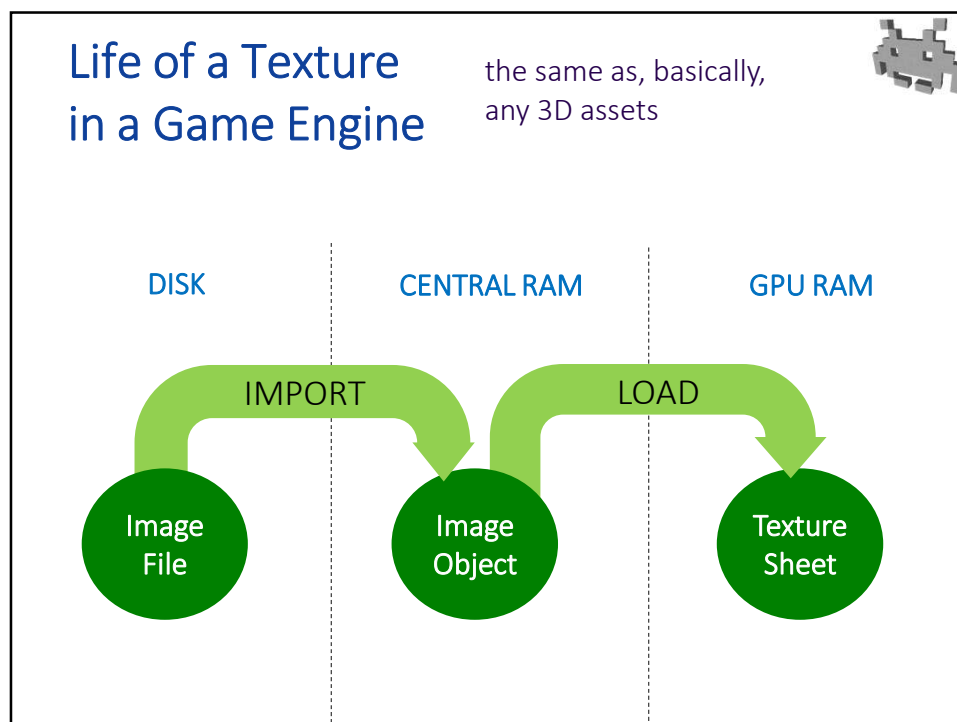
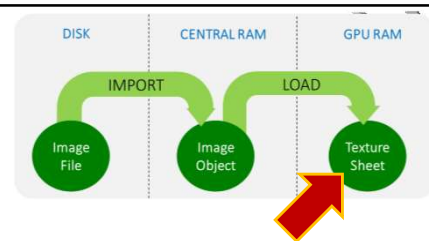


34



35

Texture Sheets (in GPU RAM)



- Rasterized images, but with peculiarities ...
 - MIP-map levels
 - channels per texel: 1,2,3, or (most typically) 4
 - bits per channels:
 - usually 8, fixed point
 - floating textures supported
 - compression: specific texture schemas (see next)
 - resolution: powers of 2 per side

36

Per-fragment Texture fetch (during rendering, hardwired in GPU)

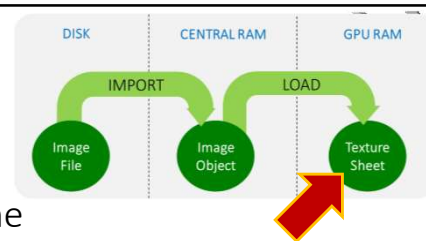


- **Hard-wired GPU** mechanisms to access the texture image at a given location: $(u, v) \rightarrow \mathbb{R}^4$
- Includes many steps:
 1. Management of out-of-bound coordinates.
E.g., repeat mode: $u \leftarrow [u]$ and $v \leftarrow [v]$
 2. De-normalization of coords, from normalized $[0..1]^2$ to texel coord $[0..Res_x] \times [0..Res_y]$
 3. Selection of the appropriate MIP-map level (how?)
 4. On-the-fly decompression of compressed image data
 5. Bilinear interpolation between 4 texels,
plus linear across MIP-map levels

number of
channels

37

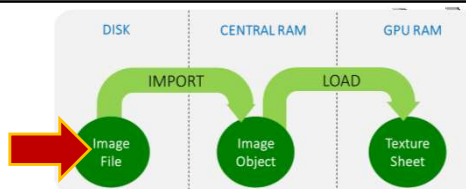
Texture compression (to save GPU RAM)



- Save RAM, but preserve the **random-accessibility** of texels
 - color quantization
 - e.g., 5 red 5 green 5 blue 1 alpha = 16 bits per texel
 - color-table, or “palette”
 - e.g., 256 color table for texture, an 8-bit index per texel
 - specialized image-compression schemas. They are:
 - Lossy (very much so)
 - Fixed compression rates (e.g. ¼)
 - Unfavourable compression/loss ratio ☹
 - Most diffuse scheme S3TC, with variants: DXT-1
 - yes/no alphas → uniform alphas (-2 -3) smooth alphas (-4 -5)

38

Textures as assets: file formats



For generic images

(decompress the entire image before accessing any pixels)

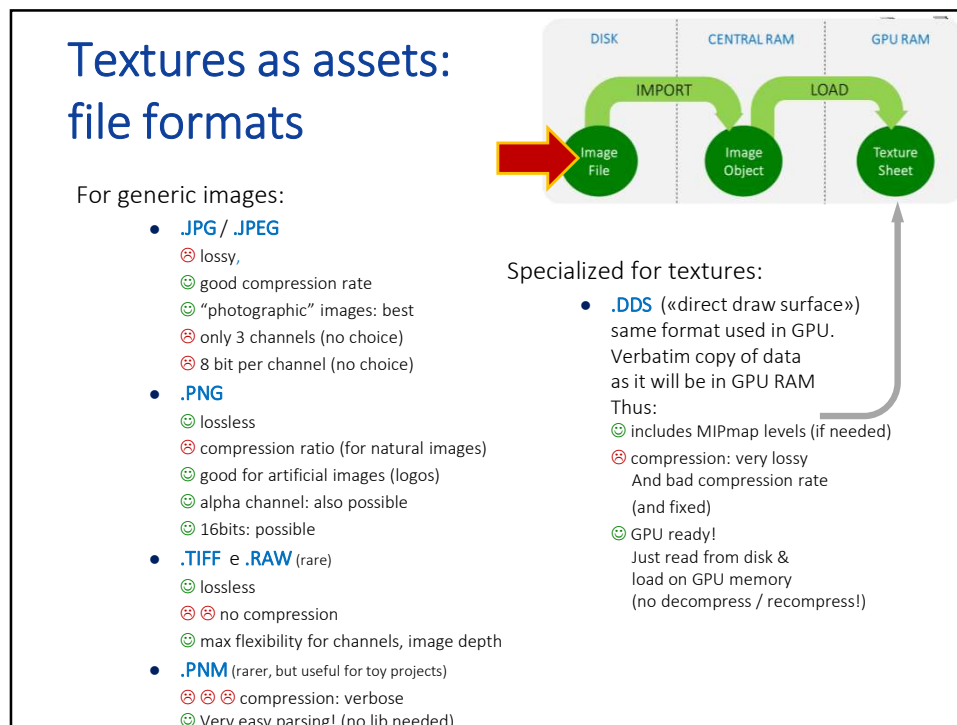
- 😊 compression: excellent
- ☹ loading: heavy:
 - Decompress from RAM, (maybe) recompress in GPU-RAM
- ☹ MIP-map levels:
 - Procedurally generated.
 - Control by the engine
- 😊 Resolution: any (can pad on load)

For textures

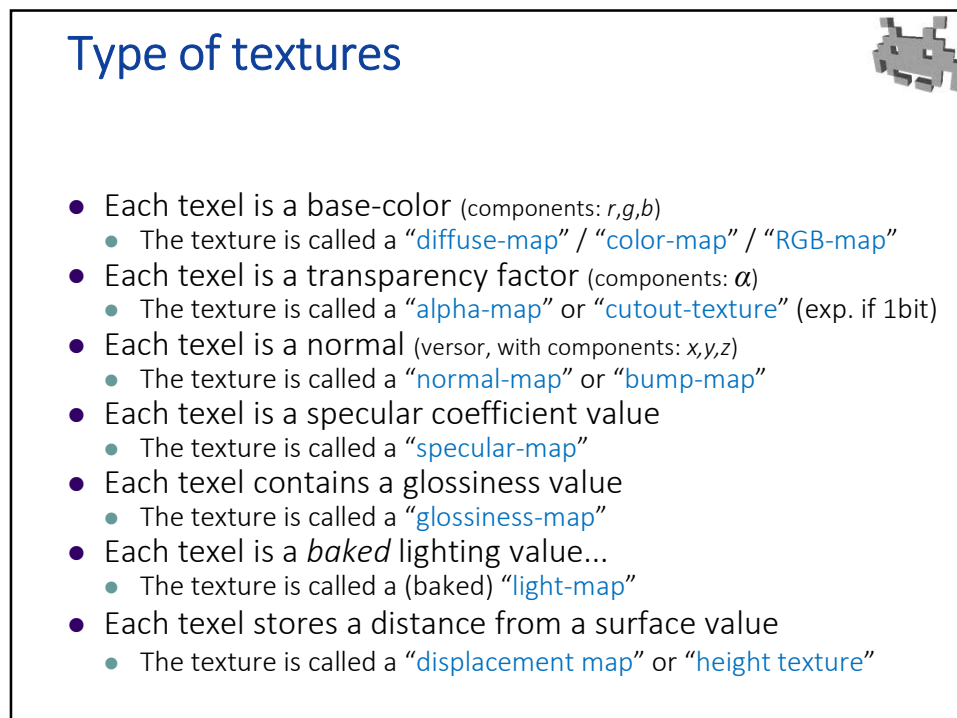
(random accessibility to texels, without uncompressing the entire image)

