1. 凹凸纹理简介

凹凸纹理（Bump Mapping）是计算机图形学中，使物体表面仿真出褶皱效果的技术，该技术通过（1）扰动对象平面的法向量，（2）利用扰动的法向量进行光照计算，来实现。物体表面呈现出来的褶皱效果，不是由于物体几何结构的变换，而是光照计算的结果，凹凸纹理技术的基本思想最早由 James Blinn 在 1978 年提出的[1]。凹凸纹理的效果如图 1 所示[2]，左图是一个光滑的小球，中间图是一张扰动纹理贴图，通过它来扰动小球平面的法向量，再经过光照计算，就能产生右图所示的带有褶皱表面的小球。

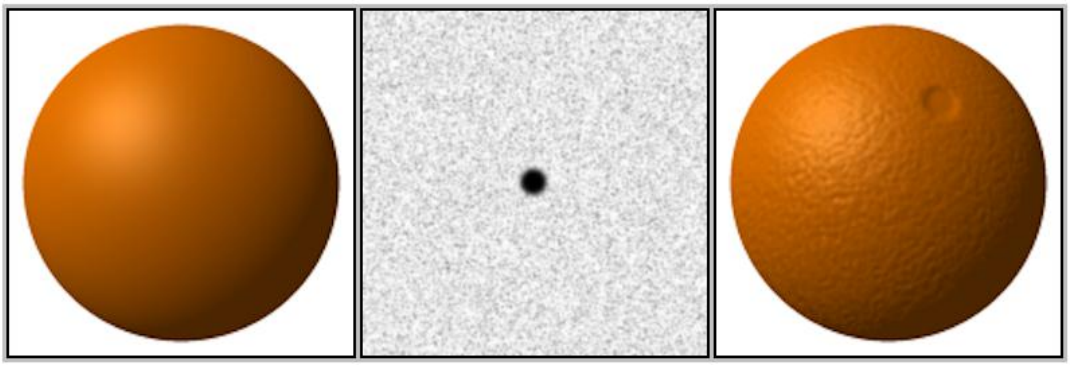


图 1. 凹凸纹理效果图

2. Blinn 技术

2.1. 数学原理

任何一个三维几何面片，可以用三个带有两个变量的参数形式的表示：

$$X = X(u,v), Y = Y(u,v), Z = Z(u,v) \qquad (1)$$

其中，u,v 的取值在区间 $[0,1]$ 之间，几何面片的局部点的偏微分表示该点的两个方向向量：

$$\overrightarrow{P_u} = (x_u, y_u, z_u), \overrightarrow{P_v} = (x_v, y_v, z_v) \qquad (2)$$

单位向量 $\overrightarrow{P_u}, \overrightarrow{P_v}$ 表示与该局部点相切的两个方向的切线，那么该点的法向量可以通过这两个切线向量的叉积得到，如图 2 所示：

$$\overrightarrow{N} = \overrightarrow{P_u} \times \overrightarrow{P_v} \qquad (3)$$

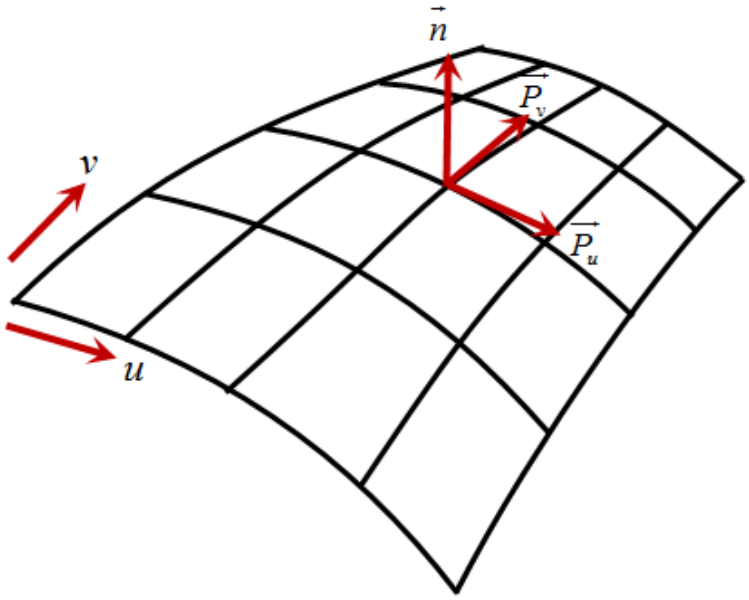
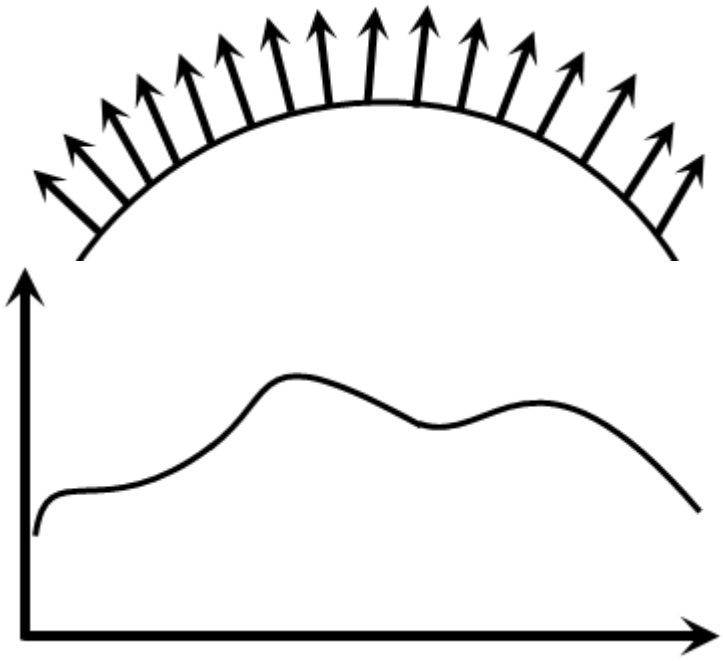


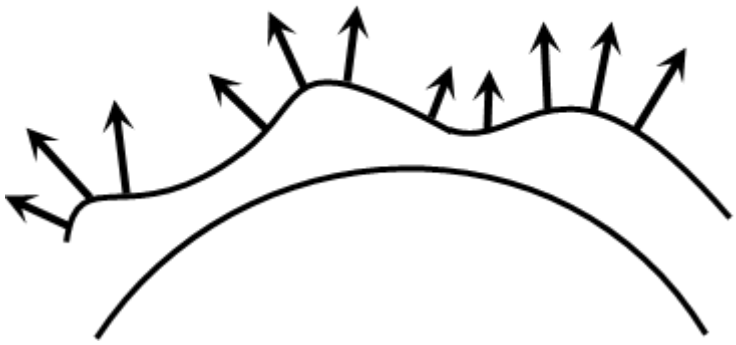
图 2. 三维几何面片的法向量

实时渲染中的光照计算，特别是漫反射光和镜面反射光（Phong 光照模型），依赖平面的法向量，Blinn 技术的原理就是，使用一个扰动函数，对平面的法向量进行扰动，再将扰动后的法向量用于光照的计算。图 3（a）~（d）描述了 Blinn 技术对法向量扰动的基本过程。



（a）

（c）



（d）

图 3. 法向量的扰动过程

设**扰动函数**表示为 $F(u,v)$，原始平面上的一个点 P，经过扰动后的位置为：

$$P' = P + F \cdot \vec{N} / \left|\vec{N}\right| \qquad (4)$$

而**扰动的法向量** $\vec{N}'$ 仍然可以由两个方向向量的偏微分得到：

$$\vec{N}' = \vec{P_u}' \times \vec{P_v}' \qquad (5)$$

计算新的位置点 P' 在 u,v 方向上的偏微分，得到：

$$\begin{cases} \vec{P_u}' = \vec{P_u} + F_u \dfrac{\vec{N}}{\left|\vec{N}\right|} + F \cdot \dfrac{d(\vec{N} / \left|\vec{N}\right|)}{du} \\ \vec{P_v}' = \vec{P_v} + F_v \dfrac{\vec{N}}{\left|\vec{N}\right|} + F \cdot \dfrac{d(\vec{N} / \left|\vec{N}\right|)}{dv} \end{cases} \qquad (6)$$

