Translucency Sorting in Sodium: Implementation and Analysis

Excerpts and Further Explanation

douira

27. April, 2025

Institute for Theoretical Computer Science
University of Lübeck

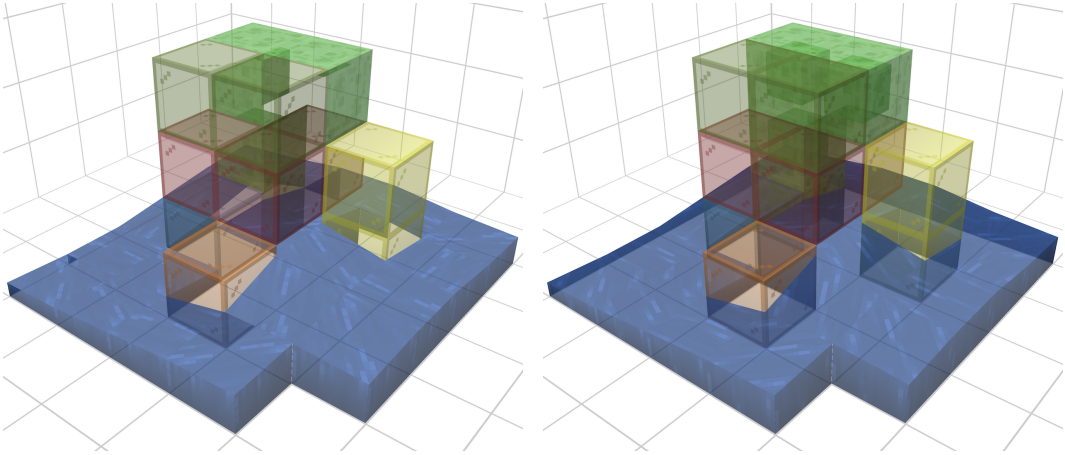
Contents

1. Motivation and Introduction
2. Rendering and Simple Translucency Sorting
3. Sorting with Graphs
4. Sorting with Partition Trees
5. Sorting the Unsortable: Quad Splitting
6. Bonus Section: The Theory of Unaligned Partitioning
7. Conclusion

Adapted from the presentation of my Master's thesis.

Motivation and Introduction

Translucent Rendering in Sodium



(a) Incorrect rendering

(b) Correct rendering

Figure 1: Many-layered translucency is a common effect in Minecraft.

Translucency in General

This is not a new problem!

- Translucency is common (but frequently avoided) in computer graphics
- Translucency (like stained glass) $\neq$ Transparency (like leaves)
- Minecraft/Sodium use ordered rendering of quads with alpha-blending
- Alternative approach: Order-independent transparency
  - Visual inaccuracy
  - Ground-truth variants slower
  - Incompatibility with targeted platforms

Rendering and Simple Translucency Sorting

Rendering Quads

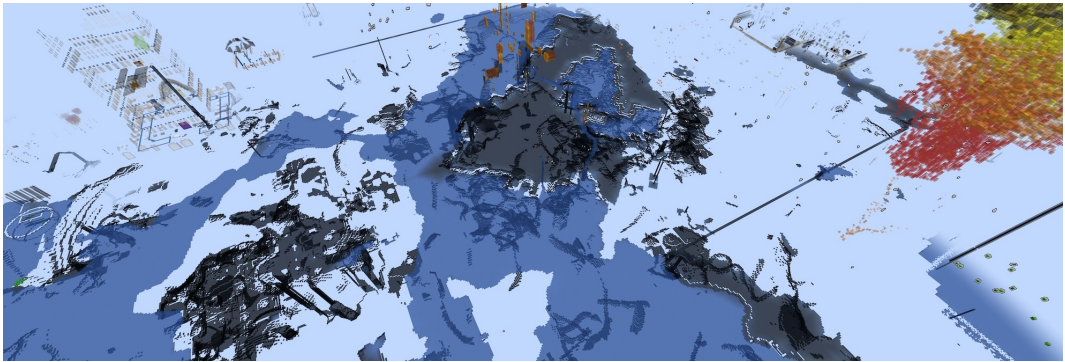


Figure 2: Only the translucents in Hermitcraft Season 8

- Blocks are rendered as multiple one-sided **quads**

→ Using **alpha blending**: interpolation with alpha factor, non-associative

→ Back-to-front ordering required for correct image (order with index buffer)

