12. Normal Mapping and Tessellation

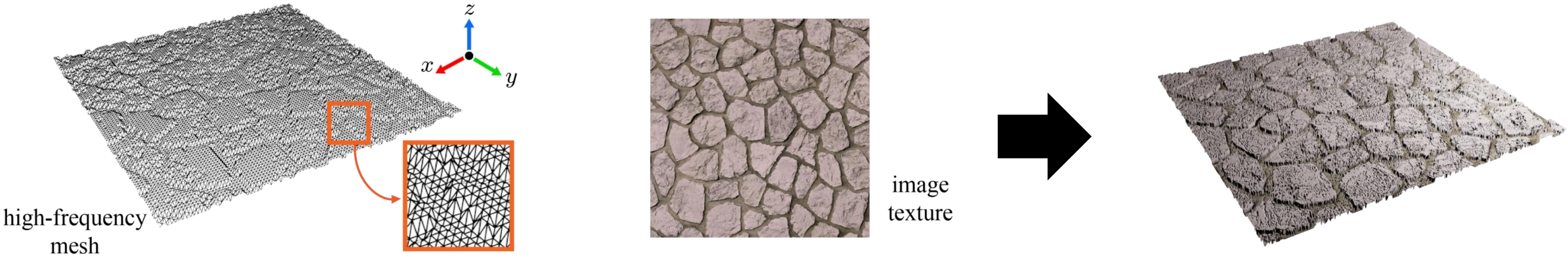
Advanced
Game Graphic Programming

Prof. HyeongYeop Kang
siamiz@khu.ac.kr

Bumpy surfaces

High-resolution polygon mesh

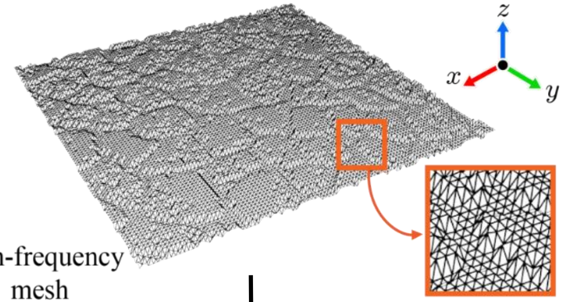
- Brick walls and paved grounds have bumpy surface.
- To produce the realistic result, high-resolution (frequency) mesh is required.



Bumpy surfaces

High-resolution polygon mesh

- Unfortunately, high-resolution meshes are expensive to render.
  - More vertices need to be transferred from CPU to GPU.
  - More vertices need to be processed in the graphics pipeline.



high-frequency
mesh

CPU cores and Caches
(multi-core)

CPU memory bus

Host Memory
(CPU side memory)

data bus

GPU cores and Caches
(many-core)

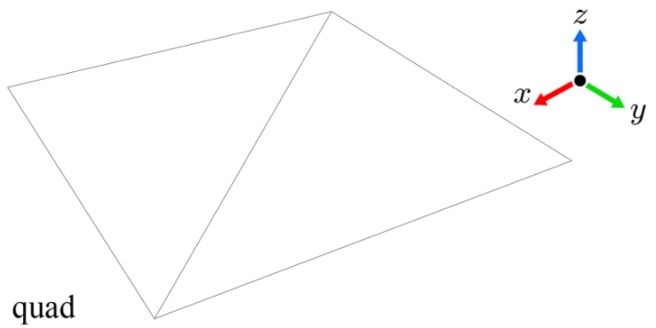
GPU memory bus

Device Memory
(GPU side memory)

Bumpy surfaces

Low-resolution polygon mesh

- Low-resolution meshes are cheaper to render.
- If they are textured with the paved-ground image, it looks not bad.



Bumpy surfaces

High-resolution mesh vs. Low-resolution mesh

- However, low-resolution meshes do not properly expose the bumpy features even though they are textured with the paved-ground image.
- This is due to the normal.

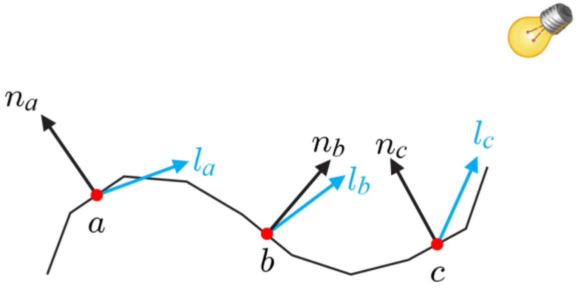
high-resolution mesh

low-resolution mesh

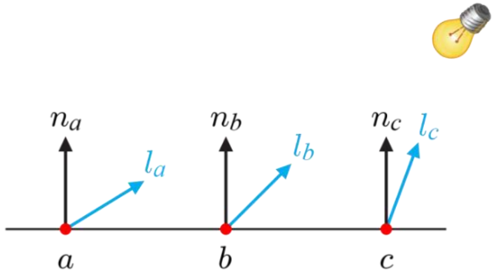
Bumpy surfaces

High-resolution mesh vs. Low-resolution mesh

- The normal data difference is the reason.



high-resolution mesh

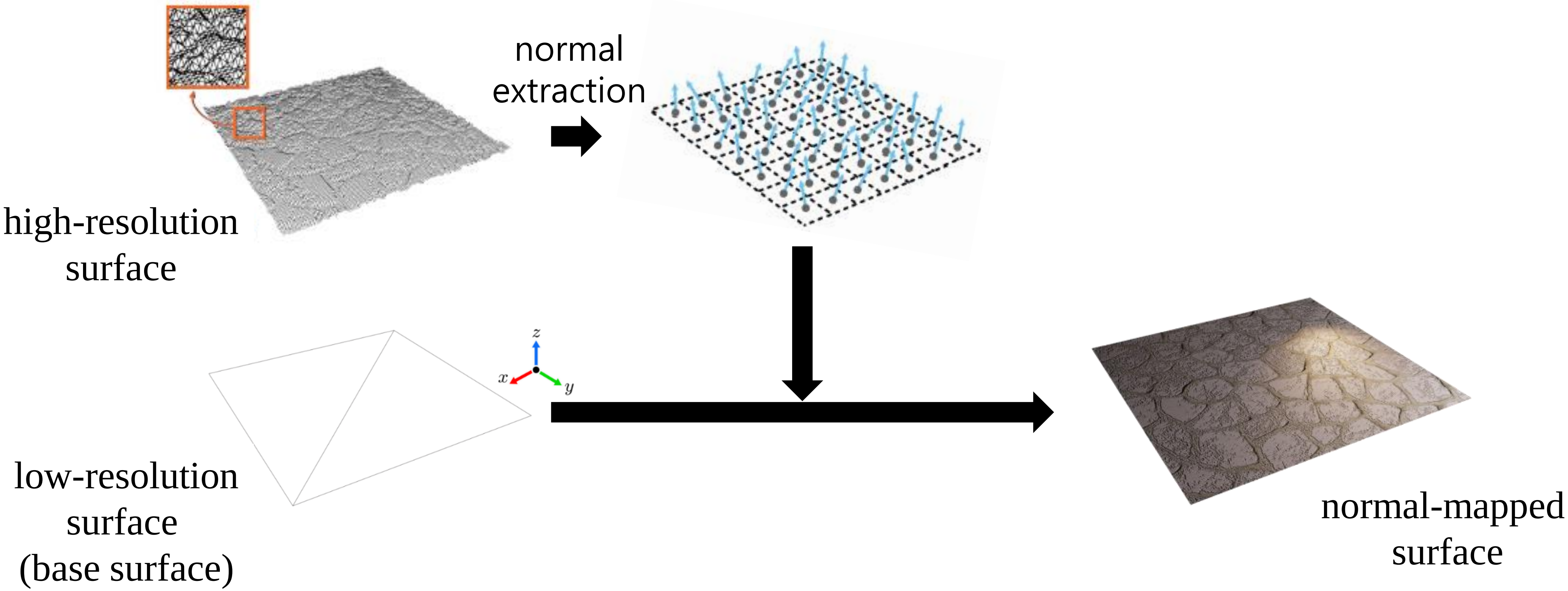


low-resolution mesh

Normal Mapping

Pre-computed normal

- A way out of this dilemma is to *pre-compute* and *store* the normal of the high-resolution surface into a special texture named *normal map*.
- A normal map can be used at run time for lighting with a lower-resolution mesh which we call *base surface*.



Normal Mapping

Normal-mapped surface vs. low-resolution surface

- Without increasing the number of vertices, rendering quality is increased.

normal-mapped surface

low-resolution surface

Normal Mapping Generation

Image to height map

- Simple image-editing operations can create a gray-scale image (height map) from an image texture.
- Height map is often visualized in gray scale.
  - If the height is in the integer range [0, 255], the lowest height 0 is colored in black, and the highest 255 is colored in white.

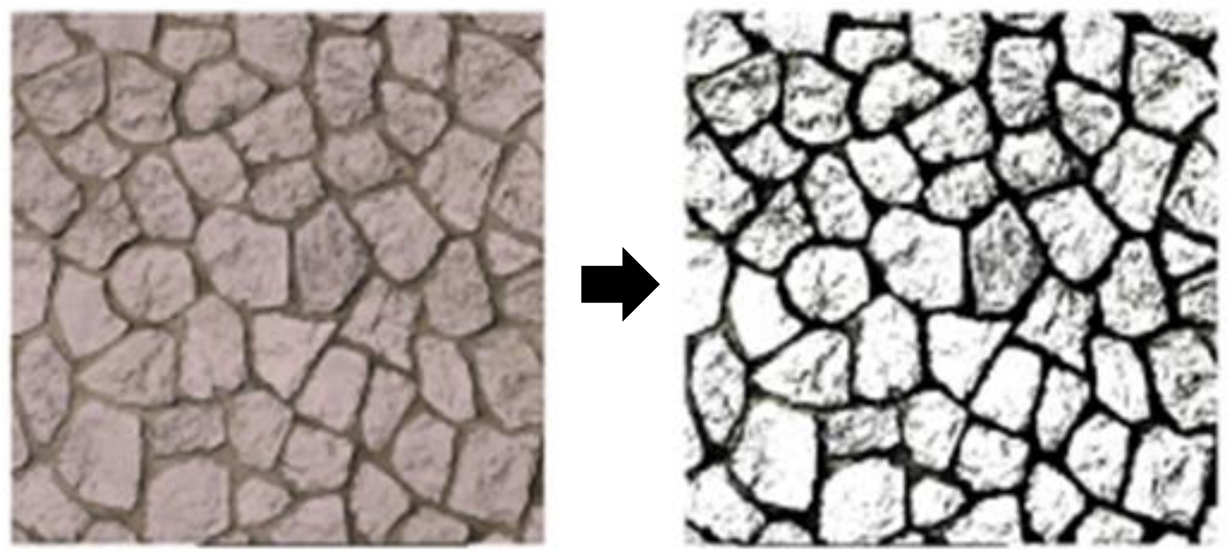
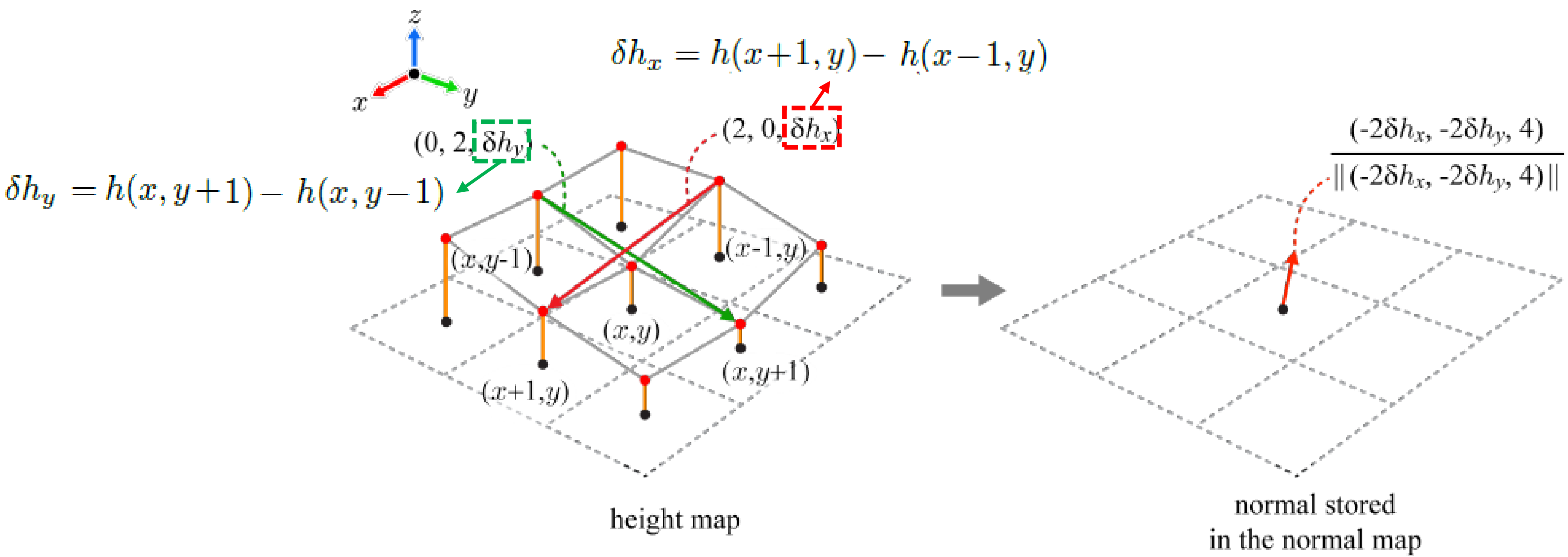


image to height map

Normal Mapping Generation

Height map to normal map

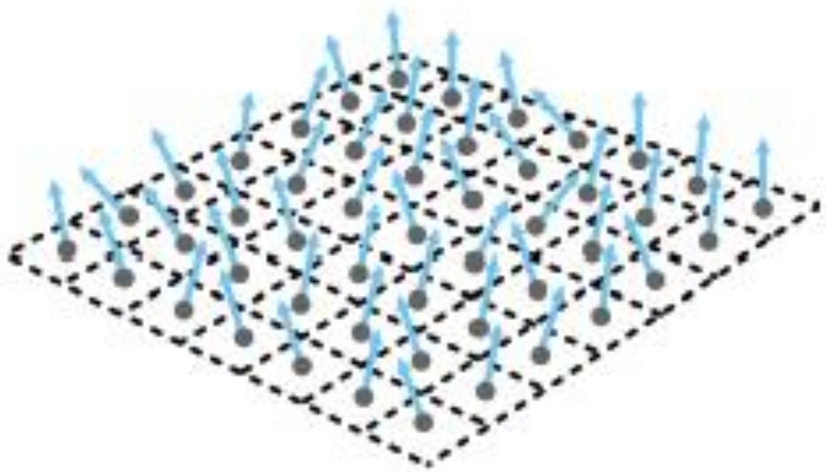
- With a height map, we can create a normal map.
- The normal at (x, y, h(x, y)), where h(x, y) represents the height at (x, y), can be determined by using the heights of its neighbors (cross product of red and green vectors in the figure).



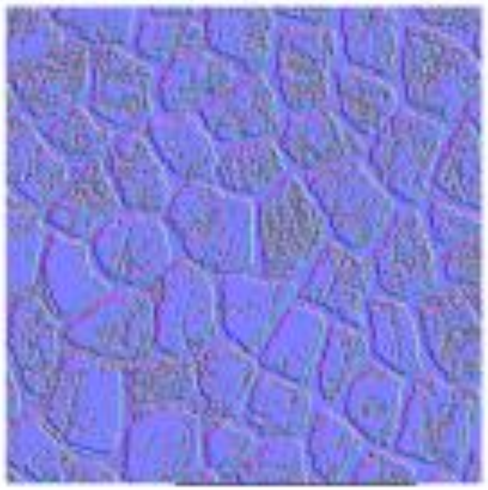
Normal Mapping Generation

Normal map visualization

- Each component of a normal can be obtained a floating-point (n_x, n_y, n_z) in the range of [-1, 1].
- In order to store the normal in a texture, where each RGB component is in the range of [0, 1], we need a range conversion.



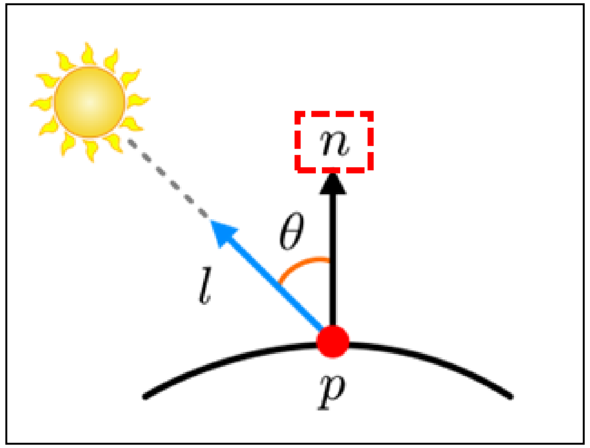
$$R = (n_x + 1)/2$$
$$G = (n_y + 1)/2$$
$$B = (n_z + 1)/2$$

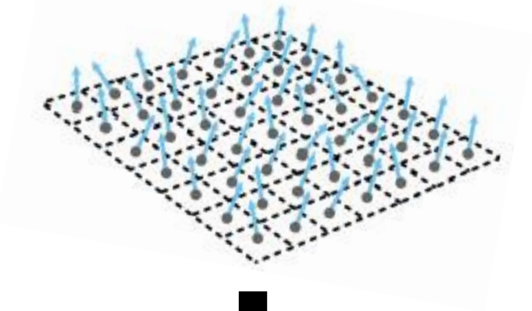


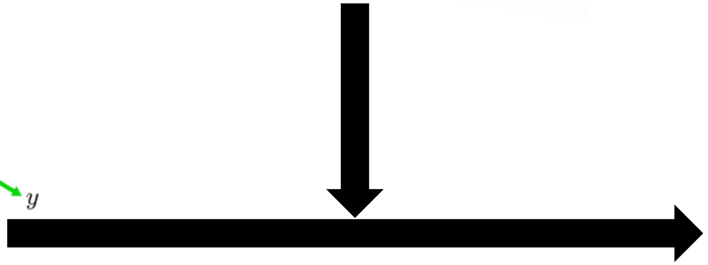
Normal Mapping

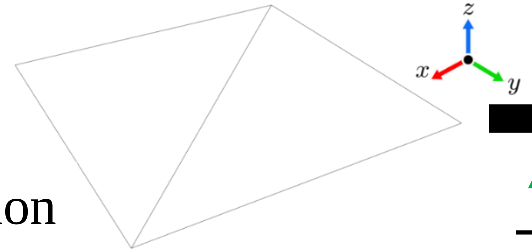
How to use normal map?

- The polygon mesh is rasterized and texture coordinates (s, t) are used to access the normal map.
- The normal at (s, t) is obtained by filtering the normal map.
- Consider the diffuse reflection term, $max(n \cdot l, 0)s_{\mathrm{d}} \otimes m_d$.
- The normal n is fetched from the normal map.
- m_d is fetched from the image texture.

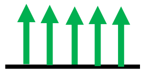




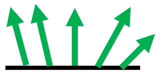




low-resolution
surface
(base surface)

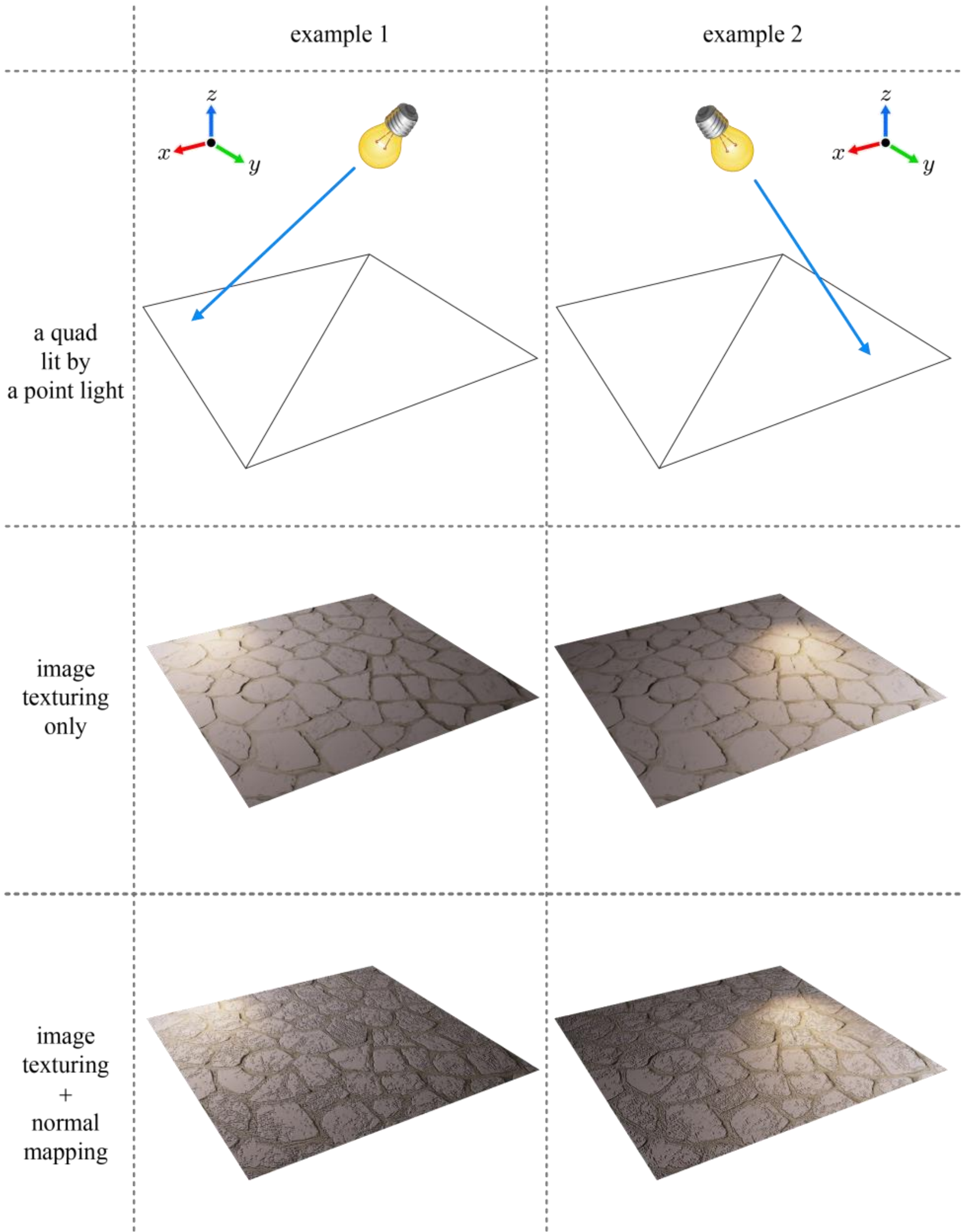


normal from
the base surface



normal from
the normal map

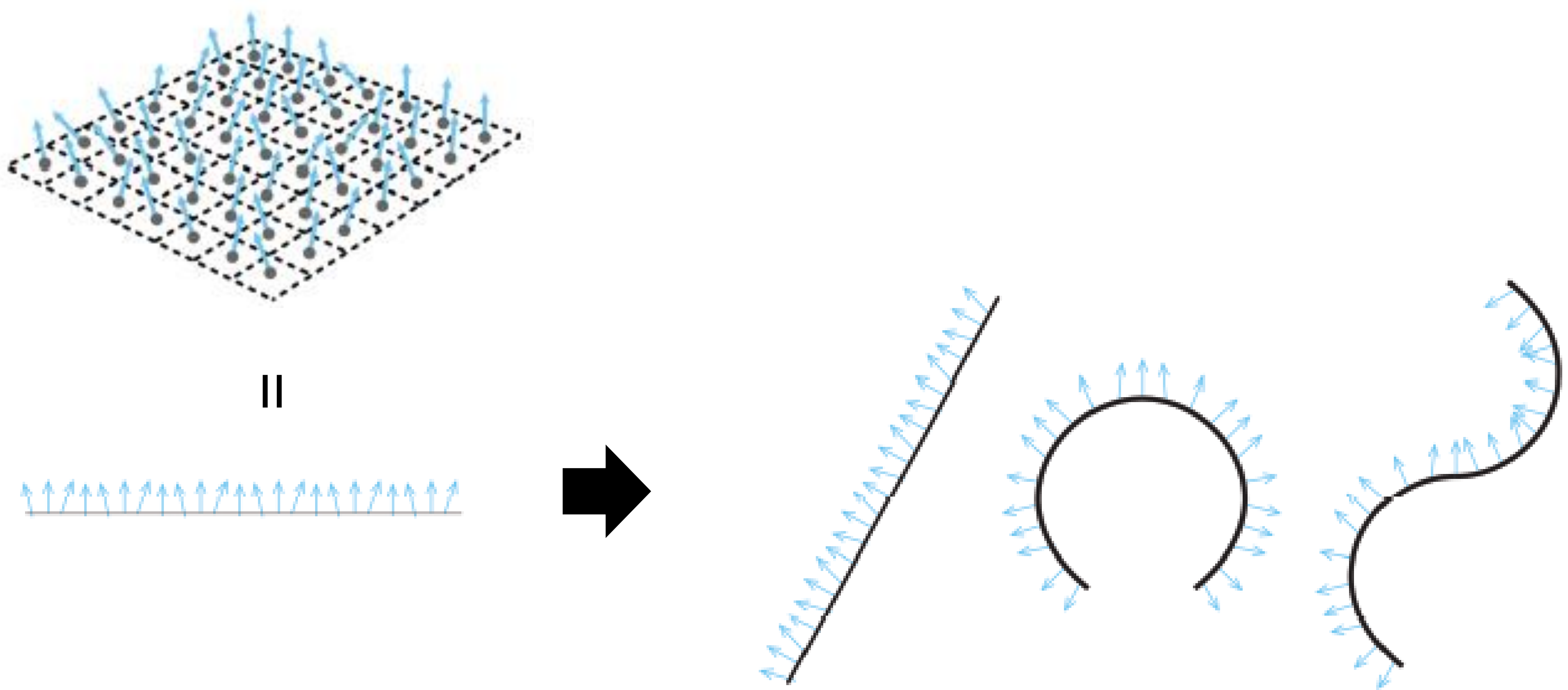
normal-mapped
surface


example 1
example 2
a quad
lit by
a point light
z
x
y
image
texturing
only
image
texturing
+
normal
mapping

Tangent-space Normal Mapping

Normal mapping = texturing

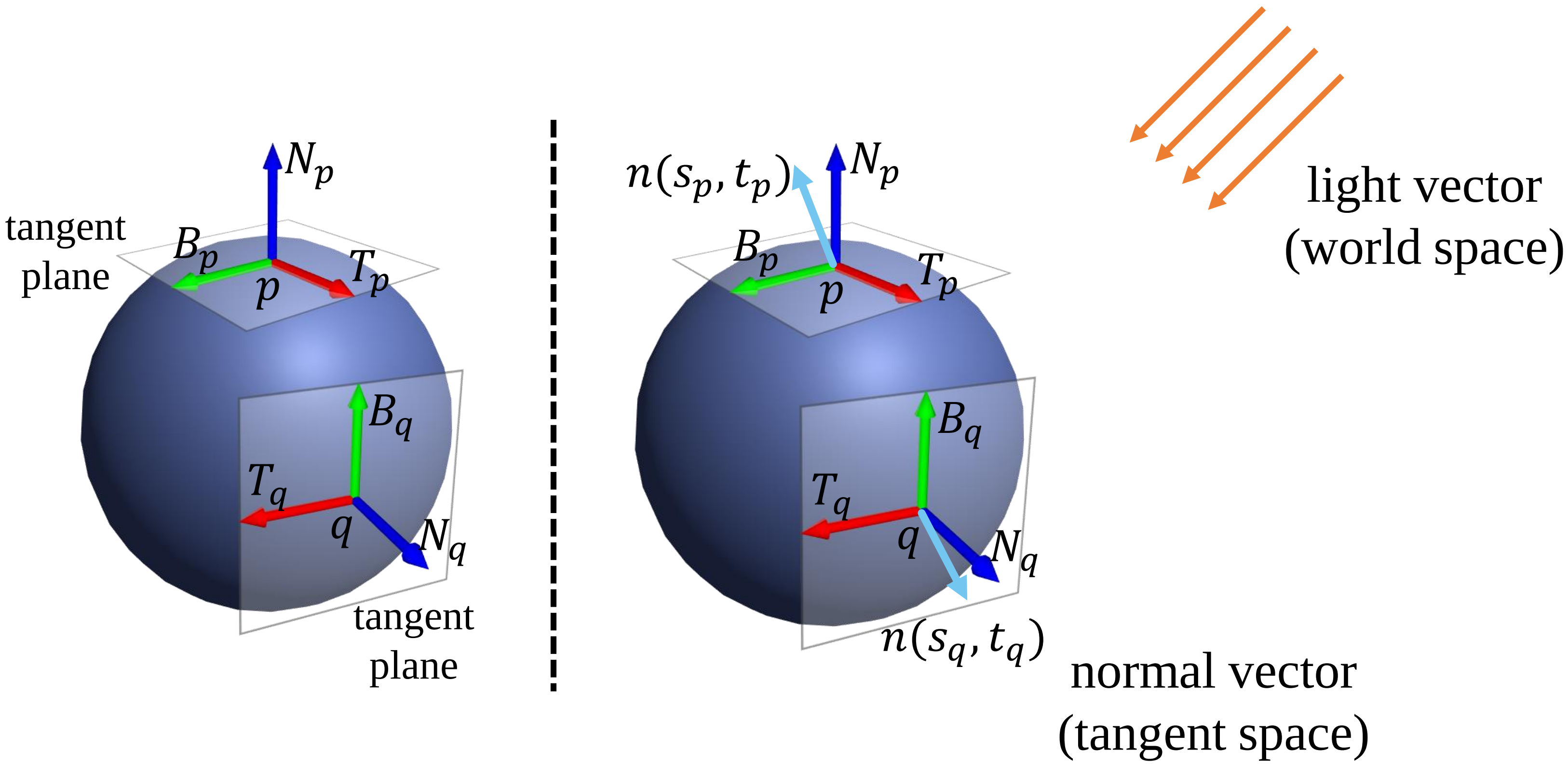
- Recall that texturing is described as wrapping a texture onto an object surface.
  - We should be able to paste it to various surfaces.
- In the same manner, we should be able to paste normal maps to various surfaces.



Tangent-space

Shaders for tangent-space normal mapping

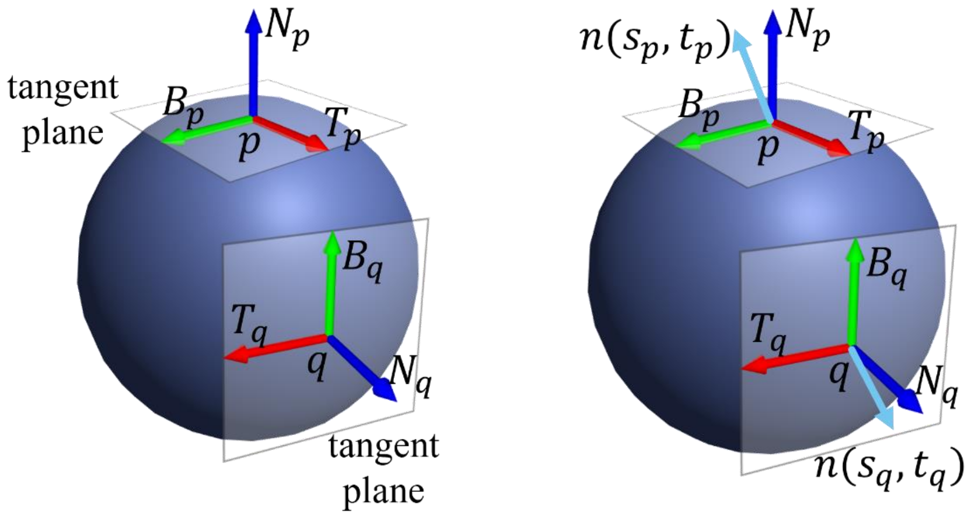
- In the previous slide, we invoked *dot*(*normal*, *light*) to achieve a normal mapping.
- However, it does not work in general because *normal* is a *tangent-space vector* but *light* is a *world-space vector*.



Tangent-space Normal Mapping

Tangent space

- For a surface point, consider a tangent space that is defined by three orthonormal vectors:
  - T (for tangent)
  - B (for bitangent)
  - N (for normal)



- The normal fetched from the normal map using q's texture coordinates, (s_q, t_q), is denoted as $n(s_q, t_q)$.
  - Without normal mapping, N_q would be used for lighting.
  - In normal mapping, however, $n(s_q,t_q)$ replaces N_q, which is (0, 0, 1) in the tangent space of q.

