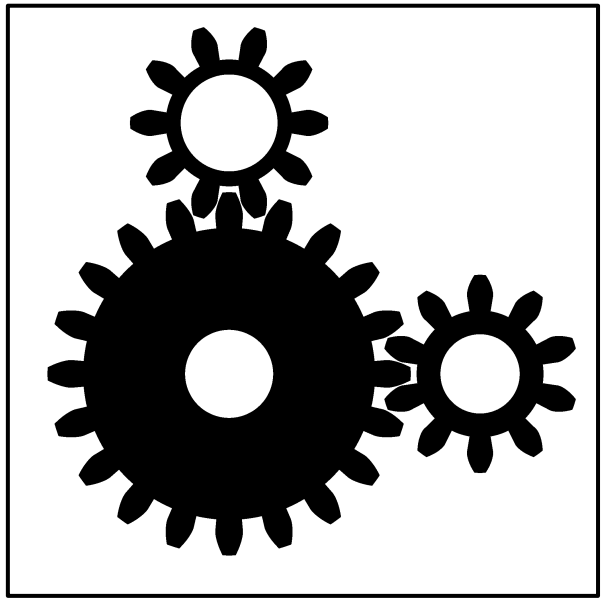
GTX 1080 Ti
2017 VR/ML workstation

Tesla V100
AWS `p3.2xlarge`



Text benchmark

- 6056 clauses (2354 min/max)
- Shows off CSG and tape pruning performance



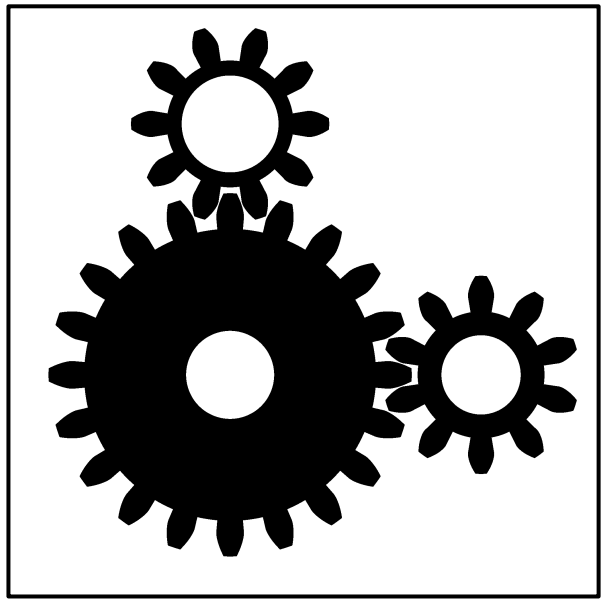
2D Gears

- 1735 clauses (374 min/max)
- Curves are mathematically correct involutes, which requires transcendental functions

But this rough magic I here abjure, and when I have required some heavenly music, which even now I do, to work mine end upon their senses that this airy charm is for, I'll break my staff, bury it certain fathoms in the earth, and deeper than did ever plummet sound I'll drown my book.

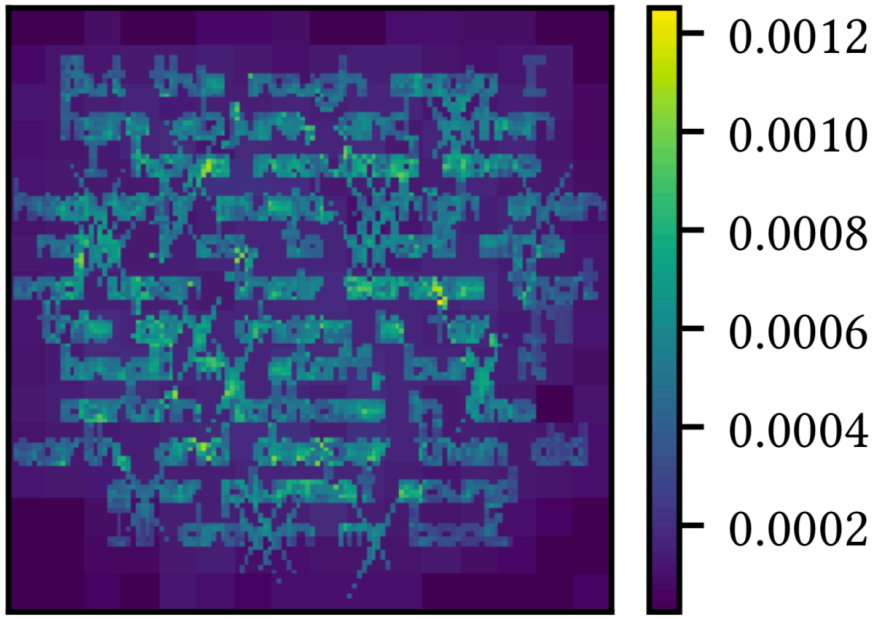
Frame time (ms)

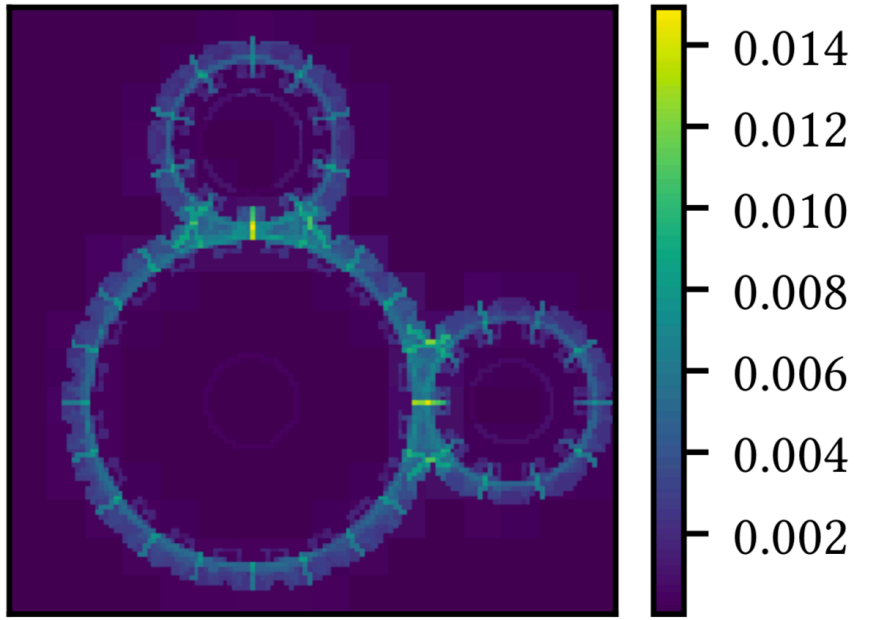
Size	GT 750M	GTX 1080 Ti	Tesla V100
256^2	17.5	8.3	5.2
512^2	14.8	6.8	4.2
1024^2	16.5	6.5	3.9
2048^2	20.7	6.6	3.9
3072^2	27.1	6.9	3.9
4096^2	35.9	7.4	4.1



Frame time (ms)

Size	GT 750M	GTX 1080 Ti	Tesla V100
256^2	9.2	4.0	2.8
512^2	9.3	3.7	2.5
1024^2	12.1	3.4	2.2
2048^2	17.3	3.4	2.2
3072^2	23.4	3.7	2.3
4096^2	30.6	4.0	2.4

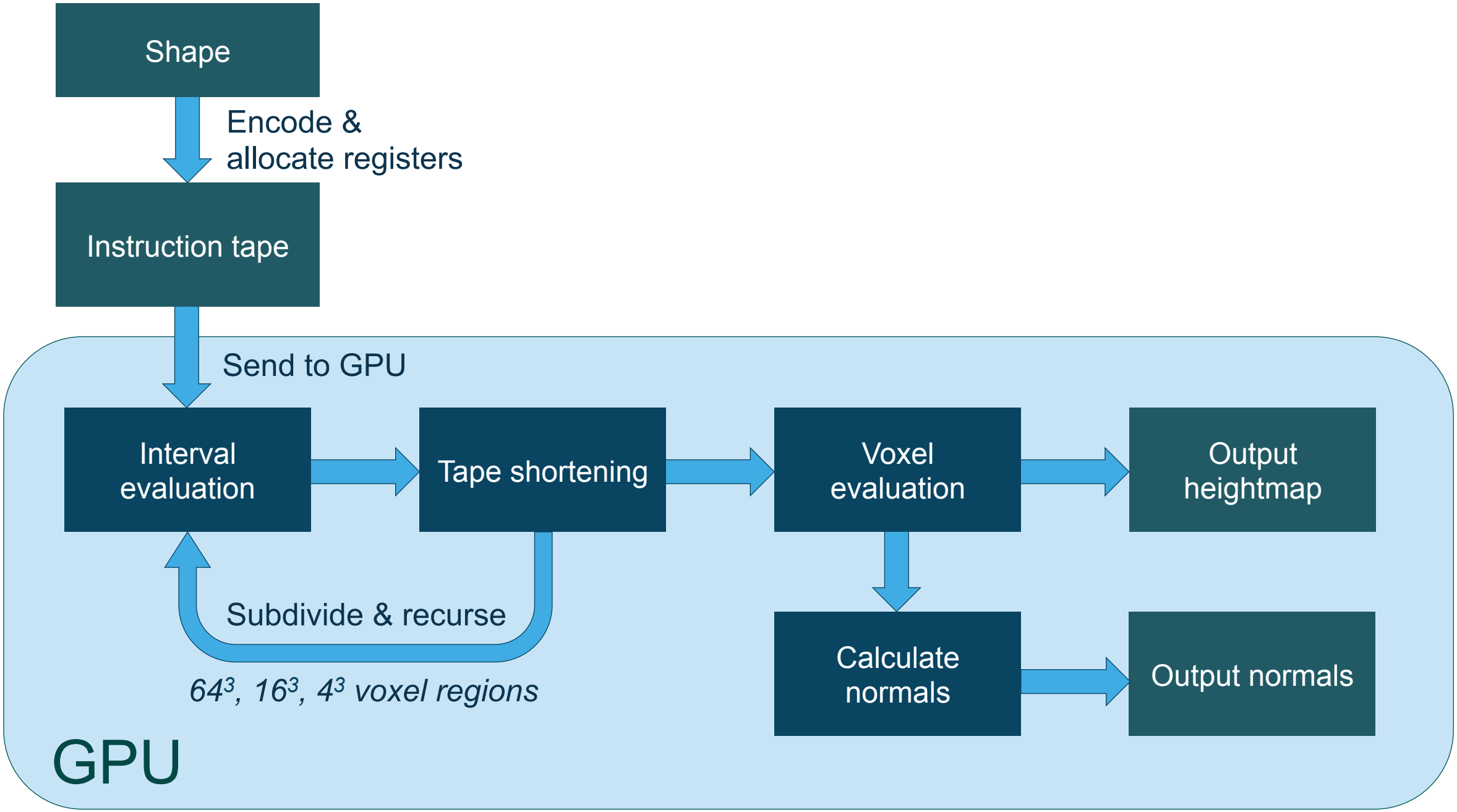




Metric: Normalized + amortized work

- A score of 1 means that the tape is fully walked once for that pixel
- Interval evaluation work is amortized across all pixels in the interval
- Less than 1 full-tape evaluation per pixel!

