Computer Graphics - Week 10

Bengt-Olaf Schneider
IBM T.J. Watson Research Center

Questions about Last Week ?

Overview of Week 10

- **Solid Modeling**
  - Sweeps
  - Boundary Representations
  - Spatial Partitioning
  - Constructive Solid Geometry (CSG)
- **Global Illumination (Part 1)**
  - Ray-tracing
  - Advanced lighting models
  - Spatial data structures

Modeling

- **Modeling is the process of generating the description of a 2D or 3D shape**
  - Approximation of a real shape, e.g. 3D capture or visualization
  - Creation or design of a new shape, e.g. CAD or DCC
- **The shape description has to satisfy several, often conflicting demands**
  - Accuracy, i.e. match between model and real shape
  - Compactness
  - Consistency, e.g. no "impossible" shapes
  - Robustness, e.g. limited errors due to floating-point calculations
  - Support of editing and queries, e.g. to support interactive manipulation

Desirable Properties of Modeling Data Structures

- **Expressive Power**
  - What kind of objects can be represented using the data structure ?
  - How accurately can objects be represented ?
- **Validity**
  - Are all values of the data structure representing valid objects ?
- **Uniqueness**
  - Does every valid object have exactly one representation ?
- **Conciseness**
  - How large is the representation of interesting objects ?
- **Closure of operations**
  - Do operations on the data structure always generate valid objects ?
- **Applicability and Computational Ease**
  - What kind of algorithms are supported by the data structure ?

Solid Modeling

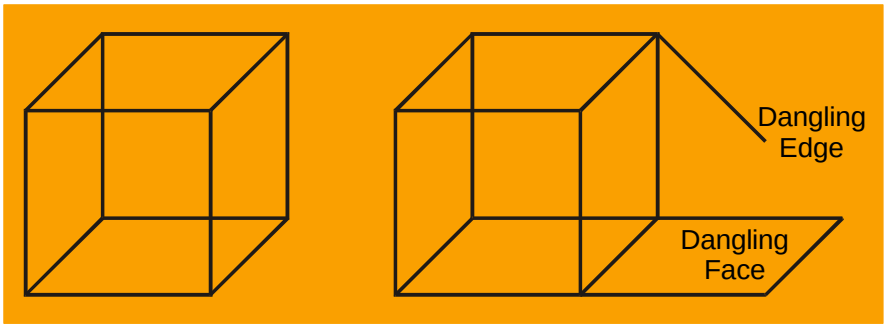
Solid Modeling

- Representation of solid models, i.e. models filled with material
- Manipulation of solids, e.g. combination or interaction of solid models

- Used in various applications, e.g.
  - Manufacturing and CAD/CAM
  - Interference detection, e.g. robot path planning
  - Physical properties, e.g. a part's mass and center of gravity
  - Automatic instructions for machining of parts

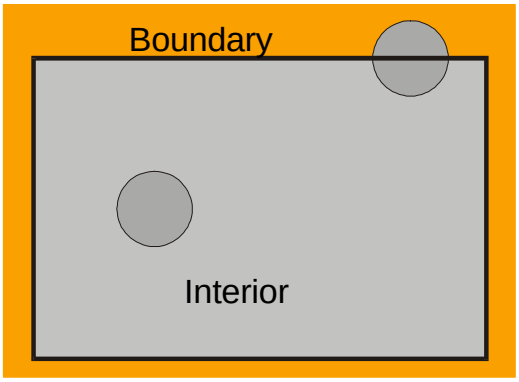
Solids

- The representations we have discussed so far, allow to model solids but are not always describing solids
- The vertex-edge-face descriptions can also describe non-solid objects



Taking apart a Solid (1)

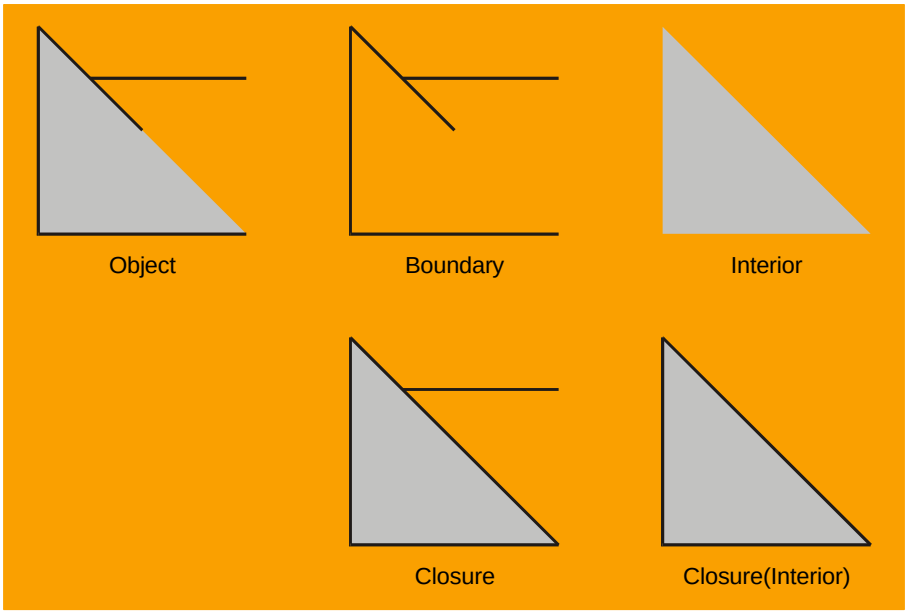
- **Interior**
  - Points that are entirely interior of the object
  - Points having a non-zero neighborhood with only interior points
- **Boundary**
  - Points with zero distance from both the object and its complement
  - There is no non-zero neighborhood that contains only interior points



Taking apart a Solid (2)

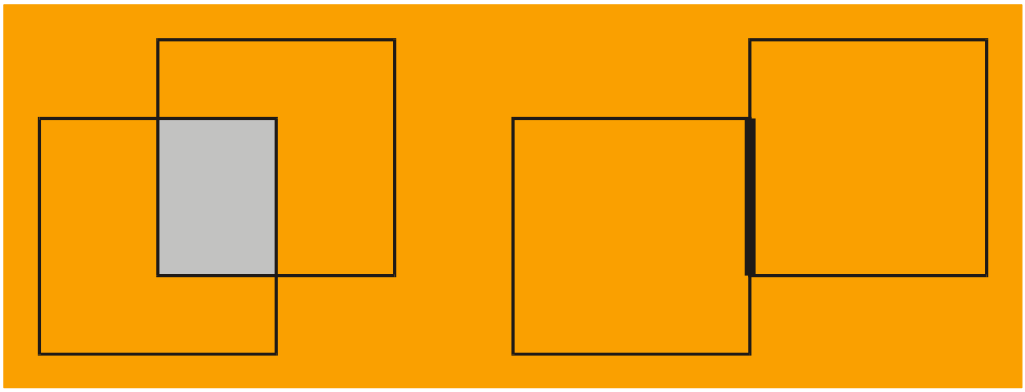
- **Open Set**
  - A set with only its interior point and none of its boundary points
- **Closed Set**
  - A set with all its interior and boundary points
- **Closure**
  - Union of a set with its boundary points
- **Regularization**
  - Closure of the interior points

Taking apart a Solid (3)



Combining Solids (1)

- Construction of more complicated solids can be accomplished through Boolean set operations
- Standard set operations can create non-solid objects !