为了简化问题，可以将上式简化为：

$$\begin{cases} \vec{P_u}' = \vec{P_u} + F_u \dfrac{\vec{N}}{\left|\vec{N}\right|} \\ \vec{P_v}' = \vec{P_v} + F_v \dfrac{\vec{N}}{\left|\vec{N}\right|} \end{cases} \qquad (7)$$

把等式（7）代入等式（5），可以得到：

$$\overrightarrow{N'} = (P_u + F_u \frac{\vec{N}}{|\vec{N}|}) \times (P_v + F_v \frac{\vec{N}}{|\vec{N}|}) = \vec{N} + D \quad (8)$$

其中， $D = \frac{F_u(\vec{N} \times \overrightarrow{P_v}) - F_v(\vec{N} \times \overrightarrow{P_u})}{|\vec{N}|}$ 。

2.2. 算法描述

Blinn 技术的算法描述如下所示：

在光照计算之前，对于物体表面上的每个顶点（像素）：

1. 查询表面上的每个点 p 在高度贴图（Heightmap）上对应的点 p_r；
2. 采用有限差分法，计算点 p_r 处的法向量 $\overrightarrow{n_r}$；
3. 设物体表面上的点 p 的法向量为 $\vec{n}$，利用法向量 $\overrightarrow{n_r}$ 来扰动“真正”的法向量 $\vec{n}$，得到扰动的法向量 $\overrightarrow{n'}$；
4. 采用扰动的法向量 $\overrightarrow{n'}$，实现点 p 的光照计算。

显然，凹凸纹理的深度感，依赖光照的位置等信息，当光照发生了变化，深度感也随之发生变化。

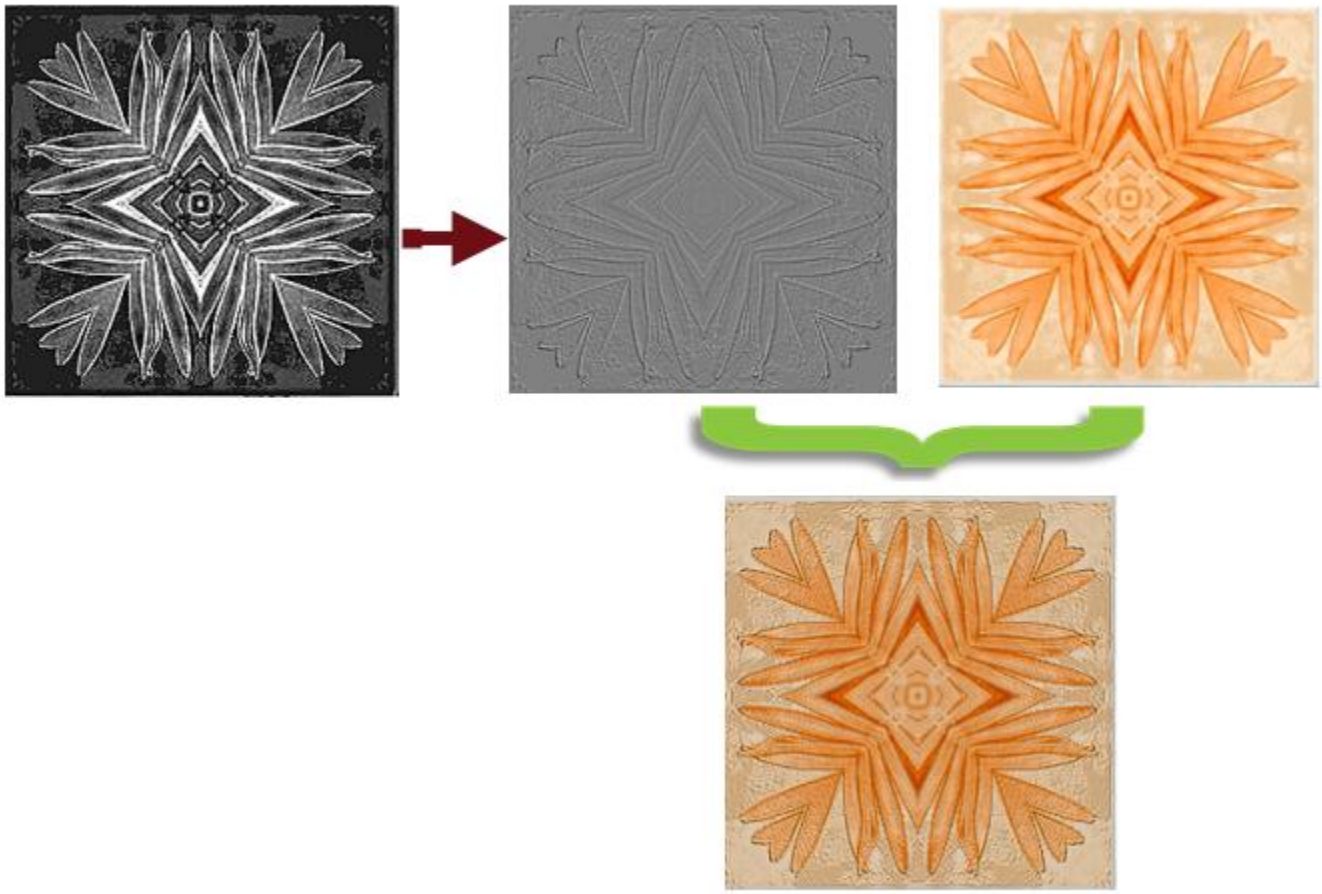


图 4. 浮雕纹理

一种 Blinn 技术的变种称为浮雕（凹凸）纹理（Emboss Bump Mapping），浮雕纹理技术使用两张纹理贴图来显示凹凸效果，一张纹理贴图用于产生浮雕效果，再与另一张纹理贴图混合，产生凹凸效果。如图 4 所示，通过对一张黑白纹理贴图的两次渲染，得到浮雕效果，再通过一次渲染，使浮雕效果与另一张彩色纹理贴图纹理混合，产生凹凸效果。

这里介绍下 NEHE[4]实现的浮雕纹理的原理，该技术只使用到漫反射分量，没有镜面反射的效果，会由于下采样造成锯齿。漫反射光的计算公式可以表示为：

$$C=\left(\vec{l}\cdot\vec{n}\right)\cdot D_l\cdot D_m \qquad (9)$$

其中，$\vec{l}$ 表示指向光源的方向向量，D_l 表示漫反射光的颜色，D_m 表示材质的漫反射颜色。

凹凸纹理的目的就是扰动法向量 $\vec{n}$，而浮雕纹理通过新的等式[5]来估计 $\vec{l}\cdot\vec{n}$，达到类似扰动的目的，即：

$$F_d+m\approx\vec{l}\cdot\vec{n} \qquad (10)$$

其中，F_d 表示初始的漫反射分量值，m 表示函数的一阶偏导，即表示凹凸函数的坡度。如图 4 所示，F_d 即右上角的彩色图表示的颜色值，m 是通过对灰度图的

可以用一个二维的灰度图来表示凹凸函数，偏导 m 可以通过下列方法来估计，具体的示例如图 5 所示。

1. 查询某个像素点 P 的高度值为 H_0，该点对应二维灰度图的纹理坐标为 (s,t)；
2. 根据像素点 P 与光源的位置 L，对纹理坐标进行一定的扰动，得到新的纹理坐标 $(s+\Delta s,t+\Delta t)$ 处的高度值 H_1；
3. 那么，$m=H_1-H_0$。

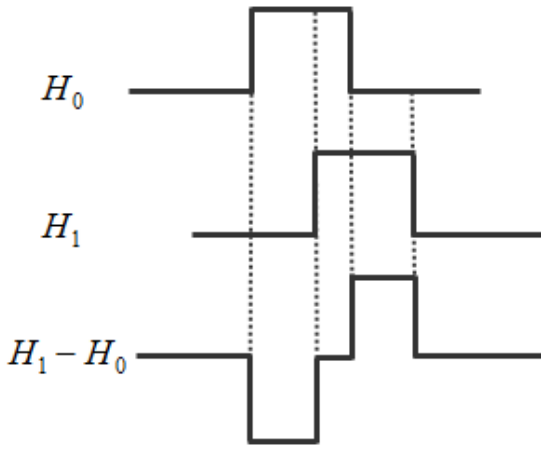


图 5. 在一维空间上，凹凸函数的一阶偏导 m 的示意图

现在考虑偏移量 $(\Delta s,\Delta t)$ 的计算，设点 P 的法向量（Normal）为 $\vec{n}$，它在切线（Tangent）和二重切线（Bitangent）分别为 $\vec{s},\vec{t}$，由点 P 指向光源 L 的方向向量表示为 $\vec{l}=(L-P)/\|L-P\|$，那么：

$$\Delta s = \lambda\left(\vec{l}\cdot\vec{s}\right), \Delta t = \lambda\left(\vec{l}\cdot\vec{t}\right) \qquad (11)$$

其中，λ是一个缩放常量。

对 NEHE 第 22 节[4]的代码进行简化，新的代码如下所示：

3. 法线贴图技术

法线贴图（Normal Mapping）也是通过对光照计算的法向量进行扰动，实现图像的凹凸效果。最早[11]由 Krishnamurthy& Levoy[8]提出从模型中提取模型信息的想法，尔后 Cohen 等[9], Cignoni 等[10]提出将模型的法向量信息存储在纹理当中的想法，就是法线贴图的基本思想。

首先介绍下什么是法线纹理，法线纹理是一张存储法向量的纹理。纹理的每个像素存储着 RGB 信息，它可以转化为法向量。设某个像素点的颜色为(r,g,b)，那么该点的法向量为$(r*2-1,g*2-1,b*2-1)$；相反，对于法向量(x,y,z)，其中，$0\le x,y,z\le 1$，它在法线纹理中存储的颜色为$(x*0.5+0.5,y*0.5+0.5,z*0.5+0.5)$。举个例子，对于法向量$(0.3,0.4,0)$，它在法线纹理中的颜色值$((0.3,0.4,0)*0.5+0.5)=(0.65,0.7,0.5)$。法线贴图纹理如图 6 所示。

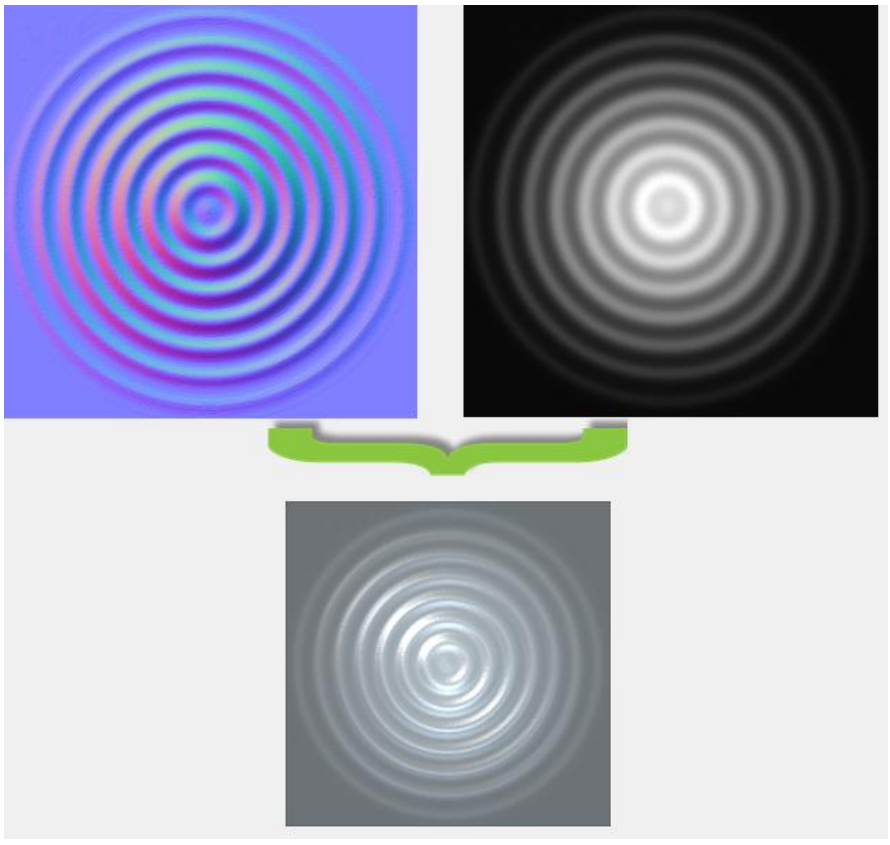


图 6. 法线纹理[12]

设人眼在位置 O(原点)，模型上存在一个点 P，对应的纹理坐标为 (s,t)，法线贴图的颜色值为 (r,g,b)，光源在点 Q(对象空间)，那么如何计算点 P 处的光照值呢？这里，引入切线空间的概念[13, 14]。

如图 7 所示表示位置 A 处的切线空间，它是一个相对于纹理坐标系的坐标系统，三条坐标轴分别为：法向量 $\vec{N}$，切线(Tangent) $\vec{T}$，双切线(Bitangent) $\vec{B}$（在世界空间下）。在切线空间下，这三条坐标轴分别为 $(0,0,1),(0,0,1),(0,0,1)$。相对于光源，要计算出光源点 Q 在点 P 上的颜色值，即需要计算视点坐标系统下光照向量 $\overrightarrow{PQ}$、视点向量 $\overrightarrow{PO}$ 在切线空间下的坐标值。

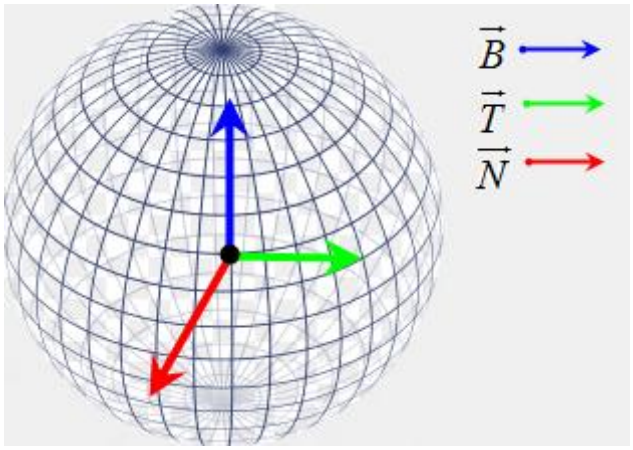


图 7. 切线空间

设三维空间上有三个点 P_1,P_2,P_3，纹理坐标分别为 $(s_1,t_1),(s_2,t_2),(s_3,t_3)$，则有：

$$\begin{cases}(s_1',t_1')=(s_2,t_2)-(s_1,t_1)\\(s_2',t_2')=(s_3,t_3)-(s_1,t_1)\end{cases},\begin{cases}D_1=P_2-P_1\\D_2=P_3-P_1\end{cases}\quad(12)$$

可以得到等式：

$$\begin{cases}D_1=s_1'\vec{T}+t_1'\vec{B}\\D_2=s_2'\vec{T}+t_2'\vec{B}\end{cases}\quad(13)$$

转化为矩阵表示为：

$$\begin{pmatrix}D_{1,x} & D_{1,y} & D_{1,z}\\D_{2,x} & D_{2,y} & D_{2,z}\end{pmatrix}=\begin{pmatrix}s_1' & t_1'\\s_2' & t_2'\end{pmatrix}\begin{pmatrix}T_x & T_y & T_z\\B_x & B_y & B_z\end{pmatrix}\quad(14)$$

通过计算 (s,t) 矩阵的逆矩阵，可以得到等式：

$$\begin{pmatrix}T_x & T_y & T_z\\B_x & B_y & B_z\end{pmatrix}=\frac{1}{s_1t_2-s_2t_1}\begin{pmatrix}t_2 & -t_1\\-s_2 & s_1\end{pmatrix}\begin{pmatrix}D_{1,x} & D_{1,y} & D_{1,z}\\D_{2,x} & D_{2,y} & D_{2,z}\end{pmatrix}\quad(15)$$

通过等式（15）计算出切线向量和双切线向量，那么可以得到变换矩阵：

$$M_1=\begin{pmatrix}T_x & T_y & T_z\\B_x & B_y & B_z\\N_x & N_y & N_z\end{pmatrix}\quad(16)$$

通过变换矩阵 M_1，可以将点由对象空间变换到切线空间，它的逆矩阵就可以把坐标由切线空间变换为对象空间下的坐标，需要计算出矩阵 M_1 的逆矩阵即可。采用 Gram-Schmidt 正交化方法，可以计算出新的切线向量和双切线向量：

$$\begin{cases} \overrightarrow{T'} = \vec{T} - \vec{N}\left(\vec{N}\cdot\vec{T}\right) \\ \overrightarrow{B'} = B - \left(\vec{N}\cdot\vec{B}\right)\vec{N} - \left(\overrightarrow{T'}\cdot\vec{B}\right)\overrightarrow{T'} \end{cases} \quad (17)$$

实际上，我们并不需计算双切线向量 $\overrightarrow{B'}$，可以通过 $\overrightarrow{N'}\times\overrightarrow{T'}$ 出 $\overrightarrow{B'}$，所以逆矩阵可以表示为：

$$M_2 = \begin{pmatrix} T_x' & T_y' & T_z' \\ B_x' & B_y' & B_z' \\ N_x' & N_y' & N_z' \end{pmatrix} \quad (18)$$

至此，我们得到了将坐标由对象空间变换到切线空间的变换矩阵 M_2。那么，光照方向 $\overrightarrow{PQ}$ 和物点位置 $\overrightarrow{PO}$ 转化为切线空间下的坐标为：$M_2\cdot\overrightarrow{PQ}$ 和 $M_2\cdot\overrightarrow{PO}$。点 P 在法线纹理的纹理颜色值为 (r,g,b)，它表示的法向量为 (x,y,z)，就可以利用 Phong 光照模型计算出 P 的 RGB 颜色了。

最后给出法线贴图技术的着色器代码如下所示：

顶点着色器代码 NormalMapping.vert：

```
#version 330 core
layout(location = 0) in vec3 Vertex;
layout(location = 1) in vec2 VertexUV;
layout(location = 2) in vec3 NormalModelSpace;
layout(location = 3) in vec3 TangentModelSpace;
layout(location = 4) in vec3 BitangentModelspace;
out vec2 TexVertUV;
out vec3 PosWorldSpace;
out vec3 LightDirTangSpace;
out vec3 EyeDirTangSpace;
uniform mat4 ModelViewProj;
uniform mat4 View;
uniform mat4 Model;
uniform mat3 MV3x3;
uniform vec3 LightPos;
void main(){
	// Output position of the vertex, in clip space : MVP * position
	gl_Position =    ModelViewProj * vec4(Vertex, 1);
	// Position of the vertex, in worldspace : Model * position
	PosWorldSpace = (Model * vec4(Vertex,1)).xyz;
	// Vector that goes from the vertex to the camera, in camera space.
	// Assume the camera is at (0,0,0) in world space.
	vec3 eyeDirCamSpace = (View * (vec4(0,0,0, 1) - vec4(PosWorldSpace,1))).xyz;
	// Vector that goes from the vertex to the light, in camera space.
	vec3 lightDirCamSpace = (View * (vec4(LightPos,1) - vec4(PosWorldSpace, 1))).xyz;
	// UV of the vertex. No special space for this one.
	TexVertUV = VertexUV;
	// model to camera = ModelView
	vec3 TangentCamSpace       = MV3x3 * TangentModelSpace;
	vec3 BitangentCamSpace     = MV3x3 * BitangentModelspace;
	vec3 NormalCamSpace        = MV3x3 * NormalModelSpace;
	// You can use dot products instead of building this matrix and transposing it. See References for details.
	mat3 TBN = transpose(mat3(TangentCamSpace,BitangentCamSpace,NormalCamSpace));
	LightDirTangSpace      = TBN * lightDirCamSpace;
	EyeDirTangSpace        = TBN * eyeDirCamSpace;
}
```

片断着色器代码 NormalMapping.frag：

```
#version 330 core
in vec2 TexVertUV;
in vec3 LightDirTangSpace;
```

```
in vec3 EyeDirTangSpace;
out vec3 color;
uniform sampler2D DiffTexSampler;
uniform sampler2D NormTexSampler;
uniform vec3 LightPos;
void main(){
	// Light emission properties
	// You probably want to put them as uniforms
	vec3 lightColor = vec3(1,1,1);
	float LightPower = 3.0;
	// Material properties
	vec3 d = texture2D(DiffTexSampler, TexVertUV).rgb;
	// Normal of the computed fragment, in camera space
	vec3 n = normalize(texture2D(NormTexSampler, vec2(TexVertUV.x, TexVertUV.y)).rgb * 2.0 - 1.0);
	// Direction of the light (from the fragment to the light)
	vec3 l = normalize(LightDirTangSpace);
	vec3 v = normalize(EyeDirTangSpace);
	float power = 0.3;
	// ambient lighting
	float iamb = 0.1;
	// diffuse lighting
	float idiff = clamp(dot(n, l), 0, 1);
	float ispec = clamp(dot(v + l, n), 0, 1) * power;
	color = d * (iamb + idiff + ispec);
}
```

4. 视差贴图技术

4.1. 综述

视差贴图技术（ParallaxMapping）可以看成是法线贴图技术的加强版，并没有改变模型的结构，而只是通过改变光照的计算达到 3D 视感。对于法线贴图技术来说，它存在一个问题，例如一个墙壁上的砖块，当你以一定的倾斜角观察它的时候，依旧可以看到砖块之间的缝隙，砖块间没有互相遮挡的效果，视差贴图技术就是为了解决这个问题，如图 8 所示，图(a)和图(b)分别是法线贴图技术和视差贴图技术达到的效果，显然图(b)的石头立体感更加显著。

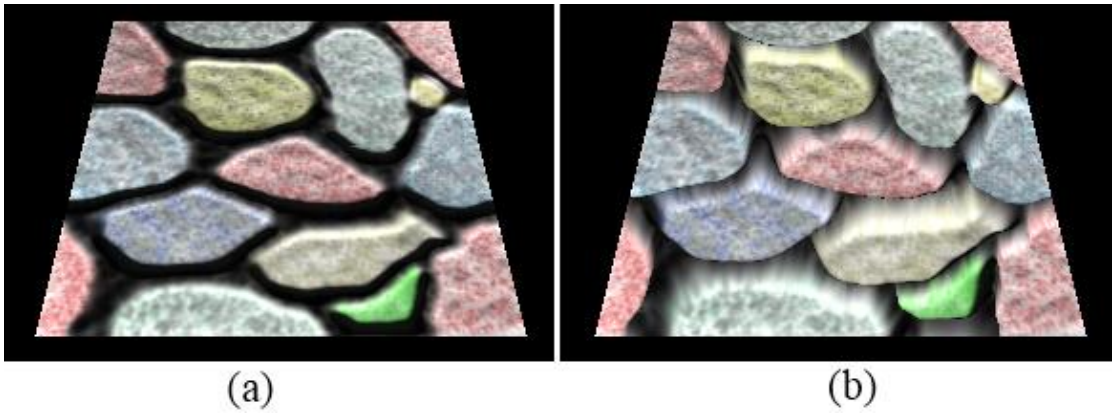


图 8. 法线贴图技术和视差贴图技术的效果图

视差贴图技术最早由 Kaneko 等[15]在 2001 年提出的，后来被持续改进，主要有下面几个变种：1）Parallax Mapping with Offset Limiting，2）Steep Parallax