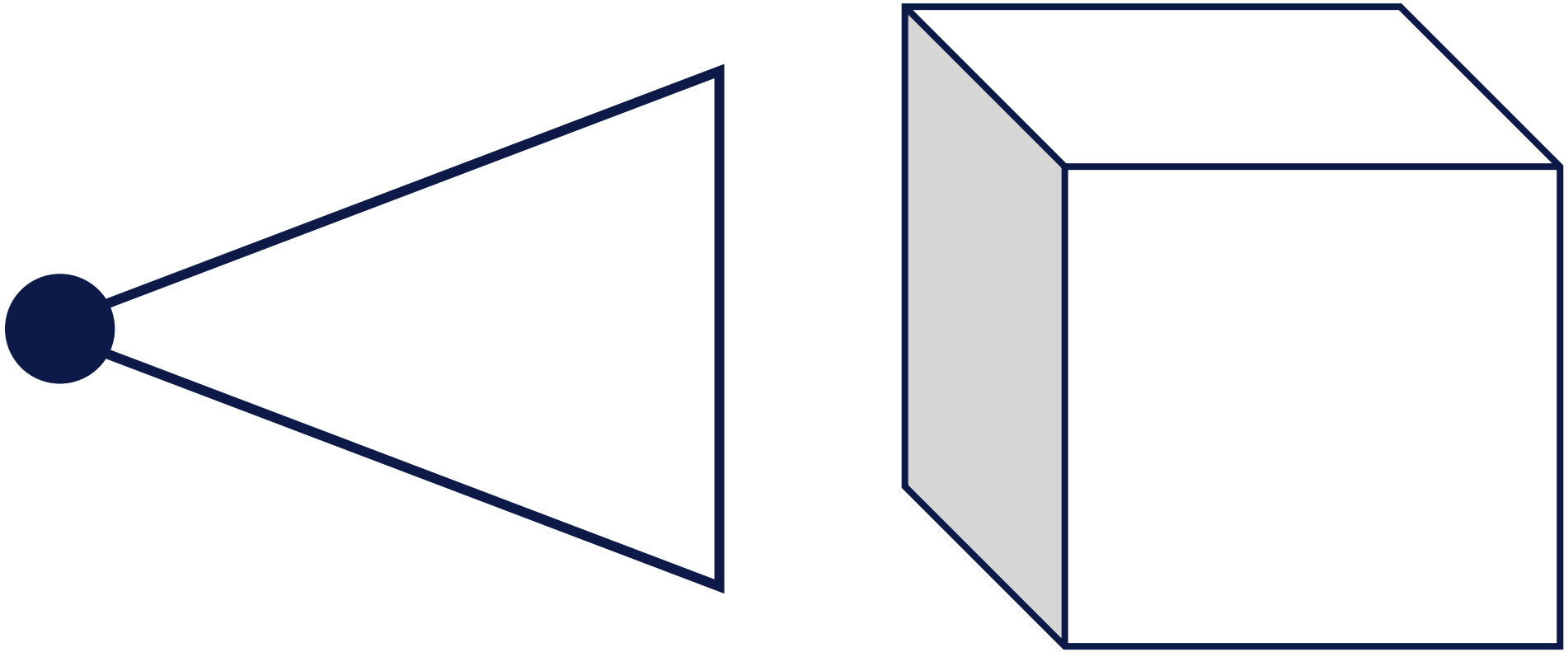
RENDERING IN 3D


Shape
Encode &
allocate registers
Instruction tape
Send to GPU
Interval
evaluation
Tape shortening
Voxel
evaluation
Output
heightmap
Subdivide & recurse
64^3, 16^3, 4^3 voxel regions
Calculate
normals
Output normals
GPU

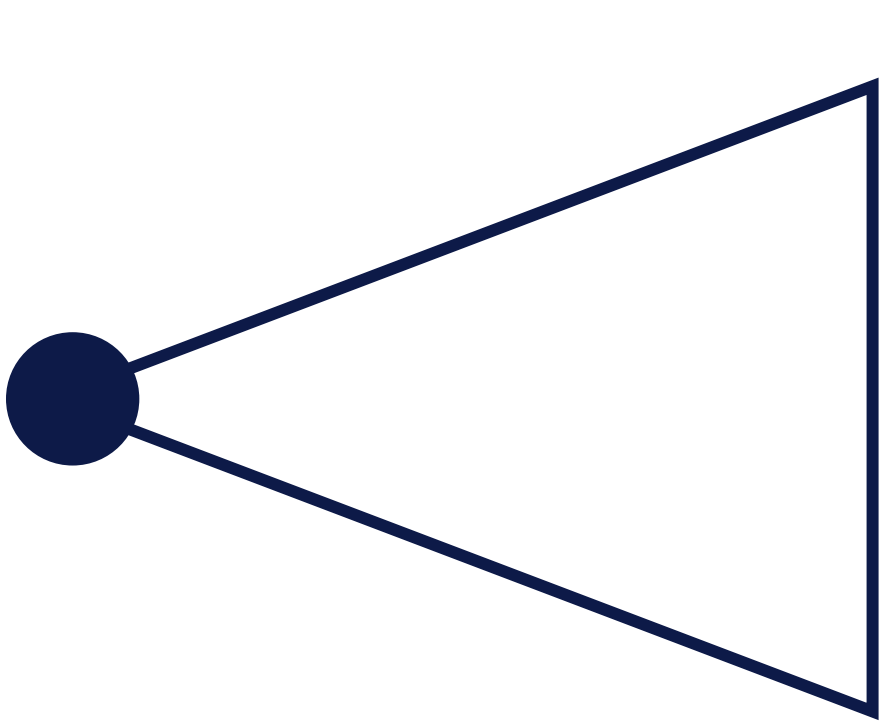
3D RENDERING PROCEDURE

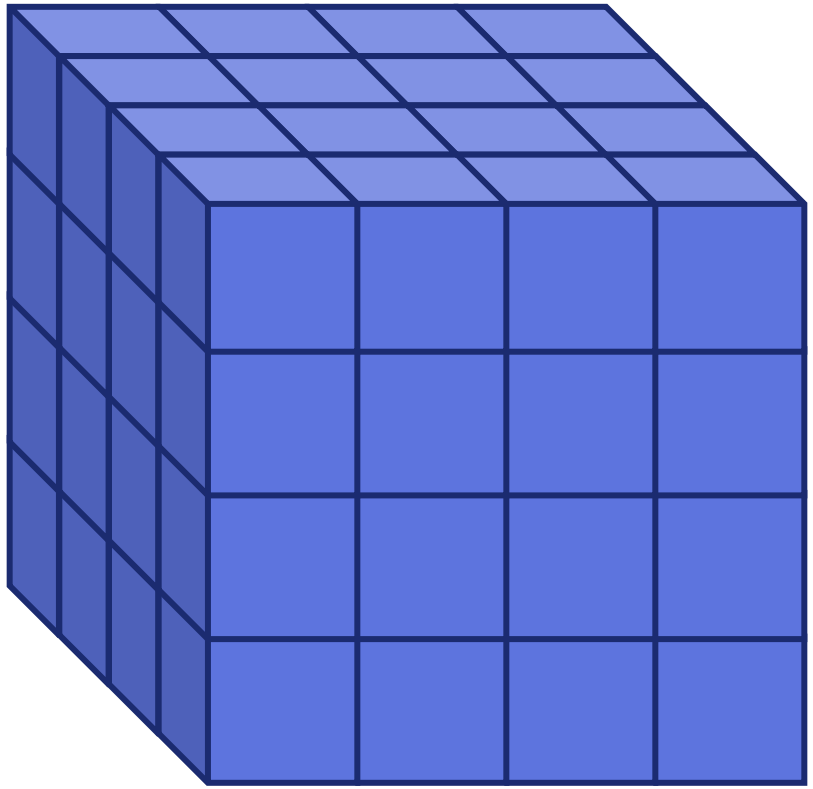
- Interval evaluation of 64x64x64-voxel **tiles**
  - Split ambiguous tiles into 64 **subtiles** (16x16x16-voxels) each
- Interval evaluation of **subtiles**
  - Split ambiguous subtiles into 64 **microtiles** (4x4x4-voxels) each
- Interval evaluation of **microtiles**
- Per-**voxel** evaluations of remaining ambiguous **microtiles**
- Per-**pixel** evaluation, using automatic differentiation to find normals
- *Optional:* post-processing depth + normal buffer to generate final image

