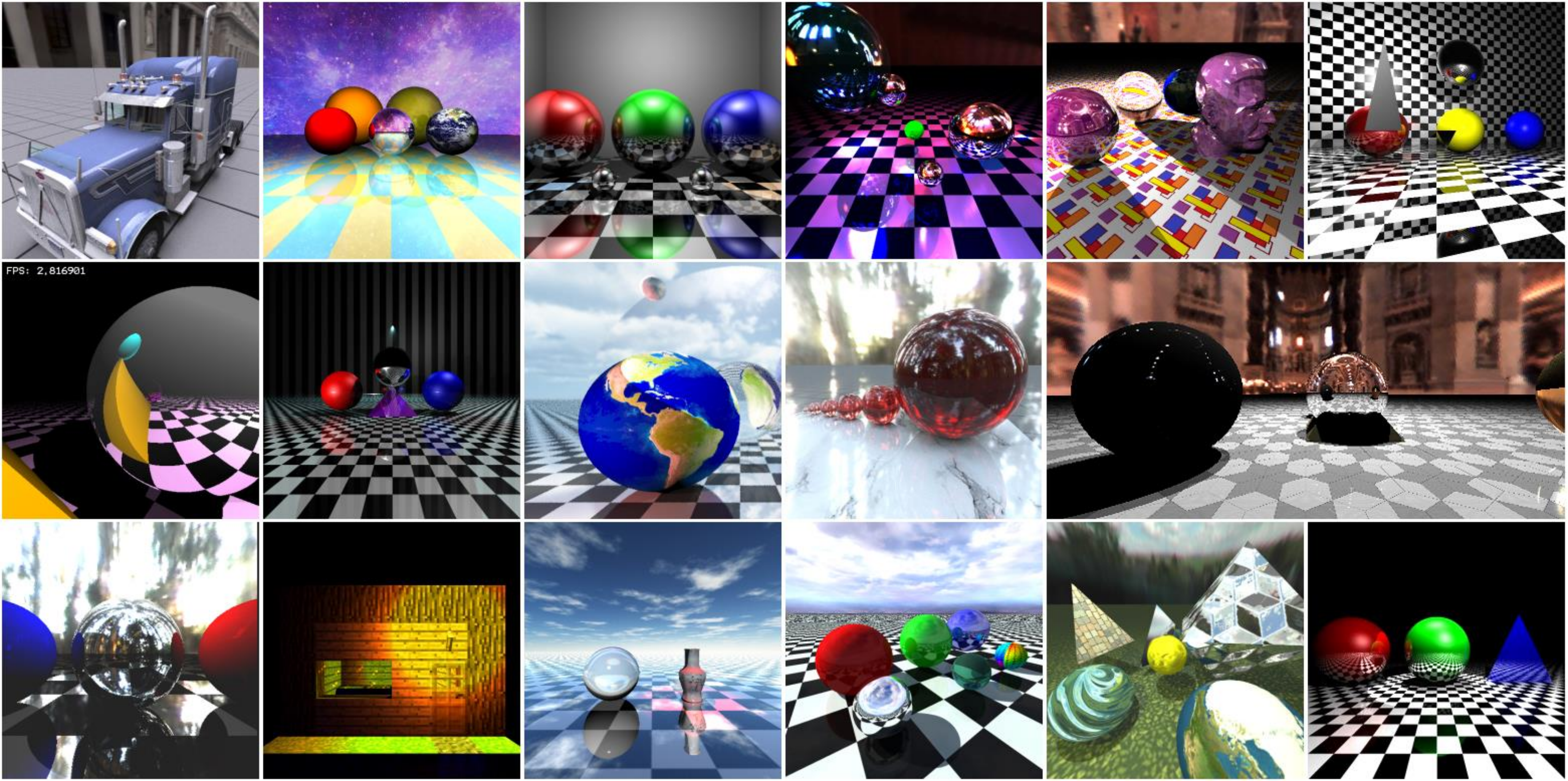
# INFOGR – Computer Graphics

J. Bikker - April-July 2016 - Lecture 11: “Visibility”

## Welcome!



```
ric
& (depth

: = insi
nt = nt
os2t = 1
O, N );
0)

at a = n
at Tr =
Tr) R =

E * diff
 = true;

efl + re

O, N );
refl * E
 = true;

MAXDEPTH

survive
 estimat
df;
radiance
e.x + ra

w = true
at brdfP
at3 fact
at weigh
at cosTh
 E * ((w

andom wa
vive)

;
at3 brdf = SampleDiffuse( diffuse, N, r1, r2, &R, &pdf
urvive;
 pdf;
n = E * brdf * (dot( N, R ) / pdf);
sion = true;
```

## Smallest Ray Tracers:

### Executable

Valeri Erling
Roderick Spaans

- 5692598 & 5683777: RTMini_minimal.exe – 2803 bytes
- 5741858: ASM_CPU_Min_Exe – 994 bytes

Marijn Suijten

### Source

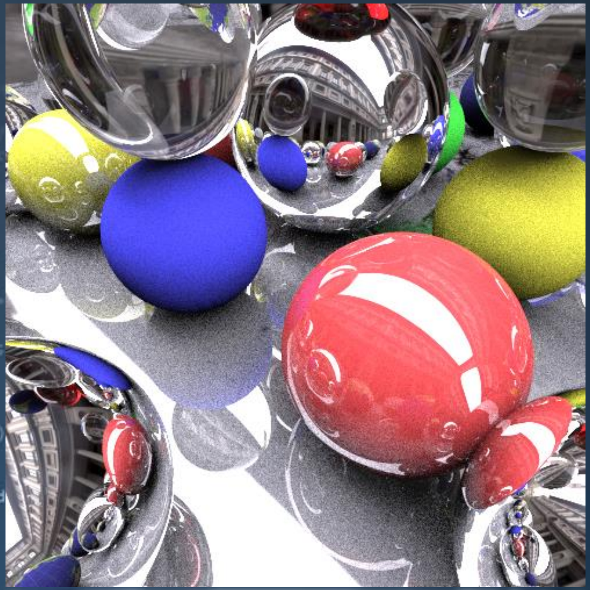
Ivo Gabe de Wolff
Lars Folkersma

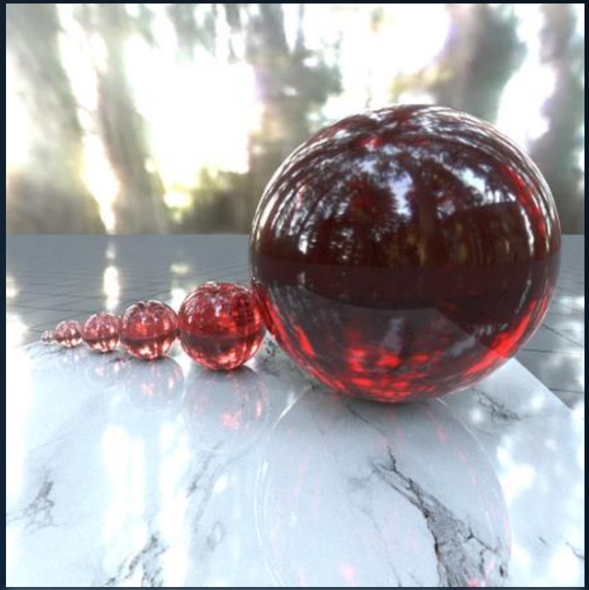
- 4279433 & 5543800: Haskell ray tracer; 2280 characters.
- 5741858: C# ray tracer, 1235 characters.

Marijn Suijten

```
using V=System.Numerics.Vector3;using static System.Math; using f=System.Single;using System.Drawing;class
S{public V P,C=V.One;public int T;public f r,R;public S(V p,f a,f b){P=p;R=a*a;r=b;}public void I(R r){V L
=P-r.O;f a=V.Dot(L,r.D),d=V.Dot(L,L)-a*a;if(a>0&&d<R){f t=a-(f)Sqrt(R-d);if(t>0&&t<r.i){r.i=t;r.N=
V.Normalize(r.O+t*r.D-P);r.p=this;}}}}class R{public V O,D,N;public S p;public f i=99;public R(V o,V d){O=
o+1e-4f*d;D=d;}public R(V d){D=d;}}class A{V P=V.One;S x=new S(V.UnitY*-500,498,.7f){T=1},y=new S(new V(-1
,0,4),.6f,1),z=new S(new V(1,0,4),.6f,0){C=V.UnitX};void D(R r){x.I(r);y.I(r);z.I(r);}A(){int S=512;Bitmap
b=new Bitmap(S,S);for(int i=0;i<S*S;i++)b.SetPixel(i%S,i/S,R(B(new R(V.Normalize(new V((f)(i%S)/S-.5f,.5f-
(f)(i/S)/S, 1)))))); b.Save("r.bmp");}V B(R r){D(r);V C=V.Zero;if(r.p!=null){V I=r.O+r.i*r.D,c=r.p.T>0?new
V((int)(I.X-9)+(int)(I.Z-9)&1):r.p.C,L=V.Normalize(P-I);f f=r.p.r,d;R a=new R(I,L);D(a);if(a.p==null){if (
(d=V.Dot(L,r.N))>0)C+=c*d*(1-f)/(V.Distance(I,P)/9+1);if((d=V.Dot(r.D,V.Reflect(L,r.N)))>0)C+=new V((f)Pow
(d,9)*f);}C+=f*B(new R(I,V.Reflect(r.D,r.N)))*c;} return C;}Color R(V v)=>Color.FromArgb(S(v.X),S(v.Y), S(
v.Z));int S(f f)=>(int)(f<0?0:f>1?1:Sqrt(f)*255);static void Main(){new A();}}
```

# Fastest Ray Tracer:





# Today’s Agenda:

- Depth Sorting
- Clipping
- Visibility

# Depth Sorting

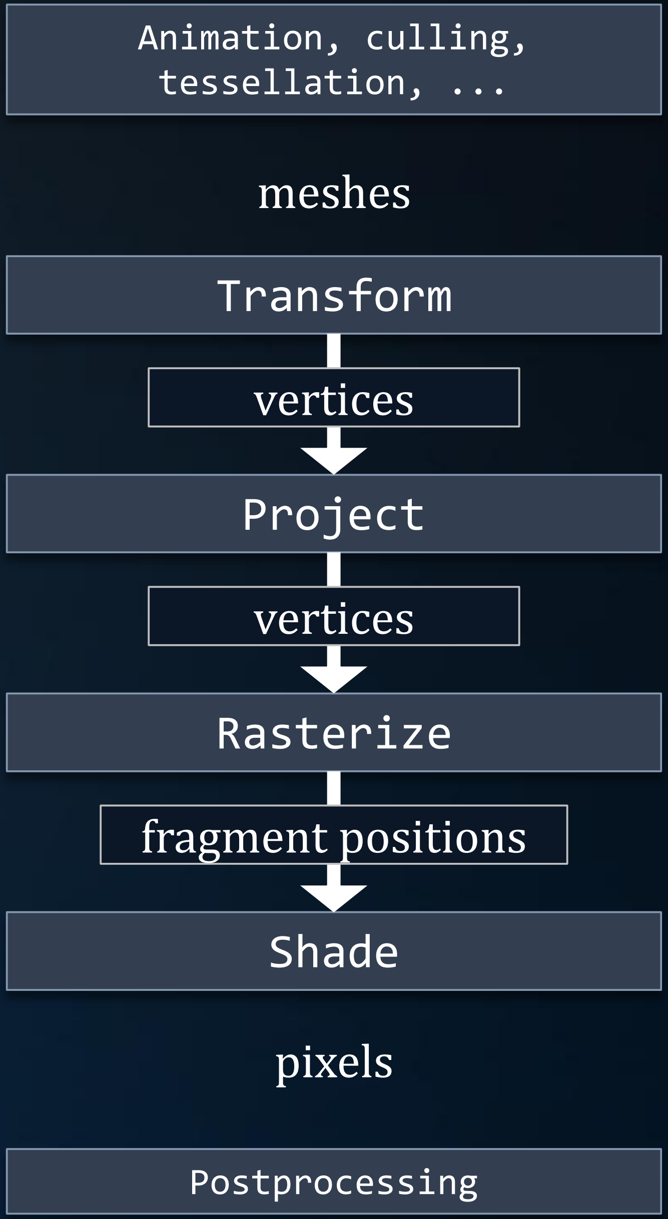
Rendering – Functional overview

1. Transform:
   translating / rotating meshes

2. Project:
   calculating 2D screen positions

3. Rasterize:
   determining affected pixels

4. Shade:
   calculate color per affected pixel



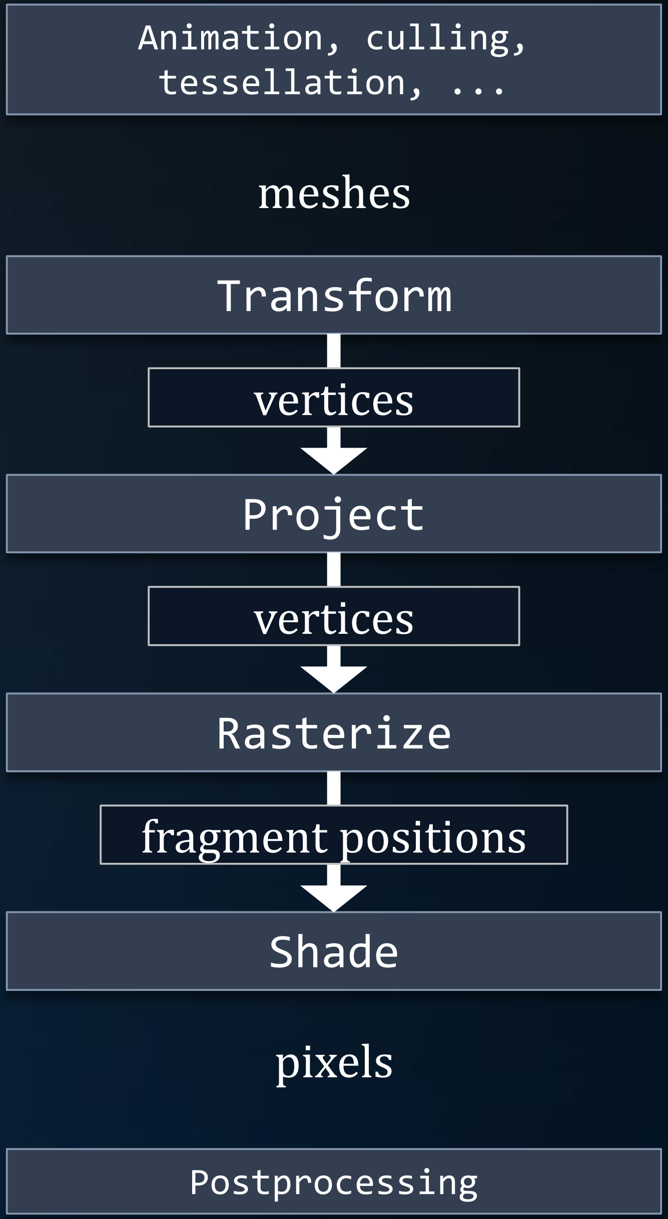
# Depth Sorting

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?



