CENG 477
Introduction to Computer Graphics
Texture Mapping
Until Now

• We assumed that objects have **fixed R, G, B reflectance** values
• Surface **orientation** and light **direction** determined the shading of objects
• Our best image so far is something like:
Texture Mapping

- **Goal:** Increase visual realism by using images to simulate reflectance characteristics of objects.
- A cheap and effective way to **spatially vary** surface reflectance.
Texture Mapping

- Image (texture) sizes and object sizes may vary
- We need a uniform way so that any object can be mapped by any texture
Texture Mapping

- **Step 1:** Associate a \((u, v)\) coordinate system with the texture image where \((u, v) \in [0,1] \times [0,1]\)
Texture Mapping

- **Step 2:** Parameterize the surface to be texture mapped using two coordinates. For instance, a sphere:

Assuming that the center is at (0, 0, 0):

\[
\begin{align*}
x &= r \sin \Theta \cos \phi \\
y &= r \cos \Theta \\
z &= r \sin \Theta \sin \phi
\end{align*}
\]

\[
\Theta = \arccos\left(\frac{y}{r}\right)
\]

\[
\phi = \arctan\left(\frac{z}{x}\right)
\]

In practice, \(\phi = \text{atan2}(z, x)\)

\((\Theta, \phi) \in [0, \pi] \times [-\pi, \pi]\)
Texture Mapping

- Assume you want to wrap the image such that:

\[
u = \frac{-\phi + \pi}{2\pi}
\]
\[
v = \frac{\Theta}{\pi}
\]
Texture Mapping

• **Step 3:** Compute a \((u, v)\) value for every surface point. For instance, a sphere:

\[
\begin{align*}
  u &= (-\phi + \pi) / (2\pi) \\
  v &= \Theta / \pi
\end{align*}
\]

• **Step 4:** Find the texture image coordinate \((i, j)\) at the given \((u, v)\) coordinate:

\[
\begin{align*}
  i &= u.n_x \\
  j &= v.n_y
\end{align*}
\]

Note that \(i, j\) can be fractional!

\(n_x = \) texture image width
\(n_y = \) texture image height
Texture Mapping

• **Step 5:** Choose the texel color using a suitable interpolation strategy
  
  – **Nearest Neighbor:** fetch texel at nearest coordinate (faster)
    \[
    \text{Color}(x, y, z) = \text{fetch}(\text{round}(i, j))
    \]
  
  – **Bilinear Interpolation:** Average four closest neighbors (smoother):
    \[
    p = \text{floor}(i) \\
    q = \text{floor}(j) \\
    dx = i - p \\
    dy = j - q \\
    \text{Color}(x, y, z) = \text{fetch}(p, q).(1 - dx).(1 - dy) + \\
    \text{fetch}(p+1, q).(dx).(1 - dy) + \\
    \text{fetch}(p, q+1).(1 - dx).(dy) + \\
    \text{fetch}(p+1, q+1).(dx).(dy)
    \]
NN vs Bilinear Interpolation

Nearest-neighbor
NN vs Bilinear Interpolation
NN vs Bilinear Interpolation

Nearest-neighbor

Bilinear
Texture Color

• Once we have the texture color, what to do with it?
• Several options possible: replace shading color, blend with shading color, replace $k_d$ component, etc.
Texture Color

• For instance, to replace the $k_d$ component:

\[
L_o^d(x, wo) = k_d \cos \theta' L_i(x, w_i)
\]

\[
\cos \theta' = \max(0, w_i \cdot n)
\]

where

\[
k_d = \text{textureColor} / 255
\]
Texture Color

• Replacing $k_d$ typically improves realism over replacing the entire diffuse color:

Diffuse color replaced

$k_d$ component replaced
Animation

- Animation, shading, and texture mapping enhances realism:
Texture Mapping Triangles

• First, we must associate (u, v) coordinates for each vertex
• The (u, v) coordinates inside the triangle can be found using the barycentric coordinates
• Recall that the position of point p at barycentric coordinates \((\beta, \gamma)\) is equal to:

\[
p(\beta, \gamma) = a + \beta(b - a) + \gamma(c - a)
\]

• Texture coordinates can be interpolated similarly:

\[
u(\beta, \gamma) = u_a + \beta(u_b - u_a) + \gamma(u_c - u_a)
\]
\[
v(\beta, \gamma) = v_a + \beta(v_b - v_a) + \gamma(v_c - v_a)
\]
Texture Mapping Triangles

- Assume we want the texture map a rectangle made of two triangles:
  - $v_3, v_1, v_2$
  - $v_1, v_3, v_0$
Texture Mapping Triangles

- We associate a texture coordinate with each vertex of each triangle:
  - $v_3, v_1, v_2$: $(1, 0), (0, 1), (1, 1)$
  - $v_1, v_3, v_0$: $(0, 1), (1, 0), (0, 0)$
Texture Mapping Triangles

- This way, the texture image gets stretched to match the triangles’ positions
  - $v_3, v_1, v_2$: $(1, 0), (0, 1), (1, 1)$
  - $v_1, v_3, v_0$: $(0, 1), (1, 0), (0, 0)$
Combined Result
Tiling

• For large surfaces, stretching may excessively distort the texture image making it unrealistic
Tiling

- Tiling is the process of repeating a texture instead of stretching
- Multiple copies of the texture fill the surface
- Tiling may also look unrealistic if repetition is clear
Tiling

- To support tiling, tex. coords. are not limited to [0, 1] range

\[ u = u_{\text{barycentric}} - \text{floor}(u_{\text{barycentric}}) \]
\[ v = v_{\text{barycentric}} - \text{floor}(v_{\text{barycentric}}) \]
Procedural Textures

• The previous approach requires us to use an image as the texture to be mapped
• We can also achieve spatial variation without using any texture image at all
• **Solution:**
  – Generate textures procedurally for each surface point
• **Problem:**
  – How to make them look natural?
Procedural Textures

- Periodic patterns are easy to generate but they do not look natural (e.g. evaluate a sine function at every point):
Procedural Textures

• We need a function that returns a random value for each (x, y, z) point in space
• However, too random is not good as it will appear as noise:
Procedural Textures

• There are various techniques such as **Perlin noise** that allow on-the-fly computation of controlled random patterns
Procedural Textures

- Image and procedural textures can be used together:
CENG 477
Introduction to Computer Graphics

Data Structures for Graphics
Until Now

- We rendered virtual objects
  - Ray tracing
  - Ray are easy: \( \mathbf{r}(t) = \mathbf{o} + \mathbf{d}t \)
  - Mathematical objects also easy: \( x^2 + y^2 + z^2 = R^2 \)
  - How about arbitrary objects embedded in 2D/3D scenes?

- Today we will learn about
  - explicitly representing those objects
    - triangle meshes
  - organizing them for efficiency
    - spatial structures
Triangle Meshes

• The most popular way of representing geometry embedded in 2D or 3D
• A triangle mesh is a piecewise linear surface representation
Triangle Meshes

- Smoothness may be achieved by using a *larger* number of *smaller* pieces
Manifolds

- Our meshes will be manifolds:
  - Each edge is shared by at most two faces
  - Faces containing a vertex form a closed or open fan:
    - Closed fan
    - Open fan

https://www.cs.mtu.edu/~shene
Non-manifolds

• The following meshes are non-manifolds:

https://www.cs.mtu.edu/~shene
Triangle Meshes

• A triangle mesh is an **undirected graph** with triangle faces
  – It has **vertices**, **edges**, and **faces**
  – The **degree** or **valance** of a vertex is the # of incident edges
  – A mesh is called **k-regular** if the degree of all vertices are k

\[ G = \langle V, E, F \rangle \]
\[ V = \{A, B, C, ..., K\} \]
\[ E = \{(A, B), (A, E), \ldots\} \]
\[ F = \{(A, E, B), (B, E, F), \ldots\} \]

\[ \text{deg}(A) = 4 \]
\[ \text{deg}(E) = 5 \]
Graph Planarity

• A graph is **planar** if it can be drawn in the plane such that no two edges cross each other (except at the vertices):

• Planarize:
Mesh Statistics

- Almost every mesh is a planar graph
- For such graphs Euler formula holds:

\[ \#V - \#E + \#F = 2 \]

\( V = 8 \)
\( E = 12 \)
\( F = 6 \)
\[ \chi = 8 + 6 - 12 = 2 \]

\( V = 3890 \)
\( E = 11664 \)
\( F = 7776 \)
\[ \chi = 2 \]
Mesh Statistics

• Based on Euler’s formula:

\[ \#F \sim 2V \]
\[ \#E \sim 3V \]
\[ \text{AVD} \sim 6 \]

Average vertex degree

\[ V = 3890 \]
\[ E = 11664 \]
\[ F = 7776 \]
Mesh Structures

• How to store geometry & connectivity of a mesh
  
  3D vertex coordinates    Vertex adjacency

• Attributes are also stored: normals, colors, texture coordinates, labels, …

• Efficient algorithms on meshes to get:
  – All vertices/edges of a face
  – All incident vertices/edges/faces of a vertex
Face-based Structures

- Face-Set Data Structure (.stl format)
  - Aka polygon soup as there is no connectivity information

  - Vertices and associated data replicated 😞
  - Using 32-bit single precision numbers to represent vertex coords, we need
    32/8 (bytes) * 3 (x-y-z coords) * 3 (# vertices) = 36 bytes per face
  - By Euler formula (F ~ 2V), each vertex consumes 72 bytes on average
Face-based Structures

- Indexed Face-Set Data Structure (.obj, .off, .ply, our XML format)
  - Aka shared-vertex data structure

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$ $y_1$ $z_1$</td>
<td>$i_{11}$ $i_{12}$ $i_{13}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$x_v$ $y_v$ $z_v$</td>
<td>...</td>
</tr>
</tbody>
</table>

- No vertex replication 😊
- We need 4 (bytes) * 3 (# indices) = 12 bytes per face (24 bytes per vertex)
- We also need 4 (bytes) * 3 (x-y-z coords) = 12 bytes per vertex
- Total = 36 bytes per vertex, half of Face-Set structure 😊
Face-based Structures

- Regardless of the structure, triangle vertices must be stored in a consistent order
  - Mostly counterclockwise (CCW)
Data Structures for Graphics

• With large number of triangles, rendering tasks tend to take too long if data is not properly organized
• Many data structures exist
  – Quadtree
  – Octree
  – BSP tree
  – k-D tree
  – BVH
Quadtree and Octree

- **Quadtrees** are used in 2D and **octrees** in 3D
Binary Space Partitioning (BSP)

- Divide the space with freely oriented lines (2D) and planes (3D)
Binary Space Partitioning (BSP)

- BSP trees are view-independent (no need to reconstruct if the viewer moves)
- BSP trees can be used to draw a scene in back-to-front order with respect to the viewer

Corresponding BSP tree
Binary Space Partitioning (BSP)

- Assume camera is at point C
- Always traverse the half-space first that does not contain C
- This guarantees back-to-front traversal w.r.t. the camera

Corresponding BSP tree

1 2 3 4 5

C
k-D Tree

- In a k-D tree, a scene is recursively divided into 2 convex sets by \textit{axis-aligned} hyperplanes: a special case of BSP tree
k-D Tree

- k-D tree is a binary tree
- Each node is a k-dimensional point
- Splitting planes are *alternately* chosen between dimensions:
  - First x, then y, then z, back to x and so on (axis = (axis + 1) % 3)
- It will be a **balanced tree** if the median element is chosen at each split
  - $\log_2(n)$ depth in that case
k-D Tree

• Basic algorithm:

```plaintext
function kdtree (list of points pointList, int depth)
{
    // Select axis based on depth so that axis cycles through all valid values
    var int axis := depth mod k;

    // Sort point list and choose median as pivot element
    select median by axis from pointList;

    // Create node and construct subtrees
    var tree_node node;
    node.location := median;
    node.leftChild := kdtree(points in pointList before median, depth+1);
    node.rightChild := kdtree(points in pointList after median, depth+1);
    return node;
}
```

wikipedia.com
k-D Tree

- A 2-D example (k = 2):
Bounding Volume Hierarchy (BVH)

- BVH is similar to a k-D tree but each node contains a **bounding volume** (aka **bounding box**)
- Rays missing the bounding boxes are not intersected with the actual objects
Bounding Volume Hierarchy (BVH)

- Bounding boxes can be made hierarchical
- Overlap between bounding boxes are possible
Bounding Volume Hierarchy (BVH)

- Bounding boxes can be made hierarchical
- Overlap between bounding boxes are possible
Bounding Volume Hierarchy (BVH)

- BBs go all the way down to individual primitives (triangles)
Bounding Volume Hierarchy (BVH)

- This is represented as a binary tree where only the leaf BBs contain the actual objects
Bounding Volume Hierarchy (BVH)

A ray missing this box cannot intersect any objects it bounds.
Bounding Volume Hierarchy (BVH)

A ray missing this box cannot intersect any objects it bounds.
Bounding Volume Hierarchy (BVH)

- Note that a ray may intersect with multiple bounding boxes
Bounding Volume Hierarchy (BVH)

- Intersecting boxes are shown in bold
- Note that this ray does not intersect with any real object
Bounding Volume Hierarchy (BVH)

• Basic traversal algorithm:

```cpp
bool BVH::intersect(const Ray& r, HitRecord& rec) const
{
    if (bbox.boxIntersect(r) == false)
    {
        // This ray entirely misses this bounding box
        return false;
    }

    HitRecord rec1, rec2;
    rec.t = INFINITY;

    bool hitLeft = left->intersect(r, rec1);
    bool hitRight = right->intersect(r, rec2);

    if (hitLeft)
    {
        rec = rec1;
    }

    if (hitRight)
    {
        rec = rec2.t < rec.t ? rec2 : rec;
    }

    return (hitLeft || hitRight);
}
```
Bounding Volume Hierarchy (BVH)

- BVH affords significant speed-up in ray tracing:
  - This scene contains 31584 objects (all triangles except 2 spheres)

No-BVH
Total rendering time: 12 mins 33 secs
Test conducted on
Intel Core i7 960 CPU
with 8 cores at 3.2GHz

With-BVH
Total rendering time: 1.2 secs
Test conducted on
Intel Core i7 960 CPU
with 8 cores at 3.2GHz