Danmarks Tekniske Universitet

Realistic texture rendering

02561: Computer graphics - Final project

Authors

Alessandro Longo - s243139

December 16, 2024

Contents

1 Introduction

The aim of this work is to collect texture assets from PolyHaven [1] with layers such as diffuse, displacement, normal, roughness, ambient occlusion, and use them to render a model (a sphere and possibly other objects) with a high level of detail.

The scene will include:

- a model, positioned in the origin, to which a set of textures will be applied
- a cubemap, which will be used to add athmosphere to the scene and, most importantly, reflection on the metallic pieces of the model, if any
- a directional light shining on the object, to highlight all its details

Finally, it will be possible to select different models, texture packs, and cubemaps, as well as using the mouse to orbit around and zoom on the center of the scene, where the object lies.

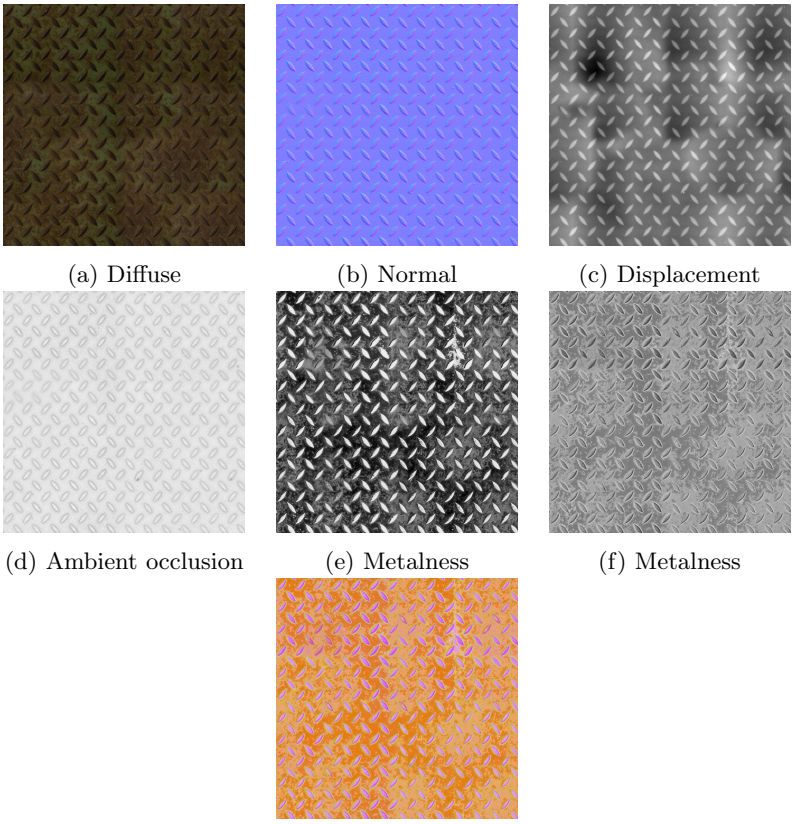
2 Method

The main goal of this project is to use different types of textures to add details to an object and get the closer possible to a realistic rendering.

To do so, we will mainly use 6 types of textures:

- diffuse, or albedo texture: the first and most trivial is an rgb texture, that simply gives the color to the object.
- normal map: each pixel in this texture encodes the XYZ components of a normal vector as RGB values. These vectors represent the direction of the surface's normal at each point, simulating detailed surface geometry without requiring additional polygons. Normal maps are easily identifiable by their characteristic blue-purple hue: this occurs because the default normal direction points along the positive Z-axis, which corresponds to the blue channel. Every other subtle color variation in the map corresponds to a slight offset from the normal.
- displacement map (or height map): another gray-scale texture, where the pixel value indicates how much the underlying vertex should be displaced, or extruded, from the object along the normal. It needs the normal vector to work properly and since it modifies the vertices position, it has to be applied directly to the object geometry. The higher the polygon count, the better the effect of this map on the object.
- arm texture: each channel in this texture corresponds to a gray-scale texture itself. ARM is an acronym that stands for Ambient occlusion, Roughness, Metalness:

– ambient occlusion: it creates occlusion shadows on the object, i.e. fake shadows that lie in the parts of the textured model that ambient light struggles to reach. In this texture, white represents an illuminated region and black a shadowed one.
– roughness: it encodes how rough the object is and is used to scatter light rays that reach it. The lighter, the smoother the material, and viceversa.
– metalness: it works like roughness, but it's used to add reflections to the material. A metalness of 100% results in all the light to be reflected by the material, essentially making it a mirror



(a) Diffuse (b) Normal (c) Displacement

(d) Ambient occlusion (e) Metalness (f) Metalness

(g) ARM

Figure 1: The "Metal plate" set of textures from Polyhaven

If the textures are high-quality, applying just these types is already enough to produce a realistic and convincing result. Nonetheless, if one wanted to create particular effects or just experiment with more, there are additional types of textures, such as emissive, transparency, sheen, etc.

3 Implementation

Each of these textures can be applied individually to contribute specific visual effects, and by layering these textures, we can achieve a more complex and realistic appearance, which can be appreciated in the final renders.

In the following sections, we will explore the implementation of each texture in detail, discussing whether they are coded in the vertex and/or in the fragment shader and why.

3.1 Diffuse (or albedo)

The diffuse map is the easiest to implement: after passing the texture to the fragment shader, we just calculate the position and apply the color found at the correspondent texture coordinate.

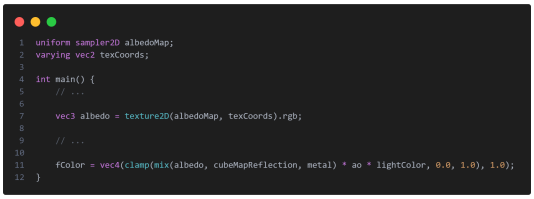
```
varying vec2 texCoords;

void main() {
    // ...

    texCoords = vec2(
        (0.5 + (atan(cubeMapCoords.x, cubeMapCoords.z) / (2.0 * PI))) * patternRepeat,
        (0.5 - (asin(-cubeMapCoords.y) / PI)) * patternRepeat
    );

    // ...
}
```

(a) Vertex shader



```
uniform sampler2D albedoMap;
varying vec2 texCoords;

int main() {
    // ...

    vec3 albedo = texture2D(albedoMap, texCoords).rgb;

    // ...

    fColor = vec4(clamp(mix(albedo, cubeMapReflection, metal) * ao * lightColor, 0.0, 1.0), 1.0);
}
```

(b) Fragment shader

Figure 2: Shader code snippets for albedo map implementation

```
// vertex shader
attribute vec4 vNormal;
varying vec3 fNormal;

fNormal = normalize((modelMatrix * vec4(vNormal.xyz, 0.0)).xyz);

// fragment shader
varying vec3 fNormal;
uniform sampler2D normalTexture;

vec3 rotate_to_normal(vec3 n, vec3 v) {
    vec3 T = normalize(vec3(-n.z, 0, n.x));
    vec3 B = cross(n, T);
    mat3 TBN = mat3(T, B, n);
    return TBN * v;
}

vec3 tangentNormal = texture2D(normalTexture, texCoords).rgb * 2.0 - 1.0;
vec3 worldNormal = rotate_to_normal(fNormal, tangentNormal);
```

Figure 3: Shader code responsible to get the normal map information

3.2 Normal

To extract information from a normal map, the process involves several steps to correctly align the normals from the texture with the 3D object.

First, in the vertex shader, the vertex normal is transformed into world space by multiplying it with the modelMatrix, and then passed to the fragment shader. The normals in the normal map are in tangent space (a local, flattened space aligned with the texture), so in the fragment shader they need to be aligned with the 3D object. To do this, they are rotated to be aligned with the vertex normal space (thanks to the rotate_to_normal helper function). This "wraps" the normals from the flat texture around the object's surface.

3.3 Displacement

As the displacement map affects the geometry of the object, its implementation resides solely in the vertex shader, where move each vertex along its normal by the amount encoded in the texture, for that specific point in the object.

