Massively Parallel Rendering of Complex Closed-Form Implicit Surfaces

Seminar Computer Graphics

Fabian Friederichs

Motivation

- Implicit surface representations are useful in many areas:
  - Computer graphics
  - Computer aided design
  - Robotics (May et al., 2014)
  - …
- Can be combined trivially using min, max and sign flip
- Some operations are much easier than on B-reps

Motivation

- Visualization in realtime still challenging
  - Esp. For highly complex models
  - Limiting factor: Number of Evaluations

- (Keeter, 2020):
  - Hierarchical evaluation scheme with high branching factor
  - Efficiently utilizes GPU compute capabilities
  - Fewer evaluations and on-the-fly reduction of complexity of the expressions

Implicit Surfaces

Implicit Surfaces

Canonical isolevel of some implicit function $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ with $c = 0$:

$$\mathcal{S}_{|c} = \left\{\mathbf{p} \in \mathbb{R}^3 \mid f\left(\mathbf{p}\right) = c\right\}$$

Example - implicit unit sphere:

$$f(\mathbf{p}) = \sqrt{p_x^2 + p_y^2 + p_z^2} - 1$$

Constructive Solid Geometry (CSG)

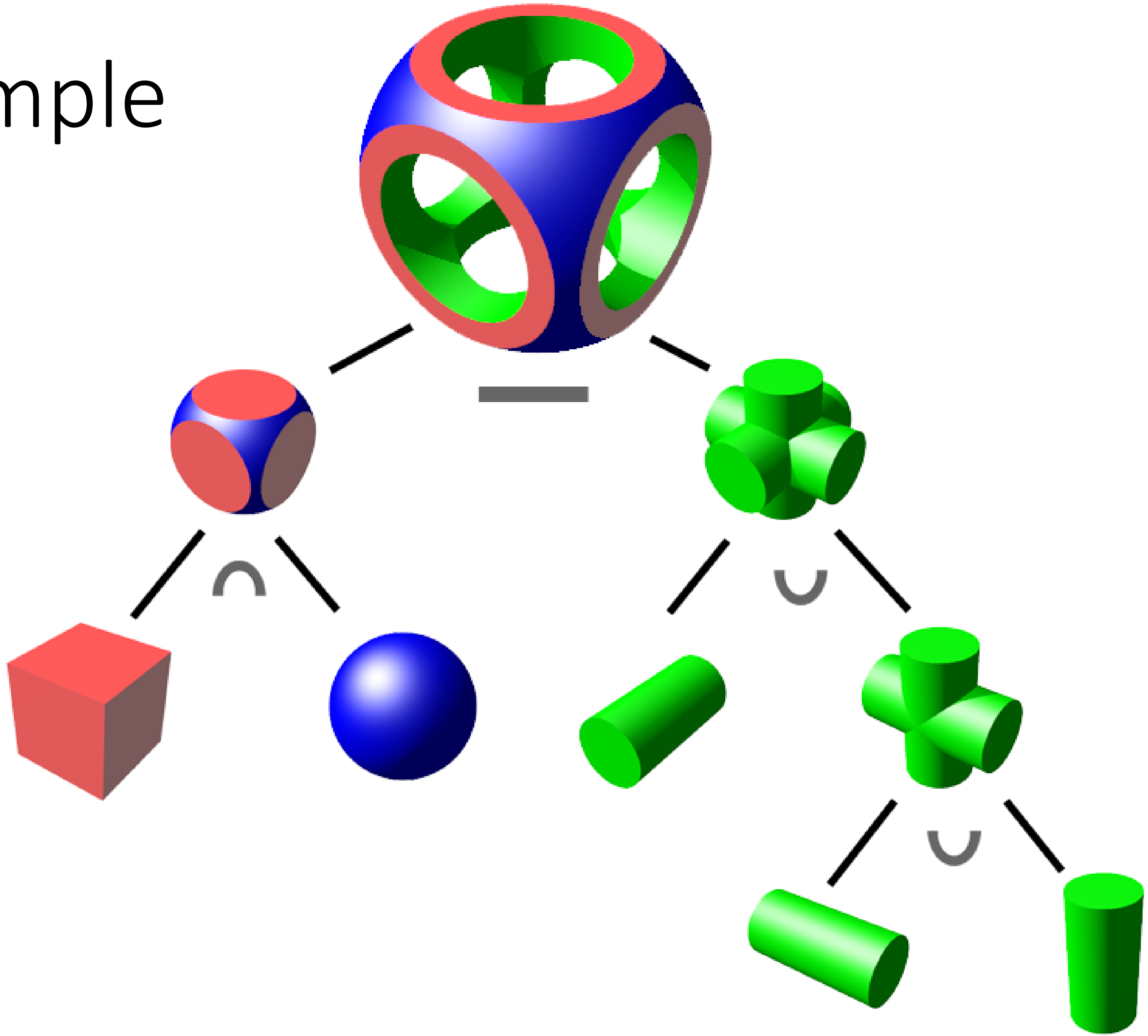
Implicit volumes (enclosed by some impl. surface) can be trivially combined using set-theoretic operations:

$$V_1 \cup V_2 = \left\{\mathbf{p} \in \mathbb{R}^3 \mid \min\left(f_1\left(\mathbf{p}\right), f_2\left(\mathbf{p}\right)\right) \leq 0\right\}$$

$$V_1 \cap V_2 = \left\{\mathbf{p} \in \mathbb{R}^3 \mid \max\left(f_1\left(\mathbf{p}\right), f_2\left(\mathbf{p}\right)\right) \leq 0\right\}$$

$$\mathbb{R}^3 \setminus V_1 = \left\{\mathbf{p} \in \mathbb{R}^3 \mid -f_1\left(\mathbf{p}\right) \leq 0\right\}$$

CSG-Tree Example



https://upload.wikimedia.org/wikipedia/commons/8/8b/Csg_tree.png, last accessed on 15.03.2021

Implicit Surface Normals

We can calculate exact normals easily if we know the gradient:

$$\mathbf{n}\left(\mathbf{p}\right) = \frac{\nabla f\left(\mathbf{p}\right)}{\left\|\nabla f\left(\mathbf{p}\right)\right\|}$$

Transforming Implicit Surfaces

Let $T : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ be a transformation s.t. $\mathbf{u} = T(\mathbf{p})$.

Our original function f transforms as follows:

$$g\left(\mathbf{u}\right) = f\left(T^{-1}\left(\mathbf{u}\right)\right)$$

Signed Distance Fields/Functions (SDF)

Implicit functions which fulfil a special case of the Eikonal Equation:

$$\left\|\nabla f\left(\mathbf{p}\right)\right\| = 1$$

are called *signed distance functions* or *-fields.*

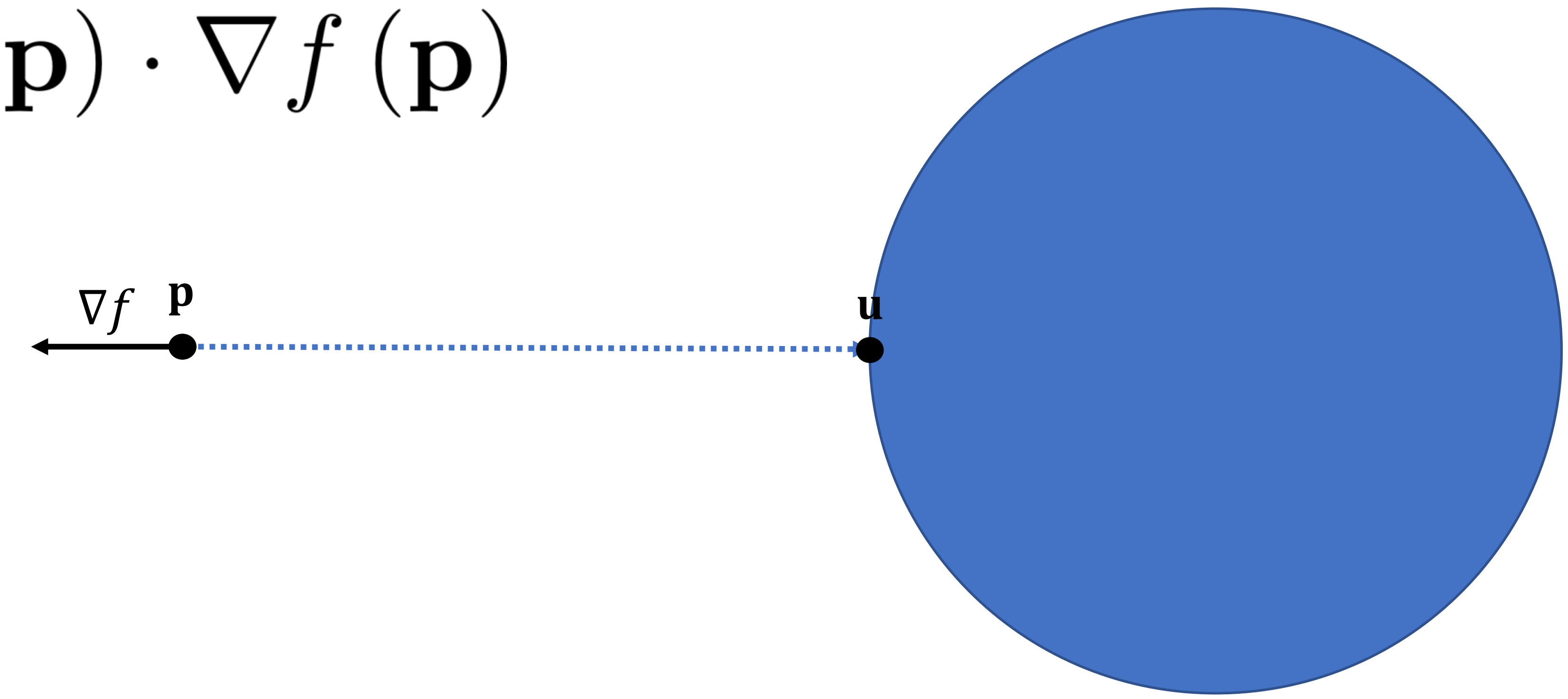
For every point in space, such a function returns the signed euclidean distance to the surface.

