UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Implementation and Analysis of Geometric Algorithms for Translucency Sorting in Minecraft

Implementierung und Analyse geometrischer Algorithmen für die Transluzenzsortierung in Minecraft

Masterarbeit

verfasst am
Institut für Theoretische Informatik

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
douira

ausgegeben und betreut von
Dr. Sebastian Berndt

Lübeck, den 8. April 2024

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

_____

douira

Zusammenfassung

Durch die Anordnung transluzenter Vierecke (Quads) in absteigender Tiefe entsteht ein korrektes Bild mit Alpha-Blending. Die effiziente Erzeugung einer solchen Reihenfolge für eine bewegliche Kamera stellt im Allgemeinen eine Herausforderung dar. Da Transluzenz in der Computergrafik weit verbreitet ist, wurden viele Verfahren entwickelt, aber keines ist universell optimal. Diese Arbeit implementiert die Transluzenzsortierung von Quads in Sodium, einer Minecraft-Modifikation, die sich auf Renderingperformance konzentriert. Eine korrekte Sortierung lässt sich durch das topologische Sortieren eines Sichtbarkeitsgraphen der Quads bezüglich eines Polytops von Sichtpunkten erzeugen. Die quadratische Laufzeit dieses Approximationsalgorithmus und die Gefahr des Versagens der Sortierung beschränken seine Eignung auf die statische Sortierung. In vielen Fällen können jedoch Multi-Partitionsbäume ohne Quad-Fragmentierung verwendet werden. Sie werden effizient mit einem projizierenden Intervall-Scan-Algorithmus konstruiert und ermöglichen eine 60 Prozent schnellere Erzeugung der Sortierung im Vergleich zur Sortierung von Quads nach Entfernung. Diese Techniken verbessern die visuelle Korrektheit und bieten ein leistungsfähiges System für die Transluzenzsortierung. Die axial unausgerichtete Partitionierung ist zwar für Minecraft nicht notwendig, nutzt aber die partitionierbare Geometrie vollständig aus und eignet sich für die Interpretation im Parameterraum. Es gibt eine polynomielle obere Schranke und Möglichkeiten der Übertragung von unteren Schranken aus der linearen Optimierung.

Abstract

Ordering translucent quadrilaterals (quads) by descending depth produces a correct image with alpha-blended translucency. Generating such an order efficiently with a moving camera is challenging in general. Since translucency is common in computer graphics, a multitude of techniques have been developed for it, but none is universally optimal. This work implements quad-based translucency sorting in Sodium, a Minecraft modification focused on rendering performance. A sort order is obtained by topologically sorting a visibility graph over the quads with respect to a polytope of view points. However, this algorithm's quadratic runtime and potential for sort failure when being approximated limit its suitability to static sorting. Axis-aligned multi-partition trees without quad fragmentation can be used instead. They are efficiently built with a projected interval scanning algorithm and generate dynamic sort orders 60 percent faster than sorting quads by distance. Together, these techniques improve visual correctness and provide a performant translucency sorting system. Unaligned partitioning, although not necessary for Minecraft, fully exploits partitionable geometry and lends itself to interpretation in parameter space. It is given a polynomial upper bound and options for transferring a lower bound from linear constraint solving.

Acknowledgements

This endeavor would not have been possible without Dr. Sebastian Berndt's encouragement to pursue this topic and his commitment to frequently providing helpful feedback. Our weekly meetings were invaluable to maintaining focus and rapidly developing my ideas.

I am deeply indebted to JellySquid and other contributors to Sodium who provided the foundations for this work and made the project possible in the first place. Their feedback and help with technical aspects were important to the success of my implementation.

Lastly, I am also grateful to my parents and friends for their support and encouragement as I was writing this thesis. Special thanks go to my mother whose tireless proofreading was indispensable.

Contents

1

Introduction

Translucency is a fixture in modern computer graphics as both a way to replicate real-world translucent objects and to give the viewer an overview of an entire scene's content by letting them see through otherwise opaque surfaces. In the block-based game *Minecraft*[1] (Mojang Studios, 2009) translucent rendering is used for blocks of stained glass and water, amongst other examples. Sodium (JellySquid et al., 2020) is a modification (mod) for Minecraft that improves the rendering performance of the game in many ways, but critically makes it significantly more convenient to modify the rendering pipeline by being open source and replacing the existing rendering engine. Until contributions by this work, Sodium did not implement correct translucent rendering. This is easily noticeable in situations where many translucent blocks are visually overlaid, especially to users accustomed to a largely correct image.

In games like Minecraft, the world consisting of blocks and derived shapes is typically rendered by turning it into a mesh made up of **quadrilaterals (quads)**. Each quad has a position in 3D space and other data necessary for texturing and shading. The order in which the quads are sent to the GPU for rendering does not matter if they are opaque and cannot be seen through. Translucent quads however, must be ordered such that new quads blended onto already rendered quads are closer to the camera, which is the first-person view controlled by the player. The rendering must thus order the quads by depth for a correct result.

This indexed rendering approach, which reorders quads on the CPU by generating an index buffer, is preferred by Sodium for its simplicity, compatibility with older graphics platforms, and good performance. Sorting quads by distance on the GPU is also possible, but has either worse performance or yields incorrect results and is not supported by all platforms. **Order-independent transparency (OIT)** is a conceptually different approach where, through special blending techniques, the ordering of the quads does not influence the resulting image. The approximations used by its various implementations either do not yield a ground-truth result, which is especially noticeable in Minecraft, or they have a performance impact and worse compatibility with the targeted platforms.

[1] This work was not approved by and is not associated with Mojang or Microsoft.

1.1 Contributions of this Thesis

This work presents a solution for translucency sorting in the context of Minecraft and investigates the complexity of the related unaligned partitioning problem. It adds indexed translucent rendering and efficient quad-based translucency sorting to Sodium by combining heuristics that enable simplifications of typical special cases and two approaches that perform sorting with different performance characteristics, namely topological visibility graph sorting and partition tree sorting.

After handling special cases that permit faster sorting methods, a sort order is generated statically or dynamically. While dynamic sorting requires updates to the sort order as the camera moves, static sorting does not until the geometry itself changes. A visibility condition is defined under which one quad can be seen through another from the camera's point of view, yielding a visibility graph for the quads in the scene. Topologically sorting this graph yields a correct sort order of the quads, if one exists. However, the implementation uses a less complex reduced visibility condition, which results in better performance, albeit at the cost of sometimes not finding a sort order at all. For dynamic sorting, the set of points that share a correct sort order forms a camera polytope bounded by a set of trigger planes derived from the scene's geometry. This allows the sort order to be accurately invalidated, as opposed to the immediate invalidation that occurs when sorting quads by distance, which is what the existing implementation in Minecraft does.

Topological graph sorting is commonly only used to generate static sort orders since in nearly all dynamic cases partition tree sorting is possible. The set of quads is partitioned using multiple axis-aligned planes to construct a partition tree from which a sort order can be derived for a given camera position in linear time. This sort order is correct for an entire camera polytope bounded by the partition planes. Using a combination of interval trees and the fact that with quantized normals there are only a constant number of quad orientations, efficiently dispatching tasks to perform dynamic sorting is possible for camera movement in each frame. Partitioning of quads is performed by scanning a sorted list of the interval endpoints of quads projected along the partitioning direction and adding a partition for every discovered gap. Nodes in the tree are sometimes able to be re-used when the geometry is only partially updated. Intersecting geometry is not sortable, but in the rare case that it does occur in Minecraft, detecting and nonetheless producing a partition tree allows the visual result to be more accurate than what is possible with distance sorting. Unaligned, i.e. not axis-aligned, partitions are not searched for since they have a far larger solution space.

In practice, dynamic sorting can be typically avoided for most or at least a significant fraction of the geometry in representative test cases. Partition tree sorting outperforms distance sorting by around 60 percent and does not need to happen as frequently due to its known invalidation behavior. The partition tree is built in around 132 nanoseconds per quad, and a sort order is generated from it in 18 nanoseconds per quad on average. The implementation in Java encompasses over 6600 changed lines across 86 files and can be found in two pull requests to the Sodium repository[2] that are part of the version 0.6 release.

An upper bound on the unaligned partitioning problem is given by a combinatorial

[2] PRs 2016 and 2352. A copy of the implementation is supplied with the digital submission of this work.

algorithm that picks three vertexes and checks the resulting partition for validity and usefulness. Partitions are defined as valid if they do not intersect with geometry and useful if they split it into multiple non-empty subsets. Parameterizing the partition planes using a linear plane equation lets useful partitions be represented by geometric objects in the resulting parameter space. An alternative solution is given by reducing the problem to a polynomial number of linear constraint sets. Various possibilities for transferring a lower bound are given by the reduction of linear constraint solving to unaligned partitioning, with the difficulty of the reduction depending on how strict the constraints on the resulting instance are.

1.2 Related Work

The painter's algorithm draws objects back to front, yielding correct translucent rendering as each quad can simply be blended with the existing image. However, implementing it is difficult due to objects that overlap with each other in a cycle or intersect. Eck describes the hidden surface problem and the typical approach using a depth buffer in (Eck, 2023, Section 3.1.4). Correctly implementing the painter's algorithm is essentially the entire problem of translucency sorting, as using a depth test with unordered geometry does not yield correct results due to blending. While Minecraft does not turn off the depth buffer for translucent rendering as described in (Eck, 2023, Section 7.4.1), which results in quads that are rendered out of order missing, doing so would not correctly blend the now visible objects. That section also mentions that sorting primitives, such as quads, by depth, which is distinct from distance, is necessary to achieve correct translucent rendering.

Hidden surface removal, which is concerned with determining which geometry is visible to the camera and not occluded by other closer geometry, is related to translucency sorting because approaches for the former also often yield results for the latter if adapted accordingly. A discrete image-space method that finds and traverses intersections of polygons can generate a translucent rendering order if, instead of culling quads, their ordering is recorded (Weiler and Atherton, 1977). Dorward gives an overview of hidden surface removal approaches in (Dorward, 1994), some of which can be modified to produce useful orderings for translucent rendering. This includes the binary partition tree approach (Paterson and Yao, 1990), which easily produces a useful quad ordering under the condition that objects may be fragmented. More recent developments obtain results in partition complexity and cases with special constraints (Tóth, 2005). The research in this area largely focuses on partition complexity under the assumption that objects can be fragmented along partition planes and rendered as separate parts, which is not the case in the context of this work.

In (Mulmuley, 1991) a graph-based approach vaguely similar to the one discussed here is presented, but for hidden surface removal and not translucency sorting. The construction of a graph is presented that concerns itself with the intersecting edges of polygons and its change over time as the camera, i.e. view point, moves. As in this work, it exploits inter-frame coherency since the result only changes when the camera has moved enough for the current graph not to be isomorphic to the previous one. For predefined

camera paths, a sliding window over an index buffer can be generated from the compact representation of a visibility dependency graph (Weber and Stamminger, 2016). Weber and Stamminger produce this graph ahead of time by rendering triangles with depth-peeling, an accurate OIT rendering method, and recording which triangles overlap in each pixel. Topological sorting between objects for translucent rendering is also employed in (Govindaraju et al., 2005) with edges between them being tested by rasterizing on the GPU.

It should be noted that throughout the literature, the terms translucency and transparency are not used consistently and sometimes interchangeably. Typically, transparency refers to a material that transmits light without scattering, but sometimes absorption, i.e. a tint, is included in its definition. Translucency covers anything between any particular definition of transparency and a fully opaque material. In this work, the term translucency is used in line with the definition in (Gerardin et al., 2019), where transparency means fully transparent and translucency includes tinting effects, amongst others. While the established term "order-independent transparency" conflicts with this definition, it is not modified to avoid confusion.

Alternatives to sorting objects are GPU-based OIT or ray tracing. Order-independent transparency largely depends on modern GPU features and can have a greater visual or performance impact compared to sorting objects. However, for most applications, its results and performance characteristics are satisfactory. Concrete per-pixel OIT methods range in sophistication and make different tradeoffs between fidelity and performance. One of the earliest techniques that produces a ground-truth result is depth-peeling, which makes multiple passes over the scene, each time rendering one layer of translucent pixels onto the last (Everitt, 2001). Later approaches improve performance or visuals, sometimes by approximation or by making use of advances in hardware (Barta et al., 2011)(McGuire and Bavoil, 2013)(Münstermann et al., 2018)(Tsopouridis, Fudos, and Vasilakis, 2022). With ray tracing, translucent rendering is performed with knowledge of the geometry that rays intersected with as the scene is traversed (Zhang, 2021). Here too, performant and accurate translucency is not yet a fully solved problem.

1.3 Structure of this Thesis

The structure of this work orients itself around the practical aspects of solving translucency sorting, with additional exploration of the theoretical concepts as they become relevant. In Chapter 2, a general introduction of the problem and solutions to some simple special cases are given. This includes the translucency sorting problem and why distance sorting, while providing a good approximation, does not fully solve it. Regarding the implementation, the separation between dynamic and static sorting is also introduced.

The graph-based translucency sorting in Chapter 3 resolves the problem of immediate invalidation that distance sorting has by modeling the visibility between quads directly. The visibility condition describes the core problem of ordering visually overlapping geometry underpinning indexed translucent rendering. The reduced visibility condition and its optimizations are an important part of the implementation, which is clear considering the effort that goes into mitigating some of the errors it introduces as

an approximation. An important part of this chapter is the plane-based triggering mechanism and the camera polytope, in which a sort order is guaranteed to remain correct. Separators are used to avoid some cycles in the graph by proving non-visibility between two quads depending on the camera position.

Similar to separators, in Chapter 4, partitions are used to build a tree over the quads and order them based on the camera position. This is the same ordering concept as with a separator, where a plane provides an ordering of objects relative to the camera. The implementation of partition trees uses axis-aligned partitions to efficiently build a tree and generate a sort order. Performance figures are presented alongside special handling for intersecting geometry and partial tree updates.

As opposed to aligned partitions, unaligned partitions are harder to find. Chapter 5 discusses the theory of the unaligned partitioning problem. First, an overview of the problem space is given, in which the alignment of geometry notably does not necessarily relate to the alignment of the partitions. Unaligned partitioning can be described as a series of operations on sets of partitions, for which the corresponding geometry in parameter space is visualized in multiple figures. Following this, two algorithms for an upper bound are given, and the chapter is concluded with a discussion on obtaining lower bounds through a reduction. While this chapter can largely be read independently, the others each depend on previously established terms and concepts.

2

Preliminaries

The sorting process can be simplified when dealing with special cases, which lets the implementation avoid the expensive techniques that are applied otherwise. Sometimes no sorting needs to be performed at all if any sort order is correct. Only as much work as necessary is performed if sections are categorized into sort types of increasing complexity. Distance sorting is only used as a fallback when other methods fail because it sometimes produces errors. Nonetheless, triggering it based on the total accumulated camera movement avoids iterating over all sections in every frame to check if their current sort order has likely been invalidated. The following graphics and algorithms use Minecraft's right-handed Y-up coordinate system, but everything can be easily transformed into any other coordinate system by simple rotation and/or mirroring.