Problem (Quad-based Translucency Sorting)

- *Given is a set of* **one-sided** *quads Q, each consisting of four vertexes $q = (v_1, v_2, v_3, v_4) \in (\mathbb{R}^3)^4$ with $V_q = \{v_1, v_2, v_3, v_4\}$. Each quad's vertexes lie in the same plane $P_q = (n_q, d_q)$ with the unit* **normal vector** *$n_q \in \mathbb{R}^3$ and* **distance** *from the origin $d_q \in \mathbb{R}$ (also referred to as* **dot product***).*
- *Quad-based translucency sorting entails finding a* **permutation** *S of the quads in Q such that quads with greater depth are rendered before those with lower depth.*
- *A quad q may be* **omitted** *from S if it is facing away from the camera's position $c \in \mathbb{R}^3$ and thus is not visible.*

Unsortable Instances

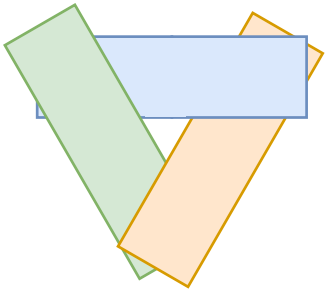


Figure 3: Cyclically overlapping geometry is unsortable.

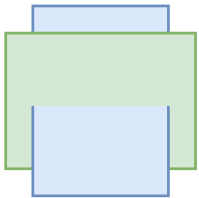
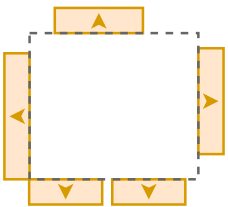


Figure 4: Intersecting geometry is unsortable.

→ Unsortable geometry requires splitting quads or per-pixel sorting

→ Quad splitting works! See Section 5



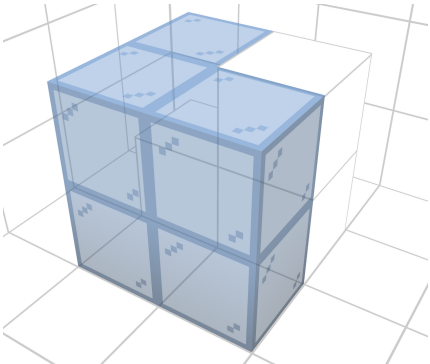


Figure 5: No sorting required if aligned to the bounding box.

Simple Special Cases ii

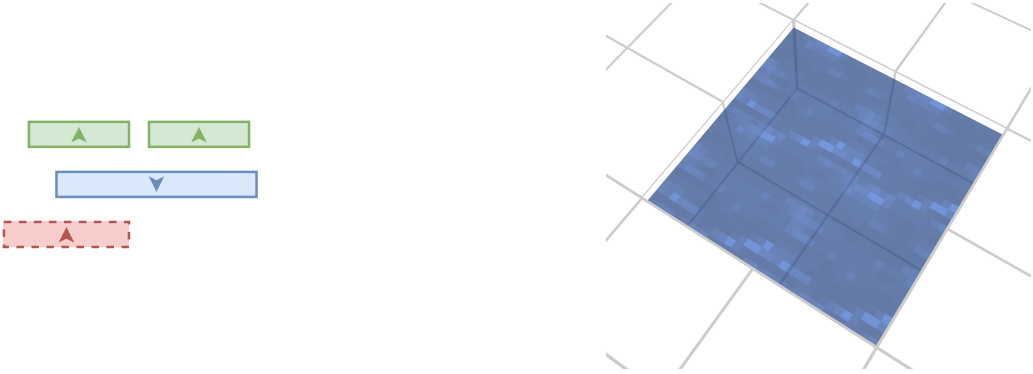
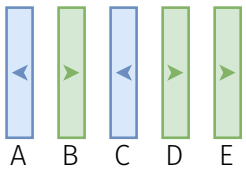


Figure 6: No sorting required if opposite facing and each only has one plane (i.e. dot product value).

