Chapter 9

Visible-Surface Detection Methods

9.1 Overview

We saw in the previous chapter how to create 3-D scenes of different kind of objects. Since parts of some objects can be occluded by other objects within the scene, the computer has to determine which object is in the front to preserve a three-dimensional impression by the viewer.

Therefore, this chapter will introduce some so-called visibility algorithms that do just that. In particular we will cover the following topics:

- Back-face culling
- Z-buffer algorithm
- A-buffer algorithm
- Ray casting

9.1 Overview

The goal of visibility algorithms is to determining as exactly as possible which parts of the scene that is to be rendered are visible and which are invisible from a given view point. Since a high frame rate is desirable, the user input, e.g. change of view point, should affect the rendered image immediately. Ideally, a real-time rendering of the scene should be achieved.

Categories of visibility algorithms:

- **Object space**

  theoretically device independent

  numerical precision depends on the machine precision

- **Image space**

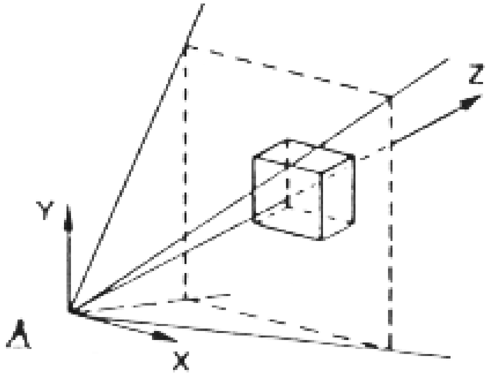
  device dependent

  numerical precision depends on the resolution of the output device

9.1 Overview

Invisibility or occlusion occurs, if due to the projection of a three-dimensional scene onto the image plane different objects are projected onto the same location.

Then, all those parts of the object are visible, that are closer to the eye of the viewer. Hence, cannot only consider the (x,y)-coordinates of the projected objects but also need to take the depth of the objects (z-coordinate) into account.



9.1 Overview

Coherence

In this case, coherence means to exploit local similarities

- **Object coherence**: if the projection of two objects do not intersect, then their surfaces do not need to be tested against each other.
- **Surface coherence**: properties of neighboring points on a single surface often do not chance significantly.
- **Depth coherence**: the depth $z(x,y)$ of a projected surface can often be computed incrementally.

9.2 Back-face Culling

Depending on the number and layout of objects within the scenes the removal of hidden surfaces can be quite costly. Therefore, a simple test which helps to reduce the complexity would be beneficial before using the visibility algorithm.

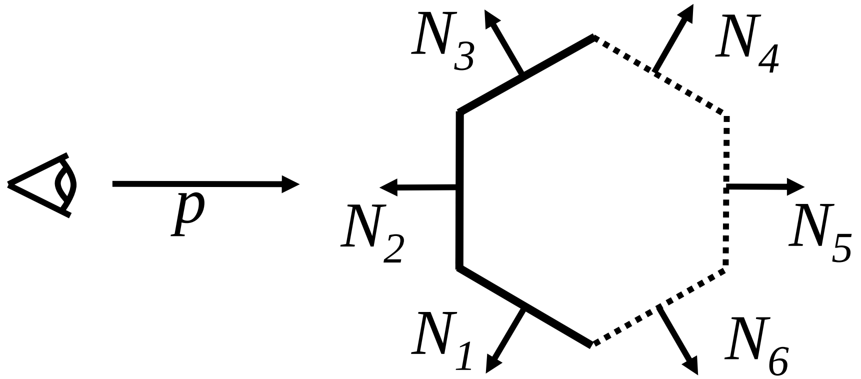
Such a simple but effective approach is back-face culling.

Depending on the position of the viewer the backsides of opaque objects are removed since these are occluded by the object itself, thus invisible.

9.2 Back-face Culling

Classification of backsides

- First, the normal vectors N_i of all surfaces are computed
- For back-facing surfaces, one component of the normal vector points in the view direction, i.e. the scalar product of the viewing direction and the normal is positive: $p \cdot N_i > 0$



$$p \cdot N_1 < 0 \qquad p \cdot N_4 > 0$$
$$p \cdot N_2 < 0 \qquad p \cdot N_5 > 0$$
$$p \cdot N_3 < 0 \qquad p \cdot N_6 > 0$$

9.2 Back-face Culling

Properties

- The number of polygons that are required for rendering the scene is approximately cut in half by removing the back-facing surfaces.
- The computational effort for computing the scalar product is minimal.
- If the scene is composed of a single convex polyhedron back-face culling already solves the visibility problem.

With scenes consisting of concave polyhedra or more than one convex polyhedron the objects can occlude themselves or each other which requires more complex algorithms.

9.3 Visibility algorithms

Known visibility algorithms:

- First solution of the **hidden-line** problem: Roberts, 1963

  object space algorithm for convex objects
- **Area sub-division** (divide and conquer): Warnock, 1969

  Exploits surface coherence using quadtrees
- **Sample spans**: Watkins, 1970

  Exploits the scan-line coherence
- **Depth list**: Newell et al., 1972

  Priority list algorithm within object space

Compared to the z-buffer algorithms above techniques did not gain very much popularity. This is mostly due to the versatility of the z-buffer algorithm. The listed algorithms all have certain limitations and can be applied only to specific scenes.

9.4 Z-buffer algorithm

Z-buffer algorithm [Catmull, 1975]

- Determines the visibility of pixels
- Works within image space
- Suitable for image output devices using rasters

Principle:

From a functional point of view, the z-buffer algorithm finds for each pixel within image space the polygon which covers this specific pixel and the z-value is minimal, i.e. it is in the front of all other polygons (for this pixel).

To implement the algorithm, additional memory is used (the so-called z-buffer) which stores for each pixel the currently smallest z-value.

9.4 Z-buffer algorithm

Algorithm

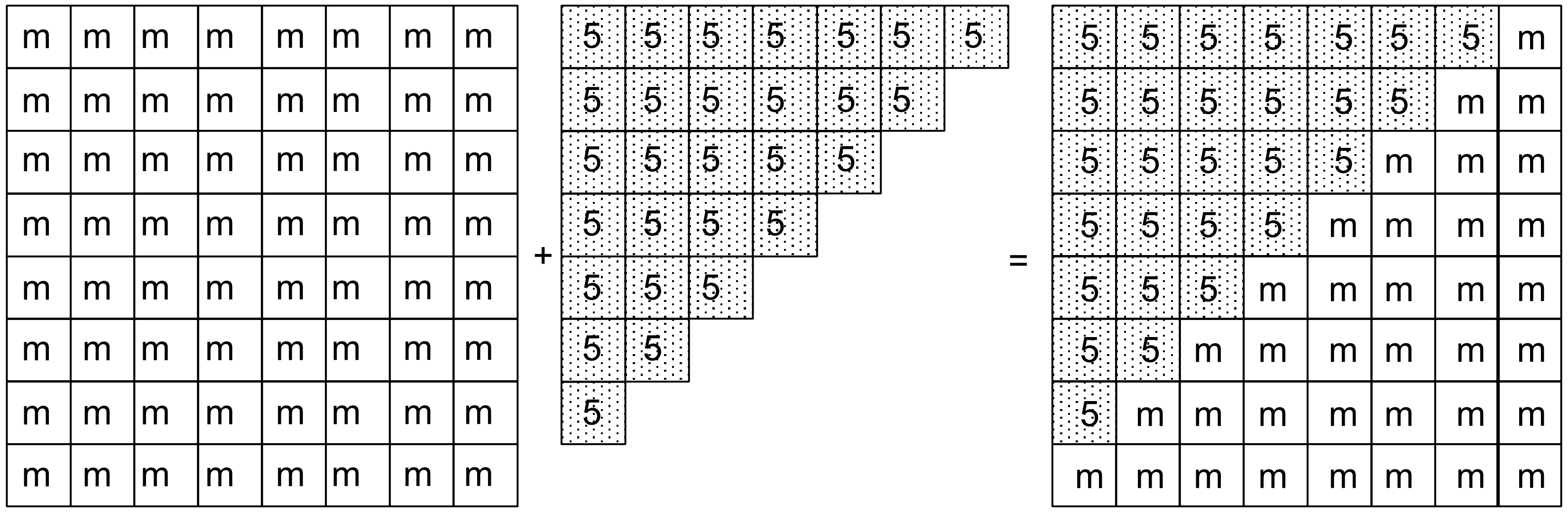
- Initialize frame buffer using the background color
- Initialize z-buffer using the maximal z-value
- Scan convert all polygons in **arbitrary order**:
    - Compute the z-value $z(x,y)$ for each pixel (x,y) of the polygon
    - If $z(x,y)$ is smaller that the current value in the z-buffer at location (x,y) change the color in the frame buffer at (x,y) to the color of the current polygon and set the z-buffer at (x,y) to $z(x,y)$.

After the algorithm is finished, the frame buffer contains the desired image and the z-buffer its depth values.

9.4 Z-buffer algorithm

Example

- Z-values are encoded by numbers where a smaller number means that the object is closer to the viewer.
- Initialize z-buffer with the maximal z-value.
- Add a polygon with constant z-value.



m	m	m	m	m	m	m	m
m	m	m	m	m	m	m	m
m	m	m	m	m	m	m	m
m	m	m	m	m	m	m	m
m	m	m	m	m	m	m	m
m	m	m	m	m	m	m	m
m	m	m	m	m	m	m	m
m	m	m	m	m	m	m	m

+

5	5	5	5	5	5	5	
5	5	5	5	5	5		
5	5	5	5	5			
5	5	5	5				
5	5	5					
5	5						
5							

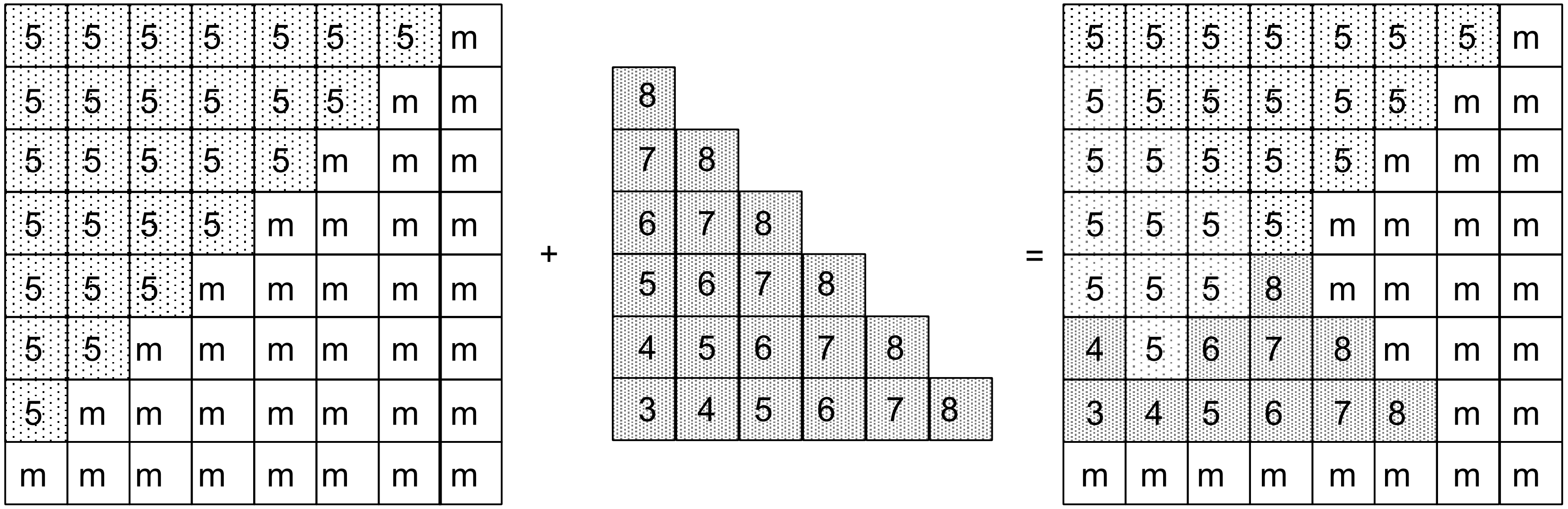
=

5	5	5	5	5	5	5	m
5	5	5	5	5	5	m	m
5	5	5	5	5	m	m	m
5	5	5	5	m	m	m	m
5	5	5	m	m	m	m	m
5	5	m	m	m	m	m	m
5	m	m	m	m	m	m	m
m	m	m	m	m	m	m	m

9.4 Z-buffer algorithm

Example

- Add another polygon which intersects the previous one.



5	5	5	5	5	5	5	m
5	5	5	5	5	5	m	m
5	5	5	5	5	m	m	m
5	5	5	5	m	m	m	m
5	5	5	m	m	m	m	m
5	5	m	m	m	m	m	m
5	m	m	m	m	m	m	m
m	m	m	m	m	m	m	m

+

8					
7	8				
6	7	8			
5	6	7	8		
4	5	6	7	8	
3	4	5	6	7	8

=

5	5	5	5	5	5	5	m
5	5	5	5	5	5	m	m
5	5	5	5	5	m	m	m
5	5	5	5	m	m	m	m
5	5	5	8	m	m	m	m
4	5	6	7	8	m	m	m
3	4	5	6	7	8	m	m
m	m	m	m	m	m	m	m

9.4 Z-buffer algorithm

Computing the z-value of polygons

For calculating the z-value $z(x,y)$ of a plane polygon (e.g. a triangle) we can exploit the scan-line coherence:

Plane: $Ax + By + Cz + D = 0$

Hence: $z = (-D - Ax - By)/C$

$$z(x + d_x, y) = z(x,y) - d_x \cdot A/C$$

Only one subtraction is required since A/C is constant and distance between two consecutive pixels on the same scan line $d_x = 1$.

9.4 Z-buffer algorithm

Advantages

+ Algorithm is simple to implement (especially in hardware).
+ Independent of the representation of the objects of the scene; the only requirement is that it needs to be possible to compute the z-value for every point of the object's surface.
+ No limitation regarding complexity of the scene.
+ No special order or sorting of the objects necessary

9.4 Z-buffer algorithm

Disadvantages

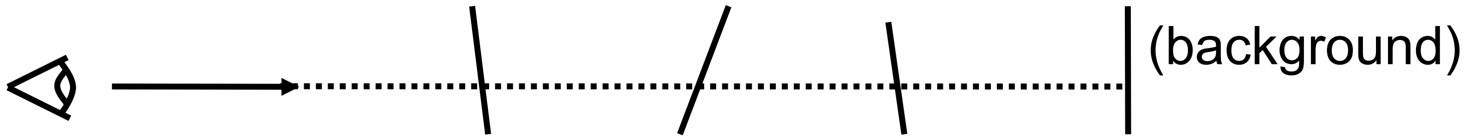
– Resolution of the z-buffer determines the discretization of the image depth, e.g. when using 20 bits 2^{20} different depth values are possible; hence, scenes with great distances between objects and high detailed objects are problematic.

– Requires additional memory – memory requirement can be reduced by sub-division of the image and processing each sub-image individually.

– Transparency (alpha-buffering) and anti-aliasing can only be integrated by costly modifications.

