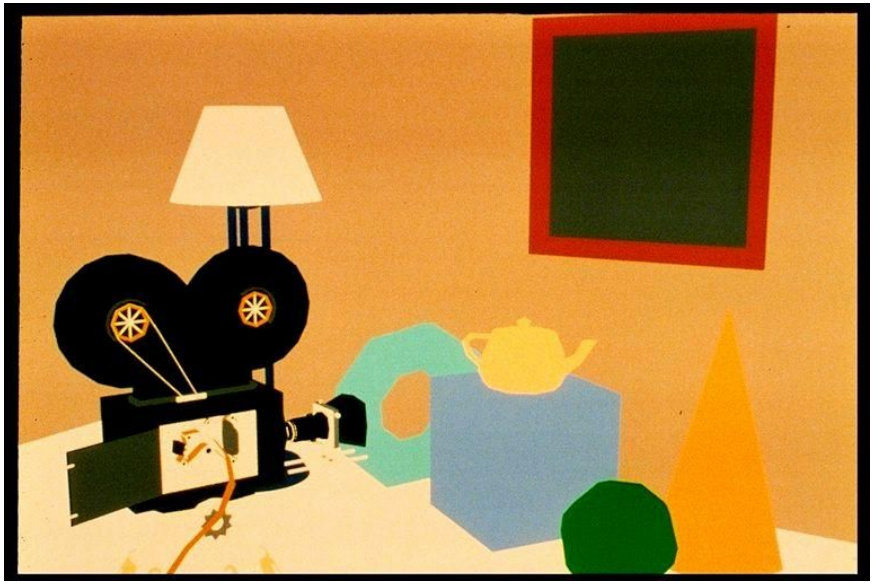
# Lecture 7:

# Texturing | Part 1

Contents

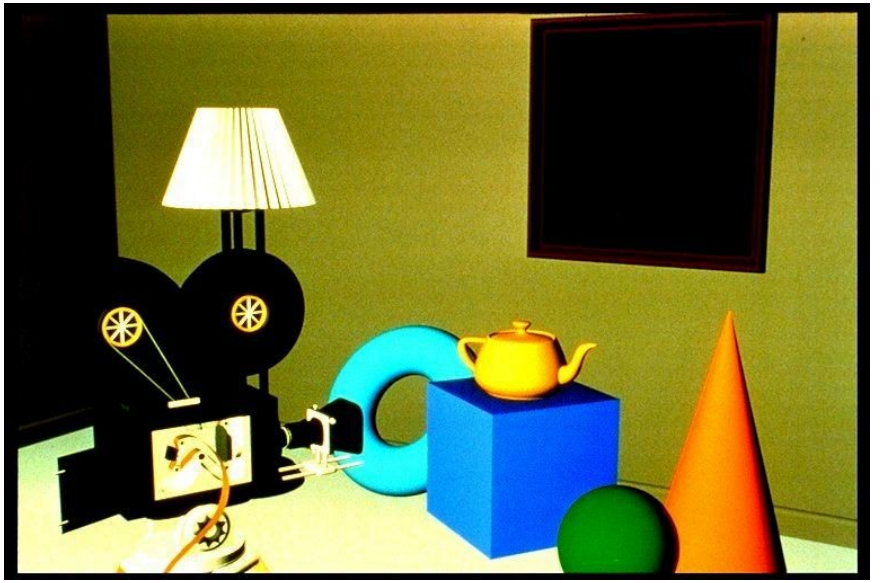
No illumination

Constant colors



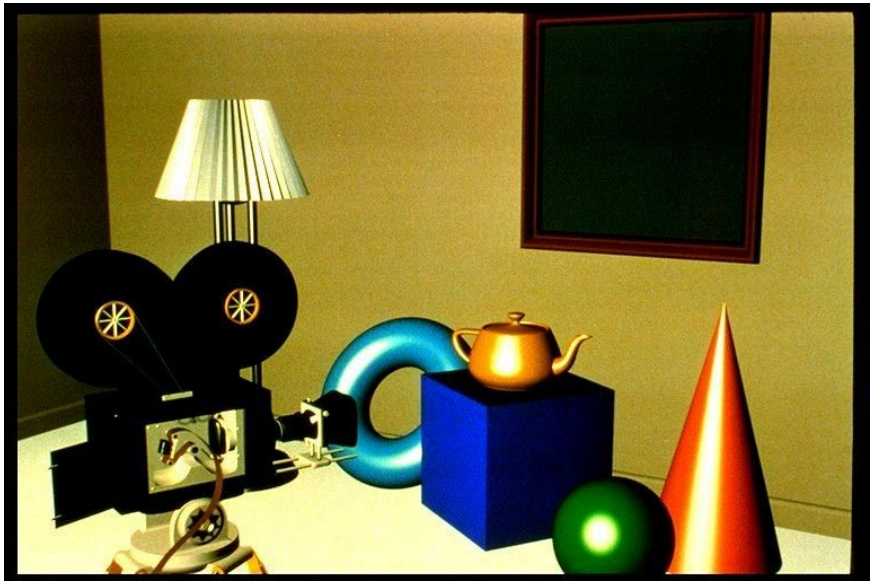
Parallel light

Diffuse reflection



Parallel light

Specular reflection



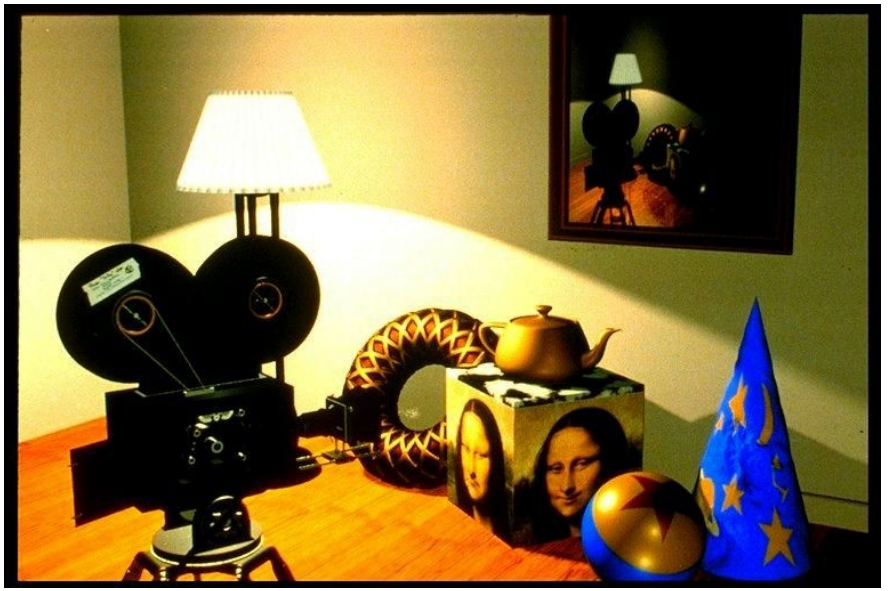
Multiple local light sources

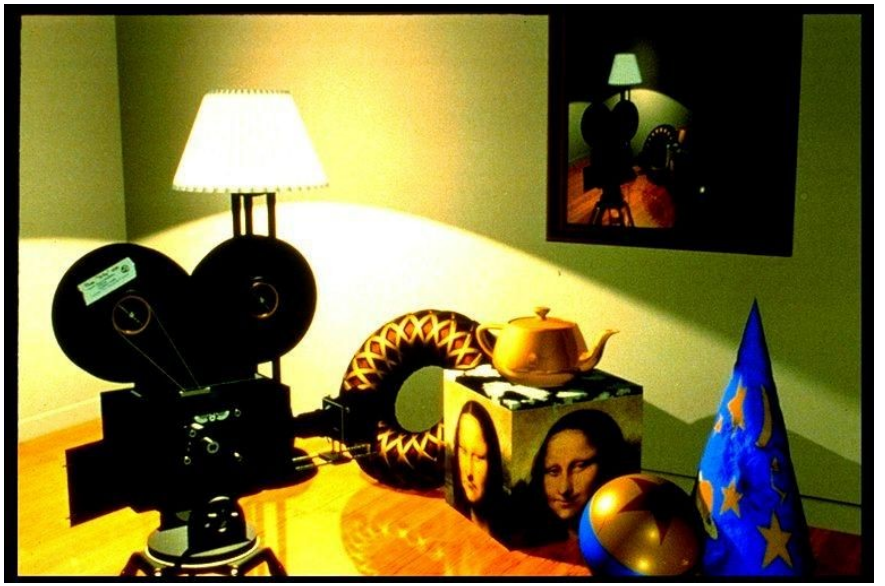
Different BRDFs

Object properties constant over surface

# Locally varying object characteristics

- 2D Image Textures
- Shadows
- Bump-Mapping
- Reflection textures





# Modulation of object surface properties

## Reflectance

- Color (RGB), diffuse reflection coefficient $k_d$
- Specular reflection coefficient $k_s$

## Opacity ($\alpha$)

## Normal vector

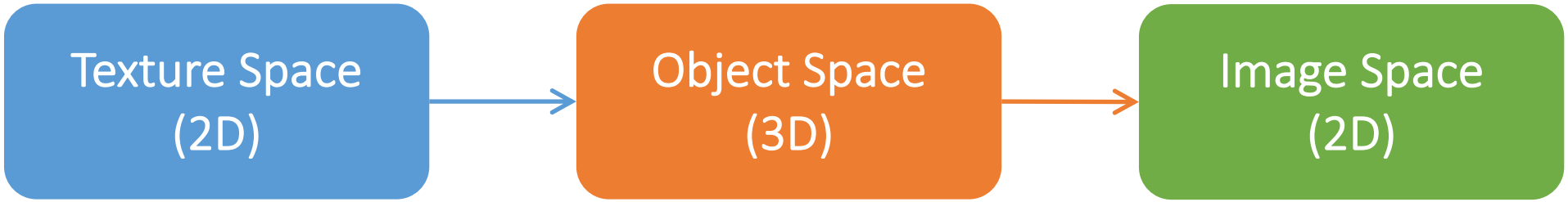
- $N(P) = N(P + tN)$ or $N = N + dN$
- „Bump mapping“ or „Normal mapping“

## Geometry

- $P = P + dP$
- „Displacement mapping“

## Distant illumination

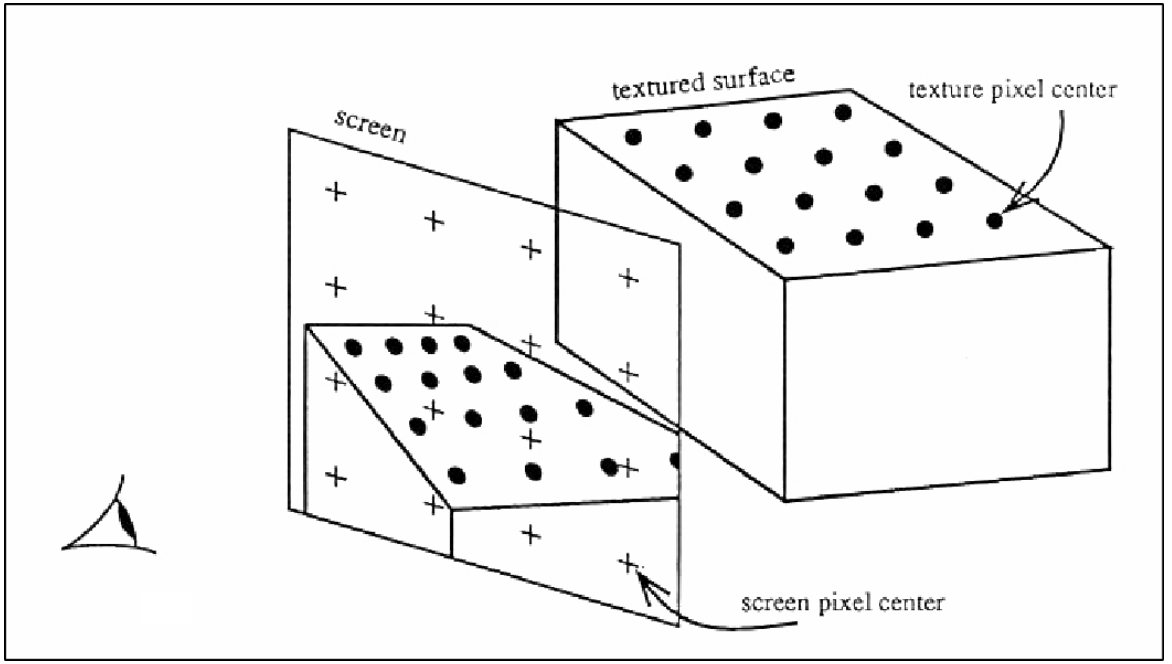
- “Environment mapping“, “Reflection mapping“

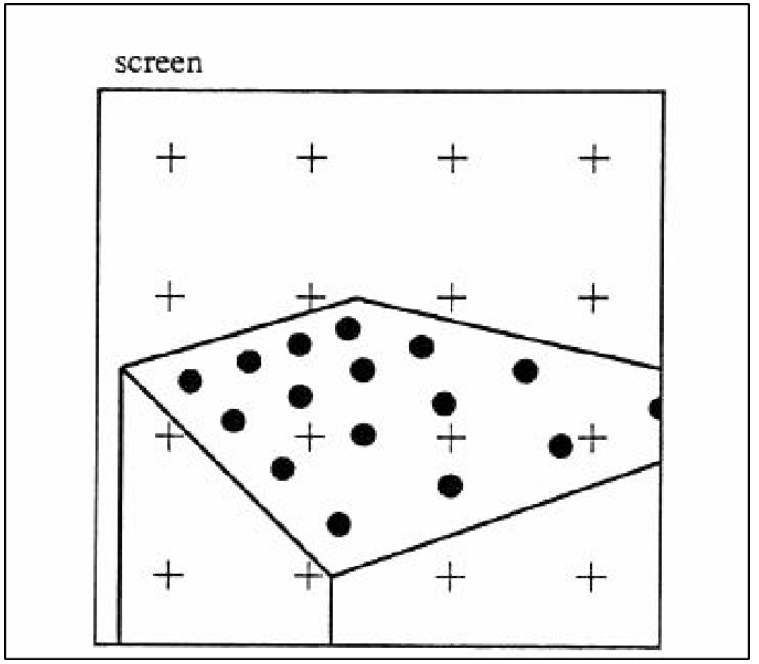



The texture is mapped onto a surface in 3-D object space, which is then mapped to the screen by the viewing projection. These two mappings are composed to find the overall 2D texture space to 2D image space mapping, and the intermediate 3D space is often forgotten. This simplification suggests texture mapping's close ties with image warping and geometric distortion.

- Texture space $(u, v)$
- Object space $(x_0, y_0, z_0)$
- Screen space $(x, y)$

# 2D Texturing



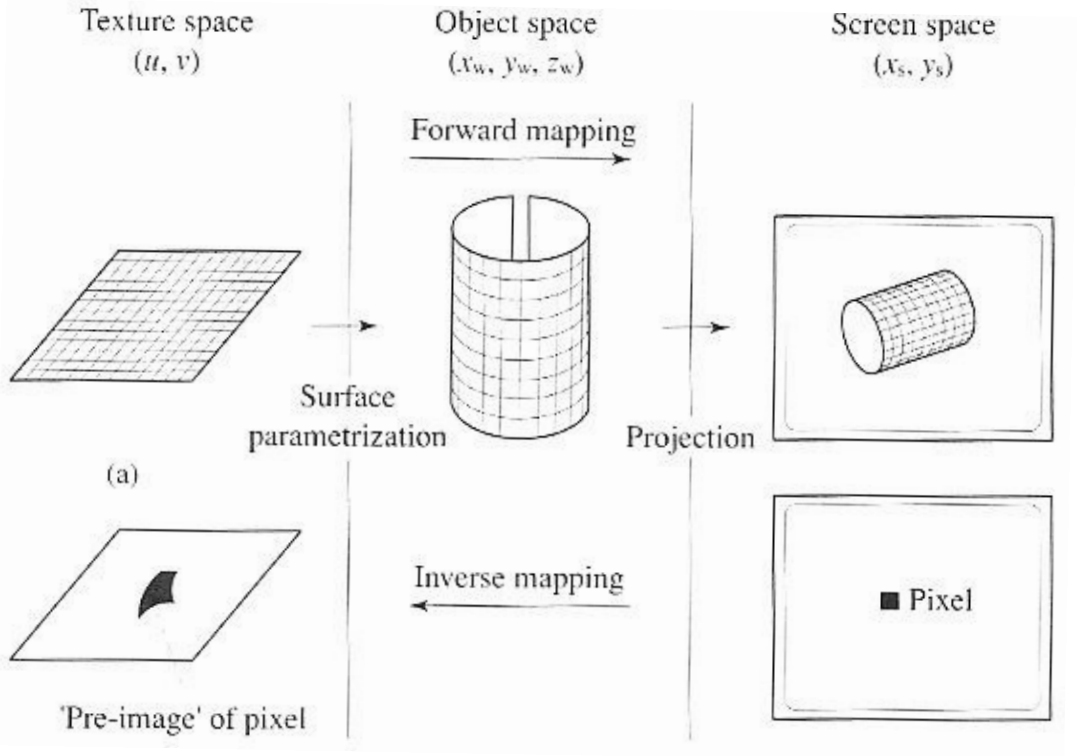

- 2D texture mapped onto object
- Object projected onto 2D screen
- 2D to 2D: warping operation
- Uniform sampling?
- Hole – filling / blending?

## Forward mapping

- Object surface parameterization
- Projective transformation

## Inverse mapping

- Find corresponding pre-image/footprint of each pixel in texture
- Integrate over pre-image



## Maps each texel to its position in the image

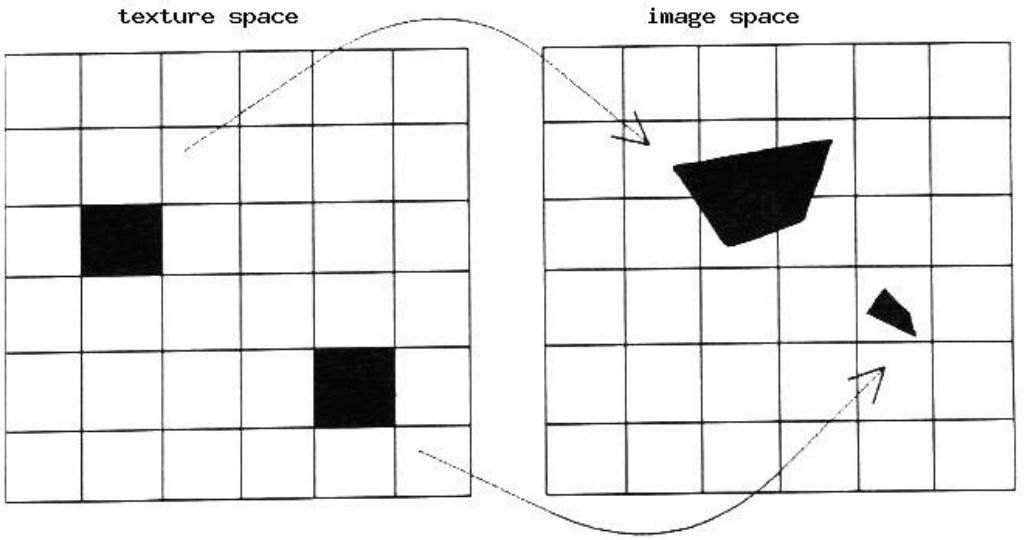
## Uniform sampling of texture space does not guarantee

## uniform sampling in screen spacespace

## Possibly used if

- The texture-to-screen mapping is difficult to invert
- The texture image does not fit into memory

```
Texture scanning:
for v
  for u
    compute x(u,v) and y(u,v)
    copy TEX[u,v] to SCR[x,y]
(or in general
    rasterize image of TEX[u,v])
```



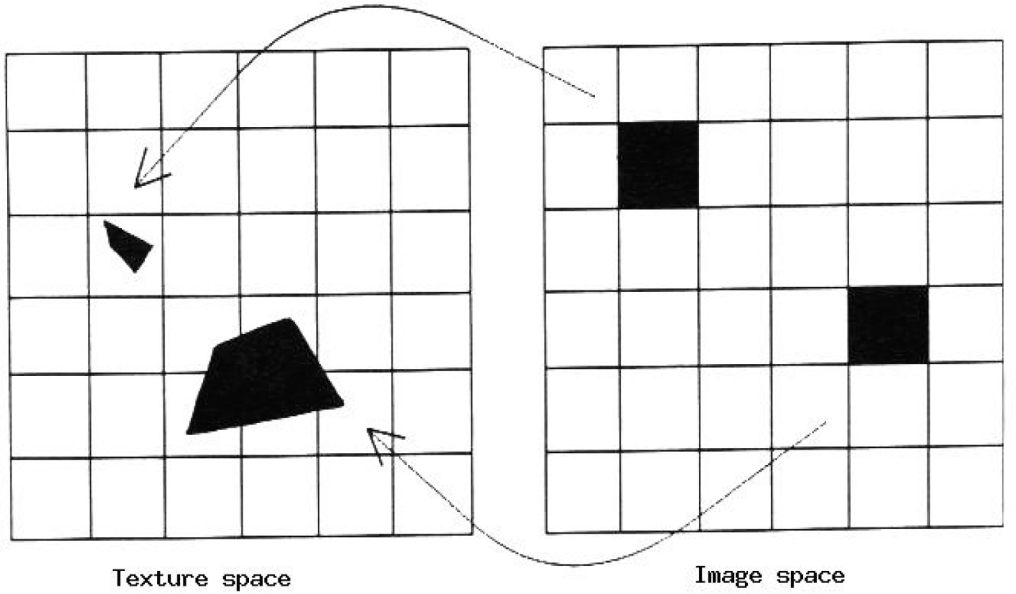
## Requires inverting the mapping transformation

## Preferable when the mapping is readily invertible and the texture image fits into memory

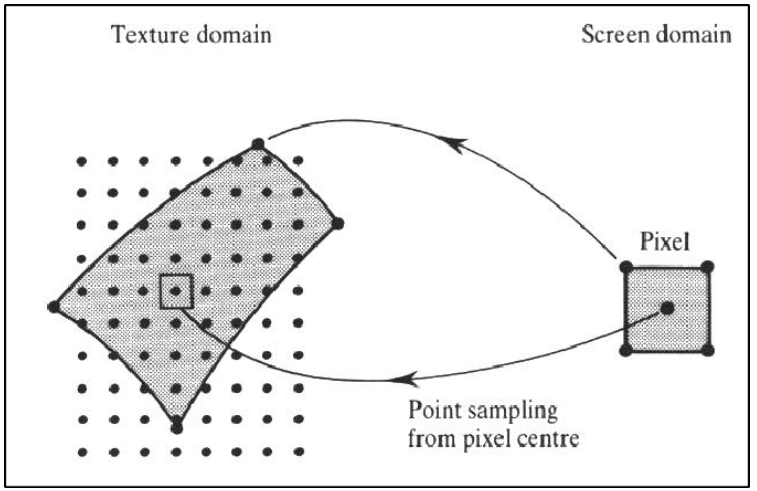
## The most common mapping method

- For each pixel in screen space, the pre-image of the pixel in texture space is found and its area is integrated over

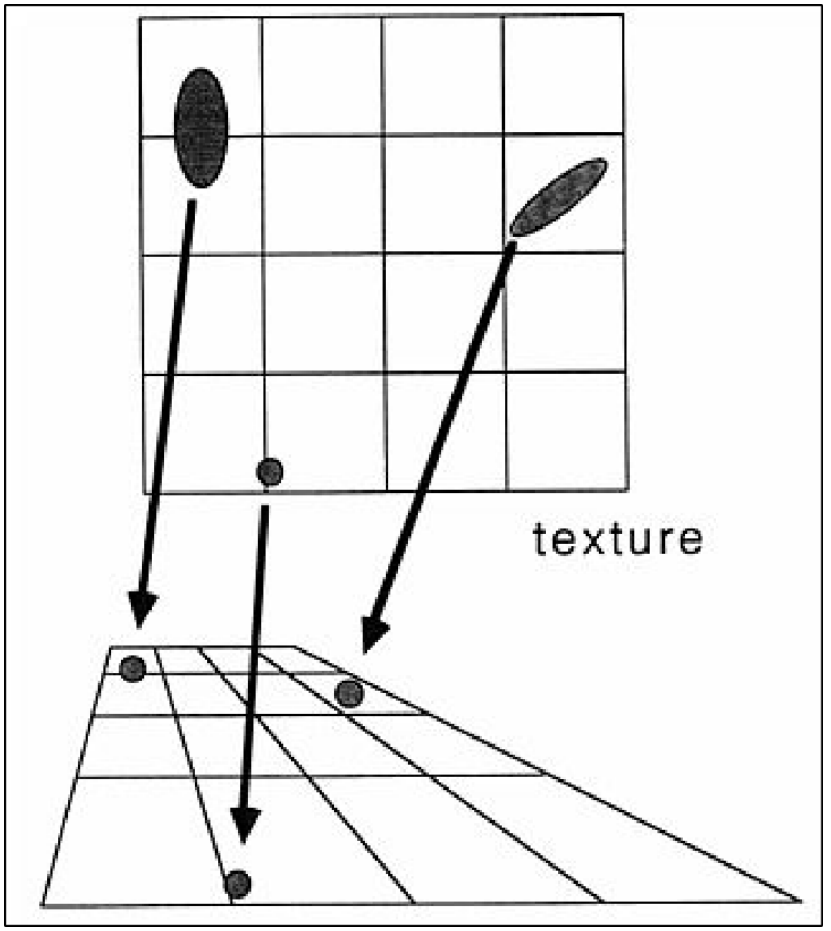
```
Texture scanning:
for y
  for x
    compute u(x,y) and v(x,y)
    copy TEX[u,v] to SCR[x,y]
(or in general
    Integrate over image of
    SCR[x,y])
```



A square screen pixel that intersects a curved surface has a curvilinear quadrilateral pre-image in texture space. Most methods approximate the true mapping by a quadrilateral or parallelogram. Or they take multiple samples within a pixel. If pixels are regarded as circles, their pre-images are ellipses




**Approximating a quadrilateral texture area with (a) a square, (b) a rectangle, and (c) an ellipse. Too small an area causes aliasing; too large an area causes blurring.**

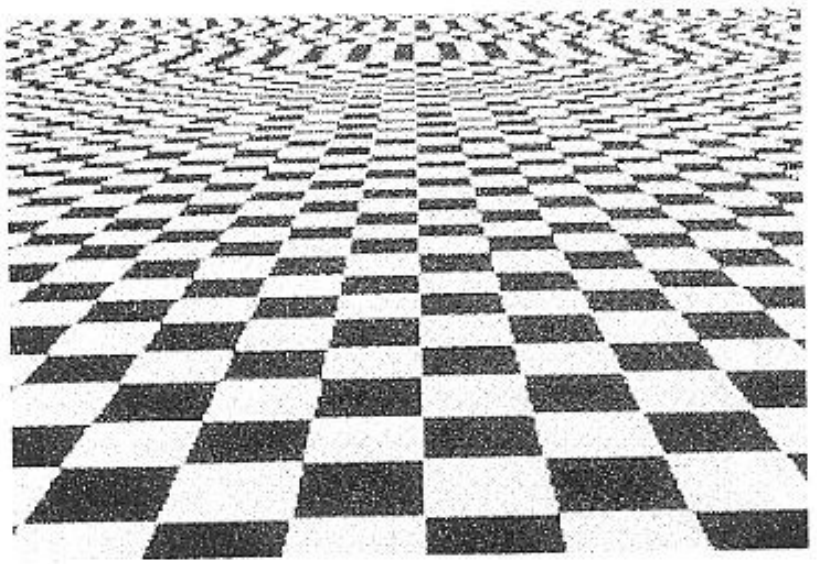


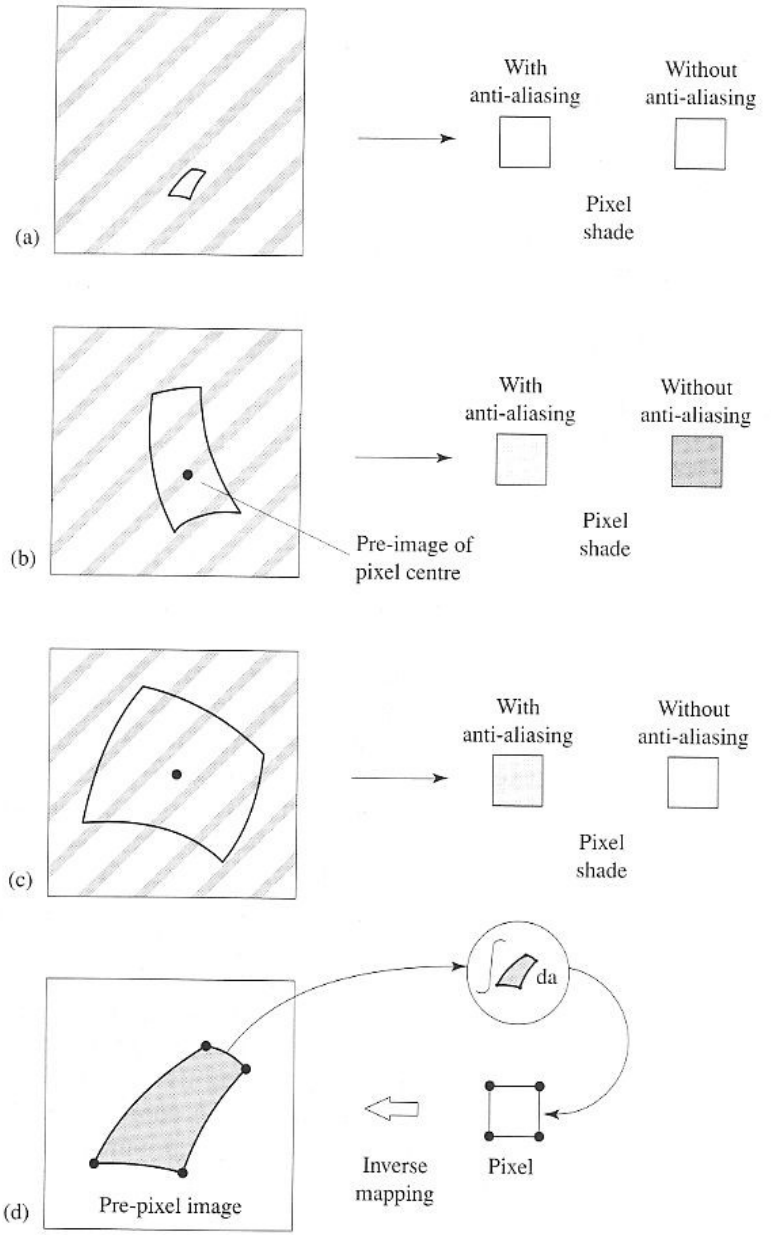
## Integration of Pre-image

- Integration over pixel footprint  in texture space

## Aliasing

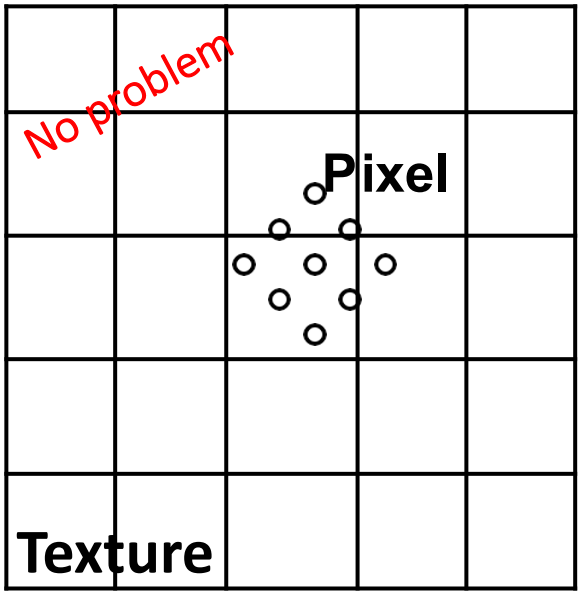
- Texture insufficiently sampled
- Incorrect pixel values
- “Randomly” changing pixels  when moving





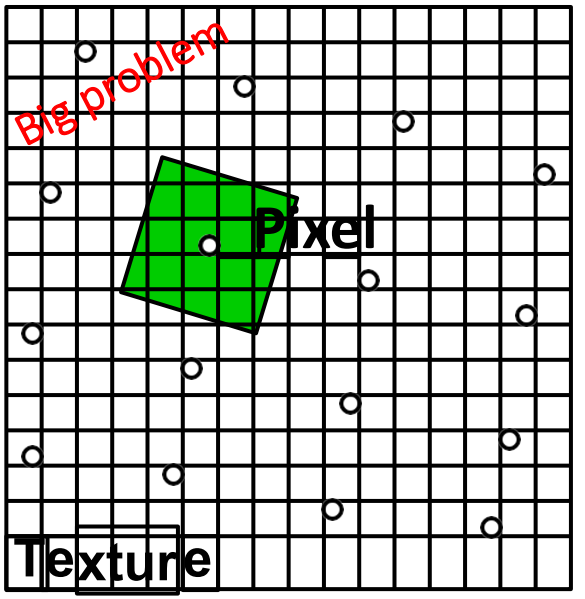
## Magnification

- Map few texels onto many pixels
- Nearest:
  - Take the nearest texel
- Bilinear interpolation:
  - Interpolation between 4 nearest texels
  - Need fractional accuracy of coordinates




## Minification

- Map many texels to one pixel
  - Aliasing:
    - Reconstructing high-frequency signals with low level frequency sampling
- Filtering
  - Averaging over (many) associated texels
  - Computationally expensive



## Space-variant filtering

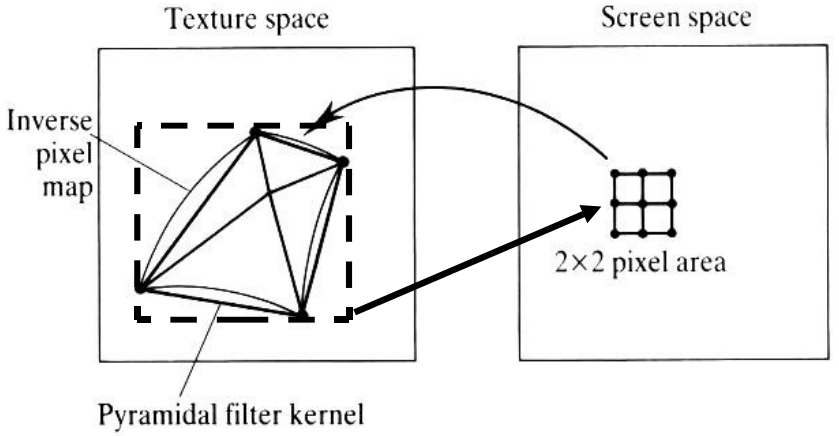
- Mapping from texture space $(u, v)$ to screen space $(x, y)$ not affine
- Filtering changes with position

## Space-variant filtering methods

- Direct convolution
  - Numerically compute the Integral
- Pre-filtering
  - Precompute the integral for certain regions – more efficient
  - Approximate footprint with regions

## Convolution in texture space

- Texels weighted according to distance from pixel center (*e.g.* pyramidal filter kernel)




## Convolution in image space

1. Center the filter function on the pixel (the image space) and find its bounding rectangle.
2. Transform the rectangle to the texture space, where it is a quadrilateral. The sides of this rectangle are assumed to be straight. Find a bounding rectangle for this quadrilateral.
3. Map all pixels inside the texture space rectangle to screen space.
4. Form a weighted average of the mapped texture pixels using a two-dimensional lookup table indexed by each sample's location within the pixel.

## Direct convolution methods are slow

- A pixel pre-image can be arbitrarily large along silhouettes or at the horizon of a textured plane
- Horizon pixels can require averaging over thousands of texture pixels
- Texture filtering cost grows in proportion to projected texture area

## Speed up

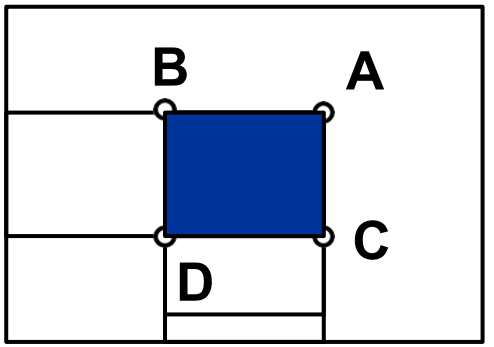
- The texture can be pre-filtered so that during rendering only a few samples will be accessed for each screen pixel

## Two data structures are commonly used for pre-filtering:

- Integrated arrays (***summed area tables***)
- Image pyramids (***mipmaps***) Space-variant filtering

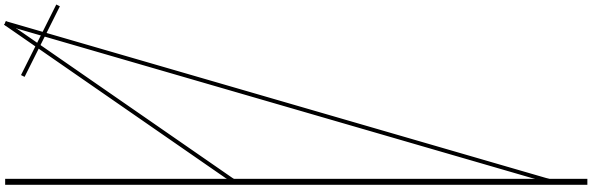
## Summed-Area-Tables

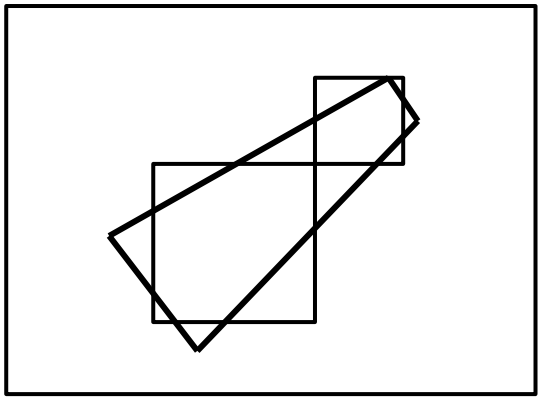
- Per texel, store sum from $(0, 0)$ to $(u, v)$
- Arbitrary rectangle:
  - Area = $A - B - C + D$
- Many bits per texel (sum!)




## Footprint assembly

- Good for space variant filtering
  - *e.g.* inclined view of terrain
- Approximation of the pixel area by rectangular texel-regions
- The more footprints the better accuracy
- Often fixed number of texels  because of economical reasons



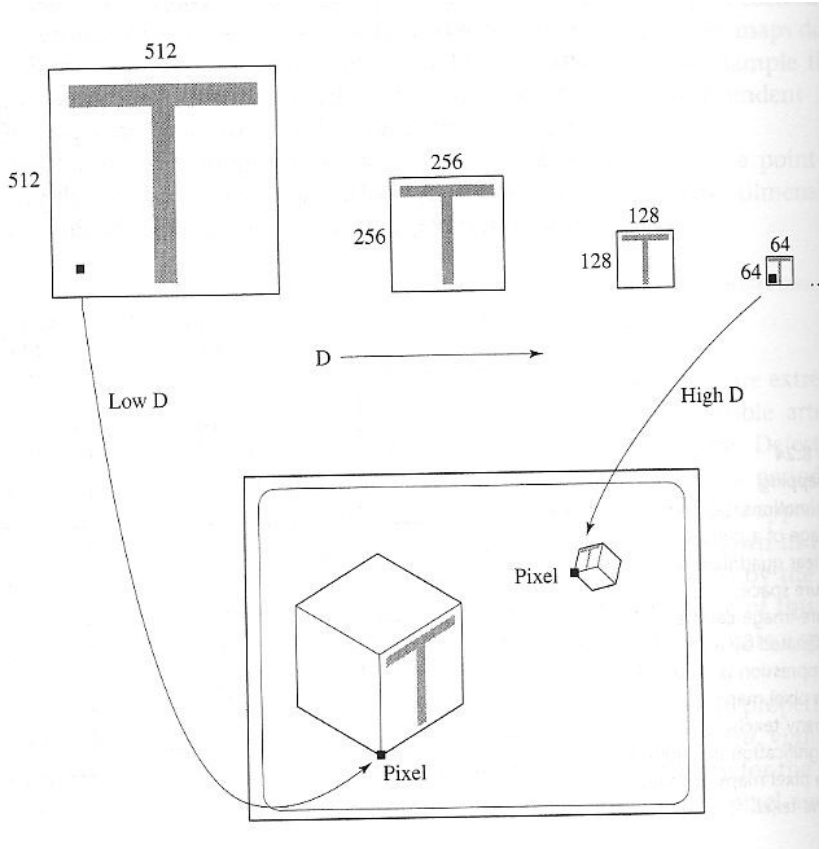


## Texture available in multiple resolutions

- Pre-processing step

## Rendering: select appropriate texture resolution

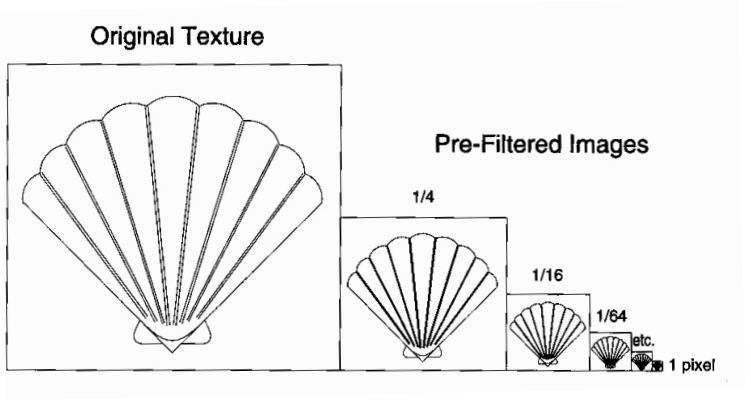
- Selection is usually per pixel!
- Texel size(n) < extent of pixel footprint < texel size(n+1)



## Multum In Parvo (MIP): much in little

## Hierarchical resolution pyramid

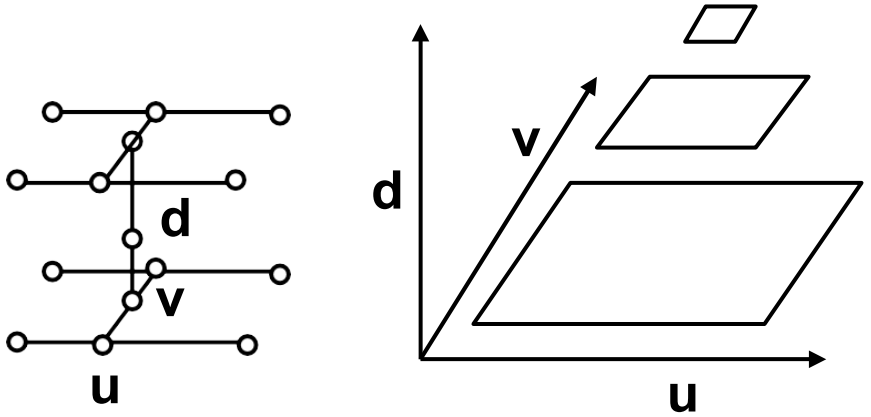
- Repeated averaging over 2x2 texels



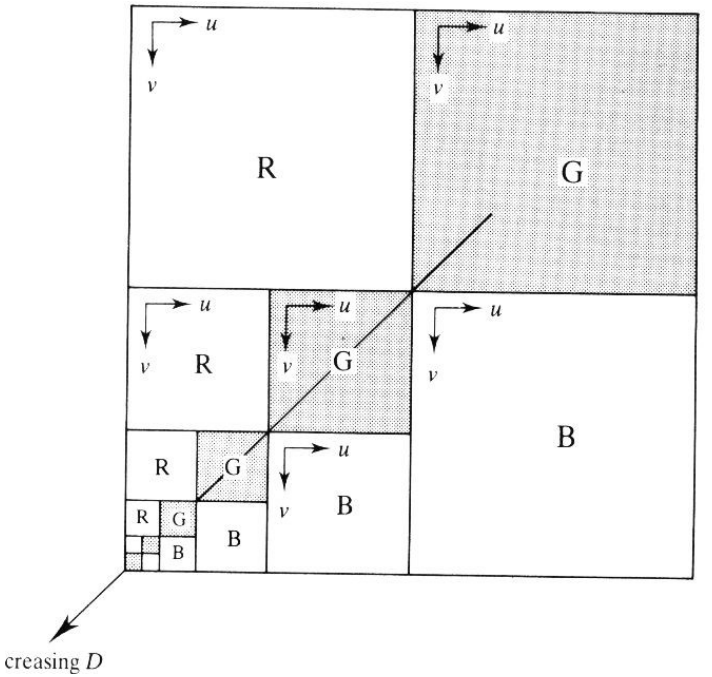

## Rectangular arrangement (RGB)

## Reconstruction

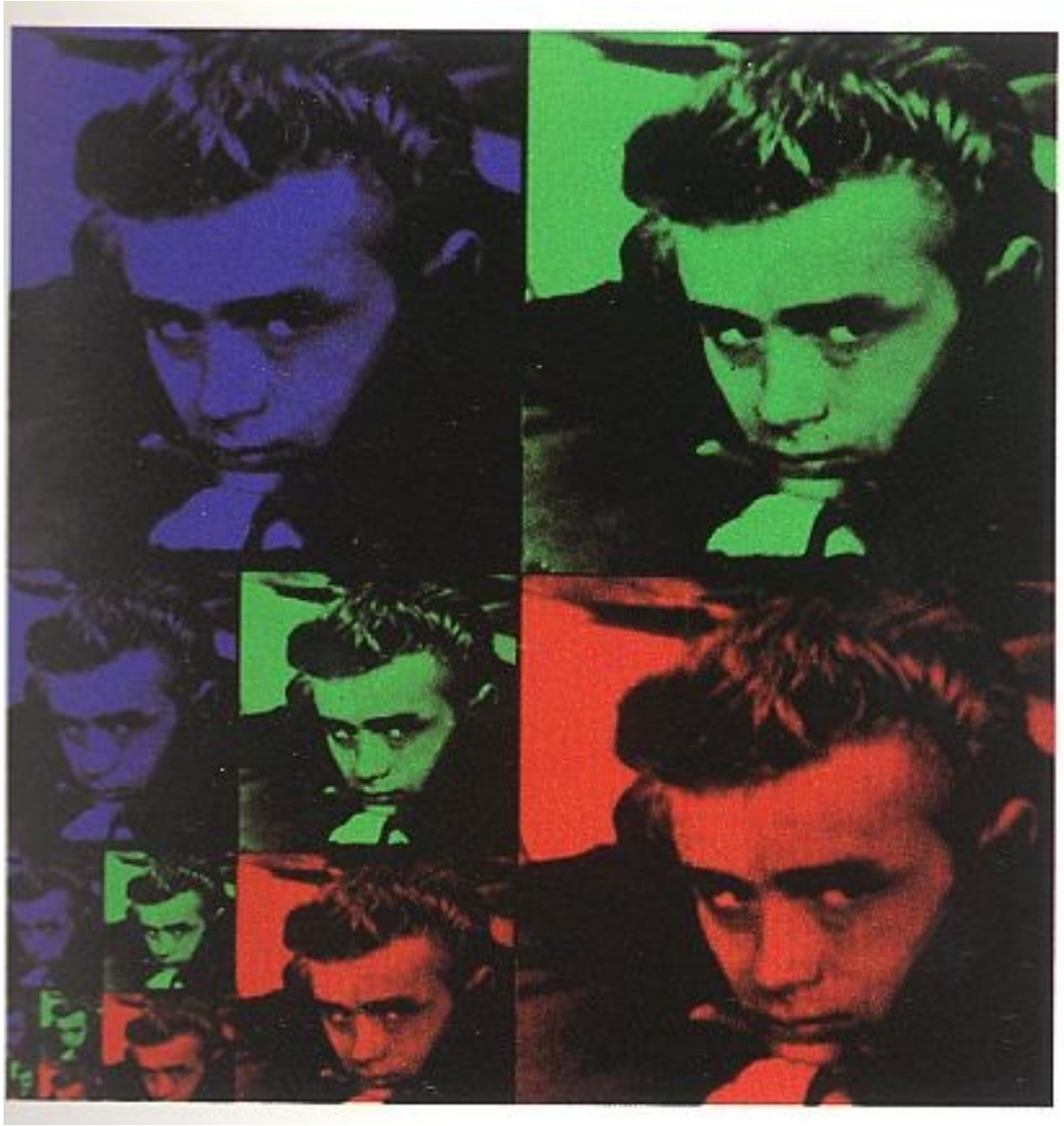
- Tri-linear interpolation of 8 nearest texels
  - Bilinear interpolation in levels *n* and *n+1*
  - Linear interpolation between the two levels




- "Brilinear": Trilinear only near transitions
  - Avoid reading 8 texels, most of the time



## Example:



## Why do graphics cards get faster by MipMapping?

- Bottleneck is memory bandwidth
- Using of texture caches
- Texture minification required for far geometry

## No MipMap

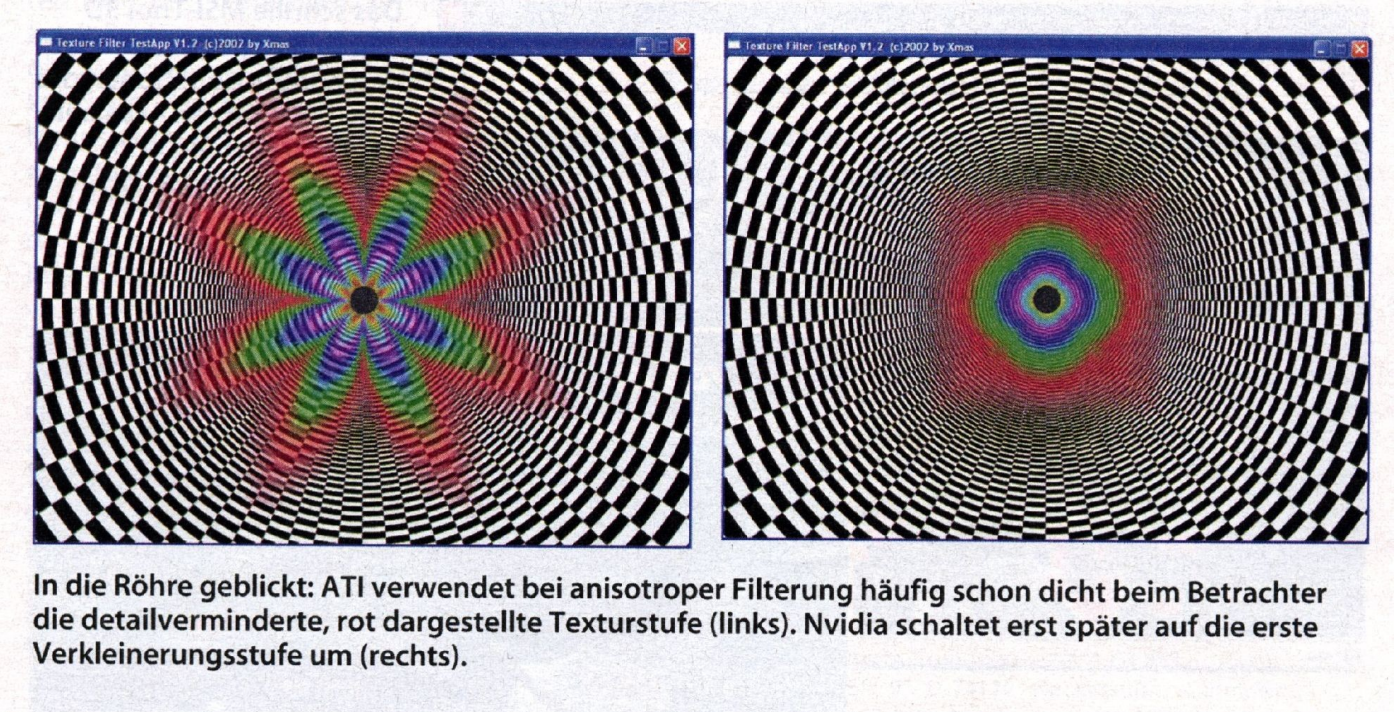
- „Random“ access to texture
- Always 4 new texels

## MipMap

- Neighboring pixel moves nearly about one texel
- Access to 8 texels at a time, but
  - Nearly all texels are still in the cache
  - Average: < 1.5 new texels per pixel

# Footprint Assembly on GPUs

- Integration Across Footprint of Pixel
- HW: Choose samples that best approximate footprint
- Weighted average of samples




In die Röhre geblickt: ATI verwendet bei anisotroper Filterung häufig schon dicht beim Betrachter die detailverminderte, rot dargestellte Texturstufe (links). Nvidia schaltet erst später auf die erste Verkleinerungsstufe um (rechts).

# Lecture 8:

# Texturing | Part 2

Contents

## To apply textures we need 2D coordinates on surfaces

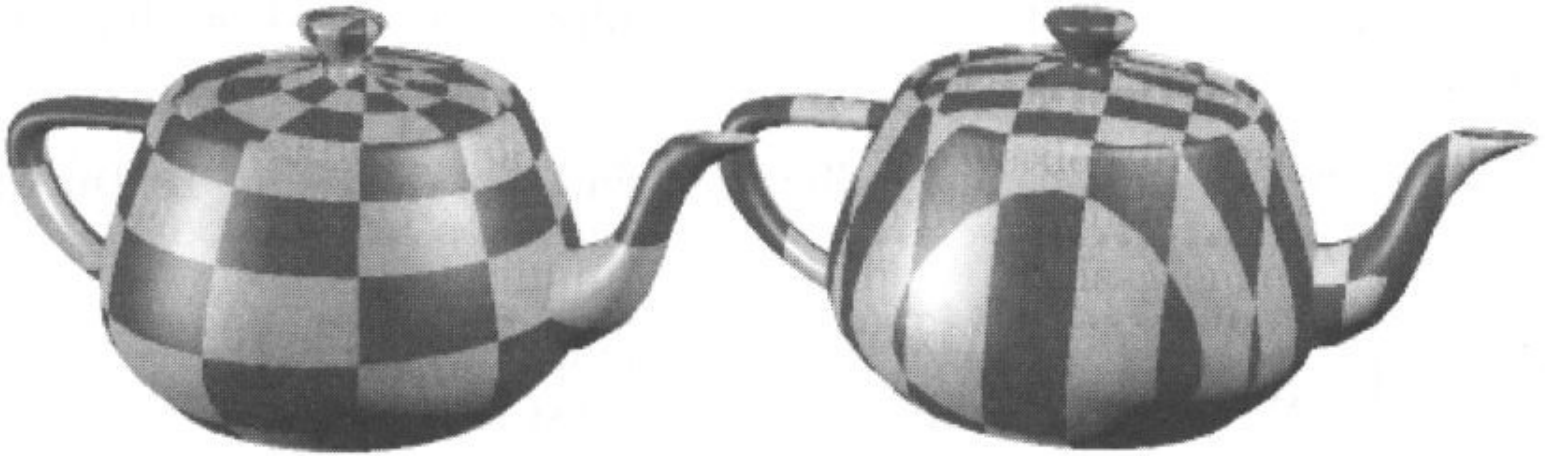
- → Parameterization!

## Some objects have a natural parameterization

- **Sphere**: spherical coordinates $(\theta, \varphi) = (2\pi \cdot u, \pi \cdot v)$
- **Cylinder**: cylindrical coordinates $(\theta, z) = (2\pi \cdot u, H \cdot v)$
- **Parametric** surfaces (such as B-spline or Bezier surfaces → later)

## Parameterization less obvious for

- Polygons, implicit surfaces, …



## Piecewise planar object surface patches

- Has implicit parameterization (*e.g.* barycentric coordinates)
- But we need more control: Placement of triangle in texture space

## Assign texture coordinates $(u, v)$ to each vertex$(x_0, y_0, z_0)$

## Apply viewing projection $(x_0, y_0, z_0) \rightarrow (x, y)$

## Yields texture transformation (warping) $(u, v) \rightarrow (x, y)$

$$x = \frac{au+bv+c}{gu+hu+i} \quad y = \frac{du+ev+f}{gu+hu+i}$$

- In homogeneous coordinates

$$(x, y) = \left(\frac{x\prime}{w}, \frac{y\prime}{w}\right)$$

$$(u, v) = \left(\frac{u\prime}{q}, \frac{v\prime}{q}\right)$$

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u' \\ v' \\ q \end{bmatrix}$$

- Transformation coefficients determined by 3 pairs $(u, v) \rightarrow (x, y)$
- Invertible if points are non-collinear

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} u' \\ v' \\ q \end{bmatrix}$$

## Inverse transform $(x, y) \rightarrow (u, v)$

$$\begin{bmatrix} u' \\ v' \\ q \end{bmatrix} = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} ei - fh & ch - bi & bf - ce \\ fg - di & ai - cg & cd - af \\ dh - eg & bg - ah & ae - bd \end{bmatrix} \begin{bmatrix} x' \\ y' \\ w \end{bmatrix}$$

- $(u, v) = \left(\frac{u\prime}{q}, \frac{v\prime}{q}\right) \qquad (x', y', w) = (x, y, 1)$

## Coefficients must be calculated for each triangle

## Scan-line rendering

- Incremental bilinear interpolation of $(u', v', q)$ in screen space

## Ray tracing

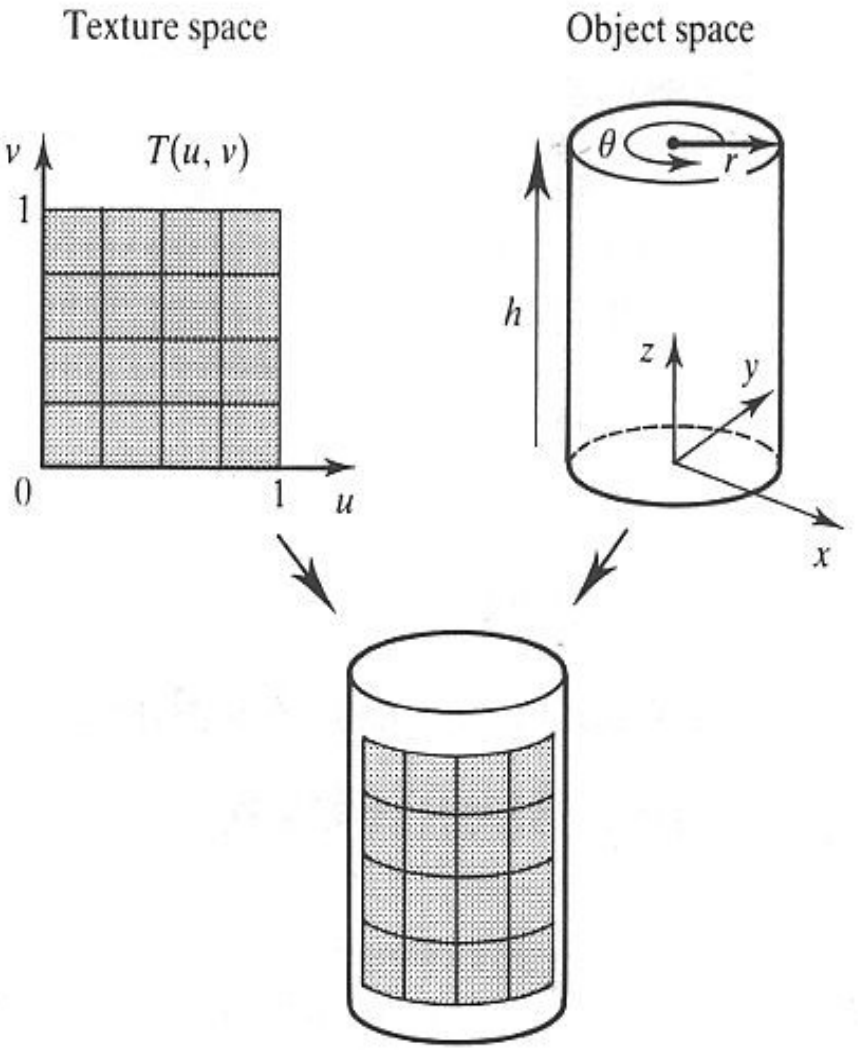
- Evaluation at each intersection

## Transformation from texture space to the cylinder parametric representation can be written as:

- $(\theta, z) = (2\pi \cdot u, H \cdot v)$
- where $H$ is the height of the cylinder.

## The surface coordinates in the Cartesian reference frame can be expressed as:

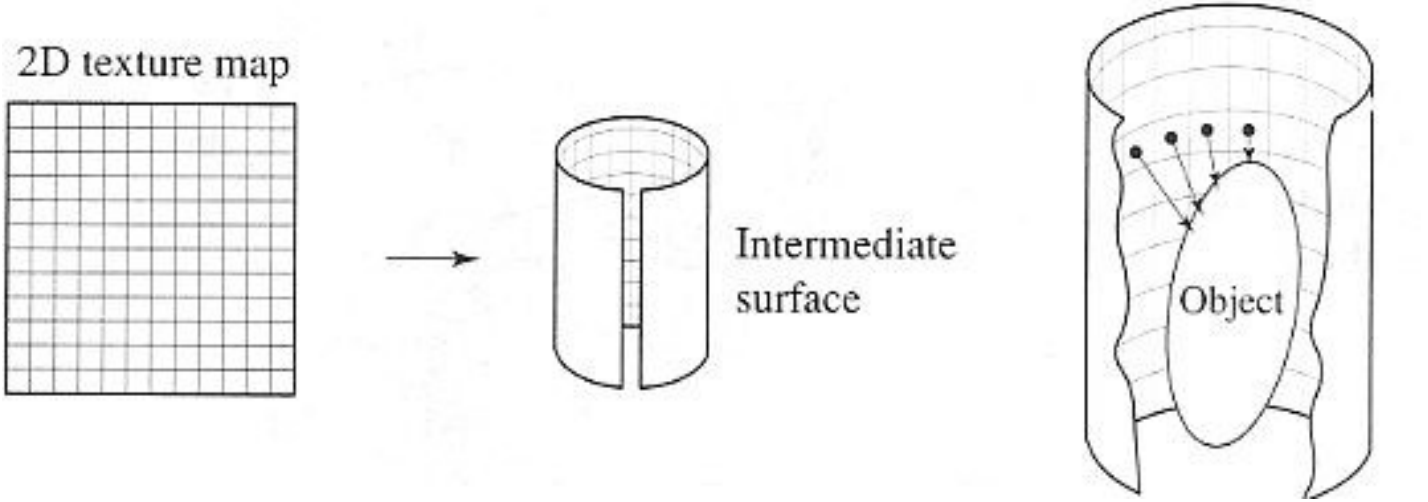
- $x_0 = r\cos\theta$
- $y_0 = r\sin\theta$
- $z_0 = h$



## Inverse Mapping for arbitrary 3D surfaces too complex

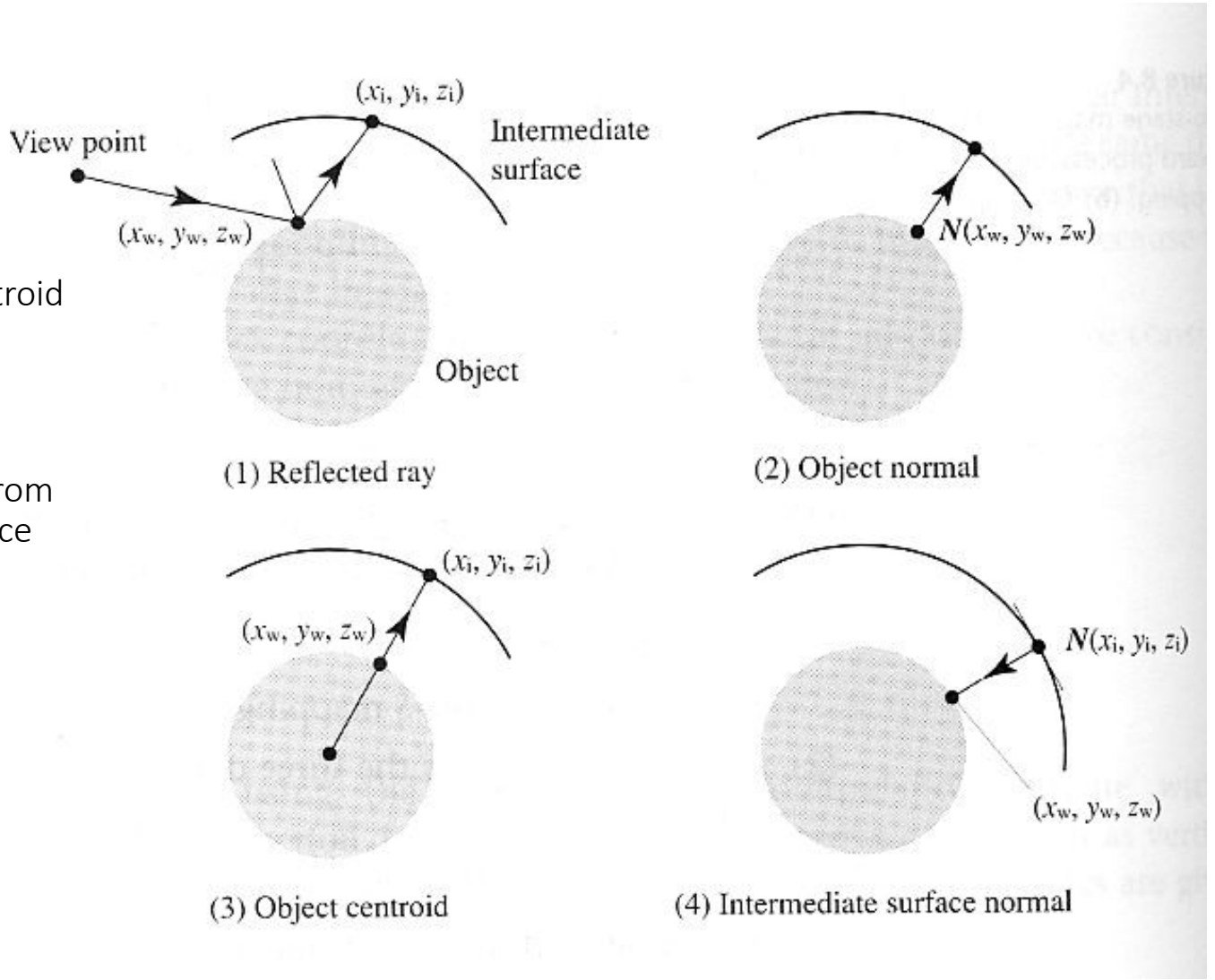
## Approximation technique is used:

- S – mapping: Mapping from 2D texture space to a simple 3D intermediate surface, which is a reasonable approximation of the destination surface (*e.g.*, cylinder, sphere)
- O – mapping: Mapping from the intermediate surface to the destination object surface



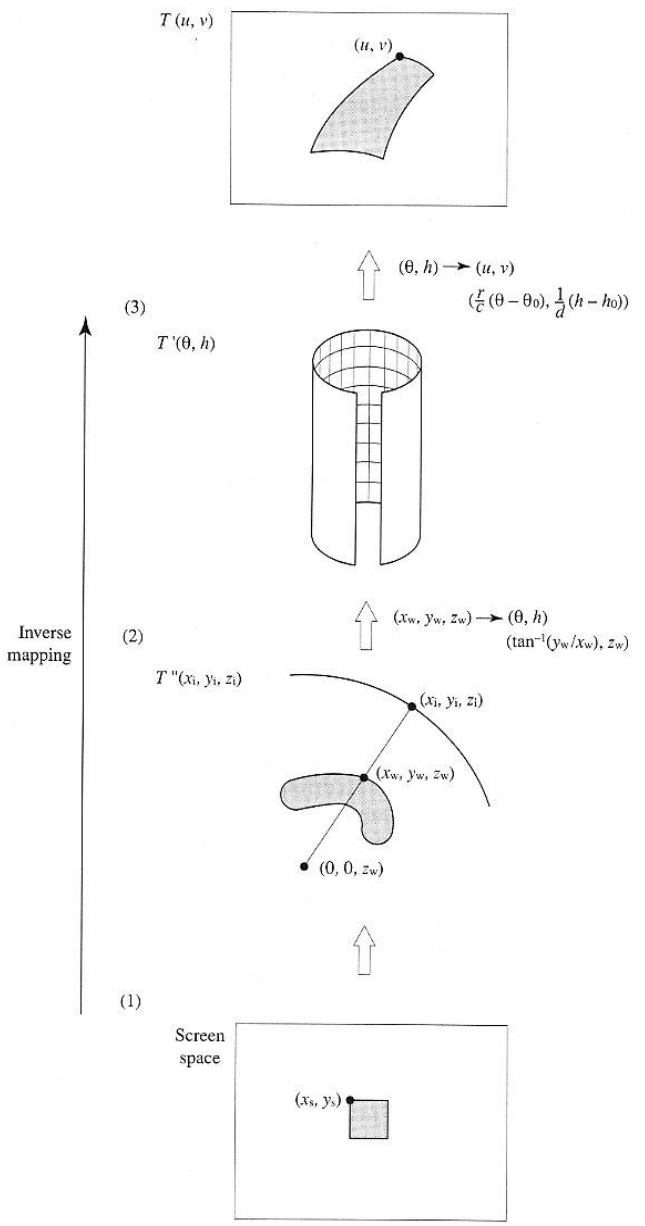
# Determine point on intermediate surface through

- Reflected view ray
  - Reflection or environment mapping
- Normal mapping
- Line through object centroid
- Shrinkwrapping
  - Forward mapping
  - Normal mapping from intermediate surface



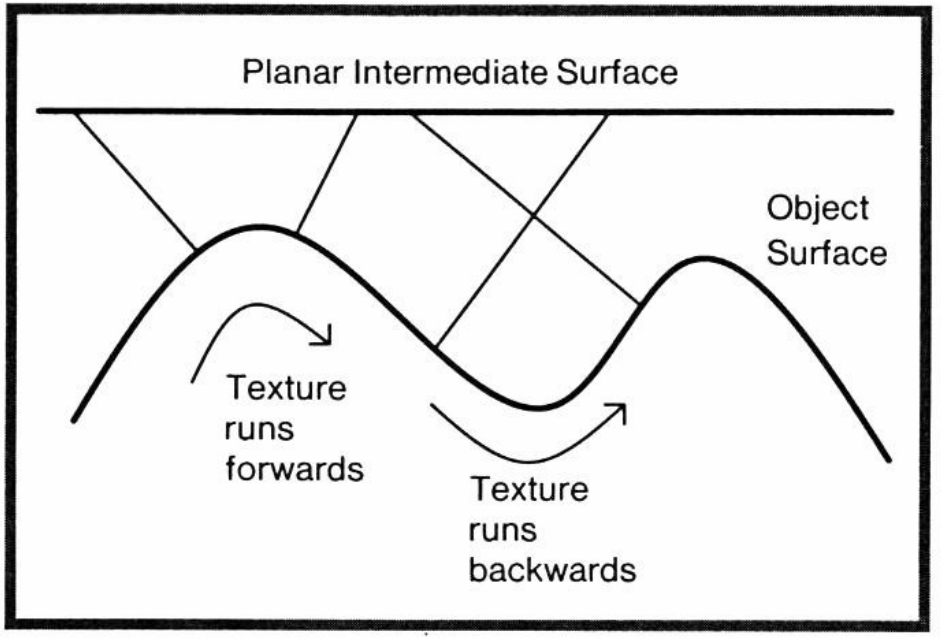
# Inverse two-stage mapping

- Map 4 screen pixels to object surface
- O - mapping
  - Shrinkwrapping: Intersection of line from cylinder axis through object point
- S - mapping
  - Inverse-map cylinder surface to texture map



# Problems

- May introduce undesired texture distortions if the intermediate surface differs much from the destination surface
- Still often used in practice because of its simplicity

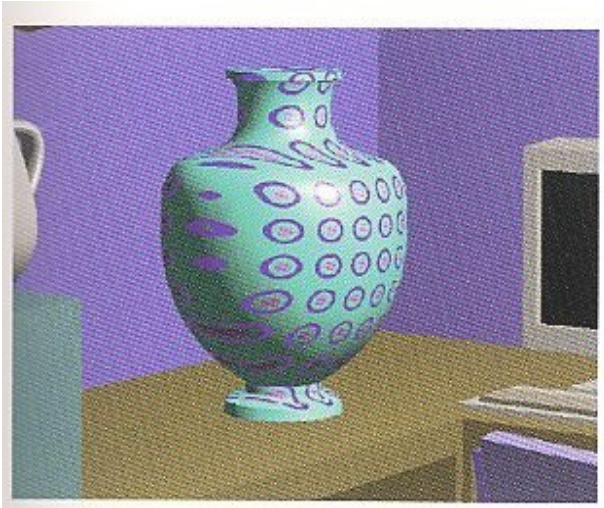



**Surface concavities can cause the texture pattern to reverse if the object normal mapping is used.**

# Example:

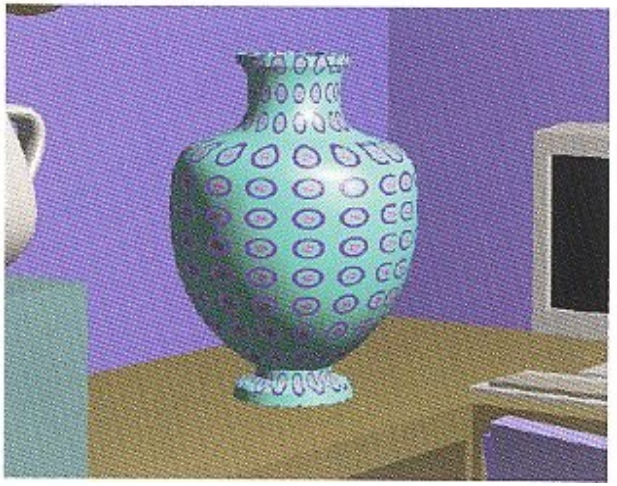
- different intermediate surfaces:

## Plane



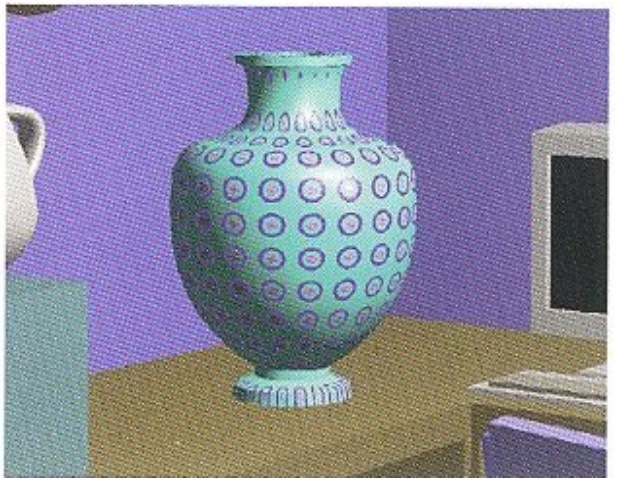
Strong distortion where object surface normal ⊥ plane normal

## Cylinder



Reasonably uniform mapping (symmetry!)

## Sphere



Problems with concave regions

## Project texture onto object surfaces

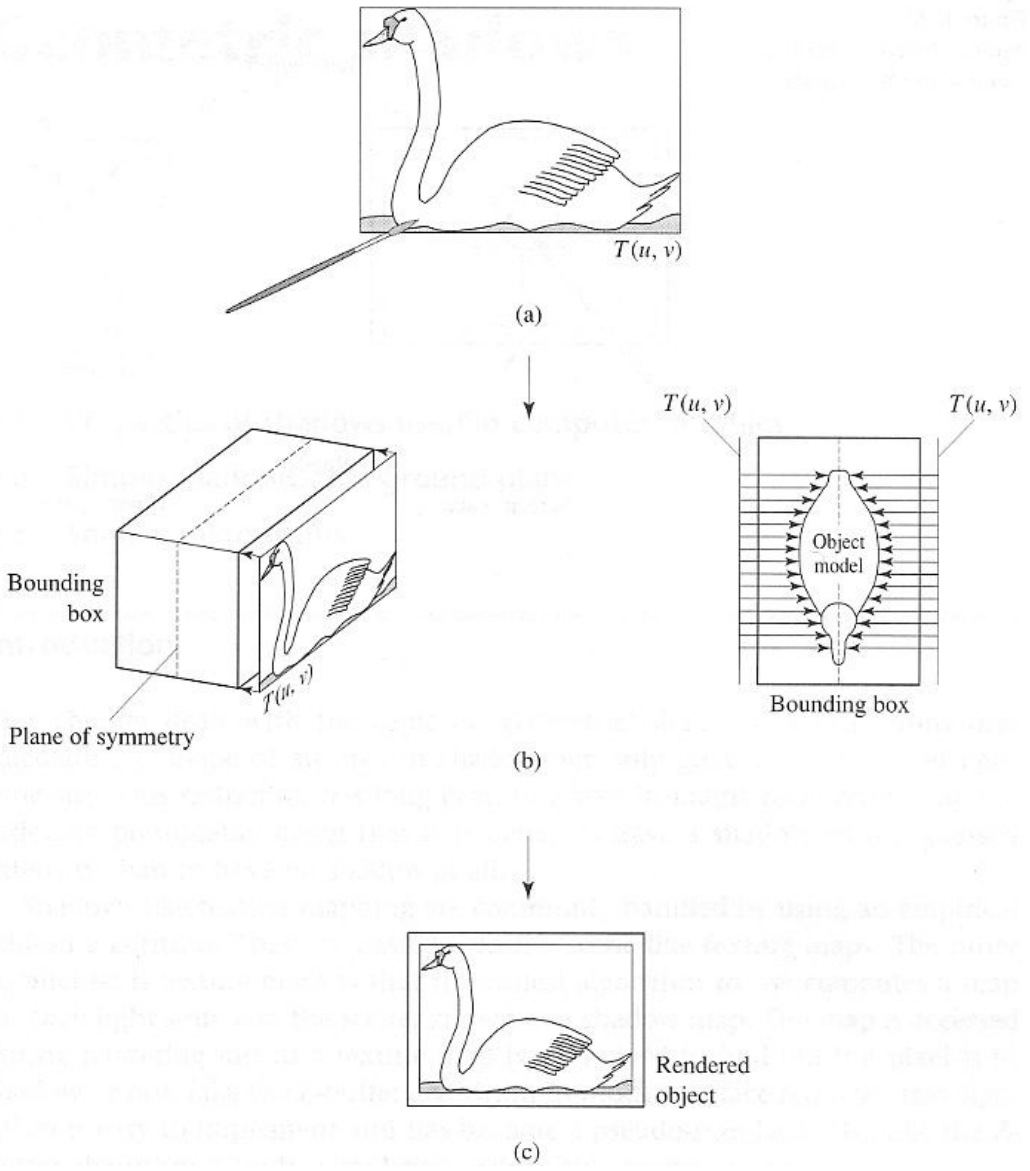
- Slide projector

## Parallel or perspective projection

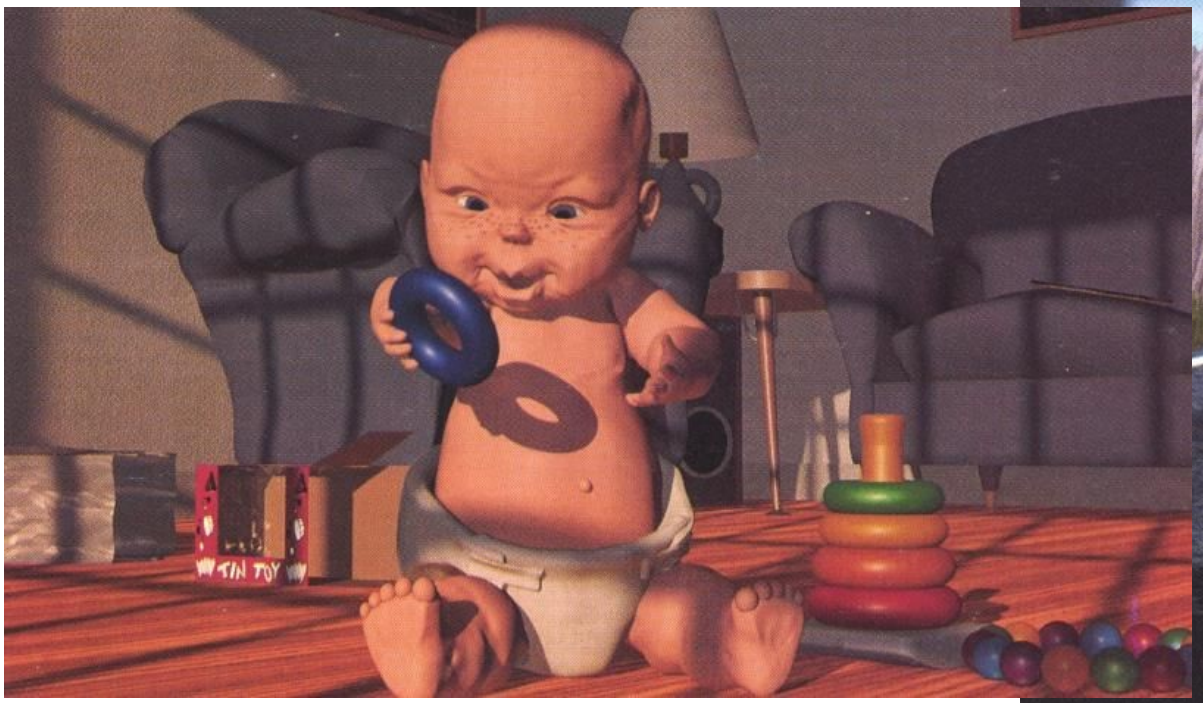
## Use photographs as textures

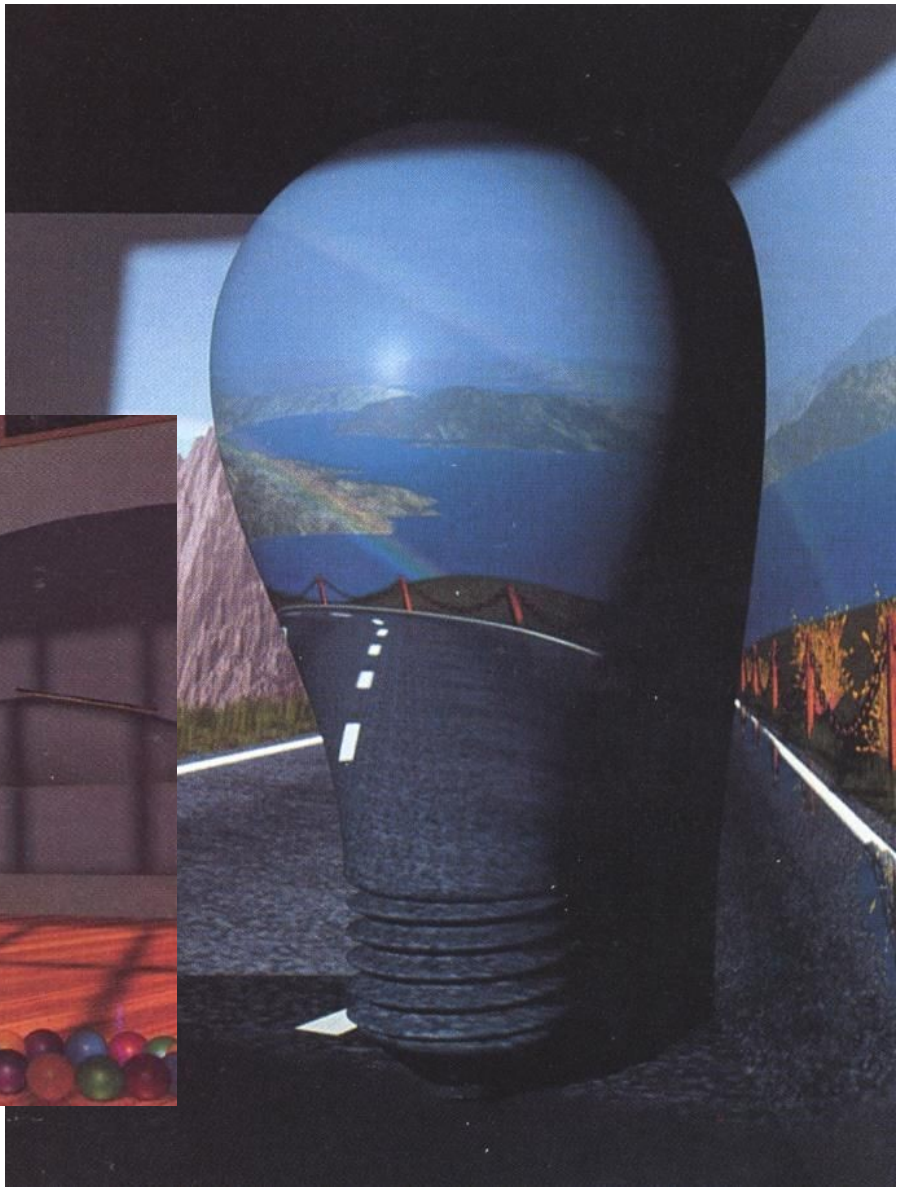
## Multiple images

- View-dependent texturing



## Examples:



## Also called Environment Mapping

## Mirror reflections

- Surface curvature: beam tracing
- Map filtering

## Reflection map parameterization

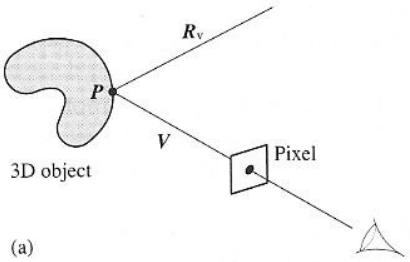
- Intermediate surface in 2-stage mapping

