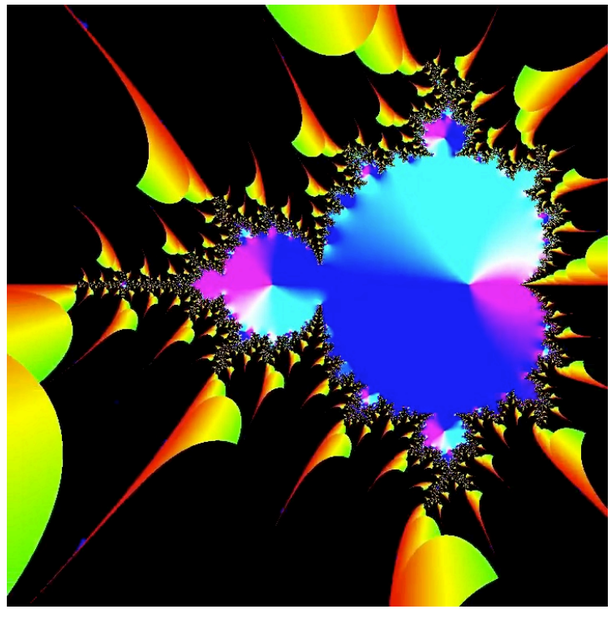
TSBK07/TSBK11

Computer Graphics

Ingemar Ragnemalm, ISY



Lecture 7

More mappings: More bump mapping, texture splatting, triplanar texture mapping

Normal matrix

Ray casting

Picking

Lecture questions

1. How can you choose between more than 3 textures in a splat/blend map?

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

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

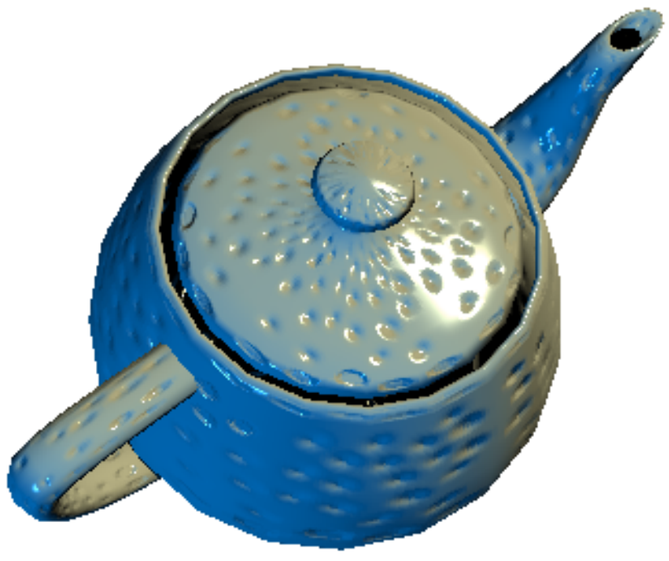
4. What problem is solved by Painter's Algorithm?

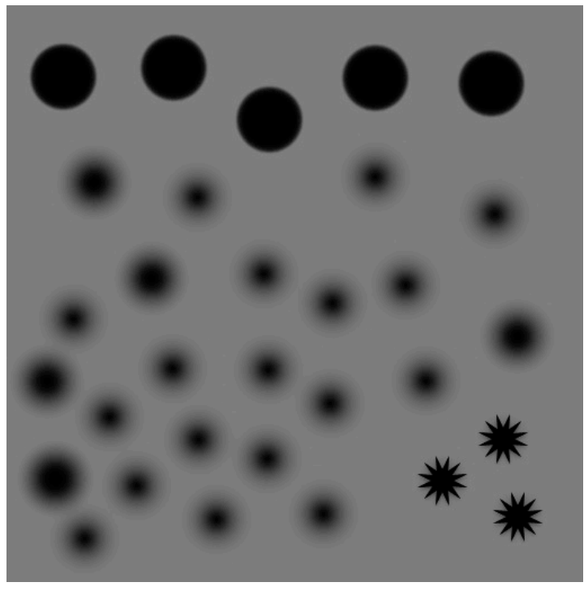
5. Why is ray marching faster than ray casting?

6. How can you perform picking in image space?

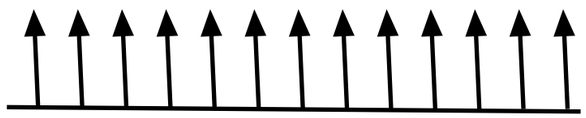
Bump mapping

Simulates surface structure by manipulating the normal vector





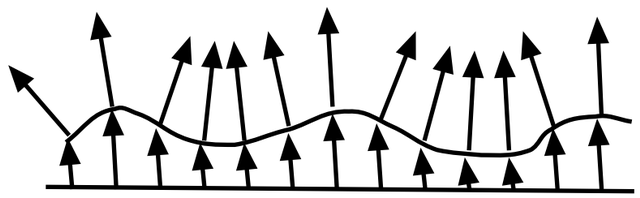
Bump mapping - model



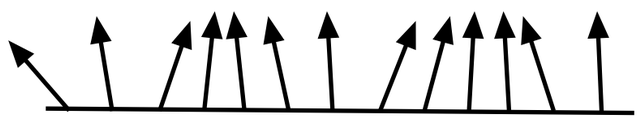
Surface with normal vectors

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 **n**
Texture coordinates s(**p**), t(**p**)
Directions of texture coordinates **s**, **t**
The bump function b(s,t)

Calculate the partial derivative of the bump function, b_s and b_t
$\mathbf{n}' = \mathbf{n} + b_t * (\mathbf{s} \times \mathbf{n}) + b_s * (\mathbf{t} \times \mathbf{n})$

or, if **s, t, n** are orthogonal

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

Texture coordinate system

n = normal vector
s = tangent
t = bitangent

s and **t** can be calculated from texture coordinate variations (e.g. Lengyel's method)

Light calculations

needs (**n, s, t**) to create **n**'
needs light source and surface positions

Must transform to the same coordinate system, e.g. view coordinates

Transforming directional vectors

Important!

n, **s** and **t** are all *directional* vectors

Must be transformed by modified model-to-world/
world-to-view matrices
without translations

More about this in a moment...

Calc of modified normal vector

$b_s = db/ds$
$b_t = db/dt$

$\mathbf{n}' = \mathbf{n} + b_s \cdot \mathbf{s} + b_t \cdot \mathbf{t}$ ("in")

or

$\mathbf{n}' = \mathbf{n} - b_s \cdot \mathbf{s} - b_t \cdot \mathbf{t}$ ("out")

Gradients are simply one step differences in s and t!

$-b_s = b[s, t] - b[s+1, t]$
$-b_t = b[s, t] - b[s, t+1]$

1

Normalize!

Variant/optimization of bump mapping: Normal mapping

Precalculate b_s och b_t, save as picture!

But it is just a simple difference! Why can this be significant?

Storage in texture

”Scale and bias”:

R = (ds+1)/2
G = (dt+1)/2

(Why?)

