Master – Computer Science
University Paris-Sud

Open GL & GLSL Course

GLSL Tutorial
& Mapping for Relief and Reflection

Christian Jacquemin

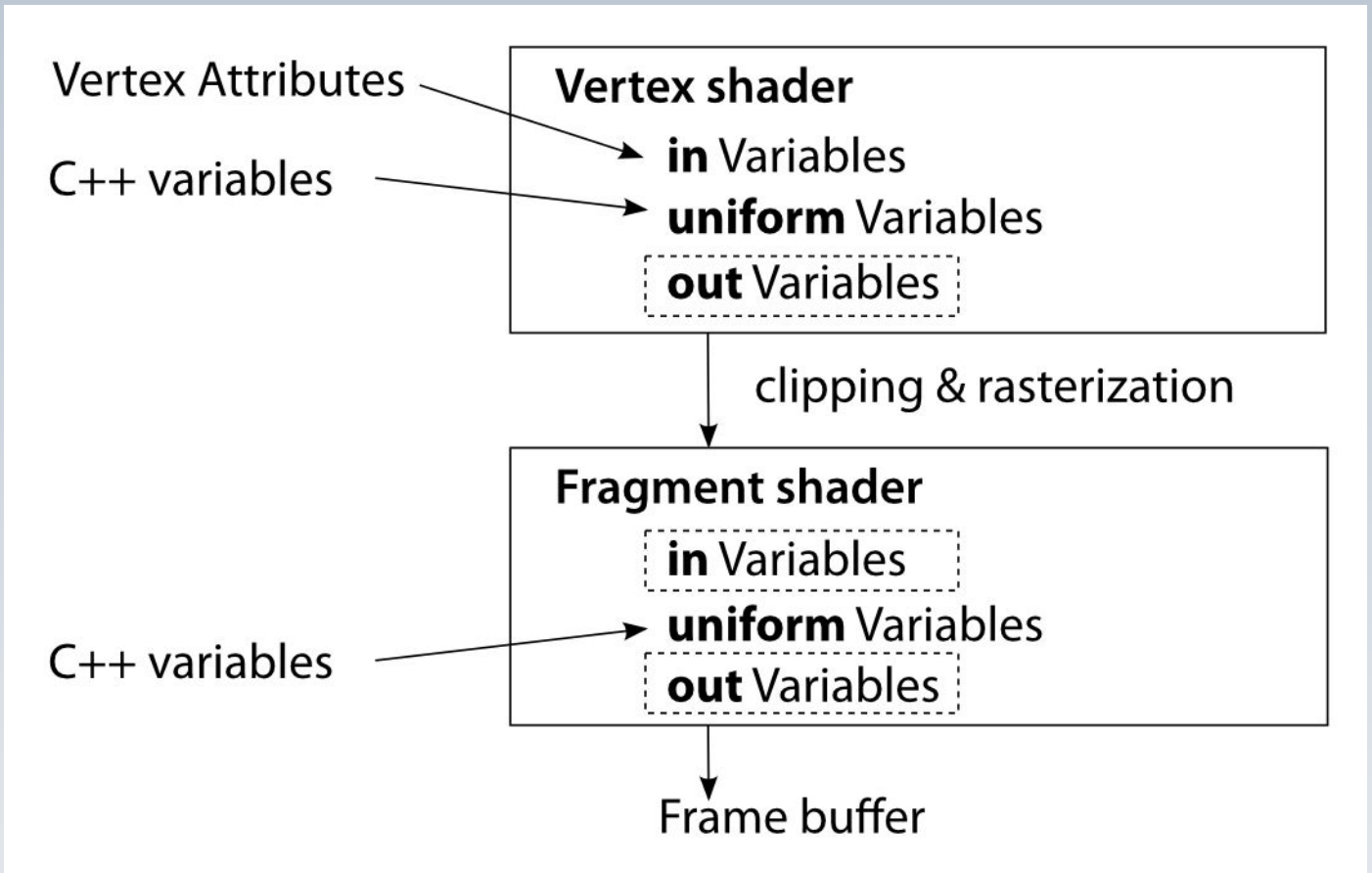
GLSL Tutorial & Relief Techniques

(I)

GLSL Tutorial

Graphic Pipe-line

vertex & **fragment shader** pipe-line



Data Types

- **elementary** data types
  - float, double, int, uint, bool
- **arrays**
  - dimension: 2, 3, or 4
  - vec, dvec, ivec, uvec, bvec
- **matrices** of floats or doubles
  - square matrices: size 2x2, 3x3, or 4x4: mat3, dmat4
  - non-square matrices: size {2,3,4}x{2,3,4}:
    mat3x4, dmat4x2
- **texture samplers**
  - sampler1D, sampler2D, sampler3D for 1, 2, 3D power-of-two textures
  - sampler2DRect for rectangle textures
  - samplerCube for cubemap textures

Data Types

- **variable declaration and initialization** very similar to C-style
  - `float a = 1.0;`
  - `vec3 point = vec3(1.0,1.0,4.0);`
  - `mat2 mat = mat2(1.0,0.0,0.0,1.0);`
    (matrices are assigned in column major order)
- **abbreviated** initialization
  - Diagonal matrix: `ident = mat4(1.0);`
  - Array with same values: `null_vec = vec3(0.0);`
- **composite** initialization
  - `vec4 point_hom = vec4(point.xy, 0.0, 1.0);`
- **constant** values
  - `#define height 1.0`

Swizzling

array components are accessed through **xyzw** or **rgba** (but xyzw and rgba have no semantics)

```
vec4 color=vec4(0.3,0.5,0.1,0.5);

color.rgb = color.rrr;  // gray level from red

color = vec4(color.rgb,1.0); // opaque color


mat4 transf;

vec3 point;

// conversion into homogeneous coordinates and
// back to Cartesian coordinates

vec3 transf_point = (transf * vec4(point,1.0)).xyz;
```

Syntax – Main Control Instructions

- usual C syntax: if/else, for, while, do..while + break and continue
- **discard** instruction inside fragment shader: does not output in the frame and/or depth buffer
- main() is the main function
  + possibility of typed subroutines with parameters
- type conversion is used with functions, not with C-style cast

0)

Variables

- **attribute** variables

```
layout(location = 0) in vec3 vp;
// 1st param location in glVertexAttribPointer
```

- **uniform** variables

```
uniform mat4 projectionMatrix;
```

- **out** variables

```
out vec3 lightOut;
```

- **local** and **global** variables
- some **built-in** variables
  - vertex shader in: `int gl_VertexID (for indexed rendering)`

    vertex shader out: `vec4 gl_Position (clip space position),`
    `float gl_PointSize (pixel width/height for point rendering)`
  - fragment shader in: `vec4 gl_FragCoord (location in window space)`

    fragment shader out: `float gl_FragDepth (fragment depth)`

GLSL Tutorial & Relief Techniques

(II)

Relief Techniques

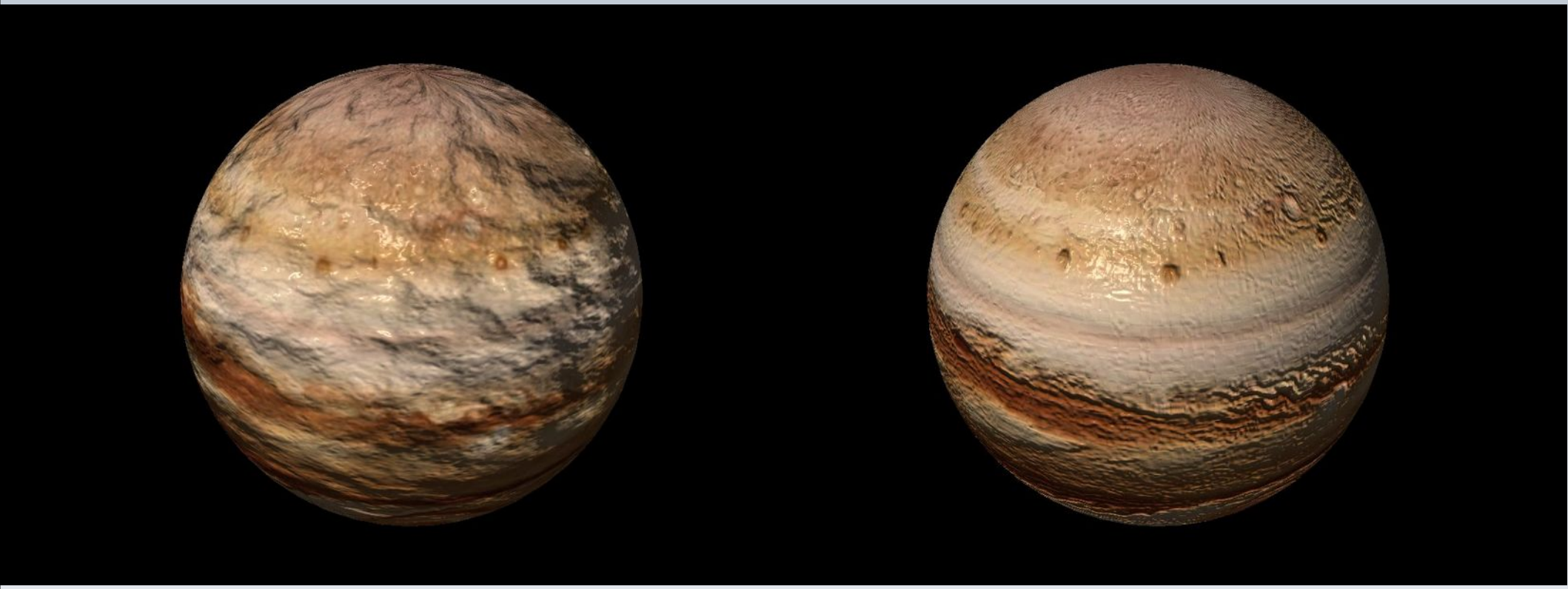
Bump- & Normal-Mapping Principles

- A perceptually satisfying mesh-less rendering of bumpy surfaces through **normal displacement** (and variations of diffuse & specular light intensity).
- Basic bump-mapping involves **neither geometry, nor pixel displacement** (whether for the geometry or for the image).
- **Bump-mapping**: the normal displacements are directly computed from height field
- **Normal-mapping**: the normals are loaded from a normal map (and are precomputed)

Bump & Normal Mapping Examples

Normal Map

Bump Map



Bump- & Normal-Mapping Pros and Cons

- BENEFITS
  - easy to implement
  - does not require complex geometrical description
  - can be detailed if texture is HD
  - works with both **diffuse and specular lighting**
  - the height field can be embedded in the alpha channel of the texture

Bump- & Normal-Mapping Pros and Cons

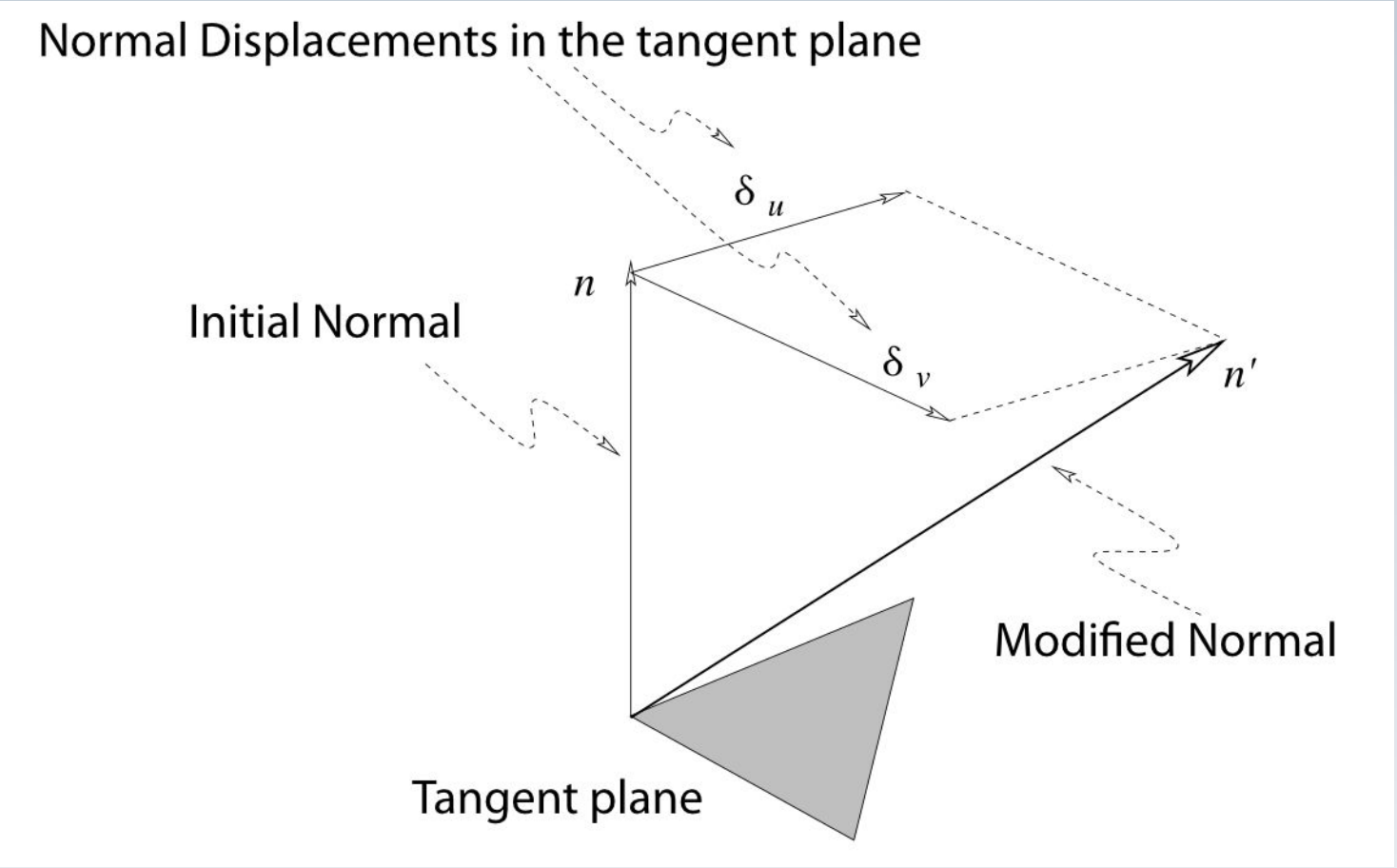
- LIMITS
  - the contours of the objects have no **relief** and the **silhouette** is not modified
  - there is no **self-shadowing**
  - there are no **parallax** effects
  - there cannot be **holes**
  - possible **aliasing** when coming close

→ Should be used for small heights, with a high definition texture

Normal Displacement Storage

Two storage techniques:

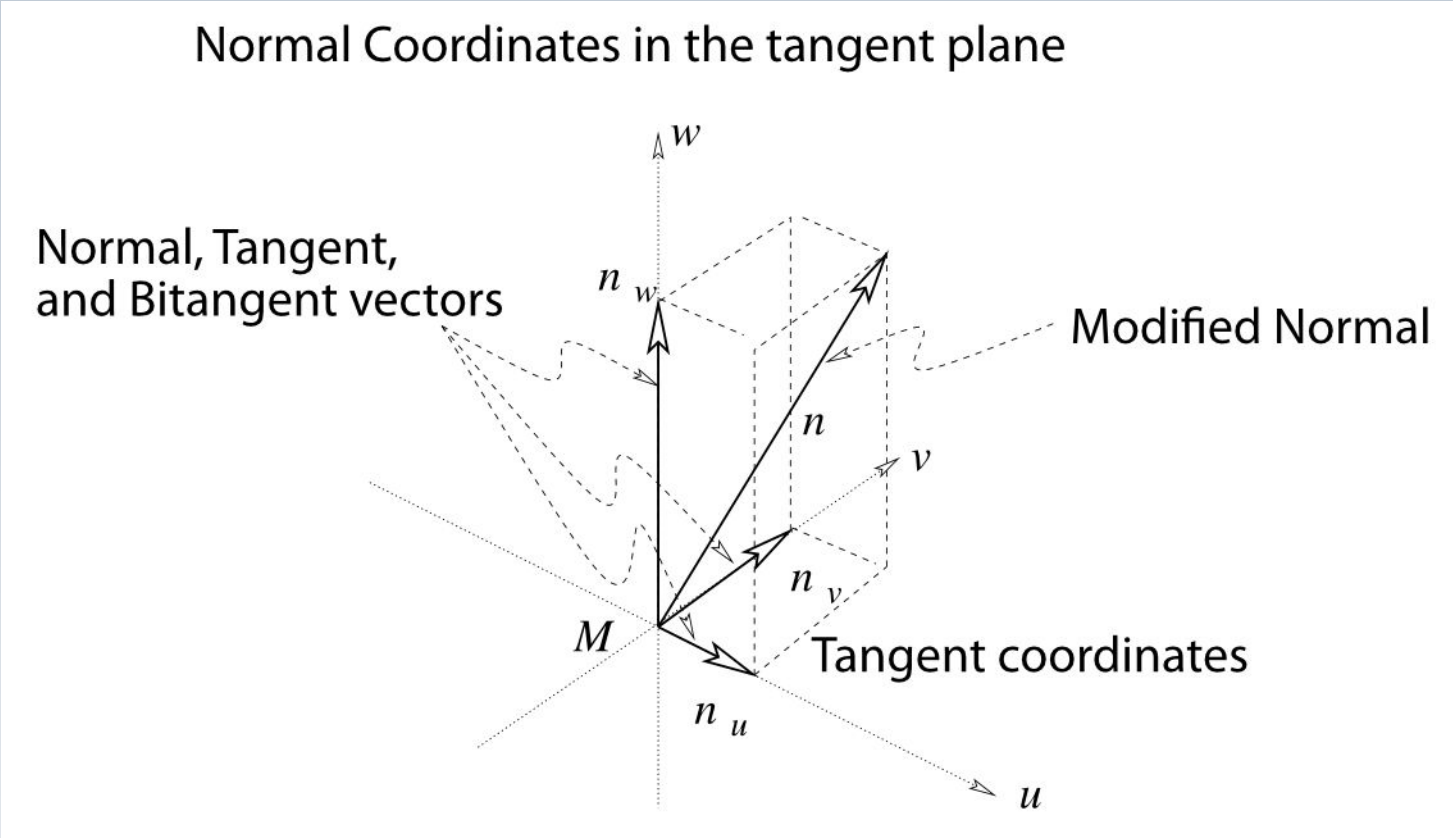
- either by storing two lateral displacements in the tangent plane (**offset map**)



Normal Displacement Storage

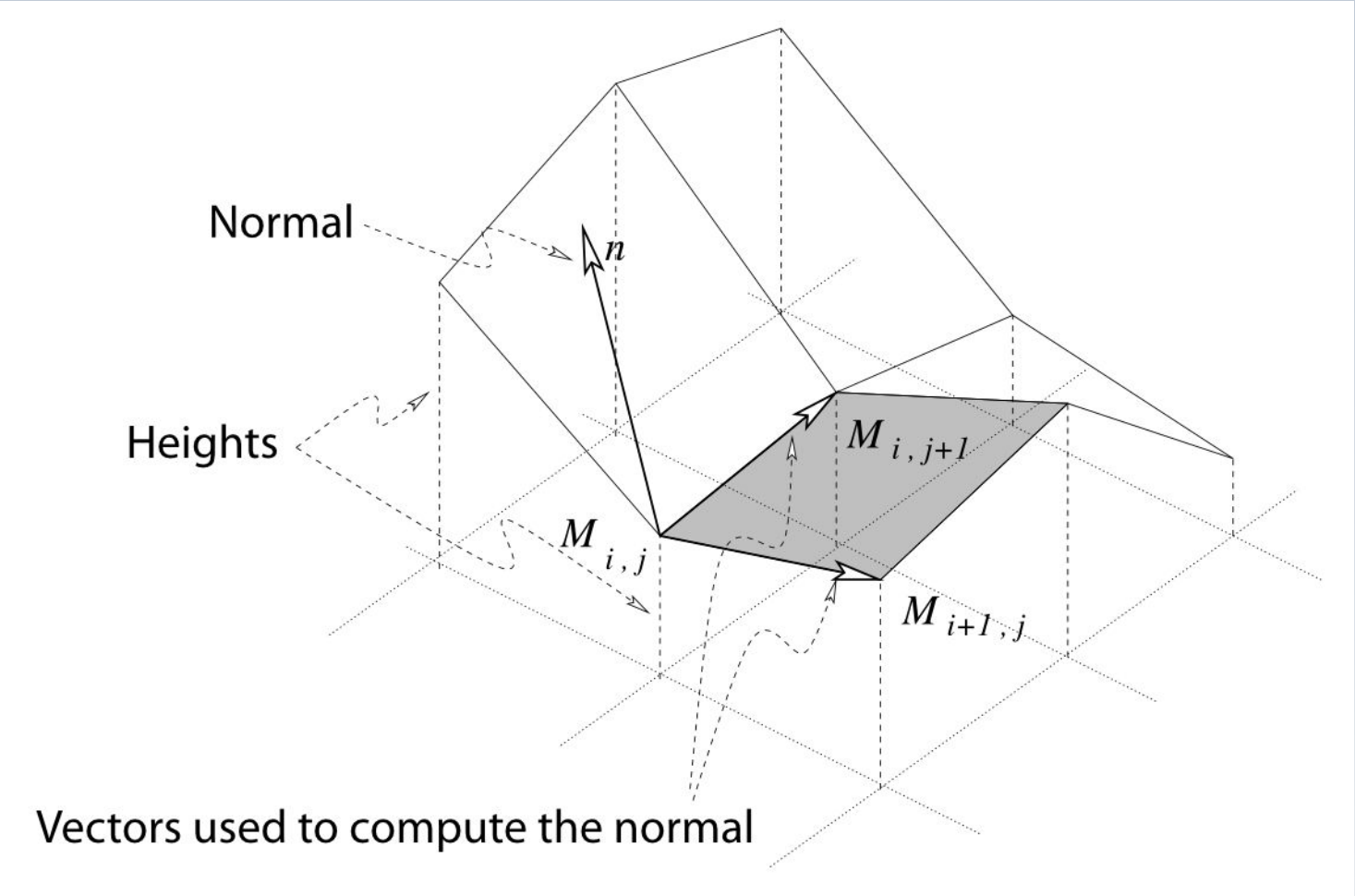
Two storage techniques:

- or by storing directly the normal coordinates



Normal Computation

Normal displacement or coordinates can be computed from a height map (a texture containing the height of each pixel in a one-channel color)



Normal Computation

- The displaced normals can be **precomputed** and stored in an **offset or a normal map**
  - for storage, the values have to be converted into positive values through linear transformation
  - the inverse computation should be made at lookup
- They can be **dynamically** computed from a **height field**
  - the computation should be made in the fragment shader for a finer resolution
    (the resolution of the image is generally higher than the one of the geometry)
  - **a scaling factor** should be used on the height displacements

Light Computation

- The standard formulas for diffuse and specular lighting are used with the displaced normal values
  - all the computations are to be made in a coherent system
  - the bump normals are displaced in the **tangent coordinate system**,
  - we choose to make all the computations in the **world coordinate system:**
    - light → light source given in world coordinates
    - view → origin of view coordinates transformed into world coordinates through inverse view matrix
    - bump normal is computed in **tangent coordinates** and projected on the (tangent, binormal, normal) given in **world coordinates**

Shader Implementation

- Vertex Shader
  - transformation of view origin into world coordinates
  - transformation of tangent and normal vector from model coordinate system to world coordinates
- Fragment Shader
  - texture look-up for normal displacement (in case of an offset map)
  - or normal displacement computation after 3 height values lookup (one locally and the two other in neighboring texels)
  - transformation of displaced normals into world coordinates by projecting it on the (tangent, binormal, normal) received in **world coordinates**
  - lighting computation from displaced normals

Vertex Shader Implementation

- Vertex Shader

```
// normal and tangent transformed into world
// coordinates
normalOut = (modelMatrix * vec4(norm,0.0)).xyz;
tangentOut = (modelMatrix * vec4(tangent,0.0)).xyz;

// point position transformed from model to world
// coordinates
posWorld = (modelMatrix * vec4(vp,1.0)).xyz;

// light vector in the world coordinates
lightOut = posWorld – lightPosition;

// eye is at the origin in the view coordinates
// view vector in the world space
viewOut = (inverse(viewMatrix) *
           vec4(vec3(0.0),1.0)).xyz – posWorld;
```

Fragment Shader Implementation

- Fragment Shader: displaced normals computation

```
// (1): displacement map texture lookup
normal = texture( displacementMap , texCoordOut ).rgb;
// Transforms normal coordinates from [0..1] to [-1..1]
normal.r = normal.r – 0.5;
normal.g = normal.g – 0.5;
normal = normalize( -normal );

// (2): height map lookup on two neighboring pixels
float h = texture( heightMap, texCoordOut ).r;
float h1 = texture( heightMap,
                    texCoordOut + vec2(1.0,0.0) ).r;
float h2 = texture( heightMap,
                    texCoordOut + vec2(0.0,1.0) ).r;

// normal calculation from cross product
// with alpha as height scaling factor
normal = normalize(cross(vec3(1.0,0.0,alpha*(h1-h)),
                         vec3(0.0,1.0,alpha*(h2-h))));
```

Fragment Shader Implementation

- Fragment Shader: displaced normals in world coordinates

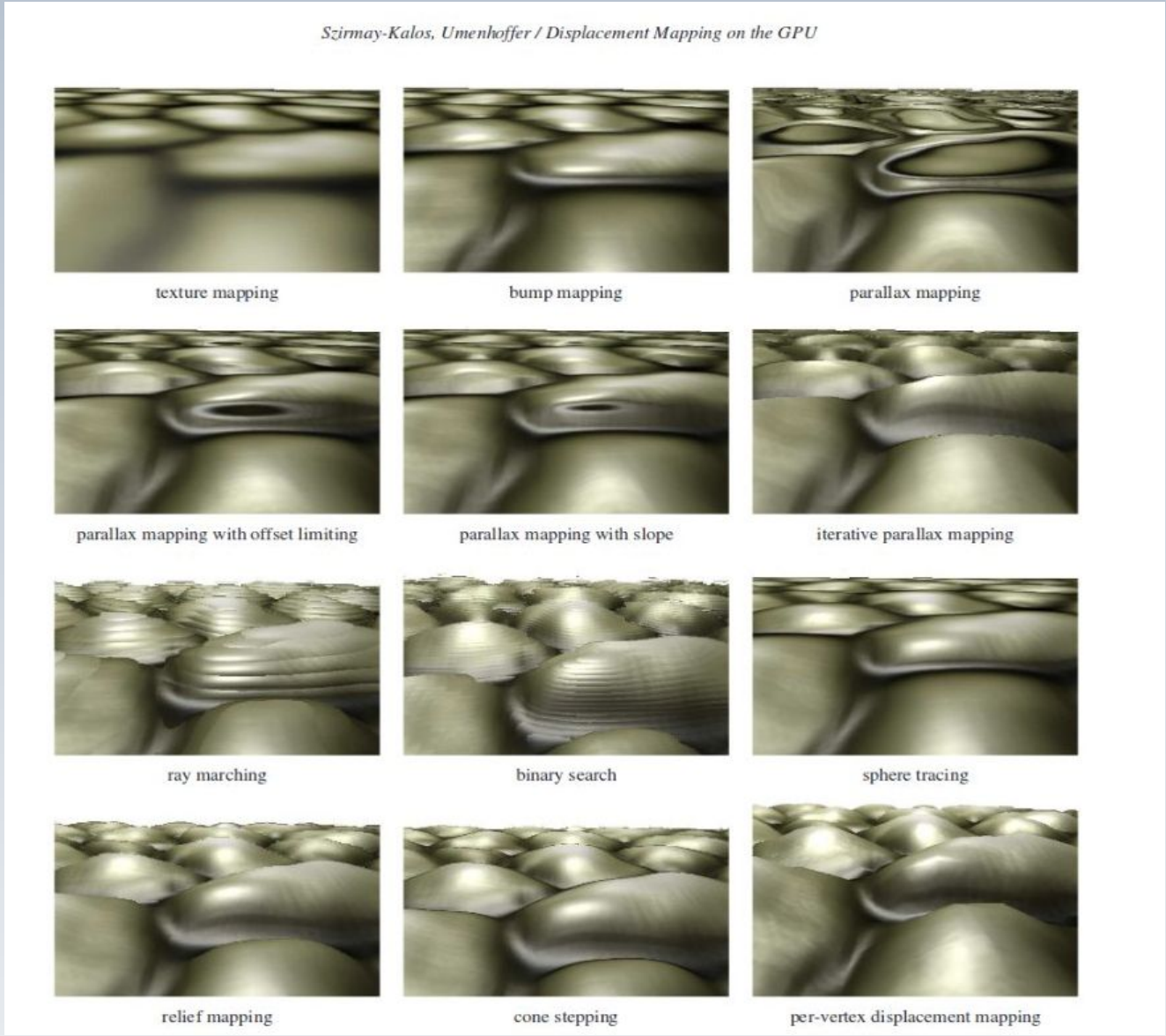
```
// binormal is the cross product of tangent
// and normal
vec3 binormal = cross( normalOut , tangentOut );

// bump normal change of coordinate system:
// from tangent to world coordinates by
// using the tangent and normal received
// from the fragment shader in world coordinates
vec3 bumpNormal = normal.x * tangentOut
                + normal.y * binormal
                + normal.z * normalOut;
```

Other Relief Techniques

comparison of different displacement mapping techniques (on the GPU)

Displacement Mapping on the GPU - State of the Art. László Szirmay-Kalos, and Tamás Umenhoffer. Comput. Graph. Forum 27(6):1567-1592 (2008)



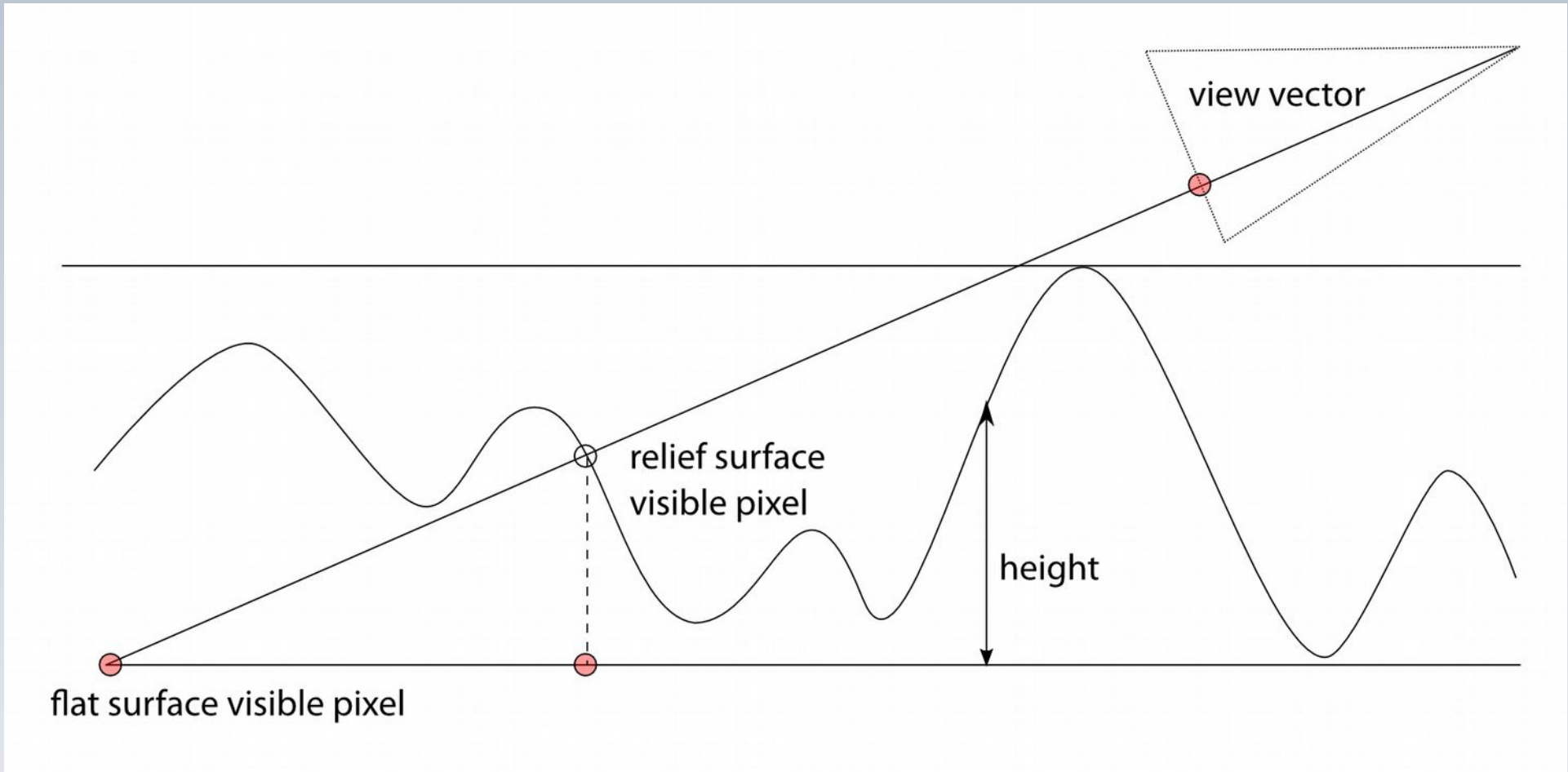
Other Relief Techniques

- same principle as Normal Mapping:
  - the micro structure is stored in a texture
  - the macro structure is stored in a mesh
- per-vertex displacement
  - the vertex shader displaces the vertices along normal
  - possibility to use geometry or tessellation shaders for finer geometrical resolution
- per-pixel displacement
  - through ray tracing, the visible pixel is accessed resulting in pixel displacement

    can only work for pixels “below” the surface level

Other Relief Techniques

- ray tracing for displacement mapping



Parallax Mapping

- displacement by fetching the pixel under: parallax mapping (the preceding figure)
- iterative intersection calculation through
  - binary search (dichotomy)
  - linear search
  - sphere tracing
  - cone stepping...
- silhouette processing with local deformation of the height field to get a smooth silhouette
- self-shadowing

GLSL Tutorial & Relief Techniques

(II)

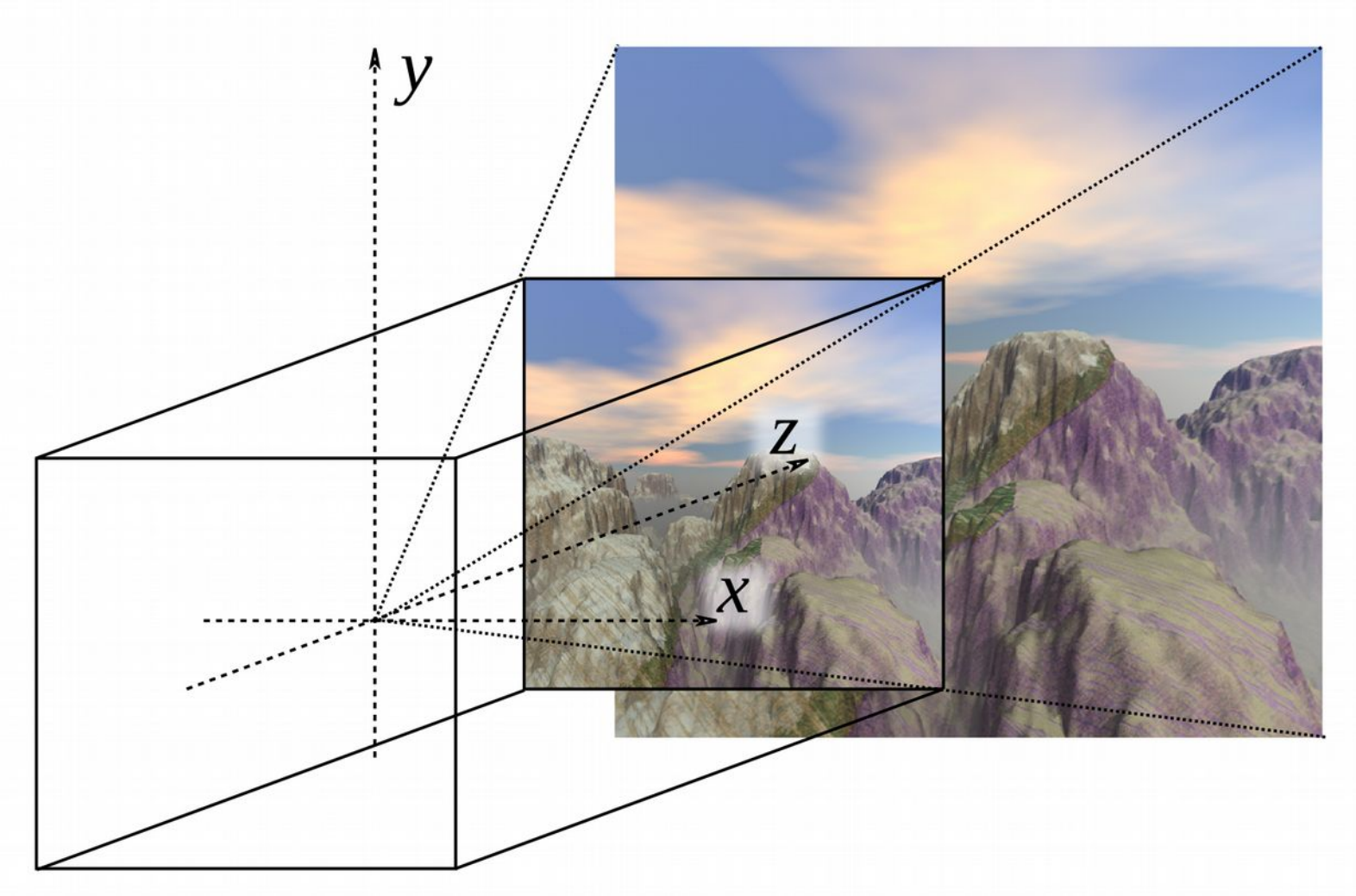
Environment Mapping

Environment Mapping

- rendering of **distant objects**
- **image-based rendering** (wo geometry)
- approximate rendering of
  - **transparency** (environment map)
  - **reflection** (environment map)
  - **lighting** (light map)
- the surfaces used to map the environment are
  - **cube**
  - **sphere**
  - cylinder...

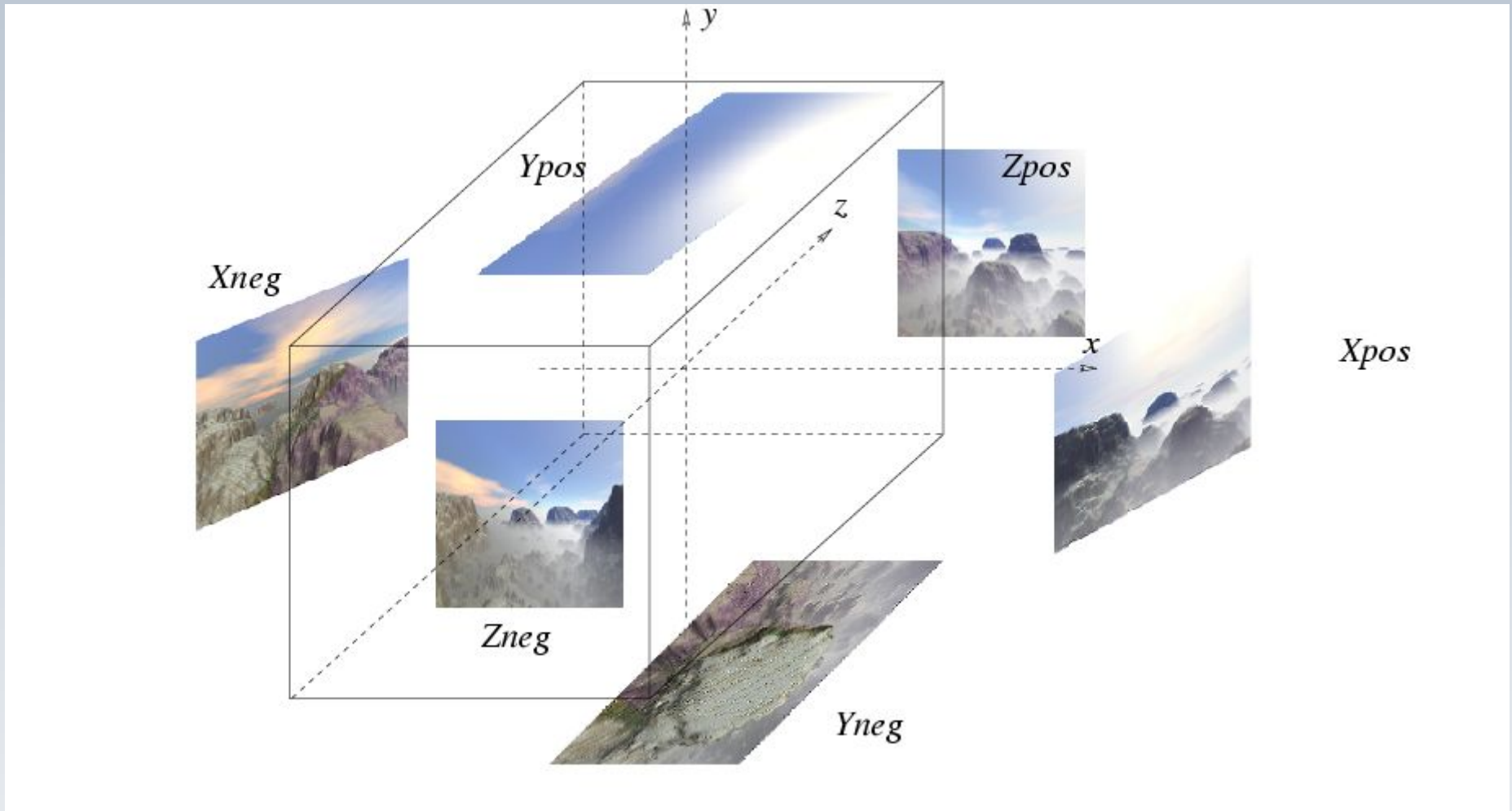
Cube Map Rendering

cube map acquisition (photos) or modeling (scene renderer)



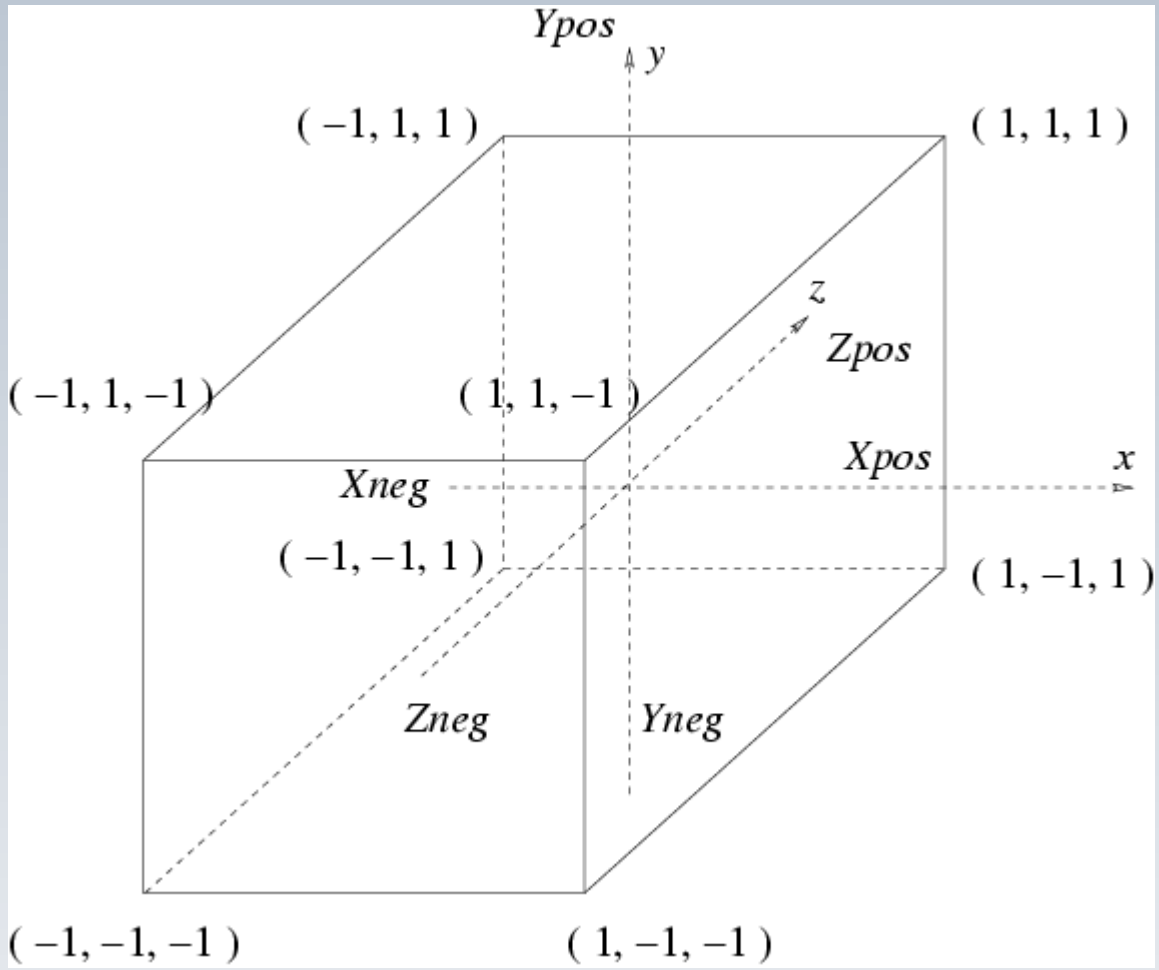
Cube Map Rendering

cube map made of 6 textures (here 3D modeling such as Bryce)



Cube Map Rendering

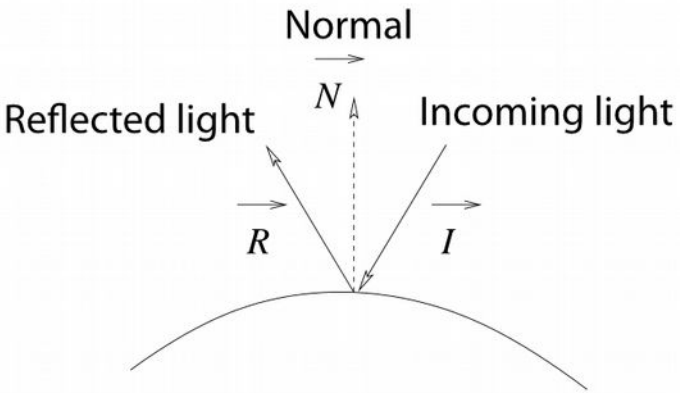
cube map normalized coordinates



Environment Map-based Reflection

instead of coming from a geometrical scene, the reflected light comes from its projection on a cube map.

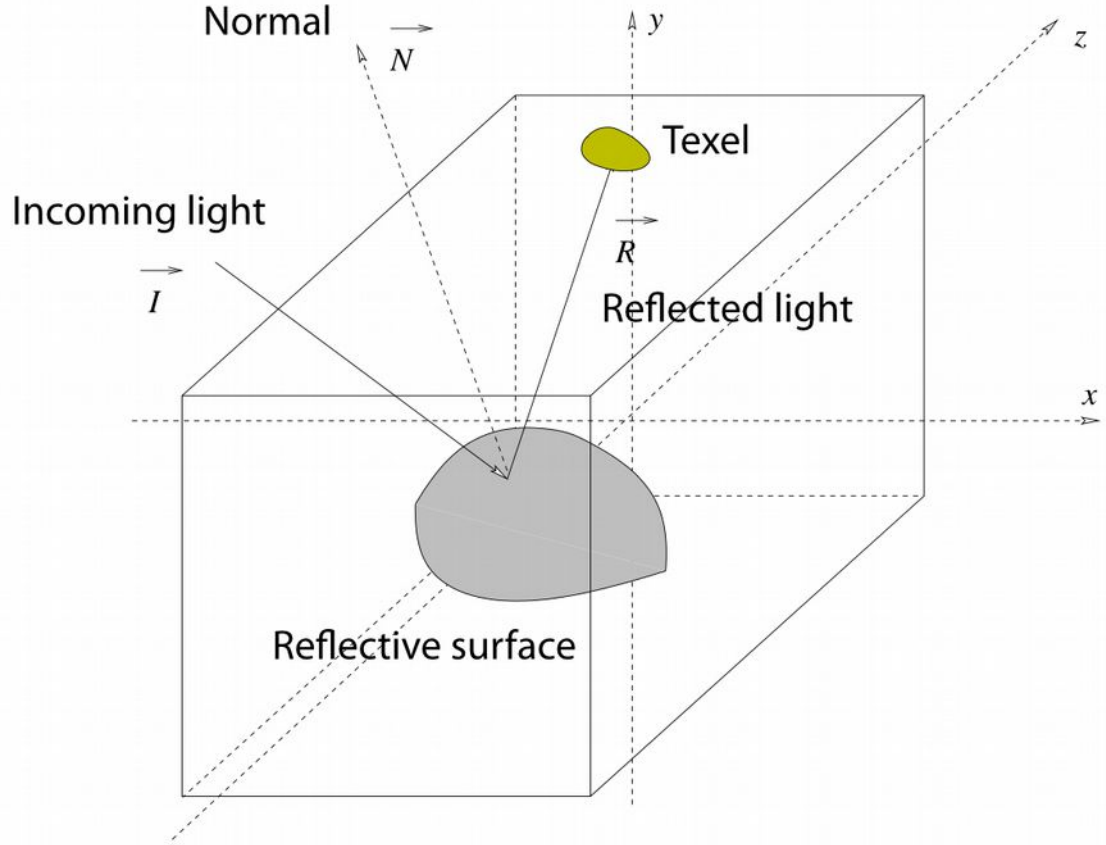
Light reflection



Reflection vector computation

If $\| N \| = 1$, $\quad R - I = 2\, N\, (N \,.\, I)$

Reflection simulation through environment cube map



Pros and Cons of Environment Map-based Reflection

- easy to implement
- low computation cost
- only works well for **distant objects** (the camera should not move significantly inside the cube or the textures should be dynamically updated)
- the distant environment must be **static**
- does not model **inter-reflections and self-reflection**
- the virtual scene cannot impact the environment (such as shadowing)

Implementation of Environment Map-based Reflection

- the cube map accesses the reflected texel through a **normalized reflection vector**
  - the reflection vector is the reflection of the view vector with respect to normal on each fragment

- the reflection vector must be computed in the **cube map coordinate system** (the cube map model transformation)

Example of Environment Map-based Reflection

Implementation of Environment Map-based Reflection

- cube map texture loading

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_EXT + i,
      0,                    //level
      GL_RGB,               //internal format
      imageCube[i].cols,    //width
      imageCube[i].rows,    //height
      0,                    //border
      GL_BGR,               //format
      GL_UNSIGNED_BYTE,     //type
      imageCube[i].ptr()); // pixel data
```

- cube map modeling:
  - regular mesh (only vertices)

Implementation of Environment Map-based Reflection

cube map rendering: cube with coordinates large enough to contain the scene (deactivate depth test)

- vertex shader
  - texture coordinates do not have to be provided (the 3D vertex coordinates are used as texture coordinates)

```
texCoordOut = vp;
```

- fragment shader
  - specific texture type

```
uniform samplerCube cubeMap;
```

  - texture lookup through 3D vector (the ray direction)

```
frag_colour = texture( cubeMap, texCoordOut);
```

Implementation of Environment Map-based Reflection

reflection rendering: compute reflection vector inside vertex shader in cube coordinate system

- vertex shader

```
// vertex in world coordinates
vec3 vertexWorld = (modelMatrix * vec4(vp,1.0)).xyz;
vec3 normalWorld = (modelMatrix * vec4(norm,0)).xyz;
vec3 VertexToEyeWorld = normalize(vertexWorld – eye);

// local light reflection vector in world coordinates
ViewReflectWorld =
   normalize(reflect(VertexToEyeWorld, normalWorld));
// reflection vector in cube map coordinates
ViewReflectWorld = (inverse(modelCubeMapMatrix)
   * vec4(ViewReflectWorld,0.0)).xyz;
```

Implementation of Environment Map-based Reflection

reflection rendering: fragment shader uses the reflection vector to access the texel in the cube map

- fragment shader

```
// reflection vector in cube map coordinates
in vec3 ViewReflectWorld;

// vertex in world coordinates
uniform sampler2DRect planeteSurface;

//Cubemap
vec4 color
   = texture(cubemap,
             normalize(ViewReflectWorld));
```