Polygon Shading and Texture Mapping
Ray tracing Pipeline

Review

- Ray tracer produces *visible samples* from model
- Samples convolved with filter to form pixel image

![Ray tracing Pipeline diagram]
Rendering Polygons

Need to render triangle meshes

- Ray tracing and implicit surface model definition go well together
- Many existing models and modeling apps are based on polygon meshes. How do we render them?
  - Ray trace polygons
- Ray-Triangle intersection not too hard
  - ray-intersect plane of the triangle.
  - check if resulting point is inside the triangle
- Ray-Polygon intersection is similar
  - decompose polygon into triangles
- Number of polygons is a real problem
  - typical mesh representations have hundreds or thousands of triangles and each has to be considered in our intersection tests
- Traditional hardware polygon pipeline is faster
  - uses very efficient, one-polygon-at-a-time, Z-buffer Visible Surface Determination algorithm
  - uses approximate shading rule to calculate most pixels’ color values
Polygonal rendering w/ Z-buffer

- Polygons (usually triangles) approximate actual geometry
- One polygon at a time, strictly local illumination model has crudest approximation to physically-based lighting at each sample point
- *Shading* (typically using linear interpolation) approximates lighting of each sample point. It’s fast, looks ok, especially for small triangles
- Per-pixel (incremental) calculation and comparison of $z$ (depth) values
  - simple arithmetic done in hardware much faster (but cruder) than in software, e.g., solving ray intersection with arbitrarily complex implicit surface equations for each pixel sample

### Viewing: Geometry Processing

<table>
<thead>
<tr>
<th>Conservative VSD</th>
<th>Lighting at Vertices</th>
<th>Transform Vertices</th>
<th>Conservative VSD</th>
<th>VSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selective traversal of object database (or traverse scene graph to get CTM)</td>
<td>Calculate light intensity at vertices (lighting model of choice)</td>
<td>To canonical view volume</td>
<td>Back-face Culling</td>
<td>View Volume Clipping</td>
</tr>
</tbody>
</table>

### Rendering Pixel Processing

<table>
<thead>
<tr>
<th>Image-precision VSD</th>
<th>Shading</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare pixel depth (Z-buffer)</td>
<td>Interpolate vertex color values (Gouraud) or normals (Phong)</td>
</tr>
<tr>
<td>More: See Next Slide</td>
<td></td>
</tr>
</tbody>
</table>

- Conservative VSD: trivial reject only
- N.B: Simplified from actual pipelines (no shadow or texture maps, or any other kinds of maps, nor anti-aliasing, transparency, ...)
Shader-based Pipeline

- Programmable GPU, still based on triangle meshes
  - allows more advanced lighting and shading models, other effects
  - programmable vertex shaders – part of your modeler assignment

Physical detail like the vents on hood are created completely in shader

Reflections, refractions, and color of car are also modified in shader using bump, normal, environment maps, and applying more lighting equations at each vertex.

(see slide 43 for more detail about car)

Former TA Alex Rice:
Deformation of Surface due to Collision

Detail has also been added to models in scene.

Did you notice water in front of statue? The left has just a texture mapped plane while the right has lighting effects, such as refraction, that can be faked with shaders.

Created with NVIDIA’s NV40 shader API and a GeForce Series6 GPU
Shading Models Compared

Constant shading: no interpolation, pick a single representative intensity and propagate it over entire object. Loses almost all depth cues.

Pixar “Shutterbug” images from:
www.siggraph.org/education/materials/HyperGraph/scanline/shade_models/shading.htm
Shading Models Compared (cont.)

Flat or Faceted Shading:
Constant intensity over each face

Constant shading:
Shading Models Compared (cont.)

Gouraud Shading: Linear Interpolation of intensity across triangles to eliminate edge discontinuity
Note: Silhouette edges may need special treatment

Flat Shading:
Shading Models Compared (cont.)

Phong Shading: Interpolation of vertex surface normals.
Note: specular highlights but no shadows - pure non-global illumination model.

Gouraud Shading:
Shading Models Compared (cont.)