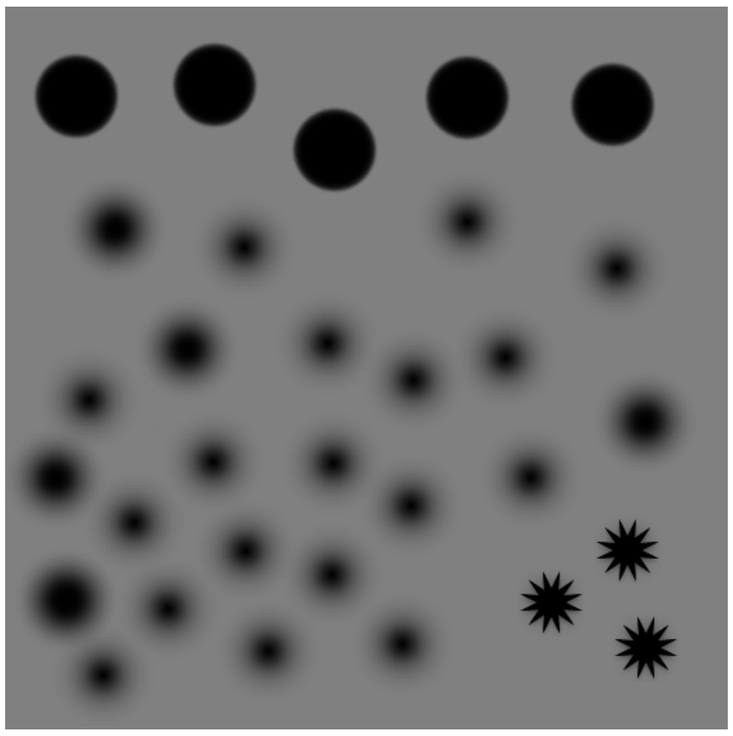
Fetch from texture:

ds = 2R - 1
dt = 2G - 1

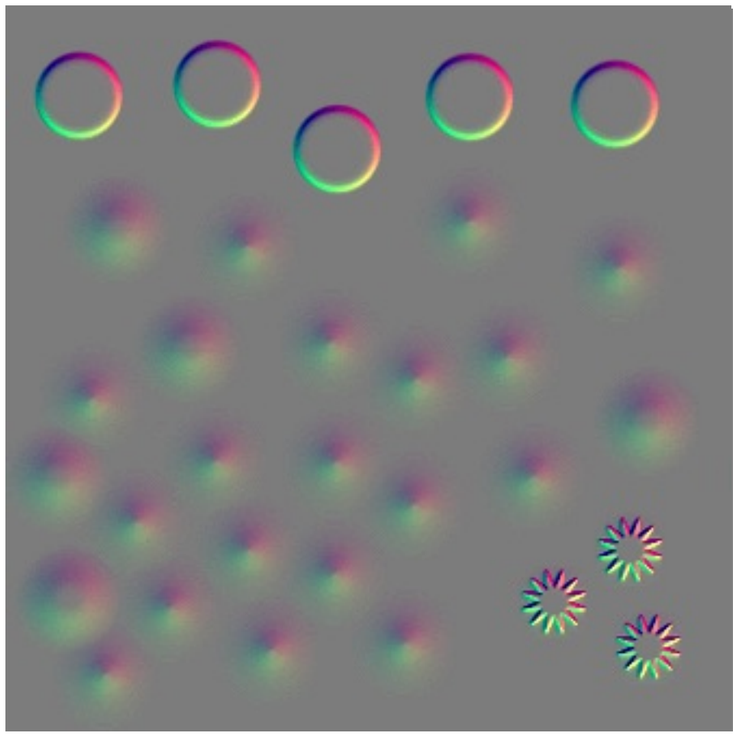
$\mathbf{n}_t = (x, y, z)$

Example of normal map

Bump map in my example



Bump map



Normal map

Bump mapping or normal mapping?

Normal mapping optimizes memory access

Bump mapping gives more information and is easier to edit

We might want both the bump map and the normal map!

Splatting

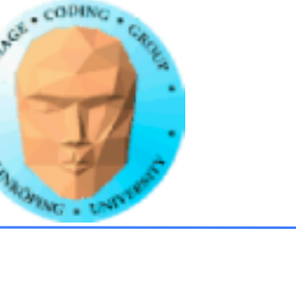
Using a texture as "map" for multitexturing.

Each channel R, G, B (and possibly A) used for one texture.

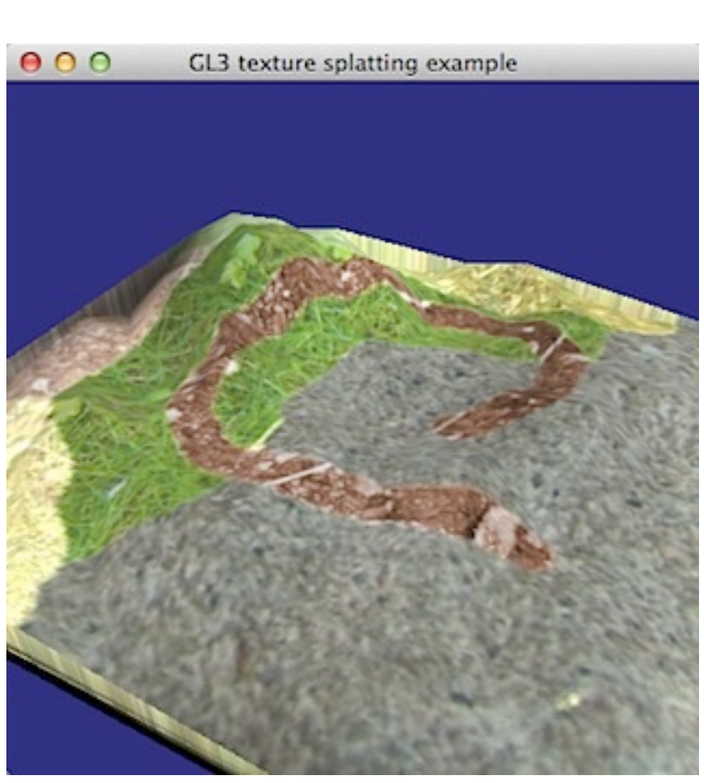
Smooth transitions between textures.

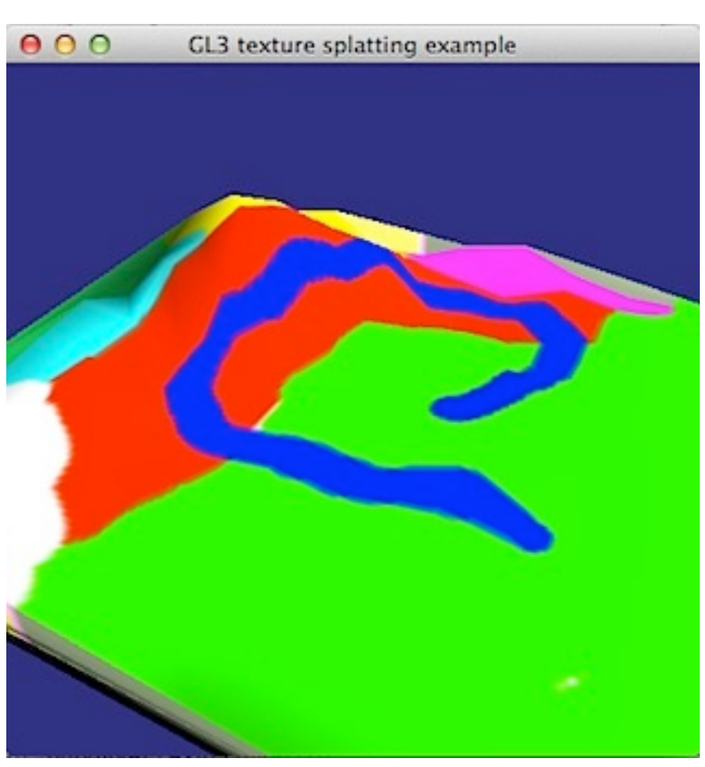
Combinations of textures possible.

