Ray Distribution Aware Heuristics for Bounding Volume Hierarchies Construction in Ray Tracing

ANONYMOUS AUTHOR(S)

The Bounding Volume Hierarchy (BVH) is a fundamental data structure in the ray tracing algorithms to accelerate the detection of ray-scene intersections. The Surface Area Heuristic (SAH), employed during the construction of the BVH, is based on the hypothesis that the ray distribution in the scene is uniform. In this paper, we show that the SAH hypothesis is not valid when importance sampling is used, and propose two novel heuristics. With the Projected Area Heuristic (PAH) we demonstrate how it is possible to estimate better the cost of a BVH. In particular, we replace the approximation of the probability an AABB is hit by a ray, from the ratio between the surface areas of the node and the root of the BVH, to the ratio of their projected areas. The plane the AABBs are projected on and the kind of projection (either orthographic or perspective) are chosen based on the local ray distribution. With the Splitting Plane Facing Heuristic (SPFH) we show how we can build higher-quality BVHs by taking into account the knowledge of the ray distribution in a local region of the scene during the selection of the orientation of the plane used to split a node.

CCS Concepts: • **Computing methodologies** → **Ray tracing**.

Additional Key Words and Phrases: Ray tracing, Bounding volume hierarchy, Surface area heuristic, Projected area heuristic, Splitting plane facing heuristic, Monte-Carlo importance sampling

ACM Reference Format:
Anonymous Author(s). 2024. Ray Distribution Aware Heuristics for Bounding Volume Hierarchies Construction in Ray Tracing. 1, 1 (July 2024), 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Surface Area Heuristic (SAH, [Goldsmith and Salmon 1987; MacDonald and Booth 1990]) is the cost function used in most of the state-of-the-art BVH builders. Given the root node of the BVH and a node N containing t triangles, the cost associated by SAH to N is computed as:

$$cost_{SAH}(N) = \frac{surface\ area(N)}{surface\ area(root)} \cdot t \tag{1}$$

The core idea behind SAH is to try to minimize the number of intersection tests that must be carried out while traversing the BVH, to find the first intersection between the ray and the scene. To achieve this result, the first term of the formula attempts to estimate the probability that a random ray hits the node N. Such probability is expressed as the ratio of the surface areas of the node and the root of the BVH. The Surface Area Heuristic is based on some hypotheses:

- All the rays are infinitely long and never intersect a primitive of the scene;
- All the rays start outside the scene;
- All the rays hit the root of the BVH;
- The rays follow a random uniform distribution in the 6-dimensional[1] space of the scene.

In this paper we challenge the last hypothesis of SAH, and introduce the theory of two novel heuristics that take into account non-uniform ray distributions in the scene. We then briefly present a customizable framework written in modern C++ where to test the proposed heuristics, and, eventually, show some metrics on their performance and accuracy. The C++ framework and its documentation can be found in the supplemental material.

2 RELATED WORK

Papers related to the design of heuristics where one of the four SAH hypotheses is not considered valid have been quite numerous in the past, as well as articles concerned with the interaction between the Surface Area Heuristic and other techniques often used in ray tracers.

In 2015 Gu, He and Blelloch in [Gu et al. 2015] proposed a method to modify the topological structure of a given BVH to make it more suited to be traversed by the rays belonging to the distribution present in the scene. In order to do so, they cast a small amount of rays to collect information about the intersections between them and the nodes of the BVH, and then decide whether a node should be further subdivided or merged with its sibling. Our approach is different, inasmuch we attempt to leverage the presence of local ray distributions in the scene to directly build the BVH.

The approach adopted by Gu et al. takes inspiration from [Bittner and Havran 2009], where Bittner and Havran use the information gathered from the traversal of the scene via a Representing Ray Distribution (RRD) to propose an enhancement to the SAH.

In [Feltman et al. 2012] Feltman, Lee and Fatahalian propose a heuristic, called Shadow Ray Distribution Heuristic (SRDH) taking into account frequent occluders of shadow rays to build higher-quality BVHs.

The last SAH hypothesis is not the only one of interest to researchers. For instance, Fabianowski, Fowler and Dingliana with [Fabianowski et al. 2009] propose a method where the second hypothesis is considered not valid.

Hunt and Mark, with [Hunt 2008], propose a correction to the SAH to take into account the interaction between the SAH and the mailboxing technique often used by BVH builders.

Another work [Hunt and Mark 2008] from Hunt and Mark proposes to split an AABB with planes parallel to a perspective grid, instead of the cartesian planes. The goal of this method is to use a splitting plane that is parallel to the rays in a particular region of the scene, which is a similar objective to the one of our SPFH, even though reached by using a different approach.

[1] 3 spatial dimensions and 3 directional dimensions.

ACM XXXX-XXXX/2024/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In 2013 Alia, Karras and Laine with [Aila et al. 2013] proposed the End-Point Overlapping heuristic (EPO), to show the negative impact of the overlapping between the AABBs of sibling nodes on the BVH traversal performance.

3 SAH LAST HYPOTHESIS FALLS

In the majority of the algorithms of the ray tracing family, Monte-Carlo integration [Caflisch 1998] is used to estimate the reflective term of the Kajiya rendering equation [Kajiya 1986]:

$$\int_{\Omega} BRDF(\bar{p}, \bar{\omega}_i, \bar{\omega}_o) \cdot \cos(\bar{n}, \bar{\omega}_i) \cdot L_i(\bar{p}, \bar{\omega}_i) d\bar{\omega}_i \tag{2}$$

In other words, the Monte-Carlo method is used to approximate the amount of light incoming (L_i) to the point $\bar{p}$ from all the directions belonging to the hemisphere Ω, oriented toward the normal $\bar{n}$ of the surface where $\bar{p}$ lies. For each direction, the incoming light is weighted with the BRDF of the material of the surface, and with the cosine geometry term.

In order to compute how much light is coming to $\bar{p}$ from a given direction $\bar{\omega_i}$, ray tracing algorithms cast probe rays toward the $-\bar{\omega_i}$ direction. If the probe ray hits a light source, it is immediately known how much light it carries; if it hits an object, new probe rays are recursively cast to evaluate the Kajiya rendering equation at the new intersection point, and with outgoing direction $\bar{\omega_i}$.

In order to estimate the incoming light accurately, many probe rays must be cast. Indeed, the error of the Monte-Carlo estimator decreases with the square root of the number of cast rays. For instance, in order to reduce the error by a factor 2, then $2^2 = 4$ times more probe rays must be traced.

An inefficiency in using plain Monte-Carlo is that, by casting the rays in random uniform directions, many of them will not contribute much to the computation of the incoming light. This can happen because a probe ray is cast toward a dark region of the scene, or because the BRDF for that specific ray direction[2] returns a weight close to zero.

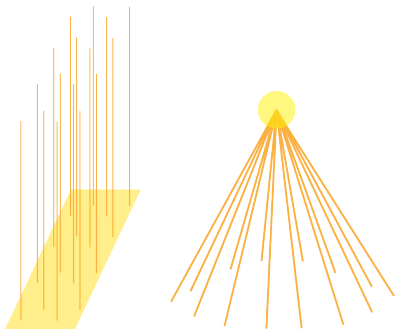
Importance sampling can be used to reduce Monte-Carlo error without increasing the number of probe rays [Kroese and Rubinstein 2012]. With importance sampling, probe rays are cast by following a Probability Density Function (PDF) proportional to the integrand, instead of uniform. In the ray tracing context the integrand is unknown, but it can be approximated. A common approach, called BRDF sampling, consists of approximating it with the BRDF function. Another relevant method is to cast probe rays in such a way that it is more likely for them to be oriented toward light sources [Pharr 2019].

However, if it is more likely that the rays are cast toward light sources, it follows that the ray distribution in the scene is not uniform.

4 INFLUENCE AREAS

In particular, in the proximity to point lights, rays tend to converge to the point in space where the light source is located. This creates a pencil of lines (a radial pattern of rays), as it can be observed in figure 1 (b), which we call *point influence area* in the context of this paper. In the proximity of planar area light sources, rays tend to form a parallel pattern, which we call *plane influence area*.

[2]And for that specific outgoing direction $\bar{\omega_o}$



(a) Parallel rays (b) Radial rays

Fig. 1. Ray distributions in the proximity of light sources.

In the next sections we propose two novel heuristics that are aware of the presence of influence areas in the scene, and leverage their presence to deliver estimates that more closely model what happens in a real-world scenario.

5 PROJECTED AREA HEURISTIC (PAH)

With the Projected Area Heuristic we propose an amendment to the formula of the Surface Area Heuristic (equation 1), in particular to the part concerning the estimation of the node N hit probability:

$$cost_{PAH}(N) = \frac{projected\ area(N)}{projected\ area(root)} \cdot t \tag{3}$$

Instead of considering the ratio of the surface areas of N and the root node, we use the ratio of their projected areas. The area of the AABB of the node can be projected by employing two different projection types, based on the influence area the node is *immersed* into:

Plane influence area In case the node to compute the cost of is placed in a region of the scene where a distribution of parallel rays is present, it is possible to project its AABB to a plane perpendicular to the direction of the rays, by applying an orthographic projection;

Point influence area If the AABB of the node is immersed in a region with a radial ray distribution, it can be projected by computing the perspective projection matrix generated by the pencil of rays.

