

# Course Plan

lec. 1: Introduction ●

lec. 2: Mathematics for 3D Games ●●●●●●

lec. 3: Scene Graph ●

lec. 4: Game 3D Physics ●●●● + ●●

lec. 5: Game Particle Systems ▸

lec. 6: Game 3D Models ●●

lec. 7: Game Textures ▸●

lec. 9: Game Materials ●

lec. 8: Game 3D Animations ▸●●

lec. 10: Networking for 3D Games ●

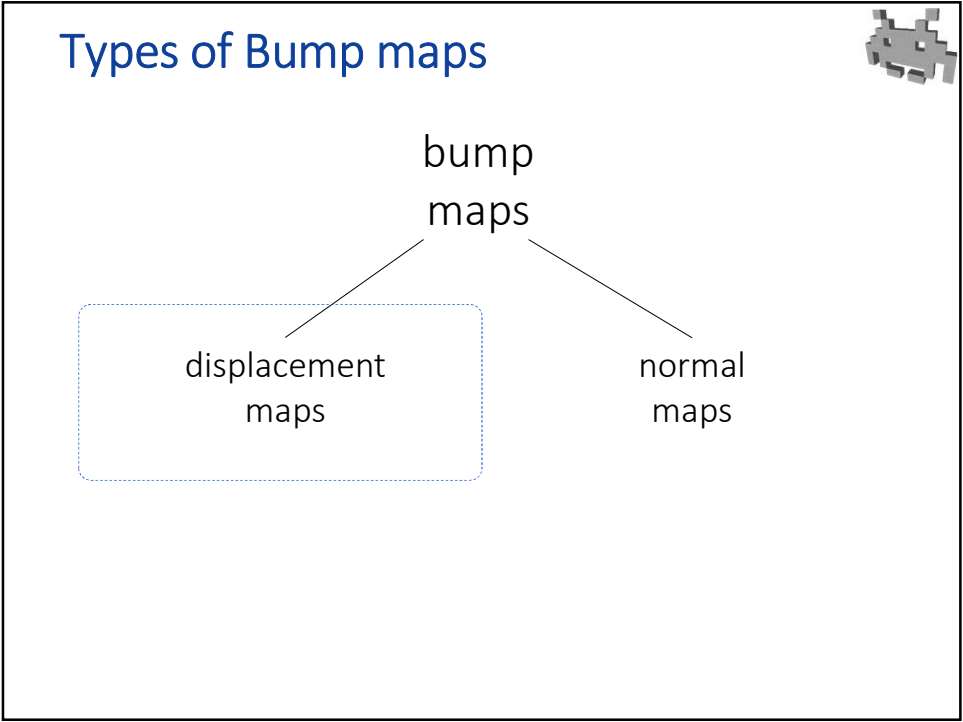
lec. 11: 3D Audio for 3D Games ●

lec. 12: Rendering Techniques for 3D Games ●

lec. 13: Artificial Intelligence for 3D Games ●

appearance

50



52

## Tassonomy of Bump maps (summary)



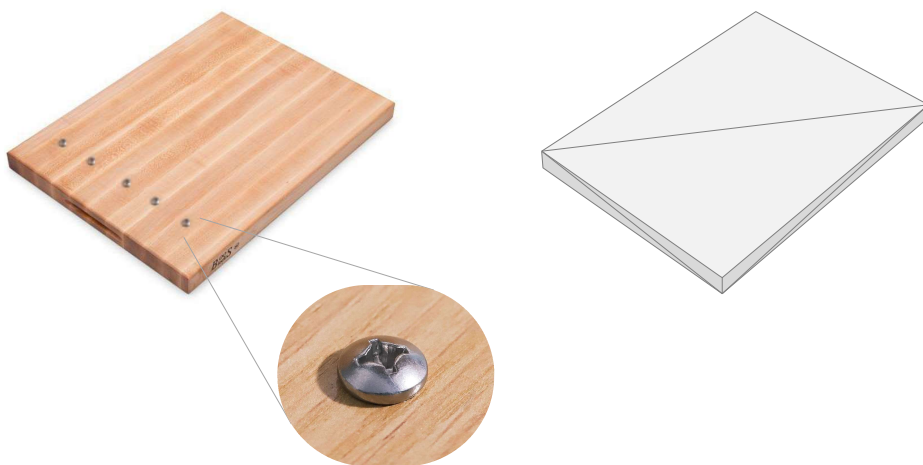
- **Bump map:**
  - Any texture encoding hi-frequency **geometric details**
- **Displacement Map:**
  - Details are encoded by storing geometric differences between the mesh geometry and the detailed surface:
  - either as **scalars** (distance along the normal), or as **vectors**
  - used for: on-the-fly re-tessellation, or *parallax mapping* technique
- **Normal Map:**
  - Details are encoded by storing the normals of the detailed surface
  - used for: lighting computation (affects lighting only)

Note: they are not mutually exclusive!

- One 3D model can have both
- (different texture “sheets” encoding different things using the same uv-map)

53

## For example

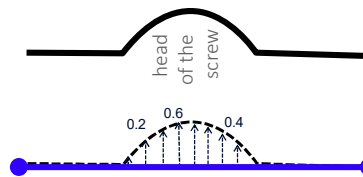


54

## (Scalar) Displacement map : concept

Stores the **distance** of the detailed surfaces  
from the plain geometry

- example: a bump-map for a screw-head



**Detailed surfaces**  
(what I would like to represent)

**low-poly mesh**  
the approximation (here: it's flat ☹)

0 0 0 0 0 0 0 1.5 6.6 7.5 4.2 0 0 0 0 0 0

**scalar displacement map**  
(1 texel = 1 scalar)

56

## Scalar 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



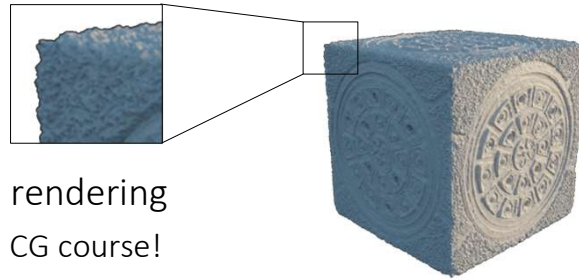
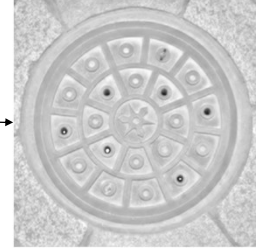
white = sticks outwards  
black = stays flat

Easy to paint (by artists)  
/ manipulate.

57

## (scalar) Displacement map rendering: parallax mapping

- Technique used to 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!

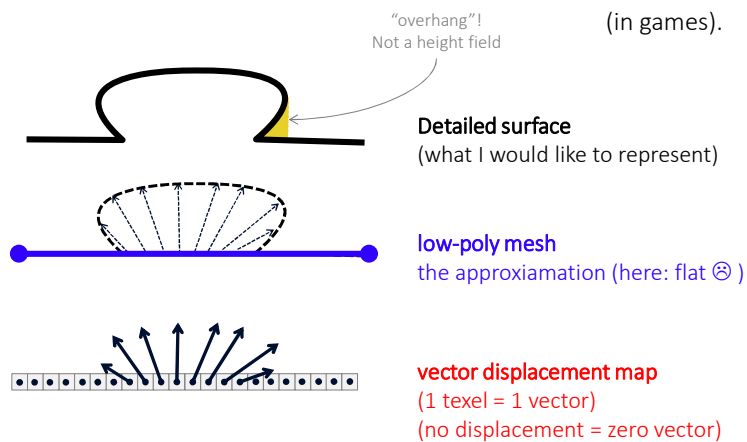
Image courtesy of <https://cgcookie.com/articles/normal-vs-displacement-mapping-why-games-use-normals>

59

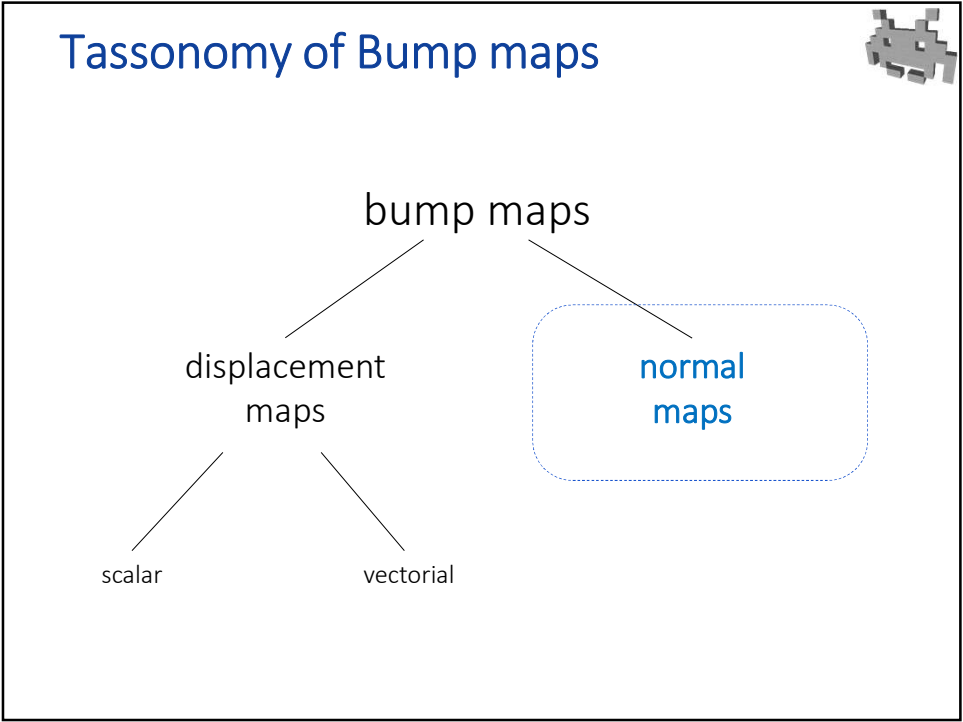
## Vectorial displacement map : concept

Store **Vectors** from the plain surface to the detailed surfaces

More expressive variant, but drastically more expensive and less usable. Not much used (in games).



60



61

### Normal Map: concept

Store the **Normals** of the detailed surfaces

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

head of the screw

**Detailed surface**  
(I would like to model)

**low-poly mesh**  
(the approximation) (here: flat ☹ )

**normal map**  
(one normal per texel)  
(no displacement = use geometric normal)

62

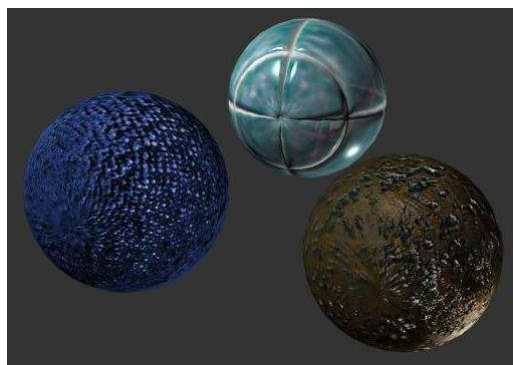
## Normal Map: notes

- Affects the lighting only
  - **not** the parallax
  - **not** the silhouette of the object
  - The lighting reflects the hi-freq detail of the object
    - dynamically (with variable lights!)
  - Resulting illusion: still convincent
    - 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 can be expressed simply in cartesian coord
  - But not always ( $\exists$  better ways to express unit vectors!)



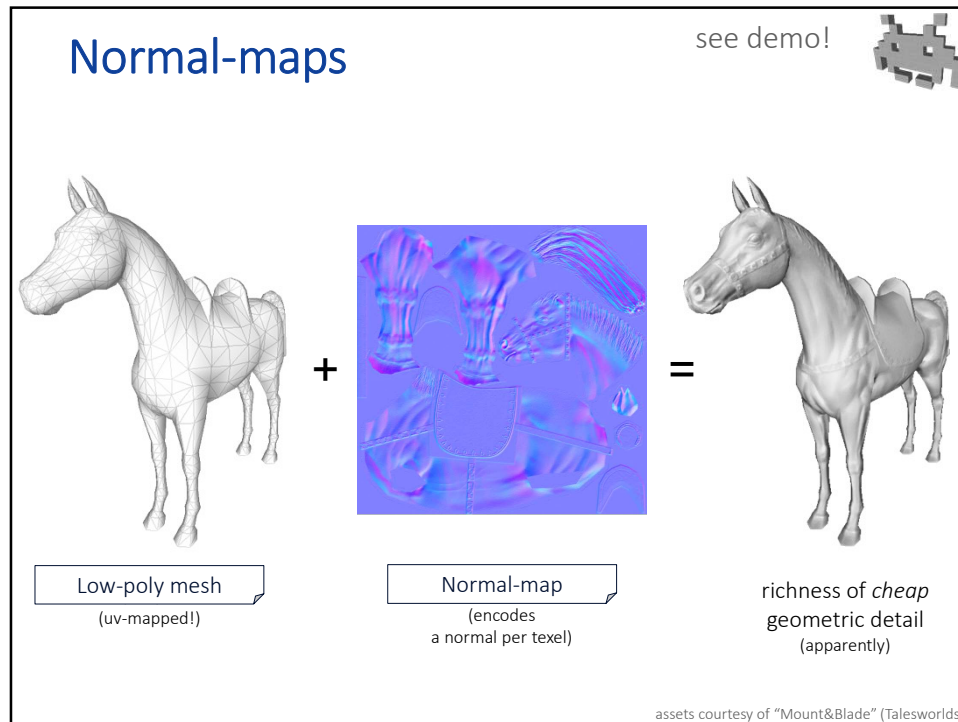
63

## Normal-maps



Same geometry (an approximated sphere)  
Different normal-maps

64



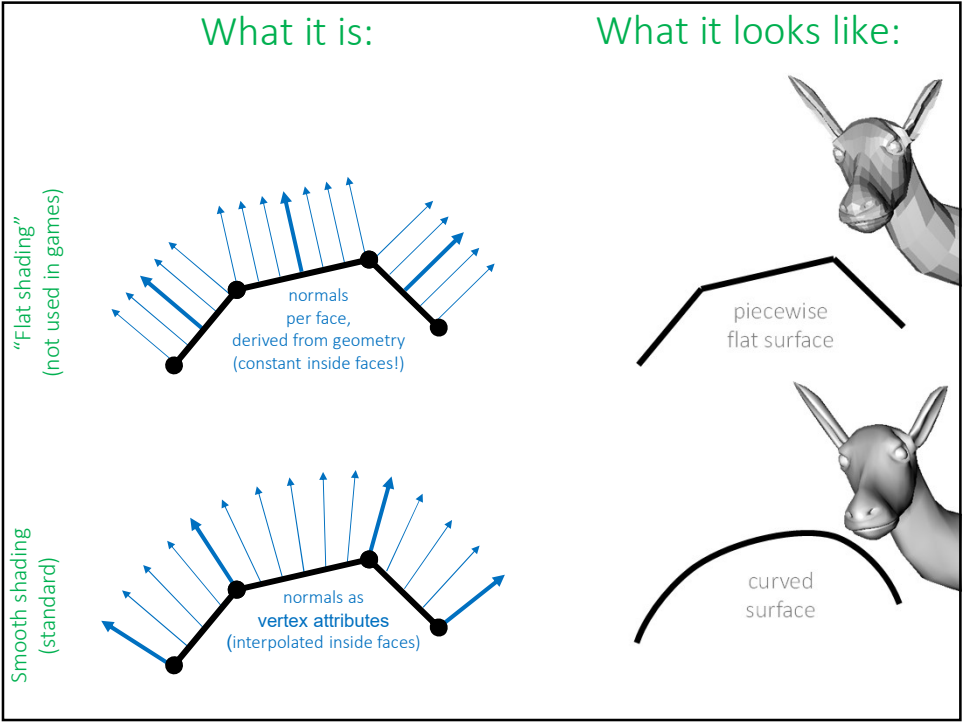
65

Normal maps only affect lighting

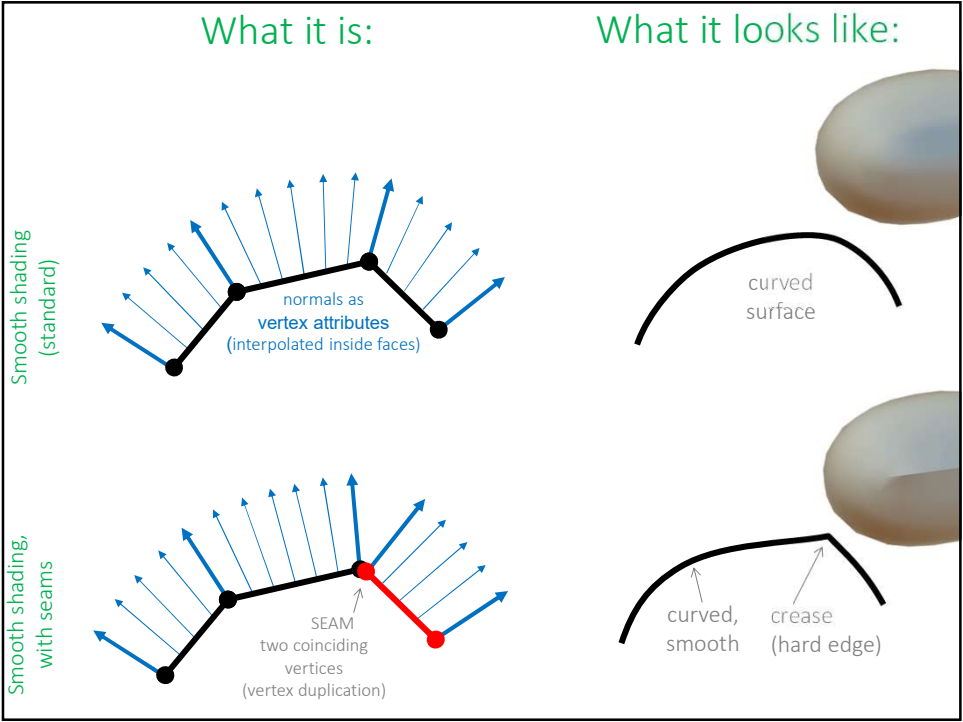
Do they provide a convincing illusion of 3D shape details?

- Normal maps won't affect:
  - Parallax, or object silhouette
- Still, illusion is very convincing
  - for small scale details, and adequate mesh/texture resolution
  - The lighting (think "*chiaroscuro*")  
tricks the eye notwithstanding the geometric shape
- It's not the first time we observe this in this course
  - We have seen other cases where the normal used for lighting is chosen independently from the actual 3D shape of the mesh
  - Normals (as vertex attributes, or texels value) don't have to be the real surface normal: small discrepancies are ok
  - For example...

66

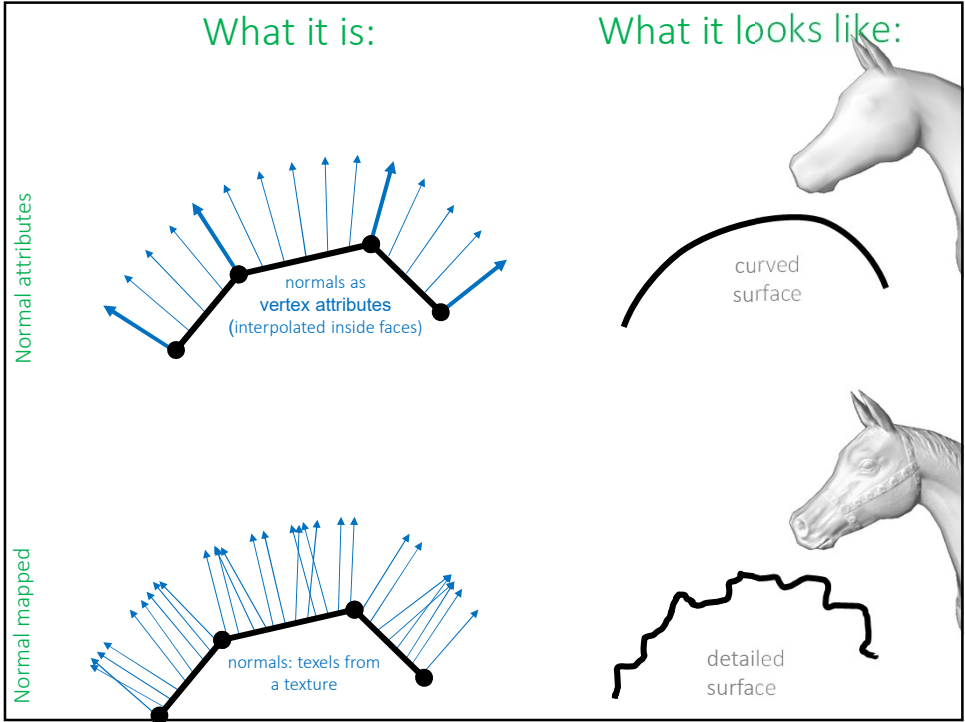


67

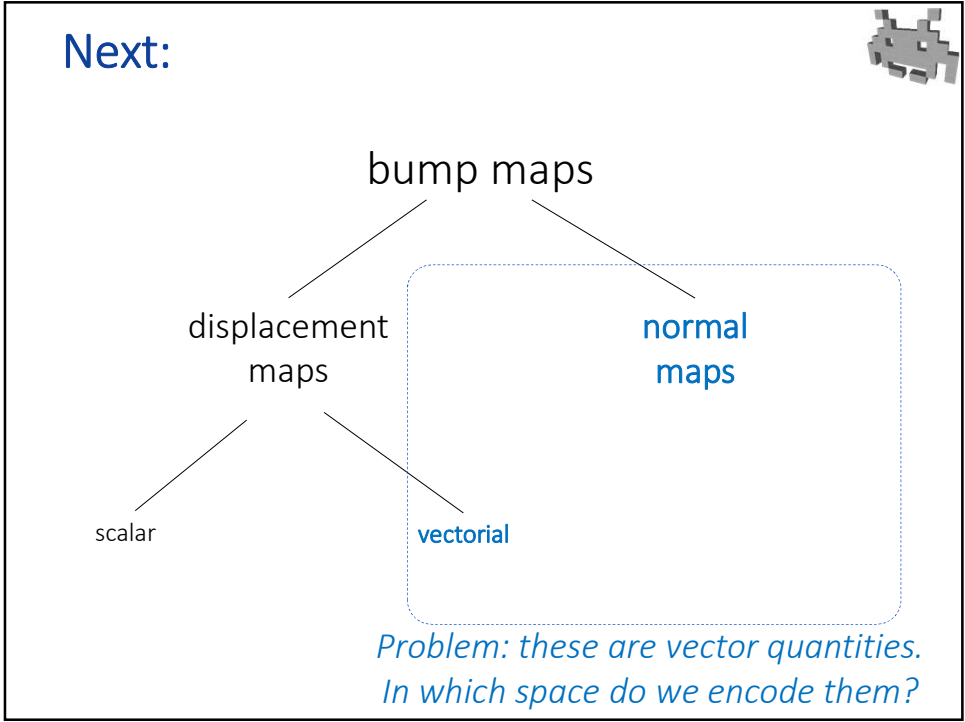


69





71



72

## Normal Maps: in which space are the normals encoded?



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

- If 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 simmetries

73

## Normal Maps: in which space are the normals encoded?



- Tangent space: **Tangent Space Normal-Maps** (the standard kind, in games)

- ☹ extra attributes are now needed per vertex:
  - Normal direction
  - Tangent direction
  - Bitangent direction
- ☺ 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

The tangent space

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

74

### Tangent directions are mesh-attributes

Tri:	W1:	W2:	W3:
T0	the connectivity		
T1			
T2			
T3			
T4			
T5			
T6			
T7			

INDEX BUFFER

vert	X	Y	Z	Tu	Tv	Nx	Ny	Nz	Tx	Ty	Tz	Bx	By	Bz	...	...	...	...
V0	the geometry		the UV-map		normals			tangent directions				...						
V1																		
V2																		
V3																		
V4																		

VERTEX BUFFER

75

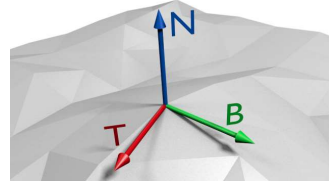
### Tangent space (aka TBN space)

- 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)

76

## Tangent space (aka TBN space)

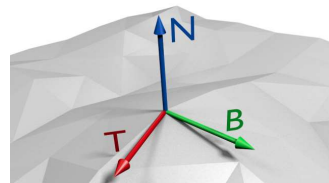
- How to store them?
  - As 3 vectors stored as (per-vertex) **attributes**
    - So, they are interpolated inside faces (like any other attribute)
  - Optimizations are possible!
    - Not necessarily stored as 3 vectors (9 scalars)
    - E.g.: instead of storing B, we store N and T, then  $B = N \times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



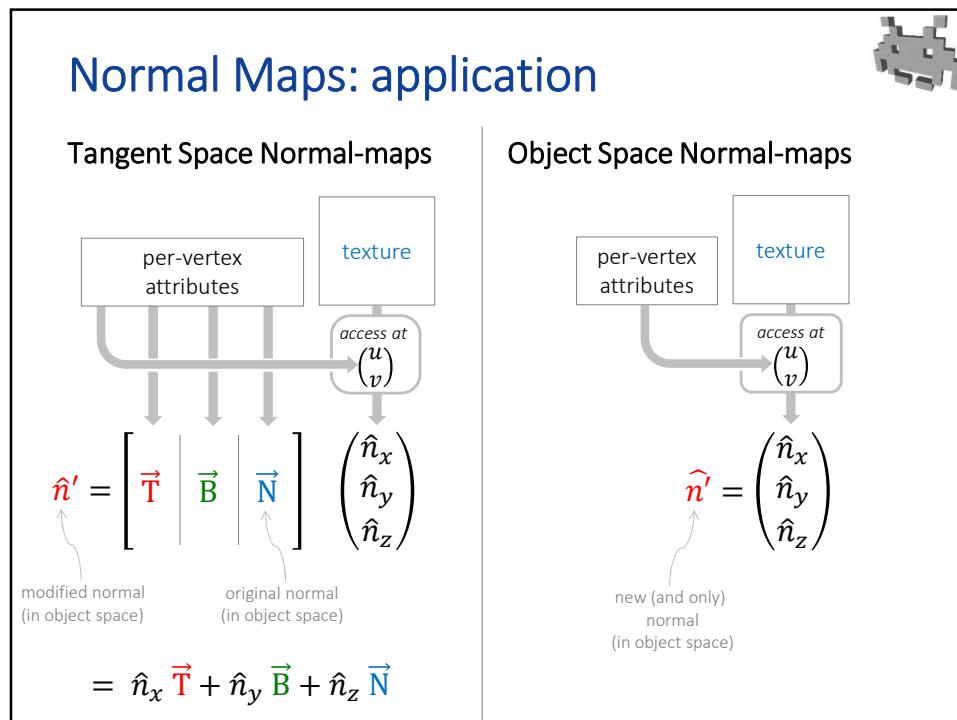
77

## Tangent space (aka TBN space)

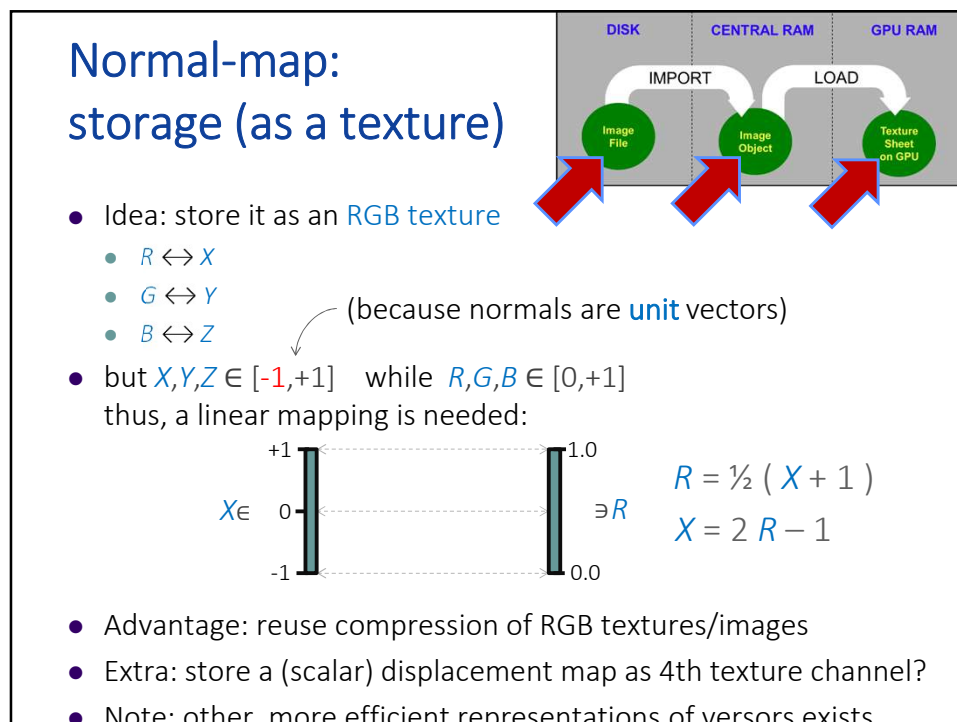
- How to compute them?
  - **Normal**
    - as usual (see lecture on mesh)
  - **Tangent & Bi-Tangent**
    - determined by the **UV-map**!
    - T = **gradient** of U coordinate
    - B = **gradient** of V coordinate
- details:
  - All three are defined (and constant) inside faces, then averaged at 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)



78



79



80

### Normal-maps: storage (as a texture)

- Examples of tangent space normal-map

DISK

CENTRAL RAM

GPU RAM

IMPORT

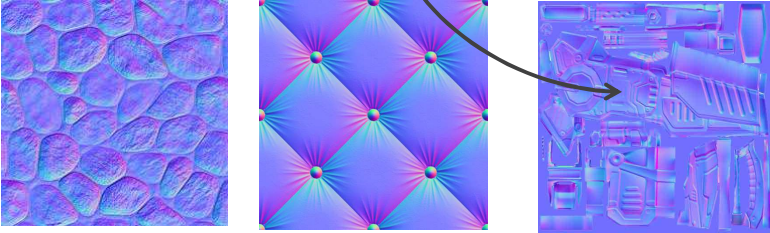
LOAD

Image File

Image Object

Texture Sheet on GPU

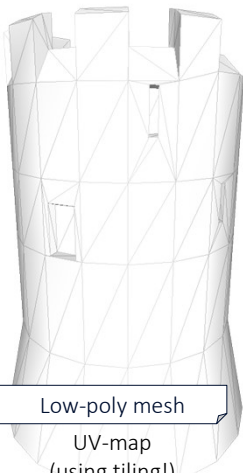
Prevailing normal :  $X \approx 0$  ,  $Y \approx 0$  ,  $Z \approx 1$   
 $\Rightarrow$   
Prevailing color:  $R \approx 0.5$  ,  $G \approx 0.5$  ,  $B \approx 1$   
(~light blue)



81

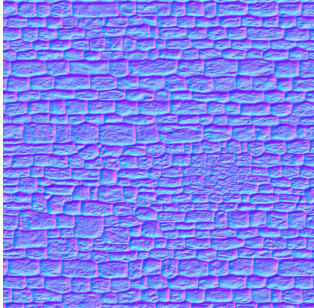
### E.g.: Tiled (tangent space) Normal Maps

not even possible, with object-space NM!




Low-poly mesh  
UV-map  
(using tiling!)  
Tangent dirs.

+



Normal-map  
Tileable!

=



assets courtesy of "Mount&Blade" (Talesworlds)

82


Marco Tarini

Università degli studi di Milano

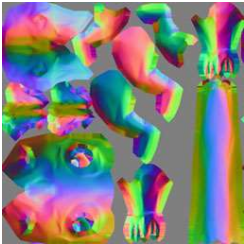
14

### Bump-maps assets at a glance

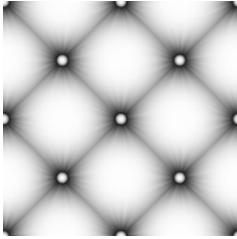
(can you tell which is which?)



Tangent Space Normal map



Object Space Normal map

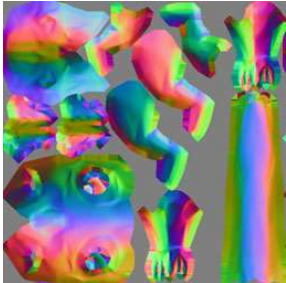


Displacement Map (scalar)

the default kind

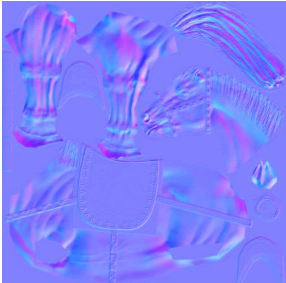
83

### Spot the difference



Object Space Normal map


1:1 UV-map  
right leg  $\neq$  left leg



(Tangent Space) Normal map

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

84

Normal map comparison (a summary) 	
Object Space Normal map:	Tangent Space Normal map:
<b>Replaces</b> the normals of the object	<b>Modifies</b> the normals of the object
No <b>normal attribute</b> required on the mesh anymore	Requires two <b>extra attributes</b> on the mesh: T an B versors (in addition to the normal)
Constructing the texture requires to <b>know the mesh</b> it will be applied to	Textures can be constructed <b>independently</b> from the mesh (just like a color map!)
E.g., a normal map cannot be constructed from a <b>displacement map</b> (w/o the mesh)	E.g., a normal map can be constructed from a <b>displacement map</b>
It's <b>impossible to share</b> a normal map between models (barring exceptions)	Normal maps <b>can be shared</b> between different models
" <b>unwrapping</b> " <b>UV-maps</b> required (barring exceptions)	Can be applied to <b>non-injective UV-maps</b>
E.g., <b>no tiled</b> textures. E.g., <b>no symmetry</b> exploitation	E.g., <b>tiled</b> 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)
<b>MUCH MORE USED IN GAMES</b>	

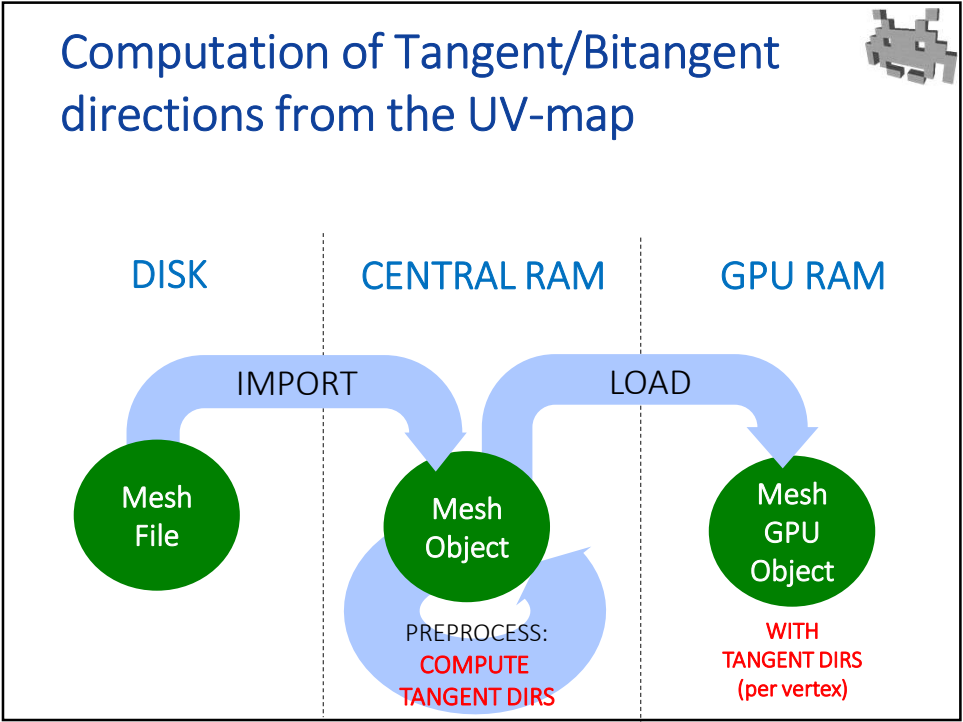
85

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

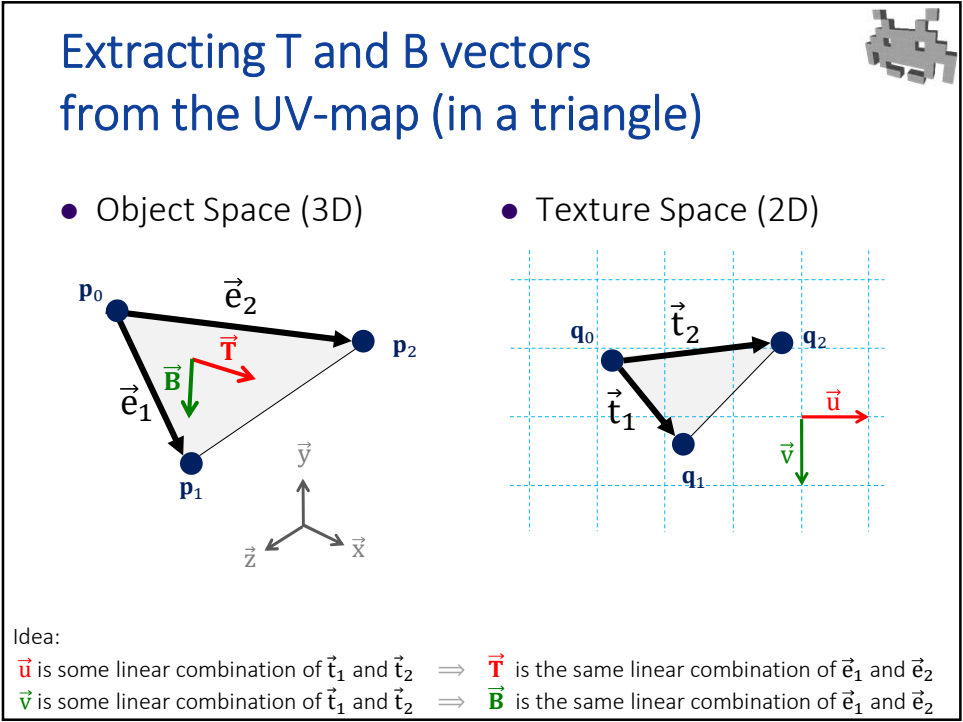
- Concept (a mental experiment)
  - STEP 1: color a texture with a grid
    - horizontal blue lines = U direction
    - vertical red lines = V direction
  - STEP 2: apply it to the Mesh!
  - STEP 3: look at it:
    - the T vectors are the Blue lines directions
    - the B vectors are the Red lines directions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

86





87



88

## 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  $\vec{\mathbf{e}}_{1,2}$   
and 2D edge vectors  $\vec{\mathbf{t}}_{1,2}$
- Find scalars  $a, b$  and  $c, d$  such that...

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

- Then

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

89

## 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:

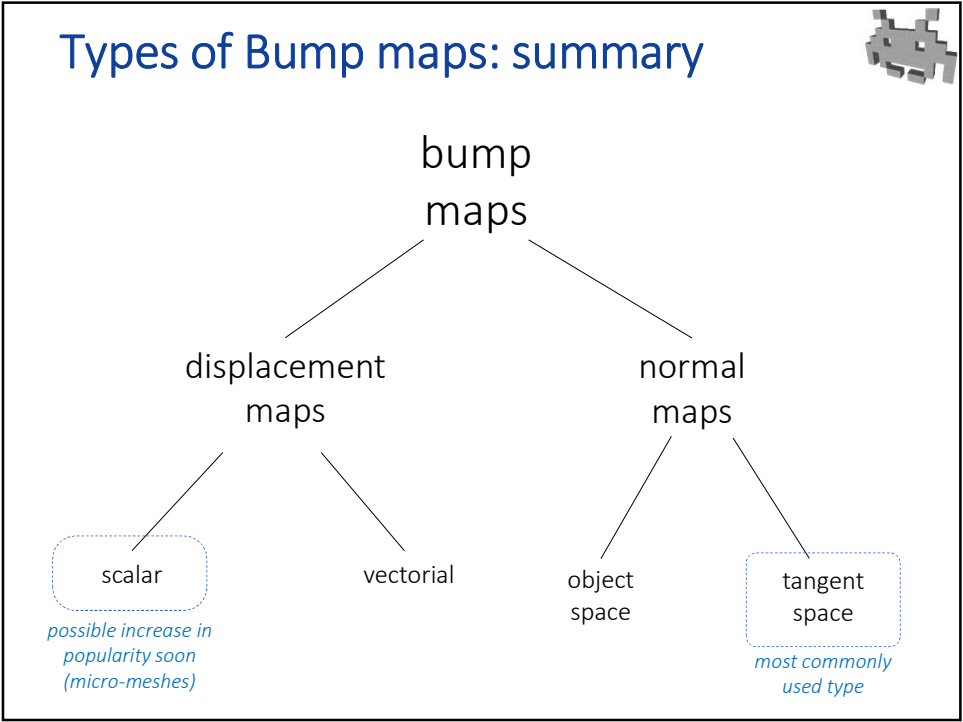
solve with a 2x2 matrix inversion

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

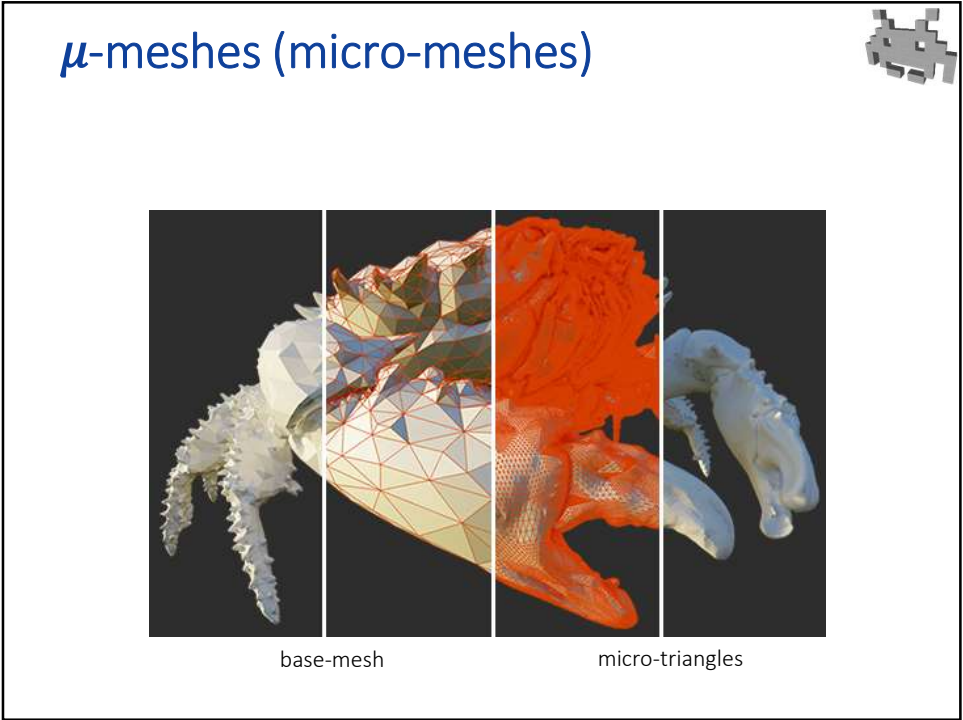
- Then

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

90

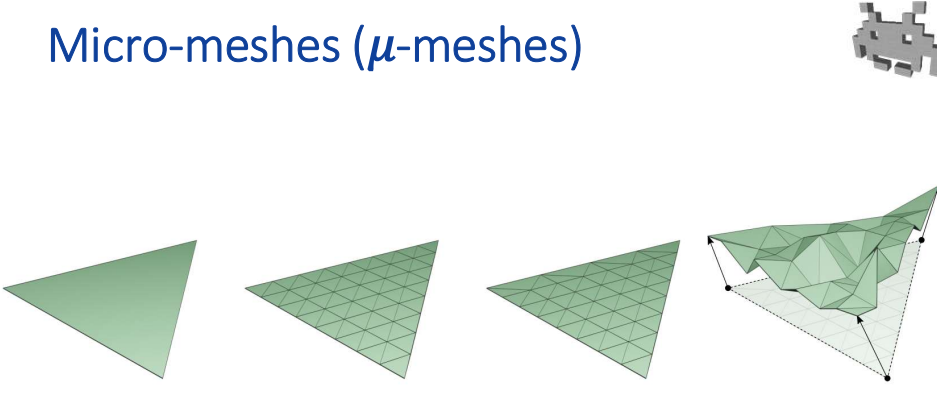


91



92

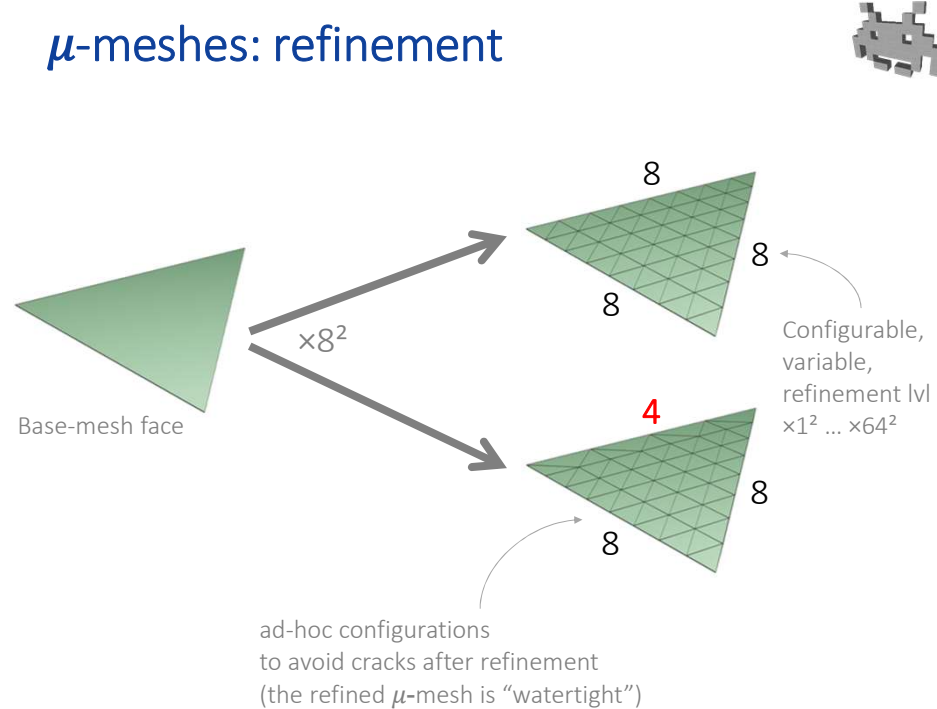
Micro-meshes ( $\mu$ -meshes)



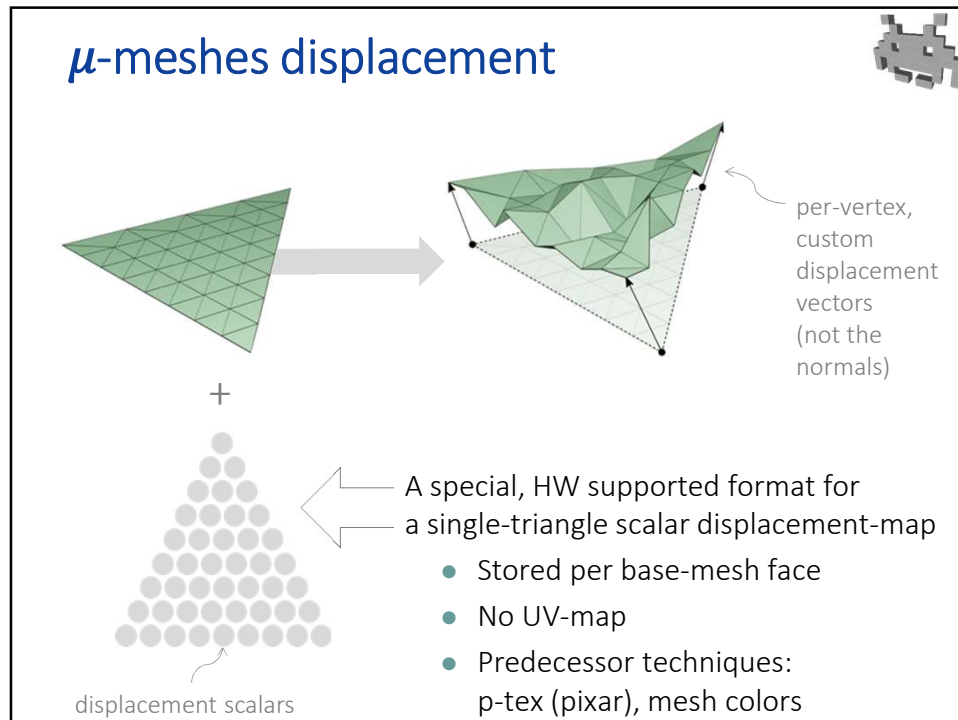
- A coarse mesh with a specially formatted scalar displacement map
- During rendering: refine + displace

93

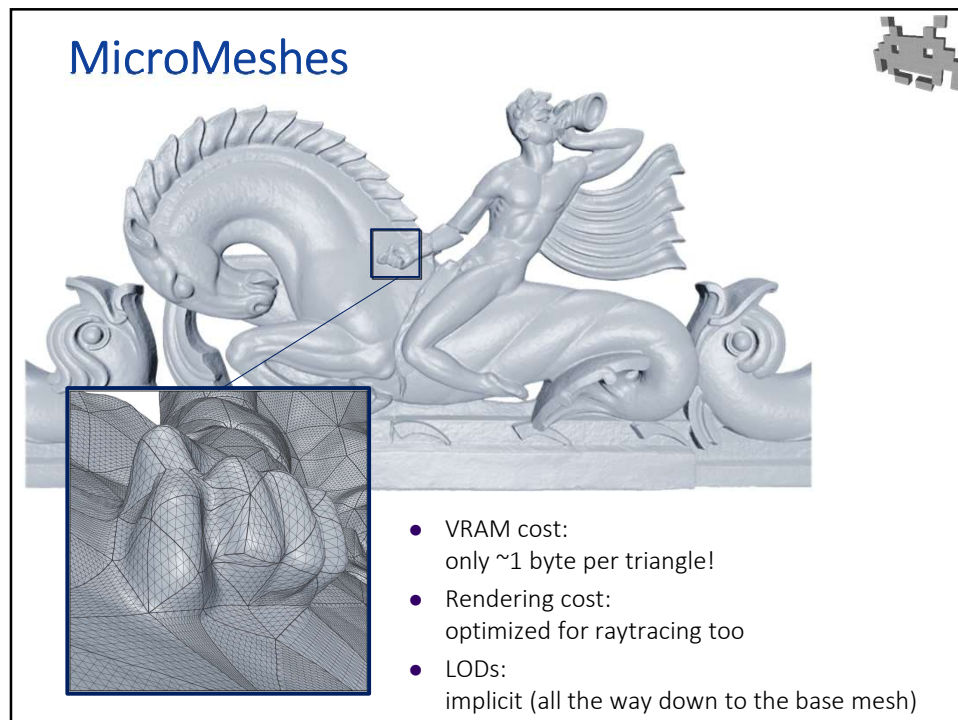
$\mu$ -meshes: refinement



94



95

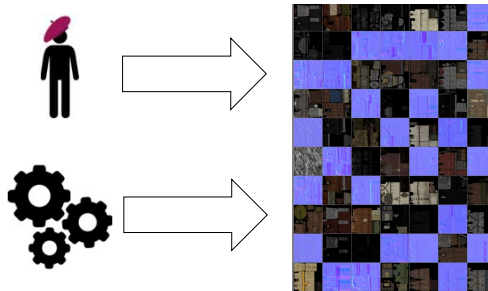


97

## Something about authoring textures

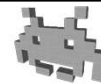


- How can textures be authored / created?
  - As a part of asset production pipelines
  - Let's see the basics on authoring of color-maps, normal maps, and so on

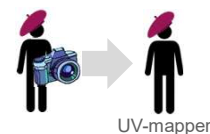


99

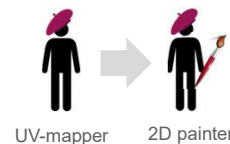
## RGB maps authoring: when (and how)?



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

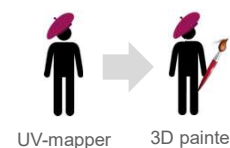


UV-mapper



UV-mapper

2D painter



UV-mapper

3D painter

100

## RGB maps authoring: when (and how)?



...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

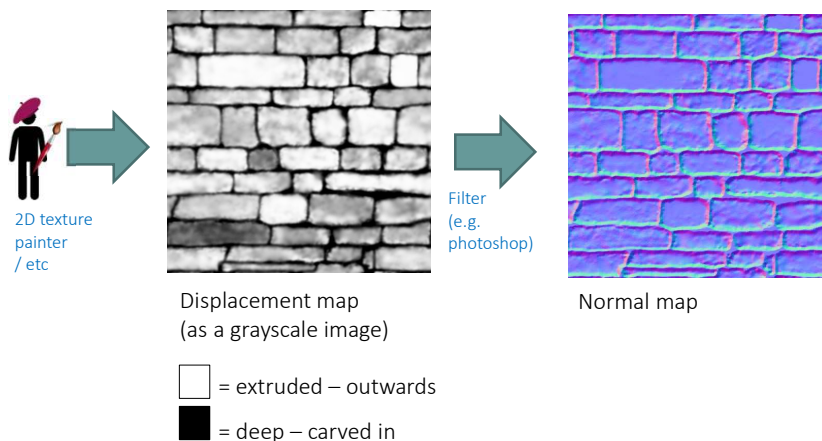
*more  
about  
this later...*

101

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



see demo!



102

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



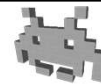
- Input: a scalar displacement map  
Output: a normal map
- Algorithm (2D image processing):
  - $\forall$  texel  $\mathbf{t}$  of displacement map, compute **best fitting plane** around  $\mathbf{t}$ 
    - Consider all 3D points in a  $3 \times 3$  patch surrounding  $\mathbf{t}$
    - Find plane 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  $5 \times 5$ ,  
or  $7 \times 7 \dots$

103

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



- Direct painting of normal- on the model
  - (can be done, e.g., with Z-brush, Blender...)
- Similar to: 3D painting of color-maps
  - but artist paints geometric surface orientations, not colors
- Similar to: mesh sculpting
  - but, for each stroke, the system updates the normal on the texture-map, not the geometry on the mesh

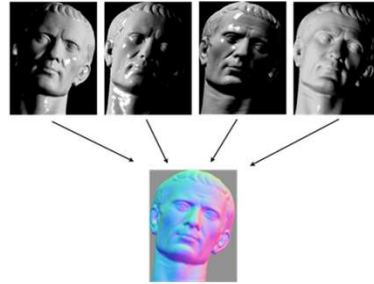
104



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

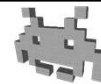


- Captured from reality, using photos
- Example: “**Photometric Stereo**”
  - a form of “inverse lighting”
  - a **computer vision** technique
- Input:  $n$  real images
  - Same viewpoint
  - Different illumination
    - possibly, controlled and known
- Output: a Normal Map
  - expressed in image space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



105

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



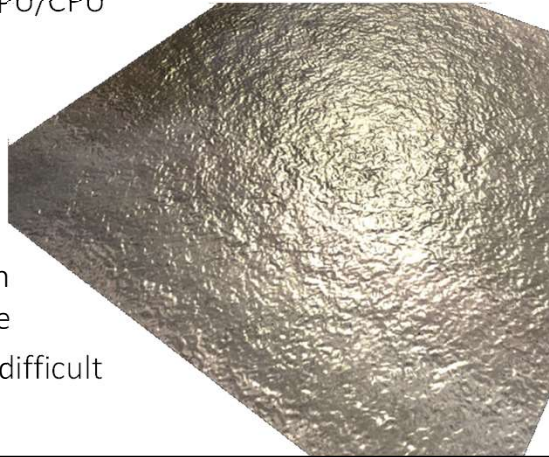
- Normal map estimation from images
  - Traditionally, many pictures are required in input
  - Traditionally, controlled illumination is required (I must place lights in known position)
  - With Machine Learning, it's becoming possible to use a single image with natural illumination
- Idea:
  - input: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

106

## How are normal-maps obtained? (4/5) procedural generation (not frequent)



- Usual considerations about **procedural**ity:
  - 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 is difficult



107

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



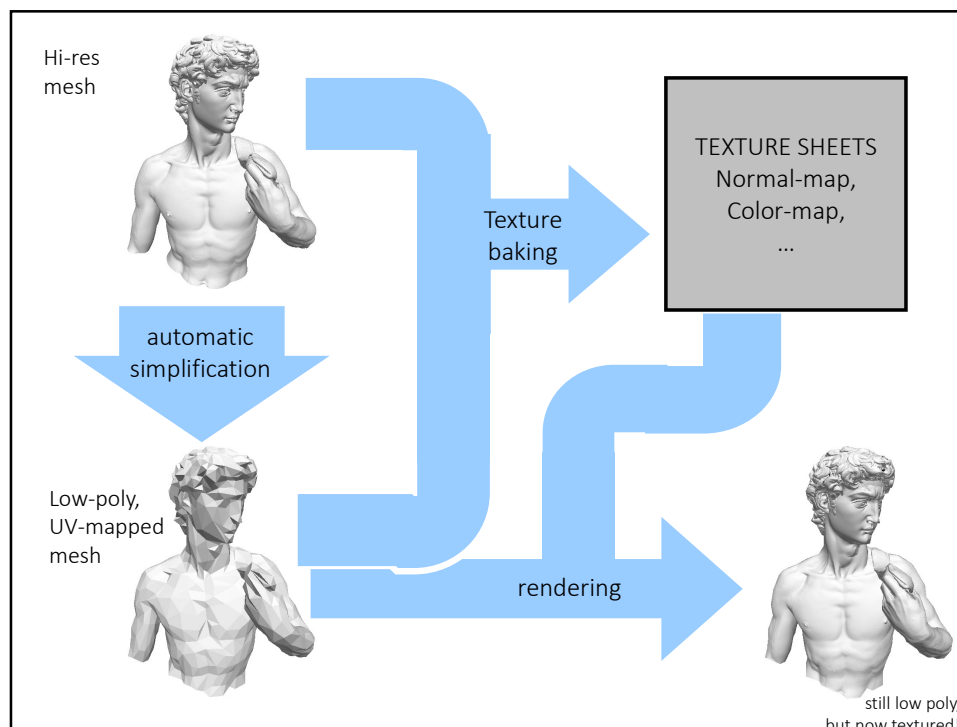
- **textures baking**
  - aka: detail recovery / “detail texture” synthesis / texture for geometry
- input:
  - hi-res mesh A with **per-vertex attributes**
  - low-poly mesh B, with an **injective UV-map**
- output:
  - textures for B storing the attributes of A
- a fully automatic process!

108

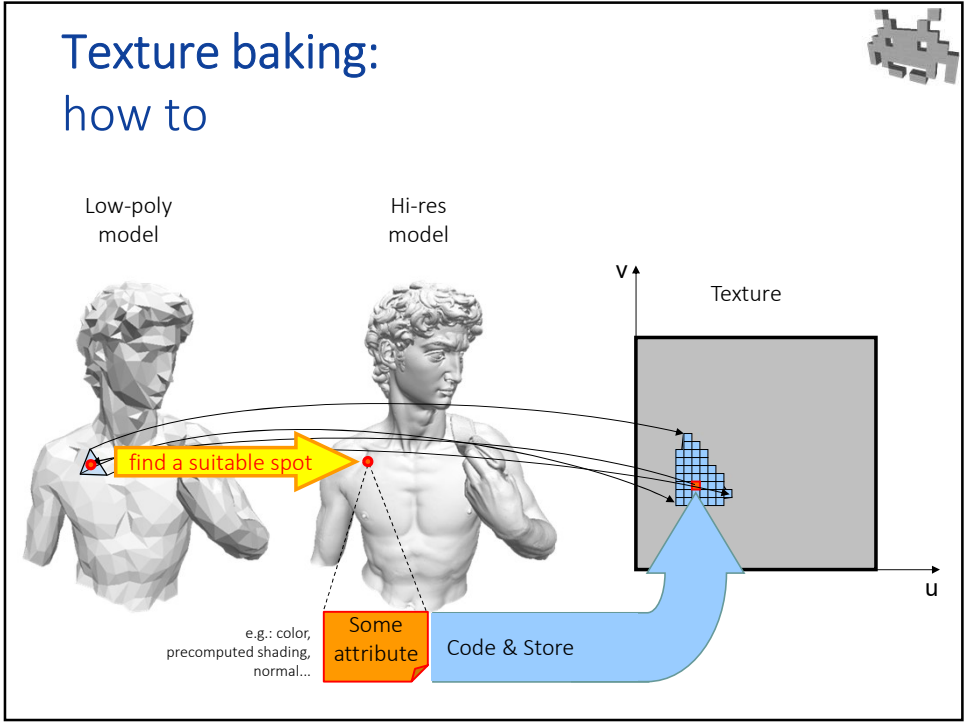
## Texture baking: texture synthesis from hi-res models

- examples of inputs:
    - 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**
  - examples of output:
    - 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!

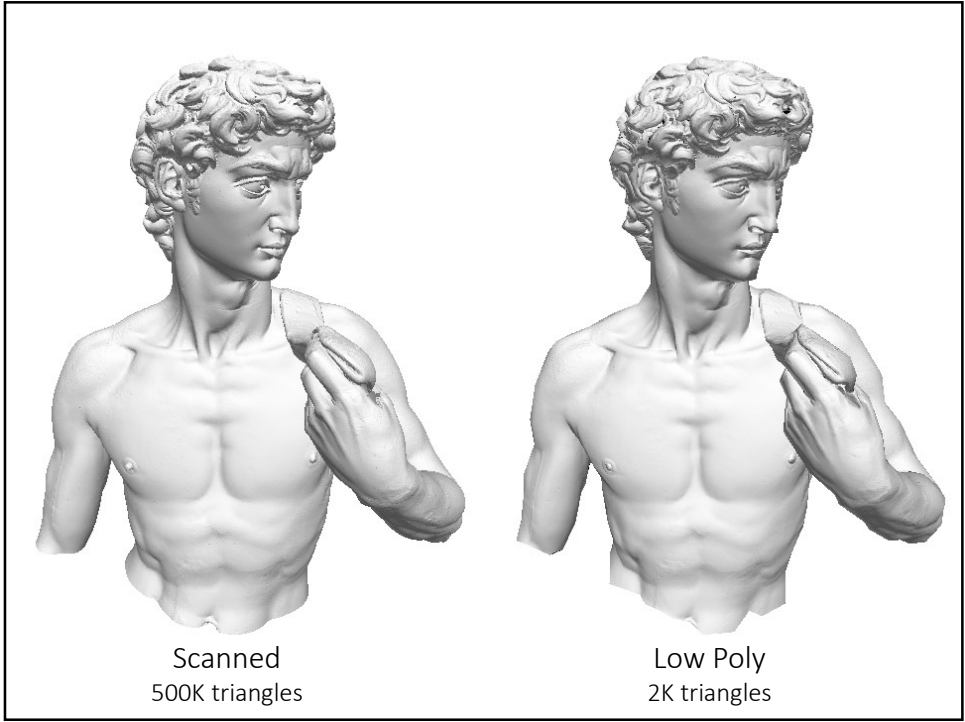
109



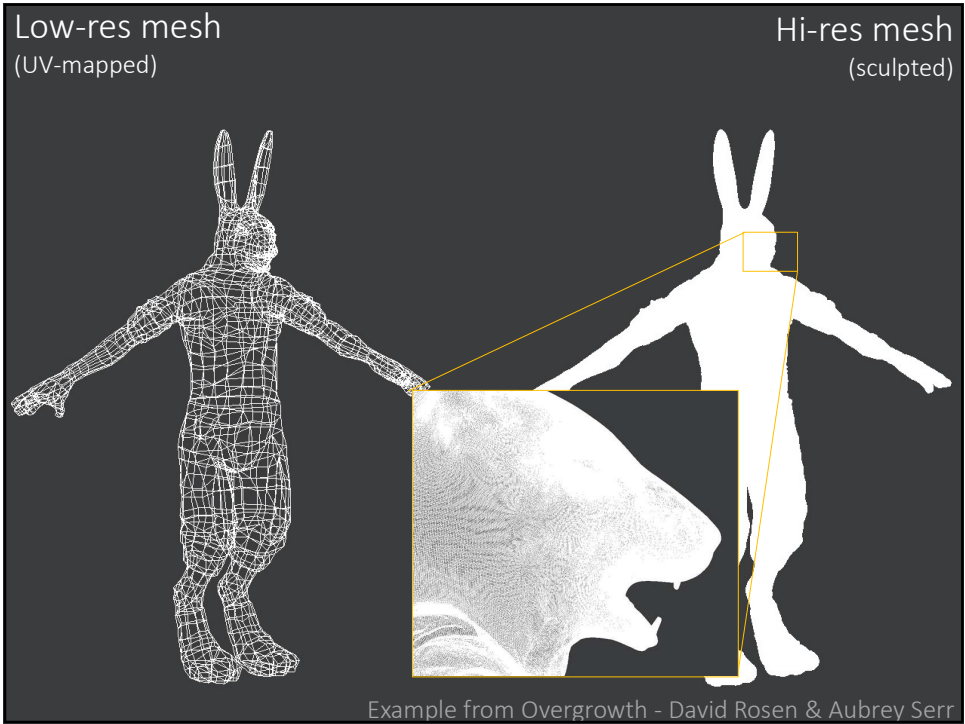
110



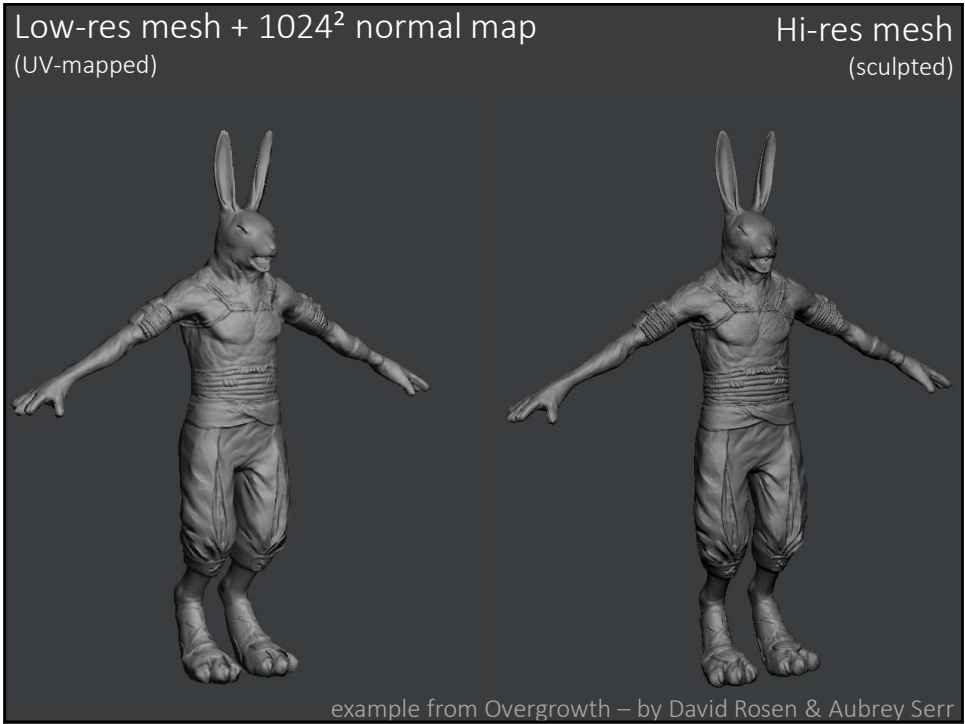
111



112



113



114

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



- A sequence of stages used to produce assets. For 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
  - low poly modelling, sculpting, uv-mapping, LOD-ding...
  - texturing, geometric proxy construction, ...
  - TODO: the parts about animations (skinning + rigging + animation...)
  - TODO: the parts about materials
- Identifying a good pipeline is not trivial!

120

## Asset production pipeline: an example



1. Concept drawings
  - by 2D artists
2. Low-poly model A
  - by 3D modelers, using low-poly editing tools
3. UV-mapping of A
  - by UV-mappers, or by automatic tools. 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 a 3D painter, and 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)

121

## Procedural Textures (in general)

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

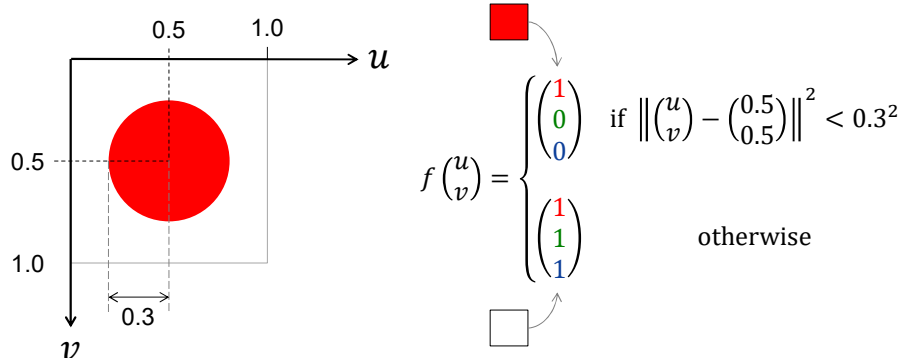
in  $[0..1] \times [0..1]$

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: reduced 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

122

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



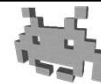
124

## Solid Textures



125

## 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?


126



Procedural Solid Textures

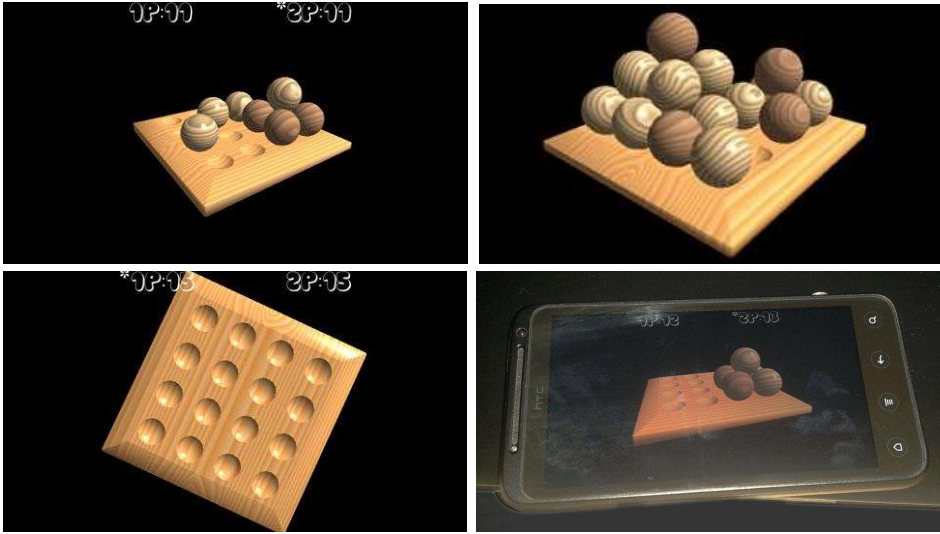


$$f\begin{pmatrix}u\\v\\s\end{pmatrix}=\begin{pmatrix}r\\g\\b\end{pmatrix}$$

example by 

127

Procedural Solid Textures



Gyross – project by Paolo P. Slepoi

128