Hidden Surface Removal

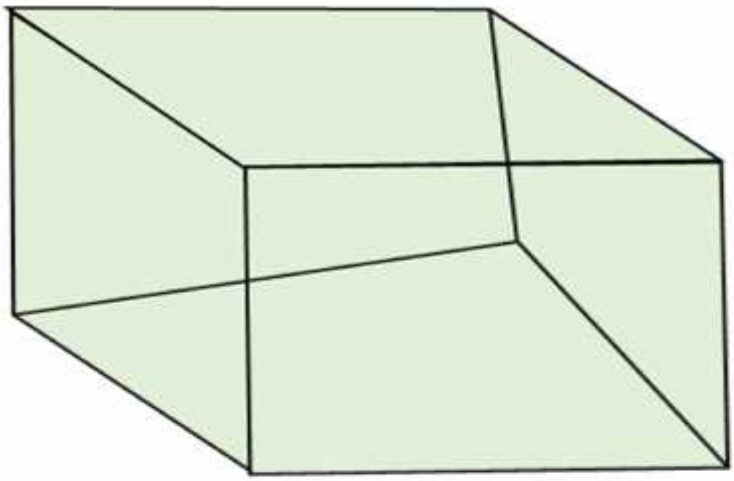
In 3D computer graphics, hidden-surface determination (also known as shown-surface determination, hidden-surface removal (HSR), occlusion culling (OC) or visible-surface determination (VSD)) is the process of identifying what surfaces and parts of surfaces can be seen from a particular viewing angle. A hidden-surface determination algorithm is a solution to the visibility problem, which was one of the first major problems in the field of 3D computer graphics. The process of hidden-surface determination is sometimes called hiding, and such an algorithm is sometimes called a hider[When referring to line rendering it is known as hidden-line removal. Hidden-surface determination is necessary to render a scene correctly, so that one may not view features hidden behind the model itself, allowing only the naturally viewable portion of the graphic to be visible.

Hidden-surface determination is a process by which surfaces that should not be visible to the user (for example, because they lie behind opaque objects such as walls) are prevented from being rendered. Despite advances in hardware capability, there is still a need for advanced rendering algorithms. The responsibility of a rendering engine is to allow for large world spaces, and as the world's size approaches infinity, the engine should not slow down but remain at a constant speed. Optimizing this process relies on being able to ensure the deployment of as few resources as possible towards the rendering of surfaces that will not end up being displayed to the user.

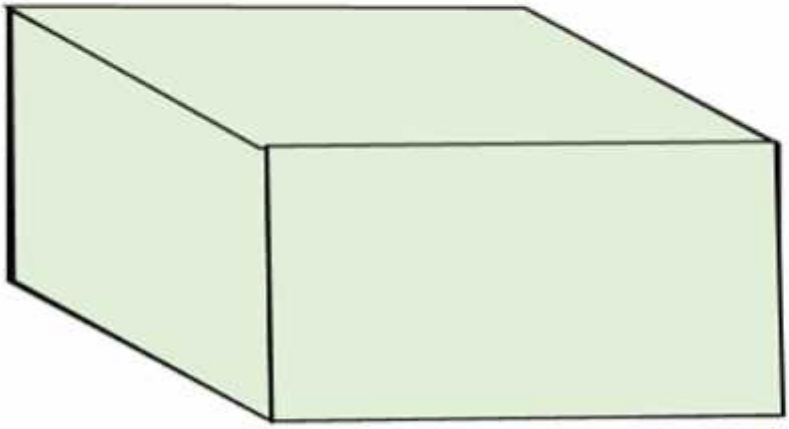
Types of hidden surface detection algorithms

1. Object space methods
2. Image space methods

Object space methods: In this method, various parts of objects are compared. After comparison visible, invisible or hardly visible surface is determined. These methods generally decide visible surface. In the wireframe model, these are used to determine a visible line. So these algorithms are line based instead of surface based. Method proceeds by determination of parts of an object whose view is obstructed by other object and draws these parts in the same color.



Object with hidden line



Object when hidden lines removed

Image space methods: Here positions of various pixels are determined. It is used to locate the visible surface instead of a visible line. Each point is detected for its visibility. If a point is visible, then the pixel is on, otherwise off. So the object close to the viewer that is pierced by a projector through a pixel is determined. That pixel is drawn is appropriate color.

These methods are also called a **Visible Surface Determination**. The implementation of these methods on a computer requires a lot of processing time and processing power of the computer.

The image space method requires more computations. Each object is defined clearly. Visibility of each object surface is also determined.

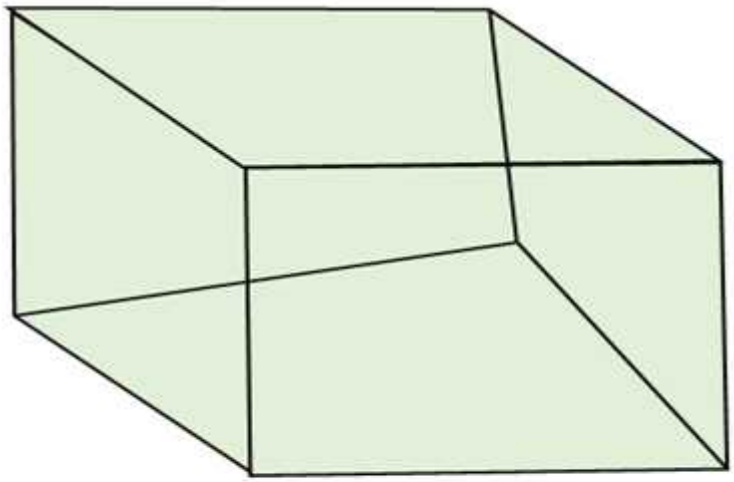
Differentiate between Object space and Image space method

Object Space	Image Space
1. Image space is object based. It concentrates on geometrical relation among objects in the scene.	1. It is a pixel-based method. It is conc the final image, what is visible within e pixel.
2. Here surface visibility is determined.	2. Here line visibility or point v determined.
3. It is performed at the precision with which each object is defined, No resolution is considered.	3. It is performed using the resoluti display device.
4. Calculations are not based on the resolution of the display so change of object can be easily adjusted.	4. Calculations are resolution base, so t is difficult to adjust.
5. These were developed for vector graphics system.	5. These are developed for raster devic
6. Object-based algorithms operate on continuous object data.	6. These operate on object data.
7. Vector display used for object method has large address space.	7. Raster systems used for image spac have limited address space.

