Texture Mapping

Bo Zhu
School of Interactive Computing
Georgia Institute of Technology

Motivational Video

- Las Vegas MSG Sphere

Motivational Video: Hydrographic Printing

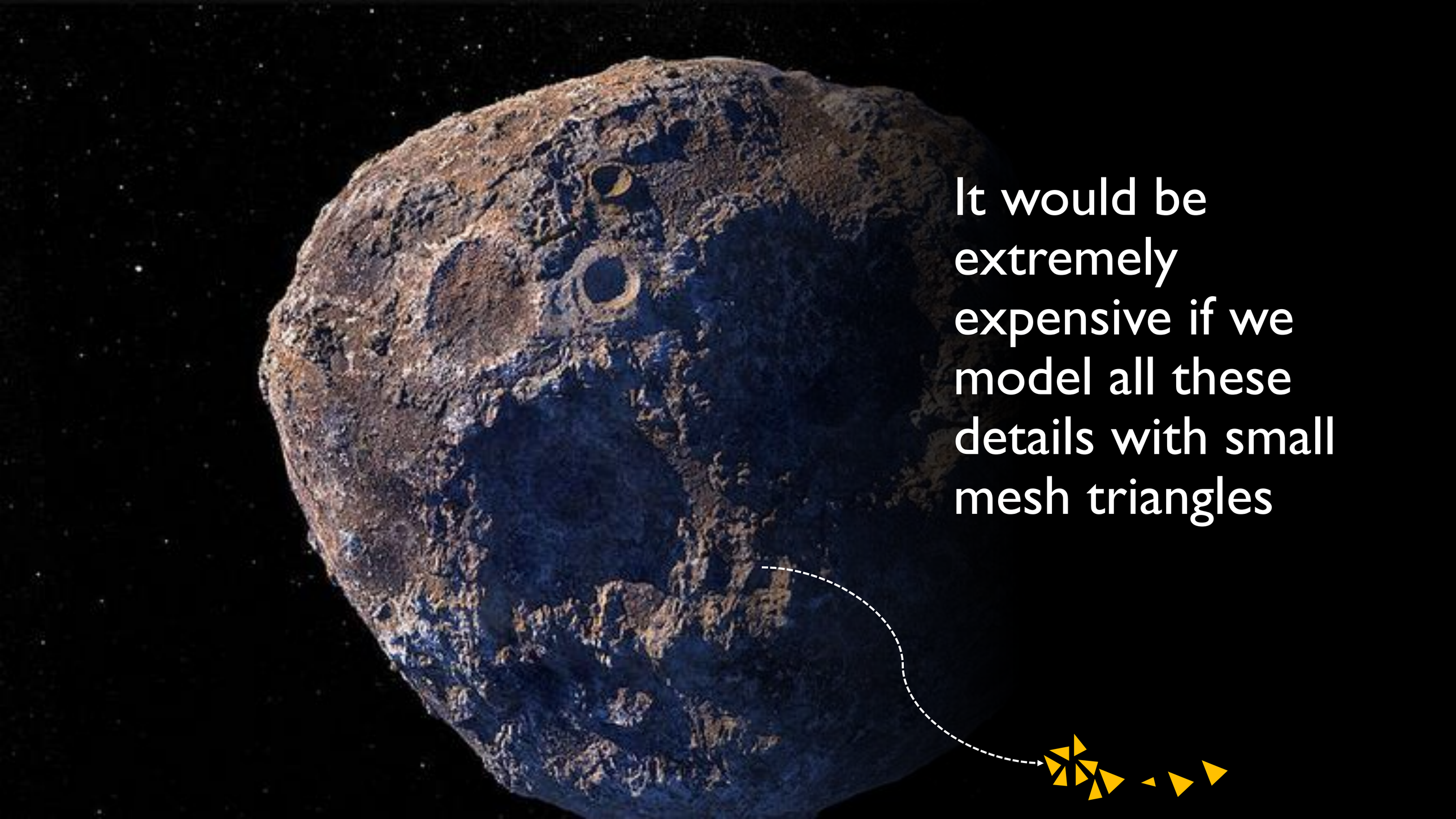



What is texture?

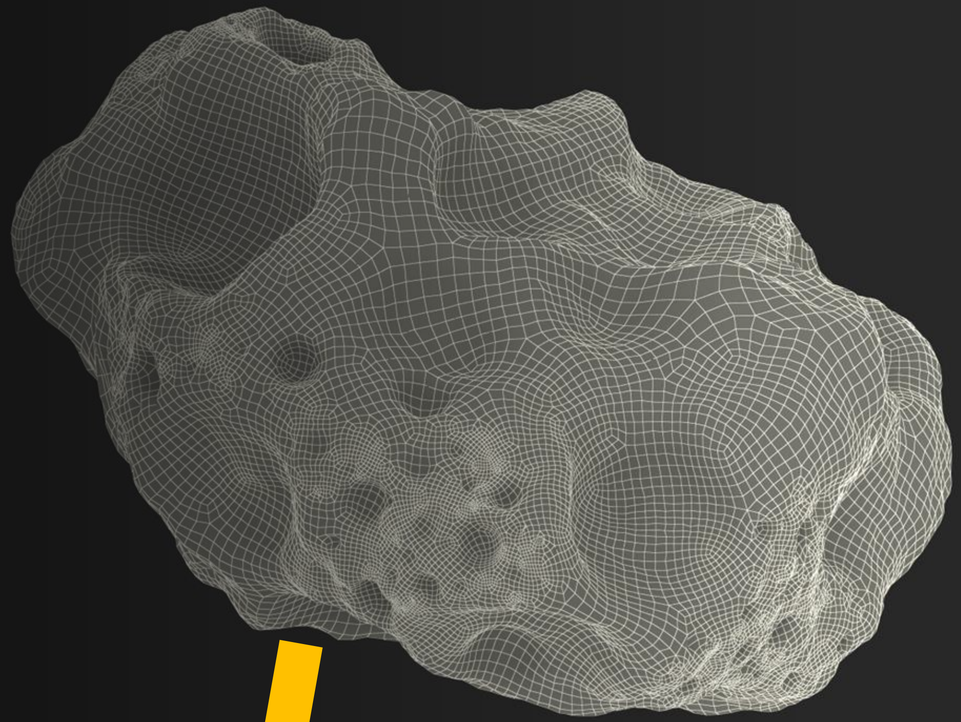
- Spatially varying details on the object surface


Texture is not just about color, but also small bumps, dents, scratches, displacements …

It is about all kinds of small visual details on the surface..



It would be extremely expensive if we model all these details with small mesh triangles



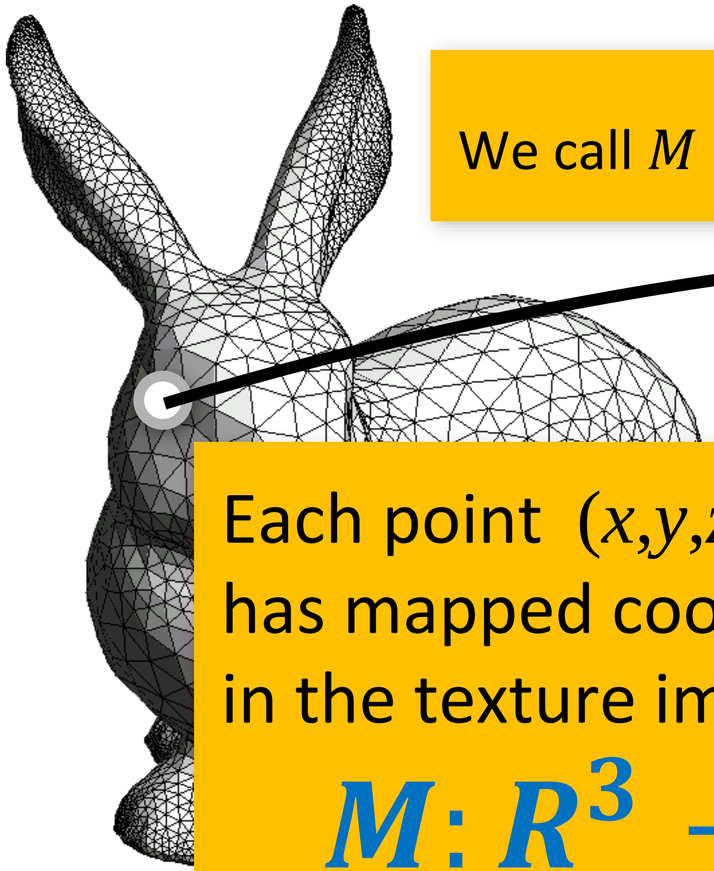
Key Idea of Texture Mapping

Store surface details in an image, and map the image to surface

https://www.turbosquid.com/3d-models/meteorite-stone-3d-model-2071245

How does texture mapping work?

- Step I: create a mapping between the surface and the image

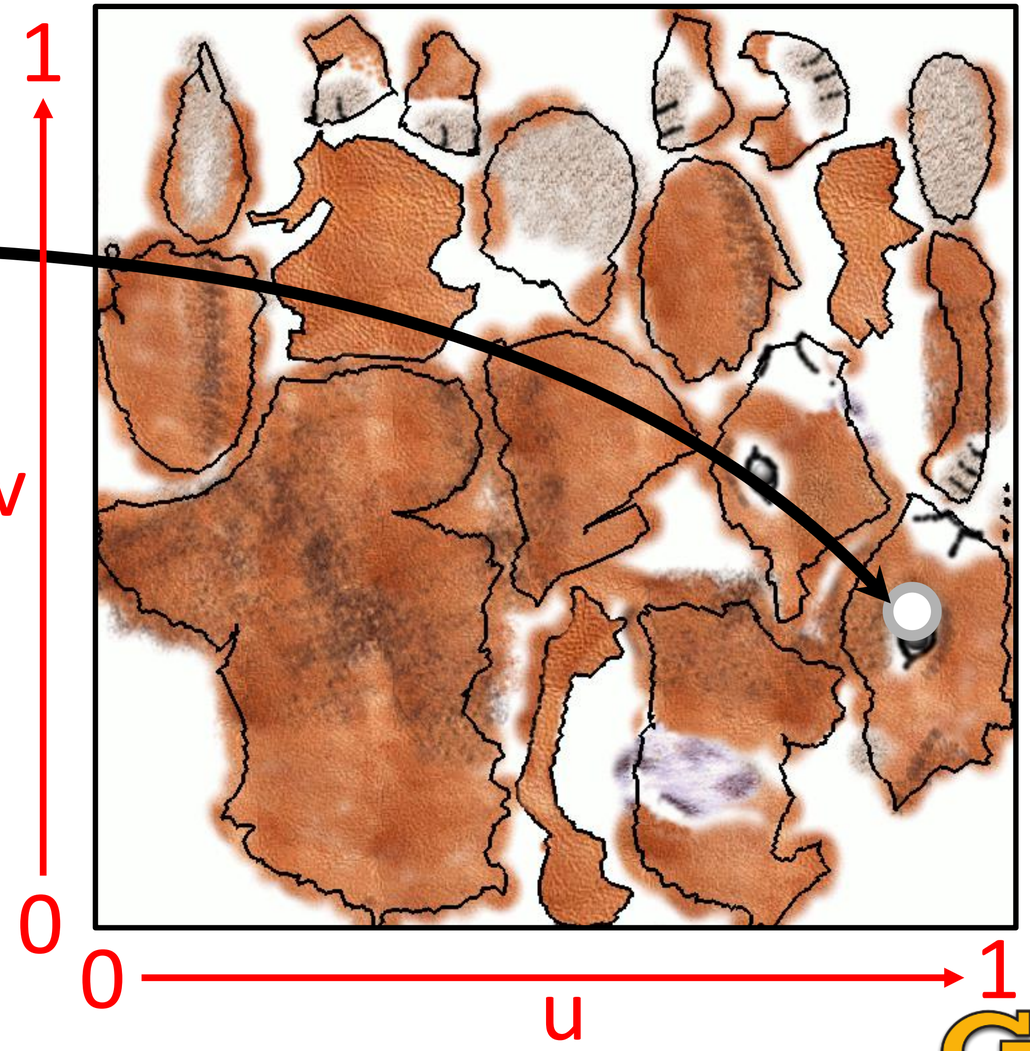


We call M **mapping function**

Each point (x,y,z) on the surface has mapped coordinates (u,v) in the texture image:

$$M: R^3 \rightarrow [0,1]^2$$

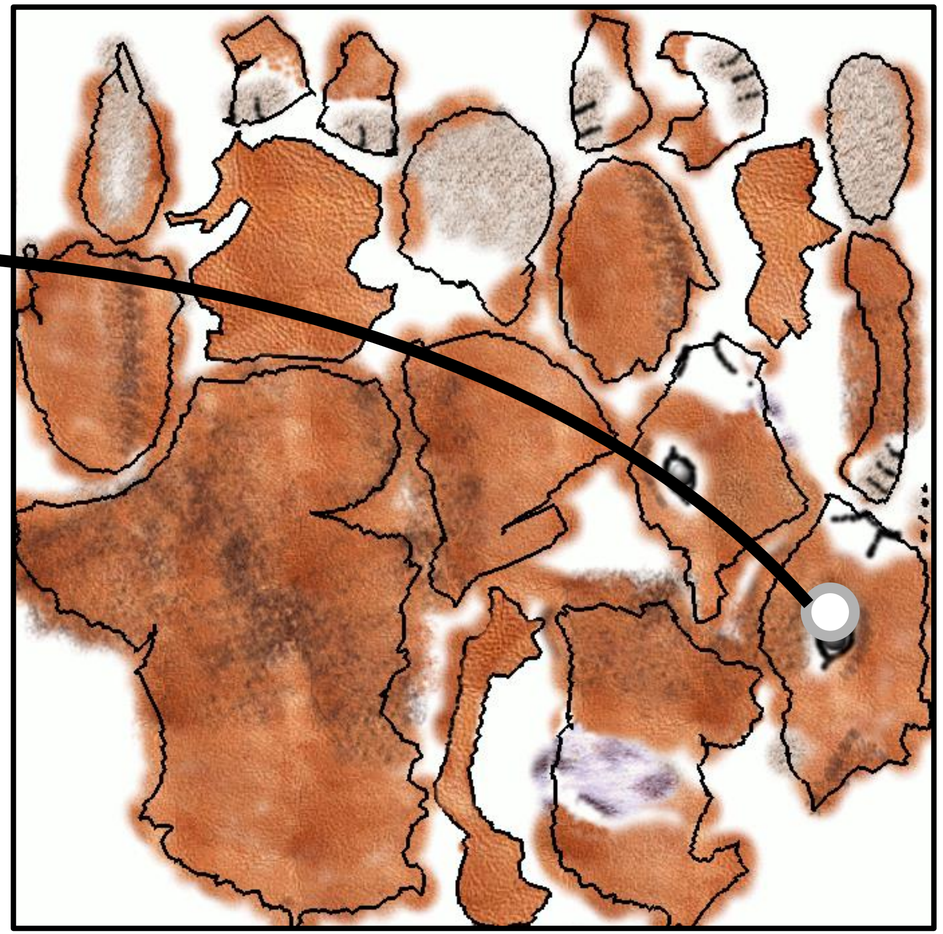
$$M(x,y,z) = (u,v)$$



How does texture mapping work?

- Step II: read the color from the image and assign it to the surface

We call T **texture function**



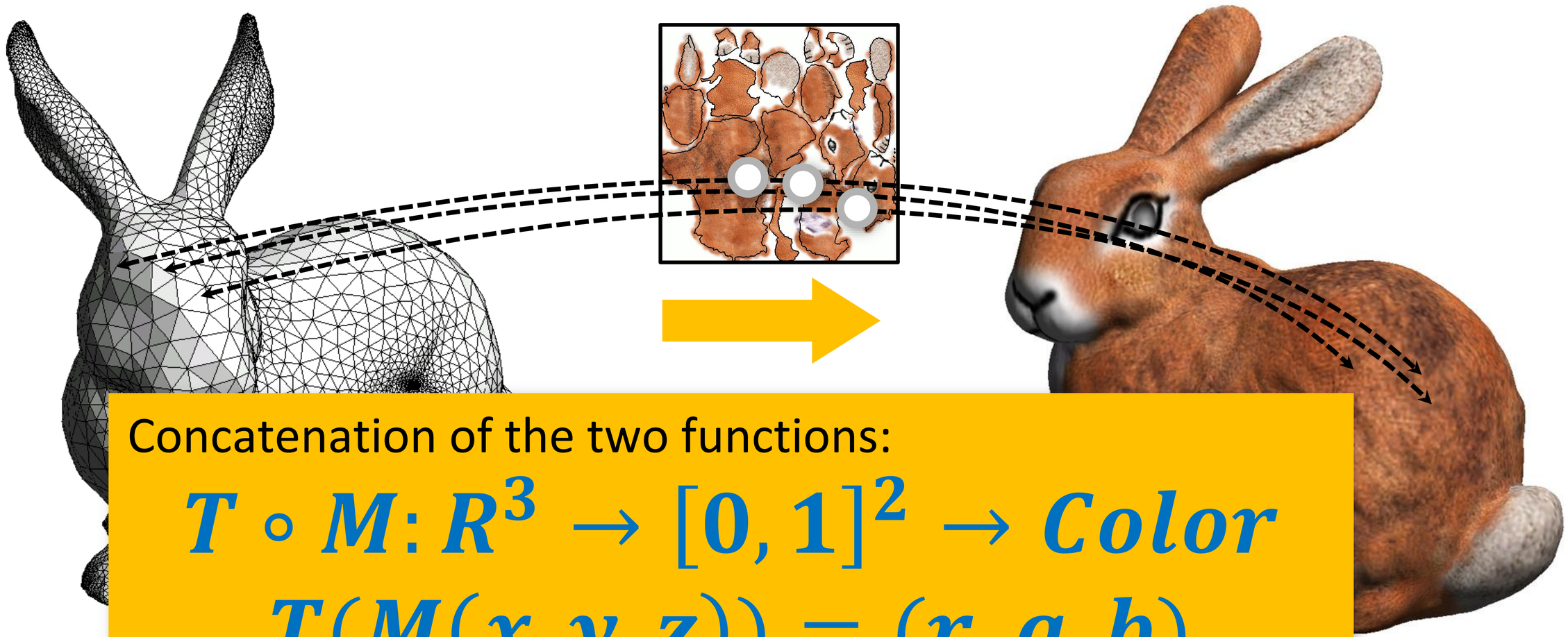
Texture itself is a function:

$$T: [0,1]^2 \rightarrow Color$$

$$T(u,v) = (r,g,b)$$

Put these two steps together

- Calculate color for every point on the object's surface by reading from a texture image



Concatenation of the two functions:

$$T \circ M : R^3 \rightarrow [0,1]^2 \rightarrow Color$$

$$T(M(x,y,z)) = (r,g,b)$$

Study Plan

Mapping Function

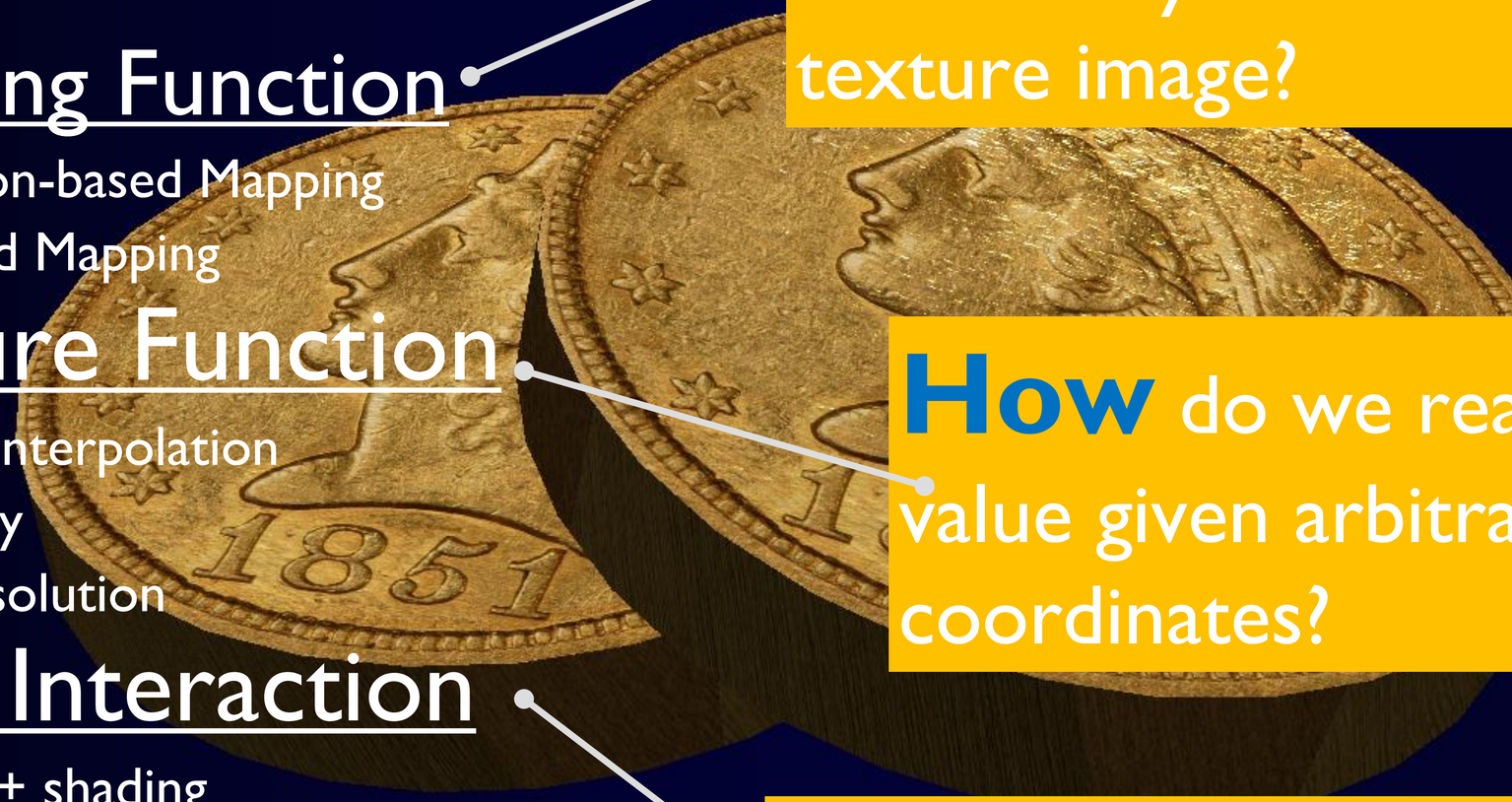
- Projection-based Mapping
- UV-based Mapping

Texture Function

- Bilinear interpolation
- Boundary
- Multi-resolution

Light Interaction

- Texture + shading
- Normal mapping



How do we map between an arbitrary surface and a texture image?

How do we read an image value given arbitrary uv coordinates?

What do we read from an image?

Study Plan

Mapping Function

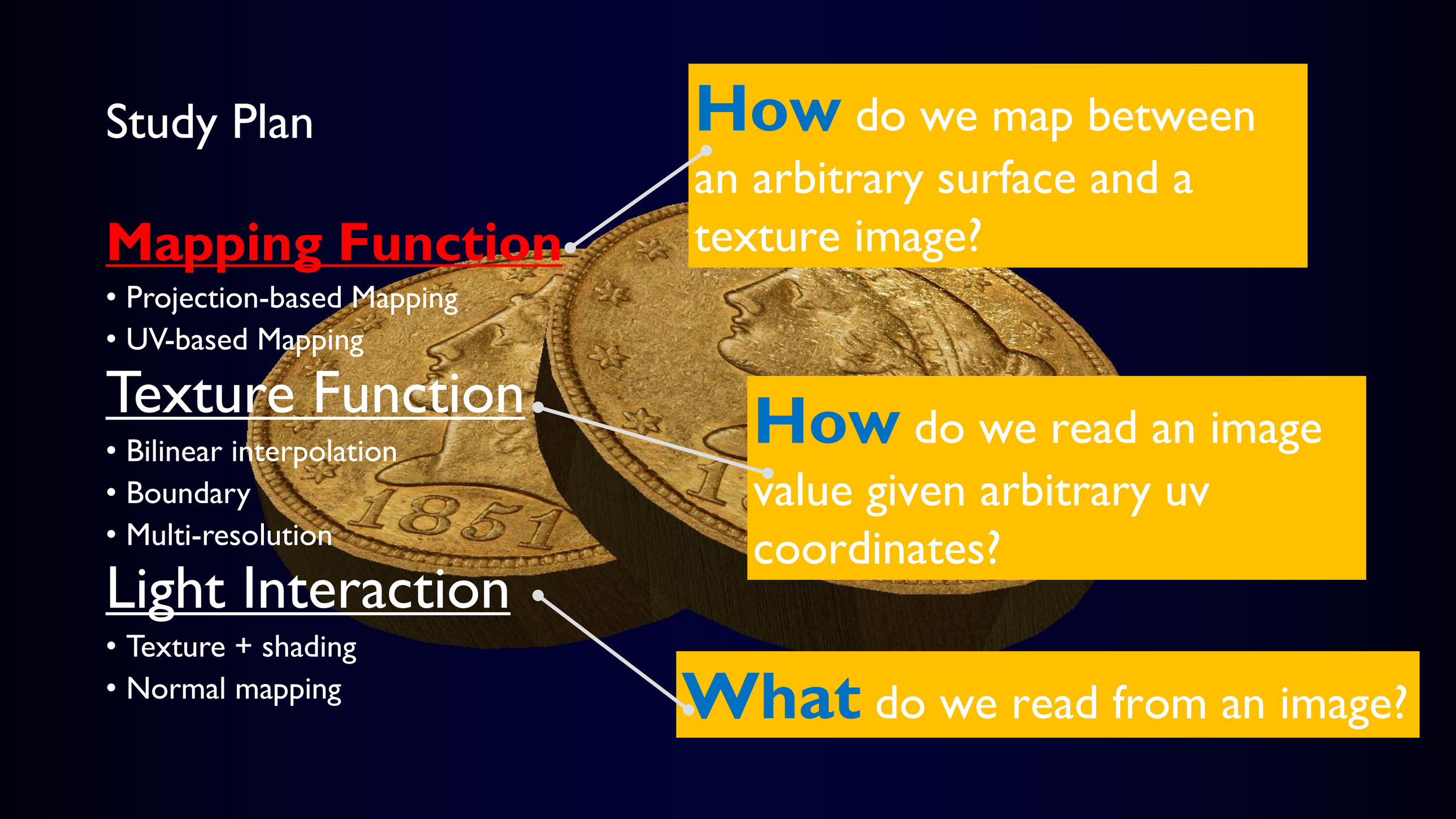
- Projection-based Mapping
- UV-based Mapping

Texture Function

- Bilinear interpolation
- Boundary
- Multi-resolution

Light Interaction

- Texture + shading
- Normal mapping

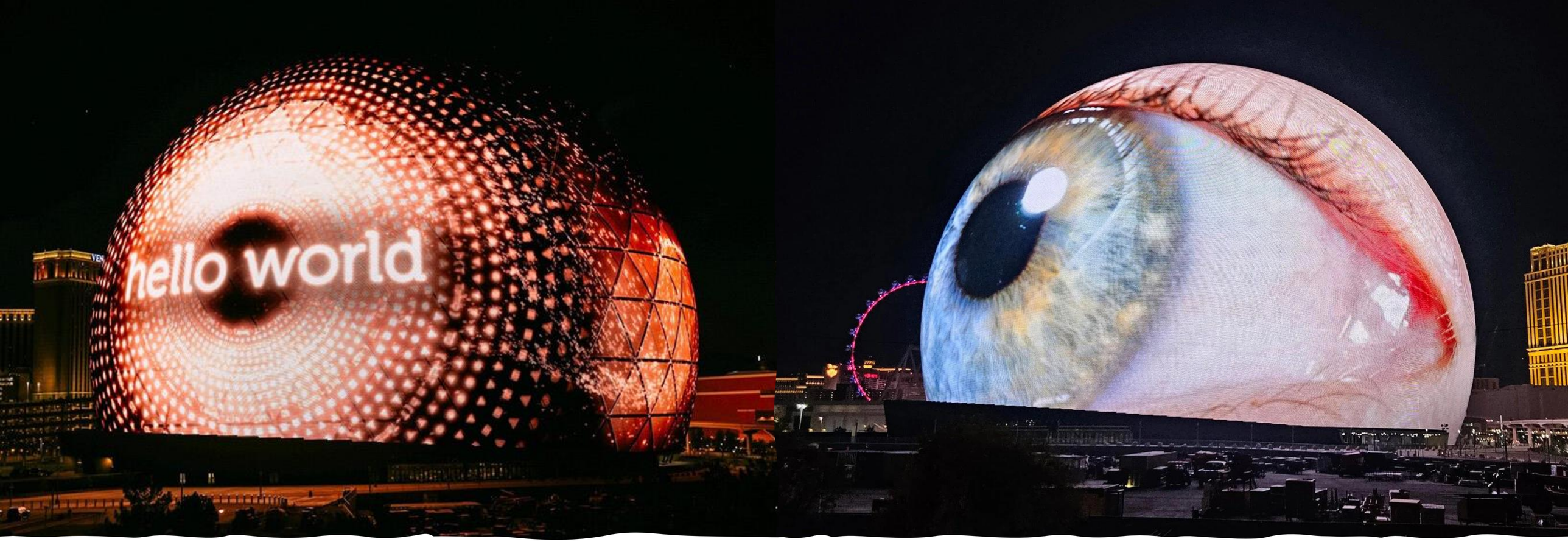


How do we map between an arbitrary surface and a texture image?

How do we read an image value given arbitrary uv coordinates?

What do we read from an image?

Projection-based Mapping

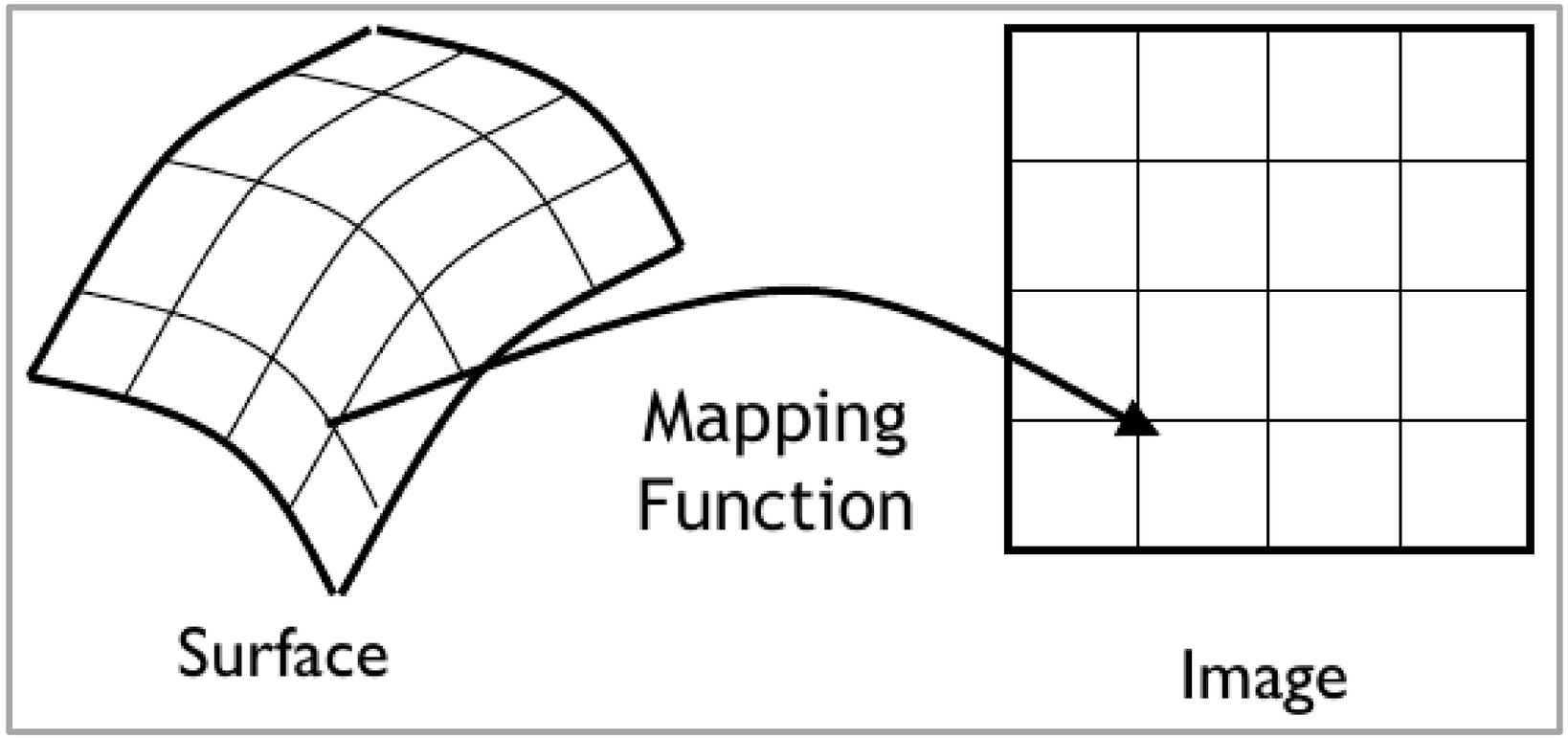


Geometric Intuition

Projection an image onto the surface of a sphere

Question 1: How to map a 3D point to a 2D point?

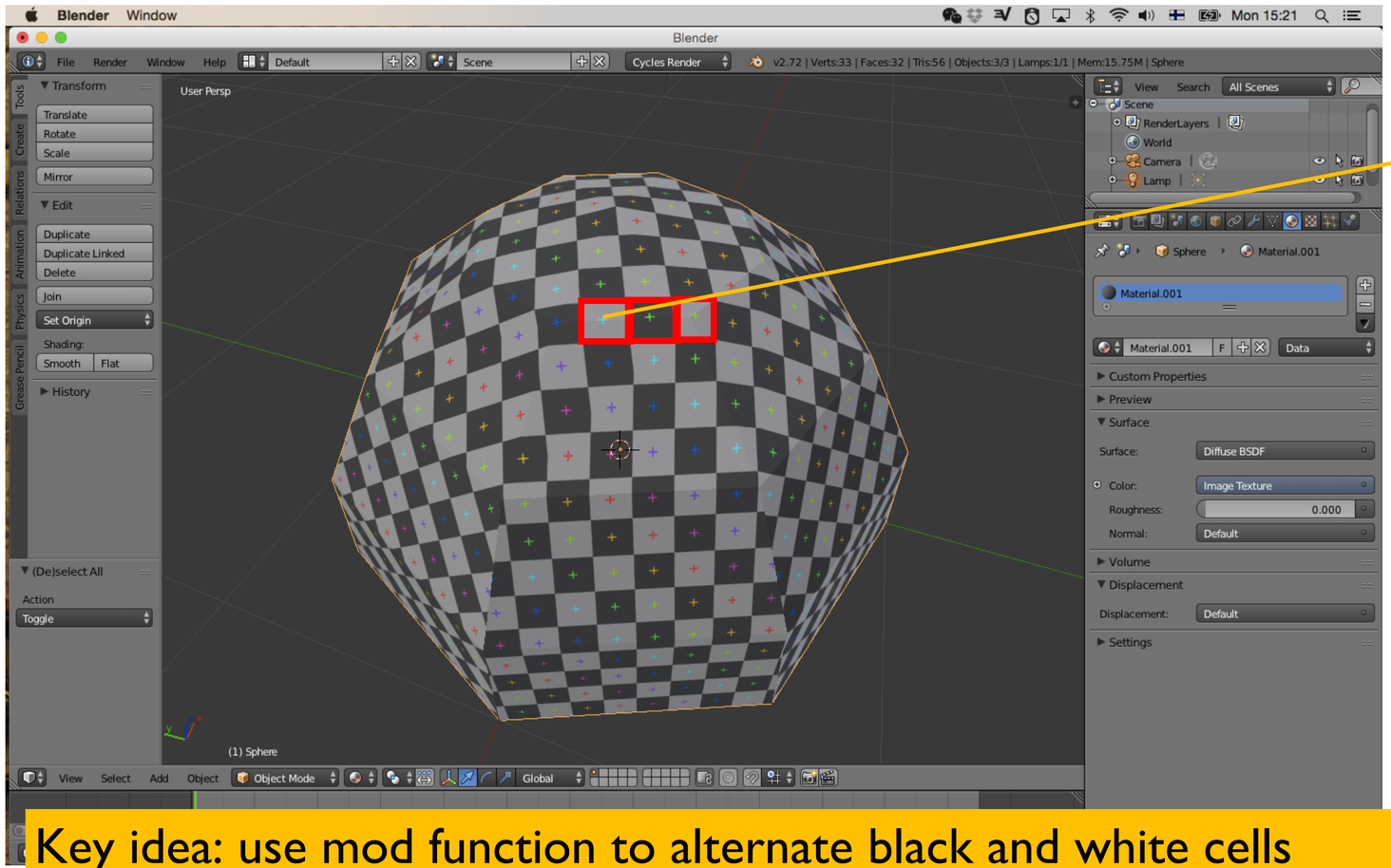
- Surfaces are 2D domains
- How do we map a surface point to a point in a texture image?

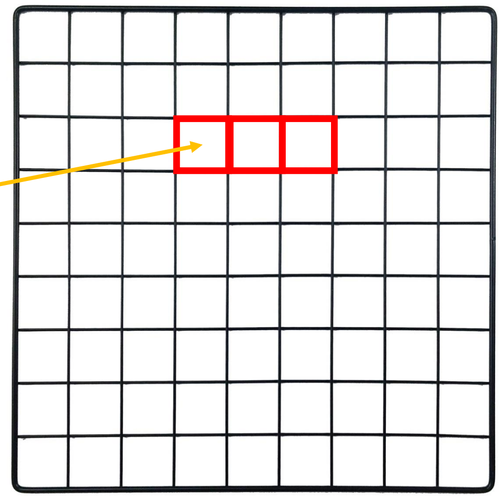


First, we need a way to visualize the mapping between surface and image!

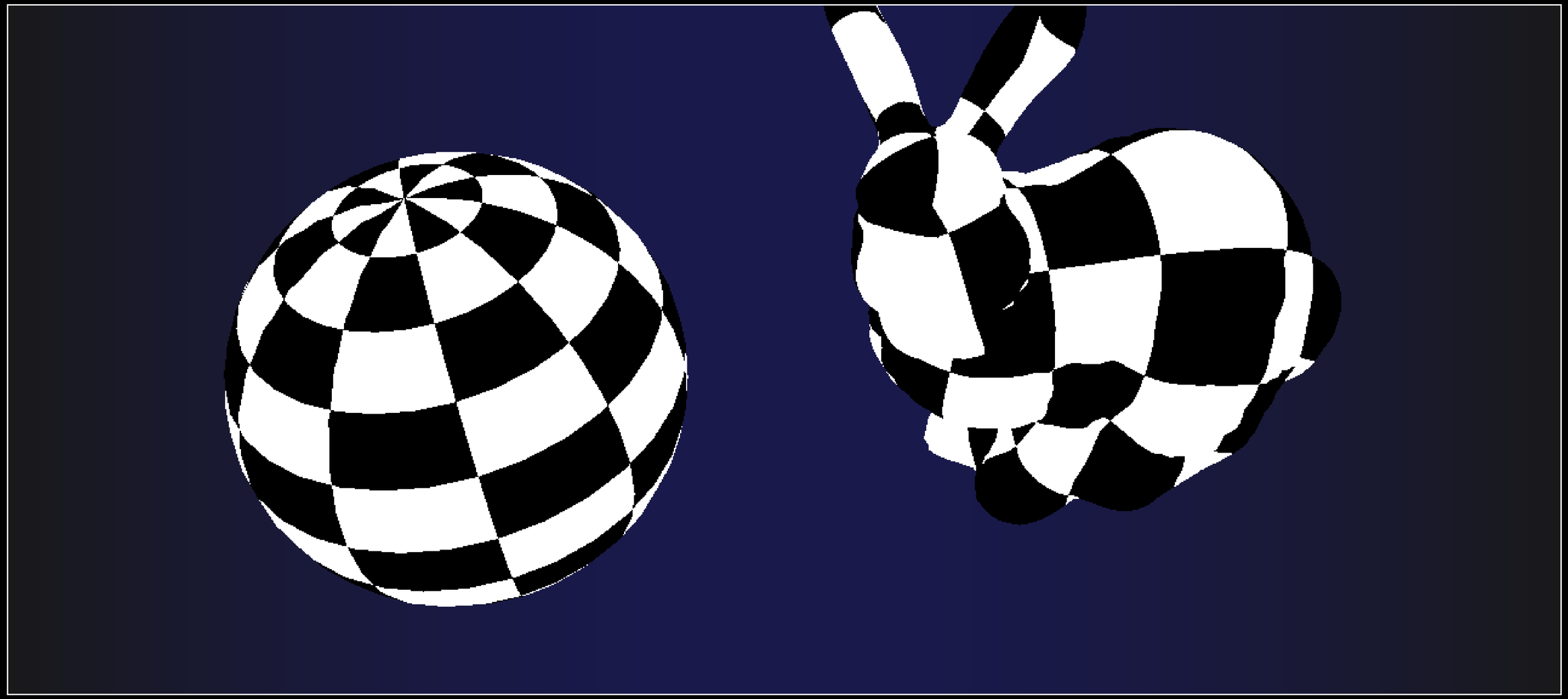
Checkerboard : A Common Practice to Visualize Mapping

- Shows the coordinate mapping between a surface and a Cartesian frame
- Illustrates the distortion of local geometry





Key idea: use mod function to alternate black and white cells
- For every input coordinate, decide whether it is black or white



A very good debug/visualization tool to check the mapping function on a mesh.

Pseudocode: Drawing a Checkerboard

- **Input:** uv
- **Output:** frag_color
- **Algorithm:**

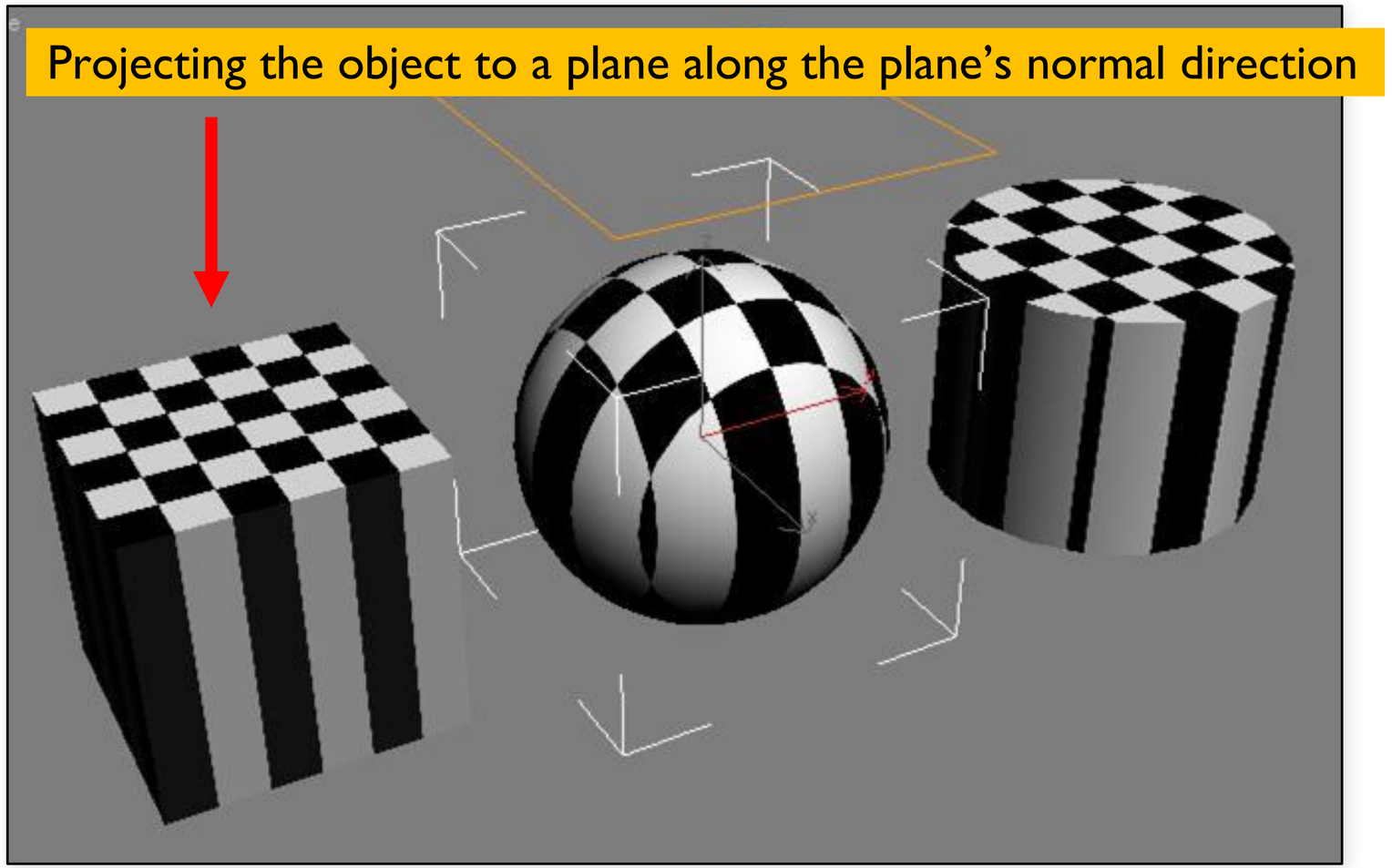
```
//// set the cell size
cell_size = 10
//// get the cell index in x and y axes

ix = floor(cell_size*uv.x)
iy = floor(cell_size*uv.y)
//// set the fragment color
if( (ix+iy) mod 2 equals 1 )
        frag_color = black
else frag_color = white
```

Planar Projection: Let's look at a simple planar projection case

- Map from a 3D surface to an image with projection along a specific direction

$$M(p) = \begin{bmatrix} p_x/w \\ p_y/h \end{bmatrix}$$

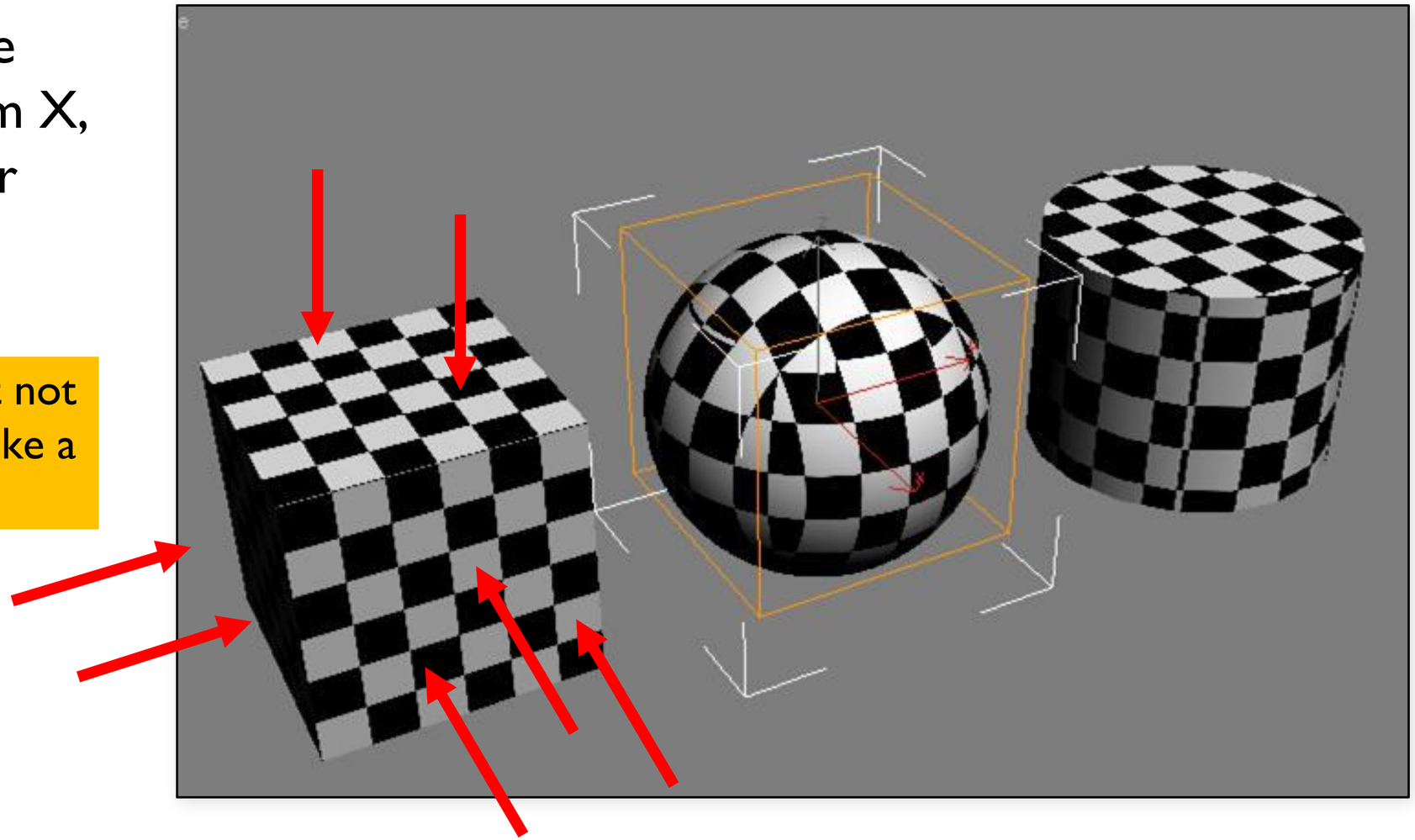
Here we assume the object's bounding box starts from the origin, i.e., $p_x \in [0, w]$, $p_y \in [0, h]$



Planar Projection: Let's take care of each facet in separate

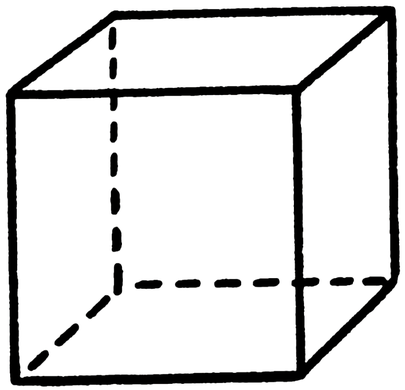
- For each 3D point, we select a direction from X, Y, Z to perform planar projection

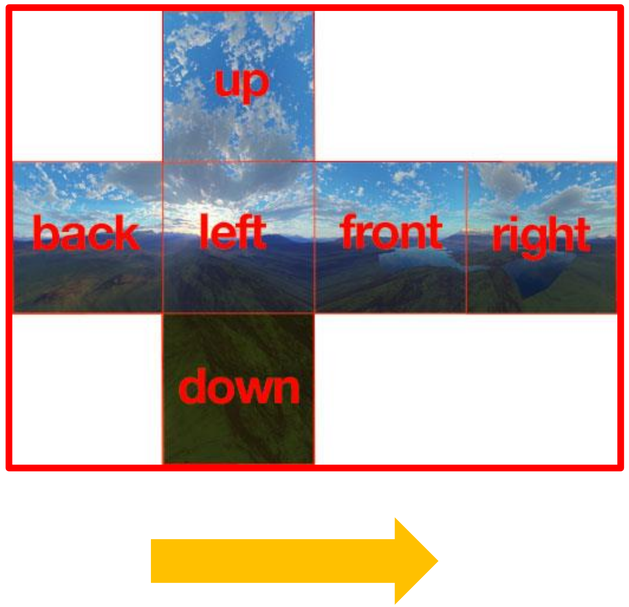
Works well for a cube, but not so great for other shapes like a sphere or a cylinder.

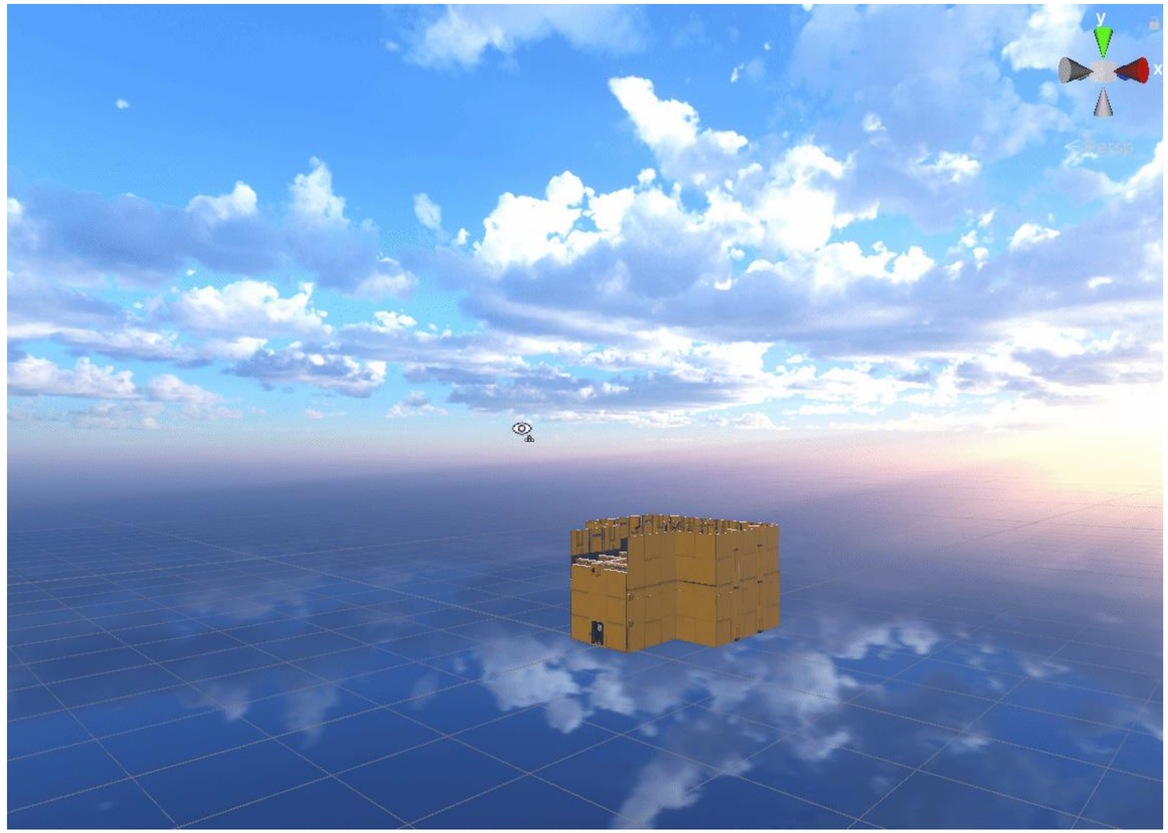


The idea of multi-faceted planar projection motivates **skybox**

- Map each facet of the box with an image
- We will discuss this in details in future lectures







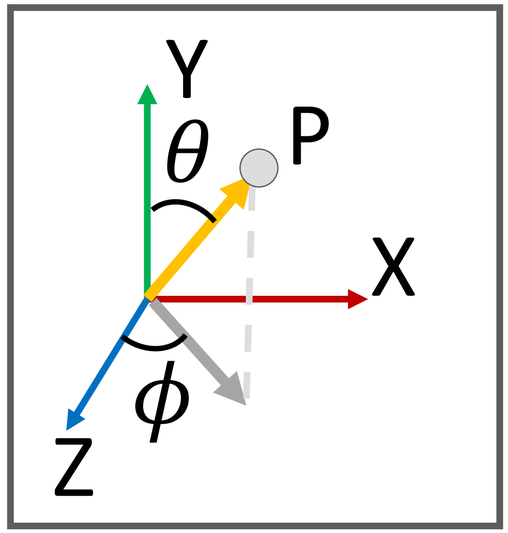
We will talk more about skybox in future lectures.

Spherical Projection: How about using spherical coordinates?

$$M(p) = \begin{bmatrix} \phi/2\pi \\ \theta/\pi \end{bmatrix}$$

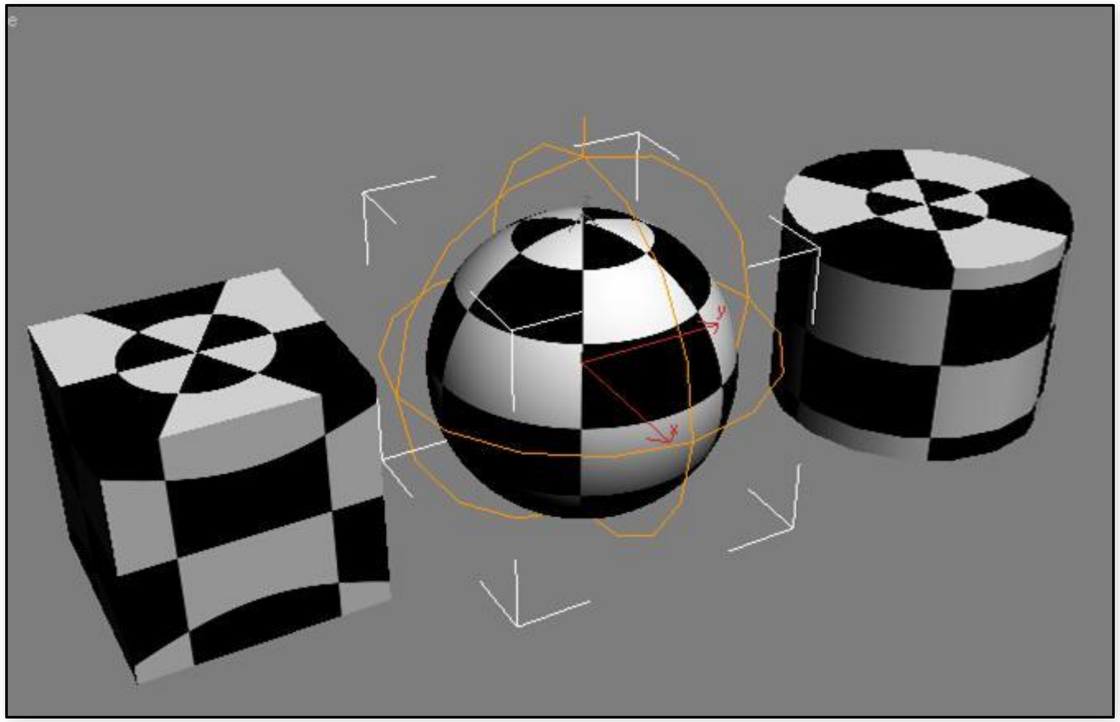
$$\theta \in [0, \pi)$$

$$\phi \in [0, 2\pi)$$



Spherical Coordinates

The idea is to first map a surface point (x, y, z) to (ϕ, θ), and then to (u, v)



Looks good for a sphere, but not so great other shapes.

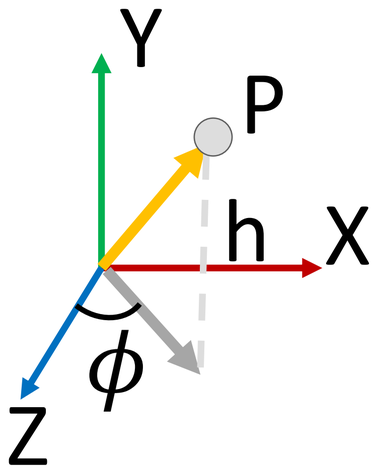
For a sphere, the mapping has singular points on the two poles.

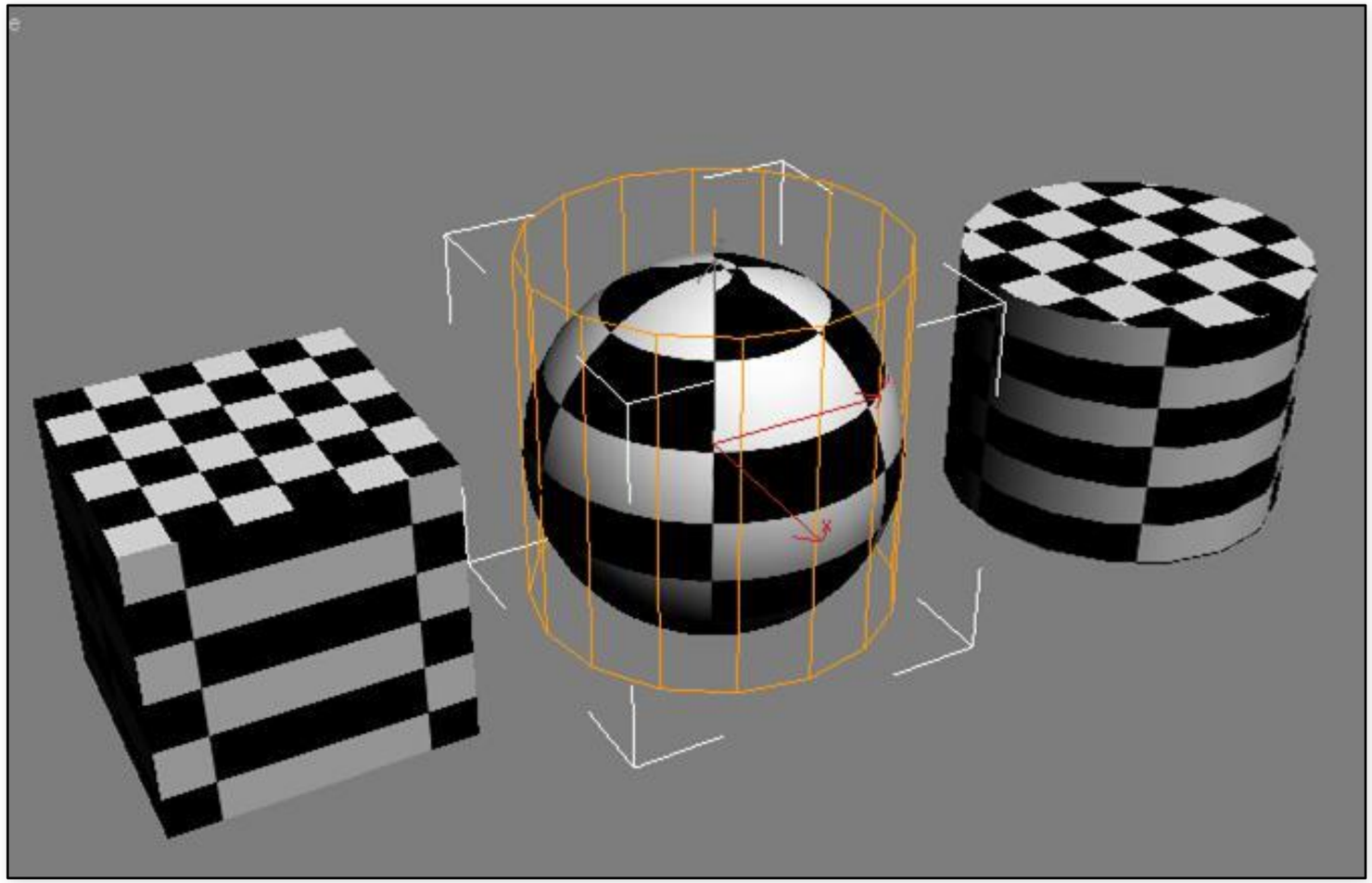
Cylindrical Projection: Use h instead of θ

$$M(p) = \begin{bmatrix} \phi/2\pi \\ p_y/h \end{bmatrix}$$

$$\phi \in [0,2\pi)$$

$$p_y \in [0, h]$$



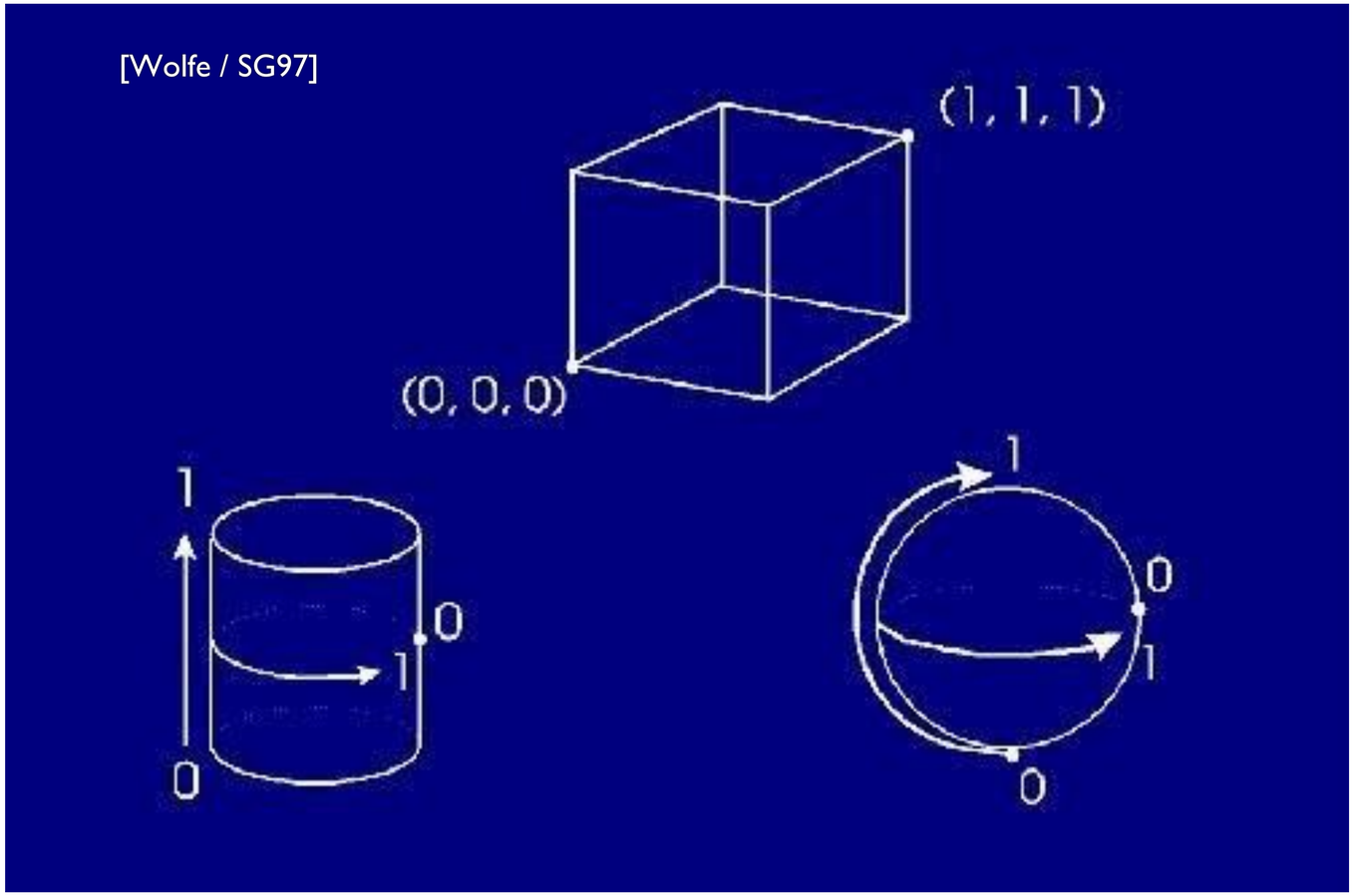


Summary: Projection Mapping

- Maps 3D surface points to 2D image coordinates

$$\boldsymbol{M: R^3 \rightarrow [0,1]^2}$$

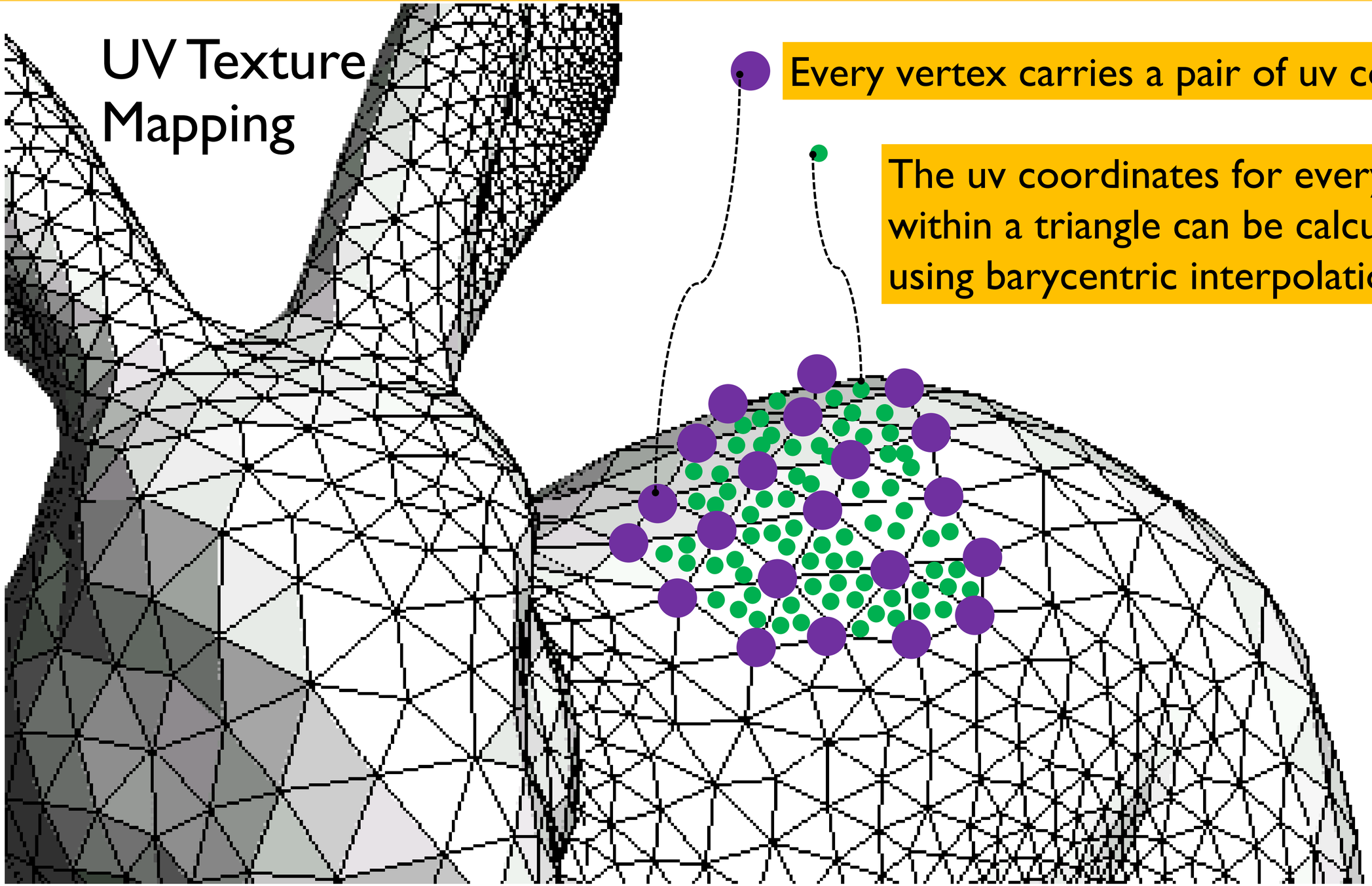
- Different types of projections often corresponding to different kinds of simple shapes
  - Useful for simple objects
  - Easy to calculate and no storage requirement



We need more general solutions for objects with arbitrary shapes!

UV-based Mapping

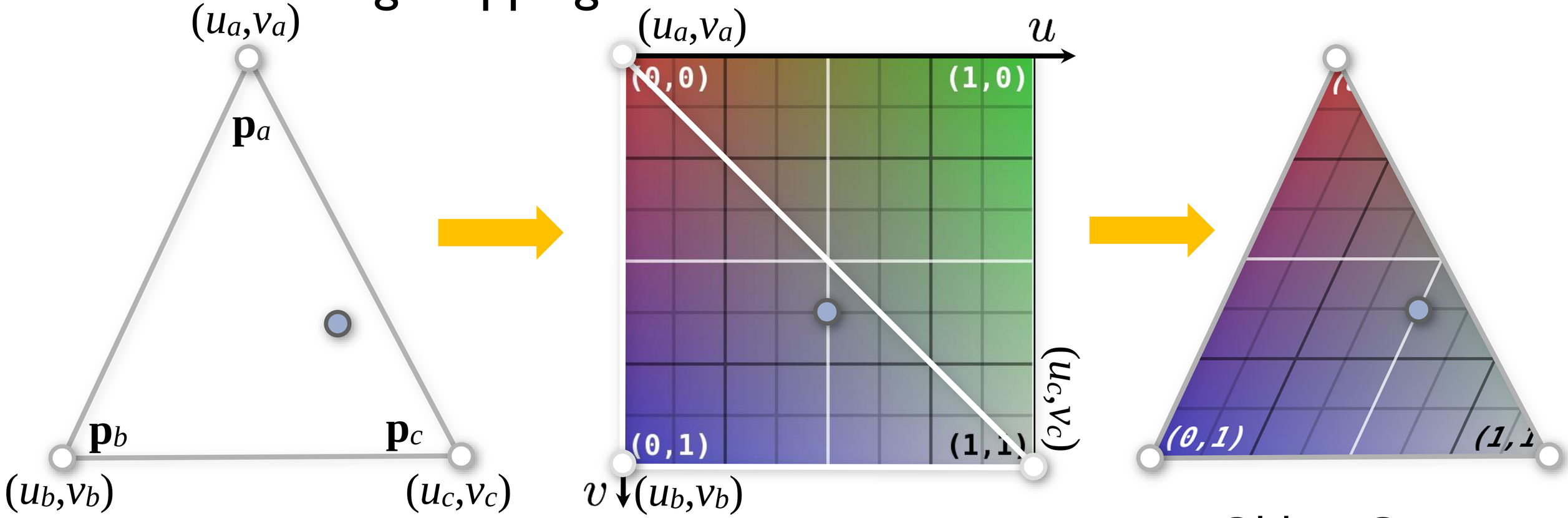
UV Texture Mapping



Every vertex carries a pair of uv coordinates

The uv coordinates for every point within a triangle can be calculated using barycentric interpolation

UV Texturing Mapping



Object Space | Texture Space | Object Space

$$(u,v) = \alpha(u_a,v_a) + \beta(u_b,v_b) + \gamma(u_c,v_c)$$

Key Idea: Assigning each vertex a texture coordinate, and then use barycentric interpolation to determine the texture coordinate for an arbitrary point within a triangle

Texture Mapping on Sphere and Bunny Meshes

How do we create such mapping?

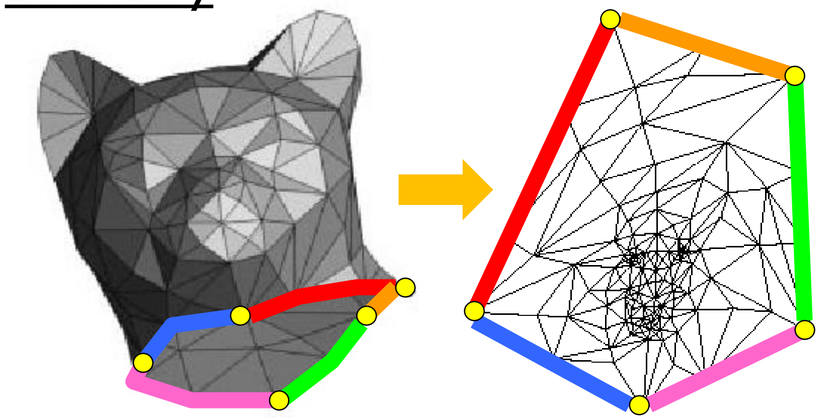
Texture creation is like garment pattern making in fashion design --- kind of labor-intensive work combining art and experience

Graphics artists need to specify ...

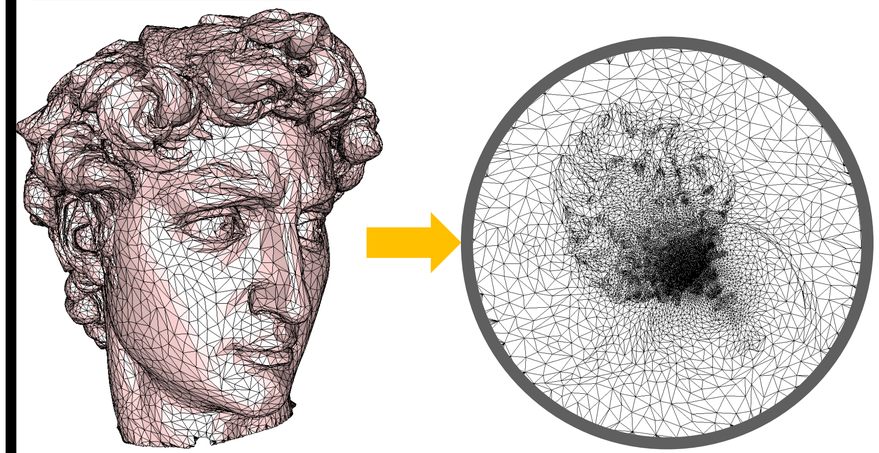
- How each triangle is deformed on the texture image plane
- How new cuts are made
- How different pieces are combined
- How the boundary is mapped

Reading: Mesh Parameterization Methods and Their Applications
https://www.cs.ubc.ca/~sheffa/papers/CGV011-journal.pdf

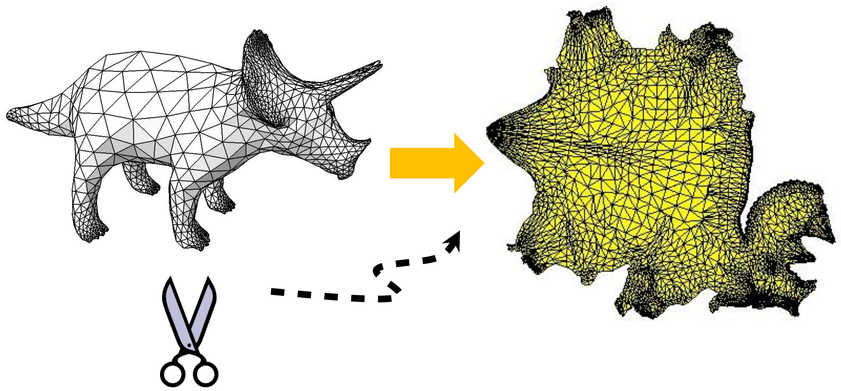
Boundary



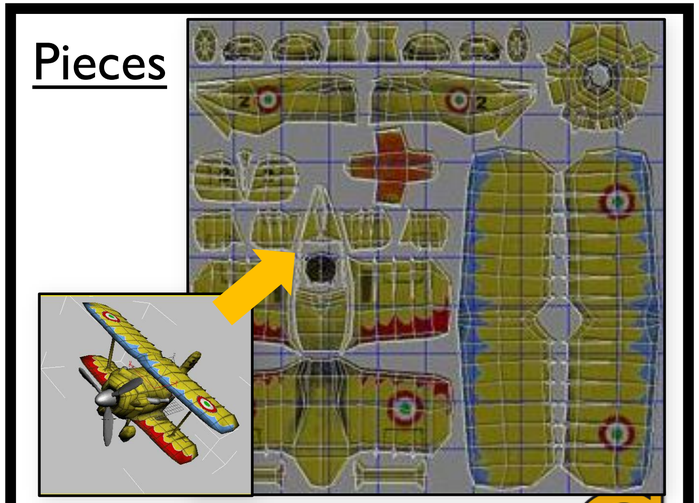
Deformation



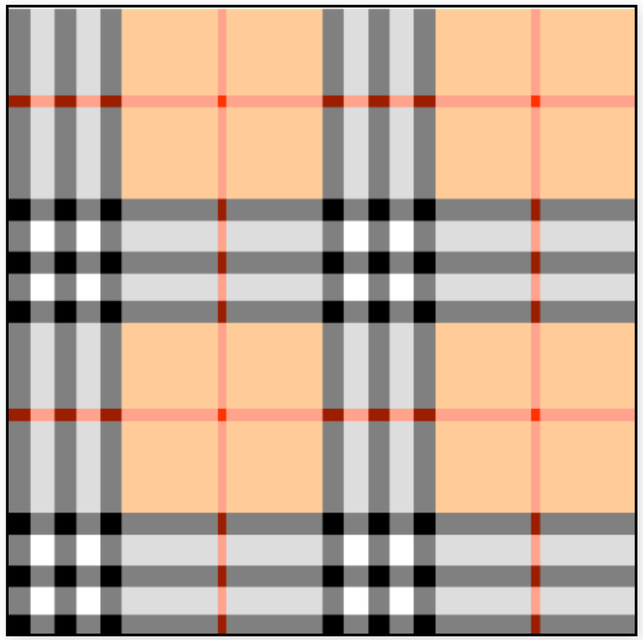
Cuts

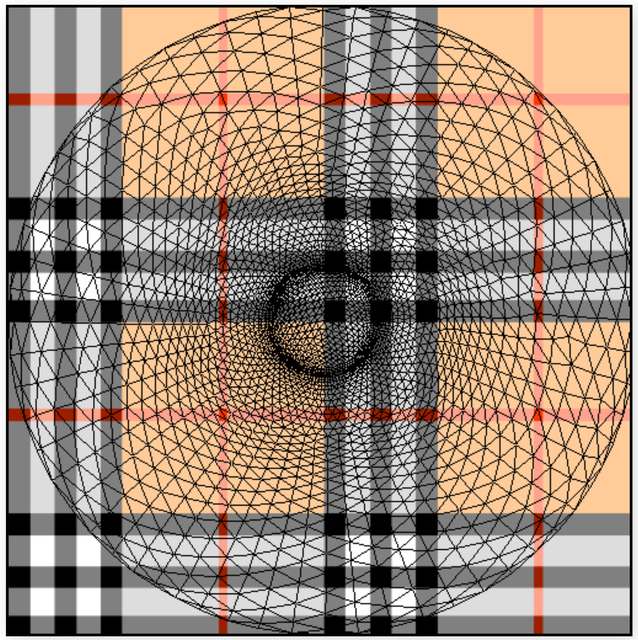


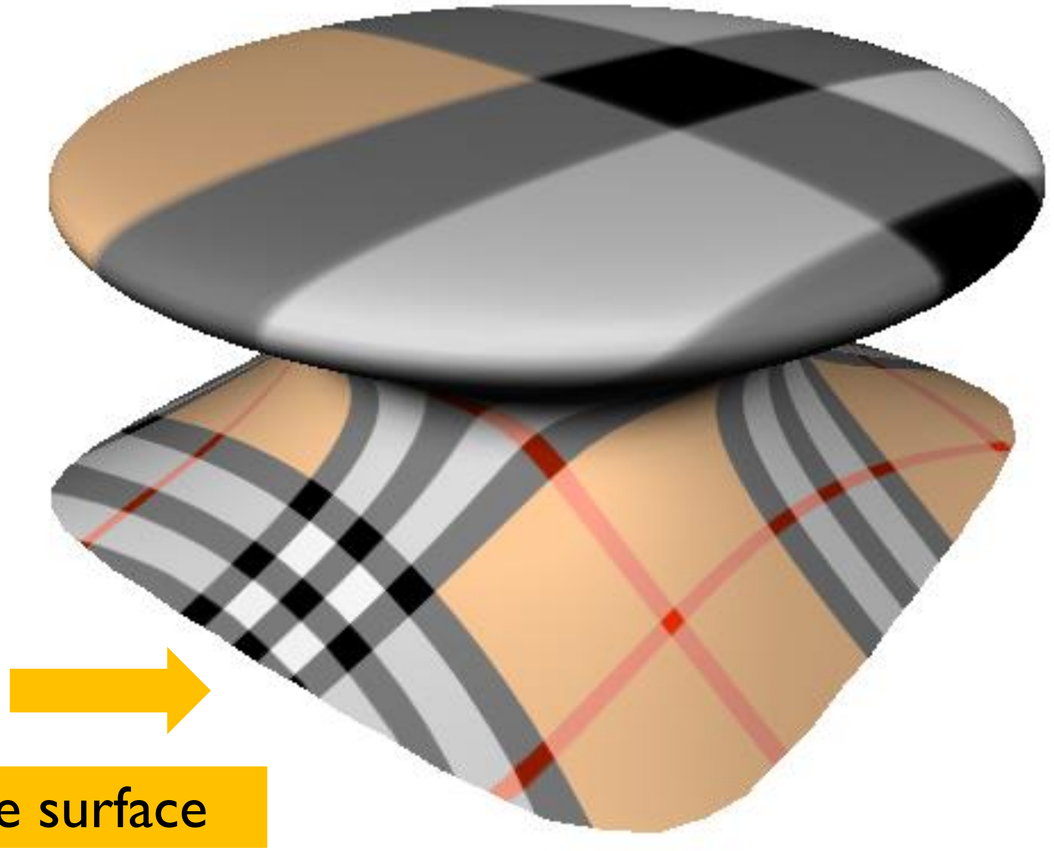
Pieces



Artifacts may emerge if texture mappings are not specified properly







E.g., distorted mapping will create distorted colors on the surface

Nowadays, AI is playing a bigger role in automating texture generation

https://www.youtube.com/watch?v=5qy56r1FAWM

Study Plan

Mapping Function

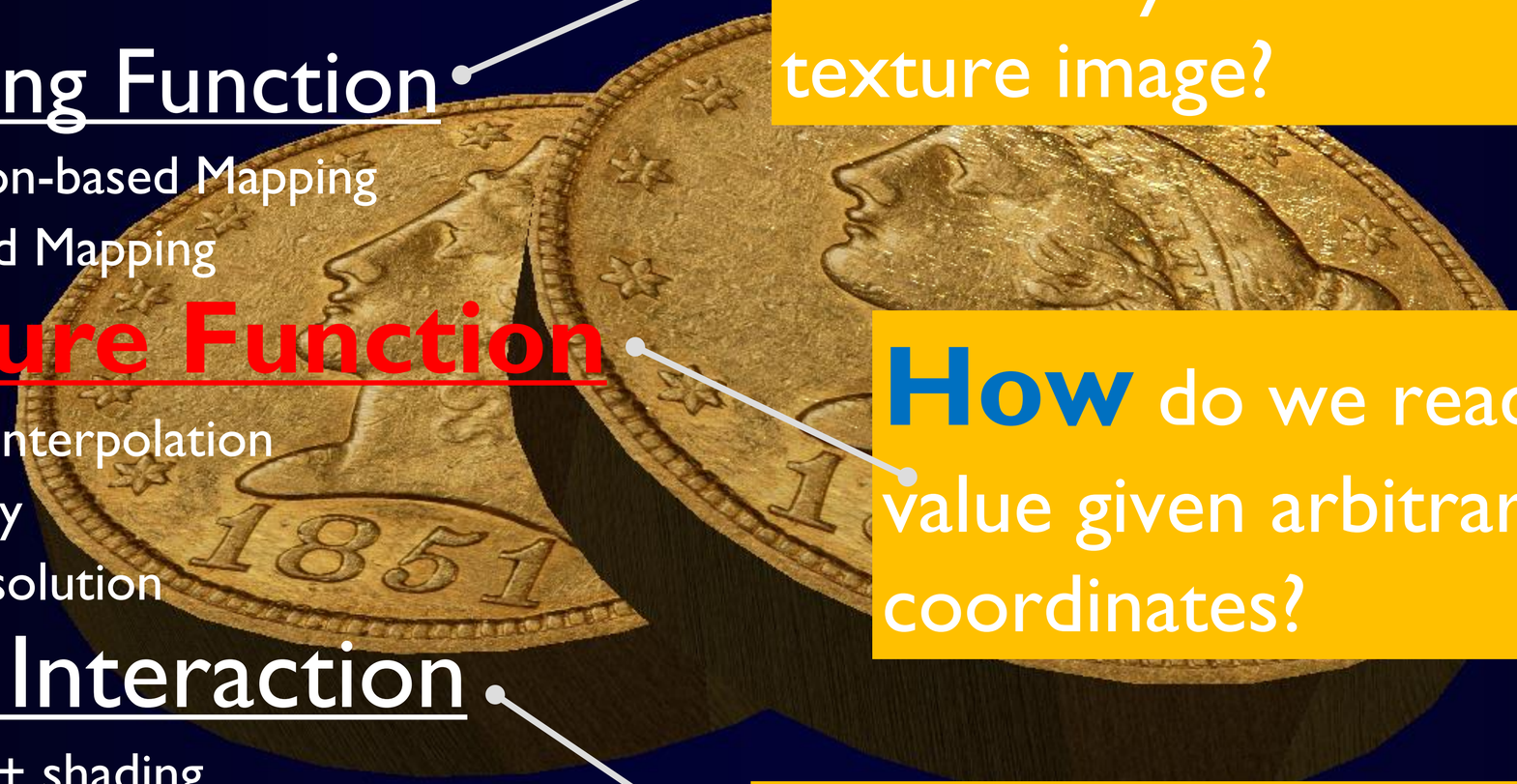
- Projection-based Mapping
- UV-based Mapping

Texture Function

- Bilinear interpolation
- Boundary
- Multi-resolution

Light Interaction

- Texture + shading
- Normal mapping



How do we map between an arbitrary surface and a texture image?

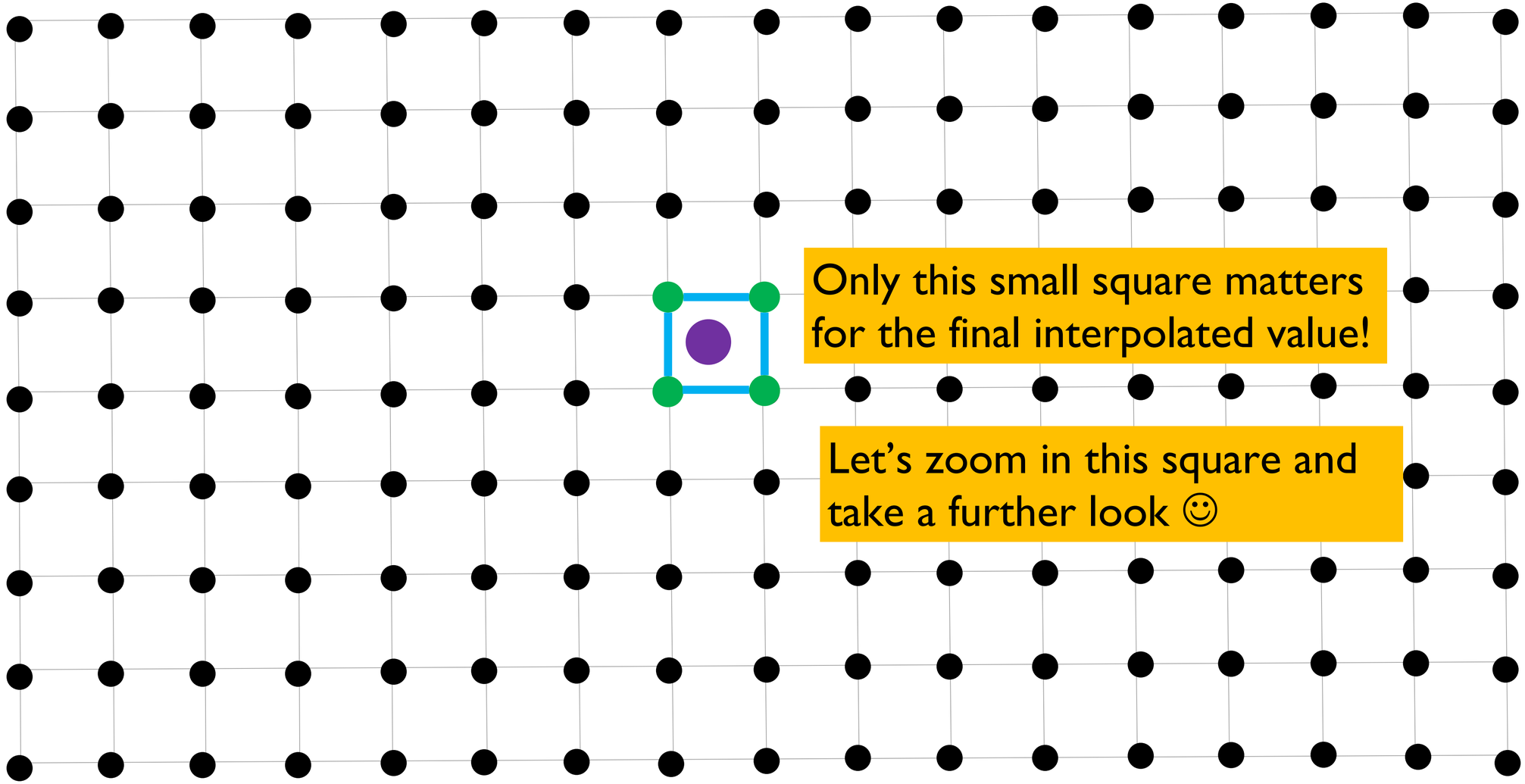
How do we read an image value given arbitrary uv coordinates?

What do we read from an image?

Bilinear Interpolation

Texture Function

Given an inquiry point (u,v), we want to read its color from the image

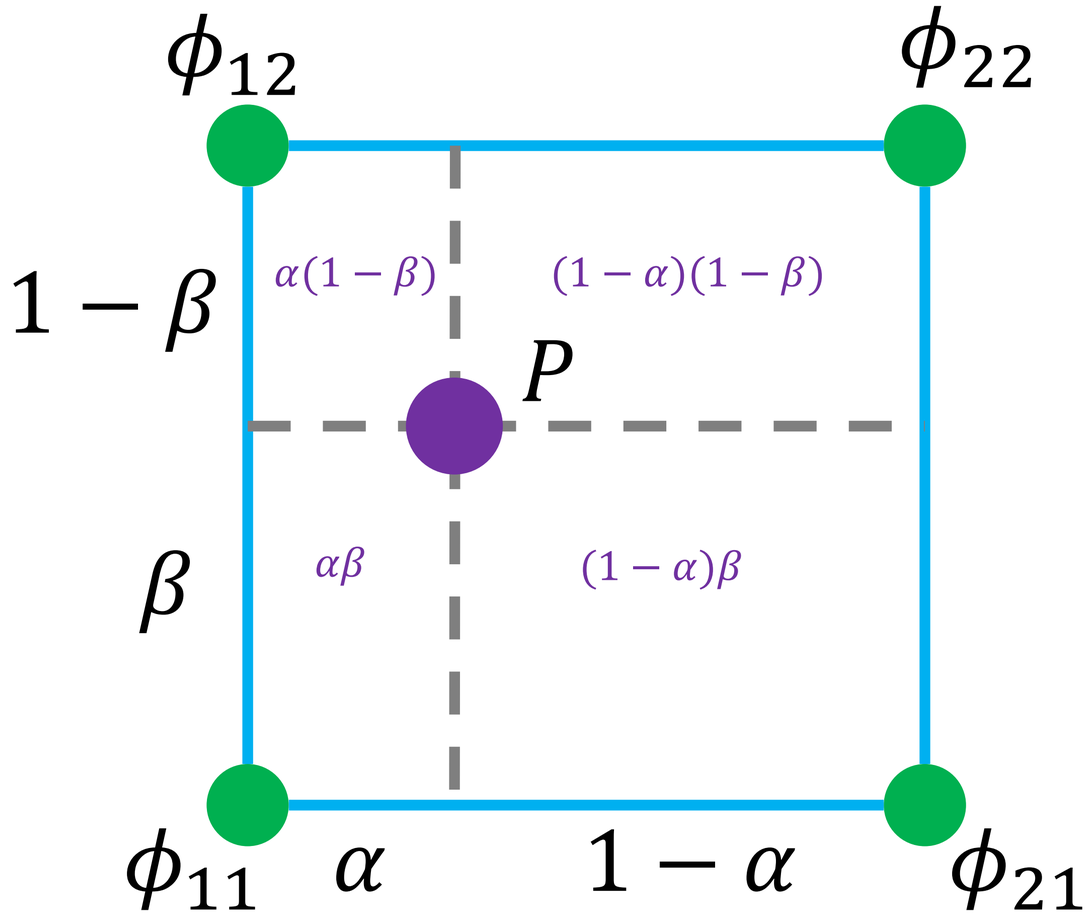


Only this small square matters for the final interpolated value!

Let's zoom in this square and take a further look ☺

We have a texture image, with black dots indicating its pixels.

Bilinear Interpolation



How to calculate the value of P given the four nodes $\phi_{11}, \phi_{12}, \phi_{21}, \phi_{22}$ and the fraction α and β?

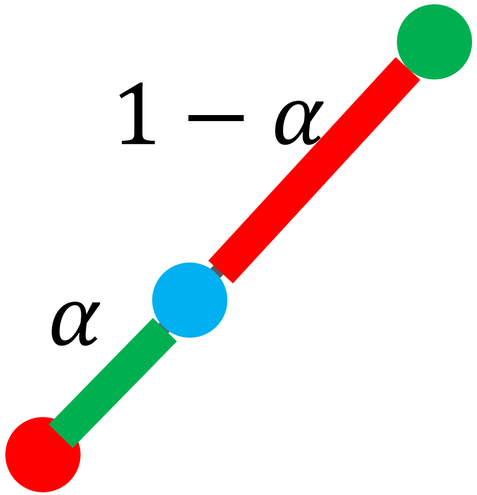
Key idea: again, we use the area of the opposite sub-rectangle!

$$
\begin{aligned}
&bilinear\_intp(P) \\
&= (1-\alpha)(1-\beta)\phi_{11} \\
&+ \alpha(1-\beta)\phi_{21} \\
&+ (1-\alpha)\beta\phi_{12} \\
&+ \alpha\beta\phi_{22}
\end{aligned}
$$

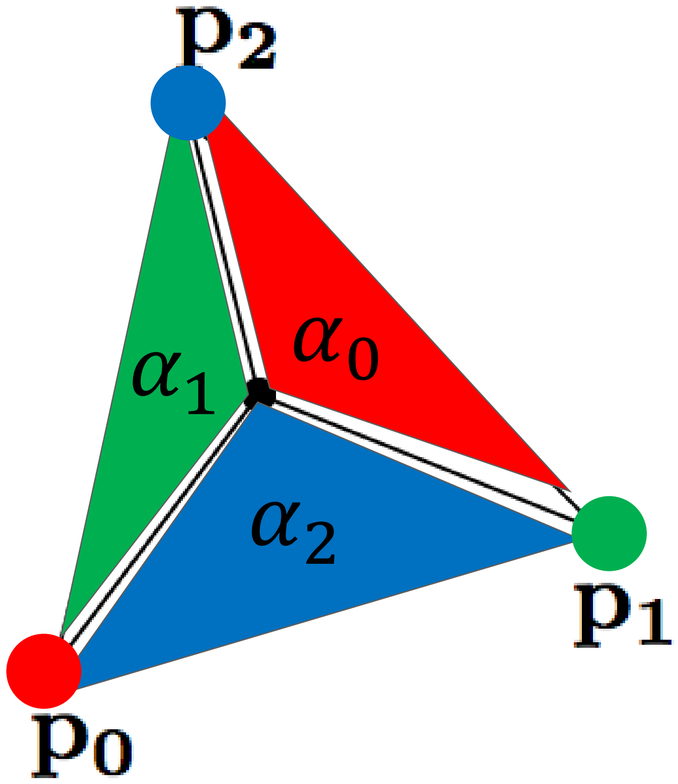
Last Question: How do we calculate α and β?
- Easy. Assuming s is pixel size, we have
$\alpha = (u \bmod s)/s, \alpha = (v \bmod s)/s$

Quick Summary: All Kinds of Interpolations in Graphics

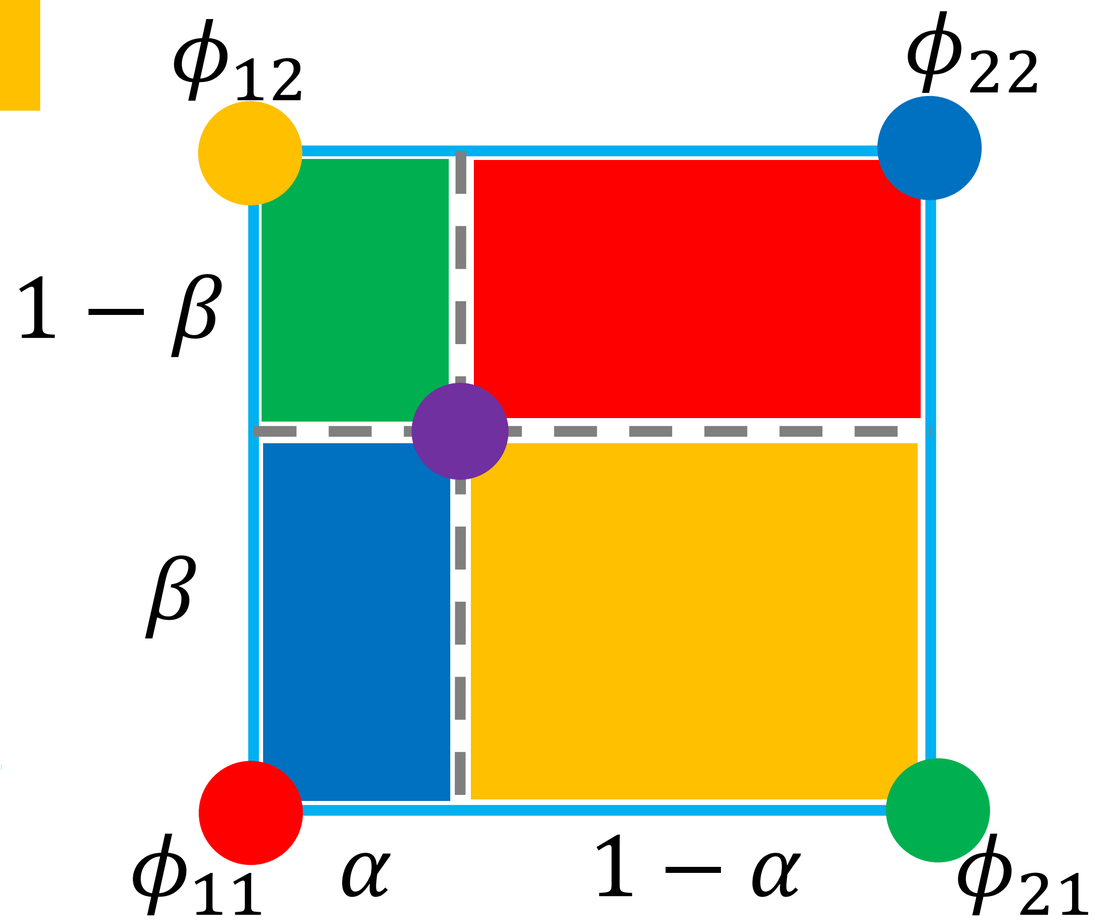
Notice the colored nodes and their correspondingly colored segment/triangle/rectangle area.



Linear Interpolation



Barycentric Interpolation



Bilinear Interpolation

Code Implementation of Texture Interpolation

//// declare the texture sampler

```
uniform sampler2D tex;
```

- Each texture sampler is linked to a texture image loaded from the CPU side.
- You can imagine a sampler as an image class with the implemented bilinear interpolation on GPU.

//// call GLSL built-in function texture to query color with uv

```
vec4 col = texture(tex, uv);
```

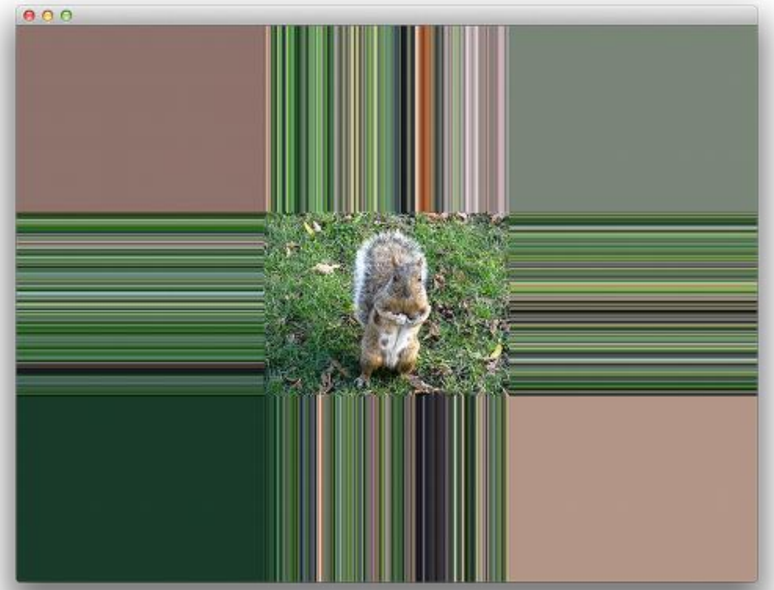
texture sampler name

texture uv coordinates

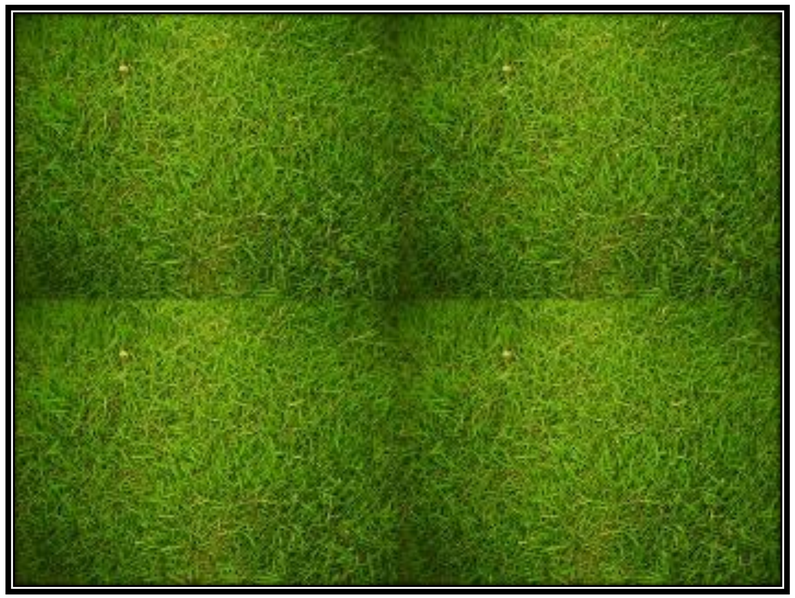
Texture Boundary

Boundary Treatment

- What if uv are out of range of [0,1]?
- Option 1, **clamp**: take the nearest pixel that is in the texture image
- Option 2, **repeat**: treat the texture as periodic, so that falling off the right side causes the look up to come in the left

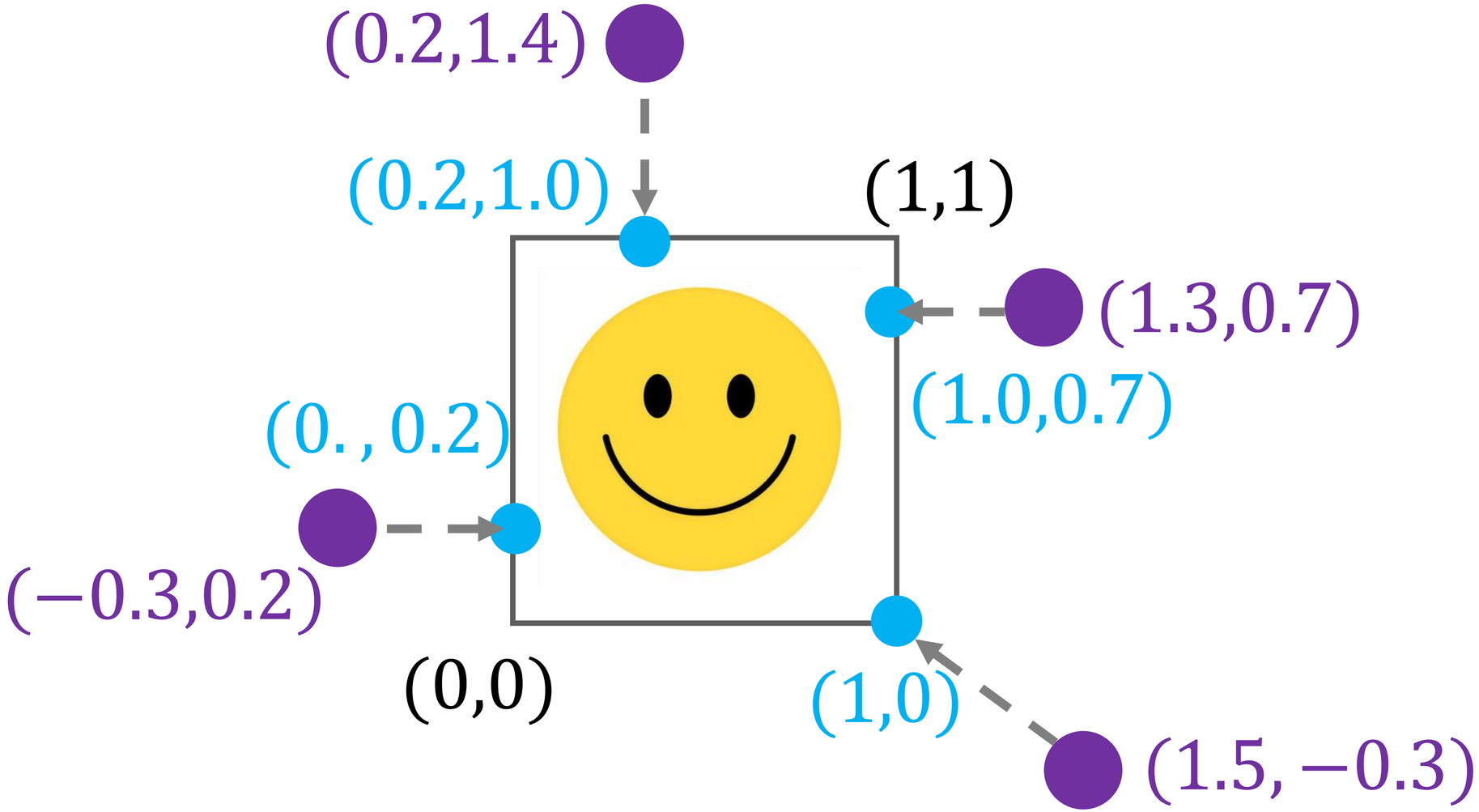


clamp



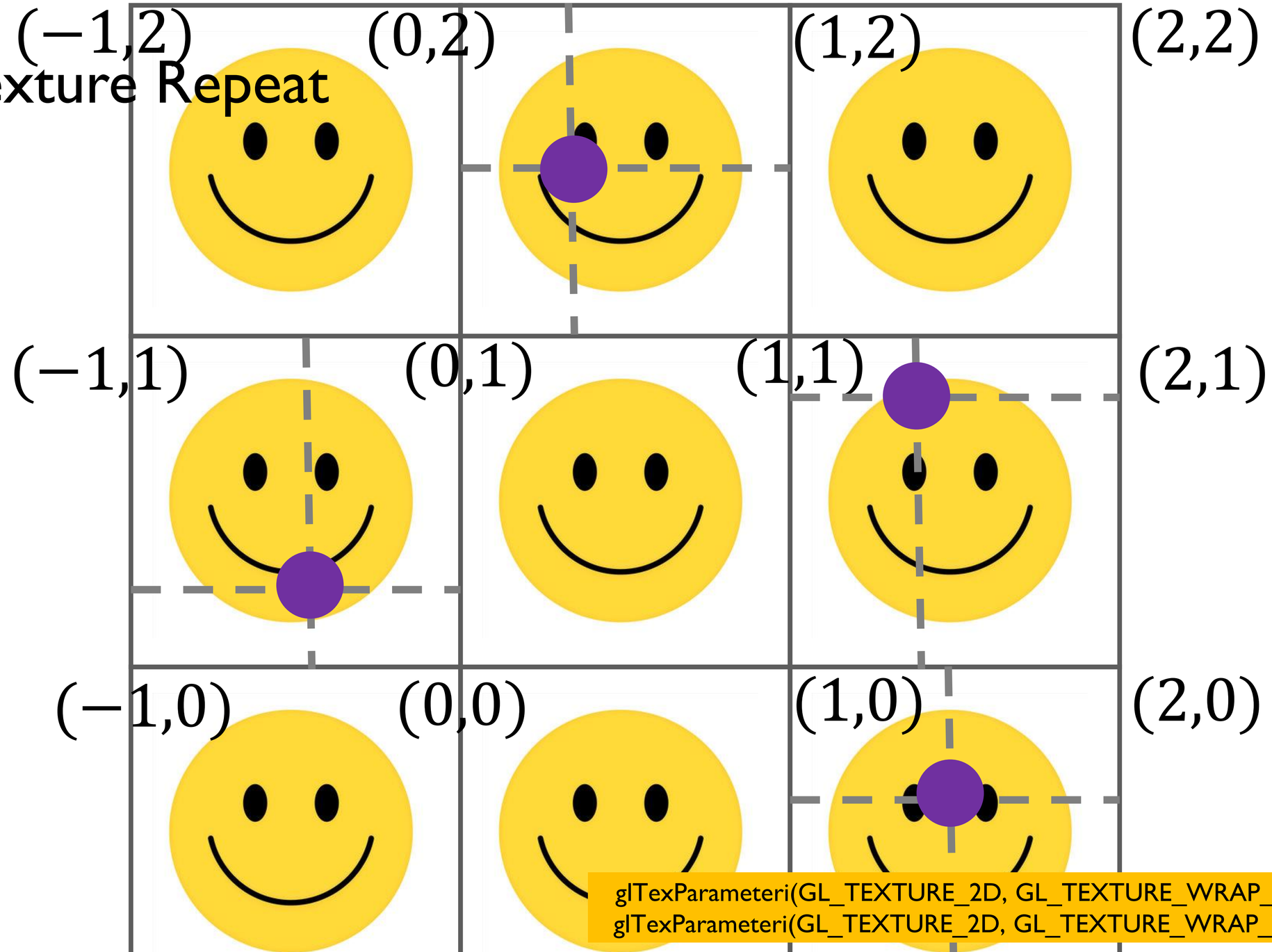
repeat

Texture Clamp




```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Texture Repeat



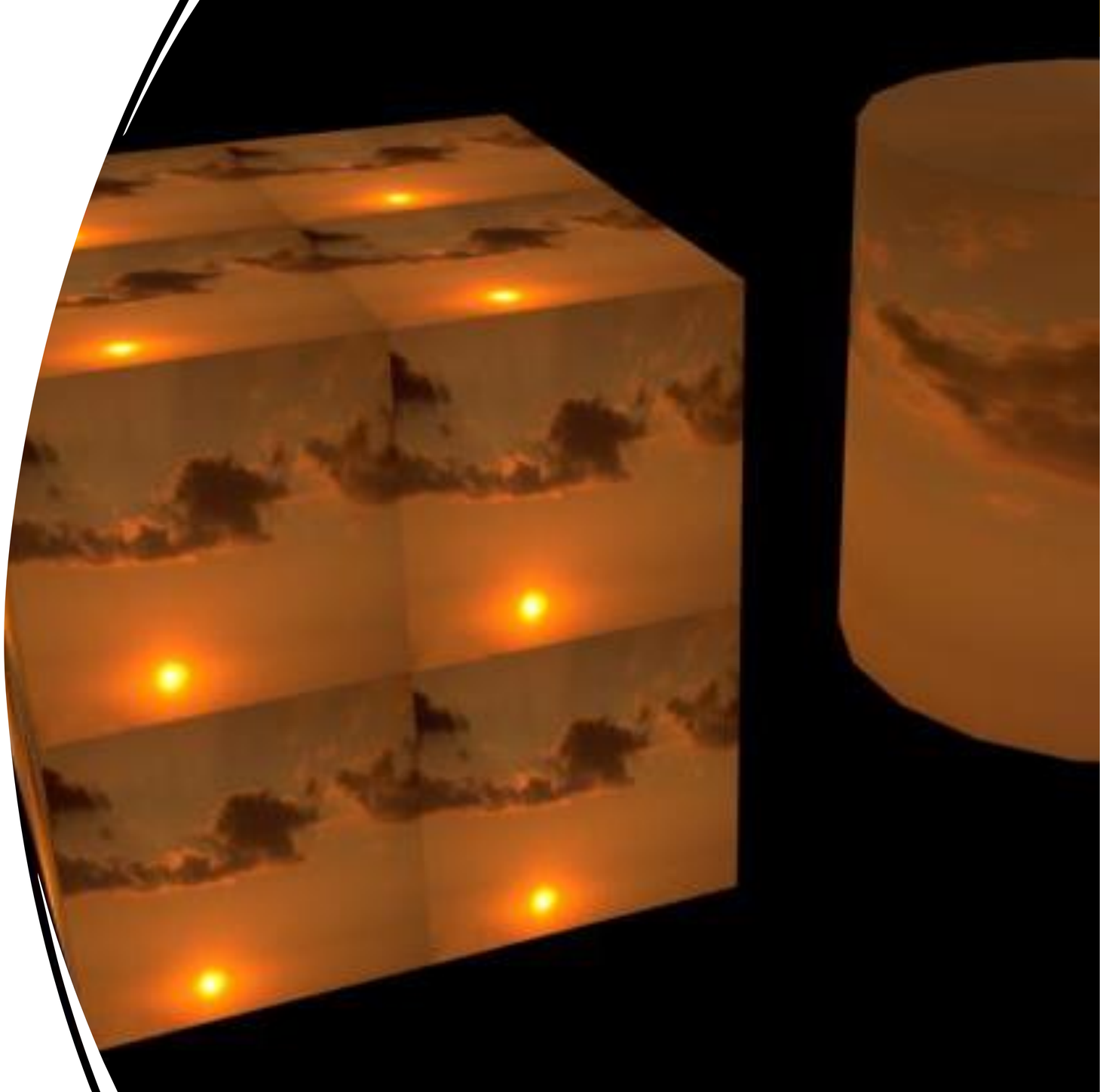

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Tileable Textures

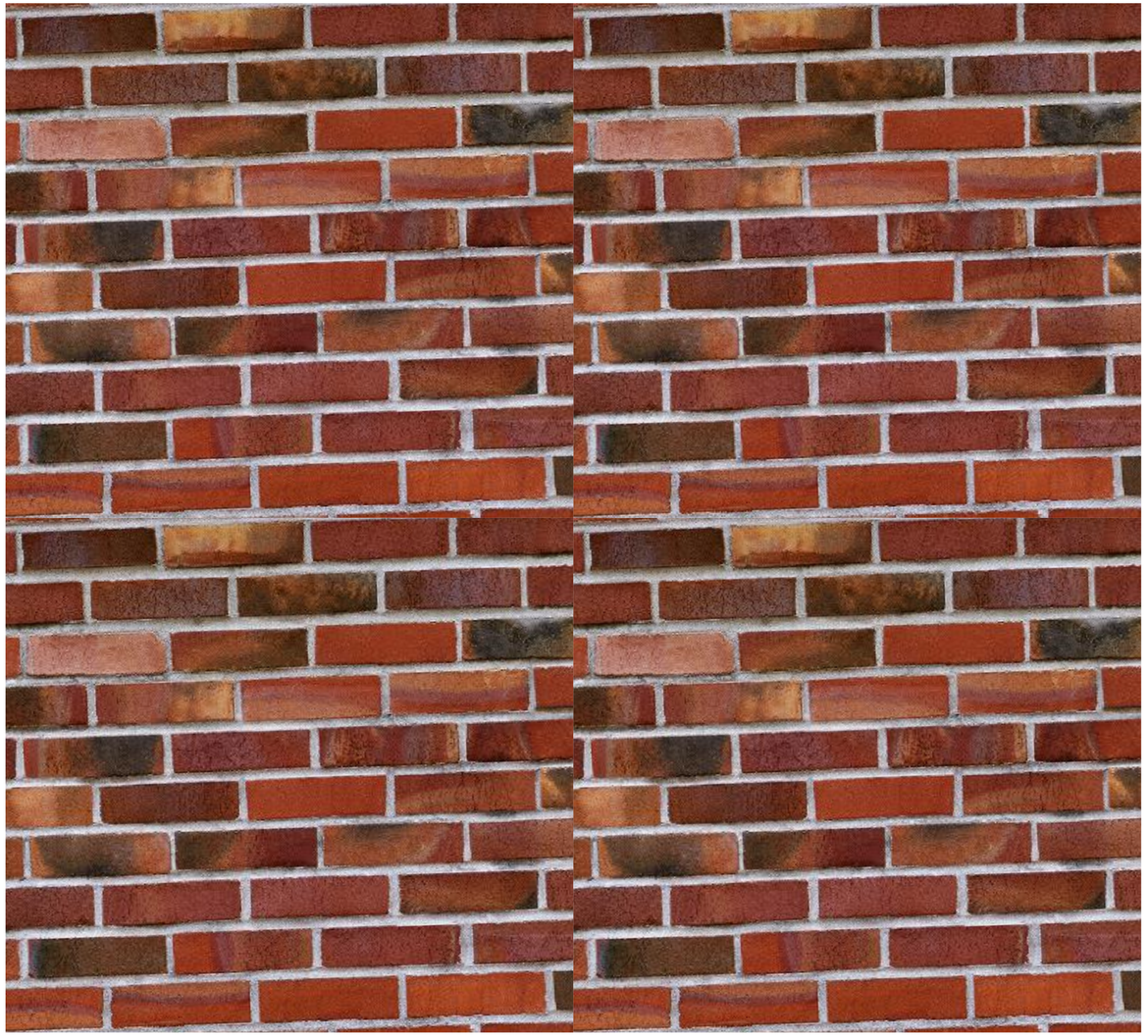
- Tiling textures might cause seams

- **Problem**: discontinuities in the mapping function

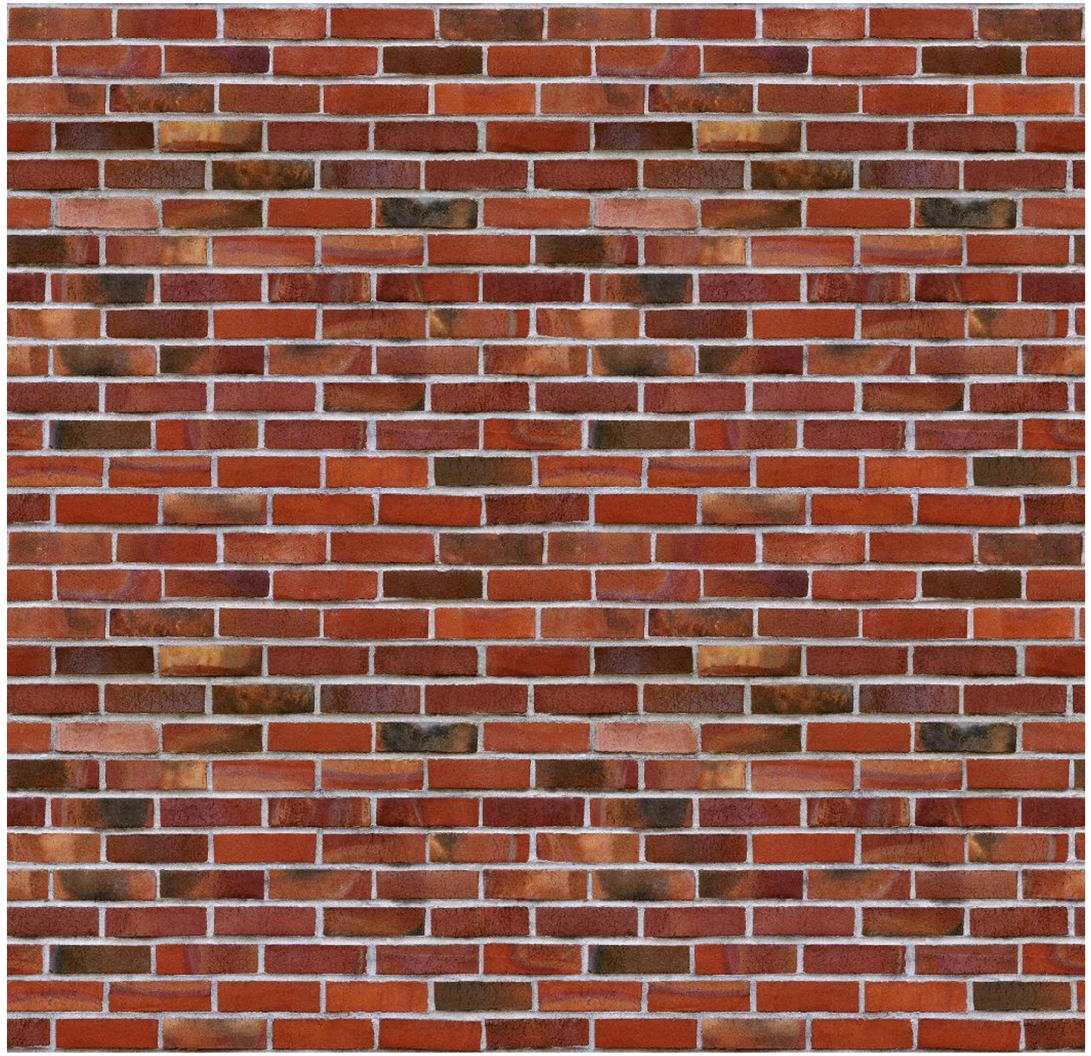
- **Solution**: change textures to be "**tileable**" when possible



Seamlessly “tileable" textures

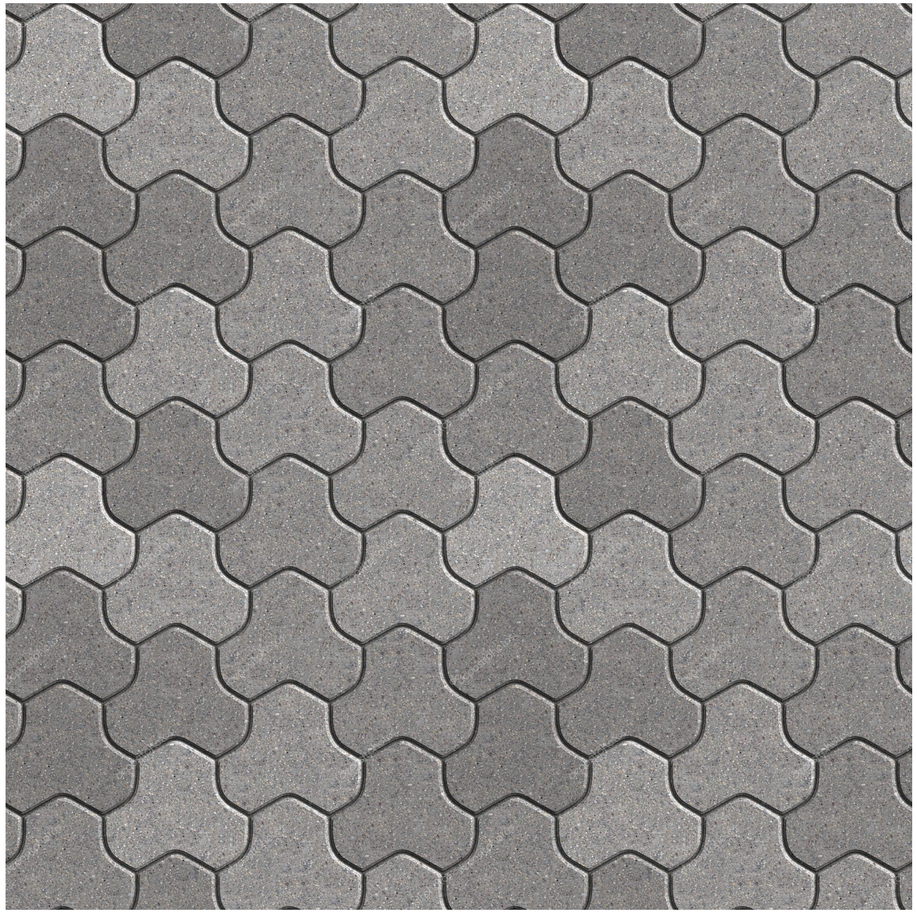


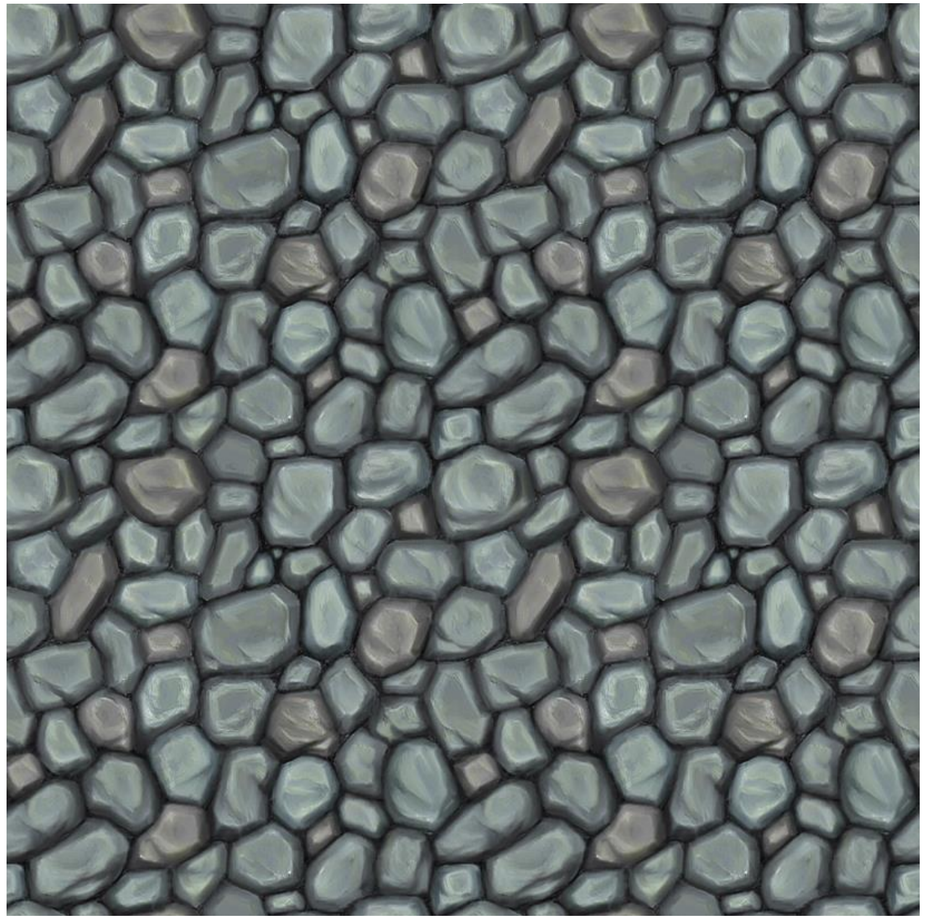
Non-tileable



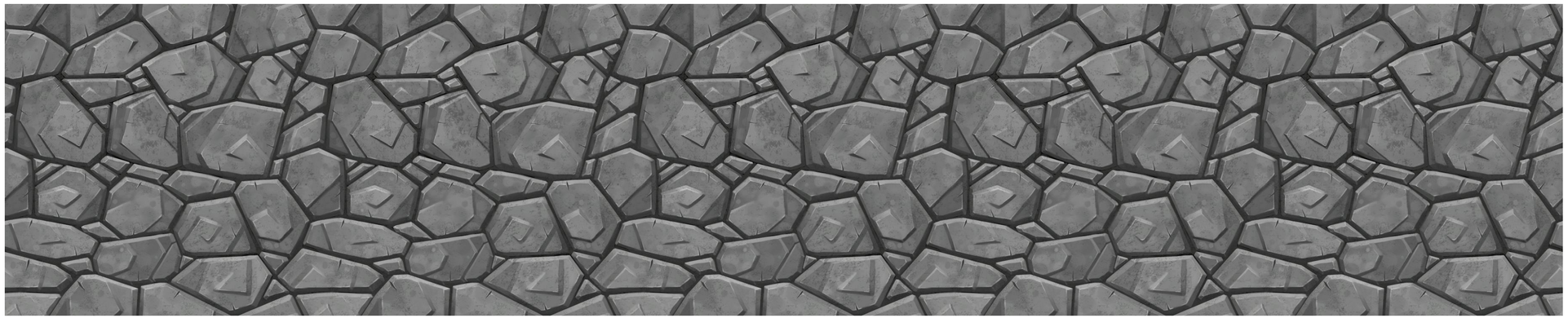
Tileable

More Examples of Tileable Textures





More Examples of Tileable Textures



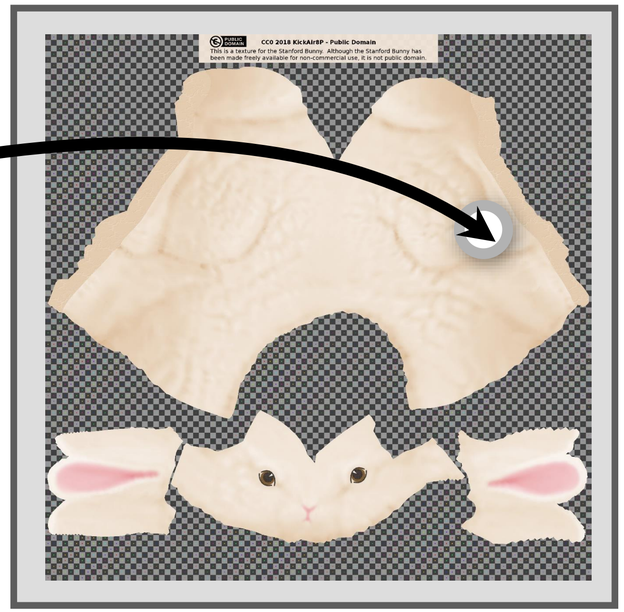
Texture Resolution

Texture Resolution

- Ideally, we hope to have a 1-to-1 mapping between the **pixels rendered on screen** and the **pixels read from texture**, e.g., my screen is 300 x 400, my texture is also 300 x 400.
  - Every rendered pixel on screen can be mapped to a unique pixel from the texture image
- In reality, a screen pixel could be smaller or larger than a texture pixel, which will cause artifacts

Screen Image

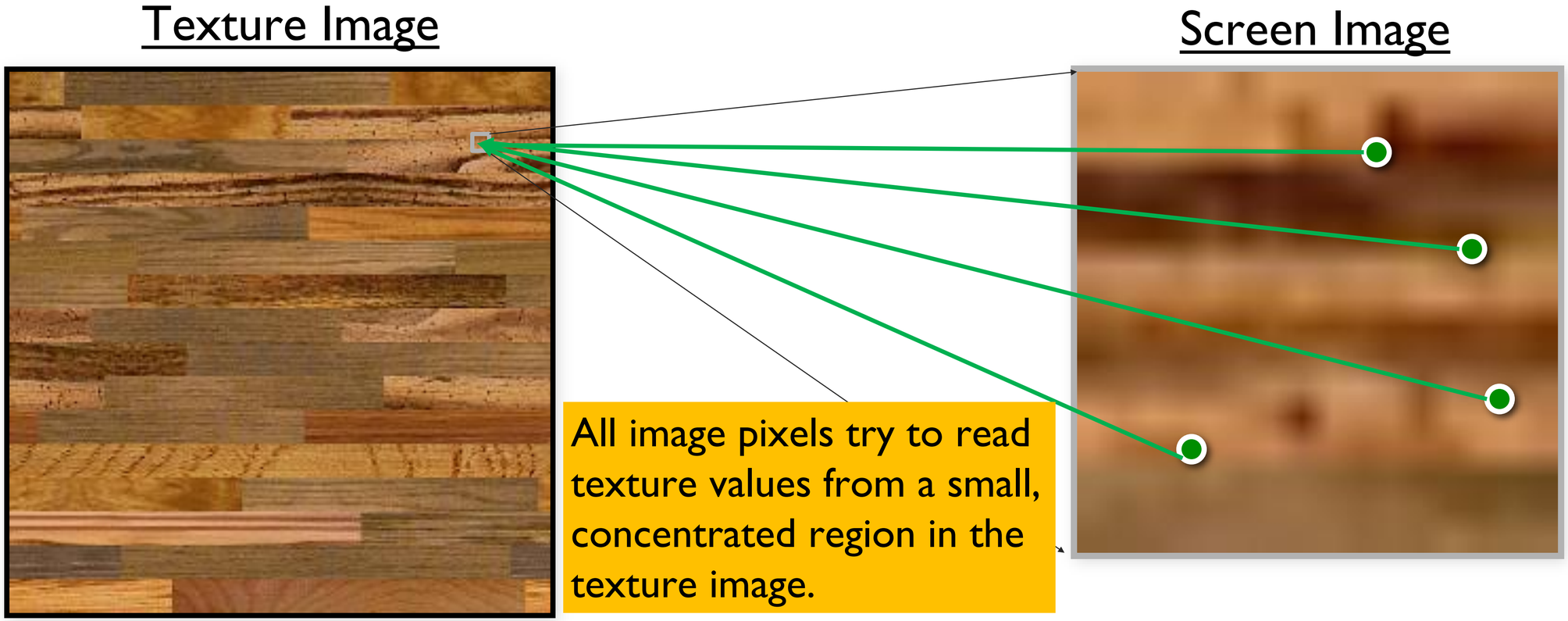
Texture Image



Ideally, we want a one-to-one mapping between screen pixels and texture pixels

If a screen pixel is too small…

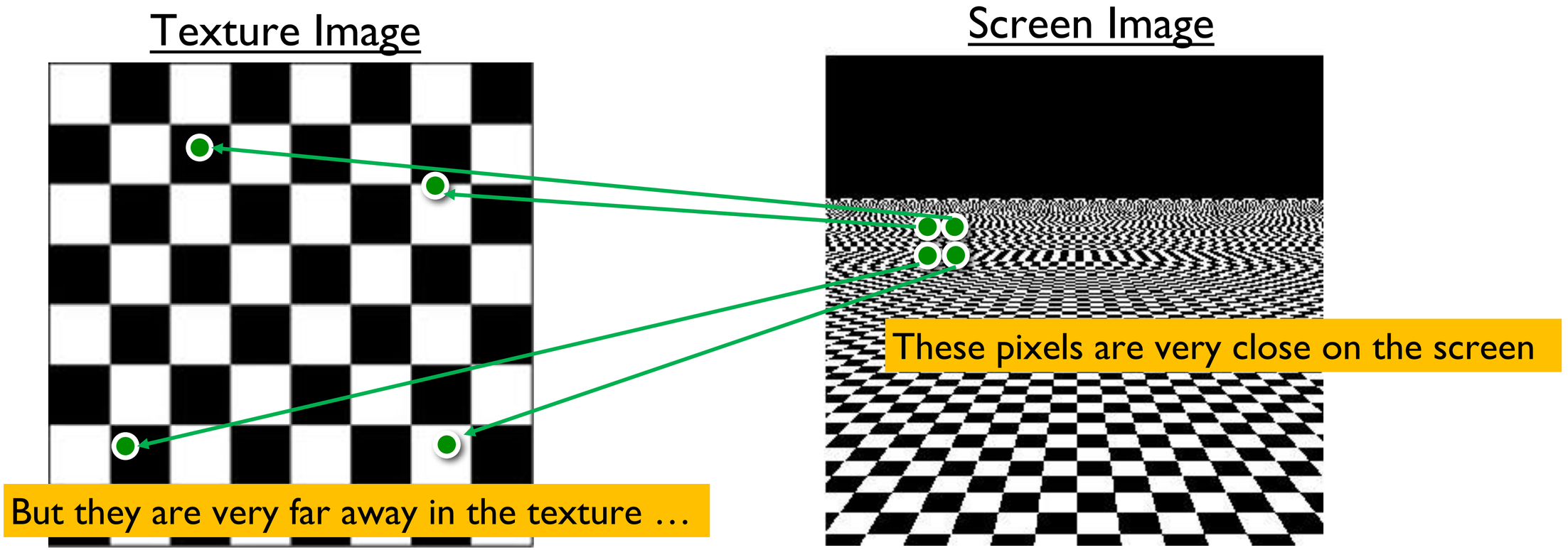
- The rendered image will be **blurred** because most of the pixel values on the screen are read from a small region of the texture image



Screen pixel too small:
Reading texture values from a small region of the texture image

If a screen pixel is too large…

- The rendered image will be **aliased** because the pixel values on the screen are read sparsely from the texture image, with each read separated from others



Screen pixel too large:
Reading texture values sparsely from a large region of the texture image

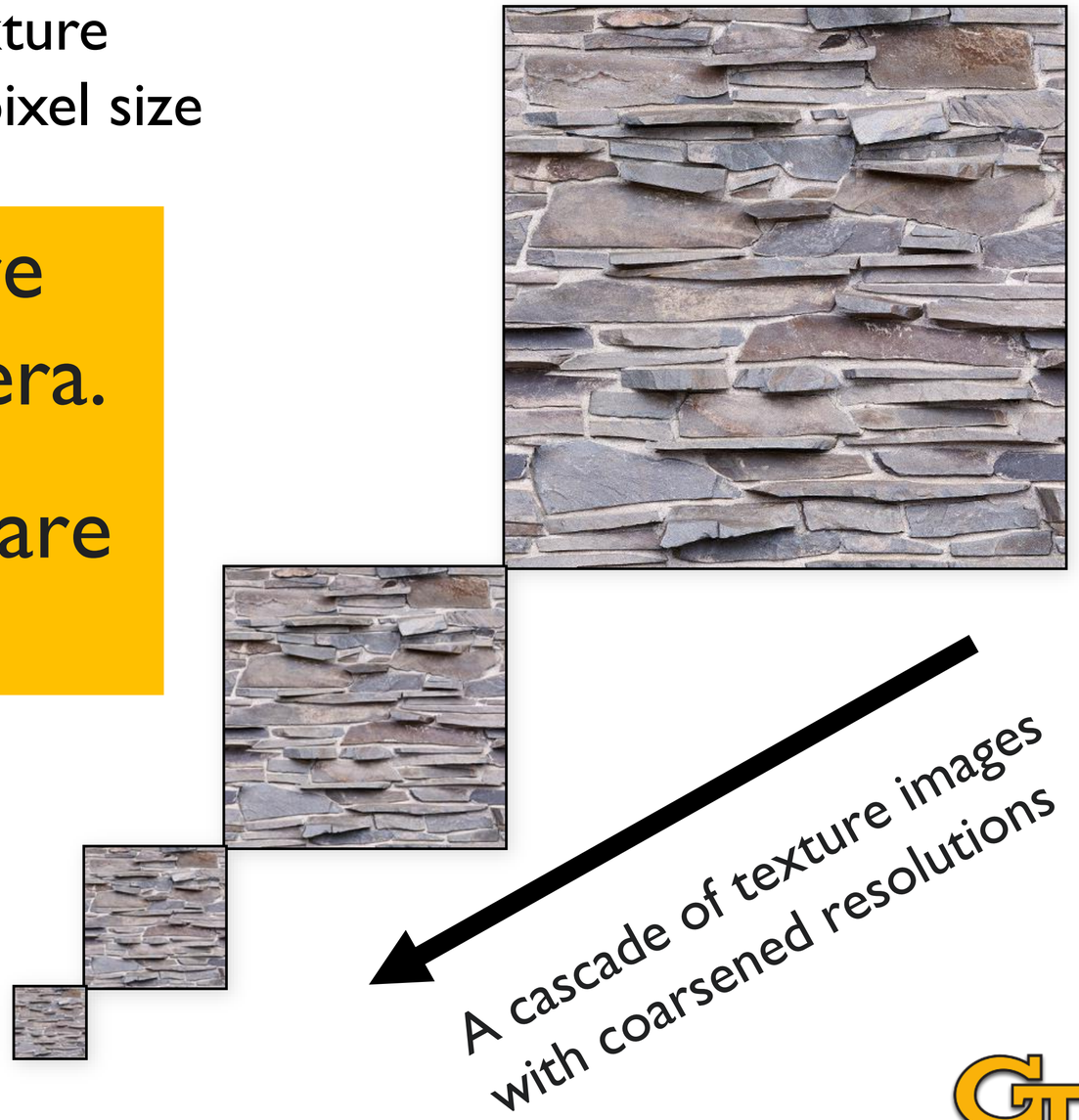
Mipmapping

- Precompute powers-of-2 lower resolutions of texture
- Look up in appropriate level based on projected pixel size

High-resolution mipmap images are used for objects close to the camera.

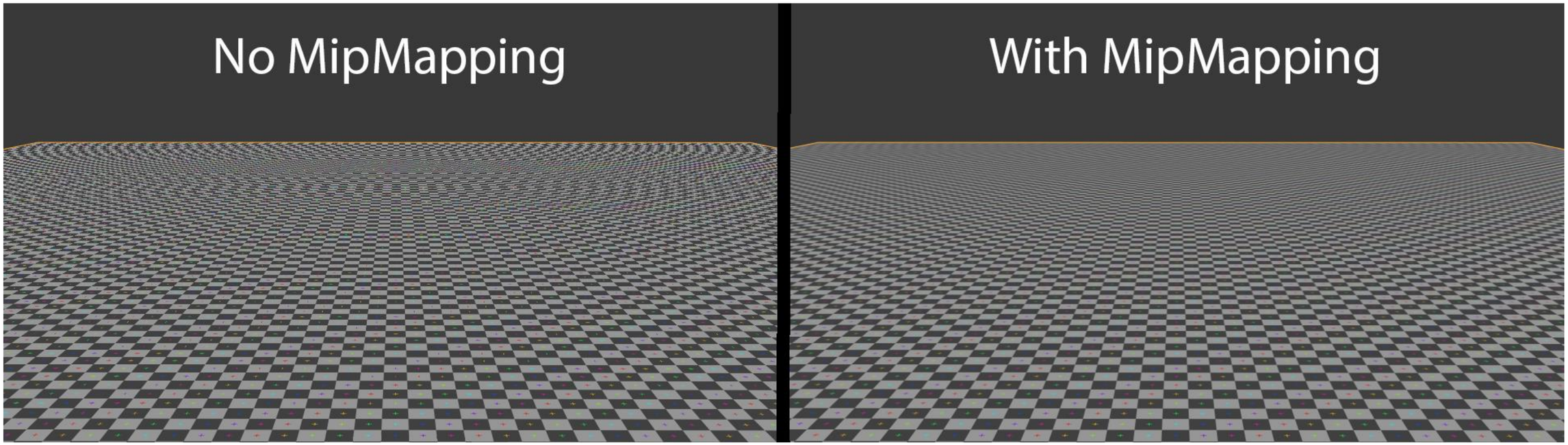
Lower-resolution mipmap images are used for object farther away.

Key Idea: Match the size of rendered pixel and texture pixel to avoid blurring or aliasing artifacts



Side-by-side Comparison

- Mipmap textures are used to increase rendering speed and reduce aliasing artifacts.



Study Plan

Mapping Function

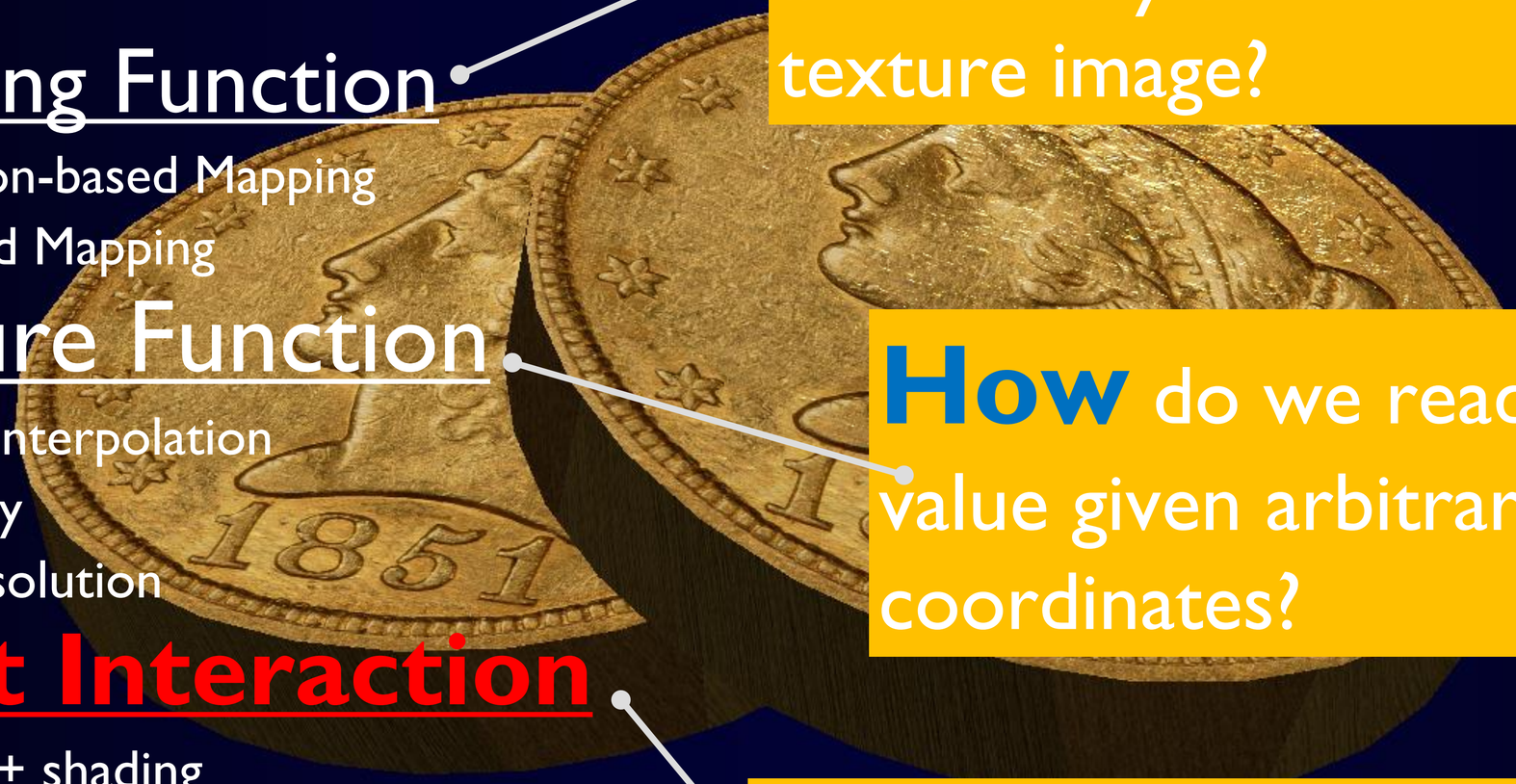
- Projection-based Mapping
- UV-based Mapping

Texture Function

- Bilinear interpolation
- Boundary
- Multi-resolution

Light Interaction

- Texture + shading
- Normal mapping



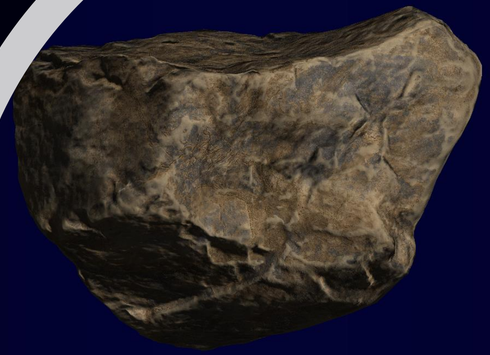
How do we map between an arbitrary surface and a texture image?

How do we read an image value given arbitrary uv coordinates?

What do we read from an image?

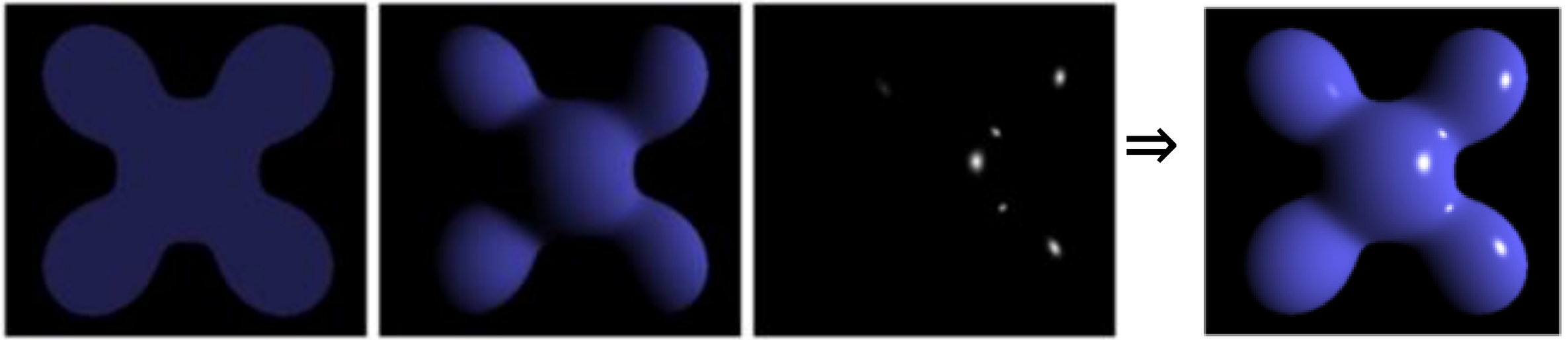
Texture + Shading

When texture mapping meets Phong shading: significantly enhanced surface details in all aspects! ☺



Quick Recap: Phong Shading

Which material parameters in Phong model can be textured?



Ambient | Diffuse | Specular | Final Image

$$\boldsymbol{L}_{Phong} = \sum_{j \in lights} \left(k_a \boldsymbol{I}_a^j + k_d \boldsymbol{I}_d^j \max\left(0, \vec{l}^j \cdot \vec{n}\right) + k_s \boldsymbol{I}_s^j \max\left(0, \vec{v} \cdot \vec{r}^j\right)^p \right)$$

Ambient | Diffuse | Specular

What do we map onto the surface?

- Reflectance (diffuse & specular coeffs., shininess, etc)
- Surface normal (bump mapping)

$$\boldsymbol{L}_{Phong} = \sum_{j \in lights} \left(k_a \boldsymbol{I}_a^j + k_d \boldsymbol{I}_d^j \max\left(0, \vec{l}^j \cdot \vec{n}\right) + k_s \boldsymbol{I}_s^j \max\left(0, \vec{v} \cdot \vec{r}^j\right)^p \right)$$

Ambient Diffuse Specular

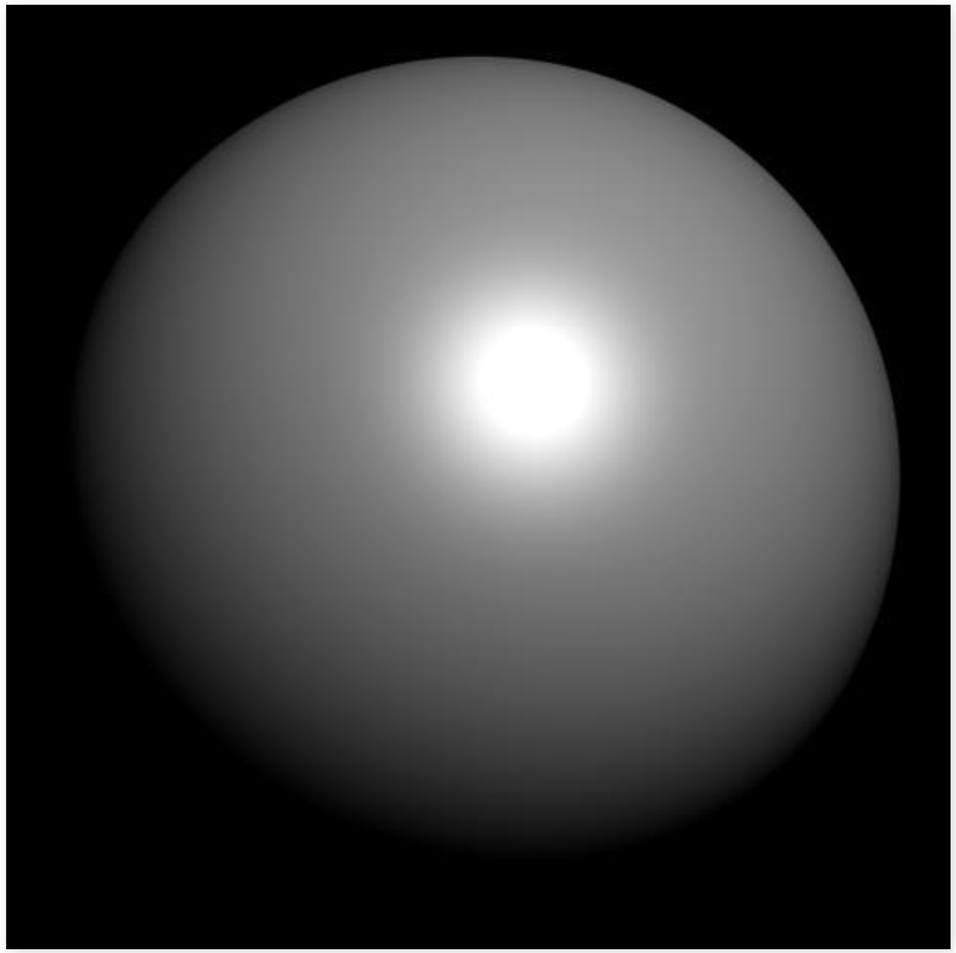
What do we map onto the surface?

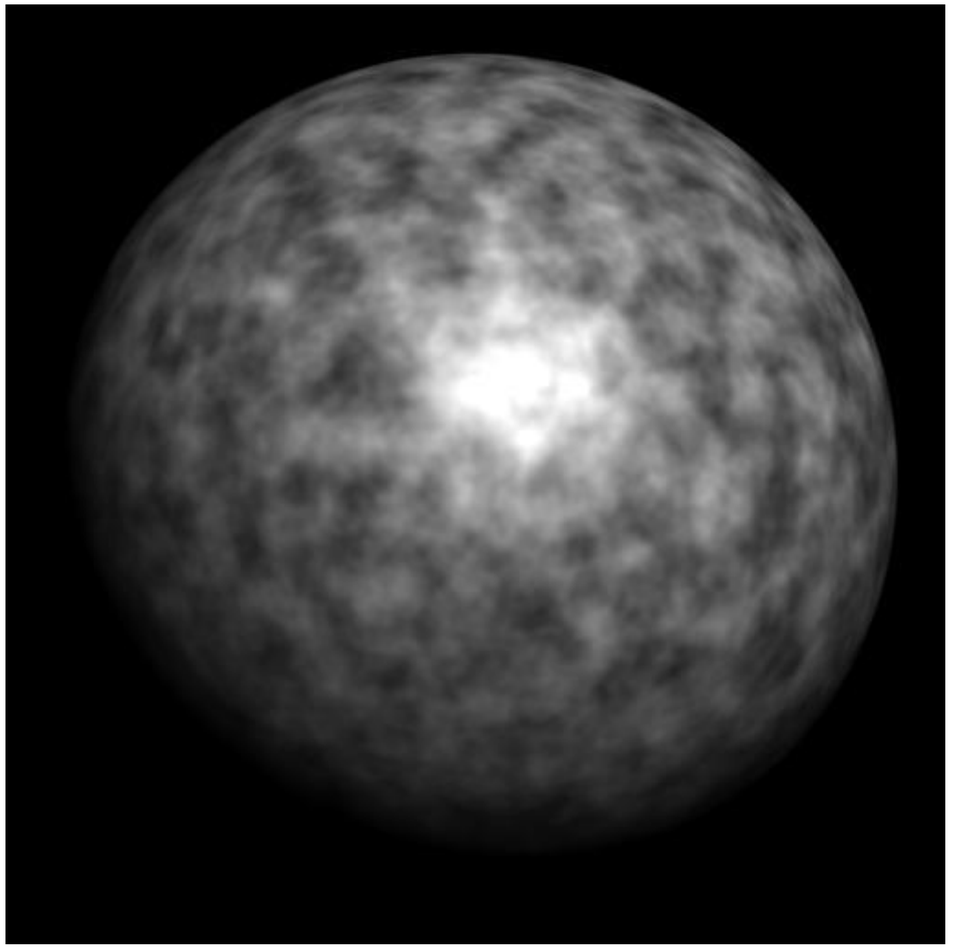
- **Reflectance (diffuse & specular coeffs., shininess, etc)**
- Surface normal (normal mapping)

$$\boldsymbol{L}_{Phong} = \sum_{j \in lights} \left(k_a \boldsymbol{I}_a^j + k_d \boldsymbol{I}_d^j \max\left(0, \vec{l}^j \cdot \vec{n}\right) + k_s \boldsymbol{I}_s^j \max\left(0, \vec{v} \cdot \vec{r}^j\right)^p \right)$$

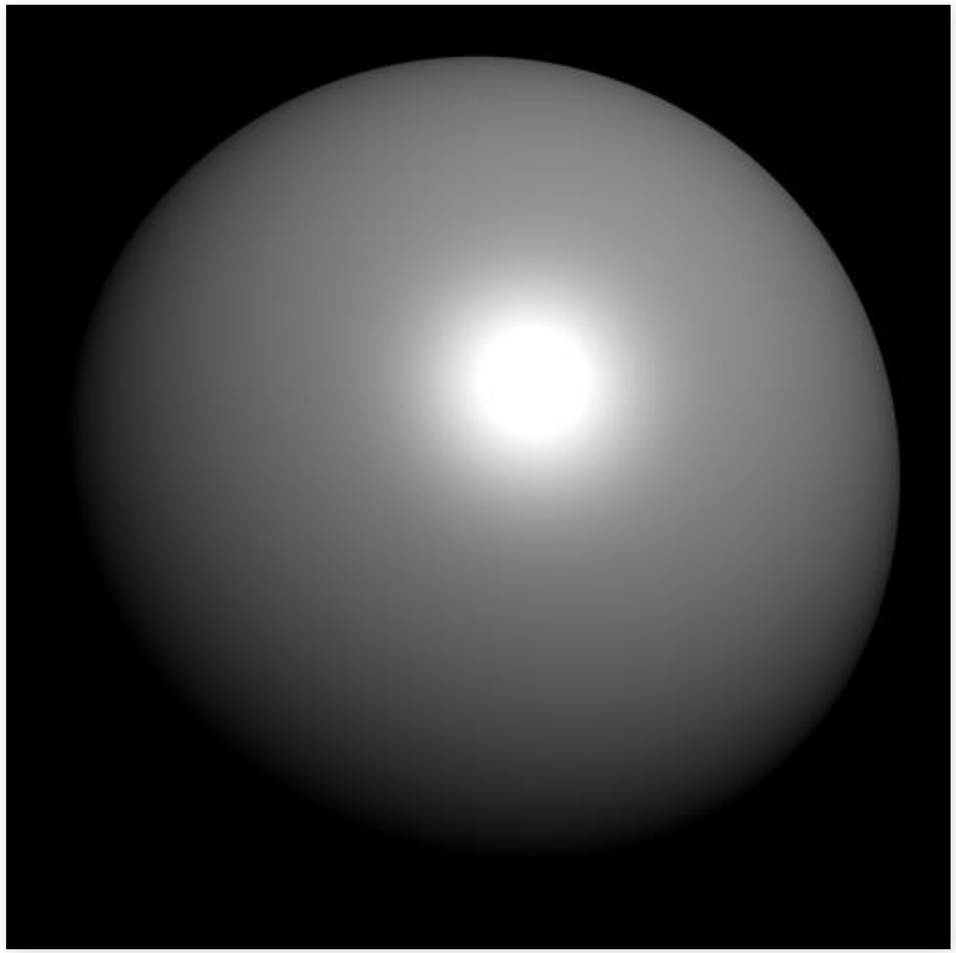
Ambient Diffuse Specular

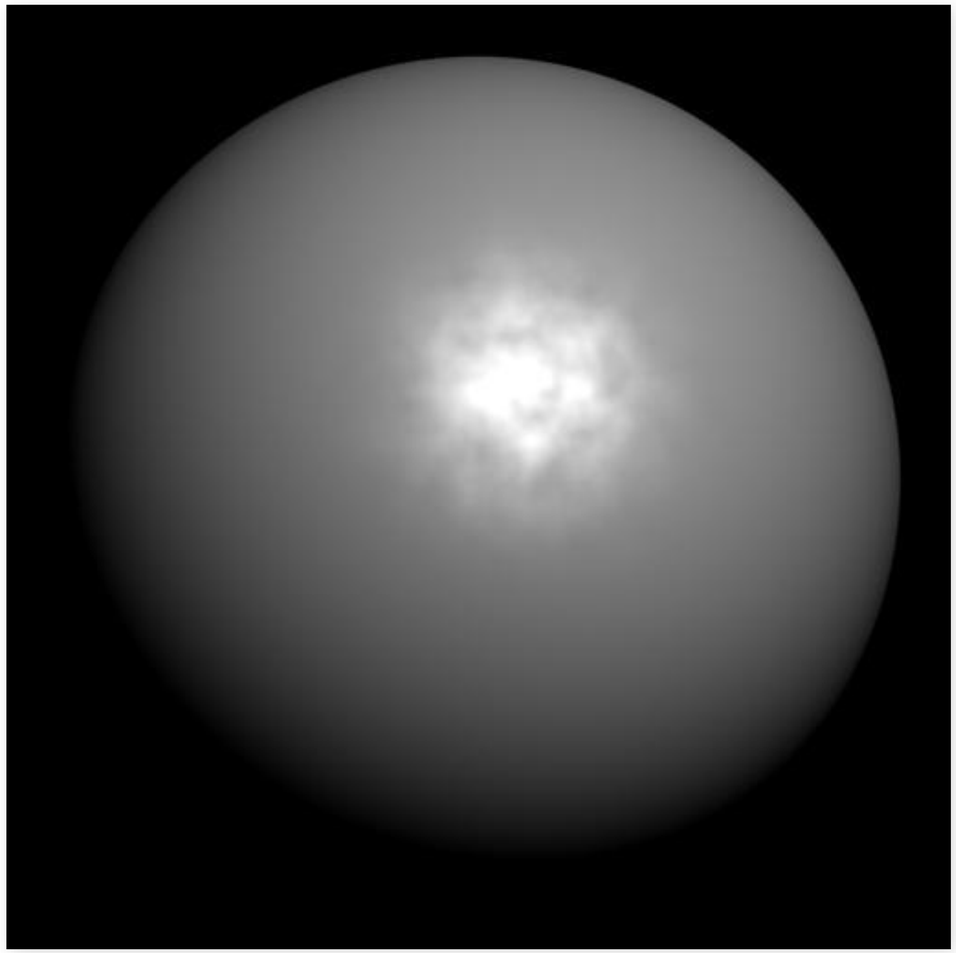
Texturing Diffuse Coefficient k_d





Texturing Specular "Shininess" Coefficient p

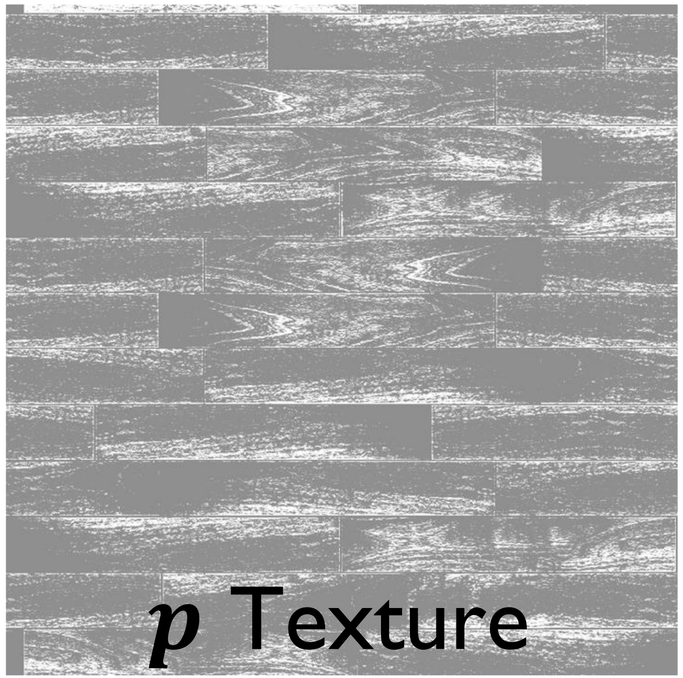





k_d Texture


Rendering


k_d Texture


p Texture


Rendering

Rendering with texture mapping and Phong shading ☺

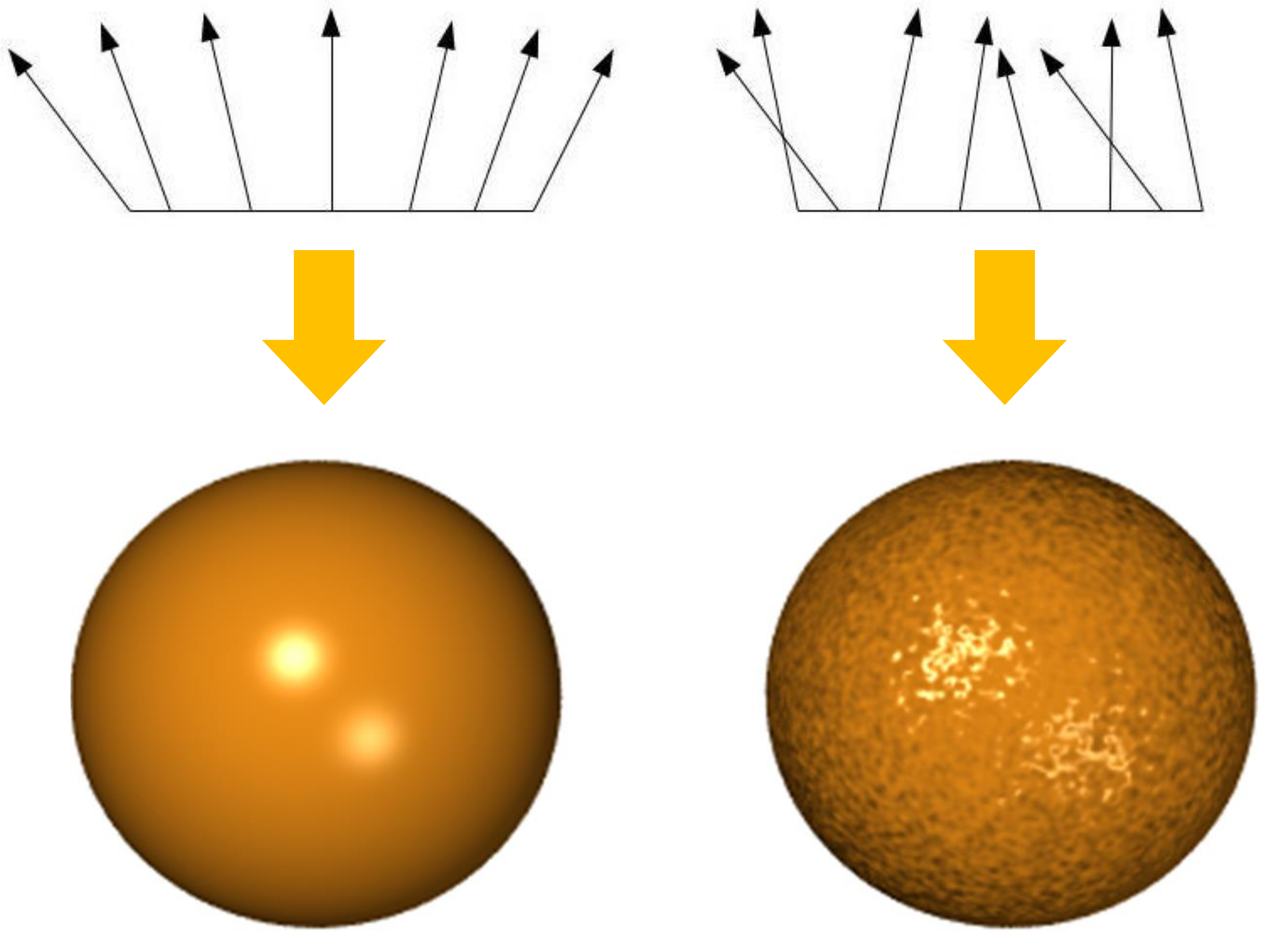
Normal Mapping

What do we map onto the surface?

- Reflectance (color, diffuse + specular coeffs., etc)
- **Surface normal (normal mapping)**

$$\boldsymbol{L}_{Phong} = \sum_{j \in lights} (\underbrace{k_a \boldsymbol{I}_a^j}_{\text{Ambient}} + \underbrace{k_d \boldsymbol{I}_d^j \max\left(0, \vec{l}^j \cdot \vec{n}\right)}_{\text{Diffuse}} + \underbrace{k_s \boldsymbol{I}_s^j \max\left(0, \vec{v} \cdot \vec{r}^j\right)^p}_{\text{Specular}})$$

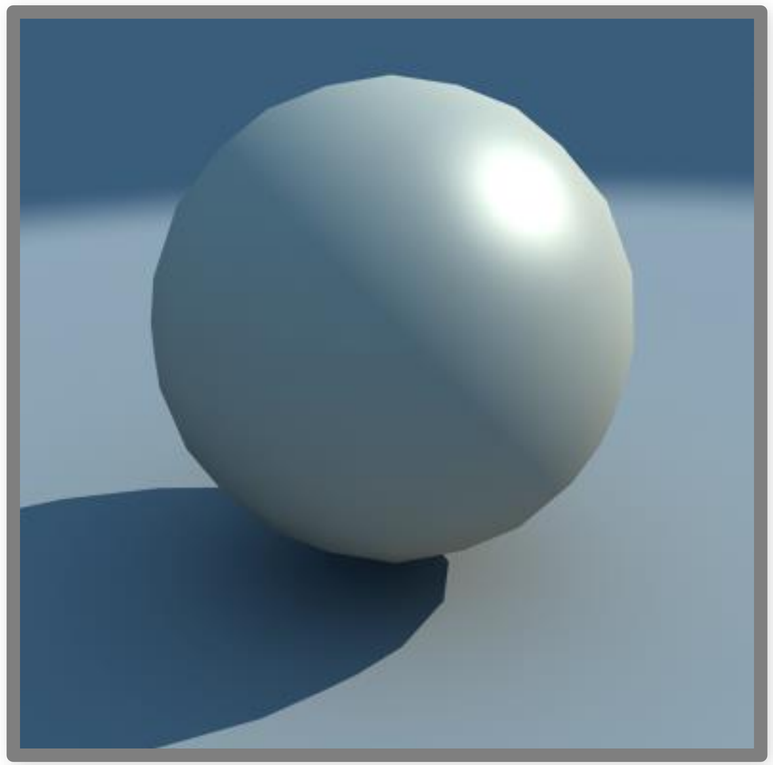
Normal Mapping

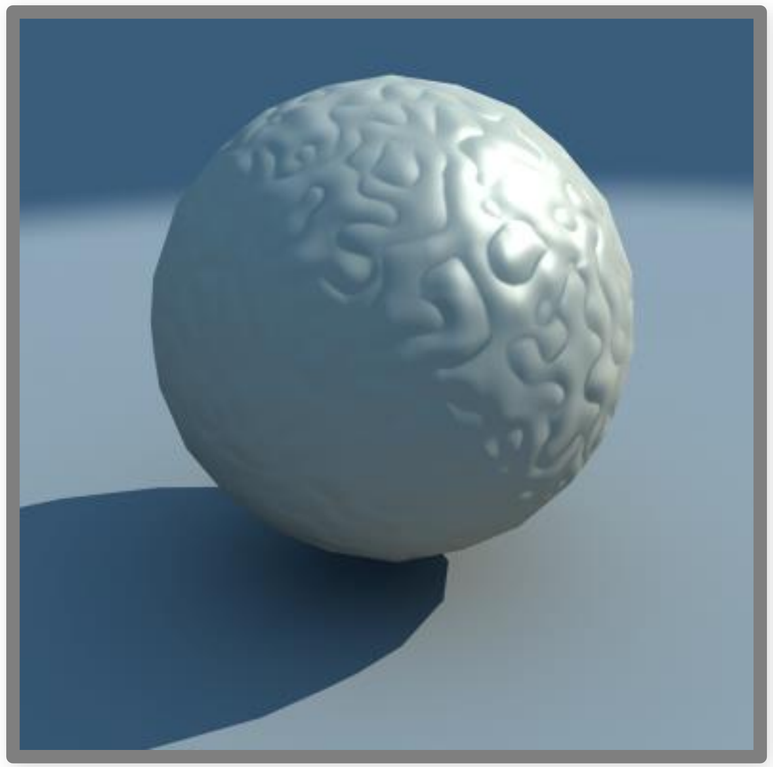


Key Idea: Read normal vectors from a texture image and map them onto the surface

Normal Mapping

- Apply normal perturbation without updating any vertex positions!

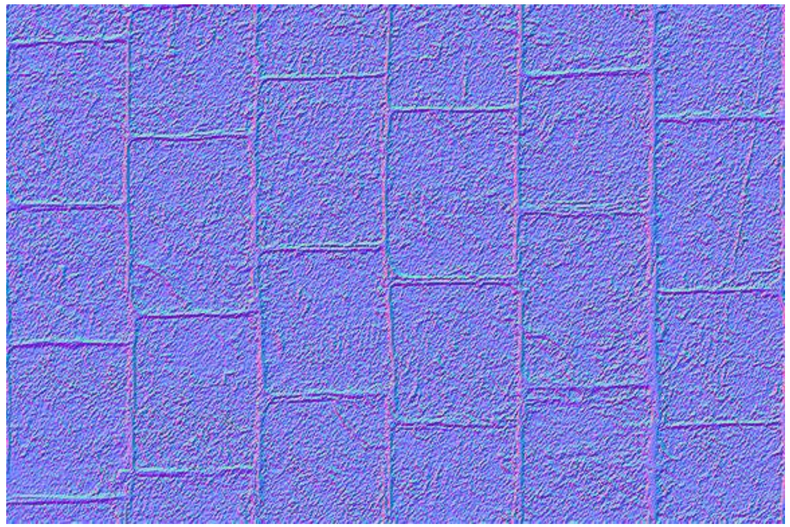




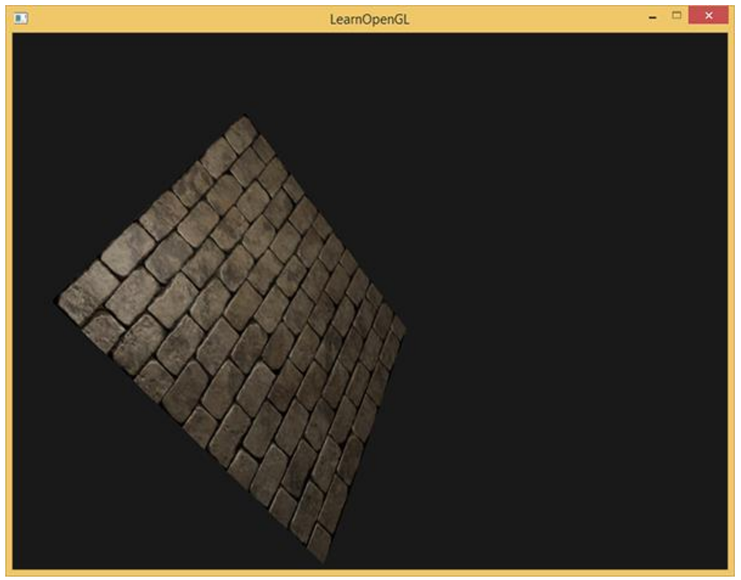
This sounds excellent! But two problems to be addressed:
(1) How do we store normal in a texture?
(2) How do we map normal to a curved surface?

Problem (1): Store Normal in Texture

- **Key Idea:** Encoding direction in color
- Each axis correspond to an RGB color channel
  - e.g., X: -1 to 1 ⟶ Red: 0-1

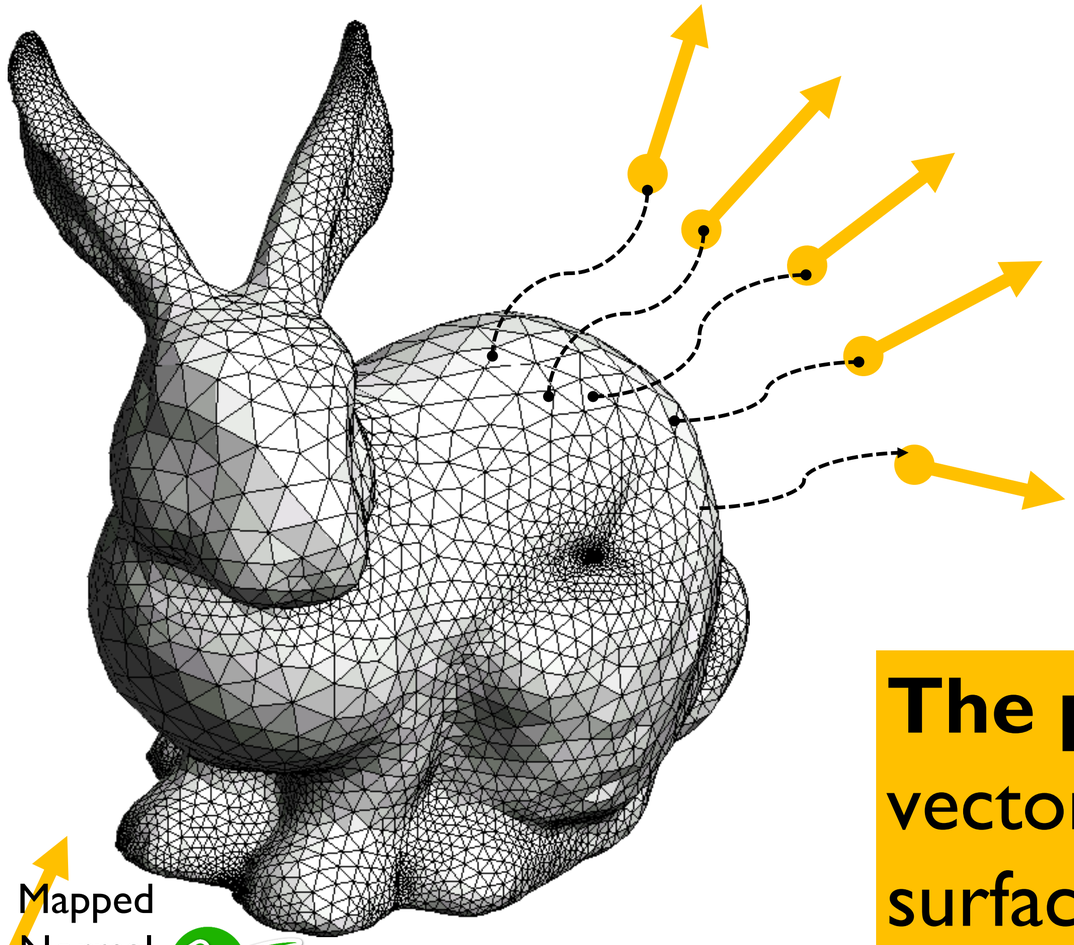


Normal Texture



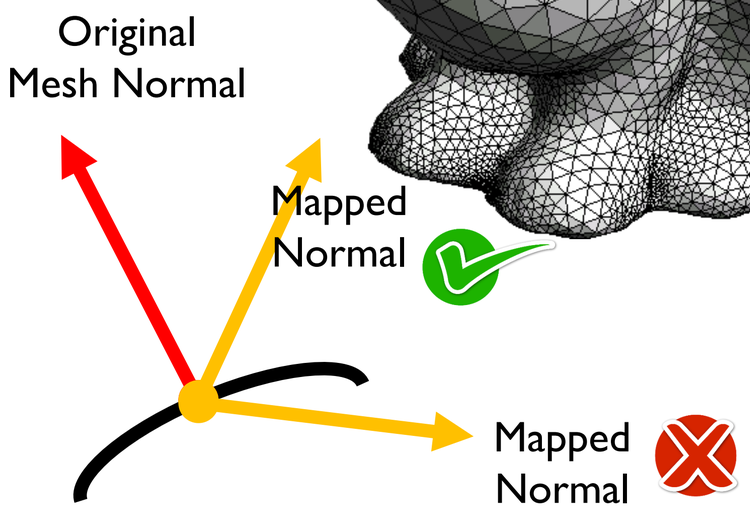
Rendering

Problem (2): Map Normal to Surface



A naïve idea:

Directly store the world-space normal vectors in texture, and map them to surface

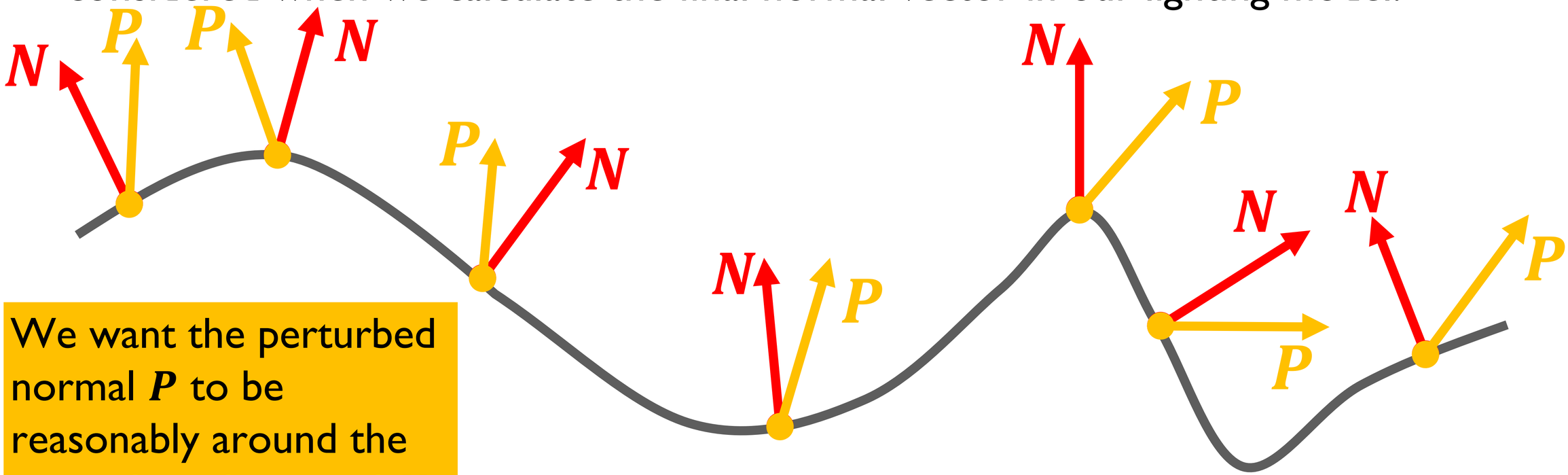


The problem is: the stored normal vectors are not aware of the actual surface it is mapped onto.

It is possible that the texture normal is completely off from the original mesh normal.

Some deeper thoughts about the stored normal vectors …

- The key point is: We only want to **perturb** the original mesh normal, rather than redefining a new normal.
- Therefore, we should only store the perturbation information in the texture, and use both the local normal and texture information to define the perturbed normal vectors
- In other words, both the original mesh normal and the texture normal should be considered when we calculate the final normal vector in our lighting model.

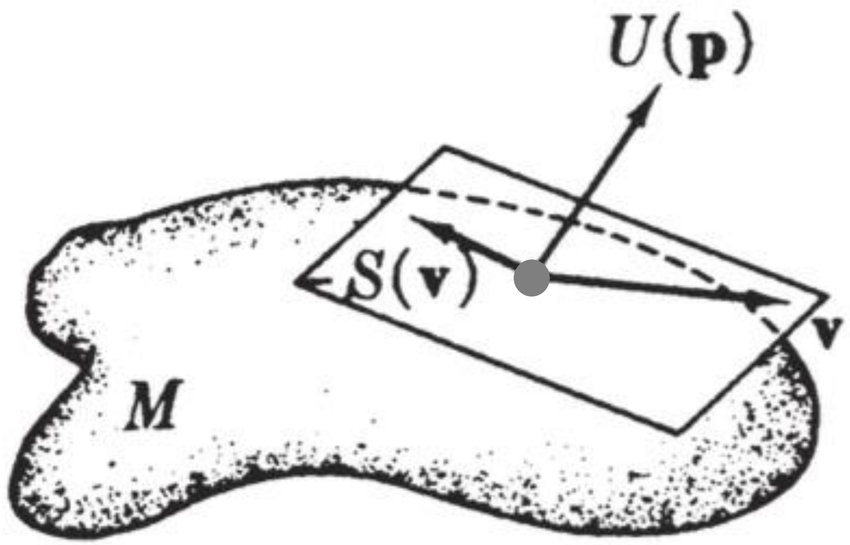


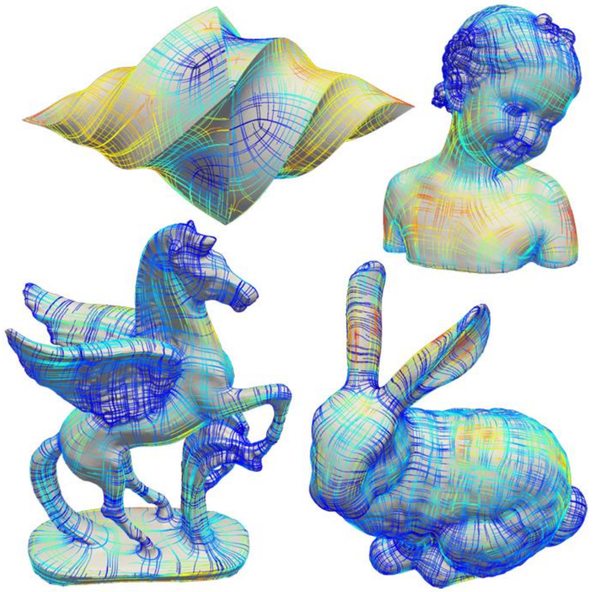
We want the perturbed normal $\boldsymbol{P}$ to be reasonably around the original mesh normal $\boldsymbol{N}$

We want to borrow some ideas from Differential Geometry

- Each point on the surface has a local frame:
  - Two tangential axes define the tangential plane
  - The normal axis define the normal vector

In addition to normal vectors, we can also define tangential vectors on vertices of a triangle mesh





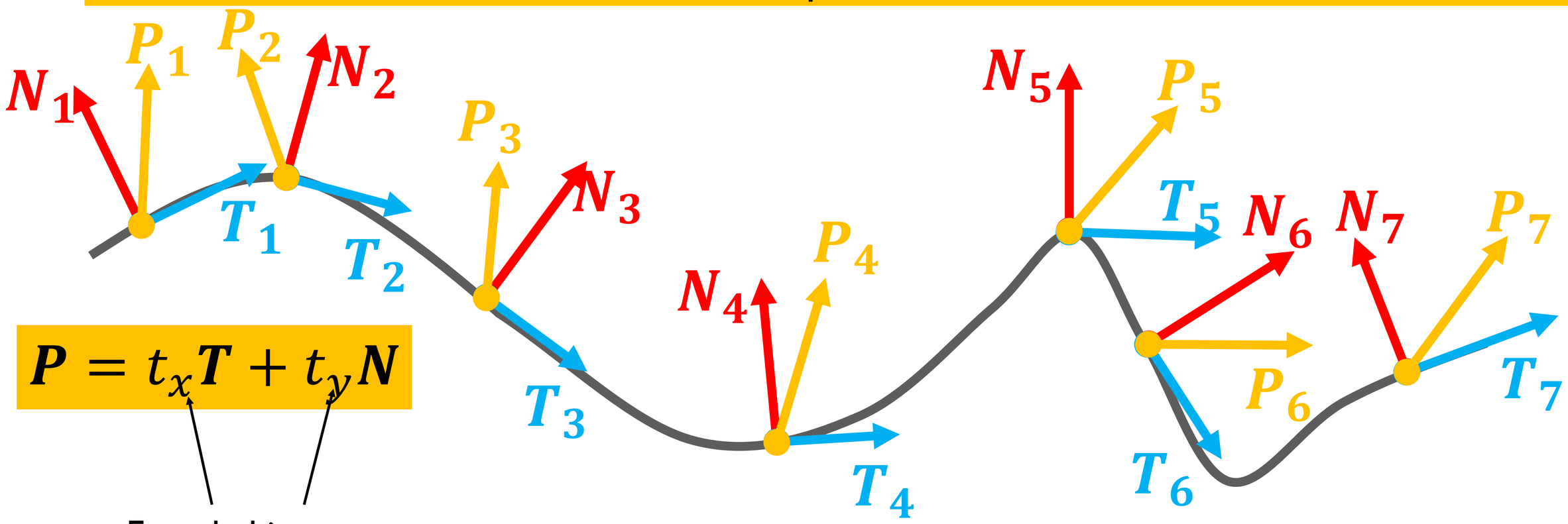
Can we leverage these tangential vectors to define perturbed normal vectors?

Image: http://www.geometry.caltech.edu/pubs/JFHB+15.pdf

Let's explore some ideas from Differential Geometry

- Think about this problem in 2D:
  - We first define a set of local frames on a 2D curve
  - Each local frame is composed of a **normal vector** and a **tangent vector**

Idea: Use these two vectors to encode a perturbed normal vector in the local frame

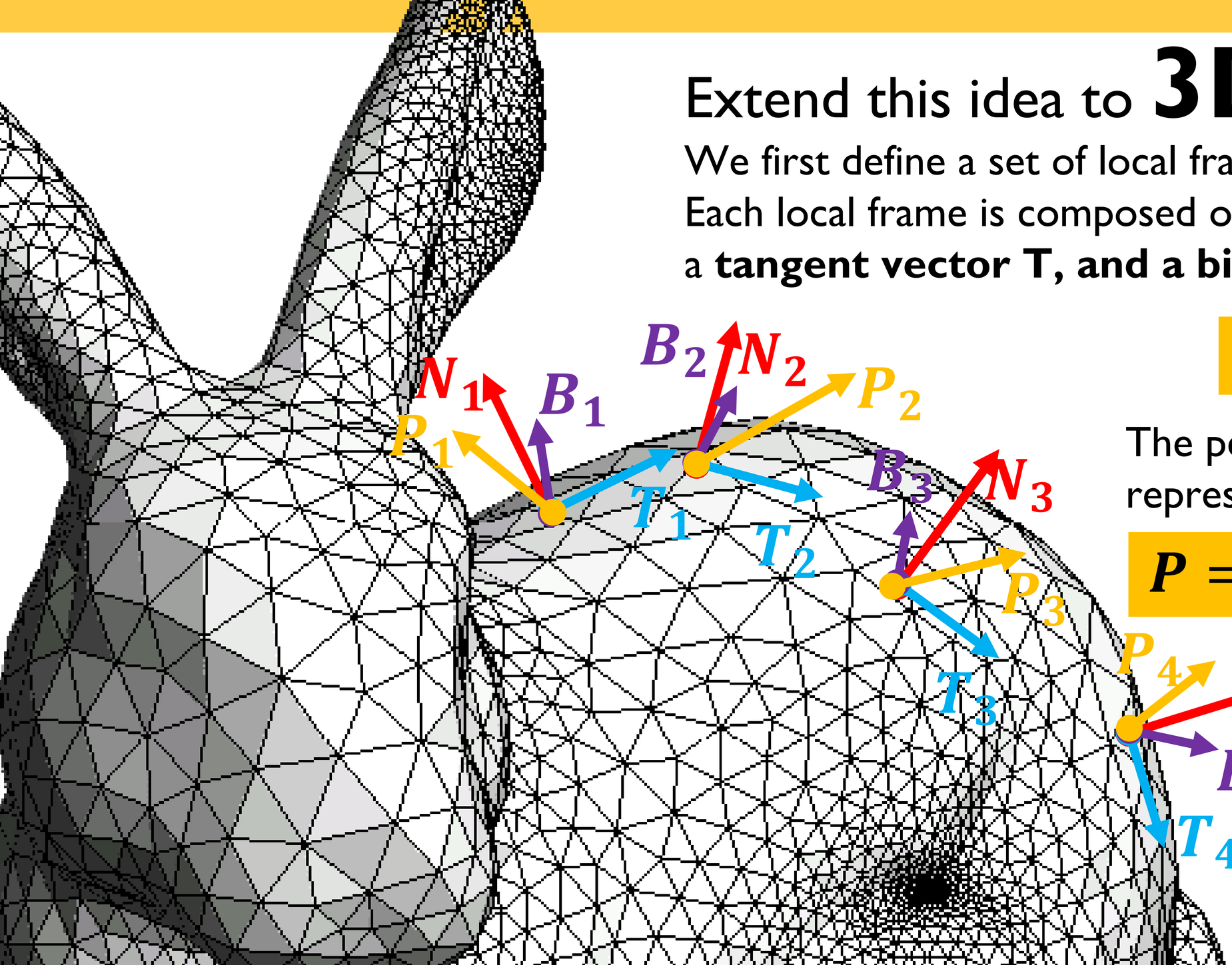


$$\boldsymbol{P} = t_x \boldsymbol{T} + t_y \boldsymbol{N}$$

Encoded in texture

Extend this idea to **3D**

We first define a set of local frames on a mesh surface
Each local frame is composed of a **normal vector N,**
a **tangent vector T, and a bi-tangent vector B**:



$$B = N \times T$$

The perturbed normal can be represented as:

$$P = t_x T + t_y B + t_z N$$

Encoded in texture

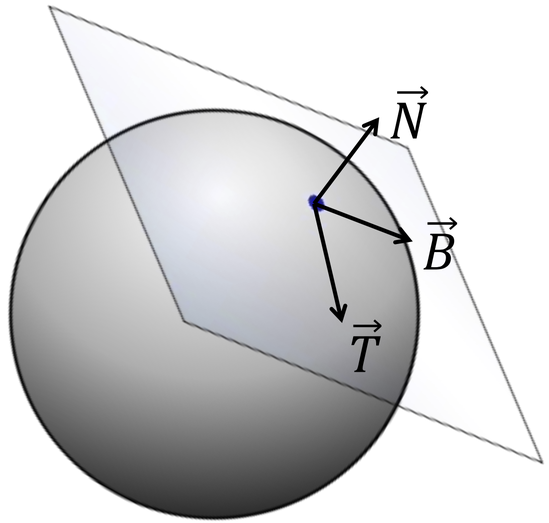
Expressing the Perturbed Normal with $\boldsymbol{T}, \boldsymbol{B}, \boldsymbol{N}$

- The original mesh normal can be represented as $(0,0,1)$ in the local frame defined by $(\boldsymbol{T}, \boldsymbol{B}, \boldsymbol{N})$, that is

$$\boldsymbol{N} = 0\boldsymbol{T} + 0\boldsymbol{B} + 1\boldsymbol{N}$$

- A perturbed normal can be represented by combining the stored normal vector (x, y, z) in the normal texture and the local frame axes $\left(\vec{\boldsymbol{T}}, \vec{\boldsymbol{B}}, \vec{\boldsymbol{N}}\right)$, that is

$$\boldsymbol{P} = t_x\boldsymbol{T} + t_y\boldsymbol{B} + t_z\boldsymbol{N}$$



Calculate Perturbed Normal with TBN and Normal Texture

- In GLSL, the expression of **P** is implemented with matrix-vector multiplication

$$\boldsymbol{P} = t_x\boldsymbol{T} + t_y\boldsymbol{B} + t_z\boldsymbol{N}$$

$$\begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix} \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

Perturbed normal

TBN matrix

Texture vector

Assemble a mat3 in GLSL by specifying its three columns as ***T***, ***B***, and ***N***

Use the assembled TBN matrix to transform the vector read from texture to calculate the perturbed normal vector

Normal Mapping Pseudocode

Input: N, T, frag_pos
Output: frag_color
In the fragment shader:

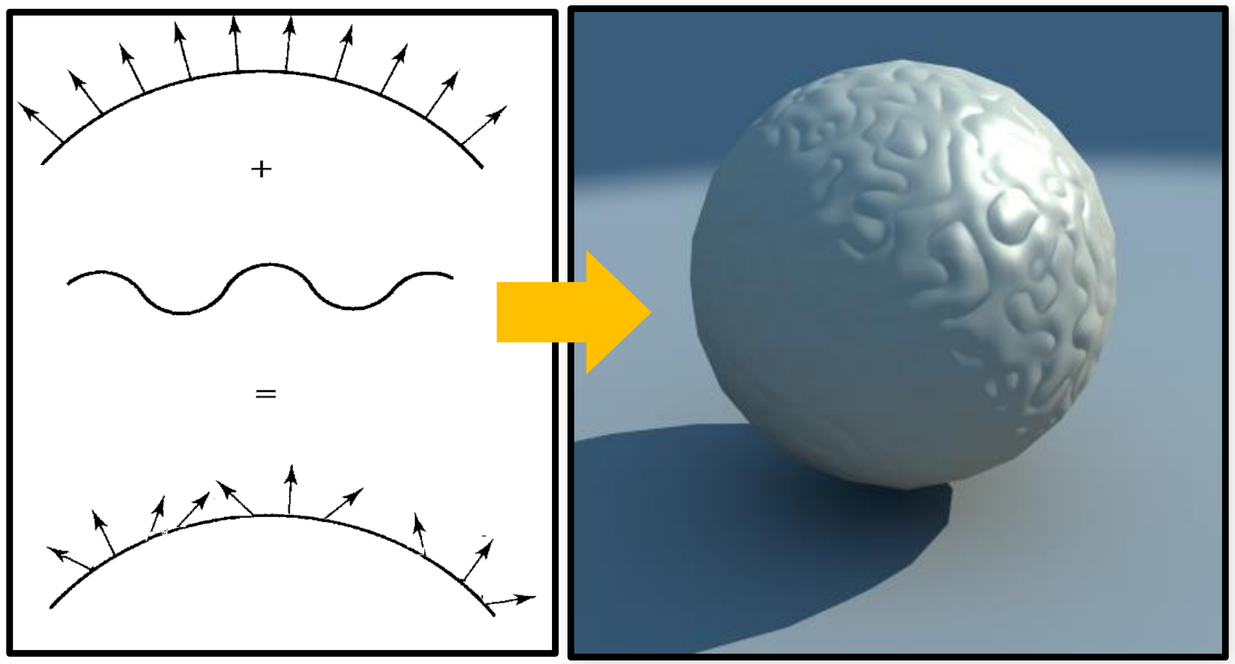
- Read ***N*** and ***T*** from the vertex shader
- Calculate ***B*** using ***N*** and ***T***: $\boldsymbol{B} = \boldsymbol{N} \times \boldsymbol{T}$
- Assemble transform matrix TBN=(***T***,***B***,***N***)
- Read (tx,ty,tz) from the texture by querying uv and map from [0,1] to [-1,1]
- Multiply the TBN matrix with (tx,ty,tz) to calculate the perturbed normal
- Use the perturbed normal to calculate frag_color with Phong model

Normalize vectors whenever needed!

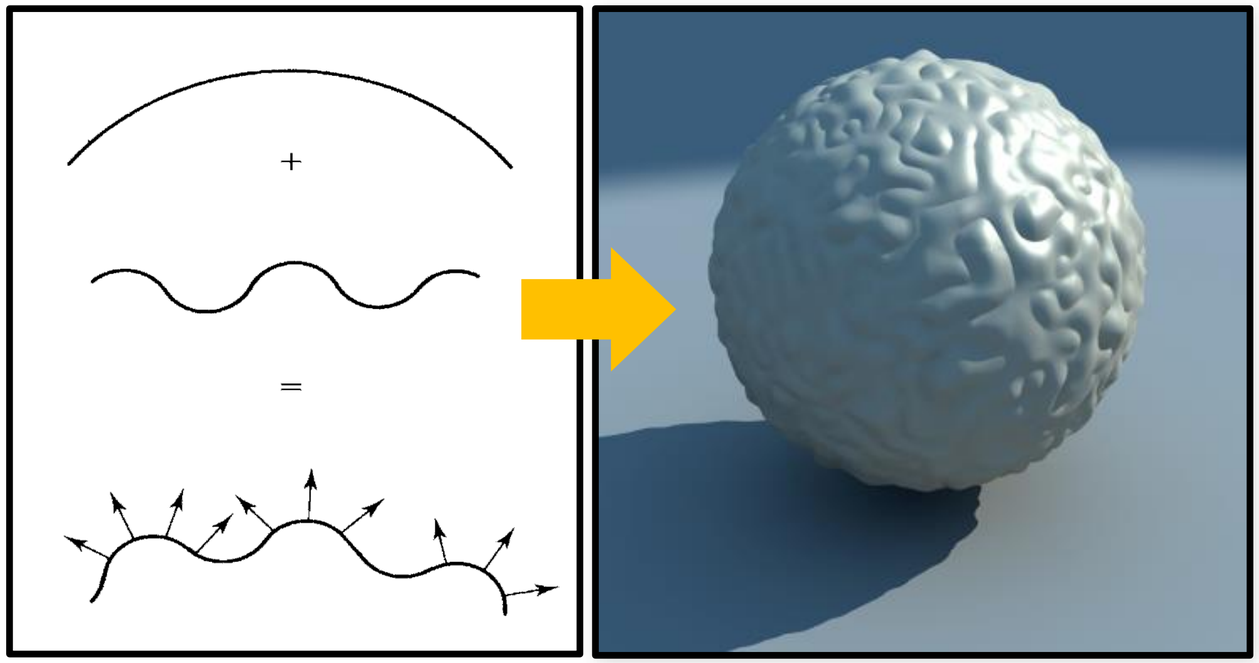
Texture Mapping + Normal Mapping + Phong Lighting

Other Texture Mapping Techniques: Displacement Mapping

- Displacement Mapping
  - Encode a displacement distance in the texture map



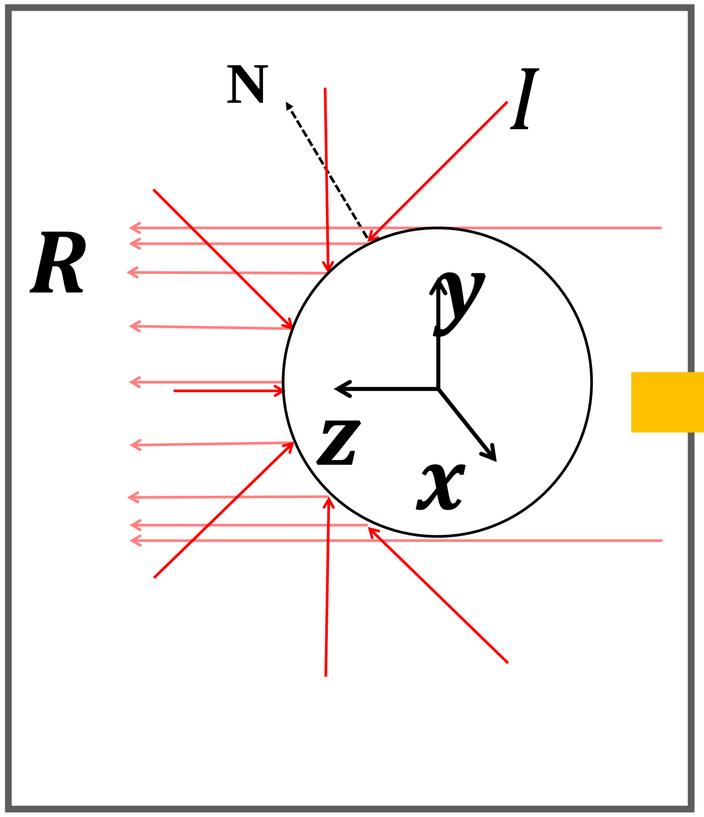
Normal Mapping



Displacement Mapping

Other Texture Mapping Techniques: Environment Mapping

- Create a photograph of a chrome sphere (light probe) containing the intensities of the environmental light shone onto the sphere from almost all directions; and then map the light onto the object surface



Light Probe

Environment Mapping

Additional Reading Materials

- Learn OpenGL Tutorial:

https://learnopengl.com/Advanced-Lighting/Normal-Mapping
http://ogldev.atspace.co.uk/www/tutorial26/tutorial26.html

- "Rendering with Natural Light"

http://www.pauldebevec.com/RNL/

- "Fiat Lux"

http://www.pauldebevec.com/FiatLux/

- History of reflection mapping

http://www.pauldebevec.com/ReflectionMapping/