2.1 Rendering in Minecraft

The following description of the rendering process is based on how Sodium works, but the base game functions similarly in concept. Minecraft's world is rendered as quads, which in turn are each rendered as a pair of triangles by the GPU. For the purposes of rendering, storage, and simulation, Minecraft partitions the world into **sections** of 16×16×16 blocks. The geometry data of a section, which may also include non-cuboid objects that consist, in part, of slanted or other irregular quads, is written into buffers and uploaded to the GPU for rendering. When the world is modified by the player or other gameplay elements, the affected sections are re-meshed and uploaded again.

To render the uploaded geometry, a second buffer containing index offsets into the list of quads determines the order in which the geometry is rendered. Even though the GPU only renders triangles, the entire pipeline works with quads and turns them into two triangles late in the process by specifying an appropriate indexing through the **index buffer**. The geometry's rendering order is important both for performance and translucent effects. When pixels for opaque geometry are rendered in positions on the screen where other pixels that are closer to the camera have already been rendered, the GPU can stop processing them early since they cannot become visible. While the ordering of geometry within a section is controlled by the index buffer, the ordering between sections is controlled by the order in which commands are sent to the GPU to render them. This

section ordering is not subject of this work and is already handled correctly by Sodium.

Translucency

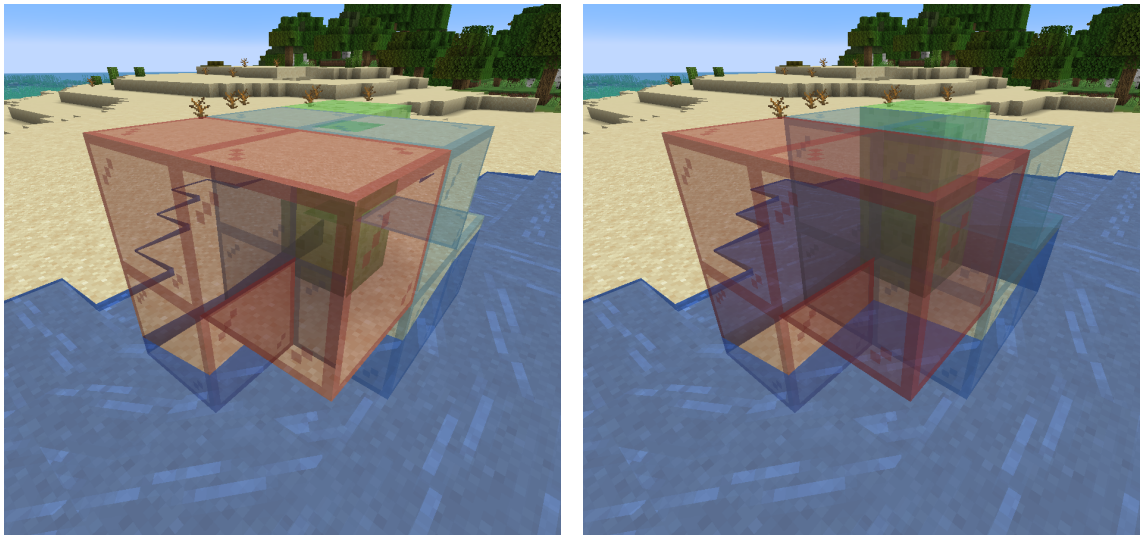
Translucent geometry lets the player see objects through other objects, such as glass panes and water. When rendering translucent geometry, visually overlapping quads must be rendered in back-to-front order, since otherwise the GPU will not render pixels farther away than those already in the image. Furthermore, to correctly display layers of translucent geometry, closer pixels must be blended with those behind them in the right order to achieve visually accurate results. Minecraft uses alpha blending, which interpolates between the existing pixel's color and the new color based on the texture's alpha, i.e. absorption strength, value. Such blending is not physically correct, but nonetheless, it produces a distinct style.

Translucent and opaque geometry is rendered in separate passes, each with their own index buffers. The rendering order of opaque geometry does not matter beyond the very small performance benefit of reducing overdraw, which is when pixels are discarded and not rendered into the position of an existing closer pixel. For the rendering of opaque geometry, Sodium takes advantage of the fact that the order is largely irrelevant and re-uses the same trivial index buffer for all sections. However, prior to this work, translucency sorting had not yet been implemented in Sodium, and always rendering the geometry with the same shared index buffer resulted in artifacts as shown in Figure 2.1 on the following page. Thus, the goal is to compute a correct quad ordering and to update it as the camera moves around and the world is modified.

Most sections in the world contain none or only few translucent quads, but this number can grow to around 2,000 in normal cases. Uncommonly, it can get extremely large in rare and player-constructed cases (50,000 or more). Usually up to around 100,000 sections are loaded for being close enough to the player, of which around 5,000 are visible at any given time. However, uncommon edge cases can inflate these numbers significantly. As the player moves through the world, sections that leave the rendered area around the player are unloaded from memory, and new sections are loaded as they become visible to the camera. When a loaded section becomes visible to the player for the first time, a meshing task is dispatched that generates its geometry data and its initial, and possibly final, index buffer.

2.2 Translucency Sorting

Problem 2.1 (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 $d_q \in \mathbb{R}$. 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. In other words, the resulting image should be correct. 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.



(a) Incorrect rendering (b) Correct rendering

Figure 2.1: These two in-game screenshots compare rendering with (b) and without (a) translucency sorting. The scene contains various translucent blocks, including green slime blocks, stained glass of different colors, and flowing water. The blocks have axis-aligned geometry, whereas the water has slanted surfaces. In Subfigure (a), a lack of translucency sorting presents itself as missing water and block faces when seen through other translucent surfaces.

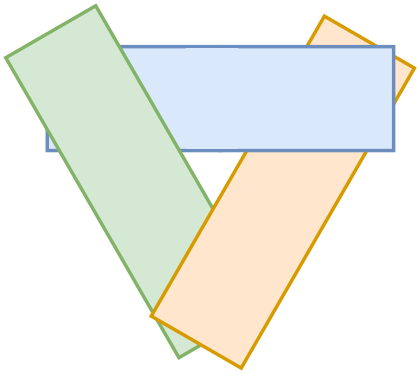


Figure 2.2: No ordering can correctly render a cycle of three overlapping translucent quads arranged like this. Per-pixel sorting or fragmentation of the quads is necessary here.

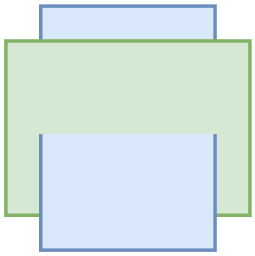


Figure 2.3: Quad sorting cannot correctly handle intersecting geometry, as every possible sort order is wrong. Some part of the quad rendered first is always wrongly rendered beneath the quad rendered second.

Correct translucent rendering can be achieved by sorting entire quads based on their geometry to generate an index buffer. The permutation of quads that a solution to Problem 2.1 consists of is turned into an indexing of the vertexes and written to the index buffer. Sorting whole quads means overlapping cycles and intersecting geometry are unsortable, as illustrated in Figure 2.2 on the previous page and Figure 2.3 respectively.

Omitting an invisible quad q from S with $(c - v_1) \cdot n_q \leq 0$ is possible. Alternatively, the quad center $c_q = \frac{1}{4} \sum_{i=1}^{4} v_i$, which is simply the mean of the vertexes, can be used for this instead of a vertex because the quad is assumed to be planar and thus all points on its surface have the same dot product. The problem is independent of the camera's field of view and rotation. In fact, rotating the camera never changes the validity of a translucency sorting result. While the projected depth of the rendered pixels may change with camera orientation, their correct layering does not. A formulation that only requires quads in the camera's field of view to be sorted is possible, but not particularly useful when the camera can see all quads or change orientation frequently.

Order-independent transparency and per-pixel sorting techniques are outlined in Section 1.2 but are not subject of this work.

Optional Constraints

The geometry for an instance of the translucency sorting problem in Minecraft fully lies within its section, and on average, a majority of the quads are axis-aligned. Though intersections do rarely occur in Minecraft's geometry, it does not implement per-pixel sorting or other techniques to avoid artifacts that arise from such a case. Instead, Minecraft's implementation performs center distance sorting and accepts any artifacts that may occur. Nevertheless, improvements in accuracy are permissible, assuming they do not deviate too far from the expected visual style.

The following approaches will always assume the geometry does not intersect and is bounded within a section, except to handle exceptions to this rule. Two coplanar quads facing in opposite directions are not considered to be intersecting. In the event that a rare intersecting case is encountered, a simple heuristic that approximates or improves upon the existing solution in Minecraft is sufficient, but the optimal solution achievable with different rendering paradigms is not a design goal.

Degenerate quads where two of the vertexes are identical can usually be treated like

regular quads. The centers of these triangle-shaped quads are calculated only from the mean position of the three unique vertexes. However, when the vertexes of a quad are not all on the same plane, the quad can still be rendered, but it appears bent when viewed from specific in-plane angles. Such quads can be simply decomposed into two triangles and treated as separate degenerate quads. This special case is not implemented though, as it is not practically relevant for visual correctness.

2.3 Sort Types and Static Sorting Heuristics

A number of special cases lend themselves well to drastic optimizations if they can be detected with little additional effort. Differentiating the various levels of constraints that make sorting easier results in a number of sort types. They can be broadly categorized into two types: static and dynamic. **Static sorting** performs the sorting right after meshing and then never again. The static sort order can be uploaded to the GPU once and only needs to be changed when the geometry itself changes. **Dynamic sorting** on the other hand, requires the sort order to be updated sometimes as the camera moves. The simplest of the static sort types are presented in the following subsection, while the more complex sort types follow in the next chapters.

When the triggering mechanism determines that a new sort order is required, it dispatches a sorting task that computes the new sort order and writes it into the index buffer. Sorting and meshing tasks take different amounts of effort to perform with the former usually being much faster. To avoid starvation, the task scheduler ensures some of both types of work is performed in each frame with any remaining budget getting filled up with any pending tasks.

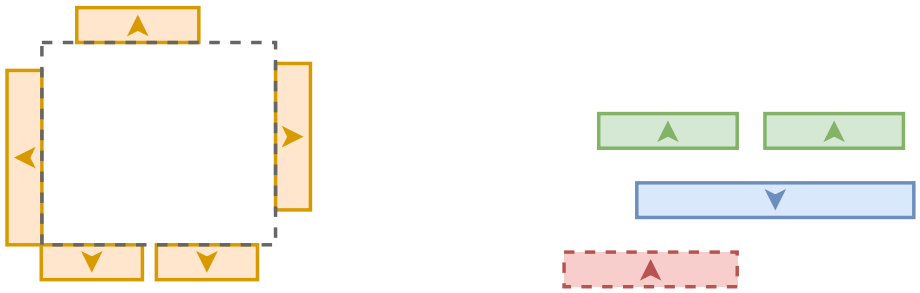
Static Sort Types

Taking the dot product of a quad's normal n and one of its vertexes v_1 yields the dot product $d = n \cdot v_1$. Conceptually, the dot product between a vector and a point indicates how far along that vector the point lies. It performs a projection of a point onto the vector.

During meshing, each translucent quad is collected, and data about the mesh as a whole is accumulated. This includes the axis-aligned bounding box, whether there are unaligned quads, which of the six directions have aligned quads, two recent unaligned quad normals and two of their dot products, and the maximum and minimum dot products of the aligned quads. There are six aligned facings, one for each axis-aligned direction, and one unaligned facing, which is always rendered. By default, Sodium does not render aligned quads with a certain aligned facing if it is impossible for geometry with that facing in a section to be visible from the current camera position. This **block face culling** is important for rendering performance but requires the geometry for each facing to be rendered independently. It is disabled when translucency sorting interleaves quads with different facings in the sort order.

The simplest static sort type is the one that requires no sorting at all. Quads are simply rendered in any order as there is no way to see one quad through another. This special case can happen when all the geometry is aligned onto the surface of its bounding box and is facing outward. In a cuboid as shown in Figure 2.4a on the next page no outside face can

be seen through another face, so no sorting is required. In fact, this concept generalizes to outward-facing quads on the surface of any convex polytope, though implementing the heuristic to take unaligned normals into account is unnecessarily complex. Sorting is also not required if there are only two opposing normals and for each there is only one dot product value, as seen in Figure 2.4b.



(a) Bounding box aligned, outwards facing

(b) Opposite facing, one dot product value

Figure 2.4: In these simple 2D scenes, quads are represented by rectangles with their facing indicated by an arrow. This directly translates to 3D geometry where all quads have the same Y (up) coordinate value and the camera is looking down on the quads edge-on. Since in Subfigure (a) all quads lie on the dashed bounding box and are facing outwards, no quad can be seen through another, and thereby this scene does not need to be sorted. In Subfigure (b), no sorting is required unless the red quad is present, as there are two opposing normals with only one dot product value each.

Sorting is required, though it can be static, in the less restrictive special case consisting of two opposing normals but with any number of dot product values. However, the quads of each normal can be sorted independently by their dot product. Figure 2.5 gives an example of this sort type, referred to as **static normal-relative (SNR)** sorting. Both SNR sorting and no sorting are compatible with block face culling as they operate on each facing independently. All following sort types require block face culling to be disabled for the section's translucent geometry.

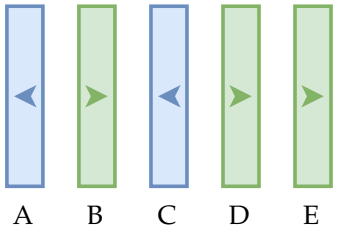


Figure 2.5: Since in this special case quads with different normals cannot be seen through each other, grouping quads by normal and sorting them by dot product suffices. The orders CABDE, CBDAE, and BDECA are valid because the sorted groups CA and BDE can be arbitrarily interleaved.

2.4 Distance Sorting

Sorting quads by euclidean distance from the camera position produces a reasonable approximation of a correct ordering, with exceptions shown in Figure 2.6. Even using other distance metrics, such as the maximum or minimum of the vertexes' distances in a quad for example, breaks down in situations like the one in Figure 2.6b. Especially with axis-aligned geometry, artifacts occur when quads are of very different sizes or intersect, but they are not very noticeable under most common circumstances. Minecraft's existing implementation uses a simple approach that performs distance sorting on all translucent geometry-containing sections frequently, but usually not every frame.

(a) Aligned geometry

(b) Unaligned geometry

Figure 2.6: Center distance sorting is not correct in specific situations where quads are visually ordered differently than their centers are. The camera's view ray passes through the two quads in a different order than sorting their centers by distance would result in.