RENDERING A HEIGHTMAP

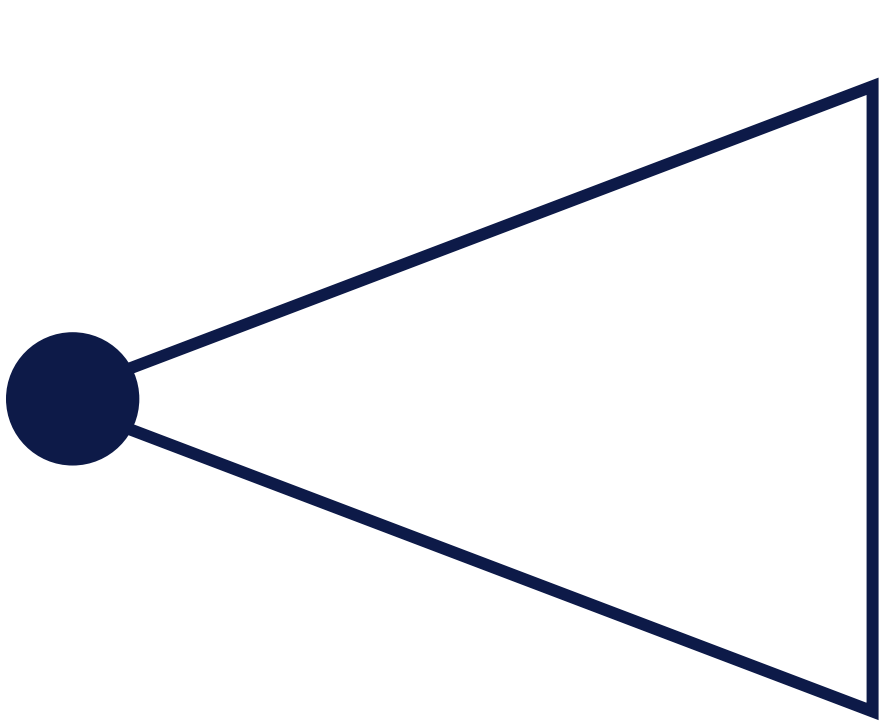


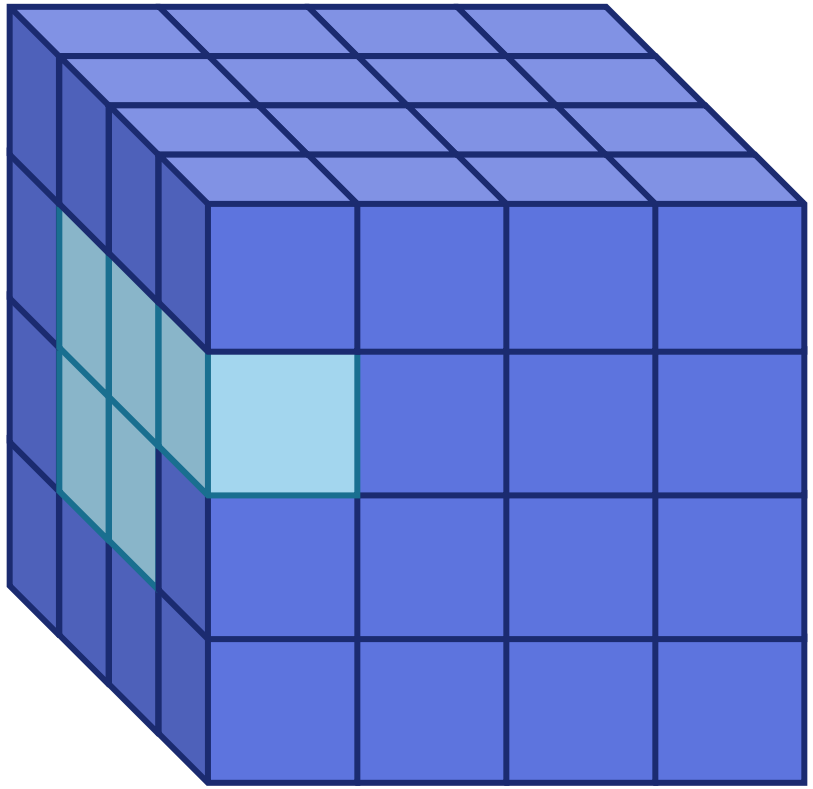
RENDERING A HEIGHTMAP



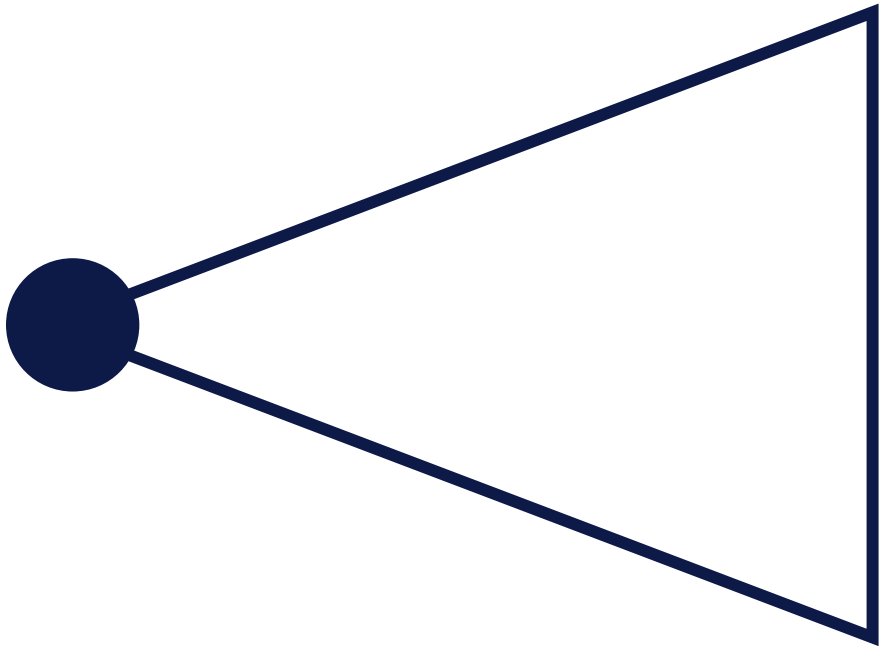


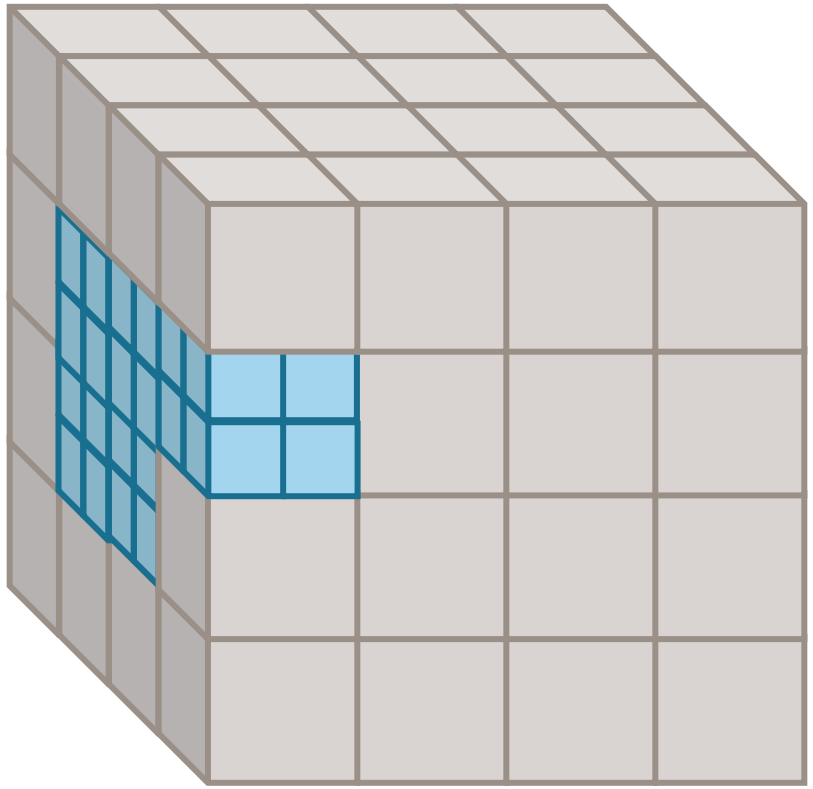
RENDERING A HEIGHTMAP



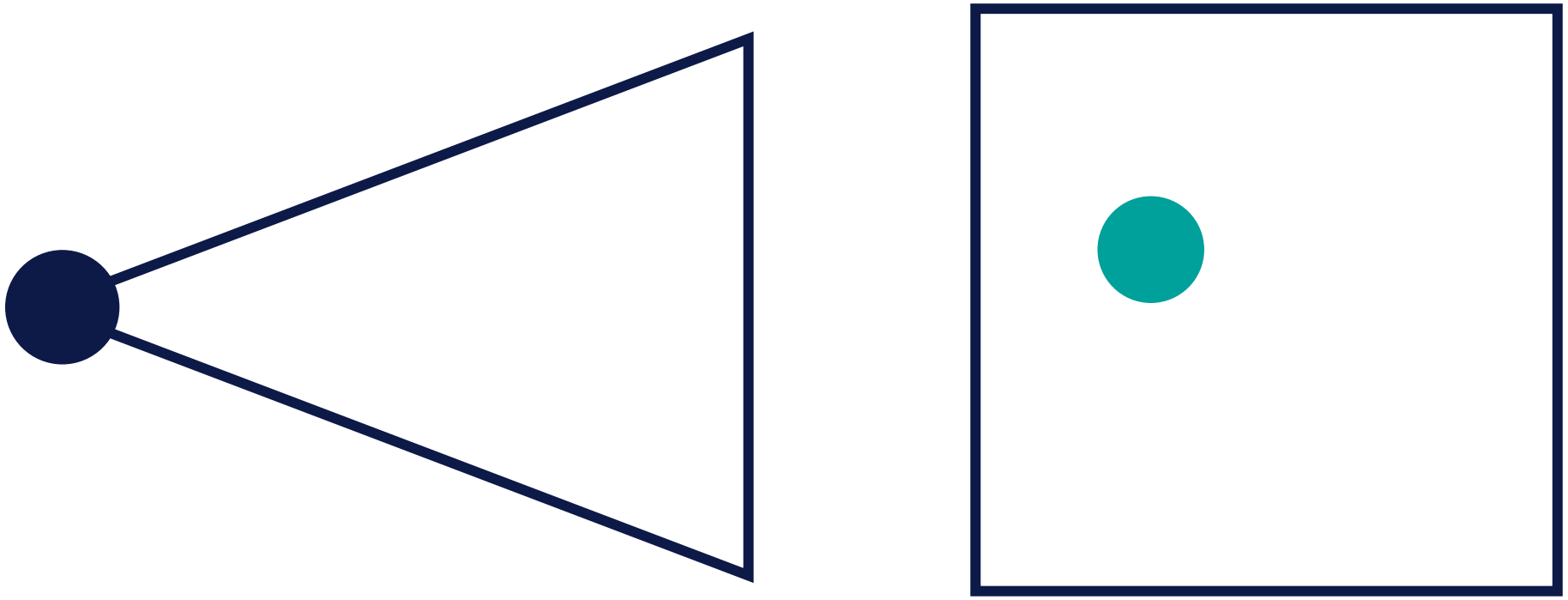


RENDERING A HEIGHTMAP

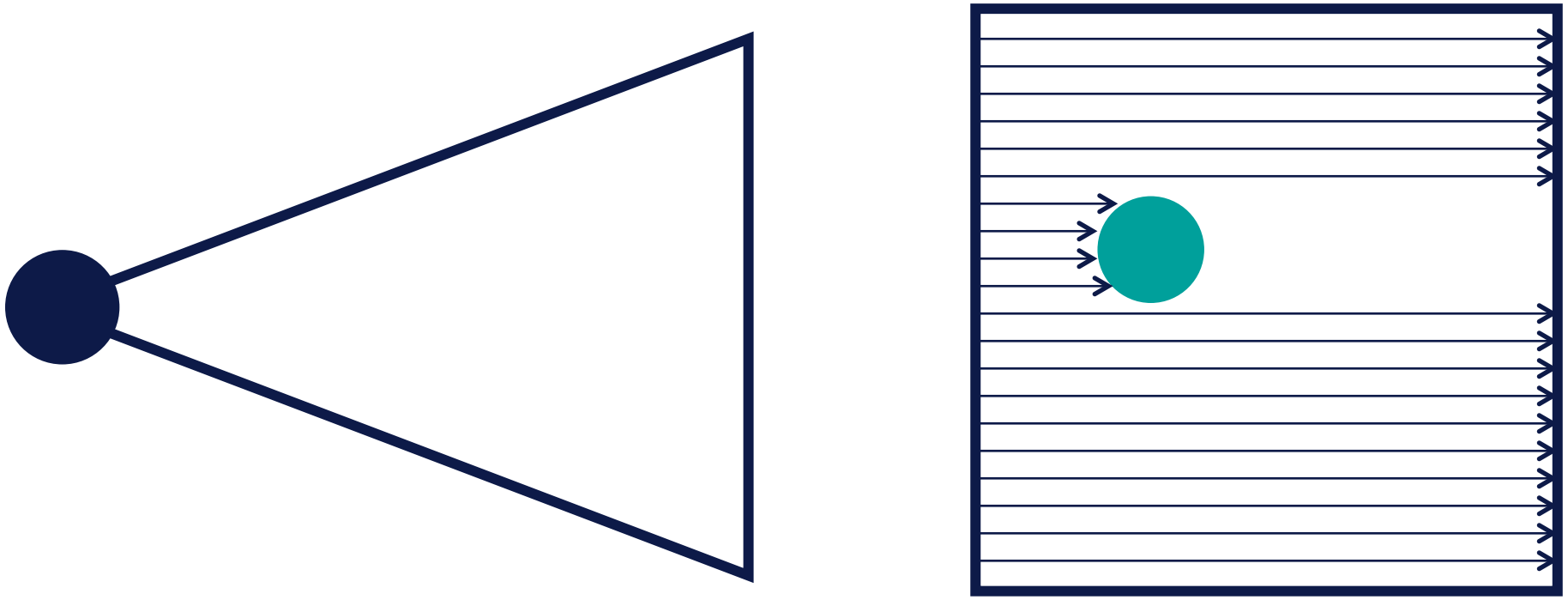




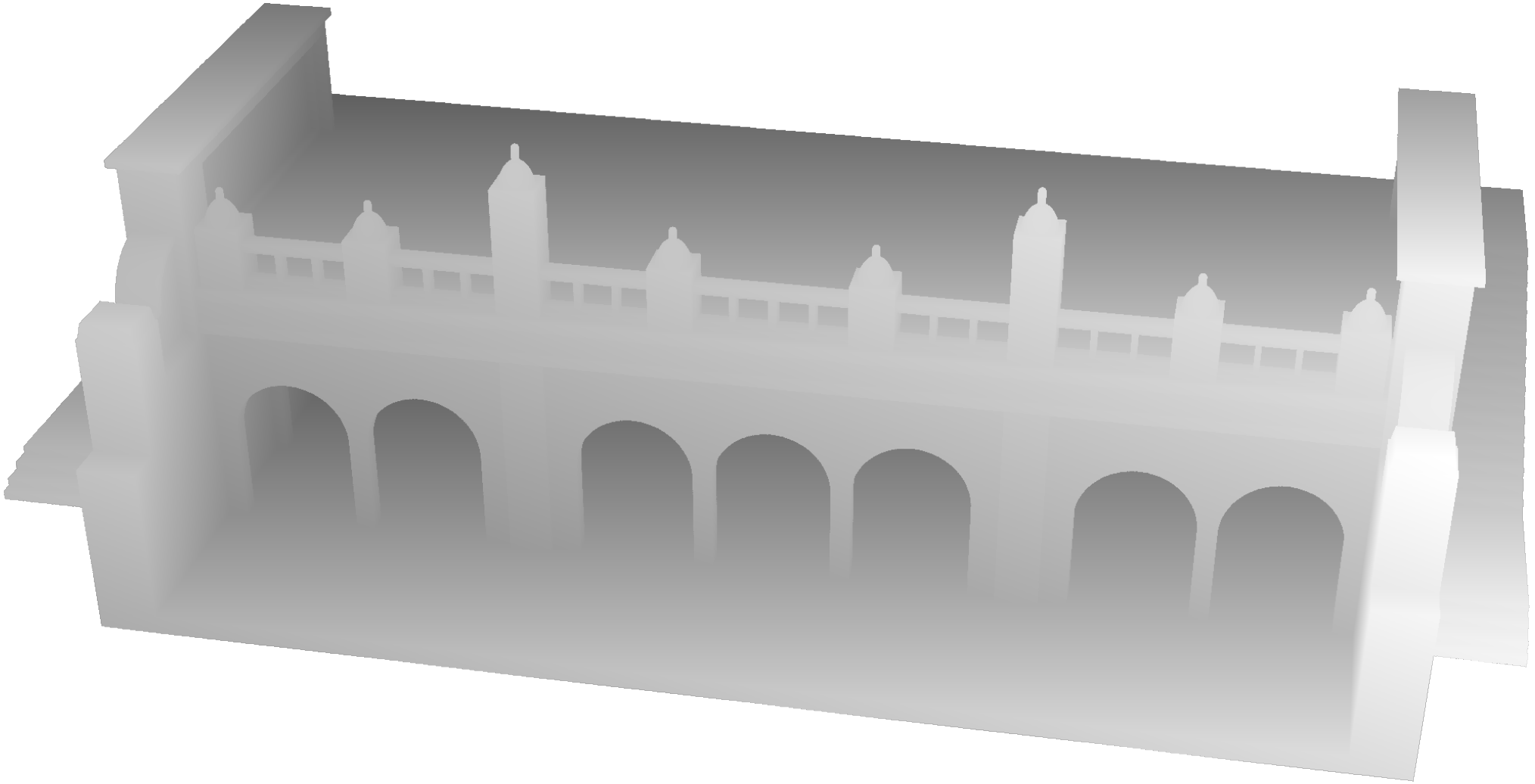
