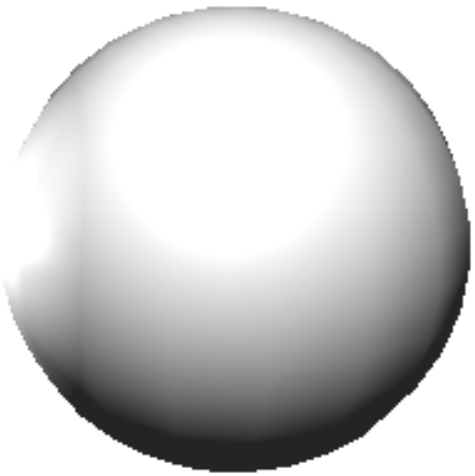
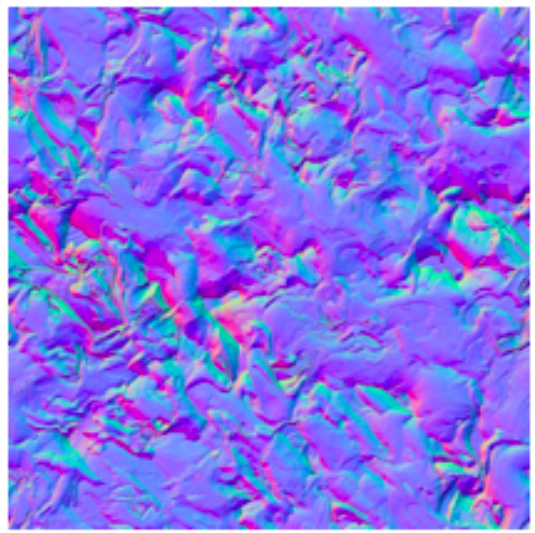
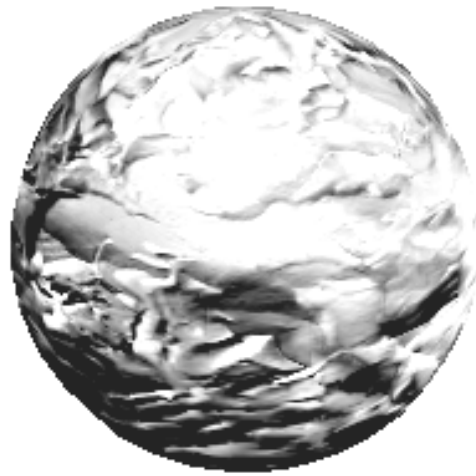
Computer Graphics: Additional effects using shaders

Table of contents

Bump (or normal) mapping

From lecture 5 on texture: Bump mapping is a technique for simulating details or wrinkles on a surface by perturbating the normal vector field.

Example of application

A detailed model is first created from an artist. From this model a normal map and a low-count poly model (left image) are extracted. At runtime, the details are added by using the normal map for the lighting computation with the low-count poly model, resulting in the image on the right.

Low count poly model

Low count poly model
with bump mapping

Normal texture

A normal texture encodes in each RGB texel a XYZ vector corresponding to the coordinates of the normal vector.
Each color component is between 0 and 1, while normal coordinates are between -1 and 1. The mapping between the two is given by:

```
normal = 2*color - 1;
```

Normals are expressed in the space of each individual triangle. We need to convert them to model space (used in our shading equation).

Tangent and bitangent

We already have the unit normal N to the surface at each vertex.
We need an additional vector T tangent to the surface. A coordinate system can then be constructed.
There are infinitely many vectors in the plane perpendicular to N and passing through the vertex.
The standard method is to pick the tangent in the same direction than the texture coordinates.
The final vector, the bitangent B, is chosen such that T, B, N forms an orthonormal coordinate system.

Tangent and bitangent computation

Let v_1, v_2 be the two vectors out of a given vertex p_0 corresponding to the two edges (adjacent to p_0) in a given triangle, and $(ds_1, dt_1), (ds_2, dt_2)$ be the corresponding differences of the texture coordinates. The tangent T and bitangent B are given by the solution of:

$$v_1 = ds_1\, T + dt_1\, B$$

$$v_2 = ds_2\, T + dt_2\, B$$

We solve this system for T and B.

Tangent and bitangent computation

```
INPUT: triangle vertices p0, p1, p2,
       texture coordinates (s0,t0), (s1,t1), (s2,t2)
OUTPUT: vector tangent and bitangent

v1 = p1 - p0;
v2 = p2 - p0;
(ds1,dt1) = (s1,t1) - (s0,t0);
(ds2,dt2) = (s2,t2) - (s0,t0);
d = 1.0 / (ds1*dt2 - dt1*ds2);
tangent = (dt2*v1 - dt1*v2)*d;
bitangent = (ds1*v2 - ds2*v1)*d;
```

Tangent space and model space

Given T, B, N, the matrix to go from tangent space to model space is:

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

Orthogonalization

The coordinate system (T, B, N) is close to, but not necessarily orthogonal. An orthogonal system can be obtained using Gram-Schmidt orthogonalization:

$$T' = T - (N \cdot T)N$$

$$B' = B - (N \cdot B)N - (T' \cdot B)T'$$

Then we can normalize these vectors to get an orthonormal basis.

Tangent space and model space

The matrix above allows to transform normals (from the texture) to model space. In practice, everything is transformed from model space to tangent space and the normal (from the texture map) is used as this.
The matrix for going from model space to tangent space is given by the transpose (the matrix is orthogonal):

$$\begin{bmatrix} T'_x & T'_y & T'_z \\ B'_x & B'_y & B'_z \\ N_x & N_y & N_z \end{bmatrix}$$

Preparing data for the shaders: attributes

The following information (attributes) needs to be sent to the shaders:

- vertex coordinates in model space
- normal (per vertex) in model space
- texture coordinates (per vertex)
- tangent and bitangent vector (per vertex); as computed above

Preparing data for the shaders: uniform

The following uniform variables will be needed for computations in the shaders:

- The model-view matrix,
- The normal matrix (for transforming normal vectors to camera space),
- The projection matrix,
- The light direction (already in camera space)

Bump mapping: vertex shader

```
1  in vec4 vPosition;
2  in vec2 vTexCoord;
3  in vec4 vNormal;
4  in vec4 vTangent;
5  in vec4 vBitangent;
6
7  out vec2 TexCoord;
8  out vec4 EyeDir_ts;
9  out vec4 LightDir_ts;
10
11 uniform mat4 MVMatrix;
12 uniform mat4 PMatrix;
13 uniform mat4 NormalMatrix;
14 uniform vec4 LightDir; // in camera space
```

Bump mapping: vertex shader

```
1  void main() {
2    // vertex coordinates in clip space
3    gl_Position = PMatrix * MVMatrix * vPosition;
4
5    // pass the texture coordinates
6    TexCoord = vTexCoord;
7
8    // Tangent, bitangent and normal in camera space
9    vec4 N = normalize(NormalMatrix * vNormal);
10   vec4 T = normalize(MVMatrix * vTangent);
11   vec4 B = normalize(MVMatrix * vBitangent);
```

Bump mapping: vertex shader

```
1   // Direction vertex to camera (in camera space)
2   vec4 Position_cs = MVMatrix * vPosition;
3   vec4 EyeDir_cs = vec4(0,0,0,1) - Position_cs;
4
5   mat4 TBN = transpose(mat4(T, B, N));
6
7   EyeDir_ts = TBN * EyeDir_cs;
8   LightDir_ts = TBN * LightDir;
9 }
```

Bump mapping: fragment shader

```
1 in vec2 TexCoord;
2 in vec4 EyeDir_ts;
3 in vec4 LightDir_ts;
4
5 out vec4 FragColor;
6
7 // for normal map
8 uniform sampler2D NormalTextureSampler;
```

Bump mapping: fragment shader

```
1 void main() {
2   // material properties
3   vec3 MaterialDiffuse = vec3(0.5,0.5,0.5);
4   vec3 MaterialAmbient = vec3(0.1,0.1,0.1);
5   vec3 MaterialSpecular = vec3(0.9,0.9,0.9);
6
7   // retrieve texture in texture space
8   vec4 N = vec4(texture2D(NormalTextureSampler,
      TexCoord).rgb * 2.0 - 1.0), 0.0);
9   N = normalize(N);
```

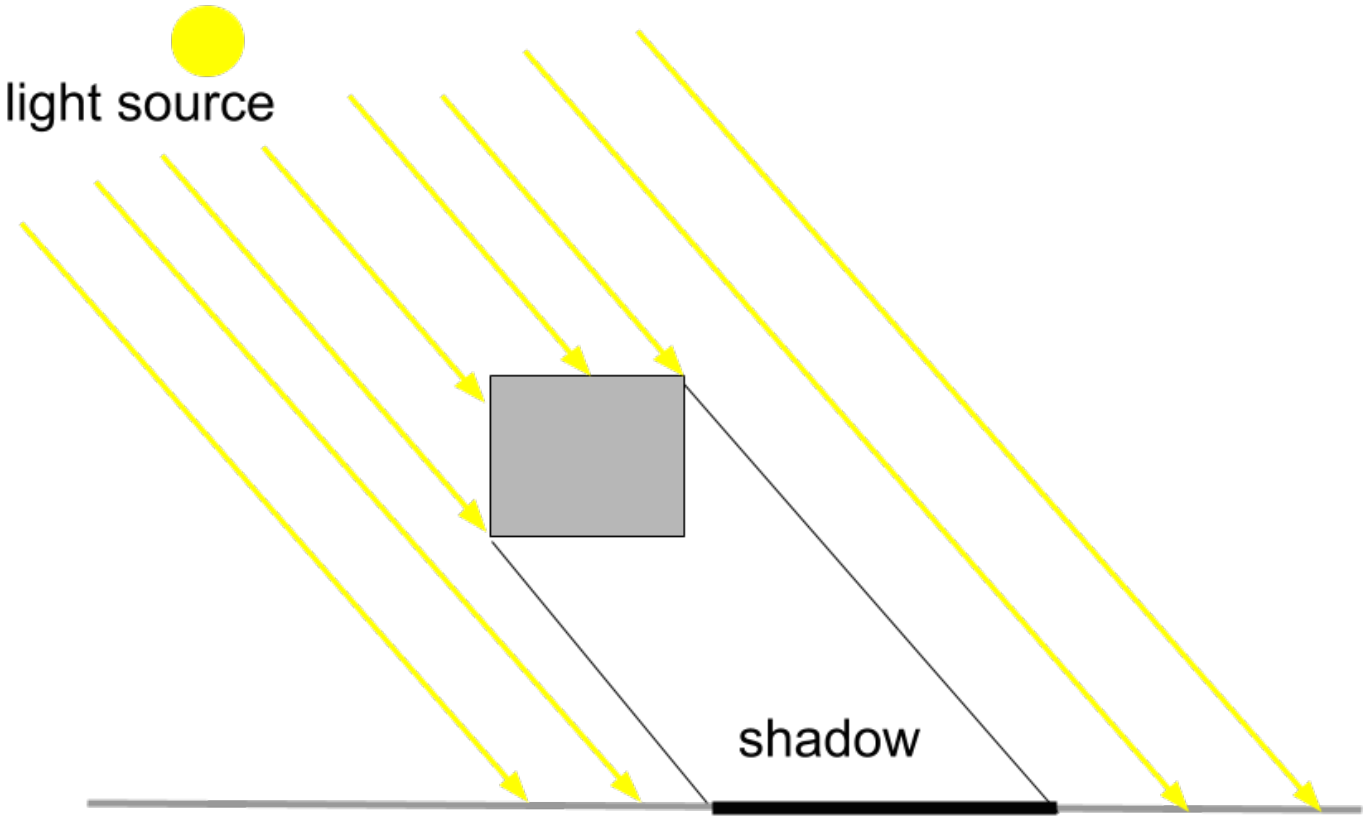
Bump mapping: fragment shader

```
1   // Diffuse coefficient
2   vec4 L = normalize(LightDir_ts);
3   float diff = clamp(dot(L, N), 0, 1);
4
5   // Specular coefficient
6   vec4 V = normalize(EyeDir_ts);
7   vec4 R = reflect(-L, N);
8   float spec = clamp(dot(V, R), 0, 1);
9
10  FragColor = vec4(MaterialAmbient + diff*
      MaterialDiffuse + spec*MaterialSpecular, 1.0);
11  }
```