Simple Special Cases iii



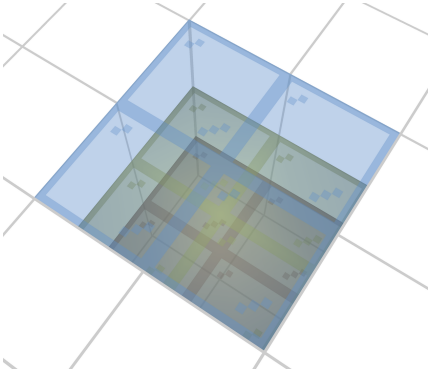
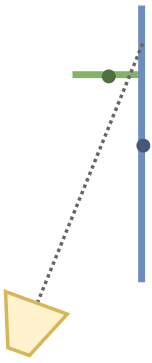


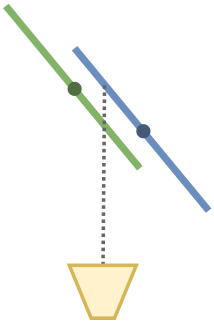
Figure 7: Quads with two opposing orientations can be sorted by dot product.
→ *Static Normal-Relative* Sorting

Sort Types

- Static: One sort order is correct for all view points
- Dynamic: Updated as camera moves and re-sorting is triggered
- Data gathered during meshing determines sort type
- Sort types:
  - None
  - Static Normal-Relative (SNR) Sort
  - Static Graph Sort
  - Dynamic Graph or Partition Tree Sort



(a) Aligned geometry



(b) Unaligned geometry

Figure 8: Center distance sorting is not correct in some situations.

Distance Sorting ii

- Sorting by distance from the camera to the center of each quad

→ Correct in many cases, but not universally, even with frequent sorting

- Invalidation is unspecific and is only estimated
- Used as a fallback when all other methods fail

Sorting with Graphs

Topological Sorting of the Visibility Graph

- Sorting based on potential visual overlap

→ One correct sort order for a set of view points: Camera polytope C

- Sorting the visibility graph G_C topologically yields a sort order

Definition (Accurate Visibility Condition)

- A quad p is visible through another quad q, referred to as q *seeing* p, if there is a view ray from a position $s \in C$ to a point $t \in S_p$ that intersects with a point in S_q. S_x is the surface of a quad $x \in Q$.
- Additionally, both quads must be facing the camera to be visible at all with $(s - t) \cdot n_p > 0$ and $(s - t) \cdot n_q > 0$.

→ Camera polytope is constrained by planes of invisible quads
- Camera exits the camera polytope when a quad becomes visible
- Triggering is implemented with a combination of interval trees and sorted lists of dot products

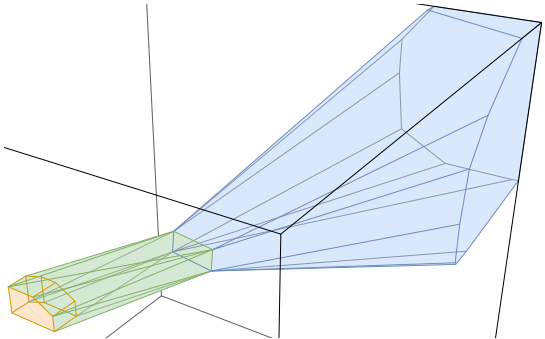


Figure 9: A quad q, the camera polytope C in orange, the visible space T_{Cq} in blue

Complexity of Visibility Graph Sorting

- The camera polytope has constant complexity with quantized normals
- Evaluating the visibility condition takes constant time
- Visibility is not transitive
- Topological sorting may need to process $O(|Q|^2)$ edges

$\rightarrow$ Finding a visible neighbor efficiently is hard

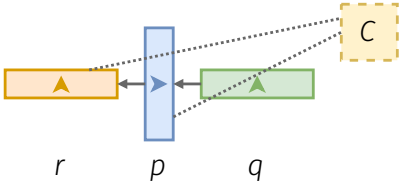


Figure 10: The arrows indicate the visibility relation, and view rays are represented by dotted lines. The relation is not transitive since q cannot see r.

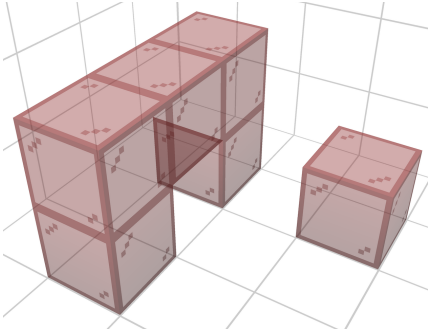


Figure 11: Many simple but non-trivial structures can be solved statically even with the reduced visibility condition.