Mapping，3）Relief Parallax Mapping，4）Parallax Occlusion Mapping（嗯，不知道怎么翻译这几个名词）。Dujgta[16]对这几个技术有非常清晰的阐述，这里分别对这几个技术进行介绍。

特别注意：视差贴图技术解决的凹凸纹理中无法解决互相遮挡的问题。

为了实现视差贴图技术，至少需要有三种纹理贴图：高度贴图纹理，法线贴图纹理和漫反射贴图纹理，分别如图 9(a)、(b)、(c)所示。深度贴图纹理的 R 通道、G 通道、B 通道和 A 通道的值是相同的，黑色的表示深度值为 0，白色的表示深度值为 1；法线贴图纹理，有 RGB 三个颜色通道，因此，可以把深度贴图纹理和法线贴图纹理合并为一张纹理，RGB 通道表示法线，A 通道表示深度；漫反射贴图纹理用于模型表示的颜色渲染。

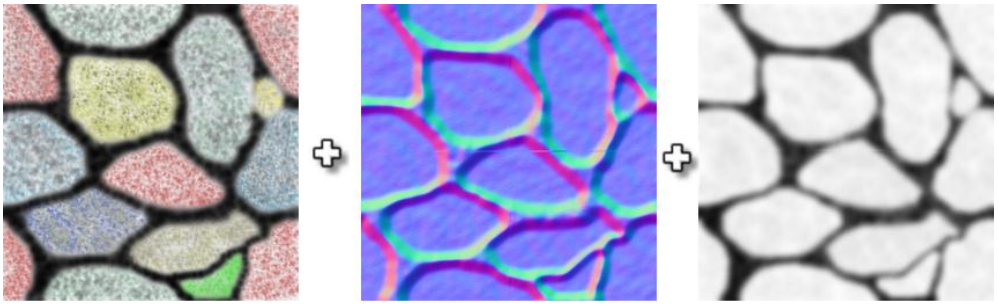


图 9. 三种贴图纹理，（a）深度贴图纹理，（b）法线贴图纹理，（c）漫反射贴图纹理

视差贴图技术跟法线贴图技术的区别在于，它需根据眼睛位置和深度贴图信息，对法线进行偏移。原理如图 10 所示，法线贴图技术计算视点向量与纹理平面的交点在 t_0 位置，但实际上，由于纹理表现存在一定的高度值，采用 t_0 处的纹理坐标计算出的颜色值存在一定的误差。视差贴图技术的目的就是：基于视点向量 $\vec{v}$，计算出真实的位置 t_1 处的纹理坐标。

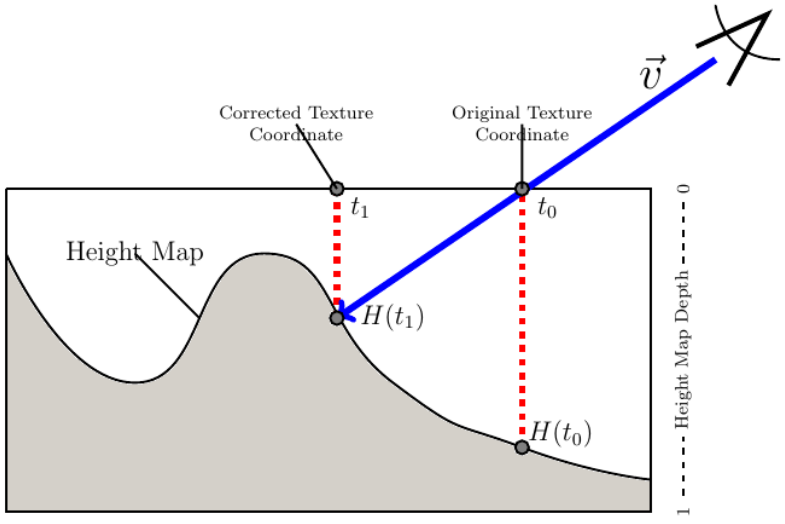


图 10. 视差贴图技术原理示意图

这里给出视差贴图技术的着色器的代码结构，顶点着色器与法线贴图技术的相同，但是片断着色器添加了对得到的纹理坐标进一步进行扰动，从而采用新的纹理坐标处的法向量进行光照的计算。

顶点差色器代码 ParallaxMapping.vert：

```
#version 330 core
layout(location = 0) in vec3 Vertex;
layout(location = 1) in vec2 VertexUV;
layout(location = 2) in vec3 NormalModelSpace;
layout(location = 3) in vec3 TangentModelSpace;
layout(location = 4) in vec3 BitangentModelspace;
out vec2 TexVertUV;
out vec3 EyeTangSpace;
out vec3 LightTangSpace;
uniform mat4 ModelViewProj;
uniform mat4 View;
uniform mat4 Model;
uniform mat3 MV3x3;
uniform vec3 LightPos;
uniform int gSwitch;
void main(){
	// Output position of the vertex, in clip space : MVP * position
	gl_Position =    ModelViewProj * vec4(Vertex, 1);
	// the View matrix is identity here.
	// Position of the vertex, in worldspace : Model * position
	vec3 posWorldSpace = (Model * vec4(Vertex,1)).xyz;
	// Vector that goes from the vertex to the light.
	vec3 lightPosWorldSpace = (View * vec4(LightPos,1)).xyz;
	vec3 lightDirWorldSpace = normalize(lightPosWorldSpace - posWorldSpace);
	vec3 camPosWorldSpace = vec3(0,0,0);
	vec3 camDirCamSpace = normalize(camPosWorldSpace - posWorldSpace);
	// UV of the vertex. No special space for this one.
	TexVertUV = VertexUV;
	// model to camera = ModelView
	vec3 TangentCamSpace = MV3x3 * TangentModelSpace;
	vec3 BitangentCamSpace = MV3x3 * BitangentModelspace;
	vec3 NormalCamSpace = MV3x3 * NormalModelSpace;
	// You can use dot products instead of building this matrix and transposing it. See References for details.
	mat3 TBN = transpose(mat3(TangentCamSpace,BitangentCamSpace,NormalCamSpace));
	LightTangSpace = TBN * lightDirWorldSpace;
	EyeTangSpace =    TBN * camDirCamSpace;
}
```

片段着色器代码 ParallaxMapping.frag：

```
#version 330 core
in vec2 TexVertUV;
in vec3 EyeTangSpace;
in vec3 LightTangSpace;
out vec3 color;
uniform sampler2D DiffTexSampler;
uniform sampler2D NormTexSampler;
uniform vec3 LightPos;
uniform int gSwitch;
float gHeightScale = 0.1;

// Calculates lighting by Blinn-Phong model and Normal Mapping
// Returns color of the fragment
vec3 normalMappingLighting(in vec2 t, in vec3 l, in vec3 v){
	// restore normal from normal map
```

```
    vec3 n = normalize(texture(NormTexSampler, t).rgb * 2.0 - 1.0);
    vec3 d = texture(DiffTexSampler, t).rgb;
    float power = 0.3;
    // ambient lighting
    float iamb = 0.1;
    // diffuse lighting
    float idiff = clamp(dot(n, l), 0, 1);
    float ispec = clamp(dot(v + l, n), 0, 1) * power;
    return d * (iamb + idiff + ispec);
}
vec2 pallaxMapping(in vec3 v, in vec2 t){
     ……
}
void main(){
    // Direction of the light (from the fragment to the light)
    vec3 l = normalize(LightTangSpace);
    // Direction of the eye (from the fragment to the eye)
    vec3 v = normalize(EyeTangSpace);
    vec2 t = pallaxMapping2(v, TexVertUV);
    color = normalMappingLighting(t, l, v);
}
```

4.2. Parallax Mapping with Offset Limiting

最简单的视差贴图技术，通过单步扰动纹理坐标来实现，称之为 Parallax Mapping with Offset Limiting，虽然它的表现却让人很失望，但它是后续几种视差技术的基础，这里先对该技术进行介绍。

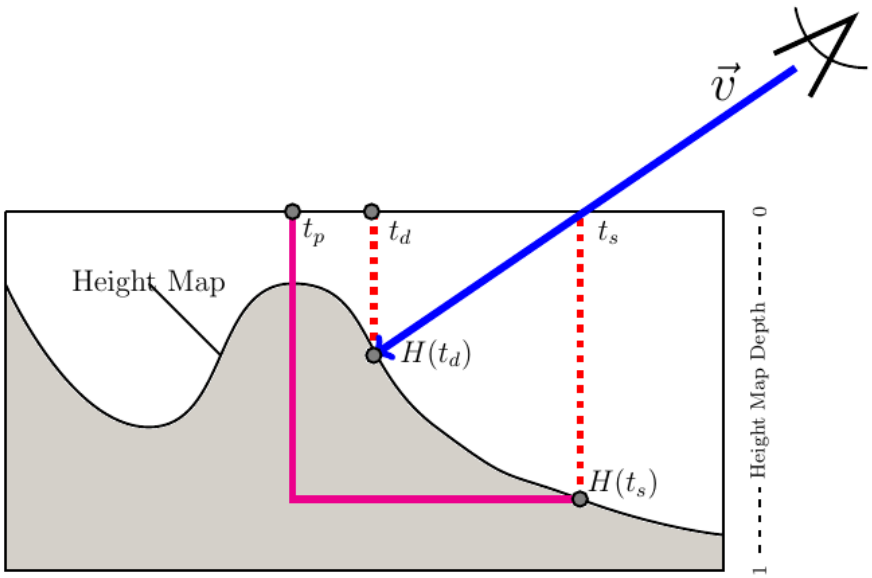


图 11. Parallax Mapping with Offset Limiting 原理示意图

如图 11 所示，设视点向量 $\vec{v}$ 是在切线空间下的数值，那么它的 z 轴与纹理平面互相垂直，我们的目的是根据纹理坐标 t_s 和它的深度值 $H(t_s)$ 估计出一个纹理

坐标 t_p 使它尽可能的靠近 t_d。估计方程式可以表示为：

$$t_p = t_s + \vec{v}_{xy} / \vec{v}_z * H(t_s) * scale \quad (19)$$

其中，通常 $scale$ 的取值介于 $[0,0.5]$ 之间[16]，该算法得到的效果如图 12 所示，相应的片断着色器代码如下所示：

```
vec2 pallaxWithOffsetLimit(in vec3 v, in vec2 t){
        float height =    texture(NormTexSampler, t).a;
        vec2 offset = v.xy / v.z * height * gHeightScale;
        return t - offset;
}
```

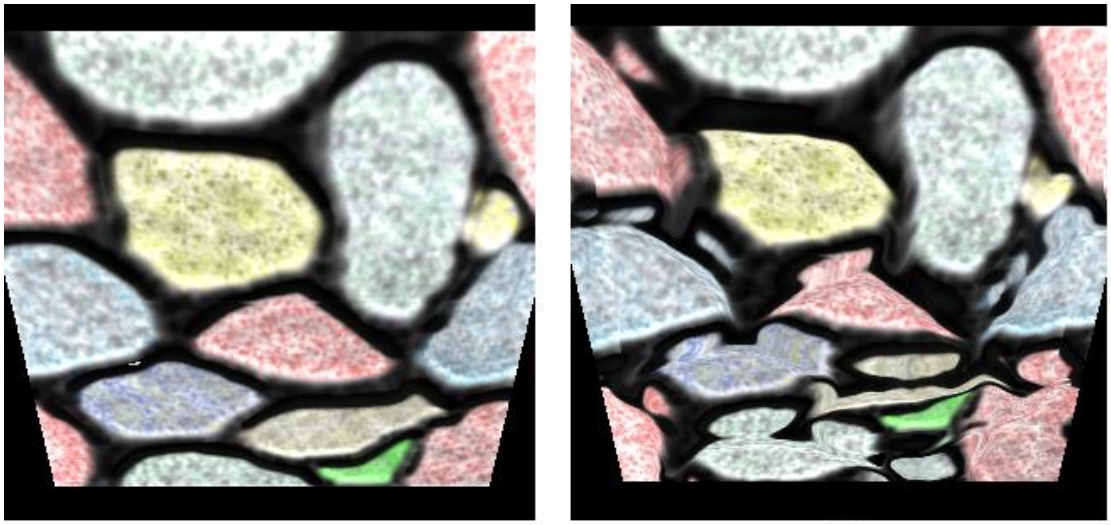


图 12. Parallax Mapping with Offset Limiting 效果图对比，（a）法线贴图效果图，(b)视差贴图效果图

4.3. Steep Parallax Mapping

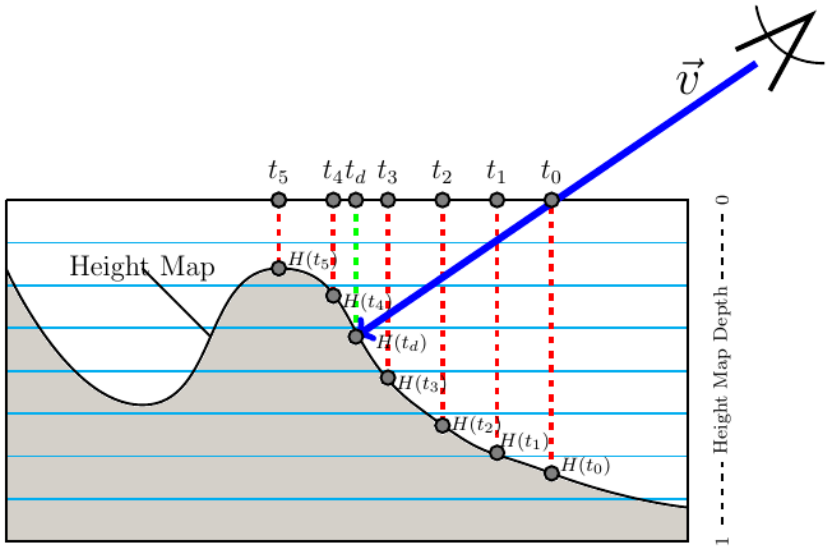


图 13. SPM 原理示意图

与 Parallax Mapping with Offset Limiting 技术不同的是，Steep Parallax

Mapping(SPM)有对计算出的新的纹理坐标进行校验，判断它是否尽可能的接近真实的纹理坐标值。它把深度图的范围$[0,1]$，平均分成n份，每次移动一点，直到找到一个点在层表面的下方（即当前的层的深度比采样点的深度值大），停止查询，选择当前点作为目标点。设$n=8$为例，如图 13 所示，第一次判断点t_0，该点在第 1 层上方（即从上往下数第一条蓝绿色线），搜索下一个点；判断点t_1，它也在第 2 层上方，搜索下一个点；直到找到t_4，即是我们要找的目标点。

如果n值越大，则计算出来的纹理坐标越接近实际值，但会降低性能；如果n的取值较小，就会产生明显的锯齿效果，如图 14 所示。

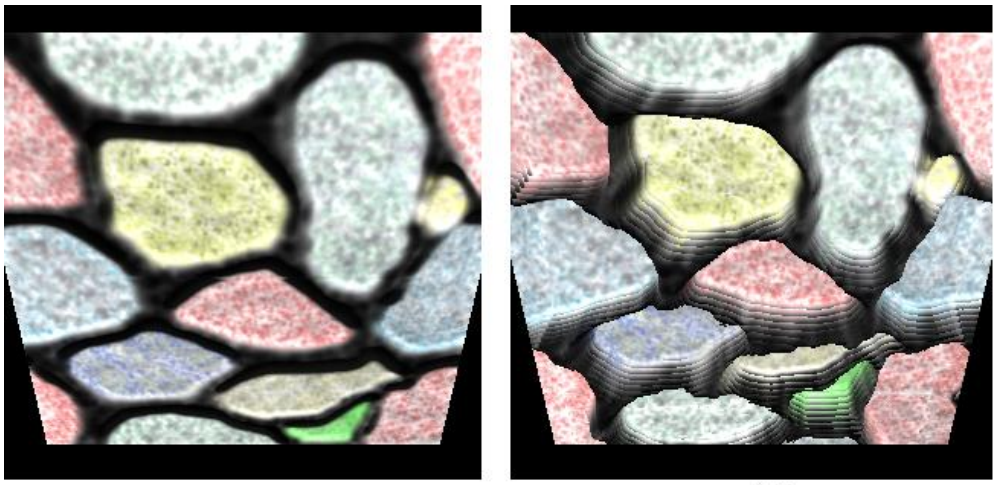


图 14. SPM 效果图对比，（a）法线贴图效果图，(b)视差贴图效果图

SPM 技术的着色器代码如下所示：

```
vec2 steepPallaxMapping(in vec3 v, in vec2 t){
        // determine number of layers from angle between V and N
        const float minLayers = 5;
        const float maxLayers = 15;
        float numLayers = mix(maxLayers, minLayers, abs(dot(vec3(0, 0, 1), v)));
        // height of each layer
        float layerHeight = 1.0 / numLayers;
        // depth of current layer
        float currentLayerHeight = 0;
        // shift of texture coordinates for each iteration
        vec2 dtex = gHeightScale * v.xy / v.z / numLayers;
        // current texture coordinates
        vec2 currentTextureCoords = t;
        // get first depth from heightmap
        float heightFromTexture = texture(NormTexSampler, currentTextureCoords).a;
        // while point is above surface
        while(heightFromTexture > currentLayerHeight) {
                // to the next layer
                currentLayerHeight += layerHeight;
                // shift texture coordinates along vector V
                currentTextureCoords -= dtex;
                // get new depth from heightmap
                heightFromTexture = texture(NormTexSampler, currentTextureCoords).a;
    }
    return currentTextureCoords;
}
```

4.4. Relief Parallax Mapping and Parallax Occlusion Mapping

Relief Parallax Mapping（RPM）与 SPM 技术相比，多了一个后处理。如图 13 所示，即找到点 t_4 后，再在 t_3 和 t_4 进行一定步骤的二分查找，但是产了较为明显的效果改进，如图 15 所示。

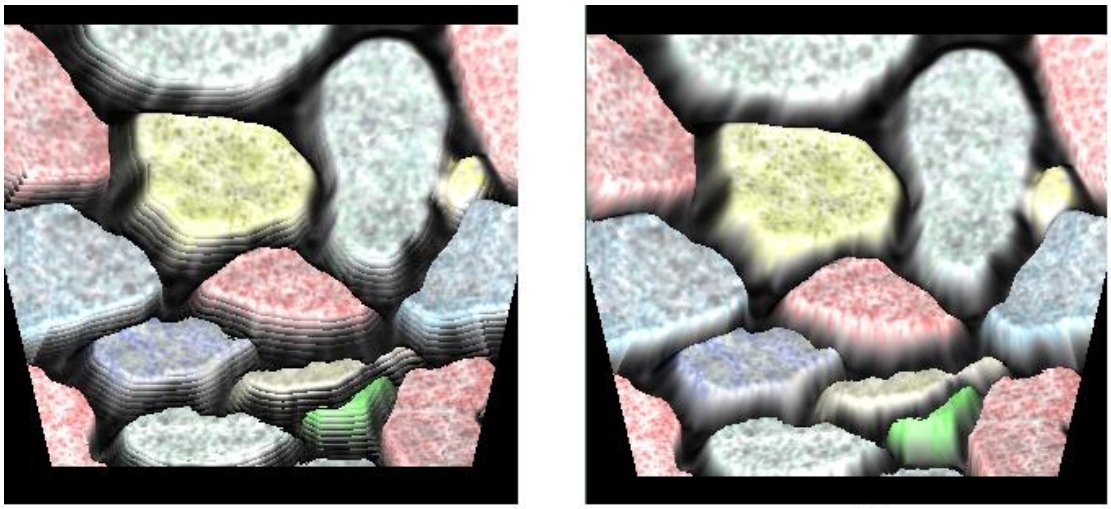


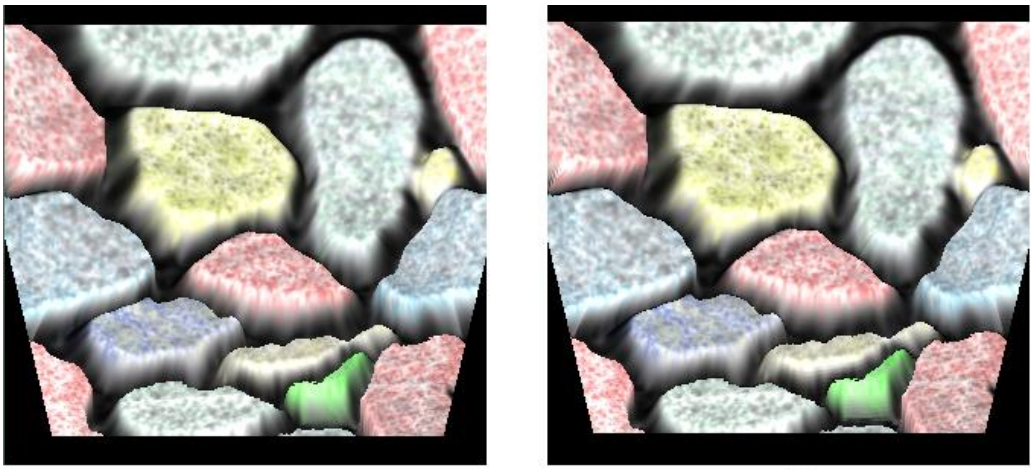
图 15.效果图，（a）SPM 效果图，(b) RPM 效果图

RPM 技术的着色器代码如下所示：

```
vec2 reliefPallaxMapping(in vec3 v, in vec2 t){
        // determine required number of layers
        const float minLayers = 10;
        const float maxLayers = 15;
        float numLayers = mix(maxLayers, minLayers, abs(dot(vec3(0, 0, 1), v)));
        // height of each layer
        float layerHeight = 1.0 / numLayers;
        // depth of current layer
        float currentLayerHeight = 0;
        // shift of texture coordinates for each iteration
        vec2 dtex = gHeightScale * v.xy / v.z / numLayers;
        // current texture coordinates
        vec2 currentTextureCoords = t;
        // depth from heightmap
        float heightFromTexture = texture(NormTexSampler, currentTextureCoords).a;
        // while point is above surface
        while(heightFromTexture > currentLayerHeight) {
                // go to the next layer
                currentLayerHeight += layerHeight;
                // shift texture coordinates along V
                currentTextureCoords -= dtex;
                // new depth from heightmap
                heightFromTexture = texture(NormTexSampler, currentTextureCoords).a;
    }
        // Start of Relief Parallax Mapping
        // decrease shift and height of layer by half
        vec2 deltaTexCoord = dtex / 2;
        float deltaHeight = layerHeight / 2;
        // return to the mid point of previous layer
        currentTextureCoords += deltaTexCoord;
        currentLayerHeight -= deltaHeight;
        // binary search to increase precision of Steep Paralax Mapping
        const int numSearches = 5;
        for(int i = 0; i < numSearches; i++){
```

```
            // decrease shift and height of layer by half
            deltaTexCoord /= 2;
            deltaHeight /= 2;
            // new depth from heightmap
            heightFromTexture = texture(NormTexSampler, currentTextureCoords).a;
            // shift along or agains vector V
            if(heightFromTexture > currentLayerHeight) // below the surface{
                currentTextureCoords -= deltaTexCoord;
                currentLayerHeight += deltaHeight;
            }
            else // above the surface{
                currentTextureCoords += deltaTexCoord;
                currentLayerHeight -= deltaHeight;
            }
        }
        return currentTextureCoords;
}
```

RPM 技术通过在 t_3 和 t_4 范围内做二分查找，得到最终的纹理坐标；但是 POM 技术只基于当前的层信息，对 t_3、t_4 进行加权计算得到最终的结果。所以 POM 的性能比 RPM 技术更佳，但是效果比它略差，由于忽略了更多的细节，有可能会产生错误的结果。RPM 和 POM 的效果对比如图 16 所示，并不会看出明显的差异。



(a) (b)

