UNIVERSITY OF COPENHAGEN
DEPARTMENT OF COMPUTER SCIENCE

Bachelor Thesis

Shamim Siddique

Tile-based software rasterization in Futhark

June 12, 2026

Advisor: Troels Henriksen

Abstract

Rasterizing triangles with edge functions requires per-point testing, which prevents precomputing exact output sizes and local segment indices for a collection of triangles. As a workaround, small, hierarchically structured bitmasks can be used to record and query this information.

This thesis extends the flattening-by-expansion technique for handling irregular parallelism in Futhark by introducing `expand_masked`, an expand-filter function based on bitmasks, enabling efficient filtering of small, bounded segment sizes on a per-segment-element basis.

Using `expand_masked`, an edge-function-based rasterizer can be expressed efficiently in a data-parallel style. While it is slightly slower than a scanline rasterizer and scales less favorably beyond 1024×1024 resolution, a tile-based variant matches the scanline rasterizer at 8192×8192 while requiring substantially less peak memory in general.

Contents

1 Introduction

Geometry offers two complementary ways to represent a 2D region: explicitly through its boundary, or implicitly through constraints on its points. A circle of radius r at the origin, for example, can be defined in either form as follows:

Explicit / Parametric	**Implicit**
$x(\theta) = r\cos\theta$	$f(x,y) = x^2 + y^2 - r^2$
$y(\theta) = r\sin\theta$	s.t. $f(x,y) = 0$

This distinction naturally extends to triangles. A triangle can be defined explicitly by its edges. Triangle rasterization is the task of determining which discrete points lie inside a triangle. Scanline rasterization exploits this explicit representation by traversing along the edges and filling the horizontal spans in between. Pineda rasterization, by contrast, treats the triangle implicitly through its edge functions and classifies sampled points by their position relative to each edge. Here, only the points inside the triangle's bounding box are sampled to reduce the number of unnecessary tests.

Rasterizing an arbitrary number of triangles is irregular and computationally demanding: there is no predictable pattern that determines how many or which grid points each triangle will cover on a screen grid. Henriksen et al. [17] describes a "flattening-by-expansion" approach for a class of irregular problems solved in a data-parallel manner through the design API `expand` in Futhark. This approach is successfully used for scanline rasterization, so the question naturally extends to whether it can also be applied to Pineda rasterization.

Additionally, it would be desirable to "tame" the irregularity of rasterization and exploit locality by assigning triangles to particular screen regions for performance. Tile-based rasterization is a natural next step once rasterization through edge functions is efficiently achievable in Futhark.

1.1 Project overview

The background section first establishes Pineda rasterization. Then it describes how it can be performed in a data-parallel manner in addition to describing sort-middle, tile-based rendering.

The design and implementation section describe a data-parallel hierarchical tiling algorithm done using `expand_masked`, an expand-filter implementation. It explains how `expand_masked` can be applied hierarchically despite only supporting small, bounded segment sizes.

The section then proceeds to describe different approaches to depth resolution for determining which triangle should be selected for each pixel.

Finally, it describes how tile-based rasterization is performed, evaluates different implemented rasterizers in terms of performance and memory usage, and suggest directions for future work.

1.2 Acknowledgments

Thanks to the public Stanford repository for the Rabbit, Dragon, Angel, and Armadillo models.

Thanks to Martin Newell for the Utah teapot model in 1975.

Thanks to Blender for the Suzanne monkey model.

And thanks to Max-Kawula for the Penger model.

The hedgehog model is an *artisanally crafted* work of programmer art by the author.

1.3 Other work

`https://github.com/abxh/lys` is a fork that extends the Futhark library to be able to load `.obj` files.

Under the writing of this thesis, some bugs were found as well:
see this `u64.set_bit`/`i64.set_bit` issue,
this globally defined bindings in non-inlined functions issue,
this merge sort deleting maximal elements issue,
and this edge case for modules, the latter reported to the thesis advisor.

2 Background

This section describes the theory underlying the rest of the thesis. First it describes the problem of rasterization, how Pineda rasterization is performed and describes an basic, programmable 3D rendering pipeline. Then it presents parallel rendering from a data-parallel perspective, illustrating key trade-offs and motivating its implementation in Futhark. Appendix B.1 provides additional background about how three-dimensional triangles can be rendered using rasterization, including the transformations required for 3D rendering. Appendix B.2 describes how pixels are shaded during rendering.

2.1 Rasterization

The problem of rasterization dates back to the early days of computing, when there arose a need to display text and geometric shapes on digital screens [40]. Rasterization is usually centered around the triangle, as the triangle is the simplest polygon that uniquely defines a plane and can be composed to form more complex polygons.

Rasterization is fundamentally the process of determining the discrete set of points on a grid that represent a given two-dimensional primitive (such as a line segment or triangle).

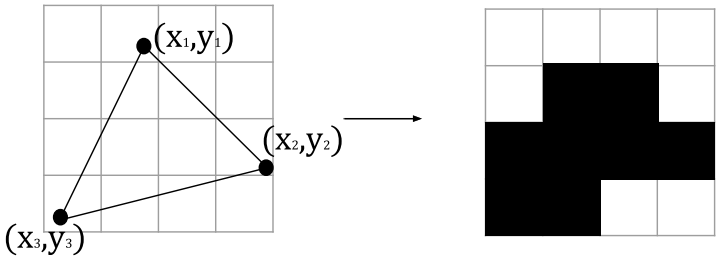
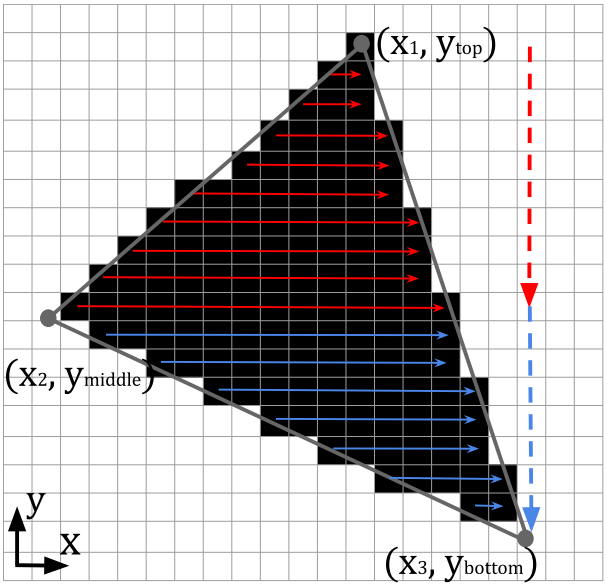


Figure 1: Triangle rasterization of covered cell midpoints

The task is usually not to rasterize a single primitive, but thousands of primitives at once, preferably as efficiently as possible. A single triangle primitive (defined by three vertex positions) may cover zero, one, or an arbitrary number of discrete points on a grid. As such, determining the set of points covered by a collection of triangles is highly irregular. The same reasoning applies to line segments as well.

There are two conventional approaches to triangle rasterization. The standard approach, referred to as *Scanline rasterization* in this paper, belongs to a family of methods early described by Wylie et al. [47] in 1967, splits a triangle at the middle vertex (neither the top nor the bottom) into two sub-triangles. Edge slope information is then computed for edge traversal, and the rasterization proceeds by traversing from the top vertex to the bottom while filling horizontal spans between the edges. Omitting the slope computations, the gist of the algorithm is as follows.



1. Iterate over y from y_{top} to y_{middle} and from $y_{middle} + 1$ to y_{bottom}.
2. For each y, calculate the left and right edge intersections x_{left} and x_{right} using the edge slopes.
3. For each (x_{left}, x_{right}) pair, fill the horizontal span between them at y.

Figure 2: Scanline rasterization

The above algorithm traverses only the cells (or pixels) covered by the triangle, thereby avoiding unnecessary work. However, as Giesen [12] notes,

> the algorithm is asymmetrical in x and y, which means that a very skinny triangle that's mostly horizontal has a very different performance profile from one that's mostly vertical.

In contrast, Pineda rasterization, introduced next, is generally simpler to implement (at least in its unoptimized form), supports better sub-pixel precision, and adapts more naturally to additional rasterization constraints, such as triangle fill convention, pixel multisampling, pixel-quad traversal, tile-based traversal, and locality-optimized access patterns [12].

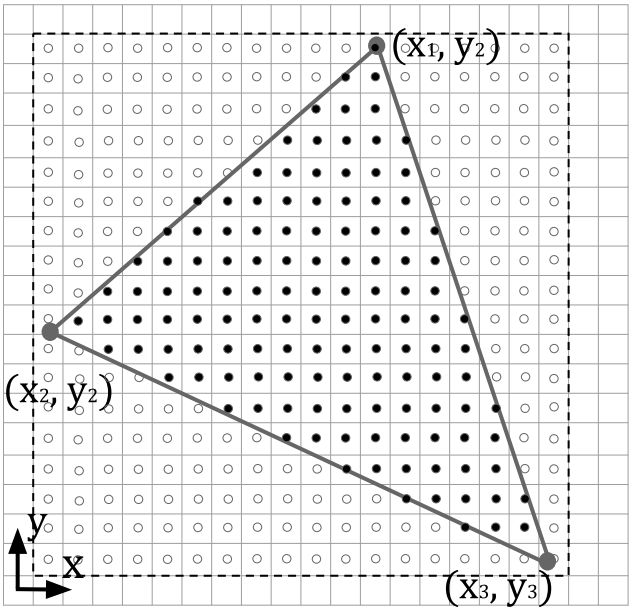
Pineda rasterization, named after the paper by Pineda [34] from 1988, is centered around analytically-derived triangle edge functions. An *edge function* is a linear function $f : \mathbb{R}^2 \to \mathbb{R}$ whose sign indicates the side of an edge a sampled point (x, y) lies on:

$$\begin{aligned} f(x,y) > 0 &\implies (x,y) \text{ is on the "left" side of the edge} \\ f(x,y) = 0 &\implies (x,y) \text{ lies exactly on the edge} \\ f(x,y) < 0 &\implies (x,y) \text{ is on the "right" side of the edge} \end{aligned} \tag{1}$$

Carefully choosing the edge functions such that the half-space corresponding to the "left" side contains the triangle interior for each edge (as described in the next sections), it becomes possible to define a logical predicate $P(x, y)$ that determines whether a sampled point (x, y) lies inside the triangle.

$$P(x,y) = \big(f_0(x,y) \geq 0 \wedge f_1(x,y) \geq 0 \wedge f_2(x,y) \geq 0\big) \tag{2}$$

For efficiency, sampling can be restricted to points within the triangle bounding box rather than the entire screen grid. A high-level description of Pineda rasterization is as follows. As illustrated in Figure 2, a cell is considered inside the triangle if its midpoint lies inside the triangle.



$$\begin{aligned} x_{min} &= \lfloor \min\{x_1, x_2, x_3\} \rfloor \\ y_{min} &= \lfloor \min\{y_1, y_2, y_3\} \rfloor \\ x_{max} &= \lceil \max\{x_1, x_2, x_3\} \rceil \\ y_{max} &= \lceil \max\{y_1, y_2, y_3\} \rceil \end{aligned}$$

1. Assigning above variables, iterate over y from y_{min} to y_{max}. For each y, iterate over x from x_{min} to x_{max}.
2. For each (x, y) pair, evaluate $P(x + 0.5, y + 0.5)$. If the predicate evaluates to true, fill the cell; otherwise, continue.

Figure 3: Pineda rasterization

Performing a row-major traversal of the bounding box, as described in Figure 5, results in unnecessary work, since around half of the traversed cells typically lie outside the triangle. Restricting the set of sampled points to the bounding box makes the asymptotic runtime complexity of scanline and Pineda rasterization similar, but the latter still incurs a significantly larger constant factor in its basic form.

Paraphrasing Giesen [12] in the following paragraph: Pineda [34] described smarter ways to traverse the points in the bounding box, at the cost of extra logic and cost/performance tradeoffs. McCormack and McNamara [30] describes more efficient traversal schemes based on tiles, and Greene [14] takes it further and divides the screen into regular, hierarchical tiles. Abrash [1] independently discovered the same approach while working on Larrabee, a canceled Intel GPGPU project from the late 2000s, combined CPU-like x86 compatibility and cache coherence with GPU-like SIMD vector units and texture sampling hardware [46].

The idea is as follows: divide the screen into a hierarchical grid and assign triangles first to coarse regions and then progressively refine into smaller subregions. The same row-major traversal can be applied hierarchically, first over coarse grids and then over finer grids.

Edge functions provide a simple and efficient method for testing triangle-rectangle subregion overlap. A quick, conservative bounding-box test can be performed first, followed by evaluating each of the triangle's edge functions at the four corners of the rectangle. If all four corners lie outside any single edge, the rectangle is considered to *not* overlap the triangle [1].

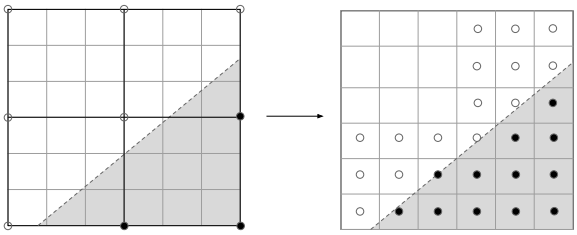


Figure 4: Hierarchical tiling / rasterization of 3x3 tiles.

Figure 4 is just an illustrative example, but the idea can be carried out across larger tiles. This way, the number of wasted evaluations can be roughly reduced logarithmically.

More importantly, by dividing the screen into hierarchical tiles and assigning triangles to those tiles, one gains additional flexibility and control over how the rasterization workload is distributed. A large triangle can be assigned to one or more smaller, manageable tile-sized screen regions corresponding to smaller units of work, while smaller triangles typically can be assigned to a single tile-sized screen region, thereby not treating the triangles themselves but more uniform work items derived thereof as the primary unit of work [13].

The problem remains inherently irregular and challenging: a triangle may overlap zero, one, or multiple tiles, while an arbitrary number of triangles are processed simultaneously, with multiple triangles potentially overlapping the same tile. As a result, the rasterization problem exhibits a many-to-many relationship between triangles and tiles.

Regardless of the strategy used to handle irregular, variable-sized work, a key question remains: how should triangle-tile overlaps be represented efficiently? This is commonly achieved using bitmasks, typically fixed-size and compact representations of binary coverage information, known as *coverage masks* [1, 14].

These coverage masks are generally small, since a separate mask is maintained at each level of the hierarchy. Furthermore, their bits can be set and queried efficiently using bitwise operations and hardware-accelerated instructions, making them a compact yet efficient data structure for representing coverage information.

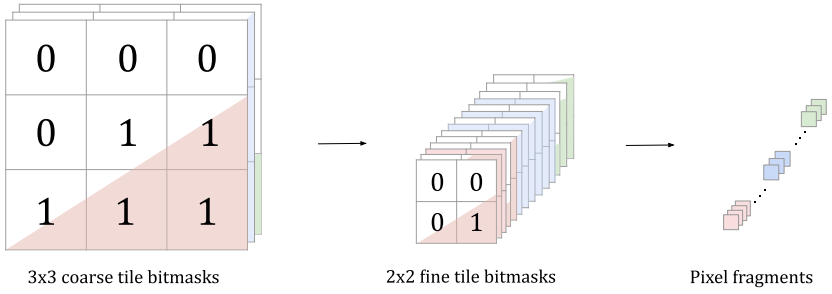


Figure 5: Hierarchical rasterization using coverage masks on 3×3 coarse tiles, with 2×2 fine tiles sampled at midpoints to generate pixel fragments.

This concludes the high-level overview of hierarchical rasterization. The following sections detail edge functions and aspects of Pineda rasterization.

2.1.1 Cross product

As a prelude to the description of edge functions (described in the next section), this section reviews a few important properties of the cross product, and defines the syntactical construct `cross2D`.

Given two vectors $\boldsymbol{u} = \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix}^T$ and $\boldsymbol{v} = \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}^T$, the 3D cross product can be found by finding associated constants with the basis vectors $\hat{i}, \hat{j}, \hat{k}$ in the following equation:

$$\boldsymbol{u} \times \boldsymbol{v} = \begin{vmatrix} \hat{i} & u_1 & v_1 \\ \hat{j} & u_2 & v_2 \\ \hat{k} & u_3 & v_3 \end{vmatrix} \tag{3}$$

Proceeding as if the basis vectors were numbers [35], the cross product can be computed as the following:

$$\begin{aligned} \boldsymbol{u} \times \boldsymbol{v} &= \hat{i} \begin{vmatrix} u_2 & v_2 \\ u_3 & v_3 \end{vmatrix} - \hat{j} \begin{vmatrix} u_1 & v_1 \\ u_3 & v_3 \end{vmatrix} + \hat{k} \begin{vmatrix} u_1 & v_1 \\ u_2 & v_2 \end{vmatrix} \\ \Leftrightarrow \boldsymbol{u} \times \boldsymbol{v} &= \begin{bmatrix} u_2 v_3 - u_3 v_2 & u_3 v_1 - u_1 v_3 & u_1 v_2 - u_2 v_1 \end{bmatrix}^T \end{aligned}$$

The cross product $\boldsymbol{u} \times \boldsymbol{v}$ has two important properties:

- It's magnitude is the signed area of the parallelogram spanned by the two vectors $\boldsymbol{u}$ and $\boldsymbol{v}$.
- It is perpendicular to both $\boldsymbol{u}$ and $\boldsymbol{v}$ in the direction following the right hand rule.

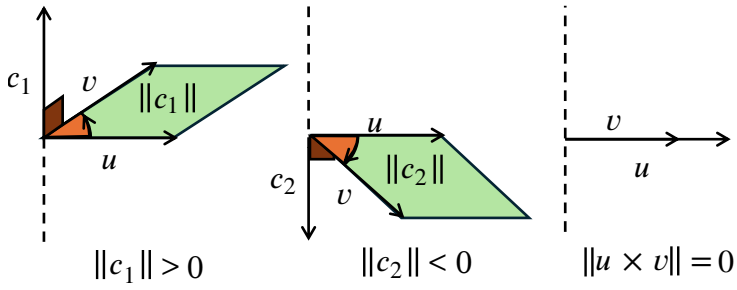


Figure 6: Properties of the cross product.

If both the operands lie on the XY-plane i.e. $v_3 = u_3 = 0$, then the magnitude of the cross product is equal to the determinant of the following matrix, which is to be defined as `cross2D`:

$$\begin{aligned} \texttt{cross2D}\left(\begin{bmatrix} u_1 & u_2 \end{bmatrix}^T, \begin{bmatrix} v_1 & v_2 \end{bmatrix}^T\right) &= \| \begin{bmatrix} u_1 & u_2 & 0 \end{bmatrix}^T \times \begin{bmatrix} v_1 & v_2 & 0 \end{bmatrix}^T \| \\ &= \begin{vmatrix} u_1 & v_1 \\ u_2 & v_2 \end{vmatrix} = u_1 v_2 - u_2 v_1 \end{aligned} \tag{4}$$

2.1.2 Edge function

This section constructs an edge function that satisfies Equation 1.

Define an edge by vectors $\boldsymbol{v_0} = [x_0, y_0]^T$ and $\boldsymbol{v_1} = [x_1, y_1]^T$, such that $\boldsymbol{v_0 v_1} = [x_1 - x_0, y_1 - y_0]^T$ lies on the edge. Using `cross2D`, the edge function f can be constructed as follows with $\boldsymbol{p} = \begin{bmatrix} x & y \end{bmatrix}^T$:

$$\begin{aligned} f(\boldsymbol{p}) &= \texttt{cross2D}\left(\boldsymbol{v_0 v_1}, \boldsymbol{v_0 p}\right) \\ \Leftrightarrow f(x, y) &= (x_1 - x_0)(y - y_0) - (y_1 - y_0)(x - x_0) \\ &= Ax + By + C \end{aligned}$$

, where $A = y_1 - y_0$, $B = x_1 - x_0$ and $C = Ax_0 + By_0$.

