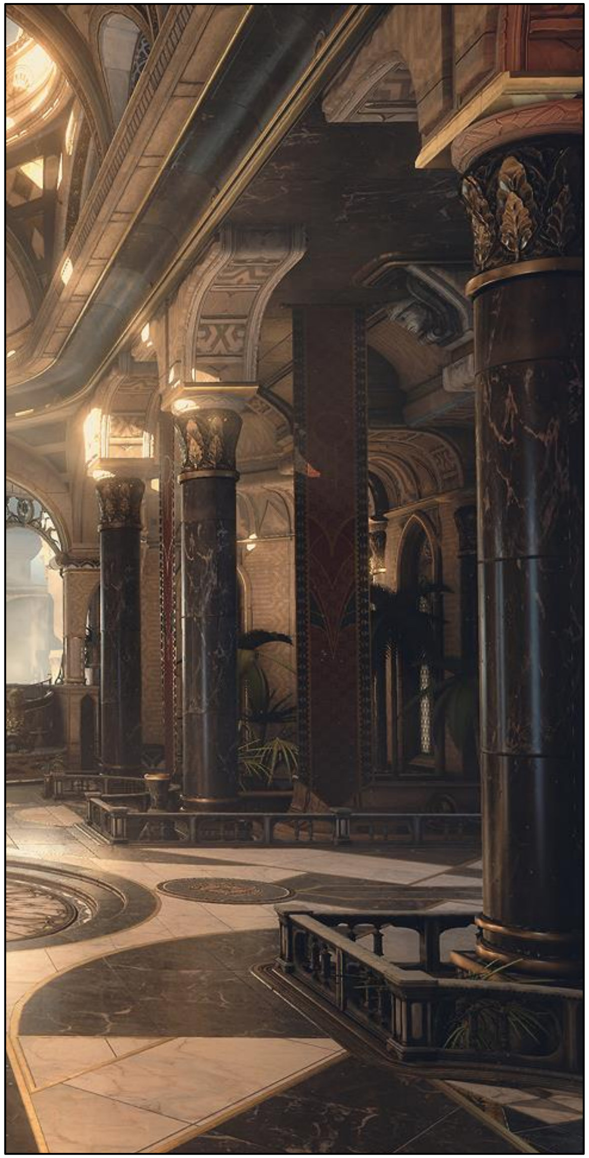
Texturing

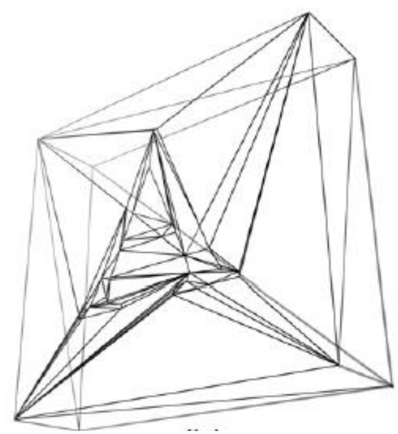


Georgios Papaioannou - 2015

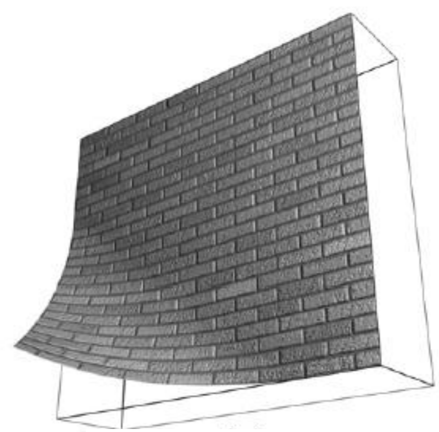
INTRODUCTION

What is Texturing?

- Spatio-temporal modification of material attributes, independent of the geometry itself
- Why do we need it?
  - It is impossible to capture these variations as geometric attributes:



Ok, this can be done with geometry modification



This cannot…

Spatial Variation of Material Properties

Types of Texturing

Image texturing:

- The spatial/temporal patterns are expressed in the form of a digitized bitmap
- A bitmap texture can be an array of values of 1/2/3 + time dimensions
- Textures are stored in GPU/CPU memory and sampled during rendering

Procedural texturing:

- The spatial/temporal patterns are generated using a function or algorithm

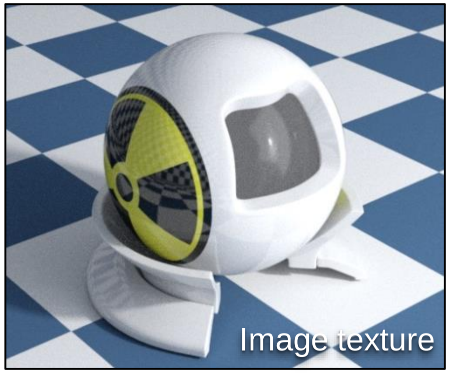
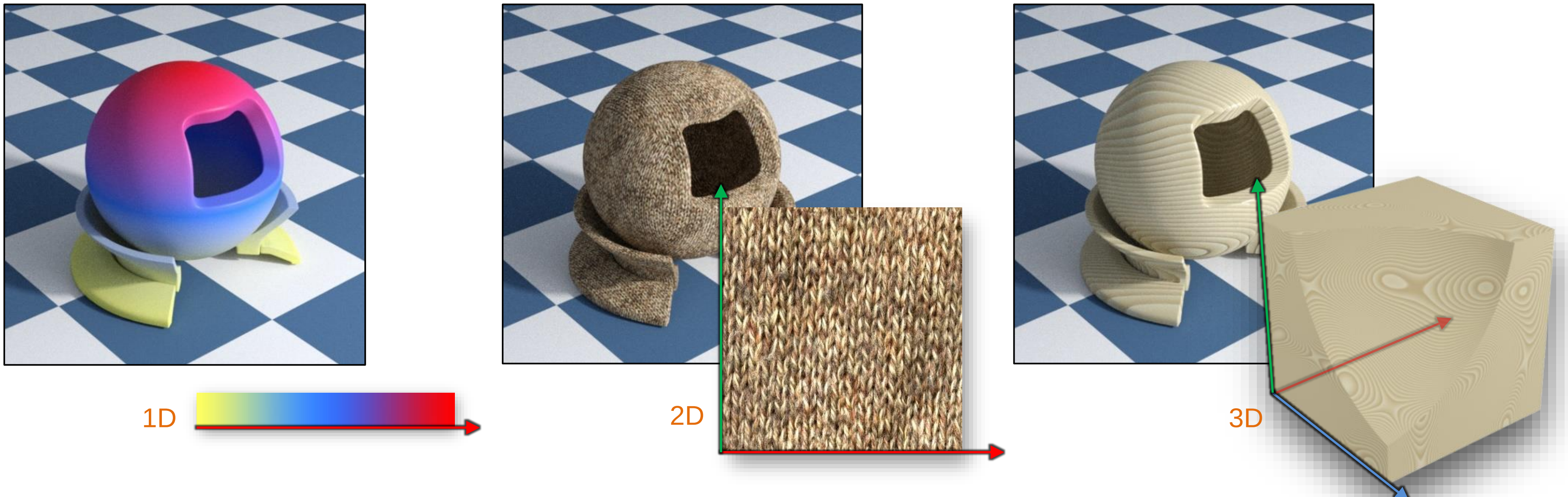


IMAGE TEXTURING

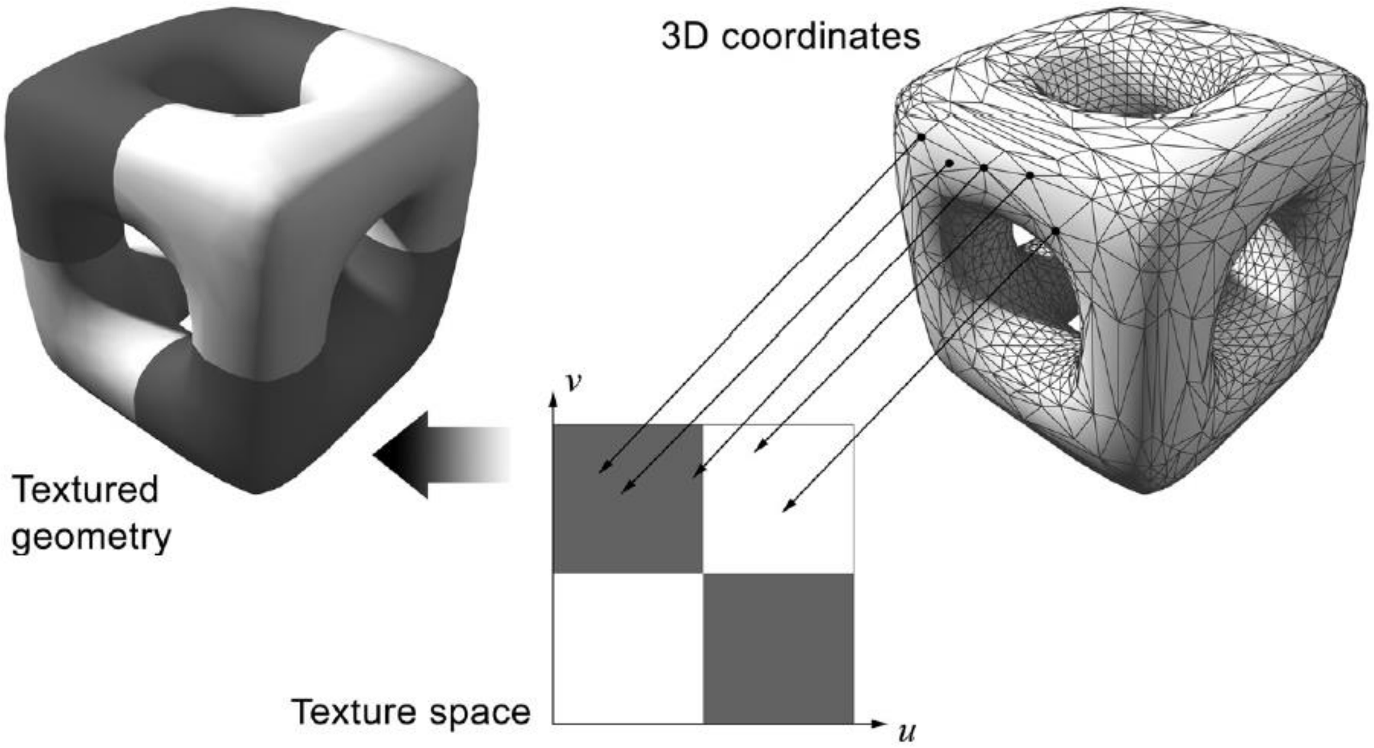
Image Texture Space



- A 1D-3D image texture is defined in a texture parametric space
- The parametric space is usually considered normalized w.r.t the dimensions of the raster
  - For example, a 2D raster is defined on a plane with two parameters (e.g. u,v)

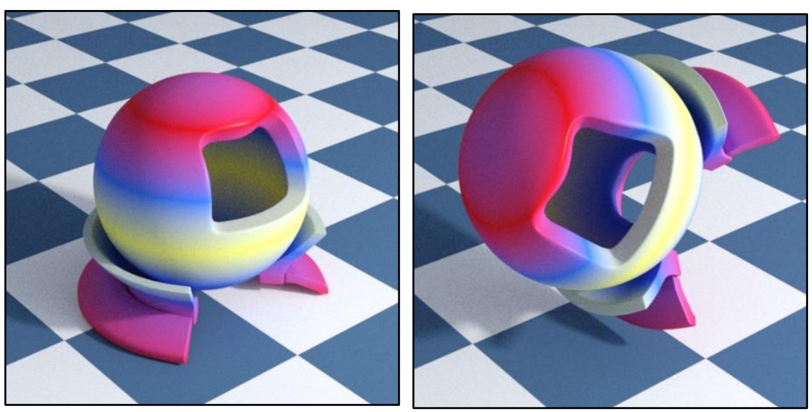
Texture Mapping (1)

- In order to apply an image texture to a surface (or solid interior), we must define a mapping from the point or vector coordinate system to the texture space

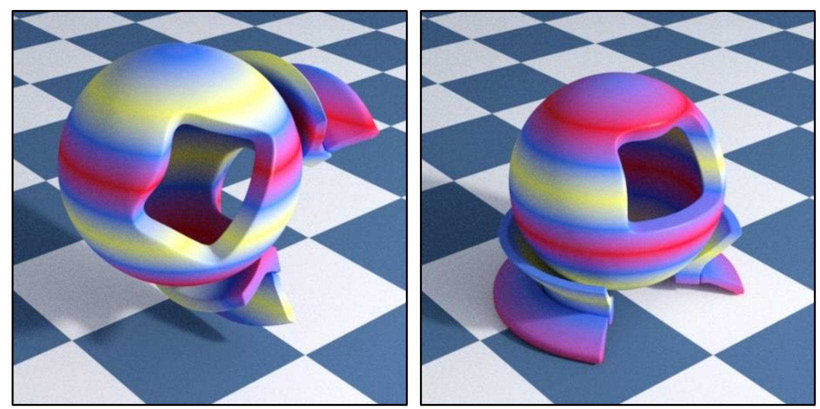


Texture Mapping (2)

- The texture mapping can be performed from any coordinate system (OCS,WCS,ECS,CSS)
- Usually, we calculate the texture parameters at modeling time (OCS) and store them on the model vertices
- The calculation of the texture parameters is done via a texture projection function



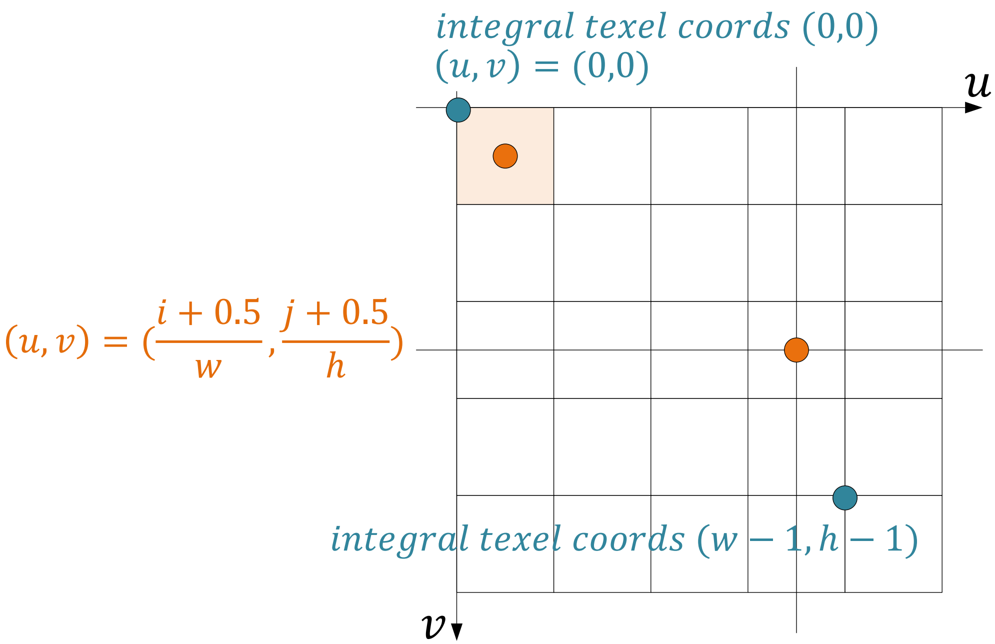
Object space texture mapping



World space texture mapping

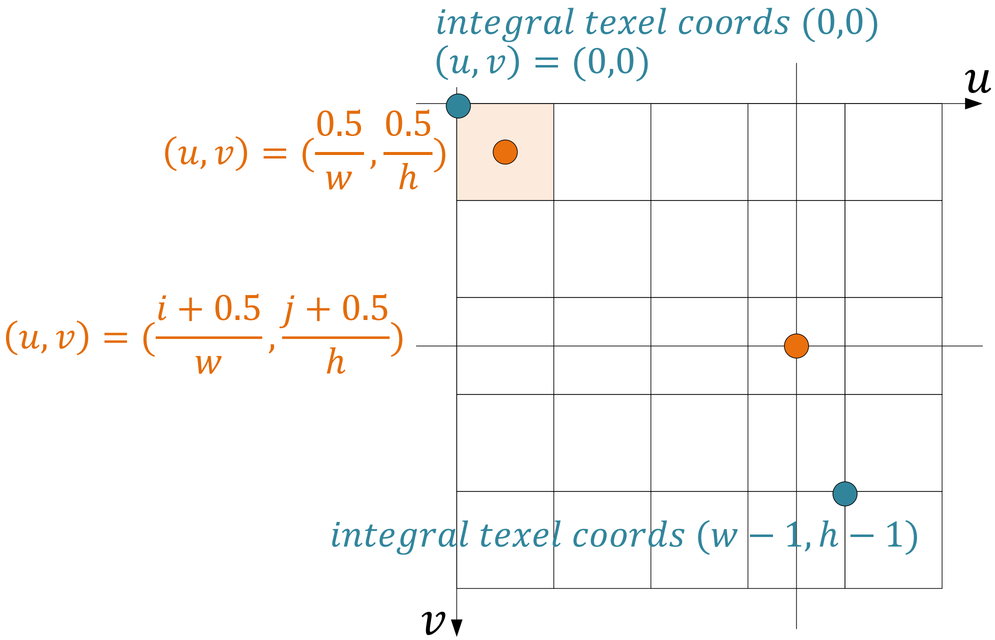
Texture Elements (1)

- The smallest accessible element in a 1D/2D/3D raster texture is the texel (texture element)
- Texels are considered discrete samples on the raster and their integral coordinates correspond to corners of the raster elements



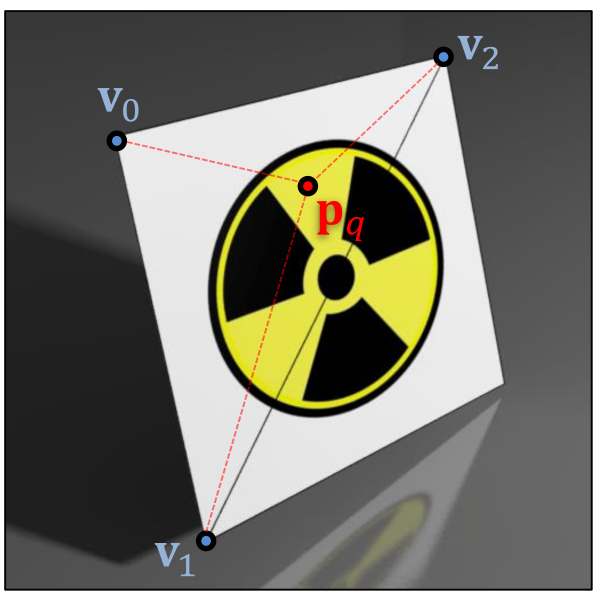
Texture Elements (2)

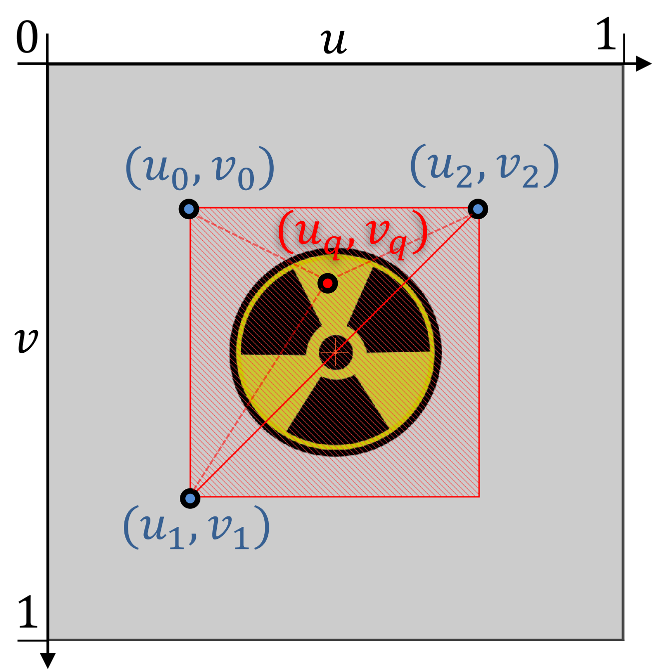
- Sampling centers do not coincide with the integral texel coordinates!
- They are mapped to the centers of the texels, as shown below



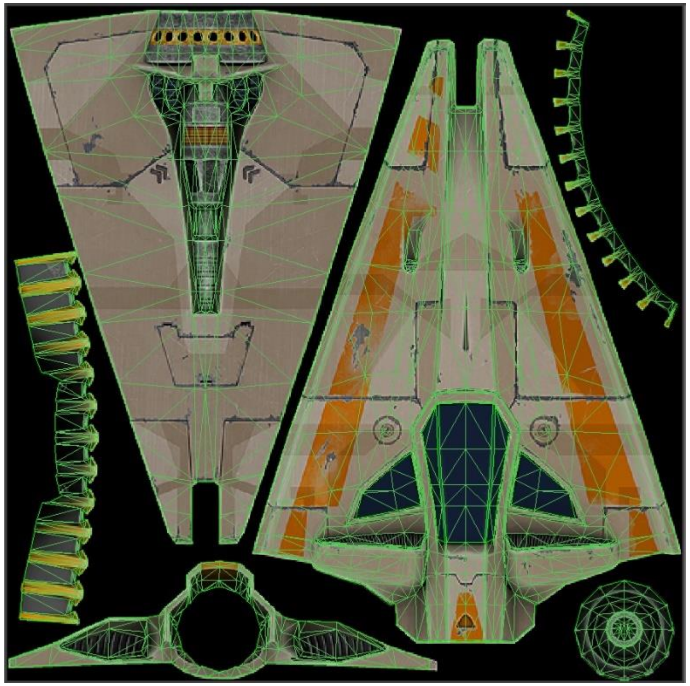
Texture Mapping Triangles (1)

- Texture coordinates on arbitrary locations on the triangles are interpolated from the tex. coordinates of the triangle vertices, using the same barycentric coordinates used for other attributes





Texture Mapping Triangles (2)

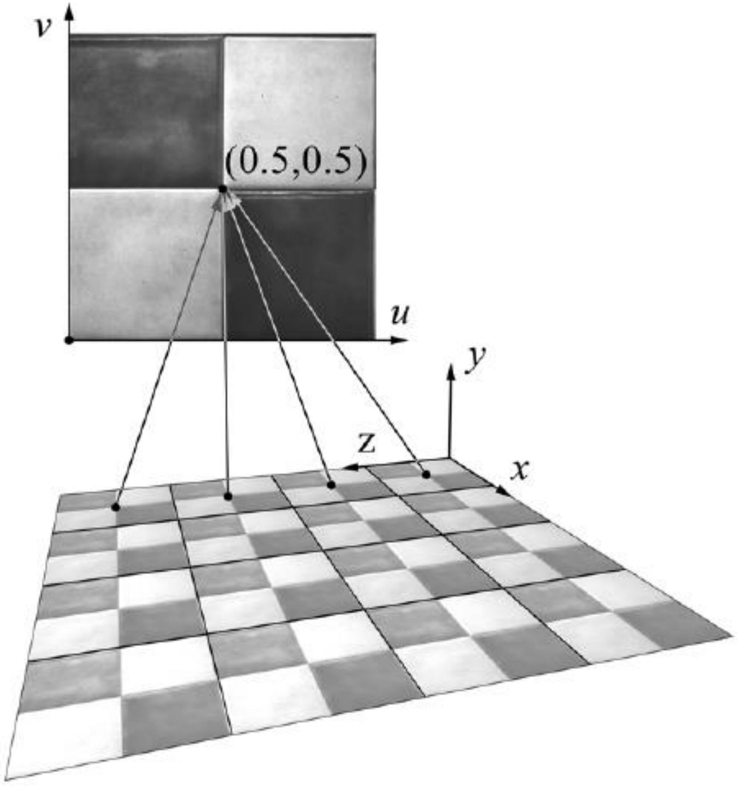


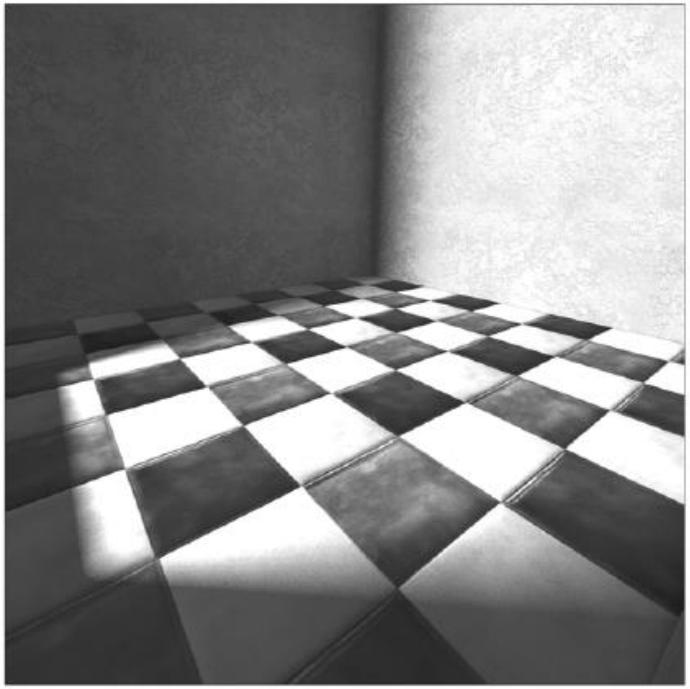
- Fine adjustments of the triangle texture coordinates can be done directly on the parametric space
  - Vertex texture coordinates can be manipulated using a 2D editor, the UV Editor
  - Texture coordinates with vertex connectivity can be rendered with orthographic projection as (u,v,0) points

Asset from the Unity 5.0 game engine example projects

Texture Coordinate Wrapping (1)

- In general, multiple points on the geometry may index the same texture coordinates → The mapping is not necessarily bijective





Texture Coordinate Wrapping (2)

- The parametric space coordinates lie in the range [0,1]
- Therefore, all texture coordinates are conformed to this range using a texture wrapping function. Typical examples:

GL_REPEAT

GL_MIRRORED_REPEAT

GL_CLAMP_TO_EDGE

GL_CLAMP_TO_BORDER

$$s = s - \lfloor s \rfloor$$

$$s = \begin{cases} s - \lfloor s \rfloor & , \lfloor s \rfloor \ even \\ 1 - s + \lfloor s \rfloor & , \lfloor s \rfloor \ odd \end{cases}$$

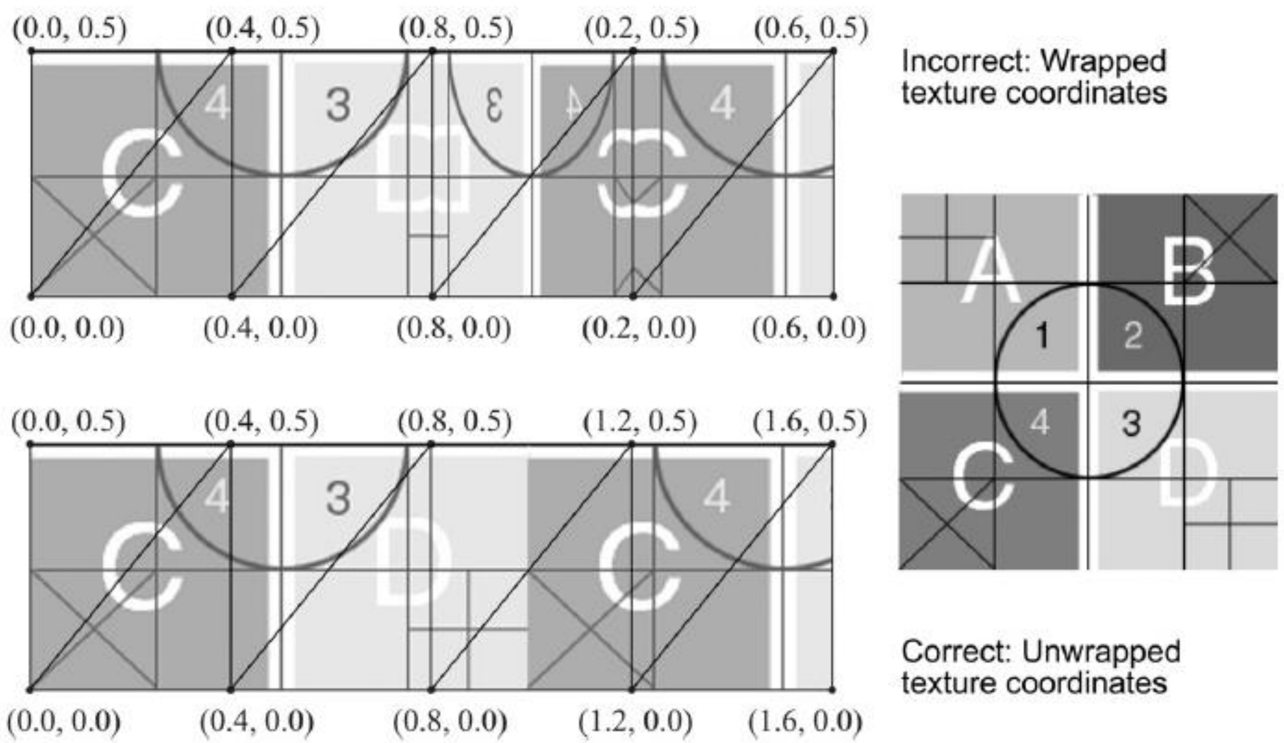
$$s = \max(\min(1, s - \lfloor s \rfloor), 0)$$

$$color = \begin{cases} Sample(s) & , 0 \leq s \leq 1 \\ border & , otherwise \end{cases}$$

Image from: http://learnopengl.com/#!Getting-started/Textures

Texture Coordinate Wrapping (2)

- However, wrapping is not performed during assignment to vertices
- Interpolation may fold them back, producing erroneous results

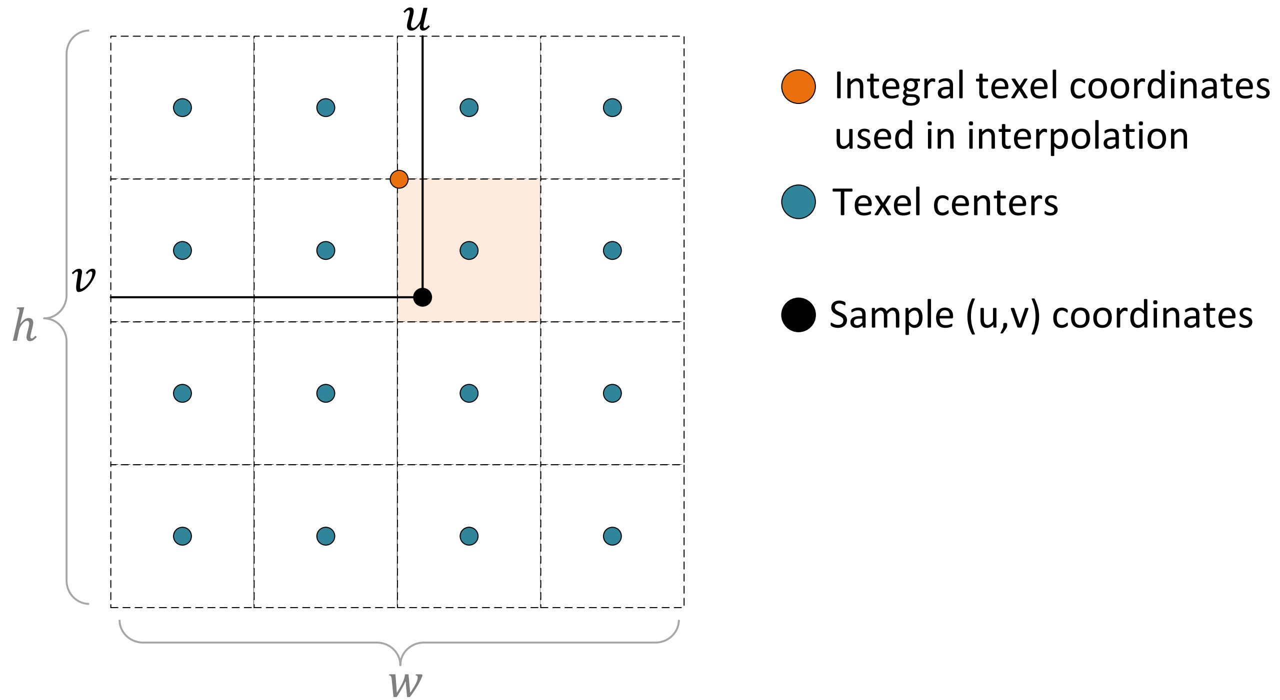


Sample Evaluation

- Given a (u,v) coordinate pair, the simplest (albeit not the best) way to evaluate the texture at (u,v) is to retrieve the nearest texel to the parametric coordinates:

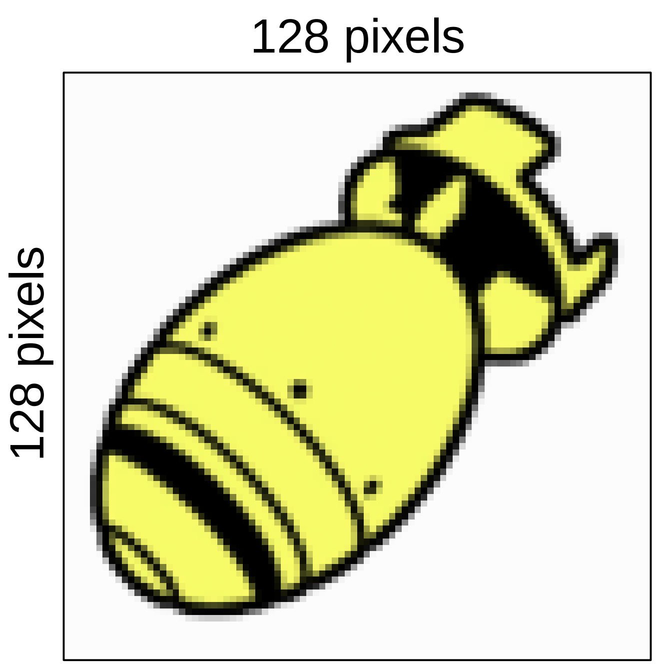
$$Sample(u, v) = Texel(\lfloor x \rfloor, \lfloor y \rfloor)$$

$$x = uw$$
$$y = vh$$



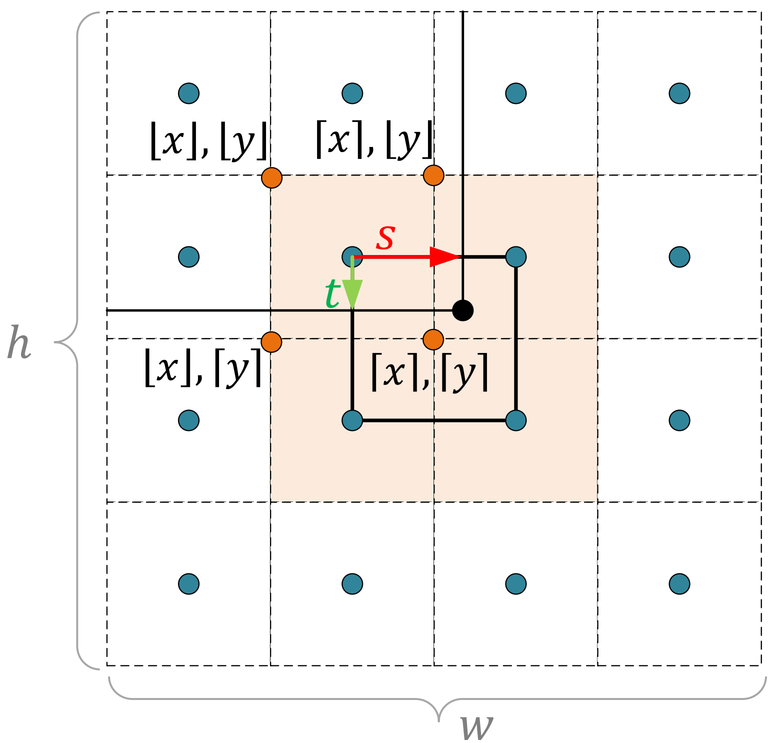
Texture Magnification

- However, nearest neighbor texture sampling produces visible artifacts (pixelization) when the texture is magnified (many pixels index the same texel)



Bilinear Texture Interpolation (1)

- To create smooth sample transitions, we can interpolate the texel values according to the distance of the (u,v) coordinate from the 4 nearest texels:



- Integral texel coordinates used in interpolation
- Texel centers
- Sample (u,v) coordinates

$$x = uw - 0.5 \qquad s = x - \lfloor x \rfloor$$
$$y = vh - 0.5 \qquad t = y - \lfloor y \rfloor$$

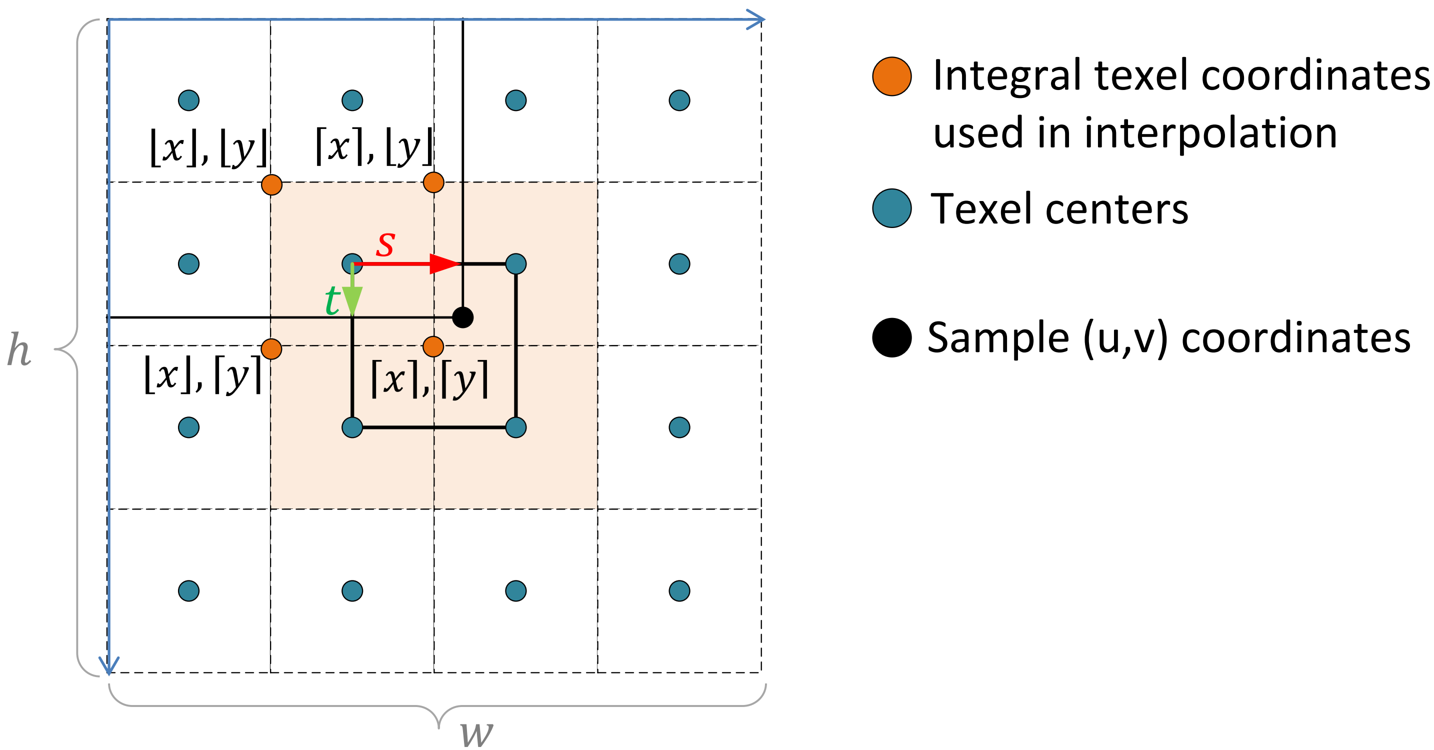
Bilinear Texture Interpolation (2)

- Using (bi-)linear interpolation:

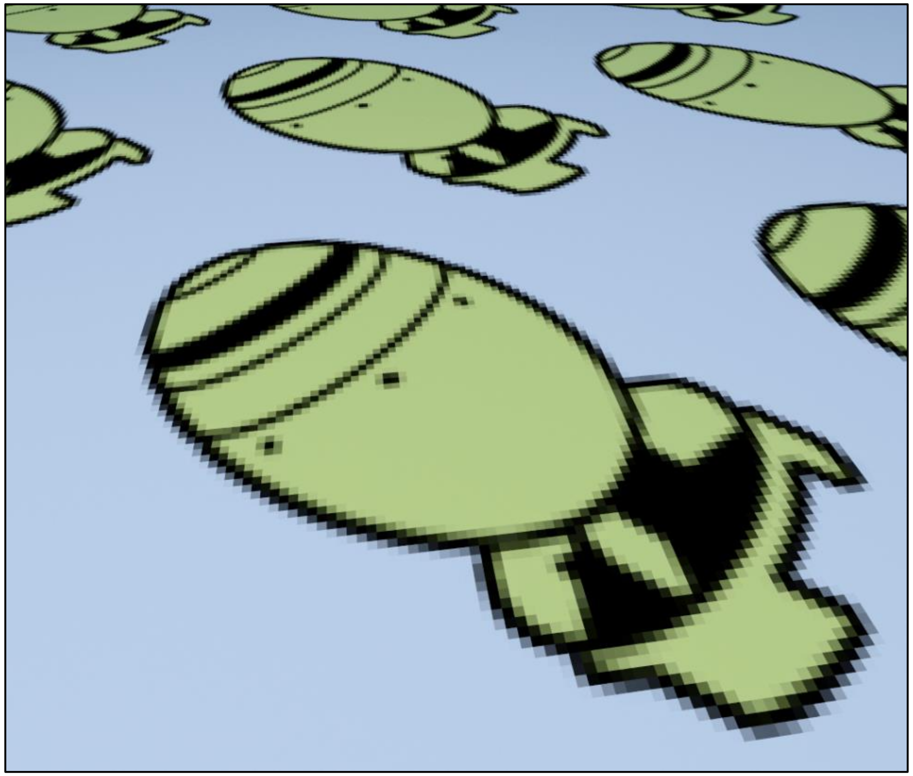
$$x = uw - 0.5 \qquad s = x - \lfloor x \rfloor$$
$$y = vh - 0.5 \qquad t = y - \lfloor y \rfloor$$

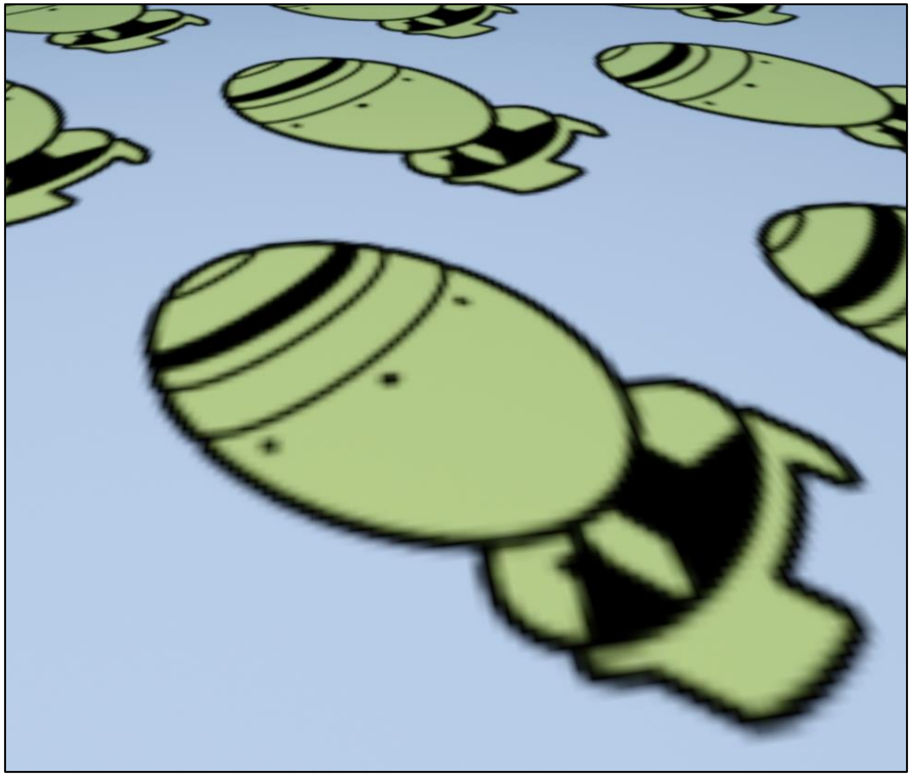
$$T_L = Texel(\lfloor x \rfloor, \lfloor y \rfloor)(1 - s) + Texel(\lceil x \rceil, \lfloor y \rfloor)s$$
$$T_H = Texel(\lfloor x \rfloor, \lceil y \rceil)(1 - s) + Texel(\lceil x \rceil, \lceil y \rceil)s$$

$$Sample(u, v) = T_L(1 - t) + T_U t$$



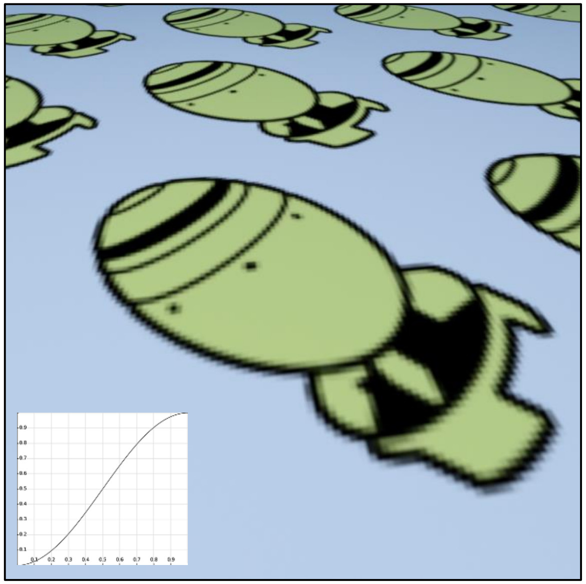
Bilinear Texture Interpolation (3)



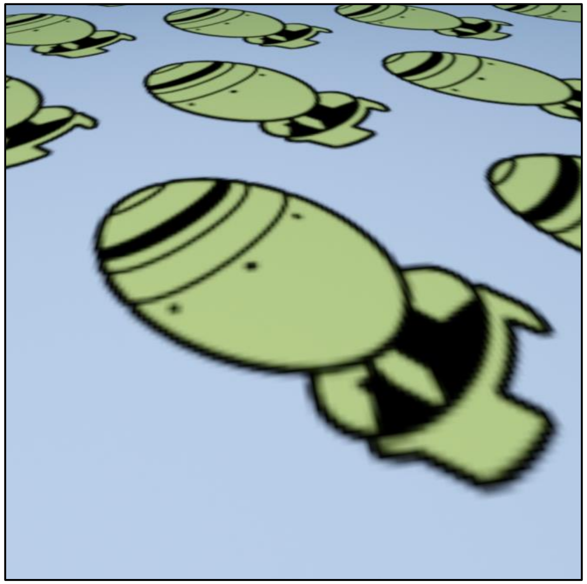


- The bilinear interpolation is standard in the GPU hardware and all production rendering software

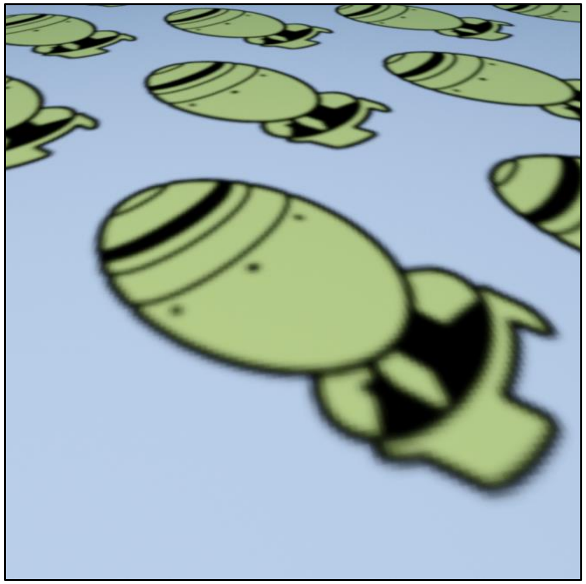
Other Texture Interpolation Functions



Cosine-weighted "sharp" interpolation



Bi-linear



16-tap radially weighted "smooth" interpolation

- Many other methods for interpolating the texel samples can be used such as the above

Texture Minification

- When many texels correspond to a single pixel sample (area), then we have texture minification
- The texture is insufficiently sampled, resulting in distortion and noise

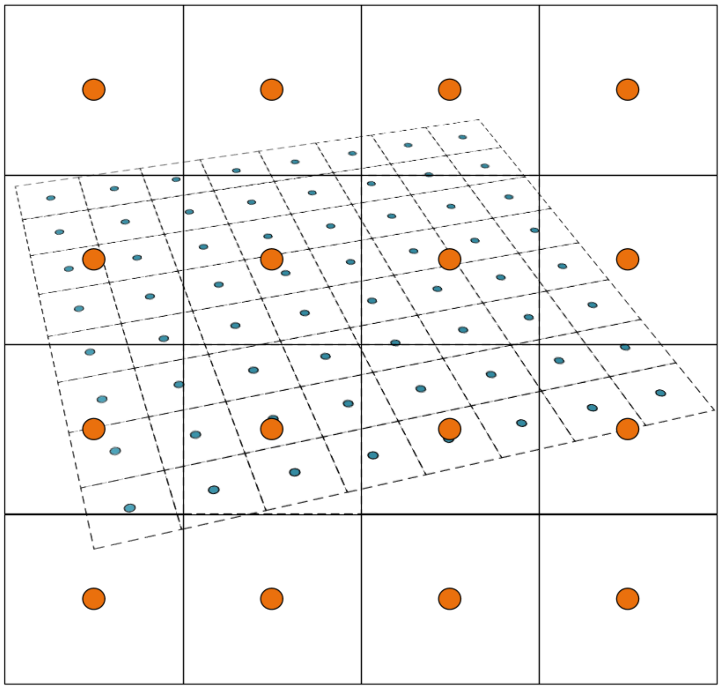
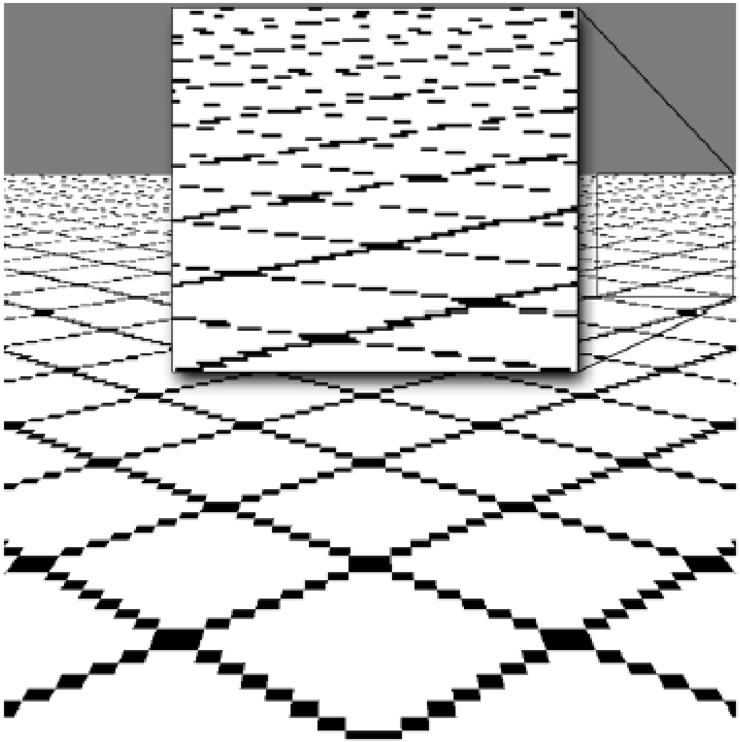
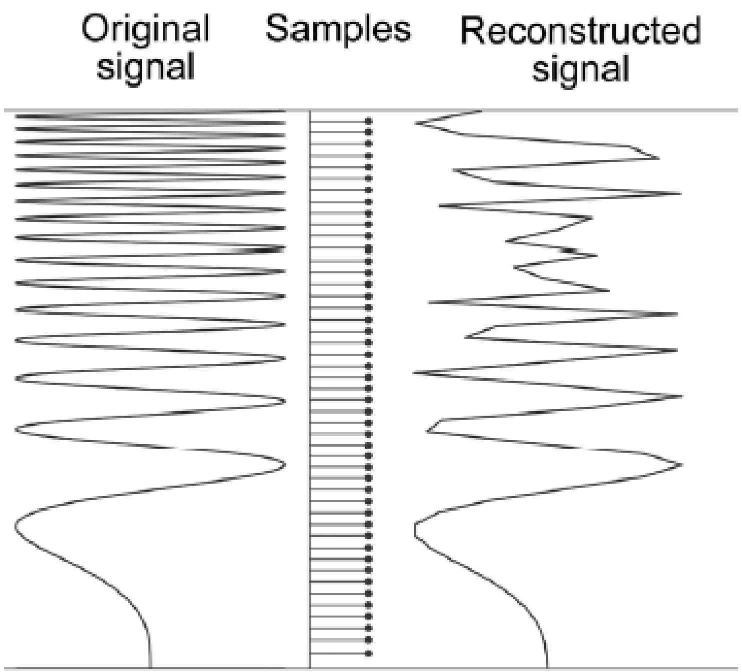


Image samples   Texel centers

Texture Minification - Aliasing

- This is due to the signal aliasing that occurs, since the rate of projected texels on screen is higher than half the sampling rate (pixel samples) – sampling theorem

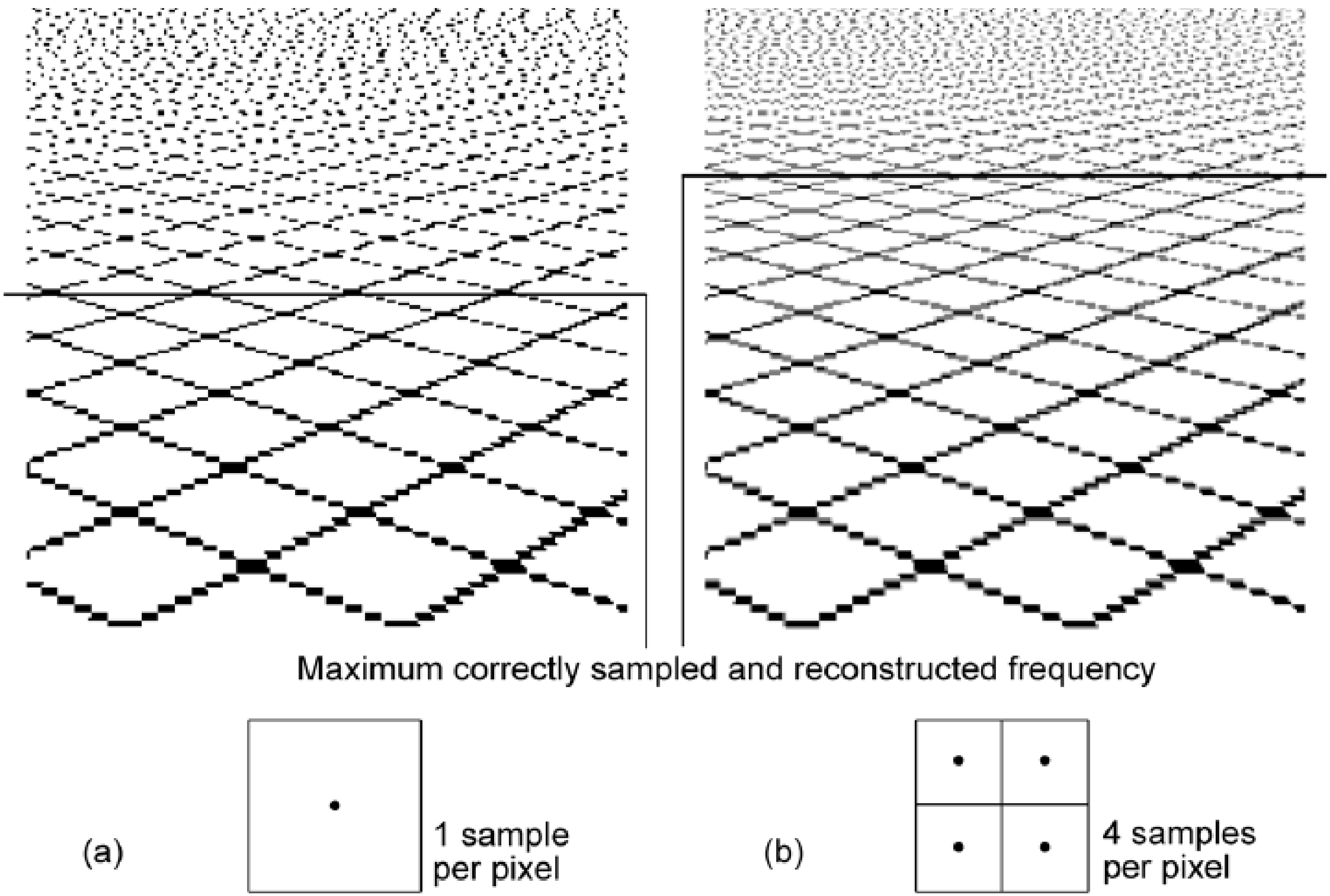




Texture Antialiasing – Supersampling? (1)

- Can we use a higher sampling rate to correct the aliasing?
  - No. Even if we effectively multiply the pixel shading rate, we only mitigate the problem to higher frequencies
  - We have no way of predicting the highest frequency of the projected texels in image space

Texture Antialiasing – Supersampling? (2)



Texture Antialiasing with Band Limiting

- The only way to get rid of aliasing is to deliberately limit the frequency of the texture before sampling (pre-filtering) so that the image sampling rate always suffices
- Therefore we apply an antialiasing filter per pixel sample
- The antialiasing filter is a low-pass filter and can be implemented in the texture domain as a weighted average

Determining the Texture Filter Shape (1)

- To determine the weighted average sample $Sample(u, v)$, we must account for all texels projected in the pixel sample area of influence (a “square pixel“ in the example below)

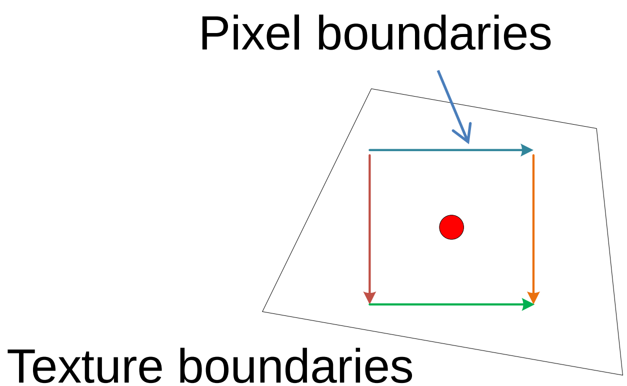
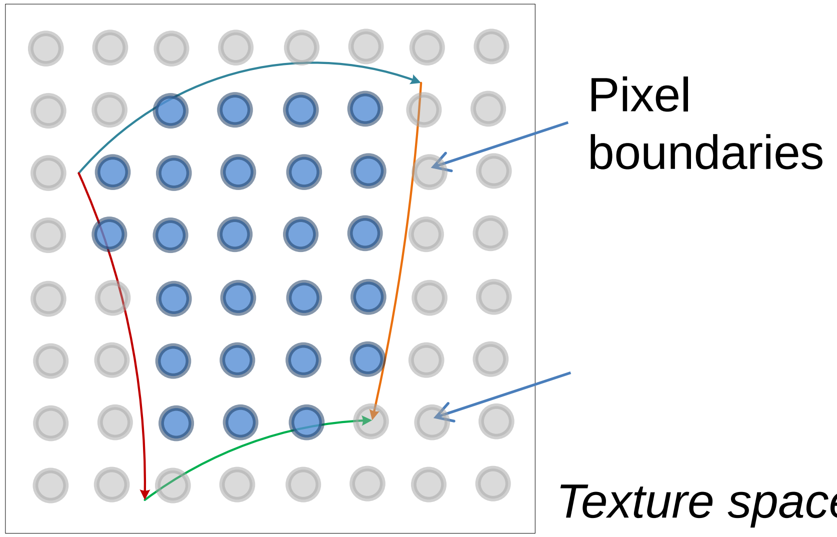


Image space

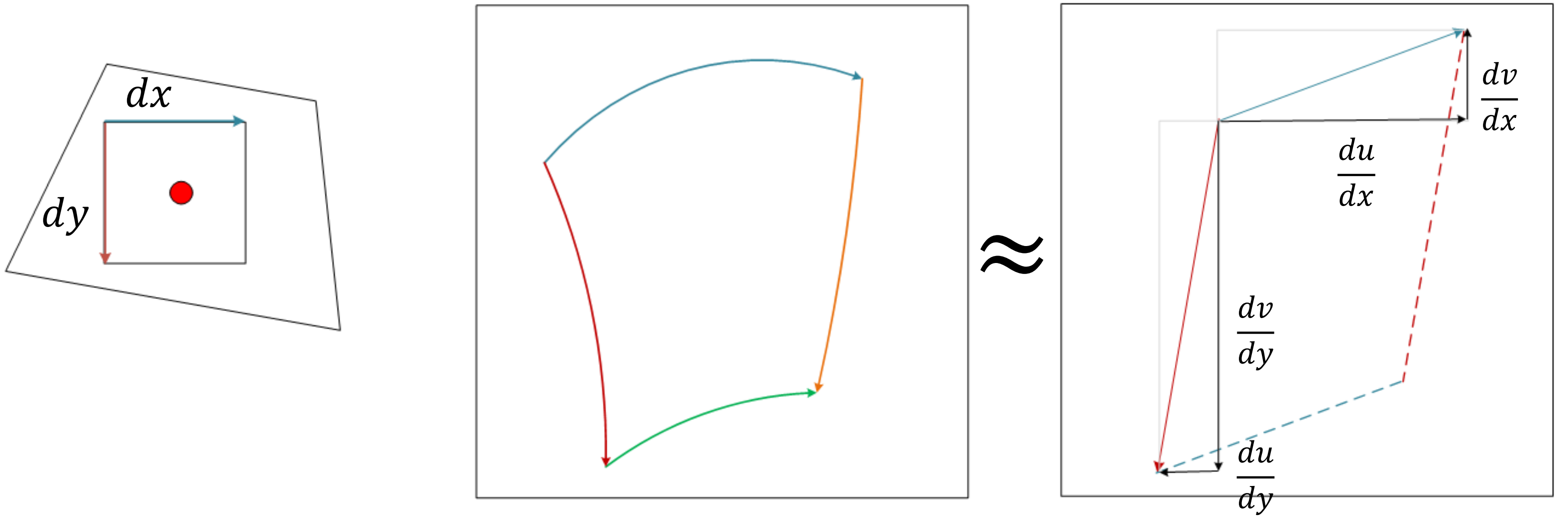
Projected pixel boundaries in texture space
“pixel pre-image”



Texture space

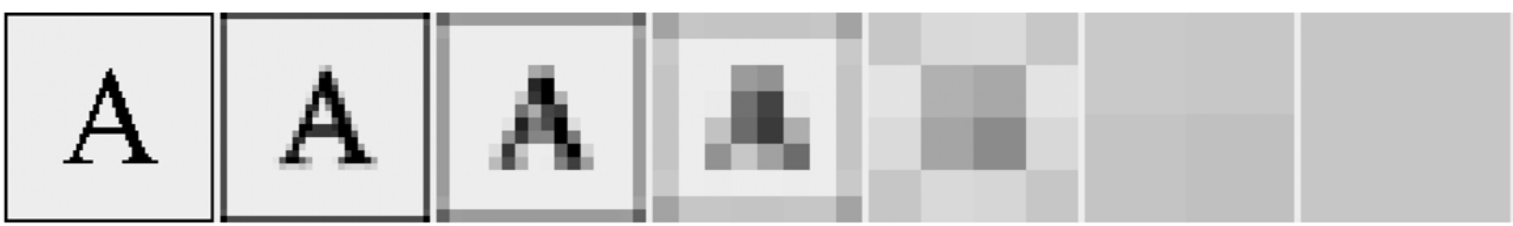
Determining the Texture Filter Shape (2)

- For relatively small image-space distortions of the projected texels in image space, we can approximate the pixel pre-image with a parallelogram
- We determine the shape of the linear approximation using the pixel derivatives of the texture parameters



Mip-Mapping (1)

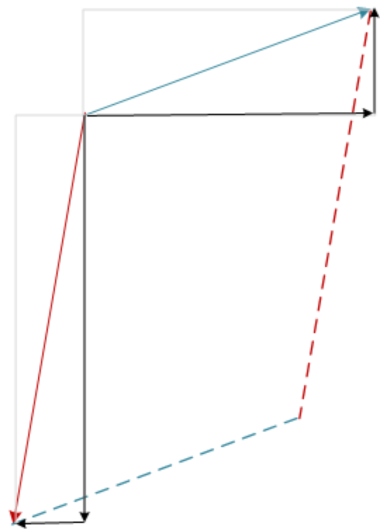
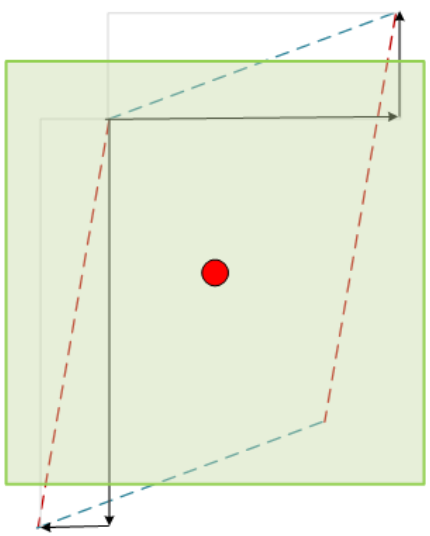
- It is impractical to determine the pre-image texels and filter them at run-time
  - The filter may just as well cover up to the entire image!
- We pre-filter the texture data using square filters of increasing size and store them
- This process is called MIP-Mapping: “Multum In Parvo” (many things in a small place)



Mip-map size	64X64	32X32	16X16	8X8	4X4	2X2	1X1
Filter size	1X1	2X2	4X4	8X8	16X16	32X32	64X64

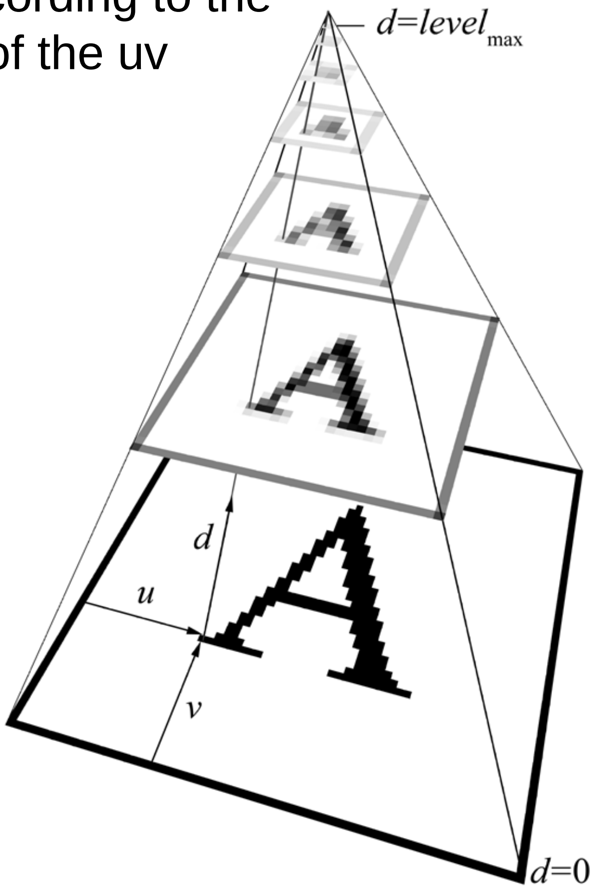
Mip-Mapping (2)

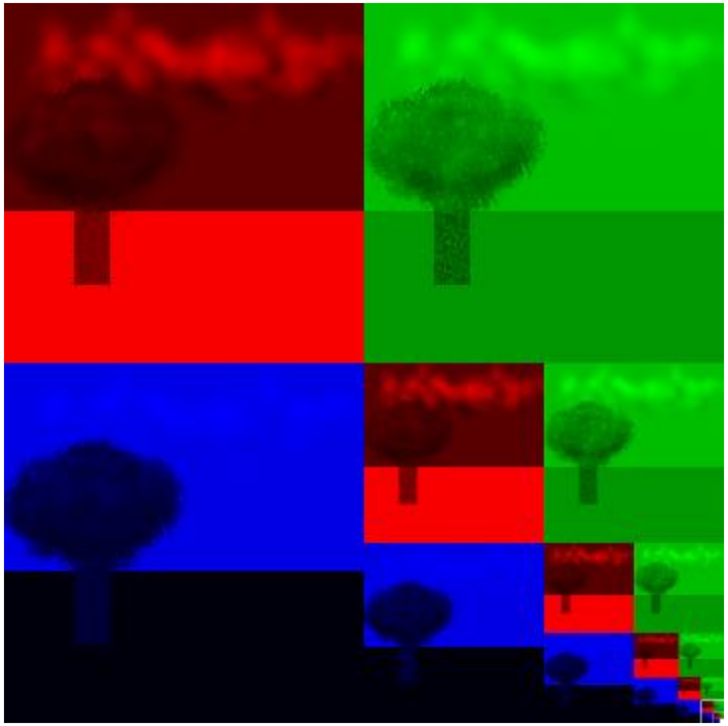
- At run time, we determine the most compatible filtered "version" of the texture and use the corresponding pre-filtered data at (u,v)
- The pre-image is approximated by a square region centered at (u,v)

MIP-Mapping (3)

MIP Map hierarchy. We select which pre-filtered version of the image to use, according to the pixel derivatives of the uv coordinates





MIP storage. The total storage area is increased by 33%

MIP Map Determination

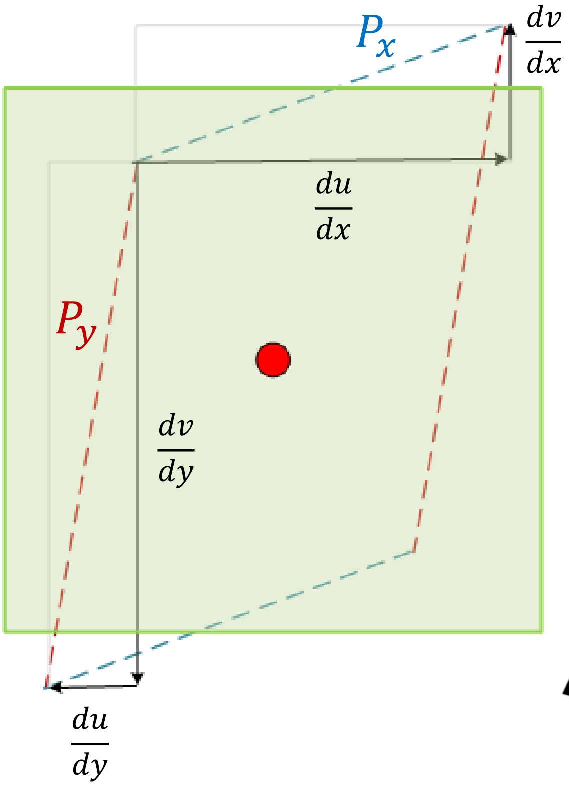
$$P_x = \sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2}$$

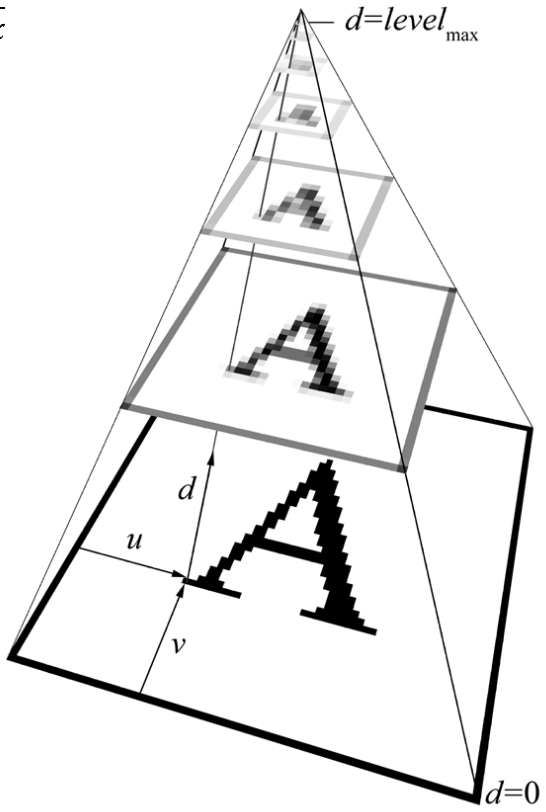
$$P_y = \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}$$

$$P = \max\{P_x, P_y\}$$

$$\lambda = \log_2(P)$$

$$d = \begin{cases} level_{max} & \lambda > level_{max} \\ 0 & \lambda < 0 \\ \lambda & otherwise \end{cases}$$





Problem of Isotropic Filtering

- The square filter shape is not appropriate for elongated pre-images, as its cannot represent the required area of support
- Results in over-blurring along the minor pre-image direction

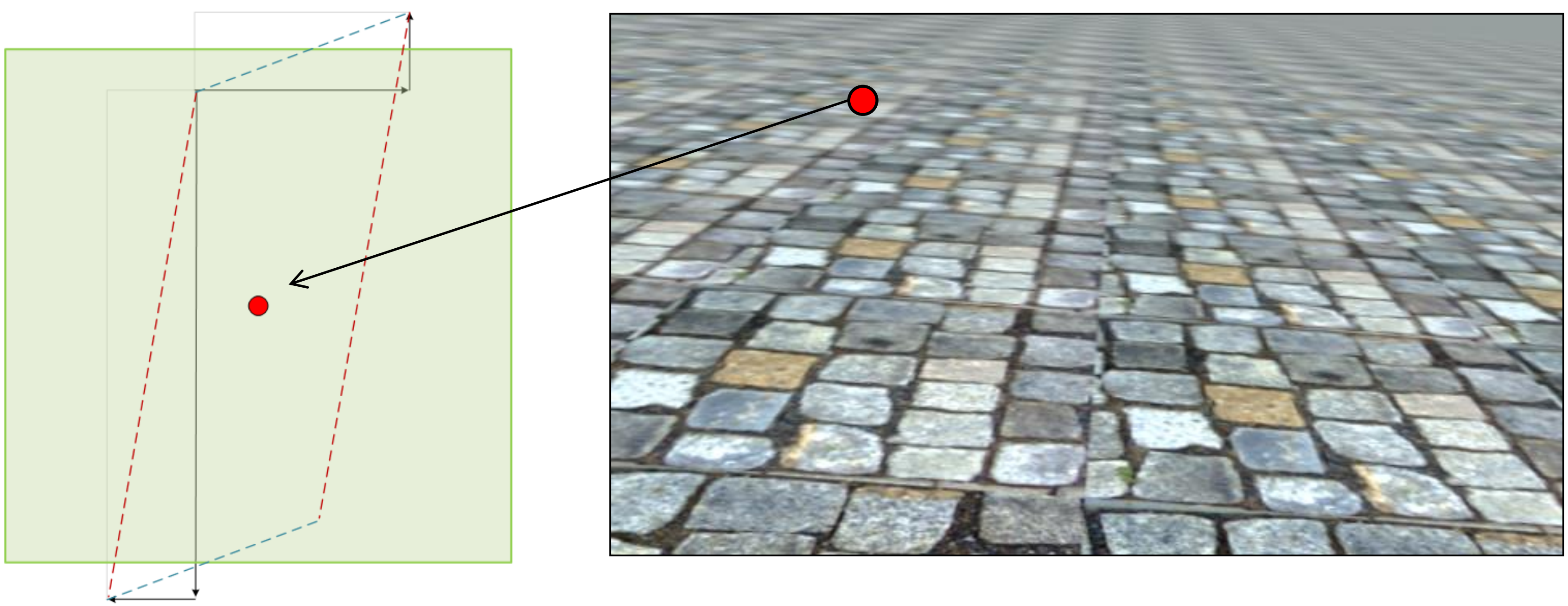
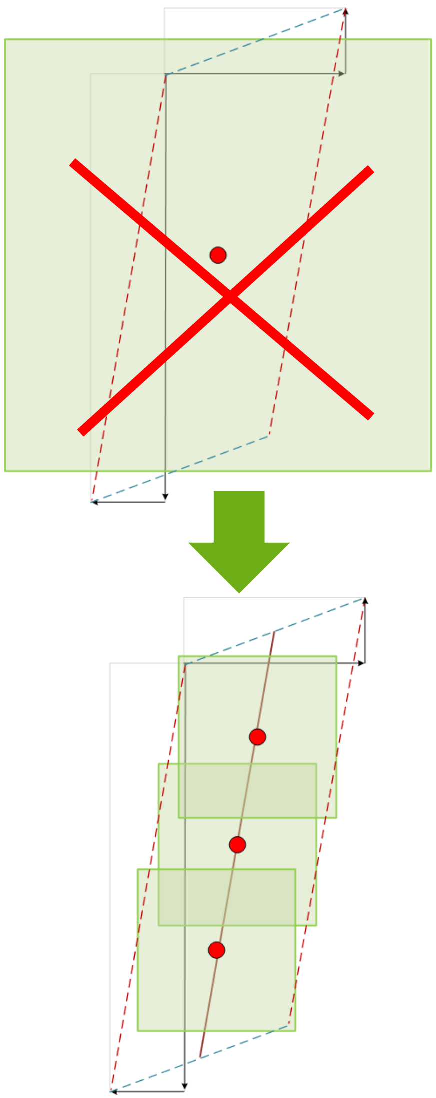


Image source: Wikipedia

Practical Anisotropic Filtering (1)

- Replace the single uniformly-sized filter with multiple smaller mipmap samples
- Approximate anisotropy by aligning the taps along the longest texture gradient axis
- Implemented in graphics harware
  - Maximum taps determined by the "maximum anisotropy" level



Practical Anisotropic Filtering (2)

$$P_x = \sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2} \qquad P_y = \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}$$

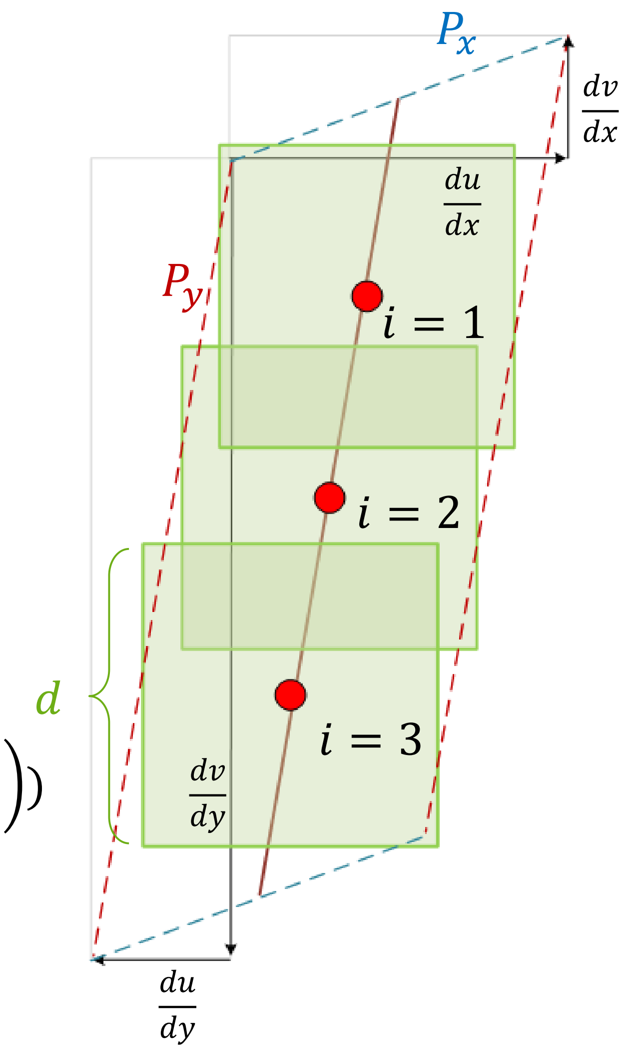
$$P_{max} = \max\{P_x, P_y\} \qquad N = \min\{a_{max}, \left\lceil \frac{P_{max}}{P_{min}} \right\rceil\}$$

$$P_{min} = \min\{P_x, P_y\} \qquad d = \frac{P_{max}}{N}$$

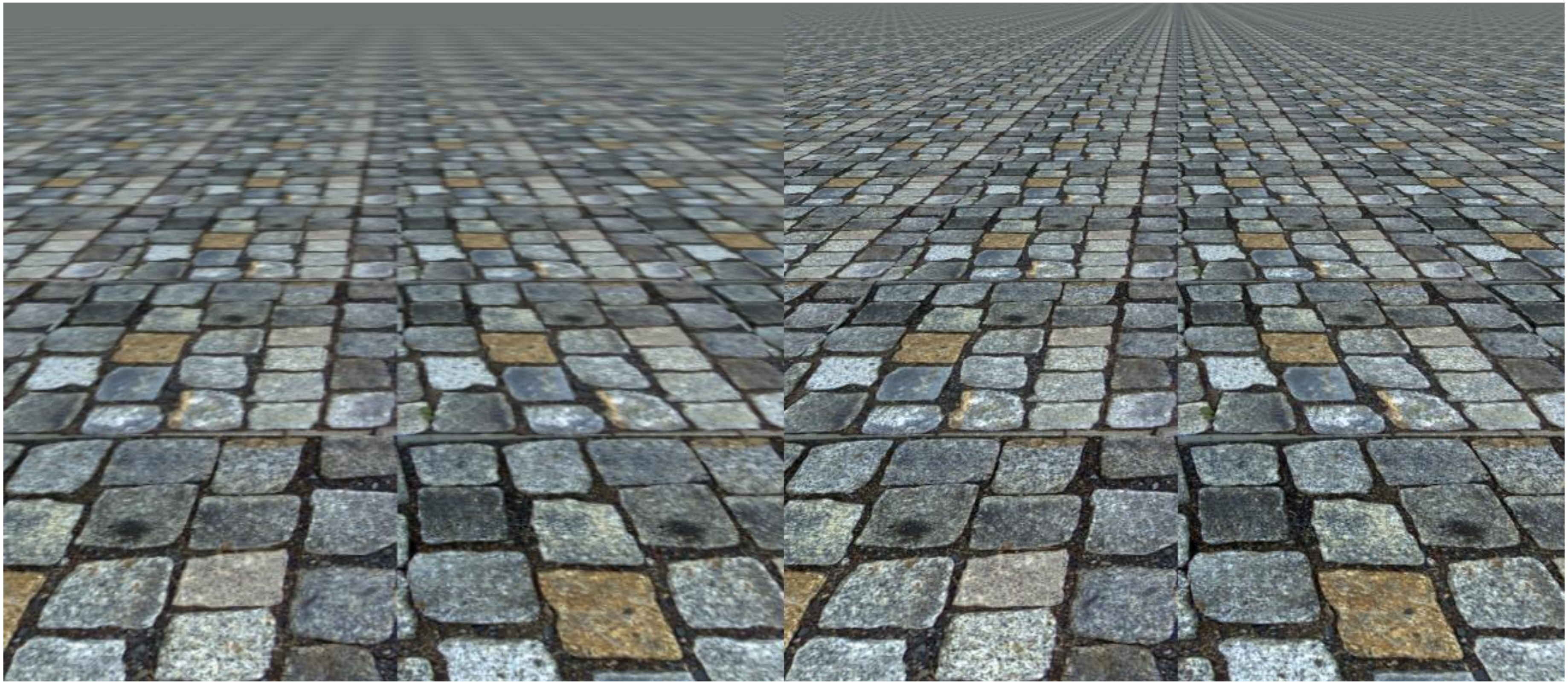
$$Sample(u, v) = \frac{1}{N}\sum_{i=1}^{N} \tau_i(u, v)$$

$$\tau_i(u, v) = \tau(u + \frac{du}{d\xi}\left(\frac{i}{N+1} - 0.5\right), v + \frac{dv}{d\xi}\left(\frac{i}{N+1} - 0.5\right))$$

$$\xi = \begin{cases} x & P_x > P_y \\ y & otherwise \end{cases}$$



Practical Anisotropic Filtering (3)



Isotropic filtering
Linear MIP map selection

Anisotropic filtering
Linear MIP map selection

TEXTURE PROJECTION FUNCTIONS

(Texture) Mapping Functions

- Mapping functions are used to apply two-dimensional textures to surfaces
- They define a transformation of a 3D to a 2D texturing coordinate pair
- The can be used for the automatic generation of texture coordinates
  - Typically used by artists as a rough uv pair assignment to vertices
  - Can be used at run-time from within shaders for automatic texture coordinate assignment

Linear Mapping - Planar Projection (1)

- In general, a linear mapping binds a texture parameter to a direction vector in 3D space and can be written in the form:

$$\mathbf{s} = \mathbf{T}_{Linear} \cdot \mathbf{p} \quad \text{or equivalently} \quad (u, v) = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

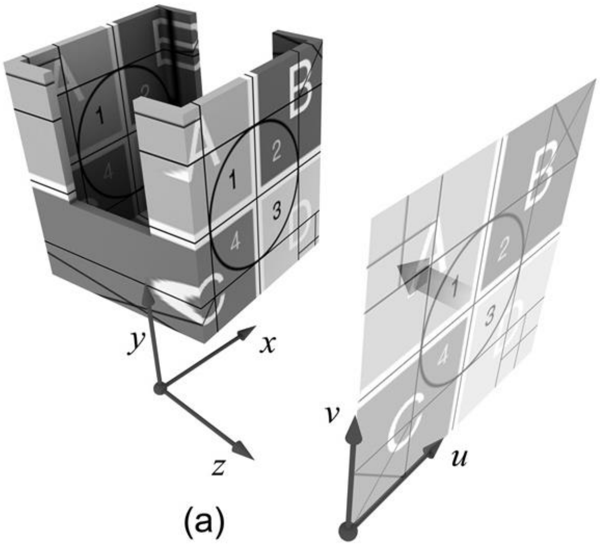
- This simple transformation collapses points onto a plane embedded in 3D

Linear Mapping - Planar Projection (2)

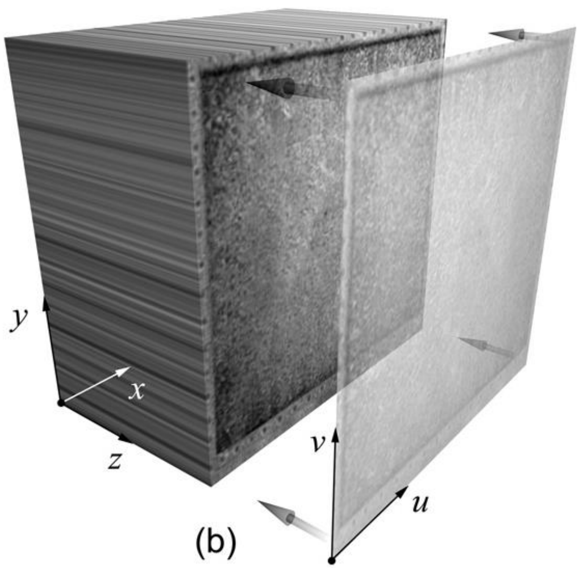
- Simple, common example: Projection on the xy plane (a)

$$(u, v) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- The z coordinate is collapsed: All points with the same x, y coordinates have equal texture coordinates regardless of z (b)



(a)



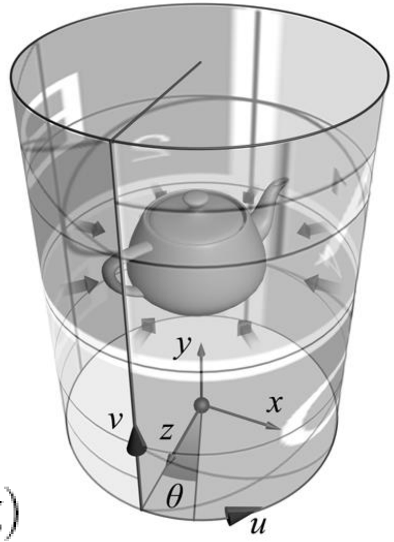
(b)

Cylindrical Mapping

- It is essentially the conversion of the Cartesian coordinates to cylindrical ones. Typically, the radius (distance to the cylinder axis) is ignored

$$\theta = \tan^{-1}(x/z),$$
$$h = y,$$
$$r = \sqrt{(x^2+z^2)},$$

$$u = \frac{1}{2} + \frac{\theta}{2\pi} = \frac{1}{2} + \frac{\tan^{-1}(x/z)}{2\pi},$$
$$v = h = y.$$



Spherical Mapping (1)

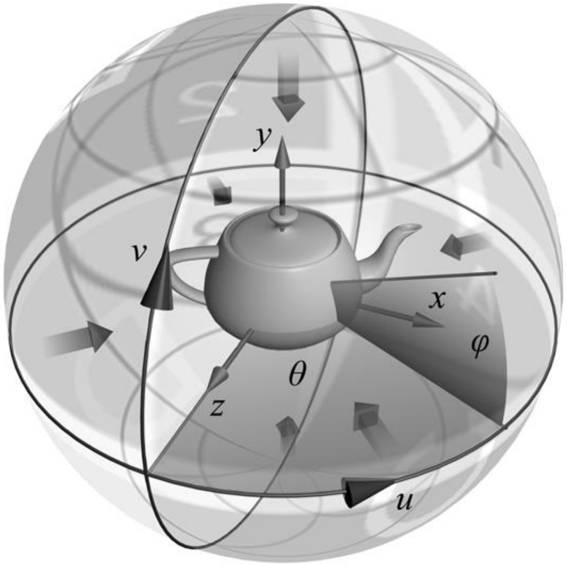
- It is essentially the conversion of the Cartesian coordinates to spherical ones. Typically, the radius (distance to the cylinder axis) is ignored

$$\theta = \tan^{-1}\left(\frac{x}{z}\right) \qquad -\pi < \theta \leq \pi$$

$$\varphi = \tan^{-1}\left(\frac{y}{\sqrt{x^2+z^2}}\right) \qquad -\frac{\pi}{2} < \varphi \leq \frac{\pi}{2},$$

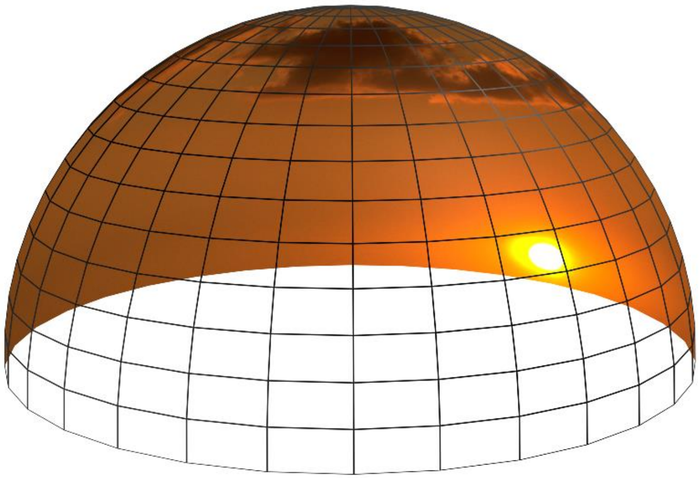
$$r = \sqrt{x^2+y^2+z^2}.$$

$$u = \frac{\theta+\pi}{2\pi}, \qquad v = \frac{\varphi+\pi/2}{\pi}.$$

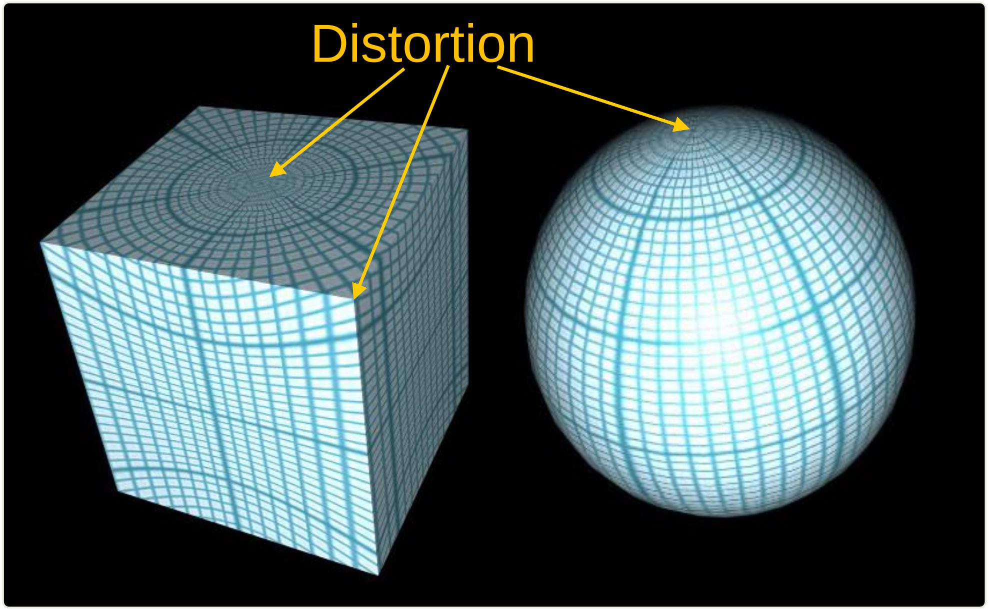


Spherical Mapping (2)

- Quite useful mapping, especially for environment effects and spherical objects

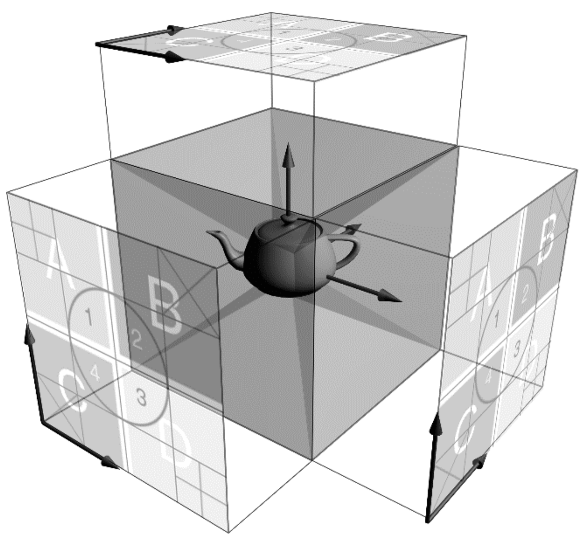


Spherical mapping inherently suffers from distortions at the poles (singularity) and variable tex. coord density



A Practical Note

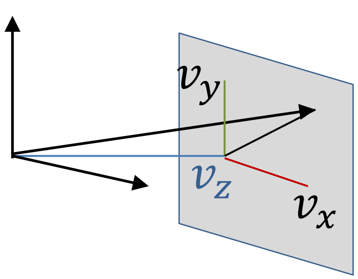
- Mapping functions can be applied in isolated surface groups to optimize coverage, uniformity and smoothness
- For example, in the figure below, all surfaces parallel to a primary plane use this plane for linear mapping:



Projective Mapping (1)

- Consider a vector $\mathbf{v} = (v_x, v_y, v_z)$ by either:
  - Connecting a reference center of projection point (e.g. **0**) to a point in space or
  - Using a direction vector (e.g. a surface normal)
- Apply perspective projection to two of its coordinates using the third as projection direction
- Without loss of generality, using the z axis for the projection:

$$(u, v) = (v_x/v_z, v_y/v_z)$$



Projective Mapping (2)

- If we additionally make sure that we project along the maximum vector coordinate (or clamp the fractions to ± 1):

$$(u, v) = \frac{1}{2}\left(\frac{u_c}{m_a} + 1, \frac{v_c}{m_a} + 1\right)$$

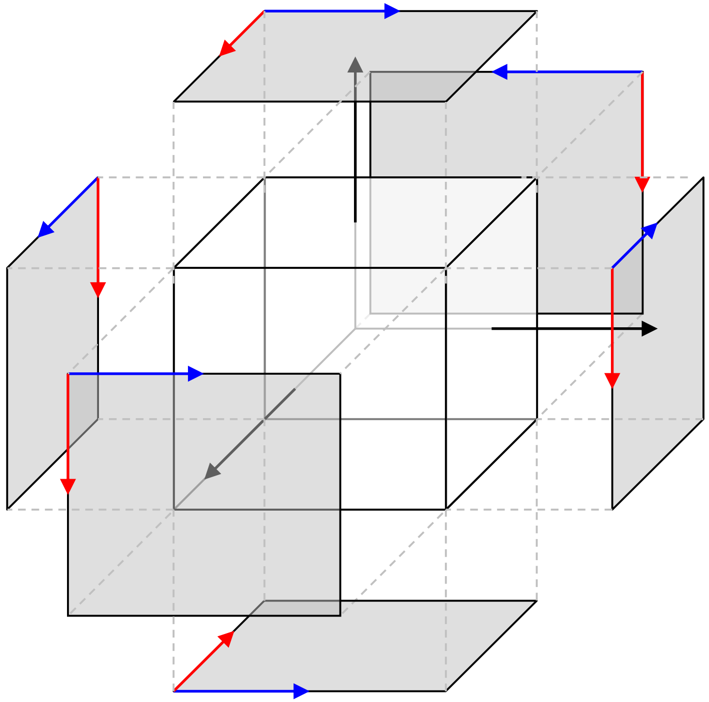
where m_a is the maximum vector coordinate or projection axis, and u_c, v_c the projected coordinates corresponding to u and v

- If a random projection direction is required, we just change the basis of the input vector to the desired one using a rotation (see change of basis transformation)

Projective Mapping – The Cube Map (1)

- An application of projective mapping is the cube map, where a direction is projected on one of the six sides of a cube using the longest half-axis as projection direction
- Each side of the cube uses a separate texture image, so the entire domain of directions can be uniquely mapped to a (stored) value

Projective Mapping – The Cube Map (2)



$$u = \frac{1}{2}\left(\frac{u_c}{|m_a|} + 1\right)$$

$$v = \frac{1}{2}\left(\frac{v_c}{|m_a|} + 1\right)$$

$$(u_c, v_c, m_a) = (-v_z, -v_y, v_x) \quad +x$$

$$(u_c, v_c, m_a) = (v_z, -v_y, v_x) \quad -x$$

$$(u_c, v_c, m_a) = (v_x, v_z, v_y) \quad +y$$

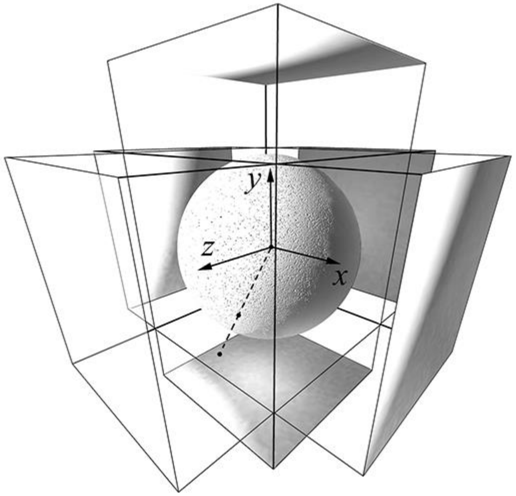
$$(u_c, v_c, m_a) = (v_x, -v_z, v_y) \quad -y$$

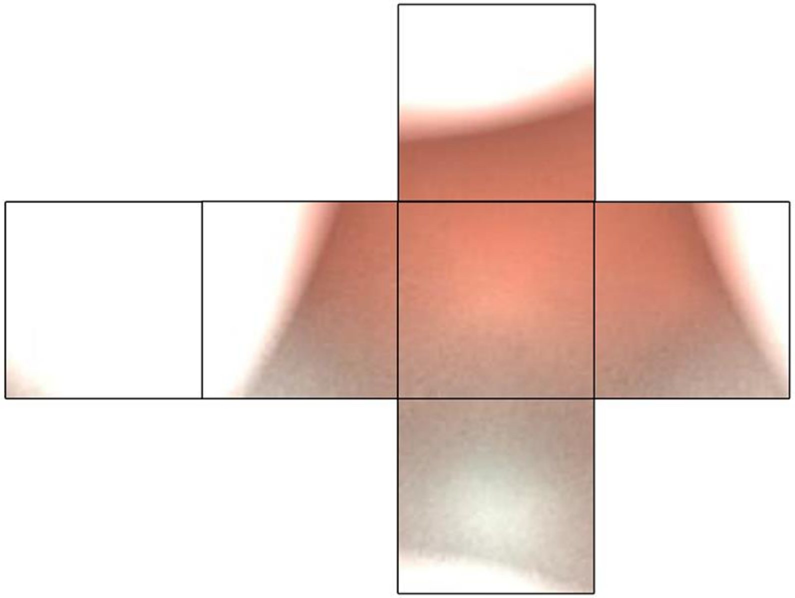
$$(u_c, v_c, m_a) = (v_x, -v_y, v_z) \quad +z$$

$$(u_c, v_c, m_a) = (-v_x, -v_y, v_z) \quad -z$$

Projective Mapping – The Cube Map (3)

- Cube maps are often used for the encoding of incoming luminance from a distant environment (environment mapping)
- Can be used to apply "baked" illumination onto surfaces, such as global illumination

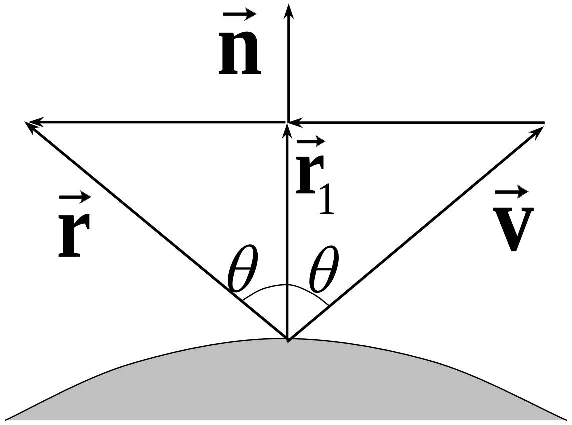




Reflection Mapping (1)

- Reflection mapping is a simple form of environment mapping, where the reflection vector at a shaded point is used to index a spherical environment map representing the surrounding space of an object
  - The map can be a spherical one or a cube map
- Gives the impression that the surface captures the smooth reflection of an environment
- Use the reflection vector:

$$\vec{\mathbf{r}} = 2\vec{\mathbf{r}}_1 - \vec{\mathbf{v}} = 2\vec{\mathbf{n}}\left(\vec{\mathbf{n}}\vec{\mathbf{v}}\right) - \vec{\mathbf{v}}$$



Reflection Mapping (2)

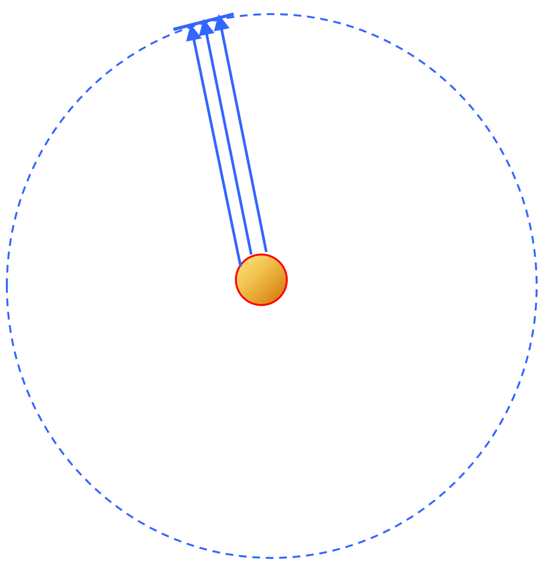
Spherical environment map

No reflection map

With reflection map

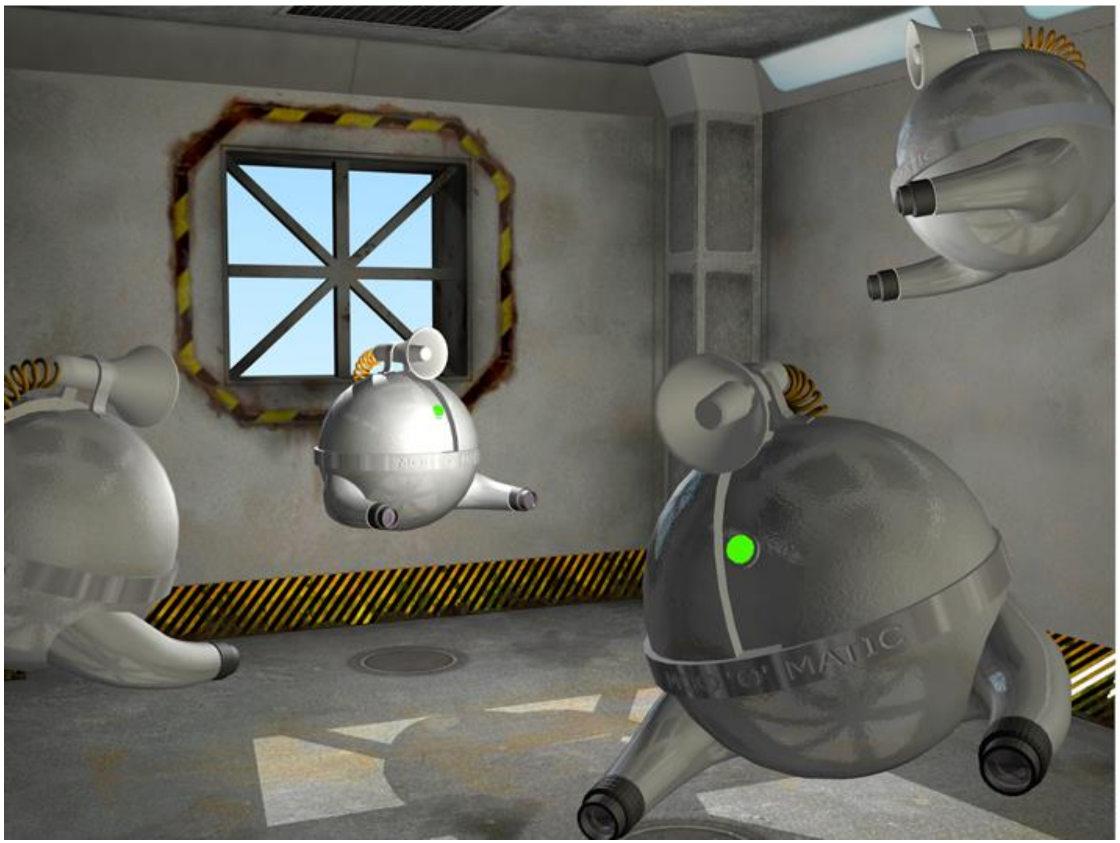
Reflection Mapping (3)

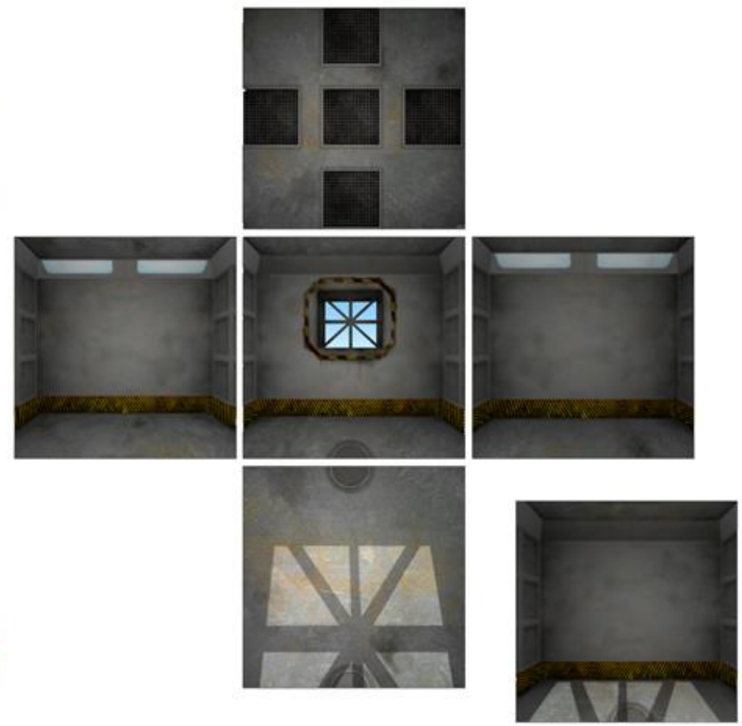
- In general, environment mapping assumes that the incident radiance is coming from distant geometry so that:
- $L_i(\mathbf{p}, \omega) \simeq L_i(\omega), \forall \mathbf{p}$ on the object
- i.e. the scale of the object is small enough (relative to the environment) so that all positions map to the same env. map texel for a given direction ω



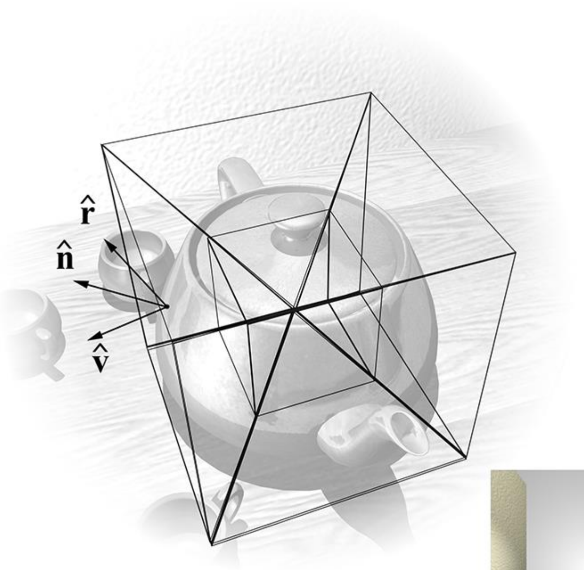
Reflection Mapping with Cube Maps (1)

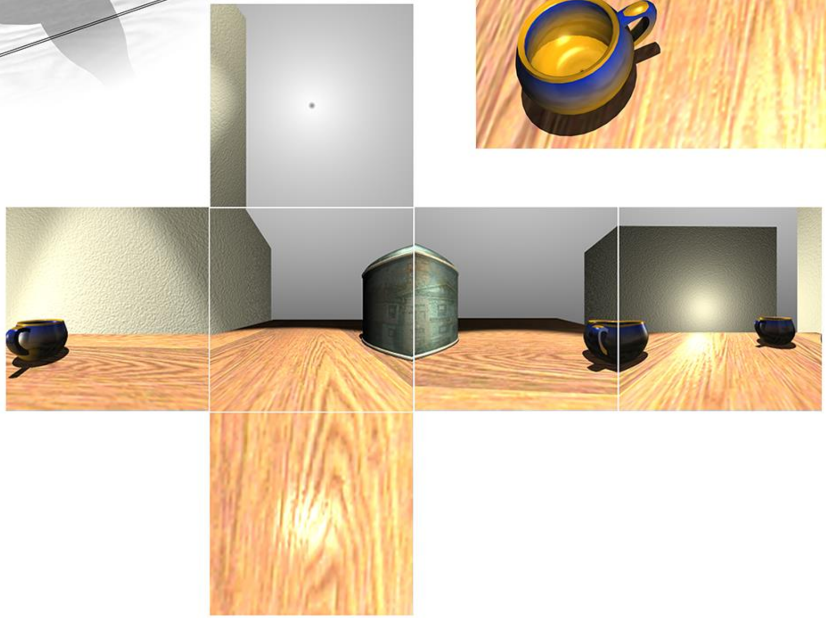
- Same principle, different mapping function
- Environment is recorded in 6 separate projections





Reflection Mapping with Cube Maps (2)





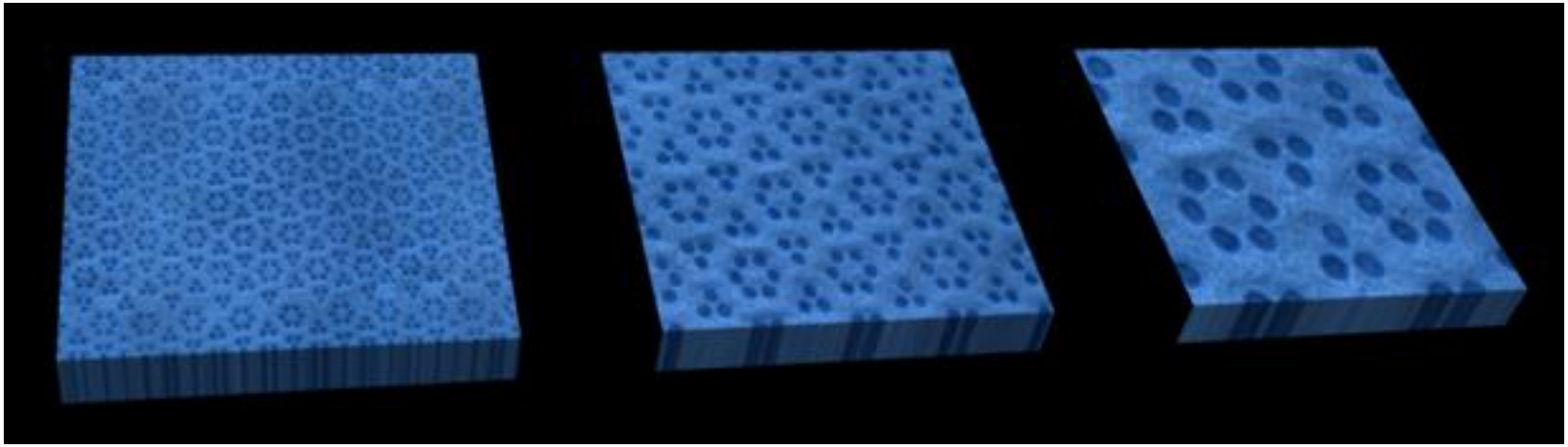
Transforming Texture Coordinates (1)

- Texture coordinates describe a domain, in which all known (linear) transformations apply, as usual. So we can:
  - Translate (offset)
  - Rotate
  - Scale

  texture coordinates to modify the appearance of models

Transforming Texture Coordinates (2)

- In the following example scaling is applied in texture space coordinates to magnify the texture appearance:
- The texture coordinates are successively scaled by 1/3 to achieve a magnification of 3



PROCEDURAL TEXTURING

Procedural Textures (1)

- A surface or volume attribute can be:
  - Calculated from a mathematical model
  - Derived in a procedural algorithmic manner
- Procedural Texturing:
  - Does not use intermediate parametric space
  - Often referred to as “procedural shaders”
- Can be used to calculate:
  - A color triplet
  - A normalized set of coordinates
  - A vector direction
  - A scalar value

Procedural Textures (2)

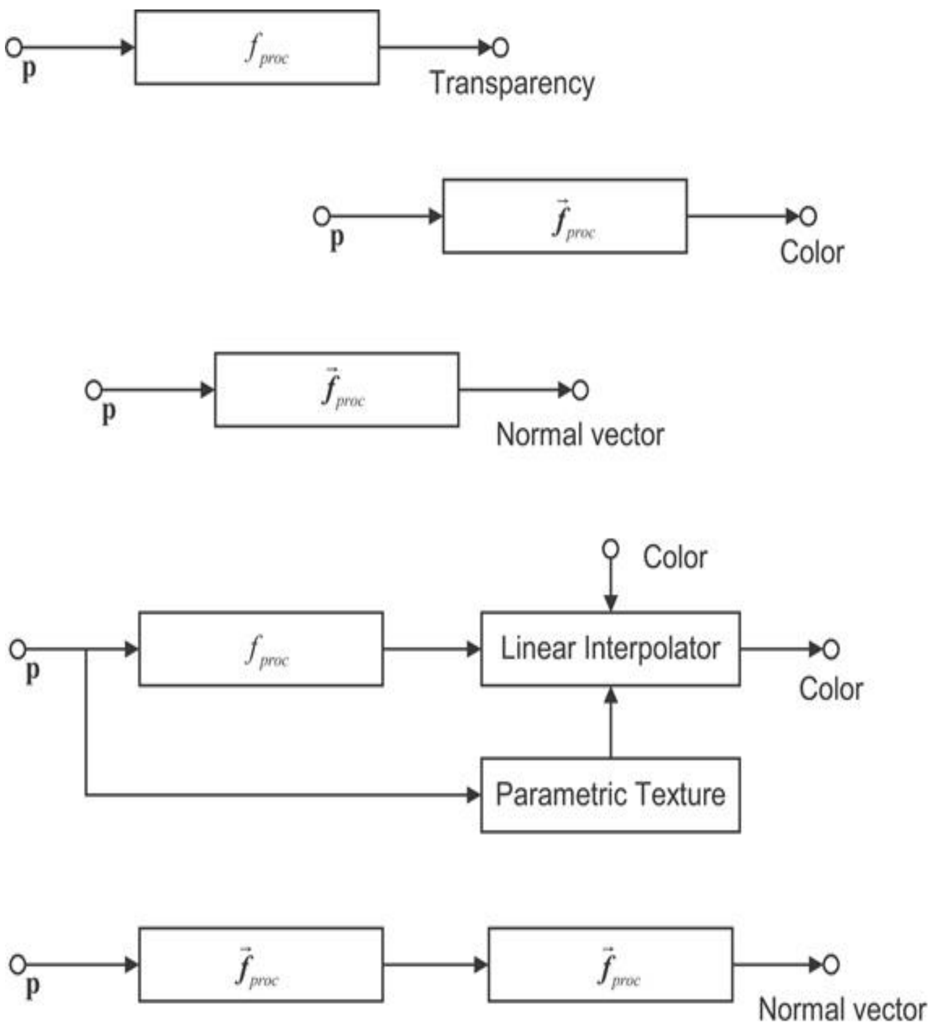
- Some forms of a procedural texture:

$$\mathbf{v} = \mathbf{f}_{\mathrm{proc}}(\mathbf{p},\mathbf{a}),$$

$$\vec{\mathbf{n}} = \vec{\mathbf{f}}_{\mathrm{proc}}(\mathbf{p},\mathbf{a}),$$

$$t = f_{\mathrm{proc}}(\mathbf{p},\mathbf{a})$$

- These output parameters can be used as:
  – Input to another procedural texture
  – A mapping function to index a texture image



Properties of Procedural Textures

- Continuous input parameters and continuous output
- No magnification artifacts
- No distortion due to parametric mapping issues
- Map the entire input domain to the output domain

- Due to lack of local control (something that texture images provide), we often combine procedural and image texturing

Procedural Textures: Example

Noise

- In nature there are materials and surfaces with irregular patterns, such as a rough wall, a patch of sand, various minerals, stones etc.
- A procedurally generated noise texture should:
  - Act as a pseudo-number generator
  - Have some controllable properties
  - Ensure a consistent output

Procedural Noise Properties (1)

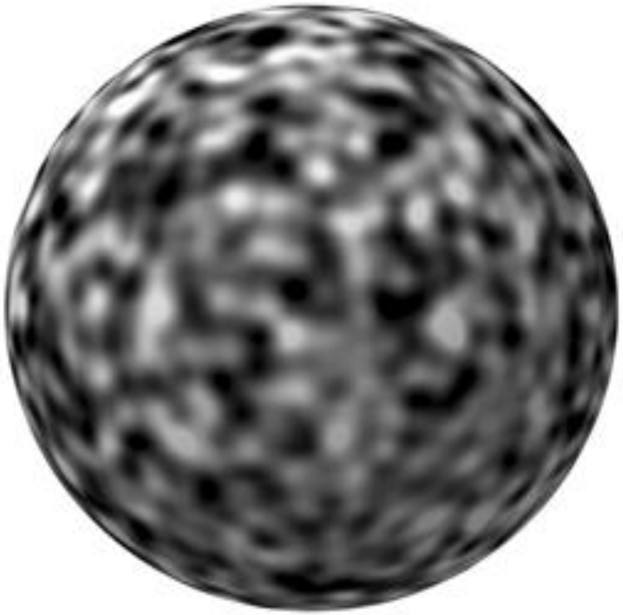
- Stateless
  - The procedural noise model must be memory-less
  - The new output should not depend on previous stages or past input values
  - Necessary if we want an uncorrelated train of outputs
- Time-invariant
  - The output has to be deterministic
  - Avoid dependence of the noise function on clock-based random generators

Procedural Noise Properties (2)

- Smooth
  - The output signal should be continuous and smooth
  - First-order derivatives should be computable
- Band-limited
  - A white-noise generator is not useful
  - Should control the max (and min) variation rate of the pattern

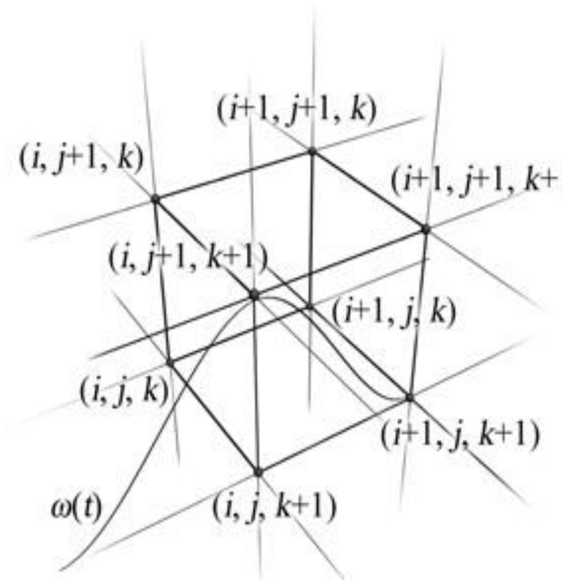
Perlin Noise (1)

- Is the most widely used noise function
- Encompasses all the above properties
- Relies on numerical hashing scheme on pre-calculated random values



Perlin Noise (2)

- Let $\Omega_{i,j,k}$ be a lattice node at integer location (i, j, k) : $i,j,k \in \mathbb{N}$
- We associate a pre-computed random value $\gamma_{i,j,k}$ with each node, deterministically defined w.r.t. (i, j, k)
- The procedural noise output is the weighted sum of the values on the 8 nodes nearest to the input point $\mathbf{p}$



Perlin Noise (3)

- The final noise pattern $f_{\text{noise}}(\mathbf{p})$ for a point $\mathbf{p} = (x, y, z)$
  - is given by trilinear interpolation of the values $\gamma_{i,j,k}$ of the 8 lattice points $\Omega_{i,j,k}$ closest to $\mathbf{p}$
  - use $\omega(t - \lfloor t \rfloor)$ as the interpolation coefficient, where:

$$\omega(t) = \begin{cases} 2|t|^3 - 3t^2 + 1 & |t| < 1, \\ 0 & |t| \geq 1 \end{cases}$$

- This function has a support of 2, centered at 0
- $\omega(t-i)$ is max at i and drops off to 0 beyond $i \pm 1$

Turbulence (1/f noise function) (1)

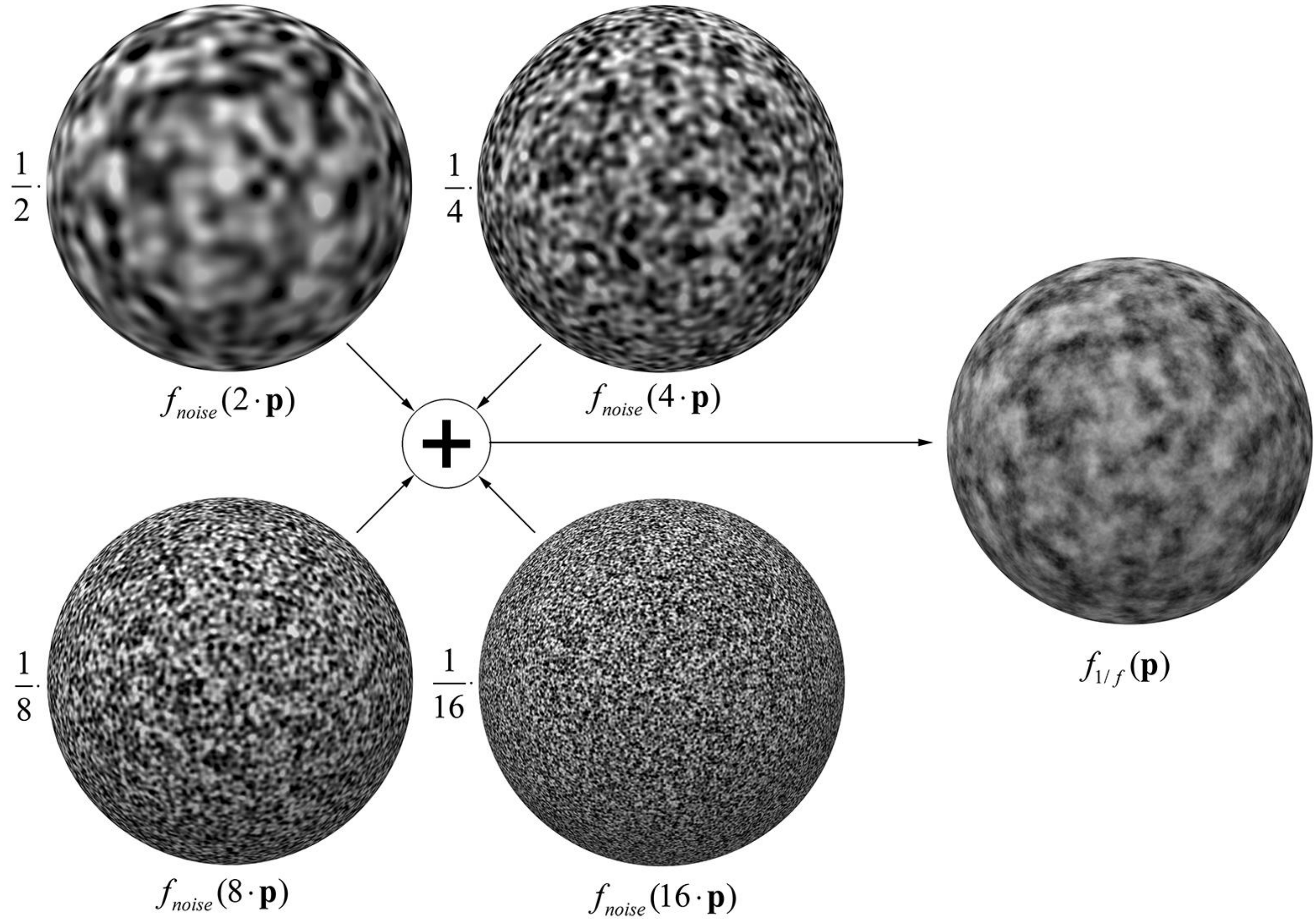
- An extension of the noise procedural texture
- Band-limited noise function
- Has a spectrum profile whose magnitude is inversely proportional to the corresponding frequency (hence the 1/f name)
- Overlays suitably scaled harmonics of a basic band-limited noise function:

$$f_{\mathrm{turb}}(\mathbf{p}) = f_{1/f}(\mathbf{p}) = \sum_{i=1}^{\mathrm{octaves}} \frac{1}{2^i f} f_{\mathrm{noise}}(2^i f \cdot \mathbf{p})$$

where f: the base frequency of the noise

$octaves$: the max number of overlaid noise signals

Turbulence (1/f noise function) (2)



Turbulence (1/f noise function) (3)

- Many interesting patterns can be generated by:
  - Adding a bias to the input points of another procedural or parametric texture
  - Using it as part of a composite texture function

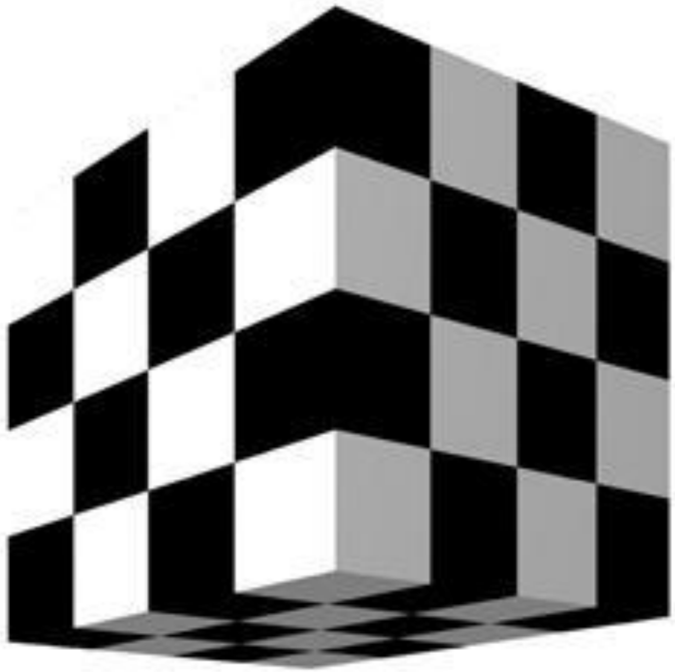
$$
\begin{aligned}
f_{\text{proc}}(\mathbf{p}) &= f_{\text{math}}\left(f_{\text{turb}}(\mathbf{p})\right), \\
f_{\text{proc}}(\mathbf{p}) &= f_{\text{math}}(\mathbf{p} + a \cdot \vec{f}_{\text{turb}}(\mathbf{p}))
\end{aligned}
$$

- Natural formations can often be achieve with combinations of:
  - A base mathematical expression
  - Turbulence
  - noise

Common Procedurals: Checker

- Interleaved solid blocks of 2 different values
  - Using a texture image at an arbitrary resolution would blur or pixelize the transitions

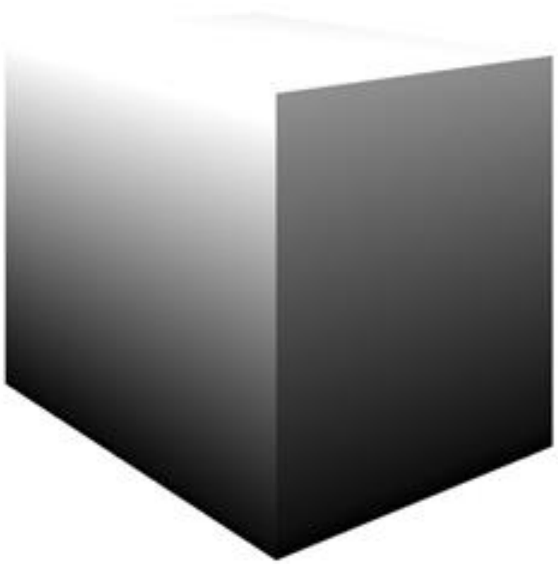
$$f_{\text{checker}}(\mathbf{p}) = \left(\lfloor x \rfloor + \lfloor y \rfloor + \lfloor z \rfloor\right) \bmod 2$$



Common Procedurals: Gradient

- Produces a high-fidelity smooth transition from one value to another
  - There is no danger of generating perceivable bands
- Can use many alternative input parameters, i.e. Cartesian coordinates, spherical parameters etc.
- Its simplest form: a (tiled) ramp along a primary axis:

$$f_{\text{gradient}}(\mathbf{p}) = y - \lfloor y \rfloor$$

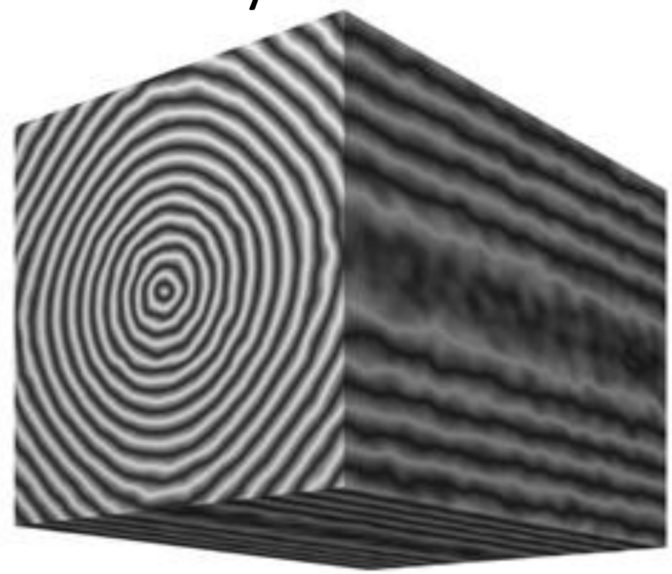


Common Procedurals: Wood

- Represented as an infinite succession of concentric cylindrical layers
  - Modeled by a ramp function over the cylindrical coordinate r
  - Add an amount of perturbation a to the input points
  - Use an absolute sine or cosine function to accent the sharp transition between layers without discontinuity:

$$f_{\text{wood}}(\mathbf{p}) = \left|\cos\left(2\pi\left(d - \lfloor d \rfloor\right)\right)\right|,$$

$$d = \sqrt{y^2 + z^2} + a \cdot f_{\text{turb}}(\mathbf{p})$$

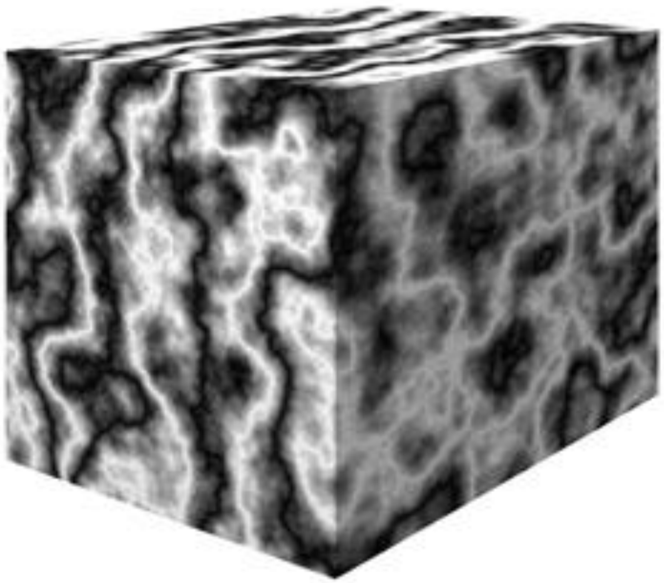


Common Procedurals: Marble

- Use a smoothly varying function to generate the compressed earth layers
- Perturb the input parameters to get a very realistic approximation

$$f_{\text{marble}}(\mathbf{p}) = \frac{1}{2} + \sin\left(2\pi\left(x + f_{\text{turb2}}(\mathbf{p})\right)\right),$$

$$f_{\text{turb2}} = \sum_{i=1}^{\text{octaves}} \frac{1}{2^i f}\left| f_{\text{noise}}\left(2^i f \cdot \mathbf{p}\right)\right|$$



SURFACE RELIEF REPRESENTATION

Using Textures to Mimic Surface Detail (1)

- Usually, it is very inefficient to represent every surface detail using geometry
- We typically resolve to using textures to mimic the appearance of complex relief patterns
- The impression of surface detail is created via the interaction of surface gradient with light
- We either "bake" lit geometry appearance on textures or use properly lit photos

Using Textures to Mimic Surface Detail (2)

- In an environment with changing lighting conditions this can be very problematic
- Baked surface appearance cannot match the lighting conditions
  - Change of emission direction
  - Shadows
  - Light color
  - …
- So the effect breaks

Representing Relief with Texturing

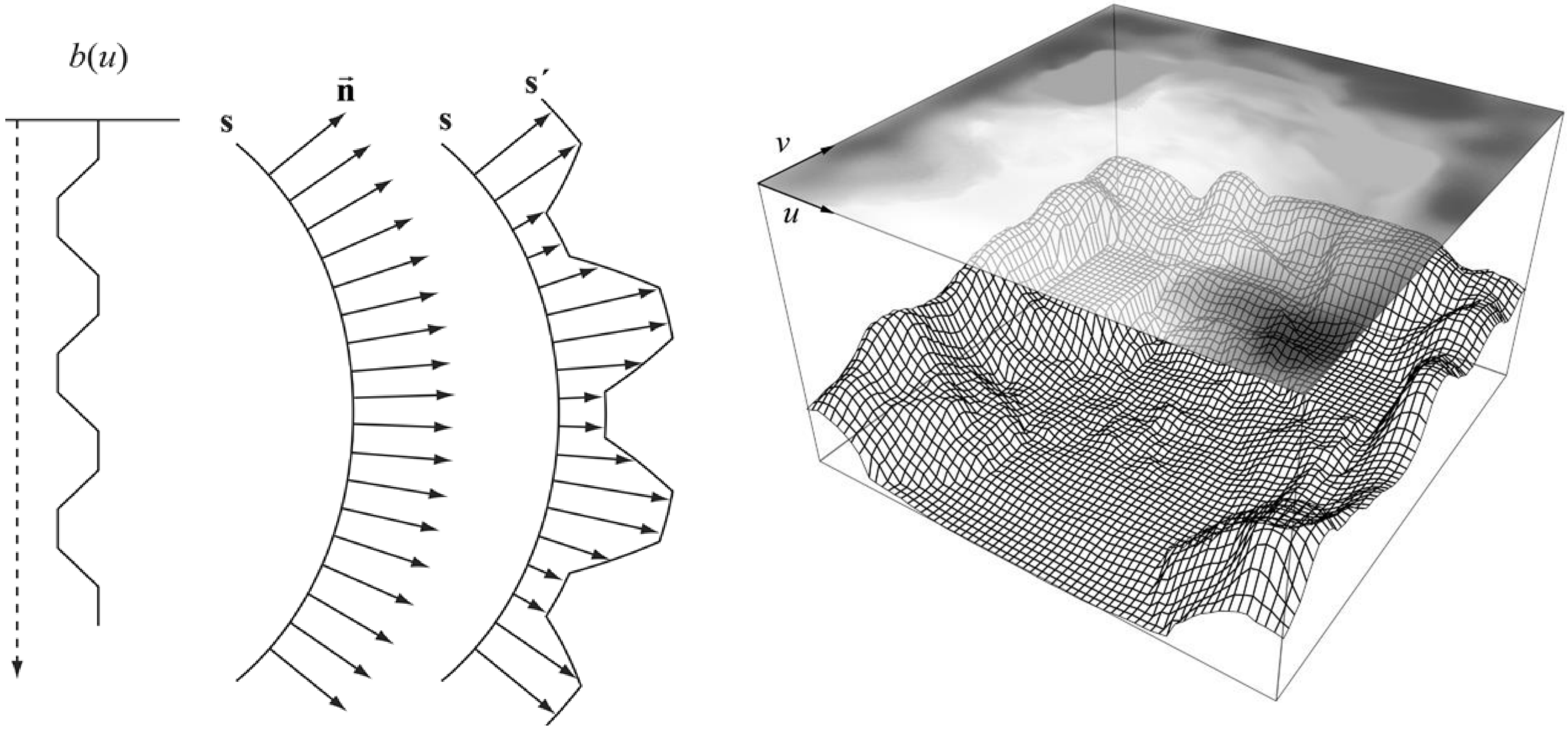
We can use texturing to:

- Locally offset the vertices of an object according to a relief (bump) map → Displacement Mapping
  - The geometry can be subdivided further prior to offsetting the vertices
- Locally modify attributes of the surface in order to give the illusion of complex geometric structure, without actually generating the surface detail:
  - Bump Mapping
  - Normal Mapping
  - Parallax Occlusion Mapping

Displacement Mapping (1)

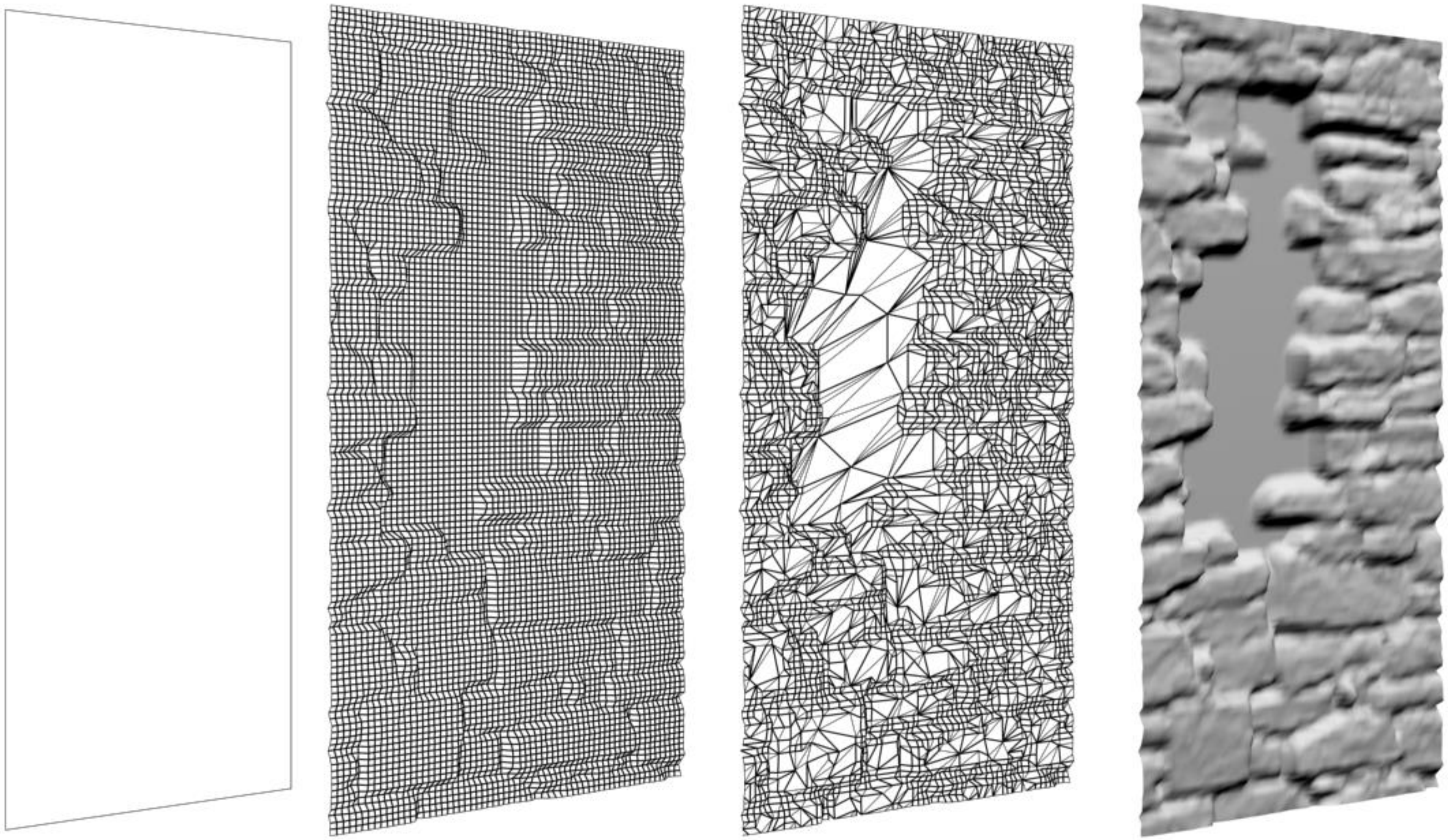
- Move vertices along the normal according to elevation values:

$$\mathbf{s}' = \mathbf{s} + \vec{\mathbf{n}} \cdot offset_{MAX} \cdot b(u,v)$$



Displacement Mapping (2)

- Requires adequately tessellated surface geometry, comparable to the scale of texture relief variation



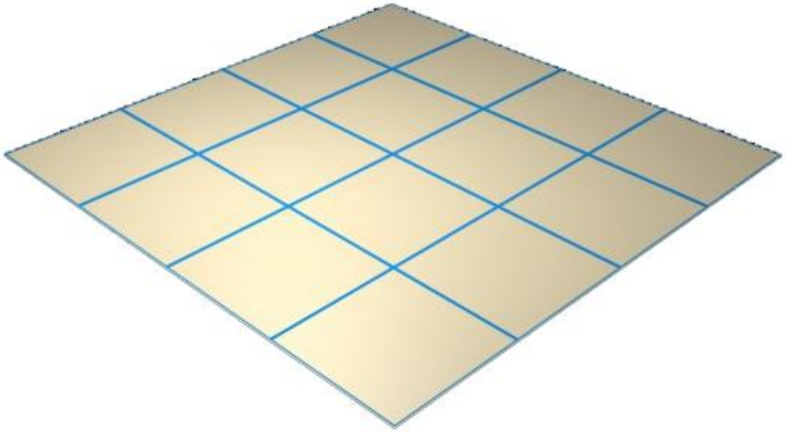
Original Uniform subdivision Adaptive subdivision Shaded result

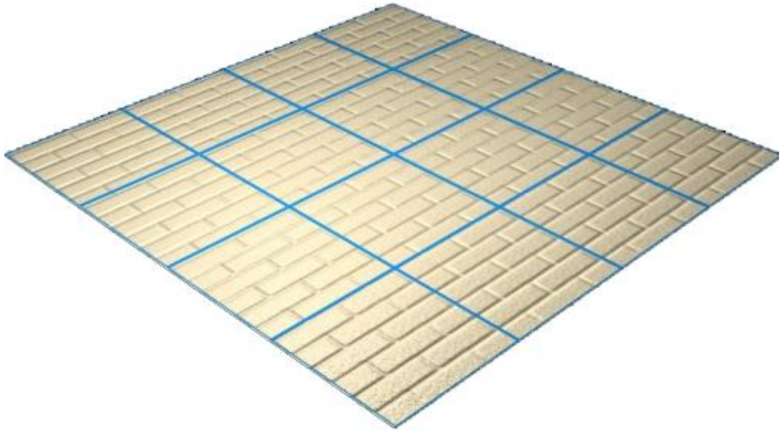
Displacement Mapping in Real Time

- Displacement mapping is typically used in offline rendering
- In real-time applications, it is expensive to use, even with GPU tessellation, but is useful when:
  - We zoom on detailed surfaces
  - View relief patterns at oblique angles or elevation differences are large: We expect to see self occlusion caused by the relief pattern

Normal Vector Relief Techniques

- For small-scale surface details, we predominantly perceive the relief pattern due to lighting variations
  - Self-occlusion and/or self-shadowing is more evident only at very oblique angles
- We can effectively and very efficiently “fake” the presence of structural detail by locally modifying the key element in a local shading calculation: the normal vector





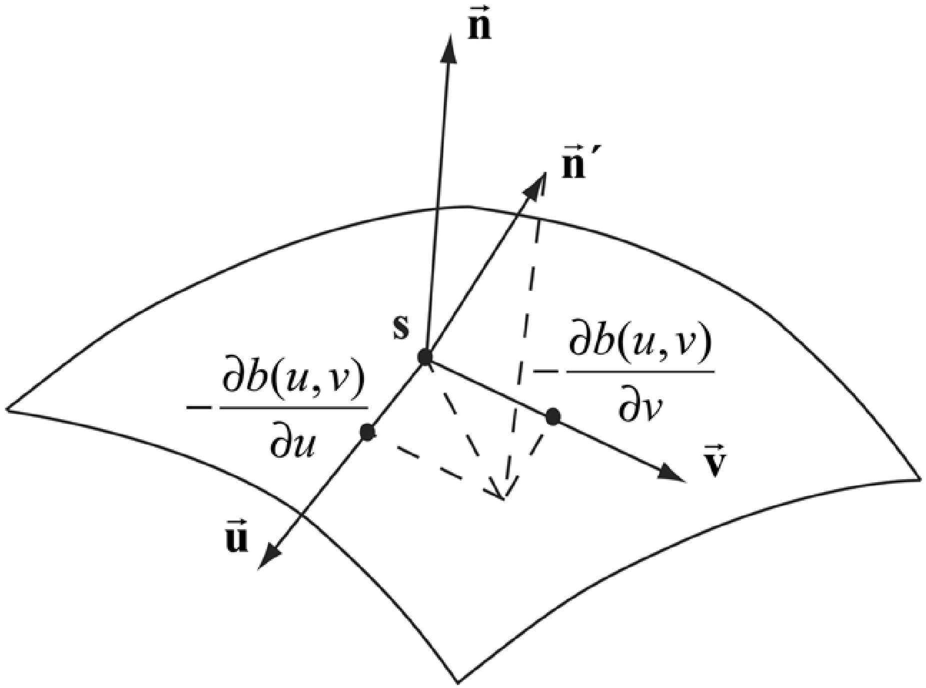
The Local Surface Tangent Space (1)

- At every point on a surface we can define a local coordinate system that is tangential to the surface gradient
  - One axis coincides with the local normal vector
  - The other two axes can coincide with the tangent (and bitangent) surface vectors:

$$\hat{\mathbf{n}} = \hat{\mathbf{u}} \times \hat{\mathbf{v}},$$

$$\hat{\mathbf{v}} = \frac{\overrightarrow{\mathbf{b}}}{|\overrightarrow{\mathbf{b}}|}, \qquad \hat{\mathbf{u}} = \frac{\overrightarrow{\mathbf{t}}}{|\overrightarrow{\mathbf{t}}|},$$

$$\overrightarrow{\mathbf{b}} = \frac{\partial \mathbf{s}(u,v)}{\partial v}. \quad \overrightarrow{\mathbf{t}} = \frac{\partial \mathbf{s}(u,v)}{\partial u},$$



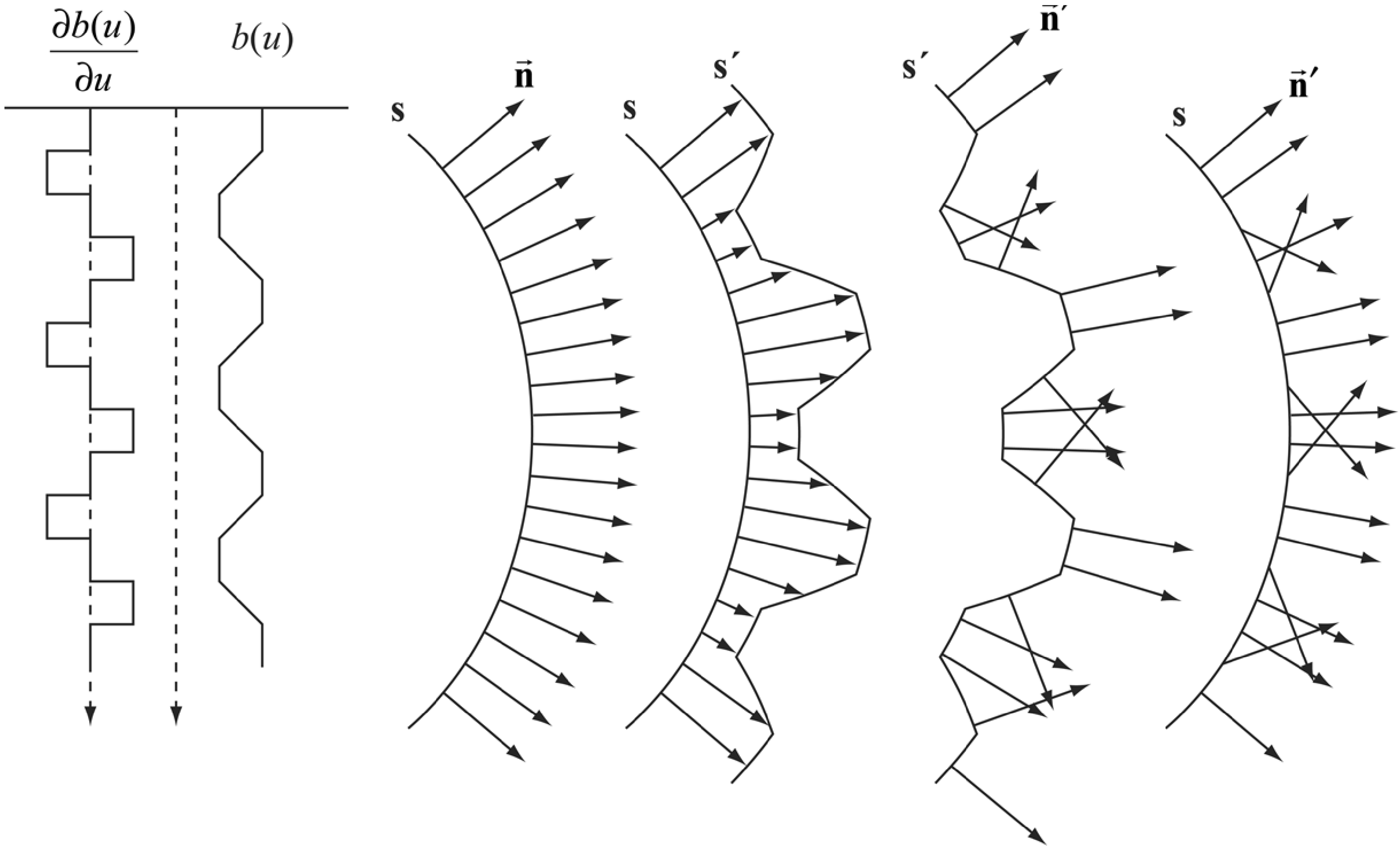
The Local Surface Tangent Space (2)

- We can choose any perpendicular directions on the tangent plane to form a “tangent-space” coordinate system
- It is more convenient if the vectors coincide with the gradient of the surface w.r.t. the texture coordinates
  - We will require in the following to map tangent vectors to texture map gradients

Bump Mapping (1)

- In the bump mapping technique, we are given a relief pattern as a height field (i.e. texture intensity represents elevation) similar to displacement mapping
- We don’t modify the surface elevation but instead calculate the distorted local normal vectors as if the surface was actually elevated

Bump Mapping (2)



Bump Mapping: Normal Estimation (1)

- If b is the given elevation at texture location (u, v), then the elevated surface should be:

$$\mathbf{s}'(u,v) = \mathbf{s}(u,v) + \hat{\mathbf{n}}(u,v) \cdot b(u,v),$$

- By definition, the normal of the new, elevated position is perpendicular to the tangent vectors at $s'(u, v)$:

$$\hat{\mathbf{n}}' = \hat{\mathbf{u}}' \times \hat{\mathbf{v}}' = \frac{\partial \mathbf{s}'(u,v)}{\partial u} \times \frac{\partial \mathbf{s}'(u,v)}{\partial v}$$

Bump Mapping: Normal Estimation (2)

- To find the new tangent vectors, we calculate the partial derivatives of the elevated point w.r.t. the texture parameters:

$$\mathbf{s}'(u,v) = \mathbf{s}(u,v) + \hat{\mathbf{n}}(u,v) \cdot b(u,v),$$

$$\frac{\partial \mathbf{s}'(u,v)}{\partial u} = \frac{\partial \mathbf{s}(u,v)}{\partial u} + \overbrace{\frac{\partial \hat{\mathbf{n}}(u,v)}{\partial u}}^{\sim 0} \cdot b(u,v) + \hat{\mathbf{n}}(u,v) \cdot \frac{\partial b(u,v)}{\partial u},$$

$$\frac{\partial \mathbf{s}'(u,v)}{\partial v} = \frac{\partial \mathbf{s}(u,v)}{\partial v} + \underbrace{\frac{\partial \hat{\mathbf{n}}(u,v)}{\partial v}}_{\sim 0} \cdot b(u,v) + \hat{\mathbf{n}}(u,v) \cdot \frac{\partial b(u,v)}{\partial v}.$$

Bump Mapping: Normal Estimation (3)

$$\frac{\partial \mathbf{s}'(u,v)}{\partial u} = \frac{\partial \mathbf{s}(u,v)}{\partial u} + \hat{\mathbf{n}}(u,v) \cdot \frac{\partial b(u,v)}{\partial u} = \overrightarrow{\mathbf{t}} + \hat{\mathbf{n}}(u,v) \cdot \frac{\partial b(u,v)}{\partial u},$$

$$\frac{\partial \mathbf{s}'(u,v)}{\partial v} = \frac{\partial \mathbf{s}(u,v)}{\partial v} + \hat{\mathbf{n}}(u,v) \cdot \frac{\partial b(u,v)}{\partial v} = \overrightarrow{\mathbf{b}} + \hat{\mathbf{n}}(u,v) \cdot \frac{\partial b(u,v)}{\partial v}.$$

- And replacing the new tangent vectors in the definition of the new normal we get:

$$\hat{\mathbf{n}}' = \left(\overrightarrow{\mathbf{t}} + \hat{\mathbf{n}} \cdot \frac{\partial b(u,v)}{\partial u}\right) \times \left(\overrightarrow{\mathbf{b}} + \hat{\mathbf{n}} \cdot \frac{\partial b(u,v)}{\partial v}\right)$$

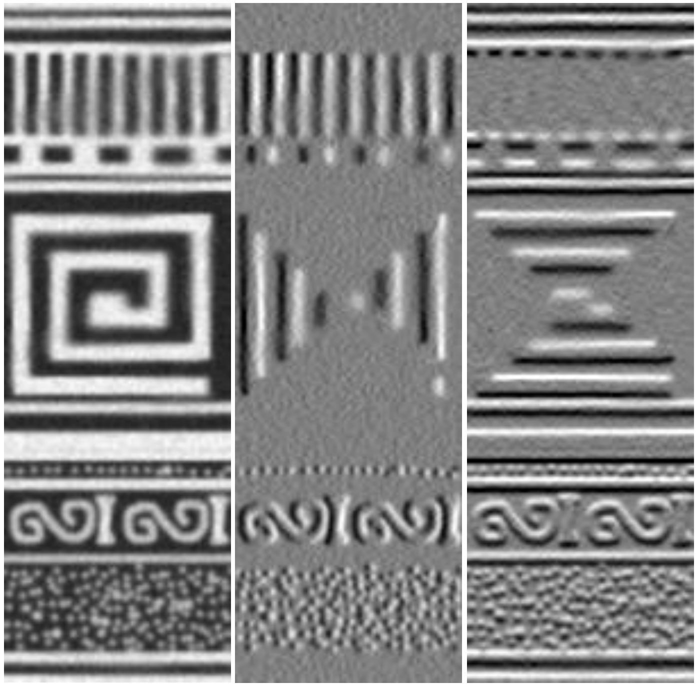
$$= \underline{\overrightarrow{\mathbf{t}} \overset{\hat{\mathbf{n}}}{\times} \overrightarrow{\mathbf{b}}} + \underline{\overrightarrow{\mathbf{t}} \overset{-\hat{\mathbf{b}}}{\times} \hat{\mathbf{n}}} \cdot \frac{\partial b(u,v)}{\partial v} + \underline{\hat{\mathbf{n}} \overset{-\hat{\mathbf{t}}}{\times} \overrightarrow{\mathbf{b}}} \cdot \frac{\partial b(u,v)}{\partial u} + \cancel{\hat{\mathbf{n}} \times \hat{\mathbf{n}}} \cdot \frac{\partial b(u,v)}{\partial u} \frac{\partial b(u,v)}{\partial v}$$

$$0$$

$$\boxed{\hat{\mathbf{n}}' = \hat{\mathbf{n}} - \overrightarrow{\mathbf{b}} \cdot \frac{\partial b(u,v)}{\partial v} - \overrightarrow{\mathbf{t}} \cdot \frac{\partial b(u,v)}{\partial u}.}$$

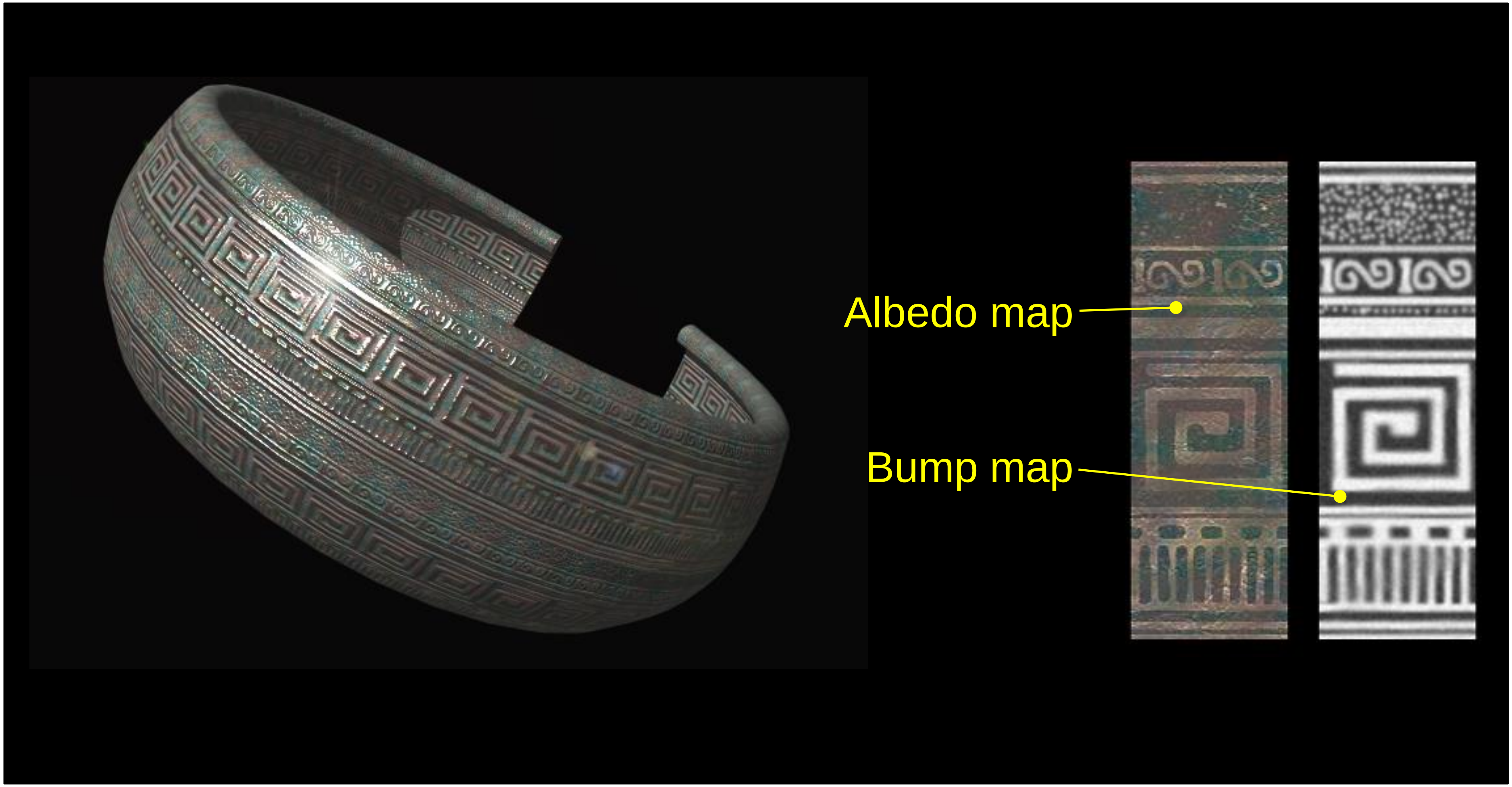
Practical Bump Mapping

- According to the bump mapping calculations, we need:
  – The un-modified normal at the shaded location
  – Two tangent vectors along the u and v parameters
  – The bump map derivatives (can be precalculated and stored in the texture as color channels)



$b(u,v)$ $\quad \dfrac{\partial b(u,v)}{\partial u}$ $\quad \dfrac{\partial b(u,v)}{\partial v}$

Bump Mapping - Results

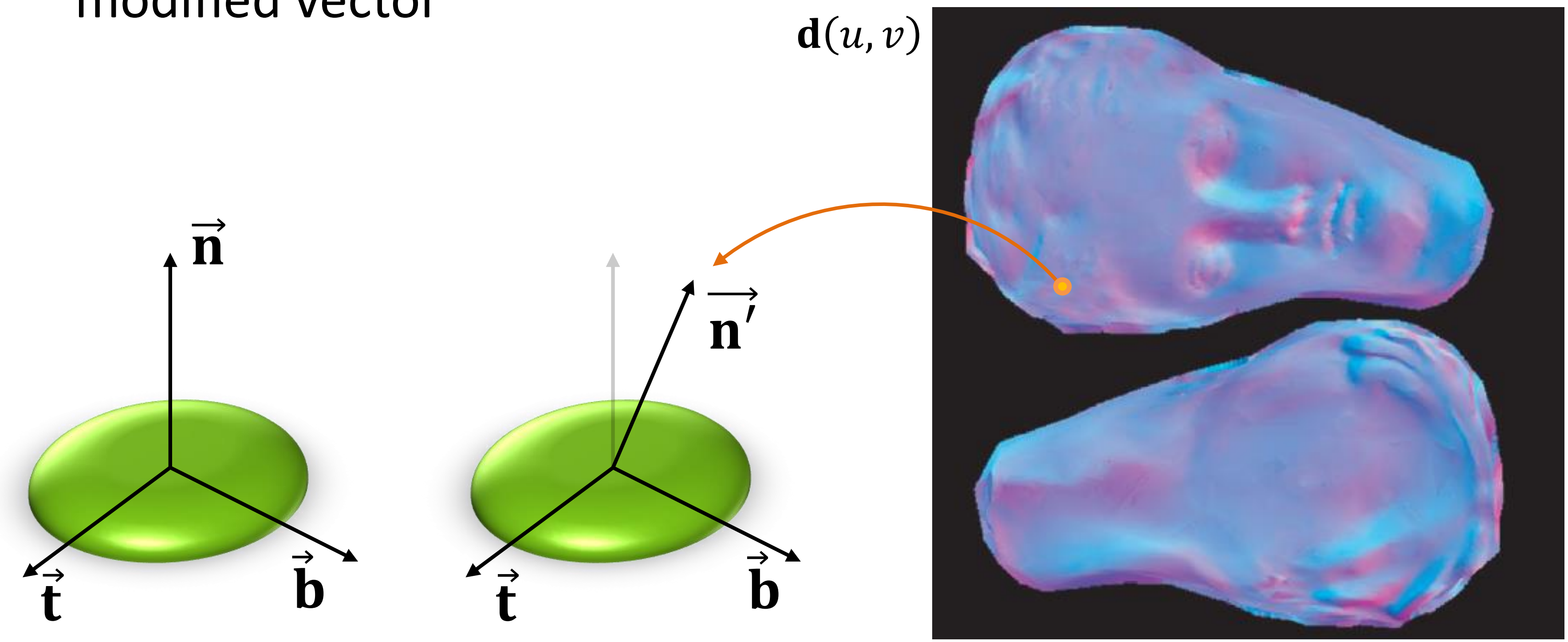


Normal Mapping

- In bump mapping we implicitly find the diverted normal due to the underlying elevation
- Normal mapping dispenses with the calculations by directly replacing the local normal with a new normal vector stored in a texture

Tangent Space Normal Mapping (1)

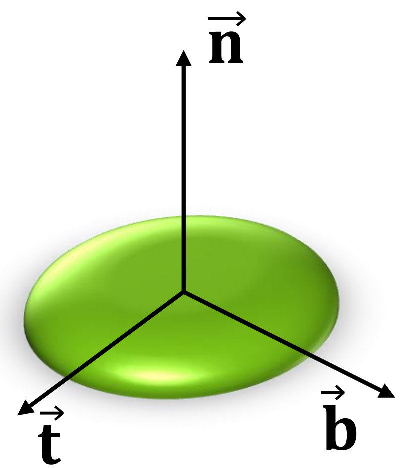
- We directly apply the new tangent-space normal fetched from the texture map
- The texture encodes the tangent space coordinates of the modified vector

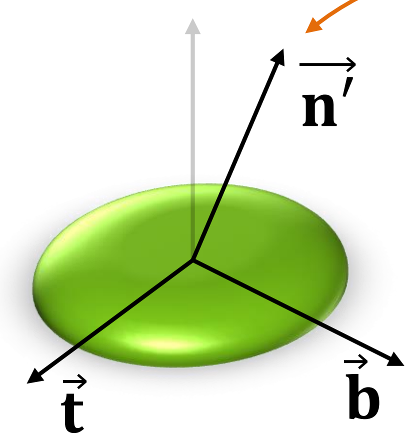


Tangent Space Normal Mapping (2)

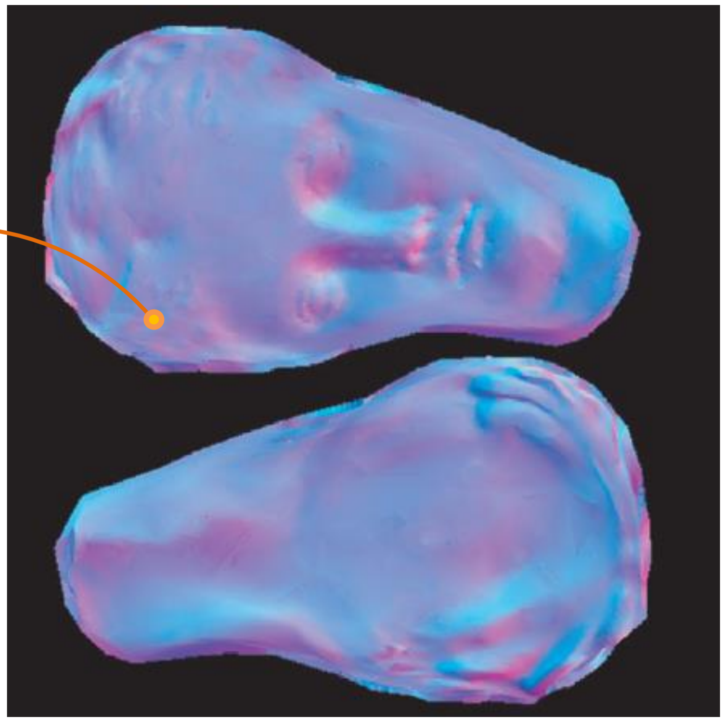
- If $\mathbf{d}(u,v)$ is the local, bent normal direction, and $\vec{\mathbf{n}}, \vec{\mathbf{t}}, \vec{\mathbf{b}}$ are the normal and tangent vectors expressed in any reference frame (e.g. WCS or ECS), then:

$$\overrightarrow{\mathbf{n}'} = d_x\vec{\mathbf{t}} + d_y\vec{\mathbf{b}} + d_z\vec{\mathbf{n}}$$





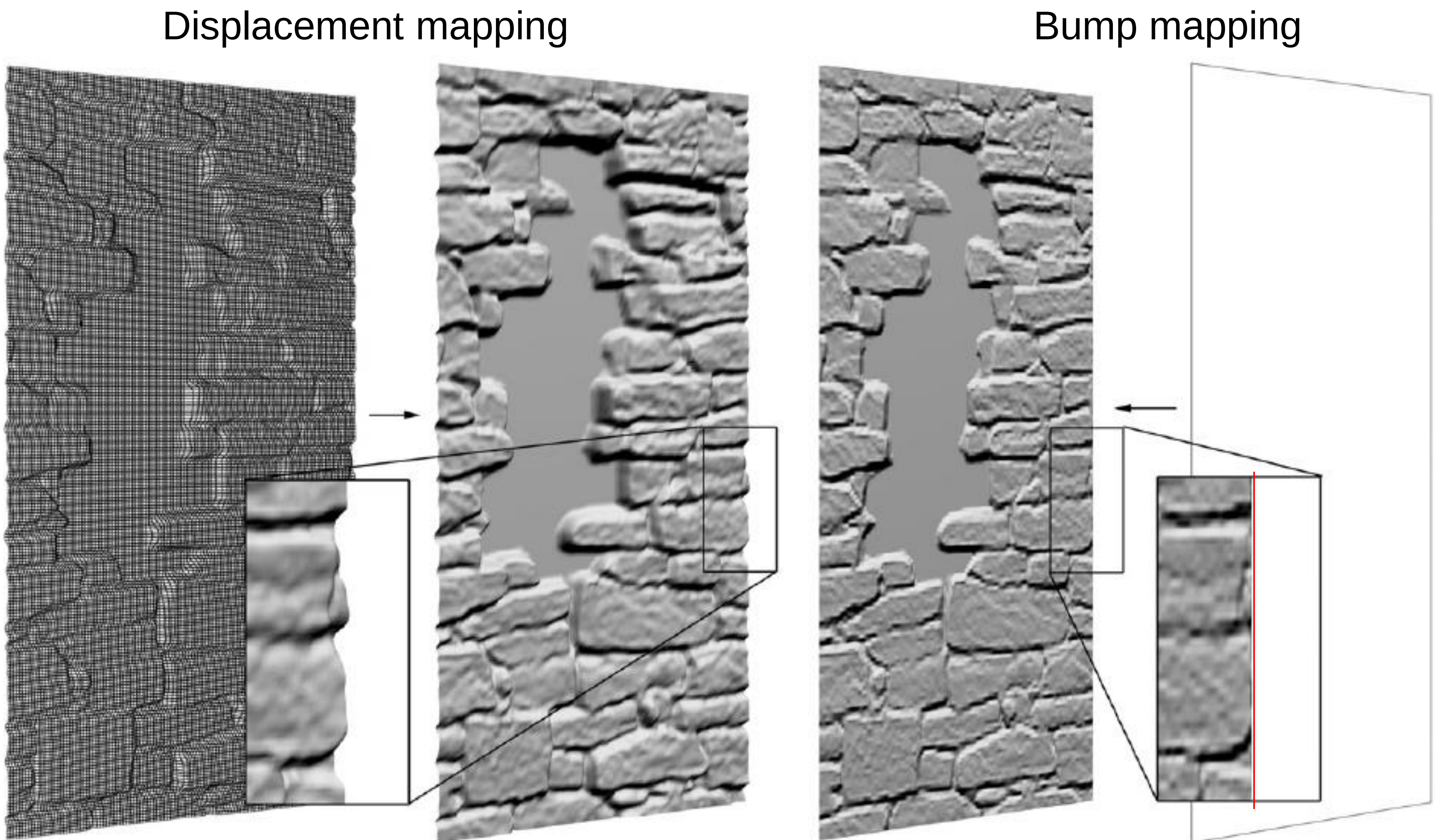
$\mathbf{d}(u,v)$



Practical Shading using Normal Maps

- Calculate and store one tangent vector as additional vertex attribute
- In the vertex shader, calculate and emit the normal and tangent in the same space as the light sources (WCS, ECS), but not in post-projective space
- In the fragment shader:
  - Calculate bitangent via cross product of normal and tangent
  - Fetch new normal from normal map $\mathbf{d}(u, v)$
  - Replace old normal with $\overrightarrow{\mathbf{n}'} = d_x\vec{\mathbf{t}} + d_y\vec{\mathbf{b}} + d_z\vec{\mathbf{n}}$

Normal Vector Techniques - Deficiencies

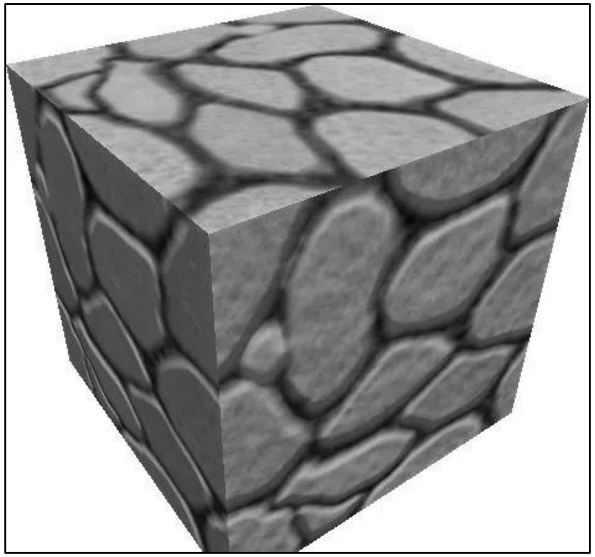


No proper self-occlusion or self-shadow

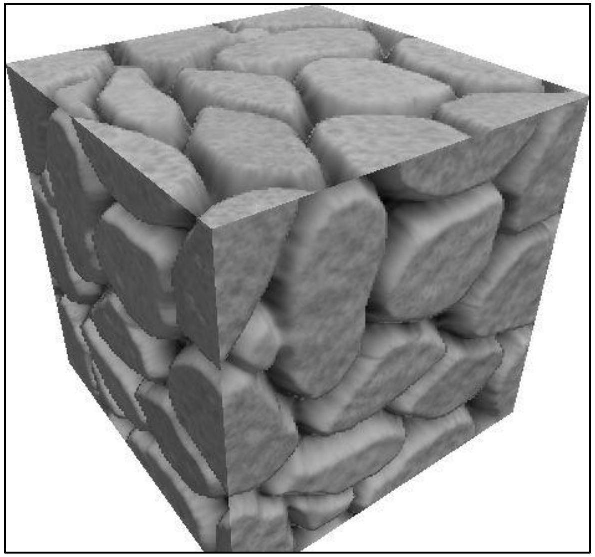
Silhouettes don't match with the relief pattern

Parallax Occlusion Mapping (POM) (1)

- The key idea behind POM is to consider the surface as a "shell" of a more complex geometry and trace a line from the visible shell point inwards until we hit the relief height field
  - The new location will be used for any shading calculations

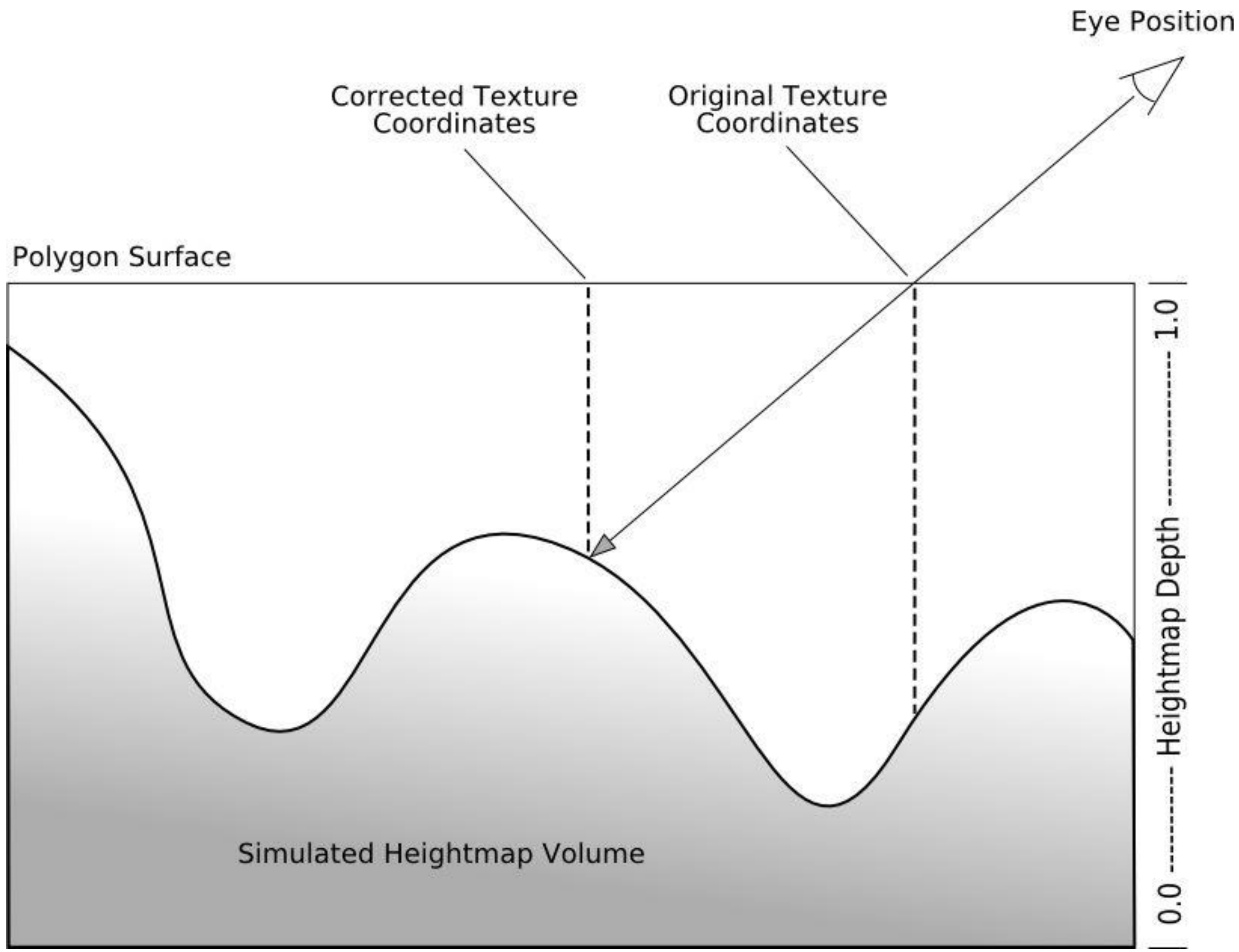


Normal mapping



POM

Parallax Occlusion Mapping (POM) (2)



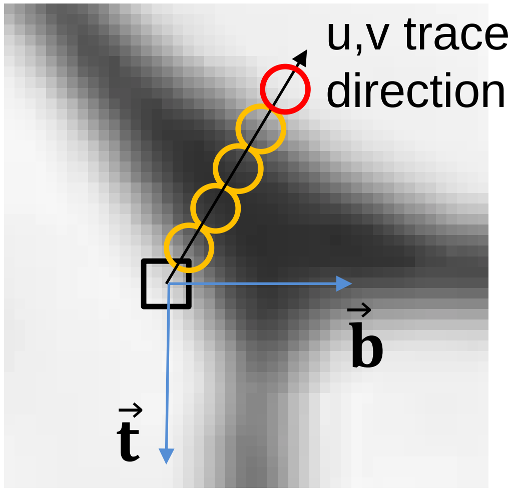
Images from [GD13]

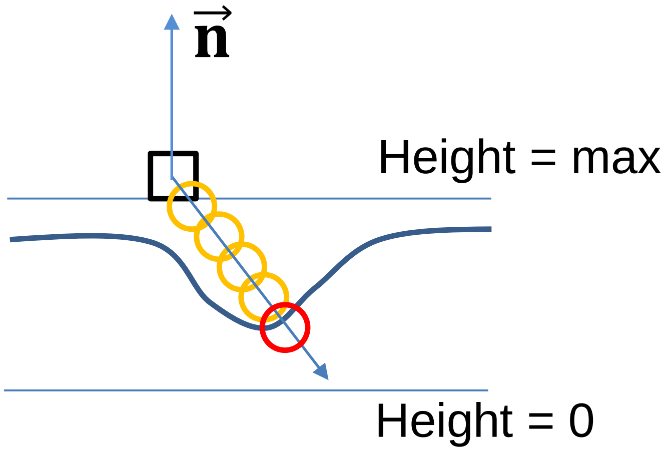
POM Steps

- Convert eye and light positions to tangent space
- Calculate incident light direction and viewing direction to tangent space
- Trace a ray inwards (assuming a start position at max elevation) – see next slide – until closest elevation point to entry is reached
- Use the current (u,v) position of the hit point to shade the surface (including tangent space normal mapping)

POM - Tracing in Tangent Space

- We sample the elevation map at regular intervals along the tangent-space view direction
  - Until ray sample “depth” below map elevation
  - Shade according to the attributes at the hit point u,v coordinates
- Must ensure dense (texel-sized) sampling
  - Fixed number of samples cannot guarantee this for oblique view directions



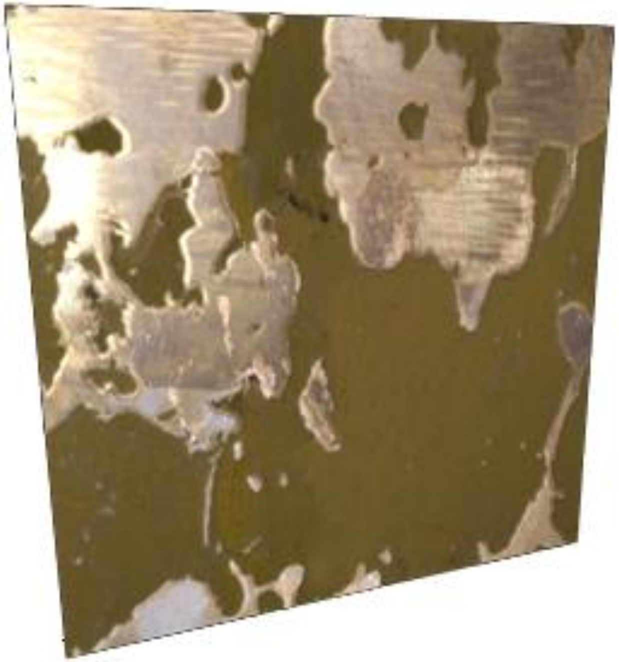


POM Results

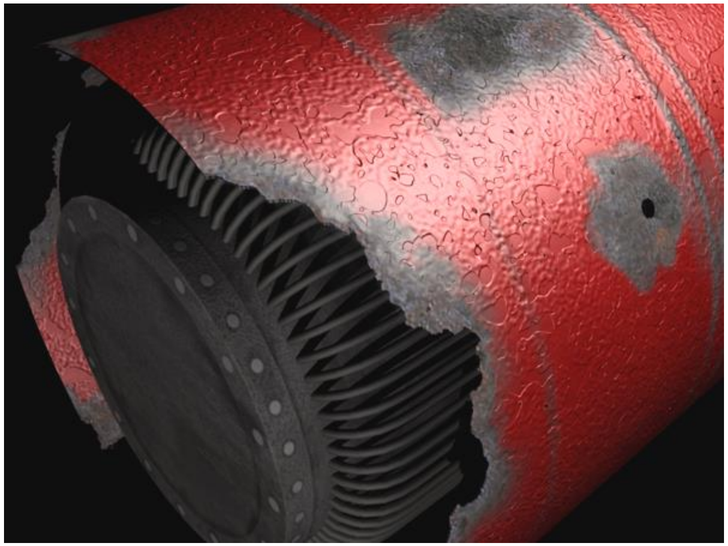
PUTTING IT ALL TOGETHER

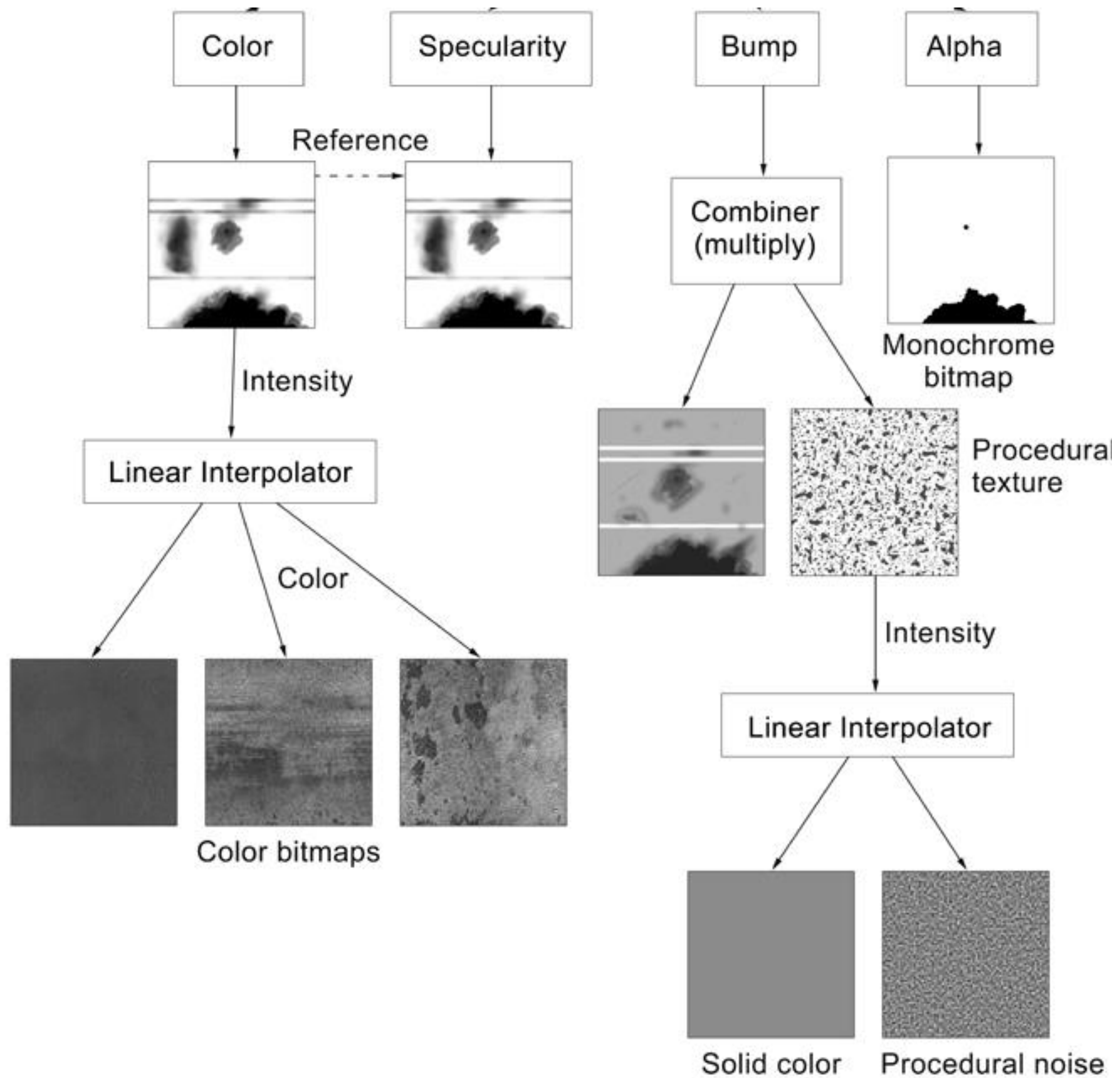
Using Multiple Textures

- To achieve the desired effect and re-use textures, we often combine multiple layers of texture during shading
- We additionally use textures as:
  - Masks: provide a blend factor between layers
  - Decals: local overlays to represent stickers, dirt, marks etc.
  - Indices: specify which texture to use at each location



Texture Graphs: An Example

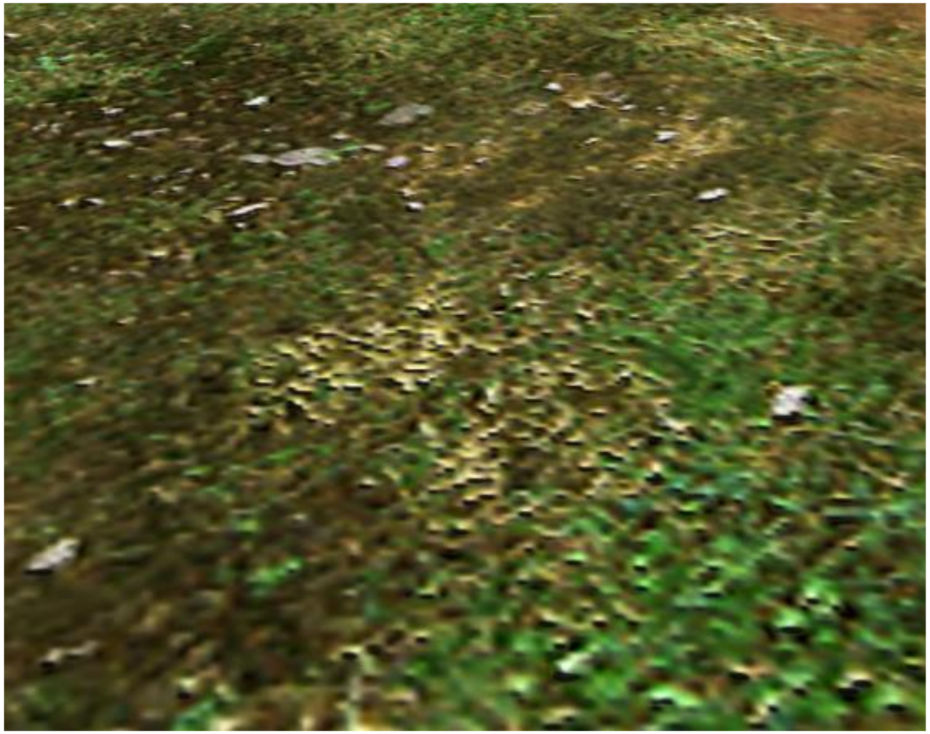




We can combine image and procedural textures to achieve the desired effect

Detail Textures (1)

- For large surfaces, in order to create enough visual variation, we often use large, non-repeated textures
- However, they cannot withstand close inspection due to the limited texture density

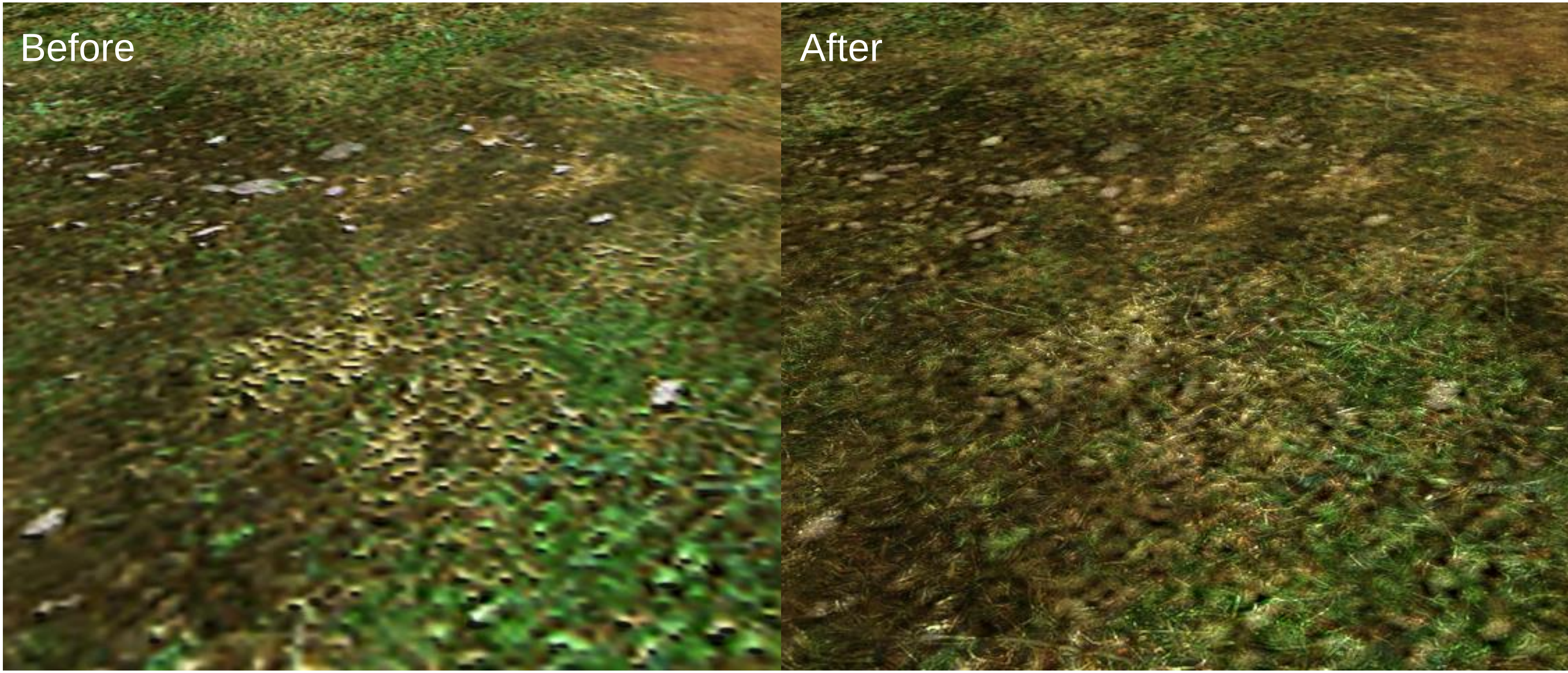


Detail Textures (2)

- To diversify the result, we superimpose (usually multiplicatively) a small repeatable texture many times over the main texture to add detail
- We typically multiply the texture coordinates used for the "detail texture" by a large factor to repeat it that many times

Detail Textures (3)

- Now the textured surface looks detailed even at high magnification



Texture and Material Layering Example

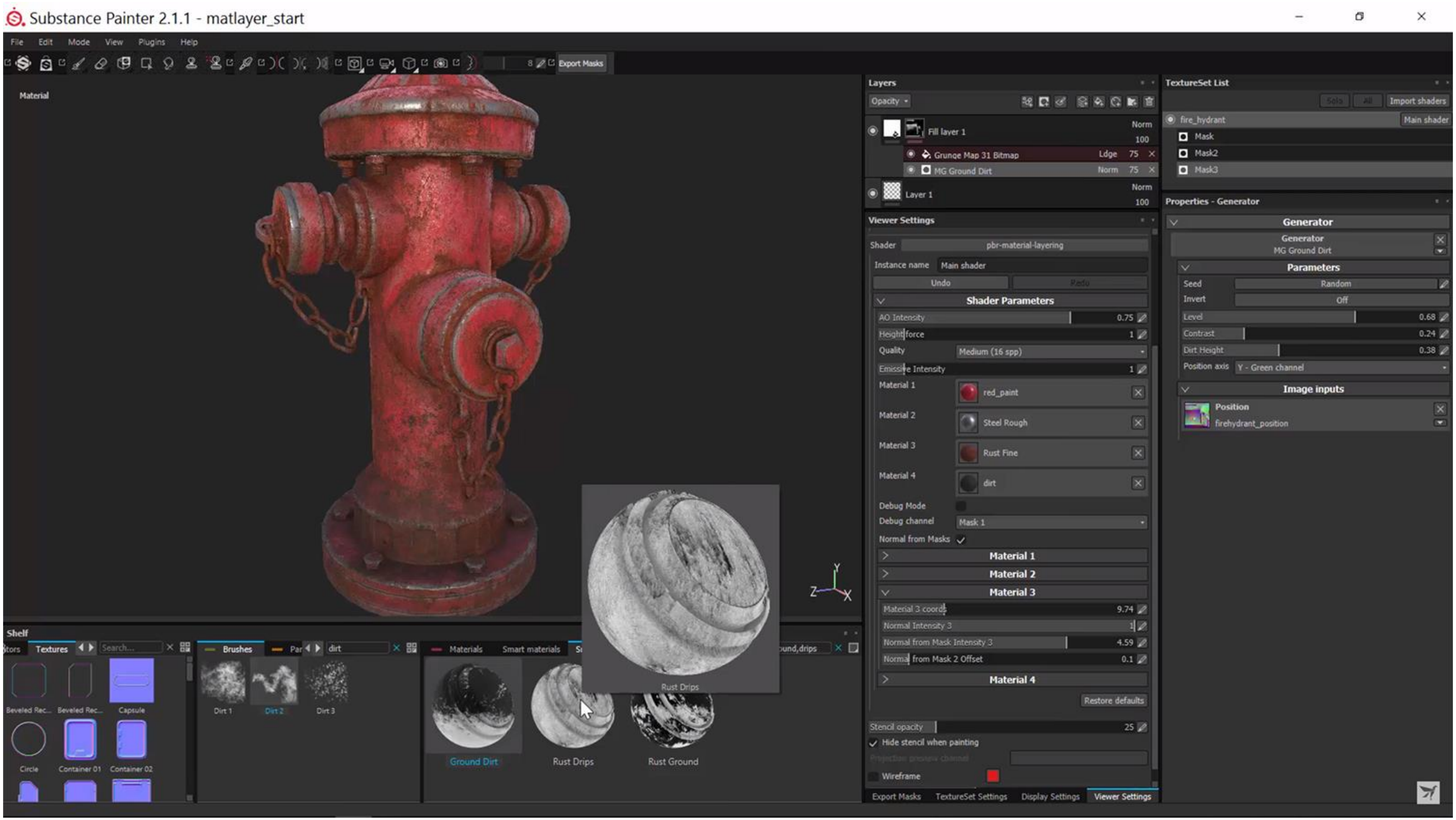


Image: Allegorithmic Substance Painter 2

Texture and Material Layering Example

Image: Allegorithmic Substance Painter 2

References

- [GD13] https://www.gamedev.net/articles/programming/graphics/a-closer-look-at-parallax-occlusion-mapping-r3262

Contributors

- Georgios Papaioannou