Global Illumination: Global illumination model with shadows, texture, bump, and reflection mapping

Phong Shading:
Polygon Mesh Shading (1/5)

Interpolating illumination for speed

- Faceted shading
  - single illumination value per polygon
  - if polygon mesh approximates a curved surface, faceted look is a problem
  - facets exaggerated by mach banding effect
Mach banding review

- Mach band effect: discrepancies between actual and perceived intensities due to bilateral inhibition
- A photoreceptor in the eye responds to light according to the intensity of the light falling on it \textit{minus} the activation of its neighbors (mutual inhibition)

\[ I(c_j) = e(c_j) - \sum_{k \neq j} \alpha_k e(c_k) \quad 0 \leq \alpha_k \leq 1 \]

Mach banding applet:
http://www.nbb.cornell.edu/neurobio/land/OldStudentProjects/cs490-96to97/anson/MachBandingApplet/
Illumination intensity interpolation

- **Gouraud shading**
  - use for polygon approximations to curved surfaces
- Linearly interpolate intensity along scan lines
  - eliminates intensity discontinuities at polygon edges; still have gradient discontinuities, mach banding is largely ameliorated, not eliminated
  - must differentiate desired creases from tesselation artifacts (edges of cube vs. edges on tesselated sphere)

- **Step 1**: since vertices don’t have normals, calculate bogus vertex normals as average of surrounding polygons’ normals:

\[
\vec{N}_v = \frac{\vec{N}_1 + \vec{N}_2 + \vec{N}_3 + \vec{N}_4}{\| \vec{N}_1 + \vec{N}_2 + \vec{N}_3 + \vec{N}_4 \|}
\]

More generally:
\[
\vec{N}_v = \frac{\sum_{i=1}^{n} \vec{N}_i}{\| \sum_{i=1}^{n} \vec{N}_i \|}
\]

- neighboring polygons sharing vertices and edges approximate smoothly curved surfaces and won’t have greatly differing surface normals; therefore this approximation is reasonable
Illumination intensity interpolation (cont.)

- **Step 2**: interpolate intensity along polygon edges
- **Step 3**: interpolate along scan lines

\[
I_a = I_1 \frac{y_s - y_2}{y_1 - y_2} + I_2 \frac{y_1 - y_s}{y_1 - y_2} \\
I_b = I_1 \frac{y_s - y_3}{y_1 - y_3} + I_3 \frac{y_1 - y_s}{y_1 - y_3} \\
I_p = I_a \frac{x_p - x_a}{x_b - x_a} + I_b \frac{x_p - x_a}{x_b - x_a}
\]
**Illumination intensity interpolation (cont.)**

- **Gouraud shading**
  - Integrates nicely with scan line algorithm:
    \[
    \frac{\Delta I}{\Delta Y} \text{ is constant along polygon edge}
    \]

- **Gouraud versus faceted shading**
What Gouraud Shading Misses

- Gouraud shading can miss **specular highlights** because it interpolates *vertex colors* instead of calculating intensity directly at each point, or interpolating *vertex normals*.
- $N_a$ and $N_b$ would cause no appreciable specular component, whereas $N_c$ would. Interpolating between $I_a$ and $I_b$ misses the highlight that evaluating $I$ at $c$ would catch.

Interpolating normal comes closer the actual normal of the surface being polygonally approximated.
- Reduces temporal “jumping” affect of highlight when rotating sphere during animation.
Phong Shading

- Also called normal vector interpolation
  - interpolate $N$ rather than $I$
  - especially important with specular reflection
  - computationally expensive at each pixel
    - recompute $N$; must normalize, requiring expensive square root
    - recompute $I_\lambda$
  - Bishop and Weimer developed fast approximation using Taylor series expansion (in SIGGRAPH ’86)
- This looks much better and is now done in hardware
- Still, we’ve lost all the neat global effects that we got with recursive ray tracing

http://en.wikipedia.org/wiki/Gourad_shading
Surface Detail

Beautification of Surfaces

- Texture mapping (ubiquitous in hardware)
  - paste photograph or bitmap on a surface to provide detail (e.g. brick pattern, sky with clouds, etc.)
  - think of a texture map as contact paper, but made of stretchable latex
  - map texture/pattern pixel array onto surface to replace (or modify) original color; can still use original intensity to modulate texture

Microsoft Flight Simulator
Texture Mapping (1/5)

Motivation

How do we increase the amount of detail?

- Expensive solution: add more detail to model
  + detail incorporated as a part of object
  - modeling tools aren’t very good for adding detail
  - model takes longer to render
  - model takes up more space in memory
  - complex detail cannot be reused

- Efficient solution: map a texture onto model
  + texture maps can be reused
  + texture maps take up space in memory, but can be shared, and compression and caching techniques can reduce overhead significantly compared to real detail
  + texture mapping can be done quickly (we’ll see how)
  + placement and creation of texture maps can be made intuitive (e.g., tools for adjusting mapping, painting directly onto object)
  - texture maps do not affect the geometry of the object

- What kind of detail goes into these maps?
  - diffuse, ambient and specular colors
  - specular exponents
  - transparency, reflectivity
  - fine detail surface normals (bumps)
  - data to visualize
  - projected lighting and shadows
  - games use “billboards” for distant detail. (sprites are effectively moving billboards)
Texture Mapping (2/5)

Mappings

- A function is a mapping (CS22)
  - functions map values in their subset of a domain into their subset of a co-domain
  - each value in the domain will be mapped to one value in the co-domain
- Can transform one space into another with a function
  - for intersect, we use linear transformations to move points and vectors into the most convenient space
  - map screen-space points to normalized camera-space points
  - then, map normalized camera-space rays into unnormalized world-space rays
  - then, map unnormalized world-space rays into untransformed object-space rays and compute object-space points of intersection
  - finally, map object space points and normals to world space for lighting (why normals in object space?)

Mapping a Texture

- We have points on a surface in object-space
  - the domain
- We want to get values from a texture map
  - the co-domain
- What function(s) should we use?
**Basic Idea**

- Definition: texture mapping is the process of mapping a geometric point to a color in a texture map
- Ultimately want ability to map arbitrary geometry to a concrete pixmap of arbitrary dimension
- We do this in two steps:
  1. map a point on the arbitrary geometry to a point on an abstract unit square representing the concrete pixmap
  2. map a point on abstract unit square to a point on the concrete pixmap of arbitrary dimension
- Second step is easier, so we present it first
- Note: $u,v$ space is not related to film plane UV space (sorry!)

![Diagram showing texture mapping concept](image)
Texture Mapping (4/7)

From unit square to pixmap

- A 2D example: mapping from unit $u,v$ square to texture map (arbitrary pixmap, possibly with alpha values)
- This should be a review from WPF slides

- **Step 1**: transform a point on abstract continuous texture plane to a point in discrete texture map
- **Step 2**: get color at transformed point in texture image
- Above Example:
  
  \[
  (0.0, 0.0) \Rightarrow (0, 0)
  \]
  
  \[
  (1.0, 1.0) \Rightarrow (100, 100)
  \]
  
  \[
  (0.75, 0.45) \Rightarrow (75, 45)
  \]
Texture Mapping (5/5)

- In general, for any point \((u, v)\) on unit plane, corresponding point in image space is:
  \((u \times \text{pix-map width}, v \times \text{pix-map height})\)

- Infinitely many points on unit plane - get sampling errors; \(uv\) plane is continuous version of pixmap, which serves as texture map

- The unit \(uv\) square acts as stretchable rubber sheet to wrap around the texture mapped object.

This scene consists entirely of texture mapped polygons, and therefore has very low scene complexity.

Texture mapping can be done completely in hardware (essentially “free” as far as performance hit)
Texture Mapping Polygons
(how it’s done in hardware)

Interpolating Texture Coordinates

- Real-time interactive applications cannot afford complex model geometry, texture mapping provides a “good-enough” solution
- Pre-calculates texture coordinates for each facet’s vertices during tessellation - store them with vertices, e.g. the cuboid below

Note: vertices will have different tex coordinates on each face

- Interpolate \( uv \) coordinates linearly across triangles as part of Gouraud shading
Texture Mapping Meshes

• Texture Mapping a Triangle:
  – specify texture coordinates for each vertex based on position within the unit square
  – Pass texture coordinate along with vertex, normal etc., while drawing
  – Interpolate across the triangle using barycentric coordinates (see next slide)

• How do you map an entire mesh?
  No standard method.

• Current Research:
  – Angle Based Flattening (A. Sheffer and E. de Sturler)
    • Angles conserved
    • Lots of wasted space
  – Spherical Parameterization (E. Praun and H. Hoppe)
    • No wasted space
    • Can cause distortions
    • Hard to work with
Barycentric Coordinates (1/3)

What are they?

• Think about interpolating along a line
  - You've got a line-segment PQ
  - You know some value, perhaps a color, that varies along the line.
  - You have the values at P and Q, and you want to know the value in the middle. How do you compute it?

\[ C(t) = (1 - t)C_p + tC_q \]

• \( C(t) \) is the color at the location \((1 - t)P + tQ\), but there's an asymmetry in that equation, and it's better to say

\[ C = sC_p + tC_q \quad s + t = 1 \]

where \( C \) is the value at point \( sP + tQ \).

• The numbers \( s \) and \( t \) are barycentric coordinates for the line segment.