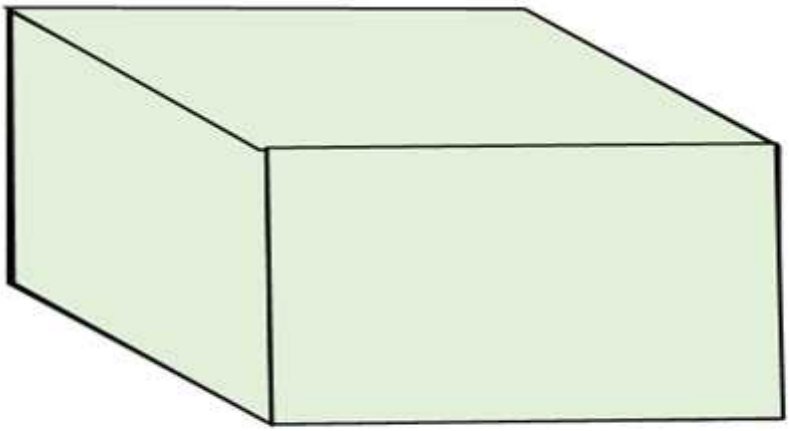
8. Object precision is used for application where speed is required.	8. There are suitable for applicati accuracy is required.
9. It requires a lot of calculations if the image is to enlarge.	9. Image can be enlarged witho accuracy.
10. If the number of objects in the scene increases, computation time also increases.	10. In this method complexity increas complexity of visible parts.

Similarity of object and Image space method

In both method sorting is used a depth comparison of individual lines, surfaces are objected to their distances from the view plane.



Object with hidden line



Object when hidden lines removed

Considerations for selecting or designing hidden surface algorithms: Following three considerations are taken:

1. Sorting
2. Coherence
3. Machine

Sorting: All surfaces are sorted in two classes, i.e., visible and invisible. Pixels are colored accordingly. Several sorting algorithms are available i.e.

1. Bubble sort
2. Shell sort
3. Quick sort
4. Tree sort
5. Radix sort

Different sorting algorithms are applied to different hidden surface algorithms. Sorting of objects is done using x and y, z co-ordinates. Mostly z coordinate is used for sorting. The efficiency of sorting algorithm affects the hidden surface removal algorithm. For sorting complex scenes or hundreds of polygons complex sorts are used, i.e., quick sort, tree sort, radix sort.

For simple objects selection, insertion, bubble sort is used.

Coherence

It is used to take advantage of the constant value of the surface of the scene. It is based on how much regularity exists in the scene. When we moved from one polygon of one object to another polygon of same object color and shearing will remain unchanged.

Types of Coherence

1. Edge coherence
2. Object coherence
3. Face coherence
4. Area coherence
5. Depth coherence
6. Scan line coherence
7. Frame coherence

8. Implied edge coherence

1. Edge coherence: The visibility of edge changes when it crosses another edge or it also penetrates a visible edge.

2. Object coherence: Each object is considered separate from others. In object, coherence comparison is done using an object instead of edge or vertex. If A object is farther from object B, then there is no need to compare edges and faces.

3. Face coherence: In this faces or polygons which are generally small compared with the size of the image.

4. Area coherence: It is used to group of pixels cover by same visible face.

5. Depth coherence: Location of various polygons has separated a basis of depth. Depth of surface at one point is calculated, the depth of points on rest of the surface can often be determined by a simple difference equation.

6. Scan line coherence: The object is scanned using one scan line then using the second scan line. The intercept of the first line.

7. Frame coherence: It is used for animated objects. It is used when there is little change in image from one frame to another.

8. Implied edge coherence: If a face penetrates in another, line of intersection can be determined from two points of intersection.

Algorithms used for hidden line surface detection

1. Back Face Removal Algorithm
2. Z-Buffer Algorithm
3. Painter Algorithm

4. Scan Line Algorithm
5. Subdivision Algorithm
6. Floating horizon Algorithm

There are many methods for hidden surface identification. They are basically a work out in sorting, and generally vary in the order in which the sort is executed and how the problem is subdivided. Sorting more values of graphics primitives is generally done by divide.

1. Z - Buffer Algorithm

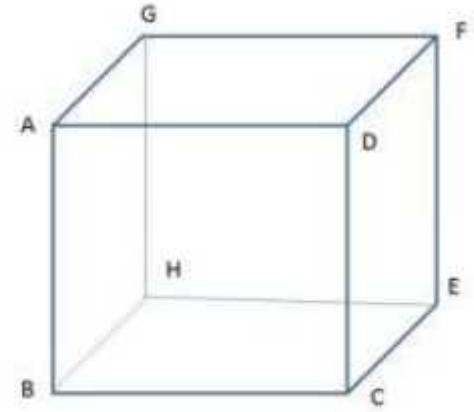


Figure 1.: Priority algorithm

In Z-buffering, the depth of 'Z'value is verified against available depth value. If the present pixel is behind the pixel in the Z-buffer, the pixel is eliminated, or else it is shaded and its depth value changes the one in the Z-buffer. Z-buffering helps dynamic visuals easily, and is presently introduced effectively in graphics hardware.

- Depth buffering is one of the easiest hidden surface algorithms
- It keeps follow of the space to nearest object at every pixel position.
- Initialized to most negative z value.
- When image being drawn, if its z coordinate at a position is higher than z buffer value, it is drawn, and new z coordinate value is stored; or else, it is not drawn
- If a line in three dimensional is being drawn, then the middle z values are interpolated: linear interpolation for polygons, and can calculate z for more difficult surfaces.