ORTHOGRAPHIC HEIGHTMAP



ORTHOGRAPHIC HEIGHTMAP



HEIGHTMAP OUTPUT

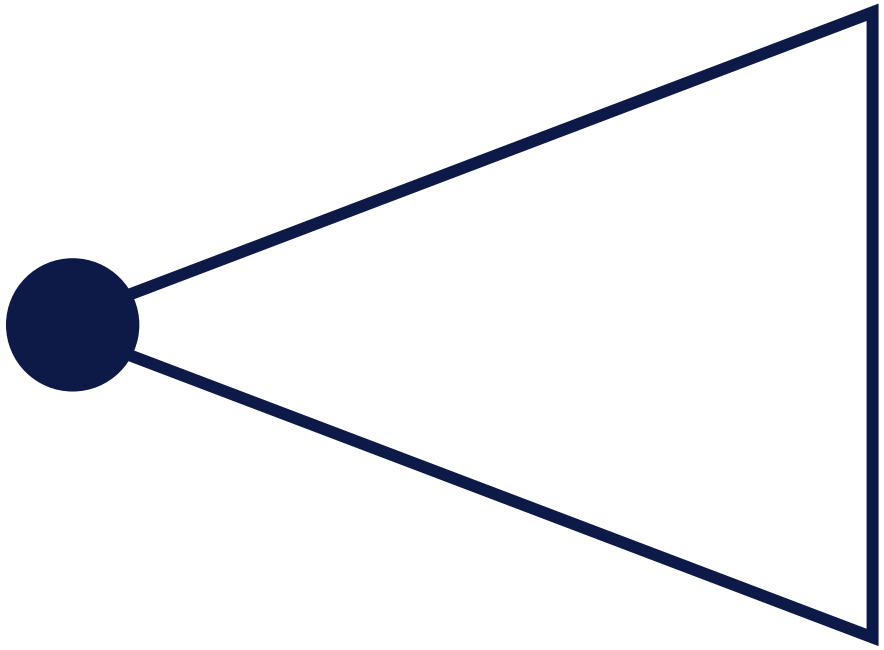


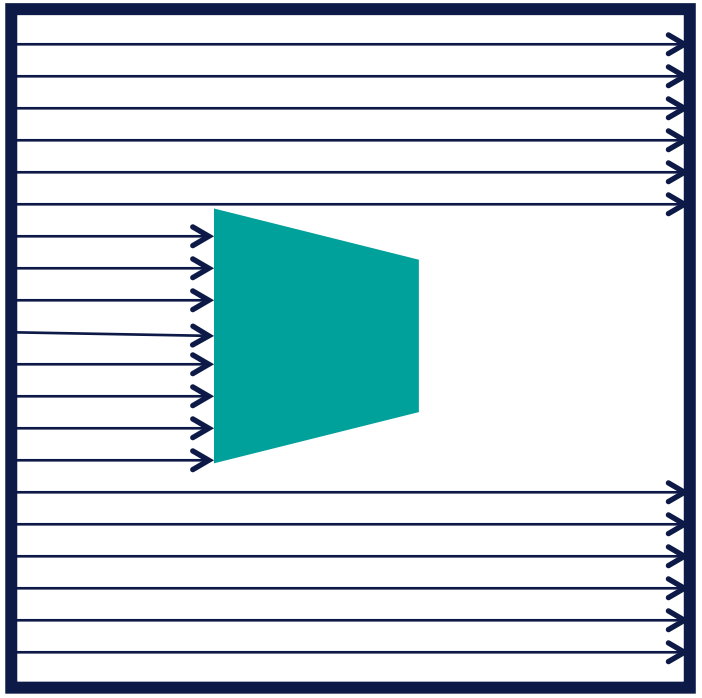
USER-PROVIDED MATRIX

opcode	lhs	rhs	out
X	–	–	slot 0
Y	–	–	slot 1
SQUARE	slot 0	–	slot 0
SQUARE	slot 1	–	slot 1
ADD	slot 0	slot 1	slot 1
SQRT	slot 1	–	slot 1
SUB	slot 1	1.0f	slot 0
SUB	0.5f	slot 1	slot 1
MAX	slot 0	slot 1	slot 1

