# GRK 6

Dr Wojciech Palubicki

## Phong Lighting Model



Ambient light

Diffuse light

Specular light

Phong lighting model

#### Color variation in space

- Diffuse lighting color is the same for each pixel
- We can assume that the color  $k_d$  differs for pixels



## Texturing

- From data:
  - Read information from 2D images





- Procedural:
  - Write a program that calculates color as a function of position



#### Texture effect



#### Model

#### Texture effect





#### Model

Model + Shading

### Texture effect







Model

Model + Shading

Model + Shading + Textures

### Texturing



3D model

**Textured model** 



We need a function that associates every surface point of a model with a coordinate of a texture





For every point we are drawing we want to retrieve the color information stored in the texture

#### Coordinates u, v

- Every vertex stores 2D coordinates (u, v) in texture space
  - Coordinates u, v define positions in the texture for every vertex
- Color information between vertices is obtained via interpolation



#### Coordinates u, v

- Texture coordinates u, v are specified:
  - Manually by the user
  - Automatically, using parametric optimization



#### Parametric optimization



## 3D model

- Necessary information:
- For vertices:
  - Positions (4D/3D coordinates)
  - Normals (3D coordinates)
  - 2D u, v coordinates
- Other information:
  - Parameters and shading method
  - Images of our textures

#### Wavefront .OBJ file



# v - vertices in x,y,z coordinates # vt - texture in u,v coordinates # vn - normals in nx,ny,nz coordinates # f - faces: map vertices, UVs, normals g cube v 8.0 0.0 8.0 v 0.0 0.0 1.0 v 0.0 1.0 0.0 v 0.0 1.0 1.0 v 1.0 0.0 0.0 v 1.0 0.0 1.0 v 1.0 1.0 0.0 v 1.0 1.0 1.0 vt 0.8 0.0 vt 1.0 0.0 vt 1.0 1.0 vt 0.0 1.0 vn 0.0 0.0 1.0 vn 0.0 0.0 -1.0 vn 0.0 1.0 0.0 vn 0.0 -1.0 0.0 vn 1.0 0.0 0.0 vn -1.0 0.0 0.0 f 1/1/2 7/3/2 5/2/2 f 1/1/2 3/4/2 7/3/2 f 1/1/6 4/3/6 3/4/6 f 1/1/6 2/2/6 4/3/6

f 3/1/3 8/3/3 7/4/3

# Notes:

## Texture mapping

- **Texturing** is the mapping of **2D** images to **3D** polygons
- The image we are mapping we call **texture**
- A pixel in the texture we call **texel**
- Pixel colors are given by texels in the texture

### Texturing



#### Texture space

- **Texture space** is the space in which the texture exists
- It is a 2D space where the horizontal and vertical axes are called u ∈
   [0, 1] and v ∈ [0, 1]
- The texture completely fills the texture space, i.e. bottom left corner is in position (0,0) and top right corner at (1, 1)
- Texel coordinates (U, V) for a  $M_x \times M_y$  texture are
  - U =  $[M_x \cdot u + 1]$
  - V =  $[M_y \cdot v + 1]$
  - If u = 0 then U = 0 and if u = 1 then U = M<sub>x</sub> + 1, therefore we must test 0 < u, v</li>
     < 1</li>



- Vertices of polygon P, correspond to vertices Q in texture space
- Every transformation should be accounted for in texture space
- Using the scan-line method, maximum
   P<sub>L</sub> is computed by interpolating
   between P<sub>1</sub> and P<sub>4</sub>
- Similarly maximum  $P_R$  is computed by interpolating between  $P_4$  and  $P_3$



- Corresponding maxima in texture space  $Q_L$  and  $Q_R$ , are computed by interpolating the edges of the texture
- Pixels between P<sub>L</sub> and P<sub>R</sub> assume the same color as texels between Q<sub>L</sub> and Q<sub>R</sub>



