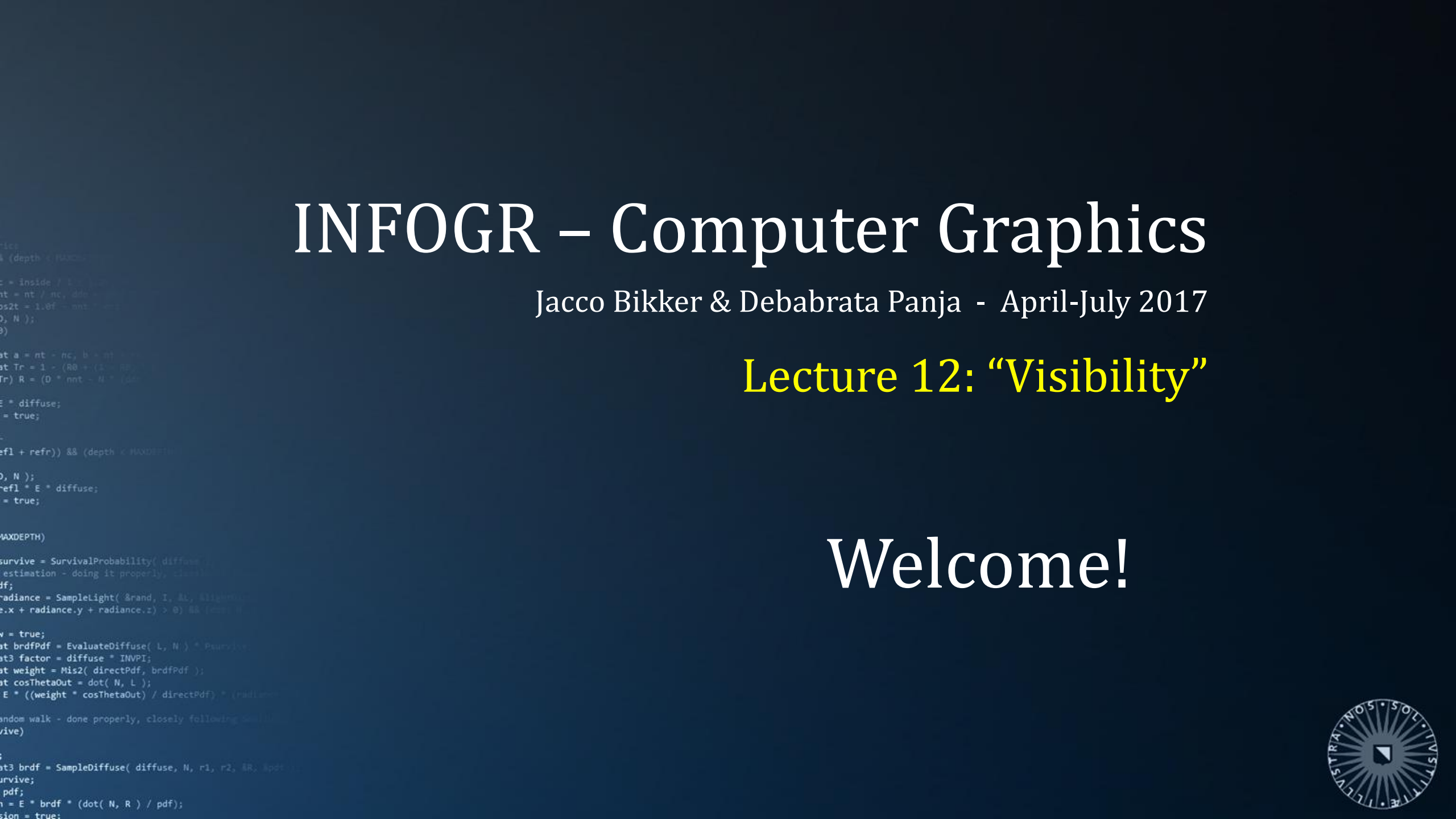
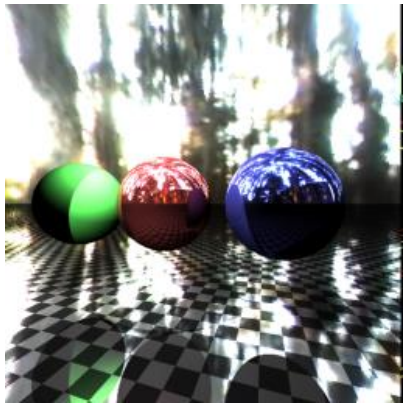
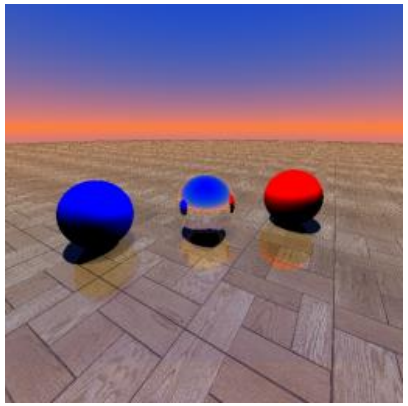
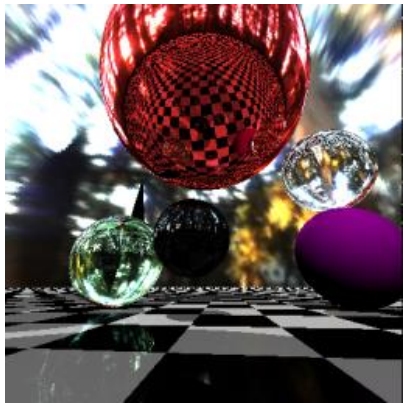
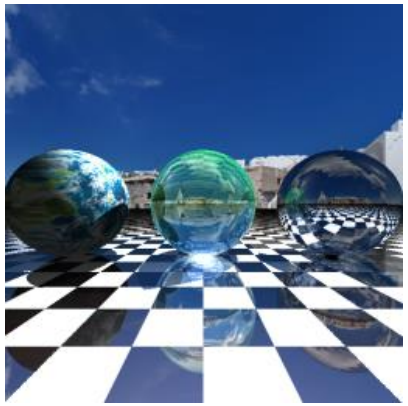
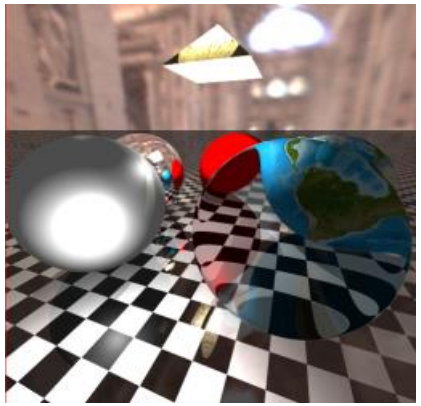
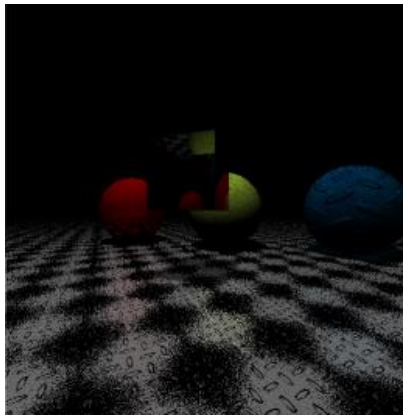
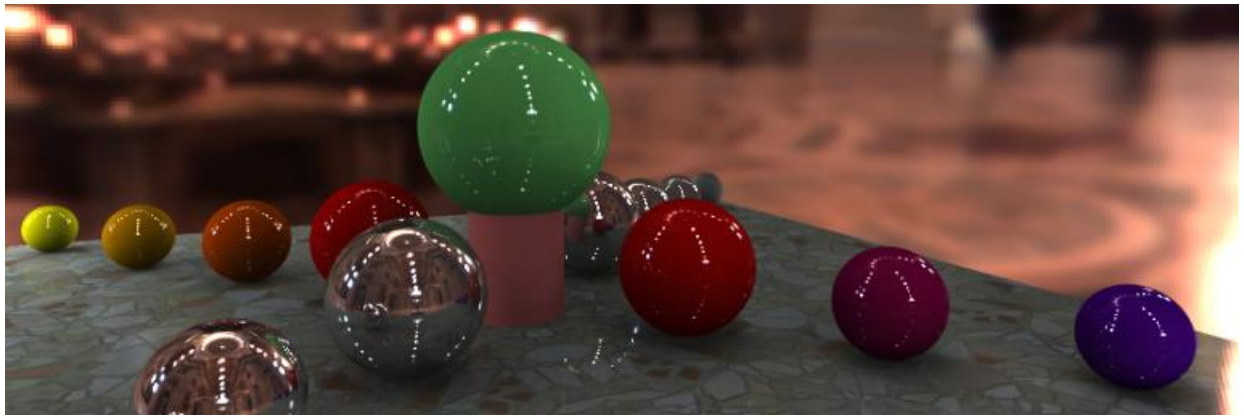
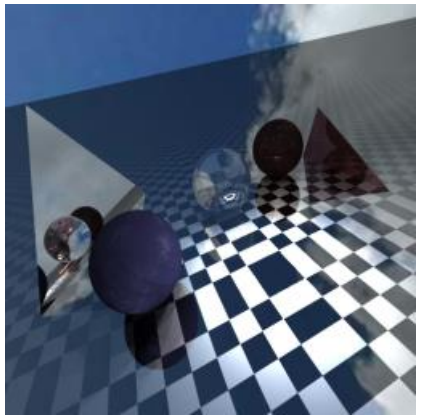
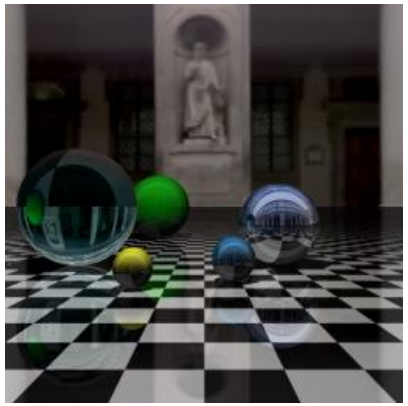
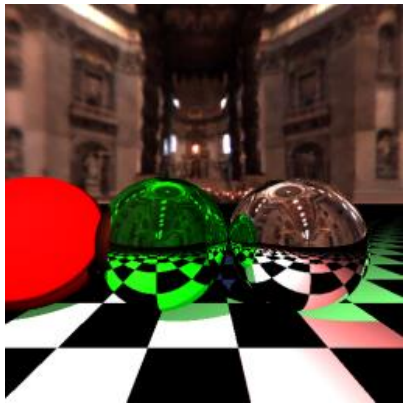
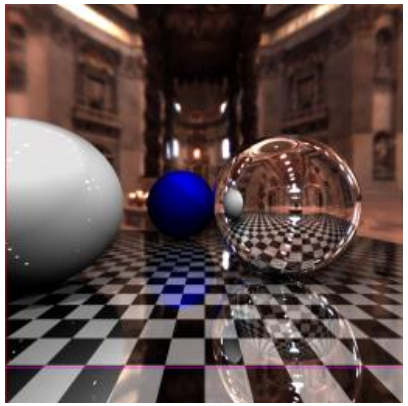
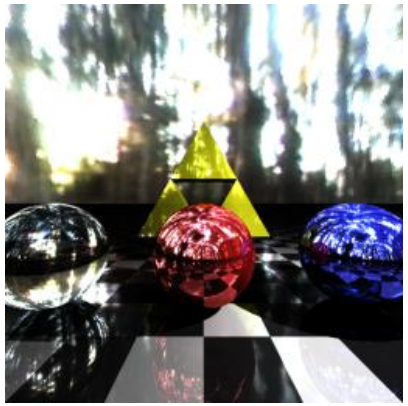
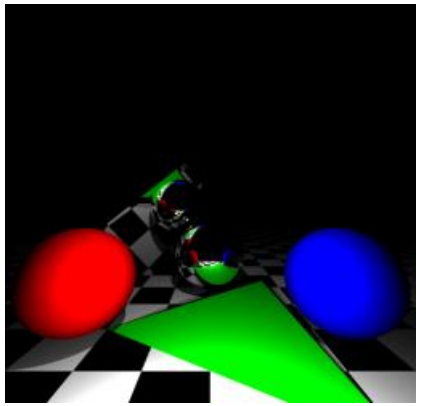
INFOGR – Computer Graphics

Jacco Bikker & Debabrata Panja - April-July 2017

Lecture 12: “Visibility”

Welcome!



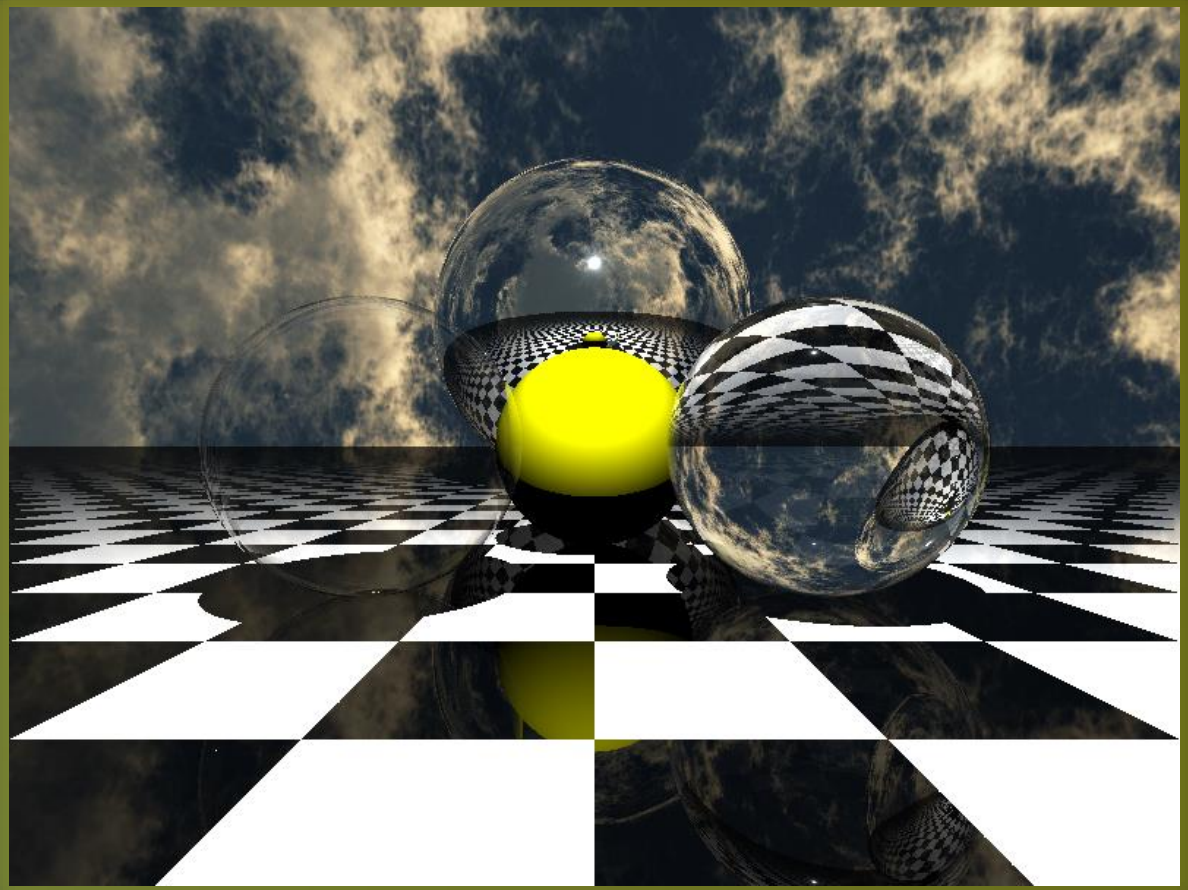
Smallest Ray Tracers:


```
#import<cmath>
#import<cstdio>
#define R return
#define O operator
#define P putchar
#define I int
#define f float
I l=2e9;struct v{f x,y,z;v(){}v(f a,f b,f c){x=a;y=b;z=c;}v O+(v o){R{x+o.x,y+o.y,z+o.z};}
v O-(){R*this*-1;}v O*(f s){R{x*s,y*s,z*s};}f O%(v o){R x*o.x+y*o.y+z*o.z;}v O!(){v t=*this;
R t*(1/sqrt(t%t));}};struct b{v p,c;f r;};f T(v o,v d,b*W,v*C,I M){if(M>9)R l;f Z=l,t,h;
v c,p,q;for(I i=3;i--;)c=W[i].p+-o,t=c%d,c=c+d*-t,(h=W[i].r-c%c)>0&(t-=sqrt(h))>.01&Z>t?
(*C=W[i].c,Z=t,c=d*Z+o,p=!(c+-W[i].p),p=d+p*-2*(p%d),T(c,p,W,&c,M+1),*C=v(C->x*c.x,C->y*c.y,
C->z*c.z),1):1;if(d.y<0){q={0,1,0};t=-o%q/(d%q);if(t>.01&Z>t)Z=t,c=d*Z+o,*C=T(c,p=!v(5-c.x,
5-c.y,-c.z),W,C,M+1)<l?v(0,0,0):v(t=I(c.x)+I(c.z)&1,t,t)*(q%p);}Z==l&d.y>0?*C={.5,1,1}:q;
R Z;}main(){printf("P6 512 512 255 ");v c;b W[]={{{2,3,6},{1,0,0},3},{{5,1,4},{0,1,0},1},{{
9,3,7},{0,0,1},4}};for(f y=512;y--;)for(f x=512;x--;)T({5,2,0},!v(1-x/256,y/256-1,1),W,&c,0)
,c=c*255,P(c.x),P(c.y),P(c.z);}
```

869

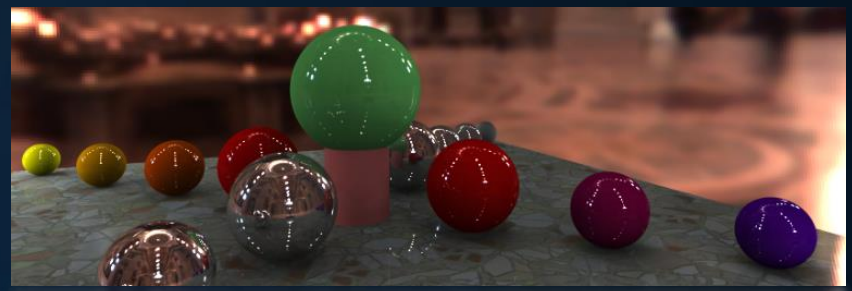
```
L=:9e9e=:1e_4h=:%:@:(+/@:*:)n=:%hd=:+/@:*Y=: :0r=:4 Yt=.(-&0.5)@:%&s(3$0);(n(t x),(t s-y),1);L)V=:3 Y(;0{y)+(0{;2{y)*;1{y)i=:4 Y
c=.(;0{0{y)-;0{xt=.c d;1{xq=.t-%:(*:;1{0{y)-+/*:c-t*;1{xe=.(<;2{x),<yif.0<*:q-tdo.if.(q<;2{x)*q>0do.e=.(<q),<yend.end.e)k=:4 Yif.
(;0{x)<;0{ydo.xelse.yend.)G=:3 Yk/1(y&i)\4 5$((_3,0,8);1;(0,0,1);0.1;0),((0,0,8);1;(0,1,0);0;0),((3,0,8);1;(1,0,0);0.4;0),
(0,_10001,8);1e4;(3$e);0.2;1)K=:4 Y0.5>(x{y)-<.x{y)c=:3 YD=:D+1F=.G ye=.0{;0{FS=:0{;1{Fm=.0{;3{Sr=.(2{.y),<ei=.V rN=.n i-;0{Su=.
(e<L)*(1-m)*((;2{S)*3$-.(;4{S)*(0 K i)=2 K i)*r A SM=.0if.(e<L)*(m>0)*D<9do.M=.m*c i;(n(;1{r)-N*((;1{r)d N)*2);Lend.u+M)T=:4 Y
p=.V;0{xs=.(;0{0{y)-pD=.n st=.h s(3$(;0{G(p+D*e);D;t)>=t)*(;1{0{y)*(1%*:t)*(;1{x)d D)F=:4 Y(0>.;x)+0>.;y)A=:4 YF/1(<@:((x;-(;0{y)-V
x)&T))\2 2$(0,9,0);3$99)P=:3 YD=:0":255<.<.255*c(s|y)r<.y%s)s=:512stdout'P3 512 512 255',1 P\i.*:sexit''
```

Fastest Ray Tracer:



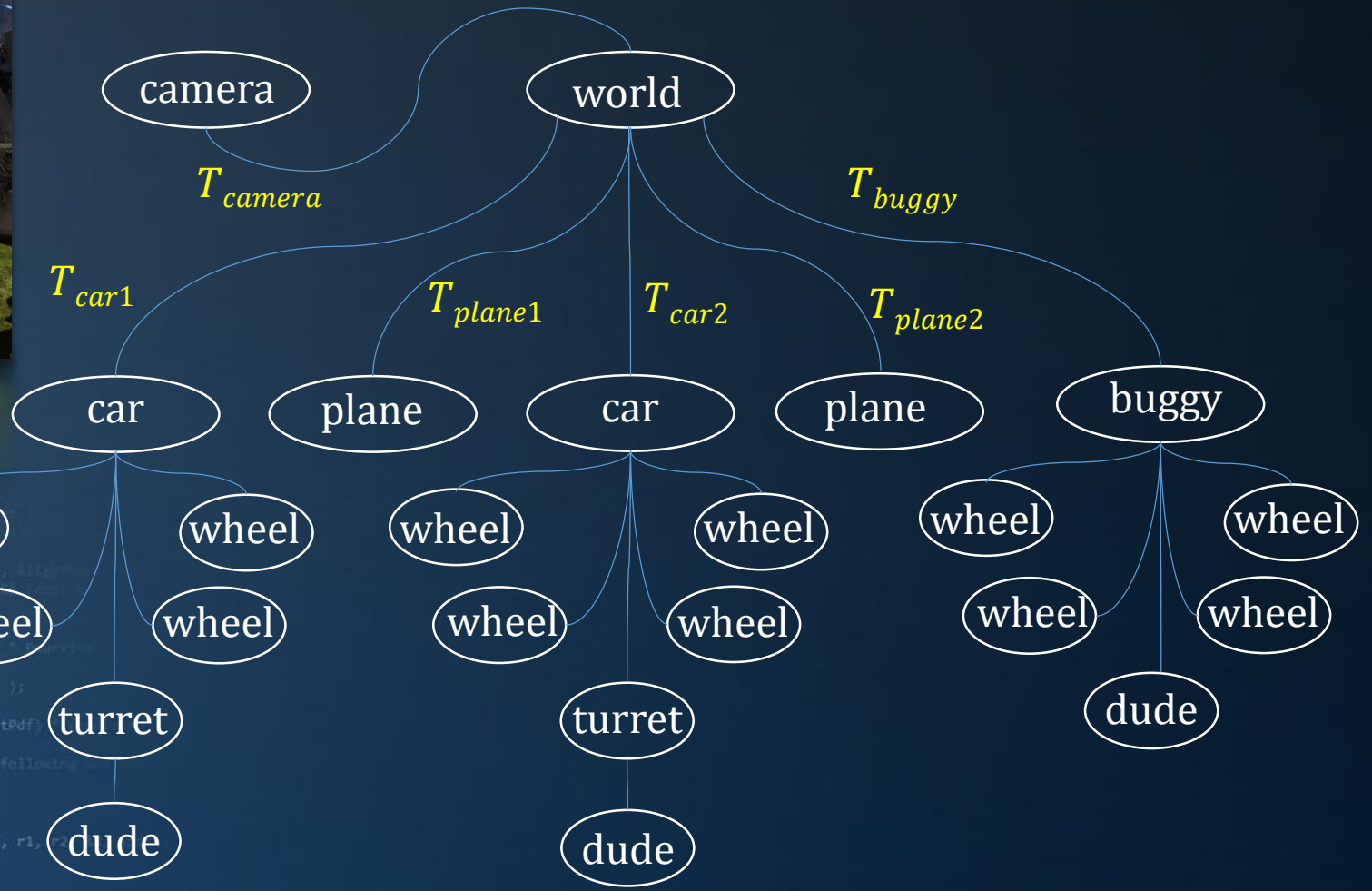
Today’s Agenda:

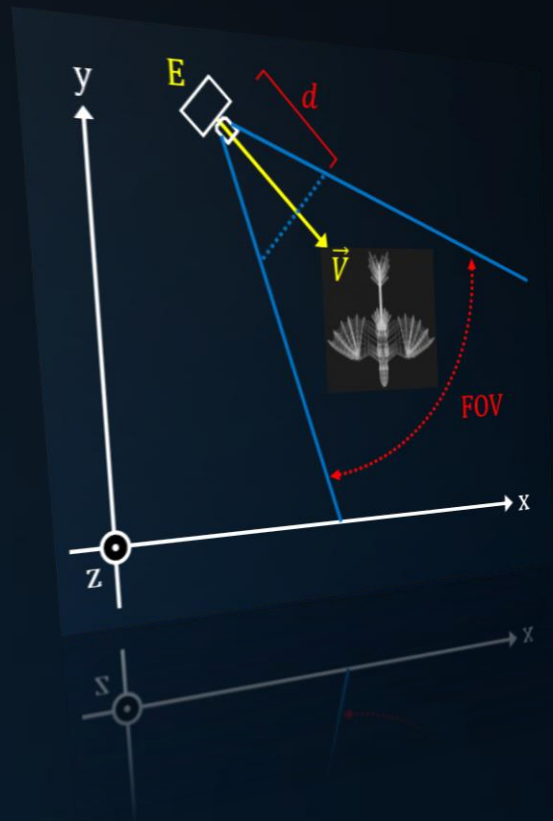
- Depth Sorting
- Clipping
- Visibility



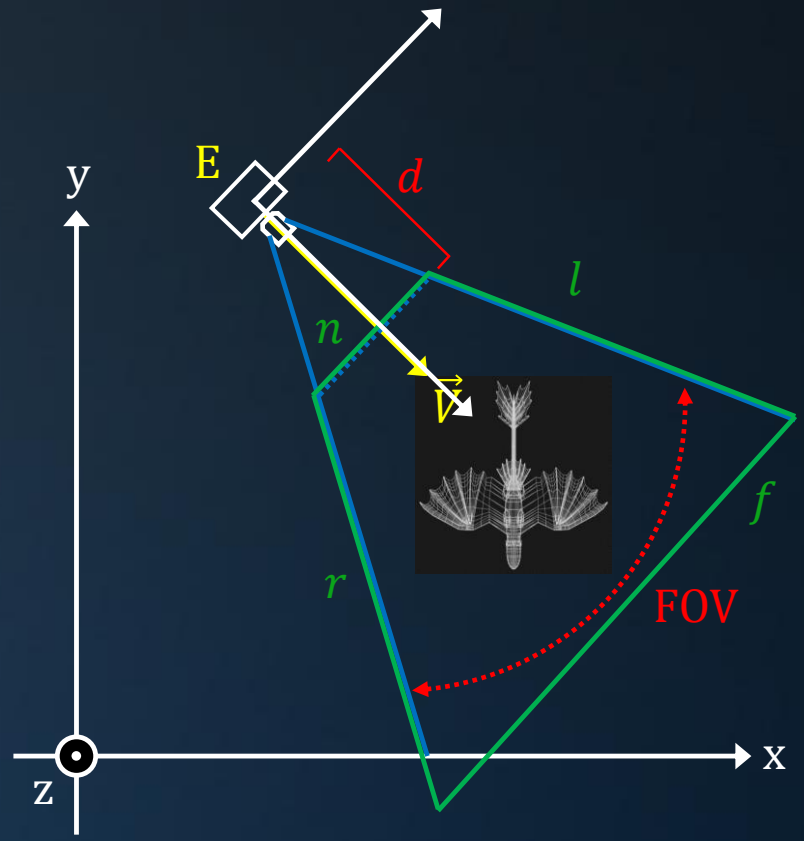
Perspective

Scenegraph





Perspective

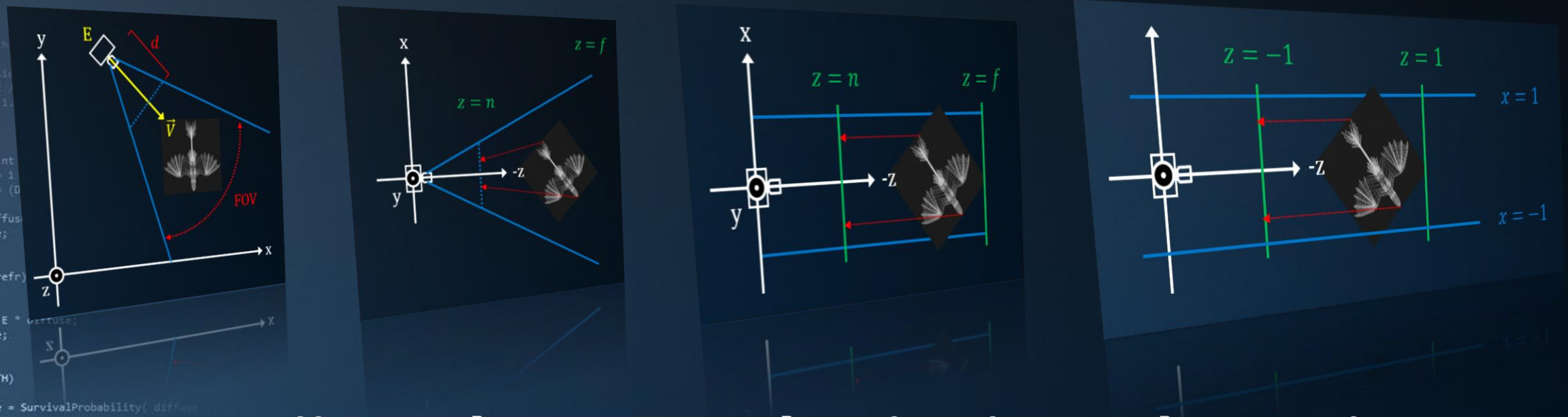


The world according to the camera:

Camera space

Perspective

Transformation Pipeline



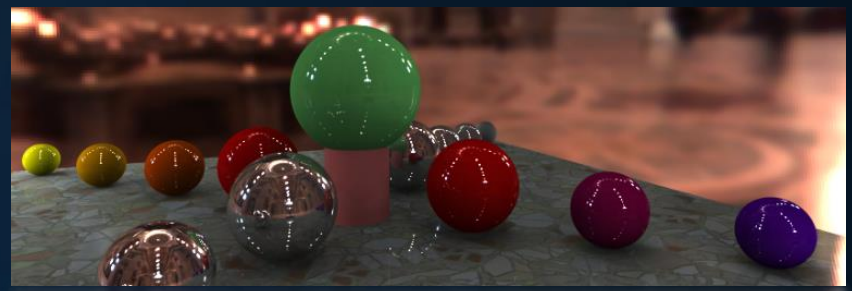
World space ➔ camera space ➔ orthographic view ➔ canonical view

$$I \times M_{camera} \times M_{ortho} \times M_{canonical}$$

These can be collapsed into a single 4×4 matrix.

Today’s Agenda:

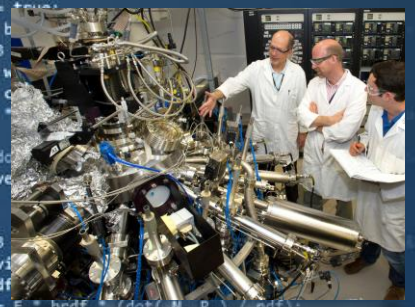
- Depth Sorting
- Clipping
- Visibility



Depth Sorting

Rendering – Functional overview

1. Transform:
   translating / rotating meshes
2. Project:
   calculating 2D screen positions
3. Rasterize:
   determining affected pixels
4. Shade:
   calculate color per affected pixel



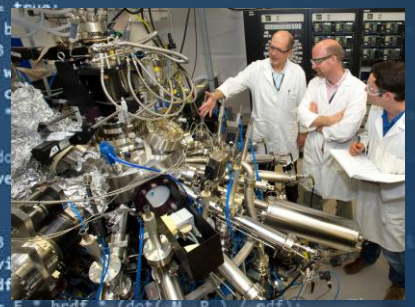
Depth Sorting

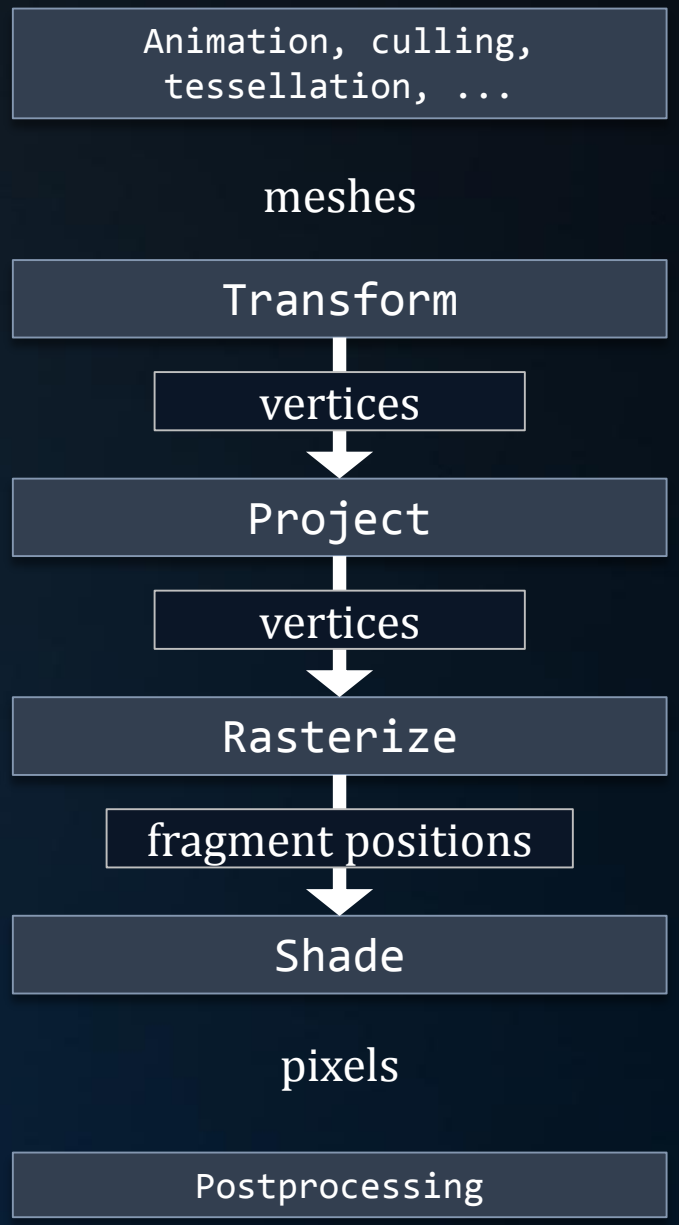
3. Rasterize:
   determining affected pixels

Questions:

- What is the screen space position of the fragment?
- Is that position actually on-screen?
- Is the fragment the nearest fragment for the affected pixel?

How do we efficiently determine *visibility* of a pixel?



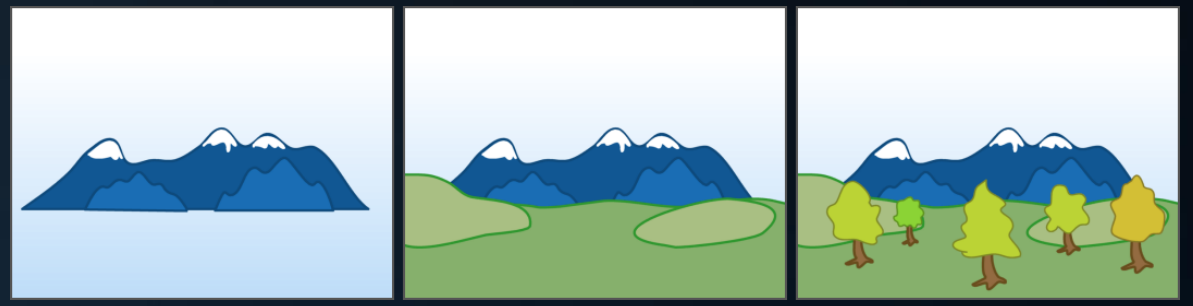



Part of the tree is off-screen
Too far away to draw
Tree requires little detail
City obscured by tree
Torso closer than ground
Tree between ground & sun

Depth Sorting

Old-skool depth sorting: Painter’s Algorithm

- Sort polygons by depth
- Based on polygon center
- Render depth-first

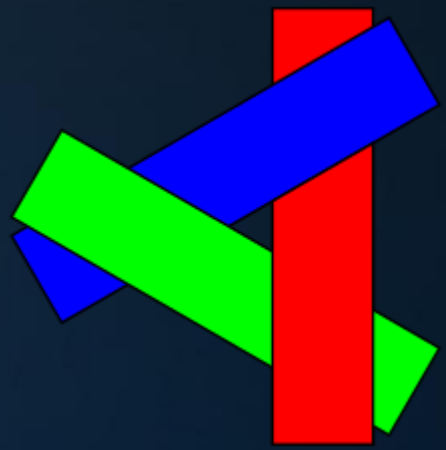


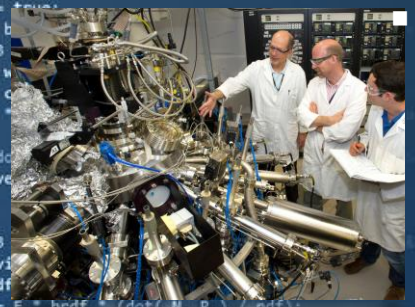
Advantage:

- Doesn’t require z-buffer

Problems:

- Cost of sorting
- Doesn’t handle all cases
- Overdraw

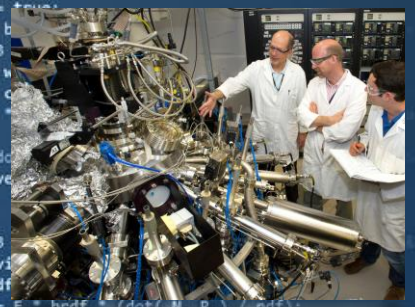


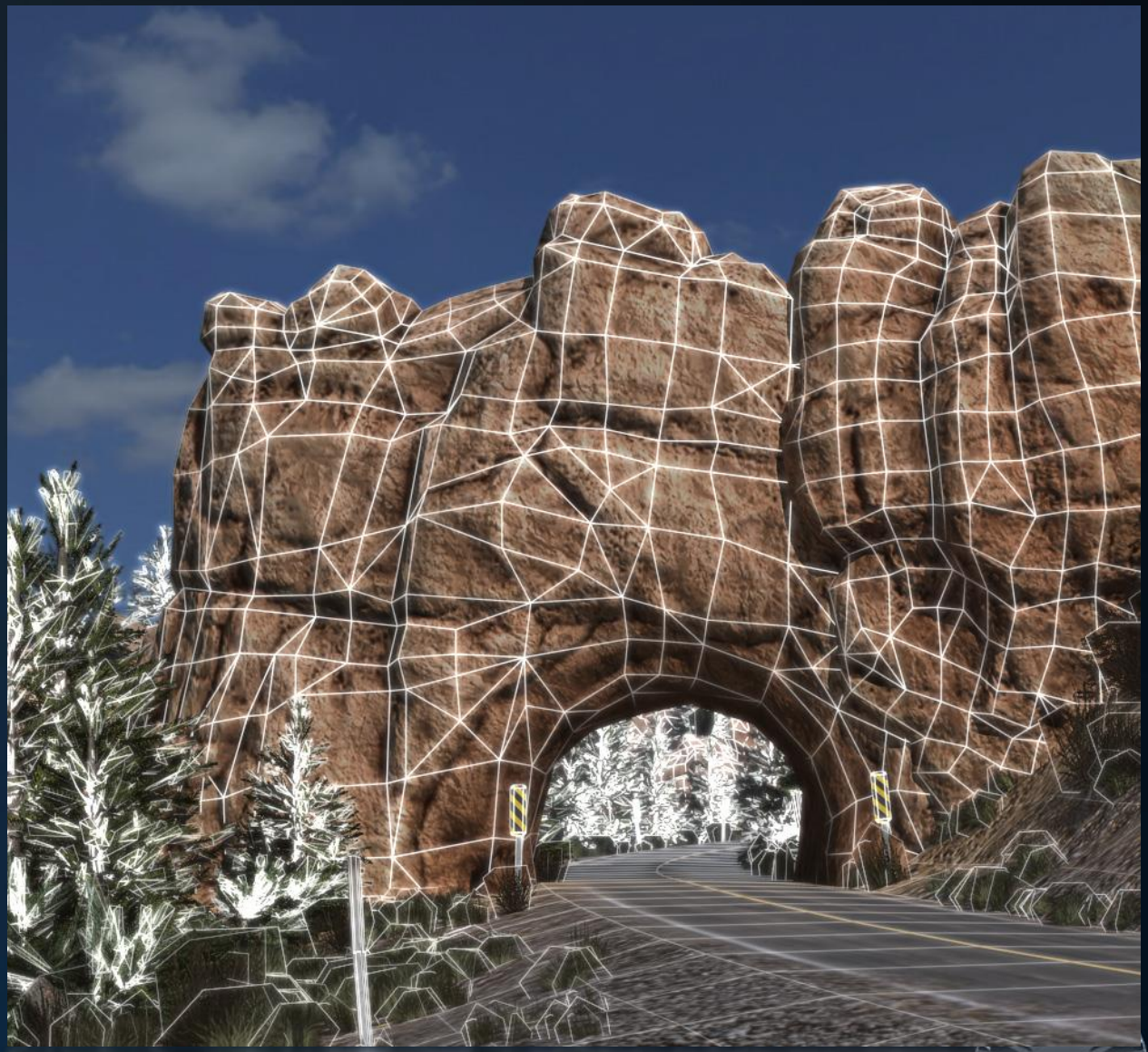


Depth Sorting

Overdraw:

Inefficiency caused by drawing multiple times to the same pixel.

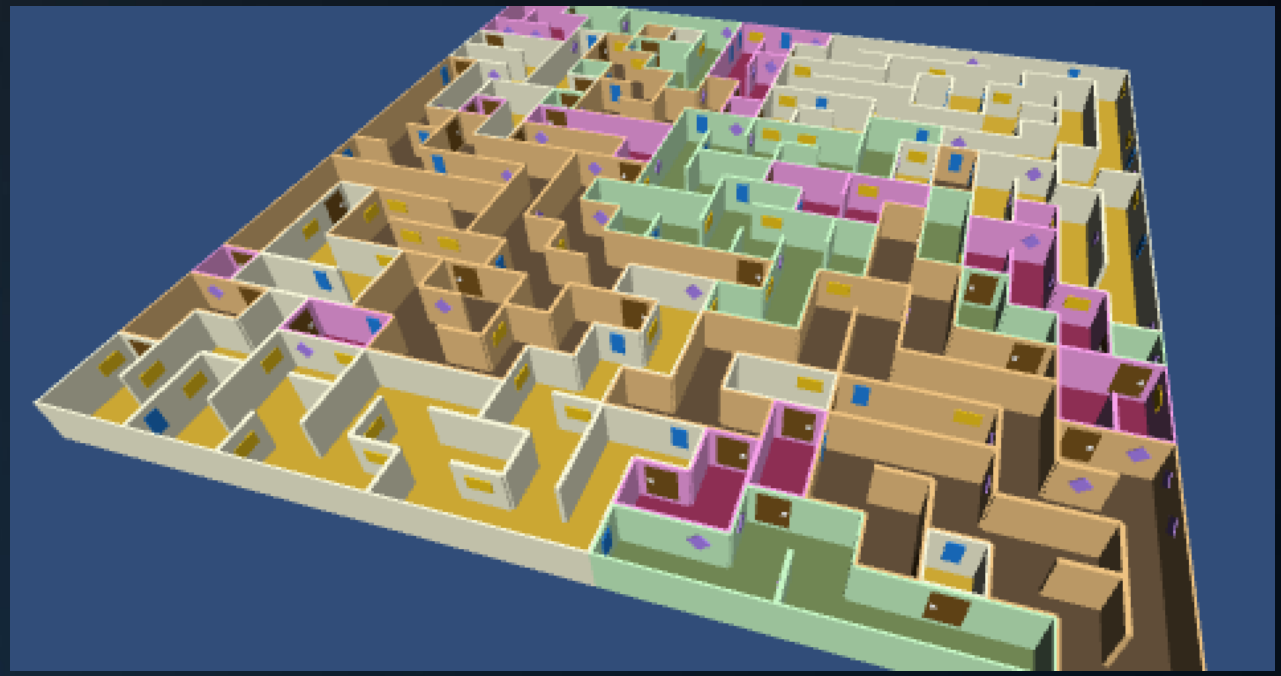


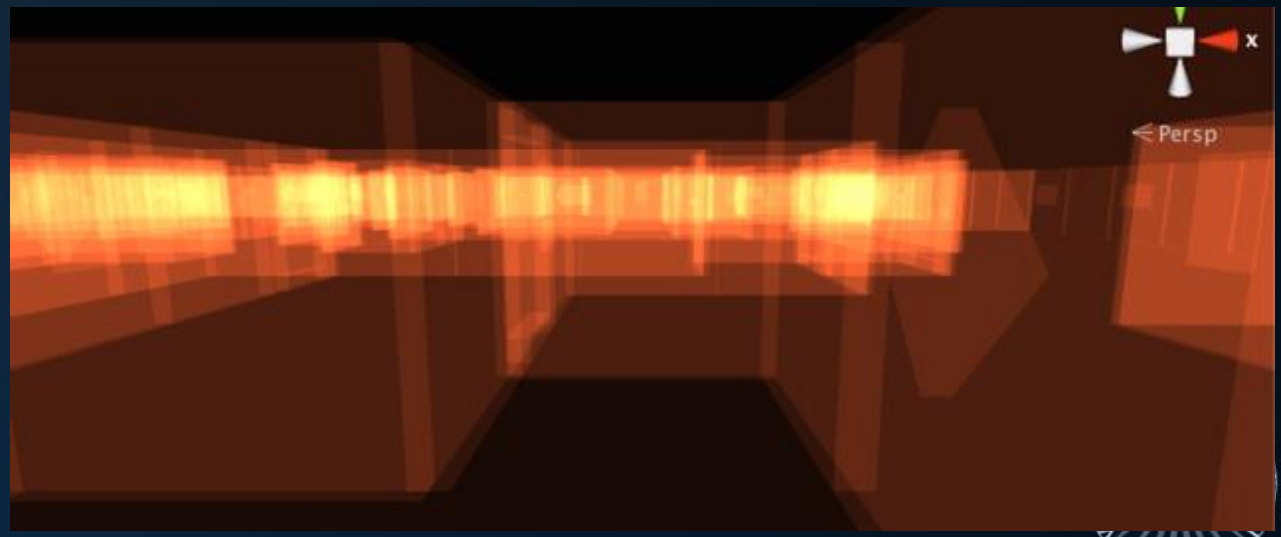


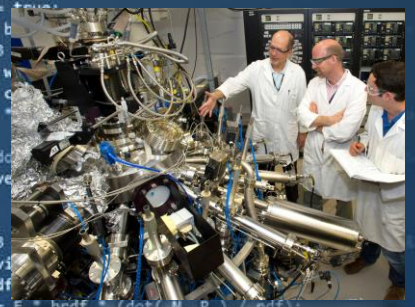
Depth Sorting

Overdraw:

Inefficiency caused by drawing multiple times to the same pixel.



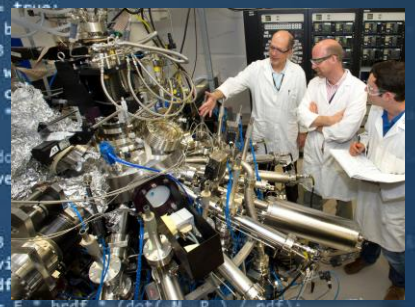




Depth Sorting

Overdraw:

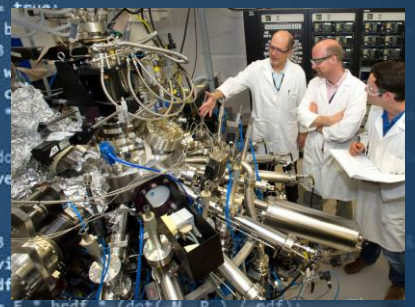
Inefficiency caused by drawing multiple times to the same pixel.

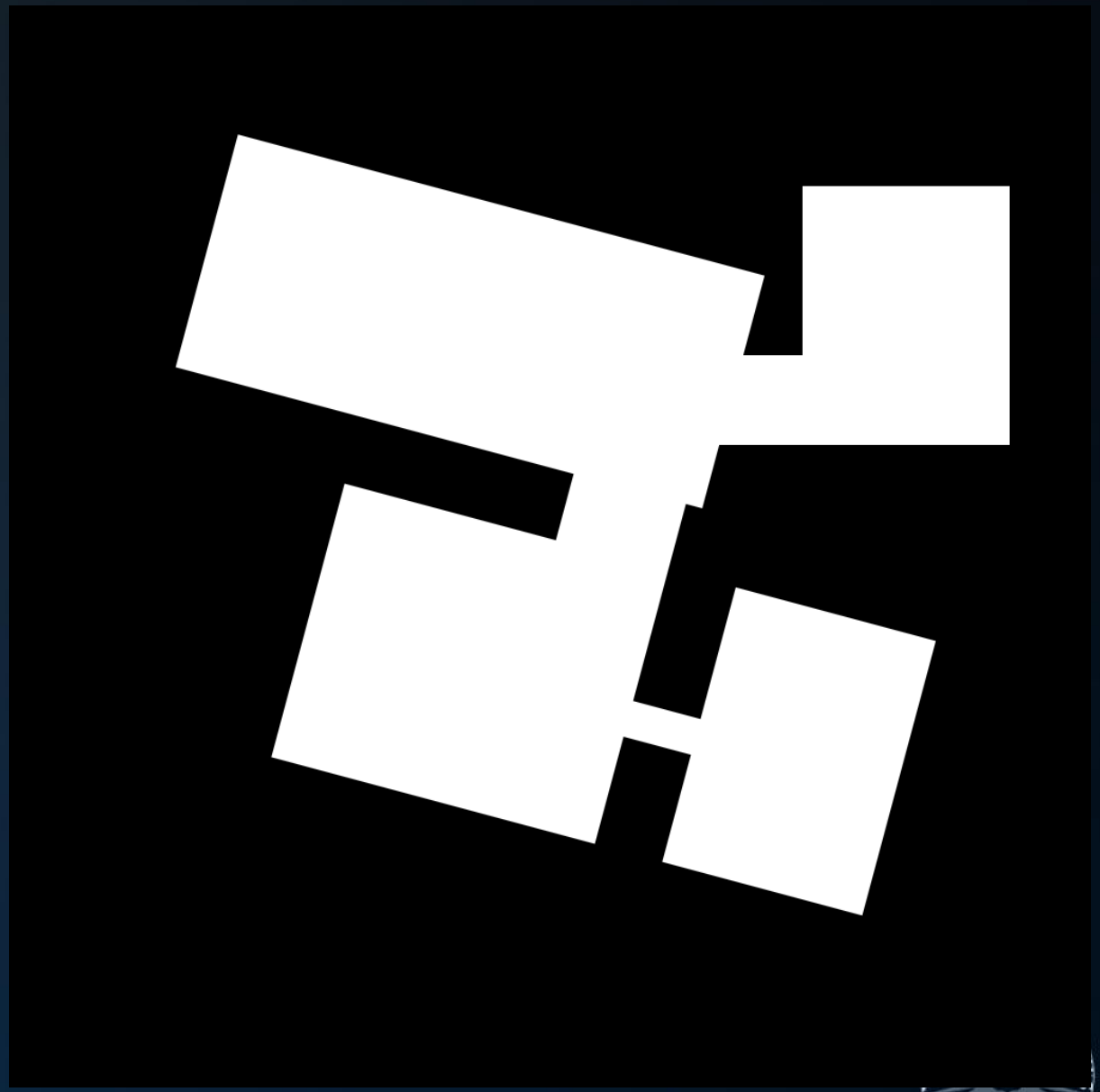


Depth Sorting

Correct order: BSP

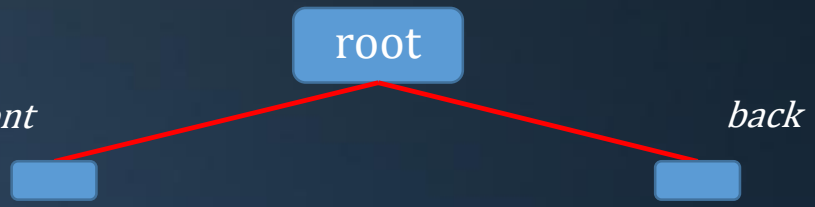
root

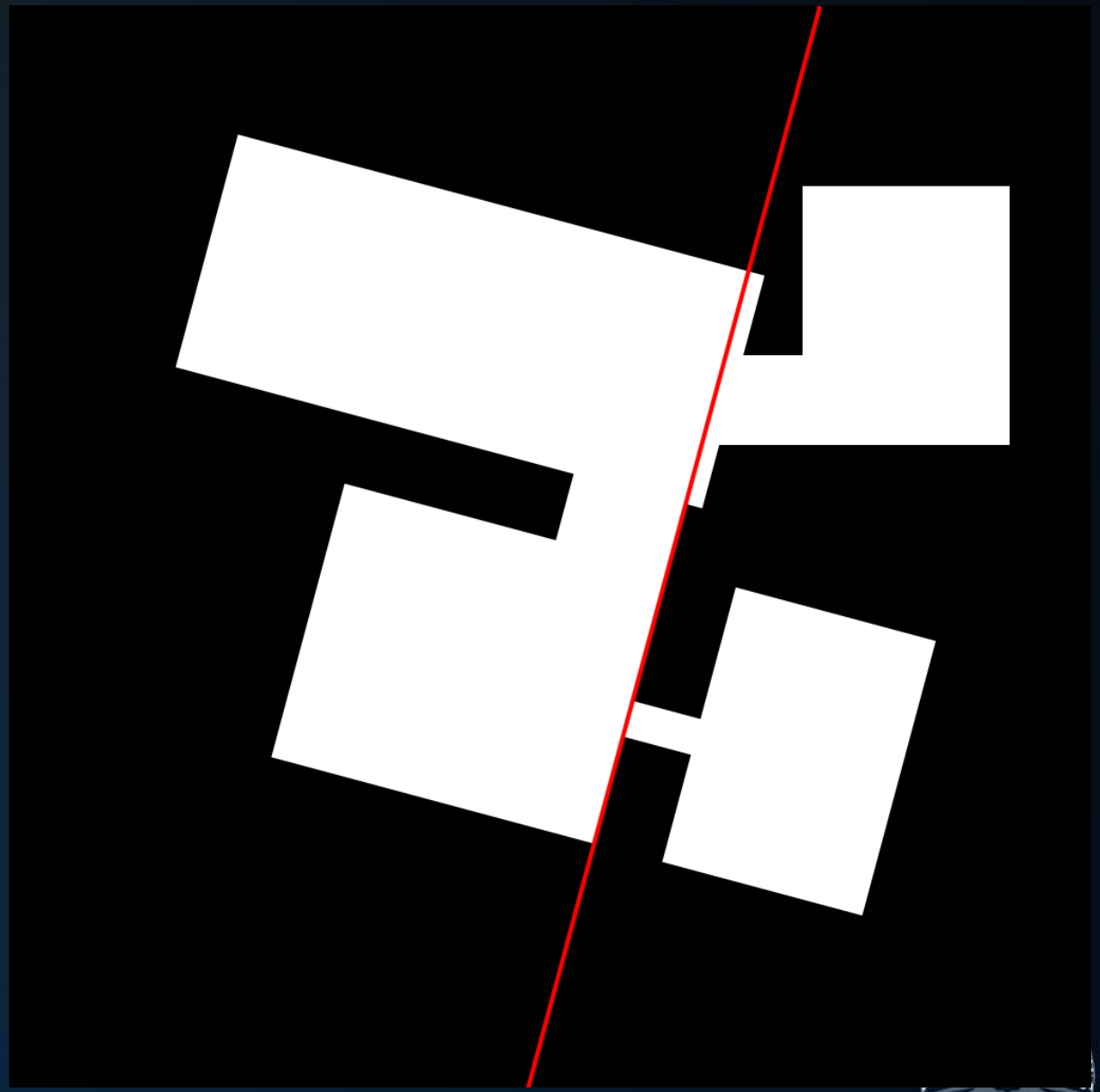


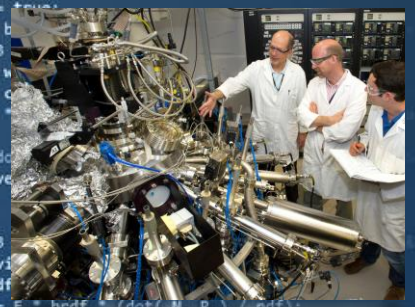


Depth Sorting

Correct order: BSP

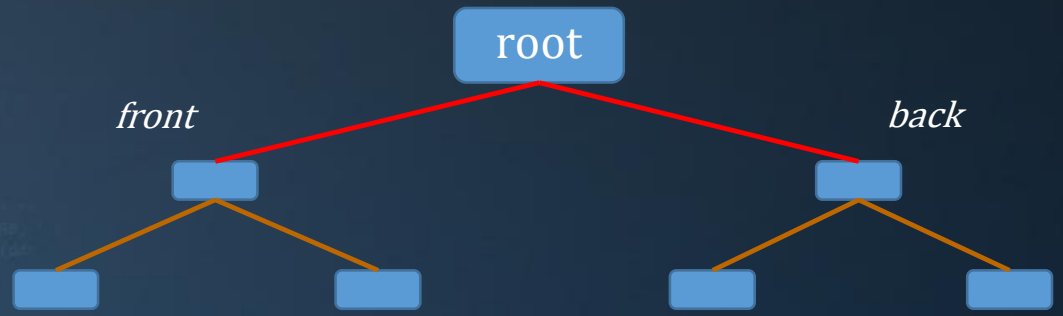


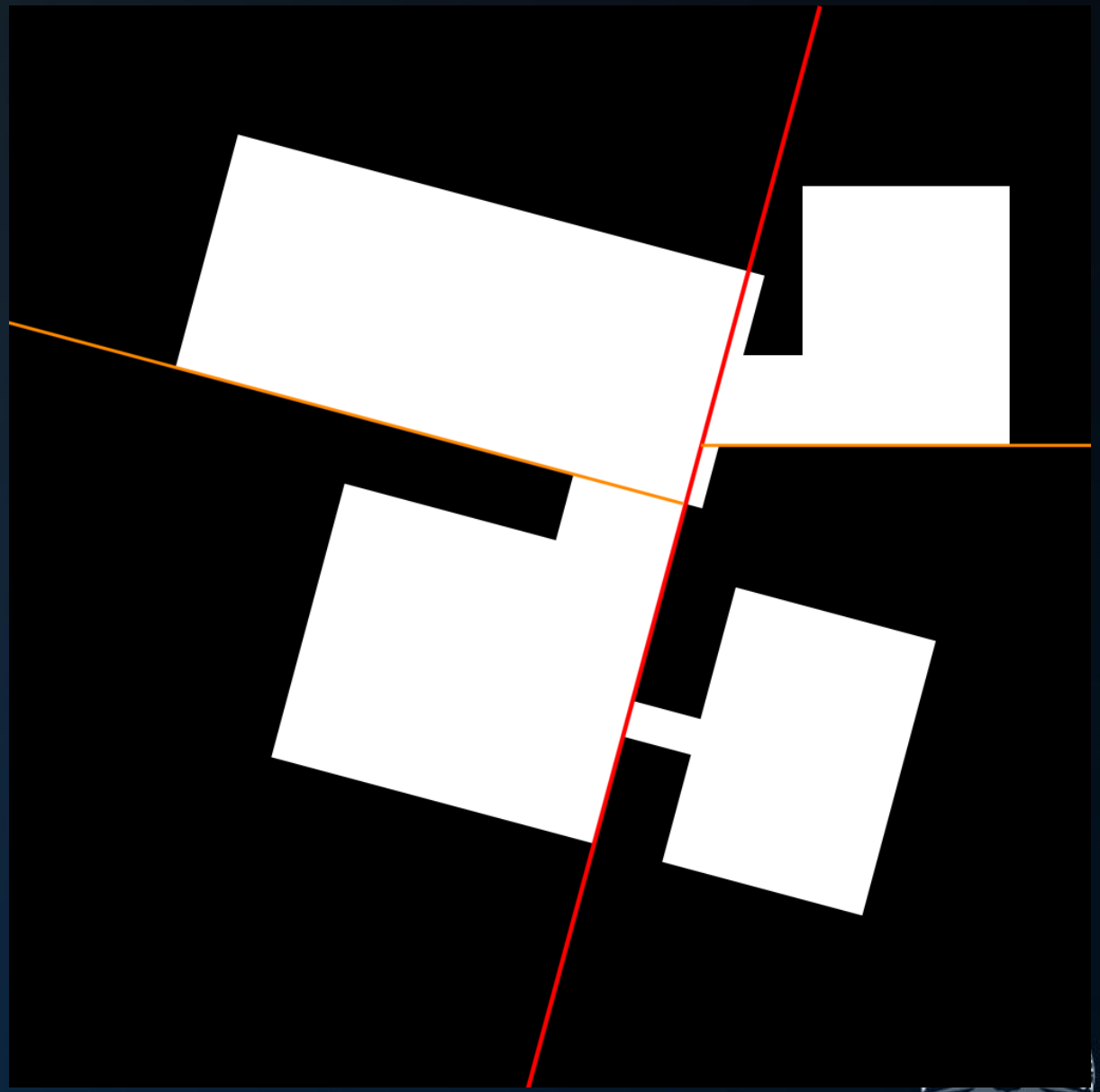


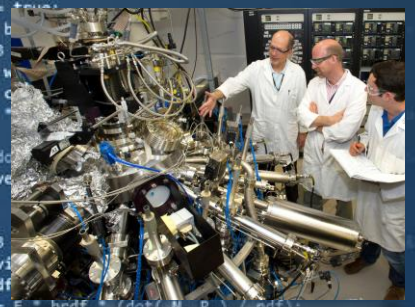


Depth Sorting

Correct order: BSP

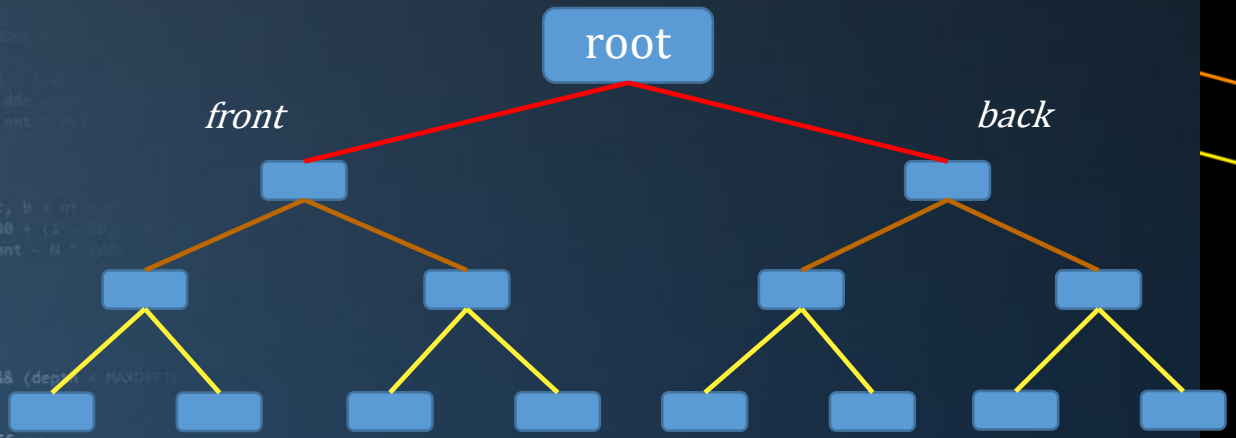


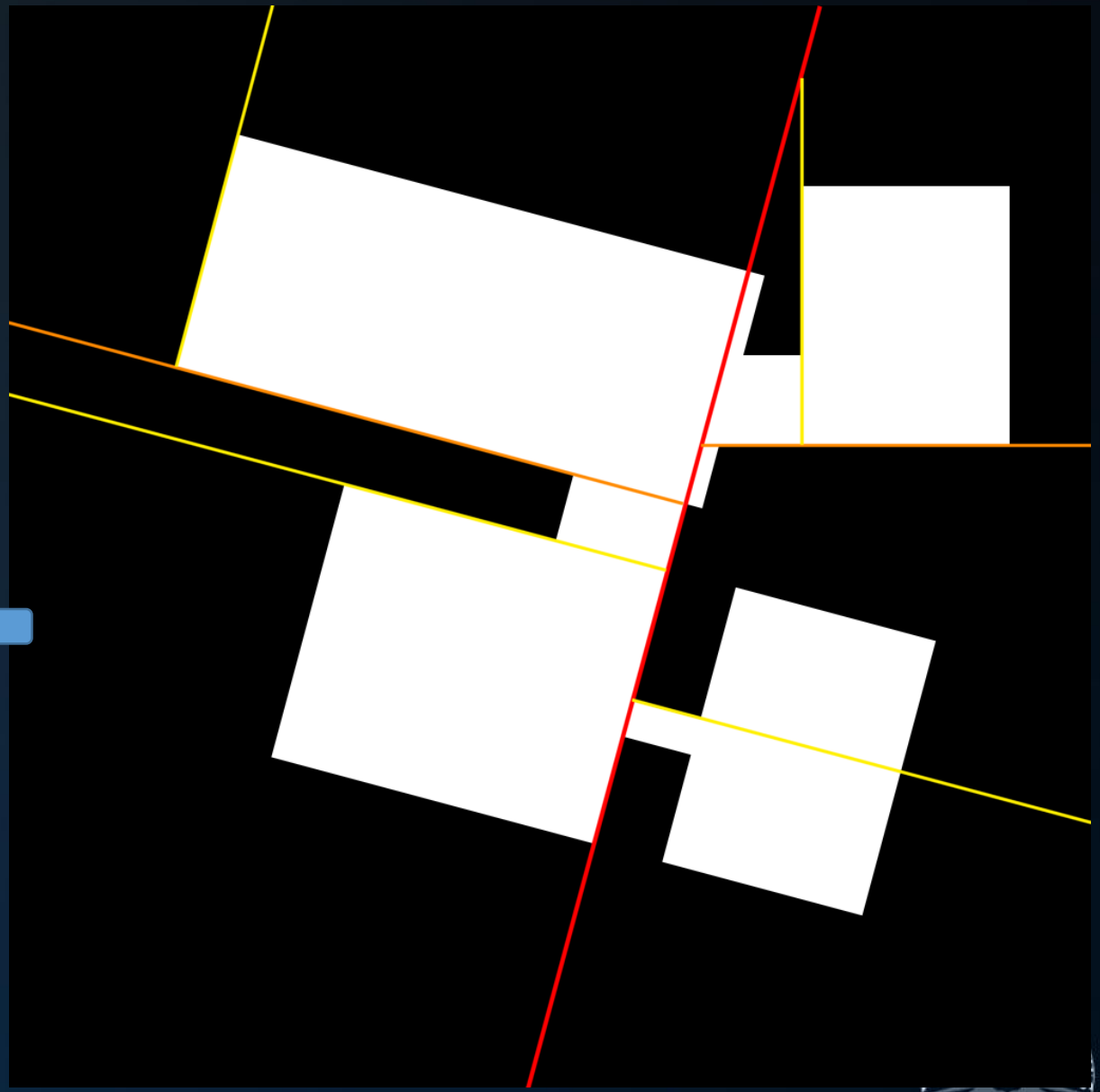


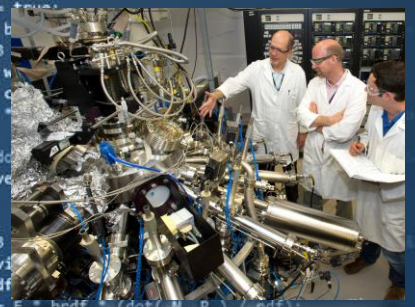


Depth Sorting

Correct order: BSP

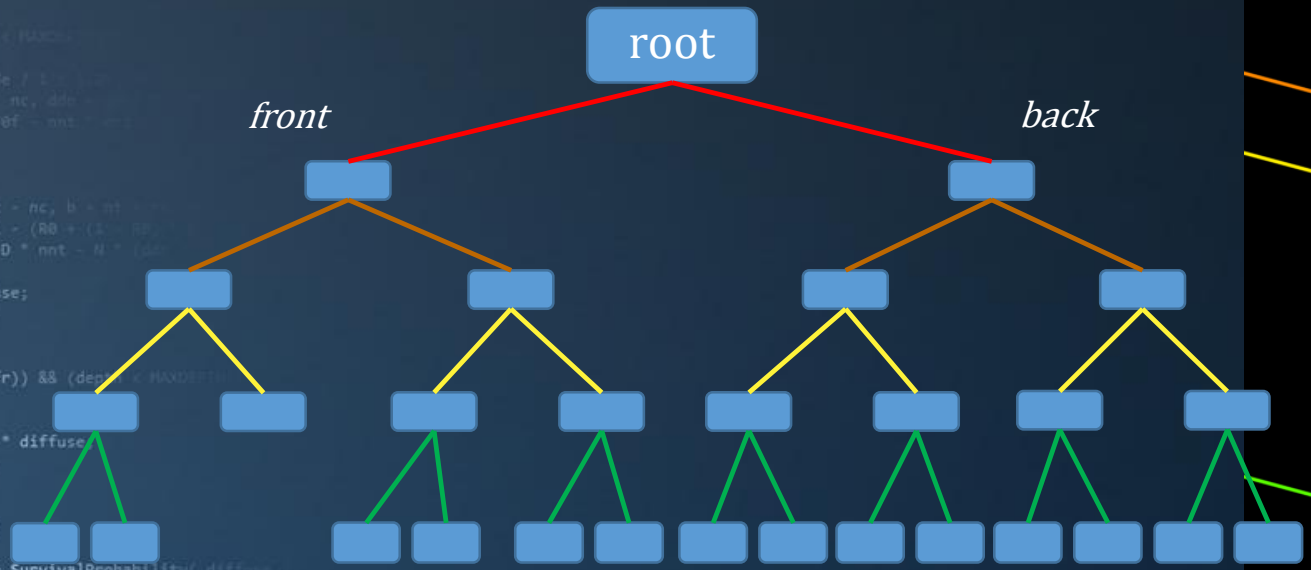


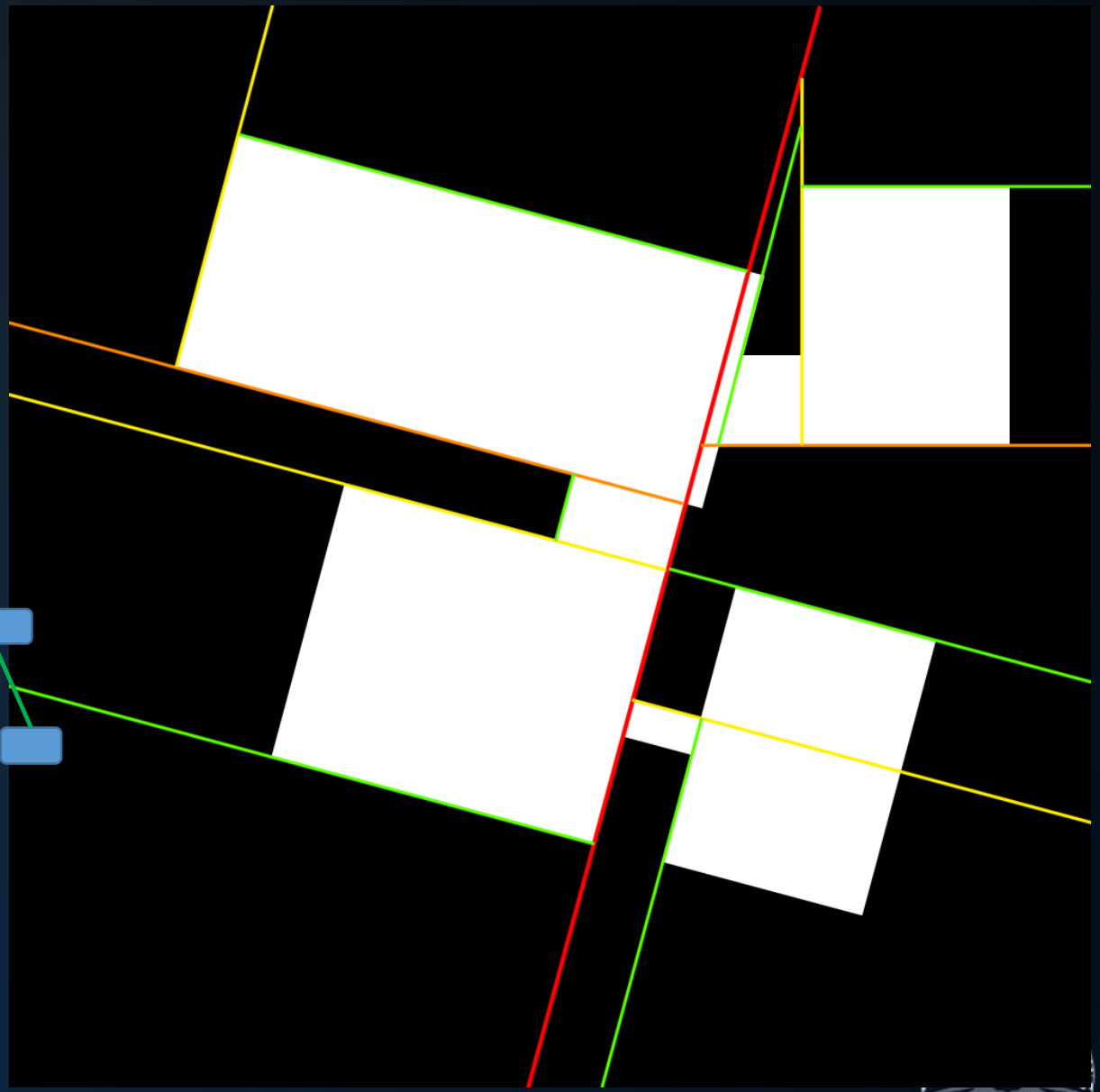


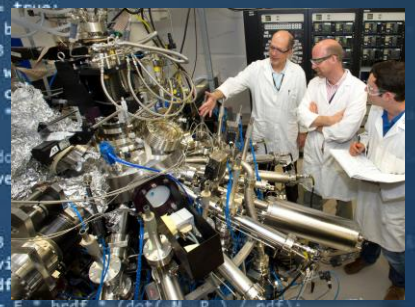


Depth Sorting

Correct order: BSP

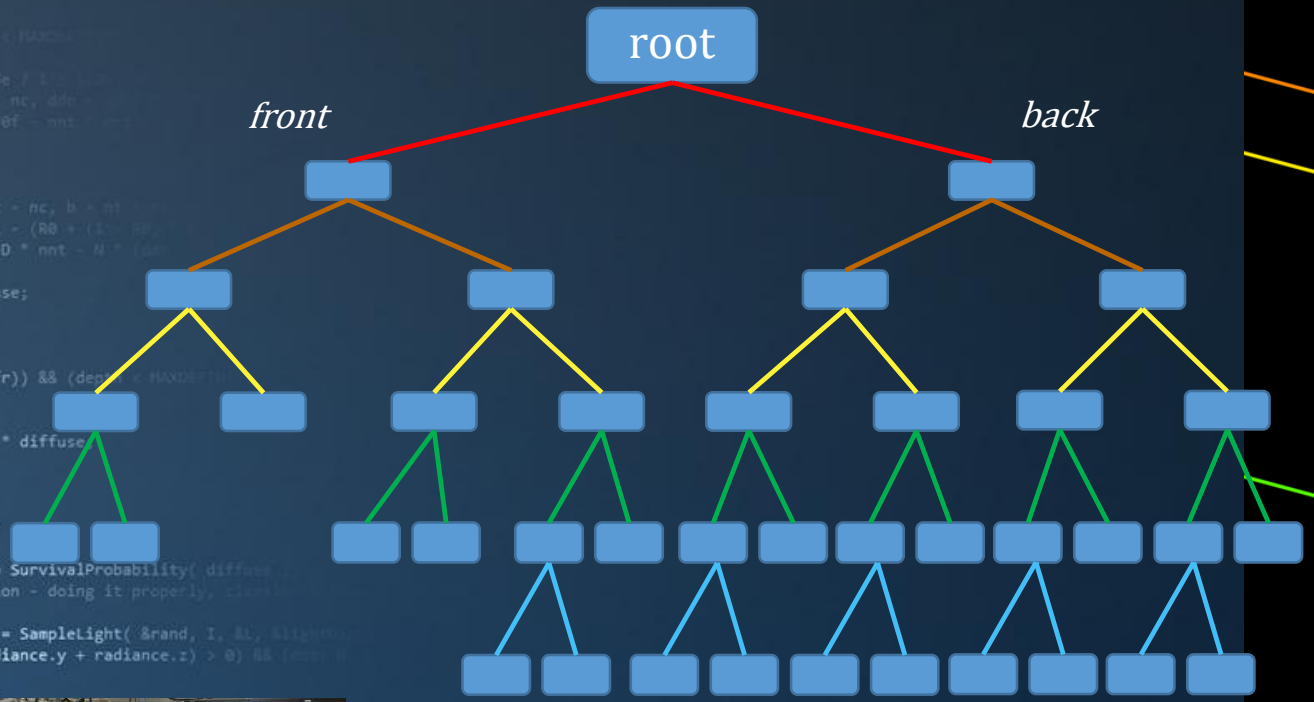


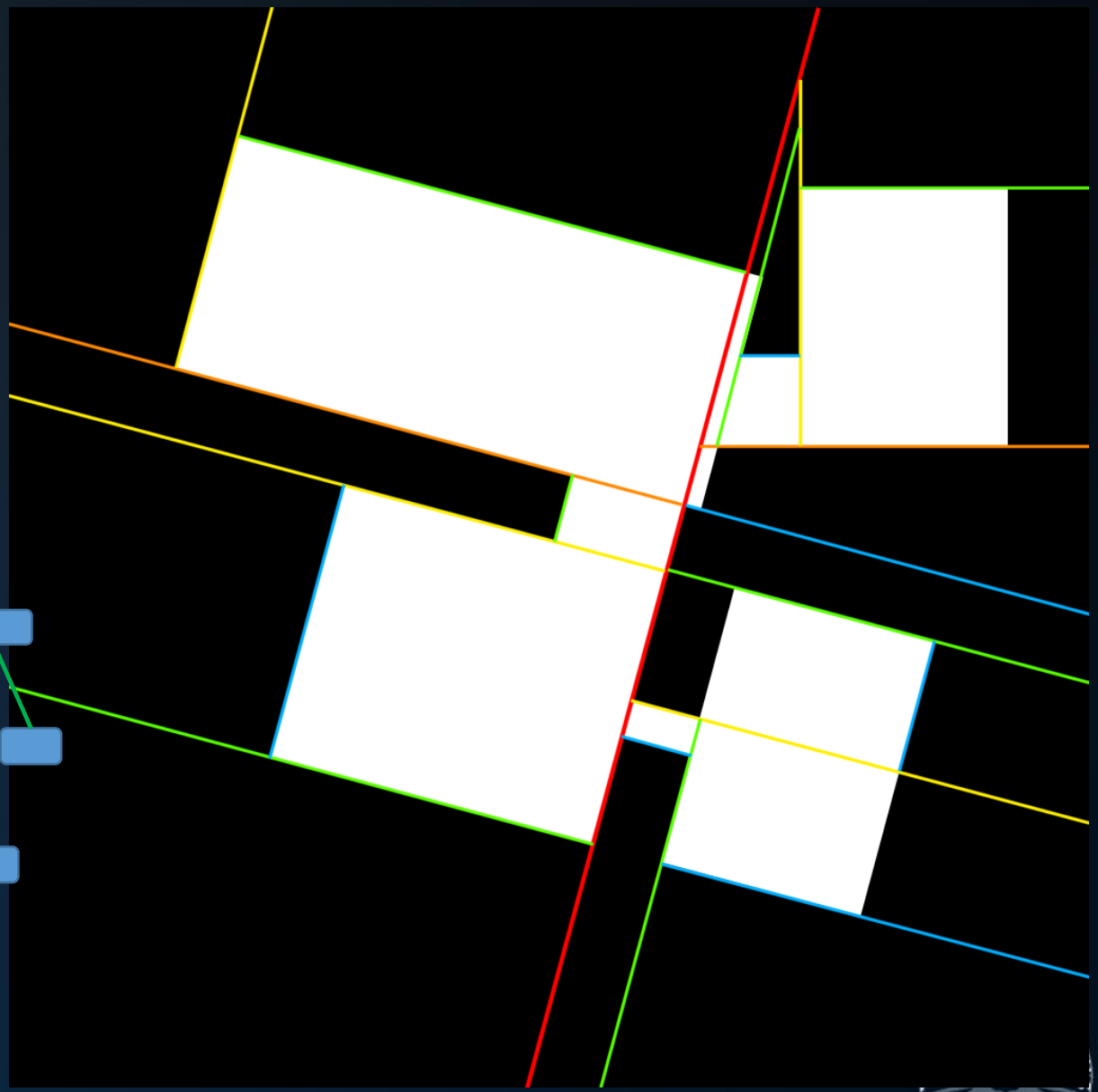


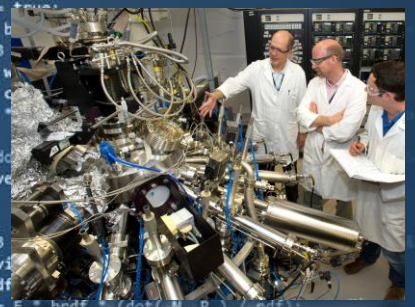


Depth Sorting

Correct order: BSP

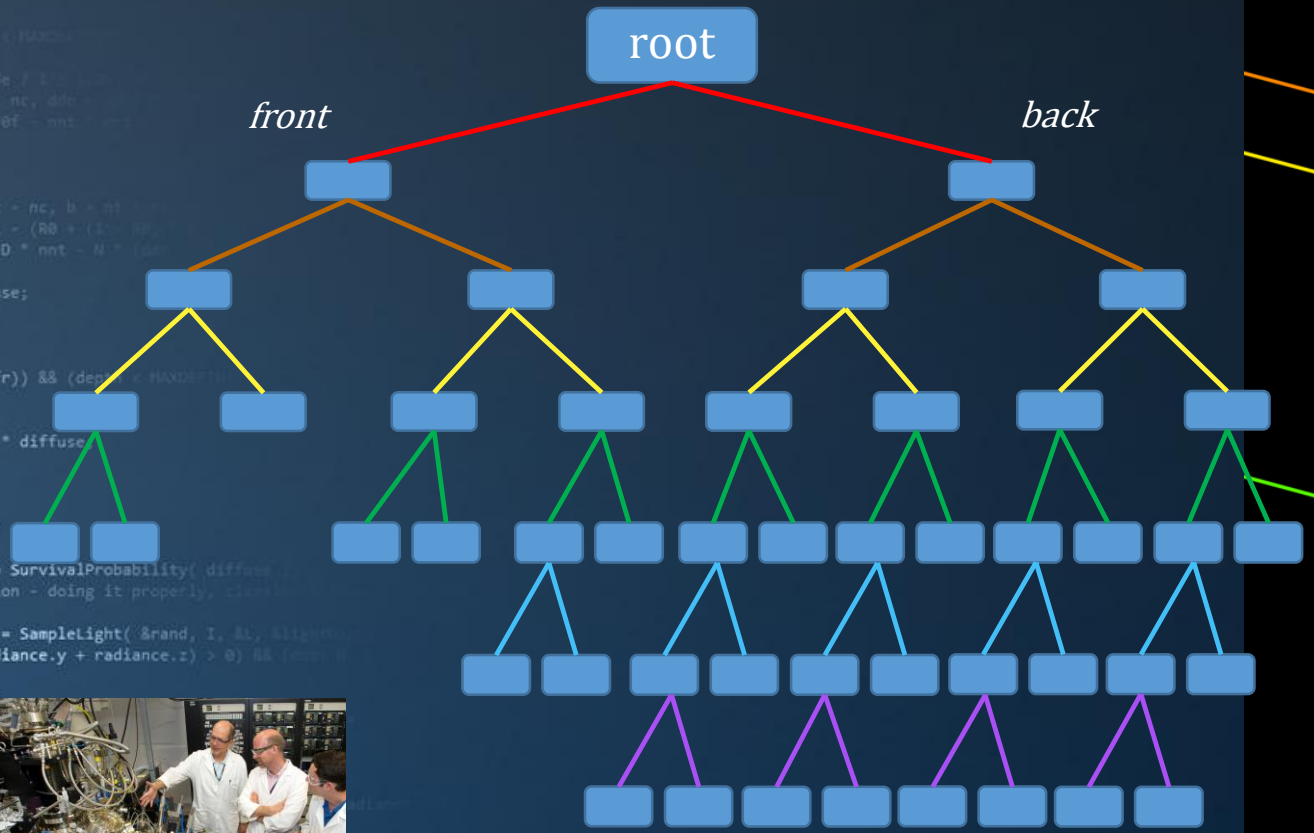


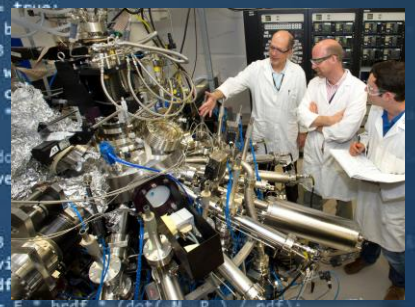


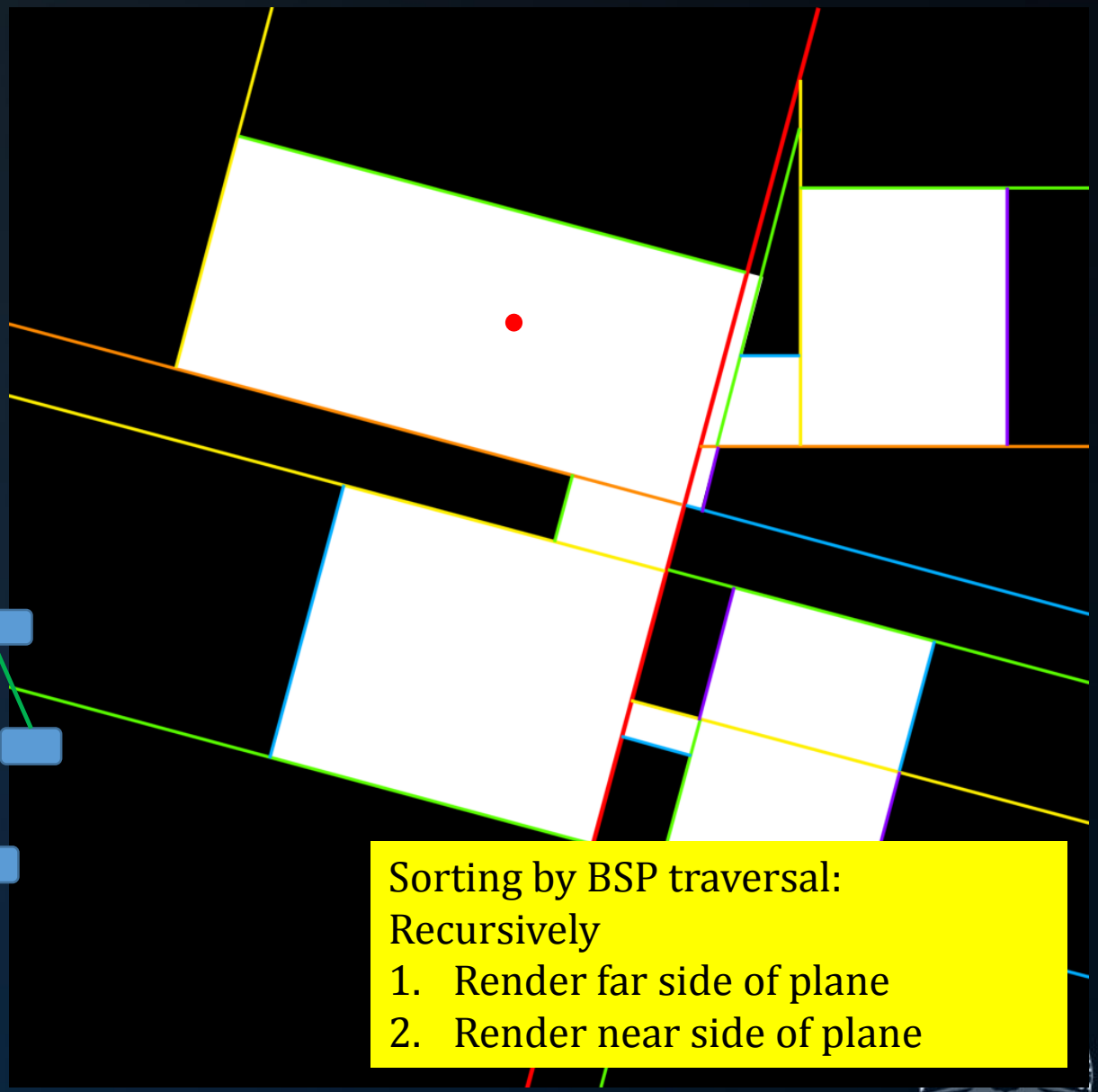


Depth Sorting

Correct order: BSP







Sorting by BSP traversal:
Recursively
1. Render far side of plane
2. Render near side of plane

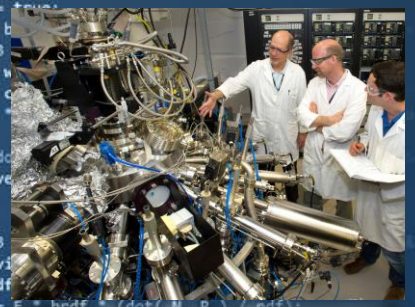
Depth Sorting

Draw order using a BSP:

- Guaranteed to be correct (hard cases result in polygon splits)
- No sorting required, just a tree traversal

But:

- Requires construction of BSP: not suitable for dynamic objects
- Does not eliminate overdraw



Depth Sorting

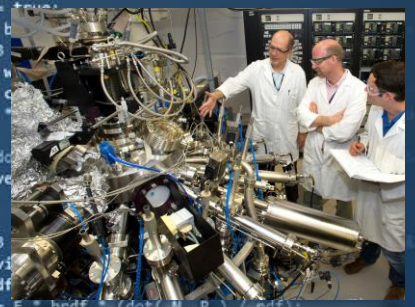
Z-buffer

A z-buffer stores, per screen pixel, a depth value.
The depth of each fragment is checked against this value:

- If the fragment is further away, it is discarded
- Otherwise, it is drawn, and the z-buffer is updated.

The z-buffer requires:

- An additional buffer
- Initialization of the buffer to z_{max}
- Interpolation of z over the triangle
- A z-buffer read and compare, and possibly a write.



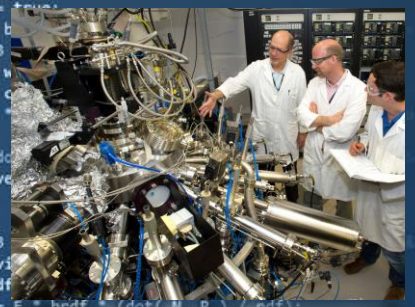
Depth Sorting

Z-buffer

What is the best representation for depth in a z-buffer?

1. Interpolated z (convenient, intuitive);
2. 1/z (or: $n + f - \frac{fn}{z}$) (more accurate nearby);
3. (int)((2^31-1)/z);
4. (uint)((2^32-1)/-z);
5. (uint)((2^32-1)/(-z + 1)).

Note: we use $z_{\text{int}} = \frac{(2^{32}-1)}{-z+1}$:
this way, any $z < 0$ will be in the range $z_{\text{adjusted}} = -z_{original} + 1 = 1..\infty$, therefore $1/z_{adjusted}$ will be in the range 0..1, and thus the integer value we will store uses the full range of $0..2^{32} - 1$.
Here, $z_{int} = 0$ represents $z_{original} = 0$, and $z_{int} = 2^{32} - 1$ represents $z_{original} = -\infty$.



Even more details:
https://developer.nvidia.com/content/depth-precision-visualized
http://outerra.blogspot.nl/2012/11/maximizing-depth-buffer-range-and.html

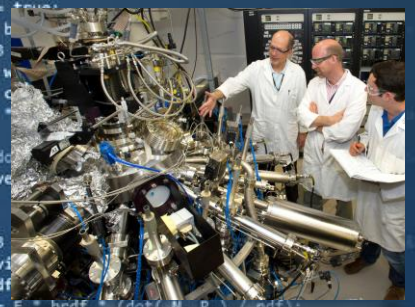
Depth Sorting

Z-buffer optimization

In the ideal case, the nearest fragment for a pixel is drawn first:

- This causes all subsequent fragments for the pixel to be discarded;
- This minimizes the number of writes to the frame buffer and z-buffer.

The ideal case can be approached by using Painter’s to ‘pre-sort’.

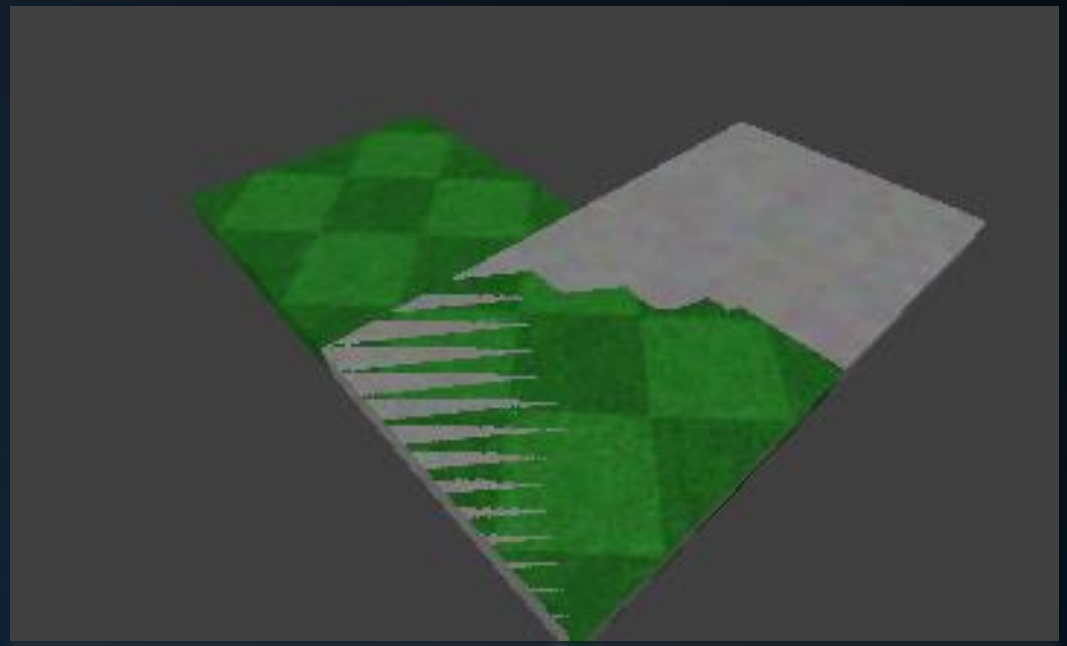


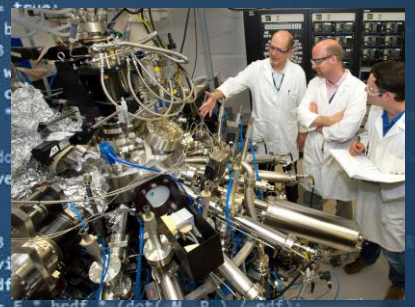
Depth Sorting

‘Z-fighting’:

Occurs when two polygons have almost identical z-values.

Floating point inaccuracies during interpolation will cause unpleasant patterns in the image.

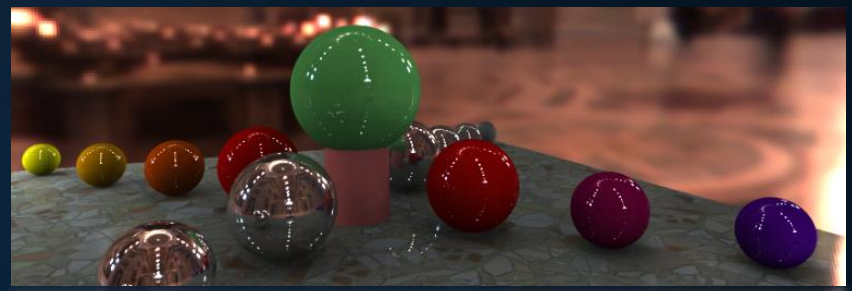





Part of the tree is off-screen
Stuff that is too far to draw
Tree requires little detail
√ City obscured by tree
√ Torso closer than ground
Tree between ground & sun

Today’s Agenda:

- Depth Sorting
- Clipping
- Visibility



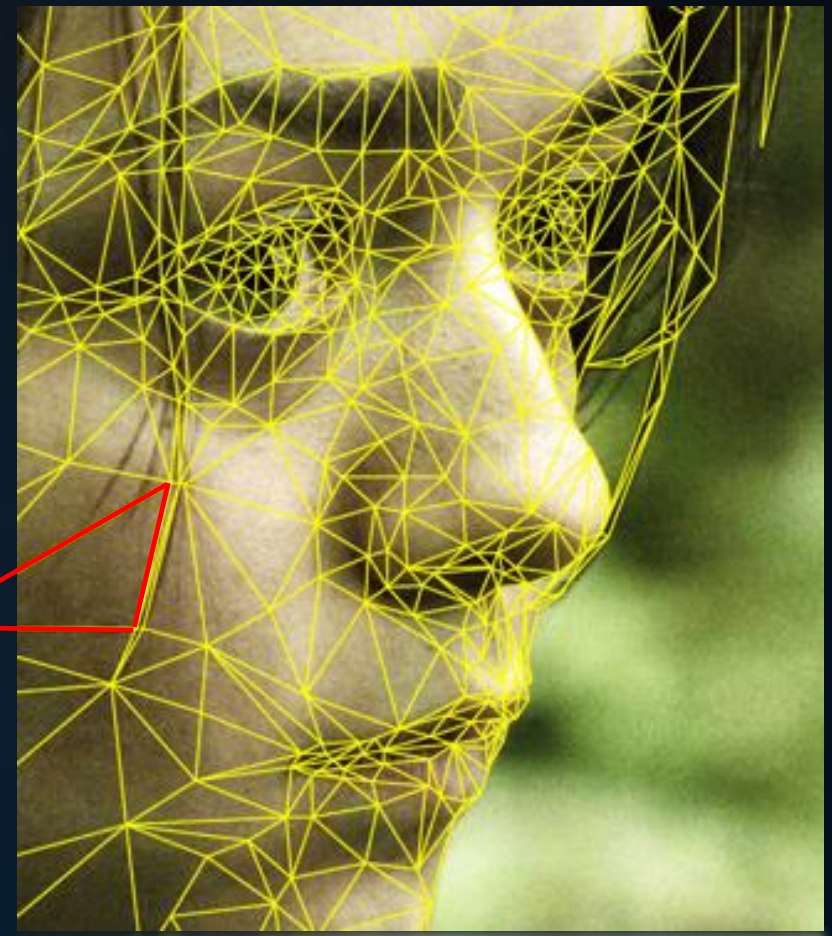
Clipping

Clipping

Many triangles are partially off-screen. This is handled by *clipping* them.

Sutherland-Hodgeman clipping:

Clip triangle against 1 plane at a time;
Emit *n-gon* (0, 3 or 4 vertices).

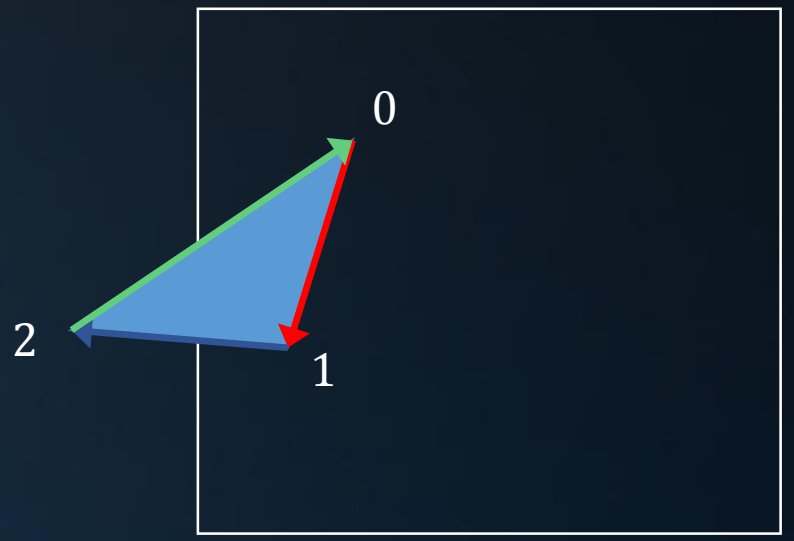


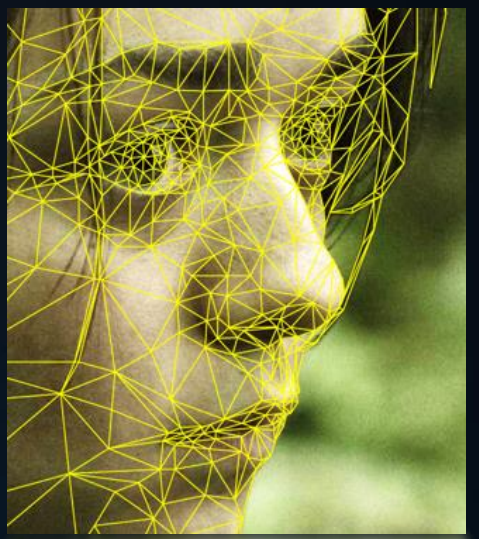
Clipping

Sutherland-Hodgeman

Input: list of vertices

Algorithm:





Per edge with vertices v_0 and v_1:

- If v_0 and v_1 are ‘in’, emit v_1
- If v_0 is ‘in’, but v_1 is ‘out’, emit C
- If v_0 is ‘out’, but v_1 is ‘in’, emit C and v_1

where C is the intersection point of the edge and the plane.

Output: list of vertices,
defining a convex n-gon.

in	out
Vertex 0	Vertex 1
Vertex 1	Intersection 1
Vertex 2	Intersection 2
	Vertex 0

Clipping

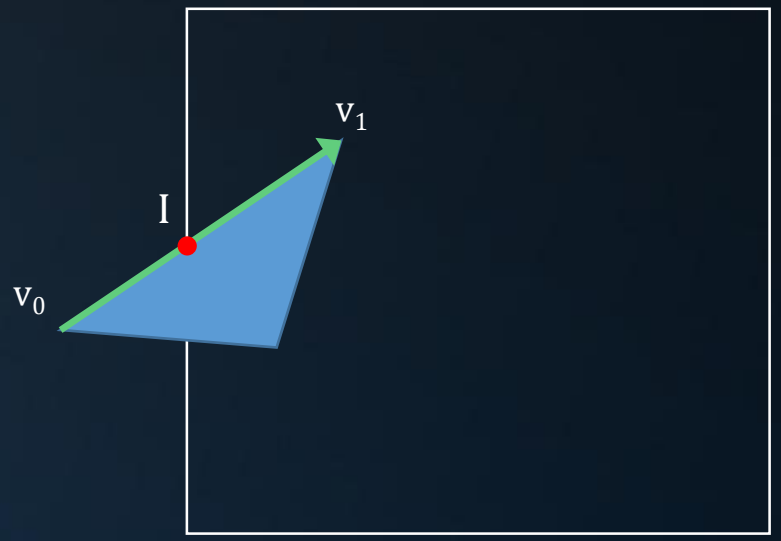
Sutherland-Hodgeman

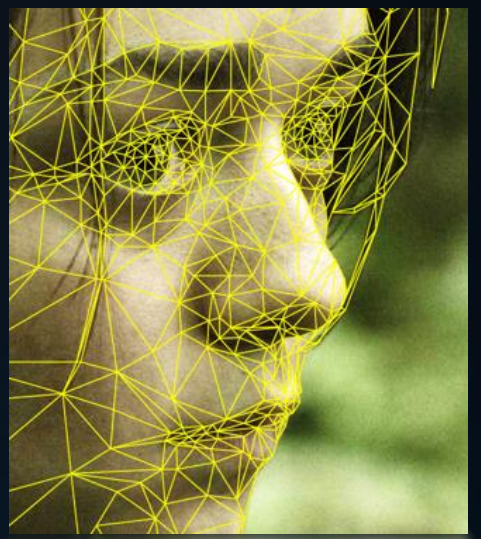
Calculating the intersections with plane $ax + by + cz + d = 0$:

$$dist_v = v \cdot \begin{pmatrix} a \\ b \\ c \end{pmatrix} + d$$

$$f = \frac{|dist_{v0}|}{|dist_{v0}| + |dist_{v1}|}$$

$$I = v_0 + f(v_1 - v_0)$$





After clipping, the input n-gon may have at most 1 extra vertex. We may have to triangulate it:

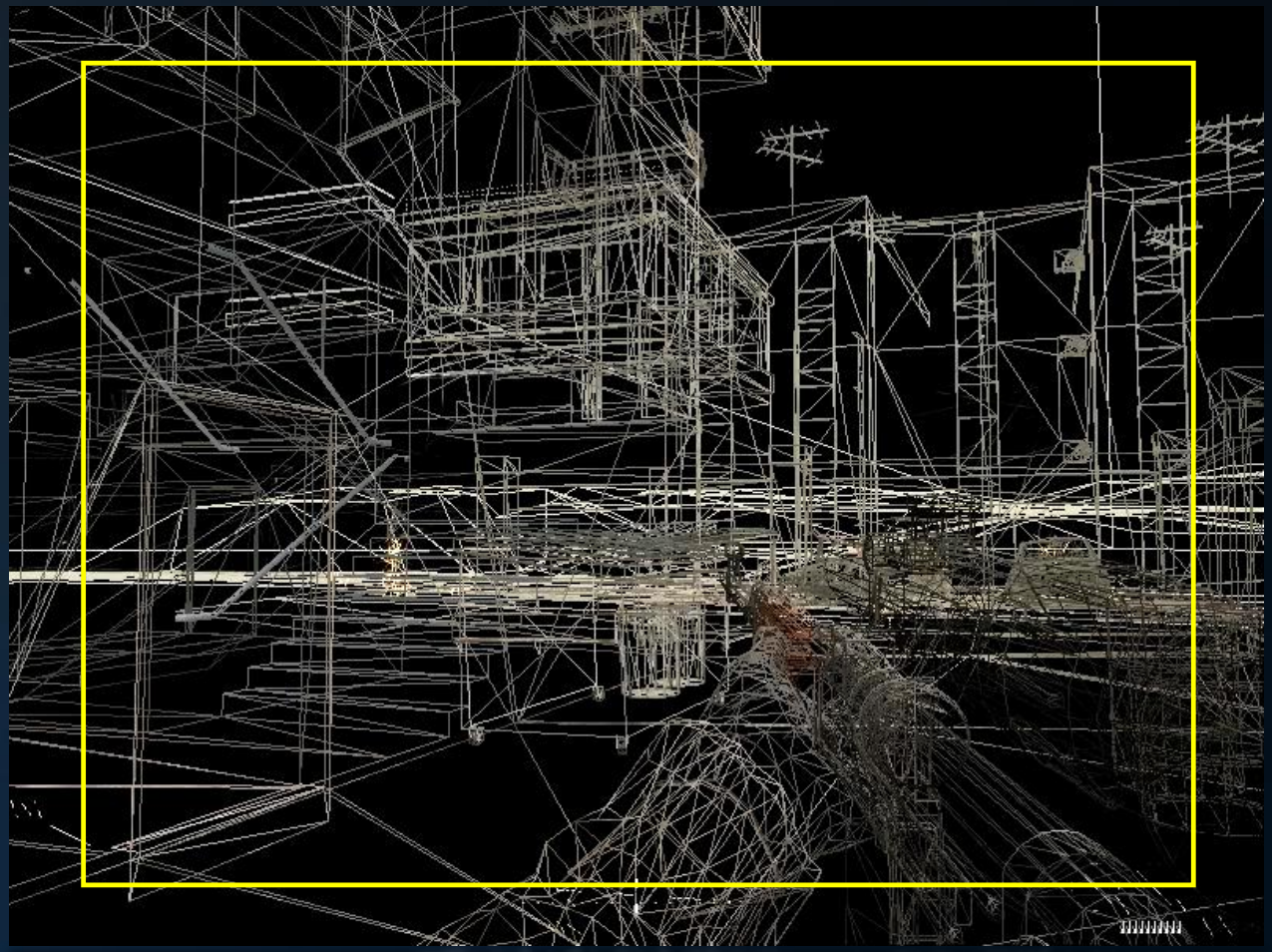
0,1,2,3,4 ➔ 0, 1, 2 + 0, 2, 3 + 0, 3, 4.

Clipping

Guard bands

To reduce the number of polygons that need clipping, some hardware uses *guard bands* : an invisible band of pixels outside the screen.

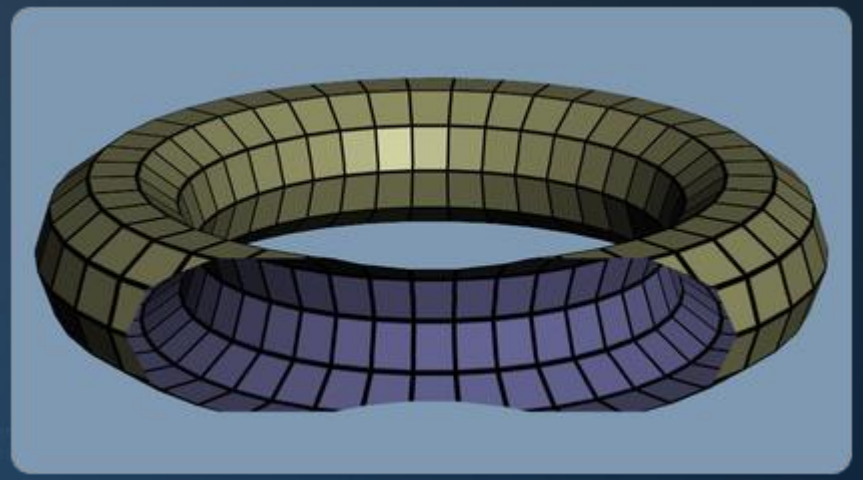
- Polygons outside the screen are discarded, even if they touch the guard band;
- Polygons partially inside, partially in the guard band are drawn without clipping;
- Polygons partially inside the screen, partially outside the guard band are clipped.

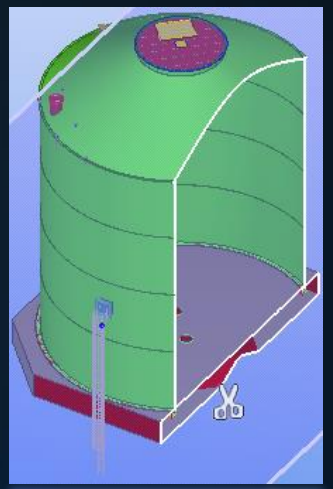


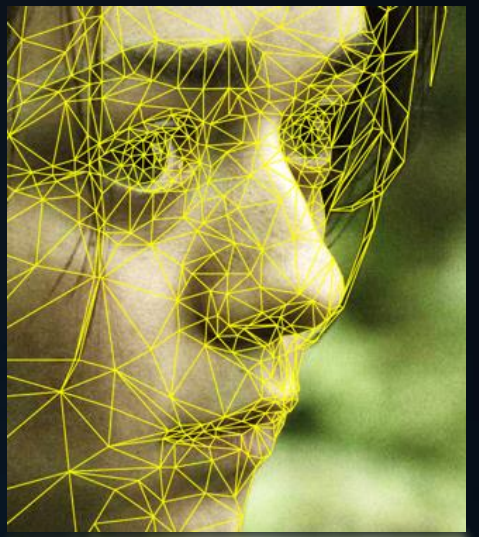
Clipping

Sutherland-Hodgeman

Clipping can be done against arbitrary planes.

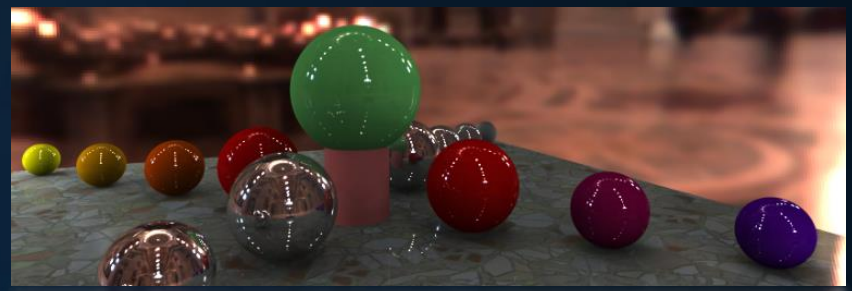






Today’s Agenda:

- Depth Sorting
- Clipping
- Visibility




√ Part of the tree is off-screen
Stuff that is too far to draw
Tree requires little detail
√ City obscured by tree
√ Torso closer than ground
Tree between ground & sun

Visibility

Only rendering what’s visible:

“Performance should be determined by visible geometry, not overall world size.”

- Do not render geometry outside the view frustum
- Better: do not *process* geometry outside frustum
- Do not render occluded geometry
- Do not render anything more detailed than strictly necessary

Visibility

Culling

Observation:
50% of the faces of a cube are not visible.

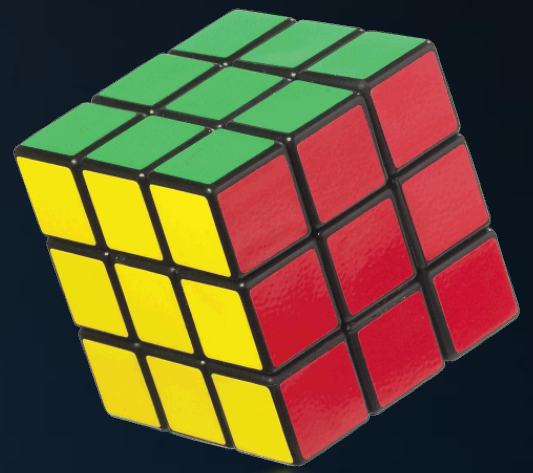
On average, this is true for all meshes.

Culling ‘backfaces’:

Triangle: $ax + by + cz + d = 0$
Camera: (x, y, z)
Visible: fill in camera position in plane equation.

$ax + by + cz + d > 0$: *visible*.

Cost: 1 dot product per triangle.



Visibility

Culling

Observation:
If the *bounding sphere* of a mesh is outside the view frustum, the mesh is not visible.

But also:
If the *bounding sphere* of a mesh intersects the view frustum, the mesh may be not visible.

View frustum culling is typically a *conservative test:* we sacrifice accuracy for efficiency.

Cost: 1 dot product per mesh.

Visibility

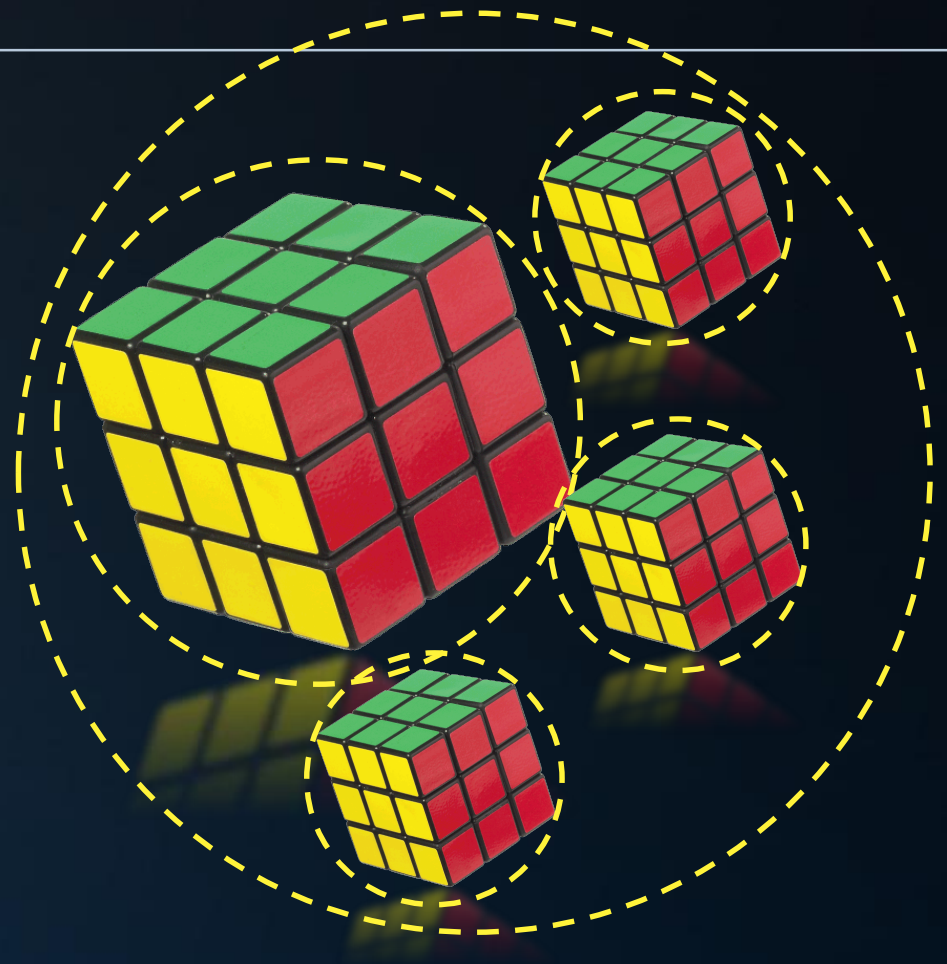
Culling

Observation:
If the *bounding sphere* over a group of bounding spheres is outside the view frustum, a group of meshes is invisible.

We can store a bounding volume hierarchy in the scene graph:

- Leaf nodes store the bounds of the meshes they represent;
- Interior nodes store the bounds over their child nodes.

Cost: 1 dot product per scene graph subtree.

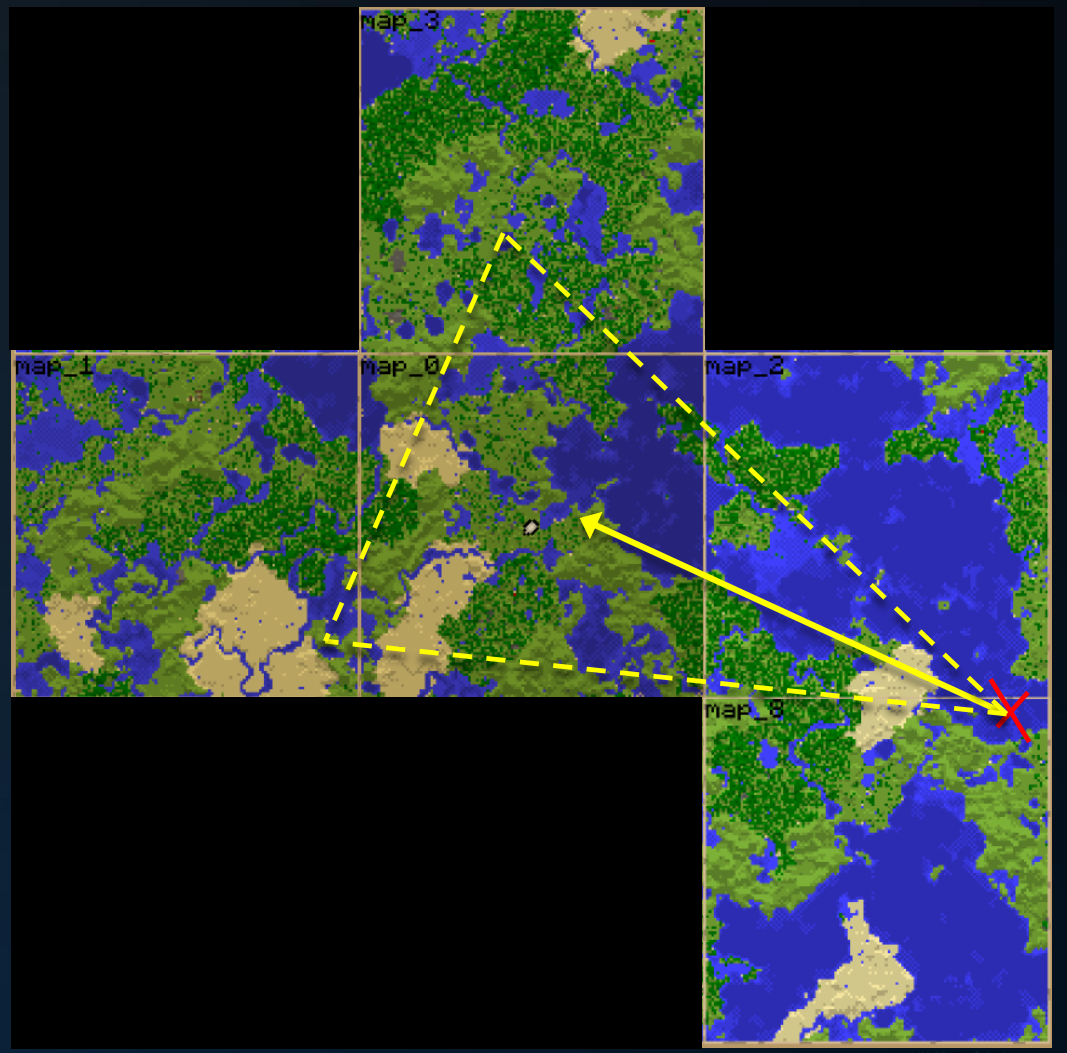


Visibility

Culling

Observation:
If a grid cell is outside the view frustum, the contents of that grid cell are not visible.

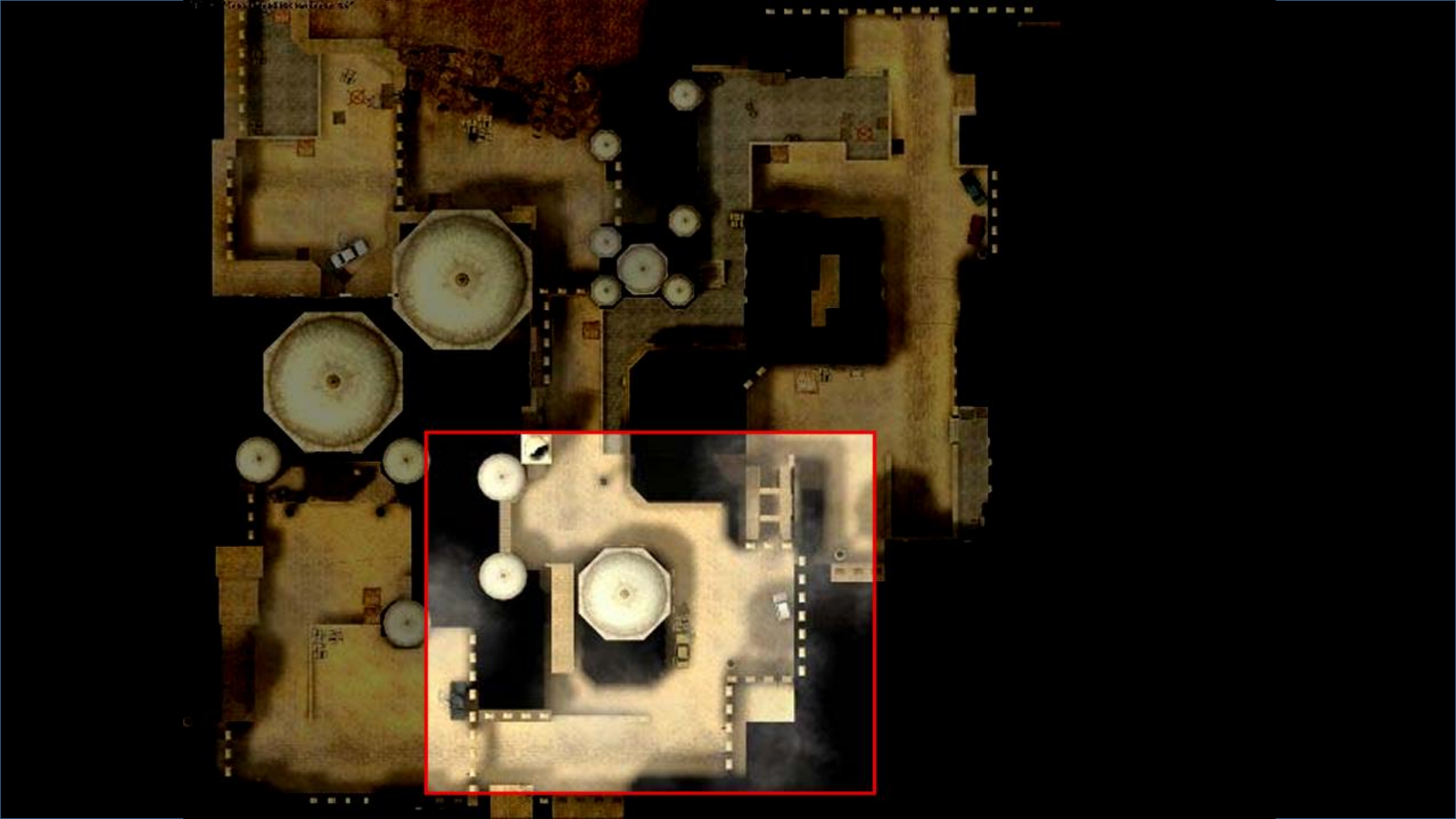
Cost: 0 for out-of-range grid cells.

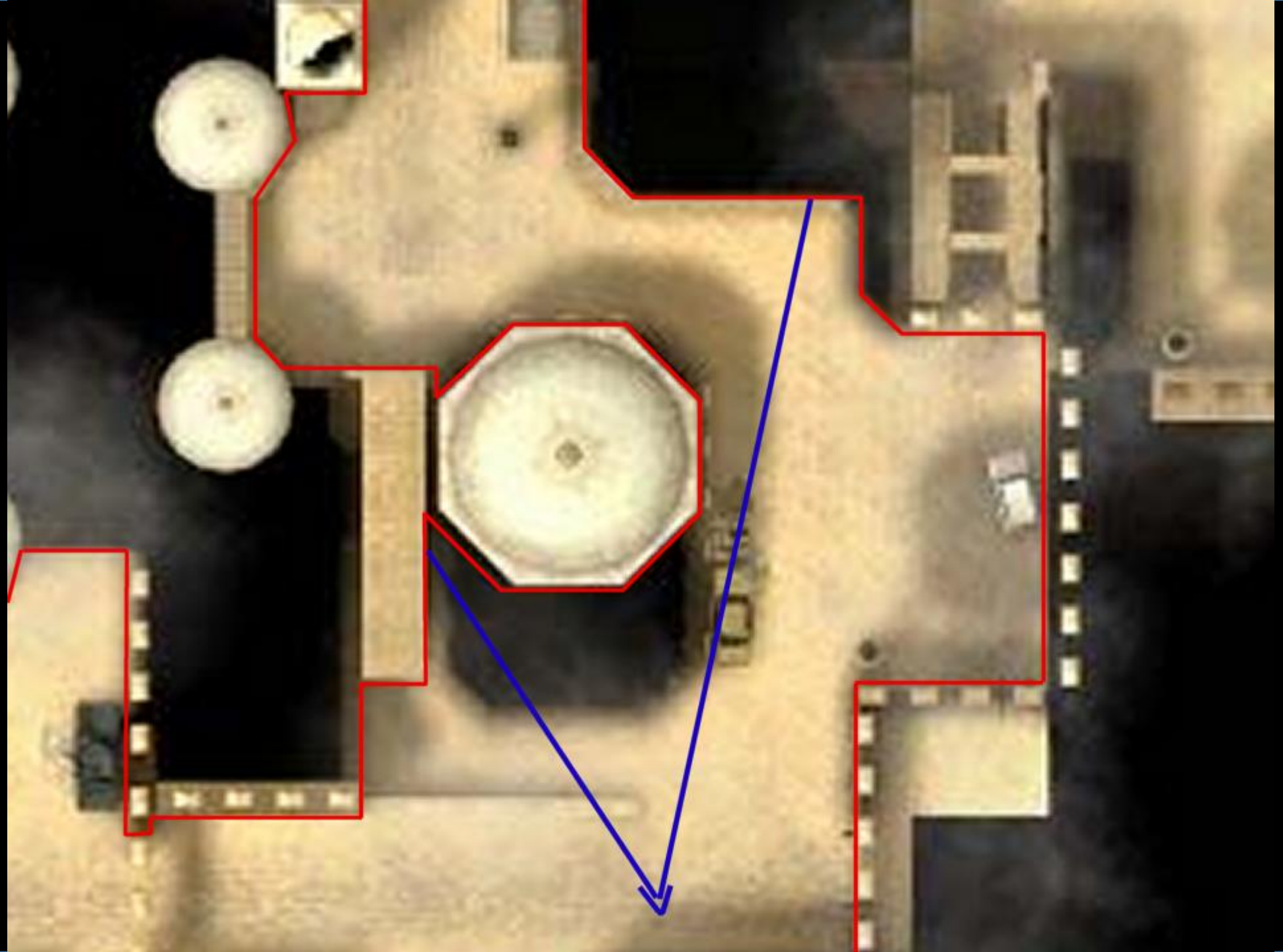


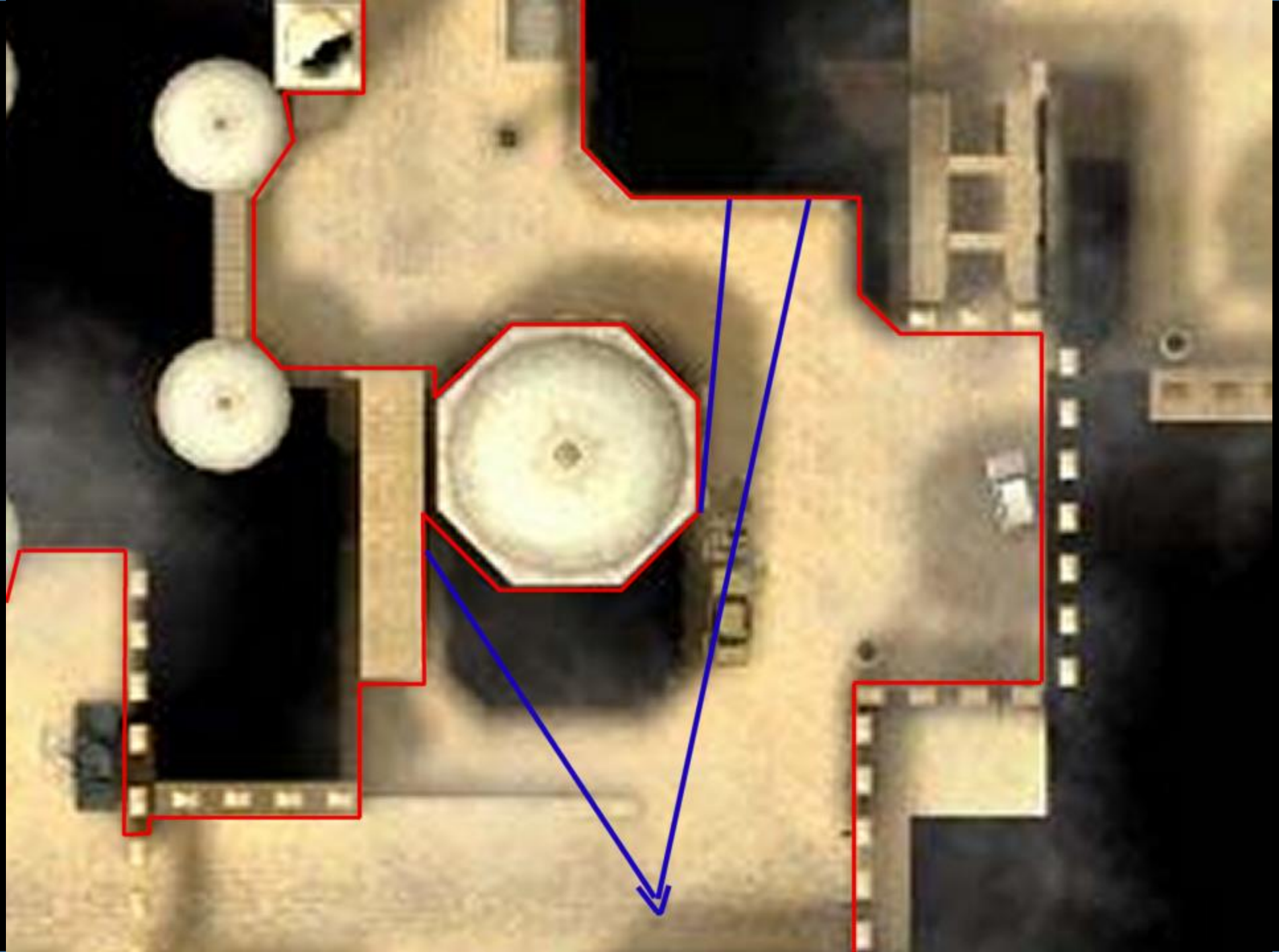
Visibility

Indoor visibility: Portals

Observation: if a window is invisible, the room it links to is invisible.








1
2
3
4


1
2
3
4


1
2
3
4


1
2
3
4


1
2
3
4


1
2
3
4

Visibility

Visibility determination

Coarse:

- Grid-based (typically outdoor)
- Portals (typically indoor)

Finer:

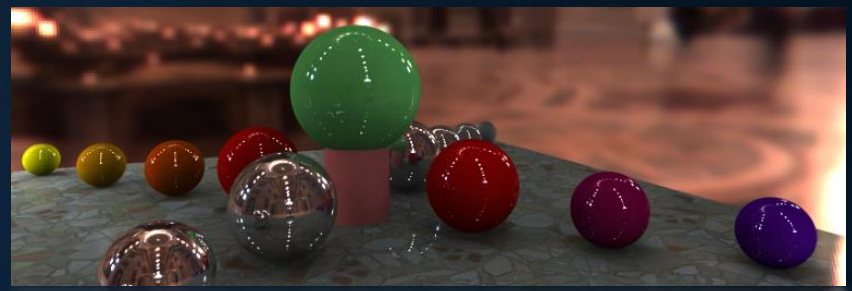
- Frustum culling
- Occlusion culling

Finest:

- Backface culling
- Clipping
- Z-buffer

Today’s Agenda:

- Depth Sorting
- Clipping
- Visibility



INFOGR – Computer Graphics

Jacco Bikker & Debabrata Panja - April-July 2017

END OF lecture 12: “Visibility”

Next lecture: “Postprocessing”