

Per-Fragment Effects (I)

Mapping Techniques → Creating Visual Details

1. Per-pixel lighting
2. Texture mapping
3. **Bump mapping**
 - a) Normal Mapping
 - b) Height Mapping
 - c) Procedural Bump Mapping
 - d) Parallax Bump Mapping

General Idea

- For each fragment (pixel), specify how it is rendered

```
vec4 ColourAtFrag()  
{  
    vec4 myCol;  
    myCol = ...  
    return myCol;  
}
```

```
void main(void)  
{  
    gl_FragColor = ColourAtFrag();  
}
```

General Idea

- Implemented in a fragment shader
- Use **textures as a means** for representing the properties of geometric objects
 - for specifying surface details
 - material properties
 - normal, height, visibility information, ...
 - position, velocity, ...
 -
 - for representing complex functions in a discrete form, such as noise functions
 - for storing intermediate rendering results

How to Use a Texture in a Fragment Shader

- In the fragment shader, declare a **uniform variable** of type **sampler** for each texture to be used
- Specify texture coordinates associated to each vertex and pass it to the fragment shader
 - Two ways:
 1. using a varying variable (compatibility profile)
 2. using keywords “out” and “in” (core profile)
- Depending on the type of the textures to be used, a built-in function can be used to access the values of a texture:
 - texture1D, texture2D, texture3D, textureCube, shadow1D, ...

2D Texture Mapping in GLSL

Vertex shader

```
//out vec2 Texcoord; // core profile based  
varying vec2 Texcoord; // compatibility profile only
```

Same as, say:
#version 450 compatibility
out vec2 Texcoord

```
void main( void )  
{  
    gl_Position = ftransform();  
    Texcoord = gl_MultiTexCoord0.xy;  
}
```

Fragment shader

```
//in vec2 Texcoord; //core profile based  
uniform sampler2D baseMap;  
varying vec2 Texcoord; // compatibility profile only
```

Same as, say:
#version 450 compatibility
in vec2 Texcoord

```
void main( void )  
{  
    gl_FragColor =  
        texture2D( baseMap, Texcoord );  
}
```

same as texture()

Multitexturing

- With shaders, it is easy to use several textures together for different purposes

```
uniform sampler2D baseMap;
```

```
uniform sampler2D maskMap;
```

```
...
```

```
vec4 TexColor1= texture2D( baseMap, Texcoord );
```

```
vec4 TexColor2 = texture2D(maskMap, Texcoord );
```

```
vec4 AddColor= TexColor1+ TexColor2;
```

```
vec4 MulColor = TexColor1*TexColor2;
```

Use a Texture as a Stencil Mask

```
uniform sampler2D baseMap;  
uniform sampler2D maskMap;  
  
...  
vec4 TexCol= texture2D( baseMap, Texcoord );  
float maskVal= texture2D( maskMap, Texcoord ).r;  
  
... ...  
gl_FragColor = TexCol;  
if (maskVal<0.6)  
    discard;
```



Bump Mapping

1. Normal Mapping
2. Height Mapping
3. Procedural Bump Mapping
4. Parallax Bump Mapping

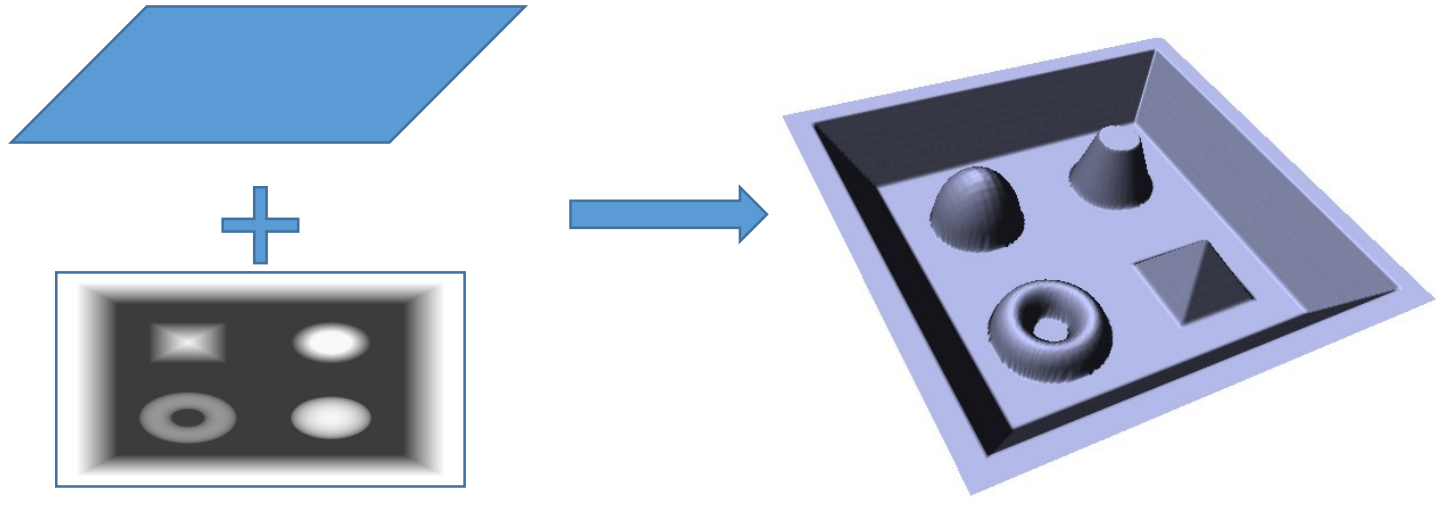
What Is Bump Mapping

- Bump mapping
 - A simple computer graphics technique to create **visual** surface details without having to change the shape of the underlying geometric model
 - Based on the fact
 - what we see is the amount of light arriving at our eyes
 - perturb the surface normal at a point on the surface will change the amount of light arriving at eyes from that point
 - diffuse light, specular light, ...

Bump Mapping Ideas

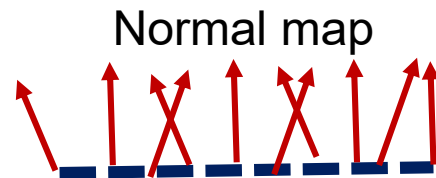
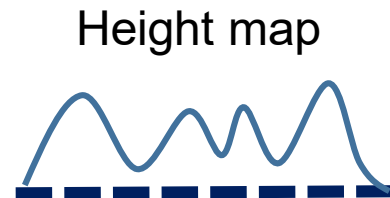
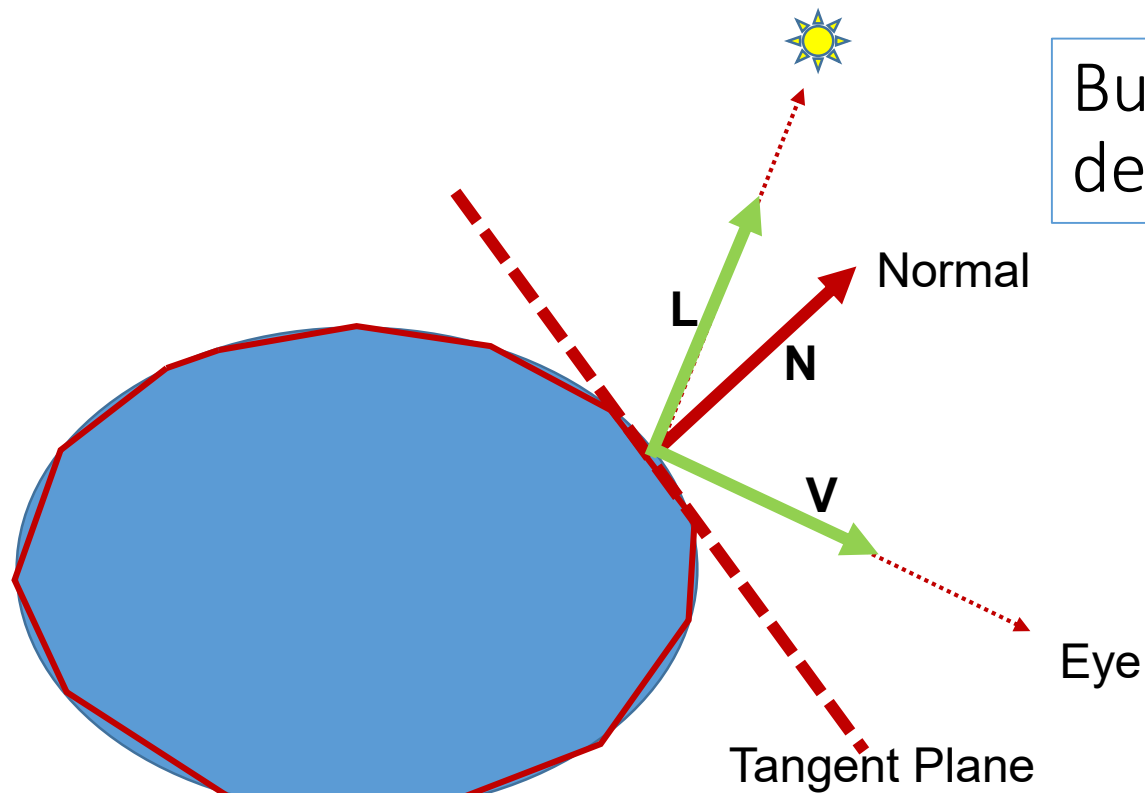
- Normal mapping
 - Surface normal is read directly from a normal map
- Height mapping
 - Surface normal is calculated from a height map
- Procedural bump mapping
 - The normal map is procedurally generated.
- Parallax mapping
 - Use two maps: a height map and a normal map
 - Height map is used for parallax compensation
- Relief mapping
 - A much finer bump mapping technique
 - Supporting self-occlusion, self-shadowing, view-motion parallax, and silhouettes

Simple, if the object is a flat plane!



But how to create bump effects if surface is curved?

Bump info is usually defined in tangent space



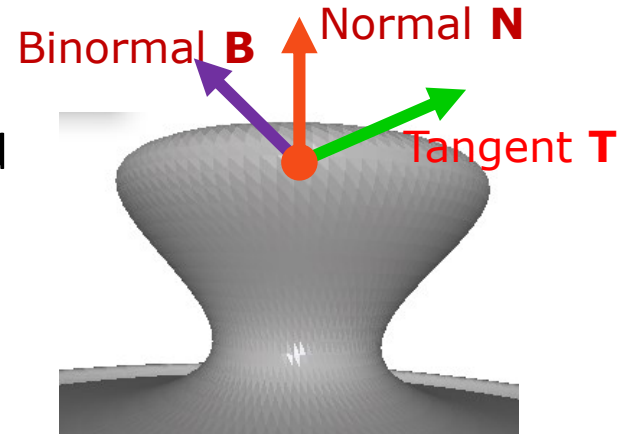
$\text{total light} = \text{ambient} + \text{diffuse} + \dots$

$\text{dot}(\mathbf{L}, \mathbf{N})$

Transform into Tangent Space

- **Tangent space**

- Defined based on each individual point on the surface using three mutually orthogonal vectors: normal, binormal and tangent (see the figure)
 - Specified as vertex attributes at the time of creating vertex buffer object
 - Different surface positions, different tangent coordinate systems
- Varying according to the normal at a point on the surface
- To do bump mapping, relevant information required for computing illumination quantities need to be transformed into the same space
 - Usually into the tangent space

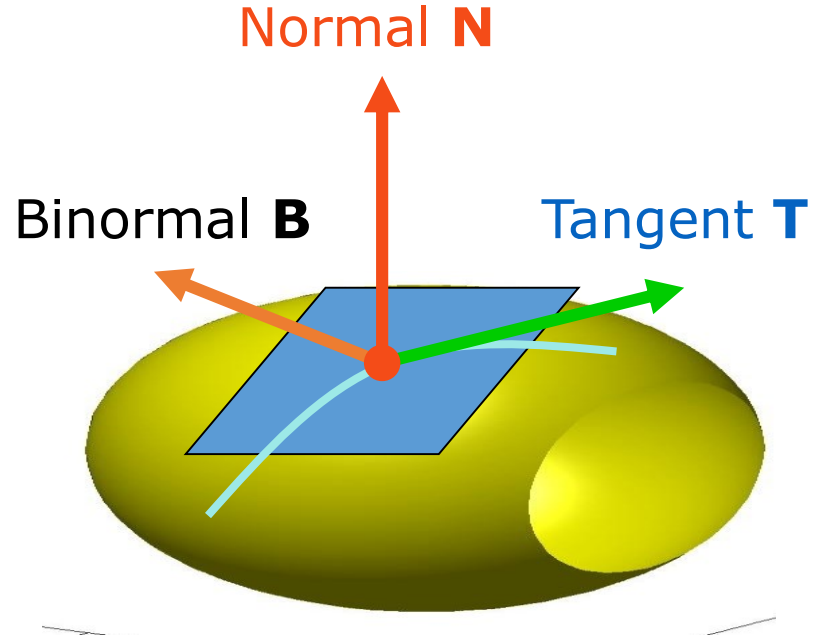


Transform into Tangent Space

Vectors **T**, **B**, **N**:

- All are unit vectors
- Mutually orthogonal
- The transformation from model-view space into tangent space can be defined directly using the three vectors

$$\begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{pmatrix}$$



This matrix depends on how the local coordinate system is defined for the underlying geometric model

Transform into View Space

... ..

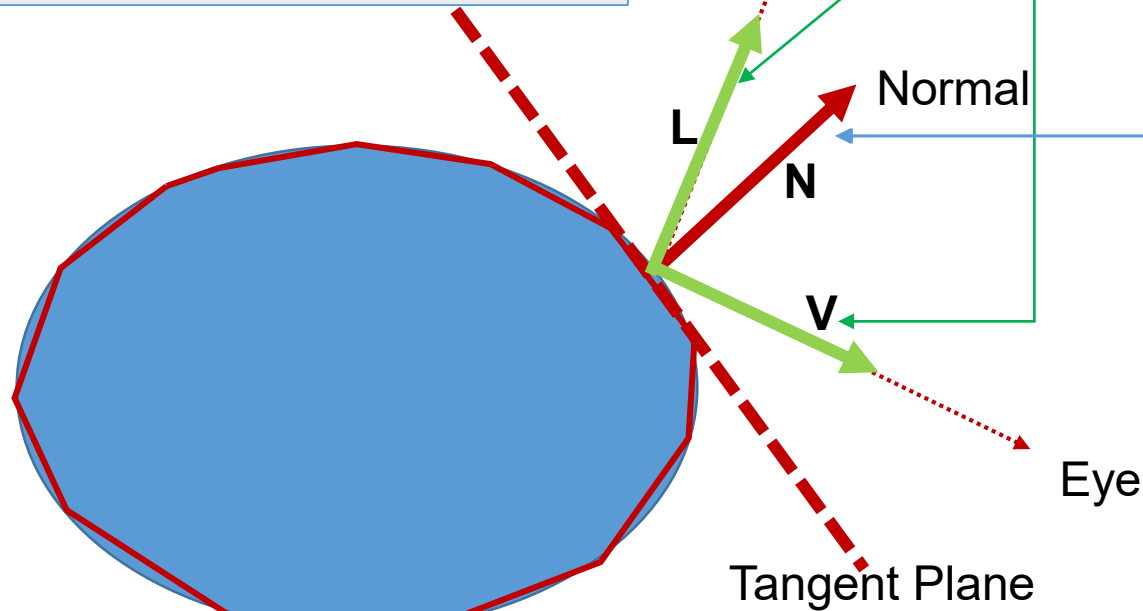
```
varying vec2 Texcoord;  
varying vec3 ViewDirInTangent ;  
varying vec3 LightDirInTangent ;
```

```
attribute vec3 rm_Binormal;  
attribute vec3 rm_Tangent;
```

```
void main( void )  
{
```

```
    ... ..  
    //Find vectors of view direction, light direction---view space based:  
    vec3 ViewDir  =  - PView.xyz;  
    vec3 LightDir =  LightPos.xyz - PView.xyz;
```

Bump mapping idea



$\text{total light} = \text{ambient} + \text{diffuse} + \dots$

$\text{dot}(\mathbf{L}, \mathbf{N})$

Transform into Tangent Space

//1. Transform Normal, Binormal and Tangent vectors into view space:

```
vec3 fvNormal      = gl_NormalMatrix * gl_Normal;  
vec3 fvBinormal    = gl_NormalMatrix * rm_Binormal;  
vec3 fvTangent     = gl_NormalMatrix * rm_Tangent;
```

//normalize these vectors into univ vectors

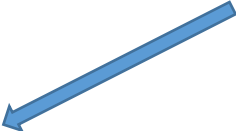
... ..

//2. Construct transformation matrix:

```
mat3 View2Tangent = mat3(fvTangent, fvBinormal, fvNormal);  
View2Tangent=transpose(View2Tangent);
```

//3. Transform Normal, Binormal and Tangent vectors into tangent space:

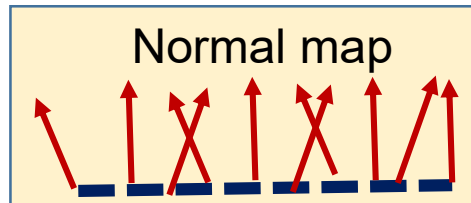
```
ViewDirInTangent = View2Tangent * ViewDir;  
LightDirInTangent = View2Tangent * LightDir;
```


$$\begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{pmatrix}$$

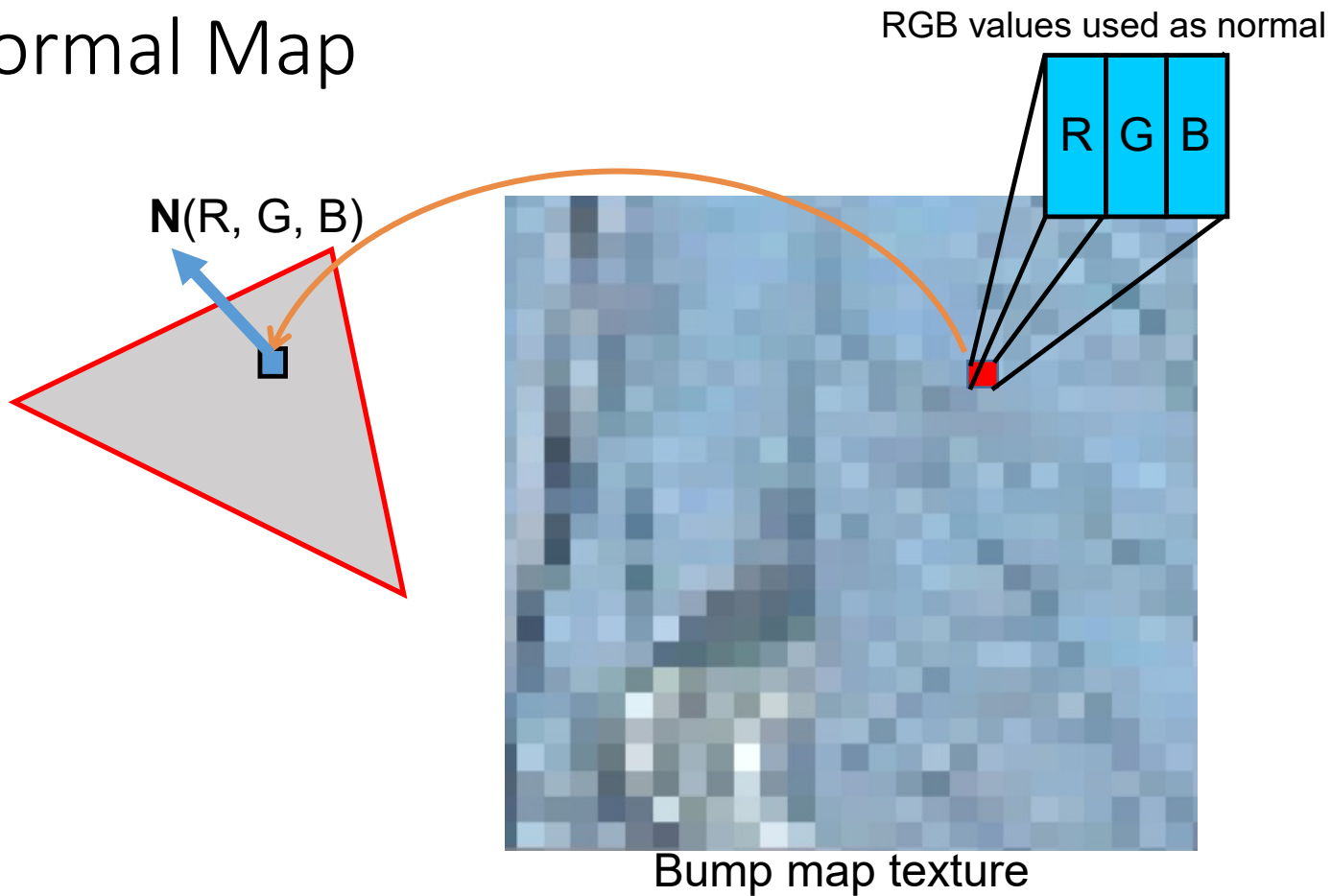
Method 1: Normal Mapping

—Read normal from a normal map

- **Normal maps** are images that store surface normals directly in RGB values
 - The RGB values of each texel in the normal map represent the x, y, z components of the normalized normal vector at the vertex associated with the texel
- Instead of using normal vectors from the geometric model to compute the light colour, the 3D vectors from the normal map are used



Normal Map



How to Use the Normal Map

- Colour values in a texture are typically constrained to $[0, 1]$
- Since the components of a normal vector, when normalized, correspond to $[-1, 1]$, they must be compressed into $[0, 1]$ when they are stored in colour
- When a normal texture is used as a bump map, RGB values must be de-compressed back from $[0, 1]$ to $[-1, 1]$:

$$\text{Colour2Normal} = 2 * \text{Normal2Colour} - 1$$

Bumpy Effect Using a Normal Map

```
vec3 NormalAsCol = texture2D(bumpMap, Texcoord ).xyz;
```

```
vec3 NormalFrCol = normalize( 2.0 * NormalAsCol - 1.0 );
```



Bumpy Effect Using a Normal Map

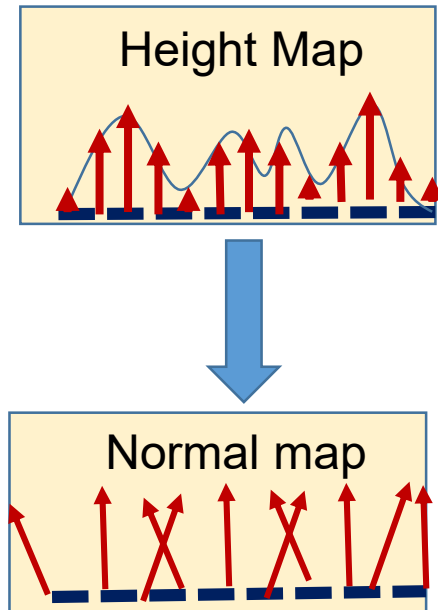
- The bump **density** can be controlled by scaling the texture coordinate and by specifying the texture wrapping mode as GL_REPEAT
- The significance of bump can also be enhanced by modifying the components of the normal vector



Method 2: Height Mapping

— Computing normal from a height map

- Normal mapping relies on a pre-made normal map, using a certain texture tool
- Height map
 - A texture used to store the surface height information
 - Normally created in the tangent space
 - Normal vectors at each pixel is then calculated from the height map
 - Implemented similarly to normal mapping
- More flexible than normal-map, allowing using any texture to create surface bumps



Height Mapping

— Computing normal from a height map

- Use a height map to represent the height function

```
float Height = texture2D( heightMap, Texcoord).r;
```

- Find the partial derivatives of the height function along x-axis and y-axis:

```
float dx= dFdx(Height);
```

```
float dy= dFdy(Height);
```

```
vec3 Normal = normalize( vec3(-dx, -dy, 1.0)
```

or

```
vec3 Normal = normalize( vec3(-dx, -dy, bumpH) );
```

A non-negative
value for enhancing
the bump details

Height Mapping

— Computing normal from a height map

- A better way of computing dx, dy:

```
float Height = texture2D( HeightMap, Texcoord).r;  
vec2 uvX= Texcoord, uvY=Texcoord;  
uvX.x -=pixelSizeX;  
uvY.y -=pixelSizeY;
```

```
float HeightX = texture2D( HeightMap, uvX).r;  
float HeightY = texture2D( HeightMap, uvY).r;
```

```
float dx= Height-HeightX;  
float dy= Height-HeightY;
```

Height Mapping

— Computing normal from a height map

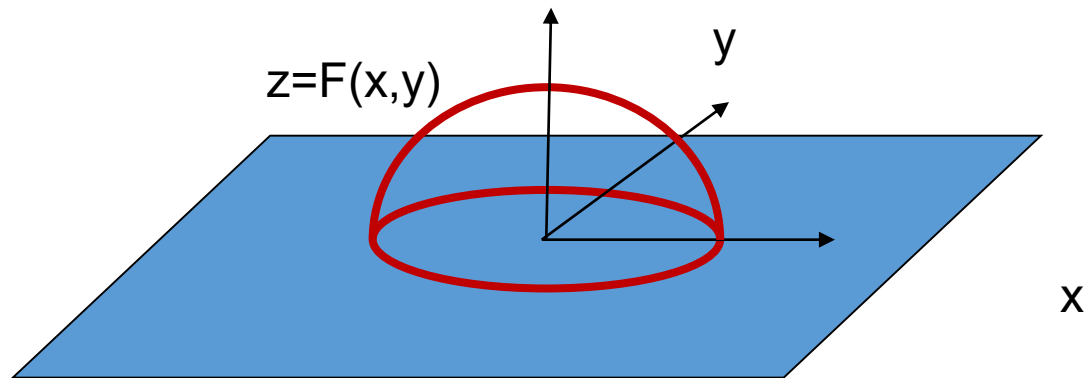


Method 3: Procedural Bump Mapping

— GLSL implementation: an example

- Define surface detail function $F(x,y)$ as:

$$F(x, y) = \begin{cases} \sqrt{r^2 - (x^2 + y^2)}, & x^2 + y^2 < r^2 \\ 0, & \text{otherwise} \end{cases}$$



Procedural Bump Mapping

— GLSL implementation: an example

The normal vector at (x, y, z) for the surface defined by $z=F(x, y)$ can be found by calculating the partial derivatives along x and y respectively:

$$\mathbf{N} = \left(-\frac{\partial F}{\partial x}, -\frac{\partial F}{\partial y}, 1 \right)$$

where

$$\frac{\partial F}{\partial x} = \begin{cases} -\frac{x}{\sqrt{r^2 - (x^2 + y^2)}}, & x^2 + y^2 < r^2 \\ 0, & \text{otherwise} \end{cases}$$

$$\frac{\partial F}{\partial y} = \begin{cases} -\frac{y}{\sqrt{r^2 - (x^2 + y^2)}}, & x^2 + y^2 < r^2 \\ 0, & \text{otherwise} \end{cases}$$

Fragment Shader

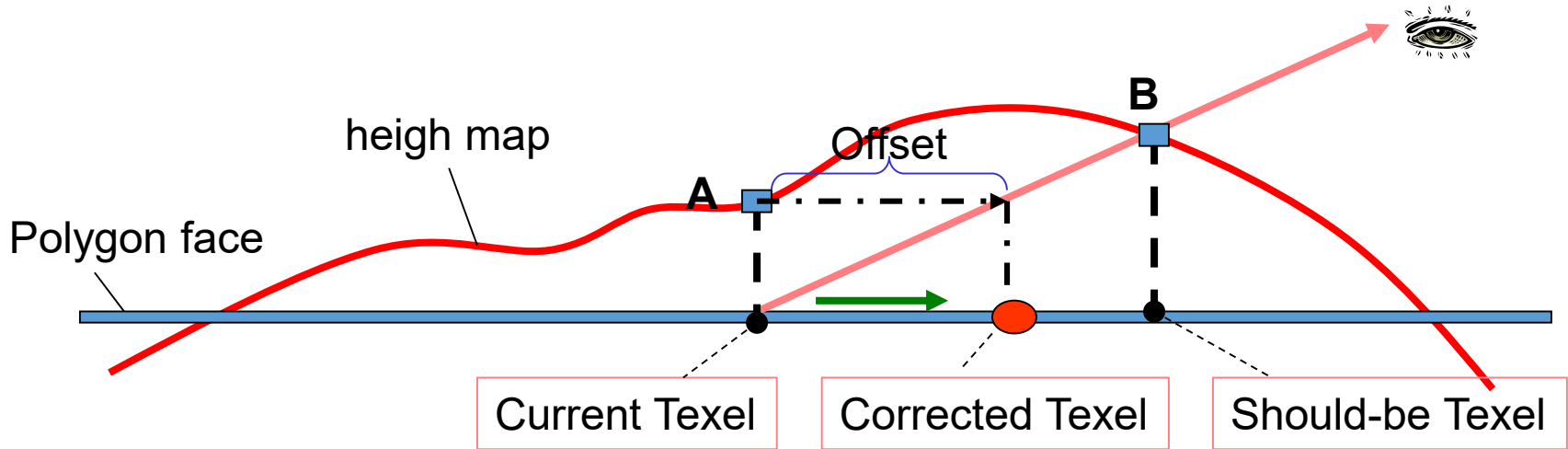
```
vec3 bumpNormal( vec2 xy) {  
    vec3 N=vec3(0.0, 0.0, 1.0);  
    //map xy to [-1, 1]x[-1, 1]:  
    vec2 st=2.0*fract(xy) -1.0;  
    float R2=radius*radius - dot(st,st);  
    if (R2>0.0)  
        N.xy=st/sqrt(R2);  
  
    return normalize(N);  
}
```

```
void main( void )  
{  
    ... ..  
    float    x = BumpDensity * Tex.x,  
             y =BumpDensity * Tex.y;  
  
    vec3 Normal = bumpNormal(x, y);  
  
    ... ..  
}
```

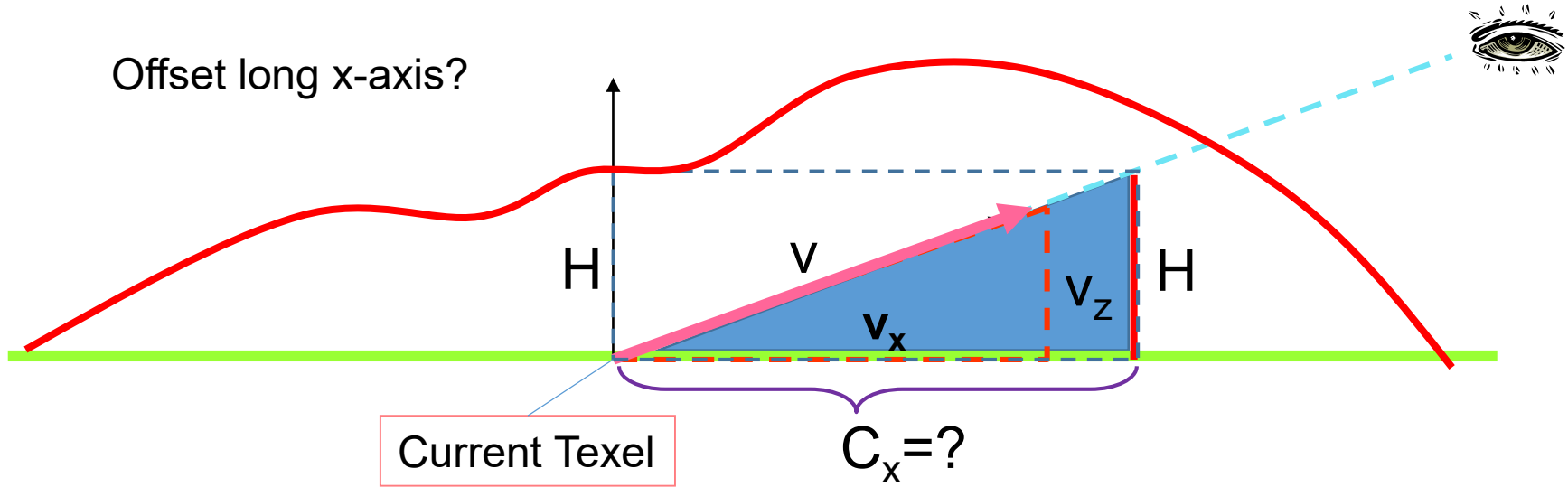


Method 4: Parallax Mapping

- Q: How can we improve traditional bump mapping such that what we see is close to the real situation?
- Solution:
 - View ray tracing and using the height map to approximate the should-be texture coordinates

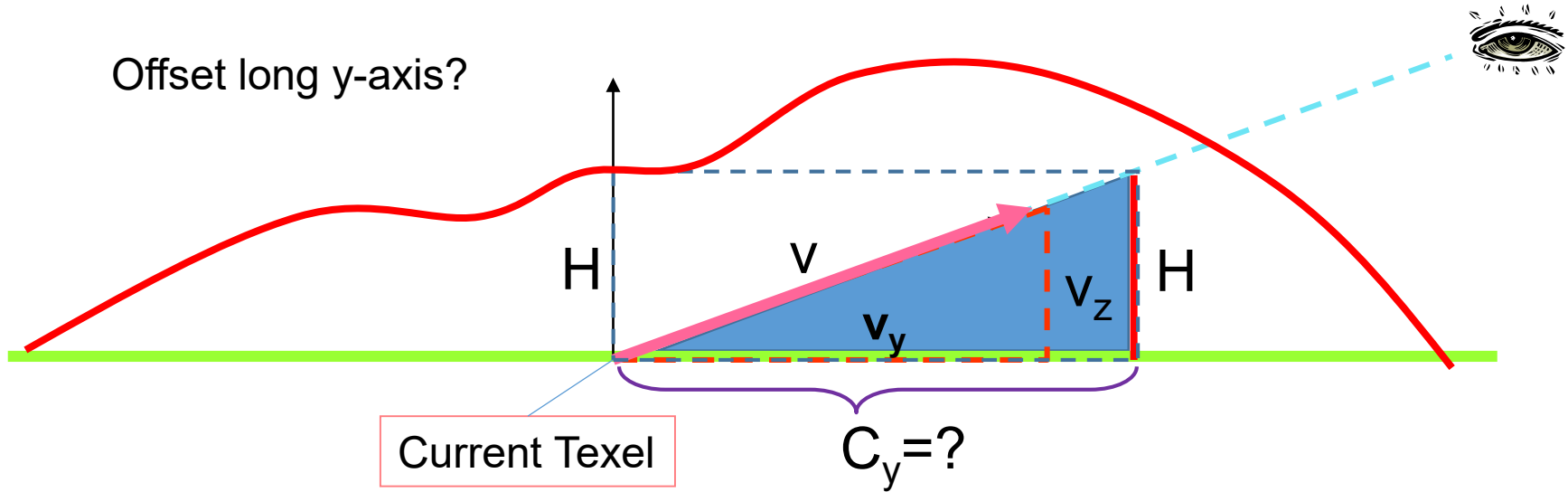


How to Find the Offset for a Height Map?



$$\frac{H}{C_x} = \frac{v_z}{v_x} \Rightarrow C_x = \frac{v_x}{v_z} H$$

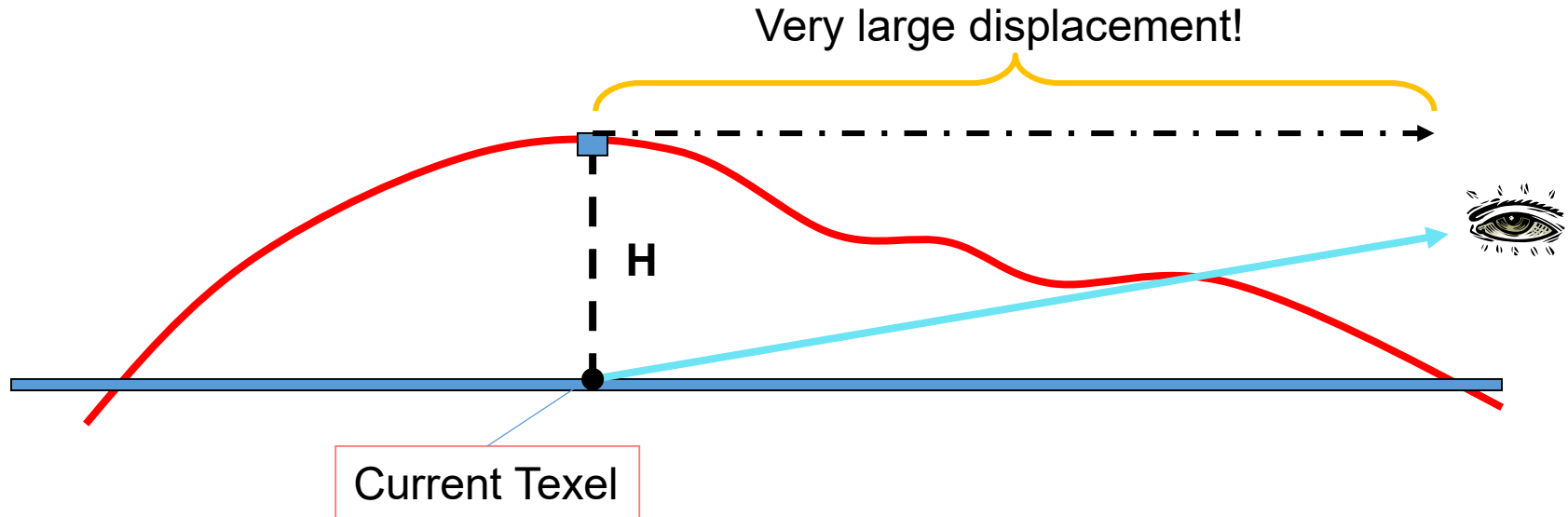
How to Find the Offset for a Height Map?



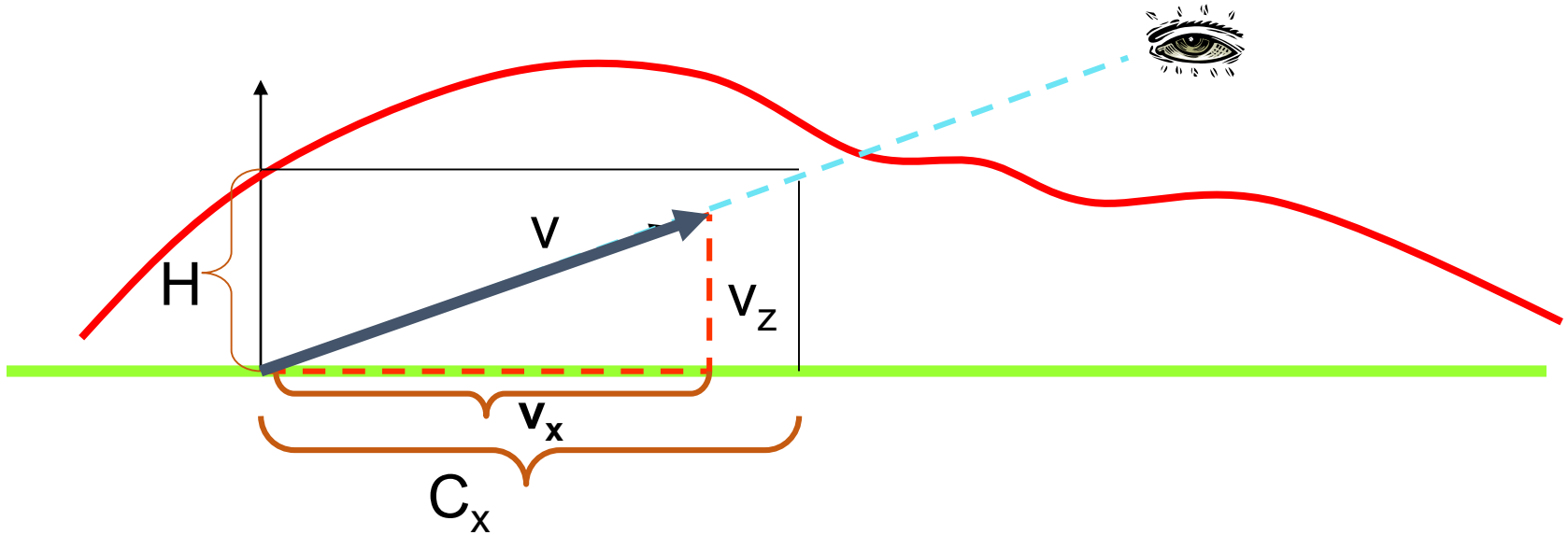
$$\frac{H}{C_y} = \frac{v_z}{v_y} \rightarrow C_y = \frac{v_y}{v_z} H$$

Problem?

- As the viewing angle becomes more shallow or when current height value is relatively large, offset values can be very large!



A Bad But Simple Solution: Set $V_z=1$!



$$C_x = \frac{v_x}{v_z} H$$

$$C_y = \frac{v_y}{v_z} H$$



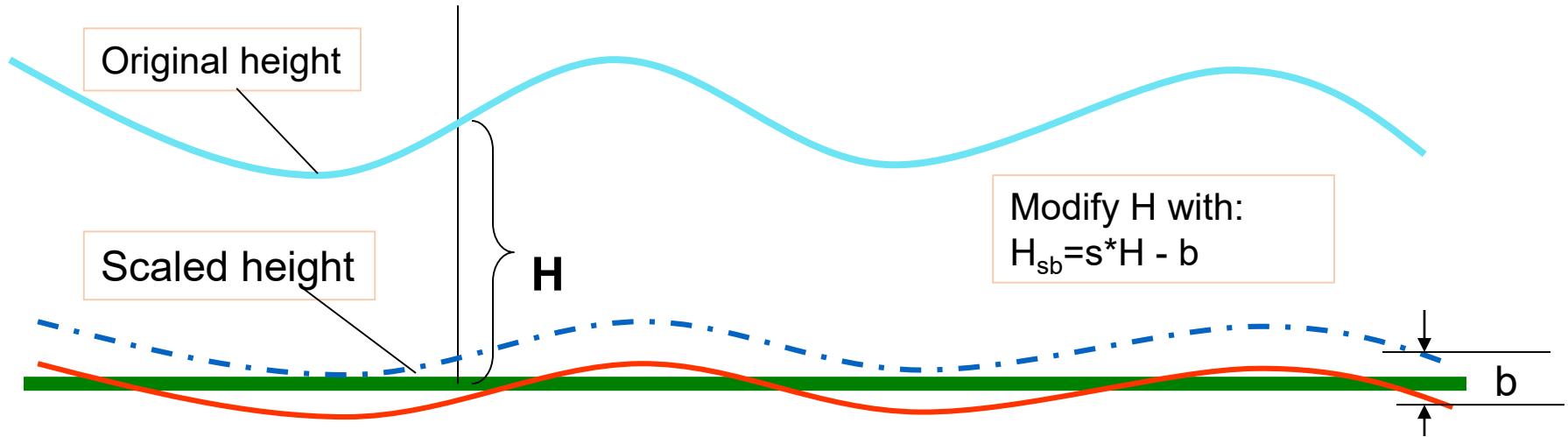
$$\begin{aligned} C_x &= H \times v_x \\ C_y &= H \times v_y \end{aligned}$$

or

$$\mathbf{C.xy} = H \times \mathbf{V.xy}$$

Use Modified Height

- Height value **H** from a height map may need to be properly scaled and biased to account for the surface being simulated



Pixel Shader: Input

... ..

```
uniform sampler2D ColorMap;  
uniform sampler2D NormalMap;  
uniform sampler2D HeightMap;
```

```
varying vec2 Texcoord;  
varying vec3 ViewDirection;  
varying vec3 LightDirection;
```

```
uniform float scale;  
uniform float bias0;
```

Pixel Shader: Main()

```
vec3 L = normalize( LightDirection );
```

```
vec3 V = normalize( ViewDirection );
```

```
// read height from a height map
```

```
float Height = texture2D( HeightMap, Texcoord ).x;
```

```
// modify height value
```

```
Height = scale* Height - bias0;
```

```
// Correct the texture coordinates
```

```
vec2 TexCorrected = Texcoord  
                    + Height * V.xy;
```

Pixel Shader (cont.)

```
//fetch texture color
```

```
vec4 BaseColor =
```

```
    texture2D( ColorMap, TexCorrected );
```

```
// fetch normal vector
```

```
vec3 Normal =2.0 * texture2D( BumpMap, TexCorrected ) - 1.0;
```

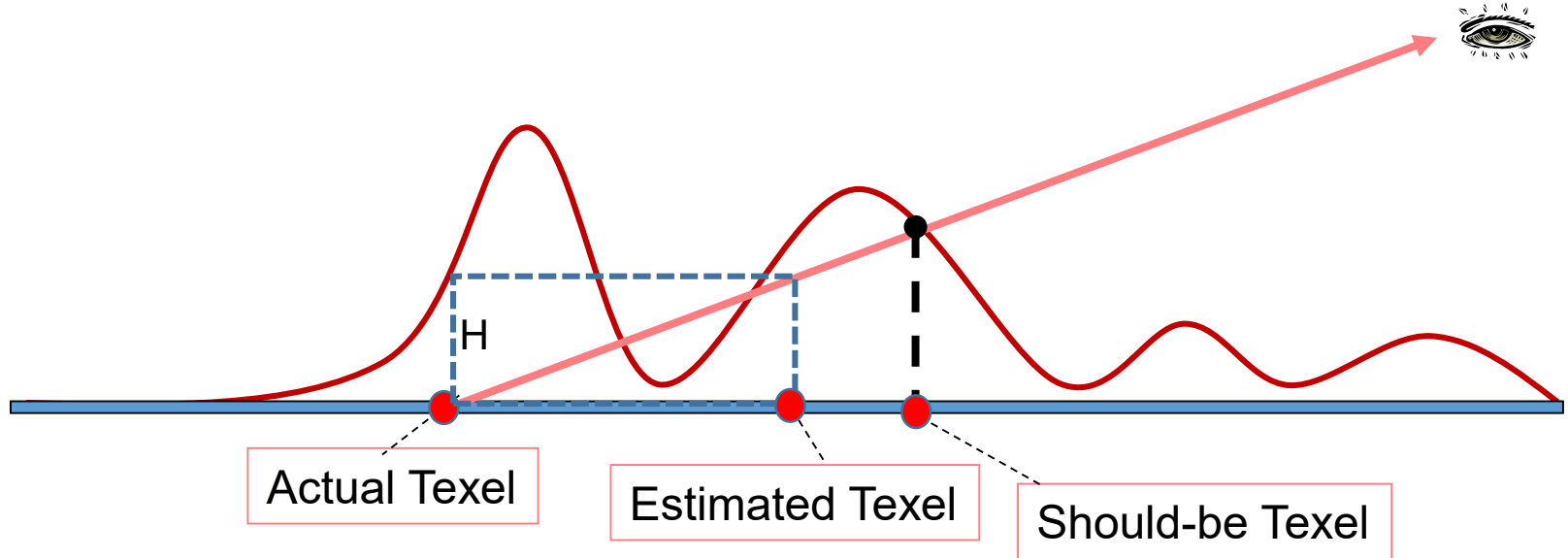
```
// Compute light colour
```

```
... ..
```



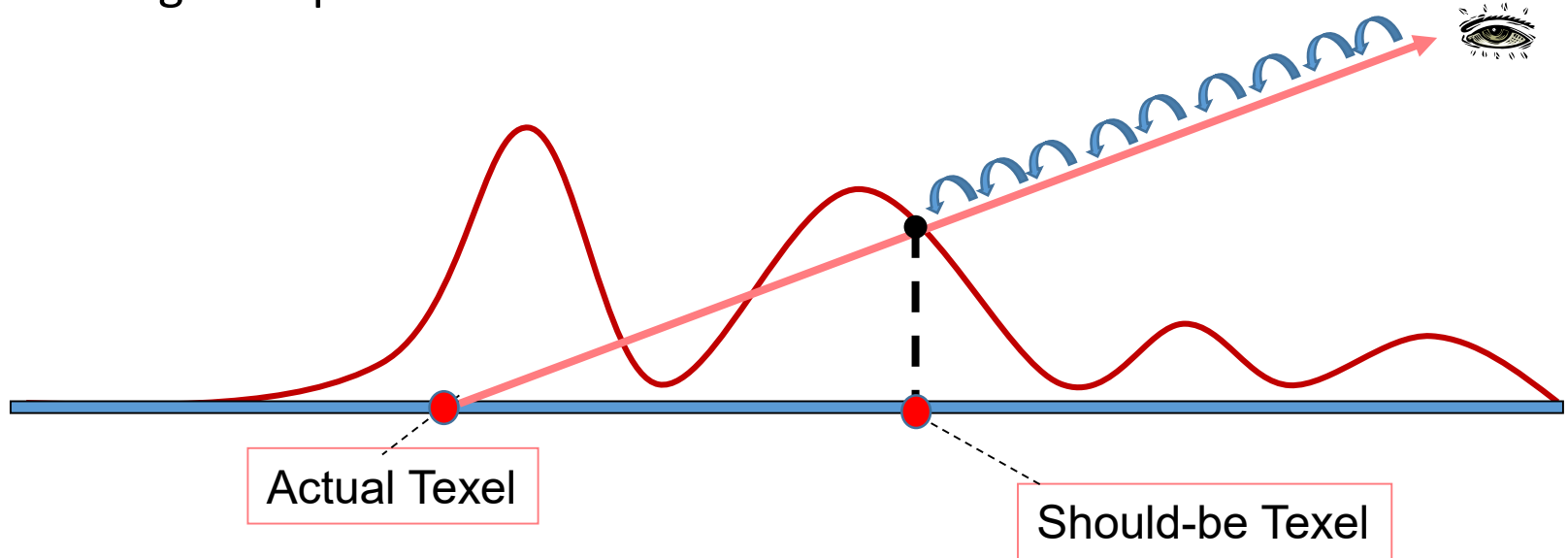
Ray marching-based parallax mapping

- The above one-step parallax mapping technique can be very inaccurate when the **height value** used for the texture coordinate correction is relatively big.



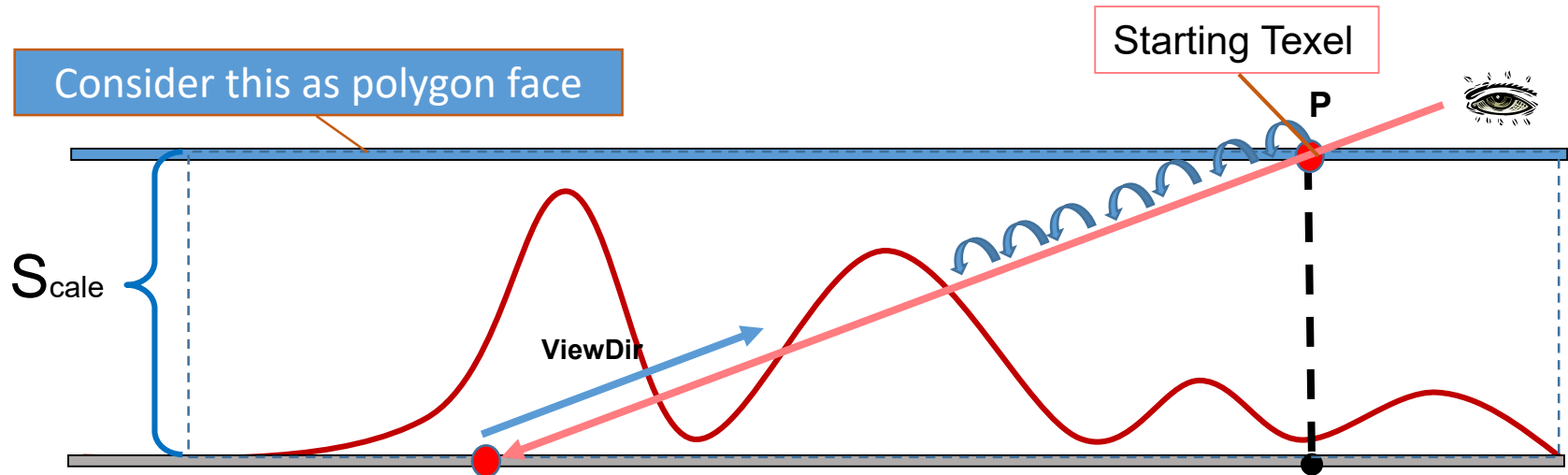
Ray marching-based parallax mapping

- Better solution can be achieved by finding the exact texture coordinates corresponding to the **nearest hit** between view ray and the height map.



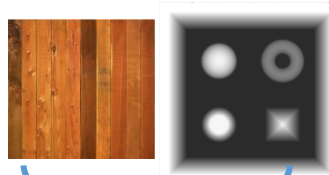
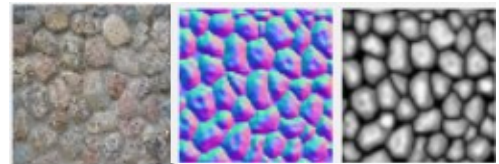
How to implement?

- Thinking the bumpy surface is under the surface!
- Then the current texture coordinates corresponding to **P**.
- Marching along the view ray from **P** based on a given marching step size until a point, at which the height map value is smaller than the **view Ray** height.



How to implement?

```
vec2 ParallaxMapping(vec2 texCoords, vec3 viewDir)
{
    float currentDepth = Scale;
    vec3 rayDir = - normalize(viewDir.xyz);
    vec2 currentTexCoords = texCoords;
    float currentDepthMapValue = Scale * texture2D(heightMap, currentTexCoords).x;
    while(currentDepth > currentDepthMapValue)
    {
        currentTexCoords += stepSize*V.xy/Vz ;
        currentDepth -= stepSize;
        currentDepthMapValue = Scale * texture2D(heightMap, currentTexCoords).x;
    }
    return currentTexCoords;
}
```



Questions?