Graphics & Visualization

Chapter 14

Texturing
Texturing – Why?

- On every surface, the human visual system can detect:
  - Small imperfections,
  - Geometric details,
  - Patterns,
  - Variations in material consistency

- The above help to:
  - Determine the physical qualities of the various media

- Simple patterns: Possible to represent the above as changes in the surface structure & vertex properties

- Irregular and complex patterns:
  - Need a different approach to modify the local material behavior across a surface
Texturing – Why?

• Appearance variation of a surface:
  (a-b) A simple pattern can be approximated via surface restructuring
  (c) Complex patterns cannot be efficiently represented this way

• With texturing, an elaborate design can be imprinted on a surface without modifying the actual geometry
Introduction

- **Texturing**: Mechanism of spatially varying one or more of the material attributes of a surface in a predefined manner without affecting the underlying topology of the geometry.

- Some of these attributes can be:
  - Color of the surface
  - Transparency of the surface
  - Local normal at a given point
  - Reflectivity at a given point
Introduction (2)

- Association between a surface point $p$ & a material value in the texture space (where the desired pattern is defined), is done via a \textit{texture-mapping function}

- Pattern itself can be:
  - A 1-, 2-, or 3-D digital image (\textit{texture map})
  - A procedurally generated material
Depending on the attribute of the material that is affected by texture mapping, the result can be a:
- Scalar value (as in case of surface's specular coefficient)
- Alpha value (transparency)
- Vector (signifying an RGB color value)
- New local normal vector
- ...

In order to modify one or more of material properties:
- Multiple textures can be applied to a single surface

Different texture-mapping functions may be associated with a single attribute & combined under a texture hierarchy (texture tree)
Parametric Texture Mapping

- Material attribute is defined as a:
  - pre-computed or
  - hand-drawn or
  - digitized
digital image → texture-mapping is called *texture* or *image mapping*

- Texture images (*textures*), can be 1-, 2-, or 3-D & are buffers of pre-calculated data that the mapping function addresses to acquire material values
Parametric Texture Mapping (2)

- In *parametric texture mapping*, the mapping function is split into 2 parts:
  - Association of 3-D Cartesian coordinates to the parametric domain of the digital image
  - Extraction of material attribute values according to color intensity of texture samples at the above parametric coordinates
Texture Space

- Discrete texture elements are called *texels*
- Digital image is usually mapped to a normalized domain of parametric space $T^D$
- $D$: Dimensionality of the pattern ($D=2$ for conventional 2-D bitmap)
- Continuous normalized parameters are called *texture coordinates*
- Mapping of 3-D coordinates to the texture space produces parameters that are either wrapped or clamped to the $[0,1]$ range
- A texture map will refer to a 2-D pattern with corresponding texture parameters $u, v \in [0,1]$
From Cartesian to Texture Coordinates (1)

- Parametric texture mapping:
  - Cartesian coordinates are mapped to texture coordinates
  - Material attributes are estimated from the discrete texel values & applied to the initial surface locations
From Cartesian to Texture Coordinates (2)

- Texture Tiling: A texture-coordinate pair is not necessarily uniquely associated with a location on the 3-D surface →
- Provides great economy when applying a periodic texture to an object
Texture Value Estimation

- The texture can be sampled in many ways. The most common techniques are:
  - Nearest neighbor texel selection
  - Bilinear interpolation
  - Bilinear interpolation with pre-filtering (see mip-mapping)
Texture Value Estimation (2)

- Nearest neighbor texel selection produces jagged texel edges when the texture is magnified, exposing the inadequate digital representation of the depicted pattern (pixelization)
Bilinear Texture Value Interpolation

- Given an \( N_x \times N_y \) texture map and using bilinear interpolation, \( I(u, v) \) can be estimated as follows:
  - Compute the scalar texel locations \( x \) & \( y \) that correspond to given \((u, v)\) parameters:
    \[
    x = u \cdot N_x \\
    y = v \cdot N_y
    \]  
    \[(14.1)\]
  - Calculate interpolation coefficients for the horizontal and vertical texel spans:
    \[
    u' = x - \lfloor x \rfloor \\
    v' = y - \lfloor y \rfloor
    \]  
    \[(14.2)\]

- Final intensity \( I(u, v) \) is given by the row-column bilinear interpolation of the intensity at the four texel centers:

\[
(\lfloor x \rfloor, \lfloor y \rfloor), (\lfloor x \rfloor, \lceil y \rceil), (\lceil x \rceil, \lfloor y \rfloor), \text{ and } (\lceil x \rceil, \lceil y \rceil)
\]
Bilinear Texture Value Interpolation(2)
Bilinear Texture Value Interpolation (3)

So we have:

\[
I_{\text{bot}} = I(\lceil x \rceil, \lceil y \rceil) \cdot (1 - u') + I(\lfloor x \rfloor, \lfloor y \rfloor) \cdot u',
\]

\[
I_{\text{top}} = I(\lceil x \rceil, \lfloor y \rfloor) \cdot (1 - u') + I(\lfloor x \rfloor, \lfloor y \rfloor) \cdot u',
\]

\[
I(u, v) = I_{\text{bot}} \cdot (1 - v') + I_{\text{top}} \cdot v'
\]

(14.3)
Texture Mapping Polygonal Surfaces

• **Usual practice:**

• Determination of the \((u, v)\)-coordinates (1\textsuperscript{st} stage of texture-mapping procedure), is performed before rendering of the surface
  - Texture coordinates for each vertex are assigned to them before the polygon rasterization

• During scan-conversion of triangles, texture coordinates for each triangle sample are interpolated from the texture parameters stored in the vertex data

• Exception from procedural textures and dynamic texture effects:
  - Texture coordinates are determined from the local surface & rendering attributes which are in turn interpolated from the vertex data
Texture Mapping Polygonal Surfaces (2)

• Directly interpolating the texture coordinates from the vertex texture parameters → does not account for the projective mapping that the vertex coordinates undergo:
  - Texture parameters are assigned to vertices → not transformed when polygons are perspectively projected

![Diagram of projective mapping]

• During scan conversion, perspectively correct vertices are linearly interpolated and so are the texture coordinates:
  - The latter have not been divided by the depth value, as the projective transformation dictates
  → Leads to an inconsistent mapping → results in visible stretching, bitmap tearing, and “texture floating”
Texture Mapping Polygonal Surfaces (3)

**Perspective correction:**
- Interpolate the \((u, v)\)-coordinates after dividing them with the \(z\) vertex value.
- The \(1/z\) quantity is also interpolated from the projected vertex depth values.
- Then, the perspectively correct parameters are obtained by dividing the interpolated \(u/z\) & \(v/z\) parameters by the estimated \(1/z\).

**Same practice can be applied to other quantities of a polygon that are not affected by the perspective transformation:**

**In modern GPUs, fragment programs can access & modify the texture coordinates produced by the rasterization algorithm**
Texture Mapping Polygonal Surfaces (4)

- Bilinear coordinate interpolation is possible:
  - when progressively sampling the polygon surface in a regular manner or
  - when the surface- and texture-coordinate parameterizations are coincident

- Bilinear coordinate interpolation is not convenient if:
  - single texture-coordinate sample is required at an arbitrary location on a triangle
  - We can use the *barycentric coordinate* representation of the triangle instead
Barycentric Coordinates

- Any point $\mathbf{p}$ on a triangle $\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$ plane can be represented as an affine combination of those 3 basis points:

$$\mathbf{p} = \lambda_1 \mathbf{p}_1 + \lambda_2 \mathbf{p}_2 + \lambda_3 \mathbf{p}_3,$$

(14.4)

- Parametric domain is a plane in $\mathbb{R}^3$

- By restricting $\lambda_1$, $\lambda_2$, $\lambda_3$ to $[0,1] \rightarrow$ *barycentric triangle form* of (14.4) becomes a function that maps an equilateral triangle in space to a range that is exactly the interior of the triangle $\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$
Barycentric Coordinates (2)

- The 3 barycentric coordinates are directly associated with the ratios of the triangle areas formed by point \( p \) to the total:

\[
\begin{align*}
\lambda_1 &= \frac{A_1}{A} = \frac{A(pp_2p_3)}{A(p_1p_2p_3)}, \\
\lambda_2 &= \frac{A_2}{A} = \frac{A(p_1pp_3)}{A(p_1p_2p_3)}, \\
\lambda_3 &= 1 - \lambda_1 - \lambda_2 = \frac{A_3}{A} = \frac{A(p_1p_2p)}{A(p_1p_2p_3)},
\end{align*}
\]

(14.6)

where \( A(v_1, v_2, v_3) \) is the area of triangle \( v_1v_2v_3 \)
Barycentric Coordinates (3)

