Textures

Computer Graphics and Visualization

Stefan Johansson
Department of Computing Science
Fall 2016

The Good Dinosaur, Pixar (2015)

“This is Bond, James Bond”, Luc Begin (2014?)

Star Wars Battlefront, DICE (2015)

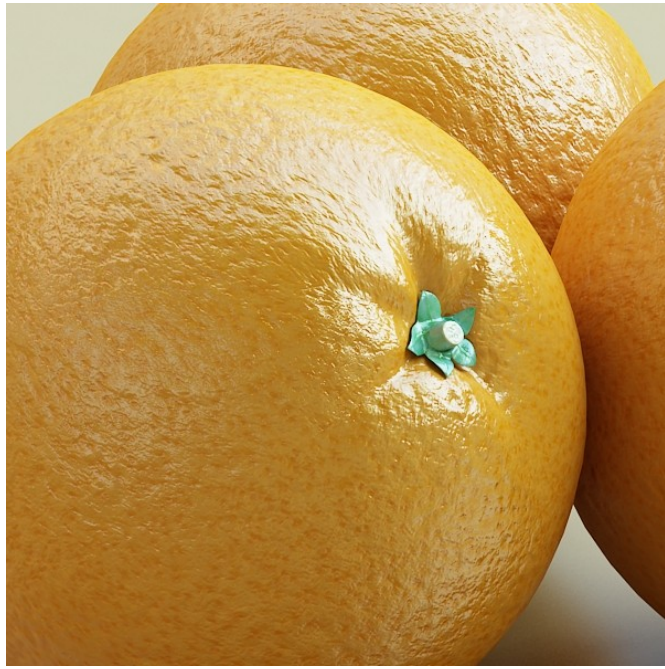

VKModels (2012)

Modelling an Orange

- Consider the problem of modelling an orange (the fruit)
- Start with an orange-colored sphere
  - Too simple
- Replace sphere with a more complex shape
  - Does not capture surface characteristics (small dimples)
  - Takes to many polygons to model all the dimples

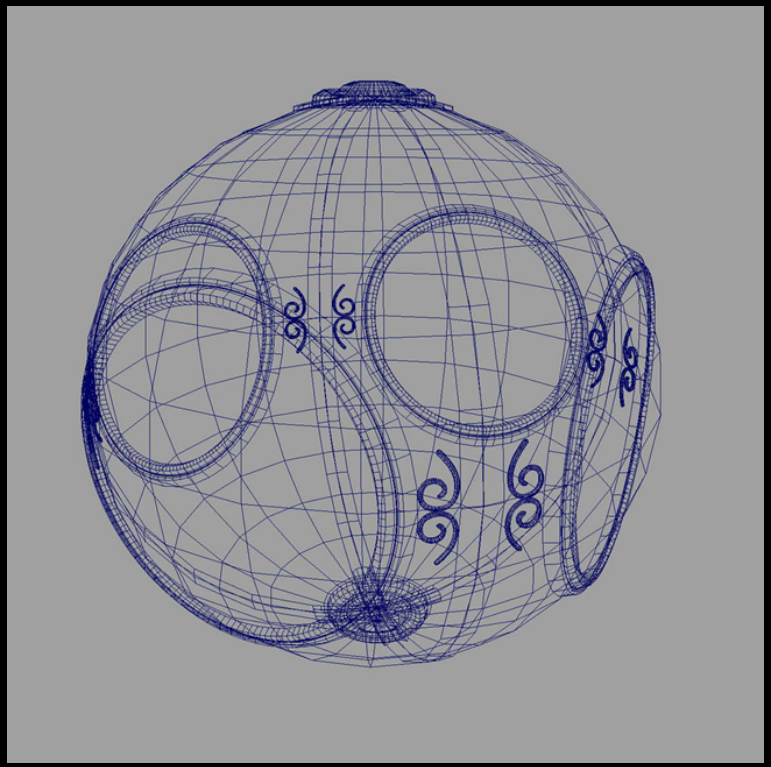
Modelling an Orange

- Take a picture of a real orange and map it onto a simple geometric model
  - This process is known as *texture mapping*
- Still might not be sufficient because resulting surface will be smooth
  - Need to change local shape
  - *Bump mapping*



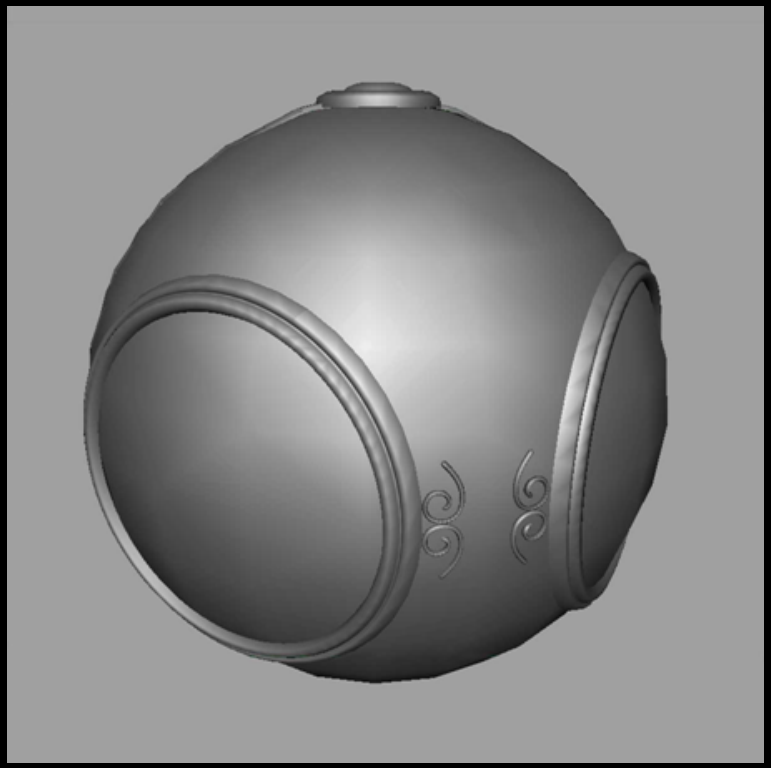
Four Types of Mapping

- **Texture Mapping**
  - Uses images that are mapped onto the polygons
- **Environment Mapping** (or reflection mapping)
  - Uses a picture of the environment as texture map
  - Allows simulation of highly specular surfaces (mirrors)
- **Bump Mapping**
  - Emulates altering normal vectors during the rendering process to simulate small changes of the surface
- **Displacement Mapping**
  - Do actual displacements of the surface

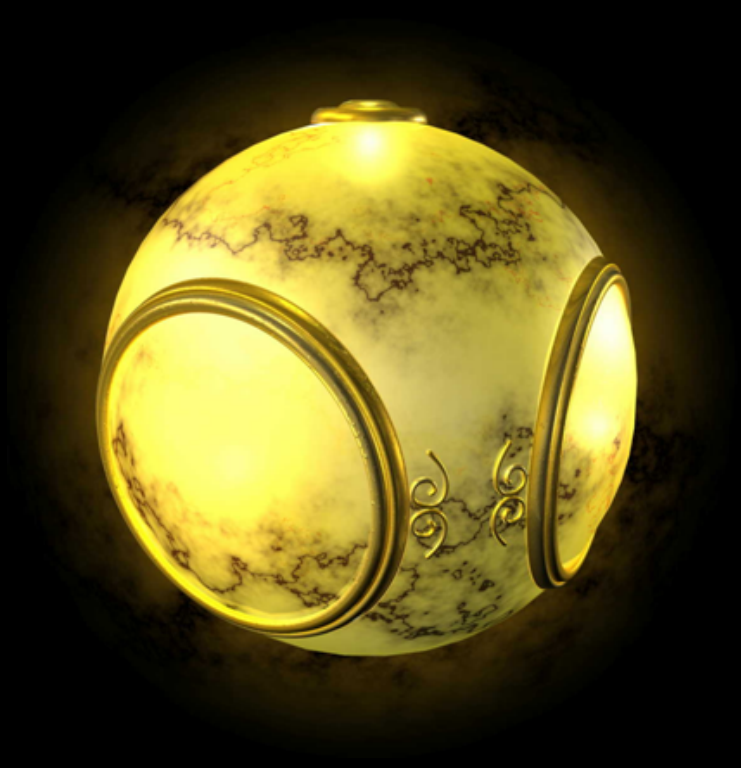


Wireframe Model

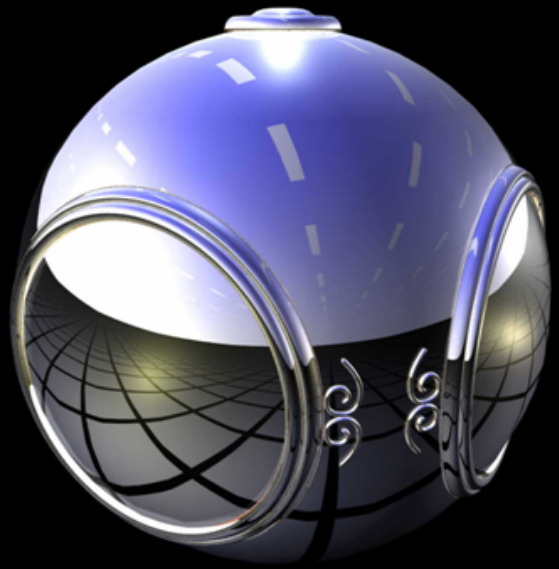
Stefan Johansson
Department of Computing Science



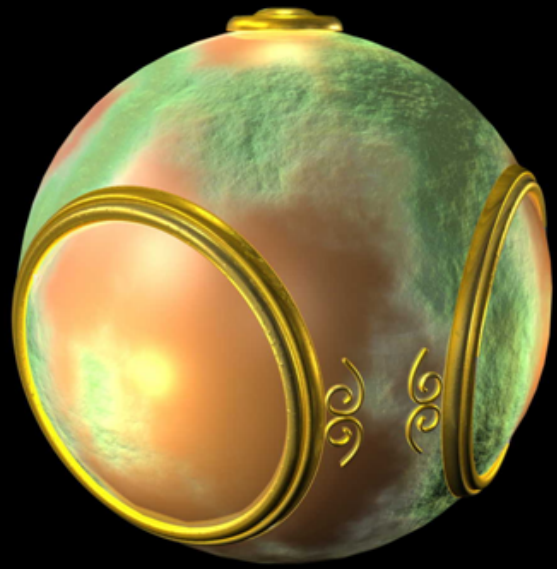
Geometric Model



Texture Mapped



Environment Mapped



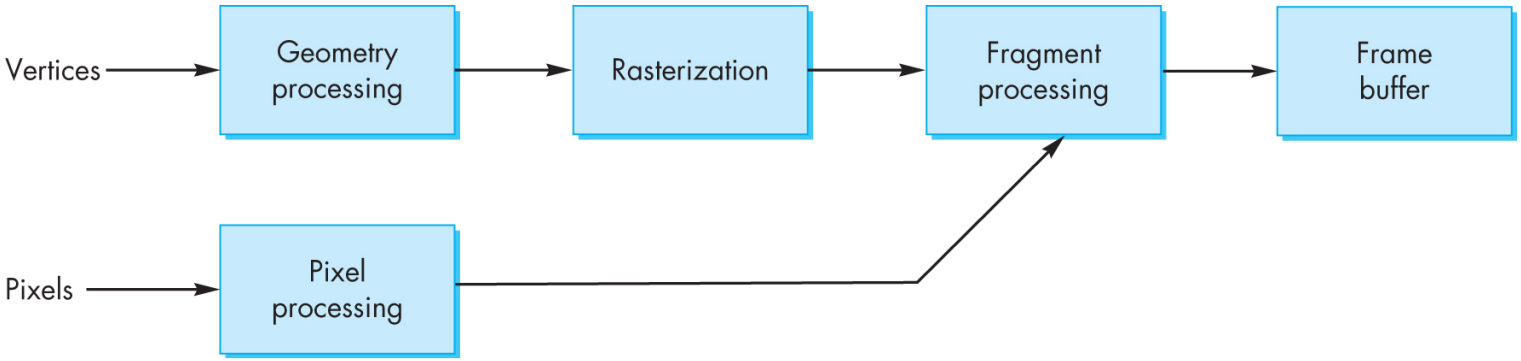
Bump Mapped

Texture Mapping

Texture mapping is implemented at the end of the rendering pipeline

- Efficient since few polygons left after the clipper

The texture mapping pipeline:



Is it Simple?

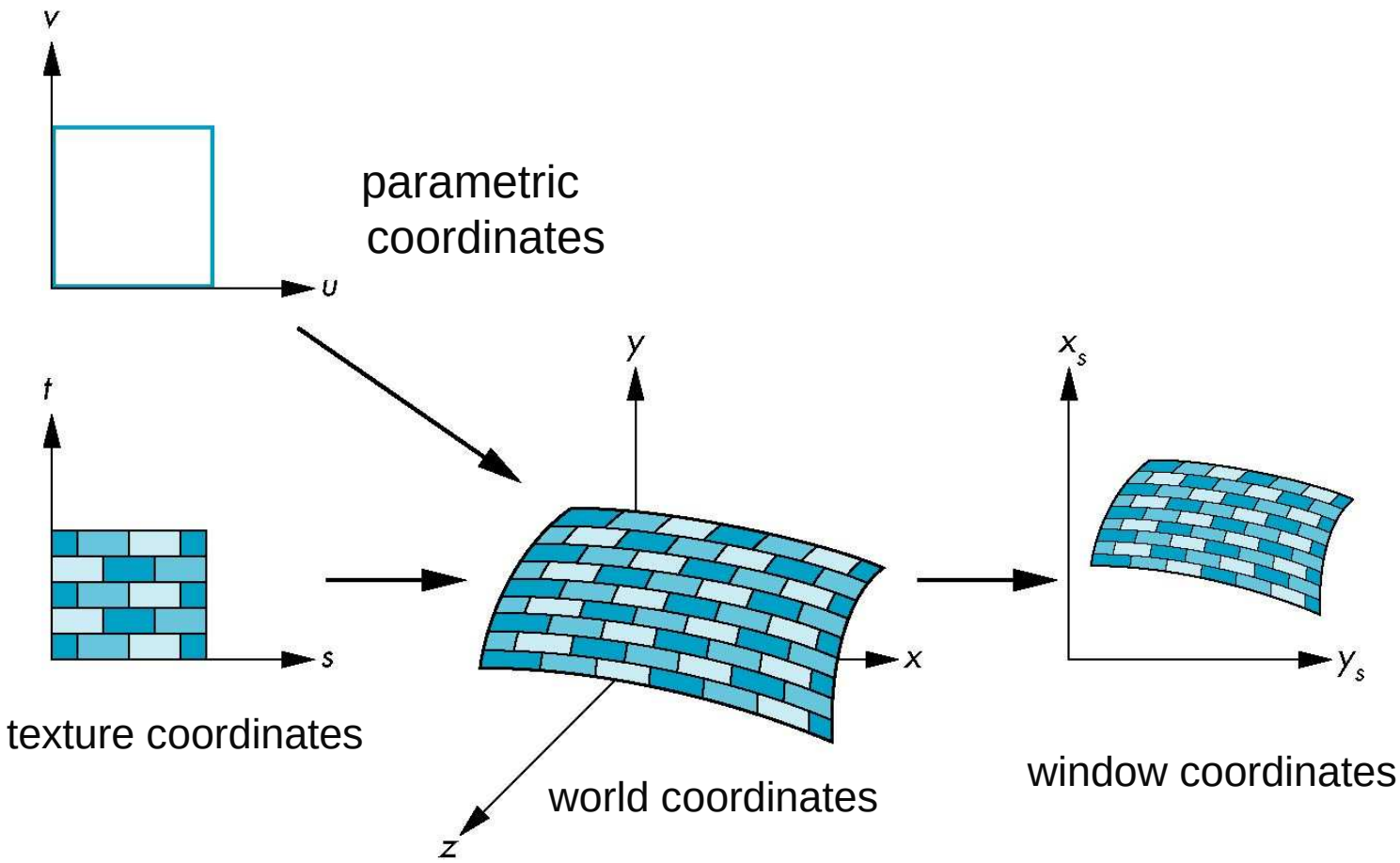
Although the idea is simple—map an image onto a surface—there are 3 or 4 coordinate systems involved

- *Image Coordinates* $(x_{\text{image}}, y_{\text{image}})$
  - Used to identify points in the image to be mapped
- *Texture Coordinates* (s, t) or (u, v)
  - May be used to model Parametric curves and surfaces
- *Object* or *World Coordinates* (x, y, z)
  - Conceptually, where the mapping takes place
- *Window Coordinates* (x_s, y_s)
  - Where the final image is really produced

Texture space (s, t)

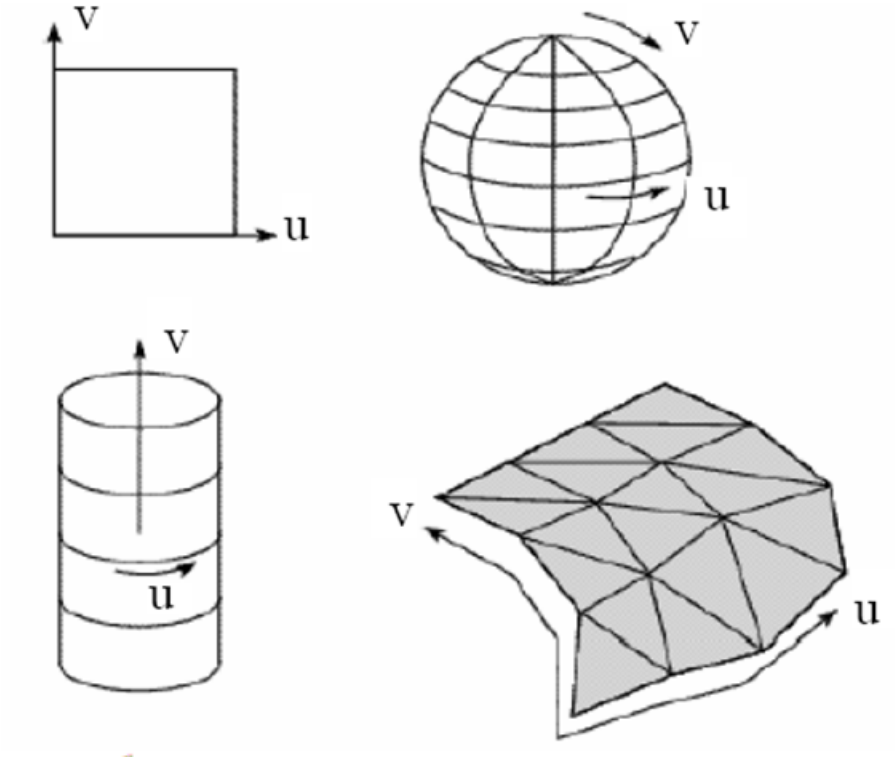
- A texture lives in its own image coordinates, parameterized by (s, t), $s \in [0, 1]$ and $t \in [0, 1]$
- Each pixel of the texture image is called a *texel*
- If an image has the size $m \times n$ (width$\times$height), then each *texel* is of size $(1/m, 1/n)$ in the *st*-space $\Rightarrow s = x_{\text{image}}/m$ and $t = y_{\text{image}}/n$
- The texel value should be located at the center of the texel
- m and n are usually powers of 2 for ease of computation.

Coordinate Systems



Mapping

- Textures can be wrapped around many different surfaces



Mapping Functions

- Seems that we need to find the map between the texture coordinates to a point on a surface

$$x = x(s,t)$$
$$y = y(s,t)$$
$$z = z(s,t)$$

- But we really want the inverse
  1. Given a pixel, we want to know to which point on an object it corresponds
  2. Given a point on an object, we want to know to which point in the texture it corresponds

Mapping Functions

- Seems that we need to find the map between the texture coordinates to a point on a surface

$$x = x(s,t)$$
$$y = y(s,t)$$
$$z = z(s,t)$$

- But we really want the inverse
  1. Given a pixel, we want to know to which point on an object it corresponds
  2. Given a point on an object, we want to know to which point in the texture it corresponds
- New maps: $s = s(x,y,z)$ and $t = t(x,y,z)$
- These maps are, in general, hard to find

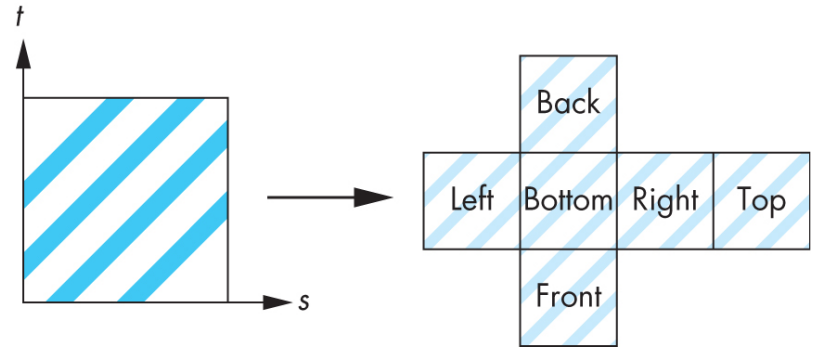
Two-Part Mapping

- First map the texture onto a simple *intermediate* geometry that can be parameterized
  - Cube/Box
  - Cylinder
  - Sphere

Two-Part Mapping

- First map the texture onto a simple *intermediate* geometry that can be parameterized
  - Cube/Box
  - Cylinder
  - Sphere

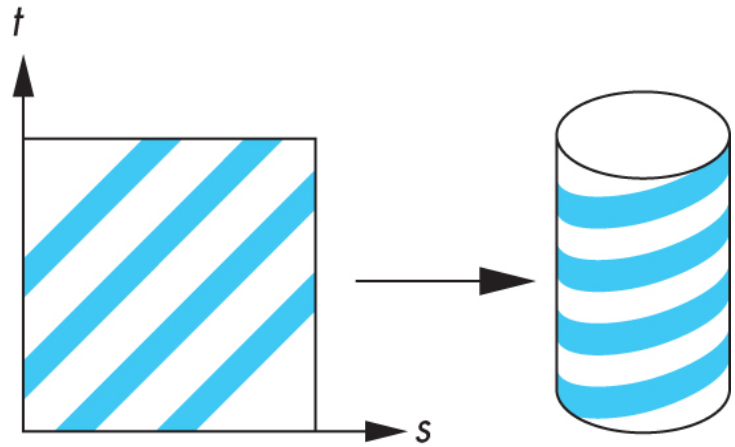
- Box mapping
  - Easy with orthographic projections



Cylindrical Mapping

- Parametric cylinder

$$x = r\cos(2\pi s)$$
$$y = r\sin(2\pi s)$$
$$z = t/h$$



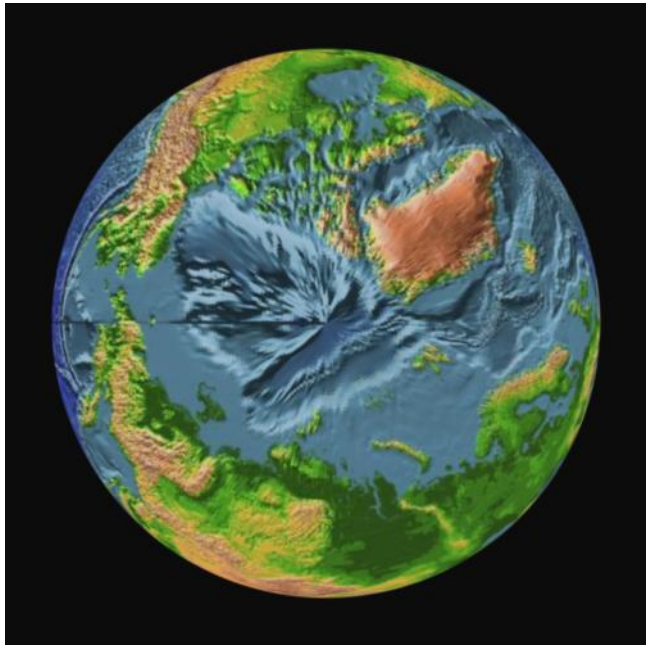
- Map from
  - Square $[0,1] \times [0,1]$ in (s,t) texture space to
  - Cylinder of radius r, height h in (x,y,z) world coordinates

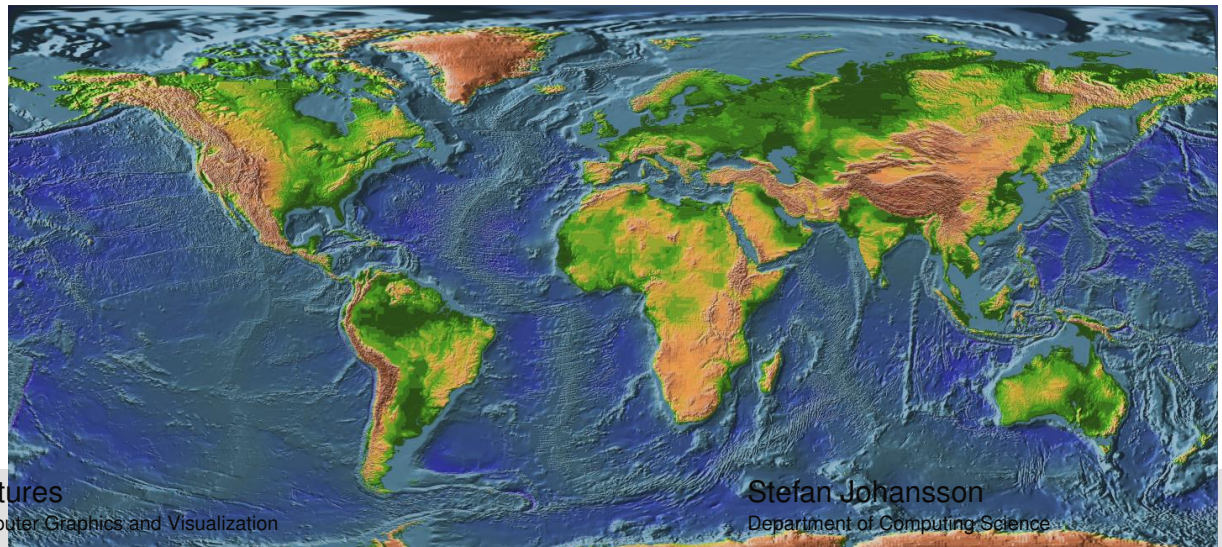
Spherical Mapping

- Parametric sphere

$$x = r\cos(2\pi s)$$
$$y = r\sin(2\pi s)\cos(2\pi t)$$
$$z = r\sin(2\pi s)\sin(2\pi t)$$

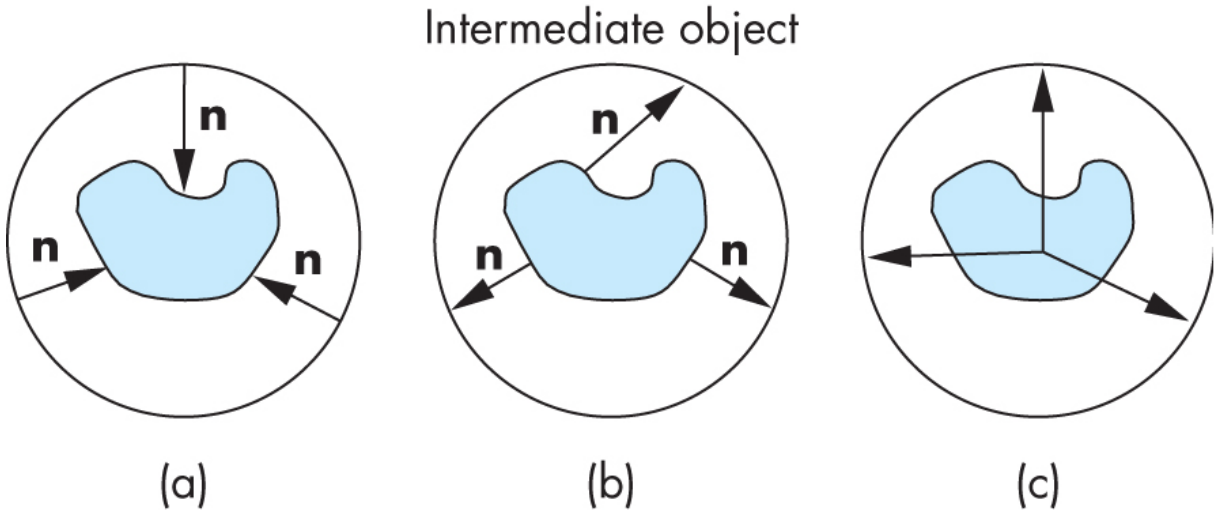
- Bad distortions at the poles





Two-Part Mapping—Second Step

- Maps the texture from the intermediate object to object surface
- Three possibilities
  - (a) Normals from intermediate to actual
  - (b) Normal from actual to intermediate
  - (c) Vectors from center of intermediate



Two-Part Mapping—How to example

We consider:

- Sphere as intermediate object
  - Assume sphere is larger than object
  - r is the radius
  - $P = (x, y, z)$ is a point on the surface
  - Sphere and object center at $(0, 0, 0)$
- Normal from actual object to intermediate
  - V_0 is a vertex on the object and N its normal

Task

Find the texture coordinate (s, t) corresponding to the point P on the sphere where the normal N intersects

Two-Part Mapping—How to example

Sphere equation is

$$x^2 + y^2 + z^2 = r^2,$$

or equivalently

$$P^2 - r^2 = 0 \tag{1}$$

The intersection point of the normal N of V_0 on the sphere is

$$V_0 + dN = P, \quad d \in \mathbb{R}$$

Inserted into (1) gives

$$(V_0 + dN)^2 - r^2 = 0$$

$$f(d) = \underbrace{N^2}_{a} d^2 + \underbrace{(2V_0N)}_{b} d + \underbrace{(V_0^2 - r^2)}_{c} = 0$$

Two-Part Mapping—How to example

Solution of $f(d) = 0$ is

$$\begin{cases} q = -\frac{1}{2}\left(b + sign(b)\sqrt{\Delta}\right), \text{ where } \Delta = b^2 - 4ac \\ d_1 = q/a, \quad d_2 = c/q \end{cases}$$

- If $\Delta > 0$, the normal intersects the sphere two times ($f(d)$ has two roots)
- If $\Delta = 0$, the normal intersects the sphere one time ($f(d)$ has one root)
- If $\Delta < 0$, no intersection!

Two-Part Mapping—How to example

Solution of $f(d) = 0$ is

$$\begin{cases} q = -\frac{1}{2}\left(b + sign(b)\sqrt{\Delta}\right), \text{ where } \Delta = b^2 - 4ac \\ d_1 = q/a, \quad d_2 = c/q \end{cases}$$

- **If $\Delta > 0$, the normal intersects the sphere two times ($f(d)$ has two roots)**
- If $\Delta = 0$, the normal intersects the sphere one time ($f(d)$ has one root)
- If $\Delta < 0$, no intersection!

However, since the object is inside the sphere we always get two intersections and the *positive root* (denoted d_+) *is in the direction of the normal*!

Two-Part Mapping—How to example

The intersection point on the sphere is

$$P = V_0 + d_+ N$$

Normalize

$$P = \frac{P}{\|P\|_2} r,$$

where $P = (x, y, z)$

Two-Part Mapping—How to example

The texture maps to the sphere as

$$
\begin{aligned}
x &= r\cos(2\pi s) \\
y &= r\sin(2\pi s)\cos(2\pi t) \\
z &= r\sin(2\pi s)\sin(2\pi t)
\end{aligned}
$$

then

$$
\begin{cases}
s = \cos^{-1}(x/r) \\
t = \arctan(z/y)
\end{cases}
$$

Verify!

We want $\{s,t\} \in [0,1]$

$$
\Rightarrow \begin{cases}
s = \dfrac{\cos^{-1}(x/r)}{\pi} \\
t = \left(\dfrac{\arctan(z/y)}{\pi} + 1\right)/2
\end{cases}
$$

Wrapping

- How should the texture behaves when tiled?
  (i.e. when the computed *s* and *t* are outside of the texture's range)

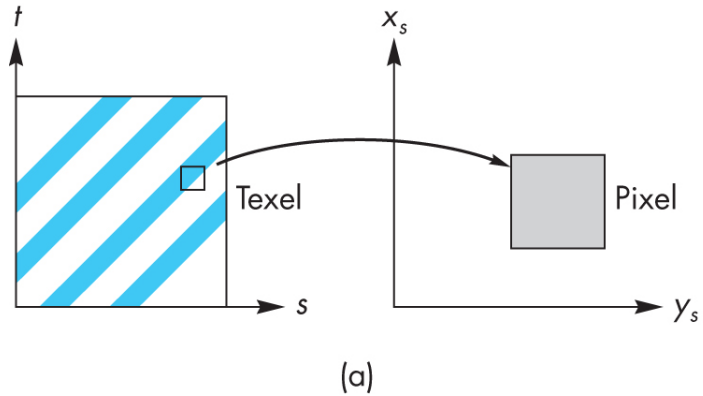
  Tile The Texture repeats (tiles) itself (`GL_REPEAT`)

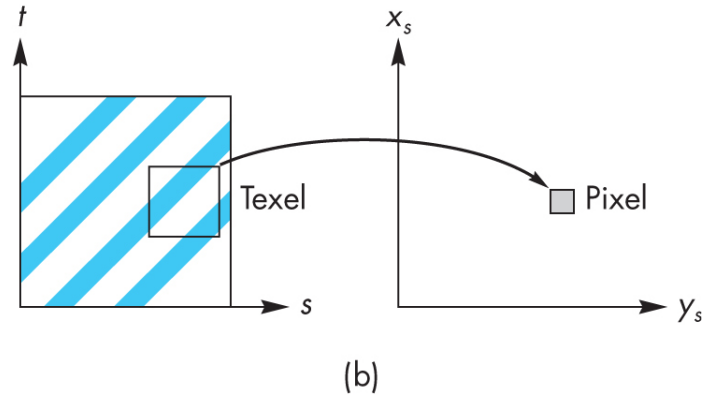
  Clamp The Texture's edges get stretched (`GL_CLAMP_TO_EDGE`, `GL_CLAMP_TO_BORDER`)

  Mirror The Texture repeats itself by adding mirrored images (in both *s*- and *t*-coordinates) (`GL_MIRRORED_REPEAT`)
- Often combinations of the above can be used

Filtering

- Magnification (Undersampling)
  - Viewing the texture close-up
  - One texel maps to severel pixels (Figure a)
- Minification (Oversampling)
  - Viewing the texture from far away
  - Many texels map to a single pixel (Figure b)





Magnification

- What do we do when a texture sample lands between texel centers?

Point sampling	Pick the nearest texel. The Texture becomes blocky up close
Bilinear sampling	Each rendered pixel performs a bilinear blend between the nearest 4 texel centers. The Texture becomes blurry up close
Bicubic sampling	Performs a bicubic blend between a 4×4 grid of texels. An enhancement to bilinear sampling, but adds some memory access costs

Magnification in OpenGL

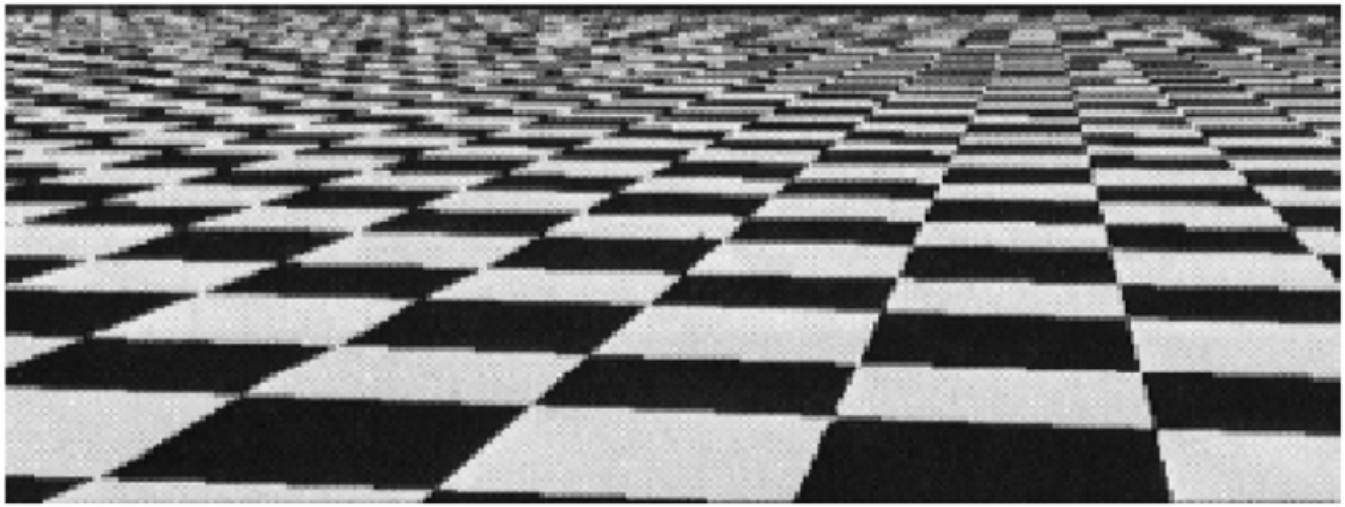
`GL_LINEAR` Interpolate between the four closest, a BILINEAR approach

`GL_NEAREST` Pick the closest one, a POINT approach

GL_LINEAR

GL_NEAREST

Minification



Reason?

- Each pixel on the screen, maps to thousands of texels. Which one should we choose? We have *oversampling*.
- Can use *point sampling*, but can lead to aliasing effects like *shimmering* or *buzzing*.

Mipmapping

MIP = ”multum in parvo” – ”much in little”

- Pre-process input textures by prefiltering it at multiple resolutions
- For example, for a 512×512 texture we store 8 mipmaps 256×256, 128×128, . . . , 1×1.
- Requires $1/3$ extra memory
- The texture do not need to be square

Mipmapping

- From the change in s/t (ds, dt) we can pick a corresponding mip-level, and sample a texel from that level.
- Linear interpolation between the two nearest levels, called *trilinear mipmapping*
- Limitations
  - Visual quality problems (bluring), especially when viewed at a narrow angle (walls and floors)
  - Example, if a 10×3 region maps to one pixel the mip-levels 4×4 and 2×2 will be used
  - One solution is *Anisotropic filtering*

Mipmapping in OpenGL

`GL_NEAREST_MIPMAP_NEAREST`

Point sampling with the best mipmap

`GL_NEAREST_MIPMAP_LINEAR`

Point sampling with linear sampling between the two best mipmaps

`GL_LINEAR_MIPMAP_NEAREST`

Linear sampling with the best mipmap

`GL_LINEAR_MIPMAP_LINEAR`

Linear sampling with linear sampling between the two best mipmaps

Mipmapping Example

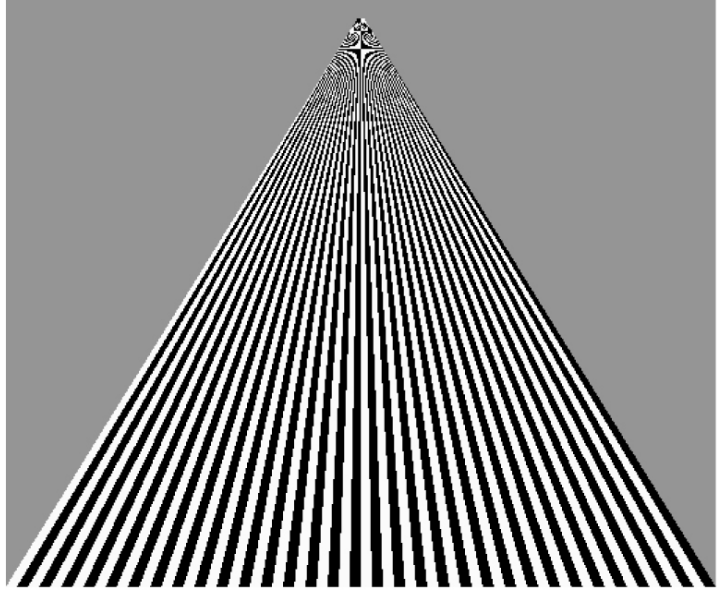
On next slide:

(a) Point sampling

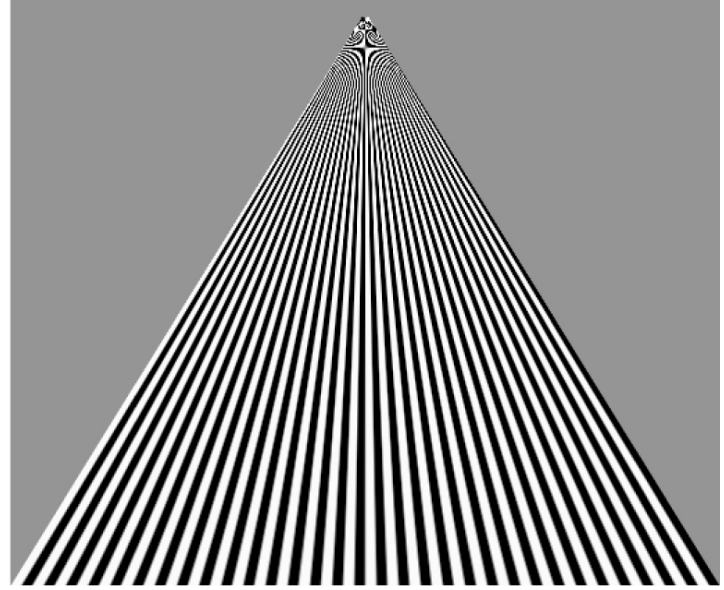
(b) Linear sampling

(c) Mipmapping Point Sampling

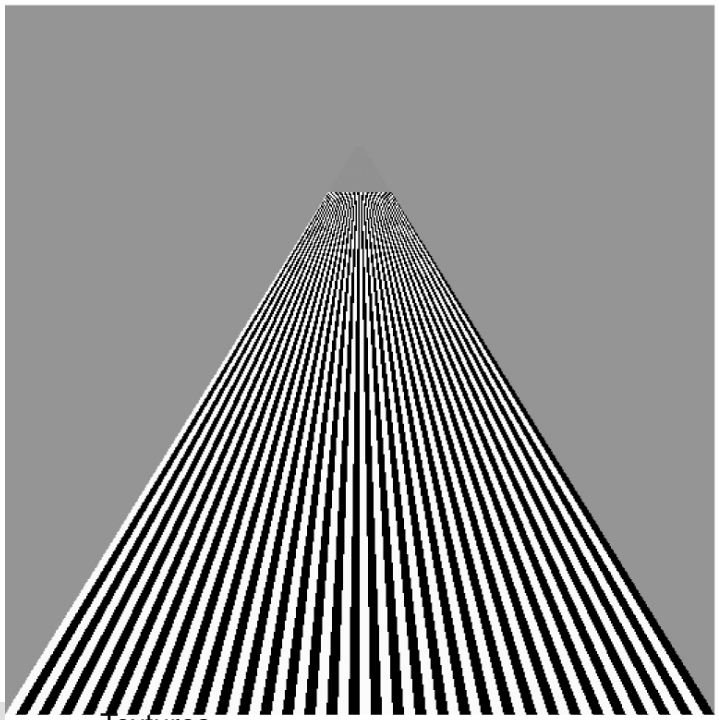
(d) Mipmapping Linear Sampling



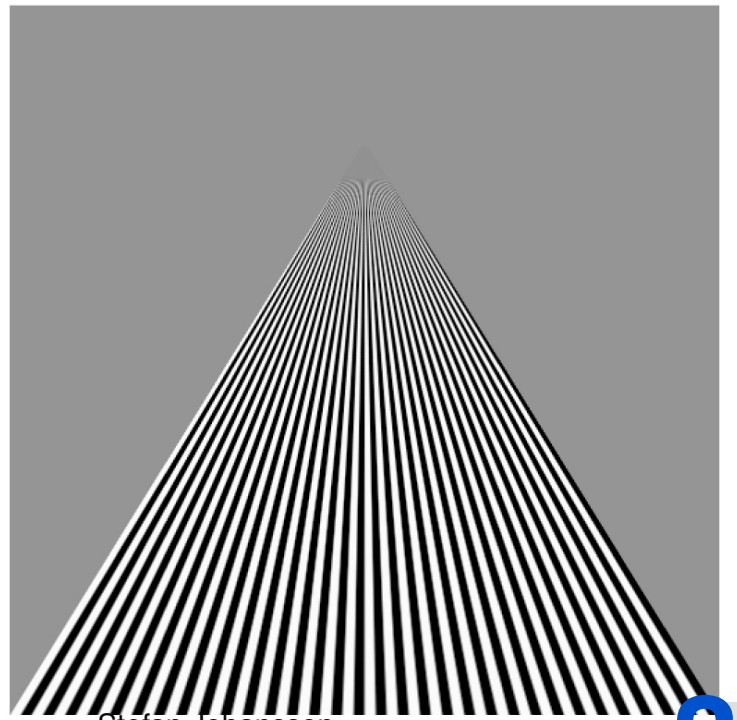
(a)



(b)



(c)



(d)

Anisotropic Filtering

- Also called *Anisotropic mipmapping*
- Addresses the edge-on blurring issue
- In addition to square mipmaps are also *rectangular* mipmaps used
- Usually aspect ratios of $2:1$ or $4:1$

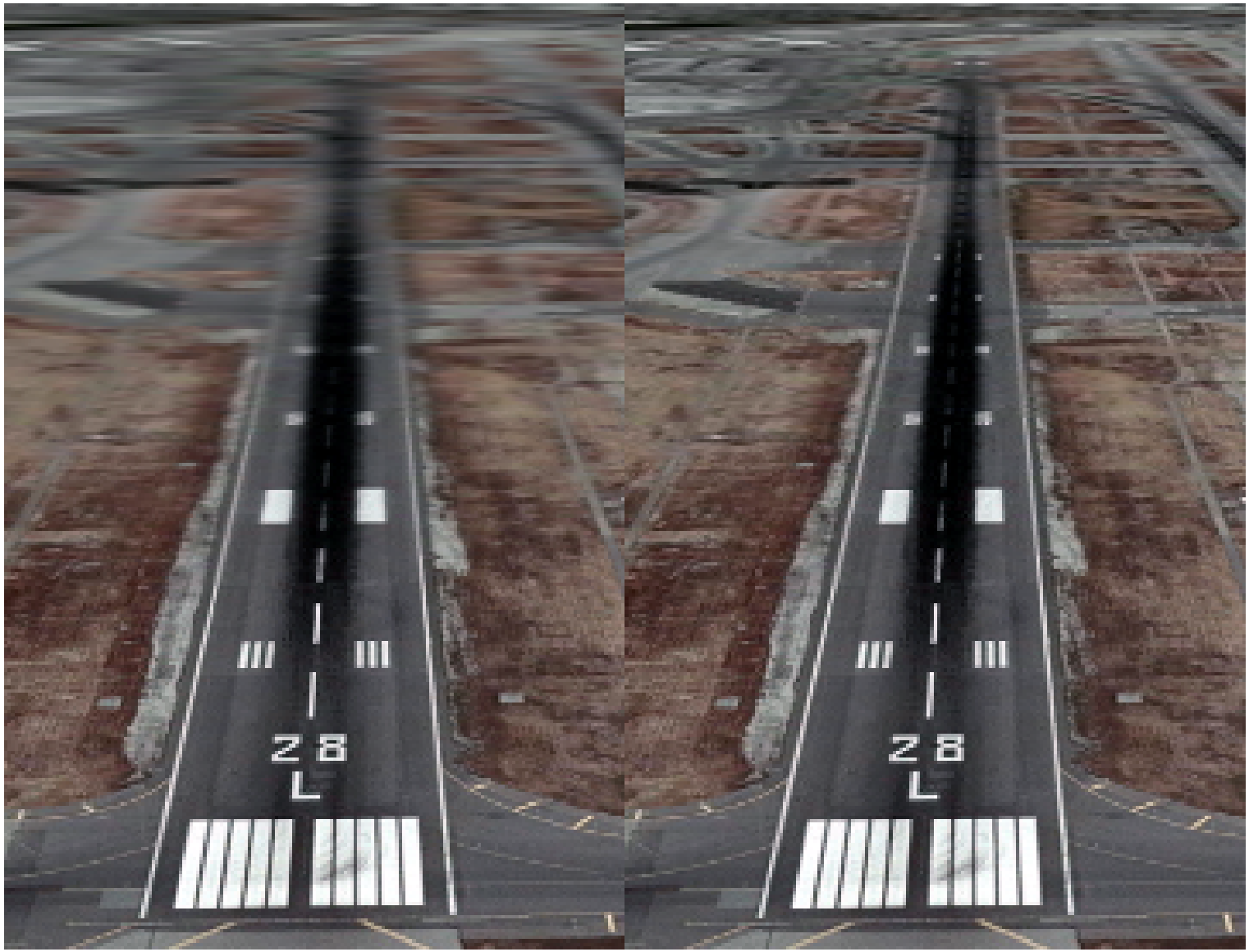
Anisotropic Filtering

- Also called *Anisotropic mipmapping*
- Addresses the edge-on blurring issue
- In addition to square mipmaps are also *rectangular* mipmaps used
- Usually aspect ratios of $2:1$ or $4:1$
- For example, for a 512×512 texture with $2:1$ anisotropic mipmapping we also store mipmaps of 256×512, 512×256, 256×256, $\ldots$, 2×1, 1×1
- $2:1$ aspect requires $2/3$ extra memory
- $4:1$ aspect requires $5/6$ extra memory

Anisotropic Filtering

- Also called *Anisotropic mipmapping*
- Addresses the edge-on blurring issue
- In addition to square mipmaps are also *rectangular* mipmaps used
- Usually aspect ratios of $2:1$ or $4:1$
- For example, for a 512×512 texture with $2:1$ anisotropic mipmapping we also store mipmaps of 256×512, 512×256, 256×256, $\ldots$, 2×1, 1×1
- $2:1$ aspect requires $2/3$ extra memory
- $4:1$ aspect requires $5/6$ extra memory
- However, do not solve all problems...

Anisotropic Filtering Example



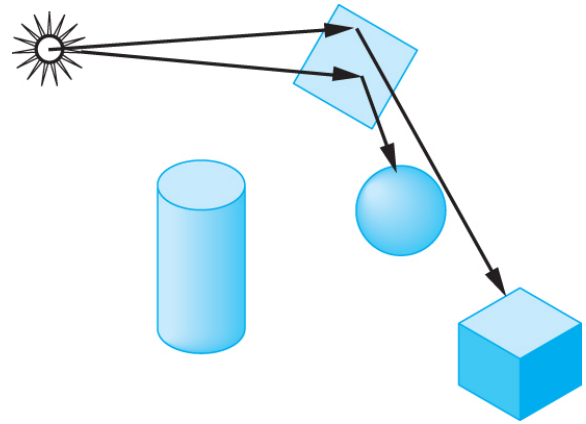
Environment Mapping

In **Environment mapping** (also known as *reflection mapping*), a texture is used to model an object's environment

- Can be done statically or dynamically
- Three common techniques
  - Ray Tracing approach
  - Sphere Mapping
  - Cube Mapping

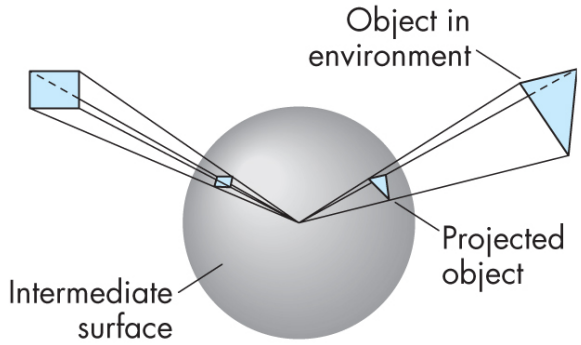
Ray Tracing approach

- *Virtual rays* are shot out from the object into the environment
- The color of the illumination is determined by the color of the environment
- “Simplified” form of ray tracing
- Only practical for one object or together with ray tracing



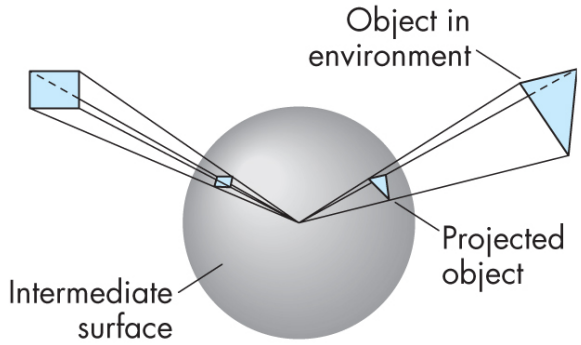
Sphere Mapping

- Project the environment on a sphere centered at the object



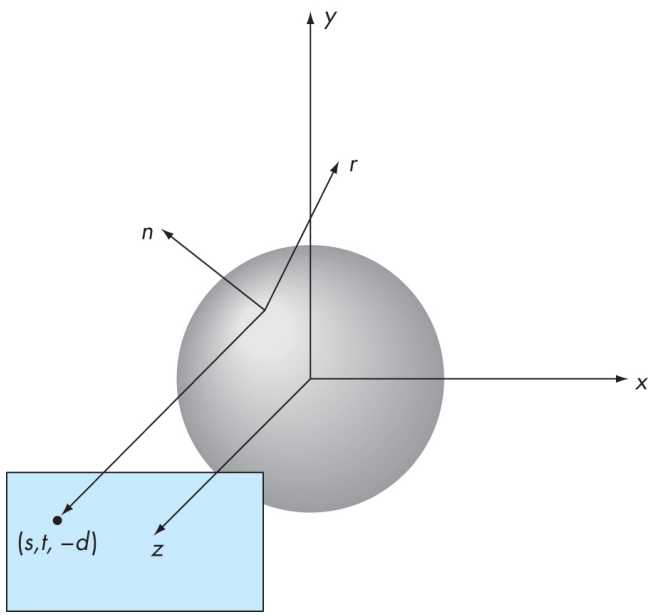
Sphere Mapping

- Project the environment on a sphere centered at the object



- Disadvantages
  - Introduce artifacts in knot-points on the sphere
  - New texture has to be generated for each viewpoint
  - The texture mapped to the sphere will be stretched and compressed, introducing aliasing effects

Sphere Mapping



$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v}, \text{ where}$$

$$\mathbf{v} = [s, t, 0]^T \text{ and}$$

$$\mathbf{n} = \left[s, t, \sqrt{1.0 - s^2 - t^2}\right]^T$$

Then we have that

$$s = r_x/f + 1/2,$$

$$t = r_y/f + 1/2, \text{ where}$$

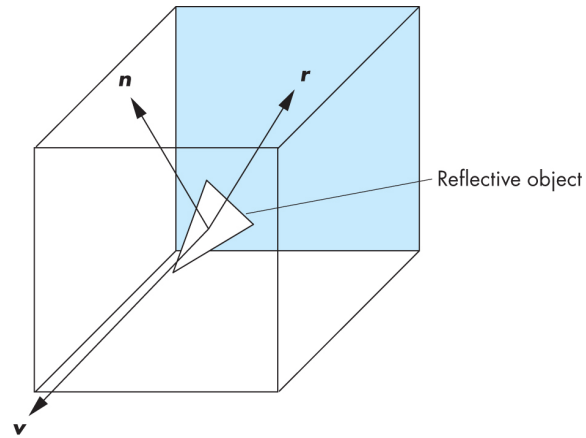
$$f = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$$

The reflection is only correct for a vertex at origin!

Cube Mapping

- The most common technique today
- Six images are (pre-)computed, one for each side
- Recomputed when needed
- Special algorithms needed to handle the seams of the box

- Used for
  - Reflection mapping
  - Skybox
  - Specular Highlights
  - Projection textures



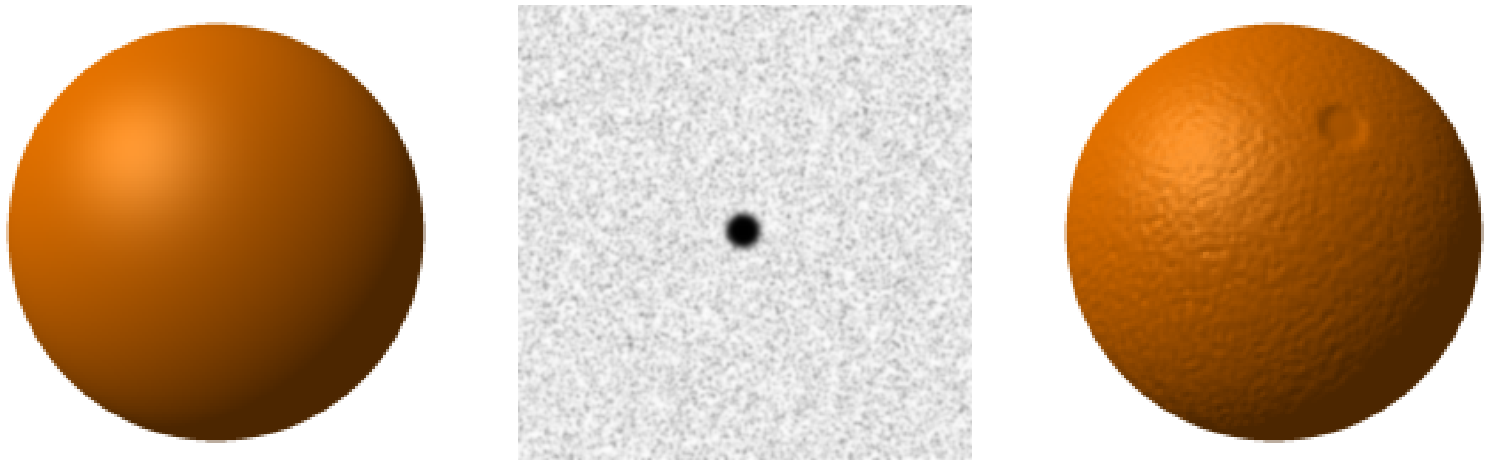
Bump Mapping

Bump mapping aims to add details to shading without using more polygons. These techniques only add visual effects

Bump Mapping	Uses a height map applied on an object that changes the face normal
Normal Mapping	Replaces the normal for each texel
Parallax Mapping	Replaces the normal depending also on the viewing angle

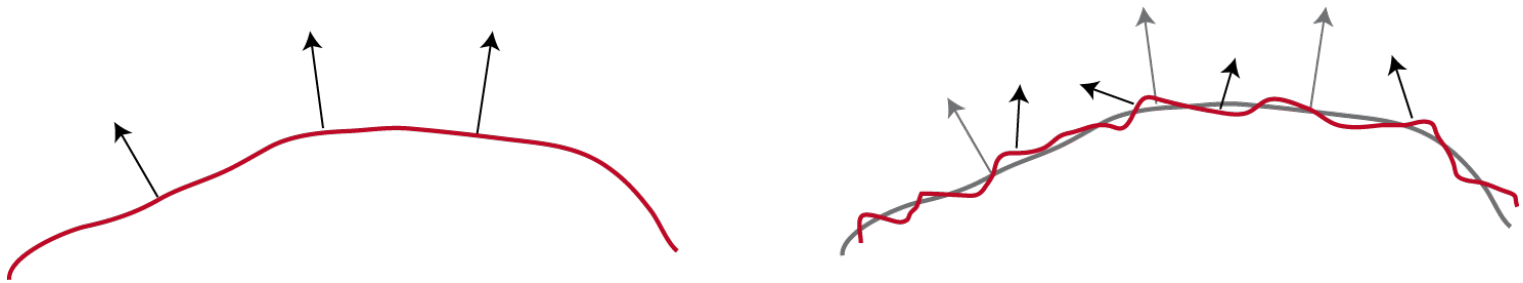
Bump Mapping

- The normal used in the diffuse component is changed using the surface normal in the height map
- The height map is a single channel image (grayscale)



Bump Mapping

- Parameterized surface
  - $\mathbf{p}(u,v) = [x(u,v), y(u,v), z(u,v)]^T$
  - Tangent vectors
    - $\mathbf{p}_u = \partial \mathbf{p} / \partial u$
    - $\mathbf{p}_v = \partial \mathbf{p} / \partial v$
  - Normal vector
    - $\mathbf{n} = (\mathbf{p}_u \times \mathbf{p}_v) / |\mathbf{p}_u \times \mathbf{p}_v|$



Bump Mapping

- *Perturbed* surface
  - $\mathbf{p}'(u,v) = \mathbf{p}(u,v) + d(u,v)\mathbf{n}$
  - $d(u,v) \ll 1$ is the *bump* or *displacement function*

Bump Mapping

- *Perturbed* surface
  - $\mathbf{p}'(u,v) = \mathbf{p}(u,v) + d(u,v)\mathbf{n}$
  - $d(u,v) \ll 1$ is the *bump* or *displacement function*
- Perturbed normals
  - $\mathbf{n}' = \mathbf{p}'_u \times \mathbf{p}'_v$, where

    $\mathbf{p}'_u = \mathbf{p}_u + (\partial d/\partial u)\mathbf{n} + d(u,v)\mathbf{n}_u$

    $\mathbf{p}'_v = \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} + d(u,v)\mathbf{n}_v$
  - If d is *small* it can be neglected

$$\begin{aligned}\mathbf{n}' &\approx \mathbf{p}_u \times \mathbf{p}_v + (\partial d/\partial u)\mathbf{n} \times \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} \times \mathbf{p}_u \\ &= \mathbf{n} + (\partial d/\partial u)\mathbf{n} \times \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} \times \mathbf{p}_u\end{aligned}$$

Bump Mapping

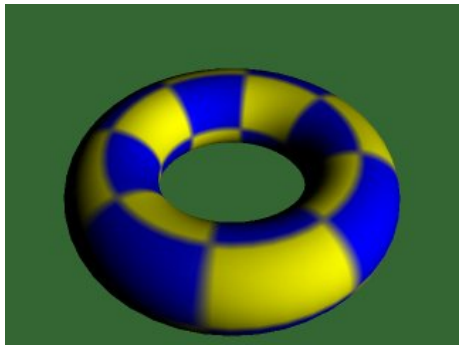
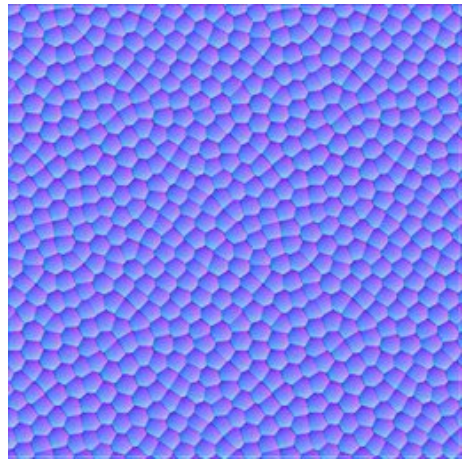
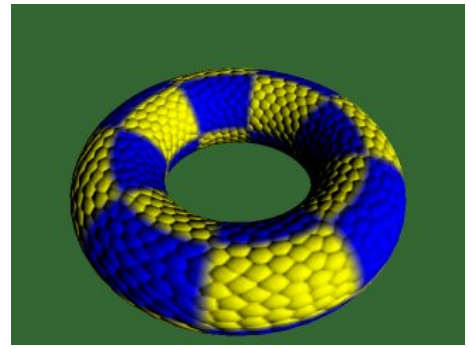
- *Perturbed* surface
  - $\mathbf{p}'(u,v) = \mathbf{p}(u,v) + d(u,v)\mathbf{n}$
  - $d(u,v) \ll 1$ is the *bump* or *displacement function*
- Perturbed normals
  - $\mathbf{n}' = \mathbf{p}'_u \times \mathbf{p}'_v$, where
    $$\mathbf{p}'_u = \mathbf{p}_u + (\partial d/\partial u)\mathbf{n} + d(u,v)\mathbf{n}_u$$
    $$\mathbf{p}'_v = \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} + d(u,v)\mathbf{n}_v$$
  - If d is *small* it can be neglected

$$\begin{aligned}\mathbf{n}' &\approx \mathbf{p}_u \times \mathbf{p}_v + (\partial d/\partial u)\mathbf{n} \times \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} \times \mathbf{p}_u \\ &= \mathbf{n} + (\partial d/\partial u)\mathbf{n} \times \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} \times \mathbf{p}_u\end{aligned}$$

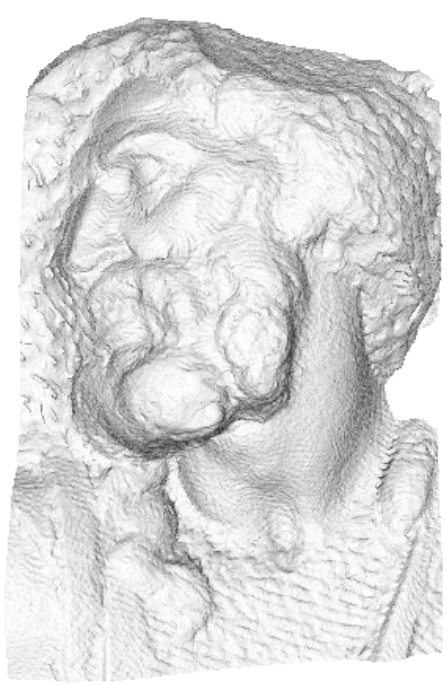
- The normal is perturbed during shading
- $(\partial d/\partial u)$ and $(\partial d/\partial v)$ can be precomputed and stored in a so called *normal map*

Normal Mapping

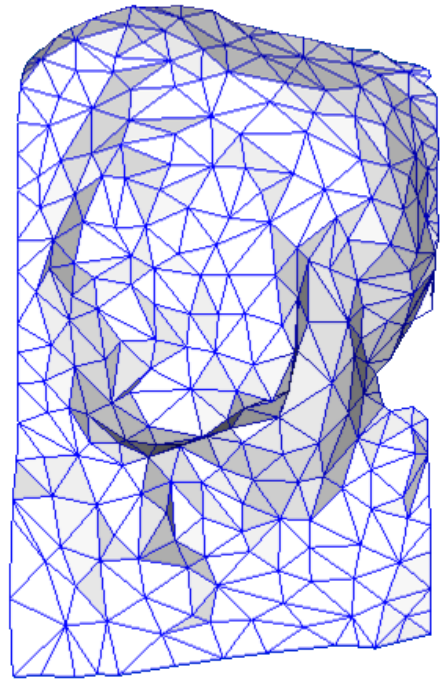
- Uses a multichannel image (RGB) to store changes in (x, y, z)
- x and y are measures of rotation around the x- and y-axis, and z is a measure of displacement on z-axis.