Input



Result


GL3 texture splatting example


GL3 texture splatting example

Number of textures

RGB: Three textures. Easiest!

RGB plus 1-magnitude: Four textures

RGBA: Four textures

RGBA plus 1-magnitude: Five textures

Need more? Use multiple splatmaps!

Beware texture scaling

Splatmaps are often large scale, rarely repeated

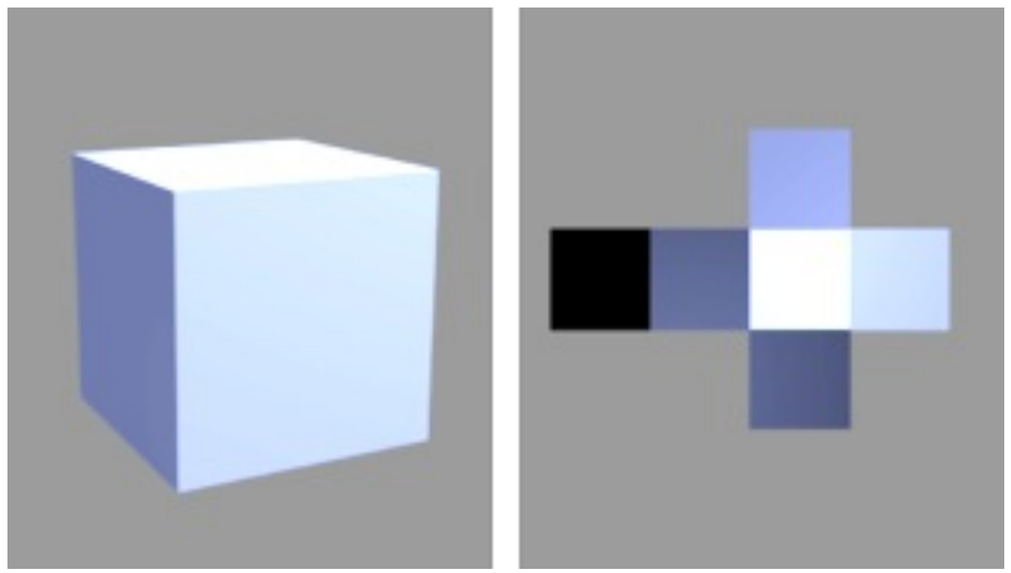
The textures it controls is often repeated

Light mapping

Applying pre-calculated lighting to a model

Saves real-time processing time for models with static lighting

Allows high-quality lighting with high performance



(Image from Wikipedia)

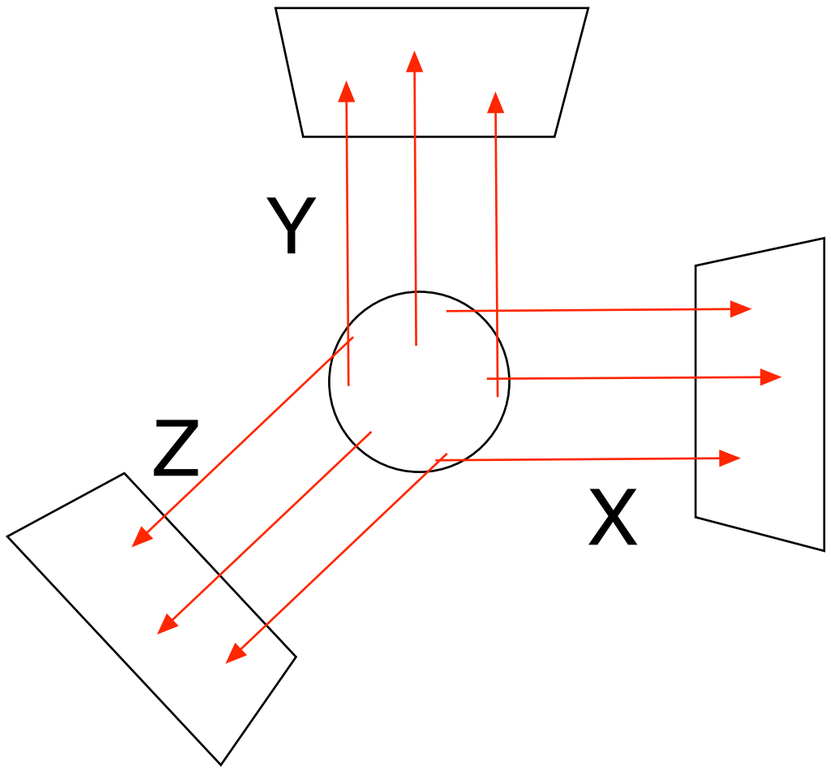
Triplanar texure mapping

Combine planar mapping with multitexturing

Apply textures - possibly the same - along all three axes!

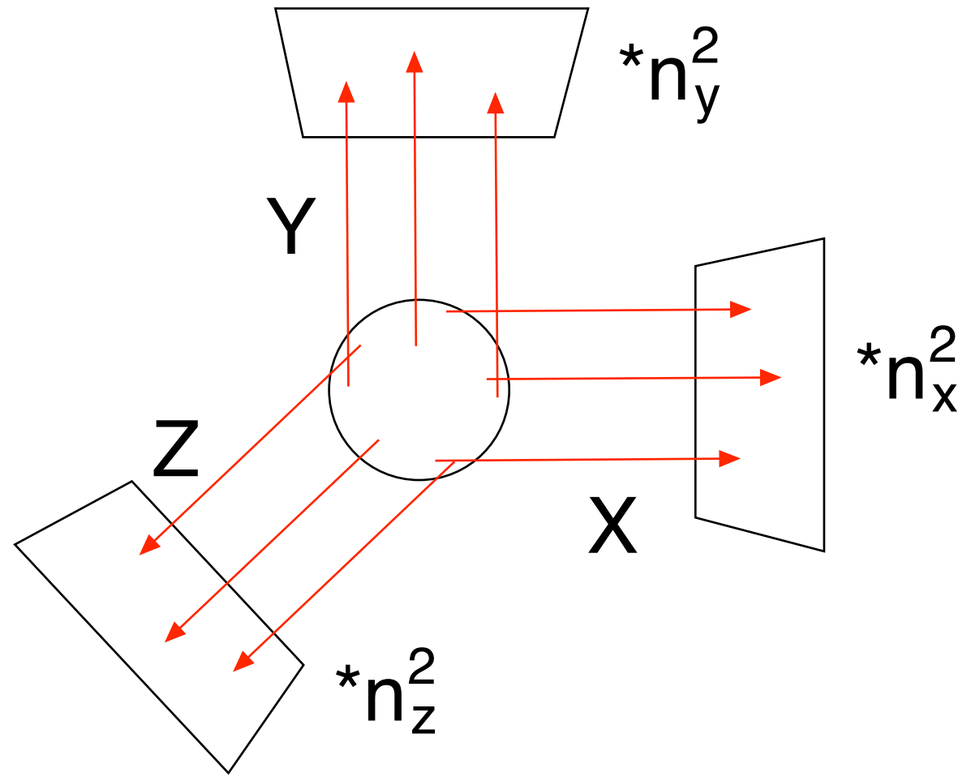
Sum together with suitable function

Map texture on object along all three axes!