Since the final project allows the user to also change the number of times the textures are repeated on their xy local plane, to ensure a consistent effect on the object, I divided the amount by the number of those repetitions. In this way, the relative displacement was constant.

```glsl
// vertex shader
uniform mat4 modelMatrix;
varying vec3 fNormal;

uniform sampler2D displacementTexture;
uniform float displacementScale;
uniform float patternRepeat;

vec4 worldPos = modelMatrix * vPosition;
float displacement = texture2D(displacementTexture, texCoords).r;
vec3 displacedPos = worldPos.xyz + fNormal.xyz * displacement * displacementScale / patternRepeat;
```

Figure 4: Shader code for displacing the vertices in the object

```glsl
varying vec2 texCoords;
uniform sampler2D armTexture;

void main() {
    // ...

    // same calculation for the texCoords

    float ao = texture2D(armTexture, texCoords).r;
    fColor = vec4(clamp(mix(albedo, cubeMapReflection, metal) * ao * lightColor, 0.0, 1.0), 1.0);

    // ...
}
```

Figure 5: Shader code for displacing the vertices in the object

3.4 Ambient occlusion

Since the ambient occlusion map has values from 0 to 1, it can be directly used to darken the diffuse texture, by multiplying the two values together as follows:

3.5 Roughness and metalness

The implementation of the last two material properties, being roughness and metalness, lies directly in the light calculations. Since they describe how light bounce off of the object, their scalar values are used to calculate the shininess and specular coefficient values:

3.6 Texture loader

The last thing worth mentioning is the way we load our textures. The pipeline is composed by the following steps:

- activating a texture unit based on the type we want to load (1 to 5)
- creating the texture object and set its filtering and wrapping parameters
- get the location and link the texture to the shader uniform
- load and process the images by setting the source to the local path of the texture

4 Results

The results are very good and even though we're not using complex techniques like ray-tracing, the rendered scene is somewhat realistic and convincing.

```glsl
uniform sampler2D armTexture;
uniform sampler2D metalTexture;

vec3 calcSpecularColor(vec3 n, vec3 w_i, vec3 pos, float s, float ks, vec3 baseColor, vec3 Ls, float metal) {
    vec3 w_r = reflect(w_i, n);
    vec3 w_o = normalize(eye - pos);
    float specular = pow(max(dot(w_r, w_o), 0.0), s) * max(dot(n, -w_i), 0.0);

    vec3 specularColor = mix(vec3(1.0), baseColor, metal);
    return ks * specular * specularColor * Ls;
}

void main() {
    // ...

    float roughness = texture2D(armTexture, texCoords).g;
    float metal = texture2D(metalTexture, texCoords).r;

    // ...

    float s = pow(1.0 - roughness, 4.0) * 128.0;
    float ks = mix(0.04, 1.0, metal);
    vec3 specularColor = calcSpecularColor(worldNormal, w_i, fPosition.xyz, s, ks, albedo, vec3(1.0), metal);

    // ...

    fColor = vec4(clamp(mix(albedo, cubeMapReflection, metal) * ao * lightColor, 0.0, 1.0), 1.0);
}
```

Figure 6: Shader code to implement the light behaviour when applying roughness and metalness textures

```javascript
const initTextures = (texturePack) => {
    initTexture("./textures/" + texturePack + "/textures/" + texturePack.slice(0, -2) + "diff_" + texturePack.slice(-2) + ".jpg", "albedoMap", 1);
    initTexture("./textures/" + texturePack + "/textures/" + texturePack.slice(0, -2) + "disp_" + texturePack.slice(-2) + ".jpg", "displacementTexture", 2);
    initTexture("./textures/" + texturePack + "/textures/" + texturePack.slice(0, -2) + "nor_gl_" + texturePack.slice(-2) + ".png", "normalTexture", 3);
    initTexture("./textures/" + texturePack + "/textures/" + texturePack.slice(0, -2) + "arm_" + texturePack.slice(-2) + ".jpg", "armTexture", 4);
    initTexture("./textures/" + texturePack + "/textures/" + texturePack.slice(0, -2) + "metal_" + texturePack.slice(-2) + ".jpg", "metalTexture", 5);
}

const initTexture = (srcPath, shader_keyword, textureUnit) => {
    gl.activeTexture(gl[`TEXTURE${textureUnit}`]);
    const texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);

    const textureLoc = gl.getUniformLocation(program, shader_keyword);
    gl.uniform1i(textureLoc, textureUnit);

    const myTexels = new Image();
    myTexels.texture = texture;
    myTexels.onload = (event) => {
        gl.activeTexture(gl[`TEXTURE${textureUnit}`]);
        gl.bindTexture(gl.TEXTURE_2D, event.target.texture);
        gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
        gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, gl.RGB, gl.UNSIGNED_BYTE, myTexels);
        gl.generateMipmap(gl.TEXTURE_2D);
    };
    myTexels.src = srcPath;
};
```

