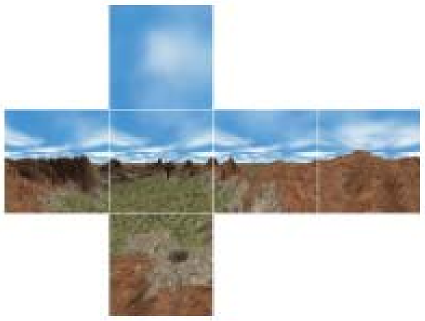
Textures and other Mappings

An essential optical property of real surfaces is their irregularity. This can either be caused by the environment, the variable coloring of the surface or the shading due to surface roughness. For these three causes we can simulate the effects with the three methods *environment mapping*, *texture mapping* and *bump mapping*.

Environment Mapping

Environment denotes the parts of the scene that surround an object. Depending on the object's surface properties, the environment has a varying effect on the object's appearance. For perfectly reflecting surfaces we can generate the environment's reflection exactly with ray tracing. For surfaces which are not perfectly diffuse we can create specular highlights as an approximation of the actual light source reflections with the Phong-model. However, even a surface which is more or less reflective reflects its whole environment more or less sharply, though with reduced precision. We use environment mapping to render objects in a complex environment efficiently without having to model the whole environment completely and without placing great demands on exactness. To achieve this, we produce an image of the environment in a preprocessing step, starting from a central point (e.g. the object's center or the center of the scene). It does not matter whether it is a calculated or photographed image. In any case the general assumption is that the environment (e.g. a sphere or a cube) is large in comparison to the objects to be rendered. During rendering we consider every surface point to lie in the environment's center. Due to this approximation, we can quickly determine which point of the environment is hit (e.g. with polar coordinates), solely based on the reflection ray's direction, without the need to resort to expensive ray tracing.



Texture Mapping

Many surfaces are not monochrome but instead have patterns, for example wood grain, pictures on the wall, writing on paper, dirt, clothes, marble. Also, when modeling only coarsely, some details can be interpreted as patterns, like windows on the wall of a house, clouds in the sky, faces, buttons and zips, paving stones. Such patterns are called textures. The placement of textures onto objects is called texture mapping.

$$\begin{tikzcd}
\boxed{\begin{array}{c}\text{Texture Space:}\\ \text{(s,t) Array}\\ \text{Coordinates}\end{array}} \arrow[r, "\begin{array}{c}\text{Texture-Surface}\\ \text{Transformation}\end{array}"'] & \boxed{\begin{array}{c}\text{Object Space:}\\ \text{(u,v) Surface}\\ \text{Parameters}\end{array}} \arrow[r, "\begin{array}{c}\text{Viewing \& Projection}\\ \text{Transformation}\end{array}"'] & \boxed{\begin{array}{c}\text{Image Space:}\\ \text{(x,y) Pixel}\\ \text{Coordinates}\end{array}}
\end{tikzcd}$$

Once textures have been generated their mapping is performed in two steps. First we must define which texture should be placed on which surface of the objects (orientation, scaling, etc.). Actually this is a modeling task, during which surfaces are assigned a specific appearance. In the second step the texture must be rendered onto the object's projection correctly, i.e. transformed into the image.

Texture Generation

Basically it does not matter where a texture comes from, as long as it is defined and accessible at each position. Usually a texture is generated as a pixel array in a preprocessing step and then it is solely read. We can use photos as well as scans and

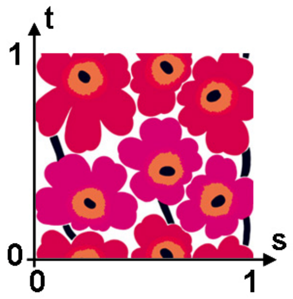
even textures generated by a program or random values. For textures like wood grain, gras, sand, marble, cobblestones or tissue, which are used frequently, a databank can be set up. Whenever textures are generated by a mathematical function, we talk about “procedural texturing“.

Texture-Object-Transformation

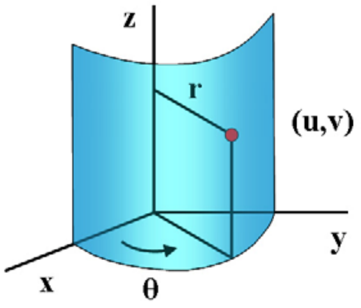
Usually a texture is given in a 2D coordinate system, which we address with (s,t)-coordinates. Furthermore we assume that the surface to which we want to attach the texture also has a parametric description, which we refer to with (u,v)-coordinates. A bilinear function for applying a texture then looks like this:

$$u = u(s,t) = a_u s + b_u t + c_u \,, \qquad v = v(s,t) = a_v s + b_v t + c_v$$

i.e. for each point on an object’s surface the corresponding color can be determined. This function is called texture-object-transformation and is named $\mathbf{M_T}$.



Example:
A texture T(s,t), $0 \le s,t \le 1$, shall be attached to a quarter cylinder with height h, whose mantle is parameterized with v along the z-direction and with u (= θ) along the curvature. To determine for each pixel T(s,t) [also called a *texel*] at which position on the cylinder it will appear, we must define the mapping M_T, which could look like this:
$u = s \cdot \pi/2$, $v = t \cdot h$ (in this way the entire texture fits exactly onto the quarter cylinder).



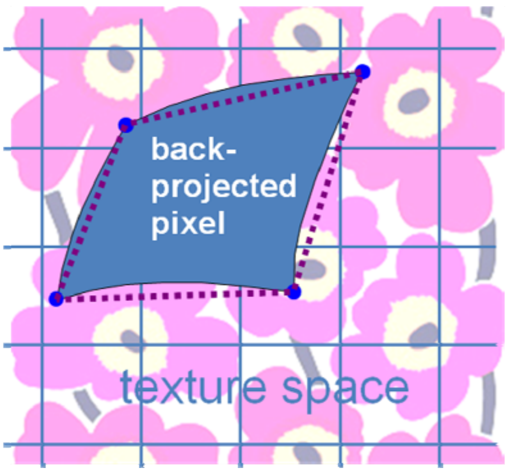
Viewing and Projection Transformation

Basically the mapping of a 3D model onto the image plane is a simple projection $\mathbf{M_{VP}}$. In order to fill each pixel of a surface’s screen projection during the raster-scan exactly once (i.e. no overdraw, no gaps), the algorithm works in reverse direction. For each pixel (x,y) it is determined which surface point is drawn there (i.e. the surface’s (u,v)-coordinates) and then we determine which texel is valid for that pixel. To achieve this, we need the inverse operators $\mathbf{M_{VP}}^{-1}$ and $\mathbf{M_T}^{-1}$,

Example (continued):
For an arbitrary projection we only need to know a point’s (x,y,z)-coordinates. In the case of the cylinder above we have: $x = r \cdot \cos u$, $y = r \cdot \sin u$, $z = v$.

Now we approach the problem starting at a pixel:
- For a pixel P we first determine the position (x,y,z) on the cylinder, which is displayed there
  (e.g. through ray casting).
- For this point we must find the parameters of the surface: $u = \cos^{-1}(x/r)$, $v = z$.
  Now we have the inverse transformation M_{VP}^{-1}.
- Finally we must find the texture to be displayed there for the parameter-pair (u,v) by inverting M_T:
  $s = 2u/\pi$, $t = v/h$ (this is M_T^{-1})

Anti-Aliasing for Textures



Textures are especially prone to aliasing effects, particularly when patterns are significantly scaled. To calculate the value of a pixel correctly we would have to calculate the average texture value of the area that is covered by the pixel to be filled, back-projected into the texture. As an approximation the polygon constructed by connecting the back-projected corner points with straight lines is precise enough. Since this would still work quite slowly, one out of two possible optimizations is normally used: In *mip-mapping* different resolutions of a texture are pre-calculated, which are used to interpolate needed values linearly. In the *summed-area-table*-method the average values of rectangular blocks can easily be determined by calculating differences from pre-calculated texture sums.

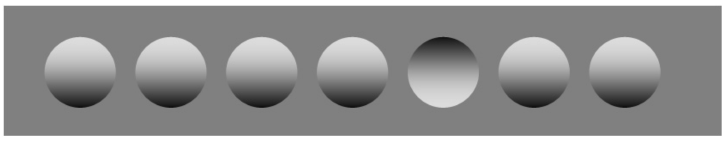
Solid Texturing

© Chen, Lai (Stanford)

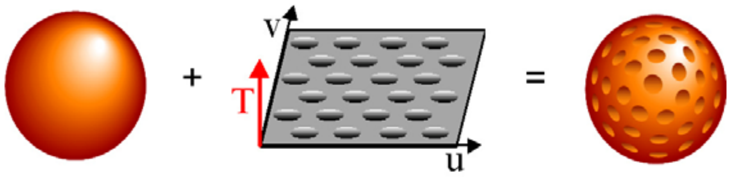
A texture need not be given in 2D, alternatively it can be defined as a 3D volume. This means, in a three-dimensional parameter space a color is defined for each point in space, which can then be queried by a surface that is located at that particular point. Employing this method the texture may either be given by a mathematical function or by a volumetric data set, from both the values for any point in space can be queried. The big advantage of solid texturing in contrast to surface texturing is that patterns continue coherently across edges, so that no seams are present at any two connected polygons. Furthermore the mapping of a 3D texture onto an object is much easier to handle.

Bump Mapping and Displacement Mapping

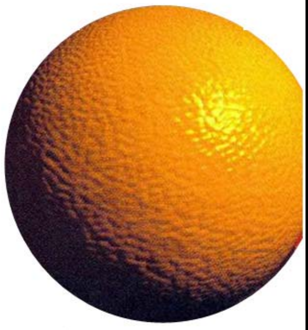
Many surfaces have a fine geometrical structure: bark, coins, plastering, leather, fabric, vegetables, planets, gravel roads, lasagna, tiles, chocolate bars, earthworm and so on. To entirely model such effects geometrically is tedious and generates a huge amount of data. Bump mapping helps to reduce this effort significantly.



When we take a look at the gray bar to the right, we get the impression that it has six protrusions and one indentation. When touching the bar, however, it is completely planar! Why do we see the bumps? Because shading alone can cause the impression of three-dimensionality. We can make use of this trick to create the impression of bumps on surfaces without much additional effort. The basic idea is to leave the surface geometry unchanged, but instead only change the normal vector according to the bumps. In this way the shading corresponds to the presence of bumps even though geometrically the surface is not changed.



Bump Mapping Algorithm



Let the bump texture be given by an array of height values b(u,v), that means that a the position P(u,v) on an object's surface given by the parameter-pair (u,v) shall appear as if it were displaced by the amount b(u,v) along the normal vector. The normal vector n is the result of the cross product of two tangent vectors, normalized to unit length:

$$N = P_u \times P_v, \qquad n = N / |N|$$

The displaced point P'(u,v) is then given by: $P'(u,v) = P(u,v) + b(u,v) \cdot n$

However, instead of N we need N', which denotes the normal vector at the displaced position:

$$N' = P_u' \times P_v'$$

From this follows:

$P_u' = \partial(P + bn)/\partial u = P_u + b_u n + b n_u$ and because b is very small: $P_u' \approx P_u + b_u n$

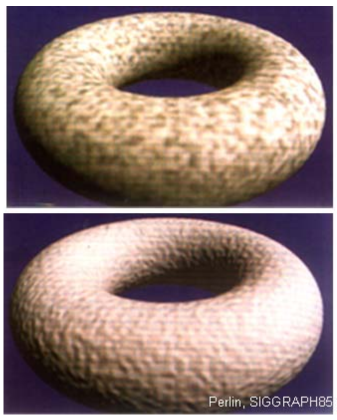
analogically it is clear that $P_v' \approx P_v + b_v n$, so that for N' we get:

$$N' = P_u' \times P_v' = P_u \times P_v + b_v(P_u \times n) + b_u(n \times P_v) + b_u b_v(n \times n)$$

and from $n \times n = 0$ we can conclude:

$$\boxed{\mathbf{N' = N + b_v(P_u \times n) + b_u(n \times P_v)}}$$

So we do not need b(u,v), but actually the partial derivatives with respect to u and v. Since in practice the parameterization of the surface and the bump texture are often the same, the derivatives can easily be pre-calculated and stored instead of b(u,v).



In the figures to right showing a donut, note the difference between a texture map (upper figure) and a bump map (lower figure). The impression of a three-dimensional surface emerges only when the shading is direction-dependent.

Of course, bump mapping is just a big deception, because it only adjusts surface shading without altering geometry. Correspondingly visible errors remain, all the more noticeable, the higher the bumps are:

1. at flat angles the structure is severely distorted
2. the silhouette stays as smooth as the original geometry was (so: smooth = wrong)
3. thus the object also casts shadows with smooth (= wrong) borders
4. there is no self-shadowing among bumps
5. surface normals on the light-averted side of an object may point towards the light source and thus receive light wrongly!

Displacement Mapping

Each of these errors can be addressed with more or less costly techniques, although the only correct solution would be to actually alter the geometry of the surface according to the height of the bumps. This method is called *displacement mapping*, which really displaces the surface points, thus naturally creating a correct silhouette. However, this is far more complex to implement and therefore not used very often. Nevertheless, there is a trend to support the technique in graphics hardware.

Combinations of several Mappings

Combining several mappings is a powerful method. In the case of textures this is also called *multi-texturing*. Examples of combined textures are: ground patterns, lighting, soiling, bumps, and also for example photos with annotations. Furthermore we can also combine environment- and displacement mapping which leads to results similar to the one shown to the right.