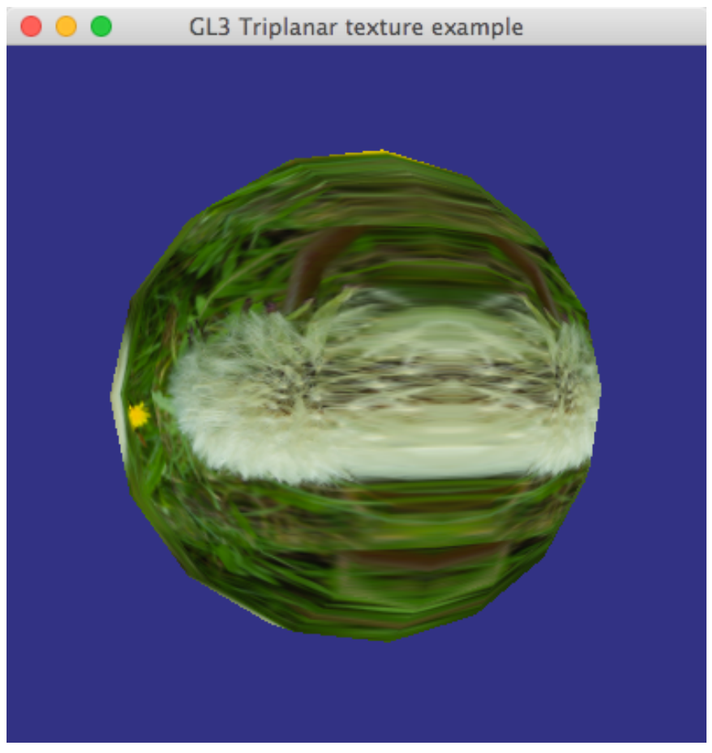


Blend together based on normal vector!

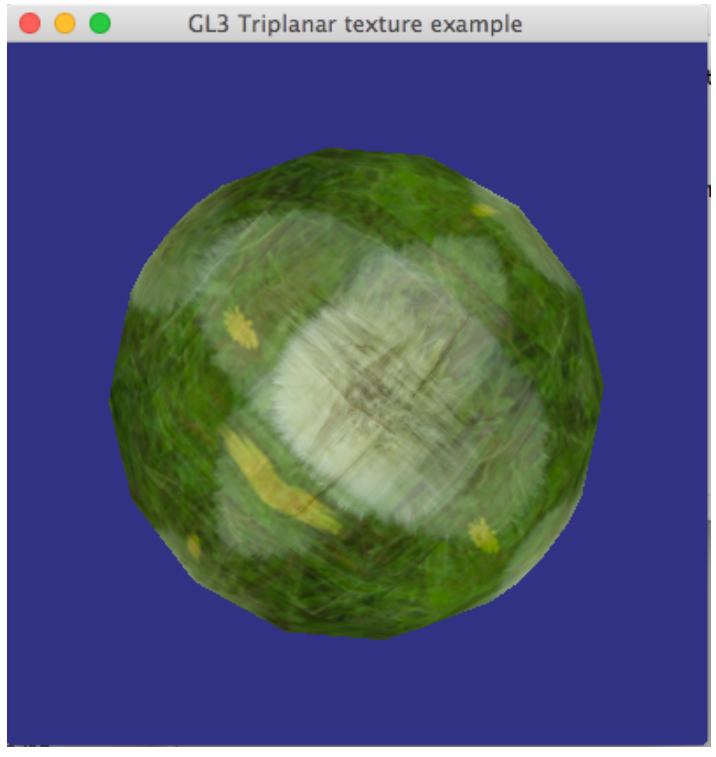
Example: Square of each component



Blend together properly



Simple planar



Linear by component - not same sum everywhere!

Square of component

Triplanar texture mapping

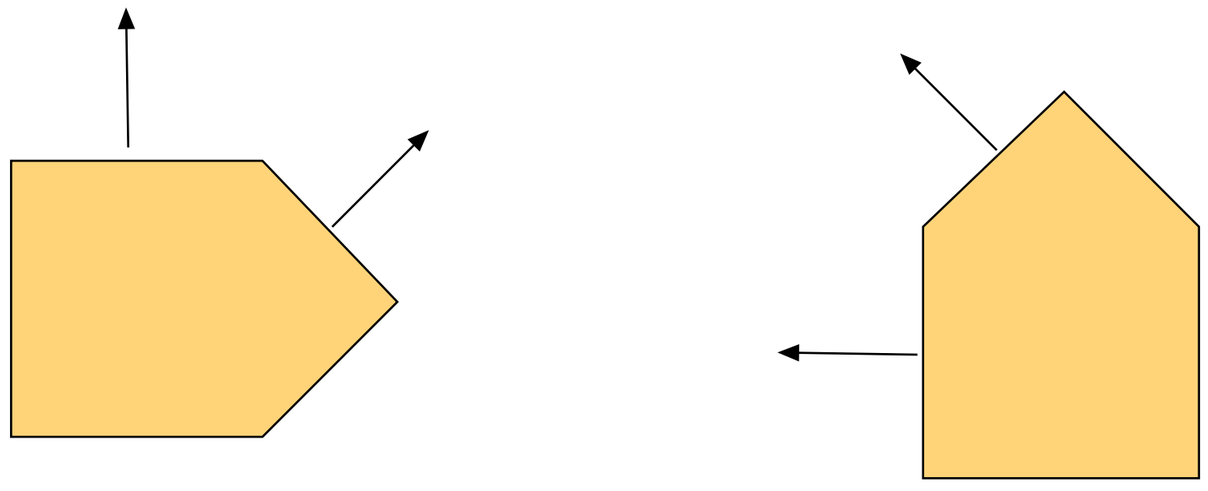
Great alternative for procedurally generated geometry

also a general purpose mapping for general materials

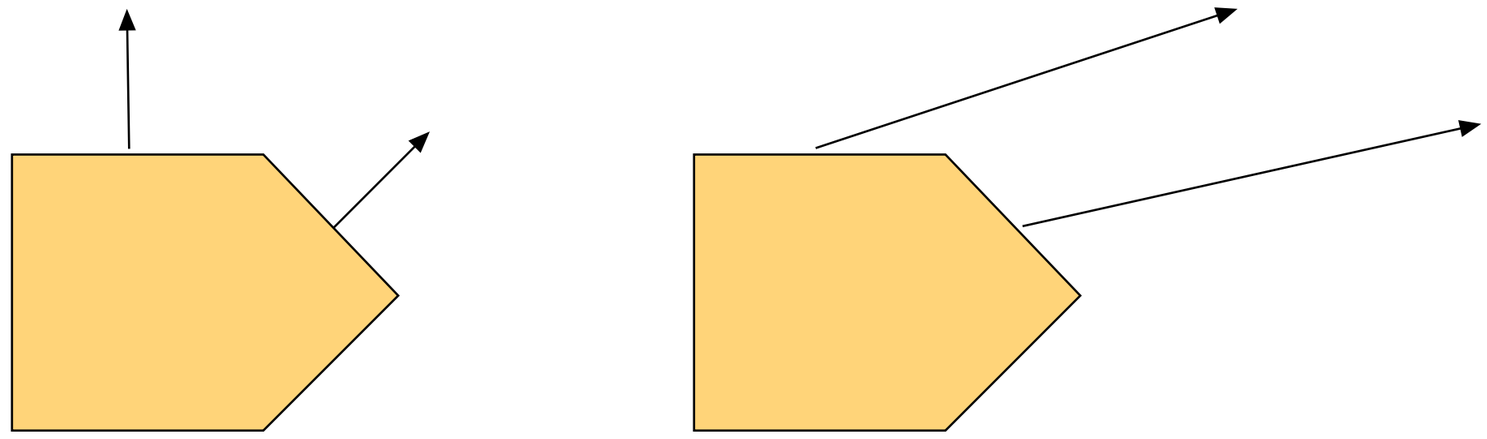
Now, lets return to that normal matrix...

