Graphics 2008/2009, period 1

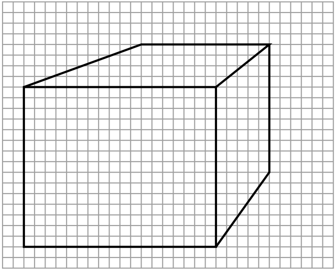
Lecture 10

Texture mapping

Texture mapping

We know how to make stunning realistic images (well, at least in theory. . .) with ray tracing.

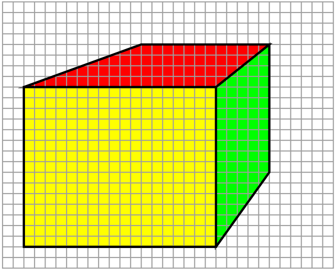
We also know how to make reasonable images blindingly fast (well, at least in theory. . .) with projective methods.



Texture mapping

We even know how to add some realism to our projective images using Gouraud shading or Phong shading

Still, our images are too smooth, and they lack detail. . .



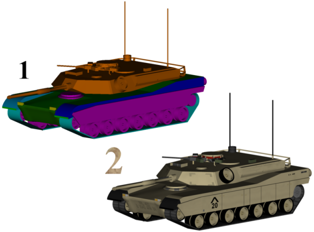
Texture mapping

Adding lots of detail to our models to realistically depict skin, grass, bark, stone, etc., would increase rendering times dramatically, even for hardware-supported projective methods.

Texture mapping

One approach to deal with this issue: texture mapping

Basic idea: instead of calculating color, shade, light, etc. for each pixel we just paste images to our objects in order to create the illusion of realism



Texture mapping

texture map

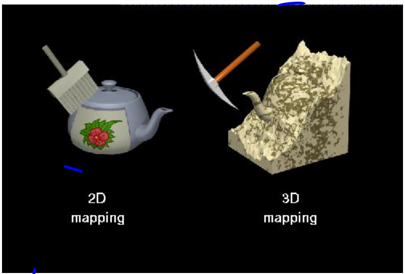
2D 3D

Different approaches exist,
for example 2D vs. 3D:

2D mapping: paste an image onto the object image texture

3D mapping: create a 3D texture and "carve" the object

solid or volume texture



3d objects

Texture mapping

Let's start with 3D mapping, which is a procedural approach, i.e. we use a mathematical procedure to create a 3D texture.

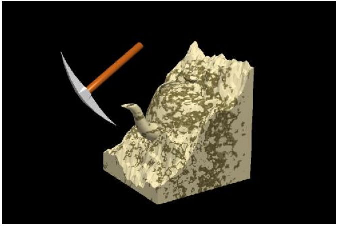
Then we use the coordinates of each point in our 3D model to calculate the appropriate color value using that procedure.

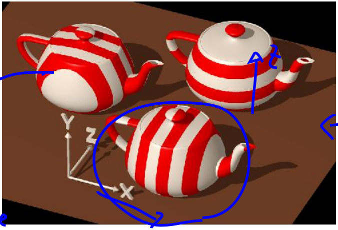
(x,y,z)

Let's look at a simple example: stripes

z = even → red

z = odd → white



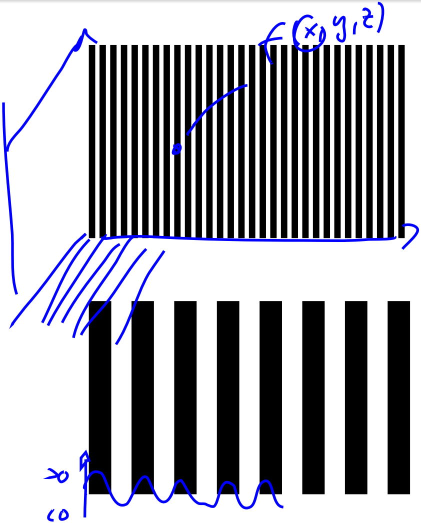


3D stripe textures

Stripe textures are simple.
One way is the following:

(x, y, z)

stripe(point p)
{
  if ($\sin x_p > 0$)
    return color0; (black)
  else
    return color1; (white)
  }
}

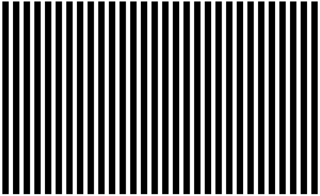


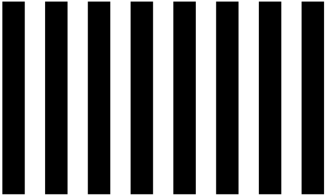
3D stripe textures

Stripes with controllable width:

stripe(point p, real width)
{
  if ($\sin(\pi\, x_p/\mathrm{width}) > 0$)
    return color0;
  else
    return color1;
  }
}

Q: why do we divide by width? Shouldn't that be a multiplication?



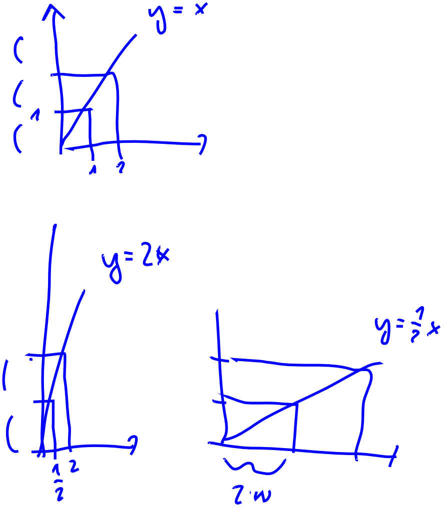


3D stripe textures

Stripes with controllable width:

stripe(point p, real width)
{
  if ($\sin(\pi\, x_p/\text{width}) > 0$)
    return color0;
  else
    return color1;
  }
}

Q: why do we divide by width? Shouldn't that be a multiplication? $w' = 2w$



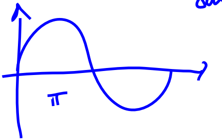
3D stripe textures

Stripes with controllable width:

stripe(point p, real width)
{
  if ($\sin(\pi\, x_p/\mathrm{width}) > 0$)
    return color0;
  else
    return color1;
  }
}

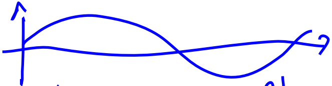
Q: why do we divide by width? Shouldn't that be a multiplication?

wavelength $\ell = 2\pi$ sin

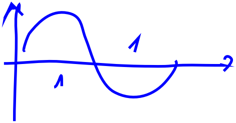


$f = \frac{1}{\ell} = \frac{1}{2\pi}$

$\Rightarrow \ell = \frac{1}{f}$



$f' = 2 \cdot f \Rightarrow \ell' = \frac{1}{2 \cdot f}$



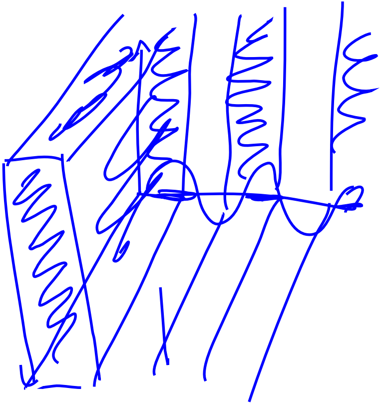
$f' = \frac{1}{2} = \frac{\pi}{2\pi} = \pi \cdot f$

3D stripe textures

Q: why is this called solid texturing?

Q: What if we'd want to vary smoothly between two colors, instead of having two distinct colors?

Q: How do we make tiles instead of stripes?

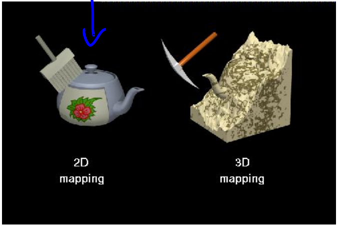


3D stripe textures

Q: why is this called solid texturing?

Q: What if we'd want to vary smoothly between two colors, instead of having two distinct colors?

Q: How do we make tiles instead of stripes?

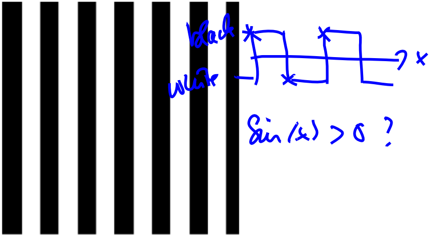


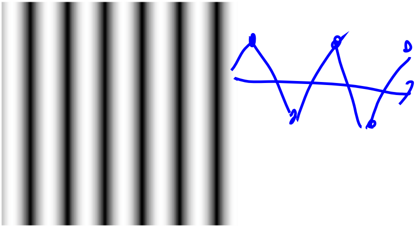
3D stripe textures

Q: why is this called solid texturing?

Q: What if we'd want to vary smoothly between two colors, instead of having two distinct colors?

Q: How do we make tiles instead of stripes?





3D stripe textures

Q: why is this called solid texturing?

Q: What if we'd want to vary smoothly between two colors, instead of having two distinct colors?

Q: How do we make tiles instead of stripes?

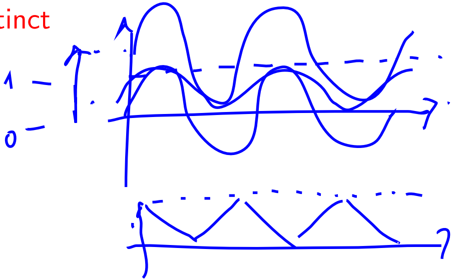
stripe(point p, real width)

{

  t $= (1 + \sin(\pi\, x_p/\text{width}))/\ 2$

  return (1 - t) c_0 + t c_1

}

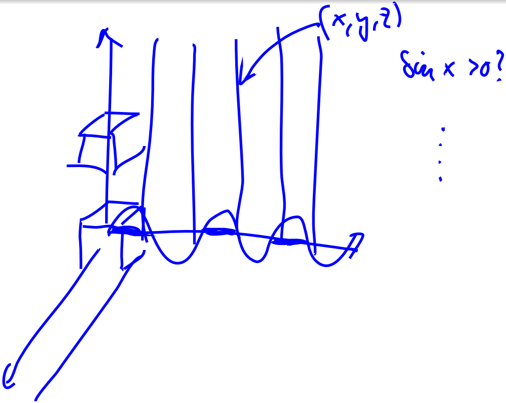


3D stripe textures

Q: why is this called solid texturing?

Q: What if we'd want to vary smoothly between two colors, instead of having two distinct colors?

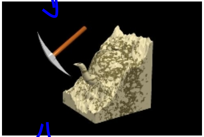
Q: How do we make tiles instead of stripes?



Texture arrays

Instead of computing color values for a point $p \in \mathbb{R}^3$, we can also do an array lookup in a 3D array (using all three coordinates of p for indexing), or in a 2D array (using only two coordinates of p).

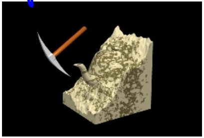
2D vs. 3D mapping

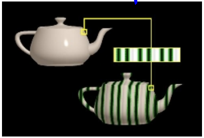


3D carving vs. 3D array lookup

procedure

table lookup





Texture arrays

Let's look at 2D arrays first.

We'll call two dimensions to be mapped u and v, and assume that we have a $w \times h$ image that we use as the texture. Every (u, v) needs to be mapped to a color in the image.

A standard way is to remove the integer portion of u and v, so that (u, v) lies in the unit square.

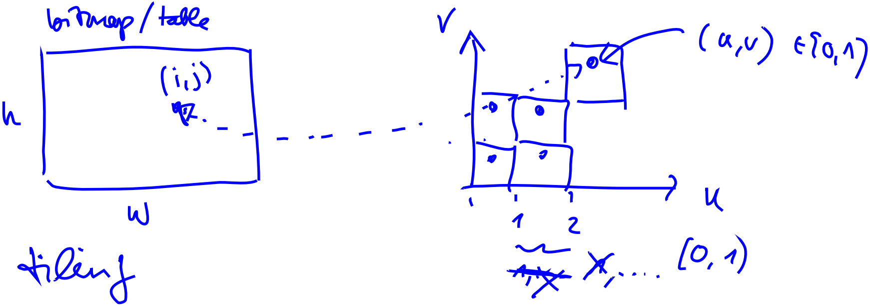
The pixel (i, j) in the $w \times h$ image for (u, v) is found by

$i = \lfloor uw \rfloor$

$j = \lfloor vh \rfloor$

$\lfloor \cdot \rfloor$ = floor function

Texture arrays



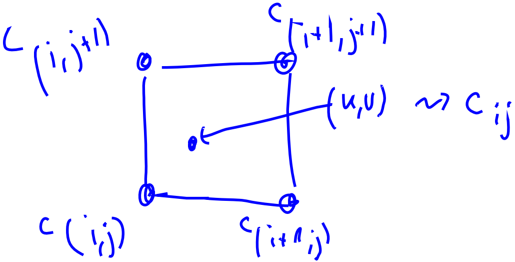
Texture arrays

For smoother effects we may use bilinear interpolation: $c(u,v) =$

$$(1-u')(1-v')c_{ij} + u'(1-v')c_{(i+1)j} + (1-u')v'c_{i(j+1)} + u'v'c_{(i+1)(j+1)}$$

where $u' = uw - \lfloor uw \rfloor$, $v' = vh - \lfloor vh \rfloor$

$0 \le u', v' \le 1$



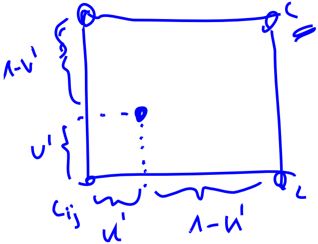
Texture arrays

For smoother effects we may use bilinear interpolation: $c(u, v) =$

$$(1-u')(1-v')c_{ij} + u'(1-v')c_{(i+1)j} + (1-u')v'c_{i(j+1)} + u'v'c_{(i+1)(j+1)}$$

where $u' = uw - \lfloor uw \rfloor$, $v' = vh - \lfloor vh \rfloor$

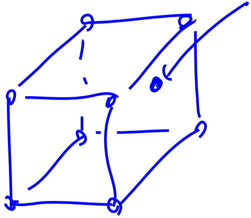
$(1-u')(1-v') + u'(1-v') + \ldots = 1$



Texture arrays

Using 2D arrays with bilinear interpolation is easily extended to using 3D arrays with trilinear interpolation:

$$
\begin{aligned}
c(u,v,w) &= (1-u')(1-v')(1-w')c_{ijk} \\
&\quad +u'(1-v')(1-w')c_{(i+1)jk} \\
&\quad +\ldots
\end{aligned}
$$



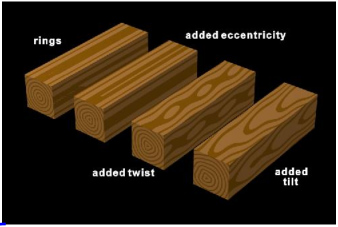
Using random noise

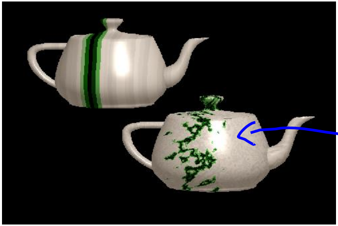
So far: rather simple textures (e.g. stripes).

We can create much more complex (and realistic) textures, e.g. resembling wooden structures.

Or we can create some randomness by adding noice, e.g. to create the impression of a marble like structure.

Perlin noise





Perlin noise

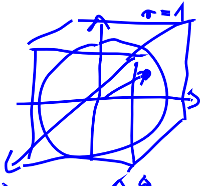
Goal: create a texture that has a random appearance, but not too random, e.g., for marble patterns or mottled textures as on birds' eggs.

Perlin noise is based on the following ideas:

- Use a 1D array of random unit vectors
  and hashing to create a virtual 3D array of random vectors;
- Compute the inner product of (u, v, w)-vectors with the random vectors
- Use Hermite interpolation to get rid of visible artifacts

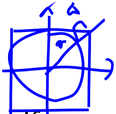
Random unit vectors

Random unit vectors are obtained as follows:



$-1 \leq v_i \leq 1$

$$\begin{aligned} v_x &= 2\xi - 1 \\ v_y &= 2\xi' - 1 \\ v_z &= 2\xi'' - 1 \end{aligned}$$



where ξ, ξ', and ξ'' are random numbers in $[0, 1]$. Then, if $(v_x^2 + v_y^2 + v_z^2) < 1$, normalize the vector and keep it; otherwise, reject it. (Why?)

Perlin reports that an array with 256 such random unit vectors works well with his technique.

Hashing

1-dim. array of pseudo random unitvectors

Perlin noise uses the following hashing function:

$$\Gamma_{ijk} = G(\underbrace{\phi(i + \phi(j + \phi(k)))})$$

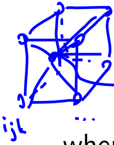
where G is our array of n random vectors, and $\phi(i) = P[i \mod n]$ where P is an array of length n containing a permutation of the integers 0 through $n - 1$.

3-dim. array ....

Hermite interpolation

With our random vectors and hashing function in place, the noise value $n(x, y, z)$ for a point (x, y, z) is computed as:

(i,j,k), (i+1, j,k), (i, j+1, k), ...



$$n(x, y, z) = \boxed{\sum_{i=\lfloor x\rfloor}^{\lfloor x\rfloor+1} \sum_{j=\lfloor y\rfloor}^{\lfloor y\rfloor+1} \sum_{k=\lfloor z\rfloor}^{\lfloor z\rfloor+1}} \underline{\Omega_{ijk}}(x - i, y - i, z - i)$$

where

$$\Omega_{ijk}(u, v, w) = \overbrace{\omega(u)\omega(v)\omega(w)}\overbrace{(\Gamma_{ijk} \cdot (u, v, w))}^{\text{rand. unit vect.}}$$

≙ noise

and

$$\omega(t) = \begin{cases} 2|t|^3 - 3|t|^2 + 1 & \text{if } |t| < 1 \\ 0 & \text{otherwise} \end{cases}$$

Hermite interpol.

Hermite interpolation

With our random vectors and hashing function in place, the noise value $n(x, y, z)$ for a point (x, y, z) is computed as:

$$n(x,y,z) = \sum_{i=\lfloor x \rfloor}^{\lfloor x \rfloor + 1} \sum_{j=\lfloor y \rfloor}^{\lfloor y \rfloor + 1} \sum_{k=\lfloor z \rfloor}^{\lfloor z \rfloor + 1} \Omega_{ijk}(x-i, y-i, z-i)$$

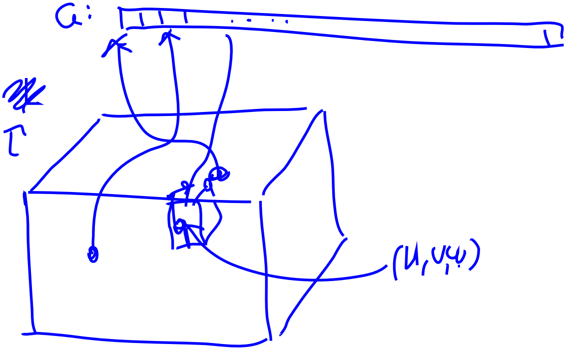
where

$$\Omega_{ijk}(u,v,w) = \omega(u)\omega(v)\omega(w)(\Gamma_{ijk} \cdot (u,v,w))$$

and

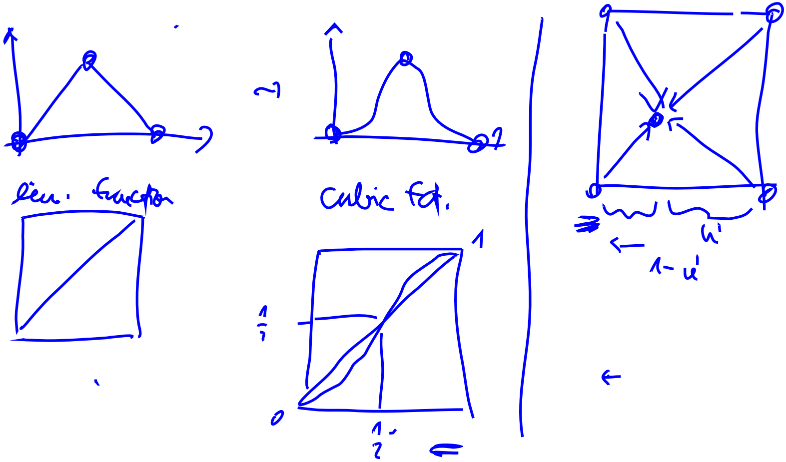
$$\omega(t) = \begin{cases} 2|t|^3 - 3|t|^2 + 1 & \text{if } |t| < 1 \\ 0 & \text{otherwise} \end{cases}$$

Perlin noise (summary)

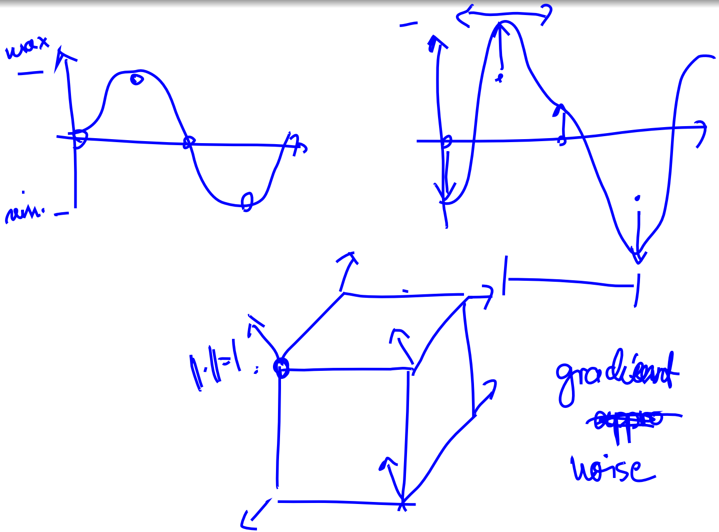


eg. 256 rand. unit vec.

Perlin noise (why do we do this again?)



Perlin noise (why do we do this again?)

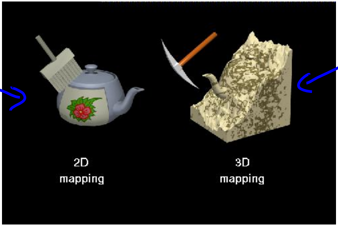


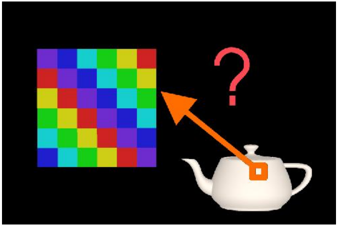
2D texture mapping

Now let's look at 2D mapping, which maps an image onto an object (cf. wrapping up a gift)

Instead of a procedural, we use a lookup-table approach here, i.e. for each point in our 3D model, we look up the appropriate color value in the image.

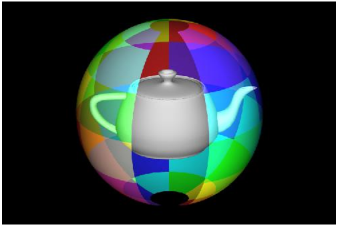
How do we do this? Again, let's look at some simple examples.





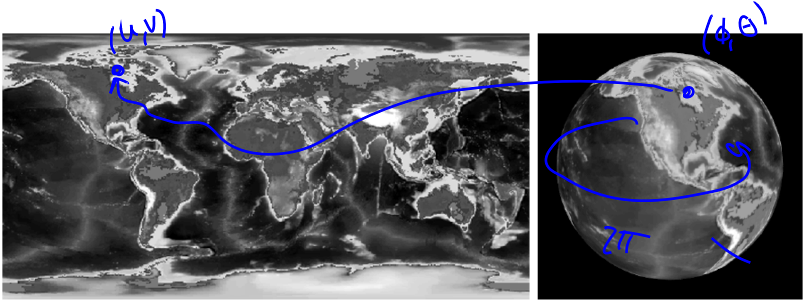
Spherical texture mapping

How do we map a rectangular image onto a sphere?



Spherical texture mapping

Example: use world map and sphere to create a globe



Spherical texture mapping

We have seen the parametric equation of a sphere with radius r and center c:

$$
\begin{aligned}
x &= x_c + r \cos\phi \sin\theta \\
y &= y_c + r \sin\phi \sin\theta \\
z &= x_c + r \cos\theta
\end{aligned}
$$

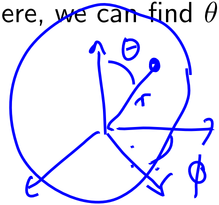
center radius

Given a point (x, y, z) on the surface of the sphere, we can find θ and ϕ by

longitude

latitude

$$
\begin{aligned}
\theta &= \arccos \tfrac{z - z_c}{r} \\
\phi &= \arctan \tfrac{y - y_c}{x - x_c}
\end{aligned}
$$



Spherical texture mapping

For each point (x, y, z) we have

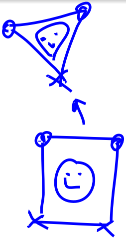
$$\begin{aligned}\theta &= \arccos \tfrac{z-z_c}{r} \\ \phi &= \arctan \tfrac{y-y_c}{x-x_c}\end{aligned}$$

Since both u and v must range from $[0, 1]$, and $(\theta, \phi) \in [0, \pi] \times [-\pi, \pi]$, we must convert:

$$\begin{aligned}u &= \frac{\phi \mod 2\pi}{2\pi} \\ v &= \tfrac{\pi-\theta}{\pi}\end{aligned}$$

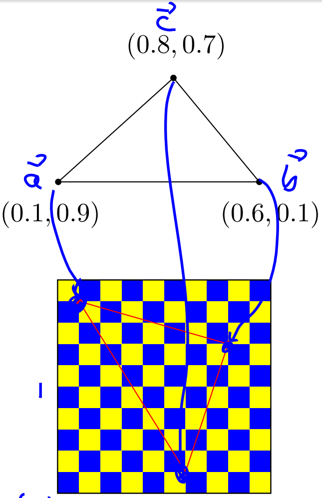
Texturing triangles

$p(\beta,\gamma) = \vec{a} + \beta(\vec{b}-\vec{a}) + \gamma(\vec{c}-\vec{a})$



Mapping an image onto a triangle is done by specifying (u, v) coordinates for the vertices.

For points in the interior of the triangle, we find the texture coordinates by expressing it in barycentric coordinates and linearly interpolating the vertex coordinates.

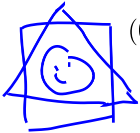


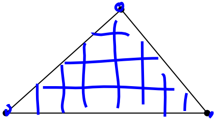
$u(\beta,\gamma) = \vec{u}_a + \beta(\vec{u}_b - \vec{u}_a) + \gamma(\vec{u}_c - \vec{u}_a)$, $v(\ldots)$, $w(\ldots)$

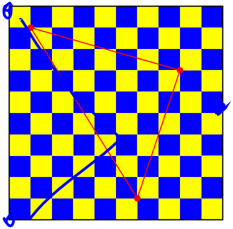
Texturing triangles

Again, we can use bilinear filtering to avoid artifacts.

Note that the area and shape of the triangle don't have to match that of the mapped triangle. Also, (u, v) coordinates for the vertices may lie outside the range $[0, 1] \times [0, 1]$.







Triangle meshes

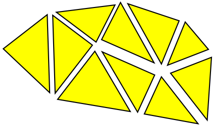
Complicated objects are commonly modeled as a triangle mesh with shared vertices.

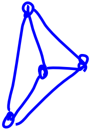
Sharing vertices not only saves space, but also has the advantage that texture varies smoothly over the mesh.

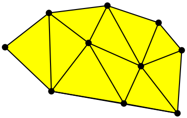
cf. normal vect. for shading

triangle "soup"

3·3 = 9 vertices







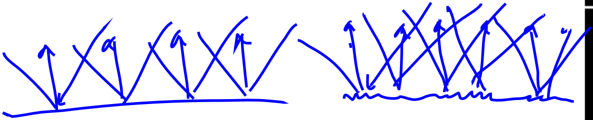
4 vertices (+ info about the mesh)

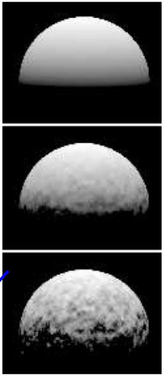
triangle mesh

Bump mapping

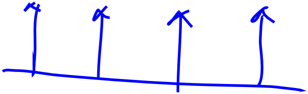
One of the reasons why we apply texture mapping:

Real surfaces are hardly flat but often rough and bumpy. These bumps cause (slightly) different reflections of the light.

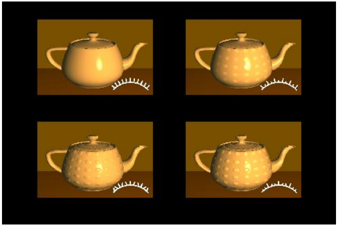




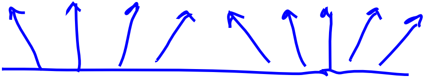
Bump mapping

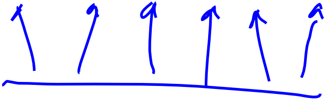


Instead of mapping an image or noise onto an object, we can also apply a bump map, which is a 2D or 3D array of vectors. These vectors are added to the normals at the points for which we do shading calculations.



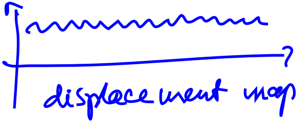
The effect of bump mapping is an apparent change of the geometry of the object.



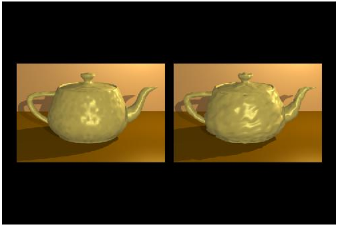


Displacement mapping

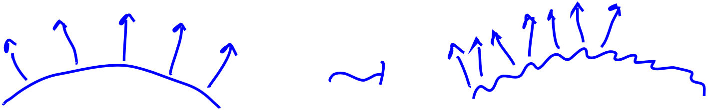
Major problems with bump mapping: silhouettes and shadows



To overcome this shortcoming, we can use a displacement map. This is also a 2D or 3D array of vectors, but here the points to be shaded are actually displaced.

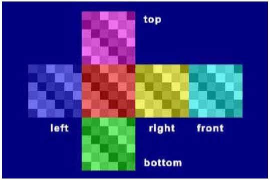


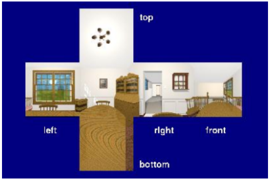
Normally, the objects are refined using the displacement map, giving an increase in storage requirements.

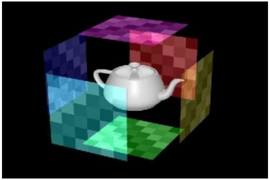


Environment mapping

Let's look at image textures again:







If we can map an image of the environment to an object ...

Environment mapping

... why not use this to make objects appear to reflect their surroundings specularly?

Idea: place a cube around the object, and project the environment of the object onto the planes of the cube in a preprocessing stage; this is our texture map.

During rendering, we compute a reflection vector, and use that to look-up texture values from the cubic texture map.

Environment mapping