

Computer Graphics: 5-The Graphics Pipeline

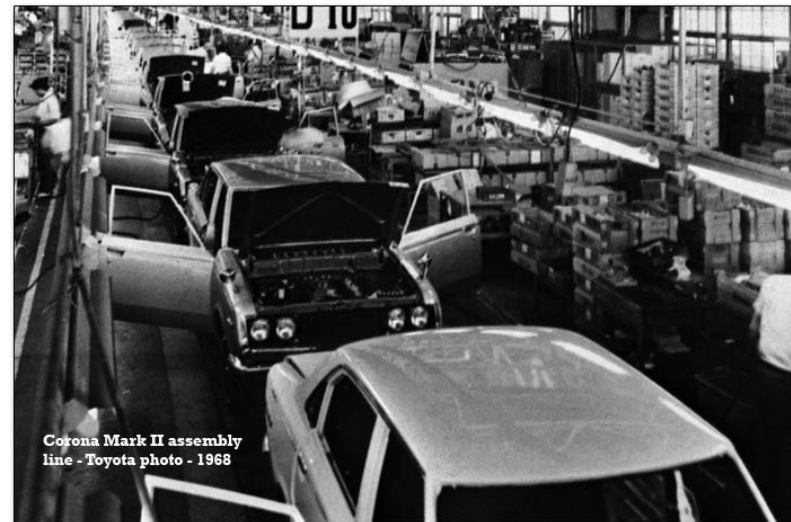
Prof. Dr. Charles A. Wüthrich,
Fakultät Medien, Medieninformatik
Bauhaus-Universität Weimar
caw AT medien.uni-weimar.de

Introduction

- Themes of the this lesson(s) will be:
 - A trip down the Graphics pipeline
 - Detailed information on some of its stages

Graphics Pipeline

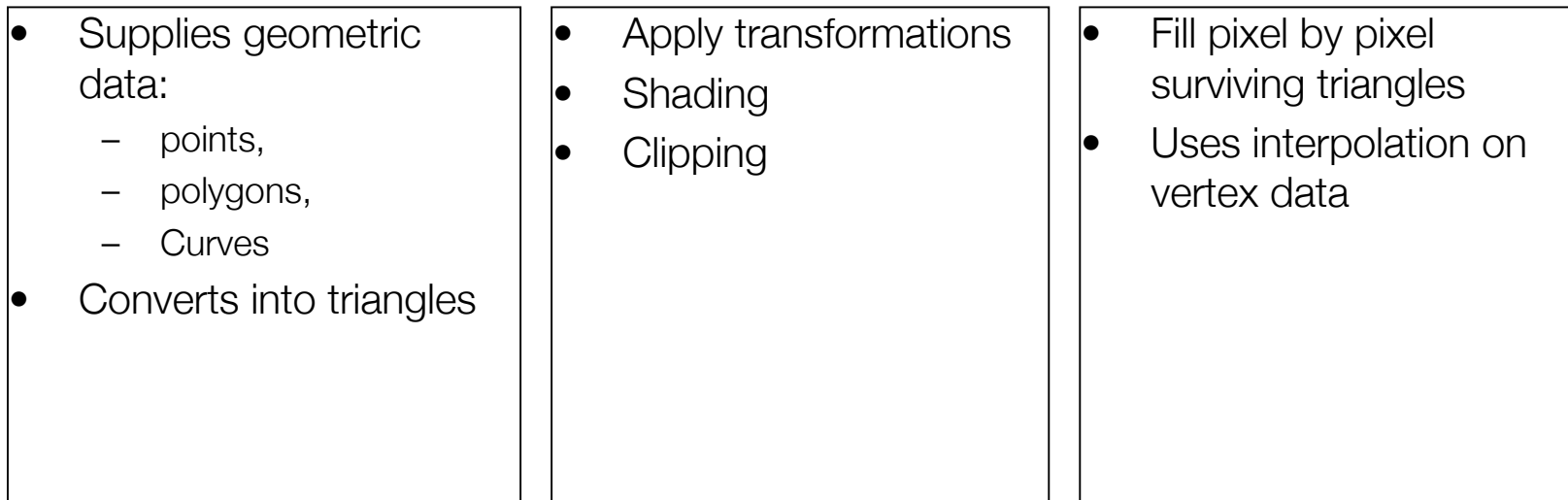
- What is the graphics pipeline?
 - Given an idea, we want to
 - Model the idea into
 - Geometry
 - Surface properties
 - Maybe model movement (for animation)
 - Generate pictures out of these models
 - Computer Graphics people have defined a workflow for generating pictures
 - Repeat over and over for each rendered picture one of a scene or animation
 - This process is done in stages, just like at a car factory



Copyright © 1968
Toyota Motor Corporation

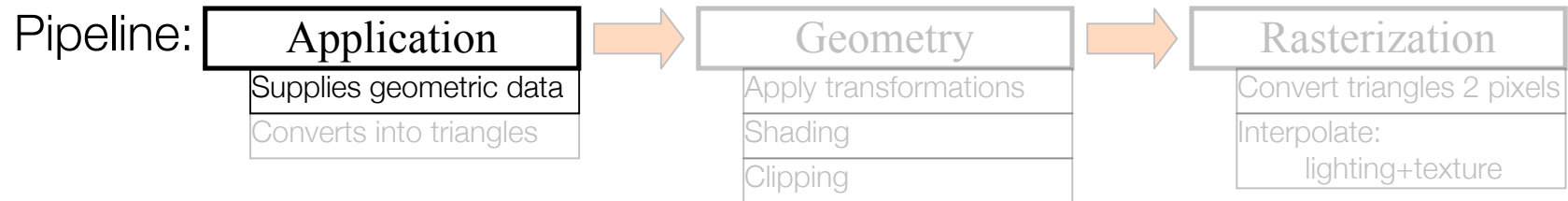
Graphics Pipeline: Overview

- The Graphics Application pipeline

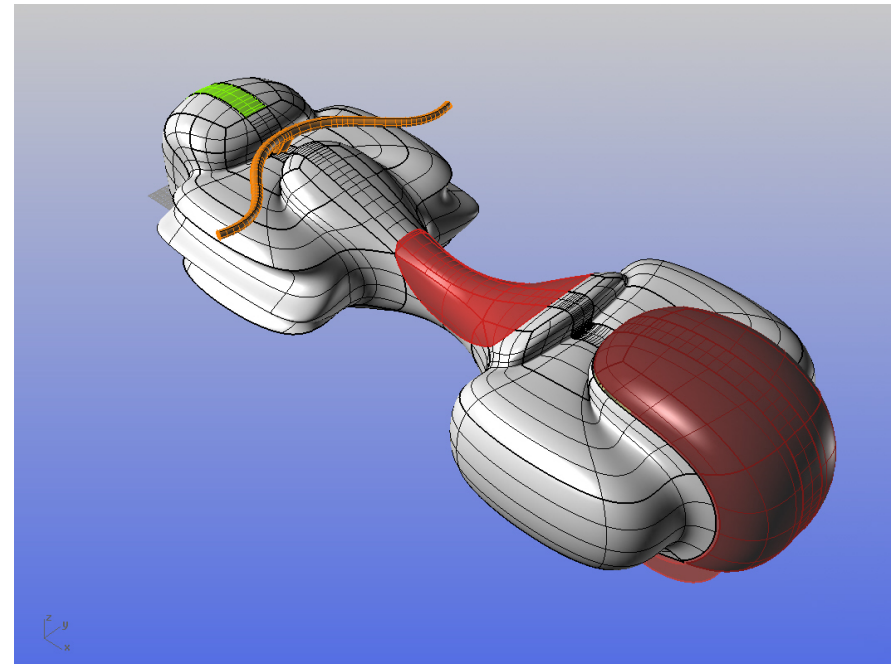


- Let us explain these different stages

Graphics Pipeline: Supply geometric data

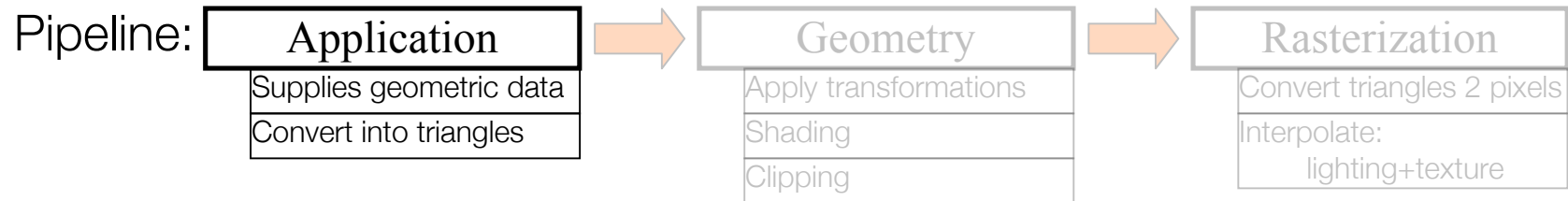


- Modeling application supplies:
 - Objects
 - Lights
 - Surface properties:
 - Colour: basic colour, diffuse and specular reflection
 - Textures
 - Surface characteristics: bumps, transparency

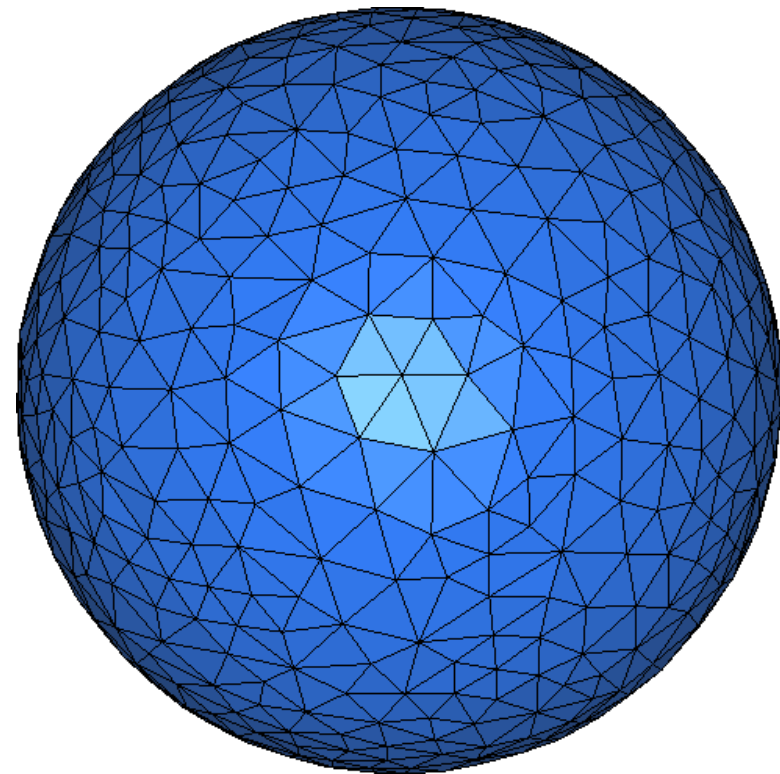


Courtesy © Rhino3D

Graphics Pipeline: Convert into triangles

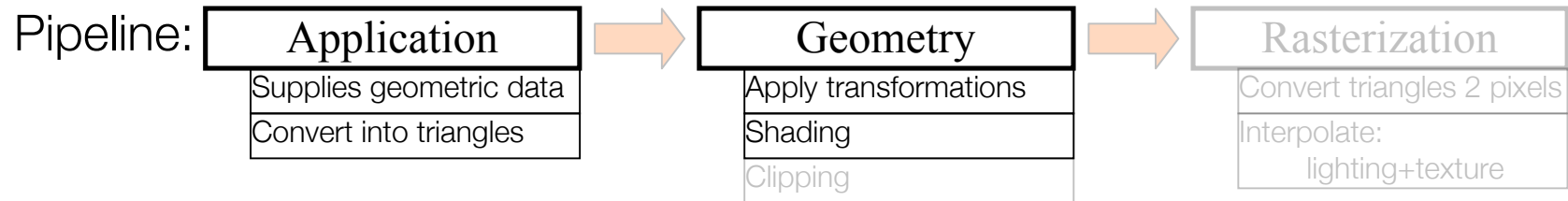


- First thing:
 - Everything triangles!
- Why?
Simple!
 - They are polygons
 - They have always the same number of sides and vertices
 - This will get us into some trouble later...

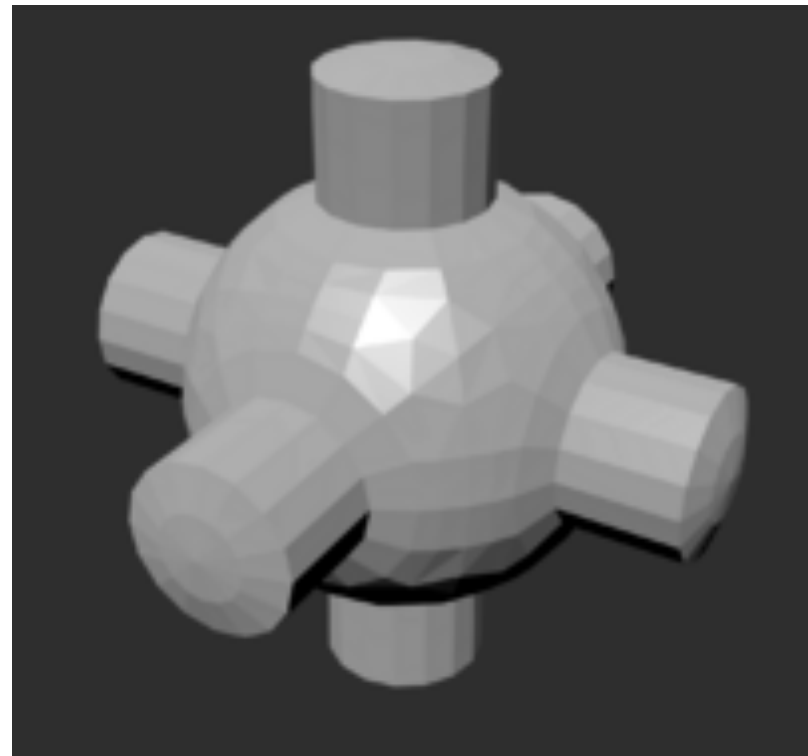


Courtesy © cgal.org

Graphics Pipeline: Shading



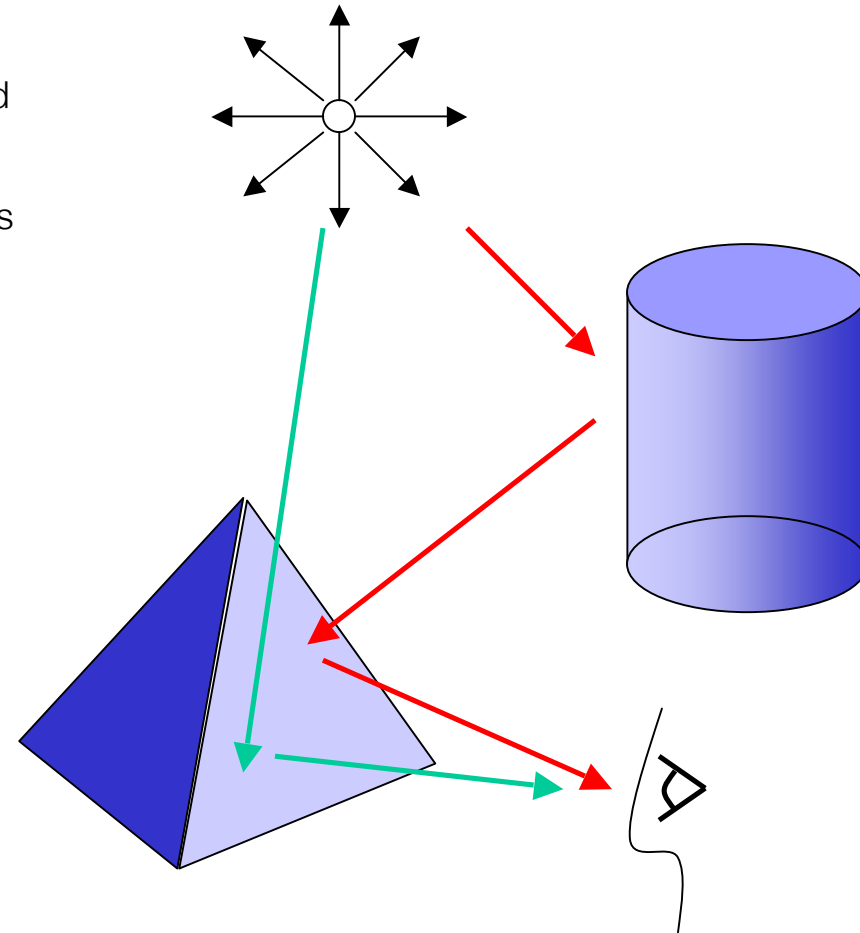
- Shading concerns simulating the interaction of light with the objects.
- The interaction is ruled by the illumination equation, which describes the resulting colour of the object.
- In its simplest form, one illumination value per triangle is computed
- But how is it computed?
 - To know this, we have to digress a bit...



Courtesy © blender.org

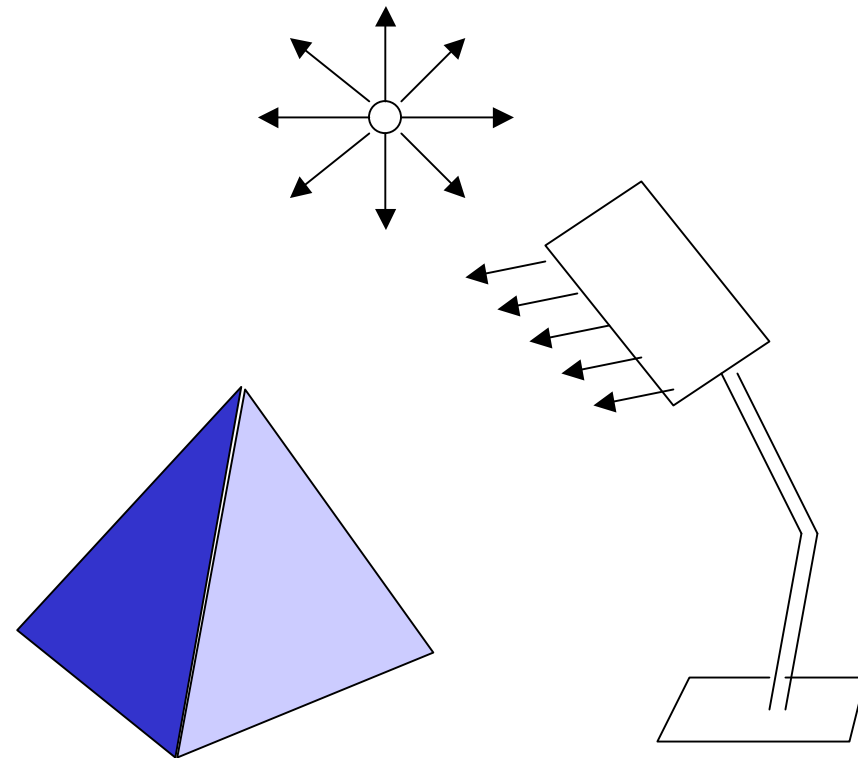
Illumination models

- There are two types of illumination models in Computer Graphics
 - Local illumination models:
 - Light reflected by a surface (and therefore its colour) is dependent only on the surface itself and the direct light sources
 - Global illumination models:
 - Light reflected by a surface is dependent only on the surface itself, the direct light sources, and light which is reflected by the other surfaces on the environment towards the current surface



Lights

- To illuminate a surface, light is needed
- Two major models for light sources
 - Point light sources
 - By putting point light source at ∞ one can simulate solar light
 - Diffusion cone can be restricted to simulate spotlights (use additional filter function for dimming light)
 - Area light sources (distributed)

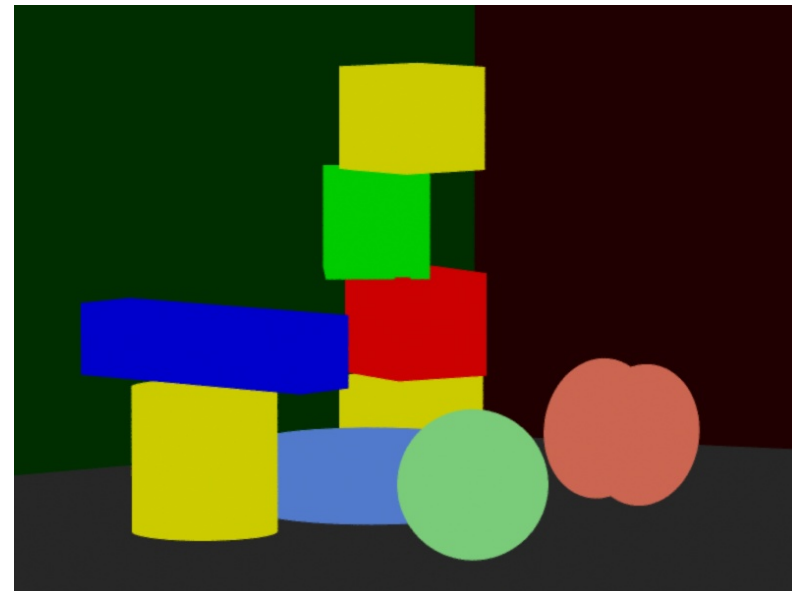


Illumination models

- For any object in the environment, a shading function describing how its surface reacts to light is necessary
- For the environment, an illumination model is used, which determines what parts of the shading functions are used while rendering the scene
- This illumination model is expressed through an illumination equation
- Given a surface and an illumination model, through the illumination equation the color of its projected pixels on the screen can be computed
- For local illumination, the Illumination Equation is composed of different terms, each adding realism to the scene

Ambient light

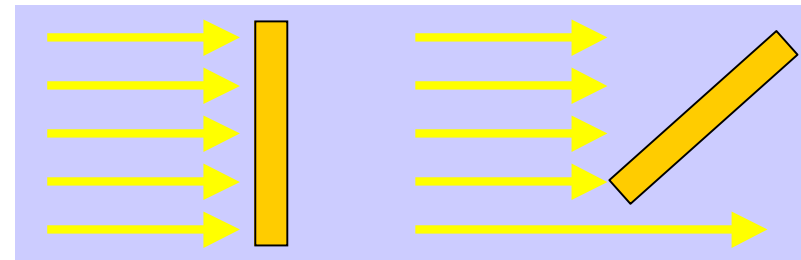
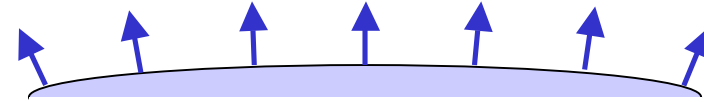
- In local illumination models, ambient light is used to model the light that does not come directly from a light source
 - i.e. under the table
 - This is the basic colour of an object, to which the other components will be added
- Illumination equation:
$$I = k_a I_a$$
where
 - k_a : says how much of ambient light is reflected by the object ($\in [0, 1]$)
 - I_a : Intensity of ambient light, equal for whole environment



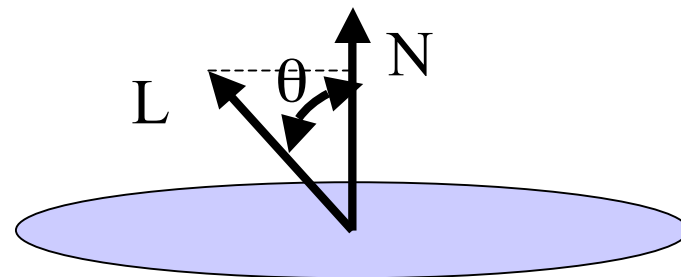
Copyright Marko Meister,
Bauhaus-University Weimar

Diffuse reflection: Lambert Law

- Suppose one has a directed light source (point or sunlight)
- Some materials reflect the light equally in all directions
 - Example: chalk
- Lambert observed that the more the incident angle to a surface parts from the surface normal, the darker the colour of the surface
- He also noticed, that objects show the same intensity even if the viewer is moving around
- The reason is that viewers perceive the same amount of light per angle on the retina, no matter what their viewing angle



- Reflected light intensity must therefore be dependent from the projection of the light vector onto the normal to the surface



Diffuse reflection: Lambert Law

- Thus, the total reflected light is given by

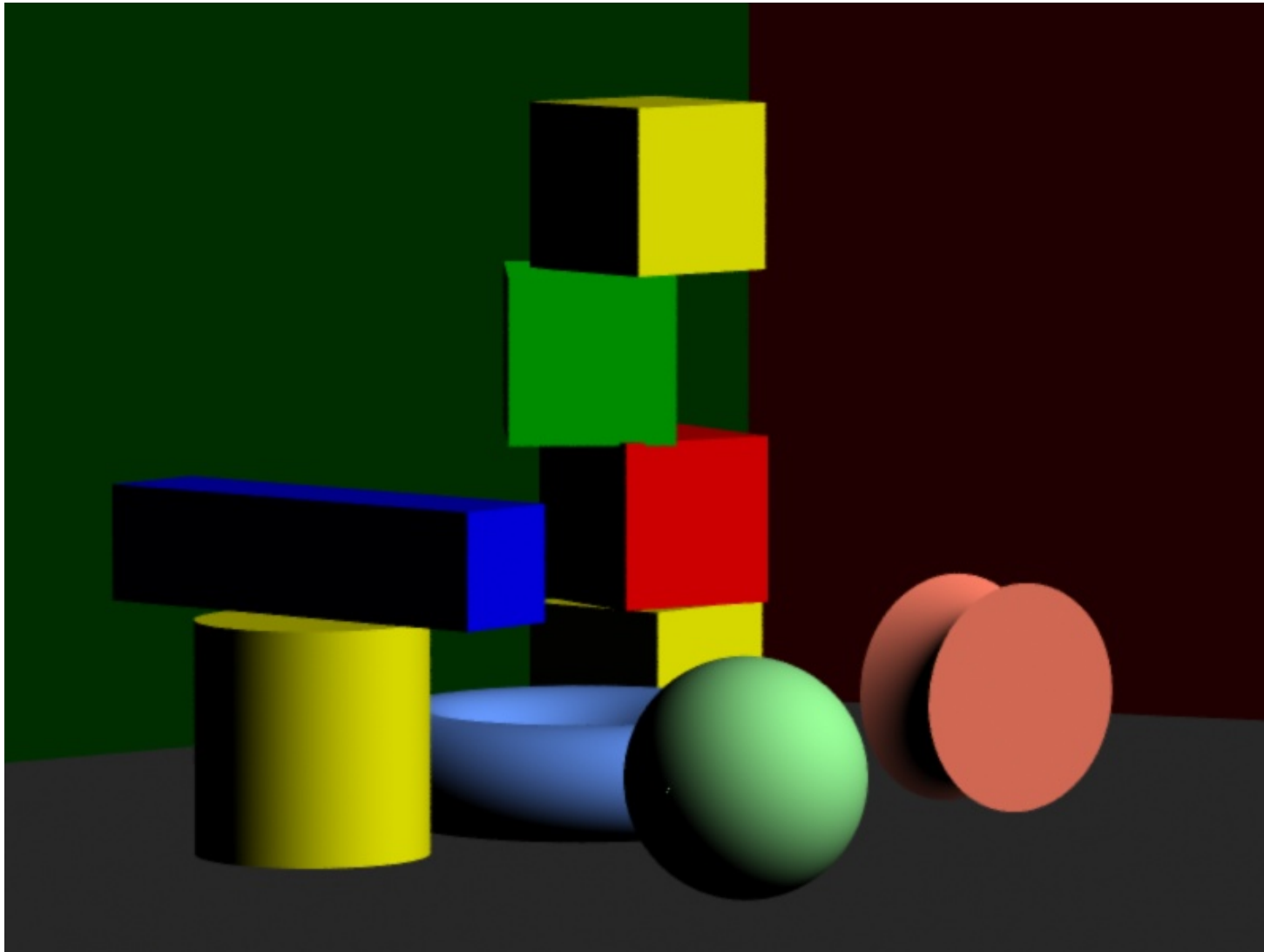
$$I_{\text{diff,Lam}} = I_p k_d \cos\theta$$

where

- I_p : Intensity of incident light at surface
 - k_d : diffuse reflection coefficient of the material $\in [0,1]$
 - Θ : angle between normal to surface and direction of incident light
- Or, by using normalized vectors N and L

