KTH Computer Science and Communication

Analysis of two common hidden surface removal algorithms, Painter's algorithm & Z-buffering

Daniel Nyberg

14/04/2011
Address: Skördevägen 61, 14170 Segeltorp
Telephone #: 070 494 8552
E-mail: dnyb@kth.se

DD143X, Degree Project in Computer Science, First Level
Royal Institute of Technology, Stockholm
Supervisor: Lars Kjelldahl

Abstract

This report addresses two algorithms within the field of hidden surface removal; Painter's algorithm and Z-buffering. An introduction is first made into the domain of computer generated graphics and more specifically hidden surface removal within that area, discussing methods of culling and definitions within hidden surface removal. The following sections documents the implementation, strengths, and weaknesses of the two algorithms as well as shortly discussing the data structure known as Binary Space Partitioning. An analysis is then given, comparing the algorithms with the help of specific scenarios. Finally, a discussion of combining the algorithms together with Binary Space Partitioning is given, labeling the strengths and weaknesses of this particular combination.

Sammanfattning

Denna rapport behandlar två algoritmer inom området borttagning av skymda ytor, Painter's algoritm och Z-buffring. En introduktion görs först om datorgenererade bilder i sin helhet, och sedan mer specifikt borttagning av skymda ytor, som diskuterer metoder för culling och definitioner inom borttagning av skymda ytor. I följande avsnitt dokumenteras implementationen, styrkorna och svagheterna av de två algoritmer samt en kort diskussion av datastrukturen Binary Space Partitioning. En analys ges därefter, där algoritmerna jämförs med hjälp av specifika scenarion. Slutligen finns en diskussion om att kombinera algoritmerna tillsammans med Binary Space Partitioning ges, med kombinationens styrkor och svagheter.

Contents

1 Introduction

In today's world, computer generated three dimensional images are abundant. CGI (computer generated graphics) exists in your movies, your video games, your phone, your computer's operating system. It is especially with the growth of powerful processors and graphics processors that the average homeowner has come to possess an extremely powerful piece of equipment, whose use is almost exclusively for playing video games. With this, the video game market has exploded, and has grown from a niche market in the late 70's into the global juggernaut of an industry it is today. With 3D computer games comes the problem of rendering an entire virtual world, sometimes seemingly infinite, without slowing down due to the graphics processor's limitations. This is where hidden surfaces enter the picture.
Hidden surfaces are abound in 3D computer graphics today. A hidden surface is where a surface cannot be viewed by the virtual camera e.g. the player in a video game. This renders the hidden surface useless, and thus is an unnecessary strain on the graphics processor. Hidden surface detection is especially numerous in video games and computer generated 3D effects, where the game cannot afford to put a heavy load on the processor. The same scene seen from the virtual camera can have vastly larger amount of polygons depending on if it utilizes hidden surface detection algorithms or not.

2 Problem statement

To determine if a surface is hidden, there are several hidden surface detection algorithms employed in the field of 3D computer graphics today. This report will aim to show two of the most common algorithms and compare them to each other: Z-buffering and Painter's Algorithms. Then a combination of the algorithms, together with the data structure Binary Space Partitioning is discussed.

3 Approach

Through thorough analysis, I plan to compare the two algorithms. I will achieve this by comparing the two algorithm's implementation through pseudocode, their weaknesses, and their strengths.
I also intend to determine if a combination of the two algorithms and a tree based data hierarchy, BSP (Binary Space Partitioning), is a valid algorithm to eliminate hidden surfaces.

4 Background

The visibility problem, or hidden surface detection, is one of the first major problems that arose when research began on computer graphics in the 1960's. The problem can be generalized to compute if a set of objects in $\mathbb{R}^3$ are visible from a certain camera's position and viewpoint direction. If an object is not deemed visible by the camera, then computing and rendering the object is unnecessary. There arose two different approaches to solve the problem: object space and image space [6].

4.1 Object space

Object space algorithms calculate which objects are visible in the image plane. A typical object space algorithm will therefore return a projection of the image plane from three dimensions onto two [4]. Thus, object space algorithms compute images of infinite or near infinite resolution, the image will be seen as "correct" no matter the size of the resolution [6]. However, object space algorithms have a major shortcoming. The image plane can be overly complex, and the computing can lead to polygons deemed visible by the algorithm that actually are substantially smaller than the pixel of a screen, the standard sample point. This means that in complex and low resolution cases, an object space algorithm is doing a lot of unnecessary work just to affect the values of a single pixel [4].

4.2 Image space

Image space algorithms, on the contrary, only compute which objects that are visible within a finite number of sample points, often one point per pixels. An image space algorithm can contain more than one sample point per pixel, but this is the deviation rather than the norm [3, 5]. For each pixel in the resolution, the image space algorithm computes which triangle is visible at that specific pixel. Thus, the complexity of an image space algorithm can theoretically be much smaller than the complexity of an object space algorithm, since the image space algorithm's complexity is limited to the amount of pixels in the resolution of the image.

5 Hidden surface removal

In order to analyze the solution to a problem, one must first define the problem. HSR (hidden surface removal) is the process used to determine which surfaces and parts of surfaces are not visible from a certain viewpoint. Hidden surfaces are thus defined as polygons that are not visible from the view point. The removal of such surfaces are needed in order to render full virtual environments. Removing hidden surfaces is one of the fundamental problems of generating computer graphics, and it has been so since CGI's birth in the late 60's. Sutherland, one of the early pioneers within CGI, describes the problem as a practice of including opaque objects that can be seen from the viewing eye, and omitting those that are concealed by other objects [6].

5.1 Culling

An important step that has arisen since Sutherland et al. published their findings in 1974 is the use of culling. Culling is the process in which certain faces of polygons are deemed to be invisible to the camera and deleted. This is usually done very early in the rendering pipeline, before any HSR algorithms are used. Culling helps remove the strain on the algorithm by deleting the surfaces that are more easily determined hidden. This is especially true if entire objects are determined to be hidden and thus culled, as these objects no longer need to be fetched, sorted, rasterized and so on. There are many forms of culling, ranging from back-face culling and view frustrum culling to occlusion culling.

5.1.1 Occlusion culling

As Hansong Zhang states in his Ph.D. Dissertation, "...the aim of occlusion culling is to detect large numbers of non-visible primitives and remove them as early as possible."[8]. Occlusion culling is a large part of the rendering pipeline, and is often done first, before other culling methods and especially before HSR algorithms. Like the figure below illustrates clearly, it is unnecessary to render the hidden objects in the scene, marked in red in the figure. The depth buffer of Z-buffering could perform this task as well and get an accurate scene, but it would strain the system because of the extra polygons. A fast occlusion culling can help eliminate a huge part of the scene, so as to speed up the rest of the rendering pipeline.

Figure 1: How occlusion culling works.