图 15.效果图，（a）RPM 效果图，(b) POM 效果图

POM 技术的着色器代码如下所示：

```
vec2 occlusionPallaxMapping1(in vec3 v, in vec2 t){
        // determine optimal number of layers
        const float minLayers = 10;
        const float maxLayers = 15;
        float numLayers = mix(maxLayers, minLayers, abs(dot(vec3(0, 0, 1), v)));
        // height of each layer
        float layerHeight = 1.0 / numLayers;
        // current depth of the layer
        float curLayerHeight = 0;
        // shift of texture coordinates for each layer
        vec2 dtex = gHeightScale * v.xy / v.z / numLayers;
        // current texture coordinates
        vec2 currentTextureCoords = t;
        // depth from heightmap
        float heightFromTexture = texture(NormTexSampler, currentTextureCoords).a;
        // while point is above the surface
        while(heightFromTexture > curLayerHeight) {
            // to the next layer
```

```
            curLayerHeight += layerHeight;
            // shift of texture coordinates
            currentTextureCoords -= dtex;
            // new depth from heightmap
            heightFromTexture = texture(NormTexSampler, currentTextureCoords).a;
        }
        // previous texture coordinates
        vec2 prevTCoords = currentTextureCoords + dtex;
        // heights for linear interpolation
        float nextH = heightFromTexture - curLayerHeight;
        float prevH = texture(NormTexSampler, prevTCoords).a - curLayerHeight + layerHeight;
        // proportions for linear interpolation
        float weight = nextH / (nextH - prevH);
        // interpolation of texture coordinates
        vec2 finalTexCoords = prevTCoords * weight + currentTextureCoords * (1.0-weight);
        // return result
        return finalTexCoords;
}
```

此外，提供另外一个版本的 POM 的实现，它是由 Zink[17]采用 HLSL 改为 GLSL 的实现版本，与前面的实现相比，边缘更加光滑，如图 16 所示，着色器代码如下所示：

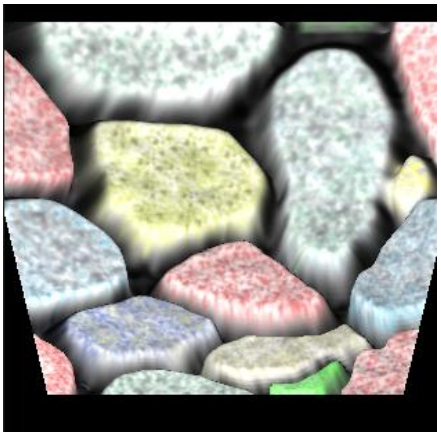


图 15. POM 效果图

```
vec2 occlusionPallaxMapping2(in vec3 v, in vec2 t){
        int             nMaxSamples                 = 15;
        int             nMinSamples                 = 10;
        float  fHeightMapScale              = 0.1;
        int nNumSamples = int(mix(nMaxSamples, nMinSamples, abs(dot(vec3(0, 0, 1), v))));
        // height of each layer
        float fStepSize = 1.0 / float(nNumSamples);
        // Calculate the parallax offset vector max length.
        // This is equivalent to the tangent of the angle between the
        // viewer position and the fragment location.
        float fParallaxLimit = length(v.xy ) / v.z;
        // Scale the parallax limit according to heightmap scale.
        fParallaxLimit *= fHeightMapScale;
        // Calculate the parallax offset vector direction and maximum offset.
        vec2 vOffsetDir = normalize(v.xy);
        vec2 vMaxOffset = vOffsetDir * fParallaxLimit;
        // Initialize the starting view ray height and the texture offsets.
        float fCurrRayHeight = 1.0;
        vec2 vCurrOffset = vec2(0, 0);
        vec2 vLastOffset = vec2(0, 0);
        vec2 dx = dFdx(t);
        vec2 dy = dFdy(t);
        float fLastSampledHeight = 1;
        float fCurrSampledHeight = 1;
        int nCurrSample = 0;
        while ( nCurrSample < nNumSamples ){
            // Sample the heightmap at the current texcoord offset.    The heightmap
```

```
            // is stored in the alpha channel of the height/normal map.
            //fCurrSampledHeight = tex2Dgrad( NH_Sampler, IN.texcoord + vCurrOffset, dx, dy ).a;
            fCurrSampledHeight = textureGrad(NormTexSampler, TexVertUV + vCurrOffset, dx, dy).a;
            // Test if the view ray has intersected the surface.
            if (fCurrSampledHeight > fCurrRayHeight){
                  // Find the relative height delta before and after the intersection.
                  // This provides a measure of how close the intersection is to
                  // the final sample location.
                  float delta1 = fCurrSampledHeight - fCurrRayHeight;
                  float delta2 = (fCurrRayHeight + fStepSize) - fLastSampledHeight;
                  float ratio = delta1 / (delta1 + delta2);

                  // Interpolate between the final two segments to
                  // find the true intersection point offset.
                  vCurrOffset = ratio * vLastOffset + (1.0 - ratio) * vCurrOffset;

                  // Force the exit of the while loop
                  nCurrSample = nNumSamples + 1;
            }
            else{
                  // The intersection was not found.    Now set up the loop for the next
                  // iteration by incrementing the sample count,
                  nCurrSample ++;
                  // take the next view ray height step,
                  fCurrRayHeight -= fStepSize;
                  // save the current texture coordinate offset and increment
                  // to the next sample location,
                  vLastOffset = vCurrOffset;
                  vCurrOffset += fStepSize * vMaxOffset;
                  // and finally save the current heightmap height.
                  fLastSampledHeight = fCurrSampledHeight;
            }
      }
      // Calculate the final texture coordinate at the intersection point.
      return TexVertUV + vCurrOffset;
}
```

参考

[1] James F. Blinn. "Simulation of wrinkled surfaces." ACM SIGGRAPH Computer Graphics, vol.12, no.3, pp.286-292, 1978.

[2] Wikipedia. “Bump Mapping.” website <https://en.wikipedia.org/wiki/Bump_mapping>.

[3] Tomas Akenine-Mӧller, Eric Haines, and Naty Hoffman. *Real-time rendering*. CRC Press, 2008.

[4] NEHE Production. "22. Bump-Mapping, Multi-Texturing & Extensions."

[5] Michael I. Gold. "Emboss Bump Mapping." NVIDIA Corporation. (ppt)

[6] John Schlag. "Fast embossing effects on raster image data." Graphics Gems IV. Academic Press Professional, Inc. 1994.

[7] Brian Lingard. “Bump Mapping.” website <http://web.cs.wpi.edu/~matt/courses/cs563/talks/bump/bumpmap.html>, 1995.

[8] Venkat Krishnamurthy and Marc Levoy. "Fitting smooth surfaces to dense polygon meshes." Proceedings of the 23rd annual conference on Computer graphics and interactive techniques. ACM, 1996.

[9] Jonathan Cohen, Marc Olano, and Dinesh Manocha. "Appearance-preserving simplification." Proceedings of the 25th annual conference on Computer graphics

and interactive techniques. ACM, 1998.
[10] Paolo Cignoni, et al. "A general method for preserving attribute values on simplified meshes." Visualization'98. Proceedings. IEEE, 1998.
[11] Wikipedia. “Normal Mapping.” website < https://en.wikipedia.org/wiki/Normal_mapping>.
[12] Christian Petry. “Normal Mapping.” website <http://cpetry.github.io/NormalMap-Online/>.
[13] Eric Lengyel. “Mathematics for 3D game programming and computer graphics.” Cengage Learning, 2012.
[14] Siddharth Hegde. “Messing with Tangent Space.” website <http://www.gamasutra.com/view/feature/129939/messing_with_tangent_space.php>.
[15] Tomomichi Kaneko, et al. "Detailed shape representation with parallax mapping." Proceedings of ICAT. vol.2001, 2001.
[16] Dujgta. “Parallax Occlusion Mapping in GLSL.” website< http://sunandblackcat.com/tipFullView.php?topicid=28>.
[17] Jason Zink. “A Closer Look At Parallax Occlusion Mapping.” website<http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/a-closer-look-at-parallax-occlusion-mapping-r3262>.