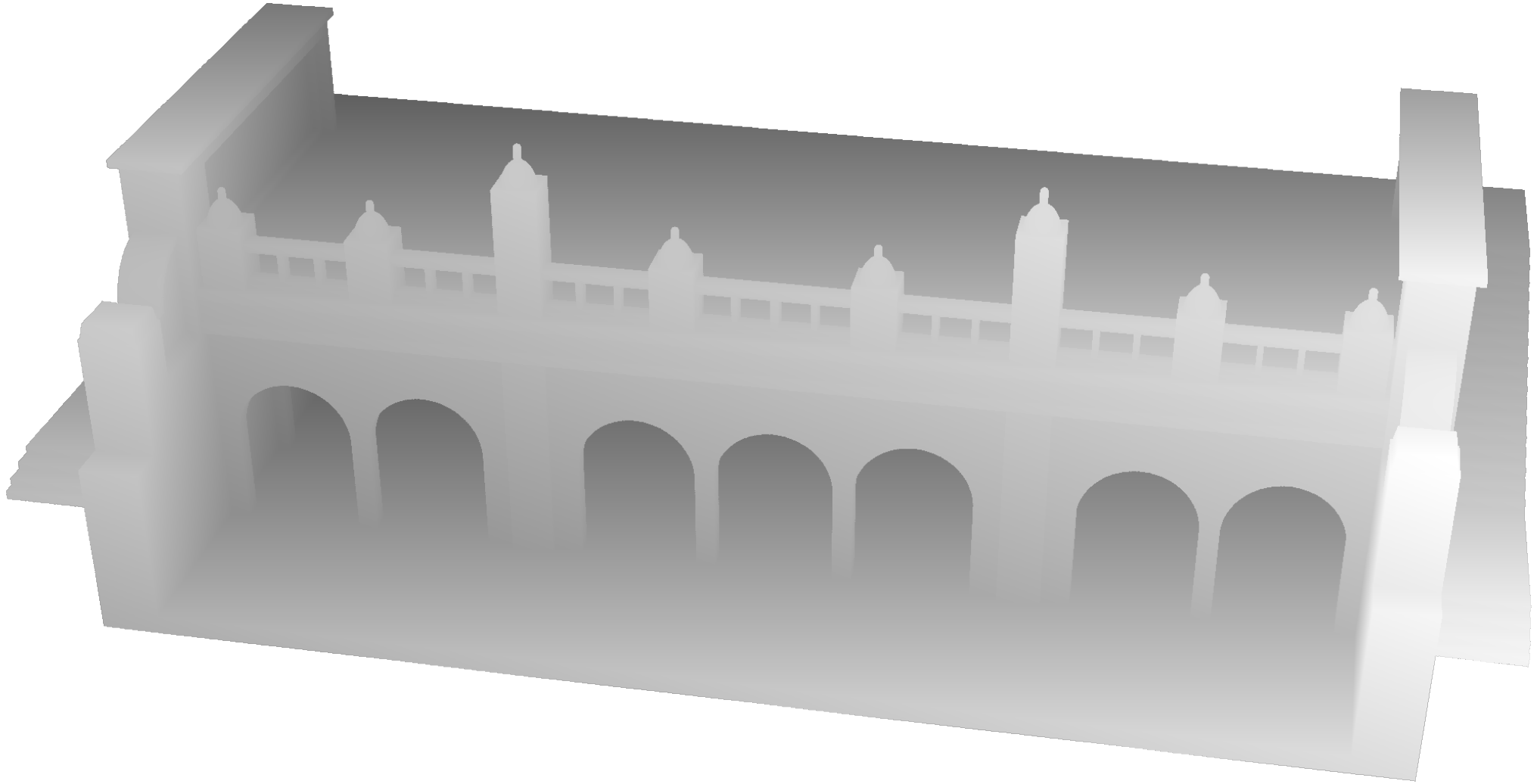
- `XYZ` opcodes return coordinates that are transformed by a 4x4 matrix
- OpenGL-style homogenous coordinates (allows for perspective)
- Also used to implementing the camera
- Faster than transforming the model and uploading a new tape

USER-PROVIDED MATRIX





HEIGHTMAP OUTPUT

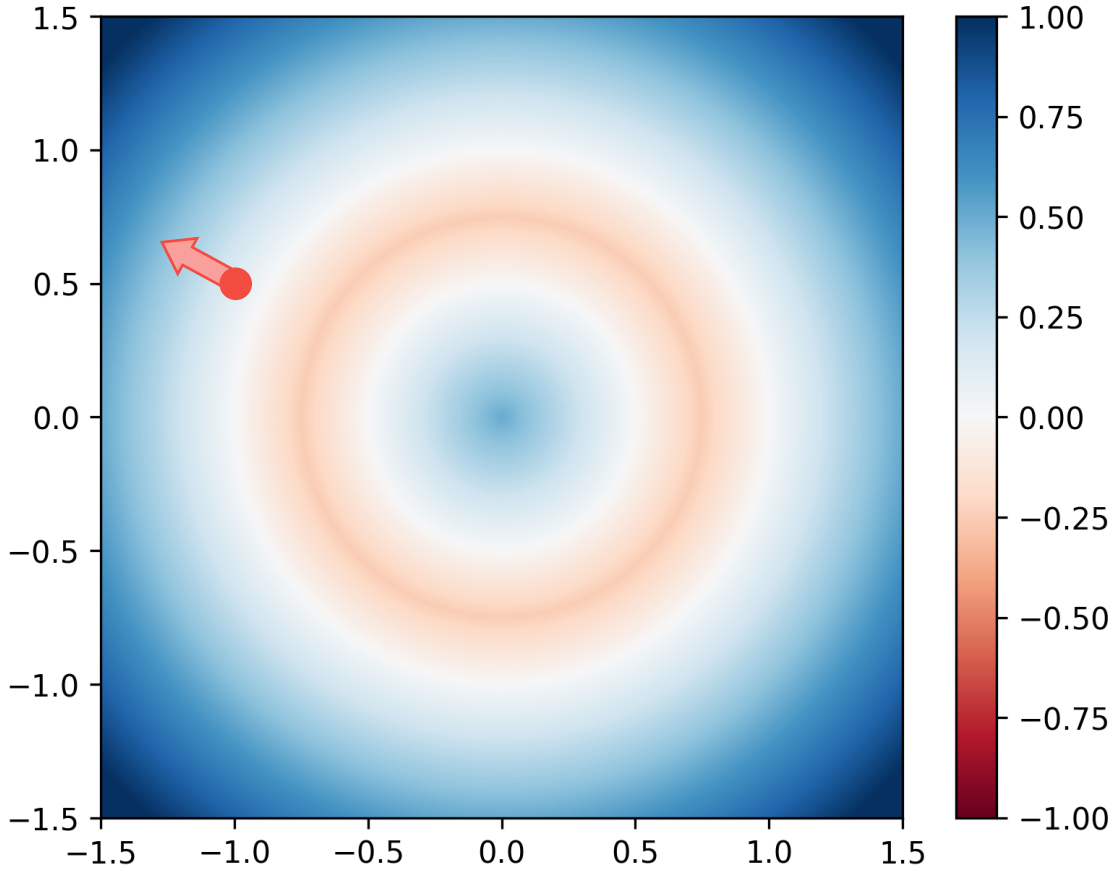


CALCULATING NORMALS

- The **partial derivatives**

  `df/dx, df/dy, df/dz`

  approximate the normal at points close to the model surface
- **Forward-mode automatic differentiation** can be implemented by interpreting the tape with **value + partial derivatives**



2D AUTOMATIC DIFFERENTIATION

opcode	lhs	rhs	out
X	–	–	-1.0, (1, 0)
Y	–	–	0.5, (0, 1)
SQUARE		–	
SQUARE		–	
ADD			
SQRT		–	
SUB		1.0f	
SUB	0.5f		
MAX			

2D AUTOMATIC DIFFERENTIATION

opcode	lhs	rhs	out
X	–	–	-1.0, (1, 0)
Y	–	–	0.5, (0, 1)
SQUARE	-1.0, (1, 0)	–	
SQUARE		–	
ADD			
SQRT		–	
SUB		1.0f	
SUB	0.5f		
MAX			