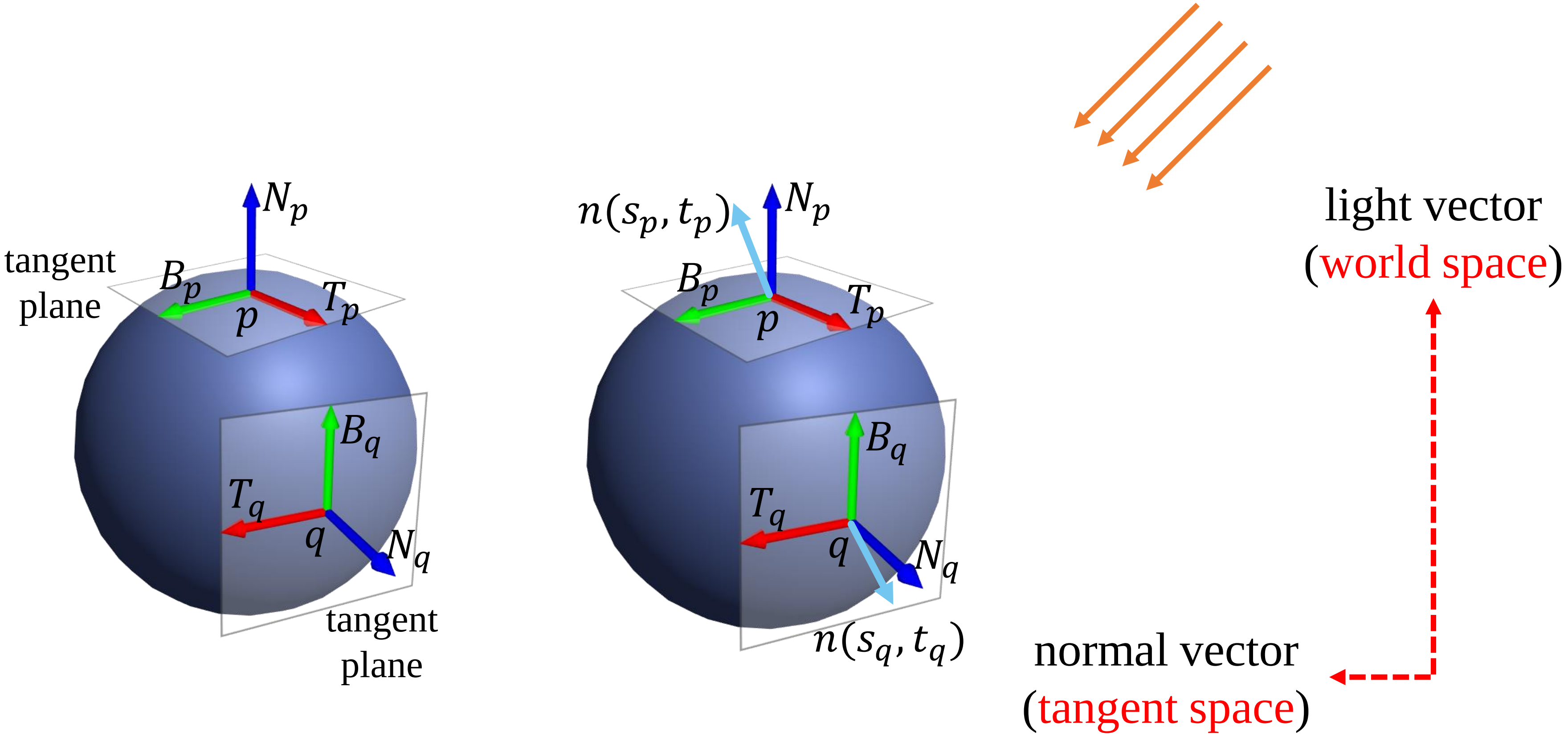
- Whatever surface point is normal-mapped, the normal fetched from the normal map is considered to be defined in the tangent space of that point.

Tangent-space

Two options to resolve the inconsistency between two vectors:

- Transform normal into the world space.
- Transform light into the tangent space.

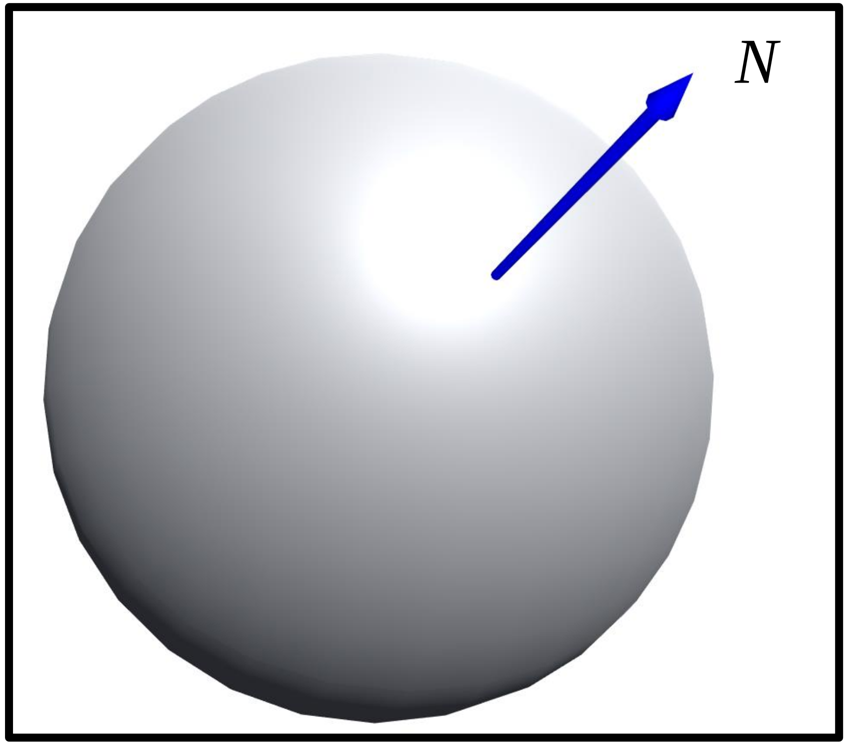
We will take the second option.



Tangent-space

The basis of tangent space $\{T, B, N\}$

- Vertex normal N – defined per vertex at the modeling stage.
- Tangent T – needs to be computed
- Bitangent B – needs to be computed

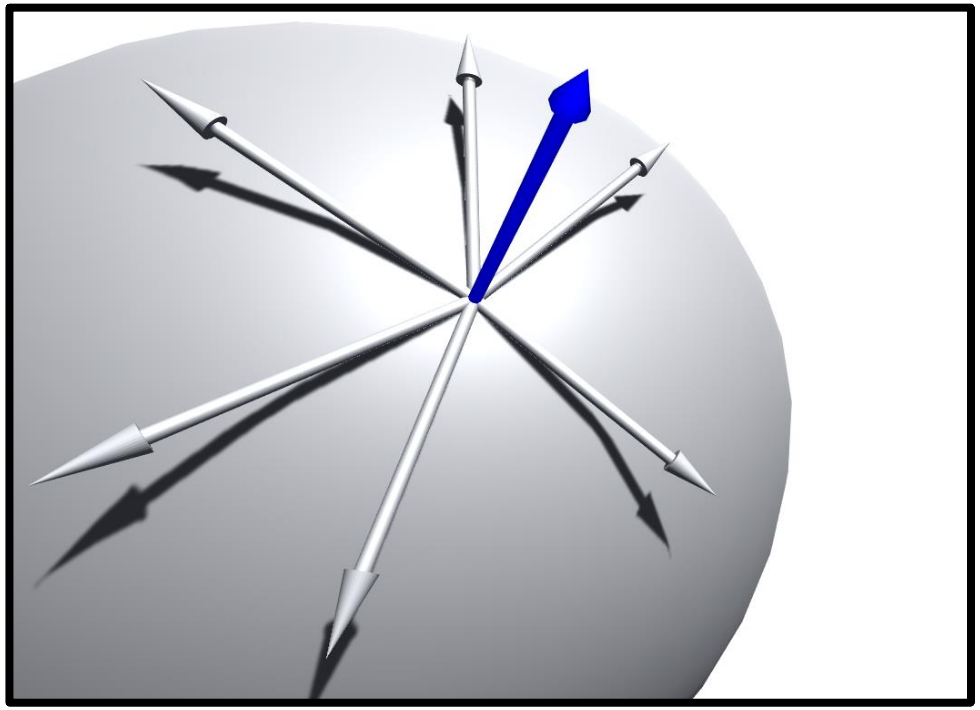


Tangent-space

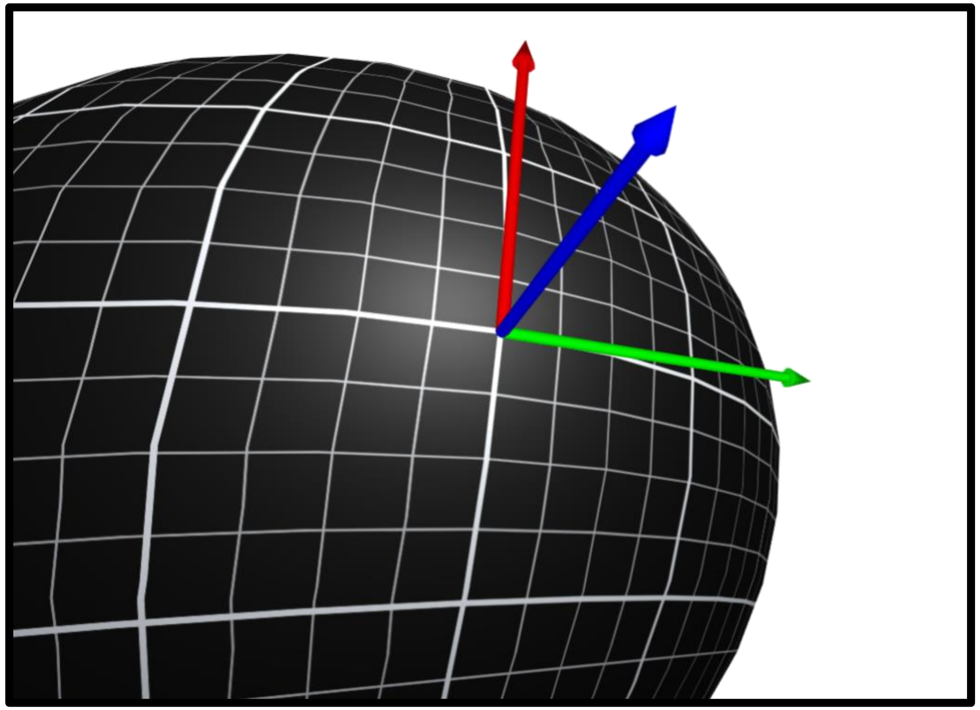
The basis of tangent space $\{T, B, N\}$

- Next we need a tangent, T: a vector parallel to the surface.
  - But there are many such vectors.

- The standard method is to orient the tangent in the same direction that the texture coordinates.



there are many vectors that are parallel to the surface



texture coordinate basis = T and B vectors

Tangent-space

The basis of tangent space $\{T, B, N\}$

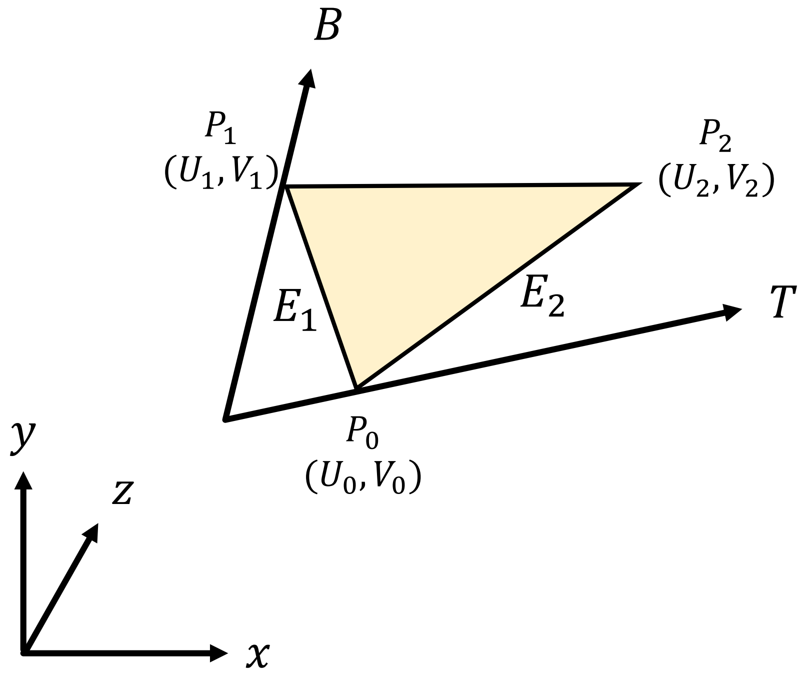
- Let's take a look at the triangle with three vertices at positions P_0, P_1, and P_2 and texture coordinates (U_0, V_0), (U_1, V_1), and (U_2, V_2).

- The two edges E_1 and E_2 can be written as a linear combination of T and B:
  - $E_1 = (U_1 - U_0)T + (V_1 - V_0)B$

- This can also be written as
  - $(E_{1x}, E_{1y}, E_{1z}) = \Delta U_1(T_x, T_y, T_z) + \Delta V_1(B_x, B_y, B_z)$
  - $(E_{2x}, E_{2y}, E_{2z}) = \Delta U_2(T_x, T_y, T_z) + \Delta V_2(B_x, B_y, B_z)$

- Now, we can make matrix:
  - $\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$

- Then we can calculate T and B:
  - $\begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$



Tangent-space Normal Mapping

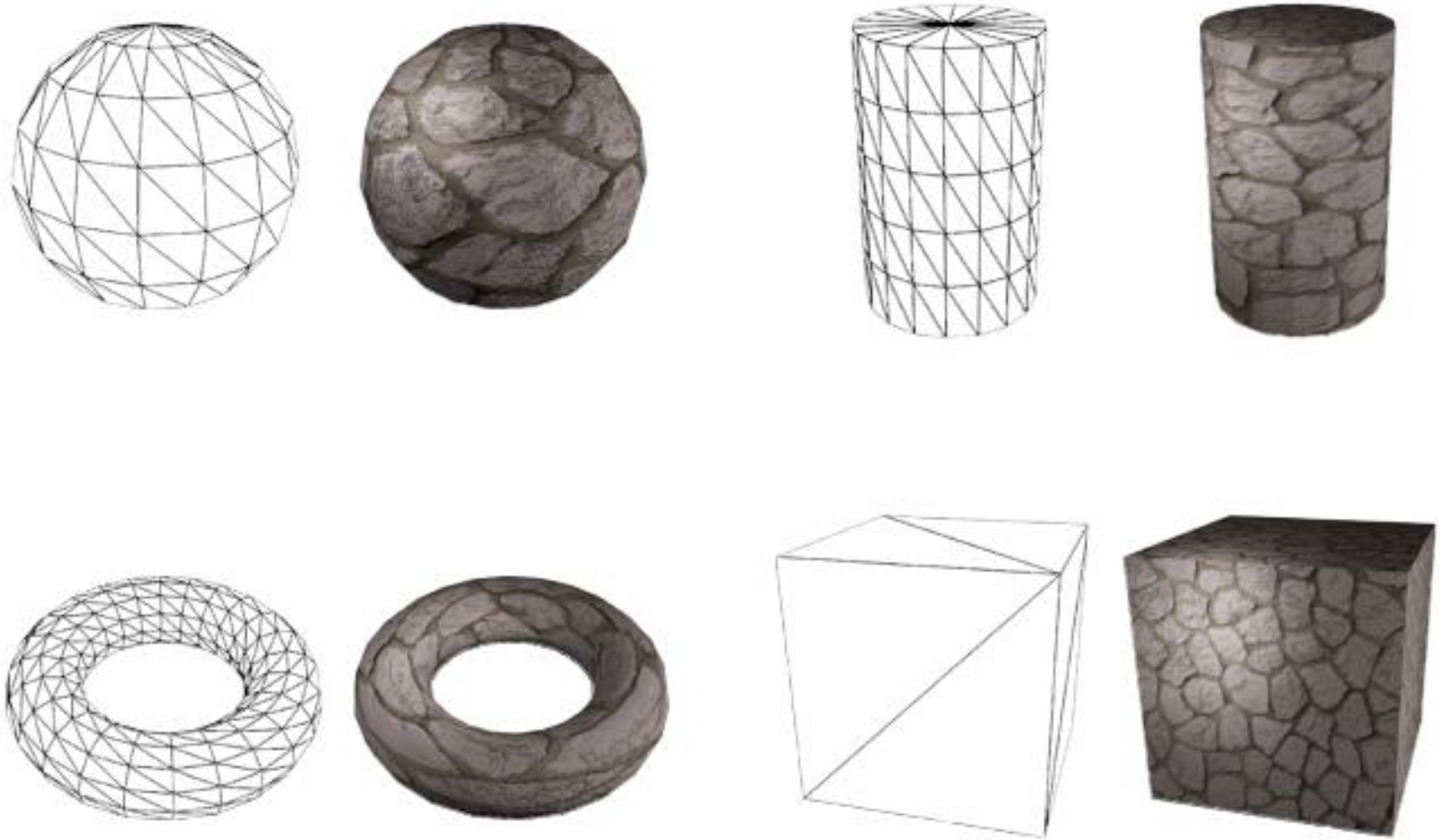
Consider the diffuse term of the Phong lighting model.

$$\boxed{max(n \cdot l, 0)s_d \otimes m_d} + (max(r \cdot v, 0))^{sh} s_s \otimes m_s + s_a \otimes m_a + m_e$$

- A light source is defined in the world space, and so is *l*.
- In contrast, *n* fetched from the normal map is defined in the tangent space.
- To resolve this inconsistency, *n* has to be transformed into the world space, or *l* has to be transformed into the tangent space.
- Typically, the per-vertex *TBN*-basis is pre-computed, is stored in the vertex array and is passed to the vertex shader.
- The vertex shader first transforms *T*, *B*, and *N* into the world space and then constructs a matrix with the world-space *T*, *B*, and *N*.
- It rotates the world-space light vector into the per-vertex tangent space.

Tangent-space Normal Mapping

Results



Tangent-space Normal Mapping

Results

Tangent-space Normal Mapping

Comparison

wireframe | rendering w/o normal mapping | rendering with normal mapping

original mesh
6,895 triangles

simplified mesh
and normal mapping
689 triangles

Normal Mapping Discussion

Depending on the use case, normal maps can be implemented in tangent or world spaces.

- Normal mapping discussed in our lecture is tangent normal map.
  - Advantages are as follows:
    - This is object-independent normal map, which means they can be reused across different objects, as they don't rely on the object's orientation or position in the world.
    - This is useful when creating asset libraries, or when different instances of an object appear in different orientations.
    - Furthermore, tangent space normal maps work well with deforming surfaces, such as a character's skin, because the normals are calculated relative to the surface of the object itself, and not relative to the world.

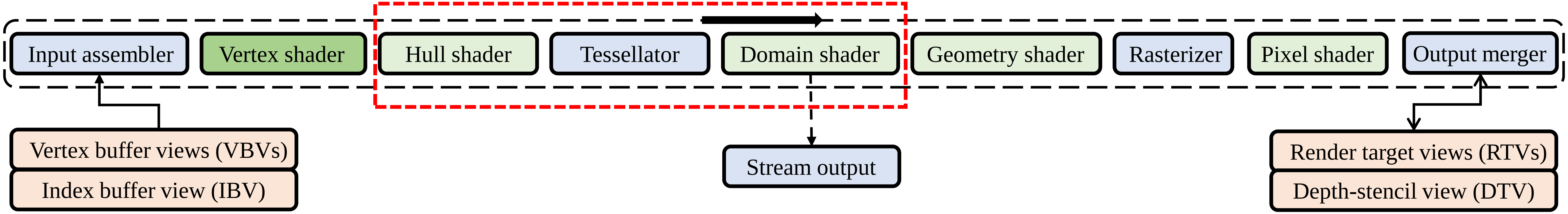
  - Disadvantages are as follows:
    - However, they need to calculate and maintain a consistent tangent space and add complexity to both the model creation process and to the shading computation.
    - They have issues with seams where UV coordinates split or where geometry is mirrored.

Hardware Tessellation

Hardware tessellation

- Hardware tessellation enables the GPU to decompose a primitive into a large number of smaller ones.

- GPU tessellation involves two new programmable stages and a new hard-wired stage.
  - The Hull Shader and Domain Shader.
  - The tessellation primitive generator called tessellator.



Displacement Mapping

Displacement Mapping

- In normal mapping, the underlying geometry of the base surface is not altered.

- The combination of displacement mapping and tessellation technique can resolve the problem.
  - Tessellation hardware tessellates the base surface first.
  - Then, tessellated vertices are displaced along the displacement vector.

base
model

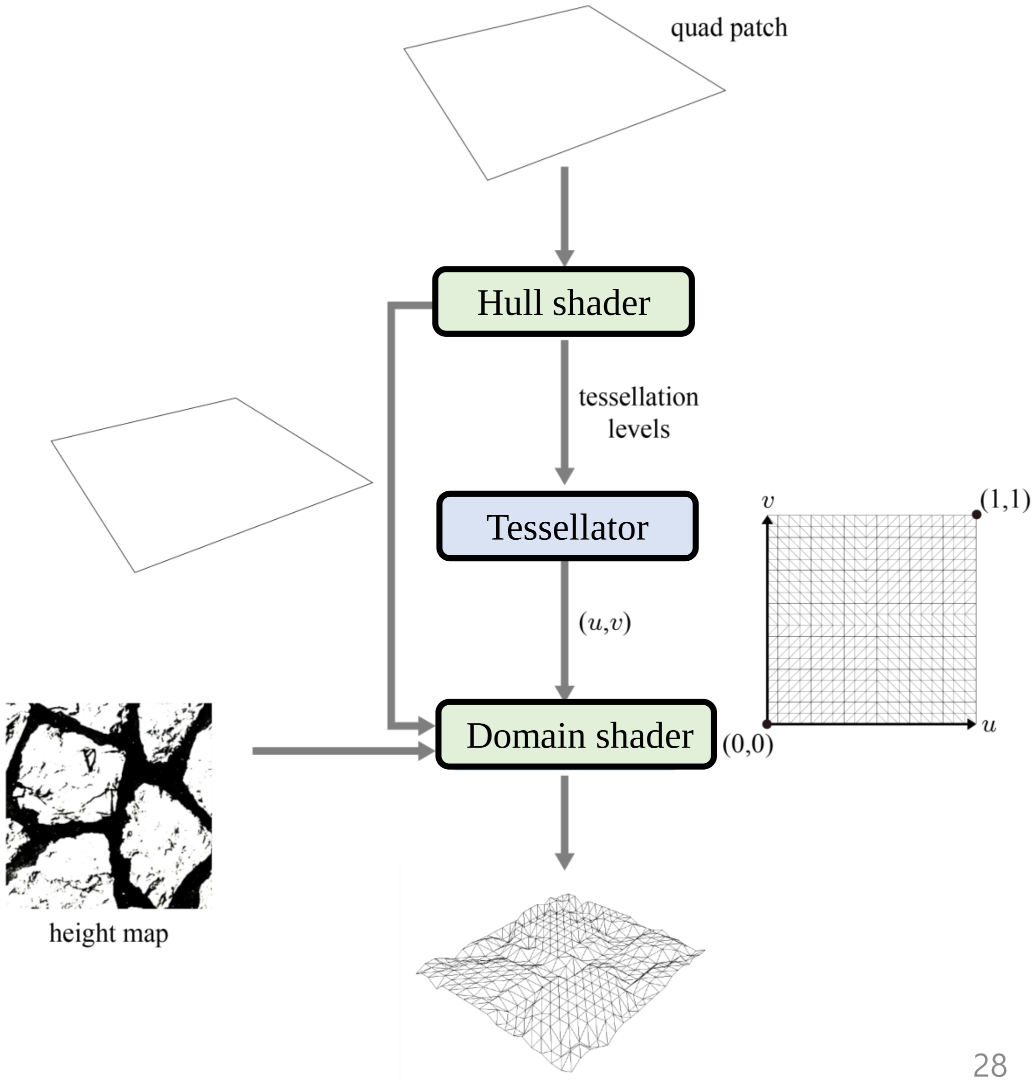
normal
mapping

displacement
mapping

Displacement Mapping

Displacement Mapping

- The input is called a *patch* or *base surface*. It is either a triangle or a quad.
- For the paved-ground example, the Hull shader takes a quad as the base surface and passes it to the Domain shader.
- The Hull shader determines the *tessellation levels* and passes them to the tessellator, which accordingly tessellates the domain of the quad into a 2D triangle mesh.
- Running once for each vertex of the 2D mesh, the Domain shader takes the quad as a bilinear patch, evaluates a point using (u, v), and displaces it using the height map.



Vertex Shader and Hull Shader

VS & HS

- Vertex shader no longer handles the space change.
  - In the previous lecture, the major roll of vertex shader is to compute positions of vertices.
  - When implementing tessellation, the clip-space vertex position will be computed by the Domain shader instead of the vertex shader.

- Hull shader declares the state required by the tessellator.
  - The state includes information such as the number of control points, the type of patch face and the type of partitioning to use when tessellating.
  - Tessellation factors are also determined in this stage.

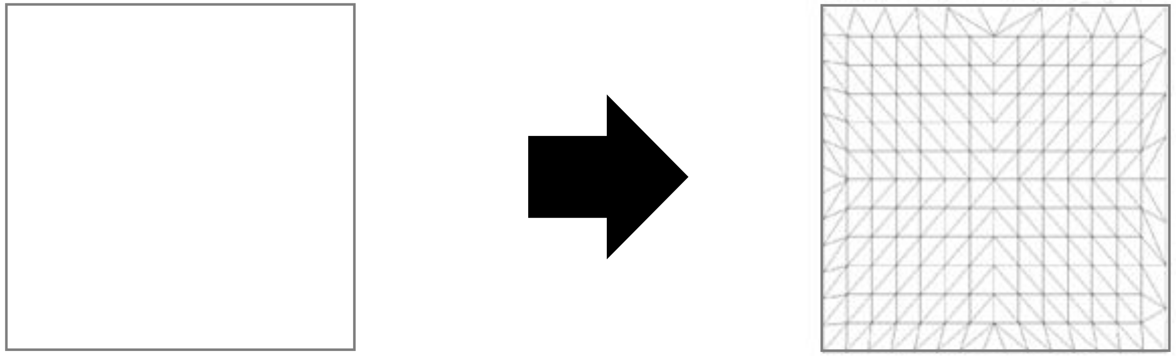
Tessellator

Tessellator

- Tessellator (or primitive generator) subdivides a patch and generates small generics represented by barycentric coordinates.

- Tessellator is similar to the vertex shader since it always has a single input (the barycentric coordinate) and a single output (the vertex).

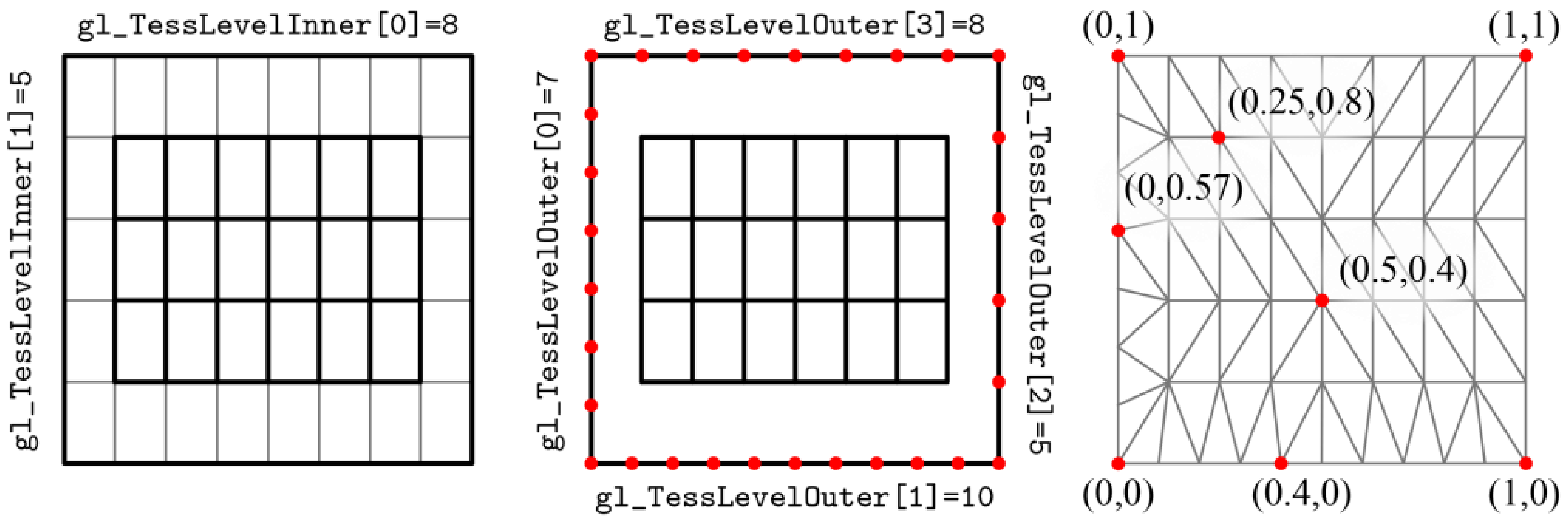
- Tessellator cannot generate more than one vertex per invocation nor drop the vertex.



Tessellator

Tessellator

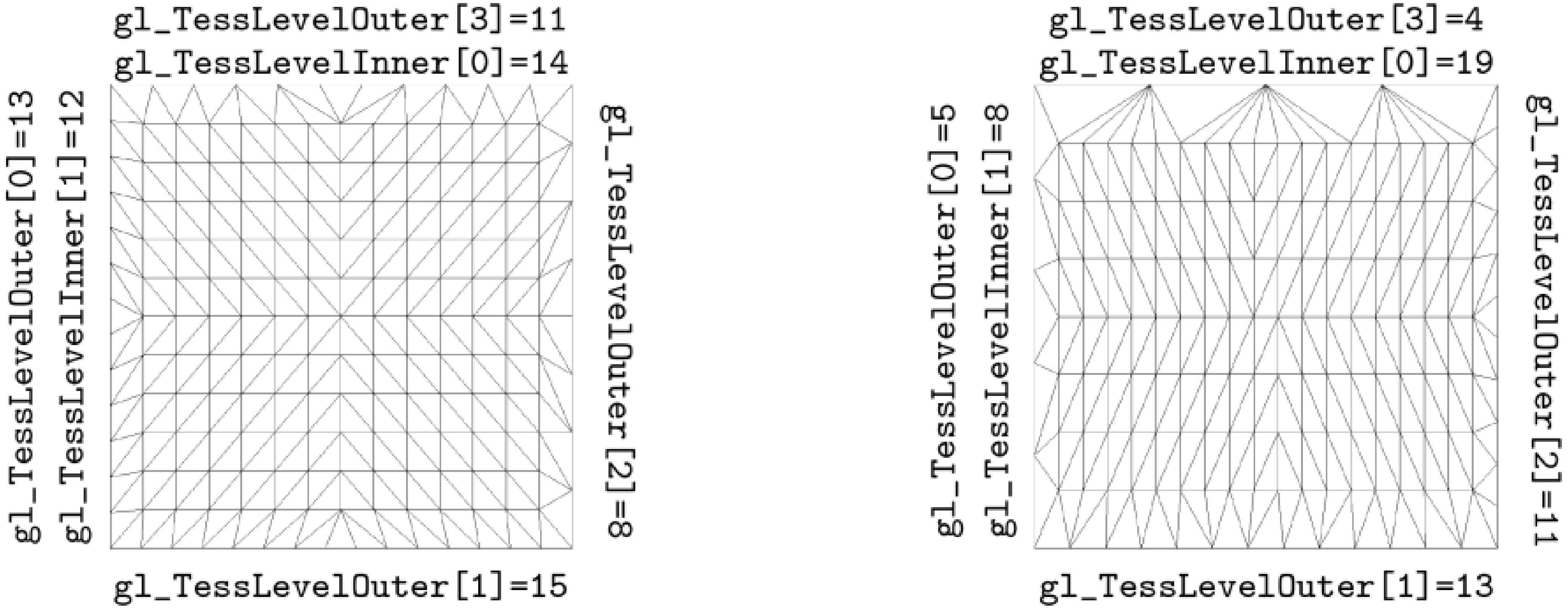
- Inner and outer tessellation levels.



Tessellator

Tessellator

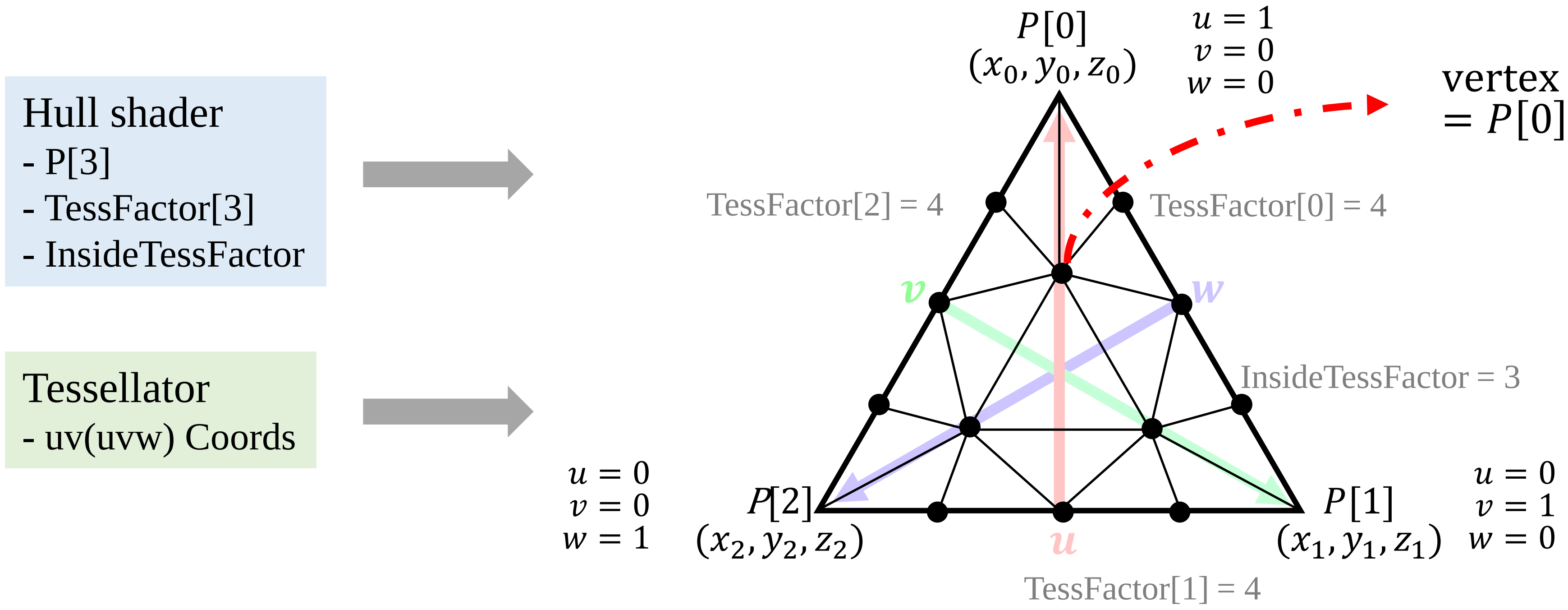
- Inner and outer tessellation levels.



Domain Shader

Domain shader

- The Domain shader takes vertex positions passed from the Hull shader, as the control points of a bilinear patch.
- Using (*u*, *v*, *w*), the patch is evaluated to return a 3D point.
- The texture coordinates are bilinearly interpolated in the same manner.



vertex position
$= P[0] \times u + P[1] \times v + P[2] \times w$

Displacement Mapping

Results

- Figure on the right shows a large paved ground generated with displacement mapping.
- The base surface is composed of 16 quads, (a).
- A quad is tessellated into 722 triangles, (b).
- Using a height map, the vertices of the tessellated mesh are vertically displaced, (c).
- The high-frequency mesh is shaded, (d).