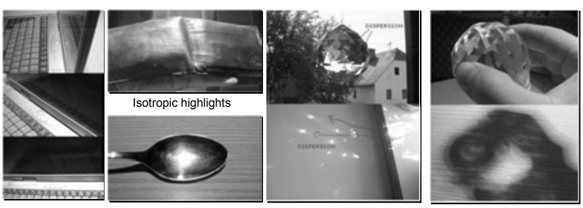
Shading

Xavier Décoret

Topic of the day

- The shading is the aspect of objects
  - what influences this aspect
  - how to model it?
  - how to compute it fastly?
- How this is (was) done on computers?
  - choosed shading model and implications
  - texture mapping
- What is the new trend?
  - programmable hardware
  - cool new effects

Different lighting effects



Isotropic highlights

Fresnel effect   Anisotropic highlights   Dispersion   Caustics

images from `http://www.maxim-capra.com`

Global illumination

- Shading of a point accounts for light interaction between objects in the scene
  - nice & realistic
    - shadows
    - inter and intra reflections
    - color bleeding
  - complex and costly
- Typical use with ray-tracing
  - photon mapping
  - radiosity

Local illumination

- Shading of a point depends only of light, observer's position and object material properties
  - lacks most visual effects
  - simpler and faster to evaluate
- Can be done with 3D APIs and hardware
- Good tricks to "emulate" missing effects

Focus of this talk

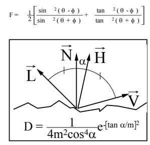
Local illumination models

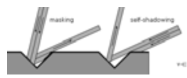
- How incoming light is reflected?
  - BRDF (Bidirectional Reflectance Distribution Function)
- Complex models
  - Cook-Torrance, Torrance-sparrow...
  - Ward
  - Kubelka-Munk, Hanrahan-Krueger, Jensen
    - Adds subsurface scatering
- Simple models

Focus of this talk

Cook-Torrance

- Accounts for surface roughness
  - physically based
  - surface is a distribution of micro-facets
- Product of 3 terms
  - Fresnel coefficient F
    - θ : incident angle
    - Φ=asin(θ/n)
    - n : refraction indice
  - Angular distribution D
    - probability of orientation for a facet
  - Masking & self-shadowing G



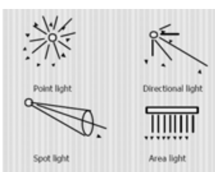


Ward

- Physically based
  - Measurement on real material
    - Gonio-réflectometer
  - Data sets
- Approximation by gaussians
- Anisotropic model

Simple shading models

- Materials/lights described by 3 components
  - an ambiant color
  - a diffuse color
  - a specular color
- Basic light sources



Light intensity can be independent or dependent of the distance between object and the light source`

Ambiant term (1/2)

- A light ambiant color I_a
  - represents light "in the scene" i.e. the "ambiance"
    - light coming from sky dome
      - ≠ of sun light which is directional (cast shadows)
    - light reflected by the scene onto itself
      - cheap emulation of global illumination
- A material ambiant coefficient K_a
  - represents the absorption of the ambiant lighting

ambient term $\boxed{A=K_a I_a}$

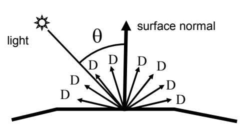
Ambient term (2/2)

- Very poor but useful
  - No physical interpretation
  - No cue of shape of objects
    - looks the same seen from anywhere no matter light position

increasing K_a

Diffuse term (1/2)

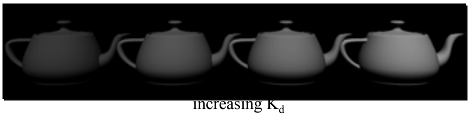
- Lambertian material
  - light reflected equally in every direction
  - reflected light depends of
    - material absorption K_d and light color I_d
    - local surface orientation



diffuse term $\boxed{D=K_d I_d \cos\theta}$

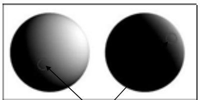
Diffuse term (2/2)