In figure 2 it is possible to visualize, through a simplified 2-dimensional example, why the projected area of an AABB can be a better proxy for the probability a ray hits it. The two AABBs have the same surface area, therefore, assuming they enclose the same amount of primitives, they are attributed the same cost by SAH. However, as it can be observed, the AABB at the top of the figure is hit by more rays than the bottom one. If we apply the Projected Area Heuristic, and compute the measure of the orthographic projection of both the AABBs on the plane parallel to the rays, the cost associated with each one of the two will better reflect what happens in the concrete real-world scenario.

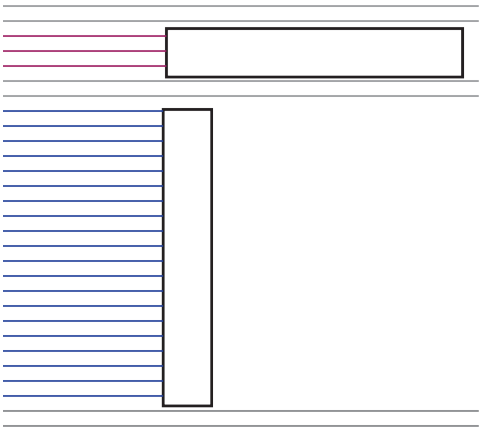


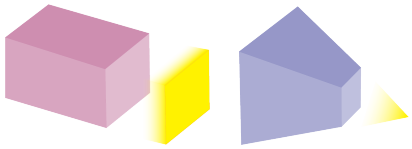
Fig. 2. Two 2D AABBs immersed in a parallel ray distribution.

6 TOP-LEVEL STRUCTURE

Unlike the example of figure 2, in a real-world scene, with many light sources, different ray distributions organically arise in different regions of the scene. It is also likely that, in some areas, no relevant ray distribution is present. This scenario is different from the one where the Surface Area Heuristic operates. Indeed, the space where the SAH works is homogeneous, in the sense that the heuristic can be computed with the same algorithm and with the same parameters for any AABB present in the scene. The Projected Area Heuristic, instead, must be computed in a different way based on the location of the AABB. Furthermore, if no influence area covers a region of the scene, PAH cannot be applied to nodes overlapping that region.

For this reason, if for BVHs built with SAH having a top-level (TLAS) and many bottom-level (BLAS) structures, as described in [Parker et al. 2010], is a performance optimization, when using PAH it becomes mandatory.

We decided to delimit the region of a plane influence area with an Oriented Bounding Box (OBB), and the region of a point influence area with a frustum, as shown in figure 3.



(a) Plane influence area (b) Point influence area

Fig. 3. Enclosures of different influence areas

Each influence area has an associated local BVH, which is built only with the subset of primitives that have at least one point inside the corresponding influence area. This implies that a primitive may end up being included in more than a local BVH, or none. Eventually, a global BVH is built with an algorithm favouring speed over quality, since, in principle, local BVHs should carry out most of the workload.

During the traversal phase, better detailed in section 6.1, a ray first attempts to traverse a local BVH it is *affine*[3] to. If an intersection with a primitive is found, it is guaranteed to be the closest one. Otherwise, the global BVH must be traversed.

It is important to note that it is guaranteed that the closest intersection is always found in the local BVH, only because we adopted a conservative approach while deciding which set of primitives is part of an influence area. Indeed, as stated above, a primitive is included in a certain local BVH if it has *at least one point inside the corresponding influence area*. If, for instance, we opted to consider a primitive part of a BVH based on the position of its barycenter alone, there would have been cases where the closest hit could be missed, as can be observed in figure 4.

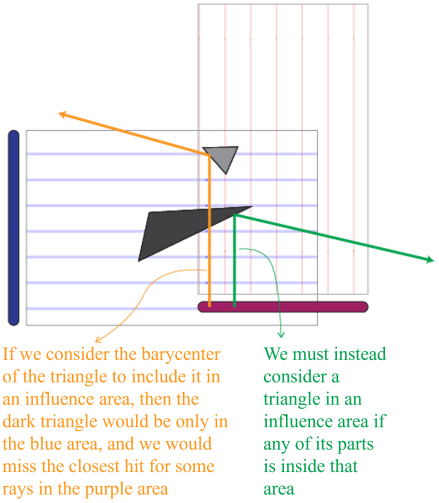


Fig. 4. If a non-conservative approach is adopted while deciding which primitives are included in a local BVH, some intersections may be missed.

6.1 Top-Level Structure Traversal

During the traversal of the BVH, it is first necessary to detect what influence areas are *affine* to the currently traced ray. With the proposition *ray affine to influence area* we require both of these two conditions to be satisfied:

- The ray origin is located inside the enclosing volume of the influence area;
- The ray direction is parallel, up to a tolerance, to the rays of the ray distribution.

The second condition, while intuitive to verify for plane influence areas, where all the rays have the same direction, is not so simple for point influence areas. In this case, indeed, the direction a ray must have for it to be parallel to the rays of the distribution, depends on the position of the ray origin. In order to find out if the ray direction is affine, it is possible to connect the origin of the ray to the focal point[4] O of the frustum, and verify if this line is parallel to the ray, up to a tolerance (figure 5).

[3]The definition of *affinity* is given in section 6.1

[4]The only point where the left, right, top and bottom planes of the frustum all intersect.

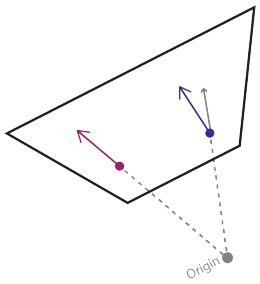
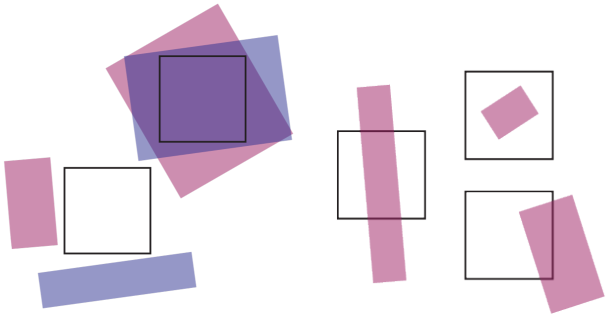


Fig. 5. Visualization of the second affinity condition for point influence areas. Purple ray is affine, while blue one is not.

For what concerns the first affinity condition, it can be checked in two ways. The most immediate one is to perform a point-OBB or point-frustum internal test for all the influence areas present in the scene. While simple to implement, this method does not scale if the number of influence areas increases. The second method is to use a spatial acceleration structure built on the influence areas. In our implementation, we opted to use an octree, whose foundations are explained in [Samet 1988].

6.1.1 Octree Construction. The information stored in the leaves of the octree is the influence areas present in the region of the scene the leaf encloses. By using an octree we approximate the influence areas by subdividing them into parallelepiped regions, conceptually similar to voxels. Therefore, the octree construction algorithm stops further subdividing a node when the 3D region it encloses is homogeneous with reference to the influence areas it contains, or when an arbitrary maximum level is set. In our tests we use 4 or 5 level octrees. Some visual examples can be observed in figure 6, in some cases the Separating Axis Test (SAT, [Eberly 2001]) must be employed, which is expensive.



(a) Leaf (b) Non-leaf

Fig. 6. Some potential cases the octree builder must handle to detect whether a node is a leaf or not.

More details on the implementation of the octree can be found in the code provided in the supplemental material.

7 SPLITTING PLANE FACING HEURISTIC (SPFH)

The second novel heuristic we propose in this paper can be adopted during the phase of the BVH construction where an orientation to cut a node into its two children has to be chosen. A common technique is to choose the axis orientation in a round-robin fashion. Another often adopted method in state-of-the-art BVH builders, as mentioned in [Meister et al. 2021] and [Bauszat et al. 2010], is to split a node into its two children by a plane perpendicular to the longest dimension of its AABB. In this paper, we refer to this heuristic to choose the best splitting direction as Longest Splitting Plane Heuristic (LSPH).

Based on the paper [Aila et al. 2013] from Alia, Karras and Laine, we propose the Splitting Plane Facing Heuristic (SPFH), whose objective is to reduce the overlapping between the projections of siblings nodes of a BVH, by leveraging the knowledge of the influence areas present in the scene.

One of the observations in the paper from Alia, Karras and Laine states that the overlapping between nodes in a BVH is a factor that degrades its traversal performance. The reason behind this is that, if a ray intersects both the children of a given node, then both branches of the BVH must be traversed, in fact negating the advantages of disposing of a hierarchical structure.

In the scenario laid out in this paper so far, we work with the projections of the AABBs of the nodes of the BVH, therefore we are interested in minimizing the overlapping area of the projections of the sibling AABBs.

To achieve this result, the SPFH makes it more likely to choose a splitting plane whose orientation is *the least perpendicular* to the rays present in the influence area where the currently split node is immersed. Indeed, assuming the primitives are uniformly distributed inside the AABB of the node, if it is cut with a plane perpendicular to the rays present in that region of the scene, the projections of the two children will end up covering most of the area of the projection of the father, therefore generating a large overlapping area between them.

If, instead, the cutting plane is *more parallel* to the rays, the overlapping projected area between the children will be reduced.

