Wrapping up the Pipeline

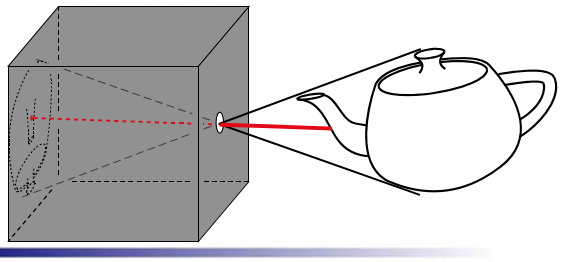
All the nitty gritty details you wished to get rid of…

Elmar Eisemann
Delft University of Technology (TU Delft)

- A long time ago in a galaxy far far away…

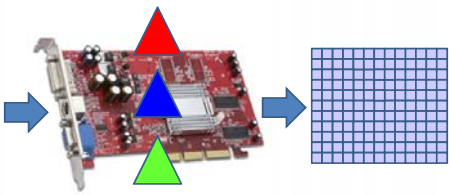
Pinhole camera

- Box with hole
- Perfect image for “point-sized” hole



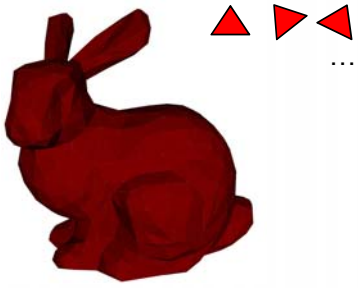
Graphics Pipeline

- Highly parallelizable ⇨GPUs



Triangle Mesh

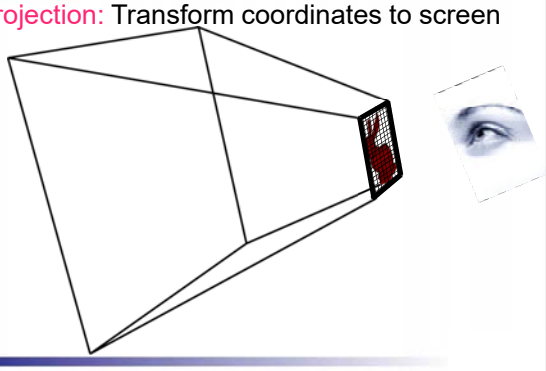
- Models are typically lists of triangles



…

Simplified Graphics Pipeline

- Projection: Transform coordinates to screen



Summary

- First steps into the Graphics Pipeline
  - Homogenous Coordinates
  - Matrix Stacks
  - Camera Model

Complete Camera Model

- Finally:

image projection orientation/location

$$P = \begin{pmatrix} a_x & 0 & x_0 \\ 0 & a_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_0 \\ r_{10} & r_{11} & r_{12} & t_1 \\ r_{20} & r_{21} & r_{22} & t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} a_x & 0 & x_0 \\ 0 & a_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Complete Camera Model

- Finally:

$$P = \begin{pmatrix} a_x & 0 & x_0 \\ 0 & a_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_0 \\ r_{10} & r_{11} & r_{12} & t_1 \\ r_{20} & r_{21} & r_{22} & t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} r_{00} & r_{01} & r_{02} & t_0 \\ r_{10} & r_{11} & r_{12} & t_1 \\ r_{20} & r_{21} & r_{22} & t_2 \end{pmatrix}$$

Complete Camera Model

- Finally (notation often used in vision literature):

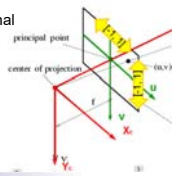
$$P = \begin{pmatrix} a_x & 0 & x_0 \\ 0 & a_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_0 \\ r_{10} & r_{11} & r_{12} & t_1 \\ r_{20} & r_{21} & r_{22} & t_2 \end{pmatrix}$$

intrinsic/ internal parameters

Change settings

extrinsic / external parameters

Move camera



Questions?

It could have been so simple…

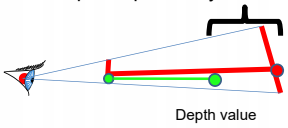
- What is the problem of this matrix for the Graphics Pipeline?

$$P = \begin{pmatrix} a_x & 0 & x_0 \\ 0 & a_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_0 \\ r_{10} & r_{11} & r_{12} & t_1 \\ r_{20} & r_{21} & r_{22} & t_2 \end{pmatrix}$$

We cannot eliminate depth!

- We need to keep a Z!

Compare new distance to stored distance
Update pixel only if new distance is nearer

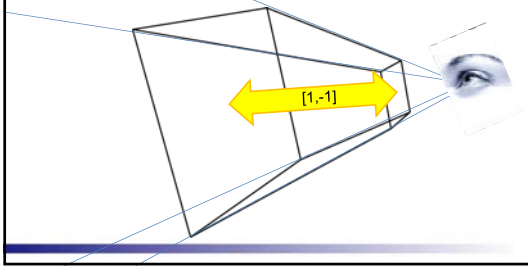


Depth value

- Do we need the “true depth”?
- Any monotonic mapping would work!

Problem:

- A 3D scene is infinite…
- How do we represent Z?
- Solution add a near and far clipping plane!



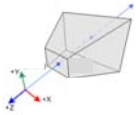
Camera in Matrix Representation

- Camera space:

screen mapping
screen mapping

$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2near\,far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

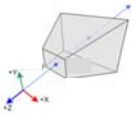
Z mapping



In practice orientation
along z-axis is inversed
…don’t bother too much…

Camera in Matrix Representation

- What does this matrix do?
  - Vertices multiplied with this matrix change their coordinates and are projected onto the screen.
  - In other words, the entire scene is transformed!



Camera Space

- Perspective Frustum is mapped on a cube

$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2near\,far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

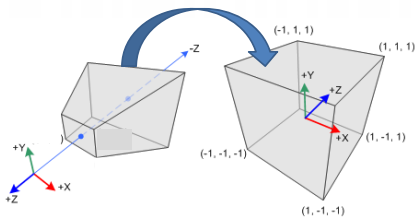
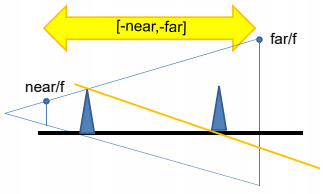


Illustration of the Camera Mapping



$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2near\,far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- Verify:
  1) (0,0,-near,1) has depth -1,
     and (0,0,-far,1) has depth 1
- 2) What does (0,0,-(far+near)/2,1) map to?
- 3) Where are those points in the image?

You will notice, the Z-Buffer is not linear, the zero depth will not map to the frustum center!

Graphics Pipeline Camera

Historically the graphics pipeline has a projection and a model/view matrix and a viewport matrix to map to the pixels.

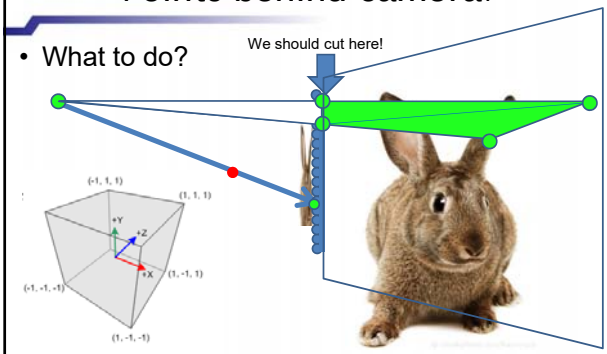
$$\begin{bmatrix} a_x & 0 & 0 & x_0 \\ 0 & a_y & 0 & y_0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{near+far}{near-far} & \frac{2near\,far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix} * \begin{pmatrix} r_{00} & r_{01} & r_{02} & t_0 \\ r_{10} & r_{11} & r_{12} & t_1 \\ r_{20} & r_{21} & r_{22} & t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

First the model/view matrix is applied, then the projection matrix, then the viewport

What is what?

Points behind camera?

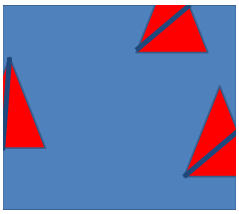
- What to do?



- Test if triangle lies in the cube!