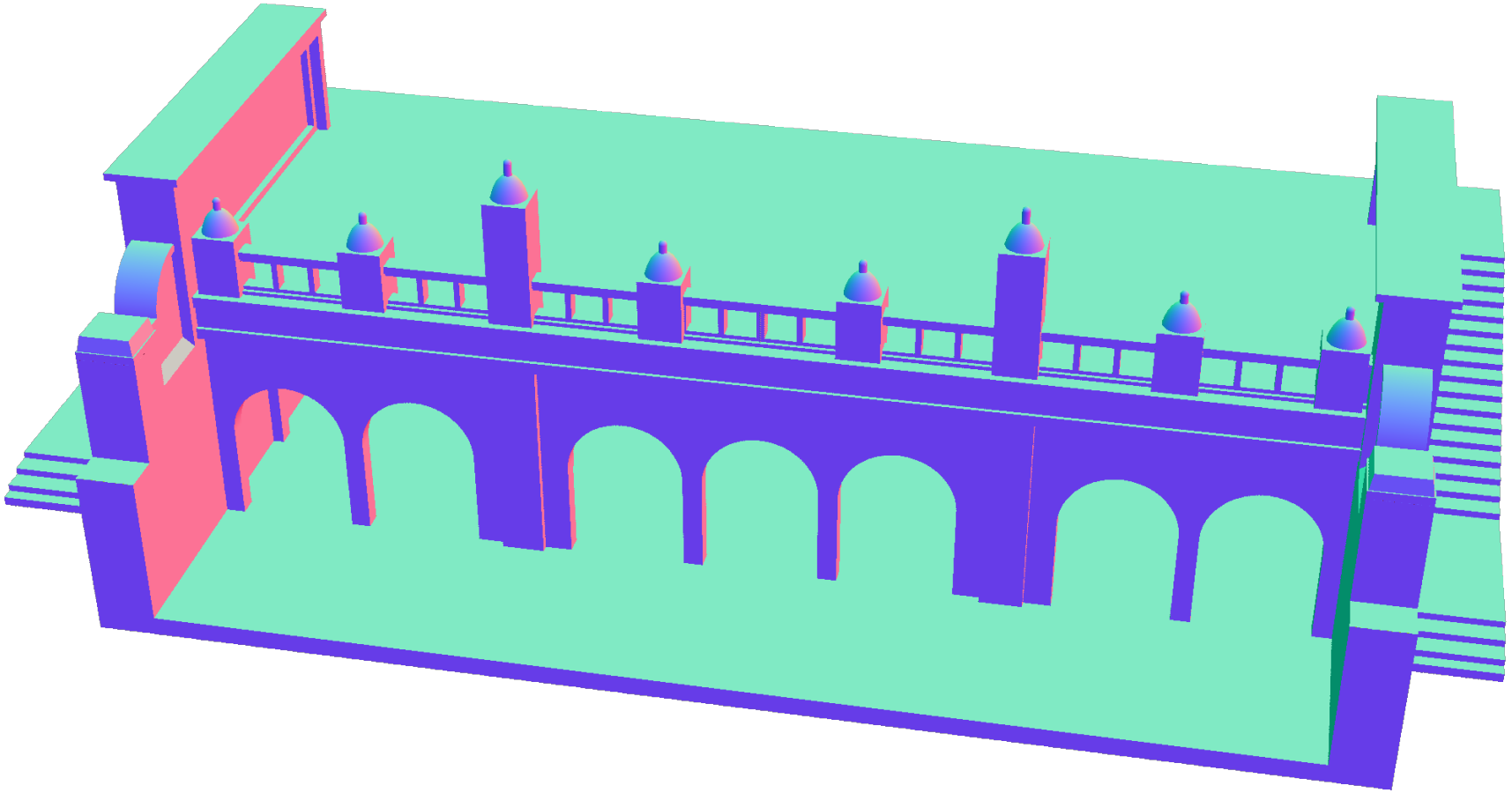
2D AUTOMATIC DIFFERENTIATION

opcode	lhs	rhs	out
X	–	–	-1.0, (1, 0)
Y	–	–	0.5, (0, 1)
SQUARE	-1.0, (1, 0)	–	1.0, (-2, 0)
SQUARE		–	
ADD			
SQRT		–	
SUB		1.0f	
SUB	0.5f		
MAX			

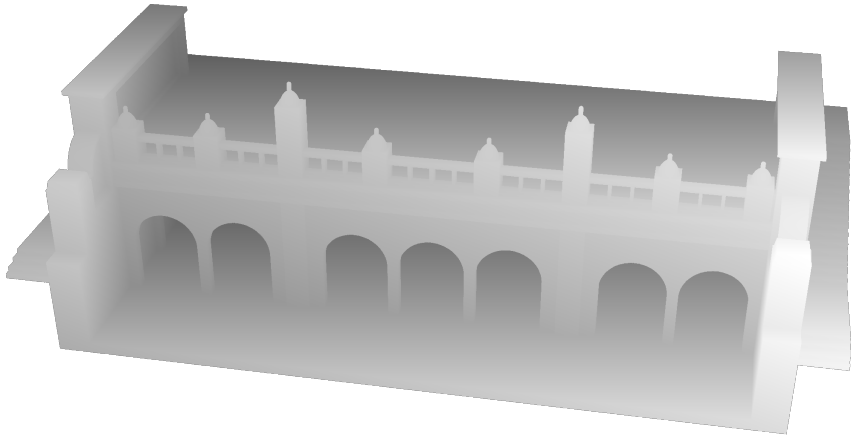
2D AUTOMATIC DIFFERENTIATION

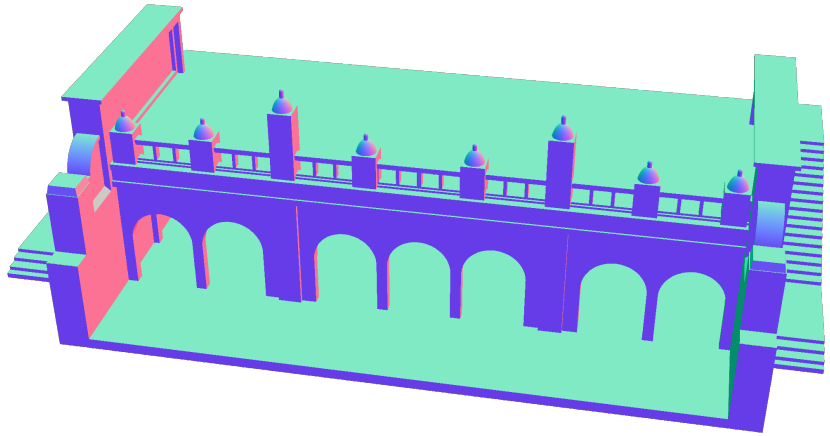
opcode	lhs	rhs	out
X	–	–	-1.0, (1, 0)
Y	–	–	0.5, (0, 1)
SQUARE	-1.0, (1, 0)	–	1.0, (-2, 0)
SQUARE	0.5, (0, 1)	–	0.25, (0, 1)
ADD	1.0, (-2, 0)	0.25, (0, 1)	1.25, (-2, 1)
SQRT	1.25, (-2, 1)	–	1.12, (-0.89, 0.45)
SUB	1.12, (-0.89, 0.45)	1.0f	0.12, (-0.89, 0.45)
SUB	0.5f	1.12, (-0.89, 0.45)	-0.62, (0.89, -0.45)
MAX	0.12, (-0.89, 0.45)	-0.62, (0.89, -0.45)	0.12, (-0.89, 0.45)