Note how this is not the only possible edge function for edge $\boldsymbol{v_0 v_1}$. In fact, any function of the form $c \cdot f(x, y)$ where $c > 0$, satisfies the definition of the edge function. However f as constructed above has a particular scaling factor, which is useful for defining barycentric coordinates (to be defined shortly).

Additionally, since the edge function f can be written as $Ax + By + C$, it can be visualized as a three-dimensional plane, which intersects the edge at $z = 0$, and whose height $z = f(x, y)$ changes sign on either side of the edge.

Following Pineda [34], define the half-space of the edge in the XY-plane where $f(x, y) > 0$ to be the "left" side of the edge, and the complementary half-space as the "right" side of the edge, with respect to the oriented edge $\boldsymbol{v_0 v_1}$. These labels aren't necessary geometry meaningful, but serves as labels to refer to a particular side of the edge with respect to the sign of $f(x, y)$.

The following figure shows both the vector setup and the plane visualization.

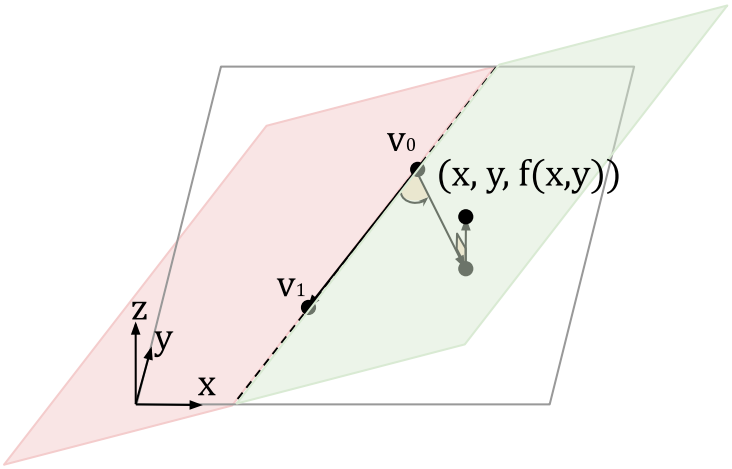


Figure 7: The edge function.

Because $f(x, y)$ can be written in the form $Ax + By + C$, it can be evaluated incrementally. Instead of recomputing the edge function at every point, a more imperative implementation evaluates a single edge rasterization by reusing intermediate results within nested loops, as shown in the following pseudocode. This reduces the computation to simple additions and memory accesses, making Pineda rasterization well-suited for hardware acceleration.

```
w = w0 = f(x_min, y_min)
inc_x = y1 - y0
inc_y = x1 - x0
for y = y_min to y_max do
    w = w0
    for x = x_min to x_max do
        w = w + inc_x
        if w >= 0 then
            plot(x, y)
    w0 = w0 + inc_y
```

It is possible to encode a (triangle) region with multiple edge functions. Note that multiple edge functions can be computed incrementally and independently with the same operations as above. Semantically, `w` can be extended as a math vector with each dimension computing some edge function incrementally, where the branch condition checks `all(`$\boldsymbol{w}$ `>= 0)` instead.

2.1.3 A triangle as a combination of edge functions

This section constructs the edge functions for $P(x, y)$ in Equation 2.

Define a triangle by vertex position vectors $\boldsymbol{v_0} = [x_0, y_0]^T$, $\boldsymbol{v_1} = [x_1, y_1]^T$, and $\boldsymbol{v_2} = [x_2, y_2]^T$.

This section assumes the vertices are defined s.t. the triangle has a counterclockwise winding order i.e. $\texttt{cross2D}\,(\boldsymbol{v_0 v_1}, \boldsymbol{v_0 v_2}) \geq 0$. This ensures the "left" side for each edge function as defined below correspond to the triangle interior; otherwise their signs are inverted. As an implementation note, incorrect winding order can be corrected by swapping two vertices.

A point (x, y) is inside the triangle if the edge functions defining the triangle region are non-negative. Consider the following figure to derive the edge functions. Note how the triangle has a counterclockwise winding order.

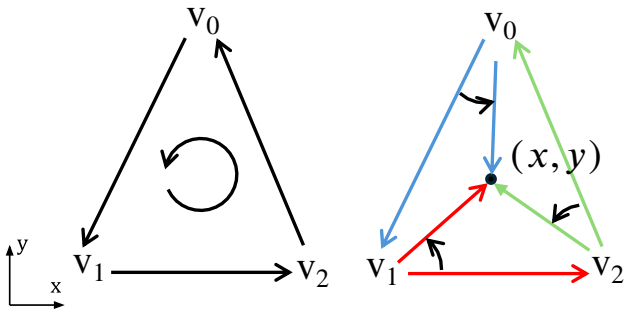


Figure 8: Edge functions defining the region inside triangle

Assuming counterclockwise winding order, a point position vector $\boldsymbol{p} = \begin{bmatrix} x & y \end{bmatrix}^T$ is in the triangle iff. $\boldsymbol{v_1 p}$ is counterclockwise to $\boldsymbol{v_1 v_2}$, $\boldsymbol{v_0 p}$ is counterclockwise to $\boldsymbol{v_2 v_0}$ and $\boldsymbol{v_1 p}$ is counterclockwise to $\boldsymbol{v_0 v_1}$. And the corresponding edge functions f_0, f_1 and f_2 can be constructed as follows:

$$\begin{aligned} f_0(\boldsymbol{p}) &= \texttt{cross2D}(\boldsymbol{v_1 v_2}, \boldsymbol{v_1 p}) \\ f_1(\boldsymbol{p}) &= \texttt{cross2D}(\boldsymbol{v_2 v_0}, \boldsymbol{v_0 p}) \\ f_2(\boldsymbol{p}) &= \texttt{cross2D}(\boldsymbol{v_0 v_1}, \boldsymbol{v_1 p}) \end{aligned} \tag{5}$$

These satisfy the predicate $P(x, y)$ in Equation 2, where if a given point (x, y) is in the triangle, then $f_0(x, y) \geq 0$, $f_1(x, y) \geq 0$ and $f_2(x, y) \geq 0$.

2.1.4 Barycentric coordinates

This section defines barycentric coordinates used for shading purposes later on, reusing the constants in the edge functions.

Let $2A = \texttt{cross2D}(\boldsymbol{v_0v_1}, \boldsymbol{v_0v_2})$, where the signed area $A \geq 0$ by the winding order convention.

From Figure 8 and the property of the cross product that it's magnitude is the (signed) area of the parallelogram spanned by operand vectors, $f_0(\boldsymbol{p})$ represents twice the area of the sub-triangle $\triangle v_1v_2p$. Similarly, $f_1(\boldsymbol{p})$ corresponds to twice the area of $\triangle v_2v_0p$, and $f_2(\boldsymbol{p})$ corresponds to twice the area of $\triangle v_0v_1p$.

This naturally leads to a barycentric coordinate system $(\alpha, \beta, \gamma) \in \mathbb{R}^3$, obtained by normalizing the areas of the subtriangles:

$$\alpha(\boldsymbol{p}) = \frac{f_0(\boldsymbol{p})}{2A}, \quad \beta(\boldsymbol{p}) = \frac{f_1(\boldsymbol{p})}{2A}, \quad \gamma(\boldsymbol{p}) = \frac{f_2(\boldsymbol{p})}{2A}$$

, which fulfill the constraint $\alpha(\boldsymbol{p}) + \beta(\boldsymbol{p}) + \gamma(\boldsymbol{p}) = 1$ for all $\boldsymbol{p}$. This follows by interpreting the edge functions as *signed* areas of the subtriangles induced by $\boldsymbol{p}$.

Considering Figure 8, observe that the vertices are mapped to canonical barycentric coordinates:

$$\boldsymbol{v_0} \mapsto (1, 0, 0), \quad \boldsymbol{v_1} \mapsto (0, 1, 0), \quad \boldsymbol{v_2} \mapsto (0, 0, 1)$$

In addition, observe by Figure 8 that for all $\boldsymbol{p}$ in the interior of triangle $\triangle v_0v_1v_2$, $0 \leq \alpha(\boldsymbol{p}) \leq 1$, $0 \leq \beta(\boldsymbol{p}) \leq 1$ and $0 \leq \gamma(\boldsymbol{p}) \leq 1$ holds.

Note that f_0, f_1, and f_2 in Equation 1 can be replaced with α, β, and γ, since they satisfy the edge function definition down by a scale factor from the original edge functions in Equation 5. However, this form is more sensitive to precision errors due to its narrower range, with the effect depending on the number representation. With floating-point numbers, the error might be tolerable, but with fixed-point numbers it is better to test cell occupancy using f_0, f_1, f_2 and only rescale to floating point when shading.

The next section explains why fixed-point numbers are preferred for enforcing consistent triangle fill conventions in Pineda rasterization.

2.1.5 Fill convention

When two triangles share an edge and are shaded with different colors, a sample located exactly on the shared edge can lead to conflicts if the order of evaluation is not defined, and result in noticeable flickering among the pixels along the shared edge. To resolve such ambiguities, graphics libraries adopt a (triangle) fill convention, typically either top-left or bottom-right [31].

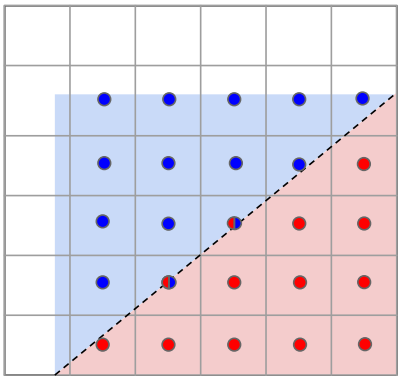


Figure 9: A shared edge resulting in possible pixel color conflicts.

This work uses the top-left convention, meaning that a triangle fills pixels along its top and left edges but excludes pixels along its bottom and right edges, ensuring that shared edges are consistently assigned to a single triangle.

The convention can be held by subtracting a very tiny ε value (typically the smallest positive value representable) from the linear edge functions, when a edge is a bottom or right edge. This corresponds to shifting the 3D planes representing the edge functions very slightly down in the z-axis.

Let an edge vector be $\boldsymbol{e} = \boldsymbol{v_i v_j} : i \neq j \wedge i, j \in \{0, 1, 2\}$. Assuming counterclockwise winding and y increasing upwards on the screen, the edge is a bottom or right edge if:

$$\neg\Big(\big(\underbrace{\underbrace{|e_y| \leq \varepsilon}_{\text{is horizontal edge}} \wedge \underbrace{e_x < 0}_{\text{edge points left}}}_{\text{is top edge}}\big) \vee \underbrace{e_y < 0}_{\text{edge points down and is left edge}}\Big) \tag{6}$$

, where $|e_y| \leq \varepsilon$ is meant to test $e_y = 0$ but accounts for precision error.

However, there is a problem with the usual floating-point representation: the difference between one floating point number and another is not constant; it depends on the interval of floating point numbers being worked on.

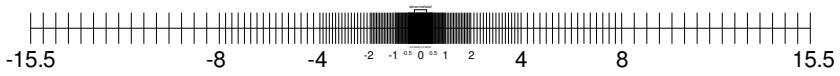


Figure 10: Non-uniform distribution of floating point numbers [36]

Thus, for uniformity and consistency, fixed point numbers lends itself better for above purpose with a clearly defined smallest positive number ε.

To reiterate, the edge functions are to be computed with fixed point numbers instead of floating point numbers, and can be modified slightly should the corresponding edge be a bottom or right edge by subtracting a tiny ε, the smallest positive number representable by the fixed point implementation.

There are a caveat: this modification should only take place when a cell occupancy test is carried out, but not when the edge functions are normalized to define barycentric coordinates for shading purposes. Additionally, for a strictly gapless rasterizer, which is outside the scope of this paper, single-pixel or very thin triangles ought to be handled with more consideration.

2.1.6 Visibility buffer

This section describes the idea of a visibility buffer, which allows shading to be deferred until after rasterization. It does not describe how it is implemented or used in practice (i.e, using 64-bit atomics together with depth values [37]), which will be covered later.

By allowing the rasterizer to store per-pixel triangle identifiers or tags (e.g., indices) during rasterization in a simple *visibility buffer*, unnecessary shading work can be avoided once the shading pass is performed. In particular, triangles that are later hidden by other geometry do not need to be shaded in advance [5].

More advanced schemes combine a mesh index with a triangle index, or use duplicated indices followed by a binary search to recover the correct triangle. For simplicity and given the expected number of triangles in this work, these approaches are not used [5, 37].

With the idea of a visibility buffer, the *graphics pipeline* can be operationally divided into three stages: 1) the geometry setup stage, which transforms 3D triangle world coordinates into 2D screen-space coordinates, alongside depth d and inverse-z $1/z$ values; 2) the rasterization stage, which rasterizes the triangles and produces both a visibility buffer and a depth buffer; and 3) the shading stage, which takes the visibility buffer as input and produces a color buffer that is ultimately rendered to the screen. If multiple passes are possibly performed, the rasterization stage overwrites the previous visibility and depth buffers, while the shading stage overwrites the previous color buffer.

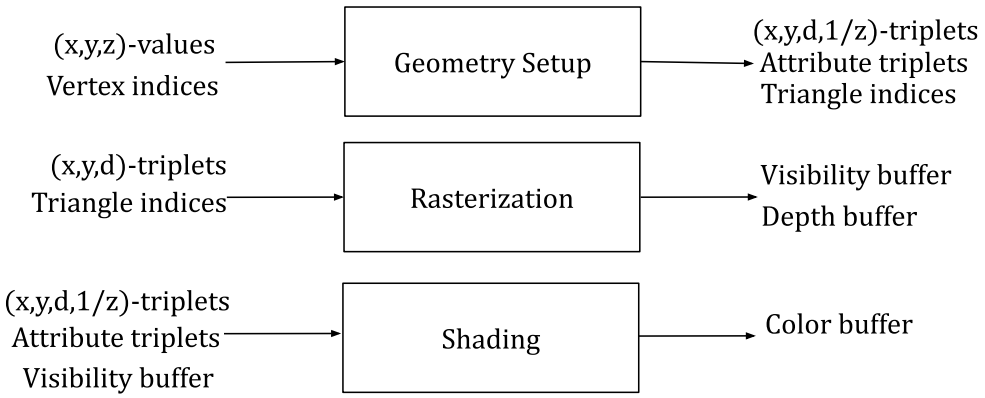


Figure 11: Operational description of a *single-pass* 3D rendering pipeline

2.1.7 Custom shader functions

Appendix B.1 describes how 3D rendering is performed through rasterization, why a depth buffer is required and, thereby, why d must be stored per fragment. It also explains what the camera frustum planes are and defines homogeneous coordinates, which unify the notions of $1/z$ division, translation through matrix multiplication, and clipping with the help of these coordinates.

Appendix B.2 describes perspective-correct interpolation and why $1/z$ must be stored per fragment and explains how interpolation is performed.

The model-view-perspective (MVP) matrix applied on vertices and vertex attributes are implicity treated as fixed in the Appendix, reflecting the pre-OpenGL 2.0 fixed-function pipeline before programmable shaders were introduced. The updated pipeline instead follows a programmable model.

In this model, vertices are first processed by a user-defined vertex ”shader”, which applies user-provided uniform constants to transform input vertices into homogeneous 3D coordinates while carrying associated attributes. Vertex indices are then used to reuse shared vertices, avoiding redundant shader execution, and triangles are assembled before clipping, possibly producing a different number of vertices.

The resulting primitives proceed to rasterization, where attributes, depth, and pixel positions are perspective-correctly interpolated and passed to a fragment shader. This shader combines interpolated attributes with uniform constants to produce the final output, typically an RGBA color.

Both clipping and shading rely on interpolation of user-defined attributes, minimally requiring support for addition between attributes and scalar multiplication with the attribute. While OpenGL supported this by restricting the now-deprecated *varyings* to a pre-defined set of types (i.e integer and floating types), this work instead allows user-defined attribute types with explicitly provided operations, generalizing the approach.

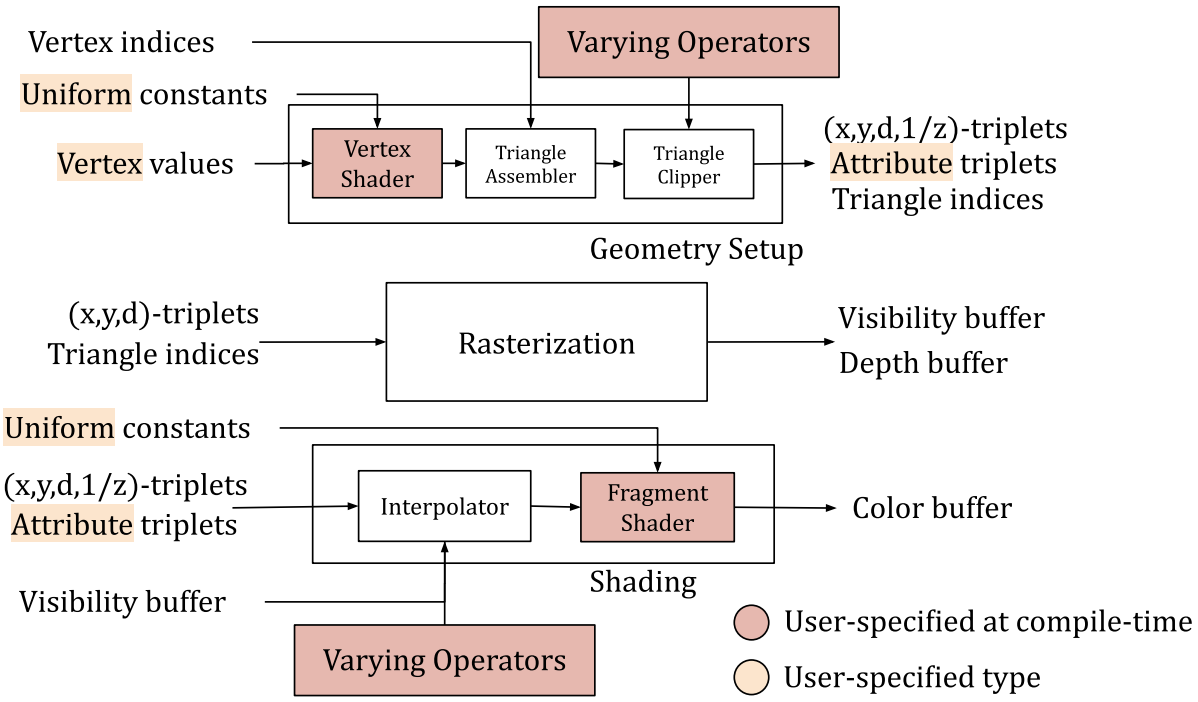


Figure 12: A basic programmable 3D pipeline (updated Figure 11)

2.2 Data-parallel rendering

The previous sections have described the operation of the individual stages in a programmable 3D rendering pipeline. The next step is to examine the forms of parallelism inherent in rendering and how they can be expressed efficiently in Futhark. Because different pipeline stages naturally expose different forms of parallelism, understanding how to represent and transform between them is essential for implementing both the immediate-mode and tile-based renderers explored in this work.