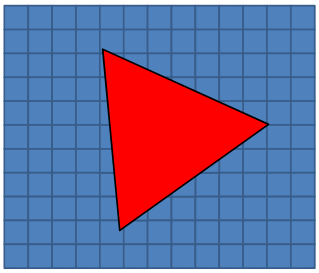
Clip Space

- Create clipped triangles



Simplified Graphics Pipeline

- Rasterization: Fill screen pixels



+ Depth Test

Summary

- Graphics Pipeline
  - Homogenous Coordinates
  - Matrix Stacks
  - Camera Model
  - Clipping/Rasterization
- Standard Pipeline easy to use!

Images are not very exciting yet…

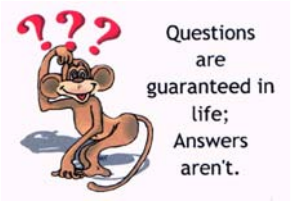
Shading

How to transform

in

?

Questions?



Graphics API

- Two major types: DirectX & OpenGL
  - Allows us to send commands to the card
  - Relatively close to hardware

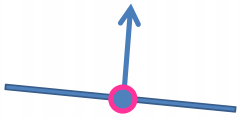
Main difference: OpenGL is a state machine
-> activate something and it stays activated

- We will use OpenGL

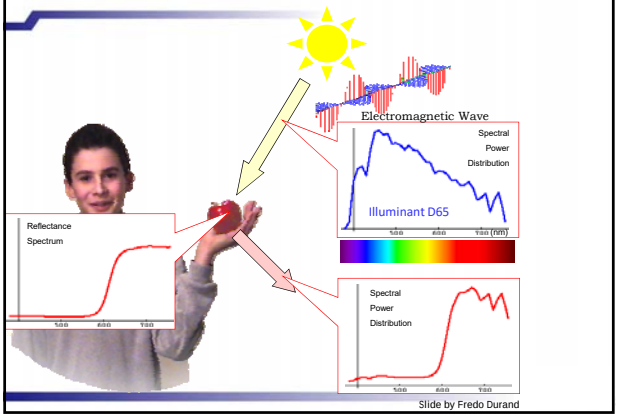
Making beautiful pictures…

TODAY'S QUESTION:

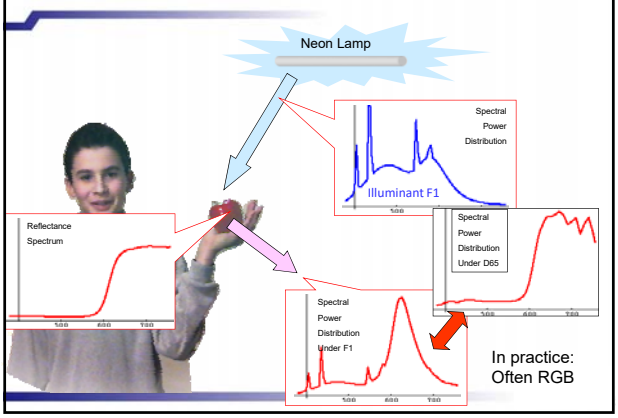
- Given point properties
  (i.e., position, normal, and other attributes)
- How to compute a realistic color???



What is Color?

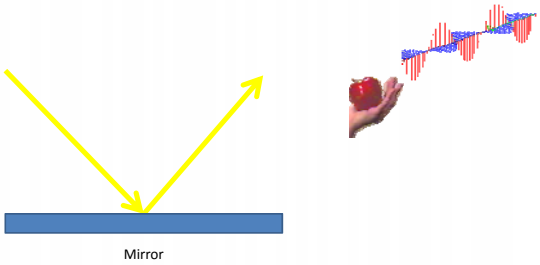


What is Color?



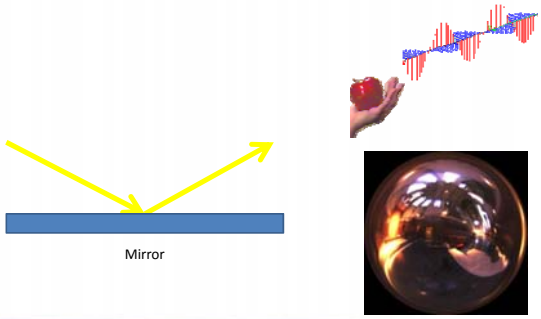
Reflection

- What happens when the light hits a surface?



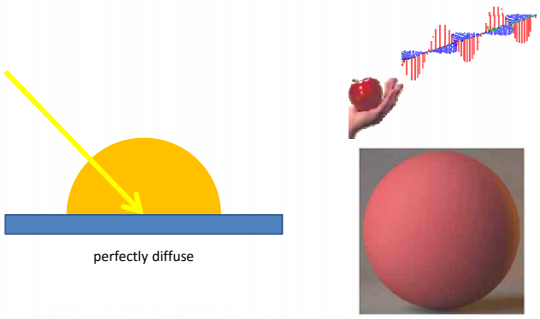
Reflection

- What happens when the light hits a surface?



Reflection

- What happens when the light hits a surface?

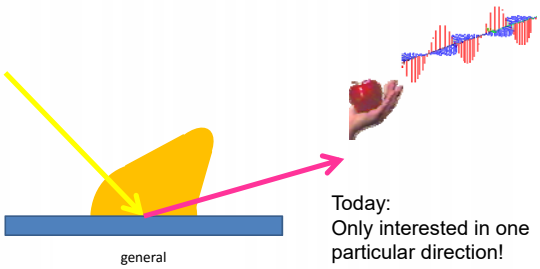


Reflection

- What happens when the light hits a surface?

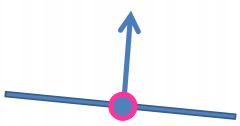
Reflection

- What happens when the light hits a surface?



Making beautiful pictures…

- Given point properties (i.e., position, normal, and other attributes)
- Apply shading model



Acquired BRDFs

- Big databases

Often costly, much data, hard to use for artists
e.g., "Can you make the blue darker?"

Simplified Models

- Mathematically describe Material Properties
- Sum of 3 terms
  - Ambient
  - Diffuse
  - Specular
    - Phong
    - Blinn-Phong

Color - Recap

- Remember:
  Visual system uses 3 cone types for color

Each component can be treated separately.
Hence, in the following, we use a single scalar.
(Imagine it to be red, green, or blue…)

Ambient Term

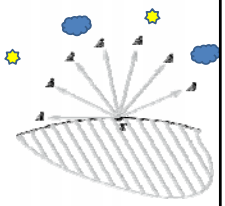
- Represent "scene light"
  - Skylight
  - Light reflected from neighboring surfaces
- Model:
  - Extremely simple

Light property (RGB)

Componentwise multiplication (RGB)! Each component can be treated separately!

$$A = I_a K_a$$

Surface property (RGB)



Ambient Term

- Very simplistic
  - No real physical basis…
  - No indications on the shape of an objet!

K_a increasing

- But often used in practice
  - Emulates indirect light from scene
  - Elements in shadow are not completely black

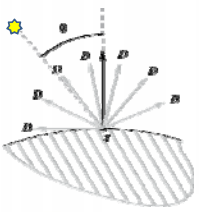
Diffuse Term

- Lambert Surfaces – isotropic reflection
  - Reflection uniformly in all directions
  - Conserves energy
- Model:
  - Uses local surface orientation

Light property (RGB)

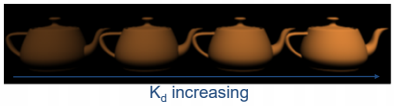
$$D = I_d K_d \cos\theta$$

Surface property (RGB)



Diffuse Term

- Hypothesis: isotropic *a.k.a* lambertian
- Shading varies along surface
  - Gives information about shape



K_d increasing

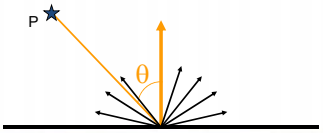
  - Does not depend on observer position!

Diffuse Term

- Where does the cosine come from?

$$D = I_d K_d \cos\theta$$

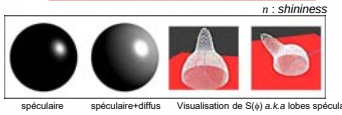
- Imagine light direction as a set of parallel rays
- What do you observe when tilting the rays on a small surface patch?

