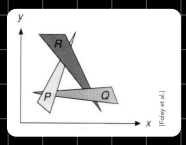
Visibility / Occlusion <Z-buffering>

Painter's Algorithm — paint from back to front, overwrite in the framebuffer

- requires sorting in depth <O(n log n) for n triangles>
- could have unresolvable depth order

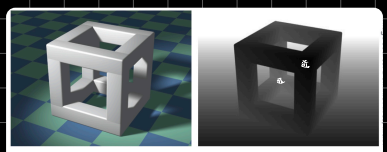


Z-Buffer

This is the algorithm that eventually won.

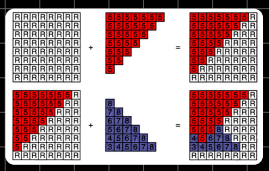
Idea:
- Store current min. z-value for each sample (pixel)
- Needs an additional buffer for depth values
  - frame buffer stores color values
  - depth buffer (z-buffer) stores depth

IMPORTANT: For simplicity we suppose z is always positive (smaller z -> closer, larger z -> further)




```
Initialize depth buffer to ∞
During rasterization:
    for (each triangle T)
        for (each sample (x,y,z) in T)
            if (z < zbuffer[x,y])          // closest sample so far
                framebuffer[x,y] = rgb;    // update color
                zbuffer[x,y] = z;          // update depth
            else
                ;                          // do nothing, this sample is occluded
```

在每一个像素内记录最浅深度



- O(n) for n triangles <On assuming constant coverage> 并未进行排序
- Most important visibility algorithm — implemented in hardware for all GPUs
- 深度完全相等<float>几乎不会出现，在此不作讨论

Illumination, Shading & Graphic Pipeline

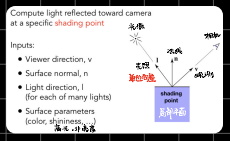
回顾

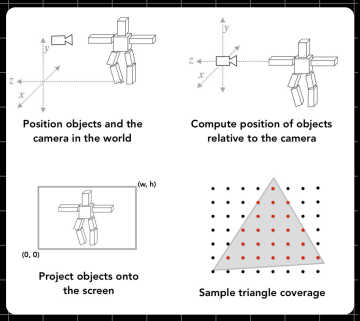
- In Merriam-Webster Dictionary

  shad-ing, ['ʃeɪdɪŋ], noun
  The darkening or coloring of an illustration or diagram with parallel lines or a block of color.

- In this course

  The process of applying a material to an object.



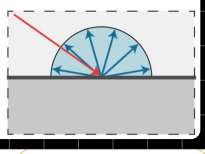


▲ Shading is local. NO SHADOW is generated.

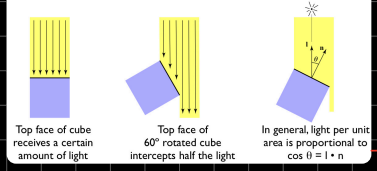
Diffuse Reflection

How much light is received?

- Light is scattered uniformly in all directions
  - Surface color is the same for all viewing directions

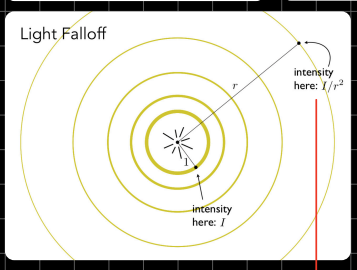


- Lambert's cosine law

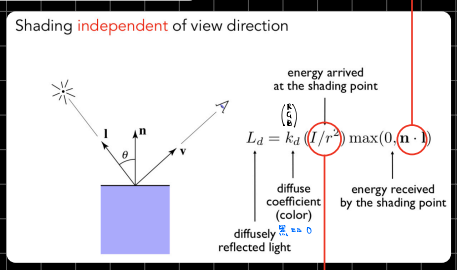


$\vec{a} \cdot \vec{b} = \|\vec{a}\| \cdot \|\vec{b}\| \cdot \cos\theta$

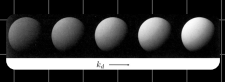
Diffusely Reflected Light Formula



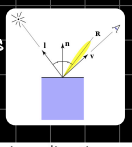
球体面积 $4\pi r^2$



漫反射成像与观测角度无关



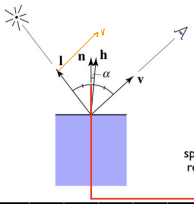
Specular Highlights



Blinn-Phong

V close to mirror direction ⇔ **half vector** near normal

- Measure "near" by dot product of unit vectors



$$\mathbf{h} = \text{bisector}(\mathbf{v}, \mathbf{l}) \quad \text{(半程向量)}$$
$$= \frac{\mathbf{v}+\mathbf{l}}{\|\mathbf{v}+\mathbf{l}\|}$$

