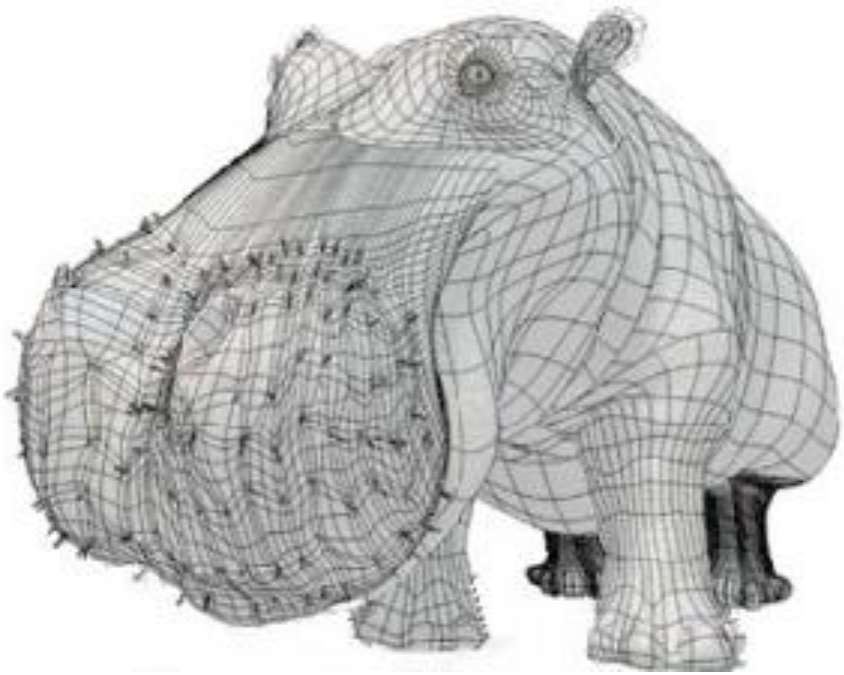
Texture Mapping

Hand with Reflecting Sphere Ferretti 2017

Texture mapping

Towards visual realism

modelling



modelling + shading

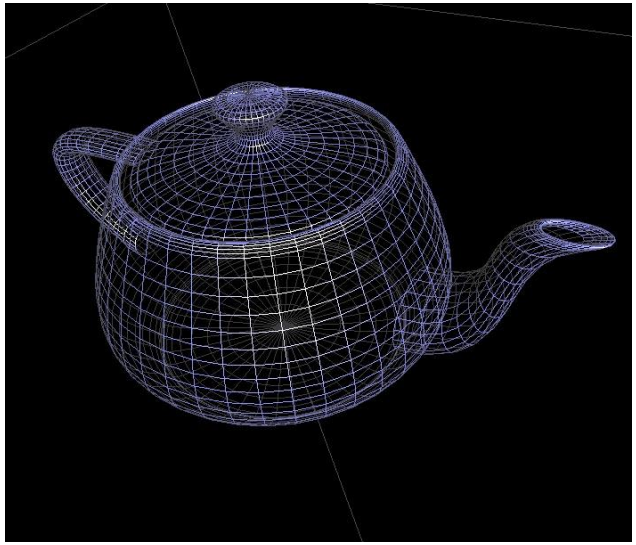
http://www.3drender.com/jbirn/productions.html

Modelling + shading + texturing

How can we get this?

Provides realistic appearance to models without having to add additional geometric fine detail

Texturing: apply images to geometric objects

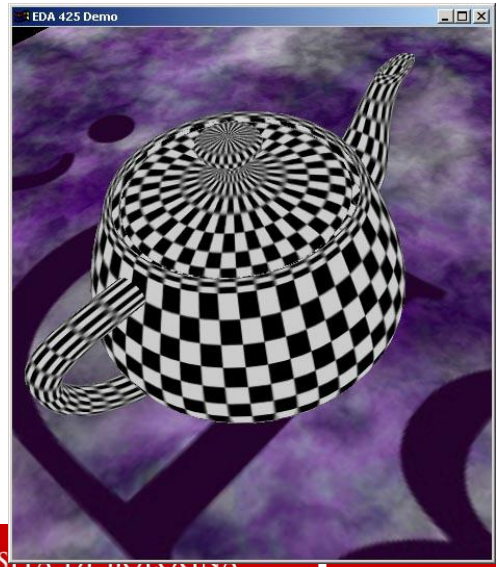


Objective: improve realism, reduce geometric complexity...

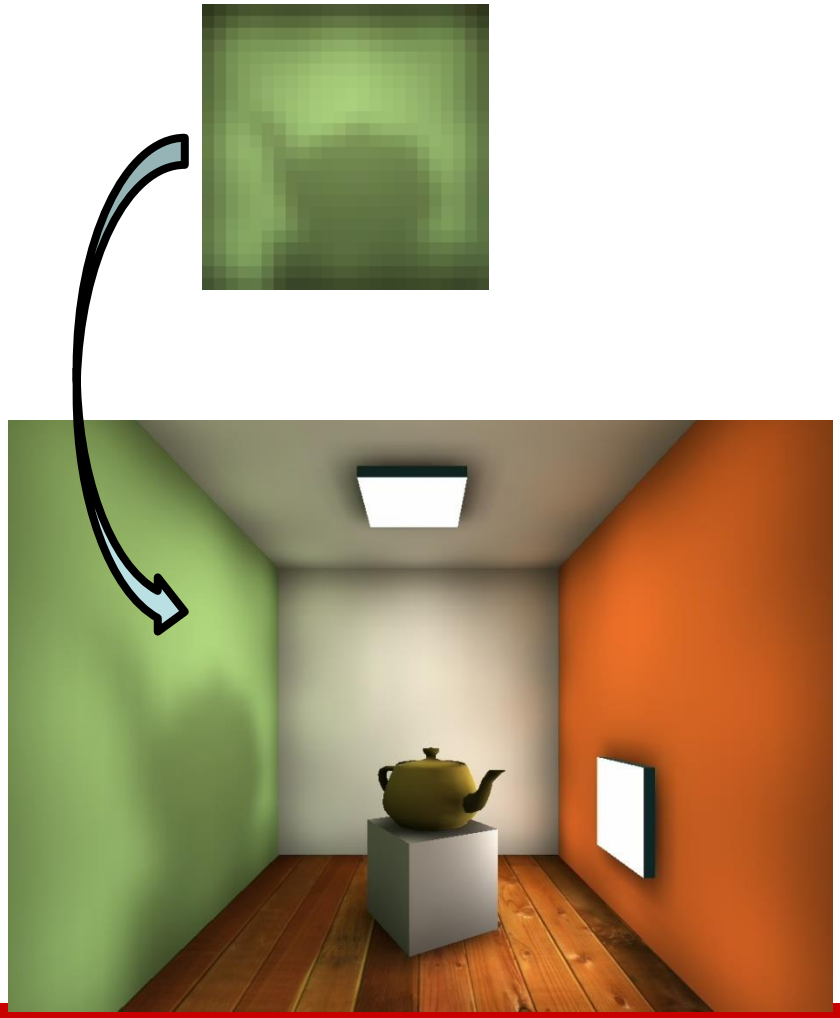
...this the easiest way to get it!

+

=



light mapping pre-computation with texture



Precompute the shading in scene:

- store the 2D light map, during shading, read the light map.
- use global illumination method view independent (i.e. radiosity) to precompute the light maps.
- a light map stores a set of 2D maps of the reflected light from the surfaces.

Images courtesy of Kasper Høy Nielsen.

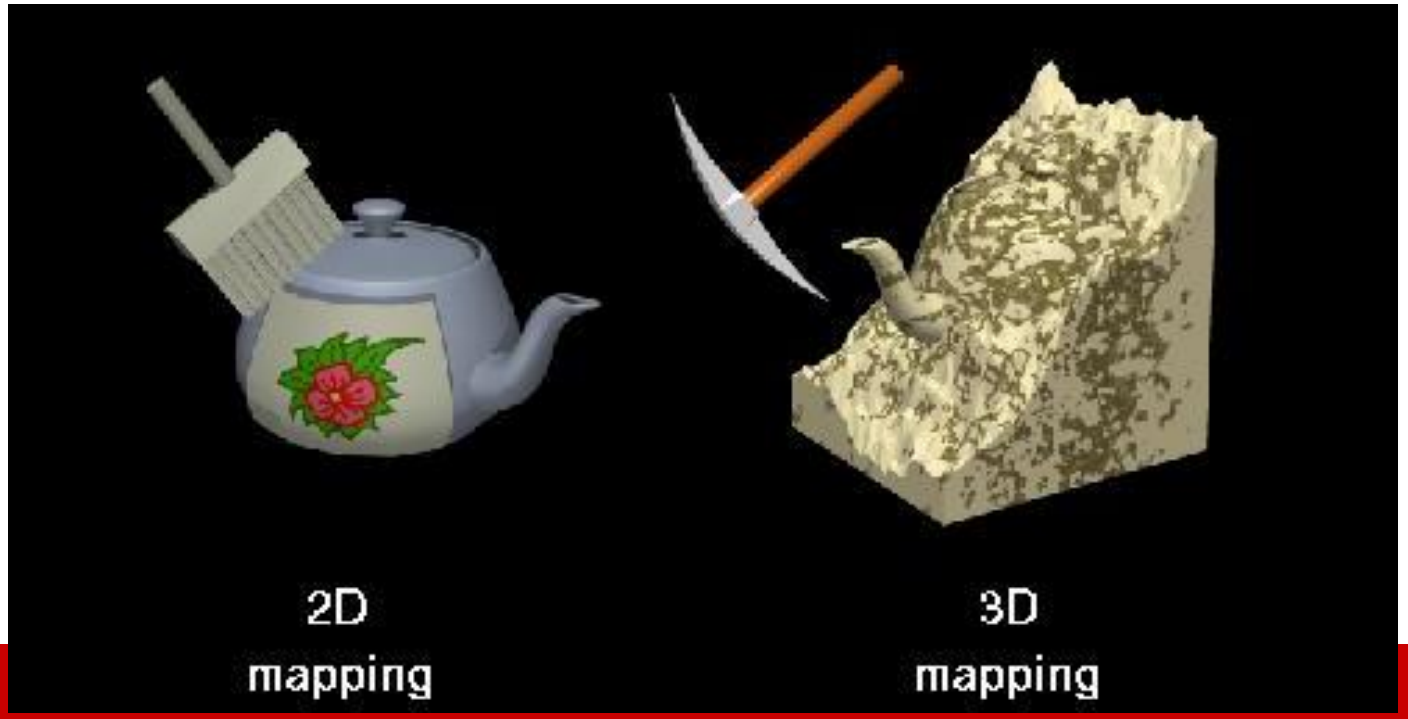
Examples in video games..

A few polygonal meshes (i.e., low geometric complexity)

- Purely local lighting computations
- Details by pre-computed texturing
  - Geometric Details
  - Smog, fire, special effects
  - Lights, shadows, reflective effects
- Bump mapping in hardware

Texturing

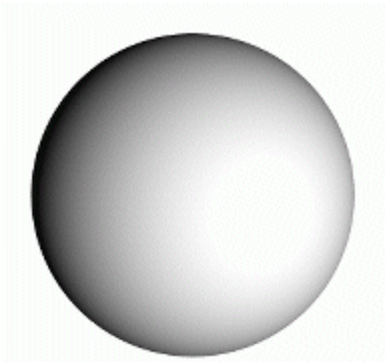
- 2D Image Texturing: maps image onto surface
- 3D Solid texturing: objects appear sculpted out of solid substance
  - 2D/3D images - Procedural texturing



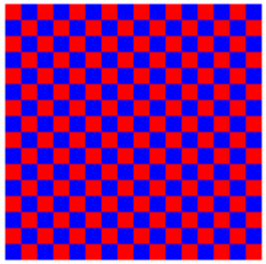
Adding TEXTURE MAPPING to illumination

Texture mapping can be used to alter some or all of the constants in the illumination equation. We can simply use the texture as the final color for the pixel, or we can just use it as diffuse color, or we can use the texture to alter the normal, or... the possibilities are endless!

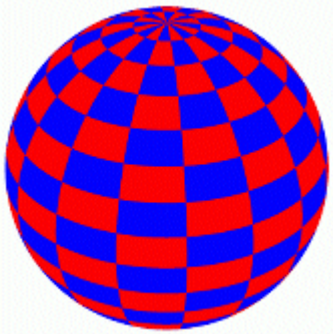
$$I_\lambda = I_a k_a + \sum_{j=1}^{lights} \frac{I_j}{s^2}[k_d(n \cdot l_j) + k_s(v \cdot r)^n]$$



Constant diffuse color



Diffuse Texture Color



Texture used as Label

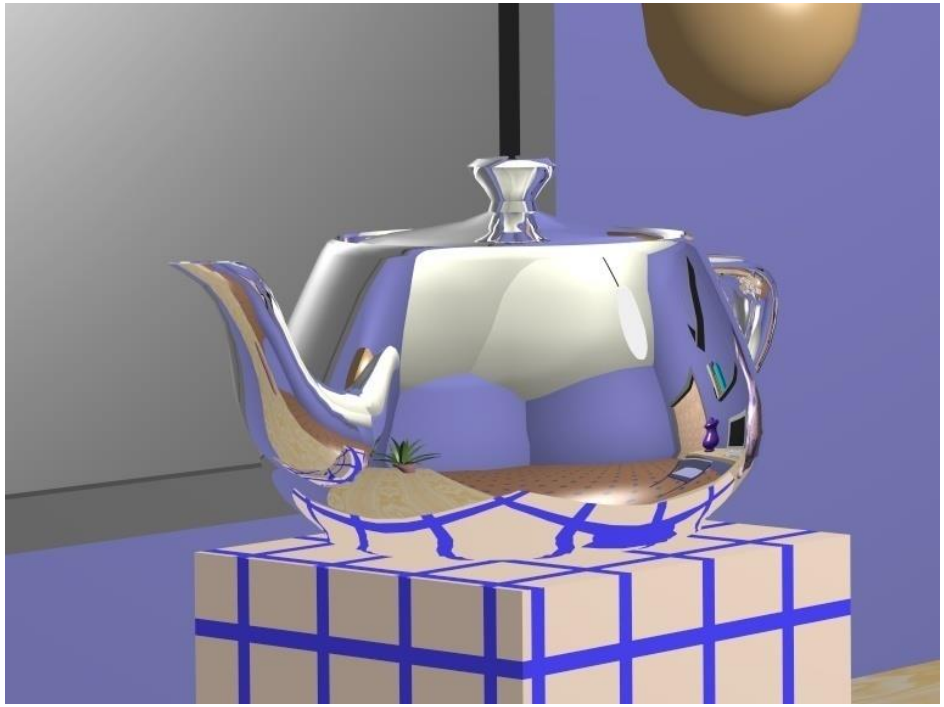
Texture used as Diffuse Color

Object properties modified by texture mapping:

- **Color (texture mapping)** modify the final color (LIT in Phong's Model/UNLIT no lighting calculation) with the texture map color

Object properties modified by texture mapping:

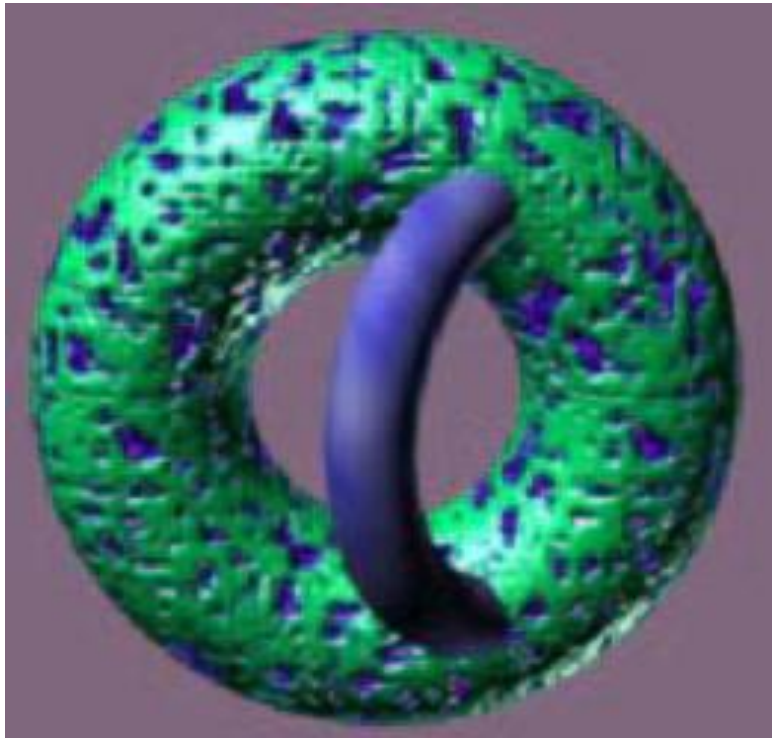
- **Reflected color (environment mapping):**
  - Uses a picture of the environment for texture maps
  - Allows simulation of highly reflective surfaces



Object properties modified by texture mapping:

- **Surface Normal (bump mapping):**

Intensity value of a source image is used to modify the normal direction of the object surface. Does not modify surface geometry, but the visual result fools us into thinking the surface is not smooth.

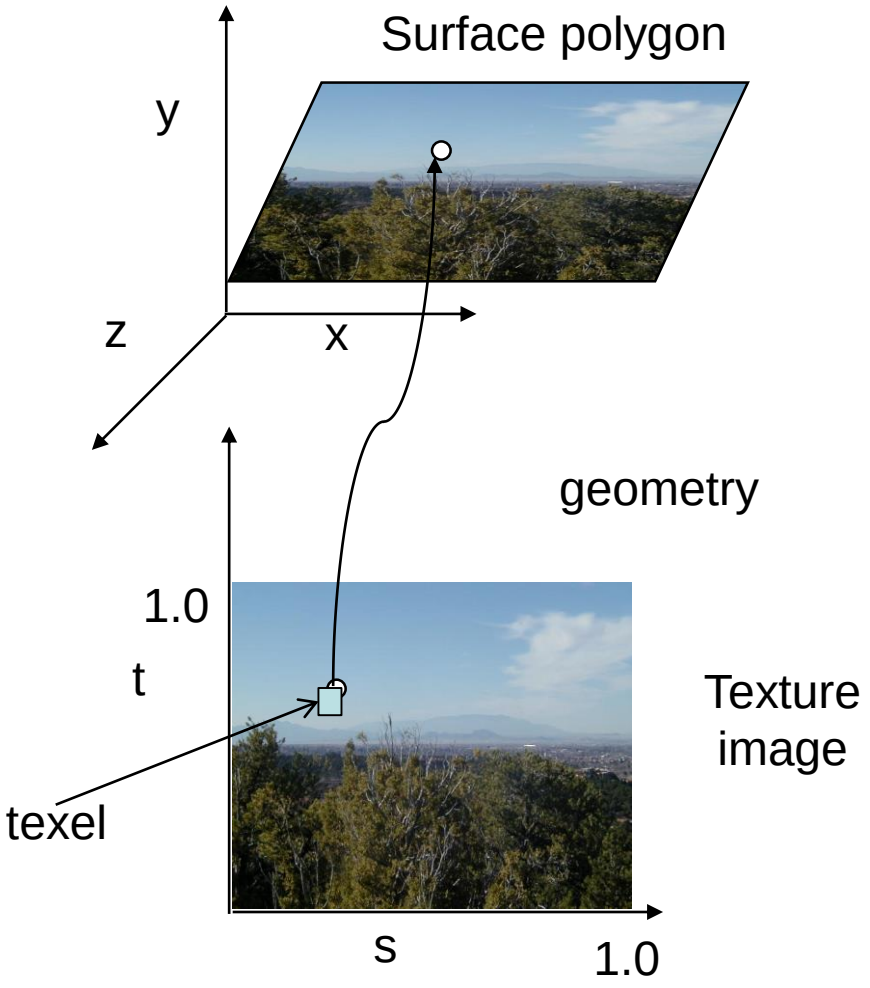


Object properties modified by texture mapping:

- **Transparency** (Opacity maps) modify the opacity of a transparent object

Ken Perlin's vase with procedurally varying index of refraction, 1985

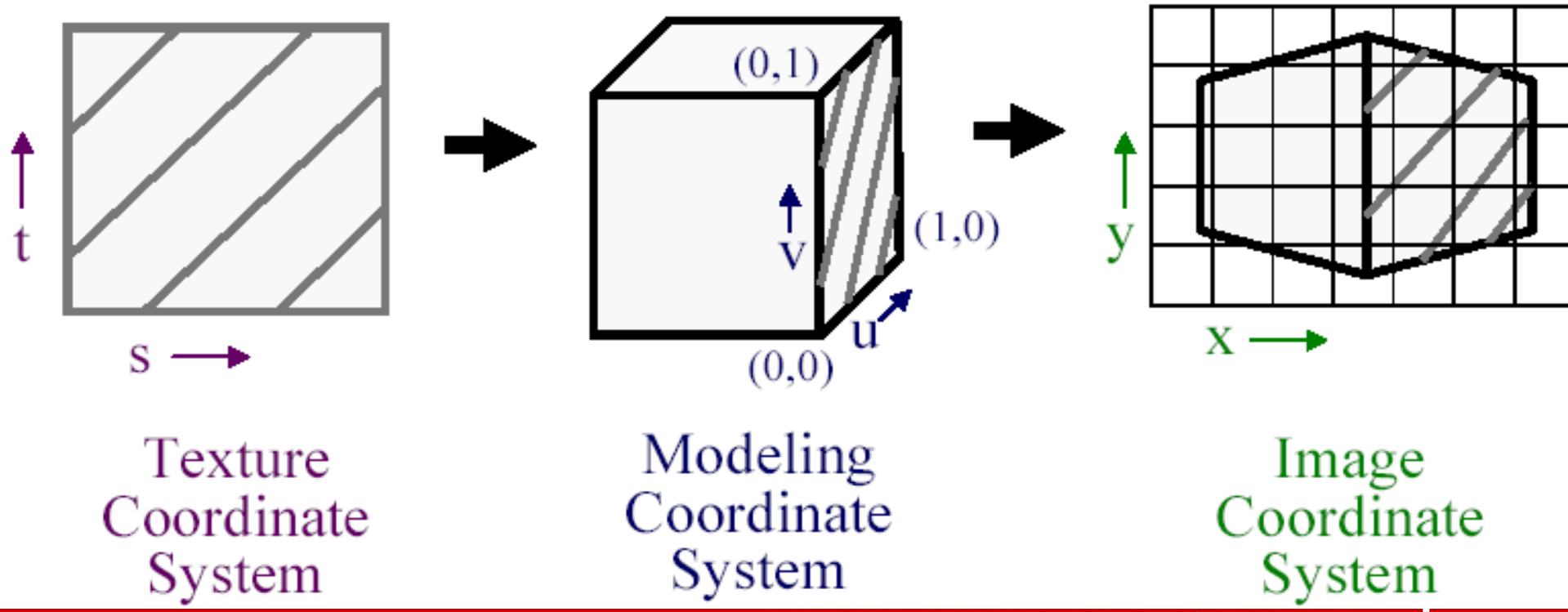
Image Texture Mapping



- 2D digital image (**Texture**) maps onto surface (polygon)
- Pixel in a texture (**Texel**) have coordinates (s,t) [0,1]x[0,1]

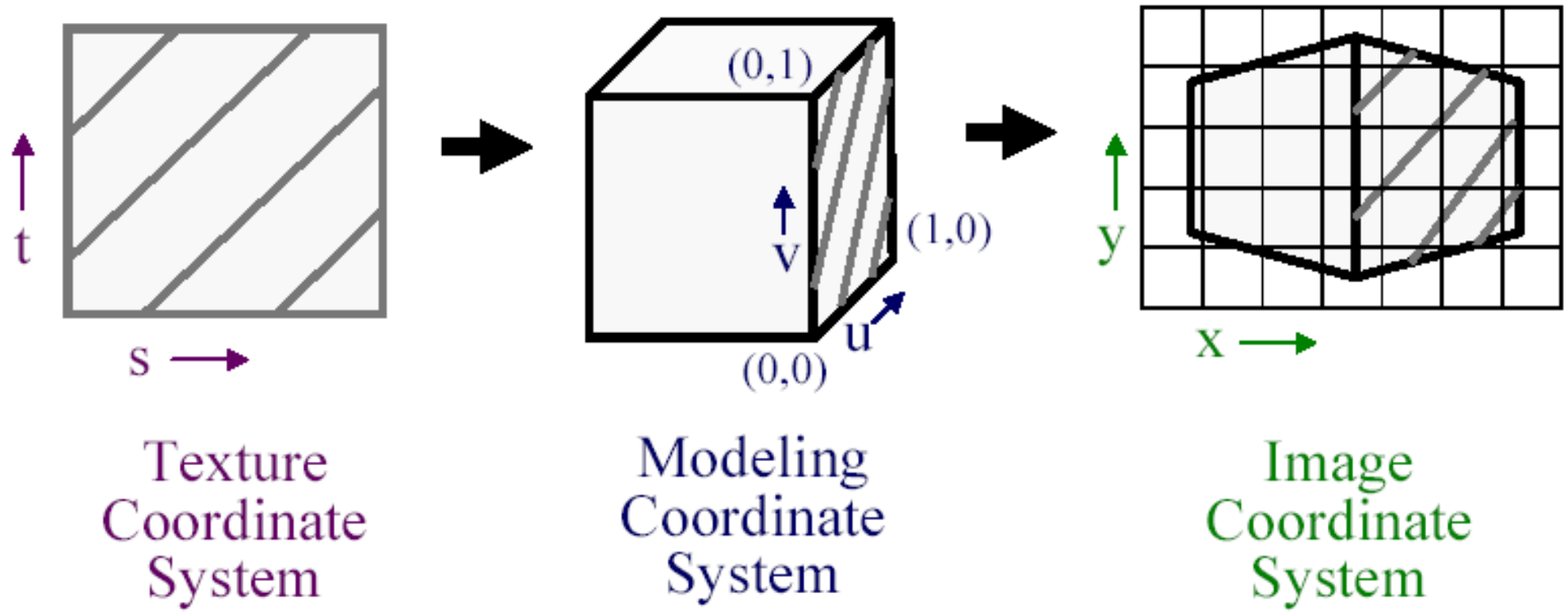
Texture mapping

- Screen space screen coords. system (x,y)
- Object space world coords. system (u,v,w)
- Texture space texture coords system (s,t)



Texture mapping

- Parameterization phase: from Texture Space to Object Space
- Projection phase: from Object space to Screen space



Parameterization

- How to apply the texture to the object
- Define a function M that associates each surface point (x,y,z) with a 2D point in the texture map (pair of coords (s,t))

$$M(x,y,z)=(s,t)$$

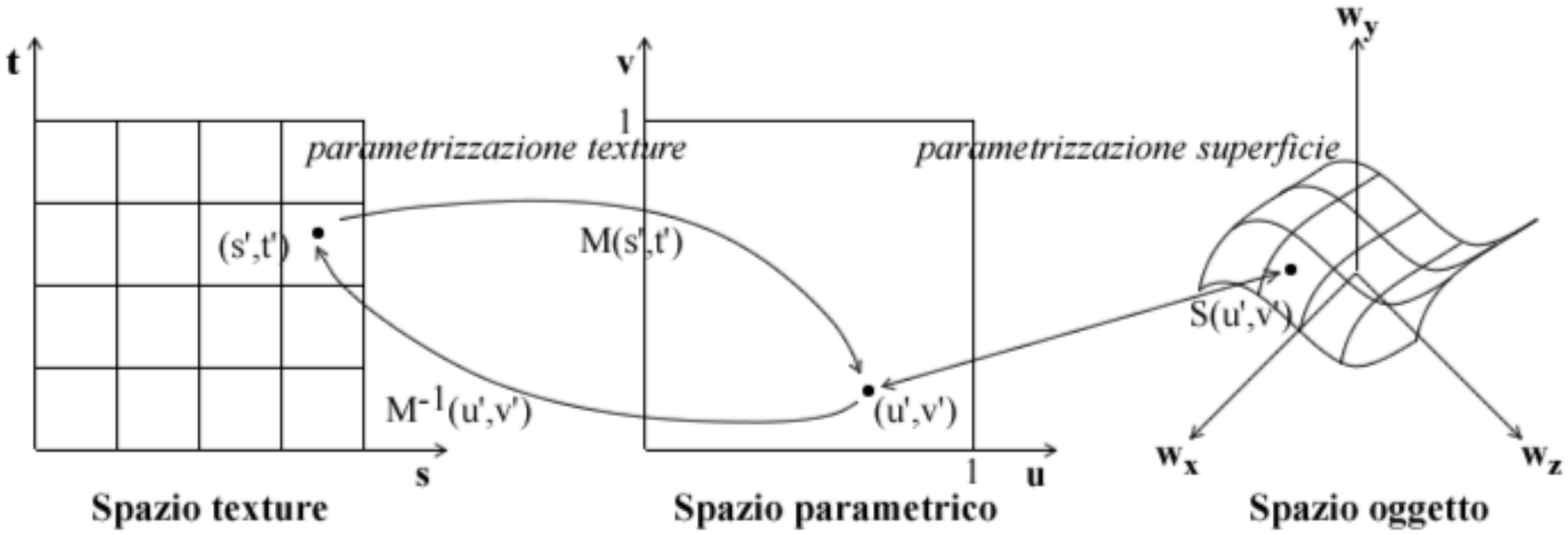
  For each point rendered, look up color in texture map
- Parameterization depends on the object representation:
  - Parametric surface
  - mesh

Parameterization: parametric surface

- If the object is represented by parametric surfaces (i.e.spline), the parameterization is naturally provided by the domain D=UxV

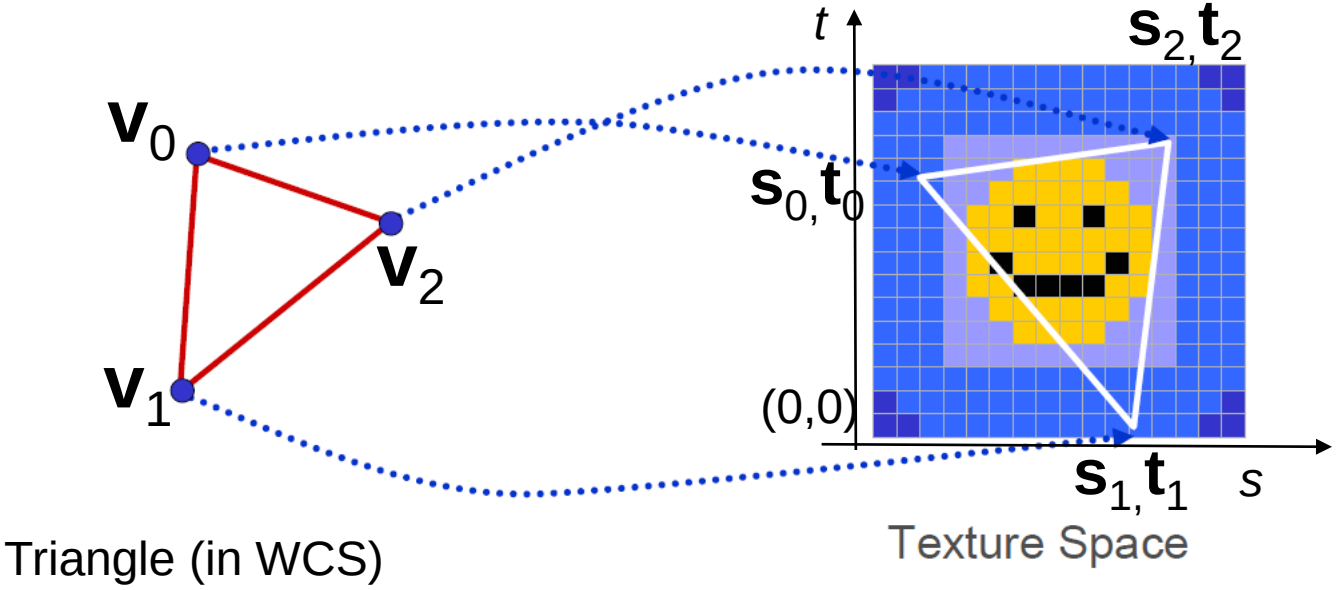
  S(u,v)=[x(u,v),y(u,v),z(u,v)], (u,v) in D

- Texture space and parametric space are related by an affine transformation M
- We need inverse mapping M^{-1}



Parameterization of a triangle

Per-vertex texture coords. manually, provided by user : assign a texture coords (s,t) to each vertex v



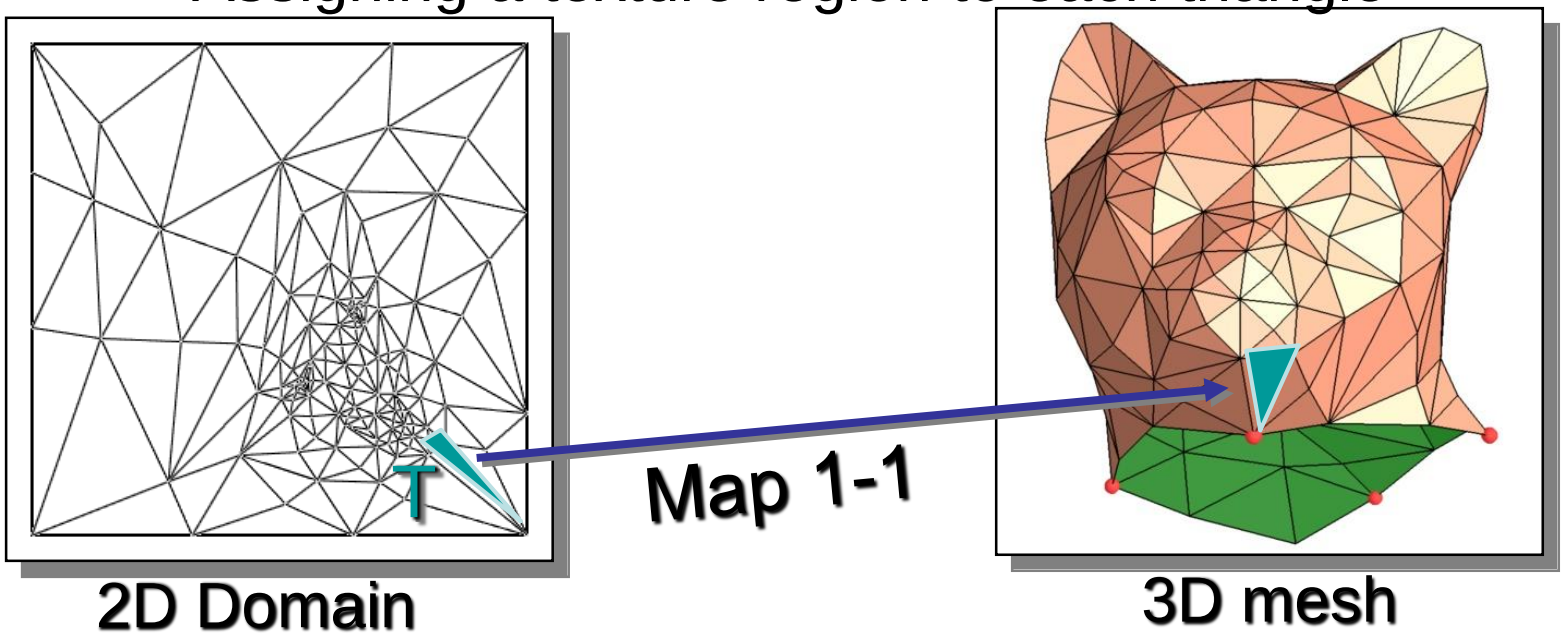
Triangle (in WCS)

Points inside the polygon: $(\alpha s_0+\beta s_1+\gamma s_2,\ \alpha t_0+\beta t_1+\gamma t_2)$

Then we interpolate using barycentric coords.

Parameterization of a Mesh

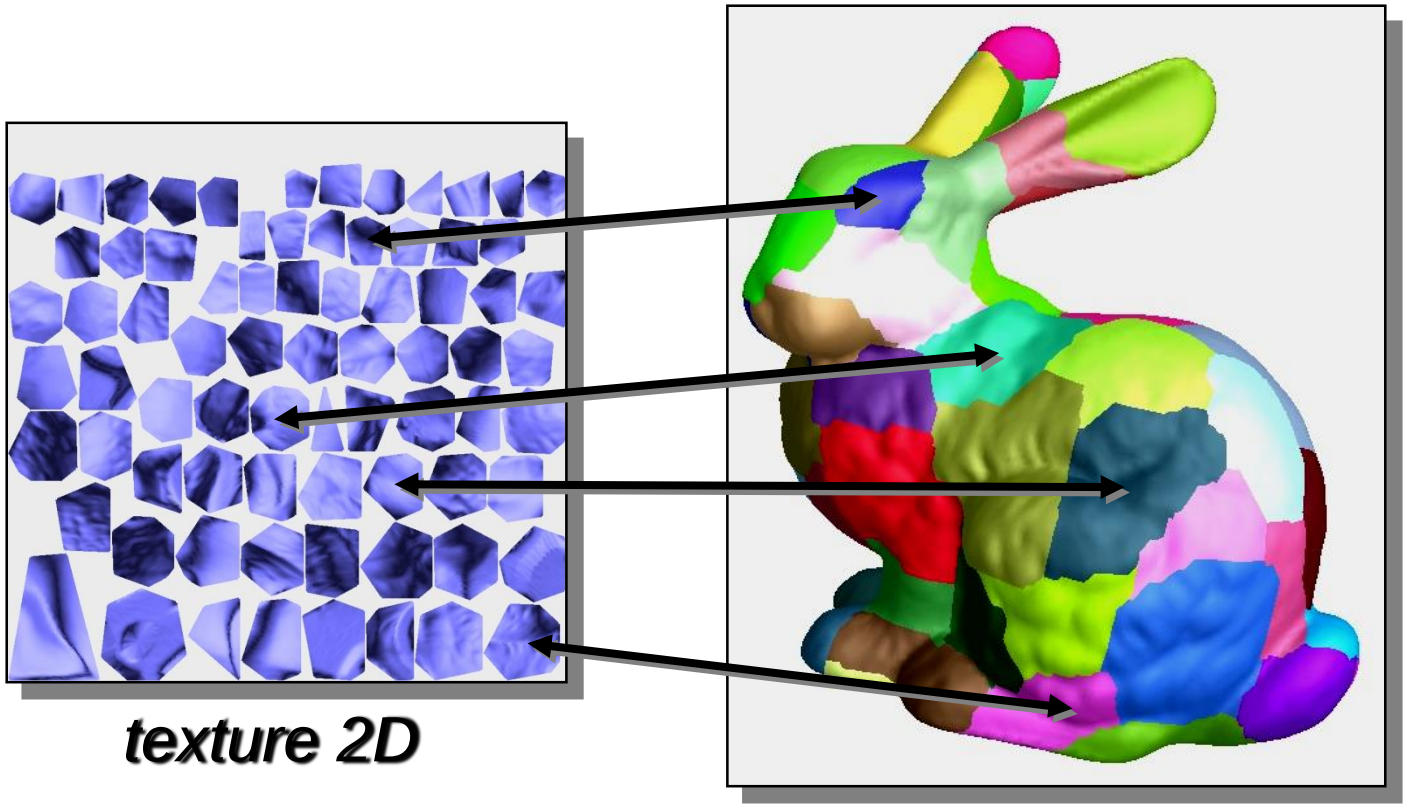
Assigning a texture region to each triangle



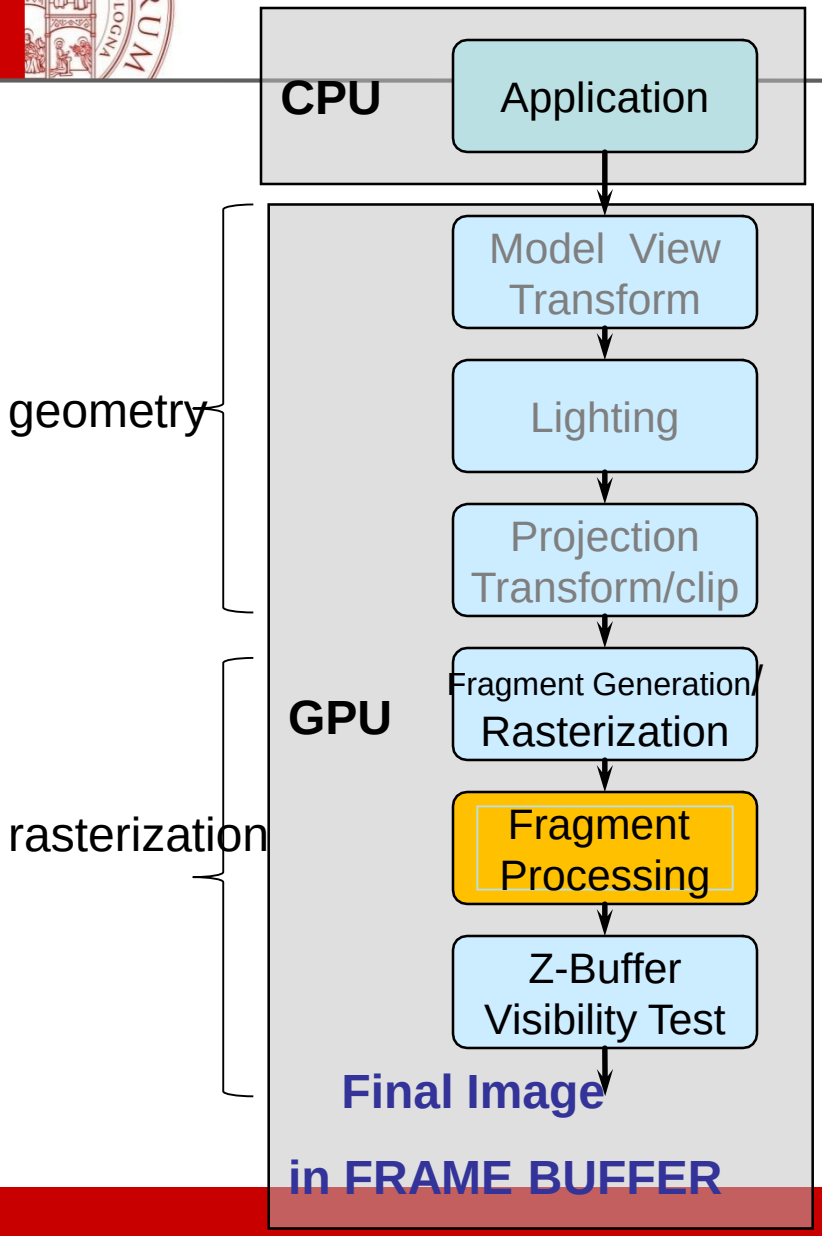
An ideal parametrization would be **isometric**.
That is, it would preserve lengths, and therefore angles too.
A **conformal** parametrization is one that preserves angles. A surface triangle can be mapped to a small triangle in the texture domain.
This leads to undersampling.

Parameterization of a mesh with genus (genere) >0

- Subdivide the mesh in **chart**
- Parameterize each chart

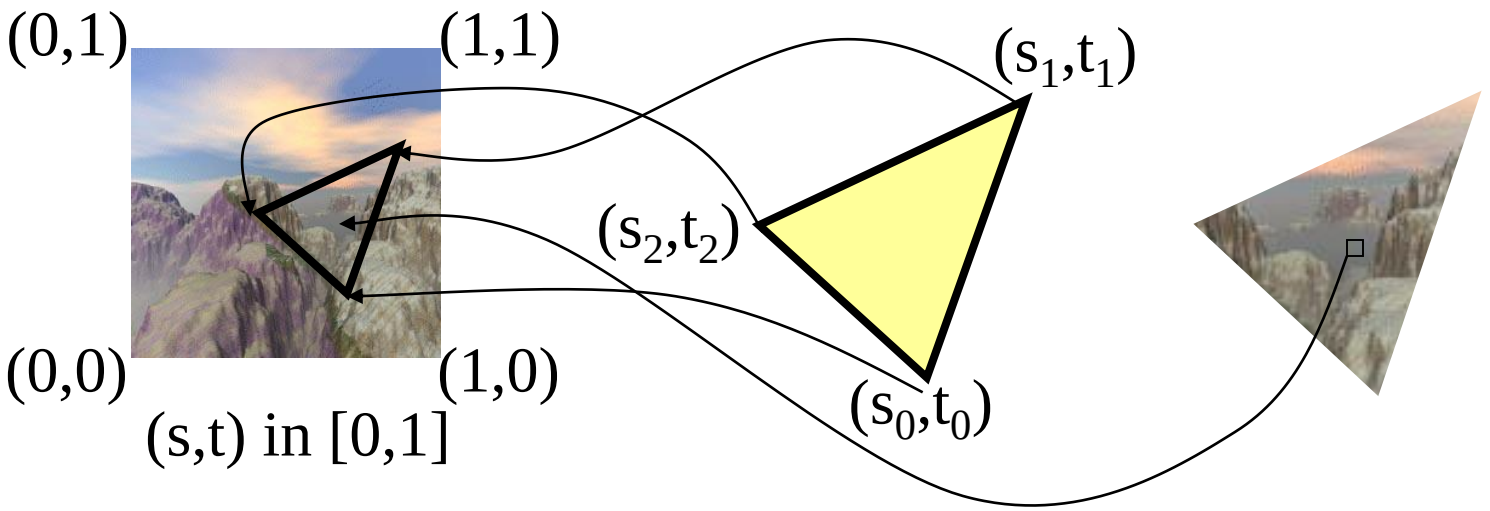


Projection phase



- The texture mapping computations take place at the scan conversion stage of the graphics pipeline
  - Very efficient because few polygons make it past the clipper
- During scan conversion, as we are looping through the pixels of a triangle, we must **interpolate** the *(t,s)* texture coordinates in a similar way to how we interpolate the *rgb* color and *z* depth values

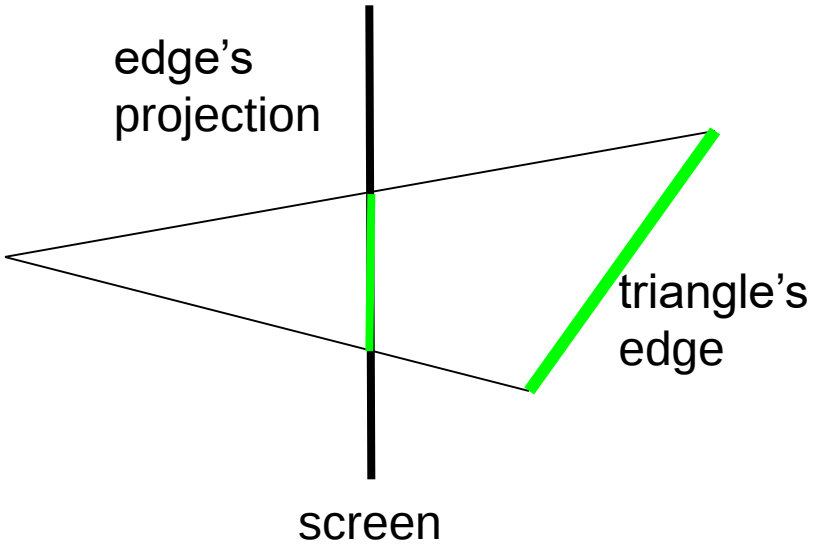
Interpolation of the Texture Coords

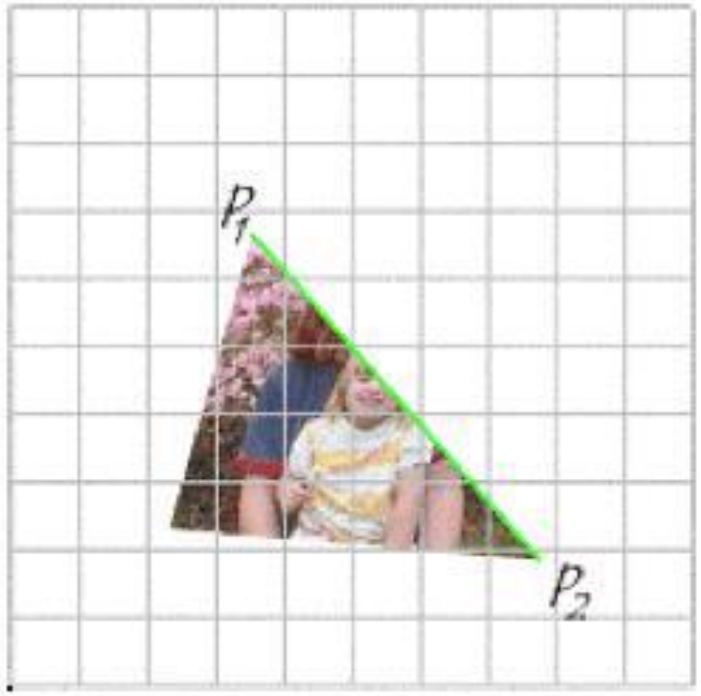


- Each vertex (x,y,z) is associated with a texel coords (s,t) (parameterization)
- A backward mapping exists only for polygon vertices not for pixel, the texture coords for pixel inside the triangle are computed by barycentric coords. interpolation (scan conversion)
- The texel coords are used to read texture map and assign pixel color in (R,G,B) form (color image)
- Modify the Phong's Illumination model

Texture distorsion

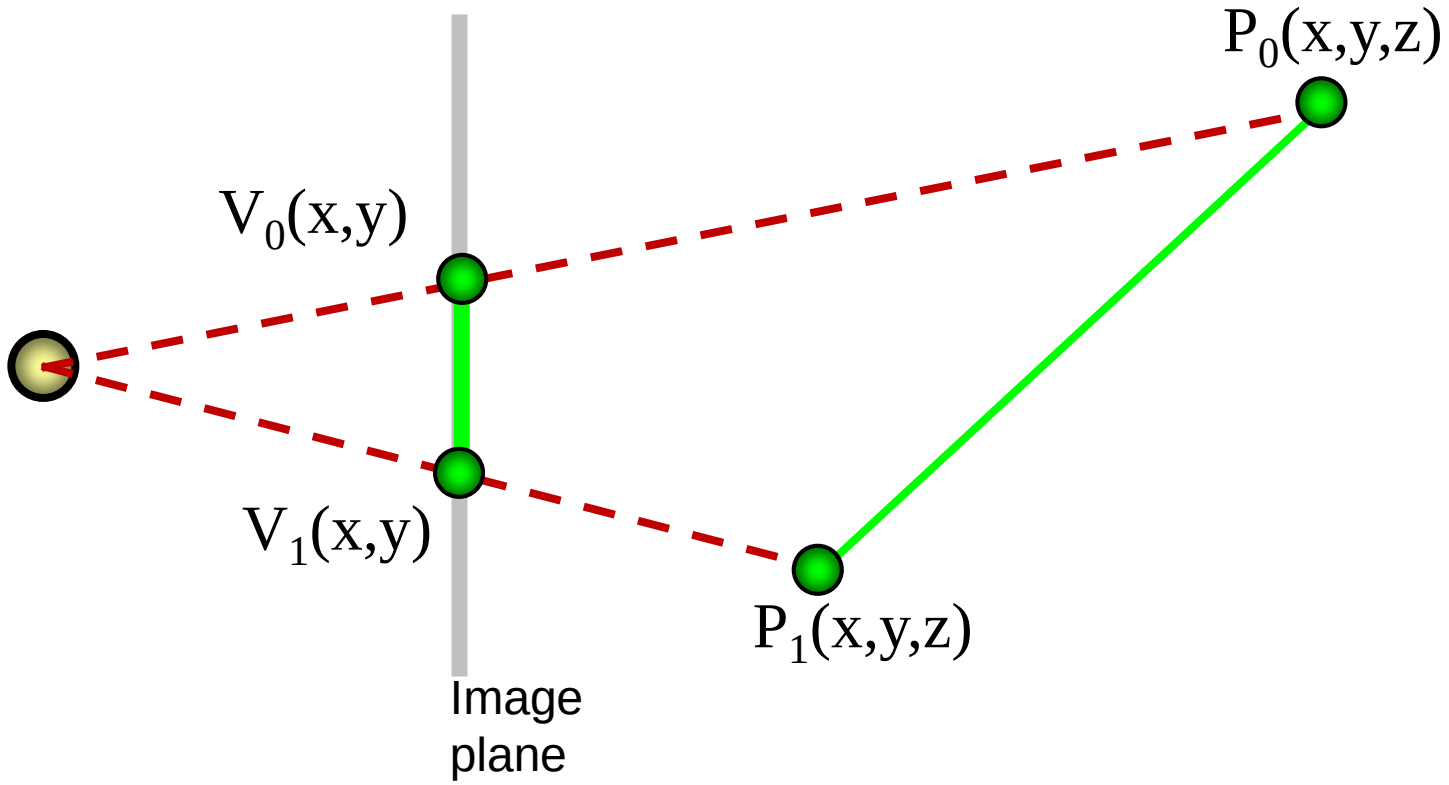
- **Consider a single edge of a triangle**
- **Project this from world space to screen space**





Interpolation in Screen space vs. world space

- A straight line of *regularly* spaced points in 3D space maps to a straight line of *irregularly* spaced points in device space (if we are using a perspective transformation)

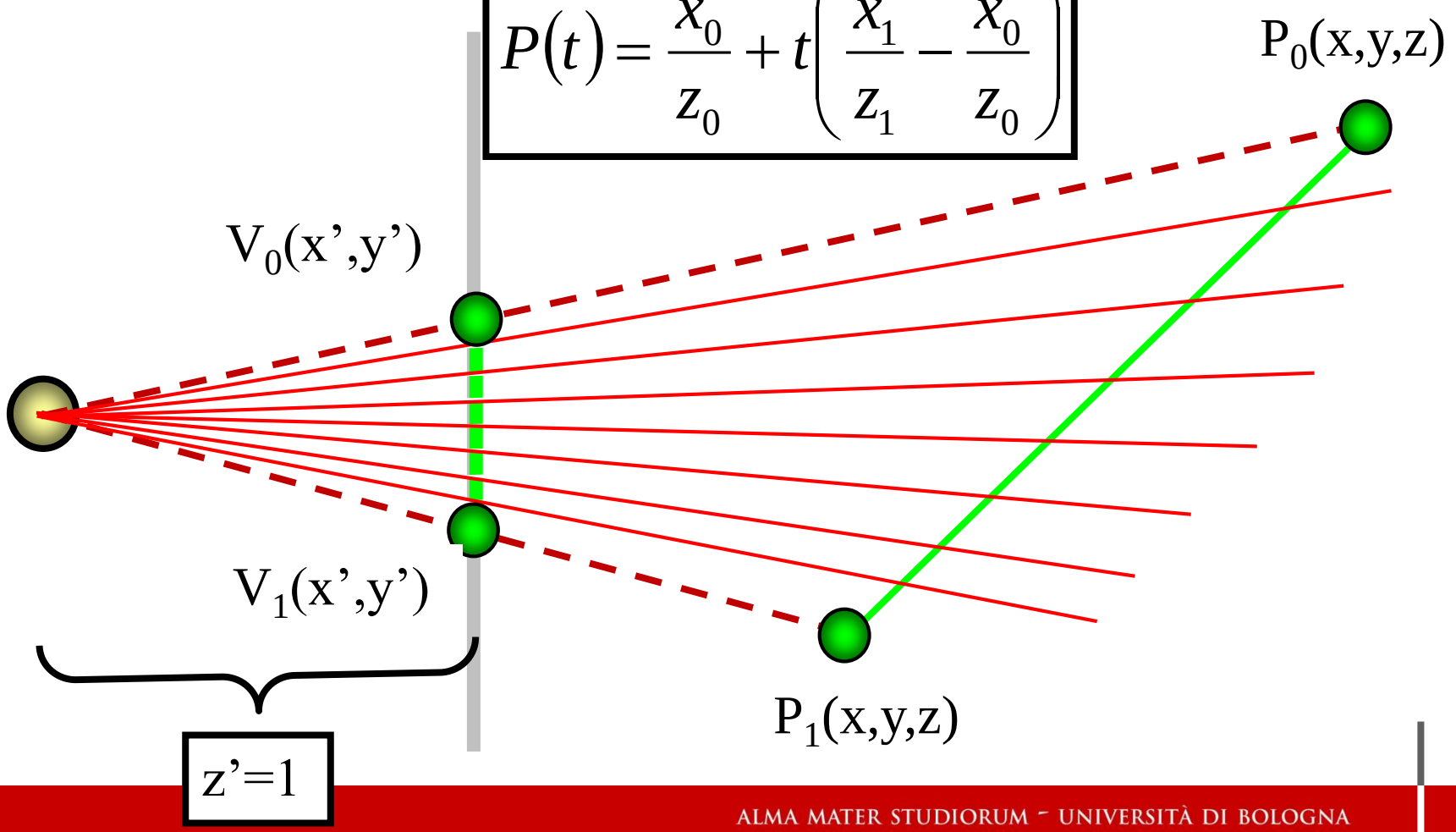


Same as in Gouraud shading: distortion

Interpolation in Screen Space

$$P(t) = V_0 + t(V_1 - V_0)$$

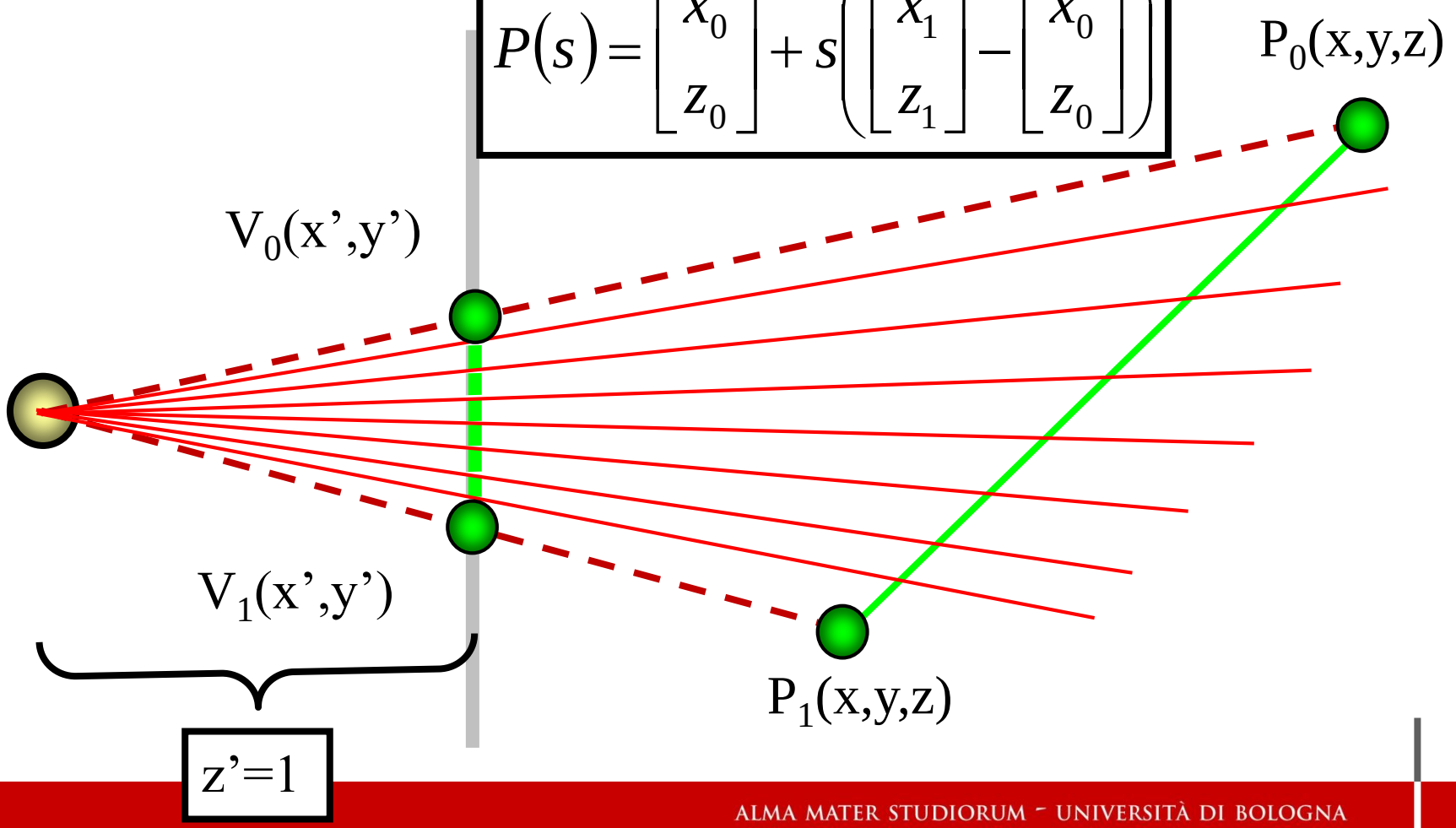
$$P(t) = \frac{x_0}{z_0} + t\left(\frac{x_1}{z_1} - \frac{x_0}{z_0}\right)$$



Interpolation in World Space

$$P(s) = P_0 + s(P_1 - P_0)$$

$$P(s) = \begin{bmatrix} x_0 \\ z_0 \end{bmatrix} + s\left(\begin{bmatrix} x_1 \\ z_1 \end{bmatrix} - \begin{bmatrix} x_0 \\ z_0 \end{bmatrix}\right)$$



Perspective Correction

- P(t)=Projection of P(s)

$$P(t) = V_0 + t(V_1 - V_0) \qquad P(s) = P_0 + s(P_1 - P_0)$$

$$P(t) = \frac{x_0}{z_0} + t\left(\frac{x_1}{z_1} - \frac{x_0}{z_0}\right) \qquad P(s) = \begin{bmatrix} x_0 \\ z_0 \end{bmatrix} + s\left(\begin{bmatrix} x_1 \\ z_1 \end{bmatrix} - \begin{bmatrix} x_0 \\ z_0 \end{bmatrix}\right)$$

- Solve for s in terms of t:

$$\frac{x_0}{z_0} + t\left(\frac{x_1}{z_1} - \frac{x_0}{z_0}\right) = \frac{x_0 + s(x_1 - x_0)}{z_0 + s(z_1 - z_0)}$$

$$s = \frac{tz_0}{z_1 + t(z_0 - z_1)}$$

Interpolate arbitrary parameters

Solving this last equation for s gives a mapping from bary coords in screen space t to bary coords. in word space s

$$s = \frac{tz_0}{z_1 + t(z_0 - z_1)}$$

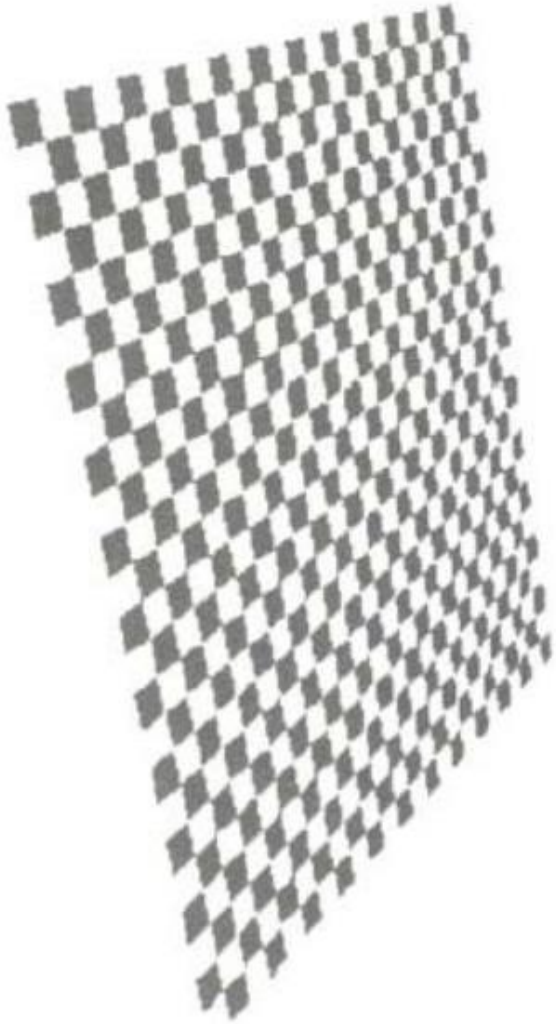
- Thus for each pixel, compute its t in screen space
- Then use the mapping from screen space to world space
- Now use the expression for s to interpolate arbitrary parameters, such as texture indices (u,v) over 3D triangle.

$$u = u_0 + s(u_1 - u_0)$$

$$v = v_0 + s(v_1 - v_0)$$

N.B. this is the formula for a segment, not for a triangle…

Linear Interpolation



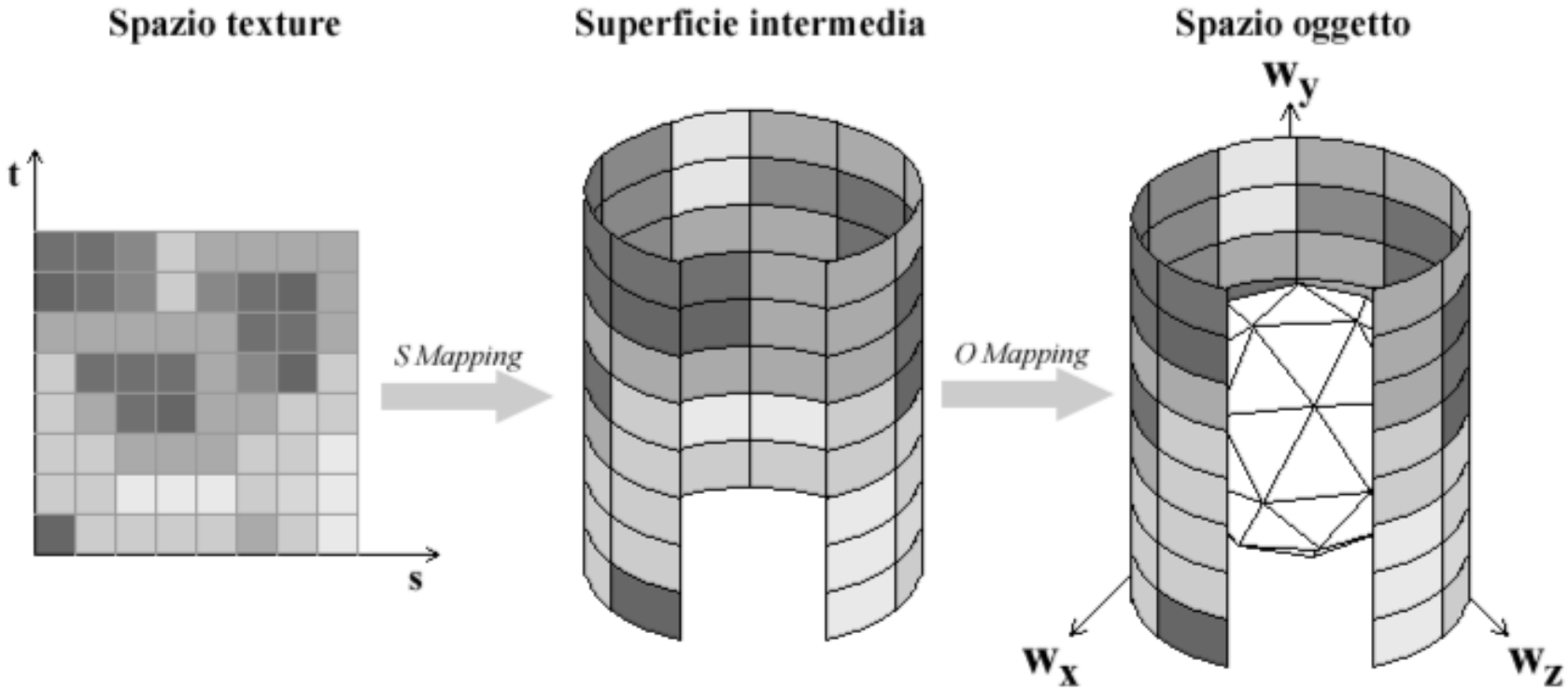
Interpolation with
Perspective correction

Pseudo code Rasterization

```
For every triangle
   Compute Projection of vertices
   Compute bbox, clip bbox to screen limits
   Setup 3 line equations
   For all pixels x,y in bbox
       If all line equations < 0      //pixel [x,y] in triangle
              Compute barycentric coordinates
              Compute perspective correction parameter s
              Compute currentZ from vertices
              If currentZ < zBuffer[x,y] //pixel is visible
                     Interpolate UV coordinates from vertices
                     look up texture color kd
                     Framebuffer[x,y] = kd
                     zBuffer[x,y] = currentZ
```

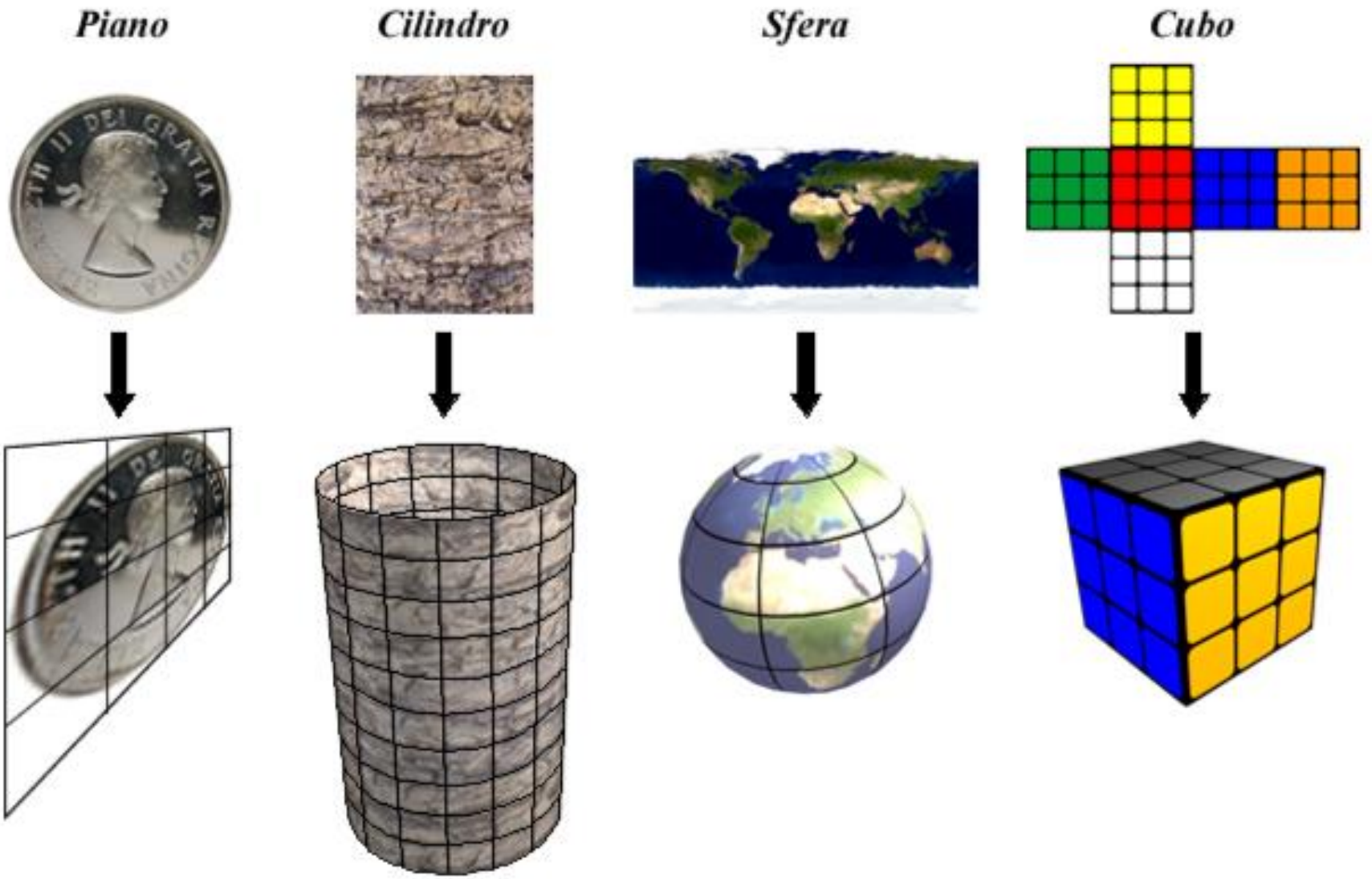
Two step mapping (Bier & Sloan '86)

– STEP 1: (S Mapping) first map the texture to a simple intermediate surface



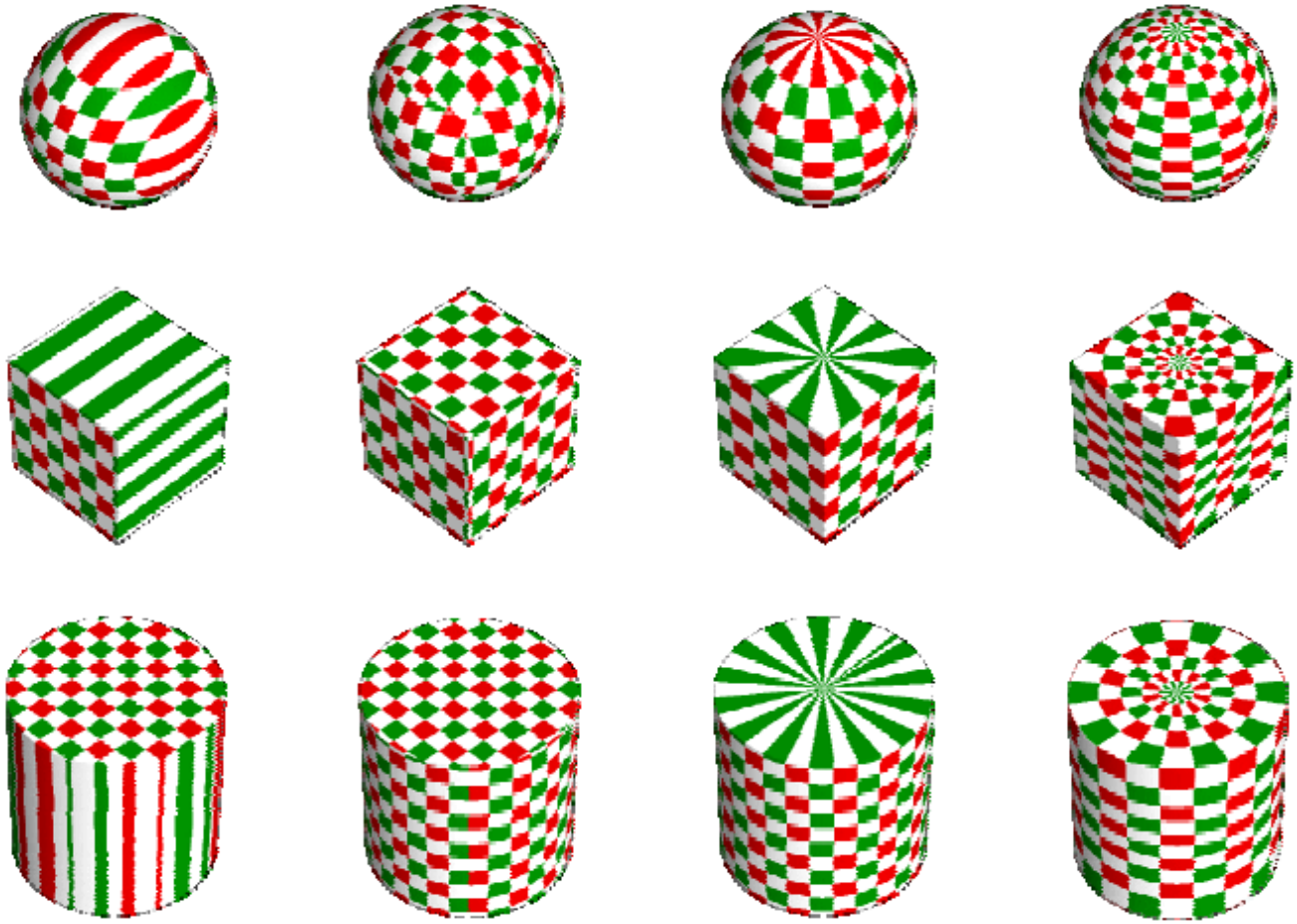
– STEP 2: (O Mapping) Map from intermediate object to actual object (projection)

Intermediate surfaces

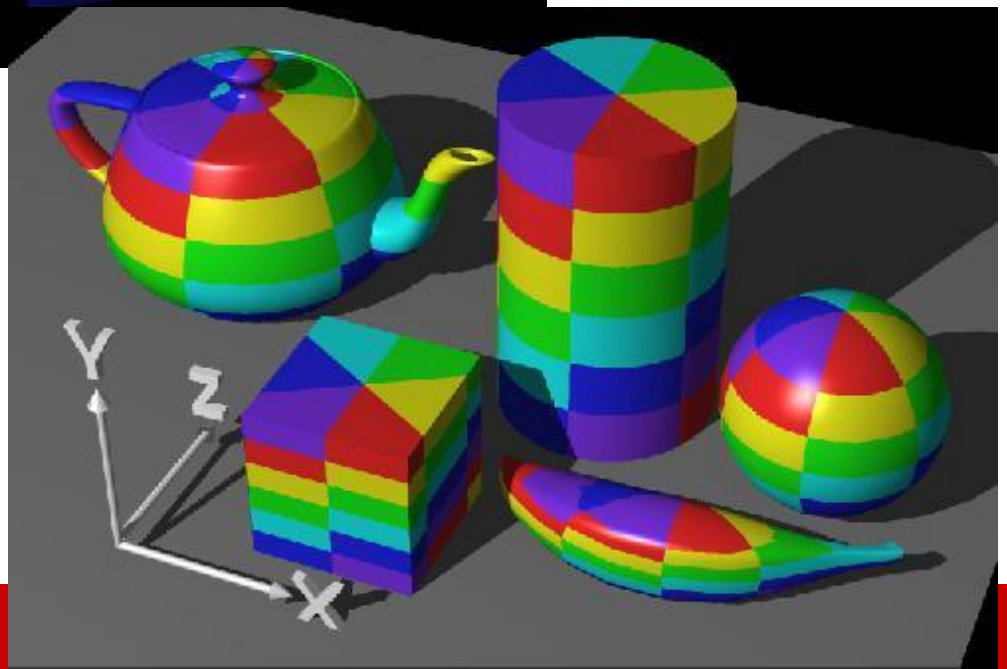


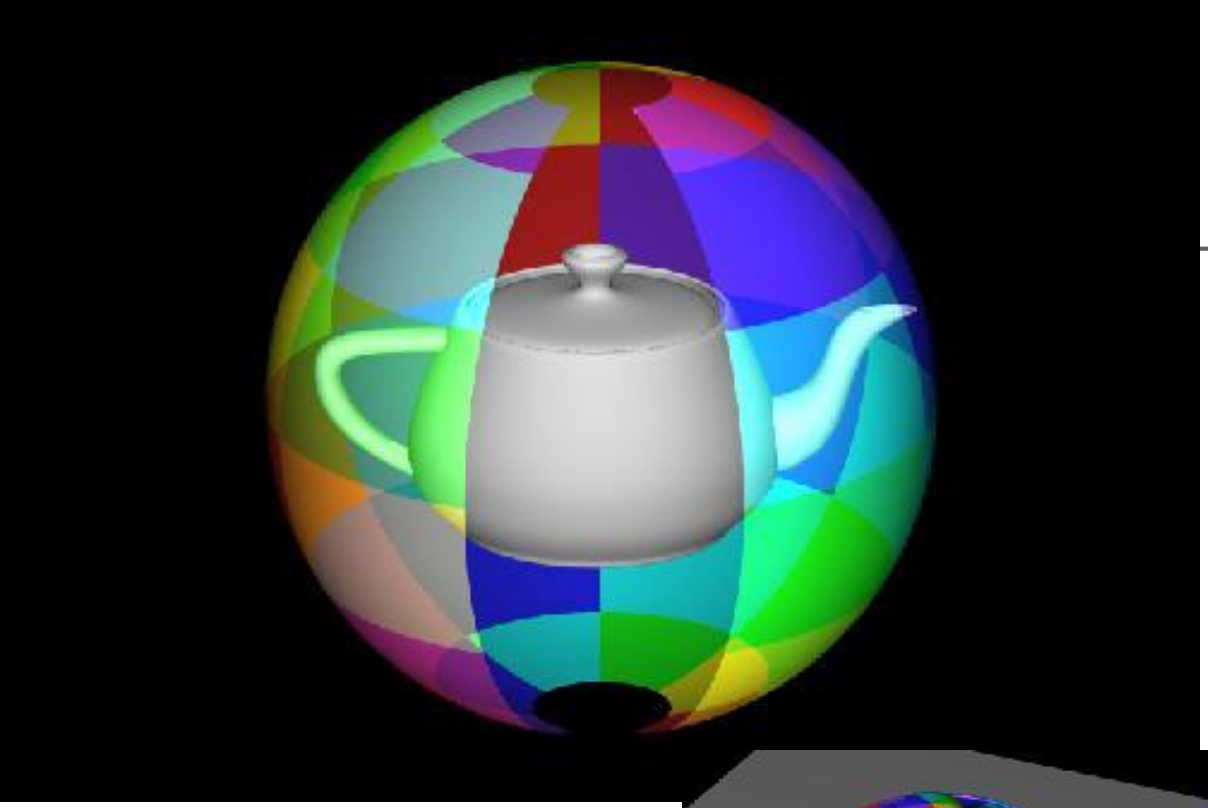
Intermediate surfaces

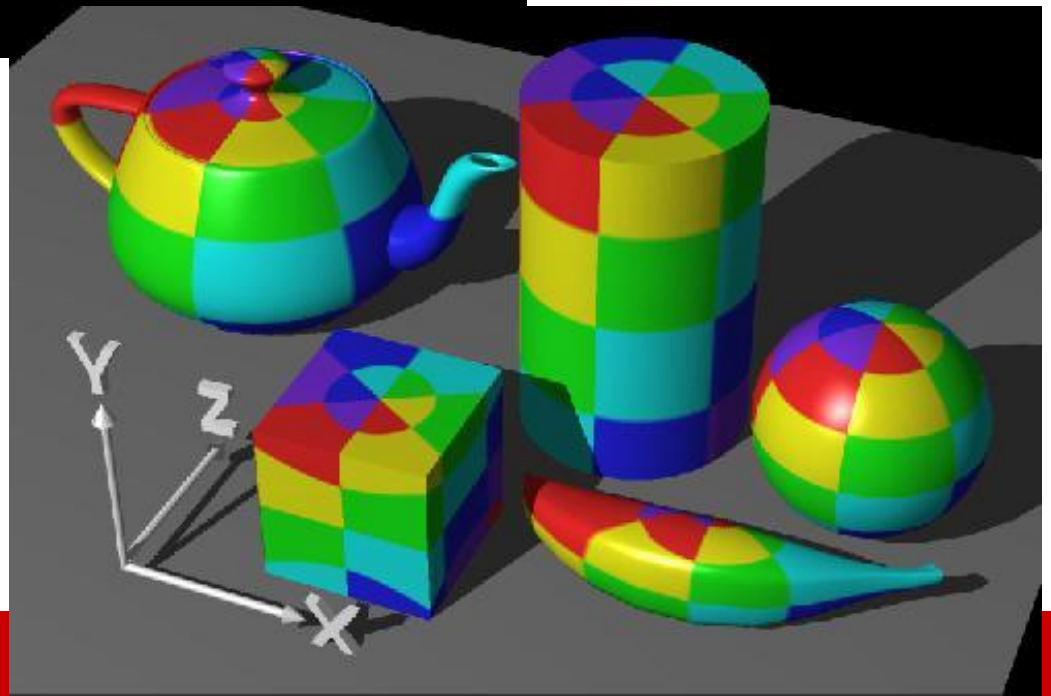
- Natural distorsion of the texture image onto the object.
- Choose the intermediate surface suitable to the object shape



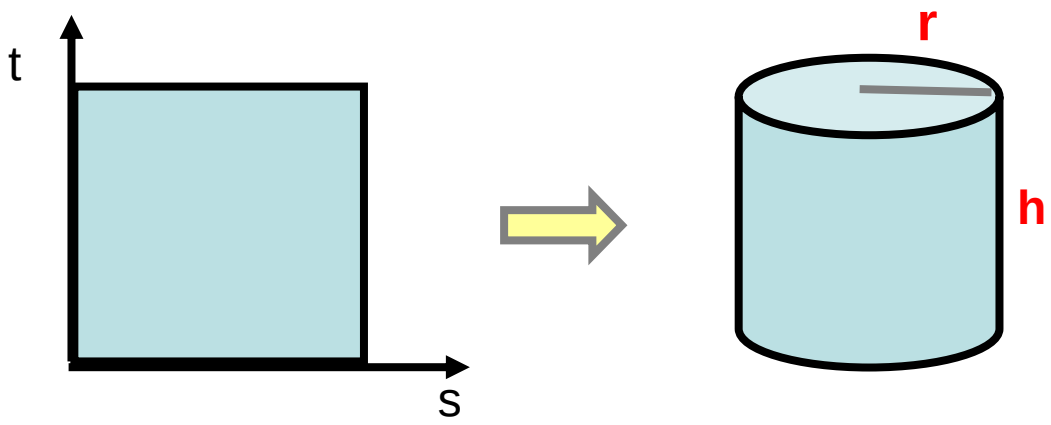
[Paul Bourke]


Y
Z
X




Y
Z
X

Cylindrical Mapping



STEP 1: (S Mapping)

Map the texture onto a cylinder with radius r and height h

parametric cylinder:

$$x = r\cos(2\pi u)$$

$$y = r\sin(2\pi u) \qquad u, v \in [0,1]$$

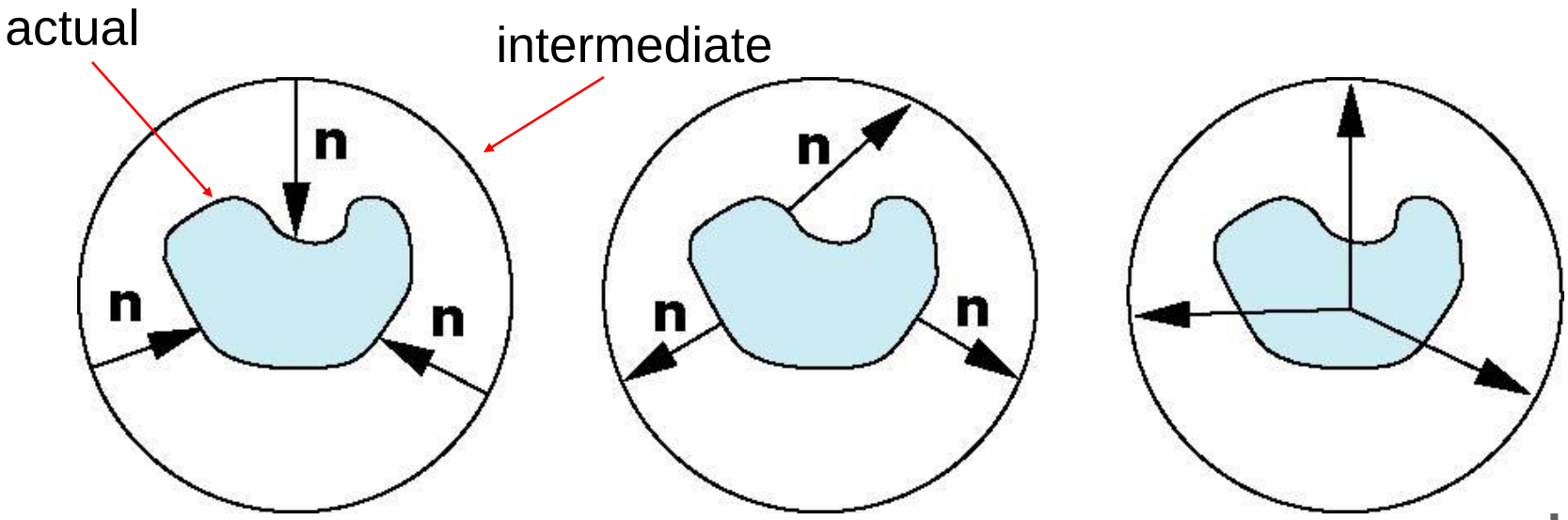
$$z = v / h$$

maps from texture space

$$mapping \quad s = u, \quad t = v$$

STEP 2: O Mapping

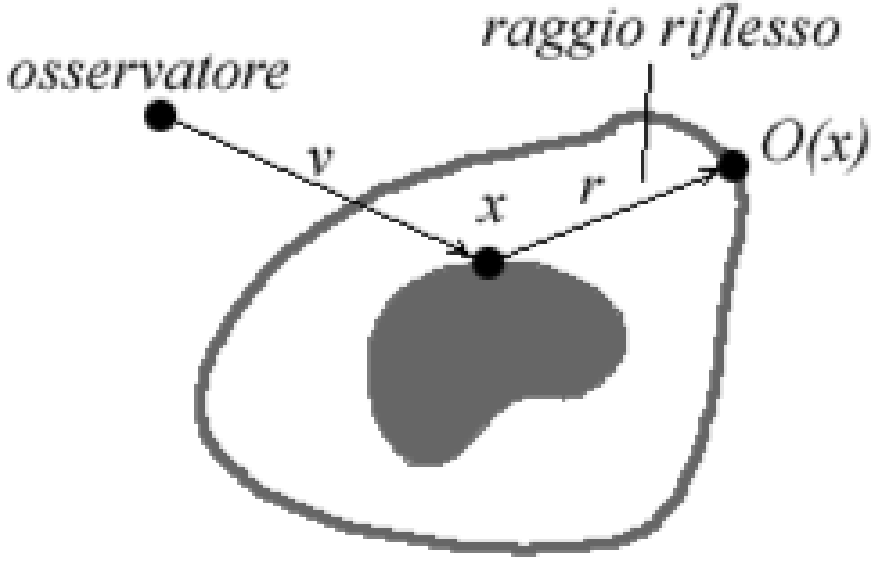
- Map from intermediate object to actual object
  1) Normals from intermediate to actual
  2) Normals from actual to intermediate
  3) Vectors from center of intermediate



O mapping

4) Reflected ray

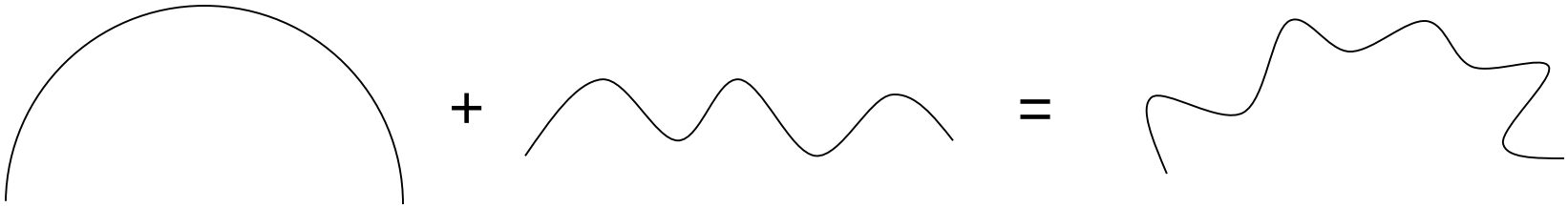
trace a ray from the camera to the object and compute the reflected ray to the intermediate surface



- **Normal/Bump Mapping**
- Procedural Texturing
- Environment Mapping

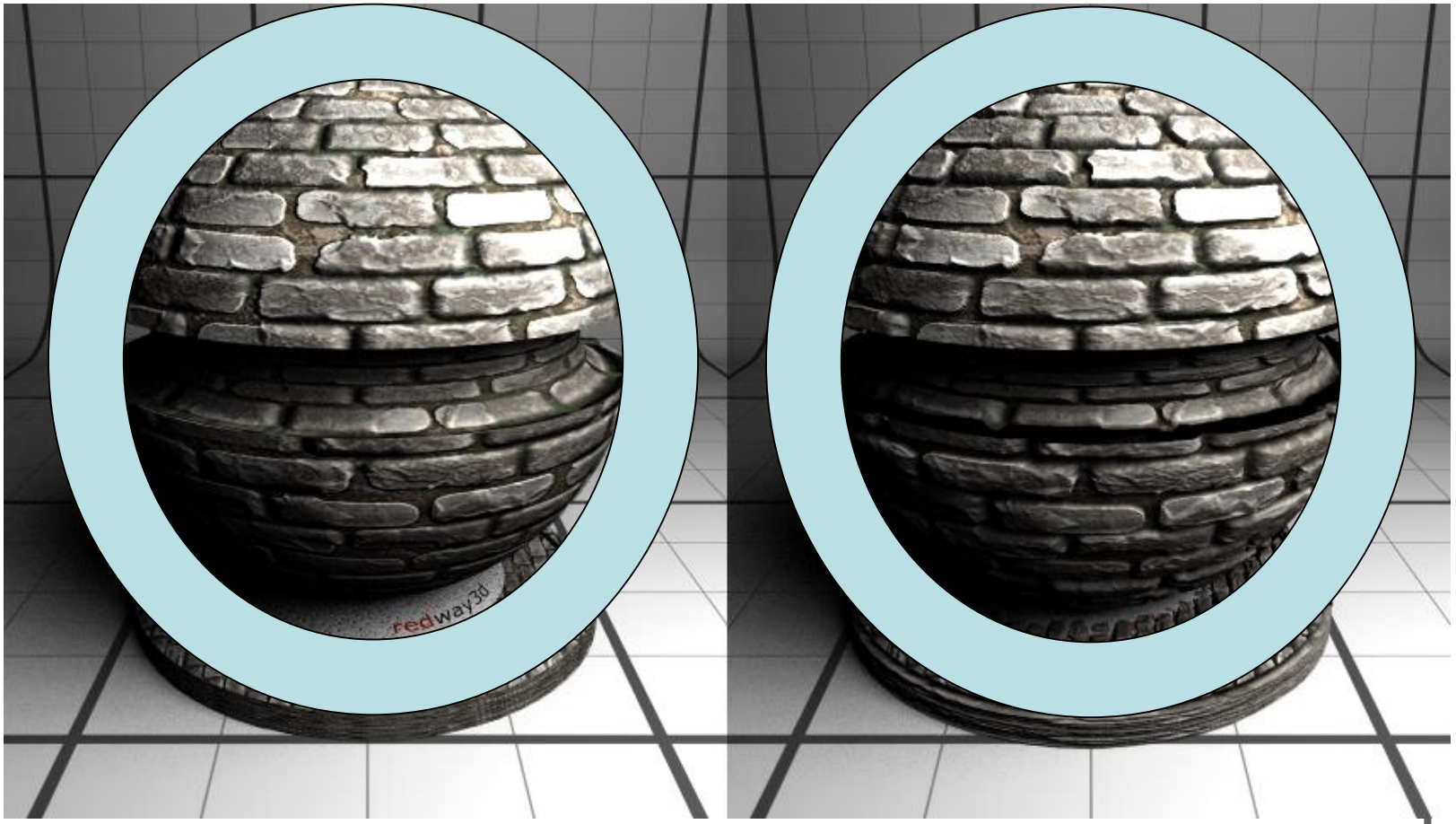
Displacement Mapping

Use texture to displace the surface geometry



Bump mapping only affects the normals,

Displacement mapping changes the entire surface (including the silhouette)


redway3d

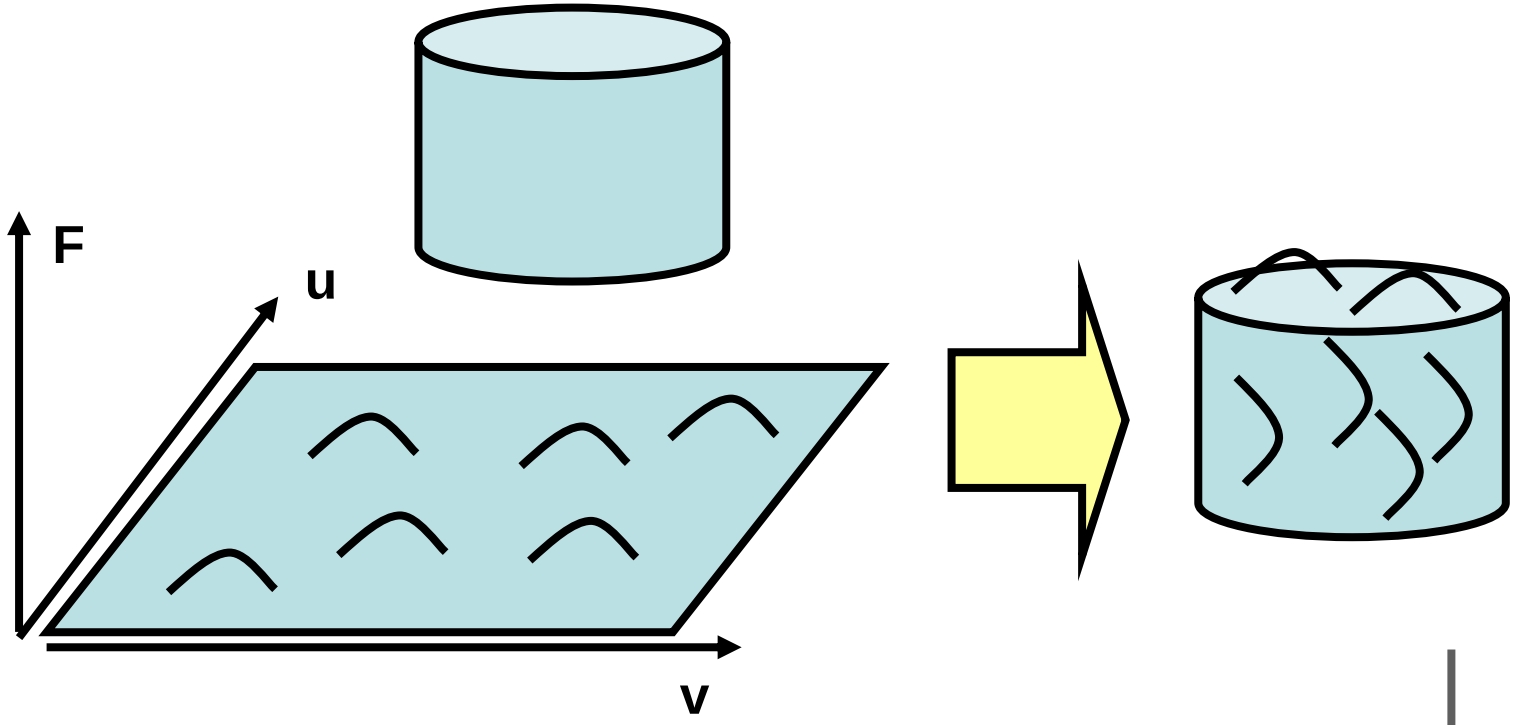
Bump/Normal Mapping

(Blinn 1978)

Use texture to perturb normals

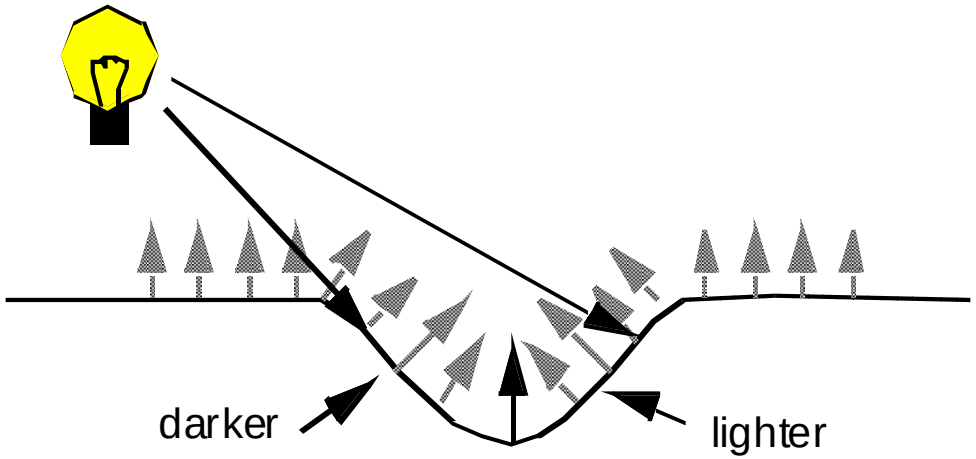
- creates a bump-like effect

- BUMP/NORMAL MAP: parametric function in (u,v) that defines the micro-structure

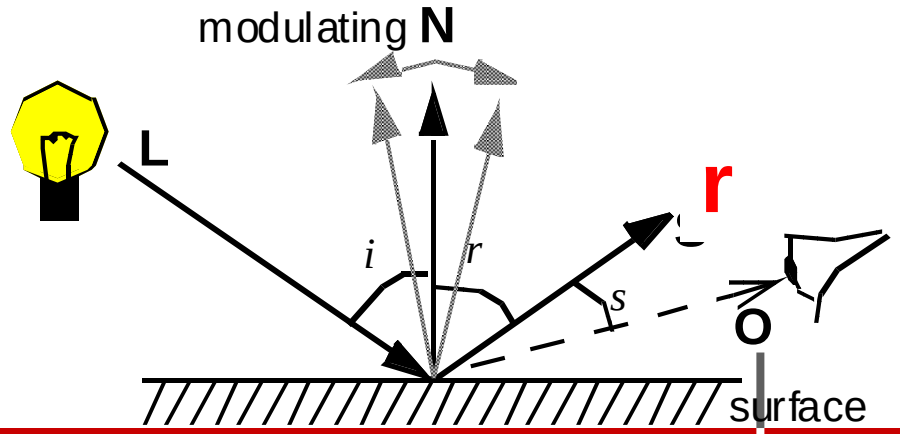


Normal Mapping or Bump Mapping: idea

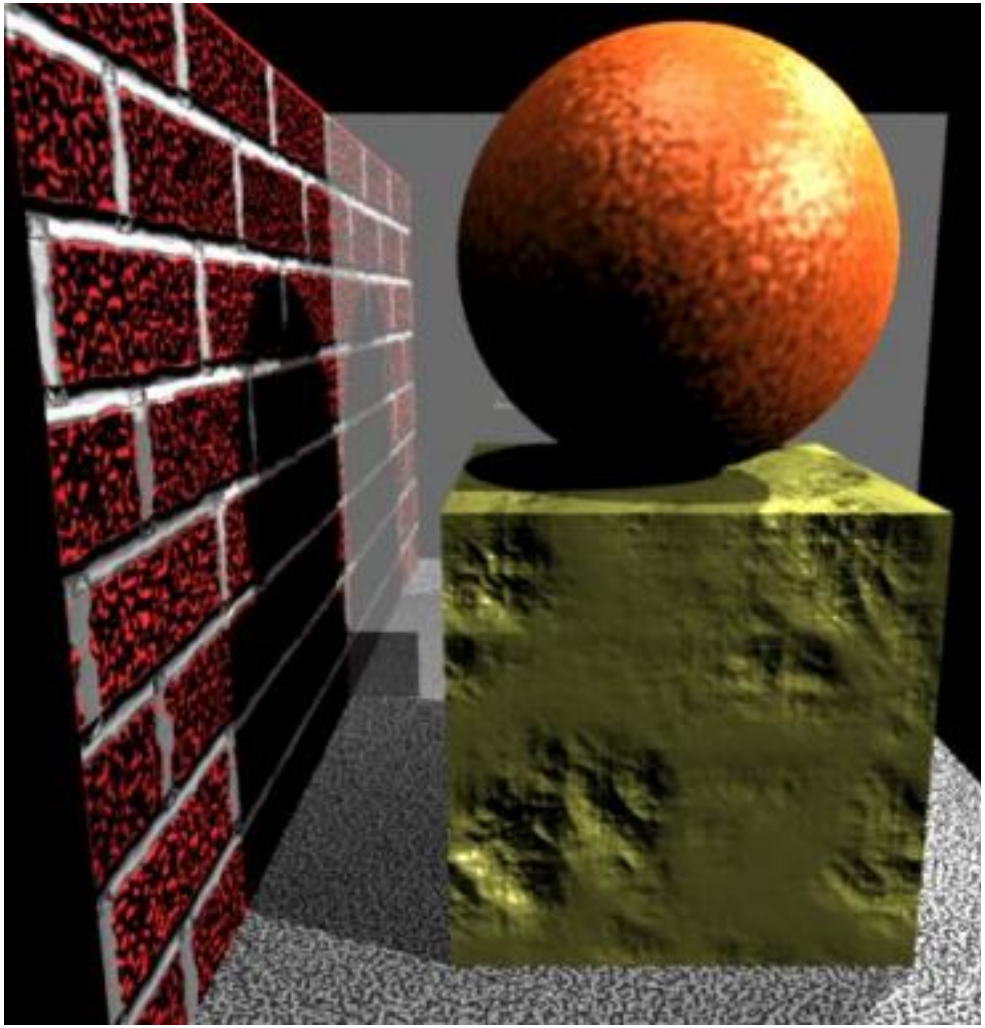
- **Shading** is computed using the normal vector to the surface



- **Modifying the normal vector**, modifies the reflected ray r and thus the light that reach the camera



Bump/Normal Mapping: problems

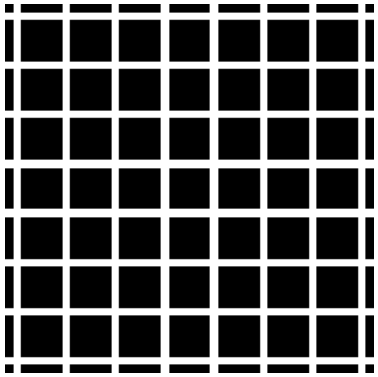


- Does not change silhouette edges

- For objects represented by meshes, the bumping effect is improved for high resolution meshes

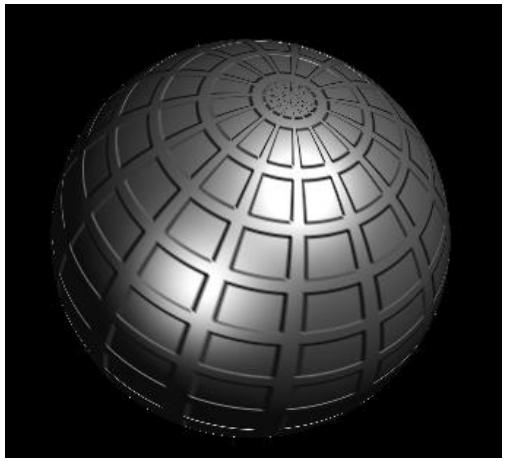
Bump mapping

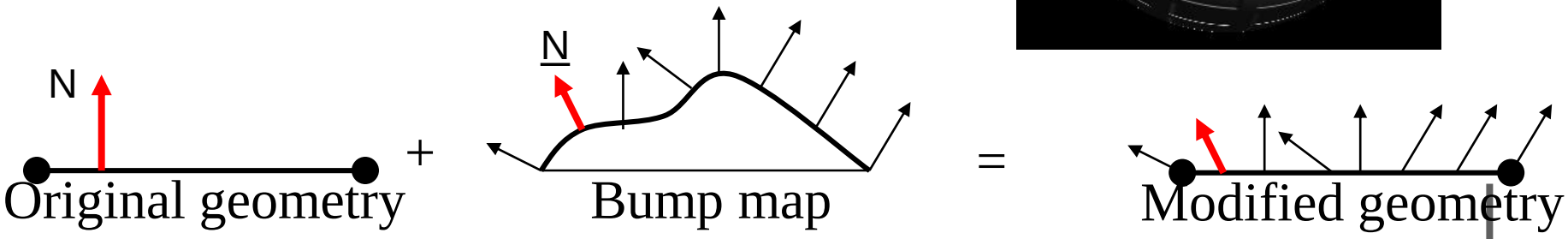
A bump map is a **single-channel (grey-scale) height map** $h(u,v)$ used to modify the normal direction of the object surface



Compute the new normal N from the bump map

Compute shading using this new normal N





Generating Bump map

- Define the tangent plane to the surface at a point P(u,v) by using the two vectors P_u and P_v.
- The normal is then given by:

$$\mathbf{N = P_u \times P_v}$$

- The new surface positions P' is then given by:

$$\mathbf{P'}(u,v) = \mathbf{P}(u,v) + B(u,v)\,\mathbf{N}$$

move P along the normal vector N of a quantity B(u,v)

This leads to the normal of perturbed surface:

$$\mathbf{N' = P'_u \times P'_v}$$

If B(u,v) is small, and N x N=0

- $\mathbf{P'_u} = \mathbf{P_u} + B_u \mathbf{N} + B(\mathbf{N})_u \quad \approx \mathbf{P'_u} = \mathbf{P_u} + B_u \mathbf{N}$
- $\mathbf{P'_v} = \mathbf{P_v} + B_v \mathbf{N} + B(\mathbf{N})_v \quad \approx \mathbf{P'_v} = \mathbf{P_v} + B_v \mathbf{N}$

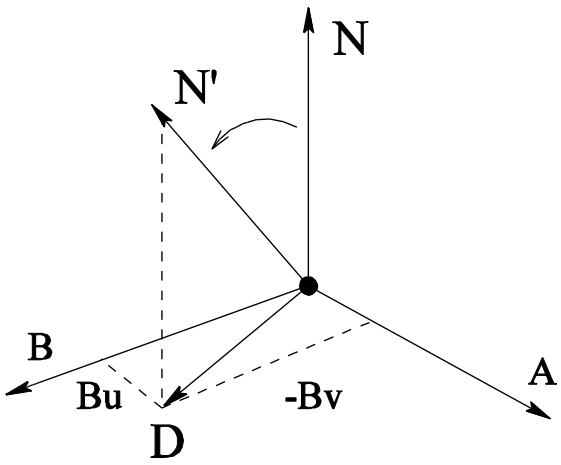
- $\mathbf{N'}(u,v) = (\mathbf{P_u} + B_u \mathbf{N})\times(\mathbf{P_v} + B_v \mathbf{N})$

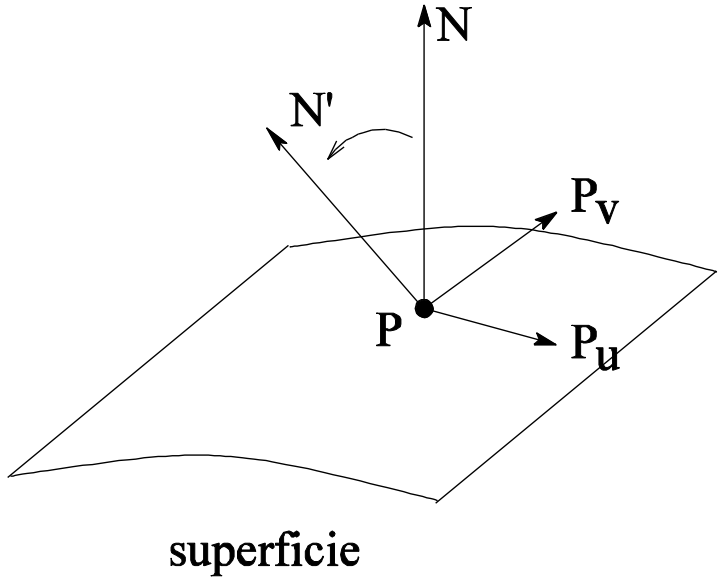
$$= \mathbf{P_u \times P_v} + B_u(\mathbf{N \times P_v}) + B_v(\mathbf{N \times P_u}) + B_u B_v(\mathbf{N \times N})$$

$$= \mathbf{N} + B_u(\mathbf{N \times P_v}) + B_v(\mathbf{N \times P_u})$$

$$= \mathbf{N + D}$$

$$\mathbf{N'}(u,v) = \mathbf{N} + B_u(\mathbf{N} \times \mathbf{P_v}) + B_v(\mathbf{N} \times \mathbf{P_u})$$
$$= \mathbf{N} + \mathbf{D}$$





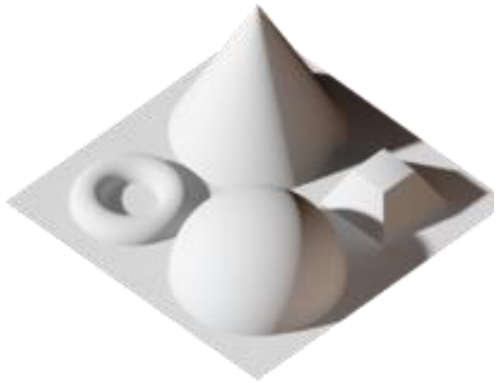
For efficiency, can store B_u and B_v in a 2-component texture map.
The cross products are geometry terms only.
N' will of course need to be normalized after the calculation and before lighting.

Normal Maps

A normal map is a **RGB image**: the RGB values of each texel represent the X,Y & Z values of a direction vector, used to modify the normal direction of the object surface

How to compute the RGB image?

Compute the normals $(\boldsymbol{n_T},\boldsymbol{n_B},\boldsymbol{n_N})$ from a height map (mode1) or object (mode2)



$(\boldsymbol{n_T},\boldsymbol{n_B},\boldsymbol{n_N})$ is a normalized vector, so each element is in the range of $[\mathbf{-1},\mathbf{1}]$ They are converted to a range of $[\mathbf{0},\mathbf{255}]$ and stored as colors in the texture

$$(\mathrm{R},\mathrm{G},\mathrm{B}) = 255 * \frac{(n_T, n_B, n_N) + (1,1,1)}{2}$$

Normal Maps

Compute lighting/shading using this new normal

Example of a normal map (center) with the scene it was calculated from (left) and the result when applied to a flat surface (right) By Julian Herzog

- Larger storage for an RGB image as opposed to single channel bump map for a height field

How to generate Normal Maps?

Mode 1: from an height map image

Height map
Dark = lower
Bright = higher

Filter
(e.g.Photoshop)

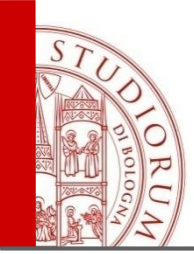
Normal map

Algorithm:

∀ texel *t of the* height map (x , y , z = height[x,y])

compute N=(n_T, n_B, n_N) from the neighbors of t

associate N to t



Compute (n_T, n_B, n_N)

-Points on the three dimensional surface are given by

$$(u, v, h(u,v))$$

-The tangent plane at the point $(u_0, v_0, h(u_0, v_0))$is given by

$$-\left.\frac{\partial h(u,v)}{\partial u}\right|_{(u_0,v_0)}(u-u_0)-\left.\frac{\partial h(u,v)}{\partial v}\right|_{(u_0,v_0)}(v-v_0)+(h-h(u_0,v_0))=0$$

-The partial derivatives can be computed using finite differences, e.g.,

$$\left.\frac{\partial h(u,v)}{\partial u}\right|_{(u_0,v_0)}=\frac{(h(u_{right},v_{center})-h(u_{left},v_{center}))}{u_{left}-u_{right}},$$

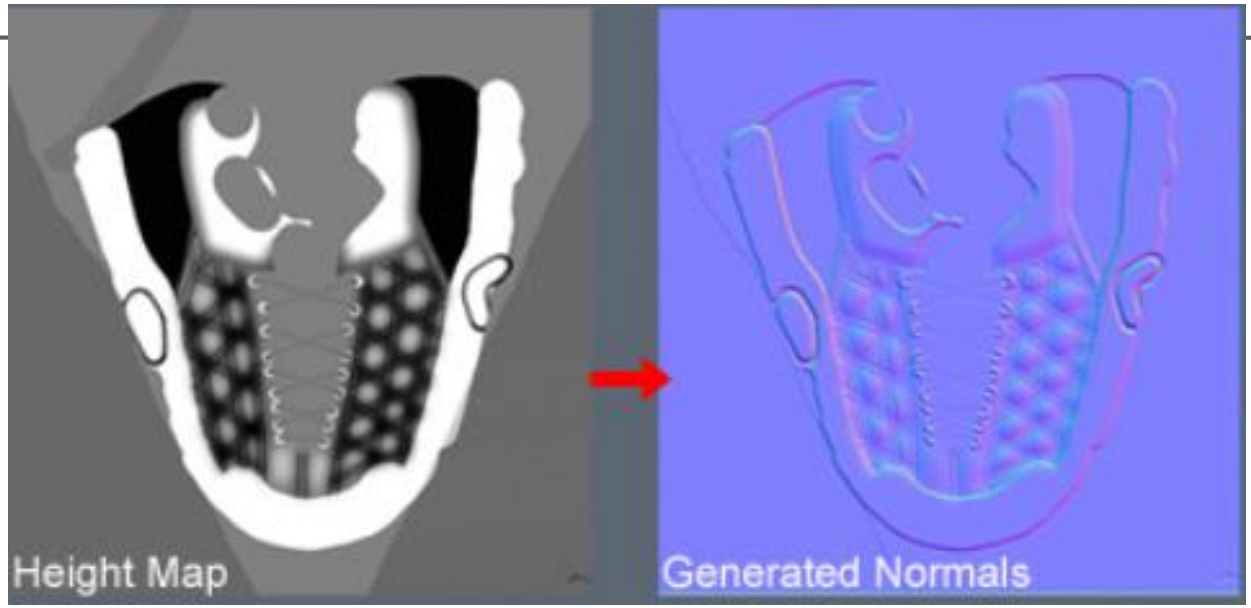
$$\left.\frac{\partial h(u,v)}{\partial v}\right|_{(u_0,v_0)}=\frac{(h(u_{center},v_{right})-h(u_{center},v_{left}))}{v_{left}-v_{right}}$$

-The outward (non-unit) normal direction to this tangent plane is

$$\left(-\left.\frac{\partial h(u,v)}{\partial u}\right|_{(u_0,v_0)},-\left.\frac{\partial h(u,v)}{\partial v}\right|_{(u_0,v_0)},1\right)$$

-Normalize to get a unit normal (n_T, n_B, n_N)

Normal maps



Normal prevailing: X=~0 , Y=~0 , Z=~1

⇨

color prevailing: R =~0.5 , G=~0.5, B=~1
(~light blue)

How to generate Normal Maps?

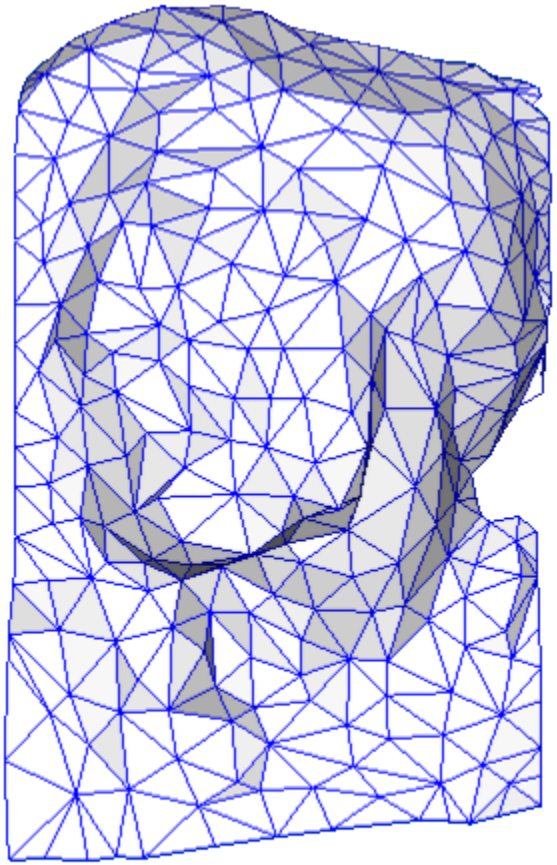
Mode 2: From a high-resolution 3D model

- Model a detailed mesh
- Generate a **parameterization** for the mesh

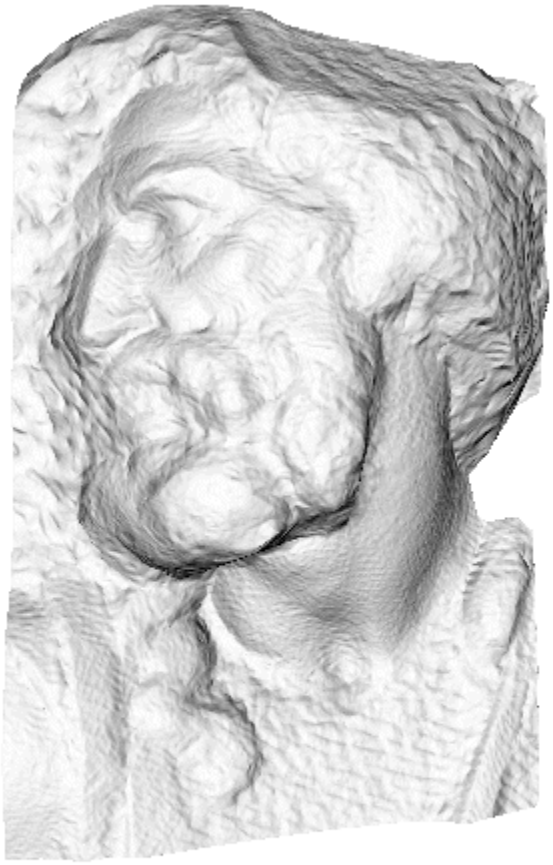
  (each 3D point has **unique** image coordinates in the 2D texture map)
- Simplify the mesh
- Overlay simplified and original model
- For each point **P** on the simplified mesh, find closest point **P**' on original model (ray casting)
- Store the normal at **P**' in the normal map. **Done!**

Generating Normal Maps (2)

original mesh
4M triangles



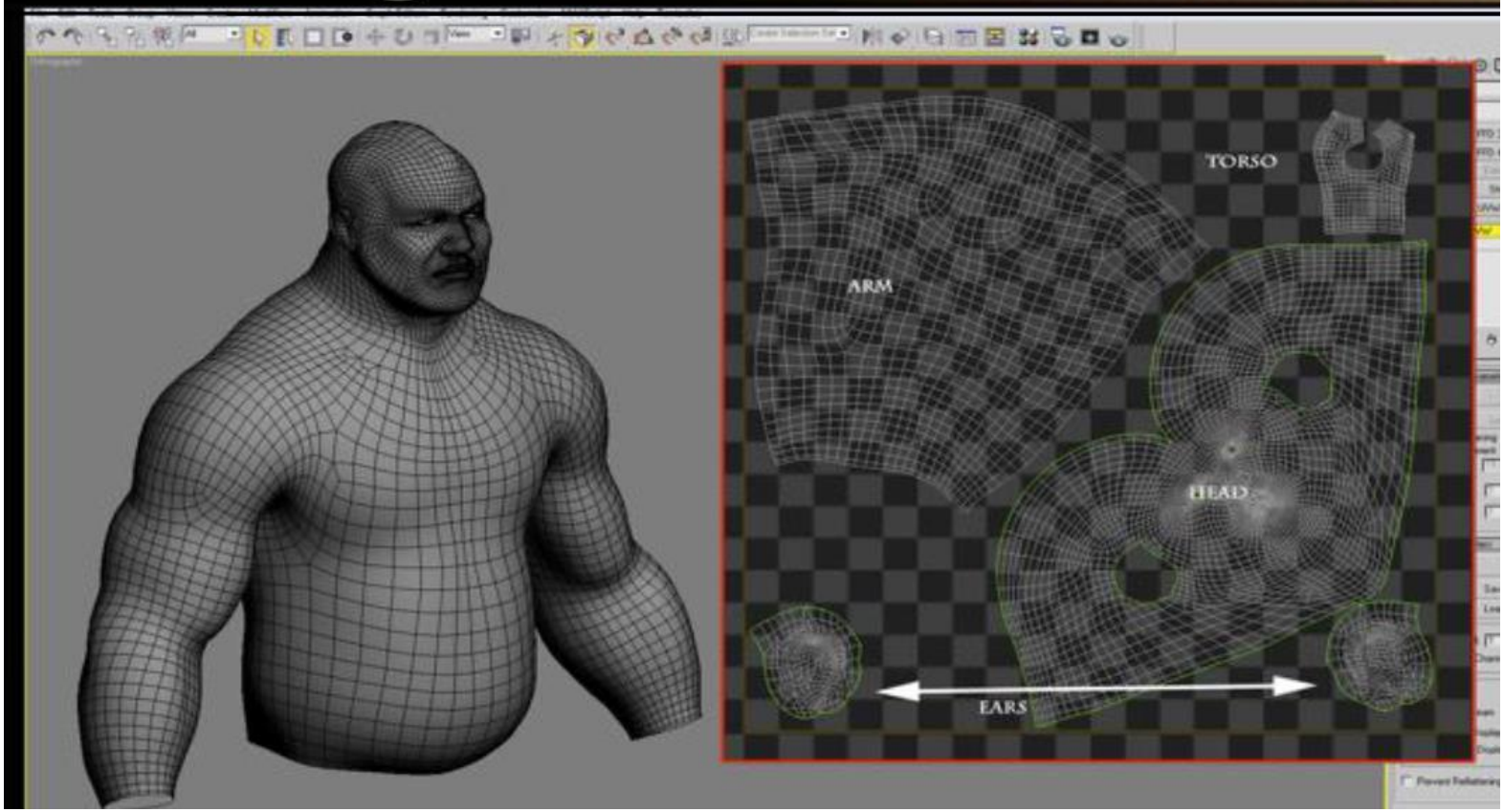
simplified mesh
500 triangles



simplified mesh
and normal mapping
500 triangles

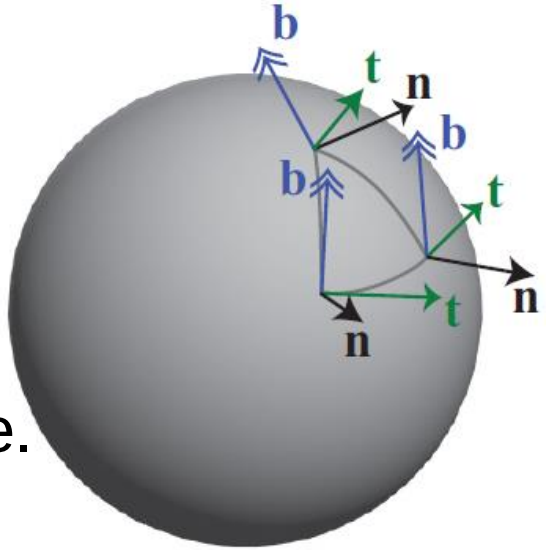
Slide from Epic Games

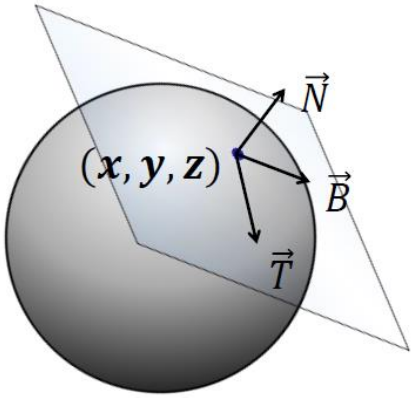
Creating Torso Portion in Max



How to use Bump/Normal maps:
Defining a local Coordinate System

Texture/Tangent space (TBN) is a coord. system attached to the local surface defined for each point (vertex) of the surface from the vectors



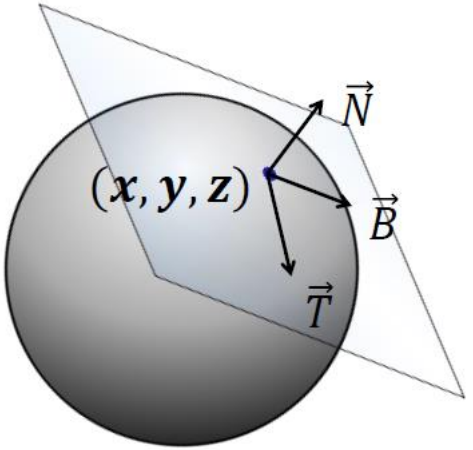


- **N**ormal (perpendicular to the surface)
- **T**angent (tangent to the surface)
- **B**i-Tangent (tangent to the surface)

where B and T span the texture.

- **Compute (N,T,B) for each vertex of the mesh**
- **Interpolate for inner points (fragment) of the triangle**

How to compute the Tangent space TBN



- Use the mapping between each vertex $(\boldsymbol{x},\boldsymbol{y},\boldsymbol{z})$ and the texture coordinates $\boldsymbol{u},\boldsymbol{v}$ to calculate T,N, and B

- Let $\boldsymbol{u}$ be the Tangent direction and $\boldsymbol{v}$ be the Bitangent direction
- Assume that $\boldsymbol{x},\boldsymbol{y}$,and $\boldsymbol{z}$ are each linear functions of $\boldsymbol{u}$ and $\boldsymbol{v}$ in each triangle:

$$x = a_0 u + b_0 v + c_0$$
$$y = a_1 u + b_1 v + c_1$$
$$z = a_2 u + b_2 v + c_2$$

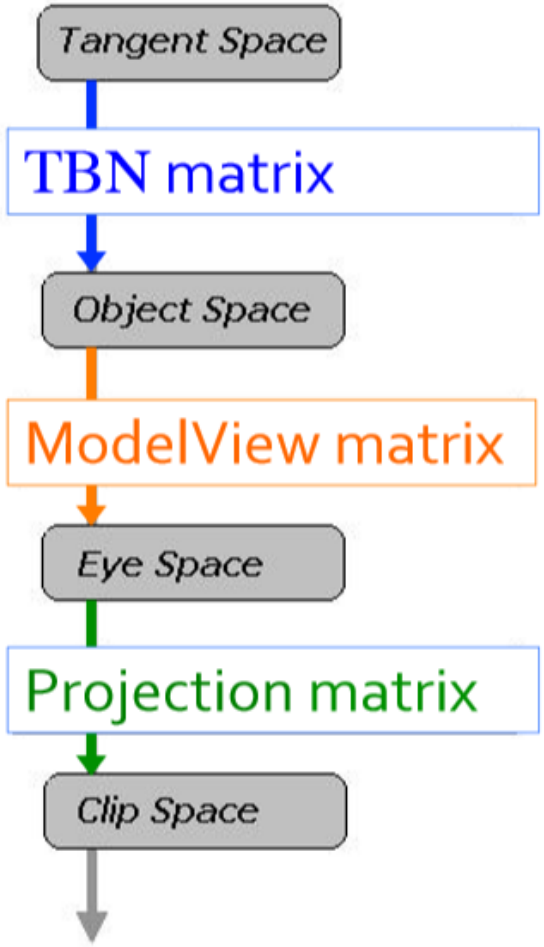
- For each of the 3 vertices, plug the known values of $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{z}$, $\boldsymbol{u}$, and $\boldsymbol{v}$ into the above equations: this gives 9 total equations, or 3 sets of 3 equations
- That is, there is a 3x3 system of equations for $\boldsymbol{a_0}$, $\boldsymbol{b_0}$, $\boldsymbol{c_0}$, and likewise a 3x3 system for $\boldsymbol{a_1}$, $\boldsymbol{b_1}$, $\boldsymbol{c_1}$, and likewise a 3x3 system for $\boldsymbol{a_2}$, $\boldsymbol{b_2}$, $\boldsymbol{c_2}$

- After solving, we set $\boldsymbol{T}=(\boldsymbol{\partial x/\partial u},\boldsymbol{\partial y/\partial u},\boldsymbol{\partial z/\partial u})=(\boldsymbol{a_0},\boldsymbol{a_1},\boldsymbol{a_2})$,
  $\boldsymbol{B}=(\boldsymbol{\partial x/\partial v},\boldsymbol{\partial y/\partial v},\boldsymbol{\partial z/\partial v})=(\boldsymbol{b_0},\boldsymbol{b_1},\boldsymbol{b_2})$, **and**
  $\boldsymbol{N}=\boldsymbol{T}\times\boldsymbol{B}$

normal mapping applied in Texture/Tangent space (TBN)

Once we have T, B, N vectors, we also have matrix TBN which enables us to go from Tangent Space to Object/Word Space :

$$TBN = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$



[Lengyel, Dreijer Madsen]

There are basically two ways we can use a TBN matrix for normal mapping:

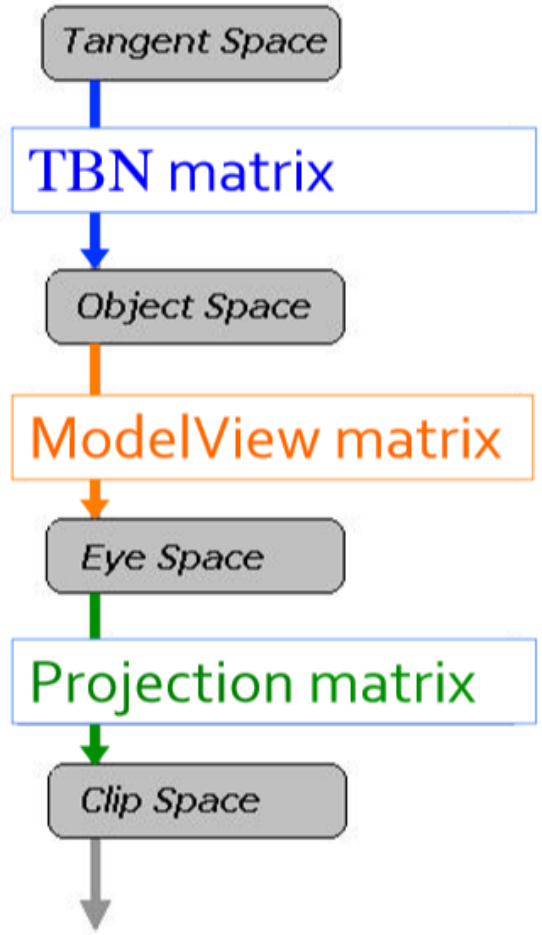
1) Transform the sampled normal from tangent space to object/world space using the TBN matrix; the normal is then in the same space as the other lighting variables.

normal mapping applied in Texture/Tangent space (TBN)

2)

We take the inverse of the TBN matrix that transforms any vector from object/world space to tangent space and use this matrix to transform not the normal, but the other relevant lighting variables (light and view pos) to tangent space; the normal is in the same space as the other lighting variables. Matrix inverse (TBN is an orthogonal matrix):

$$TBNinv = (TBN)^T$$



[Lengyel, Dreijer Madsen]

Tangent space normal mapping

In app:

- Load normal map into its own texture unit
- Pass per triangle N,T, and B to vertex shader

In vertex shader:

- Create the TBN matrix and TBNinv matrix
- Compute lighDir L and ViewDir V in tangent space,

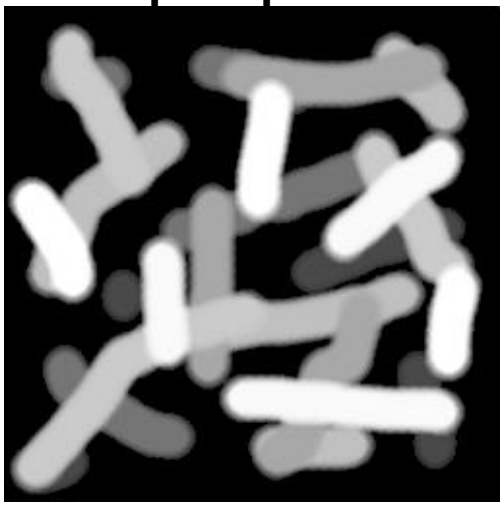
In fragment shader:

Keep the extracted normals as-is

- sample normal map (texture) per texel (n)
- convert RGB for n into [-1,1]
- compute lighting in tangent space using normalized n, l, and v

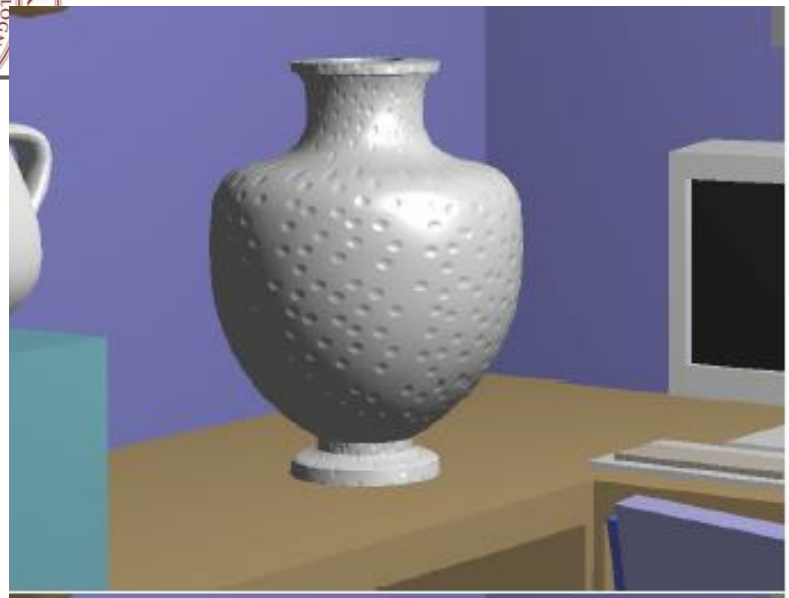
Texture mapping + Bump mapping

Bump map



Color map

(From 3D Games by Watt et al.)

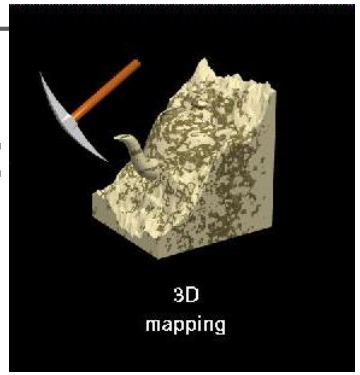



BUMP
MAPPING

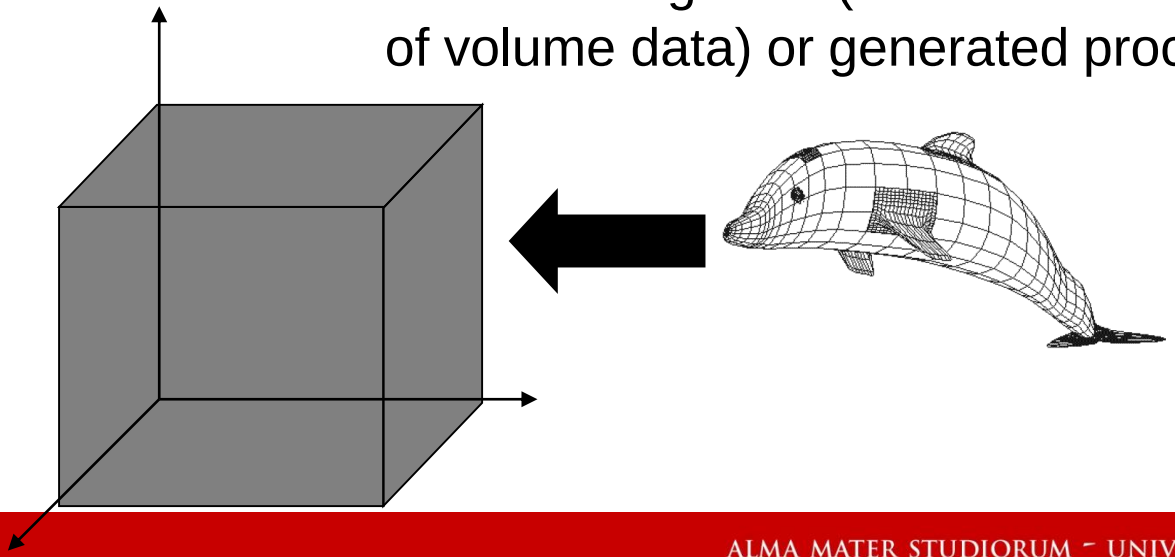
- Normal/Bump Mapping
- **Procedural Texturing**
- Environment Mapping

Image by Henrik Wann Jensen

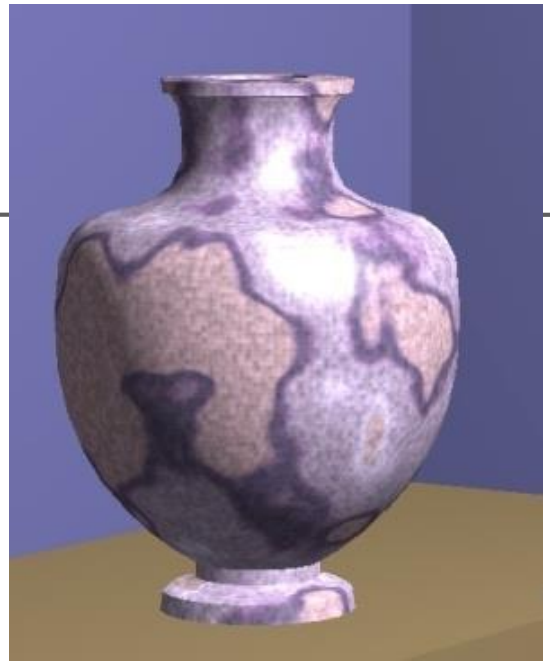
Solid texturing :Texture 3D



- Problem: mapping a 2D image/function onto the surface of a general 3D object is a difficult problem:
  - Distortion
  - Discontinuities
- Idea: use a texture function defined over a 3D domain - the 3D space containing the object -
  - Texture function can be digitized (stored as a 3D array of volume data) or generated procedurally



3D Texture



sweeping of a 2D texture

Texture 3D can be obtained by:

- 3D **Images**
- **Procedural**

Little program that computes color as a function of *x,y,z*:

color = f(x,y,z)

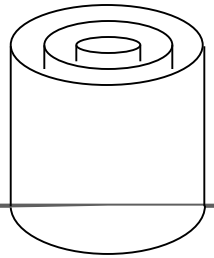
computed *on-the-fly*.

Used also for 1D,2D,4D, textures..

Texturing a model with a marble solid texture can achieve a similar effect to having carved the model out of a solid piece of marble

Procedural Texture

Procedural Texturing

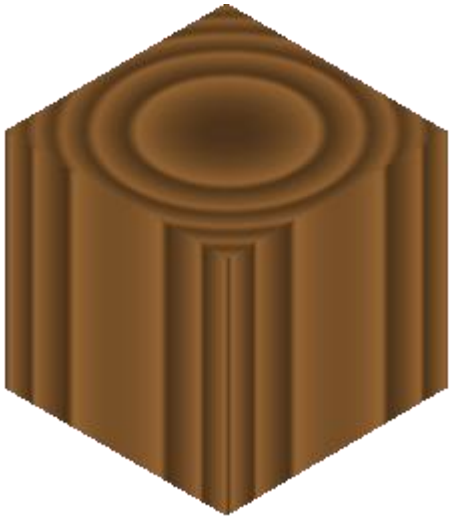


$f(s,t,r) = s^2 + t^2$

Woodmap(f)

- Texture map is a function
- Write a procedure to perform the function
  - input: texture coordinates - s,t,r
  - output: color, opacity, shading
- Example: Wood
  - Classification of texture space into cylindrical shells

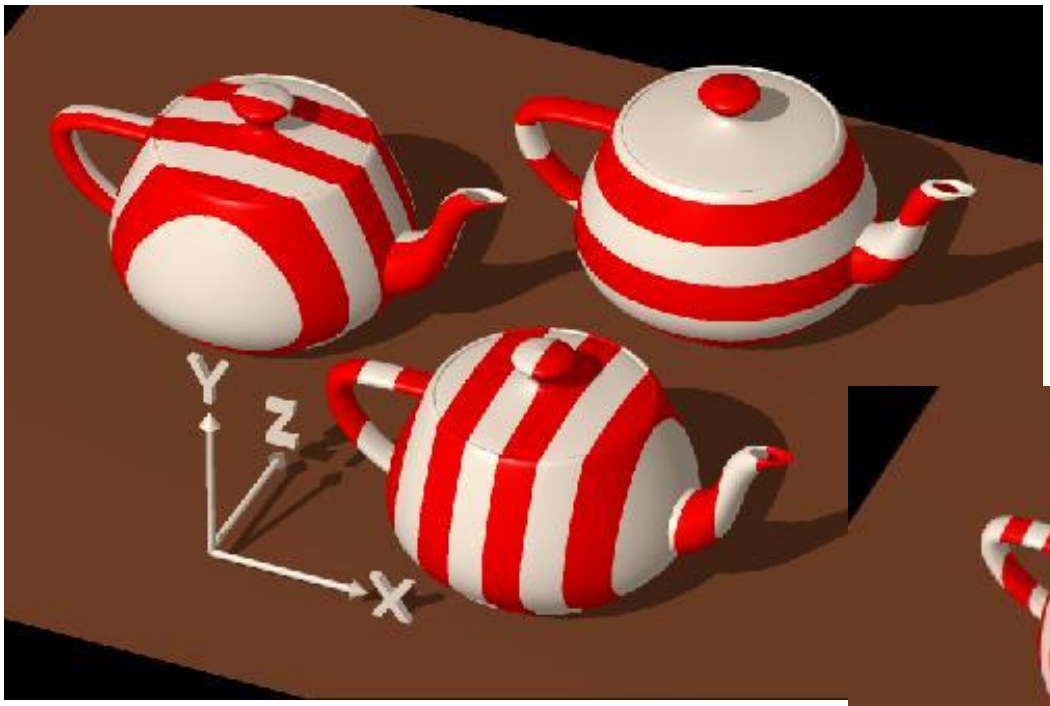
    $f(s,t,r) = s^2 + t^2$
  - Outer rings closer together, which simulates the growth rate of real trees
  - Wood colored color table
    - Woodmap(0) = brown “earlywood”
    - Woodmap(1) = tan “latewood”



Wood(s,t,r)

Wood(s,t,r) = Woodmap($f(s,t,r)$ mod 1)

Procedural texturing: Examples



Texture 2D vs 3D – Pros & Cons

Pros

- No need to parameterize surface
- No worries about distortion
- Objects appear sculpted out of solid substance
- Simulate realistic natural material

Cons

- **For 3D Images 3D**: alta occupazione di memoria delle texture (una texture 64x64x64 RGBA occupa 1MB, una texture 256x256x256 RGBA occupa 64MB)
- For **procedural texture:** computational overhead which limits the usage for real time rendering, pre-computed and stored
- Unrealistic Deformations

Solid Texture Problems

- How can we deform an object without making it swim through texture?
- How can we efficiently store a procedural texture?

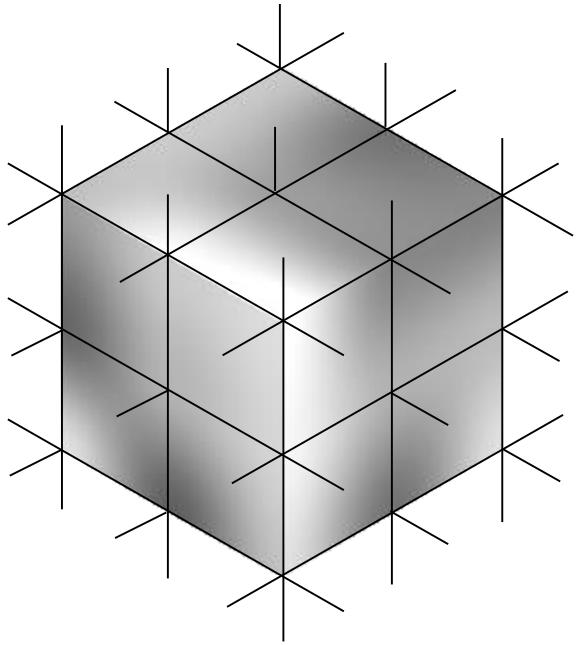
Noise Functions (K.Perlin)

- Add “noise” to make textures interesting
- Make a large grid with random numbers on each grid node

Interpolate the noise to the points inside the lattice cells

This gives spatial coherency

Ken Perlin proposed a specific method for implementing this



- *Noise*(*x*,*y*,*z*)

returns a single pseudo-random number in [-1,1]

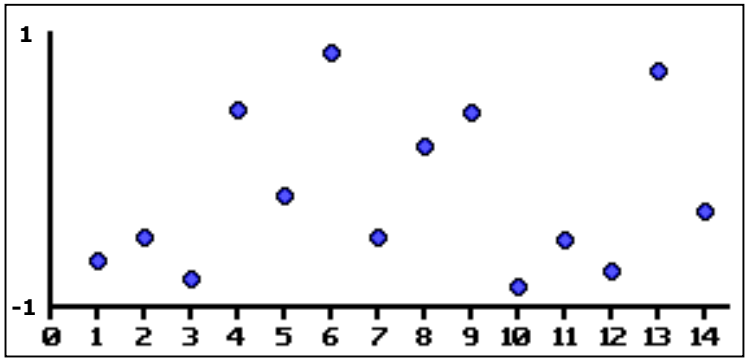
Noise( ) : $\mathbb{R}^n \longrightarrow$ **[-1, 1]**

Perlin Noise "Tron" 1981

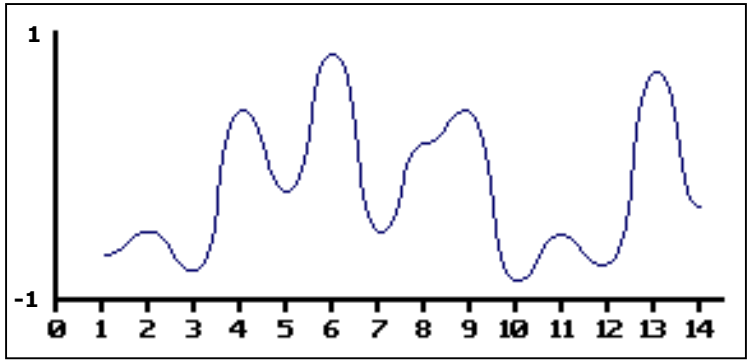
- Pseudo-random function

Noise( ) : $\mathbb{R}^n \longrightarrow$ [-1, 1]

1D



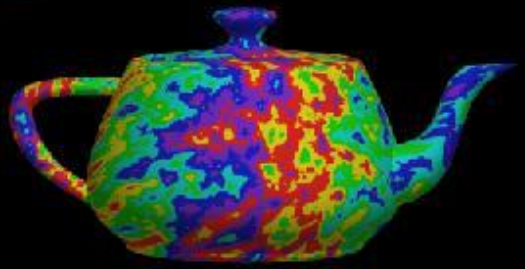
Random -points



Noise Function: interpolating function

http://mrl.nyu.edu/~perlin/noise/

Noise



rings

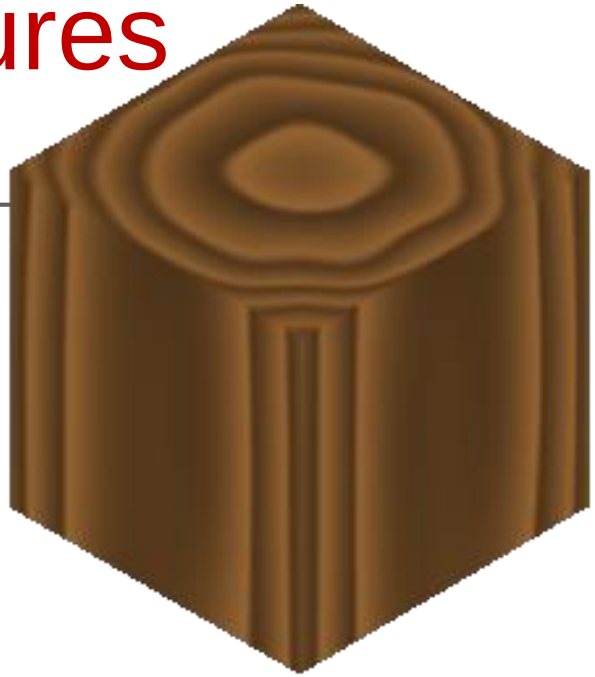
added eccentricity

added twist

added tilt

Noise For Solid Textures

WOOD

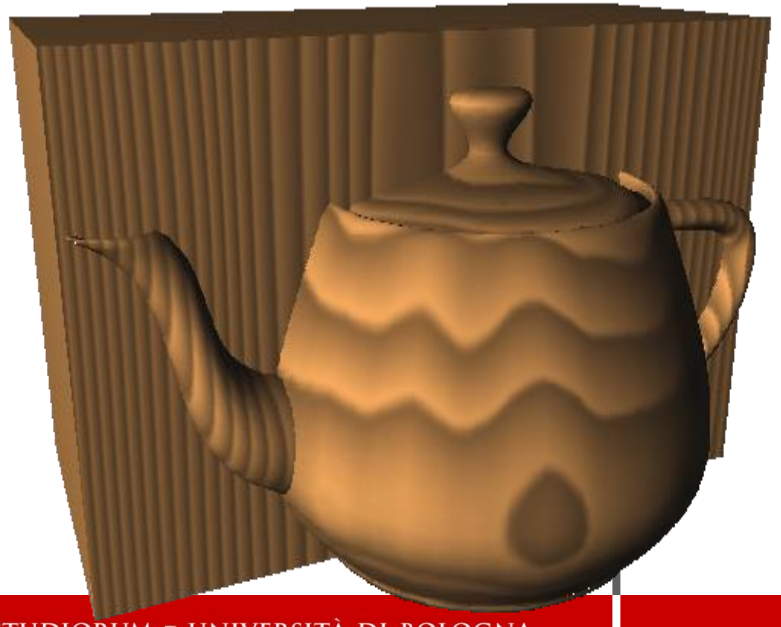


- Add noise to cylinders to warp wood
  - Wood($s^2 + t^2 + N(s,t,r)$)
- Controls
  - **Amplitude:** power of noise effect

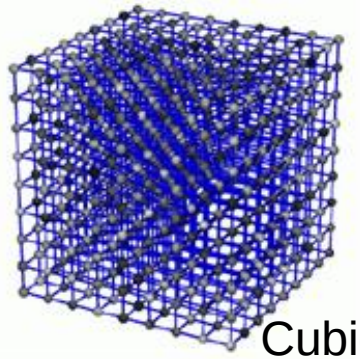
    $a\,N(s, t, r)$
  - **Frequency:** coarse vs. fine detail

    $N(f_s\,s, f_t\,t, f_r\,r)$
  - **Phase:** location of noise peaks

    $N(s + \phi_s, t + \phi_t, r + \phi_r)$

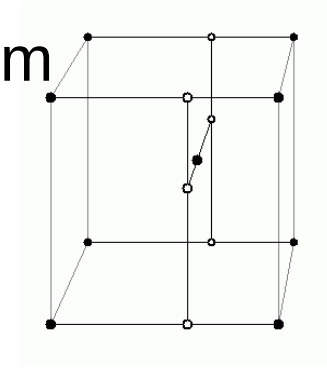


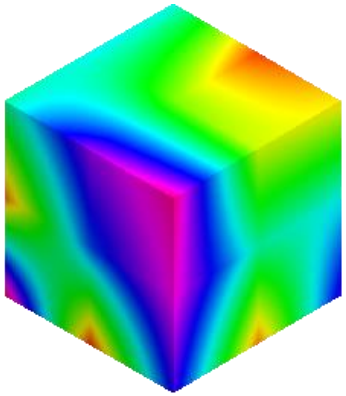
Making Noise



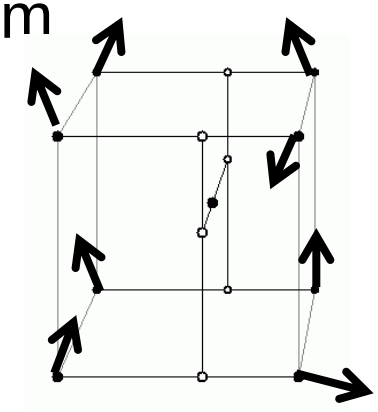
Cubic lattice

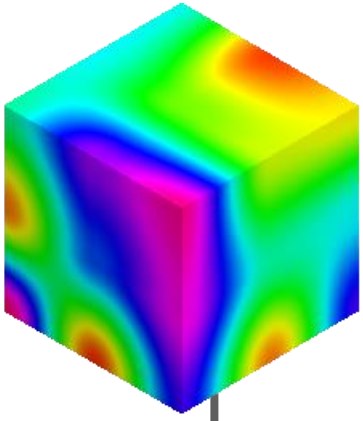
- Good:
  - Create 3-D array of pseudo-random values
  - Trilinearly interpolate





- Better
  - Create 3-D array of pseudo-random 3D-vectors
  - Splines (cubic) to interpolate



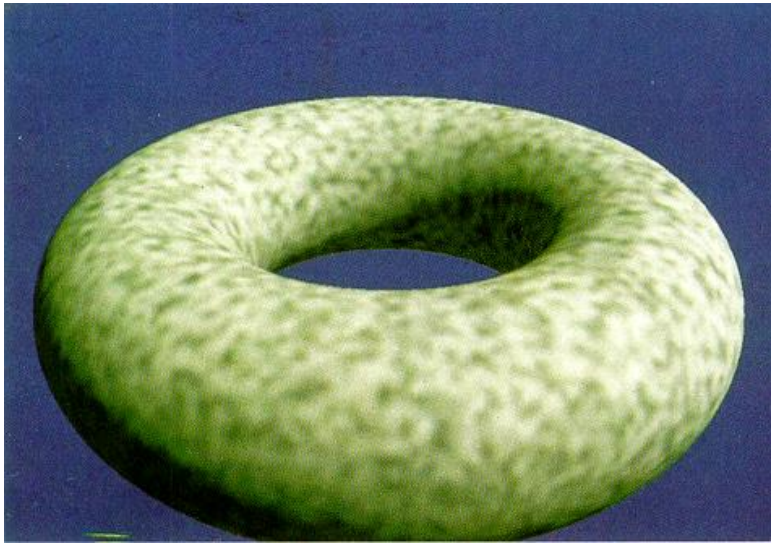


http://www.noisemachine.com/talk1/imgs/flame500.html

Noise - example

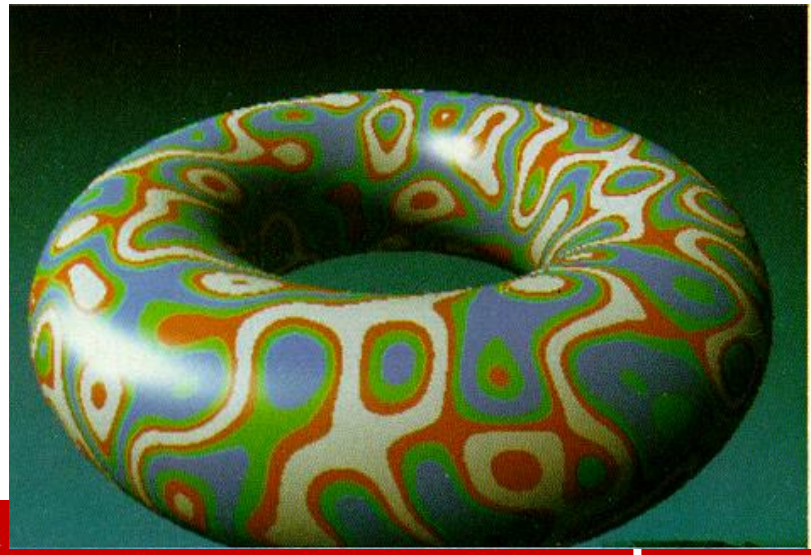
Apply noise to the object colors

```
Color = white * Noise(point);
```



```
Color = Colorful( Noise(k*point));
```

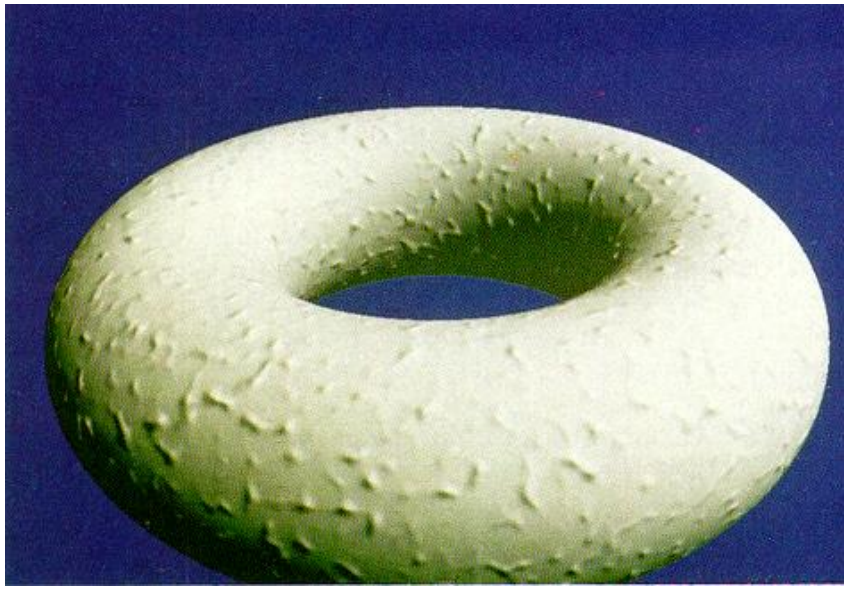
%map different ranges of values
into different colors



Bump Mapping Effect

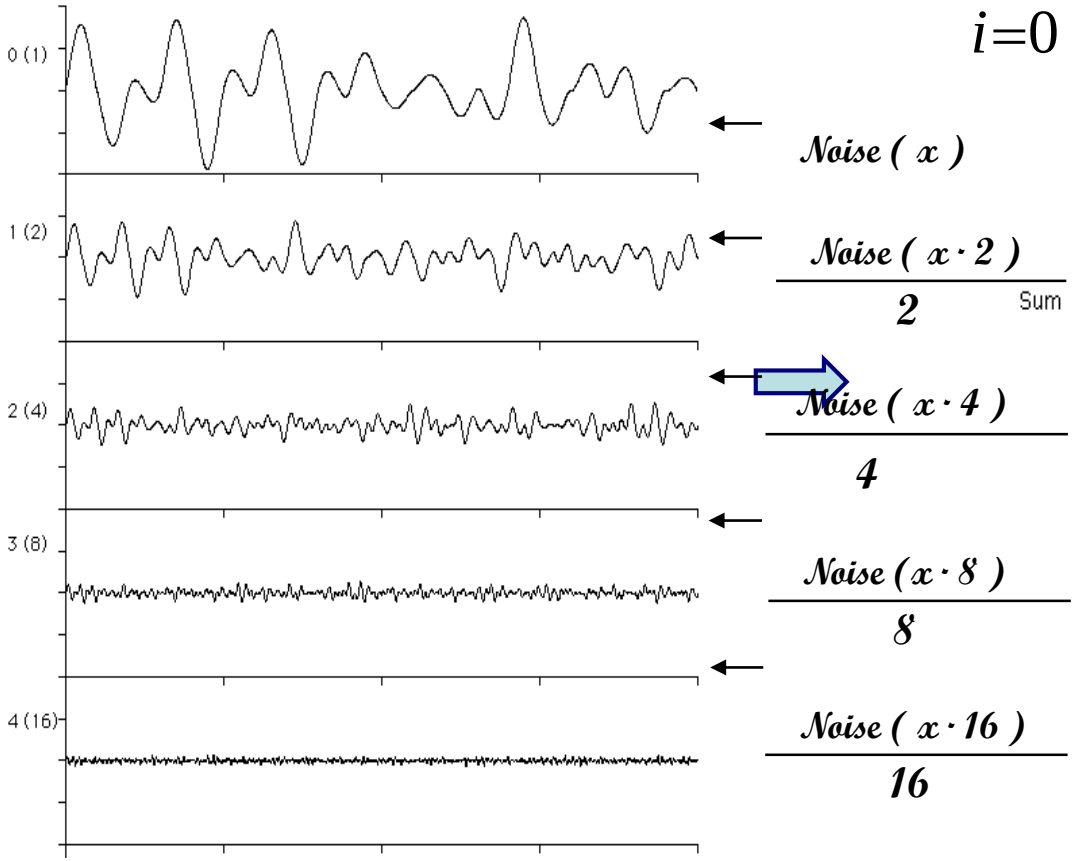
```
Normal = Normal +DNoise(point);
```

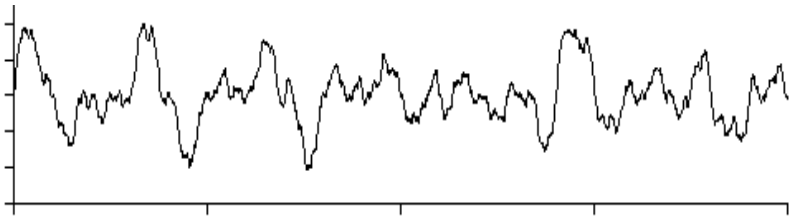
vector valued differential of *the Noise() signal, defined by the instantaneous rate of change of Noise() along the x, y, and z directions, respectively. We will call* this function *Dnoise.*



Perlin Turbulence

$$Turbulence(x) = \sum_{i=0}^{k} \frac{1}{2^i} \left| noise(2^i x) \right|$$

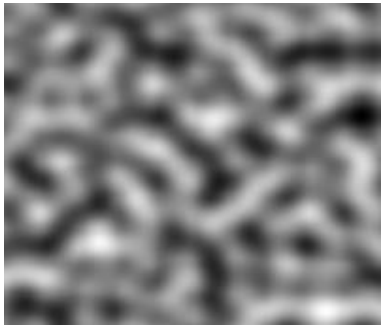




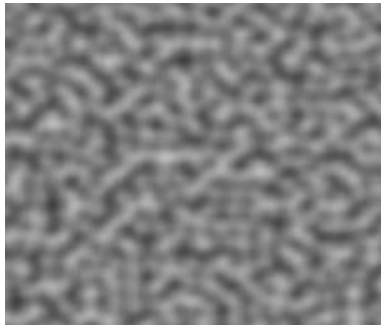
Perlin Turbolence

Add several octaves of noise function
Scale amplitude appropriately

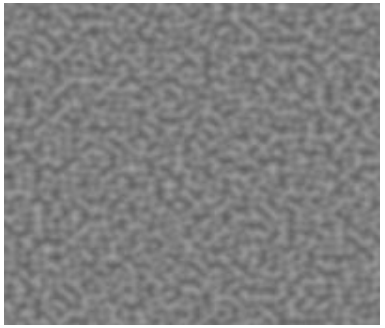
$$Turbulence(x) = \sum_{i=0}^{k} \frac{1}{2^i} \left| noise(2^i x) \right|$$



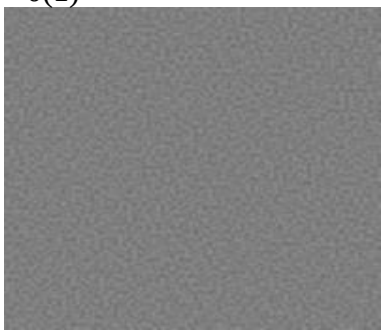
0(1)



1 (2)

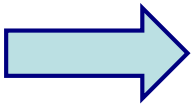


2 (4)

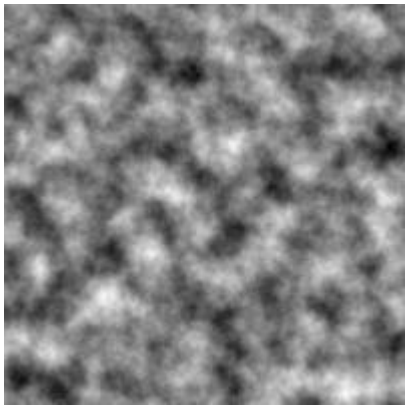


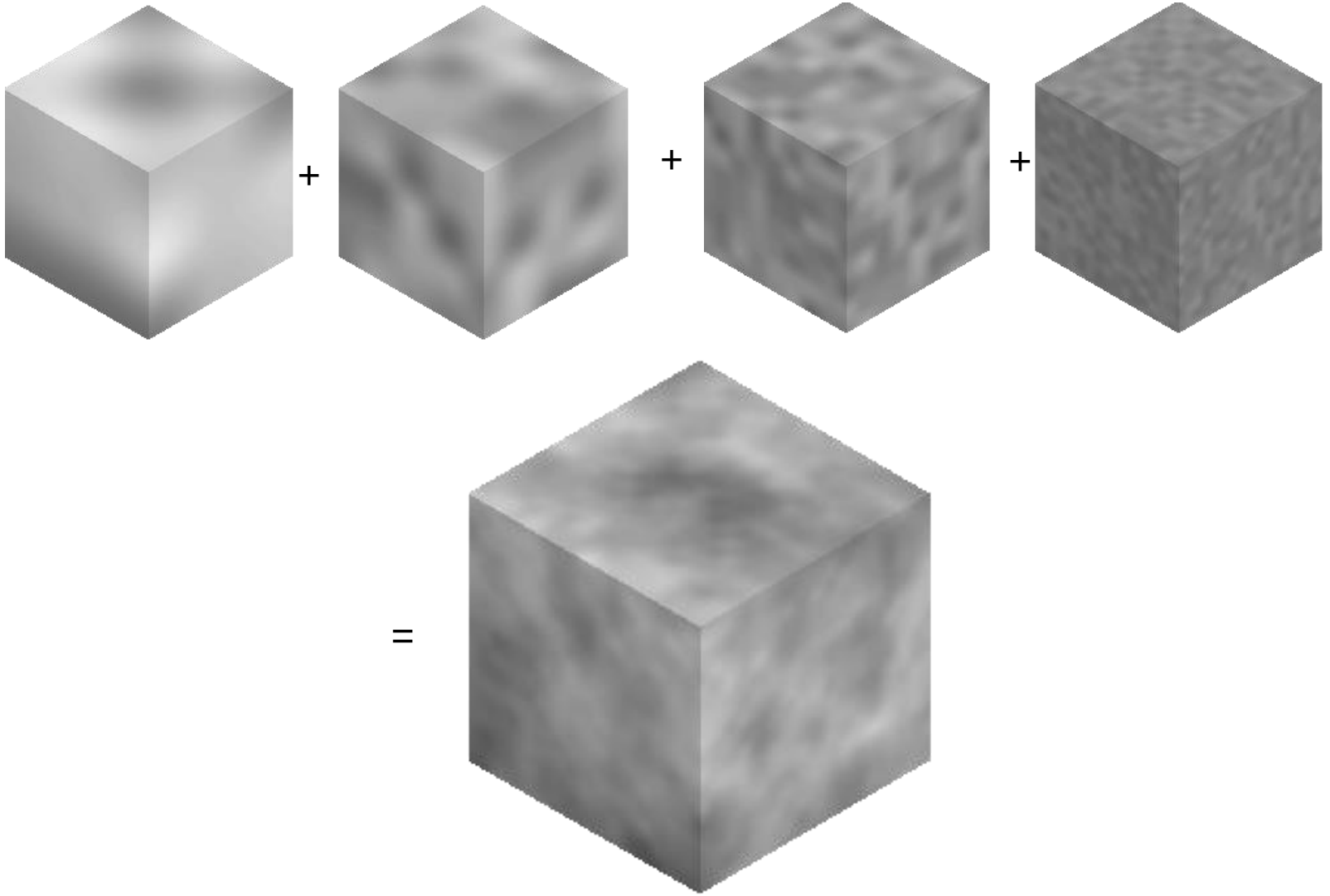
3 (8)

4 (16)

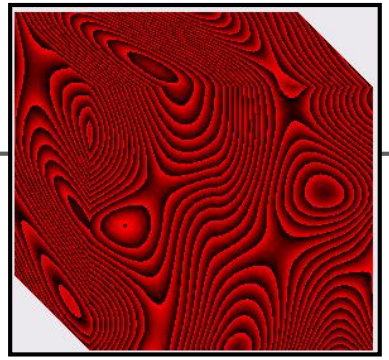


Sum





$$Turbolence() = \sum_{i=0}^{N} \frac{Noise(p \cdot b^i)}{a^i}$$



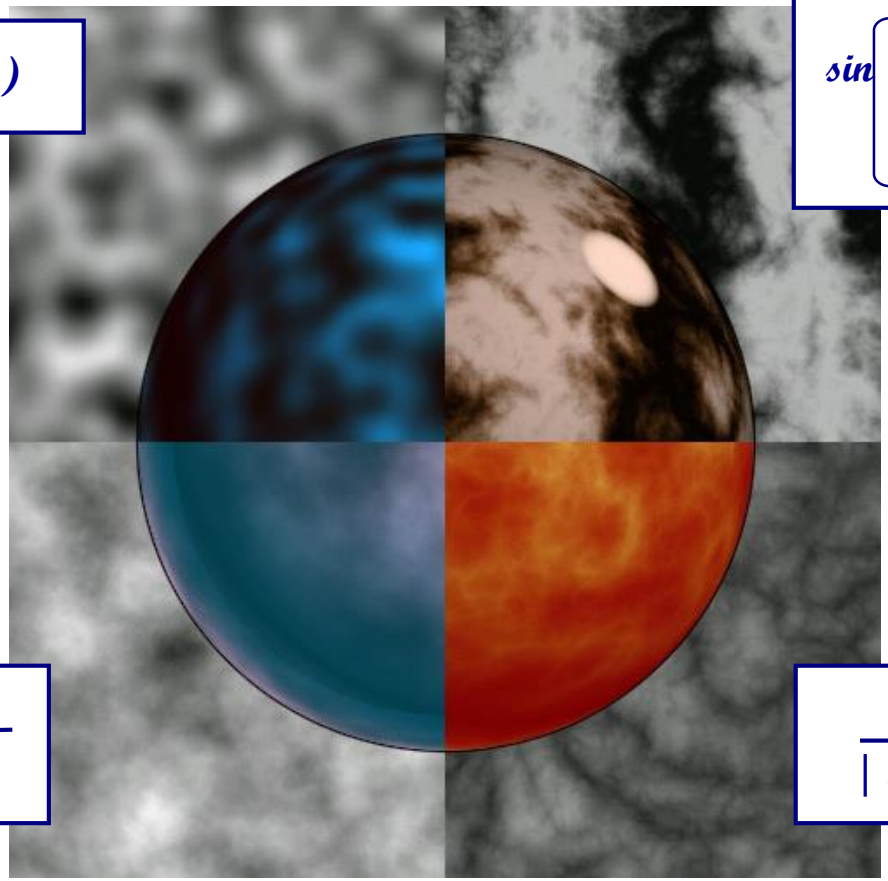
```
double Turbolence(double x,double y,double z,
                  double a,double b,int n)
{  int i;
   double val,  sum = 0;
   double p[3],  scale = 1;
   p[0] = x;  p[1] = y; p[2] = z;
   for (i=0;i<n;i++)
       {  val = noise3(p);
          sum += val / scale;
          scale *= a;
          p[0] *= b;
          p[1] *= b;
          p[2] *= b; }
   return(sum);
}
```

Perlin Noise – Examples

$Turbolence( )$

$\sin\left(x + \frac{1}{|Turbolence|}\right)$

To get more advanced effects, you **need to combine noise at different frequencies**.



$\frac{1}{Turbolence( )}$

$\frac{1}{|Turbolence( )|}$

Noise For Solid Textures
MARBLE

```
color = sin(point + A*turbulence(point))
```

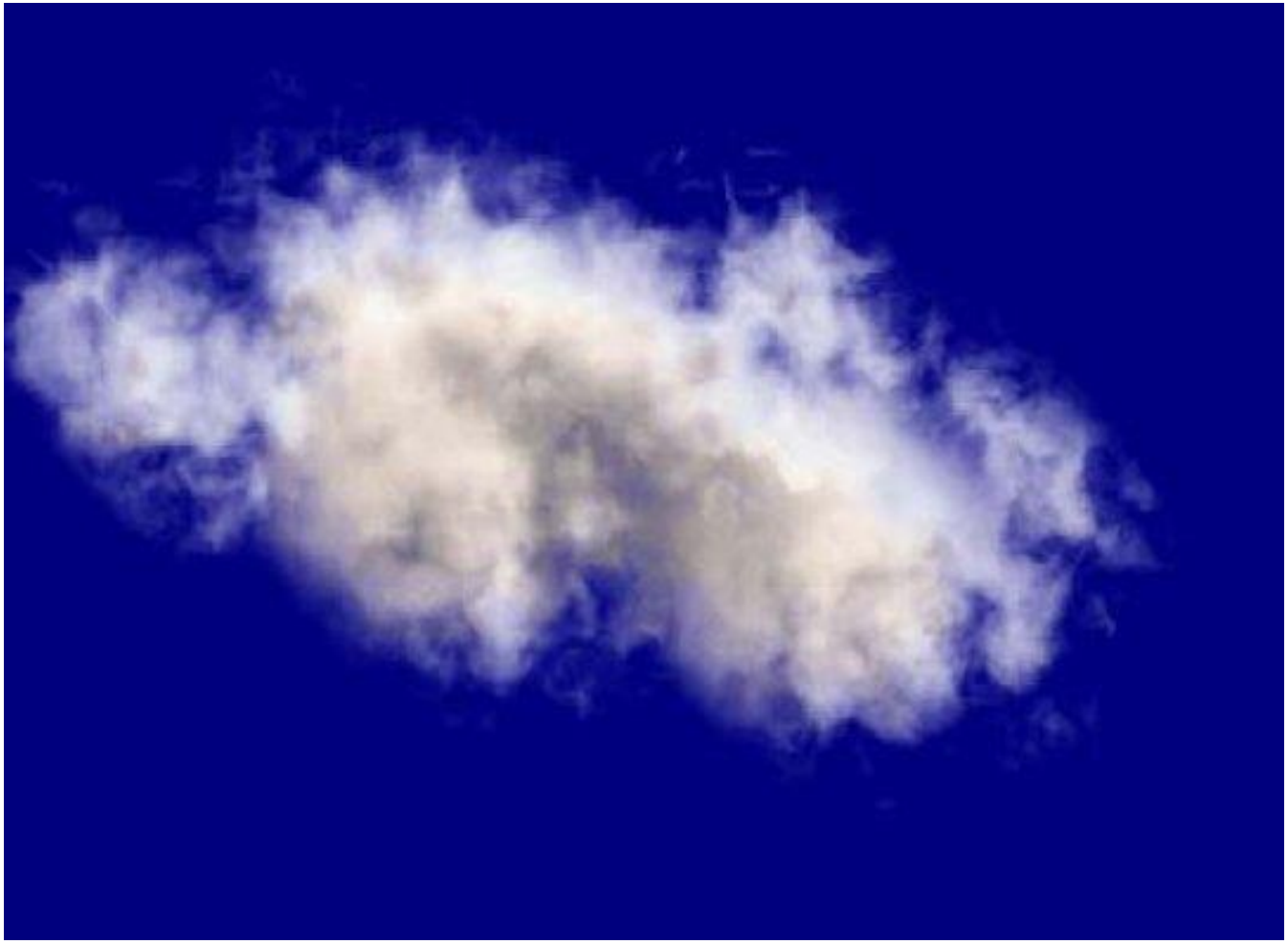
Use a sine function to create the stripes

the amplitude (A) of the turbulence controls the distortion of the veins

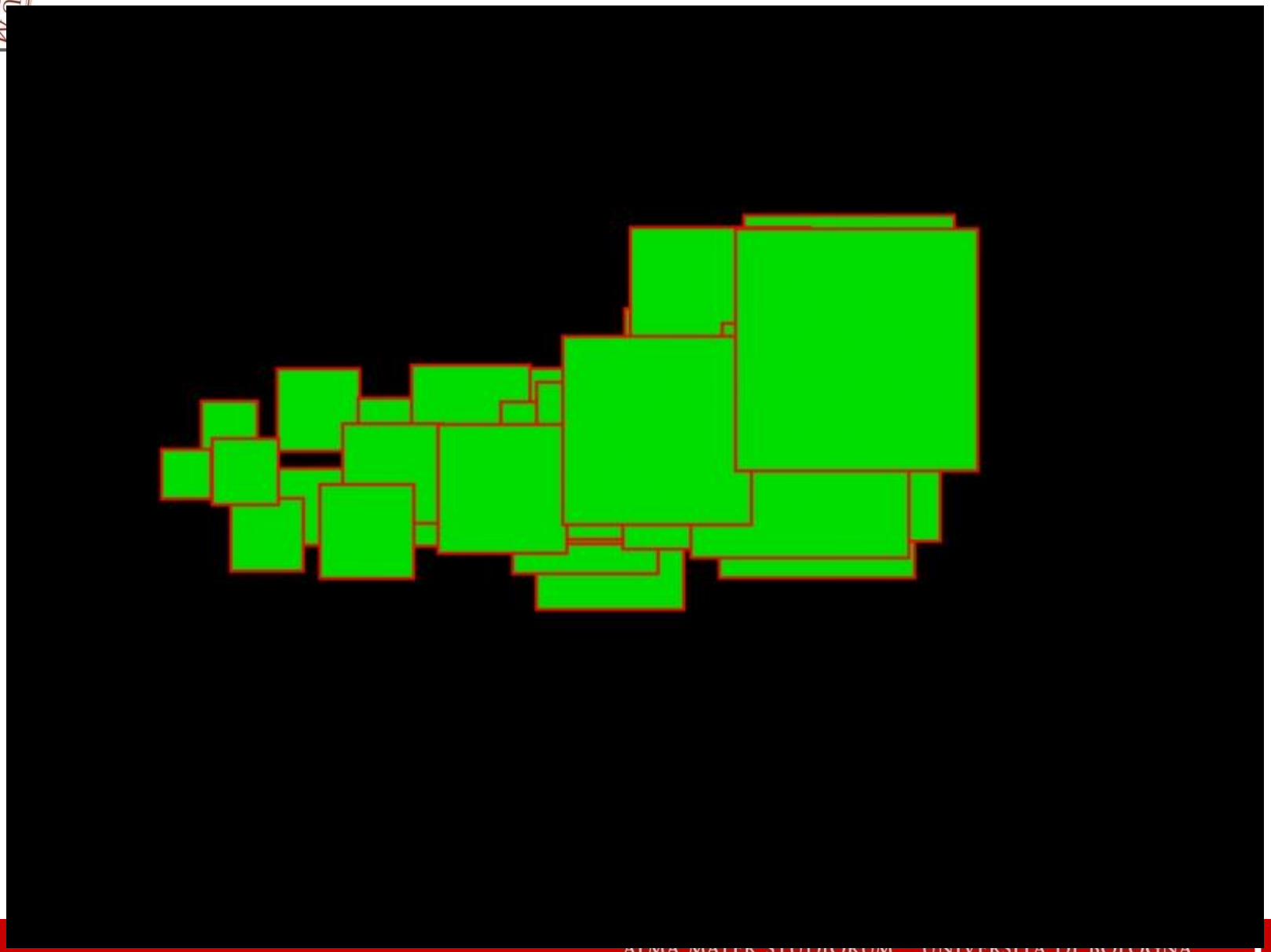
Ken Perlin, 1985

Example : create a cloud

Volumes that enclose some space

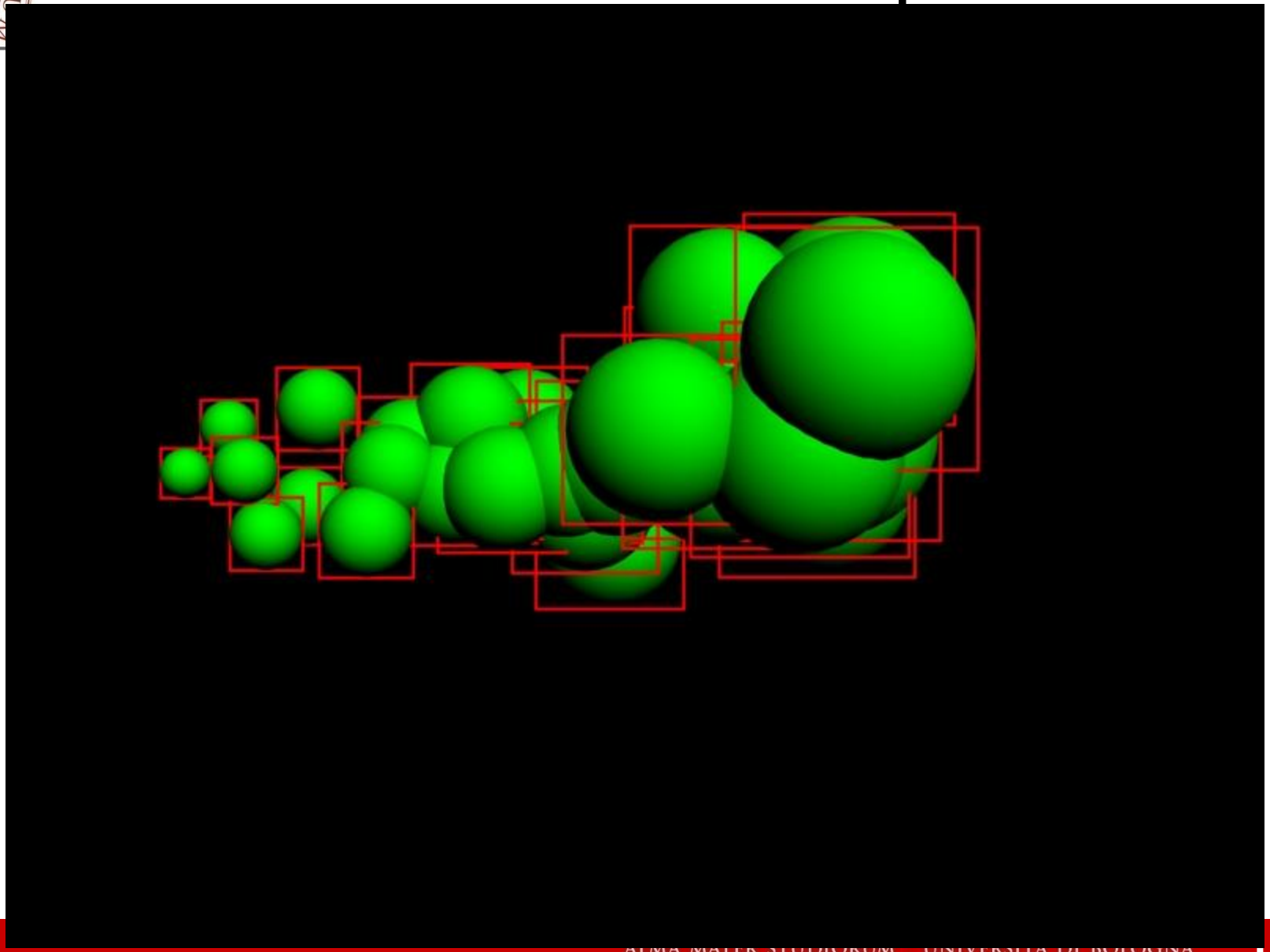


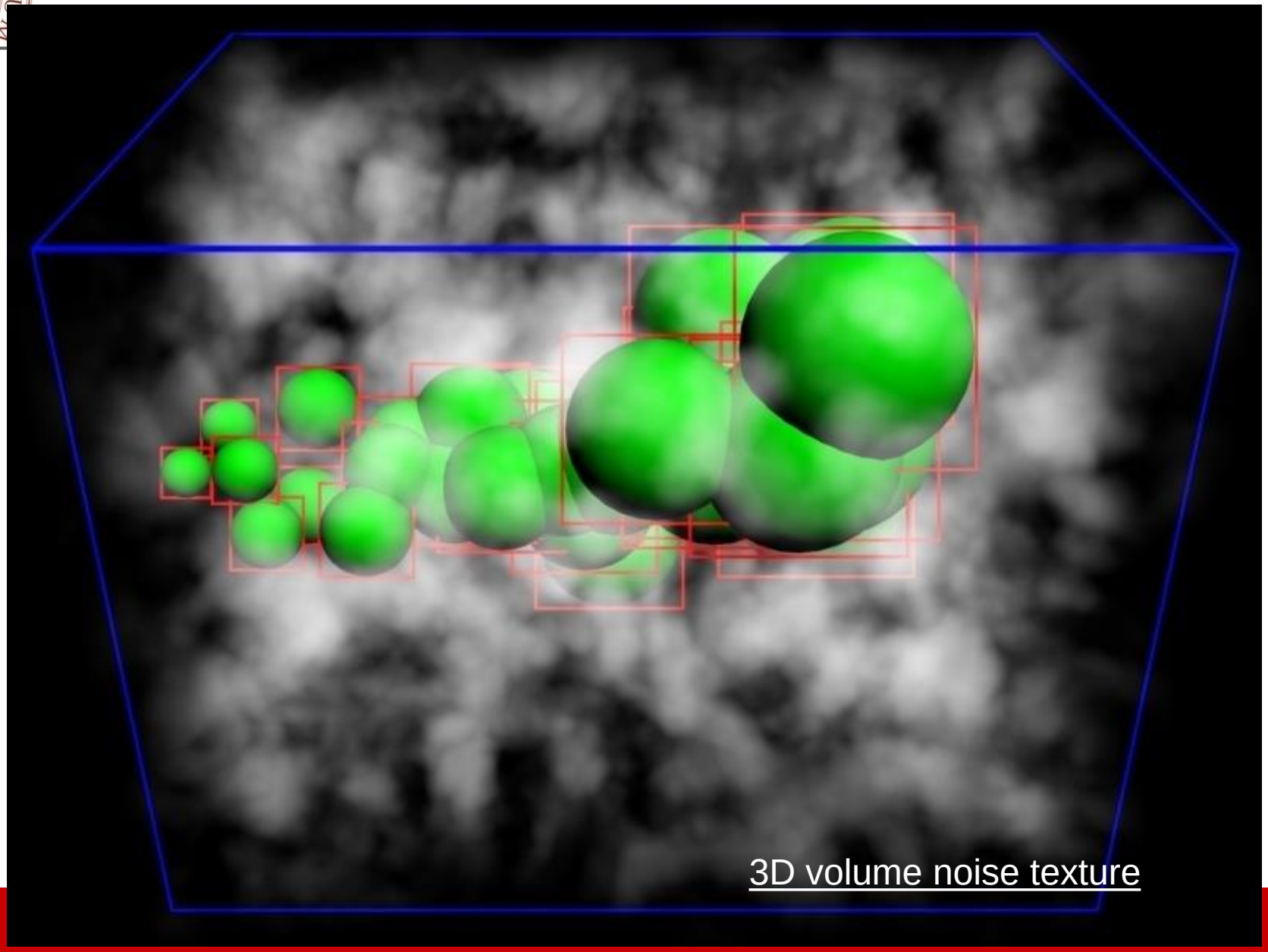
Standard 2D Billboards

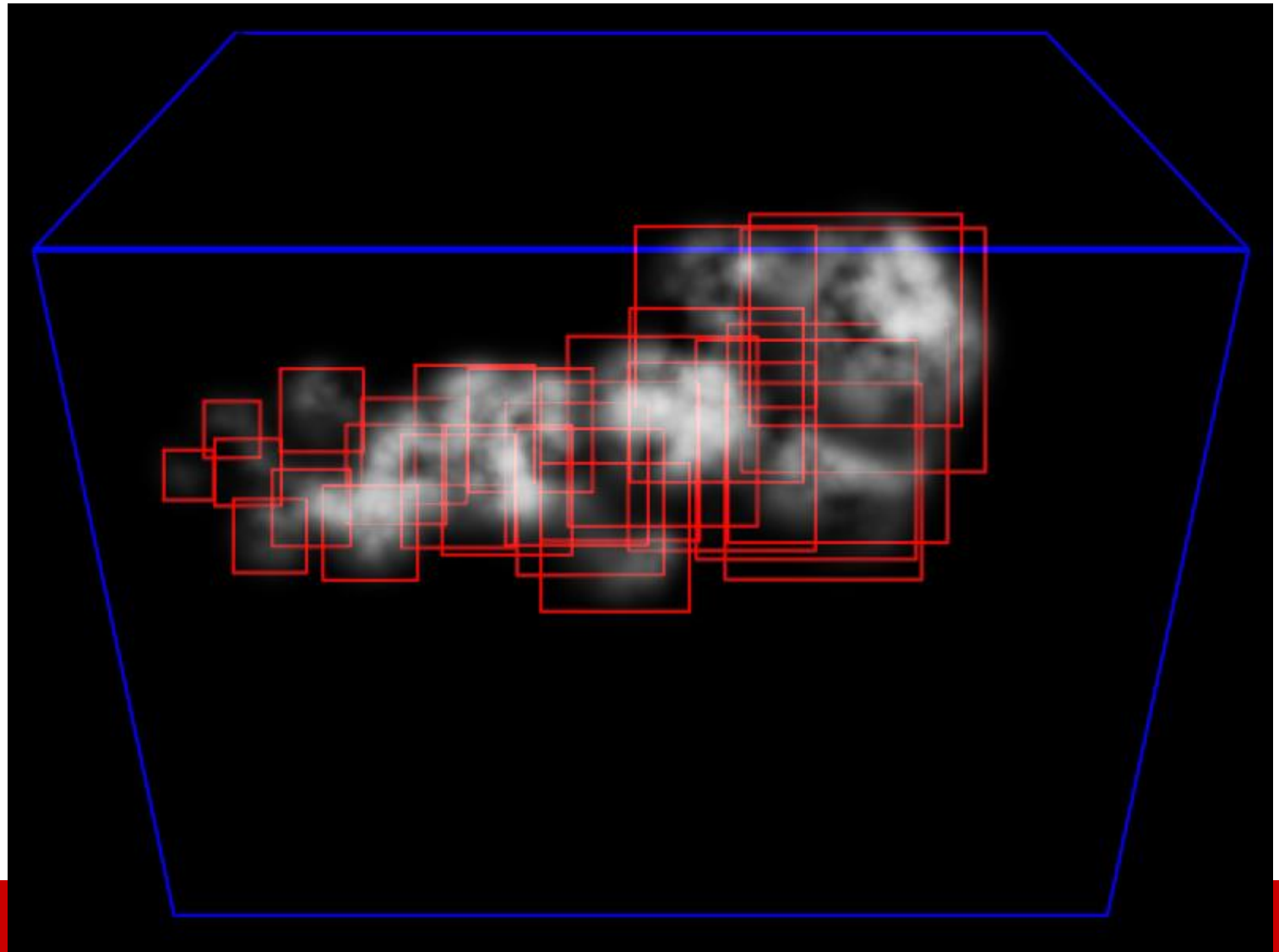


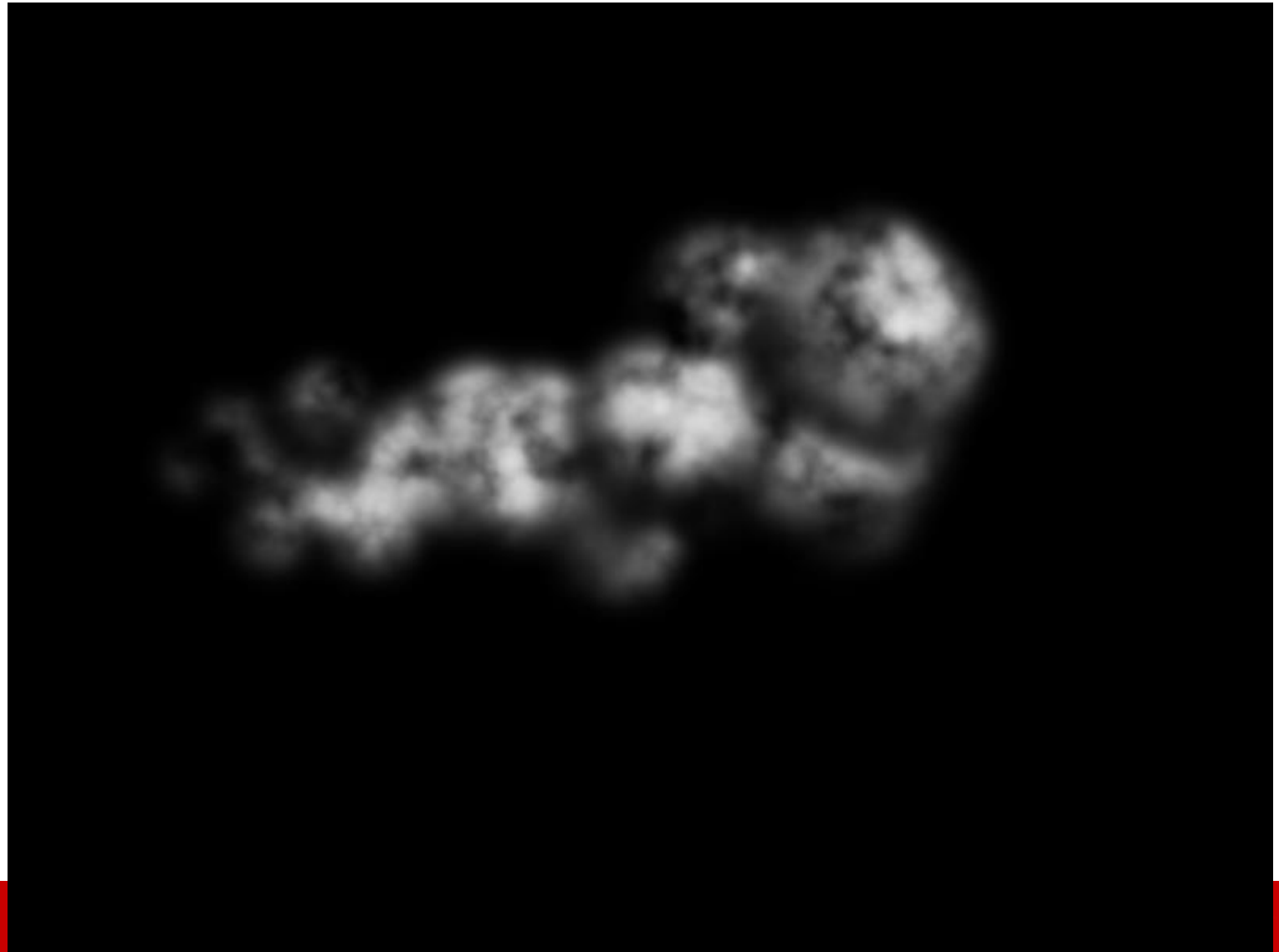
Think of Them as Guidelines

That Circumscribe Spheres

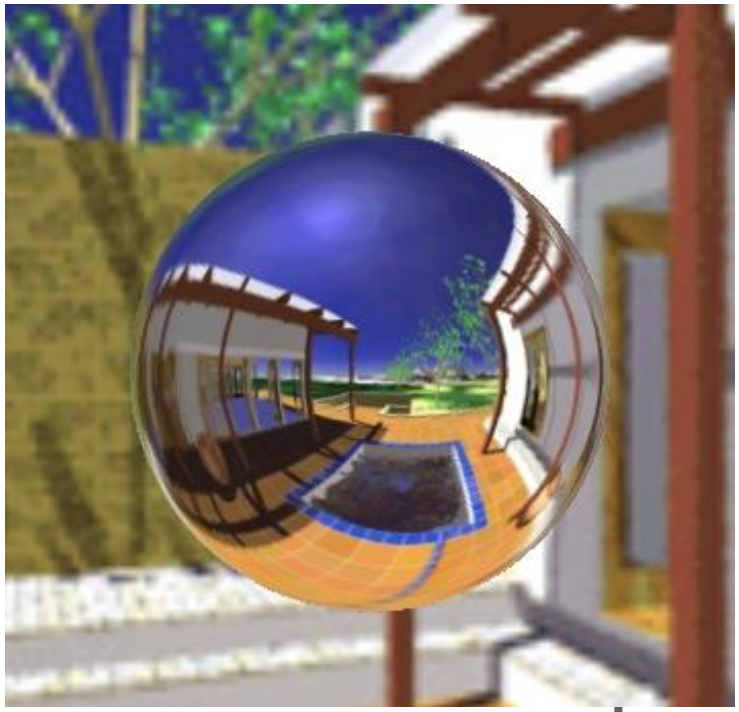



3D volume noise texture

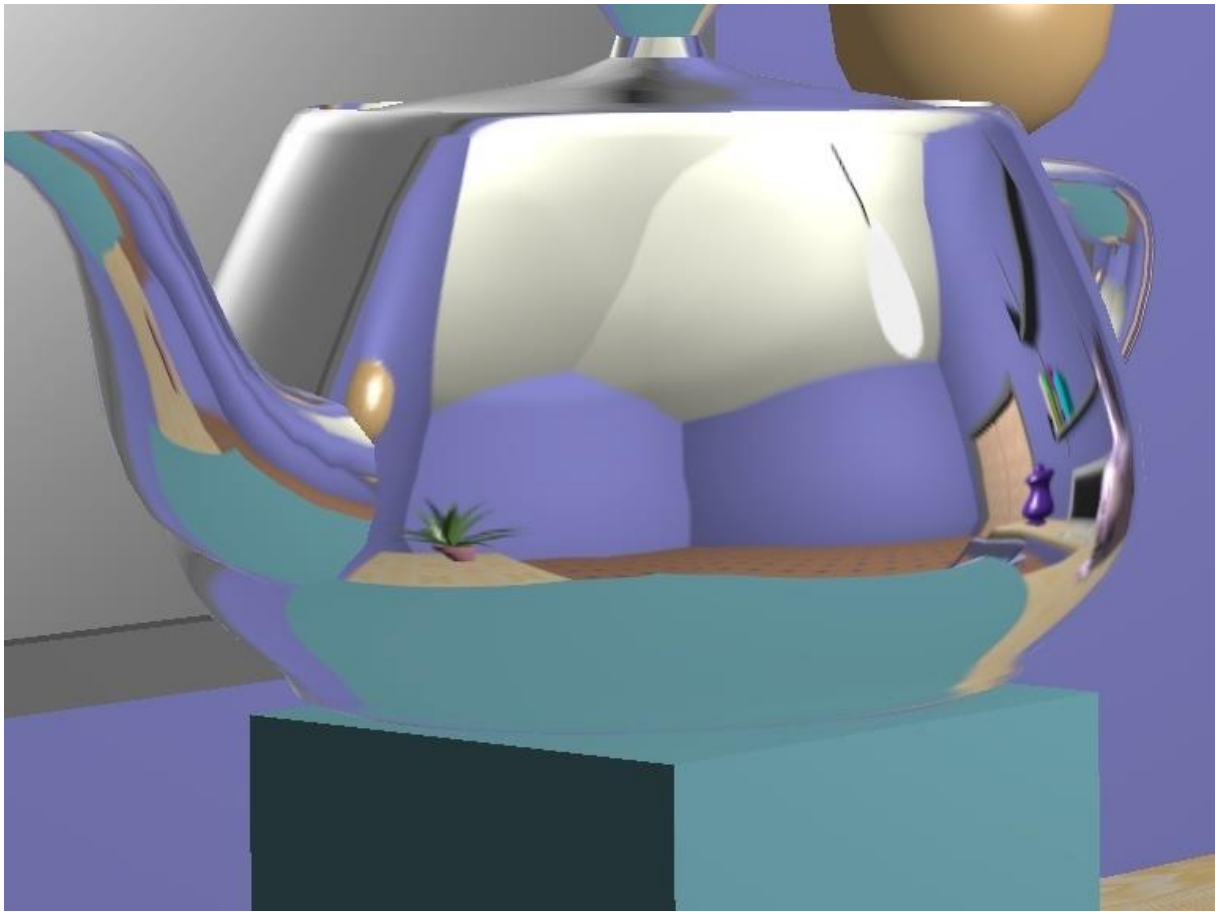




- Normal/Bump Mapping
- Procedural Texturing
- **Environment Mapping**



Environment Mapping

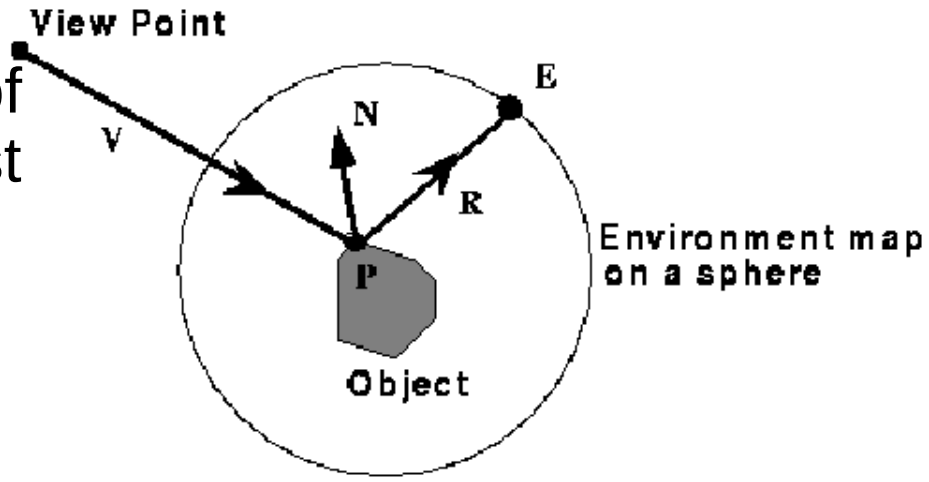


Environment mapping produces reflections on shiny objects.

Use texture to represent reflected color from the environment

Creating an Environment Map

The map should contain a view of the world with the point of interest on the object as the eye

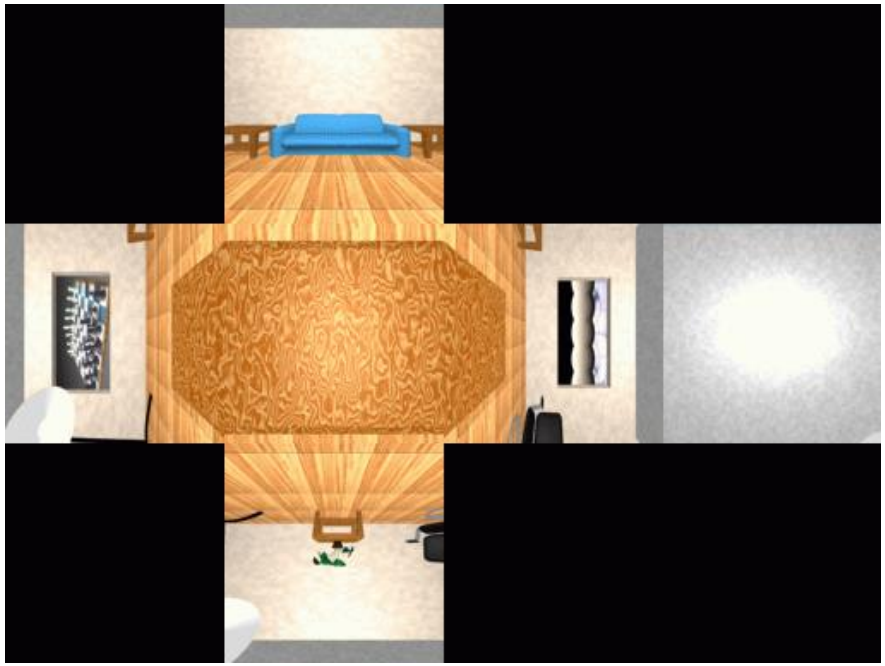


- STEP 1) PREPROCESSING
  creating an **env-map**
  rendering of the scene without the shiny object with the camera at the center of the object in normal object direction; (cube,spherical)

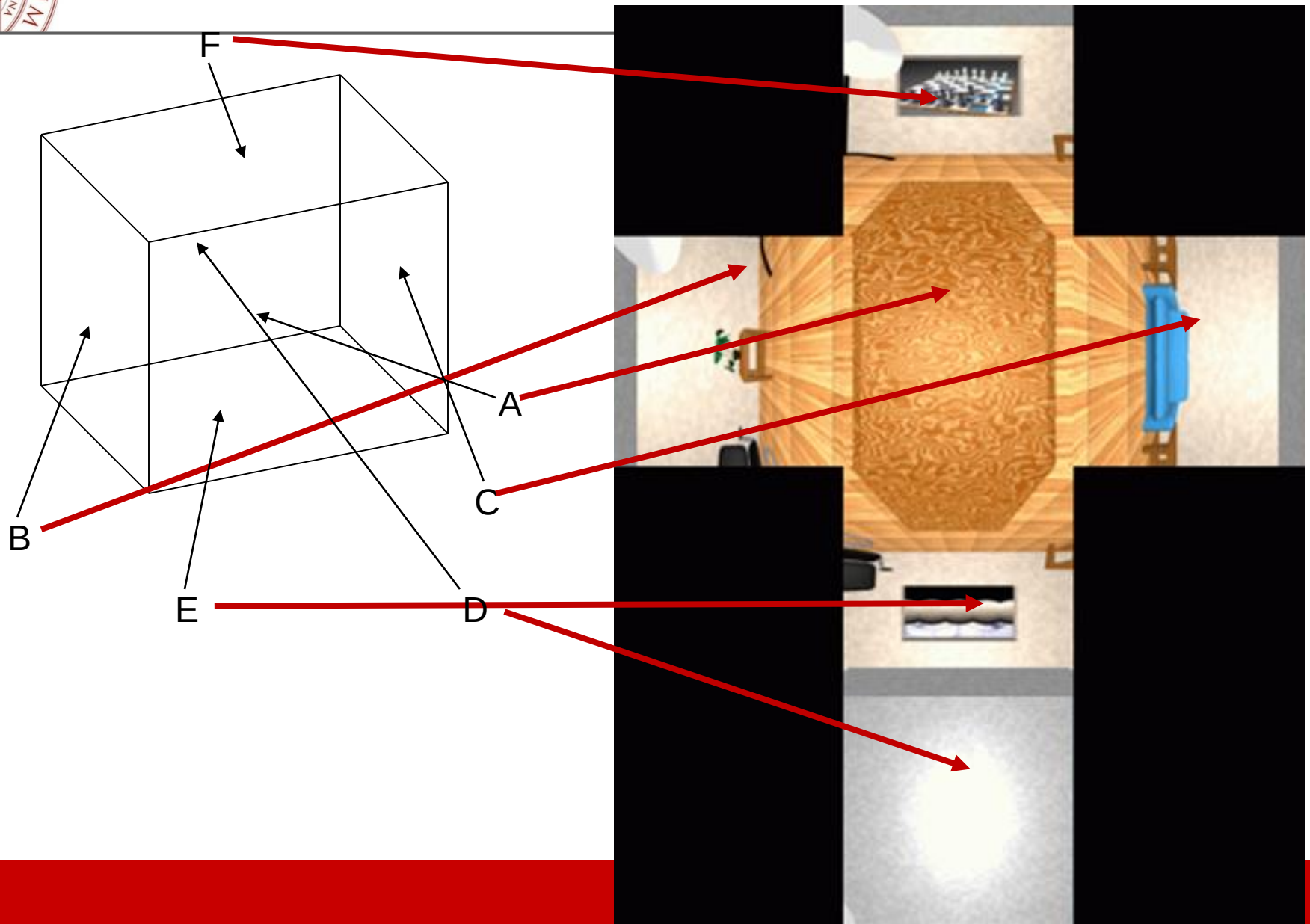
- STEP 2) RENDERING rendering using the texture (env map) indexed by reflection vector

Cube Mapping: STEP 1

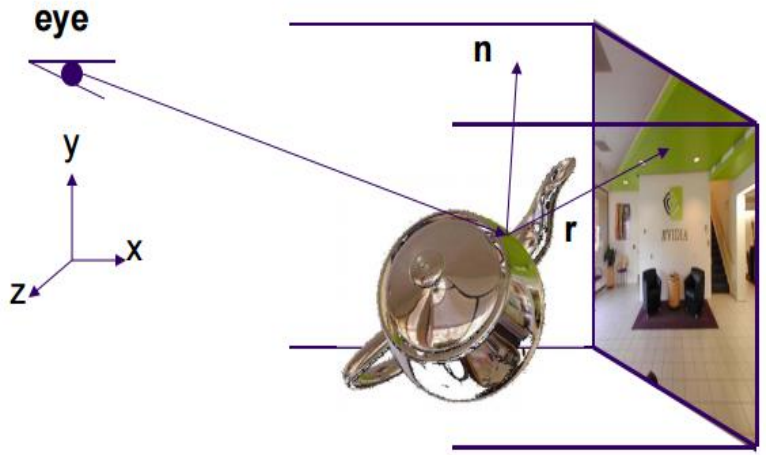
A cubical environment map consists of 6 maps that cover the cube.
Use 6 renderings with the camera at the object center towards a different direction.



Cube Mapping – Greene ‘86



Cube Mapping: STEP 2



- For a reflected ray R:

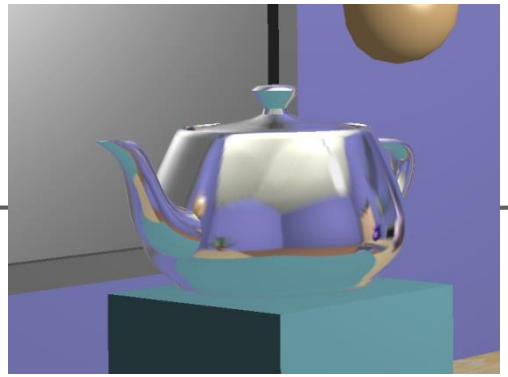
  Select the cube face by R

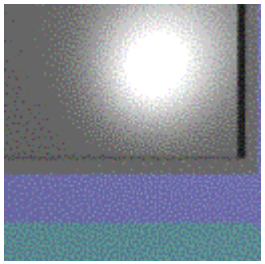
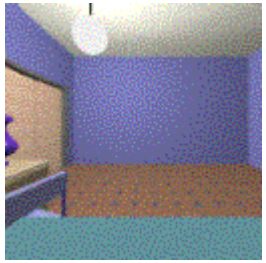
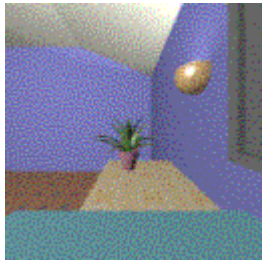
  - max coordinate

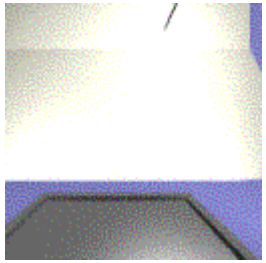
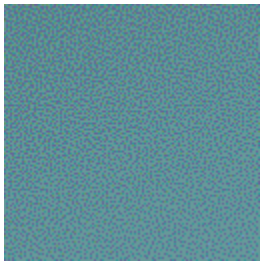
  (i.e. R = (-0.1, 0.4, -0.84) selects face -z)

- The other two coords (normalized by the third coord.) localize the pixel inside the face

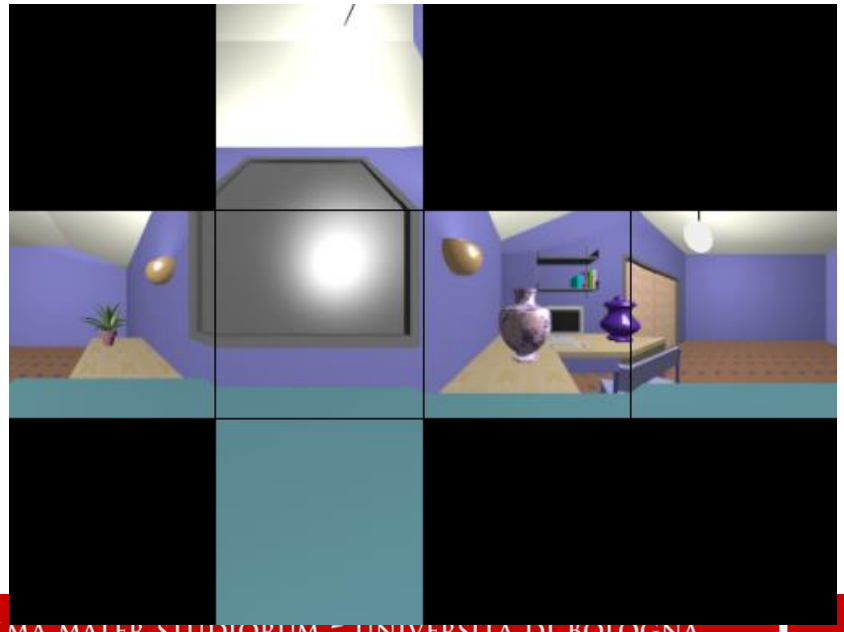
  (i.e.(-0.1,0.4) maps into (0.38,0.80)).



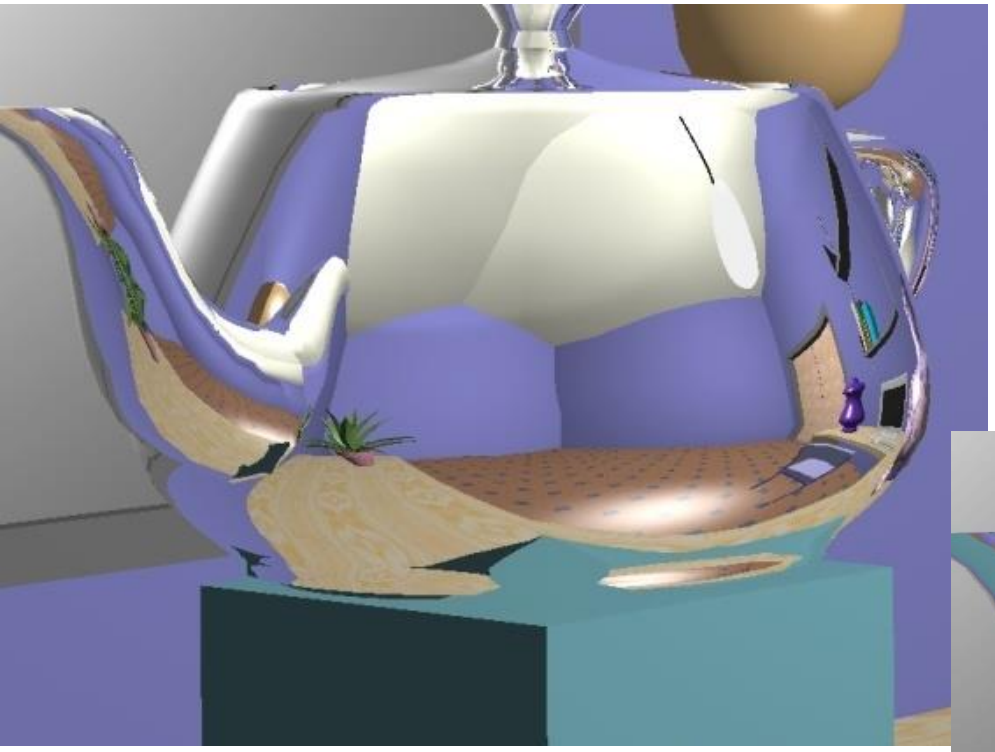




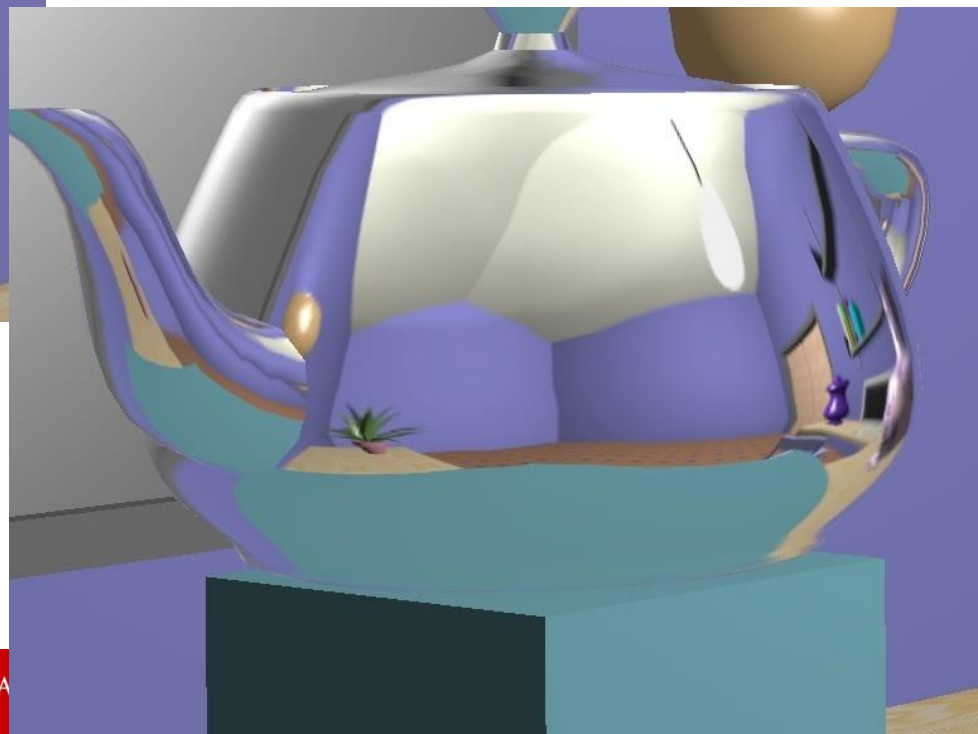



Compute env. Map without teapot and the camera at the center of it. Resolution 128x128 for each image.

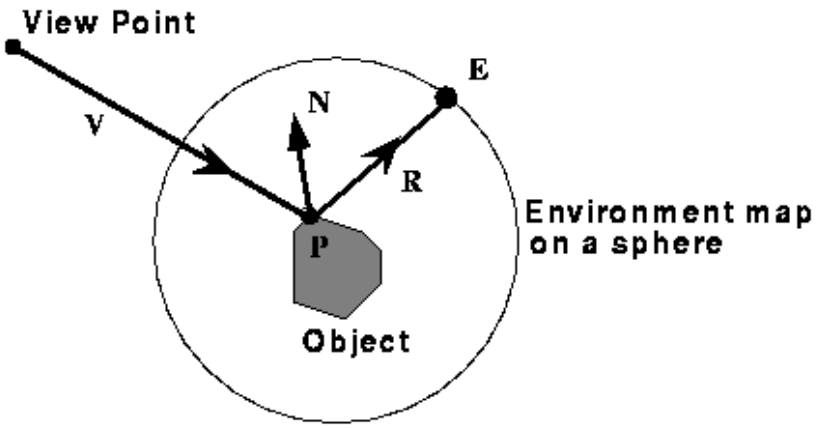


Ray tracing vs. Environment map





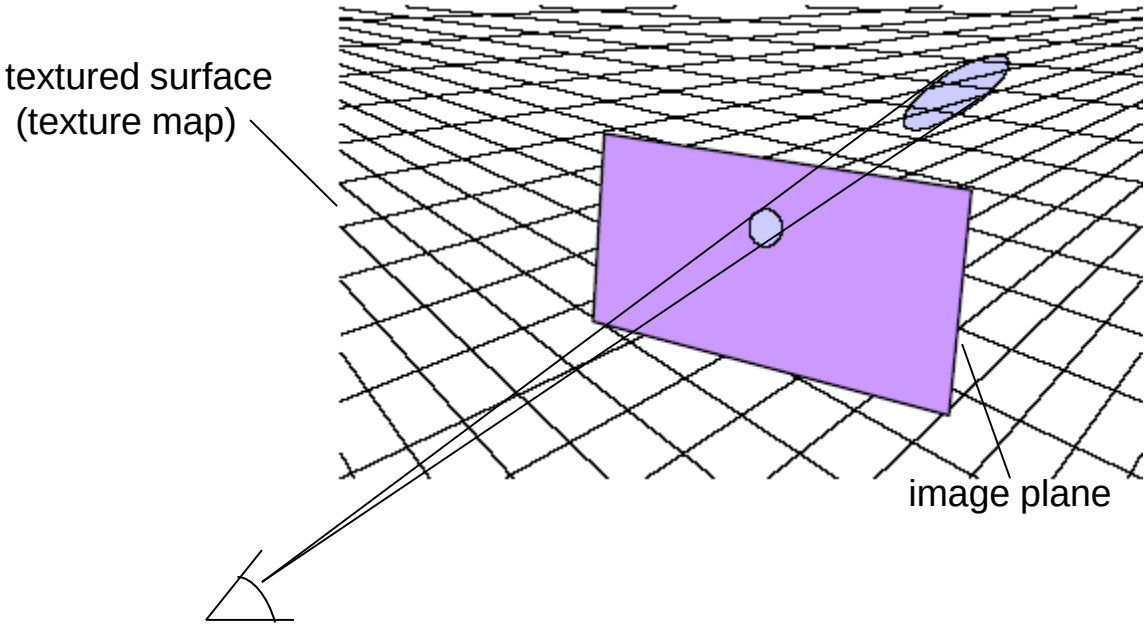
Sphere Mapping



Problems

- Approximation works when objects are far away from the reflective object and the object itself is small
- The object will not reflect itself, thus env. map is not correct for object with concavities
- It is view dependent
- Cost (off-line) for computing the environment map
- An environment map is required for each reflective object in scene;

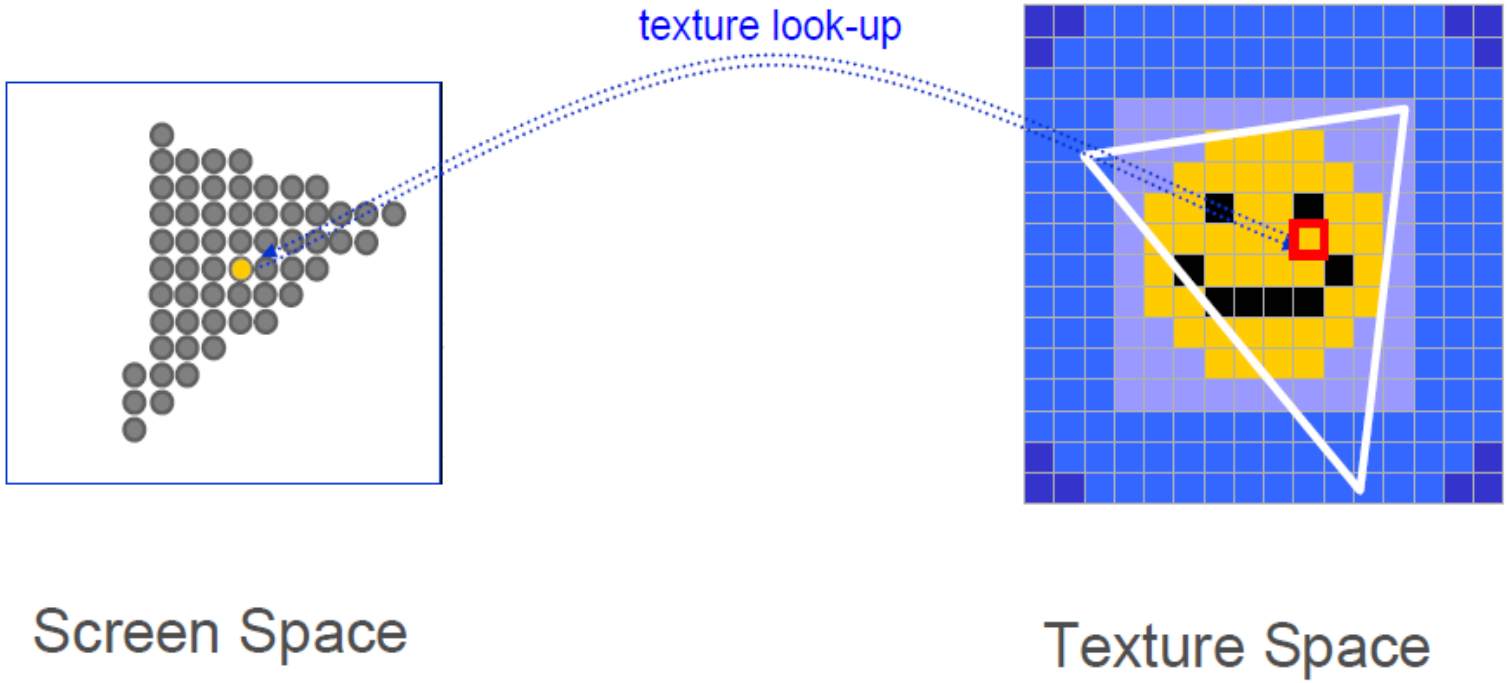
- **Sampling, Texture Filtering**
- Aliasing



How to map the texture area seen through the pixel window to a single pixel value?

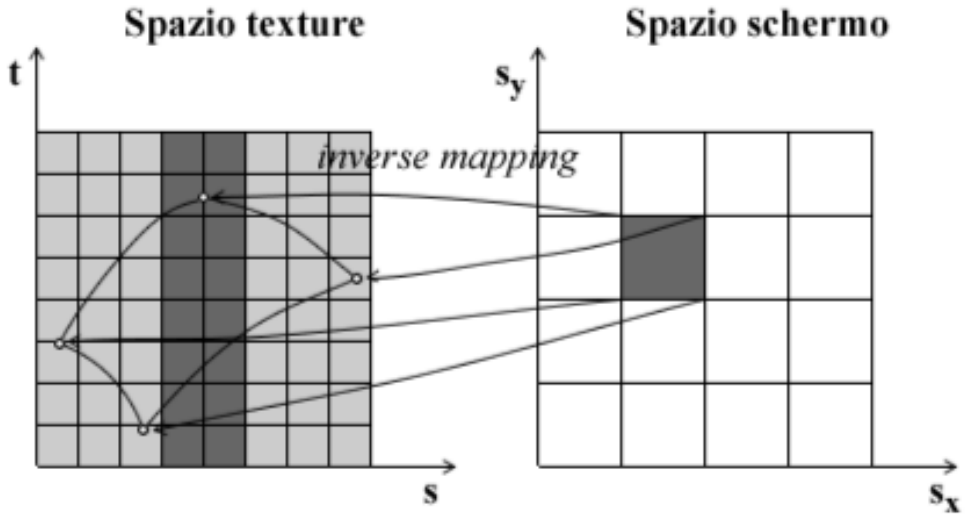
Sampling Texture Maps

When texture mapping it is rare that the screen-space sampling density matches the sampling density of the texture.



Sampling Texture Maps

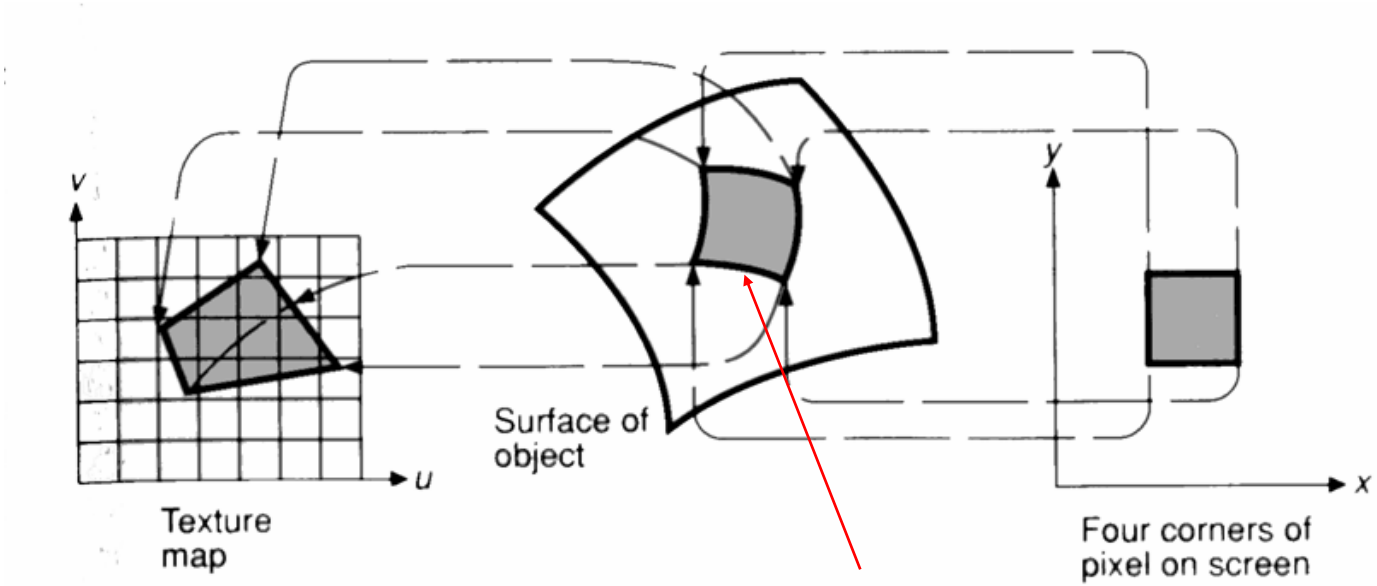
- A single screen space pixel (square area) might correspond to many pixels in the texture image (curve quadrilateral) (pre-image of the pixel)



How to map the texture area seen through the pixel window to a single pixel value?

Area sampling

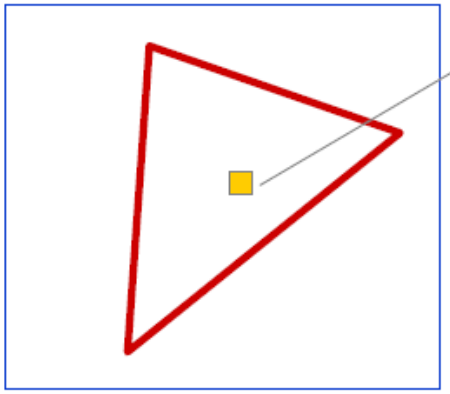
- Consider the 4 vertices of the pixel (the *preimage* of pixel is curved)
- Compute a spatial integration over the extent of the pixel
- This is equivalent to convolving the texture with a filter kernel centered at the sample (i.e., pixel center)! to compute the pixel color



TOO EXPENSIVE!!!

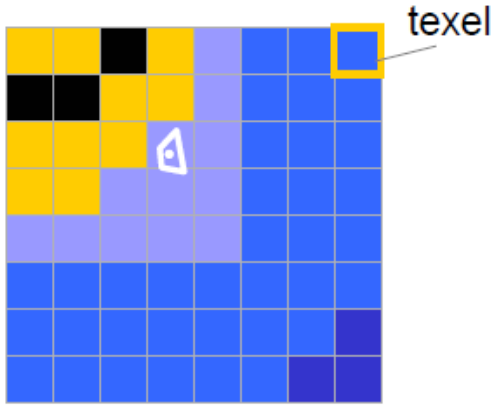
Two practical cases....

Two cases



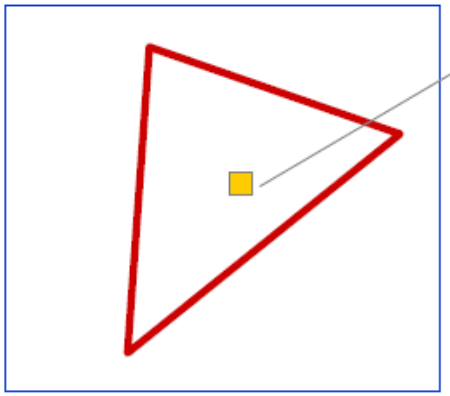
un pixel = meno di un texel

magnification



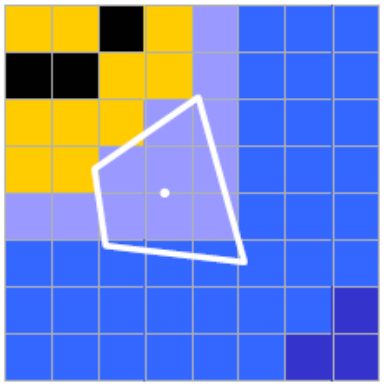
Screen Space

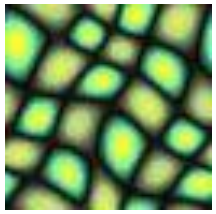
Texture Space



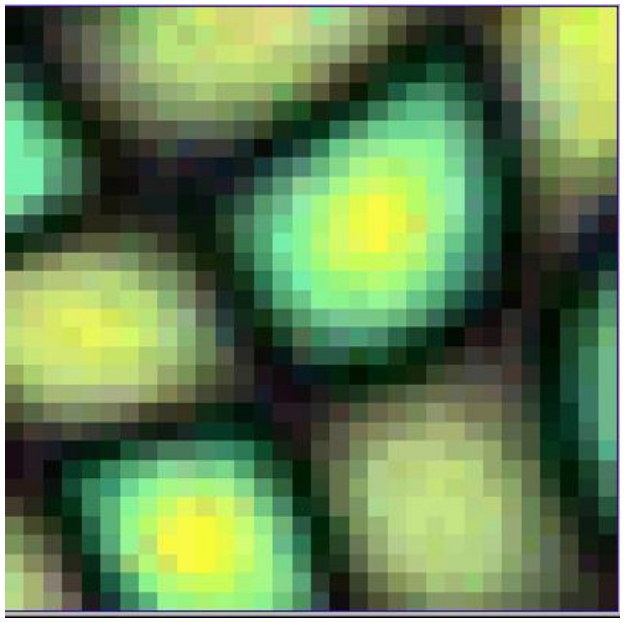
un pixel = più di un texel

minification

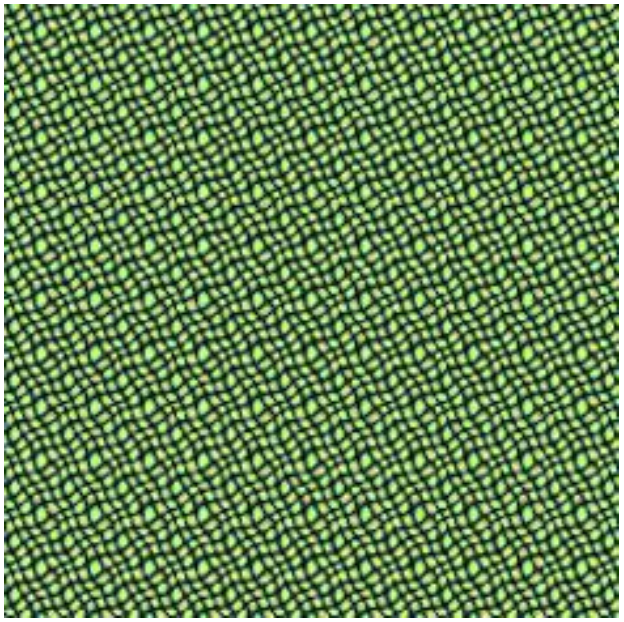




Original image
64x64 pixels

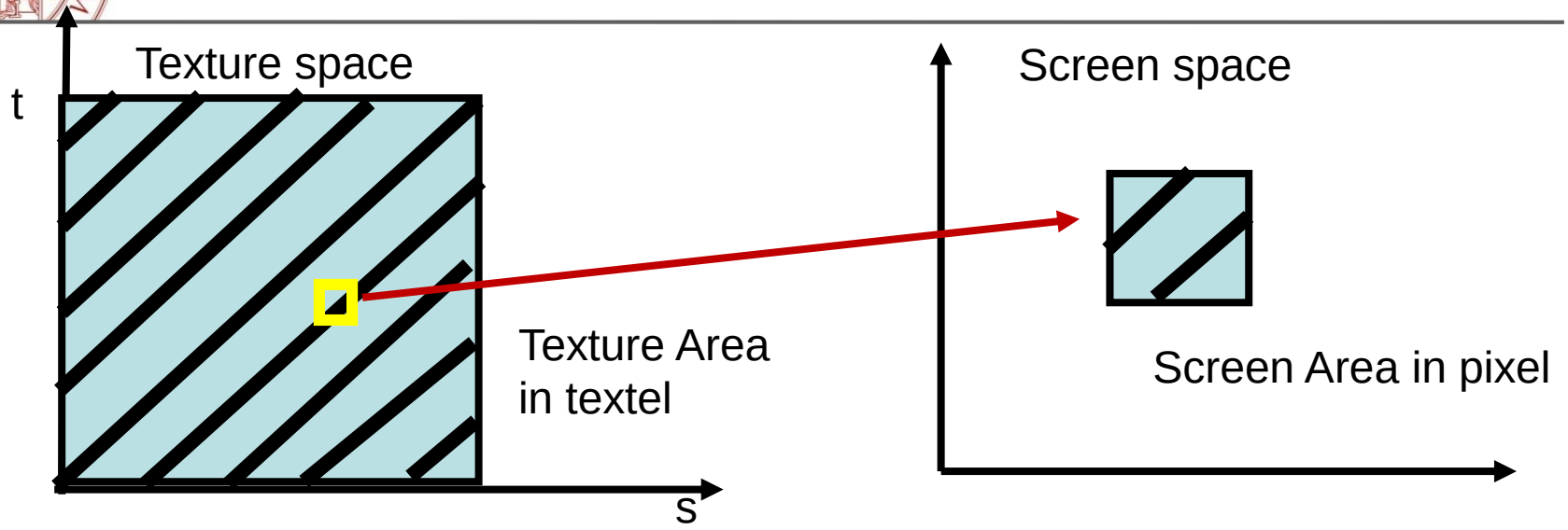


Magnification for Display



Minification for Display

Texture Magnification



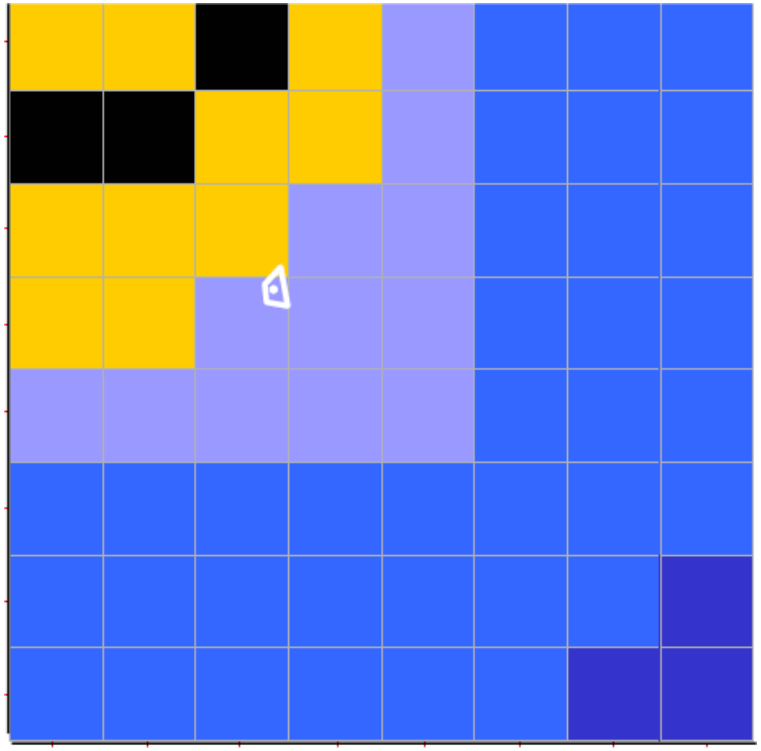
MAGNIFICATION: textures are zoomed on the screen, the pixel have preimages with area less than a texel.

(1 texel/many pixel)

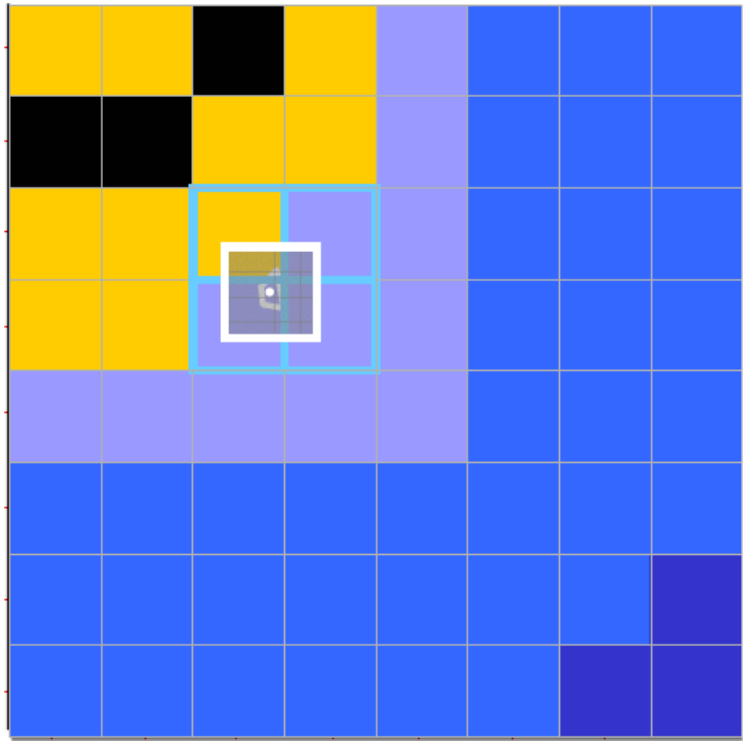
Interpolate in texture space:

nearest neighbor (choose the color of the nearest texel),

Linear interpolation interpolate the closest texels (4 for 2D, 8 for 3D)

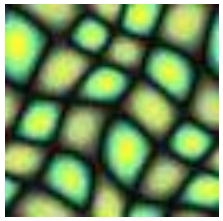


Nearest neighbor

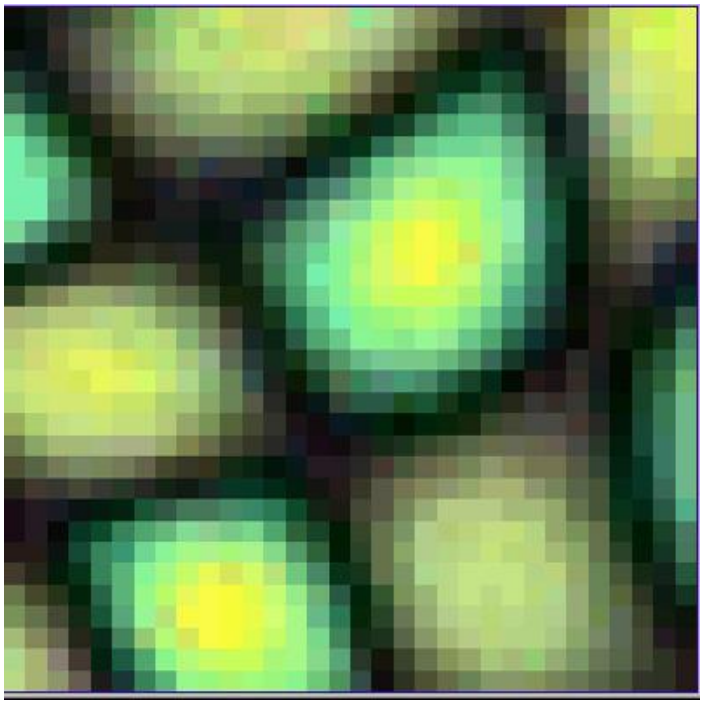


Linear Interpolation

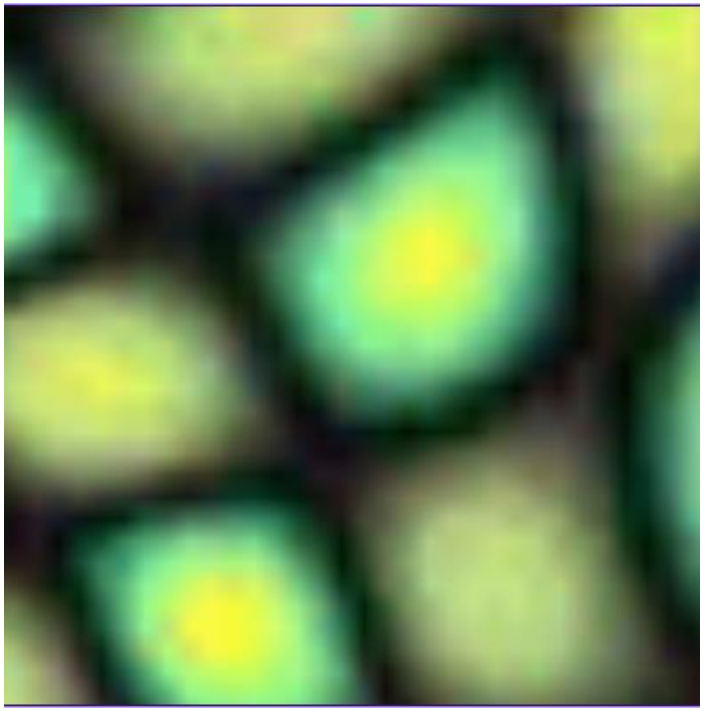
Average of the 4 texel



64x64 pixels
Original Texture

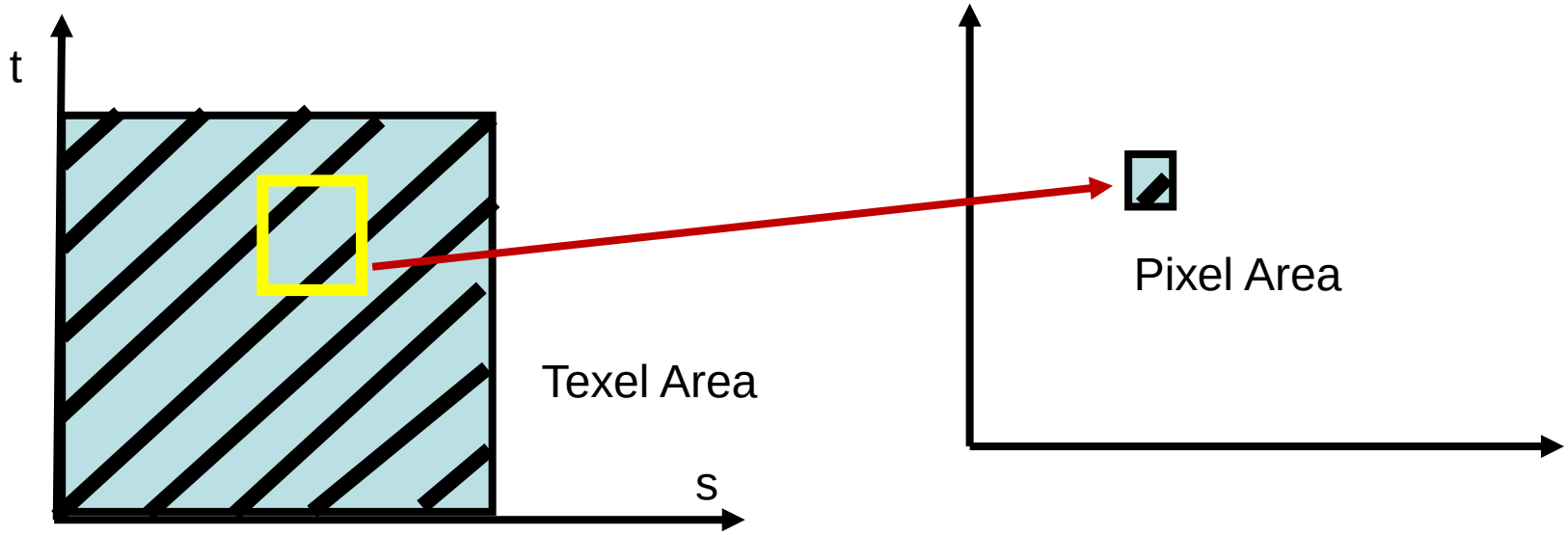


Nearest neighbor



Linear Interpolation

Texture Minification



MINIFICATION:

Texture are smaller in screen space, thus a big preimage in texture space corresponds to a pixel (1 pixel/ many texels)

⇒ **Mipmapping**

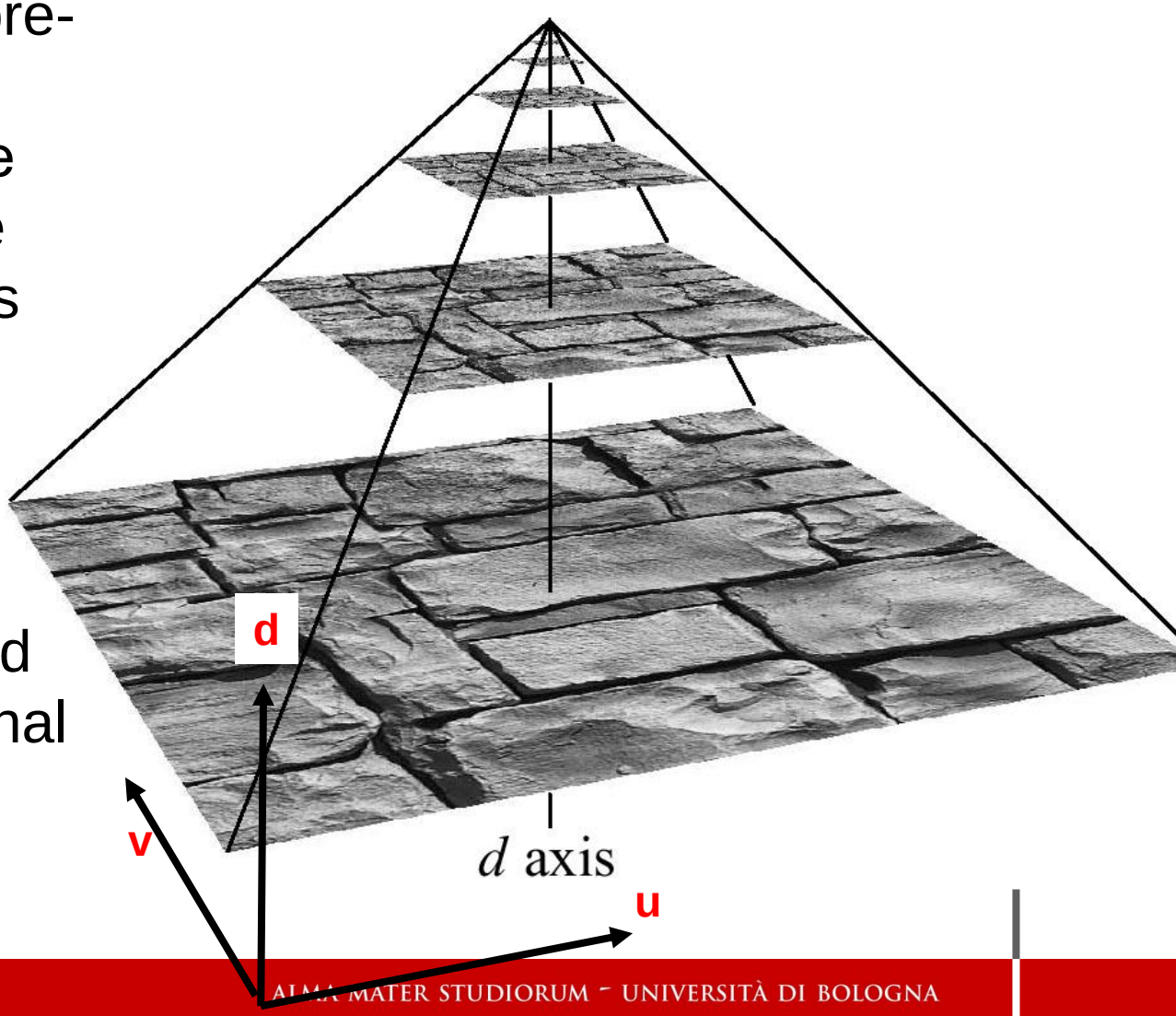
MIP mapping

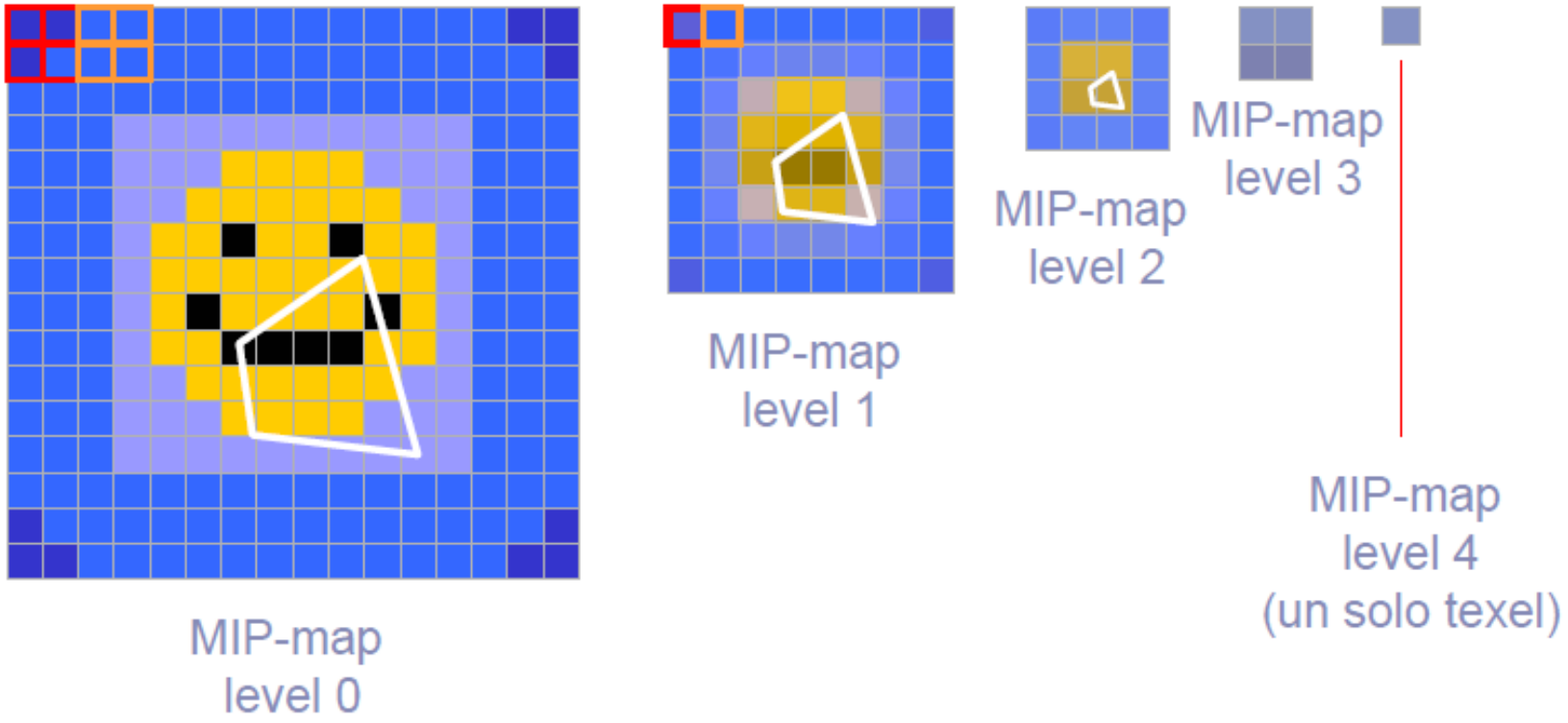
Multum In Parvo = Molte cose in un piccolo spazio

many in a small place

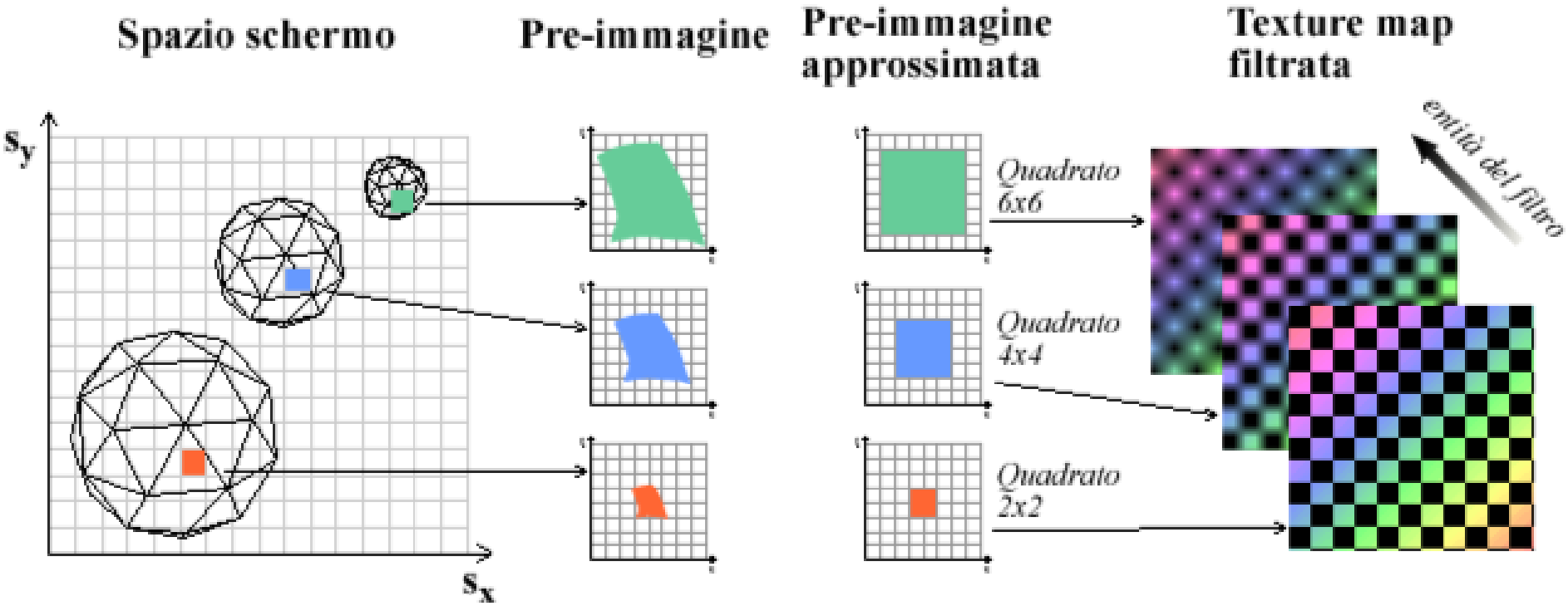
Pre-processing: Store pre-filtered scaled copies (texture “levels”) of the original source texture
Typically, each level is half the size of the previous level (512, 256, 128,...,1).

Each level is computed by averaging the original image (low pass filter)




MIP-map
level 0
MIP-map
level 1
MIP-map
level 2
MIP-map
level 3
MIP-map
level 4
(un solo texel)

MIP mapping


Spazio schermo
Pre-immagine
Pre-immagine approssimata
Texture map filtrata
s_y
s_x
Quadrato 6x6
Quadrato 4x4
Quadrato 2x2
entità del filtro

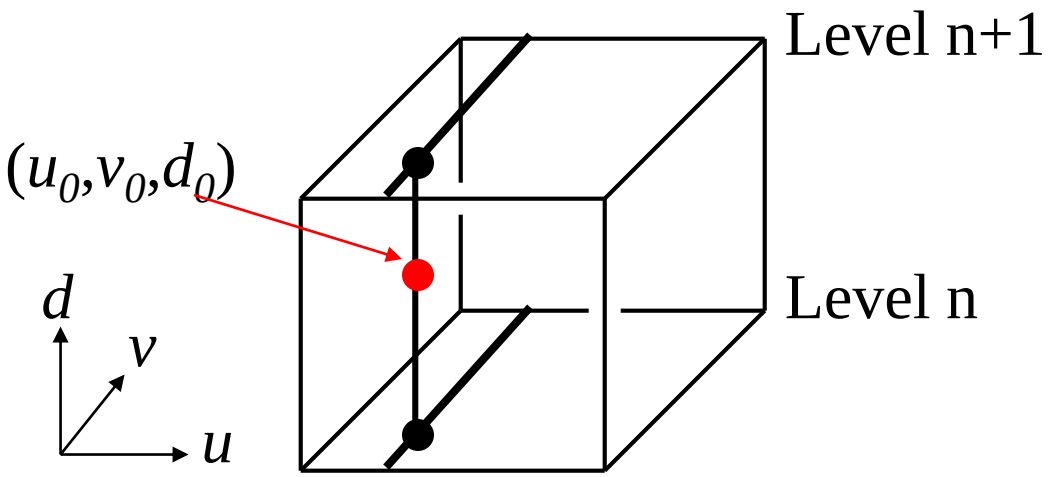
- **Approximated preimage area:**

  axis-aligned rectangle in texture space

  A =texels/pixel
- Define **scalar factor**, **b=sqrt(A)**
- The **mipmap level** to use is: **$d=\log_2 b$**
  - level 0 = max resolution
  - the level number is not necessary an integer

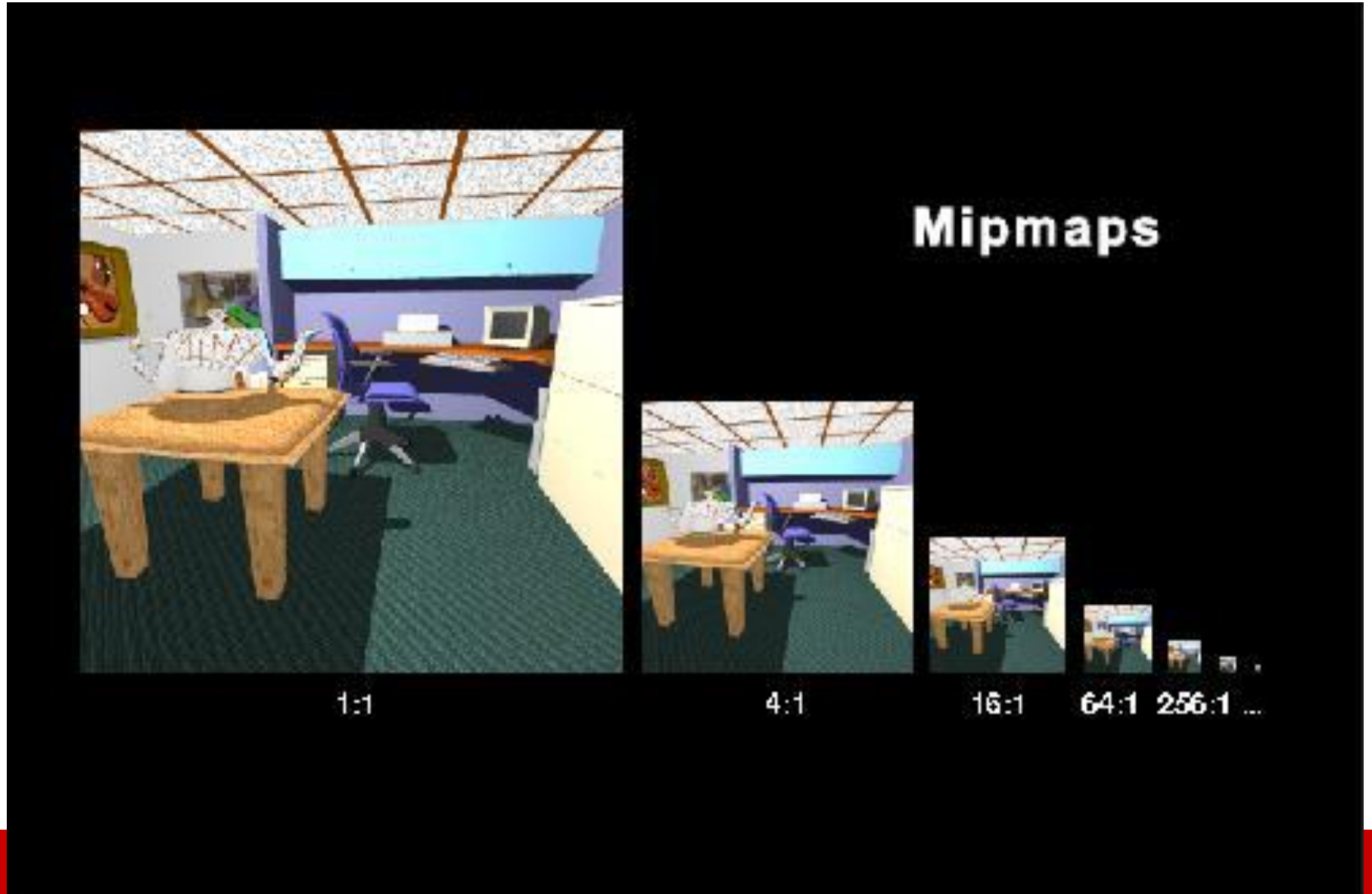
Mip-mapping

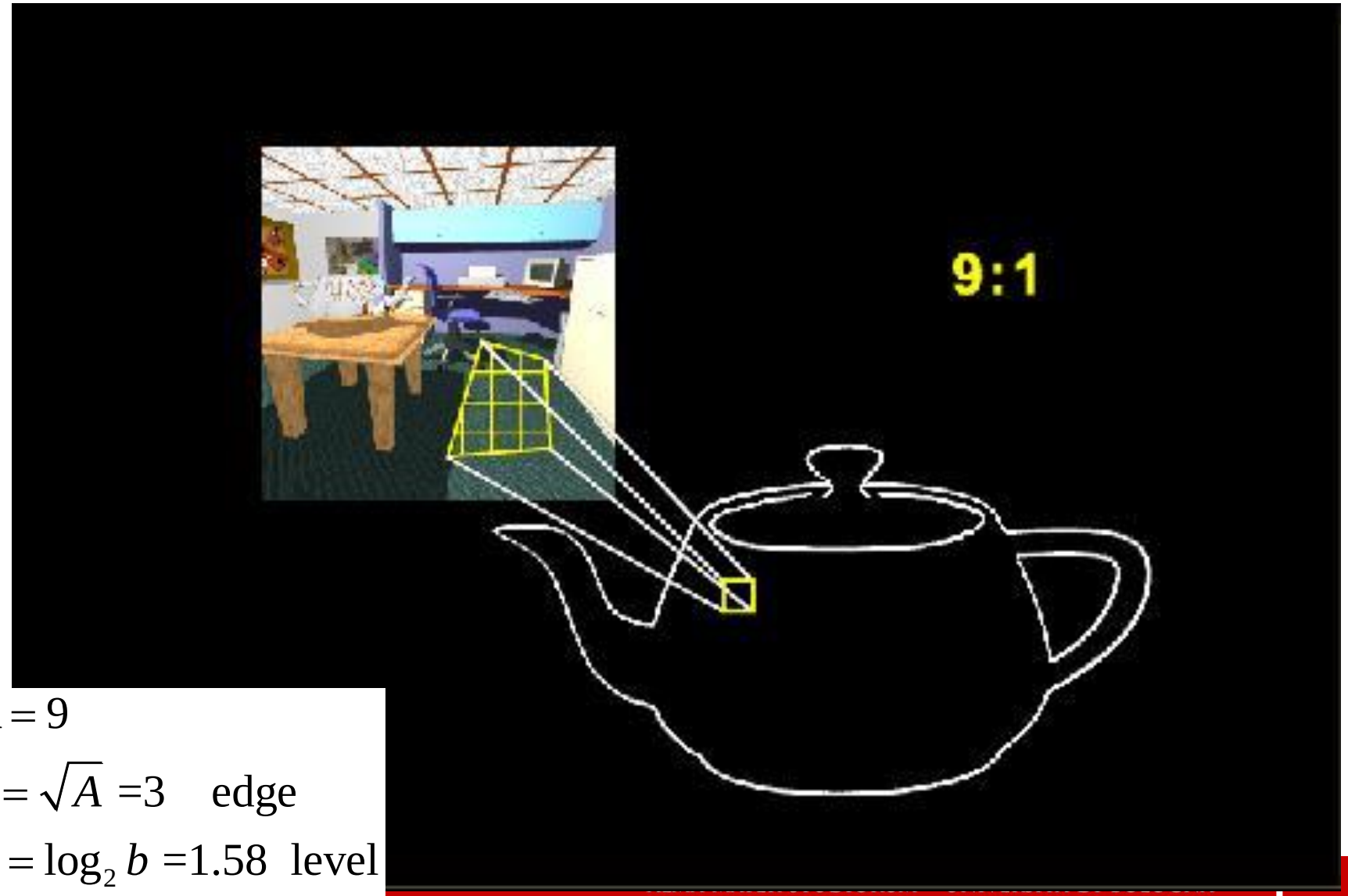
n<d<n+1 – texel : pixel ratio



The final color is given by the trilinear interpolation of the color in the textures n and n+1

Mip-mapping: example
(Williams 1983)

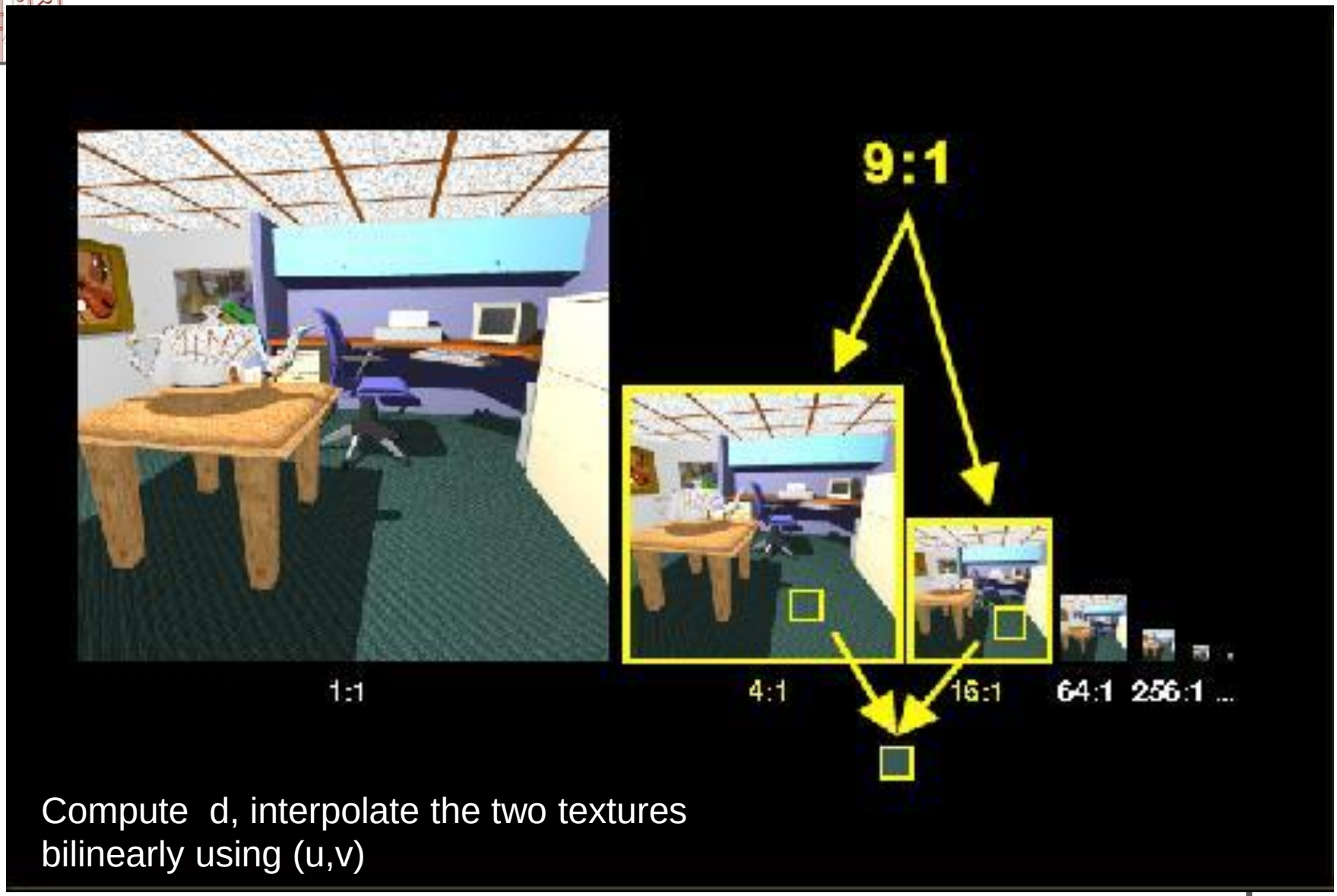




$A = 9$

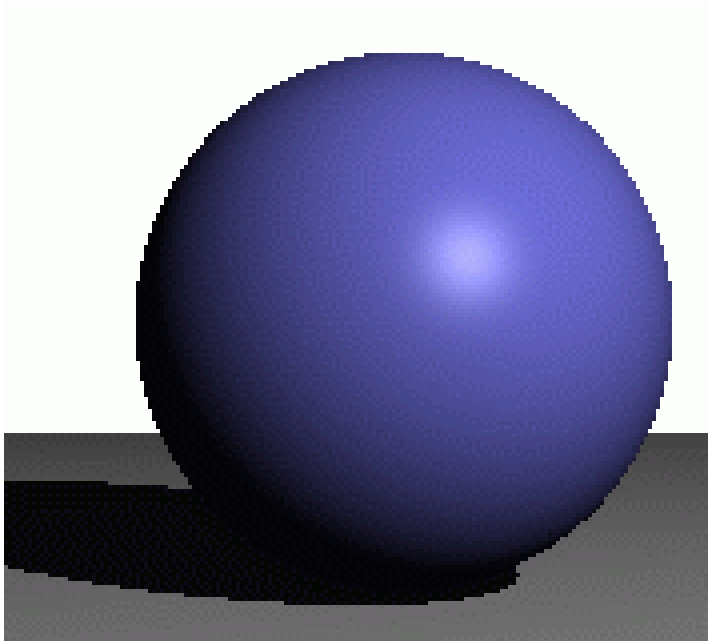
$b = \sqrt{A} = 3$ edge

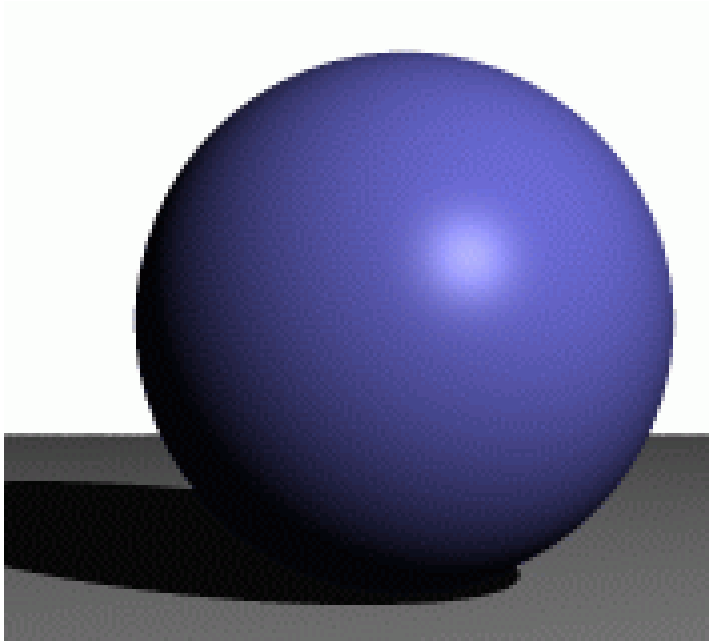
$d = \log_2 b = 1.58$ level



Compute d, interpolate the two textures bilinearly using (u,v)

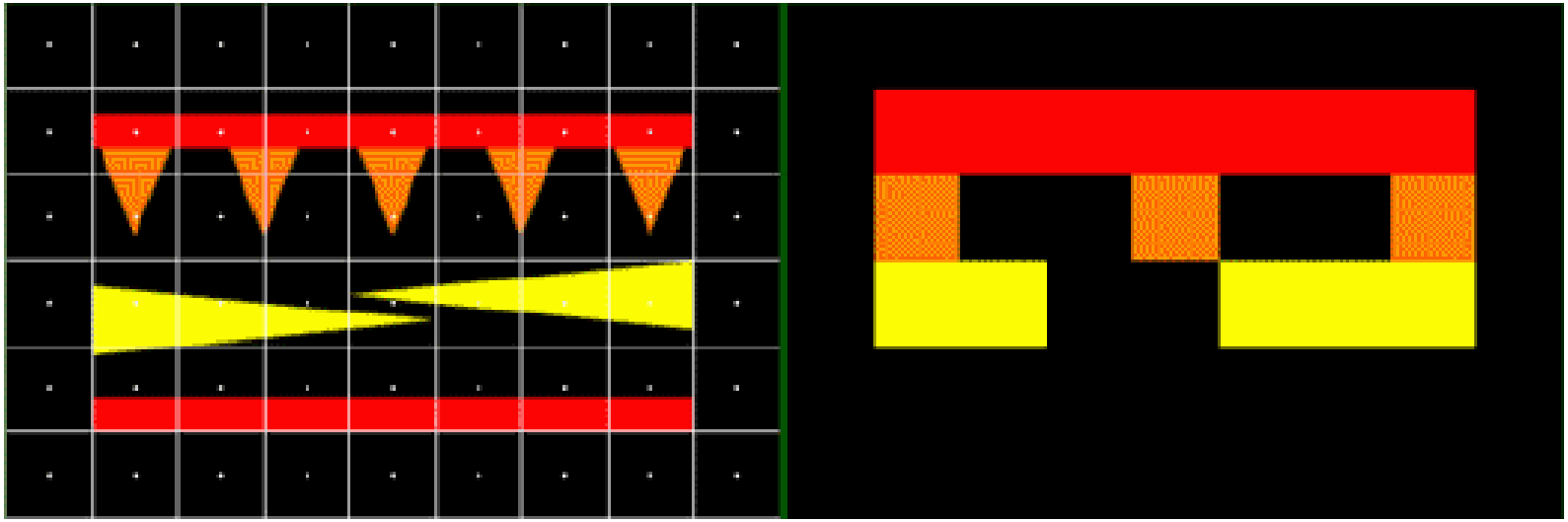
- Sampling, Texture Filtering
- **Aliasing**





Types of aliasing (1)

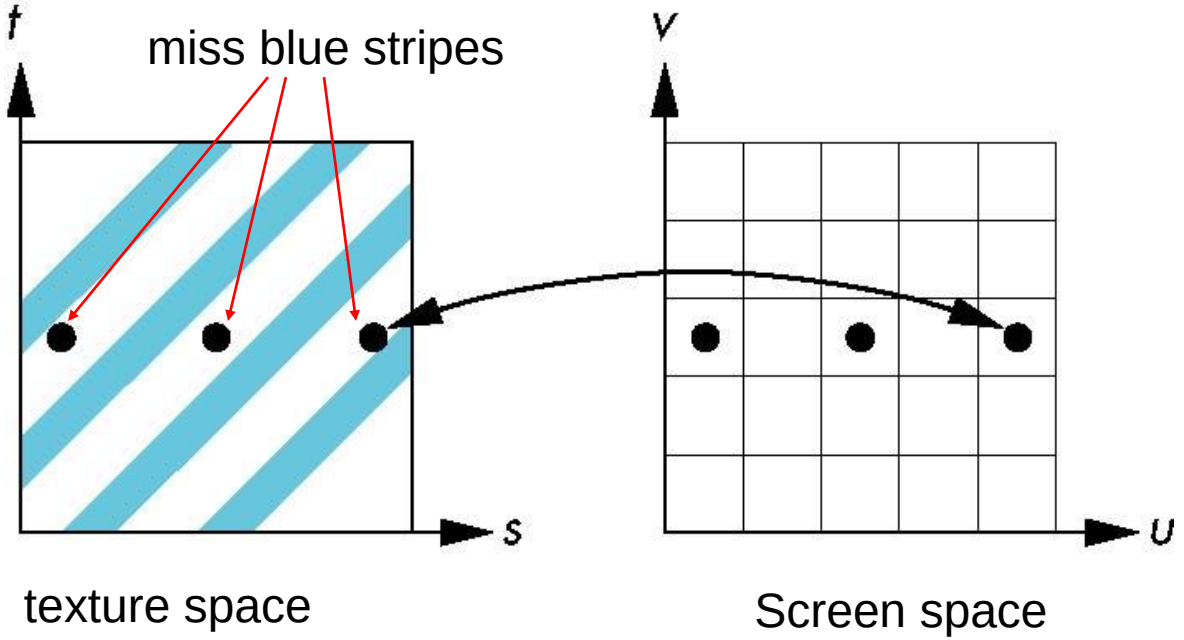
Loss of detail in the rendering



Point sampling

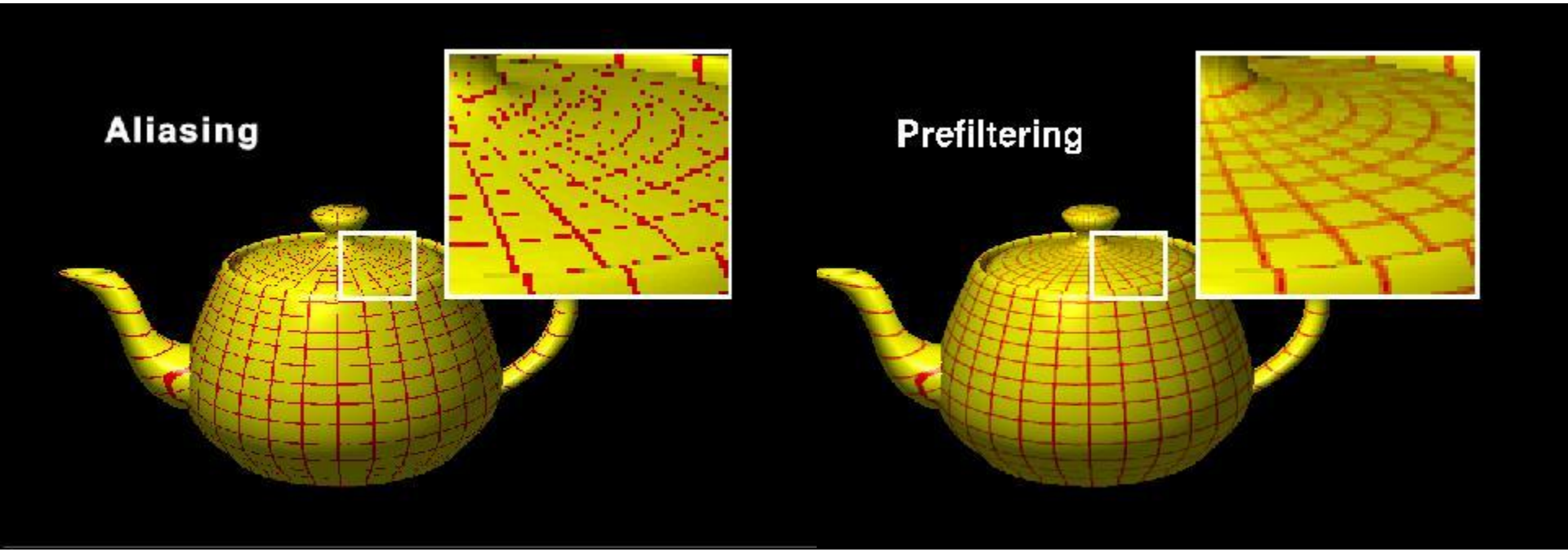
Texture mapping is a sampling operation and is prone to aliasing

If we sample the color at the center of the pixel (xs,ys) we get a single value. Point sampling of the texture can lead to aliasing errors.



Texture Filtering

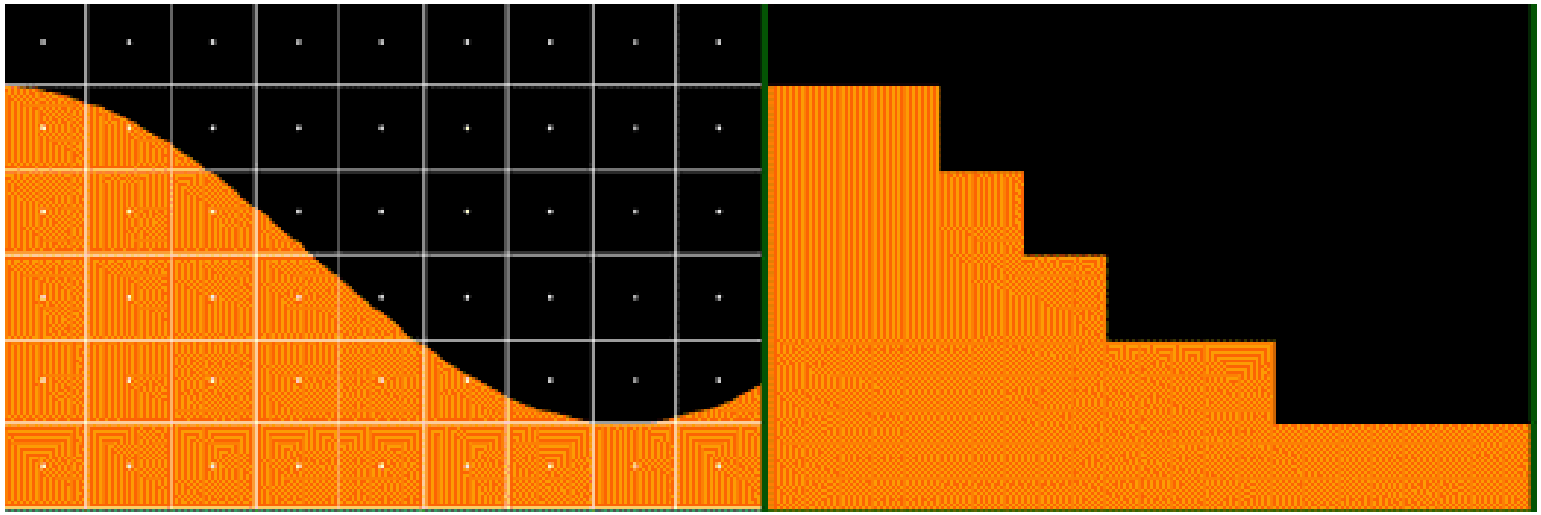
- Aliasing due to undersampling texture



Types of aliasing (2)

Jagged boundaries

Jaggies/Stairstepping is another form of *aliasing*



Rasterization causes display artifacts. They are due to the transition from the continuous to that discrete representation.

Anti-aliasing

Solutions:

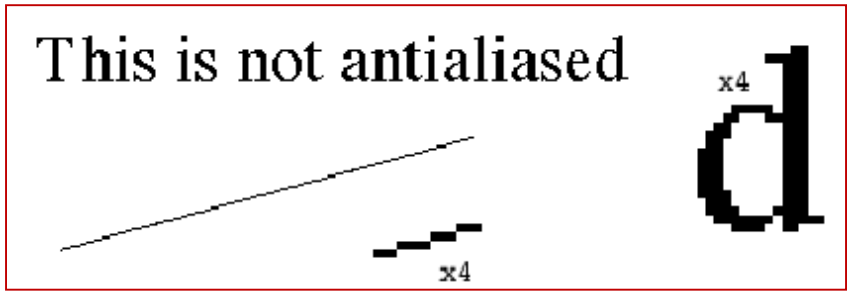
- For ray tracing/rasterization: render at higher resolution (render multiple samples per pixel), blur, resample at lower resolution **(supersampling)**

- For textures, we can also blur/prefilter the texture image before doing the lookup

  (prefilter texture map to eliminate high frequencies in texture signal)

Supersampling

- Your intuitive solution is to compute multiple color values per pixel and average them

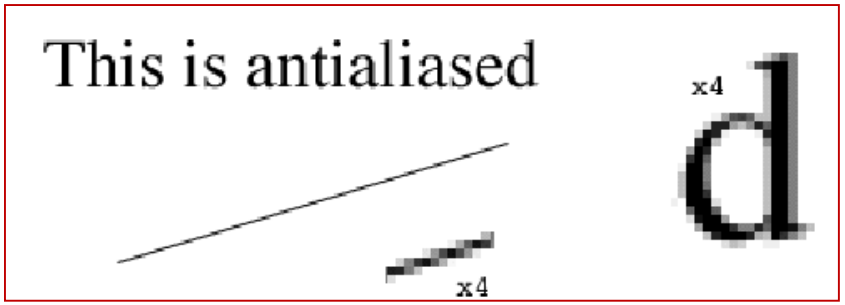


- A better interpretation of the same idea is that

–You first create a higher resolution image

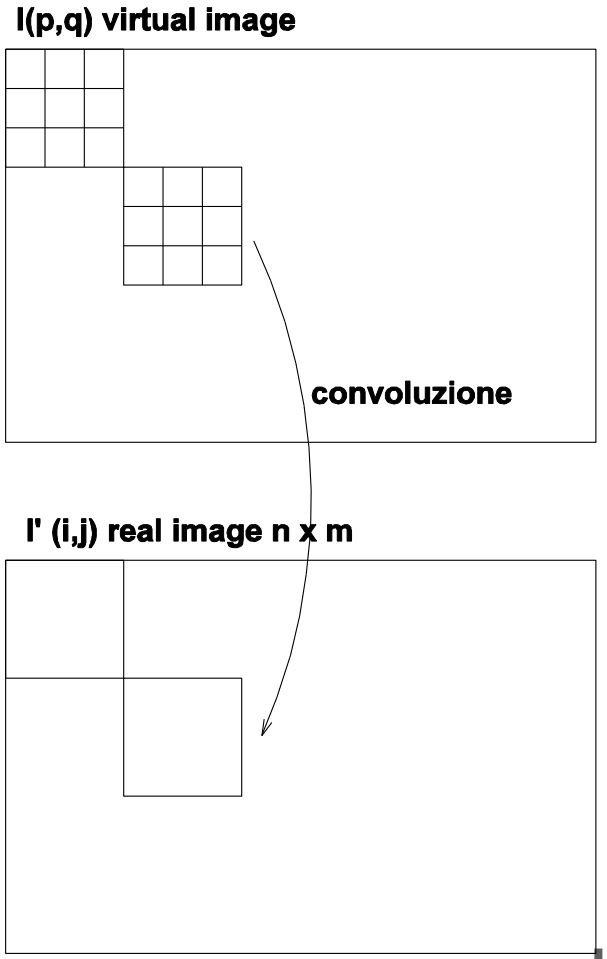
–You blur it (low pass, prefilter)

–You resample it at a lower resolution



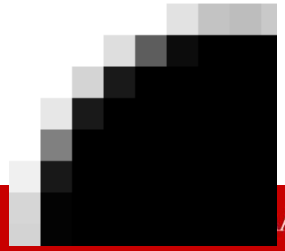
Supersampling

1. Compute a virtual image at resolution k*width, k*height
2. Blur it with a low-pass filter,
3. Resample it at a lower original image resolution.



- Advantage:

  implemented by Z-buffer.
- This is definitely a brute force approach, but is straightforward to implement and very powerful
- Works well for edges



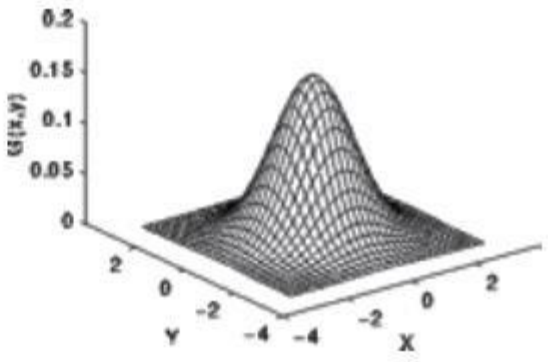
Step 2: Low-pass Filtering

- convolution of a low-pass filter h of dimension k with the virtual image :

$$I'(i,j) = \sum_{p=S_i-k}^{S_i+k} \sum_{q=S_j-k}^{S_j+k} I(p,q)h(S_i - p, S_j - q)$$

$\frac{1}{273}$

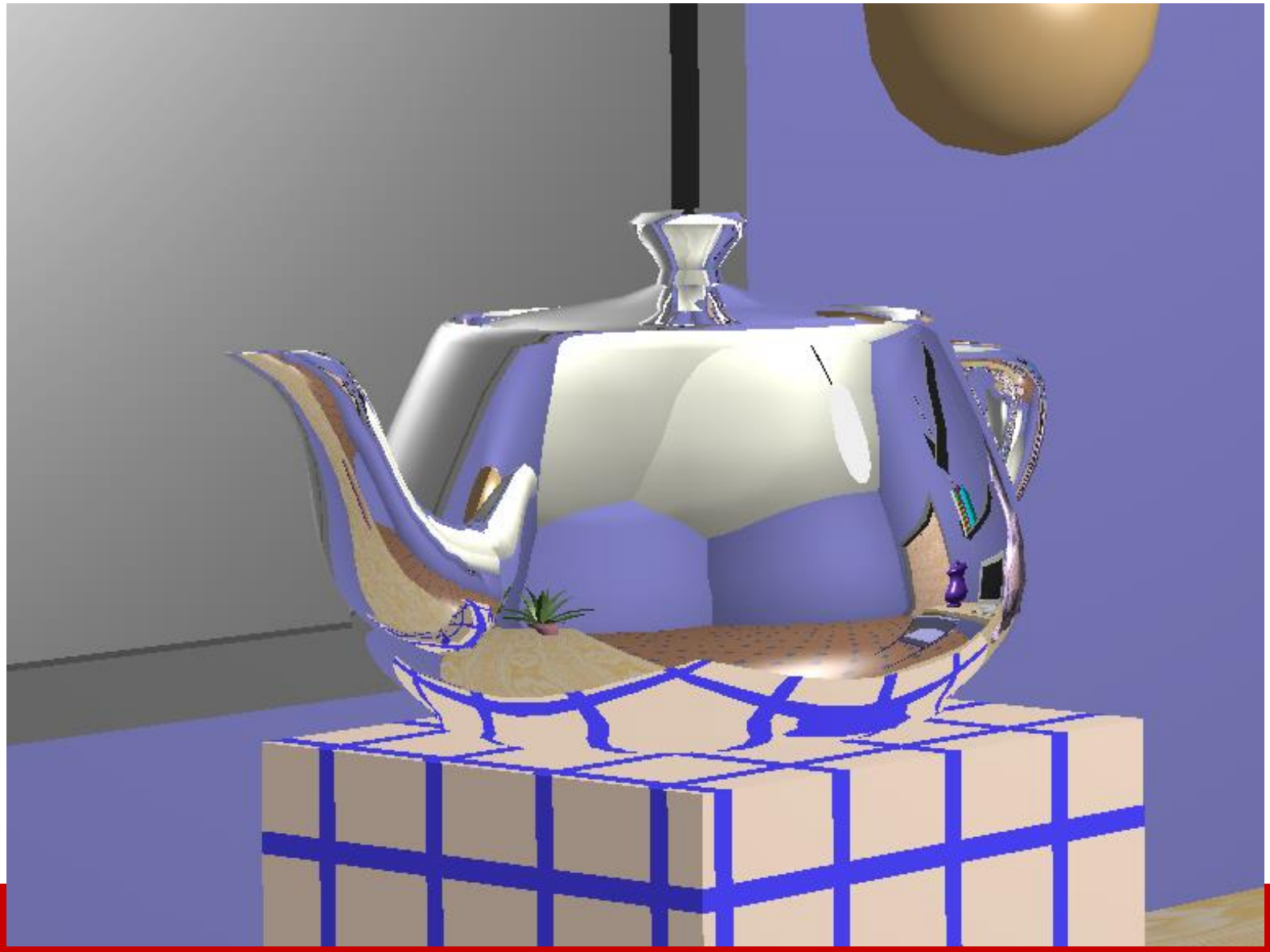
1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1



Gaussian

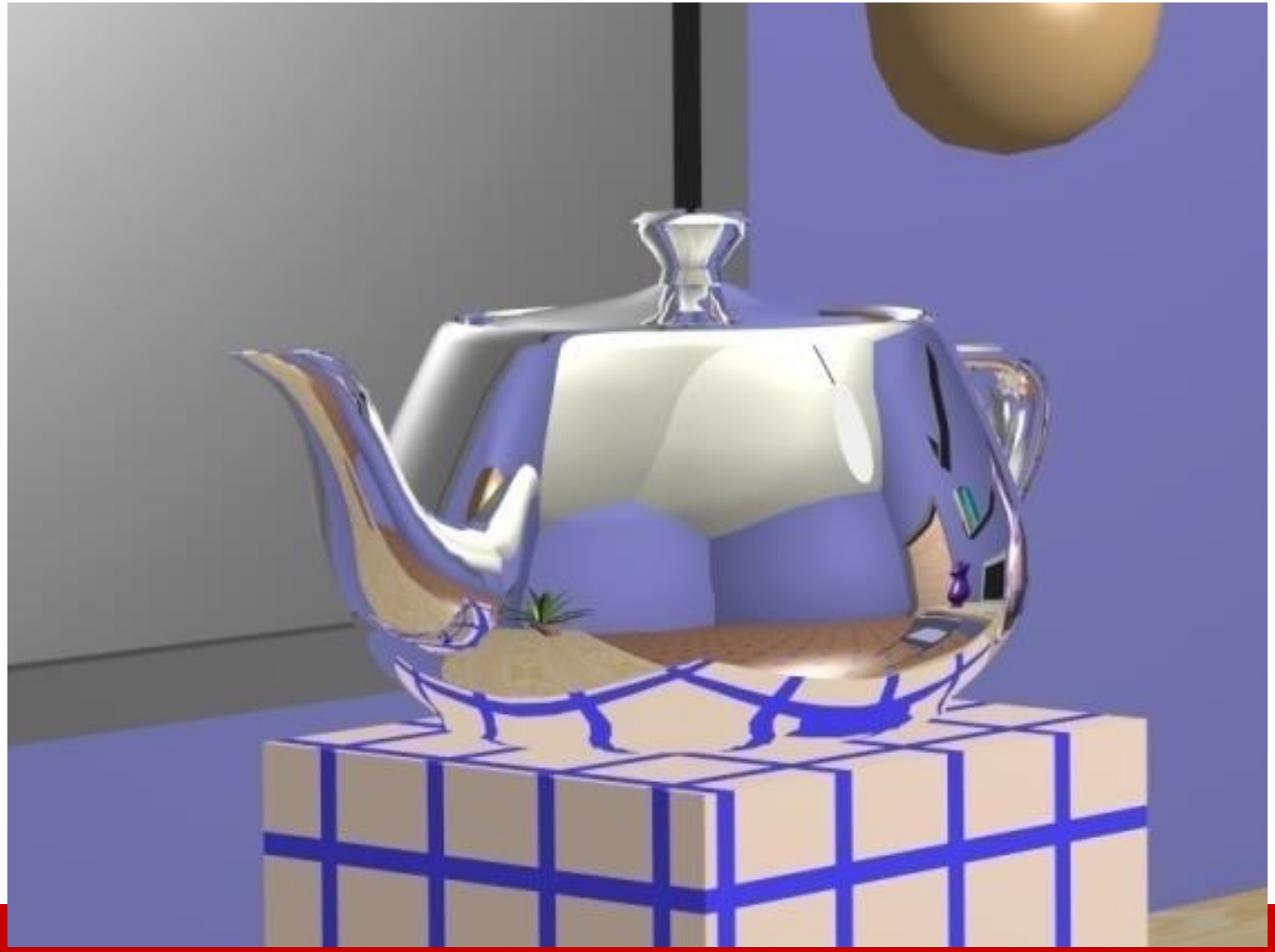
Antialiasing : example

Without antialiasing



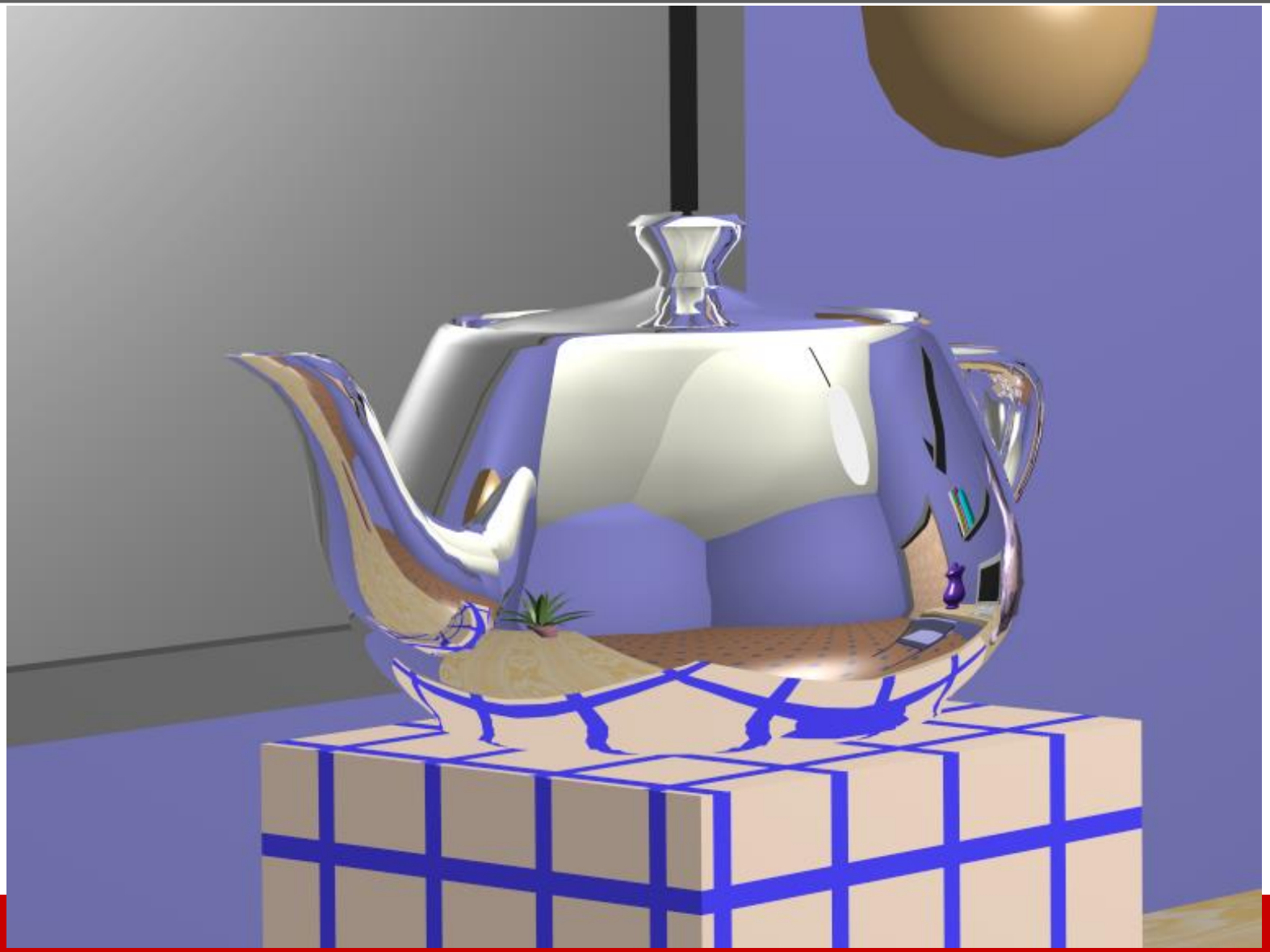
Antialiasing : example

Result from applying supersampling technique with 2x



Antialiasing : example

Result from applying supersampling technique with 3x



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Serena Morigi

Dipartimento di Matematica

serena.morigi@unibo.it

http://www.dm.unibo.it/~morigi