Combining Solids (2)

- **Instead of standard Boolean operations, we use regularized Boolean operations:**

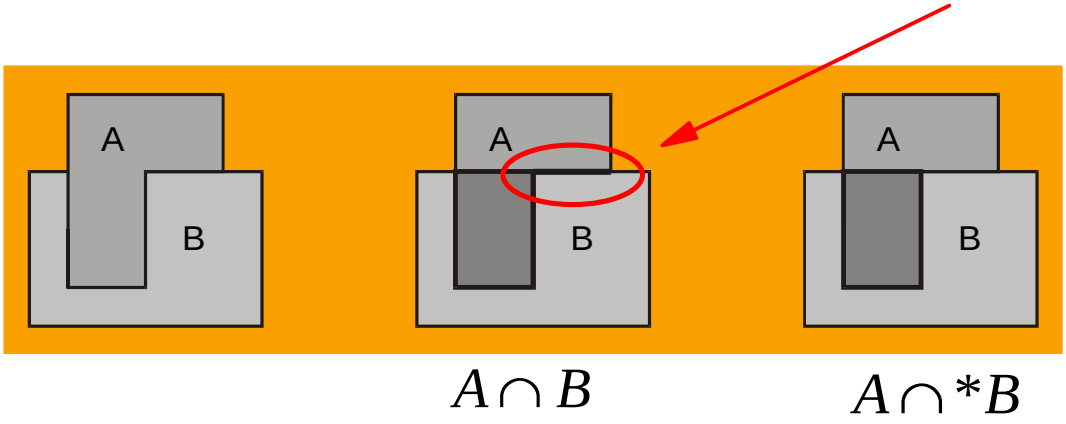
$$A \; op^* \; B = \text{closure}(\text{interior}(A \; op \; B))$$

- **Regularized Boolean operations produce the same result as standard Boolean operations if the resulting objects are solid**

- **Otherwise, they eliminate lower-dimensional features**
  - Dangling edge, faces, points

Combining Solids (3)

- **Example**
  - Standard intersection operation retains a piece of shared boundary
  - Regularized intersection avoid the generation of a dangling edge



Representing Solids

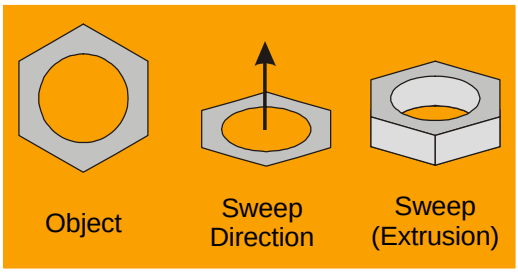
- **Solids can be represented in several ways**
  - Sweeps
  - Boundary representation (b-rep)
  - Decomposition representations
  - Constructive Solid Geometry (CSG)

Sweep Representation (1)

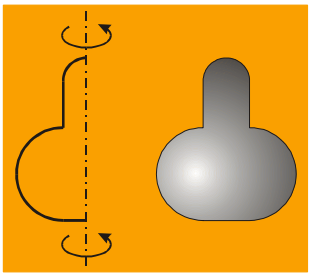
- **Sweeps move an object along a trajectory**
  - Simple and natural way to describe many objects
  - For example, the path of a cutting tool or a robot arm
- **A sweep is described by the object and a trajectory for the sweeping process**
- **Generalized sweep**
  - 2D or 3D object
  - Arbitrary trajectory
  - Transformation of the object along the trajectory

Sweep Representation (2)

- **Extrusion**
  - Simplest sweep
  - The object is a 2D area
  - The trajectory is perpendicular to the object plane



- **Rotational sweep**
  - Rotation of a 2D area about an axis



Sweep Representations (3)

- **General sweeps can generate complex objects**
  - Transformations can change the shape of the swept object
  - Complex trajectories may create self-intersecting object

- **Application of Boolean set operations to general sweeps is difficult**
  - Sweeps are not closed under Boolean operations. E.g. the union of two sweep is generally not a sweep.
  - Sweeping a 2D object within its plane does not generate a solid.

  - Therefore, sweeps are converted to another representation first
  - Sweeps are supported because they are a natural way to model objects

Boundary Representation (B-rep)

- **Only describe the boundary explicitly**
- **Interior is defined implicitly via the boundary**
- **B-reps were conceived as an extension of early ways to represent objects with vectors**

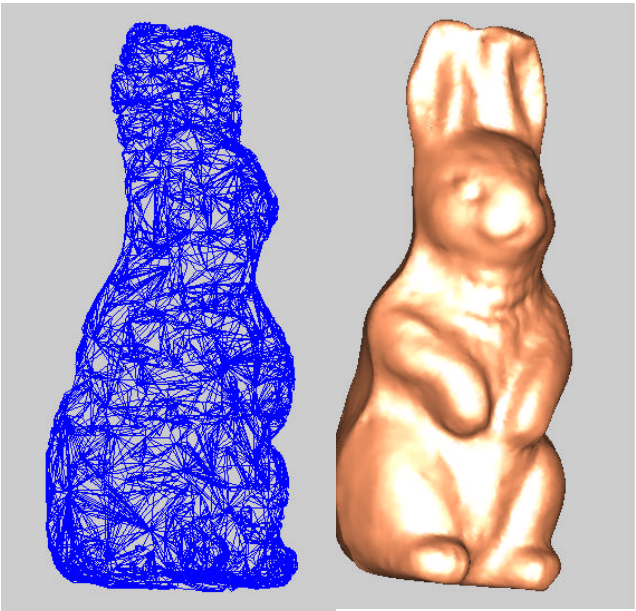
- **The boundary can be described using several techniques**
  - Polygon meshes we have discussed earlier
  - Higher-order or freeform surfaces

- **We will first look at ways to describe polygon meshes, before we look at modeling solids using b-reps.**

Polygon Meshes

- **Polygons are popular primitives to approximate shapes**
  - Linear approximation w/ easily controlled error
  - Conceptually simple, in particular for triangles
  - Efficient rendering with hardware support

- **Polygon meshes are connected sets of polygons**



Creating and Editing of Models

- **Moving of single vertex**
  - Make sure all connected edges and triangles follow
- **Moving of an edge**
  - Make sure that the delimiting vertices and triangles follow
- **Add or delete a triangle**
  - Make sure that neighboring triangles and adjacent edges/vertices are updated
- **We need a data structure that supports such operations**

Data Structures for Polygon Meshes

- **Explicit representation**
- **Indexed representation**
- **Edge-based representations**

Explicit Representation

- **The coordinates for every vertex are stored explicitely in the polygon representation**
  - $P = [(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3), \ldots , (x_n, y_n, z_n)]$
- **Problems**
  - No concept of shared vertices, i.e. two polygons with shared vertices have to replicated the vertex coordinates
  - Difficult to maintain
  - Inefficient memory utilization

Indexed Representations

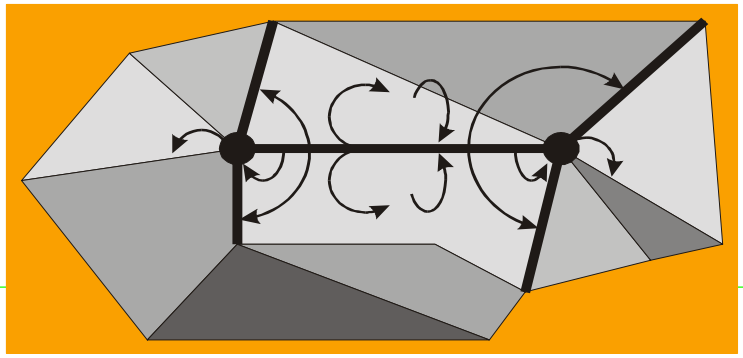
- **To avoid the problem of replicated data, indexed representation use pointers to the actual data**
  - For instance, indexed-face list in VRML
  - $P = [V_1, V_2, V_3, \ldots V_n]$
    $V_1 = (x_1, y_1, z_1)$
    $V_2 = (x_2, y_2, z_2)$
    $V_3 = (x_3, y_3, z_3)$
    ...
- **Advantages**
  - More space-efficient than explicit representation
  - Shared vertices are stored only once
  - Easier to edit and maintain
- **Problems**
  - Still not easy to find polygons sharing an edge or a vertex

Edge-based Representations

- **Polygons are represented by their enclosing edges**
- **Edges are stored using pointers to end points and adjacent polygons**
  - In typical meshes every edge has only 1 or 2 adjacent polygons
  - $P = [E_1, E_2, E_3, \ldots E_n]$
    $V_1 = (x_1, y_1, z_1)$, $V_2 = (x_2, y_2, z_2)$, $V_3 = (x_3, y_3, z_3)$ ...
    $E_1 = (V_1, V_2, P, -)$, $E_2 = (V_2, V_3, P, -)$, $E_3 = (V_3, V_4, P, Q)$ ...
- **Still does not support easy queries for ...**
  - ... all polygons adjacent to a vertex
  - ... all vertices shared by 2 polygons
  - ... all edges meeting at a vertex

Winged-edge Data Structure (1)

- **Edges are (again) the central link between vertices and polygons**
  - For each edge the following is stored
    - Pointers to the vertices V1 and V2, edge is **oriented** from V1 to V2
    - Pointers to the adjacent polygons (left and right defined by edge orientation)
    - Pointers to 4 additional edges (next edges cw and ccw at both ends)
  - For each vertex a pointer to an edge sharing that vertex is stored
  - For each polygon a pointer to one of its edges is stored

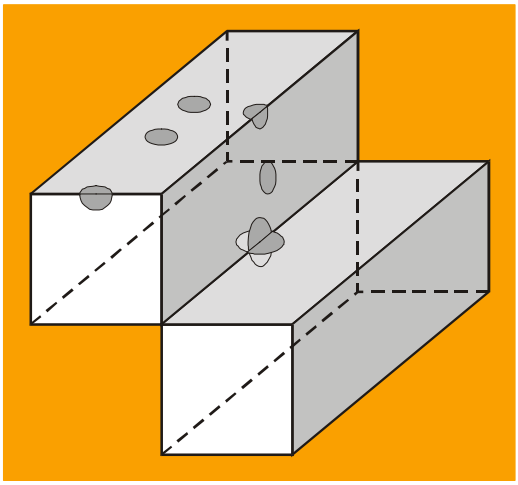


Winged-Edge Data Structure (2)