The distance is negative if the point is *inside* the surface.

Closest Point on the Surface

If f is a SDF, we can trivially calculate the closest point on the surface from any point in space:

$$\mathbf{u} = \mathbf{p} - f\left(\mathbf{p}\right) \cdot \nabla f\left(\mathbf{p}\right)$$

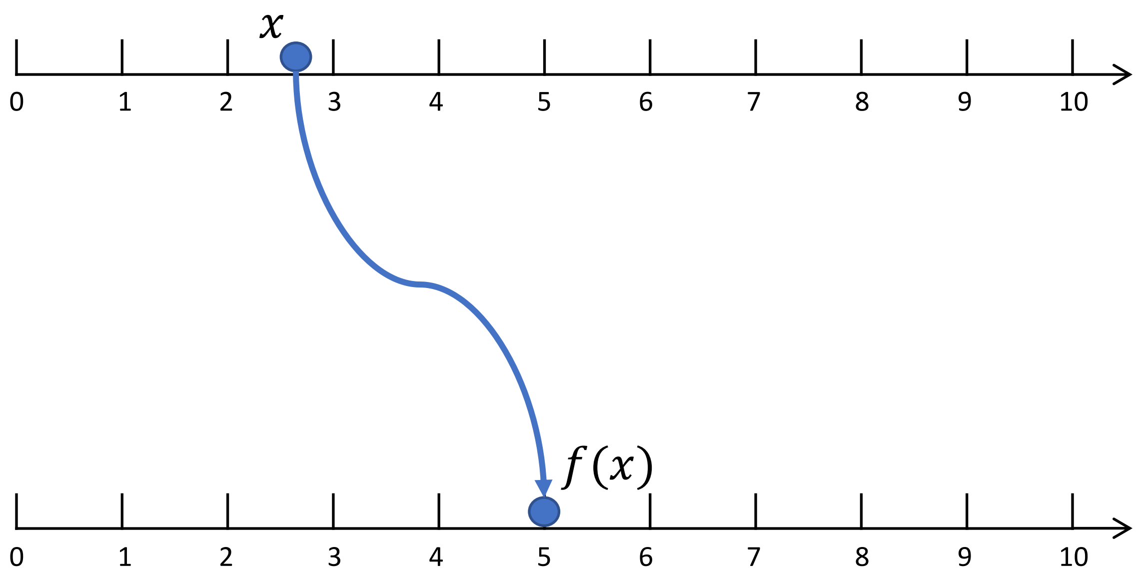


Interval Arithmetic

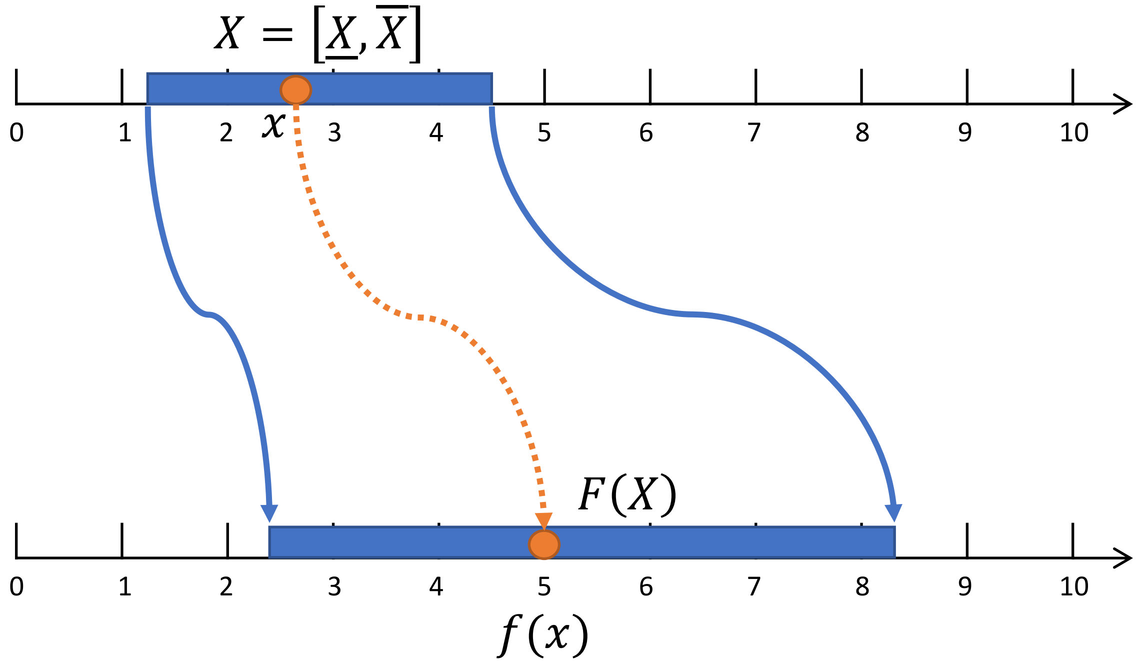
(Moore et al., 2009)

Extending the Real Numbers

Real Arithmetic



Interval Arithmetic



Interval Extensions

We’d like to know the output set of f for all possible values from an input interval $X = [\underline{X}, \overline{X}]$:

$$O = \{\, f(x) \mid x \in X \,\}$$

But: Not necessarily a single interval!

⇒ Let us define an *interval extension* of the function.

Interval Extensions

F is called an *interval extension* of f if:

$$\{ f(x) \mid x \in X \} \subseteq F(X)$$

$\Rightarrow$ The output interval is guaranteed to *contain* the image of f.

$\Rightarrow$ $[-\infty, \infty]$ is always a valid interval extension.

The goal is to find extensions with bounds as tight as possible!

Natural Interval Extensions

For our basic arithmetic operators we can define:

$$
\begin{aligned}
-X &= [-\overline{X}, -\underline{X}] \\
X + Y &= [\underline{X} + \underline{Y}, \overline{X} + \overline{Y}] \\
X - Y &= [\underline{X} - \overline{Y}, \overline{X} - \underline{Y}] \\
X \cdot Y &= [\min\{\underline{X}\,\underline{X}, \underline{X}\overline{Y}, \overline{X}\underline{Y}, \overline{XY}\}, \max\{\underline{X}\,\underline{X}, \underline{X}\overline{Y}, \overline{X}\underline{Y}, \overline{XY}\}]
\end{aligned}
$$

$$\boxed{\frac{X}{Y} = X \cdot \left(\frac{1}{Y}\right) \text{ with } \frac{1}{Y} = \left[\frac{1}{\overline{Y}}, \frac{1}{\underline{Y}}\right]}$$

Only defined if $0 \notin Y$!

Monotonic Functions

If a function is monotonically increasing or decreasing we have:

$$F(X) = \begin{cases} \left[f\left(\underline{X}\right), f\left(\overline{X}\right)\right], & \text{if } f \text{ is monot. inc.} \\ \left[f\left(\overline{X}\right), f\left(\underline{X}\right)\right], & \text{if } f \text{ is monot. dec.} \end{cases}$$

Arbitrary Functions

- No interval extensions exist in general.
- Interval extensions are never unique.
- General approach:
  - Find monotonic subsets of the domain
  - Handle cases separately

- Use existing libraries, e.g. (Melquiond et al., 2006)!