Another way to look at this is to think that, in case the cutting plane is perpendicular to the rays, the projections of the children will intrinsically tend to overlap, because the nodes are one in front of the other, with reference to the rays' direction (figure 8). Instead, if the cutting plane is parallel to the rays, the children can only overlap because the primitives they are enclosing are not point-like entities, but have an extension. If we reduced the primitives to points, the overlapping would be null.

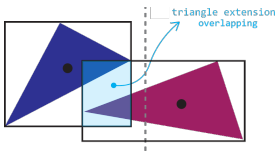
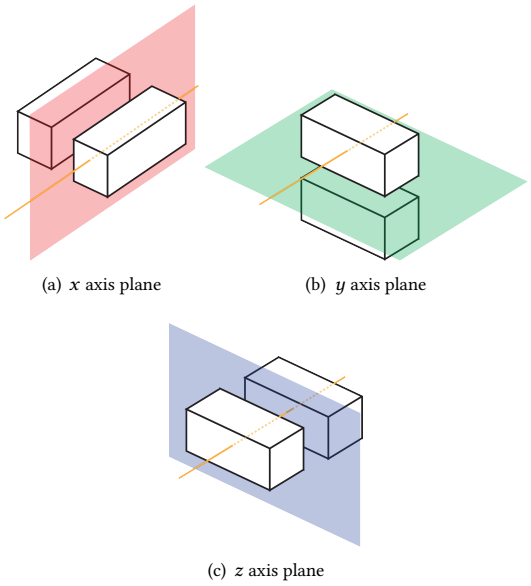


Fig. 7. The AABBs overlap because the triangles have an extension, not because they are one in front of the other.



(a) x axis plane (b) y axis plane

(c) z axis plane

Fig. 8. Visual illustration of the possible splitting plane orientations and how a ray can hit the children's AABBs. In the z axis case, the AABBs are one in front of the other with reference to the ray direction, therefore they produce overlapping projections.

In order to quantify the concept of *how much perpendicular a plane is to a direction vector*, we employed these functions, where $\bar{v}$ is the direction of a ray in the influence area:

$$value\ of\ x\ axis = \frac{|v_x|}{|v_x| + |v_y| + |v_z|}$$

$$value\ of\ y\ axis = \frac{|v_y|}{|v_x| + |v_y| + |v_z|}$$

$$value\ of\ z\ axis = \frac{|v_z|}{|v_x| + |v_y| + |v_z|}$$

First of all, only axis-aligned planes are considered. This shortcut is taken by the majority of the BVH builders, such as [Wald et al. 2014], since, cutting an Axis-Aligned Bounding Box with a non-axis-aligned plane would not be efficient. Then, for each cartesian axis, it is computed how much $\bar{v}$ is facing it compared to the other axes. The values returned by these functions can be seen as how much, in percentage, $\bar{v}$ is facing each cartesian axis.

It is now possible to use these values to find out which axis-aligned plane is less perpendicular to $\bar{v}$. Let us call a plane whose normal is parallel to the axis k the *k-axis plane*. If the $value\ of\ k\ axis$ is large, then it means $\bar{v}$ is *more perpendicular* to the k-axis plane.

Each time our local BVH builder has to choose the orientation of the splitting plane, it computes these three values. Then, based on some customizable thresholds which will be explained in section 8, it decides whether it is worth it to cut the node with a plane with normal parallel to the axis whose value is the smallest one, or if it is better to fall back to using the LSPH.

8 IMPLEMENTATION

The source code of the implementation of our heuristics can be found in the supplemental material, as a link to an online repository. We used this implementation to carry out the tests whose results are presented in section 9. In this section, we will only briefly touch on the most relevant points of the BVH builder, while not mentioning the BVH analyzer and the octree builder, which are implemented with standard techniques and are less relevant in the context of this paper.

In our implementation, we developed a framework in modern C++, where a customizable BVH builder and a BVH analyzer are present. The main goals of our implementation can be summarized in these key points:

Control over speed We haven't aimed at writing the fastest possible BVH builder, by introducing all the state-of-the-art optimizations, or writing a GPU implementation. Our goal is to use it to carry out a theoretical comparison between our heuristics and the state-of-the-art ones, in their base environment.

Customizable We wanted future researchers to be able to use our framework to test different possible variations of our heuristics in a simple way. For this reason, it is possible not only to customize some parameters controlling the building of the BVH, but also to inject custom functions to run in certain phases of the BVH construction, or during the analysis of the BVH and the top-level structure. In the supplemental material it is also possible to look at the documentation of the framework.

8.1 BVH Builder

The algorithm driving the building of the BVH is summarized in the algorithm 1

In the next sections the most relevant procedures of the algorithm will be analyzed.

8.1.1 BVH Builder Parameters. The behaviour of the BVH builder is controlled by a set of customizable parameters, as well as a set of customizable functions. The most relevant parameters are described in this list:

`maxLeafCost` If a node has a cost less than this threshold, it is not further divided and becomes a leaf.

`maxLeafArea` If a node has an area (either projected or surface depending on the cost strategy used) less than this threshold, it becomes a leaf.

`maxLeafHitProbability` If the hit probability of this node is less than this threshold, it becomes a leaf.

`maxTrianglesPerLeaf` If a node has fewer triangles than this threshold, it becomes a leaf.

`maxLevels` Maximum level of the BVH.

`bins` Once a splitting plane orientation is chosen, in our builder we adopt a binned approach to decide how to split a node into two children. This value controls how many splits to attempt. A higher value generates more accurate BVHs, but it is also more expensive.

```
BuildBvh BuildBvh(fatherNode):
    leftNode ← [ ]
    rightNode ← [ ]
    cost ← ∞
    splittingOrientations ← SortSplittingOrientations(fatherNode)
    forall splittingOrientation : splittingOrientation do
        childrenQuality leftarrow EvaluateSplittingOrientation(splittingOrientation, leftNode, rightNode)
        if childrenQuality = SATISFACTORY then
            break
        else if childrenQuality = STANDARD then
            splittingPlanePositions ← ComputeBinnedSplittingPlanesPositions(fatherNode, splittingOrientation)
            forall splittingPlanePosition : splittingPlanePositions do
                [leftTmp, rightTmp] ← SplitPrimitives(triangles, splittingPlanePosition)
                costTmp ← ComputeCost(leftTmp, rightTmp)
                if costTmp < cost then
                    leftNode ← leftTmp
                    rightNode ← rightTmp
                    cost ← costTmp
                end
            end
        else if childrenQuality = FALLBACK then
            use fallback heuristics to find the best split
    end
    if not StopCriterion(cost, leftNode, rightNode) then
        BuildBvh(leftNode.triangles)
        BuildBvh(rightNode.triangles)
    end
```

Algorithm 1: Summarized BVH construction algorithm.

`maxNonFallbackLevels` If a node is located at a level higher than this threshold, the specified fallback strategies (usually SAH and LSPH) will be used to split it, instead of the default ones (usually PAH and SPFH). This threshold can be used to avoid using an expensive strategy even at deep levels, where there is less performance to gain.

`splitPlaneQualityThreshold` $[0, 1]$

`acceptableChildrenFatherHitProbabilityRatio`

`excellentChildrenFatherHitProbabilityRatio` These parameters are described in the next sections.

8.1.2 SortSplittingOrientations. The customizable function `SortSplittingOrientations` must assign to each one of the three possible splitting plane orientations a quality value, and sort them based on this value. In our case, it implements the Splitting Plane Facing Heuristic, which is presented in section 7 of this paper, however the strategy to adopt is fully customizable.

8.1.3 EvaluateSplittingOrientation. The function `EvaluateSplittingOrientation` is responsible for deciding whether it is worth it to attempt to split the node with planes orthogonal to the provided splitting orientation. In order to take this decision, `EvaluateSplittingOrientation` takes into account the quality of the orientation computed by `SortSplittingOrientations`, the cost of the best children nodes found so far, and the custom parameters `splitPlaneQualityThreshold`, `acceptableChildrenFatherHitProbabilityRatio` and `excellentChildrenFatherHitProbabilityRatio`.

The algorithm computes the quality of the best children found up until this point. The quality is computed as the ratio of the hit probabilities of the children's AABBs and the father's AABB. The hit probability is determined by the adopted cost function, in our case it is the Projected Area Heuristic.

$$children\ quality = \frac{HitProbability_{left} + HitProbability_{right}}{HitProbability_{father}} \quad (4)$$

If the quality of the children is better than the threshold `excellentChildrenFatherHitProbabilityRatio`, the algorithm considers the two children as satisfactory, therefore no more splitting orientations should be attempted. Ideally, the sum of the hit probabilities of the children nodes should be lower than the one of the father node, however, this is not always the case, due to the possible overlapping between the two children.

In case, instead, the quality of the children is not excellent, then the quality of the current splitting plane orientation is evaluated. If the orientation has a quality greater than `splitPlaneQualityThreshold`, then the algorithm proceeds, and attempts to cut the father node with this new orientation.

On the contrary, if the orientation is low-quality, the algorithm inspects again the quality of the children nodes. If their quality is greater than `acceptableChildrenFatherHitProbabilityRatio`, then the children nodes are considered satisfactory, and no more splitting orientations are attempted. Last, if their quality is lower than `acceptableChildrenFatherHitProbabilityRatio`, then a fallback method is used to split the node into two children. In our case the fallback method is to employ the Longest Splitting Plane Heuristic and the Surface Area Heuristic.