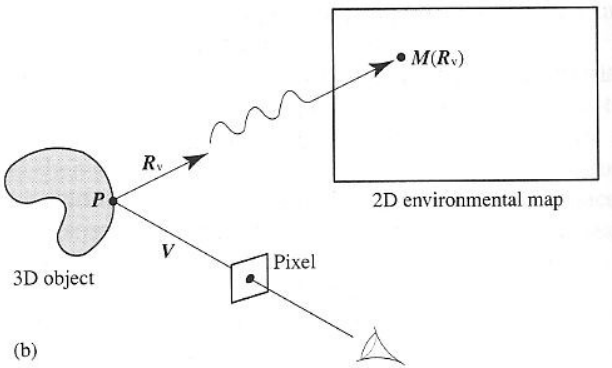
## Light sources distant from object

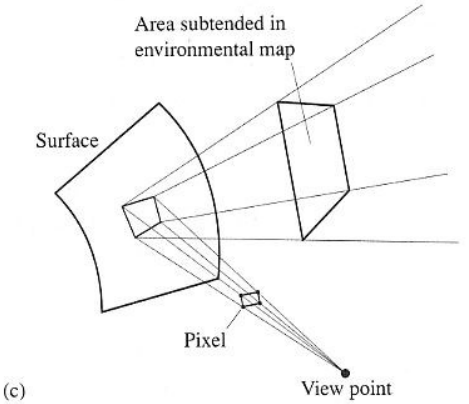
- Parallax-free illumination
- No self-reflections, object concavities

## Option: Separate map per object

- Reflections of other objects
- Maps must be recomputed after changes



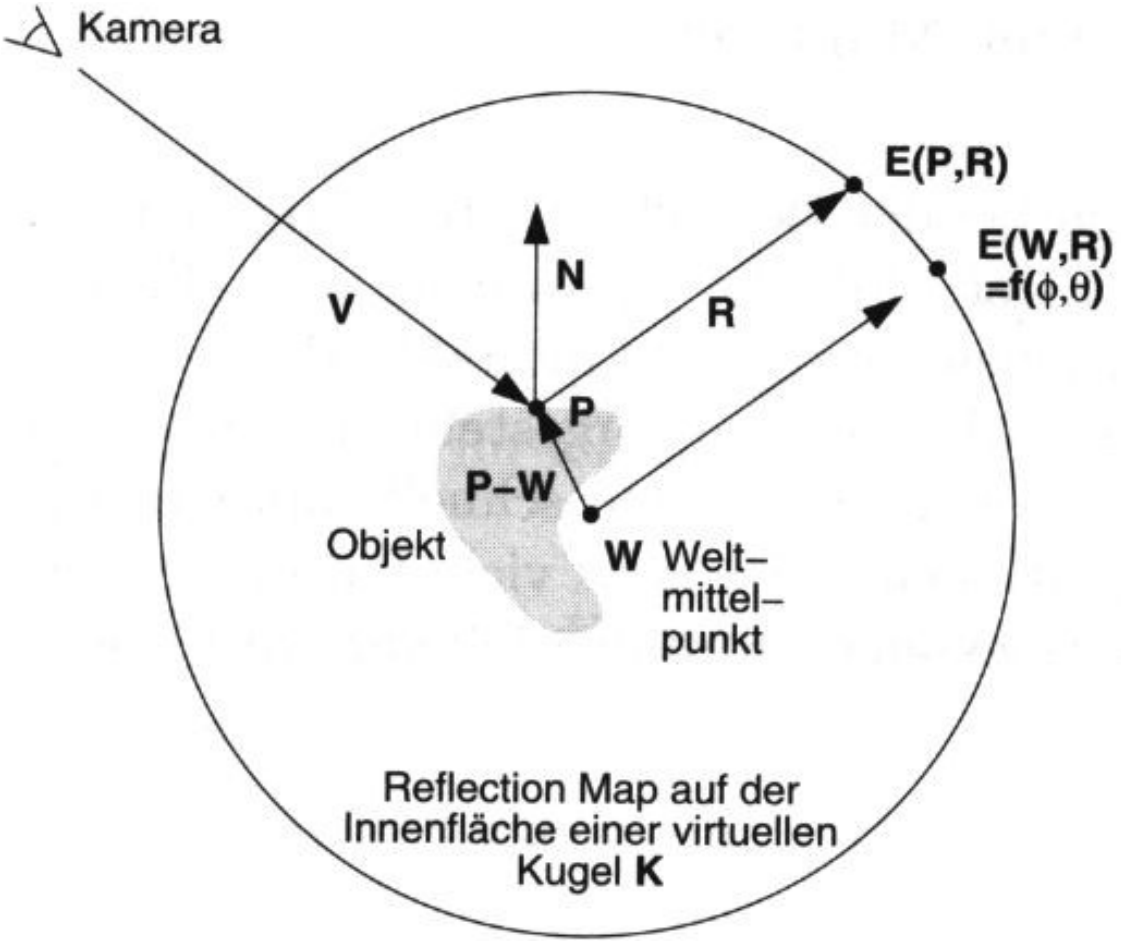
# Generating spherical maps (original 1982 / 83)

- *i.e.* photo of a reflecting sphere (gazing ball)

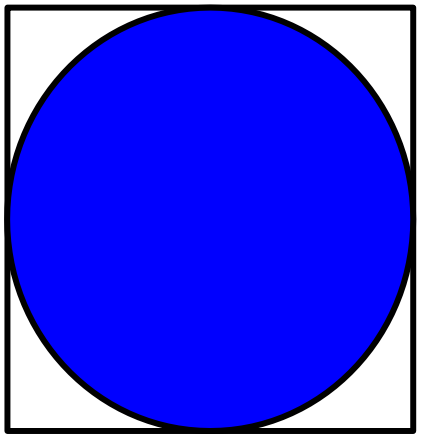
Peter Chou

## Spherical parameterization

## O-mapping using reflected view ray intersection



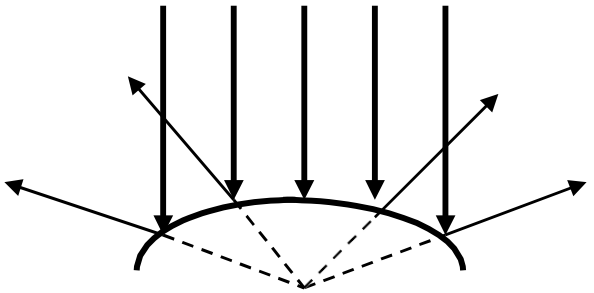
## Spherical mapping



- Single image
- Bad utilization of the image area
- Bad scanning on the edge
- Artifacts, if map and image do not have the same direction

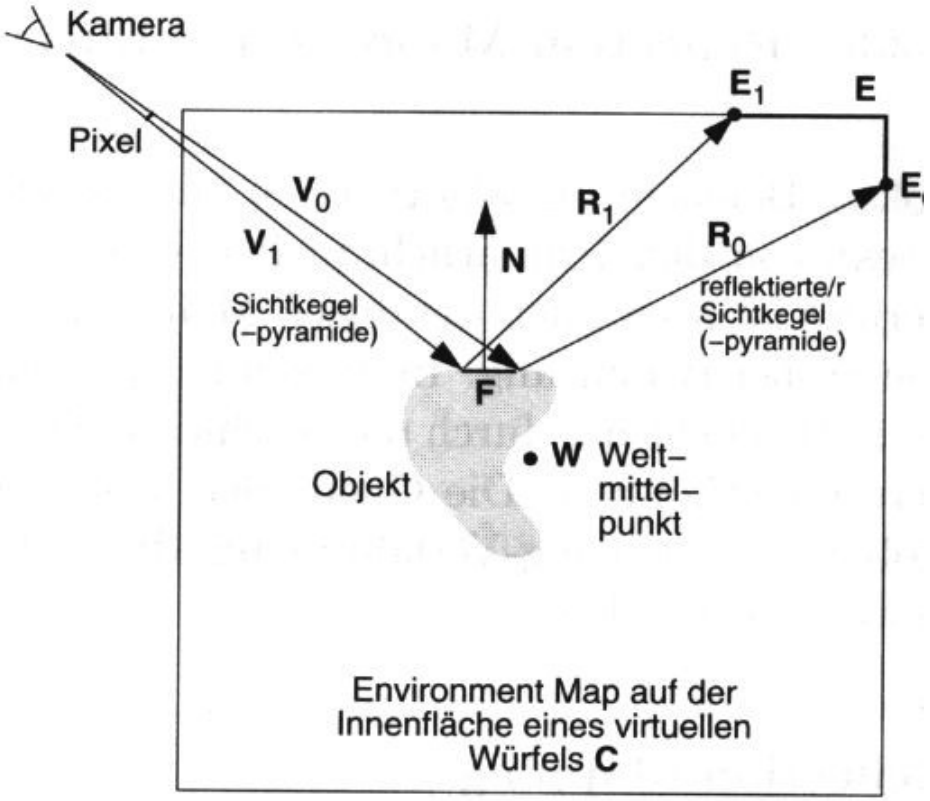
## Parabolic mapping

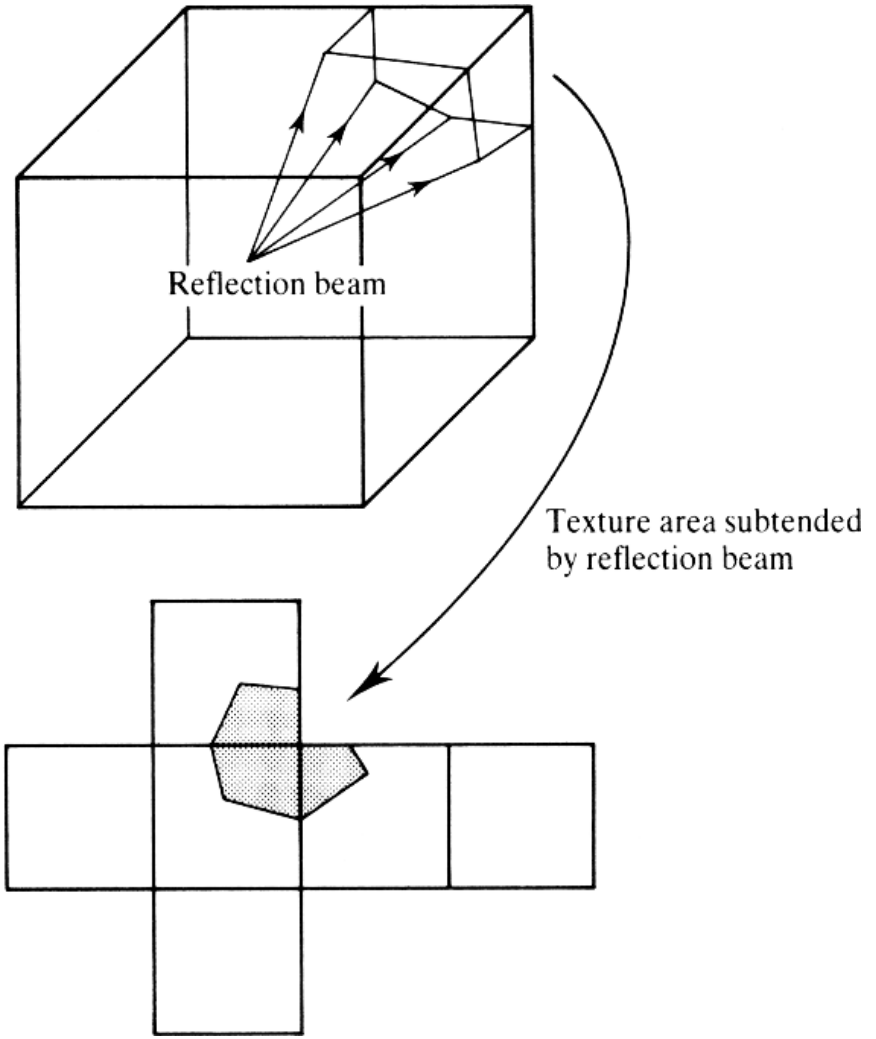
- Subdivide in 2 images (facing and back facing side)
- Less bias on the edge
- Arbitrarily reusable
- Supported by OpenGL extensions

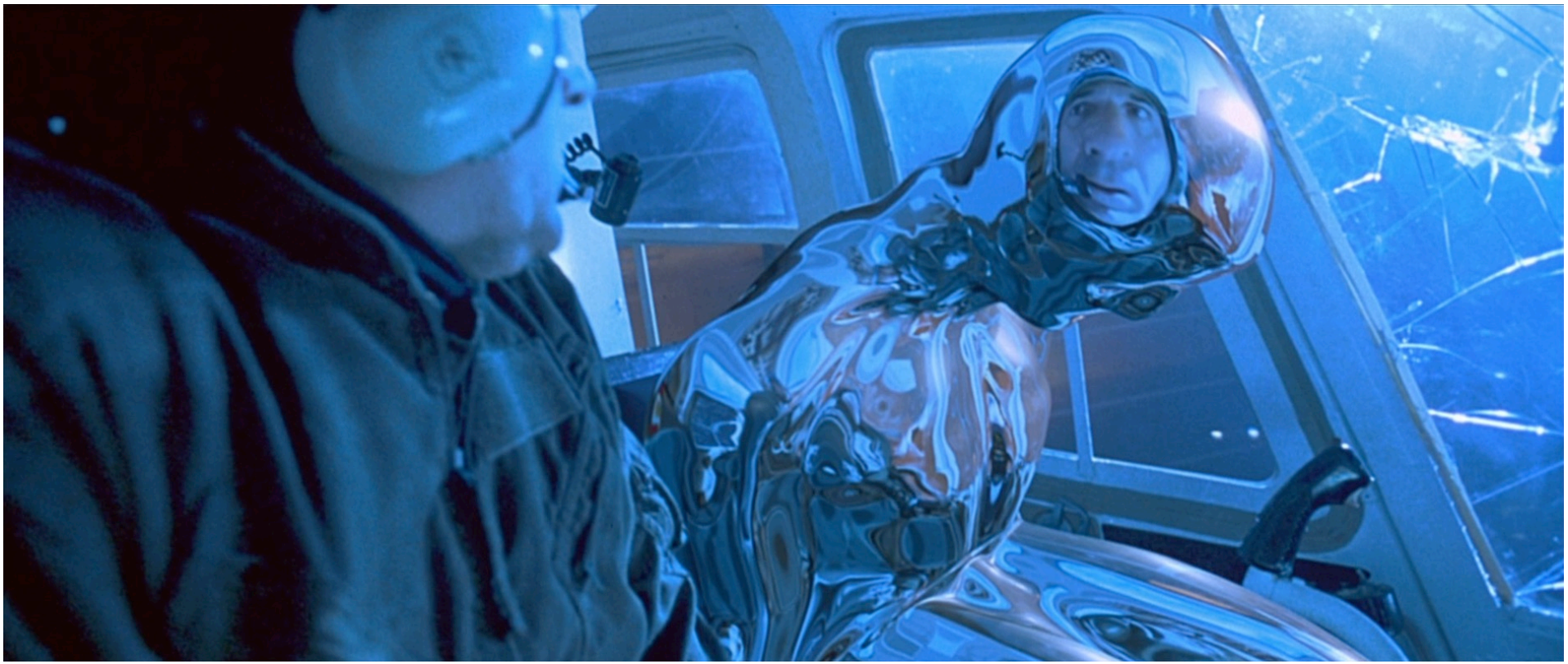


# Cubical environment map, cube map, box map

- Enclose object in cube
- Images on faces are easy to compute
- Poorer filtering at edges
- Support in OpenGL





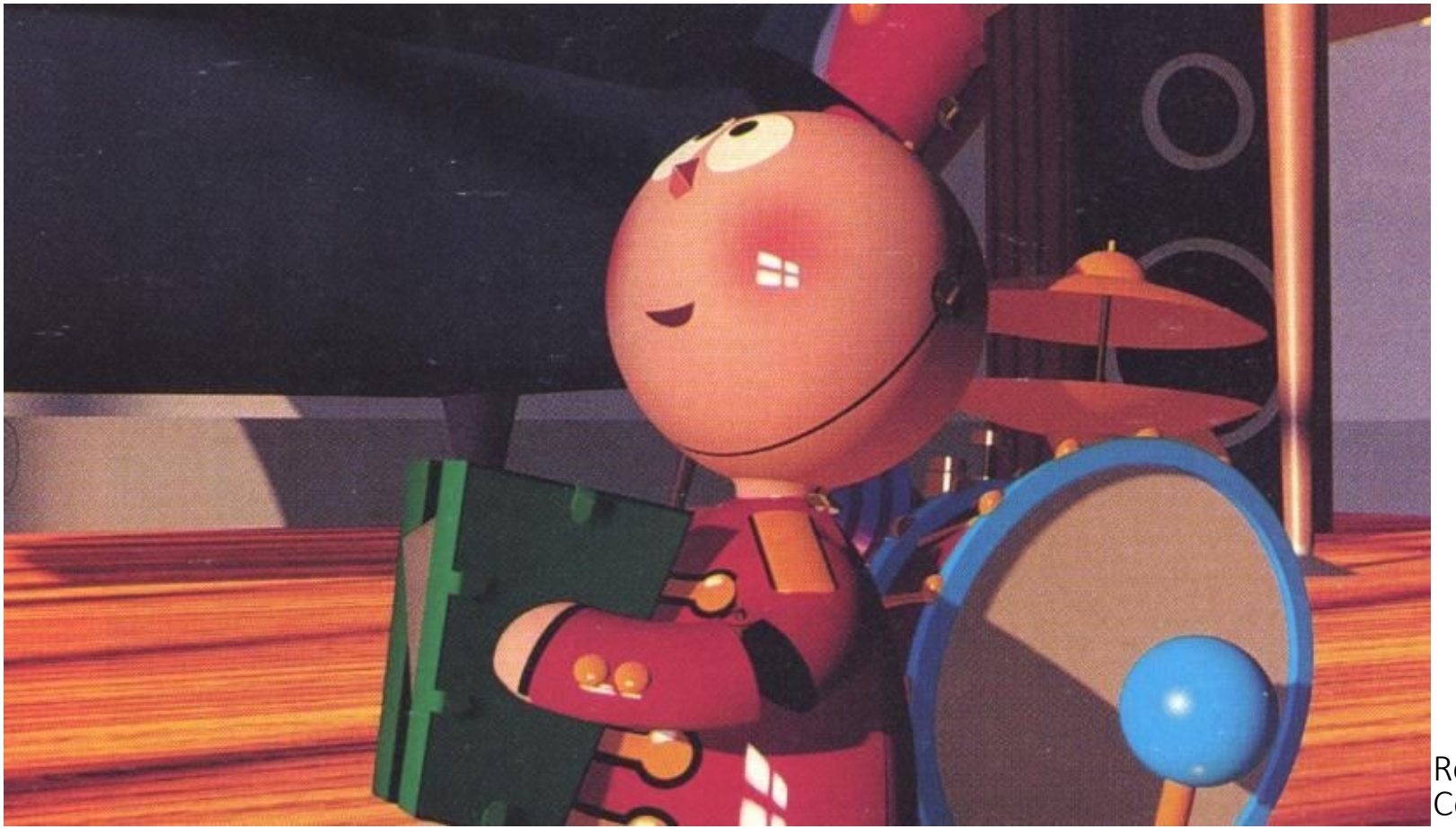
## Example:



Terminator II motion picture

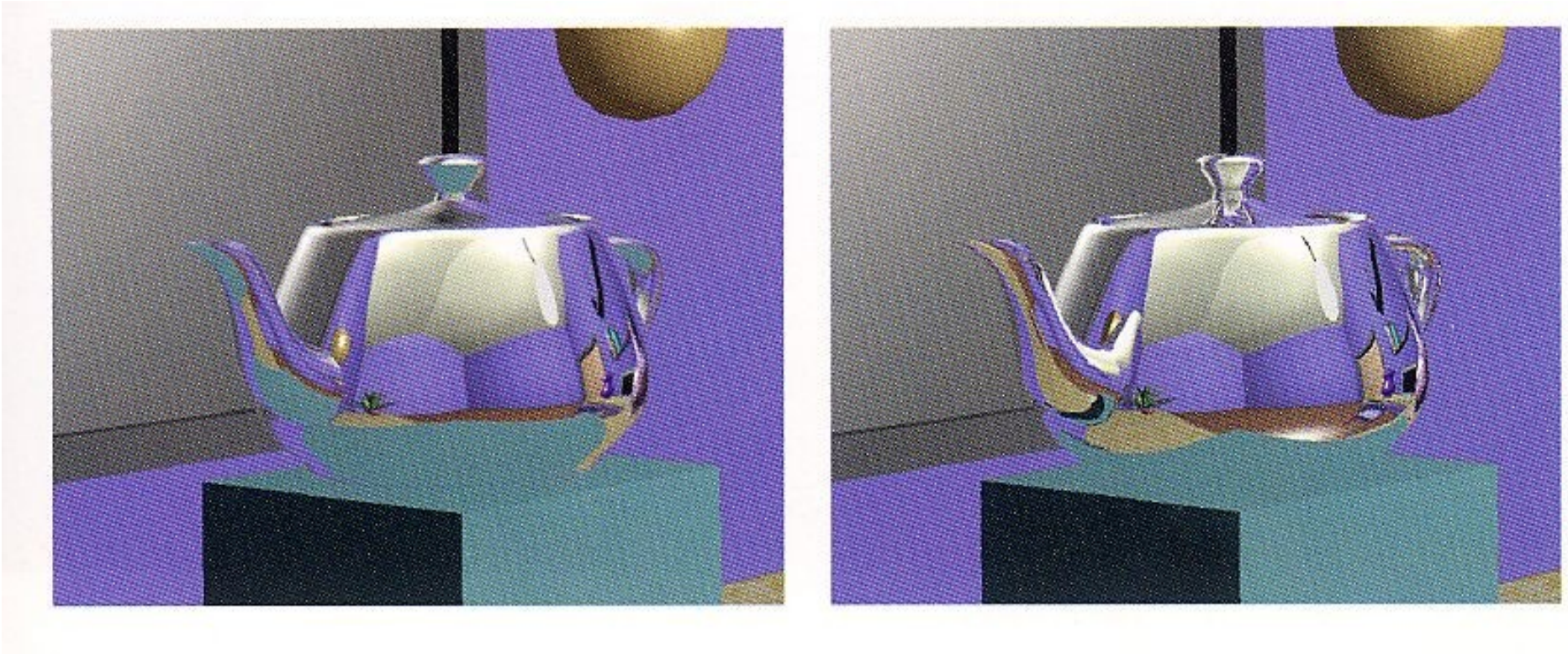
# Example 2: Reflection mapping with Phong reflection

- Two maps: diffuse & specular
- Diffuse: index by surface normal
- Specular: indexed by reflected view vector



RenderMan Companion

## Differences ?



# Recursive Ray Tracing

Raytracing depth 0

Raytracing depth 1

Raytracing depth 2

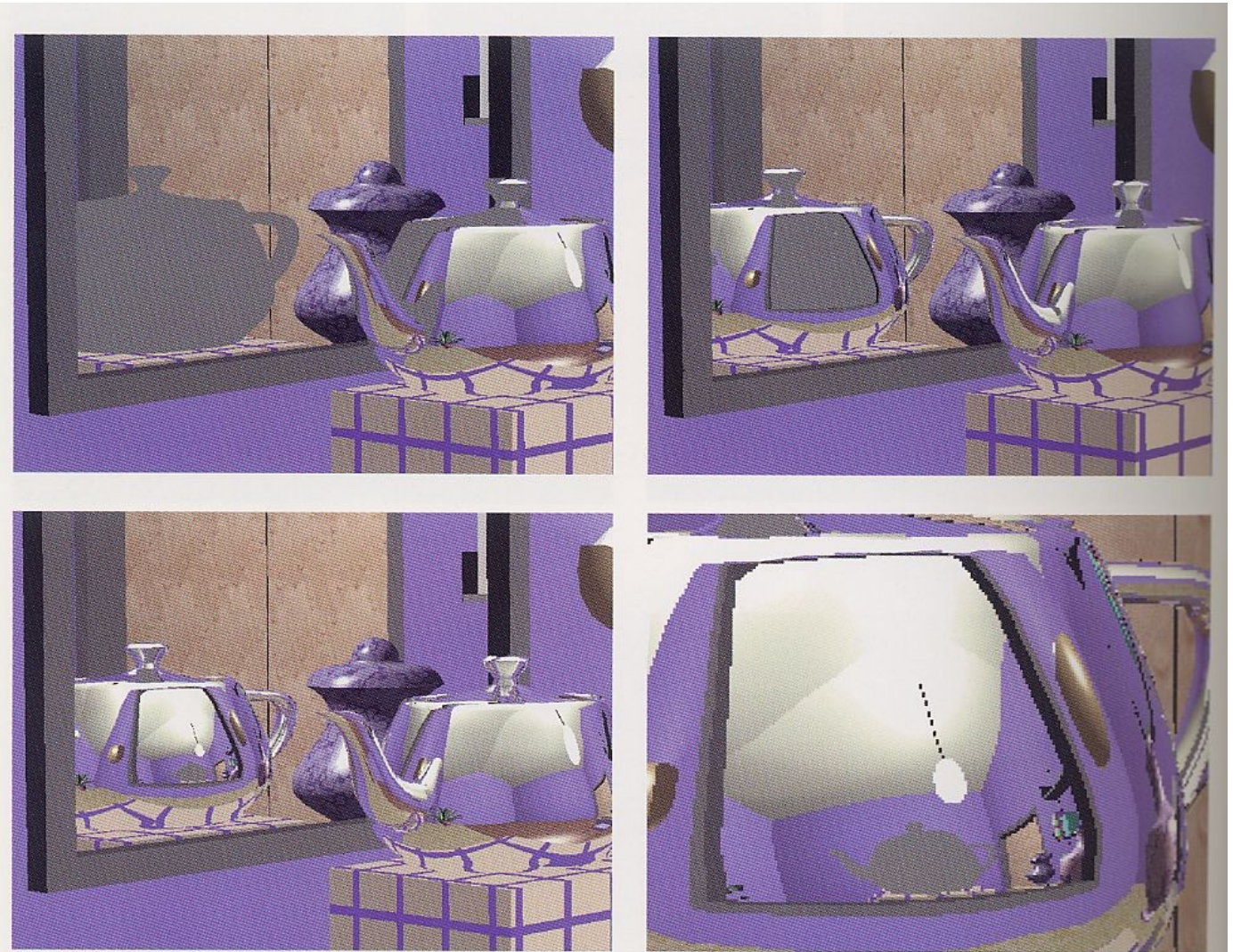


**Figure 18.11**
A recursive depth demonstration. The trace terminates at depth 2, 3, 4 and 5 (zoom image) respectively. 'Unassigned' pixels are coloured grey. Bad aliasing as a function of recursive depth (the light cable) is apparent.

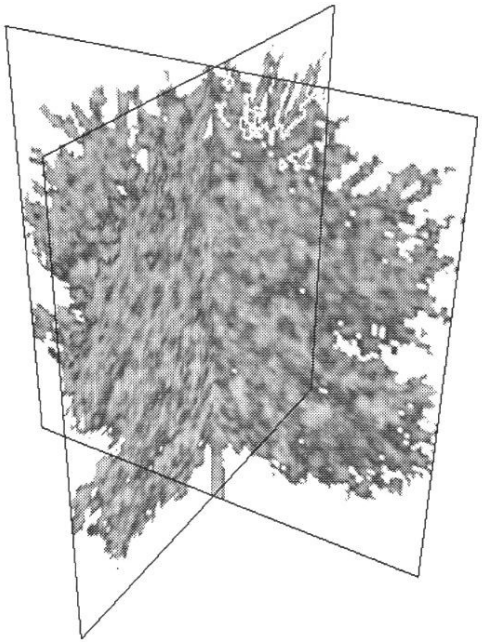
## Single textured polygons

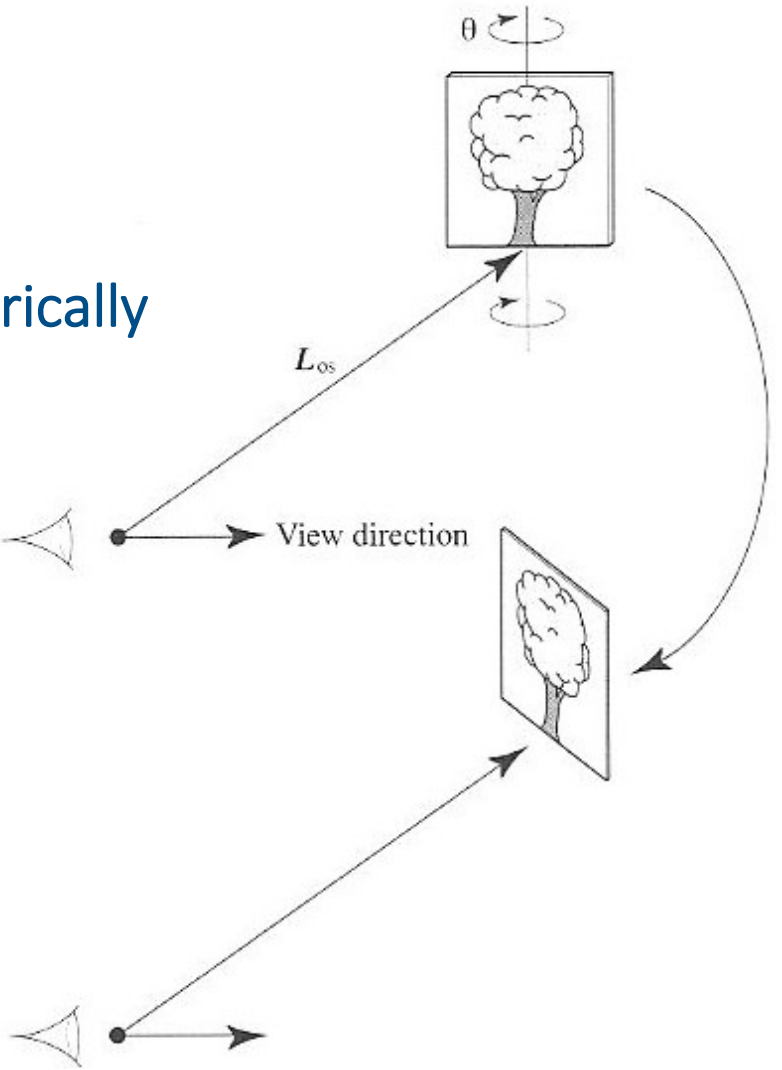
- Often with transparency texture

## Rotates, always facing viewer

## Used for rendering distant objects

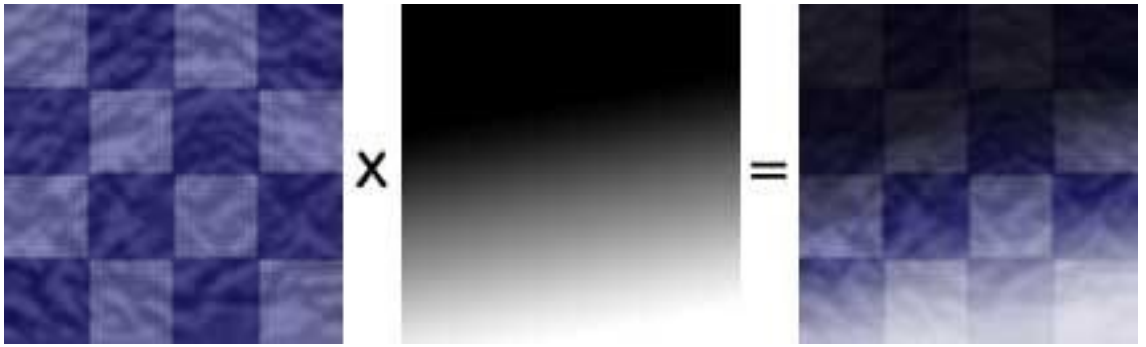
## Best results if approximately radially or spherically symmetric



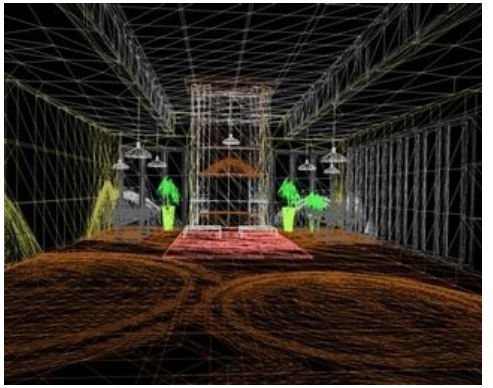


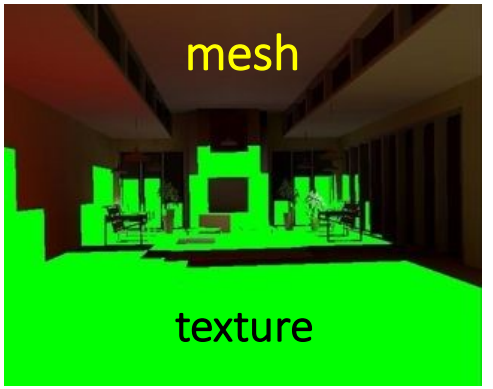
# Light maps (*i.e.* in Quake)

- Pre-calculated illumination (local irradiance)
  - Often very low resolution
- Multiplication of irradiance with base texture
  - Diffuse reflectance only
- Provides surface radiosity
  - View-independent
- Animated light maps
  - Animated shadows, moving light spots *etc.*



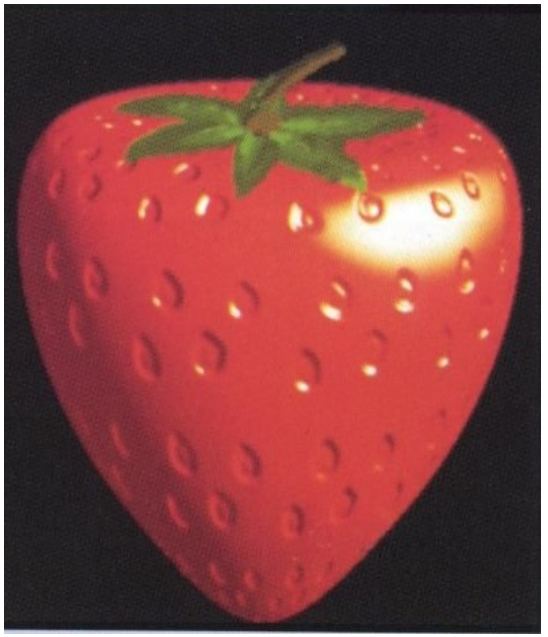

Reflectance Irradiance Radiosity

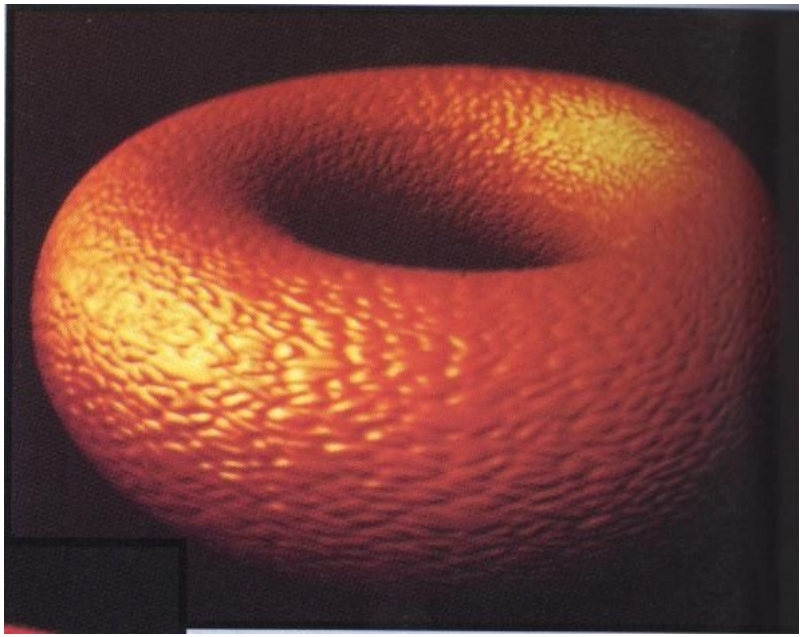





# Modulation of the normal vector

- Surface normals changed only
  - Influences shading only
  - No self-shadowing, contour is **not** altered



