

Courseware for Hidden Line/Surface Removal

Thomas Cheah.C.S.

Chia Wei, Tan

Patricia Malone

4 December 2001

Abstract

When objects are defined as surfaces filled by color or shading patterns, hidden-line/surface removal techniques are used to take out any back surfaces that are hidden by visible surfaces. Removing hidden surfaces is generally a complicated and time-consuming process, but it provides a highly realistic method for displaying objects. Added realism is attained by combining hidden-surface removal with perspective projections.

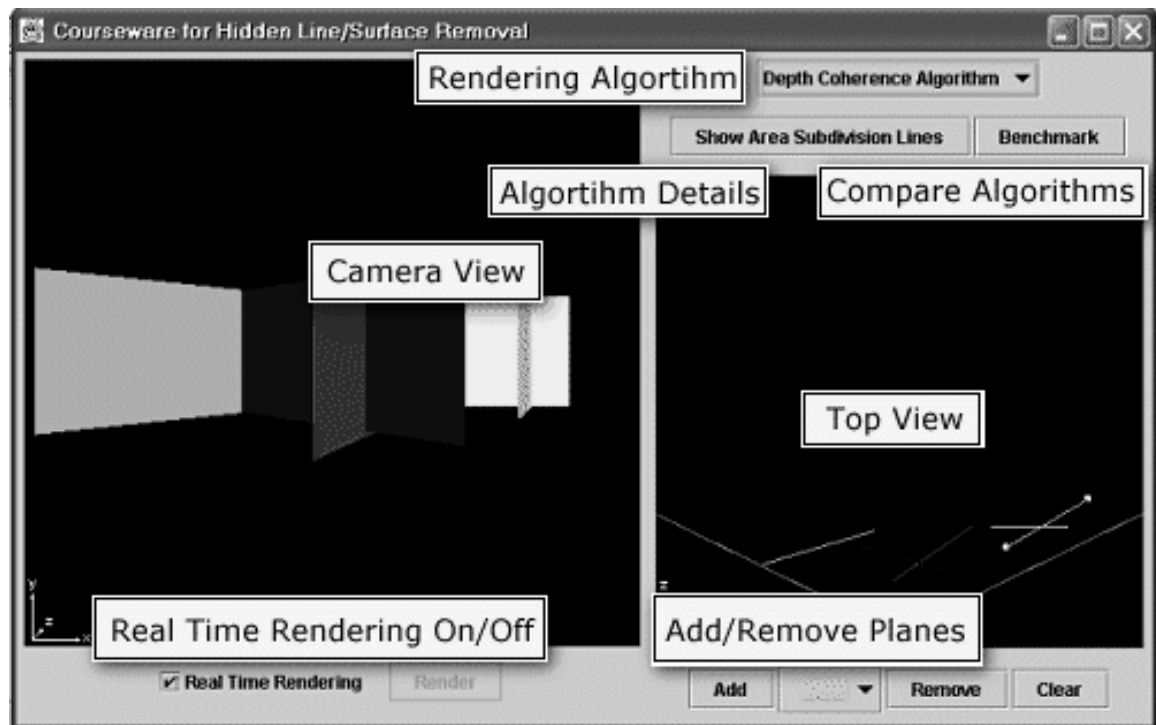
This courseware will provide you an interactive way to experiment with the three typical (and the most well known) hidden line/surface removal algorithms. They are Painter's, Z-Buffer, and Depth Coherence (Warnock's Area Subdivision) algorithm. While experimenting, you will see the results for different algorithms. You will also learn the pros and cons among each different algorithm. The documentation will provide you the details of the courseware implementation.

Courseware Descriptions

This courseware will render several rectangular planes that are placed by the user anywhere in the 3-D world. The rendering will be done by a simple 3-D engine developed by us, with the three hidden surface removal algorithm implemented. User can see the result of different algorithms in the Camera View, which is a 3-D perspective view of the 3-D world. The orientation of each planes in the world can be seen clearly in the two dimensional Top View, which viewing the 3-D world at the X-Z plane along the Y-axis.