The diagram shown in figure 9 further summarizes the procedure `EvaluateSplittingOrientation`:

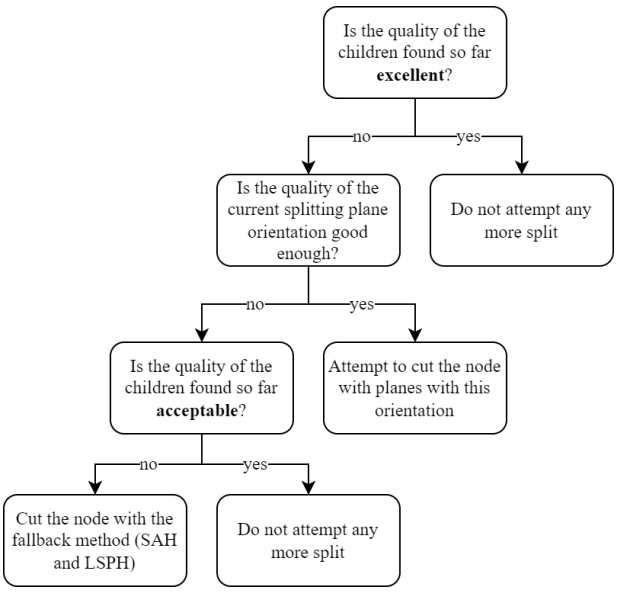


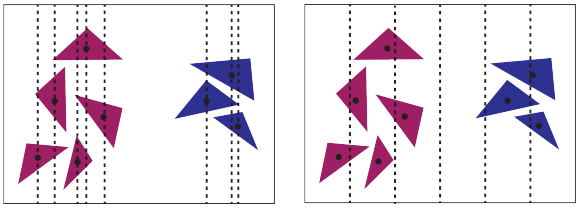
Fig. 9. Decision tree of the `EvaluateSplittingOrientation` procedure.

8.1.4 ComputeBinnedSplittingPlanesPositions. In our builder we adopt a binned approach (figure 10) to select the position of the splitting planes by which to attempt to cut a node into its two children. After selecting the orientation, instead of trying to split a node with a plane passing through the barycenter of each primitive, we divide the length of the AABB in the chosen splitting orientation into a certain amount of regular slices. The amount of bins can be controlled with the `bins` parameter.

8.1.5 SplitPrimitives. Given the collection of primitives and a splitting plane position and orientation, generates the two children nodes. In order to decide whether a primitive is to the *left* or to the *right* of the splitting plane, its barycenter is considered.

8.1.6 ComputeCost. Given two children nodes, it computes their cost by using a customizable cost function. Along with the cost, also the hit probabilities are computed. In our case we set up the Projected Area Heuristic as the standard cost function, and the Surface Area Heuristic as fallback, but they are customizable.

8.1.7 StopCriterion. This customizable procedure decides whether a node is a leaf based on the `maxLeafCost`, `maxLeafArea`, `maxLeafHitProbability`, `maxTrianglesPerLeaf` and `maxLevels` BVH parameters.



(a) Triangles splitting without binning (b) Triangles splitting with binning

Fig. 10. With binning it is possible to lose some cuts, but the most relevant ones are always found, for instance the cut dividing the blue and purple triangles.

8.2 Influence Areas Implementation

In our implementation we defined point and plane influence areas as implementing a general `InfluenceArea` interface. The most important methods of the interface are:

`GetProjectedArea(aabb)` returns the area projected by an AABB, based on the ray distribution associated with the concrete influence area.

`GetProjectedHull(aabb)` returns an array of 2D points, representing the projection of the contour points of an AABB, as seen from the point of view of the influence area.

`IsDirectionAffine(ray)` predicate deciding whether a certain ray is affine to the concrete influence area. The definition of *affinity* is given in section 6.

8.2.1 Plane Influence Area. As described in section 4, a plane influence area is described by an Oriented Bounding Box. In order to implement the `GetProjectedArea(aabb)` and `GetProjectedHull(aabb)` methods, the associated orthographic projection matrix must be computed. It can be computed starting from the eye position, which is at the centre of one of the faces of the OBB, and the extents of the OBB. An arbitrary `worldUp` direction vector must also be chosen, in our case it always points toward the positive y-axis. Given this data, the orthographic projection matrix can be computed with standard formulae.

`GetProjectedArea(aabb)` is implemented by observing that the orthographic projection of an AABB can always be split into 3 parallelograms[5], as shown in figure 11.

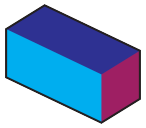


Fig. 11. The orthographic projection of an AABB can always be split into 3 parallelograms.

[5]It can happen that some parallelogram degenerates to segments, having an area of 0.

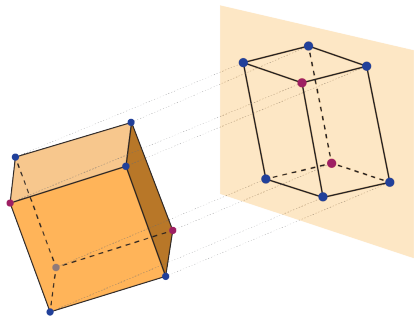


Fig. 12. The projected hull of an AABB. The blue points are contour points, the purple points are internal points.

The algorithm to compute the projected area can therefore be outlined as:

(1) Choose any vertex of the AABB, let us call it vertex A.
(2) Select the adjacent vertices to A, called B, C and D.
(3) Project the 4 vertices to the plane: $A' = viewProjectionMatrix \cdot A$.
(4) Compute the $\overrightarrow{A'B'}$, $\overrightarrow{A'C'}$ and $\overrightarrow{A'D'}$ vectors.
(5) Compute the areas of the three parallelograms via the cross product of the vectors corresponding to the sides: $Area\ parallelogram = |v \times u|$.
(6) Sum the three areas.

For what concerns the `GetProjectedHull(aabb)` method, it is implemented by following some ideas from this paper: [Schmalstieg and Tobler 1999]. First, we define as *contour point*, any point that, after its projection, is part of the perimeter of the hull. Instead, an *internal point* is a point that, after its projection, is inside the hull (figure 12).

In order to compute the projected hull, it is first necessary to detect what points of the AABB will be contour points after the projection, in order to avoid projecting more points (the internal points) than necessary. It can be observed that, given a plane where to project the AABB, the points making up the projected hull can be determined by simply looking at the normal direction to the plane. In particular, it is sufficient to know the sign of each component of the normal direction to detect which faces are visible.

In our implementation, a dictionary-like structure, called `hullTable`, is maintained, which, at each element, stores a list of vertices. The list of vertices stored as elements of the dictionary identifies what are the contour points of a projected AABB if it is observed from a certain direction.

In order to be able to perform a fast look-up of the `hullTable`, the structure is implemented as an array, with a particular indexing rule. The indexing rule states that each position of the array is a binary encoding of the relative position from which the AABB is looked at:

bit	5	4	3	2	1	0
look from	back	front	top	bottom	right	left

For example, at index 9, which in binary is 001001, the projected contour points in the case the AABB is seen from top and left are stored. Of course, in the *hullTable* array there will be some empty positions, since it is impossible, for instance, to look at an AABB from top and bottom at the same time.

With the indexing of the *hullTable* array in place, it is now possible to compute the index of the array where the contour points are stored, only based on the normal direction of the projection plane, as shown in algorithm 2.

FindHullTableIndex `FindHullTableIndex(`*dir*`)`**:**
    i $leftarrow$ 0
    if dir.x > 0 **then** i |= 1
    else if dir.x < 0 **then** i |= 2
    if dir.y > 0 **then** i |= 4
    else if dir.y < 0 **then** i |= 8
    if dir.z > 0 **then** i |= 32
    else if dir.z < 0 **then** i |= 16

Algorithm 2: Given the direction of the normal to the projection plane, returns the corresponding index in the `hullTable`.

8.2.2 Point Influence Area. A point influence area is described by a frustum, as stated in section 4. Since the radial ray distribution enclosed by the frustum can be seen as generated by a pinhole camera, a perspective matrix can be associated with the point influence area. Such a matrix can be computed starting from the horizontal and vertical fields of view of the frustum, and its near and far planes.

Differently from the plane influence area case, there is no shortcut to implement the `GetProjectedArea(aabb)` method. In this case it is necessary to project the contour points and compute the area of the generated hull, by using the algorithm presented in appendix A.1.

The `GetProjectedHull(aabb)` method, is implemented by reusing a similar concept to the one used for plane influence areas. Even in this case, it is necessary to identify the contour points to avoid useless projections, and even in this case it is possible to employ the `hullTable`, but with a different indexing rule. Indeed, for a perspective projection, the relevant bit of information to detect what faces are visible, and therefore which are the contour points, is the relative position between the eye location and the AABB, instead of the viewing direction (figure 13).

For instance, if the AABB is seen from a point of view directly above the centre of the AABB and looking directly down, the projected hull will have as contour points the vertices of the top face of the AABB. If instead the AABB is seen from the top-left, then the contour points will be those of the top and left faces.

The indexing rule is therefore replaced by the one in algorithm 3.

Once the contour points are detected, it is possible to project them with the view-projection matrix computed beforehand.

9 EXPERIMENTAL RESULTS

As already mentioned in the section 8, we used our implementation to perform tests on methods, with the aim of performing a comparative analysis between our heuristics and the state-of-the-art ones.

