Binary Space Partitioning
A Focus on Rendering and Compression Algorithms

Joshua Crotts
Department of Computer Science
University of North Carolina at Greensboro

1 Introduction

With the rapid advancement in processor speeds, graphical horsepower, and miscellaneous algorithm improvements, the need for complex and sophisticated rendering algorithms that ensure constant peak performance on all types of hardware slowly fades. However, the rise in games and software with complicated geometry and models call for an increase in either algorithm development, or expensive hardware. In this research paper, we will explore several rendering and space-partitioning algorithms that developers have used throughout the years in various scenarios, with a particular interest to those that work with two and three-dimensional spaces, as well as those that compress data to reduce the initial size and overall workload throughout processing. We will also discuss the advantages and disadvantages to each, along with some non-conventional, in-progress algorithms that, while not mathematically proven, provide a great foundation for further extending potential optimization. Special thanks to Andrew Matzureff for his contributions to the paper, primarily focused in non-conventional rendering methods.

2 Painter's Algorithm

To begin, we will discuss one of the most primitive two-dimensional rendering algorithms: *Painter's Algorithm*. When rendering a scene with overlapping geometry, it is important to deduce what shapes are on top of others (in other words, in what order are they drawn so the closest to the user's perspective is on top). We refer to this as painter's algorithm because its process is analogous to an artist painting a landscape, where background elements are rendered prior to any foreground objects [6]. Using what is known as a *depth layer*, we can order our polygons/shapes by their depth values (typically on a scale from 0.0 to 1.0, with 0.0 being as close to the camera/user-perspective as possible, and 1.0 being the farthest). Therefore, we solve the rendering order problem.

Moreover, using graphs, we can deduce that a topological ordering of some graph G with vertices $v_1.z$, $v_2.z$, ..., $v_n.z$ representing polygons, and each z value corresponding to a depth provides a correct rendering of the picture. Recall that with a topological sort of node in a graph that for every edge $(u, v) \in E$, where E is our edge set, u must come before v in the ordering [11]. It is trivial to see the translation from topological sort to the rendering problem with painter's algorithm. If, as we stated, that u is some polygon, it must be rendered *before* v, if $(u, v) \in G$. A simple algorithm (not using a topological sort) can be deduced as follows:

Algorithm 1 Painter's Algorithm

Ensure: L is a list of pairs (P_n, z_n), with P_n representing a polygon and z_n as its depth layer.

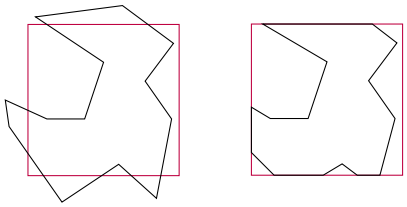
```
1: procedure PAINTER(L)
2:     Sort L by z_n in decreasing order.          ▷ Ω(n log_2 n)
3:
4:     while L is not empty do
5:         Render L.head()
6:         L.remove(L.head())
```

Two prominent issues with painter's algorithm are coping with polygons that are *cyclic overlapping*, and those that are *piercing*. With the standard approach, this procedure is incapable of deciding what to do (i.e. in what order to render) in these circumstances. We will first address and construct a polygon cutting method to deal with piercing polygons, and introduce a concept for cyclic overlaps.

2.1 Polygon Cutting Algorithms

2.1.1 Sutherland-Hodgman Cutting Algorithm

Before we dive into cutting overlapping cyclic polygons, let us introduce the idea of polygon cutting with *Sutherland-Hodgman's* cutting algorithm. We start with a list of vertices V for some convex or concave polygon P, and a polygonal object C as the viewer. We want to *clip* P such that only line segments contained in C are visible, and all others are removed. New edges are added to accommodate any segments that were removed during the cutting process that would invalidate P.



(a) *Unclipped polygon P with viewer v as the blue rectangle.*

(b) *Clipped polygon inside bounds of v.*

Figure 1: *Unclipped vs Clipped Polygon Example*

Let $v \in C$ if and only if some vertex v is strictly contained inside the viewer c. More importantly, (v_1, v_2) directs an edge from v_1 to v_2 on p. As such, treating this problem as a directed graph simplifies things. In general, there exist four properties to cutting a polygon with this method:

1. If $(v_1, v_2) \in P$ but $(v_1, v_2) \notin C$, then do not save either vertices v_1 or v_2.
2. If $(v_1, v_2) \in P$ but $v_1 \notin c$ and $v_2 \in C$, then create a new vertex v_1' at the point of intersection with c, and save both it and v_2.
3. If $(v_1, v_2) \in P$ but $v_1 \in c$ and $v_2 \notin C$, then create a new vertex v_2' at the point of intersection with c, and save both it and v_1.
4. If $(v_1, v_2) \in P$ and $(v_1, v_2) \in C$, then save both v_1 and v_2.

We use these four properties to cut and replace edges that leave C with vertices that end on the borders of C, so as to restrict the edge to C's region. The algorithm, therefore, is straightforward.

Algorithm 2 Sutherland-Hodgman's Cutting Algorithm

Ensure: P is a polygon with edges and vertices E and V, and C is a rectangular viewpoint. All edges are initially unvisited.

procedure SUTHERLAND(P, C)
  $S \leftarrow \{\}$
  for edge $e \in E$ **do**
    if e.visited = **True then**
      return S
    else if $e \notin C$ **then**
      S.add(v_1)
      S.add(v_2)
    else if $e.v_1 \in C$ **and** $e.v_2 \notin C$ **then**
      $v_2' \leftarrow$COMPUTEINTERSECTION(e, v)
      S.add(v_1)
      S.add(v_2')
    else if $e.v_1 \notin C$ **and** $e.v_2 \in C$ **then**
      $v_1' \leftarrow$COMPUTEINTERSECTION(e, v)
      S.add(v_2)
      S.add(v_1')
    e.visited $= True$

Using Algorithm 2, we compute a set of vertices that satisfy the previously described properties. Now, we need to reconstruct the now-cut polygon. All we must do is build a graph with edges from our set of saved vertices.

Algorithm 3 Build's a Polygon Cut by Sutherland-Hodgman Algorithm

Ensure: S is a set of vertices.

procedure BUILDPOLYGON(S)
  $P \leftarrow NULL$
  startEdge $\leftarrow$ S.peekFirst()
  while S is not empty **do**
    $e_1 \leftarrow$ S.removeFirst()
    if S is empty **then**
      Add (e_1, *startEdge*) to $P.E$
      return P
    $e_2 \leftarrow$ S.peekFirst()
    Add (e_1, e_2) to $P.E$

Traversing the vertices and edges in this fashion demonstrates that the run-time of both algorithms together is $O(V + E)$. In the best-case, the entire polygon is outside C (which would be pointless, but is possible), meaning the new polygon contains no vertices. Conversely, in the worst-case, every edge of P is inside C, so all vertices are processed.

How does this fit in with painter's algorithm (or other rendering methods, for that matter)? With painter's algorithm, recall that it fails whenever there exists cyclic overlap, or piercing polygons. While this does not resolve the cyclic overlap dilemma, we can remove piercing polygons as follows: Let A and B be two polygons such that A pierces B. Using B as the view, give all edges from A that exist inside B a lower z-depth, so they are rendered after B.

2.2 Cyclic Overlap

Let A, B, and C be three polygons in our scene. If these polygons exhibit *cyclic overlap*, then painter's algorithm cannot interpret their z-depths appropriately, and will not know what order to render them in.

Definition 2.1. Cyclic Overlap: When three polygons A, B, and C are layered in such a away that A is on top of B, B is over top C, and C is over top A. The entire polygon is not layered on top of another; only a specific partition is (if this were the case, this problem is trivial).