- Calculation of the barycentric triangle coordinates from ratios of triangle areas:

\[ \lambda_1 = \frac{\overrightarrow{pp_2} \times \overrightarrow{pp_3}}{\overrightarrow{p_1p_2} \times \overrightarrow{p_1p_3}}, \quad \lambda_2 = \frac{\overrightarrow{p_1p} \times \overrightarrow{p_1p_3}}{\overrightarrow{p_1p_2} \times \overrightarrow{p_1p_3}}, \quad \lambda_3 = 1 - \lambda_1 - \lambda_2 \quad (14.7) \]

- Since area of a triangle is proportional to the magnitude of the cross product of 2 of its edges \( A(\overrightarrow{v_1v_2v_3}) = \frac{1}{2} |\overrightarrow{v_1v_2} \times \overrightarrow{v_1v_3}| \), barycentric coordinates can be calculated by transforming (14.6) to:

\[ \lambda_1 = \frac{\overrightarrow{pp_2} \times \overrightarrow{pp_3}}{\overrightarrow{p_1p_2} \times \overrightarrow{p_1p_3}}, \quad \lambda_2 = \frac{\overrightarrow{p_1p} \times \overrightarrow{p_1p_3}}{\overrightarrow{p_1p_2} \times \overrightarrow{p_1p_3}}, \quad \lambda_3 = 1 - \lambda_1 - \lambda_2 \]
Barycentric Coordinates (4)

- Texture coordinates \((u,v)\) for point \(p\) can be easily interpolated from the texture coordinates \((u_i,v_i)\), \(i=1\ldots3\) stored in the vertex data of \(p_1p_2p_3\):

\[
\begin{align*}
    u &= \lambda_1 u_1 + \lambda_2 u_2 + \lambda_3 u_3, \\
    v &= \lambda_1 v_1 + \lambda_2 v_2 + \lambda_3 v_3.
\end{align*}
\]

- Parameter-interpolation is more generic than bilinear interpolation
  - BUT more costly to perform \(\rightarrow\) should be used when a progressive scan is not possible
Texture-Coordinate Generation

• \( u, v \) parameters have been deduced from the Cartesian coordinates of the vertices with the help of a texture-coordinate generation function:
  - Provides a simple mapping from the Cartesian domain to the bounded normalized domain in texture space

• Most common functions perform the mapping in 2 steps:
  - Arbitrary Cartesian coordinates \( \rightarrow \) predetermined “auxiliary” surface embedded in space (shape represented parametrically)
  - Auxiliary surface parameters are normalized to represent texture coordinates
Texture-Coordinate Generation (2)

• The parametric surface determines how a planar textured sheet is wrapped around the original object

• **Important issue:** Tiling
  
  - Texture-coordinate generation transformation should map a point in space into the bounded domain of the image map: \([0,1] \times [0,1]\)

• For texture coordinate estimation for polygons, wrapping to \([0,1]\) must be **performed after the interpolation** of the values during rasterization
Texture-Coordinate Generation (3)

• If a texture parameter is wrapped to the [0,1] range before assigned to a vertex → interpolated values between 2 consecutive vertices can be accidentally reversed:
Texture-Coordinate Generation (4)

- The 2 most common local attributes used for texture-coordinate generation:
  - Location in space of the fragment being rendered
  - Local surface normal vector
- Other local attributes can be exploited in order to address the texture space (i.e. incident-light direction)
Planar Mapping

Planar Mapping: Simplest \((u, v)\)-coordinate generation function

- Uses a plane as an intermediate parametric surface
- Cartesian coordinates are parallelly projected onto the plane

\[
\begin{align*}
    u &= x' - \lfloor x' \rfloor & x' &= ax + \text{offset}_x \\
    v &= y' - \lfloor y' \rfloor & y' &= by + \text{offset}_y
\end{align*}
\] (14.9)

- Although an arbitrary plane can be used, selecting one that is axis-aligned greatly simplifies the calculations
Planar Mapping (2)

- Planar mapping using the $xy$-plane. Note that all points with the same $x,y$-coordinates are mapped onto the same texture-map location.
Planar Mapping (3)

- In Equation (14.9), $a$, $b$ are the tiling factors and $(\text{offset}_x, \text{offset}_y)$ is the offset from the lower-left corner of the image in texture-coordinate space.

- Tiling factors determine how many repetitions of the texture image should fit in one world-coordinate-system unit.

- Planar mapping is useful for texturing flat surface regions that can be represented in a functional manner, as in $z = f(x, y)$.

- The more parallel a surface region is to the projection plane, the less distorted the projected texture is.
Cylindrical Mapping

• **Cylindrical Mapping**: Texture coordinates are derived from the cylindrical coordinates of a point \( p \) in space:

\[
\begin{align*}
\theta &= \tan^{-1}(x/z), \\
h &= y, \\
r &= \sqrt{x^2 + z^2}
\end{align*}
\]  

(14.10)

where \( \theta \) is the right-handed angle from the \( z \)- to \( x \)-axis, with \(-\pi < \theta \leq \pi\), \( h \) is the vertical offset from the \( xz \)-plane, \& \( r \) is the distance of \( p \) from the \( y \)-axis.

• \((u,v)\)-coordinates can be associated with any 2 of the cylindrical coordinates of Equation (14.10):
  
  - Usually \( u \) is derived from \( \theta \) \& \( v \) is calculated from the height \( h \)
Cylindrical Mapping (2)

- The result of cylindrical mapping is similar to wrapping a photograph around an infinitely long tube:

- All points with the same bearing and height are mapped to the same point in texture space for all $r \in [0, \infty)$
Cylindrical Mapping (3)

- Cylindrical coordinates of Equation (14.10) can be easily transformed into texture coordinates using:

\[
u = \frac{1}{2} + \frac{\theta}{2\pi} = \frac{1}{2} + \frac{\tan^{-1}(x/z)}{2\pi},
\]

\[
v = h = y
\]

- Above formulation of the \((u,v)\)-coordinate pair can be augmented to include the tiling factors \(a\) & \(b\) around & along the \(y\)-axis, respectively:

\[
u = \frac{a(\theta + \pi)}{2\pi} - \left\lfloor \frac{a(\theta + \pi)}{2\pi} \right\rfloor,
\]

\[
v = by - \left\lfloor by \right\rfloor
\]
Spherical Mapping

- **Spherical Mapping**: Texture-coordinate generation function depends on the spherical coordinates \((\theta, \varphi, r)\) of a point in space.
- \(\theta\) is the longitude of the point, \(\varphi\) is the latitude and \(r\) is the distance from the coordinate system origin.
- Spherical coordinates of a point \(\mathbf{p} = (x, y, z)\) are given by:

\[
\begin{align*}
\theta &= \tan^{-1}\left(\frac{x}{z}\right) - \pi < \theta \leq \pi \\
\varphi &= \tan^{-1}\left(\frac{y}{\sqrt{x^2 + z^2}}\right) - \frac{\pi}{2} < \varphi \leq \frac{\pi}{2} \quad (14.13) \\
r &= \sqrt{x^2 + y^2 + z^2}
\end{align*}
\]
Spherical Mapping (2)

- Spherical mapping:
  
  (a) The texture-coordinate parameterization and image wrapping
  
  (b) An evening-sky texture mapped to a dome using the spherical texture-coordinate generation function
Spherical Mapping (3)

- Spherical texture-coordinate generation usually associates the $u$- & $v$-coordinates with the 2 angular components of the above representation:

$$u = \frac{\theta + \pi}{2\pi}, \quad v = \frac{\varphi + \pi / 2}{\pi}$$  \hspace{1cm} (14.14)

- Using a pair of tiling factors $(a,b)$ for the $u$- & $v$-coordinates, respectively, the spherical mapping is given by:

$$u = \frac{a(\theta + \pi)}{2\pi} - \left\lfloor \frac{a(\theta + \pi)}{2\pi} \right\rfloor,$$

$$v = \frac{b(\varphi + \pi / 2)}{\pi} - \left\lfloor \frac{b(\varphi + \pi / 2)}{\pi} \right\rfloor$$  \hspace{1cm} (14.15)
Spherical Mapping (4)

- Spherical mapping operation wraps an image around an object like a world atlas maps to the globe.
- Spatial resolution of a texel varies according to the latitude of the point → heavy distortion at the poles:
  - a whole line of the texture is typically mapped onto a single point in space.
Spherical Mapping (5)

- For calculation of the texture parameters, normal vector coordinates can be used instead of the point location:
  - Replace \((x,y,z)\) point coordinates of Equation (14.13) with the normal vector components \((n_x , n_y , n_z )\)

Uses:
- Pre-calculate, store and index incoming diffuse illumination from distant light sources as texture map
- Replace the Phong model with any (e.g. toon shading) precalculated spherical function and apply as texture
  - Light sources are considered to be infinitely far from the objects
Cube (Box) Mapping

• Combines local surface-direction information with the Cartesian coordinates of the point → derive the texture coordinates

• One of the 3 primary axes is selected according to the principal normal direction component

• A point \( p \) is projected onto plane \( xy, yz, \) or \( xz, \) depending on whether the absolute value of the \( z-, x-, \) or \( y-\)coordinate, of the normal vector is the largest one

• Planar mapping for each one of the 3 cases → properly substituting the coordinate pairs in Equation (14.9)
Cube Mapping (2)

- Cube mapping is ideal for multifaceted geometry & for shapes with right angles
- **Useful property:** Texture map is never projected on a surface from an angle of more than 45 degrees from the surface normal
Cube Mapping (3)

- We get no significant distortion from texel stretching but
- Transition from one projection plane to another is prone to causing discontinuities
Cube Mapping

Why use this mapping paradigm?