FindHullTableIndex `FindHullTableIndex(`*eye, aabb*`)`**:**
i *leftarrow* 0
if eye.x < aabb.min.x **then** i |= 1
else if eye.x > aabb.max.x **then** i |= 2
if eye.y < aabb.min.y **then** i |= 4
else if
eye.y > aabb.max.y **then** i |= 8
if eye.z < aabb.min.z **then** i |= 32
else if eye.z > aabb.max.z **then** i |= 16

Algorithm 3: Given the eye position and the AABB, returns the corresponding index in the `hullTable`.

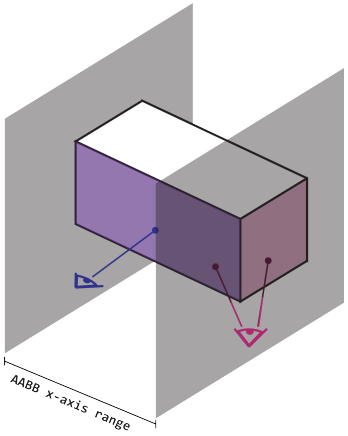


Fig. 13. Considering the x dimension, the blue eye is inside the range of the AABB, whereas the purple one is not. The purple eye can also see the right face.

Our tests have been carried out on 8 scenes[6] with different characteristics:

Random100 100 randomly created triangles in a cube region.
Random1000 1000 randomly created triangles in the same cube region. Many overlaps.
Suzanne The head of a low-poly monkey.
Cottage A cottage in a simple scenery. Comprehends large primitives.
CottageWalls The same scene as above, but enclosed in walls. All the rays hit some geometry.
Woods An house in a wood. Medium poly-count.
Crowd Some detailed people enclosed in walls.
Sponza The classic Sponza model.

In each scene we artificially created an influence area, where the majority of the primitives are immersed. The influence area can be *plane* or *point*, and its rays can be: parallel to a cartesian axis and a cartesian plane, forming a 15° angle with a cartesian axis and parallel to a cartesian plane, forming a 45° angle with two cartesian axes and parallel to a cartesian plane, oblique (not parallel to any axis or plane).

[6]The scene models can be found in the repository in the supplemental materials.

Table 1. The values of the parameters used to build the BVHs during the testing phase.

property	value
`maxLeafCost`	0
`maxLeafArea`	0
`maxLeafHitProbability`	0
`maxTrianglesPerLeaf`	2
`maxLevels`	100
`bins`	40
`maxNonFallbackLevels`	100
`splitPlaneQualityThreshold`	0.4
`acceptableChildrenFatherHitProbabilityRatio`	1.3
`excellentChildrenFatherHitProbabilityRatio`	0.9

With this setup we get to test each scene in 8 different configurations, for a total of 64 tests. Then, we repeated each test with all possible combinations of heuristics: PAH and SPFH, PAH and LSPH, SAH and SPFH, SAH and LSPH. The total amount of tests is therefore 256.

The parameters to build the BVH have been found empirically, as explained in section 9.5, and are listed in table 1.

All the numerical results can be found in the supplemental material.

9.1 PAH Accuracy

The first analysis we performed is a comparison between the error committed by PAH and SAH while estimating the cost of a BVH.

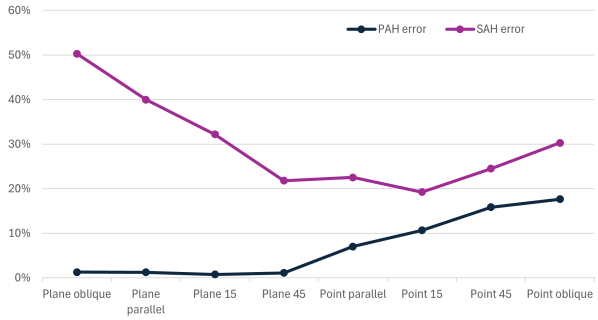


Fig. 14. PAH versus SAH mean error chart.

In order to compute the error, a method to compute the real cost of a BVH is needed. It is implemented in the traversal algorithm: each time the AABB of an internal node is hit, the cost of traversing the BVH with that ray is increased by 2, which is the number of tests that must now be completed on the children of the node. In case a leaf is hit, the cost is increased by the number of triangles contained in the node. In this way, the SAH and PAH cost functions are mimicked, where the hit probability is taken into account by the fact that we are actually traversing the BVH.

We then computed the percentage error as:

$$error\% = \left| \frac{Real\ cost - Estimated\ cost}{Estimated\ cost} \right|$$

In the chart 14 we grouped the results by influence area. It is immediately possible to note that the Projected Area Heuristic is more accurate than the Surface Area Heuristic, in case an influence area is present in the scene. The mean error for plane influence area for PAH is 2%, whereas for point influence areas is 12%.

9.1.1 Projected Area Culling. The projections of the AABBs to the near plane can sometimes exceed the area of the section of the near plane visible to the camera, as it can be observed in figure 15.

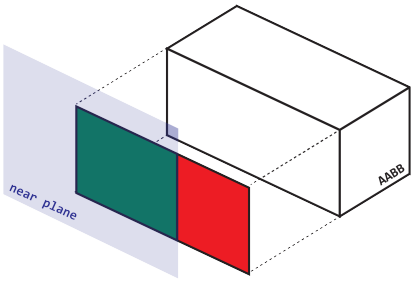


Fig. 15. The 2D projected hull of the AABB is not fully contained in the viewing area of the near plane.

However, the area of the projected hull outside the near plane visible section, cannot, by definition, be intersected by any ray of the influence area. This can potentially lead to screwed estimates of the cost of the BVH. In order to avoid this, it is possible to introduce the projected area culling technique. With this technique, only the set intersection of the projected hull with the visible section of the near plane is considered. The technique is outlined in appendix A.

9.2 PAH and SPFH Building Performance

In order to compare the PAH and the SPFH building performance to the one of the state-of-the-art heuristics, we built the respective BVHs on the exact same primitives, and compared their real traversal cost, computed as explained in the previous section. The results can be appreciated in the chart 16.

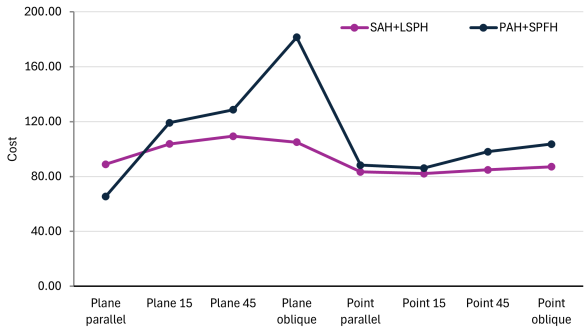


Fig. 16. Cost of BVHs built with PAH and SPFH compared to ones built with SAH and LSPH.

The results of our heuristics combined are better than the state-of-the-art ones only in the case where the plane influence area has rays parallel to a cartesian axis. Given the disappointing results, we decided to analyze each one of the two novel heuristics in isolation.

9.3 PAH in Isolation

The tests performed in this section follow the same methodology as the ones in section 9.2, with the difference that, this time, we only adopted the Projected Area Heuristic, and paired it with the standard Longest Splitting Plane Heuristic.

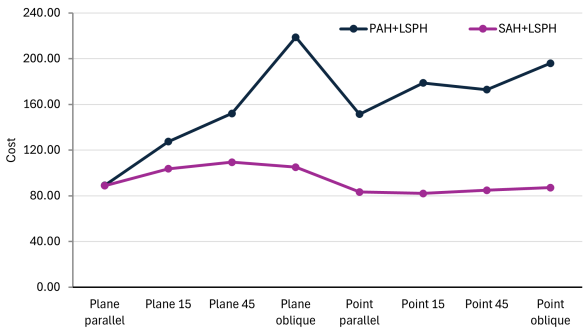


Fig. 17. Cost of BVHs built with PAH and LSPH compared to ones built with SAH and LSPH.

The results in the chart 17 show that the PAH alone, produces BVHs with a worse quality than the SAH, and also worse than using our two novel heuristics combined.

Based on the observation of the BVHs built by the PAH, we hypothesize that the reason behind this behaviour is that, in the cases where it is not possible to find a good splitting plane orientation, and therefore the children's AABBs tend to overlap regardless of the position of the splitting plane, the Projected Area Heuristic always chooses the smallest and the largest possible AABBs. This happens because the algorithm tries to minimize the cost greedily, therefore the projected area weighted by the number of triangles in each node. Since one of the two children's AABBs projections would cover almost the entirety of the father's projected area regardless, so much worth choosing the smallest possible node as its sibling, to minimize their summed cost. This does not happen with the SAH because, working with 3D AABBs, it has one more dimension where to avoid the overlapping of the siblings.

This empirical hypothesis is corroborated by the fact that BVHs built with the PAH tend to have a larger maximum level, symptom that they are not well balanced.

9.4 SPFH in Isolation

The tests performed follow the same methodology as the ones in the previous section, but this time we employ the SPFH along with the SAH for our BVHs.

In this case the results are positive, and the SPFH produces BVHs with a better quality than the LSPH, in the case their nodes are immersed in any influence area. As it is possible to observe in chart 18, using SFPH with influence areas with oblique rays, produces

BVHs of similar quality to the BVHs produced by LSPH. This is expected, since, as explained in the section about the implementation (8.1.3), SPFH falls back to using LSPH in case an acceptable split is not found, and this happens often when the rays are not parallel to any cartesian axis.

