Chapter 12: Textures

K. Jünemann
Department Informations- und Elektrotechnik
HAW Hamburg

Content

Chapter 12: Textures
- Loading textures
- Texture coordinates
- Configuration of textures
- Further applications
- Render targets

Textures

Textures provide user defined information per fragment!

Application of textures:

- pixel-based color
- normal maps: pixel-based normal vectors
- specular maps: specular reflection coefficient
- etc.

Texture values $\in [0, 1]$ used for modulation:

$$\text{Final color} = \text{Texture color} \otimes \text{Original color}$$

Similar for other maps.

Loading textures

Loading of textures in `three.js`:

```
const loader = new THREE.TextureLoader();
const txt = txtLoader.load('textureName');
```

- `txt` is of type `THREE.Texture`

Applying a texture to a material:

```
const mat = new THREE.MeshPongMaterial({map:txt});
```

Alternatively:

```
mat.map = txt;
mat.needsUpdate = true;
```

Loading textures

For security reasons browsers must not load images directly from file system!

- *Option 1* (recommended): Use a web server
  - see discussion about loading modules (chapter 3).
  - image path has to be specified relative to directory where web server runs.
- *Option 2* (exotic): Store the image as base64 encoding

```
const img = new Image();      // part of browser API
img.src = imgBase64Array[0];
const txt = new THREE.Texture(img);
txt.needsUpdate = true;
```

  - see `https://en.wikipedia.org/wiki/Base64` for details.
  - `File2Base64.html` may be used to convert images.

Loading textures

Loading of textures or other data executes *asynchronously*:

- `TextureLoader.load` returns *before* loading has finished!
- For better control load method takes `onLoad` callback:

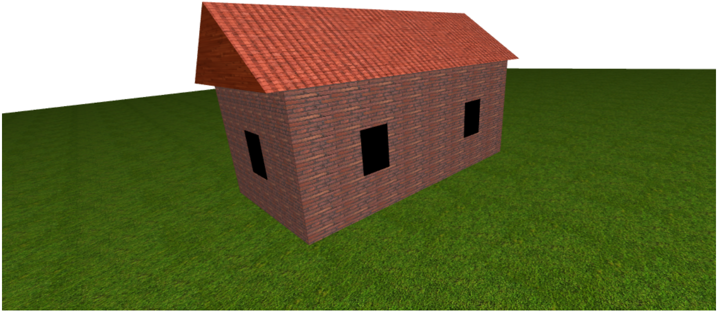
```
... = loader.load('textureName', txt => {
          mat.map = txt;
          mat.needsUpdate = true;    // don't forget!
          // whatever is useful
        }
```

- `onLoad` is called *after* loading has been finished.

Exercise 1

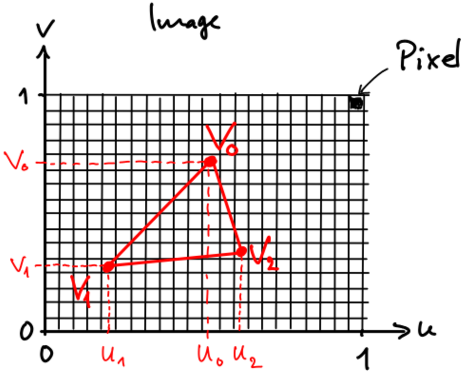
Add textures to your house program:

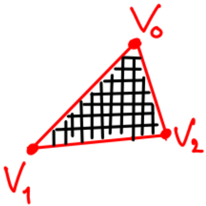
- brick wall to the body of the house
- a sun map
- a roof map



Texture coordinates

- Information per pixel stored in image
- Task: Apply image to geometry object
- Subtask: Apply part of image to face

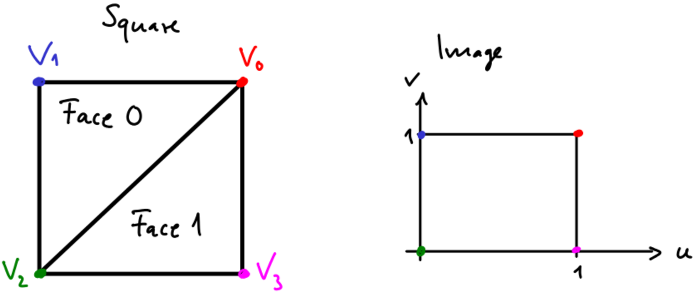




Convention: image coordinates are called u and v with $u, v \in [0, 1]$.

Texture coordinates

Each vertex has *uv*-map: Vertex $V_k \to (u_k, v_k)$.
Example: Square



- Face 0:

Vertex V_k:	0	1	2
(u_k, v_k):	(1,1)	(0,1)	(0,0)

- Face 1:

Vertex V_k:	0	2	3
(u_k, v_k):	(1,1)	(0,0)	(1,0)

Texture coordinates in `three.js`

Texture coordinates stored in *uv* buffer attribute in `BufferGeometry` object.

- uv-coordinates stored consecutively in one array
- added with `setBufferAttribute` to geometry object
- Indexed geometries: vertex has the same uvs in all faces!

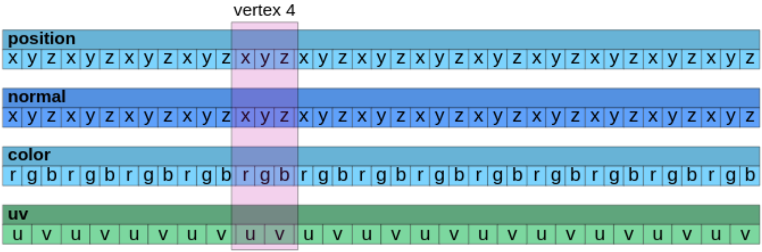


image from *Three.js Fundamentals* chapter about BufferGeometry

Texture coordinates in `three.js`

Code structure:

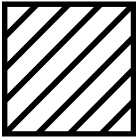
```
const uvs = new Array();
uvs.push(1, 1);    // uvs for vertex 0
uvs.push(0, 1);    // uvs for vertex 1
    ...
geo.setAttribute( 'uv',
    new THREE.Float32BufferAttribute( uvs, 2 ) );
```

- Length of uv-array: 2 $\times$ number of vertices.
- Defining *uv* coordinates only necessary when we create our own geometry!
- Predefined geometries all have the *uv* coordinates already defined.

Texture coordinates: example and exercise 2

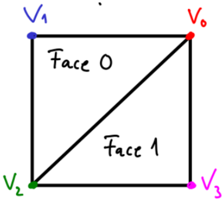
Example: Create a uv-map for a plane and apply the texture on the right:

- see `textureCoordinateDemo.js`



Exercise 2: What happens when you change the uv-map for Face 0 as follows:

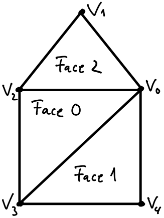
Vertex V_k:	0	1	2
(u_k, v_k):	(1,1)	(0,0)	(0,1)



1. Work out the resulting image with pen and paper.
2. Verify by changing the code in `textureCoordinateDemo.js`.

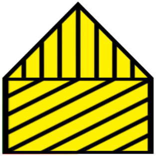
Exercise 3

Consider the `myPlaneHouseGeo` with the following vertices and faces.
(see file `textureCoordinateDemo.js`)



Create a uv-map such that with the texture `Schraffur.jpg` applied the result looks like this:

- Copy the uv-map for faces 0 and 1 from `myPlaneGeo`
- Don't apply 'trial and error' but work out uv-map for face 2 by pen and paper at first.
- Don't use an indexed geometry!



Texture coordinates: material index

It's possible to apply *several* textures to *one* geometry!

- `THREE.Mesh` objects accept an array of materials:

```
const obj
    = new THREE.Mesh(geo, [mat1, mat2, ...]);
```

Example: two materials for plane house

- `BufferGeometry.addGroup (start, count, materialIndex)`: defines which vertices use which material, e.g.:

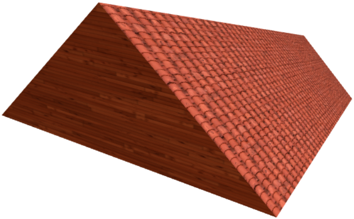
```
        geo.addGroup(0, 6, 0);
        geo.addGroup(6, 3, 1);
```

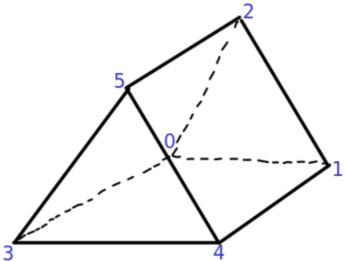
  - First two faces use `mat1`, (6 vertices starting at index 0), next faces uses `mat2` (3 vertices starting at index 6).

