Scene Rendering with (Randomized) Binary Space Partition Trees

Andrew Barbarello

December 5, 2013

Contents

1 Problem Statement **1**

2 z-buffering **2**
- 2.1 An example of z-buffer rendering . . . 3
- 2.2 Analysis . . . 3

3 Binary Space Partition Trees and the Painter's Algorithm **5**
- 3.1 BSP trees . . . 6
- 3.2 Using BSP trees for rendering with the Painter's Algorithm . . . 7
- 3.3 BSP trees via randomized incremental construction . . . 7
- 3.4 Example . . . 8
- 3.5 Analysis . . . 10

4 Conclusion **11**

1 Problem Statement

Hidden surface determination is a fundamental problem in computer graphics. Typically, we are given a description of a "scene" as a collection S of planar polygons (or, most often, a collection of *triangles*) in $\mathbb{R}^3$, along with a view point E, a viewing direction, and a field of view, which determine a *view plane* V. We want to compute the *image* of the scene as projected onto the view plane, in the same way a camera projects the world onto an image plane, by determining which surfaces should be visible from the given view and drawing their projection onto the view plane.

A point P on a surface $s \in S$ is said to be *visible* from E if the line segment $\overline{EP}$ from E to P does not intersect any other surface $s' \in S$, and passes through the view plane. We make the simplifying assumption that all surfaces are opaque, so that we need not worry about partial occlusion by transparent objects. To render a scene we must first answer the "visibility" question: for each surface $s \in S$, which points of s are visible from E? If we can answer this question, we can determine which polygonal edges of the surfaces in S should be drawn on the view plane (specifically, a discretization of the plane into "pixels" represented by a two dimensional array), and subsequently fill pixel regions with colors determined by the polygonal edges that bound them [4].

Of course, this is a highly simplified presentation of the rendering process; the final pixel "intensities" (color/gray values) in a rendered frame are typically determined by a number of factors including the position of light sources, surface reflectance properties, and so on. We will *further* restrict our attention to a lower dimensional version of the problem: our space will be a region of $\mathbb{R}^2$, the set S will consist entirely of *non-intersecting* (except possibly at the end points) *line segments*, and we will be projecting the objects in our scene onto a one dimensional *view line* (see figures 1 and 2). [1] This two dimensional

[1]Note that the requirement that our segments be non-intersecting except at the endpoints is not too restricting: if two segments intersect, we can split the pair into four non-intersecting subsegments using the point of intersection as a fifth vertex.

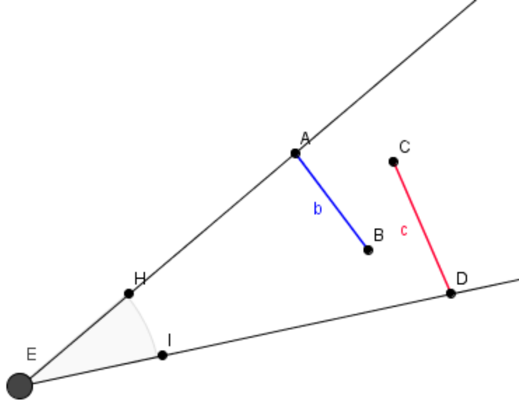


Figure 1: A two dimensional scene consisting of the two lines $S = \{b, c\}$, with view point E, and *view line* $\overline{HI}$

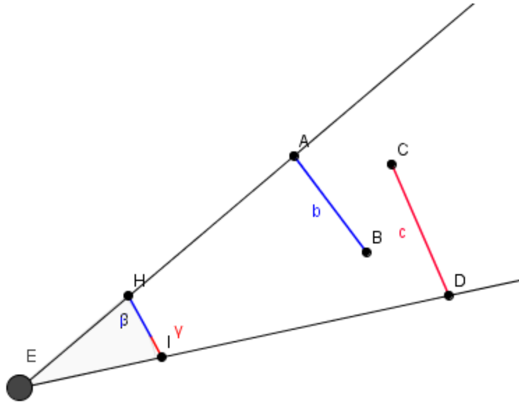


Figure 2: Scene from figure 1, where the view line $\overline{HI}$ is colored by the projections of the lines $\{b, c\}$ onto it.

problem setup will be sufficient to introduce a pair of algorithms for quickly determining which scene surfaces should occupy the viewing surface. First, we will look at a deterministic, simple, and widely used algorithm called z-buffering. Then, we consider a rendering scheme using a spatial data structure called a *Binary Space Partition* (BSP) tree that admits a straightforward construction by a randomized algorithm. [2] While our $2d$ restriction will make algorithm explanations simple, the algorithms can be extended to $3d$ in a straightforward manner.

2 z-buffering

A point P on a surface in the scene is visible from the view point E if its *distance* to E is the smallest of the distances of E to the intersections of all surfaces in S with the line $\overline{EP}$. We call the distance of a point in the scene to E the *depth* of the point, which is roughly equal to the point's distance to E in the direction *orthogonal* to the view surface. In $3d$ rendering, we typically consider the view plane to be contained in the $z = 0$ plane, and all points beyond the view plane from E to have positive z value, hence the rough correspondence between *depth* and z-value.

Consider the scene depicted in figure 2. The key to projecting the segments $\overline{AB}, \overline{CD}$ onto the view line $\overline{HI}$ is to recognize that $\overline{AB}$ has lesser depth values than $\overline{CD}$ in the region where they overlap as viewed from E, and so $\overline{CD}$ "dominates" this region. We can draw the projections in any order, so long as we never overwrite a projection of a segment of smaller depth with one of larger depth. This is the

[2]The presentation of rendering with z-buffering as an alternative to using BSP trees is also given in [1] .

reasoning behind the z-buffering algorithm, Algorithm 1.

In this algorithm, we compute the projected points of all line segments in arbitrary order. Before we fill the pixel a point projects to with the color of its line, we check that the point has smaller depth than that of the point we previously projected to the pixel.

Algorithm 1 Algorithm for rendering segments in S onto x axis-aligned view segment $(x_{\min}, x_{\max})$. We assume this interval is partitioned into N discrete intervals ("pixels") of width Δx, and that we have two arrays of length N (a frame buffer `fbuf` and a z buffer `zbuf`)

for $i = 1, \ldots, N$ **do** ▷ initialization
  `zbuf[i]` $\leftarrow \infty$
end for
for each line segment $\overline{AB}$ in S, in arbitrary order **do**
  for each $i = 1, \ldots, N$ in the discretization of the view segment **do**
    $d \leftarrow$ distance of intersection of $\overline{EC}$ with $\overline{AB}$ from E, where $C = (x_{\min} + i\Delta x + \frac{\Delta x}{2}, 0)$ is the mid point of the discrete cell of the view segment. If there is no intersection, $d \leftarrow \infty$
    if $d <$ `zbuf[i]` **then**
      `fbuf[i]` $\leftarrow$ color of segment $\overline{AB}$
      `zbuf[i]` $\leftarrow d$
    end if
  end for
end for

2.1 An example of z-buffer rendering

To give an example of the z-buffer algorithm in action, we will use it to draw the one-dimensional projection of the scene in figure 1 into a "frame buffer". This process is depicted in figure 3. Whereas algorithm 1 describes a "ray casting" approach for filling each pixel by casting a ray through it into the scene, we will take the common "rasterisation" approach of simply projecting the *end points* of our line segments onto the view segment, and filling in the space between the transformed vertices appropriately [2].

Assume the x-axis is aligned with the view segment $\overline{HI}$. We begin with an empty frame buffer, and a z-buffer initialized to ∞ (first row of figure 3). The view segment is discretized into 10 subsegments (represented by the cells of the frame buffer and corresponding z-buffer). We arbitrarily select $\overline{CD}$ as the first line to consider. The rays from E through the endpoints C and D intersect $\overline{HI}$ in the points c', I, which are discretized to the 3rd and 9th cells of the frame/z buffers. These end points are at depths $6.4, 6.3$, and the depth of every intermediary point is given by the interpolation between these two values. Since all these depth values are smaller than ∞ (the depth values currently stored in the corresponding cells of the z buffer), we set all cells of the frame buffer between c' and I to the color of $\overline{CD}$, and update the z buffer (row 2 of the figure).

Next we consider the segment $\overline{AB}$. The endpoints A and B project to points H, b' on the view segment (pixels 0 and 6). For each of these pixels and the pixels between them, we check that the depth of the point on the line projecting to that pixel is less than that in our buffer. This happens to be the case for all pixels in the projection, and so we fill them all in our frame buffer and update our z-buffer accordingly. Since we have no other segments to consider, we are done, and the current state of the frame buffer is the one dimensional rendering of our scene.

2.2 Analysis

The z-buffer rendering algorithm has the advantage of being very simple to describe and implement; no preprocessing is required and surfaces in the scene can be considered in arbitrary order. Furthermore, modern graphics processing units reserve memory for a z-buffer and provide hardware-accelerated depth testing while rendering, making this brute-force approach very fast and thus widely adopted in practice [2].

The method has a distinct disadvantage of doubling the memory requirement needed to render a frame: to render an image of N pixels, we require a N pixel z-buffer. In order to mitigate the cost of

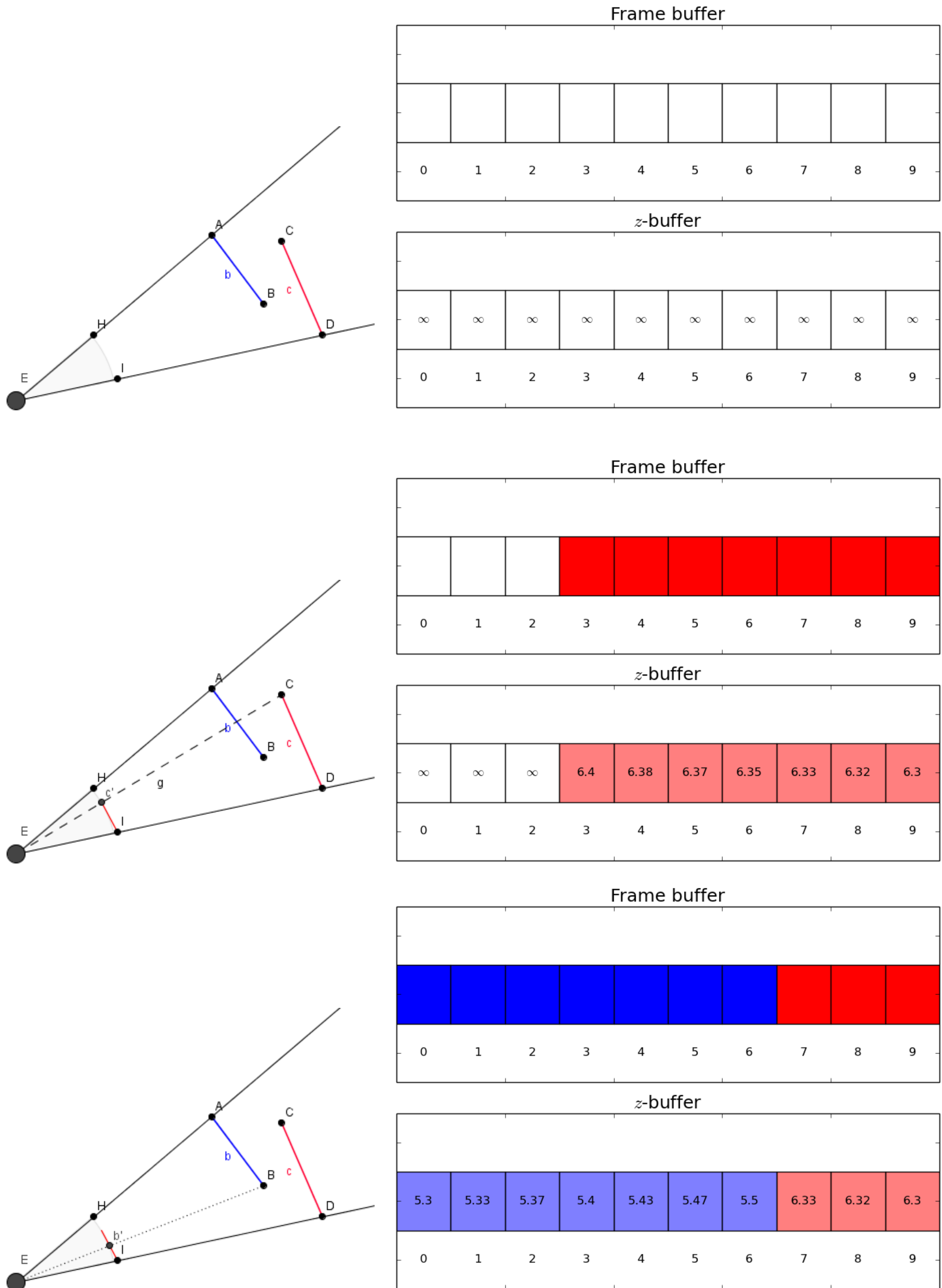


Figure 3

accessing two buffers prior to drawing any pixel, we hope to keep both buffers in GPU memory. If we wish to a render an image of larger size than we have sufficient memory for in the GPU, we will need to render it in CPU memory, which will be accompanied by a significant sacrifice of hardware acceleration. Early interactive graphical applications like *DOOM* (1993) could not rely on widespread adoption of computer graphics hardware acceleration, and instead used the technique we describe in section 3 [5].

An additional drawback of the z-buffering approach is that it naively considers all input line segments while rendering. Consider the scene depicted in figure 4. While rendering this scene, the z-buffer algorithm will perform a lot of unnecessary work projecting the end points of segments $\overline{AB}$ and $\overline{JK}$ to the viewing segment and computing their depths only to find that all projected points will be overshadowed by the segment $\overline{CD}$. Here we are assuming that no segment is transparent, since the z-buffer algorithm cannot properly render transparent objects by itself: in order for these objects to be rendered appropriately one must first render objects *behind* the transparent object, then the object itself. This requires that an ordering be imposed on segments, which the z-buffer method alone does not provide. Thus we see that this method does not exploit the potential performance gain from an assumption that all objects are opaque, yet still needs additional help to properly render transparent segments.

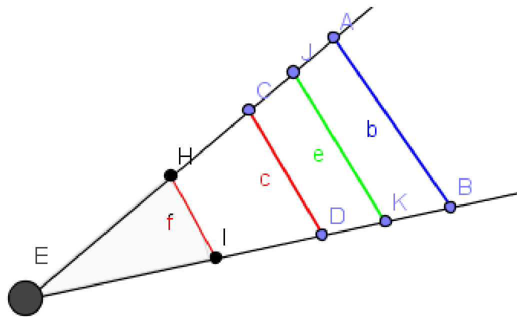


Figure 4: An example of a scene for which considering all line segments when rendering is unnecessary; the segment $c = \overline{CD}$ occludes all other segments.

3 Binary Space Partition Trees and the Painter's Algorithm

The discussion of the needless drawing performed by the z-buffer algorithm, as well as its inability to handle transparent objects, motivates taking the approach of establishing a "priority" ordering on our line segments before rendering them. That is, we seek to sort the segments in S in a linear order such that if a segment p precedes a segment q in this ordering, then q does not overlap p when viewed from E [4] .[3] Given such an ordering we can simply draw the segments in the scene directly to the framebuffer one after the other, in increasing priority order, without performing any depth checks or maintaining an extra buffer. If a segment p occludes q when viewed from E, then q will be drawn to the framebuffer first, and any part occluded by p will be overwritten when drawing p. Furthermore, transparent objects will be rendered correctly when performing the rendering in this order. Alternatively, if we assume that segments are all opaque, then we can simply draw the segments closest to the viewpoint until the entire framebuffer is filled. Using this scheme, we could draw the line in figure 4 and then stop before considering all the lines behind it.[4]

The algorithm just described for rendering a scene in increasing priority order is called the "Painter's algorithm". By using the data structure described in section 3.1 we can compute a priority ordering of the line segments in our scene from any viewpoint rapidly.

[3] Our assumption that the segments in S are non-intersecting means such an ordering will always exist (though not necessarily be unique). In three dimensions this assumption is somewhat trickier to make, and may require "splitting" polygons to disambiguate the depth ordering.

[4] The renderer for *DOOM* worked in this way; since a level could be composed of many line segments, it first computed an ordering of segments using a BSP tree (as described in section 3.2), and then drew only the closest 256 line segments to the player, clipping segments of lower priority so that they would not overwrite those of higher priority [5]

3.1 BSP trees

Given a set S of line segments in the plane $\mathbb{R}^2$, a binary space partition (or binary *planar* partition, for this specific case), is a partition of the plane by lines such that each region in the partition contains at most one segment $s \in S$ (or a subsegment of such a segment) [3]. Each partitioning line determines two halfspaces (two children of a binary tree node corresponding to the partitioning line), which are further partitioned until the desired condition is met. Figure 5 depicts such a partition of a set of three line segments.

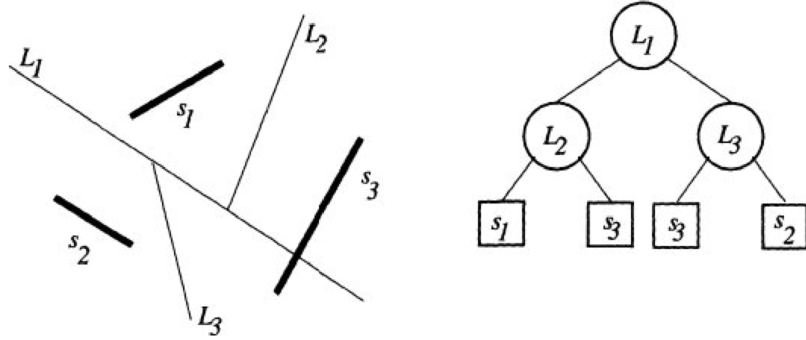


Figure 5: A figure from [3] depicting a BSP for three line segments (dark lines) split by lines L_1, L_2, L_3, and the corresponding binary tree representation. Note how the *leaf* nodes of the binary tree are single segments (or subsegments).

In order to define BSPs a bit more formally,[5] we make use of the representation of a line in the plane as a triple $\mathbf{h} = (a, -b, c)$, where $a, b, c \in \mathbb{R}$. A point $\mathbf{x} = (x, y) \in \mathbb{R}^2$ satisfies the familiar line equation $y = \frac{a}{b}x + c$ if and only if $ax - by + c = \langle(\mathbf{x}, 1), \mathbf{h}\rangle = 0$, where $\langle\cdot,\cdot\rangle$ denotes the inner product of vectors. Then, we define the positive and negative "half-spaces" determined by the line $h = (a, b, c)$ as

$$h^+ := \{(x_1, x_2) : ax_1 + bx_2 + c > 0\}$$

and

$$h^- := \{(x_1, x_2) : ax_1 + bx_2 + c < 0\}$$

Finally, given a set S of line segments in the plane, a BSP tree for this set is defined to be a binary tree T such that

- If $|S| \leq 1$, then T is a leaf node $\mathbf{v}$, and the single line subsegment of S (if there is one) is stored in $\mathbf{v}$. We denote the size 0 or 1 set stored at $\mathbf{v}$ by $S(\mathbf{v})$.
- If $|S| > 1$, then T is a binary tree of non-zero height. Letting $\mathbf{v}$ denote the root of this tree, then $\mathbf{v}$ stores a splitting line h_v, along with a set $S(\mathbf{v})$ of line segments of S that are *completely contained* in this splitting line h_v. The two children of $\mathbf{v}$ are the roots of BSP trees for the sets $S^- := \{s \cap h_v^- : s \in S\}$, $S^+ := \{s \cap h_v^+ : s \in S\}$.

Note that a line segment $s \in S$ will belong to the sets $S(\mathbf{v}_1), S(\mathbf{v}_2)$ of two distinct nodes of a BSP tree for S if s intersects both the positive and negative halfspaces of a splitting line in the tree. In this case, s is said to be *cut*, as it is partitioned into two subsegments by the splitting line. For example in figure 5 we see that segment s_3 has subsegments that belong to the set stored in the right child of the node storing L_2, and to the set stored in left child of the node storing L_3. This is inconvenient, as to store the BSP tree we now need to store *additional* vertices generated by the cuts. We define the *size* of a BSP tree to be the sum over all nodes v in the tree of $S(v)$. Thus, the size of the BSP tree depicted in figure 5 is 4.

[5] This formal definition follows that provided in [1], restricting attention to the two-dimensional setting.

3.2 Using BSP trees for rendering with the Painter's Algorithm

Given a BSP tree T for S, and viewing point E, we can render the segments of S without using a depth buffer using the Painter's Algorithm (algorithm 2). This algorithm renders segments via a postorder traversal of T, always drawing nearer segments after those farther away. The computation that needs to be performed besides rendering the segments and traversing the input tree is to compute whether the viewing point lies in the positive or negative halfspace of a splitting line. As shown in section 3.1, this can be done via a single dot product and comparison of the result to 0. So by precomputing this data structure, we can quickly determine, for any viewpoint in our scene, a priority ordering for rendering, and we have discarded the requirement of maintaining a separate z-buffer.

Note that we render the segments as stored in T, not the original input segments of S. If the splitting lines of T cut the original segments of S into many fragments, the tree will be of larger size and thus rendering will be more expensive (the entire tree needs to be traversed, and rendering a segment requires the more costly operation of projecting the segment's end points, then interpolating between the projected points). Therefore, it is desirable to produce BSP trees of small "size", as defined in section 3.1. In section 3.3, we will see that a simple randomized algorithm can produce BSP trees of expected size $O(n \log n)$, where n is the number of input segments.

Algorithm 2 Algorithm for rendering segments in S given a BSP tree T for S, and viewing point E, as presented in [1]

```
function PAINTER'S ALGORITHM(T, E)
    v ← root of T
    if v is a leaf node then
        render the line segment (if any) in S(v)
    else                                  ▷ v is an internal node, thus stores a splitting line h_v
        T⁻ ← the left child of v (a BSP for the subsegments of segments in S that are contained in h_v⁻)
        T⁺ ← the right child of v
        if E ∈ h_v⁺ then                                      ▷ First render segments behind h_v
            PAINTER'S ALGORITHM(T⁻, E)
            Render each segment contained in splitting line h_v (if any)
            PAINTER'S ALGORITHM(T⁺, E)
        else if E ∈ h_v⁻ then
            PAINTER'S ALGORITHM(T⁺, E)
            Render each segment contained in splitting line h_v (if any)
            PAINTER'S ALGORITHM(T⁻, E)
        else                ▷ E ∈ h_v, the other segments contained in h_v are not visible from E
            PAINTER'S ALGORITHM(T⁺, E)
            PAINTER'S ALGORITHM(T⁻, E)
        end if
    end if
end function
```

3.3 BSP trees via randomized incremental construction

Clearly, a BSP for a set S in the plane is not unique; there may be infinitely many such trees. We reduce our search space by considering *auto*-partitions of S. Given a line segment s, we let $l(s)$ denote the (infinite) line containing the segment s. Then an *autopartition* of the set $S = \{s_1, s_2, \ldots, s_n\}$ is a BSP tree where the splitting lines at each internal node are members of the set $\{l(s_1), \ldots, l(s_n)\}$. Note that, as shown in figure 6, even autopartitions are not unique.

The standard algorithm for BSP tree construction presented in [1], [3], and [4] constructs *autopartitions*, and is given as algorithm 3. The algorithm is simple and follows naturally from the definition of BSP trees given in section 3.1.