2.2.1 Object-parallelism vs image-parallelism

Data-level parallelism is about performing the same operation to multiple data elements in parallel.

As Holten-Lund [20] states, there are two primary forms of data-parallelism to exploit in rendering:

- **Object-parallelism** operating on triangle primitives in parallel.
- **Image-parallelism** operating on screen pixels in parallel.

Throughout the rendering pipeline, the natural unit of parallelism transitions from primitives to pixels. As established in Section 2.1, the relationship between these domains is irregular, as each primitive may generate an arbitrary number of pixel fragments, while each pixel may be covered by multiple primitives. This irregularity makes the transition between object-parallel and image-parallel computation a central challenge in rasterization. The present discussion focuses on high-level data-parallel transformations required to express this in Futhark, while the concrete mapping of parallel work to processing elements and the associated scheduling decisions are left to the Futhark compiler and runtime system.

2.2.2 Segmented operations in Futhark

Futhark is a statically-typed parallel array programming language designed to generate code for backends such as CUDA, OpenCL, or multi-threaded C. It provides a small set of language primitives optimized for high-performance data-parallel computation [16]. These are based on Blelloch [4]'s parallel primitives, including `map`, `reduce`, `filter`, and `scan`. These primitives mirror sequential operations but run in parallel over array elements.

Futhark supports *regular* segmented reductions, which allow `reduce` and `scan` primitives to be used inside a `map` primitive among other things [28].

Some parallel problems, including rasterization, require operating on segmented data (think list-of-lists), where segment lengths are irregular. In Futhark, such structures can be represented in flattened form using two arrays: a data and flag array of same size, where a true value indicates the start of each segment.

$$\begin{aligned} \boldsymbol{D} &= \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix} \\ \boldsymbol{F} &= \begin{bmatrix} \mathrm{T} & \mathrm{F} & \mathrm{F} & \mathrm{T} & \mathrm{F} & \mathrm{T} & \mathrm{T} \end{bmatrix} \\ \boldsymbol{a} &= \begin{bmatrix} \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 4 & 5 \end{bmatrix} \begin{bmatrix} 6 \end{bmatrix} \begin{bmatrix} 7 \end{bmatrix} \end{bmatrix} \end{aligned}$$

Figure 13: The data-array $\boldsymbol{D}$, flag array $\boldsymbol{F}$, and the corresponding segmented data $\boldsymbol{a}$ consisting of segments of size 3, 2, 1 and 1.

Futhark supports `segmented_reduce` and `segmented_scan` functions for such data-structures through it's standard `segmented` library [7]. For the example above, the futhark interpreter returns:

```
> let D = [1, 2, 3, 4, 5, 6, 7]
> let F = [true, false, false, true, false, true, true]
> segmented_reduce (+) 0 F D
[6, 9, 6, 7]
> segmented_scan (+) 0 F D
[1, 3, 6, 4, 9, 6, 7]
```

Here the `(+)` operator alongside the identity element `0` is passed to perform summation between segment-local elements. `segmented_reduce` performs segment-wise summation and returns an array of size equal to the number of true elements in the flag array. `segmented_scan` performs segment-wise accumulative summation similar to `scan` and returns the same number of elements as the input array.

The `segmented` standard library additionally provides the `expand` function, which is the key design pattern to perform flattening-by-expansion technique to handle the irregular problem of rasterization [17]. It has the following type:

```
val expand 'a 'b: (sz: a -> i64) -> (get: a -> i64 -> b)
                  -> []a -> []b
```

The `expand` function takes a segment size function `sz`, a segment indexing function `get`, and an array of input values. It returns an array whose length is the sum of the segment sizes computed by `sz`, with each element generated by `get` given the input value defining the segment and the element's local index within that segment.

For an example, suppose we wish to recreate the array in Figure 13 using `expand` from the segment sizes. Using the futhark interpreter:

```
> let s: []i64 = [3, 2, 1, 1]
> let o = exscan (+) 0 s
> o
[0, 3, 5, 6]
> expand (\i -> s[i]) (\i j -> o[i] + j + 1) (indices s)
[1, 2, 3, 4, 5, 6, 7]
```

Here `exscan` is a library primitive to perform `scan` excluding the first element, giving a prefix sum of the sizes. Here `expand` is passed the `indices` of `s` for the array segment descriptors (i.e. data describing each irregular segment). And the anonymous function `(ı-> s[i])` is passed as `sz`, taking upon a segment descriptor element `i` to produce the corresponding size; here it's just `s[i]`. Once the total size is precomputed and the resulting flat-array is to be filled, it passes the segment descriptor element as the first parameter to `get`, here defined as the anonymous function `(ıj -> o[i] + j + 1)`, and the segment local index `j` as the second parameter. The produced element here is the one-indexed segment-local index `j + 1` plus the offset `o[i]` to produce the original array.

If desired, one could produce a tuple with the latter element storing the result of the predicate `j == 0` and `unpack` the resulting array of tuples to produce the original data and flag arrays. Here `|>` is the functional pipe operator s.t. it transforms `g (f a)` to `f a |> g` given unary functions `f` and `g`.

```
> expand (\i -> s[i]) (\i j -> (o[i] + j + 1, j == 0)) (indices s)
  |> unzip
([1, 2, 3, 4, 5, 6, 7],
 [true, false, false, true, false, true, true])
```

The final important Futhark primitives for the purposes of rasterization is `scatter` and `reduce_by_index`, the latter allowing efficient, generalized histogram-like operations with atomics and subhistogramming [19].

2.2.3 Generalized histogram in Futhark

Both `scatter` and `reduce_by_index` provides the means to update arrays with given indices and corresponding values (atomically). Both implicitly ignore out-of-bound indices. `scatter` has following type:

```
val scatter [k] [n] 't: (dest: [k]t) -> (is: [n]i64) -> (vs: [n]t)
                        -> [k]t
```

The `scatter` function takes a destination array `dest` and element indices `is` and values `vs`, and updates the elements at corresponding indices in `dest`. On the interpreter, for the following example, the out of bounds index `-1` is ignored, `10` is written to index `0` and `11` is written to index `1`.

```
> let a = [1, 2, 3, 4, 5]
> scatter a [-1, 0, 1] [9, 10, 11]
[10, 11, 3, 4, 5]
```

`reduce_by_index` takes upon two extra parameters, an operator and it's corresponding identity element. It has the following type:

```
val reduce_by_index [k] [n] 't: (dest: [k]t)
                    -> (op: t -> t -> t) -> (ne: t)
                    -> (is: [n]i64) -> (vs: [n]t) -> [k]t
```

Before updating an element `dest[i]` at index `i`, it performs `op` between the old and new value as following pseudocode (taken from [10]). It is a generalized pattern from the *histogram problem* described in [19].

```
dest = replicate k ne
for j < length is:
  i = is[j]
  a = as[j]
  if i >= 0 && i < k:
    dest[i] = op(dest[i], a)
```

Aside from discarding out-of-bounds indices, which typically does not occur since primitives are clipped to remain within the screen bounds, this pattern corresponds directly to the desired formulation of depth buffer resolution (described in the implementation section). Here, `op` is mapped to an atomic operation when possible, or otherwise implemented by the Futhark compiler using alternative strategies that preserve atomicity.

2.2.4 Sorting classification

Molnar [32] created an useful classification of parallel renderers which produces an image through a *single* forward-pass through a graphics pipeline, based on where they perform a work-redistribution step, by dividing them into sort-first, sort-middle and sort-last. This work-redistribution can be thought of as a *spatial sorting step*, as it is about assigning triangles to screen regions, thereby reordering the triangles with respect to that screen region. Holten-Lund [20] elaborates the classification much more extensively.

Sort-last immediate-mode In a sort-last, immediate-mode renderer [2], renderer processors operate independently until a final merging step (the depth resolution step), before the pixels are merged [32]. Then the work grows axiomatically by the total number of pixels plotted, corresponding to the runtime complexity of `reduce_by_index`.

$$O(\texttt{\#primitives} \times \texttt{\#avg pixel pr primitive})$$

Hardware can take advantage of the regularity of the pixel merging operation, and make work scale linearly with the number of renderer processors $O(\texttt{\#screen size} \times \texttt{\#processors})$ [32].

Sort-middle tiled-mode In a sort-middle, tiled renderer [3], a work distribution is performed *once* the screen extents of the geometry is known, but *before* the rasterization step, by assigning geometry to screen region tile "buckets". Each bucket belongs to a processor, and a primitive may fall into several buckets. Then the work scales by the cost to put geometry into a bucket, and the average number of pixels primitives generate pr bucket [32].

$$O(\texttt{\#primitives} \times bucket_d + \texttt{\#avg primitive pr bucket} \times \texttt{\#avg pixel pr primitive})$$

, where the $bucket_d$ is the cost to put a primitive into a bucket, which can be shown to shrink quickly the larger a primitive is relative to tile size [32].

Sort-first retained-mode A sort-first, retained-mode architecture calculates the screen-region bounds *before* the geometry processing, and bins geometry using this data once processed, making use of frame-to-frame coherence to be able to do this [32]. This architecture is largely of theoretical interest for real-time graphics per Molnar [32], but has seen use in GUI systems [45] such as the Chromium engine [21].

3 Design and implementation

The Futhark implementation follows the mathematics described in the Background (and Appendix B) sections, translating them into modular and reusable scalar code. While the design and modularization of a sequential 3D rendering pipeline are interesting in their own right (and are briefly described in Appendix C.1), the primary contribution of this work lies in using Futhark's `expand` function to express the irregular data parallelism inherent in triangle rasterization.

3.1 Data-parallel hierarchal tiling with coverage masks

The rendering pipeline naturally exposes two primary forms of parallelism: object parallelism and screen parallelism, where computation transitions from processing triangle objects to processing pixels in the screen through an irregular mapping. The key challenge is expressing this mapping using the `expand` function in Futhark.

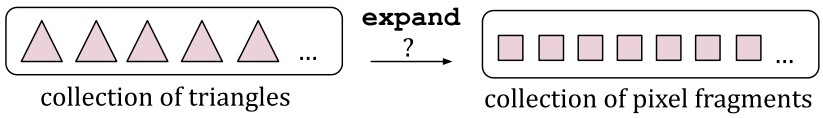


Figure 14: Triangle rasterization problem in Futhark.

Henriksen et al. [17] demonstrated how this can be achieved for an explicit triangle formulation using scanline rasterization, where points within each triangle can be indexed more or less directly after applying `expand` to generate horizontal spans along the triangle edges with a subsequent `expand` to generate the points within each span. This work applies `expand` to an implicit triangle formulation using Pineda rasterization, where points in the triangle bounding box must be first sampled and tested before rasterization.

Given an array of triangles, a naive, *conservative* [24] approach to use `expand` is to compute the bounding box of each triangle and solely use it to determine the number of points to generate, followed by row-major indexing within the box. Points outside the triangle are then ignored, either by explicitly filtering them using `filter` or by mapping them to an out-of-bounds index (e.g. -1) and using `scatter`-like Futhark SOACs, which implicitly filter out-of-bounds indices.

Such an approach both wastes computation on pixels outside the triangle and allocates significantly more memory than necessary, particularly for large triangles. The key idea for avoiding this inefficiency is to combine coverage masks with `expand`. Coverage masks enable a lightweight two-pass approach: the first pass quickly identifies which pixels (or subregions) are at least partially covered by a triangle, while the second pass computes the total number of covered pixels for pre-allocation and reconstructs their indices using efficient bitwise operations. This allows computation and memory allocation to be restricted to covered regions only.

The coverage masks are implemented as fixed-size bitmasks, since boolean arrays are less compact and a fixed-size representation are better suited to efficient GPU execution than a variable-sized bitmask. To support this representation, the screen is partitioned into hierarchical tiles, where each level of the hierarchy constructs a small stack-allocated coverage mask.

3.1.1 Coverage masks

When implementing coverage masks as a word-sized bitmask for supporting `expand`, there are three key operations it must support. In the wider literature, Zhou et al. [48] describes the latter two operations in detail.

1. `set`, which sets the bit at a given index to 1. This is to record the pixel covered and can be implemented with simple bitwise-shift and bitwise-or instruction. Instead the futhark library `set_bit` is used.
2. `rank`, which returns the total number of set bits of a given bitmask. It is implemented efficiently with hardware accelerated `popc` (population count) instruction.
3. `select`, which returns the index of the kth set bit given k.

The final operation can be implemented efficiently using the hardware-accelerated `ctz` (count trailing zeros) instruction, which returns the index of the least significant set bit (LSB). The algorithm for `select(b, k)` repeatedly clears the LSB with `b & (b - 1)` and performs `ctz(b)` once `k` bits are cleared.

```
1 while (k > 0) do
2     b = b & (b - 1)
3     k = k - 1
4 return ctz(b)
```

The above code is unrolled and applied for a 8-bit bitmask. Although, it is rather slow for larger bitmasks, so the bits in the upper and lower half are extracted for larger bitmasks, and a specialized `select` proceeds for the lower half. For the upper bits, the index offset from the lower bits can be efficiently computed with `popc`, and `select` proceeds for the upper half. This can be recursively done for 16-bit, 32-bit, and 64-bit bitmasks:

```
def select_u16 (b: u16) (k: i64) : i64 =
  let lower = u8.u16 b
  let upper = b >> 8 |> u8.u16
  let low_count = u8.popc lower |> i64.i32
  in if k < low_count
     then select_u8 lower k
     else select_u8 upper (k - low_count) + 8
```

However, the use of even larger sizes is sometimes desired, so tuples of integers are used. In Futhark, larger bitmasks can be abstracted recursively with the help of module types and parameterized modules while incurring no additional runtime overhead [9], a pattern seen in the `athas/vector` repository for a static vector implementation [15]. Given two smaller bitmask modules `L` and `R` implementing `num_bits`, `rank` and `select` functions, the `num_bits`, `select` and `rank` functions for the tuple `(L.t, R.t)` is implemented as:

```
def num_bits = L.num_bits + R.num_bits
def rank ((l, r): (L.t, R.t)) = L.rank l + R.rank r
def select ((l, r): (L.t, R.t)) (i: i64) : i64 =
  let l_rank = L.rank l
  in if i < l_rank
     then L.select l i
     else R.select r (i - l_rank) + L.num_bits
```

The recursive structure thus supports logarithmic runtime complexity for `rank` and `select` operations, assuming the 8-bit bitmask base case is $O(1)$.

3.1.2 The algorithm (`expand_masked`)

A generalized design pattern developed during the implementation of the Pineda-style rasterizers is `expand_masked`.

`expand_masked` is an `expand` wrapper using with bitmask `rank` and `select` operations, allowing individual segment elements to be filtered before they are materialized in memory. Intuitively, it enables a "masked" expansion without explicitly performing a `filter` operation to achieve the same effect.

In principle, it can be implemented as follows:

```
def expand_masked 'a 'b
                  (max_sz: a -> i64)
                  (get: a -> i64 -> b)
                  (pred: a -> i64 -> bool)
                  (arr: []a) : []b =
  let get' x i = (get x i, x, i)
  in expand max_sz get' arr
     |> filter (\(_, x, i) -> pred x i)
     |> map (.0)
```

And in practice, `expand_masked` requires support for aforementioned `rank` and `select` bitmask operations, which is enforced with a module type:

```
module type partial_bitmask = {
  type t
  val num_bits : i64
  val empty : t
  val set : t -> i64 -> bool -> t
  val rank : t -> i64
  val select : t -> i64 -> i64
}
```

After which the implementation follows in `expand_masked.fut`:

```
module expand_masked_generic (M: partial_bitmask) = {
  def expand_masked 'a 'b
                    (max_sz: a -> i64)
                    (get: a -> i64 -> b)
                    (pred: a -> i64 -> bool)
                    (arr: []a) : []b =
    let f x =
      let pred' = pred x
      in loop mask = M.empty
         for i < i64.min M.num_bits (max_sz x) do
           M.set mask i (pred' i)
    let get' (x, mask) i = get x (M.select mask i)
    in zip arr (map f arr) |> expand (\(_, mask) -> M.rank mask) get'
}
```

It performs masked flattening-by-expansion as follows:

1. For each source element in parallel, sequentially construct a bitmask by setting bits corresponding to target elements for which `pred` holds, using the bitmask-local index `k`.
2. Use the bitmask `rank` function to count the number of set bits; this serves as the return value of the segment-size function supplied to `expand`.
3. Given a segment-local index `i`, recover the corresponding bitmask-local index `k` using the bitmask `select` function, and pass it to `get` in the indexing function supplied to `expand`.

The above approach is well suited when segment sizes are small and bounded. The `max_sz` parameter additionally allows the user to further bound the sequential loop. Bitmasks are constructed sequentially, as otherwise Futhark would need to exploit nested parallelism over a small number of bits, resulting in lower occupancy and, in practice, poorer performance.

3.1.3 Hierarchal application

Despite `expand_masked` being suited for small bitmasks, this section describes a hierarchical tiling algorithm built on top of it by using `expand_masked` in a nested manner, exploiting triangle convexity to classify tiles from edge-function evaluations at their corner points.

Similar to Kenzel et al. [24], the screen is partitioned hierarchically into bins, coarse tiles, and fine tiles of the following dimensions (which can be adjusted independently of the algorithm):

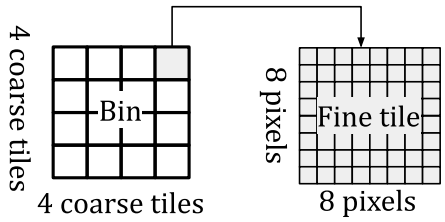


Figure 15: Illustrating the hierarchical tiling scheme

The algorithm performs rasterization in three stages:

The binning stage assigns triangles to screen bins using the aforementioned conservative triangle bounding-box rasterization strategy, but with bins as the rasterization unit instead of pixels. The bounding-box area is passed as the size to `expand`, and a subsequent bounding-box local row-major indexing is done to generate the `(bin_index, triangle_index)` pairs.

The coarse stage performs the two-step approach through `expand_masked`. Given segment-local `coarse_index`, it records overlapping coarse tiles by evaluating edge functions at tile corners in the `pred` function. Passing the number of coarse tiles per bin as `max_sz`, it combines `bin_index` and `coarse_index` into a single `tile_id`, producing (`tile_id`, `triangle_index`) pairs.

The fine stage similarly performs the two-step approach, where it instead records overlapping pixels by evaluating edge functions at the pixel midpoint in the `pred` function. Passing the number of pixels per fine tile as `max_sz`, it recovers the global screen position from `tile_id` and segment-local `pixel_index` to generate `((y,x), <pixel>)` pairs.

The `<pixel>` for the time being can be thought to be a `(color, depth)`-pair. The type of value produced depends on the depth resolution step.

Bins do not use coverage masks because the number of bins is not known at compile time. While a maximum screen size could be assumed or the runtime dispatch mechanism present in `expand_masked.fut` could be used, this was omitted for flexibility. Instead, coarse tile masks already eliminate most irrelevant regions outside triangle bounding boxes. Non-divisible screen dimensions are handled by rounding up the number of bins.