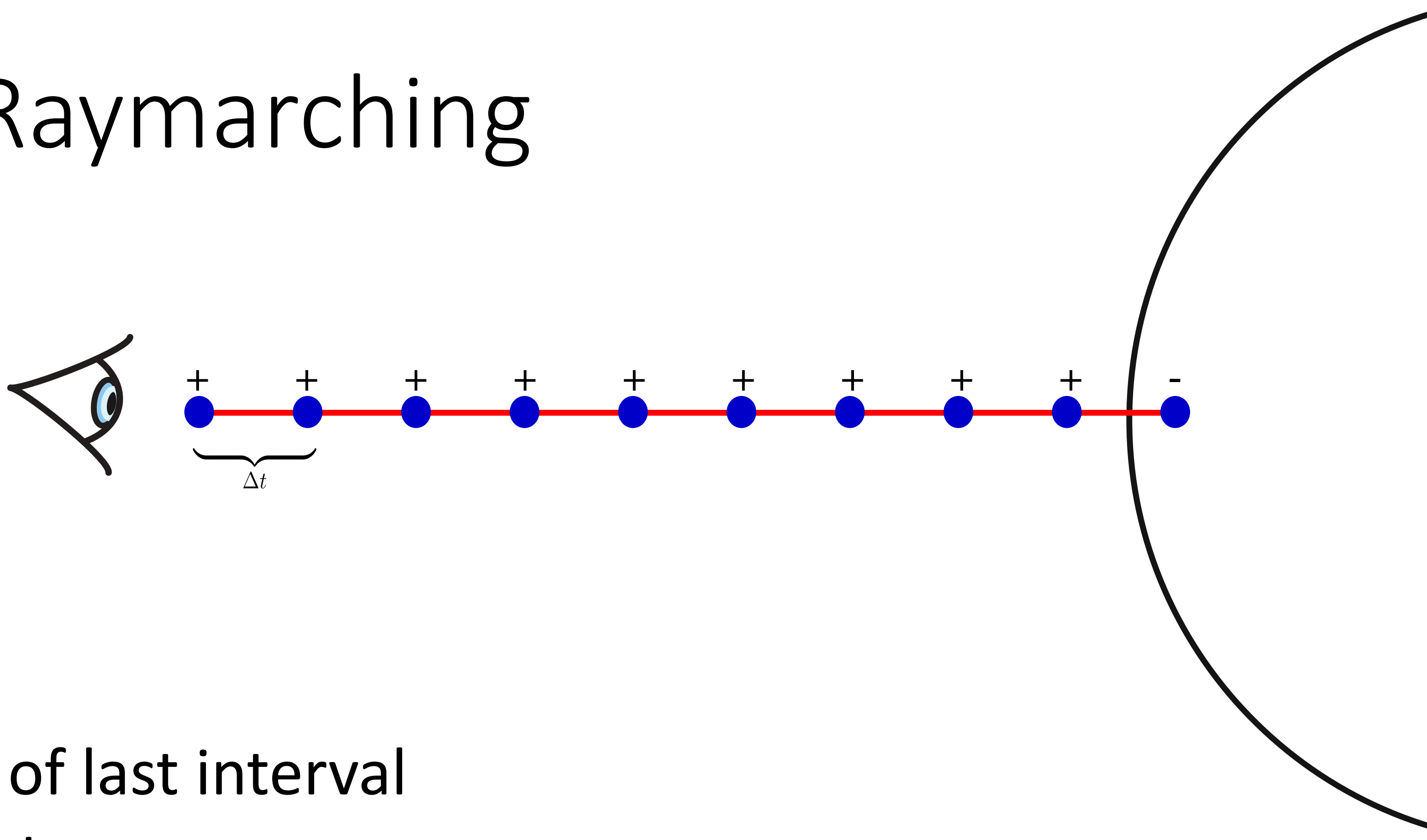
Working on finite-precision Machines

- When doing the calculations on real-world machines we have to make sure that we round correctly, i.e. conservatively.
  - Lower bounds must be round downwards
  - Upper bounds must be round upwards
- Corresponding rounding modes are defined by the IEEE 754 standard ("IEEE Standard for Floating-Point Arithmetic" 2019)
- Keeter suggests to use these special rounding modes when available, but also states that the error is unnoticeable in practical rendering applications

Related Work

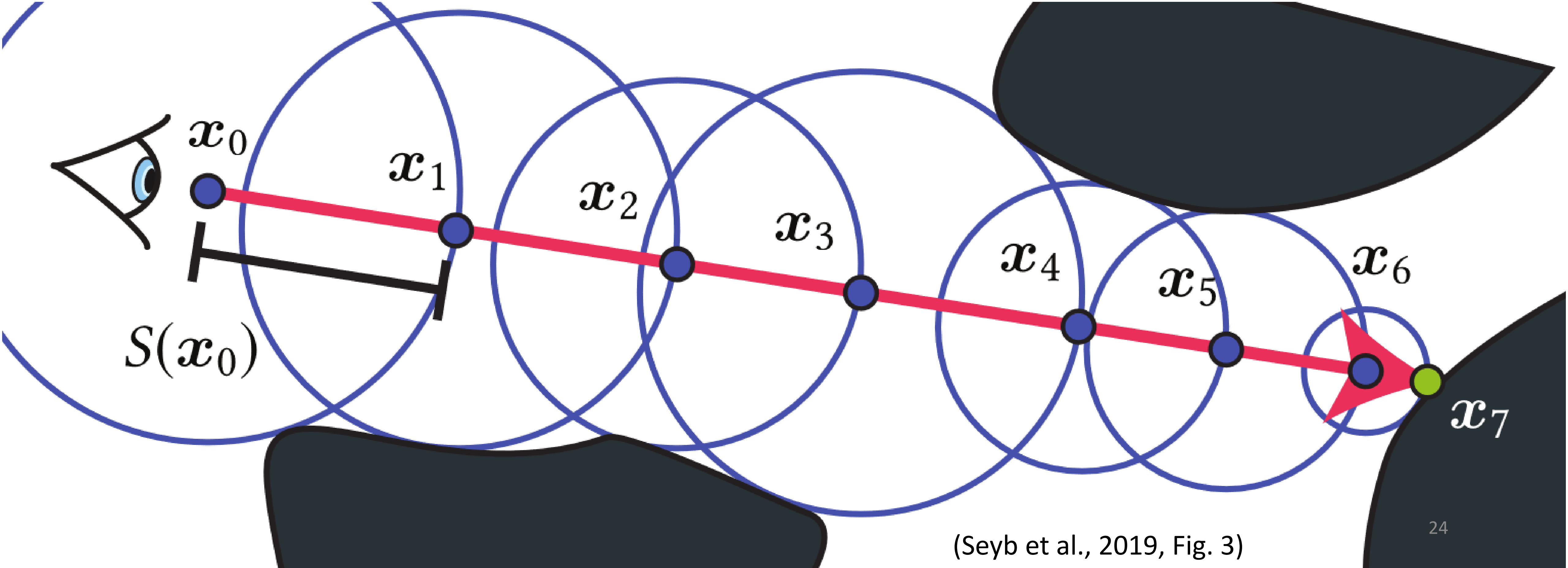
Direct Methods: Raymarching

- Start ray at the camera
- Step into the scene by Δt increments
- When the sign flips:
  - Three options:
    - Return midpoint of last interval
    - Linearly interpolate
    - Iterative root finding, e.g. bisection method or newton's method



Direct Methods: Sphere Marching (Hart, 1996)

In case of SDFs we can calculate adaptive step sizes:



(Seyb et al., 2019, Fig. 3)

Sphere Marching, more general Functions

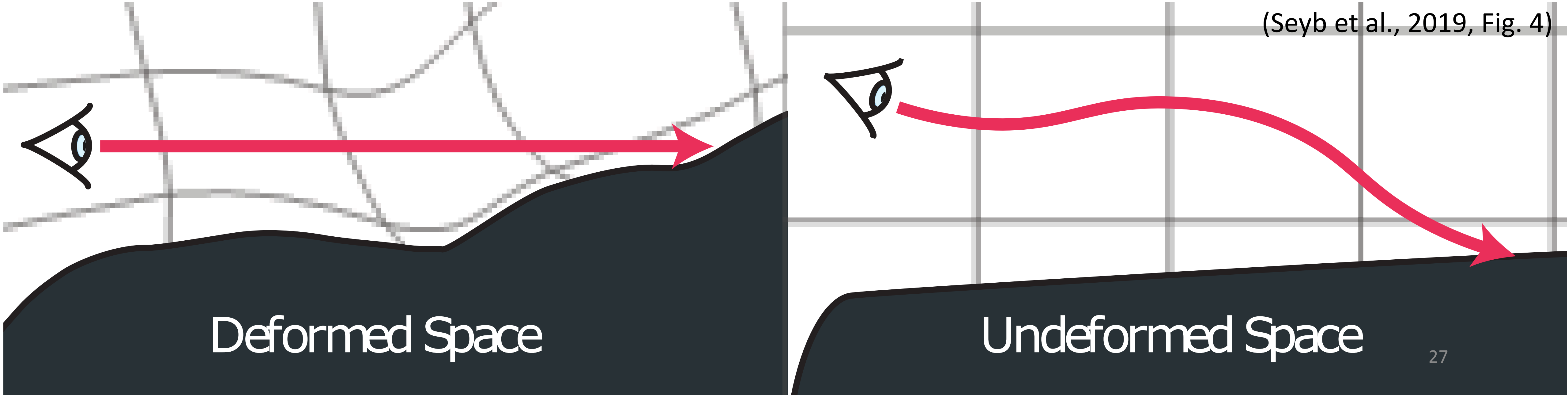
- No convergence if $\|\nabla f\| > 1$, due to “overshooting” the surface (Seyb et al., 2019).
- Suboptimal convergence if $\|\nabla f\| < 1$.

- If the function is Lipschitz-continuous, divide step size by Lipschitz constant, might lead to suboptimal step sizes (Seyb et al., 2019)
    - Also: Lipschitz constant hard or impossible to compute in general

Non-linear Sphere Tracing (Seyb et. al., 2019)

Instead of computing the inverse deformation at every step:

- Calculate inverse jacobian
- In each sphere tracing step, locally solve initial value problem using this inverse
- Adds another step size parameter



(Seyb et al., 2019, Fig. 4)

Direct Methods - Summary

- Require many evaluations of the target function
  - ⇒ Main limiting factor performance-wise
- Accurate
- Inherently adaptive to the display resolution (one ray per pixel!)