The primary issue with center distance sorting is the rapid invalidation of its results. If the quads have been sorted for camera position c_1, then this sort order may become wrong immediately after a camera movement $m = (c_1, c_2)$ from c_1 to c_2 if $c_1 \neq c_2$. How quickly a distance sorting result becomes invalid and produces visual artifacts depends on the specific geometry. Both geometry that is far apart and very close together can invalidate distance sorting results very quickly. One such example, which also occurs frequently in practice, is given in Figure 2.7 on the following page. Determining the exact volume of camera positions where the distance sort remains valid is challenging. Instead, an approximation that triggers distance sorting based on predetermined thresholds works well in practice as a fallback solution.

Distance Sort Triggering

A simple way of triggering distance sorting is based on the magnitude $\Delta_m^d = \|c_2 - c_1\|$ of a camera movement m. When this value reaches a certain threshold $\Delta_{\max}^d$, all sections that need distance sorting are triggered at once. Such a simple approach, however, is wasteful since sections that are far away do not need sorting as often as those that are close to the camera. Furthermore, the camera may have a circular motion that does not

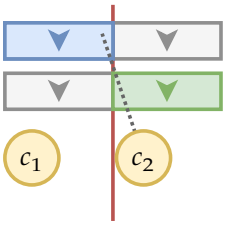


Figure 2.7: Distance sorting at camera position c_1 orders the green quad before the blue quad as its center is farther away, but this is wrong when the camera moves to c_2. There, the blue quad must be rendered before the green quad. The red line indicates the invalidation boundary for the distance sorted order.

place it farther away than $\Delta^d_{\max}$ from some starting position at any point, which should not trigger sorting.

For close sections, this problem can be solved by keeping track of the position c_s the camera was at when each section was last sorted and only sorting when the new camera position c is far enough away from it, i.e. $\|c_s - c\| \geq \Delta^d_{\max}$. The magnitudes of all camera movements are accumulated into $d = d_{n-1} + \Delta^d_m$. To avoid checking all sections each time the camera moves, the cumulative camera distance $\hat{d}_i$, at which a particular section i may require sorting, along with a reference to the section, is entered into a heap. When the camera moves, all entries less than or equal to the updated cumulative camera distance d are queried and processed as follows: The sections for which the camera has actually moved far enough with $d \geq \hat{d}_i$ are triggered while the rest are re-entered into the heap with a new lower bound $\hat{d}_i = d + \Delta^d_{\max} - \|c_s - c\|$ on the required camera movement for triggering to be necessary. In an effort to prevent a curved camera path from generating a high rate of unsuccessful entries in the heap, sections queried from the heap are triggered even if the distance from the last sort position is slightly less than the threshold, making the accurate condition $\|c_s - c\| \geq \gamma \cdot \Delta^d_{\max}$ with $0 < \gamma < 1$ chosen as 0.9 in practice. In Figure 2.8 on the next page an example of a section that is re-entered into the heap is shown.

For sections that are far enough away, the same scheme can be performed on the angle θ_m between the vectors $s - c_s$ and $s - c$ for a section at $s \in \mathbb{R}^3$. Through trigonometry, a lower bound on cumulative camera movement can be calculated based on the last sorting position c_s, the camera's angle θ_m, and the threshold angle $\theta_{\max}$. Analogously to distance-based triggering, the early trigger factor γ reduces churn in the heap.

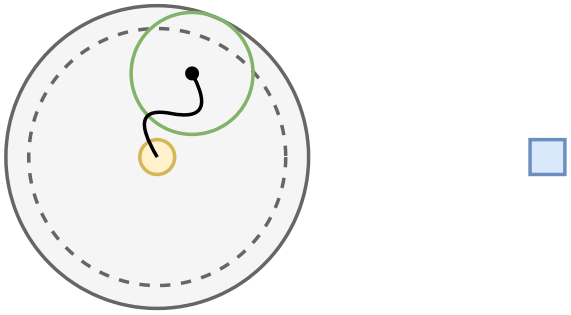


Figure 2.8: A section is indicated as the square in blue. The yellow circle represents the point c_s with solid and dashed circles for $\Delta^d_{\max}$ and $\gamma \cdot \Delta^d_{\max}$ surrounding it, respectively. The camera movement starts at c_s and its length exceeds $\hat{d}$, but it has not moved farther than $\gamma \cdot \Delta^d_{\max}$ away from c_s. The section is re-entered with a new $\hat{d}$ represented by the green circle around the new camera position. The camera cannot move farther than $\Delta^d_{\max}$ away from c_s without leaving the green circle.

3

Sorting With Graphs

At its core, translucency sorting pertains to the visibility of quads through each other and the correct ordering between quads that can visually overlap. A visibility condition that reflects this ordering relationship, interpreted as a graph between quads, directly models what it means for a sort order to be visually correct. The accurate formulation of the visibility condition results in a complex geometric object that is the combination of a quad's visible space for many view points at once. However, performing a topological sort on this graph proves to be a difficult endeavor, requiring careful consideration of the graph's size and the complexity of its traversal. The implementation of this graph-based approach takes advantage of practically relevant simplifications while also being guided by constraints of memory usage.

3.1 Topological Sorting of the Visibility Graph

Since in many instances, a single order is not correct for all camera positions, the question arises as to where a boundary may be placed. For now, the set of camera positions C for which the sort order does not need to change is assumed to be described by a convex polytope, referred to as the **camera polytope**. When the camera exits C, the new camera polytope C' around its current position may require a different sort order.

Definition 3.1 (Quad Surface). The surface $S_q \subset P_q$ of a quad q contains points x that lie on its plane with $(x - c_q) \cdot n_q = 0$ and are within the area formed by its vertexes. Quads are assumed to be convex.

Definition 3.2 (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. Additionally, both quads must be facing the camera to be visible at all with $(s - t) \cdot n_p > 0$ and $(s - t) \cdot n_q > 0$.

Definition 3.3 (Visibility Relation). The visibility relation V_C on the quads contains all pairs $(q, p) \in Q \times Q$ where p is visible through q from camera polytope C.

While this relation is irreflexive since no quad can see itself and asymmetric since no two quads can both be seen through each other, assuming they are non-intersecting, it

is not transitive and therefore does not qualify as a strict partial order. There are triplets of quads that do fulfill the transitive property, but two pairs $q \; V_C \; p$ and $p \; V_C \; r$ do not necessitate $q \; V_C \; r$. An example of this behavior is given by Figure 3.1.

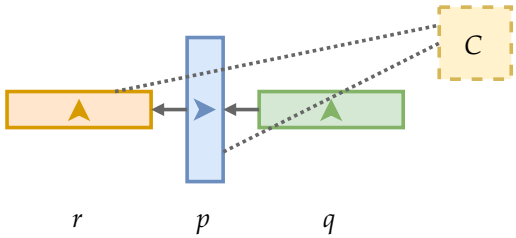


Figure 3.1: Amongst these three quads q can see p and p can see r but q and r cannot see each other as they are coplanar. The camera polytope C can have any size for this example to work. The arrows indicate the visibility relation, and view rays are represented by dotted lines.

Definition 3.4 (Visibility Graph). Interpreting the quads Q as nodes and the visibility relation V_C as a set of directed edges E_C between them yields the visibility graph $G_C = (Q, E_C)$.

A sort order is correct if it satisfies all visibility relationships, and as such, for each $q, p \in Q$ it must order p before q if q can see p. Translucency sorting can now be formulated as a problem on this graph. By these definitions, a topological ordering of the visibility graph G_C is a correct sort order for all view points in the camera polytope C.

In the worst case, the number of edges in the visibility graph is in $\Theta(|Q|^2)$. For example, this is encountered in a construction where all quads have the same orientation, their centers lie on a straight line, but none of them are coplanar. When numbered by ascending dot product and with an appropriate camera polytope, this results in each quad $i > 0$ seeing quads 0 to $i - 1$ and not seeing the rest. While explicitly generating and storing all edges can be avoided by implicitly testing two quads for visibility as needed by the search, the large number of edges means the topological sort of the visibility graph nevertheless needs to process $\Theta(|Q|^2)$ edges in the worst case. Even with a data structure that returns unvisited quads, up to $\Theta(|Q|)$ unsuccessful edge tests per quad leave the time complexity unchanged. Section 3.2 discusses the difficulty of efficiently finding an unvisited quad that is guaranteed to be a neighbor in the graph.

Static Topological Sorting

If the same sort order is correct for all camera positions and thus only a single camera polytope $C = \mathbb{R}^3$ is needed, the visibility condition looses the requirement that view rays start at a particular set of positions. Whereas the accurate visibility condition described with Definition 3.2 constrains the view ray at three points, i.e. the start in the camera polytope, the start quad, and the target quad, it now only constrains two points and the facing of the participating quads.

Definition 3.5 (Reduced Visibility Condition). A quad p is reduced visible through another quad q if there is a view ray from a point $s \in S_q$ to a point $t \in S_p$ that starts on its originating quad's back face with $(t - s) \cdot n_q < 0$ and intersects the front face of the other quad with $(s - t) \cdot n_p > 0$.

The camera cannot see a quad p through q if q is not currently visible to it, but it may move into a position where this is the case without the sort order requiring an update. This is taken into account by making no requirement that the quads be visible to the camera in the context of reduced visibility. In fact, it places no constraints on the camera polytope or its exact position since the camera is considered to be everywhere at once for static sorting.

Definition 3.6 (Half-Space). Given a quad's plane $P_q = (n_q, d_q)$ its (inner) half-space $H_q \subset \mathbb{R}^3$ consists of all points visible through the quad. This means $(x - c_q) \cdot n_q < 0$ holds for all $x \in H_q$. Half-spaces and their variants are analogously defined with the base notation H_P for arbitrary planes P that are not necessarily generated by quads.

Definition 3.7 (Closed Half-Space). The closed half-space $H_q^= = P_q \cup H_q$ contains the points on the inner side of the plane P_q as well as the points on the plane itself, unlike the regular open half-space H_q, which excludes P_q.

Definition 3.8 (Outer Half-Space). The outer half-space $\overline{H}_q = \mathbb{R}^3 \setminus H_q^=$ consists of all points outside the closed inner half-space, which are exactly those from which q is visible. The closed outer half-space $\overline{H}_q^=$ is defined as $P_q \cup \overline{H}_q$ and contains all points invisible to q.

This reduced visibility condition can also be expressed in terms of the quad's half-spaces. It allows view rays originating from $s \in S_q$ to target any point in a quad's half-space H_q, such as $t \in S_p$ on a second quad p's surface. Further, the facing requirement is fulfilled if the front face of p is visible from s, meaning $s \in S_q$ is in the set of points $\overline{H}_p$ from which p is visible. Since quads are convex, testing if any point of a quad's surface is within a half-space only requires checking each of the four vertexes. Performing such an edge test for the implicit visibility graph only takes constant time. A topological sorting algorithm based on a depth first search (Cormen et al., 2001, Section 22.4) that runs in $O(|V| + |E|)$ time can find a static sort order in $O(|Q|^2)$ time.

Outside special cases handled by the heuristic, the possibility of static sorting is determined by attempting a topological sort on the section under this assumption. When the graph is found to contain cycles, this means more than one sort order is necessary, and the quads need to be filtered by visibility.

Visibility Between Aligned Quads

The special case of aligned quads can be further optimized to only operate on **axis-aligned bounding boxes (AABBs)**. This is important since the implementation does not explicitly store the individual vertexes of the quads after they've been written into the vertex buffer. The facing, the normal vector, the dot product, and the AABB are computed from the vertexes for both aligned and unaligned quads. Even without vertexes, the test for the reduced visibility condition can be correctly performed by using just the AABBs of two

aligned quads. If either or both quads are unaligned, the test is performed by testing half-spaces against the quad centers, which is an approximate solution.

Visibility between parallel quads, which have equal or opposing normals, can be easily tested by comparing their dot products, where the special case of equal dot products $d_q = d_p$ indicating coplanar quads is considered to never yield visibility. This leaves pairs of orthogonal aligned quads. Both parts of the reduced visibility condition's formulation based on half-spaces are tested by comparing the AABBs. An AABB B_q for a quad q consists of two vectors $a^q, b^q \in \mathbb{R}^3$ describing two opposing corners with $a_i^q \leq b_i^q$ for each axis $i \in \{1, 2, 3\}$. A point x is in the AABB B_q if it lies within the interval $[a_i^q, b_i^q]$ in all axes. Aligned quads have AABBs with only four distinct corners since they have no width in the direction of their facing.

Assuming, without loss of generality, that both quads have distinct positive facings $k, j \in \{1, 2, 3\}$ for q and p respectively, a point $t \in S_p$ is in H_q if B_p extends into B_q with $b_k^p < a_k^q = b_k^q$. For t on the front face of p to be visible from a point $s \in S_q$ on the surface of q, the bounding box B_q needs to extend into $\overline{H}_p$ with $b_j^q > a_j^p = b_j^p$. Figure 3.2 gives an example of this test. Negative facings work analogously by switching the lower and upper bounds of the AABBs and the directions of the inequalities.

Figure 3.2: Quads q shown in blue can see p shown in green with the dotted line being a possible view ray between them. The arrows indicate their orthogonal facings in 2D. The AABBs for such aligned quads are flat and have no volume. The descent $b_k^q - b_k^p > 0$ of p into H_q is marked in red while the ascent $b_j^q - b_j^p > 0$ of q into $\overline{H}_p$ is marked in orange. Viewing this scene in 3D simply extrudes the quads without changing the content.

3.2 Dynamic Sorting With the Camera Polytope

In the general case, only quads visible from the camera polytope need to be sorted. While this is also true when there is only one sort order, it does not have an effect in that case. An important observation is that with topological sorting and satisfaction of the visibility condition, the camera polytope can be chosen such that the camera only exits it when a quad switches from being invisible to visible. Quads that have become invisible are no longer relevant for sorting since they cannot be seen. This means that the camera polytope around the current camera position can be described by the planes P_q of all quads q that are invisible. With this definition, the visibility graph is acyclic, and topological sorting yields a result if there is no intersecting geometry, only visible quads are processed, and a correct ordering is possible at all. Other formulations of the camera polytope are possible too, but this one yields correct sorting and lends itself well to detecting when the camera exits it. Smaller variants, such as one that is also bounded by the planes of visible quads, require re-sorting too often, while larger ones may lead to unsortable cyclic visibility graphs.

The sort order does not need to be recomputed when the camera crosses a quad's plane such that it is then invisible. Consequently, triggering any quad with normal n is only necessary when the normal-relative movement interval $m_n = [n \cdot c_1, n \cdot c_2]$ for a movement $m = (c_1, c_2)$ satisfies $n \cdot c_1 < n \cdot c_2$. The camera polytope C can be constructed out of only the inner half-spaces H_q of the quads $q \in Q$ by taking advantage of invisible quads remaining in the sort order when the camera crosses quad planes against the direction of their normal. The construction can be formulated as $C = \bigcap_{q \in Q,\, c \in H_q} H_q$. It relies on the fact that a quad q is not visible if the camera at c is inside the inner half-space H_q. When the camera exits C, it crosses into $\overline{H}_q$ of some quad q, which has thus become visible.

