POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Ray Distribution Aware Heuristics for Bounding Volume Hierarchies Construction

Tesi di Laurea Magistrale in
Computer Science and Engineering

Author: **Lapo Falcone**

Student ID: 996089
Advisor: Prof. Marco Gribaudo
Academic Year: 2023-24

Abstract

Abstract

Keywords: here, the keywords, of your thesis

Abstract in lingua italiana

Abstract Italiano

Parole chiave: qui, vanno, le parole chiave, della tesi

Contents

Introduction

Intro [3]

1 | Chapter one

Chapter 1

2 | Chapter two

Chapter 2

Bibliography

[1] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire. *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, Boca Raton, FL, USA, 2018. ISBN 978-1-13862-700-0. Chapter 22.2.

[2] G. Gribb and K. Hartmann. Fast extraction of viewing frustum planes from the world-view-projection matrix. *Online document*, 2001.

[3] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner. A survey on bounding volume hierarchies for ray tracing. In *Computer Graphics Forum*, volume 40, pages 683–712. Wiley Online Library, 2021.

[4] S. Owen. Ray-plane intersection. `https://education.siggraph.org/static/HyperGraph/raytrace/rayplane_intersection.htm`, 1999. Accessed: (11/01/2024).

[5] S. Owen. Ray-box intersection. `https://education.siggraph.org/static/HyperGraph/raytrace/rtinter3.htm`, 2001. Accessed: (10/01/2024).

A | Collision and Culling Algorithms

A.1. Ray-AABB Intersection

The algorithm we used to detect intersections between a ray and an AABB is the branchless slab algorithm [5].

Given a ray in the form: $r(t) = O + t \cdot d$, where O is the origin and d the direction, the main idea of the algorithm is to find the 2 values of t ($\overline{t_1}$ and $\overline{t_2}$) such that $r(\overline{t_{1,2}})$ are the points where the ray intersects the AABB.

Since the object to intersect the ray with is an axis-aligned bounding box in the min-max form, the algorithm can proceed one dimension at a time:

1. First, it finds the intersection points of the ray with the planes parallel to the yz plane, and sorts them in an ascending order with reference to the corresponding $\overline{t_{1,2}}$ values. We call the point with the smallest $\bar{t}$ value the *closest*, and the other one the $\overline{furthest}$.
2. Then it does the same with the xz plane:
   - As closest intersection point, it keeps the furthest between the 2 closest intersection points found so far (the one with the yz plane and the one with the xz plane).
   - As furthest intersection point, it keeps the closest between the 2 furthest intersection points found so far.
3. Then it does the same with the xy plane.
4. Finally, an intersection is detected only in the case where the furthest intersection point actually has an associated $\bar{t}$ value bigger than the one of the closest point found by the algorithm.

5. The returned $\bar{t}$ value is the smaller one, as long as it is greater or equal to 0; otherwise it means that the origin of the ray is inside the AABB, and one of the intersection points is *behind* the ray origin.