- Static sorting for limited size instances with **reduced visibility condition**
  - Avoids solving a linear program for the accurate visibility condition
  - Assumes visibility from anywhere: $C = \mathbb{R}^3$
  - Permits testing visibility with just axis-aligned bounding boxes

The Implementation of Graph Sorting and its Limits ii

- For non-static sorting, separators alleviate some cycles
- Finding unaligned separators is infeasible

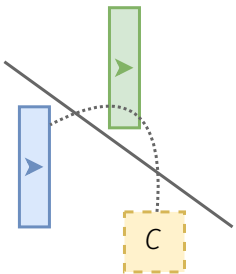


Figure 12: The separator proves that the blue quad is not visible through the green quad from any point in C. A curved view ray is impossible.

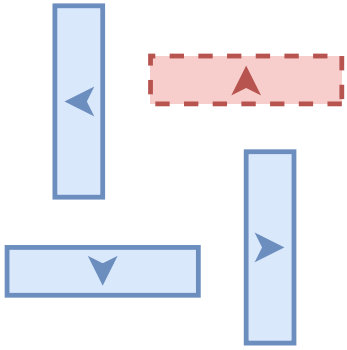


Figure 13: This scene is statically sortable with the reduced visibility condition as long as the red quad is not included.

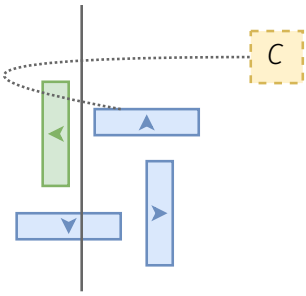


Figure 14: An aligned separator makes this scene dynamically sortable, though not statically because the camera polytope is finite.

Sorting with Partition Trees

Valid and Useful Partitions

Definition (Valid Partition)

- A partition $R = (g, d)$ of $\mathbb{R}^3$ is a plane described by normal $g \in \mathbb{R}^3$ and distance $d \in \mathbb{R}$.
- A partition is valid, also called a free cut, if no quads are intersected by the partition plane.

Definition (Useful Partition)

A partition R is useful if it partitions Q into at least two non-empty subsets.

- A tree arises from recursive multi-partitioning

→ Indexes are written for partitions in back-to-front order from the camera perspective

- It always generates a sort order in linear time
- Sorting is triggered when the camera crosses a partition plane

→ Construction in $O(n^2 \log n)$, in $O(n \log^2 n)$ if balanced

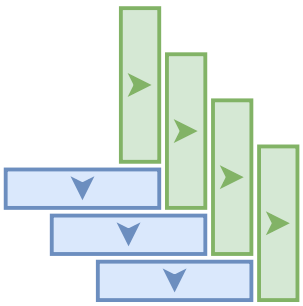


Figure 15: The tree degenerates to a list in the worst case.

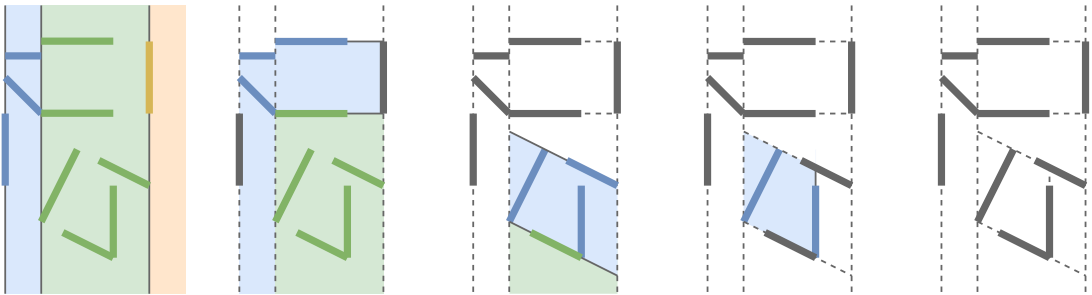


Figure 16: An example of recursive multi-partitioning on 2D lines.