Figure 7: Texture loading functions

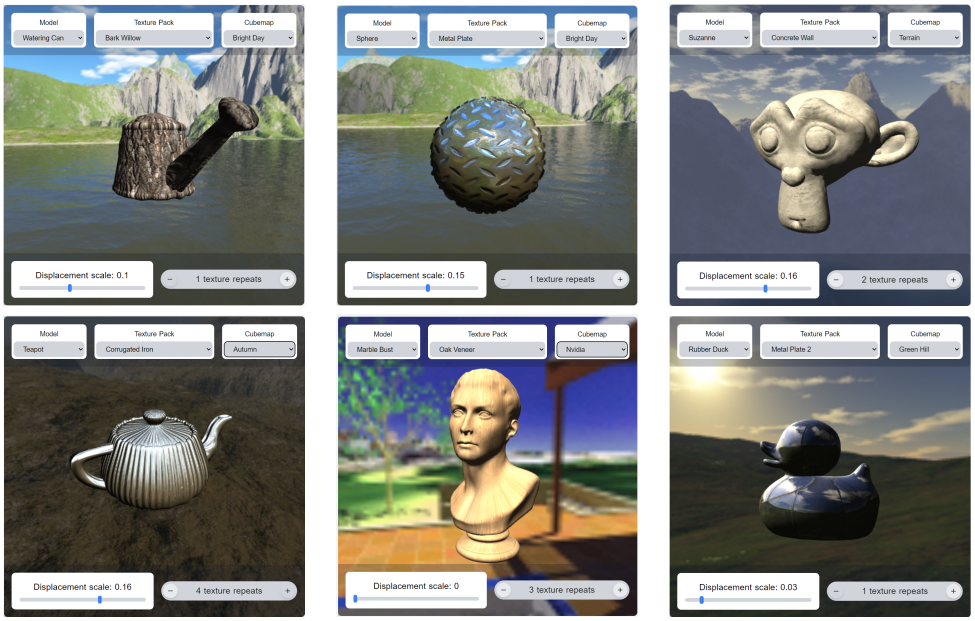


Figure 8: Some examples of the scene we can create with this method. In the UI we can see model, texture and cubemap selected and the and parameter combination is use.

Here we show some examples combinations of models, cubemaps and texture packs:

One thing that one can easily notice is that, for some models, there are points where the texture get stretched or compressed. This effect is known as "texture distortion" and is more evident if we use geometric textures with parallel lines or grids.

As we can see, it's accentuated on the poles and in surfaces that are aligned with radial vectors coming from the origin.

Figure 9: Examples of areas where the texture get distorted. In the first two examples, we highlight a compressed effect on the poles; in the last, we can see the texture strecthing on the chest, the tangents of which are directed to the origin.

This is due to how the textures are wrapped around the object. In fact, when they are applied, the default method is to use a spherical projection for the UV mapping, which projects the texture using sphere coordinates. The unwanted result is that all the points along the top and bottom border of the texture get compressed around the poles, while the ones in the middle get stretched on the equator.

As for the surfaces aligned with radial vectors, points far from each other are the target for points in the texture close to each other, so the texture appears stretched.

5 Discussion

The solution to the texture distortion problem would be to use different projections for each general obejct shape: spherical object could still use spherical projection, but more elongated model, such as the marble bust, would benefit from a cylindrical projection. This are still partial solutions, since the texture does not adapt exactly to the object shape: a complete fix would be to use a 3D graphic software to create some UV unwrapping for each model, and use those to project the textures. This would lead

equally spaced points on the texture to be mapped to equally spaced points on the object, fixing the distorting effect.

References

[1] Poly Haven. Poly haven: Free textures, hdris, and 3d models. https://polyhaven.com, 2024. Accessed: 2024-12-16.