Indirect Methods: Marching Cubes & Co.

Idea: Use isosurface extraction algorithm[1] to create an intermediate boundary representation.

Then render using the usual rasterization pipeline.

- Fast and well studied solution
- Relies on finite sampling
  ⇒ Causes aliasing artifacts
- For similar accuracy as direct methods: Shrink primitives down to sub-pixel size

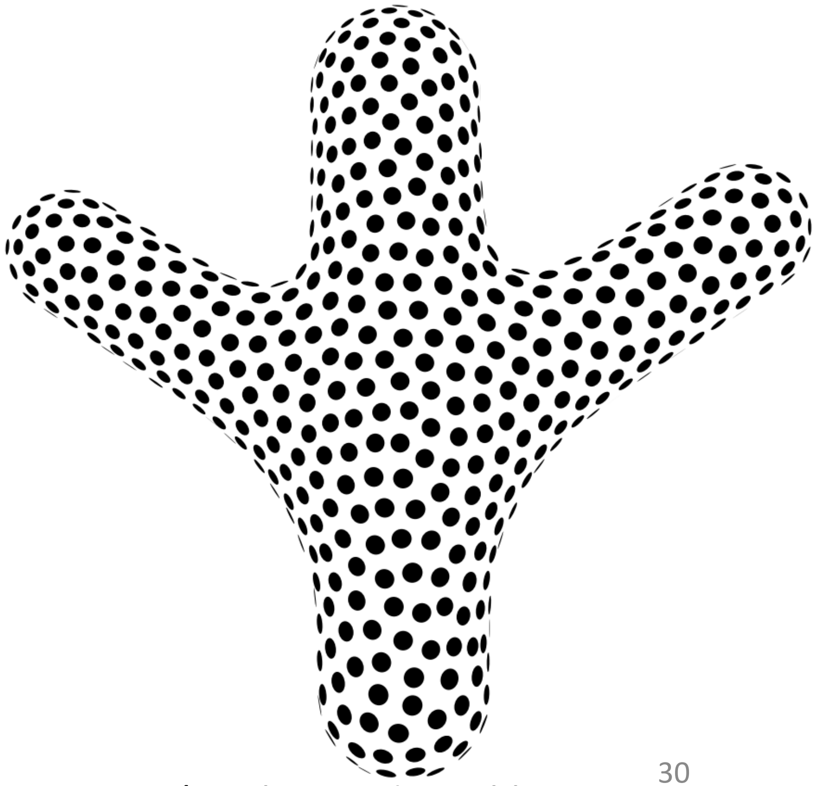
1: E.g. one of (Lorensen and Cline, 1987), (Kobbelt et al., 2001), (Ju et al., 2002), (Greß and Klein, 2004)

Particle based Methods

Idea (Hart et al., 2002; Witkin and Heckbert, 1994):

- Simulate particle system which constrains the particles to move in the zero-level set, i.e. the surface.
- Add some repulsive force for even distribution
- Visualize particles as small discs

- No artifacts caused by grid-aligned sampling
- Resolution still limited and not display-adaptive



(Witkin and Heckbert, 1994, Fig. 2)

Hierarchical CSG Rendering Method (Duff 1992)

- Partition volume of interest into some space partitioning structure (e.g. octree)
- From root to leaves:
  - Using interval arithmetic, evaluate implicit function for current cell X: $Y = F(X)$.
    - If $\overline{Y} \leq 0$ or $\underline{Y} > 0$ the whole interval is completely inside or outside, mark and stop recursion.
    - Else: Cell is *ambiguous.* Proceed with the children of the cell.
    - Recurse downwards until desired resolution is reached.

Hierarchical CSG Rendering Method (Duff 1992)

- Small spatial nodes are usually affected by only a small number of CSG leaves
  $\Rightarrow$ The tree for the child cells can be greatly simplified.[1]

- In Summary:
  - Huge volumes can be classified with very few evaluations
  - Small children down in the hierarchy only receive vastly simplified expressions

- Dyllong and Grimm (2007) implemented the idea with additional optimizations

1: For details please refer to (Duff 1992) or the handed-in report, section 4.3

Algorithm

(Keeter, 2020)

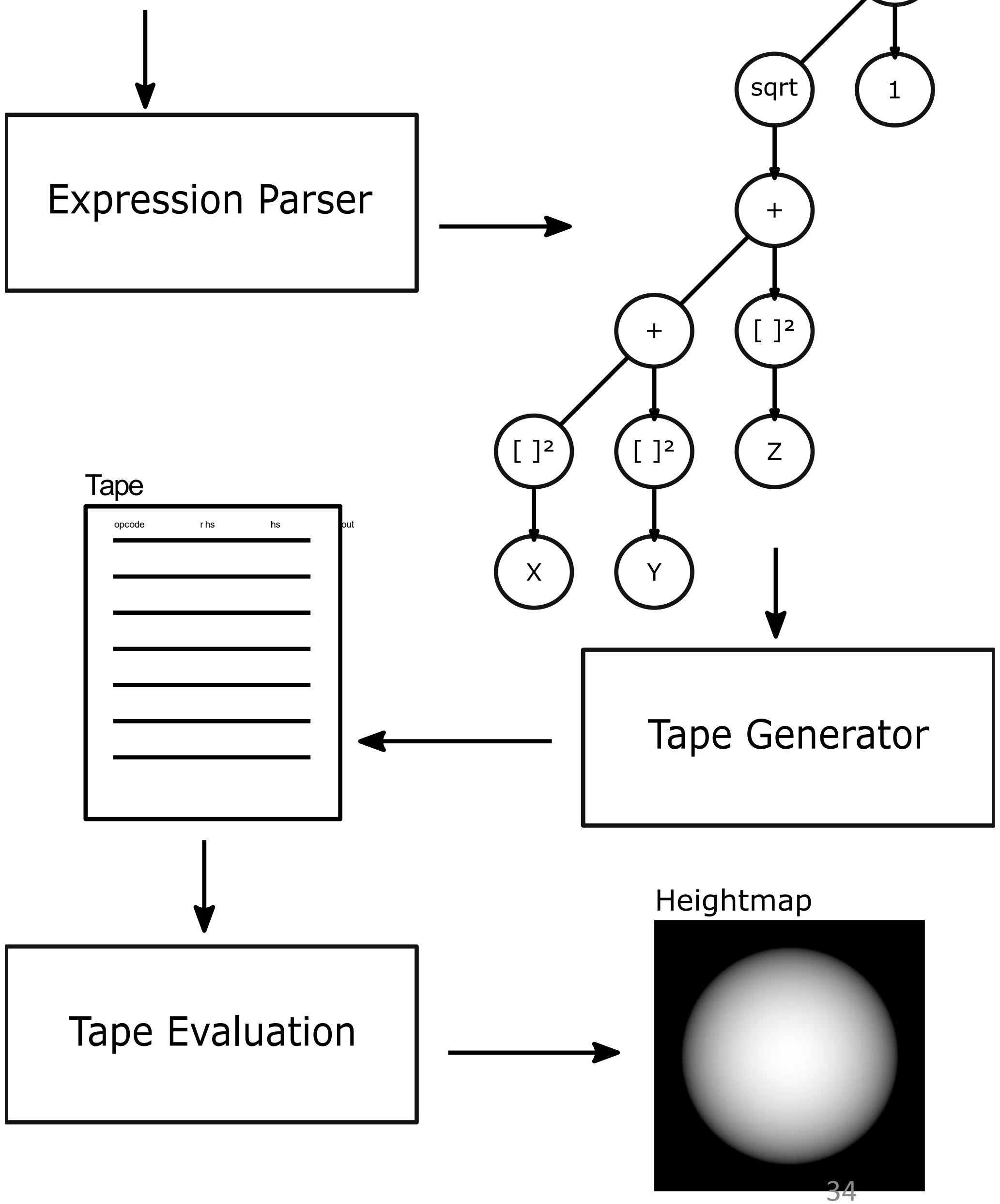
Overview

The approach of Keeter concentrates on algebraic expressions.

The basic steps are as follows:

1. Parse expression into syntax tree
2. Convert syntax tree to tape
3. In parallel, evaluate tape on the GPU while generating the output height map for the final rendering step

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



Parsing the Expression

- Algebraic expressions follow a simple, context free grammar.
- To generate the syntax tree there are a few options:
  - Use standard lexer and parser generators[1]
  - Implement a simple recursive descent parser by hand

1: E.g. lex (lex(1p) — Linux manual page 2021) and yacc (yacc(1p) — Linux manual page 2021)

Generating the initial Tape

- A tape is a series of *instructions*
- Each instruction consists of:
  - An opcode
  - lhs and rhs operands
    - Either a slot index, a constant value or empty
  - An output slot
- *Slots* index regions of memory for storage of operands and intermediate results

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

(Keeter, 2020, Table 1)

Generating the initial Tape

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

(Keeter, 2020, Table 1)

- The sequence of opcodes can be easily retrieved by sorting the syntax tree topologically[1]
- A fixed number of slots is allocated upfront
- Assigning slots is equivalent to the well-known register allocation problem (Chaitin et al., 1981)
- Tape is stored in a compact, binary format[2], suitable for streaming onto the GPU (only 8 bytes per instruction)

1: Keeter uses the optimized algorithm of Kahn (1962)

2: Please refer to section 5.3.3 in the report for details

Interpreter

- The opcodes are implemented on the GPU with nVIDIA CUDA using interval arithmetic
- A GPU thread runs an interpreter loop to evaluate the tape
- For MIN and MAX opcodes the *choices* are stored for the tape pruning step

Adapted from (Keeter, 2020, Algorithm 1).