Multi-Partitioning With Known Orientation

1. Projects the quads along a known orientation
2. Sorts the interval endpoints
3. Scans the list for gaps that permit partitions

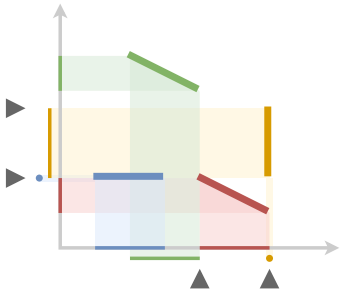


Figure 17: Projection along two axes yields two gaps for each one.

→ The orientation is known: Axis-aligned partitioning only

- Generates as many partition planes as possible
- Quads can be on the partition plane

Unpartitionable Instances

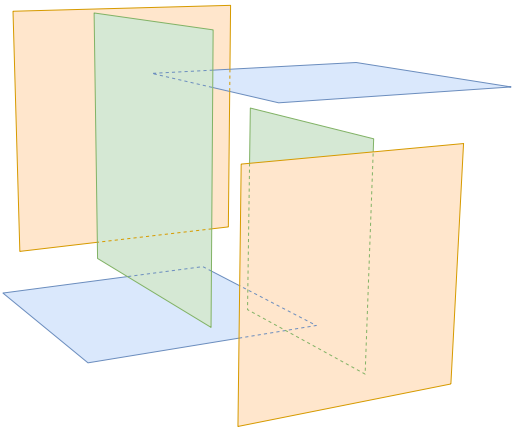
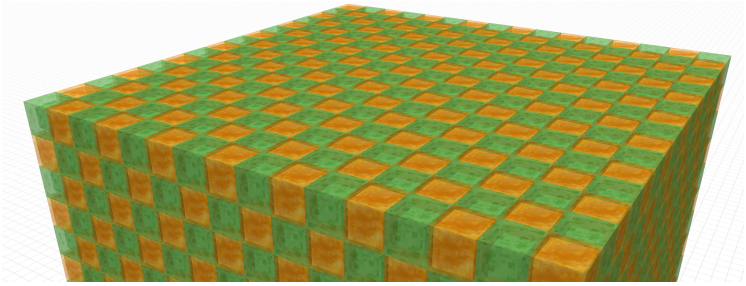


Figure 18: These aligned quads are not partitionable, even when certain quads are removed.

Figure 19: These unaligned quads are not partitionable.

Extensions and Special Cases



- Detecting special cases during partitioning allows for fewer nodes and partition planes
- Partial tree updates improve partitioning speed for large sections
- Index compression reduces memory usage very slightly
- Primary intersector detection avoids failure on intersecting geometry

Overall Implementation Performance i

- Partition tree sorting is faster than distance sorting by 60%
- Static visibility graph sorting avoids some dynamic sorting
- Detecting static special cases is important
- Distribution of sort types varies significantly with scene content

→ Distance sorting is never used in practice

Scene	No Sorting	SNR	Static Topo.	Partition Tree
Hermitcraft S9	1470	600	3113	4266
Hermitcraft S7	2793	378	1980	3339
Frozen Ocean	768	439	4597	4454
Mixed Terrain	1729	938	2053	1869
Ocean	3271	309	4345	2676

Table 1: Section counts per sort type

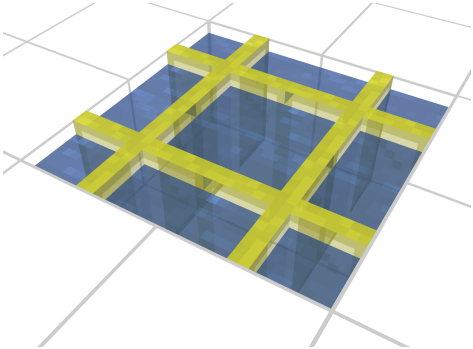
Overall Implementation Performance iii

World	Tree Build	Tree Sort	Distance Sort	Static Topo.
Hermitcraft S9	136	17	26	309
Hermitcraft S7	127	14	24	298
Frozen Ocean	135	12	23	303
Mixed Terrain	140	18	27	307
Ocean	123	27	36	316

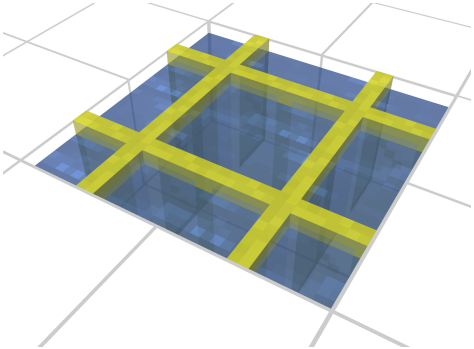
Table 2: Mean time to perform a task on a section in nanoseconds per quad

Sorting the Unsortable: Quad Splitting

Sorting the Unsortable: Quad Splitting



(a) Without quad splitting (Sodium 0.6/Vanilla)



(b) With quad splitting

→ Splitting quads fixes sorting when intersecting or unsortably arranged

- More expensive during mesh building but impact is limited
- PR#2993 is ready, merge schedule unknown. Testing builds available!

