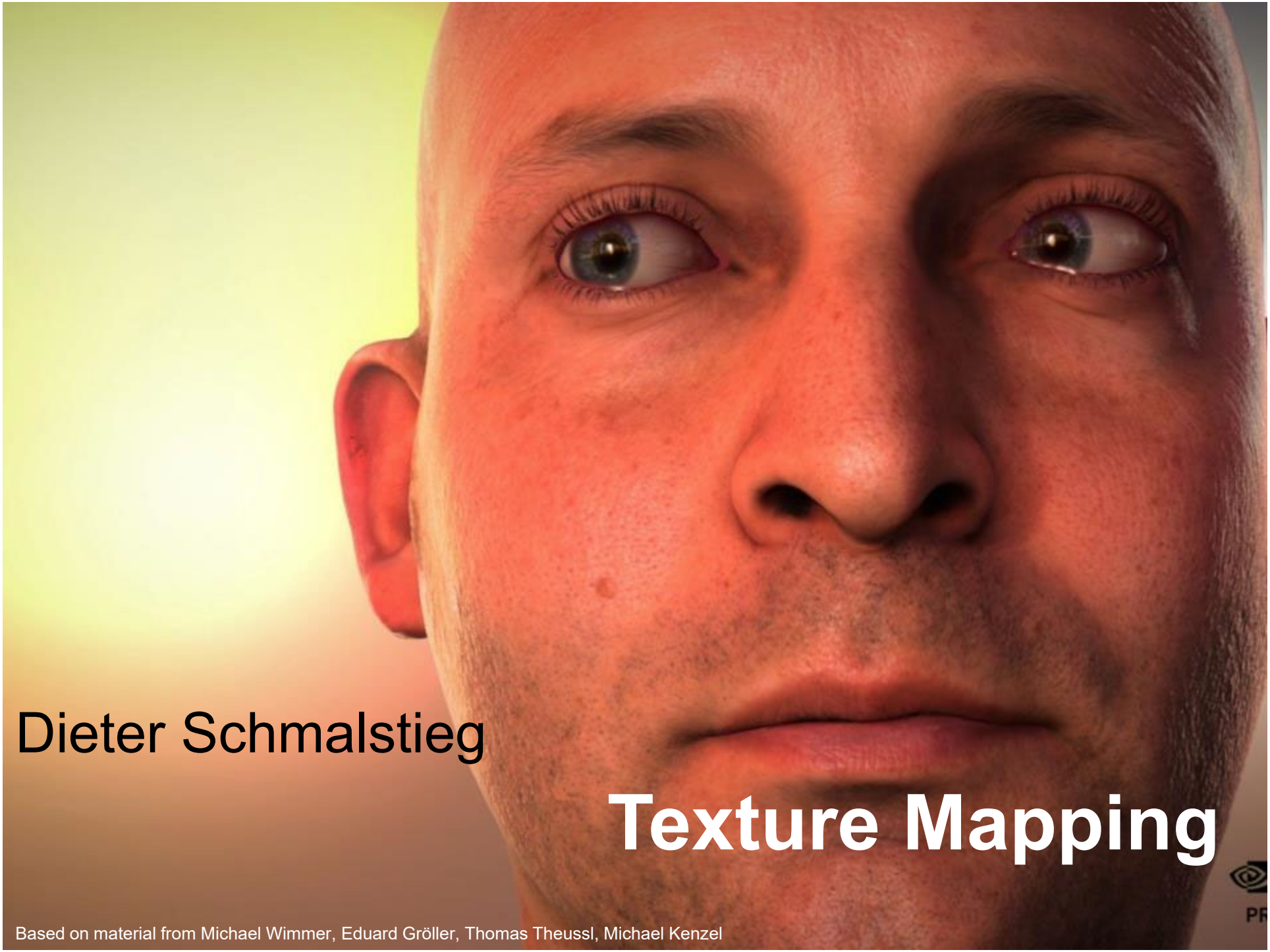


Dieter Schmalstieg

Texture Mapping

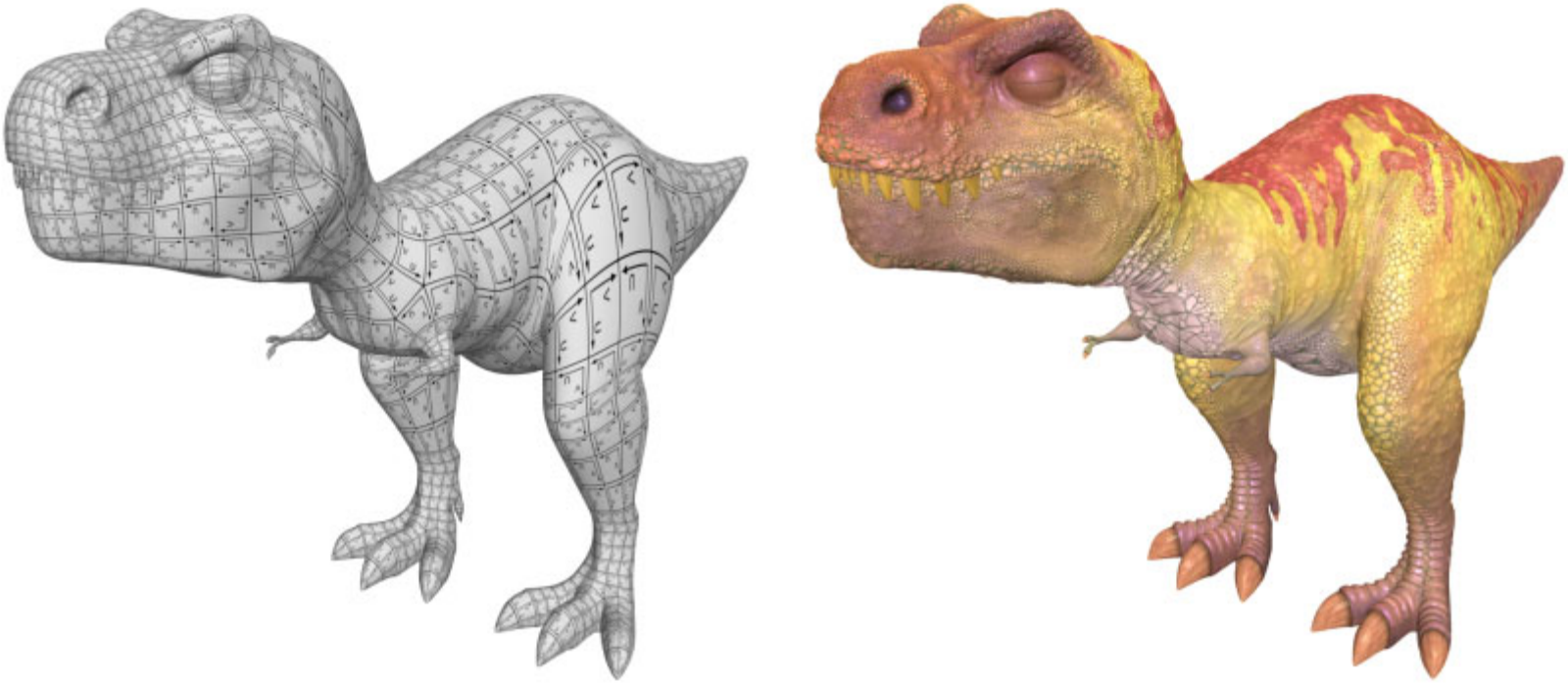
Based on material from Michael Wimmer, Eduard Gröller, Thomas Theussl, Michael Kenzel

Overview

- Basic texture mapping
  - Texture coordinate generation
  - Filtering
- Advanced texture mapping
  - Multitexture/multipass
  - Projective texturing
  - Environment mapping
  - Bump mapping
  - Per-pixel lighting

Why Texturing?

- Enhance visual appearance of plain surfaces and objects by applying fine structured details



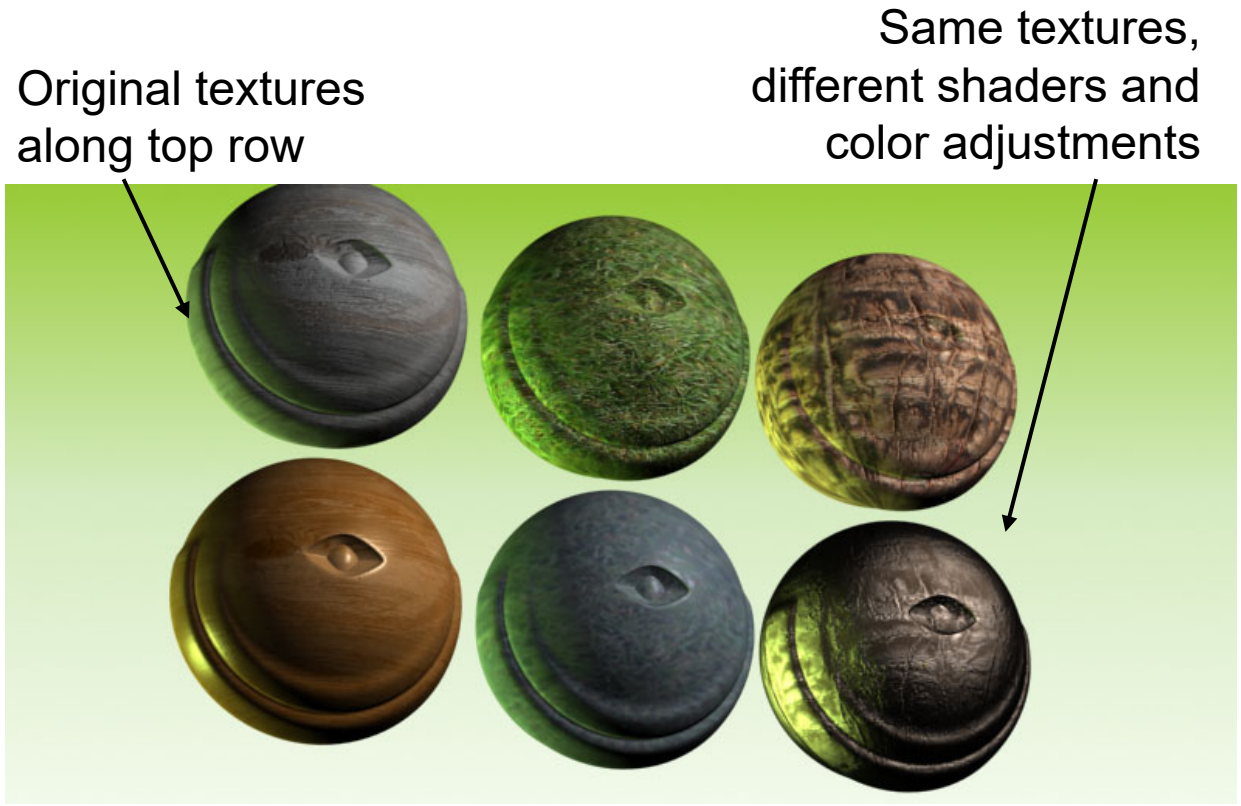
What is a Texture?

- Provides look and feel of a surface
- Definition
  - A *function* that is evaluated for every *fragment* of a surface
  - Usually given as a 2D image
- Can hold arbitrary information
  - Texture data interpreted in fragment shader
  - Basis for most advanced rendering effects

Textures as Patterns

Material properties simulation:

- Color
- Reflection
- Gloss
- Transparency
- Bumps

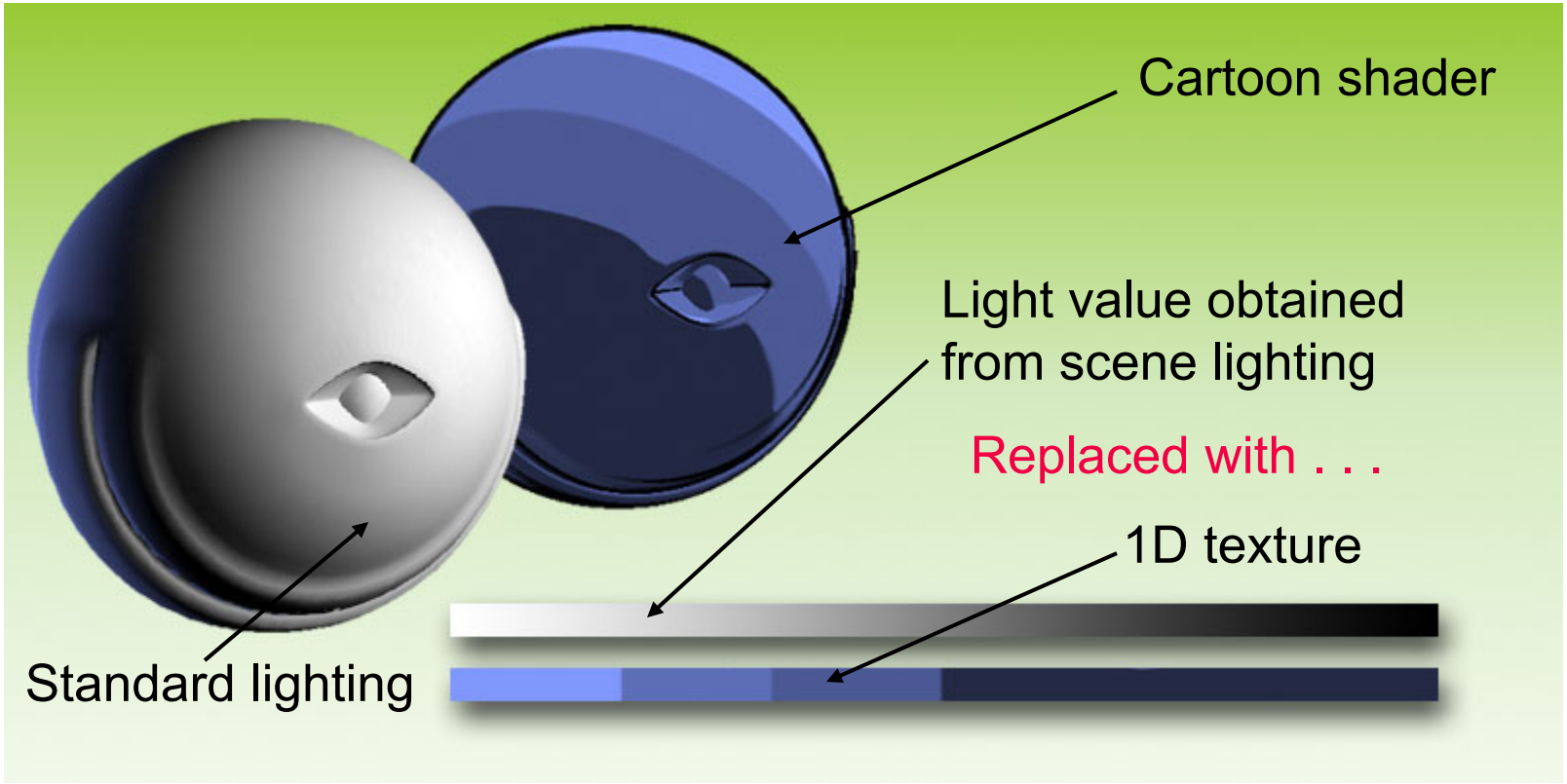


Textures are just color and value combinations, the same texture can be used in many different ways

Texture Types

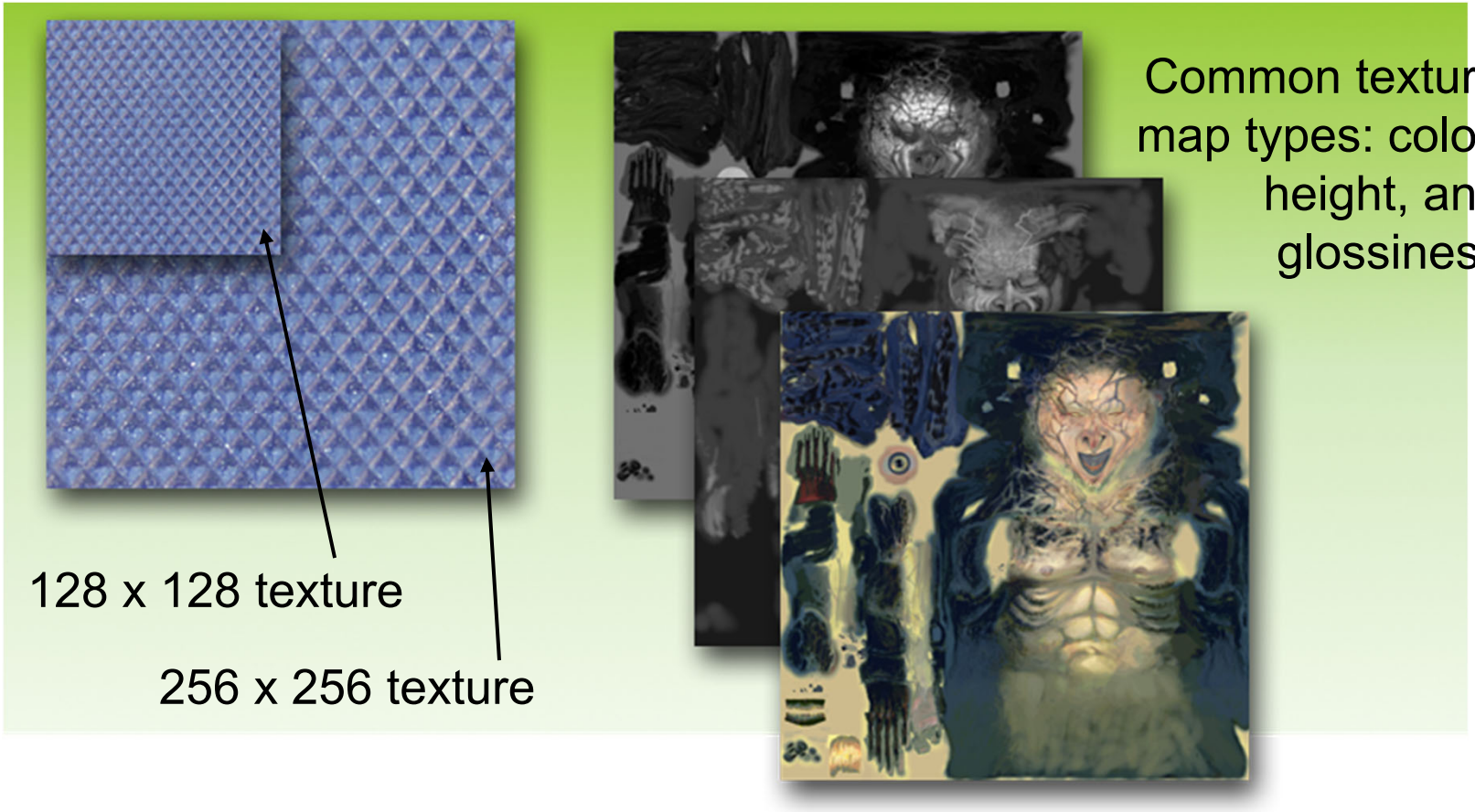
- One-dimensional functions
  - Parameter can have arbitrary domain (e.g., incident angle)
- Two-dimensional functions
  - Information is calculated for every (u,v)
  - Many possibilities
- Raster images (“texels”)
  - Most often used method
  - Images from scanner or calculation
- Three-dimensional functions
  - Volume T(u,v,w)

1D Textures



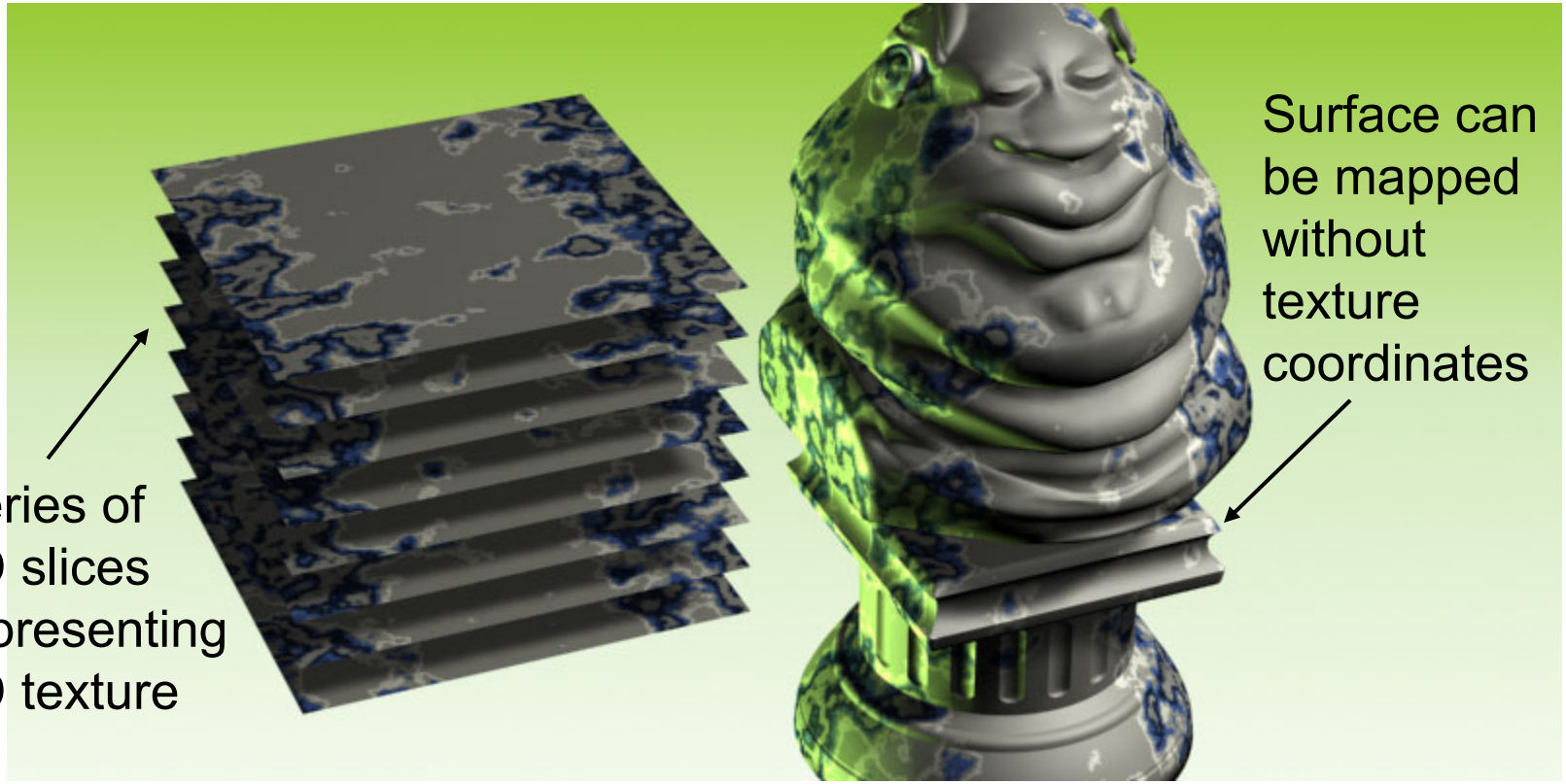
1D texture can be substituted for any linear value used in shader

2D Textures



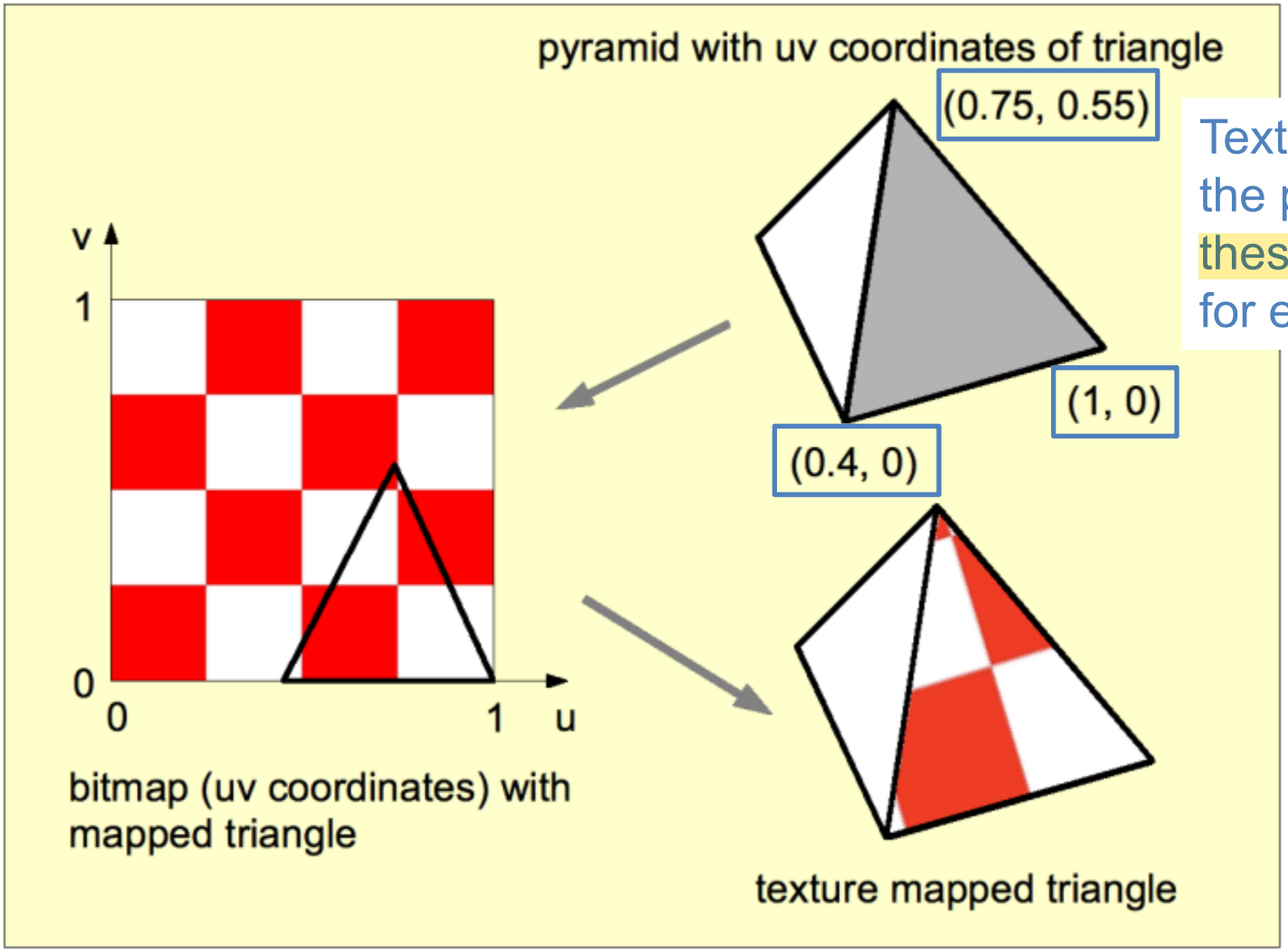
2D directly relate to surfaces

3D Textures



3D textures can represent procedural textures

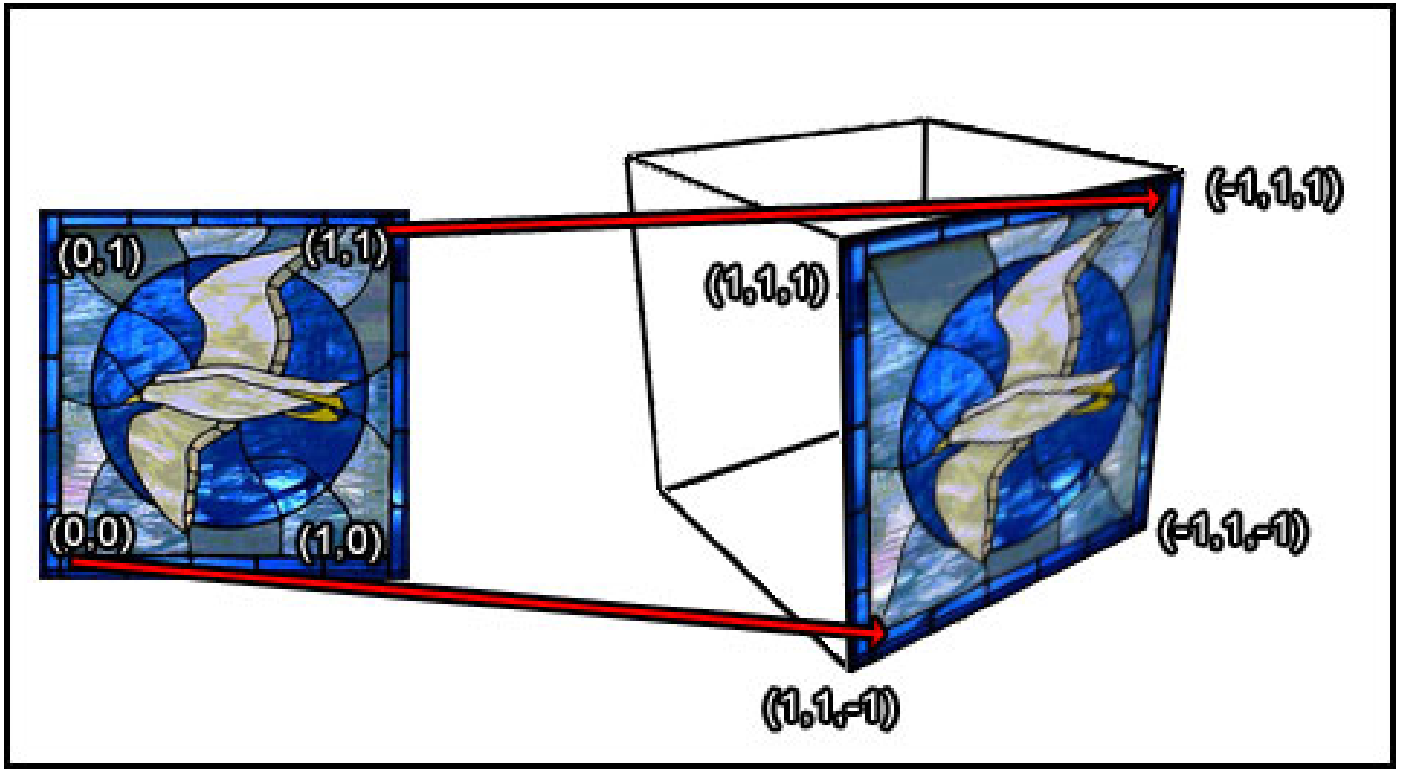
Texture Coordinates



Texture mapping is the process of finding these (u,v) coordinates for each vertex!

Parametrization

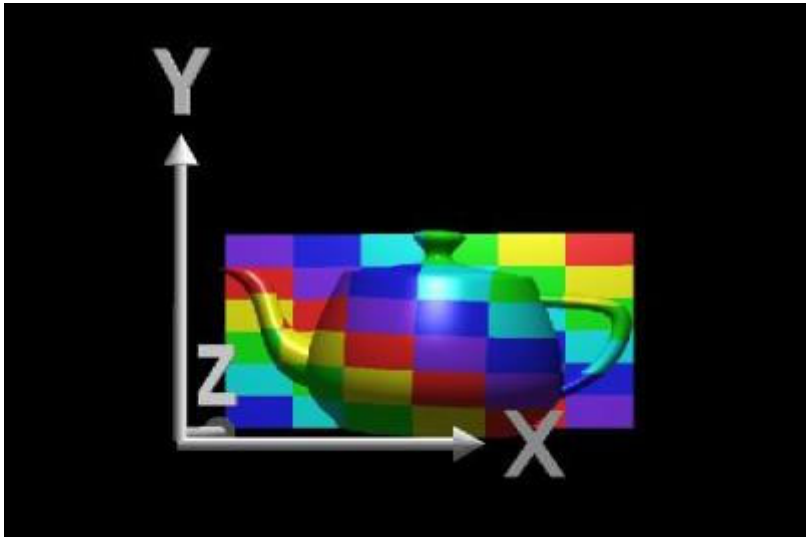
- How to do the mapping?

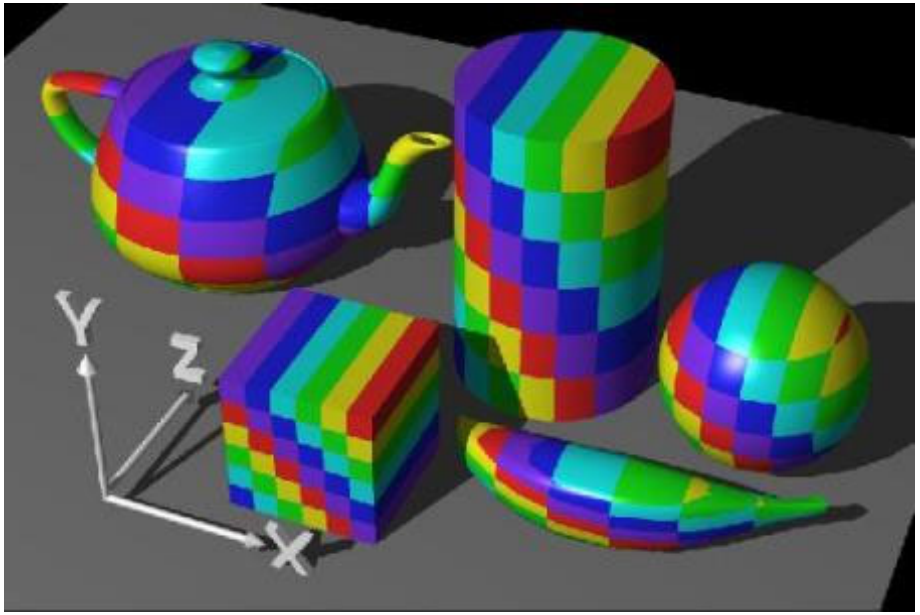


- Usually objects are not that simple

Parametrization: Planar

- Planar mapping: dump one of the coordinates

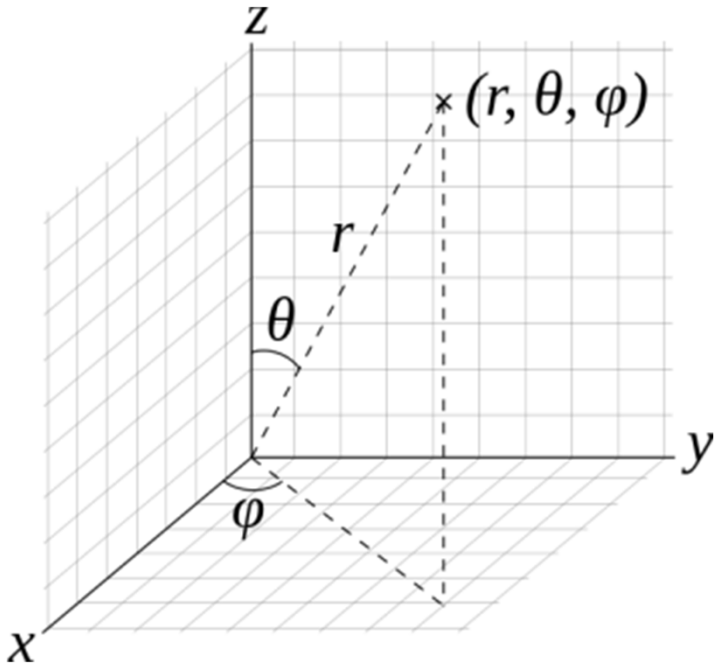




Only looks good from the front!

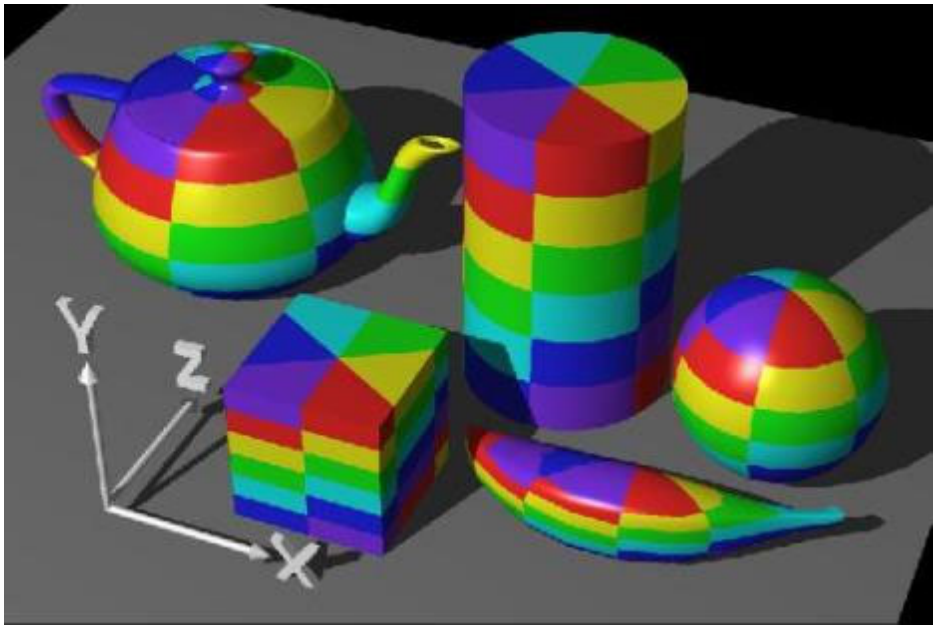
Parametrization: Cylindrical

- Cylindrical and spherical mapping: compute angles between vertex and object center
- Compare to polar/spherical coordinate systems

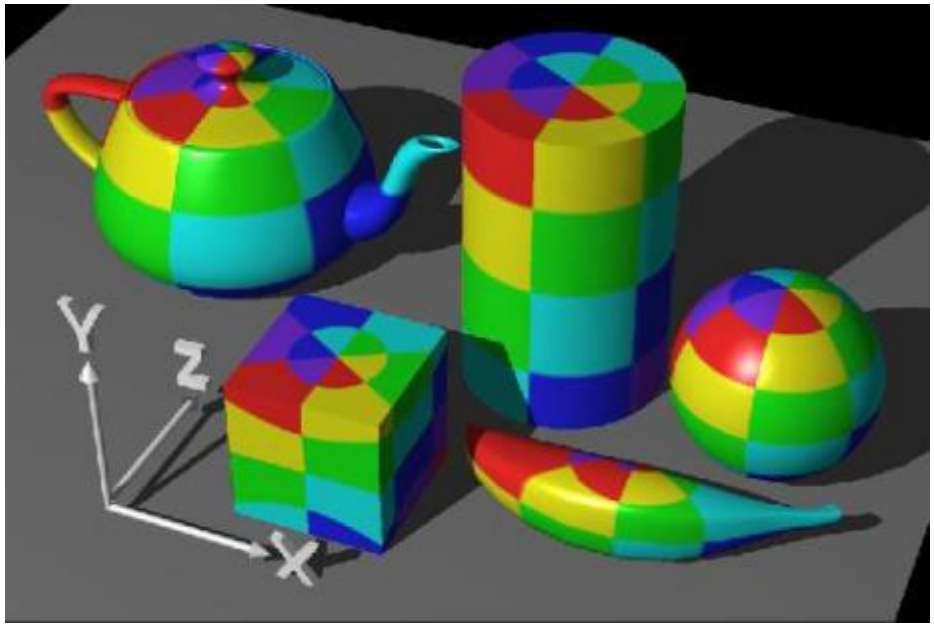


Parametrization: Polar

- Cylindrical and spherical mapping



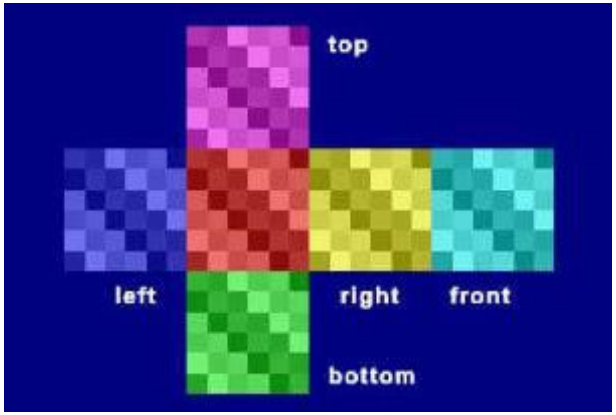
Cylindrical

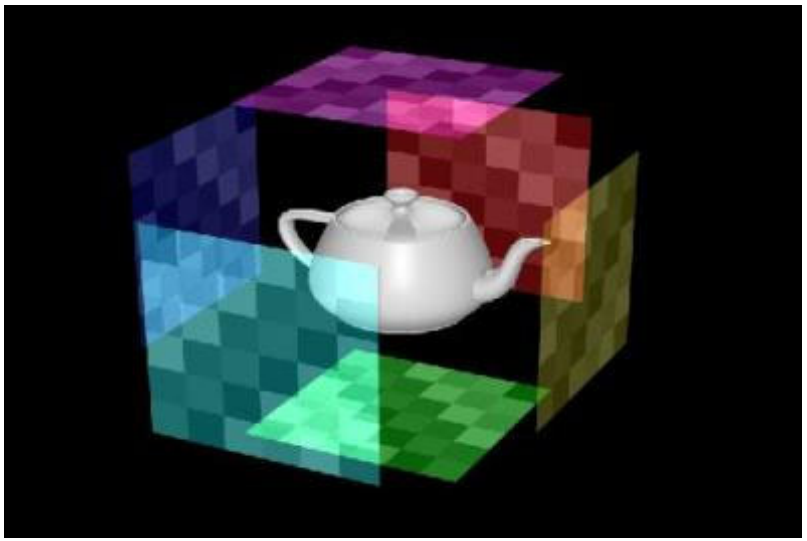


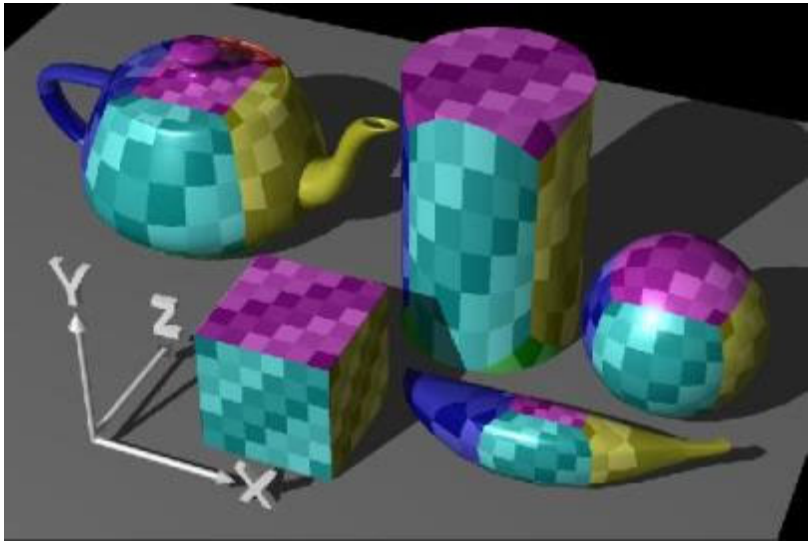
Spherical

Parametrization: Box

- Box mapping: used mainly for environment mapping (see later)

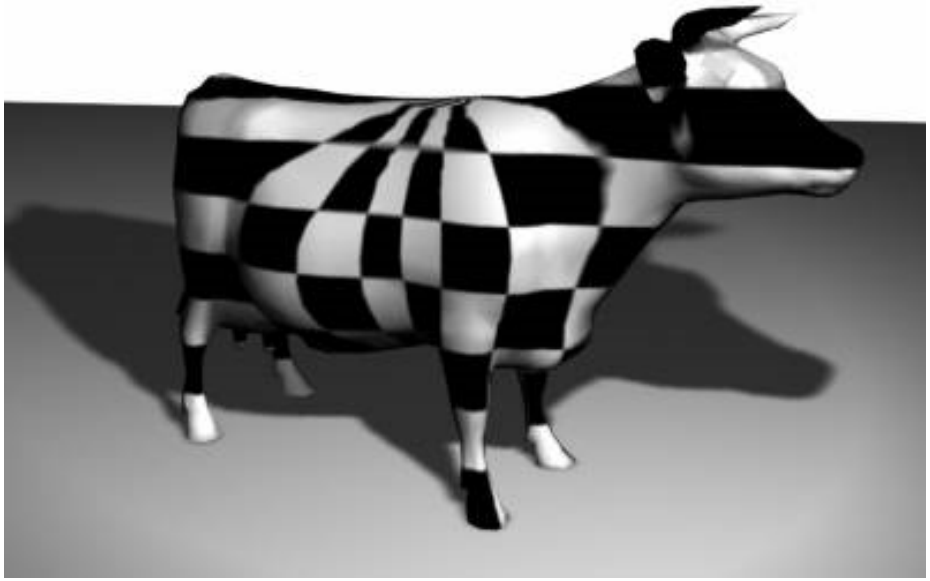






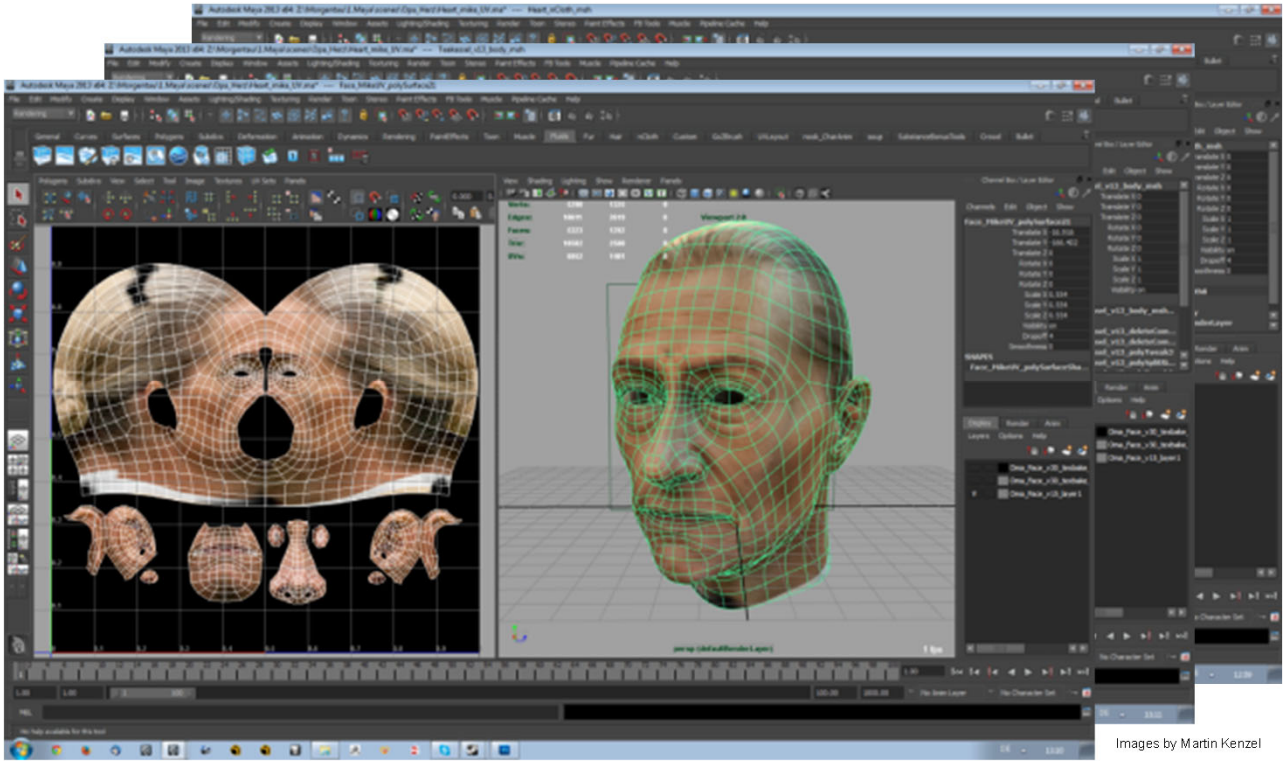
Parametrization Problems

- All mappings have distortions and singularities
- Often they need to be fixed manually (CAD software)



Manual Parametrization

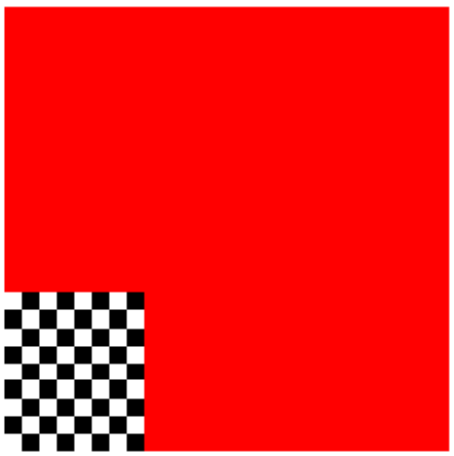
- Manual mapping using CAD Software
  - “Unwrapping”



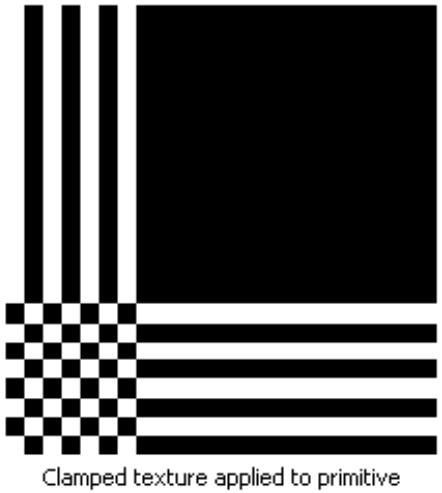
Images by Martin Kenzel

Texture Adressing

- What happens outside [0,1]?
- Border, repeat, clamp, mirror
- Also called texture addressing modes



Static color



Clamped

Repeated

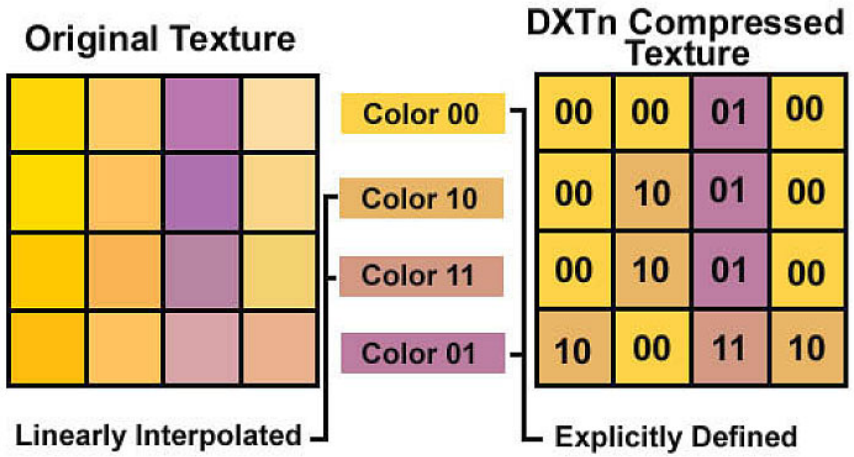
Mirrored

Texture Animation

- Texture Matrix
  - Can specify an arbitrary 4x4 Matrix, each frame!
  - `glMatrixMode(GL_TEXTURE);`
  - There is also a texture matrix stack!
  - Possibilities: moving water or sky
- Custom shader programs
  - Can use video as texture
  - Allows arbitrary texture animations

Texture Compression

- S3TC texture compression
  - Represent 4x4 texel block by two 16bit colors (RGB 5/6/5bit)
  - Store 2 bits per texel
- Uncompress
  - Create 2 additional colors between c1 and c2
  - Use 2 bits to index into color map
- Compression ratio 4:1 or 6:1



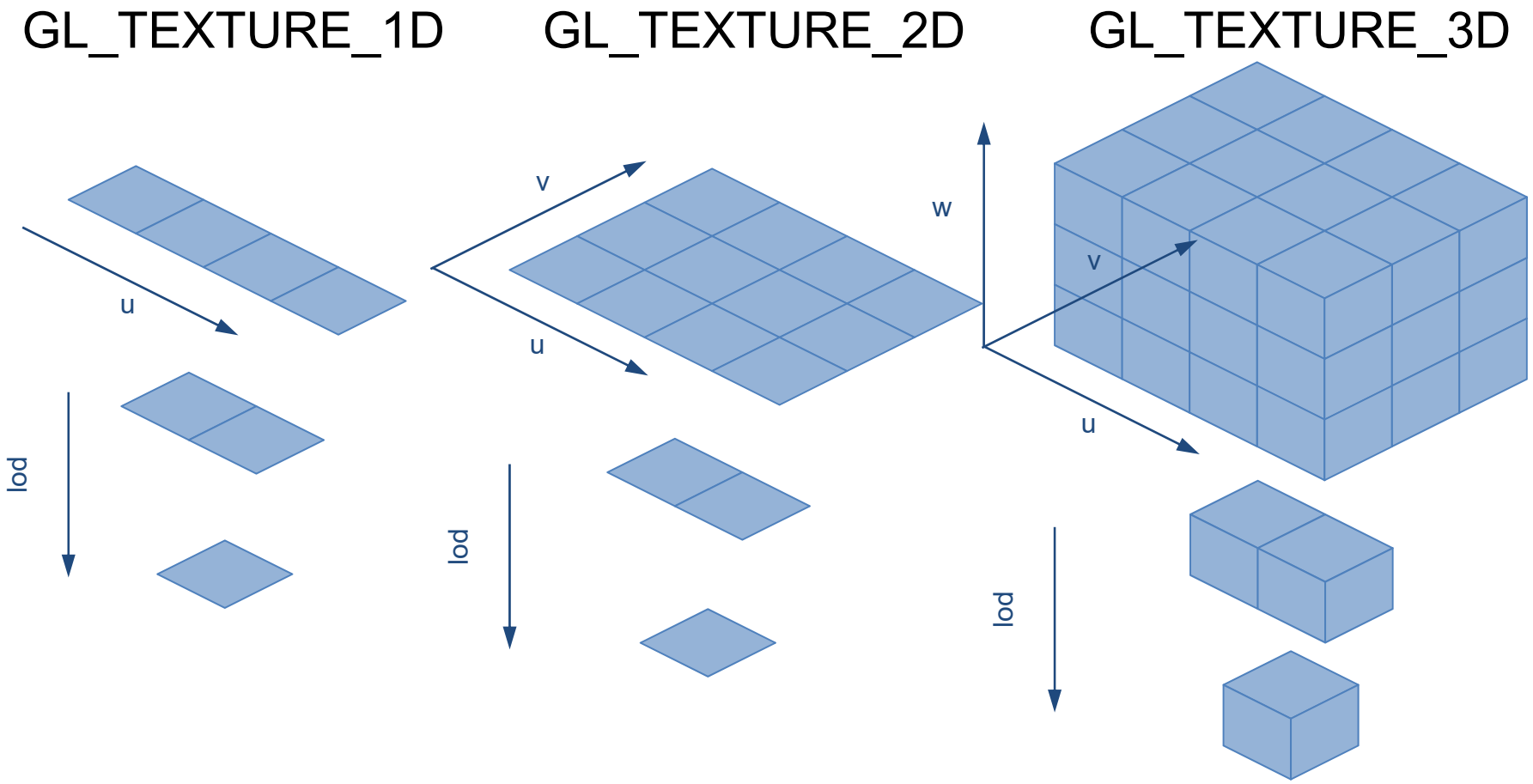
using color values based on the graphics chips color palette. It reduces the size of texture by reducing the number of colors contained within the texture without comprimising quality

Texture Representation on GPU

- Texture Image = array of *texels*
  - Certain dimensionality
  - Certain format (e. g., GL_RGBA8)
- Texture Objects
  - Group of one or more texture images
  - Images can represent mip levels, array slices…
- Sampler Objects
  - Configure texture sampling (interpolation mode…)

Ein Sampler Object ist ein Objekt, das die Einstellungen für die Art und Weise speichert, wie eine Textur auf ein 3D-Modell angewendet wird. Dazu gehören Einstellungen wie die Filterung der Textur (z.B. skalieren oder glätten).

Texture Dimensions



Texture Format and Specification

Format

- Numerical format
  - Integer, float, double
- Bits per number
  - 8, 16, 32, 64
- Channels
  - Red, green, blue, alpha, depth, stencil

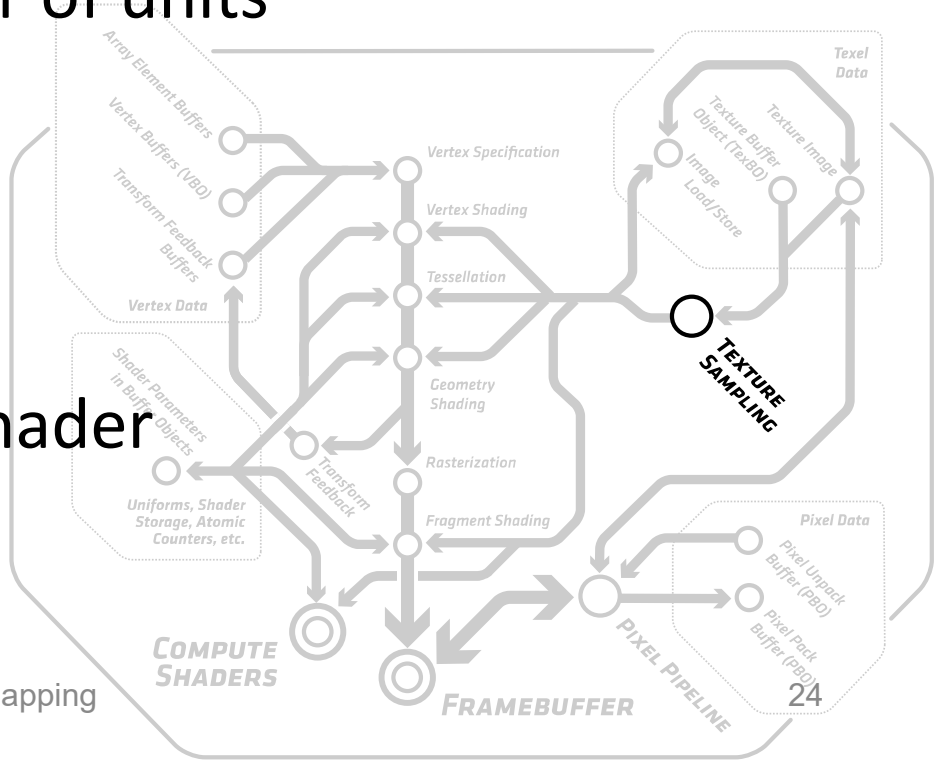
Specification

- Texure id
- Dimension
- External format
- Internal format
- Width, height
- Pointer to raw data

Sampler Objects

- Samplers configure texture units
  - Filtering, address mode, …
- How texture units work
  - There is a limited number of units
  - Activate the unit
  - Bind texture to a unit
  - Bind sampler to a unit
  - Sample texture data in shader

Zusammen arbeiten Sampler Objects und Texture Units, um die Texturen auf 3D-Modelle anzuwenden und dafür zu sorgen, dass die dargestellten Modelle realistisch und ansprechend aussehen.

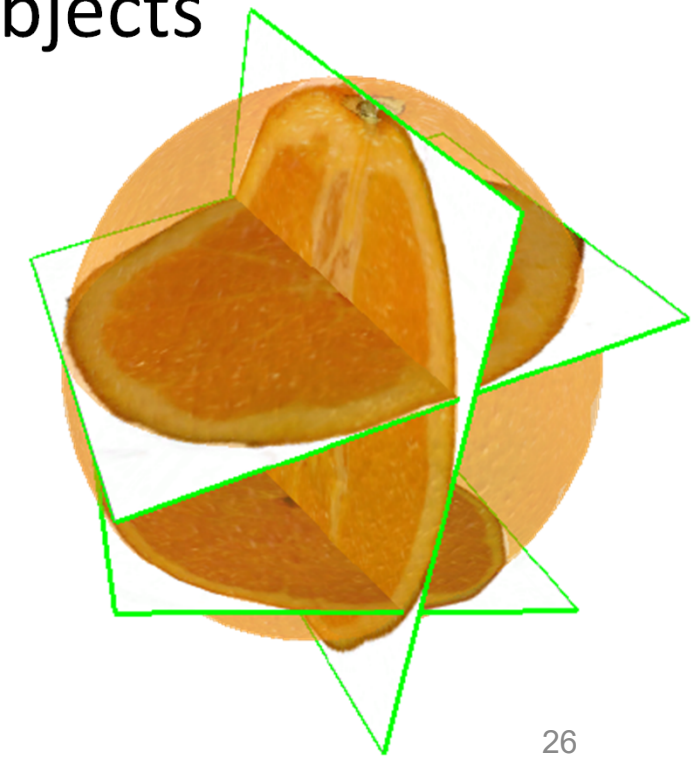


2D Texture Example: Fragment Shader

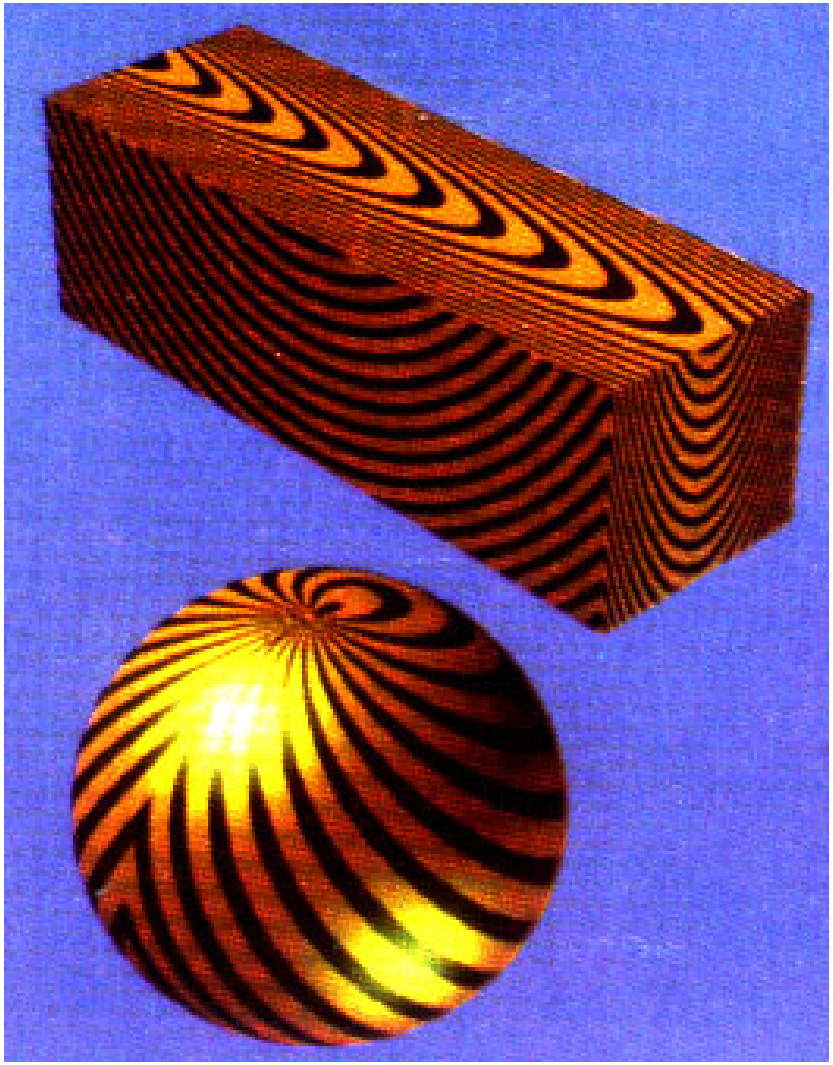
```
1  #version 330
2  uniform sampler2D my_texture;
3  in vec2 tex_coords;
4  layout(location = 0) out vec4 fragment_color;
5  void main()
6  {
7      fragment_color = texture(my_texture, tex_coords);
8  }
```

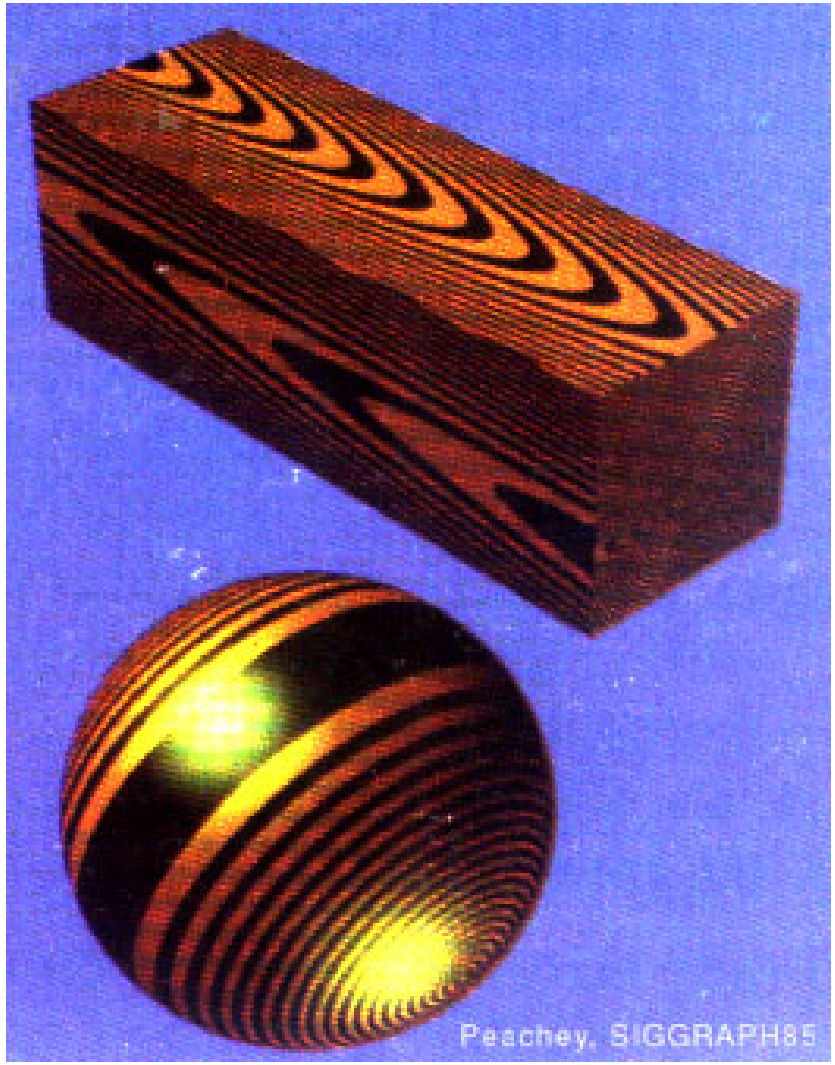
3D Texture Mapping

- Often referred to as *solid texturing*
- Directly map object space **T(x,y,z)** to **(u,v,w)**
- Possible to slice/carve/open objects



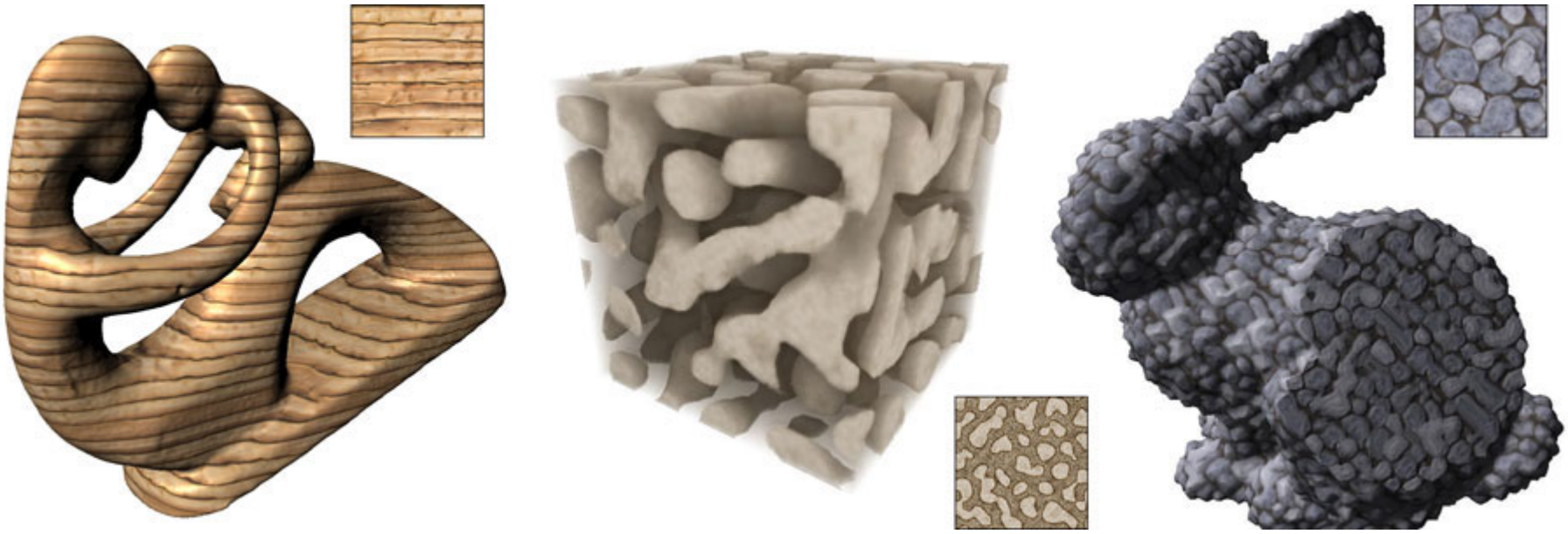
Mapping vs Solid Texturing



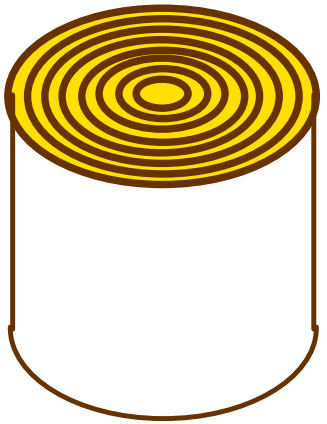


Solid Texturing Examples

- Carved objects: wood, marble, stone…



Procedural Solid Textures

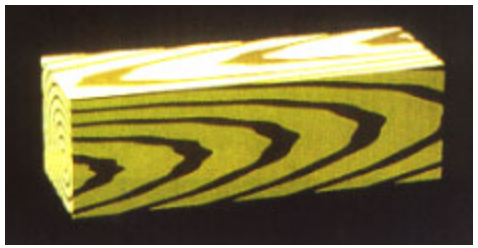


- **3D functions**

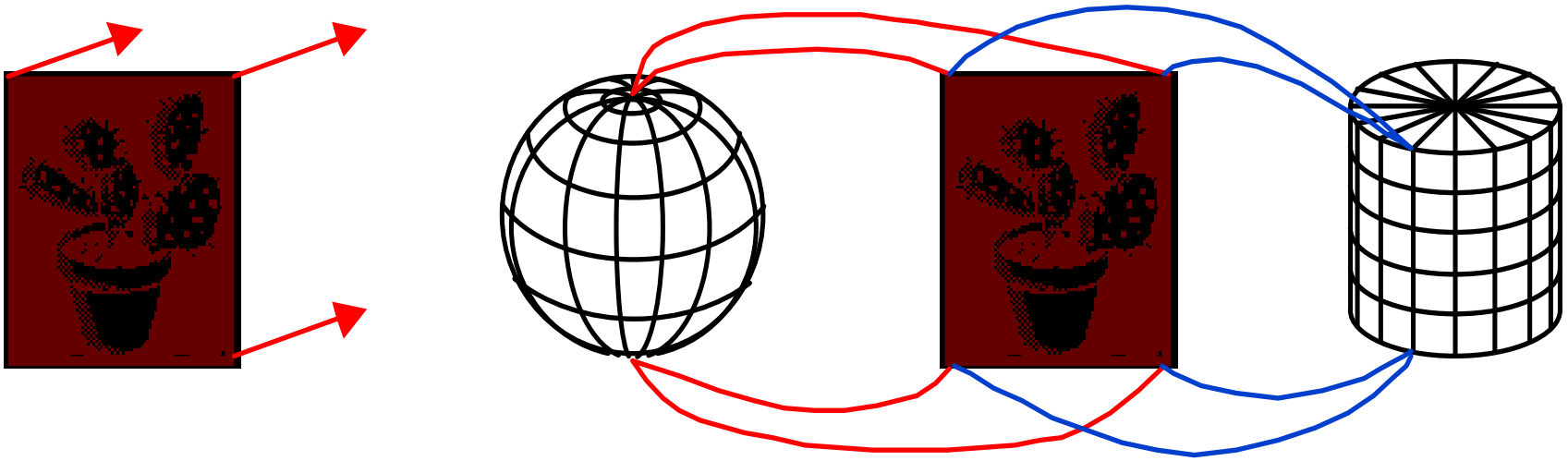
e.g., $0 \le x^2+y^2 < 1 \Rightarrow$ light brown

$1 \le x^2+y^2 < 2 \Rightarrow$ dark brown

$2 \le x^2+y^2 < 3 \Rightarrow$ yellow etc.



- **Projections of 2D data**



Volumetric Textures

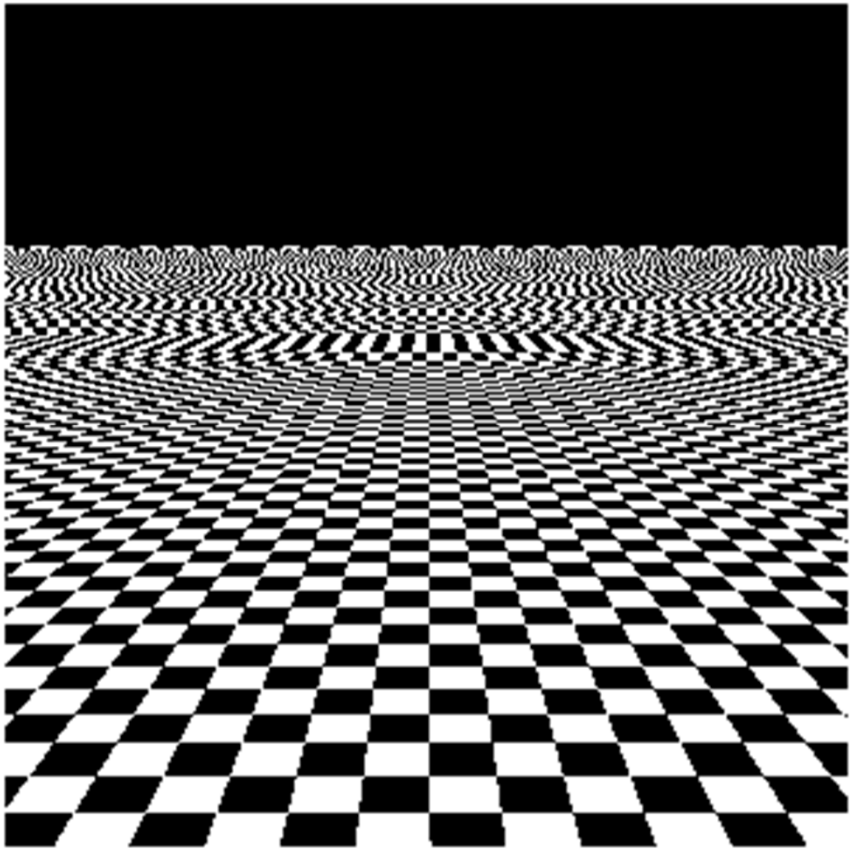
- 3D „voxel“ array of texture values
- Explicit, sampled function
- E.g., from medical scanners (CT, MRT, ...)
  + Works in real time
  - Lots of memory needed…

Textur ist 3D

Volumetric Fog

Aliasing

One screen pixel maps to many texels → Aliasing!



In der Entfernung -> Kantenbildung

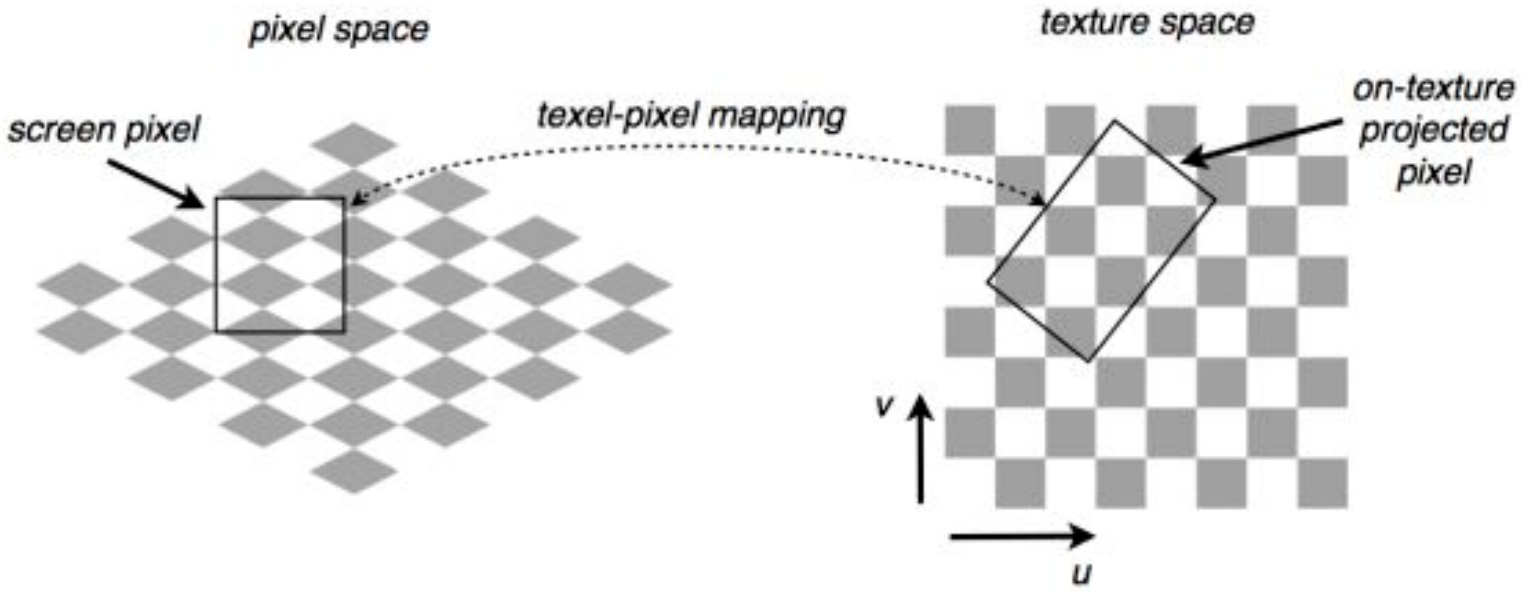
Hair

Without antialiasing

With antialiasing

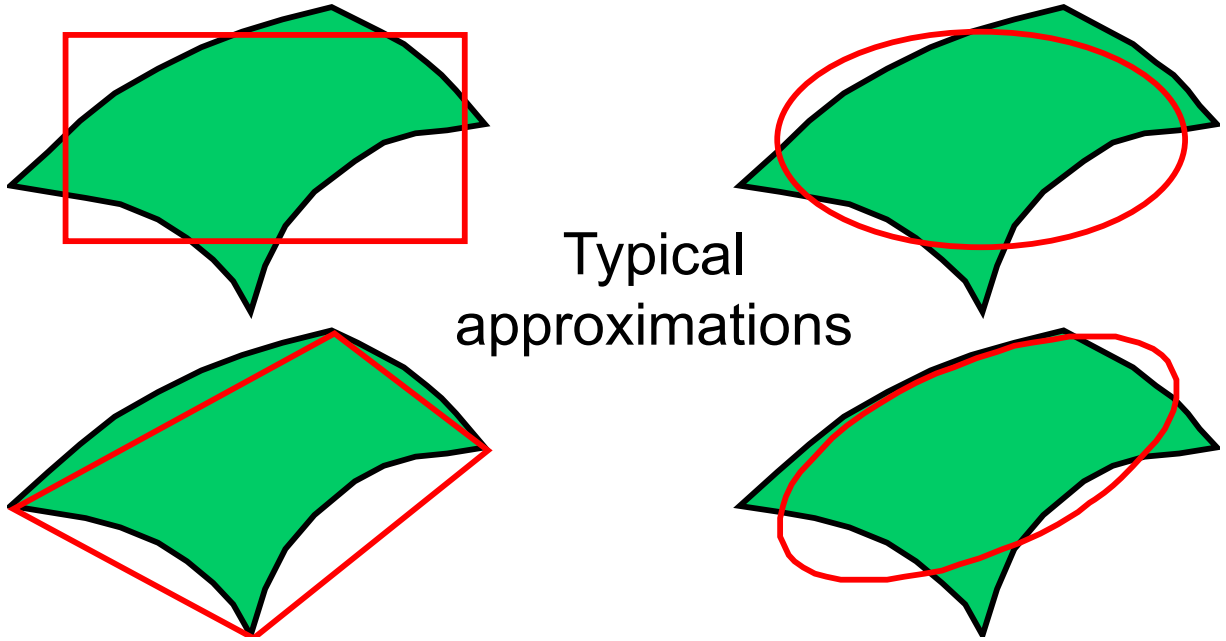
Antialiasing of Textures

- Correct pixel value is (weighted) mean of pixel area projected (back) into texture space
- Two approaches:
  - Direct convolution
  - Prefiltering



Direct Convolution

Calculate weighted mean of relevant texels



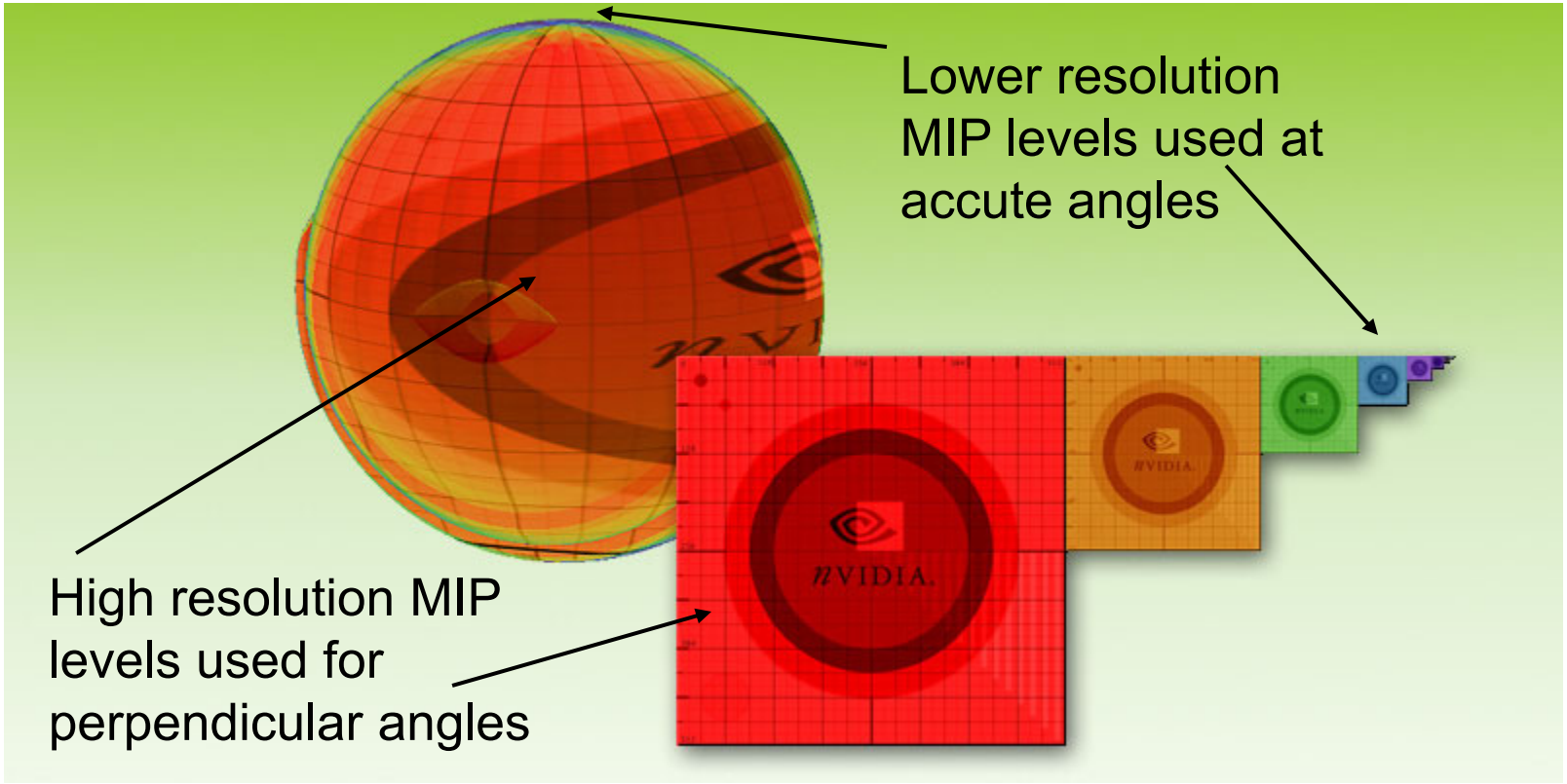
Real-time: approximate with fixed sample num.

Mip Mapping

downsampling $\Rightarrow$ much in a small area
= “multum in parvo” (mip)

- Texture is precalculated for different sizes
- Heavily used in real-time graphics
- Texture is reduced by factors of two
  - Simple
  - Memory efficient
- The last image is only one texel

Mip Mapping Example

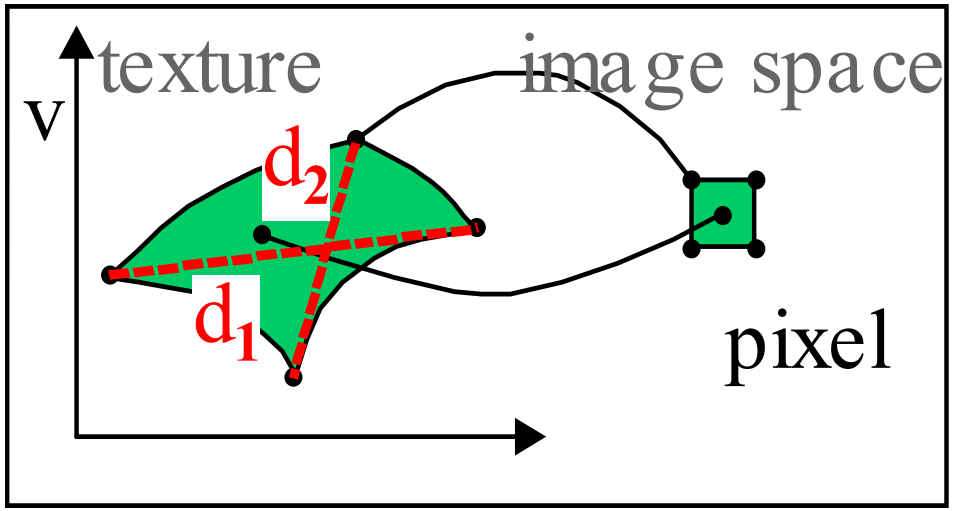


Mip Maps are reduce bandwidth requirements of distant textures and minimize scintillating pixels

Mip Mapping Algorithm

$D := ld(max(|d_1|, |d_2|))$

D ... Detailierungsgrad LoD
d … Distanz zwischen Eckpunkten auf der Textur



$T_0 := \text{value from table } D_0 = \text{trunc } (D)$
$T_1 := \text{value from table } D_1 = D_0+1$

T0 und T1 sind Indizes für die verschiedenen Mip-Map-Stufen und Bilinear Interpolation wird verwendet, um die Textur des Pixels auf der Grundlage dieser Mip-Map-Stufen und des Detaillierungsgrades zu berechnen.

- T_0, T_1 obtained with bilinear interpolation
- How to combine T_0, T_1?

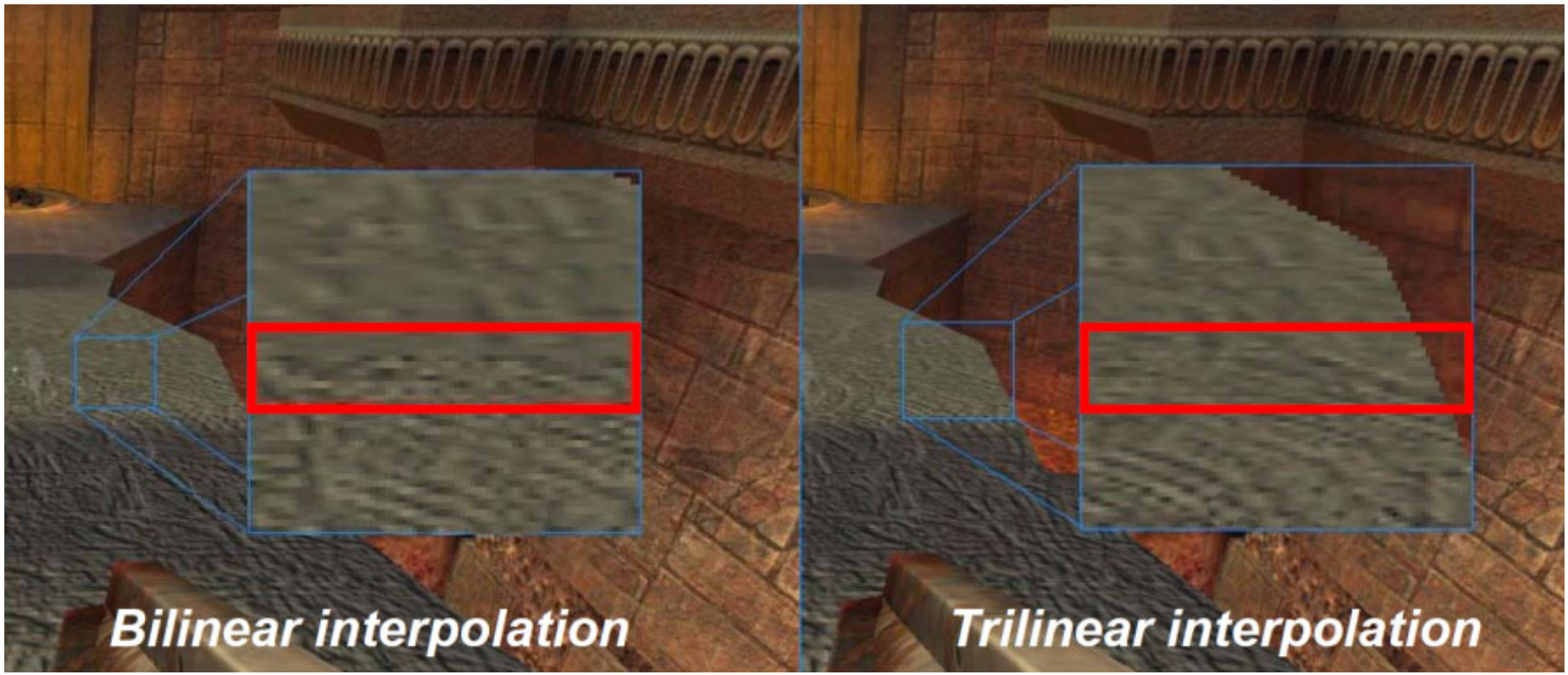
Bilinear vs Trilinear

- **Bilinear** := $(D_1\text{-}D>0.5)\ ?\ T_1 : T_0$
- **Trilinear** := $(D_1\text{-}D)\cdot T_0 + (D\text{-}D_0)\cdot T_1 =$

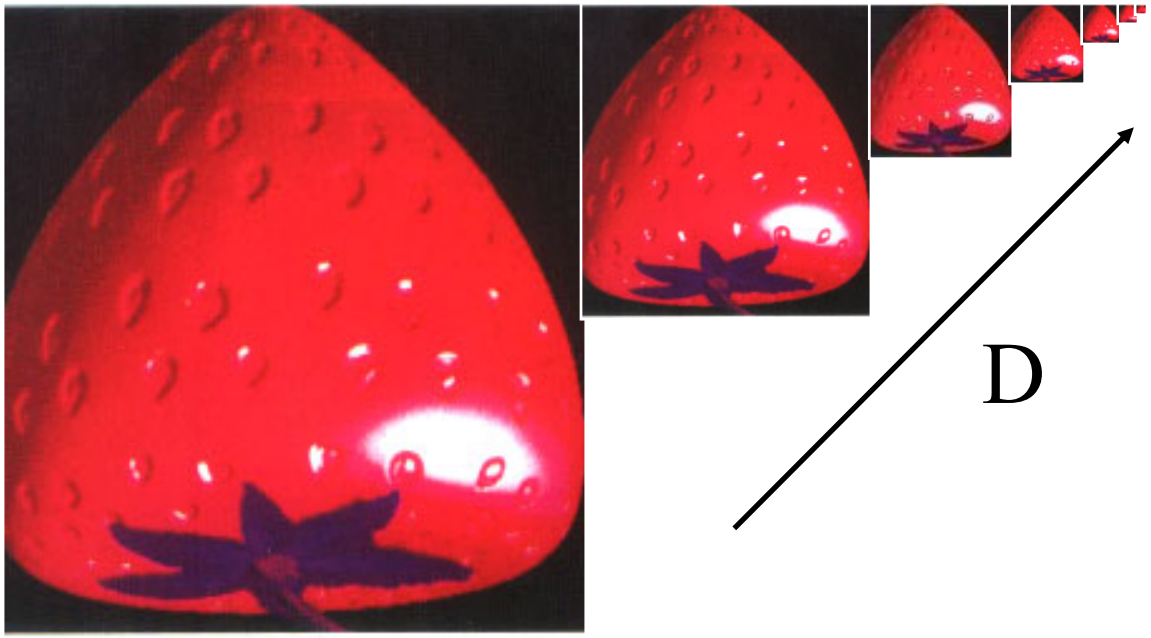
  $= (D_1\text{-}D)\cdot(T_0\text{-}T_1) + T_1$

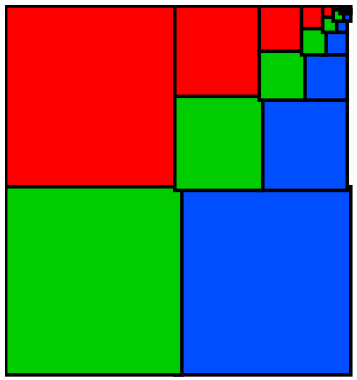
Bilinear:
4 benachbarte Pixel auf gleicher Ebene Interpoliert

Trilinear:
8 benachbarte Pixel auf 2 Ebenen interpoliert



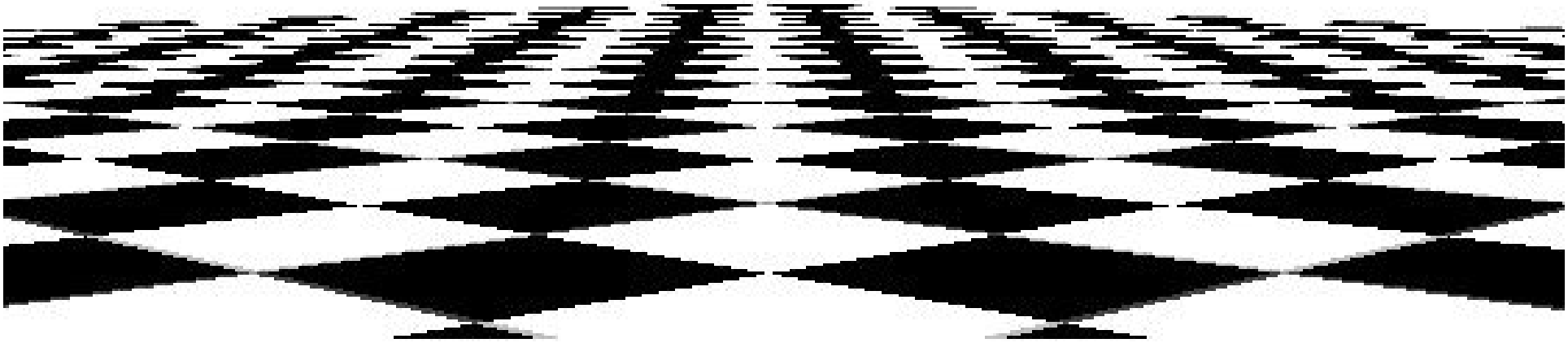
Mip Mapping Memory Layout





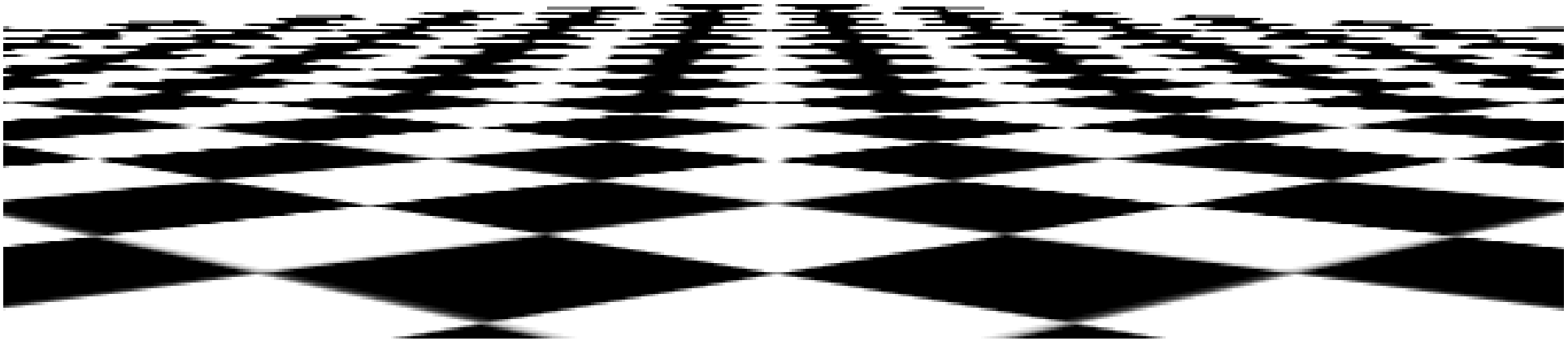
Memory overhead is just 33%
when channels are packed this way

Comparison: Nearest Neighbor



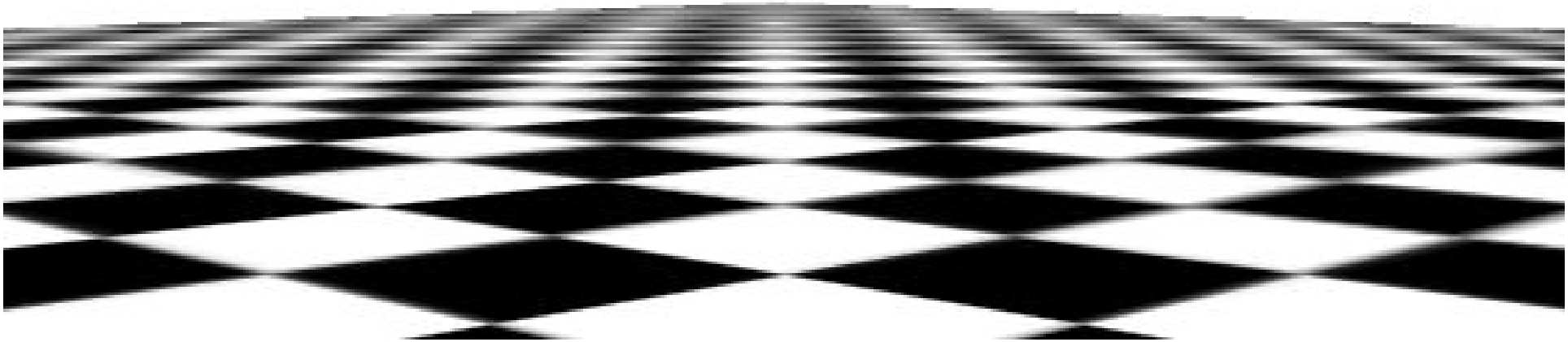
Simple, but bad quality

Comparison: Linear Interpolation



Fixes some aliasing in the front,
but in the back it‘s still not good

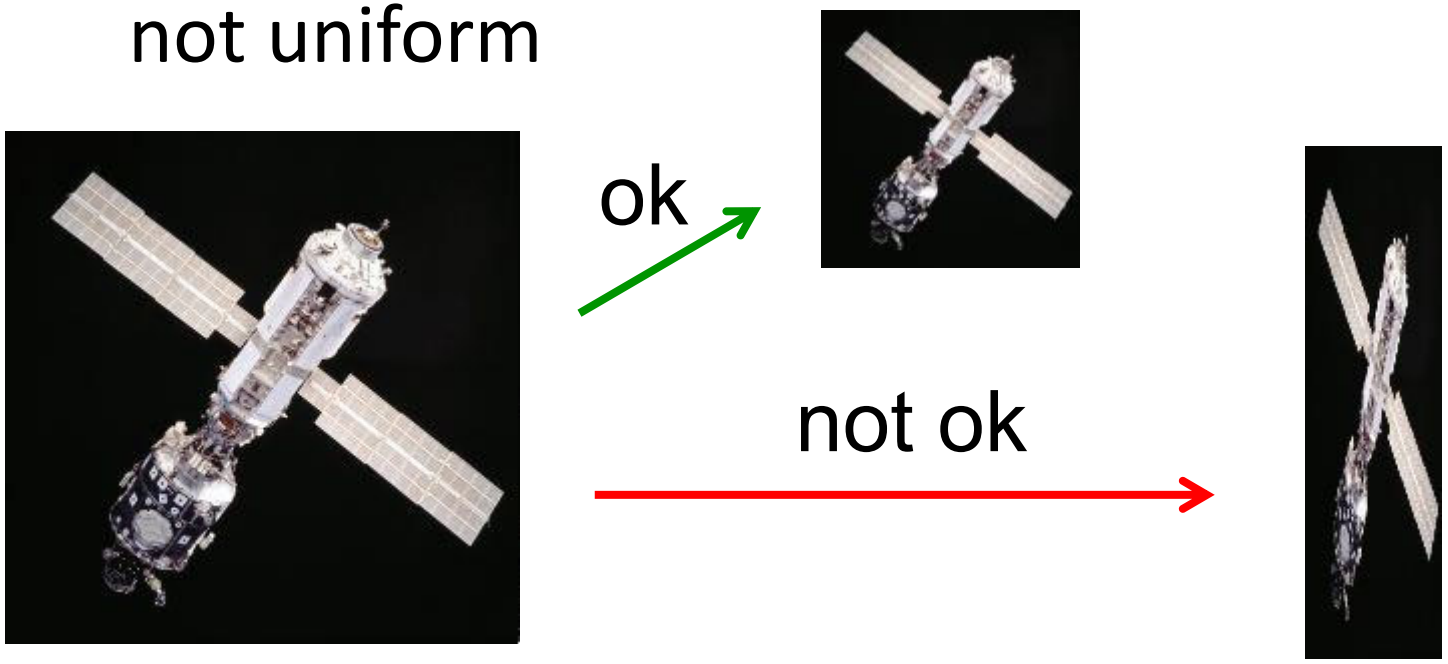
Comparison: Mip Mapping



Replaces objectionable artifacts with blur

What Causes this Problem?

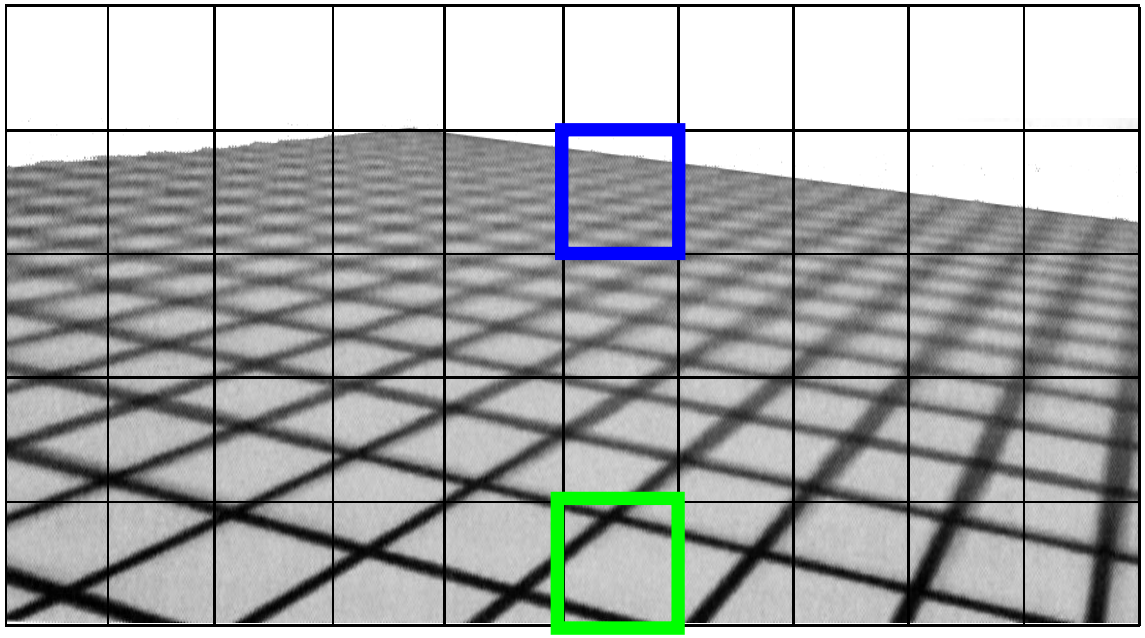
- Problem with mip mapping
  - Blurs in every direction equally
- Not optimal if distortion due to projection is not uniform

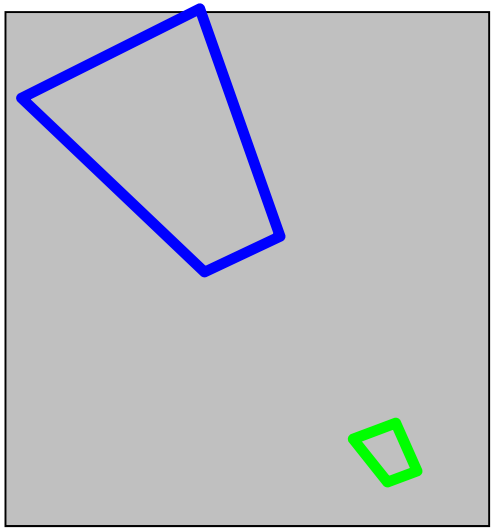


Anisotropic Filtering

More sharpness and clarity in the distance. Anisotropic filtering works by taking multiple texture samples on the suface and combining them to create higher quality textures.

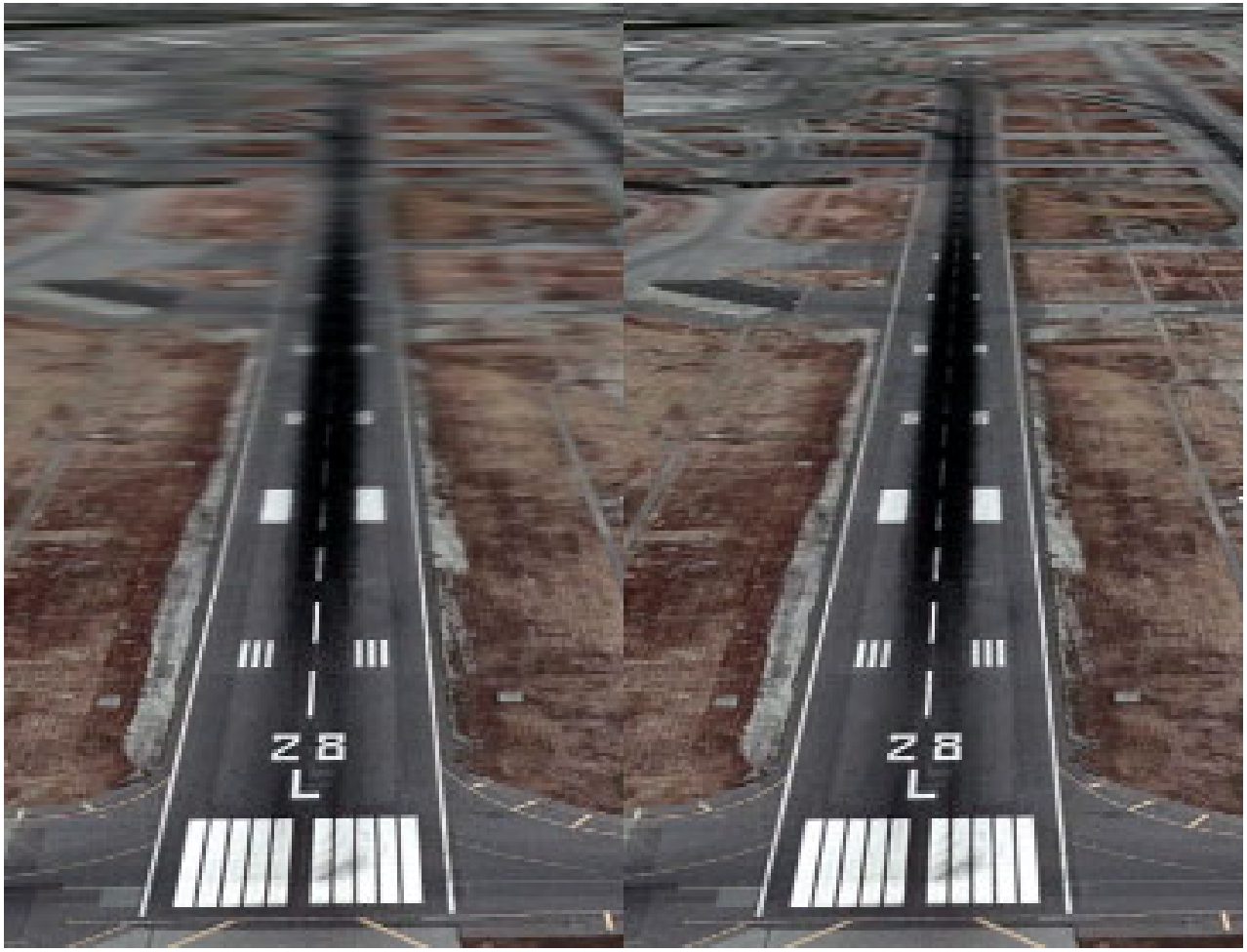
Needs a view-dependent filter kernel





Texture space

Anisotropic Filtering Example



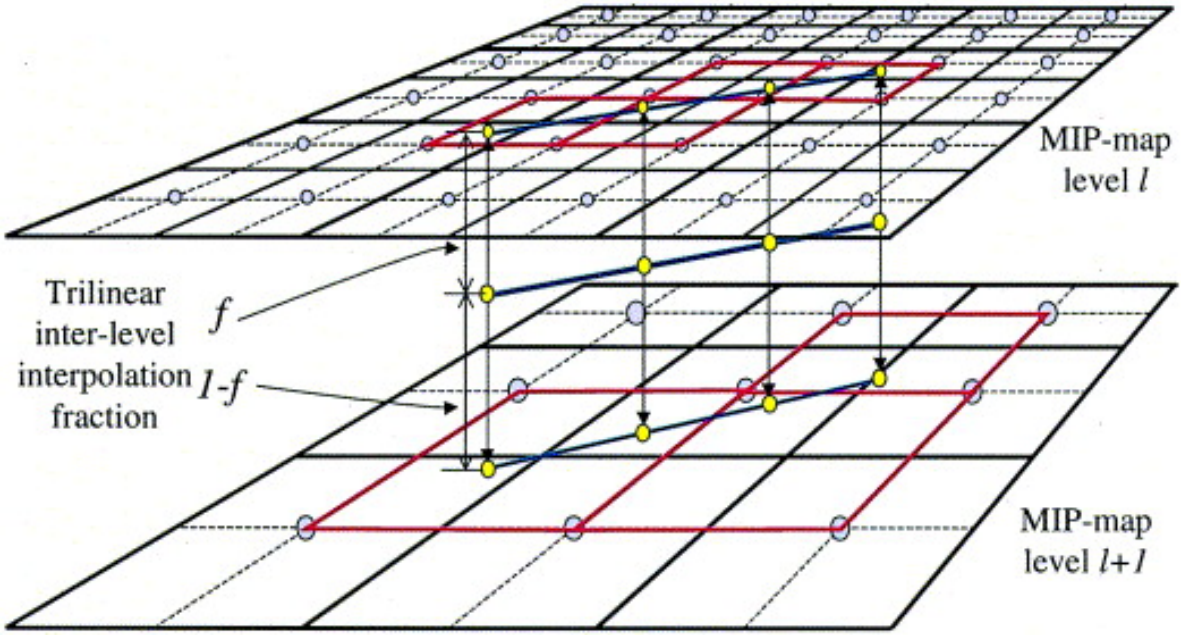
Mip mapped With anisotropic filtering

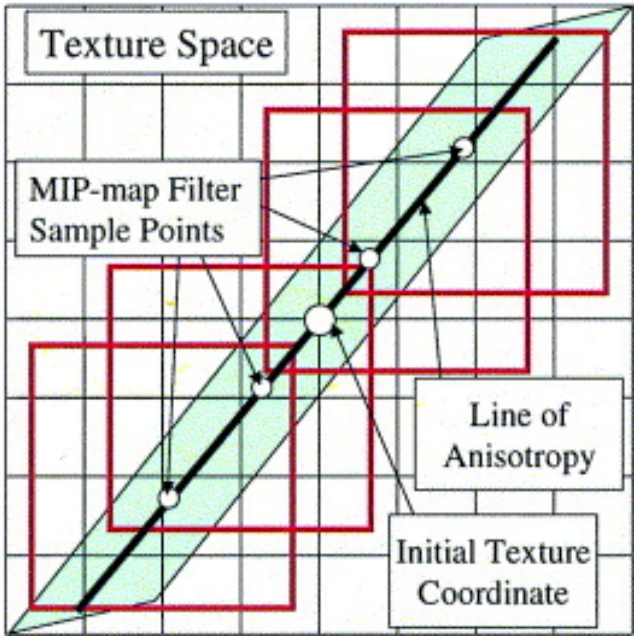
Rip Mapping

- Anisotropic filtering as extension to mip map
- Prefilter also anisotropic (non-uniform) scales
- Computed offline
- Needs 4x the memory

Anisotropic Real-Time Sampling

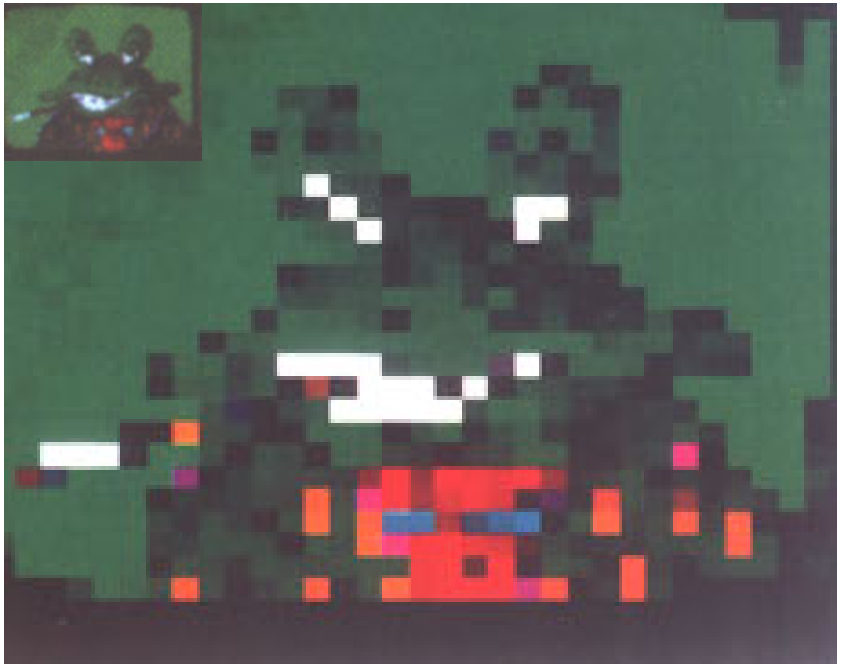
- Determine major line of anisotropy (=longer axis)
- Fixed number of sample along line of anisotropy
- Trilinear interpolation for each sample





Mip Mapping Upsampling

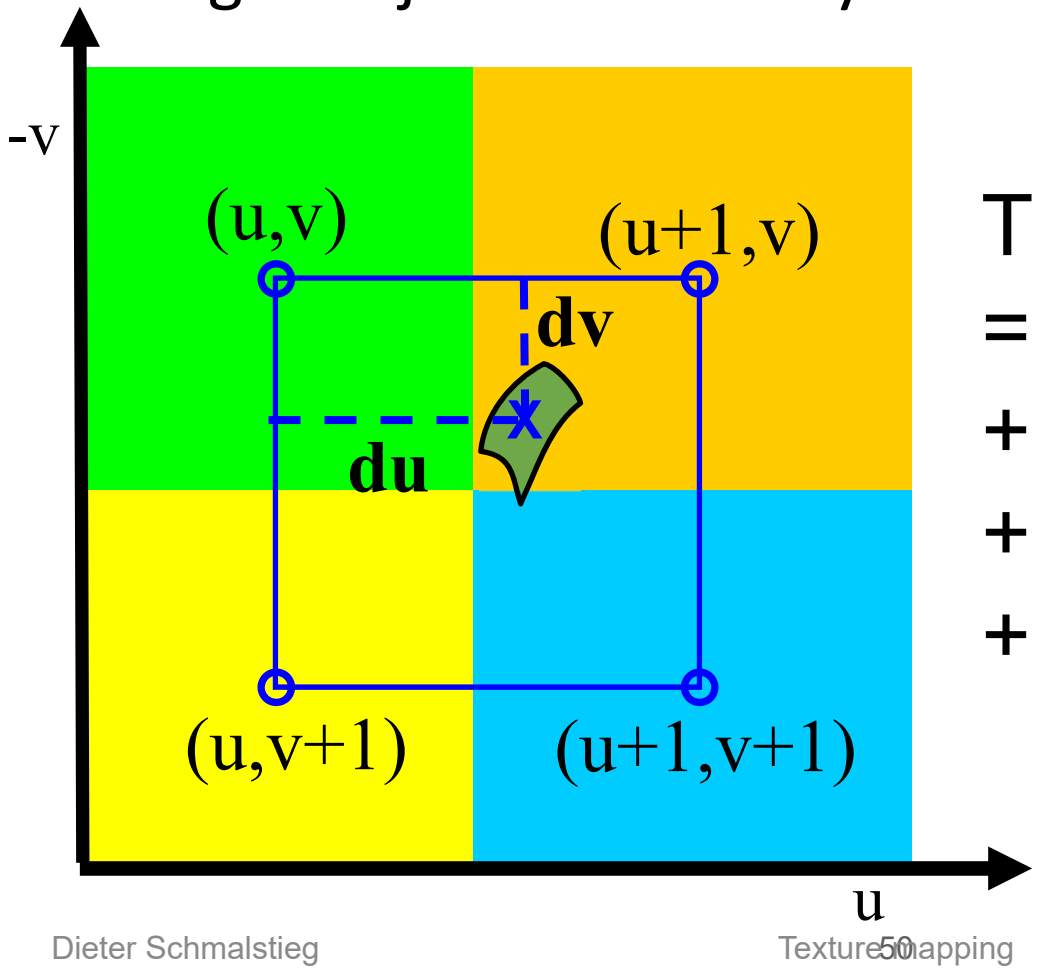
- When the texture resolution is too small: Upsampling of the mip level with highest res.
- „Magnification“



Mipmap ...

Bilinear Upsampling

- Bilinear reconstruction for texture magnification (*D<0*)
- Weight adjacent texels by distance to pixel position



$$T(u+du,v+dv) = du\cdot dv\cdot T(u+1,v+1) + du\cdot(1-dv)\cdot T(u+1,v) + (1-du)\cdot dv\cdot T(u,v+1) + (1-du)\cdot(1-dv)\cdot T(u,v)$$

Der Prozess besteht darin, jeden Pixel in der niedrig aufgelösten Textur mit den vier umgebenden Pixeln in der höher aufgelösten Textur zu interpolieren. Dies wird erreicht, indem die Farbwerte der umgebenden Pixel gewichtet nach ihrer Entfernung zum Zielpixel berechnet werden.

Bilinear Upsampling Result

Original image

Nearest neighbor

Bilinear filtering

Multipass Rendering

- Basic GPU lighting model is
  - Local
  - Limited in complexity
- Many effects possible with multiple passes
  - Environment maps
  - Shadow maps
  - Reflections, mirrors
  - Transparency

Multipass rendering ist eine Technik in der Computergrafik, bei der ein 3D-Szenenbild in mehreren Schritten oder "Passes" berechnet wird. Jeder Pass fokussiert sich auf einen bestimmten Aspekt des Bildes und die Ergebnisse werden zusammengeführt, um das endgültige Bild zu erstellen.

Multipass Rendering: How?

Two main methods

- Render to auxiliary buffers (Zwischenspeicher), use result as texture
  - E.g.: environment maps, shadow maps
  - Requires FBO support
- Redraw scene using fragment operations
  - E.g.: reflections, mirrors, light mapping
  - Uses framebuffer blending
  - Uses depth, stencil, alpha, … tests
- Can mix both techniques

Multipass via Render to Texture

Auxiliary buffer method = *render to texture* (RTT)

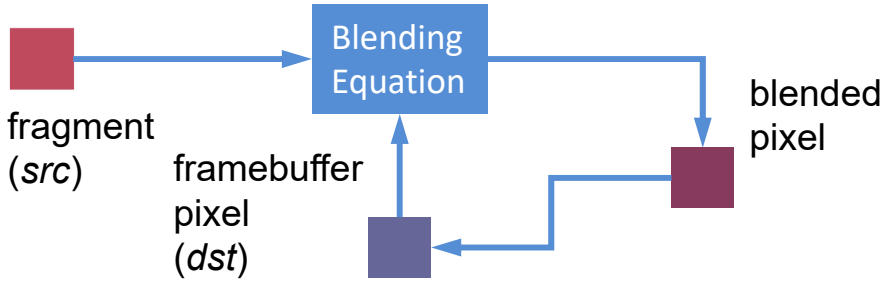
1. Frame buffer with color attachment is bound
2. Scene (or subset) is rendered,
   possibly using a custom shader program
3. Frame buffer is unbound
   Color attachment is bound as texture
4. Scene is rendered to the back buffer,
   Rendered texture can be used either by
   fixed function pipeline or a custom shader

Example: Fragment Shader Multiple Outputs

```
1  #version 330
2  layout(location = 0) out vec4 color0;
3  layout(location = 1) out vec4 color1;
4  void main()
5  {
6      color0 = …
7      color1 = …
8  }
```

Blending

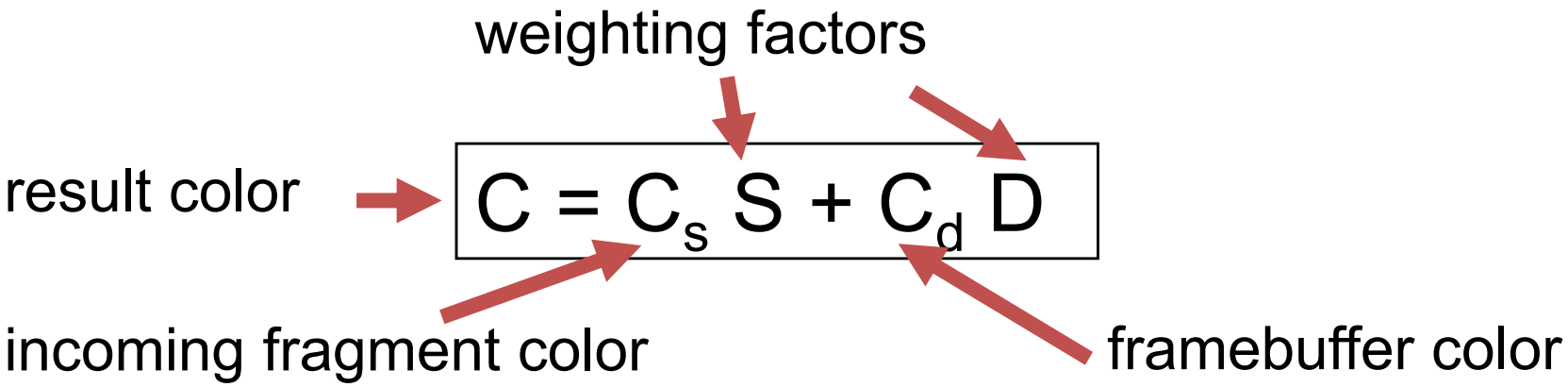
- So far, every new pixel simply overwrote existing framebuffer



- Blending combines new pixels with what is already in the framebuffer
- Blending does not always require alpha buffer

Multipass – Framebuffer Blending

- Select a blend equation
- Hardware supports a finite number of equations
- Most common is linear blending



Prüfungsfrage!

Other Blend Equations

- `GL_FUNC_ADD`

$$\mathbf{c}_d \leftarrow \mathbf{c}_s \otimes \mathbf{w}_s + \mathbf{c}_d \otimes \mathbf{w}_d$$

- `GL_FUNC_SUBTRACT`

$$\mathbf{c}_d \leftarrow \mathbf{c}_s \otimes \mathbf{w}_s - \mathbf{c}_d \otimes \mathbf{w}_d$$

- `GL_FUNC_REVERSE_SUBTRACT`

$$\mathbf{c}_d \leftarrow \mathbf{c}_d \otimes \mathbf{w}_d - \mathbf{c}_s \otimes \mathbf{w}_s$$

- `GL_FUNC_MIN`

$$\mathbf{c}_d \leftarrow \min(\mathbf{c}_s, \mathbf{c}_d)$$

- `GL_FUNC_MAX`

$$\mathbf{c}_d \leftarrow \max(\mathbf{c}_s, \mathbf{c}_d)$$

Blending Source and Destination

- Weights can be defined arbitrarily
- Alpha/color weights can be defined separately
- Choices of weight
  - *One*, *zero*, *dst color*, *src color*, *alpha*, 1 - *src color*…
- Example: transparency blending (window)

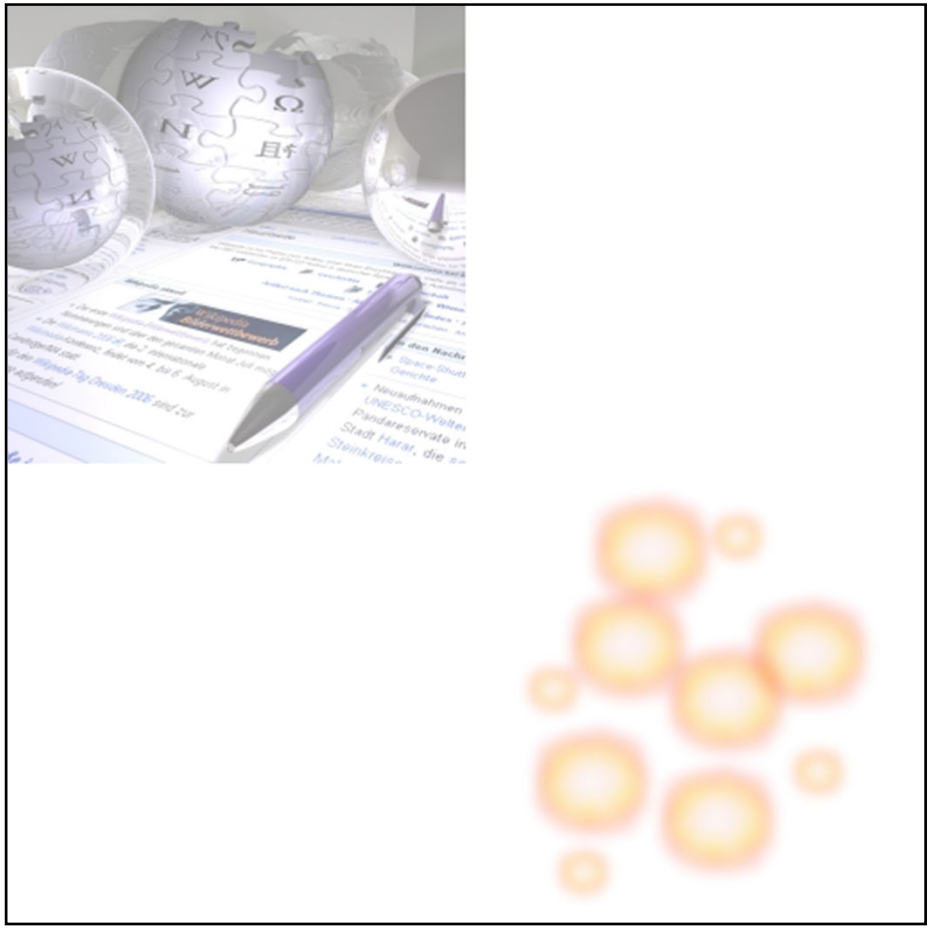
$$C = C_s \cdot \alpha + C_d \cdot (1- \alpha)$$

Alpha Buffer Transparancy Buffer

- Measure of opacity to
  - Simulate translucent objects (glass, water, etc.)
  - Composite images
  - Antialiasing
- Do not forget to enable blending first!
- Beware
  - Usage of alpha requires depth-sorting of objects
  - Z-Buffer not helpful for partially transparent pixels

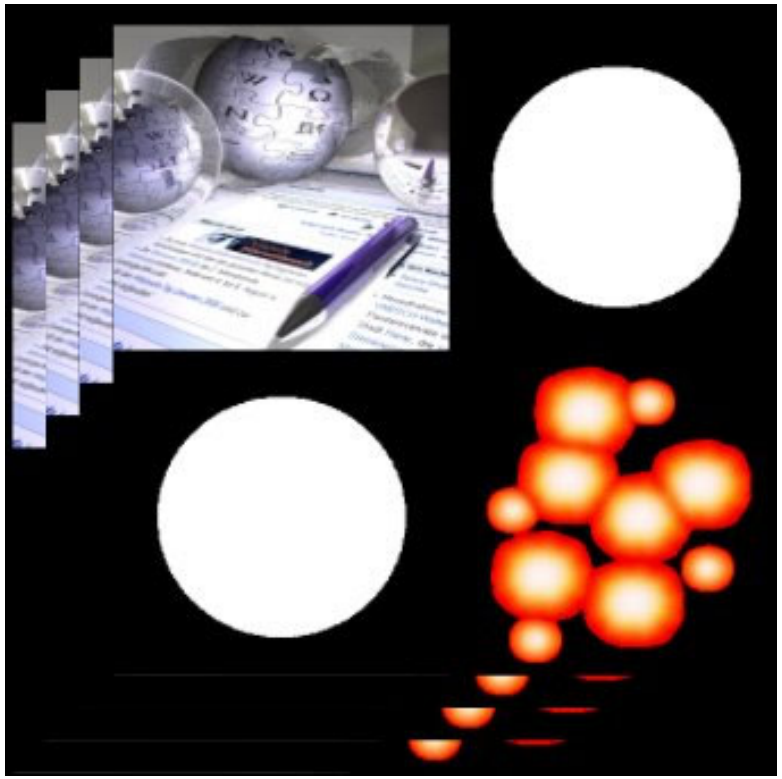
Blending Examples (1)

Texture with alpha channel for the following examples…



Blending Examples (2)

- glBlendFunc(GL_ONE, GL_ZERO);
  - Everything works as if blending was disabled
  - Alpha values also ignored

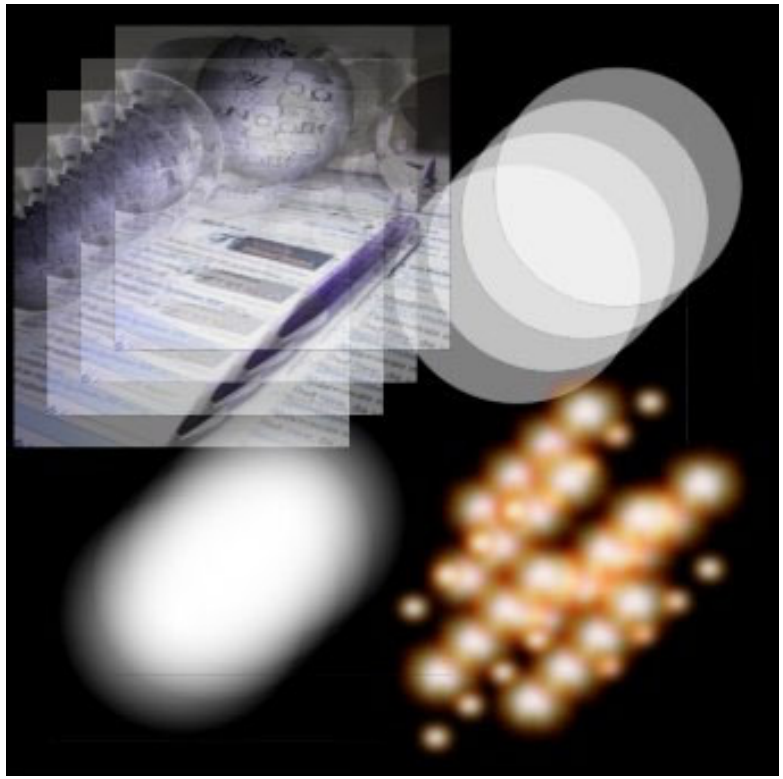


Blending Examples (3)

- `glBlendFunc(GL_ZERO, GL_ONE);`
  - Nothing is drawn
  - Not really useful…

Blending Examples (4)

- `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);`
  - Most common blending method
  - Simulates traditional transparency
  - Source color's alpha (usually from texture) defines opaqueness of object to render



Blending Examples (5)

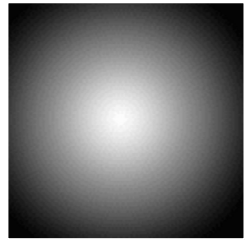
- `glBlendFunc(GL_ONE, GL_ONE);`
  - Source and destination color are simply added
  - Values >1.0 are clamped to 1.0 (saturation)
  - Good for fire effects

Multitexturing

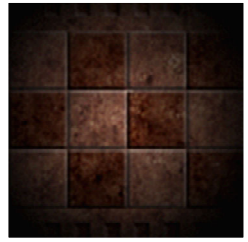
- Idea: Apply multiple textures in one pass
- Textures can be combined arbitrarily (add, mul, …)
- Indirect lookup
  - 1st texture lookup is used to index into 2nd texture
- Today, just use multiple *texture samplers* in a fragment shaders
- Texture samplers correspond to dedicated hardware units for accessing texture memory

Light Mapping

×

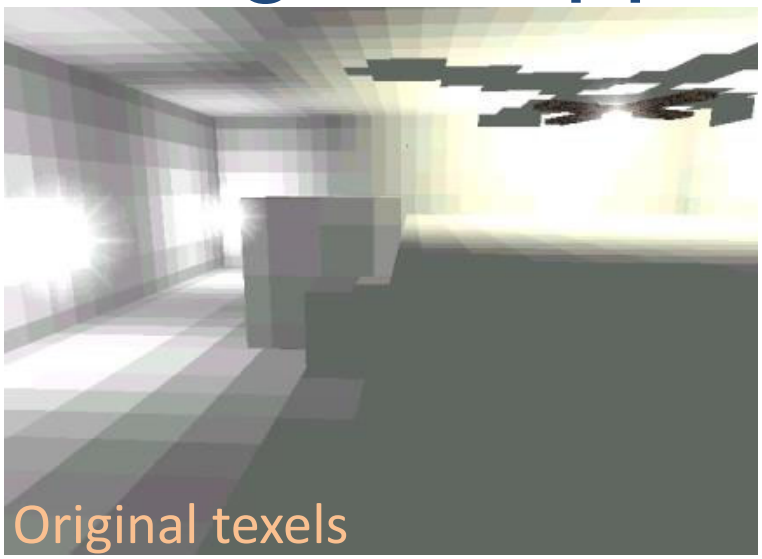


=

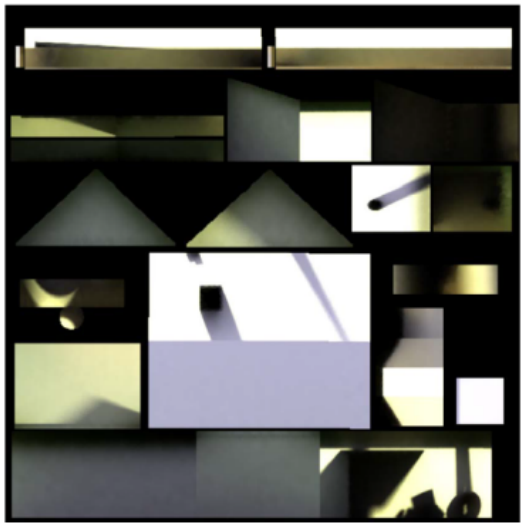


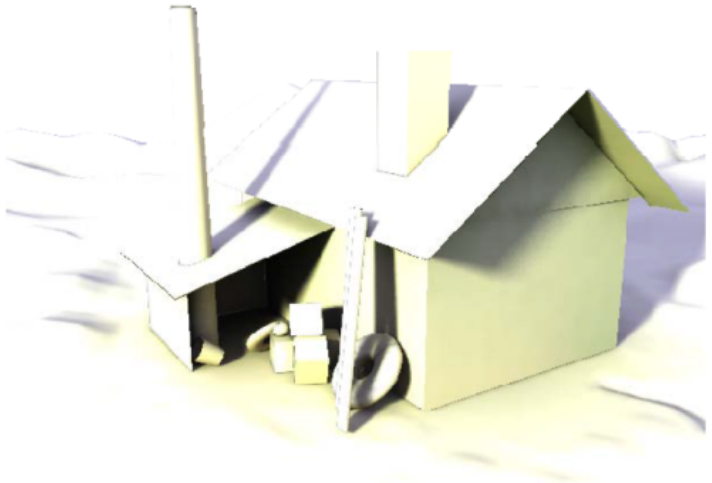
- Used in most first-person shooters
- Precalculate (*bake*) diffuse lighting on static objects
  - Only low resolution necessary
  - Diffuse lighting is view independent!
- Advantages
  - No runtime lighting necessary (good for older cards)
  - More accurate than vertex lighting (no need to tesselate for highlights)
  - Can take global effects (shadows, radiosity) into account

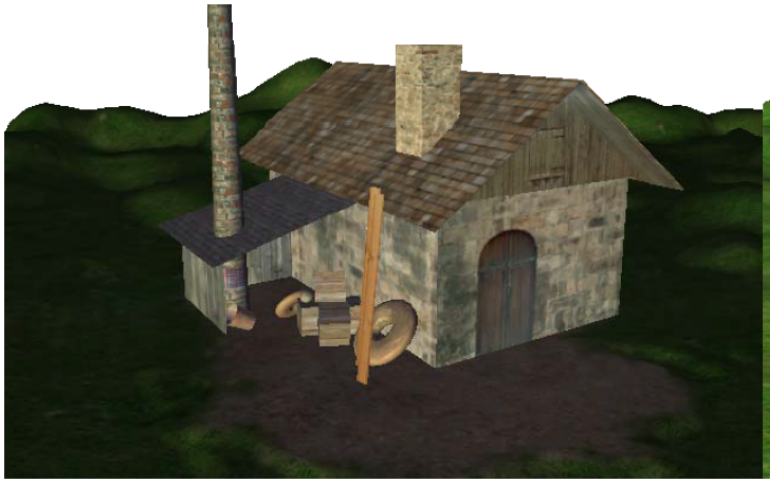
Light Mapping Example 1



Light Mapping Example 2





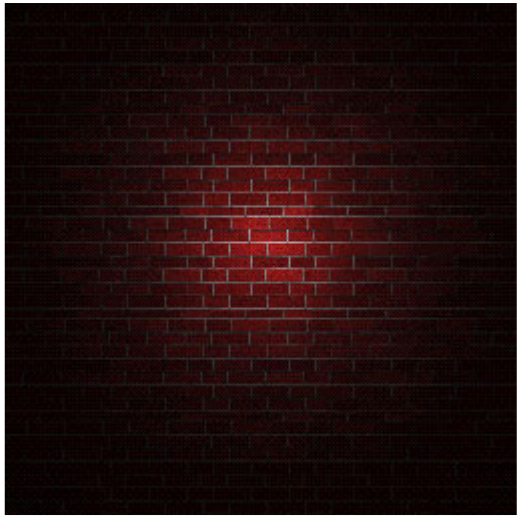


Light Mapping Procedure

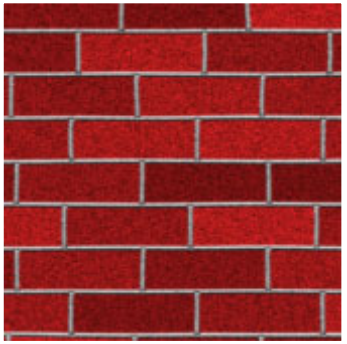
- Map generation
  - Use single map for group of coplanar polys
  - Map back to worldspace to calculate lighting
- Map application
  - Premultiply textures by light maps
    - Large textures, no dynamics
  - Multitexturing at runtime
    - Fast, flexible

Light Mapping Issues

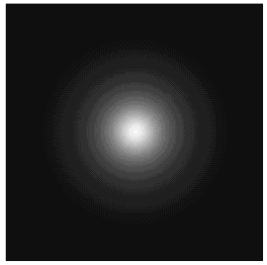
- Why premultiplication is bad…



vs.



+



Full Size Texture
(with Lightmap)

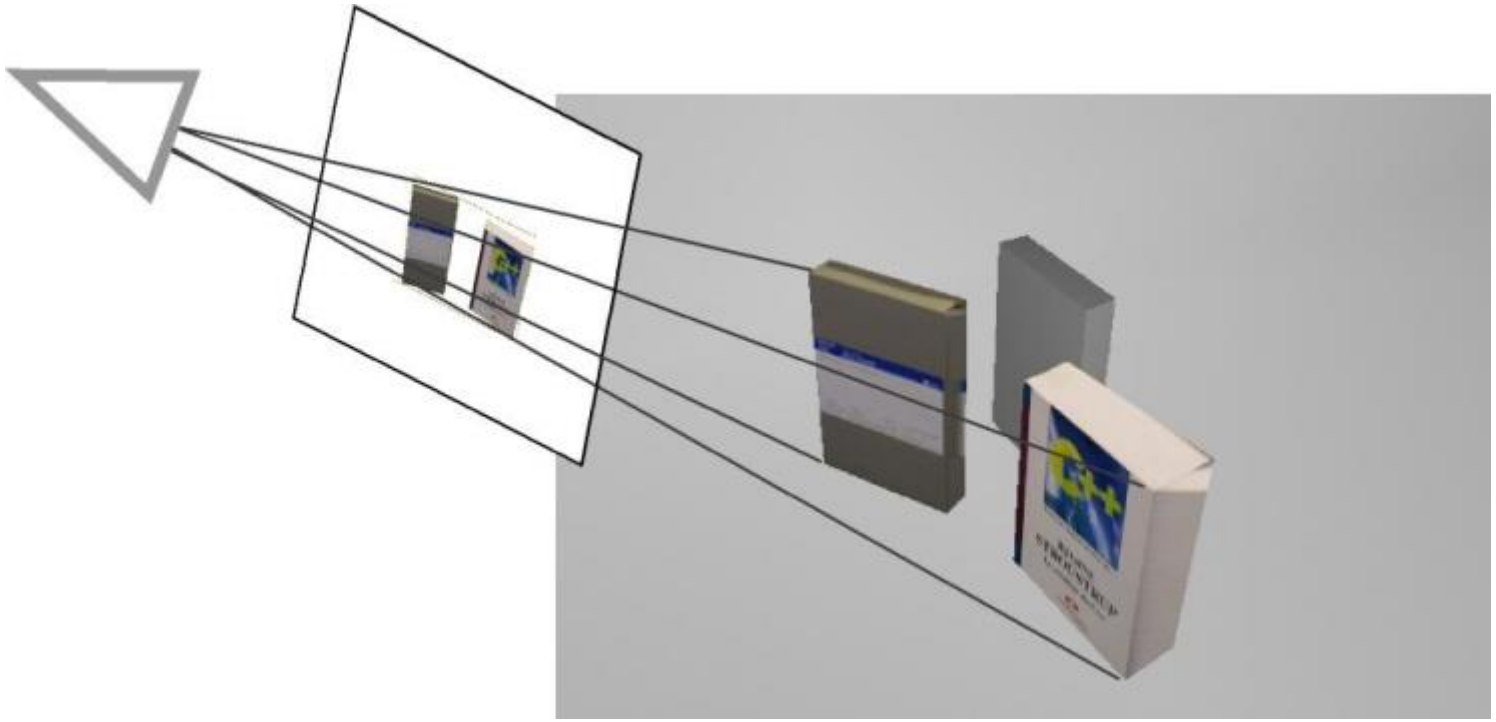
Tiled Surface Texture
plus Lightmap

→ Use small surface textures and small maps

→ Maybe calculate low-res light maps on the fly

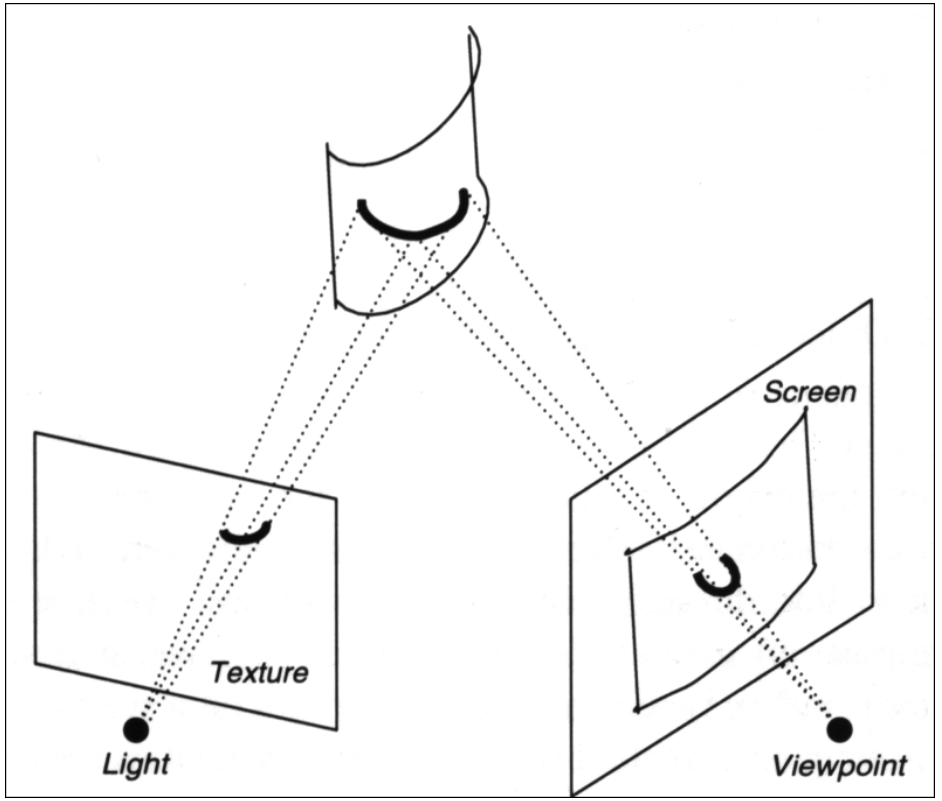
Projective Texture Mapping

- PTM simulates a video projector
  - Or a flashlight, slide projector, …
- Using perspective projection



PTM: Geometry

- Map object coordinates to light frustum
- Express this as a projective transform
- Usually a 4x4 matrix
- Similarity to video projector observed by camera



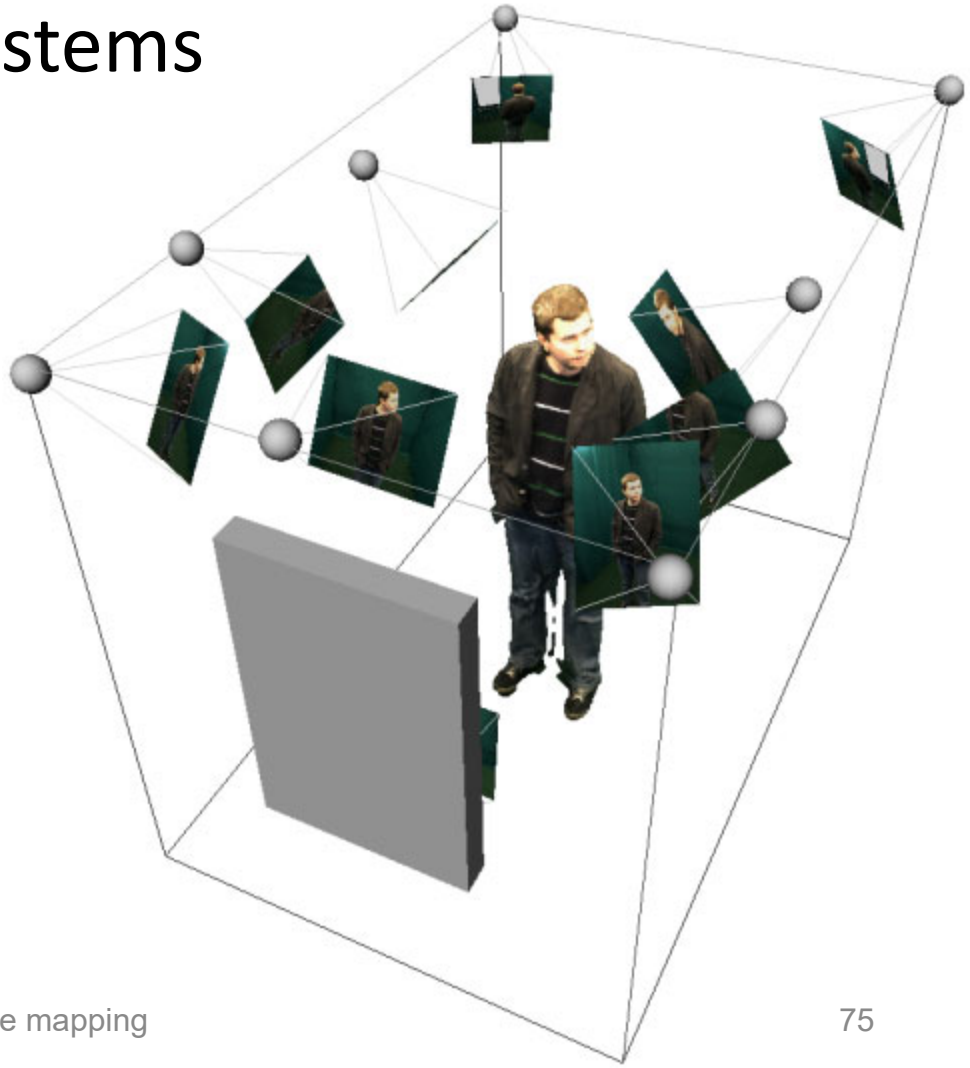
PTM: Flashlights Examples

STALKER

Doom 3

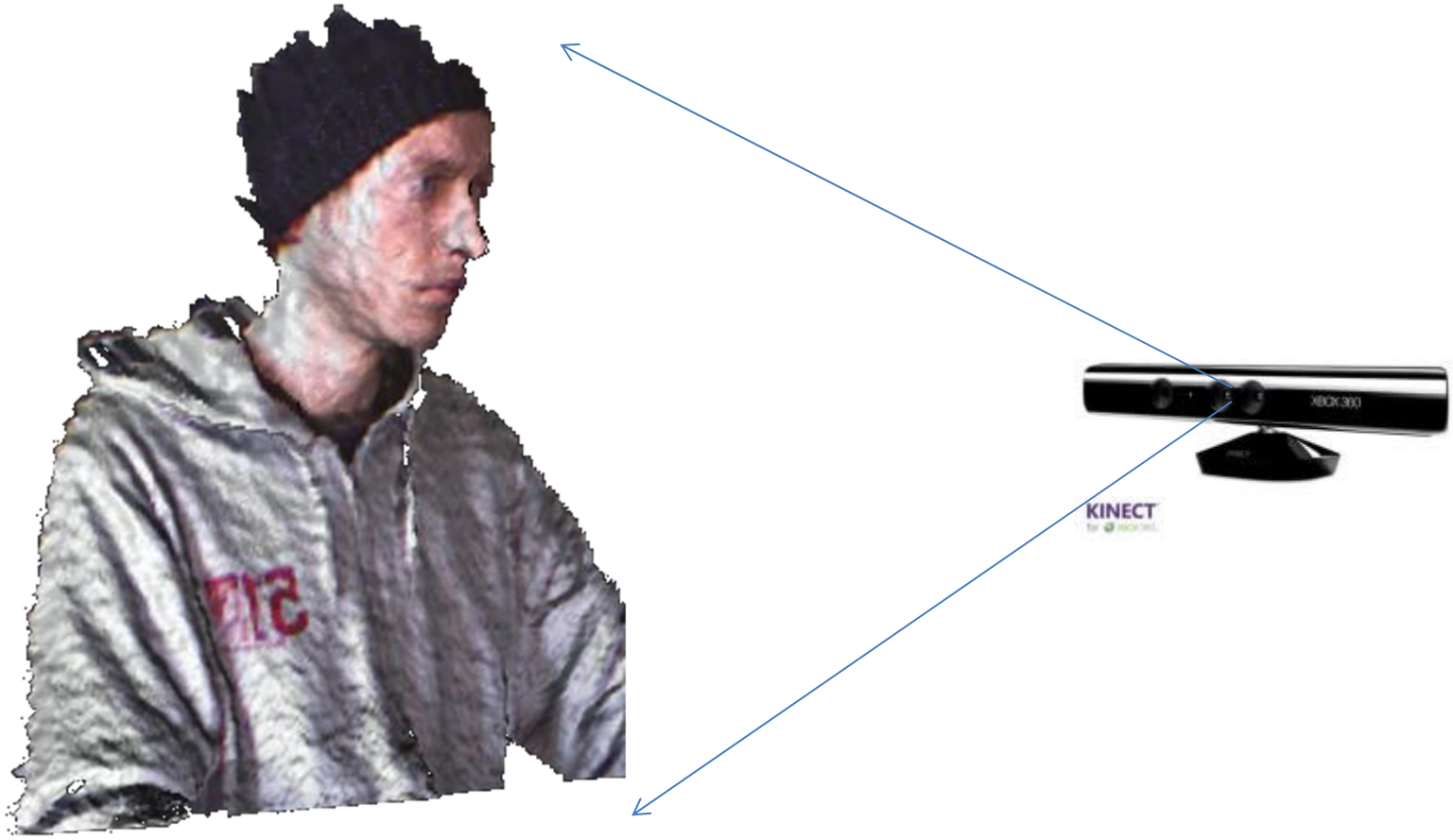
PTM from Multi-Camera

User in multi-camera systems



PTM from Depth Camera

Example: display a Microsoft Kinect point cloud

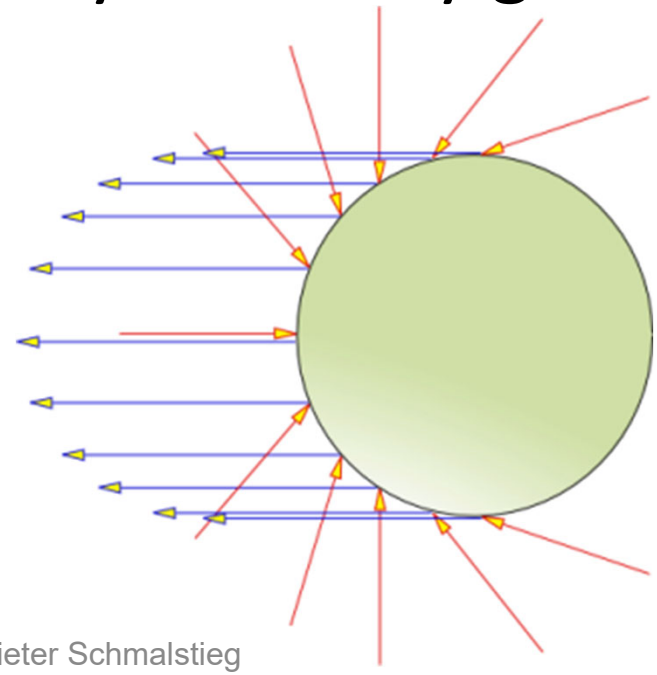


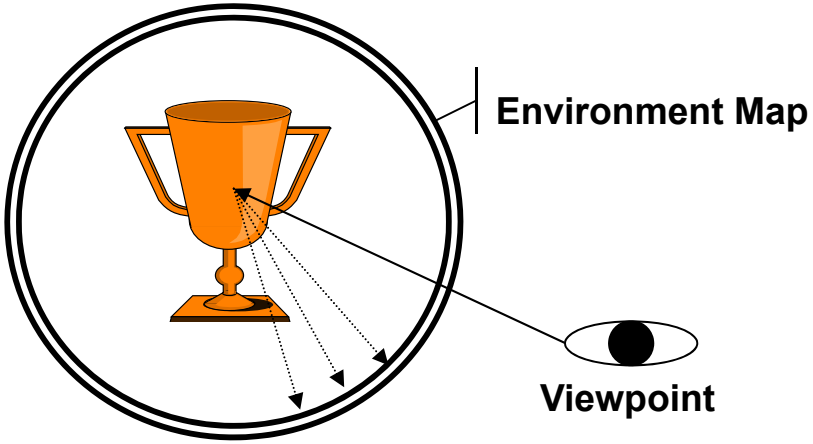
Environment Mapping

- Idea: use texture to create reflections
- Uses texture coordinate generation, multitexturing, RTT, …

Environment Mapping Indexing

- Assumption: index envmap via orientation →
  - Reflecting object shrunk to a single point
  - Or: environment infinitely far away
- Eye not very good at discovering the fake

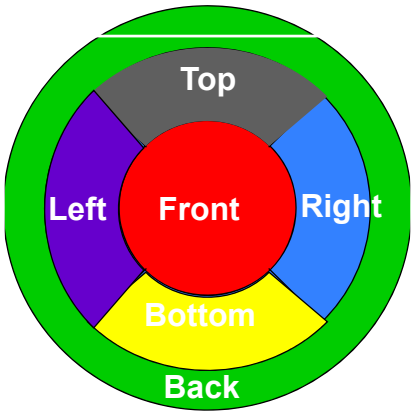




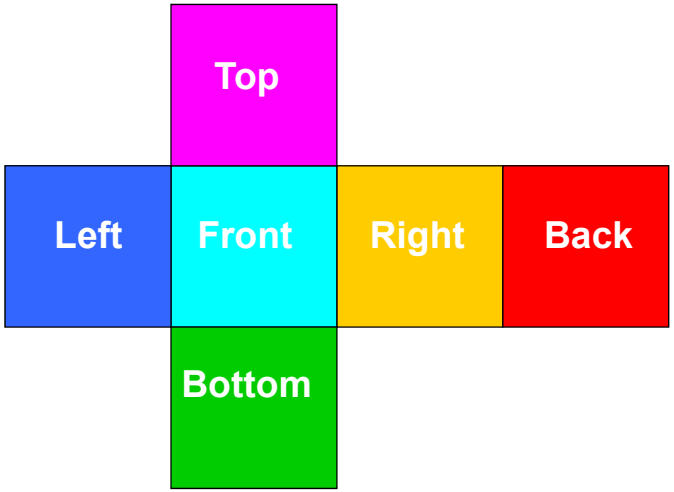
Environment Mapping Types

- Typical mappings:
  Each maps all directions to a 2D texture

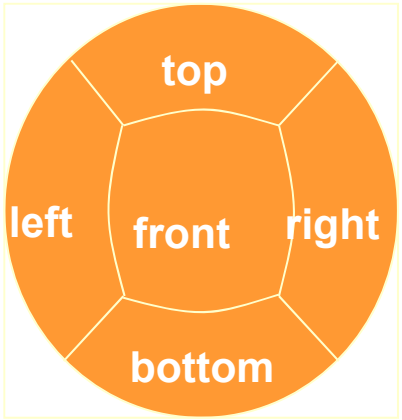
Sphere



Cube

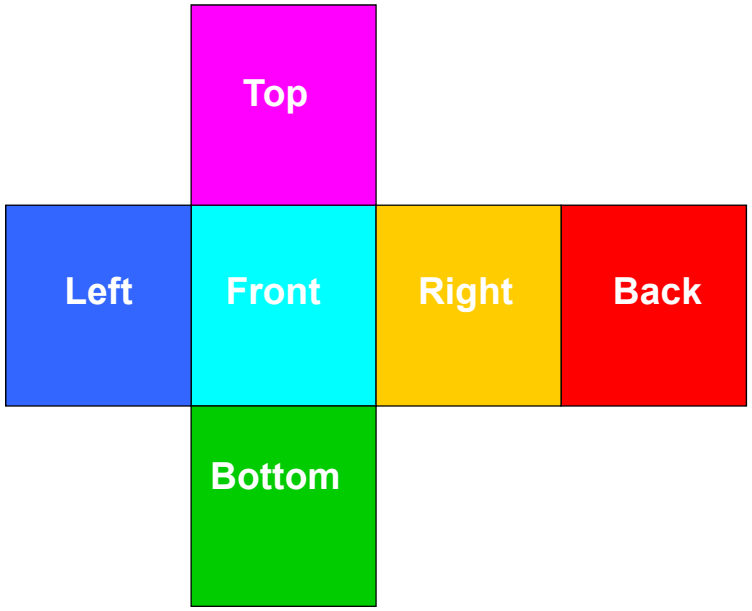


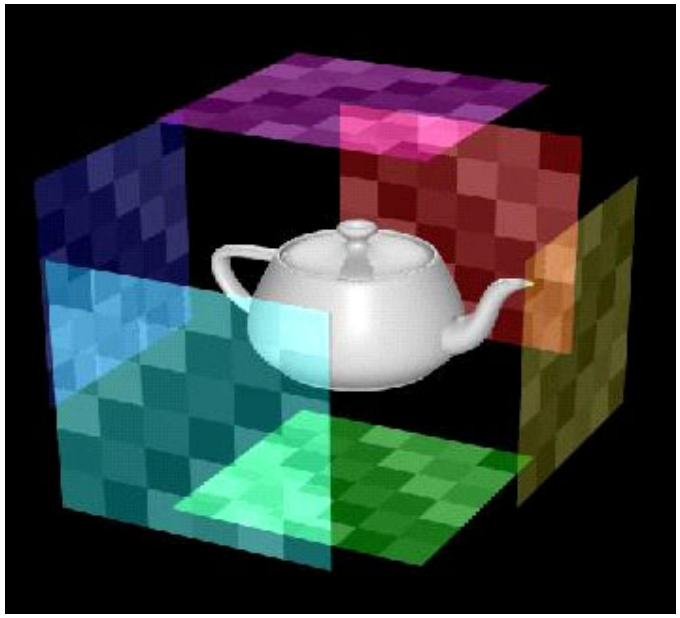
Dual paraboloid



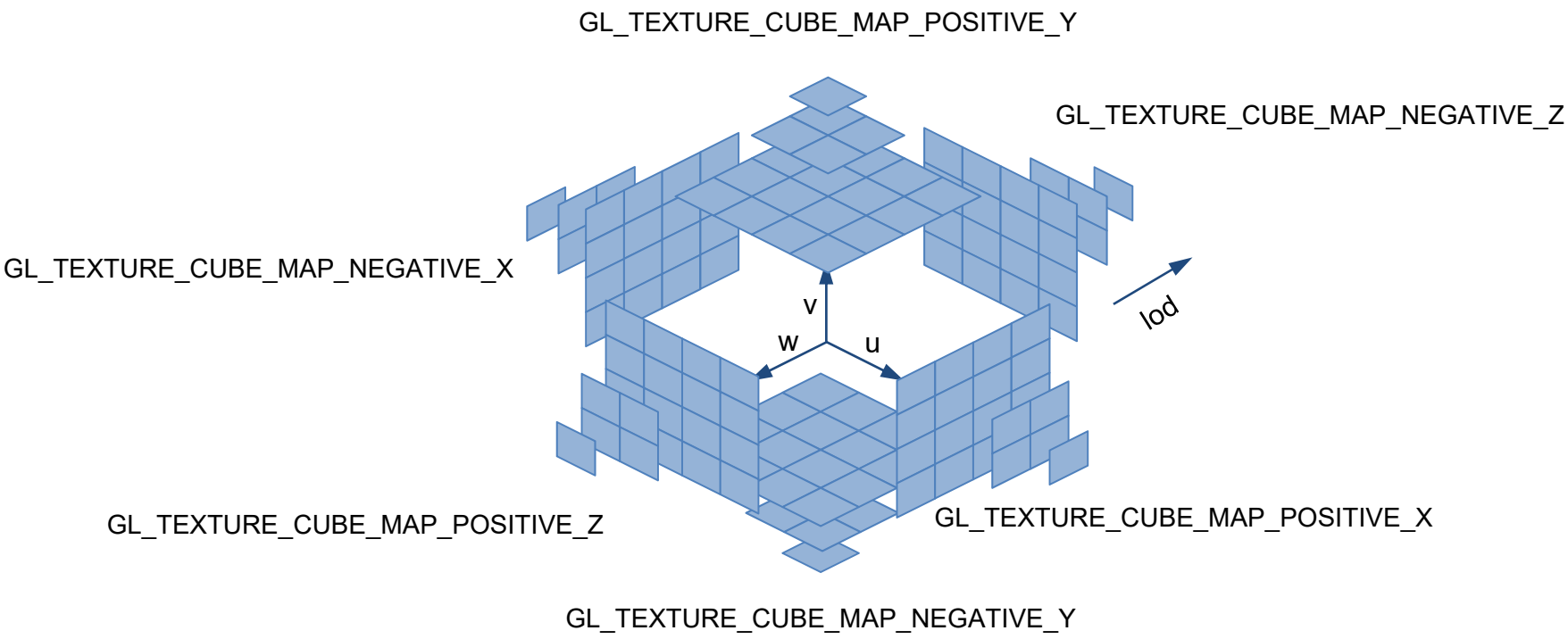
Cube Mapping

- Implemented in the OpenGL pipeline

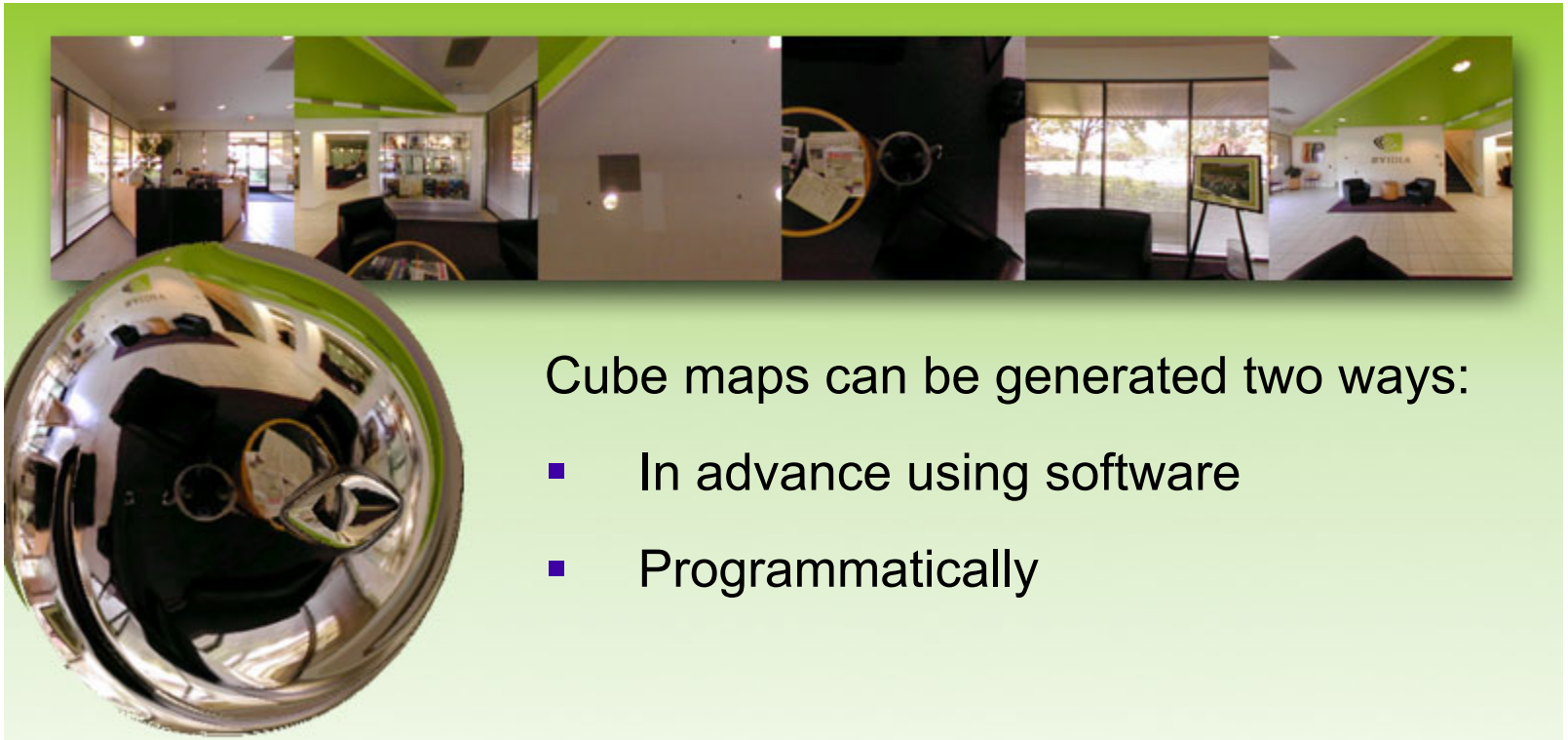




OpenGL Cube Map Targets



Cube Map Generation



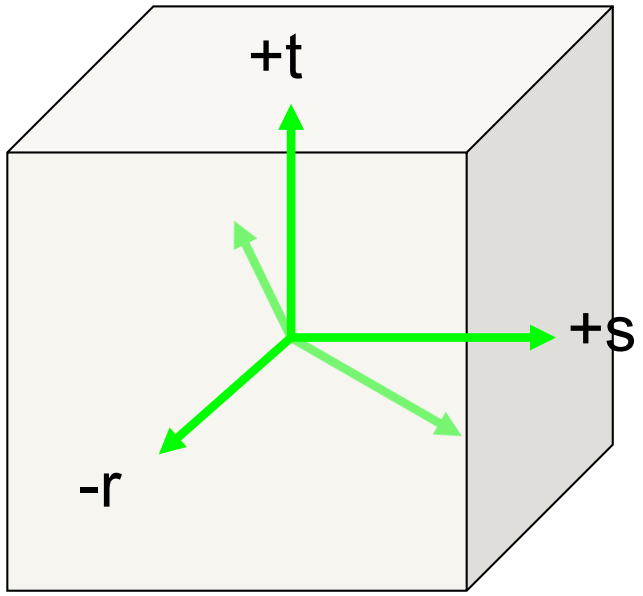
Cube maps can be generated two ways:

- In advance using software
- Programmatically

Cube maps are the primary method of generating reflections in real-time 3D

Cube Map Addressing

- Cube map accessed via **vectors** expressed as 3D texture coordinates (s, t, r)



Cube Map Address Calculation

- 3D → 2D projection done by hardware
- Correct ray/quad intersection too slow, instead
  - Highest magnitude component selects which cube face to use (e.g., -*t*)
  - Divide other components by this, e.g.:
    $s' = s / -t$
    $r' = r / -t$
  - (s', r') select a texel from this face
- Does not define environment mapping itself

  → Need to generate useful texture coordinates

Cubic Environment Mapping

- Generate views of the environment
  - One for each cube face
  - 90° view frustum
  - Use hardware to render directly to a texture
- Use reflection vector to index cube map
  - Use hardware mode for sampling in cube map

Reflection Vector

- Angle of incidence = angle of reflection

$$R = V - 2 (N N^T) V$$

- OpenGL uses eye coordinates for R
- Cube map needs reflection vector in world coordinates (where map was created)

→ Load texture matrix with inverse 3x3 view matrix

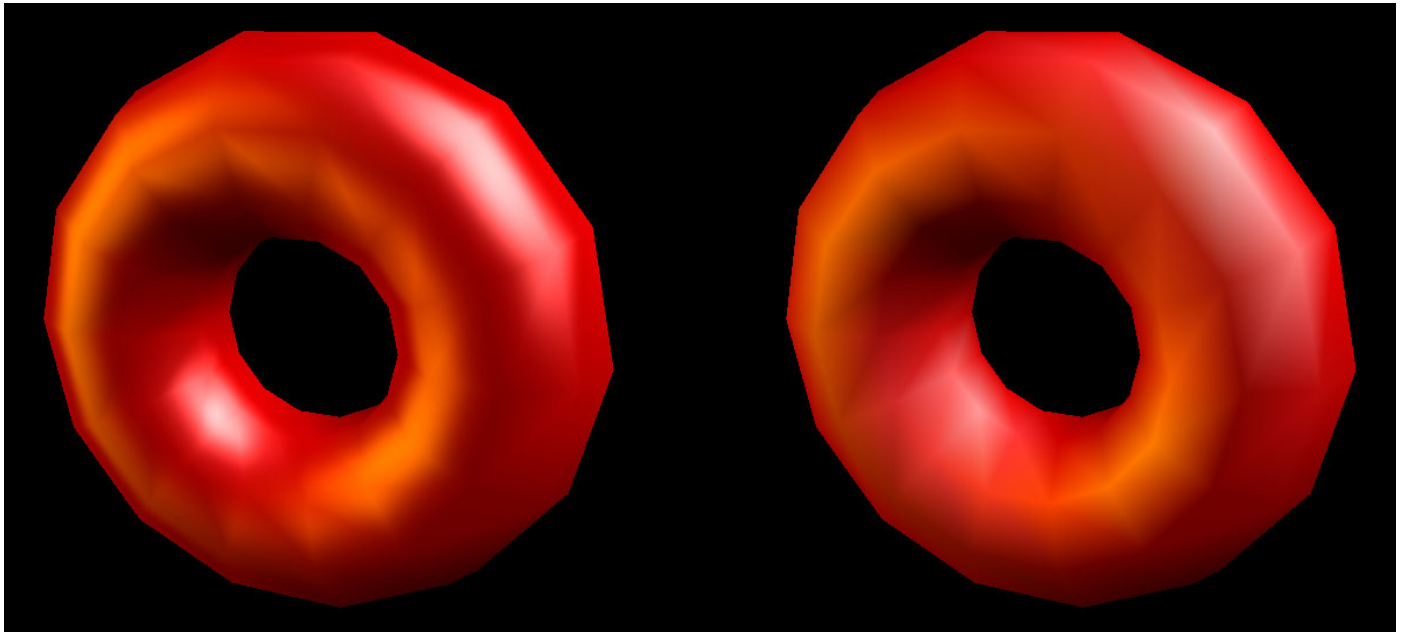
Cube Mapping Discussion

- Advantages
  - Minimal distortions
  - Creation and map entirely hardware accelerated
  - Can be generated dynamically
- Optimizations for dynamic scenes
  - Need not be updated every frame
  - Low resolution sufficient

Per-Pixel Lighting

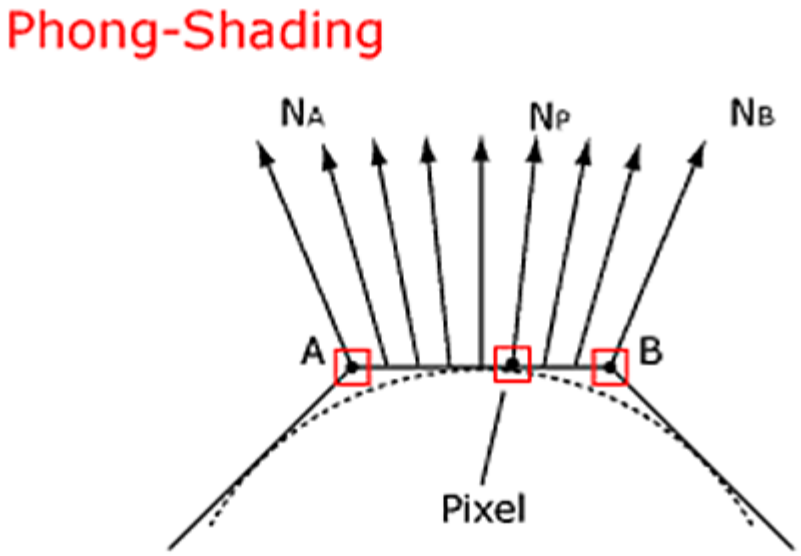
- Simulating smooth surfaces by calculating illumination at each pixel (e.g., Phong shading)
- Advantage: better specular highlights

per-pixel evaluation linear intensity interpolation



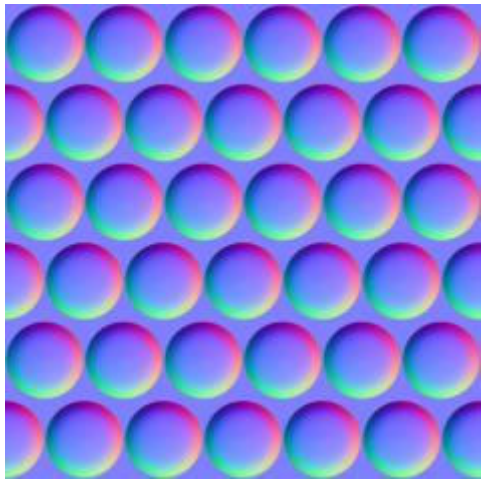
Per-Pixel Lighting Algorithm

- Done in fragment shader programs
- Vertex normals are passed to fragment shader
- Rasterizer interpolates them

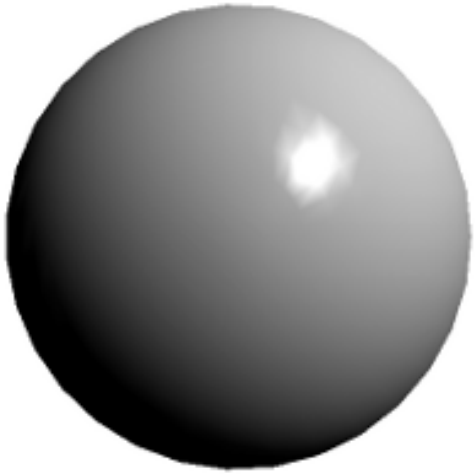


Bump Mapping

- Per-pixel normals are a prerequisite for BM
- Simulating rough surfaces by calculating per-pixel illumination
- Replace geometry with (faster) texture

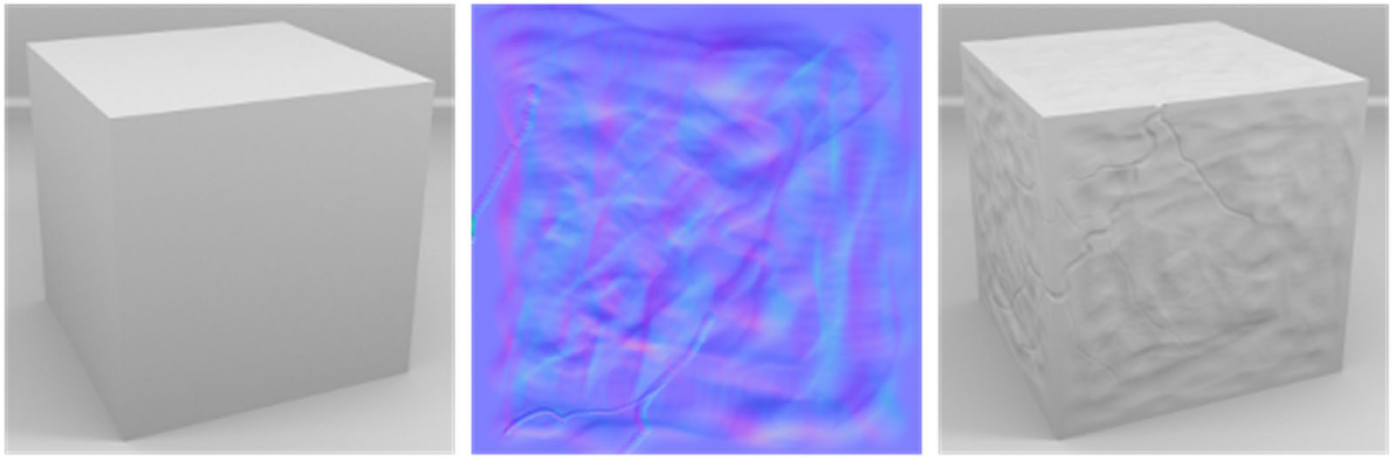


+



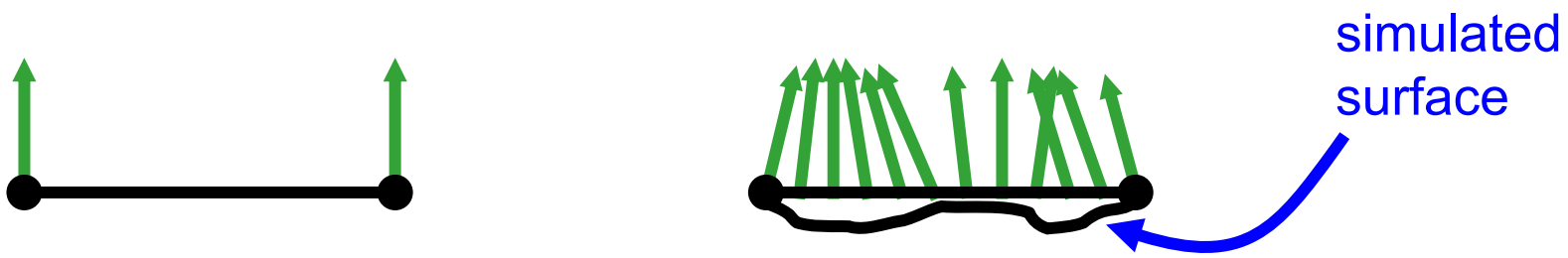
=

Bump Mapping Examples



Bump Mapping Basics

- Original idea from ray tracing [Blinn 1978]
- Simulate surface features with illumination only
- Instead of interpolating normals, take them from a map

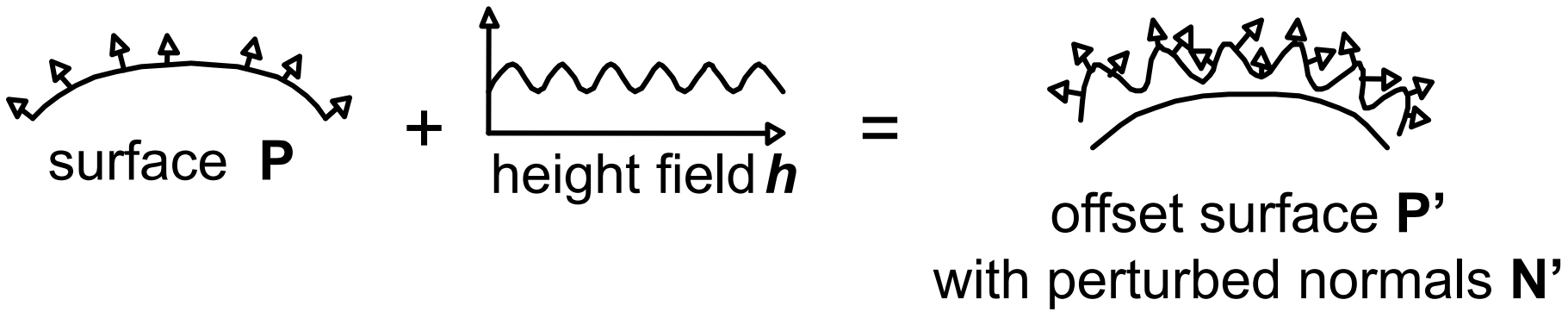


Per-Pixel vs. Bump Mapping

- Where is the difference?
- Per-pixel lighting = bump mapping *without the bump map!*
- Per-pixel lighting is a special case where normal vector is interpolated instead of looked up

Bump Mapping Basics

- Assume a (u,v)-parameterization
  - Points **P** on the surface given as **P**(u,v)
- Surface **P** is modified by 2D height field h



Mathematics of Bump Mapping

- Displaced surface:

$$\mathbf{p}'(u,v) = \mathbf{p}(u,v) + h(u,v)\mathbf{n}(u,v)$$

- Partial derivatives:

$$\frac{\partial \mathbf{p}'}{\partial u} = \frac{\partial \mathbf{p}}{\partial u} + \frac{\partial h}{\partial u} \cdot \mathbf{n} + h \cdot \underbrace{\frac{\partial \mathbf{n}}{\partial u}}$$

$$\frac{\partial \mathbf{p}'}{\partial v} = \frac{\partial \mathbf{p}}{\partial v} + \frac{\partial h}{\partial v} \cdot \mathbf{n} + h \cdot \underbrace{\frac{\partial \mathbf{n}}{\partial v}}$$

$= 0$ for locally flat base surface

- Perturbed normal:

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

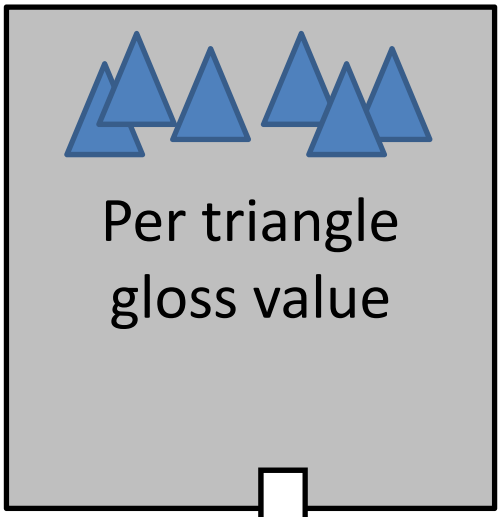
after some computation…

$$\Rightarrow \mathbf{n}' = \mathbf{n} + \frac{\partial h}{\partial u} \cdot \left(\mathbf{n} \times \frac{\partial \mathbf{p}}{\partial v}\right) - \frac{\partial h}{\partial v} \cdot \left(\mathbf{n} \times \frac{\partial \mathbf{p}}{\partial u}\right)$$

Multitexturing Example

Bump Map

Diffuse Color



Multitexturing Result

Note: Gloss map defines where to apply specular

Final result!

Bump Mapping Representations

- Height fields
  - Must approximate derivatives during rendering
  - Simple case: emboss bump mapping
- Offset maps
  - Encode orthogonal offsets from unperturbed normal
  - e.g.: $\left(\frac{\partial h}{\partial u}, \frac{\partial h}{\partial v}\right)$, or whole vector, …
  - Require renormalization
  - Enable bump environment mapping
  - Calculate using finite differencing on height field
- Normal (perturbation/rotation) maps
  - Encode direction vectors (in whatever space)
  - No normalization required
  - Standard method

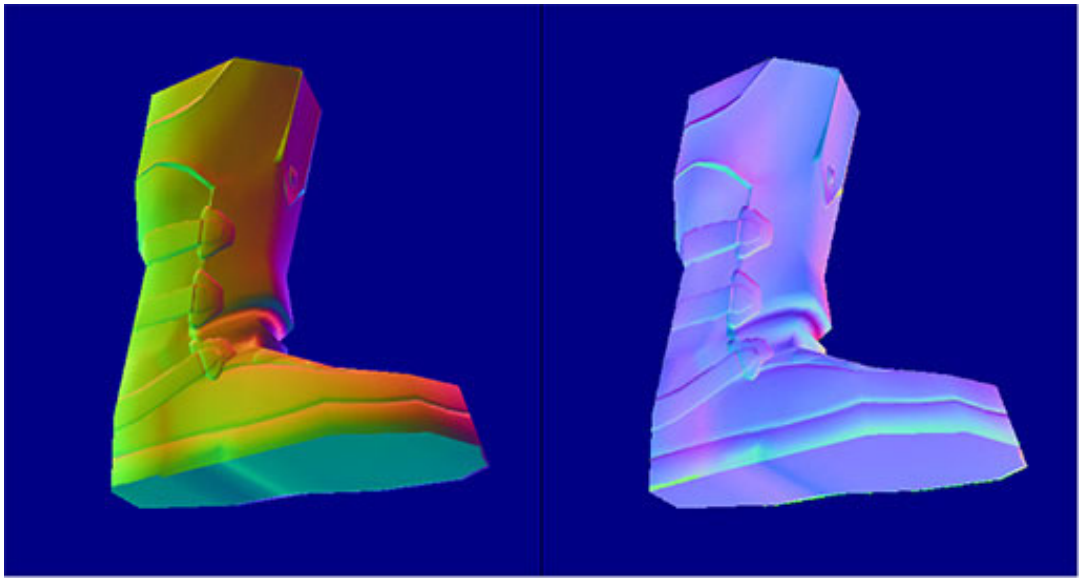
Coordinate Systems

Problem: where to calculate lighting?

- Object coordinates
  - Native space for object coordinates ($\mathbf{P}$)
- World coordinates
  - Native space for light vector ($\mathbf{l}$), env-maps
  - Not explicit in OpenGL
- Eye Coordinates
  - Native space for view vector ($\mathbf{v}$)
- Tangent Space
  - Native space for bump/normal maps ($\mathbf{n}$)

Tangent Space

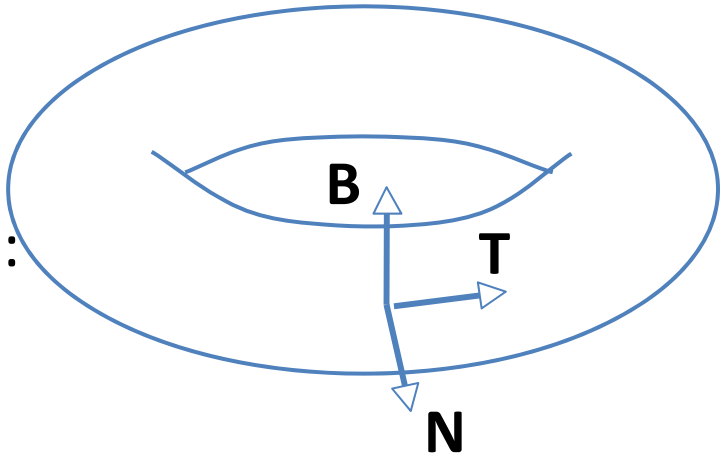
- Tangent space is most often used
- This variant is called normal mapping
- Normal maps usually blue-ish in RGB space, because perturbed normals are still mostly up-facing (0,0,1)



Object space Tangent space

Tangent Space Basics

- Concept from differential geometry
- Set of all tangents on a surface
- Orthonormal coordinate system (frame) for each point on the surface:



$$\mathbf{n}(u,v) = \frac{\partial \mathbf{p}}{\partial u} \times \frac{\partial \mathbf{p}}{\partial v}$$

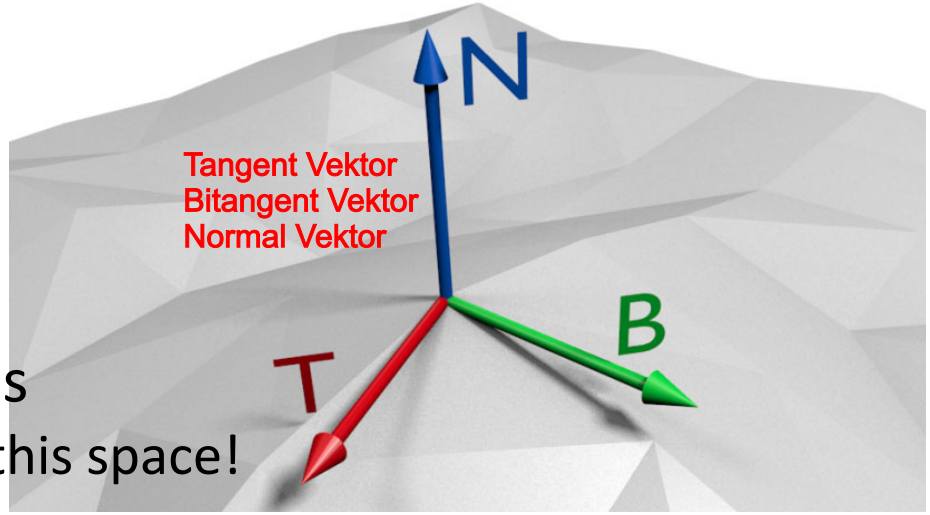
$$\mathbf{t} = \frac{\partial \mathbf{p}}{\partial u} \qquad \mathbf{b} = \frac{\partial \mathbf{p}}{\partial v}$$

- Square patch assumption:

$$\mathbf{b} = \mathbf{n} \times \mathbf{t}$$

(watch out for handedness!)

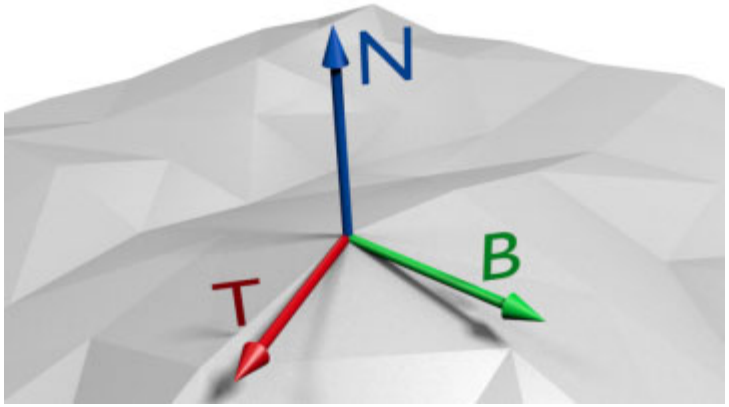
- A natural space for normal maps
  - Vertex normal $\mathbf{n} = (0,0,1)^T$ in this space!



Tangent Space Matrix

- Every vertex needs these three vectors
  - E.g., in attribute arrays
- They are interpolated for each fragment
- Fragment program can set up tangent space matrix **TSM**

$$\mathbf{TSM} = \begin{pmatrix} T_x & B_x & N_x & 0 \\ T_y & B_y & N_y & 0 \\ T_z & B_z & N_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Tangent Space Algorithm

- For each vertex
  - Transform **l** and **v** with **TSM** and normalize
  - Compute normalized **h** from **v** and **l**
- For each fragment
  - Interpolate **l** and **h**
  - Renormalize **l** and **h**
  - Fetch **n**’ = texture(u,v)
  - Compute max(**l** · **n**’, 0) and max(**h** · **n**’, 0)^s
  - Combine using standard Phong equation

Der Tangent Space Algorithmus wird verwendet, um Normalenvektoren auf einer 3D-Oberfläche zu berechnen. Ein Normalenvektor ist ein Vektor, der senkrecht auf der Oberfläche eines Objekts verläuft und die Richtung anzeigt, in die die Oberfläche ausgerichtet ist. Diese Normalenvektoren sind für die Berechnung von Beleuchtung und Schattierung erforderlich, da sie bestimmen, wie Licht auf ein Objekt trifft und wie es reflektiert wird. Der Tangent Space Algorithmus berechnet die Normalenvektoren an einem gegebenen Punkt auf der Oberfläche, indem er die lokale Tangentialebene an diesem Punkt bestimmt und dann den Normalenvektor auf dieser Ebene berechnet.

Reflective Bump Mapping

- EMBM (Environment Map Bump Mapping)

EMBM in World Space

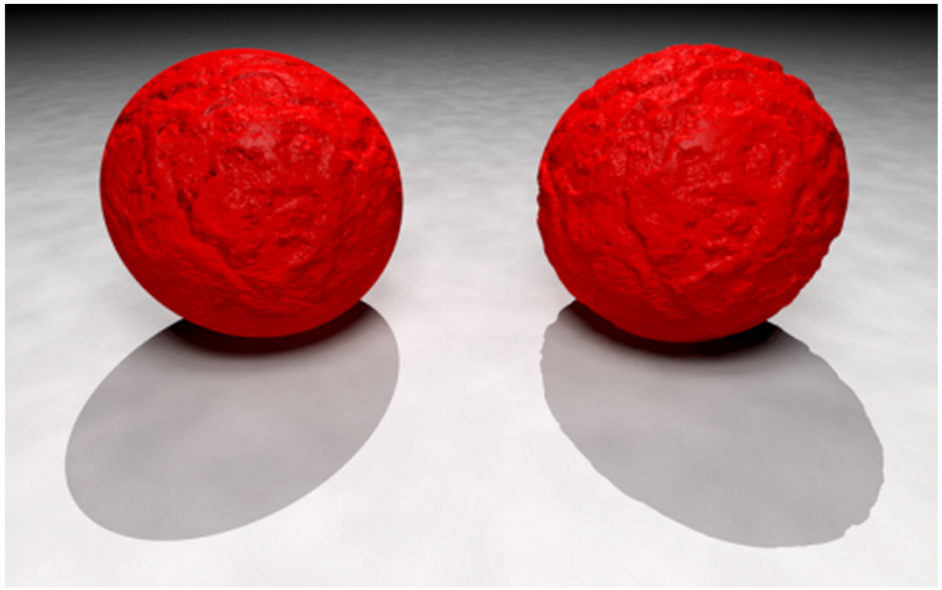
Environment Map Bump Mapping ist eine Technik in der Computergrafik, die verwendet wird, um die Illusion von Reliefstrukturen auf einer Fläche zu erzeugen. Es ermöglicht es, eine einzelne Textur auf eine Oberfläche zu projizieren und die Illusion von Tiefe und Schatten zu erzeugen.

- For each vertex
  - Transform **v** to world space
  - Compute tangent space to world space transform (**t**, **b**, **n**)
- For each fragment
  - Interpolate and renormalize **v**
  - Interpolate frame (**t**, **b**, **n**)
  - Lookup **n'** = texture(u,v)
  - Transform **n'** from tangent space to world space
  - Compute reflection vector **r** (in world space) using **n'**
  - Lookup **c** = cubemap(**r**)
- Note: this is an example of dependent texturing

Es funktioniert indem man eine Umgebungskarte (Environment Map) verwendet, die ein Abbild der Umgebung um das Objekt herum enthält und diese Karte wird auf die Oberfläche des Objekts projiziert. Dann wird eine Bump Map verwendet, die Höheninformationen enthält. Diese Höheninformationen werden verwendet, um die Normale der Oberfläche zu verändern und somit die Illusion von Reliefstrukturen zu erzeugen.

Bump Mapping Issues

- Artifacts
  - Shadows
  - Silhouettes edgy
  - No parallax



Bump mapped Real geometry

- Effectiveness
  - No effect if neither light nor object moves
  - In this case, use light maps
  - Exception: specular highlights

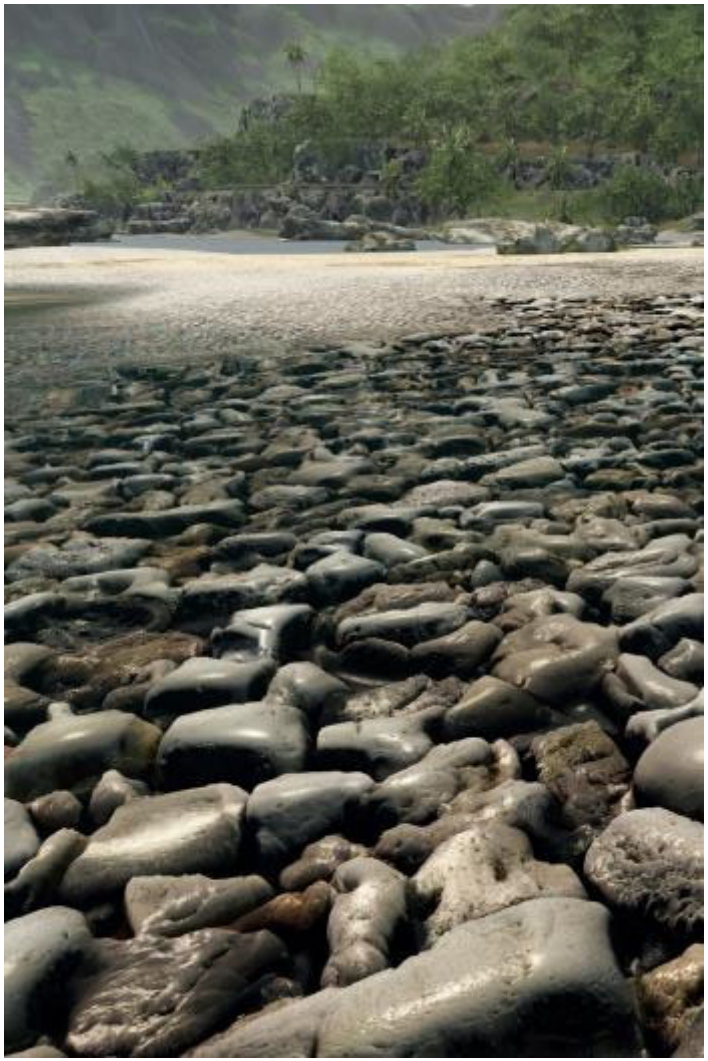
Als Parallaxe) bezeichnet man die scheinbare Änderung der Position eines Objektes durch verschiedene Positionen des Beobachters.

Parallax Mapping

- Also called *Relief Mapping*
- Enhanced bump mapping
  - Texture lookup is displaced according to viewing angle

Parallax Mapping ist eine Technik in der Computergrafik, die verwendet wird, um Tiefeneffekte auf 2D-Texturen zu erzeugen. Es ermöglicht es, die Illusion von Tiefe und Volumen in 2D-Oberflächen zu erzeugen, indem es Schichten von Texturen verwendet, die auf verschiedenen Ebenen angeordnet sind. Dies ermöglicht es, die Tiefenillusion zu erzeugen, ohne dass eine 3D-Modellierung erforderlich ist. Es wird häufig in Spielen und anderen Anwendungen verwendet, um die visuelle Qualität zu verbessern.

Example: Crysis



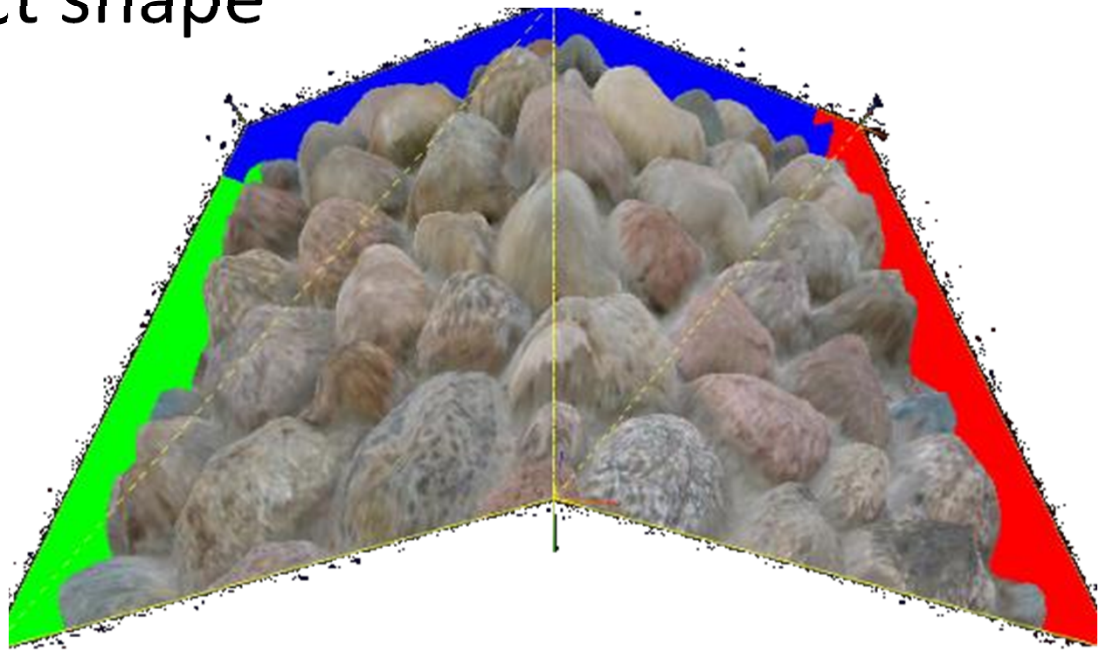
Parallax Mapping Example

- Allows for more realism with motion parallax and steeper height maps with occlusions

https://commons.wikimedia.org/w/index.php?title=File%3AParallax_mapping.ogv https://youtu.be/6PpWqUqeqeQ

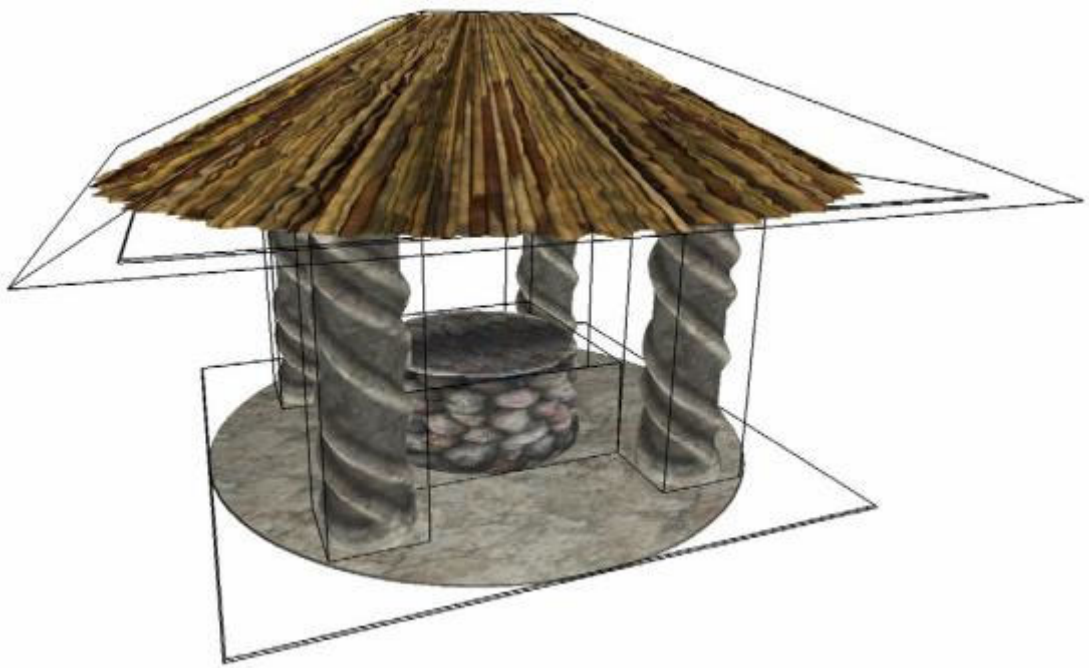
Parallax Mapping Procedure

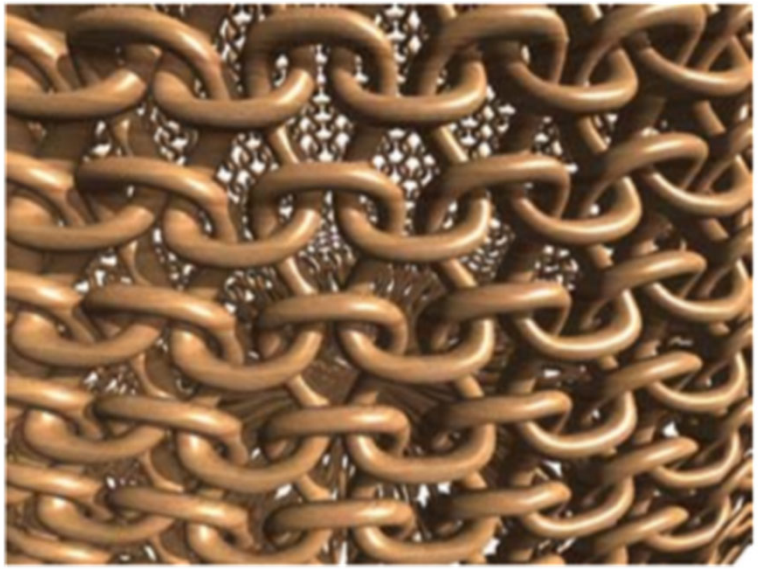
- Parallax occlusion mapping uses raycasting in the fragment shader
- Expensive, but silhouttes and shadows have correct shape



Parallax Mapping Discussion

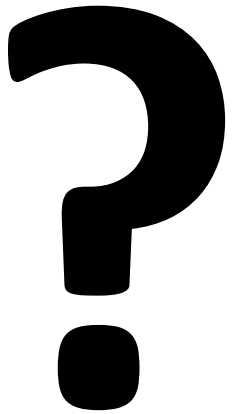
- Can replace almost all geometry with good relief maps
- Result: a raycasting engine





Thank You!

Questions?



Diligent Engine Texture Example

https://github.com/DiligentGraphics/DiligentSamples/tree/master/Tutorials/Tutorial03_Texturing

Vertex Shader

```
cbuffer Constants { float4x4 g_WorldViewProj;
};
struct VSInput { float3 Pos : ATTRIB0;
                 float2 UV  : ATTRIB1; // texture coordinates in
};
struct PSInput { float4 Pos : SV_POSITION;
                 float2 UV  : TEX_COORD; // texture coordinates out
};
void main(in  VSInput VSIn, out PSInput PSIn) {
        PSIn.Pos = mul(float4(VSIn.Pos,1.0), g_WorldViewProj);
        PSIn.UV  = VSIn.UV; // just copy the input
}
```

Fragment Shader

```
Texture2D g_Tex; // texture object
SamplerState g_sampler; // sampler object
struct PSInput { float4 Pos : SV_POSITION;
                 float2 UV : TEX_COORD;
};
struct PSOutput { float4 Color : SV_TARGET; };
void main(in  PSInput  PSIn, out PSOutput PSOut) {
   PSOut.Color = g_Tex.Sample(g_sampler, PSIn.UV);
}
```

Initializing the Pipeline State

- We need to define mutable resource variable g_texture

```
ShaderResourceVariableDesc Vars[] = {
    {SHADER_TYPE_PIXEL, "g_Texture", SHADER_RESOURCE_VARIABLE_TYPE_MUTABLE}
};
```

- Pass it to pipeline state object

```
PSOCreateInfo.PSODesc.ResourceLayout.Variables    = Vars;
PSOCreateInfo.PSODesc.ResourceLayout.NumVariables = _countof(Vars);
```

- Define an immutable sampler (sampling does not change)

```
SamplerDesc SamLinearClampDesc {
    FILTER_TYPE_LINEAR, FILTER_TYPE_LINEAR, FILTER_TYPE_LINEAR,
    TEXTURE_ADDRESS_CLAMP, TEXTURE_ADDRESS_CLAMP, TEXTURE_ADDRESS_CLAMP
};
ImmutableSamplerDesc ImtblSamplers[] = {
    {SHADER_TYPE_PIXEL, "g_Texture", SamLinearClampDesc}
};
PSOCreateInfo.PSODesc.ResourceLayout.ImmutableSamplers    = ImtblSamplers;
PSOCreateInfo.PSODesc.ResourceLayout.NumImmutableSamplers=_countof(ImtblSamplers);
```

Load Texture and Bind It

- Diligent loads jpg, png, tiff

```
TextureLoadInfo loadInfo;
loadInfo.IsSRGB = true;
RefCntAutoPtr<ITexture> Tex;
CreateTextureFromFile("DGLogo.png", loadInfo, m_pDevice, &Tex)
```

- Get shader resource view from the texture

```
m_TextureSRV = Tex->GetDefaultView(TEXTURE_VIEW_SHADER_RESOURCE);
```

- Create shader resource binding object from PSO

```
m_pPSO->CreateShaderResourceBinding(&m_SRB, true);
```

- Set shader resource view

```
m_SRB->GetVariableByName(SHADER_TYPE_PIXEL, "g_Texture")->Set(m_TextureSRV);
```

Render Textured Object

- … (set vertex buffer etc. omitted here)
- Select our prepared pipeline

```
m_pImmediateContext->SetPipelineState(m_pPSO);
```

- Commit the shader resources

```
m_pImmediateContext->CommitShaderResources(m_SRB, RESOURCE_STATE_TRANSITION_MODE_TRANSITION);
```

- Draw Call (here: indexed draw)

```
DrawIndexedAttribs DrawAttrs = {...}; // This is an indexed draw
m_pImmediateContext->DrawIndexed(DrawAttrs);
```