Precondition: Slots of the first input variable instructions initialized with the corresponding input interval vector $\mathbf{X}$
Input: tape, preallocated array of slots
Output: result interval, stack of choices

```
choices ← an empty stack
foreach clause in tape do
    lhs ← getValue(clause.lhs)
    rhs ← getValue(clause.rhs)
    switch clause.opcode do
        case OP_MIN do
            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 do
            Similar logic to push a choice ...
            slots[clause.out] ← max (lhs, rhs)
        case OP_ADD do
            slots[clause.out] ← lhs + rhs
        case OP_SUB do
            slots[clause.out] ← lhs − rhs
        ... and so on for other opcodes
clause ← the last clause in the tape
return (slots[clause.out], choices)
```

Adapted from (Keeter, 2020, Algorithm 2)

Tape pruning

- Tapes can be greatly simplified when they contain CSG operations, i.e. MIN and MAX opcodes
- After evaluation for a parent tile:
  - If tile is ambiguous, try to shorten the tape given to the children
  - If the choice made for a MIN or MAX opcode is consistent for the *whole* tile, we can prune the inactive branch by replacing the corresponding operand by the active one

```
Input: tape, choices
Output: shortened output tape
output ← empty tape
active ← array of all false, one item per slot
active[final output slot] ← true
foreach clause in tape.reversed() do
    choice ← CHOICE_BOTH
    if clause.opcode ∈ [OP_MIN, OP_MAX] then
        choice ← choices.pop()
    if active[clause.out] then
        active[clause.out] ← false
        if choice == CHOICE_LHS then
            active[clause.lhs] ← true
            clause.rhs ← clause.lhs
        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
```

Hierarchical Evaluation, 2D-Case

- Subdivide the area of interest into a set of *tiles*
- Evaluate every tile in parallel. If a tile is completely inside, mark occupied. If completely outside, ignore. Otherwise the tile is *ambiguous.*
- For all ambiguous tiles in parallel, try to shorten the tape using Alg. 2.

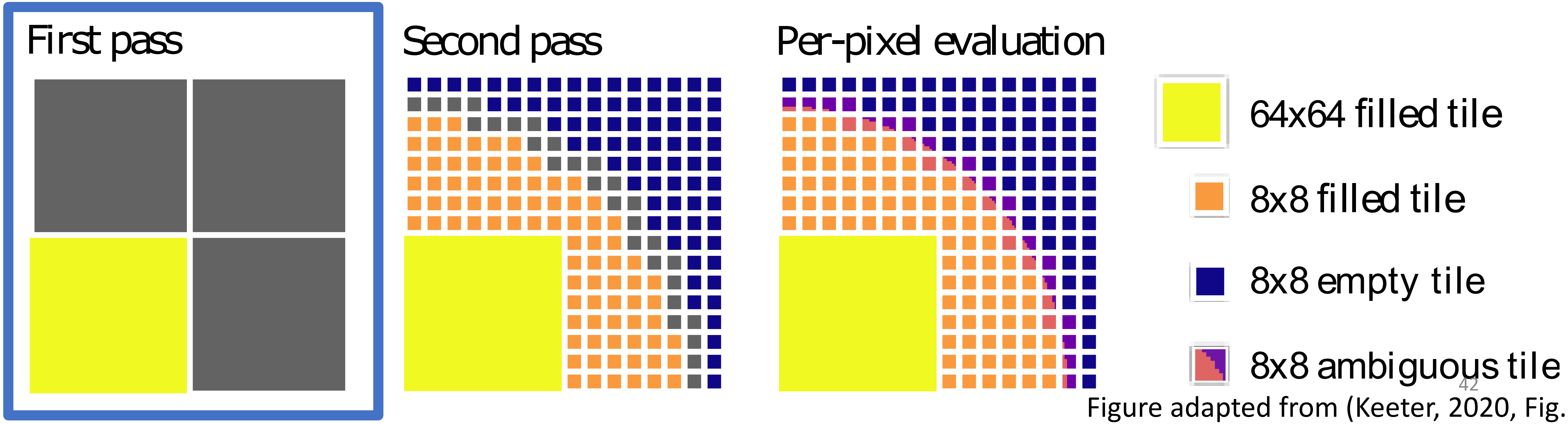
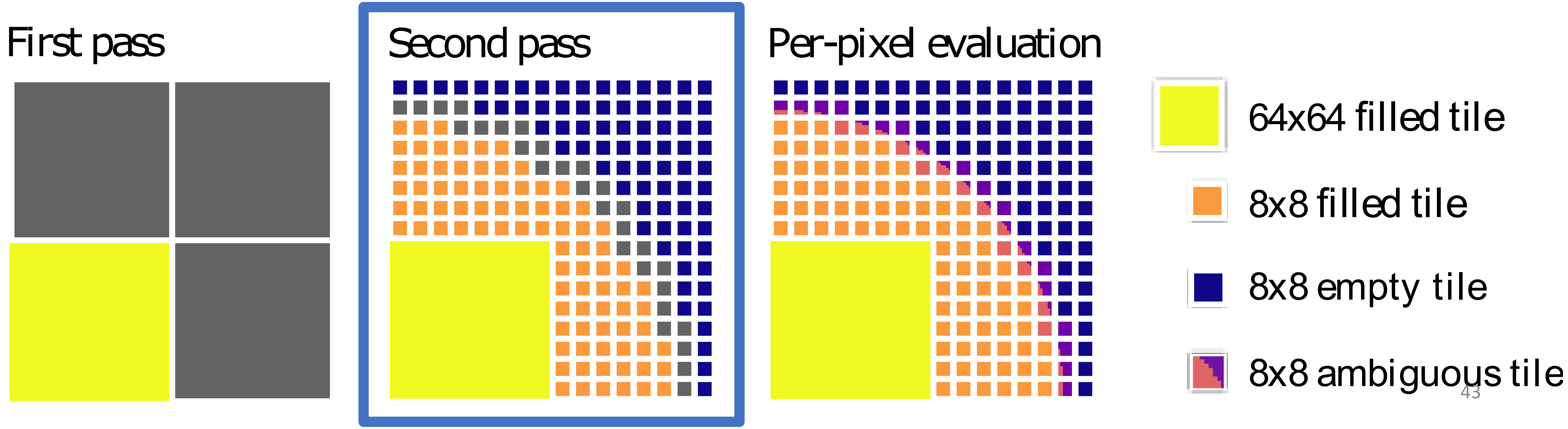


Figure adapted from (Keeter, 2020, Fig. 2)

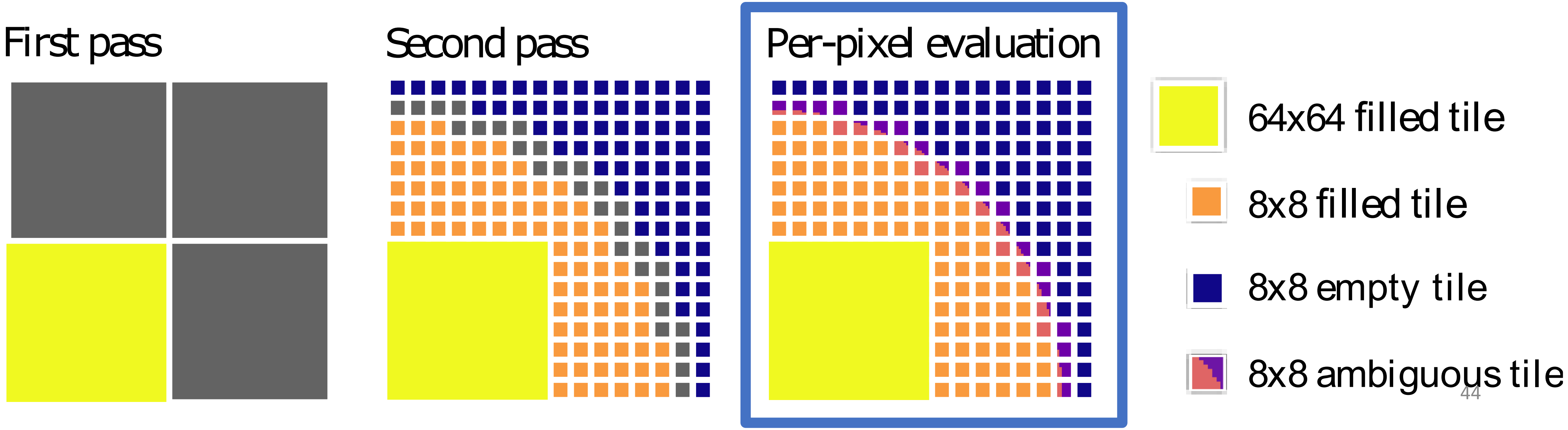
Hierarchical Evaluation, 2D-Case

- Subdivide ambiguous tiles from the last step. For each subtile in parallel:
  - Evaluate the shortened tape from the last step
  - Mark unambiguously occupied tiles, shorten ambiguous tapes…