Algorithm

```
loop on y;

loop on x;

zbuf[x,y] = infinity;

loop on objects

{

    loop on y within y range of this object

  {

    loop on x within x range of this scan line of this object

    {

      if z(x,y) < zbuf[x,y] compute z of this object at this pixel & test

      zbuf[x,y] = z(x,y) update zbuffer

      image[x,y] = shade(x,y) update image (typically RGB)

    }

  }

}
```

Basic operations

1. Compute y range of an object
2. Compute x range of a given scan line of an object
3. Calculate intersection point of a object with ray through pixel position (x,y)

OR

Z-Buffer Algorithm

It is also called a **Depth Buffer Algorithm**. Depth buffer algorithm is simplest image space algorithm. For each pixel on the display screen, we keep a record of the depth of an object within the pixel that lies closest to the observer. In addition to depth, we also record the intensity that should be displayed to show the object. Depth buffer is an extension of the frame buffer. Depth buffer algorithm requires 2 arrays, intensity and depth each of which is indexed by pixel coordinates (x, y).

Algorithm

For all pixels on the screen, set depth [x, y] to 1.0 and intensity [x, y] to a background value.

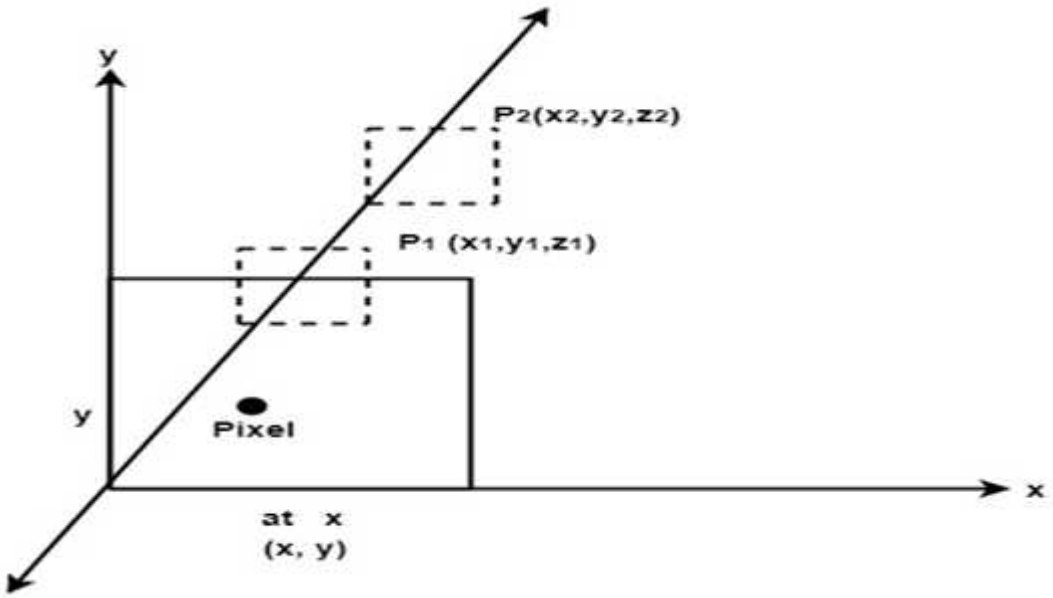
For each polygon in the scene, find all pixels (x, y) that lie within the boundaries of a polygon when projected onto the screen. For each of these pixels:

(a) Calculate the depth z of the polygon at (x, y)

(b) If z < depth [x, y], this polygon is closer to the observer than others already recorded for this pixel. In this case, set depth [x, y] to z and intensity [x, y] to a value corresponding to polygon's shading. If instead z > depth [x, y], the polygon already recorded at (x, y) lies closer to the observer than does this new polygon, and no action is taken.

3. After all, polygons have been processed; the intensity array will contain the solution.

4. The depth buffer algorithm illustrates several features common to all hidden surface algorithms.



5. First, it requires a representation of all opaque surface in scene polygon in this case.

6. These polygons may be faces of polyhedral recorded in the model of scene or may simply represent thin opaque 'sheets' in the scene.

7. The IInd important feature of the algorithm is its use of a screen coordinate system. Before step 1, all polygons in the scene are transformed into a screen coordinate system using matrix multiplication.

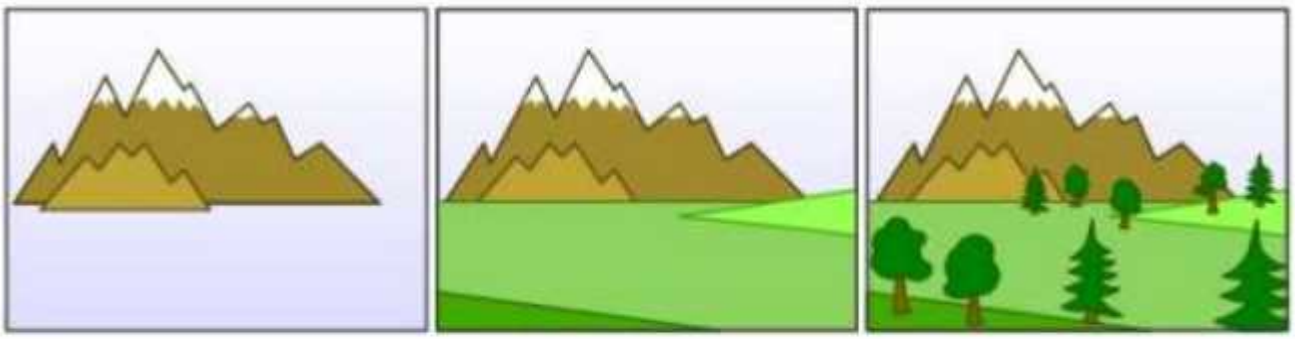
Limitations of Depth Buffer

1. The depth buffer Algorithm is not always practical because of the enormous size of depth and intensity arrays.
2. Generating an image with a raster of 500 x 500 pixels requires 2, 50,000 storage locations for each array.
3. Even though the frame buffer may provide memory for intensity array, the depth array remains large.

4. To reduce the amount of storage required, the image can be divided into many smaller images, and the depth buffer algorithm is applied to each in turn.
5. For example, the original 500 x 500 faster can be divided into 100 rasters each 50 x 50 pixels.
6. Processing each small raster requires array of only 2500 elements, but execution time grows because each polygon is processed many times.
7. Subdivision of the screen does not always increase execution time instead it can help reduce the work required to generate the image. This reduction arises because of coherence between small regions of the screen.

2. Painter'salgorithm

The painter's algorithm is called as a priority fill, is one of the easiest results to the visibility issue in three dimensional graphics. When projecting a 3D view onto a 2D screen, it is essential at various points to be finalized which polygons are visible, and which polygons are hidden.



