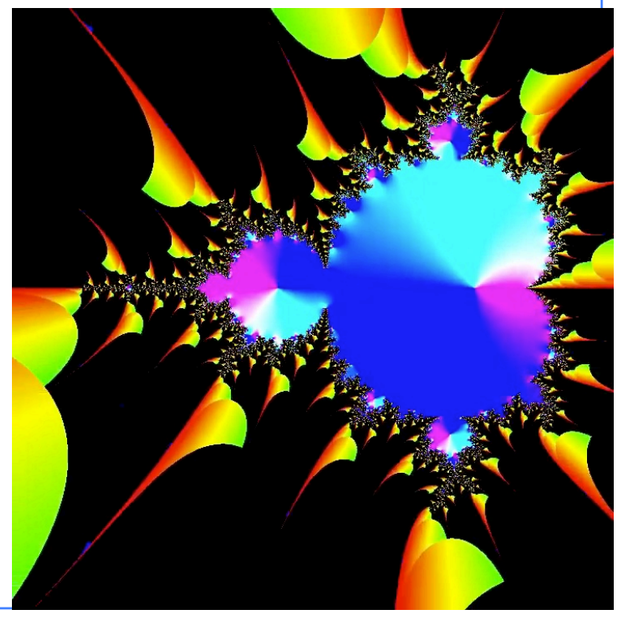
TSBK07/TSBK11/ETE378

Computer Graphics

Ingemar Ragnemalm, ISY





Lecture 7

Bump mapping

Normal matrix

Visible surface detection

Transparency

Lecture questions

1. What is the difference between bump mapping and normal mapping?

2. When is mat3() not enough for a proper normal matrix?

3. What problem is solved by Painter's Algorithm?

4. Why is ray marching faster than ray casting?

5. How can you test whether a point is inside a triangle?
Preferrably not a very slow test.

6. How can you perform picking (detect a click in a specific object)?

Dugga rules clarified

- You can do any number of duggas on the retake, one, all, or some
- You can *not* lose points on the retake! If you score lower than the previous, the highest counts!
- TSBK07: 20 points to pass, 30 for 4, 40 for a 5
- TSBK11: 14 points to pass, 21 for 4, 28 for a 5

Good projects can get bonus points.

TSBK07: Good = 5 points. Excellent = 10 points

TSBK11: Good = 4 points. Excellent = 7 points

Lecture plan changes

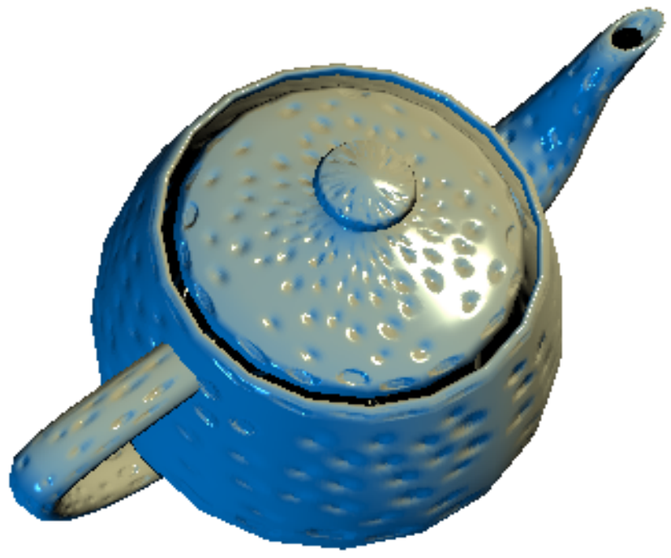
Some issues moved and simplified to free up space for this:

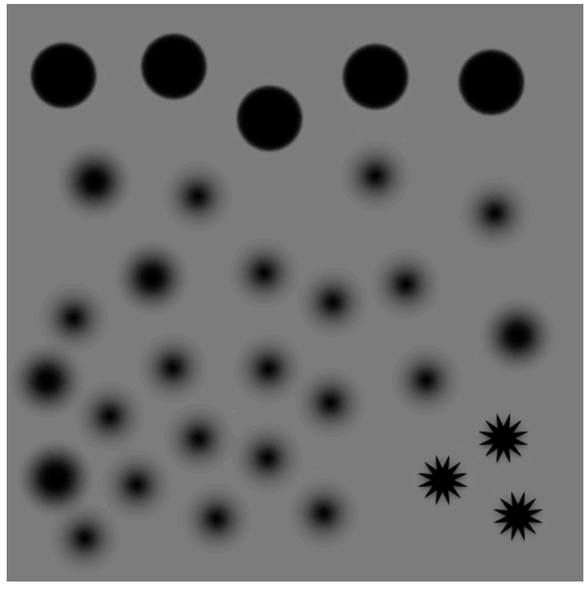
Last lecture: AI image generation using the diffusion algorithm.
Guest lecture by dr Souad Mohaoui.

Instructions on relevant chapters for the duggas will be updated.

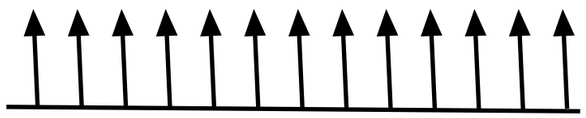
Bump mapping

Simulates surface structure by manipulating the normal vector





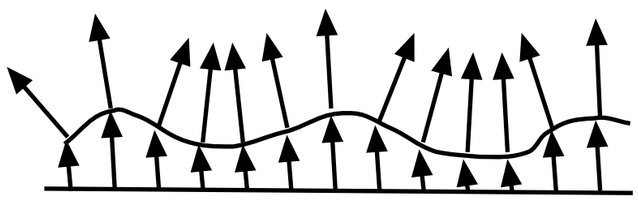
Bump mapping - model



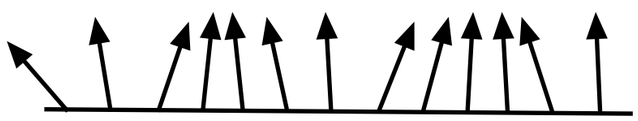
Surface with normal vectors

Bump map: scalar function of the texture coordinates

Modulate the surface by the bump function, along normal



Calculate new normals



Resulting normal vectors

Bump mapping - the coordinate systems

Input:
A point **p**, normal vector $\hat{n}$
Texture coordinates s(**p**), t(**p**)
Directions of texture coordinate system $\hat{s}$, $\hat{t}$
The bump function b(s,t)

Calculate the partial derivative of the bump function, b_s and b_t

$$\mathbf{n'} = \hat{n} + b_t * (\hat{s} \times \hat{n}) + b_s * (\hat{t} \times \hat{n})$$

or, if $\hat{s}$, $\hat{t}$, $\hat{n}$ are orthogonal

$$\mathbf{n'} = \hat{n} + b_s * \hat{s} + b_t * \hat{t}$$

Texture coordinate system

$\hat{n}$ = normal vector
$\hat{s}$ = tangent
$\hat{t}$ = bitangent

How do we find the $\hat{s}$ and $\hat{t}$ vectors? We have the texture coordinates but no coordinate system!

