CENG 477
Introduction to Computer Graphics
Data Structures for Graphics
Until Now

• We rendered virtual objects
  – Ray tracing
  – Ray are easy: \( \mathbf{r}(t) = \mathbf{o} + d\mathbf{t} \)
  – 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
  - And faces containing a vertex form a closed or open fan:

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

• The following meshes are non-manifolds:

https://www.cs.mtu.edu/~shene
• 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

\[ G = \langle V, E, F \rangle \]
\[ V = \{A, B, C, ..., K\} \]
\[ E = \{(A, B), (A, E), ...\} \]
\[ F = \{(A, E, B), (B, E, F), ...\} \]
\[ \text{deg}(A) = 4 \]
\[ \text{deg}(E) = 5 \]
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:

http://www.personal.kent.edu/~rmuhamma
Mesh Statistics

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

$$\#V - \#E + \#F = 2$$

Example:

- $V = 8$
- $E = 12$
- $F = 6$
  
  $$\chi = 8 + 6 - 12 = 2$$

- $V = 3890$
- $E = 11664$
- $F = 7776$
  
  $$\chi = 2$$
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 ($F \sim 2V$), 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>$i_1$ $i_2$ $i_3$</td>
</tr>
<tr>
<td>\ldots</td>
<td>\ldots</td>
</tr>
<tr>
<td>$x_v$ $y_v$ $z_v$</td>
<td>\ldots</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

![Diagram showing a binary space partitioning tree and its corresponding partitioned space]
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
Binary Space Partitioning (BSP)

• BSP trees do not need to be recreated if the camera moves
  – Their traversal depends on the camera position

• How to create a BSP tree?

• **Step 1:** Select a polygon and create a plane aligned with it
  – Put that polygon to your root

• **Step 2:** Separate the other polygons into two sets
  – One above the plane and other below the plane
  – If the plane intersects some polygons, split them and place them to their corresponding sets

• **Step 3:** Recursively apply Step 1 and 2 until you reach a desired stopping condition
Binary Space Partitioning (BSP)

Sample Input

Step 1

Step 2

Step 3

See: https://en.wikipedia.org/wiki/Binary_space_partitioning
Binary Space Partitioning (BSP)

Step 4

Step 5

Step 6

Step 7

See: https://en.wikipedia.org/wiki/Binary_space_partitioning
Binary Space Partitioning (BSP)

See: https://en.wikipedia.org/wiki/Binary_space_partitioning
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

wikipedia.com
k-D Tree

• k-D tree is a binary tree
• Each node is a k-dimensional point
• Splitting planes are **alternatingly** 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:

```javascript
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;
}
```
k-D Tree

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

• While k-D trees partition space into disjoint regions, a BVH partitions objects into disjoint polygons
  – k-D tree: space subdivision
  – BVH: object subdivision
• Objects are contained within bounding boxes (aka bounding volumes)
Bounding Volume Hierarchy (BVH)

- 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 (whereas in a k-D tree overlap between regions is not possible)
Bounding Volume Hierarchy (BVH)

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

- BBs may 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.rayIntersect(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)
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
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