- In spherical mapping significant distortion at the poles is caused due to the inherent mapping singularity
- Cube mapping avoids this pitfall by always selecting the side of a cube that is most perpendicular to the vector associated with the current point
- 6 maps are prepared and indexed instead of 1
Cube Mapping Example

- Cube mapping using vector coordinates to apply pre-calculated diffuse illumination to an object:

  (a) Set-up   (b) 6 cube maps
  (c) Texture-shaded object in its final environment
Cube Mapping Calculations

- Cube-mapping texture coordinates are calculated as the normalized coordinates in the range \([0,1]\) of a vector \(\vec{v} = (v_x, v_y, v_z)\).

- Appropriate cube texture is selected according to the largest-in-magnitude signed component of the vector provided.

- Cube mapping on vector coordinates is implemented both in OpenGL and Direct3D in a consistent manner.
Cube Mapping Calculations (2)

\[ u = \frac{1}{2} \left( \frac{u_c}{|m_a|} + 1 \right), \]

\[ v = \frac{1}{2} \left( \frac{v_c}{|m_a|} + 1 \right) \]

(14.16)

where:

\[(u_c, v_c, m_a) = (-v_z, -v_y, v_x) \quad v_x = \max \{|v_x|, |v_y|, |v_z|\},\]

(14.17)

\[(u_c, v_c, m_a) = (v_z, -v_y, v_x) \quad -v_x = \max \{|v_x|, |v_y|, |v_z|\},\]

\[(u_c, v_c, m_a) = (v_x, v_z, v_y) \quad v_y = \max \{|v_x|, |v_y|, |v_z|\},\]

\[(u_c, v_c, m_a) = (v_x, -v_z, v_y) \quad -v_y = \max \{|v_x|, |v_y|, |v_z|\},\]

\[(u_c, v_c, m_a) = (v_x, -v_y, v_z) \quad v_z = \max \{|v_x|, |v_y|, |v_z|\},\]

\[(u_c, v_c, m_a) = (-v_x, -v_y, v_z) \quad -v_z = \max \{|v_x|, |v_y|, |v_z|\}.\]
The Use of Cube Mapping

- Cube mapping is very frequently used for representing 3-D environment extents, such as:
  - distant landscapes,
  - buildings in cityscapes,
  - sky boxes, and
  - sky domes
- When the interpolated normal vector of a surface is used to generate the texture coordinates, cube mapping can be exploited to apply pre-computed diffuse illumination on a surface
Environment Mapping

- Indexes pre-calculated or recorded illumination data representing a “distant” environment
- Relies on the dynamic texture coordinate generation and sub-texture selection using local fragment attributes
- Can use either spherical or cube mapping
- *Environment Mapping:* The general category of mapping-coordinate calculations that treats the texture map as a storage medium for directionally indexed incident light
Reflection Mapping

- Let \( \hat{r} \) be the direction vector that results from reflecting an imaginary ray from the viewpoint to an arbitrary surface point with normal vector \( \hat{n} \)

\[
\hat{r} = 2 \hat{n}(\hat{n} \cdot \hat{v}) - \hat{v} \quad (14.18)
\]

- Vector \( \hat{r} \) points to the direction from which the light from the environment comes, before being reflected

- Reflection direction is used for generating \((u,v)\)-coordinates according to the mapping function

- (14.18) combined with (14.16) & (14.17), implements this idea

- Cube maps have low distortion \( \rightarrow \) ideal for environment mapping
Reflection Mapping (2)
Reflection Mapping (3)

- Reflected environment elements are assumed to reside adequately far from the reflective object
- Otherwise, different, location-dependent reflection maps should be made available during render time by pre-rendering the environment on the texture(s) for each location
- **Common practice:** Render the environment from the center of the object using a 90° square field of view into low-resolution textures 6 times
- Reflected image bending is usually large & the surfaces are not smooth → low-resolution reflection maps work extremely well
- Low-resolution environment textures have the advantage of being able to frequently recalculate them, even in real time
Reflection Mapping (4)
View-Dependent Texture Maps

- View-dependent texture selection goes way back to the first versions of the 3D game engines: *Sprite* bitmaps
- Sprite selection is the simplest form of *image-based rendering (IBR)*:
- Instead of rendering a 3-D entity, the appropriate view of the object is reconstructed from the interpolation or warping of pre-calculated or captured images, accompanied by depth or view-direction information
View-Dependent Texture Maps (2)

- At render-time, the image of the object as seen from the viewpoint is approximated by the closest pre-calculated views.
View-Dependent Texture Maps (3)

- **Advantage of using image-based rendering:** Decoupling of scene complexity from the rendering calculations, providing a constant frame rate, regardless of the detail of the displayed geometry

But:

- **Image-based techniques may become very computationally intensive when:**
  - missing depth information or
  - gaps need to be extrapolated

- **Can be unconvincing if presented information is inconsistent with the current view or lighting**
View-Dependent Texture Maps (4)

- Most popular IBR methods are the simplest ones
- In the easiest case of an IBR impostor, a 3-D placeholder or geometry proxy, is texture-mapped with a view-dependent criterion for selecting/mixing the textures
- This method is very popular in computer games and virtual reality for rendering complex distant geometry (vegetation, crowds)
View-Dependent Texture Maps (5)

- Reverse approach to image-based rendering is QuickTime VR
  - An environment map is constructed from a fixed point in space that represents the view of the 3-D world from that particular vantage point
- This technique is very used in multimedia applications
View-Dependent Texture Maps (6)

- View-dependent texture maps: A hybrid compromise between simple IBR proxies and actual 3-D geometry.
- View-dependent texture maps are used on 3-D objects that represent a simplified version of the displayed geometry.
- Need for alternative, view-dependent texture maps arises when texturing low-polygon meshes for real-time applications.
View-Dependent Texture Maps (7)
View-Dependent Texture Maps (8)

• Way to imprint the geometric detail of the high-resolution model onto the lower-resolution one:
  - render the object from a viewpoint that ensures maximum visibility & project the image as a texture on the low-detail model

• Bump mapping alleviates part of the problem, but cannot provide the correct depth cue and shading/shadow information

• Much more realistic appearance can be achieved by adding multiple (view-dependent textures) at the expense of texture memory
Texture Magnification & Minification

- When a map is applied to a surface, texels are stretched to occupy a certain area, according to:
  - the local spacing of the texture coordinates
  - the size of the primitive being textured

- Projection of a textured surface on the viewing plane → a texel covers a certain portion of the image space. The area depends on:
  - projection-transformation parameters
  - viewport size
  - distance from center of projection (perspective projection)

- When the projected texel in image space covers an area of more than a pixel, it is locally magnified

- In the opposite case, when its footprint is less than a pixel, the texture is minified or compressed
Texture Magnification

- Texture magnification makes intensity discontinuities apparent in a semi-regular manner.
- Step-ladder effect (pixelization) $\rightarrow$ poor texturing:
  - Interpolation methods used for extracting a texture value from the neighboring texels $\rightarrow$ smoothing the texture.
- Bilinear interpolation implemented by hardware rasterizers offers a good compromise between quality & speed:
  - Higher-order filtering generates far better images that can be subjected to further texture magnification.
Texture Minification

- Minification $\rightarrow$ Texture undersampling
- Visual problems are more serious as image-space and time-varying or view-dependent sampling artifacts are produced
- Texture patterns are erratically sampled, leading to a noisy result & a Moire pattern at high frequencies $\rightarrow$ texture aliasing
Texture Minification (2)
Texture Antialiasing