Cross product with normal vector? With what?

Faking it

Cross product with absolutely anything!

$$\hat{s} = \hat{x} \times \hat{n} / |\hat{x} \times \hat{n}|$$

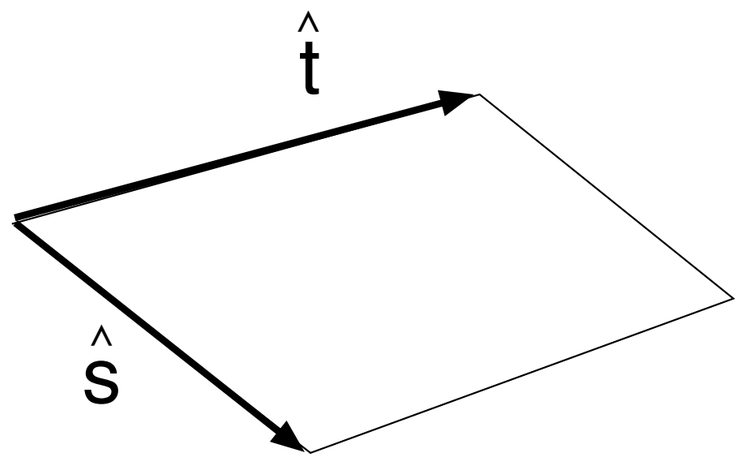
$$\hat{t} = \hat{n} \times \hat{s}$$

Works for some cases. (Noise bump maps in particular.

But we can do better!

Trivial geometry

Very easy for a cube or a plane. Comfortable test case.



Waves on water

Bump maps on flat surfaces

Coordinate system known

Modify normals with sin waves (or better)

Full general-pupose methods

Analyze s and t variation over each triangle to derive the $\hat{s}$ and $\hat{t}$ basis vectors

Example: Lengyel's method

(Advanced course.)

Normal mapping

Normal mapping = bump mapping with pre-calculated b_s and b_t!

Same result, an optimization of bump mapping.

Bump mapping conclusions

Good example of dealing with multiple coordinate systems

Transforming directional vectors is vital.

Good effects can be made even for some simple cases.

Now, let's return to that normal matrix...

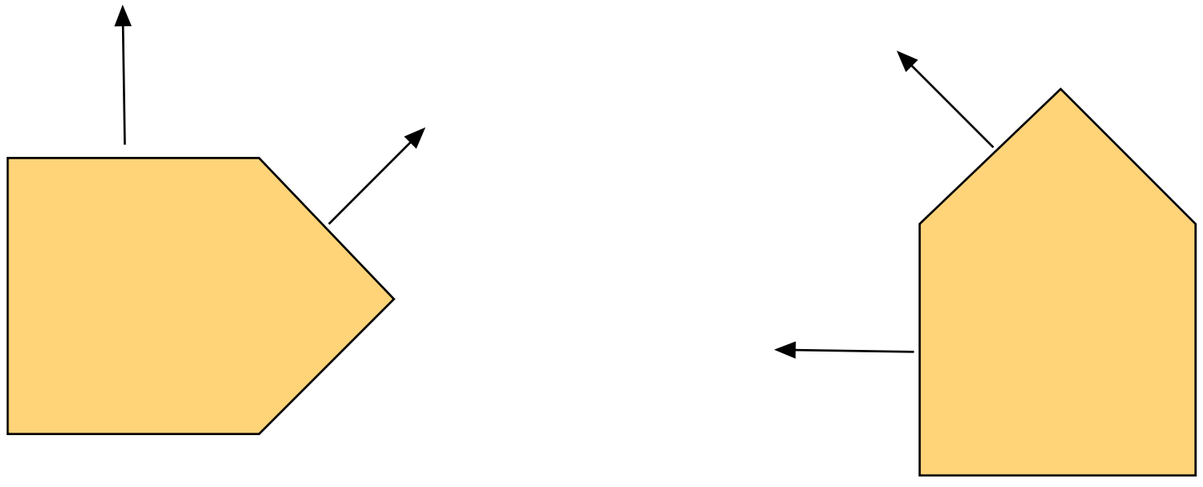
Normal vectors must be rotated with the model.

```
...
in  vec3 inNormal;
...
out vec3 exNormal; // Phong

uniform mat4 modelviewMatrix;
...

void main(void)
{
   exNormal = mat3(modelviewMatrix) * inNormal;
   gl_Position = projectionMatrix * modelviewMatrix * vec4(inPosition, 1.0);
}
```

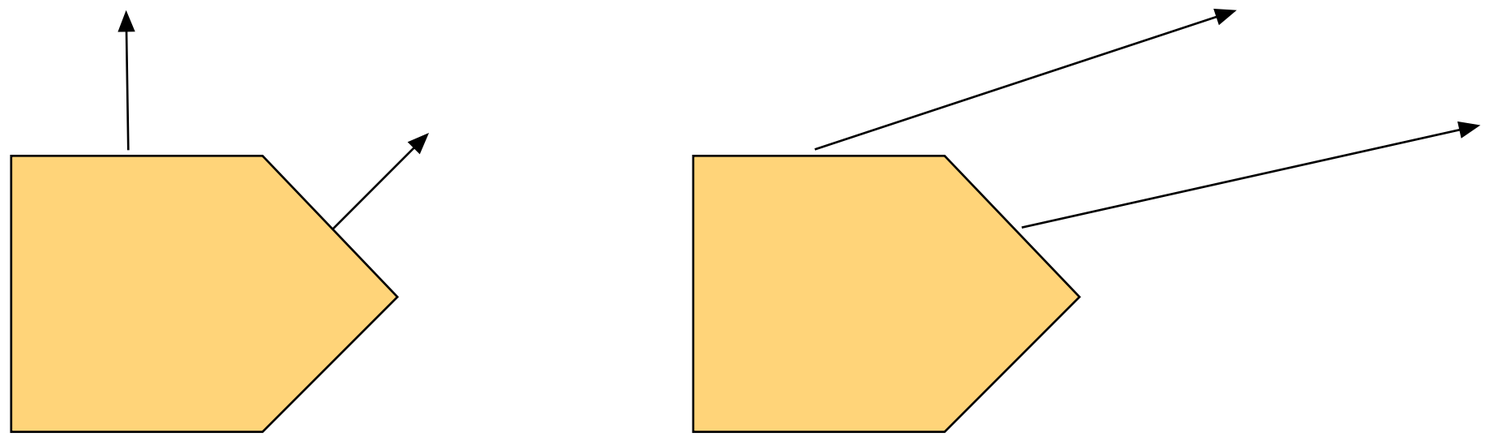
Parts of a typical vertex shader

The Normal matrix

When placing a model in the world, normals must be rotated...