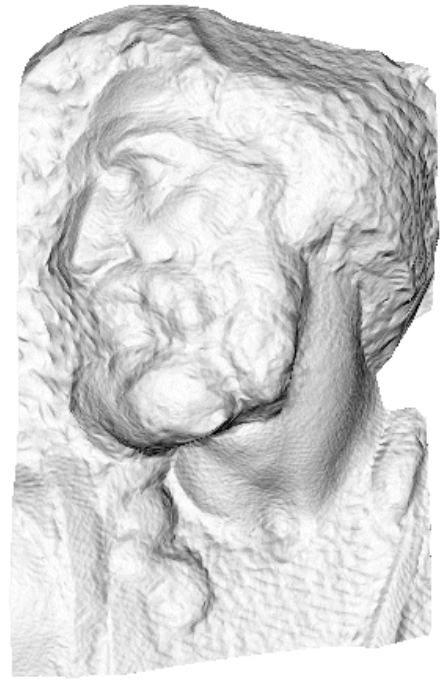
Normal Mapping Example



original mesh
4M triangles



simplified mesh
500 triangles



simplified mesh
and normal mapping
500 triangles

Parallax Mapping

- The effect of the bump map is a function of the viewing angle.
- An increased viewing angle gives an increased effect.

Texture

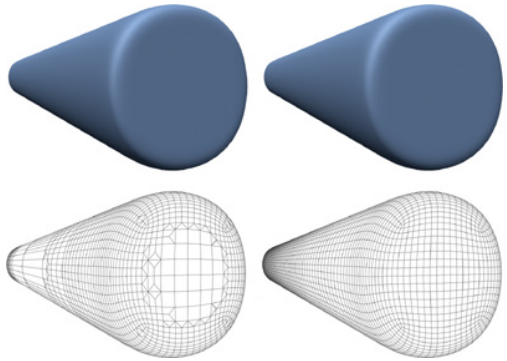
Bump

Parallax

Displacement Mapping

Displacement mapping perturbs the actual surface

- Requires a fine mesh (triangles down to about 1 pixel in size)
- Finer mesh computed by *Adaptive Tessellation of Subdivision Surfaces*

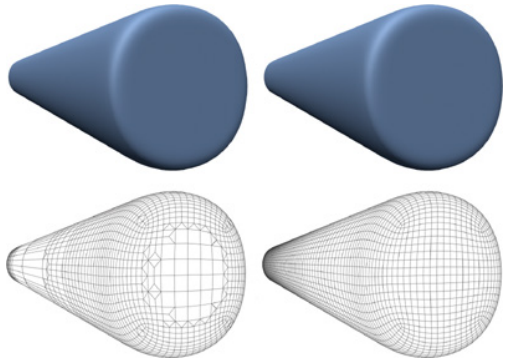


Adaptive Tesselation

Displacement Mapping

Displacement mapping perturbs the actual surface

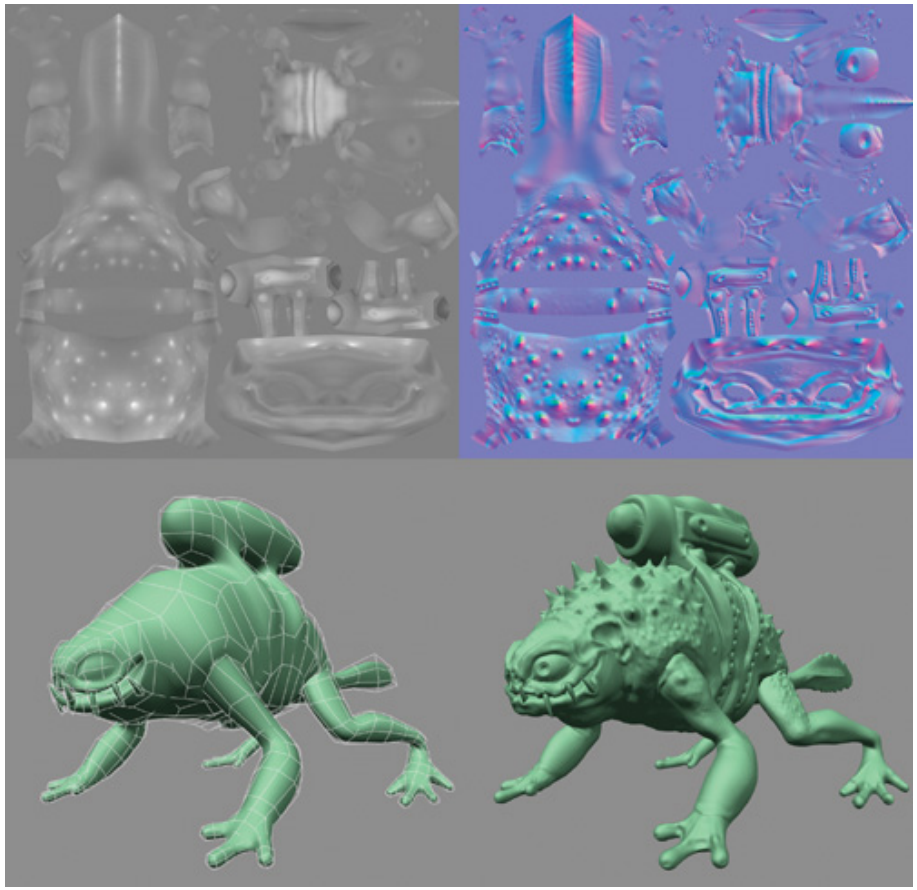
- Requires a fine mesh (triangles down to about 1 pixel in size)
- Finer mesh computed by *Adaptive Tessellation of Subdivision Surfaces*



Adaptive Tesselation

- Expensive, but used for high quality rendering, e.g., in movies
- Also exists *per-pixel* displacement mapping
  - Done in the pixel shader

Displacement Mapping Example



(Top) Tangent and normal map textures
(Bottom) Before and after applying displacement mapping

An Example in OpenGL – A Textured Sphere

The example code is inspired by the following introductory texts on using textures in OpenGL:

- *Textures objects and parameters*, https://open.gl/textures
- *Texturing and Lighting with OpenGL and GLSL*, http://www.3dgep.com/texturing-and-lighting-with-opengl-and-glsl (an updated version of SOIL can also be found on this page)

Example – Init Texture

Initialization of texture buffer

```
GLuint idTexture;
glGenTextures(1, &idTexture);
// Set active texture unit to 0
// (not needed if only one texture is used in the shader)
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, idTexture);

// Wrapping
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

// Filtering
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

// Generate mipmaps
glGenerateMipmap(GL_TEXTURE_2D);

glBindTexture(GL_TEXTURE_2D, 0);
```

Example – Load Image File

OpenGL has no support for reading image-files

Have to use external libraries, for example:
SOIL2 (bitbucket.org/SpartanJ/soil2),
GLI (www.g-truc.net/project-0024.html),
FreeImage (freeimage.sourceforge.net/), or
SOIL (www.lonesock.net/soil.html)

```
// Define a 2D texture image to be read by the shaders
glTexImage2D( GL_TEXTURE_2D, // Target texture format
              0,             // Level-of-detail, 0-base image level
              GL_RGB,        // Internal format
              width, height, // Width and height of image
              0,             // Must be 0!
              GL_RGB,        // Format of pixel data
              GL_UNSIGNED_BYTE,  // Data type of pixel data
              image);        // The image
```

Example – Vertex Shader

```
// Explicitly set the attribute indices
layout (location=0) in vec3 vPosition;
layout (location=1) in vec3 vNormal;
layout (location=2) in vec2 TexCoord;

out vec4 W_position; // Position in world space
out vec4 W_normal;   // Surface normal in world space
out vec2 f_texcoord; // Texture coordinates

uniform mat4 MVP;          // ModelViewProjection matrix
uniform mat4 ModelMatrix;

void main()
{
  gl_Position = MVP*vec4(vPosition, 1);

  W_position = ModelMatrix * vec4(vPosition, 1);
  W_normal = ModelMatrix * vec4(vNormal, 0);
  f_texcoord = TexCoord;
}
```

Example – Fragment Shader

```
in vec4 W_position;
in vec4 W_normal;
in vec2 f_texcoord;

// Lightning and material properties...

uniform sampler2D tex; // By default texture unit 0

out vec4 color;

void main()
{
  // Compute the illumination...

  color = (ambient + diffuse + specular) * texture( tex, f_texcoord);
}
```

Example – Load Geometry

Prerequisites:
Each vertex of the sphere must include **position** (x, y, z), **normal**, and **texture coordinate** (s, t) (see Slide 20)

```
// Vertex attributes for a sphere object
vector<vec3> positions;
vector<vec3> normal;
vector<vec2> texCoords;
vector<GLuint> indices;

// Populate position, normal, texCoords,
// and generate indices (triangulate)...

// Bind buffers
GLuint vao;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );

GLuint vbos[4]; // One buffer for each vertex attribute
glGenBuffers( 4, vbos );
```

Example – Load Geometry

```
// Bind position and normal data...

// Bind texture coordinates
glBindBuffer( GL_ARRAY_BUFFER, vbos[2] );
glBufferData( GL_ARRAY_BUFFER, texCoords.size() * sizeof(vec2),
  texCoords.data(), GL_STATIC_DRAW );

glVertexAttribPointer( 2, // Use attribute index 2 in vertex shader
  2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );
glEnableVertexAttribArray( 2 );

// Bind indices...
```

Example – Draw Textured Sphere

```
// Bind texture to the next render call
glBindTexture( GL_TEXTURE_2D, idTexture);

// Set MVP-matrix, light properties, etc...

glDrawElements( GL_TRIANGLES, numIndices, GL_UNSIGNED_INT,
  BUFFER_OFFSET(0) );
```