- Shading varies along surface
  - gives cue of object's shape



increasing K_d

- A point looks the same wherever you are

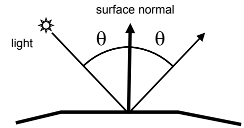


same light/object position
different observer's positions

same color

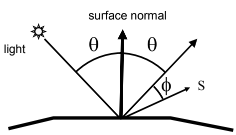
Specular term (1/3)

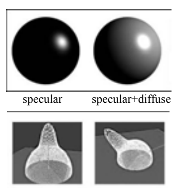
- The ideal case: mirrors
  - Snell's law (loi de Descartes)
    light is reflected with an outgoing angle equals to incoming angle
  - problem : the reflection of a point light is visible at only one point on the surface



Specular term (2/3)

- The real life: glossy objects
  - light is reflected
    - "around" the reflected vector
    - with exponential decay n (shininess)
    - material absorption K_s light color I_s



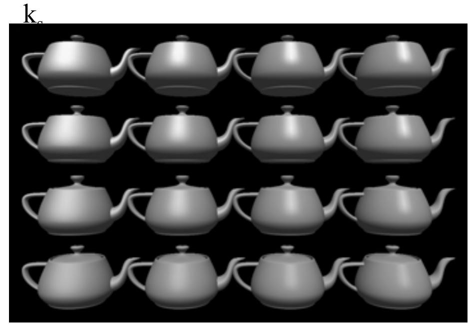


specular lobes

specular term $\boxed{S=K_s\, I_s\,(\cos\phi)^n}$

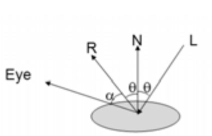
Specular term (3/3)

k_s

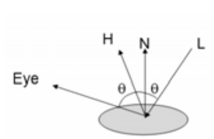


Phong & Blinn-phong models (1/2)

- The formula for specular is the Phong model
  - not physically correct [1975]
  - looks nice in practice and very simple to evaluate
- Blinn proposed a simplification
  - use angle with half-vector
  - also standard in Computer Graphics



Phong

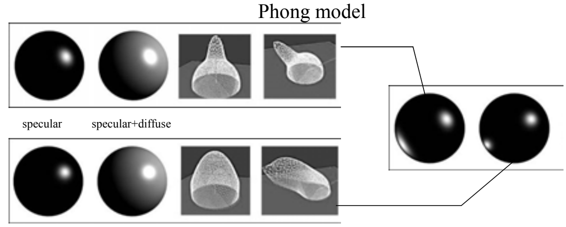


Blinn-Phong

Phong & Blinn-phong models (2/2)

- Difference is a matter of taste!
- Blinn-phong tends to be more predictable

Phong model



Blinn-Phong model

Adding all terms

- We get the color of a pixel as

$$I = \sum_{\text{lights}} \left(K_a I_a + K_d I_d \cos\theta + K_s I_s (\cos\phi)^n \right)$$

or

$$K_s I_s (H.N)^n$$

- Model used by 3D APIs (OpenGL,DirectX)
  - Hardware support

OpenGL shading (1/2)

- How pixels are produced?
  - CPU : API calls to
    - specify light attributes
    - specify vertices & attributes
      - 3D position
      - normal
      - K_a, K_d,Ks,n

$\left.\begin{array}{l}\text{normal}\\ K_a, K_d, Ks, n\end{array}\right\}$ per vertex or per face

An example

```
const float Lw[4] = { 0.0f,0.0f,8.0f,1.0f };
glLightfv(GL_LIGHT0,GL_POSITION,Lw);

const float red[3]   = { 1.0f,0.0f,0.0f };
const float blue[3]   = { 0.0f,0.0f,0.5f };
const float green[3]  = { 0.0f,1.0f,0.0f };
const float yellow[3] = { 1.0f,1.0f,0.0f };
const float black[3]  = { 0.0f,0.0f,0.0f };
glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT,blue);
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,yellow);
glMaterialfv(GL_FRONT_AND_BACK,GL_EMISSION,black);
glMaterialf(GL_FRONT_AND_BACK,GL_SHININESS,20.0f);

glBegin(GL_TRIANGLES);
// A normal per face
glNormal3f(0,0,1);
glVertex2f(0,0);
glVertex2f(2,0);
glVertex2f(1,1);

// A normal per vertex
glNormal3fv(-1, 0,1);glVertex2f(0, 0);
glNormal3fv( 1, 0,1);glVertex2f(2, 0);
glNormal3fv( 0,-1,1);glVertex2f(1,-1);
glEnd();
```


OpenGL shading (1/2)

- How pixels are produced?
  - CPU : API calls to
    - specify light attributes
    - specify vertices & attributes
      - 3D position
      - normal
      - K_a, K_d,Ks,n

$\left.\begin{array}{l}\text{normal}\\ K_a, K_d, Ks, n\end{array}\right\}$ per vertex or per face

  - GPU : dedicated hardware to
    - project vertices according to camera
    - rasterize interior pixels and compute color
    - blend fragment with pixel
  - possibility for multi-pass (accumulation buffer)

OpenGL shading (2/2)

- How pixels are shaded?
  - Flat shading
    - Apply Phong model to get a color per face
  - Gouraud shading
    - Apply Phong model at vertices to get color
    - Interpolate color across pixels

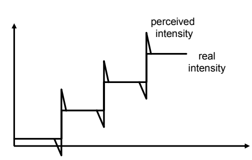
standard hardware

  - Phong shading
    - Interpolate model parameters
      - normal
      - light vector
    - Apply Phong model at each pixel

Flat *vs.* Gouraud shading

- Flat shading creates “facetted” objects
  - requires highly tesselated surfaces

- Flat shading creates Mach bands



human eye perceives intensity’s changes

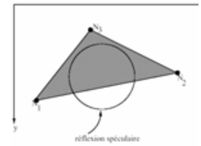
Show html example

Gouraud *vs.* Phong shading

- Means per-pixel *vs.* per vertex shading
- Per pixel is much nicer
  - And more "correct"

Gouraud *vs.* Phong shading

- Phong shading is slower
  - there are usually more pixels than vertices!
- Phong shading is nicer
  - renders highlights inside faces
  - but is not yet exact
    - light vector interpolation is only approximate
  - technical details
    - interpolated vectors must be renormalized
    - transforming normals is tricky



Normal transformation

- A plane is characterized by

$$\underbrace{[a,b,c,d]}_{n}.[x,y,z,1]^T = 0$$

- What is the characterization of the image of the plane by an affine transformation M?
  - We search n' such that

$$n'.(M[x,y,z,1])^T = 0$$

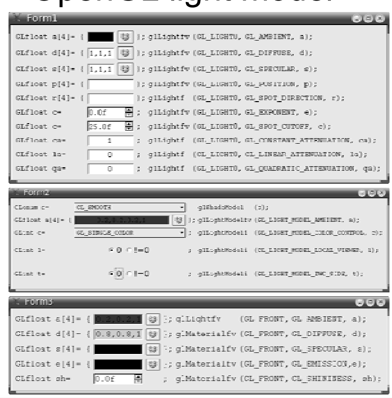
  - A solution is

$$n' = (M^T)^{-1} n$$

- Normals transformed by <u>inverse transpose</u>

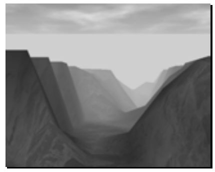
OpenGL light model

Lots of parameters!



Tricks for complex effects

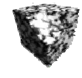
- We have seen ambient term
- Atmospheric Effects (fog)
  - blend with background color
  - based on
    - distance to eye
    - chosen attenuation model



- Texture mapping

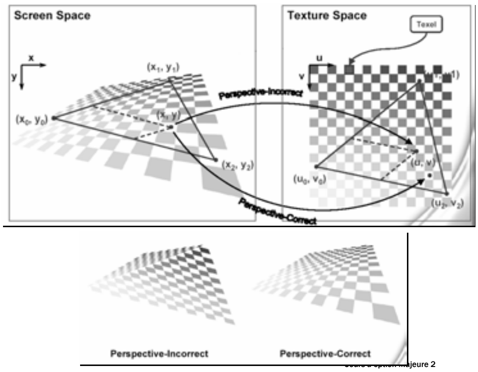
Texture mapping

- Ability to look up a value for each pixel
  - ambient/diffuse color
  - normal (bump mapping)
  - more to come...



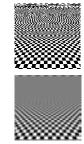
- Lookups specified with texture coordinates
  - specified for each vertex `glTexCoord{123}{fi}`
  - interpolated for each fragment
    - perspective correct interpolation

Perspective correct interpolation



Texture mapping and aliasing

- Undersampling
  - take nearest or interpolate neighbours
- Supersampling
  - ideal solution
    - integrate over samples → expensive
  - practical solution
    - prefilter textures → mipmaps
    - interpolate between levels (and neighbours)

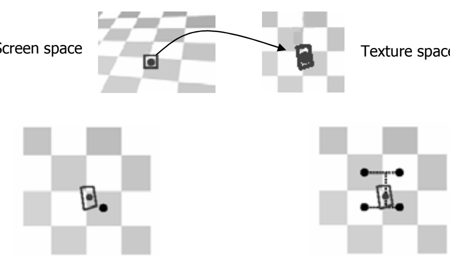


http://en.wikipedia.org/wiki/Anti-aliasing

Undersampling

- Pixel "smaller" than texel

Screen space



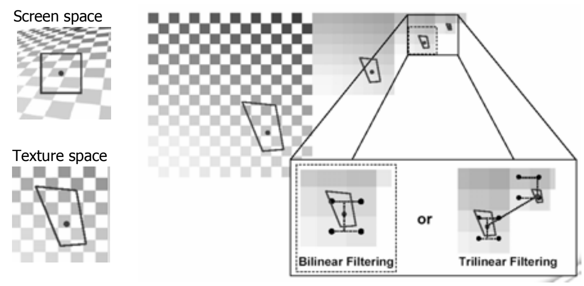
Texture space

Color of nearest texel
`GL_NEAREST`

Bi-linear filtering
`GL_LINEAR`

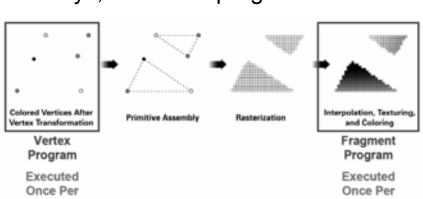
Supersampling

- Pixel "larger" than texel



Programmable hardware

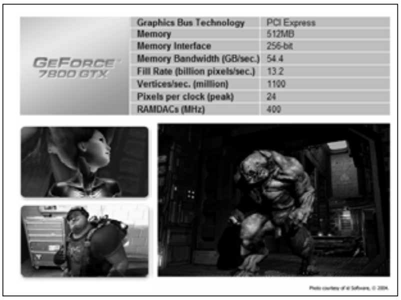
- What we've just seen is outdated!
  - It was "ol' times" way of doing
    - everything hardwired
    - fixed pipeline
- Nowadays, cards are programmable!



The need for programmability

The need for programmability

- Nowadays…

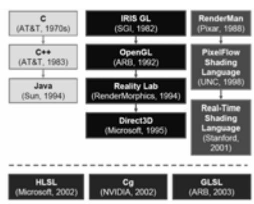


The need for programmability

- Available power raises expectations
  - complex geometry <u>and</u> appearance
  - movie like quality <u>in real-time</u>
- Hardware is actually programmable
  - to some extents (rapidly changing)
  - just need to expose it
- Realistic shading is complex
  - creation must be human-friendly
  - shading must be modular = reusable

What language?

- Low level
  - like assembly but a bit simpler
  - historic approach
  - standardized as OpenGL extensions
- High level
  - C/C++ like syntax
  - well known for movies
  - today's trend for real-time
  - different languages



Low *vs.* high level (1/2)

- Low level
  - match hardware closely
    - ⊙ easier to understand what's done
    - ⊙ allows for tight optimizations
  - hard to program
  - not hardware independent
    - ⊙ must be rewritten/optimized for each hardware
    - ⊙ no anticipation of harware evolution

Assembly

```
...
DP3 R0, c[11].xyzx, c[11].xyzx;
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
...
```


Low *vs.* high level (2/2)

- High level
  - easy to program
  - easy to reuse
  - easy to reuse
  - compiled
    - ⊙ harware independent
    - ⊙ harder to understand bottlenecks

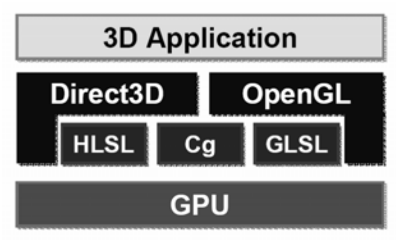
High-Level Language

```
...
float3 cSpecular = pow(max(0, dot(Nf, H)),
                       phongExp).xxx;
float3 cPlastic = Cd * (cAmbient + cDiffuse) +
                  Cs * cSpecular;
...
```


Compilation strategies

- External compiler to low level (HLSL,Cg)
  - offline or on the fly
  - "by hand" optimization of compiled code
  - profiles
- In-driver compilation (GLSL)
  - trust the driver!

Shading languages



Language features (1/3)

- Control flow statements
  - if, for, while,break, continue
  - pas de goto
- User defined functions
  - good for modularity and reusability
- Built-in functions
  - math :abs, pow, sqrt, step, smoothstep...
  - geometry :dot, cross, normalize, reflect, refract...
  - texture lookups
  - fragment functions

Language features (1/3)

- Support for vector and matrices
  - **Component-wise + - * / for vectors**
  - **Dot product**
    - `dot(v1,v2); // returns a scalar`
  - **Matrix multiplications:**
    - **assuming a** `float4x4 M` **and a** `float4 v`
    - **matrix-vector:** `mul(M, v); // returns a vector`
    - **vector-matrix:** `mul(v, M); // returns a vector`
    - **matrix-matrix:** `mul(M, N); // returns a matrix`

Language features (3/3)

- New operators
  - **Swizzle operator extracts elements from vector or matrix**

```
a = b.xxyy;
```

  - **Examples:**

```
float4 vec1 = float4(4.0, -2.0, 5.0, 3.0);
float2 vec2 = vec1.yx;      // vec2 = (-2.0,4.0)
float scalar = vec1.w;      // scalar = 3.0
float3 vec3 = scalar.xxx;   // vec3 = (3.0, 3.0, 3.0)
float4x4 myMatrix;

// Set myFloatScalar to myMatrix[3][2]
float myFloatScalar = myMatrix._m32;
```

  - **Vector constructor builds vector**

```
a = float4(1.0, 0.0, 0.0, 1.0);
```

NVIDIA.

Limited programmability

- Limited number of instructions
- Limited number of variables
- Some restrictions on loops/branching
  - cost of else/then branches
  - no dependent loops
    - loops are unrolled → limits

  changes quickly!
  no longer true for cutting edge cards
- Some specific limitations
  - no texture lookup in vertex programs
  - no dependent texture lookup
  - some undocumented : driver bug or limitation ?!?

Overview of using shaders

- Use API calls to
  - specify vertex & fragment shaders
  - enable vertex & fragment shaders
  - pass “global” parameters to shaders
- Draw geometry as usual
  - vertex shader will execute for each vertex
  - fragment shader will execute for each fragment

GLSL : toon shaders

file `toon.vert`

```
varying vec3 normal;

void main() {
   normal = gl_NormalMatrix * gl_Normal;
   gl_Position = ftransform();
}
```

file `toon.frag`

```
varying vec3 normal;
uniform vec3 t;

void main()  {
   vec4 color;
   vec3 n = normalize(normal);
   float i = dot(vec3(gl_LightSource[0].position),n);
   if (i>treshold[0]) color = vec4(1.0,0.5,0.5,1.0);
   else if (i>threshold[1]) color = vec4(0.6,0.3,0.3,1.0);
   else if (i>threshold[2]) color = vec4(0.4,0.2,0.2,1.0);
   else color = vec4(0.2,0.1,0.1,1.0);

   gl_FragColor = color;
}
```


