Data Structures & Algorithms for Geometry

- Agenda:
  - Quiz #3
  - BSP trees, part 2:
    - Traversing / using BSP trees
    - Advanced split-plane selection
    - Optimization
  - Assignment #3 due
  - Begin assignment #4

Intersecting a Point w/Solid BSP

- Operates as you would expect:
  - If the point is in the positive half-space, traverse the positive child.
    - If the child is a leaf, the point is outside the solid.
  - If the point is in the negative half-space, traverse the negative child.
    - If the child is a leaf, the point is inside the solid.

Intersecting a Point w/Solid BSP (cont.)

```
int BSP_node::test_point(const point &p) const
{
    BSP_node *n = this;
    int visit_child = 0;

    while (!n->is_leaf()) {
        const plane split = n->get_plane();
        const float dist = plane.n.dot3(p) + plane.d;

       visit_child = (dist <= EPSILON);
        n = n->child[visit_child];
    }

    return (visit_child == 0)
        ? POINT_INSIDE : POINT_OUTSIDE;
}
```

Intersecting a Point w/Solid BSP (cont.)

- What if we need to know when the point is on the boundary?

Intersecting a Point w/Solid BSP (cont.)

- What if we need to know when the point is on the boundary?
  - If the point is within ε of the plane, traverse both subtrees.
  - If both subtrees produce the same result, that is the answer.
  - If each subtree produces a different result, the point is on the boundary.

Intersecting a Ray w/Solid BSP

- Obvious answer:
  - Clip the ray by the split-plane.
  - Send each non-empty piece down the corresponding subtree.
  - The piece of the ray closest to the ray's origin that ends in solid space is the first intersection.

Intersecting a Ray w/Solid BSP

- Obvious answer:
  - Clip the ray by the split-plane.
  - Send each non-empty piece down the corresponding subtree.
  - The piece of the ray closest to the ray's origin that ends in solid space is the first intersection.
- What's the problem with this approach?

Intersecting a Ray w/Solid BSP

- Obvious answer:
  - Clip the ray by the split-plane.
  - Send each non-empty piece down the corresponding subtree.
  - The piece of the ray closest to the ray's origin that ends in solid space is the first intersection.
- What's the problem with this approach?
  - Lots of repeated clipping of the same ray results in lots of accumulated floating-point error.

Intersecting a Ray w/Solid BSP (cont.)

- Use the parametric form of the ray:

$$R(t) = P_0 + t \times d$$

- Intersection routine takes t_{min}, t_{max}, P_0, and d as parameters.
  - Calculate $t_{intersect}$.
  - Repeat using [t_{min}, $t_{intersect}$] and [$t_{intersect}$, t_{max}].
    - If $t_{min} = t_{intersect}$ or $t_{intersect} = t_{max}$, then that portion does not contain part of the ray.

Intersecting a Polytope w/Solid BSP

- Obvious answer:
  - Test each face of the polytope against BSP.
  - The face test proceeds like face insertion, but the tree is not modified.

Intersecting a Polytope w/Solid BSP

- Obvious answer:
  - Test each face of the polytope against BSP.
  - The face test proceeds like face insertion, but the tree is not modified.
- What's the problem with this method?

Intersecting a Polytope w/Solid BSP

- Obvious answer:
  - Test each face of the polytope against BSP.
  - The face test proceeds like face insertion, but the tree is not modified.
- What's the problem with this method?
  - **S-L-O-W**
  - Misses intersections of a disjoint solid completely contained within polytope

Intersecting a Polytope w/Solid BSP

- Obvious answer:
  - Test each face of the polytope against BSP.
  - The face test proceeds like face insertion, but the tree is not modified.
- What's the problem with this method?
  - **S-L-O-W**
  - Misses intersections of a disjoint solid completely contained within polytope
    - Can solve this by converting polytop to a BSP tree and calculating the union of the two trees.

Merging BSP Trees

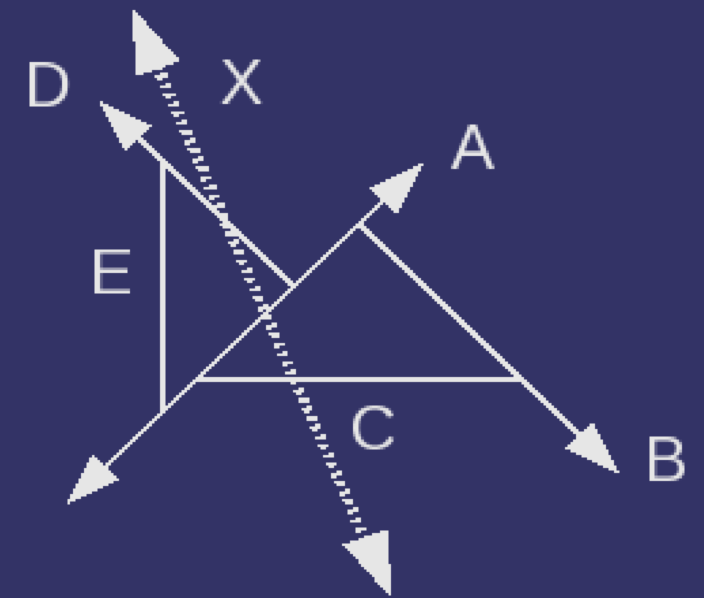
- *Merging* two BSP trees can be used to perform numerous operations on the trees:
  - Union
  - Intersection
  - Difference
  - etc.

Merging BSP Trees (cont.)

- Conceptually very simple recursive operation:
  - If T_1 or T_2 is a leaf, merge the leaf into the tree.
    - Insert each of the polygons in the leaf in the other tree.
  - Otherwise, partition T_2 by T_1's split-plane
  - Merge the portion of T_2 in T_1's negative half-space to T_1's negative child
  - Merge the portion of T_2 in T_1's positive half-space to T_1's positive child
- Fundamental operation is splitting a tree.

Splitting a BSP Tree

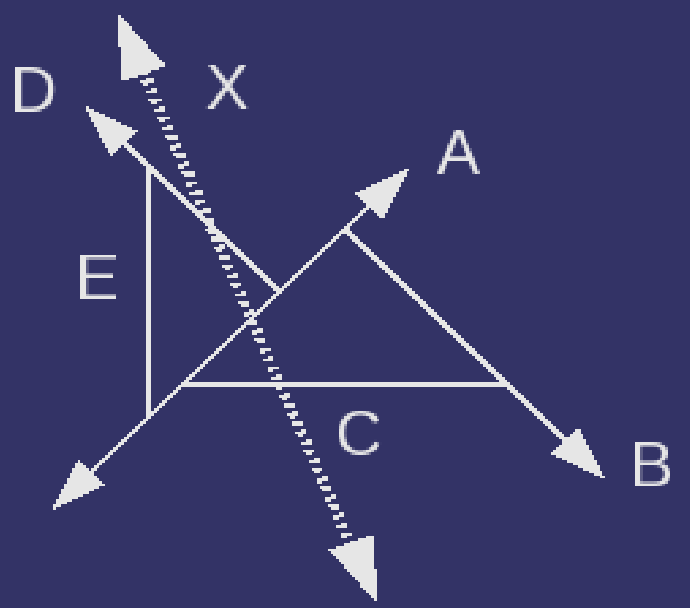
- Want to split a BSP tree, *T*, by a split-plane, *X*.
  - At any time *X* will have a set of zero or more edges defined in the plane.
  - Each edge represents a previous intersection with a plane of *T*.
    - Initially *X* is an infinite plane.
    - Intersecting with the root, *A*, of *X* splits in half.
    - Intersecting with the positive node from *A*, *D*, splits it again.



- Sound familiar?

Splitting a BSP Tree (cont.)

- Track a (*k*-1)d BSP tree for *X.*
  - This enables determining that subspaces of *T*'s nodes can be discarded.
  - Only one subspace of *B* needs to be considered.
- *Real* work begins when a leaf of *T* is reached.
  - *X* becomes a new split-plane, and contents of the leaf are re-split by *X*.
    - Contents of *E* and *C* are divided by live portions of *X*.



CSG Operation Using the Merge

- After merge complete, each leaf is tagged as having come from either T_1 or T_2 or both.
- Perform appropriate logical operation on the leaves.
  - $T_1 \wedge T_2$: delete leaf nodes that come from only one of the original trees.
  - $T_1 - T_2$: delete leaf nodes from T_2 or from both.
  - $T_1 \neq T_2$: delete leaf nodes from both.
  - etc.

References

http://www.mcs.csuhayward.edu/~tebo/papers/siggraph90.pdf

Smarter Split-plane Selection

- Level 0 heuristics:
  - Pick random split-plane, hope for the best.
- Level 1 heuristics:
  - Least-crossed – pick plane that causes least splits
  - Most-crossed – pick plane most likely to repeatedly split later
  - Balancing cuts – pick plane that evenly divides number of polygons to child nodes

Level 2: Conflict Minimization

- Pick the split-plane that produces the least total splits at this iteration and the next.
- For each potential split-plane, *P*:
  - Count the number of planes in the positive space of *P* that intersect planes in the negative space of *P* (and vice-versa).
  - Subtract a weighing of the number of planes split by *P.*
- Pick the plane with the highest score.

Level 3: Conflict Neutralization

- For each polygon, track 3 lists:
  - Depth 1: Set of planes that split it.
  - Depth 2: Set of planes that block each of the splitters.
  - Depth 3: Set of planes that block each of the blockers from blocking each of the splitters.
- Plane's score: -1 each time it appears at depth 1 or 3, +1 each time it appears at depth 2.
  - Pick the plane with the highest score.

References

http://mysite.wanadoo-members.co.uk/dradamjames/PHD/download.html#CN

- This is the Conflict Neutralization paper.

http://www.cs.unc.edu/~fuchs/publications/VisSurfaceGeneration80.pdf

- This is the Conflict Minimization paper.

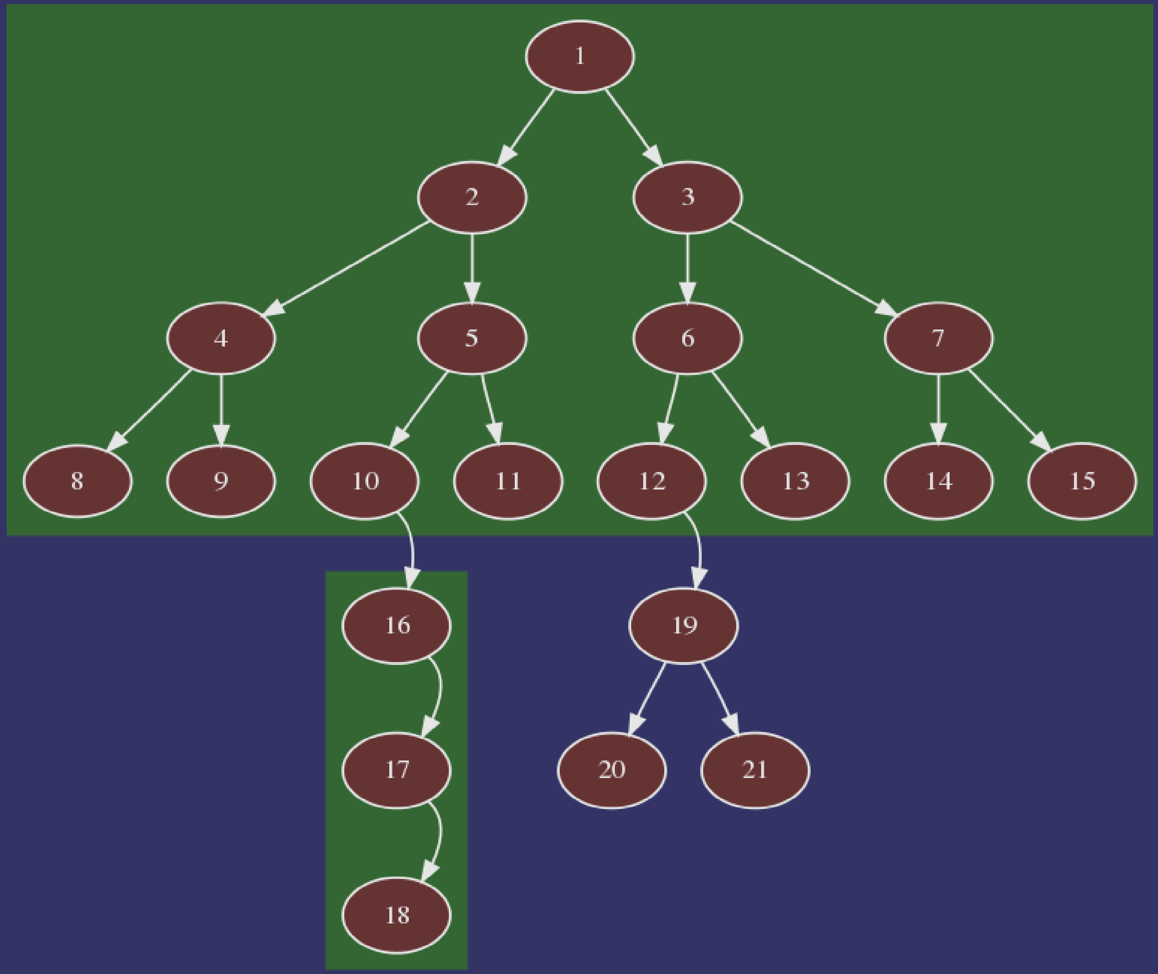
Memory Usage

- Obvious node structure is 28 bytes:

```
struct bsp_node {
    plane split_plane;
    bool leaf;
    union {
        bsp_node *children[2];
        struct {
            polygon **p;
            unsigned num_polygons;
        } brushes;
    } data;
};
```

Tree Structure Observations

- Common tree structures:
  - Top portion of tree will typically be complete.
  - May be sections of linearized split planes.
- How do we take advantage of this?



Compacted Complete Subtree

- Represent the complete subtree by the split-planes of the inner nodes and the pointers to the outgoing leaves.

```
struct bsp_complete_node {
    unsigned depth;   /* Depth of subtree. */
    plane    *split;  /* (2^depth)-1 split-planes */
    bsp_node **children; /* 2^depth children */
};
```

- Reduction in storage: 28n bytes → 12+20n bytes
  - Saves 20% on 15-node subtree
  - Saves 28% on 31-node subtree!

Fused Linear Nodes

- Pack all linear nodes into a single node.

```
struct bsp_linear_node {
    plane *split_planes;
    unsigned char num_split_planes;
    bool leaf;
    union { /* ... */ } data;
};
```

- Reduction in storage: 28n bytes → 16+16n bytes
  - Saves 24% on 3-node group

Special-case Fused Nodes

- Handle the case of 3 linear nodes apart from the general case.
  - Depending on the data, may not need general case

```
struct bsp_linear3_node {
    plane split_planes[3];
    bool leaf;
    union { /* ... */ } data;
};
```

- Reduction in storage: 84 bytes → 60 bytes
  - Saves 29% on 3-node group

Subtree Nodes

- Represent small, fixed size subtree in one node

```
struct bsp_subtree_node {
    plane split_panes[3];
    bool leaf[2];
    union { /* ... */ } data[2];
};
```

- Only need child data for the two leaves.
- Reduction in storage: 84 bytes → 68 bytes
  - Saves 20% on 3-node group

References

http://www.cgg.cvut.cz/~havran/ARTICLES/compugr97.pdf

Next week...

- No class next Saturday (11/24)!
  - Next meeting is 12/1.
- Optimization
  - Measuring code performance
  - Memory hierarchy in real computers
    - Tree node packing to optimize for CPU caches
    - Structure of arrays vs. array of structures
  - Avoiding re-calculations
- Assignment #4 due

Legal Statement

- This work represents the view of the authors and does not necessarily represent the view of IBM or the Art Institute of Portland.
- OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.
- Khronos and OpenGL ES are trademarks of the Khronos Group.
- Other company, product, and service names may be trademarks or service marks of others.