

# Texture Handling

## Color and Normal Maps

Francesco Andreussi

Bauhaus-Universität Weimar

9 January 2020

Bauhaus-Universität Weimar

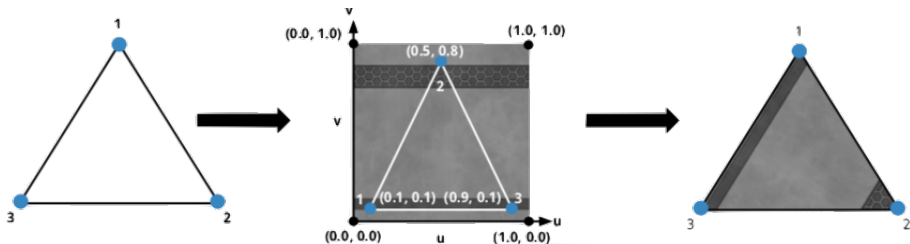
Faculty of Media

# Texture Mapping (1)

## The Basics

Texturing is a tool to enhance the level of detail of a model.

The most common and convenient way of passing texture data are images (i.e. matrices of 3D vectors). In order to apply the data to a geometry is necessary to assign to every vertex a **texel** (pixel of a texture), that is identified by a pair of floats in the interval  $[0,1]$  (for a 2D texture). The other values, are obtained with interpolation.



**Fig. 1:** Basic Texturing Example

# Texture Mapping (2)

## Wrapping

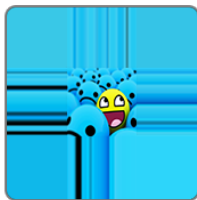
Sometimes we want to use a small texture on a large object and, to maintain a good resolution, we need to assign to the vertices texels values outside the  $[0,1]$  interval. These values will be processed in different ways w.r.t. the specified wrapping mode: `GL_REPEAT`, `GL_MIRRORED_REPEAT`, `GL_CLAMP_TO_EDGE`, `GL_CLAMP_TO_BORDER`.



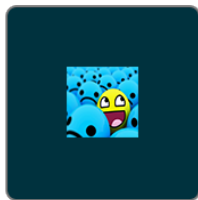
GL\_REPEAT



GL\_MIRRORED\_REPEAT



GL\_CLAMP\_TO\_EDGE



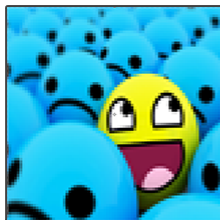
GL\_CLAMP\_TO\_BORDER

Fig. 2: Wrapping Possibilities

# Texture Mapping (3)

## Filtering

The amount of texels is finite, but the texture coordinates are floats. How to map values that are not exactly corresponding to a texel? OpenGL provides two alternatives: `GL_NEAREST` (nearest texel position, default, pixel-ish effect), `GL_LINEAR` (bilinear interpolation of neighbouring texels, blurred effect).



GL\_NEAREST



GL\_LINEAR

**Fig. 3:** Filtering Options

# Texture Mapping (4)

## Mipmapping

Using mipmapping, it is possible to assign textures with different resolutions to objects w.r.t the required level of detail we need at a certain moment (distance from the camera, scaling, etc.). In OpenGL is possible to pass special texture and generate a mipmapping texture set with `glGenerateMipmaps`.



**Fig. 4:** A Mipmapping Texture

Switching between two levels, some artifacts could appear. In order to minimise these effects it is possible to apply some filtering operations to the mipmaps: `GL_NEAREST_MIPMAP_NEAREST`, `GL_NEAREST_MIPMAP_LINEAR`, `GL_LINEAR_MIPMAP_NEAREST`, `GL_LINEAR_MIPMAP_LINEAR`.

# Texture Initialisation (1)

## Theory

An OpenGL Context can handle only a certain number of textures at a time (`GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`).

It is possible to assign a texture to any of the **Texture Units** via binding. The Texture Units have several binding points (one for every kind of allowed texture), but it is mandatory to use **only ONE** binding point per Texture Unit.

In every moment of the program execution, there must be only and only one **active** texture.

# Texture Initialisation (2)

## Practice

### Initialise Texture

```
glActiveTexture(GL_TEXTURE*)  
glGenTextures(tex_num, &texture_object)  
glBindTexture(target, texture_object)
```

### Define Texture Sampling Parameters (mandatory)

```
glTexParameter_i(target, GL_TEXTURE_MIN_FILTER, GL_LINEAR)  
glTexParameter_i(target, GL_TEXTURE_MAX_FILTER, GL_LINEAR)
```

### Define Texture Data and Format

```
glTexImage2D(target, level, internalformat, width, height, border,  
format, type, data)
```

# Texture Formatting

```
glTexImage*(target, level, internalformat, width, height,  
            border, format, type, data)
```