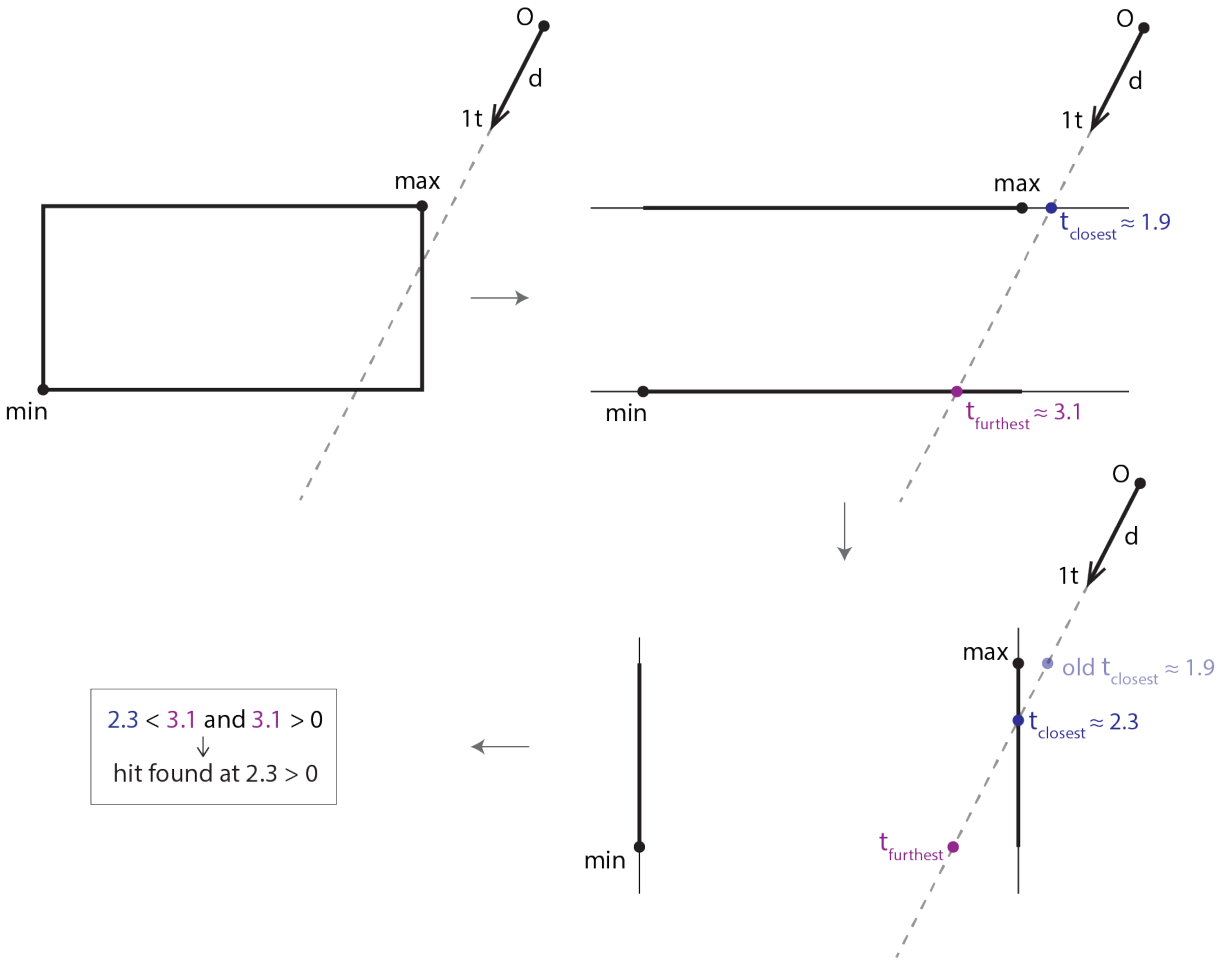


Figure A.1: Visual representation of the presented algorithm in 2 dimensions. An interactive simulation of this algorithm can be found at: `https://www.geogebra.org/m/np3tnjvb`.

Algorithm A.1 Ray-AABB branchless slab intersection algorithm in 3 dimensions

function INTERSECT($ray, aabb$)
  $tx1 \leftarrow (aabb.min.x - ray.origin.x)/ray.direction.x$ ▷ *yz plane*
  $tx2 \leftarrow (aabb.max.x - ray.origin.x)/ray.direction.x$
  $tMin \leftarrow min(tx1, tx2)$
  $tMax \leftarrow max(tx1, tx2)$
  $ty1 \leftarrow (aabb.min.y - ray.origin.y)/ray.direction.y$ ▷ *xz plane*
  $ty2 \leftarrow (aabb.max.y - ray.origin.y)/ray.direction.y$
  $tMin \leftarrow max(tMin, min(ty1, ty2))$
  $tMax \leftarrow min(tMax, max(ty1, ty2))$
  $tz1 \leftarrow (aabb.min.z - ray.origin.z)/ray.direction.z$ ▷ *xy plane*
  $tz2 \leftarrow (aabb.max.z - ray.origin.z)/ray.direction.z$
  $tMin \leftarrow max(tMin, min(tz1, tz2))$
  $tMax \leftarrow min(tMax, max(tz1, tz2))$
  $areColliding \leftarrow tMax > tMin$ **and** $tMax \geq 0$
  $collisionDist \leftarrow tMin < 0?tMax : tMin$
  return $\langle areColliding, collisionDist \rangle$

It is interesting to note how, under the floating-point IEEE 754 standard, the algorithm also works when it is not possible to find an intersection point along a certain axis (i.e. when the ray is parallel to certain planes). Indeed, in such cases, the values $\overline{t_{1,2}}$ will be $\pm\infty$, and the comparisons will still be well defined.