Quad Splitting in Detail

- Splitting triggered when there's no aligned partition
- Simple in concept!
  1. Pick split plane with a simple heuristic
  2. Split all quads into inside/outside fragments along split plane
  3. Generate partition tree node as usual
- Geometry amplified up to $O(n^2)$ ($\rightarrow$Not relevant in practice)
- Average case is simple
- Limit on generated geometry size prevents performance impact

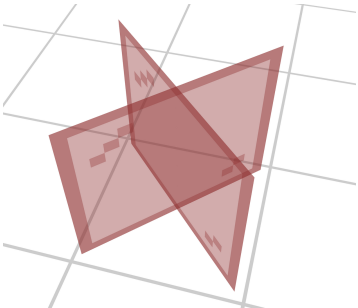
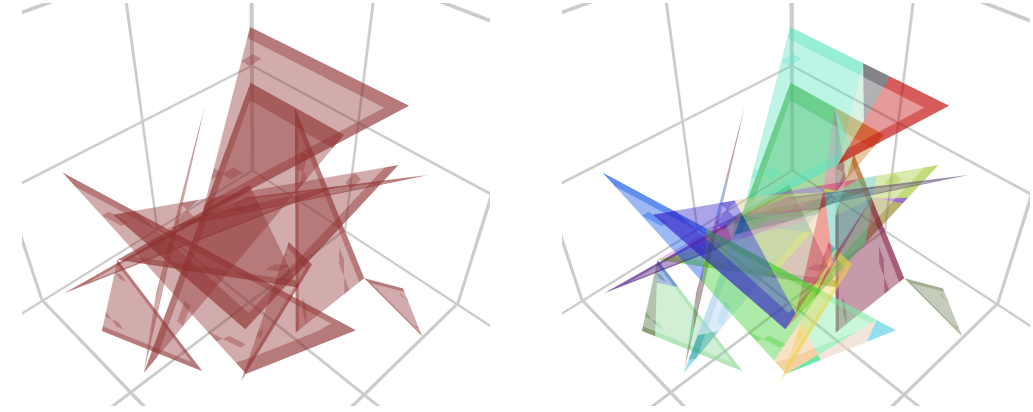


Figure 21: Two intersecting quads

Fuzzing Quad Splitting



(a) Randomly generated quads

(b) Splitting result visualized with random colors

Figure 22: Fuzzing the quad splitting system with randomly positioned quads helped me find many bugs.

Numerical Stability

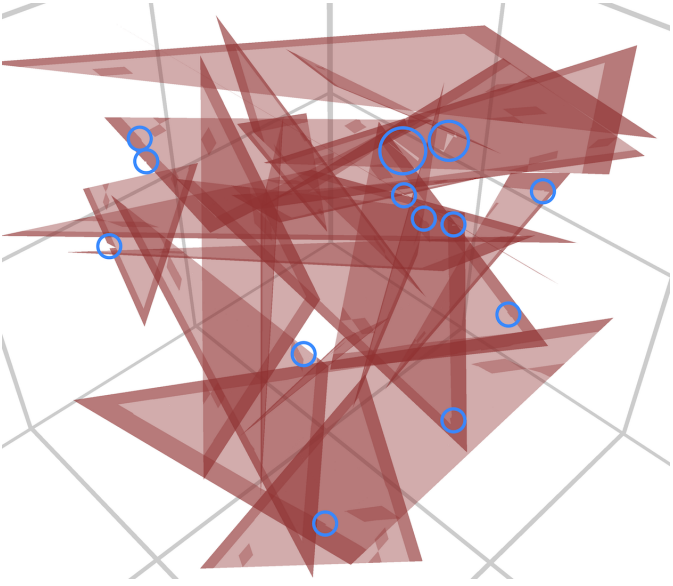


Figure 23: Errors (marked in blue) accumulate with only 32-bit floating point accuracy under repeated re-interpolation of vertex attributes.
→ The algorithm could be redesigned to not accumulate the error (difficult)