• This generalizes to any \( n \)-dimensional simplex
  - A simplex is an \( n \)-dimensional analogue of a triangle (line, triangle, tetrahedron, etc.)

\[ C = sC_p + tC_q + uC_r + \ldots \quad s + t + u + \ldots = 1 \]
Barycentric Coordinates (2/3)

How does this apply to rendering?

- When you shoot a ray at a polyhedral object (a mesh composed of many triangles), it’s convenient to get back the triangle vertices that you intercepted and the barycentric coordinates \((s, t, u)\) of the intersection point \(P\)

\[
P = sV_s + tV_t + uV_u
\]

\[
s + t + u = 1
\]

- You can then use \(s\), \(t\), and \(u\) to interpolate data from the three vertices (color, texture coordinates, normals, etc.)

- A simple way to express Gouraud shading
Texture Mapping in Ray tracing

Ray tracing Textures

• Using ray tracing we obtain a point in object space, 
  \((x, y, z)\)
• The goal is to go from point \((x, y, z)\) to a color
• We know how to map a 2D point \((u, v)\) in the unit square 
  to a color in the pixmap; only need to map 
  \((x, y, z)\) to \((u, v)\)
• 3 easy cases:
  – planes
  – cylinders
  – spheres

• Note: as with ray-object intersection, easiest to calculate 
  the mapping from \((x, y, z)\) to \((u, v)\) from simpler, 
  untransformed object space
  – drawback: texture map is transformed with object into 
    world space, e.g., if a texture-mapped sphere is scaled 
    by 2.0 in \(y\), then the texture map will stretch by a 
    factor of 2.0 in \(y\) as well
    – this implies scaling of the bitmap
    – don’t forget to filter!!!
Methods of Texture Mapping

**Tiling**

- Create a brick wall by applying brick texture to plane

- Produces realistic image, but there are very few bricks in the wall

- Tiling increases the apparent number of bricks
Methods of Texture Mapping

**Stretching**

- Create a sky backdrop by applying a sky image to a plane
- Would look unnatural if tiled
- Stretch over whole plane
Texture Mapping Planes

Tiling

- How to map from a point on an infinite plane to a point on the unit plane?
- Tiling: use decimal portion of $x$ and $z$ coordinates to compute 2D $(u, v)$ coordinates

$$u = x - \text{floor}(x)$$
$$v = z - \text{floor}(z)$$

To map to tiles of size $nxm$:

$$u = (x \% n + (x - \text{floor}(x))) / n$$
$$v = (z \% m + (z - \text{floor}(z))) / m$$
Mapping Circles (1/2)

Texture mapping cylinders and cones

- Imagine a standard cylinder or cone as a stack of circles
  - use position of point on perimeter to determine $u$
  - use height of point in stack to determine $v$
  - map top and bottom caps separately, as onto a plane
- The easy part: calculating $v$
  - height of point in object space, which ranges between $[-.5, .5]$, gets mapped to $v$ range of $[0, 1]$
- Calculating $u$: map points on circular perimeter to $u$ values between 0 and 1; 0 radians is 0, $2\pi$ radians is 1
- Then $u = \theta/2\pi$
- These mappings are arbitrary: any function mapping the angle around the cylinder ($\theta$) to the range 0 to 1 will work
  - let’s look at a mapping that follows the “law of least astonishment”
Texture Mapping Cylinders (2/2)

Mapping a circle

- Want to convert a point P on the perimeter to an angle. Theta is measured clockwise due to the (x, y, z) right-handed coordinate system, but one still expects $u$ to increase as we travel circle counter-clockwise, as shown by colored arrows below.

- Circle radius is not necessarily unit length
  - we need to use arctangent rather than sine or cosine (arctangent only considers the ratio of lengths) of $P_z/P_x$

- Use standard function $\text{atan2}$ because it yields the entire circle (values ranging from $-\pi$ to $\pi$, whereas $\text{atan}$ only returns a half circle)
  - going around the circle in the direction the image is mapped, $\text{atan2}$ returns angles that range from 0 to $-\pi$, then abruptly changes to $+\pi$ and returns to 0

\[ \theta = -\pi/2 \quad u = 0.25 \]
\[ \theta = -\pi \quad u = 0.5 \]
\[ \theta = \pi \quad u = 0.0 \]
\[ \theta = \pi/2 \quad u = 0.75 \]

Note: arrows point in the direction of increasing $u$, not $\theta$
Texture Mapping Spheres

- Imagine a sphere as a stack of circles of varying radii
- \( P \) is a point on the surface of the unit sphere
- Defining \( u \) - use previous mapping for circle
  - if \( v = 0 \) or \( 1 \), there is a singularity and \( u \) should equal some specific value (i.e., \( u = 0.5 \))
  - never really code for these cases because rarely see these values within floating point precision
- Defining \( v \)
  \[
  \varphi = \sin^{-1} \left( \frac{P_y}{r} \right) \quad -\frac{\pi}{2} \leq \varphi < \frac{\pi}{2}, \quad r = \text{radius}
  \]
  \[
  v = \frac{\varphi}{\pi} + 0.5
  \]
Texture Mapping
Complex Geometries (1/5)

How do I texture-map my Möbius strip?

- Texture mapping of simple geometrical primitives was easy. How do I texture map more complicated shapes?
- Say we have a relatively “simple” complex shape: a house
- How should we texture map it?
- We could texture map each polygonal face of the house (we know how to do this already as they are planar).

- This causes discontinuities at edges of polygons. We want smooth mapping without edge discontinuities
- Intuitive approach: reduce to a solved problem. Pretend the house is a sphere for texture-mapping purposes
**Complex Geometries (2/5)**

**Texture mapping a house with a sphere**

- Intuitive approach: place bounding sphere around complicated shape
- Texture map in two stages:
  - Find ray’s object-space intersection point with sphere instead of house.
  - Then easily convert to \( uv \)-coords using spherical mapping

Sphere intersection calculation is extra effort. If we’re at the texture-mapping stage, we’ve already found the intersection point with the complicated shape!

Non-intuitive approach: treat point on the complicated object as a point on a sphere, and project using spherical \( uv \)-mapping
Complex Geometries (3/5)

- Calculate object-space intersection point with geometrical object (house), intersecting against each face.

- Then, use object-space intersection point to calculate $uv$-coords of sphere at that point using spherical projection.

- Note that a sphere has constant radius, but our house doesn’t. Distance from the center of the house to the intersection point (radius) changes with the point’s location.
Complex Geometries (4/5)

- How do we decide what radius to use for the sphere in our \( uv \)-mapper? Intersections with our house happen at different radii.
- Answer: spherical mapping of \( (x,y,z) \) to \( (u,v) \) presented above does not assume the sphere has unit radius.
- Use a sphere with center at the house’s center, and with radius equal to the distance from the center to the current intersection point.

Diagram: Near intersection point = small radius, far intersection point = large radius.
Choosing appropriate functions

- We have chosen to use a spherical uv-mapper to simplify texture mapping. However, any mapping technique may be used.

- Each type of $uv$-projection has its own drawbacks
  - spherical: warping at poles of sphere
  - cylindrical: discontinuities at edges of caps
  - planar: $y$-component of point ignored for $uv$-mapping

- Swapping $uv$-projection techniques allows drawbacks of one technique to be traded for another. Deciding which drawbacks are preferable depends on user input.
Shadow Hacks

Geometry Takes Time

- Projective Shadows (Shadow Mapping)
  - Use orthographic or perspective matrix as CTM to project object (shadow caster) against another object.
  - Render the shadow object as another primitive.
  - Good: fast for single lights and planes
  - Bad: Difficult to project map onto anything but planes. Projecting the shadow object changes its normals, making it difficult to light the shadow object.

- Shadow Volumes
  - Extrude silhouette volume from object (shadow caster) to infinity in the direction opposite the light source
  - Silhouette edge is made up of adjacent polygons
  - Any point within the volume is in shadow
Shadow Hacks (cont.)

Shadow Maps

- Transform camera to each directional light source
- Render the scene from light source PoV (keeping the same far clipping plane), only updating Z buffer
- Read Z buffer into a texture map (the shadow map)
- Apply this texture projectively light’s PoV
- Render the scene from the original eye point
- At each pixel, if distance from light is greater than value in shadow map, it is in shadow
- Major aliasing problem
- Implemented on hardware

Note: By keeping the same far clipping plane, relative distances in Z are preserved
Environment Mapping

Faking Reflections

- Environment (Reflection) Mapping:
  - Simulates reflection with texture map on enclosing sphere or cube serving as crude approximation of the surrounding world
  - Texture map can either be a photograph or generated from a rendered view of the scene from that object’s perspective
  - Great for glossy reflections which don’t have to be very crisp
  - Can add self-reflections by applying technique recursively

- Cube environment map is created by stitching together 6 projective textures.
- The scene is rendered 6 times with the cube sides as different film planes and the results are stored as environment textures.
- Ray from eye is reflected about the surface normal and intersected with environment map to get texture coordinates.
- Sphere environment is similar – map of photo or prerender to a sphere. Intersect with sphere instead of cube to get reflection.
Surface Detail

Brief Overview

Different ways to approximate the lighting and shading effects of desired geometry

Ideal Geometry

Adjust geometric normals with bump map

Approximate geometry at render time with displacement map

Replace normals of simpler geometry with normal map
Surface Detail—Bump Mapping

- Texture mapping a rough surface onto an object doesn’t look right - illumination is all wrong
- Blinn’s hack: use an array of values to perturb surface normals (calculate gradient and add it to the normal)
- Evaluate illumination equation with perturbed normals
- Effect is convincing across surface, but silhouette edges still appear unperturbed
- Consider an orange:

Not actual bump map for orange, just an abstraction

Bogus curve

Spherical Surface

Red line shows standard diffuse hemisphere, blue line shows theoretical effect of bump map

nVidia

Intensity
The tin foil on this Hershey's Kiss seems a bit too smooth…

Bump Mapping to the Rescue – Tin foil with no increase in model complexity!

Bump Mapping Example
Normal Mapping

- Recently bump mapping is being replaced by normal mapping for many applications
  - similar idea with much better results
  - normal mapping handles drastic variation in normals better than bump mapping

- *Bump mapping* uses a single-channel (grayscale) map to *perturb* existing normals on the surface;
  - treat grayscale map as height field
  - perturb the surface normals by the gradient of the map
  - lighting is calculated using the normal

- *Normal mapping* uses a multichannel (rgb) map to completely *replace* the existing normals. RGB values of each pixel corresponding to the x,y,z components of the normal vector
  - x,y,z components of the mapped normals are usually in object-space but other spaces are sometimes used
  - level of detail of output renders is limited by resolution of normal maps instead of by the number of polygons in rendered meshes!
  - runtime surface normal determination is trivial—it’s just a look-up—making this technique particularly useful for real-time applications

- Limitations:
  - silhouettes do not reflect added detail
  - though runtime is easy, initial creation of normal maps is not straight-forward (this is a big deal -- more on slide 46)
Normal Mapping 2

- Normal maps are most useful for adding detail to simple meshes, allowing low-res polygon models to *appear* to be much higher-resolution.

![Image courtesy of www.anticz.com](https://www.anticz.com)

- This approach can be extended to almost completely alter the perceived geometry of a model.

  - **Left half** – Anisotropic reflections
  - **Right half** – Normal map is used to fake additional geometry

Environment map is used to provide faked reflections.

render showing actual, simple underlying geometry

when a normal map is applied, there appears to be much more to the model than there is
Normal Map Creation (1/2)

- As mentioned, the creation of meaningful normal maps is not simple
- Unlike bump maps, normal maps can not simply be painted by hand — color in a normal map is constantly varied and r,g,b values are spatially meaningful

Normal map for a human face. Try paining that in Photoshop!

Note how red, green and blue correspond to the direction a normal at each pixel would face

Normal map for brick street

So how are normal maps created?
Normal Map Creation (2/2)

- Normal maps must be generated by specialized software, such as Pixologic’s Zbrush (check out www.pixologic.com).
  
a high resolution model is created by the artist and its normals are used to generate maps for lower-res versions of the model.

- Low-res mesh (~5,000 polygons)
- High-resolution model (2million polygons!)—this is the source of the normal map.

So many polygons that you can’t even see the edges of the triangles!

Final render with normal map applied to low-res mesh (and lighting and color maps). This is only 5,000 triangles!

Images courtesy of www.unrealtechnology.com
Surface Detail

Other Approaches

- Displacement Map: the actual geometric position of points over the surface are *displaced* along the surface normal according to the values in the texture.


The actual map: intensity values used to determine height of geometry
Surface Detail

Other Approaches

- Parallax Mapping: displaces texture coordinates of a surface point by a function of the view angle relative to the surface normal and the value on the height map at that point.

- Steep Parallax Mapping: similar scheme to produce parallax, self-occlusion, and self-shadows but more efficient and reduces swim effect for high frequency maps (Morgan McGuire Ph.D. ’06 and Max McGuire).

![Texture Mapped](image1) ![Parallax Mapped](image2)
Surface Detail

Texture Mapped

Normal Mapped

Increased depth in rock wall

Parallax Mapped

Increased depth in whole texture, but has ‘swim’ effect

Steep Parallax Mapped

Increased depth in whole texture and self shadowing