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?

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

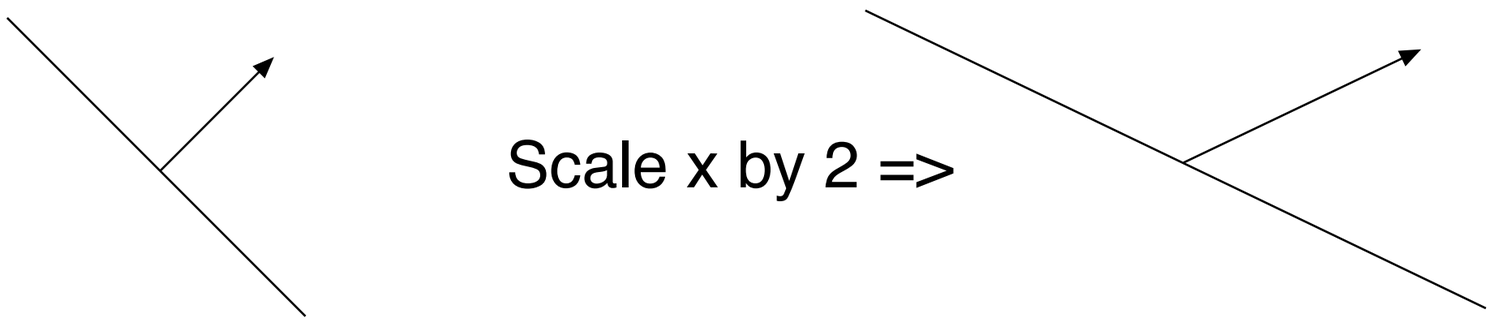
or we zero the translation part:

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

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

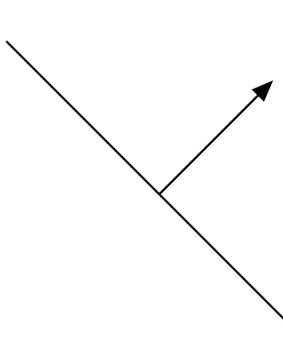
But wait!

For non-uniform scaling, this does not work!

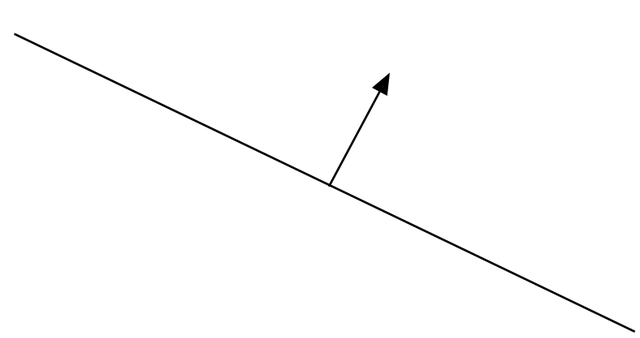


The normal vector is no longer perpendicular to surface!

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 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 rotation

=> Use inverse transpose of rotation part

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

Please don't miss this!

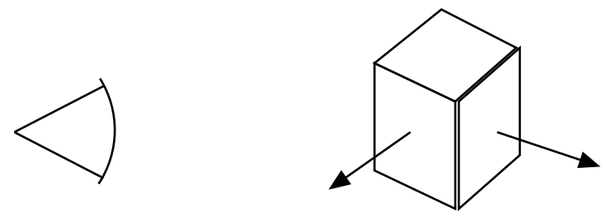
More visible surface detection

sometimes applicable to entirely different problems!

Backface culling

Object space method

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

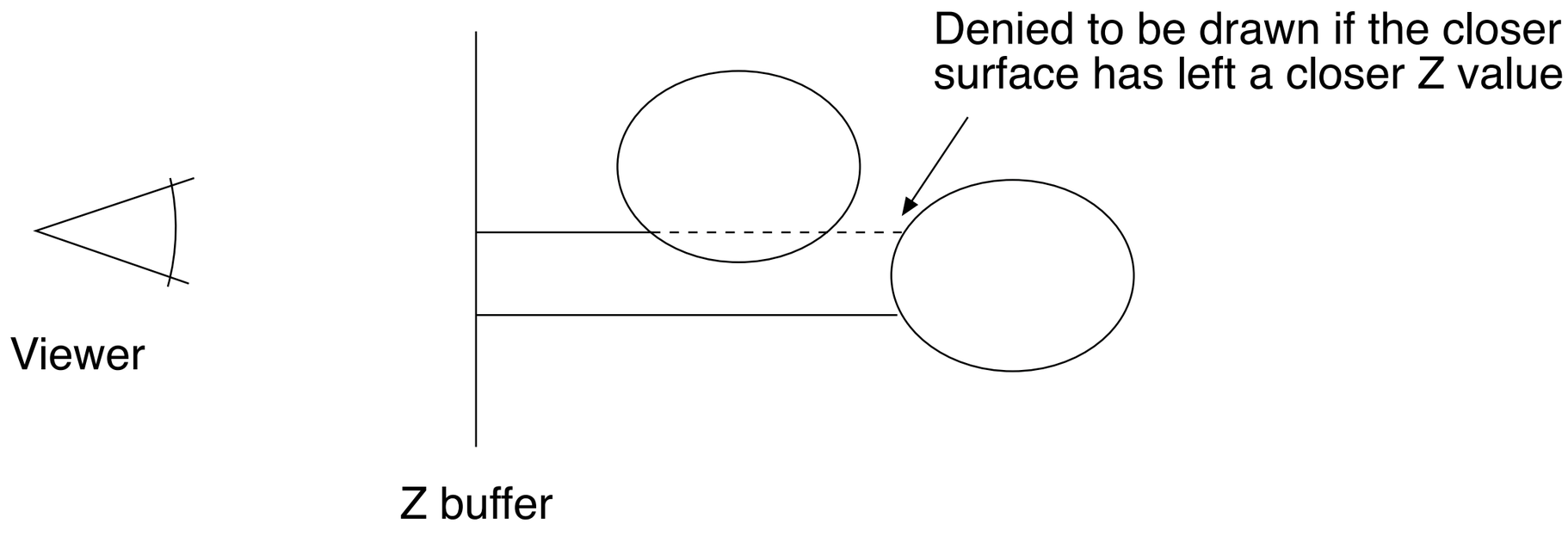


Removes ≈50% of all polygons that would otherwise be in view!

Z-buffer

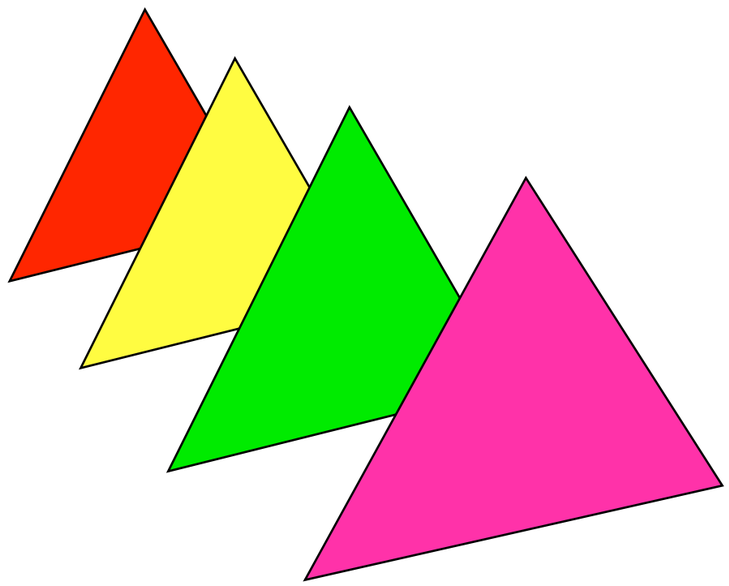
Depth-buffer method

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



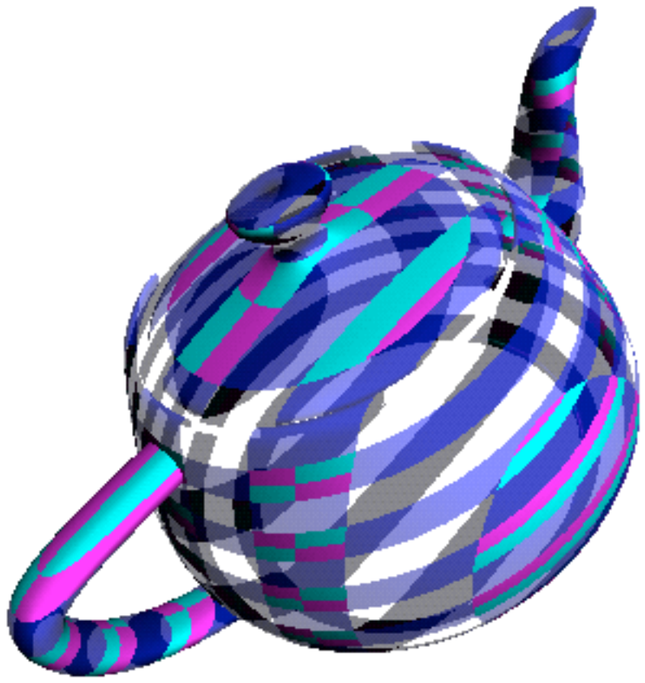
Render from back to front

Both image and object space method

Drawing with transparency

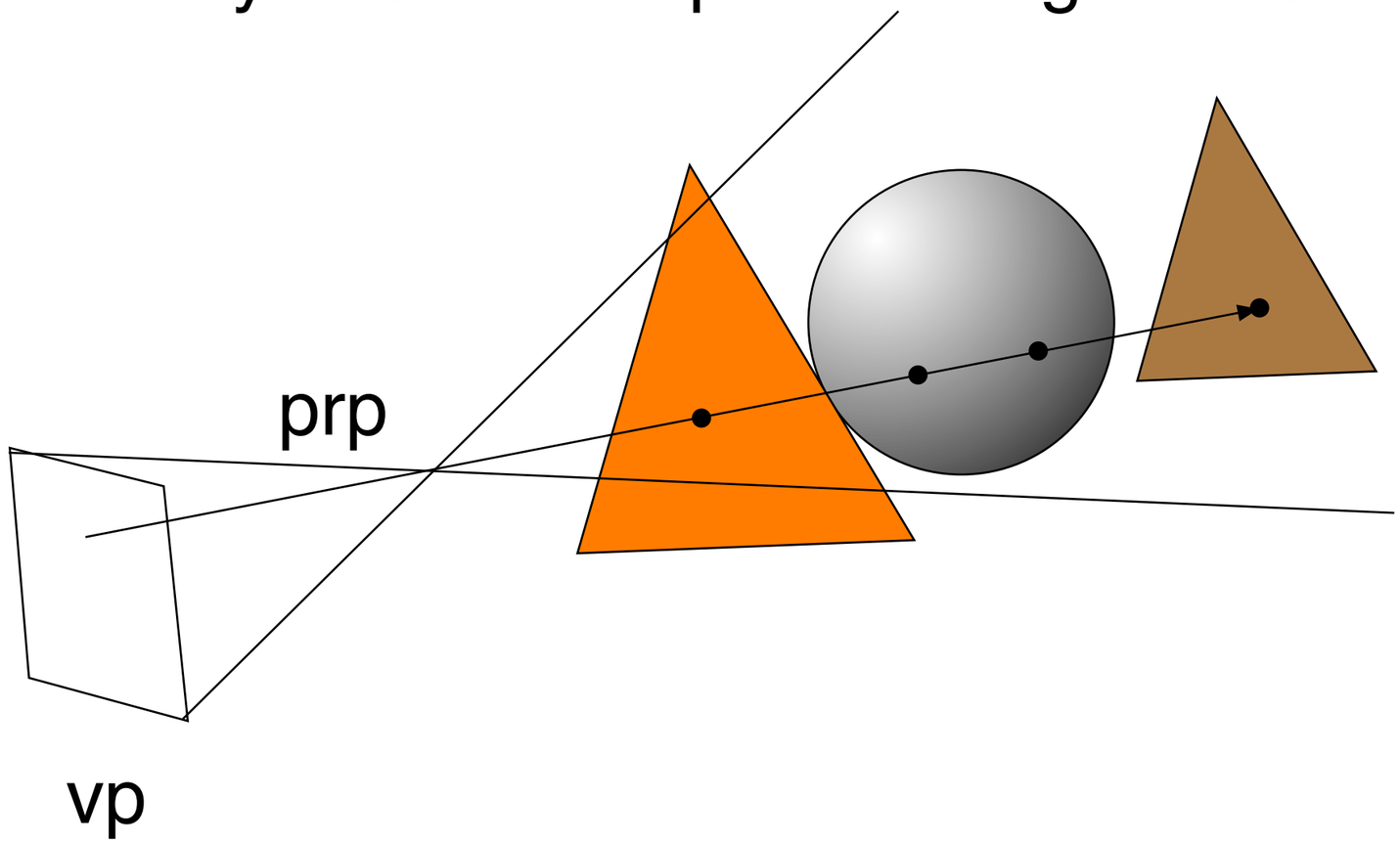
The relevant case for Painter's algorihm.

Sort on object level.



Ray-casting

Follow rays from each pixel through the scene



Full 3D raycasting

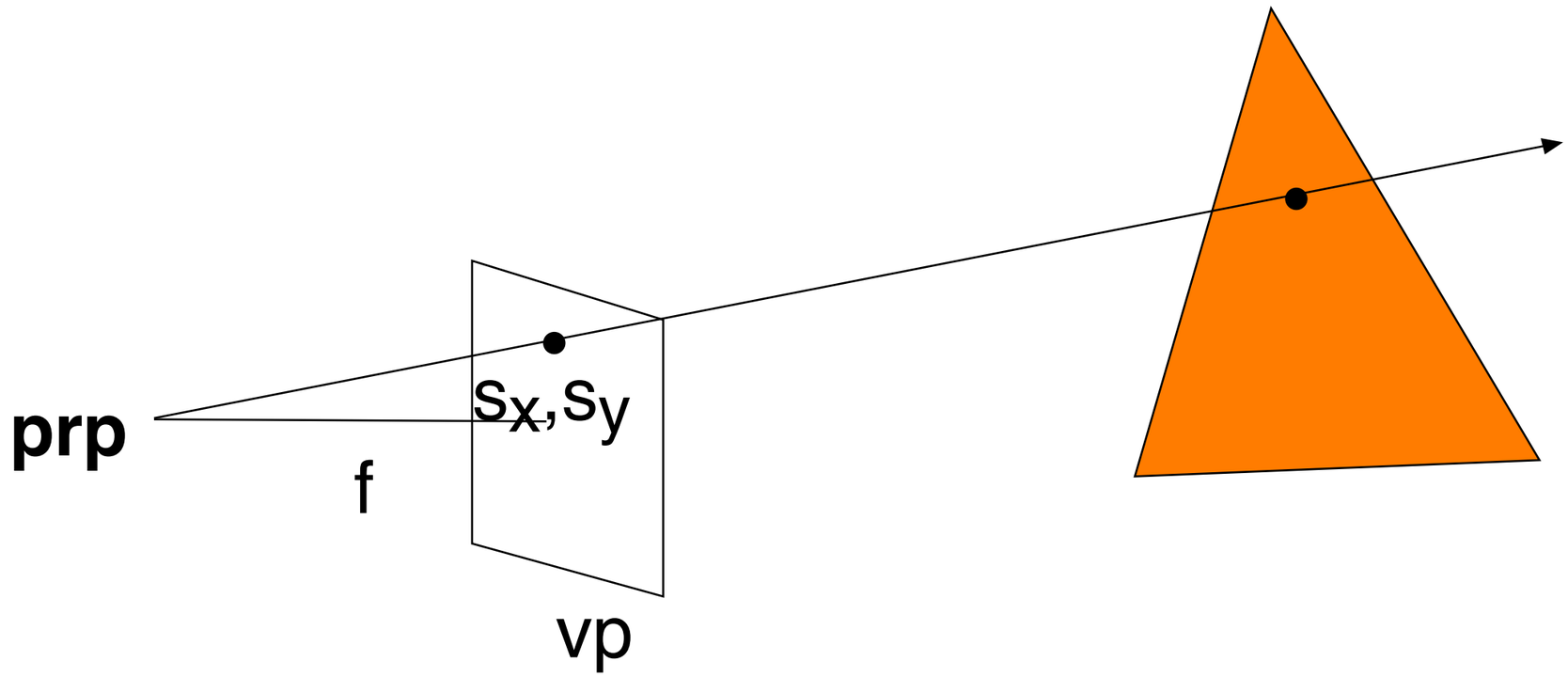
for every pixel (x,y) in the image

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

calculate intersections with all objects in the scene

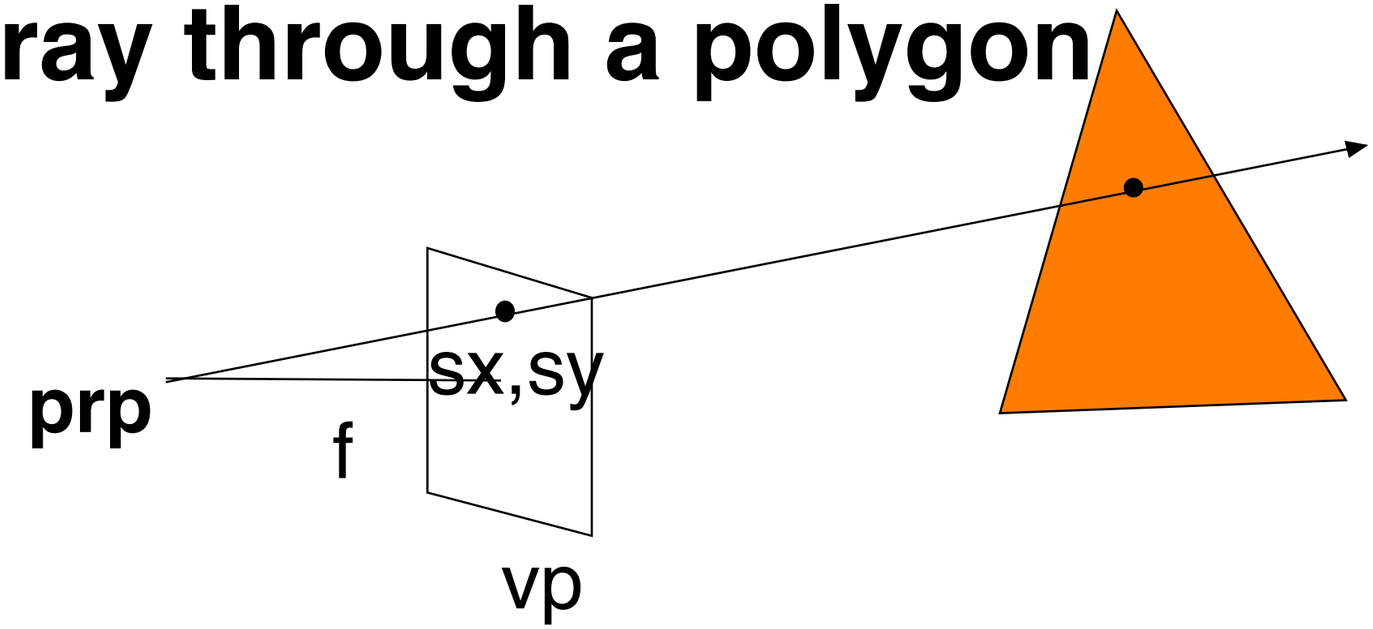
the pixel value is calculated from the closest intersection found

The ray



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

A ray through a polygon



Take the case prp = (0,0,0)

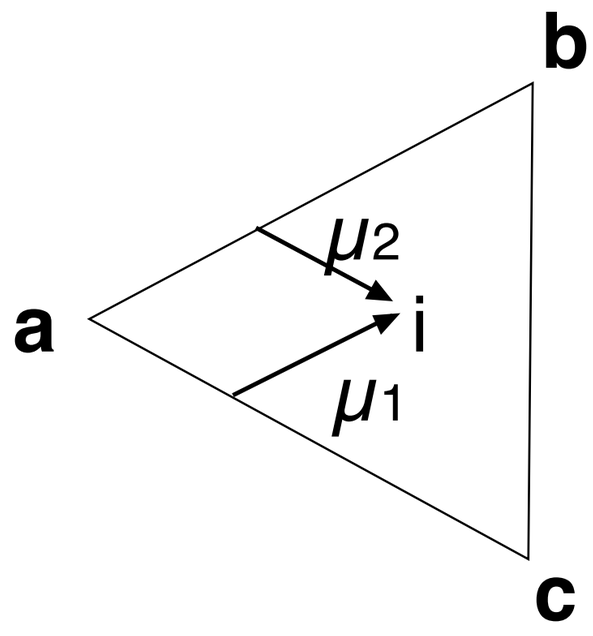
Insert (0,0,0) + $\mu(s_x,s_y,-f)$ in plane equation:

$$A\mu s_x + B\mu s_y - C\mu f + D = 0$$

$$\mu = -D / (As_x + Bs_y - Cf)$$

Is the point in the polygon?

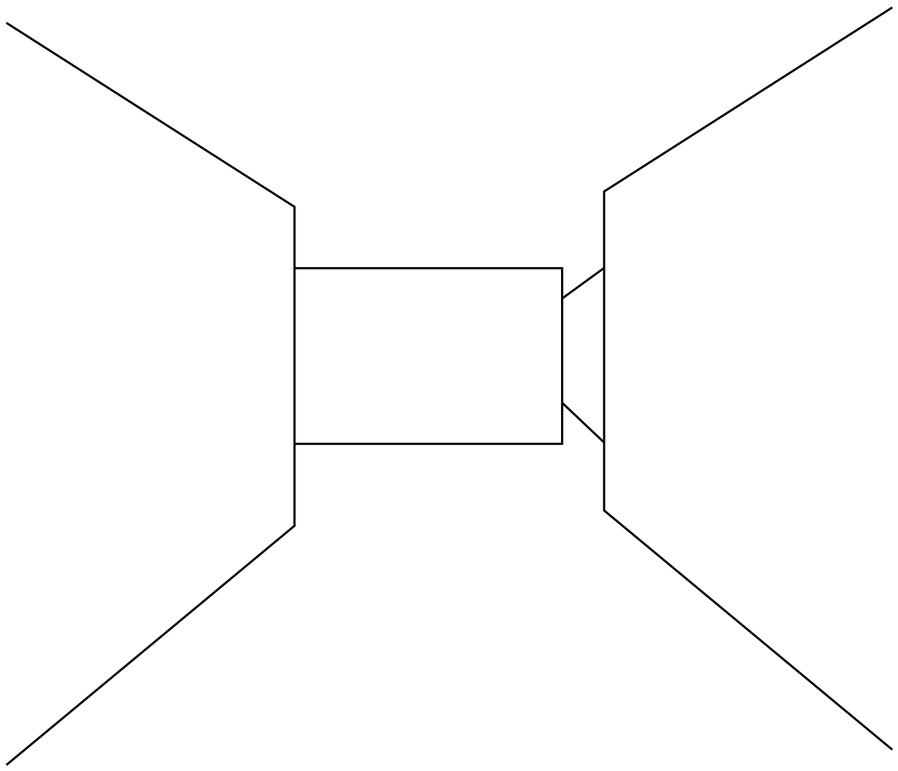
Triangle: Straight-forward
Several methods possible.

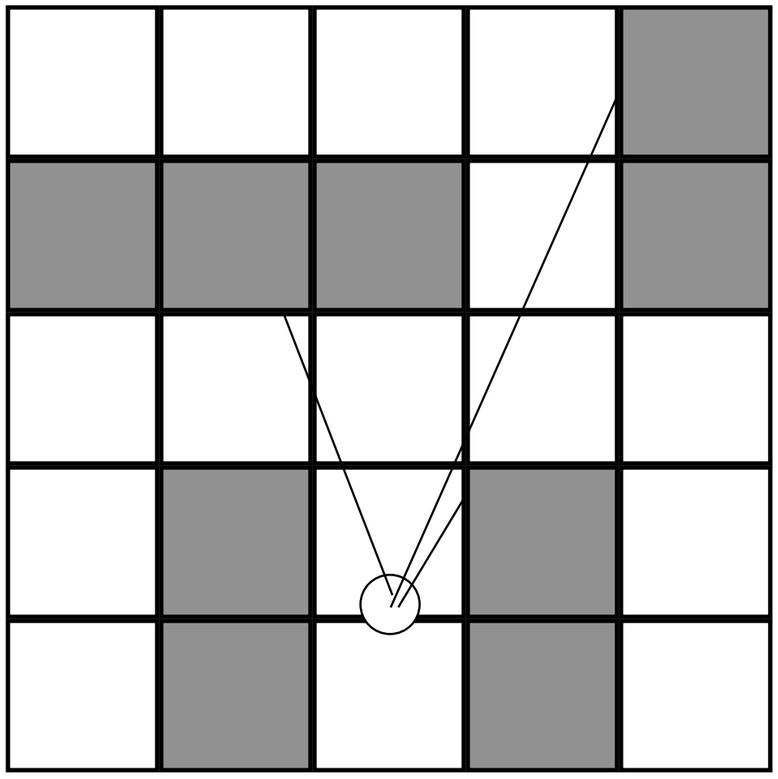


$$\mathbf{i} = \mathbf{a} + \mu_1 * \mathbf{ab} + \mu_2 * \mathbf{ac}$$

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

Raycasting in a grid: Ray marching



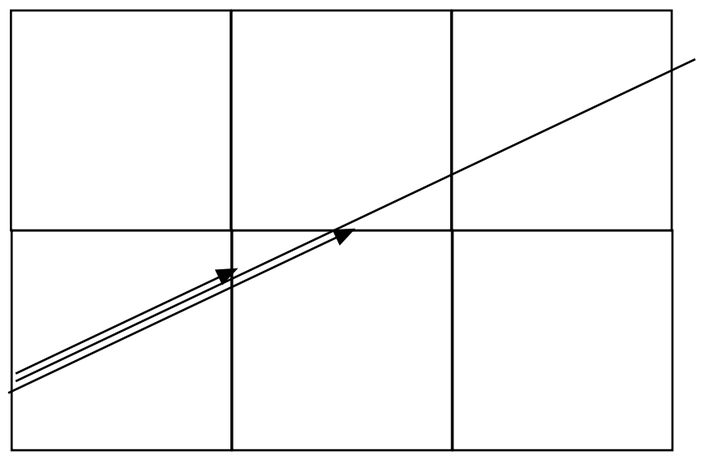


Ray marching relatively easy

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

Pick the closest, check neighbor space

Repeat until filled space is found.

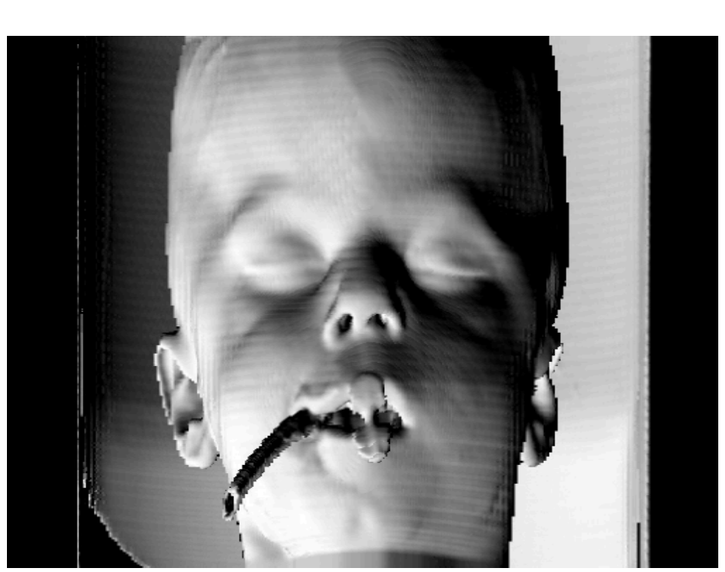


Essentially a line drawing algorithm!

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 lecture

Rotation around arbitrary axis

Large worlds: Frustum culling and other high-level method