- target: binding point for the texture (must match the sense of the function name)
- level: number of levels of detail in the texture (0 if there's only the base image)
- internalformat: internal format of the texture
- width, height: dimensions of the image (valid for 2D images)
- border: thickness of the border (0 means no border)
- format: format of the texel data (should match the internalformat)
- type: data type of the texel data
- data: pointer to the texture data

glTexImage2D



# Texture Usage

## Bind for Accessing

```
glActiveTexture(GL_TEXTUREk)  
glBindTexture(target, texture_object)
```

## Upload Texture Unit data to shader

```
int sampler_location = glGetUniformLocation(program_handle,  
    'YourTexture')  
glUseProgram(program_handle)  
glUniform1i(sampler_location, k)
```

## Use the Sampler in the Shader

```
uniform Sampler2D YourTexture  
vec4 colour_from_tex = texture*(YourTexture, tex_coords)
```

# Normal Transformation

Normals are not always affected in the same way of vertices by the normal transformation: translations have no effect on normals (that are directions), rotations transform normals and vertices (and surfaces) in the same way, (non-uniform) scalings, instead, transform normals in the **opposite** way w.r.t. the vertex positions.

Hence, it is necessary to transform the normals applying the inverse of the transformations applied to the vertices ( $M_{normal} = M_{vertex}^{-1}$ ).

This operation affects also the rotation values. Luckily, the inverse of a rotation matrix is its *transpose*, yet the scaling matrices are not modified by this operation:  $R^T = R^{-1}$  and  $S^T = S$ .

In order to transform normals correctly, then, we need to invert the vertices transformation and transpose the result, so the side effects on the rotation values are erased (i.e. apply the **inverse-transpose**):

$$M_{normals} = (M_{vertex}^{-1})^T = M_{vertex}^{-T}.$$

# Normal (or Bump) Mapping

Since a **normal** is a 3D vector, it could be represented with a colour. It means that we can pass some normal information via an image.

This results in the possibility of achieving a superior grade of detail and refinement of the environment, not being forced to use a huger number of vertices.

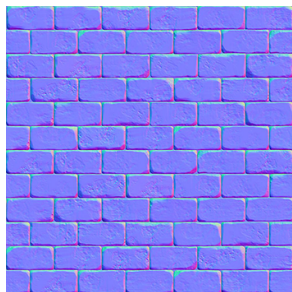


Fig. 5: A Normal Map

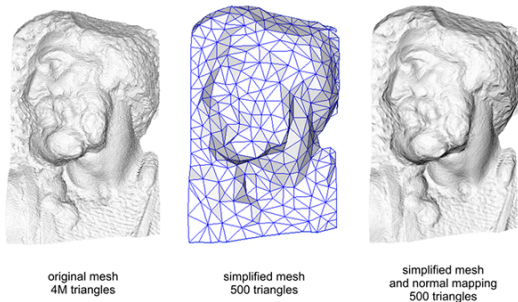


Fig. 6: Normal Mapping Effect

# Tangent Space

To be completely precise, the values of the Normal Map are to be interpreted w.r.t. the **tangent** of a certain fragment. In fact, the Map does not “encode” in colours the shape of the object which will be applied to.

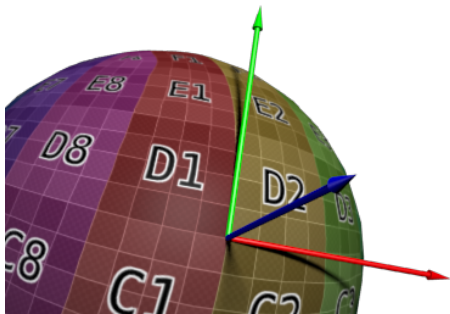


Fig. 7: Tangent Space

This explains also the predominance of the blue colour in the Bump Maps: the vector  $[0,0,1]$  represents the unchanged normal, that is the vector exiting from the surface in the direction perpendicular to the **two** tangents ( $\vec{t}$ ,  $\vec{b}$  forming an orthonormal basis).

# Tangent Calculation (1)

## Theory

In order to be able to perform the Normal Mapping, it is necessary to compute in the fragment shader the tangent space values and apply to them the new values (passed with the Normal Map).

For this purpose, the differential GLSL functions  $dFdx$  and  $dFdy$  are going to be used. They measure the variation in the values between one element and its neighbours, to the right and up respectively. This operation is to be applied to every fragment in the scene computing the tangent and the bitangent (inherited from the model shape) of every fragment and to every pixel of every Bump Map calculating the variation in the position of the values from pixel to pixel.

Now, the tangent and bitangent of each vertex can be computed; transforming the coordinates of the Map into the model space of the object.

# Tangent Calculation (2)

## Practice

```
vec3 q0 = dFdx( vertex_pos.xyz );
vec3 q1 = dFdy( vertex_pos.xyz );
vec2 st0 = dFdx( uv.st );
vec2 st1 = dFdy( uv.st );

vec3 S = normalize( q0 * st1.t - q1 * st0.t );
vec3 T = normalize( -q0 * st1.s + q1 * st0.s );
```

# Normal Mapping

## Pros & Cons

### Pros:

- Much simpler and lighter geometry objects,
- Flexibility in the use of the Map (tiling, reusing for different objects...).

### Cons:

- The silhouette is not affected from the bump mapping, i.e. the edge are still flat,
- No self-shadowing (but recent works solved this problem).

# Thanks for the Attention!