M30242 – Graphics and Computer Vision

Lecture 06: Texture Mapping

Use Textures

- One of the dilemma in computer graphics is visual fidelity vs. rendering speed.
  - With refined geometric models, better visual fidelity will be achieved, but the rendering will take longer.
  - With coarse models, faster rendering is possible, but the quality will suffer.
- Another problem is some objects, especially the surface properties of the objects, are difficult to model.
- Texture mapping is a techniques for solving the modelling and rendering problems of such objects.

2D Texture

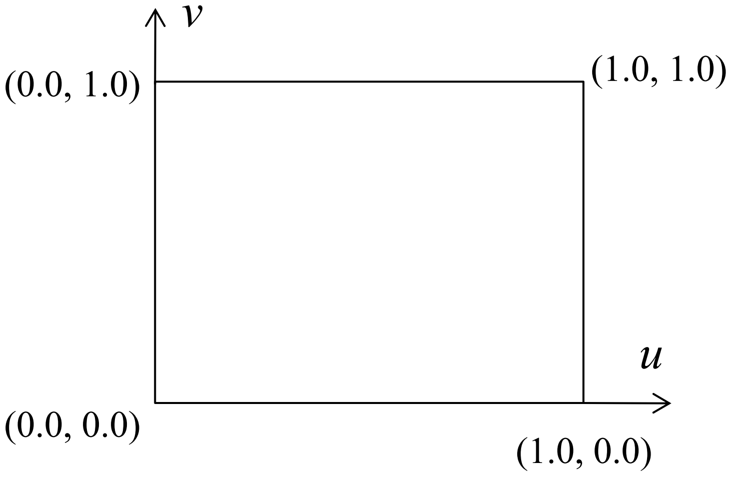
- The most basic form of texture mapping uses a single image as a texture – 2D texture.
- Regardless of what being used as textures, coordinate systems are needed to specify where texels should be sampled from the texture and where on the object they should appear.
- Texels may be regarded as the pixels of a texture, but sometimes *a texel may consists of many pixels of a texture image*.
- To fetch a texel from the texture is called texture sampling.

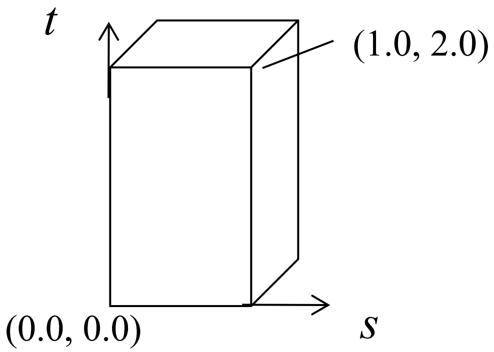
Texture Coordinates

- To map a texture to a surface, we need to establish the correspondence between the texels on the texture and the pixels (fragments) on the surface of a model.
- This is done by specifying texture coordinates on both the texture image and the object model.
- The texture coordinates for a texture are usually represented by a pair of values, (u, v), and called u, v coordinates.
- The texture coordinates for a model (it might be a surface of a single or a few polygons) is called s, t coordinates and s corresponds to u and t corresponds to v.

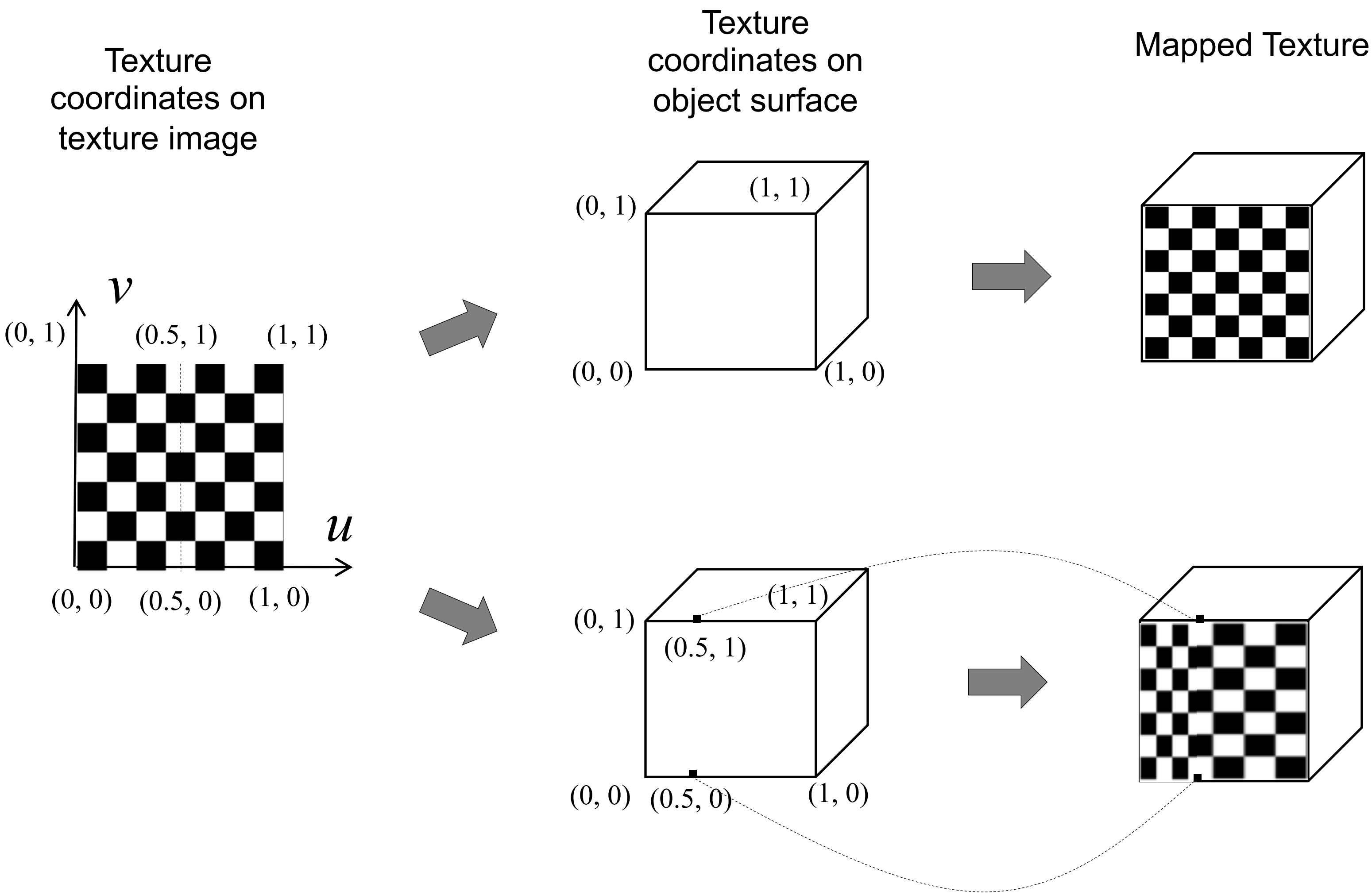
Cont’d

- Texture coordinates for a texture, (u, v), are normalised coordinates:
  - The lower-left corner of the texture is defined as the origin and has the coordinates (0.0, 0.0).
  - The upper-right corner of the texture always has the coordinates (1.0, 1.0), regardless of its size or whether the texture is a square.
- In contrast, the texture coordinates on a model, (s, t), could be greater than 1.0, e.g., (0.0, 2.0). In this case, texture will be repeated (or tiled, as it is called).





Map (u,v) to (s, t)

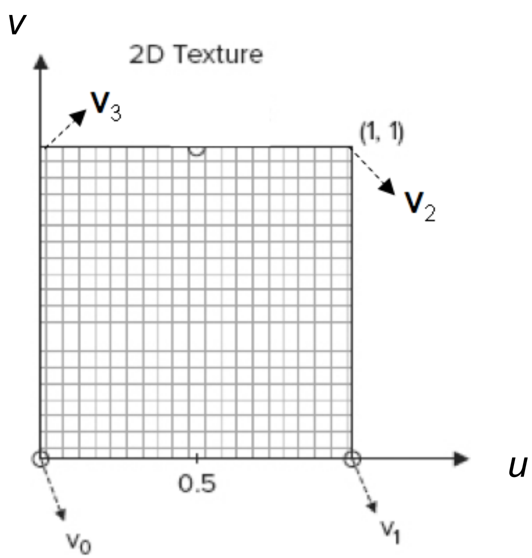


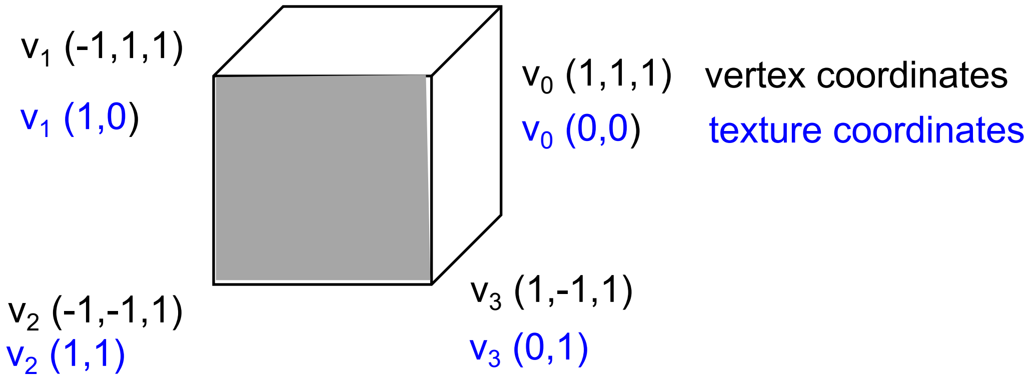
The result of mapping when texture coordinates are linearly interpolated.

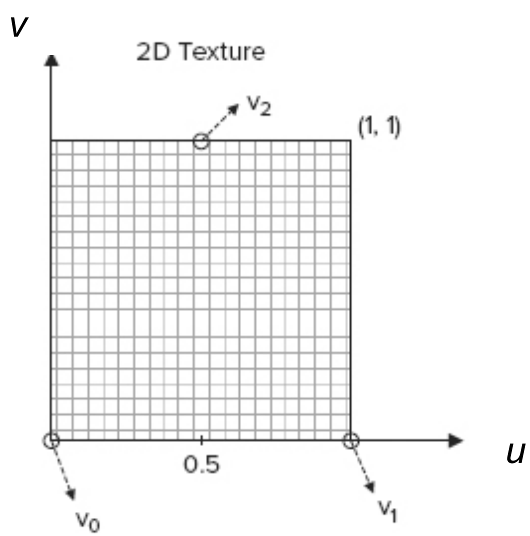
Cont’d

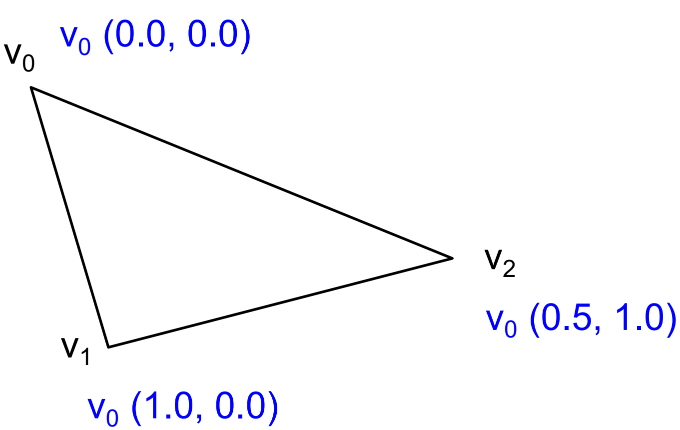
- The example shows clearly how the assignment of texture coordinates on the vertices affects the texture being mapped.
- In general, it is difficult, or even impossible, to achieve uniform, distortion-free texture mapping on curved surfaces.
- Texture mapping algorithms that produce acceptable visual effect for some common shapes, such as triangle, cube, cylinder, sphere, etc, are available and implemented in graphics libraries and/or APIs.

Assign Texture Coordinates



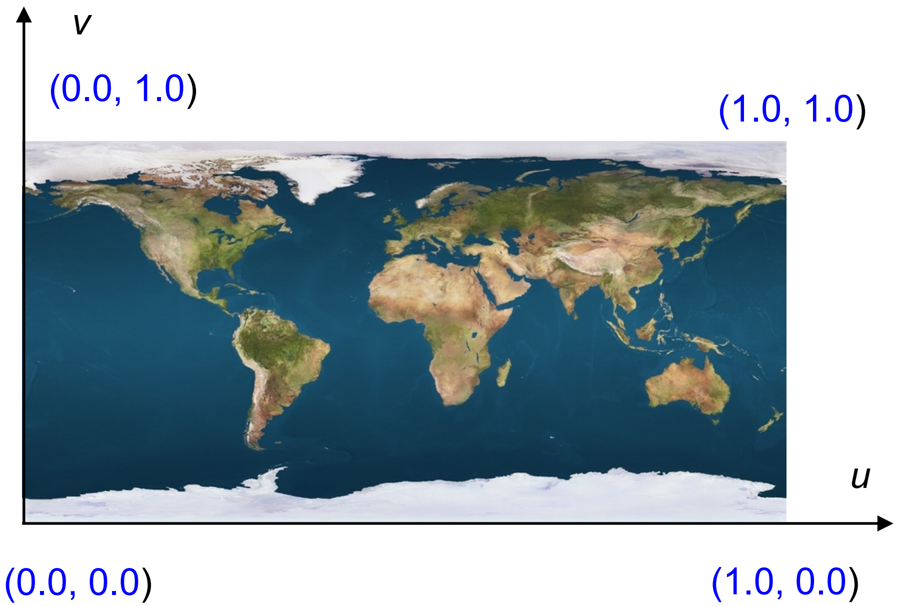


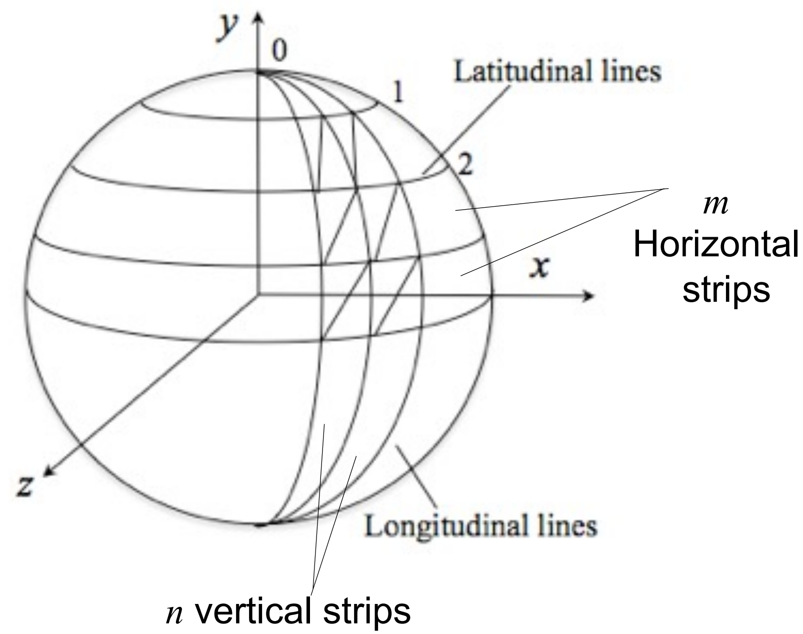




How Would Assign Texture Coordinates for a Sphere?

Revise Lecture 4 on sphere tessellation





Cubemap Textures

- In addition to the usual 2D textures, WebGL also supports a feature called cubemap textures.
- A cubemap texture is handled as a single texture object but is composed of six **square** textures each representing a face of a cube.

WebGLTexture Objects

- The first step of texture mapping in WebGL is to **create a WebGLTexture object for each of the textures used.** A texture object is a container object which the actual texture image/video can be uploaded to.
- WebGL provides two functions for creating and deleting texture objects:
  - `gl.createTexture()` and
  - `gl.deleteTexture(texture).`
- A texture object is created by the function call, e.g.,

```
var texture = gl.createTexture();
```

- The following call deletes a WebGLTexture object named `aTexture`:

```
gl.deleteTexture(aTexture);
```

- You don't have to delete a texture object when you have finished using it. The texture object will be deleted by the JavaScript garbage collection. The method only gives you a greater control over when a texture object is destroyed.

Binding Texture

- After creation, a WebGLtexture object must be bound to a target (i.e., specifying the texture type):

  2D TEXTURE or

  CUBEMAP texture.

- E.g., to bind a WebGLTexture object `texture` as a 2D texture:

```
var texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);
```

Cont'd

- The prototype of `gl.bindTexture()` is as the following:

```
void bindTexture(GLenum target, WebGLTexture texture);
```

  where `target` could be
  - **`gl.TEXTURE_2D`** for 2D texture, or one of the following for cubemap:
  - **`gl.TEXTURE_CUBE_MAP_POSITIVE_X`**
  - **`gl.TEXTURE_CUBE_MAP_NEGATIVE_X`**
  - **`gl.TEXTURE_CUBE_MAP_POSITIVE_Y`**
  - **`gl.TEXTURE_CUBE_MAP_NEGATIVE_Y`**
  - **`gl.TEXTURE_CUBE_MAP_POSITIVE_Z`**
  - **`gl.TEXTURE_CUBE_MAP_NEGATIVE_Z`**

Set Texture Data & Filters

- After binding a WebGLTexture object to a target, the next step is to load the actual image data and set texture filtering parameters:
  - Texture loading: `texImage2D()` and
  - Texture filtering `texParameteri()`.

Loading Image

- Actual texture data can be loaded into the bound texture object by using method `gl.texImage2D()`.

  ```
  void texImage2D(GLenum target, GLint level, GLenum
      internalformat, GLenum format, GLenum type,
      TextureType texture) raises (DOMException)
  ```

  The meaning of the other arguments are:
  - **target** is either **`GL_TEXTURE_2D`** or **`GL_TEXTURE_CUBE_MAP`**.
  - **level** specifies the mipmap level (discussed later).
  - **internalformat** and **format** specify the formats of the texture. In WebGL they must be the same, e.g., `gl.RGBA`.
  - **type** specifies the data type to store the data for the texels.
  - The last argument, **TextureType**, could take one of the following forms:
    - **`HTMLImageElement image`**, when the texture data is an HTML image object.
    - **`HTMLCanvasElement canvas`** when texture data is an HTML5 canvas element.
    - **`HTMLVideoElement video`** when texture data is a video element.

Cont'd

- For example

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

loads an HTML image object, `image`, as **`gl.TEXTURE_2D`** texture in the format of `gl.RGBA` and stores the texel using **`gl.UNSIGNED_BYTE`** (each texel occupies four bytes of memory)

HTML Image Object

- HTML image objects can be created by using `<img>` tag in an HTML document, e.g.,

```
<img id="aName" src="someimage.png" >
```

- or alternatively in JavaScript

```
var image = new Image();//create a JavaScript image object
image.onload = function(){
   gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true)
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
                                   gl.UNSIGNED_BYTE, image);
}
image.src = "someimage.png";
```

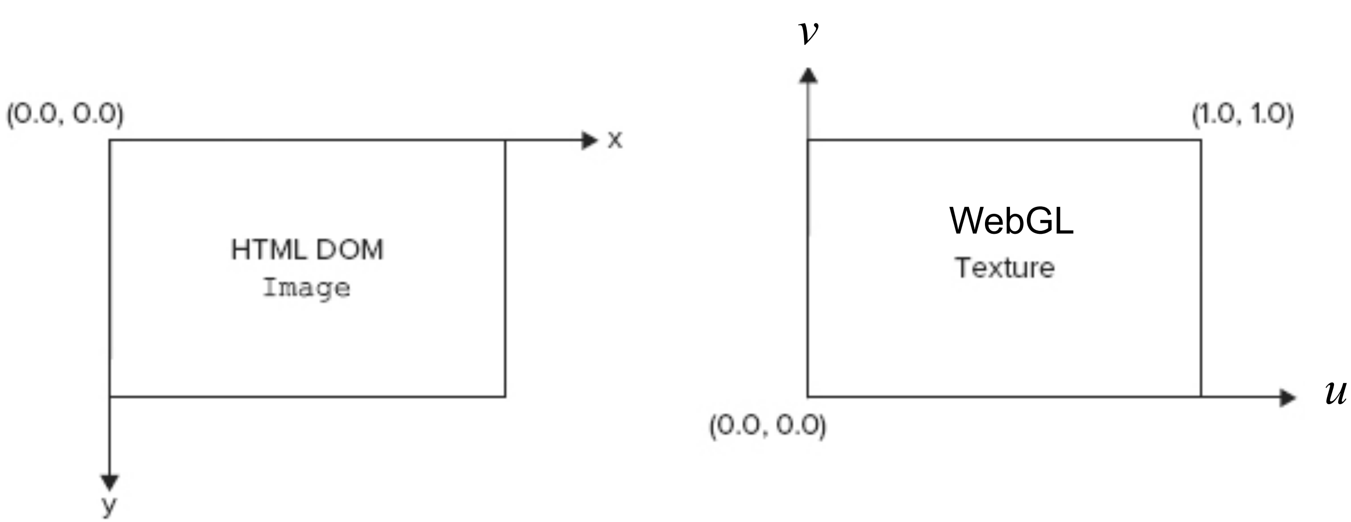
A JavaScript image object is created first. As soon as the URL is assigned to the `src` property, the image is loaded asynchronously. When the image has finished loading, the `onload` event triggers the anonymous function, which load the texture image to GPU.

- Notice that, before the texture data is loaded into the texture, there is a call:

```
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
```

The first argument, `gl.UNPACK_FLIP_Y_WEBGL` sets that the image (its y-axis) being flipped around the x- (horizontal) axis.

- The reason is that the coordinate system used for textures in WebGL (and all versions of OpenGL as well) is different from the coordinate system used for the *Image* object in HTML.



Shaders Setup

- Both shaders must be prepared for texture mapping.
- Vertex shader

```
<script id="shader-vs" type="x-shader/x-vertex">
   attribute vec3 aVertexPosition;
   attribute vec2 aTextureCoordinates; //input
   varying vec2 vTextureCoordinates;   //vertex texture coord
   void main() {
     gl_Position = vec4(aVertexPosition, 1.0);
     vTextureCoordinate = aTextureCoordinates;
   }
</script>
```

- Fragment shader

```
<script id="shader-fs" type="x-shader/x-fragment">
   precision mediump float;
   varying vec2 vTextureCoordinates; //fragment texture coord.
   uniform sampler2D uSampler;
   void main() {
     gl_FragColor = texture2D(uSampler, vTextureCoordinates);
   }
</script>
```

Although their names are the same, the meanings are different: In fragment shader, the texture coordinates are the interpolated values for a fragment

Shaders Setup

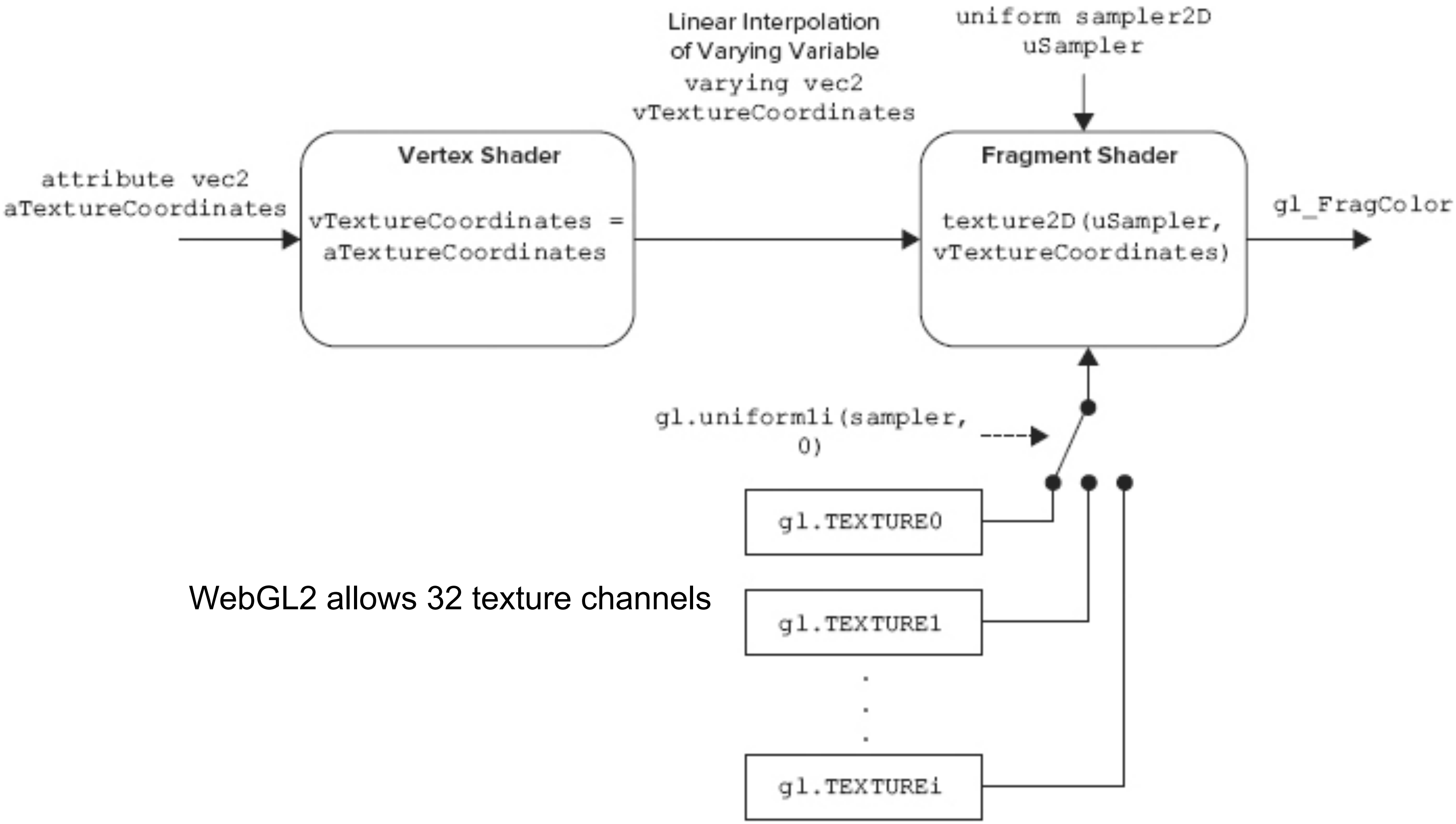
- The way of using texture in vertex shader is very similar to using colour:
  - A variable of **attribute** type `aTextureCoordinates` of the type `vec2` is used for input of texture coordinates
  - a variable of **varying** type `vTextureCoordinates` of the type `vec2` is used for output of texture coordinates to the fragment shader.
- In the fragment shader,
  - the **varying** variable `vTextureCoordinates` is used to fetch texture coordinates of a fragment.
  - The special **uniform** variable `uSampler` of type `sampler2D` is declared. The type `sampler2D` is a type of OpenGL Shading Language for handling/sampling texture data.
  - In the main method, GLSL function `texture2D(),` using the texture coordinates and the sampler, returns a vec4 representing the color fetched from the texture for the current fragment.

In Main WebGL Program

- To use the uniform `uSampler` to sample a texture, we need to get its pointer in the shader program and specify an active texture unit (channel) in WebGL (an ID, e.g., gl.TEXTURE0, …, gl.TEXTURE31).
- After binding the texture to the active texture unit, assign the texture unit ID (e.g., 0) to the sampler using the method `gl.uniform1i()`:

```
// Get the location of the uniform uSampler
uniformSamplerLoc = gl.getUniformLocation(shaderProgram, "uSampler");
...
// make a texture unit, e.g., gl.TEXTURE0 active
gl.activeTexture(gl.TEXTURE0);
// bind the texture
gl.bindTexture(gl.TEXTURE_2D, texture);
...
// Set uSampler in fragment shader to have the value 0
// so that it matches the texture unit gl.TEXTURE0
gl.uniform1i(uniformSamplerLoc, 0);
```

Active texture ID



WebGL2 allows 32 texture channels

Texture Filtering

- Texture mapping produces good results only if the size and resolution of a texture matches those of the object which the texture is mapped to.
- Consider a texture image of the size 512×512 is mapped to a square surface (of any size). If the size of the rendered image of the square is close to 512×512 pixels, the texture will look accurate and natural. Otherwise, some visual defects will appear.
- If the rendered image of a surface is bigger (smaller) than the texture image, the texture will need to be stretched (compressed) to properly cover the surface. This change to texture size is called magnification (minification).
- The process of calculating pixel colours from the texels of a magnified or minified texture is called texture filtering.
- The aim of texture filtering is to decide the “best” colour for a pixel in such cases.

Texture Filtering in WebGL

- Texture filtering control is done by calling the method `gl.texParameteri()`. The syntax of this function is

  `void texParameteri(GLenum` **`target`**`, GLenum` **`pname`**`, GLint` **`param`**`)`

  where
  - `target` can be either **`gl.TEXTURE_2D`** or **`gl.TEXTURE_CUBE_MAP`**.
  - `pname` specifies the parameter that you want to set.
  - `param` contains the value for the parameter you want to set.

- For example, to specify a minification filter, we set

  `pname` to `gl.TEXTURE_MIN_FILTER` and

  `param` to `gl.LINEAR`.

- Possible filters are:
  - *Magnification* filter: `gl.TEXTURE_MAG_FILTER`,
  - *Minification* filter: `gl.TEXTURE_MIN_FILTER`,
  - *Wrapping* filters: `gl.TEXTURE_WRAP_S`, or `gl.TEXTURE_WRAP_T`
- Allowed parameter values :

For *magnification* filter:

- `gl.NEAREST`
- `gl.LINEAR`

For *minification* filter:

- `gl.NEAREST`
- `gl.LINEAR`
- `gl.NEAREST_MIPMAP_NEAREST`
- `gl.LINEAR_MIPMAP_NEAREST`
- `gl.NEAREST_MIPMAP_LINEAR`
- `gl.LINEAR_MIPMAP_LINEAR`

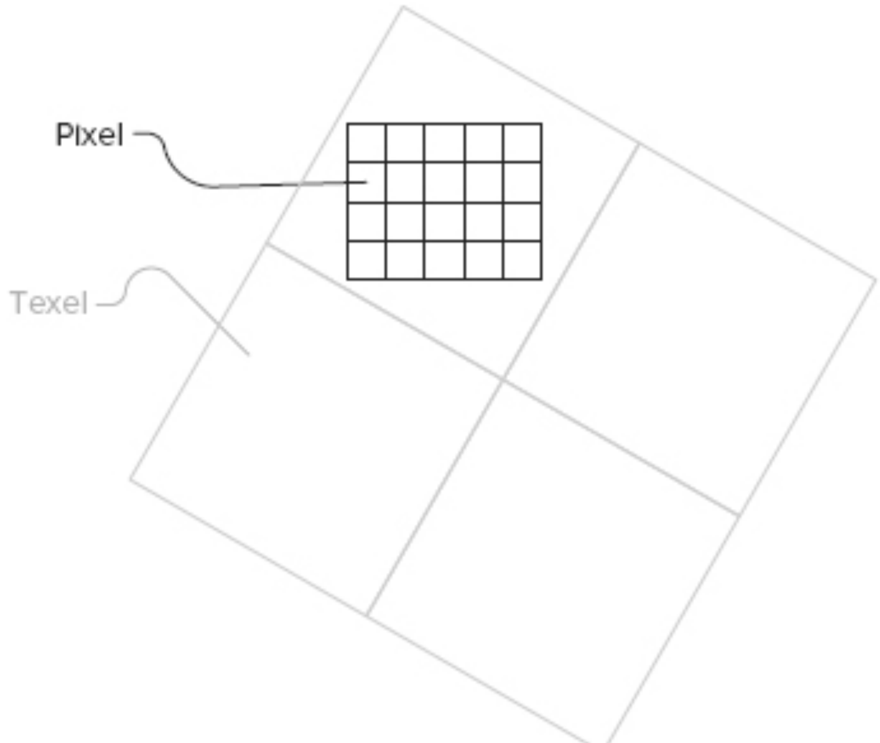
These are mipmap filters

For *wrapping* filter

- `gl.REPEAT`
- `gl.CLAMP_TO_EDGE`
- `gl.MIRRORED_REPEAT`

Magnification

- In the case of texture magnification, the texture appears, visually, to have been magnified. That is, one texel covers (corresponds to) several rendered/on-screen pixels.

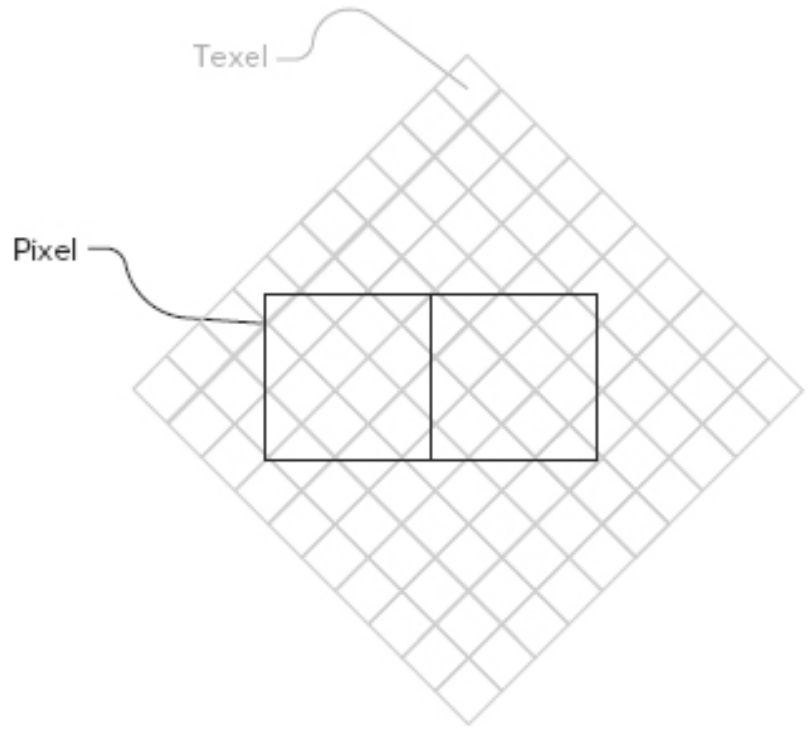


Filtering: Nearest or Linear

- The nearest filter takes the colour of the nearest texel to as the colour of the pixel.
  - This filtering is fast since only one texel is considered in colour calculation.
  - Many pixels fall upon one texel and they all take the same colour, which would result in a blocky appearance – an effect called pixelation.
- The linear filter assigns colour to a pixel by linear filtering or bilinear filtering - it takes a (distance-) weighted average of the four texels surrounding the texture coordinate of a pixel.
  - Computationally more complex than nearest filtering,
  - Alleviates pixelation,
  - But results in blurred texture.

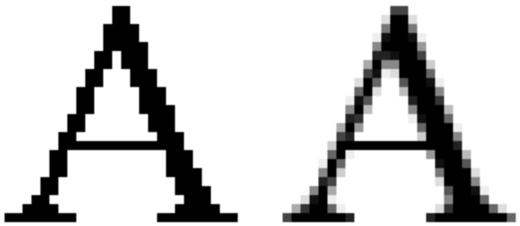
Minification

- In minification, several texels correspond to one pixel on the screen. This means that the colours of several texels would affect the colour of one pixel.



Filtering: Nearest or Linear

- As in the case of magnification, the nearest neighbor filter fetches the colour from the texel that is closest to the current pixel (i.e., its texture coordinates).
- A defect of the nearest filtering is aliasing – the smooth curves, boundaries or lines become jagged.



Aliased and anti-aliased letters

- The linear filter uses the weighted average colour of the four closest texels as the color for a given pixel.
- The linear filter can give a slightly better result than the nearest filter, but the aliasing problem still exists.

Mipmapping

- As the distance between an object and the viewer/camera can change, we cannot decide beforehand when to apply magnification or minification filtering to reduce pixelation or aliasing.
- A better approach is to use a set of textures of different sizes and dynamically choose a texture size that can roughly maintain the one-to-one correspondence between the pixels and the texels.
- This techniques is called mipmapping – using a pre-calculated, optimised collections of images that accompany a main texture, intended to increase rendering speed and reduce aliasing or pixelation artifacts.
- The image collection used in mipmapping is called mipmap chain.
- A mipmap chain has images of several levels of resolutions. In practice, *at each level, the image size is half the size of the previous level*.

Mipmap Chain

- The textures do not have to be square, but the chain continues until the last texture has the size 1 × 1, e.g., a mipmap chain could contain the textures of sizes 256 × 256, 128 × 128, 64 × 64, 32 × 32, 16 × 16, 8 × 8, 4 × 4, 2 × 2, and 1 × 1.

level 0, level 1, level 2, . . .

- A complete mipmap chain occupies approximately *one-third more memory than without mipmapping*.

Use Mipmapping

- WebGL generates the mipmap chain by function call `gl.generateMipmap()`:

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
                                  gl.UNSIGNED_BYTE, image);
gl.generateMipmap(gl.TEXTURE_2D);
```

- If more control over the image series is wanted, the series can be created offline and load each image using `gl.texImage2D()` with the specified the mipmap level:

```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,
                    gl.UNSIGNED_BYTE, imageLevel_0);
gl.texImage2D(gl.TEXTURE_2D, 1, gl.RGBA, gl.RGBA,
                    gl.UNSIGNED_BYTE, imageLevel_1);
gl.texImage2D(gl.TEXTURE_2D, 2, gl.RGBA, gl.RGBA,
                    gl.UNSIGNED_BYTE, imageLevel_2);
                    . . ., 3, . . .
```

Mipmap Filters

- Mipmap filters look like this:

  gl.OPTION1_MIPMAP_OPTION2

  Both OPTION1 and OPTION2 can be NEAREST or LINEAR

  OPTION1 specifies the filtering algorithm **within a level of mapmap**

  OPTION2 specifies the filtering algorithm **between two levels of mapmap**

- For example
  - gl.NEAREST_MIPMAP_NEAREST — Get pixel value from the nearest neighbour *between* the nearest mipmap level.
  - gl.NEAREST_MIPMAP_LINEAR — the pixel value is decided from the linear interpolation of the two nearest neighbours (texel values) from the two closest mipmap levels ()

Cont’d

- More fillters
  - gl.LINEAR_MIPMAP_NEAREST — Linear filtering is applied to the nearest mipmap level.
  - gl.LINEAR_MIPMAP_LINEAR — The two nearby mipmap levels are selected. Within these two levels, linear filtering is used to get an intermediate result within each mipmap level. Linear interpolation is then applied to the two intermediate results to get the final result. This filter typically produces the best result of all the filters. It is referred to as trilinear filtering.

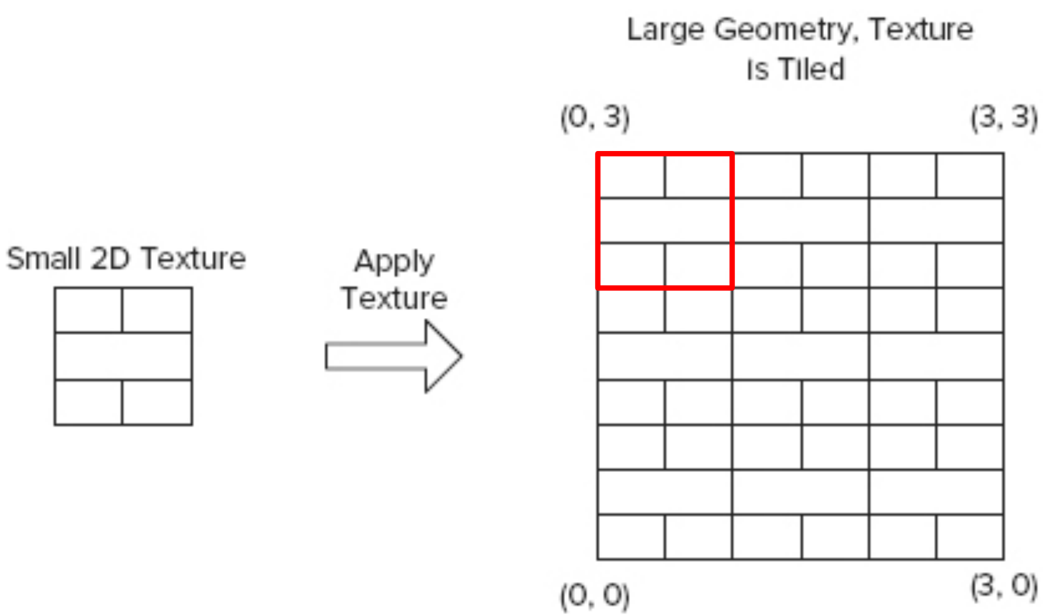
Texture Wrapping

- As discusses previously, a texture is described by a coordinate system: its lower-left corner has the coordinates (0, 0) and the upper-right corner has the coordinates (1, 1), regardless of its size or whether it is a square.
- When the values of texture coordinates on an object are greater than “1.0”, e.g., (0.0, 2.0), the texture has to be repeated. WebGL handles this by using different wrap mode:
  - **`gl.REPEAT`**
  - **`gl.MIRRORED_REPEAT`**
  - **`gl.CLAMP_TO_EDGE`**

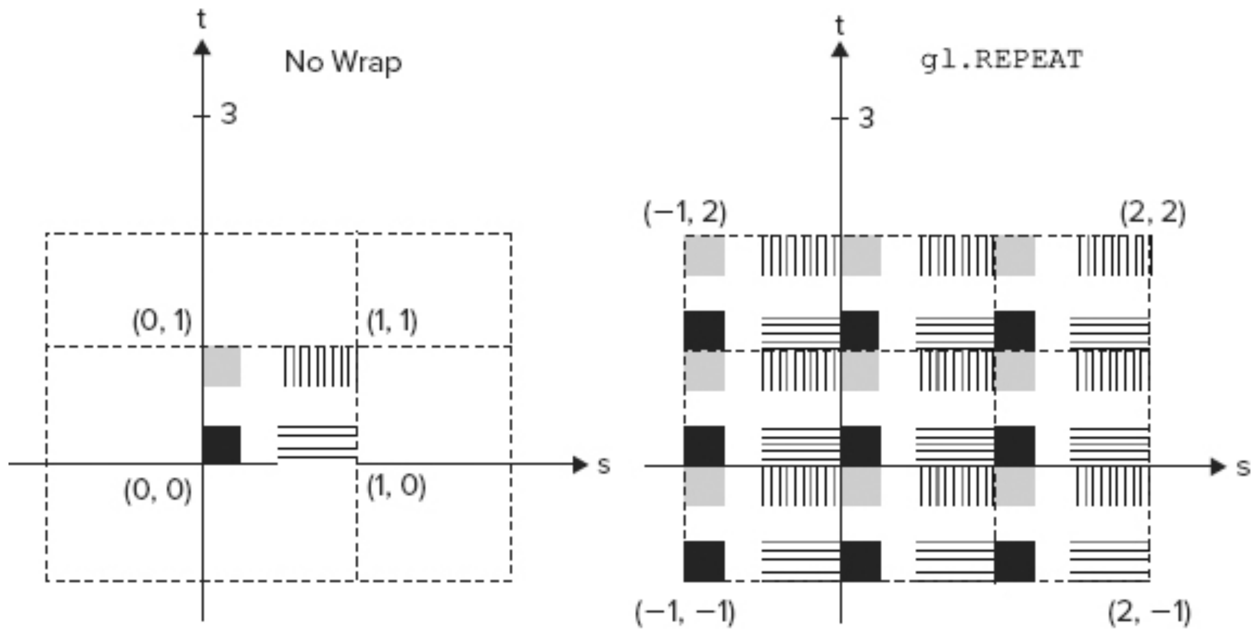
- *Note that independent, different wrap modes can be applied to s-direction and t-direction of the texture coordinates*

Repeat Mode

- E.g., according to the texture coordinates of the vertices provided, the original texture is repeated three times in both s- and t-directions.

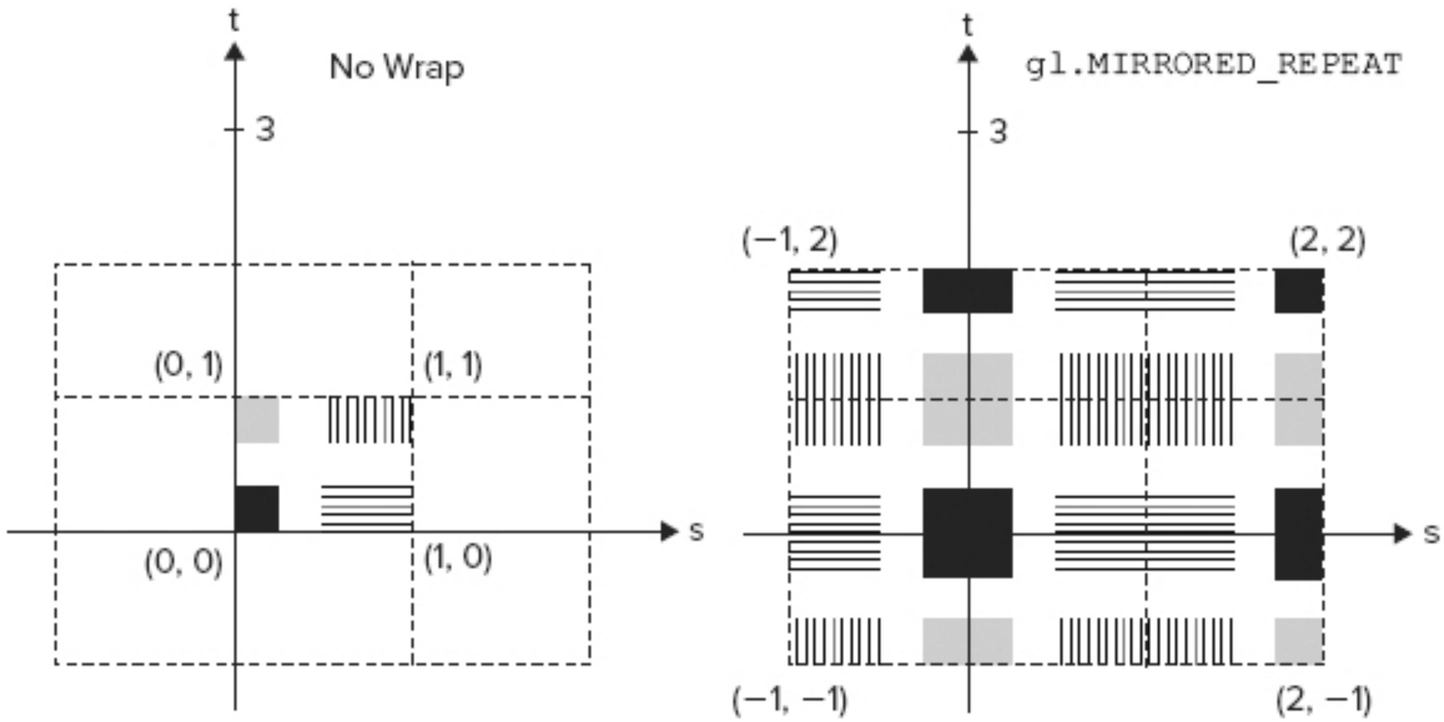


Repeat Mode



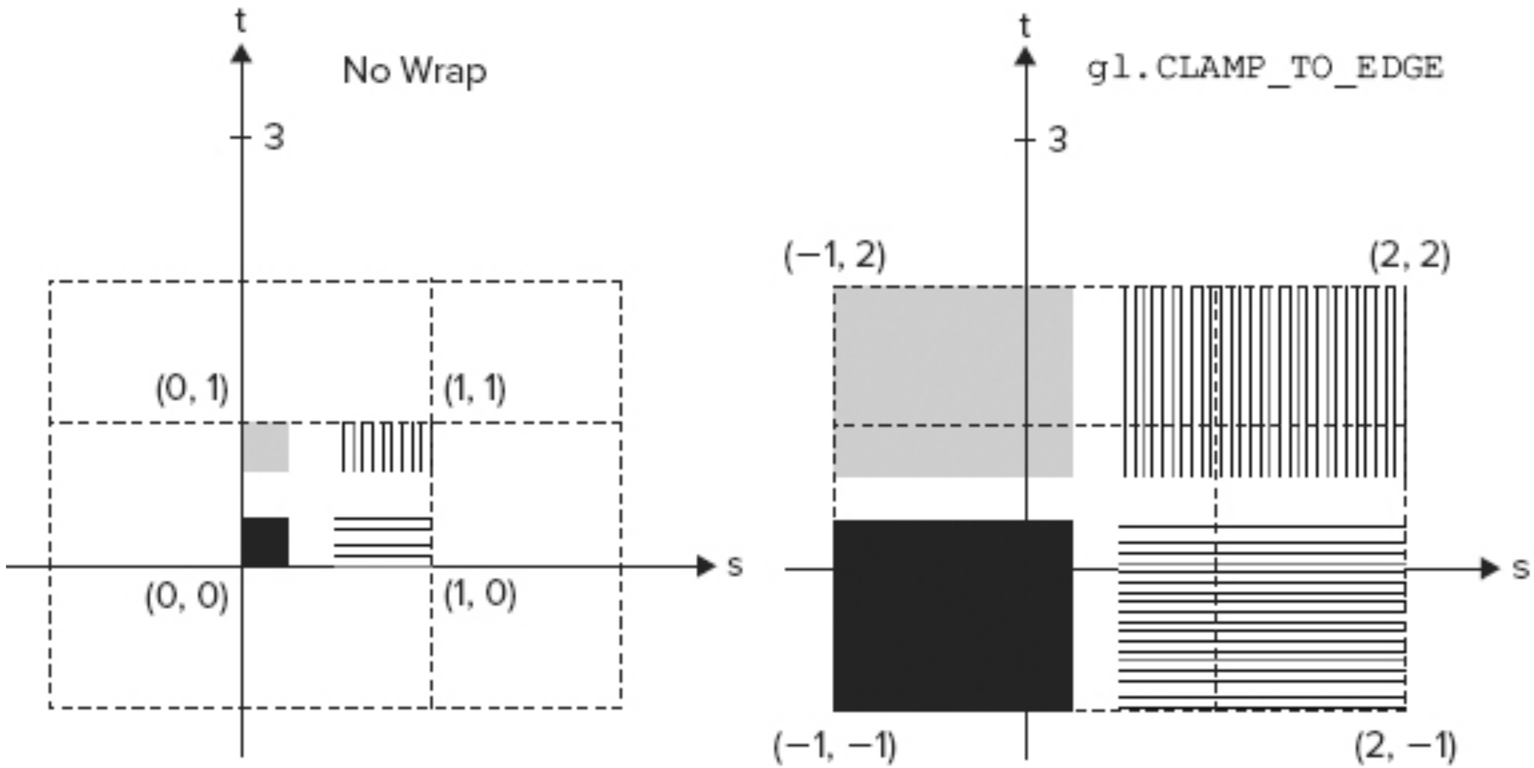
Mirrored-Repeat Mode

- In this mode, texture is mirrored while being repeated:



Clamp to Edge Mode

- In this mode, all texture coordinates are clamped to the range [0, 1]. Texture coordinates that are outside this range will sample from the closest edge of the texture.



Set Wrap Mode

- A wrap mode can be set via gl.texParameteri() by assign `pname` to **`gl.TEXTURE_WRAP_S`** or **`gl.TEXTURE_WRAP_T`** and `param` an appropriate mode.

- E.g., 1

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
```

- E.g., 2

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S,
   gl.MIRRORED_REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T,
   gl.MIRRORED_REPEAT);
```

- E.g., 3

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
```

Further Reading

- Anyuru, A., WebGL Programming – Develop 3D Graphics for the Web
  - Chapter 5