Demo

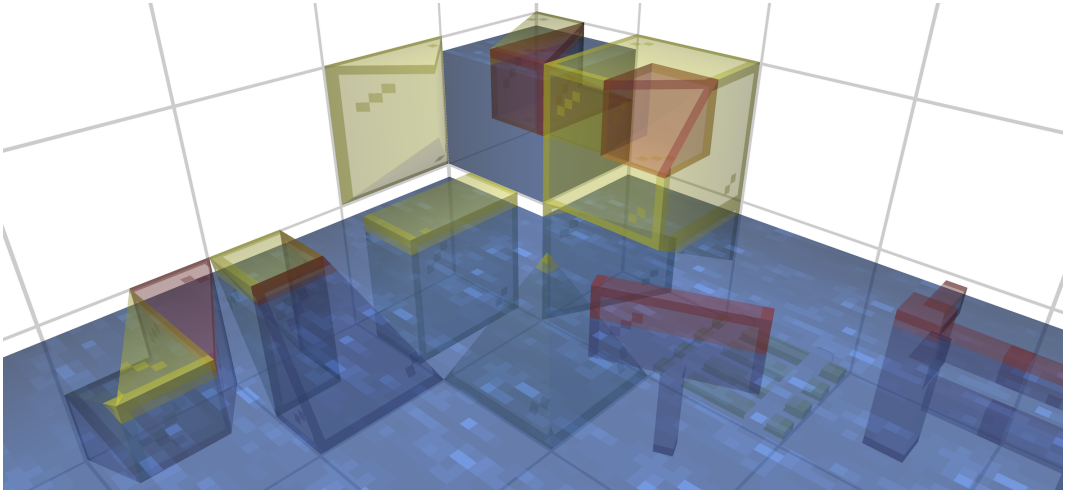
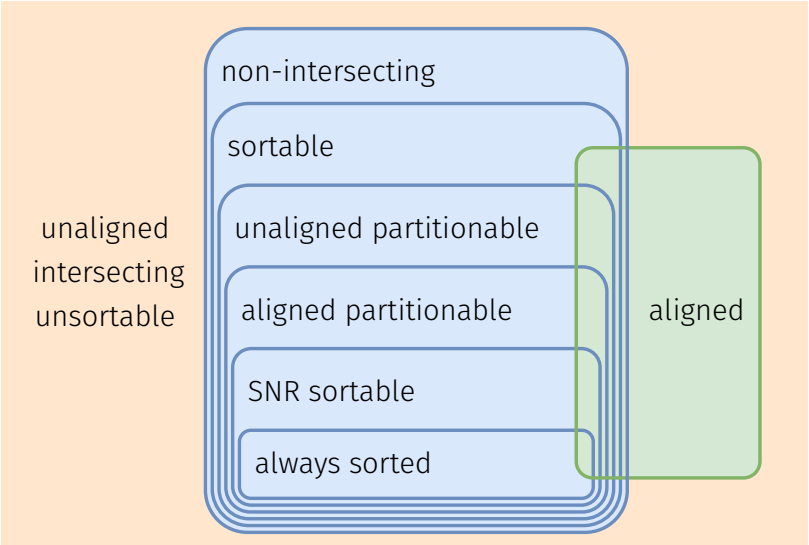


Figure 24: Works well with FramedBlocks! Thanks to XFactHD for the use case example.

Bonus Section: The Theory of Unaligned Partitioning

Overview of the Problem Space



Unaligned Partitioning

Problem (Unaligned Partitioning (UAP))

Given a set of n partitionable quads Q, does a valid and useful partition R exist, and what are its parameters?

- Only known-orientation partitioning is fast

→ Large solution space of unknown-orientation partitions

- No hard cases of UAP naturally appear in Minecraft

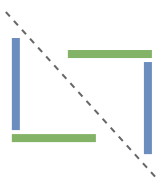


Figure 25: Aligned geometry can require an unaligned partition.