## Computing the maxima

- Vertex coordinates are scaled to display size
- $x = max \left[ 1, (x_s + 1) \frac{N_x}{2} \right], \qquad y = max \left[ 1, (y_s + 1) \frac{N_y}{2} \right]$
- Maxima  $P_L$  and  $P_R$  are initialized with value of the greatest coordinate  $y_s$  (i.e.  $P_4$ )
- Coordinate y of  $P_L$  and  $P_R$  is calculated by deducting 1 from the current coordinate
- $x_L$  is calculated by interpolating between  $P_4$  and  $P_1$

• 
$$x_L = x_4 - \frac{x_4 - x_1}{y_4 - y_1}$$

### Computing the maxima

• 
$$x_L = x_4 - \frac{x_4 - x_1}{y_4 - y_1}$$

- Gradient is constant
- $x_L = x_L \Delta x_L$
- $x_R = x_R \Delta x_R$
- Where

• 
$$\Delta x_L = \frac{left \ edge \ length}{left \ edge \ height} = \frac{x_4 - x_1}{y_4 - y_1}$$
  
•  $\Delta x_R = \frac{right \ edge \ length}{right \ edge \ height} = \frac{x_4 - x_3}{y_4 - y_3}$ 

#### Computing the maxima



#### Computing the maxima in texture space

 v<sub>L</sub> translates between v<sub>4</sub> and v<sub>1</sub> similarly as y between y<sub>4</sub> and y<sub>1</sub>

• 
$$\Delta u_L = \frac{u_4 - u_1}{y_4 - y_1}$$

• 
$$\Delta v_L = \frac{v_4 - v_1}{y_4 - y_1}$$

- If  $P_L$  lowers to the next scan line:
- $u_L = u_L \Delta u_L$
- $v_L = v_L \Delta v_L$
- Similarly for  $u_R$  and  $v_R$



### Interpolation of scan lines

- Starting with  $(x_L, x_R)$  we initialize  $u = u_L$  and  $v = v_L$
- For each pixel we translate on the scan line and update the coordinates (u, v)
- $u = u + \Delta u$
- $v = v + \Delta v$
- We move from  $u_L$  to  $u_R$  in the same linear span as for  $x_L$  to  $x_R$ , therefore

• 
$$\Delta u = \frac{u_R - u_L}{x_R - x_L}$$

• 
$$\Delta v = \frac{v_R - v_L}{x_R - x_L}$$







Texture space

Screen space





## Why do we see artifacts?

- This texturing method relies on the assumption that mapping from 3D to 2D is a linear transformation
- Relations between x and x<sub>s</sub> as well as y and y<sub>s</sub> are linear, relations between z and z<sub>s</sub> are NOT linear
- To remove the artifacts we have to take into account the distance of the observer

Perspective projection



Perspective projection



Perspective projection





- Point P in world space is projected onto screen space P<sub>s</sub>
- Triangles OAP and OBP are similar:

• 
$$\frac{x_s}{n} = \frac{x}{z} \implies x_s = \frac{n}{z}x$$
  
•  $\frac{y_s}{n} = \frac{y}{z} \implies y_s = \frac{n}{z}y$   
•  $z_s = n$ 



- Line equation in world space is:
- $x = \alpha z + \beta$
- Reversing the perspective projection:

• 
$$x = \frac{z}{n} x_s$$



- Line equation in world space is:
- $x = \alpha z + \beta$
- Reversing the perspective projection:

• 
$$x = \frac{z}{n} x_s$$
  
• Giving us  $\frac{z}{n} x_s = \alpha z + \beta$ 



- Line equation in world space is:
- $x = \alpha z + \beta$
- Reversing the perspective projection:

• 
$$x = \frac{z}{n} x_s$$
  
• Giving us  $\frac{z}{n} x_s = \alpha z + \beta$   
•  $z = \beta n \frac{1}{x_s} + \frac{\beta}{\alpha}$ 



• 
$$z = \beta n \frac{1}{x_s} + \frac{\beta}{\alpha}$$

• This is not a linear relation of z and x<sub>s</sub>



• 
$$z = \beta n \frac{1}{x_s} + \frac{\beta}{\alpha}$$

- This is not a linear relation of z and x<sub>s</sub>
- We can use the inverse relation