Painter Algorithms

The 'painter's algorithm'shows to the method employed by most of the painters of painting remote parts of a scene before parts which are close thereby hiding some areas of distant parts. The painter's algorithm arranges all the polygons in a view by their depth and then paints them in this order, extreme to closest. It will paint over the existing parts that are usually not visible hence solving the visibility issue at the cost of having painted invisible areas of distant objects. The ordering used by the algorithm is referred a

’depth order’, and does not have to respect the distances to the parts of the scene: the important characteristics of this ordering is, somewhat, that if one object has ambiguous part of another then the first object is painted after the object that it is ambiguous. Thus, a suitable ordering can be explained as a topological ordering of a directed acyclic graph showing between objects.

Algorithm

```
sort objects by depth, splitting if necessary to handle intersections; loop on
objects (drawing

from back to front)

{

  loop on y within y range of this object

  {

      loop on x within x range of this scan line of this object

    {

        image[x,y] = shade(x,y);

  }

    }

}
```

Basic operations

1. Compute ‘y’range of an object
2. Compute ‘x’range of a given scan line of an object
3. Compute intersection point of a given object with ray via pixel point (x,y).
4. Evaluate depth of two objects, determine if a is in front of b, or b is in front of a, if they don’t overlap in xy, or if they intersect
5. Divide one object by another object

Advantage of painter’s algorithm is the inner loops are quite easy and limitation is sorting operation.

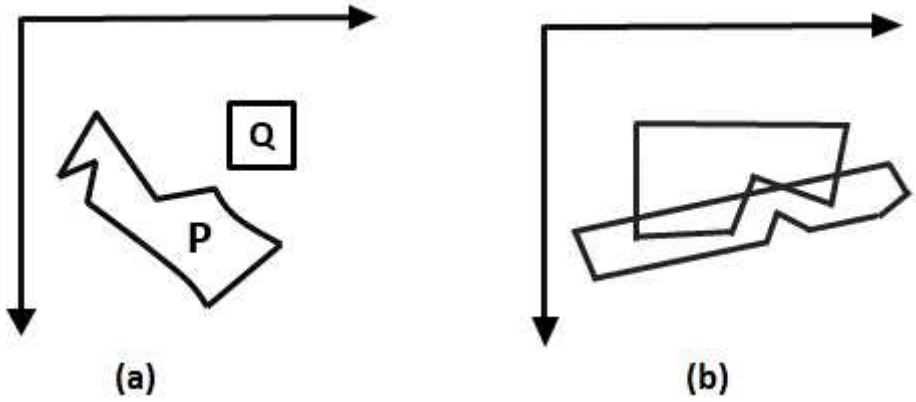
OR

Painter Algorithm

It came under the category of list priority algorithm. It is also called a **depth-sort algorithm**. In this algorithm ordering of visibility of an object is done. If objects are reversed in a particular order, then correct picture results.

Objects are arranged in increasing order to z coordinate. Rendering is done in order of z coordinate. Further objects will obscure near one. Pixels of rear one will overwrite pixels of farther objects. If z values of two overlap, we can determine the correct order from Z value as shown in fig (a).

If z objects overlap each other as in fig (b) this correct order can be maintained by splitting of objects.



Depth sort algorithm or painter algorithm was developed by Newell, sancha. It is called the painter algorithm because the painting of frame buffer is done in decreasing order of distance. The distance is from view plane. The polygons at more distance are painted firstly.

The concept has taken color from a painter or artist. When the painter makes a painting, first of all, he will paint the entire canvas with the background color. Then more distance objects like mountains, trees are added. Then rear or foreground objects are added to picture. Similar approach we will use. We will sort surfaces according to z values. The z values are stored in the refresh buffer.

Steps performed in-depth sort

1. Sort all polygons according to z coordinate.
2. Find ambiguities of any, find whether z coordinate overlap, split polygon if necessary.
3. Scan convert each polygon in increasing order of z coordinate.

Painter Algorithm

Step1: Start Algorithm

Step2: Sort all polygons by z value keep the largest value of z first.

Step3: Scan converts polygons in this order. Test is applied

1. Does A is behind and non-overlapping B in the dimension of Z as shown in fig (a)
2. Does A is behind B in z and no overlapping in x or y as shown in fig (b)
3. If A is behind B in Z and totally outside B with respect to view plane as shown in fig (c)
4. If A is behind B in Z and B is totally inside A with respect to view plane as shown in fig (d)

The success of any test with single overlapping polygon allows F to be painted.