Topological sorting is more expensive than distance sorting but needs to be performed less often, as a new sort order is only necessary when the camera crosses a quad's plane. Testing all planes in the scene for crossing each time the camera moves is too much work. Instead, an important constraint on the geometry and a tree structure are used to make dynamic triggering efficient. Minecraft's geometry is largely axis-aligned, but the unaligned quads have hundreds of unique normals that arise from the complex geometry generated for water surfaces. Quantizing the normals to be at most a few dozen values makes creating structures for each unique normal feasible. Each normal is projected onto the surface of a unit cube, snapped to the closest of a few evenly spaced grid points, and normalized.

Now that there are only a constant number of normals $N = \{n_q \mid q \in Q\}$ to deal with, the problem of triggering can be solved for each one separately. For each normal $n \in N$ an interval tree (Cormen et al., 2001, Section 14.3) keeps track of the dot product interval $[d_{\min}, d_{\max}]$ for each section s produced by its quad's dot products d_q for $n_q = n$ if any such quads exist in section s. Each entry in turn contains a simple sorted list of the dot products for this normal in the section. When a section is updated, this list can be regenerated from the geometry and replaced in the section's entry in the tree. To check which quad planes a camera movement m crosses, for each normal $n \in N$ the interval tree is queried with the normal-relative movement interval m_n to get a set of candidate sections. A normal is skipped if the camera moved in the wrong direction with respect to it. Each candidate section is then verified by performing a binary search to check if any of its dot product values fall within m_n. The triggered sections are scheduled for re-sorting with the new camera polytope around c_2.

Querying the interval tree takes $O(\log n)$ time and checking k candidate sections for relevance takes $\Theta(k \log n)$ time. Updates to the interval tree also only take $O(\log n)$ time each. In total, finding the plane crossings for a camera movement takes $O(|N|\, k \log n)$ time. The time complexity depending on k is a trade-off to avoid storing all dot products in one big heap for each normal since, in that case, updates are more expensive. Such a shared heap would take $O(m \log n)$ time to add or remove a section with m values for a single normal and $O(|N|\, m \log n)$ query time for all normals together. It also poses the challenge of associating a set of potentially many sections with each dot product value, leading to an increase in memory consumption. This problem also exists in the interval tree, but to a much lesser extent as the intervals are less likely to be exactly equal and there are fewer of them in total.

The quantization of the normals means the position at which the section is triggered

becomes an approximation, but usually the camera polytopes overlap enough that this rarely becomes problematic. Triggering only for camera movement in the direction of the quad's facing means the camera polytopes are not necessarily disjoint because multiple sort orders can be correct for the same camera position. It should also be noted that the triggering system tracks the triggering planes from all sections, but each section is only sorted with the camera polytope generated by the planes from its own geometry.

Solving the Visibility Condition

The camera polytope is the intersection of half-spaces $C = \bigcap_{q \in B_c} H_q$ where $B_c \subseteq Q$ is a minimal set of required quads for the camera polytope around the position c. Each invisible quad's half-space H_q is either part of the construction with $q \in B_c$ or it has a larger dot product $d_q > d_p$ than another quad $p \in B_c$ and does not constrain the polytope. The camera polytope C is convex because it is formed by the intersection of half-spaces and can thus be expressed as a set of linear constraints (Cormen et al., 2001, Chapter 29, Page 775). Additionally, only a constant number $|B_c|$ of half-spaces, and thereby linear constraints, are needed because there are only a constant number of normals after quantization. In Figure 3.3, an example of the space T_{Cq} visible through a quad q from the camera polytope C is given.

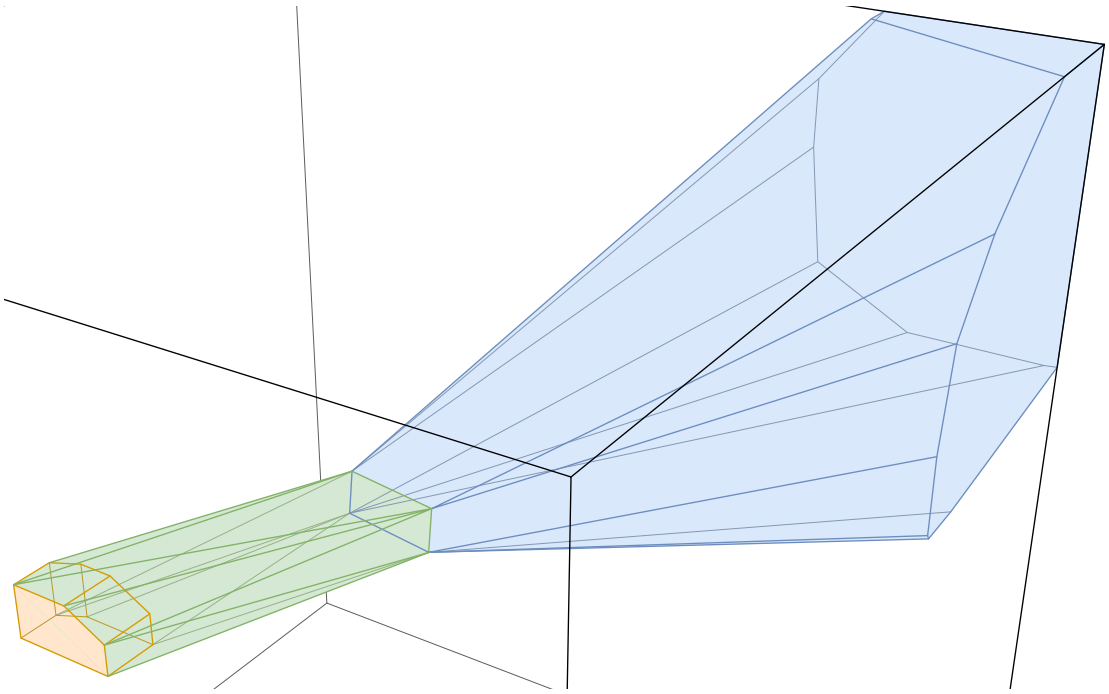


Figure 3.3: This figure gives an example of a quad q and the camera polytope C, shown in orange. The points that are visible through the quad from any point in the camera polytope form the convex blue polytope T_{Cq}. Any geometry within this space needs to be rendered before q. The space between the camera polytope and q is highlighted in green, and any geometry within it must be rendered after q. Both volumes are bounded by the section, whereas the camera polytope can also extend outside the section.

To test if a point x falls within the visible polytope T_{Cq}, the intersection $C \cap \overline{H}_q \cap \bigcap_{i=1}^{4} K_i$ of the camera polytope C, the outer half-space $\overline{H}_q$ of the quad itself, and four half-spaces K_i, with $i = 1, \dots, 4$ constructed from each edge of the quad q and x, is tested for emptiness. If it is not empty, the point can be seen from the camera polytope through the quad. Instead of checking if x is within the visible polytope, this construction checks if there is any point in the camera polytope from which x can be seen through the quad. The point x and the vertexes v_i and $v_{i+1 \bmod 4}$ construct the inner half space K_i such that it includes the quad's other vertexes if $x \in H_q$. The feasibility of a linear program for this construction with three variables and a constant number of constraints yields the result of the visibility condition. This means testing the visibility condition only takes constant time if quantized normals are used.

Extensions to Topological Sorting

The topological sorting algorithm based on a depth-first search presented here explores many of the possible edges in the graph. The visibility graph with its $O(|Q|^2)$ edges is often a nearly complete graph, which makes it harder to sort depending on the order in which quads happen to be iterated. Each edge can be thought of as an ordering constraint in the visibility relation from which this graph is constructed. If there are many constraints for the ordering and all of them are checked, constructing an ordering that satisfies all of them is slow. Transitivity can also not be used to simplify traversal of the graph.

Instead of performing an edge test between the current quad and all unvisited nodes, all the edges could also be generated ahead of time, but this incurs a high memory cost and also takes $O(|Q|^2)$ time. Only checking a subset of each quad's neighbors $N(q)$ does not bring any benefit because, by the nature of the depth-first search, only one unvisited node is visited at a time. Each neighbor needs to be visited eventually, and assuming the neighbors are selected efficiently, there is no reason to prefer visiting them in any particular order.

However, efficiently selecting an unvisited neighbor quad $p \in N(q) \subset Q$ can reduce how often all quads Q are iterated through. A runtime of $\Theta(|Q|^2)$ arises when each of the $|Q|$ quads test $\Theta(|Q|)$ many other quads for being a neighbor, which means testing the visibility condition each time. Special cases not caught by the heuristics can however have a far lower runtime in $O(|Q|)$ when the graph has few edges and the ordering of the quads allows finding these edges immediately without unsuccessful edge tests.

If the next neighbor to visit is determined efficiently, only $O(|Q|)$ many edge tests are ever performed in total, since each one is successful or does not happen in the first place, such as when the graph contains disconnected components. In fact, no edge tests need to be performed at all if a neighbor that already fulfills the visibility condition is given. In geometric terms with Figure 3.3 on the previous page in mind, this requires efficiently finding a visible and unvisited quad within the visibility polytope marked in blue. Under the reduced visibility condition, the problem becomes finding a visible quad in an entire inner half-space instead of only within the accurate but potentially complex visibility polytope. The acceleration data structure for this can be a simple space tree over the vertexes of all quads that can answer half-space queries in $O(\log n)$ time. Assuming

it can be constructed for the visible quads in $O(n \log n)$ time and quads that are visited can be efficiently removed, this structure would perform topological translucency sorting in $O(n \log n)$ time under the unrealistic condition that the camera polytope is able to see the entire inner half-space of each quad. It can thus only accelerate the cases in which the graph is fully sortable with the reduced visibility condition.

Since this is not actually the case, the acceleration structure needs to support queries for quads only contained within the convex visibility polytope of each quad. Simply storing unconnected vertexes in the structure also does not work under this requirement, for the polytope can be pointy, and only testing if it contains vertexes is insufficient. The polytope could intersect with a quad without containing any of its vertexes by sticking through the quad's surface or only grazing one of its edges. Using the reduced visibility condition, this problem does not occur as the query is an entire half-space and at least one vertex is within a half-space if the quad intersects with it. Literature suggests that while 3D interval trees of orthogonal objects are possible (Edelsbrunner and Maurer, 1981)(Finkler and Pashchenko, 2000), they do not support this type of query efficiently, if at all. More general spatial structures could fulfill polytope queries, but without yielding a good worst-case complexity.

3.3 The Implementation and its Limits

While the quadratic worst-case time complexity of the topological sort makes its application to sections with many quads infeasible, this threshold depends on how fast each edge test can be performed. The accurate visibility condition is complex and testing it is expensive. Instead, the reduced visibility condition, which is also used in static topological sorting and does not restrict the origin of view rays to the camera polytope, is used for acceptable performance. Only quads visible to the camera are sorted to avoid some of the cycles in the visibility graph. However, this does not avoid all cycles, which may appear because of the reduced visibility condition, since it overapproximates the existence of some edges.

The implementation also approximates the shape of unaligned quads to avoid a complex test for their intersection with other geometry. Given the discovery of a much more efficient dynamic sorting approach as described in Chapter 4, making the implementation of dynamic topological sorting very fast is not necessary in practice, especially since the tricks presented here leave the time complexity unchanged.

Separators

Edges in the graph resulting from view rays that originate at positions the camera cannot get into without exiting its current camera polytope and thereby triggering the sorting process from a new position can be eliminated by checking for separators. A **separator** is a plane that puts the camera and the target quad p on one side and the source quad q on the other. Now the edge from q to p is impossible to realize unless the camera moves across the separator plane. If the separator plane is outside the camera polytope, the edge between q and p does not exist. Figure 3.4 on the following page illustrates the effect of a separator.

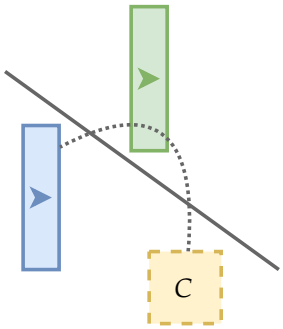


Figure 3.4: The camera polytope C and the blue target quad p are on the same side of the unaligned diagonal separator, while the green quad q is on the other side. Only impossible bent view rays like the one shown with a dotted line can come from C, pass through q, and hit p. The separator proves that p is not visible through q from any point in C.

Separators that intersect the camera polytope can still be used by introducing the separator plane as an additional half-space constraining the camera polytope. In this case, the triggering mechanism needs to be made aware of the new triggering plane that is not directly derived from a quad. Separator planes are ideally chosen from existing quads or at least with normals that are already present in the construction of C. Otherwise, a linear program needs to be solved to check if the separator intersects with the camera polytope. Static sorting cannot take advantage of separators because it does not use the bounded camera polytope they require. The simplification of the visibility condition to allow view rays to come from any direction is exactly what static sorting requires for the sort order to be correct from every view point. This means it cannot take advantage of constraints on possible camera positions for the construction of separators.

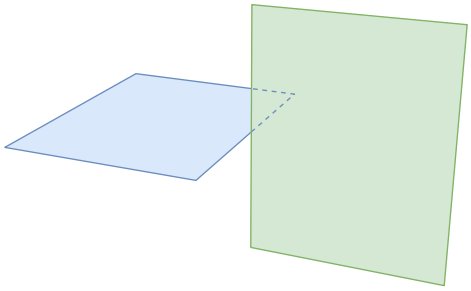


Figure 3.5: Two orthogonal quads that only touch at one point on their edges without intersecting. A separator plane with a new orientation is required between the two to allow this scene to be sorted with the reduced visibility condition.

The simple case illustrated by Figure 3.5 where two quads with orthogonal normals only touch at exactly one point along an edge is enough to render a scene unsortable using the reduced visibility condition. Both quads in this scenario can see each other, but the camera can actually only ever see one through the other. A separator plane between the two quads fixes the cycle in the visibility graph by removing one of the edges, depending on which side of the separator the camera is on. This type of construction can be extended

to require multiple separator planes with different orientations. Multiple separators with the same orientation do not have to be entered into the triggering system because the closest valid separator plane triggers first and thus performs the same function as ones farther away.

If the separator plane has a new and unaligned orientation, it is referred to as an **unaligned separator**, of which an example is given in Figure 3.4 on the preceding page. Searching for unaligned separators is hard as the solution space is very large. The implementation does not attempt to do so and instead only uses aligned separators as available. When a cycle in the visibility graph is encountered, even though separators were used for testing edges, it cancels the topological sort. A simple heuristic based on the number of unique normals and quads decides whether to attempt static or dynamic topological sorting, and falls back to distance sorting if they fail or performing them takes too long. Since topological sorting may be successful with a different camera position after an initial failure, the fallback mechanism attempts topological sorting a few more times when the section is triggered, depending on the measured execution time. When camera polytope triggering is no longer required, the section is removed from the plane-based triggering structure, and only the distance- or angle-based triggering remains.

4

Sorting With Partition Trees

A very useful but common property present in Minecraft's geometry is one already hinted at in the previous chapter with the introduction of separators. Nearly all geometry is on the blocks' faces or otherwise fully contained within each block. If the space is partitioned into two half-spaces and no geometry crosses the partitioning plane, there is no way for the camera to see objects in the half-space it is in through objects in the other half-space. Separators similarly use a plane to obtain a view point-dependent ordering of quads. This concept, when applied recursively, leads to the construction of a **partition tree**, from which a sort order can then be generated.

4.1 Partition Trees

Definition 4.1 (Valid Partition). A partition $R = (g, d)$ of $\mathbb{R}^3$ is a plane described by normal $g \in \mathbb{R}^3$ and distance $d \in \mathbb{R}$. Its half-space H_R and plane surface P_R are defined analogously to the terms for quads. A partition is valid, also called a **free cut**, if no quads are intersected by the partition plane. Every quad $q \in Q$ is either fully in P_R with $q \in Q_R^= \subseteq Q$, or some of its vertexes lie in either H_R with $q \in Q_R^< \subseteq Q$ or $\overline{H}_R$ with $q \in Q_R^> \subseteq Q$. These sets reconstruct Q with $|Q| = |Q_R^=| + |Q_R^<| + |Q_R^>|$ if the partition is valid.

This definition, though not the concept of a free cut, diverges somewhat from common formulations of space partitions like the one given in (Paterson and Yao, 1990), as usually valid partitions can intersect with objects, resulting in their fragmentation along the partition plane. While the concept of partition complexity, which measures how many fragments of geometry are generated by non-free cuts, is usually interesting in the context of space partitioning, it is not of particular concern here. As mentioned in the introduction, this work only focuses on free cuts because of both the implementation, where additional geometry has a significant cost, and practical considerations, namely that geometry fragmentation is never really necessary.

Definition 4.2 (Useful Partition). A partition R is useful if it partitions Q into at least two non-empty subsets. This means that out of the pairwise disjoint sets $Q_R^<$, $Q_R^>$, and $Q_R^=$ at least two must be non-empty. If Q only contains one item, no useful partition exists, but there is also no need to perform further partitioning.

A **partitioning operation** $\varphi(Q) := (R, \varphi(Q_R^<), Q_R^=, \varphi(Q_R^>))$ defined over a set of quads Q constructs the partition tree where $\varphi(Q) := Q$ if $|Q| \leq 1$. Each partition must be useful to avoid unnecessary partitions and to ensure the recursion eventually terminates. If a set of quads permits no useful partition, it is called **unpartitionable**. Figures 4.1 and 4.2 give examples of unpartitionable aligned and unaligned geometry, respectively. A partitionable set of quads with an axis-aligned partition is aligned partitionable; otherwise, it is unaligned partitionable. A partition where quads lie on its plane is an **autopartition**.

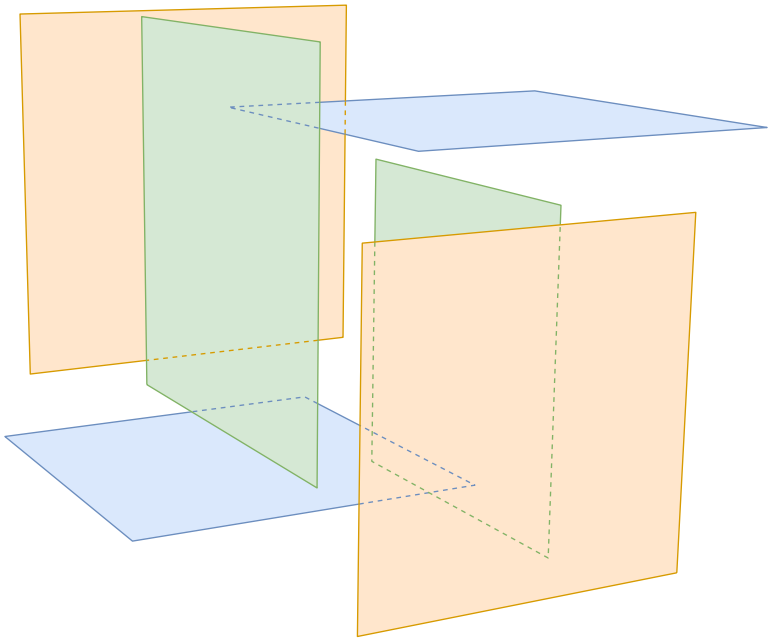


Figure 4.1: This set of aligned quads in 3D space is not partitionable. The upper horizontal blue quad lies above the front orange quad in such a way that no partition plane can separate the orange quad from the rest. Such a relationship holds for each quad. This construction remains unpartitionable until any three quads or two quads orthogonal to each other are removed.

Figure 4.2: With just three unaligned quads, here shown as a 2D cut, this minimal scene is unpartitionable even in 3D space. Unlike the structure in Figure 4.1, removing any quad makes it easily partitionable along the plane of a remaining quad.

Known Orientation Partitioning

Constructing a partition tree from arbitrary geometry is not easy in general, but an efficient partitioning approach is enabled by taking advantage of the special case presented by Minecraft's mostly aligned partitionable geometry. Aligned partitioning is a spe-

cial case of known-orientation partitioning where the partitions are known to have axis-aligned orientations. However, it also works for any other constant-sized set of known partition orientations. When the set of possible orientations is not known, there are a far greater number of possible solutions, since the orientation of each partition must then also be determined in addition to its dot product. The problem of unknown-orientation partitioning is the subject of Chapter 5. In the context of Minecraft, it is also referred to as **unaligned partitioning**, because only axis-aligned partitions are known to be common.

Finding a partition of n quads can be performed in $O(n \log n)$ time if its orientation, given by a normal g, is known. Each quad's vertexes $V_q = \{v_1, v_2, v_3, v_4\}$ are projected, and the endpoints of the resulting interval $[\min_{v \in V_q} v \cdot g, \max_{v \in V_q} v \cdot g]$ are sorted. Each interval endpoint (d_n, t, q) contains a projected dot product d_g, its type $t \in \{\top, \bot, \circ\}$, namely an interval start $\top$, end $\bot$, or zero-width interval $\circ$, and a reference to the originating quad q. Partitions are possible at all dot product values where the number of overlapping intervals is zero when scanning over the sorted list of endpoints. Special care is taken to order groups of endpoints that have the same dot product d_g as $(d_n, \bot, \cdot) < (d_n, \circ, \cdot) < (d_n, \top, \cdot)$ to ensure zero-width gaps permitting partitions between quads are detected. Ends of intervals are ordered before starts of intervals, and zero-width points are put between the two. If one orientation does not yield any partitions, the process is repeated with a different known orientation that has not yet been tried. Figure 4.3 gives an example of the projection and partition scanning for gaps.

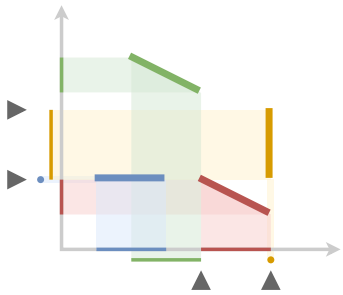


Figure 4.3: Each quad is projected in each of the two orientations: vertical and horizontal. They produce either intervals with endpoints or single points if they are projected into an interval with no width. Gaps indicated by the triangles along the axis lines are found by scanning the sorted interval endpoints starting at the origin.

Known-orientation partitioning with this scanning method often produces more than one partition plane if scanning continues after the first partition is found. Instead of repeating the partitioning step for the remaining quads in recursion, all available partitions are combined into one multi-partition node. A **multi-partition** node is equivalent to a sequence of nested binary partition nodes, which can each have quads on the partition plane and a child node in either half-space. Which axis, or more generally, orientation, is checked first for partitioning alternates in round-robin fashion. The orientation that was just used to create a multi-partition cannot, by definition, immediately yield any further partitions. The geometry must first be partitioned along a different orientation. In Minecraft, often only three or fewer levels of multi-partitioning are necessary when the

geometry is aligned to the faces of full blocks. In Figure 4.4, recursive multi-partitioning is performed on an example scene that includes unaligned objects and multiple partitions on the same level. While the unaligned partitions are easy to find in this example, the aforementioned alignment properties of the geometry make an implementation of unaligned partitioning largely unhelpful.

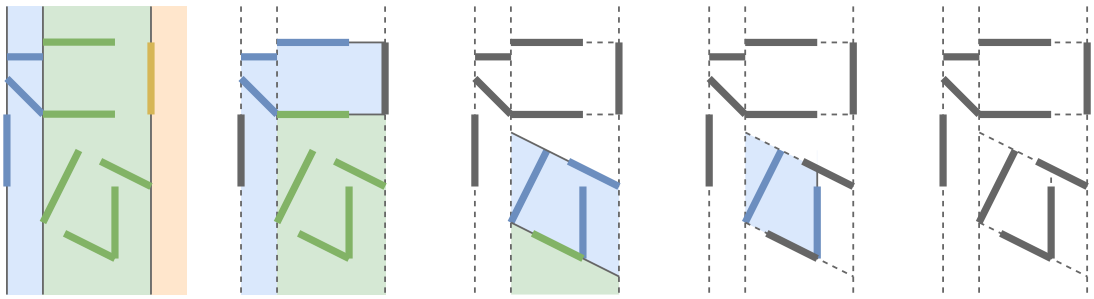


Figure 4.4: Recursively multi-partitioning this set of 2D lines leads to a partition tree. Each partition plane can take advantage of storing on-plane quads in the partition node instead of needing to further partition them. The third round of partitioning consists of an unaligned multi-partition, which the implementation does not support.

The time complexity of known-orientation partitioning is $\Theta(n^2 \log n)$ when an imbalanced tree is generated and each quad requires processing all remaining quads again. Multi-partitioning obviates the decision of picking one of multiple partitions with the same orientation to ensure balancing since, in such situations, all partitions are taken at once. It does not actively balance trees if the geometry naturally leads to imbalanced partitions with alternating orientations. An example of this worst case is given in Figure 4.5. However, if balanced partitioning is possible, the complexity drops to $O(n \log^2 n)$ as shown by the Master Theorem (Cormen et al., 2001, Theorem 4.1) with $T(n) = m\,T(\frac{n}{m}) + n \log n$ and the partition count per level $m \geq 2$. While the construction of a partition tree depends on how well it can be balanced, generating a sort order with one always only takes linear time.

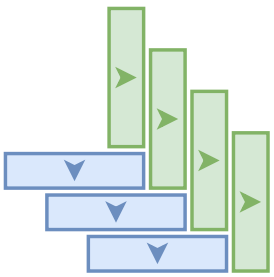


Figure 4.5: In this worst-case situation, each quad needs to be partitioned separately, and the axis of the partitions alternates. Multi-partitioning degenerates to binary partitioning here since only one partition is possible in each step.

Generating a Sort Order

Figure 3.4 on page 23 pertains to separators, but the concept translates directly to a partition ordering the two half-spaces it generates. The quads in the half-space that the camera is in are ordered after those on the plane and on the other side of it. If the camera is on the plane, the half-spaces can be ordered in any way, but before the quads lying on the plane. The recursive structure of the tree further orders the quads within each subset. Quads on the plane do not need further ordering as they cannot be seen through each other. Collecting the sort order from a multi-partition node alternates between quads on the planes and visiting child nodes between the planes, starting at those farthest from the camera. The sort orders from the partitions before and after the camera can be generated independently and in any order, as there is no visibility between the geometry of the two sides. The quads from the partition the camera is in or the plane it is on must be added to the sort order last.

Reading a sort order out of the partition tree for a given camera position takes place in linear time, as each of the at most $O(|Q|)$ partition nodes is visited once and each quad index only needs to be written into the buffer once. Even if the tree is not balanced, visiting each node and writing each quad index only once results in linear runtime.

The sort order generated with a partition tree is only correct until the camera crosses one of the partition planes; since then the order of at least two nodes in the output needs to be swapped. After the construction of the partition tree, its planes are added to the triggering system, making them part of the half-spaces that constrain the camera polytope and trigger sections to be re-sorted. Notably, they trigger bidirectionally instead of just in one direction, as with visibility graph sorting.

4.2 Implementation of (Multi) Partition Trees

While usually there are not many partitions in each section, often 16 at most, due to the section's size in full blocks, there can be very many interval endpoints to sort. Each quad can generate four points, one for each vertex, if it is unaligned with regard to the current partitioning orientation. In practice, however, quads are only partitioned using their axis-aligned bounding boxes, and only aligned partitions are considered. Even with aligned partitions of bounding boxes, the number of points to sort can be large, meaning a fast way to sort these points, each consisting of three values (d_n, t, q), is required. The ordering depends on both the dot product d_n and the type t, and while the association to quad q does not change the order, it does have to remain associated with the point. Packing d_n as a 32-bit float, t as two bits, and the index of q as a 30-bit integer into a 64-bit integer encodes the point such that comparing it by value gives the correct order. Floats can be compared using their binary representation if some simple bitwise transformations are applied. Sorting an array of points encoded in this way is significantly faster than using Java's objects and sorting the array with a comparator. The sorted points are decoded as needed for the gap detection scan.

Memory bandwidth and latency play an important role in the implementation because they limit how fast the index buffer can be written and the partition tree can be read. As a side effect, extra work can be performed during index buffer writing without

incurring much additional latency because it is mostly hidden by the existing unavoidable memory latency.

The quad indexes in the partition tree's leaf nodes and in the sets of on-plane quads often follow repeating patterns or stay within a small range of values. Above a certain size, this allows quad index sets to be compressed. However, the overall memory usage of the partition trees is not very large to begin with. Compression only affects the integer arrays containing the quad indexes and the objects making up the tree's structure. To compress a set of quad indexes, the array is sorted, and the differences between consecutive indexes are packed tightly into the 32-bit integers, which otherwise store a list of indexes, alongside a header containing compression parameters. Especially the common case of a full $[0, k]$ index interval without holes is compressed to just eight bytes with this scheme. Both compression and decompression bring no improvement in measured partitioning or sorting performance but reduce memory consumption slightly. This lack of a performance benefit likely stems from the memory bottleneck of writing the index buffer, which cannot be reduced in length.

Handling of Special Cases

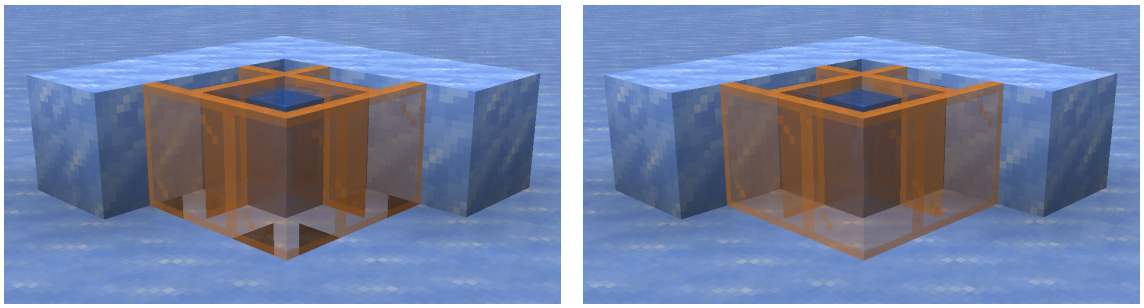
The partitioning implementation can take advantage of multiple special cases to improve performance or memory usage. When two quads $q, p \in Q$ with opposing normals $n_p = -n_q$ are partitioned, the resulting tree node does not need to order them and simply writes their indexes into the sort order upon being visited. This concept can be expanded to handle multiple coplanar quads or quads with opposing normals. In fact, the same special case handling that is applied to whole sections can be transferred to each partitioning step, which includes the SNR and convex box-aligned cases.

Just as whole sections are sorted with the topological graph sort, if the heuristic decides they are small and simple enough, subsets of the quads can also be sorted in this way. Both static and dynamic topological sorting can be applied as if it were a partition tree node. The static or dynamic sort order is generated as needed. The partition tree generates the sort order very efficiently and is largely memory-bound, which other types of nodes, such as ones generated by a special case, are as well. Advantages possibly gained from reducing the number of traversed objects in the tree, each of which incurs a memory latency cost, do not appear to have a relevant impact. Applying special cases inside the partition tree is only beneficial if they are cheap to test and significantly reduce the number of partitions or tree nodes.

A visually important special case is intersecting geometry, because the result is more appealing if quads that intersect with the rest of the geometry are rendered first. The graph sorting implementation is only able to detect aligned intersecting geometry, and in that case, it attempts to achieve a good result by approximation. It falls back to other sorting methods when it fails to sort intersecting unaligned geometry. When the partitioning algorithm fails, it checks if the geometry is intersecting or if it is actually unpartitionable. In the latter case, it falls back to other dynamic sorting methods.

However, if the geometry is intersecting, which is the likelier alternative in Minecraft, static topological sorting is attempted, and if that too fails, it applies **primary intersector** detection. This means that when a minority of quads intersects a majority of the other

quads, the detected set of primary intersector quads is logically partitioned from the rest. Regular recursive partitioning is performed on these two subsets, usually successfully. Always rendering the set of primary intersectors before the other geometry yields a visually appealing result that is often better than distance sorting for such intersecting geometry. An example of a simple scene with and without primary intersector handling is given by Figure 4.6. Section 2.4 discusses the advantages and problems of distance sorting in more detail.



(a) Disabled (using distance sorting) (b) Enabled (using partition tree)

Figure 4.6: The water and the orange glass panes in this scene are intersecting. Without primary intersector detection, distance sorting is used as a fallback and produces errors in the form of dark triangles where the ground is rendered too late in the sort order (a). With it enabled, however, the section can be sorted with the partition tree, and visual artifacts are avoided (b).

Partial Tree Updates

Even for small changes to a section's geometry, the entire partition tree is rebuilt, which results in repeating the same calculations for many of the partitioning steps as the previous time it was rebuilt. To avoid this duplicate work, nodes with unchanged content can be re-used in the new tree. They are considered unchanged if the number, shape, and order of the axis-aligned bounding boxes are the same. Retaining this reuse data, which consists of bounding boxes and the indexes of the input to the partition process, in order to perform this check costs memory and is thus only done if it is likely that the section will change. A node's reuse data is only generated the second time a section is rebuilt in an attempt to avoid generating it for sections that are only built once and then never modified. Node reuse is also only enabled above a certain size threshold. The decision of when to allocate and deallocate this data constitutes a caching problem. A similar caching problem occurs in the management of CPU-side index buffers for dynamic sorting.

Reusing a matching node requires generating a remapping function that adapts the potentially compressed indexes contained within the node to match the new partition tree's indexing. Even if a node's quads, or rather their bounding boxes, have not changed, the indexing may need to be modified to account for changes in the indexes that precede it. The remapping function is computed by matching the indexes of the sequence of quads that was used to construct the node being reused to the new sequence of quads to be

partitioned. While it is sensitive to the order of quads, which is actually irrelevant to the sorting problem, this is unproblematic as the meshing algorithm can be assumed to generate quads for each block and its faces in a deterministic order. The function is simplified to be a constant offset if all the reused nodes' indexes are remapped in the same way.

4.3 Implementation Performance

The implementation is written with the goal of achieving an optimal visual result with minimal delay in both the loading of sections and dynamic sorting updates. At the same time, the complexity of the partition tree and topological sorting algorithms require it to be well organized, considering that Java code can become hard to read if it is highly optimized in terms of raw throughput. Even without optimizing the Java code to produce optimal compiled assembly, the improvements in algorithmic complexity mean that it generally outperforms simpler to optimize approaches like distance sorting. The primary comparisons made here are against no sorting, meaning that the implementation of translucency sorting does not significantly slow down the process of meshing sections, and against a very fast distance sorting implementation using radix sort.

Table 4.7 lists the sort type counts of the sections containing translucent geometry for a representative selection of common scenes. Of these, the Hermitcraft[3] scenes were picked as instances of player-created worlds with many glass and water structures. The other randomly generated worlds were intentionally selected to show a variety of natural scenarios with differing characteristics. These figures include all sections within the circular render distance, whereas a typical player might not cause all underground sections to become loaded if they are never visible. The distribution of sort types depends on the specific geometry and thus varies between the scenes. This variation can also impact the performance observed when the player moves the camera around the world. Dynamic sorting requires updates, while static sorting performs its work during meshing and avoids sort order updates.

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 4.7: This table shows the section counts per sort type for a set of worlds representative of various common scenarios. Sections with translucent geometry are counted based on the sort type they have after meshing, sorting, and handling of sort failures. The sort type distribution varies greatly depending on the specific scene and the overall nature of the geometry.

[3] https://hermitcraft.com/

Measured timings of the partition tree building and static topological sorting are presented in Table 4.8. While building the tree takes significantly longer than generating a sort order from it, it still only represents a small fraction of the overall time it takes to mesh a section. Importantly, the slower process of building the tree is performed during meshing and not each time a dynamic sort update is necessary. In these test cases, sorting the quads by distance takes 60 percent longer on average than generating a sort order with the partition tree. It should be noted that sorting with a partition tree has complex implications for the timing of sort order updates; distance sorting requires regular updates as the camera changes its angle or distance relative to the section, while geometry-based triggering schedules dynamic sorting at irregular intervals. The advantage is that aligned geometry only needs to be sorted when the player crosses its planes, which is not the case for the vast majority of sections at any given time. Static topological sorting is again somewhat slower than partition tree-based sorting, but it can produce a static sort order that never needs to be updated. However, since it has quadratic runtime characteristics, the sections on which it is attempted are limited in size to prevent too slow meshing.

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 4.8: All timings are the mean time it took to perform the task for a section in nanoseconds per quad. For the algorithms with non-linear runtimes, these values depend on the average section size. The first two columns refer to partition tree building and generating an initial sort order from the tree. The distance sort is performed for comparison on the sections that are processed with the partition tree. Timings of the static topological sorting are only recorded from successful attempts on its filtered set of sections.

In an extreme case with just under 50,000 quads, node reuse improves partition tree building from around 230 down to 142 nanoseconds per quad on average. The overall performance in such a large section is also worse than the average case seen in the tables above. This is likely due to logarithmic factors in the partitioning algorithm growing with the large size and effects of cache residency. Notably, the time to generate a sort order barely increases to just 23 nanoseconds per quad in this special case, emphasizing the practically linear complexity of partition tree sorting in practice.

Testing subsets of quads for being sortable with SNR during partitioning is only effective if iterating over the geometry twice is avoided, and it does not introduce too much overhead. Integrating SNR detection entails keeping track of the types of facings during the quad iteration that produces interval endpoints and afterwards checking if only oppositely aligned facings are present. In the Hermitcraft S9 test case, this has no effect on tree building or sorting performance, but it does reduce the overall number of trigger planes by 5 percent, meaning dynamically sorted sections need to be sorted slightly less often in response to camera movement.

5

Unaligned Partitioning

Sometimes the construction of a fragmentation-free partition tree requires partitioning geometry along a plane of unknown orientation. The partitioning problem greatly differs in complexity depending on whether the orientations of the partitions are known ahead of time. Known orientation partitioning can be easily solved by sorting interval endpoints as described in Section 4.1. Without loss of generality, the three axis-aligned orientations are treated as known since they are common in Minecraft, whereas the many remaining unaligned orientations are treated as unknown. Picking a different constant-sized set of known orientations does not make the problem easier. Problem 5.1 defines the general problem of finding valid and useful partitions without prior knowledge of their orientation.

Problem 5.1 (Unaligned Partitioning (UAP)). Given a set of n partitionable quads Q, does a valid and useful partition R exist, and what are its parameters? This means that no quad intersects R and that at least two of $Q_R^<$, $Q_R^=$, and $Q_R^>$ are non-empty. The orientations of the partitions are not known ahead of time.

Solving UAP is performed on the parameterized representation of partition planes, which can be linearly mapped from their corresponding quad geometry. The expression of UAP as a series of set operations on partitions is easily visualized as intersections of geometric volumes in parameter space. UAP has a combinatorial solution, giving it an upper bound of $O(n^4)$. It can also be turned into a set of linear constraints and solved with typical linear programming techniques, but this approach does not yield a better upper bound. The transfer of a lower bound to UAP can be achieved by reducing linear constraint solving to it.

Both aligned and unaligned geometry can be used to force the only useful partition to be unaligned. In the unaligned case, a stack of sufficiently large and parallel but unaligned quads requires the partition planes to be nearly or exactly the same orientation as the quads. With aligned geometry, the construction is somewhat more complex since autopartitions are not unaligned partitions when the quads are aligned. Slightly pulling an unpartitionable construction consisting of four aligned quads similar to the one shown in Figure 4.2 on page 26 apart yields the scene in Figure 5.1 on the following page, forcing the partition to be unaligned.

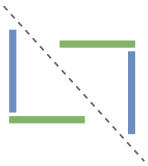


Figure 5.1: This construction, here shown as a 2D cut, consisting of aligned geometry only permits unaligned partitions that diagonally cross through the center, as indicated by the dashed line.

The continuous-valued parameters of a plane can describe an infinite number of useful partitions, if the geometry permits it. There are always an infinite number of potentially not useful valid partitions that lie far away from any geometry. An instance can have zero, a finite number, or an infinite number of useful partitions.

Overview of the Problem Space

While UAP is a hard problem, it does not have the highest difficulty in the hierarchy of translucency sorting problems. Figure 5.2 on the next page gives an overview of some selected classes that instances of the most general translucency sorting problem can fall into. The simplest case is geometry that either permits any sort order or is at least statically sortable with SNR. Harder instances include those permitting aligned or unaligned partitions, and finally those that are sortable but unpartitionable. Topological sorting with the accurate visibility condition can handle all sortable cases, but the implementation, which uses the reduced visibility condition, falls somewhere between aligned partitionable and sortable, depending on how well separators can be used and which approximations for unaligned quads come into play. The last class in the hierarchy is made up of non-intersecting but not sortable cases, such as shown in Figure 2.2 on page 8.

Non-intersecting aligned geometry that is not sortable does not exist in this diagram, as it is considered impossible. The structure that produces a cyclic overlap relies on unaligned quads, whereas a similar structure with only aligned quads would either be topologically sortable or contain intersections. Notably, the diagram shows that the alignment of partitions and geometry are not directly related. As shown in Figure 5.1, aligned geometry can force unaligned partitions, and aligned partitions are sometimes enough for unaligned geometry.

5.1 Parameter-Space Interpretation

A plane in 3D space has essentially three degrees of freedom, which can be expressed in different ways. The slope and offset formulation $y = ux + vz + d$ of a plane lets the parameter space encompass all relevant solutions with just three parameters $(u, v, d) \in \mathbb{R}^3$. Planes parallel to the Y-axis cannot be expressed with this parametrization since it would require an infinite slope u or v. These singularities are eliminated by using three rotated instances of the parametrization, one for each axis. Any method for determining unaligned partitions by searching the parameter space can be applied to each instance

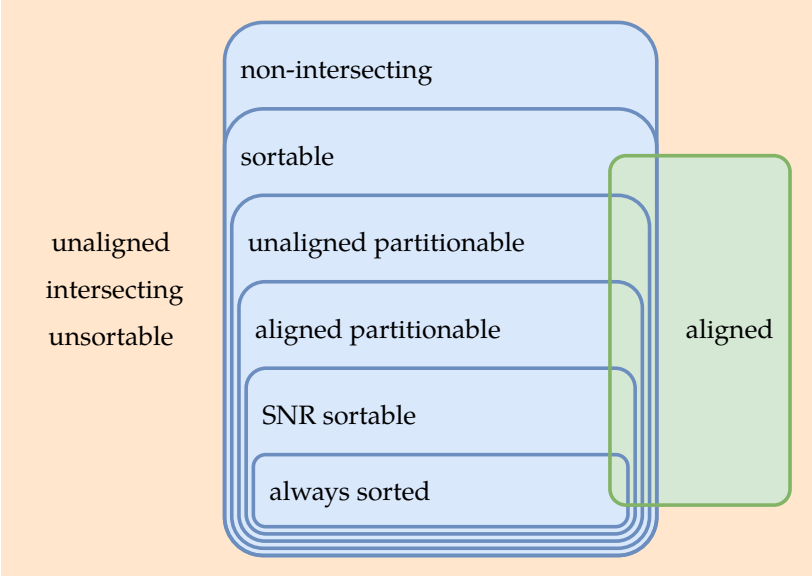


Figure 5.2: An instance of the translucency sorting problem, i.e. a set of quads, can have the shown attributes, which can include or intersect each other. The arrangement of sortable and non-intersecting in relation to aligned implies that there are no aligned cases that are not sortable but still non-intersecting.

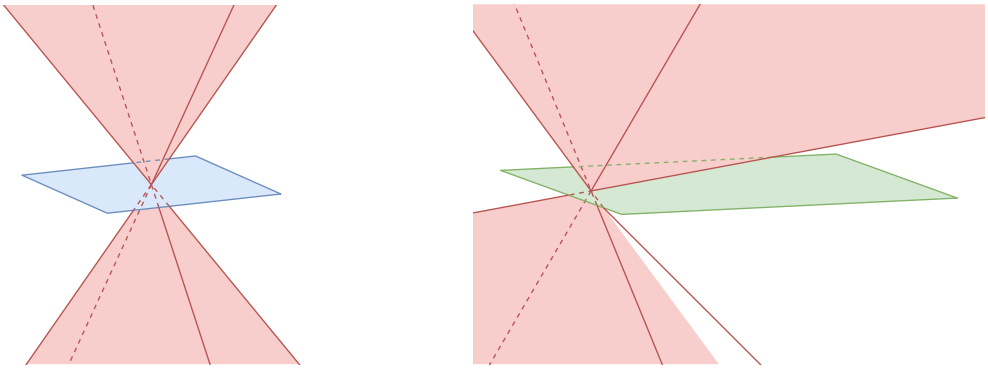
without increasing the total complexity, as this only happens a constant number of times. This ensures that no solution remains inaccessible, even if only one of the three instances is able to represent and thus identify it.

Parameter-space points are noted as $(u, v, d) \in \mathbb{L} = \mathbb{R}^3$ to differentiate them from real-space points $(x, y, z) \in \mathbb{R}^3$ despite both using the same underlying set $\mathbb{R}^3$. The **slope-offset parametrization** of the inner half-space $H^{=}_{R}$ of a partition $R = (u, v, d) \in \mathbb{L}$ is $y \leq ux + vz + d$ with a point $p = (x, y, z) \in \mathbb{R}^3$ while $y \geq ux + vz + d$ corresponds to the outer half-space $\overline{H}^{=}_{R}$. The plane equation $y \leq ux + vz + d$ can be inverted by rearranging and negating its terms to yield the equivalent plane equation $d \geq -xu - zv + y$ in parameter space. The function $\Pi : \mathbb{R}^3 \to \mathbb{L}$ performs the linear transformation $\Pi(x, y, z) := (-x, -z, y)$ of p into the parameters of the outer parameter-space half-space $\overline{H}^{=}_{\Pi(p)}$ containing all $R \in \mathbb{L}$ for which p is in the real-space half-space $H^{=}_{R}$. The representation of points as the set of half-spaces that contain them as a half-space in parameter space simplifies working with sets of partitions as they can now be intersected geometrically.

Other ways of parameterizing planes do not have this useful linear relationship between their real and parameter spaces. Although the parametrization of a plane as its unit normal $g \in \mathbb{R}^3$ and dot product $d \in \mathbb{R}$ as $(d, g) \in \mathbb{R}^4$ seems intuitive based on its previous appearances, it leads to a four-dimensional parameter space consisting of a unit sphere infinitely extruded into the fourth dimension. This shape does not lend itself to a simple transformation from points to their corresponding set of containing half-space planes $\Pi(p)$. Other approaches that come with similar problems include using polar coordinates, encoding slopes as angles, the axis intercept form, and other combinations of these concepts.

Unaligned Partitioning in Parameter-Space

A valid partition R puts all vertexes of each quad in either $H_R^=$ or $\overline{H}_R^=$. Applying this condition in parameter-space representation means the vertexes $v_1, \dots, v_4$ of a quad q are in $H_R^=$ if R is in each of the parameter-space half-spaces in which all encoded partitions contain their respective vertex, i.e. R is in their intersection $\Gamma_q^\leq = \bigcap_{i=1}^4 \overline{H}^=_{\Pi(v_i)}$. An analogous parameter-space intersection $\Gamma_q^\geq = \bigcap_{i=1}^4 H^=_{\Pi(v_i)}$ is possible when all vertexes are in $\overline{H}_R^=$. From this, it follows that R is valid with regard to q if the **valid partition set** $\Gamma_q = \Gamma_q^\leq \cup \Gamma_q^\geq$ contains it. Both of the constituent sets $\Gamma_q^\leq$ and $\Gamma_q^\geq$ have the shape of a four-sided pyramid with their tips touching at the point in parameter space representing the quad's own plane P_q. This plane is the only one where all vertexes are in both the inner and outer half-space, which means it is the only point at the intersection with $\Gamma_q^\leq \cap \Gamma_q^\geq = \{P_q\}$. The shape and behavior of the parameter-space valid partition sets for a single quad are shown in Figure 5.3.



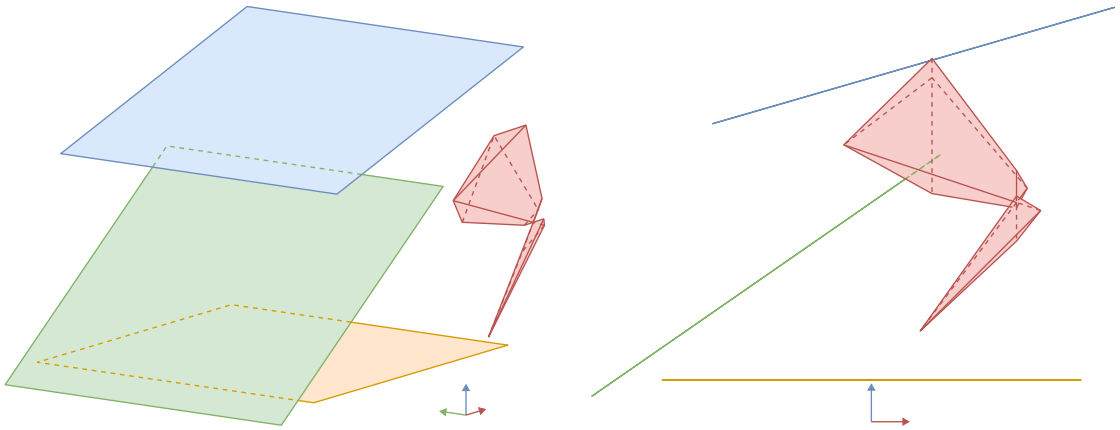
(a) Centered on origin

(b) Offset from origin

Figure 5.3: The boundaries of Γ_q and Γ_p are shown in red for the blue quad q in (a) and green quad p in (b), respectively. The upper pyramid-shaped $\Gamma_q^\geq$ contains the parameter points for all partitions R that contain the vertexes of q in $H_R^=$. Since both quads are aligned with the Y-axis (up), the parameter-space point of each quad's own plane, i.e. P_q and P_p, is at the origin where the two parts of the valid partition sets touch. This graphic combines parameter and real space for the purpose of visualization.

Viewing the UAP as a problem in parameter space makes it a geometric intersection problem where validity and usefulness can be expressed in terms of parameter-space intersection of half-spaces. In a scene with multiple quads Q, a partition R is valid if the overall valid partition set $\Gamma_Q = \bigcap_{q \in Q} \Gamma_q$ contains it. The partition is useful if it partitions the quads into at least two non-empty sets, which means there must be a quad $q \in Q$ such that $R \in \Gamma_q^\leq$ and a different quad $p \in Q$ such that $R \in \Gamma_q^\geq$. The **useful partition set** Λ_Q containing all such partitions is a strict subset of Γ_Q as there are always valid but useless partitions arbitrarily far away from the geometry. It can be expressed as $\Lambda_Q = \Gamma_Q \setminus (\Gamma_Q^\leq \cup \Gamma_Q^\geq)$ where $\Gamma_Q^\leq = \bigcap_{q \in Q} \Gamma_q^\leq$ and $\Gamma_Q^\geq = \bigcap_{q \in Q} \Gamma_q^\geq$ are the sets of valid but useless partitions R, which put all quads in $Q_R^<$ or $Q_R^>$, respectively. They exclude the only element of $\bigcap_{q \in Q} \{P_q\}$ if it exists, which implies that all quads are coplanar, as it is not a useful

partition even though it may be an intersection of a $\Gamma^{\leq}_q$ and $\Gamma^{\leq}_p$ for some quads $q, p \in Q$. However, writing Λ_Q in terms of $\Gamma^{<}_q = \Gamma^{\leq}_q \setminus \{P_q\}$ and $\Gamma^{>}_q = \Gamma^{\geq}_q \setminus \{P_q\}$ instead would exclude useful autopartitions, which are formed when not all quads are coplanar. Figure 5.4 gives an example of a useful partition set with multiple parts.



(a) Full (b) Right side

Figure 5.4: The set of useful partitions Λ_Q for the three quads is highlighted in red. It consists of two parts, each of which corresponds to one of the gaps shown between the quads in the side view (b). They touch at the parameter-space point for the plane P_q of the green quad q in the center. This graphic once again combines parameter and real space for the purpose of visualization.

The set of valid partitions is always unbounded. Figure 5.3 on the preceding page shows an example of this for single quads where the d parameter of a valid partition $R = (0, 0, d)$ can grow to any value. Unbounded useful partition sets arise from the intersection of appropriately overlapping unbounded valid partition sets. Alternatively, an instance can have exactly $k \in \mathbb{N}$ useful partitions by constraining each one by at least four on-plane vertexes with at least one on either side. This construction requires $\Theta(k)$ quads, which can be non-intersecting if they are small or spaced far enough apart to avoid intersection.

5.2 Upper Bounds

Faces of Λ_Q correspond to vertexes in real space. Each vertex $v \in \mathbb{R}$ generates a plane $\Pi(v) \in \mathbb{L}$, which can then contribute one or more faces to the useful partition set's boundary. A vertex $R \in \Lambda_Q$ of the useful partition set is the intersection of the boundary planes $\{R\} = \bigcap_{v \in V} \Pi(v)$ given by a set of at least three real-space vertexes $V \subseteq \bigcup_{q \in Q} V_q$. It follows that the encoded real-space plane R touches each of the vertexes in V if and only if R is a vertex of the useful partition set.

In the following, it is useful to think of the set of useful partitions as split into **families of partitions**. Each partition R in a family of partitions $F \subseteq \Lambda_Q$ produces the same partition result given by $Q^{<}_R$, $Q^{=}_R$, and $Q^{>}_R$. Families can share vertexes in parameter-space that

represent autopartitions. The simplest example of this is the families $\Gamma_q^{\leq}$ and $\Gamma_q^{\geq}$ sharing P_q at their tips, assuming partitions in the area around P_q are rendered useful by other geometry. However, the autopartition P_q may not be useful, even though all non-shared partitions around it are. A construction of two coplanar quads with a gap between them generates such a case since the shared autopartition does not separate them. A partition coplanar with all quads is referred to as a **global useless autopartition**. It can remain valid and become useful if small perturbations are applied to it.

Points in useful partition families are either at a vertex or can be moved to one without leaving the family, except for the global useless autopartition. The vertexes of a partition family are connected to each of its points with a straight line that does not exit the family's convex space. Following the correspondence between parameter-space vertexes and sets of real-space vertexes, this means all useful partitions can be moved to touch at least three vertexes. Conversely, this means only the vertexes of Λ_Q need to be considered when searching for useful partitions.

This observation limits the number of relevant partitions for n quads to $\binom{4n}{3} \in \Theta(n^3)$ and allows a simple algorithm that tests each of these candidates against all quads to solve UAP in $\Theta(n^4)$ time. It simply picks three of the quads' vertexes to construct a partition plane, checks that all quads lie on either side of or on the plane, and that the partition is useful by recording which side they lie on. The global useless autopartition, which only occurs when all quads are coplanar, is the exception to this rule. When the plane generated by the selected vertexes is coplanar with all quads, an additional supporting real-space vertex is arbitrarily created that does not lie in this plane. Assuming a partition the parametrization cannot represent is avoided, one of the planes resulting from this special case will be the useful partition if one exists. If there is additional non-coplanar geometry, it either renders the autopartition useful or prevents it from being valid. In either case, all useful partition families only have vertexes representing useful partitions now.

Solving Unaligned Partitioning with Linear Programming

Both halves $\Gamma_q^{\leq}$ and $\Gamma_q^{\geq}$ of a valid partition set for a single quad q can be formulated and solved as a set of linear constraints. The parameter space $\mathbb{L}$ serves as the domain yielding three variables u, v, and d, and each of the intersected half-spaces contributes a constraint. Such a set of m linear constraints over three variables is equivalent to a linear program (LP) without an optimization function, here referred to as a **linear constraint set** (LCS). Since any feasible solution to the set of constraints suffices as a valid partition, no optimization is necessary. Finding any feasible solution to a set of constraints is at most as hard as finding the optimal solution to the equivalent linear program with an arbitrary optimization function. Each quad's two valid partition sets are encoded with an LCS, one for each half.

Reducing the useful partition set to LCS feasibility requires many LCS instances, as each one must be convex. For every permissible partition of the quads into $Q_R^<$ and $Q_R^>$, which correspond to placing the quad above or below the partition plane, an LCS for the intersection of the useful partition set $\bigcap_{q \in Q_R^<} \Gamma_q^{\leq}$ for quads below and the set $\bigcap_{q \in Q_R^>} \Gamma_q^{\geq}$ for quads above the partition plane is tested for feasibility. There are $2^n - 2$ possibili-

ties in total, since the valid but useless partitions that put all quads on the same side are excluded. For $n = |Q|$ quads this results in solving $\Theta(2^n)$ LCS instances with $\Theta(n)$ constraints for feasibility through linear programming, each of which takes $O^*(n^{2+1/6} \cdot L)$ time with $\delta = 2^{-L}$ and $\omega = 2$ (Cohen, Lee, and Song, 2018). The global useless autopartition is handled separately without changing the overall complexity. This approach is clearly worse than the aforementioned upper bound $O(n^4)$ given by the combinatorial search.

Taking advantage of the aforementioned upper bound of $O(n^3)$ on the number of vertexes in parameter-space allows the complexity of this LCS-based approach to be reduced. The following algorithm that incrementally adds and updates useful partition sets is only possible because their number is limited in each step. A subset of quads $K \subseteq Q$ is initially empty, and in each of the n iterations a quad q is added to yield $K \subset K' = K \cup \{q\}$. The algorithm maintains a set of currently useful partition families $U_K \subset 2^{\mathbb{L}}$ as well as the intersections $\Gamma^{\leqslant}_K$ and $\Gamma^{\geqslant}_K$ that may turn into useful partition families when quads are added. All sets of partitions are described as LCS instances where the intersection is the concatenation of the constraints and solving the instance tests them for emptiness. The useful partition families are updated as

$$U_{K'} = \left(\{\Gamma^{\leqslant}_K \cap \Gamma^{\geqslant}_q, \Gamma^{\geqslant}_K \cap \Gamma^{\leqslant}_q\} \cup \bigcup_{F \in U_K} \{F \cap \Gamma^{\leqslant}_q, F \cap \Gamma^{\geqslant}_q\} \right) \setminus \emptyset$$

by adding new useful partitions that may be generated between q and K in the first part and ensuring each currently useful partition remains so after the addition of q in the second part. Removing the empty set is implemented by checking the merged LCS instances for emptiness. The intersections are updated as $\Gamma^{\leqslant}_{K'} = \Gamma^{\leqslant}_K \cap \Gamma^{\leqslant}_q$ and $\Gamma^{\geqslant}_{K'} = \Gamma^{\geqslant}_K \cap \Gamma^{\geqslant}_q$. At every point in the iteration the number of families in U_K remains in $O(n^3)$ as each vertex can be shared by at most two families, and each family must have at least one vertex. The global useless autopartition can be shared by more families and is handled separately.

Each of the n iterations requires solving up to $O(n^3)$ LCS instances, resulting in a total of $O(n^4)$ LCS instances. Together with the complexity of solving the instances themselves with linear programming, this results in a polynomial runtime, albeit still worse than the combinatorial approach. It should be noted that arithmetic operations are assumed to take constant time to ensure the calculations involved in testing partitions and solving LCS instances remain well-behaved.

Definition 5.2 (LCS-SAT). An instance of the decision problem LCS-SAT consists of a set of LCS instances K with n variables and m constraints and a boolean formula $f : 2^K \to \{0, 1\}$. It is accepted if the intersection, i.e. the concatenation, of a subset $\hat{K} \subseteq K$ is feasible and $f(\hat{K}) = 1$ holds.

UAP can be solved using LCS-SAT by making it test the intersections of valid partition sets for emptiness and constraining the results using the boolean formula to only allow useful partitions. Let K contain the LCS corresponding to $\Gamma^{\leqslant}_q$ and $\Gamma^{\geqslant}_q$ for each quad $q \in Q$. Further, let f restrict the selection to useful partitions with

$$f(\hat{K}) = \left(\bigwedge_{q \in Q} (\beta_{\hat{K}}(\Gamma^{\leqslant}_q) \vee \beta_{\hat{K}}(\Gamma^{\geqslant}_q)) \right) \wedge \neg \left(\bigwedge_{q \in Q} \beta_{\hat{K}}(\Gamma^{\leqslant}_q) \right) \wedge \neg \left(\bigwedge_{q \in Q} \beta_{\hat{K}}(\Gamma^{\geqslant}_q) \right),$$

where $\beta_{\hat{K}}$ returns the boolean value determining whether the LCS for a set of partitions is in $\hat{K}$. The first part of the formula ensures the partition is valid for all quads, while the following parts ensure the partition is not above or below all quads, which would render it useless. This constructed instance of LCS-SAT is accepted if and only if the corresponding UAP problem for Q is accepted too.

5.3 Lower Bounds

Under the assumption of constant-time arithmetic operations, a naive upper bound for LCS that simply tests all vertexes of the constraint set can be given. With just three variables, iterating over all $\binom{n}{3}$ combinations of the n constraints and computing their intersection in constant time puts it in $O(n^3)$. Obviously, this means it becomes infeasible if there are $\Theta(n)$ variables. While this naive approach is outperformed by the previously cited complexity of linear programming, especially if there are an equal number of constraints and variables, it does have a strong similarity to the combinatorial algorithm for UAP, which also picks triples of quads and tests vertexes, although they only exist in parameter space. Differences between real and parameter space become indistinct when all problems are parameterized for the purpose of optimization, except when the geometric interpretation of a linear program is the problem instance itself. Turning this notion of parameter and real space around leads to a reduction from LCS to UAP that takes a set of constraints and models them as quads such that there is a solution to the UAP instance if the LCS is feasible.

Given that solving UAP already requires solving LCS, this reduction is not particularly surprising, and it only requires isolating the LCS-solving capabilities of a UAP solver from the logic UAP introduces on top of the underlying geometry. Such a reduction transfers any lower bound on LCS to UAP, since solving LCS instances indirectly cannot possibly be faster than the established lower bound on solving them without the reduction. It also leads to an upper bound on LCS, as its instances can be indirectly solved using the reduction in, at most, the time it takes to solve the equivalent UAP instances. However, this upper bound for LCS is not new, since UAP already uses either a solver for LCS or the combinatorial algorithm that essentially corresponds to the combinatorial algorithm for LCS. Given an LCS instance I with n constraints and three variables, the goal is to generate a set of quads Q constituting a UAP instance that is accepted if and only if I is feasible.

The difficulty of creating an appropriate set of quads depends on how far the definition of UAP can be stretched. At first, quads are allowed to have any number of vertexes, and the size of the UAP instance is measured in vertexes instead of quads. Generating a quad with one vertex per constraint in I only works if all constraints fit the format $y \leq ux + vz + d$, since the half-spaces always face upwards or downwards, depending on which side of the partition the quad is supposed to be on. If this is in fact the case, the reduction works with just two quads, a first one q with a vertex for each constraint in I and a second one p with only one vertex far away to ensure any valid partition is useful. The quad p can be given four unique vertexes by making it small enough that any solution to I can exist in the UAP parameter space without becoming invalidated by interference

from p.

If both $\leq$ and $\geq$ are used by the constraints in I, each one is represented by a vertex in the quads $q_{\leq}$ or $q_{\geq}$ and no additional quad p is necessary to ensure useful partitions since the useful partition property is needed to combine the upper and lower half-spaces of the two quads in the right way. The resulting real-space quads each have many vertexes and likely will not resemble usual quads at all. A construction that only uses quads with four vertexes each needs to distribute the constraints over the quads in groups of four. A problem with this is that useful partitions can arise between quads where they are not intentional. Unpartitionable constructions can be used to constrain the parameter-space in which valid partitions can form, though it requires that the solution space of the LCS is known and bounded, or at least that it can be bounded by an unpartitionable arrangement of quads.

An even greater difficulty is presented if only planar quads are acceptable, which means that at least two of each quad's vertexes cannot be controlled by a constraint and instead might unintentionally exclude solutions. Interference can be avoided if only one face of a quad's parameter-space pyramids, which one depends on the direction of the constraint's inequality operator, is used by placing it such that the other planes fall outside the LCS's solution space. While reasonable quad geometry has generally been assumed so far, non-intersecting geometry has not. However, if such a requirement does exist, the problem becomes hard since the resulting quads in real space do not easily correspond to the parameter-space geometry. Avoiding intersections between quads in the UAP instance while preserving an accurate reduction from LCS might be possible, but doing so likely increases the complexity of the reduction itself, as it needs to solve a kind of packing problem in this scenario.

6

Conclusion and Outlook

The game's geometry is rendered as quads, most of which are aligned and even more of which, even if unaligned, are at least bounded within each block's extents. This insight led to many optimizations that are not otherwise possible in more general cases, such as other games or graphical applications, especially when it comes to alignment and aligned partitions. Efficiently generating a sort order and taking advantage of the many special cases required theoretical work, while the initial implementation of indexed rendering in Sodium proved to be technically difficult. Integrated tooling that is able to generate measurements and can be used to tune the many parameters controlling the decision-making was important for creating a useful implementation of the theoretical concepts. A number of interesting ideas turned out not to be useful in the implementation upon measurement of their performance, for reasons such as limited memory bandwidth or simply a realistic data set.

Many of the sections Minecraft renders do not need complex sorting or any sorting at all, both of which can be determined with simple heuristics. Static sorting types like SNR work alongside techniques to detect when they are applicable using only a constant small amount of data that is gathered during the meshing process. The class of dynamic sorting methods is used for all other sections for which a single sort order is not enough. The existing implementation in Minecraft only does distance sorting, which is a type of dynamic sorting. While it often yields correct results, it does not fully solve the problem, both visually and in terms of performance. Distance sorting is nevertheless used as a fallback after other methods have failed. Two approximations for triggering it were presented that use the camera's total movement length as a lower bound on the earliest trigger point.

Translucency sorting can be formulated as a graph problem since correct rendering requires only visually overlapping quads to be in the correct order. This led to the definition of a visibility condition and subsequently the corresponding visibility graph, of which the topological sort is a solution to the problem. Capturing the positions at which a sort order stays constant as a camera polytope and its boundaries as trigger planes gives certainty about the validity of each sort order, as opposed to distance sorting where any camera movement may cause immediate invalidation. This concept is central to continuously maintaining a visually correct result. Globally triggering sections for sorting using a collection of planes was efficiently implemented using a structure of interval trees and

sorted lists of dot products. Critically, efficient use of the triggering structure is only possible because unaligned normals are quantized, as there would otherwise be too many. To reduce complexity and make the performance acceptable, a reduced version of the visibility condition is implemented, and a number of supporting optimizations were presented, such as separator planes. Separators are an important concept both for graph-based sorting and the following partition tree sorting approach. In practice, topological sorting has performance issues with larger instances due to its quadratic complexity, but with the right size limits, it becomes a good tool for static sorting that entirely avoids triggering and sort updates.

Building a partition tree of the geometry and then generating sort orders based on the camera position was a big improvement over dynamic topological sorting. Building the partition tree only out of free cuts was an implementation constraint, though not a hindrance given the typically excellent partitionability and alignment of Minecraft's geometry. Two important concepts are valid partitions, which avoid intersection with the geometry, and useful partitions, which productively split the set of quads into at least two disjoint subsets. Apart from the two sets on either side of the partition, quads may also lie on the plane itself. Using multi-partition nodes whenever possible was a natural step given that the binary equivalents do the same work but with a greater overhead. The implemented partition tree can be built and used to sort a set of quads faster than the equivalent topological sort would usually take, and it can also generate a sort order faster than if the quads were sorted by distance. In part, this good performance can be attributed to the linear complexity of reading a sort order from the tree. Additionally, visual problems introduced by distance sorting were fixed by either of the dynamic sorting approaches, with partition tree sorting being the more practical of the two in the general case. Primary intersector detection was added as a workaround for the rare cases in which there is intersecting geometry, avoiding the fallback to distance sorting and maintaining visual correctness where the default implementation in Minecraft cannot.

Finally, possible solutions to the difficult UAP problem were investigated. Contrary to aligned partitioning, which was implemented and works by the presented sorting and scanning algorithm, no constant-sized set of partitioning orientations can be used for UAP. Picking three vertexes to form a partition and testing for its validity and usefulness yielded a combinatorial algorithm with an upper bound of $O(n^4)$. Parameterizing partition planes with a linear plane equation yielded the parameter space $\mathbb{L}$, in which points represent real-space partitions. In parameter space, the sets of valid and useful partitions resulting from the geometry were visualized as geometric objects. This interesting correspondence leads to a solution for UAP using an LCS solver, i.e. an LP feasibility test. However, this approach's polynomial runtime did not yield a better upper bound than the already established combinatorial solution, largely because very many LCS instances need to be tested. To obtain a lower bound for UAP, a sketch of how LCS may be reduced to UAP was presented under various degrees of constraints on the resulting UAP instance.

6.1 Outlook

The implementation of an acceleration structure for topological sorting using the reduced visibility condition was discussed in Section 3.2, with the main problem being that it would have significant overhead and cannot implement the full visibility condition. However, it may be useful as a tool for static sorting on sets of quads above a certain size. It remains to be investigated what this threshold is and if simple iteration over a quad array can be outperformed. Factors such as the alignment of the current quad, the general arrangement of the quads, and memory latency from resolving object references could all play a role. Especially with unaligned quads or unpartitionable geometry, a performant implementation that is better than the existing alternatives may be challenging given that there is a soft upper limit at around 5,000 on the number of quads that typically need to be sorted.

A related topic pertains to the combination of static topological sorting and partition tree sorting. The current implementation only performs topological sorting within the partition tree if it encounters geometry that it cannot partition. An optimization that may reduce the number of partition planes and thereby the frequency of dynamic sort updates is to attempt static topological sorting of quads, even if they can be partitioned but doing so is not necessary. Finding a static sort order for larger groups of quads within the partition tree could also further improve the speed at which a sort order is generated since it reduces the number of tree nodes involved. The nuance of this approach lies in the decision of when to attempt such sorting, since it can result in wasted effort if it turns out not to be possible statically. As part of this heuristic, additional special cases, which only whole sections are otherwise tested for, may be integrated into the partitioning process. There may also be a benefit to sharing an acceleration structure with the static topological sorting system.

Finally, **convex partitioning** is a superset of UAP that splits the geometry into convex groups of quads and then determines a sort order between them. This problem is even harder than UAP since it degenerates into simply partitioning each quad into its own convex partition and then sorting them individually. It may present an interesting theoretical perspective as a bridge between topological sorting and partition-based sorting while also potentially yielding insights valuable to the implementation of a combined approach to dynamic sorting.

Bibliography

Barta, P., Kovács, B., Szécsi, L., and Szirmay-kalos, L. (2011). Order independent transparency with per-pixel linked lists. Budapest University of Technology and Economics *3*.

Cohen, M.B., Lee, Y.T., and Song, Z. (2018). Solving Linear Programs in the Current Matrix Multiplication Time. CoRR *abs/1810.07896*. arXiv: 1810.07896. URL: http://arxiv.org/abs/1810.07896.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. (2001). Introduction to algorithms. 2nd ed. The MIT Press. ISBN: 0-262-03293-7.

Dorward, S.E. (1994). A survey of object-space hidden surface removal. International Journal of Computational Geometry & Applications *4*, 325–362.

Eck, D.J. (2023). Introduction to Computer Graphics. Hobart and William Smith College.

Edelsbrunner, H. and Maurer, H.A. (1981). On the Intersection of Orthogonal Objects. Inf. Process. Lett. *13*, 177–181. DOI: 10.1016/0020-0190(81)90053-3. URL: https://doi.org/10.1016/0020-0190(81)90053-3.

Everitt, C. (2001). Interactive order-independent transparency. White paper, nVIDIA *2*, 7.

Finkler, U. and Pashchenko, I. (2000). *Multidimensional Interval Trees*. Tech. rep. RC21837. IBM T.J. Watson Research Center. URL: https://dominoweb.draco.res.ibm.com/reports/RC21837.pdf.

Gerardin, M., Simonot, L., Farrugia, J.-P., Iehl, J.-C., Fournel, T., and Hébert, M. (2019). A translucency classification for computer graphics. Electronic Imaging *31*, 1–7.

Govindaraju, N.K., Henson, M., Lin, M.C., and Manocha, D. (2005). Interactive visibility ordering and transparency computations among geometric primitives in complex environments. In Proceedings of the 2005 symposium on Interactive 3D graphics and games, pp. 49–56.

JellySquid et al. (2020). *Sodium*. https://github.com/CaffeineMC/sodium-fabric.

McGuire, M. and Bavoil, L. (2013). Weighted blended order-independent transparency. Journal of Computer Graphics Techniques *2*.

Mojang Studios (2009). *Minecraft*. Sweden.

Mulmuley, K. (1991). Hidden surface removal with respect to a moving view point. In Proceedings of the twenty-third annual ACM symposium on Theory of computing, pp. 512–522.

Münstermann, C., Krumpen, S., Klein, R., and Peters, C. (2018). Moment-based order-independent transparency. Proceedings of the ACM on Computer Graphics and Interactive Techniques *1*, 1–20.

Paterson, M.S. and Yao, F.F. (1990). Efficient binary space partitions for hidden-surface removal and solid modeling. Discrete & Computational Geometry *5*, 485–503. ISSN: 1432-0444. DOI: 10.1007/BF02187806. URL: https://doi.org/10.1007/BF02187806.

Tóth, C.D. (2005). Binary space partitions: recent developments. Combinatorial and Computational Geometry *52*, 525–552.

Tsopouridis, G., Fudos, I., and Vasilakis, A.-A. (2022). Deep hybrid order-independent transparency. The Visual Computer *38*, 3289–3300.

Weber, C. and Stamminger, M. (2016). Topological triangle sorting for predefined camera paths. In. VMV '16. Bayreuth, Germany: Eurographics Association, pp. 153–160. ISBN: 9783038680253.

Weiler, K. and Atherton, P. (1977). Hidden surface removal using polygon area sorting. ACM SIGGRAPH computer graphics *11*, 214–222.

Zhang, T. "Tanki" (2021). Handling Translucency with Real-Time Ray Tracing. Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX, 127–138.