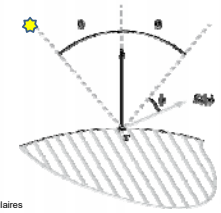


Specular Term

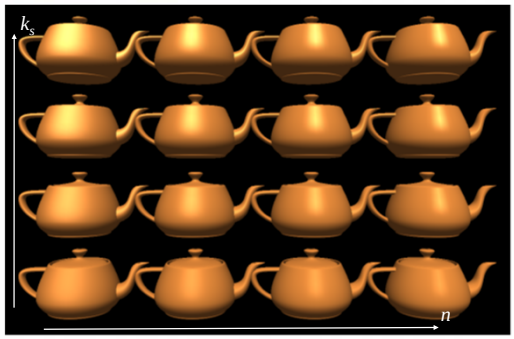
- Represent glossy surfaces
  - Ideal case: mirror
- Model:
  - reflection around mirror ray
  - Exponential decrease

$$S(\phi) = I_s K_s (\cos\phi)^n$$





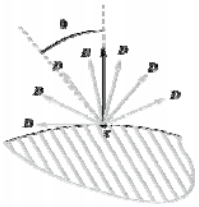
Specular Term



Example

$$A = I_a K_a \qquad D = I_d K_d \cos\theta$$

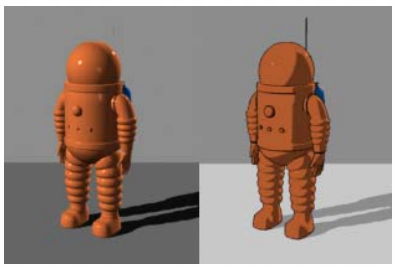
- K_d diffuse: (0.9,0,0)
- I_d diffuse light: (0.9,0.5,1.0)
- K_a ambiante: (0,0,0.1)
- I_a ambiante light: (1,1,0.1)



- Normal at x: (0,0,1)
- Position x: (0,0,0)
- light position : (0,0,10)
- Resulting color: (0.81,0,0.01)

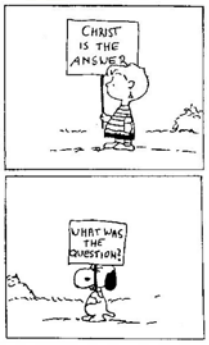
The other extreme…

- Non-photorealistic materials



- Cel-shading: threshold on the diffuse shading

Questions?

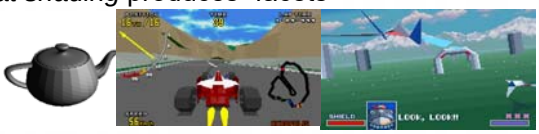


And now in practice?

- We know how to compute shading of a point, but how is it applied on a mesh?

Shading

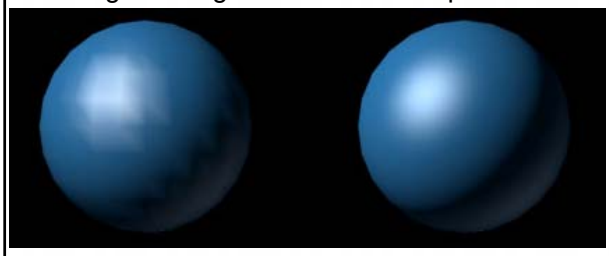
- *Early days - compute color per face:* *Flat shading* produces "facets"



- Later – compute color per vertex: produces *Gouraud Shading* produces a smooth look

Phong shading

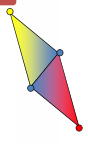
- Today: compute result per pixel
- Phong Shading leads to smooth specularities



Diccan.com

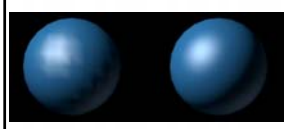
On the practical side: Shading types

- How are the three different types computed?
  - *Flat shading*
    - Applies Phong Model to produce a color per face
  - *Gouraud shading – default in OpenGL*
    - Applies Phong to produce a color per vertex
    - Interpolate color from vertices over triangle
  - *Phong shading* (2 MEANINGS!!!)
    - Interpolate parameters of Phong model :
      - Parameters are :
      - Normal
      - Vector to the lightsource (*light vector*)
    - Applies Phong to produce a color per pixel