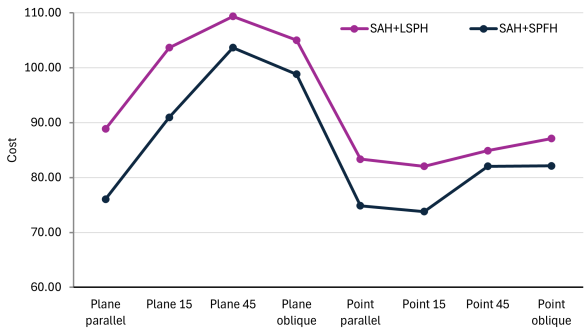


Fig. 18. Cost of BVHs built with SAH and SPFH compared to ones built with SAH and LSPH.

We also measured the average execution time of the computations for the SPFH, compared to the time the LSPH takes. The measurements have been carried out on our system, therefore cannot be exactly replicated. However, we are interested in the comparison between the two heuristics, rather than their absolute values. The timings are shown in the chart 19.

Fig. 19. Average time spent inside the compute splitting plane function for SPFH and LSPH.

The time to compute SPFH and LSPH is the same. However, we measured that, by using the BVH properties presented in table 1, the average amount of splitting plane orientations attempted by the SPFH is 2.1, whereas the LSPH attempted 1.5 different splitting plane orientations on average. The criteria to decide how many splitting plane orientations to employ are explained in section 8.1.3.

9.4.1 Siblings Overlapping. Sibling nodes overlapping is an occurrence that lowers the quality of a BVH fast. If a ray hits a region of the scene where both the children of a node are present, both branches of the BVH must be traversed, negating the advantages brought by the hierarchical structure, as it can be observed in the chart 20. Our heuristic works in the 2-dimensional space where AABBs are projected, therefore we decided to analyze the overlapping of the projections of the AABBs, and evaluate whether the SPFH brings the theorized benefits.

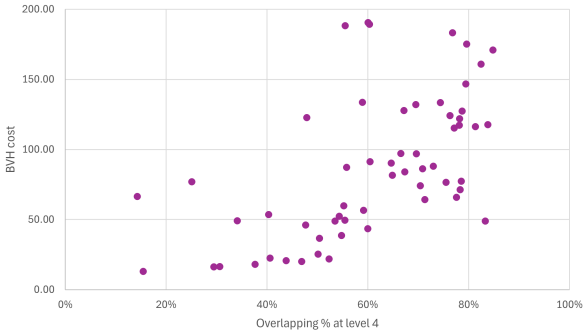


Fig. 20. High overlapping leads to worse BVH cost.

.

The overlapping percentage between siblings has been measured at different depths of the BVH, in particular at levels 4, 7, 10, 15 and 100. The overlapping area can be computed with the same procedure we adopted to perform projected area culling, in section 9.1.1. The overlapping percentage between two AABBs having areas A_1 and A_2 can then be defined as:

$$Node\ overlapping\ percentage = \frac{V}{min(A_1, A_2)}$$

Where V is the area of the intersection shape between the two projected AABBs.

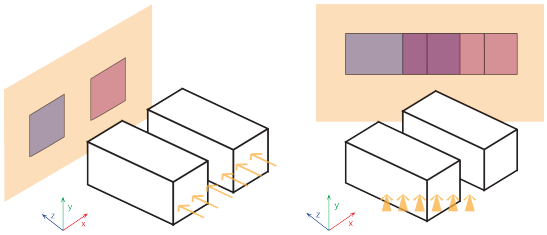
To compute the average overlapping percentage of the whole BVH, it is possible to sum together all the overlapping areas ($V_1 + V_2 + ...$) of each pair of siblings, along with the area of the smallest of the two siblings ($A_{min1} + A_{min2} + ...$), and to compute the total overlapping percentage in the same way as presented above:

$$Total\ overlapping\ percentage = \frac{V_1 + V_2 + ...}{A_{min1} + A_{min2} + ...}$$

From the chart 22 it is possible to note how SPFH (59%) is better than LSPH (66%) at minimizing the overlapping. SPFH works better when rays tend to be parallel to one cartesian axis. This can be associated with two factors:

- As seen in section 9.4, SPFH falls back to LSPH more often in oblique test cases;
- Since BVHs are made up of AABBs, if the rays are parallel to a cartesian axis, it is possible to maximize overlapping reduction, as it can be observed in figure 21.

We also noted that SPFH is more effective at reducing overlapping in the shallow levels of the BVH. This fact can be exploited, and it is therefore possible to use SPFH only on the higher levels of a BVH, and then directly fallback to LSPH, by tuning the `maxNonFallbackLevels` parameter.



(a) Parallel rays (b) Oblique rays

Fig. 21. Overlapping reduction works best if the rays are parallel to a cartesian axis.

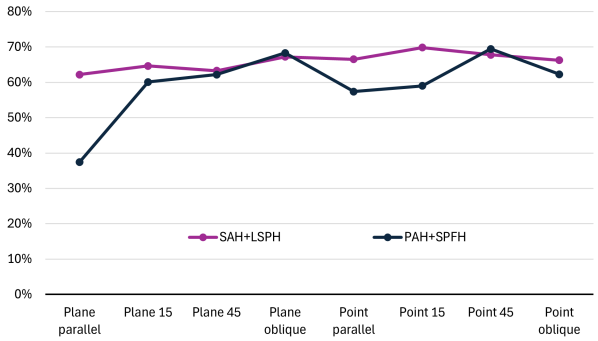


Fig. 22. Overlapping percentage at BVH level 4.

9.5 Parameters Tuning

In this section we will briefly analyze how, by modifying some properties during the building phase of a BVH, the resulting BVH is impacted.

In particular, we will focus on the `splitPlaneQualityThreshold`, `acceptableChildrenFatherHitProbabilityRatio` and `excellentChildrenFatherHitProbabilityRatio` properties, and measure the BVH quality and the amount of nodes generated by using the fallback heuristics instead of the PAH and the SPFH. All the results are presented in an aggregated form, where the average value of all the executed test cases is considered. We will run the analysis on all the test cases where both PAH and SPFH are used, which are 64 in total.

With the base values shown in table 1, we obtain an average BVH quality of 110.5. The amount of attempted splitting planes per node is 1.8. The fallback splitting method, namely using LSPH and SAH, generates the 34% of the nodes[7].

First, we tried to lower the `splitPlaneQualityThreshold`. The result we expect from this change is to build BVHs with a better quality, at the cost of an increased build time. Indeed, with a less strict threshold on the minimum quality of a splitting plane orientation, more orientations will be attempted while trying to divide a node into its two children. The results we obtained fulfill our expectation: the average cost of the BVHs is now 88.4, with an amount of attempted orientations per node of 2.4. Only the 13% of nodes are generated by the fallback method, because, by attempting more splitting plane orientations, it is more likely to find a good split without resorting to the SAH and the LSPH.

We then tried to raise the `splitPlaneQualityThreshold`, and obtained the opposite results. The average BVH cost increased to 148.3, while the amount of attempted splitting plane orientations decreased to 1.5 per node. The fallback method is used to create on the 68% of the nodes.

The next set of tests we performed keeps the `splitPlaneQualityThreshold` constant, while modifying the `acceptableChildrenFatherHitProbabilityRatio` and `excellentChildrenFatherHitProbabilityRatio` properties.

We first tried to decrease the two properties mentioned above to values of 1.1 and 0.7 respectively. We expect that, with a more demanding threshold to consider a split as *good enough*, the quality of the BVH will increase at the cost of having more nodes generated with the fallback heuristics. The experimental results confirm our hypotheses, as the average BVH cost decreased to 69.2, while the fallback method is employed to create the 40% of the nodes. Also, the amount of splitting plane orientations attempted increased to 2.0 per node.

Eventually we raised `acceptableChildrenFatherHitProbabilityRatio` to 1.5 and `excellentChildrenFatherHitProbabilityRatio` to 1.1. Now the average cost of a BVH increased to 124.6, the fallback heuristics generates only the 29% of the nodes. The average amount of attempted splitting plane orientations per node is decreased to 1.7.

In table 2 it is possible to look at all the results described above in a condensed form.

9.6 Memory Footprint Theoretical Analysis

In this section we try to estimate the memory consumption of the objects needed to support our heuristics, and compare them with the memory footprint of a standard BVH. The memory layout of each object can be observed in our implementation, provided in the supplemental material.

9.6.1 Estimate of the Cost of a BVH. In order to estimate the memory occupied by a BVH, it is first necessary to estimate the number of nodes in it. On average the BVHs of the test cases have 20.9 levels, which can be approximated to 21. A balanced binary tree with 21 levels, has $2^{21} - 1 = 2097151$ nodes, half of which are leaf nodes.

In our implementation, a node is represented by a structure holding an `Aabb`, two pointers to the children nodes and a list of triangles. We decided to keep the list of triangles in internal nodes too, for debugging and visualization reasons, even though it is only necessary for leaf nodes. For the sake of this analysis, only the leaf nodes are considered to contain a list of pointers to triangles.

An `Aabb` is represented by a pair of 3D vectors of floating point numbers, therefore it has a weight of $4B \cdot 3 \cdot 2 = 24Bytes$.

Assuming that floating points are stored in 4 bytes, and pointers in 8 bytes, an internal node has a weight of $24B + 2 \cdot 8B = 40Bytes$.