- In terms of signal-theory, the rendering procedure records samples of the textured surfaces at specific locations on the resulting image at a predefined spatial resolution.
- For a signal to be correctly reconstructed, the original signal has to be band-limited & highest frequency must be at most half the sampling rate (uniform sampling theorem).
- **Spatial frequency of texture**: Rate at which a transition from one projected texel on the image plane to the next occurs.
- For texture compression, a texel corresponds to less than 2 pixel samples and when a texture is severely minified, many texels are skipped.
Texture Antialiasing (2)

- 2 solutions to this problem:
  - Texture super-sampling in image space → ensure that source signal is sampled above Nyquist sampling rate and band-limit the resulting signal to image's actual spatial resolution (post-filtering)
  - Band-limit the original signal before rendering the geometry into the image buffer (pre-filtering). Dominant technique for RT-graphics: pyramidal pre-filtering
Texture Antialiasing – Super-sampling

- Only moves the aliasing problem to higher frequencies
Texture Antialiasing – Super-sampling (2)

- **Post-Filtering advantages:**
  - Provides a means for global antialiasing
  - Contributes to the solution of texture aliasing problem when combined with other techniques

- **Post-Filtering disadvantages:**
  - Suffers from an obvious problem in the case of texture mapping
  - Transposes aliasing problem higher in the spatial frequency domain
  - Poor solution in the case of real-time rendering algorithms → not easy to predict the required number of samples & samples are limited by the capacity of graphics system & can decrease rendering performance
Texture Antialiasing – Pre-filtering

- Band-limit the texture signal → create a filter, whose frequency response is 0 outside the band limits & then multiply it with the spectrum of the input texture.

- Try to predict how many texels contribute to the intensity of each pixel after projection:
  - Contributing texels are first averaged (low-pass filtering) and then used for rendering the surface texture.

- Filtering is performed in texture space by convolution of a filter kernel $f(s,t)$ of finite spatial support $G$ with texture values $i(u,v)$:

$$ (f * i)(u,v) = \int \int_G f(s,t) \cdot i(u-s, v-t) ds dt $$

(14.19)
Texture Antialiasing – Pre-filtering (2)

• Matter is not so simple:
  ■ A naive box filter has an infinite impulse response (IIR), i.e., it has an infinite support in the spatial domain

• An IIR filter could not be appropriately applied → we would need an infinite filter kernel

• Many good practical finite impulse response (FIR) low-pass filters & their discrete counterparts exist like:
  ■ B-spline approximation to the Gaussian filter
Texture Antialiasing – Pre-filtering (3)

- In practice filtering is a weighted average in a finite area centered at the sample point \((u,v)\)
- To obtain the texel filtering area, we seek the projection of a pixel in image space to the texture space (\textit{pixel pre-image})
- The pixel pre-image is in general a curvilinear quadrilateral in texture space
Texture Antialiasing – Pre-filtering (4)

- Correct texture sampling:
  - Shape & area of a pixel’s pre-image have to be estimated by mapping its area from image space to texture space
  - Texture values must be integrated over it
- Corresponding filter shape & size need to adapt to the pre-image in order to:
  - appropriately limit the texture spatial frequency
  - avoid unnecessarily blurring of the texture
Texture Antialiasing – Pre-filtering (5)

Observations:

- Larger pixel pre-image → larger number of texture samples need to be averaged and vice versa
- Due to nature of texture mapping & pixel-dependent variance of the pixel pre-image shape:
  - A filter kernel should be estimated for each pixel sampled
- Minification & magnification may occur at the same image location since pixel pre-image may be elongated
Mip-Mapping

- Re-computing or dynamically selecting pre-constructed filter kernels & performing the texture filtering in real time is computationally expensive

- Simplify the problem →:
  - Filter kernel has a constant aspect ratio and orientation
  - Variable size

- Pre-filtered versions of the texture map can be a priori generated and stored

- At render-time, for each pixel:
  - Determine the proper kernel size → obtain the proper pre-filtered version of the texture
Mip-Mapping – Pre-filtering

- Original texture map is recursively filtered & down-sampled into successively smaller versions *(mip-maps)*
- Each mip-map has half the dimensions of its parent
- A simple 2x2 box filter is used for averaging the parent texels to produce the next version of the map
- Result is a hierarchy of mip-maps that represent the result of the convolution of the original image with a square filter
Mip-Mapping – Pre-filtering (2)

- Filter is power-of-two pixels wide in each dimension
- Initial image is the 0th level of the pyramidal texture representation and corresponds to a filter kernel of $2^0 = 1$
- Successive levels are sequentially indexed and correspond to filter kernels of length $2^i$, where $i$ is the *mip-map* level
- For an $N \times M$ original texture, there are at most $\left \lfloor \log_2 (\max(N, M)) \right \rfloor + 1$ levels in the mip-map set
- $i$-th level has dimension given by:
  \[ \max \left( \frac{N}{2^i}, 1 \right) \times \max \left( \frac{M}{2^i}, 1 \right) \] (14.20)
Mip-Mapping – Pyramidal representation
Mip-Mapping – Evaluation

- At each level, bilinear interpolation is used on the nearest discrete texels.
- To approximate an image pre-filtered with a filter kernel of arbitrary size, interpolation between two fixed-size mip-map levels is used.
- A 3\textsuperscript{rd} parameter $d$ is introduced: $d \in [0, \text{level}_{max}]$.
- $d$ moves up and down the hierarchy & interpolates between the nearest mip-map levels.
Mip-Mapping – Evaluation

Implications:

• All filters in the mip-mapping pre-filtering procedure are approximated by linearly blended square box filters:
  ■ Far from the ideal filtering
  ■ Does not take into account any affine transformation of the kernel BUT usually works very well when an appropriate value for $d$ is selected

• The shape of the filter kernel can be non-isotropic by performing mip-mapping individually for each texture coordinate:
  ■ Requires pre-computation of all combinations of mip-map sizes
  ■ $d$ is calculated individually for each texture coordinate

• Computation of $d$ is of critical importance
  ■ Poorly selected $d \rightarrow$ failure to antialias the texture or blurring
Mip-Mapping – Level Selection

- Consider the rate of change of texels in relation to the pixels (fragments) in image space.

- Partial derivatives of the applied texture image with respect to the horizontal and vertical image-buffer directions are:
  \[
  \frac{\partial u'}{\partial x}, \frac{\partial u'}{\partial y}, \frac{\partial v'}{\partial x}, \frac{\partial v'}{\partial y}
  \]
  where: \( u' = u \cdot N, \quad v' = v \cdot M, \quad u, v \in [0,1] \).

- **Simplified case**: (square filter). The linear scaling \( \rho \) of the filter kernel is roughly equal to the max dimension of the pixel pre-image (worst-case aliasing scenario):

  \[
  \rho = \max \left\{ \sqrt{\left(\frac{\partial u'}{\partial x}\right)^2 + \left(\frac{\partial v'}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u'}{\partial y}\right)^2 + \left(\frac{\partial v'}{\partial y}\right)^2} \right\}
  \]

  (14.22)
where $\lambda = \log_2 \rho$ and $\text{level}_{\text{max}} : \text{max} \text{ pre-calculated mip-map level}$
Applying Mip-Mapping

- $d$ can be used either as a nearest-neighbor decision variable or as a 3rd interpolation parameter to perform tri-linear interpolation between adjacent levels:
- Texture pre-filtering with mip-maps significantly speeds up rendering due to the fact that all filtering takes place when the texture is first loaded
- Commodity graphics hardware systems implement the mip-mapping functionality in its full extent
- Mip-mapping performs sub-optimal filtering in the case where the pre-image deviates from a quadrilateral shape
Applying Mip-Mapping (2)

- Limited to algorithms relying on measured quantities in image space: They are screen driven & applicable only to incremental screen-order rendering techniques

- Other rendering methods (e.g. ray-tracing) cannot directly benefit from mip-mapping → no knowledge of a pre-image area for an arbitrary isolated sample on a textured surface
Procedural Textures

- A surface or volume attribute can be:
  - Calculated from a mathematical model
  - Derived in a procedural algorithmic manner

- Procedural Texturing:
  - Does not use intermediate parametric space
  
  Input set of coordinates $\stackrel{\text{directly and uniquely}}{\text{association with}}$ Output texture value

  - Often referred to as procedural shaders

- Can be used to calculate:
  - A color triplet
  - A normalized set of coordinates
  - A vector direction
  - A scalar value
Some forms of a procedural texture:

\[ v = f_{\text{proc}}(p, a), \]
\[ \mathbf{n} = \mathbf{f}_{\text{proc}}(p, a), \]
\[ t = f_{\text{proc}}(p, a) \]

where \( a \): an attribute parameter vector
\( p \): input point

These output parameters can be used as:
- An input to another procedural texture
- A mapping function to index a parametric texture
Properties:

1. Operate on continuous input parameters & generate a continuous output
   - Not blurring when sampling a surface at a high resolution
   - Not suffering from magnification problems

2. No distortion ⇔ no intermediate parametric representation

3. Map the entire input domain to the output domain

- A useful visualization tool
- Until recently, was applicable only for non-real time rendering
- Now, thanks to GPUs, procedural shaders are extensively used
Procedural Textures (4)
Noise Generation

- In nature there are materials and surfaces with irregular pattern, such as a rough wall, a patch of sand, various minerals, stones etc
- Many examples where nature texture looks like a noisy pattern
- The procedural noise texture should:
  - Act as a pseudo-number generator
  - Also exhibit some more convenient & controllable properties
- Noise generators must adhere to a number of rules to ensure a consistent output
Noise Generation (2)

- The procedural noise should be:
  
a. **Stateless**
   - The procedural noise model must be memory-less
   - The new output should not depend on previous stages or past input values
   - Necessary if we want an uncorrelated train of outputs
  
b. **Time-invariant**
   - The output has to be deterministic
   - Avoid dependence of the noise function on clock-based random generators
  
c. **Smooth**
   - The output signal should be continuous and smooth
   - First-order derivatives should be computable
  
d. **Band-limited**
   - A white-noise generator is not useful
   - Should control the max (and min) variation rate of the pattern
Perlin Noise

- Is the most widely used noise function
- Encompasses all the above properties
- Relies on numerical hashing scheme on pre-calculated random values
Perlin Noise (2)

- Assume a lattice formed by all the triplet combinations of integer values so that node $\Omega_{i,j,k}$ lies on $(i, j, k) : i,j,k \in \mathbb{N}$
Perlin Noise (3)

- Every node is associated with a pre-generated pseudo-random number \( \gamma_{i,j,k} \in [-1.0, 1.0] \)

- The procedural noise output is the weighted sum of the values on the 8 nodes nearest to the input point \( p \)

- More specifically, use \( \Omega_{i,j,k} \) as spline nodes, which contribute to the sum as follows:
  \[
  \omega(t) = \begin{cases} 
  2|t|^3 - 3t^2 + 1 & |t| < 1, \\
  0 & |t| \geq 1
  \end{cases}
  \]

- This function has a support of 2, centered at 0

- For an integer \( i, \omega(t-i) \) is max at \( i \) and drops off to 0 beyond \( i \pm 1 \)
Perlin Noise (4)

- The final noise pattern \( f_{\text{noise}}(p) \) for a point \( p = (x, y, z) \)
  - is given by trilinear interpolation of the values \( \gamma_{i,j,k} \) of the 8 lattice points \( \Omega_{i,j,k} \) closest to \( p \)
  - use \( \omega(t - \lfloor t \rfloor) \) as the interpolation coefficient

- For repeatable, time-invariant results
  - \( \gamma_{i,j,k} \) is selected from a table \( G \) of \( N \) pre-computed uniformly distributed scalars using a common modulo-based hashing mechanism:

\[
\gamma_{i,j,k} = G\left[ \text{hash}\left(i + \text{hash}\left(j + \text{hash}(k)\right)\right)\right],
\]

\[
\text{hash}(n) = P[n \mod N]
\]

where \( P \): table containing a pseudo-random permutation of the first \( N \) integers
Turbulence

- Also introduced by Perlin
- Is an extension of noise procedural texture
- Also called 1/f noise function
- Is a band-limited noise function
- Has a spectrum profile whose magnitude is inversely proportional to the corresponding frequency
- Overlay suitably scaled harmonics of a basic band-limited noise function:

\[ f_{\text{turb}}(p) = f_{1/f}(p) = \sum_{i=1}^{\text{octaves}} \frac{1}{2^i f} f_{\text{noise}}(2^i f \cdot p) \]

where \( f \): the base frequency of the noise

\textit{octaves}: the max number of overlaid noise signals
Turbulence (2)

- 1/f noise pattern composition
Turbulence (3)

- The visual result is that of Brownian motion
- A more generalized form:

\[ f_{1/\sigma}(p) = \sum_{i=1}^{\text{octaves}} \omega_i \cdot f_{\text{noise}}(\lambda_i \cdot p) \]

where \( \omega > 0 \): regulates the contribution of higher frequencies

\( \lambda > 1 \): modulates the chaotic behavior of the noise

- When \( \lambda \rightarrow 1 \Rightarrow f_{1/\sigma}(p) \):
  - Appears as a scaled version of noise (p)
  - Larger values give a more swirling look to the result
- If \( \hat{f}_{1/\sigma}(p) \) is the resulting offset of point \( p \) after performing a random walk:
  - \( \omega \) = corresponds to the speed of motion
  - \( \lambda \) = simulates the entropy of the system under Brownian motion
Turbulence (4)

• Many interesting patterns can be generated by:

\[ f_{\text{proc}}(p) = f_{\text{math}}(f_{\text{turb}}(p)), \]

\[ f_{\text{proc}}(p) = f_{\text{math}}(p + \alpha f_{\text{turb}}(p)) \]

• The noise function can act as:
  - A bias to the input points
  - Part of a composite function
Common 3D Procedural Textures

A. Solid Checker Pattern

- Interleaved solid blocks of 2 different colors
- Using a texture image at an arbitrary resolution → blurred at checker limits

\[ f_{\text{checker}}(\mathbf{p}) = (\lfloor x \rfloor + \lfloor y \rfloor + \lfloor z \rfloor) \mod 2 \]
B. Linear Gradient Transition

- Is a useful pattern and easy to implement
- Produces a high quality smooth transition from one value to another
- There is no danger of generating perceivable bands
- When using texture maps with fixed-point arithmetic, these bands are a result of color quantization
- Can use many alternative input parameters, i.e.:
  - single Cartesian coordinates
  - spherical parameters
- Its simplest form: a ramp along a primary axis:

\[ f_{\text{gradient}}(p) = y - \lfloor y \rfloor \]
Common 3D Procedural Textures (3)

- Natural formations are combinations of:
  - A base mathematical expression
  - Turbulence
  - Noise

C. Wood
- Represented as an infinite succession of concentric cylindrical layers
- Modeled by a ramp function over the cylindrical coordinate $r$
- Add an amount of perturbation $a$ to the input points
- Use an absolute sine or cosine function to accent the sharp transition between layers without discontinuity:

$$f_{\text{wood}}(p) = \left| \cos \left( 2\pi \left( d - \lfloor d \rfloor \right) \right) \right|,$$

$$d = \sqrt{y^2 + z^2} + a \cdot f_{\text{turb}}(p)$$
D. **Marble**

- Use a smoothly varying function to generate the compressed earth layers
- Perturb the input parameters to get a very realistic approximation

\[ f_{\text{marble}}(p) = \frac{1}{2} + \sin\left(2\pi\left(x + f_{\text{turb2}}(p)\right)\right), \]

\[ f_{\text{turb2}} = \sum_{i=1}^{\text{octaves}} \frac{1}{2^i} \left| f_{\text{noise}}(2^i \cdot f \cdot p) \right| \]
Texture Transformations

- To produce turbulence:
  - successively higher frequencies of the texture were overlaid on each other
  - what mechanism was employed to shrink the texture?
- When a point \( p \) is expressed as a set of coordinates \((x, y, z)\) relative to a reference coordinate \( L = \{\mathbf{o}, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\} \):
  
  transform \( p \rightarrow \) inversely transform the reference frame:

  \[
  p' = (x', y', z') = \mathbf{M}(p) : p = \mathbf{o} + x\mathbf{e}_1 + y\mathbf{e}_2 + z\mathbf{e}_3 \quad \Leftrightarrow \\
  p' = \mathbf{o}' + x'\mathbf{e}'_1 + y'\mathbf{e}'_2 + z'\mathbf{e}'_3 : \{\mathbf{o}', \mathbf{e}'_1, \mathbf{e}'_2, \mathbf{e}'_3\} = \mathbf{M}^{-1}(\{\mathbf{o}, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3\})
  \]

- It allows for the creation of procedural texture variations without requiring any modification of the pattern shader itself
- E.g. a gradient along the \( x \)-axis

  \[
  f_{\text{gradient}X}(p) = \mathbf{R}_{-90,z} \left( f_{\text{gradient}}(p) \right) = f_{\text{gradient}}(\mathbf{R}_{90,z} \cdot p)
  \]
Texture Transformations (2)

• Any texture transformation can be implemented by applying the inverse transformation to the input domain:

\[ M(f_{\text{proc}}(\mathbf{p})) = f_{\text{proc}}(M^{-1} \cdot \mathbf{p}) \]

• This transformation also applicable to parametric texture mapping
• Is easier to inversely transform the vertices for which the new \((u,v)\)-coordinates are to be estimated.