UAP in Parameter Space

- Slope-offset parametrization $y \geq ux + vz + d$ of planes yields parameter space $\mathbb{L} = \mathbb{R}^3$

$\rightarrow$ There is a linear mapping from $p \in \mathbb{R}^3$ to the parameter-space half-space containing all $R \in \mathbb{L}$ for which p is in the real-space half-space $H_R^{=}$

$\rightarrow$ Parameter-space faces correspond to real-space vertexes

- Valid partition sets and useful partition sets as expressions over parameter-space half-spaces

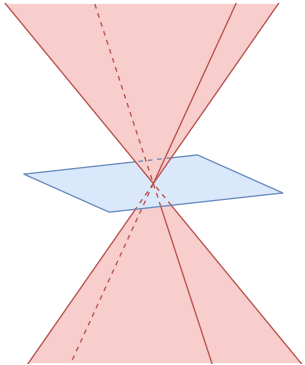
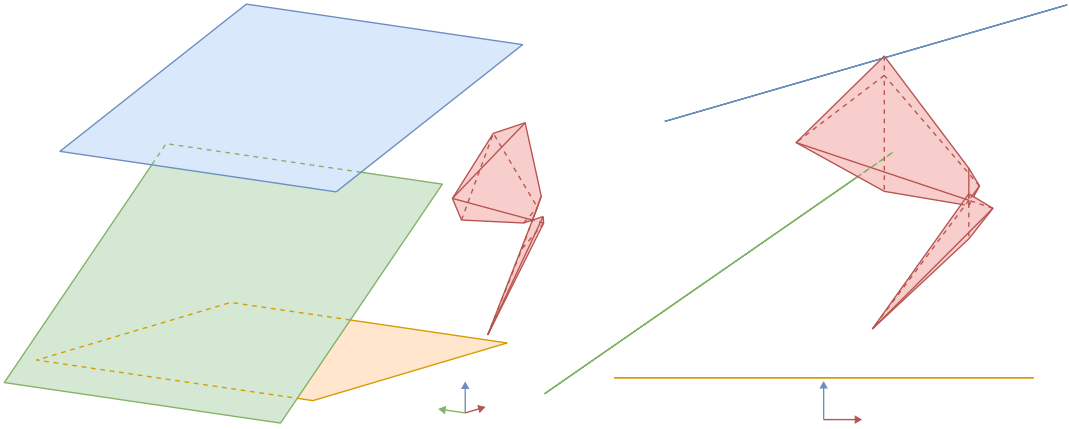


Figure 26: Valid partition set highlighted in red for blue quad q.

Useful Partitions in Parameter Space



(a) Full

(b) Right side

Figure 27: Useful partition set in parameter space highlighted in red for the three quads in real space.

Upper Bounds on UAP

- Combinatorial solution in $\Theta(n^4)$
  1. Pick all triples of vertexes to form a plane
  2. Test the partition for validity and usefulness
  3. Mitigate a global useless autopartition with an artificial vertex
- Incremental solution possible as linear constraint sets (LCS) instance, but worse time complexity
- Can also be solved with LCS-SAT

Lower Bounds on UAP

- Depending on the input constraints to UAP, it can solve LCS
- Transfer of lower bound on LCS to UAP

→ Maps half-space constraints to vertices

- Requires more quads and expensive collision avoidance if for strict formulations of UAP

Conclusion

Conclusion

- Usually translucency requires OIT
- Minecraft's specific type of geometry makes quad-based translucency sorting tractable

→ Taking advantage of many special cases and well partitionable geometry

- Ground-truth results are possible, faster than distance sorting
- Correct translucent rendering without hardware support
- Quad splitting solves all remaining cases
- Unaligned partitioning is infeasible, but polynomial

Outlook

- Is an acceleration structure for the reduced visibility condition feasible and useful?
- Can visibility graph sorting within the partition tree improve its characteristics?
- To what extent can convex partitioning, a superset of unaligned partitioning, form a bridge between partition and graph-based sorting?
- How can numerical instability be mitigated in recursive quad splitting?

Acknowledgements

I would like to thank

- my thesis advisor Dr. Sebastian Berndt, without whom this would not have been possible,
- my family and friends for supporting me during this journey,
- JellySquid and other contributors to Sodium for creating Sodium and providing both a platform and valuable feedback,
- members of the CaffeineMC discord server who helped me test and debug both the original implementation and the new quad splitting features,
- and Muzikbike for helping me perfect quad splitting by discovering numerous edge cases.

Thank you for listening!

Time for Q&A