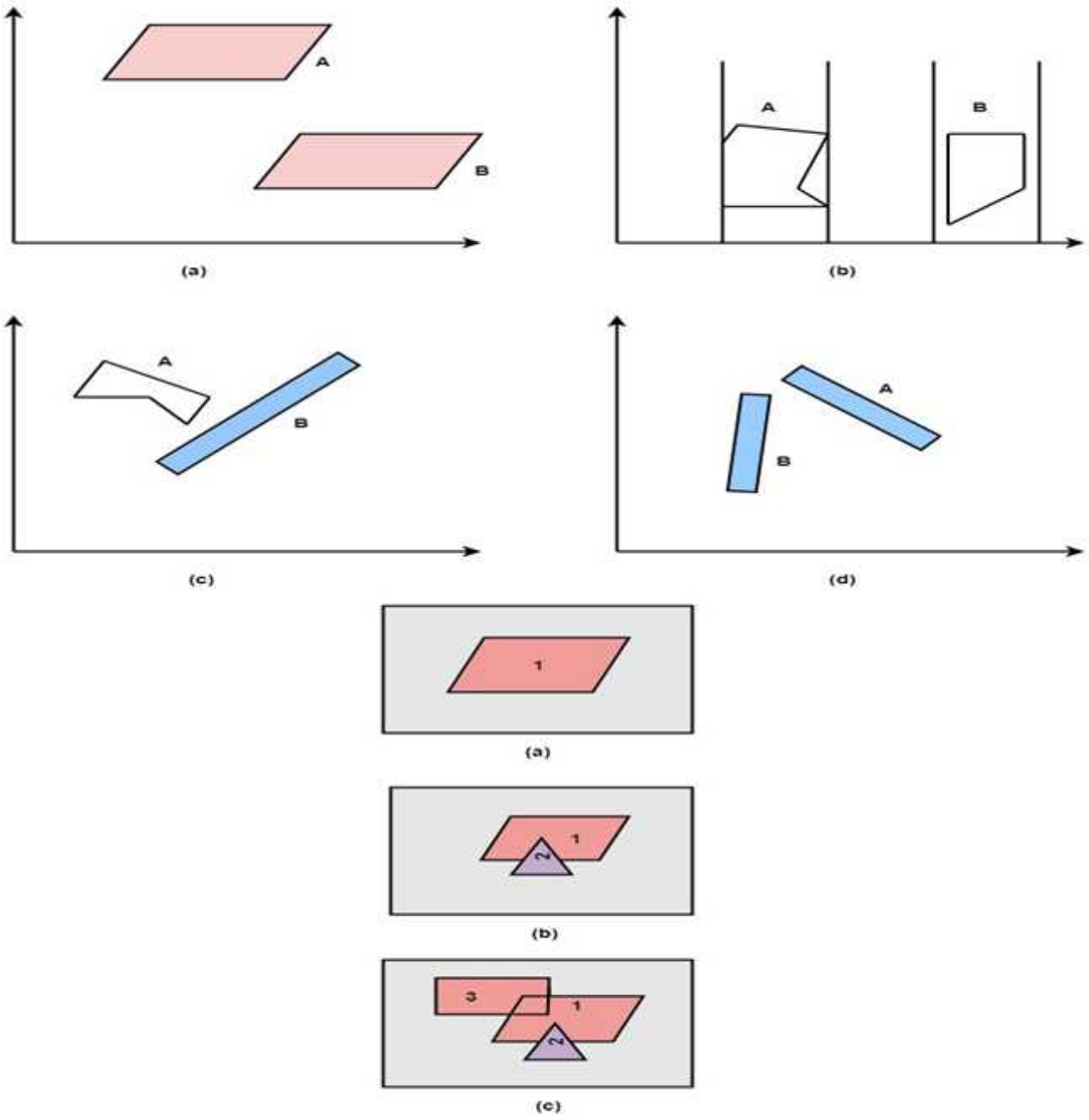


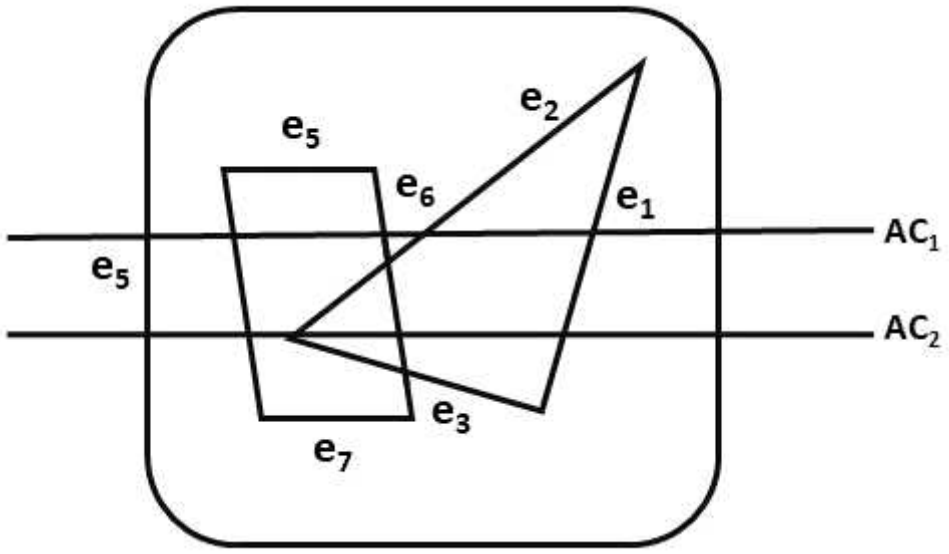
Figure showing addition of surface one by one and then painting done using painter algorithm

Scan Line Algorithm

It is an image space algorithm. It processes one line at a time rather than one pixel at a time. It uses the concept area of coherence. This algorithm records edge list, active edge list. So accurate bookkeeping is necessary. The edge list

or edge table contains the coordinate of two endpoints. Active Edge List (AEL) contain edges a given scan line intersects during its sweep. The active edge list (AEL) should be sorted in increasing order of x. The AEL is dynamic, growing and shrinking.

Following figures shown edges and active edge list. The active edge list for scan line AC_1contain e_1,e_2,e_5,e_6 edges. The active edge list for scan line AC_2contain e_5,e_6,e_1.



Scan line can deal with multiple surfaces. As each scan line is processed, this line will intersect many surfaces. The intersecting line will determine which surface is visible. Depth calculation for each surface is done. The surface rear to view plane is defined. When the visibility of a surface is determined, then intensity value is entered into refresh buffer.

Algorithm

Step1: Start algorithm

Step2: Initialize the desired data structure

1. Create a polygon table having color, edge pointers, coefficients
2. Establish edge table contains information regarding, the endpoint of edges, pointer to polygon, inverse slope.
3. Create Active edge list. This will be sorted in increasing order of x.
4. Create a flag F. It will have two values either on or off.

Step3: Perform the following steps for all scan lines

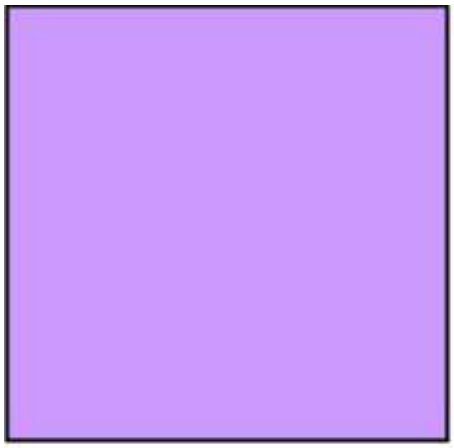
1. Enter values in Active edge list (AEL) in sorted order using y as value
2. Scan until the flag, i.e. F is on using a background color
3. When one polygon flag is on, and this is for surface S_1enter color intensity as I_1into refresh buffer
4. When two or image surface flag are on, sort the surfaces according to depth and use intensity value S_n for the nth surface. This surface will have least z depth value
5. Use the concept of coherence for remaining planes.

Step4: Stop Algorithm

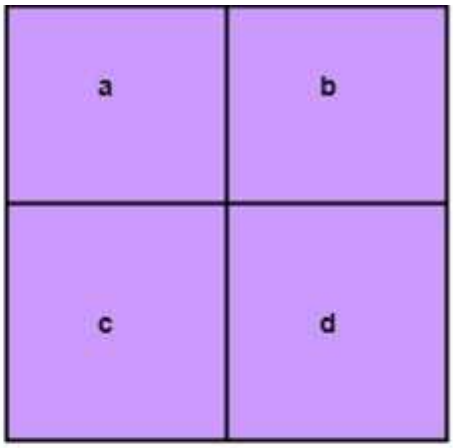
Area Subdivision Algorithm

It was invented by John Warnock and also called a Warnock Algorithm. It is based on a divide & conquer method. It uses fundamental of area coherence. It is used to resolve the visibility of algorithms. It classifies polygons in two cases i.e. trivial and non-trivial.

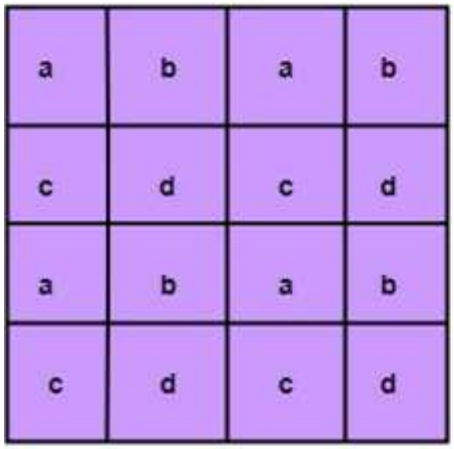
Trivial cases are easily handled. Non trivial cases are divided into four equal subwindows. The windows are again further subdivided using recursion until all polygons classified trivial and non trivial.



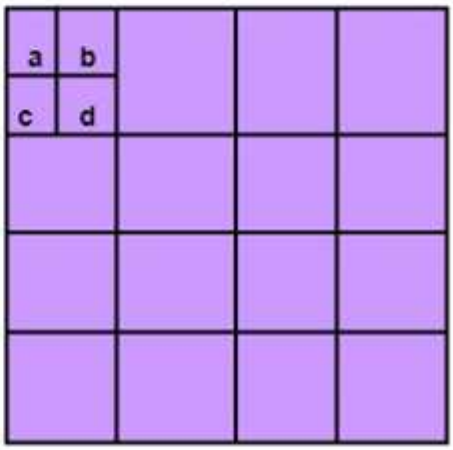
Original area
(a)



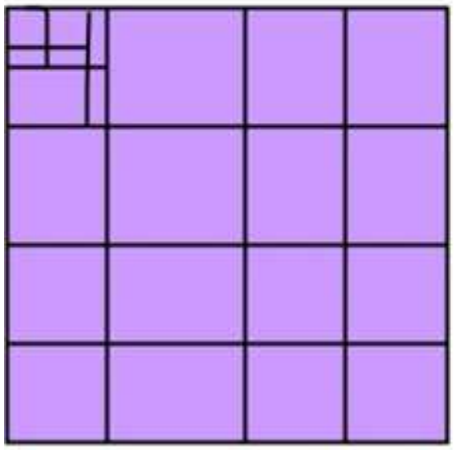
First division or subdivision of area
(b)



Second division
(c)



Third subdivision
(d)



Fourth subdivision
(e)

Classification of Scheme

It divides or classifies polygons in four categories:

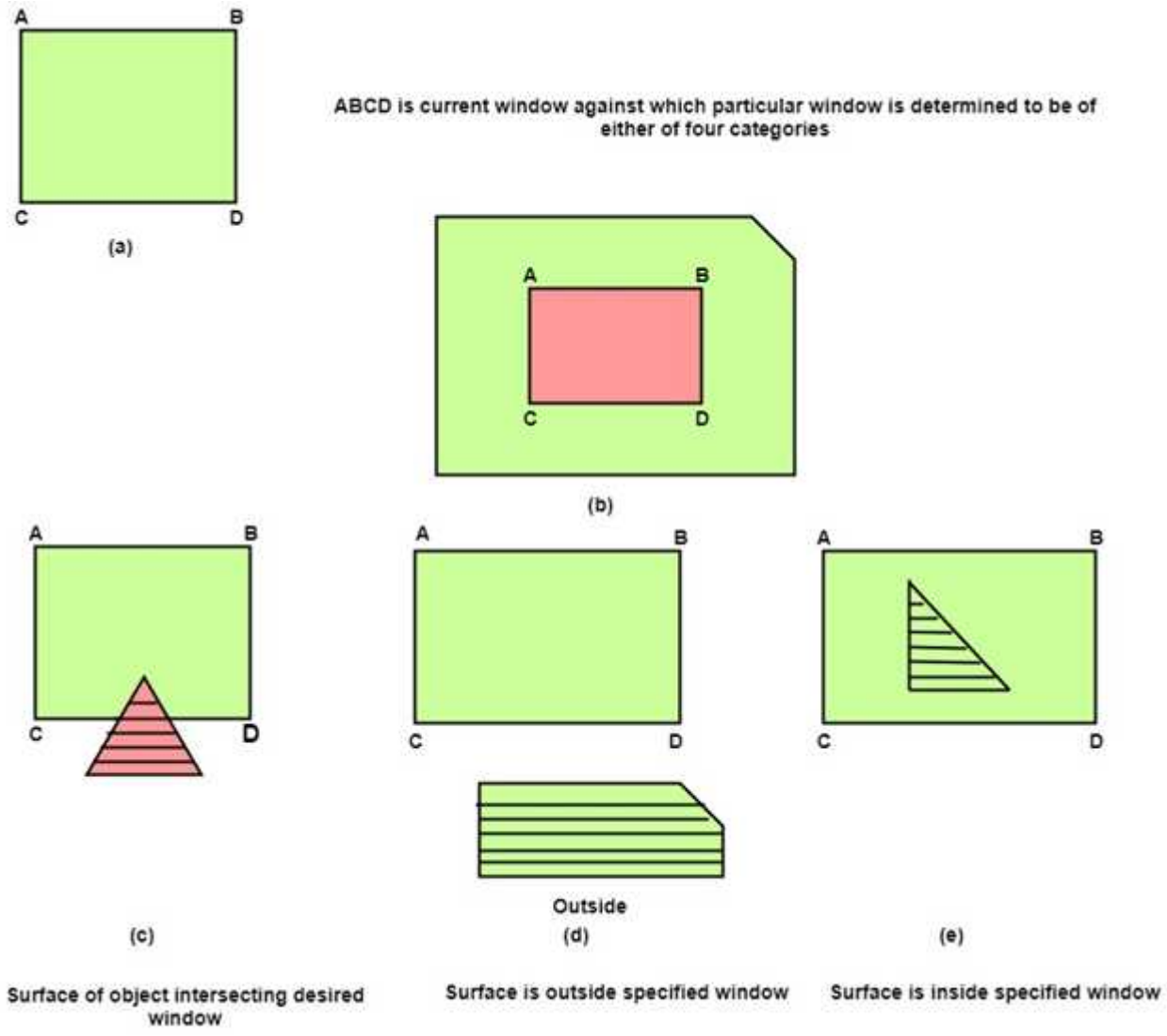
1. Inside surface
2. Outside surface
3. Overlapping surface
4. Surrounding surface

1. Inside surface: It is surface which is completely inside the surrounding window or specified boundary as shown in fig (c)

2. Outside surface: The polygon surface completely outside the surrounding window as shown in fig (a)

3. Overlapping surface: It is polygon surface which completely encloses the surrounding window as shown in fig (b)

4. Overlapping surface: It is surface partially inside or partially outside the surface area as shown in fig (c)



12.2 - Hidden Surface Removal

Hidden surface removal determines which triangles, lines, and points of a scene are visible from a virtual camera. This is a difficult problem to solve efficiently, especially if geometric primitives intersect each other. Many algorithms have been developed to solve this problem. Let's discuss just two of them.

The Painter's Algorithm

A human artist creates a painting by applying the background paint first and then painting layer upon layer until the foreground objects are painted last. This can be simulated in a computer by sorting the models in a scene according to their distance from the camera and then rendering them from back to front. It is a simple algorithm, but it has the following drawbacks:

- Sorting is time consuming. If the camera or the models are moving, sorting is required before every render.
- The individual triangles that compose a model must also be sorted based on their relationship to the camera. Fast rendering is dependent on a model's data being stored in a GPU's memory and never being modified. Sorting a model's triangles breaks this scheme.
- If triangles intersect, they can't be sorted such that one of them is closer to the camera than the other one. To render them accurately, their intersection must be found, or the triangles must be split into smaller triangles that don't intersect and then sorted.
  This is called the painter's algorithm and it is rarely used because of its drawbacks. However, it is used to render transparent models, which we will discuss in lesson 12.4.

The Z-Buffer Algorithm

A "z-buffer" is a 2D array of values equivalent in size to the *color buffer* that stores a rendered image. Each value in a *z-buffer* represents the distance between an object rendered at that pixel and the camera. Remember that the camera is always at the origin looking down the -Z axis. Therefore the Z value of an element represents its distance from the camera.

To render a scene, each element in a *z-buffer* is set to some "maximum value". As each pixel that composes a graphics primitive is rendered, the z-component of its geometry is compared to the current value in the *z-buffer*. If the z-component is less than the value in the *z-buffer*, this object is closer to the camera, so its color is placed in the *color buffer* and the *z-buffer*'s value is update. If an object's z value is greater than the current *z-buffer* value, the object is not visible to the camera because there is a closer object in front of it. This algorithm is explained nicely in the following pseudocode.
The *clearBuffers* function is called once to initialize a rendering. The *renderPixel* function is called for each pixel of every primitive that is rendered. (Note: These pseudocode functions are "hardcoded" into the graphics pipeline hardware; you don't implement them.)

```
void clearBuffers() {

  for (x = 0; x < image_width; x++) {

    for (y = 0; y < image_height; y++) {

      z_buffer[x][y]     = maximum_z_value;  // depth buffer

      color_buffer[x][y] = background_color;

    }

  }
```

```
}


void renderPixel(x, y, z, color) {
  if (z < z_buffer[x][y]) {
    z_buffer[x][y]     = z;                  // depth buffer
    color_buffer[x][y] = color;
  }
}
```

The [z-buffer algorithm] is the most widely used method for solving the hidden surface problem. It has the following major advantages over other *hidden surface removal* algorithms:

- No sorting is required. Models can be rendered in any order.
- No geometric intersection calculations are required. The algorithm produces the correct output even for intersecting or overlapping triangles.
- The algorithm is very simple to implement.

Disadvantages of the z-buffer algorithm include:

- A *z-buffer* requires a non-trivial amount of memory. For example, assuming each value in a *z-buffer* is a 32 bit floating point value, a rendered image that is 1024x768 pixels requires 3MB of memory to store its *z-buffer*.
- Every pixel of every primitive element must be rendered, even if many of them never write their color to the *color buffer*.
- If two primitives are in exactly the same place in 3D space, as their positions are interpolated across their respective surfaces, the z values for each object will typically be different by a very small amount due to floating-point round-off errors. These small differences will alternate between primitives for adjacent pixels resulting in random and weird patterns in a rendering. This is called “z-fighting” and it can be avoided by never placing two primitives in the same location in 3D space.