• Flexibility to perform transformations in the texture parameter domain as well
• Texture coordinates can be directly shifted, rotated, and translated
Relief Representation

- A texture map can be used:
  - To alter the material characteristics of a surface
  - To modulate its geometry to create patterns in relief
  - To imitate the apparent effect of a bumpy surface without extra detail

- Three approaches:
  a) **Displacement mapping**
     Distorts the geometry to create the relief pattern
  b) **Bump mapping + Normal mapping**
     Creates the illusion of a bump surface without altering the actual surface
  c) **Parallax mapping**
     Advanced technique that ray-traces relief on a surface according to an elevation map
Displacement Mapping

- Creates relief patterns on a surface
- Moves the vertices along the original surface normal direction or along a predefined vector according to the intensity of the texture
- Bump map:
  - Is a scalar-valued map used in both the displacement + bump mapping
  - Is the elevation pattern
- It requires that:
  - The surface is highly tessellated from the beginning, or
  - An adaptive algorithm adds more surface elements at high slope areas
- Cannot be applied on a per-pixel basis during a shading illusion
  (+): it generates a pattern in relief and not a shading illusion
  (-): it is unfavorable for real-time rendering due to the highly detailed surface.
Displacement Mapping (2)

- Darker bump-map values represent high elevation
Bump Mapping

• Is a method for faking the appearance of detailed bumpy surfaces
• Let $b(u)$: the elevation pattern and $s = s(u)$: the surface location
• The displacement surface is $s'(u) = s(u) + b(u)\hat{n}(u)$
• The new surface has a different normal vector $\hat{n}'(u)$
• How to trick the eye:
  ■ Only the normal vector contributes to the shading calculation
  ■ Calculate how the normal vector would be perturbed if the surface was elevated
  ■ The surface points are not actually moved
Bump Mapping (2)

- Given a surface parameterization \((u, v): s = s(u, v)\)
- The local unit-length normal vector \(\hat{n} = \hat{n}(u, v)\) of the surface is:

\[
\hat{n} = \hat{u} \times \hat{v},
\]

\[
\hat{u} = \frac{\vec{t}}{|\vec{t}|}, \quad \hat{v} = \frac{\vec{b}}{|\vec{b}|},
\]

\[
\vec{t} = \frac{\partial s(u, v)}{\partial u}, \quad \vec{b} = \frac{\partial s(u, v)}{\partial v}
\]

- The normalized vectors on the tangent plane at \(s(u,v)\):
  - \(\hat{u}\) = the tangent vector
  - \(\hat{v}\) = the bitangent vector (also called binormal vector)
The elevated surface $s'(u, v)$ is given by:

$$s'(u, v) = s(u, v) + \hat{n}(u, v) \cdot b(u, v) \quad (1)$$

The perturbed normal vectors:

$$\hat{n}' = \hat{\mathbf{u}}' \times \hat{\mathbf{v}}' = \frac{\partial s'(u, v)}{\partial u} \times \frac{\partial s'(u, v)}{\partial v} \quad (2)$$

Evaluate the partial derivatives:

$$\frac{\partial s'(u, v)}{\partial u} = \frac{\partial s(u, v)}{\partial u} + \frac{\partial \hat{n}(u, v)}{\partial u} \cdot b(u, v) + \hat{n}(u, v) \cdot \frac{\partial b(u, v)}{\partial u}, \quad (3)$$

$$\frac{\partial s'(u, v)}{\partial v} = \frac{\partial s(u, v)}{\partial v} + \frac{\partial \hat{n}(u, v)}{\partial v} \cdot b(u, v) + \hat{n}(u, v) \cdot \frac{\partial b(u, v)}{\partial v}$$
For slow (smooth) normal vector variations, $\frac{\partial \hat{n}}{\partial u}$ and $\frac{\partial \hat{n}}{\partial v}$ are negligible, so (3) becomes:

$$
\frac{\partial s'(u, v)}{\partial u} = \frac{\partial s(u, v)}{\partial u} + \hat{n}(u, v) \cdot \frac{\partial b(u, v)}{\partial u} = \hat{t} + \hat{n}(u, v) \cdot \frac{\partial b(u, v)}{\partial u},
$$

$$
\frac{\partial s'(u, v)}{\partial v} = \frac{\partial s(u, v)}{\partial v} + \hat{n}(u, v) \cdot \frac{\partial b(u, v)}{\partial v} = \hat{b} + \hat{n}(u, v) \cdot \frac{\partial b(u, v)}{\partial v}
$$

Substituting the partial derivatives of (4) into (2):

$$
\hat{n}' = \begin{pmatrix} \hat{t} + \hat{n} \cdot \frac{\partial b(u, v)}{\partial u} \\ \hat{b} + \hat{n} \cdot \frac{\partial b(u, v)}{\partial v} \end{pmatrix} \times \begin{pmatrix} \hat{b} + \hat{n} \cdot \frac{\partial b(u, v)}{\partial v} \\ \hat{t} + \hat{n} \cdot \frac{\partial b(u, v)}{\partial u} \end{pmatrix} = \hat{t} \times \hat{b} + \hat{t} \times \hat{n} \cdot \frac{\partial b(u, v)}{\partial v} + \hat{n} \times \hat{b} \cdot \frac{\partial b(u, v)}{\partial u} + \hat{n} \times \hat{n} \cdot \frac{\partial b(u, v)}{\partial u} \frac{\partial b(u, v)}{\partial v} \Rightarrow
$$

$$
\hat{n}' = \hat{n} - \hat{b} \cdot \frac{\partial b(u, v)}{\partial v} - \hat{t} \cdot \frac{\partial b(u, v)}{\partial u}
$$

The new perturbed normal vector:

- Is expressed relative to the object world coordinate system
- Depends only on the tangent vectors + the partial derivatives of the bump map
Displacement Vs Bump Mapping

- **Displacement mapping**
  
  (+): produces accurate elevation parallax
  
  (-): has a high geometry tessellation

- **Bump mapping**

  (+): very simple & convincing

  (-):  
  1. poor representation on deep depressions
  2. lacks in accuracy
  3. fails to produce correct edges
Normal Maps

• The bump mapping method:
  ■ Uses as input an elevation image map (i.e. a grayscale elevation image)

• Normal mapping: the normal vector used in the shading calculation is directly supplied

• Object-space normal mapping:
  ■ Must transfer a pre-calculated relief pattern onto a model
  ■ The normal vector is stored as a color-coded triplet an a bitmap file
  ■ R,G,B components of the image \( c(u,v) \) keep properly scaled and quantized Cartesian coordinate values of the unit-length vector:

\[
\hat{n}(u, v) = \frac{c(u, v)}{2^{N\text{bits}}} - (0.5, 0.5, 0.5)
\]

where \( N\text{bits} \): the number of bits used for storing each color channel
Normal Maps (2)

- Advantages
  1. The normals can be calculated intuitively from high resolution surfaces and stored in normal maps on simplified texture-mapped version
  2. Significantly fewer calculation to derive the local normal & perform shading

- Disadvantages
  - Need to be performed along with the object
  - When an object undergoes any kind of distortion, the object space normals are wrong and result in unconvincing images
Tangent-Space Normal Maps

- Overcome the limitations of object-space normal mapping
- Normal vectors defined using the local tangent space as a reference frame
- Look like edge-filtered versions of the relief patterns
- Are immune to transformations and deformations
- Support texture tiling
- require some extra calculations to integrate them with a shading model
The light-direction (\( \hat{n}_L \)) and view-direction (\( \hat{n}_V \)) vectors have to be transformed to the local tangent-space coordinate system.

After replacing the normal direction according to the normal map values, we get a new normal vector \( \hat{n}'(u,v) \).

The resulting coordinate system is not necessarily orthonormal.

