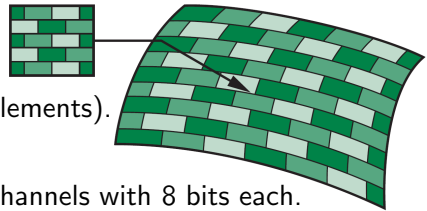
02561 Computer Graphics

Texture mapping

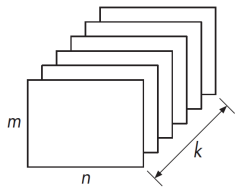
Jeppe Revall Frisvad

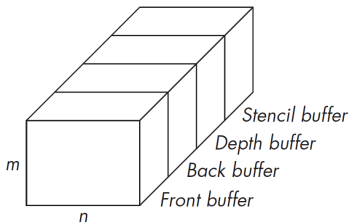
October 2020

Images, textures, and buffers



- A **digital image** is an array of pixels (picture elements).
- Image resolution and pixel data type vary.
- The more common type is R8G8B8: 3 colour channels with 8 bits each.
- A **texture** is an image that we can use to influence the colour of a fragment.
- A **buffer** is a block of memory: a discrete multidimensional (possibly layered) array.

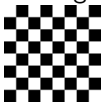




- Vertex buffer (1D example): position, colour, texture coordinate, and index layers.
- Framebuffer (2D example): front colour, back colour, depth, and stencil layers.

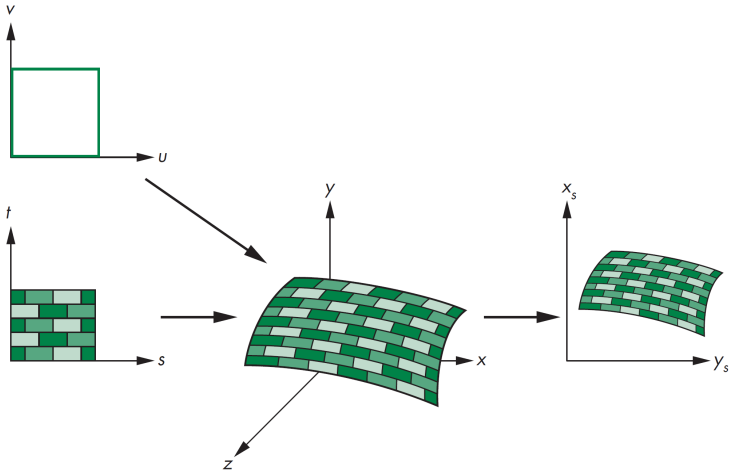
Texture mapping

- We need:
  - An image.



  - Texture coordinates.
- Quad example:

```
var texCoords = [
    vec2(0.0, 0.0),
    vec2(1.0, 0.0),
    vec2(0.0, 1.0),
    vec2(1.0, 1.0) ];
```



- This data needs to be uploaded to the GPU.
- Upload texture coordinates as another layer in the vertex buffer (like normals).
- We can load or generate an image and then upload it as a texture image.

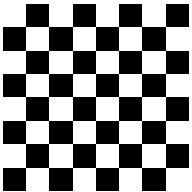
Loading or generating an image

- Generating an image: create an array of the right type and size:

```
var myTexels = new Uint8Array(4*texSize*texSize); // 4 for RGBA image, texSize is the resolution
```

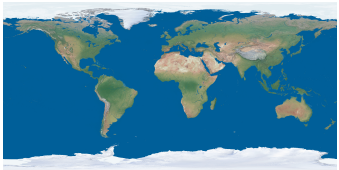
- Checkerboard example:

```
for(var i = 0; i < texSize; ++i)
  for(var j = 0; j < texSize; ++j)
  {
    var patchx = Math.floor(i/(texSize/numRows));
    var patchy = Math.floor(j/(texSize/numCols));
    var c = (patchx%2 !== patchy%2 ? 255 : 0);
    var idx = 4*(i*texSize + j);
    myTexels[idx] = myTexels[idx + 1] = myTexels[idx + 2] = c;
    myTexels[idx + 3] = 255;
  }
```

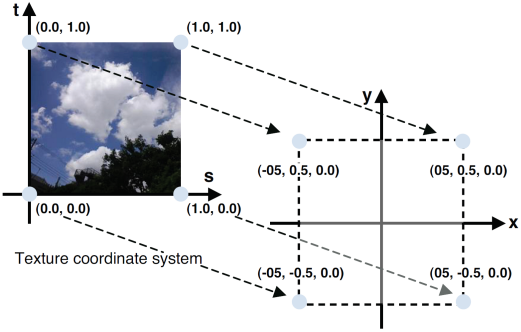


- Loading an image (asynchronously):

```
var image = document.createElement('img');
image.crossorigin = 'anonymous';
image.onload = function () { /* Insert WebGL texture initialization here */ };
image.src = 'earth.jpg';
```



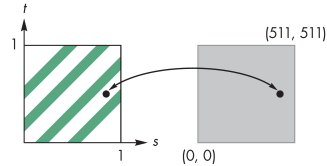
Texture coordinates



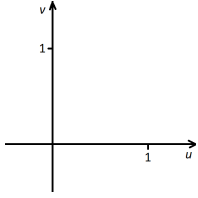
The resulting image produced by WebGL

- We assign texture coordinates (u, v) to each vertex.
- A modeling tool like Blender can be used to generate texture coordinates.
- This is called a *uv*-mapping of the surface.
- We can then load texture coordinates together with our triangle mesh.
- **Project suggestion**: Extend the OBJ parser so that it can load texture coordinates.

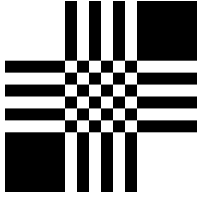
Texture wrapping



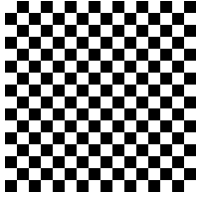
- Let us map the texture image to an *st*-coordinate system where the image is always in the region with $s, t \in [0, 1]$.
- We map *st*-coordinates to *uv*-coordinates by either clamping *u* and/or *v* to $[0, 1]$ or by removing the integer part(s) of *u* and/or *v* so that the texture repeats itself throughout the texture space.



texture space *uv*-to-*st*:



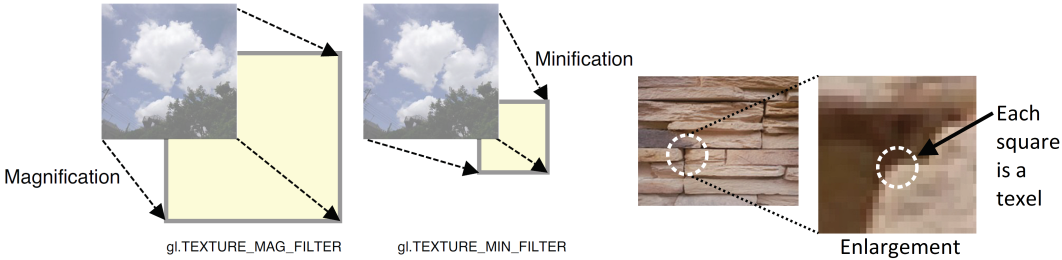
clamping `gl.CLAMP_TO_EDGE`



repeating `gl.REPEAT`

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, uv-to-st);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, uv-to-st);
```

Texture magnification and minification



- Texture magnification: When a texel covers multiple pixels.
- Texture minification: When a pixel covers multiple texels.
- Magnification requires interpolation, minification requires averaging.
- If we don't care, we use nearest-neighbor filtering:

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
```

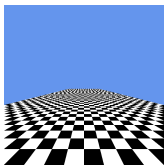
Exercise: texturing a quad (W06P1)

- Render a quad with
  - vertex positions: $(-4, -1, -1)$, $(4, -1, -1)$, $(4, -1, -21)$, $(-4, -1, -21)$
  - texture coordinates: $(-1.5, 0)$, $(2.5, 0)$, $(2.5, 10)$, $(-1.5, 10)$
- Use identity view matrix and a projection matrix generating a 90° field of view.
- Generate a checkerboard image and upload it to the GPU using

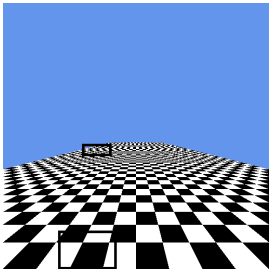
```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, texSize, texSize, 0, gl.RGBA, gl.UNSIGNED_BYTE, myTexels);
```

- Set minification and magnification filtering modes to `gl.NEAREST`.
- Set *uv*-to-*st* wrapping mode in the *s* and *t* directions to `gl.REPEAT`.
- Create attribute and varying variables for the texture coordinates in your shaders.
- Create a uniform sampler2D and set it to the integer 0 in JavaScript.
- Use the fragment shader to texture your quad:

```
uniform sampler2D texMap;
varying vec2 fTexCoord;
void main() {
  gl_FragColor = texture2D(texMap, fTexCoord);
}
```



Texture filtering (bilinear interpolation)



aliasing due to magnification

aliasing due to minification

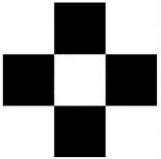
- When texels are seen up close, we see staircase artifacts.
- When texels are seen from afar, we don't see the right pattern.
- Filtering techniques:

  Use `gl.LINEAR`
  for bilinear filtering.

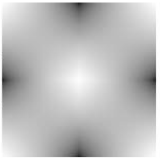
texel values

1	0	1
0	1	0
1	0	1

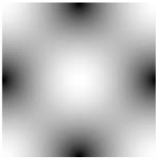
nearest neighbor



bilinear



Hermite

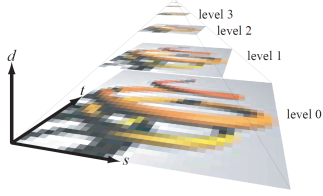


Texture mipmapping

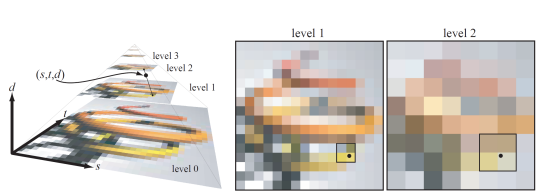
- Linear filtering has little effect on minification issues.
- When a texture has a resolution that is powers of two, we can use a box filter.


```
gl.generateMipmap(gl.TEXTURE_2D);
```

- We can apply the filter iteratively until we reach a level where to the number of texels is similar to those covered by a pixel.



`gl.NEAREST_MIPMAP_NEAREST`



`gl.LINEAR_MIPMAP_LINEAR`

References

- Lance Williams. Pyramidal parametrics. Computer Graphics (SIGGRAPH '83) 17(3), pp. 1–11, July 1983.
- Tomas Akenine-Möller. Some notes on Graphics Hardware. Course notes, Lund University, November 2012.

Exercise: try different wrapping and filtering techniques (W06P2)

- We have the following texture wrapping options:
  - `gl.CLAMP_TO_EDGE`
  - `gl.REPEAT`
- For magnification, we have the following filtering options:
  - `gl.NEAREST`
  - `gl.LINEAR` (`gl.NEAREST + 1`)
- For minification, we have the following filtering options:
  - `gl.NEAREST`
  - `gl.LINEAR` (`gl.NEAREST + 1`)
  - `gl.NEAREST_MIPMAP_NEAREST`
  - `gl.LINEAR_MIPMAP_NEAREST` (`gl.NEAREST_MIPMAP_NEAREST + 1`)
  - `gl. NEAREST_MIPMAP_LINEAR` (`gl.NEAREST_MIPMAP_NEAREST + 2`)
  - `gl.LINEAR_MIPMAP_LINEAR` (`gl.NEAREST_MIPMAP_NEAREST + 3`)
- Create buttons and selection menus, so that the user can switch between these texturing modes.
- Remember to request a new animation frame when the user selects a new mode.

Spherical inverse mapping

- Spherical coordinates provide a uv-mapping of the unit sphere: $(u, v) = \left(1 - \frac{\varphi}{2\pi}, \frac{\theta}{\pi}\right)$.
- The corresponding Euclidean space coordinates are: $(x, y, z) = (\sin\theta\cos\varphi, \sin\theta\sin\varphi, \cos\theta)$.
- Alternatively, if we want y to be the up direction, as in eye space (and in the figure to the right), then $(x, y, z) = (\sin\theta\cos\varphi, \cos\theta, \sin\theta\sin\varphi)$.
- Inserting u and v, we have the uv-mapping $(x, y, z) = (\sin(\pi v)\cos(2\pi(1-u)), \cos(\pi v), \sin(\pi v)\sin(2\pi(1-u)))$.
- The inverse mapping provides texture coordinates given a position on the unit sphere (such as a surface normal).
- We have $y = \cos(\pi v)$ and $\frac{z}{x} = \tan(2\pi(1-u))$, then $u = 1 - \frac{\tan^{-1}\frac{z}{x}}{2\pi} = 1 - \frac{\text{atan}(z,x)}{2\pi}$ and $v = \frac{\cos^{-1} y}{\pi}$.