DCGI

KATEDRA POČÍTAČOVÉ GRAFIKY A INTERAKCE

Visibility Algorithms

Jiří Bittner

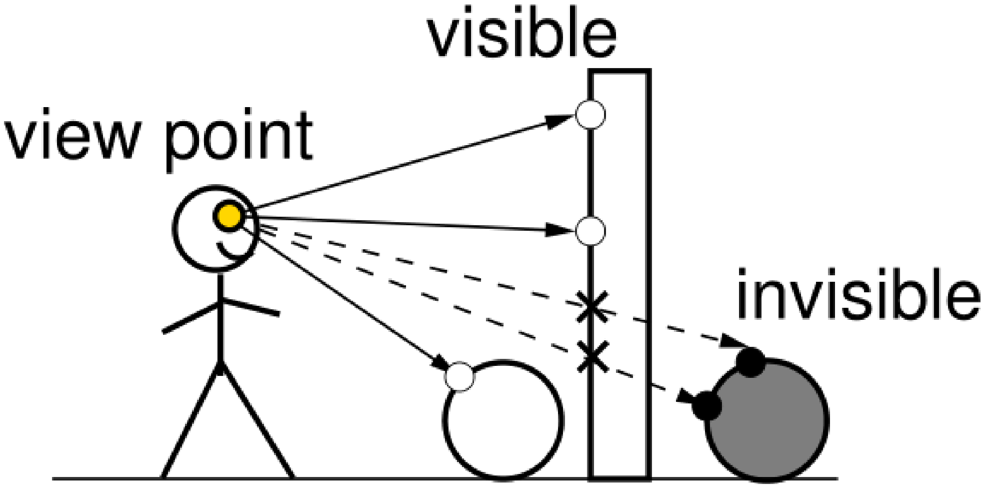
Outline

- Visibility in graphics
- Depth Buffer
- Ray Casting
- Painter's algorithm
- BSP Trees
- Warnock's Algorithm
- Specialized Visibility Algorithms

MPG – chapter 11

Visibility - Introduction

- Points A,B visible ⇔ line segment AB does not intersect opaque object
- Example: visibility from a view point



Visibility in Computer Graphics

- Hidden surface removal
- Shadows
- Radiosity
- Ray Tracing
- Visibility culling

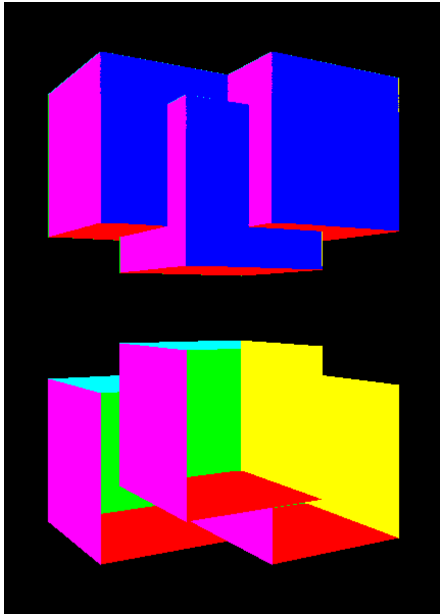
- Games / Multi-User Environments
- Streaming

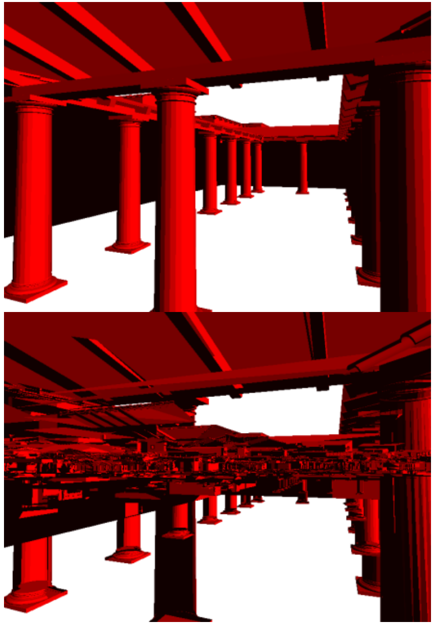
Hidden surface removal

- Creating “correct” 2D image of 3D scene
  - Finding visible objects and their visible parts
  - Eliminating invisible objects and invisible parts

ON

OFF





(6)

Visibility algorithms

- Raster algorithms (image space)
  - Solve visibility for pixels
  - For each pixel
    - Find nearest object projected to pixel
    - Shade the pixel using object color
  - Algorithms: z-buffer, ray casting, painters alg.
- Vector algorithms (object space)
  - Vector based description of visibility
  - For each object
    - Find object parts not hidden by others
    - Draw visible/invisible parts
  - Algorithms: Naylor, Weiler-Atherton, Roberts
  - CAD systems, technical drawings, special applications

Complexity: O(P.N)

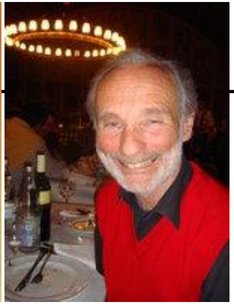
P .. #pixels
N .. #objects

Complexity: $O(N^2)$

Outline

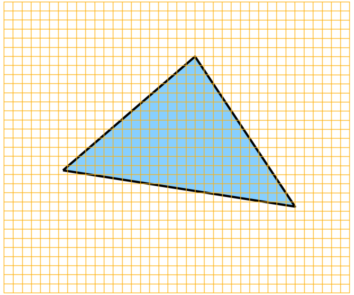
- Visibility in graphics MPG – chapter 11
- Depth Buffer
- Ray Casting
- Painter's algorithm
- BSP Trees
- Warnock's Algorithm
- Specialized Visibility Algorithms

Depth buffer (Z-buffer)



- Ed Catmull – 1975
  - Co-founder and president of Pixar
- Wolfgand Strasser - 1975
- For each pixel depth of the nearest object
- Process objects in **arbitrary order**

1. Rasterize to fragments
2. Compare depth of each fragment with z-bufer content
3. If closer overwrite z-buffer and pixel color



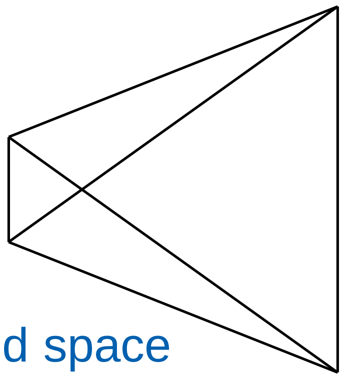
Depth buffer – pseudocode

- Two arrays: z_buffer, color_buffer

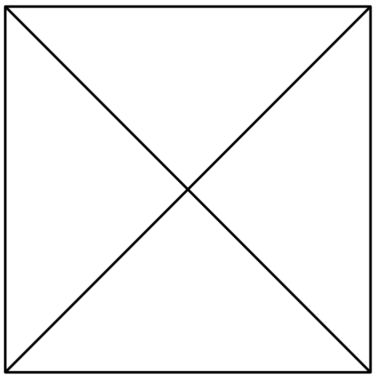
```
Clear color_buffer;
Set z-buffer to "infinity";

for (each object) {
    for (each object pixel P[x,y]) {
     if (z-buffer[x,y] > P[x,y].depth) {
          z_buffer[x,y] = P[x,y]. depth;
          color_buffer[x,y] = P[x,y].color;
     }
    }
}
```