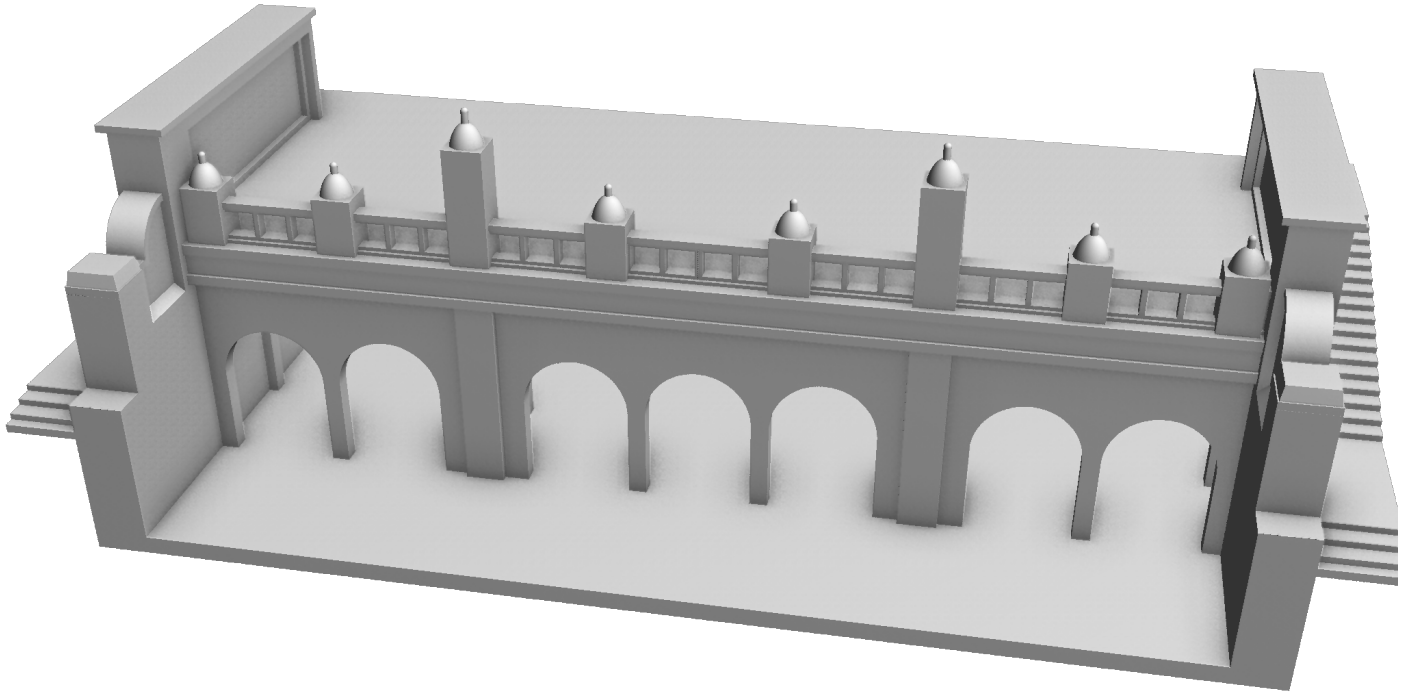
NORMALS OUTPUT



POST-PROCESSING







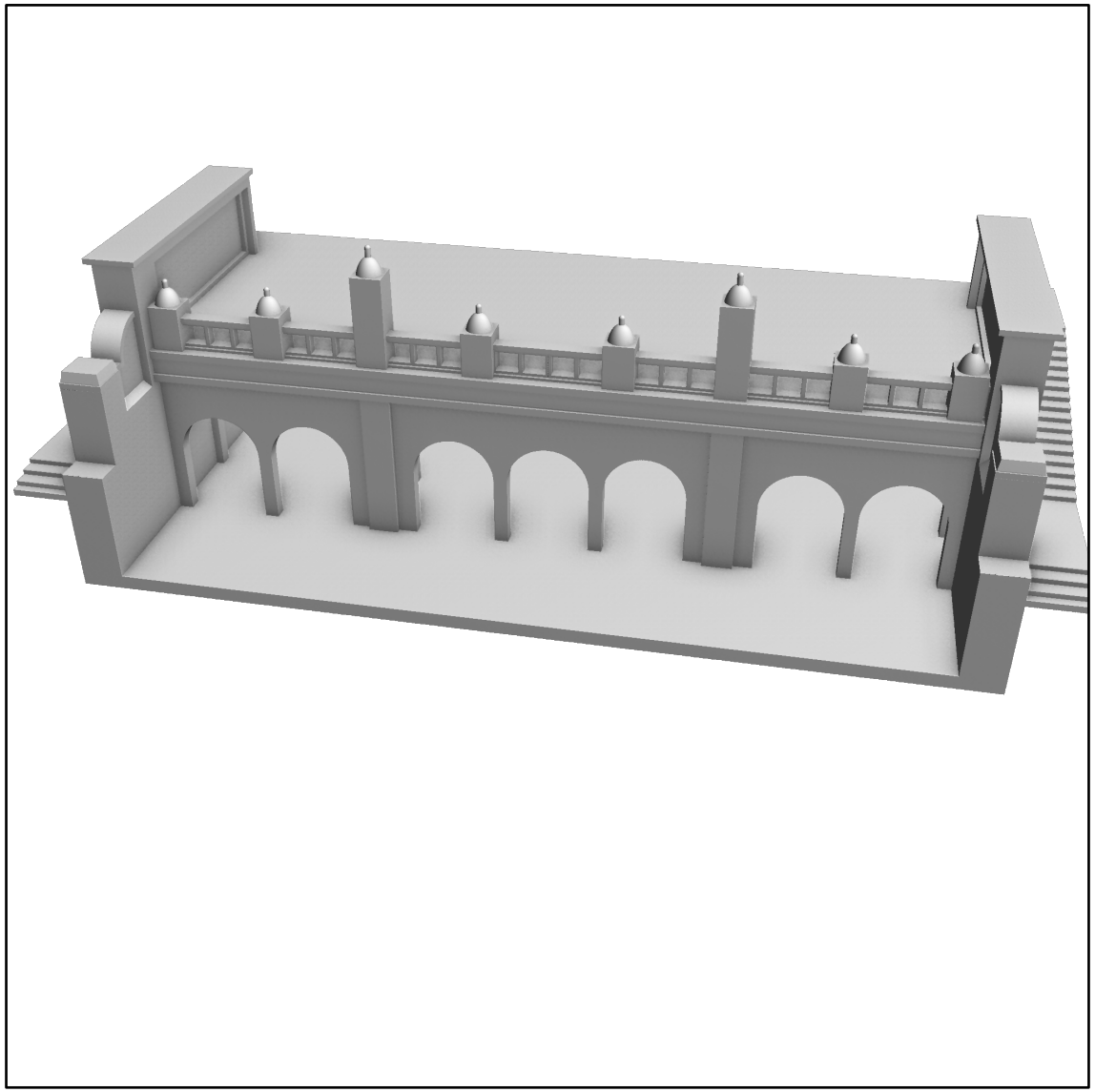
BENCHMARKING MACHINES

GeForce GT 750M
2013 MacBook Pro

GTX 1080 Ti
2017 VR/ML workstation

Tesla V100
AWS `p3.2xlarge`

ARCHITECTURE MODEL

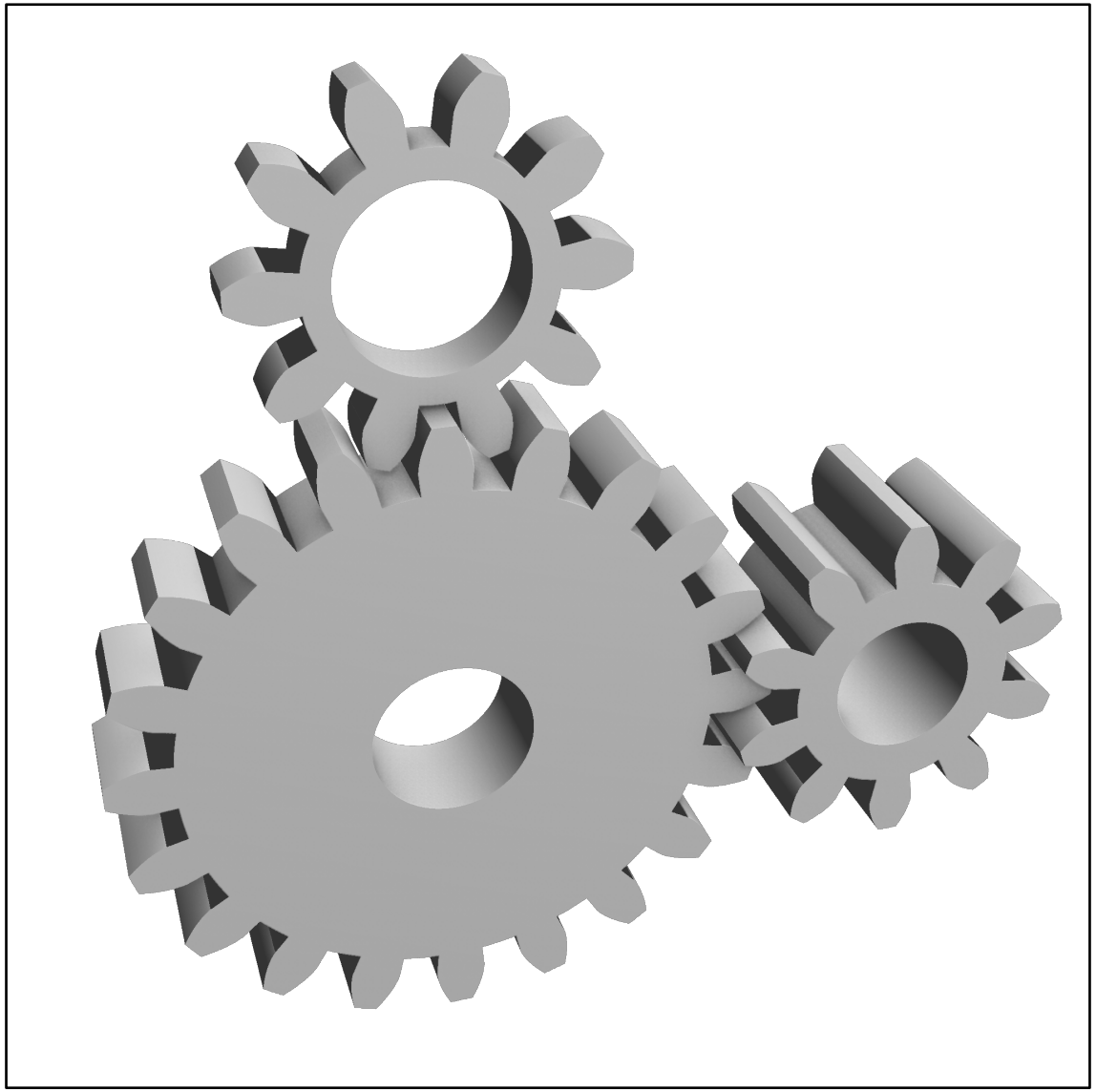


Frame time (ms)

Size	GT 750M	GTX 1080 Ti	Tesla V100
256^3	34.3	5.5	3.2
512^3	73.9	9.9	5.3
1024^3	189.9	22.6	12.2
1536^3	331.9	39.3	20.8
2048^3	510.7	60.6	31.9

- 961 clauses (465 min/max)
- CSG heavy model
- Best case for our algorithm

GEARS MODEL

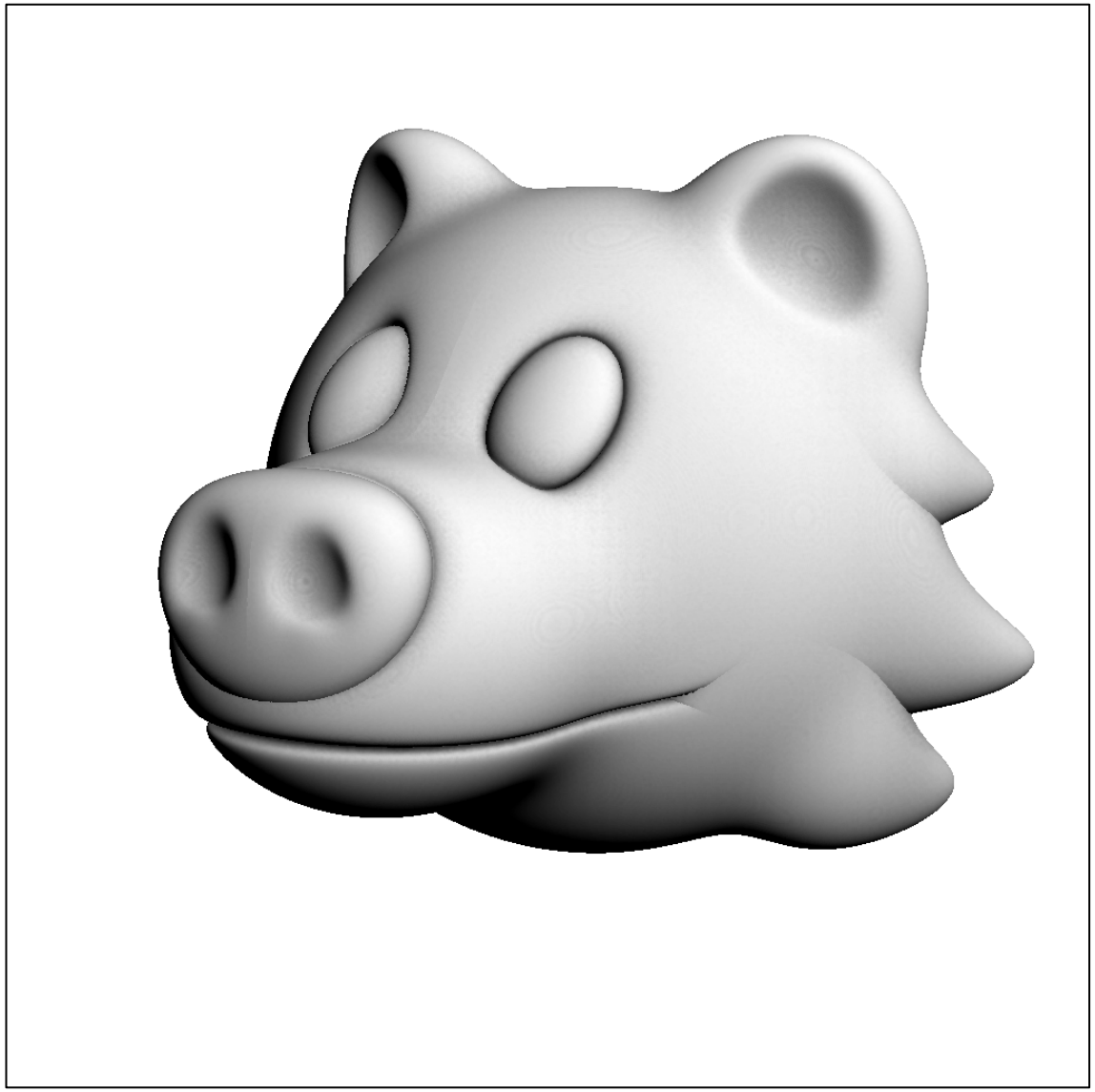


Frame time (ms)