Applying the Gram-Schmidt orthogonalization formula and then normalizing the tangent vectors we get:

\[
\hat{u}' = \hat{u} - (\hat{n}' \cdot \hat{u}) \cdot \hat{n}',
\]

\[
\hat{v}' = \hat{v} - (\hat{n}' \cdot \hat{v}) \cdot \hat{n}' - (u' \cdot \hat{v}) \cdot \hat{u}'
\]
Tangent-Space Normal Maps (3)

• To move a WCS vector (e.g. view and light vectors) to the tangent coordinate system, use the change-of-basis transformation:

\[
R_{\text{Tangent}} = \begin{bmatrix}
  u'_x & u'_y & u'_z & 0 \\
  v'_x & v'_y & v'_z & 0 \\
  n'_x & n'_y & n'_z & 0 \\
  0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
Tangent-Space Calculation

- Given an arbitrary texture-mapped point \( s \) on a polygonal surface
  - Calculate the object-space tangent vectors \( \mathbf{t} \) and \( \mathbf{b} \)
  - Calculate the respective normalized ones \( \mathbf{u} \) and \( \mathbf{v} \)
- Let the texture coordinate pairs of a triangle \( p_1p_2p_3 \) be:
  
  \[(u_1,v_1),(u_2,v_2),(u_3,v_3)\]
- Constrain the tangent and bitangent vectors to be locally parallel to the texture isoparametric curves
- Express an arbitrary point \( p \) with respect to the tangent and bitangent vectors by:

\[
p = p_1 + (u-u_1)\mathbf{u} + (v-v_1)\mathbf{v}
\]
Tangent-Space Calculation (2)

- All 3 triangle vertices share the same tangent coordinate system:
  \[
  \begin{align*}
  \textbf{p}_2 - \textbf{p}_1 &= (u_2 - u_1) \mathbf{\hat{v}} + (v_2 - v_1) \mathbf{\hat{v}}, \\
  \textbf{p}_3 - \textbf{p}_1 &= (u_3 - u_1) \mathbf{\hat{v}} + (v_3 - v_1) \mathbf{\hat{v}}.
  \end{align*}
  \]

- In a more compact form,
  \[
  \begin{align*}
  \mathbf{\hat{q}}_2 &= u_{21} \mathbf{\hat{u}} + v_{21} \mathbf{\hat{v}}, \\
  \mathbf{\hat{q}}_3 &= u_{31} \mathbf{\hat{u}} + v_{31} \mathbf{\hat{v}}.
  \end{align*}
  \]

- Express the above linear system in a matrix form with respect to the six unknown coordinates of \( \mathbf{\hat{t}}, \mathbf{\hat{b}} \):
  \[
  \begin{bmatrix}
  u_{21} & v_{21} \\
  u_{31} & v_{31}
  \end{bmatrix}
  \begin{bmatrix}
  t_x & t_y & t_z \\
  b_x & b_y & b_z
  \end{bmatrix}
  =
  \begin{bmatrix}
  q_{2x} & q_{2y} & q_{2z} \\
  q_{3x} & q_{3y} & q_{3z}
  \end{bmatrix}
  \]
Tangent-Space Calculation (3)

- Solve this linear system, using the determinant formula for the inversion of the coefficient matrix:

\[
\begin{bmatrix}
  t_x & t_y & t_z \\
  b_x & b_y & b_z \\
\end{bmatrix} = \frac{1}{u_{21}v_{31} - u_{31}v_{21}} \begin{bmatrix}
  v_{31} & -v_{21} \\
  -u_{31} & u_{21} \\
\end{bmatrix} \begin{bmatrix}
  q_{2x} & q_{2y} & q_{2z} \\
  q_{3x} & q_{3y} & q_{3z} \\
\end{bmatrix}
\]

- The resulting coordinate system is
  - not normalized
  - possibly not even orthogonal.

- The vectors should be adjusted and then normalized

- \( \vec{t} \) and \( \vec{b} \) represent the triangle's tangent space
Texture Atlases

- A *texture atlas* is a surface parameterization where connected parts of the object's surface (*charts*), are each mapped onto contiguous regions of the texture domain:
  - An object is 1\textsuperscript{st} partitioned into charts
  - Each chart is parameterized so that the surface patch is unfolded on a 2-D domain to ensure a unique mapping for each point in the chart
    - At this stage, texture-domain pieces may overlap
  - Finally, individual map pieces & corresponding parameter ranges are packed in a single texture → charts no more overlap in texture space
- Final atlas ensures the unique mapping between Cartesian coordinates on the surface & locations on the bounded texture domain of the image map
Texture Atlases (2)

- Numerous algorithms have been proposed for each one of the 3 stages of the texture-atlas-generation procedure
- Parameterization stage depends on the constraints chosen for the surface segmentation
- When cutting a surface patch and unfolding it into the 2-D parameter space, a set of implementation-dependent criteria must be satisfied in order to:
  - Minimize texture distortion and artifacts
  - Distribute the texels over the surface as evenly as possible
  - Ensure continuity & conformity of mapping among the charts, if possible
  - Maximize the area coverage of the charts & minimize the # of separate charts
The 3rd stage, i.e., texture packing, is a NP-complete problem:
- A number of objects of different volumes need to be packed into a finite set of bins, while minimizing the vacant space.

In 2-D case of texture-atlas packing, a finite # of atlas elements of known shape need to be arranged in the final texture map → gaps are minimized & the coverage of the atlas texture is maximized.

Only near-optimal approximate solutions have been proposed.

“Polypacks”: Simple approach & relatively easy-to-implement solution for texture atlas parameterization & a common parameterization method.

Algorithm is complemented with the kd-tree approach to texture packing.
Surface Segmentation - Polypacks

- Cut surface into regions (*polypacks*) and map each one to a plane with as little distortion as possible.
Surface Segmentation – Polypacks (2)

- **Easiest way:** Cut surface into areas of connected polygons that face the same primary half-axis and use the corresponding plane as the projection plane for the planar mapping.
- **Problem:** For relatively complex surfaces & models with creases or irregular curved regions, there are too many polypacks produced which are also relatively small.
Surface Segmentation – Polypacks (3)

- Seek to minimize the number of charts in the final atlas because:
  - Unused space in the atlas texture is increased with the # of atlas elements
  - Seams between adjacent textured patches are noticeable as the texture parameterization changes in scaling or direction across the surface
- If bilinear filtering or mip-mapping is used:
- Averaging of neighboring texels →
- We need extra space between charts to act as guard space and avoid averaging texels of different surface regions
Texture Allocation and Packing

- After splitting & parameterizing the surface, each cluster is mapped to the normalized parametric domain ([0,1],[0,1])
- To avoid stretching the texture sub-image of each chart:
- Each chart needs to be assigned a bitmap, whose aspect ratio \( r(i) \) matches the aspect ratio of the planar projection of the \( i \)-th polygon cluster, prior to packing
- Size \( N_{\text{texels}}(i) \) of each atlas element in texels is decided according to the ratio of the area coverage \( A_{\text{proj}}(i) \) of the bounding rectangle of the projected polygons on the plane to the sum of all \( A_{\text{proj}}(i) \):
  \[
  N_{\text{texels}}(i) = \alpha N_{\text{total}} \cdot \frac{A_{\text{proj}}(i)}{\sum_{j=1}^{\text{Charts}} A_{\text{proj}}(j)}, \quad 0 < \alpha < 1
  \]
  \[(14.52)\]
Texture Allocation and Packing (2)

• Dimensions of $i$-th atlas element are easily determined by the aspect ratio and the number of texels allocated to it:

\[
\begin{align*}
    w_i \cdot h_i &= N_{\text{texels}}(i) \\
    h_i &= \sqrt{\frac{N_{\text{texels}}(i)}{r(i)}} \\
    w_i &= r(i) \cdot h_i
\end{align*}
\]  

(14.53)

• When inversely projecting a texel onto a polygon of the polygon cluster $\rightarrow$ resulting patch is a parallelogram

• There is small distortion in the texture parameterization phase $\rightarrow$ patch is approximately square:

  - Surface is almost uniformly sampled & the texture atlas can be ideally used for surface resampling or the mapping of pre-calculated illumination
Texture Allocation and Packing (3)

- To Avoid using texture values from neighboring elements in the final packed atlas texture $\rightarrow$ texture coordinates of the polygon clusters need to be pulled inward to create:
  - “Active texels”: Texels actually mapped to the polygons after shrinking the texture coordinates
  - “Sand texels”: Unused pixels left in the atlas element outside the active texels (guard space)

- After calculating texture values for active texels, sand texel values are iteratively extrapolated or copied from the nearest texels already evaluated
Texture Allocation and Packing (4)

- Transformation of texture coordinates in an atlas element in order to leave guard space around the parameterized polypack:

- $N_{\text{sand}}$ are the desired # of unused texels on each side of the atlas element

- Corresponding transformation matrix for shrinking the $i$-th polygon cluster texture coordinates is given by:

$$
M_{\text{sand}} = T \left( \frac{1}{w_i} N_{\text{sand}}, \frac{1}{h_i} N_{\text{sand}} \right) S \left( \frac{w_i - 2N_{\text{sand}}}{w_i}, \frac{h_i - 2N_{\text{sand}}}{h_i} \right)
$$