Algorithm 3 computes a *randomized, incremental construction* of an autopartition: at each stage it selects a new splitting line at random from the set of all lines through the input segments. This is

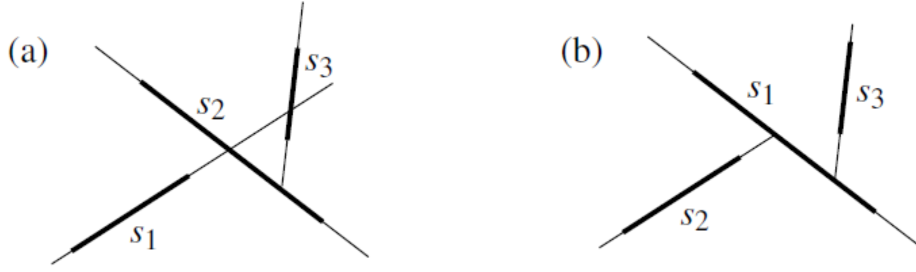


Figure 6: An example from [1] demonstrating that autopartitions are not unique

Algorithm 3 The randomized algorithm for constructing autopartitions as given in [1], [3], and [4]

function RANDAUTO($S = \{s_1, \ldots, s_n\}$)
  if $|S| \leq 1$ **then**
    Create a tree T consisting of the single leaf node storing the segment (if any) in S
  else
    Choose a segment s_i at random from S ▷ Use $l(s_i)$ as the splitting line
    $S^+ \leftarrow \{s \cap l(s_i)^+ : s \in S\}$
    $T^+ \leftarrow$ RANDAUTO(S^+)
    $S^- \leftarrow \{s \cap l(s_i)^- : s \in S\}$
    $T^- \leftarrow$ RANDAUTO(S^-)
    $S(v) \leftarrow \{s \in S : s \subset l(s_i)\}$
    Create a tree T with root node v storing $S(v)$, left child T^-, and right child T^+
  end if
  return T
end function

equivalent to computing a random permutation of the segments in S before calling `RANDAUTO`, and then always using the *first* segment $l(s_1)$ as the next splitting line in each invocation.

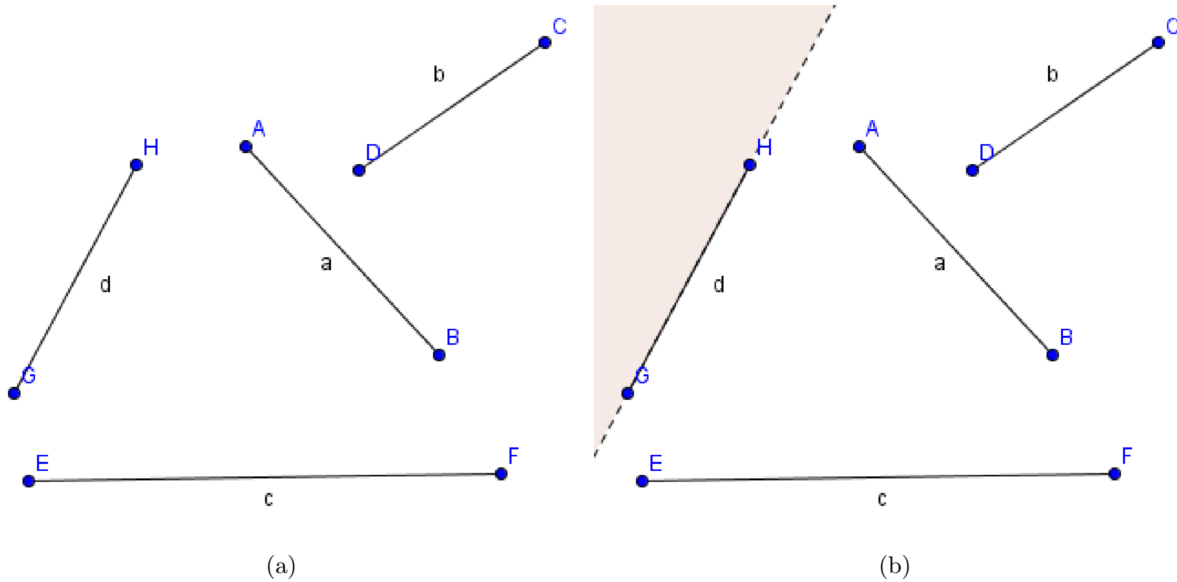
Computing the subsegment sets S^+, S^- determined by a splitting line requires a loop through all input segments, and can be done in constant time per segment by computing a line intersection. Thus, the *processing* time needed in each call to the `RANDAUTO` function (ignoring recursive calls) is $O(n)$, where $n = |S|$. In section 3.5 we will get a bound on the total number of recursive calls required, but first we will walk through an example execution of `RANDAUTO`.

3.4 Example

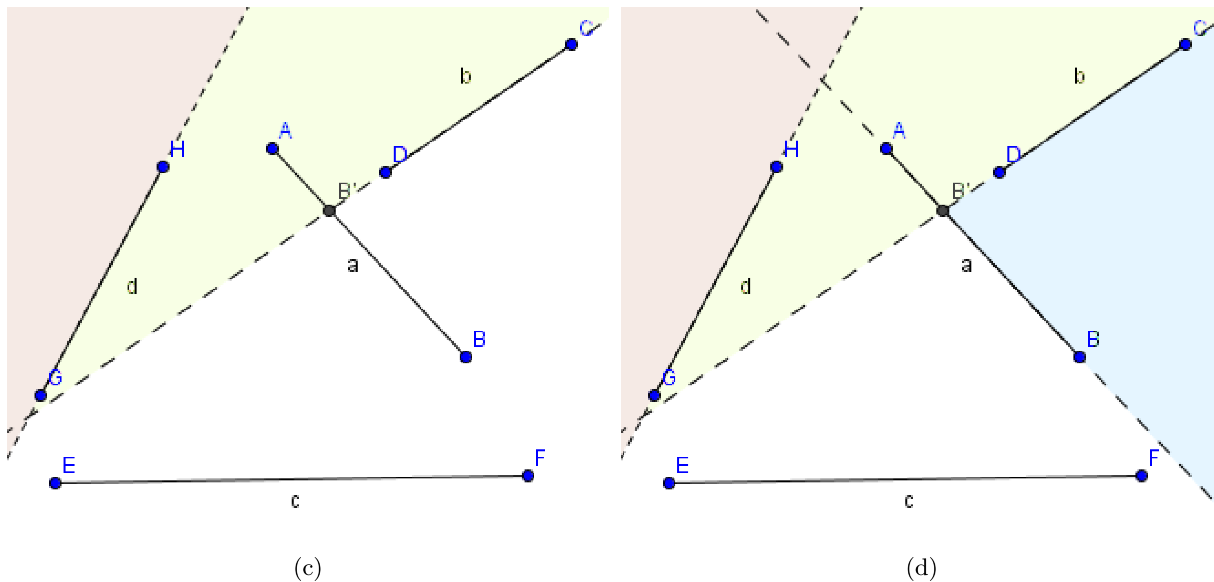
We will use `RANDAUTO` to compute an autopartition of the set $S = \{a, b, c, d\}$ depicted in figure 7a. Suppose that the random permutation we operate on is $\sigma = (d, b, a, c)$. Then on the first call to `RANDAUTO`, we will see that $|S| > 1$, and choose $l(d)$ as the splitting line, partitioning the plane into a negative halfspace (highlighted in figure 7b), and a positive halfspace that happens to contain the remaining segments in their entirety. Therefore, $S^- \leftarrow \emptyset$, and `RANDAUTO` (S^-) returns an empty leaf. $S^+ \leftarrow (b, a, c)$.

When called on S^+, `RANDAUTO` chooses $l(b)$ as the splitting line, cutting the input segment a into two segments $\overline{AB'}, \overline{B'B}$ (figure 7c). The segment $\overline{AB'}$ is the sole member of ${S'}^-$, and so the tree ${T'}^-$ is a leaf node storing this segment. The positive halfspace ${S'}^+$ consists of the two segments $\overline{B'B}, \overline{EF}$.

Finally, when called on ${S'}^+$, `RANDAUTO` produces a tree with root node storing $l(a)$, and two child leaf nodes: an empty left child, and a right child storing segment c (figure 7d). The final tree is depicted in figure 8.



(a) (b)



(c) (d)

Figure 7

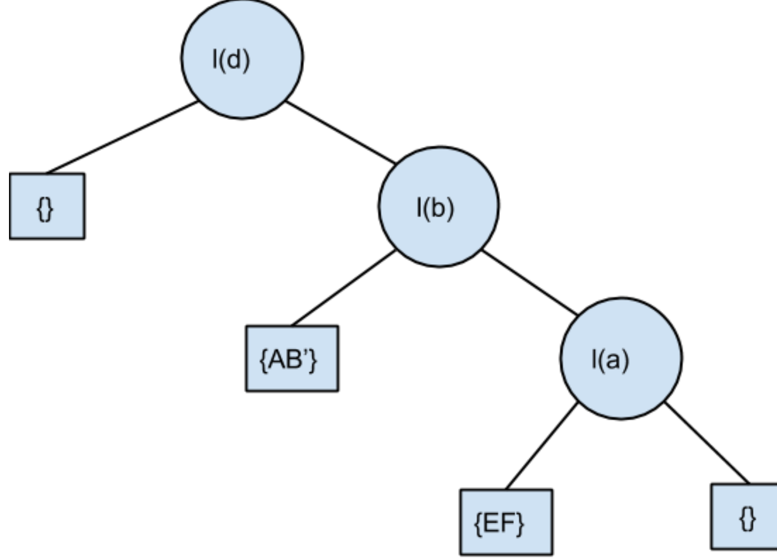


Figure 8: The tree representation of the autopartition produced in figure 7.

3.5 Analysis

As discussed in section 3.2, we would like the *size* of the BSP tree we produce (the total number of segments produced after input segments are (potentially) cut by splitting lines) to be as small as possible, for more efficient rendering. Also, the total number of subsegments of a BSP tree provides an upper bound on the total number of recursive calls to `RANDAUTO` used to compute it.

Theorem 1. *The expected size of a BSP produced by* `RANDAUTO` *on input* S *is* $O(n \log n)$*, where* $|S| = n$.

Proof. Here, the expectation is over the random permutation we use to select partitioning lines. We present the standard proof as given in [1], [3], and [4].

Given two segments $u, v \in S$, we define the *index* of the pair to be $\text{index}(u, v) = i$ if $l(u)$ intersects $i - 1$ other segments of S when traveling from u to v, and $\text{index}(u, v) = \infty$ if $l(u)$ *does not* intersect v. Figure 9 gives an example of computing the index of pairs of segments.

This "index" score helps us to determine the likelihood that v is *cut* when $l(u)$ is added to the BSP tree. Denoting the $k := \text{index}(u, v) - 1$ segments between u and v as $u_1, \ldots, u_k$, the segment v is cut by $l(u)$ only if $l(u)$ is added to the BSP tree before $l(v)$ and also prior to *any* of the $\{l(u_i)\}_{i=1}^k$, since any of these $l(u_i)$ would "block" v from being cut by $l(u)$. Thus, the event that u "cuts" v (denoted, as in [3], by $u \dashv v$) only occurs when u occurs before v and the k other segments $\{u_i\}_{i=1}^k$ in the random permutation σ we use to select partitioning lines. This occurs with probability

$$\Pr(u \dashv v) \leq \frac{1}{\text{index}(u, v) + 1}$$

So if we let $I_{u,v}$ be the indicator random variable of the event $u \dashv v$, then $E[I_{u,v}] \leq \frac{1}{\text{index}(u,v)+1}$. Now we are ready to compute the expected size of the tree produced by `RANDAUTO`(S). The tree is of size n, plus the number of intersections produced by cuts. So, by linearity of expectation, the expected size is

$$n + E\left[\sum_u \sum_{v \neq u} I_{u,v}\right] \leq n + \sum_u \sum_{v \neq u} \frac{1}{\text{index}(u, v) + 1} \tag{1}$$

Note that for any positive integer k, and any given segment $u \in S$, *at most* 2 other segments $s_1, s_2 \in S$ can satisfy $\text{index}(u, s_1) = \text{index}(u, s_2) = k$ (by extending either end of the segment u to infinity). Therefore, for any $u \in S$

$$\sum_{v \neq u} \frac{1}{\text{index}(u, v) + 1} \leq \sum_{i=1}^{n-1} \frac{2}{i + 1}$$

Combining this with (1), we have that the expected size of the BSP produced by RANDAUTO is bounded above by $n+2nH_n$, where H_n is the nth Harmonic number. Therefore, the expected size is $O(n \log n)$. □

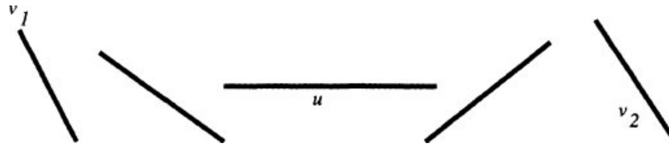


Figure 9: A figure from [3] used to demonstrate the "index" of a pair of line segments. Here $\text{index}(u, v_1) = \text{index}(u, v_2) = 2$

In [3] it is noted that since the *expected* size of a BSP tree for any input set S is $O(n \log n)$, there *must* exist a BSP tree of at most this size, by the probabilistic method. It has been proven that there are segment sets for which any BSP tree must have size $\Theta(n \log n / \log \log n)$ [1].

Also note that, as stated in section 3.2, the expected size of the BSP tree produced by RANDAUTO gives an expected upper bound on the number of recursive calls required to compute the tree. Since every call performs $O(n)$ work, we see that the RANDAUTO algorithm runs in expected time $O(n^2 \log n)$ [1].

Theorem 2. *The RANDAUTO algorithm runs in expected time* $O(n^2 \log n)$

4 Conclusion

We have seen that given a "scene" S composed of line segments, we can quickly determine visibility from any viewing point E by precomputing a (preferably small) BSP tree. The RANDAUTO algorithm (3) was presented as a simple randomized algorithm to compute BSP trees that follows almost directly from the definition of BSP trees. A probabilistic analysis of the algorithm gave us insight into the size of BSP trees in general: we determined that for any input S there exists a BSP tree of size $O(n \log n)$, by the probabilistic method.

As it turns out, there exists a *deterministic* algorithm that constructs a BSP tree of size $O(n \log n)$ in time $O(n \log n)$ (compare to RANDAUTO's expected $O(n^2 \log n)$ running time), but the algorithm is not nearly as simple to describe, and tends to produce larger BSP trees than the randomized algorithm in practice [1]. Since the construction of a BSP tree is presented as a "one-shot" preprocessing step, one can argue that the longer running time of the randomized algorithm is not a terrible trade-off for its simplicity.

References

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[2] John F. Hughes, Andries van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, and Kurt Akeley. *Computer graphics: principles and practice (3rd ed.)*. Addison-Wesley Professional, Boston, MA, USA, July 2013.

[3] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

[4] K. Mulmuley. *Computational Geometry: An Introduction through Randomized Algorithms*. Prentice-Hall, Englewood Cliffs, 1994.

[5] Fabien Sanglard. Doom engine code review. `http://fabiensanglard.net/doomIphone/doomClassicRenderer.php`, January 2010.