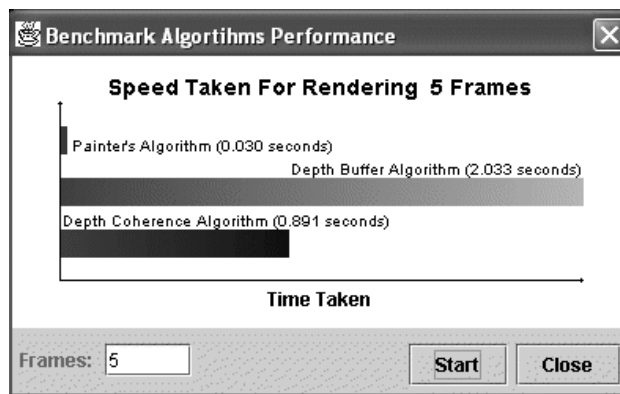
The User Interface

(done by Thomas Cheah.C.S. & Chia Wei, Tan)



- **Camera View** shows all the planes in a 3-D perspective view, rendered using the selected hidden surface algorithm.
- **Top View** provides a 2-D view of the planes position and orientation. User can drag to move and resize the planes or change its orientation.

- **Add/Remove Planes** provides buttons to add a plane of the selected color, or remove the selected plane.
- **Real Time Rendering On/Off** allows you to choose whether to render the Camera View in real time, i.e. whenever you are moving a plane, it will be rendered automatically, thus give you real time realism.
- **Rendering Algorithms** contains a list of hidden surface removal algorithms to choose for rendering.
- **Algorithm Details** show you additional information related to the currently selected Rendering Algorithm. This allows you a better idea how each algorithm behaves under different circumstances. For example, the Painter's algorithm will show the list of planes sorted by the algorithm.
- **Compare Algorithm** let you benchmark the three rendering algorithms. It will show you a chart to compare the time taken for rendering a specified amount of frame using different hidden surface removal. An example of the result is as below.



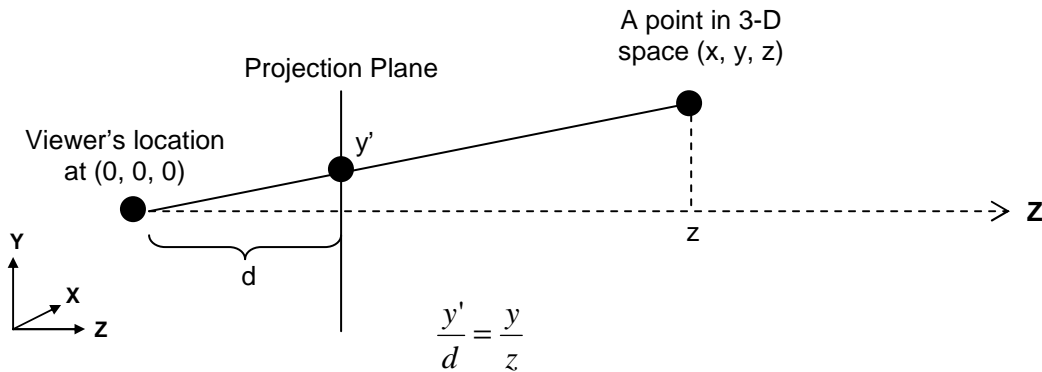
The 3-D Engine

(done by Thomas Cheah.C.S.)

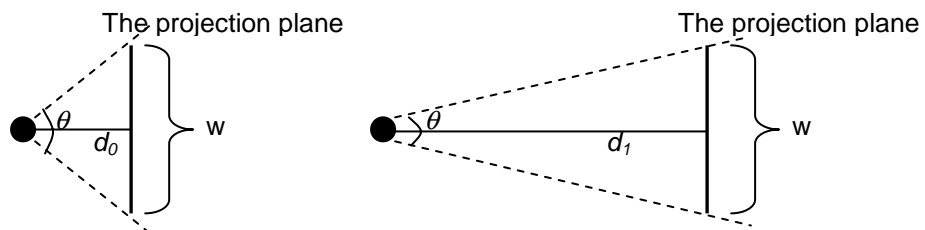
The 3-D engine implemented in this program uses a simple 3-D perspective-rendering algorithm, with rectangular planes as its primitive geometry. Any point in the 3-D space can be rendered into a two dimensional point on the viewing plane using the following equation,

$$x' = \frac{d \times x}{z}, \quad y' = \frac{d \times y}{z}$$

where (x, y, z) is a coordinate in 3-D space, and (x', y') is the projected point on the viewing plane. This equation can be found out simple triangle equation and a little knowledge of trigonometry.



What actually is d value? This can be imagined as the distance from the viewpoint to the projection plane. Different d value will produce a different projection on a same coordinate in 3-D space. The d value also determines the *Field of View* of the viewer. A

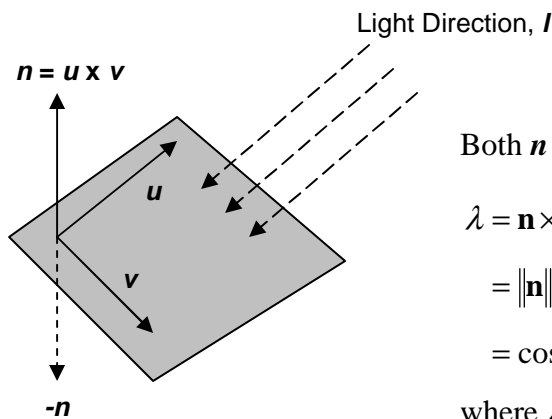


$$\tan\left(\frac{\theta}{2}\right) = \frac{w/2}{d}$$

w is the width or height of the projection plane, which is usually the width/height of the viewing area (window).

larger d value will produce a narrow field of view and small but positive d value will widen the field of view. This can be calculated by using some basic trigonometry as depicted in the illustration above.

The shading used in the 3-D engine is flat shading as it illustrates better, and does not need a lot of computational power. The lighting used in the 3-D engine is a *Directional Light*. This can be thought of a light coming from infinity, which has parallel rays. (Just like the sun shining the Earth.) Thus, the light intensity on each plane can be calculated by a simple vector math.



Both \mathbf{n} and \mathbf{l} is in its normalized form.

$$\begin{aligned} \lambda &= \mathbf{n} \times \mathbf{l} \\ &= \|\mathbf{n}\| \|\mathbf{l}\| \cos \theta \\ &= \cos \theta \end{aligned}$$

where λ will be the light intensity coefficient.

It will have value of 1 when θ is at 0° (direct incident of light), and 0 when θ is 90° (the light is parallel to the plane).

The Painters' Algorithm

(done by Thomas Cheah.C.S.)

This algorithm is by far the easiest algorithm. As the name implies, it just draw all the projected planes like how painters did on the canvas. Which draw from the object that is at the farthest, to the nearest, so that the nearer objects will overlapped the farther ones.

In our case, a plane has four vertices. To produce the optimum result, the average z-value of the four vertices is calculated. And we sort all the planes in the 3-D world from

1. Set all pixels to background color
2. Sort all planes in the descending order of the average z-value.
3. For each plane
 Begin
 (i) Calculate the projected points of the four vertices.
 (ii) Fill the plane with the color calculated by the shading algorithm.
 End

the highest average z-value (farthest plane), to the lowest.

Pros and Cons

- ✓ Fastest hidden surface removal algorithm.
- ✓ Require the least computational power.
- ✓ Its implementation is easy and straightforward.
- × Cannot show intersecting planes properly. (Perhaps this was the biggest disadvantage that makes this algorithm impractical.)

The Z – buffer Algorithm

(done by Patricia Malone)

The z-buffer algorithm is also known as the depth buffer. The z-buffer is the second simplest algorithm to understand and implement. This algorithm, unlike the Painters' algorithm, can handle intersecting polygons. The z-buffer is usually implemented in hardware because it is expensive to compute but it can still be done in software. In fact, most hardware graphics systems implement this algorithm.

The implementation of this algorithm involves using two matrices or 2-dimensional arrays. The first array is called the depth array (D[x, y]), this array contains the depth of every pixel in the image. Initially, this array is set to "infinity", which can be implemented by the use of -1; since no object in the viewable image can be behind the point of view. The second matrix is the intensity matrix (I[x, y]). The intensity matrix holds the current intensity or color value for each of the pixels in the image plane. Initially this array is set to the background color (in our case black). The initializing process is done in a for loop at the start of the rendering process.

After all the initializing has been completed, we start the actual implementation of the z-buffer algorithm. We take each polygon that is to appear in the image one at a time. For this algorithm, the order in which we take the polygons do not matter. I take the four vertices of the rectangle and convert them to projected coordinates using the camera view. I then create a polygon containing those four vertices. Next, I test to see if the pixels of the camera plane are contained in the polygon. If they are not then I do nothing, but if they are I calculate the depth that pixel would be in world coordinates by determining the slope of the polygon by:

$$\text{slope} = \frac{p0.z - p1.z}{(a.x - b.x)};$$

where p0 and p1 are the world coordinates of the top of the rectangle and, a and b are the projected coordinates respectively. After I have the slope I calculate the actual value of z by

$$z = (x-b.x)*\text{slope} + p1.z;$$

where p_1 is the world coordinate and b is the projected coordinate of the same vertex. x is the x value of the current pixel.

Next, I test to see if the calculated z is $z < D[x,y]$ and if it is I replace

$D[x, y] = z$ and $I[x, y] = \text{color of that rectangle.}$

I repeat this process for every pixel and every rectangle.

Pros and Cons

- ✓ It shows intersecting polygons despite the ordering in which the polygons are rendered.
- ✓ It provides accurate depiction of the polygons since it is done pixel by pixel.
- ✗ It is slow to render the new view (since the z -buffer is implemented pixel by pixel). Fortunately, however the z - buffer is implemented in most graphic hardware systems.

The Z-buffer Courseware

To make the implementation of the z -buffer into a teaching tool, I stored all the intermediate results of the two matrices. I saved the initial matrices and a version of it after every rectangle. This allowed me show the two matrices at different stages of the rendering the z -buffer algorithm. The user selects a region on the camera view by the press and drag of the mouse (to draw a diagonal line) then click a button, and up pops two windows; one displaying the depth matrix and the other displaying the intensity matrix for the selected region. The user can then select between the different stages by the click of a button. This will allow easy demonstration of how the two matrices are initialized, and how they alter after each rectangle is rendered. To allow comparison of the depth and intensity matrix, I number the rows and columns of each window sampling.

(B Intensity Matrix)

	(193)	(194)	(195)
(138)	[r=0,g=0,b=0]	[r=0,g=0,b=0]	[r=0,g=0,b=0]
(139)	[r=0,g=0,b=0]	[r=0,g=0,b=0]	[r=0,g=0,b=0]
(140)	[r=0,g=0,b=0]	[r=0,g=0,b=0]	[r=0,g=0,b=0]
(141)	[r=0,g=0,b=0]	[r=0,g=0,b=0]	[r=0,g=0,b=0]
(142)	[r=0,g=0,b=0]	[r=0,g=0,b=0]	[r=0,g=0,b=0]
(143)	[r=0,g=0,b=0]	[r=0,g=0,b=0]	[r=0,g=0,b=0]
(144)	[r=0,g=0,b=0]	[r=0,g=0,b=0]	[r=0,g=0,b=0]
(145)	[r=0,g=0,b=0]	[r=0,g=0,b=0]	[r=0,g=0,b=0]
(146)	[r=0,g=0,b=0]	[r=0,g=0,b=0]	[r=0,g=0,b=0]

(B Depth Matrix)

	(193)	(194)	(195)	(196)	(197)	(198)
(138)	132.03593	131.73653	131.43713	131.13773	130.83832	130.53892
(139)	132.03593	131.73653	131.43713	131.13773	130.83832	130.53892
(140)	132.03593	131.73653	131.43713	131.13773	130.83832	130.53892
(141)	132.03593	131.73653	131.43713	131.13773	130.83832	130.53892
(142)	132.03593	131.73653	131.43713	131.13773	130.83832	130.53892
(143)	132.03593	131.73653	131.43713	131.13773	130.83832	130.53892
(144)	132.03593	131.73653	131.43713	131.13773	130.83832	130.53892
(145)	132.03593	131.73653	131.43713	131.13773	130.83832	130.53892
(146)	132.03593	131.73653	131.43713	131.13773	130.83832	130.53892

Depth Coherence Algorithm

(done by Chia Wei, Tan)

Depth Coherence Algorithm, also called Warnock Area-subdivision Algorithm remove the hidden surfaces by repeatedly subdividing the view panel into rectangular areas until the areas can be trivially painted. That is, we can fill the area with only one color, either the color of one of the polygons or the background color. The repetitive subdivision indicates that this is a divide and conquer algorithm, so it can be implemented using recursive method.

Recursive function consists of base body and recursive body. For depth coherence algorithm, there are 3 base cases. The view panel is subdivided into rectangular areas until:

- Area is disjoint with all polygons: fill area with background color
- Area is surrounded by the closest polygon: fill area with the polygon's color
- Area is in pixel size: choose the closest polygon's color

So, the basic algorithm is:

```
DepthCoherence (Area)
    If area disjoint with all polygons
        Fill area with background color and return
    Else if area is surrounded by the closest polygon
        Fill are with the polygon's color and return
    Else if area is in pixel size
        Paint pixel with the closest polygon's color
    Else
        Divide area equally into A1, A2, A3 and A4
        DepthCoherence (A1)
        DepthCoherence (A2)
        DepthCoherence (A3)
        DepthCoherence (A4)
    End
```

To determine whether the area is either disjoint or surrounded by a polygon, we have to consider 3 scenarios:

1. The polygon is surrounding the area

2. The polygon is intersecting or contained in the area
3. The polygon is disjoint with the area

For 4, disjoint = [not surrounding and not (intersecting or contained)]. Whether a polygon is intersecting an area can be determined simply by checking whether any of the polygon's edge goes through the area. This can be done by using Cohen Sutherland Window-Clipping Algorithm.