9.5 α-buffer algorithm

Transparent surfaces have – besides the color values – an additional attribute determining the opacity $\alpha \in [0,1]$ (0=transparent, 1=opaque). The color of a pixel is composed partly of the color of a polygon and the color of all objects located behind at a ratio of $\alpha : 1 - \alpha$.



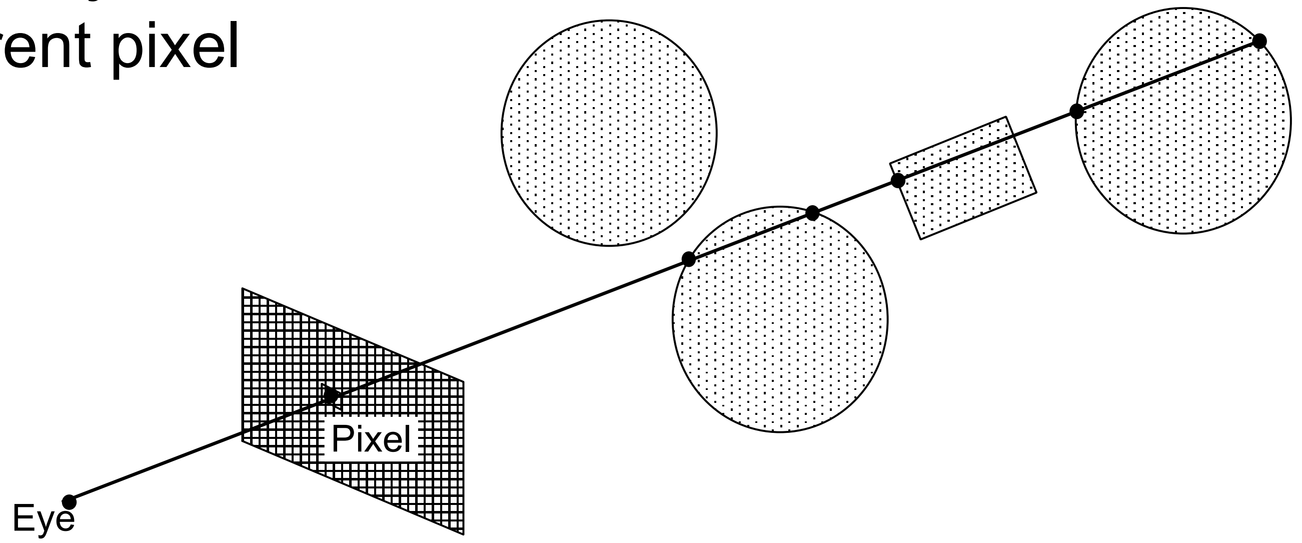
α–value:	0.2	0.5	0.3	1.0
Color contribution:	0.2	0.4	0.12	0.28

α-buffering works in the same way as z-buffering. The only difference is that the scene needs to be assembled back-to-front. Hence, the polygons need to be depth sorted first.

9.6 Ray casting

Idea:

- Trace a ray from the view point (eye) through each pixel within the image plane
- Compute the intersections with all objects of the scene
- The object with the closest intersection is visible at the current pixel



9.6 Ray casting

Computing the intersection with the ray

$$r(t) = e + t \cdot v$$

e	viewpoint (eye)
v	viewing direction (Pixel - e)
t	ray parameter

Example: Intersection with an implicitly given sphere

$$\|x - m\|^2 - r^2 = 0$$

Plugging in this equation into the definition of the ray then gives us (replacing x):

9.6 Ray casting

$$\|e + t\cdot v - m\|^2 - r^2 = 0$$

$$(e + t\cdot v - m)\cdot(e + t\cdot v - m) - r^2 = 0$$

$$((t\cdot v) + (e - m))\cdot((t\cdot v) + (e - m)) - r^2 = 0$$

$$t^2 v\cdot v + 2tv\cdot(e - m) + (e - m)\cdot(e - m) - r^2 = 0$$

Solving this quadratic equation with t as the unknown gives us maximally two intersections:

$$s_{1,2} = r(t_{1,2}) = e + t_{1,2}\cdot v$$

The intersection with the smaller t value with $t>0$ is closest to the view point (eye).

9.6 Ray casting

Example: intersection with a plane

p point on the plane

n normal vector

Plugging in the equation for the ray into the normal form of the plane:

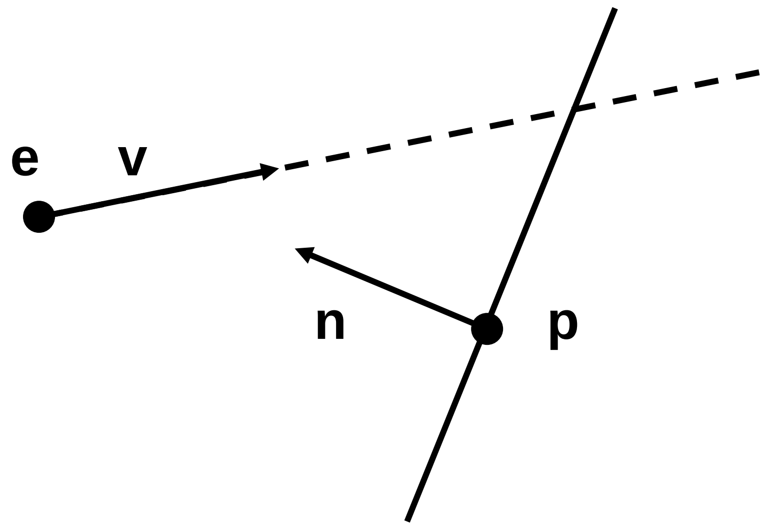
$$(x - p) \cdot n = 0$$

$$(e + t \cdot v - p) \cdot n = 0$$

$$(t \cdot v \cdot n + (e - p) \cdot n = 0$$

$$t = (p - e) \cdot n \, / \, v \cdot n$$

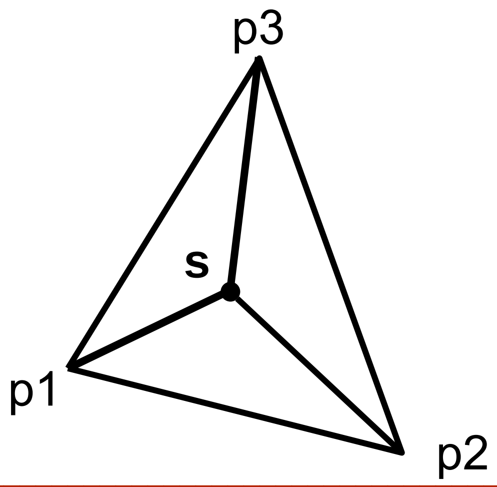
Intersection: $s = r(t) = e + t \cdot v$

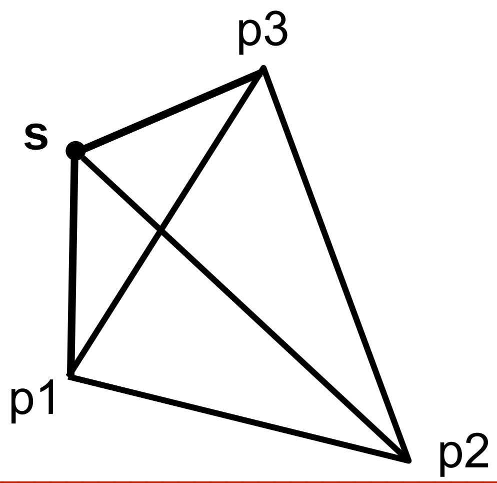


9.6 Ray casting

When computing the intersection with a polygon we also need to verify the resulting point:

- Consider the sum of all triangular areas introduced by the new point
- If the sum is larger than the area of the entire polygon, then the point is outside the polygon





9.6 Ray casting

Disadvantages:

- For every ray, the intersection with all objects need to be computed. Depending on the complexity of the scene, this can mean a lot of computations.
- At a resolution of 1024×1024 and a scene with 100 objects about 100 million intersections need to be computed!
- With a typical scene, up to 95% of the computational effort is spent on computing the intersections.

9.6 Ray casting

Speed-up approaches

- Transformation of the rays onto the z-axis; if all objects are transformed using the same transformation, then the intersection will always occur at $x=y=0$.
- Bounding box: for complex objects, a simple quad is computed that encloses the object; Then, if there is no intersection with the bounding box the ray will not intersect the enclosed objects either.
- Avoiding the computation of unnecessary intersections: hierarchies and space sub-division.

9.6 Ray casting

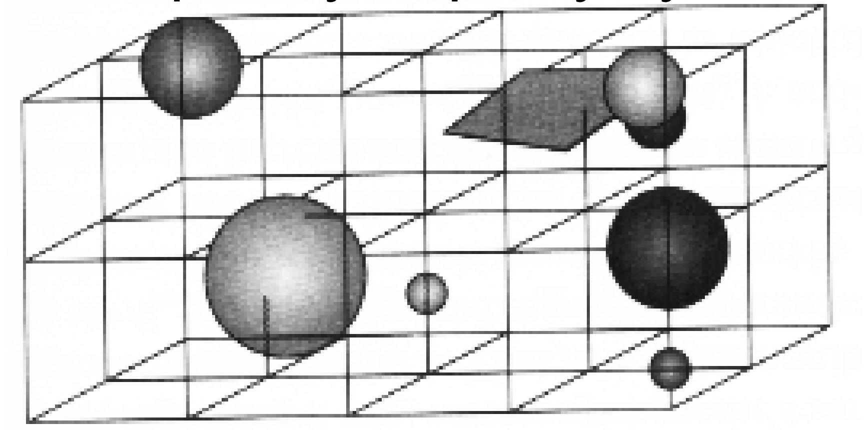
Hierarchies

- Tree-like data structures of bounding boxes
  - Leafs: objects of the scene
  - Inner node: bounding box of objects with sub-trees
- If a ray does not intersect a bounding box of an inner node, all objects within this bounding box, i.e. the node's sub-tree, does not need to be checked.
- Problem: it might be difficult to generate a suitable hierarchy.

9.6 Ray casting

Space sub-division

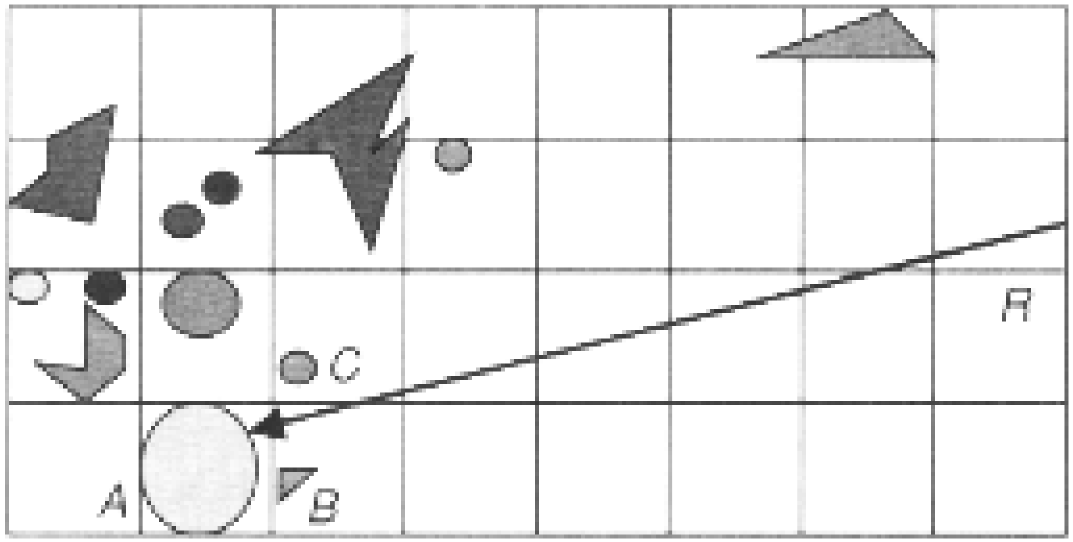
- Top-down approach
- First, the bounding boxes of a scene are computed.
- Then, this bounding box is sub-divided into equal parts.
- Each sub-division includes a list of all objects, that are contained completely or partly by the sub-division.



9.6 Ray casting

Space sub-division (continued)

- Only if a ray intersects a sub-division, the intersections with the contained objects need to be computed,
- Traverse the sub-divisions in the direction of the ray.



9.7 OpenGL Visibility Detection Methods

OpenGL implements two of the previously described visibility methods:

- Back-face culling
- Depth-buffer (z-buffer,α-buffer)

The implemented depth-buffer works fine in case where there are only opaque objects. However, the problem with the depth-buffer is that it requires additional care if there are transparent objects within the scene. Transparent objects need to be depth-sorted manually!

9.7 OpenGL Visibility Detection Methods

OpenGL back-face culling

To enable OpenGL's back-face culling you need to issue the command:

```
glEnable (GL_CULL_FACE);
glCullFace (mode);
```

The parameter `mode` can have the values `GL_BACK` (remove back-faces), `GL_FRONT` (remove front faces), or `GL_FRONT_BACK` (remove both front and back faces).

To disable back-face culling you can use the command:

```
glDisable (GL_CULL_FACE);
```

9.7 OpenGL Visibility Detection Methods

OpenGL depth-buffer

First, we need to request a visual that supports depth-buffering from the window system. With `GLUT` this can be done by adding `GLUT_DEPTH` to the list of parameters when initializing the display:

```
glutInitDisplay (GLUT_SINGLE, GLUT_RGB,
                 GLUT_DEPTH);
```

The depth-buffer values are initialized by:

```
glClear (GL_DEPTH_BUFFER_BIT);
```

9.7 OpenGL Visibility Detection Methods

OpenGL depth-buffer (continued)

The depth-buffer can be enabled or disabled if necessary:

```
glEnable (GL_DEPTH_TEST);
glDisable (GL_DEPTH_TEST);
```

You can also add graphics primitives, e.g. triangles or polygons, without changing the depth-buffer, i.e. setting the depth-buffer read-only:

```
glDepthMask (GL_FALSE);
```

To make the depth-buffer writable again issue:

```
glDepthMask (GL_TRUE);
```

9.7 OpenGL Visibility Detection Methods

OpenGL depth-buffer (continued)

In addition, you can manipulate the way the depth values are compared by changing the test condition:

```
glDepthFunc (testCondition);
```

The following parameters are valid:

```
GL_LESS

GL_GREATER

GL_EQUAL

GL_NOTEQUAL
```

…

9.7 OpenGL Visibility Detection Methods

Additional depth-cues

Sometimes it is desirable to enhance the three-dimensional impression by adding more depth-cues. Common depth-cues are:

- Visibility (objects in the front occlude objects in the back
- Shadows and shades
- Change in brightness reflecting distance (objects in the back are darker)

OpenGL does not support shadows directly. To create shadows, special shader-programs can be used.

9.7 OpenGL Visibility Detection Methods

Additional depth-cues

To achieve a decrease in brightness based on the distance to the view point, we can introduce a linear depth function:

$$f_{depth}(d) = \frac{d_{\max} - d}{d_{\max} - d_{\min}}$$

In OpenGL, we can then use fog to achieve the desired effect:

```
    glEnable (GL_FOG);

    glFogi (GL_FOG_MODE, GL_LINEAR);
```