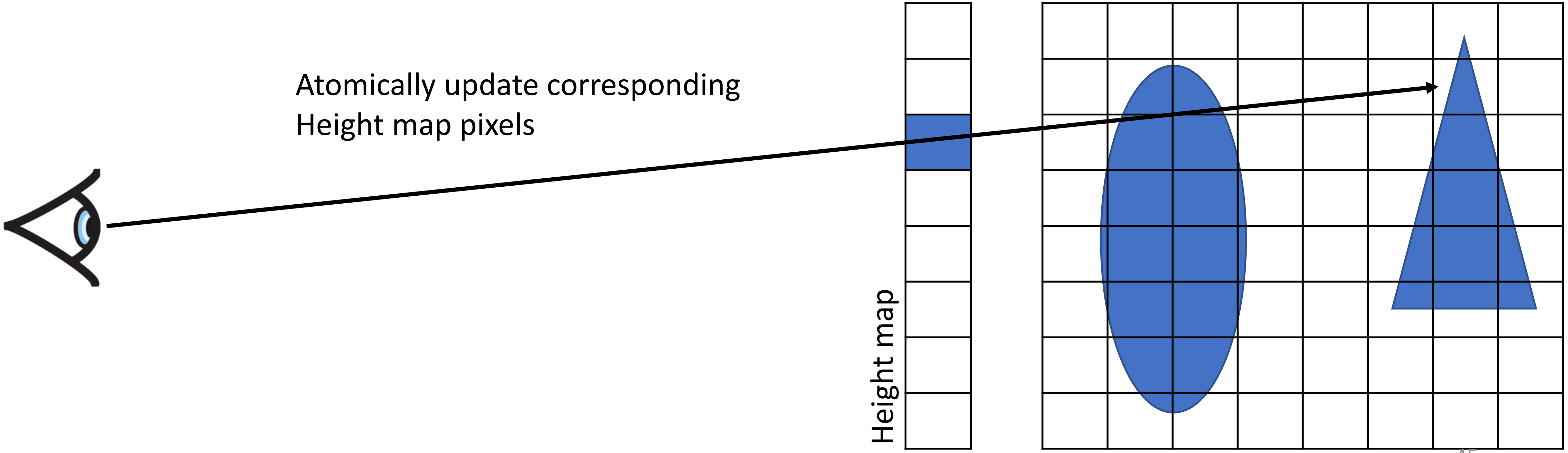
Hierarchical Evaluation, 2D-Case

- For every *pixel* of the ambiguous subtiles from last step in parallel:
  - Evaluate the shortened tape from the last step
  - Mark occupied pixels



3D Rendering

- Just like the 2D-case but one more subdivision step
- Instead of marking pixels occupied, update a height map using atomic max operations

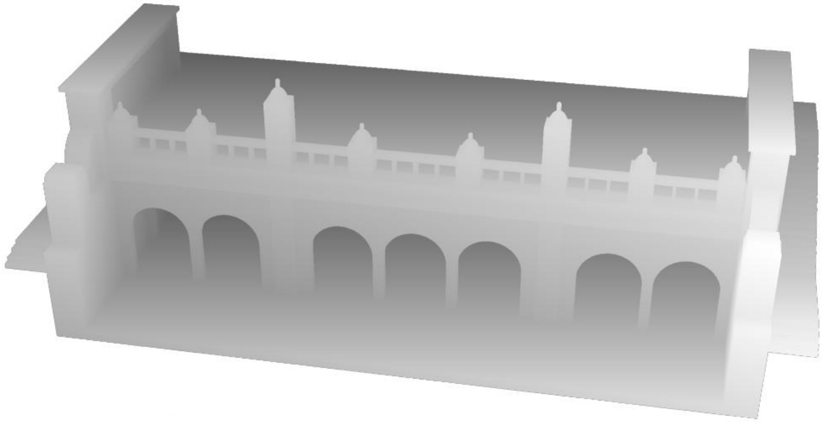


3D Rendering: Camera Transform

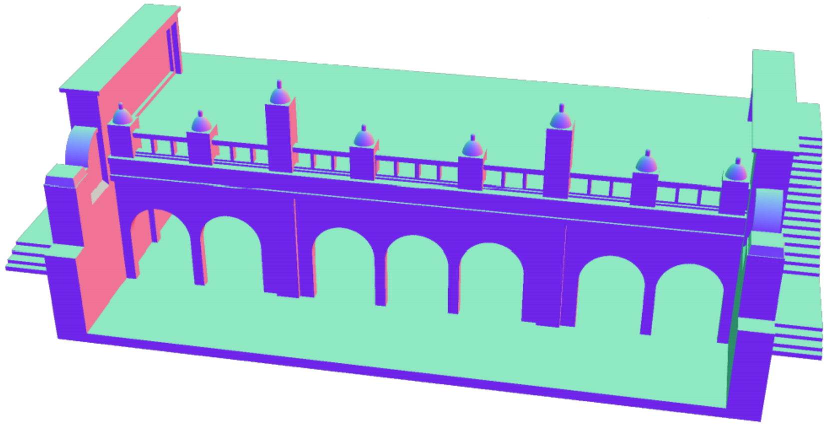
- Camera transform is easy, just apply the inverse of the camera transform and projection to the input coordinates
- OpenGL “projection” is actually invertible if the depth buffer is used

3D Rendering

- What's left is calculating the surface normals.
- Two options:
  - Apply some finite difference scheme to the height map
  - What Keeter did: Use automatic differentiation of the shortened tape closest to the respective point on the surface
    ⇒ Exact normals!



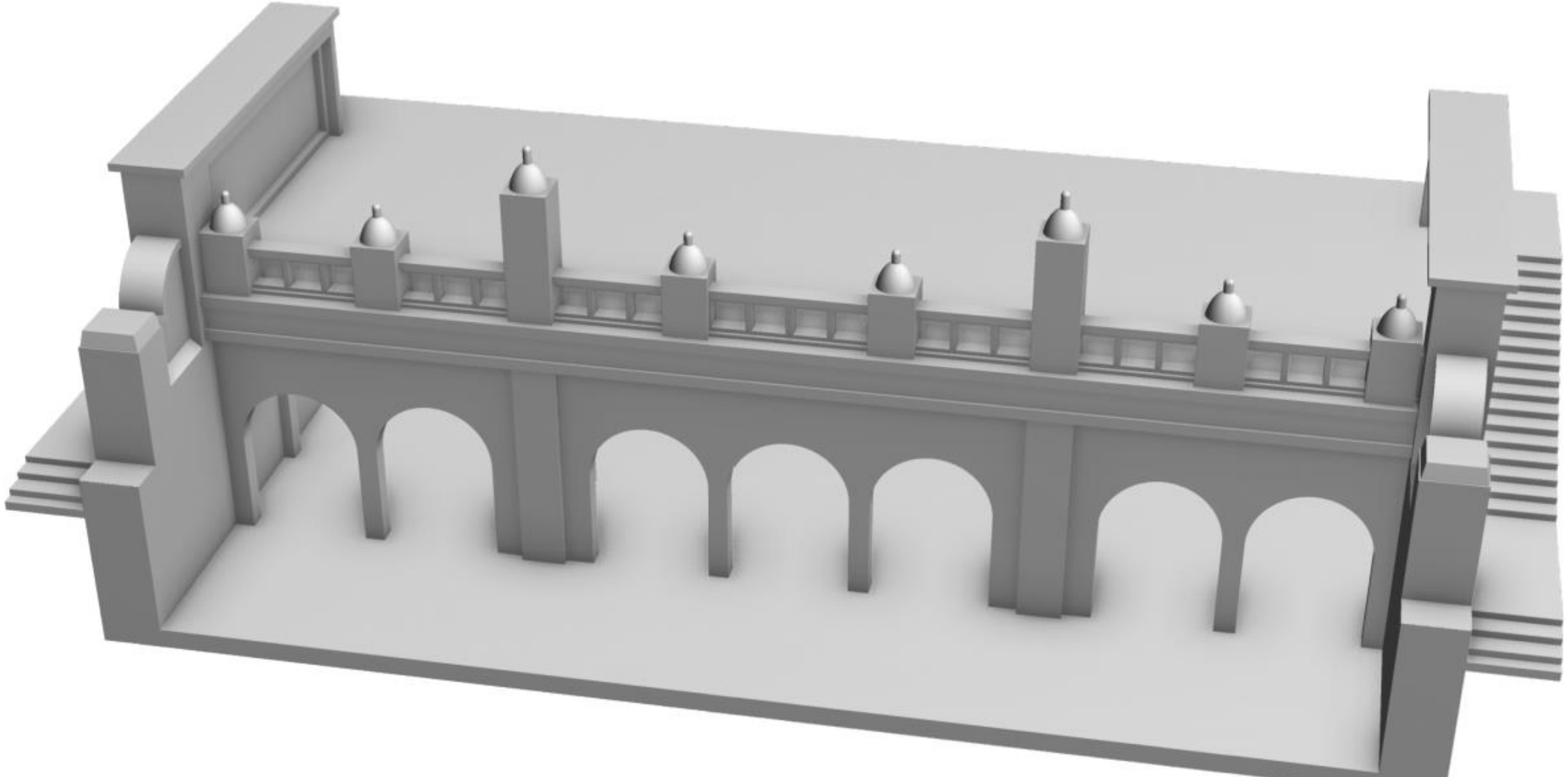
(Keeter, 2020, Fig. 7, top)