Animation, culling, tessellation, ...

meshes

Transform

vertices

Project

vertices

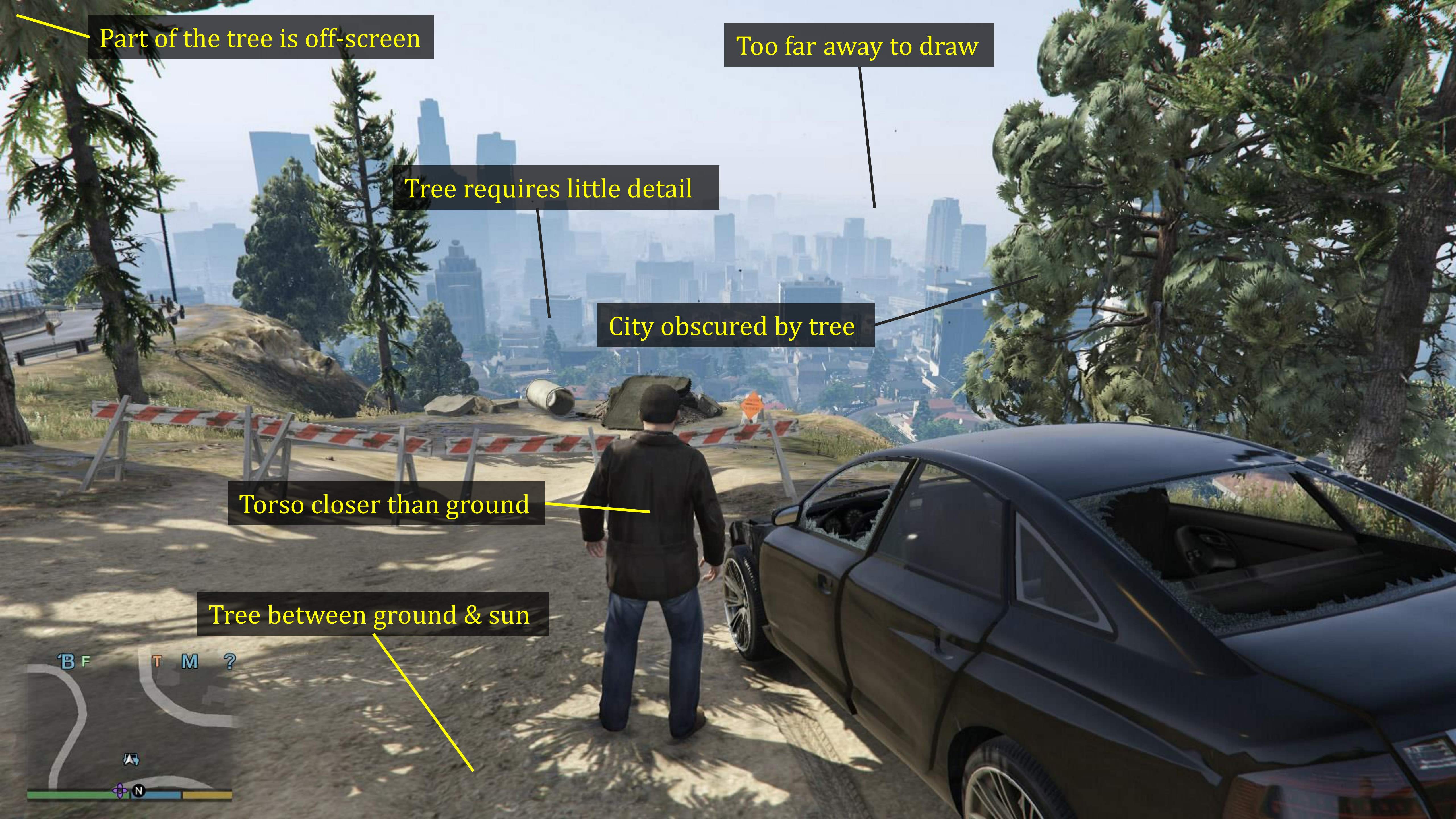
Rasterize

fragment positions

Shade

pixels

Postprocessing



Part of the tree is off-screen

Too far away to draw

Tree requires little detail

City obscured by tree

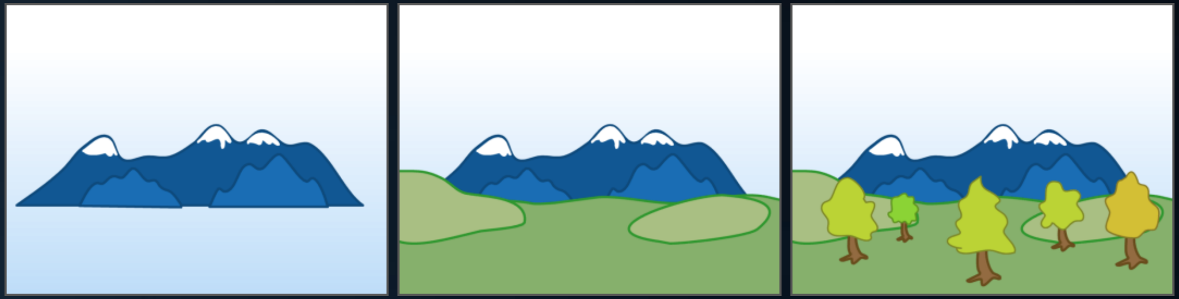
Torso closer than ground

Tree between ground & sun

# Depth Sorting

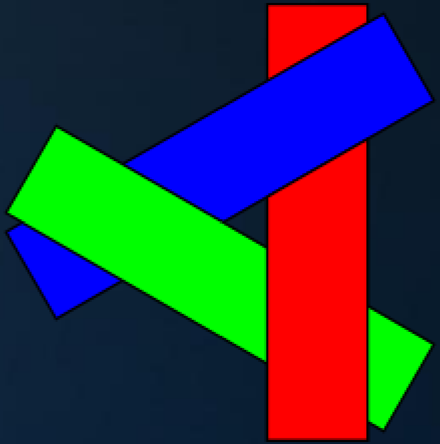
Old-skool depth sorting: Painter’s Algorithm

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



Advantage:

- Doesn’t require z-buffer



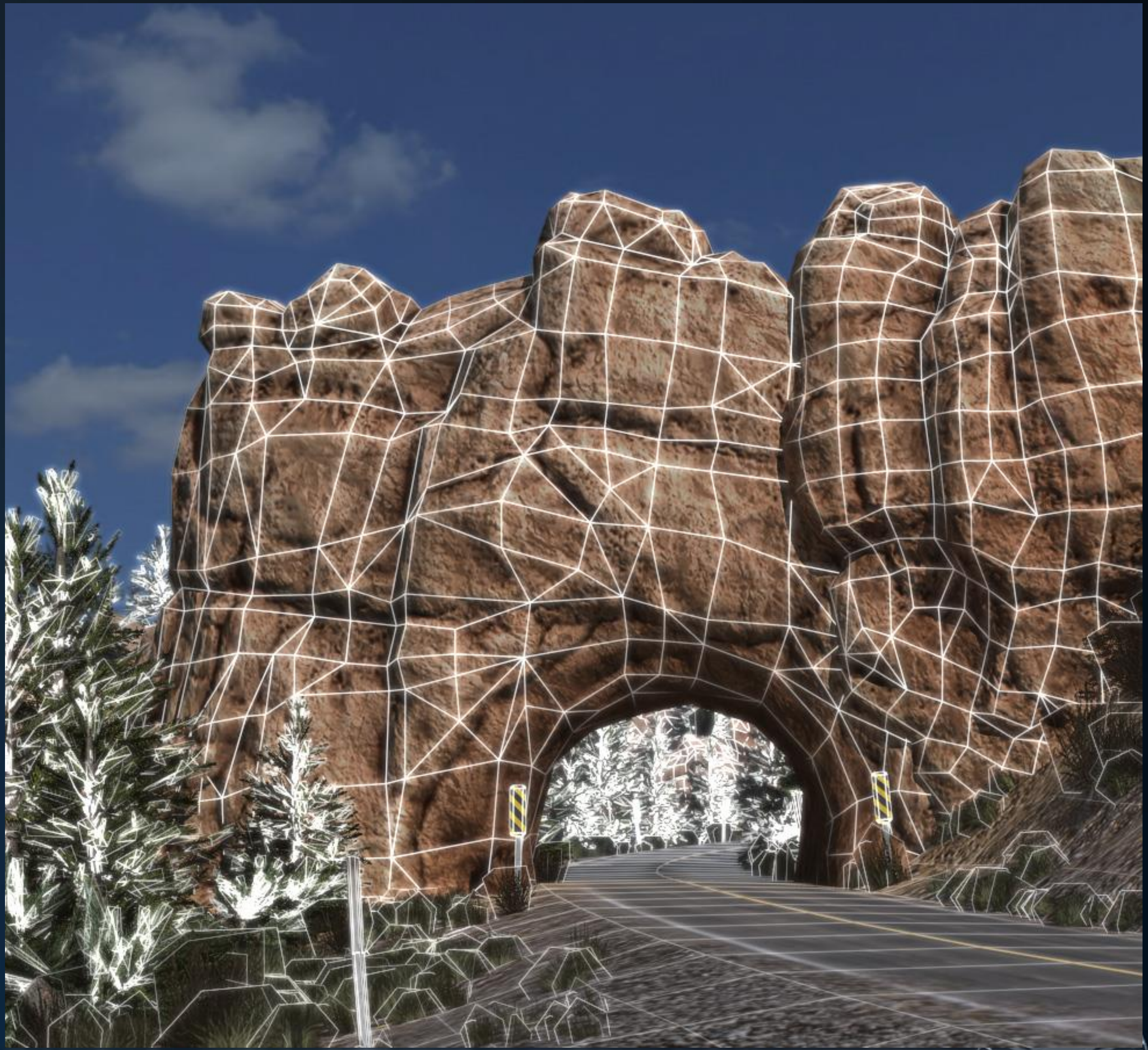
Problems:

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

# Depth Sorting

Overdraw:

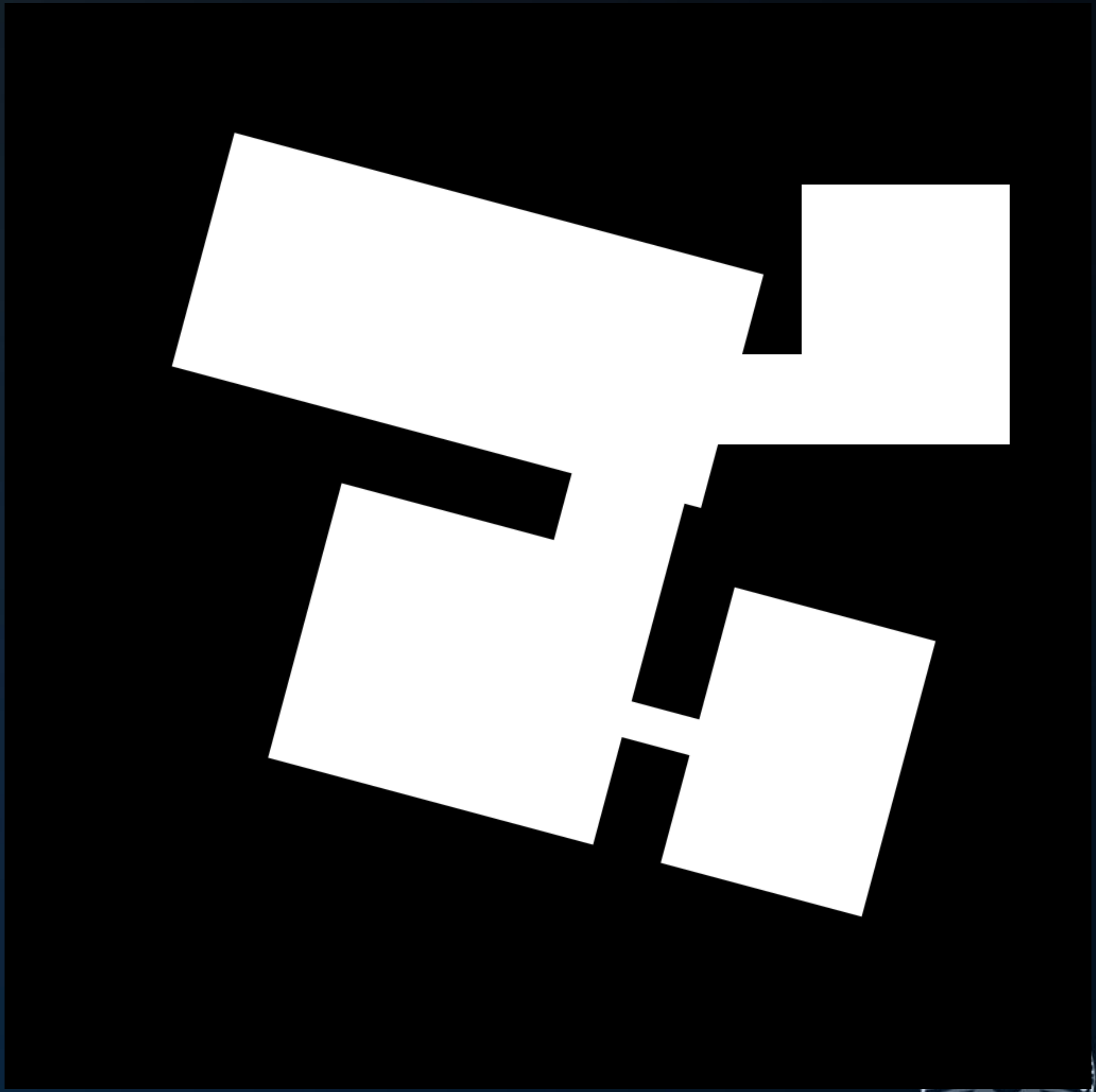
Inefficiency caused by drawing multiple times to the same pixel.