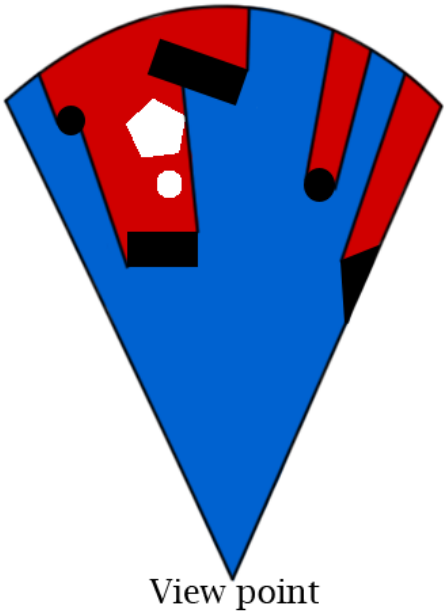
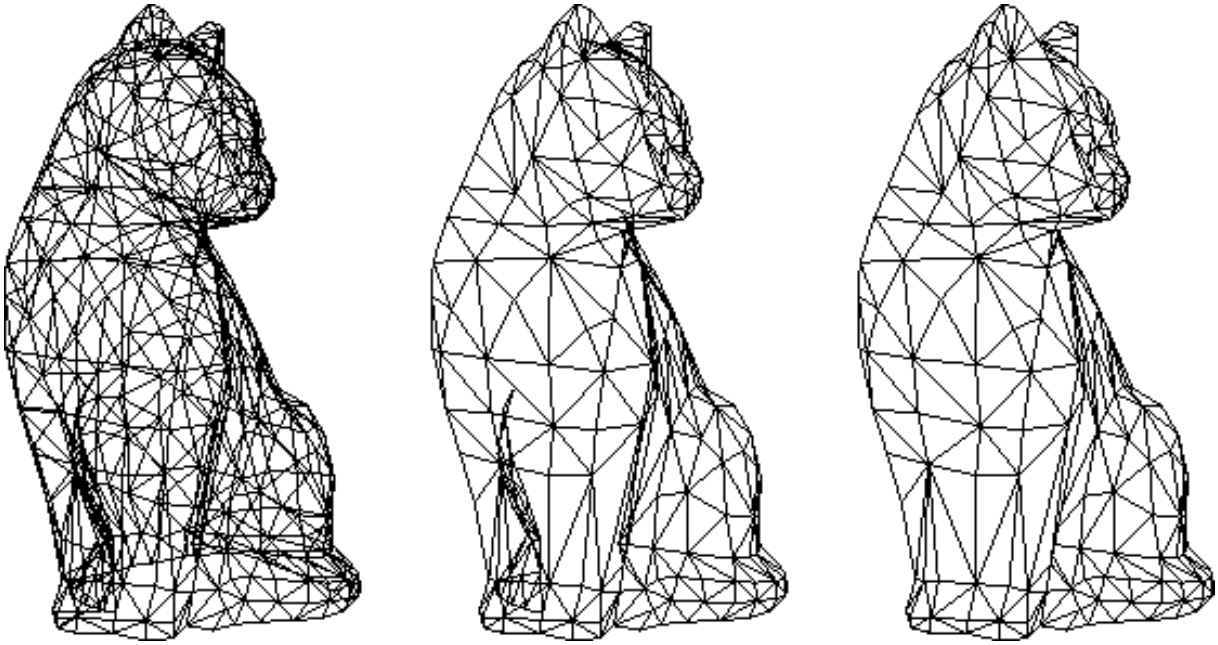


Figure 1 helps illustrate how occlusion culling works. The blue area is the area that is visible from the view point. The red area is hidden from the view point. The white objects are occluded, since no polygon that is part of the object can be seen from the view point.

5.1.2 Back-face culling

Back-face culling is a useful tool to use prior to using a HSR algorithm. In it the faces of the polygons of an object as subjected to a back-face test. If this test is failed, the face isn't drawn. A typical back-face test is to calculate the dot product between the vector formed by the viewpoint to the polygon, and the polygon's normal. Depending on the method used, a negative or a positive value of the dot product identifies a polygon that is facing away from the view point, and thus is subjected to culling [7]. A fast back-face culling implementation can help speed up the rest of the scene rendering considerably. It can also create fewer surfaces that need to be detected later in the hidden surface removal phase.
To further illustrate the usefulness of back-face culling before executing a HSR algorithm, see figure 2. The cat on the left is the original wireframe version of the rendered cat, with all it's polygons intact.

Figure 2: A demonstration of the usefulness of back-face culling [10].



The cat in the middle has been altered with the help of back-face culling. As can be seen, several polygons have been removed, as they were deemed to be facing away from the camera. The cat on the right has had it's remaining hidden surfaces removed with a HSR algorithm. Note how few polygons actually had to be tested for hidden surfaces compared to the original cat.

6 Painter's algorithm

Painter's algorithm was one of the first solutions proposed for the HSR problem and it's understandable why. Painter's algorithm, like its name implies, is based on the method an artist paints a scene on a canvas. The objects which are farthest away from the virtual camera are "painted" first, then the objects in the image's middle ground, and finally the objects that are closest to the camera are painted. In this way, objects that are not visible are simply "painted" over, resolving the visibility problem. Figure 3 illustrates this.

Figure 3: Painter's algorithm paints the image from back to front, much like a real painting.

The idea behind Painter's Algorithm is simple to grasp and was widely used until other more efficient algorithms for Hidden Surface Removal arose.

6.1 Implementation

Painter's algorithm can easily be described in the following pseudocode:

Algorithm 1 Painter's algorithm in pseudocode

```
sort polygons by z;
for all polygon p do
  for all pixel(x, y) ∈ p do
    paint polygon.colour;
  end for
end for
```

As an object space algorithm, all objects in the scene must be taken in consideration. Therefore, all the polygons in the scene are first sorted by depth, using some sorting algorithm. A polygon's depth can be defined in a number of ways, ranging from determining the z-value in the middle of the polygon, to the z-value farthest from the view point. Depending on the definition used, they give different inherent problems in special cases, some of which are discussed later in this report. After all the polygons have been sorted according to their z-values, they are painted from back to front. Each polygon in turn gets all it's pixels painted, until the final and closest object is painted.
We define k as the number of polygons in a scene, and n as the number of pixels in

the resolution of the screen. It is evident that Painter's algorithm painting from back to front, looping through each pixel of each object, has a complexity of $O(kn)$. This is disregarding the sorting algorithm used for sorting each object according to it's z-value. The efficiency of Painter's algorithm essentially depends on the method of sorting, but using a simple sorting algorithm a worst case complexity of $O(n^2)$ is achieved, with an average complexity of $O(nlog(n))$.

6.2 Strengths

The main strength of Painter's algorithm is that its theory is uncomplicated. This facilitates its implementation immensely and is the sole reason that it is still taught today, even though it is widely regarded as one of the worst HSR algorithms, due to it's many faults.
Painter's algorithm does however have another strength when it comes to transparent polygons. A common problem with transparent polygons and HSR is that the surfaces behind the transparent polygon have already been removed. Painter's algorithm doesn't have this problem as it paints over pixels from the back to the front. When the time comes for the transparent polygon to be painted, the polygons behind the transparent polygon have already been painted. Thus, the transparent polygon can remain truly transparent.

6.3 Weaknesses

Painter's algorithm suffers from many inherent flaws and weaknesses, made evident by it's implementation.
First and foremost, it is highly dependent on an efficient depth sorting algorithm. If a slow sorting algorithm is chosen, Painter's algorithm usefulness will be heavily affected for the worse. This, however, is also part of Painter's algorithm's strengths. If a suitable preprocessor is used to, for instance, produce a tree of the z-values, a simple depth first searching algorithm could be used to identify which polygons are hidden by others, allowing an implementation where the polygons could be painted from front to back, locking each subsequent polygon's pixels so they can't be repainted.
This brings us to the next major imperfection in Painter's algorithm. It paints from back to front, meaning that there is always a risk that the majority of a polygon's surface may be repainted. This is profoundly inefficient and can lead to a long rendering time in scenes with many polygons slightly covering other polygons behind them.
The third unquestionable flaw of Painter's algorithm is that the 2D projection produced is dependent on how the depth of an object is determined. As mentioned earlier, the z-value of an object can be defined as the middle of the polygon, the closest edge of the polygon in relation to the viewing point, vice versa the farthest edge, and so on. These definitions give rise to certain cases that Painter's algorithm either can't handle or is exceedingly slow at handling. Cyclic images like the one in figure 4, are especially troublesome for the algorithm.

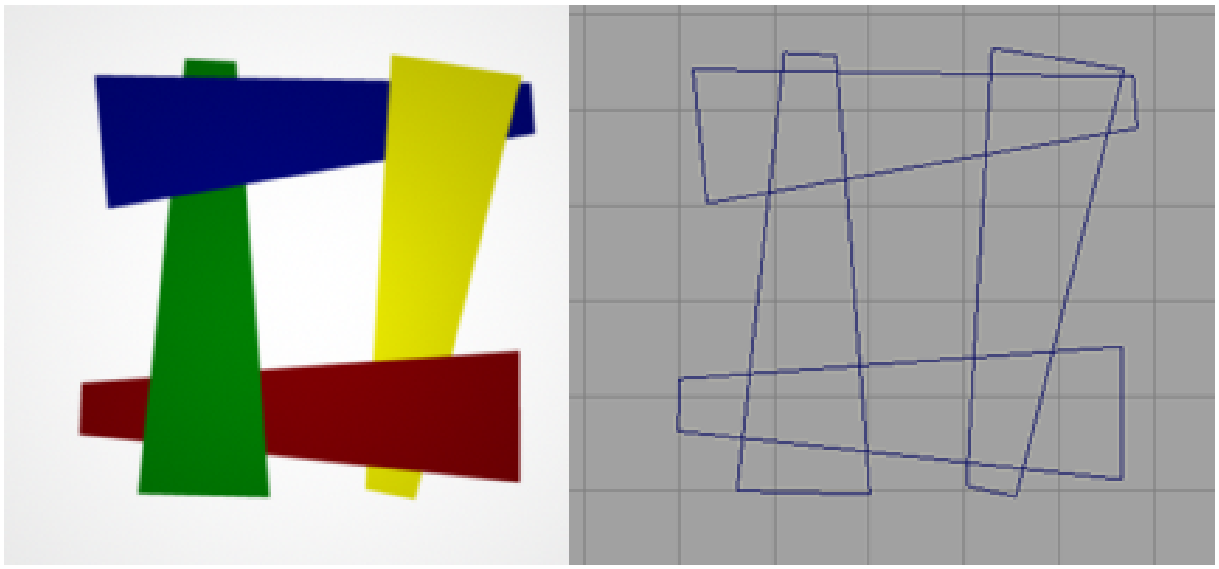


Figure 4: A cyclic image that Painter's algorithm finds complicated

This is a fairly common problem, and the more complex an image is, the harder it is for Painter's algorithm to handle. In order for Painter's algorithm to be able to paint the intended image, the polygons must first be divided into smaller polygons with different z-values. This facilitates the painting part of the Painter's algorithm's implementation, but greatly lengthens the sorting, the major hangup of Painter's algorithm's efficiency. However, to create such an algorithm that divides polygons into smaller polygons, and does this in the general case as opposed by the specific case, is both a very time consuming as well as complex endeavor.

7 Z-buffering

A much more modern algorithm that is in full use in today's CGI is Z-buffering. First introduced in 1974 by Edwin Catmull, it has grown to a staple in modern graphics processing and is implemented in a majority of today's GPU:s [2].

7.1 Implementation

The Z-buffering algorithm can be described with the following pseudocode:
That Z-buffering is an image space algorithm is clear to see from its implementation.

Algorithm 2 Z-buffering in pseudocode

```
for all polygon p do
  for all pixel(x, y) ∈ p do
    if p.pixeldepth(x, y) < zbuffer(x, y) then
      zbuffer(x, y) = p.pixeldepth(x, y)
      framebuffer(x, y) = p.pixelcolour(x, y)
    end if
  end for
end for
```

The algorithm goes through every polygon in the scene and determines the z-value of each pixel that creates those polygons. If the determined z-value of the pixel is smaller, which means it's closer to the view point, then the Z-buffers value at that point is changed to the pixel's value, and that pixel's colour is saved in the frame buffer. Z-buffering uses two matrices the size of the resolution of the screen, the z-buffer and the frame buffer. The z-buffer contains the z-values of each individual pixel and is initialized as the max depth of the scene. The frame buffer contains the RGB values of each individual pixel and is initialized as the scene's background colour. As an image space algorithm, the resolution of the scene is the most relevant when it comes to time complexity, the amount or the complexity of the polygons can nearly be disregarded.

7.2 Strengths

Z-buffering has numerous strengths, which is unsurprising for an algorithm that is implemented in nearly all GPU:s today.
The biggest strength of the Z-buffering algorithm is that it is online. This means that the algorithm doesn't need all the data from the scene right away. As soon as Z-buffering gets a polygon from the scene, it can start computing the z-values of the pixels in the polygon and putting the values in the z-buffer and frame buffer. This makes Z-buffering very efficient, as it doesn't have to wait for the entire scene to load, but can compute while it's still receiving data of the scene.
Another big advantage is that Z-buffering is a fairly simple algorithm, so it can easily be implemented in the hardware of a graphics card. For instance, the graphics processor company ATI implements Z-buffering with the ATI Hyper Z technique in their ATI

Radeon graphics cards.

7.3 Weaknesses

Z-buffering is not without its flaws, however.
The matrices that Z-buffering uses for the z-buffer and the frame buffer can take up a lot of memory depending on the resolution of the scene. For instance, if the scene is 1080p or 1920 x 1080 pixels large, then both the z-buffer and the frame buffer need to be 1920 x 1080 memory units large. This is quite a bit of memory allocated to the algorithm's needs. However, the actual testing used by Z-buffering is so fast that it makes up for this fact.
Z-buffering also cannot handle transparent polygons, as the polygons that are behind the transparent polygon have already been culled by the algorithm. A simple solution to this is to flag any transparent polygons and render them last. This means that the transparent polygon is still rendered and the polygon behind it can still be seen. In rare cases where a transparent polygon are placed behind another transparent polygon however, the view point will not be able to see through both polygons, as the polygons behind the second transparent polygon has already been culled.
Like Painter's algorithm, Z-buffering suffers a bit of repainting. Certain pixels in a scene may be repainted many times for each scene, though this is highly unlikely, and is a much larger problem for Painter's algorithm.

7.3.1 Z-fighting

The major problem that Z-buffering suffers from is z-fighting. Z-fighting occurs when two polygons are so close together that they overlap on virtually the same z-value. This makes it impossible for the depth test to determine which polygon is on top of the other and thus what colour the pixels should have. Instead, the algorithm chooses the pixel's colour at random.
The cause of z-fighting can be a myriad of factors, but in most cases it is caused by rounding errors or a poor choice of depth precision. A depth precision is chosen when the algorithm is initialized and constitutes the amount of possible z-values in the scene that the z-buffer can handle. The standard precision used is 16 bit, however 24 bit and even 32 bit precision z-buffers are becoming more and more prominent. A 16 bit z-buffer can handle up to 65,536 different values, while a 24 bit z-buffer can handle up to 16,777,216 values.

With more possible values for the z-buffer to take, a high precision helps combat z-fighting. An example of z-fighting can be seen in figure 5. The image on the left is the intended image and the image on the right shows z-fighting. Notice how the dark polygon seems to create a pattern or rasterization across the surface of the lighter polygon. A higher precision in the z-buffer fixes the image so that it looks like the one on the left.

Figure 5: An example of z-fighting

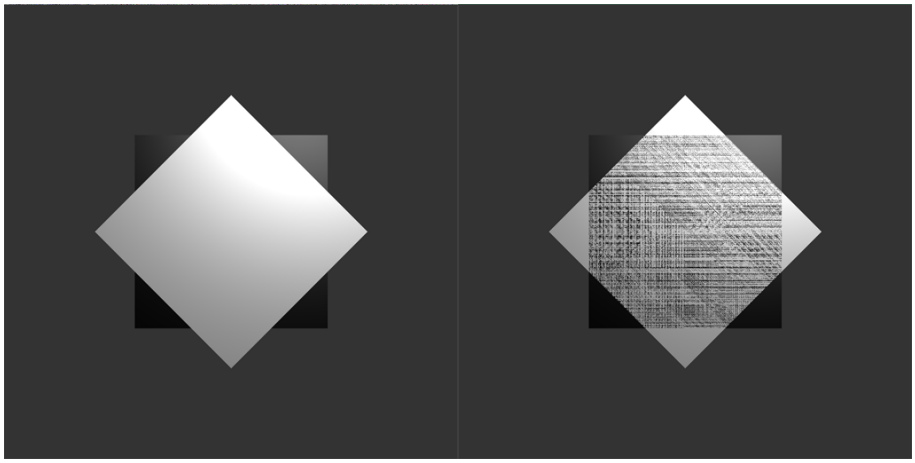
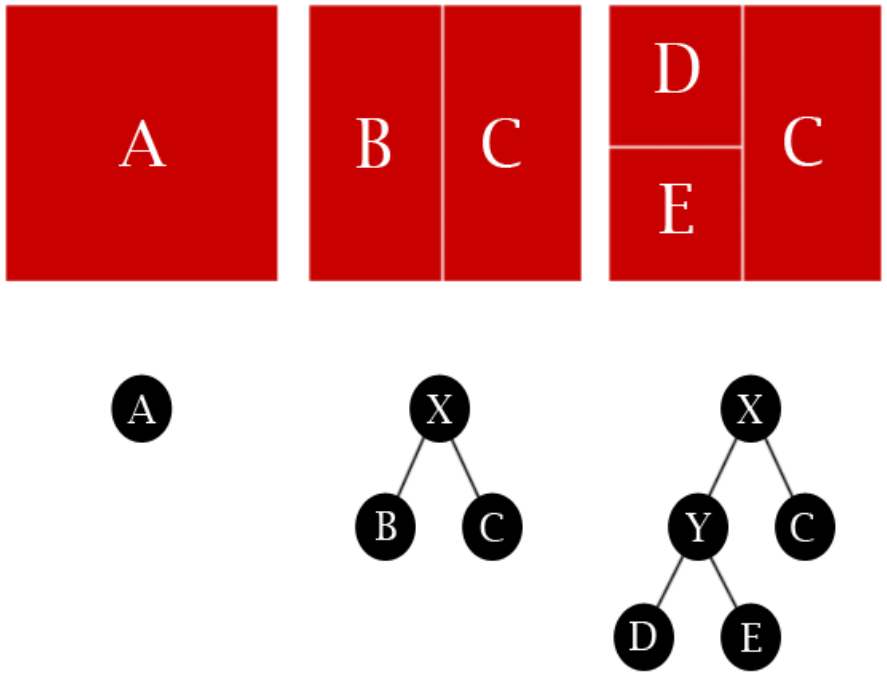


Figure 6: The first three stages of a BSP



8 Binary Space Partitioning

Binary Space Partioning, or BSP, is a common method for dividing up space in a scene so that it can easily be sorted and traversed when needed. A BSP tree can be seen as a data structure utilizing recursion and hierarchies within the subdivision of an n-dimensional space into convex subspaces [9]. BSP trees incorporate powerful sorting and excellent organization structures, and are used in many fields. Among them are solid modeling, ray tracing hierarchies, and our field, hidden surface removal. BSP trees are also used to store levels in popular games such as Doom and Quake, and help determining hidden surfaces in those games [1].

8.1 Implementation

BSP trees is a standard binary tree that has been adapted to space partitioning, where the entire tree is the total space in the scene, and each node is a convex subspace consisting of a hyperplane dividing it's space into two halves [9]. To more easily understand how a BSP tree divides space, see figure 6. It represents only two dimensions, but will do fine for explanation purposes. The first division cuts our polygon in half across the X axis, the second along the Y axis. This is then done recursively until a suitably small and manageable space is found in each leaf of the tree.

8.2 Combining BSP trees with other algorithms

The main strength of the BSP tree is the efficiency boost that it provides for other algorithms. As a BSP tree is precomputed before the rendering process of a scene and is view independent during it's construction, it is ideal for speeding up HSR algorithms, such as Painter's algorithm and Z-buffering.

8.2.1 Combining with Painter's algorithm

Thanks to a BSP tree sorting in a depth first traversal, it is very efficient at speeding up Painter's algorithm, as it paints from back to front. The tree also helps the algorithm handle cyclic images by splitting the space up into manageable chunks for the algorithm to render. This slows down the render time, but ultimately means that the algorithm can handle cases that were near impossible or tediously long to compute prior.
With the BSP doing the initial sorting, Painter's algorithm would take considerable less time however. It takes the BSP $O(nlog(n))$ to sort and divide up the space, and a further $O(n)$ to traverse the created tree, as opposed to $O(n^2)$. However, the worst case still stays the same [9].
The big downside to this implementation is that the preprocessing time would skyrocket, making it virtually impossible to incorporate Painter's algorithm in an environment where polygons move and interact. The BSP tree would need to be rebuilt for every frame, making the rendering very slow.

8.2.2 Combining with Z-buffering

While a combination of Painter's algorithm and a BSP tree means that it is useless when it comes to rendering scenes with moving or movable objects; a combination of Z-buffering and a BSP tree can handle such scenes. This is done by letting the z-buffer merge with the movable objects. In the case of a computer game, this can be objects such as movable doors, monsters, and so on. The BSP eliminates the need for Z-buffering to check each pixel's depth, as each polygon's depth has already been precomputed.
A BSP tree also helps to eliminate z-fighting, as the polygons are split into smaller chunks that can be processed individually. This means that the chunk is handled with a much higher precision than if it was part of an entire polygon, giving more accurate z-values.

9 Analysis

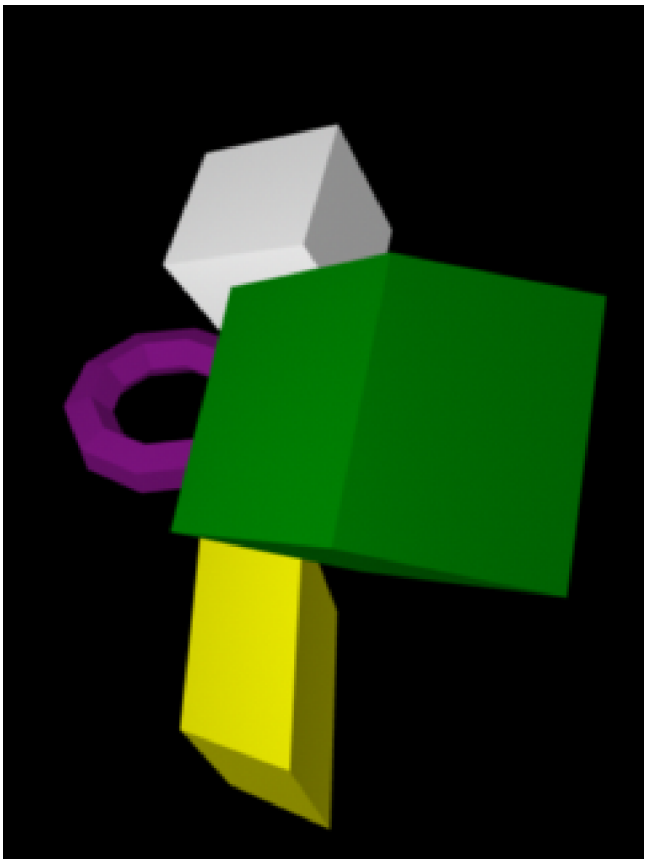
9.1 Comparison of Painter's & Z-buffering

To compare two algorithms such as Painter's and Z-buffering, which lie on two separate ends of the spectrum when it comes to hidden surface removal, one needs to test them objectively. To do this I propose the following theoretical scenarios to measure. To help visualize the different scenes that will be analyzed, I have rendered them in the modeling program Autodesk Maya.

9.1.1 Low resolution & low polygon count

The scene being rendered can be seen in the figure below. It contains 68 polygons and has a resolution of 320 x 240 pixels.

Figure 7: Low resolution & low polygon count



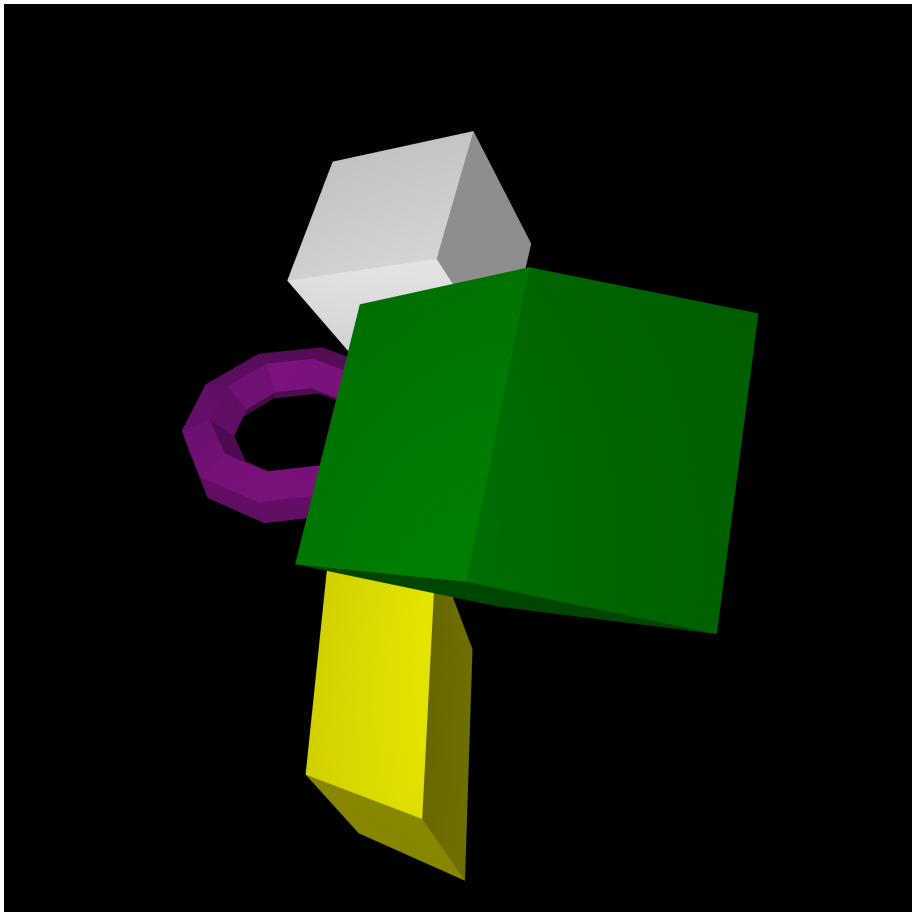
As the scene contains few polygons and has a small resolution, both Painter's algorithm and Z-buffering will paint the scene quickly. With a little perception however, one can see that even in this very simple scene Painter's algorithm will repaint a portion of

the pixels. Nevertheless, while using BSP in combination with the algorithms this shortcoming is surpassed. Painter's largest flaw, the slow sorting, is not a problem in this scene as it contains so few polygons. Z-buffering on the other hand has no problem with this scene, as it is fairly small in resolution and consequently removes any hidden surfaces quickly.

9.1.2 High resolution & low polygon count

The scene being rendered can be seen in the figure below. It contains 68 polygons and has a resolution of 4096 x 4096 pixels.

Figure 8: High resolution & low polygon count

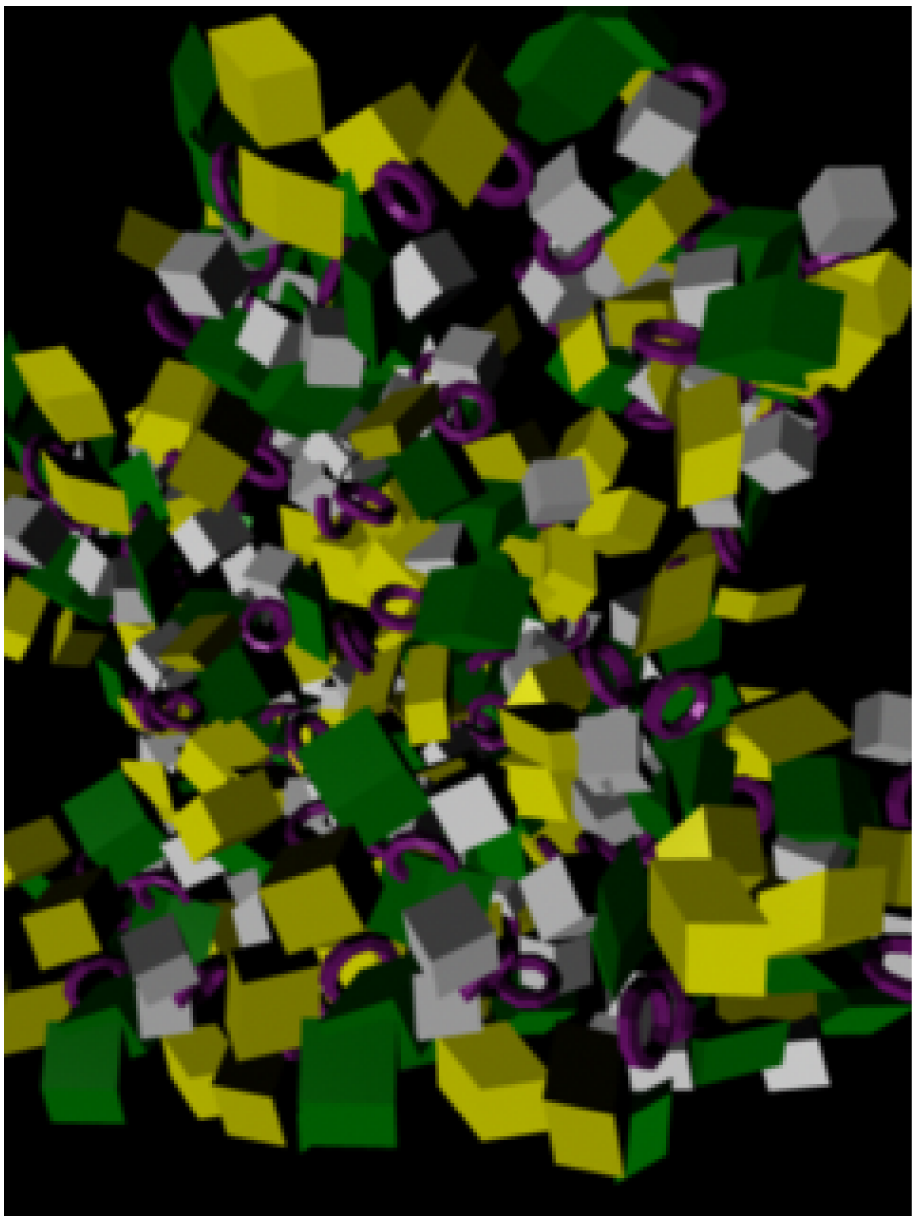


This scene is identical to the above scene, except that it has a far higher resolution. While this impacts Painter's algorithm's performance slightly, with it having to paint more pixels per polygon, it is ultimately Z-buffering that takes a toll. For every pixel, all 16,777,216 of them, the z-buffer needs to check the z-value in relation to the z-buffer. Both the z-buffer and the frame buffer are of immense size, which slows Z-buffering down.

9.1.3 Low resolution & high polygon count

The scene being rendered can be seen in the figure below. It contains 8704 polygons and has a resolution of 320 x 240 pixels. The scene contains a prodigious amount of polygons. This will slow down both of the algorithms considerably, albeit Painter's algorithm takes the heavier penalty.

Figure 9: Low resolution & high polygon count



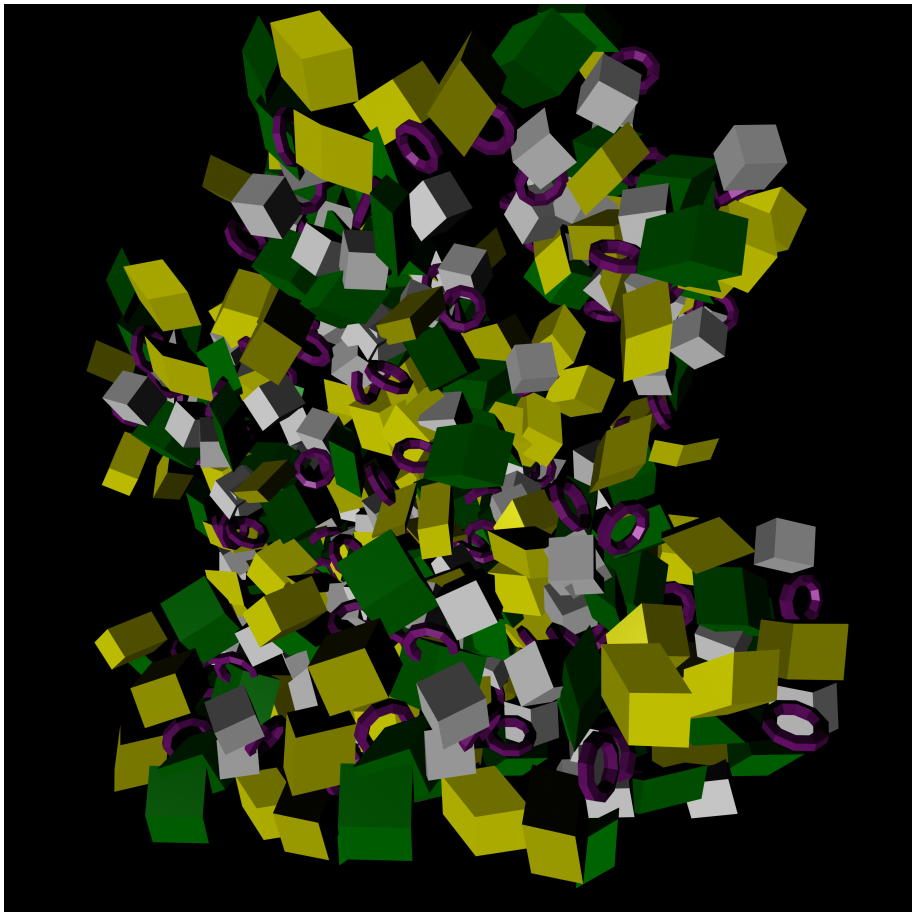
Painter's first has to wait for all the polygons in the scene to be sorted before it can start painting them. This takes considerable time, and even with a BSP implementation, it takes an average time of $O(nlog(n))$. Even with taking this into account, the algorithm

will still repaint a majority of the pixels, as the low resolution of the image means that every pixel in the scene has at least one hidden polygon.
Z-buffering will also encounter problems with the scene. Unlike Painter's algorithm, Z-buffering is online, so it doesn't need to get every polygon from the scene before it can start. Despite this, it is still possible for the algorithm to repaint many pixels if it gets the polygons in the scene in a back to front order. This is highly unlikely however, and Z-buffering is still fast at this low resolution as a majority of the polygons, due to the depth involved in the scene, are implausible to have a significant amount of pixels per polygon. Each polygon reasonably only has a handful of pixels each.
Both algorithms also have to deal with the problem of complex images. Be it a near impossible case or z-fighting, it is likely to come up while eliminating hidden surfaces in the scene. This can be combated by combining the algorithms separately with a BSP tree, though the preprocessing time of each will increase, even if Painter's algorithm gains some speed in the painting stage thanks to the sorting of depth.

9.1.4 High resolution & high polygon count

The scene being rendered can be seen in the figure below. It contains 8704 polygons and has a resolution of 4096 x 4096 pixels. This scene, like the previous one hampers the two algorithms and takes them to their limit. This is the scene that closest mimics a HSR algorithm's work in the real world; a complex scene with a high resolution that needs to be rendered as quickly and efficiently as possible.

Figure 10: High resolution & high polygon count



Painter's algorithm will, like in the previous low resolution version, will have to wait for all the polygon's data before it can start eliminating hidden surfaces. The biggest difference between the low and high resolution scene though, is that the algorithm will with a high likelihood repaint a large amount of pixels in the scene. There will be a lot of unnecessary work during this stage.
As in the previous case of a higher resolution, Z-buffering needs a larger z-buffer and frame buffer, meaning more memory is needed for it to remove hidden surfaces. The inner loop of the algorithm will increase the performance time, and there will still be some cases of repainting certain pixels, but not to the same extent as Painter's algorithm.

9.2 Conclusions

In lieu of the strengths and weaknesses of the two algorithms gleaned in the previous section, it is clear that the algorithms' efficiency is dependent on the scene's complexity and resolution. If the scene has a small resolution the algorithm used doesn't matter as much as the speed of the algorithm.
The lower the complexity of the scene, the more Painter's algorithm is favored in this

scenario, especially if it's combined with a Binary Space Partitioning tree. However, it doesn't have the advantage of being online like Z-buffering, and must therefore wait for all the polygons' data before starting, so depending on the scene that the advantage of Painter's algorithm because of low scene complexity is negligible.
When a scene with a large resolution is handled, Painter's isn't impacted as severely as Z-buffering, except for having more costly repainting. Being offline comes back to haunt the algorithm though, so Z-buffering is the preferred method to use. Z-buffering can guarantee a correct and intended image, while Painter's algorithm needs to be combined with a BSP tree and a front to back implementation for it to paint a correct image. The drawback of Z-buffering is the amount of memory needed for the matrices, but in today's world where most graphics cards have at least 1 Gigabyte of memory, this is not a big problem.

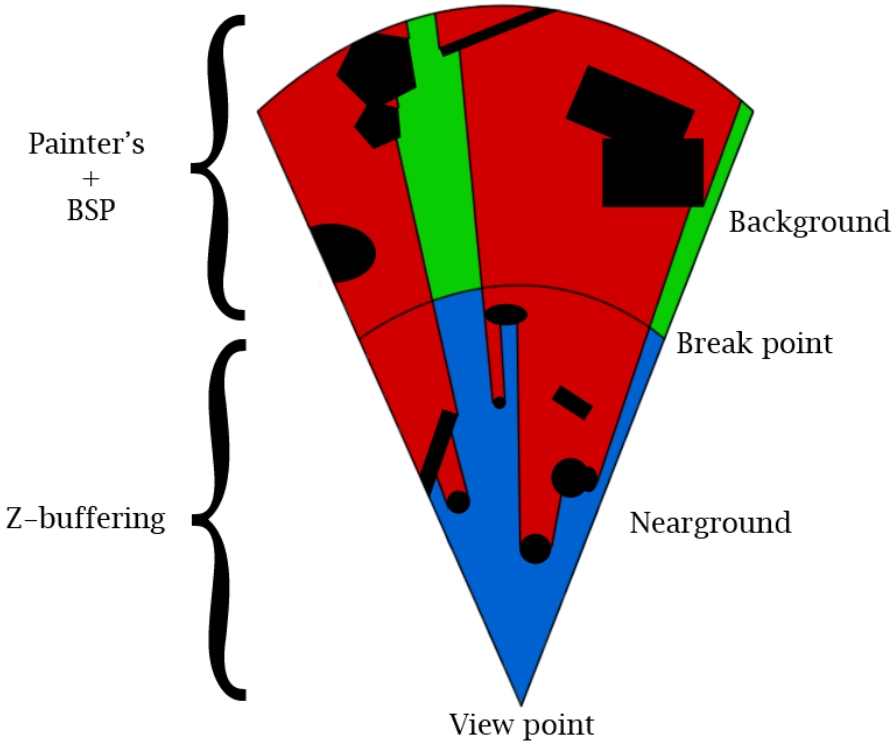
10 Discussion

Clearly, Painter's algorithm cannot cope with today's CGI computing needs. It is, however, an elegant and simple solution to a problem as old as CGI itself. If one were to combine the straightforwardness of Painter's algorithm with the z-buffer depth tracking of Z-buffering, and with the hierarchical tree structure of a BSP tree, a more ingenious solution to the hidden surface removal problem could be found.

10.1 Combining Painter's with Z-buffering and BSP trees

So how does one combine two algorithms and a hierarchy? What strengths are kept from which algorithm, and will the new algorithm have new problems?
I propose implementing a break point of sorts for the calculation of hidden surfaces. First in the rendering pipeline a occlusion culling is performed, followed by back-face culling. This will remove most of the hidden polygons in the scene. Next, a suitable clipping plane is established so that most of the detail, or if you will, a high percentage of the polygons in the scene is in the space between the near clipping plane and the break point. This space will hereby be called the nearground. Then a BSP tree is built with the depths of the polygons in the space between the break point and the far clipping plane. This space will hereby be called the background. A BSP is great for establishing backgrounds, and has been used for this in games like Doom and Quake.
While the BSP tree is being built, an implementation of Z-buffering is applied in the nearground. The nearground is great for details such as characters, objects, weapons, and the like, as they can have a high level of details with many polygons. Z-buffering will ensure that the polygons are placed correctly, and the z-buffer will also help render movement more easily, as creating a z-buffer is faster than creating a new BSP in conjunction for each frame. The z-buffer and frame buffer will subsequently lock down which pixels will obscure the background. The highest value in the z-buffer, the break point in the clipping plane, will help determine which pixels the background space may use. The frame buffer at this break point will create a transparent polygon to let the background through.
When the BSP is completely built, a front to back implementation of Painter's algorithm is used to render the polygons in the background space, using a copy of the z-buffer from the nearground to determine which pixels may be painted or not. The algorithm will still calculate where the polygons in the plane are, but will not fill in those that are already covered in the nearground, thus eliminating repainting of pixels.

Figure 11: High resolution & high polygon count



The figure illustrates my combined algorithm more clearly. The basic principle is that pixels covered by the z-buffer in the nearground can't be repainted and aren't even considered in the front to back implementation of Painter's algorithm in the background. The blue area signifies what the Z-buffer will paint. The red area indicates hidden space, that will not be rendered because of objects in front of the space hiding it. The green area denotes the area that the front to back implementation of Painter's algorithm will paint.

10.1.1 Strengths

The main strength of this combined algorithm is that it removes any and all repainting that occurs in the algorithms separate implementations. This is especially helpful for Painter's algorithm, stopping it from repainting parts of the background over and over again.
The Z-buffer gets a better precision, as it won't have to compute z-values all the way to the far clipping plane. This higher precision helps eliminate z-fighting, as discussed earlier in this report.
Another strength is that the background won't need to be wholly rendered, as the z-buffer copied from the nearground informs the algorithm which pixels need a background, reducing redundancy and, ultimately, performance time.

Z-buffering is widely supported by hardware in GPU:s which also helps speed up that part of the algorithm.

10.1.2 Weaknesses

A major weakness of the algorithm is the processing time. It will take time to make a suitable, general case break point, and build a BSP tree. If this algorithm were to be used in a game, this could be a problem if the environments are too large or overly complex. The algorithm does suit fine though with more "indie" styled games that don't focus on the pinnacle of graphics effects, and don't contain as many polygons as a state of the art game.
Another weakness is that the Z-buffering algorithm won't be online any longer. It will first need to wait for the data of the polygons that are in the nearground before it can start. It will still be online in the remark that it doesn't matter which order it gets these polygons however, just that they need to be sorted in nearground and background first. While this may seem insignificant, it can extend the performance time of the overall algorithm longer, as Z-buffering waits for its polygons to work on. If the polygons are tested, by chance, back to front in the scene, then Z-buffering will have to wait a long time before it can begin. This is a highly implausible scenario though, even if it is a bottleneck of the algorithm.
Movement that occurs in the background will slow down the algorithm further, as each time the background needs to be repainted, a new BSP tree must first be constructed or the existing tree reevaluated. This can be a major hangup of the system, slowing down the frame rate without the user understanding why. A solution to this problem could be to have a separate z-buffer in the background that keeps track of the movement of polygons, merging it with the BSP tree. Coupled with levels of detail, this solution could be made fairly efficient, though it goes beyond the scope of this report.
Thanks to using a BSP tree with Painter's algorithm, the cases that the algorithm finds difficult are of no concern, although the same can not be said for Z-buffering's problems. The frame buffer and the z-buffer will still need an immense amount of memory during large resolution rendering, and z-fighting will still occur, albeit not as much thanks to the added precision. As well, the trouble with transparent polygons still remains.

10.2 Algorithm for today's needs

Which algorithm is the most effective for today's CGI computing needs? This is a very complicated question. From the years of the pioneers in the field, Sutherland et al., much has changed in the world of CGI, many new algorithms, structures, and methodologies have arisen and fallen into the waters of oblivion, forgotten. Painter's algorithm was used before more effective means were found, and is today only a part of hidden surface removal's history. Z-buffering on the other hand, being the most widespread solution, integrated into the hardware of our computers and found in most game engines, will likely go down in history as one of the first really successful HSR algorithms. It will assuredly be replaced in time by another, more flexible and efficient algorithm, but it stands to say that in this day and age it rules supreme over the CGI market as the go-to for hidden

surface removal.

11 References

1. Brogan, Bunt 2006 *Oblique Reflections: Software Art & the 3D Games Engine*, Cyber Games'06, ISBN: 86905-901-7

2. Catmull, Edwin 1974 *A Subdivision Algorithm for Computer Display of Curved Surfaces*, Thesis at the University of Utah, order number: AAI7504786

3. De Berg, Mark 1993 *Generalized Hidden Surface Removal*, SCG '93 Proceedings of the ninth annual symposium on Computational geometry, ISBN: 0-89791-582-8

4. Erickson, Jeff 2000 *Finite-Resolution Hidden Surface Removal*, SODA '00 Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms, ISBN: 0-89871-453-2

5. Sharir, Micha 1992 *A Simple Output-Sensitive Hidden Surface Removal*, ACM Transactions on Graphics (TOG), Volume 11 Issue 1, ISSN: 0730-0301

6. Sutherland, Ivan E. & Sproull, Robert F. & Schumacker, Robert A. 1974 *A Characterization of Ten Hidden-Surface Algorithms*, ACM Computing Surveys (CSUR), Volume 6 Issue 1, ISSN: 0360-0300

7. Zhang, Hansong & Hoff, Kenneth E. 1997 *Fast Backface Culling Using Normal Masks*, I3D '97 Proceedings of the 1997 symposium on Interactive 3D graphics, ISBN: 0-89791-884-3

8. Zhang, Hansong 1998 *Effective Occlusion Culling for the Interactive Display of Arbitrary Models*, Ph.D. Phesis at the University of North Carolina at Chapel Hill, UML Order number GAX99-14933

9. Pan, Zhigeng & Tao, Zhiliang & Cheng, Chiyi & Shi, Oiaoying 2000 *A New BSP Tree Framework Incorporating Dynamic LoD Models*, VRST '00 Proceedings of the ACM symposium on Virtual reality software and technology, ISBN: 1-58113-316-2

10. Demonstration of back-face culling on a cat model, University of Waterloo, Introduction to Interactive Computer Graphics, http://medialab.di.unipi.it/web/IUM/Waterloo/node70.html#SECTION00116000000000000000, published June 11 1996, downloaded April 14 2011