$$I_{\text{diff,Lam}} = I_p k_d (\mathbf{N} \cdot \mathbf{L})$$

Diffuse reflection: Lambert Law



Copyright Marko Meister,
Bauhaus-University Weimar

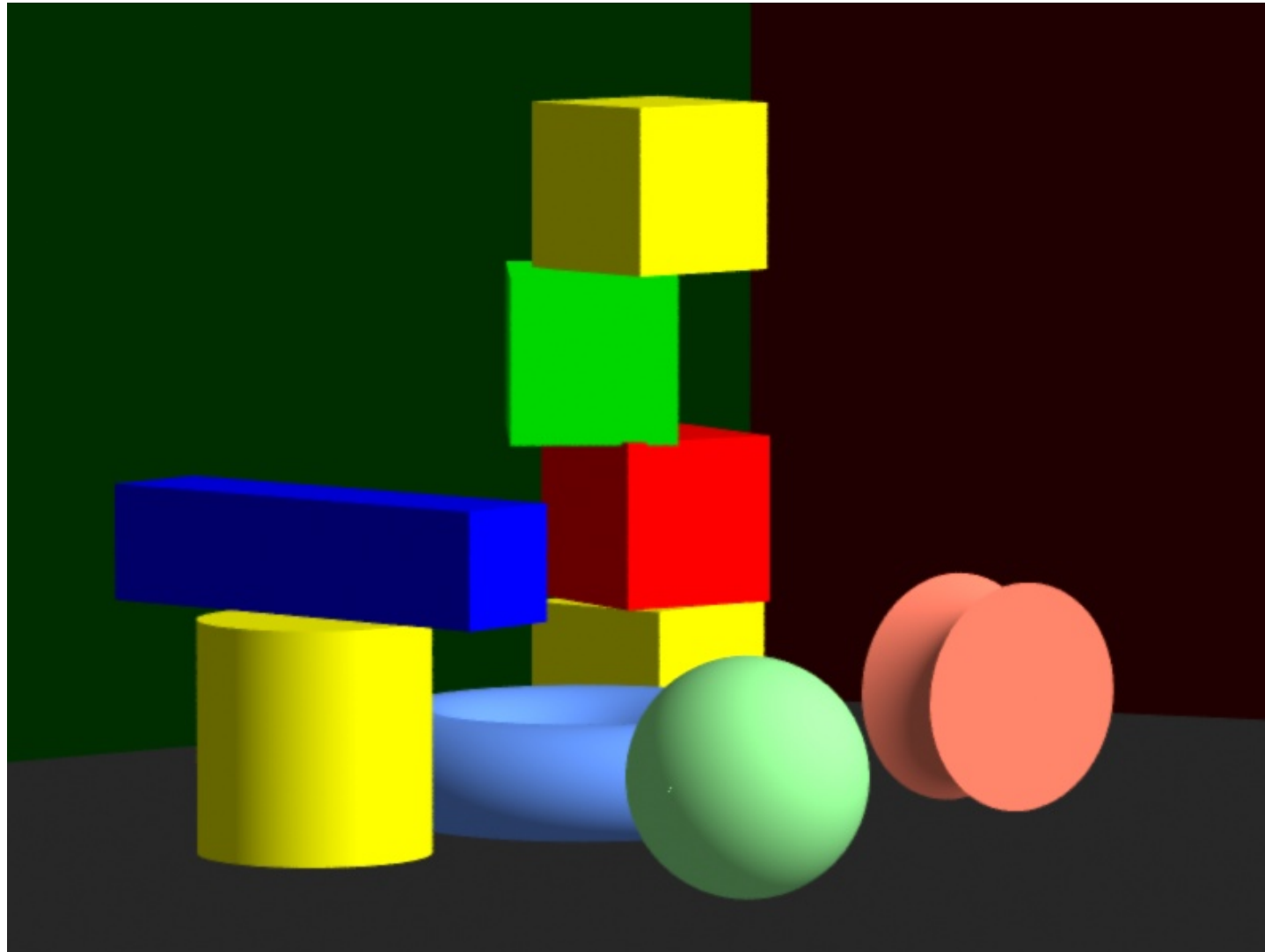
Ambient + Diffuse Illumination

- The two lighting models presented above can be combined

$$I_{\text{diff}} = I_a k_a + I_p k_d \cos\theta$$

to obtain an illumination equation that encompasses both illumination methods presented

Ambient + Diffuse Illumination



Copyright Marko Meister,
Bauhaus-University Weimar

Light source attenuation

- In fact, light does not travel through space keeping its illuminating power at the same level.
- Farther objects get less light than closer ones, because it is partially absorbed by particles in the air
- If constant light intensity is used, then one would get a kind of illumination which is similar to the sunlight
- To solve this problem, an attenuation factor is added to the illumination equation, decreasing light intensity with distance from the light source

- The resulting illumination equation is

$$I = I_a k_a + f_{att} I_p k_d (\overline{N} \cdot \overline{L})$$

where the attenuation factor is:

- $f_{att} = \frac{1}{d_L^2}$
which gives a too hard decay of light
- or $f_{att} = \frac{1}{\text{Max}(|c_1 d_L^2 + c_2 d_L + c_3|, 1)}$

where the coefficients are chosen ad hoc

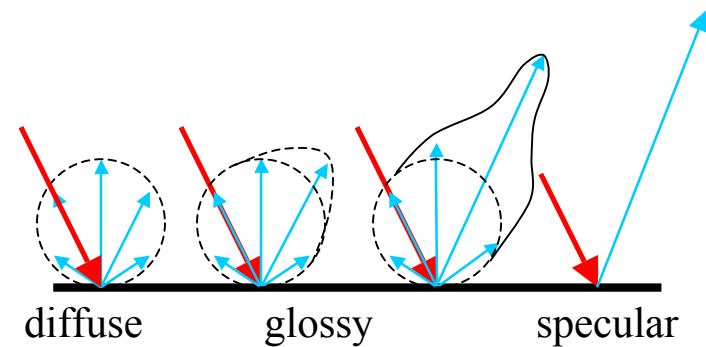
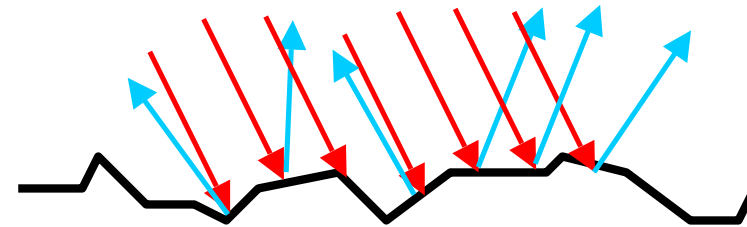
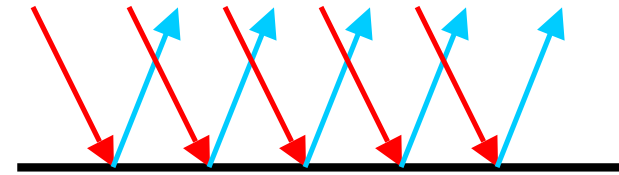
Adding colour

- In order to add colour, the computation of the illumination function is repeated three times, one for each of the colour components RGB.
- The illumination equation is therefore repeated for each one of the three components, and colour components are computed separately
- In general, for a wavelength λ in the visible spectrum, we have that

$$I_{\lambda} = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} k_d O_{d\lambda} (\bar{N} \cdot \bar{L})$$

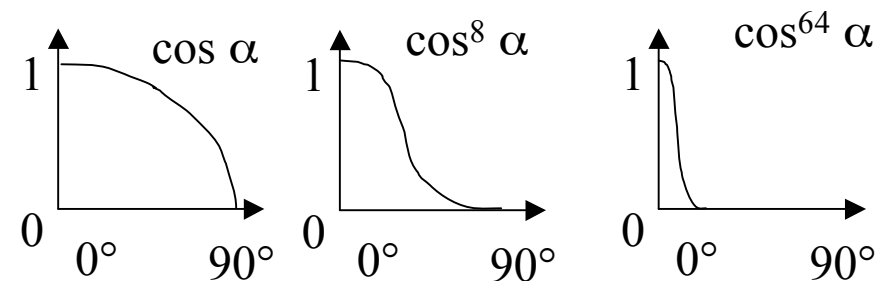
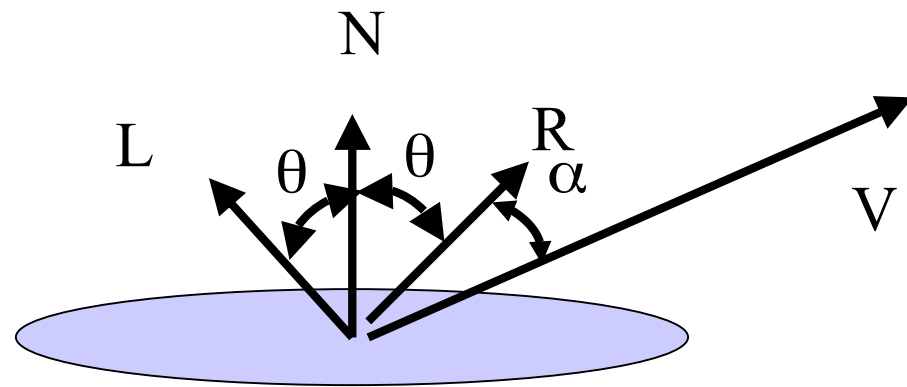
Specular highlights

- In real life, most surfaces are glossy, and not matte
 - Think for ex. at an apple
- Gloss is due to the non-plainness at the microscopic level of surfaces (microfacet theory) , and do micro reflecting surfaces of the material
- Since their distribution is approximately gaussian, then most mirror like reflected light is reflected in direction of the specular reflection, and some is scattered in other directions



Specular highlights

- Around the direction of the viewer, reflection rays are scattered and generate a highlight
- Empirically, scattering decay from the direction of the viewer can be seen to behave similarly as the power of the cosine of the angle α



Specular light

- Heuristic model proposed by Phong Bui-Tuong
- Add term for specular highlight to equation

$$I_{\lambda} = I_{a\lambda} k_a O_{d\lambda} + f_{att_i} I_{p\lambda_i} \left[k_d O_{d\lambda} (\bar{N} \cdot \bar{L}) + k_s O_{s\lambda} (\bar{R} \cdot \bar{V})^p \right]$$

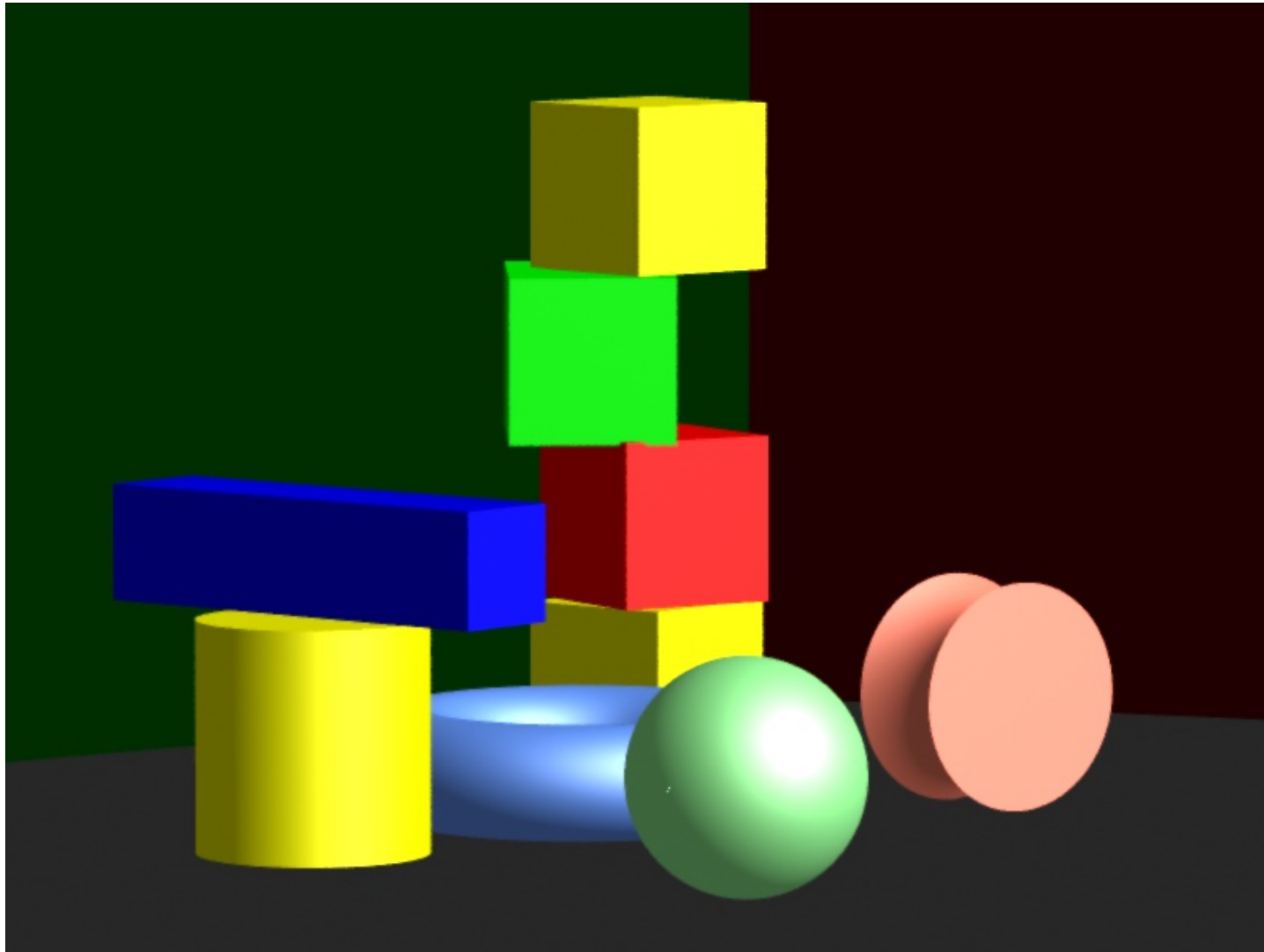
where

- k_s = specular reflection constant of surface
- $(\bar{R} \cdot \bar{V})^p = \cos^n \alpha$

Specular highlights

- Applying Phong illumination allows to obtain quite convincing images
- Note: these illumination models are implemented in hardware nowadays

Specular highlights

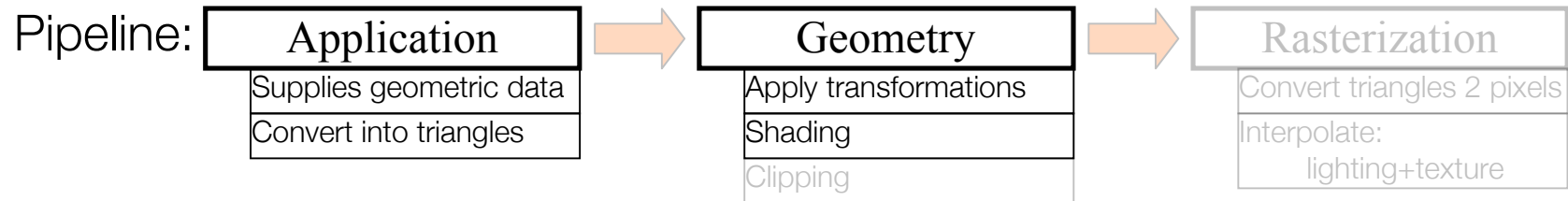


Copyright Marko Meister,
Bauhaus-University Weimar

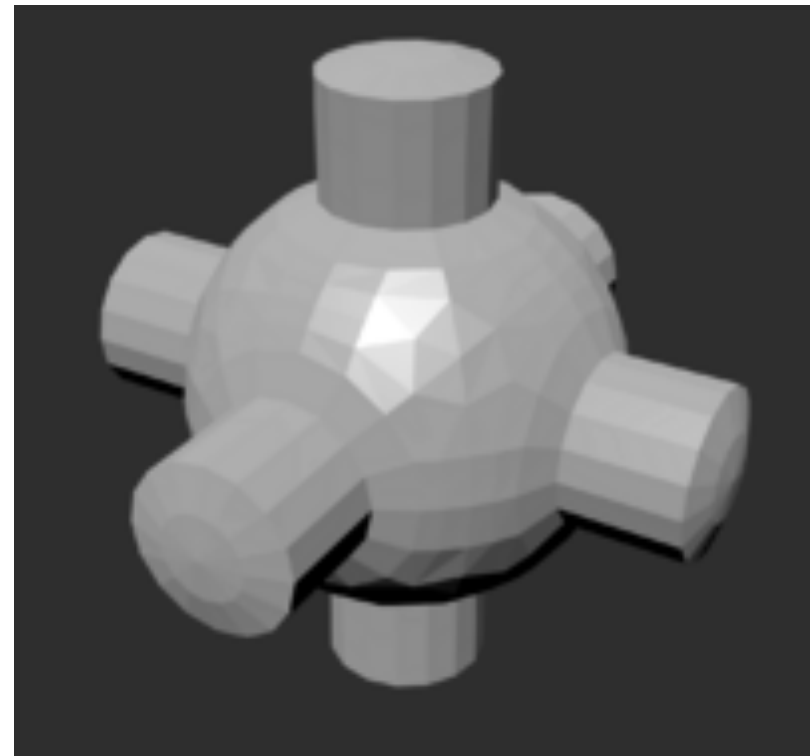
Multiple lights

- Adding multiple light sources to the illumination model in an environment is simple:
 - Just add contributions due to the single lights
 - This, of course, doubles the computations in the case of two lights

Graphics Pipeline: Shading

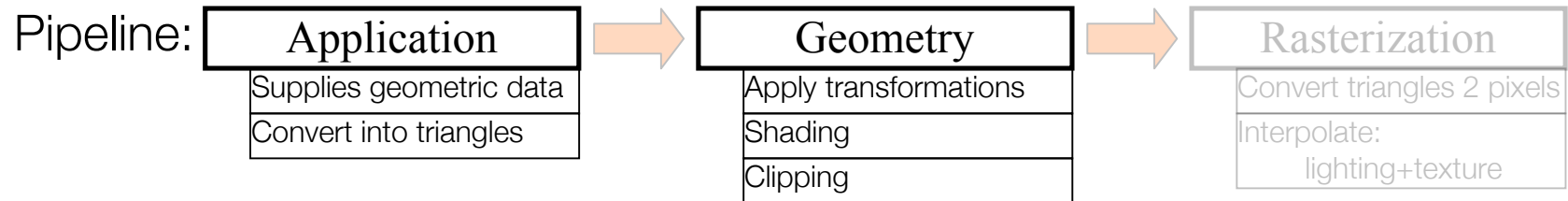


- Shading concerns simulating the interaction of light with the objects.
- The interaction is ruled by the illumination equation, which describes the resulting colour of the object.
- In its simplest form, one illumination value per triangle is computed
- But how is it computed?
 - To know this, we have to digress a bit...

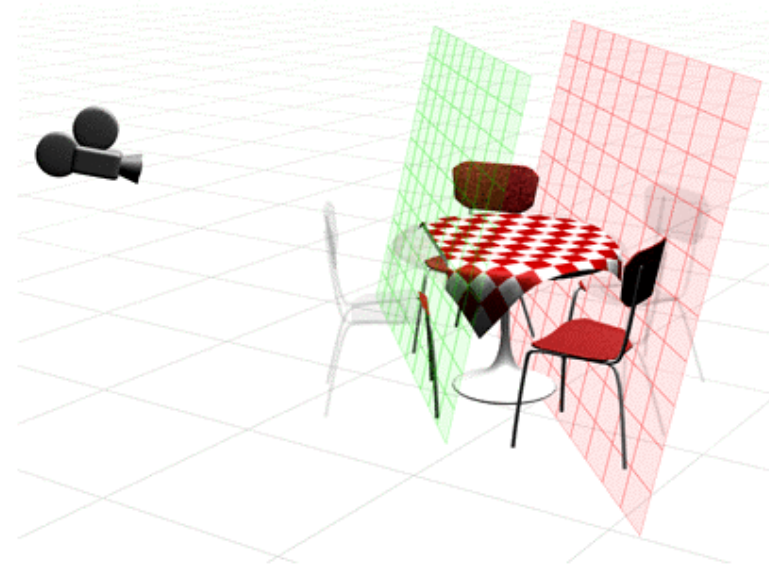


Courtesy © blender.org

Graphics Pipeline: Clipping

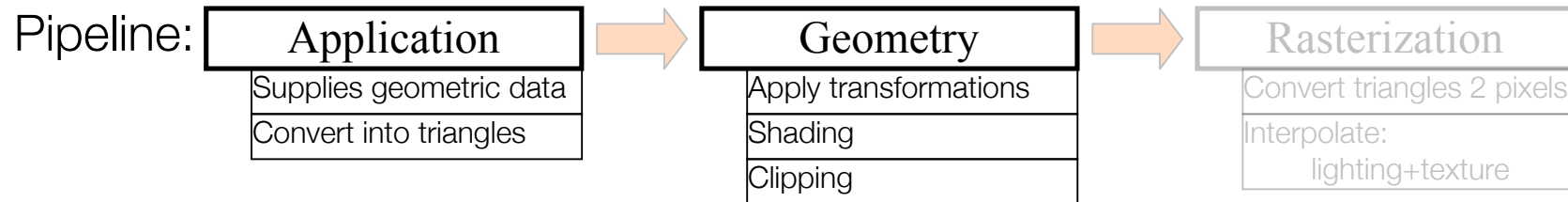


- Clipping “clips” the scene and eliminates the polygons that do not need to be displayed
- There are two clippings being done:
- 3D clipping, eliminating all polygons
 - Farther than the far plane
 - Nearer than the near plane

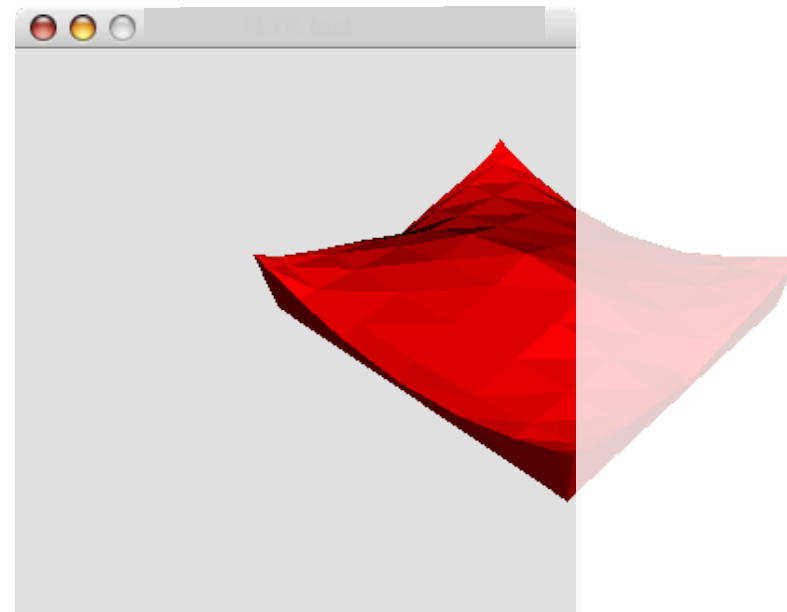


Courtesy © Autodesk

Graphics Pipeline: Clipping

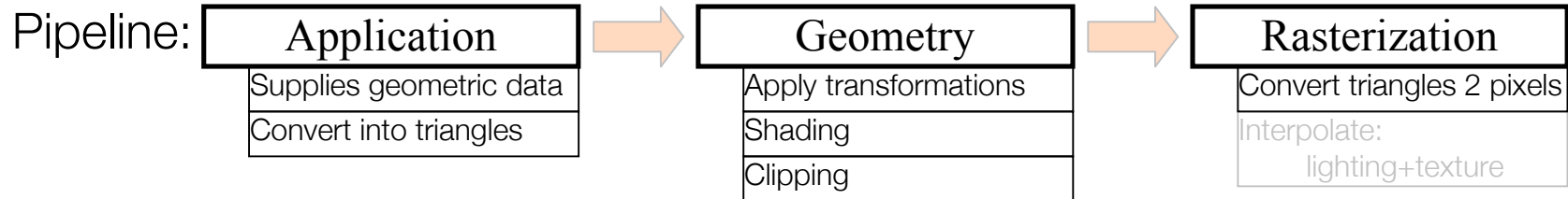


- 2D clipping, eliminating the polygons and lines I cannot view in the image window and would therefore eat up computing power

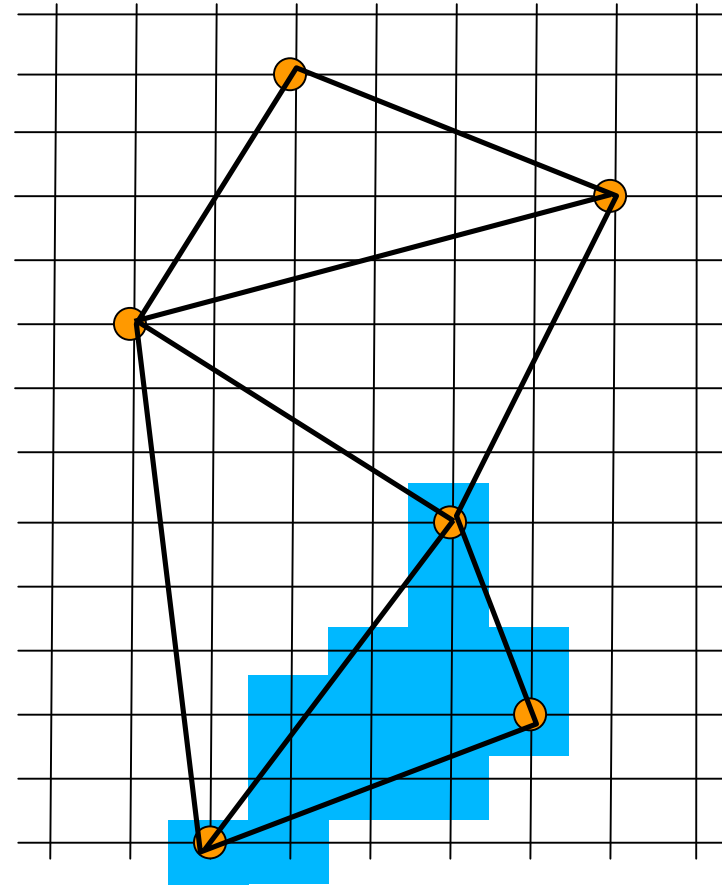


Courtesy Stephen Chenney,
University of Wisconsin at Madison

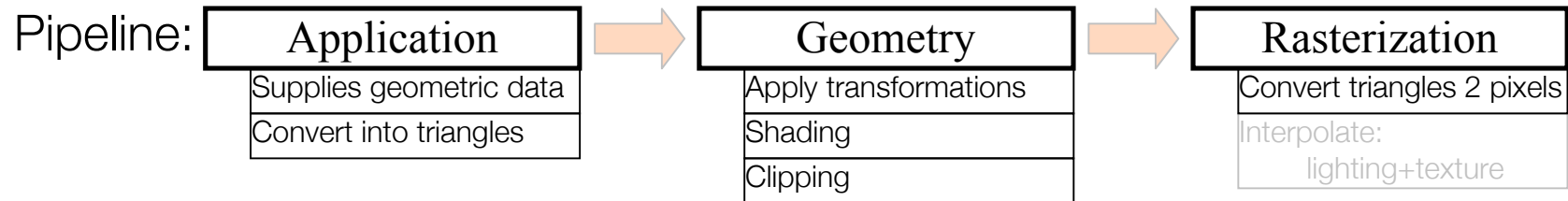
Graphics Pipeline: Triangle Rasterization



- While nature is continuous, screens are made of pixels:
 - Placed on a square grid
 - Integer coordinates
- Convert the triangles into pixels is a complex operation:
 - Find out which pixels have to be coloured
 - At the same time, draw only triangles which are not covered by other triangles

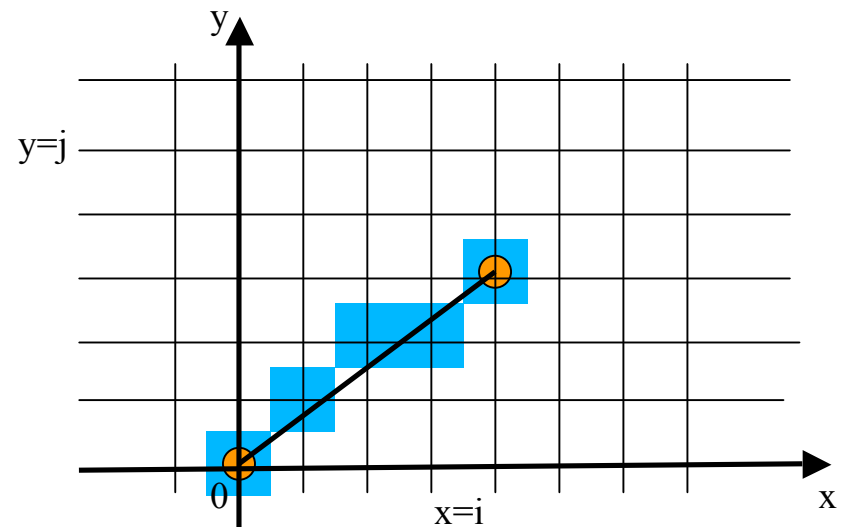


Graphics Pipeline: Triangle Rasterization

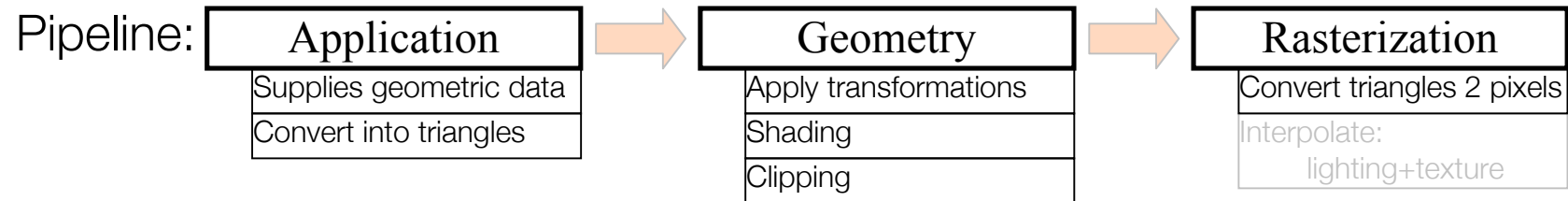


- First we have to know how to draw a line to draw the borders
- Line through $P_I=(x_I, y_I)$, $P_F=(x_F, y_F)$:

$$y = \frac{y_F - y_I}{x_F - x_I} x + y_I - \frac{y_F - y_I}{x_F - x_I} x_I$$
 or, more simply: $y=mx+q$.
- Let us find the pixels we need to switch on to draw the line
- Let us start from $(0,0)$, i.e. let us suppose that $P_i=(0,0)$ and $P_F=(4,3)$
- I need to find the intersections of the line $y=3/4x$ with the lines $x=j$ between 0 and 4



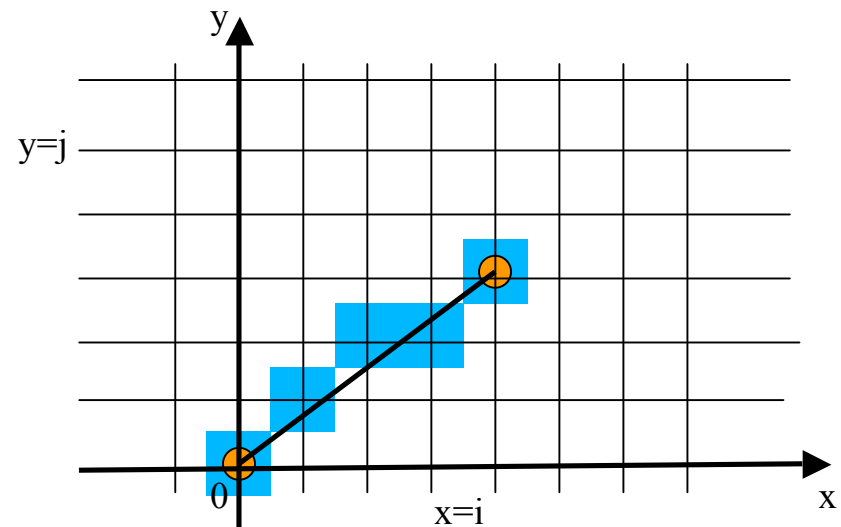
Graphics Pipeline: Line Rasterization



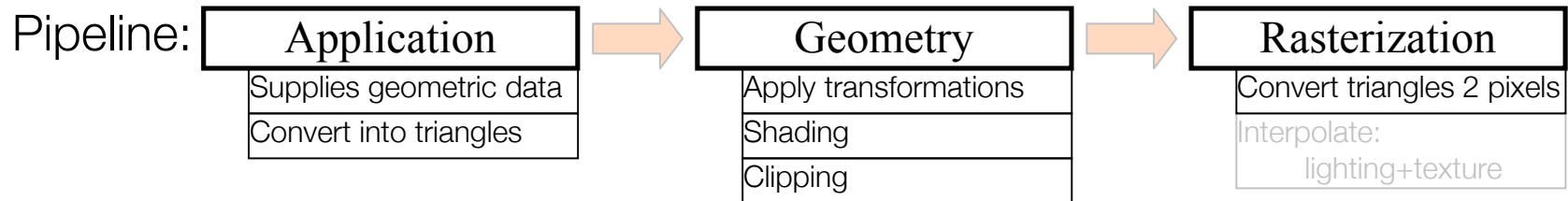
- Let's compute: $y=3/4x$

x	y
0	0
1	$3/4=0.75$ round to 1
2	$6/4=1.50$ round to 2
3	$9/4=2.25$ round to 2
4	$12/4=3$

So I draw the pixels obtained by rounding the intersections with the vertical straight lines



Graphics Pipeline: Line Rasterization



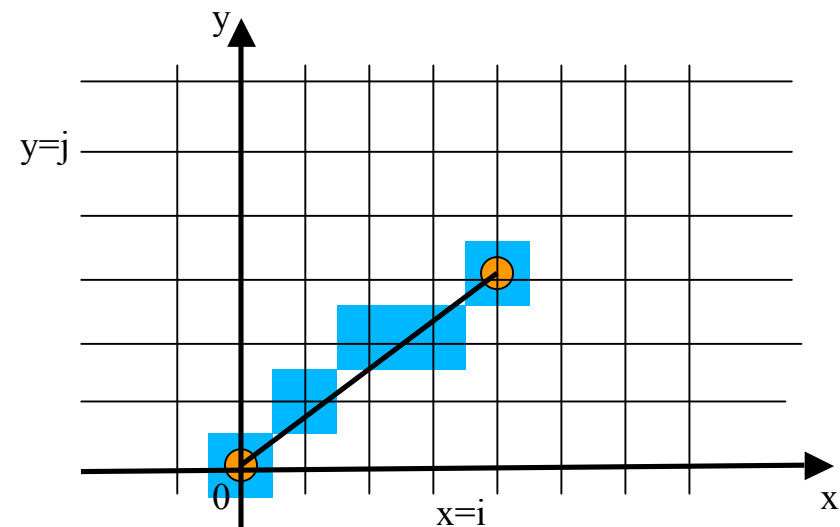
- And for a generic line? $y=mx+q$

x	y
0	q
1	m+q
2	2m+q=m+(m+q)
3	3m+q=m+(2m+q)
4	4m+q=m+(3m+q)

Hey!

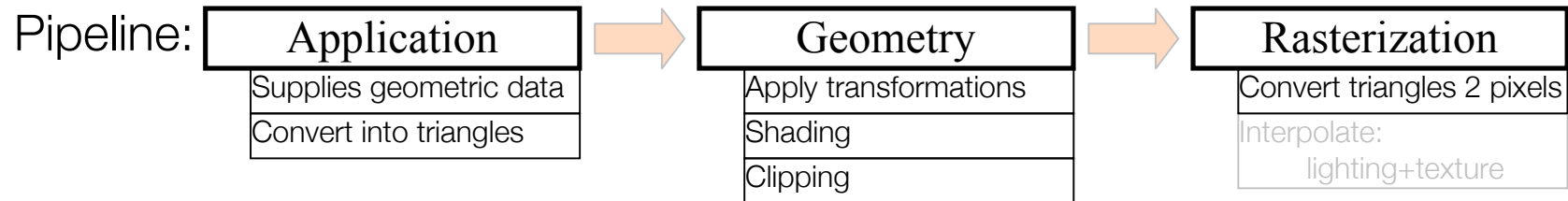
Each time I add m!!!

- This is exactly what we do to draw lines: add m each step till endpoint is reached

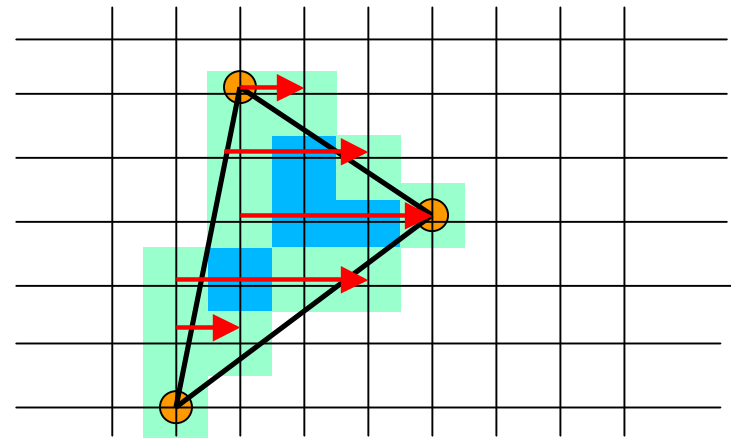


$$y = \underbrace{\frac{y_F - y_I}{x_F - x_I}}_m x + \underbrace{y_I - \frac{y_F - y_I}{x_F - x_I} x_I}_q$$

Graphics Pipeline: Triangle Rasterization

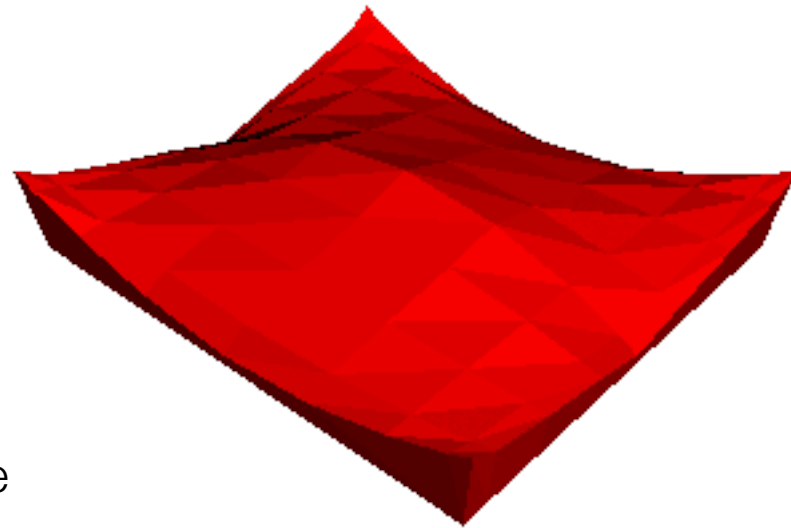


- And for triangles? How do I rasterize them?
 - First draw borders
 - Then fill one scanline at the time between border points
 - For example, from top to bottom, until triangle is finished



Flat shading

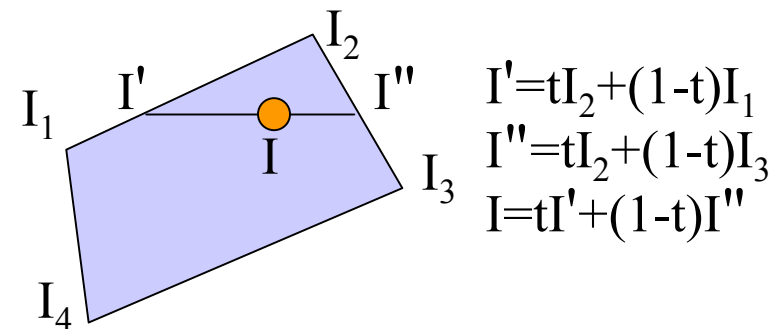
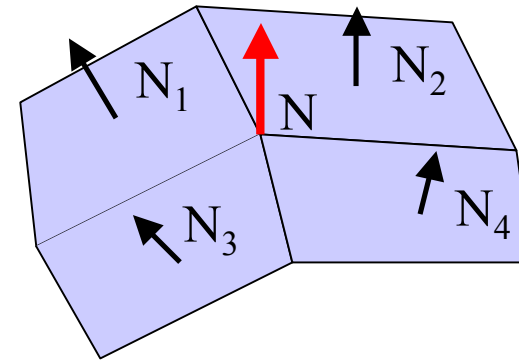
- And which colour do I draw the polygons with?
- We saw the problem of flat shading: you saw the faces!
- For each polygon in the mesh, compute illumination equation
- Render the whole polygon with the obtained colour
- Results are already good for giving an idea of the shape
- However, all polygonal facets are seeable, and this is mostly unwanted
- We will come back to this later, and learn the tricks used to avoid visible facets



Courtesy Stephen Chenney,
University of Wisconsin at Madison

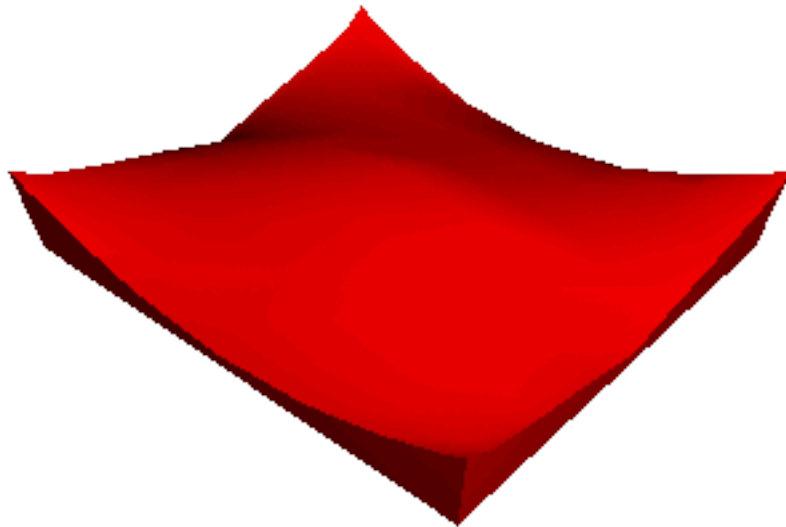
Gouraud shading

- To avoid this problem, Gouraud in 1971 proposed a method to smooth polygonal surface rendering
- The idea is to
 - compute normal vectors at vertices of each polygon (average adjacent polygon normals
 $N=N_1+N_2+N_3+N_4$)
 - compute illumination at vertices
 - linearly interpolate colour values along the edges of the polygons
 - linearly interpolate colour values between edges to find out colour at a given point
- Bilinear interpolation



Gouraud shading

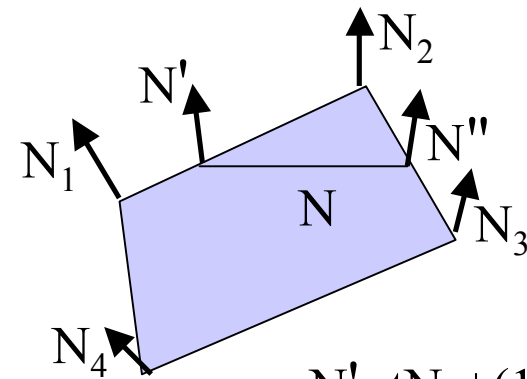
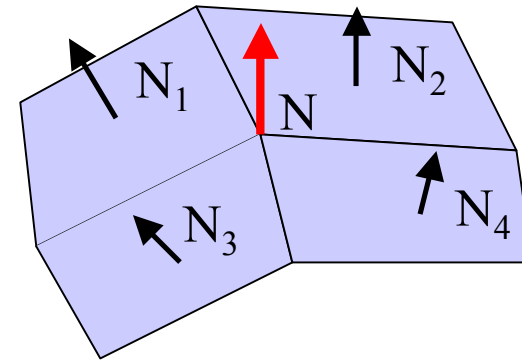
Courtesy Stephen Chenney,
University of Wisconsin at Madison



- Gouraud shading gives smooth surfaces
- Sometimes highlights are missed

Phong shading

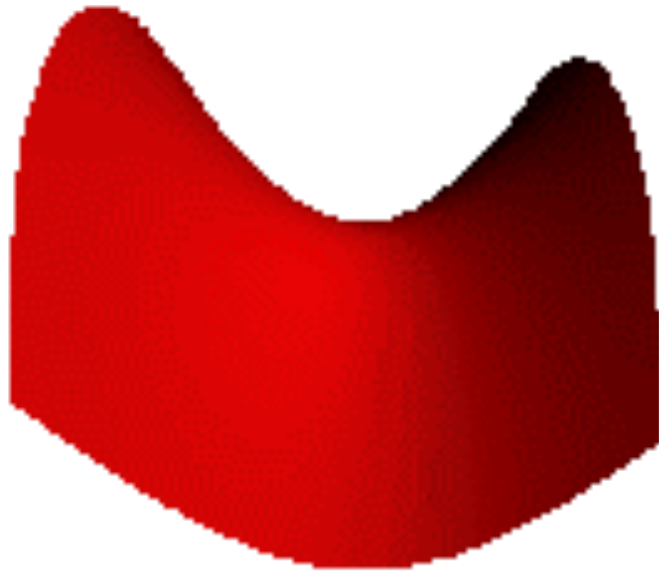
- Phong proposed an improvement to Gouraud's idea
- Phong interpolates between normals and not between colour to find the normal at a point inside the polygon
- Note that both interpolations (Phong, Gouraud) have three components to compute
- Only once known the interpolation normal at the point, illumination is computed at each point
- Much more computationally intensive than Gouraud



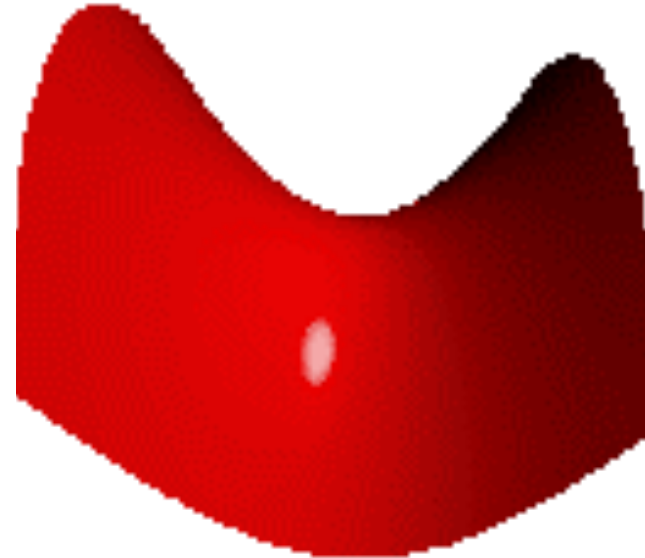
$$\begin{aligned}N' &= tN_2 + (1-t)N_1 \\N'' &= tN_2 + (1-t)N_3 \\N &= tN' + (1-t)N''\end{aligned}$$

Comparing shading methods

Gouraud

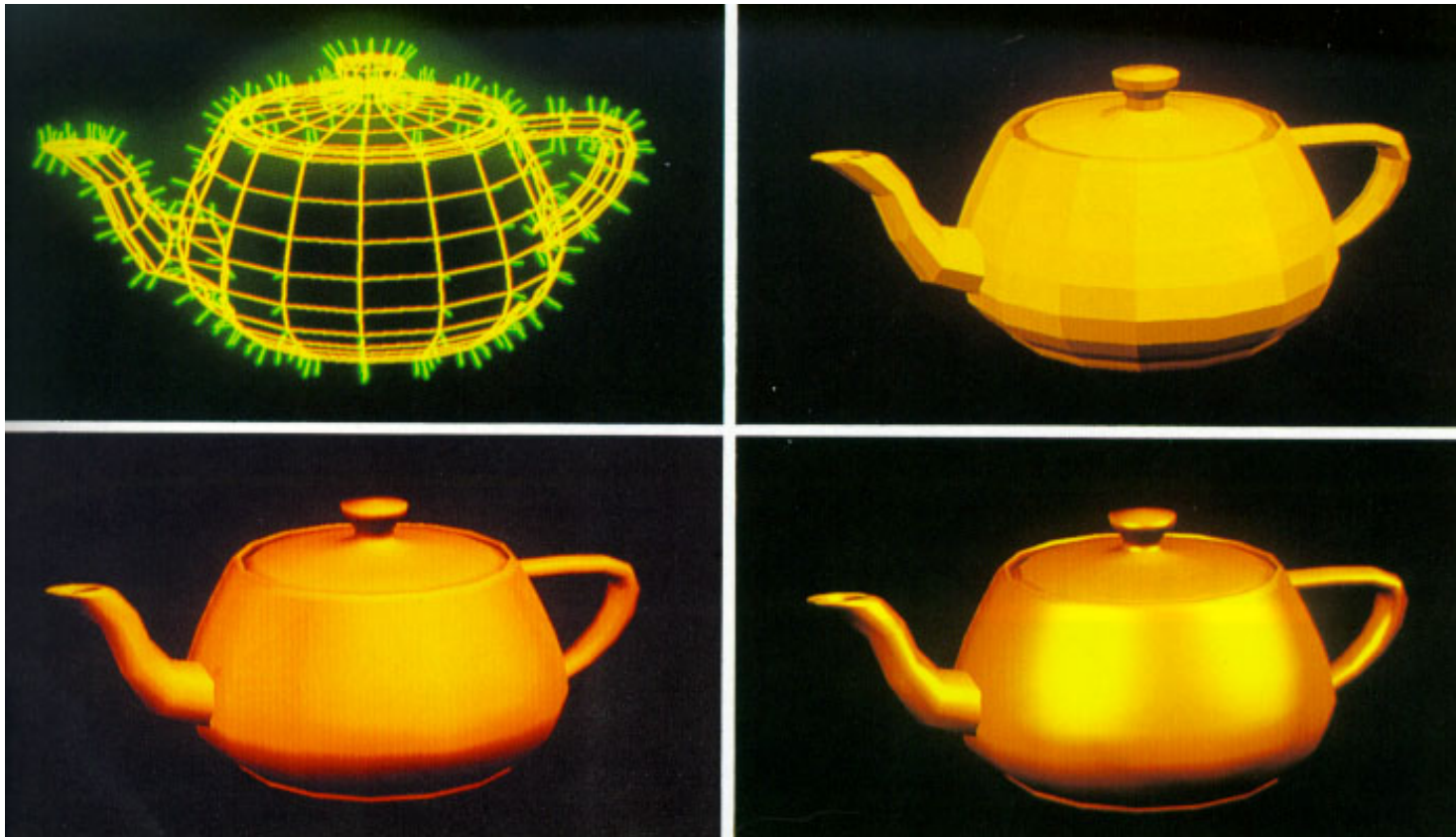


Phong

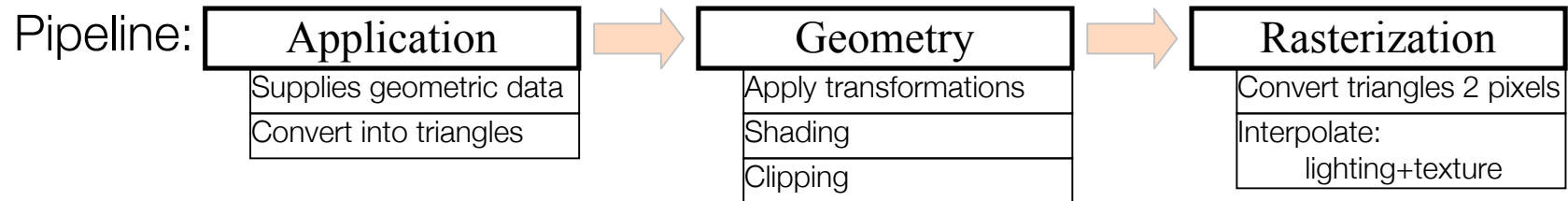


Courtesy Stephen Chenney,
University of Wisconsin at Madison

Comparing shading methods

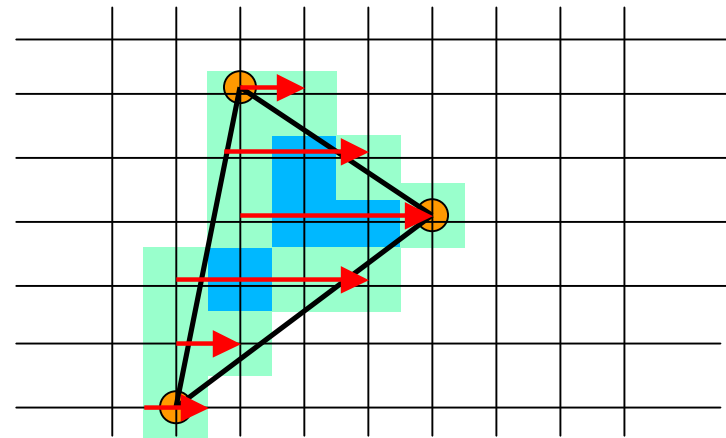


Graphics Pipeline: Z-buffering

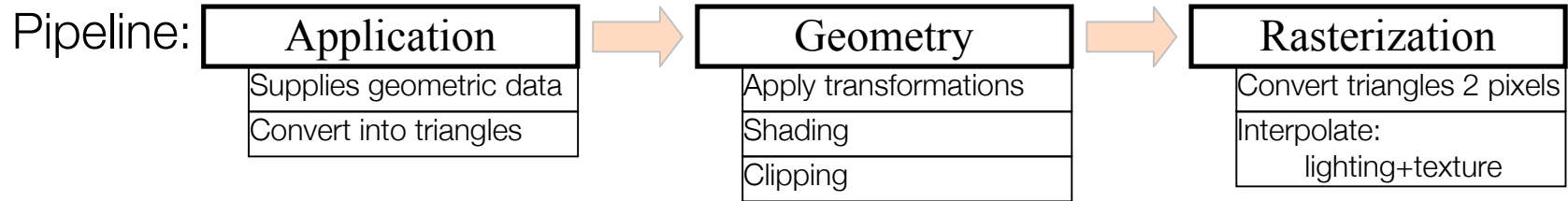


- Wait! Some triangles might cover others: *hidden surface removal*.
- How do I know which ones to draw?
- Simple: I compute the depth (=z value) of each pixel I am about to draw
- At the start, I enter infinite in a buffer (= table as big as screen) at each position of the table
- When I am about to write a pixel, I look the z value of what I am drawing is smaller of the current value at the table

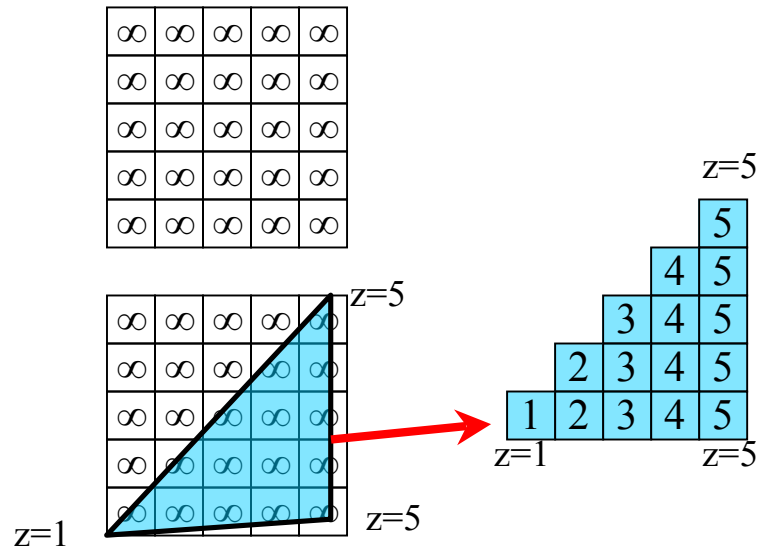
- If it is, then I write the current pixel in the image and update the value of the “parallel” table to the z value of the pixel
- If not, I leave the table untouched and do NOT write the pixel of the triangle



Graphics Pipeline: Z-buffering

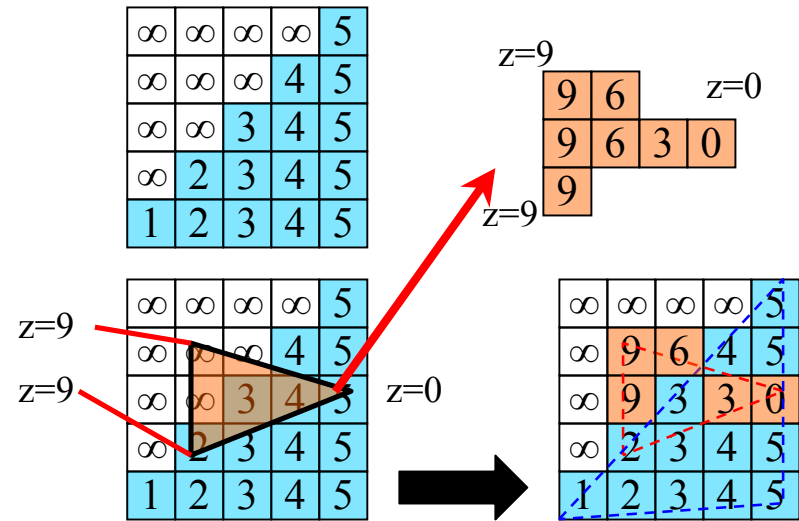


0: Empty buffer



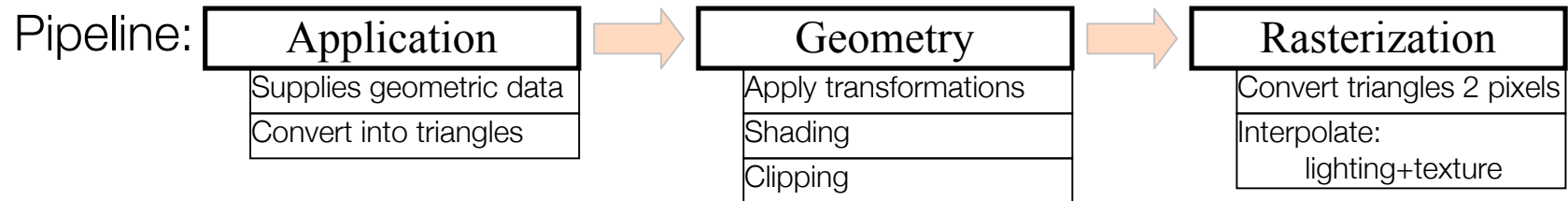
1: Setup 1st triangle

1: Draw 1st triangle

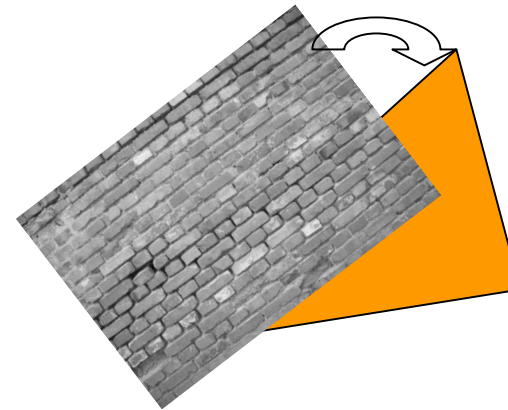


1: Draw 2nd triangle

Graphics Pipeline: Textures

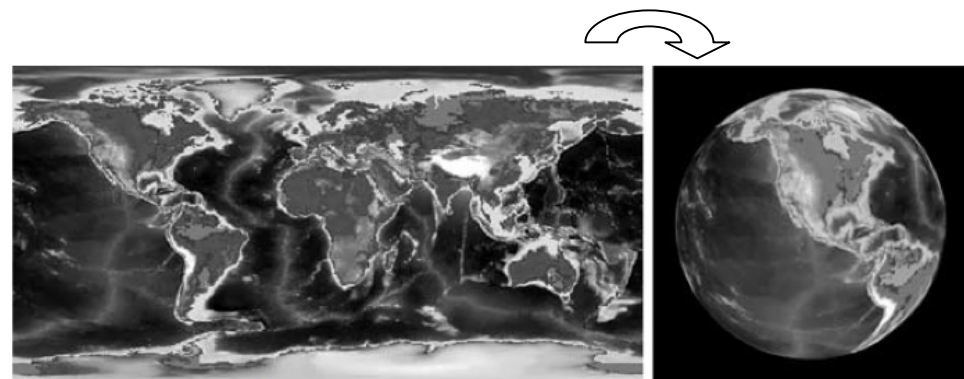
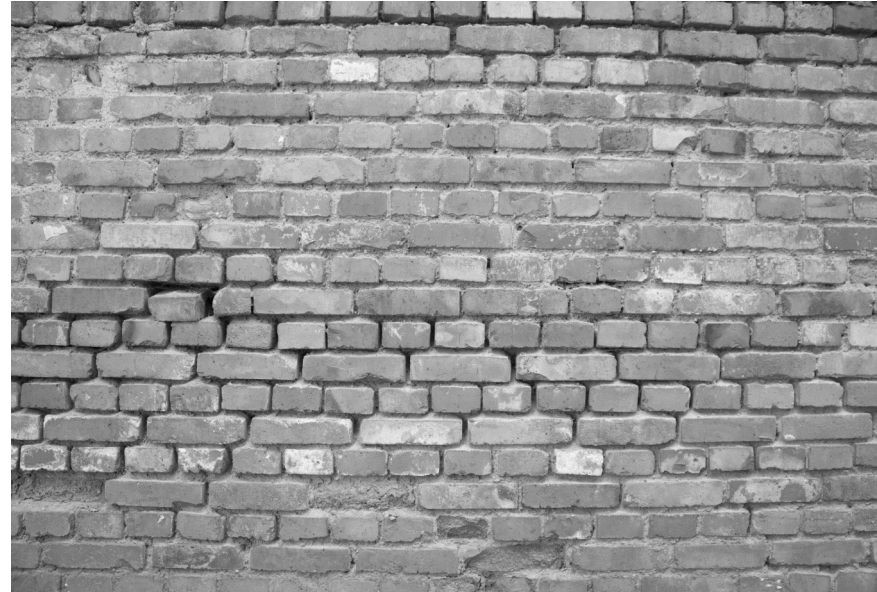


- So I compute lighting equation almost at every pixel to interpolate it with Phong shading
- This for drawing solid triangles: at each point, I take basic colours and compute illumination either by using Gouraud or Phong to obtain the colour
- And what if I want more interesting looking objects?
- I could glue wrapping paper on them!
- But how? With textures!



Textures

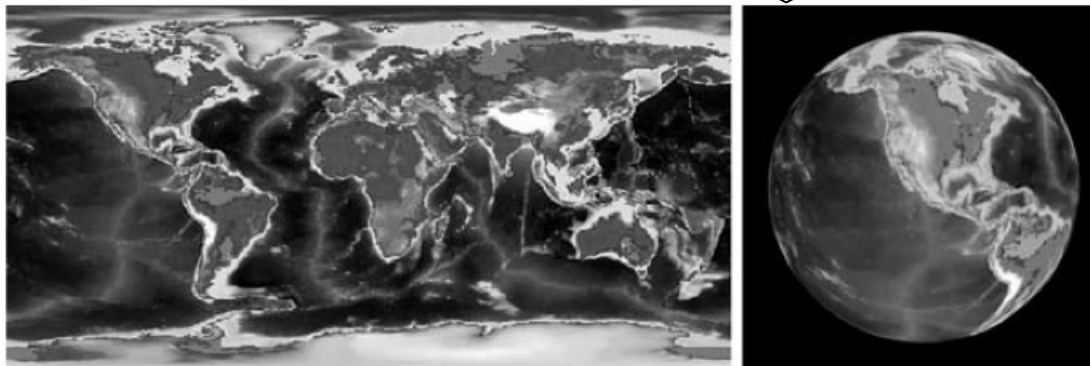
- Reality shows much more richness of surface detail than the one obtainable through local illumination: look at the wall!!
- One could model the surface with detailed geometry
- However, this would increase greatly the complexity of the model.
- A better approach is therefore to „paint“ detail on simple geometry
- The image, called *texture*, is „glued“ to a simple geometry to obtain detail
- First approaches due to Catmull (74) and Blinn & Newell (76)



Textures

- There are basically two ways of texture mapping:
 - 2D
 - 3D
- Let us look first at 2D textures
- Image data (surface pixel colors) is stored in a 2D image, the pixels of which are called *texels*
- Let's assume the coordinates of the image are called u, v and that u and v vary in the interval $[0, 1]$
- To compute what colour is reflected by the sphere, one must find a correspondence between sphere and the texture space
- Parametric sphere:

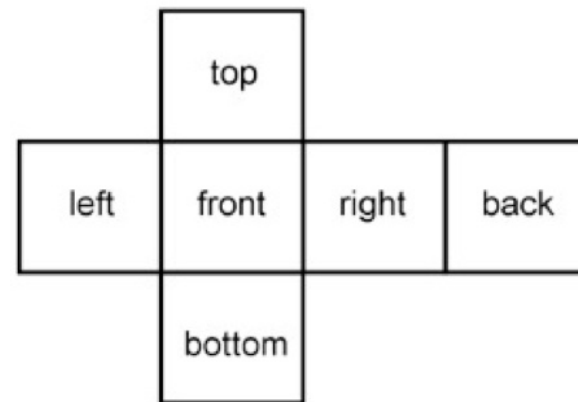
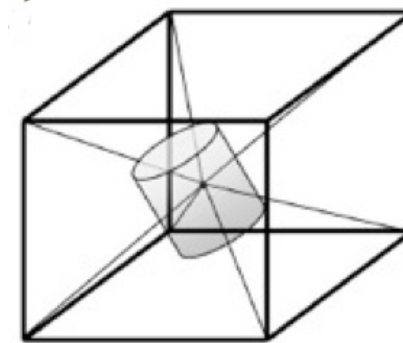
$$\begin{cases} x=x_c+R \cos\psi \sin\vartheta \\ y=y_c+R \sin\psi \sin\vartheta \\ z=z_c+R \cos\pi \end{cases}$$
 - $\vartheta = (z-z_c)/R$ } longitude
 - $\psi = \arctan((y-y_c)/(x-x_c))$ } +latitude
 - $u = \psi / 2\pi$ } to texture
 - $v = (\pi - \vartheta) / \pi$ }



- Similarly, for other simple maps
 - Cube
 - Cylinder
 - Plane

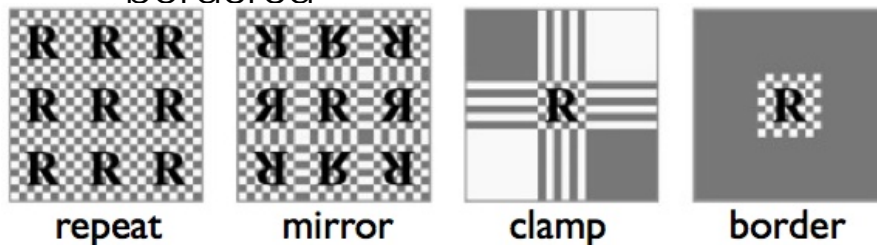
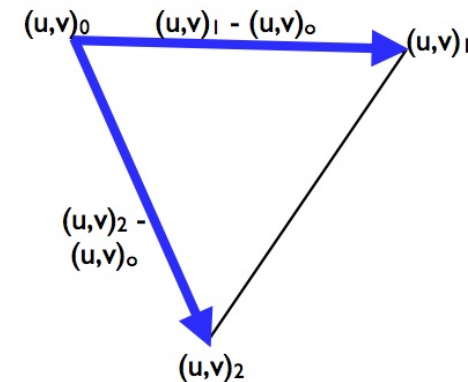
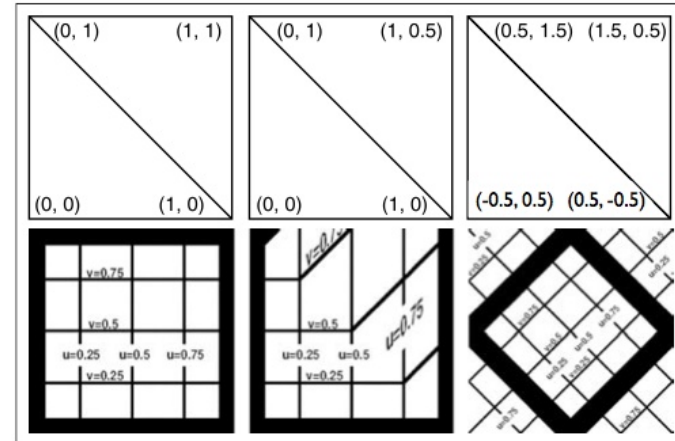
Textures

- There are different ways of applying textures for complicated objects:
 - Surround object to be textured with a simple object: cube, sphere, cylinder
 - Choose a center for the object
 - Texture the cube (invisibly)
 - When one draws pixel of the object, project this pixel to the cube, and pick the cube texture color



Textures

- And what if my object is in mesh form?
- Determine texture coordinate for each vertex of the mesh (by projection as before)
- Bilinear interpolation between vertices
 - For triangles, use baricentric coordinates (same as done for normals)
- If texture coordinates are beyond the image, then texture is repeated, mirrored, clamped or bordered

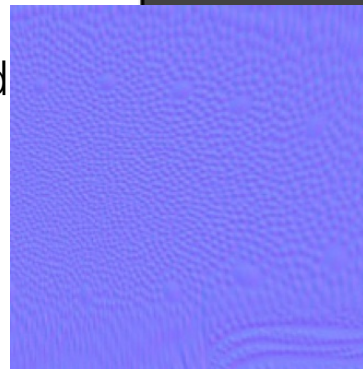
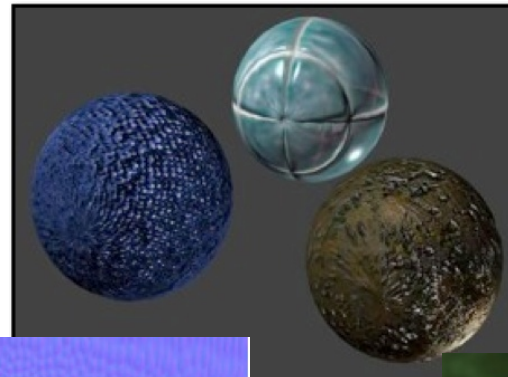
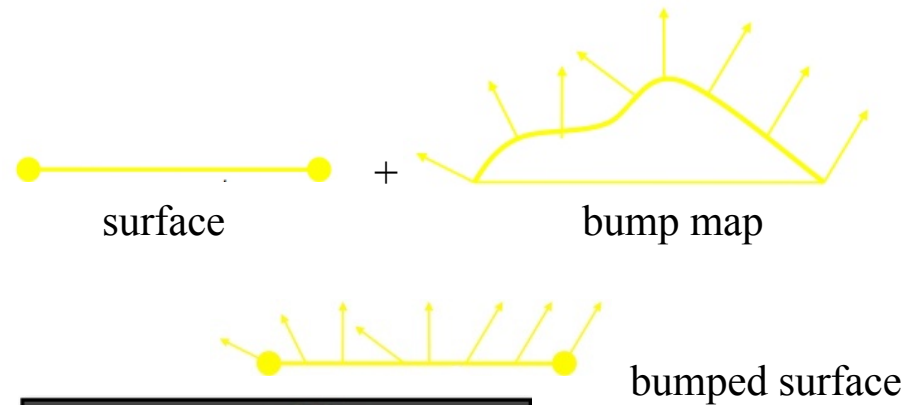


$$u(\beta, \gamma) = u_0 + \beta(u_1 - u_0) + \gamma(u_2 - u_0)$$

$$v(\beta, \gamma) = v_0 + \beta(v_1 - v_0) + \gamma(v_2 - v_0)$$

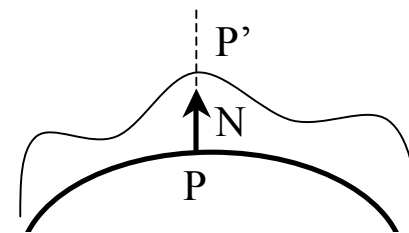
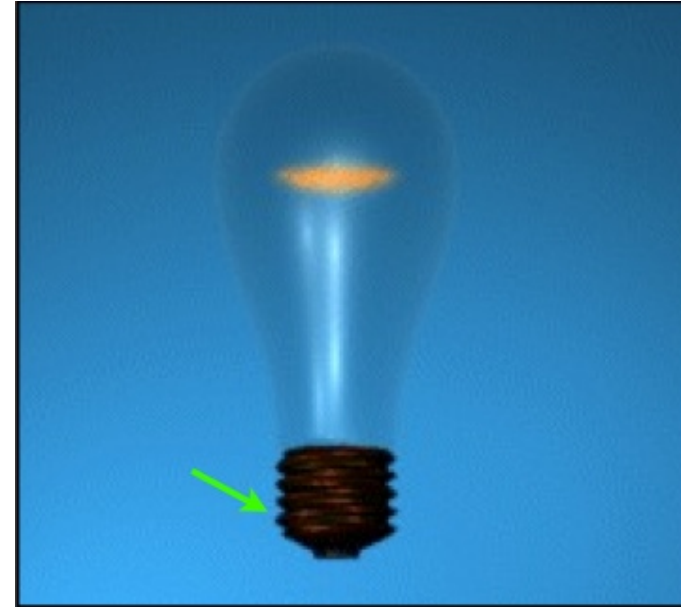
Bump maps

- Textures help with the color of the pixels to be drawn
- However, the resulting objects still look flat
- To improve this, one can store in a texture (bump map) normal variations, and use it for lighting computations while rendering
- This achieves a bumpy surface, because the varying normals change the shading computations
- However, when bump mapped polygons are seen from a flat angle they show their flatness



Displacement maps

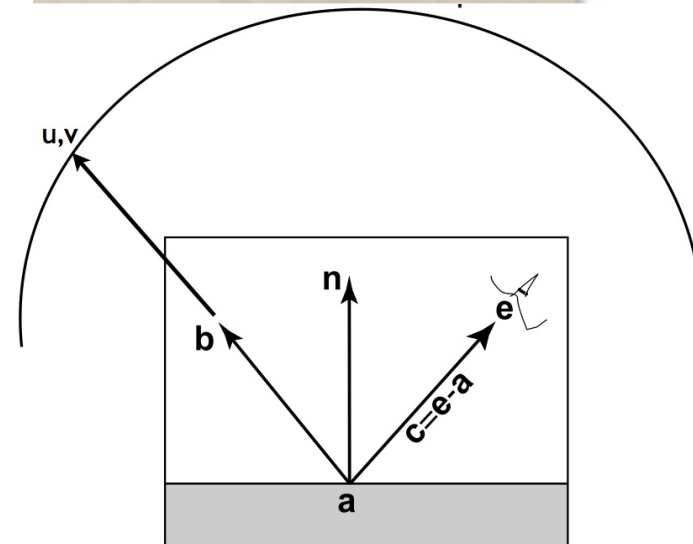
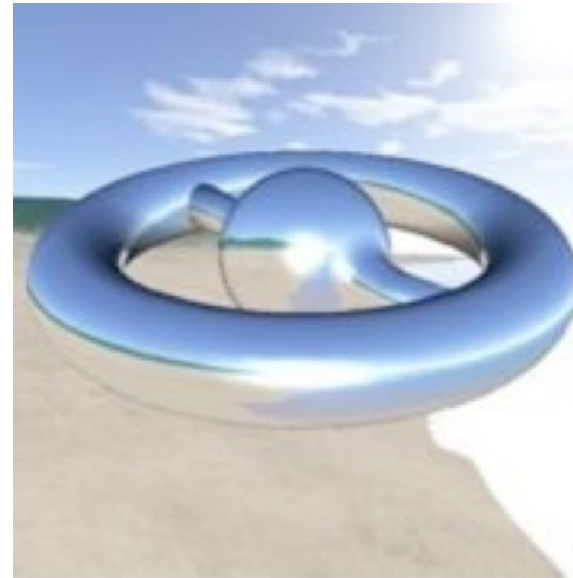
- Bump maps do not modify geometry height, which does not look good from the profile
- A way to correct this is to interpret an additional black and white texture as displacement offsets along the normal
- This is called a displacement map
- Since the displacement map “modifies” the surface to add detail to it, usual lighting computations can be done in the result



Surface + displacement

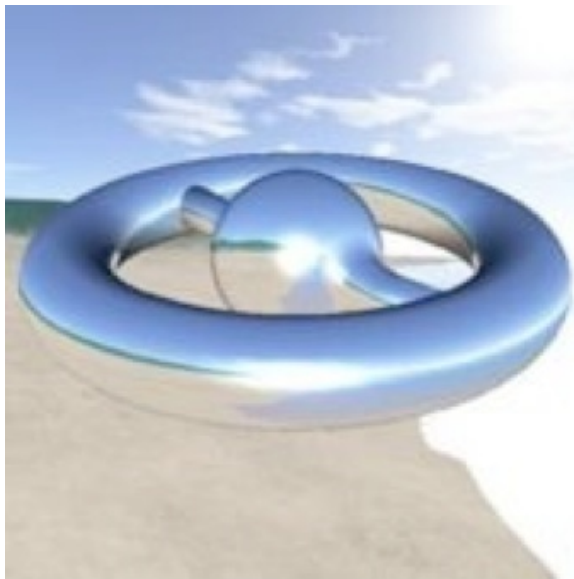
Environment maps

- There are many ways to use textures to obtain special effects in a picture
- *Environment maps* are used to simulate reflections on objects
- In this case, the world is surrounded by a closed surface having a texture
- The colour at the pixel to be rendered is looked up on the texture according to the reflection ray

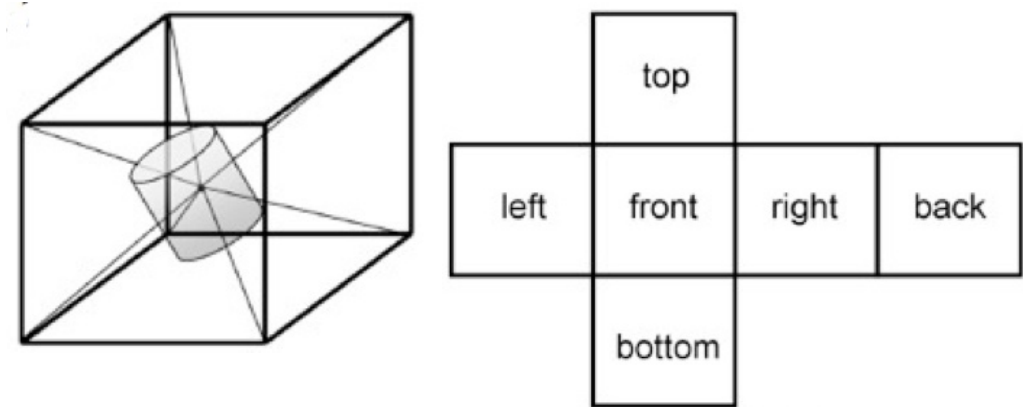


Environment maps

- There are two different ways of surrounding the world with a surface
- With a sphere: *spherical maps*

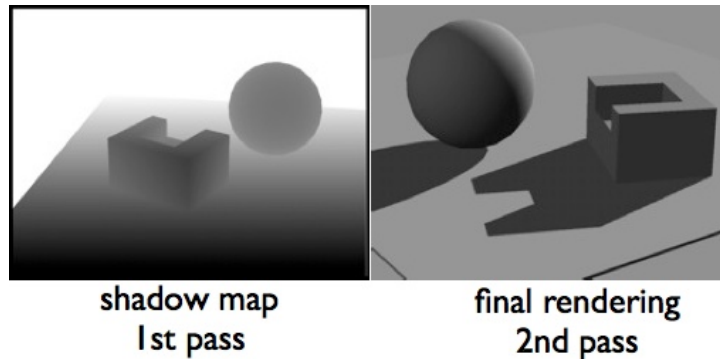
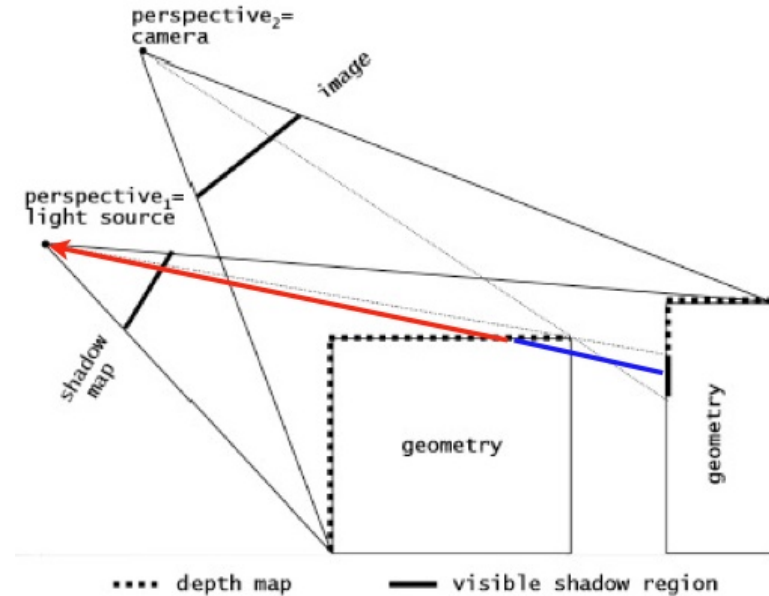


- With a cube: *cube maps*



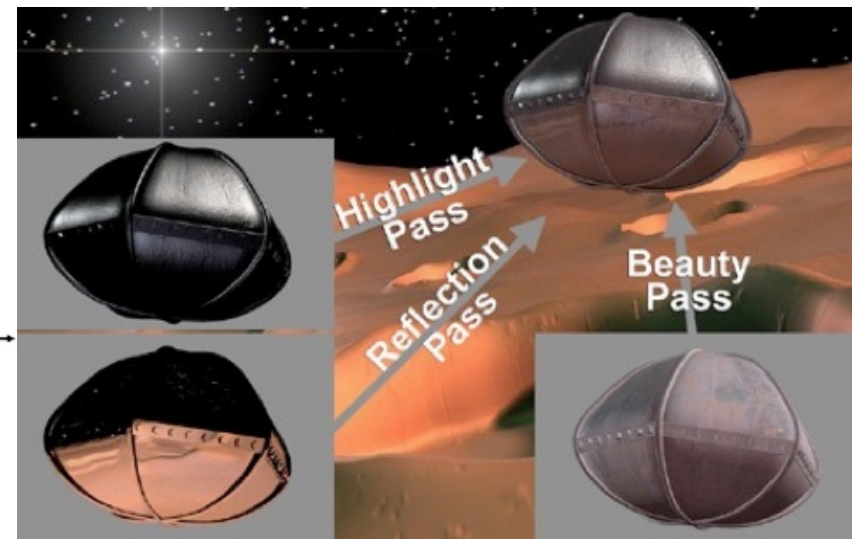
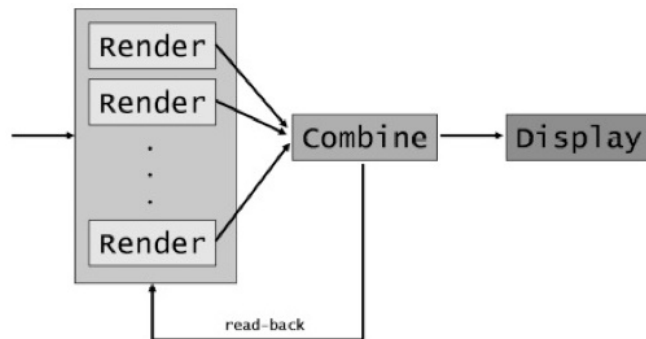
Shadow maps

- Through textures I can also do shadows
- In this case, you set perspective₁ at the light source, and image₁ contains z-buffer values of the scene from the light source
- When rendering the scene, at each pixel one is rendering one looks if its distance from the light source is smaller or bigger than the z-buffer from the light source
- In case it is bigger, then this point is in the shadow of something else

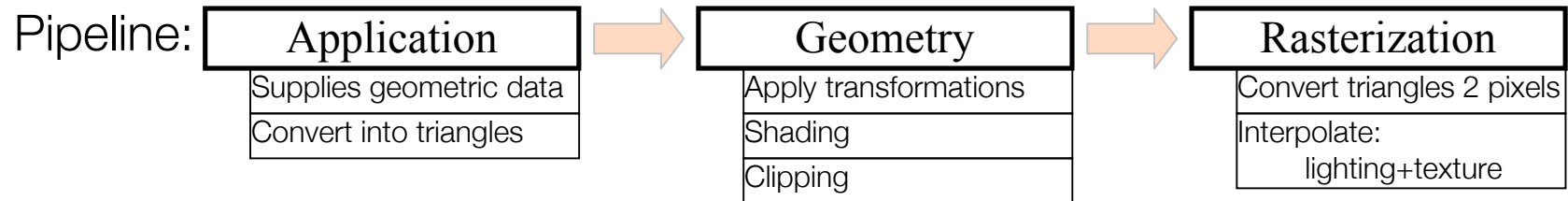


Multi-pass rendering

- To achieve more complex rendering effects, different texture rendering passes are rendered to a texture and not displayed
- This allows the layering of different effects, by blending the results of different rendering passes
- This is called multi-pass rendering



Graphics Pipeline: Wrap up



- What we can do:
 - Represent a fine variety of objects with detailed and rich surfaces
 - Represent convincing illumination of objects
 - “fake” reflections and shadows
- What we cannot do well:
 - Reflection
 - Refraction
 - Surface colour bleeding
- What we have presented here, is state of the art gaming technology

End

+++ Ende - The end - Finis - Fin - Fine +++ Ende - The end - Finis - Fin - Fine +++