# Depth Sorting

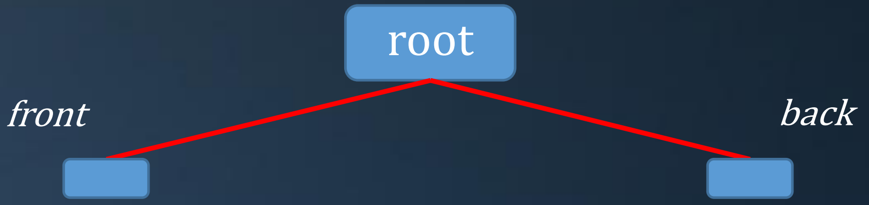
Correct order: BSP

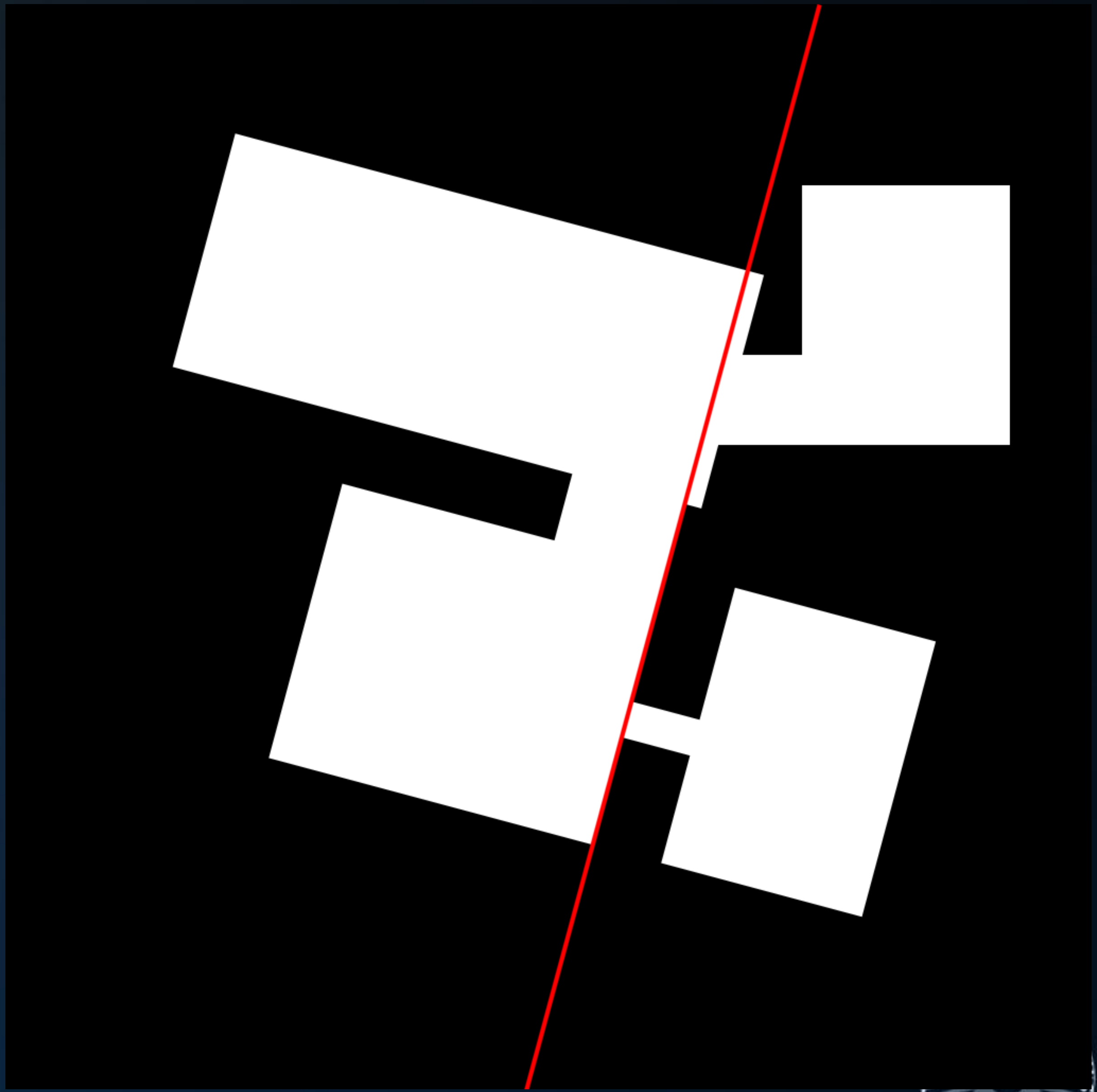
root



# Depth Sorting

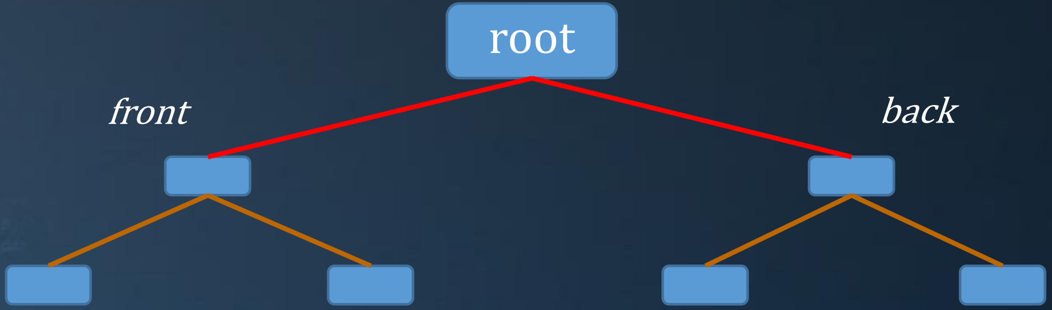
Correct order: BSP

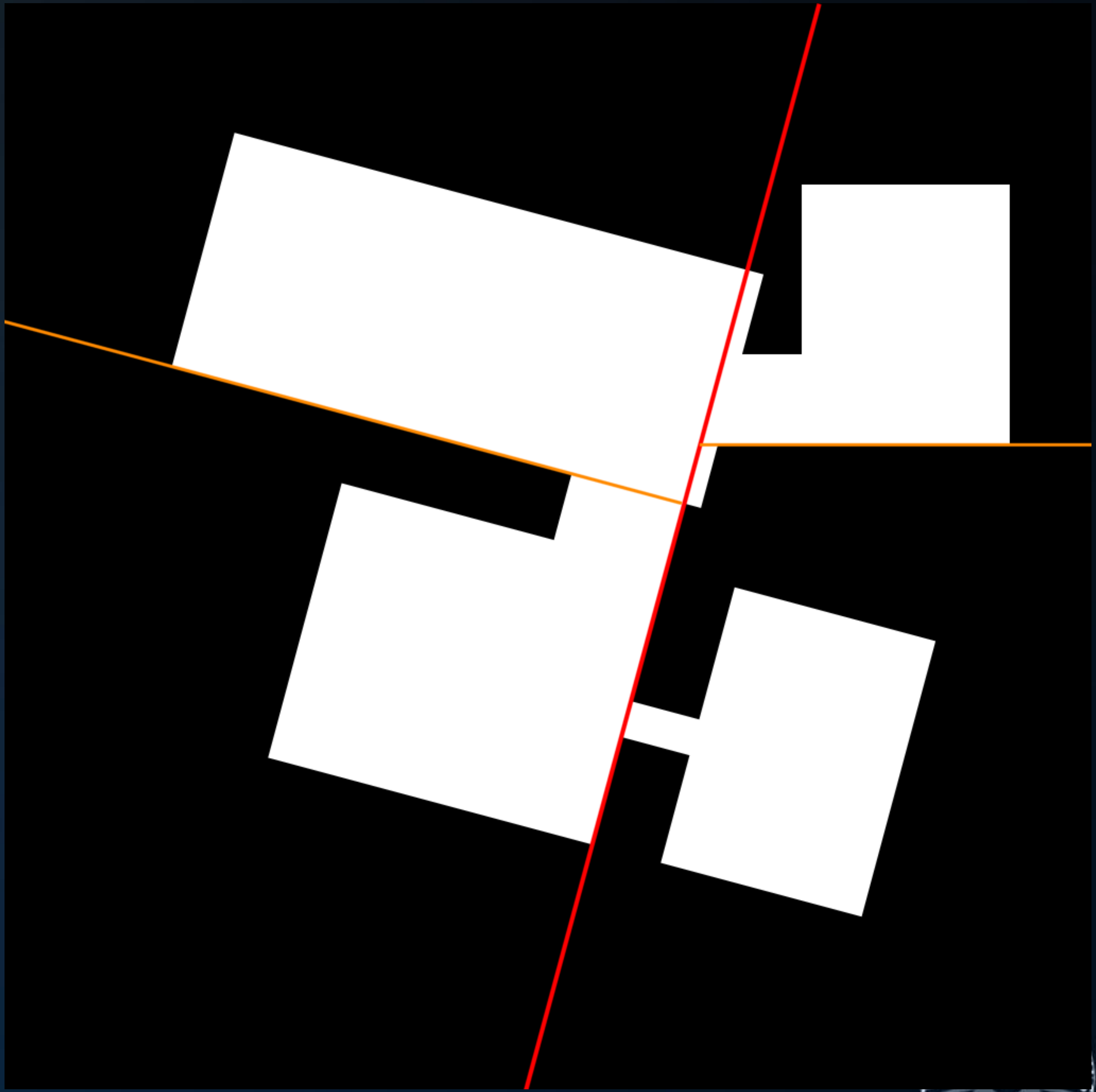




# Depth Sorting

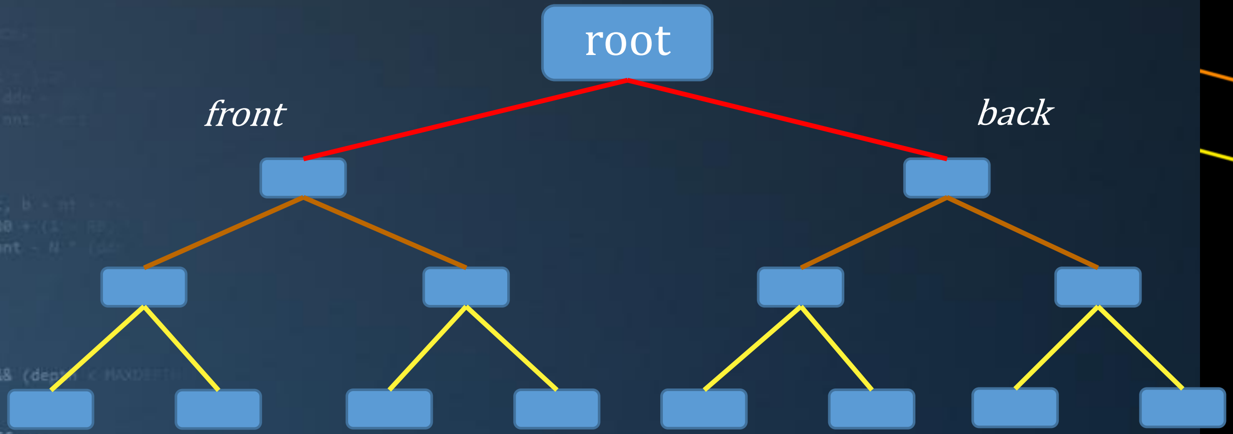
Correct order: BSP

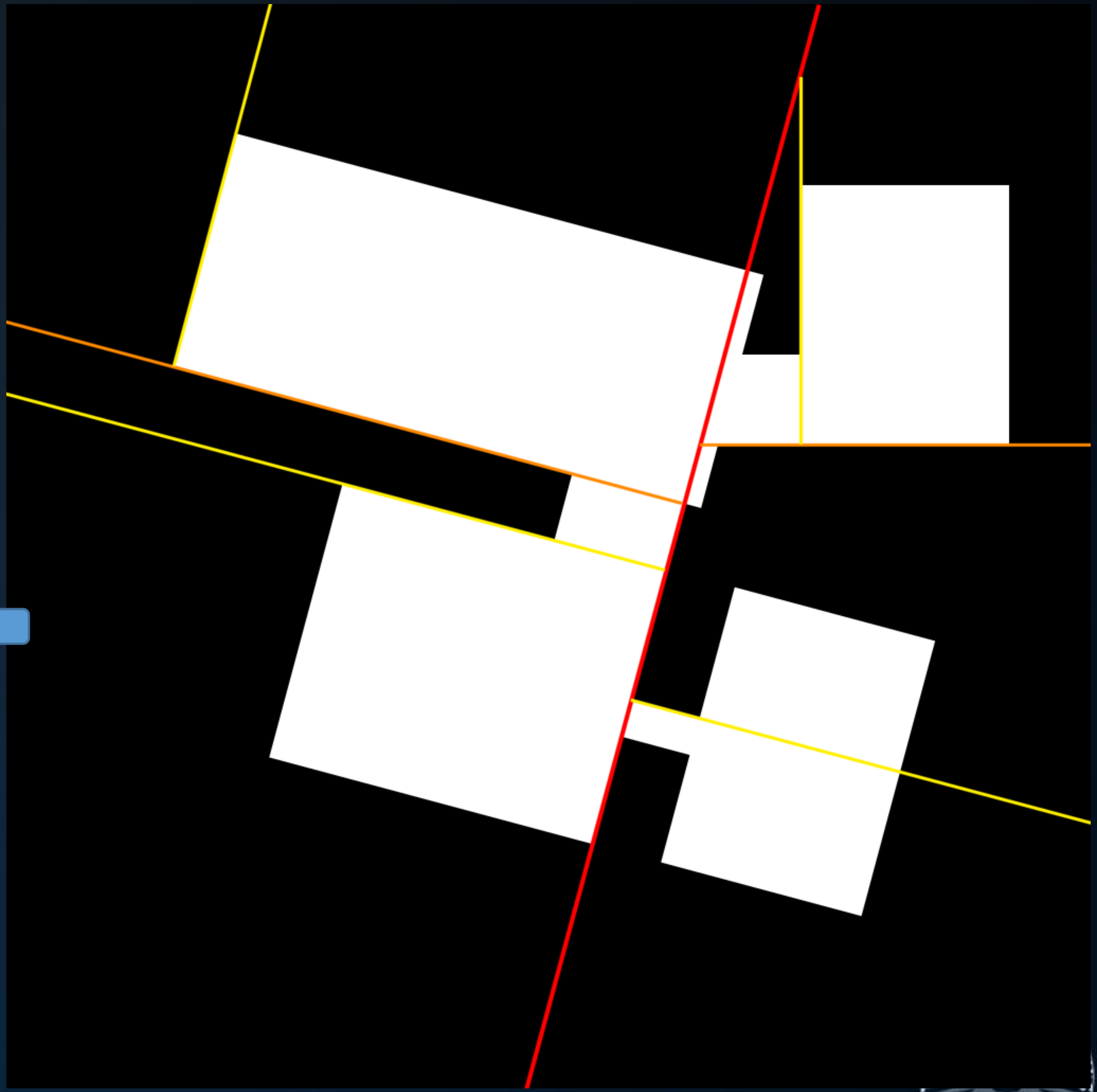


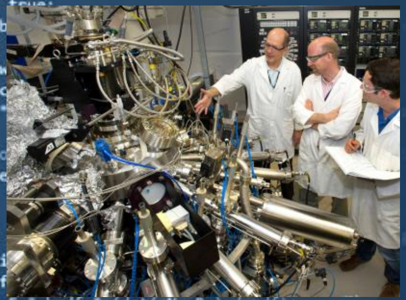


# Depth Sorting

Correct order: BSP

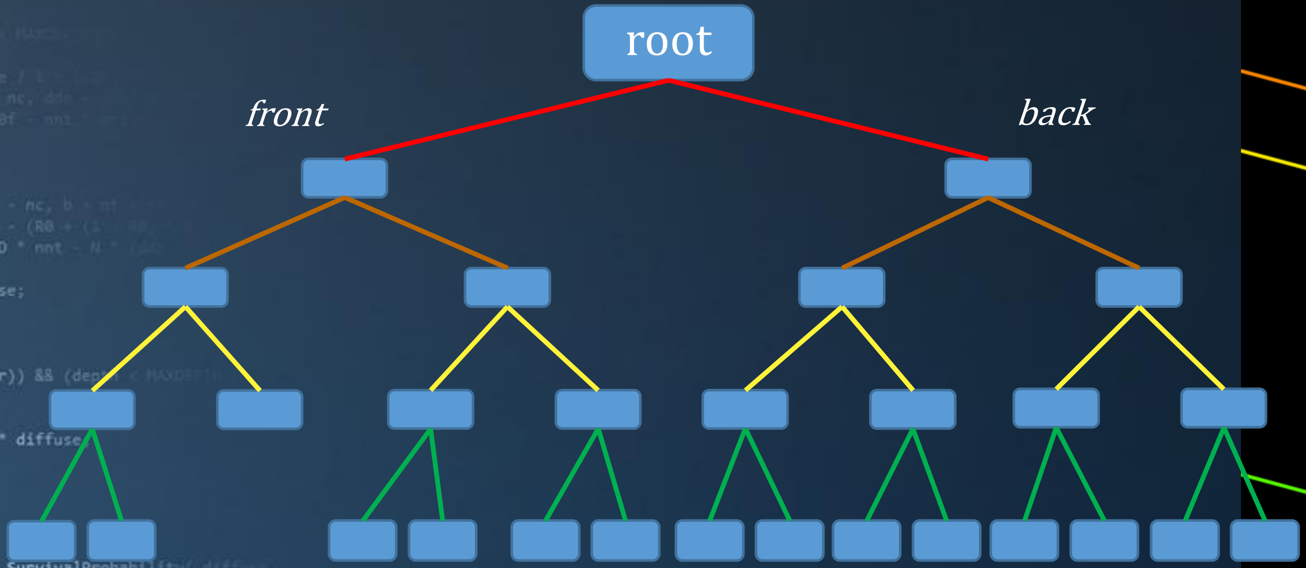


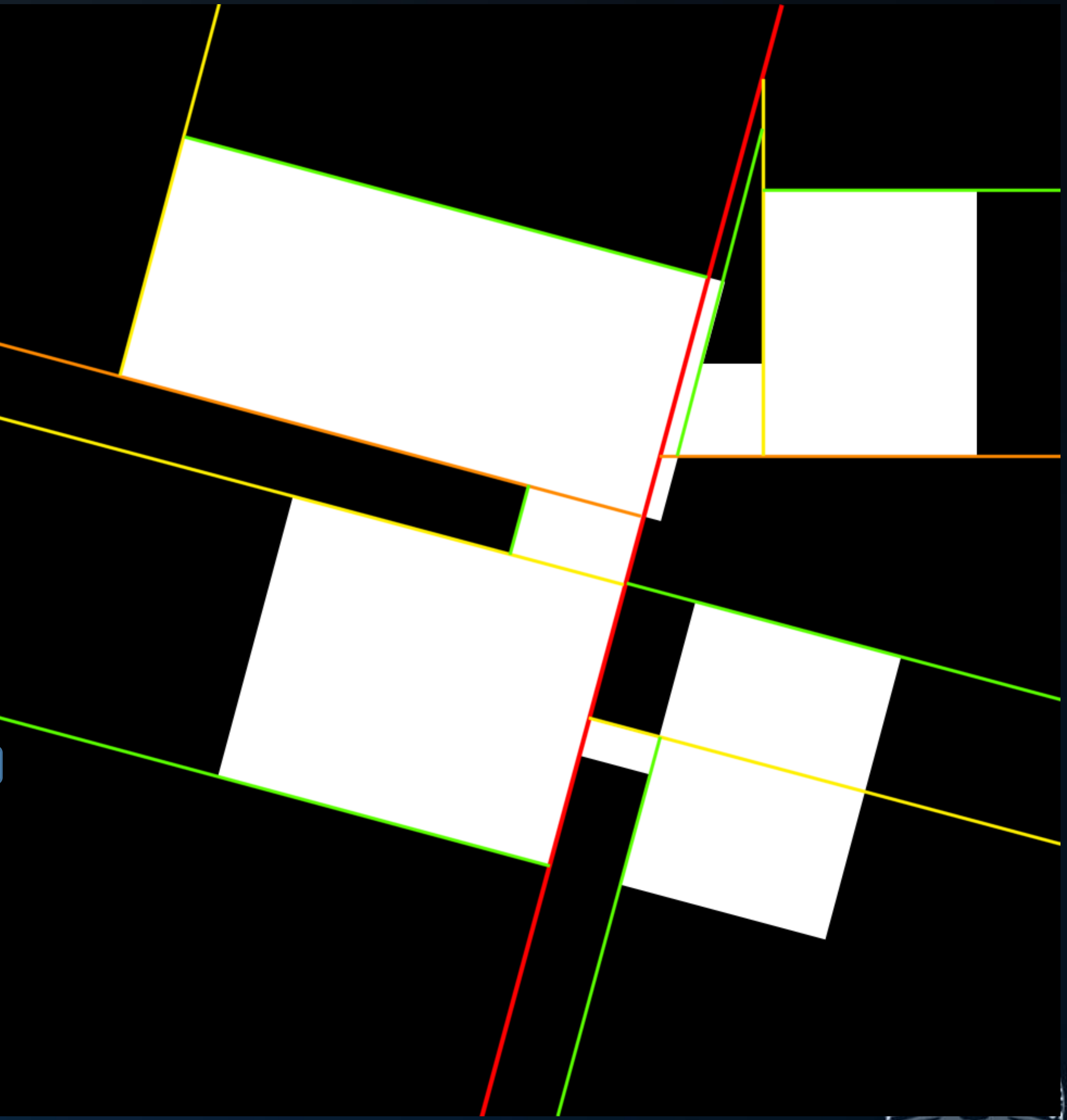




# Depth Sorting

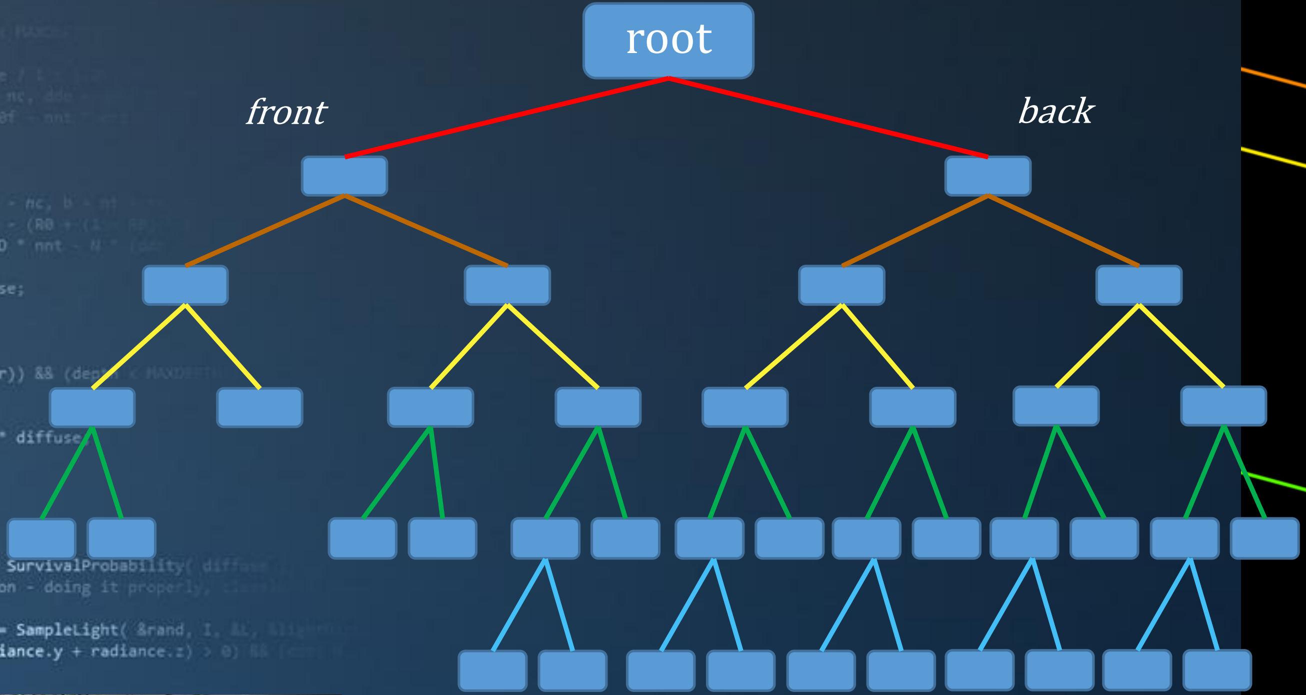
Correct order: BSP

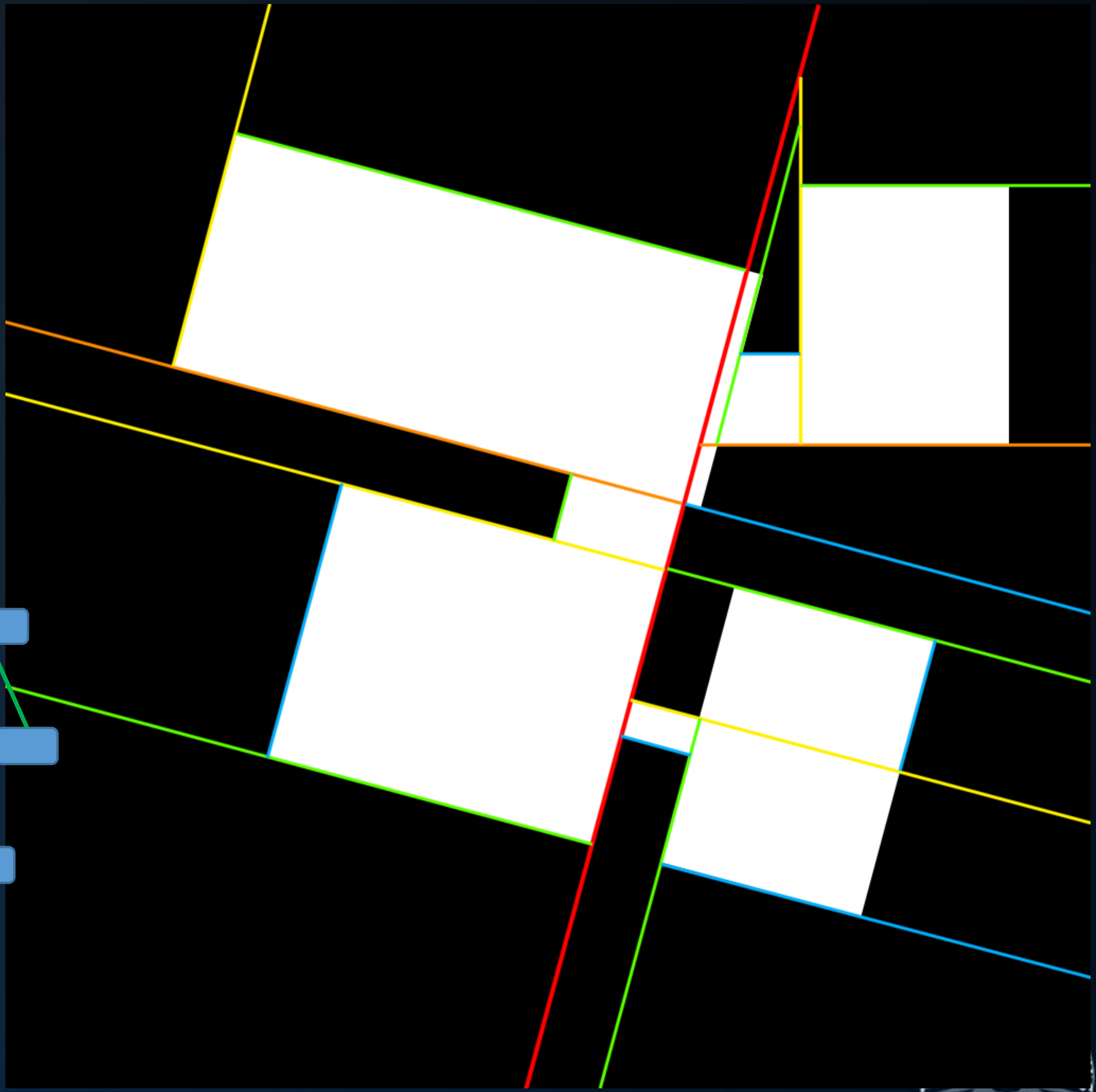


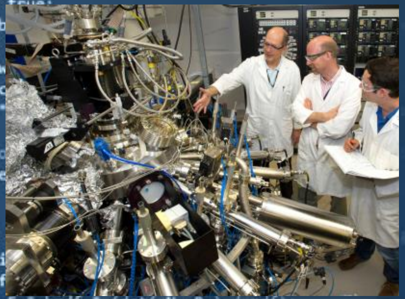


# Depth Sorting

Correct order: BSP

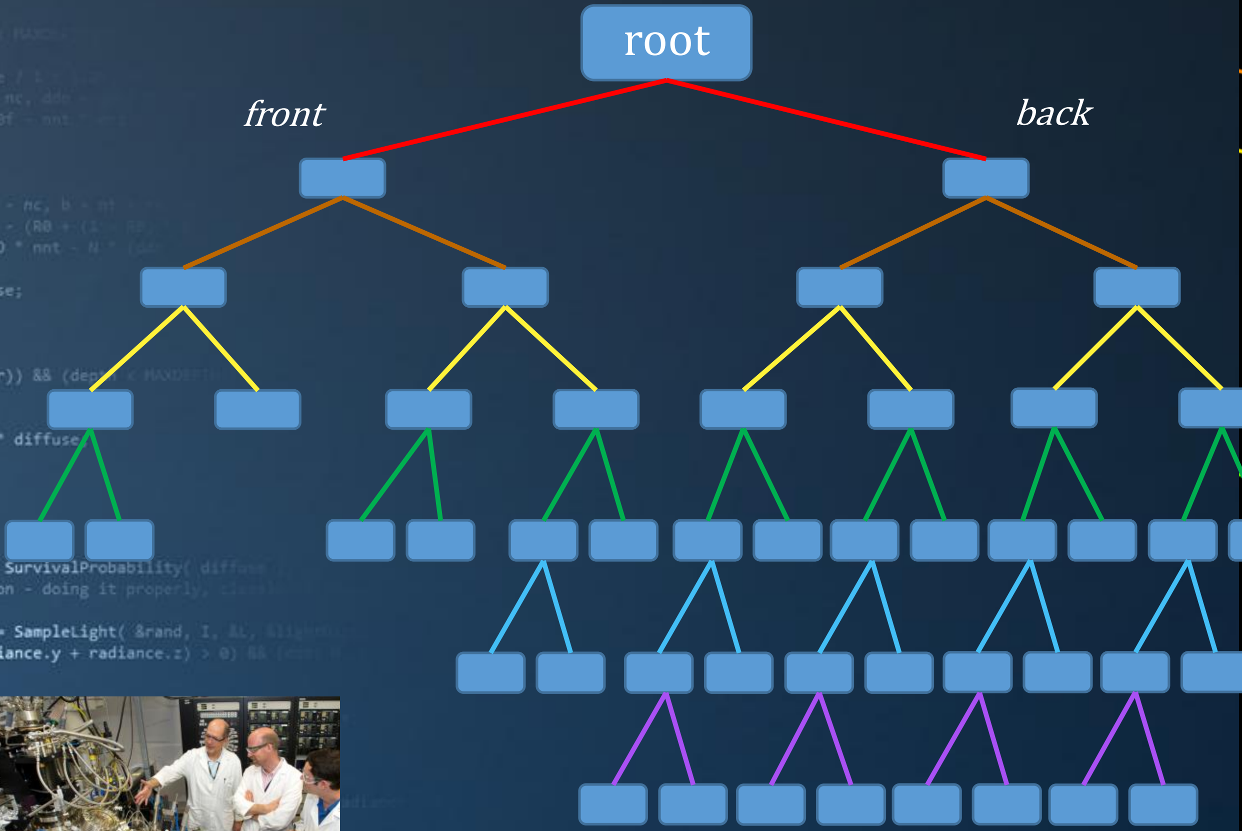


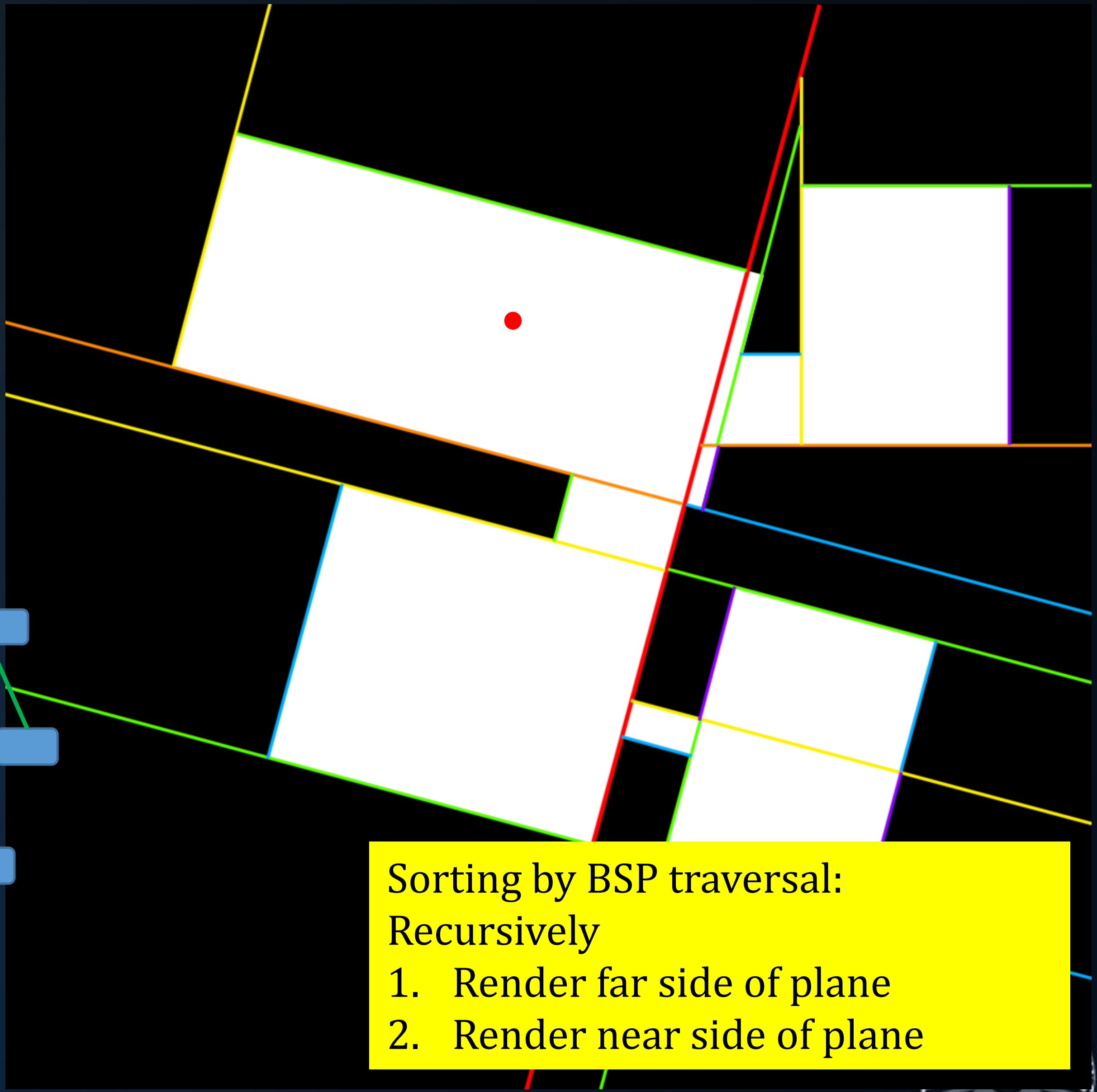




# Depth Sorting

Correct order: BSP





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

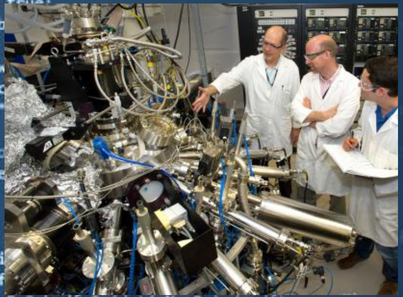
# Depth Sorting

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 $\mathrm{z_{int}} = \frac{(2^{32}-1)}{-z+1}$:
this way, any z < 0 will be in the range $\mathrm{z_{adjusted}} = -z_{original} + 1 = 1..\infty$, 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 $z_{original} = -\infty$.

# 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.

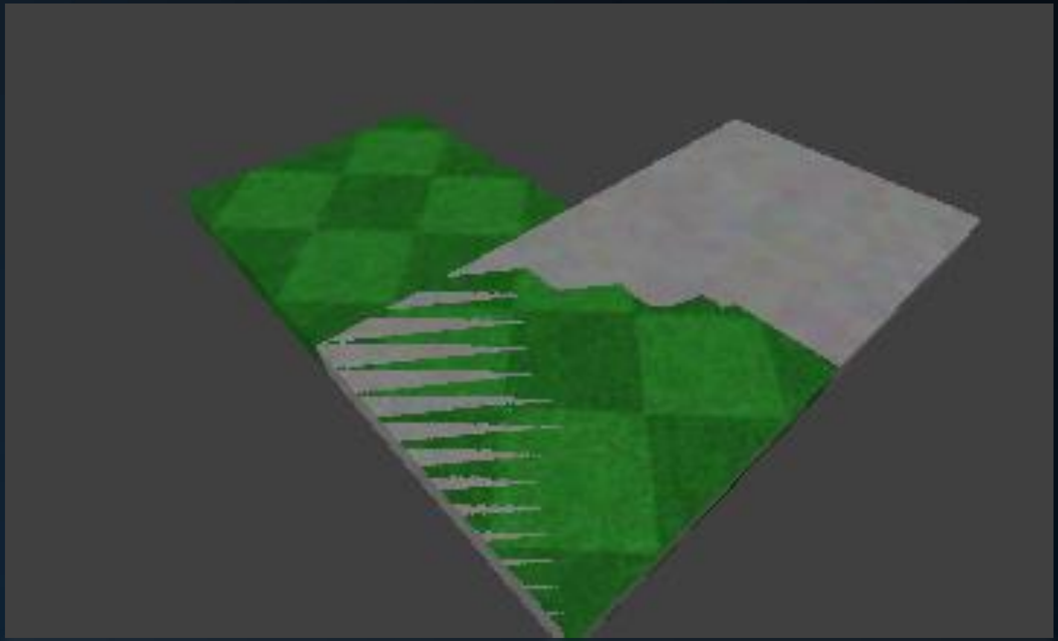
The ideal case can be approached by using Painter’s to ‘pre-sort’.

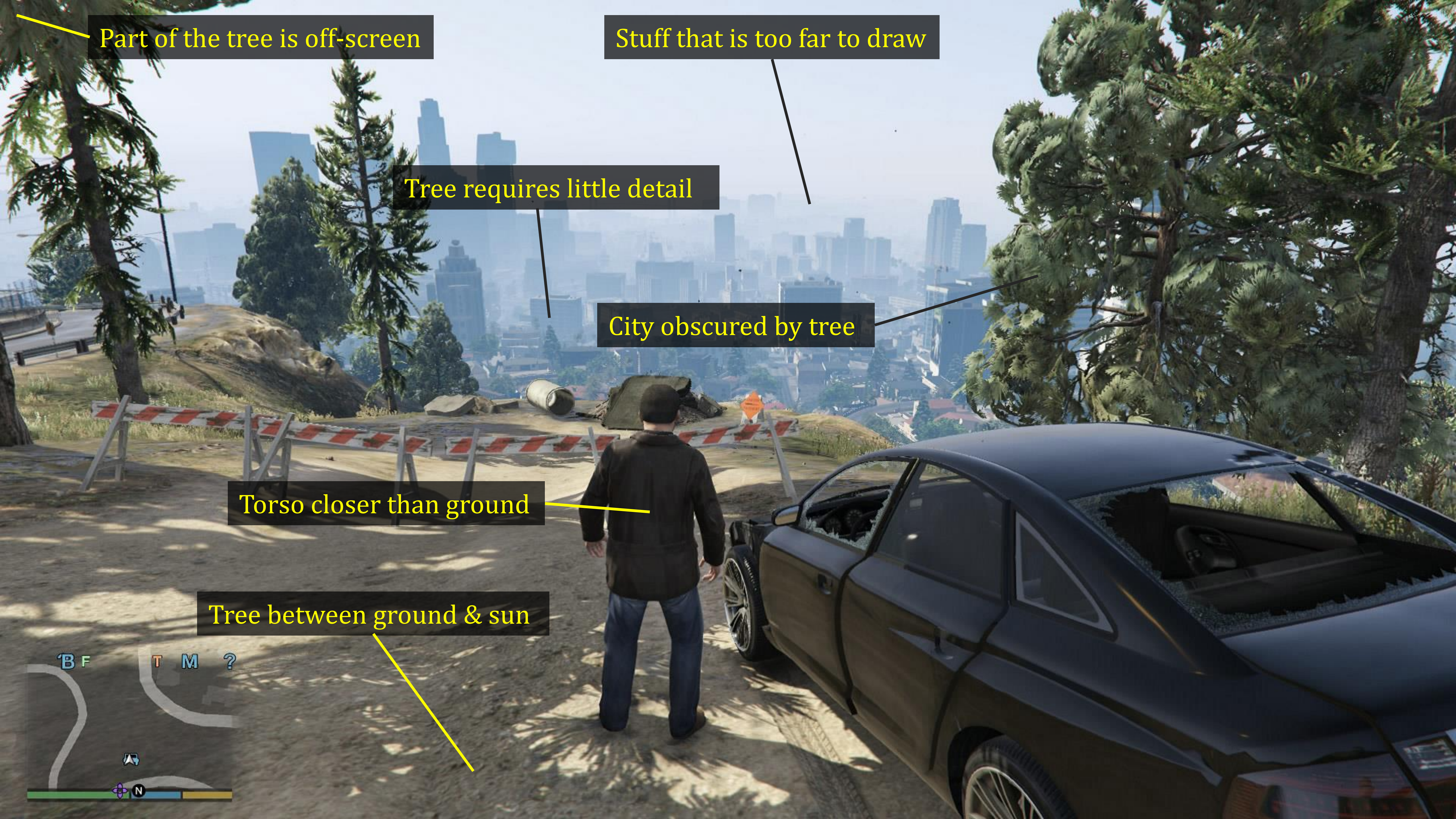
# Depth Sorting

‘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

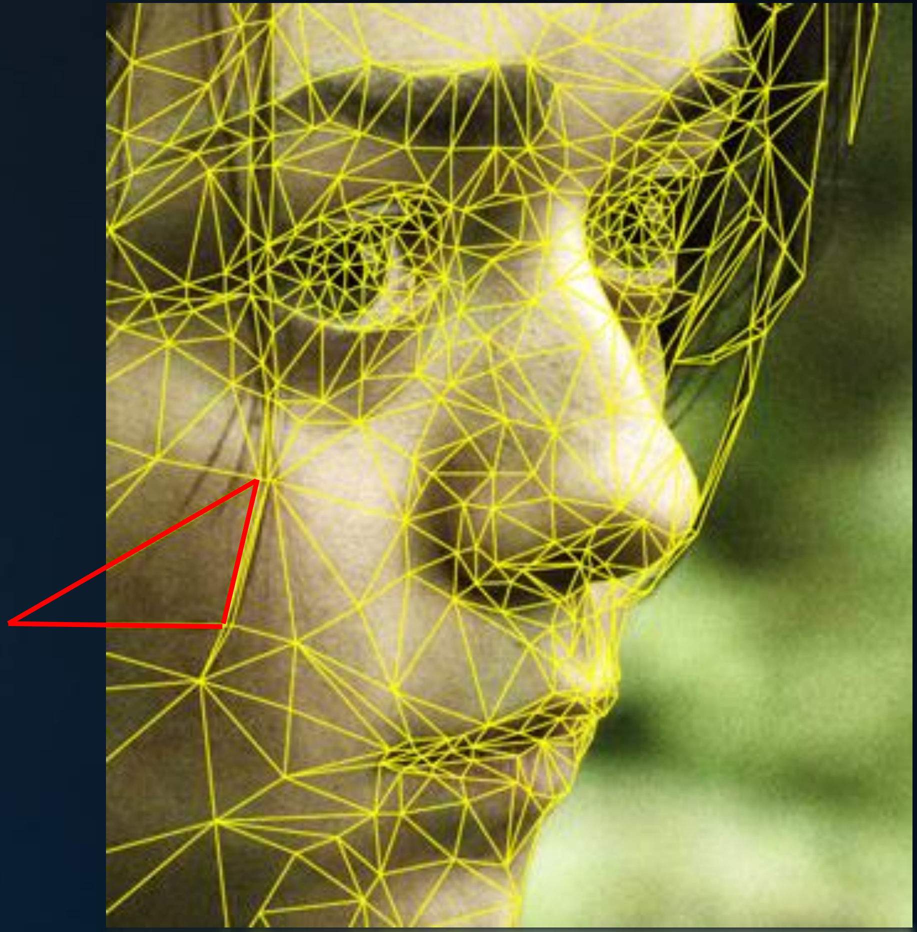
# Clipping

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

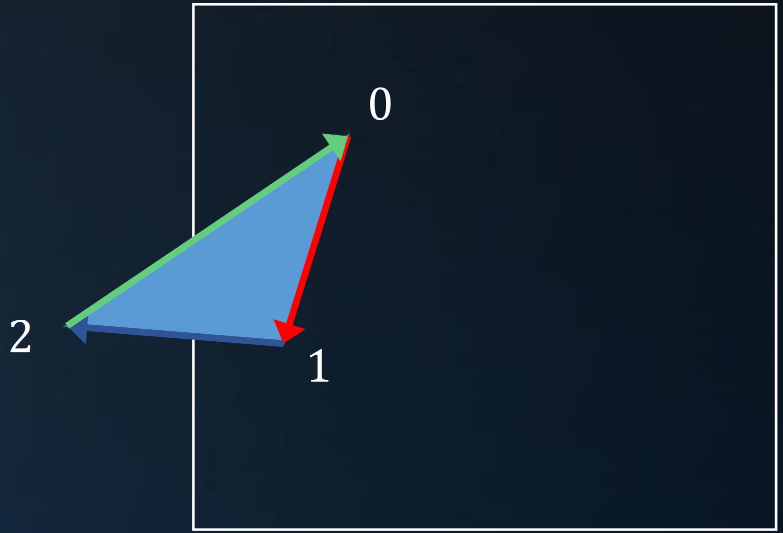
Algorithm:

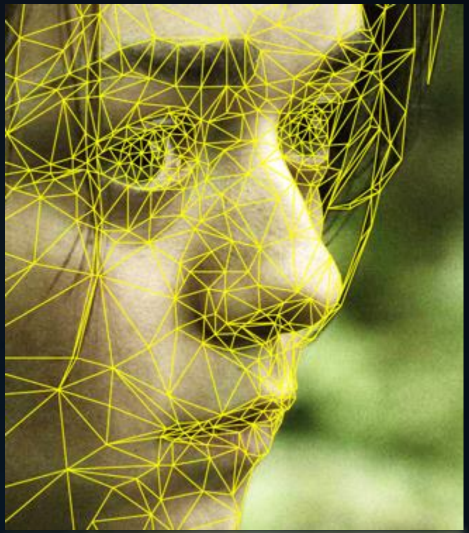
Per edge with vertices $v_0$ and $v_1$:

- 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

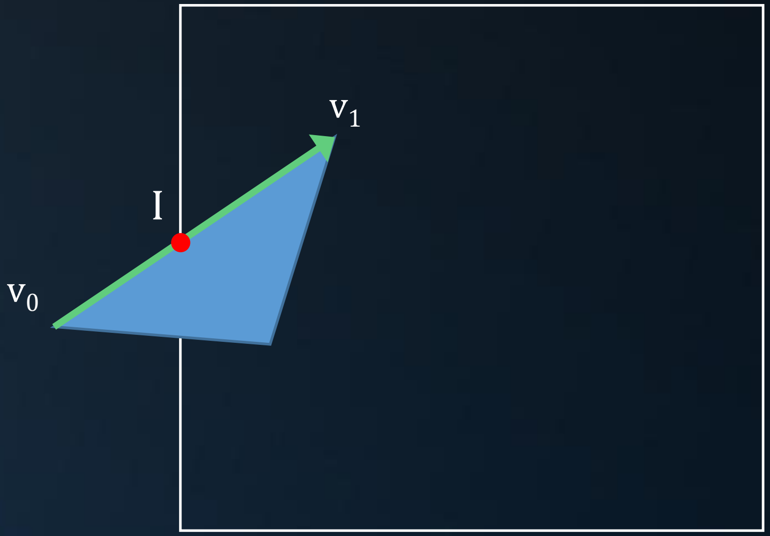
Sutherland-Hodgeman

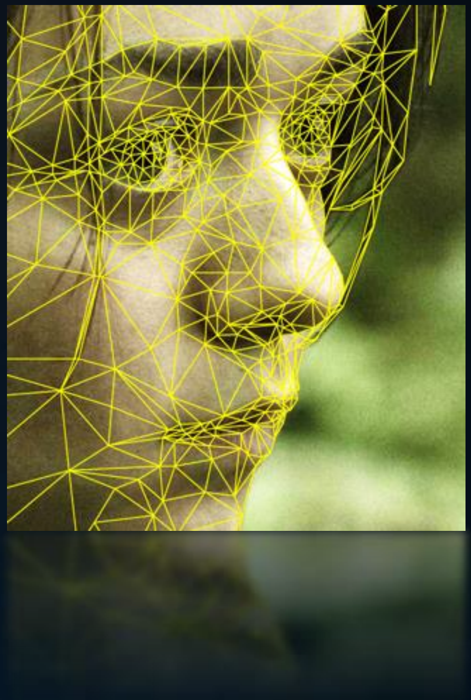
Calculating the intersections with plane $ax + by + cz + d = 0$:

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

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

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





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

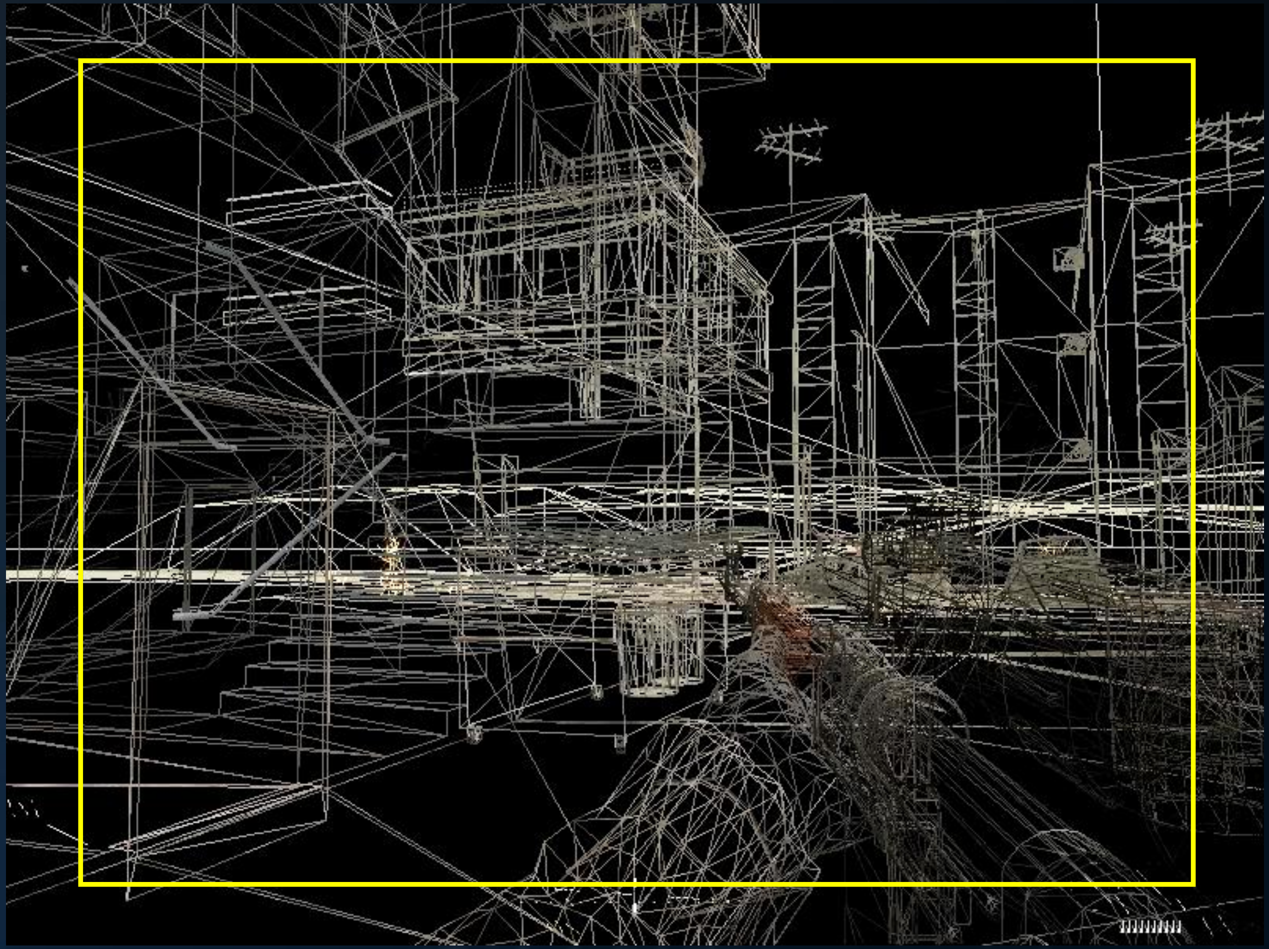
0,1,2,3,4 ➔ 0, 1, 2 + 0, 2, 3 + 0, 3, 4.

# Clipping

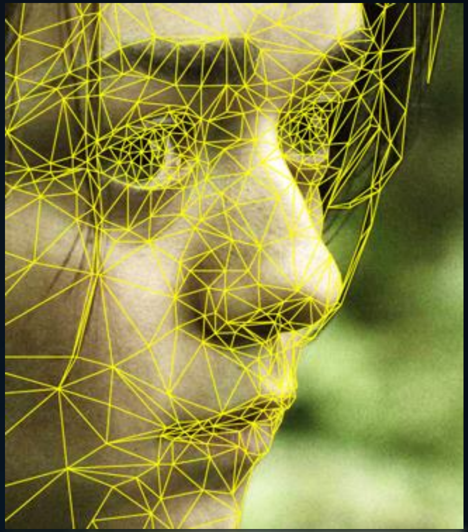
Guard bands

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.

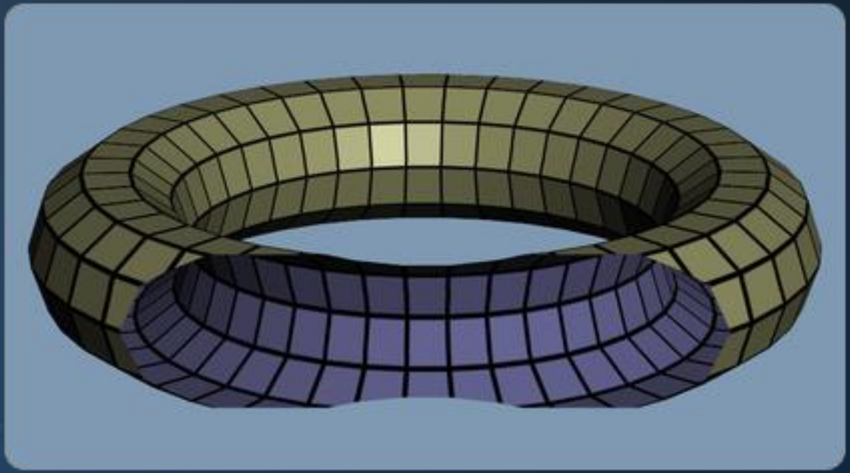


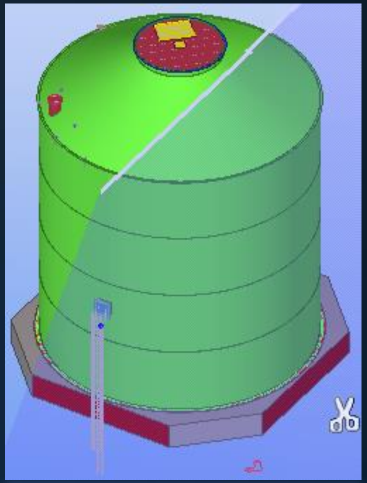
# Clipping

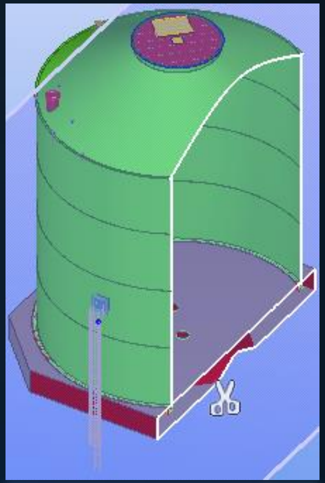


Sutherland-Hodgeman

Clipping can be done against arbitrary planes.

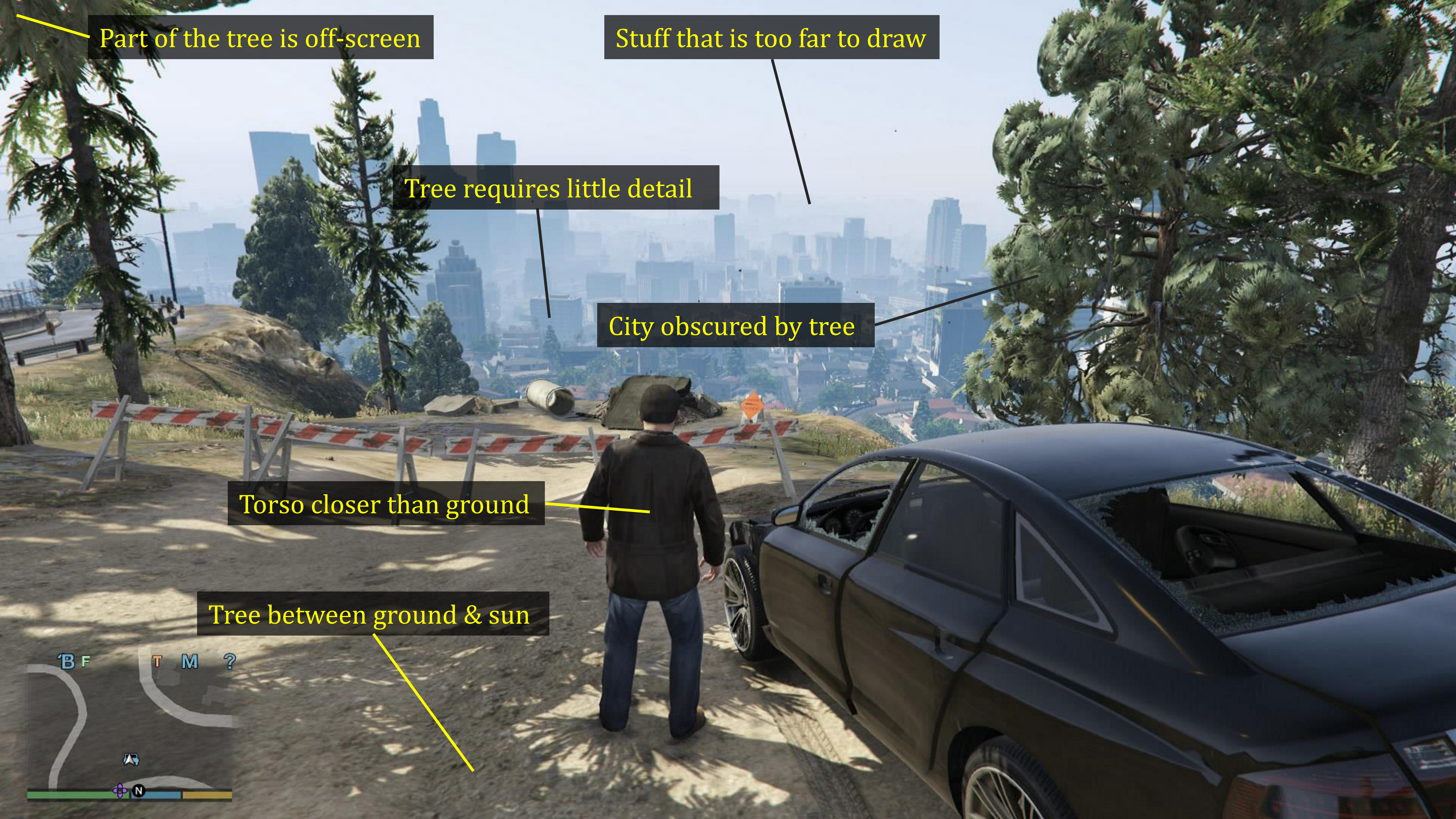






# Today’s Agenda:

- Depth Sorting
- Clipping
- Visibility



Part of the tree is off-screen

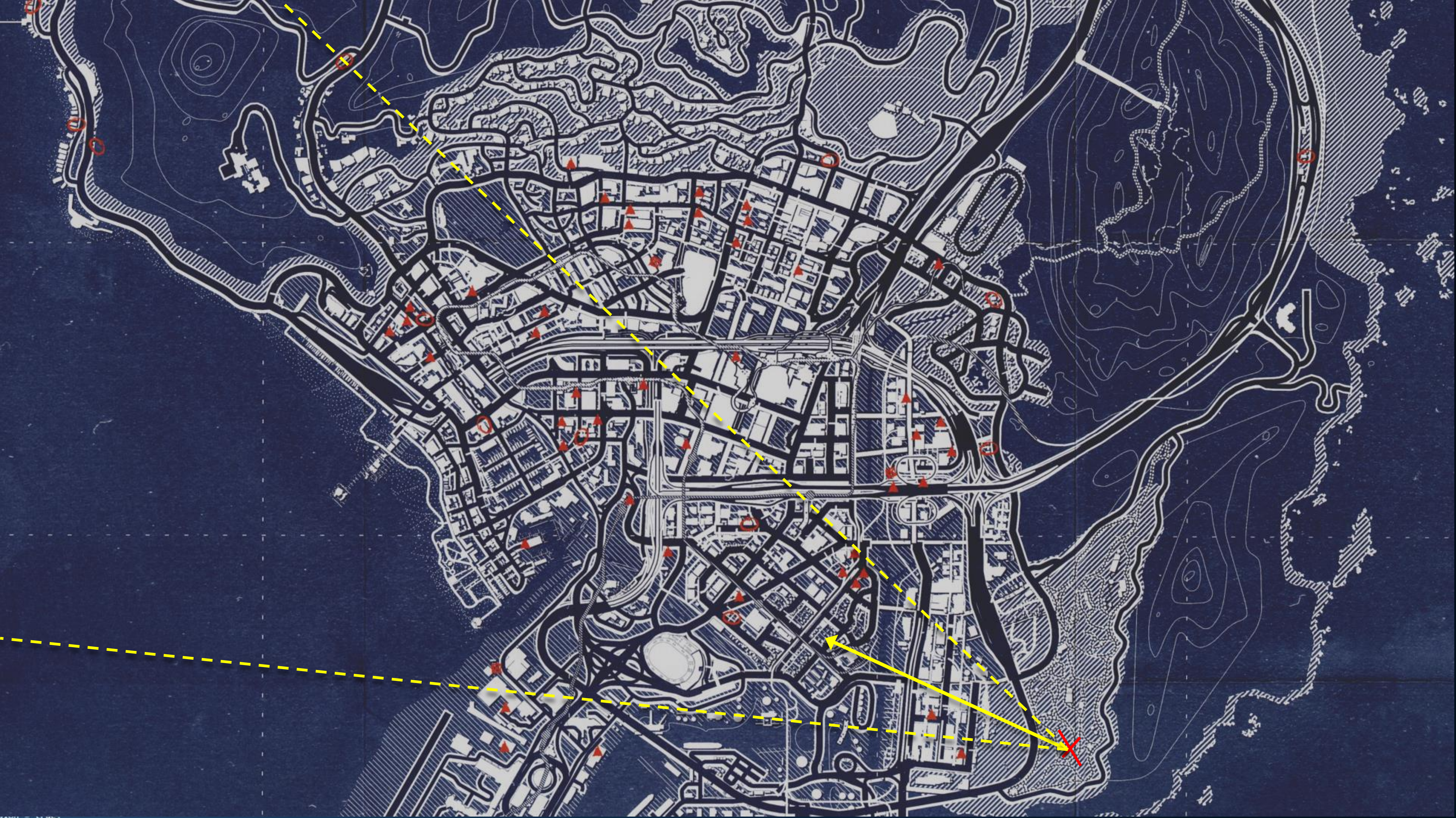
Stuff that is too far to draw

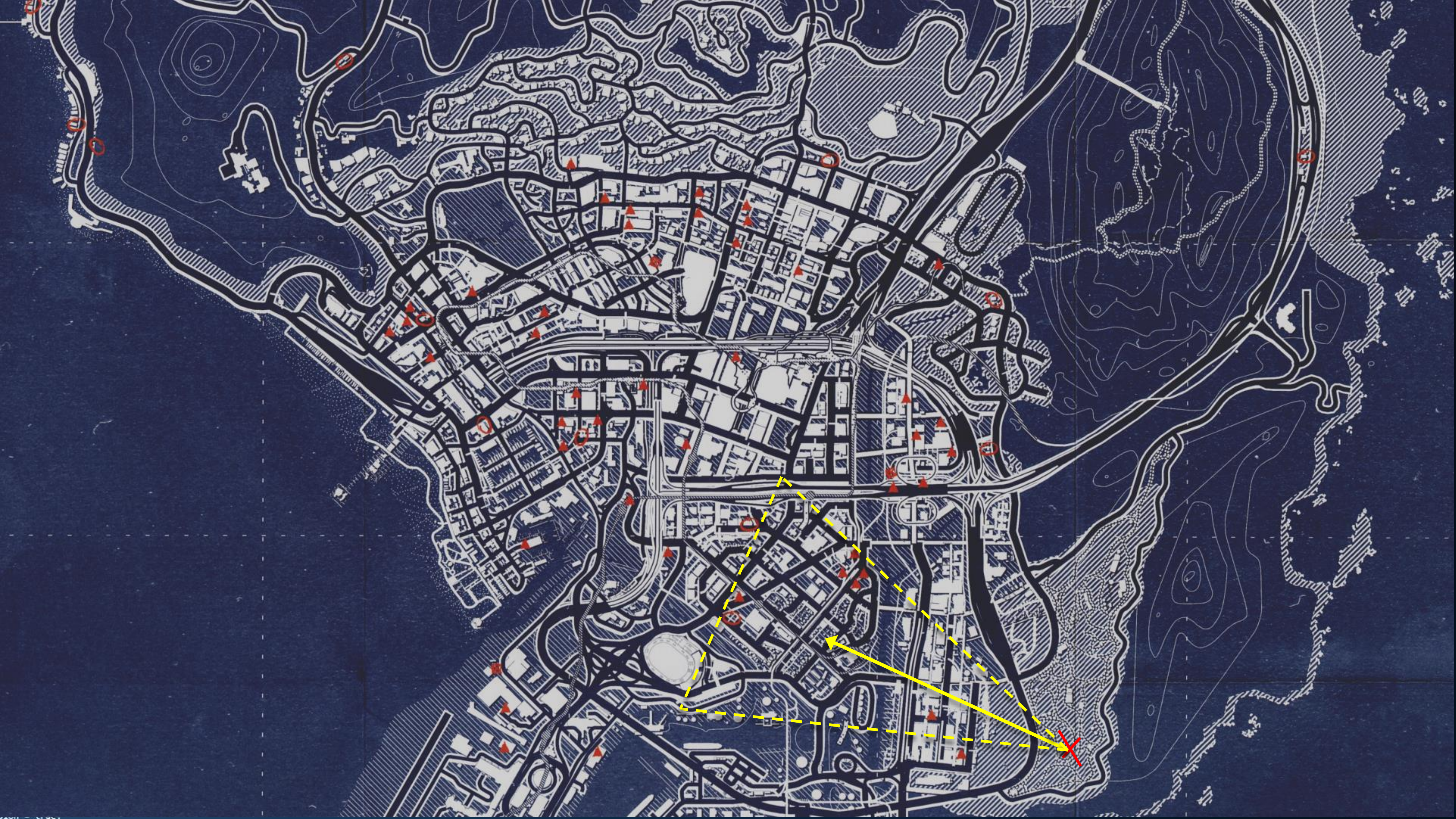
Tree requires little detail

City obscured by tree

Torso closer than ground

Tree between ground & sun



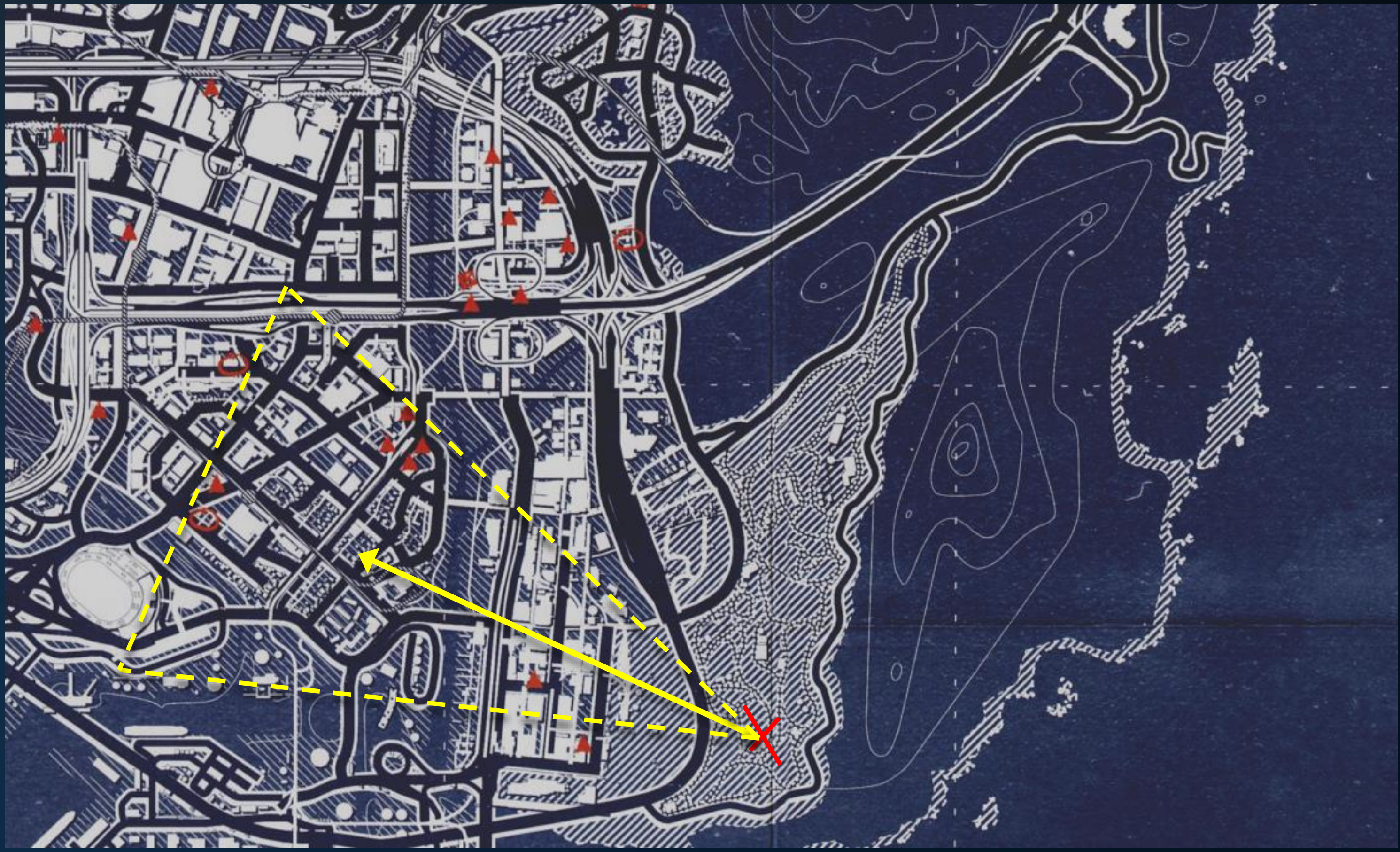


# Visibility

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

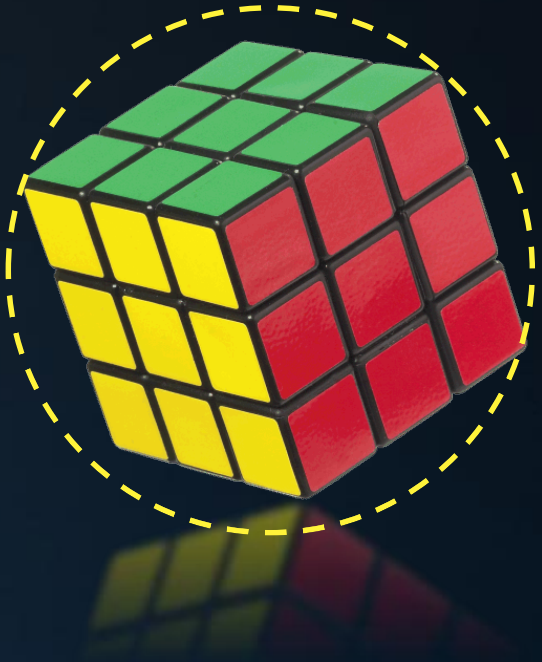
Culling

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

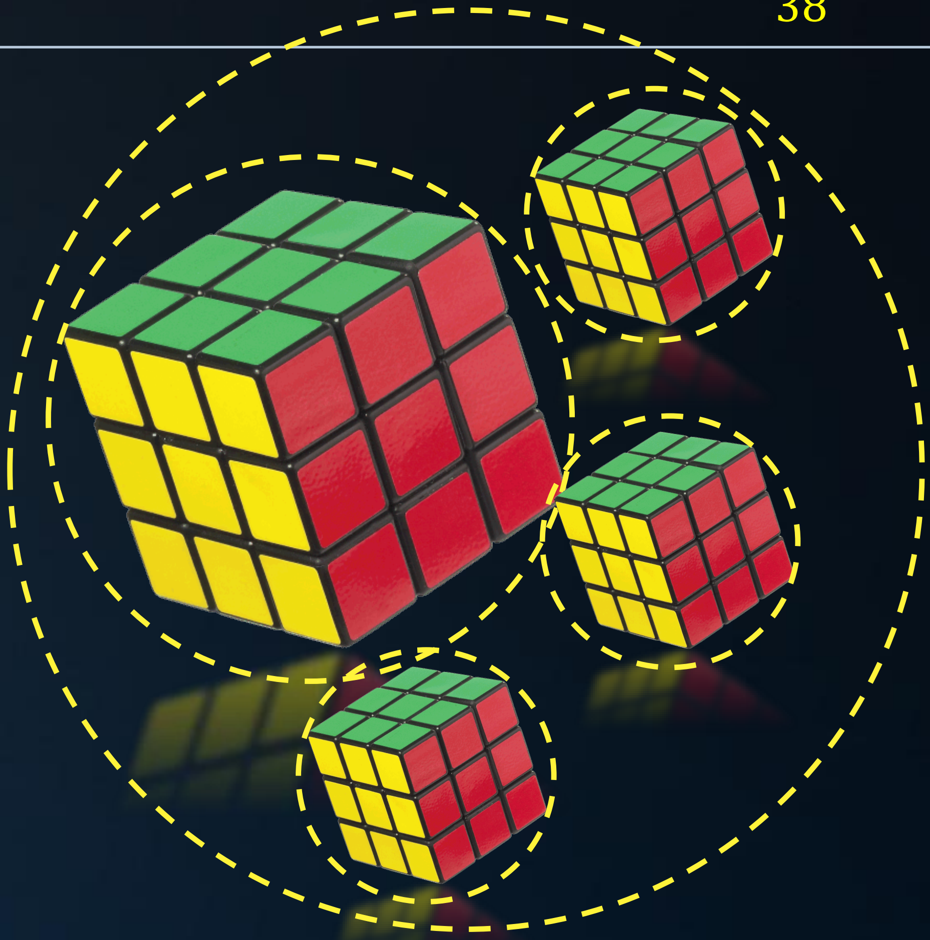
Culling

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.**

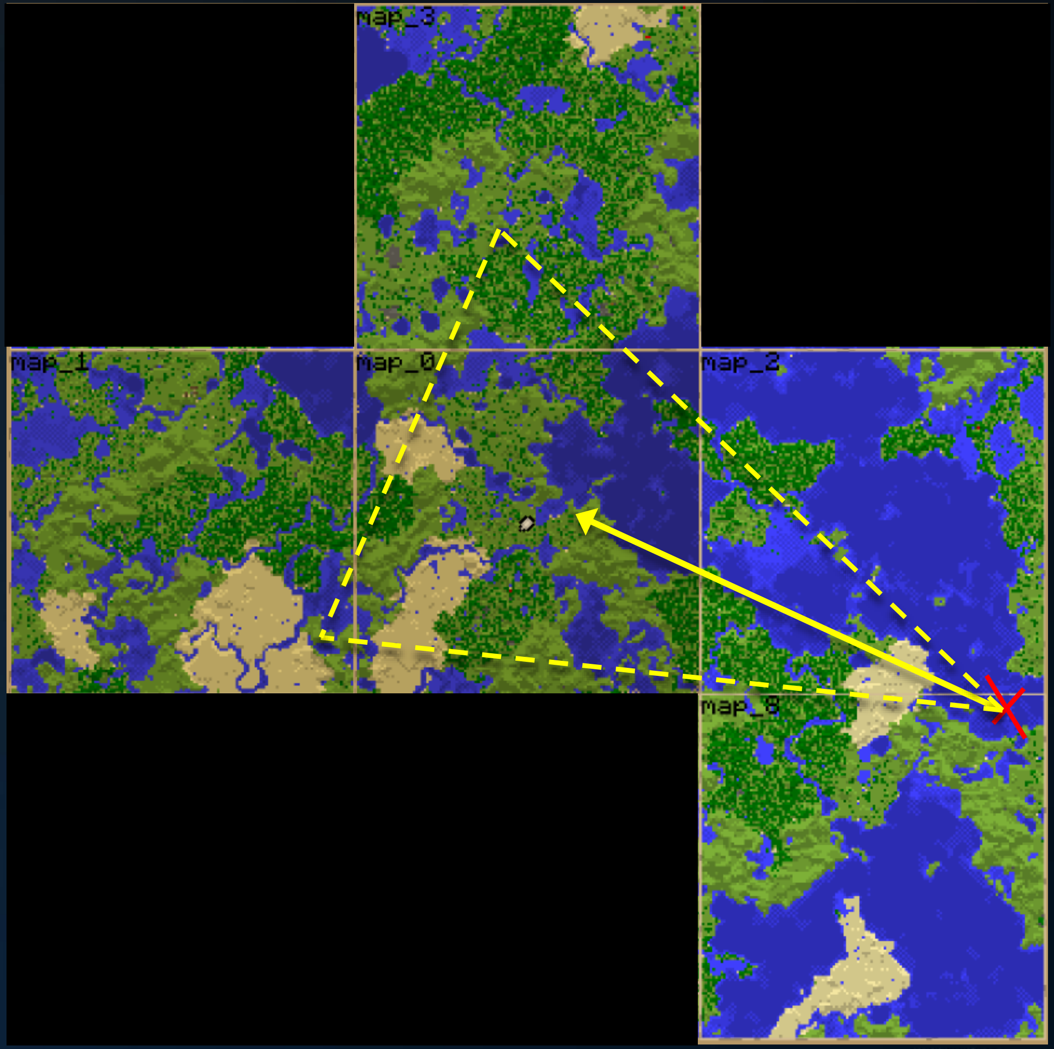


# Visibility

Culling

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

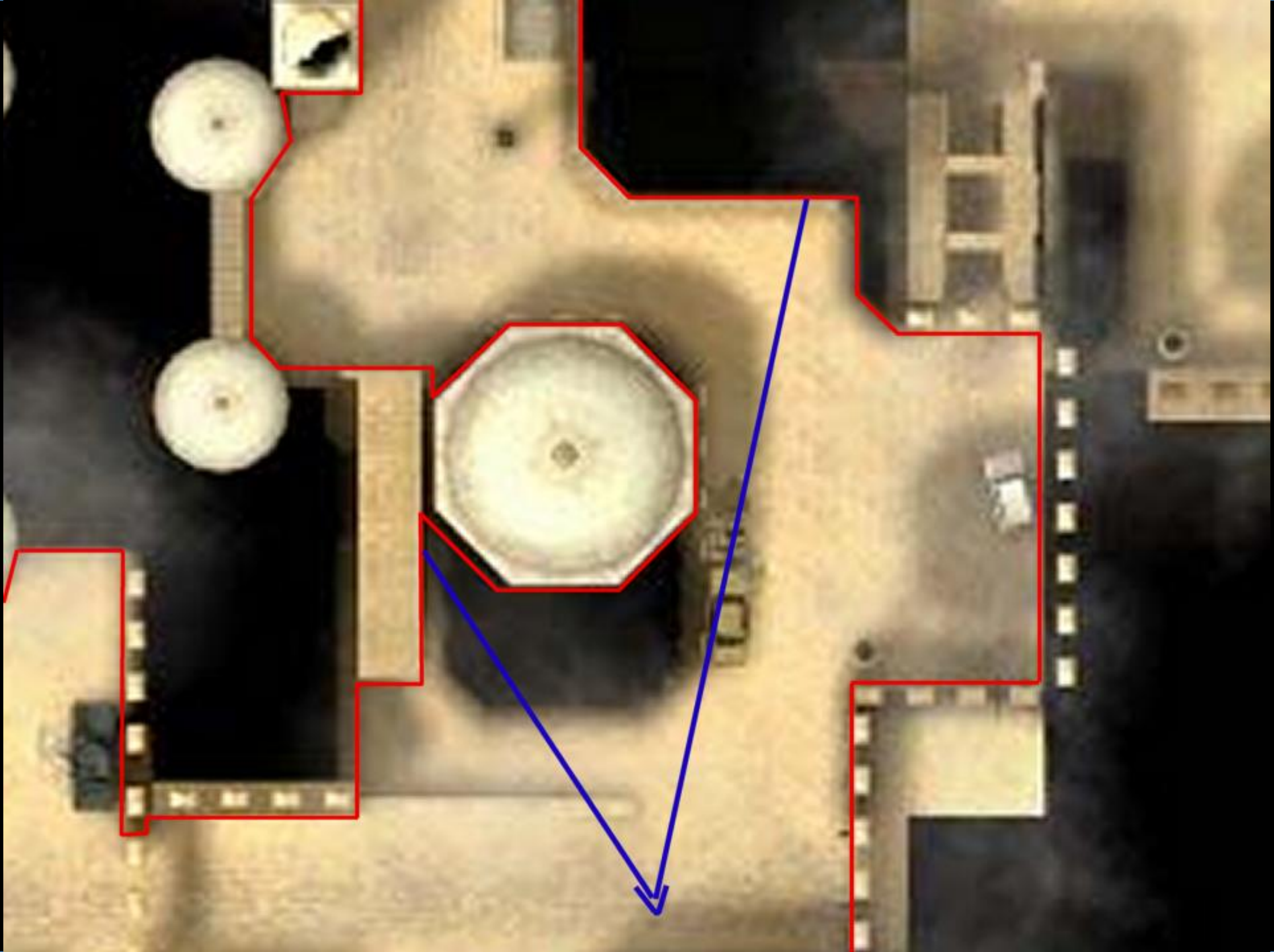
**Cost: 0 for out-of-range grid cells.**

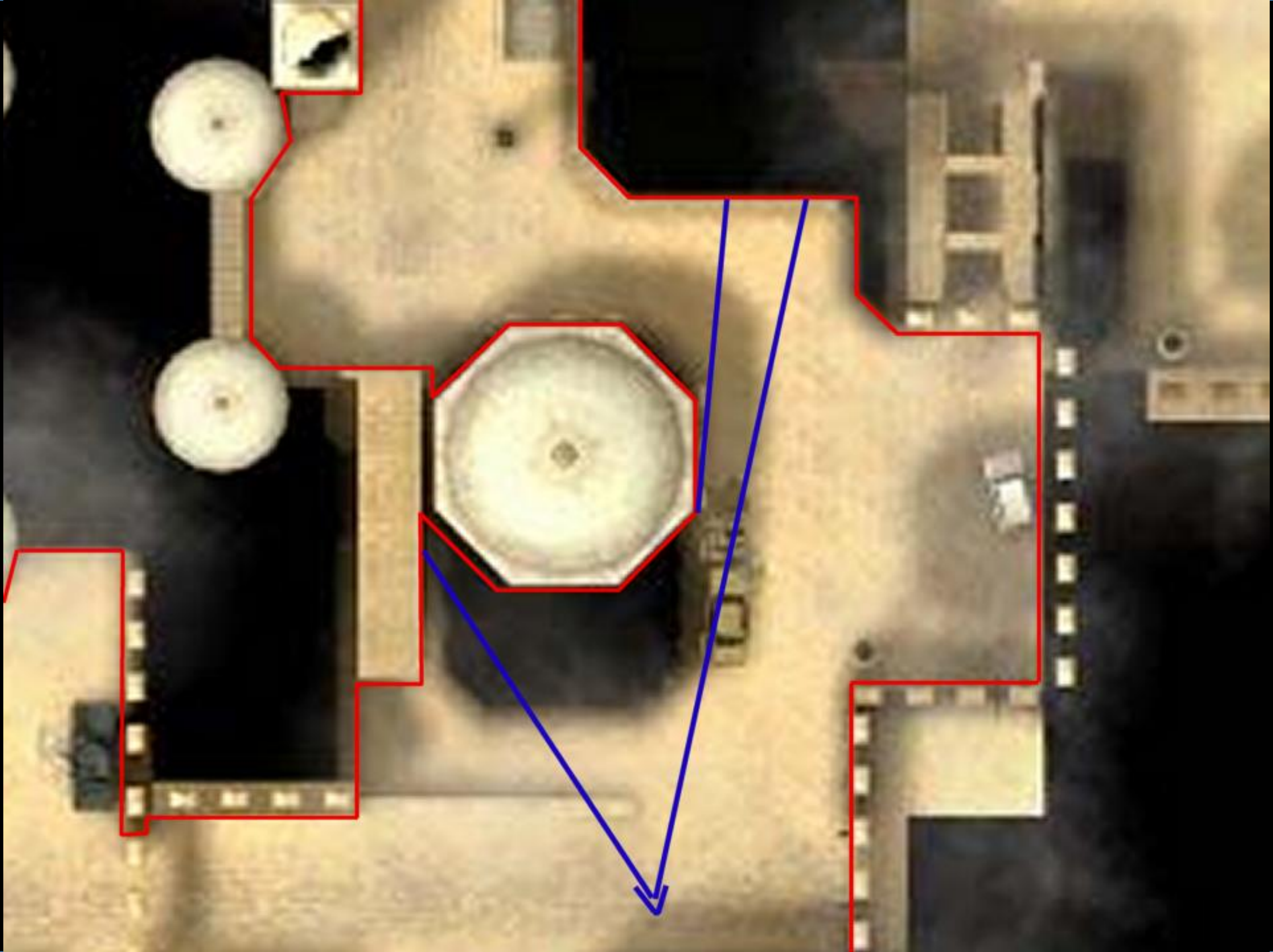


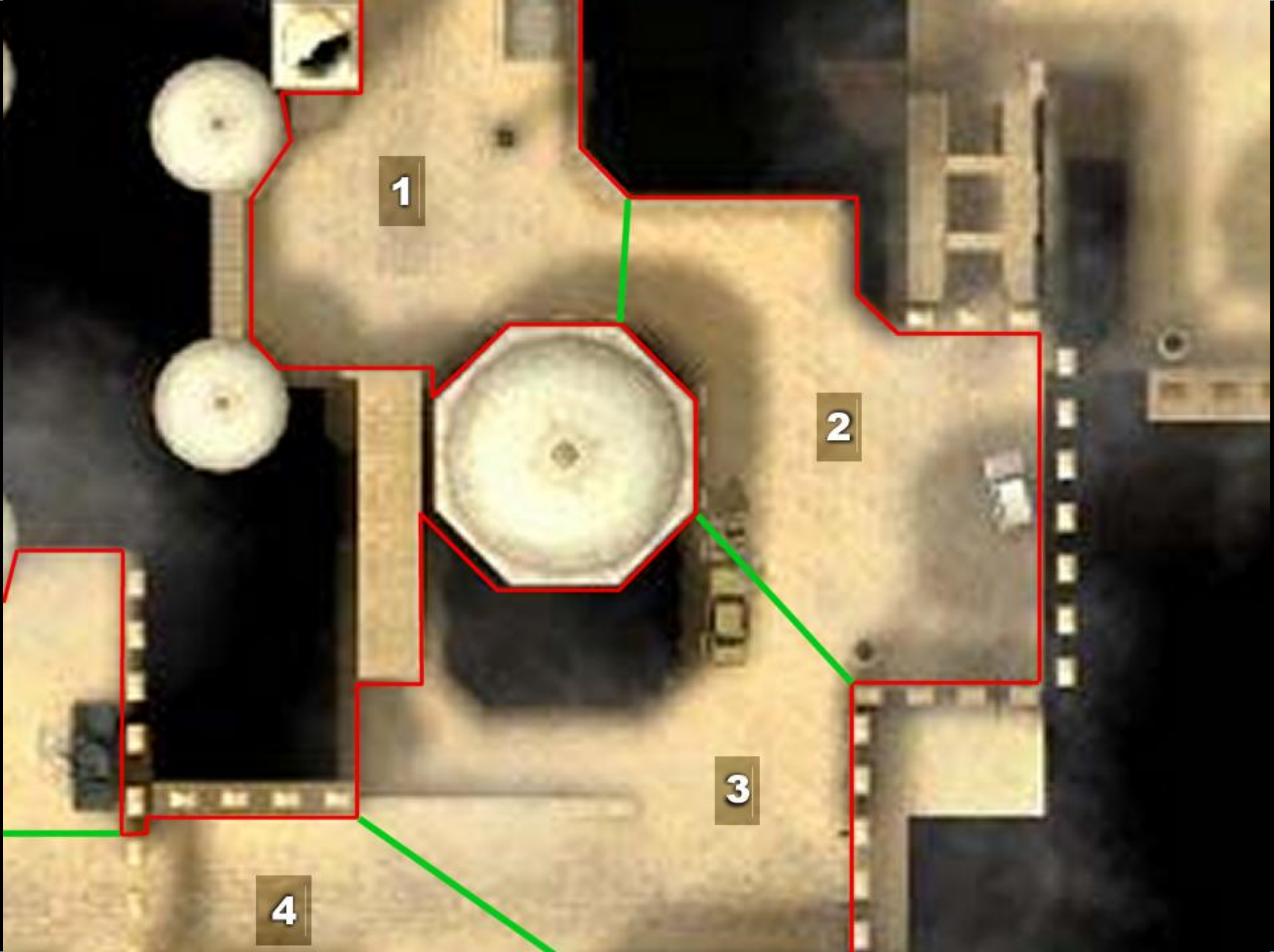
# Visibility

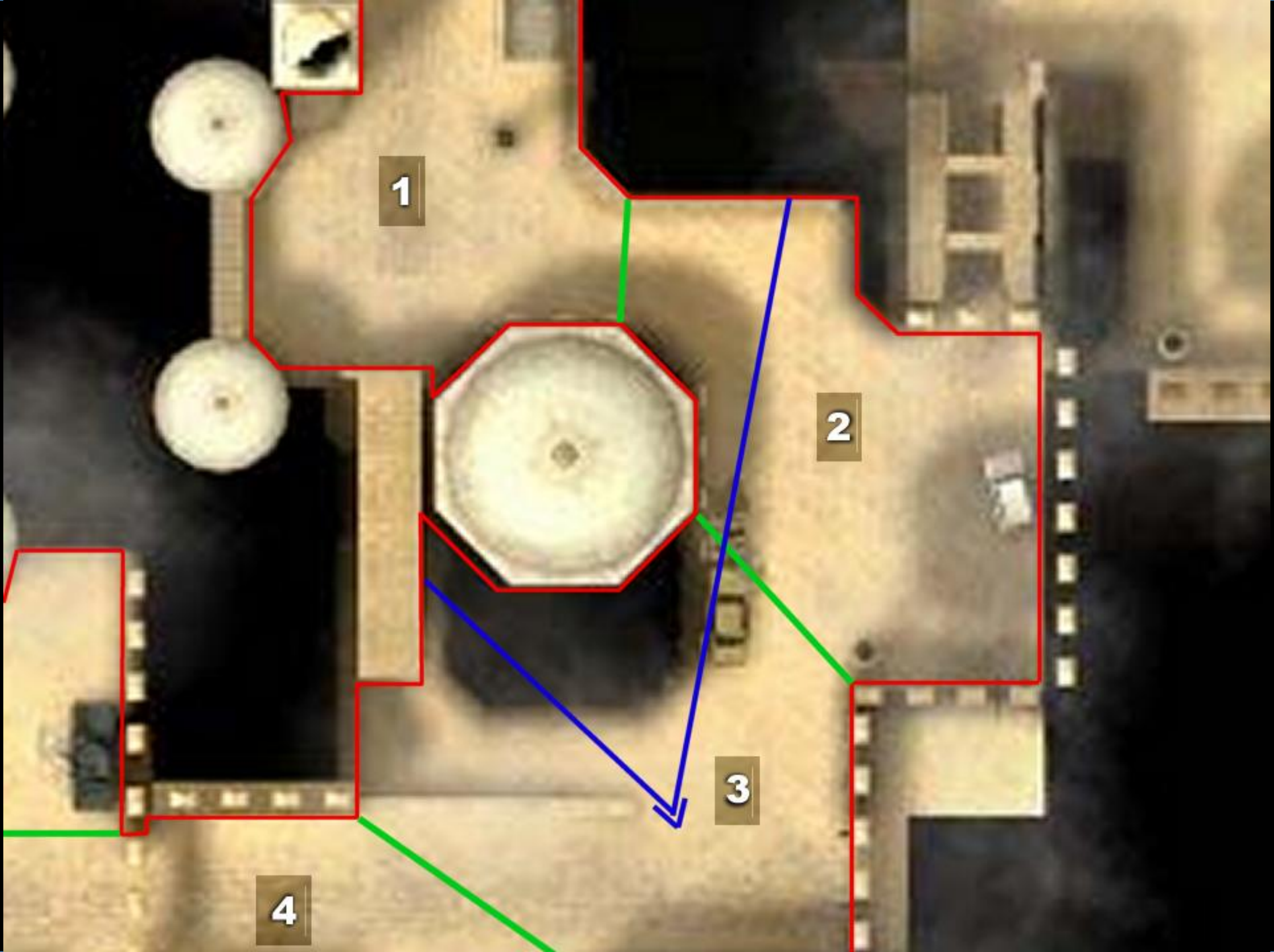
Indoor visibility: Portals

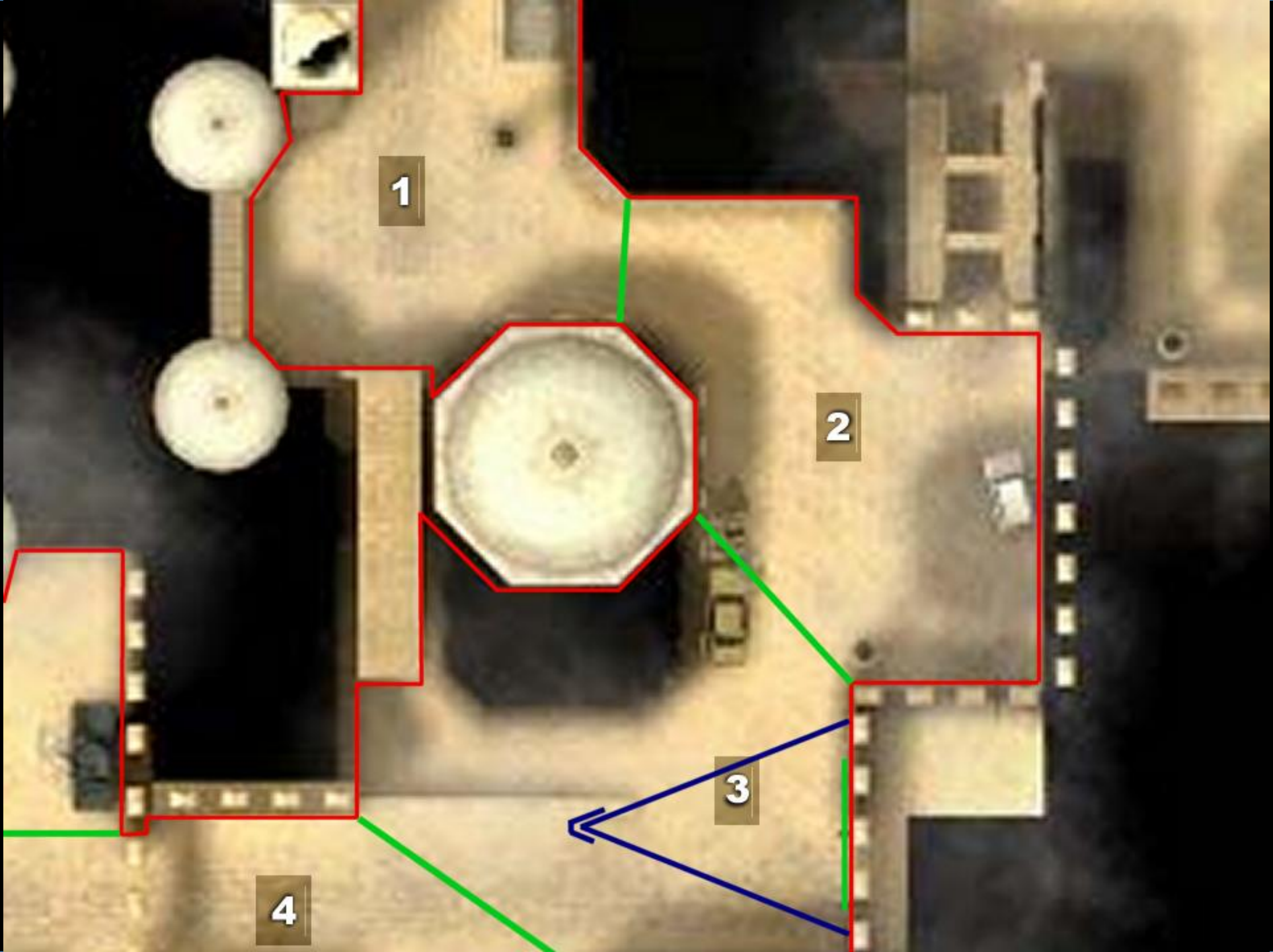
Observation: if a window is invisible, the room it links to is invisible.

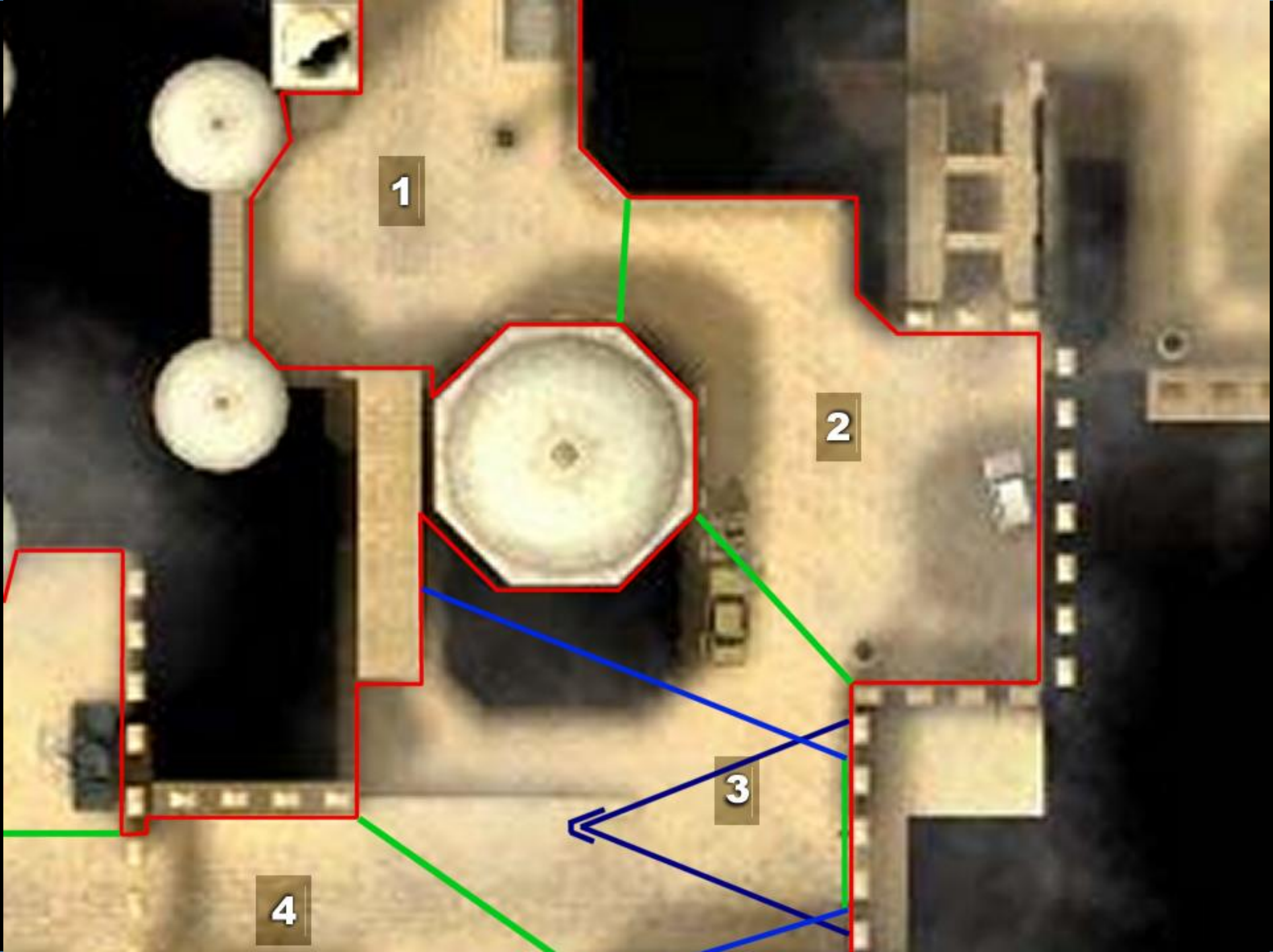




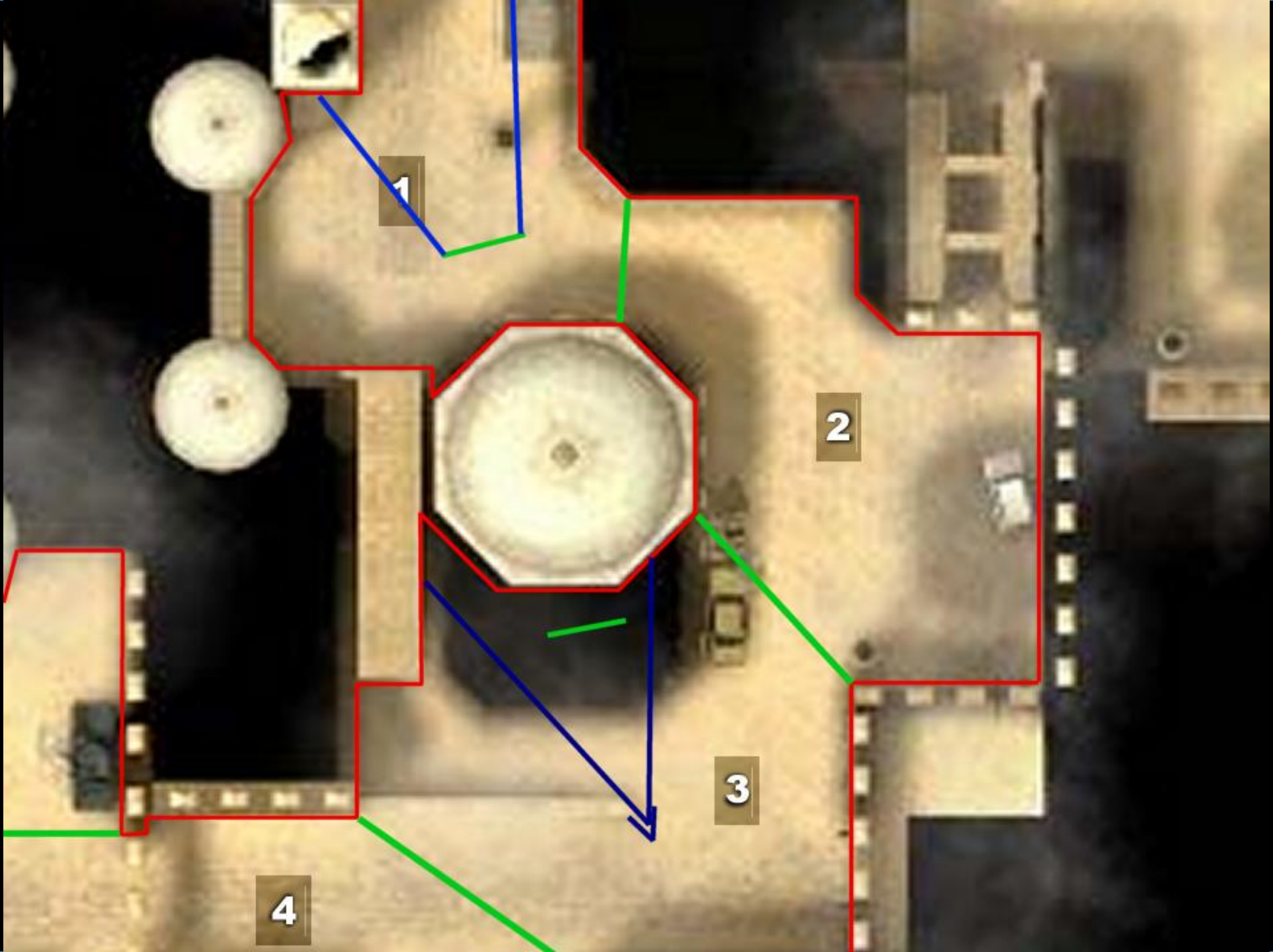

1
2
3
4


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

J. Bikker - April-July 2016 - Lecture 11: “Visibility”

# END of “Visibility”

next lecture: “Advanced Shading”