To compute the weight of a leaf node, the size of the array of triangles' indices array must be added. Since, in our test cases, the algorithm stopped when the number of triangles in a node was

[7]The algorithm 1 explains under what conditions the fallback method is used.

Table 2. Summary of the experiments with different BVH properties values. Underlined values are the default ones.

	Case 0	Case 1	Case 2	Case 3	Case 4
splitPlaneQualityThreshold	0.4	0.1	0.8	0.4	0.4
acceptableChildrenFatherHitProbabilityRatio	1.3	1.3	1.3	1.1	1.5
excellentChildrenFatherHitProbabilityRatio	0.9	0.9	0.9	0.7	1.1
Average BVH cost	110.5	88.4	148.3	69.1	124.6
Avg. splitting planes attempted	1.8	2.4	1.5	2.0	1.7
Fallback percentage	0.34	0.13	0.68	0.41	0.29

less than 3, it is possible to assume[8] that all the leaf nodes have 2 triangles. In this case their weight is $40B + 2 \cdot 4B = 48B$.

We can now compute the cost of a balanced BVH with 21 levels:

$$(2^{21-1} - 1) \cdot 40B + 2^{21-1} \cdot 48B \approx 92.27MB$$

9.6.2 Estimate of the Cost of an Influence Area. When a BVH has an associated influence area, its cost must be considered.
A `PlaneInfluenceArea` is made up of a `Plane`, a $4x4$ view projection matrix (used to accelerate the projection process), and an `AabbForObb`, which is an OBB enclosed in an AABB to be used as rejection test for intersections.

The `Plane` holds two 3D floating-point vectors, one for the passing point and one for the normal ($24Bytes$). The `AabbForObb` contains 6 3D floating-point vectors, respectively for the centre, forward, right and up directions, and the half size in all directions. Furthermore, it contains an `Aabb` for the rejection tests: the total cost is $6 \cdot 3 \cdot 4B + 24B = 96Bytes$.

The total weight of a `PlaneInfluenceArea` is:

$$24B + 96B + 4 \cdot 4 \cdot 4B = 184Bytes$$

A `PointInfluenceArea` is made up of a `Pov` and a `Frustum`. The `Pov` contains two 3D floating-point vectors for the viewing direction and position, and 2 floats for the x and y fields of view ($32Bytes$).

The `Frustum` is a bigger structure, because it contains redundant information to improve ray-frustum collision detection. It contains a $4x4$ view-projection matrix, an array of 6 3D directions for the face normals, an array of 6 3D vectors for the edges and 8 3D vertices. Furthermore, it contains an `Aabb` for rejection tests.

The total cost is:

$$32B + 4 \cdot 4 \cdot 4B + 6 \cdot 3 \cdot 4B + 6 \cdot 3 \cdot 4B + 8 \cdot 3 \cdot 4B = 336Bytes$$

The average cost of an influence area is therefore:

$$\frac{184B + 336B}{2} = 260Bytes$$

9.6.3 Estimate of the Cost of an Octree. In order to compute the size of all the structures needed for our heuristics, we first have to make some assumptions on the number of influence areas present in the scene. We will first analyze the case where just one influence area is present, and then the case where there are 20 influence areas.

The cost of the top-level octree can be computed in a similar way to the BVH. An octree with 5 levels is assumed, which is made up of $8^5 - 1 = 32767$ nodes, half of which are leaves.

A node of an octree holds an `Aabb`, 8 pointers to its children and, in case it is a leaf, an array of pointers to the `Bvhs` included in the leaf. It follows that an internal node has a weight of $24B + 8 \cdot 4B = 56Bytes$. A leaf node, assuming it contains just one BVH, has a cost of $56B + 4B = 60B$.

Therefore, the cost of the octree is:

$$(8^{5-1} - 1) \cdot 56B + 8^{5-1} \cdot 60B \approx 1.9MB$$

9.6.4 Putting All Together. In case there is only one influence area covering most of the scene, the total cost of our structures is:

$$\begin{aligned} Total\ cost &= Cost\ local\ BVH + Cost\ influence\ area + \\ &+ Cost\ global\ BVH + Cost\ octree \\ &= 92.27MB + 260B + 92.27MB + 1.9MB \approx 186.44MB \end{aligned}$$

If we used only the global BVH, the cost would have been only $92.27MB$, which means that using our heuristics costs twice as much in terms of memory.

In case there are 20 influence areas, we assume that they contain roughly the same amount of triangles, and that only about 65% of all the triangles in the scene are contained in at least one influence area. Before we assumed that the BVH has 21 levels, is balanced and each leaf contains 2 triangles.

This means that the total number of triangles in the scene is $2^{21-1} \cdot 2 = 2097152$. Only 65% of these are included in a BVH, therefore each BVH approximately contains $\frac{2097152 \cdot 0.65}{20} \approx 65000$ triangles. A balanced BVH with 65000 triangles has a number of levels equal to[9]:

$$\left\lceil \log_2 \left(\frac{65000}{2} \right) + 1 \right\rceil = 16$$

A BVH with 16 levels has a cost of:

$$(2^{16-1} - 1) \cdot 40B + 2^{16-1} \cdot 48B \approx 2.88MB$$

It is now possible to compute the total cost of the structures needed to use our heuristics in this scenario:

$$\begin{aligned} Total\ cost &= (Cost\ local\ BVH + Cost\ influence\ area) \cdot 20 + \\ &+ Cost\ global\ BVH + Cost\ octree \\ &= (2.88MB + 260B) \cdot 20 + 186.44MB + 1.9MB \approx 191.22MB \end{aligned}$$

[8] It is also possible to have a leaf node with more than 2 triangles, in case it was impossible to split due to the relative position of the triangles' centres.

[9] The division by 2 comes from the fact that each leaf contains 2 triangles on average, as we explained above.

Since the 20 BVHs contain a smaller amount of nodes the cost of each BVH is greatly reduced. This relation allows us to use our structures even in cases with many influence areas of modest size.

10 FUTURE DEVELOPMENTS

One important development our work would benefit from, is an improvement of the employed algorithms and data structures. While in this paper we showed the merit of PAH and SPFH from a theoretical point of view, it is now fundamental to test them in more concrete use cases, before continuing the development in other directions.

In order to do so, we suggest looking at the state-of-the-art algorithms used to build BVHs with SAH and LSPH, and adapt them to PAH and SPFH. In principle, many of the optimizations used for standard heuristics should perform equally well for our heuristics too. For instance, Stich et al. [Stich et al. 2009] idea of splitting big triangles can help SPFH in further reducing the overlapping. Tessari et al. [Tessari et al. 2023] suggest building BVHs only on a stochastically chosen subset of the primitives of the scene; despite the final BVH can be slightly lower quality, it may prove extremely useful in the building of the fallback BVH. These are just two examples of techniques that can be applied to our BVH builder, potentially with few modifications.

A different approach is instead needed for the top-level structure. In this case it is not possible to perform a one-to-one translation from existing techniques, even though it may be worth it to study how TLAS and BLAS [Parker et al. 2010] are implemented.

Another important step would be to port the construction and traversal algorithms on parallel architectures. It may be beneficial to do this process in two steps. First, implement a concurrent builder on the CPU, by using a multi-threaded approach. This could be helpful to get insights on what are the most critical points in the change of programming model. Second, transition to a fully parallel builder on the GPU. As shown in [Karras and Aila 2013], building a BVH is a problem that can be effectively resolved by using a parallel approach, and doing so with our heuristics should be no different, except for some considerations on the data layout to make the top-level acceleration structure as efficient as possible, especially during traversal.

In case the previous developments produce acceptable results, the next step to make our techniques operative would be to create a method to detect influence areas autonomously. At the moment, we artificially generate influence areas. While this approach could be used even in a real-world scenario, by adding influence areas where light sources are present, it could bring many improvements if it was possible to detect influence areas by providing as input the rays traced in previous frames. Thanks to this improvement, it would be possible to detect influence areas even in parts of the scene far from light sources, such as near a reflective surface like a mirror. The first method that comes to mind is clustering on the 6-dimensional space of the rays, but even other directions can be explored. In particular, it could be beneficial to try to recycle results from previous frames in order to make the influence area detection algorithms more lightweight and accurate.

Another possible development is to try a different top level structure. For example, one structure that could bring some benefits over the octree is the kd-tree [Fussell and Subramanian 1988].

11 USE CASES AND CLOSING THOUGHTS

Bounding Volume Hierarchies are a key component of ray tracing. Their hierarchical structure is fundamental in accelerating the ray-scene intersection process, which inherently makes ray tracing faster.

Our work is too general to find specific use cases, nonetheless we can summarize the strong points of our techniques, and some possible broad application contexts.

Both the Projected Area Heuristic and the Splitting Plane Facing Heuristic work best when the rays of the influence area are parallel to a cartesian axis. Since this property is known as soon as an influence area is discovered, it is possible to decide on-the-fly whether it is worth it to use our heuristics or not. For instance, in a rendering scenario where there is a static light source, it is possible to immediately decide whether it is appropriate to generate an influence area for it. In case of a dynamic light, things can become more difficult, as it would require to decide on a per-frame basis whether it is worth it to use the BVH associated with the influence area or not. In a real-time scenario this could prove too expensive.