Shadow mapping

One possible way to produce shadows is by creating lightmaps.
However, it doesn't work with animated models.
Dynamic shadows can be done with shadow mapping

Shadow mapping



Shadow mapping

Basic idea: render the scene in two passes.
First pass: render the scene from the light viewpoint, and compute the depth at each fragment. Store the result in a texture: the shadow map.
Second pass: render the scene normally with an extra test to check if the current fragment is in the shadow or not.
To test if a fragment is in the shadow, compare its depth with the shadow map at the same point.

Depth texture

We render the scene (only its depth) in a texture:

```
1 GLuint fbObj = 0;
2 glGenFramebuffers(1, &fbObj);
3 glBindFramebuffer(GL_FRAMEBUFFER, fbObj);
```

Depth texture

```
1 GLuint dtexObj;
2 glGenTextures(1, &dtexObj);
3 glBindTexture(GL_TEXTURE_2D, dtexObj);
4 glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
      shadowMapWidth, shadowMapHeight, 0,
5 GL_DEPTH_COMPONENT, GL_FLOAT, 0);
6 // ... set the Min/Mag filter and Wrap mode
```

Depth texture

```
1 // write the depth values to the texture
2 glFramebufferTexture2D(GL_FRAMEBUFFER,
      GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, dtexObj, 0);
3 glDrawBuffer(GL_NONE); // only depth no color buffer
      drawn to
```

Depth texture rendering preparation

The Model-View projection used to render the scene is from the point of view of the light.

```
// Assume light_pos contains the light position
// and light_lookAt is where the light look at
// The view matrix is obtained by:
mat4 depthMVMatrix = LookAt(light_pos[0], light_pos[1],
    light_pos[2],
light_lookAt[0], light_lookAt[1], light_lookAt[2], 0,
   1, 0);
```

Depth texture rendering

```
1 glBindFramebuffer(GL_FRAMEBUFFER, fbObj);
2 glViewport(0,0,shadowMapWidth,shadowMapHeight);
3 glClear(GL_DEPTH_BUFFER_BIT);
4 glUseProgram(depthShader); // use shader for depth
      rendering
5 // ... set up the Model-View and projection matrices
6 // ... the buffer data and render (glDrawArrays)
```

Depth texture rendering vertex shader

```
1 in vec4 vPosition;
2 uniform mat4 DepthMVMatrix;
3 uniform mat4 DepthPMatrix;
4
5 void main() {
6   gl_Position = DepthPMatrix * DepthMVMatrix *
      vPosition;
7 }
```

Depth texture rendering fragment shader

```
1 out float FragDepth;
2
3 void main() {
4   FragDepth = gl_FragCoord.z;
5 }
```

Scene rendering with the shadow map

Given the shadow map, we need to render the scene with the usual method.
Only difference: for each fragment, test if it is in the shadow or not.
So for each fragment, two computations are needed:

- one with the transformation from the usual model-view and projection matrix (normalMVP)
- one with the same transformation that we used to compute the shadow map (depthMVP)

Bias mapping

After multiplication by the depth projection matrix and the depth model-view matrix, the vertex positions are in normalized device coordinates. $[-1, 1]$
For sampling the depth texture, coordinates in $[0, 1]$ are needed.
The transformation is done as follows: divide coordinates by 2 (from $[-1, 1]$ to $[-0.5, 0.5]$), then shift by 0.5 (from $[-0.5, 0.5]$ to $[0, 1]$).
It corresponds to the matrix (bias matrix):

$$\begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rendering setup

```
1  // render to screen instead of to texture
2  glBindFramebuffer(GL_FRAMEBUFFER, 0);
3  glViewport(0, 0, winWidth, winHeight);
4  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
5
6  // use our rendering shader
7  glUseProgram(renderShader);
8
9  // ... compute the model-view and projection matrices (
       as usual)
10 // ... form the bias matrix as described above
11 // ... set up the buffer data and render
```