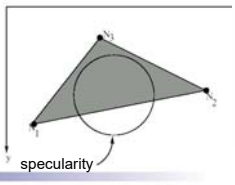


Per vertex Gouraud *vs.* Phong shading *Per pixel*

- Phong more expensive than Gouraud
  - Usually: more pixels than vertices
- Phong more beautiful and standard today
  - Captures specularities between faces



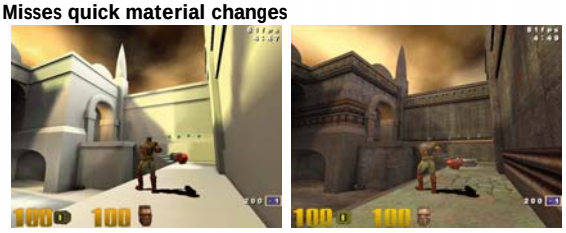
Diccan.com



Questions?

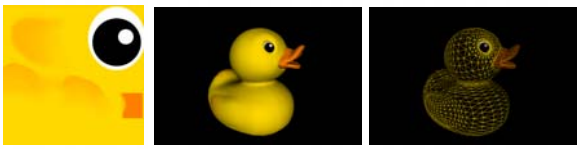
Something is still missing…

Misses quick material changes



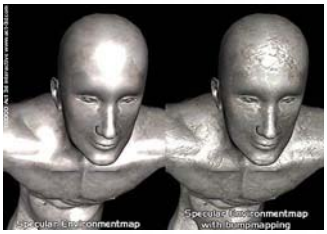
Textures

- Can be used to provide, e.g.,
  - parameters for ambient/diffuse material models



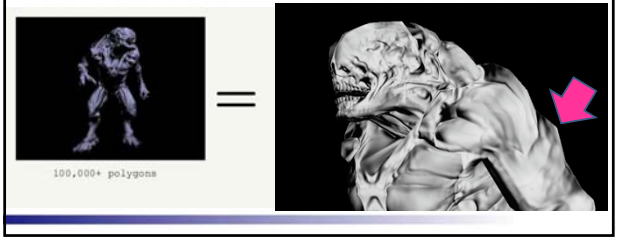
Textures

- Can be used to provide, e.g.,
  - Colors for ambient/diffuse
  - Normals (*bump mapping*)



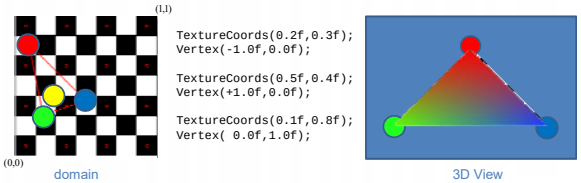
Textures

- Can be used to provide, e.g.,
  - Colors for ambient/diffuse
  - Normals (*bump mapping*)



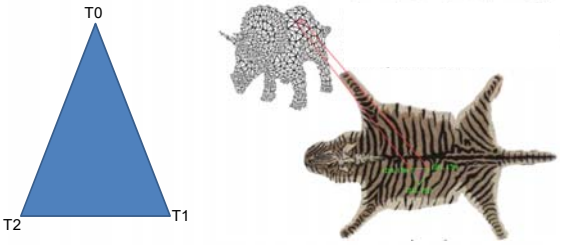
Textures

- Image mapped on the surface via texture coordinates
  - Specified at each vertex `glTexCoord{123}{fi}`
  - Interpolated over triangles



How to define Texture Coordinates?

- We will provide tex coords by hand!



Texture coordinates are handled like attributes and interpolated over triangle!!!

Textures

- Extremely useful and efficient!
- What are the problems?
- We will see about this next time…

Questions?

Summary

- Overview of the “practical” graphics pipeline
- Graphics Pipeline
  - Clipping of elements outside the frustum
  - Shading
    - material models (Diffuse, Specular)
    - Interpolation of shading results (Flat, Gouraud, Phong)
- Textures
  - Applications
  - Texture coordinates

Questions?