- **Determine all edges incident to a vertex !**
  - Determine the edge associated with the vertex
  - Follow the edges around the vertex by reading the ccw next edge
  - Stop when the first edge is encountered
- **Determine all faces sharing an edge !**
  - Simply retrieve the adjacent faces from the edge information
- **Determine the adjacent faces for a vertex !**
  - Determine all edges incident to the vertex (see above)
  - For all edges:
    Report left (right) face if edge starts (ends) at the vertex

Winged-Edge Data Structure (3)

- **Winged-edge data structures describes 2-manifolds**
  - 2-manifolds are surfaces where every point on the surface has a (arbitrarily small) neighborhood that is a topological disk
- **Non-manifold surfaces can be described using the radial-edge data structure**



Solids in Boundary Representation (1)

- Only describe the boundary explicitly
- Interior is defined implicitly via the boundary

- Computation of Boolean set operations is somewhat complex because of the many elements involved

Solids in Boundary Representation (2)

- Properties
  - Generally, there is no unique b-rep for a given solid, i.e. the same solid can be described by different b-reps
  - Validity of B-reps is difficult to establish and enforce, e.g. dangling faces, non-manifolds, "open" objects or self-intersecting objects.
    - Topological integrity can be assured by data structures, e.g. winged edge data structure
    - However, there is still the possibility of geometric integrity, e.g. self-intersection.
  - B-reps typically produce only approximations and therefore suffer from accuracy problems
  - B-reps are not as compact as other representations but are convenient for graphics and therefore efficient to display
  - B-rep models are tedious to generate and edit. However, there are algorithms that can convert other representations into B-rep.

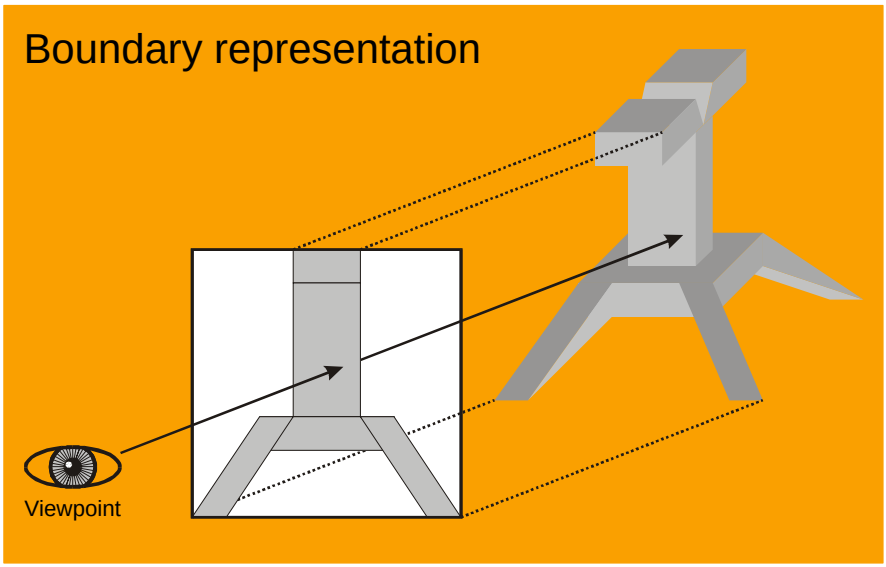
Decomposition Representations

- **Solids are described by combining basic building blocks.**
- **The type of building blocks leads to various representations**
  - Exhaustive Enumeration (Voxel Representations)
  - Cell Decomposition
  - Space Subdivision
    - Quadtrees and Octrees
    - Binary Space-Partitioning Trees (BSP Trees)

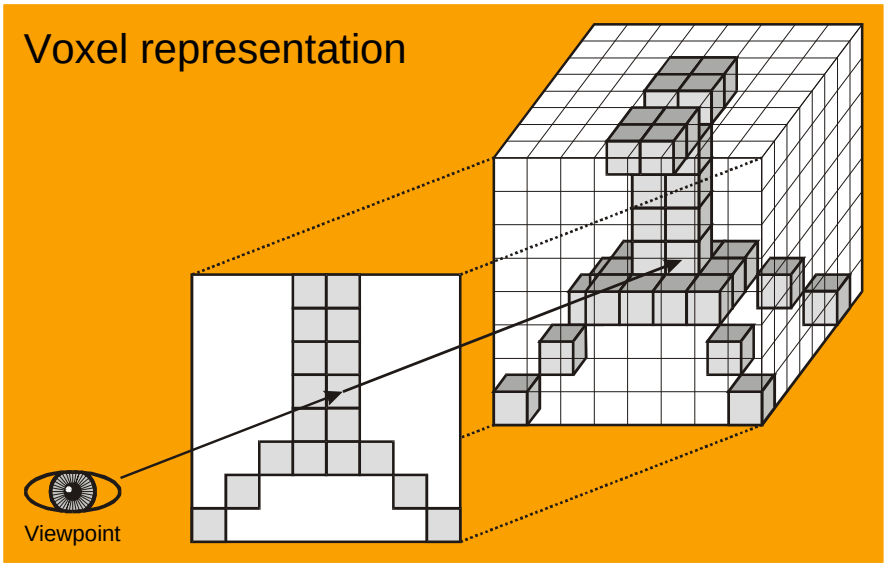
Exhaustive Enumeration

- **The solid is decomposed into identical cells on a regular grid**
  - Grid cells are called *voxels* or (less often) *cuberille*
  - Decomposition is similar to scan-conversion of 2D primitives
- **Voxels**
  - Binary voxels only indicate whether they are occupied (1) or not (0)
  - Multi-valued voxels can represent several values, e.g. color, transparency, or material properties

Exhaustive Enumeration: Example



Exhaustive Enumeration: Example



Exhaustive Enumeration: Properties

- **Only approximates the actual shape (sampling !)**
- **Represents always valid solids (if connectedness is not a requirement)**
- **Voxel representations are unambiguous and unique**
- **Voxel representations are close under Boolean operations**
- **Not very compact**
- **Algorithms are typically simple but slow because of the size of the data.**
  - Simplicity and regularity of algorithms allow easy parallelization.

Cell Decomposition

- **Generalization of the exhaustive enumeration**
  - Instead of identical voxels, cells can be of different shape
  - Cells must be topological spheres, i.e. must not contain holes
  - Cells can be "glued" together to describe a solid
  - Cells may touch but must not have common interior points, i.e. must not intersect
  - Topologically, cells are either disjoint or touch in exactly one corner, edge or face
  - Various cell types are possible, e.g.
    - Polyhedra
    - Curved polyhedra, i.e. a polyhedron bounded by bi-quadratic or bi-cubic patches

Cell Decomposition: Properties

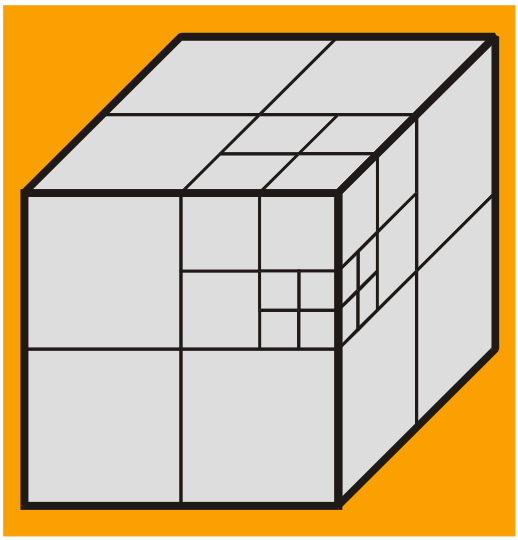
- **Cell decomposition can be very accurate up to the degree of the cell boundaries (typically quadratic)**
- **Hard to establish valid decompositions. Requires test for intersection between all pairs of cells !**
  - There is not structural support as in voxel reps or octrees.
- **Cell decompositions are not unique.**
- **Can be fairly compact**
- **Cell decompositions are usually not close under Boolean set operations**
  - Only few general algorithms for manipulating cell decompositions
  - Cell decomposition is mostly used for analysis purposes, e.g. FEA
  - Cell decompositions are usually generated from another representation of the solid