A.2. Ray-Plane Intersection

For ray-plane intersection we decided to use this algorithm presented in the educational portal of the SIGGRAPH conference [4].

Given a ray in the form: $r(t) = O + t \cdot d$, where O is the origin and d the direction, and a plane whose normal n and a point P are known, we first check whether the plane and the ray are parallel, in which case no intersection can be found.

Then, if they are not parallel, we obtain the analytic form of the 3-dimensional plane:

$$A \cdot x + B \cdot y + C \cdot z + D = 0$$

In particular, we know a point P that is part of the plane, therefore we can obtain the D

parameter:

$$
\begin{aligned}
& A \cdot P.x + B \cdot P.y + C \cdot P.z + D = 0 \\
\implies & D = -(A \cdot P.x + B \cdot P.y + C \cdot P.z)
\end{aligned}
$$

By definition, the vector formed by the parameters $[A, B, C]$ is perpendicular to the plane, therefore:

$$
\begin{aligned}
& D = -(n.x \cdot P.x + n.y \cdot P.y + n.z \cdot P.z) \\
\implies & D = -\langle n \cdot P \rangle
\end{aligned}
$$

Now that we have the parametric equation of the plane, we can force a point of the plane to also be a point of the ray:

$$
\begin{aligned}
& A \cdot r(t).x + B \cdot r(t).y + C \cdot r(t).z + D = 0 \\
\implies & A \cdot (O.x + t \cdot d.x) + B \cdot (O.y + t \cdot d.y) + C \cdot (O.z + t \cdot d.z) + D = 0 \\
\implies & t = \frac{-\langle n \cdot O \rangle + D}{\langle n \cdot d \rangle}
\end{aligned}
$$

Finally, if the found $\bar{t}$ value is negative, it means that the intersection point between the ray and the plane is *behind* the ray origin, therefore no intersection is found. Else the ray intersects the plane at point $r(\bar{t})$.

Algorithm A.2 Ray-plane intersection algorithm

function INTERSECT($ray, plane$)
    $d \leftarrow ray.direction$
    $O \leftarrow ray.origin$
    $n \leftarrow plane.normal$
    $P \leftarrow plane.point$
    if $\langle n \cdot d \rangle = 0$ **then** ▷ *Ray is parallel to plane*
        return $\langle false, \_ \rangle$
    $D \leftarrow -\langle n \cdot P \rangle$
    $t \leftarrow \frac{-\langle n \cdot O \rangle}{\langle n \cdot d \rangle}$
    if $t < 0$ **then** ▷ *Intersection point is behind ray origin*
        return $\langle false, \_ \rangle$
    else
        return $\langle true, t \rangle$

A.3. AABB-AABB Intersection

To detect a collision between 2 axis-aligned bounding boxes in the min-max form, it is sufficient to check that there is an overlap between them in all 3 dimensions. By naming the 2 AABBs as A and B we get:

$$\begin{cases} A.min.x \leq B.max.x \\ A.max.x \geq B.min.x \\ A.min.y \leq B.max.y \\ A.max.y \geq B.min.y \\ A.min.z \leq B.max.z \\ A.max.z \geq B.min.z \end{cases}$$

A.4. Frustum-AABB Intersection

In order to detect an intersection between a frustum and an axis-aligned bounding box in the min-max form, we used a simplified version of the separating axis test (a special case of the separating hyperplane theorem) [1]. The simplification comes from the fact that we need to find the intersection of a frustum and an AABB, and not two 3D convex hulls, meaning that we can exploit some assumptions on the direction of the edges of the two objects, as we'll note below.

Before proceeding with the separating axis test, we first try a simpler AABB-AABB collision test, between the given AABB and the AABB that most tightly encloses the frustum. In case this *rejection test* gives a negative answer, we can deduce that the frustum and the AABB are not colliding. Otherwise, we must use the more expensive SAT.