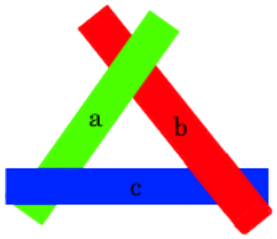
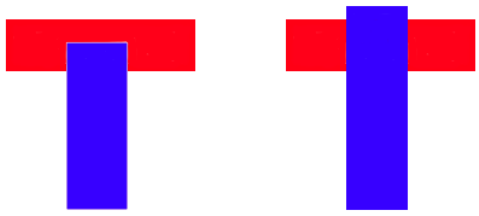


Figure 2: *Three polygons that overlap in a cycle.*

Martin and Dick Newell introduced a polygon cutting algorithm that solves this specific problem [8], but due to its complexity, we are going to build a simpler yet custom algorithm. Firstly, we need to make an assumption about our polygons: across the entire polygon, the z-depth is *not* the same for any polygons that exhibit cyclic overlap. This is why, for instance, polygon A is covered by C at certain points, and covers B at another.

We can define an algorithm to iterate through a list of polygon L, testing each one for overlap with another. Given that polygon A *completely overlaps* B at some arbitrary point on B, split B into polygons B_1, B_2, B_3 (respective to the pieces not covered by A, as well as the one that is still covered).

Definition 2.2. Complete Overlap: If a polygon A has a section of it where $A.z$.depth $< A.z$.depth, and any vertices that enter B from A exit on some side of B, then A *completely overlaps* B.



(a) *Non-complete overlap.* **(b)** *Complete overlap.*

Figure 3: *Non-complete vs complete overlap.*

We will construct our algorithm on the grounds that any polygon either completely overlaps another, or does not overlap at all. For every polygon $P \in L$, if P overlaps some other polygon Q, split Q at the overlap point into three pieces Q_1, Q_2, Q_3. Insert these at the rear of L, and remove Q from the list. Continue testing until there are no discrepancies. Then, sort all polygons by their z-depth in descending order.

3 Scan Line Polygon Fill Algorithm

Continuing with our discussion of polygons, in a two-dimensional (or even three-dimensional) space, we need a way of computing the color of a given polygon P. When we call our render function on any complex polygon, we cannot assume that the procedure nondeterministically knows how to color geometry; rather only where to place it in our scene. There are several algorithms for filling polygons, but we will focus on the *Scan Line Polygon Fill* algorithm, predominately for its simplistic implementation.

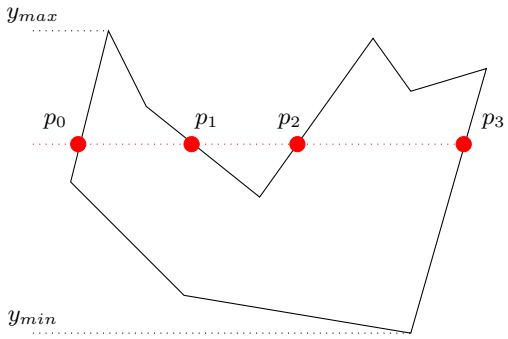


Figure 4: *Illustration of Scan Line Algorithm on a Convex Polygon*

The idea for our brute-force approach is as such: first compute the minimum and maximum y-coordinates of the polygon. In Figure 4, these values are *y-min*, and *y-max* respectively. Assume this is a typical Cartesian-coordinate plane, where the origin is located at the bottom-left of our scene. From this, we can use a horizontal vector referred to as a *scan line* to compute intersection points on the polygon. In our example, these are conveniently labeled P_0, P_1, P_2, and P_3. Iterating from *y-min* to *y-max* (the converse also works), draw the scan line from the far-left of the scene to the far-right, and gather all intersection points that lie on the boundary of our polygon. For instance, in Figure 4, pixels between the interval $(P_0, P_1) \cup (P_2, P_3)$ are filled with the color of the respective polygon. Notice how we ignore the interval (P_1, P_2). This is because pixels in this region are not within the bounds of our polygon. So, during the intersection-point collection process, P_1 marks the end of a intersection set (indicating points will no longer be "inserted" into that set), whereas P_2 marks the start of a new one.

Algorithm 4 Color a Convex or Concave Polygon

Ensure: P's position is defined by a previous ordering.

```
1: procedure ColorPolygon(P)
2:     x-coords ← {}
3:
4:     for y = P.y-min to P.y-max do
5:         x-coords.add(DrawScanLine(P, y))
6:     while x-coords is not empty do
7:         Remove x-coords.head()
8:         Color at column x-coords.y from x-coords.StartX to
9:         x-coords.EndX
```

One important note about this algorithm is that it is only for demonstrative purposes; this brute-force algorithm is rarely used in practice. In reality, basing the edges of a polygon as vectors, testing them against the scan line, and sorting in order of increasing x coordinate is a better alternative than the naive method. This approach provides a consistent performance cost across all screen resolutions and screen coverage per polygon. With the brute-force

Algorithm 5 Draw Scan Line from Left to Right

```
1:  W ← Scene.Width
2:  H ← Scene.Height
3:
4:  procedure DrawScanLine(P, y)
5:      InsidePolygon ← False
6:      StartX ← −∞          ▷ Starting point of an intersection.
7:      EndX ← −∞            ▷ Ending point of an intersection.
8:      x-coordSet ← {}      ▷ Pairs of intervals of intersection
9:
10:     for x = 0 to W do
11:         if Pixel (x, y) is on P then          ▷ If p intersects P.
12:             if InsidePolygon = True then      ▷ End interval.
13:                 EndX ← x
14:                 x-coordSet.add((StartX, EndX)
15:                 InsidePolygon ← False
16:             else                  ▷ Initialize intersection interval.
17:                 StartX ← x
18:                 InsidePolygon ← True
19:
20:     x-coordSet.y ← y          ▷ Specify what column to color.
21:     return x-coordSet
```

solution, wider/bigger polygons have more pixels to check. However, using the improved vector-edge method increases the overall complication, and is harder to visualize. Compared to the worse brute-force approach that looks at *all* intervals, however, this is slightly faster in that we only consider intersections that contain points inside the polygon, and none outside. This suggests that once we have our intersection set complete, no post-processing has to occur (i.e. deciding whether an interval contains pixels or not).

The best approach for our algorithm is to define each canvas as one that contains strictly one polygon (and only one polygon) to reduce the complexity of the overall problem. Moreover, if we render more than one polygon at the same time (i.e. contained in the same canvas), we run into the predicament of polygon color differentiation. Note the trivial nature of the problem if all shapes are the same color, though.

With our assumption, let H be our canvas height defined by $H = P.y\text{-}max - P.y\text{-}min$. Hence, we can place a loose lower-bound on our algorithm at $\Omega(WH)$, where W and H are the width and height of our canvas, as aforementioned. The simplest polygon is one that contains three sides (because it must be enclosed by the definition of a polygon) with only two points of intersection per iteration of the scan line (a triangle for example). Therefore, we add one set to x-coords every iteration, so at the end, we have $1 \cdot H$, or just H pairs. However, in the worst case, there may exist a polygon that contains n intersection intervals per iteration (where $n < W$, imagine a square with a vertical line every other pixel). As such, since we must color in every interval, no matter how small it is, we may end up with $n \cdot H$ pairs in x-coords. Therefore, the true upper-bound is $O(W \cdot H \cdot n)$. The run-time wildly differs based on the type of polygon we need to render.

3.1 Scan Line Improvements

There are speed improvements by taking advantage of *coherence*. Often times, pixels contained in a span do not change much, and the visibility (as well as the color) likewise vary little from one previous scan line to the next [4]. Moreover, as we described, it is also possible to treat each edge as a vector, and the entire scan

line as a vector from the left side of our canvas to the right (of width W and height $H = 1$) with the use of a line-intersection formula that determines the edges that cross our scan-line (given that they exist). Again, though, this means we will need to sort the intersection points by increasing x coordinate, so we render these sections in the right location and order.

Generally speaking, this is a software-rendering approach to drawing polygons, and depending on the number and variations of polygons in the canvas (as well as the implementation of the algorithm), it changes significantly in terms of performance.

4 Ray-Casting

Ray casting is a technique introduced to allow for fast rendering of a three-dimensional space from a two-dimensional map. Not to be confused with *ray-tracing*, which is a rendering algorithm used to compute real-time shadows and reflections from a light source that is absorbed (and possibly reflected) by other objects, ray-casting uses a grid of squares to compute boundaries and other objects in a region. In the past, this strategy was used by *Wolfenstein* and *Doom*, revolutionary games developed by ID Software. Our algorithm works as follows:

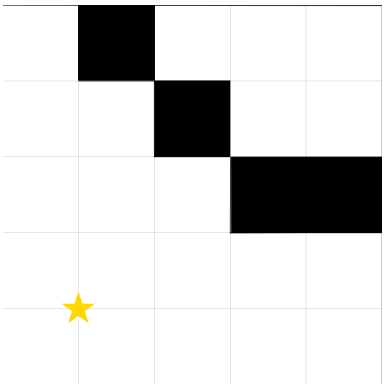


Figure 5: *2-D grid scene with camera v as the star.*

As a visual aid, Figure 5 shows a camera v with several walls in front of its view port. Each black square represents a wall, whereas the gray lines indicate separations for objects. Assume that in the system, each square corresponds to a one-bit flag, where 1 denotes a wall/object, and 0 indicates free space. For every x in the view port of v, fire a *ray* from v, and continue outwards until a wall is hit, or the edge of our scene is reached, whichever is first. Hence, our scene s consists of a matrix of bits. Upon firing this ray, whenever an object is hit, compute the distance from v. Then, when drawing in the third dimension, use this as a metric for how close or how far the object is away from v. The smaller the number is, the higher an object should be drawn. Conversely, the higher the number, the lower an object is drawn to simulate depth.

Algorithm 6 Shoots a Ray Across the Viewport to Any Object Within Sight of v

```
1: procedure RayCast(v)
2:     for x = v.xMin to v.xMax do
3:         ShootRay(v, x)
```

Defining the subroutine ShootRay is not as trivial; we have to determine *how* an intersection is computed. For instance, a human is more than capable of identifying the wall quickly, whereas the procedure has to algorithmically decide when an object intersects the ray. The ShootRay routine could, in theory, check every coordinate the two-dimensional space between v and an object (or the bounds of the scene), but this would be ridiculously slow, since a boundary might be very far from v with nothing in between, so all of those checks are useless. However, suppose we place a limit on how many checks are performed, k (where k is a numerical value in pixels). Every k "steps"in the two-dimensional scene, a point is checked.

Algorithm 7 Determines the Points of Intersection from a Ray r

```
1: procedure ShootRay(x, k)
2:     s ← k                          ▷ Interval for checking collisions
3:     while no intersections with the ray have occurred do
4:         if Check(x, s) then
5:             return k
6:             s ← s + k
7:     return −1
```

One glaring flaw with this approach is that it might *miss* a point in the two-dimensional scene if k is too infrequent. Moreover, a wall may be drawn at an incorrect distance from v.

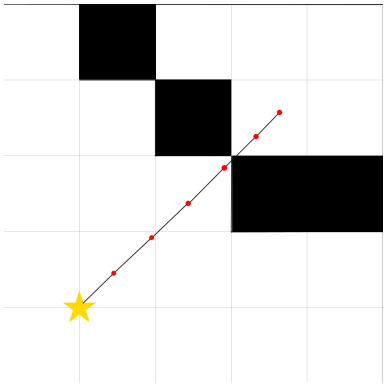


Figure 6: *Ray that misses the wall object.*

As k increases, the probability that a collision is detected decreases. Likewise, when k is lower, false negatives are less common, but the number of computations by our algorithm increases. A fair balance between the number of checks that detects objects every time can be achieved by checking the faces of any square in the two-dimensional matrix that defines s.

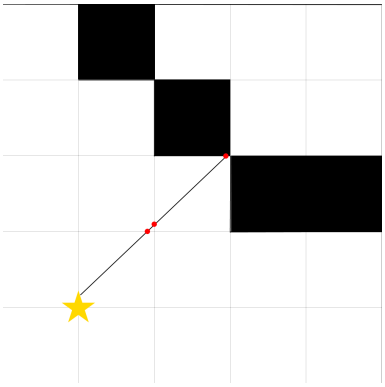


Figure 7: *Ray that checks only instances where a wall might exist.*

Due to limitations with this type of rendering engine, only certain types of geometry are able to be rendered: those that are modellable in a two-dimensional scene of squares (or pixels). So, the scene (when visible in the third dimension) looks pixelated. However, for a quick pseudo-3D rendering method is good for what it is meant for, but when comparing to a computationally-heavy ray-tracing implementation, while it may best it in speed, the overall quality drops significantly.

One supplemental problem solved by ray-casting is the hidden surface problem, which we will discuss later in the binary space partition section. Because a ray renders the first column that it intersects with, any structures or objects that exist behind the ray are not rendered, thus reducing computation time. In addition, there exists a problem in computational geometry known as the *Ray Shooting Problem*: if provided a d-dimensional space, construct a data structure in such a way that if we perform "ray queries", we quickly find any objects that are intersected by the ray. Our trade off is that if we preprocess the information, more storage is used, but queries are given a performance boost. This data structure and algorithm design goes beyond the scope of this paper.

4.1 Ray-Marching

A potential optimization to the primitive ray-casting algorithm is known as *ray-marching*, where a fixed or variable-sized step is calculated to determine where and when the succeeding intersection check is performed. For our purposes, we will investigate a specific variant of ray-marching known as *sphere-tracing*. Upon initialization, we compute the shortest distance from v to any arbitrary polygon P in the scene. Using this as a radius r, we draw a circle around the viewer v, then extend a ray out to the circumference of the circle we defined. Repeat this process recursively, while ensuring the ray never changes direction (i.e. it is a static vector; it cannot change direction once it has started). Once the radius of a circle is smaller than the threshold k, we detect a collision with a polygon, then proceed to the next ray.

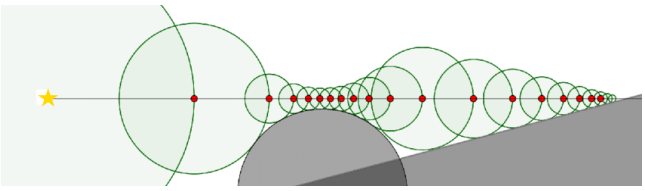


Figure 8: *Example of ray-marching algorithm using sphere tracing [2].*

We need to define a procedure that computes the shortest distance from a point v to every object in the scene. Let L be the list of polygons in the scene, and assume we have defined a mathematical function to compute the distance between one polygon and another, f.

Algorithm 8 Computes the Shortest Distance from a Point to All Other Polygons

```
1: procedure ShortestDistance(v, L)
2:     min ← ∞
3:     for polygon l in L do
4:         dist ← f(v, l)
5:         if dist < min then
6:             min ← dist
7:     return min
```

Our ray continues until the radius of the circle it generates is small enough to be considered a collision. Note the similarities between ray-marching and ray-casting. The differences between the two are that ray-marching guarantees that we never pierce a polygon, and that eventually, either the ray collides with a polygon, or it goes out of the scope of our scene. k is a very small number, in that the closer to 0, the more accurate a collision detection is (at a cost of computation time).

Algorithm 9 Uses ray-marching with a variable step to draw one individual ray.

```
1: procedure SphereTrace(v, k, L)
2:     r ← ∞
3:     pov ← v          ▷ Re-position the viewer after each circle.
4:     while r > k do
5:         r ← ShortestDistance(pov, k)
6:         if r ≤ k then
7:             break              ▷ Upon detection, we terminate.
8:         else
9:             Extend ray from pov by r.      ▷ Pre-determined d.
10:            pov ← pov + r
11:    return min
```

If there exists plentiful geometry in a scene, but a wall or boundary is far from the viewer, then we still make several superfluous calls to this procedure. However, for accuracy, this subroutine works well, especially when a scene is not defined by a grid of set bits (that determines if any arbitrary boundary exists there or not). In using this, we completely remove false-negatives in exchange for a prospective (slight) increase of intersection checks. This approach does, however, outperform checking every pixel using the brute-force approach.

5 Binary Space Partitioning

5.1 BSP Trees

Similar to several tree structures, binary space partitioning involves recursively partitioning/splitting a scene into two or more pieces, until some arbitrary requirement is satisfied. Each node in a BSP tree stores a split known as a hyper plane, which defines the place at which a scene is subdivided. A parent node a of some child node b encompasses b, along with all of b's children. Each time we subdivide, we ensure that two new children nodes are added to the tree, where the parent represents the split that instantiated those child nodes. Children b and c of parent node a represent the placement of some object, and are either "in front of", or "behind" the hyper plane described by a.

Before we continue, let us discuss the advantages of utilizing a BSP tree. Firstly, we cut down on duplicated rendering. Recall that painter's algorithm renders *any* polygons that are in the scene, regardless of its visibility in the canvas, suggesting that some pieces of a polygon (if not the entire polygon) are hidden to the viewer. Thus, that rendering is unnecessary, wasting valuable CPU time. Moreover, it does not correctly handle the error of overlapping polygons. Lastly, there is the required time to sort the z-indices of each polygon. BSP trees eliminate unnecessary rendering altogether by partitioning polygons in such a way that the redundant nature of painter's algorithm is removed.

Certain semantics (such as the order in which front and back nodes are stored) for the BSP tree are up to the implementation of the user. Furthermore, the criteria for ending the recursive BSP calls, and the plane partition algorithm are based on the end-goal. For our purposes, we will use the computer-graphics approach (as opposed to back-face culling), where each scene is subdivided until all scenes contain polygons in an order that does not matter when rendered (for that particular scene!).

Now, let us examine how to select a root for every sub-BSP tree. We need a way of choosing a polygon such that when we pick a location to partition, we avoid unbalancing the tree in one direction or the other, and fast query and look-up operations are retained. Let us reorganize our input to be a list of *planes*, defined as the edges of a polygon. Hyper planes are line segments that span across and extend beyond a polygon's plane.

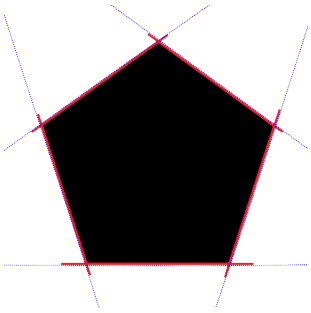
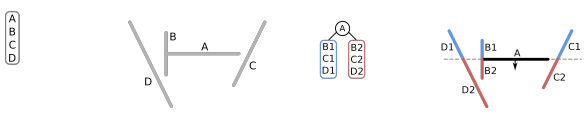


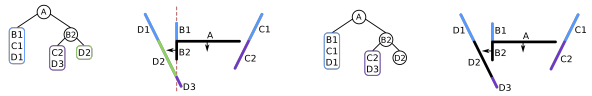
Figure 9: *(Red) Planes, (Blue) Hyper planes of a pentagon.*

Figure 2 illustrates this example. In previous implementations, researchers have used "random" selection of a polygon to be the root of the BSP tree [5], and in our pseudo code, we will use this as our baseline. Mark de Berg proved that a randomized algorithm (which picks the partition at random) guarantees an upper-bound of $O(n^2 \log_2 n)$ [7]. However, if we want to reduce the possibility of a highly-unbalanced tree, we could choose the edge that is closest to the center of our current scene. That way, on average, we get half the edges on the left, and half our edges to the right.

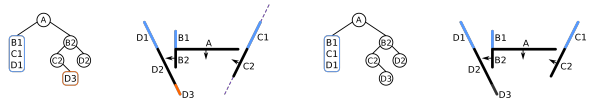
To reduce the overall complexity of drawing an example, let us draw the BSP tree for a simple scene with only wall segments, or straight lines [12]. For this example, and the algorithm that we will write, we assume the input is one-dimensional polygons instead of *planes* (to be discussed later):



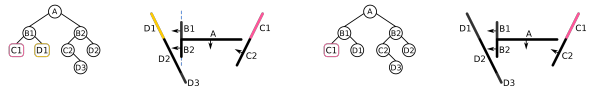
(a) *Step 1* **(b)** *Step 2*



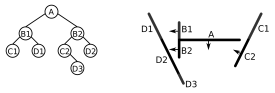
(a) *Step 3* **(b)** *Step 4*



(a) *Step 5* **(b)** *Step 6*



(a) *Step 7* **(b)** *Step 8*



(a) *Step 9*

Let us walk through the generation procedure. First, we initialize a list of polygons (or wall segments, in this case). Now, poll the top polygon, and make it a node in the BSP tree. We will refer to this root as r. Decide which sides represent "front", and "back". In our example, the front of a plane is denoted by the black arrow. We add all polygons that are entirely in front of r to a list F, and all that are entirely behind r to a list G. We define "entirely in front/behind" as having no intersections with the hyper plane generated by r. However, in the event that an intersection occurs (as in several steps, but first appears in 2),the polygon is split into two new polygons at the point of intersection, and are inserted into the correct corresponding list. Repeat this process recursively for both lists, and terminate when there are no remaining polygons in the list.

Algorithm 10 Builds a Binary Space Partition Tree from a List of Polygons P

```
1:  T ← []
2:  procedure BUILDBSPTREE(P)        ▷ P is a list of polygons.
3:      if P is empty then
4:          return NIL
5:      F ← []        ▷ Polygons entirely in front of the hyper-plane.
6:      G ← []        ▷ Polygons entirely behind the hyper-plane.
7:      r ← P.poll()      ▷ Removes the first polygon from the list.
8:      T.add(r)      ▷ Add to left if in front, add to right if behind.
9:
10:     for p ∈ P do
11:         if p intersects r.hyperPlane then
12:             q ← p.splitAt(r.hyperPlane)
13:             P.add(q)
14:         else if p is entirely in front of r then
15:             F.add(p)
16:         else
17:             G.add(p)
18:
19:     BUILDBSPTREE(F)
20:     BUILDBSPTREE(G)
```

Keep in mind that this algorithm can and does work with polygons that are more than one dimension (such as lines). In order to process more complex geometry, recall how we mentioned the idea of reducing our input from polygons to *planes*, and adding those to a list. Because we have to test every possible plane, this increases the complexity of building the BSP tree, but reduces the complexity when traversing. Our algorithm needs a modification to implement this behavior, however. If a hyper plane l intersects some arbitrary plane p, then we need to break the polygon that produces that plane into two or more separate polygons, and add those planes to our global list. The issue with this lies with the specific plane chosen by the algorithm to partition at. By picking a poor plane, we could split a polygon into many pieces, and all of these polygons must eventually be added to the tree, thus increasing the amount of objects to render and compute [9].

Algorithm 11 Builds a Binary Space Partition Tree from a List of Planes Generated by Polygons

Ensure: L is the list of planes for all polygons in the current scene.

```
1:  T ← []
2:  procedure BUILDBSPTREE(L)
3:      if L is empty then
4:          return NIL
5:      F ← []                    ▷ Planes entirely in front of the hyper-plane.
6:      G ← []                    ▷ Planes entirely behind the hyper-plane.
7:      r ← L.poll()              ▷ Removes the first plane from the list.
8:      T.add(r)                  ▷ Add to left if in front, add to right if behind.
9:
10:     for l ∈ L do
11:         if l intersects r.hyperPlane then
12:             q ← l.getPolygon().splitAt(r.hyperPlane)
13:             L.add(q.planes)                ▷ Adds all planes.
14:         else if l is entirely in front of r then
15:             F.add(l)
16:         else
17:             G.add(l)
18:
19:     BUILDBSPTREE(F)
20:     BUILDBSPTREE(G)
```

Recall that BSP trees are fast for rendering three-dimensional scenes. Let us transition to that domain now, using what we know about BSP trees. Suppose a viewer v is placed directly below $D3$ from Step 9 of the example.

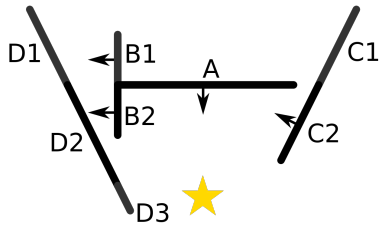


Figure 15: *Yellow-star is our viewer v*

Using the BSP tree, we only want to render those polygons that are visible to v. We have another recursive traversal algorithm for this process with increased performance over painter's algorithm, and others such as ray-casting due to the preprocessing work done when building the structure. Because no polygons intersect in the BSP tree, we do away with that limitation of painter's algorithm. We also do not have to sort the polygons at any point. However, we still use a *version* of painter's algorithm; just with minor alterations to adjust it to the expectations from a BSP tree.

Algorithm 12 Traverses Through a BSP Tree to Render a 3-D Scene.

```
1:  procedure TRAVERSETREE(T, v)   ▷ v is the current location of the viewer object in the scene.
2:      if T is empty then
3:          return NIL
4:      else if T.F is null and T.G is null then
5:          Render T.r
6:
7:      if v is in front of T.r then
8:          TRAVERSETREE(T.G, v)
9:          Render T.r
10:         TRAVERSETREE(T.F, v)
11:     else if v is behind of T.r then
12:         TRAVERSETREE(T.F, v)
13:         Render T.r
14:         TRAVERSETREE(T.G, v)
15:     else if v is on T.r then
16:         TRAVERSETREE(T.F, v)
17:         TRAVERSETREE(T.G, v)
```

Our idea, comparable to the approach for painter's algorithm, is to ensure correct ordering/rendering of each node in the BSP-tree. The following property must be satisfied: for any node n in a BSP tree T, all nodes behind n must be rendered prior, then n is rendered. Afterwards, all of those in front of n are drawn, once again assuring the basic topological sort property. One significant advantage of building the BSP tree in this fashion (and designing the traversal algorithm as we did) is that no matter where our viewpoint v is located, the algorithm will work the same and produce the correct ordering of objects in the scene.

Let us now trace through our example. Let $m \leftarrow\!\!\prec n$ signify some object or node m is in front of another object or node n. We will demonstrate a loop invariant that each node drawn progressively approaches v (or more formally, given that every object has a respective z-depth z, object $n.z > n + 1.z$, where $v.z = 0$). Starting at the root A, $v \leftarrow\!\!\prec A$, so we render all children behind A first, meaning we traverse down the left side of the BSP tree. $B1 \leftarrow\!\!\prec v$, so render all children in front of $B1$, meaning we traverse down the right path. $D1$ is a leaf node, so we render it. Recursively backtrack, now render $B1$. Travel to $C1$. $C1$ is a leaf node, so we render it. Recurse back up the tree to A, and render it, since we have rendered all children behind A, and follow the right subtree. $B2 \leftarrow\!\!\prec v$, so we investigate $D2$, which is a leaf node, so we render it. Now, draw $B2$. Finally, traverse all children in behind $B2$. $v \leftarrow\!\!\prec C2$, hence we check all children behind $C2$. Though, there are no children behind $C2$. Now, render $C2$. Finally, check all children in front of $C2$, leaving $D3$ which is a leaf, so it is drawn. Hence, we have completed the BSP tree traversal. The correct ordering *for this position of* v is

$$\langle D1, B1, C1, A, D2, B2, C2, D3 \rangle$$

Notice that our loop invariant remains true throughout iteration; every element in our list is further from v than all its successors.

However, earlier we mentioned that the leading purpose of BSP

trees was to remove redundant rendering, which is not achieved in the back-to-front traversal. Using this approach *does* address the issues with the primitive painter's algorithm, but we need to redefine our method to render front-to-back [3].

A way to implement this is by using a scan-line data structure to keep track of pixels that we have previously rendered. Suppose we have a data structure that, as we render nodes in our BSP tree, continuously updates a table of x-coordinates of the nodes already drawn [4]. That way, when deciding how to draw another object, we only draw pieces visible to our viewer v, thus reducing the amount of an object to draw. We will denote this data structure as a W-bit array (where W is the width of the scene visible to v) called L. Each bit in this array corresponds to an x-coordinate having been rendered or not. Assume each object k in the BSP tree contains two integers $minX$ and $maxX$, marking their starting and ending positions along the x axis in the canvas. Further assume that $x \in [1, W]$ so we avoid going out of bounds of the field of view of v). Now, let us adjust the procedure slightly to accommodate this change:

Algorithm 13 Render BSP Tree from Front to Back

```
1:  L = W-bit array                          ▷ W is the width of the canvas.
2:
3:  procedure RenderFrontToBack(T, v)        ▷ v is our viewpoint, T is our tree.
4:      if T is empty or L is completely set then
5:          return NIL
6:      if v is in front of T.r then
7:          RenderFrontToBack(T.F, v)
8:          RenderObject(T.r)
9:          RenderFrontToBack(T.G, v)
10:     else
11:         RenderFrontToBack(T.G, v)
12:         RenderObject(T.r)
13:         RenderFrontToBack(T.F, v)
14:
15: procedure RenderObject(k)        ▷ k is some 2D/3D object.
16:     if k.minX to k.maxX bits are set in L then
17:         return []
18:     i ← 1                        ▷ Index offset for rendering k at a column.
19:
20:     for x = k.minX to k.maxX do
21:         if L[x] is not set then  ▷ Only render a piece of k.
22:             Sub-render k[i] at x-coordinate x
23:             Flag L[x] as set.
24:         i ← i + 1
```

After these alterations, we need to address the assumptions made. Firstly, we created a sub-procedure RENDEROBJECT that checks to see if the xth bit in L is not set, and if so, we "sub-render" k and index i. To sub-render an object means to only draw one column of said object, instead of drawing the entire object. However, this does not mean "draw the xth pixel of k", because that could go out of bounds of the object. We define a variable i to keep track of which specific column of pixels to render for k. For example, if we have an object a that spans across $x = 1$ to $x = 300$, with another object b behind it spanning $x = 170$ to $x = 670$. We render a first, and set bits 1 to 300 in L. Upon reaching b, we see that pixels $170 - 300$ are flagged. So, we start rendering b at $x = 301$, and therefore, $b[i]$ when $i = 131$. This mechanism was introduced to ensure no confusion about what column is rendered in which location. Moreover, line 4 provides a way to escape the recursive calls if the entire scene has polygons, because it is unnecessary processing to continue the walk of the tree if there is no room left for other objects. Likewise, line 16 breaks out of the render call if there exists no room to render one specific polygon at all, because every x-coordinate that it encompasses is already full in L (in other words, is taken up by another object in the canvas).

Lastly, let's trace through this front-to-back procedure with the example from before. Unfortunately, we have no x-coordinates explicitly labeled, so we will make a mental note when all objects that are visible to v are rendered.

Beginning at the root A, $v \leftharpoondown A$, so we render all children in front of A first, meaning we traverse down the right side of the BSP tree. $B2 \leftharpoondown v$, so we traverse through the left-branch to $C2$. $v \leftharpoondown C2$, so we reach $D3$. $D3$ is a leaf, so we render it. Back up the tree, and we reach $C2$. $C2$ has no remaining children, so it is rendered next. Back up the tree once again, and we get to $B2$, which is rendered. Now, traverse through all children in front of $B2$, in which the only one is $D2$, so it is rendered. Recurse back to the root A, render it, then process all children behind A. $B1 \leftharpoondown v$, so we visit children behind $B1$, leaving us with $C1$, a leaf node. It is drawn, and we return to $B1$ and render it. Finally, $D1$ is visited and rendered. So, the complete front-to-back ordering as as follows:

$$\langle D3,\, C2,\, B2,\, D2,\, A,\, C1,\, B1,\, D1 \rangle$$

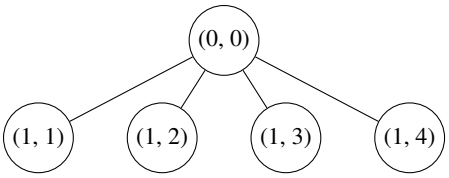
If this looks familiar, that is because it is identical to the back-to-front ordering, but reversed. The reason in complicating the algorithm for a front-to-back approach is to reduce the amount of rendering (i.e. hidden surface removal!) performed by our algorithm. And, by using our data structure to its full potential, we can remove $B2$ and $B1$ in their entirety from the draw calls, as $D3$ shadows them. Additionally, parts of A are reduced because of $C2$ and $D3$.

In conclusion, BSP trees are great for drawing both two-dimensional and three-dimensional scenes, but they thrive when drawing complex three-dimensional polygons in some given space, particularly when the amount of polygons needed to process becomes large. However, it is dependent on the structure and algorithms used to process and build the BSP tree.

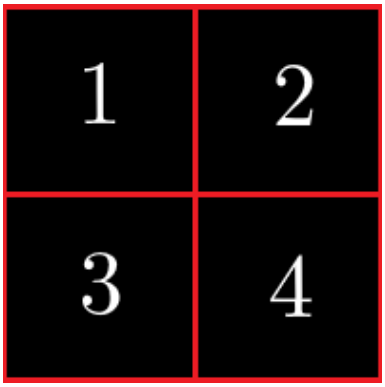
5.2 Quadtrees

In a manner that closely resembles BSP trees, we will now turn our discussion to the data structure known as a $Quadtree$. Quadtrees are recursively-defined trees that have exactly zero or four children as their respective subtrees. Typically used for two-dimensional regions/planes, we will define a *region* as a square somewhere as a node in the quadtree.

Initially, the root is (geometrically/pictorially) represented by one (presumably large) square of size $n \times n$, where n is the size of our initial region that encompasses the entire canvas. Suppose we want to add an object to this space, in particular, a pixel (point). We will further elaborate on our primitive definition of regions; suppose each region R has a list L of pixels, denoted as R_L. Moreover, whenever we want to add a point p to our canvas, we need to traverse through the quadtree to find the appropriate region $R_{(i,j)}$ to add p to, where i is the depth of the node, and j is a number in the range 1 to 4, representing the children.



(a) *Graphical Representation*



(b) *Blank "Canvas" for QuadTree*

Figure 16: *One root, four-children Quadtree*

Assume we want to add p to the canvas to an arbitrary region. We first need an algorithm to determine if a point already exists in a region. If so, then we must *subdivide* that region into four equally-sized (smaller) regions. Let M be the root of our quadtree. Initially, there are no points in the M (namely, $|M_L| = 0$). As soon as we insert more than one point p to M_L, four regions are generated, and p is placed into whichever region it is enclosed by. More formally, whenever $|M_L| > 1$, we recursively subdivide.

Algorithm 14 Add Points to QuadTree

```
1:  procedure AddPoint(T, p)
2:      if p is not inside bounds of T then
3:          return []
4:      else if |T_L| ≤ 1 then    ▷ If valid space exists in the region.
5:          T_L.add(p)
6:      else
7:          Subdivide(T)                    ▷ Otherwise, subdivide.
8:          T.TopLeft.add(p)
9:          T.TopRight.add(p)
10:         T.BottomLeft.add(p)
11:         T.BottomRight.add(p)
12:
13: procedure Subdivide(T)
14:     if T.Subdivided = False then
15:         T.TopLeft ← new QuadTree(T.TopLeft())
16:         T.TopRight ← new QuadTree(T.TopRight())
17:         T.BottomLeft ← new QuadTree(T.BottomLeft())
18:         T.BottomRight ← new QuadTree(T.BottomRight())
19:         T.Subdivided ← True
```

Definition 5.1. Enclosed By: Object A is enclosed by Object B if the box-bounding rectangle that surrounds A is fully within (inside) the rectangular box-bounds of B.

Therefore, if some p is enclosed by a region R, it is added to R_L. More specifically, in Algorithm 2, we attempt to add the point into every newly-generated region. We do this to determine which node in the quadtree our point p belongs to. The run-time of this algorithm, assuming n nodes, is $\Theta(\log_2 n)$.

Proof. We can derive a recurrence equation along with its respective solution using the master theorem as follows:

$$T(n) = T(\frac{n}{4}) + \Theta(1)$$

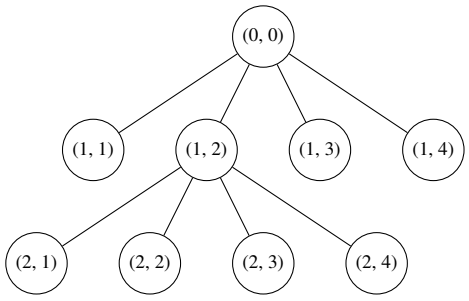
We know that $a = 1$, $b = 4$, and $f(n) = \Theta(1)$. Moreover, $\Theta(n^{\log_b a}) = \Theta(n^{\log_4 1})$. If we take any base for a logarithm of 1, we get 0. Thus, $\Theta(n^0) = \Theta(1)$. This falls under case two of the master theorem, because $f(n) = \Theta(1)$.

Therefore,

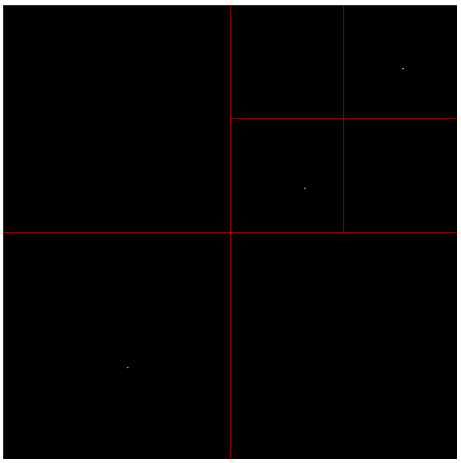
$$\begin{aligned} T(n) &= \Theta(n^{\log_b a} \cdot \log_2 n) \\ &= \Theta(n^{log_4 1} \cdot \log_2 n) \\ &= \Theta(1 \cdot \log_2 n) \\ &= \Theta(\log_2 n) \end{aligned}$$

■

Figure 17 shows two images: (a) shows a quadtree with four children, and (b) is the pictorial representation. If we added a node to the top-right region, node $(1, 2)$ would subdivide into four extra regions.



(a) *Subtree representation.*



(b) *Lightly-populated quadtree*

Figure 17: *Quadtree with a sub-quadtree*

We can add as many points to the quadtree that we desire, until we run out of space to subdivide our plane. Namely, if our initial

region, or canvas, is of size $n \times n$, then we can only subdivide the tree $\lfloor \log_2 n \rfloor$ times, assuming our pixel is of size 1, because once a region has dimensions less than 1, no free space exists to insert new points.

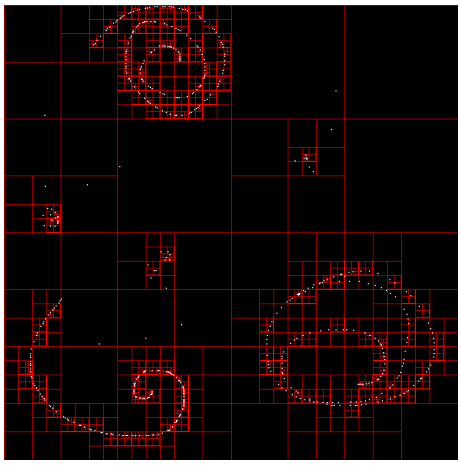


Figure 18: *Densely-populated quadtree*

We will now focus on one aspect of quadtrees when processing images: data compression. Suppose we have an image that shares a lot of color throughout the image in close proximity. For instance, take the logo for the University of North Carolina at Greensboro:

Figure 19: *UNCG Spartan Logo*

Using a standard, raw (no prior compression) image file type, we can represent every pixel as a four-byte integer, where each byte corresponds to an 8-bit (0-255) value for translucency (or alpha), red, green, and blue channels. Therefore, an image takes up $m \times n \times 4$ bytes of data, which can rapidly grow depending on the image. In the case of Figure 19, however, we see that there is a lot of redundant coloration; much of the black-pixel data could be reduced, alongside the yellow and dark-blue. So, instead of our regions R containing a list of points, we can denote that each region consists of one color (and only one color). More so, if a region has more than one color, subdivide that region untilonly one color populates that region.

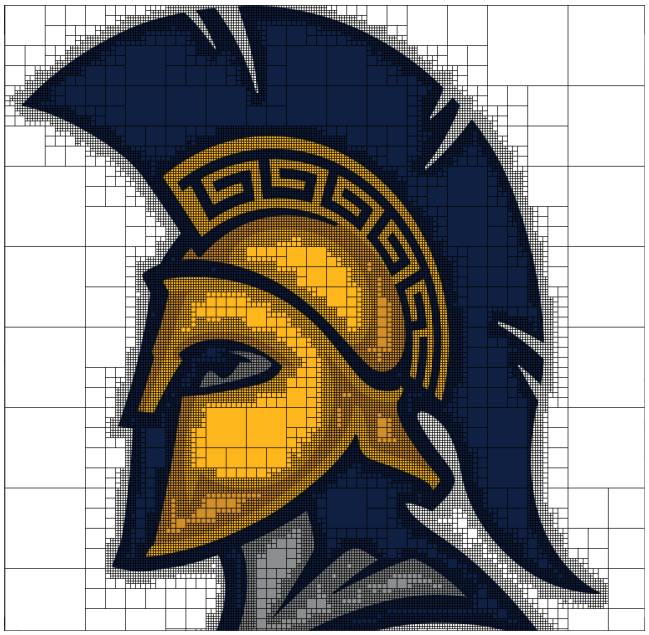


Figure 20: *Compression of the UNCG Spartan Logo*

By collectively gathering all closely-related pixels (those that are of the same color, or those that share pixel data), we can build a quadtree where any particular leaf node corresponds to a location in the canvas, in addition to the respective color that belongs to that region. Figure 20 demonstrates the compression capabilities for an image that is 1024x1024 pixels. I developed an algorithm that computes and draws the corresponding quadtree for a supplemented image in the Java programming language, that also provides simple statistics about the compression data. Before compression, the image has a raw size of 4194304 bytes. However, after compression (in a tree that has depth $h = 64$), it is 2891392 bytes in size, which is a 31.06% decrease in size.

Quadtree Compression Limitations

A couple of severe problems limit quadtrees in what they are able to compress. Firstly, the image is axis-aligned, meaning rotations are disallowed. Secondly, an image must be of a fixed size for the compression to work correctly in the quadtree (meaning the nodes must be square-shaped), and it is preferable that the image be of size $n \times n$ to ensure that shape is achieved. Moreover, the compression is slightly subjective; in that if we shift or translate the image over by any arbitrary number of pixels in any direction, we could produce a vastly different tree, effectively rendering the compression random at best, unless the conditions (and the canvas) are in the optimal position and orientation to get an effective tree. Some possible remedies include an algorithm that produce "blobs" of any arbitrary shape, as opposed to solely squared ones in stagnant locations. If this were the case, it may be feasible to create a similar tree structure, even if alterations to the original pixel data are present.

5.3 Octrees

Resembling its two-dimensional counterpart the Quadtree, an Octree is a tree data structure that represents a three-dimensional space, where every node has eight children instead of four, corresponding to a cube. As with BSP trees, octrees are commonly used to store objects in computer graphics. They also serve as a nearest-neighbor data structure in logarithmic time [13]. First, some definitions on the terminology used by an octree:

Definition 5.2. Region: A octant (or subdivision) in an octree.

Definition 5.3. Point Region: If an arbitrary region R in some octree O has a object p somewhere within its bounds, it is referred to as a *point region* These nodes are always leaf nodes (similar to quadtree nodes).

Definition 5.4. Spacial Region: If an octree O is subdivided, its root is a *spacial region*. Any subdivided regions (with children) in the tree has a spacial region as the root of that subtree.

Definition 5.5. Empty Region: Any leaf node in the octree that is not a point region with no subdivisions (or children) are *empty*.

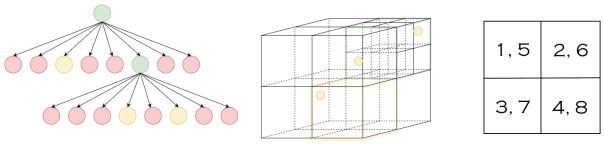


Figure 21: *Octree Spacial Representation with Two-Dimensional Tree [13].*

In the above figure, small points are scattered throughout various regions in the three-dimensional model. The far-right square shows the ordering in which the nodes are placed in the tree structure from left-to-right. In the tree, green nodes are *spacial regions*, red nodes are *empty regions*, and *yellow* regions are the point regions. We can use octrees for quick collision detection with an arbitrary point, nearest neighbor queries, data organization (almost identical to the quadtree counterpart), and others.

6 Non-conventional Rendering Methods

Our final section begins our discussion on some non-conventional methods of rendering objects or polygons in a world. These methods have not been rigorously tested or proven; they are here to demonstrate some custom approaches to the problems that we have previously mentioned.

6.1 Occlusion Rendering of Polygons

Minimizing the number of draw calls made to the renderer is of utmost priority, primarily when certain polygons or objects are drawn unnecessarily, as depicted by painter's algorithm. Acceleration structures come into play with our next topic. Suppose that we have a pseudo-three-dimensional scene as follows:

Figure 22: *Subsequent Occluding Polygons behind our viewer v.*

Where the final rendered image is

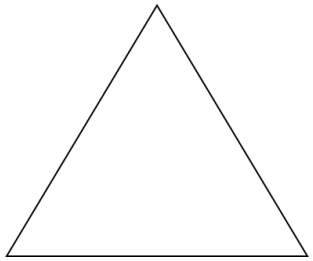


Figure 23: *Triangles rendered on top of each other.*

If these polygons happen to be oriented in the render list such that the one furthest from the viewer is at the front of the list, it is rendered first. Afterwards, the next one in succession will pass the z-depth test, and continues through our list until all other polygons are tested and rendered, entirely unnecessarily, as they are out of sight from the viewer. This is where the use of a *scene graph* comes into play.

Definition 6.1. Scene Graph: A logical or spatial ordering of objects in a scene, stored in a tree data structure. A scene graph's hierarchy is structured so that all parent nodes of children affect those children throughout the rendering process. A parent can be a process (instruction), or a node itself.

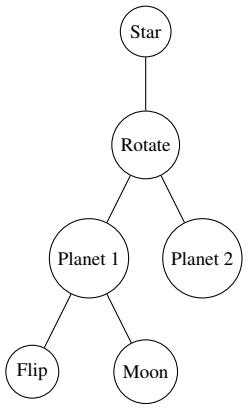


Figure 24: *Example objects where each child depends on its parent.*

In the preceding example, we render the star object first, then apply a rotation to the canvas rendering pipeline. This rotation affects all subsequent subtrees. So, both planets and their children are rotated. By using this type of graph that incorporates spatial association between geometry in the scene, we can implement a smarter system that "understands" when to stop rendering objects, because those behind it (or lower in the tree) are masked from visibility. If the children of a node are completely obscured, then we omit the draw calls altogether. Though, we still have to test z-depths because a child node may be only partially hidden behind several others. For instance, suppose a is in front of three nodes, b, c, and d. However, c is only partially covered by a. We cannot erroneously exclude drawing this subsection of the polygon simply because b is entirely hidden. This approach to ordering objects has a similar methodology to what games like *Doom* use alongside their BSP engines.

6.2 Differing Tree Structures

6.2.1 Triagonal Structure

With quadtrees, we saw that we subdivide a square region into four smaller squared regions within the parent. Our question is what if instead of using a square, we decide to use a different shape overall? Imagine a world defined by a Sierpinski triangle, where all "tiles" are triangular. Our initial triangle looks like

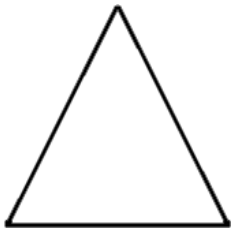


Figure 25: *Initial step of the recursive triangle.*

With one and two recursive subdivisions respectively, we get

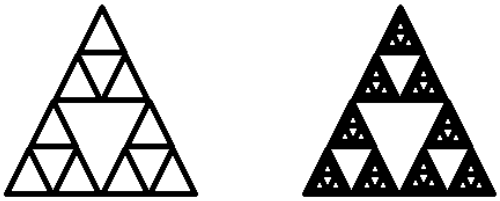


Figure 26: *One and two subdivisions of Sierpinski triangle..*

The hierarchy is similar in that we can define detail in the scene, then have the graph recursively subdivide in those specific portions. Unfortunately, we discovered one immediate drawback to this idea: there exists "dead zones" in our division.

Definition 6.2. Dead zone: Any region in a geometric representation that cannot be further subdivided.

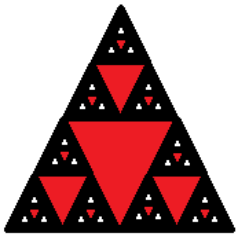


Figure 27: *Initial step of the recursive triangle.*

Though, for this example in particular, notice that the largest red triangle is identical to the top triangle, just mirrored. All we need to do for a workaround is to geometrically align and mirror the fully-red triangle with the counterpart node above it, and no dead zones are to be found.

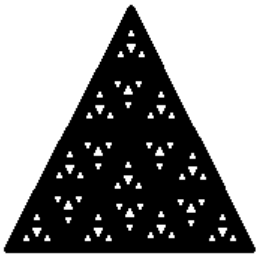


Figure 28: *Mirroring the triangle above our dead-zone fixes the issue.*

However, dealing with the geometry in this type of tree structure is less intuitive because the geometry from the node that was formerly a dead zone is now a habitable subdivision by some object or detail, and must be treated differently from the other three, as it is not a natural subdivision created by our recursive subroutine. This problem is reducible from the quadtree problem, in that we still work with four nodes at a time, just where one is treated differently.

One benefit that branches from this ideology is that if our scene is defined with triangular-regions in mind, we can use a modified quadtree to accommodate this shape and structure. Oppositely, if our scene resembles what we have previously worked with, this structure does not save or facilitate the compression or rendering process.

6.3 Parallel Rendering

We will briefly mention the concept of *parallel rendering*. Up until now, we have viewed space partitioning and rendering algorithms that are not parallel. As we have seen, rendering geometry and data to a screen is computationally expensive, and because of that, researchers have investigated methods to distribute the workload of the render problem across processors. In general, there are two processes in the typical graphics pipeline: *geometric transformation*, and *rasterization*. When we parallelize geometric transformation, each processor P is given a region R of polygons to transform. With the parallelization of rasterization each processor is given a set of pixels to manage.

We have seen that the purpose of a rendering pipeline is to determine what happens at any arbitrary pixel p in some scene s. This idea is reducible to sorting geometric polygons. In computer graphics, we can classify three sorting stages: sort-first, sort-middle, and sort-last. Each sort procedure defines how processes are distributed, as well as the overarching render pipeline structure. Sort-first is focused on distributing the polygons that we have in our list to the geometric transformation processors. In general, this is achieved by subdividing a scene into regions, such that each processor in the geometric transformation process is responsible for any polygons in the bounds of that particular region. Sort-middle uses polygons that are pre-converted into their geometric counterparts, but are not yet in the rasterization step. During each frame, the geometric processors will perform any necessary transformations on the polygons assigned to it, then pass the modified data long to the appropriate rasterizer. Lastly, a sort-last approach waits until after rasterization to perform any sorting. All renderers are, similar to their geometric processor counterparts, assigned a subset of pixels to manage. The renderers pass along the information to a composition algorithm, which solves the visibility problem of every pixel in the scene [10].

For complex scenes that are high-definition, video games, or software that works with real-time graphics, the importance of parallel rendering is apparent, peculiarly when said scenes transform overtime or contain non-static objects.

7 Conclusion

In essence, we have discussed and reviewed different researchers' approaches to solving problems in computer graphics, including potential optimizations. Some non-conventional, experimental methods were shown, alongside parallel rendering procedures to reduce the workload placed on one processor, across a network of them. Future work and research is dedicated to improving performance in scenes that change overtime, and those that contain hundreds of polygons or objects. A focus on compression is also a pertinent issue, because as the quality of scenes improves, and the number of complex objects storable in a scene increases, the required data capacity and processing power likewise rise.

References

[1] Andrew E. Matzureff. 2020. Personal communication.

[2] Csaba Bálint and Gábor Valasek. 2018. *Interactive Rendering Framework for Distance Function Representations.* Annales Mathematicae et Informaticae. 48. 5-13.

[3] Dan Gordon and Shuhong Chen. 1991. *Front-to-Back Display of BSP Trees.* IEEE Computer Graphics and Applications. 11. 79 - 85. DOI: https://doi.org/10.1109/38.90569

[4] David S. Ebert. 2000. *Region Filling*. University of Maryland, MD. Retrieved from https://www.csee.umbc.edu/~ebert/435/notes/435_ch5.html

[5] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. 1980. *On visible surface generation by a priori tree structures.* In Proceedings of the 7th annual conference on Computer graphics and interactive techniques (SIGGRAPH '80). Association for Computing Machinery, New York, NY, USA, 124–133. DOI: https://doi.org/10.1145/800250.807481

[6] Mark de Berg. 1993. *Ray Shooting, Depth Orders and Hidden Surface Removal*. Springer-Verlag. Heidelberg, Berlin.

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

[8] Martin Newell, Richard "Dick" Newell, and Tom Sancha. 1972. *A solution to the hidden surface problem*. In Proceedings of the ACM annual conference - Volume 1 (ACM '72). Association for Computing Machinery, New York, NY, USA, 443-450. DOI: https://doi.org/10.1145/800193.569954

[9] Navendu Jain, Sorav Bansal, and Sanjiv Kapoor. 2000. *Efficient Object BSP Trees.* Indian Conference on Computer Graphics, Vision and Image Processing (ICVGIP '00).

[10] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. 1994. *A Sorting Classification of Parallel Rendering.* IEEE Comput. Graph. Appl. 14, 4 (July 1994), 23–32. DOI: https://doi.org/10.1109/38.291528

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms* (3rd. ed.). The MIT Press. Cambridge, MA, USA.

[12] Wikipedia. 2012. Wikipedia: the Free Encyclopedia. Retrieved from https://www.wikipedia.org/*These images are licensed under a Creative Common License: https://creativecommons.org/licenses/by-sa/3.0/deed.en*

[13] Yash Aggarwal. 2019. *Octree data structure*. Retrieved from https://iq.opengenus.org/octree/