Depth buffer - details

- Computing pixel depth - interpolation
- Linear interpolation of z'' ~ 1/z (z'' - device coordinates)
- For perspective projection depth resolution is non-uniform
  - Nearer objects have higher depth resolution



world space

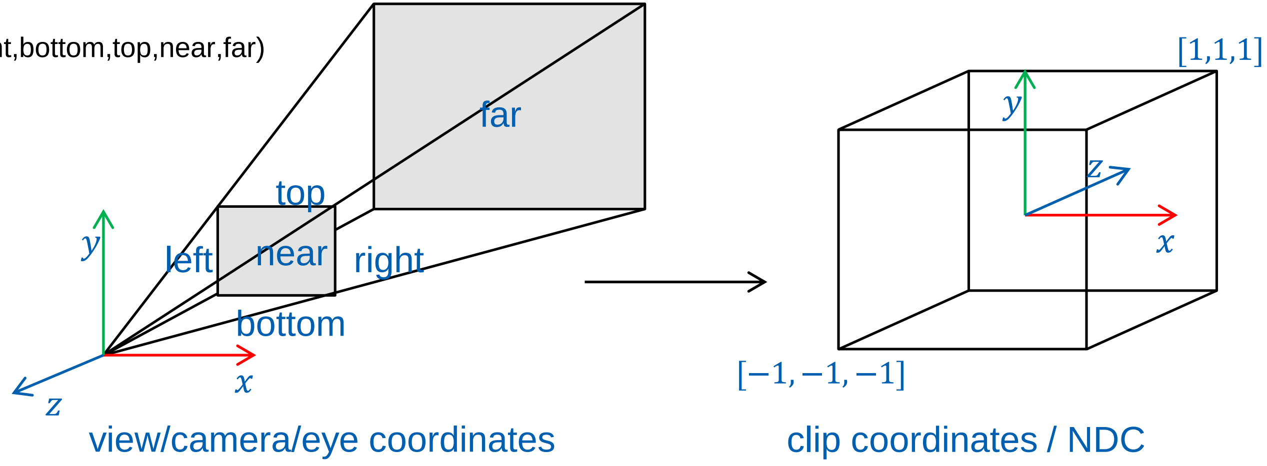


NDC

- **z-fighting** when rendering farther objects

Perspective projection - OpenGL

- glFrustum(left,right,bottom,top,near,far)



$$M = \begin{bmatrix} \frac{2near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2\,far\,near}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Perspective projection

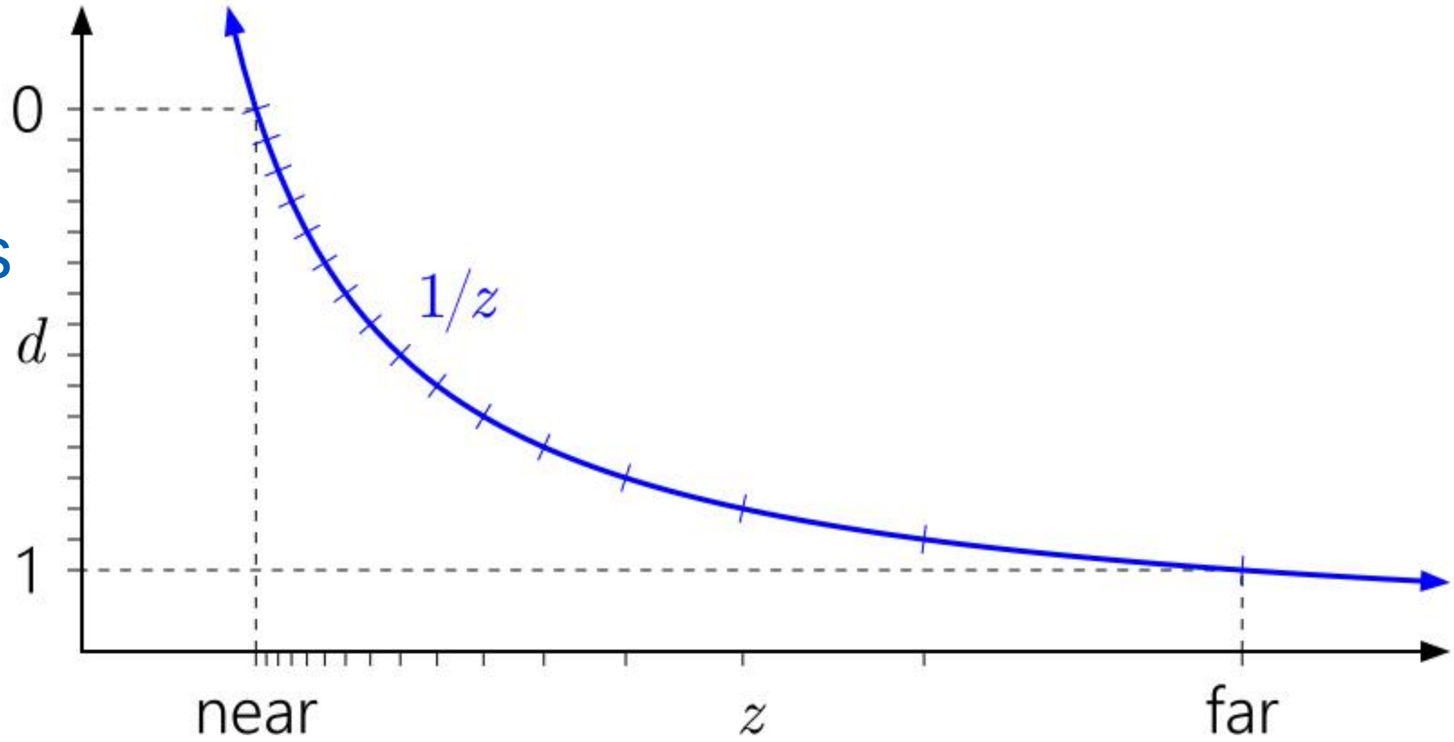
$$M = \begin{bmatrix} \frac{2near}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2\,far\,near}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$x' = \frac{2\ \mathrm{near}}{\mathrm{left}-\mathrm{right}}\frac{\mathrm{x}}{\mathrm{z}} - \frac{right+left}{right-left}$$

$$z' = \frac{near+far}{far-near} + \frac{2\,far\,near}{far\ -near}\frac{1}{z}$$

Depth Precision Issues - Example

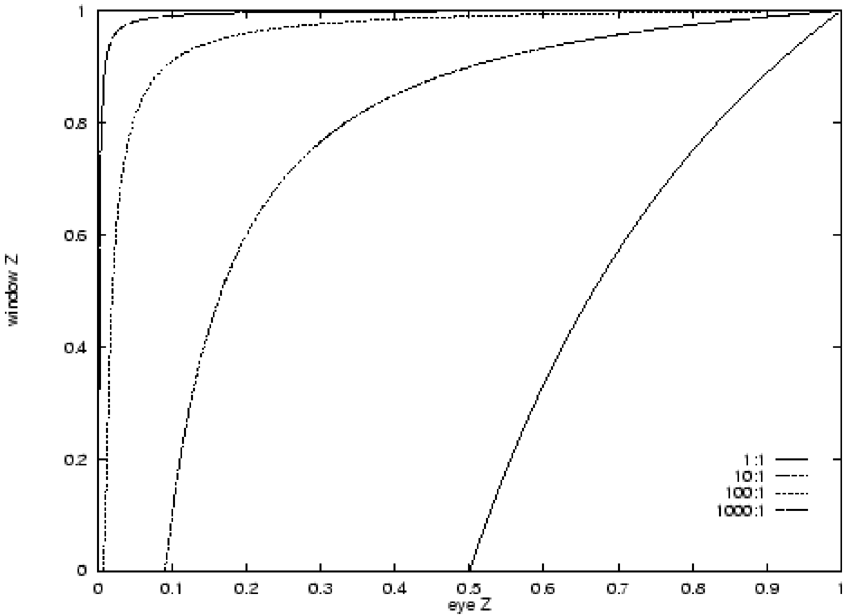
4bits



source: https://developer.nvidia.com/content/depth-precision-visualized

Depth distributions in z-buffer

- Careful setting of near-far planes
  - near = 1 / far = 10 : 50% between 1.0 a 1.8
  - near = 0.01 / far = 10 : 90% between 0.01 – 0.1
  - Median = 2*near*far/(near + far)

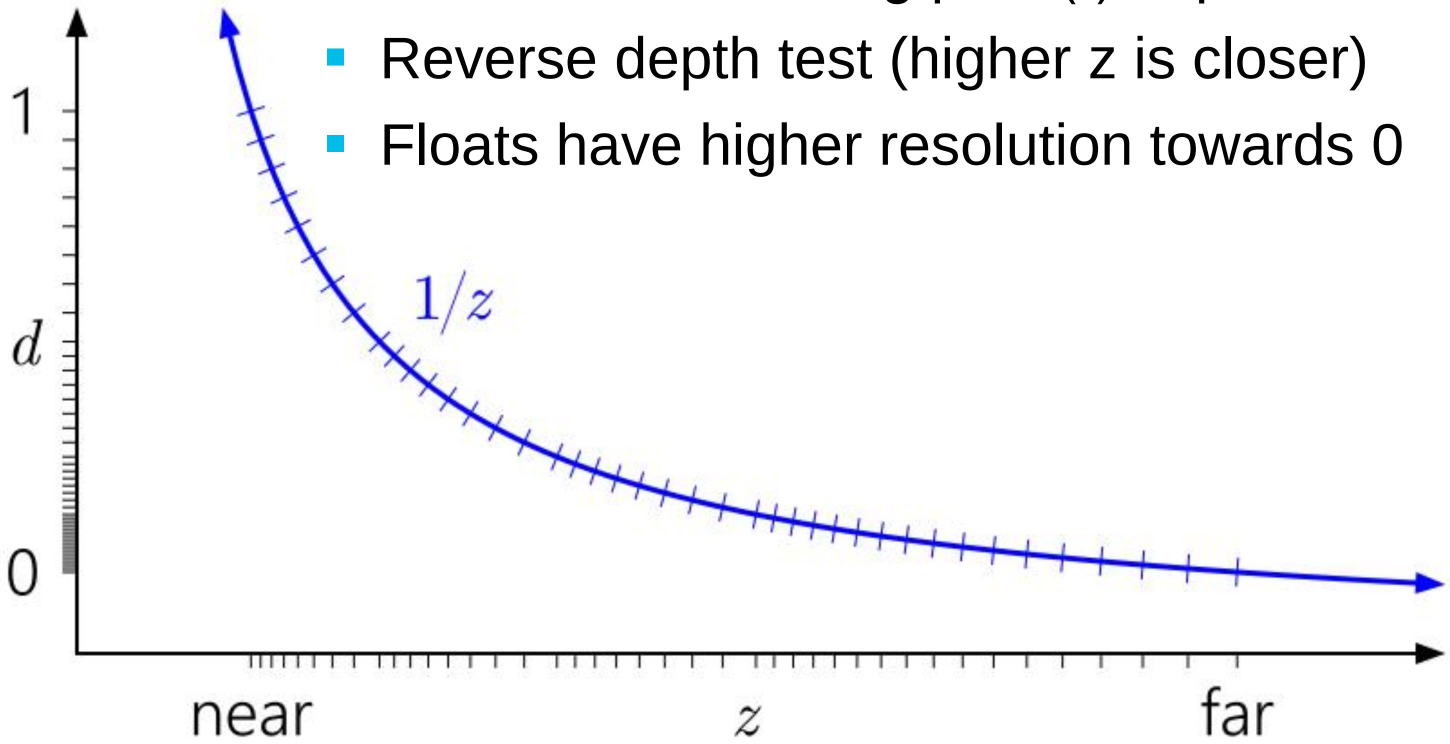


Resolving Z-fighting

- Careful settings of near(!) and far planes
- Rendering close and far objects
  - Several passes, updating near/far
  - Combine using stencil
- W-buffer
  - Stores eye space z, linear depth distribution
  - Reciprocal of z_i' for each pixel
- Reverse z
  - Lapidous and Jiao. Optimal depth buffer for low-cost graphics hardware. HWWS '99.
  - https://developer.nvidia.com/content/depth-precision-visualized

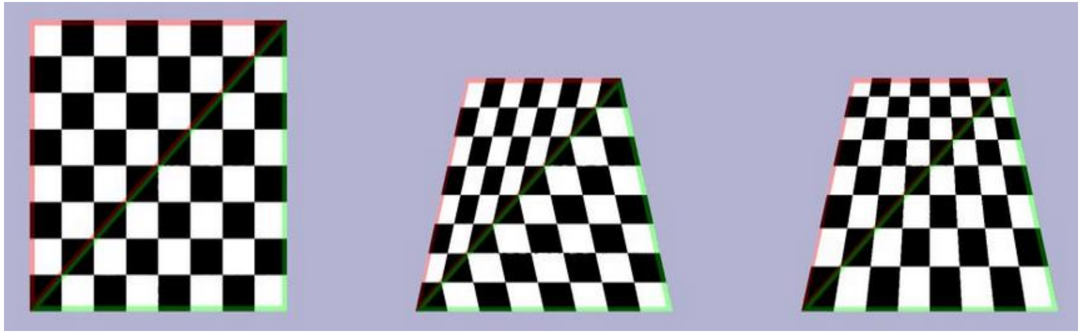
Reverse Z

- Use 1-z' and floating point(!) depth buffer
- Reverse depth test (higher z is closer)
- Floats have higher resolution towards 0



source: https://developer.nvidia.com/content/depth-precision-visualized

Perspectively correct interpolation



- LERP in screen space
  - non linear in object space (hyperbola) !
- Solution for color
  - Compute $c' = c/z$ and $z' = 1/z$
  - LERP of c' and z'
  - For each pixel $c_i = c_i'/z_i'$
- The same for texture coordinates u, v (!)
- Note: OpenGL stores 1/z in w' component after persp. divide
  - Compute $w' = 1/z$ and $c' = c*w'$
  - LERP of c' and w'
  - For each pixel $c_i = c_i'/w_i'$

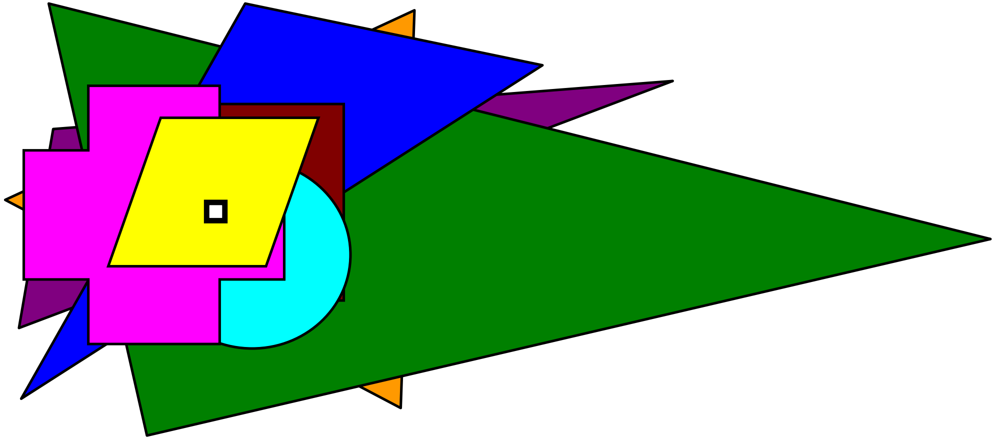
Depth buffer - properties

- Benefits
  - Simplicity
  - No preprocessing or sorting
  - Easy parallelization and HW implementation
- Issues
  - Pixel overdraw
  - Mapping depth to z-buffer bit range
  - Transparent objects
  - Alias

Quiz – number of overdraws

- 10 polygons project to pixel in random order
- What is the average number of overdraws?

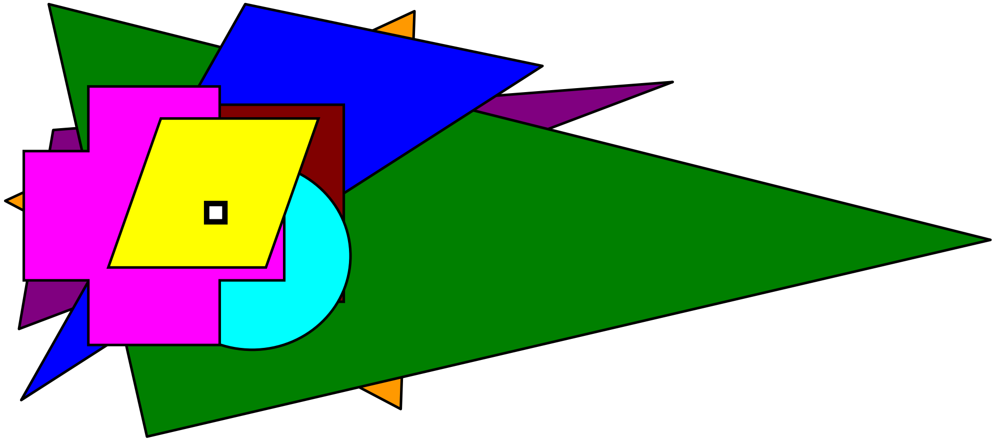
a) 3
b) 5.5
c) 7



Source: Eric Haines - Subtle Tools

Intuitive answer

- Front-to-back 1x, back-to-front 10x
- So the average is 5.5 overdraws



Correct answer

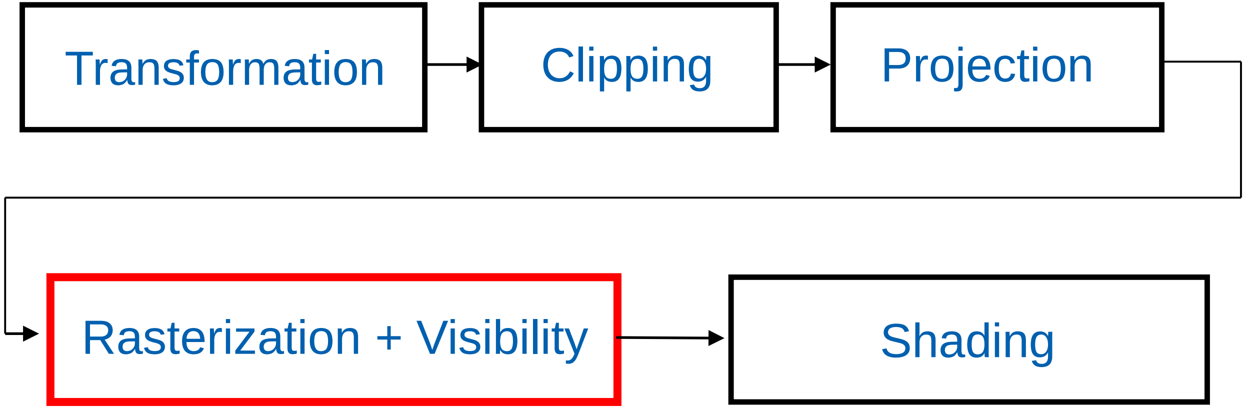
- The first polygon must cause overdraw: 1
- The second is either back or front
  - Chance of overdraw: ½
- Third polygon
  - 1/3 chance that it is the closest and causes overdraw
- Harmonic series: 1 + 1/2 + 1/3 + … + 1/10 = 2.9289

1 poly	1x
4 polys	2.08x
11 polys	3.02x
31 polys	4.03x
83 polys	5.00x
12,367 polys	10.00x

Aproximation for big N

overdraw(N) = ln(N) + 0.57721

Depth buffer in image pipeline



Depth buffer in OpenGL

- glutInitDisplayMode (… | GLUT_DEPTH | …);

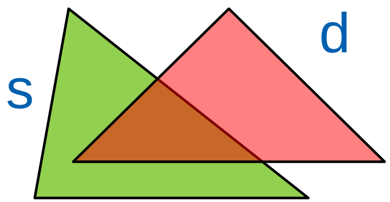
- glEnable(GL_DEPTH_TEST);
- glDepthFunc(GL_LESS);
- glClear(GL_DEPTH_BUFFER_BIT);
- glDepthMask(mask);
  - GL_TRUE read/write
  - GL_FALSE read only

Depth buffer and transparent objects

- Draw all non-transparent objects using z-buffer
- Sort all transparent objects back-to-front
- Render transparent objects with alfa-blending
  - OpenGL:
    - glDepthMask(GL_FALSE);
    - glBlendFunc(gl.ONE, gl.ONE_MINUS_SRC_ALPHA);
    - glEnable(GL_BLEND);
- Depth peeling
  - Iterative rendering layer by layer
  - Additional depth check: if z(current_layer) >= z(prev_layer) -> cull
  - Use “shadow test + alpha test” (Everitt 2001)

Alpha blending – Over & Under operator

- $C=(r, g, b, \alpha)$
- α opacity
  - $\alpha = 0$ transparent
  - $\alpha = 1$ opaque



$$C'_d = \alpha_s C_s + (1 - \alpha_s) C_d$$ s over d

$$C'_d = \alpha_s (1 - \alpha_d) C_s + \alpha_d C_d$$
$$\alpha'_d = \alpha_s (1 - \alpha_d) + \alpha_d$$ s under d

Depth buffer – Questions

- Should we draw back to front or front to back?
- How to increase depth resolution?

- When to perform the depth test?
- How to handle transparent objects?

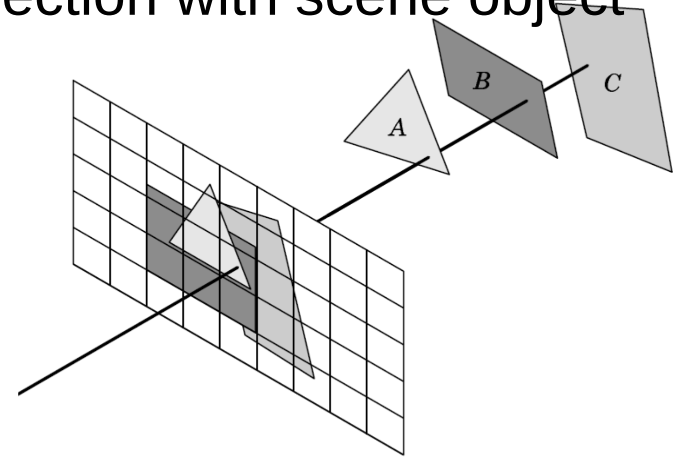
Outline

- Visibility in graphics
- Depth Buffer
- Ray Casting
- Painter's algorithm
- BSP Trees
- Warnock's Algorithm
- Specialized Visibility Algorithms

MPG – chapter 11

Ray casting

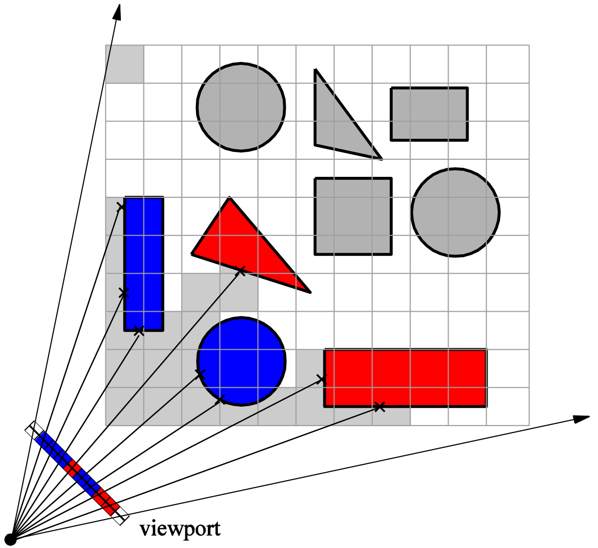
- Cast ray for each image pixel [Appel68]
- Find the nearest intersection with scene object



- Complexity
  - Naive: $O(R.N)$
  - With spatial data structure: $O(R.\log N)$

Accelerated ray casting

- Step 1: construct spatial DS
  - Preprocessing
  - BVH, kD-tree, octree, 3D grid
- Step 2: find the nearest intersection
  - Walk through cells intersected by the ray
  - Intersection found: terminate

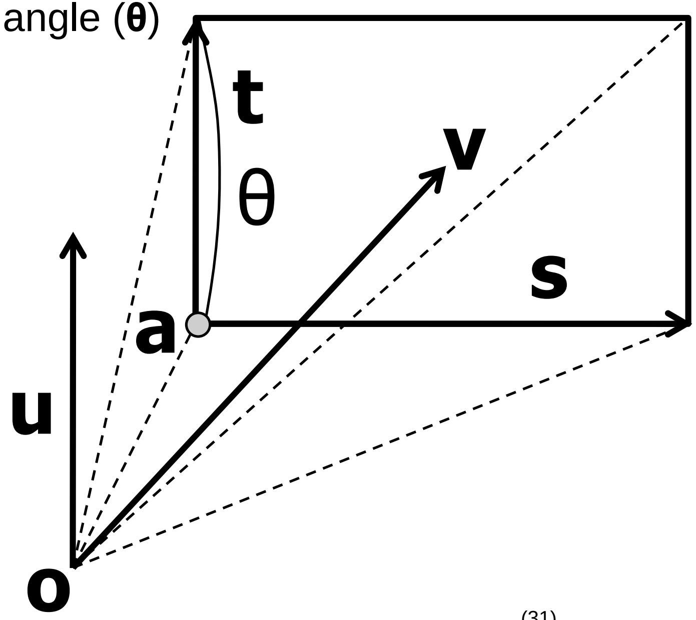


Ray Casting – Generating (Primary) Rays

- Implicit camera parameters
  - MVP matrix inversion
- Explicit knowledge of camera parameters
  - position (**o**), view direction (**v**), up vector (**u**), view angle (**θ**)

1. Compute view coordinate system: **a**, **s**, **t**
2. Ray through pixel x, y (image size width x height):

ray_origin = o;
ray_dir = Normalize(**a** + x/width***s** + y/height***t** – **o**);



Ray casting - properties

- Benefits
  - Flexibility (adaptive raster, ray tracing)
  - Efficient culling of occluded objects
- Drawbacks
  - Lower use of coherence
  - Requires spatial DS
    - Issue for dynamic scenes and HW implementation

Z-buffer vs. Ray Casting

	Scan-line coherence	Requires preprocessing	Efficient handling of occluded objects
Z-buffer	yes +	no +	no -
Ray casting	no -	yes -	yes +

Z-buffer better for dynamic scenes with low occlusion

Ray casting better for complex highly occluded scenes

Z-buffer GPU optimizations

- Z-cull
  - z_{min}, z_{max} for 8x8 pixel blocks
  - If $tri_{zmin} > tile_{zmax}$ discard
- Early-z test (for each pixel)
  - Apply z-test before shader execution
  - On newer GPUs used by default
  - Switched off when modifying “z” in shader
- HW occlusion queries, conditional rendering

Outline

- Visibility in graphics
- Depth Buffer
- Ray Casting
- Painter's algorithm
- BSP Trees
- Warnock's Algorithm
- Specialized Visibility Algorithms

MPG – chapter 11

Painter's algorithm

- Rendering back to front
- Farther patches overwritten by closer ones
- Used in 2D drawing tools (layers)

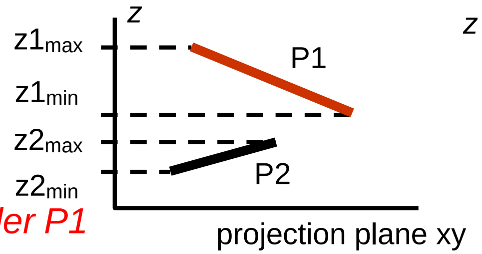
- In 3D without explicit ordering more complicated
- Depth sort algorithm [Newell72]

Depth Sort Painter's algorithm

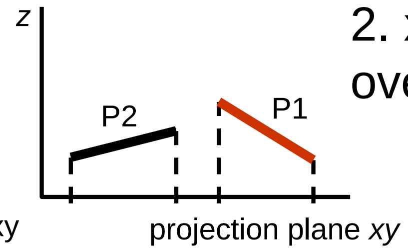
- Sort patches using z_{max} of each patch
- Farthest patch = **candidate for rendering (P1)**
- **Series of tests to confirm the candidate** using remaining patches

Depth Sort Painter’s algorithm – cont.

1. depth overlap



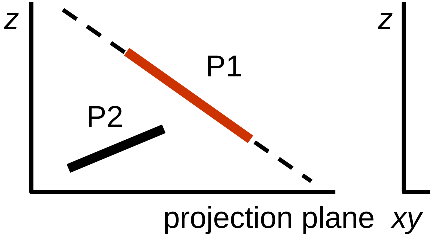
no overlap – render P1



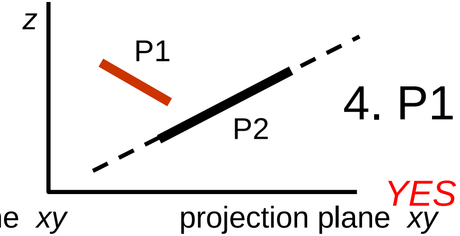
2. xy-projection overlap

no overlap – next patch

3. P2 before P1



YES – next patch



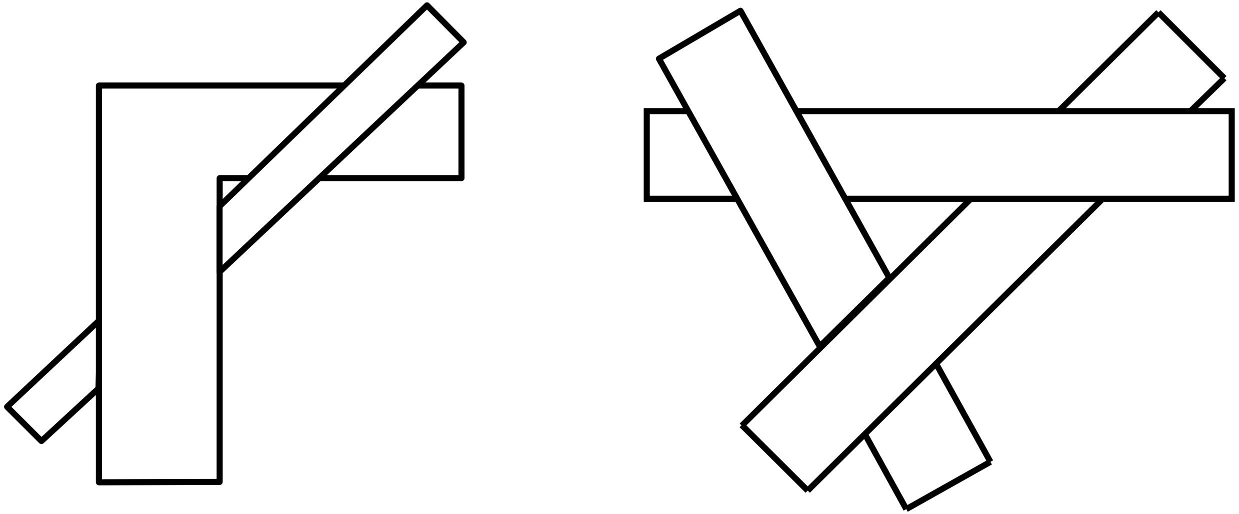
4. P1 behind P2

YES – next patch

Tests failed: swap (P2 = new candidate)

Cycle of candidates

- Can be detected using counter for candidate
- Solved by cutting the patch



Painter’s algorithm - properties

- Benefits
  - No depth buffer needed
  - Simplified version: easy implementation
- Issues
  - Overdraw
  - Correct depth order
  - Self intersections of patches not allowed

Outline

- Visibility in graphics
- Depth Buffer
- Ray Casting
- Painter's algorithm
- BSP Trees
- Warnock's Algorithm
- Specialized Visibility Algorithms

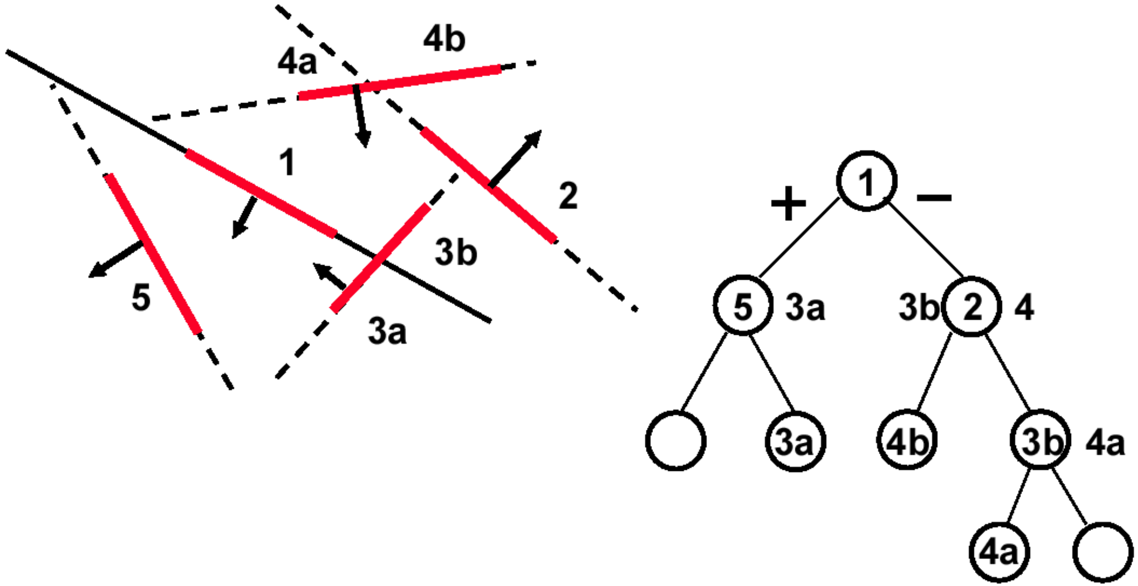
MPG – chapter 11

Binary Space Partitioning (*BSP*)

- View independent sorting of the scene [Fuchs80]
- Two phases
  - BSP tree construction (1x)
  - Tree traversal and rendering (as painter’s alg.)

BSP Tree Construction

- Recursive splitting by planes
- Planes typically defined using scene polygons (autopartition)
- Other planes can be used as well (e.g. axis-aligned)!



Rendering with BSP tree

```
void RenderBSP (Node S)
if (camera in front of S.plane) {
        RenderBSP (S.back);
        Render(S.polygons);
        RenderBSP (S.front);
}
else {
        RenderBSP (S.front);
        Render(S.polygons);
        RenderBSP (S.back);
}
}
```

BSP tree and Z-buffer

- Reduce number of overdraws
- Traverse front-to-back (reverse order compared to painter’s alg.)
- Alternatives to BSP tree
  - kD tree, octree, BVH

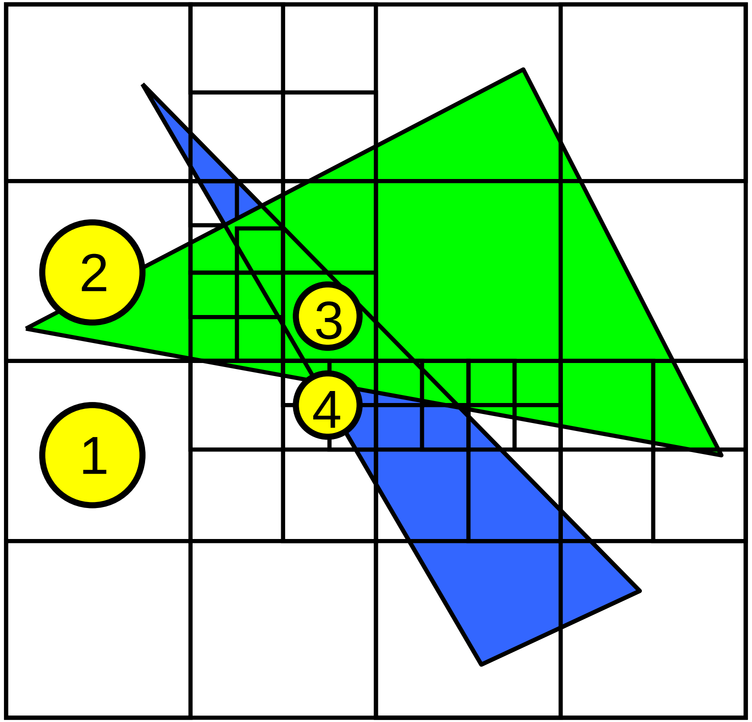
Outline

- Visibility in graphics
- Depth Buffer
- Ray Casting
- Painter’s algorithm
- BSP Trees
- Warnock’s Algorithm
- Specialized Visibility Algorithms

MPG – chapter 11

Image Subdivision – Warnock's alg.

- Recursive fast rectangle clipping tests
- Recursion terminates in pixel /subpixel



Divide and Conquer [Warnock69]

1. No object: *background color*
2. One object: *render*
3. More objects, one closest and fully covers: *render closest*
4. *Recursively subdivide*

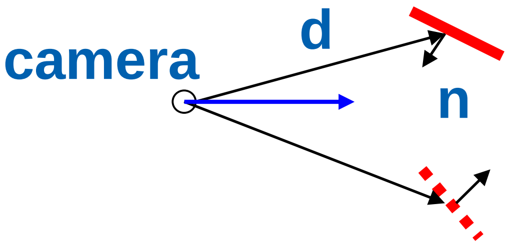
Outline

- Visibility in graphics
- Depth Buffer
- Ray Casting
- Painter's algorithm
- BSP Trees
- Warnock's Algorithm
- Specialized Visibility Algorithms

MPG – chapter 11

Back-face Culling

- Eliminates ~ 50% polygons
- If d*n > 0 : cull
- In NDC: just check for sign of n'_z
  - $n_z = e_x^1 e_y^2 - e_x^2 e_y^1$
  - Computed from transformed vertices (not shading normal)

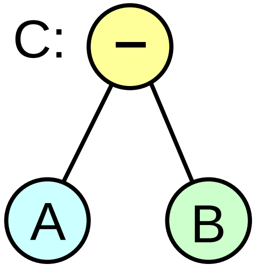


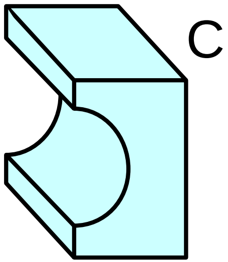
- OpenGL:

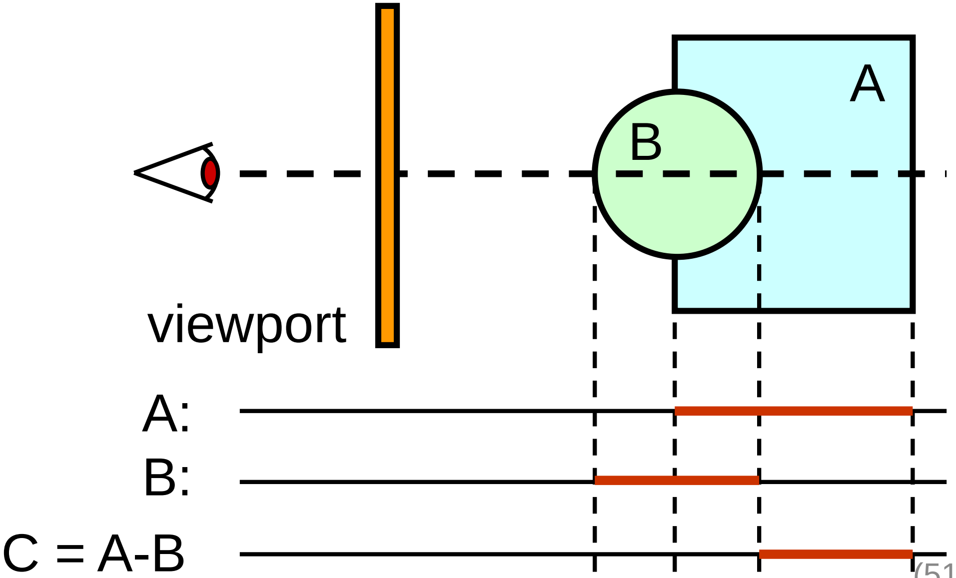
```
glFrontFace(GL_CCW);
glCullFace(GL_FRONT);
glEnable(GL_CULL_FACE);
```

Direct rendering of CSG models

- Specialized ray casting
- Intervals of ray/object intersections
- Solving set operations = set operations on intervals

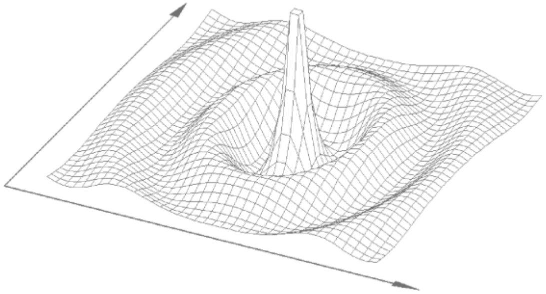


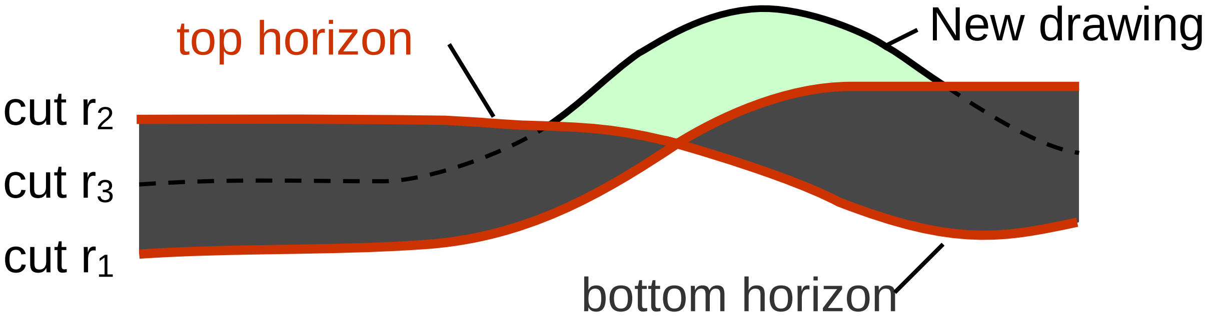




Floating horizon algorithm

- Graphs of functions z = (x,y)
- Terrains (height field)
- Algorithm outline
  - Render front-to-back
  - Keep bottom and top horizon



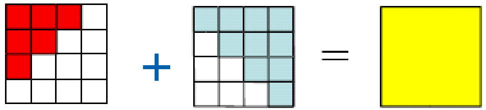


A-Buffer

- Antialiasing, correct transparency
  - [Carpenter84], Lucasfilm: “The Road To Point Reyes”
  - Later used in RenderMan (Pixar)
- Ordered list of primitives for each pixel
- Storing not just depth
  - transparency, coverage, object ID, normal,…
- Polygon rasterization
  - Non-transparent polygon covers the whole pixel – add to list and **remove** farther ones
  - Transparent polygon or partial pixel coverage – insert to list, **do not remove** farther ones

A-Buffer

- Rendering pass
  - For each pixel process the list front-to-back
  - Composition (subpixel rasterization, coverage mask 4x4 or 4x8)
  - Similar to MSAA
    (shading once, visibility multiple times)
- Benefits
  - More general than z-buffer
  - Handles transparency
  - Antialiasing
  - Used in production rendering



```
fragment_ptr    next;
short_int       r, g, b;         /* color, 12 bit */
short_int       opacity;         /* 1 - transparency */
short_int       area;            /* 12 bit precision */
short_int       object_tag;      /* from parent surface */
pixelmask       m;               /* 4x8 bits */
float           zmax, zmin;      /* positive */
```

Figure 3. Fragment definition.

Loren Carpenter. 1984. The A -buffer, an antialiased hidden surface method. SIGGRAPH '84.

Other buffers…

A-buffer - Carpenter, 1984
G-buffer - Saito & Takahashi, 1991
M-buffer - Schneider & Rossignac, 1995
P-buffer - Yuan & Sun, 1997
T-buffer - Hsiung, Thibadeau & Wu, 1990
W-buffer - 3dfx, 1996?
Z-buffer - Catmull, 1973 (?)
ZZ-buffer - Salesin & Stolfi, 1989

Accumulation Buffer - Haeberli & Akeley, 1990
Area Sampling Buffer - Sung, 1992
Back Buffer - Baum, Cohen, Wallace & Greenberg, 1986
Close Objects Buffer - Telea & van Overveld, 1997
Color Buffer
Compositing Buffer - Lau & Wiseman, 1994
Cross Scan Buffer - Tanaka & Takahashi, 1994
Delta Z Buffer - Yamamoto, 1991
Depth Buffer - 1984
Depth-Interval Buffer - Rossignac & Wu, 1989
Double Buffer - 1993

Escape Buffer - Hepting & Hart, 1995
Frame Buffer - Kajiya, Sutherland & Cheadle, 1975
Hierarchical Z-Buffer - Greene, 1993
Item Buffer - Weghorst, Hooper & Greenberg, 1984
Light Buffer - Haines & Greenberg, 1986
Mesh Buffer - Deering, 1995
Normal Buffer - Curington, 1985
Picture Buffer - Ollis & Borgwardt, 1988
Pixel Buffer - Peachey, 1987
Ray Distribution Buffer - Shinya, 1994
Ray-Z-Buffer - Lamparter, Muller & Winckler, 1990
Refreshing Buffer - Basil, 1977
Sample Buffer - Ke & Change, 1993
Shadow Buffer - GIMP, 1999
Sheet Buffer - Mueller & Crawfis, 1998
Stencil Buffer - 1997?
Super Buffer - Gharachorloo & Pottle, 1985
Super-Plane Buffer - Zhou & Peng, 1992
Triple Buffer
Video Buffer - Scherson & Punte, 1987
Volume Buffer - Sramek & Kaufman, 1999

Source: Eric Haines - Is the Hardware Z-Buffer Doomed?

Outline

- Visibility in graphics
- Depth Buffer
- Ray Casting
- Painter's algorithm
- BSP Trees
- Warnock's Algorithm
- Specialized Visibility Algorithms

MPG – chapter 11

DCGI

KATEDRA POČÍTAČOVÉ GRAFIKY A INTERAKCE

Questions?