- ☹ compression: bad
- 😊 loading: light
 - direct streaming possible
Disc => RAM => GPU RAM
- 😊 MIP-map levels:
 - Baked.
 - Control by the artist
- ☹ Resolution:
 - must be a pow of 2

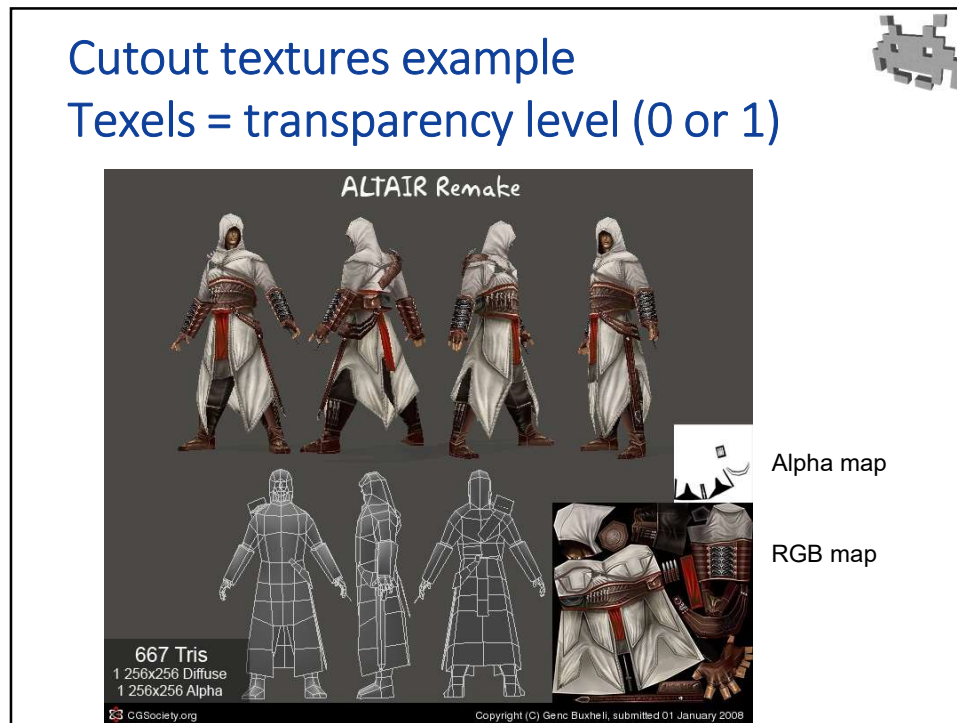
39



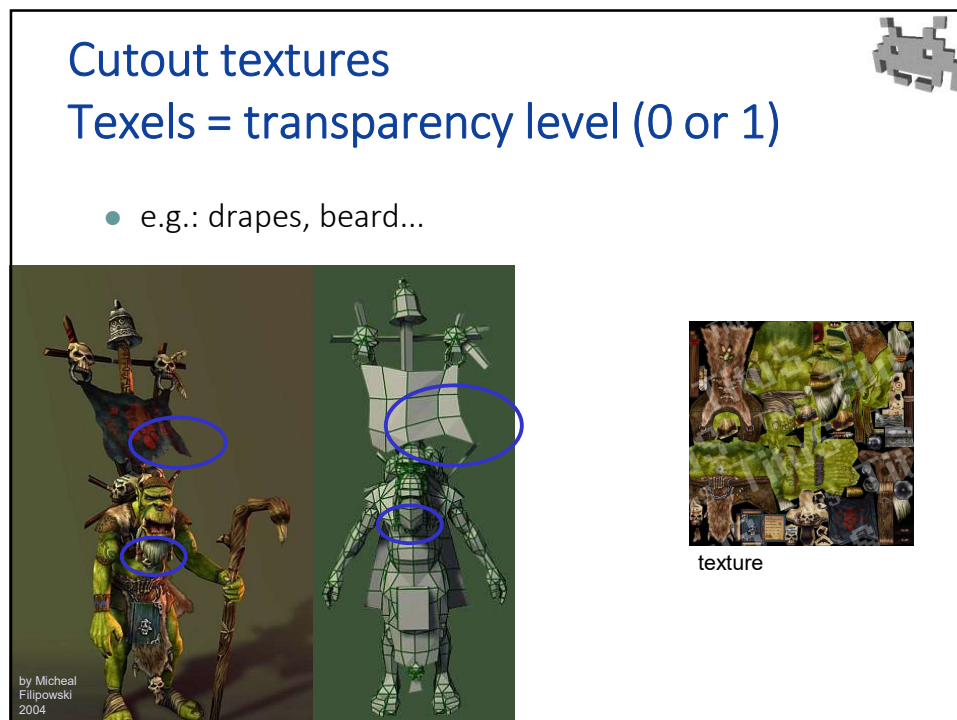
40



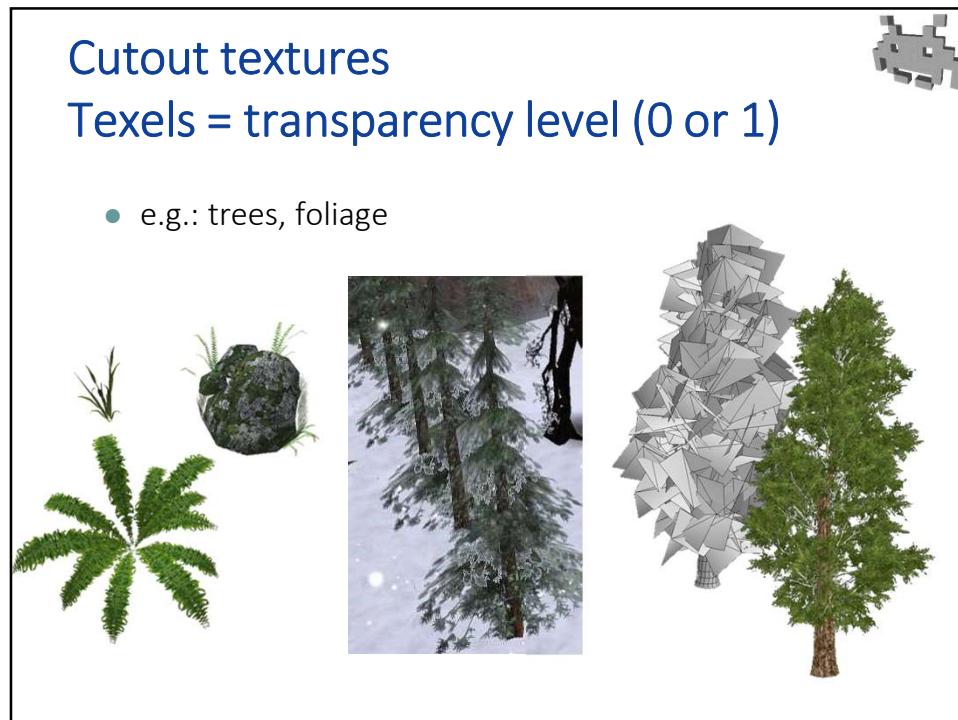
41



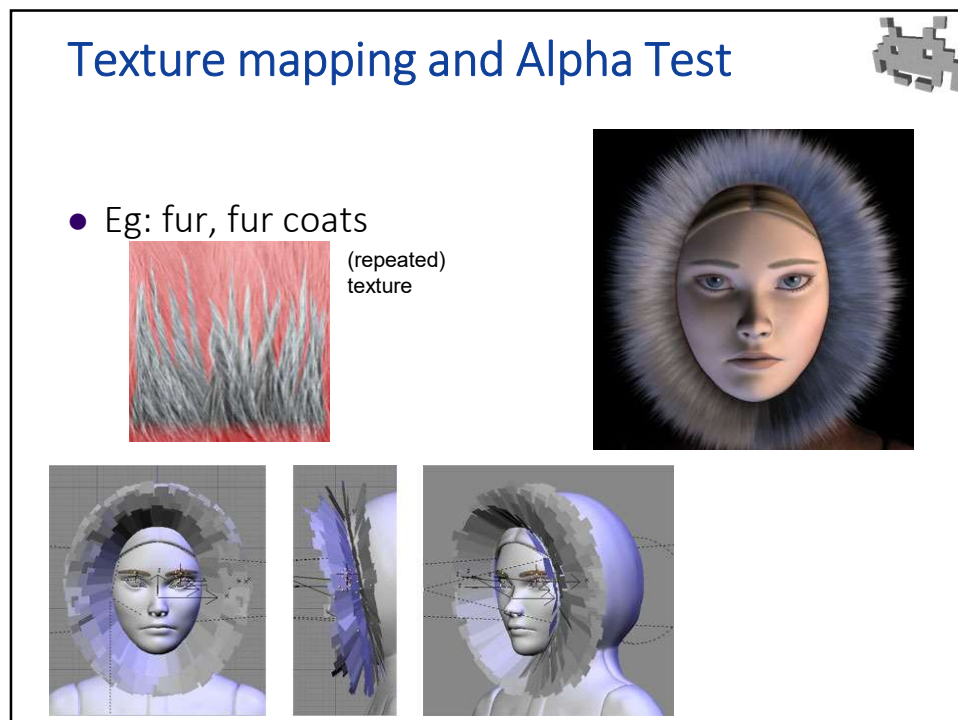
42



43



44



45

Bump-Map (*)

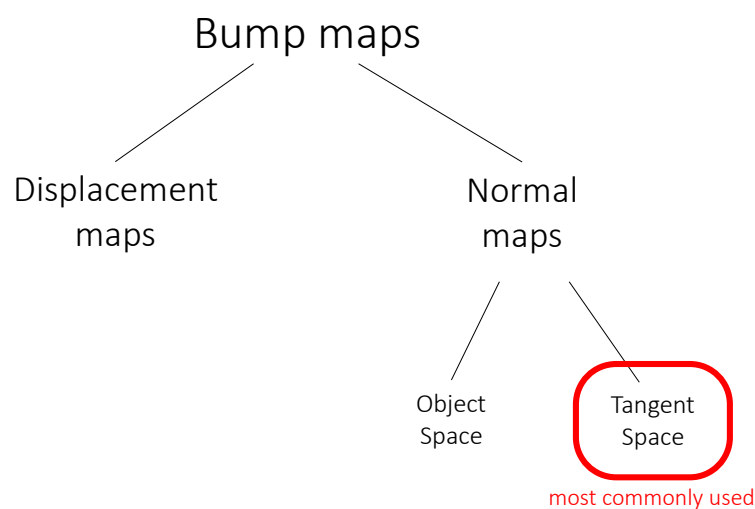


- a **texture** modelling (or, providing an illusion of **shape details** (i.e., high-frequency geometric features))
 - details not modeled by the “real” geometry (the mesh)
 - remember: meshes tend to be low-poly
 - not much detail in them
 - approach also known as “**Texture-for-Geometry**”
 - rationale: texels are cheaper to render/store than vertices!
 - geometric details may extrude **out** or be engraved **in** the “real” (mesh) surface
 - in many cases: the detail affects lighting only
 - sufficient to trick the eye
 - especially true with dynamic lighting

(*) This terminology not universal: «Bump-map» can mean just «displacement map»

46

Types of Bump maps



47

Types of Bump maps



- **Bump map:**
 - A texture encoding hi-frequency details
- **Displacement Map:**
 - Details are encoded by storing differences between mesh geometry and detailed surface:
 - as **scalars** (distance along the normal), or as **vectors**
 - used for: on-the-fly re-tessellation, and *parallax mapping* technique
- **Normal Map:**
 - Details are encoded by storing the normals of the detailed surface
 - used for: affecting the lighting
 - In which frame?
 - In **Object Space**: (requires 1:1 UV-map)
 - In **Tangent Space**: (**TBN** space)
 - Usable on more surfaces independently from the orientation
 - Requires Tangent-Bitangent direction and normals on surface

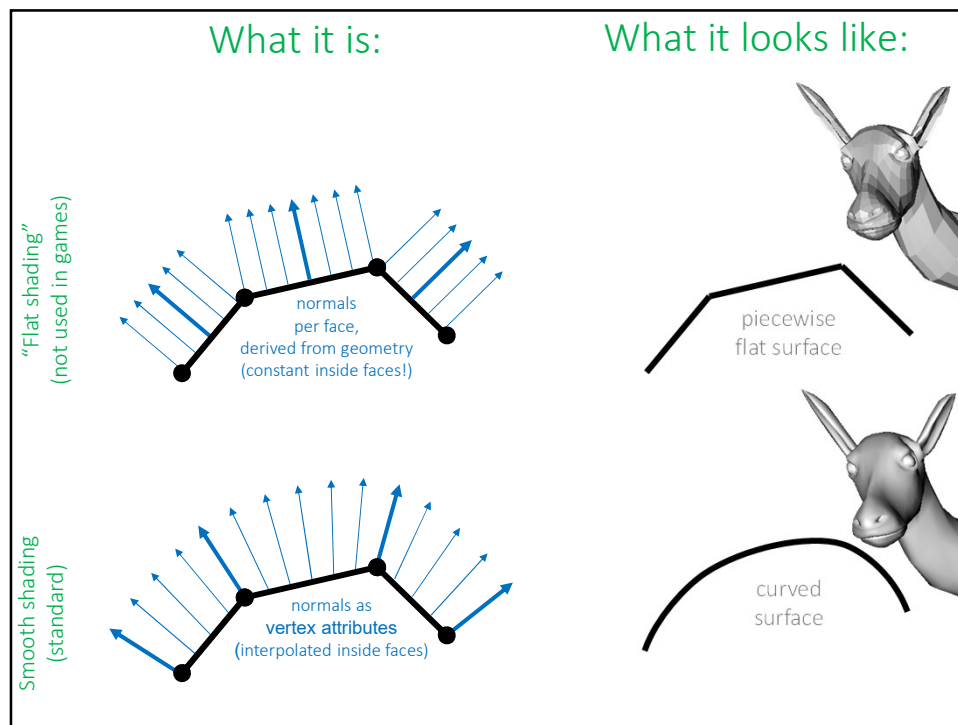
49

Bump-Map: from the modeler perspective

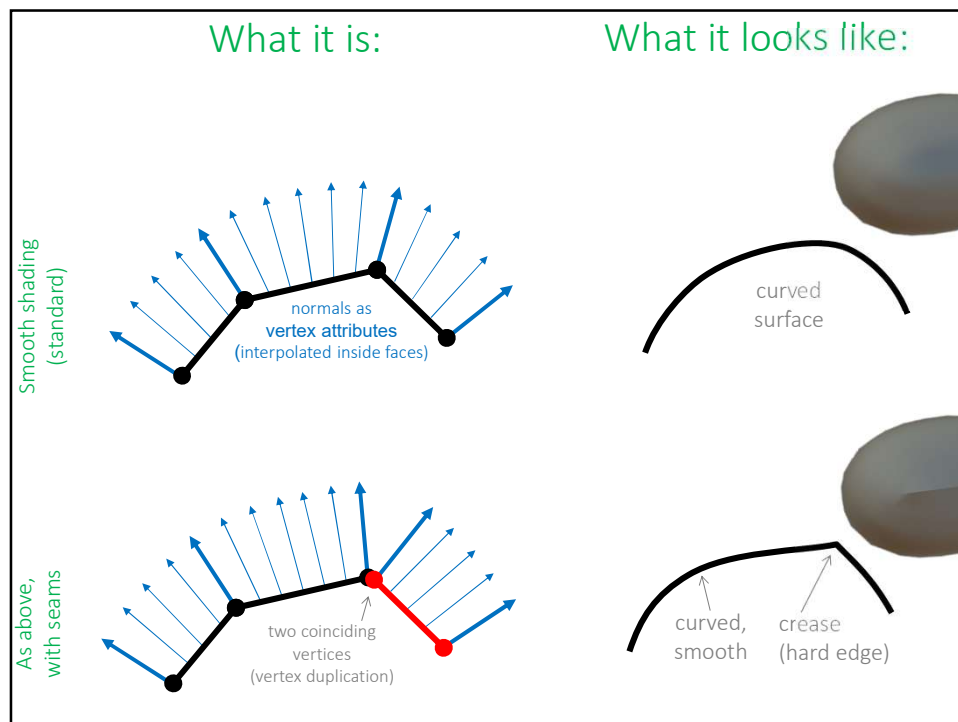


- **macro-structure** of the object → **low-poly mesh**
 - e.g.: the general shape of the horse
 - e.g.: the general shape of the face
 - e.g.: the general shape of the dragon
- **meso-structure** of the object → **bump-map**
 - e.g.: the musculature of the horse
 - e.g.: the wrinkles of the face
 - e.g.: the flakes of the dragon
- **micro-structure** of the object → **material parameters**
 - e.g.: the velvet-like fur of the horse
 - e.g.: the structure of the dermis / sebum
 - e.g.: the micro roughness / smoothness of the flakes

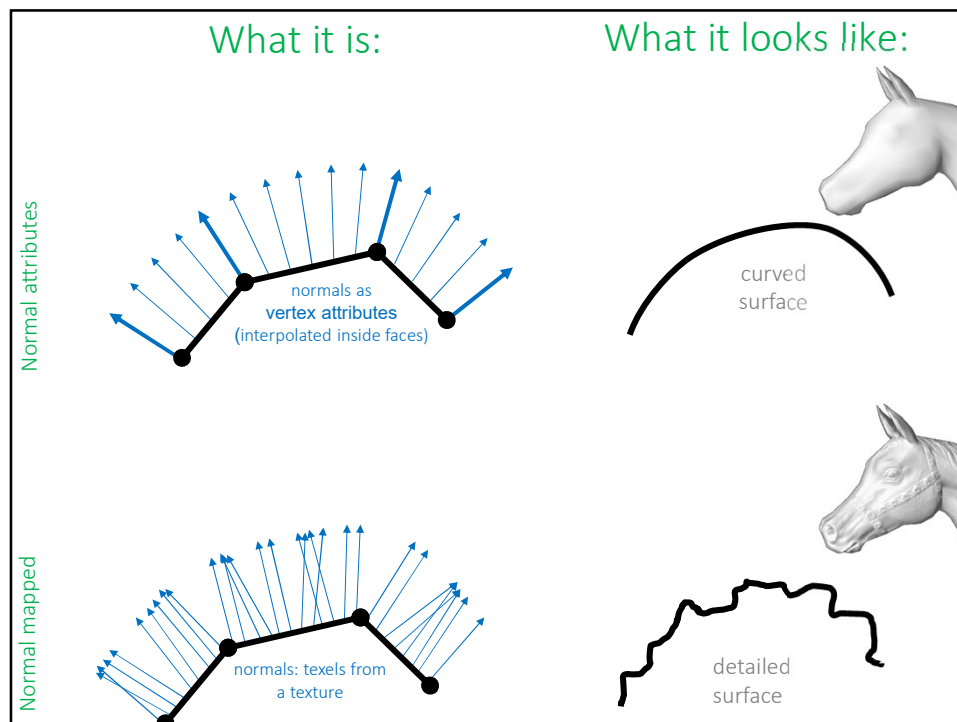
50



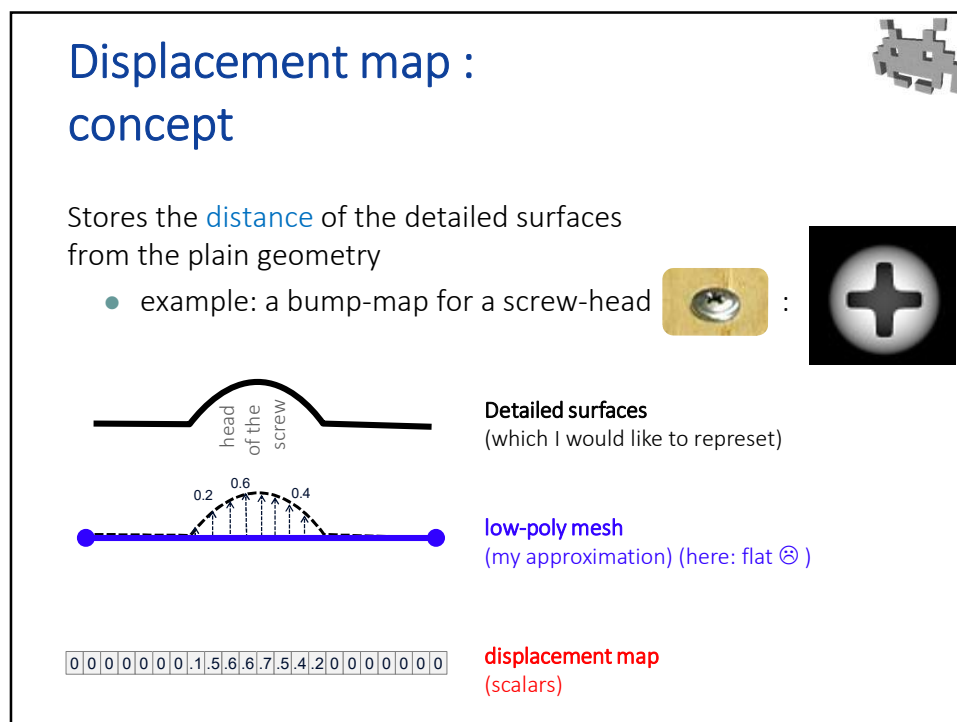
53



54



55



56

Displacement map: notes

- Each texel stores: a **distance** of the detailed surface
 - Along the **normal** direction (of low-poly mesh)
 - 1 **scalar** per texel → 1 channel texture
- Which way:
 - outwards (*extrusions*)
 - inwards (*excavations*)
 - or both (signed displacements)
- Storage:
 - **gray-scale** image (1 scalar per pixel)
 - remap values within the interval [0..1]
 - global scale factor (on the fly)
- Possible uses:
 - Direct lighting of implied normals: “embossing” effect (old effect: it’s a bad approximation, not common anymore)
 - Global illumination (ambient occlusion) See later
 - «Parallax mapping» technique See later
 - Intermediate data for the construction of a normal map See later



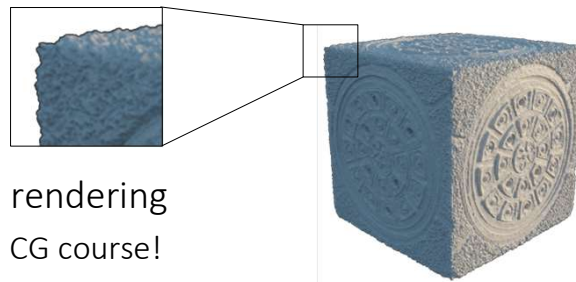
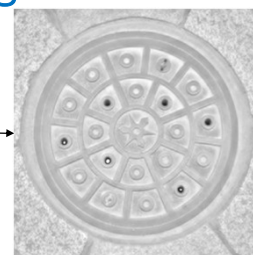
white = outwards
black = flat

Easy to paint and
manipulate!

57

(scalar) Displacement map: Rendering – parallax mapping

- Technique used render a mesh with a Displacement Map
 - Bonus: the silhouette of the object can be affected



- See lecture on rendering
 - And Real time CG course!

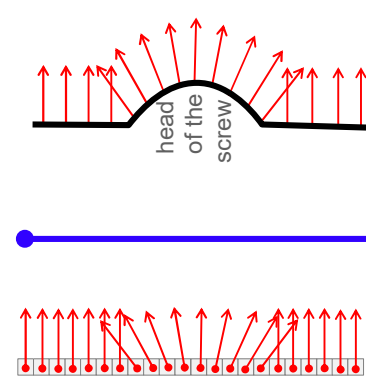
Image courtesy of <https://cgcookie.com/articles/normal-vs-displacement-mapping-why-games-use-normals>

60

Normal Map: concept

Store the **Normals** of the detailed surfaces

- example -- a normal-map for a screw-head



Detailed surface
(I would like to model)

low-poly mesh
(approximation of ^) (here: flat ☹)

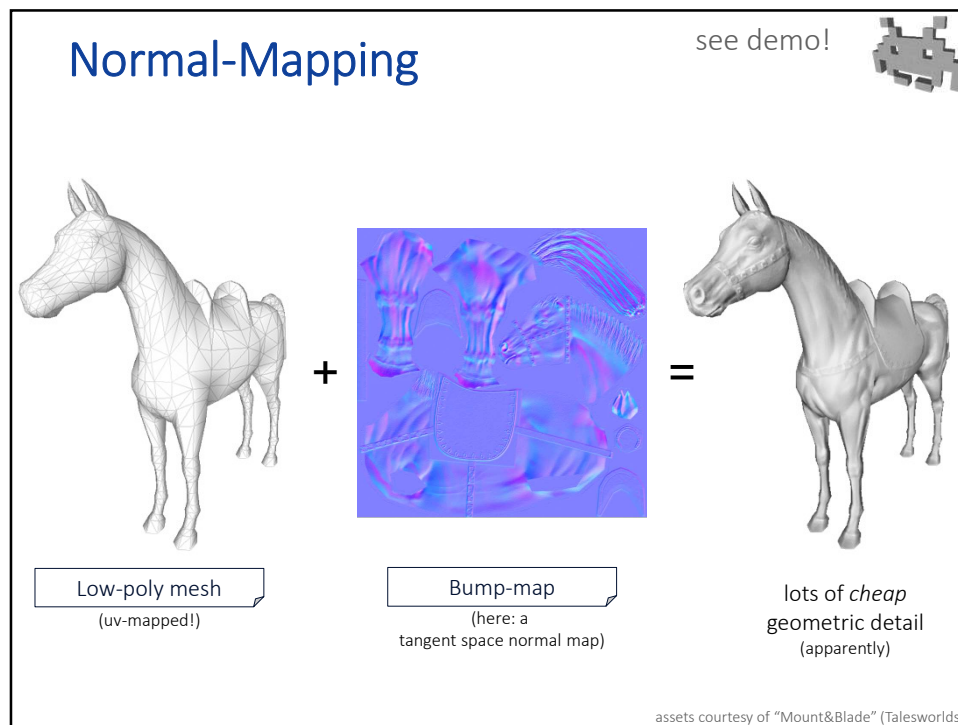
normal map
(one normal per texel)

61

Normal Map: notes

- Affects the lighting only
 - not** the parallax
 - not** the shape of the object
 - The lighting reflects the hi-freq detail of the object
 - dynamically (with variable lights!)
 - Total illusion: very convenient
 - If we are not trying to model a macro-structure
- In rendering: use the normal from the texture
 - (for lighting)
 - Instead of the interpolated per vertex normal
- Normals are expressed in cartesian coord
 - Often
 - But not always (\exists better ways to express unit vectors!)
 - Question: ok, but in which space??? *more later*

62



63

Normal Maps: in which space are the normals encoded?

i.e., texture normals and mesh vertices are expressed in the same space

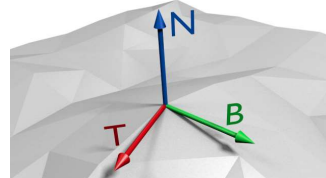
- Object space: **Object-Space Normal-Maps**
 - ☺ the per-vertex normal becomes unnecessary!
 - The normal from texture substitute it
 - ☺ Trivial to apply (during rendering)
 - just use the normal fetched from the texture for lighting
 - ☹ normal-map is bound to a specific object
 - cannot be reused for different objects
 - ☹ Each region of the normal map is bounded to one specific area region of the object!
 - Injective UV-maps only!
 - e.g. no tiling, no exploitation of symmetries

65

Tangent space (aka TBN space)

- A vector space defined \forall point of the surface:

- Z axis: **Normal**
 - orthogonal to surface
- X and Y axis: tangent vectors
 - parallel to the surface
 - X = **Tangent**
 - Y = "**Bi-Tangent**"
(sometimes, but inappropriately: *Bi-Normal)

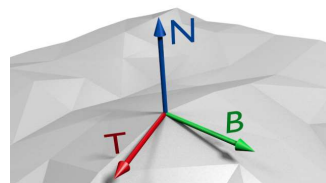


66

Tangent space (aka TBN space)

- How to store them?

- As 3 vectors stored as (per-vertex) **attributes**
 - So, they are interpolated inside faces (like any other attribute)
- Optimizations are possible!
 - Not necessarily stored as 3 vectors (9 scalars)
 - E.g.: instead of storing B, we store N and T, then $B = N \times T$
- Note: they have discontinuities
 - seams (vertex duplications) are necessary
 - In first approximation, the same ones required by the UV-map (but non only! why?)

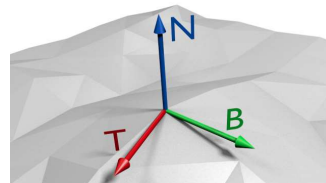


67

Tangent space (aka TBN space)

- How to compute them?

- Normal
 - as usual (see lecture on mesh)
- Tangent & Bi-Tangent
 - determined by the UV-map!
 - T = gradient of U coordinate
 - B = gradient of V coordinate



- details:

- All three are defined and constant inside faces, then averaged at vertices (see per-vertex normal computation)
- T,B,N can be *only approximatively* orthogonal to each other
- T,B,N reference frame can be left-handed or right-handed (even different “handedness” in different parts of the same mesh)

68

Normal Maps: in which space are the normals encoded?

- Tangent space: Tangent Space Normal-Maps
(the standard «bump-map», in games)

- ☹ extra attributes are now needed per vertex:

The
tangent
space

- Normal direction
- Tangent direction
- Bitangent direction

← basically, a TS normal map specifies how to **modify** the per-vertex normal instead of **replacing** it

- 😊 normal-map can be shared by different objects
- 😊 non injective UV-maps can be used
 - e.g., the normal-map can be tiled
 - e.g., symmetries can be exploited
- 😊 normal-map is independent from the mesh
 - e.g. can be constructed without knowing the mesh

69

Normal-map: storage

DISK CENTRAL RAM GPU RAM

IMPORT LOAD

Image File Image Object Texture Sheet on GPU

- Idea: store it as an RGB texture
 - $R \leftrightarrow X$
 - $G \leftrightarrow Y$
 - $B \leftrightarrow Z$
- but $X, Y, Z \in [-1, +1]$ and $R, G, B \in [0, +1]$
thus a linear mapping is needed:

$X \in [-1, +1]$

$\ni R$

$R \in [0, 1.0]$

$$R = \frac{1}{2} (X + 1)$$

$$X = 2R - 1$$
- Advantage: reuse compression of RGB textures/images
- Extra: store a (scalar) displacement map in 4th texture channel
- But, note: other, more efficient representations of versors exists

70

Normal-maps: Storage

DISK CENTRAL RAM GPU RAM

IMPORT LOAD

Image File Image Object Texture Sheet on GPU

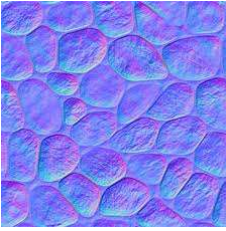
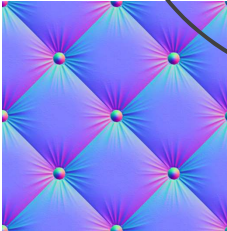
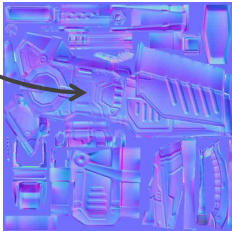
- Examples of tangent space normal-maps

Prevailing normal : $X \approx 0$, $Y \approx 0$, $Z \approx 1$

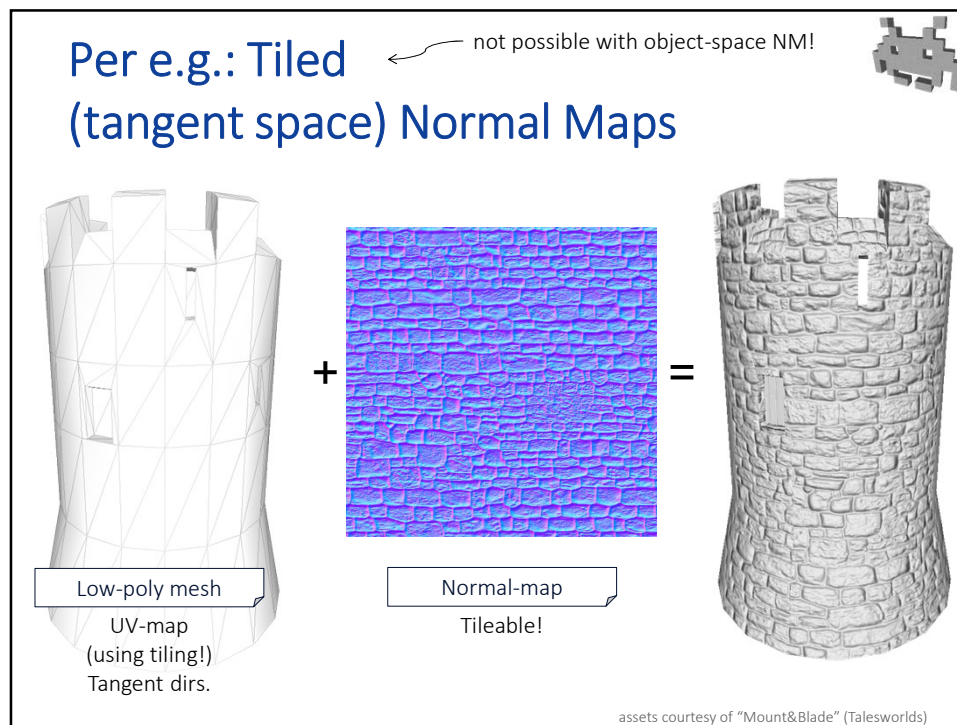
\Rightarrow

Prevailing color: $R \approx 0.5$, $G \approx 0.5$, $B \approx 1$

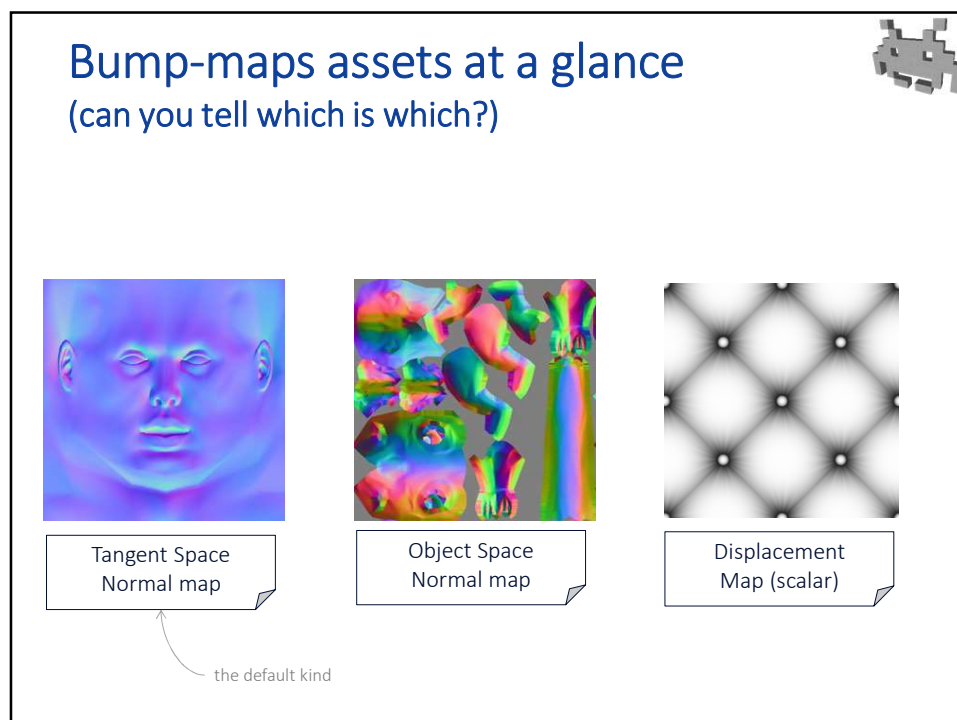
(~light blue)

71



72



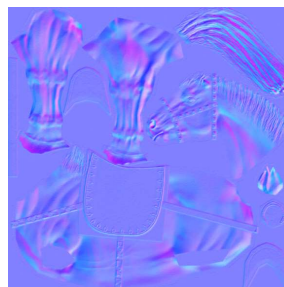
73

Observe



Object Space
Normal map

1:1 UV-map
right leg != left leg



(Tangent Space)
Normal map

UV-map NOT injective
Exploited symmetries!
Left side of head = right side of head

74

Normal map comparison (summary)

Object Space Normal map:	Tangent Space Normal map:
Replaces the normals of the object	Modifies the normals of the object
No normal attribute required on the mesh any more	Requires two extra attributes on the mesh: T and B versors (in addition to the normal)
Constructing the texture requires to know the mesh it will be applied to	Textures can be constructed independently from the mesh (just like a color map!)
E.g., a normal map cannot be constructed from a displacement map (w/o the mesh)	E.g., a normal map can be constructed from a displacement map
It's difficult to share a normal map between models	Normal maps can be shared between different models
" unwrapping " UV-maps required (except a few lucky cases)	Can be applied to non-injective UV-maps
E.g., no tiled textures. E.g., no symmetry exploitation	E.g., tiled textures ok, E.g., symmetry exploitation ok
E.g., east-wall and south-wall of a castle: different normal maps required	E.g., east wall and south wall of a castle: same normal map.
Looks colorful (if encoded as RGB)	Looks azure-ish (if encoded as RGB)

MUCH MORE USED IN GAMES

75

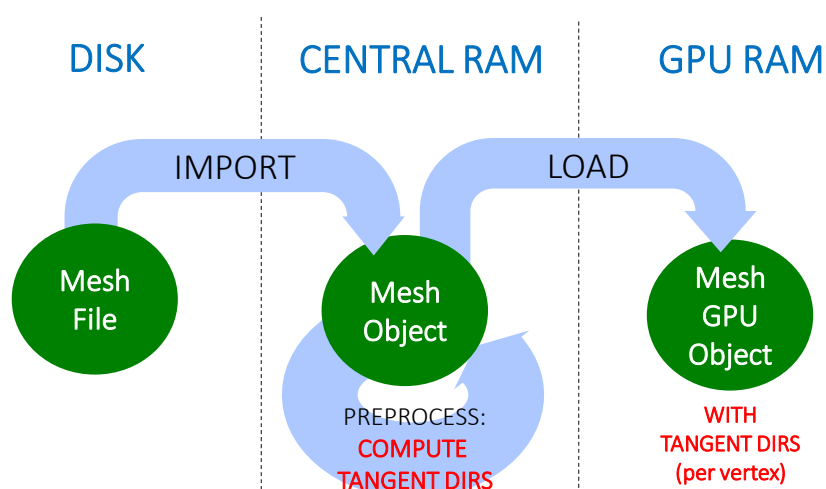
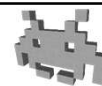
How to extract T and B vectors from the UV-map



- Concept (a mental experiment)
 - STEP 1: color a texture with a grid
 - horizontal blue lines = U direction
 - vertical red lines = V direction
 - STEP 2: apply it to the Mesh!
 - STEP 3: look at it:
 - the T vectors are the Blue lines directions
 - the B vectors are the Red lines directions
- T and B directions are defined in a triangular face
 - then, they are averaged at vertices
 - (just like the normal directions!)

76

Tangent Dirs (Tangent and Bitangent) as per vertex attributes



77

Extracting T and B vectors from the UV-map (in a triangle)

- Object Space (3D)
- Texture Space (2D)

Idea:

\vec{u} is some linear combination of \vec{t}_1 and $\vec{t}_2 \Rightarrow \vec{T}$ is the same linear combination of \vec{e}_1 and \vec{e}_2

\vec{v} is some linear combination of \vec{t}_1 and $\vec{t}_2 \Rightarrow \vec{B}$ is the same linear combination of \vec{e}_1 and \vec{e}_2

78

Extracting T and B vectors from the UV-map (in a triangle)

- Input: 3D vertices $\mathbf{p}_{0,1,2}$ and 2D vertices $\mathbf{q}_{0,1,2}$
- Find 3D edge vectors $\vec{e}_{1,2}$ and 2D edge vectors $\vec{t}_{1,2}$
- Find scalars a, b and c, d such that...

$$a \vec{t}_1 + b \vec{t}_2 = \vec{u} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad c \vec{t}_1 + d \vec{t}_2 = \vec{v} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

- Then

$$\vec{T} = a \vec{e}_1 + b \vec{e}_2 \quad \vec{B} = c \vec{e}_1 + d \vec{e}_2$$

79

Extracting T and B vectors from the UV-map (in a triangle)



- Input: 3D vertices $\mathbf{p}_{0,1,2}$ and 2D vertices $\mathbf{q}_{0,1,2}$
- Find $\vec{\mathbf{e}}_1 = \mathbf{p}_1 - \mathbf{p}_0$ $\vec{\mathbf{t}}_1 = \mathbf{q}_1 - \mathbf{q}_0$
 $\vec{\mathbf{e}}_2 = \mathbf{p}_2 - \mathbf{p}_0$ $\vec{\mathbf{t}}_2 = \mathbf{q}_2 - \mathbf{q}_0$
- Find scalars a, b and c, d such that...

in matrix form:

solve with a 2x2 matrix inversion

$$\begin{bmatrix} \vec{\mathbf{t}}_1 & \vec{\mathbf{t}}_2 \end{bmatrix} \begin{bmatrix} a & c \\ b & d \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} a & c \\ b & d \end{bmatrix} = \begin{bmatrix} \vec{\mathbf{t}}_1 & \vec{\mathbf{t}}_2 \end{bmatrix}^{-1}$$

- Then

$$\vec{\mathbf{T}} = a \vec{\mathbf{e}}_1 + b \vec{\mathbf{e}}_2 \quad \vec{\mathbf{B}} = c \vec{\mathbf{e}}_1 + d \vec{\mathbf{e}}_2$$

80

RGB maps: How are they obtained?



- Image *first, then* UV-map
 - e.g., images that are photos
 - e.g., tileable images
- UV-map *first, then* paint 2D
 - paint with 2D app (e.g. photoshop)
- UV-map *first, then* paint 3D
 - paint within 3D modelling software,
 - or: 1. export 2D rendering,
2. paint over with e.g. photoshop,
3. reimport images
4. goto 1



UV-mapper



UV-mapper



2D painter



UV-mapper



3D painter

82

RGB maps: How are they obtained?



...or:

- *first* paint 3D
 - on hi-res model,
 - “paint” on vertex attributes
 - e.g. with Z brush...
- *then* coarsen
 - build / autobuild final low-poly version
- *then* UV-map
 - the low-poly model
 - must be a 1:1 UV-map!
- *then* texture backing
 - auto build texture

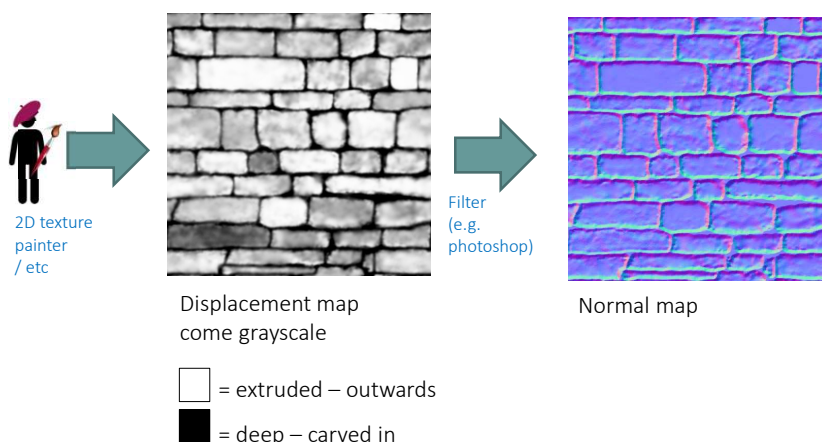
*more
about
this later...*

83

How are normal-maps obtained? (1/5) from a displacement map



see demo!



84

How are normal-maps obtained? (1/5) from a displacement map



- Input: a scalar displacement map
Output: a normal map
- Algorithm (2D image processing):
 - \forall texel \mathbf{t} of displacement map, compute **best fitting plane** around \mathbf{t}
 - Consider all 3D points in a 3×3 patch surrounding \mathbf{t}
 - Find plane minimizing the summed squared distance from them
 - It's a least-squares minimization problem
 - The normal of this plane is the normal for \mathbf{t}
- Resulting normal map is expressed in **tangent-space**
 - By definition! (one big advantage of Tangent Space NM)
 - Can be converted into Object-Space if needed (for a given UV-mapped mesh – injective maps only of course)

a texel at coords u, v
corresponds to
a 3D point
 $(u, v, \text{height}[u, v])$

or 5×5 ,
or 7×7 ...

85

How are normal-maps obtained? (2/5) painting on 3D



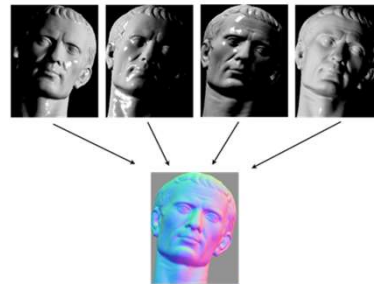
- Direct painting of normal- on the model
 - (can be don, e.g., with Z-brush, Sculpttris Alpha...)
- Similar to a painting of color-maps
 - but artist paints geometric details not colors
- Similar to mesh sculpting too
 - but, for each stroke, the system directly updates the normal on the texture-map, not the geometry on the mesh

86

How are normal-maps obtained? (3/5) captured from reality

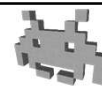


- Captured from reality, using photos
- Example: “**Photometric Stereo**”
 - a form of “inverse lighting”
 - a **computer vision** technique
- Input: n real images
 - Same viewpoint
 - Different illumination
 - possibly, controlled and known
- Output: a Normal Map
 - expressed in image space
 - can be converted in object space, or in tangent space



87

How are normal-maps obtained? (3/5) captured from reality



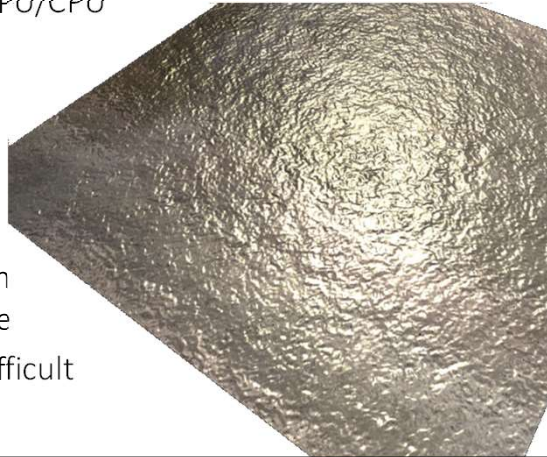
- Normal map estimation from images
 - Traditionally, many pictures are required in input
 - Traditionally, controlled illumination is required (I must place lights in known position)
 - With Machine Learning, it's becoming possible to use a single image with natural illumination
- Idea:
 - input: a photo of a brickwall
 - output: a diffuse map + a normal map + a specular map
- It's an active area of research!

88

How are normal-maps obtained? (4/5) procedural generation (rare)



- Usual considerations about **procedurality**:
 - Saves RAM, costs GPU/CPU
 - Can be baked in preprocessing (becomes an asset)
 - Can be build at run-time
 - Bonus: no repetition artifacts, animatable
 - Problem: control difficult



89

How are normal-maps obtained? (5/5) from a high-resolution model



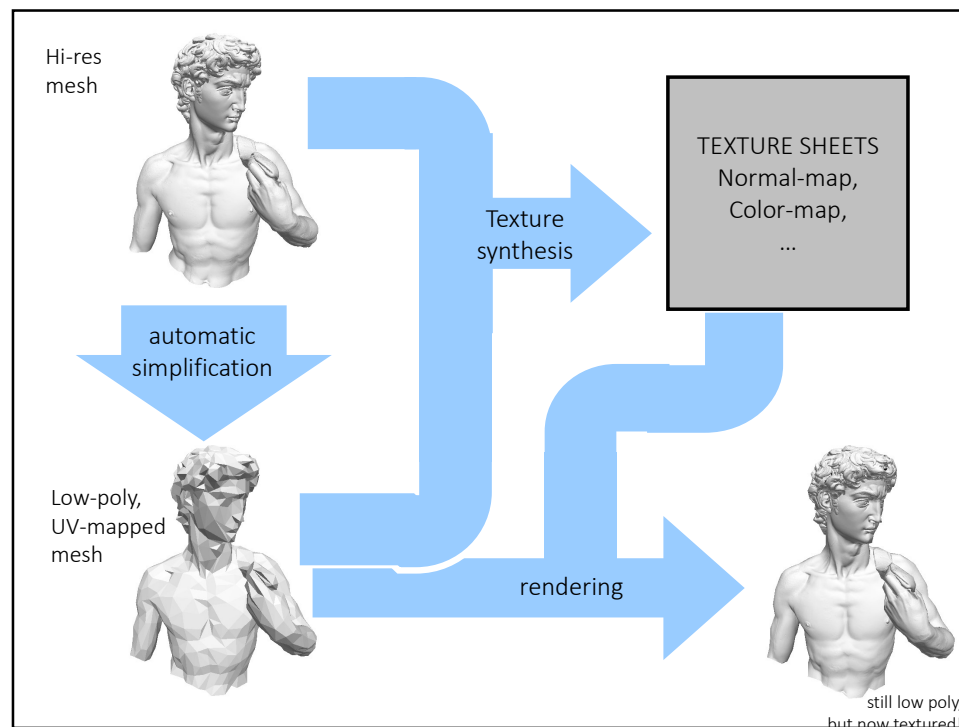
- textures baking / detail recovery / “detail texture” synthesis / texture for geometry
- input:
 - hi-res mesh A with **per-vertex attributes**
 - low-poly mesh B, with an **injective UV-map**
- output:
 - textures for B storing the attributes of A
- a fully automatic process!

90

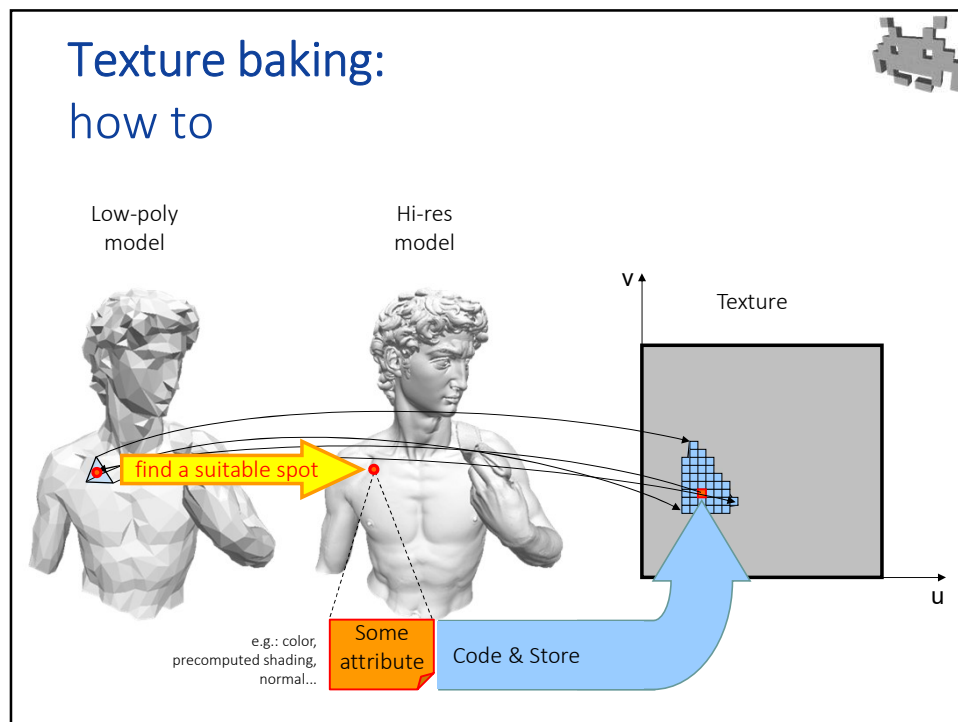
Texture baking: texture synthesis from hi-res models

- input examples:
 - low-poly mesh A obtained from hi-res mesh B via **automatic simplification** or **manual retopology**
 - hi-res mesh B obtained from low-poly mesh A via **sculpting**
 - output examples:
 - attributes = normals
→ an **object-space normal map** is produced
 - attributes = base colors
→ a **diffuse maps** is produced
 - attributes = baked (global) lighting / AO
→ a **light-map** / **AO-map** is produced
 - store distances between A and B (no attribute required)
→ a **displacement map** is produced
- then converted to tangent space (using mesh A)
- common case!

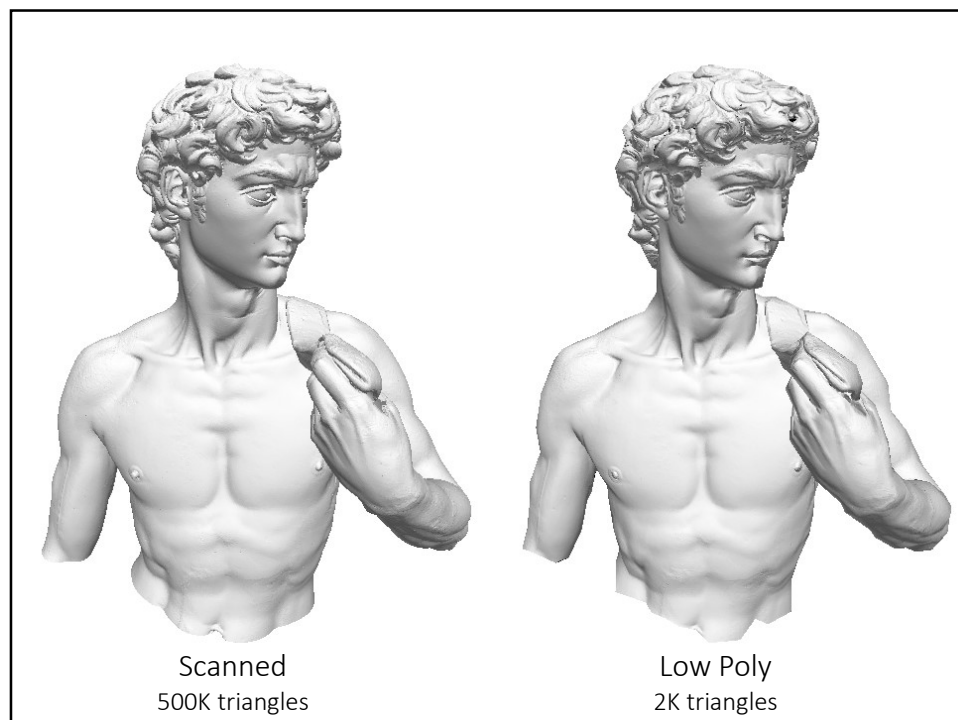
91



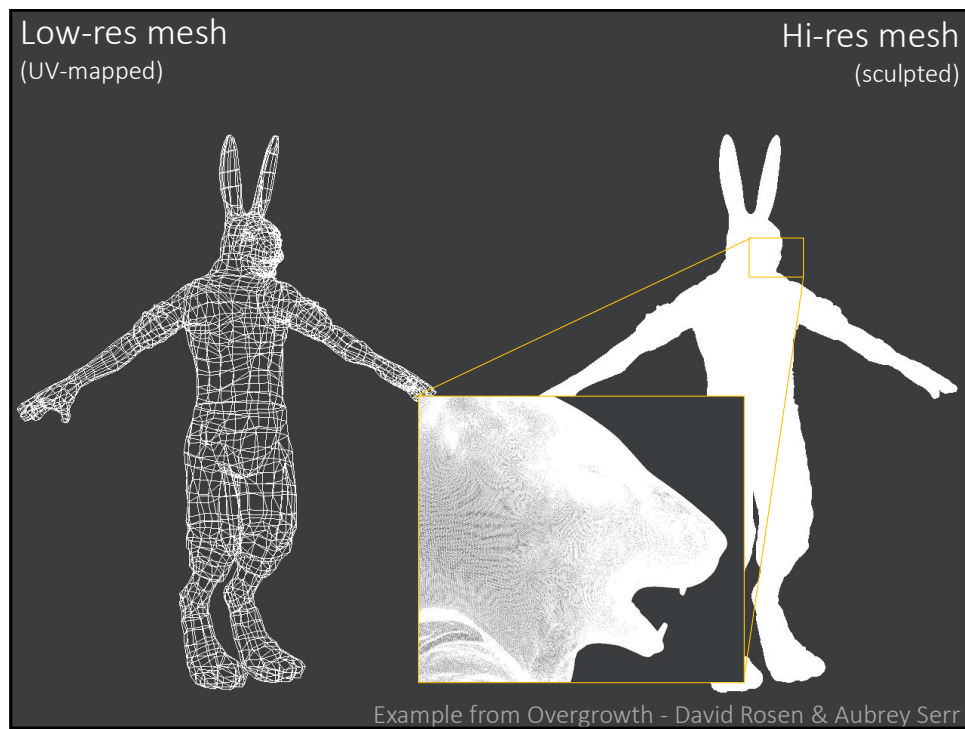
92



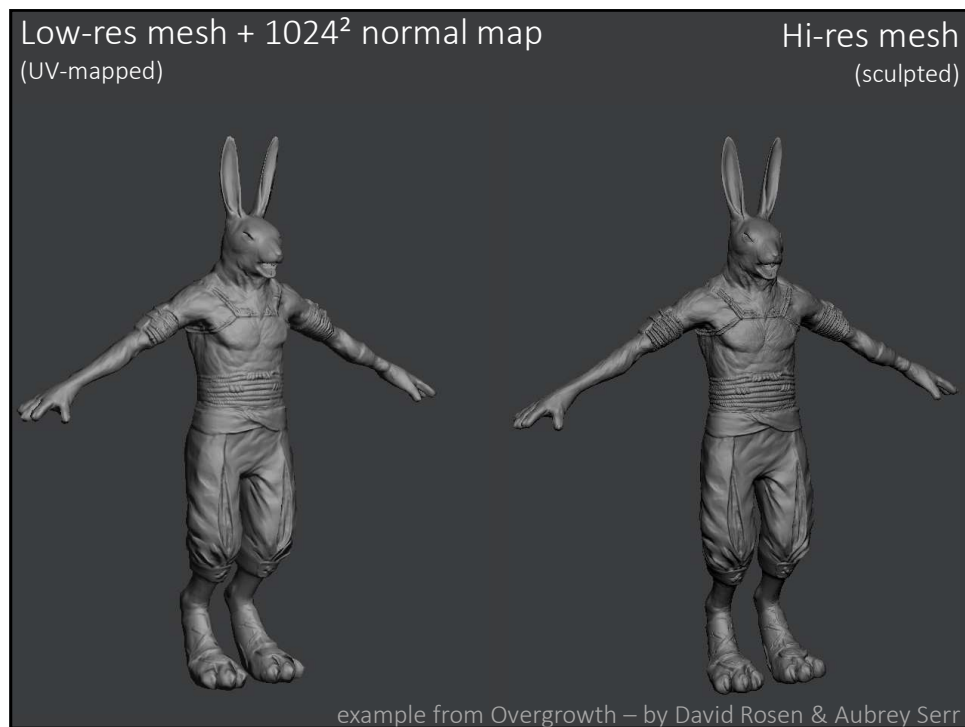
93



94



95



96

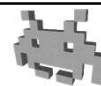
Asset production pipeline (a general concept in game-dev)



- A sequence of stages used to produce assets. Each stage:
 - what is produced, starting from what
 - using which tool(s), by which artist(s)
 - storing which intermediate result(s), in which format, etc.
- Different pipelines for different classes of objects
 - E.g. characters ≠ sceneries (“props”) ≠ equippable armours ≠ ...
 - Note: within a given game, all assets in a class are usually quite uniform (comparable resolution, same set of texture sheets, same formats, etc.)
- In the past lectures, we mentioned many possible steps
 - modelling (low poly modelling, sculpting, uv-mapping, LOD-ding...)
 - texturing, geometric proxies, ...
 - TODO: the parts about animations (skinning + rigging + animation...)
 - TODO: the parts about materials
- Identifying a good pipeline is not trivial!

102

Asset production pipeline: an example



1. Concept drawings
 - by a 2D artists
2. Low-poly model A
 - by a 3D modeler, using low-poly editing tools
3. UV-mapping of A
 - by a UV-mapper, or by automatic tool. output: an injective UV-map of A
4. Subdivision, then digital sculpting of Hi-Res model B
 - by a 3D modeler, using digital sculpting tools
5. Painting over B
 - using 3D painter, producing per-vertex colors
6. Texture baking
 - Automatic construction of three Textures for A with attributes from B:
 - Normals from B, (produces a normal map)
 - Colors from B (produces a diffuse map)
 - Baked lighting from B (produces a light-map)

103

Procedural Textures (in general)

$$f \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} r \\ g \\ b \end{pmatrix}$$

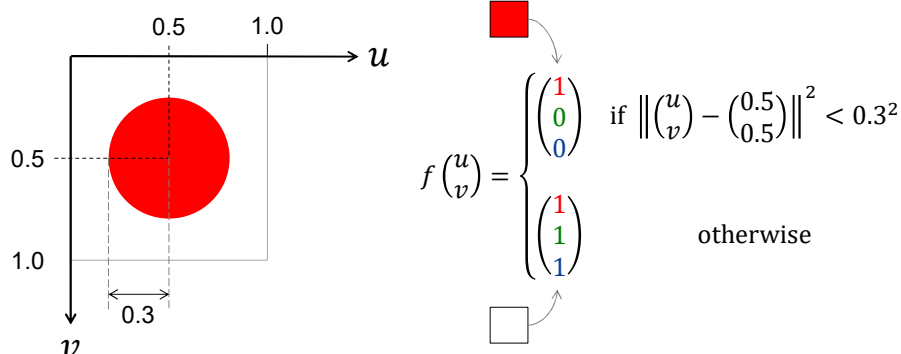
in $[0..1] \times [0..1]$

e.g. diffuse colors,
normals,
transparency, etc

- A function from (u,v) to texel values
 - Plainly *replaces* a texture fetch!
 - Computed *during rendering* for each pixel (fragment shader)
 - Therefore, implemented in shader languages (e.g. GLSL, HLSL)
- Costs/benefits (the usual ones): see Lecture on Rendering and Real Time Graphics course
 - RAM / bandwidth / storage cost: reduces to almost nothing
 - GPU usage: can be substantial (it's per pixel!)
 - resolution independent (similarly to a vector image)
 - control / authoring: can be difficult to get the desired effect
- Usually limited to simple images

104

Example: the flag of Japan as a procedural RGB-map



106



107

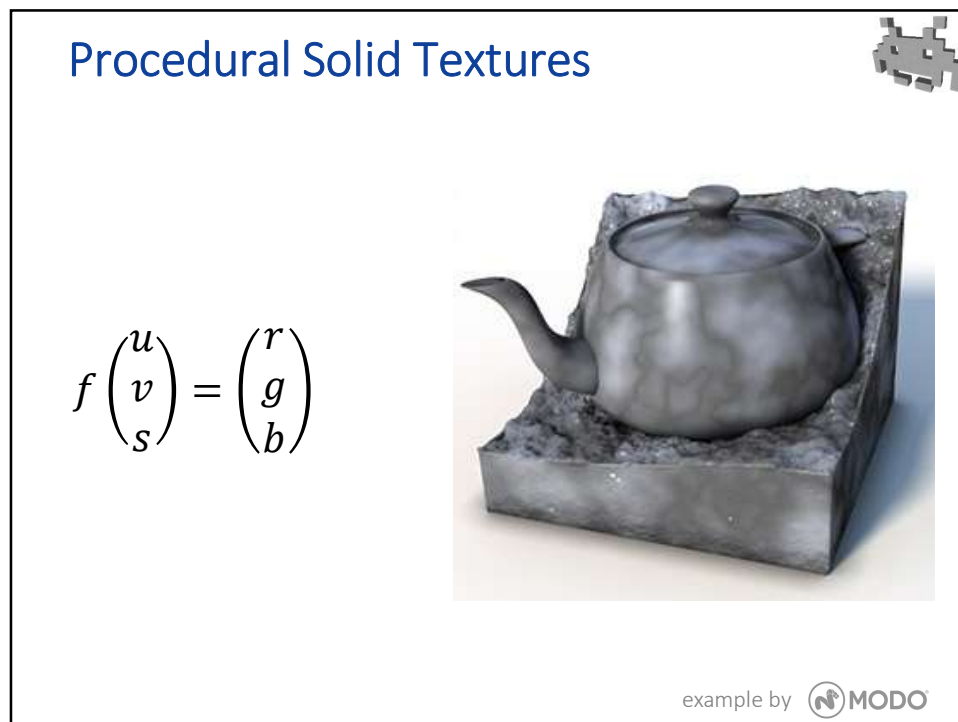
Solid Textures

- Volumetric voxellized **Texture**: 3D array of texels
- 1 texel == 1 voxel
 - E.g. each voxel one color RGB → **solid RGB textures**
- As all the textures:
 - In video RAM
 - Fast access during rendering
 - filtering (**tri**-linear) in access, MIP-mapping ...
- Model color onto volume
 - surface + internal
 - useful, e.g., for fractures
- Note: no need of **UV-map**!
 - Texture indexed by geometric mesh (rescaled)

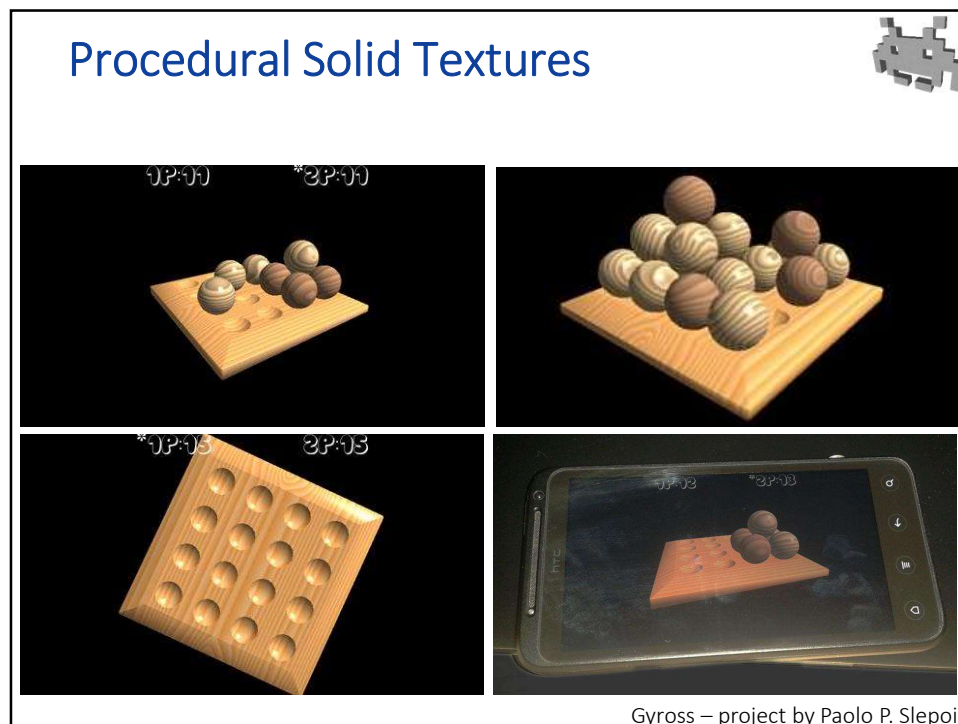
⚠ Problem: ram space

- Cubic wrt the resolution
- Solution: procedural 3D texture?

108



109



110