The Splitting Plane Facing Heuristic has proven to work well, without being more expensive to use than the state-of-the-art LSPH, therefore it may be worth it to use it both in real-time and non-real-time scenarios. The drawback of adopting SPFH is that it can be effectively employed only in specific regions of the scene, where influence areas are present, meaning that it is necessary to build the underlying top-level acceleration structure (section 6). This fact can hinder the use of SPFH in a real-time scenario, especially for dynamic use cases, such as dynamic lights.

Without carrying out further tests or introducing the enhancements suggested in section 10, we think that our heuristics are more suited for non-real-time use cases, where a faster-to-traverse BVH can reduce the rendering time even more effectively. This could change in the future.

Our work, while it has been tested on concrete scenes, is mostly a theoretical one. We showed how the novel heuristics we proposed can work in some specific scenarios. However, we think that the way to actually adopt them on concrete use cases is realistically long, but, given the potential benefits we showed they can bring, we believe it is worth it to travel it.

REFERENCES

Timo Aila, Tero Karras, and Samuli Laine. 2013. On quality metrics of bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (Anaheim, California) *(HPG '13)*. Association for Computing Machinery, New York, NY, USA, 101–107. https://doi.org/10.1145/2492045.2492056

Pablo Bauszat, Martin Eisemann, and Marcus A Magnor. 2010. The Minimal Bounding Volume Hierarchy.. In *VMV*. 227–234.

Jiří Bittner and Vlastimil Havran. 2009. RDH: ray distribution heuristics for construction of spatial data structures. In *Proceedings of the 25th Spring Conference on Computer Graphics* (Budmerice, Slovakia) *(SCCG '09)*. Association for Computing Machinery, New York, NY, USA, 51–58. https://doi.org/10.1145/1980462.1980475

Russel E Caflisch. 1998. Monte carlo and quasi-monte carlo methods. *Acta numerica* 7 (1998), 4–8.

David Eberly. 2001. Intersection of convex objects: The method of separating axes. *WWW page* (2001), 2–3.

Bartosz Fabianowski, Colin Fowler, and John Dingliana. 2009. A Cost Metric for Scene-Interior Ray Origins.. In *Eurographics (Short Papers)*. 49–52.
Nicolas Feltman, Minjae Lee, and Kayvon Fatahalian. 2012. SRDH: specializing BVH construction and traversal order using representative shadow ray sets. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics* (Paris, France) *(EGGH-HPG'12)*. Eurographics Association, Goslar, DEU, 49–55.
Donald Fussell and Kalpathi R Subramanian. 1988. *Fast ray tracing using kd trees*. University of Texas at Austin, Department of Computer Sciences.
Jeffrey Goldsmith and John Salmon. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications* 7, 5 (1987), 14–20.
Yan Gu, Yong He, and Guy E Blelloch. 2015. Ray specialized contraction on bounding volume hierarchies. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library, 309–318.
Warren Hunt and William R Mark. 2008. Ray-specialized acceleration structures for ray tracing. In *2008 IEEE Symposium on Interactive Ray Tracing*. IEEE, 3–10.
Warren A Hunt. 2008. Corrections to the surface area metric with respect to mail-boxing. In *2008 IEEE Symposium on Interactive Ray Tracing*. IEEE, 77–80.
James T. Kajiya. 1986. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86)*. Association for Computing Machinery, New York, NY, USA, 143–150. https://doi.org/10.1145/15922.15902
Tero Karras and Timo Aila. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference* (Anaheim, California) *(HPG '13)*. Association for Computing Machinery, New York, NY, USA, 89–99. https://doi.org/10.1145/2492045.2492055
Dirk P Kroese and Reuven Y Rubinstein. 2012. Monte carlo methods. *Wiley Interdisciplinary Reviews: Computational Statistics* 4, 1 (2012), 113–118.
Younhee Lee and Woong Lim. 2017. Shoelace formula: Connecting the area of a polygon and the vector cross product. *The Mathematics Teacher* 110, 8 (2017), 631–636.
J David MacDonald and Kellogg S Booth. 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer* 6 (1990), 153–166.
Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J Doyle, Michael Guthe, and Jiří Bittner. 2021. A survey on bounding volume hierarchies for ray tracing. In *Computer Graphics Forum*, Vol. 40. Wiley Online Library, 683–712.
Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. In *ACM SIGGRAPH 2010 Papers* (Los Angeles, California) *(SIGGRAPH '10)*. Association for Computing Machinery, New York, NY, USA, Article 66, 13 pages. https://doi.org/10.1145/1833349.1778803
Matt Pharr. 2019. On the Importance of Sampling. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs* (2019), 207–222.
Hanan Samet. 1988. An overview of quadtrees, octrees, and related hierarchical data structures. *Theoretical Foundations of Computer Graphics and CAD* (1988), 51–68.
Dieter Schmalstieg and Robert F Tobler. 1999. Fast projected area computation for three-dimensional bounding boxes. *Journal of graphics tools* 4, 2 (1999), 37–43.
Martin Stich, Heiko Friedrich, and Andreas Dietrich. 2009. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (New Orleans, Louisiana) *(HPG '09)*. Association for Computing Machinery, New York, NY, USA, 7–13. https://doi.org/10.1145/1572769.1572771
G Taylor. 1994. Point in polygon test. *Survey Review* 32, 254 (1994), 479–484.
Lorenzo Tessari, A Dittebrand, Michael J Doyle, and Carsten Benthin. 2023. Stochastic Subsets for BVH Construction. In *Computer Graphics Forum*, Vol. 42. Wiley Online Library, 255–267.
Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: a kernel framework for efficient CPU ray tracing. *ACM Trans. Graph.* 33, 4, Article 143 (jul 2014), 8 pages. https://doi.org/10.1145/2601097.2601199
Y-K Zhu, J-H Yong, and G-Q Zheng. 2005. Line segment intersection testing. *Computing* 75 (2005), 337–357.

A HULLS INTERSECTION COMPUTATION AND AREA

In order to find the overlapping region between two 2D hulls in 2-dimensional space, we can proceed as outlined in the diagram 23:

To find out what vertices of a hull are inside the other one, it is possible to loop over them and adopt the point inside convex hull test, such as the one presented in [Taylor 1994].

To detect an intersection between two segments, the algorithm shown in [Zhu et al. 2005] can be employed.

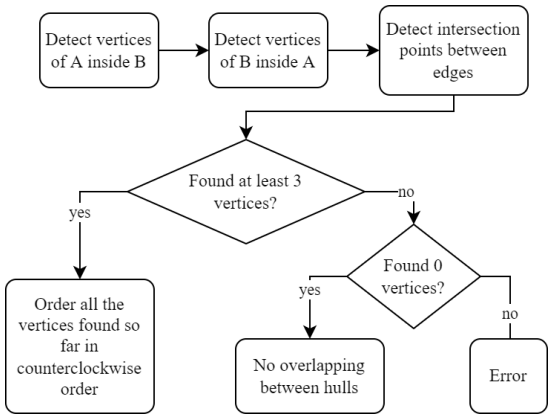


Fig. 23. General algorithm to find the overlapping region between two convex hulls called A and B.

After finding the internal and intersection points of the two convex hulls, the obtained set must be ordered counterclockwise (or, arbitrarily, clockwise), so that two consecutive vertices form an edge of the convex hull.

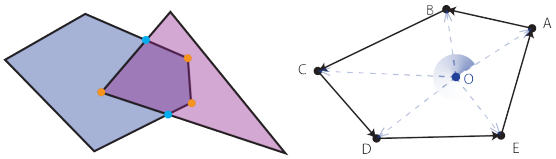


Fig. 24. Yellow vertices are found in the first 2 steps (vertices inside), whereas light blue vertices are found in the third step (edges intersections). Vertices are then sorted counterclockwise by using the `atan2` function.

To do so it is possible to compute the barycenter O of the set of points. Given that the intersection of two convex hulls is necessarily convex, we know for certain that O will be inside the convex hull formed by the vertices we found so far.

Now, for each vertex A_k it is possible to calculate the vector $\overrightarrow{OA_k}$, and sort the vertices based on $atan2(\overrightarrow{OA_k}_y, \overrightarrow{OA_k}_x)$. The `atan2`$(v_y, v_x)$ function returns the angle between the positive x-axis and the vector $v = \langle v_x, v_y \rangle$. Differently from the arctangent function, the returned angle ranges in the interval $(-\pi, \pi]$, therefore it is well suited for the purpose of sorting the convex hull vertices.

A.1 2D Hull Area Computation

To calculate the area of a 2-dimensional hull, the Gauss's area formula, also known as the shoelace formula [Lee and Lim 2017], can be employed.

Given a polygon with vertices $P_0, P_1, ..., P_n$, where each vertex has coordinates: $P_k = (x_k, y_k)$, its area can be found with this formula:

$$Area = \frac{1}{2} \cdot \left| \left(\begin{vmatrix} x_0 & y_1 \\ y_0 & y_1 \end{vmatrix} + \begin{vmatrix} x_1 & y_2 \\ y_1 & y_2 \end{vmatrix} + ... + \begin{vmatrix} x_{n-1} & y_n \\ y_{n-1} & y_n \end{vmatrix} + \begin{vmatrix} x_n & y_0 \\ y_n & y_0 \end{vmatrix} \right) \right|$$

$$= \frac{1}{2} \cdot \left| \sum_{i=0}^{n} (x_i \cdot y_{i+1} - y_i \cdot x_{i+1}) \right| \quad where\ P_0 = P_{n+1}$$