(Keeter, 2020, Fig. 7, bottom)

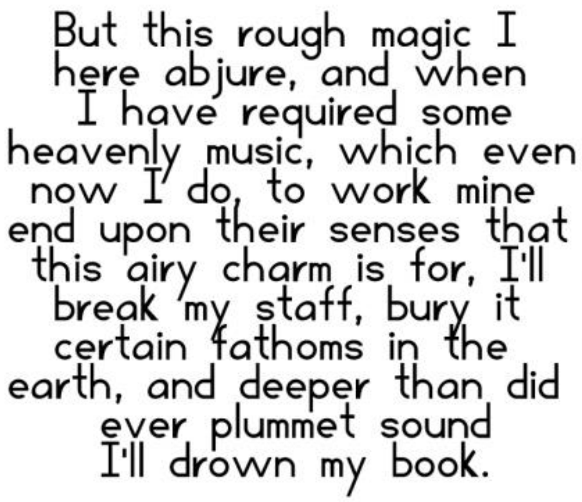
3D Rendering

- With both height and normal map at hand we can apply the standard deferred rendering scheme:

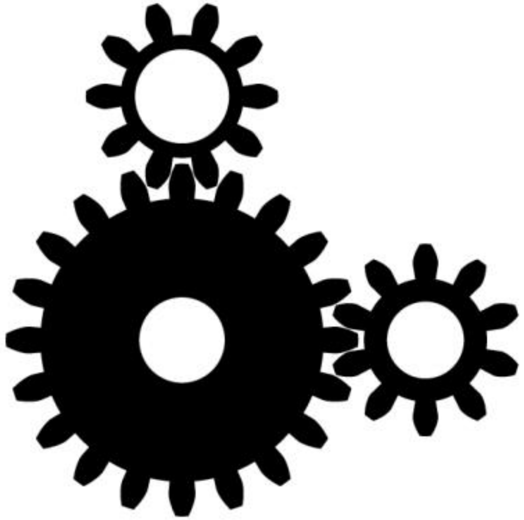


(Keeter, 2020, Fig. 1, right)

Evaluation



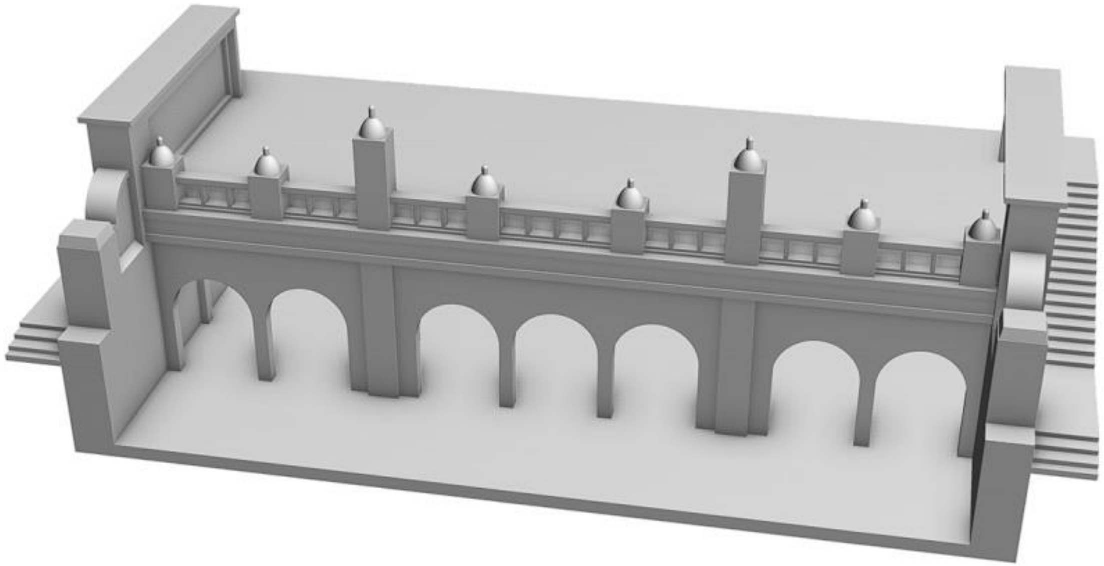
(a) Text benchmark



(b) Gears 2D

(c) Hello world 3D text

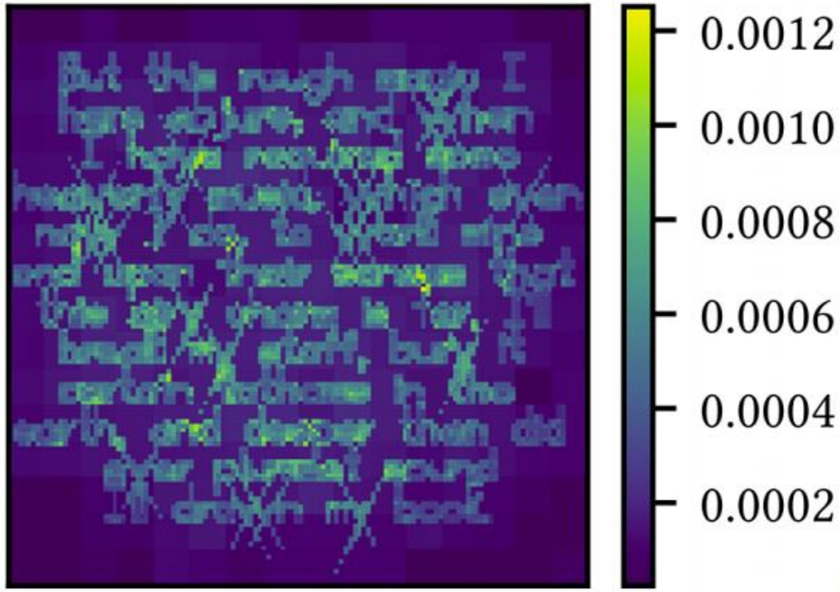
(d) Bear head sculpt



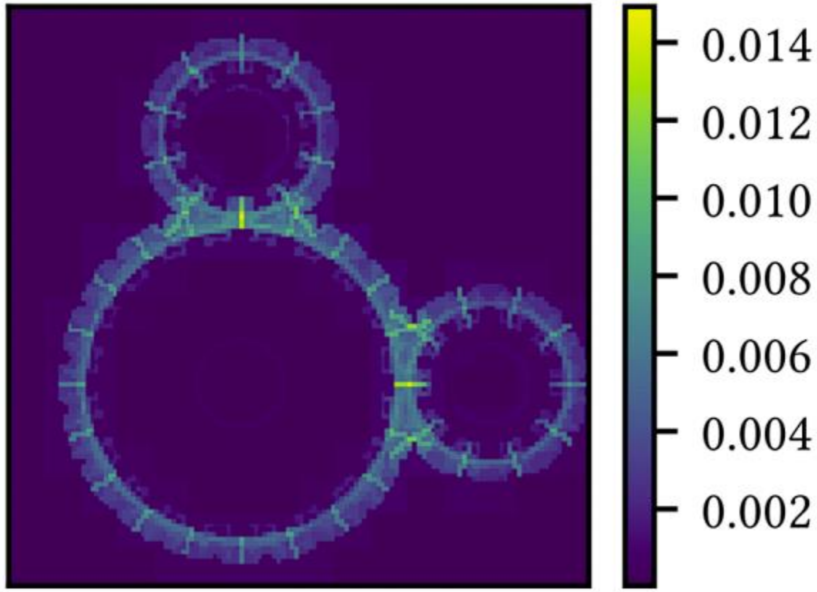
(e) Architectural model, with SSAO

(f) Gears 3D

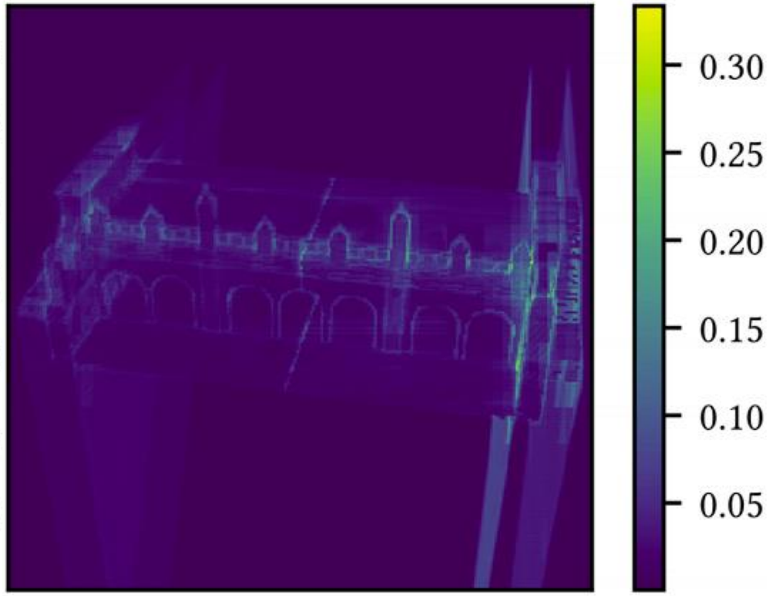
Adapted from (Keeter, 2020, Figs. 1, 4, 8)