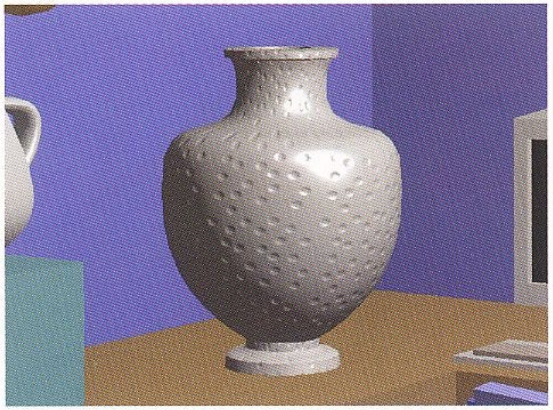


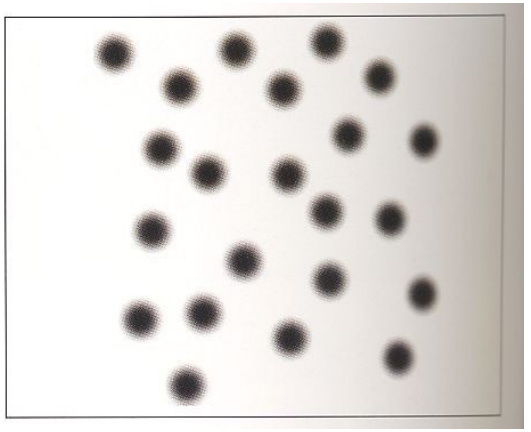
## Original surface $O(u, v)$

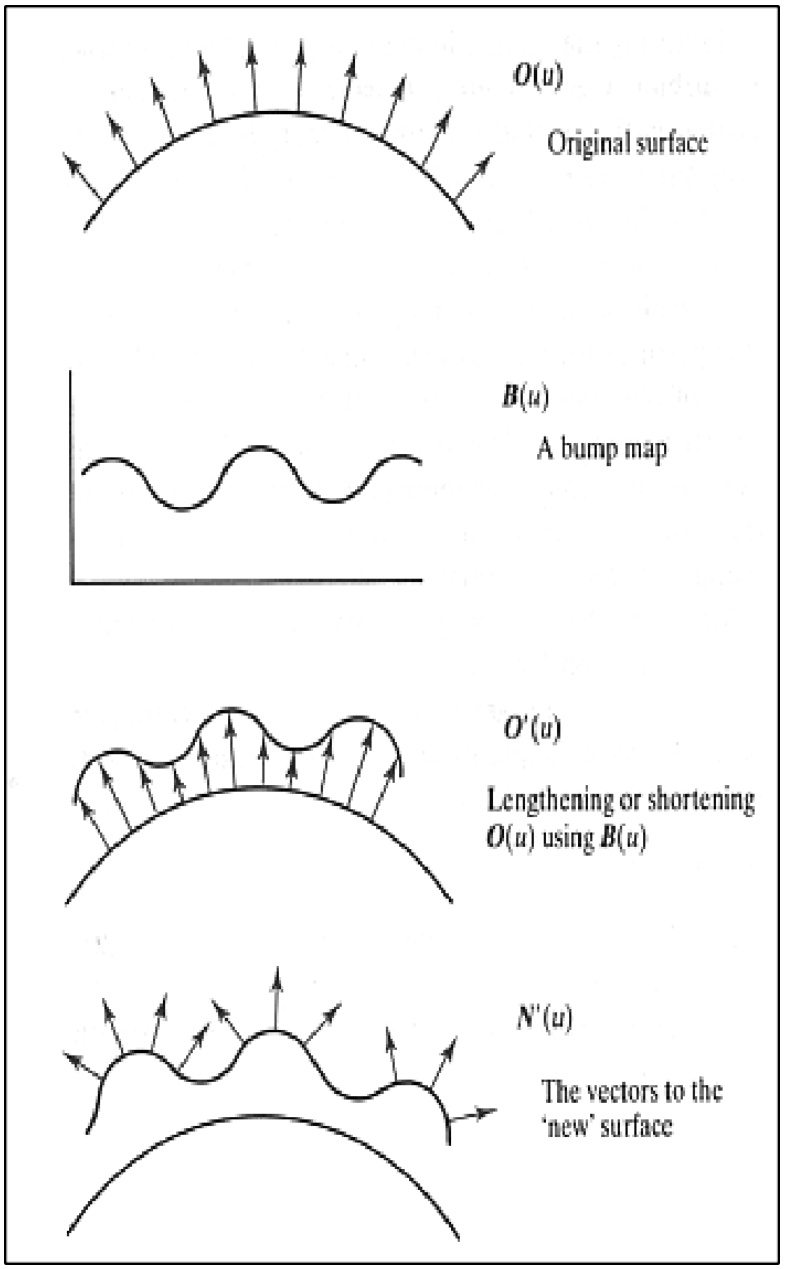
- Surface normals known

## Bump map $B(u, v) \in R$

- Surface is offset in normal direction according to bump map intensity
- New normal directions are calculated $N'(u, v)$ based on displaced surface $O'(u, v)$
- Originals surface is rendered with new normals $N'(u, v)$







$$O'(u,v) = O(u,v) + B(u,v)\frac{N}{|N|}$$

Now differentiating this equation gives:

$$O'_u = O_u + B_u\frac{N}{|N|} + B\left(\frac{N}{|N|}\right)_u$$

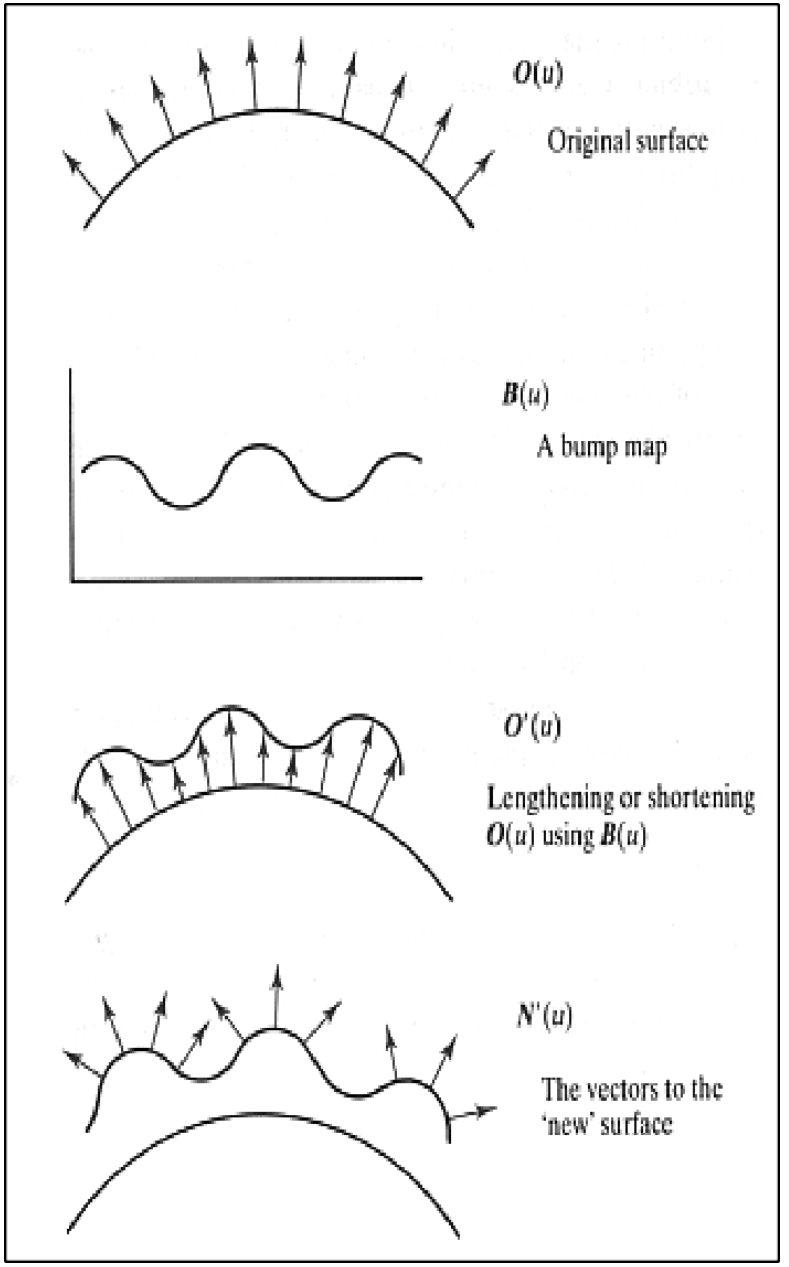
$$O'_v = O_v + B_v\frac{N}{|N|} + B\left(\frac{N}{|N|}\right)_v$$

If $B$ is small (that is, the bump map displacement function is small compared with its spatial extent) the last term in each equation can be ignored and

$$N'(u,v) = O_u \times O_v + B_u\left(\frac{N}{|N|} \times O_v\right) + B_v\left(O_u \times \frac{N}{|N|}\right) + B_u B_v\left(\frac{N \times N}{|N|^2}\right)$$

The first term is the normal to the surface and the last term is zero, giving:

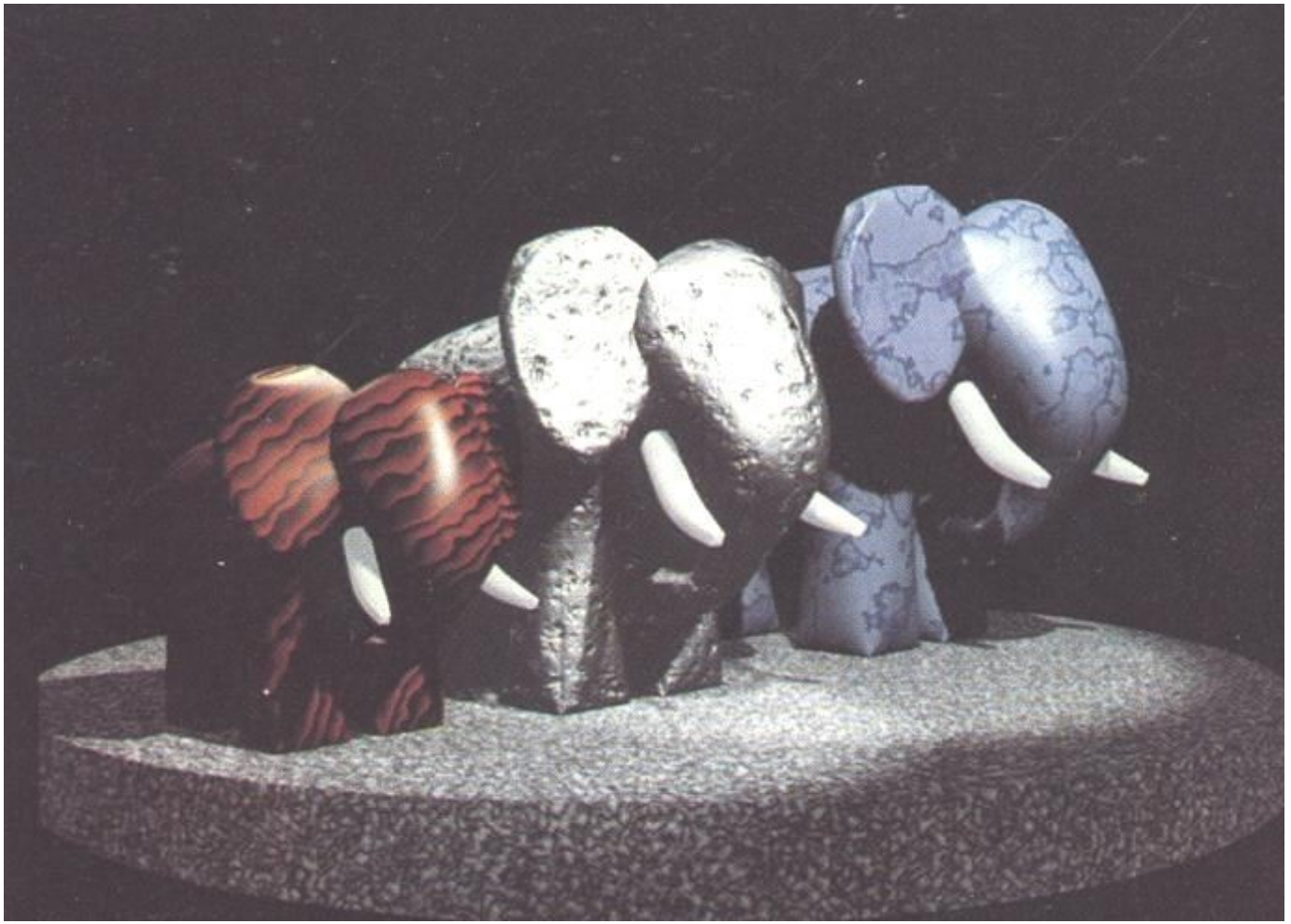
$$D = B_u(N \times O_v) - B_v(N \times O_u)$$



# Carving object shape out of material block

Solid 3D textures (wood, marble)

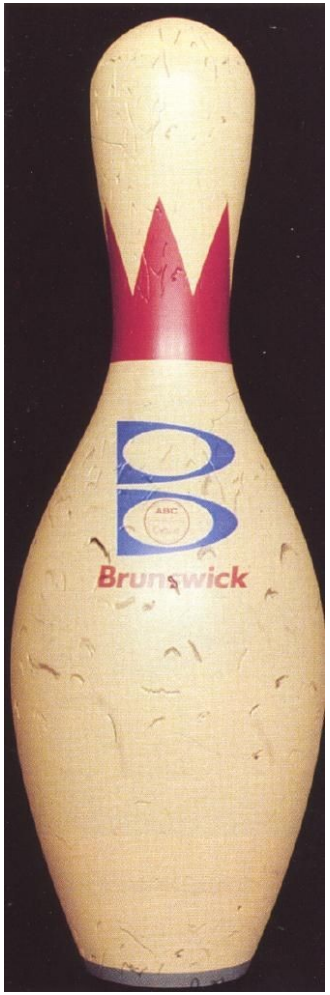
Bump map (middle)



RenderMan Companion

# Complex optical effects

- Combination of multiple textures







RenderMan Companion

## Texture maps (photos, simulations, videos, ...)

- Simple acquisition
- Illumination during acquisition
- Limited resolution, aliasing
- High memory requirements
- Mapping difficult

## Procedural textures

- Non-trivial programming
- Flexibility
- Parametric control
- Unlimited resolution, anti-aliasing possible
- Low memory requirements
- Low-cost visual complexity
- Adapts to arbitrary geometry

## Analytic scalar function of world coordinates $(x, y, z)$

## Texturing: evaluation of function on object surface

- Ray tracing: 3D intersection point with surface

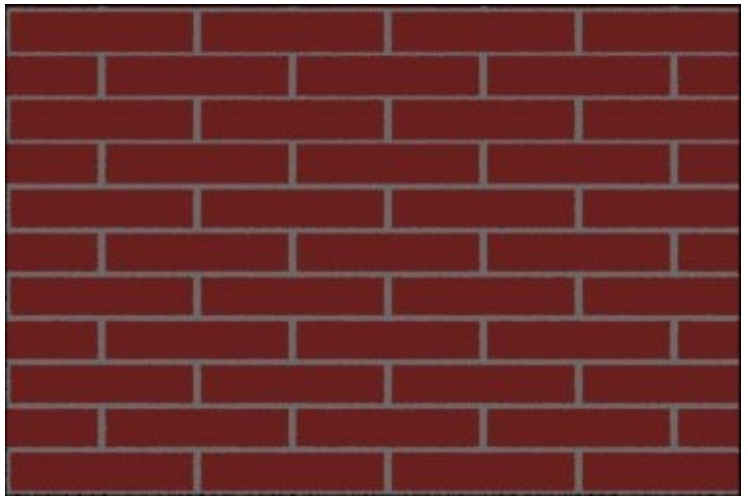
## Textures of natural objects

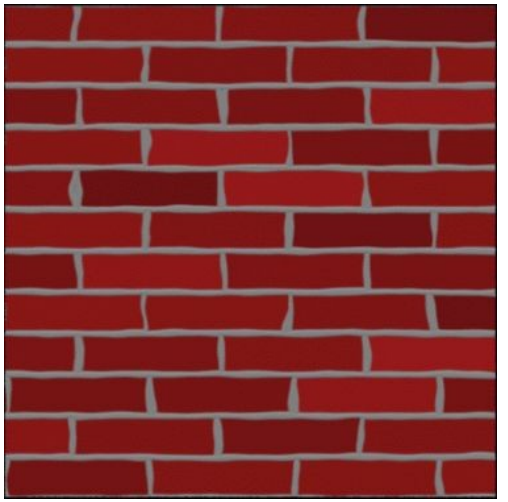
- Similarity between different patches
  - Repetitiveness, coherence
- Similarity on different resolution scales
  - Self-similarity
- But never completely identical
  - Additional disturbances, turbulence, noise

## Procedural texture function

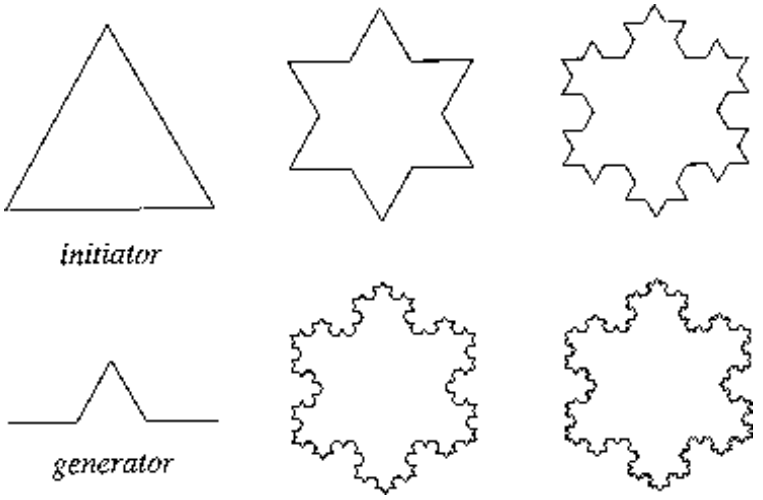
- Mimics statistical properties of natural textures
- Purely empirical approach
  - Looks convincing, but has nothing to do with material's physics

## Translational similarity



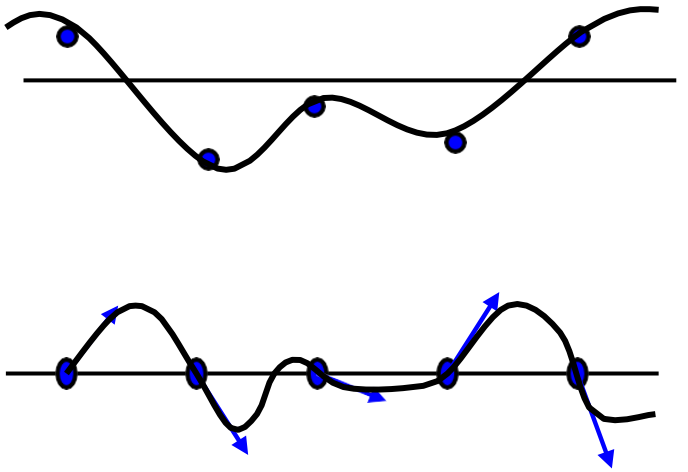


## Similarity on different scales



## Noise $(x, y, z)$

- Statistical invariance under rotation
- Statistical invariance under translation
- Narrow bandpass limit in frequency



## Integer lattice $(i, j, k)$

- Random number at each lattice point $(i, j, k)$
  - Look-up table or hashing function
- Gradient lattice noise
  - Random gradient vectors

## Evaluation at $(x, y, z)$

- Tri-linear interpolation
- Cubic interpolation (Hermite spline → later)

## Unlimited domain

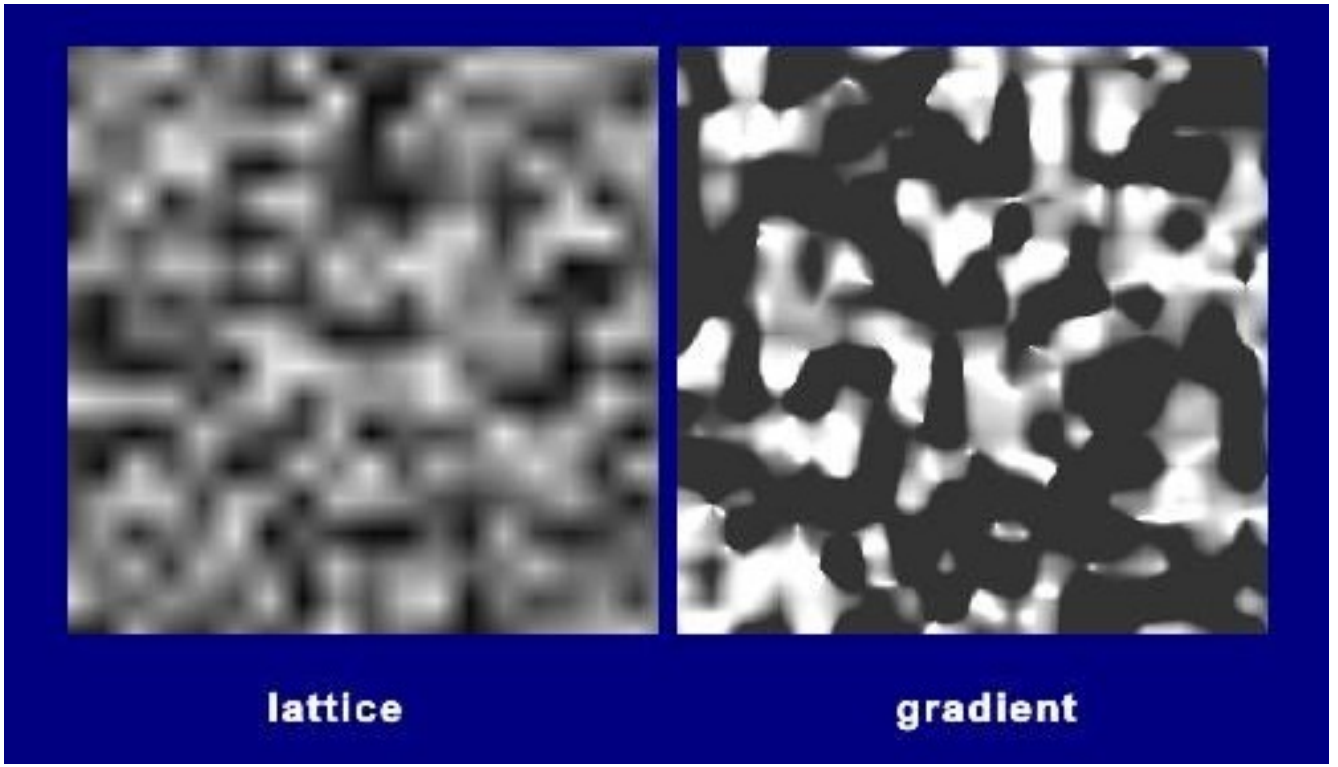
- Lattice replicated to fill entire space

## Fixed fundamental frequency of ~1 Hz over lattice

## Smooth interpolation of interim values

# Gradient noise better than value noise

- less regularity artifacts
- more high frequencies in noise spectrum
- even tri-linear interpolation produces good results



## Noise function

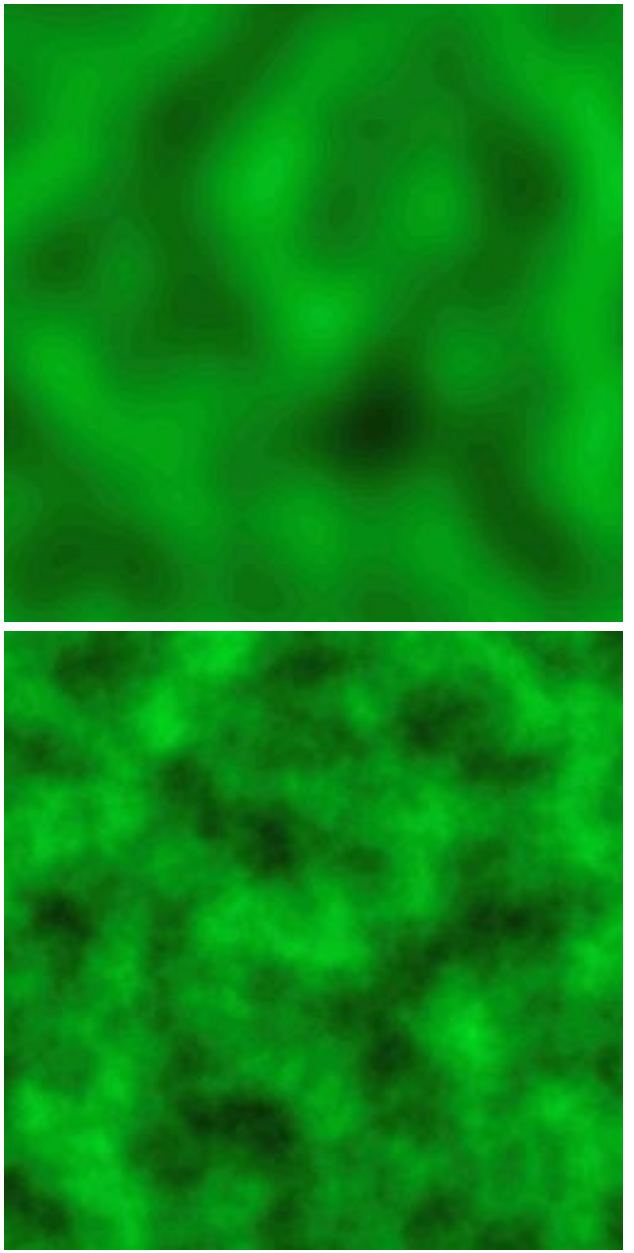
- "White" frequency spectrum

## Natural textures

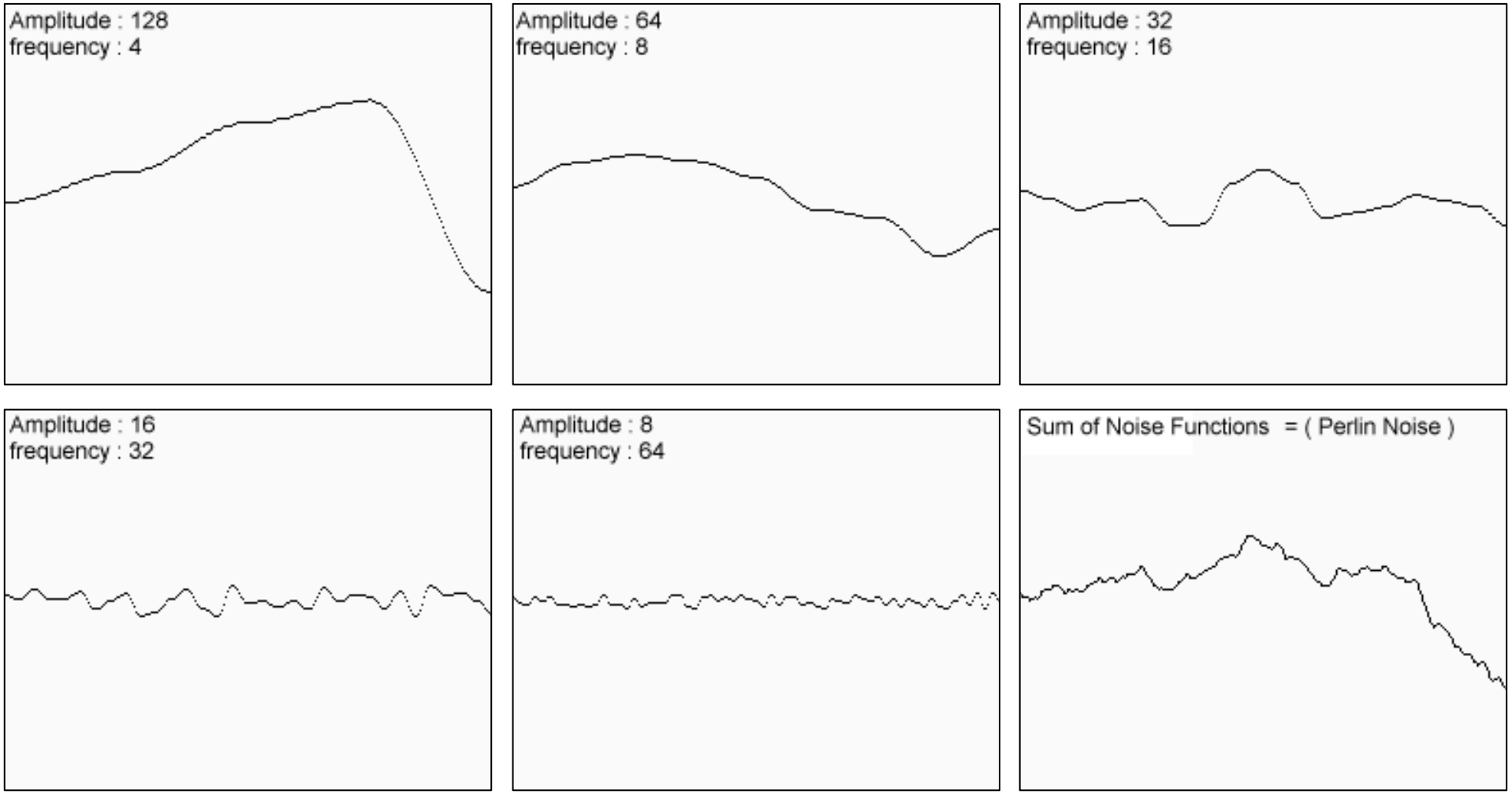
- Decreasing power spectrum towards high frequencies

## Turbulence from noise

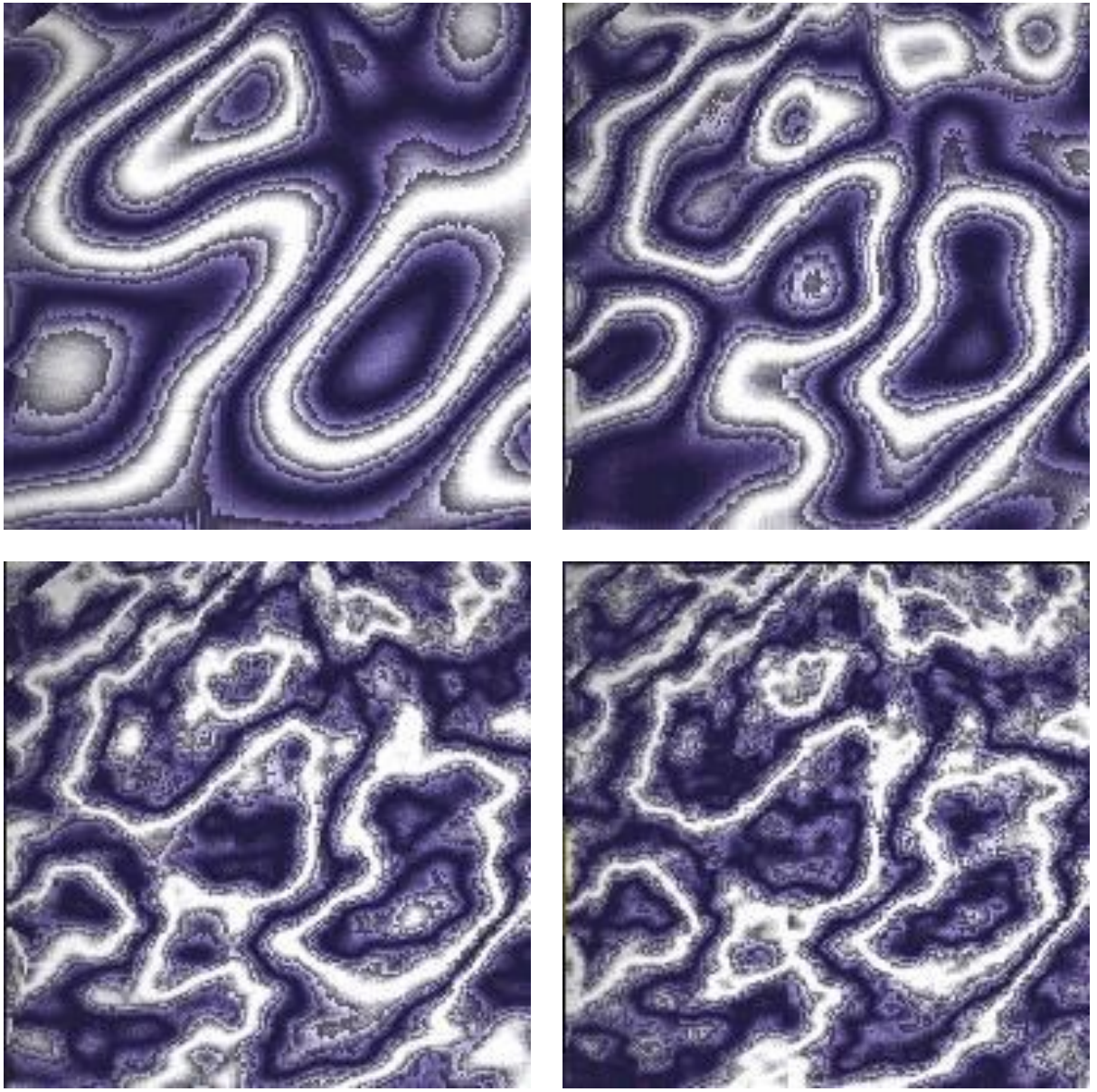
- $Turbulence(x) = \sum_{i=0}^{k} \left| \frac{noise(2^i x)}{2^i} \right|$
- Summation truncation
  - $\frac{1}{2^{k+1}} <$ size of one pixel (band limit)
- 1. Term: $noise(x)$
- 2. Term: $\frac{noise(2x)}{2}$
- …
- Power spectrum: $\frac{1}{f}$
- (Brownian motion: $\frac{1}{f^2}$)



# Synthesis of Turbulence (1D)

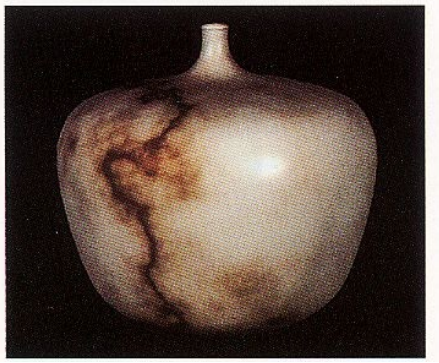


# Synthesis of Turbulence (2D)



## Overall structure: alternating layers of white and colored marble

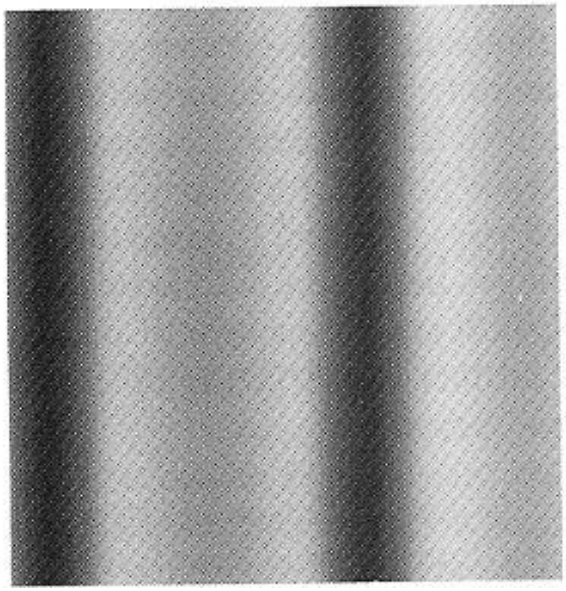
- $f_{marble}\,(x, y, z) := marble\_color(\sin x)$
- $marble\_color$: transfer function

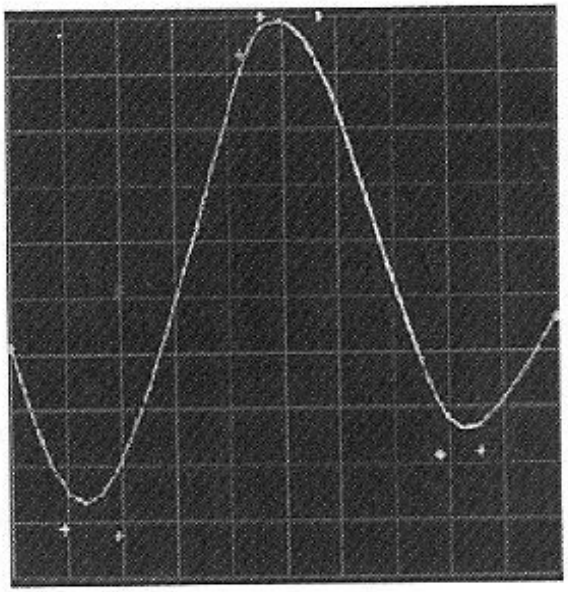


## Realistic appearance: simulated turbulence

- $f_{marble}\,(x, y, z) := marble\_color\big(\sin\big(x + turbulence(x, y, z)\big)\big)$

## Moving object: turbulence function also transformed





## Bark

- Turbulated sawtooth function
- Bump mapping

## Clouds

- White blobs
- Turbulated transparency along edge
- Transparency mapping

## Animation

- Vary procedural texture function’s parameters over time

## Procedural generation of geometry

## Complex geometry at virtually no memory cost

- Can be difficult to ray trace !!

## Coarse triangle mesh approximation

## 1:4 triangle subdivision

- Vertex insertion at edge-midpoints

## New vertex perturbation

- Displacement along normal
- Random amplitude
- Perturbation scale depends on subdivision level
  - Decreasing power spectrum
  - Parameter for model roughness

## Recursive subdivision

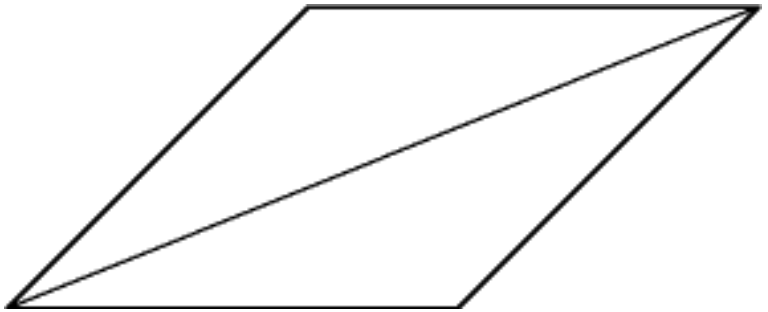
- Level of detail (LOD) determined by # subdivisions

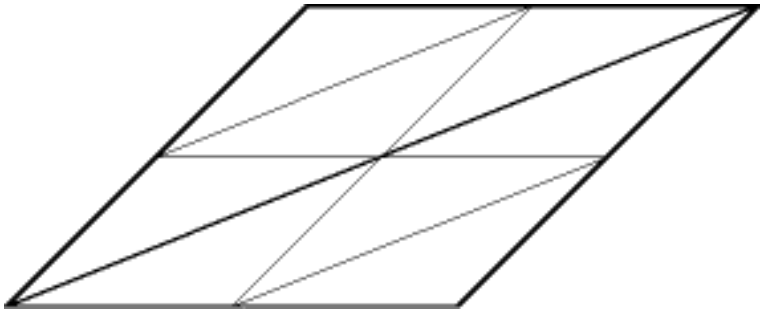
## All done inside renderer !

- LOD generated locally when/where needed (bounding box test)
- Minimal I/O cost (coarse mesh only)

## Triangle subdivision

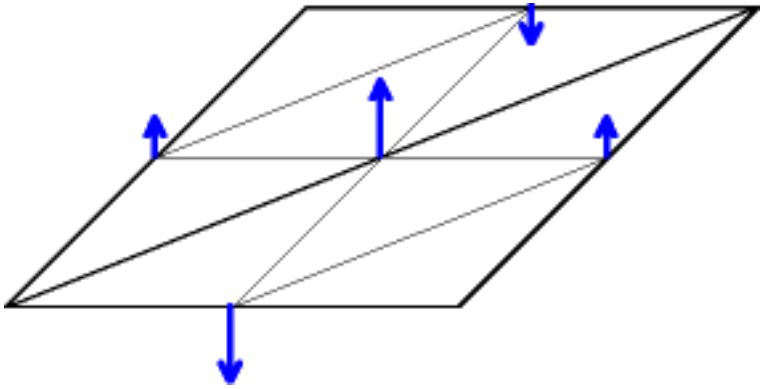
- Insert new vertices at edge midpoints
- 1:4 triangle subdivision

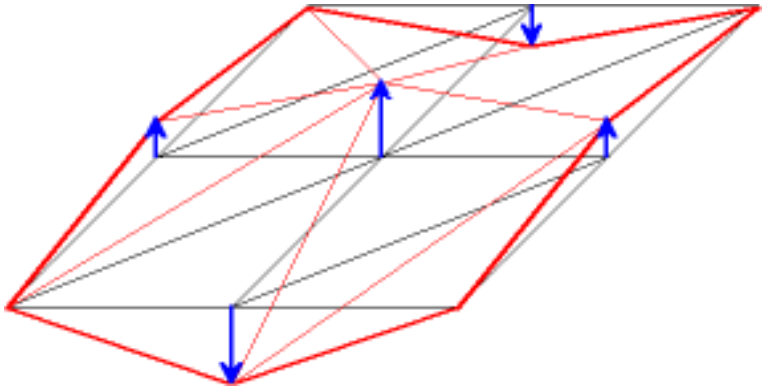




## Vertex displacement

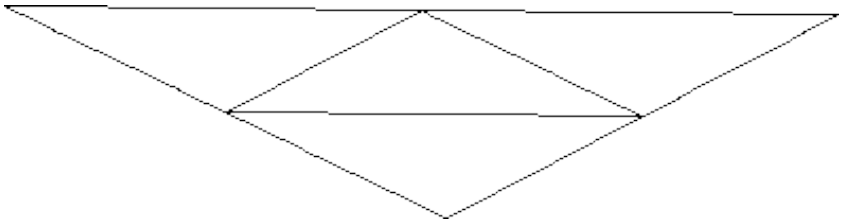
- Along original triangle normal

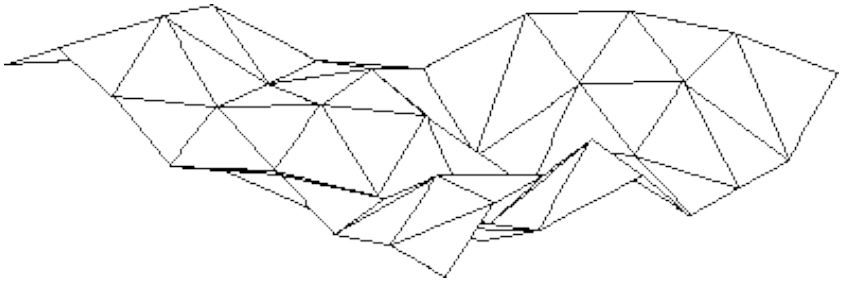


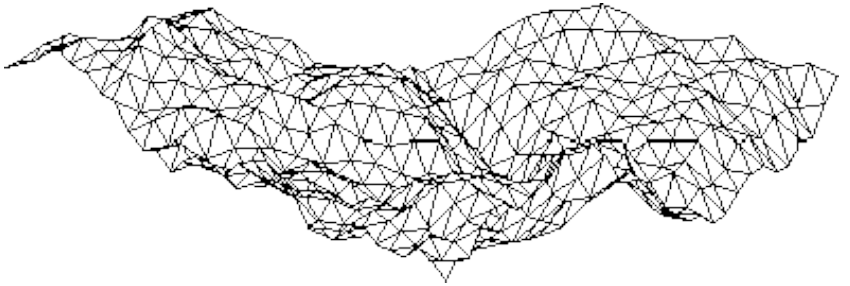


# Fractal Landscape Generation

- Base mesh
- Repeated subdivision &  vertex displacement
- Shading
- + Water surface
- + Fog
- + …







## Fractal terrain generated on-the-fly

## Problem: where is ray-surface interaction?

- Triangle mesh not a-priori known

## Solution: bounding boxes

- Maximum possible bounding box around each triangle
- Decreasing displacement amplitude: finite bounding box

## Algorithm

- Intersect ray with bounding box
- Subdivide corresponding triangle
- Compute bounding boxes of 4 new triangles
- Test against 4 new bounding boxes
- Iterate until termination criterion fulfilled (LOD / pixel size)