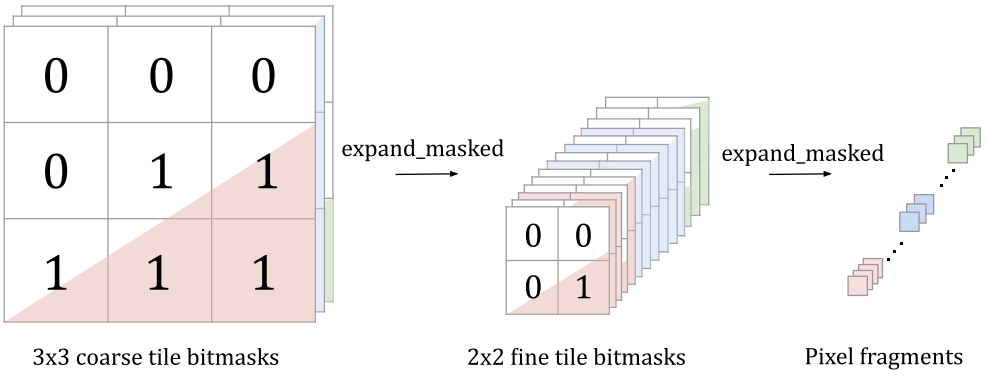


Figure 16: The solution to the rasterization problem (updated Figure 4).

In the concrete code, `triangle_imm_pineda.fut` performs the binning, coarse and fine stage in `bin_rasterize`, `coarse_rasterize` and `fine_rasterize` as described above. With one notable extension being that `bin_rasterize` additionally peforms a `partition` over triangle bounding box area, classifying triangles as small if their bounding box area is smaller than 128 pixels similar to Schütz et al. [37], and proceeds normally to the coarse stage otherwise. If a triangle is small, it's index and bounding box `(tri_index, tri_bbox)` is passed to a specialized small-triangle stage, implemented as a `rasterize_small_triangles` function. This minimizes the binning overhead for small triangles. Note an alternative is to sort and split triangles into more groups by bounding box area like Laine and Karras [26].

Small triangle stage also uses coverage masks through `expand_masked` with slightly different indexing. Given segment-local `bbox_index`, it performs a row-major unflattening based on the bounding box width and records overlapping pixel midpoints in the `pred` function. Passing the bounding box area as `max_sz`, it unflattens `bbox_index` with bounding box width, and recovers the global screen position from `bbox_index` and triangle bounding box minimum position to generate `((y,x), <pixel>)`-pairs.

3.2 Atomic depth resolution

The previous section described handling irregular work in the rasterization problem from Section 2.1. However, overlapping triangles with different colors require per-pixel depth testing to select the front-most pixel (Section B.1.3), which means jointly computing the maximum depth while selecting the corresponding color to produce the final color and depth buffers. While Futhark provides `reduce_by_index` for atomic histogram-like computations in a data-parallel setting [19], the challenge is how to use it effectively to simulate a depth buffer while meeting performance requirements.

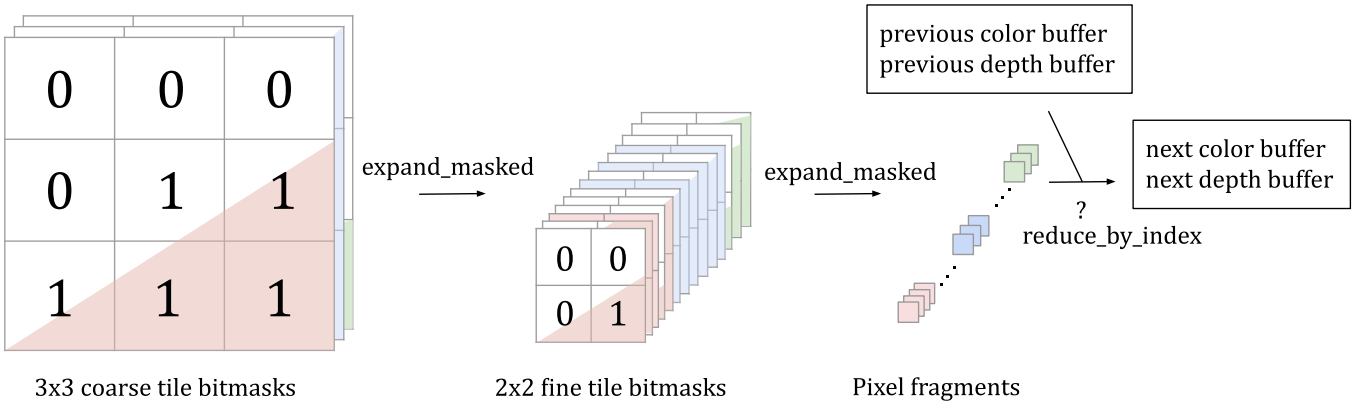


Figure 17: The depth resolution problem (updated Figure 16).

3.2.1 With tuples

Henriksen et al. [17] describes a simple approach in which each pixel is paired with its depth, and `reduce_by_index` is given a binary selection function that chooses the operand with the front-most depth value, where the depth convention held in this work (reversed-z) is to select the larger value.

```
let ne = (argb.black, 0)
let dest: [][](argb.t, f32) = tabulate_2d (const (const ne))
let positions: [](i64, i64) = [ ??? ]
let fragments: [](argb.t, f32) = [ ??? ]
let f lhs rhs = if lhs.1 < rhs.1 then lhs else rhs
let dest' = reduce_by_index_2d dest f ne positions fragments
```

Figure 18: The basic idea behind resolving paired color-depth values

There are two downsides to this approach. The Futhark compiler must perform atomic updates on a composite (depth, colour) pair, which is not directly supported in hardware and typically requires workarounds such as spinlocking or similar mechanisms. In addition, it is not well-suited to shader-style pipelines, where one may prefer deferring shading until after rasterization to avoid invoking a user-defined shader function on pixels that will ultimately be discarded. This shader function may be much more computationally expensive than the rasterization workload itself, so the API should aim to minimize the number of calls, as real graphics APIs usually prefers.

3.2.2 With deferred linear search

In order to defer calling the user-defined shader, the rasterizer must not produce color values immediately, but produce the intermediate input values used by the shader to produce the color. The intermediate value's type is:

```
type fragment 'varying =
  { pos: {x: f32, y: f32}, depth: f32, Z_inv: f32, attr: varying }
```

`fragment` is a parametric type of `varying` to accommodate for user-specified varying attributes. These will elaborated upon later, but the important part is that the rasterizer is passed a user-given `plot` function of type `fragment varying -> target` to produce a user-specified `target`, usually a `argb` color value. `plot` hence represents the fragment "shader", which the rasterizer desires to only call for visible pixels. Furthermore, vertices are initially passed as `fragment`'s, containing screen position and attributes for interpolation.

The main `rasterize` function for the `point.fut` rasterizer is as follows. It illustrates the deferred linear search approach to shade pixels.

```
def rasterize 'varying 'target [n] [h] [w]
            (plot: (fragment varying -> target))
            ((target_buffer, depth_buffer): ([h][w]target, [h][w]f32))
            (frags: [n]fragment varying) : ([h][w]target, [h][w]f32) =
  let ne_target = copy target_buffer[0, 0]
  let (is, frag_values, depth_values) =
    frags
    |> map unpack_fragment
```

```
  |> unzip3
let depth_buffer =
  reduce_by_index_2d (copy depth_buffer) f32.max 0 is depth_values
let (is, target_values) =
  zip is frag_values
  |> map (\((y, x), f) ->
            if (0 <= x && x < w) && (0 <= y && y < h)
            && depth_buffer[y, x] == f.depth
            then ((y, x), plot f)
            else ((-1, -1), ne_target))
  |> unzip2
let target_buffer = scatter_2d (copy target_buffer) is target_values
in (target_buffer, depth_buffer)
```

Here `unpack_fragment` is a simple helper function of type that constructs a triplet of three values: rounded the screen positions, the fragment itself, and the extracted depth value.

Note the use of two passes: a depth-reduction pass using `reduce_by_index` with hardware-accelerated atomic `f32.max`, followed by a second pass that linearly searches for the winning depth using a data-parallel `map`-combinator. In this second pass, `plot` is called *only* for the winning depth, while all other entries are mapped to `-1` in idiomatic Futhark style. Finally, `scatter` is applied to the color values, specified to ignore out-of-bounds indices and otherwise atomically update the target buffer at the corresponding positions.

The same approach is also held for a line rasterizer in `line.fut`, based on Henriksen et al. [17], but extended to shade lines with linear interpolation.

This is more efficient than the previous approach, both in terms of deferring shading and by leveraging a hardware-accelerated `f32.max` atomic operation.

However, while performing the first pass, it must retain the relatively expensive fragment values in memory along with user-specified attributes, which can become significantly large in real production use. Once again, an ideal graphics API should be designed with the worst-case user in mind.

3.2.3 With deferred lookup with visibility buffer

With the rise in support of 64-bit atomics in GPGPU programming, Nanite, according to Schütz et al. [37], demonstrated the effectiveness of visibility buffers in software rendering for depth resolution purposes. The visibility buffer itself could be thought of as a simple buffer for identifying rasterized triangles conceptually as described in Section 2.1.6. But by packing the identifier together with depth values as a 64-bit integer, 64-bit atomics can be efficiently used to both select the depth *and* store the corresponding winning triangle identifier, removing the need to perform a linear search unlike the previous approach.

Non-negative floats sort like integers as per Schatz [36]. This is applicable, as the clipping process ensures triangle fragments have a non-positive depth which lies between 0 and 1. Choosing a suitable sentinel for the indices (e.g. 0 after shifting indices by 1), it is thus possible to pack the 32-bit depth floating point value as the upper 32-bits and the index as the lower 32-bits, making it into a sortable depth key with `u64.min`. The basic idea is:

```
def encode_depth_index d tri_index =
    (u64.u32 (f32.to_bits d) << 32) | (u64.u32 tri_index)
```

Since the sign bit is always 0, following depth conventions, it can be ignored and the index can use 33-bits. Shifting the triangle indices by one, the following functions are derived:

```
def highest_tri_count : i64 = (1 << 33) - 1
def encode_depth d = f32.to_bits d
def encode_depth_index d tri_index =
    (u64.u32 (encode_depth d) << 33)
    | (u64.i64 (tri_index + 1) & ((1 << 33) - 1))
def decode_depth dvis = dvis >> 33 |> u32.u64 |> f32.from_bits
def decode_index dvis = dvis & ((1 << 33) - 1) |> i64.u64 |> (i64.- 1)
def ne_dvis = encode_depth_index 0 (-1)
```

Figure 19: Depth-index packing scheme

Above is used by all the triangle rasterizers, as they generate many pixels. The pixel fragment produced is thus the return value of `encode_depth_index`.

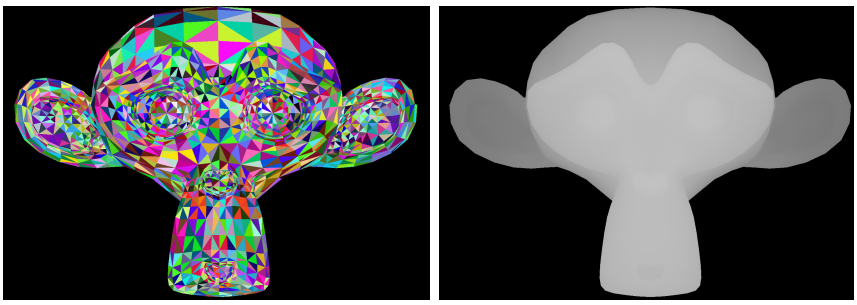


Figure 20: Flat shading demonstrating triangle uniqueness, and depth shading demonstrating depth interpolation through shading.

The idea for using a visibility buffer in Futhark is described with follows:

1. Initialize a packed depth-index buffer by encoding each depth with a sentinel triangle index into a single `u64` per pixel.
2. Generate pixel fragment candidates as `(pixel index, packed depth-index)` pairs using hierarchical tiling and edge functions.
3. Apply `reduce_by_index` with `u64.max` on said buffer to select the winning fragment per pixel.
4. Iterate the packed buffer with `map` and decode each entry; skip sentinel values by producing a sentinel position, otherwise produce position, depth, and color values with edge coefficients and user-specified shader.
5. `scatter` the computed color and depth values at the corresponding positions into the previous color and depth buffers.

A key advantage of this approach is that fragment information is consumed immediately after rasterization. As a result, intermediate data, including user-defined attributes, need not be retained longer in memory than necessary, unlike the previous approach. This lowers memory usage and allows multiple rasterization passes to be executed efficiently while postponing shading done in a single pass. These properties are described in Schütz et al. [37] for more demanding rasterization workloads. Furthermore, since the triangle edge information is computed multiple times, both for rasterization and for shading, it is a good idea to cache the values for reuse and consistency.

A key limitation when using a visibility buffer for depth resolution is that the maximum number of triangles rasterized at a time is significantly lowered compared to the deferred linear search.

3.3 Tile-based rasterization

Using the above techniques for hierarchical tiling and depth resolution with a visibility buffer, Pineda rasterization achieves performance characteristics similar to scanline rasterization implemented with `expand` in Futhark. This raises the question of whether the Pineda rasterization can be used to explore alternate forms of parallel rendering in Futhark.

It is possible to implement a tile-based sort-middle architecture in Futhark, although several limitations and challenges remain, as discussed later. The implementation `triangle_tiled_pineda.fut` employs a purely tile-based rasterization strategy with `rasterize_tiled`, while `triangle_hybrid_pineda.fut` combines immediate-mode and tile-based rasterization. Small triangles are processed using immediate-mode rasterization with `partition` and `rasterize_small_triangles` as described before, whereas larger triangles are handled by `rasterize_tiled`, similar to a design to Schütz et al. [37].

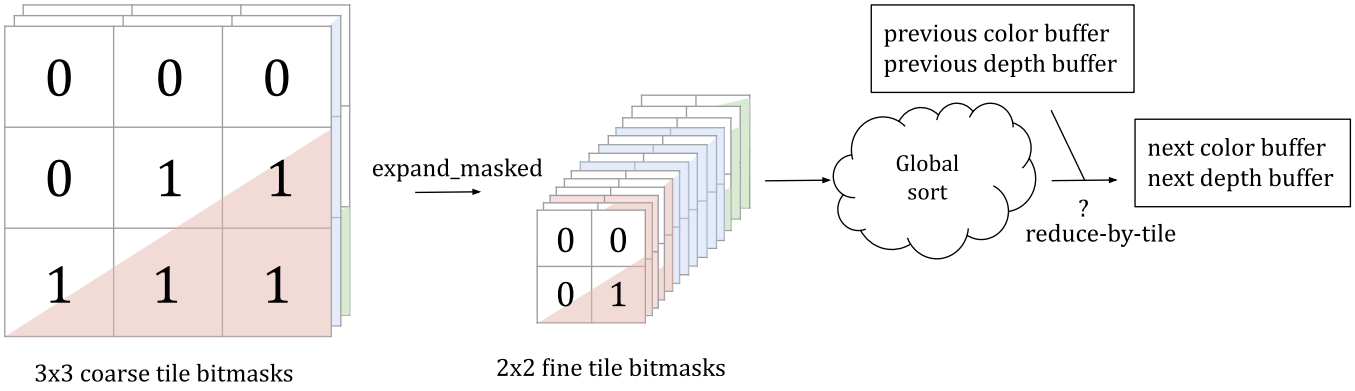


Figure 21: Tile-based rasterization (updated Figure 17).

3.3.1 Spatial sorting step

This is a prerequisite for performing the ”reduce-by-tile” step used in depth resolution for sort-middle tile-based rasterization. A data-parallel interpretation of the work redistribution step described by Molnar [32] is to explicitly sort triangles according to the tiles they are assigned to. `diku-dk/sorts` [6] provides a radix sort implementation, which generally achieves better performance on the GPU than the bucket sorting approach (also provided by `diku/sorts`) employed by Holten-Lund [20]. So that is simply done by sorting after `tile_id` in the `rasterize` function in `triangle_tiled_pineda.fut` once `coarse_rasterize` is performed.

```
let (tile_ids, tri_idxs) =
  bin_rasterize {h,w} tris
  |> coarse_rasterize {h,w} tris
  |> uncurry zip
  |> radix_sort_by_key (.0) (i32.i64 num_bits_to_sort) u32.get_bit
  |> unzip
```

The number of bits to sort is determined by the number of bins and tiles per bin. This can be rather small, depending on the tile size setup and screen size.

Once `(tile_id, tri_idx)` pairs are sorted in a flat array, it allows efficient segmented operations. The pairs are passed to `rasterize_tiled` instead of `fine_rasterize`, where tile flags are derived directly from index adjacency in the sorted data, with care taken at segment boundaries.

```
def calc_segment_flags [n] (is: [n]u32) : [n]bool =
  map (\i -> if i == 0 then true else is[i] != is[i - 1]) (iota n)
let tile_flags = calc_segment_flags tile_ids
```

Using these flags, unique tile IDs and counts are computed via a fusion-friendly segmented reduction: tile IDs are selected using `u32.min`, counts are accumulated as ones, and offsets are obtained via an exclusive scan.

```
let (unique_tile_ids, unique_tile_counts) =
  segmented_reduce (\(i0, s0) (i1, s1) -> (i0 `u32.min` i1, s0 + s1))
                   (u32.highest, 0)
                   tile_flags
                   (zip tile_ids (replicate n 1))
  |> unzip
let unique_tile_offsets = exscan (+) 0 unique_tile_counts
```

3.3.2 Depth resolution with shared memory

The general idea of tile-based rasterization is to exploit locality by loading triangle-tile pairs into workgroups, where they can be processed together using *shared memory*. In Futhark terminology, GPU workgroups are referred to as *intrablocks*, which execute batches of threads that share fast shared memory. The workgroups are exploited by structuring the code to expose a suitable amount of parallelism, enforced explicitly using the `#incremental_flattening(only_intra)` attribute [18]. The following is the main kernel `f` used by the workgroups, exploiting exactly `tile_size` amount of parallelism and using the previously calculated segment information.

```
let f segment_index =
  let tile_id = unique_tile_ids[segment_index]
  let tri_count = unique_tile_counts[segment_index]
  let tri_offset = unique_tile_offsets[segment_index]
  -- recover tile indices and global position ...
  let f pixel_index =
    -- recover pixel position ...
    in if x < w && y < h then dvis_buffer[y, x] else ne_dvis
  let tile_size = fine_size * fine_size
  in loop tile_buf = tabulate tile_size f
     for i < tri_count do
      let tri_index = tri_idxs[tri_offset + i]
      -- retrieve triangle information using tri_index
      let g j =
        -- calculate edge function coefficients ...
        in if -- edge functions are all non-negative
           then -- further calculations ...
                in encode_depth_index depth tri_index
           else ne_dvis
      in map2 u64.max (tabulate tile_size g) tile_buf
```

Figure 22: Triangles processed on a per-tile basis

In above code, note that for the depth resolution step, `map2` is used together with `u64.max`, as the destination indices are already known when processing triangles on a per-tile basis, instead of a `reduce_by_index`. Nonetheless, the Futhark compiler probably uses atomic operations to perform the update.

The kernel is to be launched in the following manner, once the visibility buffer `dvis_buffer` is initialized:

```
let dvis_buf =
  let xs =
    #[incremental_flattening(only_intra)]
    tabulate k f |> flatten
  let is = tabulate (k * (fine_size * fine_size)) g
  in scatter (copy (flatten dvis_buffer)) is xs
```

Figure 23: Launching the kernel `f` as independent workgroups

Here, `g` computes the global buffer index corresponding to a tile index traversed in row-major order. Using above makes Futhark launch the maximum number of workgroups available on the GPU[2].

The kernel `f` processes `(tile_id, tri_idx)`-pairs and sequentializes said processing, so span grows with the maximum number of triangle-tile overlaps. This changes the rasterizer's runtime complexity, so it behaves more like a sort-middle rasterizer.

The above approach is used for the hybrid rasterizer, since it accesses the global visibility buffer in a subsequent pass. A more bandwidth-friendly and hardware-representative implementation also performs *shading* using fast on-chip memory [22], i.e. shared memory within the kernel `f`. This can be achieved by `scatter`'ing to a paired color-depth buffer rather than the visibility buffer shown in Figure 23, and using the per-tile visibility buffer in `f` to perform shading on *per-tile* color and depth buffers within `f` before writing the final results back to the global color and depth buffers. Nonetheless, the asymptotic runtime behavior is primarily defined by the depth resolution step rather than the deferred shading step.

[2] According to Troels Henriksen, this is the current default behavior in Futhark.

4 Discussion and evaluation

The performance claims made in previous section is validated in this section, while the results are discussed.

It is worth noting the results here only applicable after simplifying assumptions are taken compared to more production-grade software rasterizers. Neither multi-sampling is implemented, dramatically increasing the cost of per-pixel work as Molnar [32] states, nor is alpha-blending supported, simplifying the implementation unlike Kenzel et al. [24] and Laine and Karras [26].

As the benchmarks will show, the tiled rasterizer performs rather poorly for large number of triangle-tile overlaps. This can be explained by suboptimal load balancing due to the lack of a more flexible, dynamic balancing like that of Laine and Pantaleoni [27]. This can also be explained by the lack of classical tile-based rasterizer optimisations such Hiearchical Z-culling and lack of use of clustering algorithms, though the latter improves performance for immediate-mode rasterizers as well. The former is not implemented, as it would also need to be maintained for point and line rasterizers, complicating the implementation effort as rasterizers are cleanly seperated otherwise.

Hierarchical Z-culling Hierarchical Z-culling is about maintaining separate per-tile minimum depth values (assuming reversed-z here), thereby allowing early rejection of fully occluded triangles. The idea is to reject tile-triangle pairs early in the hierarchy, once a tiled is completely rasterized by the triangles in front by updating a minimum depth value, dramatically cutting down the number of pixels to plot. Greene [14] elaborates this further.

Triangle clustering algorithms Holten-Lund [20], while implementing a sort-middle renderer, describe a mesh partitioning algorithm where partitions (or clusters) can be culled as a unit, reducing the number of individual depth tests required. Nanite [38] describe a hierarchical clustering scheme with per-camera level-of-detail (LOD) selection, which merges sub-pixel triangles together, and thereby reducing the number of rendered triangles further.

There is no dedicated support for texture sampling, although it is minimally doable through shaders once textures are loaded into Futhark. Rasterization is not done in pixel-quads, allowing LOD calculation and mipmapping. In tile-based rendering, dedicated support for a texture is typically available and done by loading it as tiles in fast, shared-memory while shading [23].

4.1 Experimental setup

The benchmarks are done a relatively old, discrete laptop GPU. The models tested have relatively few triangles compared to modern rasterization workloads with millions of triangles. Nonetheless, the expectation is better hardware, alongside the parallel rendering architecture implemented as software, scales with larger number of triangles. The screen size (i.e. the color and depth buffer size used) matters as well for the timings, as it affects number of pixels plotted.

GPU	VRAM	Compute capability	Emulated screen size
`Nvidia Quadro M2000M`	4 GB	`5.0`	1024×1024

4.2 Benchmarks

The tables note the average timing and rounds values to single-decimal milliseconds for presentation purposes. The variance in timing across several measurements isn't usually high, so is therefore not noted.

Effect of hardware

First, the expectation for scalability across more powerful machines is validated through using run time as a proxy. Under the writing of this thesis, the author switched from a older laptop with a AMD GPU to a laptop with above GPU. AMD naturally runs fastest on the Futhark HIP backend, while NVIDIA naturally runs fastest on the Futhark CUDA backend. Comparing the performance of the two, alongside multicore C backend as control gives:

Hardware \ Model	Bunny	Monkey	Penger	Dragon	Lucy	Armadillo
(number of triangles)	4968	3936	1144	249881	99970	99976
`i7-6820HQ` (multicore)	30.5 ms	36.5 ms	20.5 ms	71.8 ms	26.2 ms	35.9 ms
`Vega 8` (HIP)	10.9 ms	15.7 ms	9.7 ms	37.9 ms	16.1 ms	20.8 ms
`M2000M` (CUDA)	4.6 ms	6.0 ms	4.3 ms	13.6 ms	5.9 ms	6.7 ms

Table 1: Single-frame depth shading for `triangle_imm_pineda.fut`.

Effect of rendering architecture

Molnar [32] describes scaling laws for different rendering architectures. In order to observe these scaling laws, the two following parameters are varied: the proportion of triangles rendered for each model and their associated run time and memory usage, and the screen resolution used to render to.

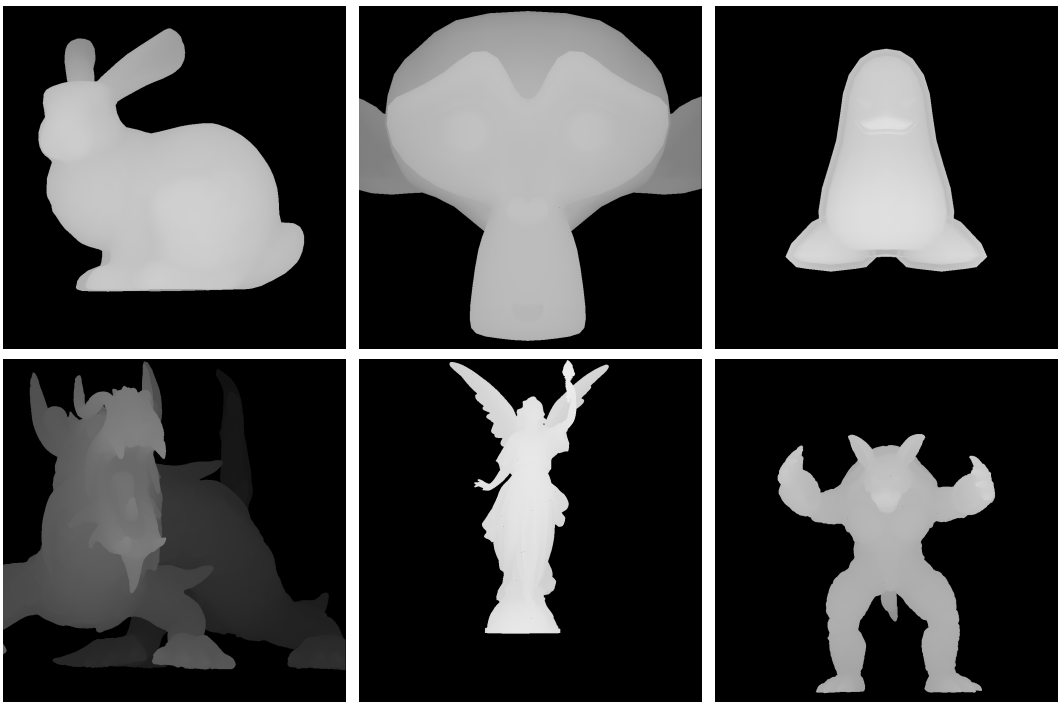


Figure 24: The rendered frame used for benchmarking depth-shaded models

The (Stanford) Bunny, (Suzanne) Monkey, and Penger models contain relatively few large triangles. The Dragon, Lucy, and Armadillo models contain many smaller triangles in much larger numbers. These models are used to represent different kind of models present in usual rasterization workloads.

The proportion of triangles rendered provides a parameter for varying the number of triangles within a model with a fixed triangle size distribution. The expectation is that immediate-mode rasterizers perform well when rasterizing many small triangles, but use a disproportionate amount of memory compared to tile-based rasterizers. A tile-based rasterizer performs best when the average number of triangle-tile overlaps is low and binning cost is small.

The peak memory usage is recorded from `futhark profile` data. It is rounded to the nearest MB with single-decimal precision for presentation purposes. The proportions used are 25%, 50% and 100% of the model triangles.

Rasterizer \ Model	Bunny (5k)	Monkey (3.9k)	Penger (1.1k)	Dragon (250k)	Lucy (100k)	Armadillo (100k)
(25%)	3.7 ms 18.9 MB	3.7 ms 22.0 MB	2.8 ms 17.4 MB	5.6 ms 37.2 MB	3.5 ms 21.2 MB	3.9 ms 22.3 MB
(50%)	4.3 ms 22.1 MB	3.9 ms 26.0 MB	2.8 ms 19.5 MB	7.6 ms 47.1 MB	4.3 ms 25.9 MB	4.1 ms 27.1 MB
immediate scanline (100%)	4.9 ms 24.8 MB	5.1 ms 33.0 MB	4.3 ms 25.7 MB	11.0 ms 77.3 MB	4.9 ms 34.1 MB	5.8 ms 35.7 MB
(25%)	4.2 ms 18.8 MB	4.4 ms 22.0 MB	3.1 ms 17.5 MB	6.7 ms 33.7 MB	3.9 ms 21.2 MB	4.3 ms 21.3 MB
(50%)	5.4 ms 21.8 MB	4.6 ms 25.7 MB	3.8 ms 19.5 MB	10.0 ms 40.9 MB	4.6 ms 23.9 MB	5.7 ms 25.0 MB
immediate pineda (100%)	6.4 ms 24.4 MB	6.0 ms 32.5 MB	4.4 ms 25.8 MB	13.9 ms 63.7 MB	5.6 ms 30.7 MB	7.5 ms 32.4 MB
(25%)	3.5 ms 9.2 MB	3.8 ms 9.3 MB	2.9 ms 9.0 MB	10.9 ms 22.2 MB	4.7 ms 13.1 MB	5.6 ms 13.4 MB
(50%)	4.1 ms 9.3 MB	4.6 ms 9.3 MB	3.3 ms 9.1 MB	17.4 ms 28.3 MB	7.8 ms 14.0 MB	8.3 ms 14.1 MB
tiled pineda (100%)	4.6 ms 9.4 MB	6.1 ms 9.5 MB	3.7 ms 9.1 MB	29.3 ms 52.4 MB	12.2 ms 21.2 MB	13.1 ms 21.5 MB
(25%)	4.6 ms 17.7 MB	4.8 ms 17.7 MB	3.4 ms 17.5 MB	7.4 ms 33.7 MB	4.6 ms 21.2 MB	4.7 ms 21.2 MB
(50%)	4.9 ms 17.8 MB	5.9 ms 17.8 MB	3.8 ms 17.5 MB	10.3 ms 40.9 MB	6.1 ms 23.9 MB	5.8 ms 25.0 MB
hybrid pineda (100%)	5.4 ms 17.9 MB	6.4 ms 20.0 MB	4.8 ms 17.6 MB	15.5 ms 63.6 MB	7.6 ms 30.6 MB	8.6 ms 32.3 MB

Table 2: Effect of varying model triangle proportion

Rasterizer \ Model	Bunny (5k)	Monkey (3.9k)	Penger (1.1k)	Dragon (250k)	Lucy (100k)	Armadillo (100k)
immediate scanline	3.7 ms	4.9 ms	3.2 ms	11.0 ms	5.2 ms	5.8 ms
(2048×2048 render)	10.2 ms	13.8 ms	9.2 ms	23.1 ms	11.0 ms	12.5 ms
(4096×4096 render)	34.3 ms	47.0 ms	31.0 ms	66.7 ms	30.0 ms	35.3 ms
(8192×8192 render)	126.0 ms	174.1 ms	114.2 ms	219.3 ms	97.5 ms	112.7 ms
immediate pineda	4.5 ms	5.7 ms	4.1 ms	13.2 ms	5.8 ms	6.8 ms
(2048×2048 render)	12.0 ms	17.1 ms	11.5 ms	29.2 ms	12.2 ms	14.0 ms
(4096×4096 render)	41.4 ms	57.2 ms	39.1 ms	82.8 ms	32.2 ms	37.4 ms
(8192×8192 render)	156.0 ms	222.8 ms	147.4 ms	280.9 ms	107.7 ms	128.8 ms
tiled pineda	4.3 ms	5.4 ms	3.4 ms	29.4 ms	12.0 ms	12.8 ms
(2048×2048 render)	10.3 ms	13.5 ms	8.0 ms	43.7 ms	17.2 ms	19.1 ms
(4096×4096 render)	32.3 ms	44.1 ms	25.6 ms	88.6 ms	34.2 ms	40.3 ms
(8192×8192 render)	114.1 ms	158.9 ms	94.1 ms	237.3 ms	94.0 ms	113.1 ms
hybrid pineda	5.1 ms	6.0 ms	3.9 ms	15.3 ms	7.2 ms	7.9 ms
(2048×2048 render)	11.7 ms	15.7 ms	10.0 ms	35.2 ms	13.1 ms	15.5 ms
(4096×4096 render)	38.0 ms	50.6 ms	31.1 ms	96.6 ms	37.2 ms	44.5 ms
(8192×8192 render)	136.0 ms	183.4 ms	113.2 ms	267.8 ms	110.8 ms	130.3 ms

Table 3: Effect of screen size

Note for the 8196×8196 render for the immediate Pineda rasterizer, a bin size of 64×64 is used together with 8×8 to make the index fit into a `u16`.

Three interesting observations stand out. First, the immediate-mode, scanline rasterizer, based on Henriksen et al. [17], generally performs best. Second, the execution times of the hybrid rasterizer generally fall between those of the immediate-mode and tiled-mode Pineda rasterizers. In cases where one of the two approaches performs particularly poorly, however, the hybrid rasterizer attempts to switch to the more efficient method, thereby avoiding the worst-case behavior of either approach. Third, the tile-based rasterizer use significantly less memory to rasterize an increasing number of triangles.

The first observation can be explained by the simplicity of the implementation and its more effective parallelization with Futhark SOACs. This suggests that the `expand_masked` implementation is not as scalable as desired and could be improved. As expected, both immediate-mode rasterizers scale with the number of pixels rasterized and are ultimately dominated by the asymptotic cost of `reduce_by_index`. Consequently, increasing the screen resolution increases memory traffic, eventually bottlenecking the immediate-mode rasterizers, making them slower than the tile-based rasterizer.

The tile-based rasterizer instead scales with the average number of triangle-tile overlaps and the cost of binning triangles. While binning introduces overhead, it can improve locality and reduces global memory traffic. As resolution increases, the relative cost of binning decreases, making the tile-based rasterizer increasingly competitive. At 8192×8192, it can even render the 100k-triangle models just as fast as the immediate-mode scanline rasterizer (likely with smaller peak memory usage).

Nonetheless, the implemented sort-middle, tile-based architecture is likely less competitive than those of Laine and Karras [26] and Kenzel et al. [24]. This may partly be due to the lack of low-level CUDA optimizations, though the Dragon model's memory usage suggests another factor. Both prior approaches extensively use shared memory throughout the pipeline, including both for the binning/tiling and depth resolution step, processing triangles in batches to maintain small working sets and exploit locality. In contrast, the present implementation allows much larger intermediate structures in global memory, reducing locality benefits and increasing memory traffic, which limits some advantages of tile-based rasterization, particularly its low-bandwidth efficiency making it useful for mobile GPUs [23]. Another interesting optimization employed is static indexing patterns [25] to improve load balancing, especially for when the number of rasterizer processors (workgroups) is small.

The second observation is the desired outcome following careful tuning of the hybrid rasterizer. Inspired by Schütz et al. [37], the hybrid rasterizer addresses the scaling limitations of both immediate-mode and tiled rasterizers. According to Molnar [32], large triangles are generally cheaper to bin, whereas small triangles incur higher binning costs. Separating triangles into small and non-small categories therefore allows the rasterizer to better balance the trade-offs of both approaches. Additionally, it inherits the locality and memory-efficiency benefits of tile-based rasterization for large triangles.

Note, one is not restricted to a purely Pineda-style rasterizer, unlike what might seem to be implicitly assumed here. Since the immediate-mode scanline rasterizer performs well on models with many small triangles, it could be combined with a tiled Pineda rasterizer for larger triangles. Holten-Lund [20] demonstrates an approach of tile-based rasterization, where edge functions are used for binning, and scanline rasterization is used *after* triangles are clipped against the tile borders. Since clipping is performed in 2D, efficient *guard-band clipping* [11] can be implemented by allocating a guard band much larger than the screen and clipping only when a triangle extends beyond both the screen and the guard-band boundaries.

Effect of rasterizer parameters

This section tunes the exposed rasterizer parameters.

The first parameter common to Pineda-style rasterizers is the tile size. To isolate its effect, tile sizes are benchmarked using a custom immediate-mode Pineda rasterizer *without* a small-triangle stage, on the depth-shading of the Dragon model. Let c = bin area/fine tile area denote the coarse tile area. Implementation constraints require both c and the fine tile area to be powers of two, with $8 \leq c \leq 256$, as the implementation only supports aligned 8-bit masks and sequential bitmasks beyond 256 bits become too slow.

bin \ fine	4x4	8x8	16x16
16x16	14.4 ms 85.6 MB		
32x32	16.0 ms 91.2 MB	18.3 ms 79.0 MB	
64x64	21.4 ms 98.4 MB	20.7 ms 85.4 MB	37.7 ms 84.9 MB
128x128		26.9 ms 92.2 MB	45.6 ms 93.4 MB

Table 4: Effect of bin tile area

The results suggest, smaller bitmask combinations perform faster. However, the binning becomes more memory-expensive as well, as triangles overlap more bins thereby materializing more values. So the 32×32 bin and 8×8 fine size is used as the default option for the immediate-mode Pineda rasterizer.

Then, suppose small triangle optimization is used. At what threshold, should triangle bounding box area be used to classify triangles as small and non-small triangles. Schütz et al. [37] suggests the empirical boundary lies at 128 pixels. Observing the Dragon model in the following table, 128 pixels seems to be a reasonable threshold.

Threshold \ Model	Bunny (5k)	Monkey (3.9k)	Penger (1.1k)	Dragon (250k)	Lucy (100k)	Armadillo (100k)
32 pixels	6.2 ms	8.2 ms	5.4 ms	26.0 ms	6.5 ms	8.6 ms
64 pixels	6.3 ms	8.3 ms	5.4 ms	15.1 ms	5.7 ms	6.7 ms
128 pixels	6.3 ms	8.2 ms	5.4 ms	13.1 ms	5.7 ms	6.7 ms
256 pixels	6.0 ms	8.3 ms	5.5 ms	13.6 ms	5.5 ms	6.4 ms

Table 5: Effect of small triangle bounding box area threshold

While a 32×32 bin size and 8×8 fine size work well for the immediate-mode rasterizer, the tiled and hybrid rasterizers use the fine size for a slightly different purpose. It forms part of the workgroup size and additionally influences the average number of triangle-tile overlaps. Varying it provides a form of "oversubscription" that helps address load-balancing issues arising from varying numbers of triangle-tile overlaps. An alternative to address load-balancing is full flattening, but this largely defeats the purpose of tile-based rasterization, namely exploiting locality. With the same constraints now on only imposed the coarse tile size, the fine tile size is varied.

bin \ fine	4x4	8x8	16x16
16x16	32.2 ms 59.22 MB		
32x32	37.4 ms 63.3 MB	29.4 ms 52.4 MB	
64x64	45.3 ms 68.8 MB	35.0 ms 57.2 MB	55.2 ms 49.8 MB
128x128		43.8 ms 61.1 MB	67.3 ms 54.6 MB

Table 6: Effect of fine tile size on tile-based rasterisation

Even though the memory usage pattern changed, the optimal tile size combination seems to still be 32×32 bin size and 8×8 fine size. That concludes the tuning. The next section provides the author's reflection on using a data-parallel model for software rasterization.

4.3 Reflections on the data-parallel approach

Figure 25: A fancy perspective render of the teacup model with shaders

The data-parallel Blelloch [4] primitives, such as `map`, `reduce`, `filter`, and `scan`, naturally and succinctly express the different stages of a 3D rendering pipeline. The transition from operating on a collection of triangles to operating on a collection of pixels poses the main challenge, and for this purpose the `expand` design API from Henriksen et al. [17] can be used to aid this transition. Once the transition is made, hardware-accelerated atomic min/max operations ought to be used to emulate the speed of GPU depth buffers as closely as possible, and Futhark offers `reduce_by_index` for this purpose.

The mapping from triangles to pixels can be done in various ways as described previously in this thesis. With edge functions, and really with scalar fields in general, the challenge of using them together with `expand` lies in the fact that one has to test a point before it is used (resulting in inefficiencies if done naively), but `expand` requires the size to be predetermined and provides local segment indices. This is worked around with small, compact bitmasks used hierarchically together with `rank` and `select` operations.

A different runtime complexity for tile-based rasterization is achieved in Futhark, but shortcomings in low-level implementation details likely have lead to suboptimal performance.

4.4 Future work

Based on the results presented in the previous section, two promising directions for future work emerge.

First, `expand_masked` (expand-filter) could be extended to efficiently handle both small and large segments. The current implementation relies on bitmasks and therefore requires segment sizes to remain small and bounded. A more general implementation could instead be built from fundamental Futhark SOACs, similarly to `expand_reduce` in diku-dk/segmented [7]. The existing bitmask-based implementation provides a useful starting point, but the results indicate that its performance degrades as bitmask sizes increase. However, using larger bitmasks may materialize fewer values, resulting in memory usage improvements, so trade-offs should be measured.

An efficient expand-filter SOAC would also broaden the range of applications of the flattening-by-expansion approach to problems involving scalar fields. In rasterization, it can be used to evaluate edge functions for triangles, circles, and other two-dimensional scalar fields. More generally, it could enable efficient processing of three-dimensional scalar fields $f : \mathbb{R}^3 \to \mathbb{R}$ in algorithms such as marching cubes *without* needing to materialize and process dummy values.

See Dyken and Ziegler [8], which describe a marching cubes implementation based on generalized stream compaction and expansion driven by a predicate function.

Second, a primitive for resolving the tension between load balancing and exploiting locality could be developed. For rasterization, one possible direction is to exploit the sorted structure of data in tile-based rasterization using a form of `segmented_reduce_by_index`. Such a primitive may use more low-level optimizations and intricate scheduling to achieve more balanced load distribution. It would alleviate the need to control such details at a high level in Futhark through tile sizes and/or chunking. It is unclear whether it could be built on top of existing primitives. It is also unclear how to develop it generically, as the tile size and load distribution depend on the problem and input distribution. Here a dynamic strategy may be used under the hood.

5 Conclusion

This thesis has explored how to perform Pineda rasterization using a data-parallel hierarchical tiling algorithm based on `expand_masked`, and implemented both sort-last, sort-middle Pineda-style rendering architectures and a hybrid thereof, with the goal of evaluating the suitability of Futhark as a high-level data-parallel language for implementing software rasterizers.

An expand-filter implementation, `expand_masked`, has been developed to perform Pineda-style rasterization. The current implementation relies on bitmasks, and therefore requires segment sizes to be small and bounded.

Using `expand_masked`, the sort-last Pineda rasterizer is slightly slower than the scanline variant and degrades further at higher resolutions than 1024×1024, while the sort-middle, tile-based Pineda rasterizer matches its speed at 8192×8192 with substantially lower peak memory usage.

It remains to be seen how to provide a more general `expand_masked` implementation from more fundamental Futhark SOACs, efficiently processing both small and large segments, and making Pineda-style rasterization just as scalable as scanline rasterization through a data-parallel, flattening-by-expansion approach. An efficient `expand_masked` implementation could be used to efficiently perform other algorithms involving scalar fields, such as the well-known marching cubes algorithm.

A Appendix - Source code

The code is provided as two github repositories, one for `expand_masked` and one for `futhrast`, the library implemented in this thesis.

The following software is required:

- `futhark` compatible with version `0.26.3`
- `libsdl2-dev` (for the examples)
- `libsdl2-ttf-dev` (for the examples)

To build `futhark`, set up `ghc` and `cabal`, and proceed as follows:

```
git clone https://github.com/diku-dk/futhark
cd futhark
make configure
make install
```

The code for `expand_masked` can be found in https://github.com/abxh/expand_masked (delivered commit: 0894bda). The code for `futhrast` can be found in https://github.com/abxh/futhrast (delivered commit: 11e0efe).

For setting up the examples, proceed as follows:

```
git clone https://github.com/abxh/futhrast --depth=1
cd futhrast
futhark pkg sync
cd examples
futhark pkg sync
cd <NUM-example>
make
```

For setting up the benchmarks, proceed as follows:

```
git clone https://github.com/abxh/futhrast --depth=1
cd futhrast
futhark pkg sync
cd benchmarks/<a-particular-benchmark>
# edit the Makefile to point to the correct environment variables
# and desired futhark backend
make build
make run
```

B Appendix - Background

B.1 Rendering with rasterization

So far, only rasterization in 2D have been considered in Section 2.1. In computer graphics, the goal is to simulate how light interacts with objects and how those interactions are perceived by the eye or a camera.

Two standard approaches for doing this are ray tracing and rasterization.

- *Ray tracing* simulates the path of light by casting rays from the eye or camera into a objects in 3D space and recursively following their interactions with surfaces, including reflection and refraction.

  This continues until a termination condition is reached such as hitting a light source or exceeding a recursion limit.

- *Rasterization* instead determines visibility by asking which geometry covers each pixel when projected onto the screen. It works by projecting 3D objects into 2D screen space and filling in the pixels they cover.

  After projection, lighting and shading are applied as approximations.

B.1.1 Projection

In the context of 3D rendering, a *projection* function is a function that maps 3D coordinates $(x, y, z) \in \mathbb{R}^3$ from world space to 2D coordinates $(x, y) \in \mathbb{R}^2$ in screen space.

There are two primary methods for projection: orthographic and perspective projection.

The former simply ignores the effect of Z-dimension on X and Y; it isn't very realistic over long Z-distances.

The latter is modeled after the *camera obscura* principle, a phenomenon that occurs when light passes through a small hole, which cameras utilize with the camera pinhole model.

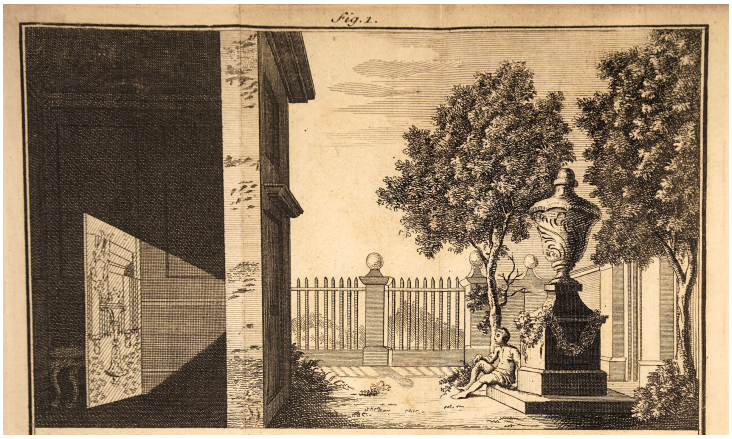


Figure 26: Camera obscura principle by James Ayscough from 1755 [44]

For the derivation, assume a simplified model of the real phenomenon; notably, a real camera produces an image inverted in both the x- and y-directions, an effect corrected by the camera lens.

Assume an infinitesimally small "eye" point is located at the origin, viewing along the positive Z-axis. The image plane (what the eye "sees" with it's idealized retina) is placed at $Z = 1$, and consider how a point $p = (x, y, z)$ is projected onto the image plane as the intersection point $P_{intersect} = (x', y', 1)$.

Additionally, define a field-of-view angle θ on both the ZX- and ZY-planes at the origin. Following the conventional definition of the field of view, consider half of this angle, $\theta/2$. By simple trigonometry, the line extending from the origin along the angle $\theta/2$ to $Z = 1$ intersects the image plane at $X = \tan(\theta/2)$ on the ZX-plane, and similarly at $Y = \tan(\theta/2)$ on the ZY-plane.

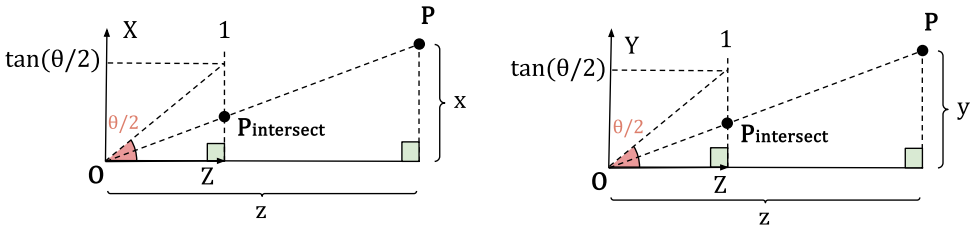


Figure 27: The basic idea behind perspective projection

By similar triangles, it follows that

$$x' = \frac{x}{z} \qquad\qquad y' = \frac{y}{z}. \tag{7}$$

We, however, restrict the projected points to a smaller subset of the image plane

$$\{(x, y, 1) \ : \ -1 \leq x \leq 1, \ -1 \leq y \leq 1\},$$

so that only points within this region are mapped to screen device coordinates in a well-defined manner.

So, to account for the field of view, ensure that points project to $x' = \pm \tan(\theta/2)$ is mapped to $x'' = \pm 1$ and similarly for y' by means of rescaling. Additionally, define the aspect ratio a, the ratio of screen width to screen height, to account for non-square screens. Including these additional scaling factors, it follows that

$$x'' = \frac{x}{\tan(\theta/2) \cdot a \cdot z} \qquad\qquad y'' = \frac{y}{\tan(\theta/2) \cdot z}. \tag{8}$$

With orthographic projection, we wish to account for non-square screens as well. So the mapping for orthographic projection is simply $x' = x/a$ and $y' = y$.

That concludes the brief overview of projection.

B.1.2 Camera frustum planes

Limiting the range of projected points, known as the (camera) *view* points, to

$$\{(x, y, 1) \ : \ -1 \leq x \leq 1, \ -1 \leq y \leq 1\}$$

defines a subset of the input world space $\mathbb{R}^3$ that is visible on the screen, thus forming the *left*, *right*, *top*, and *bottom* camera (frustum) planes that bound the region in which points are rendered, as shown in the following figure.

We additionally restrict visible points to lie in front of the camera (or eye), since points behind the camera should not be rendered and the projection becomes undefined at $z = 0$ (as $1/z$ is a division by zero then). To avoid this, we impose the constraint $z \geq z_{\text{near}} > 0$, where z_{near} defines the *near plane*.

Furthermore, for both performance and mathematical convenience, we introduce a *far plane* $z_{\text{far}} > z_{near}$, beyond which points are no longer rendered.

Using z_{near} and z_{far}, the z-coordinate can be mapped to a depth value $d \in [0, 1]$ (as described in the next section). Values outside this interval are clipped and not rendered (clipping is discussed later). This defines a *Normalized Device Coordinate* (NDC) space,

$$\{(x, y, d) \ : \ -1 \leq x \leq 1, \ -1 \leq y \leq 1, \ 0 \leq d \leq 1\},$$

which unifies both perspective and orthographic projections and allows the graphics pipeline to handle both cases uniformly.

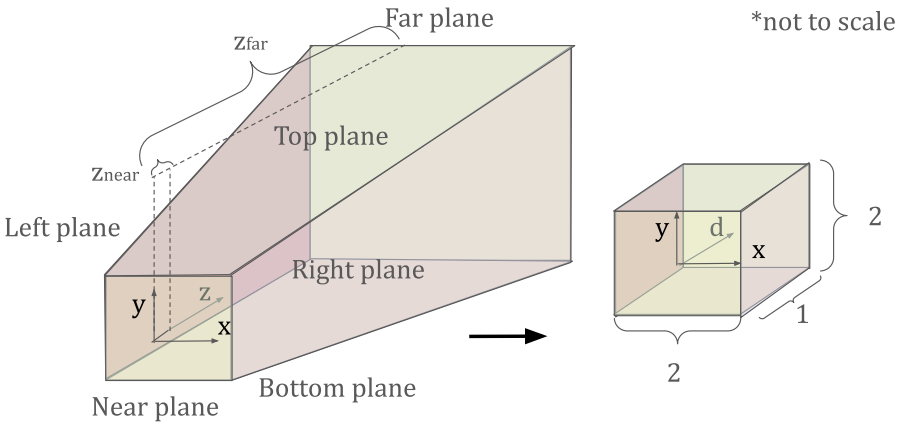


Figure 28: Camera frustum planes and the effect of depth mapping

B.1.3 Depth

This section describes depth, derived from z, as a means of determining which pixel lies in front of another, thereby providing a way to select the pixel closest to the "eye" or camera.

Unlike ray tracing, rendering by projection alone provides no direct way to determine the closest triangle at each pixel. One option is to sort triangles by their minimum z-value; this is known as the *painter's algorithm*. However, it fails in cases when there is a cyclic overlap between triangles such as the following:

Figure 29: Edge case for painters algorithm

Thus, for rasterizing 3D geometry, a per-pixel depth buffer must be maintained and updated by selecting the pixel fragment with the smallest depth value, corresponding to the point closest to the camera.

For the z-mapping, define z_{near} and z_{far} along the Z-axis. We wish to map z-values in the interval $[z_{near}; z_{far}]$ to the interval $[0; 1]$, since the depth value will be stored as a 32-bit floating point value which has the highest amount of precision at that output interval.

Additionally, since floating point values are (roughly) logarithmically distributed around 0, it turns out that it is ideal to map z_{near} to 1 and z_{far} to 0 to counteract the distortion created by $1/z$ projection mapping, thereby minimizing the precision loss [33]. Note, the interval $[-1; 1]$ is *not* used, as that would concentrate the distribution *in between* z_{near} and z_{far}.

Thus, for orthographic projection, the goal is to find a (linear) function $d_{ortho}(z)$ that maps $d_{ortho}(z_{near}) = 1$ and $d_{ortho}(z_{far}) = 0$. This is because a linear mapping proves to be advantageous for a matrix formulation of the depth mapping to be described later.

Given:

$$\begin{aligned} d_{ortho}(z_{near}) &= a(z_{near}) + b = 1 \\ d_{ortho}(z_{far}) &= a(z_{far}) + b = 0 \end{aligned} \tag{9}$$

The following equation can be deduced algebraically, fulfilling above constraints:

$$\Leftrightarrow d_{ortho}(z) = \frac{z_{far}}{z_{far} - z_{near}} - \frac{1}{z_{far} - z_{near}} \cdot z \tag{10}$$

Similarly, with perspective projection the goal is instead to find the constants for the function $d_{pers}(z) = a\frac{1}{z} + b$ s.t. $d_{pers}(z_{near}) = 1$ and $d_{pers}(z_{far}) = 0$. Note, the $1/z$ (as both the input and output) is chosen for perspective correct interpolation of depth (to be described later).

Given:

$$\begin{aligned} d_{pers}(z_{near}) &= a\frac{1}{z_{near}} + b = 1 \\ d_{pers}(z_{far}) &= a\frac{1}{z_{far}} + b = 0 \end{aligned} \tag{11}$$

The following equation can be deduced algebraically, fulfilling above constraints:

$$d_{pers}(z) = \frac{z_{near} \cdot z_{far}}{z_{far} - z_{near}} \cdot \frac{1}{z} - \frac{z_{near}}{z_{far} - z_{near}} \tag{12}$$

Taking the limit of d_{pers} as z_{far} approaches infinity, the following function is obtained:

$$d_{pers\_inf}(z) = \lim_{z_{far} \to \infty} d_{pers}(z) = z_{near}\frac{1}{z}, \tag{13}$$

where above function correspond to having a infinitely far far-plane i.e. when the far-plane is non-existent, while still mapping z values to the range $[0; 1]$ when $z_{near} \leq z < \infty$.

To reiterate, the above functions map z-coordinates to the interval $[0; 1]$, assigning the largest value to the pixel closest to the camera depending on the kind of projection made, while preserving linearity for interpolation purposes and taking advantage of floating-point bit representation.

B.1.4 Homogenous coordinates

This section briefly describes homogeneous coordinates, which unify the concept of $1/z$ perspective division while also allowing translations through matrix multiplication, thus enabling most of the desired *affine* transformations in graphics — affine in contrast to linear, which leave the origin fixed.

To convert a ordinary 3D coordinate $P = (x, y, z)$ to a homogeneous 3D coordinate, simply append a w coordinate of 1 as follows:

$$(x, y, z) \mapsto (x, y, z, 1) \tag{14}$$

To convert *back* to ordinary 3D, when $w \neq 0$, divide by w as follows:

$$(x, y, z, w) \mapsto (x/w, y/w, z/w) \tag{15}$$

When $w = 0$, the vector $\begin{bmatrix} x & y & z & 0 \end{bmatrix}^T$ is to be interpreted as a direction vector in homogeneous space. When $w = 1$, the vector $\begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$ is to be interpreted as a position vector in homogeneous space.

One of the advantages of this representation is that, it is possible to perform translation on 3D vectors by means of 4x4 matrix multiplication:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix} \tag{16}$$

Other ordinary linear transformations in 3D can be represented as a 3x3 matrix embedded in a homogeneous 4x4 matrix as follows:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \mapsto \begin{bmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{17}$$

Note how a direction vector with $w = 0$ in homogeneous space remain unaffected by the translation column vector. The line going the direction of $(x, y, z, 0)$ in homogeneous space represent the set $\{(\lambda x, \lambda y, \lambda z, \lambda w) : w \neq 0\}$, all mapped to the same point in 2D space by $1/w$ division. Moreover, the representation follows intuitive algebraic laws of direction + position = position + direction = position, and direction + direction = direction [39].

Gathering on Section B.1.1 and B.1.3 on projection and depth, the homogeneous matrix formulation of orthographic, perspective, and perspective with infinitely far far-plane is as follows:

$$P_{ortho} = \begin{bmatrix} \frac{1}{a} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{1}{z_{far} - z_{near}} & \frac{z_{far}}{z_{far} - z_{near}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{18}$$

$$P_{pers} = \begin{bmatrix} \frac{1}{\tan(\theta/2) \cdot a} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\theta/2)} & 0 & 0 \\ 0 & 0 & -\frac{z_{near}}{z_{far} - z_{near}} & \frac{z_{near} \cdot z_{far}}{z_{far} - z_{near}} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{19}$$

$$P_{pers\_inf} = \begin{bmatrix} \frac{1}{\tan(\theta/2) \cdot a} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\theta/2)} & 0 & 0 \\ 0 & 0 & 0 & z_{near} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{20}$$

Note how the perspective projection leaves w to become equal to z with above matrices. Thus, the constants used in above matrix is found by *multiplying* by z with the depth functions d_{pers} and d_{pers_inf}.