• 
$$\frac{1}{z} = \frac{x_s}{\beta n} - \frac{\alpha}{\beta}$$

• The relation between  $\frac{1}{z}$  and  $x_s$  is linear



#### Perspective texturing

- To apply perspective projection but still use linear interpolation we have to compute the value  $\frac{1}{z}$  for the maxima P<sub>L</sub> and P<sub>R</sub>
- The vertices in texture space are divided by z giving  $\frac{u}{z}$  and  $\frac{v}{z}$
- Linear interpolation is used as before but calculated for coordinates of texture space  $(\frac{u}{z}, \frac{v}{z})$  which correspond to coordinates in screen space  $(x_s, y_s, \frac{1}{z})$





• Faster texturing



• Faster texturing





• Faster texturing





Without mipmapping

mipmapping

## Texturing and lighting









Shading



#### Applied texture

Lighting and texturing

## Fast Phong shading

• Texturing can be used to speed up lighting



## Texturing

- From data:
  - Read information from 2D images





- Procedural:
  - Write a program that calculates color as a function of position



## Procedural texturing

• Alternatively, to texture mapping we can write a program that calculates the pixel color as a function of position (x, y, z)





## Procedural texturing

- Pros:
  - Uses up less memory
  - Infinite resolution
- Cons:
  - Less intuitive
  - Usually hard to find a function that simulates a given pattern





• Stripes along axis x:

```
Stripes(x<sub>s</sub>, y<sub>s</sub>, z<sub>s</sub>)
{
    If(sin x<sub>s</sub> > 0) return color0
    Else return color1
}
```





• Stripes along axis z:

```
Stripes(x<sub>s</sub>, y<sub>s</sub>, z<sub>s</sub>)
{
    If(sin z<sub>s</sub> > 0) return color0
    Else return color1
}
```





• Stripes with variable width

```
Stripes(x<sub>s</sub>, y<sub>s</sub>, z<sub>s</sub>, width)
{
    If(sin (π * x<sub>s</sub> / width) > 0)
      return color0
    Else return color1
}
```





• Stripes with variable width

Stripes(x<sub>s</sub>, y<sub>s</sub>, z<sub>s</sub>, width)
{
 If(sin (π \* x<sub>s</sub> / width) > 0)
 return color0
 Else return color1
}



## 3D stripes

• Gradual variation of colors

```
Stripes(x<sub>s</sub>, y<sub>s</sub>, z<sub>s</sub>, width)
{
    t = (1 + sin (π * x<sub>s</sub> / width)) / 2
    Return (1 - t) color0 + t color1
}
```



## Normal mapping



Texture

Normal map

## Normal Mapping



Without normal mapping

Normal mapping

## Tangent space



## Where to calculate lighting?



## Calculate lighting in tangent space



#### Vertex Shader

- Create a vertex shader with 4 attributes
  - layout(location = 0) in vec3 vertexPosition;
  - layout(location = 1) in vec2 vertexTexCoord;
  - layout(location = 2) in vec3 vertexNormal;
  - layout(location = 3) in vec3 vertexTangent;
- Calculate the normal, tangent and bitangent in world space (multiply modelMatrix with normal and tangent vectors – bitangent is the cross of transformed normal and tangent)
- Transform light, camera position and vertex position by tangent basis, e.g.
  - l.x = dot (lightDir, t);
  - l.y = dot (lightDir, b);
  - l.z = dot (lightDir, n);
- Pass the transformed vectors to fragment shader

### Fragment Shader

- Create 2 sampler2D variables for texture and normal map
- Instead of the interpolated normal use the normal stored in the normal map (you have to scale the normal  $[0,1]^3 \rightarrow [-1,1]^3$ )
- Calculate lighting model as before but use the transformed vectors

## Normal mapping







original mesh 4M triangles simplified mesh 500 triangles simplified mesh and normal mapping 500 triangles