GLSL: setting up shaders

```
GLhandleARB v = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
GLhandleARB f = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
char* vs = vs = textFileRead("toon.vert");
char* fs = textFileRead("toon.frag");
const char* vv = vs;
const char* ff = fs;
glShaderSourceARB(v, 1, &vv,NULL);
glShaderSourceARB(f, 1, &ff,NULL);
free(vs);
free(fs);
glCompileShaderARB(v);
glCompileShaderARB(f);

GLhandleARB p = glCreateProgramObjectARB();
glAttachObjectARB(p,v);
glAttachObjectARB(p,f);
glLinkProgramARB(p);

glUseProgramObjectARB(p);
```


GLSL : using shaders

```
// once for all (in QGLViewer::init())
GLint thresholdParam = glGetUniformLocationARB(p,"threshold");

// at every frame (in QGLViewer::draw())
glUseProgramObjectARB(p);
glUniform3fARB(thresholdParam,0.95f,0.5f,0.25f);
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
glBegin(GL_TRIANGLES);
// draw teapot
glNormal3fv(...);glVertex3v(...);
...
glEnd();
glUseProgramObjectARB(0);
```


On parameters passing

- From CPU to shaders
  - per vertex attributes
    - use standard OpenGL attributes
    - forget about them as position/normal/colors/texcoords just think of them as general attributes
  - uniform parameters
    - need a handle on them
    - specified per primitive i.e `glBegin()/glEnd()`
  - textures
    - think of them as general lookup tables
- From vertex shader to fragment shader
  - varying parameters gets interpolated

References

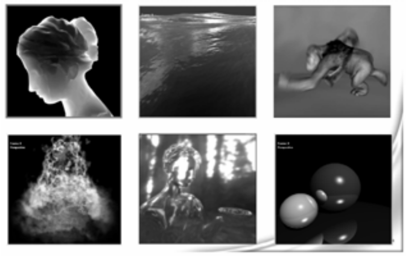
- More about GLSL
  - Official site
    `http://developer.3dlabs.com/openGL2/index.htm`
  - Tutorial
    `http://www.lighthouse3d.com/opengl/glsl/index.php?intro`
  - Complete reference
    `http://developer.3dlabs.com/openGL2/index.htm`
- More about Cg
  `http://developer.nvidia.com/page/cg_main.html`

GPU is horse power

- SIMD processor
- Parallel execution
  - multiple vertex/pixels units
- Hardwired instructions
  - trigonometric, vector manipulation...
- Very fast
  - performances increase everyday

Many applications

- In graphics
  - many beautiful shaders



Many applications

- In graphics
  - many beautiful shaders
  - see *GPU Gems I & II*
  - ray tracing on GPU!

http://developer.nvidia.com/object/gpu_gems_2_home.html

- In other domains
  - general purpose GPU based computations
    - linear algebra
    - scientific simulation
    - ...
  - questionable?

Various remarks

- Interaction with fixed pipeline not always clear
- No access to frame buffer
  - difficult because of parallelism
  - but people would like to have it!
- Precision issues (fixed,float,half)
- Performance issues
- Much much more to say!!!

What's next?

- Make it faster and faster
  - and even faster!
- Loosen current limitations
  - not always possible
- Open other parts of the pipeline
  - programmable interpolation?
  - programmable z-test?
- Add new components on chip
  - new buffers?
  - new hardwired functionalities?

Deferred Shading

- A pixel shader may be very complex
- It is evaluated at each fragment
- Fragment may then fail z-test → waste
  - shaders compute color/depth
  - fragment is tested *after* fragment shader
- Use G-buffer instead
  - render shading attributes (not too much)
  - evaluate shading in a last pass

Conclusion

- visit `http://developer.nvidia.com`