These matrices are typically used as the final transformation step in a sequence applied to triangle vertices. First, a model (or object) specific matrix M transforms the vertices from local object space to global world space. Next, a view (or camera) matrix V positions and orients the scene relative to the camera, ensuring that visible vertices lie within the camera frustum. Finally, the projection matrix P is applied, transforming the vertices from world space into homogeneous clip space. This delays the division by w until after clipping, allowing the rasterizer to safely process vertices while maintaining a non-negative z-coordinate. Together, these three matrices form the model-view-projection matrix:

$$MVP = P \cdot V \cdot M$$

B.1.5 Clipping

This section briefly describes the high-level role and properties of a clipper. It does not describe clipping algorithms in detail, as the implementation is sequential (with respect to each primitive) and reference implementations are available in other programming languages than Futhark. The Liang-Barsky algorithm [42] is used for line clipping with the camera frustum, and the Sutherland-Hodgman algorithm [43] is used for triangle clipping with the camera frustum.

As mentioned earlier, clipping against the near plane must be performed to both avoid perspective division by zero and *cull* (remove) triangles completely behind the camera. Clipping against the other planes is optional. However, clipping against the planes corresponding to the screen borders helps keep edge-function coefficients within the range representable by fixed-point arithmetic, while far-plane clipping is primarily a performance optimization from the clipper's perspective.

In preparation for the shading stage, the user provides per-vertex attributes (colors, normal vectors, etc.) that are interpolated during shading to produce the final pixel color. However, the clipper itself must also interpolate vertex attributes, since clipping a primitive modifies it's vertices and therefore requires the associated attributes to be interpolated accordingly.

With the defined NDC space in Section B.1.2, a homogeneous point is in the camera frustum if

$$-w \leq x \leq w, -w \leq y \leq w, 0 \leq z \leq w \tag{21}$$

all hold true (after substituting d with the variable z). These inequalities can be used to deduce the following equivalent inequality.

$$\begin{aligned} \text{left plane:} \quad & x + w \geq 0 \\ \text{right plane:} \quad & -x + w \geq 0 \\ \text{bottom plane:} \quad & y + w \geq 0 \\ \text{top plane:} \quad & -y + w \geq 0 \\ \text{far plane:} \quad & z \geq 0 \\ \text{near plane:} \quad & -z + w \geq 0 \end{aligned} \tag{22}$$

The term on the left-hand side of each inequality additionally represents the distance between the point and the corresponding plane.

Using these distances, it is possible to determine the intersection between a line and a given camera-frustum (clipping) plane. By representing the intersection point as an abstract quantity $t : 0 \leq t \leq 1$, vertex attributes can likewise be interpolated.

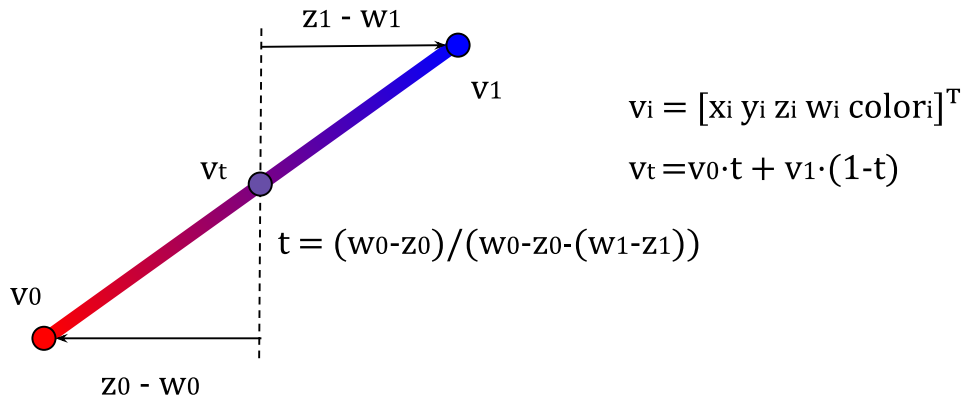


Figure 30: The basic idea behind Liang-Barsky clipping (for the near plane)

Clipping triangles has an additional wrinkle: clipping a triangle against a plane may produce 0, 1, or 2 triangles, i.e. at most one additional vertex per plane. Clipping against all 6 frustum planes therefore produces a variable number of triangles, and the Sutherland-Hodgman algorithm consequently requires a temporary buffer of up to 9 vertices per clipped triangle.

Since clipping is expensive, triangles should first be classified as either trivially rejected or trivially accepted. A triangle is trivially rejected if all vertices lie outside the same clipping plane, and trivially accepted if all vertices satisfy the inequalities in 22. Otherwise, the algorithm proceeds to either trim the triangle by interpolating edge vertices, or split it into two triangles by a case-by-case classification of vertices as shown below.

Both Liang-Barsky and Sutherland-Hodgman then repeat the process for the remaining planes, with the additional complication that the latter algorithm must also clip all triangles generated in the previous iteration.

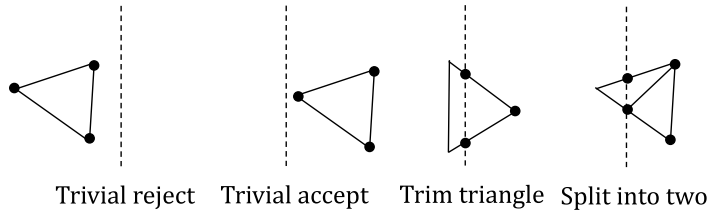


Figure 31: Basic triangle clipping idea, with left side of the plane representing the space outside the frustum

B.2 Shading

This section describes how barycentric coordinates (See Section 2.1.4) are used to interpolate triangle vertex attributes such as depth, color and normals. It does not describe texture sampling, which in the current implementation can instead be performed externally by the user, although GPU hardware typically provides dedicated texture-sampling units. Additionally, it clarifies the role of user provided "shader" functions in updated overview of the rendering pipeline, as a prelude to the section about how it can be parallelized.

B.2.1 Use of barycentric coordinates

Assume the vertex color space lives in a vector space, such that addition between colors and scaling with a constant factor in $\mathbb{R}$ is well-defined.

Given the colors c_1, c_2 and c_3 for each triangle vertex, the color for a given pixel in the triangle can be obtained using barycentric coordinates as follows:

$$c(\boldsymbol{p}) = \alpha(\boldsymbol{p}) \cdot c_1 + \beta(\boldsymbol{p}) \cdot c_2 + \gamma(\boldsymbol{p}) \cdot c_3 \quad (23)$$

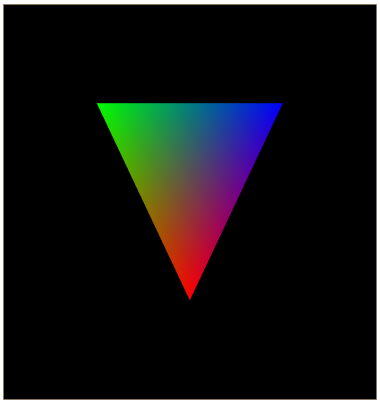


Figure 32: Result of barycentric interpolation of red, blue and green vertices

It is often desirable to interpolate vertex normals and 3D vertex positions across the 2D triangle; this can be done using similar mathematics.

There is, however, one important and well-known caveat: if a vertex position is transformed by a model-view matrix MV, its normal must instead be transformed by $((MV)^{-1})^T$.

Nevertheless, the barycentric interpolation itself remains unchanged.

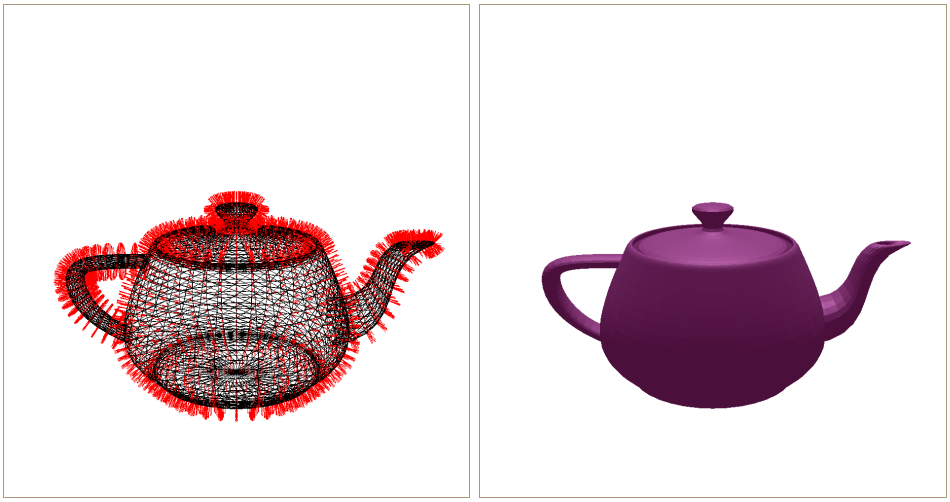


Figure 33: Phong shading [41] using interpolated vertex 3D positions and normals

B.2.2 Perspective-correct interpolation

Interpolating vertex attributes in 2D has a complication: the projected half-way point in 2D is not always equal to the half-way point in 3D. Straightforward linear interpolation in 2D does not produce visually correct results (for 3D objects projected with perspective projection).

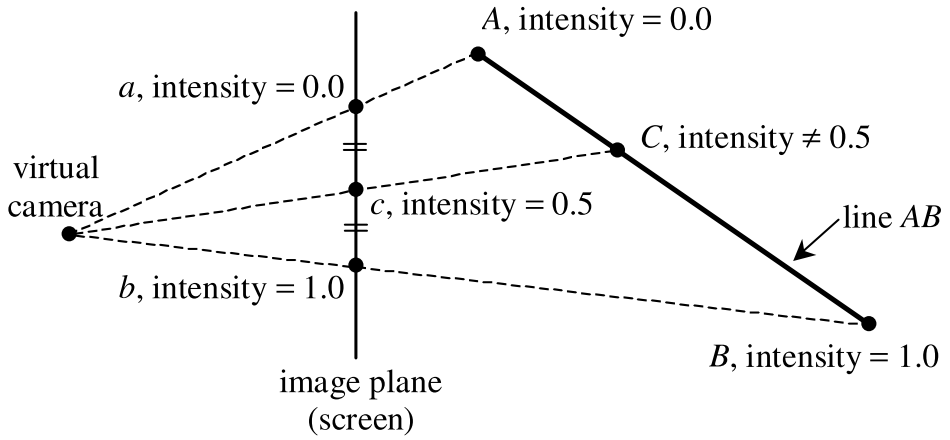


Figure 34: Figure taken from Low [29] to illustrate above point

Using the math derived in Low [29], it follows that interpolating $1/z$ instead of z preserves linearity under perspective projection. This justifies the depth representation used in Section B.1.3.

The same must be done for other vertex attributes as well. Given 3D vertex positions v_1, v_2, v_3 and vertex attributes a_1, a_2, a_3, which live in a vector space similar to that defined in Section B.2.1, the attributes must instead be interpolated using the following formula for perspective correct interpolation following the derivation in Low [29]:

$$c(\boldsymbol{p}) = \frac{\frac{1}{\boldsymbol{v}_{1z}} \cdot \alpha(\boldsymbol{p}) \cdot a_1 + \frac{1}{\boldsymbol{v}_{2z}} \cdot \beta(\boldsymbol{p}) \cdot a_2 + \frac{1}{\boldsymbol{v}_{3z}} \cdot \gamma(\boldsymbol{p}) \cdot a_3}{\alpha(\boldsymbol{p}) \cdot \frac{1}{\boldsymbol{v}_{1z}} + \beta(\boldsymbol{p}) \cdot \frac{1}{\boldsymbol{v}_{2z}} + \gamma(\boldsymbol{p}) \cdot \frac{1}{\boldsymbol{v}_{3z}}} \tag{24}$$

For lines, whose attributes can be interpolated with a simple linear interpolation, their attributes must also be interpolated with in a perspective correct manner for correctness. Given a abstract quantity t s.t $0 \leq t \leq 1$ derived from pixel position $\boldsymbol{p}$, line vertex interpolation with vertices v_1, v_2 and attributes a_1, a_2 is done as follows:

$$c(\boldsymbol{p}) = \frac{\frac{1}{\boldsymbol{v}_{1z}} \cdot (1-t) \cdot a_1 + \frac{1}{\boldsymbol{v}_{2z}} \cdot t \cdot a_2}{\frac{1}{\boldsymbol{v}_{1z}} \cdot (1-t) + \frac{1}{\boldsymbol{v}_{2z}} \cdot t} \tag{25}$$

Note that, to correctly render 2D triangles derived from projected 3D triangles, the inverse depth value must be used to interpolate vertex attributes in a perspective-correct manner. Thus, after applying the MVP matrix to triangle vertices, performing clipping, and dividing the remaining coordinates by w, the $1/w$ value (corresponding to the original inverse-z) must be preserved for shading purposes, as illustrated in Figure 11.

C Appendix - Design and implementation

C.1 Modularization and setup

This section briefly describes the code surrounding the rasterizers, which supports 3D rendering via rasterization.

C.1.1 Rasterizers

Firstly, the rasterizers themselves must be modularized to allow flexible use. The ”rasterizers” perform both rasterization and shading, since both operate on the same input information. The triangle rasterizer modules have the following module type:

```
module type TriangleRasterizerSpec =
  (V: VaryingSpec)
  -> {
    val rasterize 'target [n] [h] [w] :
      (plot: fragment V.t -> target)
      -> ([h][w]target, [h][w]f32)
      -> [n](fragment V.t, fragment V.t, fragment V.t)
      -> ([h][w]target, [h][w]f32)
  }
```

It takes a `plot` function acting as the fragment shader, producing a `target`, typically an RGBA color. It then takes the input color and depth buffers, and the triangle vertices, treated as fragments containing world positions, and produces updated color and depth buffers. The varying type `V.t` is passed as a module, as the varying module `V` also provides an addition `V.+` and scaling `V.*` operator.

An example instance is provided at the end of each file defining the triangle rasterizers. For simplicity, a boolean attribute is first defined using a module. This does not strictly satisfy the algebraic properties of a vector space, but is defined as such for illustrative purposes.

```
local
module V : VaryingSpec with t = bool = {
  type t = bool
  def (+) = (||)
  def (*) s x = if bool.f32 s then x else false }
```

This module is then passed to the rasterizer, a parameterized module taking a `VaryingSpec`-compliant module defining `M`. Using `M`, the following is defined:

```
def test_triangle_pineda_tiled [n] (h: i64) (w: i64)
    (vs: [n]((f32, f32), (f32, f32), (f32, f32))) : [h][w]i32 =
  let target_buffer = replicate h (replicate w false)
  let depth_buffer = replicate h (replicate w 0)
  let frags =
    vs
    |> map (\(f0, f1, f2) ->
      ( {pos = {x = f0.0, y = f0.1}, depth = 1, Z_inv = 1, attr = true}
      , {pos = {x = f1.0, y = f1.1}, depth = 1, Z_inv = 1, attr = true}
      , {pos = {x = f2.0, y = f2.1}, depth = 1, Z_inv = 1, attr = true}
      ))
  let plot = (\(f: fragment bool) -> f.attr)
  in M.rasterize plot (target_buffer, depth_buffer) frags
    |> (.0)
    |> map (map i32.bool)
```

This is the basic structure external users are expected to follow when using the rasterizer. It is more general than a conventional 2D rasterizer, as it is designed to handle projected 3D triangles. The fragment shader implementation is user-defined and, in this case, simply returns the fragment attribute. The function can be used in an interpreter as follows:

```
> test_triangle_tiled_pineda 8 8 [((0,0),(7.5,0),(0,7.5))]
[[1, 1, 1, 1, 1, 1, 1, 0],
 [1, 1, 1, 1, 1, 1, 0, 0],
 [1, 1, 1, 1, 1, 0, 0, 0],
 [1, 1, 1, 1, 0, 0, 0, 0],
 [1, 1, 1, 0, 0, 0, 0, 0],
 [1, 1, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0]]
```

C.1.2 Varying

Both clipping and shading require operators for interpolating *user-defined* attributes. The user is therefore required to define these operators once in a module and pass the module itself, rather than supplying types and operators separately as parameters.

```
module type VaryingSpec = {
  type t
  val (+) : t -> t -> t
  val (*) : f32 -> t -> t
}
```

These operators are expected to satisfy algebraic rules similar to those of a vector space, although the module type is incomplete since a notion of magnitude is not required.

Once a user provides a `VaryingSpec`-compliant module, a parameterized module `VaryingExtensions` is used to extend the abstract type `V.t` with `V.-` and `V./` operators, as well as linear and barycentric interpolation. Perspective-correct interpolation functions are also derived. `VaryingExtensions` is intended for internal use.

C.1.3 Fragment

The code distinguishes between "unprojected" and projected fragments, referred to as `vertex_out` and `fragment`, respectively. The "unprojected" fragments are defined in homogeneous clip space, while the latter are ultimately represented in screen space.

```
type vertex_out 'varying =
  { pos: {x: f32, y: f32, z: f32, w: f32}
  , attr: varying }
type fragment 'varying =
  { pos: {x: f32, y: f32}, depth: f32, Z_inv: f32
  , attr: varying }
```

A `proj` function of type `vertex_out varying -> fragment varying` is provided to divide coordinates by `w`, set `depth` to `z/w`, and store `1/w` as `Z_inv`. An `unproj` function of type `fragment varying -> vertex_out varying` uses `Z_inv` to reverse this division.

C.1.4 Shaders

Shader function types are represented using polymorphic function types:

```
type^ vertex_shader 'uniform 'varying 'vertex =
  uniform -> vertex -> vertex_out varying
type^ fragment_shader 'uniform 'varying 'target =
  uniform -> fragment varying -> target
```

Both functions are expected to receive user-defined `uniform` constants. Once partially applied, the `fragment_shader` can be passed to the rasterizers for fragment (or pixel) shading purposes. The vertex shader operates on user-defined `vertex` values, and the user is expected to produce a `vertex_out`, which is exposed to the rest of the pipeline.

C.1.5 Setup

The model data is expected to be packed in the following structure:

```
type~ model_data 'vertex =
  { primitive_type: #points | #lines | #triangles
  , vertices: []vertex
  , indices: []i64 }
```

Currently, the `render_config` is set up to allow runtime selection of the triangle winding order, with a `#neither` option that corrects the winding order to the convention used by the rasterizer modules.

```
type render_config =
  { triangle_winding_order: #clockwise | #counterclockwise | #neither }
```

The setup module uses a user-defined varying module to instantiate setup functions such as `render`, `render_wireframe`, and `render_normals`, which act as implicit geometry-shader stages in the pipeline. Additional setup functions can be defined to support alternative geometry-shader-like behavior.

The primary entry point, `render`, takes the triangle winding order, model data, uniform constants, and vertex and fragment shaders, and updates given target color and depth buffers. The alternative functions `render_wireframe` and `render_normals` use the line rasterizer while interpreting given vertices as triangle components. And, notably, `render_wireframe` applies `expand` to edges derived from vertex triplets.

The following paragraphs describes how `render` constructs vertices before rasterization according to `model_data`'s `primitive_type`. The function `f` performs `proj` and scales screen coordinates by the buffer dimensions to obtain pixel indices.

Independently of the primitive type, all vertices `d.vertices` in the model data structure `d` are first processed by the vertex shader `on_vert` with uniform constants `u`. They are then indexed using the vertex indices `d.indices`.

```
let vs = map (on_vert u) d.vertices
let vs = map (\i -> vs[i]) d.indices
```

Points are then rasterized as follows. First, a quick test is performed to check whether they lie inside the camera frustum (Section B.1.2). The homogeneous coordinates are then projected and scaled by the screen dimensions, before being passed to a specialized point rasterizer.

```
vs
|> filter test_point_bounds
|> map (\(v0) -> f v0)
|> Point.rasterize (on_frag u) (target_buffer, depth_buffer)
```

Lines are first assembled as vertex pairs. They are then clipped to fit inside the camera frustum using the Liang-Barsky algorithm (Section B.1.5). The clipper performs interpolation using a `lerp_vertex_fragment` function defined via the varying operators. Finally, `f` is applied to the line vertices, which are then passed to the line rasterizer.

```
(iota (length vs / 2))
|> map (\i -> (vs[2 * i], vs[2 * i + 1]))
|> clip_lines lerp_vertex_fragment
|> map (\(v0, v1) -> (f v0, f v1))
|> Line.rasterize (on_frag u) (target_buffer, depth_buffer)
```

Triangles follow a similar setup to lines, but use vertex triplets. They are clipped using a specialized triangle clipper via `clip_triangles` and then culled if they do not satisfy the correct winding order or have zero area, before being passed to the screen rasterizer. Before the culling step, the last two vertices are swapped, since mapping to screen dimensions flips the `y` coordinate to match SDL conventions, thereby reversing the winding order.

```
(iota (length vs / 3))
|> map (\i -> (vs[3 * i], vs[3 * i + 1], vs[3 * i + 2]))
|> map (\(v0, v1, v2) -> (v0, v2, v1))
|> clip_triangles lerp_vertex_fragment
|> map (\(v0, v1, v2) -> (f v0, f v1, f v2))
|> filter (\tri -> winding_order_test c tri && zero_bbox_area_test tri)
|> Triangle.rasterize (on_frag u) (target_buffer, depth_buffer)
```

The triangle clipper is particularly interesting. Since sequential clipping is expensive, triangles are `partition`'ed into three categories: (1) triangles completely outside the frustum, (2) triangles fully inside the frustum, and (3) triangles partially inside the frustum. Only case (3) requires clipping, while case (1) is fully culled. The new triangles produced by clipping are first stored in a temporary buffer of capacity 9 per input triangle, then `expand`'ed, and finally appended together with the triangles that do not require clipping.

C.1.6 Additional module parameterization

Common to both the setup and rasterizer modules is that they expose additional parameters as module parameters. The setup module takes an triangle rasterizer module parameter, since multiple rasterizers are available, although a default setup is provided. The Pineda-style rasterizers expose modules for configuring tile size, and tile masks, while also providing a default configuration.

C.1.7 Miscellaneous math

Functions for both orthographic and perspective projection with the expected depth interval and coordinate system are provided.

A small vector implementation is included, based on `athas/vector` [15], with modifications tailored to the author's mental model. A `fixedpoint` implementation is provided, intended for internal use by the Pineda-style rasterizers to introduce a small bias to the edge coefficients.

A `transform` module defines basic affine transformations through 4×4 matrices, including translation, scaling, and shearing. A `rotation` module based on unit quaternions is provided for constructing 4×4 rotation matrices, avoiding issues associated with orientation representation when using Euler angles alone while enabling smooth angular interpolation.

D Appendix - Discussion and evaluation

D.1 Correctness

It is difficult to create good automated tests on correctness for graphics. One can provide tests for the exact picture generated on one computer, and validate the same picture is generated on another computer or Futhark backend. But such tests only test reproducibility and reveal inconsistencies between implementations (possibly unrelated to the library). So, for visual correctness, one is better off interactively testing models and find edge cases.

Throughout the development, the following process has been used to ensure visual correctness. Taking near-plane clipping as an example:

1. start with a renderer without known bugs,
2. implement near-plane clipping,
3. move a model interactively so that it intersects the near plane,
4. inspect the output for visual inconsistencies,
5. and fix any bugs observing the visual cues before proceeding to the next feature, i.e. step (1).

Otherwise, ensure the mathematical formulae typed in code remains consistent with theory; for this purpose, effectively expressing math syntax is a good idea, something which Futhark does well.

Having said that, a known bug that remains unresolved is the implementation of the top-left fill convention in the scanline rasterizer. Without this convention, triangles draw on top of each other at their shared edges, and barycentric coordinates can fall outside the range $[0, 1]$. Clamping these values reduces the effect but still produces few black artifacts shown below.

The image shows said artifacts at the rabbit ears and feet. Otherwise, they can be reproduced through code. Nonetheless, the artifacts should not affect the performance much as it is a result of some small overdraw, patched through clamping, and doesn't seem to in practice either.

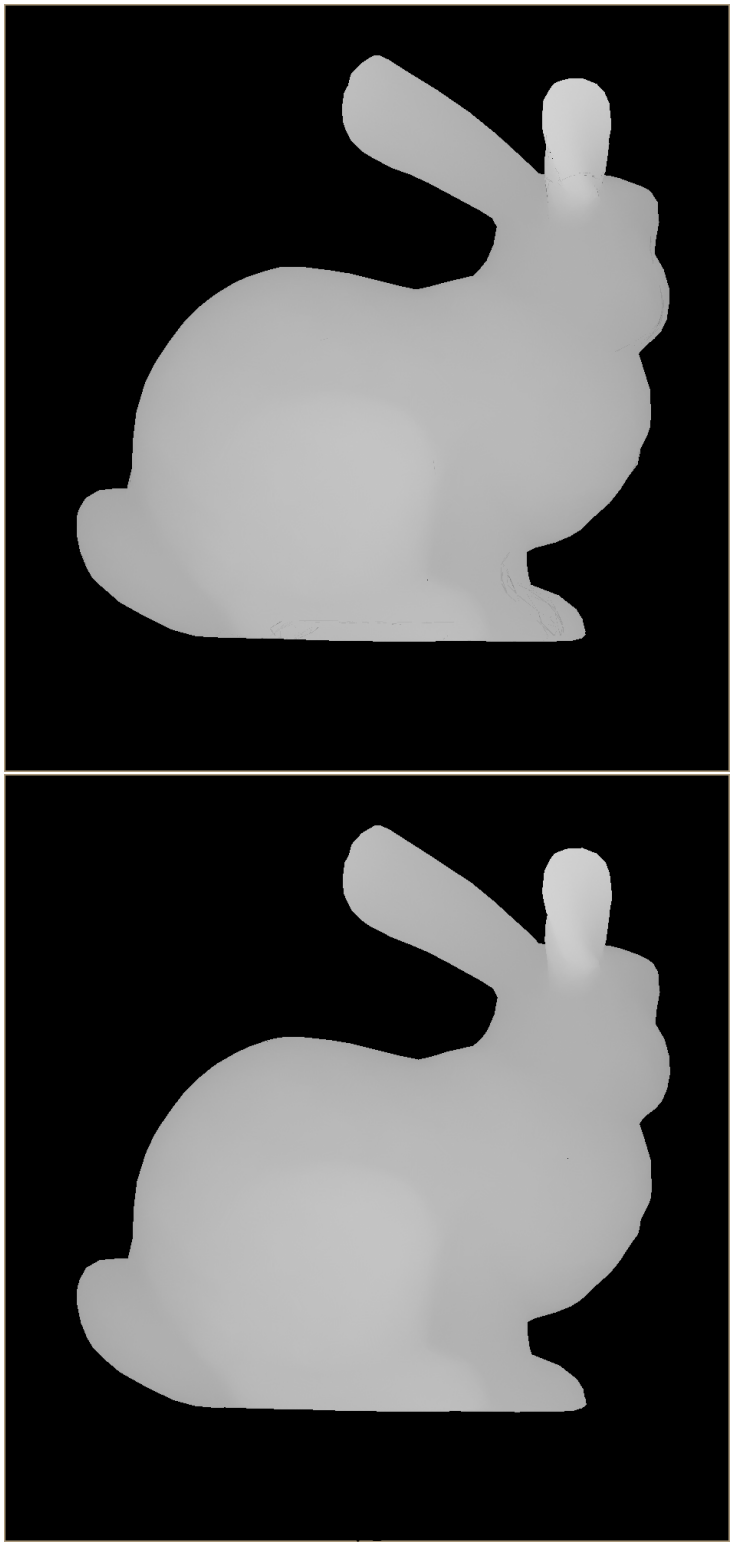


Figure 35: Scanline rasterizer (top), and Pineda-style rasterizer (bottom).

E Appendix - AI declaration

☐ I/we have used generative AI as an aid/tool.
✓ I/we have not used generative AI as an aid/tool.

List which GAI tools you have used and include the link to the platform:

Not applicable.

Describe how generative AI has been used in the exam paper:

Not applicable.

References

[1] M. Abrash. Rasterization on larrabee, 2009. URL https://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/readings/abrash09_lrbrast.pdf. Reprinted in CMU course 15-869 reading list; originally published in Dr. Dobb's Journal / GDC 2009.

[2] Arm Limited. Immediate-mode gpus, 2024. URL https://developer.arm.com/documentation/102662/0100/Immediate-Mode-GPUs. Accessed: 2026-04-23.

[3] Arm Limited. Tile-based gpus, 2024. URL https://developer.arm.com/documentation/102662/0100/Tile-based-GPUs. Accessed: 2026-04-23.

[4] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie Mellon University, School of Computer Science, Nov. 1990. URL https://www.cs.cmu.edu/~guyb/papers/Ble90.pdf.

[5] C. A. Burns and W. A. Hunt. The visibility buffer: A cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques*, 2(2):55–69, 2013. URL https://jcgt.org/published/0002/02/04/.

[6] DIKU-DK. Futhark sorts library, . URL https://github.com/diku-dk/sorts. Futhark standard library for sorting, accessed: 2026-05-27.

[7] DIKU-DK. Futhark segmented library, . URL https://github.com/diku-dk/segmented. Futhark standard library for irregular segmented operations, accessed: 2026-05-27.

[8] C. Dyken and G. Ziegler. Histopyramid stream compaction and expansion, Oct. 2007. URL https://paperzz.com/doc/7956075/histopyramid-stream-compaction-and-expansion. Advanced Computer Graphics / Vision Seminar, TU Graz, October 23, 2007.

[9] M. Elsman, T. Henriksen, D. Annenkov, and C. E. Oancea. Static interpretation of higher-order modules in futhark: Functional gpu programming in the large. *Proceedings of the ACM on Programming Languages*, 2(ICFP):97:1–97:30, 2018. doi: 10.1145/3236792. URL https://hjemmesider.diku.dk/~zgh600/Publications/futhark-modules-icfp18.pdf.

[10] Futhark Project. Computing histograms in futhark. URL https://futhark-lang.org/examples/histograms.html. Accessed: 2026-05-27.

[11] F. Giesen. A trip through the graphics pipeline 2011, part 5, 2011. URL https://fgiesen.wordpress.com/2011/07/05/a-trip-through-the-graphics-pipeline-2011-part-5/. The ryg blog, Accessed: 2026-06-08.

[12] F. Giesen. Optimizing the basic rasterizer, 2013. URL https://fgiesen.wordpress.com/2013/02/10/optimizing-the-basic-rasterizer/. The ryg blog, Accessed: 2026-05-14.

[13] GPUOpen. Work graphs api – compute rasterizer learning sample, 2024. URL https://gpuopen.com/learn/work_graphs_learning_sample/. Accessed: 2026-04-23.

[14] N. Greene. Hierarchical polygon tiling with coverage masks. In *Proceedings of ACM SIGGRAPH 1996*, pages 65–74. ACM, 1996. URL https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15869-f11/www/readings/greene96_hpt.pdf. Also available via CiteseerX mirror.

[15] T. Henriksen. vector: Efficient statically sized vectors (futhark library). URL https://github.com/athas/vector. Futhark library repository, accessed: 2026-05-27.

[16] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea. Futhark: Purely functional gpu programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 556–571. ACM, 2017. doi: 10.1145/3062341.3062354. URL https://doi.org/10.1145/3062341.3062354.

[17] T. Henriksen, N. G. W. Serup, M. Elsman, F. Henglein, and C. E. Oancea. Data-parallel flattening by expansion. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '19)*, 2019. doi: 10.1145/3315454.3329955. URL https://doi.org/10.1145/3315454.3329955.

[18] T. Henriksen, F. Thorøe, M. Elsman, and C. E. Oancea. Incremental flattening for nested data parallelism. In *Proceedings of the 24th*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2019), 2019. URL https://futhark-lang.org/publications/ppopp19.pdf.

[19] T. Henriksen, S. Hellfritzsch, P. Sadayappan, and C. E. Oancea. Compiling generalized histograms for gpu. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, 2020. URL https://hjemmesider.diku.dk/~zgh600/Publications/gen-histo-sc20.pdf.

[20] H. E. Holten-Lund. *Design for Scalability in 3D Computer Graphics Architectures*. PhD thesis, Technical University of Denmark, Informatics and Mathematical Modelling, Mar. 2002. URL https://www2.imm.dtu.dk/pubdb/edoc/imm888.pdf. Ph.D. thesis, IMM-PHD-2002-97.

[21] G. Humphreys, M. Houston, R. Ng, R. Frank, S. Ahern, P. D. Kirchner, and J. T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters. In *ACM SIGGRAPH 2002*, pages 693–702. ACM, 2002. doi: 10.1145/566570.566639. URL https://graphics.stanford.edu/papers/cr/.

[22] Imagination Technologies. Per-tile rasterization (renderer), . URL https://docs.imgtec.com/starter-guides/powervr-architecture/html/topics/per-tile-rasterization.html.

[23] Imagination Technologies. Tile-based deferred rendering (tbdr), . URL https://docs.imgtec.com/starter-guides/powervr-architecture/html/topics/tile-based-deferred-rendering-index.html.

[24] M. Kenzel, B. Kerbl, D. Schmalstieg, and M. Steinberger. A high-performance software graphics pipeline architecture for the gpu. *ACM Transactions on Graphics*, 37(4):1–15, 2018. doi: 10.1145/3197517.3201374. URL https://doi.org/10.1145/3197517.3201374. SIGGRAPH 2018.

[25] B. Kerbl, M. Kenzel, D. Schmalstieg, and M. Steinberger. Effective static bin patterns for sort-middle rendering. In *Proceedings of High Performance Graphics (HPG) 2017*, 2017. URL https://www.tugraz.at/fileadmin/user_upload/Institute/ICG/Downloads/team_steinberger/Publications/EffectiveStaticBinPatternsForSortMiddleRendering.pdf.

[26] S. Laine and T. Karras. High-performance software rasterization on gpus. In *Proceedings of High Performance Graphics 2011*, 2011. URL https://research.nvidia.com/sites/default/files/pubs/2011-08_High-Performance-Software-Rasterization/laine2011hpg_paper.pdf. HPG 2011.

[27] S. Laine and J. Pantaleoni. Gpu software rasterization. Beyond Programmable Shading (BPS) 2011 Workshop, 2011. URL https://web.archive.org/web/20130908130121/http://bps11.idav.ucdavis.edu/talks/08-gpuSoftwareRasterLaineAndPantaleoni-BPS2011.pdf. Presentation slides, accessed: 2026-04-23.

[28] R. W. Larsen and T. Henriksen. Strategies for regular segmented reductions on gpu. In *Proceedings of the ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC'17)*, 2017. URL https://futhark-lang.org/publications/fhpc17.pdf.

[29] K.-L. Low. Perspective-correct interpolation. Technical report, University of North Carolina at Chapel Hill, Mar. 2002. URL https://www.comp.nus.edu.sg/~lowkl/publications/lowk_persp_interp_techrep.pdf. Technical Report.

[30] J. McCormack and R. McNamara. Tiled polygon traversal using half-plane edge functions. In *Proceedings of the ACM SIGGRAPH / EUROGRAPHICS Workshop on Graphics Hardware*, 1999. URL https://people.csail.mit.edu/ericchan/bib/pdf/p15-mccormack.pdf.

[31] Microsoft. Rasterization rules. URL https://learn.microsoft.com/en-us/windows/win32/direct3d9/rasterization-rules. Accessed: 2026-05-09.

[32] L. Molnar. Sorting, 1994. URL https://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15869-f11/www/readings/molnar94_sorting.pdf. Carnegie Mellon University lecture notes.

[33] NVIDIA Developer. Visualizing depth precision, 2018. URL https://developer.nvidia.com/blog/visualizing-depth-precision/. Accessed: 2026-05-21.

[34] J. Pineda. A parallel algorithm for polygon rasterization. In *Proceedings of ACM SIGGRAPH 1988*, volume 22, pages 17–20. ACM, 1988. URL https://people.csail.mit.edu/ericchan/bib/pdf/p17-pineda.pdf.

[35] G. Sanderson. Cross products in the light of linear transformations. https://www.youtube.com/watch?v=BaM7OCEm3G0, 2016. Chapter 11 of the *Essence of Linear Algebra* series, 3Blue1Brown YouTube channel, accessed 2026-06-01.

[36] V. Schatz. Floating-point formats and ieee 754. https://www.volkerschatz.com/science/float.html. Accessed: 2026-05-15.

[37] M. Schütz, L. Lipp, E. Kristmann, and M. Wimmer. Curast: Cuda-based software rasterization for billions of triangles. 2026. URL https://arxiv.org/abs/2604.21749.

[38] SIGGRAPH Advances in Real-Time Rendering. Nanite virtualized geometry: A deep dive. https://www.youtube.com/watch?v=eviSykqSUUw. YouTube video, accessed 2026-06-06.

[39] D. V. Sokolov. Lesson 4: Perspective projection. URL https://github.com/ssloy/tinyrenderer/wiki/Lesson-4:-Perspective-projection#wait-a-second-it-is-forbidden-to-divide-by-zero. Accessed: 2026-05-25.

[40] I. E. Sutherland. Sketchpad: A man-machine graphical communication system. MIT Lincoln Laboratory Technical Report, 1963. URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-574.pdf. Reproduced Cambridge technical report (UCAM-CL-TR-574). Accessed: 2026-05-14.

[41] Wikipedia contributors. Phong reflection model — wikipedia, the free encyclopedia, . URL https://en.wikipedia.org/w/index.php?title=Phong_reflection_model&oldid=1311915114. Accessed: 2026-05-26.

[42] Wikipedia contributors. Liang–barsky algorithm — wikipedia, the free encyclopedia, . URL https://en.wikipedia.org/w/index.php?title=Liang%E2%80%93Barsky_algorithm&oldid=1321323832. Accessed: 2026-05-26.

[43] Wikipedia contributors. Sutherland–hodgman algorithm — wikipedia, the free encyclopedia, . URL https://en.wikipedia.org/w/index.php?title=Sutherland%E2%80%93Hodgman_algorithm&oldid=1321323896. Accessed: 2026-05-26.

[44] Wikipedia contributors. Camera obscura — wikipedia, the free encyclopedia, . URL https://en.wikipedia.org/w/index.php?title=Camera_obscura&oldid=1355189705. Accessed: 2026-05-20.

[45] Wikipedia contributors. Immediate mode (computer graphics), . URL https://en.wikipedia.org/wiki/Immediate_mode_(computer_graphics). Accessed: 2026-04-23.

[46] Wikipedia contributors. Larrabee (microarchitecture), . URL https://en.wikipedia.org/wiki/Larrabee_(microarchitecture). Accessed: 2026-05-14.

[47] C. Wylie, G. W. Romney, D. C. Evans, and A. Erdahl. Halftone perspective drawings by computer. In *Proceedings of the AFIPS Fall Joint Computer Conference*, volume 31, pages 49–58, 1967. URL https://apps.dtic.mil/sti/tr/pdf/AD0761965.pdf.

[48] D. Zhou, D. G. Andersen, and M. Kaminsky. Space-efficient, high-performance rank & select structures on uncompressed bit sequences. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA 2013)*, June 2013. URL https://www.istc-cc.cmu.edu/publications/papers/2013/zhou-sea2013.pdf.