Size	GT 750M	GTX 1080 Ti	Tesla V100
256^3	65.0	9.4	6.2
512^3	154.5	16.6	9.2
1024^3	426.2	40.3	23.1
1536^3	930.3	72.0	39.5
2048^3	–	115.4	62.0

- 1735 clauses (374 min/max)
- Medium amount of CSG
- Transcendental functions

BEAR HEAD MODEL



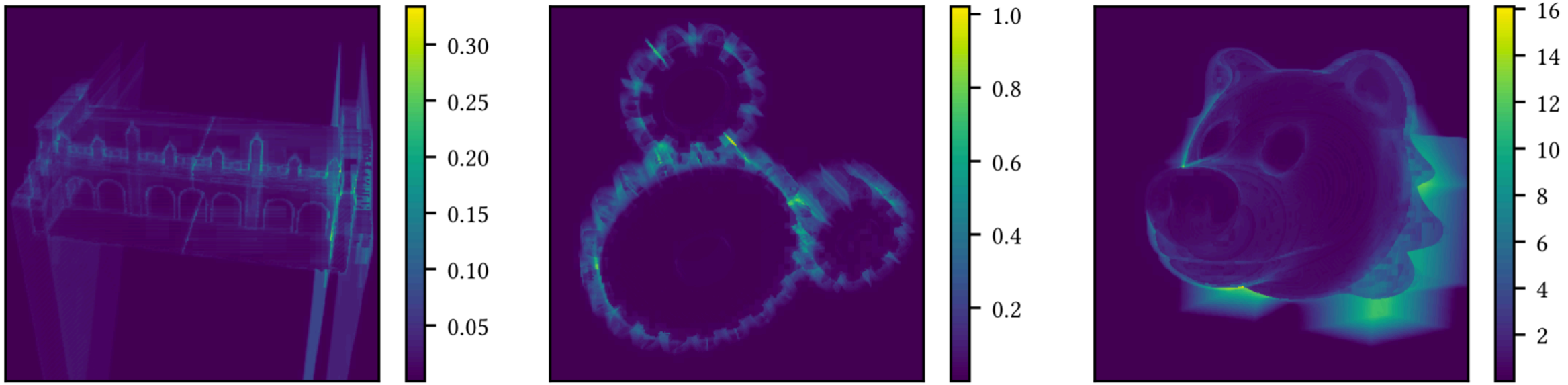
Frame time (ms)

Size	GT 750M	GTX 1080 Ti	Tesla V100
256^3	111.3	11.3	5.2
512^3	503.6	41.1	20.3
1024^3	2352.1	191.0	88.3
1536^3	–	504.2	228.3
2048^3	–	1053.2	437.3

- 541 clauses (27 min/max)
- Very little CSG
- Many smooth blends (`exp/log`)

Based on a design by
Hazel Fraticelli and Anthony Taconi

AMORTIZED WORK (1024^3 VOXELS)



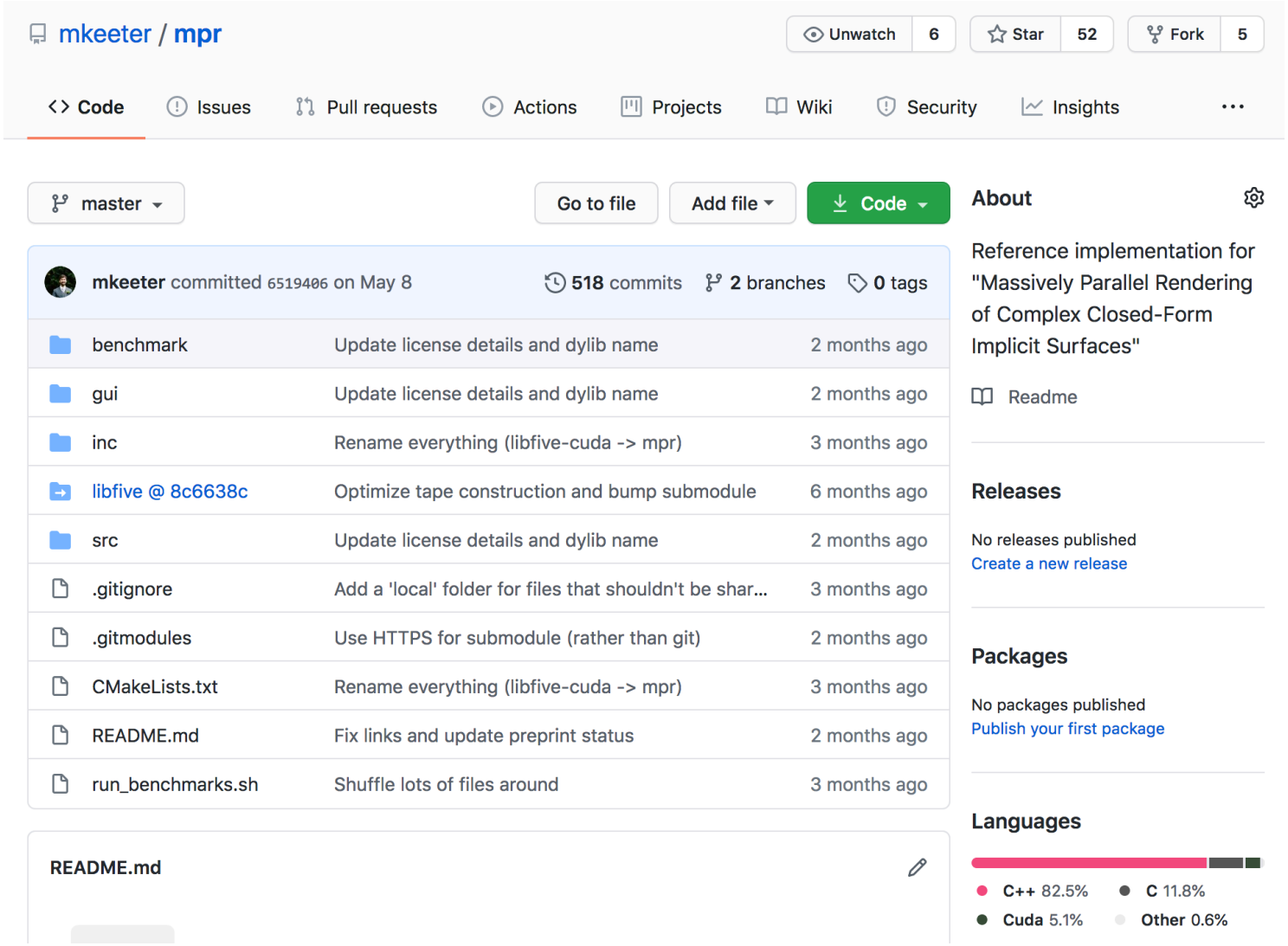
CONCLUSIONS

- New method for rendering complex closed-form implicit surfaces on the GPU
- No triangulation or conventional raytraycing!
- Limited to rendering pure mathematical shapes
- Scales well with GPU power
- Works best on hard-surface CSG
- Interpreter speed is gated by global RAM access
  - Being able to generate executable code on the GPU would be great!
- Lots of RAM required for storing shortened tapes

POTENTIAL FUTURE WORK

- Use this technique for fast voxelization, then render with other GPU-first algorithms for rendering voxel data
  - Sparse Voxel Octrees (Laine and Karras ’10)
  - GVDB (Hoetzlein ’16)
- Use reduced affine arithmetic for better intervals (Fryazinov et al. ’10)
- Better depth culling of tiles and voxels
- Implementing black-box “oracles” for interoperability with other model formats
  - `libfive` already does this for CPU-side evaluation

REFERENCE IMPLEMENTATION



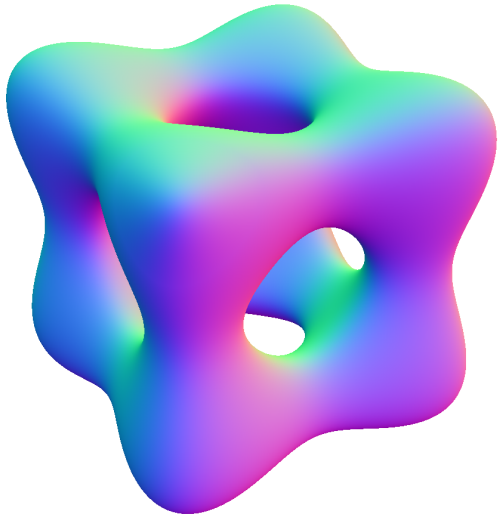
github.com/mkeeter/mpr

Released under MPL v2 (weak copyleft)

Reproduce my results on AWS for about $5!

ACKNOWLEDGEMENTS AND THANKS

- **Beta readers:** Jonathan Bachrach, Blake Courter, Martin Galese, Neil Gershenfeld, Raph Levien, Brian Merchant, Doug Moen, and Amira Abdel Rahman.
- **Benchmarking:** Martin Galese
- **Models:**
  - The bear head is based on a design by Hazel Fraticelli and Anthony Taconi
  - The involute curve math was derived by Peter Fedak
  - The architecture model is based on a design by Jennifer Keeter
- The anonymous reviewers for their feedback and insight
- My colleagues at **Formlabs**, for encouraging this work
- **nTopology**, for their support of the `libfive` kernel



THANK YOU!

Matt Keeter
Independent researcher

mattkeeter.com/research/mpr
matt.j.keeter@gmail.com
@impraxical