Exercise 4

Add a *uv*-map for two materials to the roof of our house

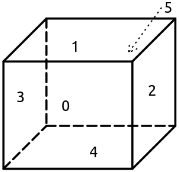
- front and back shows different texture than other parts
- textures provided in house directory



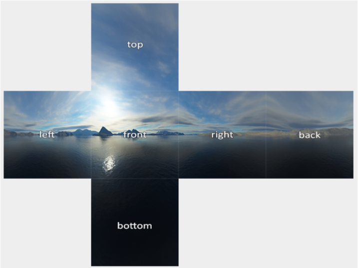


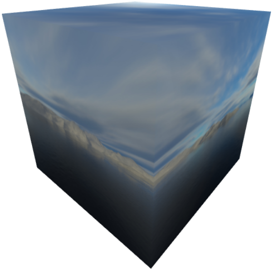
Texture coordinates: cube maps

`BoxGeometry` has predefined material index:



Cube map: apply 6 suitable textures to the inner sides of large cube and place camera *inside* cube.





Images from `https://learnopengl.com/Advanced-OpenGL/Cubemaps`

Configuration of textures: wrapping

Repeating textures: a special case of texture wrapping:

```
const txt = new THREE.Texture();
// wrap mode in u-direction
txt.wrapS = THREE.RepeatWrapping;
// wrap mode in v-direction
txt.wrapT = THREE.RepeatWrapping;
txt.repeat.set(2,3);
```

This repeats the texture 2 times in *u*-direction and 3 times in *v*-direction.

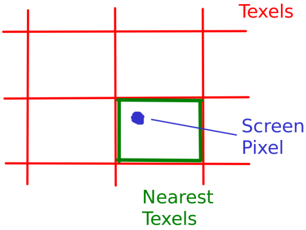
- *S* corresponds to *u* coordinate in uv-map
- *T* corresponds to *v* coordinate in uv-map

Configuration of textures: filtering

- *Texel*: pixel of texture image
- Problem: texels and screen pixels can have different sizes.
  $\Longrightarrow$ this leads to various problems!

Magnification: occurs when zooming in

- texel covers more than one pixel
- choosing nearest texel looses screen resolution!



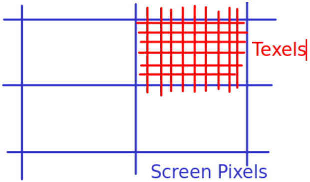
Texel selection is controlled by `Texture.magFilter`:

- `LinearFilter` (default): interpolate between 4 closest texels
- `NearestFilter`: pick closest texel

Configuration of textures: filtering

Minification: occurs when zooming out

- many texels cover one screen pixel
- choosing texel at center of pixel leads to uneasy impression!

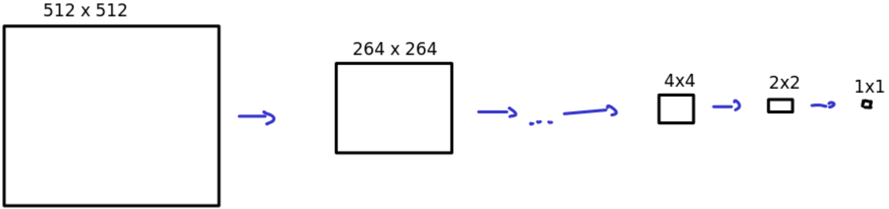


Texel selection or averaging is controlled by `Texture.minFilter`:

- `LinearFilter` and `NearestFilter`: similar to `magFilter`
- Additional concept: *mipmapping*!
  - Texel averaging in real time is slow
  - Idea: Do this averaging at texture load time!
- Default of `minFilter` is `LinearMipmapLinearFilter`

Configuration of textures: mipmaps

- Precompute averaged images of all smaller powers of 2
- Renderer picks image(s) such that texel and screen pixels are of similar size



- Mipmapping done on GPU
- Works only with texture size $= 2^N$

Configuration of textures: mipmaps

Isn't mipmapping a lot of memory overhead?

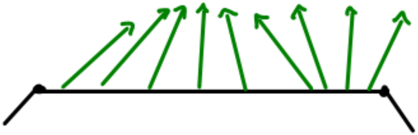
Size of mipmapping with geometric series for original image of size $2^N \times 2^N$:

$$\begin{aligned}
\text{Mipmap size} &= \sum_{k=0}^{N} 2^k \cdot 2^k = \sum_{k=0}^{N} 4^k \\
&= \frac{1-4^{N+1}}{1-4} \approx \frac{4^{N+1}}{3} = \frac{4}{3}4^N = 4^N + \frac{1}{3}4^N \\
&= \text{original size} + 33\%
\end{aligned}$$

$\Longrightarrow$ Mipmapping leads to (just) 33% memory overhead.

More applications of textures: normal map

- Normal maps: User defined per-pixel surface normals.



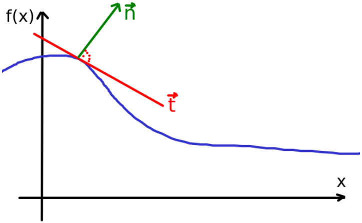
  - Three color components of an image used for normal vector definition!
  - Allow to generate bumpy surface.
  - Used to show detailed surface structure without increasing number of faces and vertices!
- In `three.js`:

```
nMap = txtLoader.load('textureName');
mat = new THREE.MeshPongMaterial({normalMap:nMap});
```

More applications of textures: bump map

A bump map provides height information as pixel-based scalar values

- stored in gray-scale image
- simple alternative to normal maps
- normal vectors can be calculated from derivative



- In `three.js`:

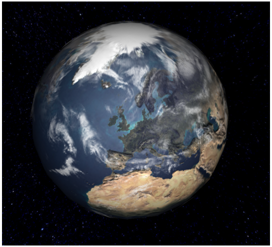
```
bMap = txtLoader.load('textureName');
mat = new THREE.MeshPongMaterial({bumpMap:bMap})
```

More applications of textures

- *Specular* map: controls amount of specular reflection
  - use case: different specular reflectivity of water and land
- *Metalness* and *roughness* maps of PBR materials
- a lot more ...

More applications of textures

Final application: good old mother earth



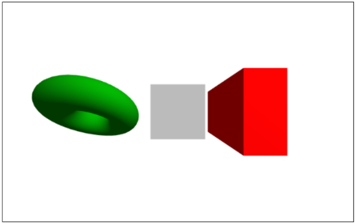
Ingredients:

- cube map of milky way
- texture map with earth surface
- specular map (scalar value) modulating specular reflection
- normal map indicting mountains
- texture map for clouds

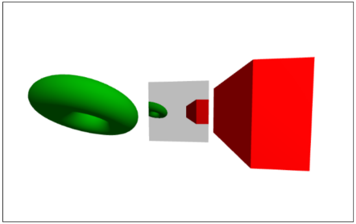
Render Targets (yet another use of textures)

The render process creates an image!

- By default it is displayed in the canvas area.
- The rendered image can be used for any other purpose, i.e. for another render *target*.



Two objects and a plane



Scene also rendered to the plane

- Alternative render targets are represented by a `WebGLRenderTarget` object, the rendered image is stored in the `texture` field.
- The rendering process is configured by `WebGLRenderer.setRenderTarget`.

Render Targets (yet another use of textures)

Code structure:

```
1  // configure the render target
2  const rt = new THREE.WebGLRenderTarget(width,
3                                         height);
4  // use the rendered image as a texture
5  const mat = new THREE.MeshPhongMaterial({
6    map: rt.texture});
7    ...
8  // in the render loop
9  renderer.setRenderTarget(rt);
10 renderer.render(scene, camera);
11 renderer.setRenderTarget(null);
12 renderer.render(scene, camera);
```

- Lines 9 and 10: Rendering to target
- Lines 11 and 12: Rendering to canvas
- Both rendering processes can use different scenes and cameras.

Render Targets (yet another use of textures)

Note: just as for canvas rendering camera and render target aspect ratio shoud coincide!

```
// configure the render target
const rt = new THREE.WebGLRenderTarget(width,
                                       height);
  ...
camera.aspect = width/height;
camera.updateProjectionMatrix();
  ...
renderer.setRenderTarget(rt);
renderer.render(scene, camera);
```