To determine which polygon is the closest to the area, we need to determine first its distance to the area. This is achieved by interpolating the depth-value of the polygon between its top vertices (since the polygons in this project are constrained to rectangular planes) to the area's vertices. Then we can compare the depth values to determine the closest polygon in the area.

Pros and Cons

Note that in order to implement this algorithm, all the polygons' vertices have to be projected into the view panel first before any calculation. Since the projected polygons will be subdivided into pixel level if necessary, depth coherence algorithm has image precision. At the same time, all the projected vertices are associated with a depth value, so in terms of depth comparison, the algorithm has object precision too. In addition, the characteristic (projection of polygons into view panel) of the algorithm enables intersecting polygons to be drawn correctly.

The drawback of this algorithm is that for complex scene with many polygons. A lot of redundant subdivisions and calculations are done in most cases, the area will be subdivided recursively into pixel size. This makes the algorithm worse than depth-buffer algorithm due to its extra subdivisions before coming to pixel size. Beside that, due to its recursive nature, for every function call, a data section is accumulated. So, this algorithm requires large memory consumption for complex scenes.

Project Development

<i>Date</i>	<i>Development</i>
26 October 2001	<ul style="list-style-type: none">• Came out the idea of doing the courseware for hidden surface/line removal algorithms.
27 October 2001	<ul style="list-style-type: none">• Draft planning for the program and its user interface.
28 October 2001	<ul style="list-style-type: none">• Draft the proposal for this project.
29 October 2001	<ul style="list-style-type: none">• Submit the project proposal to Prof. Herb Yang.• Commence research on the project.• Search for books and papers for the algorithms.
5 November 2001	<ul style="list-style-type: none">• Start assimilating all the information found related to the project.• Start coding on the project.
11 November 2001	<ul style="list-style-type: none">• Completed the core part of the 3-D engine and the specifications for implementing the rendering engine.• Completed the core user interface, i.e. the Camera View and Top View.
13 November 2001	<ul style="list-style-type: none">• Completed the <code>PaintersRenderingEngine</code>.• Implementing drag-N-drop interaction in Top View.
15 November 2001	<ul style="list-style-type: none">• Completed the drag-N-drop interaction in Top View.• Completed <code>WarnocksRenderingEngine</code> but with overflows of bugs.• Fixed 3-D perspective bugs in the 3-D engine.
16 November 2001	<ul style="list-style-type: none">• Commence serious debugging on the <code>WarnocksRenderingEngine</code>.

	<ul style="list-style-type: none"> Spent time determining what is involved in implementing the z-buffer algorithm, how it works, and how to use the <code>CameraView</code>.
18 November 2001	<ul style="list-style-type: none"> Start implementing the <code>zBufferRenderingEngine</code>.
22 November 2001	<ul style="list-style-type: none"> Got the <code>zBufferRenderingEngine</code> displaying intersecting planes properly. But there is some missing (un-rendered) pixels on the planes.
29 November 2001	<ul style="list-style-type: none"> Implemented the interpolation of each polygon to fill in missing (un-rendered) pixels.
30 November 2001	<ul style="list-style-type: none"> Cleansed most of the bugs in <code>WarnocksRenderingEngine</code>.
1 December 2001	<ul style="list-style-type: none"> Refining the user interface and putting together all the individual UI components. Start writing the project documentation.
2 December 2001	<ul style="list-style-type: none"> Start of implementation of courseware component of the <code>zBufferRenderingEngine</code>.
3 December 2001	<ul style="list-style-type: none"> Implementing some of the courseware features for <code>zBufferRenderingEngine</code>.
4 December 2001	<ul style="list-style-type: none"> Completely fixed the missing (un-rendered) pixels in <code>zBufferRenderingEngine</code>, cleaning up code, and courseware. Perform usability testing on program. (And fix any bugs found.) Wrapping up the project documentation.

References

Donald Hearn, M. Pauline Baker. Computer Graphics.

Steven Harrington. Computer Graphics, A Programming Approach.

James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes. Computer Graphics : Principles and Practice, Second Edition in C