(14.54)
Texture Allocation and Packing (5)

- Texture packing is performed by recursively partitioning the final atlas texture.
- Method presented performs a non-uniform binary partitioning of the image space that results in an unbalanced *kd-tree*.
• Internal nodes of the tree are decision nodes corresponding to the splitting lines of an image partitioned into 2 non-equal areas, either vertically or horizontally

• Leaf nodes represent an empty space or a region occupied by a texture map

• At each internal level \( i \) of tree traversal, the \( i \mod(k) \)-coordinate of the \( k \)-D input value is compared with corresponding threshold stored in the node to select left or right branch

• Alternatively, the dimension leaving most space an occupied can be used
Texture Allocation and Packing (7)

- Construction of a texture atlas by packing individual chart textures with a kd-tree binary image subdivision:

E = Empty space
Tx = Texture of chart x
H = Horizontal split
V = Vertical split
Texture Allocation and Packing (8)

• Initially: 1 empty node, representing entire atlas texture space
• Before inserting any elements into the tree, sort elements by their longest edge →
  - Insert the bulkiest subtextures first
  - Avoid any unnecessary fragmentation of the available space
Texture Allocation and Packing (9)

- New elements traverse the tree in search of an empty space that can accommodate them
- Procedure is repeated until all elements have been placed in the texture atlas or there is an error encountered in fitting a map
- **Why the packing may fail:** Size of the atlas texture is too small to accommodate the atlas elements
- An element must be at least 1x1 texels in size:
  - Furthermore, leaving some guard space requires chunks of at least 3x3 pixels
- Final texture is too small \( \rightarrow \) no way to fit the subtextures into the atlas
Texture Allocation and Packing (10)

- Insertion of an element into the tree structure as well as the basic node definition are described in the following piece of code:

```cpp
class TreeNode
{
    //DATA MEMBERS:
    //atlas texture area extents subtended by the node
    int width, height, xmin, ymin, xmax, ymax;
    //node type: branch (vert/horiz), empty, leaf (atlas element)
    int type;
    // In case of leaf node, pointer to allocated element bitmap
    TextureMap *map;

    //MEMBER FUNCTIONS:
    TreeNode(TreeNode *parent, int minx, int maxx, int miny, int maxy);
    bool isBranch(); bool isLeaf(); bool insert(Element * element);
};
```
bool TreeNode::insert (Element * element) {
    // Case A. terminal, occupied node: cannot insert
    if (isLeaf ())
        return false;

    // CASE B. Branch node, try inserting in children nodes
    if (isBranch())
    {
        if (child[0]->insert (element)) // try to insert in left child
            return true;
        else
            if (child[1]->insert (element)) // failed, try the right child
                return true;
            else // failed, element cannot fit
                return false;
    }
}
//CASE C. Unused node, try inserting element
//1) Check if the remaining space is adequate for this element
if (width < element->width || height < element->height)
{
    // doesn't fit as is, try to fit it sideways:
    if (height < element->width || width < element->height)
    {
        // doesn't fit either way, insertion failed
        return false;
    }
    else
    {
        // fits sideways, rotate the element by swapping params
        element->swapUVs();
    }
}
// 2) Choose splitting direction, split space and insert element
if (width - element->width >= height - element->height)
{
    // i. if the map leaves more space horizontally, split the
    // cell horizontally, creating a left and a right child
type = NODE_HORIZONTAL_SPLIT;
child[CHILD_LEFT] = new TreeNode (this,
    xmin, xmin + element->width - 1, ymin, ymax);
child[CHILD_RIGHT] = new TreeNode (this,
    xmin + element->width, xmax, ymin, ymax);
    // Now split the left child vertically into:
    // a leaf node (top)...
child[CHILD_LEFT]->type = NODE_VERTICAL_SPLIT;
child[CHILD_LEFT]->child[CHILD_TOP] =
    new TreeNode (child[CHILD_LEFT], xmin, xmin + element->width - 1,
        ymin,
ymin + element->height - 1);
child[CHILD_LEFT]->child[CHILD_TOP]->type = NODE_LEAF;
child[CHILD_LEFT]->child[CHILD_TOP]->map = element->map;
// ... and an empty node (bottom)
child[CHILD_LEFT]->child[CHILD_BOTTOM] =
    new TreeNode (child[CHILD_LEFT], xmin, xmin + element->width - 1,
                  ymin + element->height, ymax);
}
else
{
    // ii. split the cell vertically, creating a top and bottom
    child
    type = NODE_VERTICAL_SPLIT;
    child[CHILD_TOP] = new TreeNode (this, xmin, xmax, ymin, ymin +
                                      element->height - 1);
    child[CHILD_BOTTOM] = new TreeNode (this, xmin, xmax, ymin +
                                         element->height, ymax);
// Now split the top child into a leaf node (left)...
child[CHILD_TOP]->type = NODE_HORIZONTAL_SPLIT;
child[CHILD_TOP]->child[CHILD_LEFT] = new TreeNode (child[CHILD_TOP],
        xmin, xmin + element->width - 1, ymin, ymin + element->height - 1);
child[CHILD_TOP]->child[CHILD_LEFT]->type = NODE_LEAF;
child[CHILD_TOP]->child[CHILD_LEFT]->map = element->map;
// ... and an empty node (right)
child[CHILD_TOP]->child[CHILD_RIGHT] = new TreeNode (child[CHILD_TOP],
        xmin + element->width, xmax, ymin, ymin + element->height - 1);
}
return true;
}
There is a more complex packing approach:
- Suitable for large polygon charts with low compactness
- Operates in the discrete texture space

Functionality:
- Rotate the charts so that their longest diameter is vertically aligned
- Sort charts according to height and insert into the atlas
- Incoming charts are stacked on top of the existing clusters in the atlas
- Topmost texels occupied by the charts already in the atlas form a "horizon", which the new chart's underside texels ("bottom horizon") cannot penetrate
- New chart's position is optimized so that the space left between the existing horizon and the bottom horizon is minimized
- Then, horizon is updated, taking into account the upper texels of the new chart
Applications of Texture Atlases

• Storage of pre-calculated ("baked"), view-independent illumination
  - A 3-D model is parameterized into a texture atlas (light map or illumination map) & the incident direct & indirect diffuse illumination is stored in the texels of the map

• When object is rendered, the pre-recorded information on the light map is used, provided that:
  - Geometry is part of a static environment
  - Moving objects' contribution to diffuse illumination is negligible

• Above assumption is valid for most static 3-D environments → light-mapping is extensively used for the accelerated real-time rendering of realistic scenes

• In practice: Resolution of the light map does not need to be very high → illumination varies more slowly on a surface than a color or bump pattern
Applications of Texture Atlases (2)

- For most cases: Surface already has at least 1 set of texture parameters, associated with the modulation of the surface material:
  - Separate set of parameters for light mapping is stored on the polygon vertices
- Light map is applied as a 2nd pass to the surface, modulating the underlying high-detail color & bump shading
- In hardware, where multiple texture units operate in parallel → different textures are blended in 1 pass:
  - Making rendering overhead of the pre-calculated illumination negligible
- If object to be normal-mapped is not a trivial model case or if it does not bear any repetitive geometric features → texture atlas is necessary for its parameterization
Applications of Texture Atlases (3)

- Extending the idea behind the normal mapping, *geometry images* store in the R, G, and B components of the texture the surface locations that correspond to each texel of the object’s atlas
  - Above efficient 3-D representation provides a regular sampling of the surface and can be used for 3-D pattern recognition (on 2D input data), easy multi-resolutional object representation, fast transmission, re-meshing & many other important applications
Texture Hierarchies

- Complex surface materials & finishes can be achieved by using parametric & procedural textures in a hierarchical tree-like structure (a *texture tree*)
- Individual textures can be blended, multiplied, added, or combined to produce a new output
- Output of one texture can be used as input to another or as a weighting function in an interpolated blending of textures
- Texture trees contain texture transformations, transfer function filters or output format converters
- Texture trees may be utilized to calculate any material attribute
- In a texture tree, nodes can be instantiated
Hierarchical textures are heavily used in off-line rendering

- Allow the creation of material libraries consisting of basic, reusable building blocks that can be combined in an easy & reconfigurable manner to produce the final result
- In real-time rendering, texture trees are implemented via the use of multiple texture units & hardware texture combiners, along with fragment shader programs that are executed in the GPUs' programmable cores
Texture Hierarchies (3)

- Practical example of texture hierarchies:

![Texture Hierarchy Diagram](image)
Texture Hierarchies (4)

- Complex surface finishes can be achieved by hierarchically combining textures to model material attributes: