02562 Rendering - Introduction

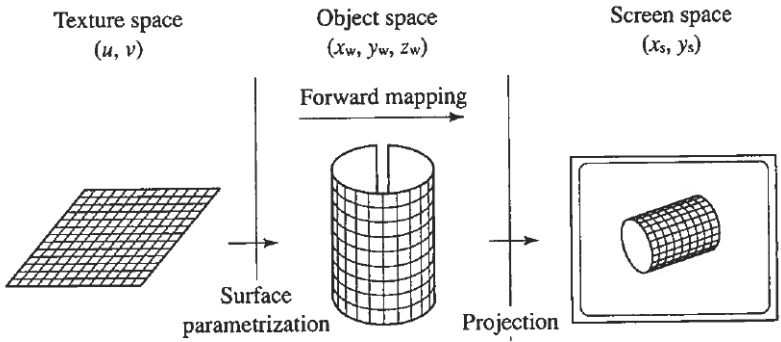
Texture Mapping

Jeppe Revall Frisvad

September 2023

Texture in computer graphics

- A texture is a 1- to 3-dimensional array of vector elements.
- Each element in the array is called a texel (texture element).
- Texture mapping is to map the texels onto an object in 3D space and then project them back into 2D image space.

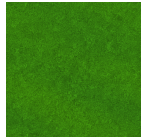


- The texels are used to modulate the object properties.

Obtaining textures

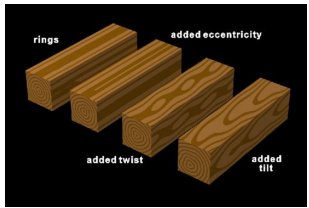
- Textures are usually loaded from image files.
- There are numerous online resources

  (e.g. https://texturehaven.com/, https://www.textures.com/)



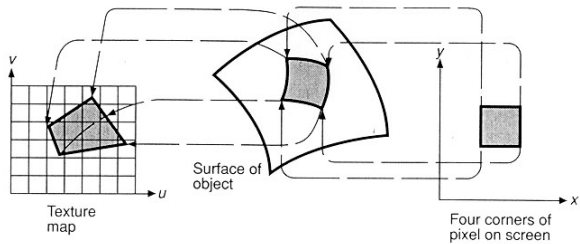
- If you want to repeat the texture across a surface, you need a *tileable* texture.
- Tileable versus non-tileable textures:

- It is also possible to generate textures procedurally.
- Example of procedural solid texturing:



Texture coordinates

- We use texture coordinates (u, v) to map a 2D texture onto the surface of a 3D object.



- This is a parametrization of the surface, where the parameters (u, v) map to a point on the surface in 3D space.
- The parametrization of a plane is $\boldsymbol{x} = \boldsymbol{x}_0 + u\vec{b}_1 + v\vec{b}_2$, where $\vec{n} = \vec{b}_1 \times \vec{b}_2$ is the normal of the plane while $\vec{b}_1$ and $\vec{b}_2$ are orthogonal vectors (tangent and binormal) in the plane.
- In ray tracing, we find a ray-surface intersection point $\boldsymbol{x}$. To find the texture coordinates (u, v) of an intersection point $\boldsymbol{x}$, we need an inverse mapping.

Mapping examples

- Planar map:
  $(u, v) = s(x, y)$

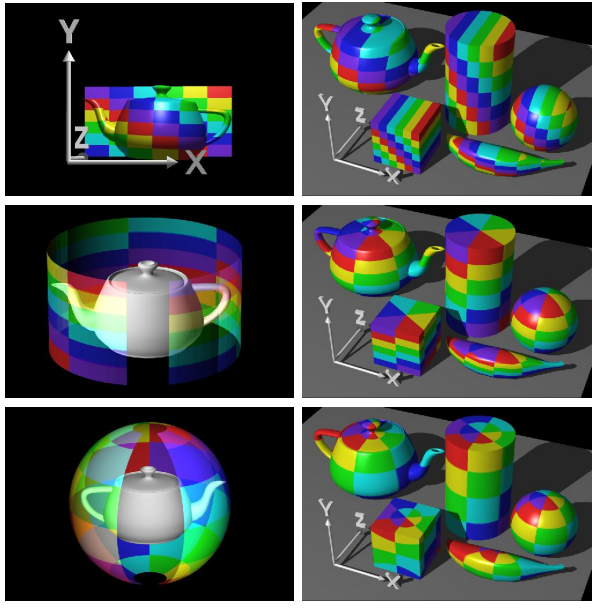
  s is a scale.

- Cylindrical map:
  $(u, v) = (\frac{\theta}{2\pi}, h)$

  $\theta \in [0, 2\pi]$, $h \in [0, 1]$.

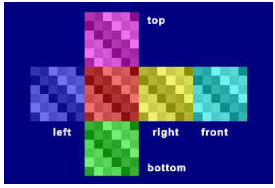
- Spherical map:
  $(u, v) = (\frac{\theta}{\pi}, \frac{\varphi}{2\pi})$

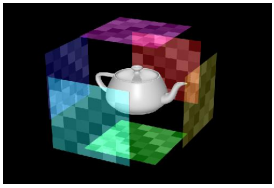
  $\theta \in [0, \pi]$, $\varphi \in [0, 2\pi]$.

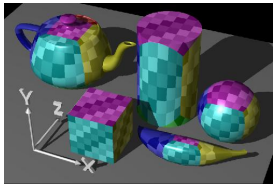


Cube map and environment mapping

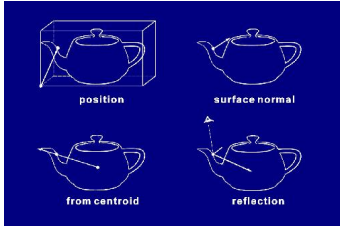
- Cube map:



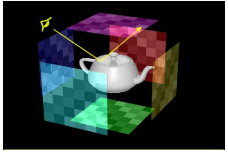




- Choosing a map entity (examples):
  - position relative to the object's bounding box,
  - surface normal at the point being rendered,
  - vector from the centroid through the point,
  - reflection vector at the current point.



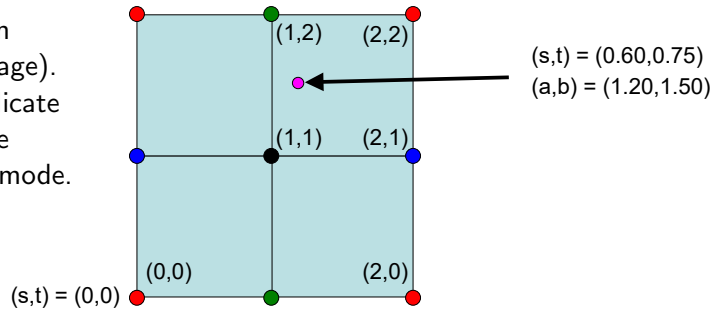
- Environment mapping
  - Using ray direction for look-ups when no object was hit.
  - Using the reflection entity for reflective objects.



Texture look-up

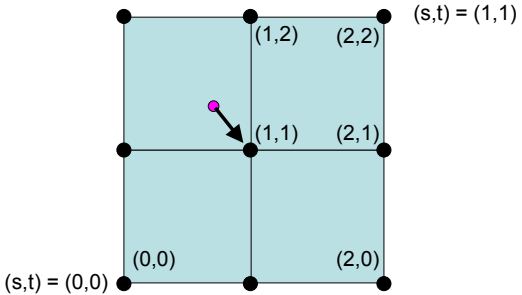
- From texture coordinates (u, v) to texture image index i.
- Two new sets of coordinates:
  - (s, t) texture space $[0, 1] \times [0, 1]$.
  - (a, b) image space $[0, W] \times [0, H]$, where $W \times H$ is the image resolution.
- In texture repeat mode: $s = u - \text{floor}(u)\,, \quad t = v - \text{floor}(v)\,.$

  $a = s * W\,, \qquad b = t * H\,.$
- Illustration (2×2 image). Colors indicate texel reuse in repeat mode.

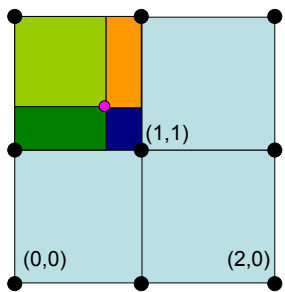


Texture filtering

- Find index i of nearest texel (nearest neighbor filtering):
  $U = (\text{int})(a + 0.5) \bmod W\,, \quad V = (\text{int})(b + 0.5) \bmod H\,, \quad i = U + V * W\,.$
- Find four nearest texels, do weighted average (bilinear filtering):
  $U = (\text{int})a\,, \quad V = (\text{int})b\,, \quad c_1 = a - U\,, \quad c_2 = b - V\,,$
  where c_1 and c_2 are the weights in the bilinear interpolation.



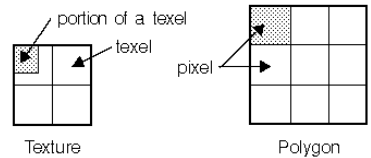
nearest



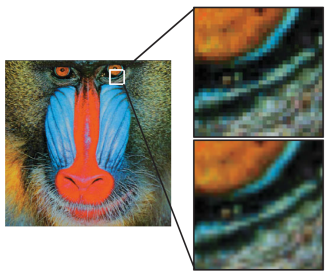
linear

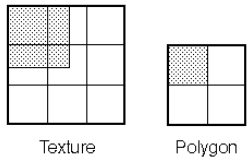
Texture aliasing

- Individual texels rarely correspond to individual pixels of the final image.
  - Magnification is when a texel covers multiple pixels.
  - Minification is when a pixel covers multiple texels.

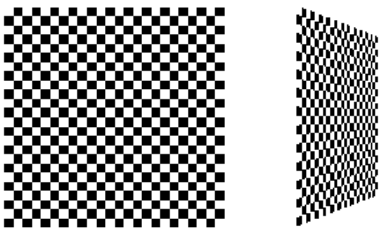


Magnification





Minification

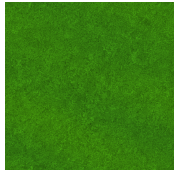


Drawing a texture using WebGPU

- A minimalistic fragment shader:

```
@group(0) @binding(0) var my_sampler: sampler;
@group(0) @binding(1) var my_texture: texture_2d<f32>;

@fragment
fn main_fs(@location(0) coords: vec2f) -> @location(0) vec4f
{
  let uv = coords*0.5 + 0.5;
  return textureSample(my_texture, my_sampler, uv);
}
```



- Setup on the JavaScript side:

```
const texture = await load_texture(device, "data/grass.jpg");
const bindGroup = device.createBindGroup({
  layout: pipeline.getBindGroupLayout(0),
  entries: [
    { binding: 0, resource: texture.sampler },
    { binding: 1, resource: texture.createView() }
  ],
});
function animate() { render(device, context, pipeline, bindGroup); }
animate();
```

- The `render` function creates a command encoder to set and submit a render pass.
- Call with a different `bindGroup` using a different `sampler` to change texture mode.

Loading a texture for WebGPU

- Loading is asynchronous and uses HTML elements:

```
async function load_texture(device, filename)
{
  const img = document.createElement("img");
  img.src = filename;
  await img.decode();

  const imageCanvas = document.createElement('canvas');
  imageCanvas.width = img.width;
  imageCanvas.height = img.height;
  const imageCanvasContext = imageCanvas.getContext('2d');
  imageCanvasContext.drawImage(img, 0, 0, imageCanvas.width, imageCanvas.height);
  const imageData = imageCanvasContext.getImageData(0, 0, imageCanvas.width, imageCanvas.height);
```

- Copying the image data to a `Uint8Array` (reversing the vertical axis):

```
  let textureData = new Uint8Array(img.width*img.height*4);
  for(let i = 0; i < img.height; ++i)
    for(let j = 0; j < img.width; ++j)
      for(let k = 0; k < 4; ++k)
        textureData[(i*img.width + j)*4 + k] = imageData.data[((img.height - i - 1)*img.width + j)*4 + k];
```

- Streaming the image data to the GPU:

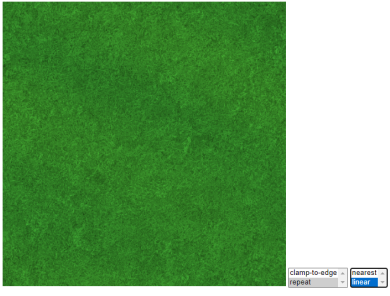
```
  const texture = device.createTexture({
    size: [img.width, img.height, 1],
    format: "rgba8unorm",
    usage: GPUTextureUsage.COPY_DST | GPUTextureUsage.TEXTURE_BINDING
  });
  device.queue.writeTexture({ texture: texture }, textureData,
    { offset: 0, bytesPerRow: img.width*4, rowsPerImage: img.height, }, [img.width, img.height, 1]);
```

Texture addressing and filtering modes

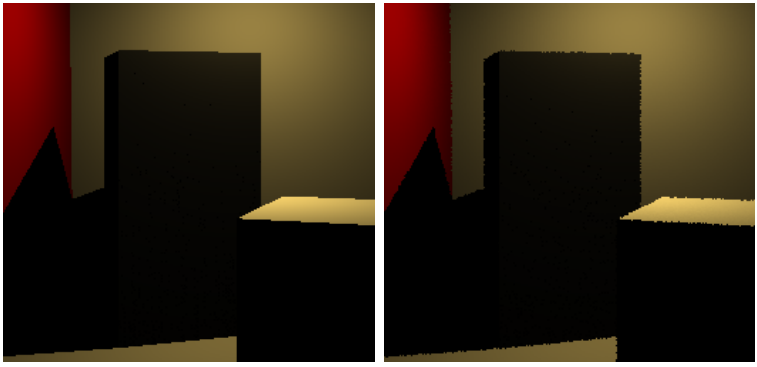
- Conclude the `load_texture` function by creating a sampler:

```
  texture.sampler = device.createSampler({
    addressModeU: "clamp-to-edge",
    addressModeV: "clamp-to-edge",
    minFilter: "nearest",
    magFilter: "nearest",
  });
  return texture;
}
```

- Another option for an address mode is `repeat`.
- Another option for a filter mode is `linear`.
- Create an array of samplers if you would like to switch between different modes.

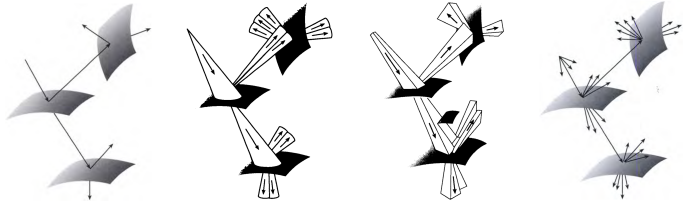


Aliasing artifacts



- Staircase artifacts ('jaggies') are obvious for hard shadows, but occur for all edges.
- Regularity artifacts appear when we always trace our rays through the pixel center.
- Trying with rays through random pixel positions leads to noise. It's a trade-off.
- How do we find out if a pixel covers multiple surfaces or texels?

Pixel coverage?

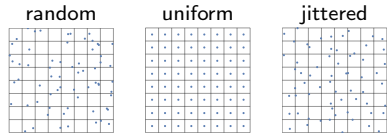


- Cone tracing. A cone (or pyramid) with apex at the eye traced through each pixel. Cone-object intersections for anti-aliasing [Amanatides 1984].
- Beam tracing. One beam for the view frustum. Split into a new beam for every polygon intersection [Heckbert and Hanrahan 1984].
- Stochastic sampling. Multiple randomly directed rays through each pixel [Cook 1986]. What is the best sampling procedure?

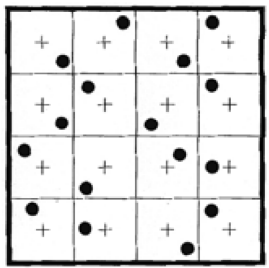
References

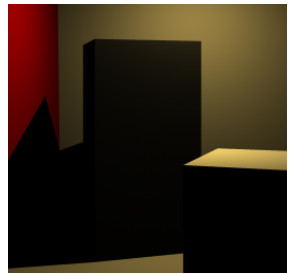
- Heckbert, P. S., and Hanrahan, P. Beam tracing polygonal objects. *Computer Graphics (SIGGRAPH 84) 18*(3), pp. 119–127. July 1984.
- Amanatides, J. Ray tracing with cones. *Computer Graphics (SIGGRAPH 84) 18*(3), pp. 129–135. July 1984.
- Cook, R. L. Stochastic sampling in computer graphics. *ACM Transactions on Graphics 5*(1), pp. 51–72. January 1986.
- Glassner, A. S. *Principles of Digital Image Synthesis*. Morgan Kaufmann. 1995.

Stratified jitter sampling



- Random numbers have a tendency to cluster.
- Stratified sampling: place samples in a grid of subpixels (substrata).
- Jitter the sample in each subpixel to avoid aliasing.





- How to make it practical when the renderer traces rays through pixel centers?
- 2D displacement vectors in the image plane. One for each subpixel.
- Using the same pattern for each pixel reduces noise.

Exercises

- Define the plane by an orthonormal basis $(\vec{b}_1, \vec{b}_2, \vec{n})$:

```
struct Onb {
  tangent: vec3f,
  binormal: vec3f,
  normal: vec3f,
};
const plane_onb = Onb(vec3f(-1.0, 0.0, 0.0), vec3f(0.0, 0.0, 1.0), vec3f(0.0, 1.0, 0.0));
```

- Texture the ground plane using an inverse mapping to obtain (u, v)-coordinates.

$$\boldsymbol{x} = \boldsymbol{x}_0 + u\vec{b}_1 + v\vec{b}_2 \quad \Rightarrow \quad u = (\boldsymbol{x} - \boldsymbol{x}_0) \cdot \vec{b}_1$$
$$\boldsymbol{x} = \boldsymbol{x}_0 + u\vec{b}_1 + v\vec{b}_2 \quad \Rightarrow \quad v = (\boldsymbol{x} - \boldsymbol{x}_0) \cdot \vec{b}_2 \,.$$

- Perform texture sampling using nearest-neighbor and bilinear filtering.
- Use stratified jitter sampling for anti-aliasing.

Figure sources

- Woo, M., Neider, J., and Davis, T. *OpenGL Programming Guide. Second Edition. The Official Guide to Learning OpenGL, Version 1.1*. Addison-Wesley, 1997. `http://www.glprogramming.com/red/index.html`
- Wolfe, R. SIGGRAPH 97 Education Slide Set: Mapping Techniques. *Computer Graphics 31*(4), pp. 66-69, November 1997. `https://education.siggraph.org/archive/slide-sets`
- Watt, A. *3D Computer Graphics*, third edition. Pearson / Addison-Wesley, 2000.
- Shreiner, D., Sellers, G., Kessenich, J., and Licea-Kane, B. *OpenGL Programming Guide. Eighth Edition. The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley, 2013.
- RenderMan 20 Documentation. *Holdout Workflow*. `https://renderman.pixar.com/resources/RenderMan_20/risHoldOut.html`
- Glassner, A. S. *Principles of Digital Image Synthesis*. Morgan Kaufmann. 1995.