but they must not be translated...



so we just cast to mat3, right?

$$\begin{bmatrix} r & r & r & t_x \\ r & r & r & t_y \\ r & r & r & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} r & r & r \\ r & r & r \\ r & r & r \end{bmatrix}$$

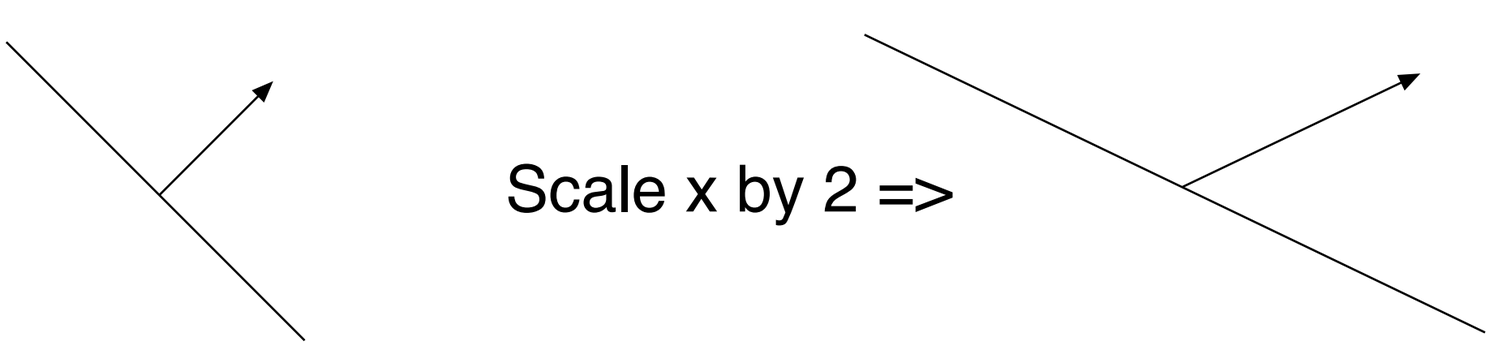
or we zero the translation part:

$$\begin{bmatrix} r & r & r & t_x \\ r & r & r & t_y \\ r & r & r & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} r & r & r & 0 \\ r & r & r & 0 \\ r & r & r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It works most of the time... but...

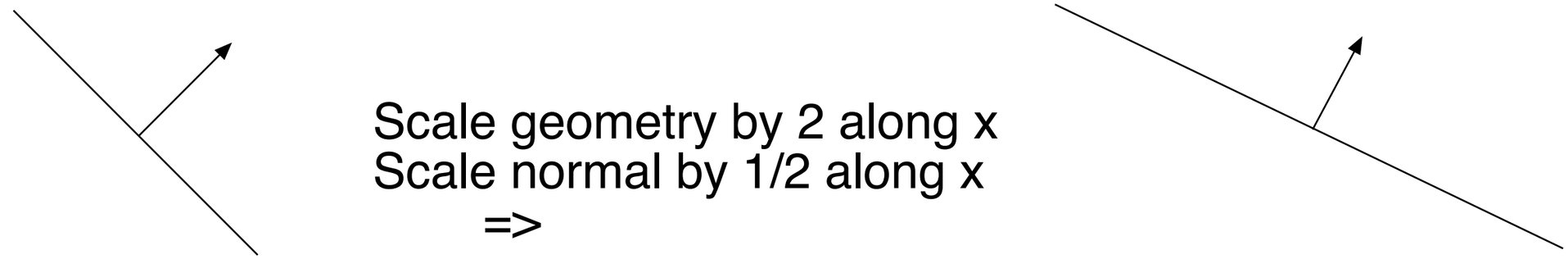
But wait!

For non-uniform scaling, this does not work!



The normal vector is no longer perpendicular to surface!

But what if we do the opposite...



Scale geometry by 2 along x
Scale normal by 1/2 along x
=>

Suddenly things look better...

but what happens if we mix in rotations?

Normal matrix, full solution

Invert scaling, keep rotation

1) Invert to reverse both
2) Transpose to reverse (restore) rotation

=> Use inverse transpose of rotation part

$$N = (M^{-1})^T$$

Normal matrix, full solution

Invert scaling, keep rotation

1) Invert to reverse both
2) Transpose to reverse (restore) rotation

=> Use inverse transpose of rotation part

$$N = (M^{-1})^T$$

Please don't miss this!

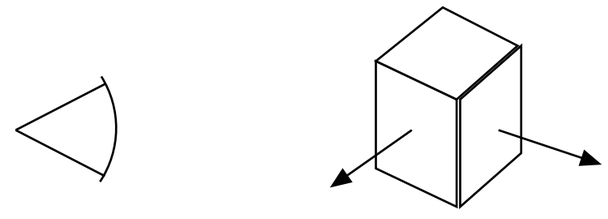
More visible surface detection

sometimes applicable to entirely different problems!

Backface culling

Object space method

Removes all polygons that are "looking away" from the camera.

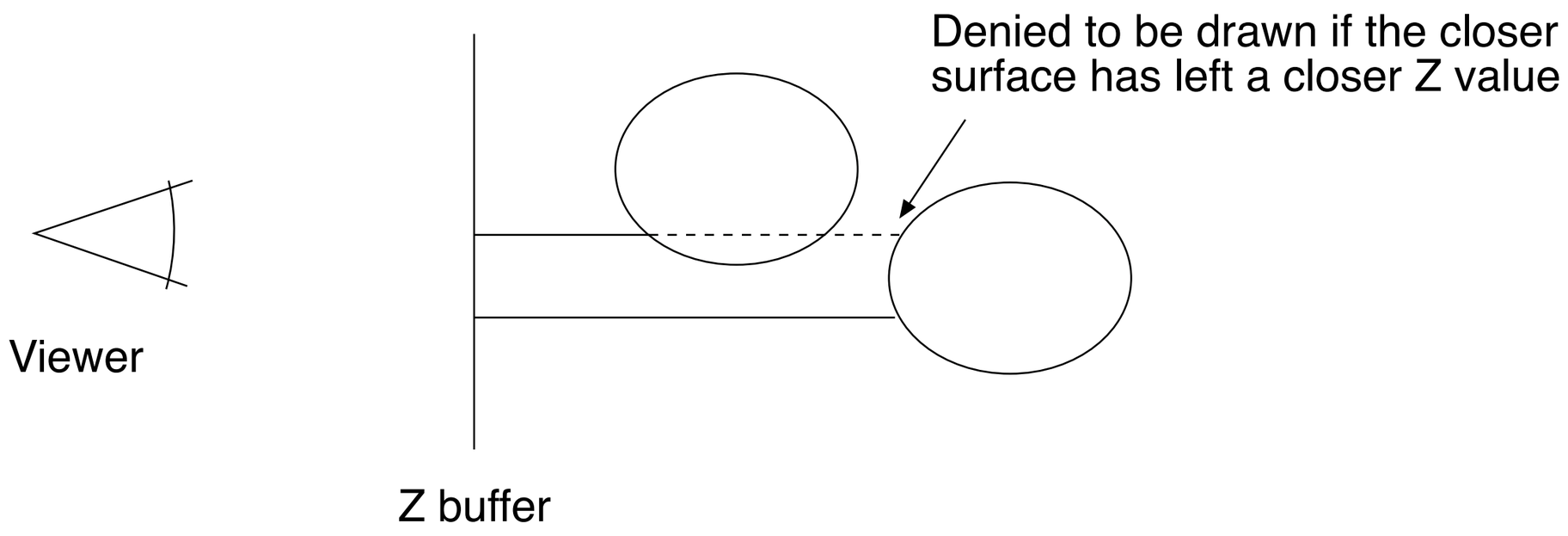


Removes ≈50% of all polygons that would otherwise be in view!

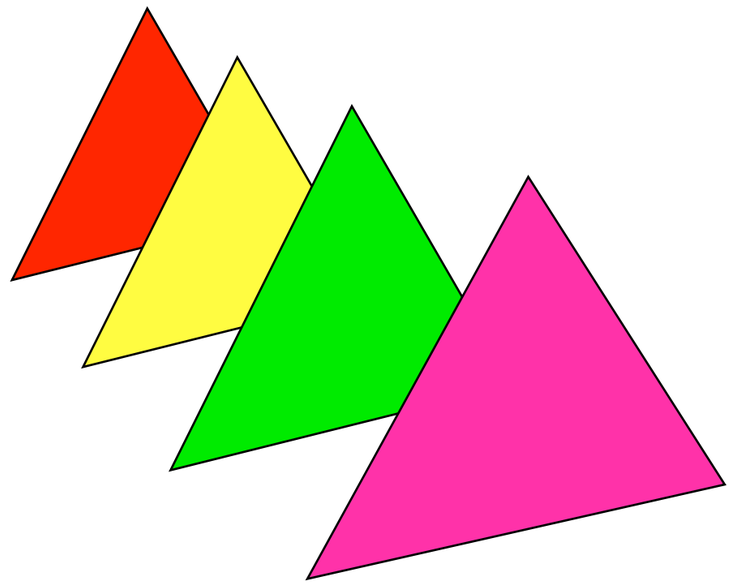
Z-buffer

Depth-buffer method

Only draw if a pixel is closer than the closest drawn before. Z value saved in the depth buffer, the "Z buffer"



Painter's algorithm
Depth-sorting method



Render from back to front

Both image and object space method

Painter's algorithm

Sorting on polygon level.

But some scenes can not be sorted at all!

Solution: Figure out a way to split polygons to resolve the sort. But how?

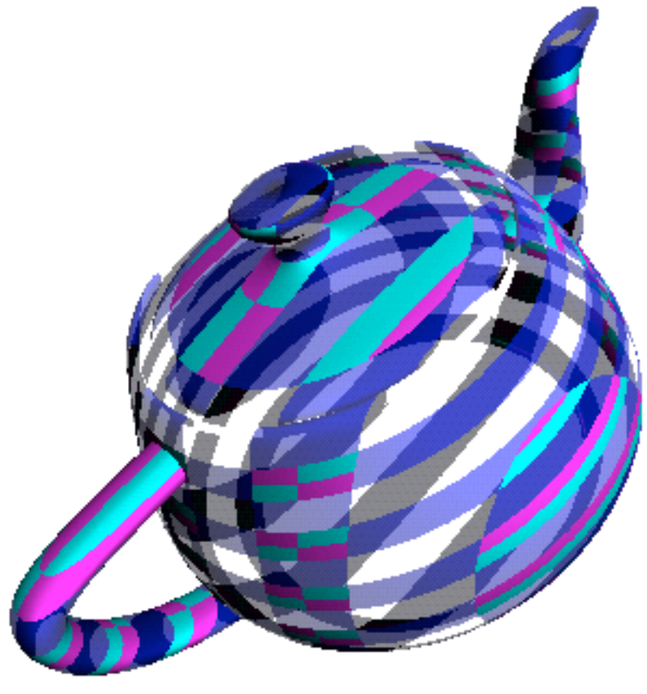
Painter's algorithm

- Slow – may paint many pixels more than once
- Slow and complicated in its full form - can be solved with BSP trees
- Practically useful at object level, sorting transparent objects only
- Approximative sorting is often sufficient

Drawing with transparency

A (alpha) in RGBA can be used for transparency

Alpha values exist in textures as well as color etc



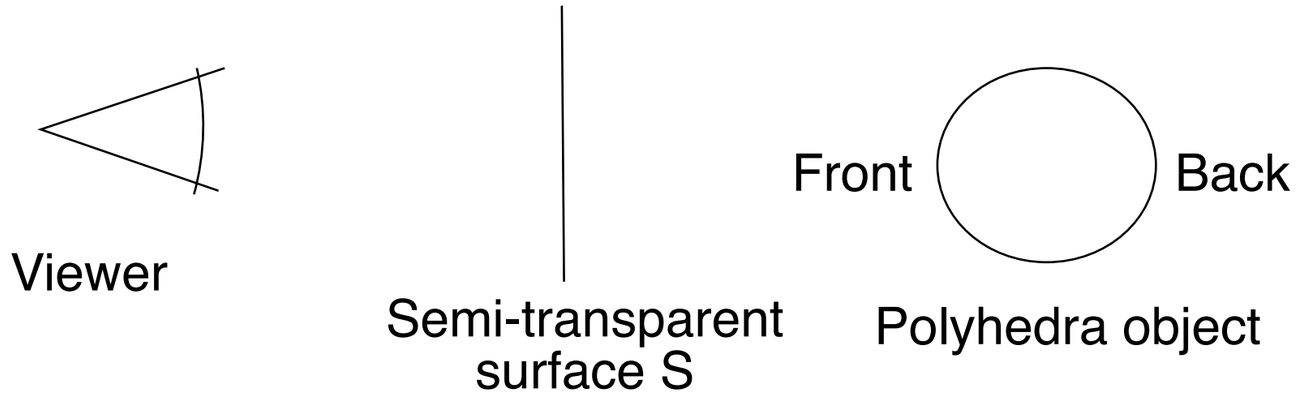
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);
(glBlendEquation for even more control)

dest = source*α + dest*(1-α)

Note that alpha does not have to be taken from the source!

Problem: Drawing order causes problems with Z-buffer!

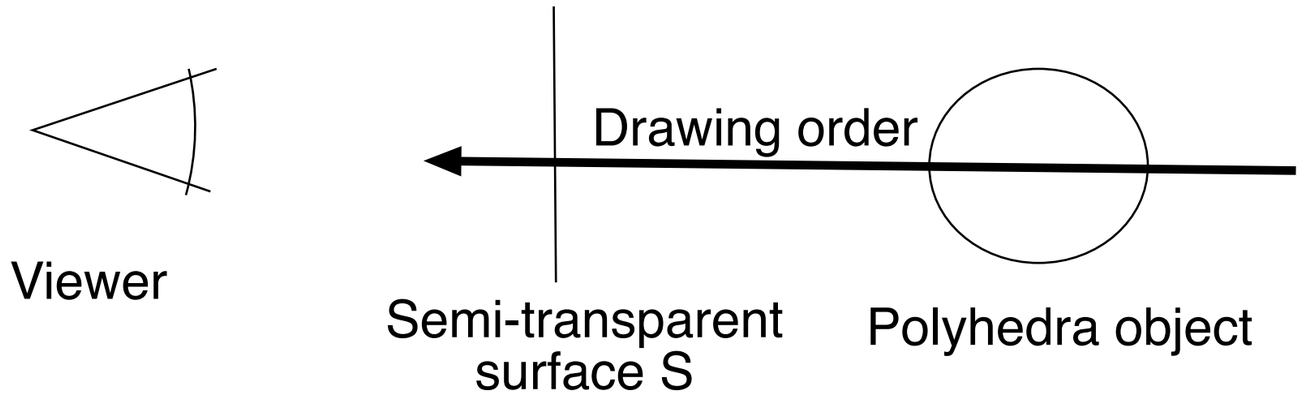
The Z-buffer problem with transparency



If S is drawn first, the other scenery will not be drawn!

For a single object, its inside will be obscured by its front.

The Z-buffer problem, solutions

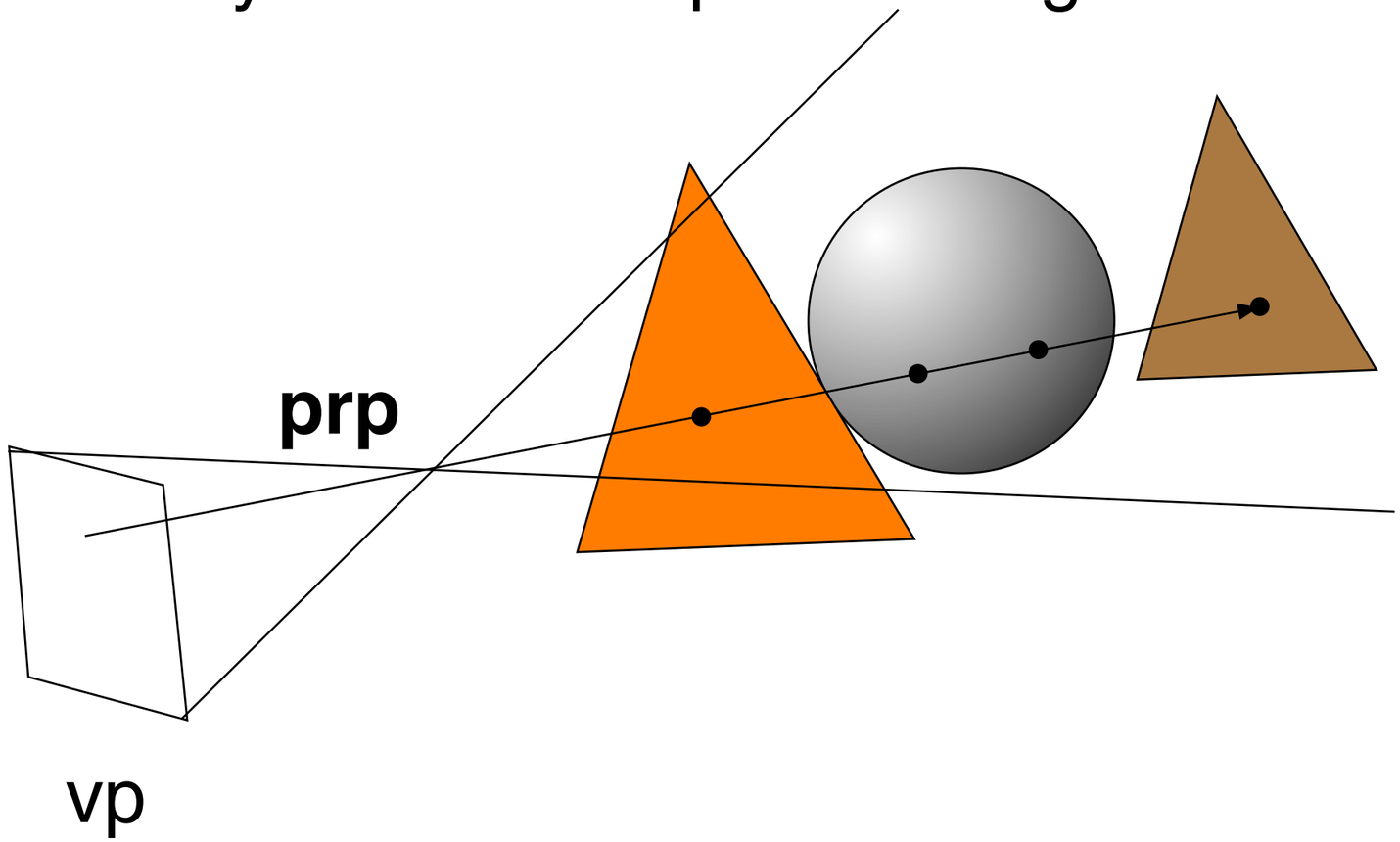


Solution for entire scene: Draw the scene back-to-front. ”Painter’s algorithm”

For a single object, draw its inside first, front later. Can be done with culling.

Ray-casting

Follow rays from each pixel through the scene



Full 3D raycasting

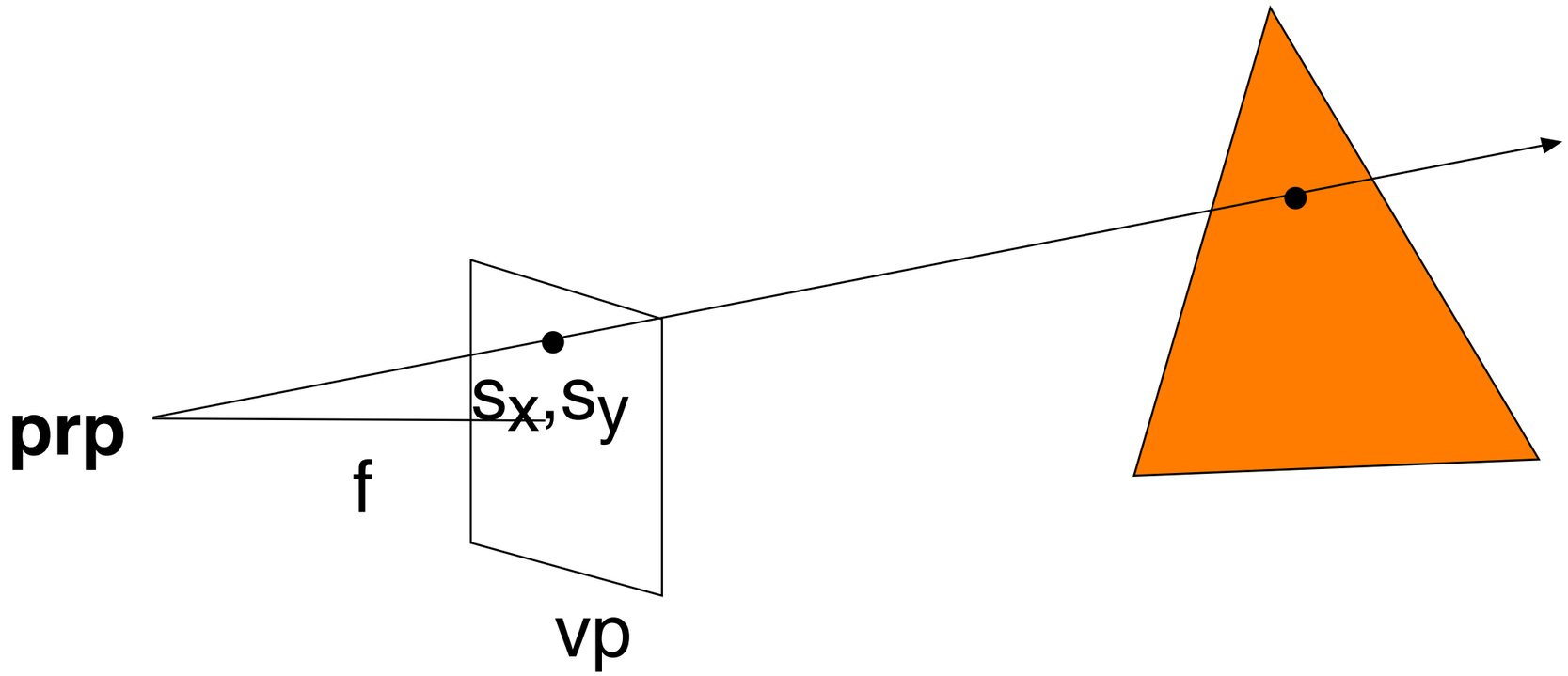
for every pixel (x,y) in the image

calculate a ray from the pixel through the camera (**prp**) and through the scene

calculate intersections with all objects in the scene

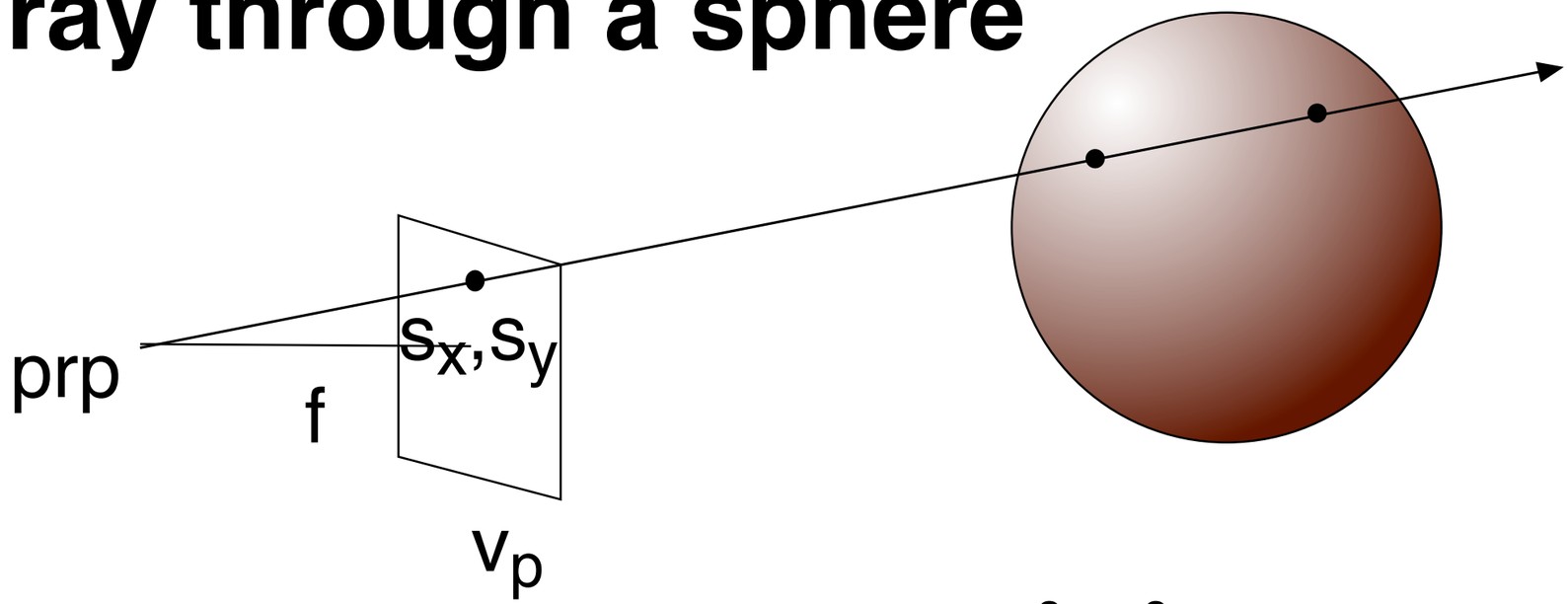
the pixel value is calculated from the closest intersection found

The ray



Line equation: **prp** + $\mu(s_x,s_y,-f)$

A ray through a sphere

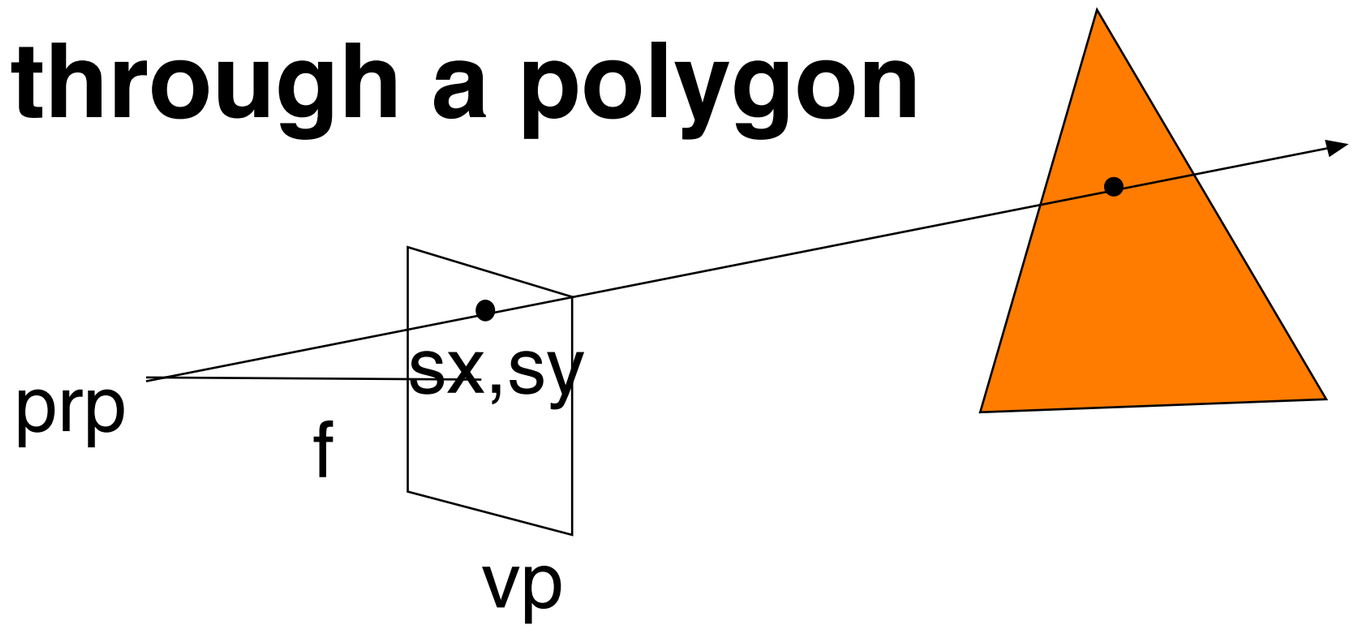


Combine line equation with sphere equation: $|\mathbf{p} - \mathbf{c}|^2 = r^2$

p = point on sphere
c = center
r = radius

Find **p**. Solves to two cases or none.

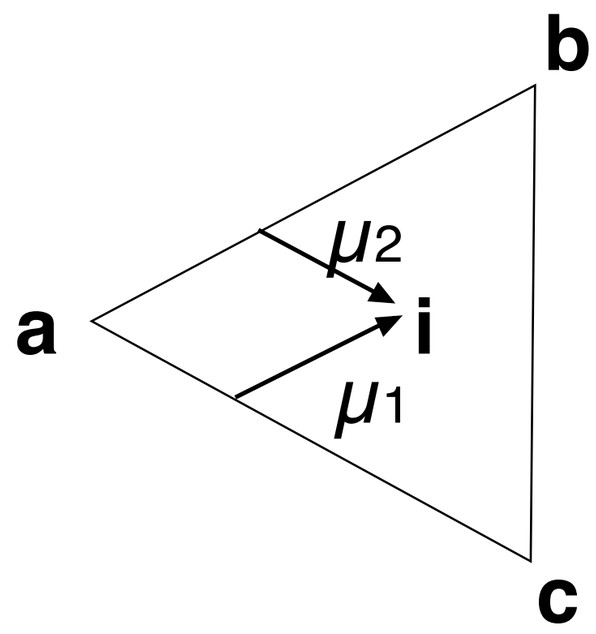
A ray through a polygon



- Are we actually crossing the plane?
- Where do we intersect the plane?
- Are we inside the polygon?

Is the point in the polygon?

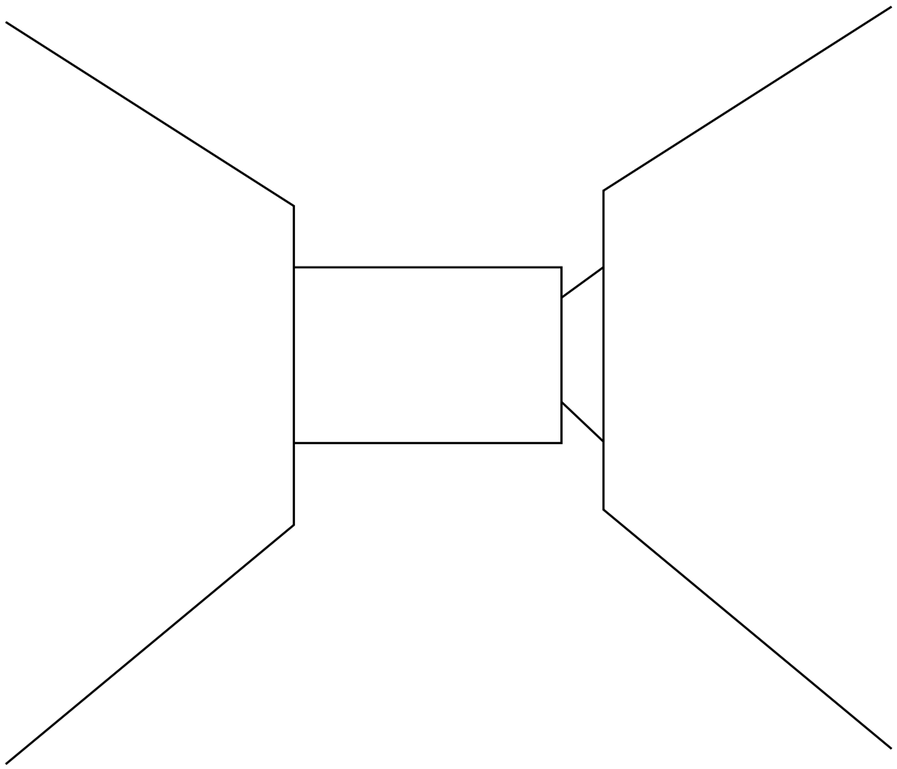
Triangle: Straight-forward
Several methods possible.

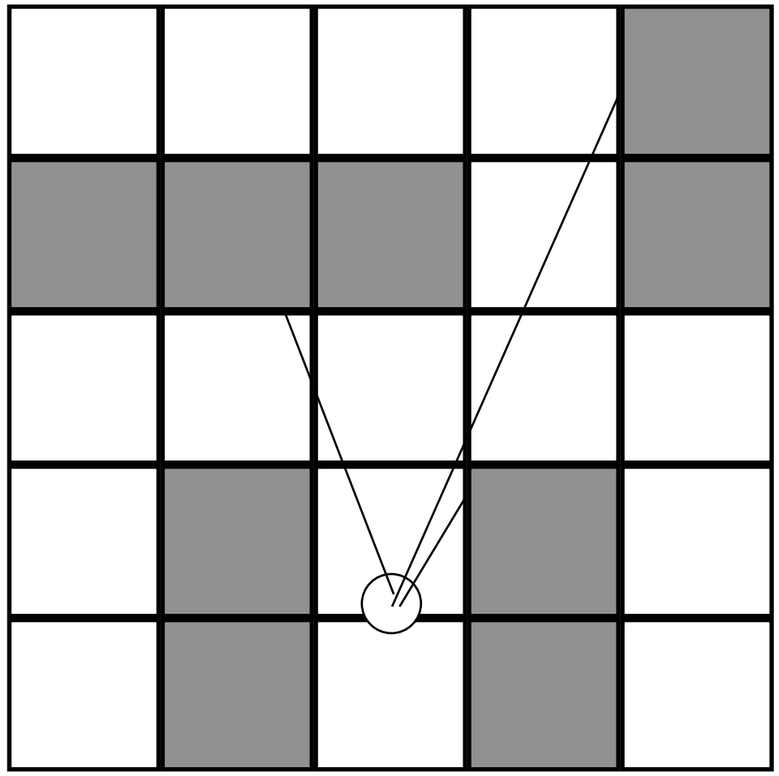


$$\mathbf{i} = \mathbf{a} + \mu_1 * \mathbf{ab} + \mu_2 * \mathbf{ac}$$

$$0 < \mu_1$$
$$0 < \mu_2$$
$$\mu_1 + \mu_2 < 1$$

Raycasting in a grid: Ray marching



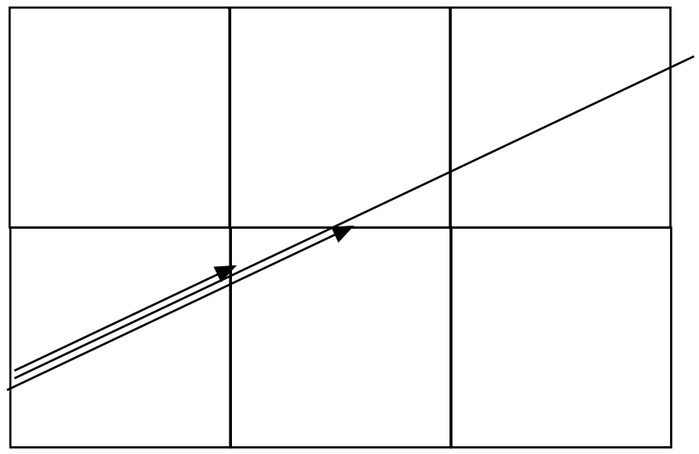


Ray marching relatively easy

Step to next potential voxel wall (3 possible in 3D)

Pick the closest, check neighbor space

Repeat until filled space is found.

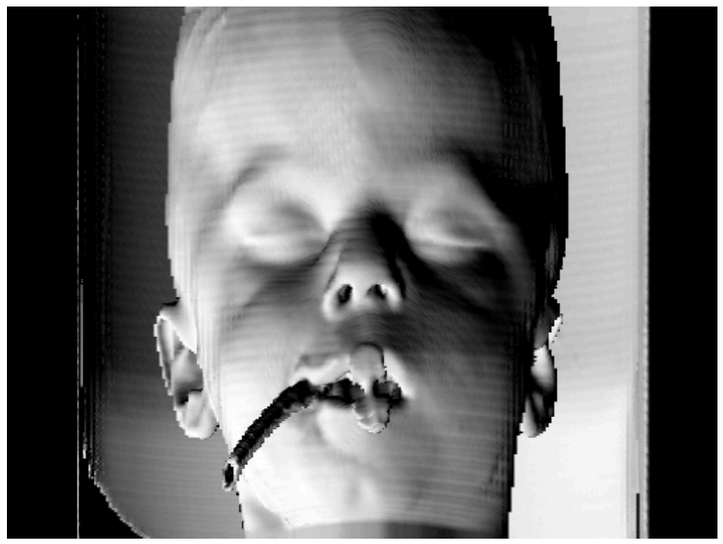


Essentially a line drawing algorithm!

Ray-casting applications

- VSD in 2D or 3D grids
- Visibility tests for AI
- Visibility tests for global illumination
- First step of ray-tracing
- Picking

Example application



Medical visualization of CT scans

CT and MRI scanners output 3D volumes.

Ray-marching in the 3D volume common method for visualizing the result.

Next time

Picking

Rotation around arbitrary axis

Large worlds part 1