WebGL Implementation of the Z-buffer Algorithm

The WebGL graphics pipeline does not automatically perform *hidden surface removal*. You must enable it with this command:

```
gl.enable(gl.DEPTH_TEST);
```

Since WebGL is a “state machine”, you only need to execute this command once, unless you want to turn *hidden surface removal* on and off for special types of rendering. To disable *hidden surface removal*:

```
gl.disable(gl.DEPTH_TEST);
```

There are three buffers that typically need clearing before a rendering begins. These are identified using *enumerated type* constants defined inside the WebGL API. (Never use the numerical values; always use the constant names.) These values are “bit flags”. Notice that each value has a single bit set. You can combine “bit flags” into a single value using a *bit-wise or* operation, which in JavaScript is a single vertical bar, `|`. (Note that any value specified with a leading `0x` is a hexadecimal value (base 16).)

```
const GLenum DEPTH_BUFFER_BIT   = 0x00000100;

const GLenum STENCIL_BUFFER_BIT = 0x00000400;

const GLenum COLOR_BUFFER_BIT   = 0x00004000;
```

To clear the *color buffer* and the *depth buffer* (z-buffer) at the beginning of a rendering call `gl.clear(bit_flags)`. The input argument is a single integer containing “bit flags” that indicate which buffers to clear. You can clear one, two, or three buffers simultaneously. The command

```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

clears the *color buffer* and *depth buffers*. Every pixel in the *color buffer* is set to the background color (`gl.clearColor(red, green, blue, alpha)`). Every element in the *depth buffer* is set to the maximum depth value (which defaults to 1.0, but can be changed using `gl.clearDepth(depth)`).

WebGL Context Configuration

The default behaviour of a WebGL context is to automatically clear the “off-screen frame buffer” after it is copied to the “on-screen canvas window”. You can modify this behaviour by setting the `preserveDrawingBuffer` attribute of the WebGL context to `true`. This must be done when the context is initially created like this:

```
context = canvas.getContext('webgl', { preserveDrawingBuffer : true } );
```

Preserving the contents of the *draw buffers* between rendering cycles is not recommended.

WebGL Context Configuration

WebGL context configuration must be done when the context is initially created. (See this WebGL API page for a list of all the possible attributes of a WebGL context.)

Fine Grain Control of a Depth Buffer

WebGL provides tools for fine grain control of its z-buffer (*depth buffer*) for special rendering problems.

- `gl.depthMask(bool flag)` : Enables or disables writing to the *depth buffer*. When the *depth buffer* is disabled, this renders a model to the *color buffer* but does not update the depth of those pixels. This can be used for rendering transparent surfaces.
- `gl.clearDepth(float depth)`, where `depth` is a percentage value between 0.0 and 1.0. This sets the value used to clear the *depth buffer*. The "depth" is a percentage of the range of values that can be stored in the *depth buffer*. The default value is 1.0, which clears a *depth buffer* to its maximum value.
- `gl.depthFunc(enum func)`, where the parameter can be one of: `gl.NEVER`, `gl.ALWAYS`, `gl.LESS`, `gl.EQUAL`, `gl.LEQUAL`, `gl.GREATER`, `gl.GEQUAL`, `gl.NOTEQUAL`. This provides fine grain control over the test that determines whether a color is written to the *color buffer*. The default value is `gl.LESS`.
  Given the ability to set these extra values for the *z-buffer algorithm*, we can describe the algorithm in more detail using the following pseudocode. This is a description of the logic that is hard-coded into the graphics pipeline.

```
int     depth_test_func = LESS;  // DEFAULT

boolean write_depth     = true;  // DEFAULT

float   maximum_z_value = 1.0;   // DEFAULT


void gl.depthMask(bool flag) {

  write_depth = flag;

}


void gl.clearDepth(float depth) {

  maximum_z_value = depth;

}


void gl.depthFunc(enum func) {
```

```
  depth_test_func = func;
}

void gl.clear() {
  for (x = 0; x < image_width; x++) {
    for (y = 0; y < image_height; y++) {
      depth_buffer[x][y] = maximum_z_value;
      color_buffer[x][y] = background_color;
    }
  }
}

void renderPixel(x, y, z, color) {
  if (depth_test_is_enabled) {           // gl.enable(gl.DEPTH_TEST);
    if (passes_depth_test(x, y, z)) {
      if (write_depth) depth_buffer[x][y] = z;
      color_buffer[x][y] = color;
    }
  } else {                               // gl.disable(gl.DEPTH_TEST);
    color_buffer[x][y] = color;
  }
}

boolean passes_depth_test(x, y, z) {
  switch (depth_test_func) {             // gl.depthFunc(enum func);
    case NEVER:    condition = false;
```

```
    case ALWAYS:   condition = true;
    case LESS:     condition = (z <  depth_buffer[x][y]);  // DEFAULT
    case EQUAL:    condition = (z == depth_buffer[x][y]);
    case LEQUAL:   condition = (z <= depth_buffer[x][y]);
    case GREATER:  condition = (z >  depth_buffer[x][y]);
    case GEQUAL:   condition = (z >= depth_buffer[x][y]);
    case NOTEQUAL: condition = (z != depth_buffer[x][y]);
  }
  return condition;
}
```

WebGL Experimentation

Using the WebGL program below (which is a simple scaling example from a previous lesson), make the following suggested changes to see the effect of these z-buffer commands on a rendering.

- In line 123, change `gl.enable(gl.DEPTH_TEST);` to `gl.disable(gl.DEPTH_TEST);`. This turns off *hidden surface removal*. After re-starting the program, rotate the model to see different views. The result is basically the "painter's algorithm" without any sorting. Can you determine which cube is always drawn last?
- In the render function around line 71, add a call to clear the *color buffer*: `gl.clear(gl.COLOR_BUFFER_BIT);`. This changes the background to white because of the `gl.clearColor(0.98, 0.98, 0.98, 1.0);` command in the constructor.
- In the render function, before calling `gl.clear()`, set `clearColor` to a random color. That is: `gl.clearColor(Math.random(), Math.random(), Math.random(), 1.0);`
- To investigate round-off errors in the z-buffer algorithm, render two versions of the model that take up the same 3D locations.
    - In the constructor, create a matrix for translation: `let translate = matrix.create();`.
    - In the render function, render the model again, but with a translation:

```
o  matrix.translate(translate, 0.5, 0.0, 0.0);
o  matrix.multiplySeries(transform, transform, translate);
o  for (let j = 0; j < scene_models.length; j += 1) {
```

```
o    scene_models[j].render(transform);
o  }
```

The flickering in color is called “z-fighting” and is due to round-off errors in the z values. There are two triangles at the same z-depth and it can’t resolve which one should be drawn.

- Enable the depth buffer, set the “clear depth” value to 0.0 (it’s minimum value), clear both the *color buffer* and the *depth buffer*, and change the “depth test” to `gl.GREATER`. That is:

```
• gl.enable(gl.DEPTH_TEST);
• gl.clearDepth(0.0);
• gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
```

```
gl.depthFunc(gl.GREATER);
```