方便计算

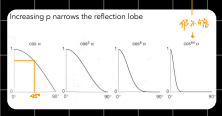
$$L_s = k_s\,(I/r^2)\max(0, \cos\alpha)^p$$
$$= k_s\,(I/r^2)\max(0, \mathbf{n}\cdot\mathbf{h})^p$$

specularly reflected light — specular coefficient

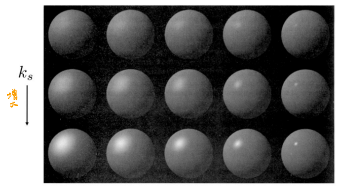
usually white

100 5 200

使用 cos 函数形"容忍度"过高，即像相差达到 45° 仍未明显差值



Blinn-Phong $L_s = k_s\,(I/r^2)\max(0, \mathbf{n}\cdot\mathbf{h})^p$



Note: showing Ld + Ls together

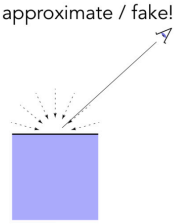
$p \longrightarrow$

法线方向与半程向量相近

Ambient lighting 环境光 与光源，观测角度无关

Shading that does not depend on anything

- Add constant color to account for disregarded illumination and fill in black shadows
- This is approximate / fake!

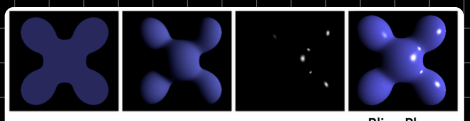


$$L_a = k_a\,I_a$$

ambient coefficient

reflected ambient light

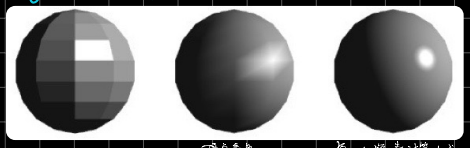
Blinn-Phong Reflection Model



Ambient + Diffuse + Specular = Blinn-Phong Reflection

$$L = L_a + L_d + L_s$$
$$= k_a\,I_a + k_d\,(I/r^2)\max(0, \mathbf{n}\cdot\mathbf{l}) + k_s\,(I/r^2)\max(0, \mathbf{n}\cdot\mathbf{h})^p$$

Shading Frequency 着色频率



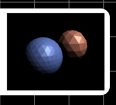
顶点着色 差值过渡

每一个像素计算法线再进行上色

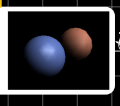
Flat Shading | Gouraud Shading | Phong Shading

高色逼真色

Flat shading
- Triangle face is flat — one normal vector
- Not good for smooth surfaces



Gouraud shading
- Interpolate colors from vertices across triangle
- Each vertex has a normal vector (how?)

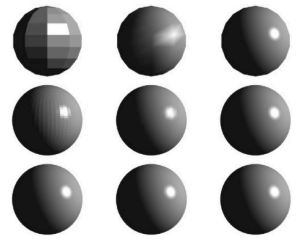


如何得

求点法线

Phong shading
- Interpolate normal vectors across each triangle
- Compute full shading model at each pixel
- Not the **Blinn-Phong Reflectance Model**

Shading Frequency: Face, Vertex or Pixel



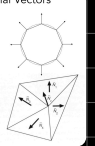
	Face	Vertex	Pixel
Shading freq. :			
Shading type :	Flat	Gouraud	Phong

Defining Per-Vertex Normal Vectors

Best to get vertex normals from the underlying geometry
- e.g. consider a sphere

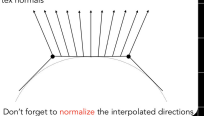
Otherwise have to infer vertex normals from triangle faces
- Simple scheme: **average surrounding face normals**

$$N_v = \frac{\sum_i N_i}{\|\sum_i N_i\|}$$



Defining Per-Pixel Normal Vectors

Barycentric interpolation (introducing soon) of vertex normals

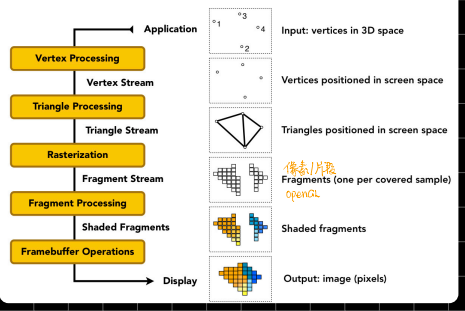


Don't forget to normalize the interpolated directions

相邻三角形，加面积加权法线很平滑

Graphics < Real-time Rendering > Pipeline

Graphics Pipeline



光栅化

Shader Programs

practice - shader toy

- Program vertex and fragment processing stages
- Describe operation on a single vertex (or fragment)

Example GLSL fragment shader program

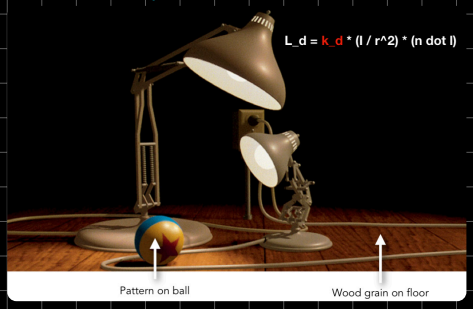
```
uniform sampler2D myTexture;
uniform vec3 lightDir;
varying vec2 uv;
varying vec3 norm;

void diffuseShader()
{
  vec3 kd;
  kd = texture2d(myTexture, uv);
  kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);
  gl_FragColor = vec4(kd, 1.0);
}
```

- Shader function executes once per fragment.
- Outputs color of surface at the current fragment's screen sample position.
- This shader performs a texture lookup to obtain the surface's material color at this point, then performs a diffuse lighting calculation.

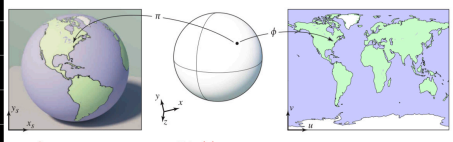
Texture Mapping

How to define k_d for every pixel



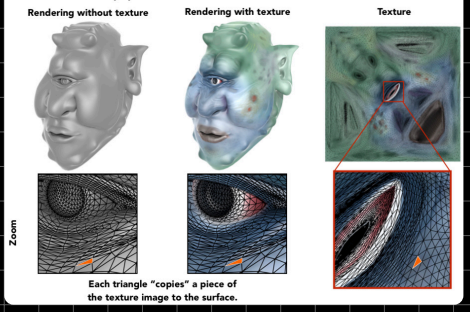
Surface lives in 3D world space

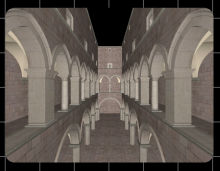
Every 3D surface point also has a place where it goes in the 2D image (**texture**).

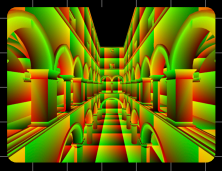


三维物体的表面是二维的

Texture Applied to Surface





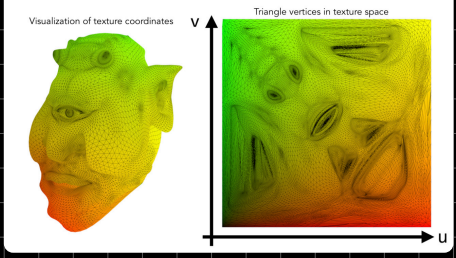


texture coordinates

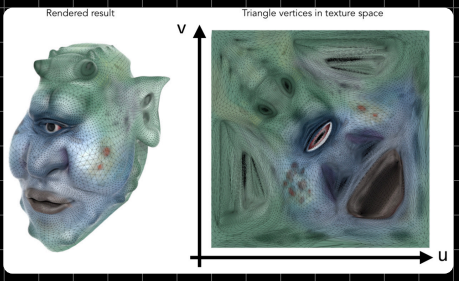
tileable texture

Visualization of Texture Coordinates

Each triangle vertex is assigned a texture coordinate (u,v)

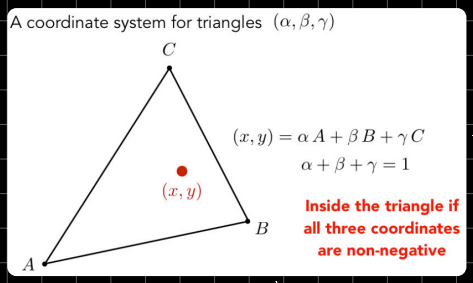


0~1 always regardless original aspect ratio



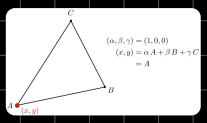
Barycentric Coordinates 重心坐标 - 计算三角形内部的插值 <像素的法线>

Interpolation across triangle - 一个三角形对应一个重心坐标

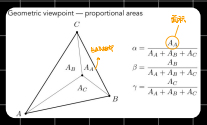


通过三个顶点之线性组合获得到点(x,y)

$\gamma = 1 - \alpha - \beta$

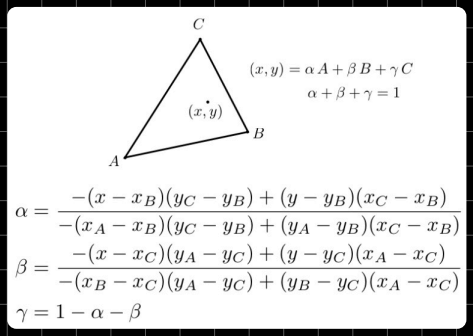


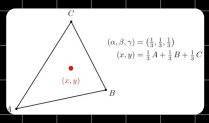
例：1点三重心坐标



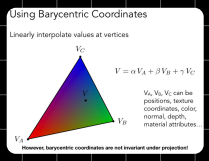
通过面积求重心坐标

公 式





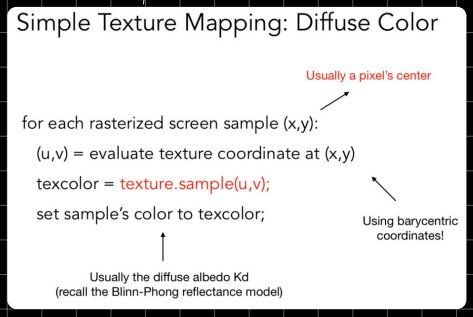
重心 <centroid> 为重心坐标

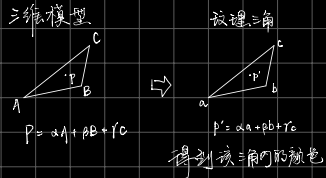


使用重心坐标计算属性

法线 颜色 纹理 位置 etc

重心坐标，在投影 <变形> 后会改变，因此要在三维空间内先算好重心坐标，再投影成二维

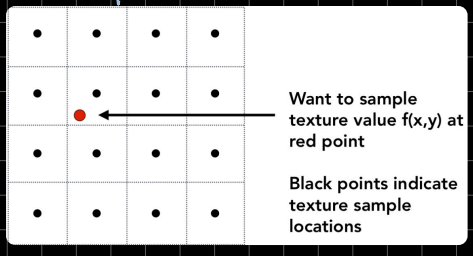


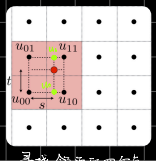


Texture Magnification 纹理放大

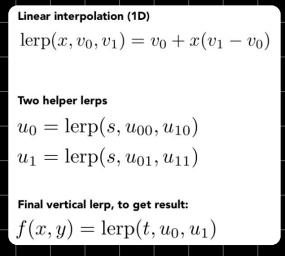
A pixel on texture - texel 纹理像素/纹素

Bilinear Interpolation 双线性插值





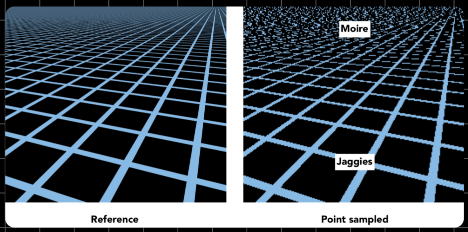
寻找邻近的四个点，并量距离左下角点的水平竖直距离

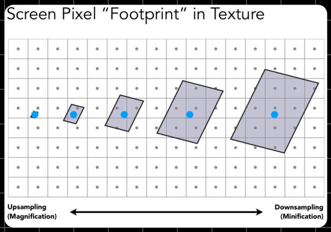


采样点非整数

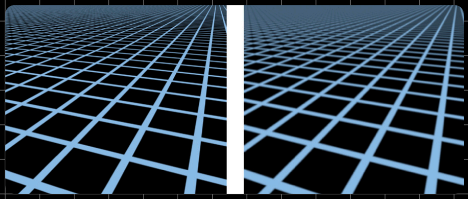
Bicubic Interpolation 双三次插值 - 取周围16格计算

纹理过大



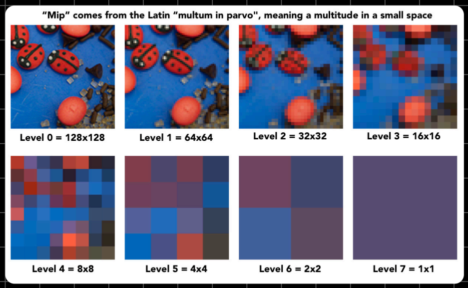


单个像素覆盖的纹理区域不断变大



使用512倍超采样可解决但太贵

Mipmap allowing < fast, approx. square > range queries 只能做近似.正方形



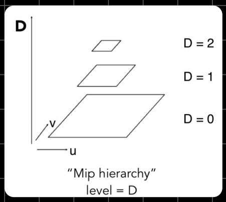
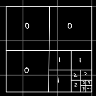


image pyramid

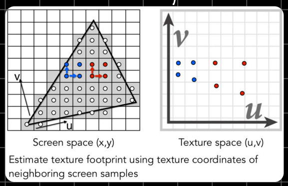
相比单一uv图,新增后的图总量 $\frac{4}{3}$

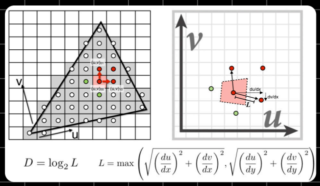


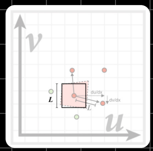
×3后理解

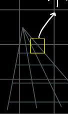
多出1/3

Computing Mipmap Level D







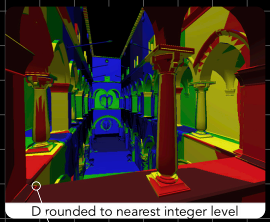


屏幕像素点(包含了过多细节)在u.v图上查询到的是大片的细节,通过测量u.v图中的边长,返回Mipmap中对应近似纹理以避免失真.

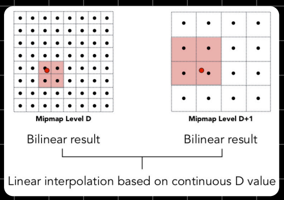
通过映射邻近点到u.v图来近似出所需查询的区域大小

Mipmap Limitation

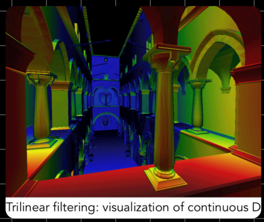
overblur

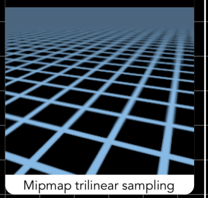

D rounded to nearest integer level

但是不连续
会有断裂感

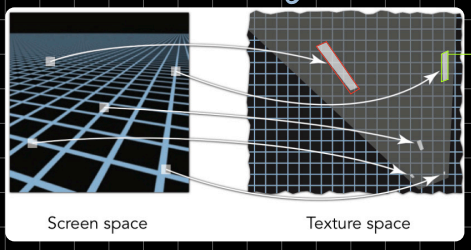


Trilinear Interpolation


Trilinear filtering: visualization of continuous D


Mipmap trilinear sampling

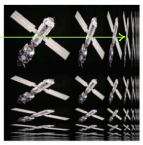
Anisotropic Filtering 各项异性过滤



Irregular footprint in Texture

Anisotropic Filtering

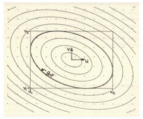
Ripmaps and summed area tables
- Can look up axis-aligned rectangular zones
- Diagonal footprints still a problem



Wikipedia

Ripmap 空间中轴对齐的预先计算 <显存>

EWA filtering
- Use multiple lookups
- Weighted average
- Mipmap hierarchy still helps
- Can handle irregular footprints



Greene & Heckbert '86

消耗计算

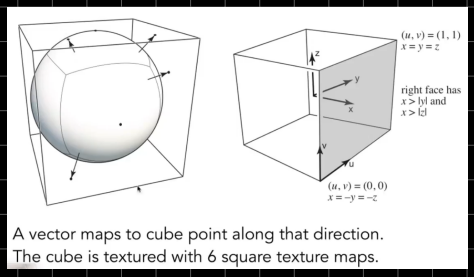
Texture = Memory + Range Query < filtering >

Environment Map 记录各方向的光照并形成纹理

环境光照/贴图

Spherical Environment Map

Cube Map 用六张图记录环境光



A vector maps to cube point along that direction.
The cube is textured with 6 square texture maps.

裁少扭曲

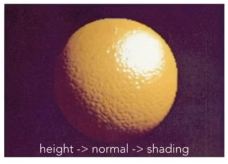
事先计算在哪个面上

凹凸贴图 法线贴图

Bump / Normal mapping

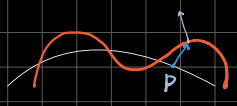
- Textures doesn't have to only represent colors
  - What if it stores the height / normal?
  - Bump / normal mapping
  - **Fake** the detailed geometry

Relative height to the underlying surface



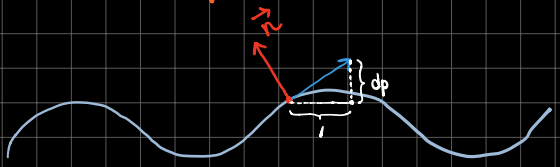
假的凹凸感觉

高低 → 法线 → 着色



增加细理
但不改变模型三角形

How to perturb the normal <flatland>



Original surface normal $n\langle p\rangle = \langle 0, 1\rangle$

Derivative at P is $dp = c * [h(p+1) - h(p)]$ (四点取向前差)

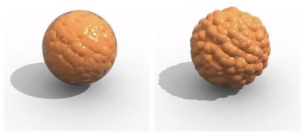
perturbed normal is then $n(p) = \langle -dp, 1\rangle$.normalized

How to perturb the normal (in 3D)

- Original surface normal n(p) = (0, 0, 1)
- Derivatives at p are
  - dp/du = c1 * [h(**u**+1) - h(**u**)]
  - dp/dv = c2 * [h(**v**+1) - h(**v**)]
- Perturbed normal is n = **(-dp/du, -dp/dv, 1)**.normalized()
- Note that this is in **local coordinate**! More will be elaborated in FAQ of HW3

位移贴图 Displacement mapping 微表面位移 顶点

- **Displacement mapping** — a more advanced approach
  - Uses the same texture as in bumping mapping
  - Actually **moves the vertices**



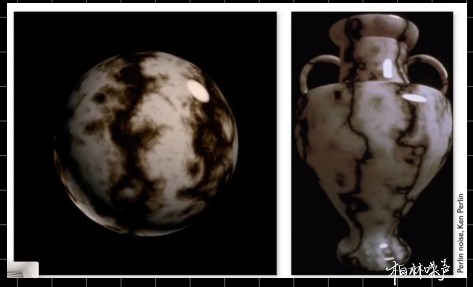
Bump / **Normal** mapping Displacement mapping

对模型精细度有更高要求

粗糙模型 → 位移贴图 → 模型细分 → 大三角变小三角 → DIRECT ✗ Dynamic tessellation 动态曲面细分

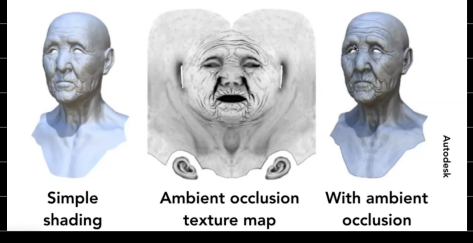
3D Procedual Noise + Solid Modeling

过程纹理



环境光遮蔽 Ambient Occlusion

Provide Precomputed Shading



Simple shading | Ambient occlusion texture map | With ambient occlusion

3D Textures and Volume Rendering

3D Textures and Volume Rendering