The separating axis theorem in 3 dimensions states that 2 convex hulls are not colliding if and only if there is a plane that divides the space into 2 half-spaces each fully containing one of the two convex hulls.
To find whether such a plane exists, we project the two convex hulls on certain axes, and check whether their 1D projections are overlapping. The theorem also states that if there is an axis where the projections are not overlapping it must be either:

- An axis perpendicular to one of the faces of the convex hulls, or
- An axis parallel to the cross product between an edge of the first convex hull and an edge of the second convex hull.

This consideration makes it possible to use the theorem in a concrete scenario.

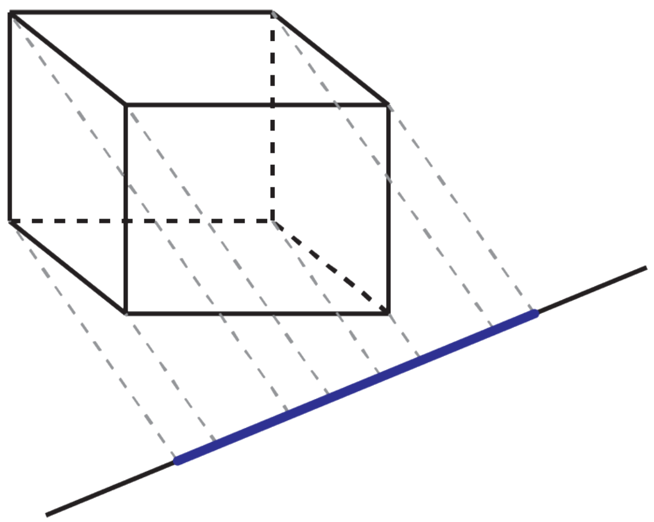
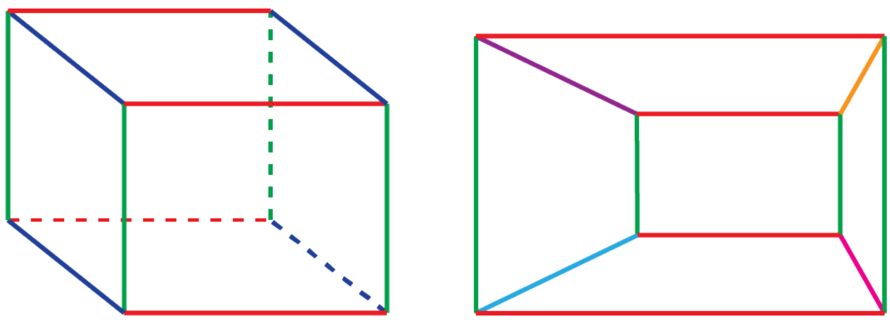


Figure A.2: The projection of an AABB on an axis.

In principle, given 2 polyhedra with 6 faces each (such as a frustum and an AABB), there should be $(6+6)_{normals} + (12 \cdot 12)_{cross\,products} = 156$ axis to check; but, since:

- The AABB has edges only in 3 different directions, and faces normals only in 3 different directions, and
- The frustum has edges only in 6 different directions, and faces normals only in 5 different directions

the number of checks is reduced to $(3+5)_{normals} + (3 \cdot 6)_{cross\,products} = 26$.



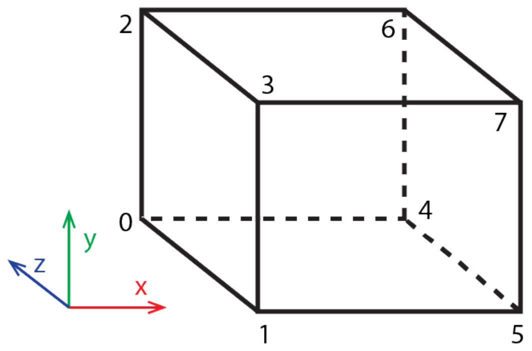
(a) AABB: 3 different edges directions

(b) Frustum: 6 different edges directions

Figure A.3: In the figure the edges having the same direction are colored in the same color.

A.4.1. 1D Projections Overlapping Test

In order to detect if the 1D projections of the 3D hulls are overlapping, we identify the outermost points of each projection (namely $A_{min}, A_{max}, B_{min}, B_{max}$) and check that $B_{min} \leq A_{max}$ & $B_{max} \geq A_{min}$.
For the AABB another optimization is possible, where we detect what points will be the outermost after the projection without actually projecting them, based on the direction of the axis:



(a) AABB vertices layout.

axis direction	extremes
$x \geq 0$ & $y \geq 0$ & $z \leq 0$	1, 6
$x \leq 0$ & $y \leq 0$ & $z \geq 0$	6, 1
$x \geq 0$ & $y \geq 0$ & $z \geq 0$	0, 7
$x \leq 0$ & $y \leq 0$ & $z \leq 0$	7, 0
$x \geq 0$ & $y \leq 0$ & $z \leq 0$	3, 4
$x \leq 0$ & $y \geq 0$ & $z \geq 0$	4, 3
$x \geq 0$ & $y \leq 0$ & $z \geq 0$	2, 5
$x \leq 0$ & $y \geq 0$ & $z \leq 0$	5, 2

Algorithm A.3 Ray-AABB branchless slab intersection algorithm in 3 dimensions

function INTERSECT($frustum, aabb$)
  if $!intersect(frustum.aabb, aabb)$ **then** ▷ *AABB-AABB test*
    return $false$
  $axesToCheck \leftarrow (\perp\ frustum\ faces) \cup (\perp\ AABB\ faces) \cup (\times edges)$
  for all $axis \in axesToCheck$ **do**
    $frustumExtremes \leftarrow findFrustumExtremes(frustum, axis)$ ▷ *Returns the vertices of the frustum that, after the projection, will be the extremes*
    $aabbExtremes \leftarrow findAabbExtremes(aabb, axis)$ ▷ *Same as above, but uses the discussed optimization*
    $A_{min} \leftarrow \langle aabbExtremes.first \cdot axis \rangle$
    $A_{max} \leftarrow \langle aabbExtremes.second \cdot axis \rangle$
    $B_{min} \leftarrow \langle frustumExtremes.first \cdot axis \rangle$
    $B_{max} \leftarrow \langle frustumExtremes.second \cdot axis \rangle$
    if $!(B_{min} \leq A_{max}$ & $B_{max} \geq A_{min})$ **then**
      return $false$
  return $true$ ▷ *If we havent't found any axis where there is no overlap, boxes are colliding*

A.5. Point inside AABB Test

To check if a point P is inside an axis-aligned bounding box in the min-max form, it is sufficient to compare its coordinates with the minimum and maximum of the AABB component-wise:

$$\begin{cases} min.x \leq P.x \leq max.x \\ min.y \leq P.y \leq max.y \\ min.z \leq P.z \leq max.z \end{cases}$$

A.6. Point inside Frustum Test

It is possible to detect whether a point is inside a 3-dimensional frustum by projecting it with the perspective matrix associated with the frustum and then comparing its coordinates, as suggested by [2].

Given the perspective matrix M associated with the frustum, we can project a point P and get: $P' = M \cdot P$; and perform the perspective division.

$P'' = \frac{P'}{P'.w}$ P'' is now in normalized device coordinates (NDC) space, where the frustum is an axis-aligned bounding box that extends from $\langle -1, -1, -1 \rangle$ to $\langle 1, 1, 1 \rangle$ [1].

It is now immediate to see that P is inside the frustum if and only if P'' is inside the AABB (see section A.5).

A simple optimization allow us to avoid the perspective division. Indeed, since in homogeneous coordinates:

$$\langle x', y', z', w' \rangle = \langle \frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'}, \frac{w'}{w'} \rangle = \langle x'', y'', z'', 1 \rangle$$

We can change the inequalities to check whether the point is inside the frustum from:

$$\begin{cases} -1 \leq x'' \leq 1 \\ -1 \leq y'' \leq 1 \\ -1 \leq z'' \leq 1 \end{cases} \qquad \text{to:} \qquad \begin{cases} -w' \leq x' \leq w' \\ -w' \leq y' \leq w' \\ -w' \leq z' \leq w' \end{cases}$$

We created a 2D visual demonstration of how it is possible to detect if a point is inside a frustum at `https://www.geogebra.org/m/ammj5mxd`.

[1] Based on the convention used, it is possible that the AABB in NDC space has a different size. For example, it is common an AABB extending from $\langle -1, -1, 0 \rangle$ to $\langle 1, 1, 1 \rangle$

A.7. Point inside 2D Convex Hull Test

A.8. 2D Hull Culling

List of Figures

List of Tables

List of Symbols

Symbol	Description	Unit
alpha	symbol 1	km

Acknowledgements

Ringrazio...