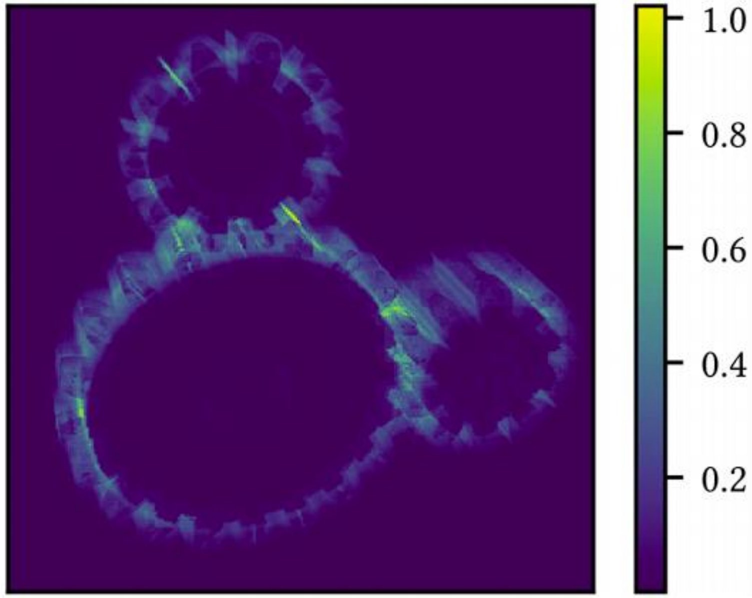
(a) Text benchmark



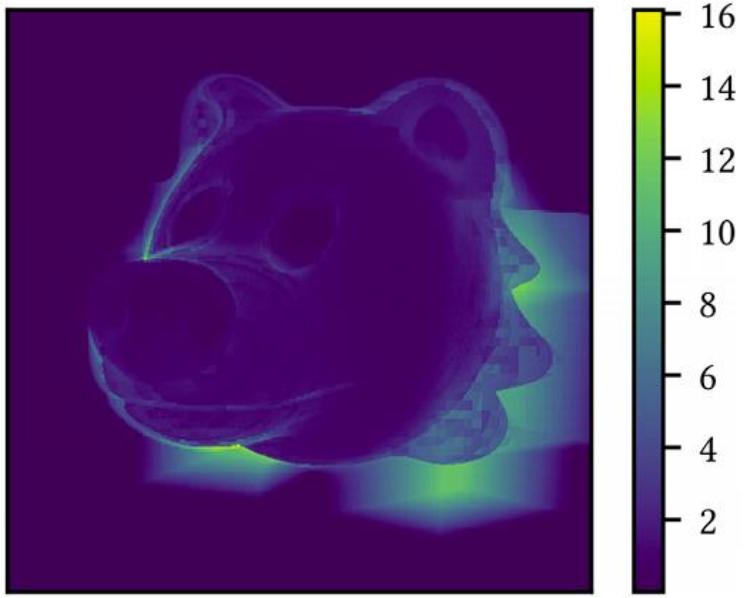
(b) Gears 2D



(c) Architectural model



(d) Gears 3D



(e) Bear head sculpt

Adapted from (Keeter, 2020, Figs. 9, 10)

Performance Numbers

Size	GeForce GT 750M	GTX 1080 Ti	Tesla V100
256^2	17.5	8.3	5.2
512^2	14.5	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

(a) Text benchmark. Frametimes in ms.

Size	GeForce 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

(b) Gears 2D. Frametimes in ms.

2D Results, adapted from (Keeter, 2020, Table 4)

Size	GeForce 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

(a) Architectural model. Frametimes in ms.

Size	GeForce 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

(b) Gears 3D. Frametimes in ms.

Size	GeForce 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	88.2
1536^3	–	504.2	228.3
2048^3	–	1053.2	437.3

(c) Bear head sculpt. Frametimes in ms.

3D Results, adapted from (Keeter, 2020, Table 5)

Discussion

Benefits

- Highly efficient translation of the method of Duff (1992) to the GPU
- Very effective if the number of CSG operations is high
- High branching factor and shallow recursion depth
- Other uses:
  - Efficient voxelization tool
- Interpreter leads to less complex GPU code
  - No more implementing dozens of primitives by hand!

Limitations

- The method is limited to closed-form algebraic expressions
  - Sampled representations like voxel grids, octrees or ASDFs could be used as a “black box”, but tape pruning wouldn’t have any effect
  - Basically the same as evaluating the data structure directly
  - But: Could be still useful for unification of the approaches
- Less effective when number of CSG operations is low or the primitives have large support (as in the “Bear head sculpt” case)
- In 3D the extra dimension adds computational complexity. Might be too slow in some cases for real-time *and* high-resolution application.

Future Work

- Use the method as a low-resolution voxelization tool as pre-process for classical techniques
- Use more advanced formalism like affine arithmetic (Fryaznikob et at., 2010) to achieve tighter bounds
- Height map generation could be optimized by appropriate culling mechanism
- An efficient extension of sampled representations like hierarchical voxel data would be desirable

Questions?

References

Chaitin, G. J., M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein (1981). “Register allocation via coloring”. In: *Computer languages* 6.1, pp. 47–57.
Duff, T. (1992). “Interval arithmetic recursive subdivision for implicit functions and constructive solid geometry”. In: *ACM SIGGRAPH computer graphics* 26.2, pp. 131–138.
Dyllong, E. and C. Grimm (2007). “Verified Adaptive Octree Representations of Constructive Solid Geometry Objects.” In: *SimVis*. Citeseer, pp. 223–236.
Frisken, S. F., R. N. Perry, A. P. Rockwood, and T. R. Jones (2000). “Adaptively sampled distance fields: A general representation of shape for computer graphics”. In: *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pp. 249–254.
Fryazinov, O., A. Pasko, and P. Comninos (2010). “Fast reliable interrogation of procedurally defined implicit surfaces using extended revised affine arithmetic”. In: *Computers & Graphics* 34.6, pp. 708–718.
Greß, A. and R. Klein (2004). “Efficient representation and extraction of 2-manifold isosurfaces using kd-trees”. In: *Graphical Models* 66.6, pp. 370–397.
Hart, J. C. (1996). “Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces”. In: *The Visual Computer* 12.10, pp. 527–545.
Hart, J. C., E. Bachta, W. Jarosz, and T. Fleury (Aug. 2002). “Using Particles to Sample and Control More Complex Implicit Surfaces”. In: *SMI ’02: Proceedings of the Shape Modeling International 2002 (SMI’02)*. Washington, DC, USA: IEEE Computer Society, p. 129.
“IEEE Standard for Floating-Point Arithmetic” (2019). In: *IEEE Std 754-2019 (Revision of IEEE 754-2008),* pp. 1– 84.
Ju, T., F. Losasso, S. Schaefer, and J. Warren (2002). “Dual contouring of hermite data”. In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pp. 339–346.
Kahn, A. B. (1962). “Topological sorting of large networks”. In: *Communications of the ACM* 5.11, pp. 558–562.
Keeter, M. J. (2020). “Massively parallel rendering of complex closed-form implicit surfaces”. In: *ACM Transactions on Graphics (TOG)* 39.4, pp. 141–1.
Kobbelt, L. P., M. Botsch, U. Schwanecke, and H.-P. Seidel (2001). “Feature sensitive surface extraction from volume data”. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 57–66.
lex(1p) — Linux manual page (2021). url: https://man7.org/linux/man-pages/man1/lex.1p.html (visited on 15/03/2021).
Lorensen, W. E. and H. E. Cline (1987). “Marching cubes: A high resolution 3D surface construction algorithm”. In: *ACM siggraph computer graphics* 21.4, pp. 163–169.
May, S., P. Koch, R. Koch, C. Merkl, C. Pfitzner, and A. Nüchter (2014). “A Generalized 2D and 3D Multi-Sensor Data Integration Approach based on Signed Distance Functions for Multi-Modal Robotic Mapping.” In: *VMV*, pp. 95–102.
Melquiond, G., S. Pion, and H. Brönnimann (2006). (Boost) Interval Arithmetic Library. url: https://www.boost.org/doc/libs/1_71_0/libs/numeric/interval/doc/interval.htm (visited on 15/03/2021).
Moore, R. E., R. B. Kearfott, and M. J. Cloud (2009). *Introduction to interval analysis.* SIAM.
Seyb, D., A. Jacobson, D. Nowrouzezahrai, and W. Jarosz (2019). “Non-linear sphere tracing for rendering deformed signed distance fields”. In: *ACM Transactions on Graphics* 38.6.
Witkin, A. P. and P. S. Heckbert (1994). “Using particles to sample and control implicit surfaces”. In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 269–277.
yacc(1p) — Linux manual page (2021). url: https://man7.org/linux/man-pages/man1/yacc.1p.html (visited on 15/03/2021).