Rendering: vertex shader

```
1 in vec4 vPosition;
2
3 out vec4 ShadowCoord;
4
5 uniform mat4 MVMatrix;
6 uniform mat4 PMatrix;
7 uniform mat4 DepthMVMatrix;
8 uniform mat4 DepthPMatrix;
9 uniform mat4 BiasMatrix;
```

Rendering: vertex shader

```
1 void main() {
2   // vertex coordinates in NDC
3   gl_Position = PMatrix * MVMatrix * vPosition;
4
5   // same but using the light's view matrix
6   ShadowCoord = BiasMatrix * DepthPMatrix *
      DepthMVMatrix * vPosition;
7 }
```

Rendering: fragment shader

```
1  in vec4 ShadowCoord;
2
3  out vec4 FragColor;
4
5  uniform sampler2DShadow ShadowMap;
6
7  void main() {
8    vec3 materialColor = vec3(0.9,0.9,0.9);
9    float f = textureProj(ShadowMap, ShadowCoord);
10   FragColor = vec4(f*materialColor, 1.0);
11 }
```

Projective texturing and comparison function

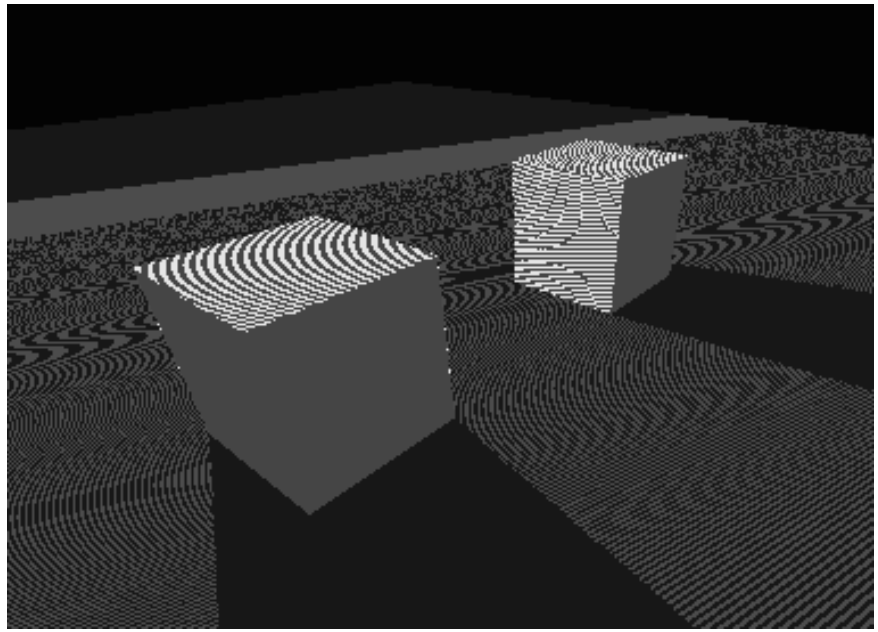
The call to textureProj(ShadowMap, ShadowCoord) performs a projective texture lookup and filtering.
ShadowCoord.xy/ShadowCoord.w is used for the texture lookup into the shadow map.
Then ShadowCoord.z/ShadowCoord.w is compared to the sampled depth from the shadow map:

- if it is less or equal, the result is 1.0. It corresponds to the case where the fragment is not in the shadow;
- otherwise, the result is 0.0. It corresponds to the case where the fragment is in the shadow.

This value is used to modulate the fragment color.
The same result can be achieved using regular texture sampling (with the function texture).

Self-shadowing

Unfortunately, because of limited precision of the depth buffer, self-shadowing can occur (this is also known as Z-fighting).



Self-shadowing

A common practical approach to solve the problem consists also in adding a small bias during sampling:

```
// in the fragment shader for rendering
float bias = 0.005;
ShadowCoord.z = ShadowCoord.z - bias;
```

Self-shadowing

Result after using an additional bias: