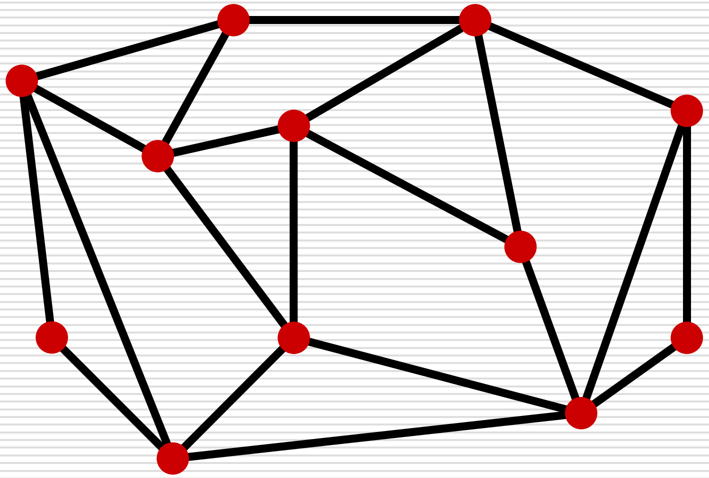
Game Programming

Bing-Yu Chen
National Taiwan University

Game Geometry

- Graph and Meshes
- Surface Properties
- Bounding Volumes
- Spatial Partitioning
- Level-of-Details

Standard Graph Definitions

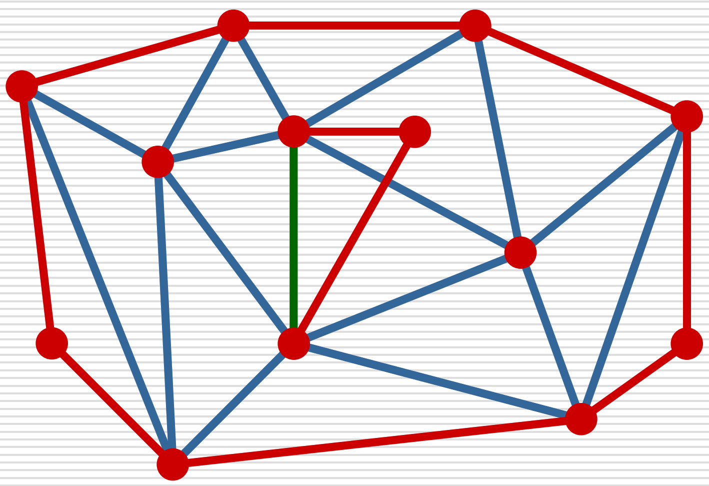


G=<**V**,**E**>
V=vertices={A,B,C,D,E,F,G,H,I,J,K,L}
E=edges=
{(A,B),(B,C),(C,D),(D,E),(E,F),(F,G),
(G,H),(H,A),(A,J),(A,G),(B,J),(K,F),
(C,L),(C,I),(D,I),(D,F),(F,I),(G,K),
(J,L),(J,K),(K,L),(L,I)}

Vertex degree (valence)=number of edges incident on vertex
Ex. deg(J)=4, deg(H)=2
***k*-regular** graph=graph whose vertices all have degree *k*

Face: cycle of vertices/edges which cannot be shortened
F=faces=
{(A,H,G),(A,J,K,G),(B,A,J),(B,C,L,J),(C,I,J),(C,D,I),
(D,E,F),(D,I,F),(L,I,F,K),(L,J,K),(K,F,G)}

Meshes



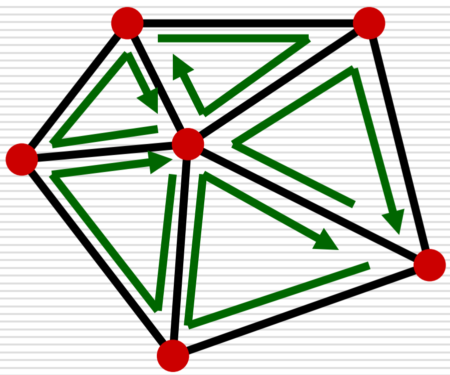
Mesh: straight-line graph embedded in R^3

Boundary edge: adjacent to exactly *one* face
Regular edge: adjacent to exactly *two* faces
Singular edge: adjacent to more than *two* faces

Corners $\subseteq$ V x F
Half-edges $\subseteq$ E x F

Closed Mesh: mesh with no boundary edges
Manifold Mesh: mesh with no singular edges

Orientability



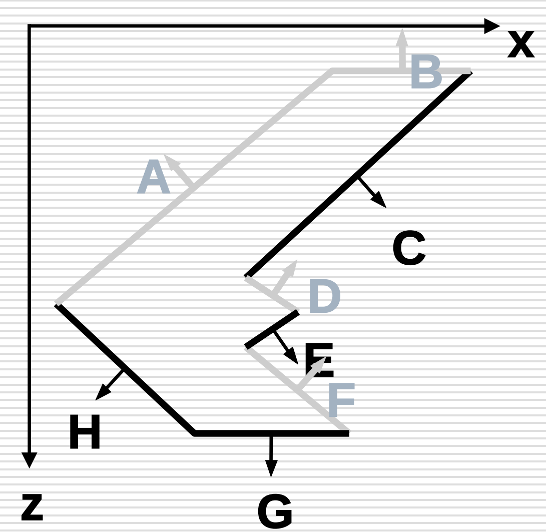
Orientation of a face is clockwise or anticlockwise order in which its vertices and edges are lists

This defines the direction of face **normal**

Oriented
F={(L,J,B),(B,C,L),(L,C,I),
(I,K,L),(L,K,J)}

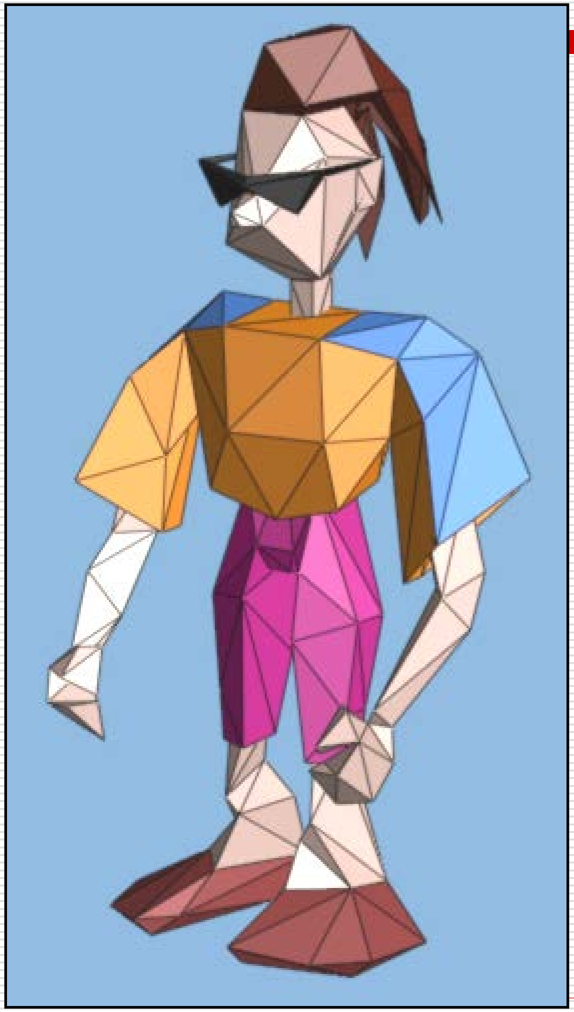
Not Oriented
F={(B,J,L),(B,C,L),(L,C,I),
(L,I,K),(L,K,J)}

Straight line graph is **orientable** if orientations of its faces can be chosen so that each edge is oriented in *both* directions



Back Face Culling = Front Facing

Definitions of Triangle Meshes



[Hoppe 99']

$\{f_1\} : \{ v_1 , v_2 , v_3 \}$
$\{f_2\} : \{ v_3 , v_2 , v_4 \}$
…

connectivity

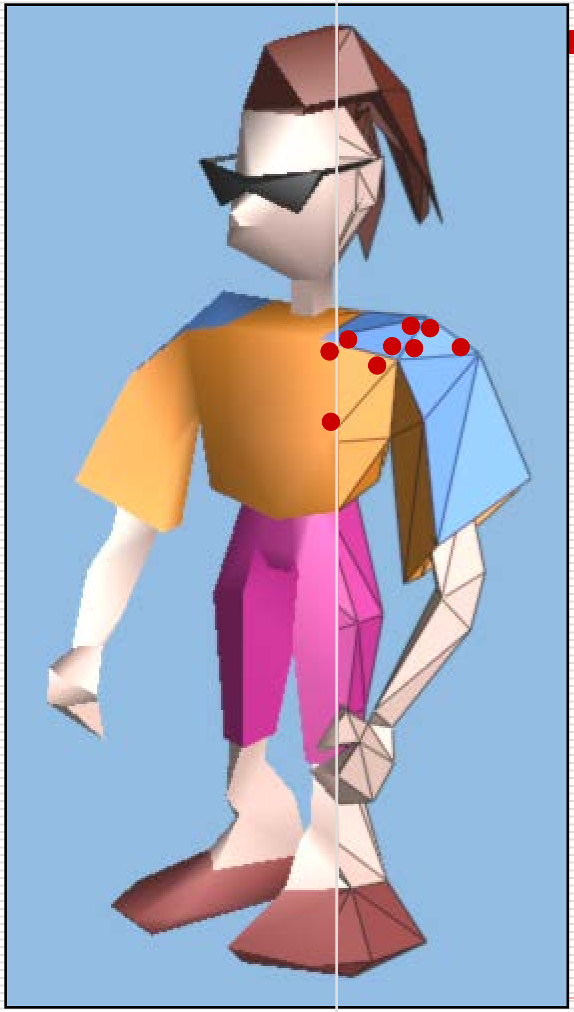
$\{v_1\} : (x,y,z)$
$\{v_2\} : (x,y,z)$
…

geometry

$\{f_1\}$: *"skin material"*
$\{f_2\}$: *"brown hair"*
…

face attributes

Definitions of Triangle Meshes



[Hoppe 99']

$\{f_1\} : \{ v_1 , v_2 , v_3 \}$
$\{f_2\} : \{ v_3 , v_2 , v_4 \}$
…

connectivity

$\{v_1\} : (x,y,z)$
$\{v_2\} : (x,y,z)$
…

geometry

$\{f_1\}$: *"skin material"*
$\{f_2\}$: *"brown hair"*
…

face attributes

$\{v_2,f_1\} : (n_x,n_y,n_z)\ (u,v)$
$\{v_2,f_2\} : (n_x,n_y,n_z)\ (u,v)$
…

corner attributes

Mesh Data Structures

- Uses of mesh data:
  - Rendering
  - Geometry queries
    - What are the vertices of face #3?
    - Are vertices i and j adjacent?
    - Which faces are adjacent face #7?
  - Geometry operations
    - Remove/add a vertex/face
    - Mesh simplification
    - Vertex split, edge collapse
- Storage of generic meshes
  - hard to implement efficiently
- Assume: **orientable**, **manifold** and **triangular**

Storing Mesh Data

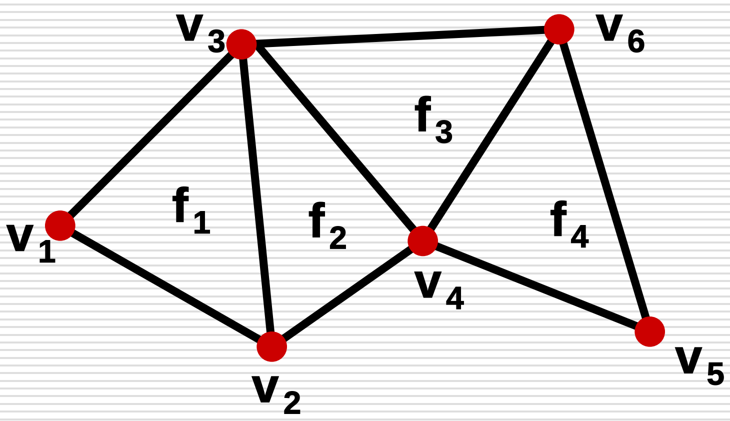
- How “good” is a data structure?
  - Time to construct – preprocessing
  - Time to answer a query
  - Time to perform an operation
    - update the data structure
  - Space complexity
  - Redundancy

1. List of Faces

- List of vertices (coordinates)
- List of faces
  - triplets of pointers to face vertices (c_1, c_2, c_3)
- Queries:
  - What are the vertices of face #3?
    - O(1) – checking the third triplet
  - Are vertices i and j adjacent?
    - A pass over all faces is necessary – NOT GOOD

1. List of Faces

- Example



vertex	coordinate
v_1	(x_1,y_1,z_1)
v_2	(x_2,y_2,z_2)
v_3	(x_3,y_3,z_3)
v_4	(x_4,y_4,z_4)
v_5	(x_5,y_5,z_5)
v_6	(x_6,y_6,z_6)

face	vertices (ccw)
f_1	(v_1,v_2,v_3)
f_2	(v_2,v_4,v_3)
f_3	(v_3,v_4,v_6)
f_4	(v_4,v_5,v_6)

1. List of Faces

- Pros:
  - Convenient and efficient (memory wise)
  - Can represent non-manifold meshes
- Cons:
  - Too simple – not enough information on relations between vertices and faces

OBJ File Format (simple ver.)

- v x y z
- vn i j k
- f v1 // vn1 v2 // vn2 v3 // vn3

2. Adjacency matrix

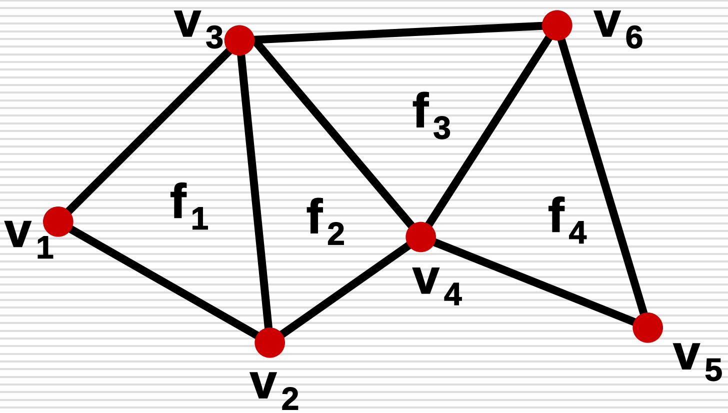
- View mesh as connected graph
- Given n vertices build nxn matrix of adjacency information
  - Entry (i,j) is TRUE value if vertices i and j are adjacent
- Geometric info
  - list of vertex coordinates
- Add faces
  - list of triplets of vertex indices (v_1,v_2,v_3)

2. Adjacency matrix

- Example

vertex	coordinate
v_1	(x_1,y_1,z_1)
v_2	(x_2,y_2,z_2)
v_3	(x_3,y_3,z_3)
v_4	(x_4,y_4,z_4)
v_5	(x_5,y_5,z_5)
v_6	(x_6,y_6,z_6)

face	vertices (ccw)
f_1	(v_1,v_2,v_3)
f_2	(v_2,v_4,v_3)
f_3	(v_3,v_4,v_6)
f_4	(v_4,v_5,v_6)



	v_1	v_2	v_3	v_4	v_5	v_6
v_1		1	1			
v_2	1		1	1		
v_3	1	1		1		1
v_4		1	1		1	1
v_5				1		1
v_6			1	1	1	

2. Adjacency matrix

- Queries:
  - What are the vertices of face #3?
    - O(1) – checking the third triplet of faces
  - Are vertices i and j adjacent?
    - O(1) – checking adjacency matrix at location (i,j)
  - Which faces are adjacent of vertex j?
    - Full pass on all faces is necessary

2. Adjacency matrix

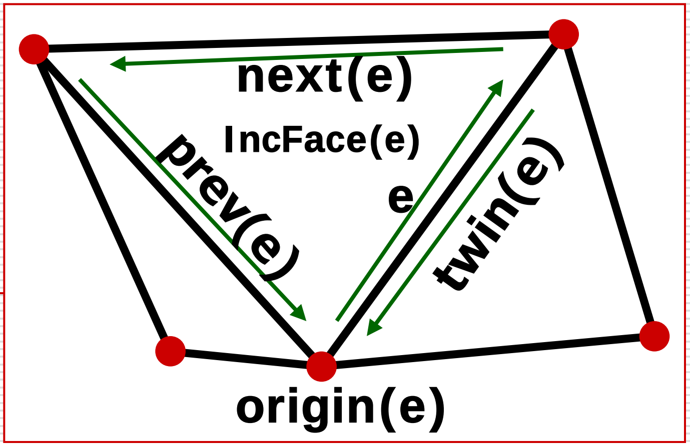
- Pros:
  - Information on vertices adjacency
  - Stores non-manifold meshes
- Cons:
  - Connects faces to their vertices, BUT NO connection between vertex and its face

3. DCEL (Doubly-Connected Edge List)

- Record for each face, edge and vertex
  - Geometric information
  - Topological information
  - Attribute information

- aka Half-Edge Structure

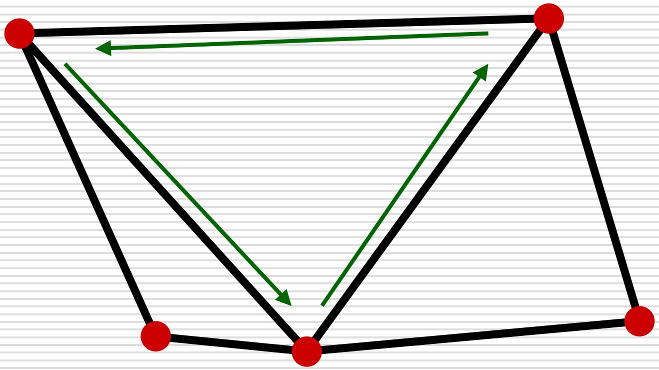
3. DCEL



- Vertex record:
  - Coordinates
  - Pointer to one half-edge that has v as its origin
- Face record:
  - Pointer to one half-edge on its boundary
- Half-edge record:
  - Pointer to its origin, origin(e)
  - Pointer to its twin half-edge, twin(e)
  - Pointer to the face it bounds, IncidentFace(e)
    - face lies to left of e when traversed from origin to destination
  - Next and previous edge on boundary of IncidentFace(e), next(e) and prev(e)

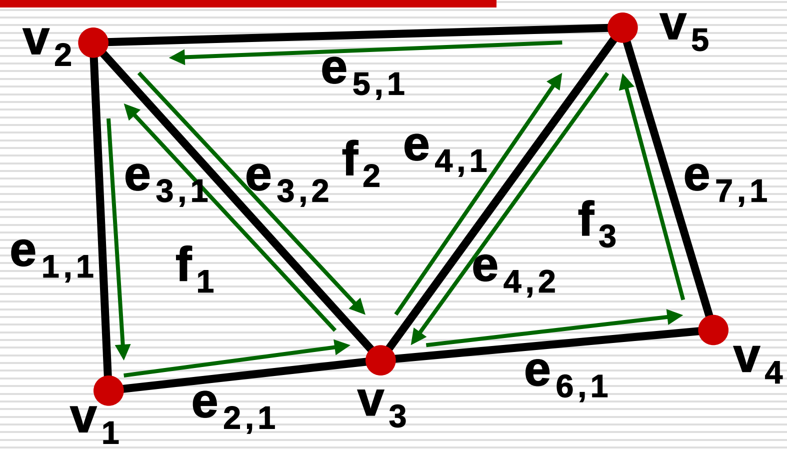
3. DCEL

- Operations supported:
  - Walk around boundary of given face
  - Visit all edges incident to vertex v
- Queries:
  - Most queries are O(1)



3. DCEL

- Example

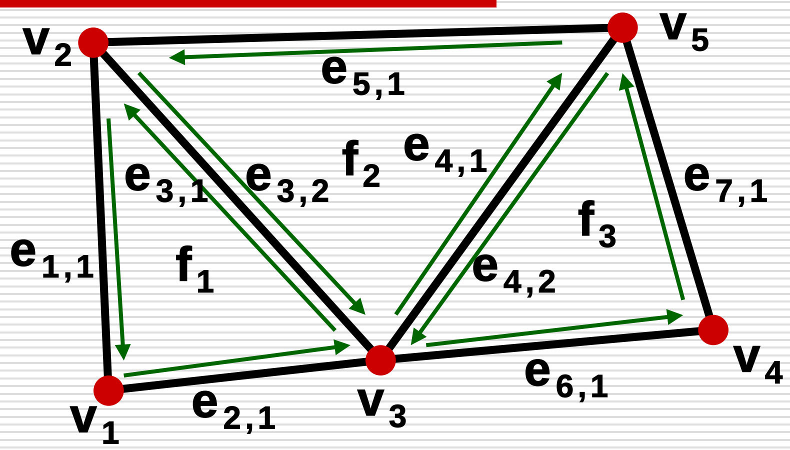


vertex	coordinate	IncidentEdge
v_1	(x_1,y_1,z_1)	$e_{2,1}$
v_2	(x_2,y_2,z_2)	$e_{1,1}$
v_3	(x_3,y_3,z_3)	$e_{4,1}$
v_4	(x_4,y_4,z_4)	$e_{7,1}$
v_5	(x_5,y_5,z_5)	$e_{5,1}$

face	edge
f_1	$e_{1,1}$
f_2	$e_{3,2}$
f_3	$e_{4,2}$

3. DCEL

- Example



Half-edge	origin	twin	Incident Face	next	prev
$e_{3,1}$	v_3	$e_{3,2}$	f_1	$e_{1,1}$	$e_{2,1}$
$e_{3,2}$	v_2	$e_{3,1}$	f_2	$e_{4,1}$	$e_{5,1}$
$e_{4,1}$	v_3	$e_{4,2}$	f_2	$e_{5,1}$	$e_{3,2}$
$e_{4,2}$	v_5	$e_{4,1}$	f_3	$e_{6,1}$	$e_{7,1}$

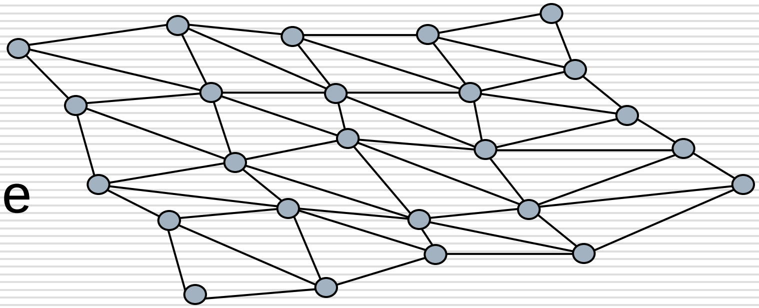
3. DCEL

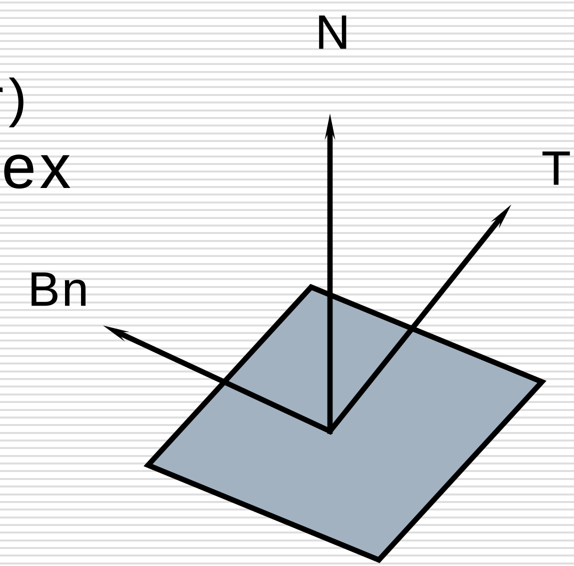
- Pros:
  - All queries in O(1) time
  - All operations are (usually) O(1)

- Cons:
  - Represents only manifold meshes

Geometry Data

- Vertex position
  - (x, y, z, w)
  - in model space or screen space
- Vertex normal
  - (n_x, n_y, n_z)
- Vertex color
  - (r, g, b) or (diffuse, specular)
- Texture coordinates on vertex
  - $(u_1, v_1), (u_2, v_2), \ldots$
- Skin weights
  - $(bone_1, w_1, bone_2, w_2, \ldots)$
- Tangent and bi-normal

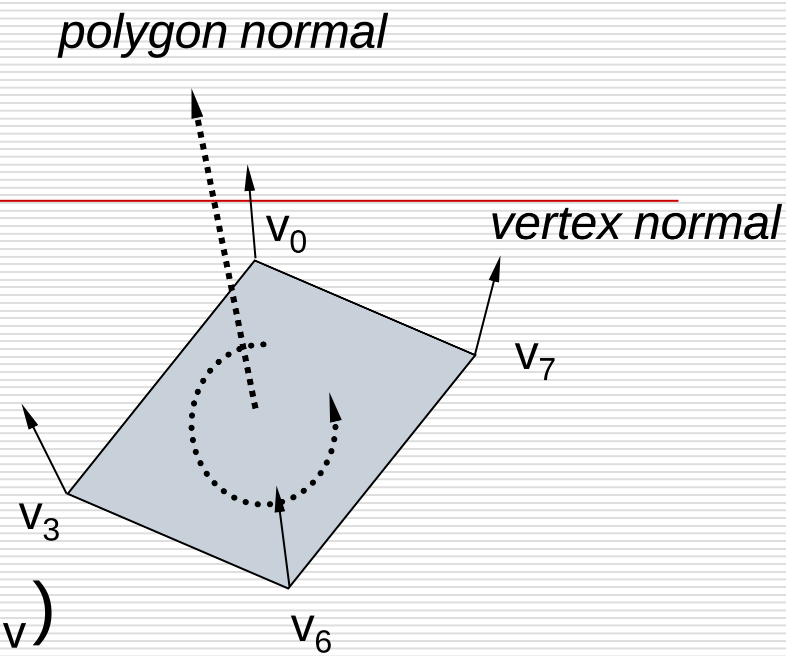




Topology Data

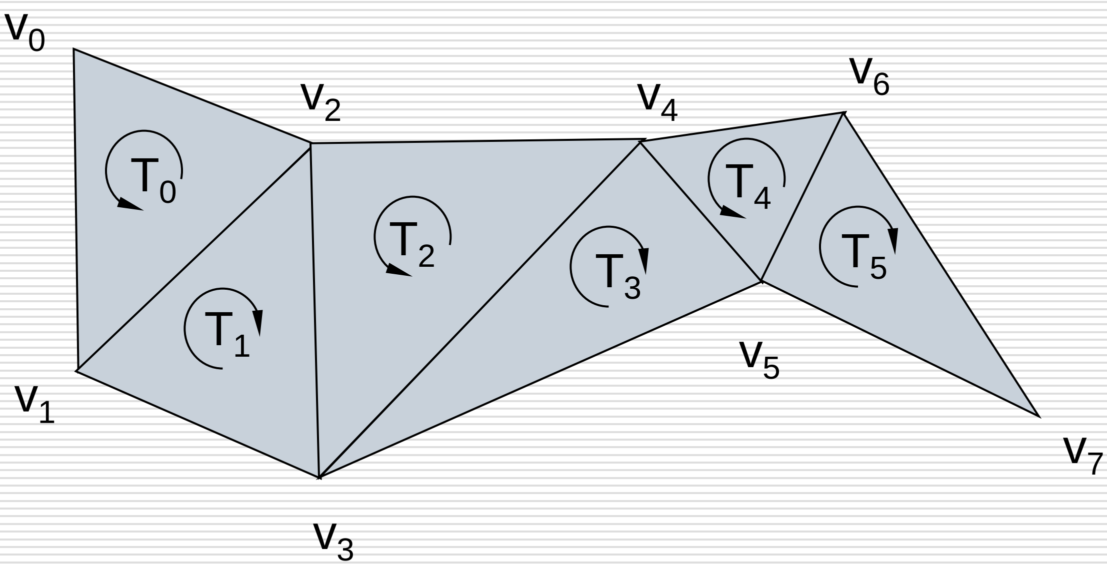
- Lines
  - Line segments
  - Polyline
    - Open / closed
- Indexed triangles
- Triangle strips/fans
- Surfaces
  - Non-Uniform Rational B-Spline (NURBS)
- Subdivision

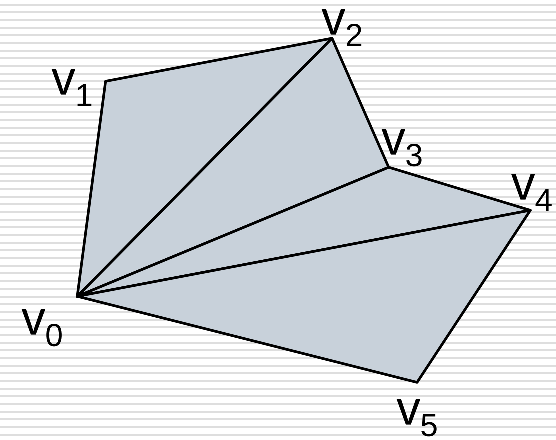
Indexed Triangles



- Geometric data
  - Vertex data
  - v_0, v_1, v_2, v_3, ...
  - $(x, y, z, n_x, n_y, n_z, t_u, t_v)$
  - or $(x, y, z, c_r, c_g, c_b, t_u, t_v, ...)$
- Topology
  - Face v_0 v_3 v_6 v_7
    - *right-hand rule for index*
  - Edge table

Triangle Strips/Fans





$v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$

$v_0, v_1, v_2, v_3, v_4, v_5$

Get great performance to use triangle strips/fans for rendering on current hardware

Surface Properties

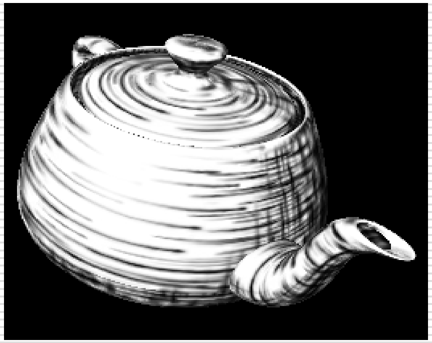
- Material
- Textures
- Shaders

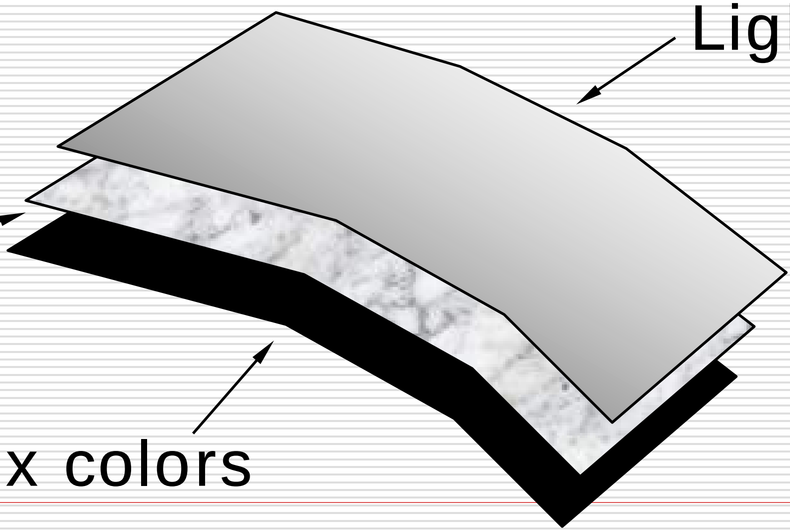
Materials

- Material
  - Ambient
    - Environment
    - Non-lighted area
  - Diffuse
    - Dynamic lighting
  - Emissive
    - Self-lighting
  - Specular with shineness
    - Hi-light
    - View-dependent
    - Not good for hardware rendering
- Local illumination

Textures

- Textures
  - Single texture
  - Texture coordinate animation
  - Texture animation
  - Multiple textures
  - Alphamap

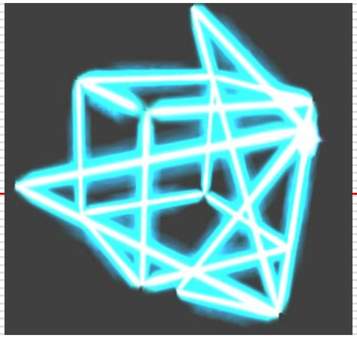




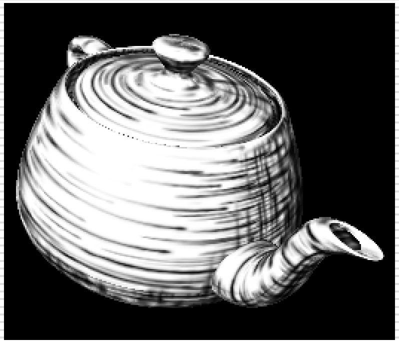
Shaders

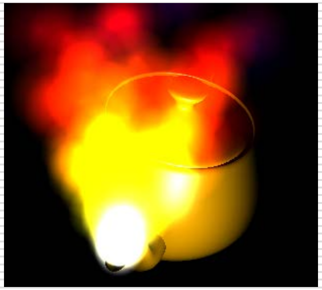
- Programmable shading language
  - Vertex shader
  - Pixel shader
- Procedural way to implement some process of rendering
  - Transformation
  - Lighting
  - Texturing
  - BRDF
  - Rasterization
  - Pixel fill-in
  - Post-processing for rendering

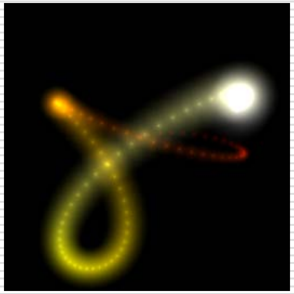
Powered by Shaders

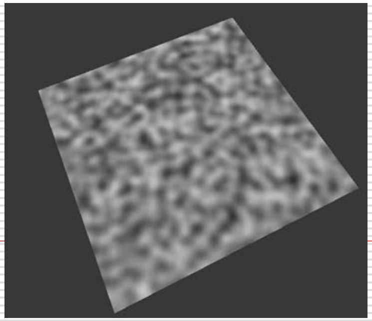


- Per-pixel lighting
- Motion blur
- Volume / Height fog
- Volume lines
- Depth of field
- Fur rendering
- Reflection / Refraction
- NPR
- Shadow
- Linear algebra operators
- Perlin noise
- Quaternion
- Sparse matrix solvers
- Skin bone deformation
- Normal map
- Displacement map
- Particle shader
- Procedural Morphing
- Water Simulation



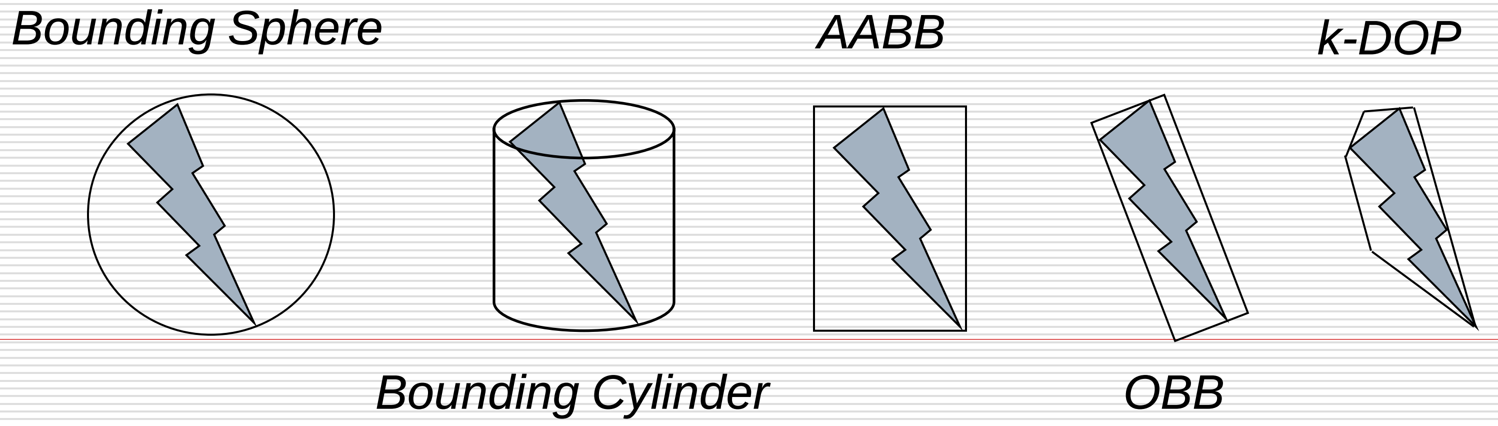






Bounding Volumes

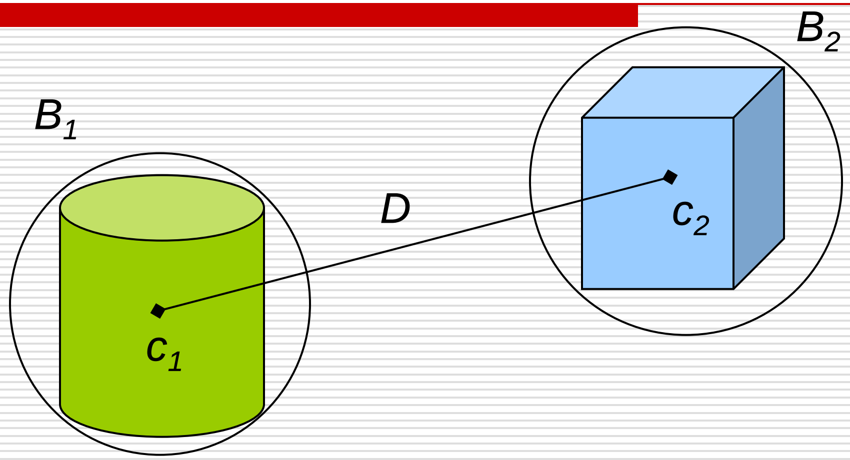
- Bounding sphere
- Bounding cylinder
- Axis-aligned bounding box (AABB)
- Oriented bounding box (OBB)
- Discrete oriented polytope (k-DOP)



Bounding Volume - Application

- Collision detection
- Visibility culling
- Hit test
- Steering behavior
  - in “Game AI”

Application Example – Bounding Sphere



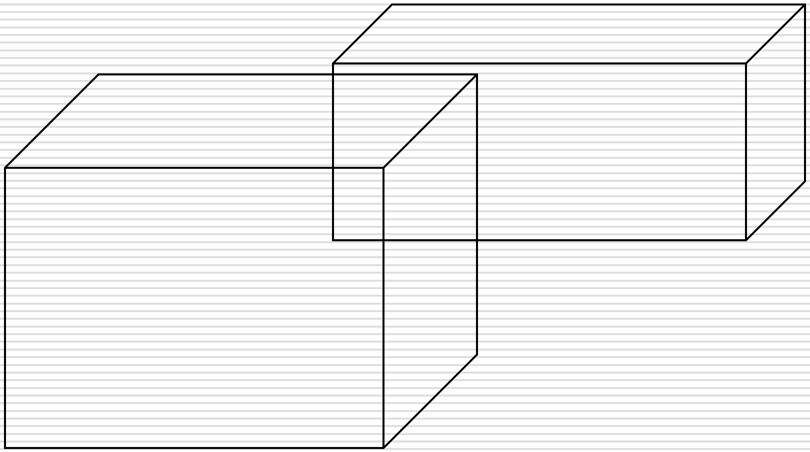
Bounding sphere $B_1(c_1, r_1)$, $B_2(c_2, r_2)$

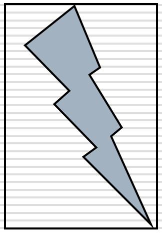
If the distance between two bounding spheres is larger than the sum of radius of the spheres, than these two objects have no chance to collide.

$D > Sum(r_1, r_2)$

Application Example - AABB

- Axis-aligned bounding box (AABB)
  - Simplified calculation using axis-alignment feature
  - But need run-timely to track the bounding box

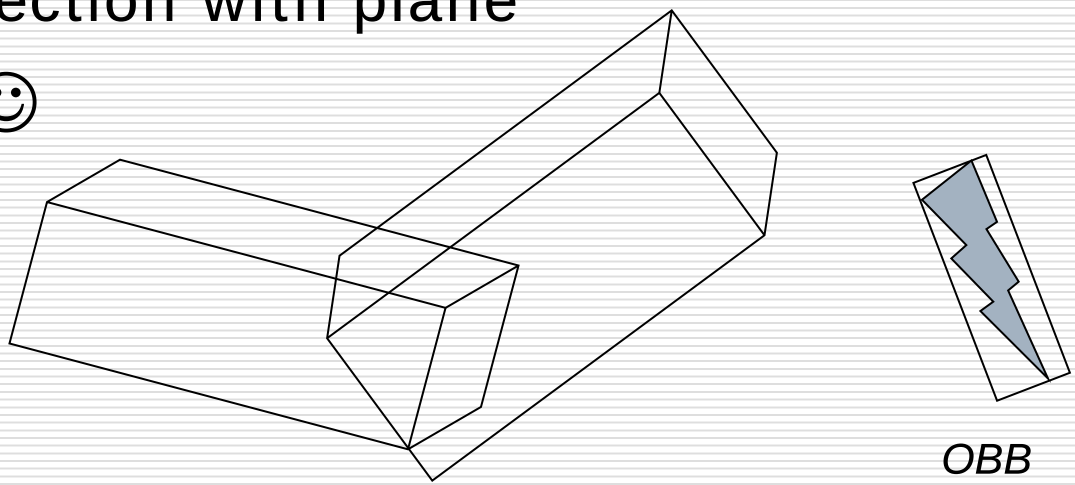




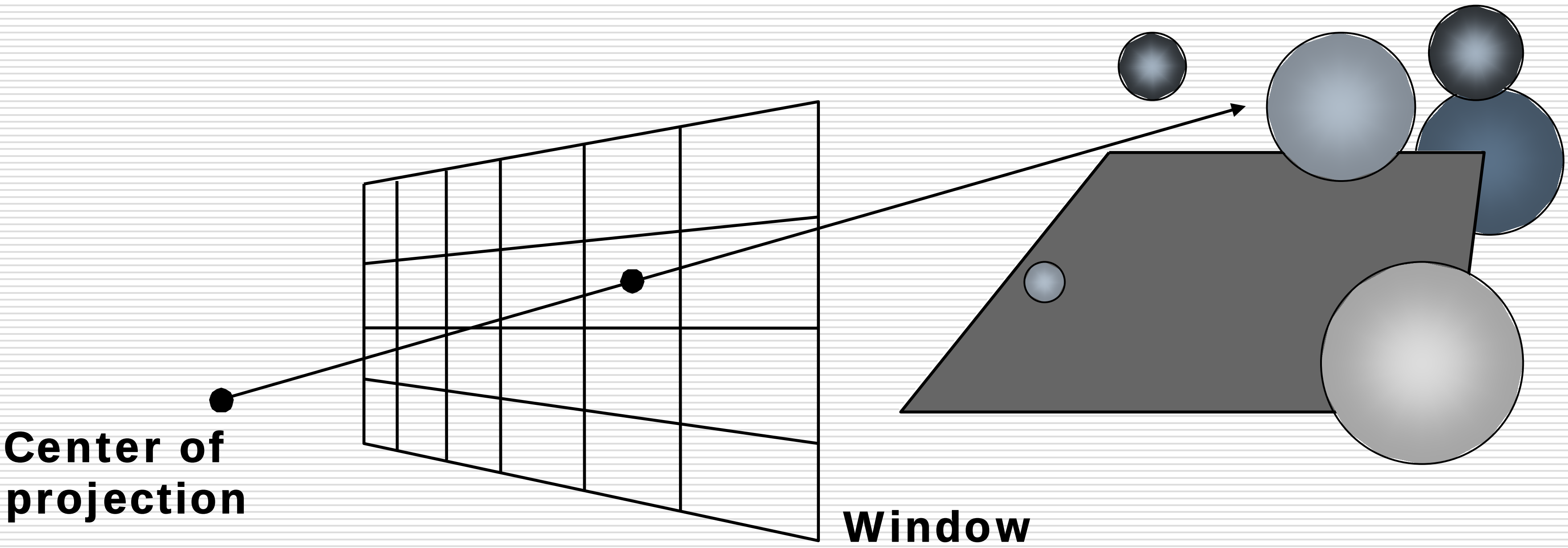
AABB

Application Example - OBB

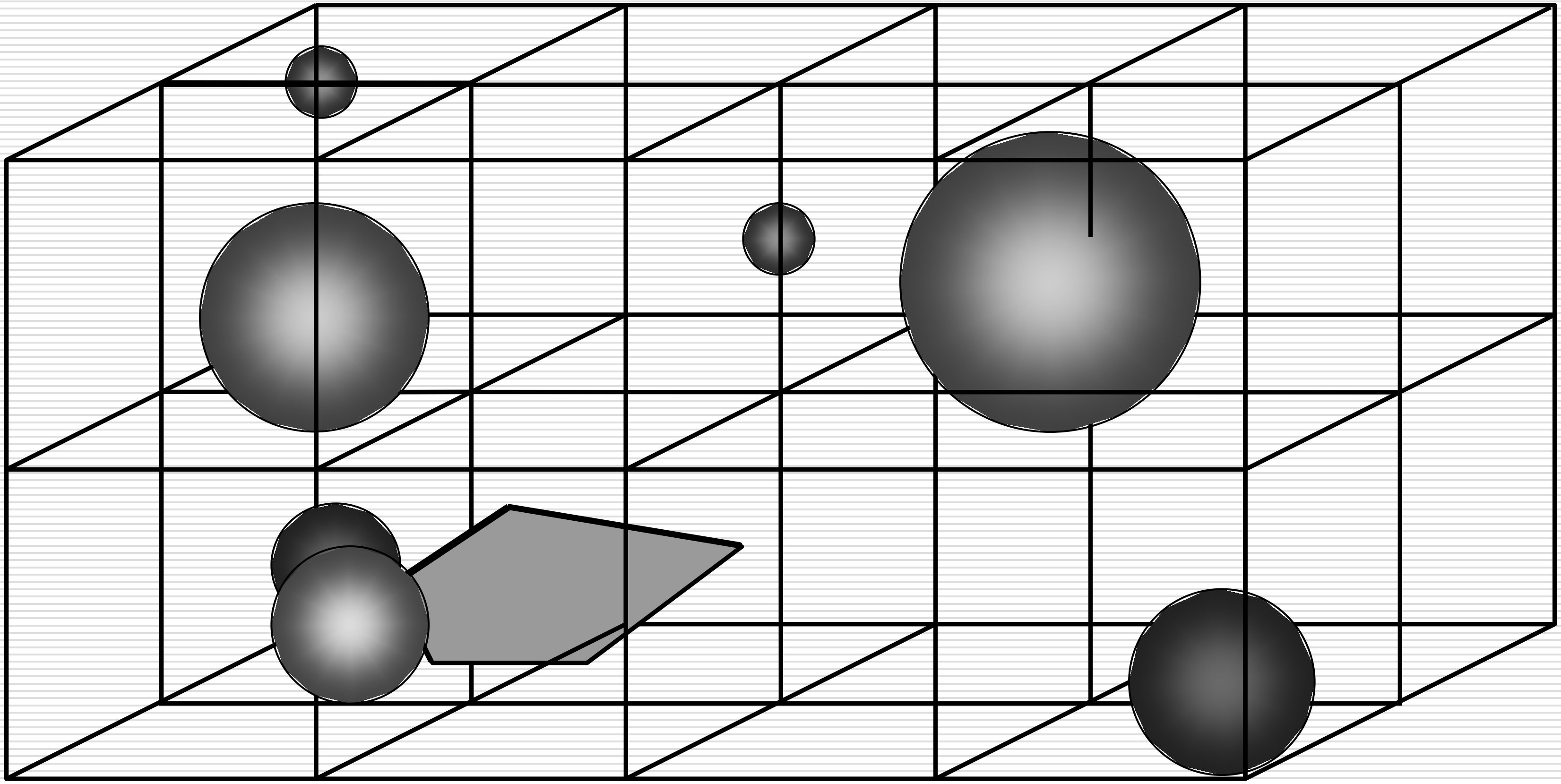
- Oriented bounding box (OBB)
  - Need intersection calculation using the transformed OBB geometric data
    - 3D containment test
    - Line intersection with plane
- For games, ☺



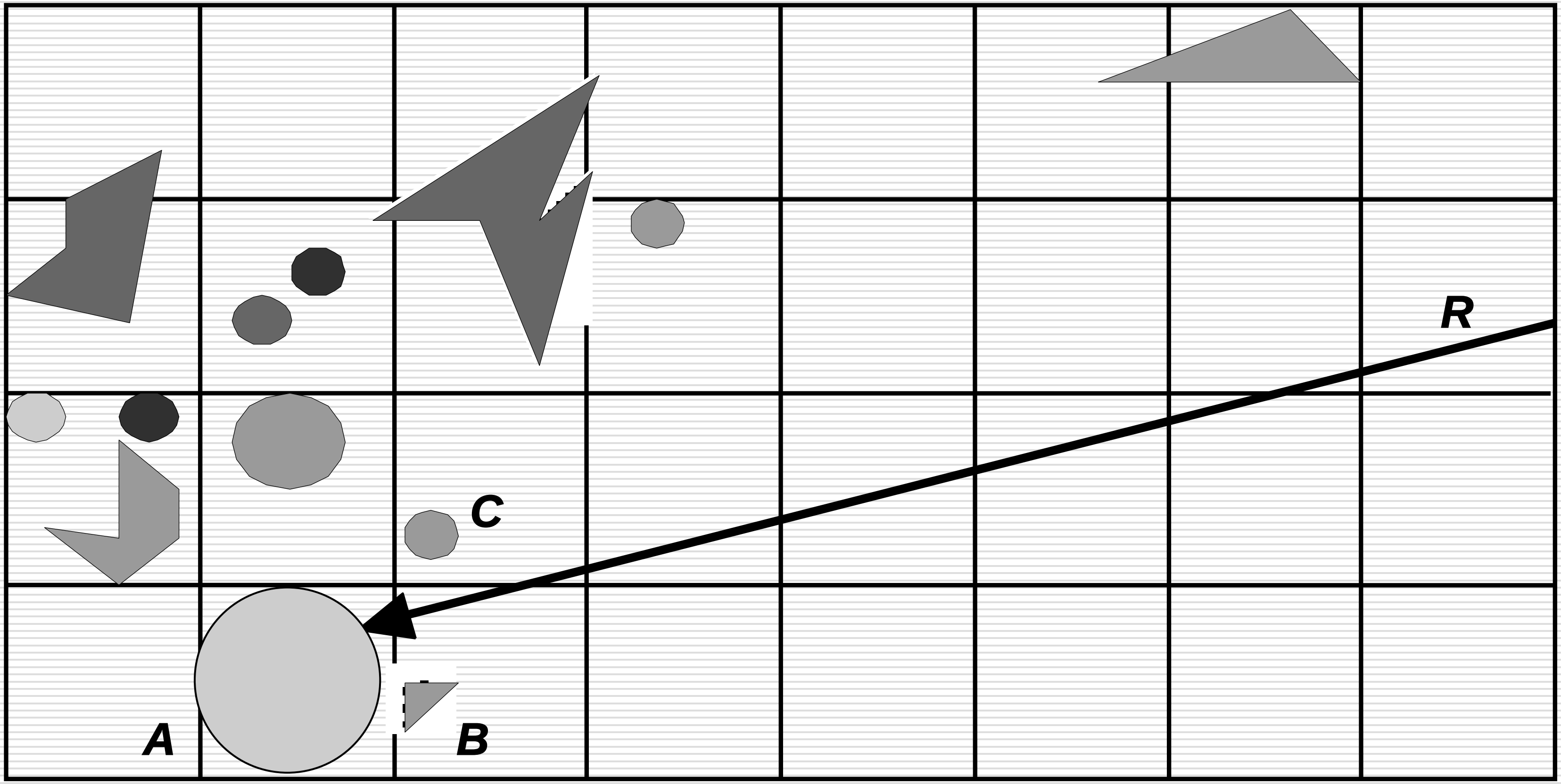
Ray Casting



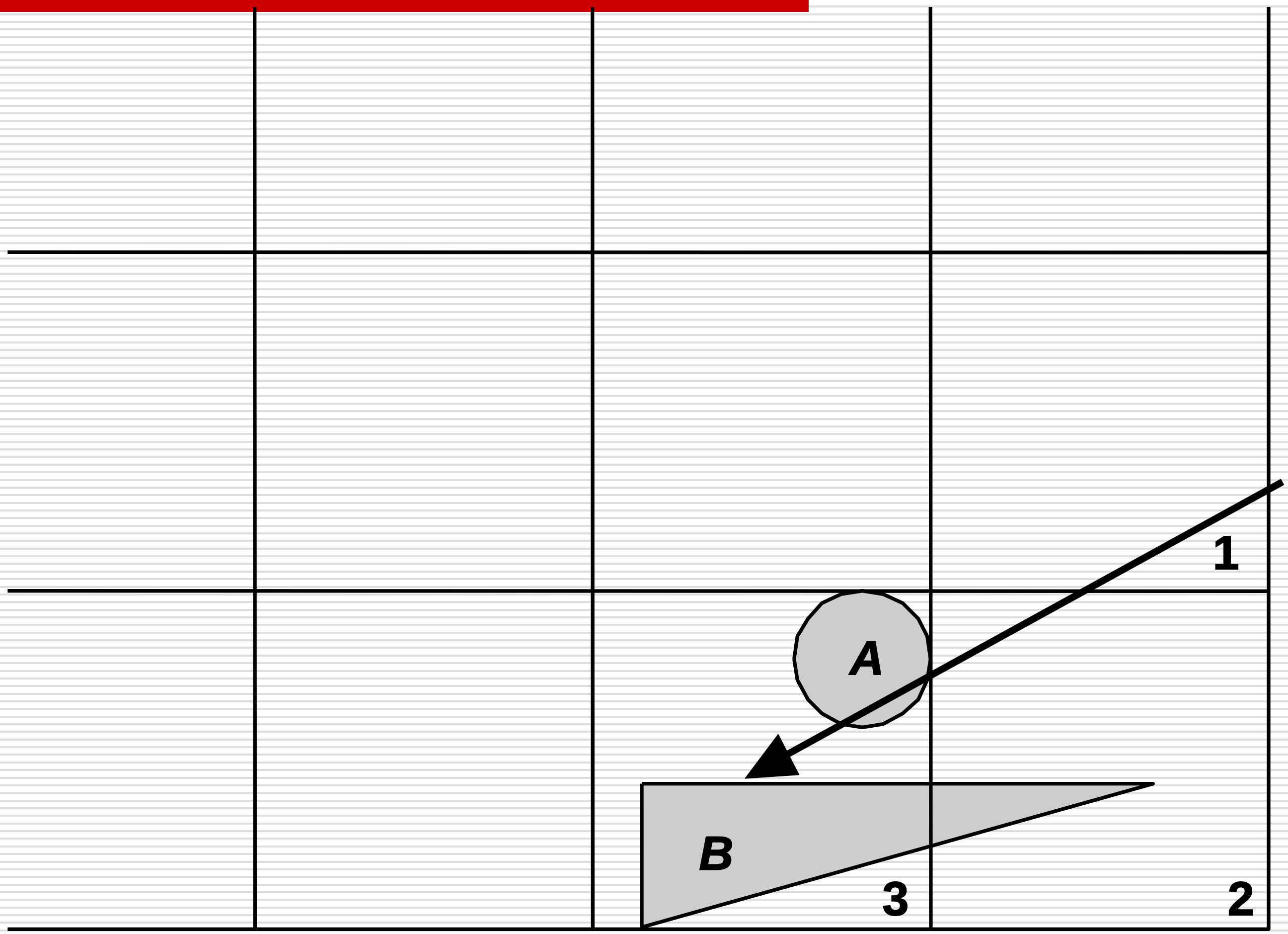
Spatial Partitioning



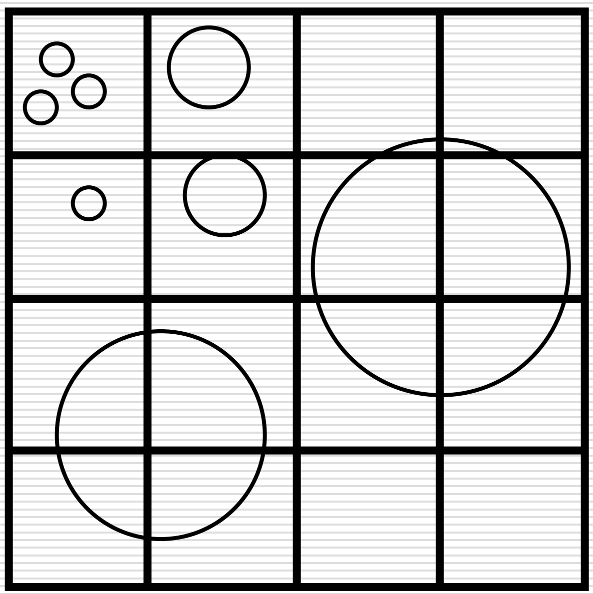
Spatial Partitioning



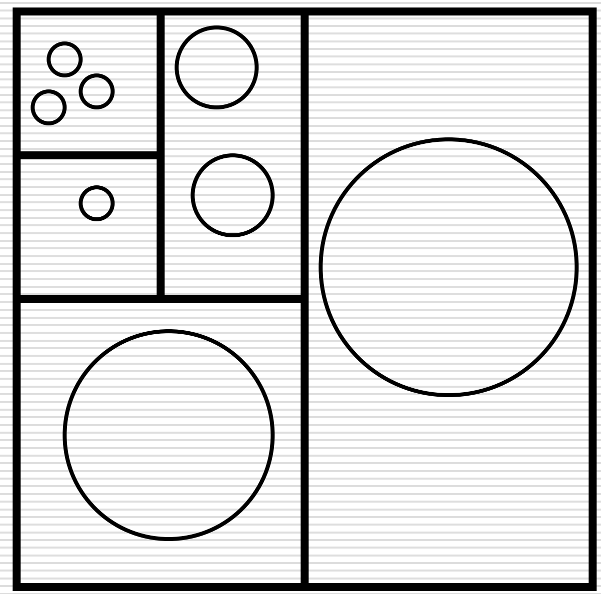
Spatial Partitioning



Space Subdivision Approaches

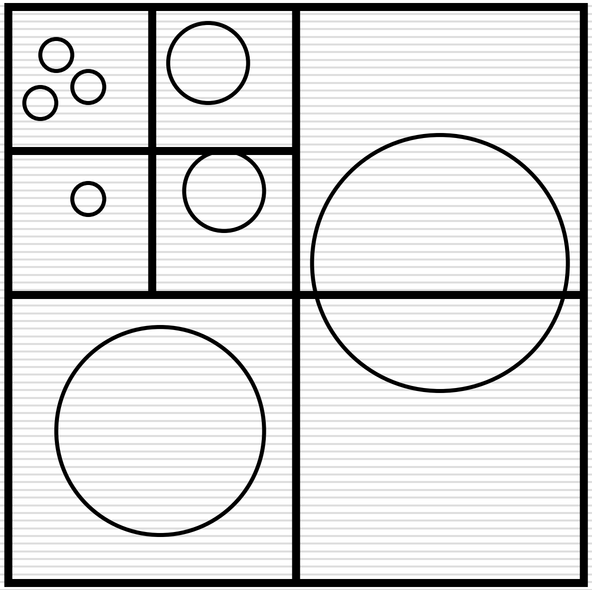


Uniform grid

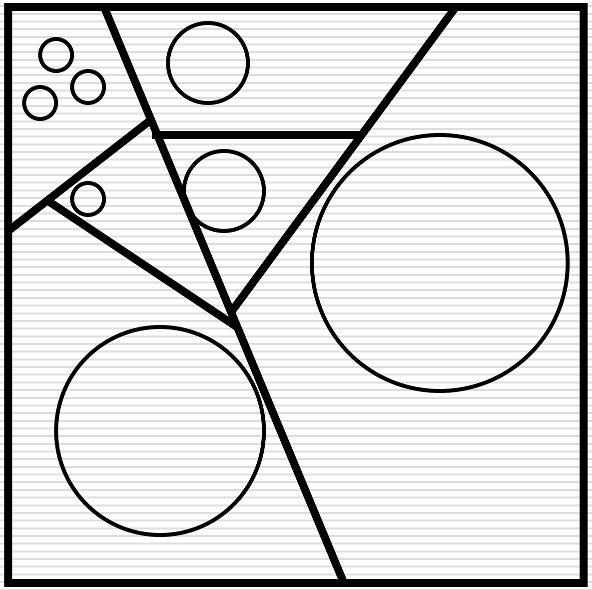


K-d tree

Space Subdivision Approaches

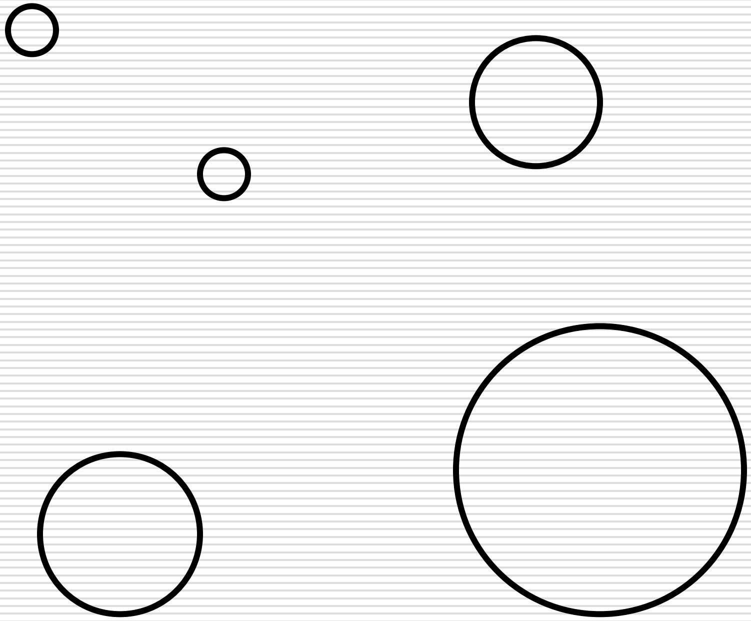


Quadtree (2D)
Octree (3D)

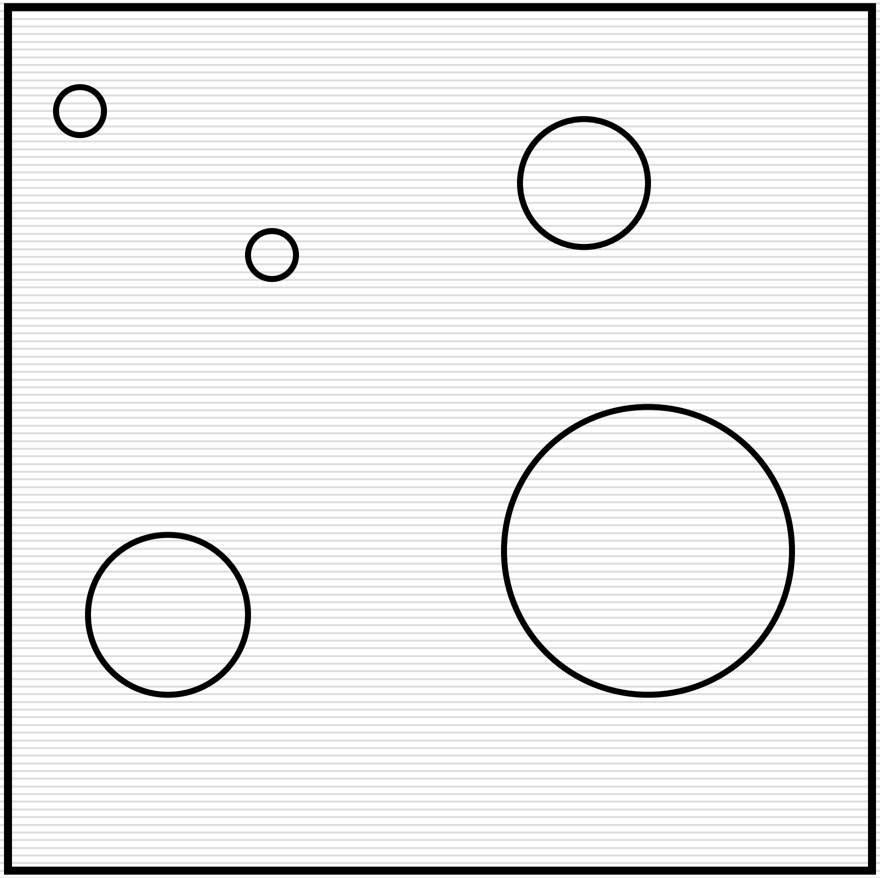


BSP tree

Uniform Grid



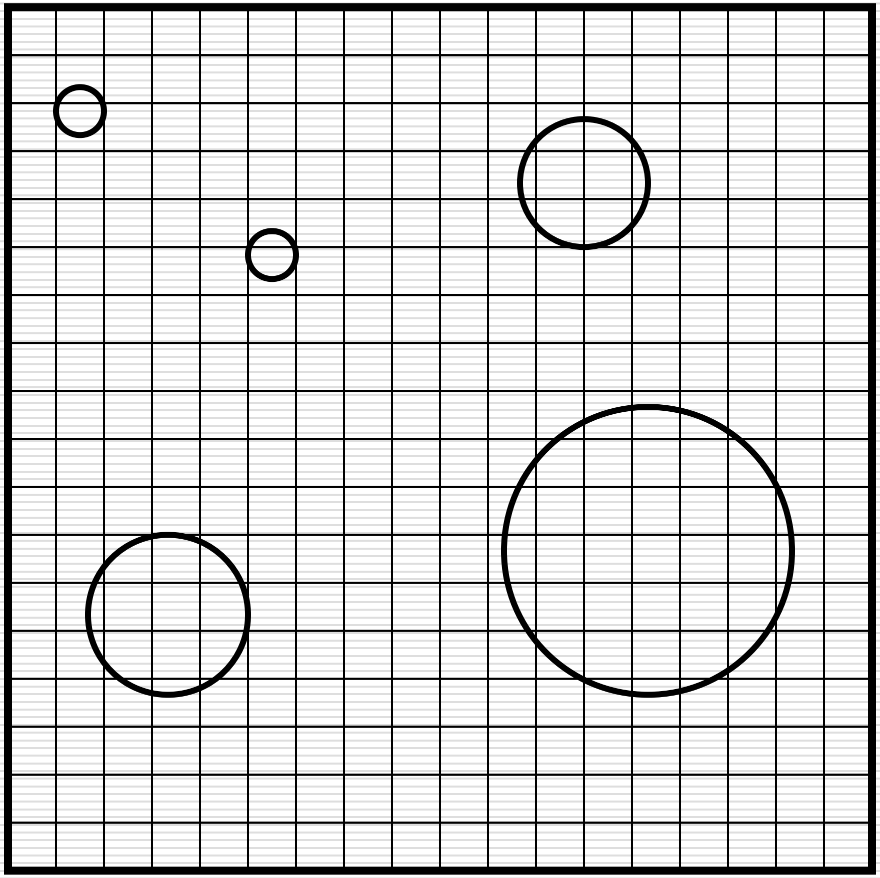
Uniform Grid



Preprocess scene

1. Find bounding box

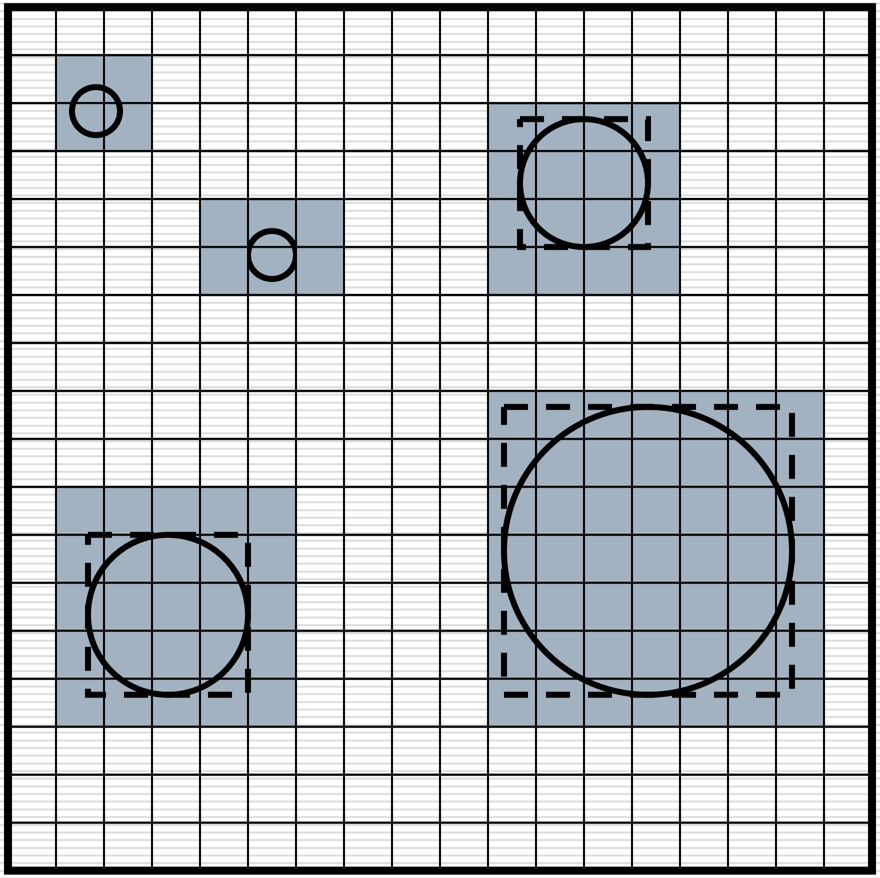
Uniform Grid



Preprocess scene

1. Find bounding box
2. Determine grid resolution

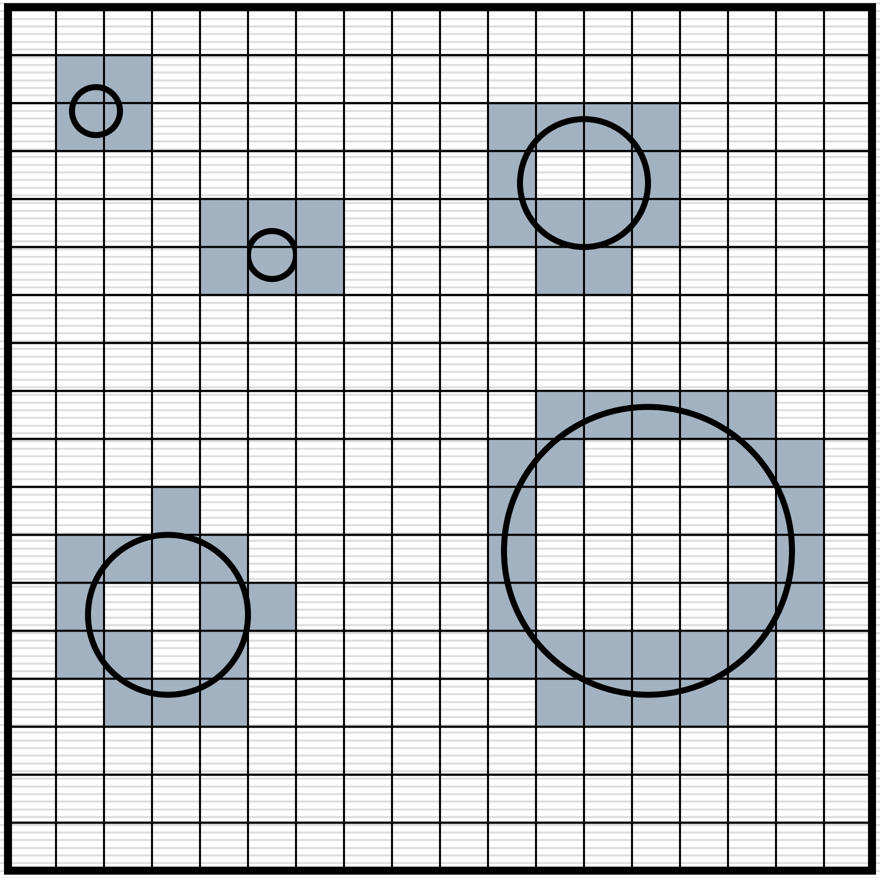
Uniform Grid



Preprocess scene

1. Find bounding box
2. Determine grid resolution
3. Place object in cell if its bounding box overlaps the cell

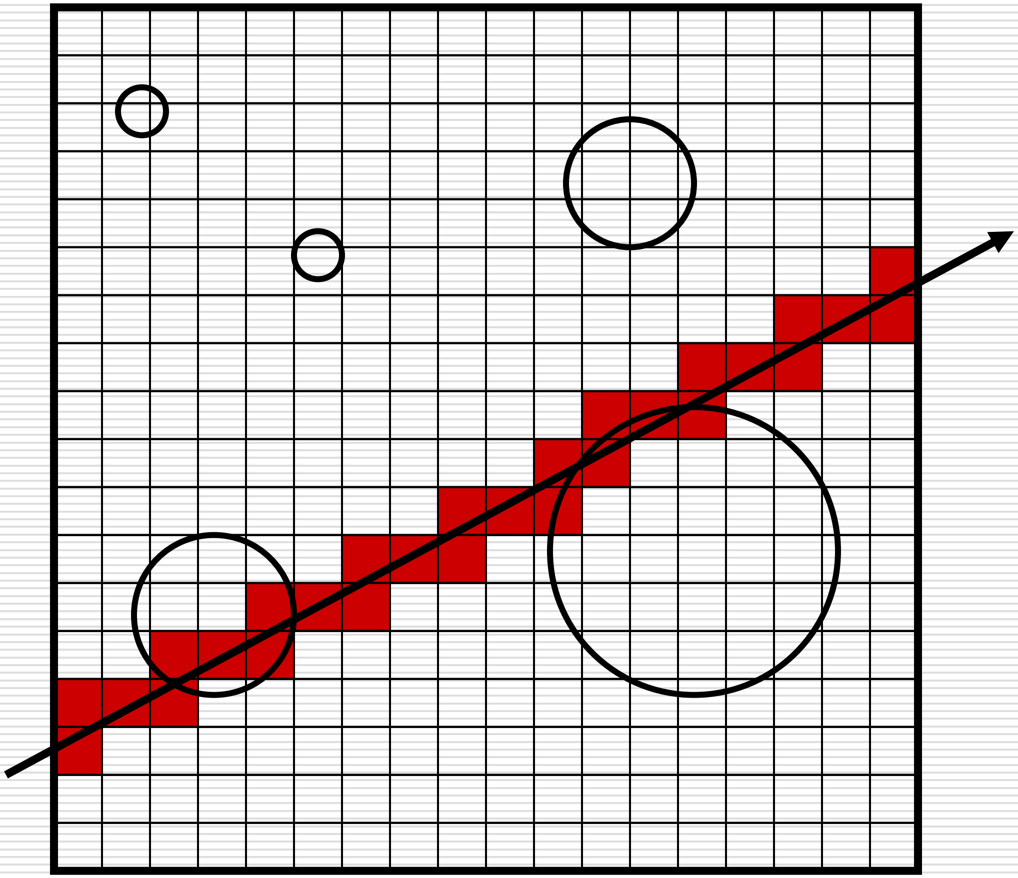
Uniform Grid



Preprocess scene

1. Find bounding box
2. Determine grid resolution
3. Place object in cell if its bounding box overlaps the cell
4. Check that object overlaps cell (expensive!)

Uniform Grid Traversal

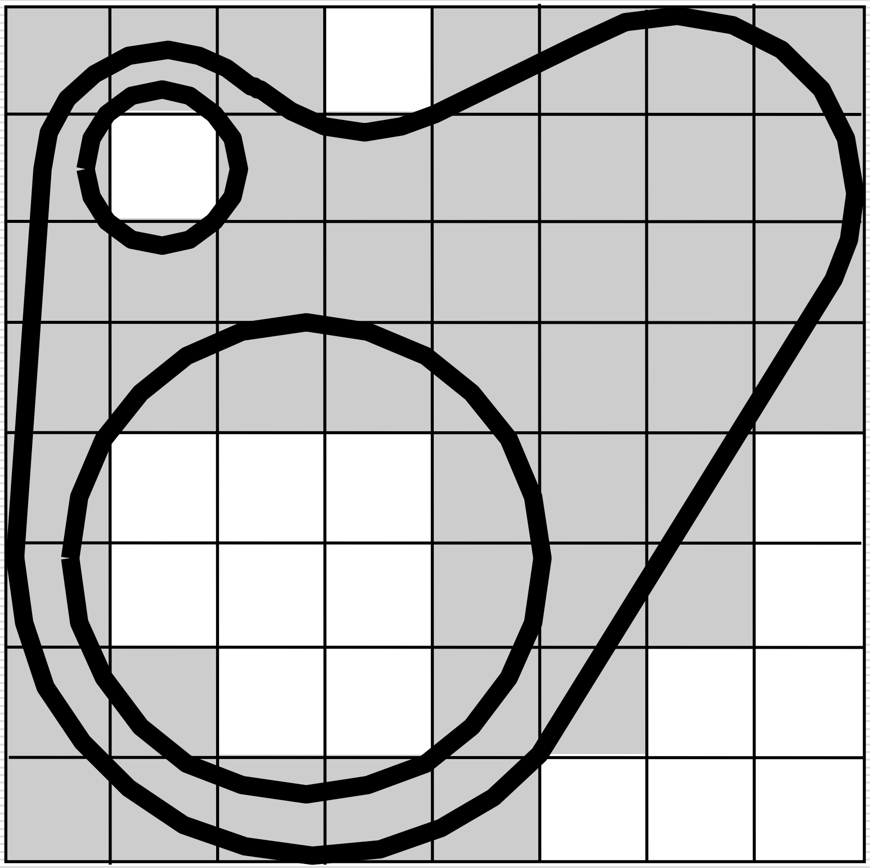


Preprocess scene

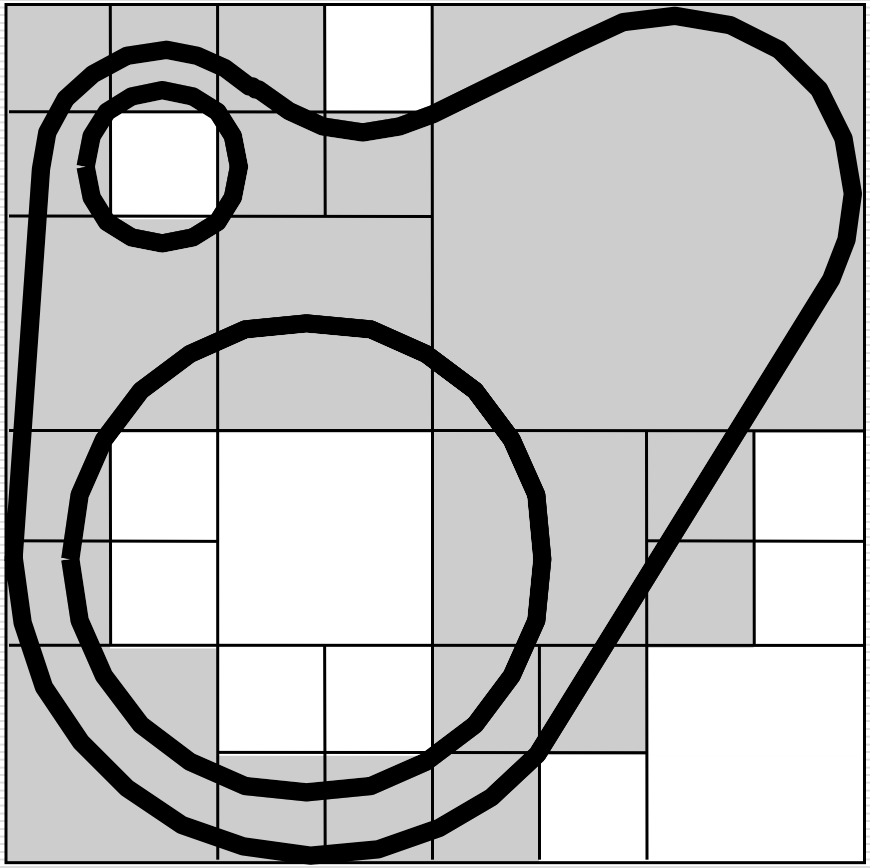
Traverse grid

3D line = 3D-DDA

From Uniform Grid to Quadtree

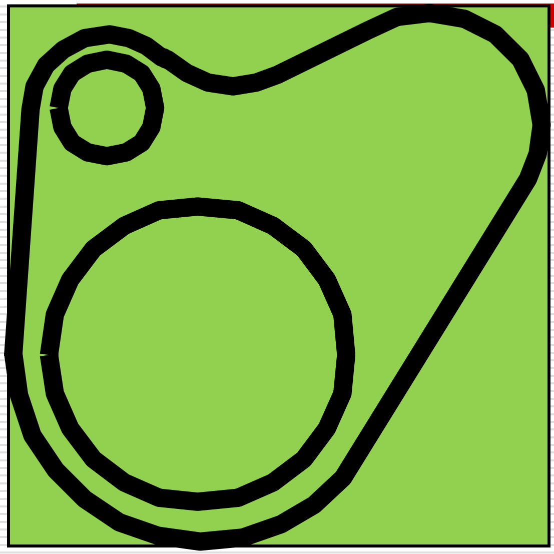


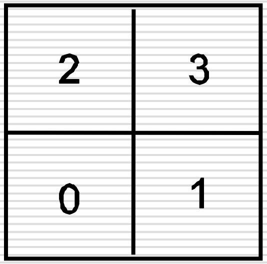
Quadtree (Octrees)



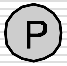
subdivide the space adaptively

Quadtree Data Structure

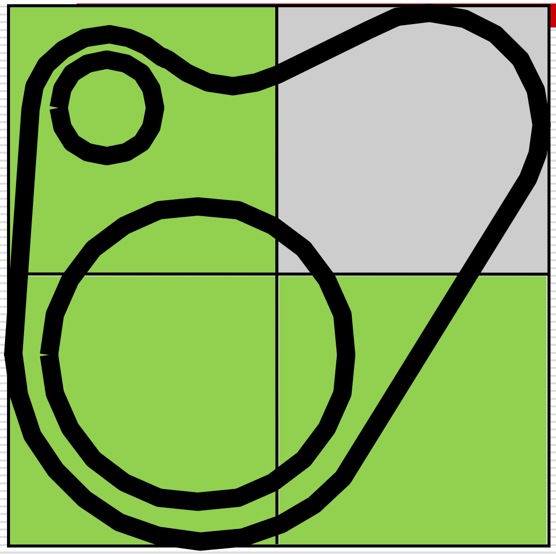


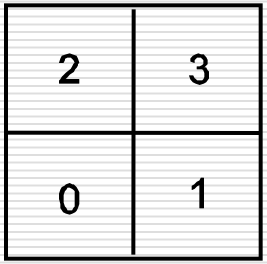


Quadrant Numbering

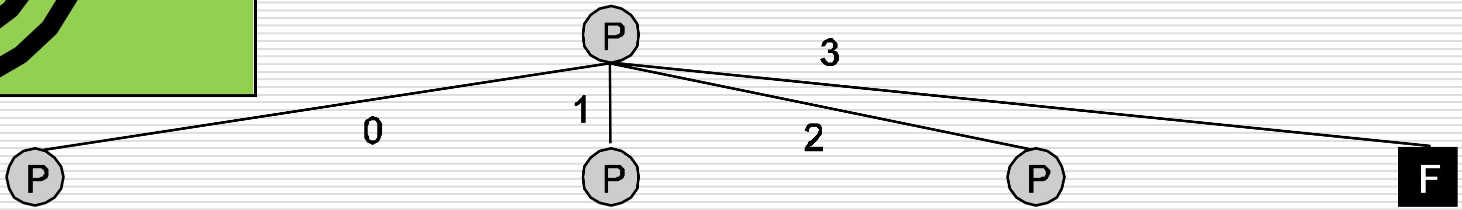


Quadtree Data Structure

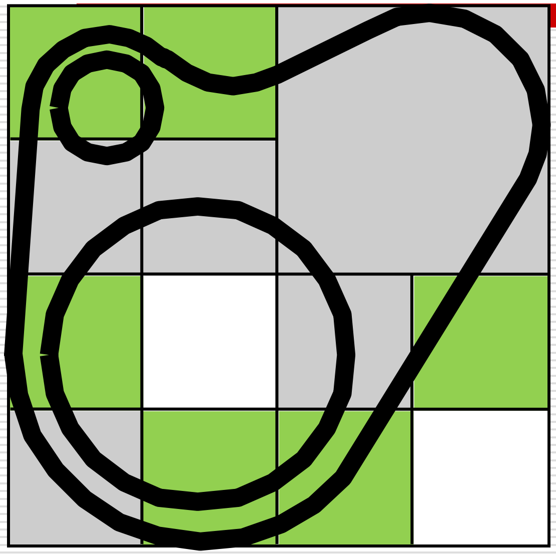


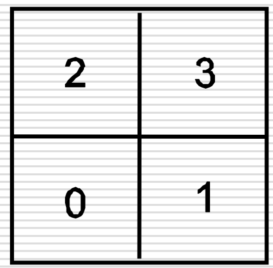


Quadrant Numbering

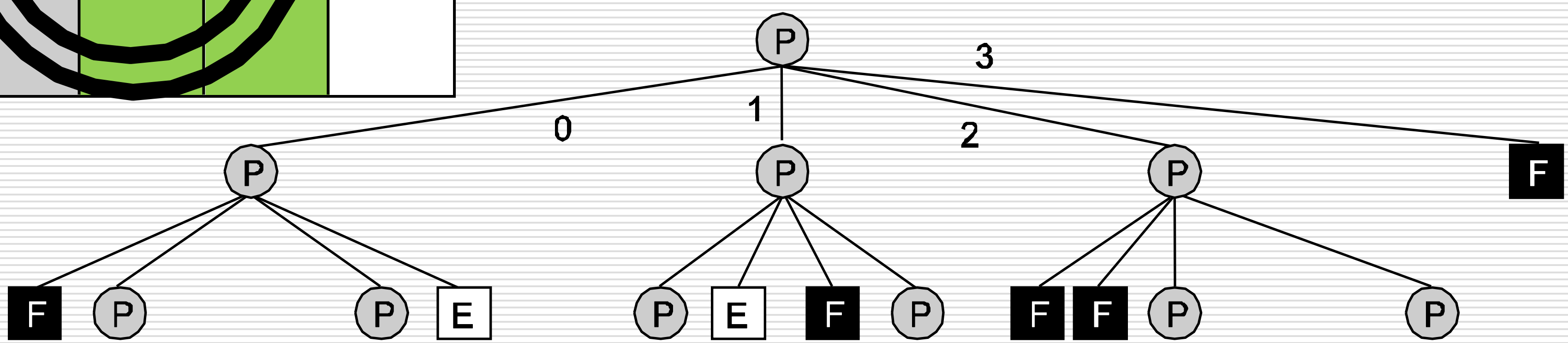


Quadtree Data Structure

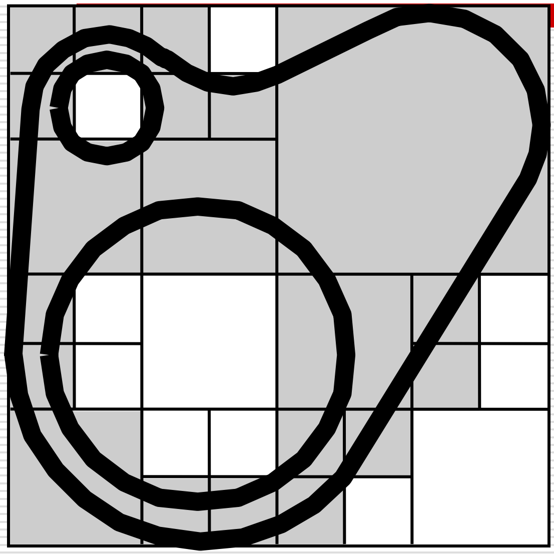


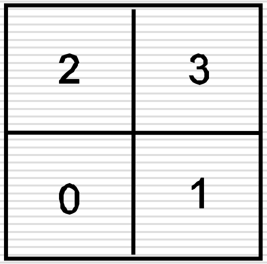


Quadrant Numbering

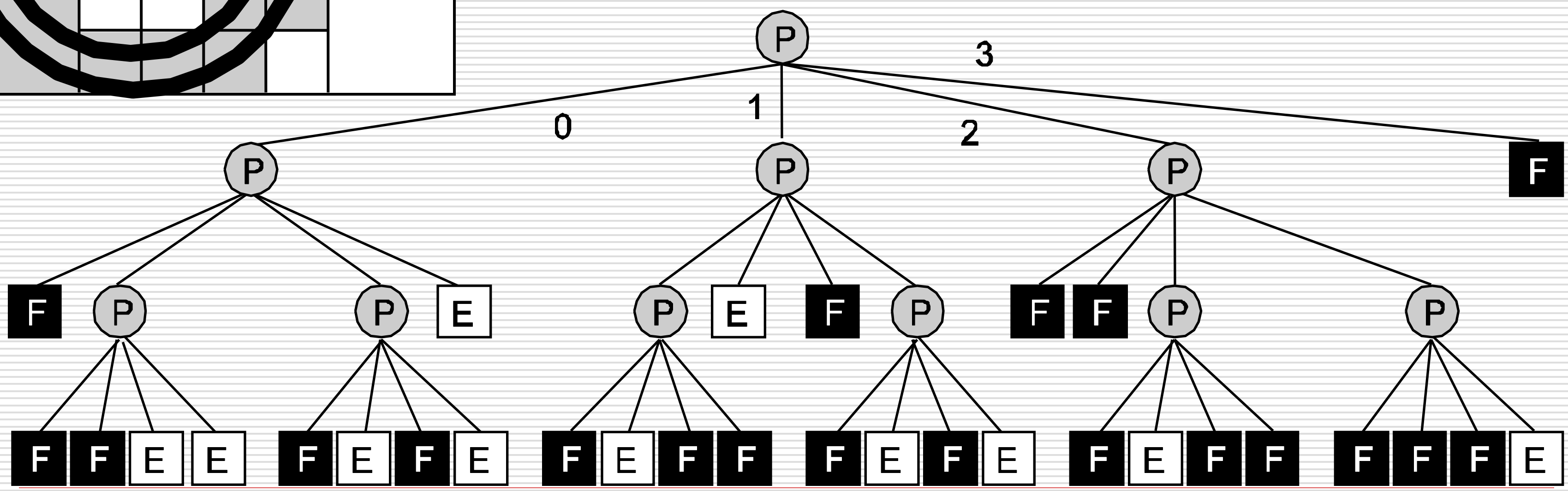


Quadtree Data Structure

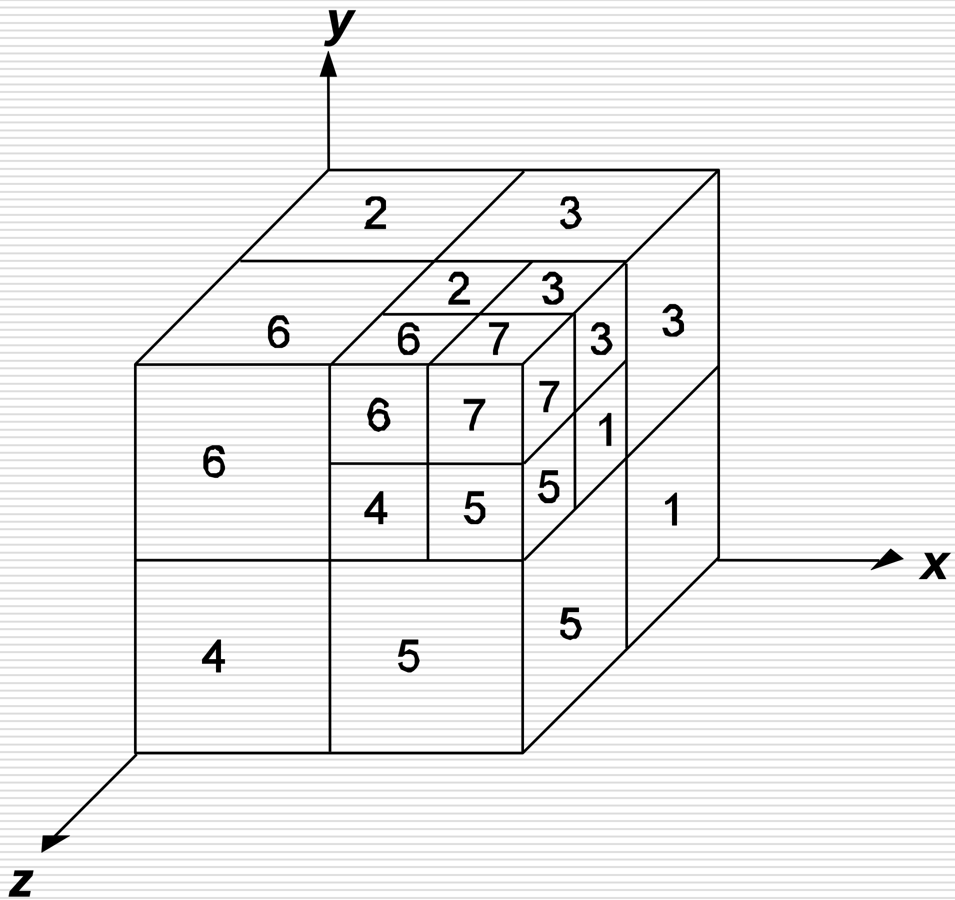




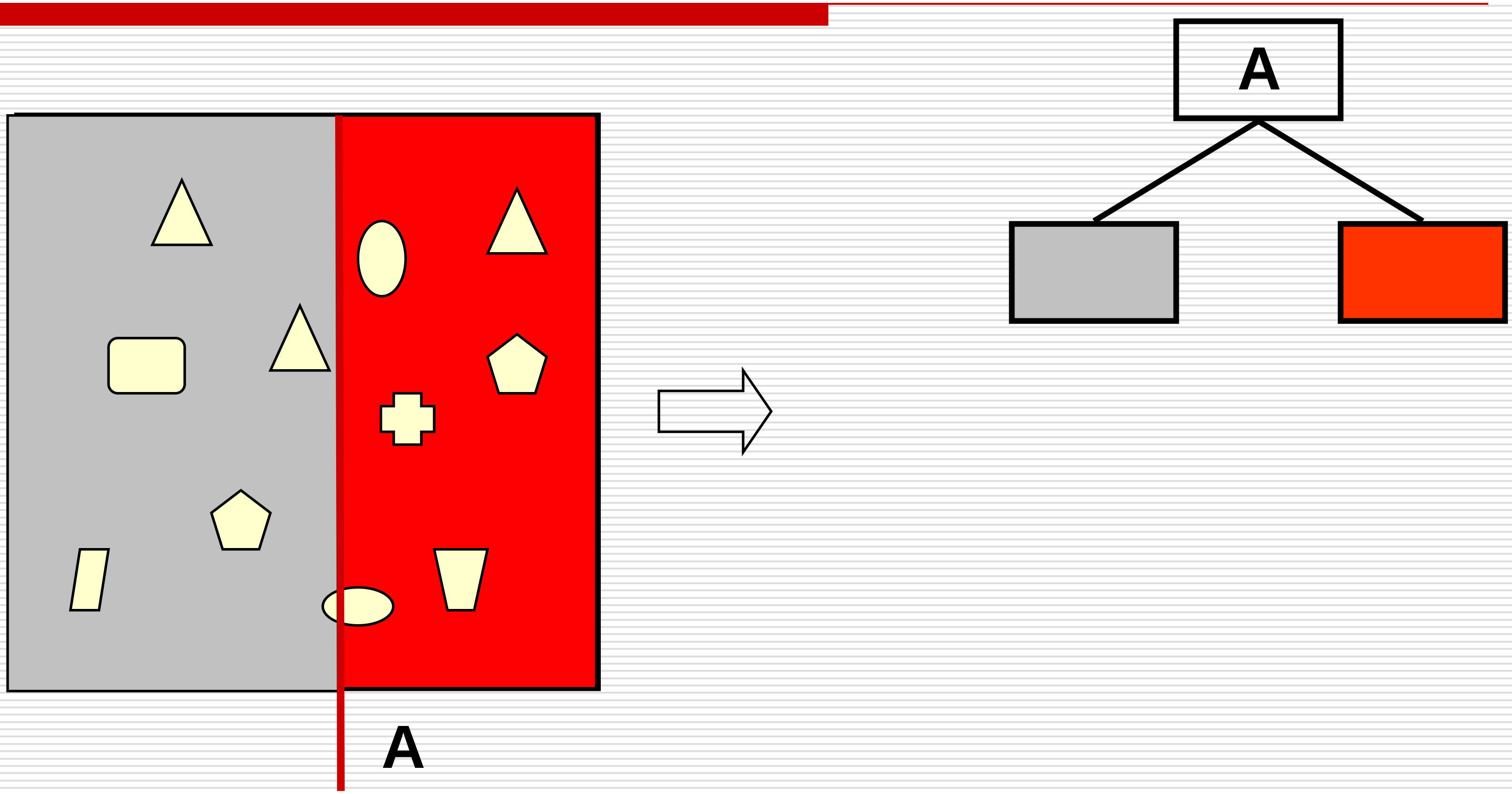
Quadrant Numbering



From Quadtree to Octree

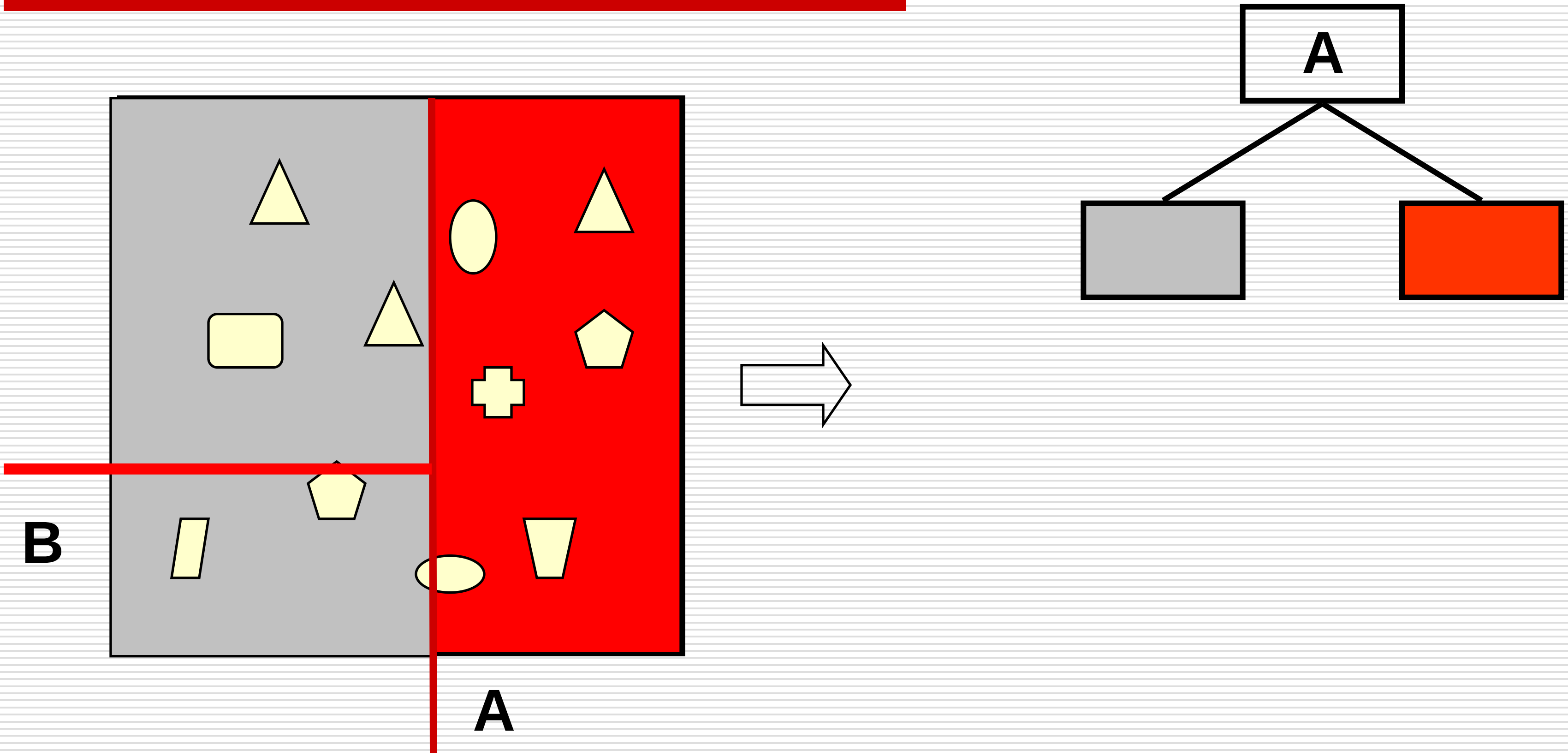


K-d Tree



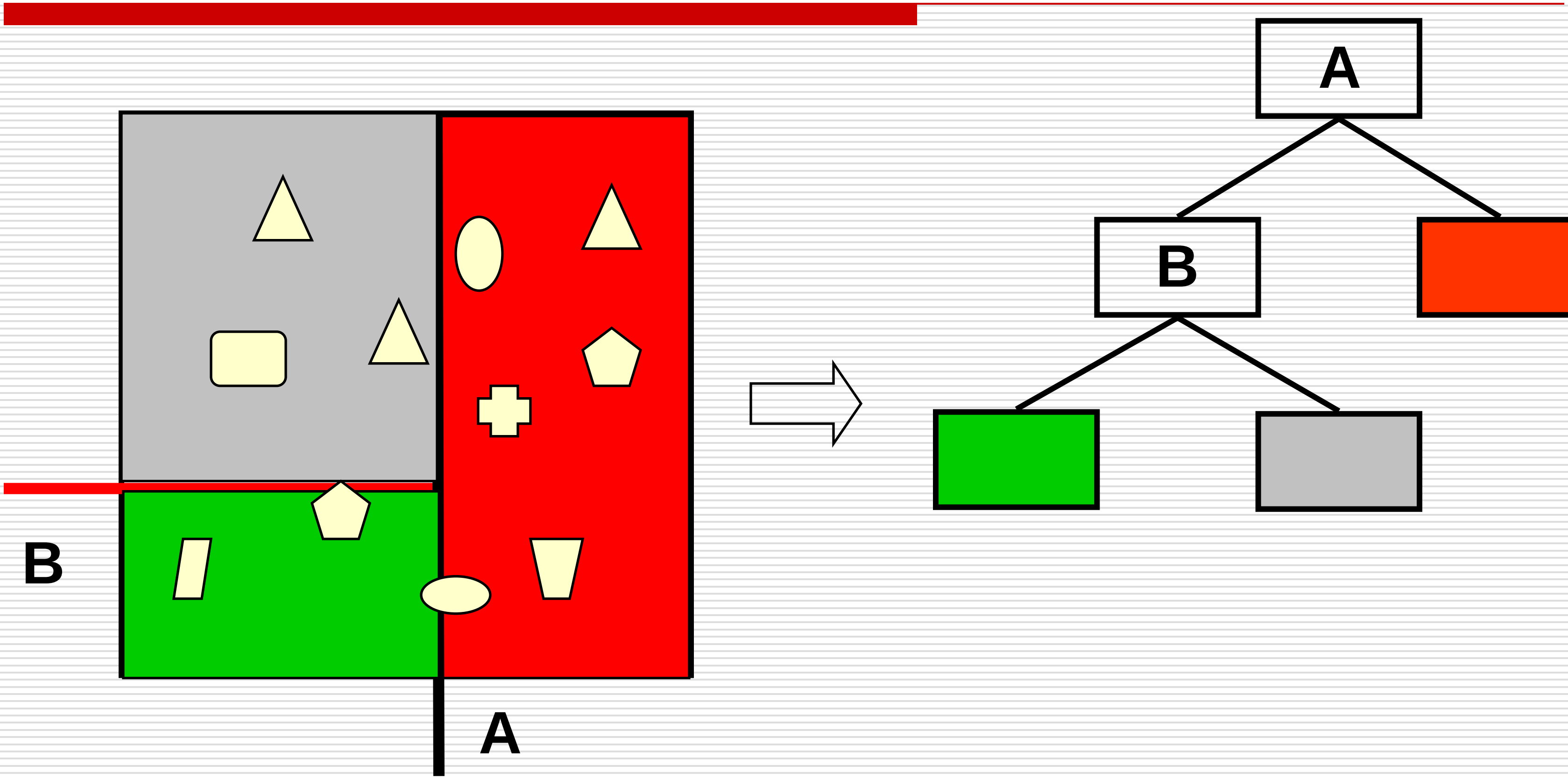
Leaf nodes correspond to unique regions in space

K-d Tree



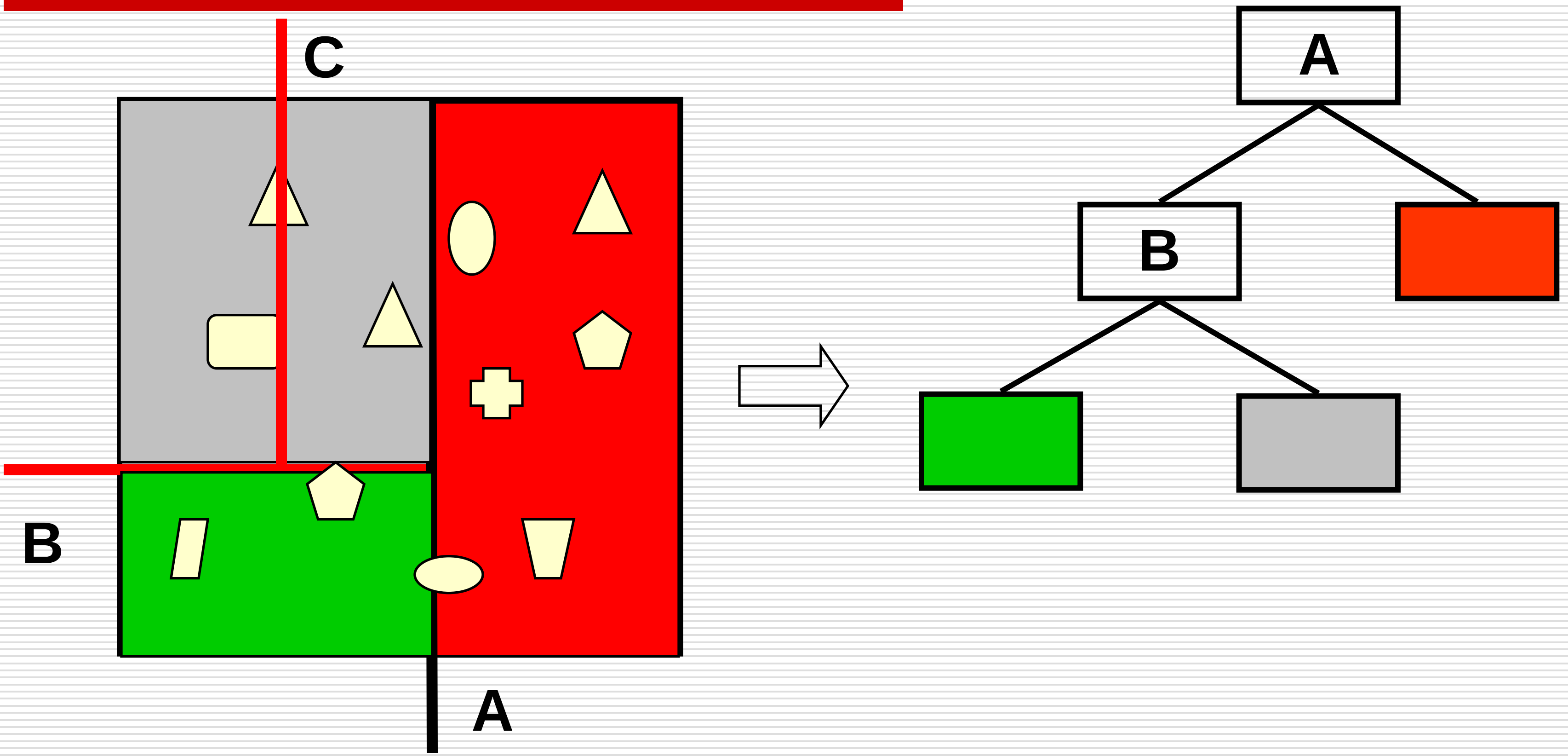
Leaf nodes correspond to unique regions in space

K-d Tree

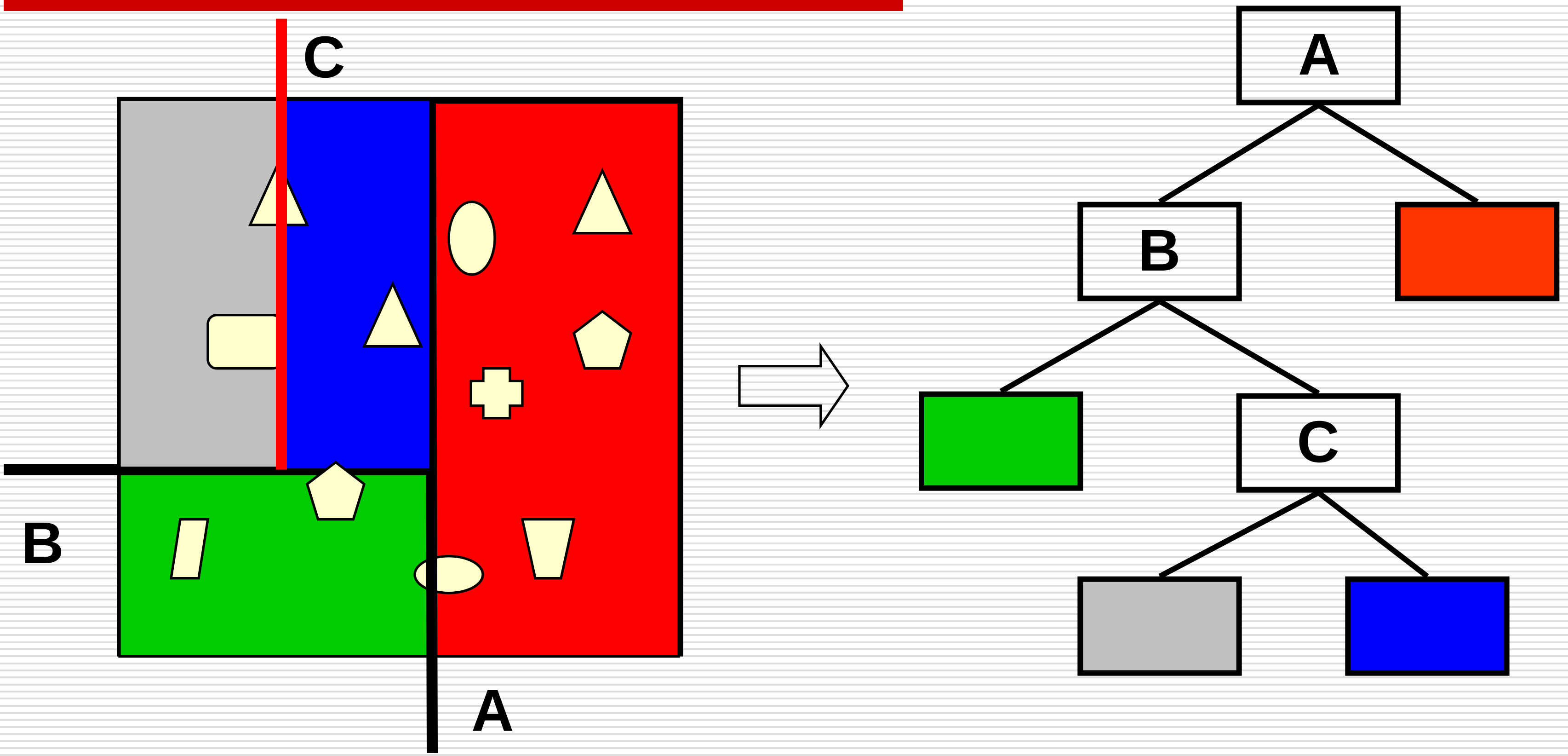


Leaf nodes correspond to unique regions in space

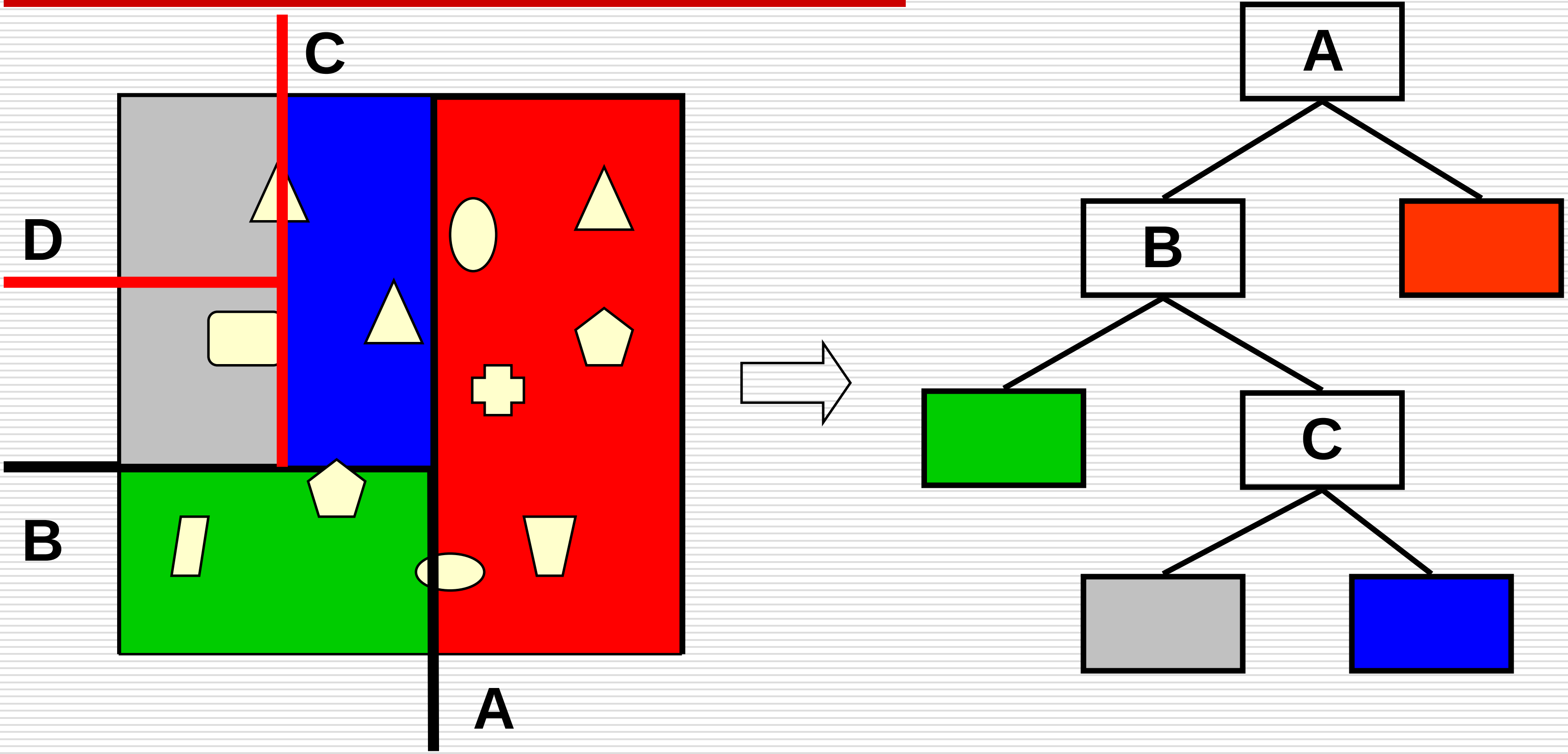
K-d Tree



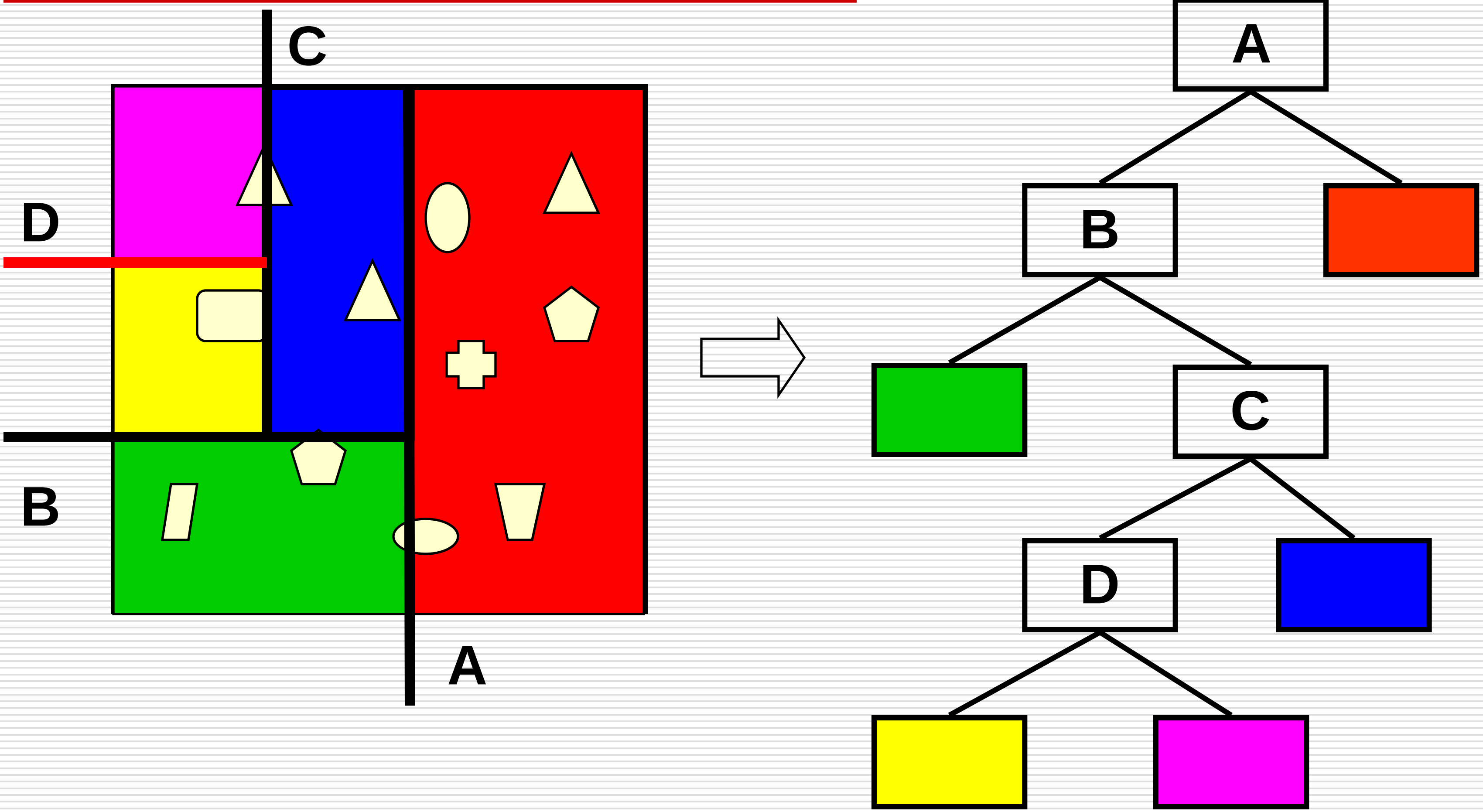
K-d Tree



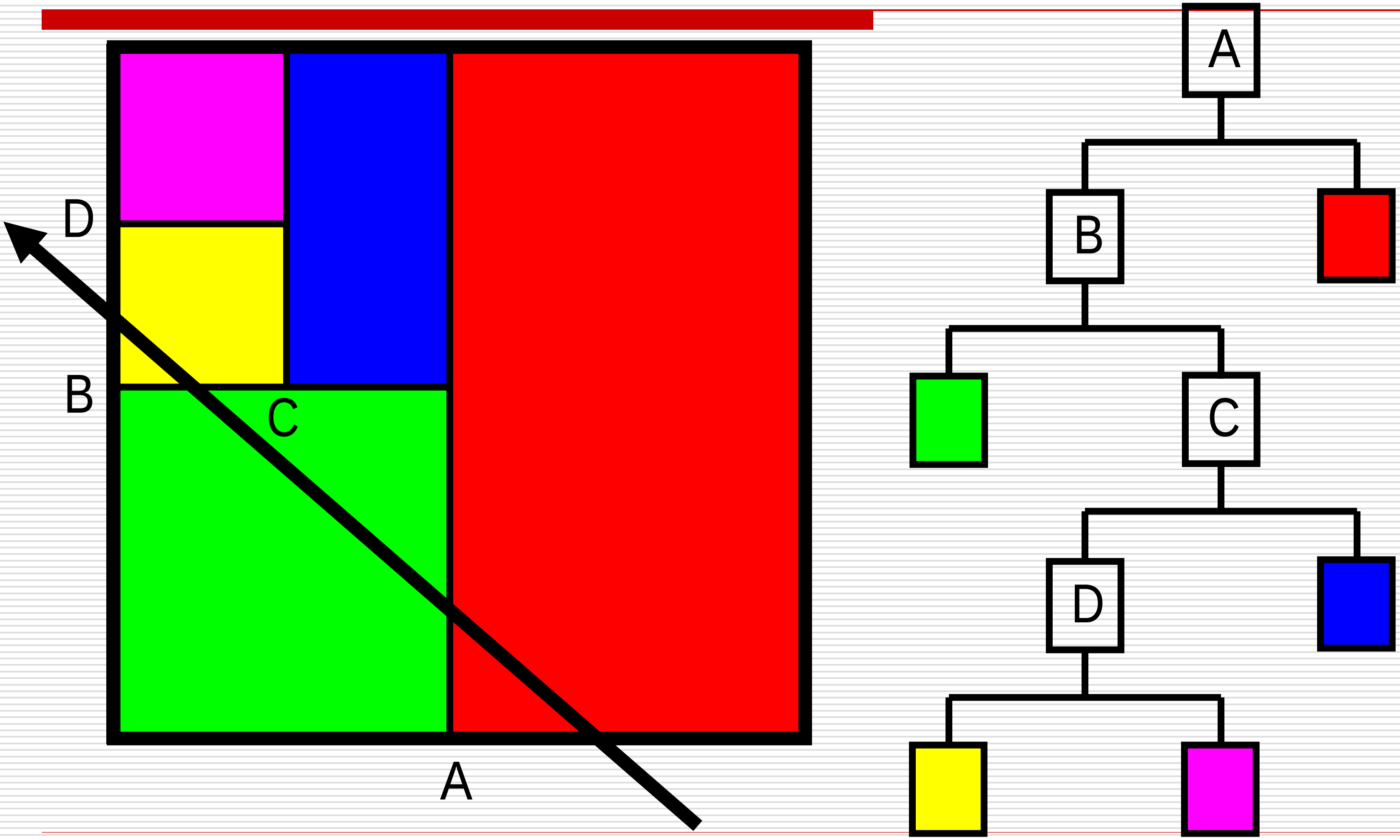
K-d Tree



K-d Tree

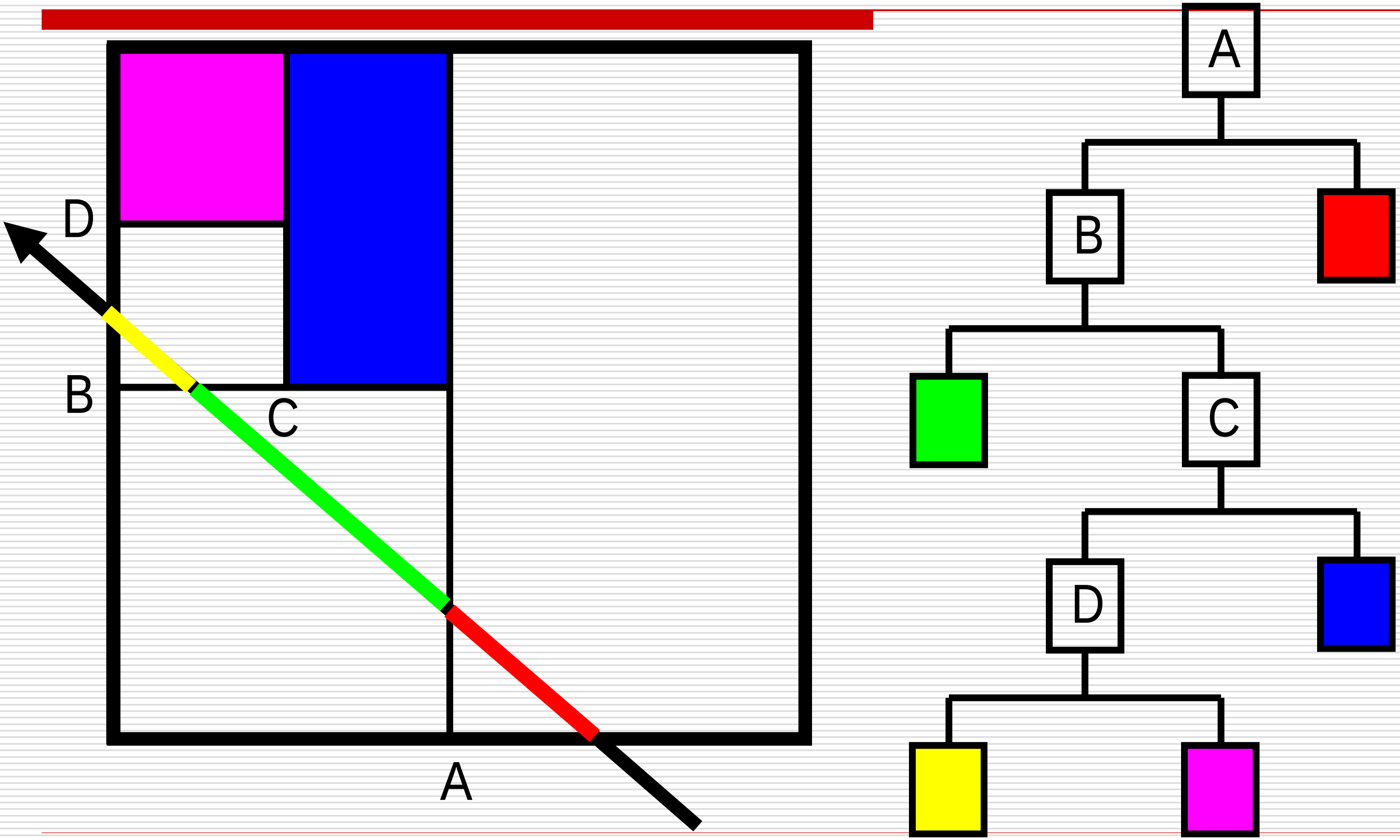


K-d Tree



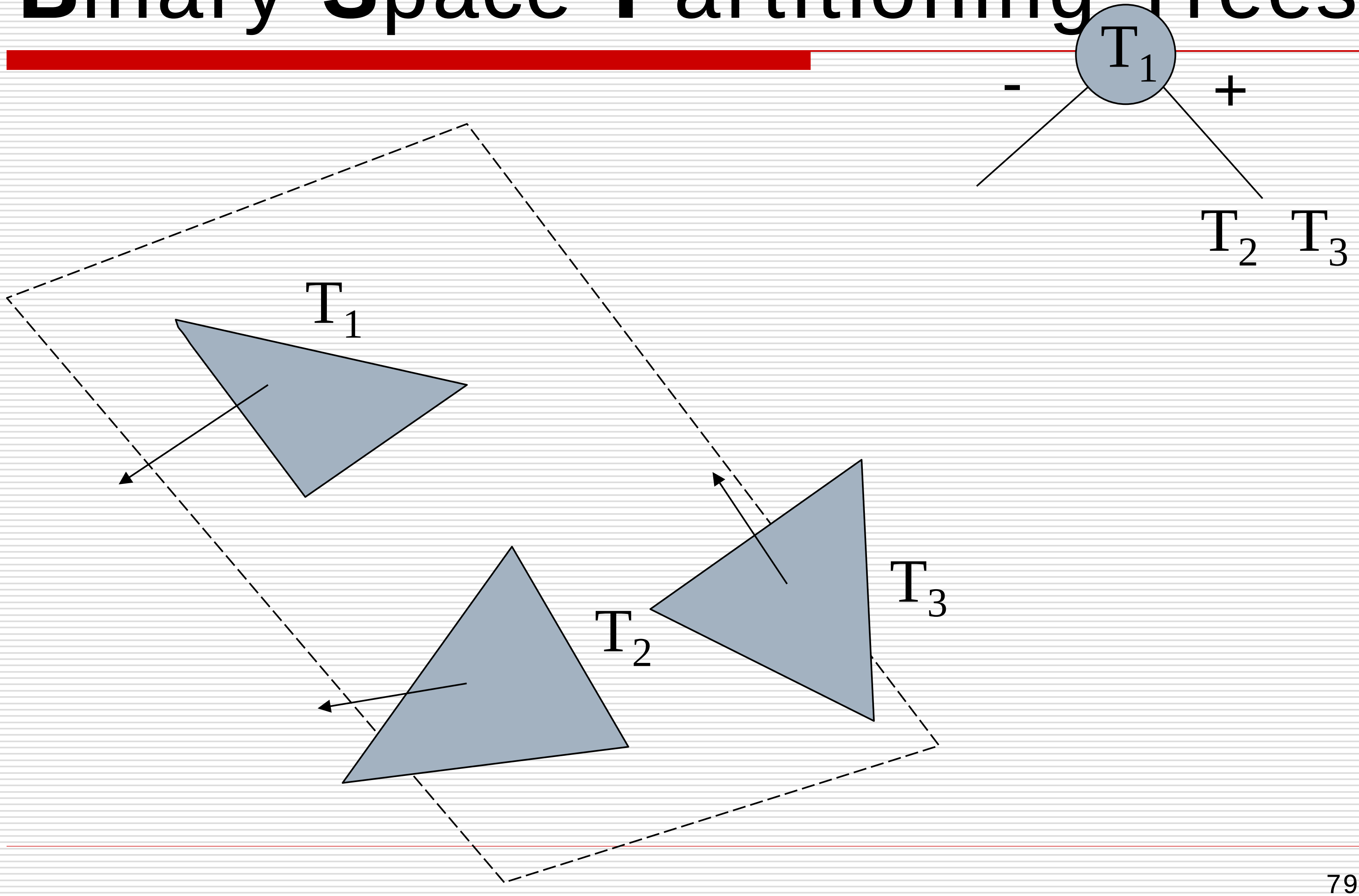
Leaf nodes correspond to unique regions in space

K-d Tree Traversal

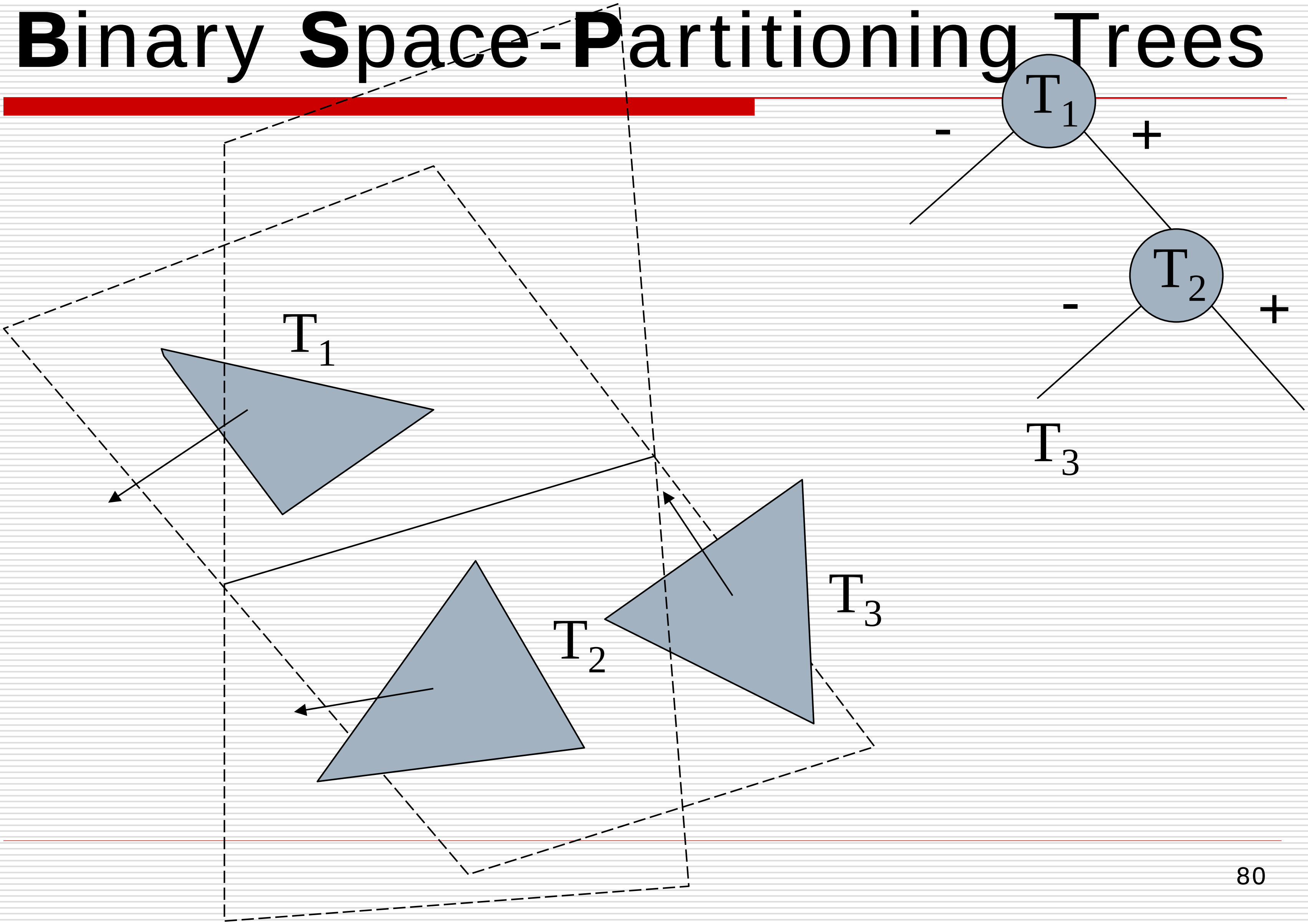


Leaf nodes correspond to unique regions in space

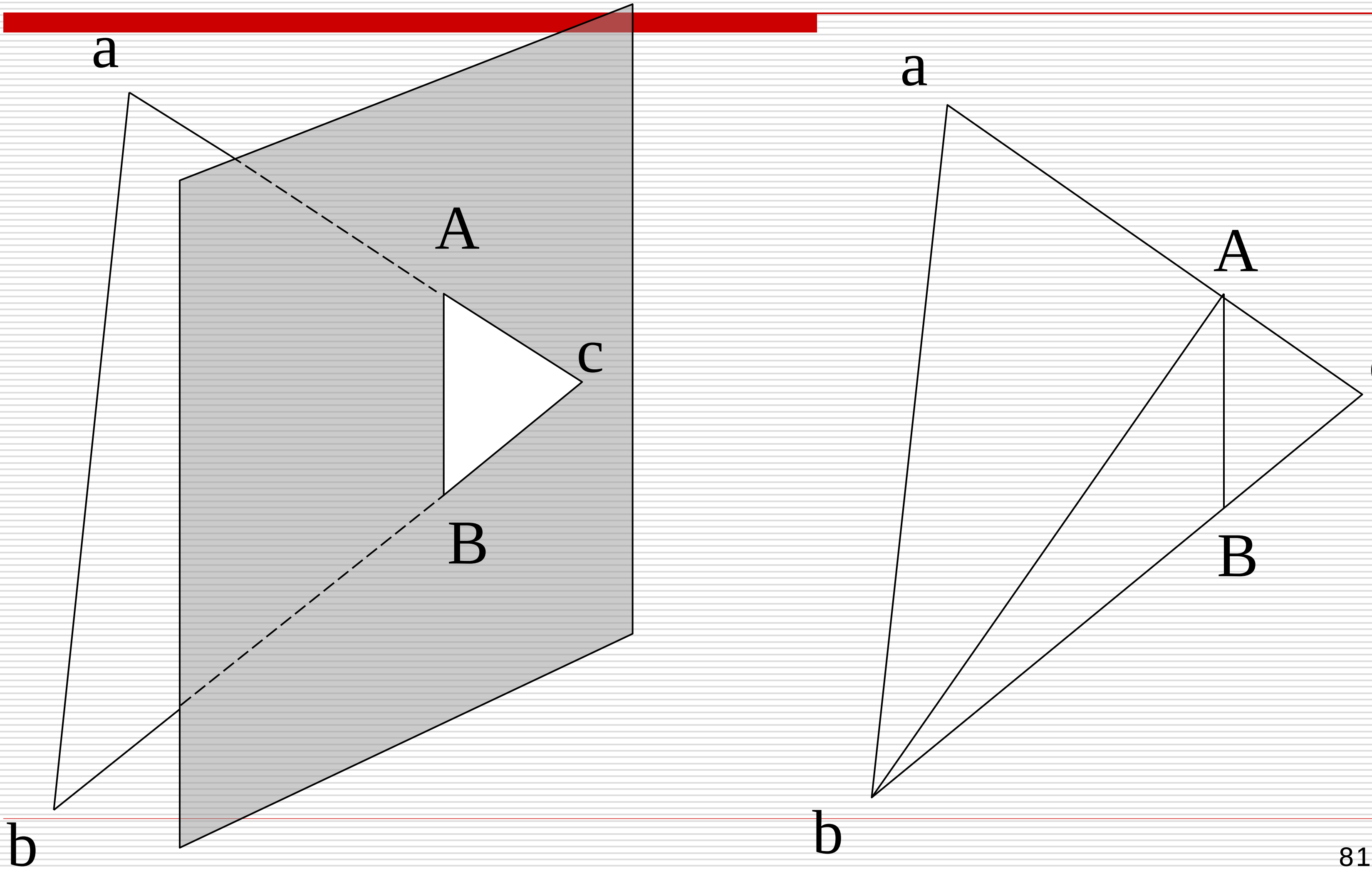
Binary **S**pace-**P**artitioning Trees



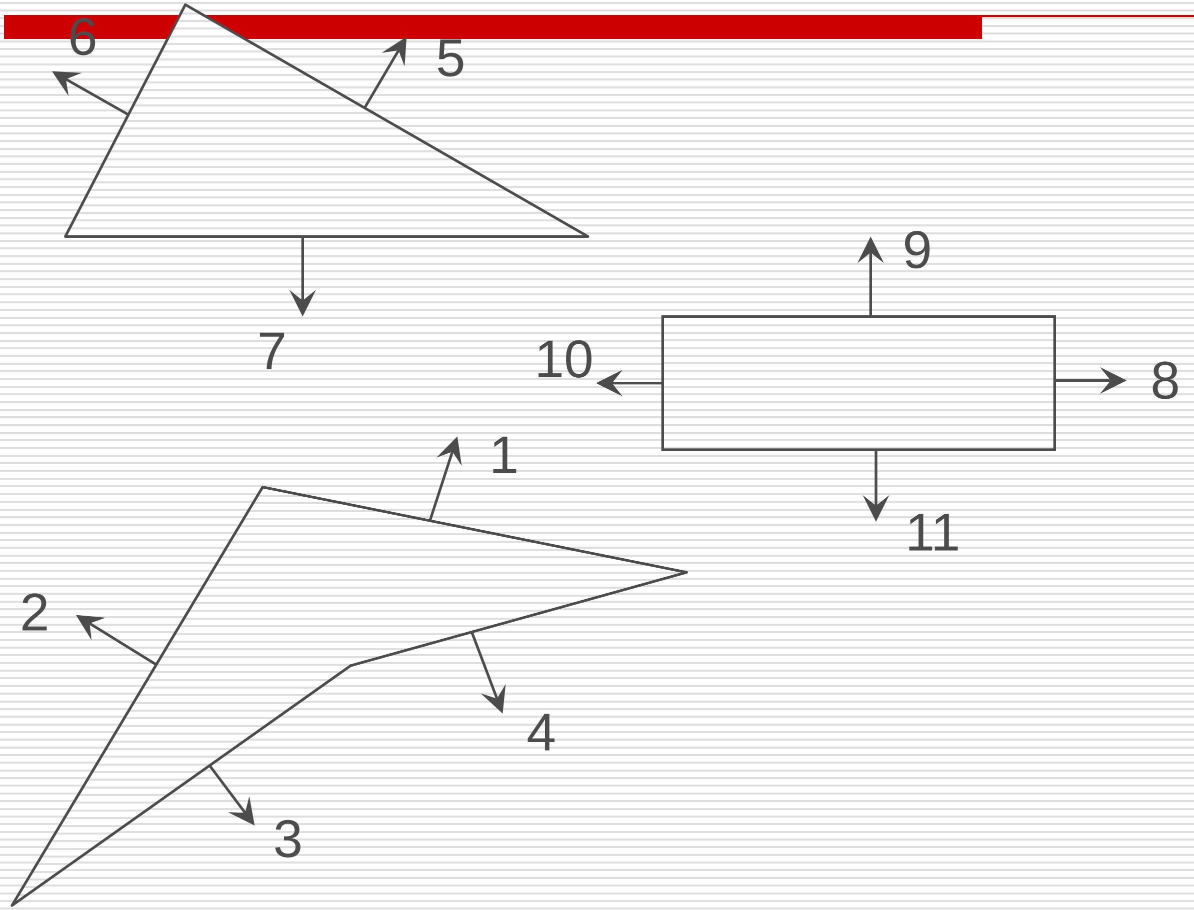
Binary Space-Partitioning Trees



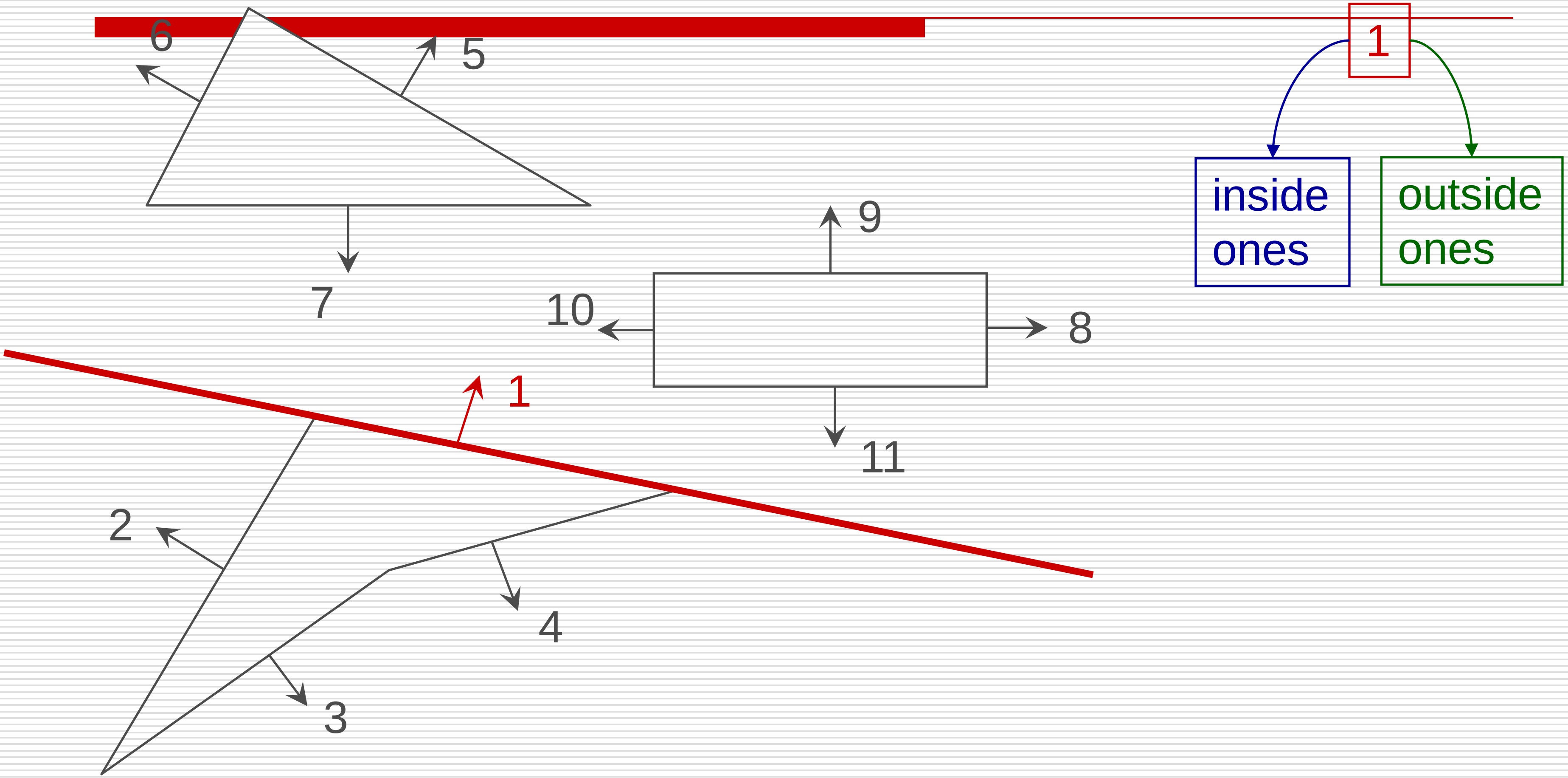
Splitting triangles



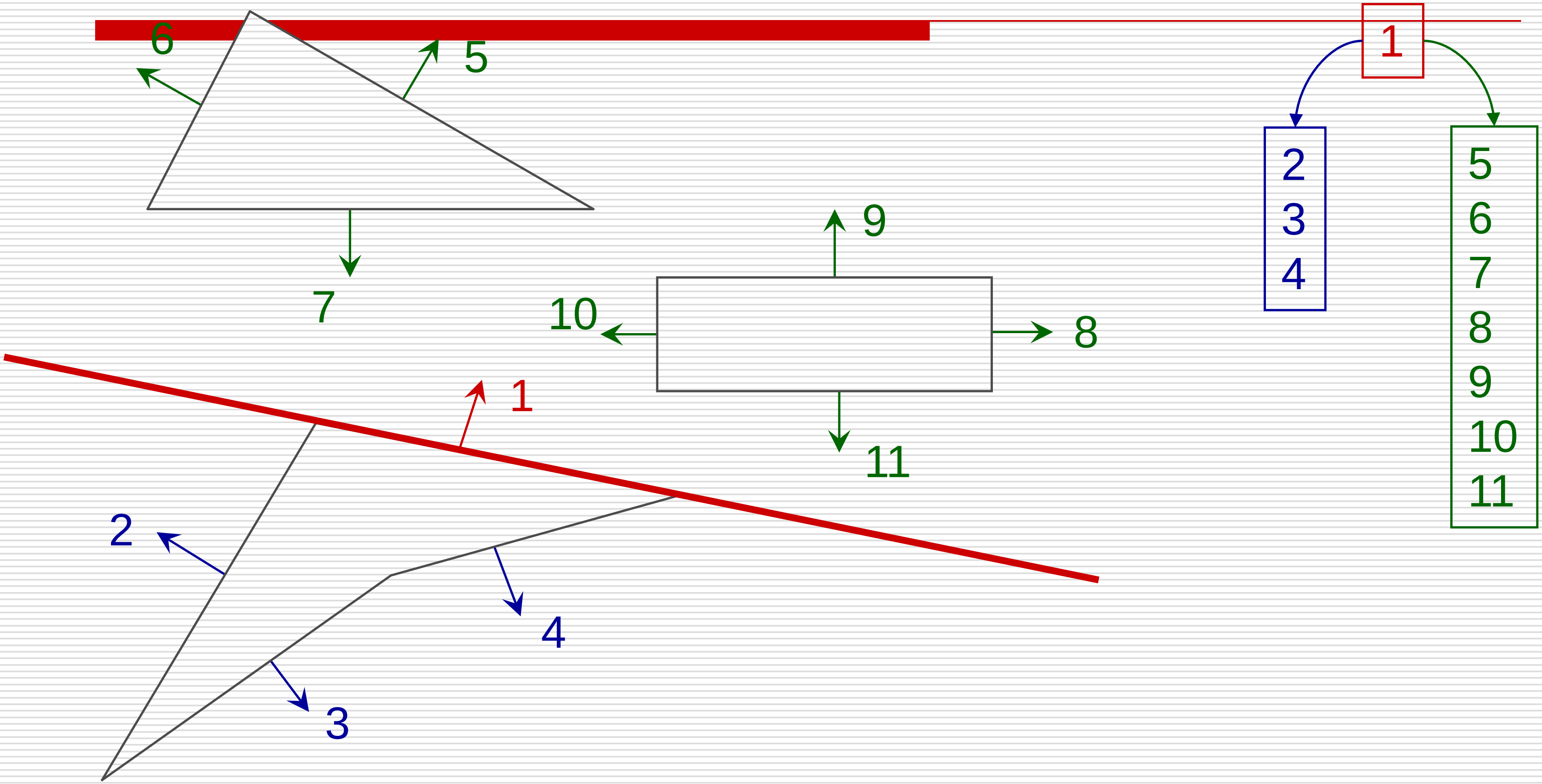
BSP Tree



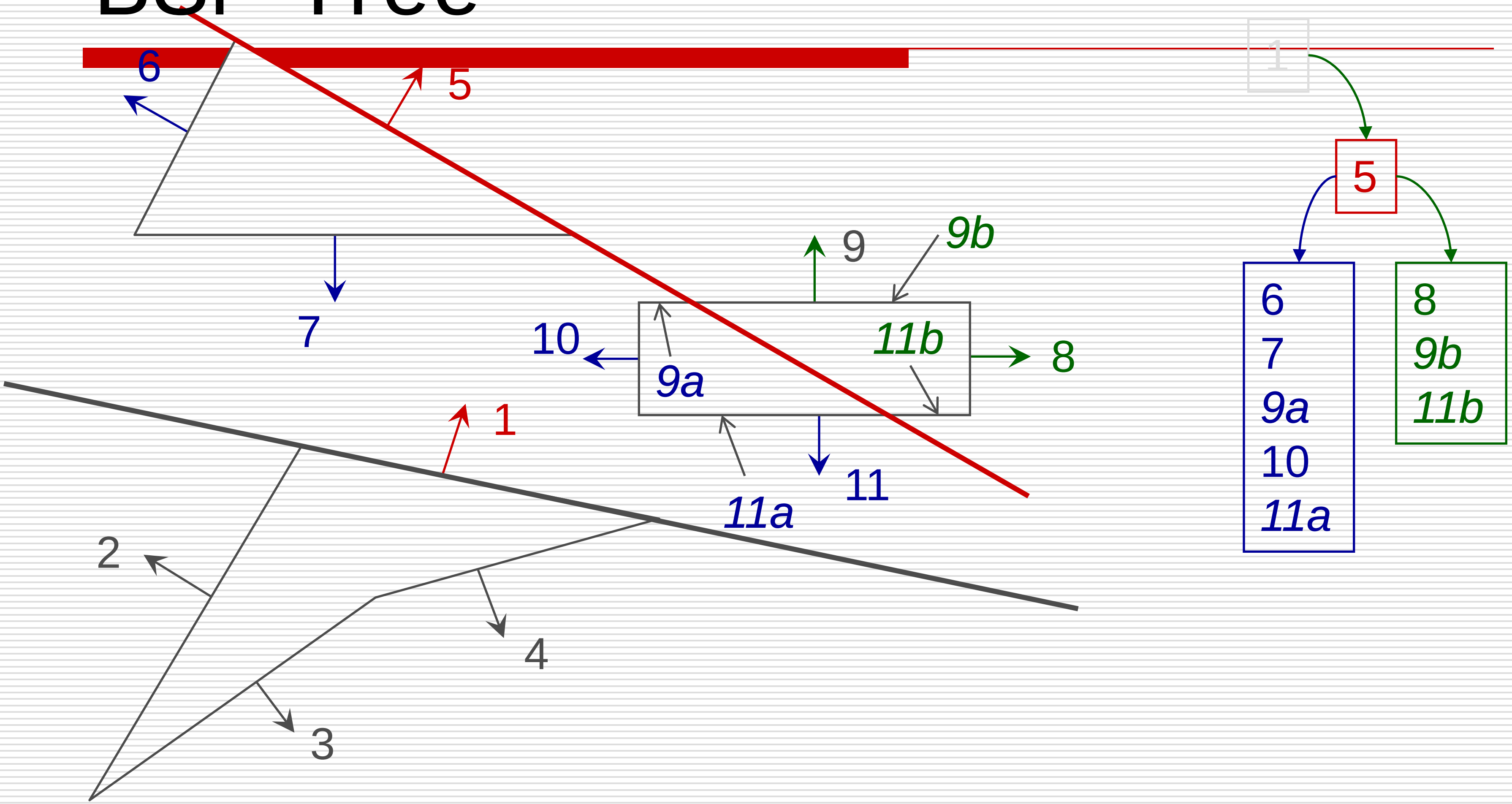
BSP Tree



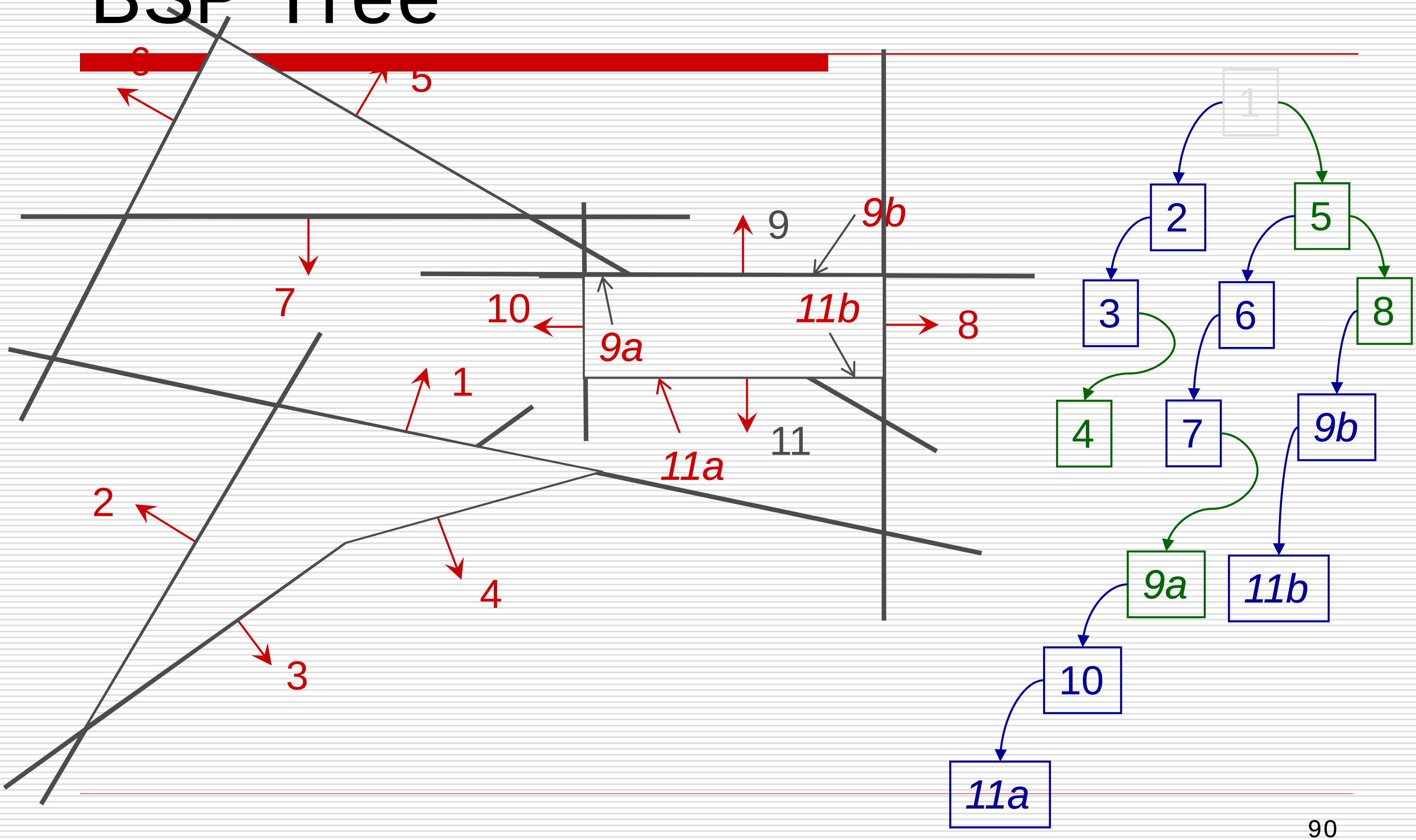
BSP Tree



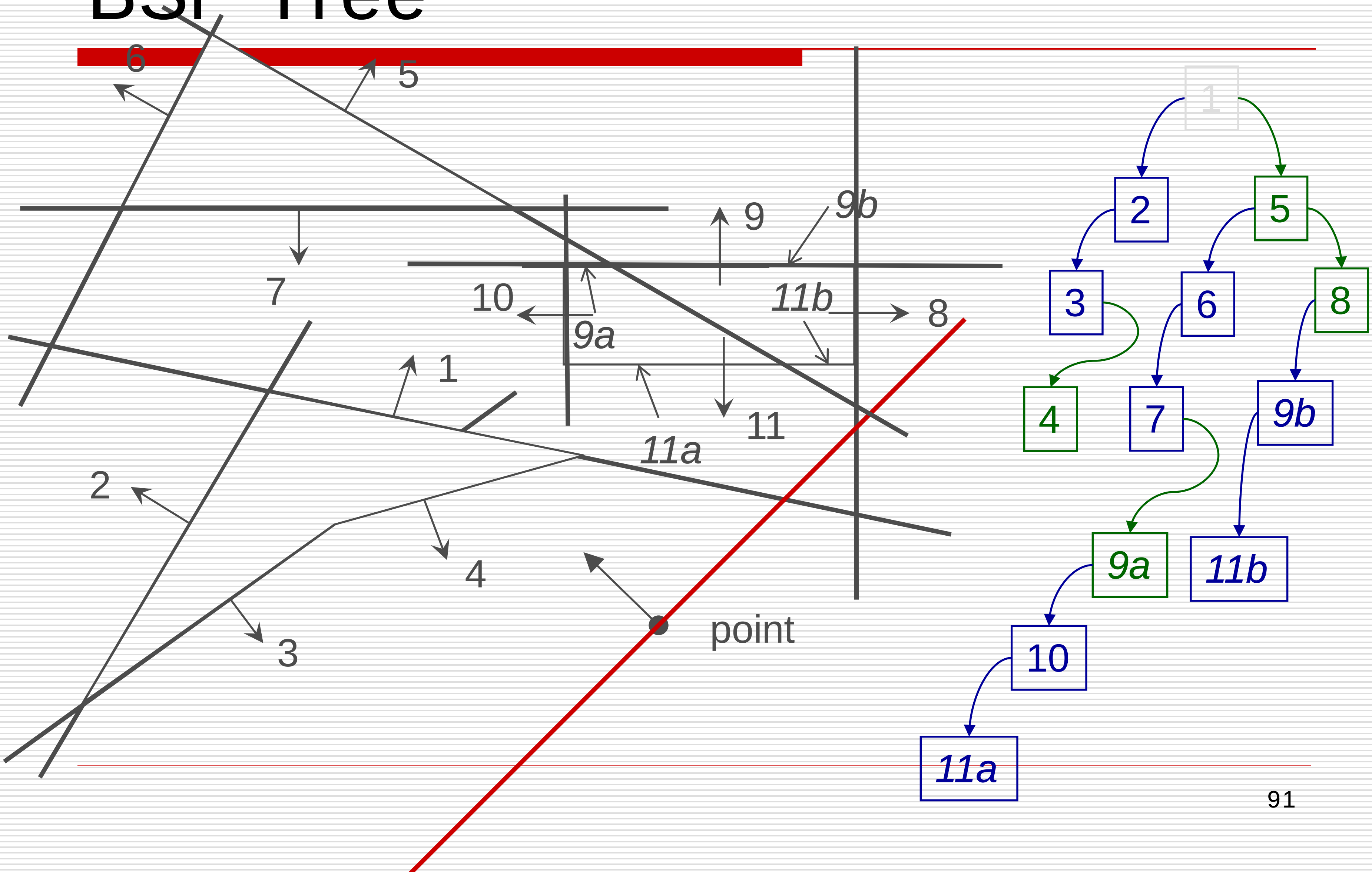
BSP Tree



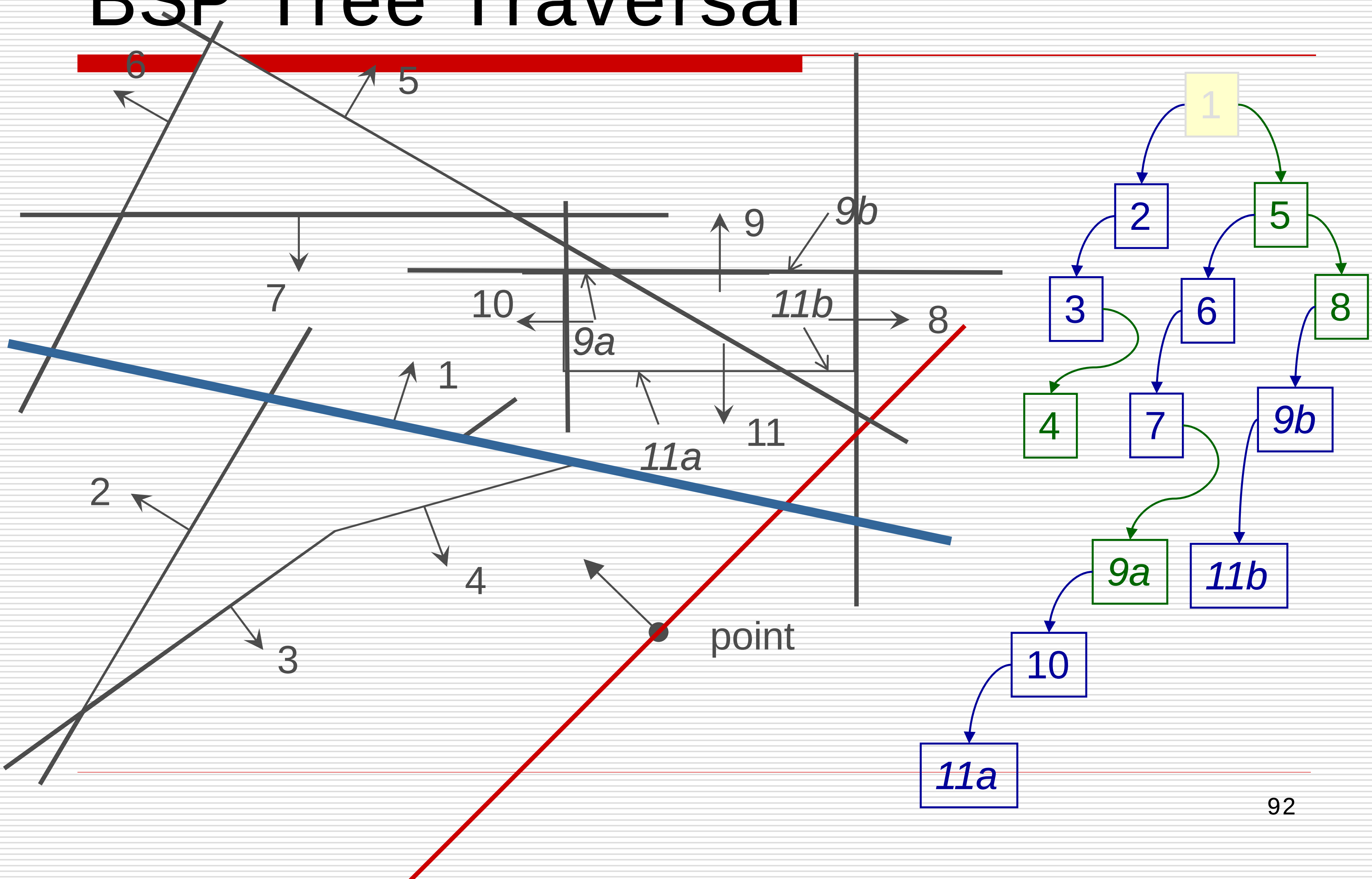
BSP Tree



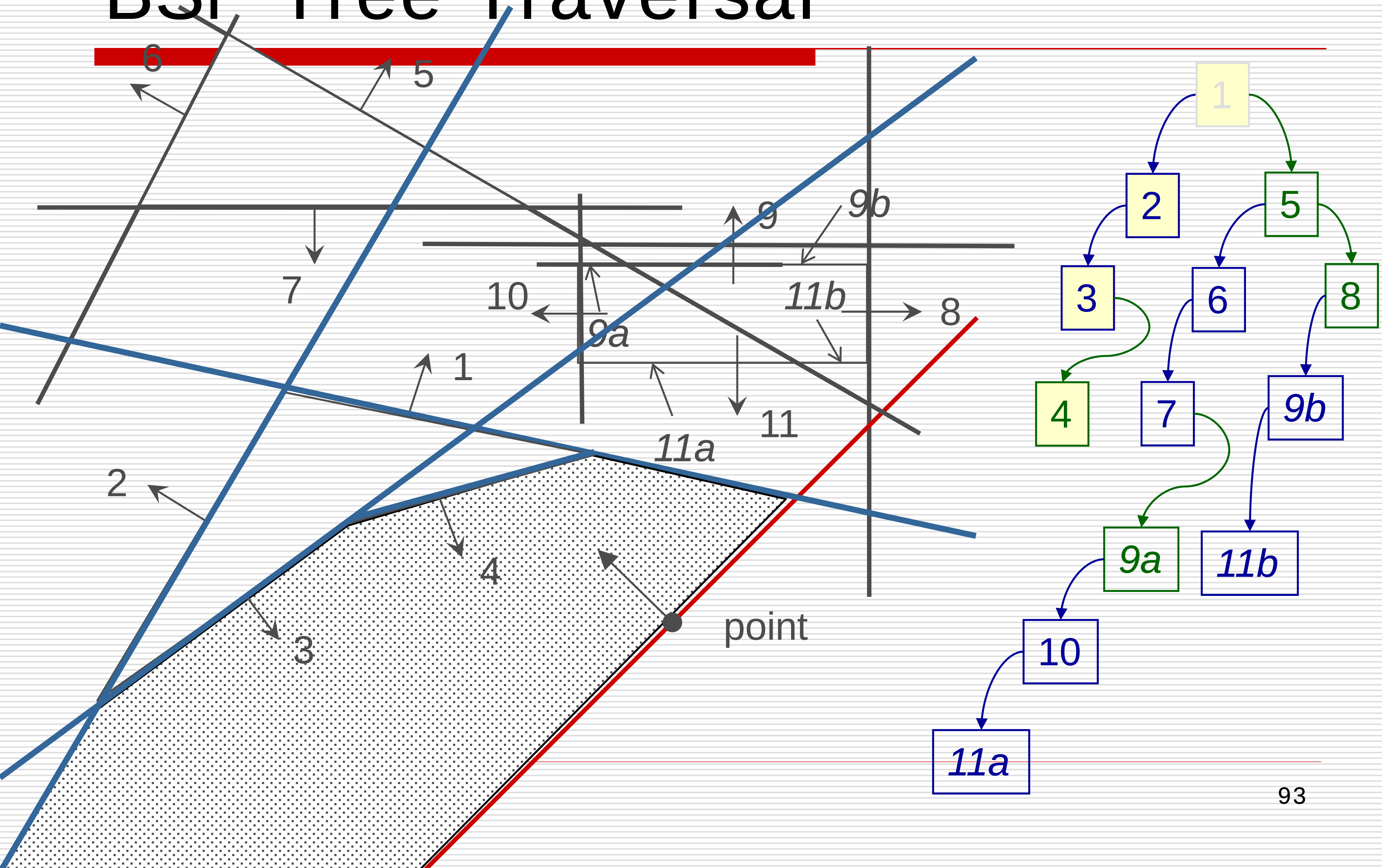
BSP Tree



BSP Tree Traversal



BSP Tree Traversal



Level-of-Details

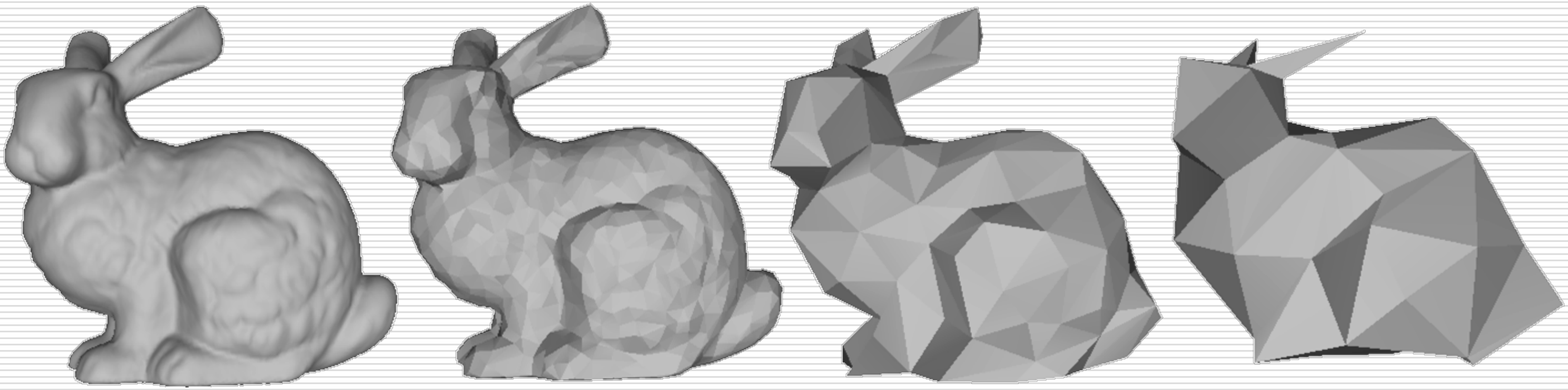
- Discrete LOD
  - Switch multiple resolution models run-timely
- Continuous LOD
  - Use progressive mesh to dynamically reduce the rendered polygons
- View-dependent LOD
  - Basically for terrain

Level of Detail: The Basic Idea

- One solution:
  - Simplify the polygonal geometry of small or distant objects
  - Known as *Level of Detail* or *LOD*
    - a.k.a. polygonal simplification, geometric simplification, mesh reduction, multiresolution modeling, …

Level of Detail: Traditional Approach

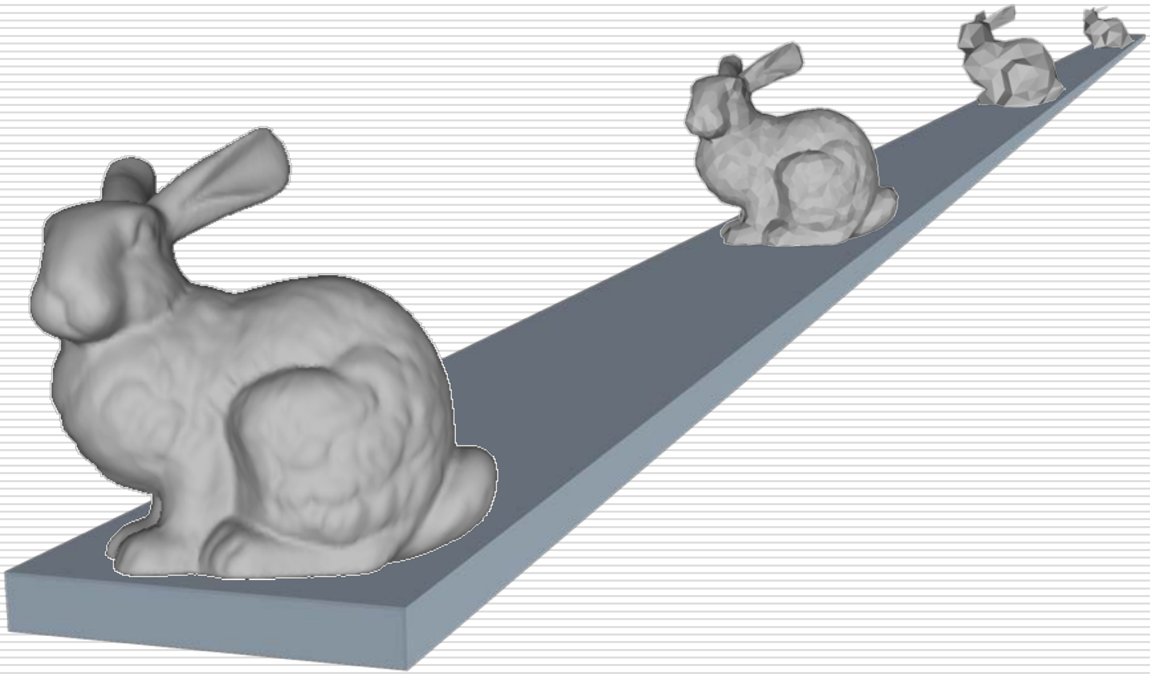
- Create *levels of detail* (*LODs*) of objects:



69,451 polys　　2,502 polys　　251 polys　　76 polys

Level of Detail: Traditional Approach

- Distant objects use coarser LODs:



Traditional Approach: Discrete Level of Detail

- Traditional LOD in a nutshell:
  - Create LODs for each object separately in a preprocess
  - At run-time, pick each object's LOD according to the object's distance (or similar criterion)
- Since LODs are created offline at fixed resolutions, this can be referred as Discrete LOD

Discrete LOD: Advantages

- Simplest programming model; decouples simplification and rendering
  - LOD creation need not address real-time rendering constraints
  - Run-time rendering need only pick LODs

Discrete LOD: Advantages

- Fits modern graphics hardware well
  - Easy to compile each LOD into triangle strips, display lists, vertex arrays, …
  - These render *much* faster than unorganized polygons on today's hardware (3-5 x)

Discrete LOD: Disadvantages

- So why use anything but discrete LOD?
- Answer: sometimes discrete LOD not suited for *drastic simplification*
- Some problem cases:
  - Terrain flyovers
  - Volumetric isosurfaces
  - Super-detailed range scans
  - Massive CAD models

Continuous Level of Detail

- A departure from the traditional static approach:
  - Discrete LOD: create individual LODs in a preprocess
  - Continuous LOD: create data structure from which a desired level of detail can be extracted at *run time*.

Continuous LOD: Advantages

- Better granularity → better fidelity
  - LOD is specified exactly, not chosen from a few pre-created options
  - Thus objects use no more polygons than necessary, which frees up polygons for other objects
  - Net result: better resource utilization, leading to better overall fidelity/polygon

Continuous LOD: Advantages

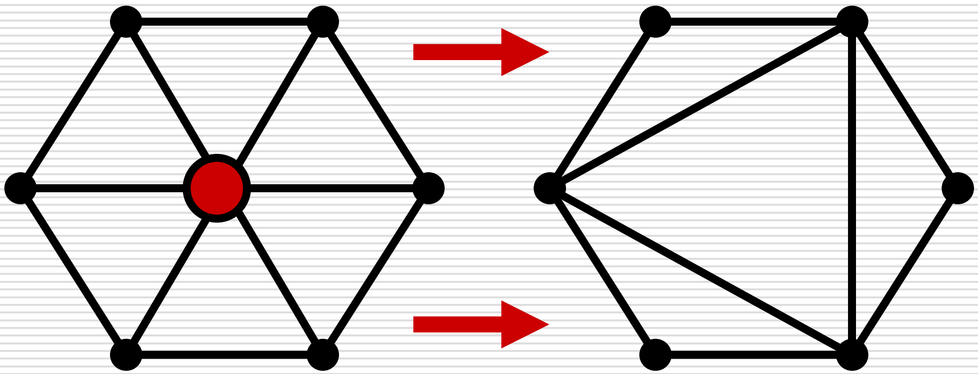
- Better granularity → smoother transitions
  - Switching between traditional LODs can introduce visual “popping” effect
  - Continuous LOD can adjust detail gradually and incrementally, reducing visual pops
    - Can even *geomorph* the fine-grained simplification operations over several frames to eliminate pops [Hoppe 96, 98]

Continuous LOD: Advantages

- Supports progressive transmission
  - *Progressive Meshes [Hoppe 97]*
  - *Progressive Forest Split Compression [Taubin 98]*
- Leads to *view-dependent LOD*
  - Use current view parameters to select best representation for *the current view*
  - Single objects may thus span several levels of detail

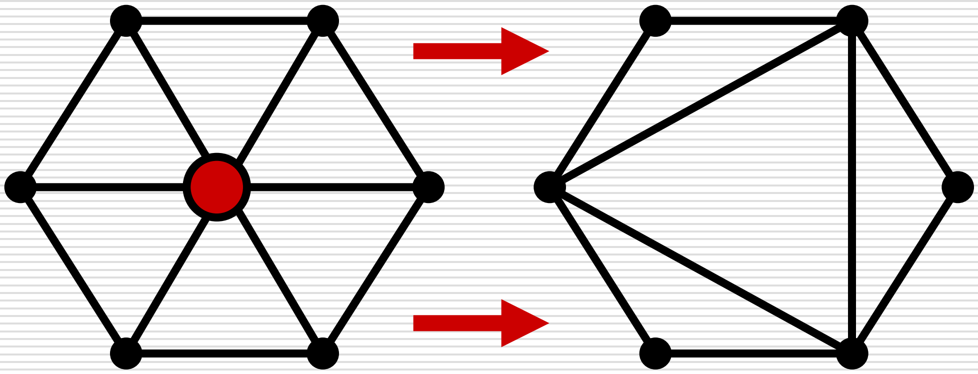
Methodology

- Sequence of local operations
  - Involve near neighbors - only small *patch* affected in each operation
  - Each operation introduces error
  - Find and apply operation which introduces the least error



Simplification Operations

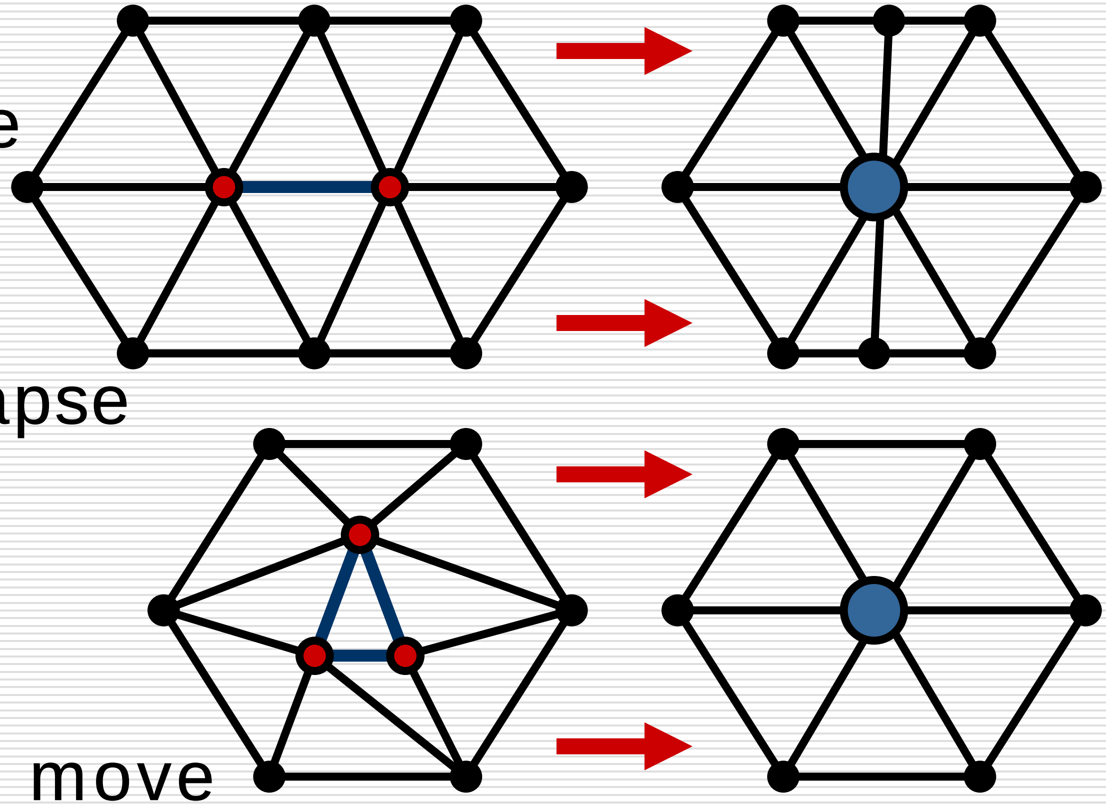
- Decimation
  - Vertex removal
    - v ← v-1
    - f ← f-2



  - Remaining vertices - subset of original vertex set

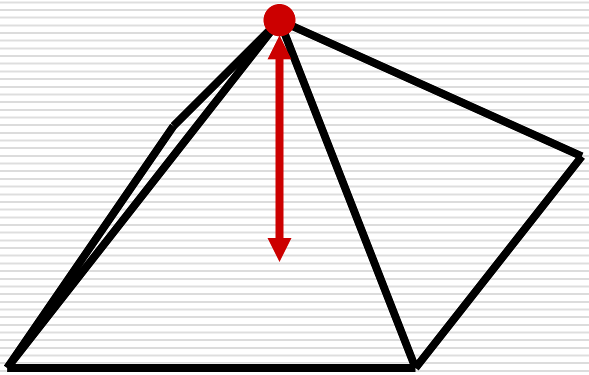
Simplification Operations

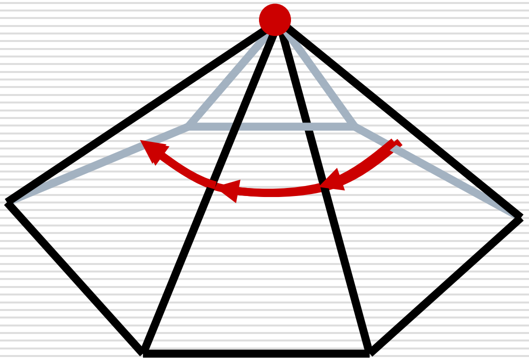
- Decimation
  - Edge collapse
    - $v \leftarrow v-1$
    - $f \leftarrow f-2$
  - Triangle collapse
    - $v \leftarrow v-2$
    - $f \leftarrow f-4$
  - Vertices may move



Simplification Error Metrics

- Measures
  - Distance to plane
  - Curvature
- Usually approximated
  - Average plane
  - Discrete curvature





$$\sum \alpha / 2\pi$$

The Basic Algorithm

- Repeat
  - Select the element with minimal error
  - Perform simplification operation
    - (remove/contract)
  - Update error
    - (local/global)

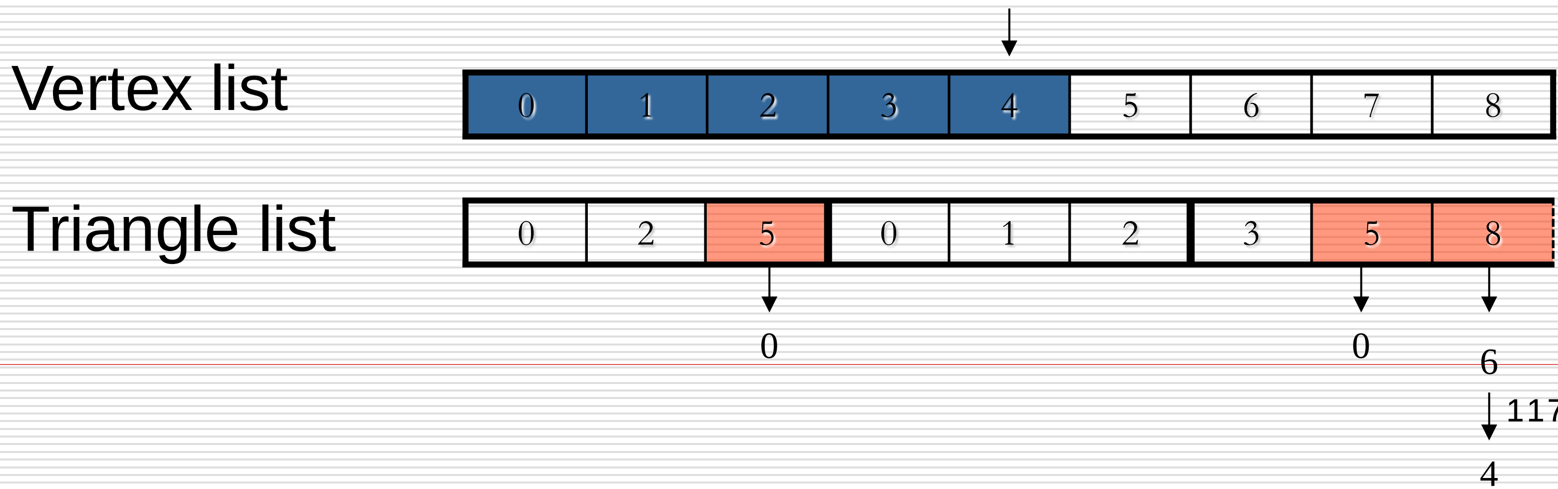
- Until mesh size / quality is achieved

Progressive Meshes

- Render a model in different Level-of-Detail at run time
- User-controlledly or automatically change the percentage of rendered vertices
- Use collapse map to control the simplification process

Collapse map

Index	0	1	2	3	4	5	6	7	8
Map	0	1	1	2	3	0	4	5	6



View-dependent LOD for Terrain - ROAM

- Real-time Optimal Adapting Meshes (ROAM)
- Use height map
- Run-timely to re-construct the active (for rendering) geometric topology (re-mesh) to get an optimal mesh (polygon count) to improve the rendering performance
- Someone calls this technique as the **view-dependent level-of-detail**
- Very good for fly-simulation-like application

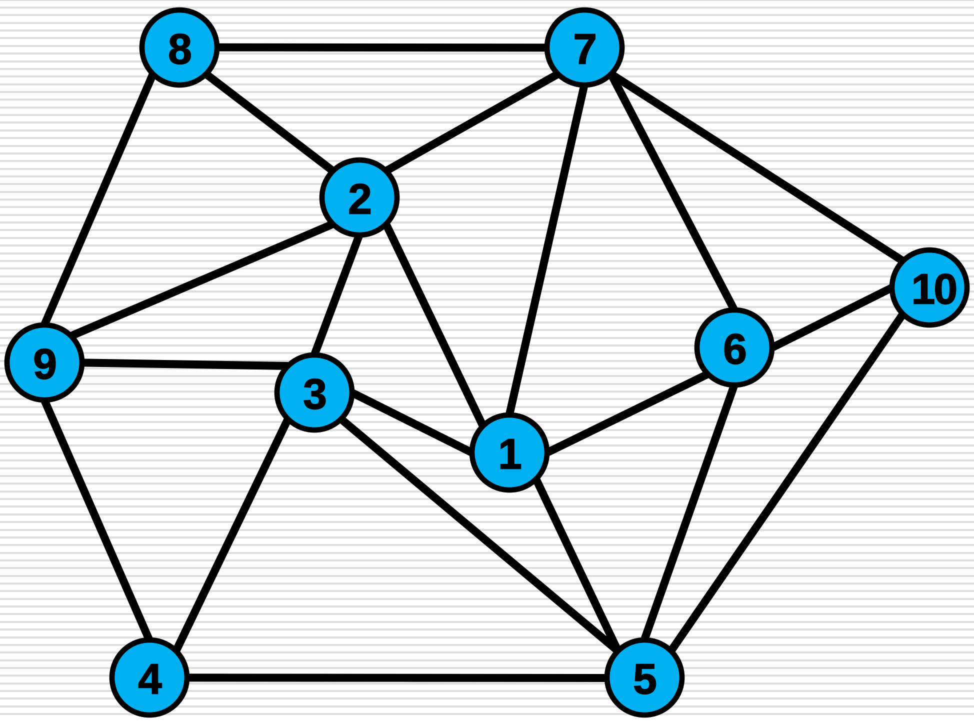
Vertex Tree & Active Triangle List

- The Vertex Tree
  - represents the entire model
  - a hierarchical clustering of vertices
  - queried each frame for updated scene
- The Active Triangle List
  - represents the current simplification
  - list of triangle to be displayed

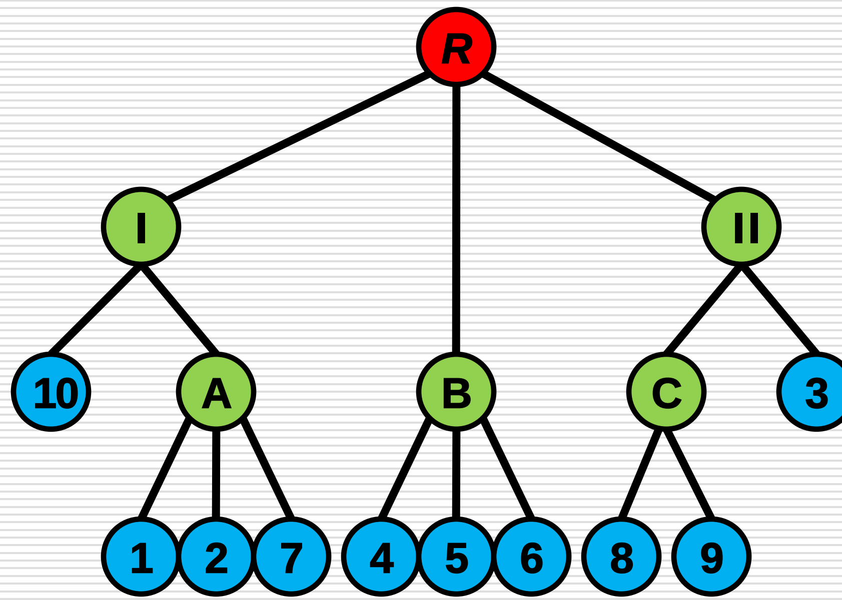
The Vertex Tree

- Each vertex tree node contains:
  - a subset of model vertices
  - a representative vertex or repvert
- *Folding* a node collapses its vertices to the repvert
- *Unfolding* a node splits the repvert back into vertices

Vertex Tree Example

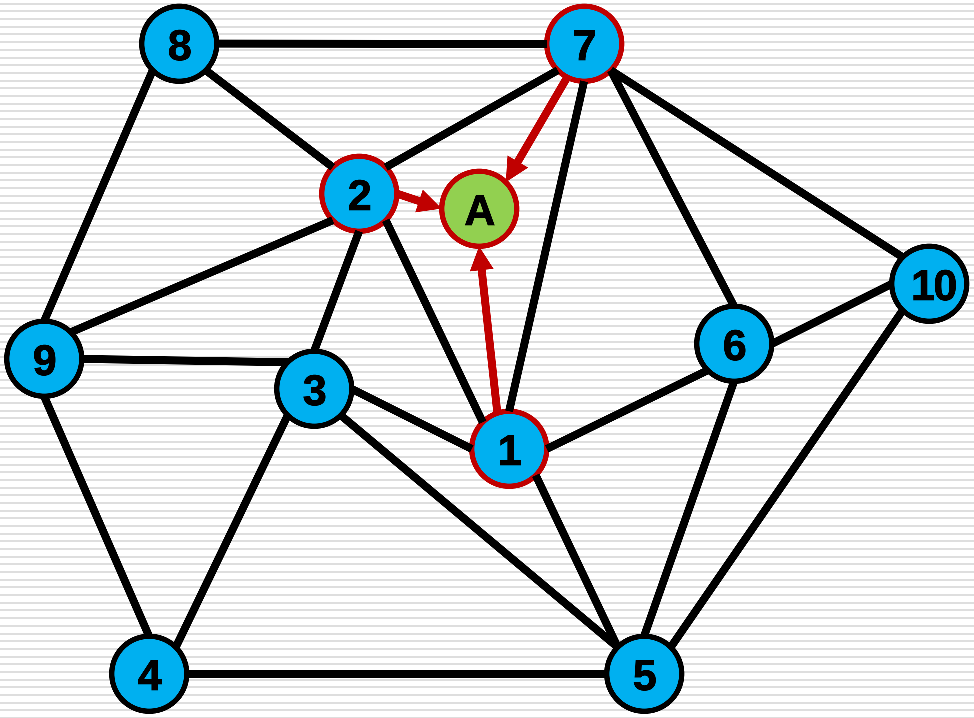


Triangles in Active List

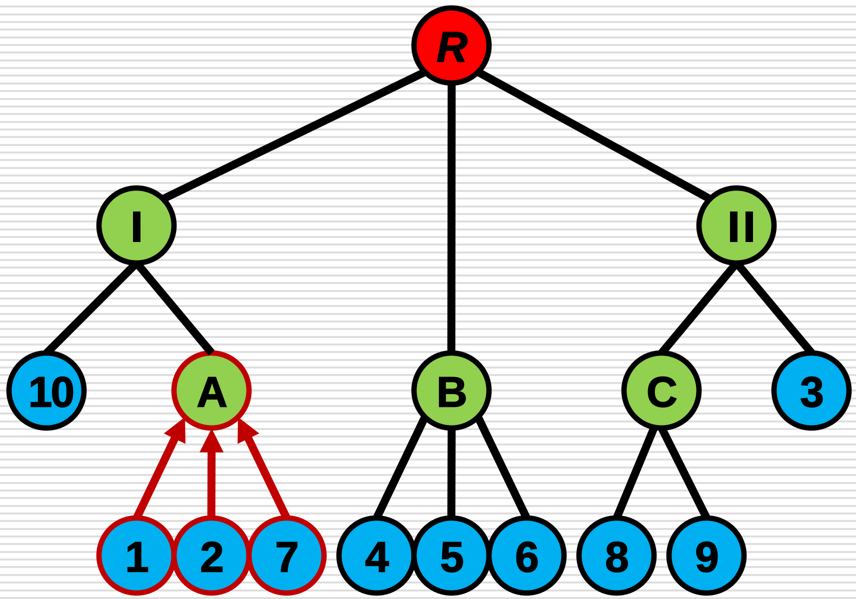


Vertex Tree

Vertex Tree Example

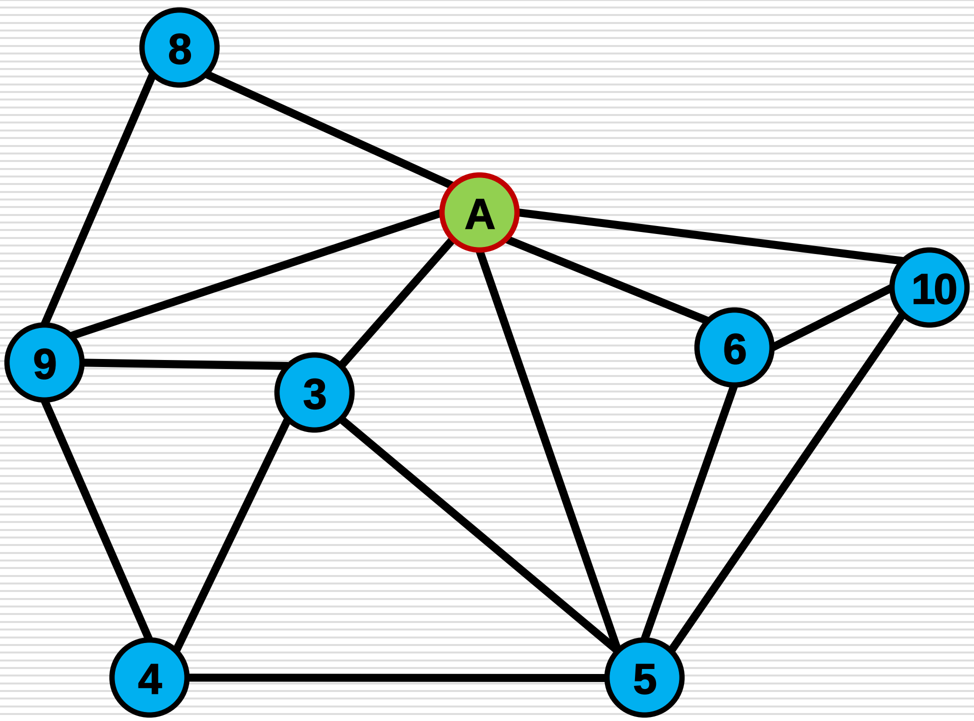


Triangles in Active List

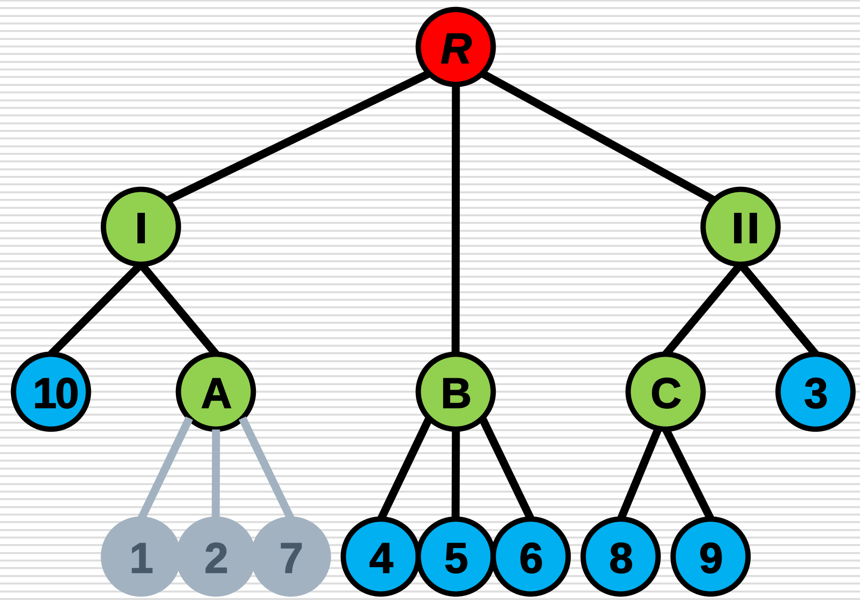


Vertex Tree

Vertex Tree Example

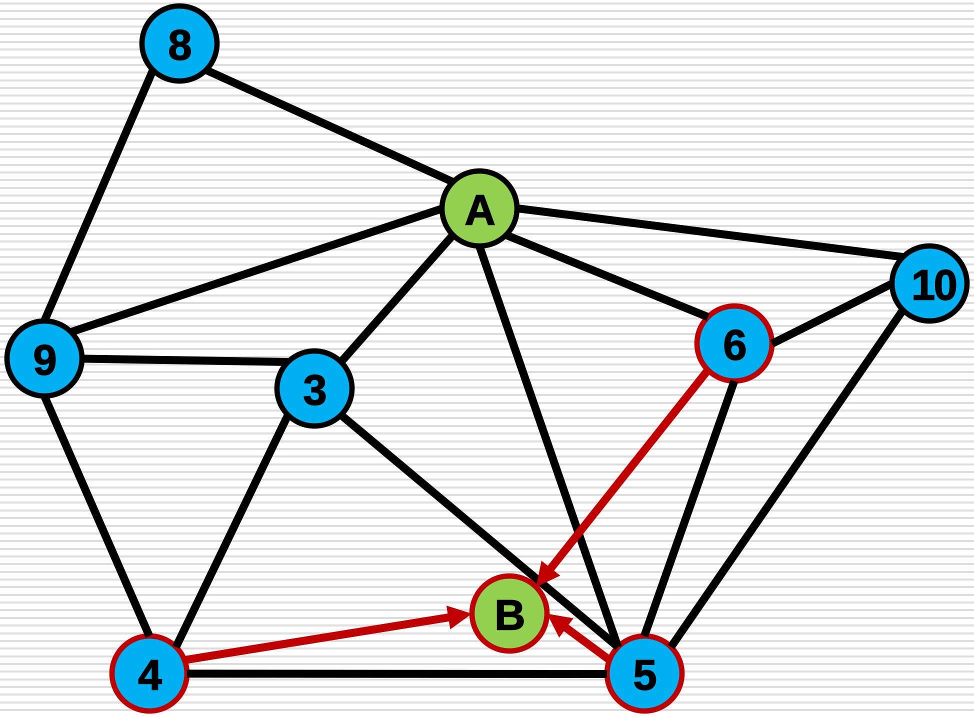


Triangles in Active List

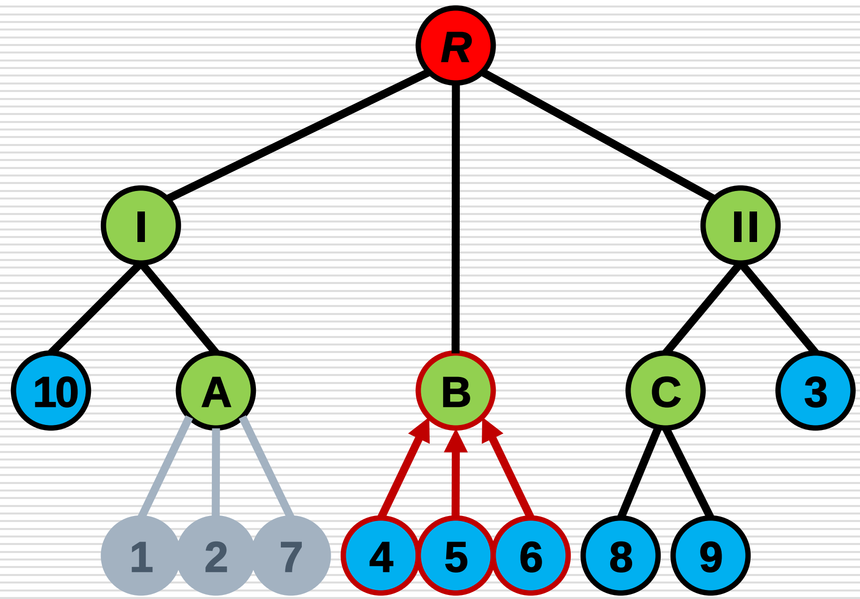


Vertex Tree

Vertex Tree Example

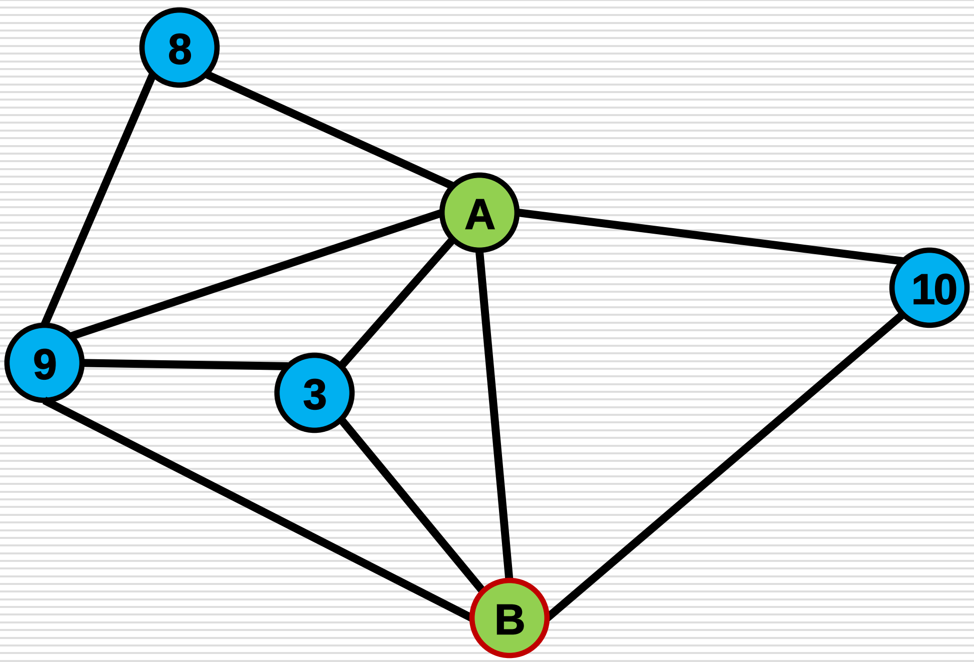


Triangles in Active List

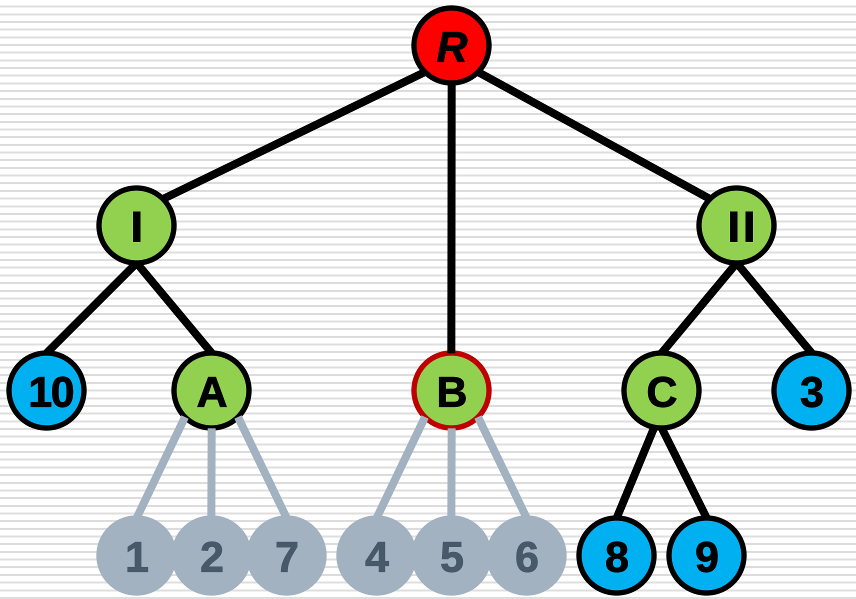


Vertex Tree

Vertex Tree Example

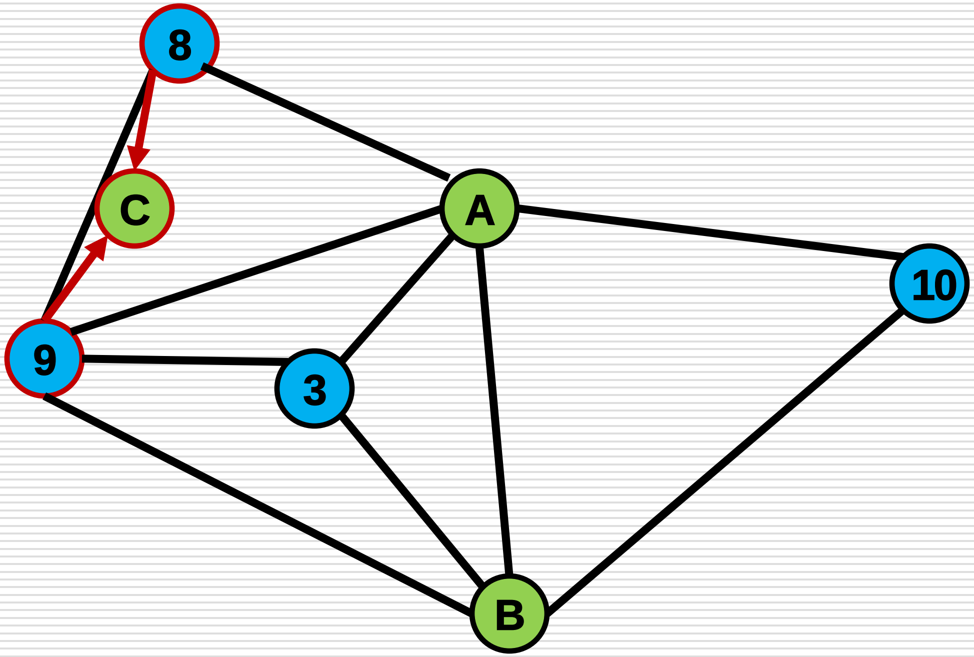


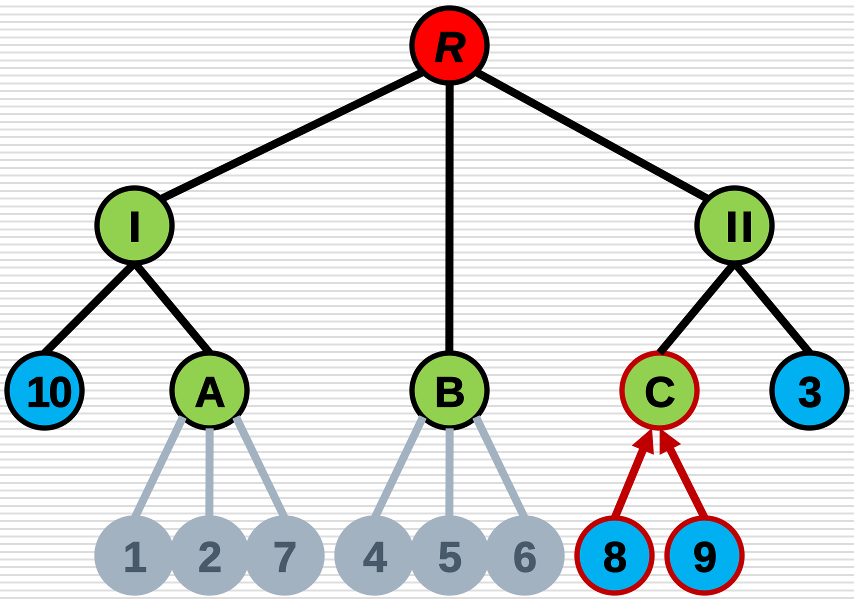
Triangles in Active List



Vertex Tree

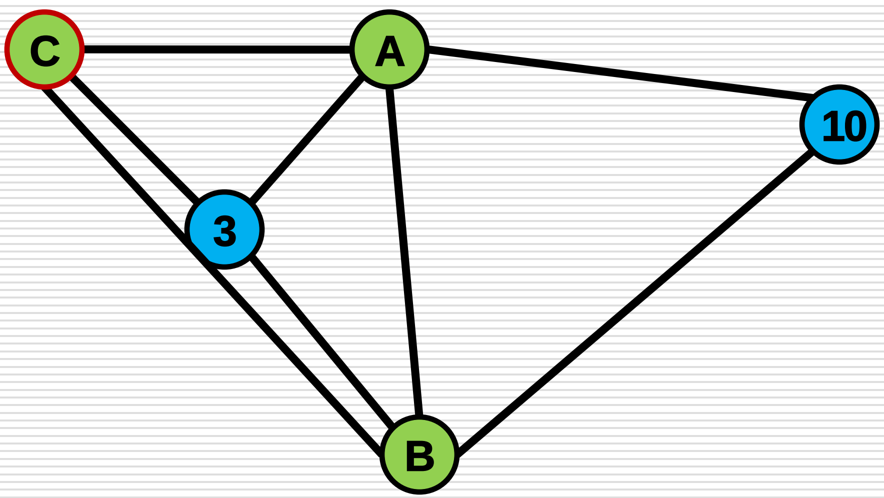
Vertex Tree Example



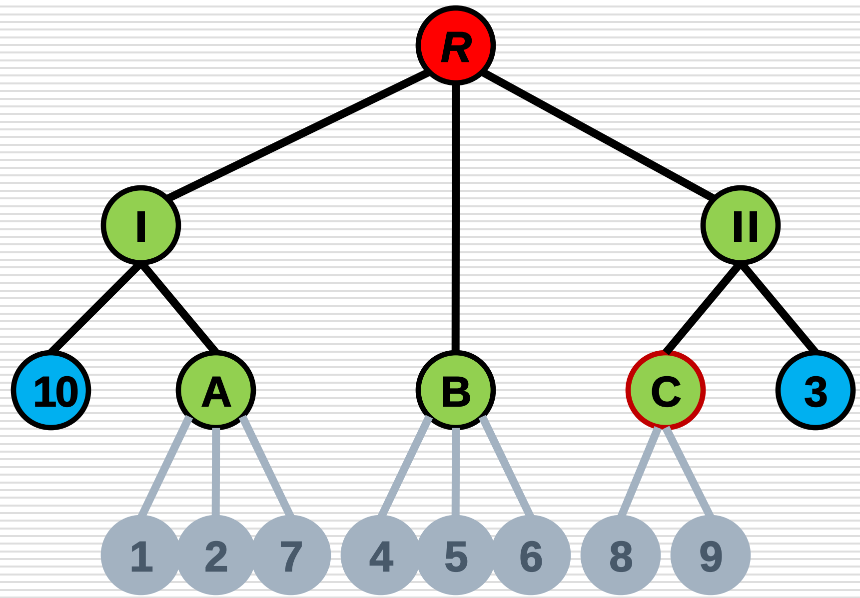


Triangles in Active List **Vertex Tree**

Vertex Tree Example

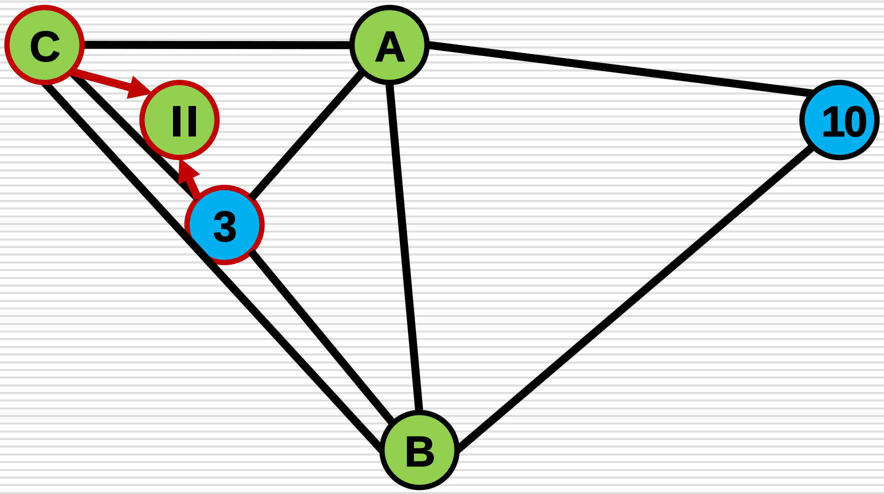


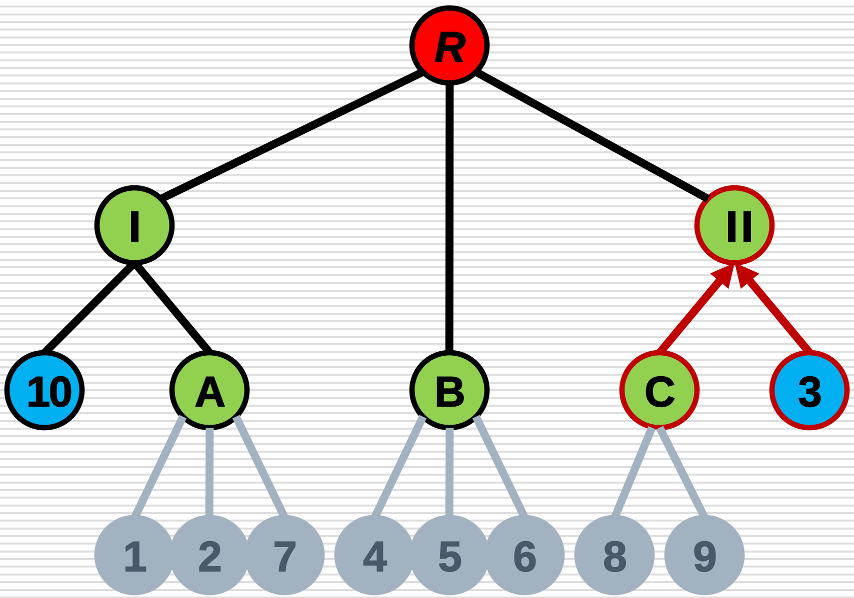
Triangles in Active List



Vertex Tree

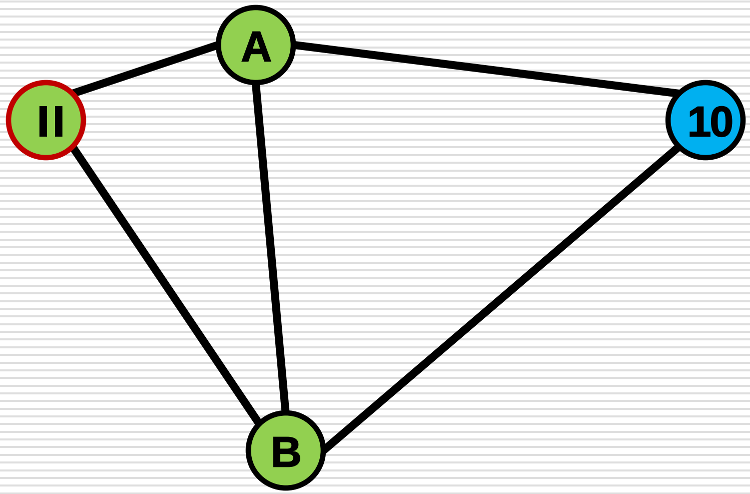
Vertex Tree Example



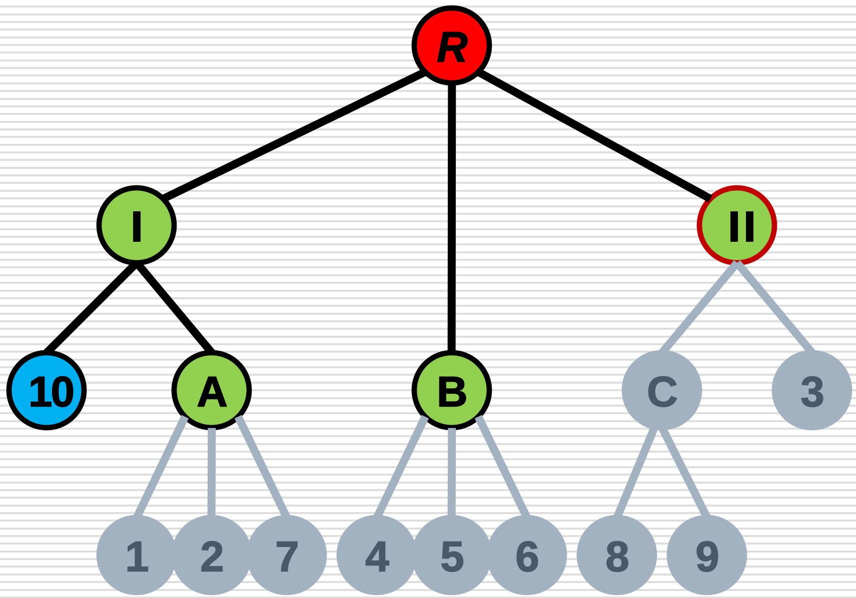


Triangles in Active List **Vertex Tree**

Vertex Tree Example

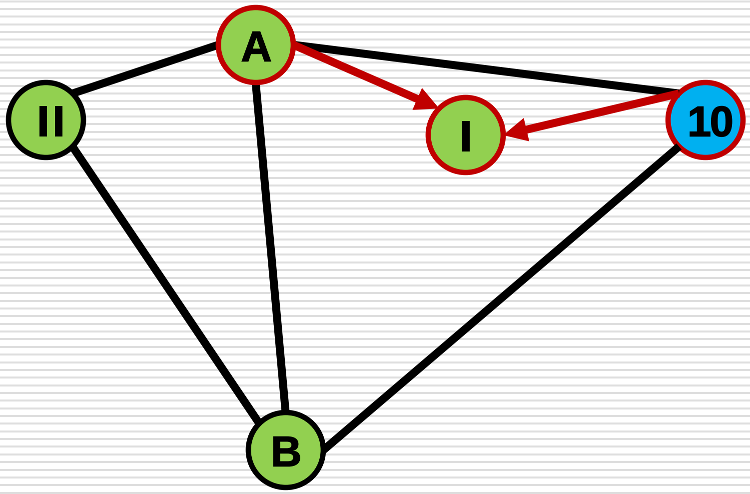


Triangles in Active List

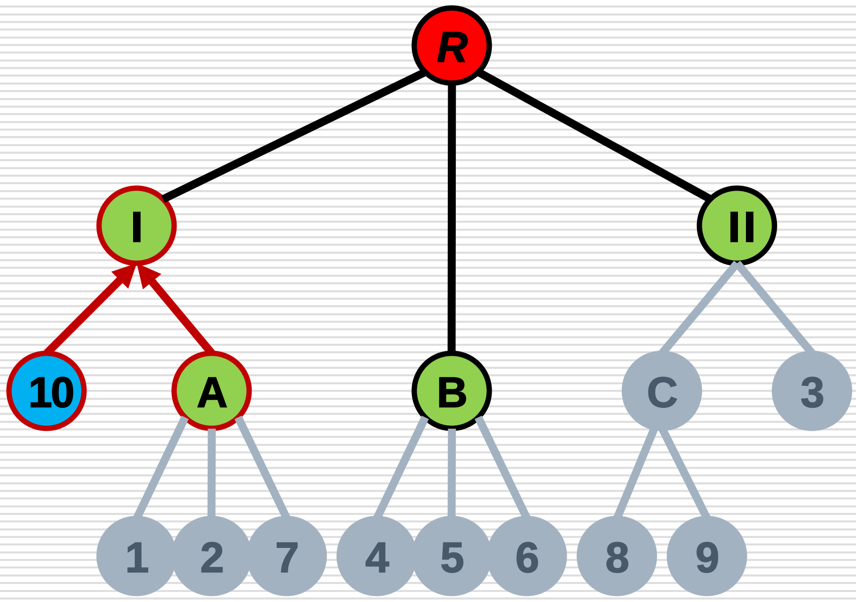


Vertex Tree

Vertex Tree Example

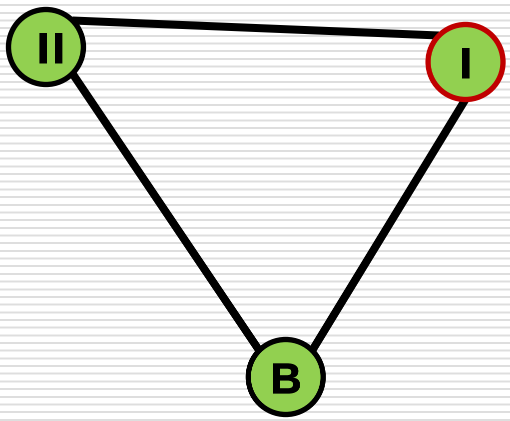


Triangles in Active List

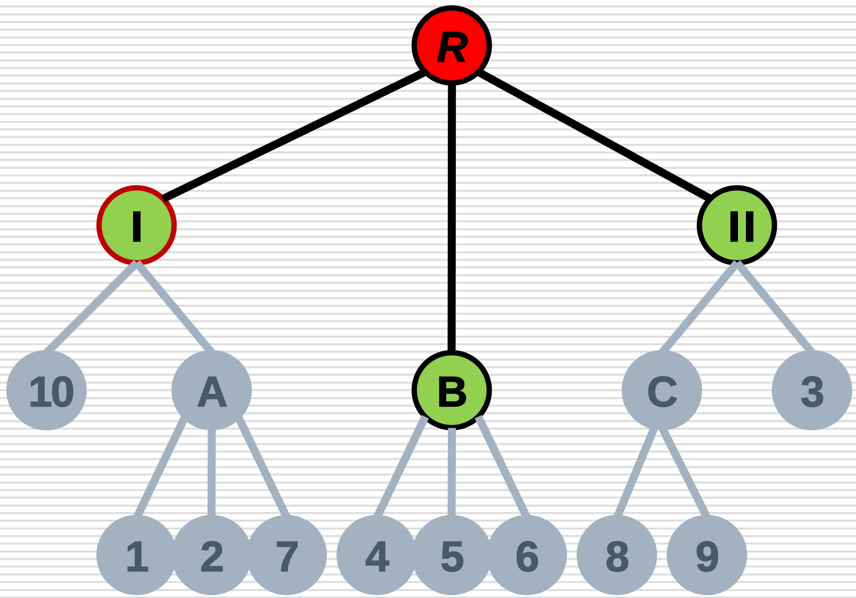


Vertex Tree

Vertex Tree Example

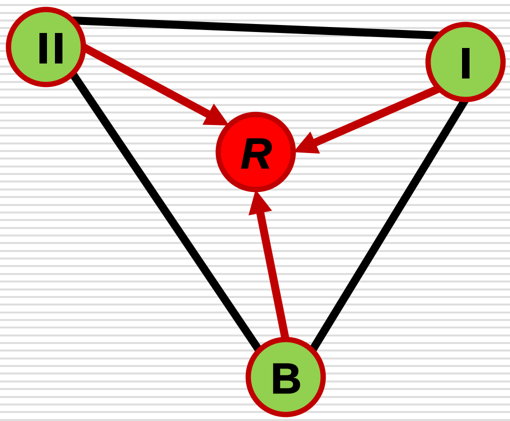


Triangles in Active List

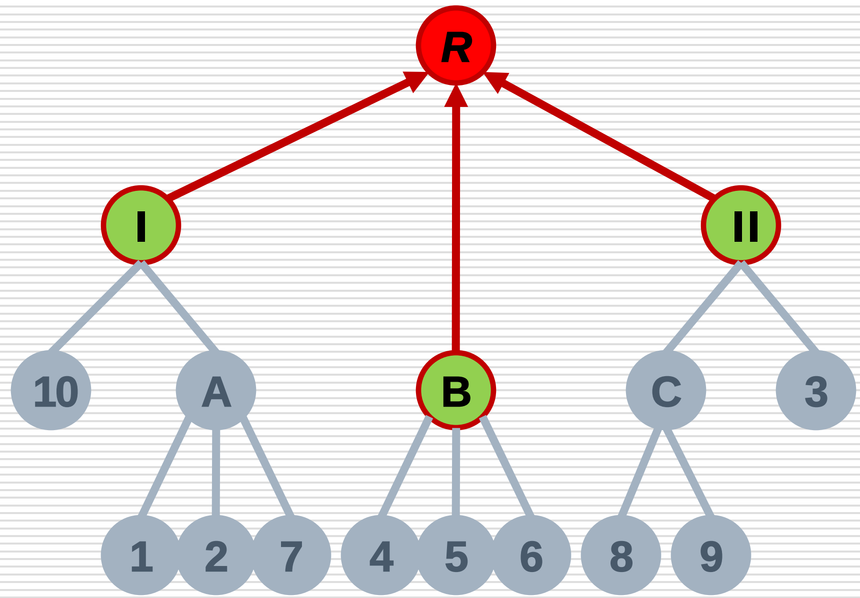


Vertex Tree

Vertex Tree Example



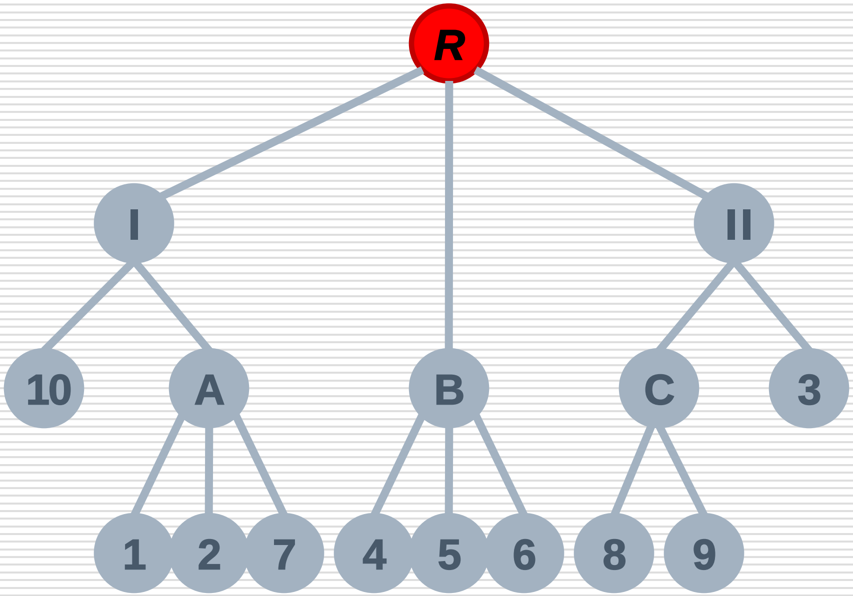
Triangles in Active List



Vertex Tree

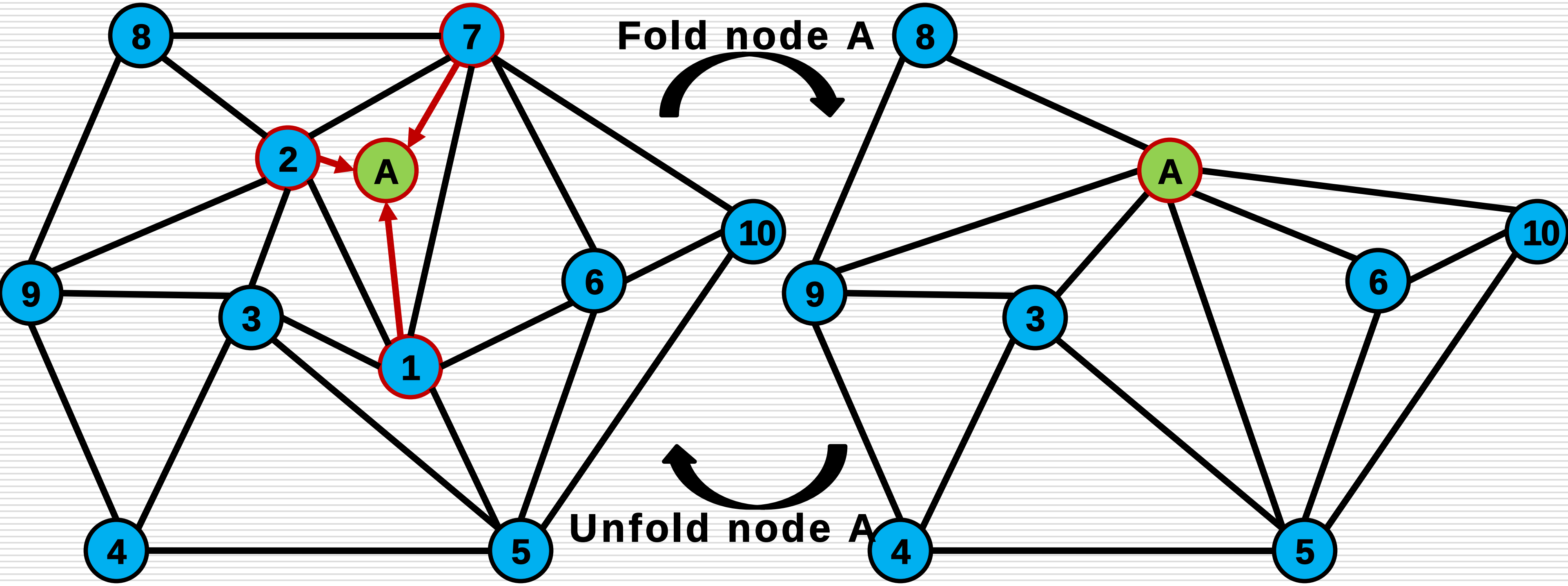
Vertex Tree Example

Triangles in Active List

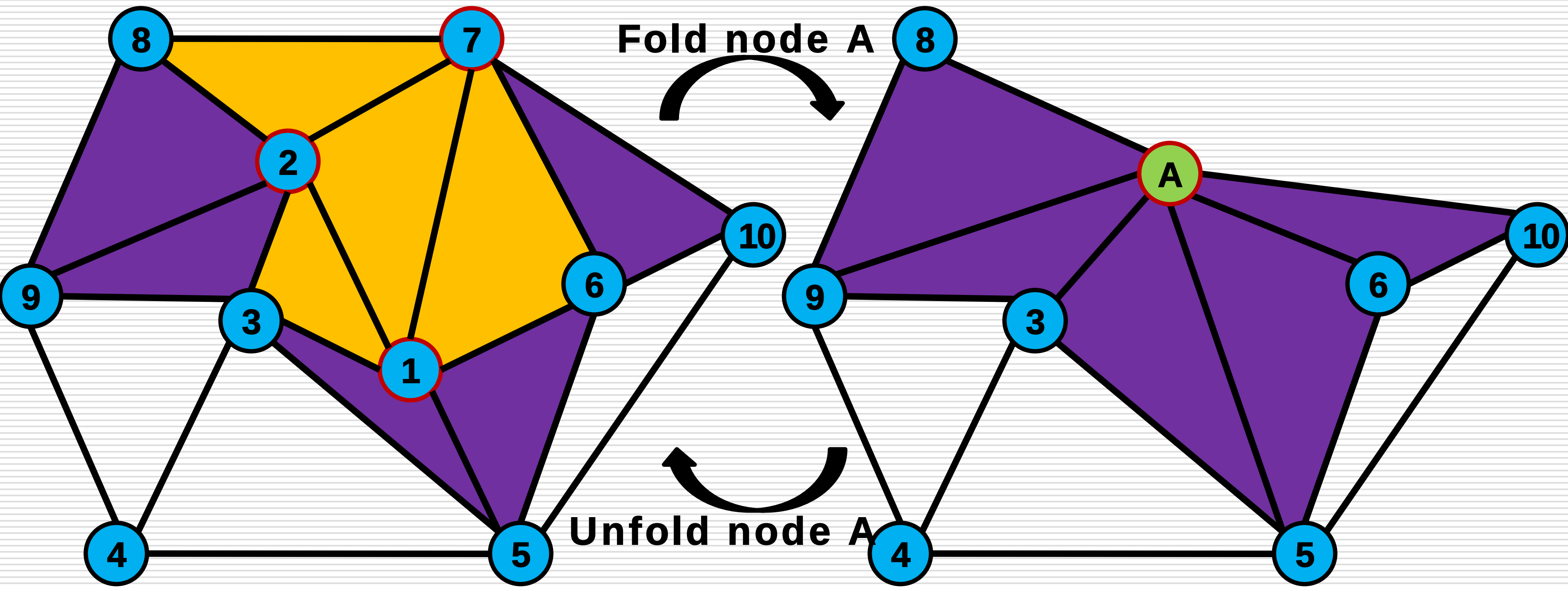


Vertex Tree

The Vertex Tree: Folding & Unfolding



The Vertex Tree: Tris & Subtris



Tris: triangles that change shape upon folding

Subtris: triangles that disappear completely

Level-of-detail Suggestion

- Apply progressive mesh for multi-resolution model generation
- Use in-game discrete LOD for performance tuning
- Why ?
  - For modern game API / platform, dynamic vertex update is costly on performance
  - Lock video memory stall CPU/GPU performance