Space-Subdivision Representations

- **Exhaustive enumeration and cell decomposition subdivide space in fairly regular fashion**
  - Requires large amount of storage
  - Inefficient to process
- **Adaptive subdivision overcomes these problems**
  - Only subdivide space along the solid's boundaries
  - There are usually large areas without a boundary
  - Exploits the fundamental property that the number of cells needed is proportional to the surface area.
    - Proportional to the square $O(r^2)$ of the desired resolution r
    - Exhaustive enumeration requires $O(r^3)$ cells
- **We will look at 2 space subdivision schemes: Octrees and BSP-trees**

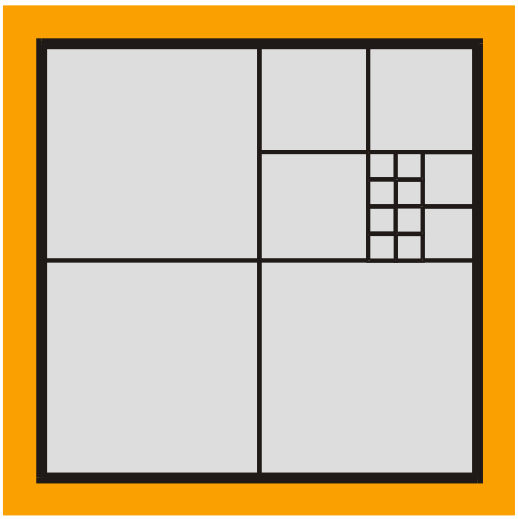
Octrees

- **Recursive, binary subdivision of space**
  - Each subdivision step generates 8 cells
- **Subdivision is controlled by the application**
  - To describe solids, cells are subdivided if they intersects with the solid's boundary
  - Subdivision stops at a pre-determined maximum subdivision level or when the cell interior is homogeneous

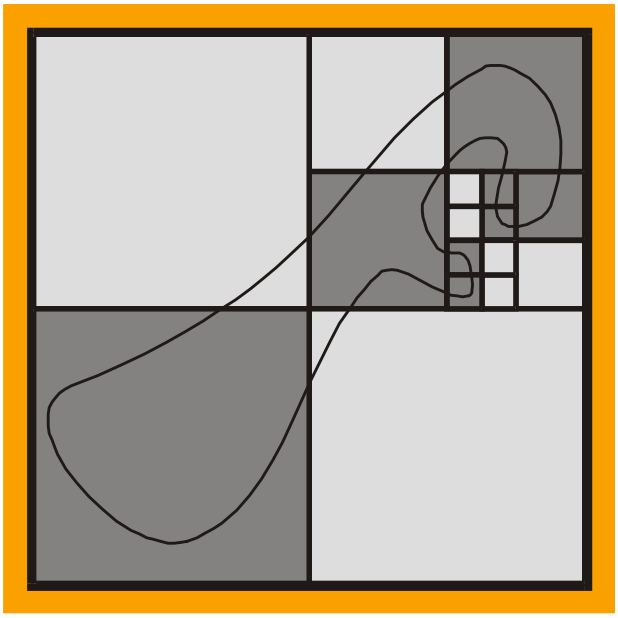


Quadtrees

- **Quadtrees are the 2D equivalent of octrees**
  - Frequently used to subdivide and compress images
- **Concepts apply similarly to octrees and quadtrees**

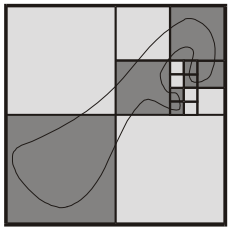


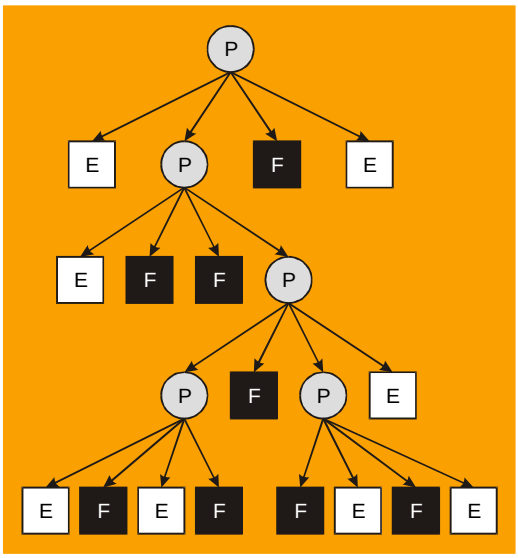
Quadtrees: Example



Describing Quadtrees and Octrees

- **The binary subdivision process is described by a tree structure**
- **The tree contains three types of nodes:**
  - Partially filled nodes
  - Empty leaves
  - Full leaves





Labeling Quadtrees and Octrees

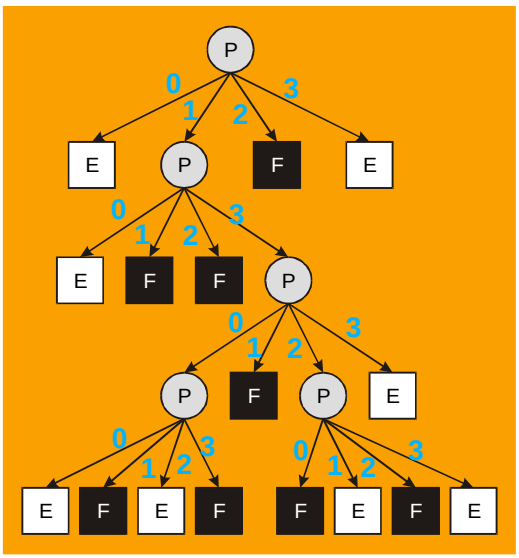
- **The cells in the tree can be labeled and numbered by their location**
  - There is no common convention for labels and numbers
  - Quadtrees (north/south, east/west)
    NW(0), NE(1), SW(2), SE(3)
  - Octrees (front/back, up/down, left/right):
    FUL(0), FUR(1), FDL(2), FDR(3), BUL(4), BUR(5), BDL(6), BDR(7)
- **Number of leaves and nodes is limited**

$$L \le n^{h-1}$$

$$N \le \sum_{i=0}^{h-1} n^i = \frac{n^h - 1}{n - 1}$$

Quadtrees & Octrees: Linear Notations (1)

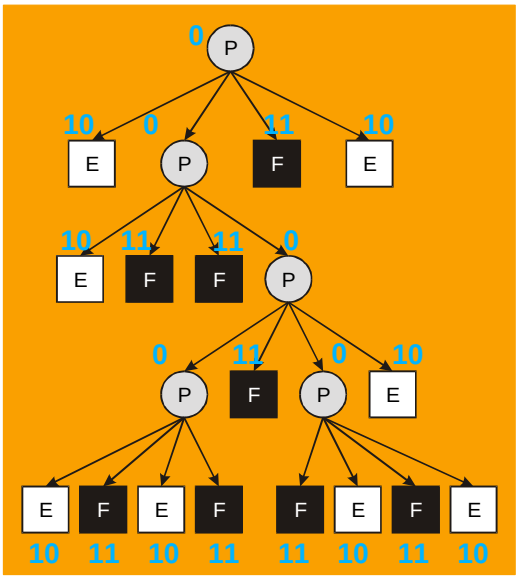
- **Several ways to describe the by a string**
  - Only full (or empty) leaves are enumerated
- **Linear addresses**
  - Terminator symbol for leaves not at the lowest level: X
  - Bits per digit: 2^n+1
    Bits per leave: $h * (2^n+1)$
  - Example (n=2, h=4):
    11XX, 12XX, 1301, 1303, 131X, 1320, 1322, 2XXX
    (Bit count: 8 * 4 * 5 = 160)



Quadtrees & Octrees: Linear Notations (2)

- **Node/leave encoding**
  - Breadth-first traversal
  - Bit 1:
    Internal node (0) or leave (1)
  - Bit 2 (only for leaves):
    Empty (0) or full (1)
  - 1 bit per internal node
    2 bits per leave

  - Example:
    0
    10 0 11 10
    10 11 11 0
    0 11 0 10
    10 11 10 11, 11 10 11 10
    (Bit count: 37)



Octrees: Boolean Set Operations

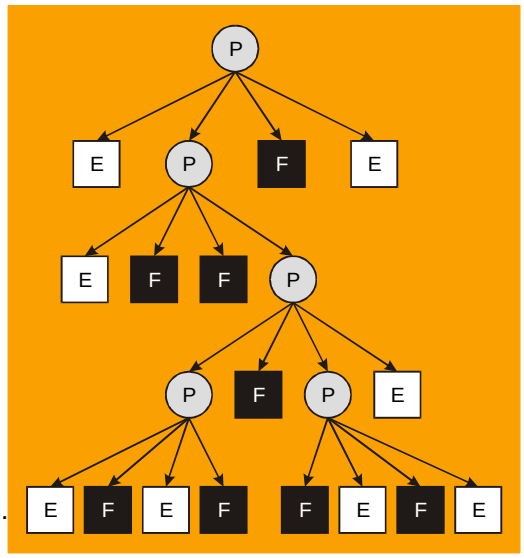
- **Combining two octrees is performed by synchronous traversal of the input trees**
  - An output octree is built by copying nodes from the input trees
- **Nodes are combined according to simple rules**
  - If both nodes are leaves, the leaves are combined according to the Boolean set operation and appended to the output tree.
  - If both nodes are internal nodes, recursive traversal of the children
  - If one node is a leaf and the other is an internal node, the operation determines, whether only the leaf node or the leaf node and the children of the internal node are copied to the output tree.
    - Example: Leaf=1, Operation = AND: Copy leaf node and children of internal node
      Leaf=0, Operation = AND: Only copy leaf node
- **The output octree may not be compact,**
  - Internal nodes may have children that are all full or all empty
  - Post-processing can easily compact an octree into a unique form

Octrees: Neighbor Finding (1)

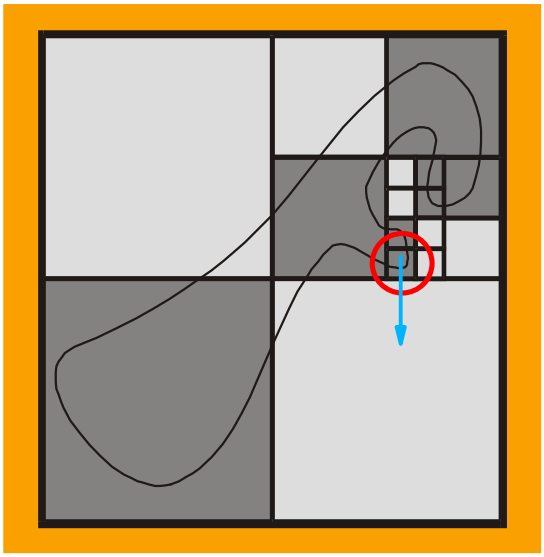
- **Several applications / algorithms require to find the neighbor of a given octree node in a given direction**
  - Space traversal along a given path (ray)
  - Averaging of neighboring nodes to compute normals for rendering
- **Problem**
  - Find that octree node that borders the original in direction D
- **Basic Algorithm**
  - Ascend from the node to a common ancestor with the target node
    - May require ascent all the way to the root node
  - Descend from that common ancestor to the target node

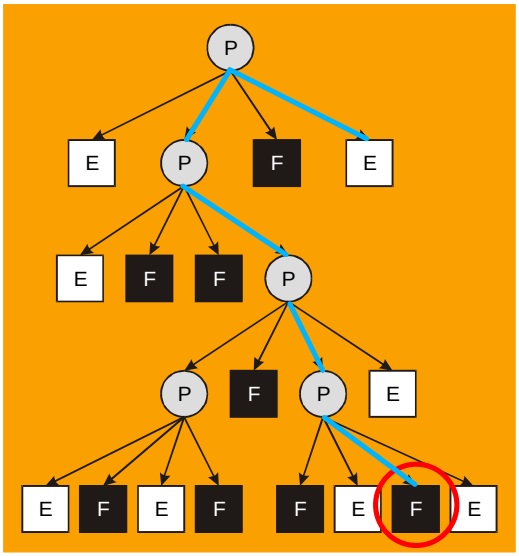
Octrees: Neighbor Finding (2)

- **Finding the Common Ancestor**
  - From the original node, find the first ancestor node that has not been reached from direction D
  - Example:
    - Find a node's Southern neighbor.
    - Climb the tree on an "upward path".
    - For every step upwards, check whether the path is in the Southern half of the node.
    - If not, the common ancestor is found.



Octrees: Neighbor Finding (3)





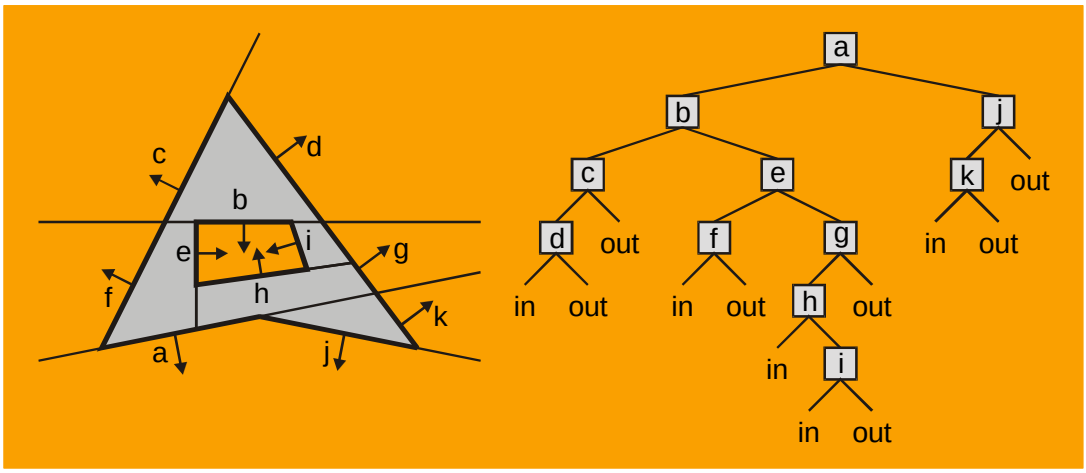
Octrees: Properties

- Octrees approximate the actual shape of the solid
- Like spatial enumeration and cell-based descriptions, octrees are always valid
- A compacted octree is a unique and unambiguous description of the solid
- The number of nodes in the octree are roughly proportional to the the solid's surface area. Although still quite large, octrees are more compact than voxel or cell representations.
- Operations on octrees are closed for Boolean ops
- Octree algorithms are fairly simple as they rely on tree traversal.

Binary-Space Partitioning Trees (BSP Trees)

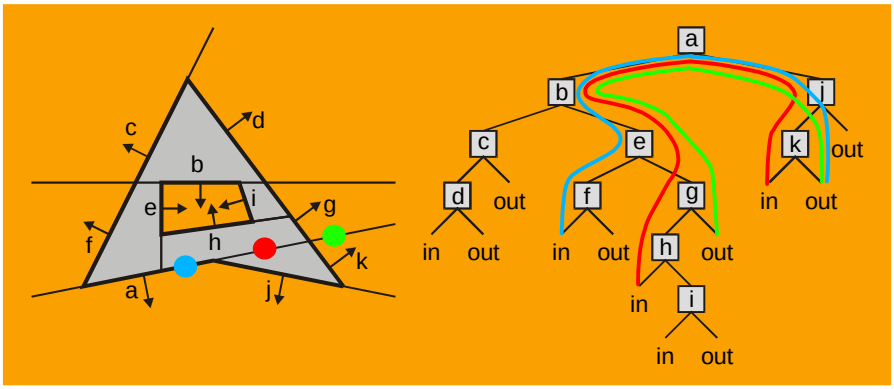
- **Generalization of Octrees**
  - Octrees partition space using orthogonal planes
  - BSP trees use arbitrary planes to subdivide space
- **Binary Space Partitioning (BSP)**
  - Oriented planes divide space into IN cells and OUT cells
  - IN and OUT portions can be subdivided further by more planes
  - To account for limited numerical precision, planes have "thickness"
    - Thickness is a numerical tolerance. Points within the thickness are ON the plane.
- **BSP Trees**
  - Each plane is a node in a tree
  - IN and OUT cells are the children of a node

BSP Trees: Example



BSP Trees: Point Classification

- **Determine whether a point is inside or outside the solid**
  - The point is passed down the BSP tree, starting at the root
  - At every node, the point is tested against the associated plane
  - The point is then recursively descending the tree to a leave
  - If the point is classified as ON, it is passed to **both** nodes



BSP Trees: Properties

- **Dimension independent**
  - Concept of dividing (hyper)planes extends into higher dimensions
- **Linear approximation of actual solid shape**
- **BSP description are not always valid**
  - Can describe objects that are not closed (open half-spaces !)
- **Non-unique. Several BSP trees for the same solid.**
- **BSP trees tend to be more compact than octrees.**
- **Algorithms for closed Boolean operations on BSP trees**
- **BSP tree algorithms are more complicated than octree algorithms**
  - Often involve splitting objects on dividing planes
  - Require attention to numerical precision

Space Subdivision: Applications

- **Space subdivision representations and algorithms were also developed for other application domains**

- **Hidden Surface Removal**
  - Subdivide space along polygons
  - Space subdivision allows to traverse objects front-to-back or back-to-front

- **Collision Detection**
  - Animations and simulations require detection of collisions between objects
  - Dynamically updated spacial partitioning allows to quickly determine candidates for potential object collisions

Constructive Solid Geometry (CSG)

Constructive Solid Geometry (CSG)

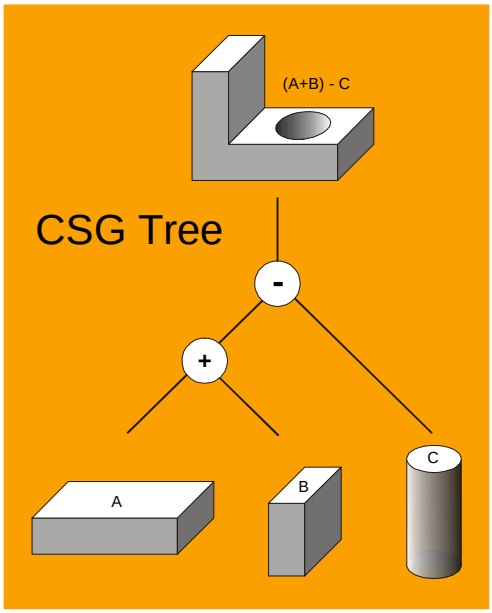
- **CSG builds solid models by hierarchically combining primitive solids**
- **Most basic primitives are half-spaces, e.g.**
  - Planes: $ax + by + cd + e < 0$
  - Cylinders: $x^2 + b^2 - r^2 < 0$
  - Spheres: $x^2 + y^2 + z^2 - r^2 < 0$
- **Half-spaces and resulting solids are combined using Boolean set operations**
  - Union A + B
  - Intersection A * B
  - Difference A - B = A * (-B)

CSG Tree

- **The combination of half-spaces using the binary Boolean operators creates a binary tree, the CSG tree**
- **The tree mirrors the Boolean expression describing the CSG object**
- **CSG expressions can be manipulated to create equivalent expression of the same object**
  - Boolean algebra, e.g. distribution of terms or De Morgan's law
  - In particular, CSG expressions can be brought into conjunctive or disjunctive normal forms
  - Then, the CSG tree has only 2 levels, e.g. sum of intersections

CSG: Example

- The union of the blocks forms the bracket
- Subtracting the cylinder creates the hole



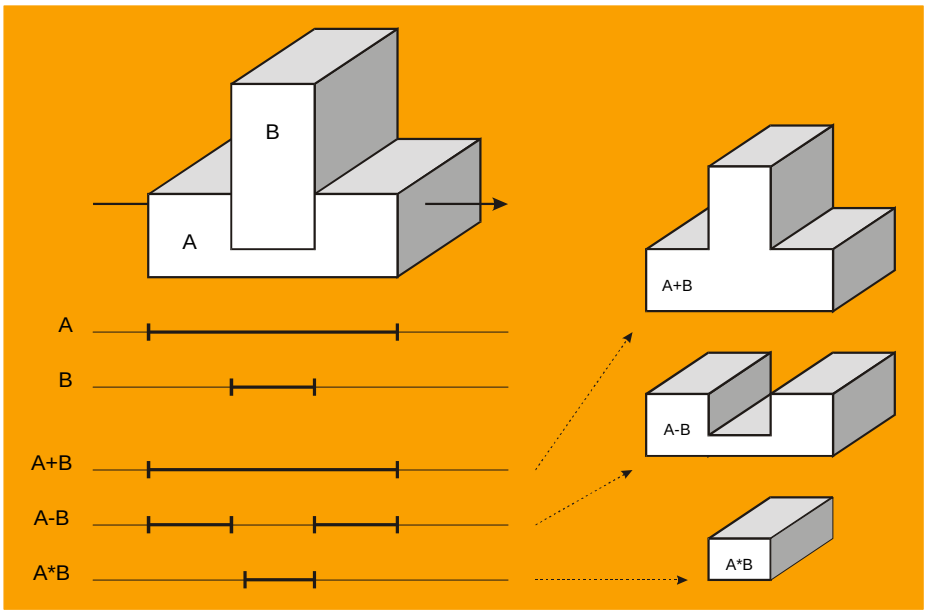
Rendering CSG Objects

- **Boundary Evaluation**
  - Calculate the boundary of the CSG solid
  - Render the boundary using standard (polygon) rendering techniques
  - Requires intersection of faces, edges and points
  - Intersection calculations can be complicated when allowing higher-order halfspaces like cylinders or tori
  - Can result in hairy case analyses, in particular in the presence of numerical errors
- **Ray-Casting**

Ray-Casting CSG Objects (1)

- **Ray Casting**
  - Cast rays from the eye into the scene
  - Determine first intersection of rays with objects to determine visible object
  - Approximation as the object is sampled by the rays
- **Ray-casting CSG objects**
  - Much simpler than boundary evaluation as it only requires intersection of lines with half-spaces
  - Each half-space segments the line into IN and OUT segments
  - These segments are combined using the Boolean expression describing the CSG object
  - Requires computation of **all** intersections of ray and object

Ray-Casting CSG Objects (2)



Ray-Casting CSG Objects (3)

- **The IN segments can also be used to approximate volume/mass and center of gravity of the CSG object**

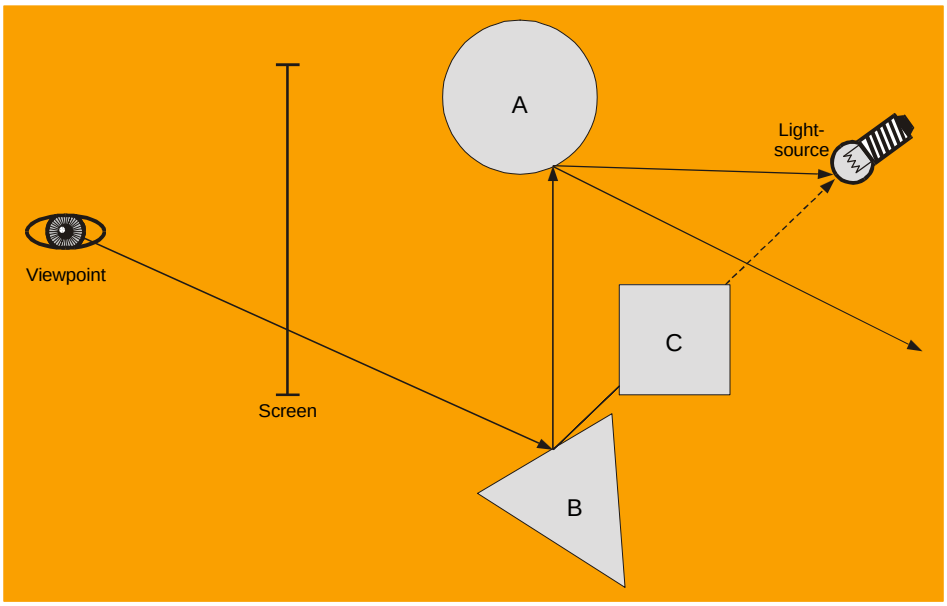
- **Generalized CSG primitives**
  - Ray-casting requires only a small set of operations to be supported for a primitive, namely intersection with a ray
  - Therefore, other primitive types than only half-spaces can easily be integrated into CSG
  - For instance, polygonal models can be easily incorporated in a CSG modeler. This bridges the gap between CSG and b-reps.
  - Another example is the integration of sweeping primitives with CSG
  - (Sweeping, moves a shape along a trajectory cover, i.e. sweeping out, a part of space)

Ray Tracing

Ray Tracing (1)

- **Ray Tracing follows rays through the scene**
- **Typically, rays are traced from the eye into the scene**
- **As rays hit objects new rays are generated**
  - Shadow rays to determine whether the object is lit
    - If exposed to a light source, the lighting model is computed
  - Reflected ray to create inter-object reflections
  - Refracted ray if the object is transparent
- **The pixel color is computed as the combination of several contributions**
  - Light directly received from the light source(s) and
  - Light received from other objects
  - Ambient light

Ray Tracing (2)



Forward vs. Backward Ray Tracing

- **Forward Ray Tracing**
  - Rays are starting at the light source(s) and traced through the scene until they hit the eye
  - Approximates how light propates in the physical world
  - Chances of finding a ray from a light source that actually hits the eye are very small.
  - Therefore, forward ray tracing is very inefficient
- **Backward Ray Tracing**
  - Inverts the forward ray tracing process by tracing rays from the eye back to the light source.
  - Only uses rays that are in fact hitting the eye ... more efficient

Global Illumination

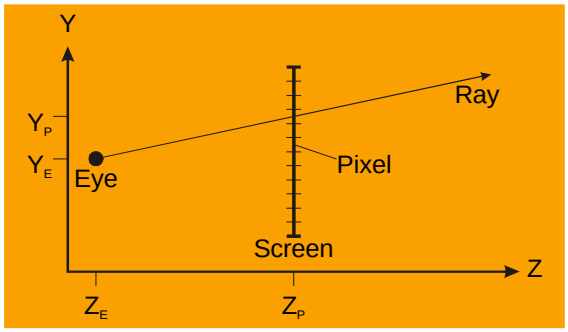
- **Simplistic rendering algorithms, e.g. raster pipeline (OpenGL) only account for direct interactions between light and objects**
  - Secondary effects are crudely approximated using ambient light
- **Global Illumination describes a class of methods that try to capture the overall distribution of light (energy) in a scene**
  - Models higher-order effects, e.g. self-shadowing, inter-object reflections or light attenuation
  - For reflective surfaces, ray-tracing is adequate (ray optics)
  - Radiosity algorithms capture the interaction of diffuse reflectors (global energy balance)

Ray Tracing: Steps

- **We will now look more carefully at the steps involved in the ray-tracing process**
  - Generate rays
  - Find ray-object intersections and choose the closest one
  - Cast rays towards all light sources and compute lighting model if lit
  - Generate secondary rays and recursively trace them
  - Combine the contribution of all rays at a surface point, shadow rays and secondary rays

Ray Generation

- **At least one ray has to be generated for every pixel**
  - Several rays per pixel can be used to implement anti-aliasing
  - Rays are determined by the view geometry, i.e. position of the eye and the screen with respect to the scene
- **Ray is described as point and direction**
  - *t*>0 for numerical precision



$$\mathbf{R} = \mathbf{R}_{orig} + t \cdot \mathbf{R}_{dir}$$

with $|\mathbf{R}_{dir}| = 1$ (Normalized)

and $t > 0$ (Semi - infinte ray)

Here: $\mathbf{R}_{orig} = \mathbf{E}$ and $\mathbf{R}_{dir} = \dfrac{\mathbf{P} - \mathbf{E}}{|\mathbf{P} - \mathbf{E}|}$

Ray-Object Intersection

- **Once rays enter the scene, their intersections with the objects in the scene must be computed**
- **Each primitive type needs special intersection routine**
  - (Suggests object-oriented programming approach !)
  - Simple for polyhedra or spheres
  - More complicated for higher-order primitives
- **Typically, only the nearest intersection with the object is needed**
  - CSG operations require all intersections to define the IN and OUT segments

Ray-Object Intersection: Polygon (1)

- **We assume that polygons are planar**
  - First, compute ray-plane intersection
  - Then, determine whether intersection is inside the polygon
- **Plane:**

$$Ax + By + Cz + D = 0 \quad \text{with } A^2 + B^2 + C^2 = 1$$

$$\mathbf{N} \cdot \mathbf{P} = 0 \quad \text{with} \quad \mathbf{N} = \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} \quad \text{and} \quad \mathbf{P} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

- **Ray-Plane Intersection:**

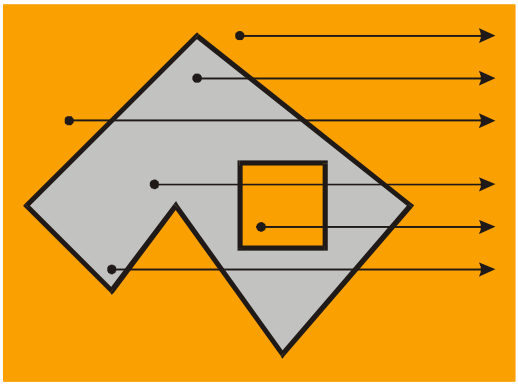
$$\mathbf{N} \cdot \left(\mathbf{R}_{orig} + t_i \cdot \mathbf{R}_{dir} \right) = 0 \quad \Rightarrow \quad t_i = -\frac{\mathbf{N} \cdot \mathbf{R}_{orig}}{\mathbf{N} \cdot \mathbf{R}_{dir}}$$

Ray-Object Intersection: Polygon (2)

- **Ray intersects the plane iff:** $t_i > 0$
- **Intersection point:** $\mathbf{P}_i = \mathbf{R}_{orig} + t_i \cdot \mathbf{R}_{dir}$
- **Now the intersection point has to be tested against the polygon (point in polygon test):**
  - Simplify this test by projecting the polygon along one of the axes
  - Pick axis by finding the largest coordinate in the normal vector ***N***
  - Project by "dropping" that coordinate from all polygon vertices and the intersection point. Call the remaining axes U and V.
  - Example: If ***N*** = (1, 4, 2) then project onto XZ-plane, i.e. drop Y.
    - X axis becomes U axis and Y axis becomes V axis.
  - This process does **not** change the polygon topology or the location of the intersection point with respect to the polygon !

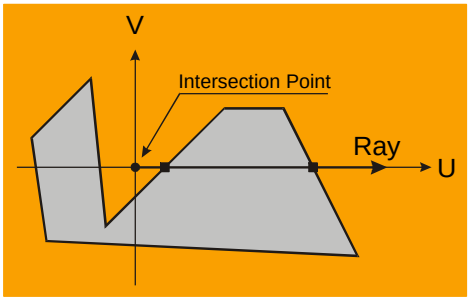
Ray-Object Intersection: Polygon (3)

- **Now we have to solve the 2D point in polygon problem**
  - For general polygons, the interior can be defined in several ways.
  - We will use the Jordan curve theorem:
    A point is inside the polygon, if an infinite ray from the point intersects an odd number of polygon edges.



Ray-Object Intersection: Polygon (4)

- **Practical implementation:**
  - Translate the projected intersection point falls onto the origin
  - Pick one positive axis as the ray to be tested against the polgon
  - Count number of intersections of that axis with the polygon edges
    - Attention: Each vertex must belong to only one edge to avoid double-counting !
  - True intersection calculation is only necessary if the two vertices are in diagonally opposed quadrants.
  - Other cases can be handled trivially.



Ray-Object Intersections: Sphere (1)

- **Spheres are frequently used in ray-traced images**
  - "Spheres floating over checker boards"
  - Ray-Sphere intersection is very simple to compute

- **The sphere is described by center S_C and radius S_R**

$$S_R^2 = (X - X_C)^2 + (Y - Y_C)^2 + (Z - Z_C)^2$$

- **Substituting the ray coordinates for *X*, *Y* and *Z*:**

$$S_R^2 = (X_{orig} + t \cdot X_{dir} - X_C)^2 + (Y_{orig} + t \cdot Y_{dir} - Y_C)^2 + (Z_{orig} + t \cdot Z_{dir} - Z_C)^2$$

- **This simplifies to a quadratic expression:**

$$0 = A \cdot t^2 + B \cdot t + C$$

$$A = X_{dir}^2 + Y_{dir}^2 + Z_{dir}^2 \quad (= 1 \text{ if ray direction is normalized})$$

$$B = 2 \cdot \left(X_{dir} \cdot (X_{orig} - X_C) + Y_{dir} \cdot (Y_{orig} - Y_C) + Z_{dir} \cdot (Z_{orig} - X_C) \right)$$

$$C = (X_{orig} - X_C)^2 + (Y_{orig} - Y_C)^2 + (Z_{orig} - Z_C)^2 - S_R^2$$

Ray-Object Intersections: Sphere (2)

- **Solving the quadratic expression for A=1:**

$$t_{i1} = \frac{-B - \sqrt{B^2 - 4C}}{2} \quad \text{and} \quad t_{i2} = \frac{-B + \sqrt{B^2 - 4C}}{2}$$

  - No intersection if t_{i1} and t_{i2} are complex or both t_i less than zero
  - Ray starts inside the sphere if only one solution is real and positive
  - For two intersections: nearest intersection for smaller t_i

- **The normal at the intersection point is**

$$N_S = \begin{pmatrix} (X_i - X_C)/S_R \\ (Y_i - Y_C)/S_R \\ (Z_i - Z_C)/S_R \end{pmatrix}$$

Ray-Object Intersections: Cylinder

- **Cylinders are frequently used to describe pipes, connections etc. or to create holes in a CSG object**

- **We will consider the intersection of an arbitrary ray with an axis-aligned infinite cylinder**
  - The cylinder is described by its radius:

$$C_R^2 = (X - X_C)^2 + (Y - Y_C)^2$$

  - Substituting the ray coordinates gives:

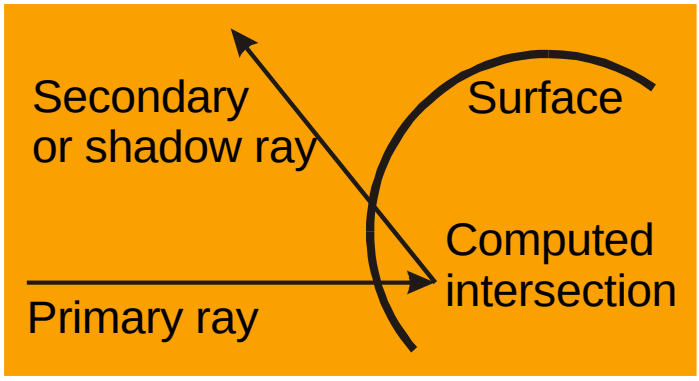
$$C_R^2 = (X_{orig} + t \cdot X_{dir} - X_C)^2 + (Y_{orig} + t \cdot Y_{dir} - Y_C)^2$$

  - This results in a quadratic equation:

$$0 = t^2 \cdot (X_{dir}^2 - Y_{dir}^2) + t \cdot 2\left[X_{dir}(X_{orig} - X_C) + Y_{dir}(Y_{orig} - Y_C)\right] + \left[(X_{orig} - X_C)^2 + (Y_{orig} - Y_C)^2 - C_R^2\right]$$

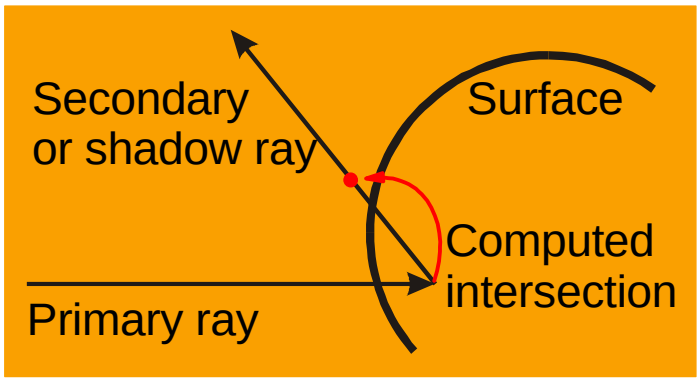
Ray-Object Intersections: Numerical Precision (1)

- **Intersection calculations are done using floating point operations, giving rise to numerical problems**
  - The computed intersection may actually fall inside the object
  - Secondary rays will then be intersecting the object again
  - This effect is known as self-shadowing (a.k.a. surface acne)



Ray-Object Intersections: Numerical Precision (2)

- **Possible solutions**
  - Exclude the object from intersection calculations for secondary rays
  - Introduce a numerical tolerance for parameter t indicating that the ray starts on the surface
  - Move the intersection point along the ray to be on the proper side of the surface



Ray-Object Intersections: Coordinates

- **All objects and rays are specfied in world coordinates**
  - Objects are transformed from local coordinates into world coordinates: modeling transformation
  - Pixels must be transformed from screen space into world coordinates: inverse viewport mapping
- **Intersection of ray and object is computed by transforming the ray into the local object coordinates**
  - Simply apply the inverse of the modeling transformation to the ray
  - Intuition: Transform ray **and** object back into the local coordinates

Ray-Object Intersection: Nearest Object

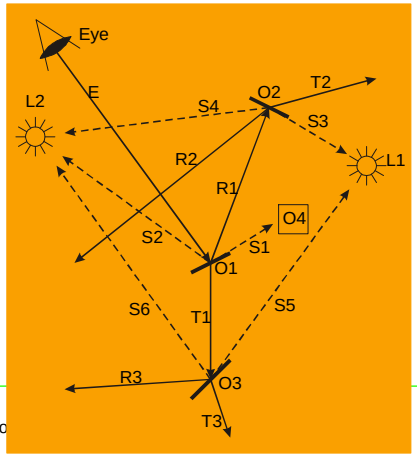
- **To determine the visible object from a given point, the closest object intersect by the ray must be found**
- **For all ray-object intersections, find the one that has the smallest ray parameter *t***
- **For this intersection point, compute the surface color**

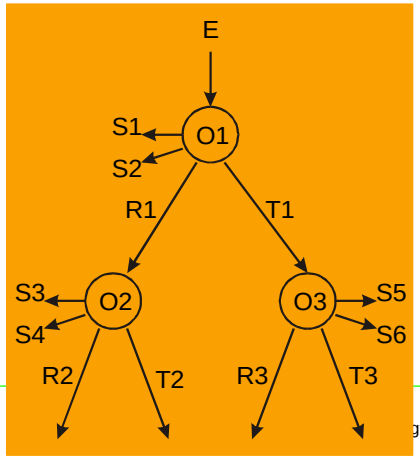
Light/Shadow Rays

- **From every ray-object intersection point, rays are traced to the light source(s)**
  - Light rays are tested against the scene to determine whether the intersection point can see the light source, i.e. whether it is lit
- **Other information available at the intersection point:**
  - Surface normal
  - Ray direction
  - Material properties
- **This are all the parameters needed to compute a lighting model at the intersection point**

Secondary Rays

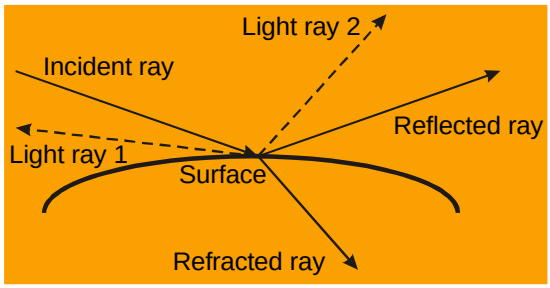
- **At every intersection point secondary rays are generated according to ray optics**
  - Reflected rays to model surface mirroring
  - Refracted rays for translucent materials
- **Recursive tracing of rays creates a ray tree**





Combining Ray Intensities

- **The color/intensity reflected from a surface point depends on two components**
  - Light directly received from light sources (light rays)
  - Light received indirectly from other objects (reflected/refracted rays)



- **The components are superimposed (summed up) to compute the color of the incident ray**

$$I_{\text{Ray}} = I_{\text{Reflected}} + I_{\text{Refracted}} + \sum_k I^k_{\text{Light}}$$

Ray Tracing Optimizations (1)

- **Many ray tracers spend 80% of the time performing intersection calculations.**
- **Optimizations try to reduce the number of intersection calculations to speed up ray tracing**

- **Hierarchical object description**
  - Rays are first tested against higher hierarchy levels
  - Intersections with lower hierarchy levels are only computed if the ray intersects the higher hierarchy
  - Frequently, bounding boxes (or other bounding volumes) are computed for objects. Rays are first tested against the bounding volumes.

Ray Tracing Optimizations (2)

- **Space partitioning**
  - To avoid testing of objects entirely, space is partitioned
  - If the ray does not enter a cell, none of the objects in this cell are tested against the ray
  - Popular partitioning schemes include octrees and BSP trees

Ray Tracing: Anti-aliasing

- **Aliasing is introduced by several sources**
  - Geometry (spatial aliasing)
  - Object motion (temporal aliasing)
  - Material properties (light aliasing) --> radiosity
- **Geometric aliasing can be alleviated by supersampling**
  - Not a complete cure but reduces the artifacts
  - Implementation is simple, shoot several rays at every pixel and filter the resulting sub-pixel values
  - Adaptive supersampling adds samples only in pixel with high color gradient
  - Stochastic supersampling uses randomly distributed sub-pixel. Introduces noise instead of aliasing.

Summary

- **Solid modeling**
  - Definition of what makes an object a solid
  - Different techniques to describe a solid object
  - Properties of each representation
- **Ray Tracing**
  - Global illumination rendering technique
  - Simulates (backwards) the propagation of light rays
  - Works with surfaces and solids

Further Study

- **Solid Modeling**
  - Christoph M. Hoffmann, *Geometric & Solid Modeling*, Morgan Kaufmann Publishers, 1989
  - Martti Mäntylä, *An Introduction to Solid Modeling*, Computer Science Press, 1988
- **Ray Tracing**
  - Andrew Glassner (editor), *An Introduction to Ray Tracing*, Academic Press, 1989.
  - Andrew Glassner, *Principals of Digital Image Synthesis*, Morgan Kaufmann Publishers, 1995

Homework

- **Study textbook: solid modeling (chapter 12) and ray tracing (chapter 15.10 and 16.12)**
- **Prepare for next week by reading about radiosity (chapter 16.13)**
- **Start working on final assignment**
  - To be handed out on Friday
  - Start early !!!