Geometric Modeling

Summer Semester 2010

Triangle Meshes and Multi-Resolution Representations

Representations · Hierarchical Data Structures · Rendering

UNIVERSITÄT DES SAARLANDES

CLUSTER OF EXCELLENCE

max planck institut informatik

Overview...

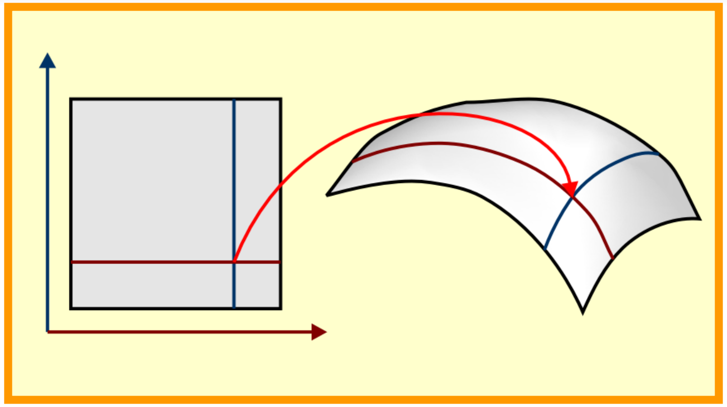
Topics:

- Blossoming and Polars
- Rational Spline Curves
- Spline Surfaces
- Triangle Meshes & Multi-Resolution Representations
  - Mesh Data Structures
  - Triangulations
  - Spatial Data Structures and Algorithms
  - Mesh Simplification
  - Appearance Approximation

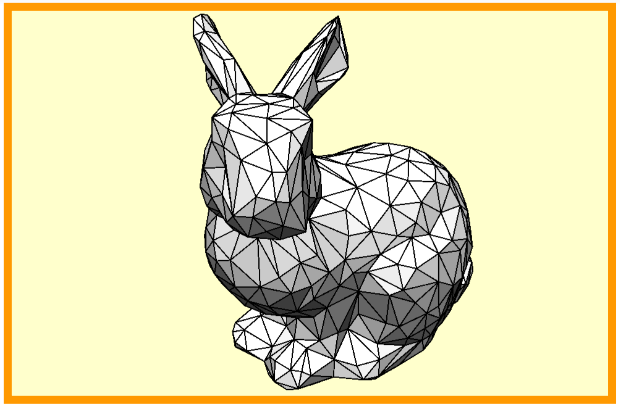
Triangle Meshes

Data Structures

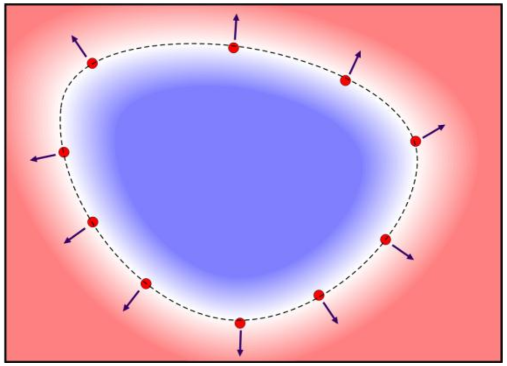
Modeling Zoo



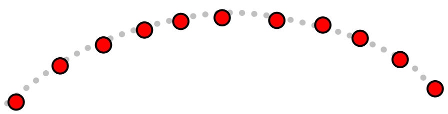
Parametric Models



Primitive Meshes



Implicit Models



Particle Models

Triangle Meshes

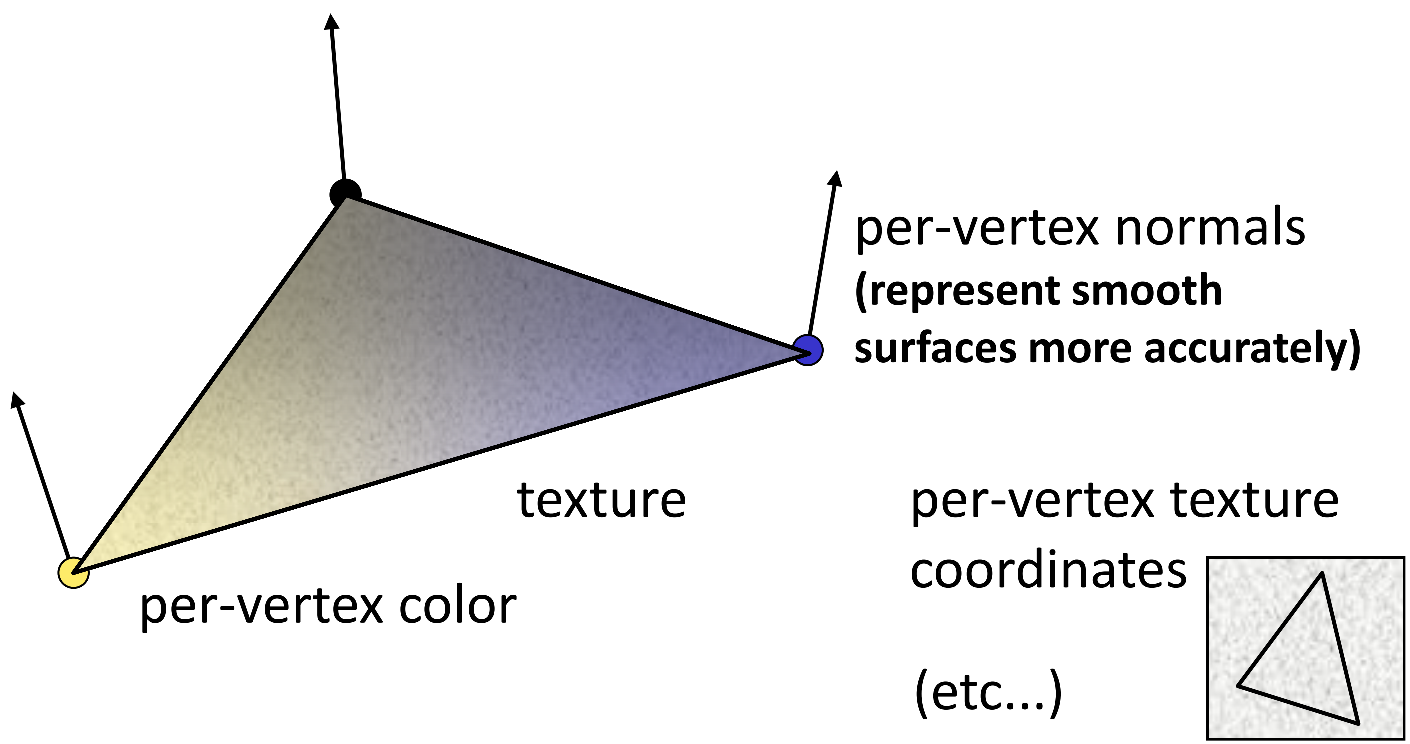
Triangle Meshes:

- Triangle meshes are probably the most common surface representation in computer graphics
- Triangles are probably the simplest surface primitives that can be assembled into meshes
  - Rendering can be implemented in hardware (z-buffering)
  - Simple algorithms for intersections (raytracing, collisions)

Attributes

How to define a triangle?

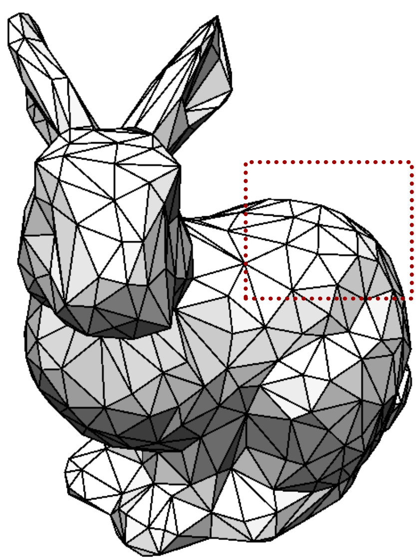
- We need three points in $\mathbb{R}^3$ (obviously).
- But we can have more:

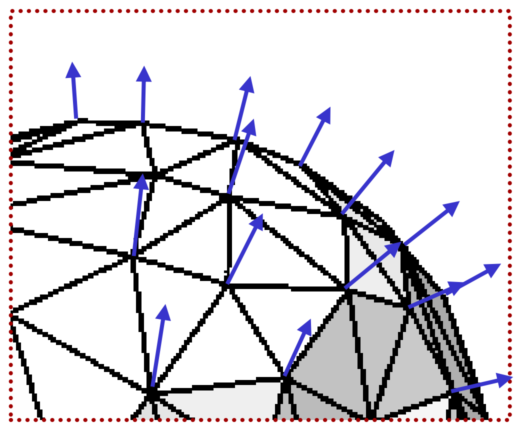


Shared Attributes in Meshes

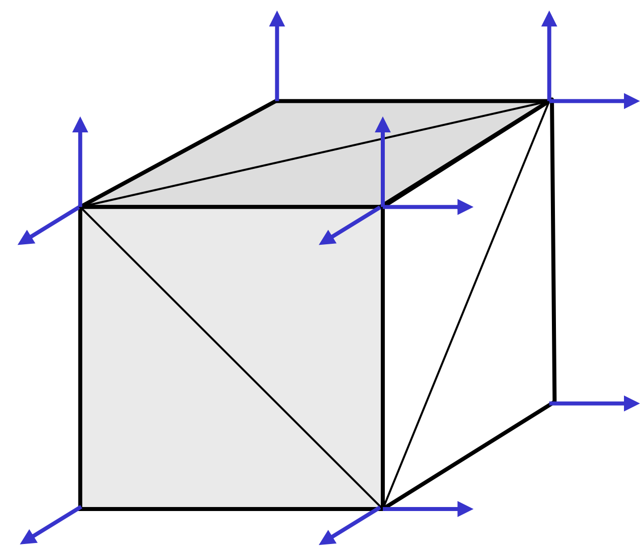
In Triangle Meshes:

- Attributes might be shared or separated:





adjacent triangles
share normals



adjacent triangles
have separated normals

“Triangle Soup”

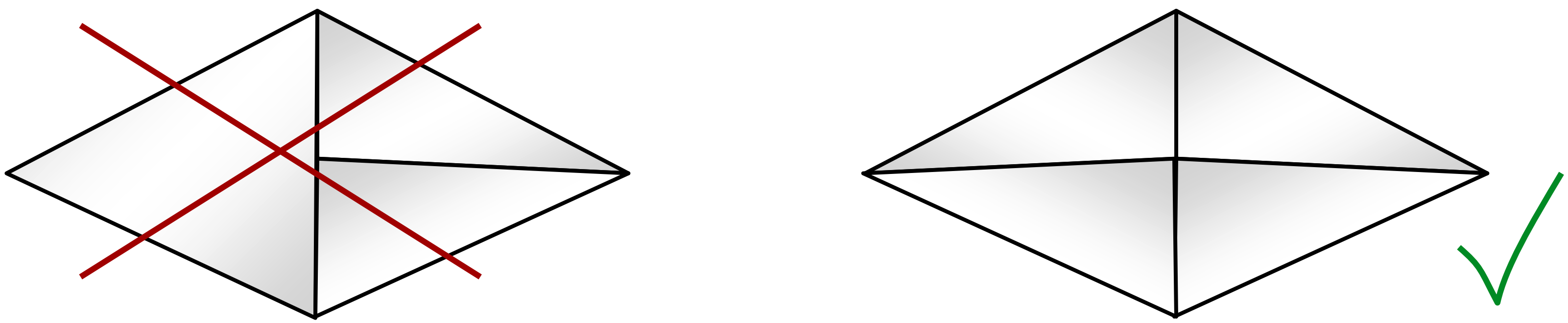
Variants in triangle mesh representations:

- “*Triangle Soup*”
  - A set $S = \{t_1, \ldots, t_n\}$ of triangles
  - No further conditions
  - This is “the most common” representation (if you download models from the web, you never know what you get)
- *Triangle Meshes*: Additional consistency conditions
  - Conforming meshes: Vertices meet only at vertices
  - Manifold meshes: No intersections, no T-junctions

Conforming Meshes

Conforming Triangulation:

- Vertices of triangles must only meet at vertices, not in the middle of edges:

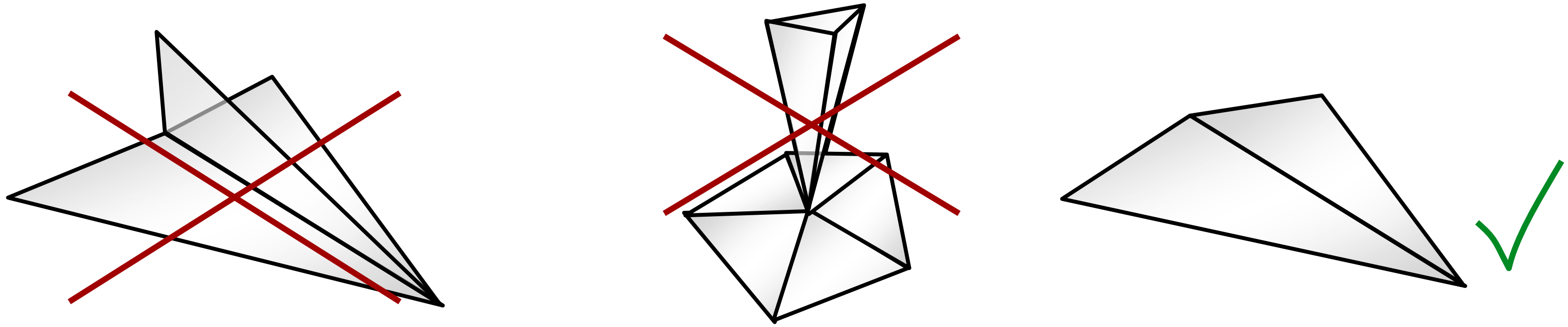


- This makes sure that we can move vertices around arbitrarily without creating holes in the surface

Manifold Meshes

Triangulated two-manifold:

- Every edge is incident to exactly 2 triangles
  (closed manifold)
- ...or to at most two triangles (manifold with boundary)
- No triangles intersect (other than along common edges or vertices)
- Two triangles that share a vertex must share an edge



Attributes

In general:

- Vertex attributes:
  - Position (mandatory)
  - Normals
  - Color
  - Texture Coordinates
- Face attributes:
  - Color
  - Texture
- Edge attributes (rarely used)
  - E.g.: Visible line

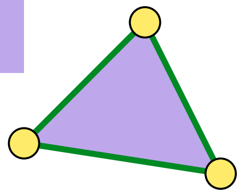
Data Structures

The simple approach: List of vertices, edges, triangles

v_1: (posx posy posy), $attrib_1$, ..., $attrib_{nav}$
...
v_{nv}: (posx posy posy), $attrib_1$, ..., $attrib_{nav}$

e_1: ($index_1$ $index_2$), $attrib_1$, ..., $attrib_{nae}$
...
e_{ne}: ($index_1$ $index_2$), $attrib_1$, ..., $attrib_{nae}$

t_1: (idx_1 idx_2 idx_3), $attrib_1$, ..., $attrib_{nat}$
...
t_{nt}: (idx_1 idx_2 idx_3), $attrib_1$, ..., $attrib_{nat}$



Pros & Cons

Advantages:

- Simple to understand and build
- Provides exactly the information necessary for rendering

Disadvantages:

- Dynamic operations are expensive:
  - Removing or inserting a vertex
    $\rightarrow$ renumber expected edges, triangles
- Adjacency information is one-way
  - Vertices adjacent to triangles, edges $\rightarrow$ direct access
  - Any other relationship $\rightarrow$ need to search
  - Can be improved using hash tables (but still not dynamic)

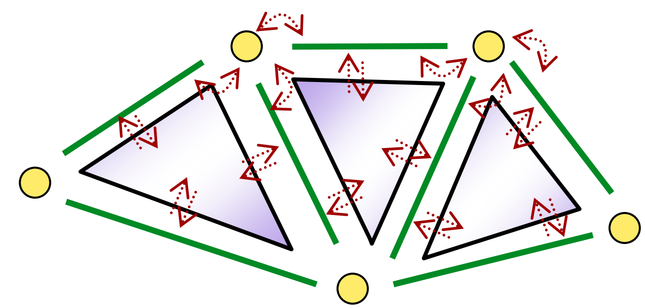
Adjacency Data Structures

Alternative:

- Some algorithms require extensive neighborhood operations (get adjacent triangles, edges, vertices)
- ...as well as dynamic operations (inserting, deleting triangles, edges, vertices)
- For such algorithms, an *adjacency based* data structure is usually more efficient
  - The data structure encodes the graph of mesh elements
  - Using pointers to neighboring elements

First try...

Straightforward Implementation:

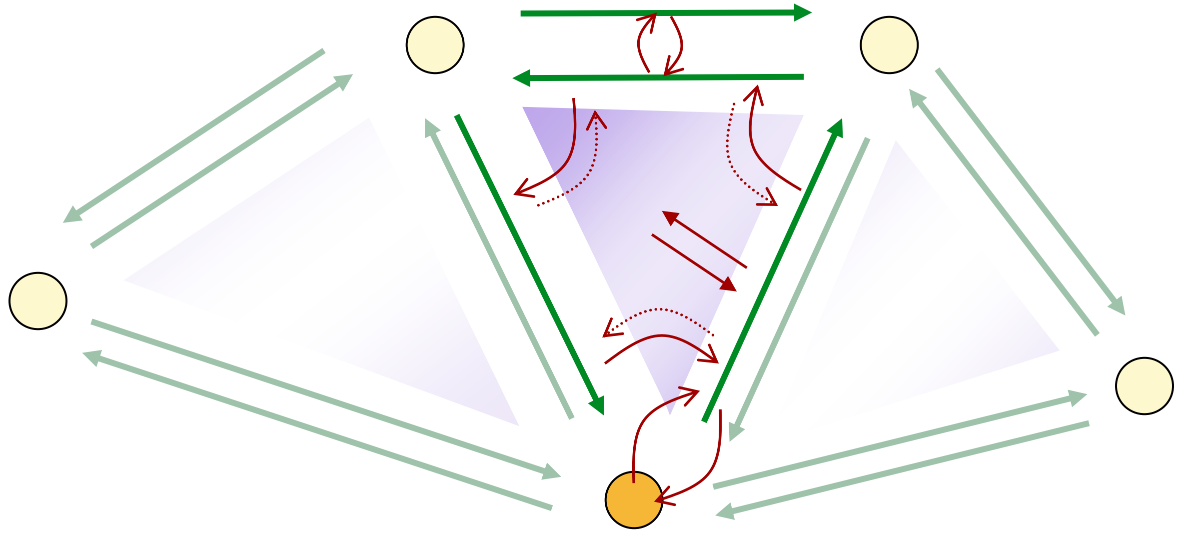


- Use a list of vertices, edges, triangles
- Add a pointer from each element to each of its neighbors
- Global triangle list can be used for rendering

Remaining Problems:

- Lots of redundant information – hard to keep consistent
- Adjacency lists might become very long
  - Need to search again (might become expensive)
  - This is mostly a “theoretical problem” (O(n) search)

Less Redundant Data Structures



Half edge data structure:

- Half edges, connected by clockwise / ccw pointers
- Pointers to opposite half edge
- Pointers to/from start vertex of each edge
- Pointers to/from left face of each edge

Implementation

```
// a half edge
struct HalfEdge {
   HalfEdge* next;
   HalfEdge* previous;
   HalfEdge* opposite;

   Vertex* origin;
   Face* leftFace;
   EdgeData* edge;
};

// the data of the edge
// stored only once
struct EdgeData {
   HalfEdge* anEdge;
   /* attributes */
};
```

```
// a vertex
struct Vertex {
   HalfEdge* someEdge;
   /* vertex attributes */
};

// the face (triangle, poly)
struct Face {
   HalfEdge* half;
   /* face attributes */
};
```

Implementation

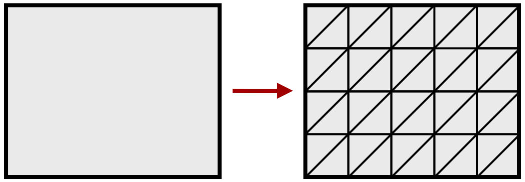
Implementation:

- The data structure should be encapsulated
  - To make sure that updates are consistent
  - Implement abstract data type with more high level operations that guarantee consistency of back and forth pointers
- Free Implementations are available, for example
  - OpenMesh
  - CGAL
- Alternative data structures: for example winged edge (Baumgart 1975)

Triangulations

Algorithms and Data Structures

Triangulation



Problem Statement:

- Given a 2-dimensional domain
- We want to triangulate the domain
- We need this for example for rendering parametric surfaces by triangle rasterization
- Adaptive triangulation: Higher resolution in more important area

Different Problem:

- Triangulating a point cloud in $\mathbb{R}^3$
- This is the surface reconstruction problem (we will look at that later)

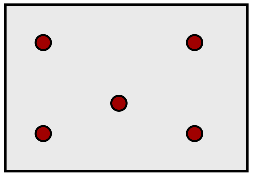
Problem Variations

Simplest Version

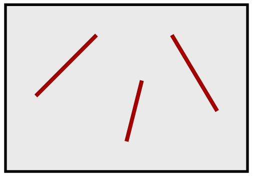
- Domain is a rectangle or a triangle
- Uniform or adaptive tessellation

More Complex: Constrained Triangulation

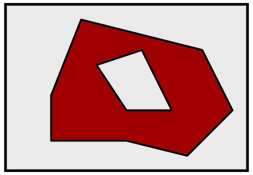
- Point constraints:
  specific points must be included



- Edge constraints:
  specific edges must be included



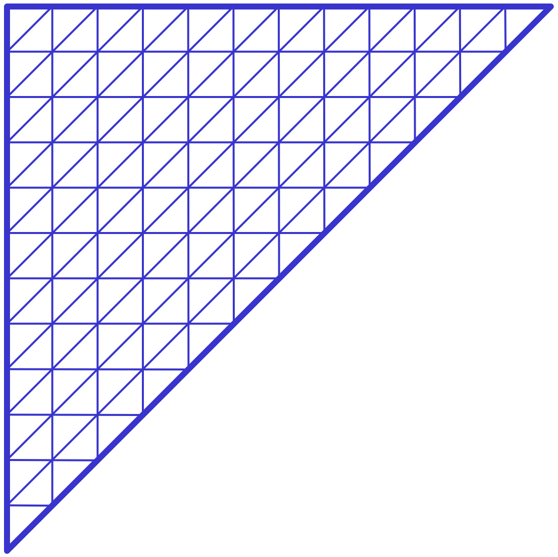
- Boundary constraints:
  triangulate within some area only

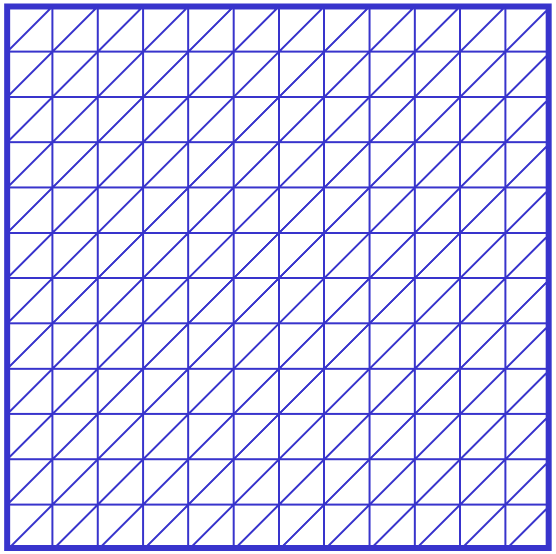


Unconstrained Uniform Triangulation

Unconstrained uniform triangulation:

- This is simple

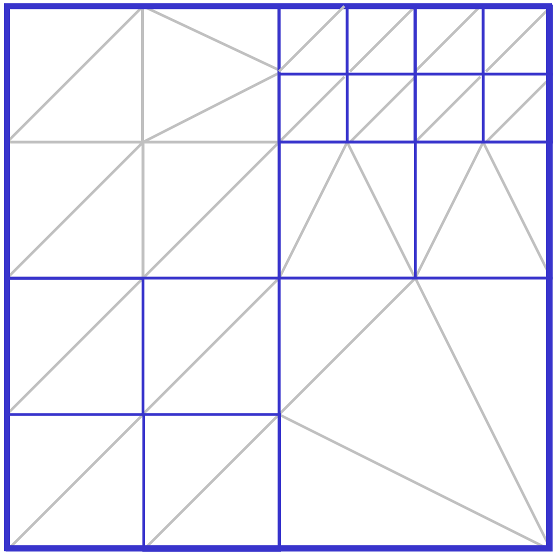


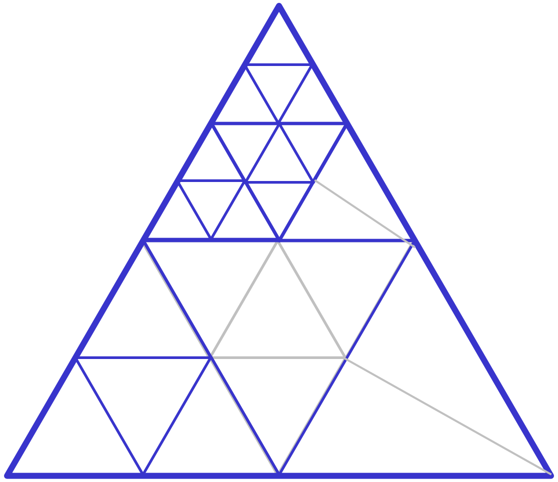


Adaptive Triangulation

Unconstrained adaptive triangulation:

- Hierarchy of rectangles / triangles (Quadtree)
- Use “balancing” to limit depth differences
- Balancing will increase the number of nodes in the tree by a factor of at most O(1)
- Finally, create a conforming triangulation (fixed number of cases per node)

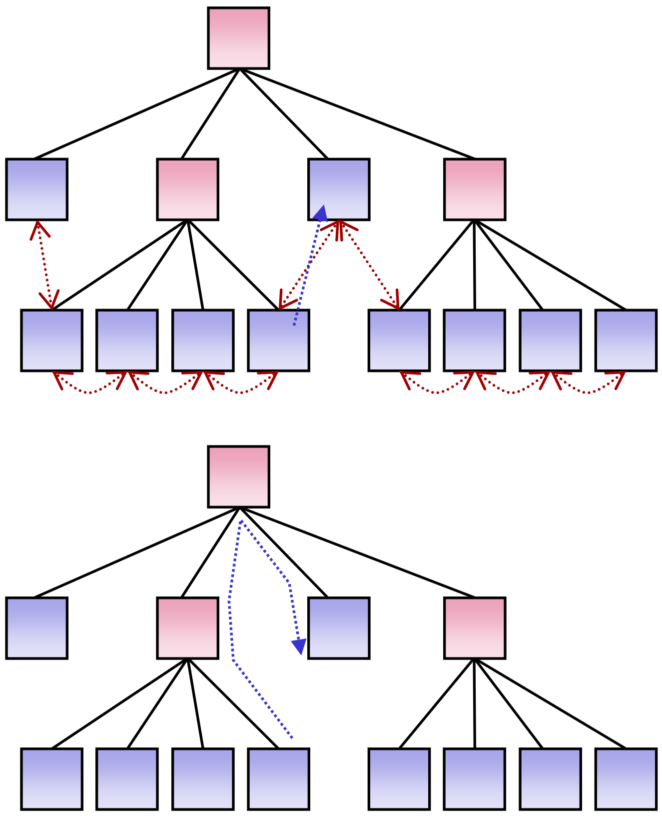




Implementation

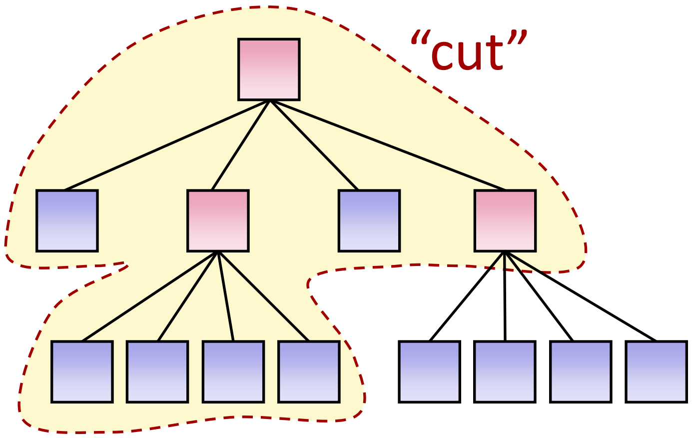
Storage: Tree Structure

- Tree can be represented directly
- Neighbor search for balancing:
  - We can store fixed pointers to neighboring cells
    (not that elegant, easy to mess up the consistency)
  - Alternative: use neighborhood search
    - Go up in tree until common ancestor is found
    - Then go down again
    - O(1) expected running time



Adaptive Rendering

Adaptive rendering algorithm

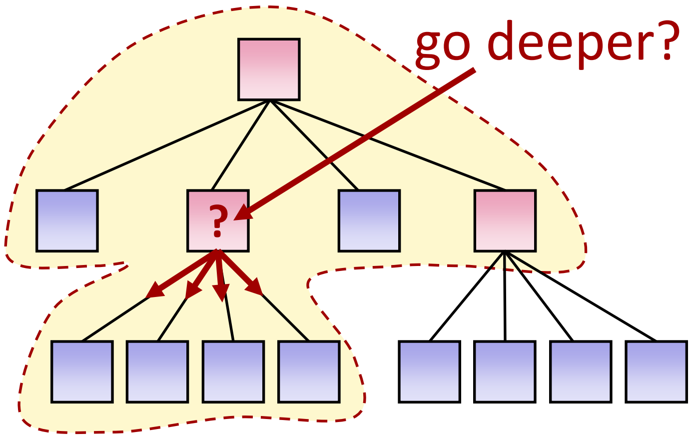


- Recursive algorithm
- Starts at root node
- Is precision sufficient?
  - If so → stop recursion
  - Otherwise → go to child nodes
- The recursion extracts a subgraph of the tree ("cut")
- Next: The subgraph needs to be balanced
- Then, a triangulation can be created

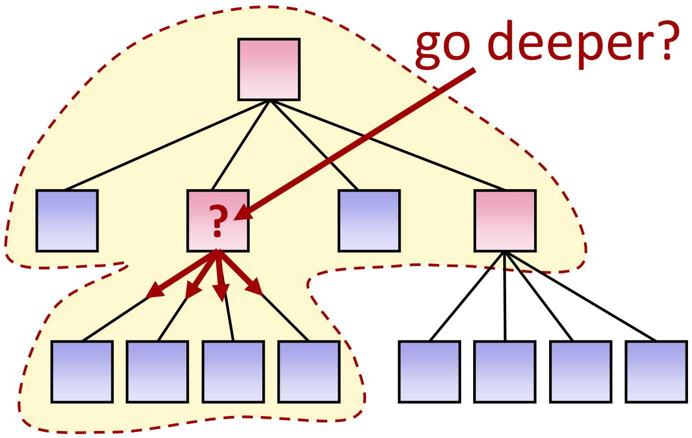
Adaptive Rendering

Termination Criteria:

- Rendering error:
  - Projected size on screen shrinks with $1/z$ (where z is the depth in camera coordinates)
  - Might also depend on viewing angle (typically, this is neglected)
- Geometric error:
  - Tessellating a curved surface with planar faces is only an approximation
  - Error depends on curvature



Adaptive Rendering



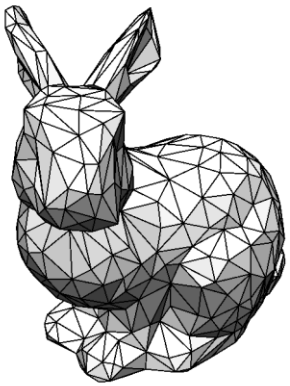
Termination Criteria:

- Typically: divide geometric error by z
- To estimate z, use a bounding box (for splines: convex hull property)
- Chooses nearest z (conservative estimate)
- REYES algorithm [Cook, Carpenter, Catmull 1987] (Pixar's RenderMan)
  - Stop subdivision when BB below one pixel on screen size
  - Subdivision connectivity not really necessary in that case

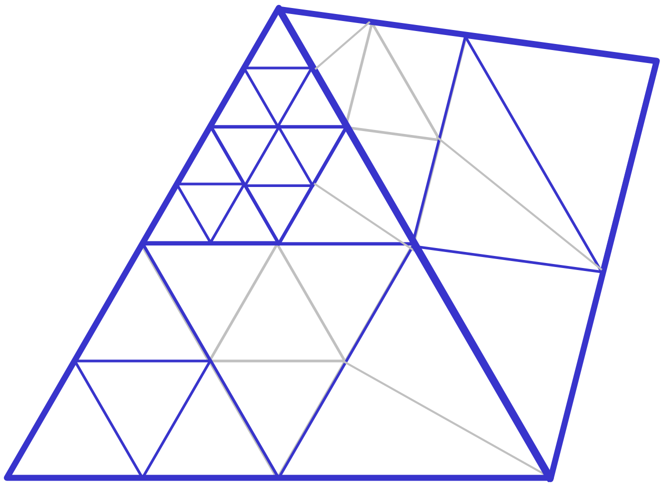
Subdivision Connectivity Meshes

Generalization: Arbitrary Domains

- Start with a base mesh
  - "3D parametrization"
  - A conforming two-manifold mesh in 3D used as parametrization domain
- The base mesh fixes the topology
- Subdivide recursively as needed
- Now: Balancing/triangulation, *also across borders*
- Then compute the final surface



base mesh



consistency across boundaries

Hardware Friendly Version

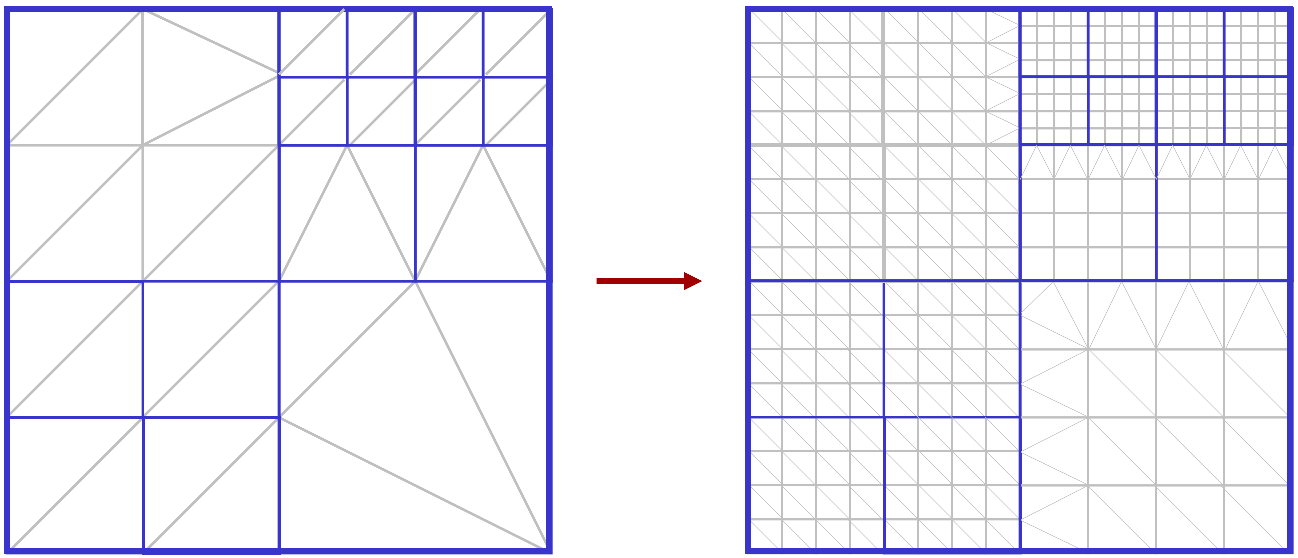
Problems:

- Costs for hierarchy creation / balancing are quite large
- In particular: Problematic for rendering
- Rendering triangles is very cheap these days
- But we still need adaptivity (moving camera, we can get arbitrarily close)
- Solution: *Subdivision connectivity grids*

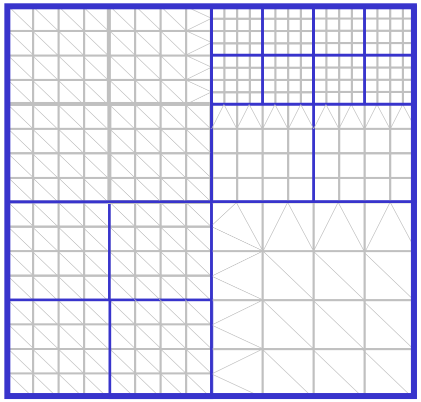
Subdivision Connectivity Grids

Idea:

- Do the same thing (hierarchical triangulation)
- But use a grid of many triangles in each node:



Subdivision Connectivity Grids



Advantage:

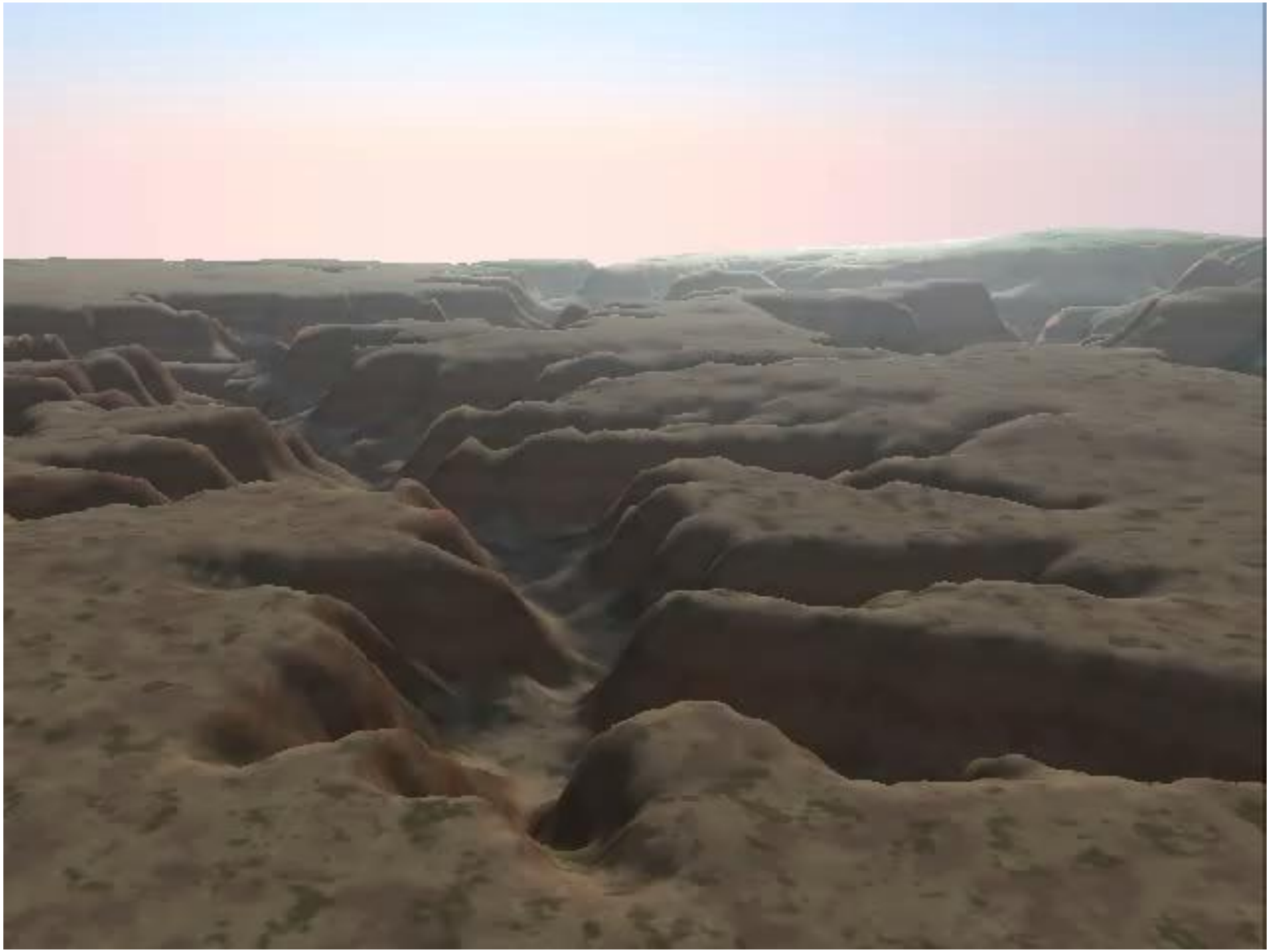
- Amortizes hierarchy creation / traversal costs over many triangles
- Well suited for graphics hardware (GPU) implementations (regular structure)

Disadvantage:

- Less adaptivity
- This is ok for the $1/z$ term in perspective rendering (we will see that later)
- But geometry will be oversampled

Example

Example



Example

Example

Constraint Triangulations

Additional Constraints:

- Vertices, edges, area
- Need to augment subdivision algorithm

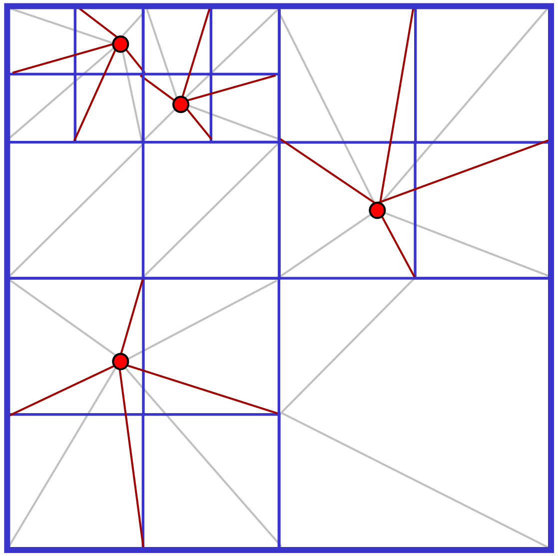
Hierarchical Subdivision:

- Subdivide until a simple case is found
  - At most one vertex in each cell
  - At most one line segment intersecting each cell
  - At most two boundary / cell intersections
- Then triangulate according to fixed rules

Vertex Constraints

Vertex Constraints:

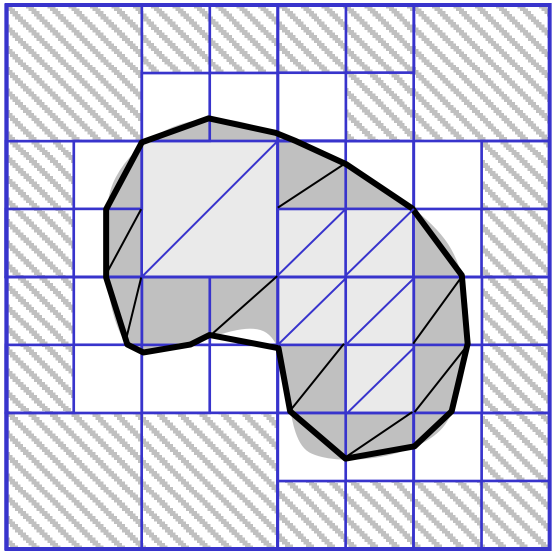
- When only one point is left in each box
- Subdivide once more
- Move center to point
- Then balance and triangulate (proceed as before)



Edge / Area Constraints

Edge and area constraints

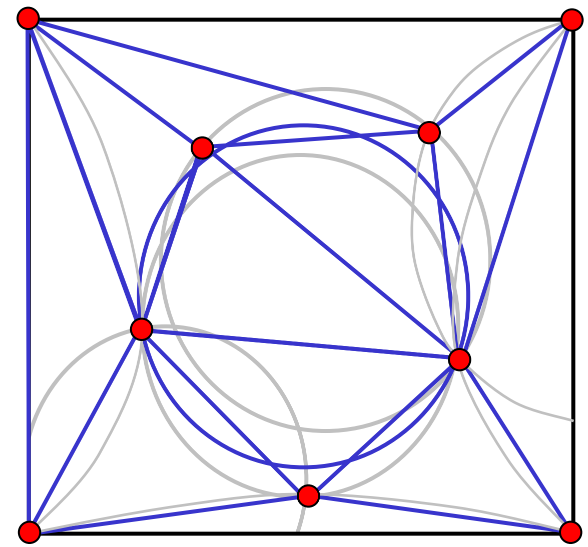
- Subdivide until intersection with edges / boundary curves has constant complexity (e.g. two intersections per cell)
- Then apply fixed subdivision rule
- Edge constraints:
  - Keep all triangles
- Area constraint:
  - Delete outside triangles



Alternative Algorithm

Alternative: (constrained) Delaunay triangulation

- Delaunay triangulation of a point set:
  - Triangulation in which the circumcircle of each triangle is empty
  - This triangulation *maximizes* the *minimum angle* in any triangle
  - The triangulation always exist
  - Can be computed by iterated edge flipping or (more efficiently) by line sweep algorithms ($O(n \log n)$ time for n points)



- Constrained Delaunay triangulation:
  - Additional edge / polygonal area constraints
  - More involved to compute

Spatial Data Structures

Range Queries, Collision Detection

Spatial Data Structures

Motivation:

- Common problems:
  - Select a handle point by mouse click (millions of handles)
  - Click on other stuff (edges, triangles, patches)
  - Find the nearest point in a point set
  - Find the *k* nearest points (e.g. for surface fitting)
  - Find all geometry within a range (cube, sphere, etc.)
- This should work on large models
  - Billions of primitives
  - Frequent operations
    - E.g.: compute 20 nearest points for 1.000.000 points
    - Quadratic runtime is unacceptable
- Such operations can be speed up tremendously using spatial indexing data structures

Spatial Data Structures

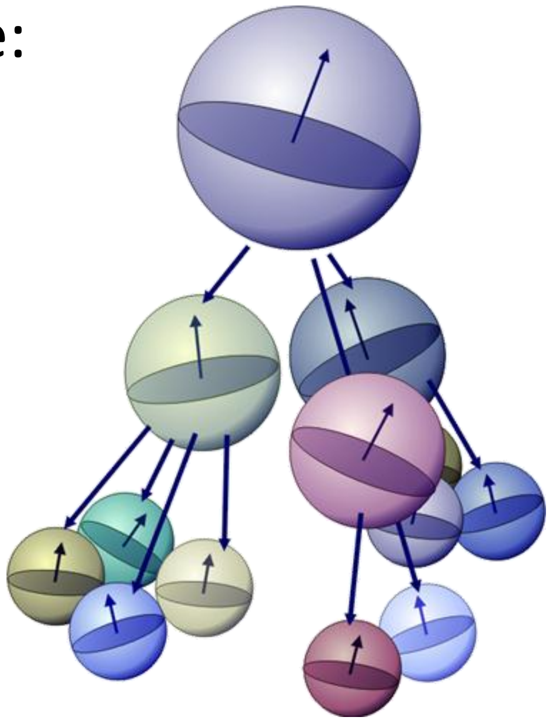
Basic Idea: Hierarchical decomposition of space

- Almost all approaches commonly used in practice are based on hierarchical spatial decompositions
- For some problems, there are more sophisticated data structures from computational geometry, but they often have to large space requirements
- In practice, anything beyond linear space is out of question

Spatial Data Structures

Basic Idea: Hierarchical decomposition of space

- If the number of objects is still too large:
  - Cluster geometry into a small number of spatially coherent groups
  - Compute a simple bounding volume for each group
  - Apply this principle recursively to all subgroups formed
- We obtain a tree of bounding volumes



Hierarchical Space Partitioning

Formally:

- We have a set of objects $\Omega = \{s_1, ..., s_n\}$, $s_i \subseteq \mathbb{R}^d$ (where d is small, usually d = 2..3)
- We form a hierarchy of nodes N_i.
  - Let C(N_i) be the set of child nodes, ...
  - ...and P(N_i) the unique parent node, or *null*, if N_i is the root node R.
- We associate a set of objects S(N_i) with each node N_i.
- We demand S(R) = Ω (root contains everything) and $N_j \in$ C(N_i) $\Rightarrow$ S(N_j) $\subseteq$ S(N_i) (inner nodes represent the whole subtree)

Hierarchical Space Partitioning

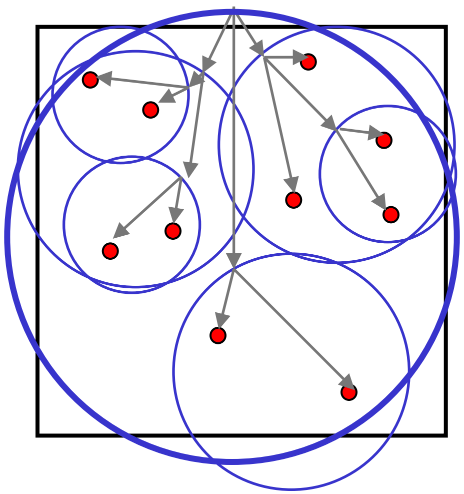
Formally:

- Bounding volumes: let $B(N_i)$ be a bounding volume of node N_i, $B(N_i) \subseteq \mathbb{R}^d$.
- This means: $S(N_i) \subseteq B(N_i)$
  (objects are contained in the bounding volume)
- Typically, a bounding volume is a much simpler object than the stored geometry $S(N_i)$.
  - It should be easy to test for intersections with other bounding volumes, geometric ranges and objects to be sorted into the hierarchy.
  - Usually, the memory footprint of $B(N_i)$ is $O(1)$.
  - Axis aligned boxes, spheres and the similar are popular.

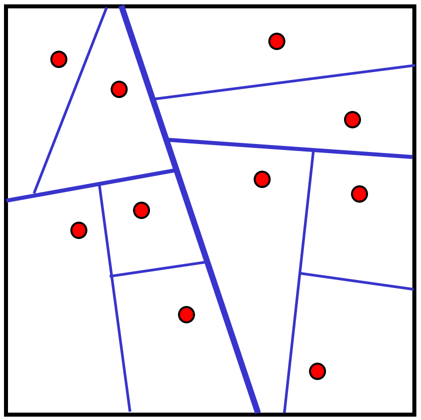
Variants

Variants:

- Bounding volume hierarchy
  - Most general definition, we can use any bounding volumes
  - Each inner node represents the union of objects in the subtrees



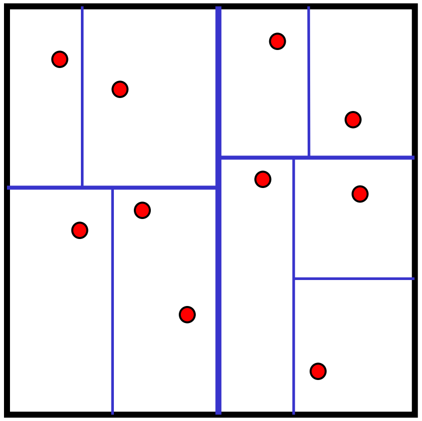
- BSP-tree
  - Use planes to split the nodes into half-spaces
  - Usually stored as a binary tree (“binary space partition”)
  - Cells are not O(1), but each tree level cuts of a half space, which can be tested incrementally.



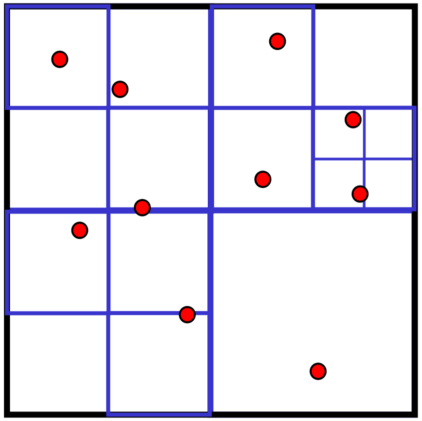
Variants

Variants

- kD-tree / axis aligned BSP tree
  - Use axis parallel splitting planes
  - Special case kD-tree:
    - Cyclically alternating splitting dimensions
    - Use median cut



- Quadtrees / Octrees
  - Always divide into 4 (8) cubes of the same size
  - This is a special case of a BSP- / kD-tree (identifying 3 consecutive binary splits with one octree node)



Extended Objects

Construction for extended objects (other than points)

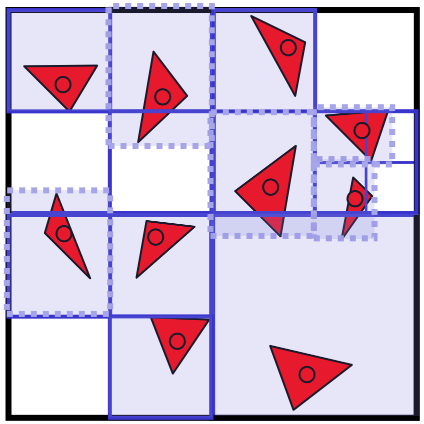
- Extended objects:
  - Triangles
  - Polygons
  - Patches
  - Line segments
  - etc...
- Division of space might intersect with object
- Two solutions
  - Splitting objects
  - Overlapping nodes

Splitting Objects

First solution: splitting objects

- For example, sorting triangles into a BSP tree:
  - Split each triangle along splitting plane, if necessary
  - Try to optimize such that as few as possible triangles are split
- (Rather) easy to see:
  - A BSP tree needs at least worst case $O(n^2)$ fragments for n triangles (in practice typically $\approx O(n \log n)$)
  - This is worst-case quadratic storage
  - The same bound also applies to kD trees, octrees etc (special cases)
- Splitting objects is usually too expensive
  - Used in early low-polygon 3D engines for visibility computation

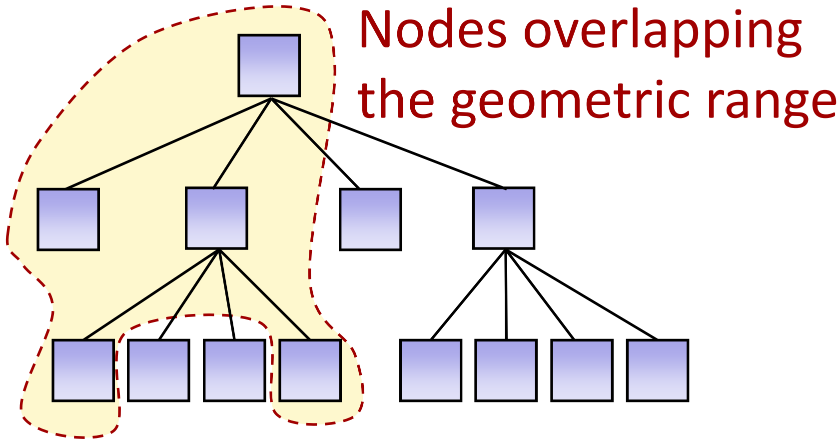
Overlapping Regions



Other alternative:

- Allow objects to exceed the region associated with each node
- Store a second, extended bounding box to reflect this information
- Typical strategy:
  - Allow up to 10% oversize (exceeding node limits by 10% in each direction)
  - If this does not fit into leaf nodes, use an inner node.
- Effective bounding volumes may overlap now
  - Limiting the percentage limits the amount of space covered multiple times (e.g. 10% in each direction means $1.2^3 \approx 1.7\times$)

Range Query Algorithm

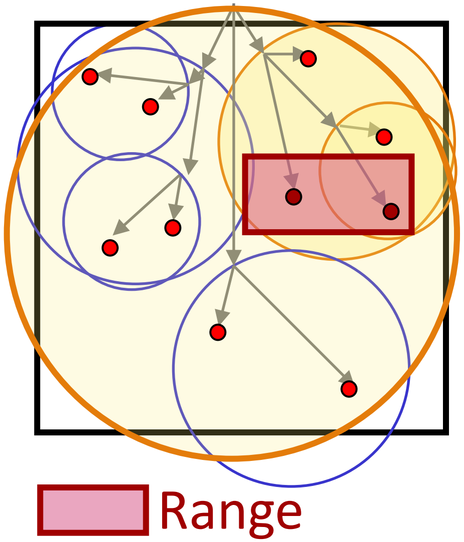


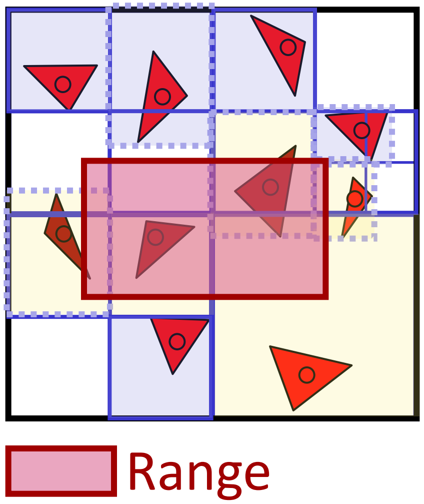
Start at root node: Then, recursively

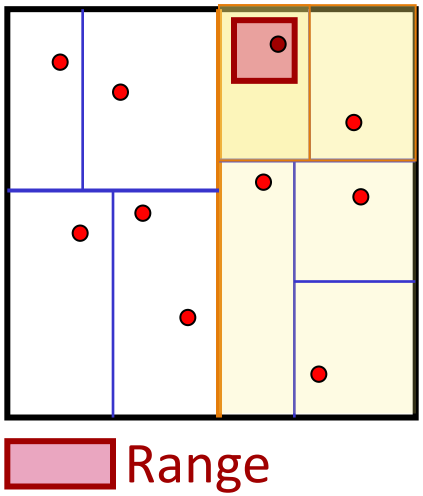
- If range overlaps bounding box
  - Collect inner node primitives
  - Test for range intersection
  - Go on recursively for child nodes
- If range does not overlap bounding box
  - End recursion

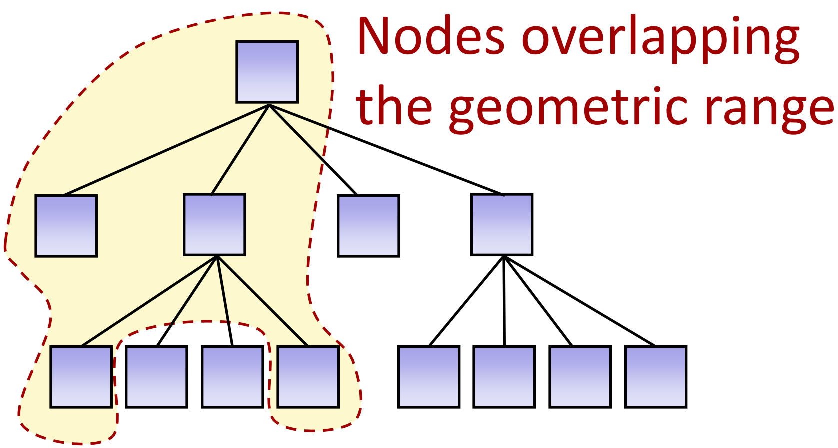
works for all
hierarchy types

Examples









Parametric Surfaces

In case every primitive itself is a parametric object:

- We can “continue” the hierarchy
- Use a regular subdivision of the parameter domain (binary splits, quadtree)
- Form bounding volumes dynamically (e.g. convex hull of subdivided control points)

Abstract Implementation

Geometric Ranges:

- We just need to define two methods:
  - Intersection primitive $\leftrightarrow$ range
  - Intersection bounding volume $\leftrightarrow$ range
- With this information, we can implement a generic hierarchical range search algorithm
- Important special cases:
  - Boxes, Spheres, etc...
  - Rays (raytracing)
  - Projective extrusions (2D curve extended into space by central projection; this can be used for drawing selection regions on screen and retrieving the corresponding objects)

Collision Detection

Related Problem: Collision Detection

- We want to compute whether two geometric objects intersect with each other
- Important problem for dynamic simulations
- Also useful for CAD applications (arrange objects that do not collide)

Simple Solution:

- Test every part of object A for collision with every part of object B (e.g. each triangle with each other triangle)
- This is usually to expensive [O(mn)]

Hierarchical Collision Detection

Hierarchical Collision Detection

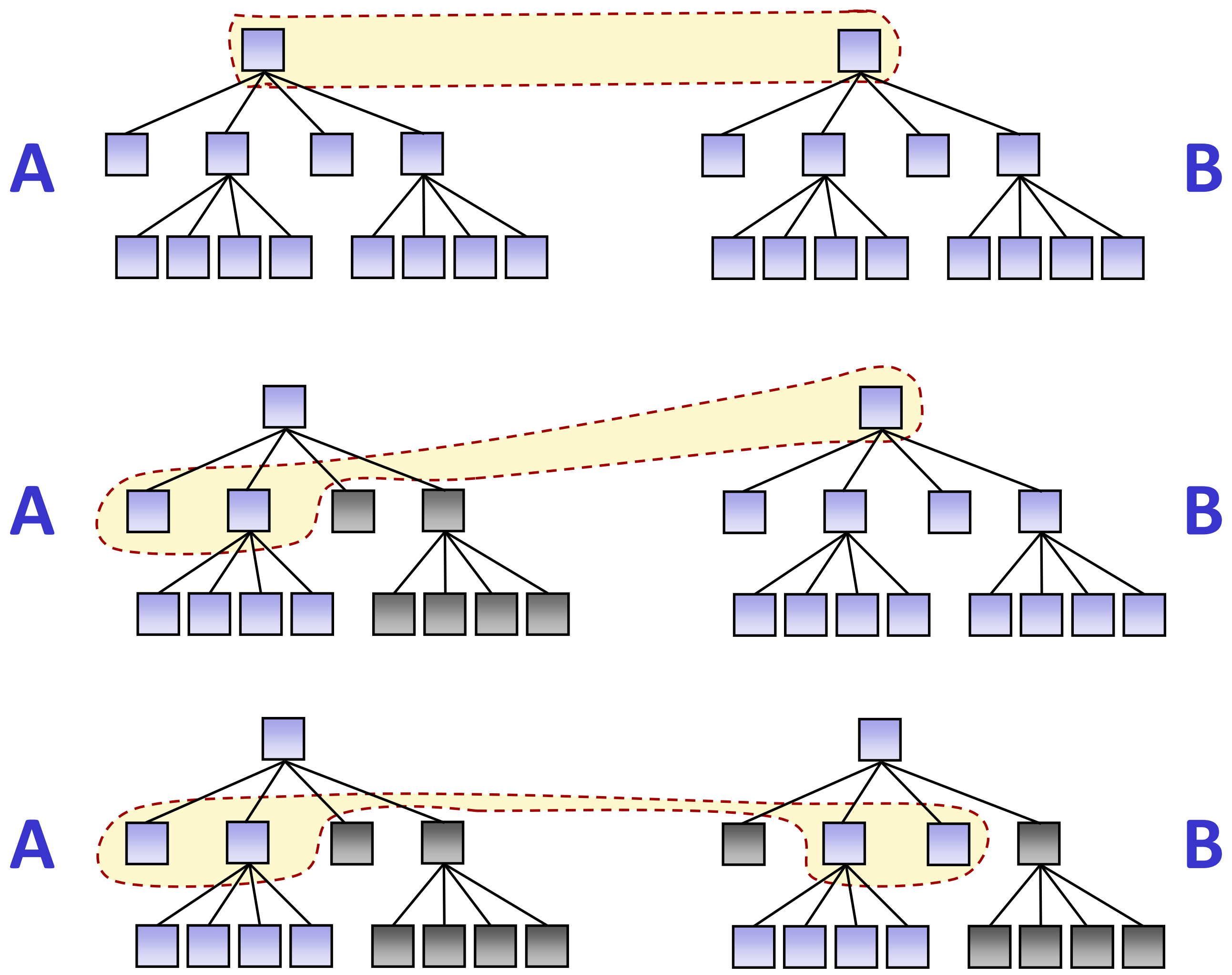
- Precompute a hierarchy for both objects A and B that should be tested for collision.
- Then apply a hierarchical collision test (next slide)

Hierarchical Collision Test

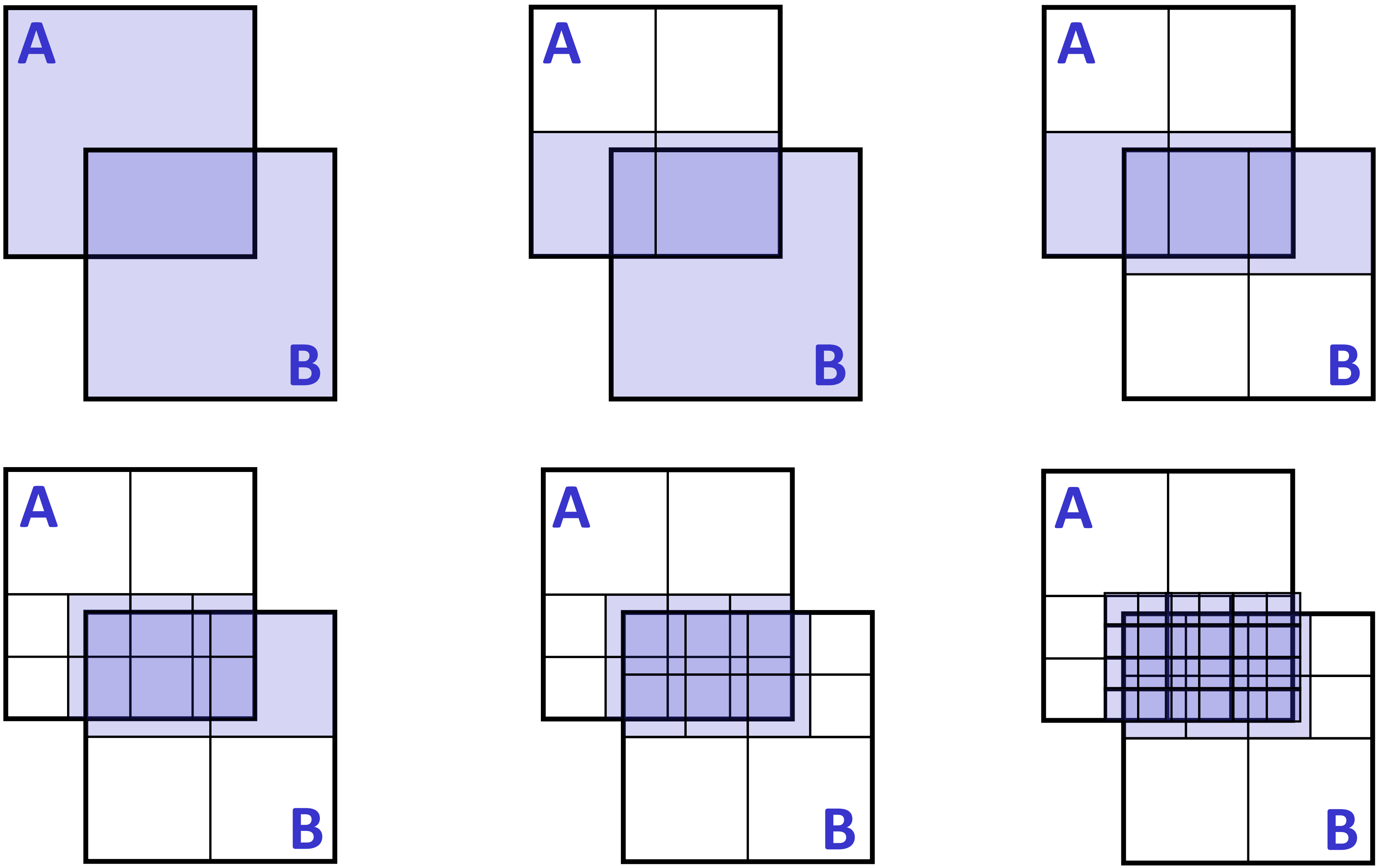
Collision Test: Input – nodes N_A, N_B from objects A, B.

- Test bounding volumes $\mathrm{B}(N_A)$, $\mathrm{B}(N_B)$ for intersection
- **If** $\mathrm{B}(N_A) \cap \mathrm{B}(N_B) \neq \emptyset$**:**
  - Test all objects $\mathrm{S}(N_A)$, $\mathrm{S}(N_B)$ for intersection
  - Output those objects that do intersect
  - **If** diameter($\mathrm{B}(N_A)$) > diameter($\mathrm{B}(N_B)$)**:**
    - For all children $C \in \mathrm{C}(N_A)$
      - CollisionTest(C, N_B)
  - **Otherwise:**
    - For all children $C \in \mathrm{C}(N_B)$
      - CollisionTest(C, N_A)

Illustration



Illustration



Parametric Objects

Collision of parametric objects:

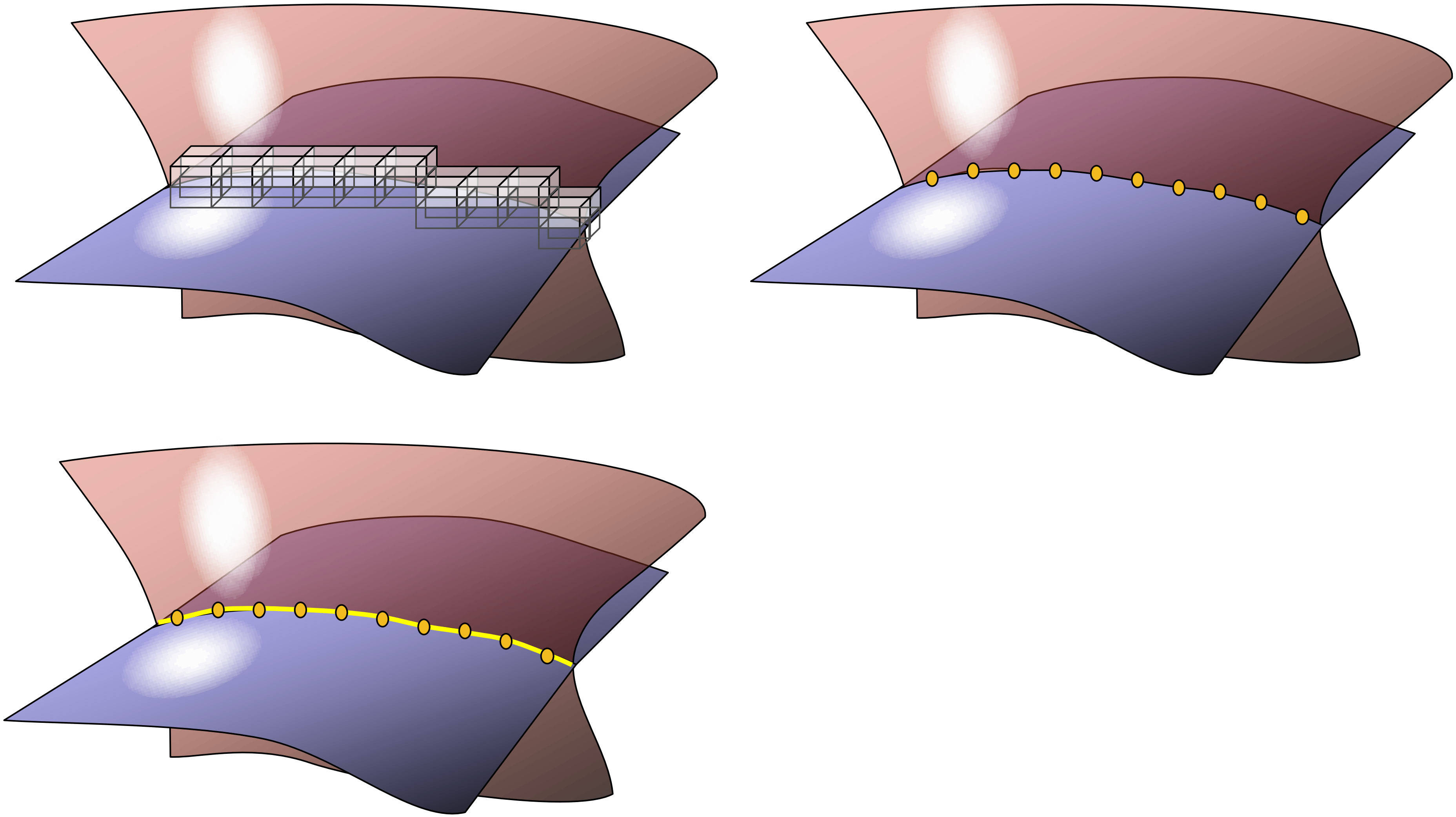
- Again, we can "continue" the hierarchy in the parametric domain
- Useful for speeding up patch-patch collision detection
- We can also compute intersection lines hierarchically

Parametric Objects

Computing intersection lines:

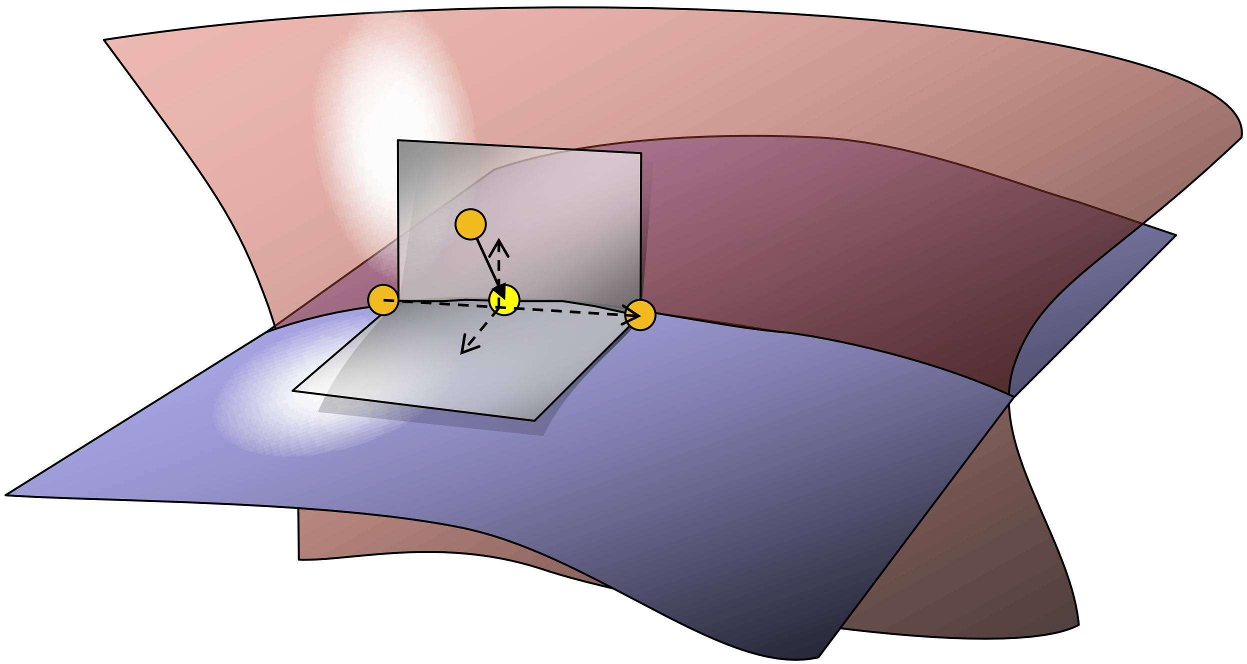
- Hierarchical intersections until a number of small boxes is left
- Place a control point in each box
- Use a Newton iteration to project points on intersection line
  - Move points in direction orthogonal to line only (avoid degeneracies)
- Fit a spline through the control points (spline interpolation problem, linear system)
- Can be additionally constrained to lie on intersection line
  - Minimize integral residual of distances to patches
  - But this is a non-linear optimization problem (Newton solver)

Intersection lines



Projecting a Point

Quasi-Newton Scheme



Nearest Neighbor Queries

Problem:

- Given *n* objects s_i and a point **p** in space
- Two variants:
  - Find the object that is closest to **p**
  - Find the *k* closest objects (*k*-nearest neighbors, kNN)

Operations:

- Compute distance point $\leftrightarrow$ primitive
- Compute distance point $\leftrightarrow$ bounding volume

Hierarchical Query Algorithm

Data Structures:

- The query algorithm needs some bounding volume hierarchy for the objects
  - A kD tree works best in practice, but other data structures also do the job
- In addition, two auxiliary data structures are needed:
  - A priority queue of objects Q_{obj}
  - A priority queue of bounding volumes Q_{BB}
  - Both sorted by distance to the query point

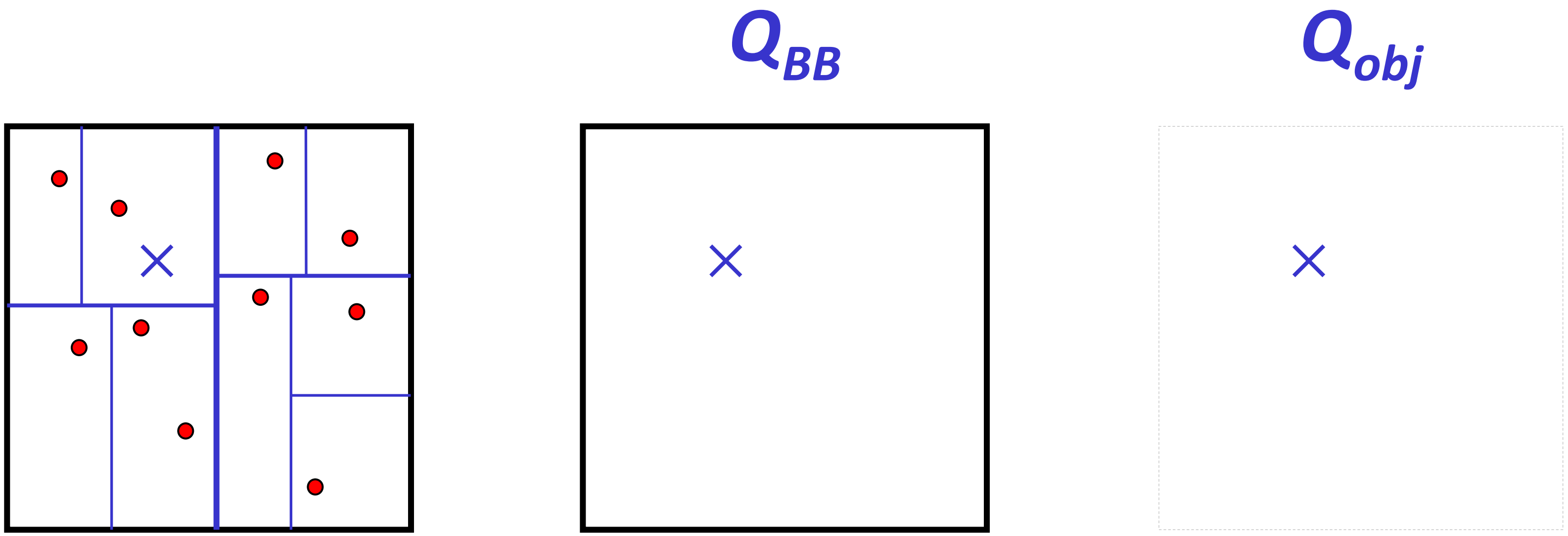
Hierarchical Query Algorithm

Algorithm: Compute k nearest neighbors

Input: Hierarchy of objects N, query point $\mathbf{p}$

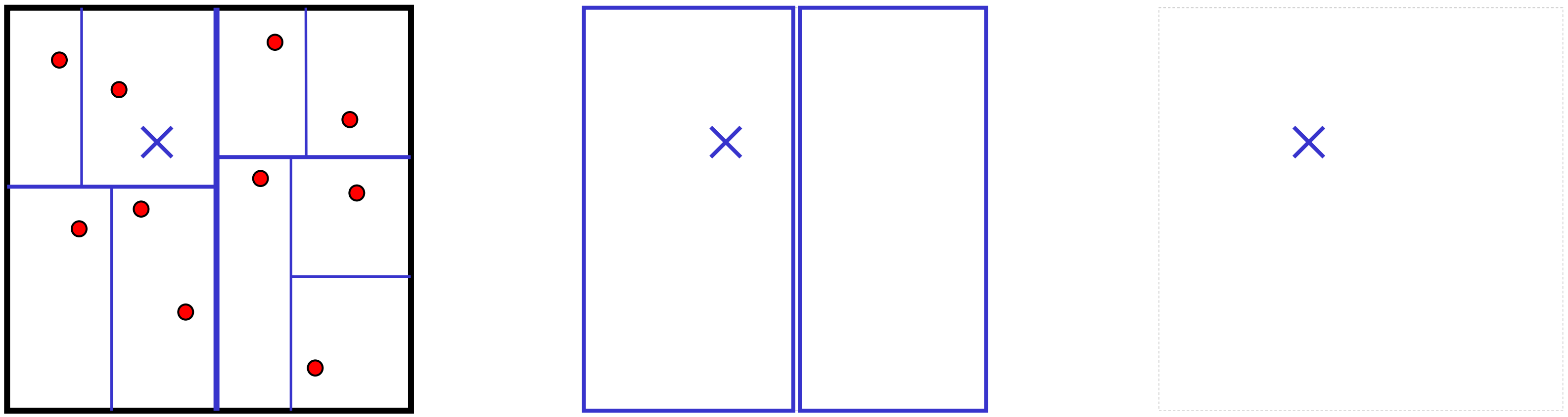
- **Initialization:** Put root node on Q_{BB}
- **While** *#output* < k **and** both priority queues non-empty
  - Compute distance to min(Q_{BB}) and min(Q_{obj})
  - **If** an object is closer
    - output the object
  - **Otherwise, if** a box is closer
    - Take the box from the queue
    - Insert all objects into Q_{obj} and all child nodes into Q_{BB} (for this, the corresponding distances need to be computed)

Illustration

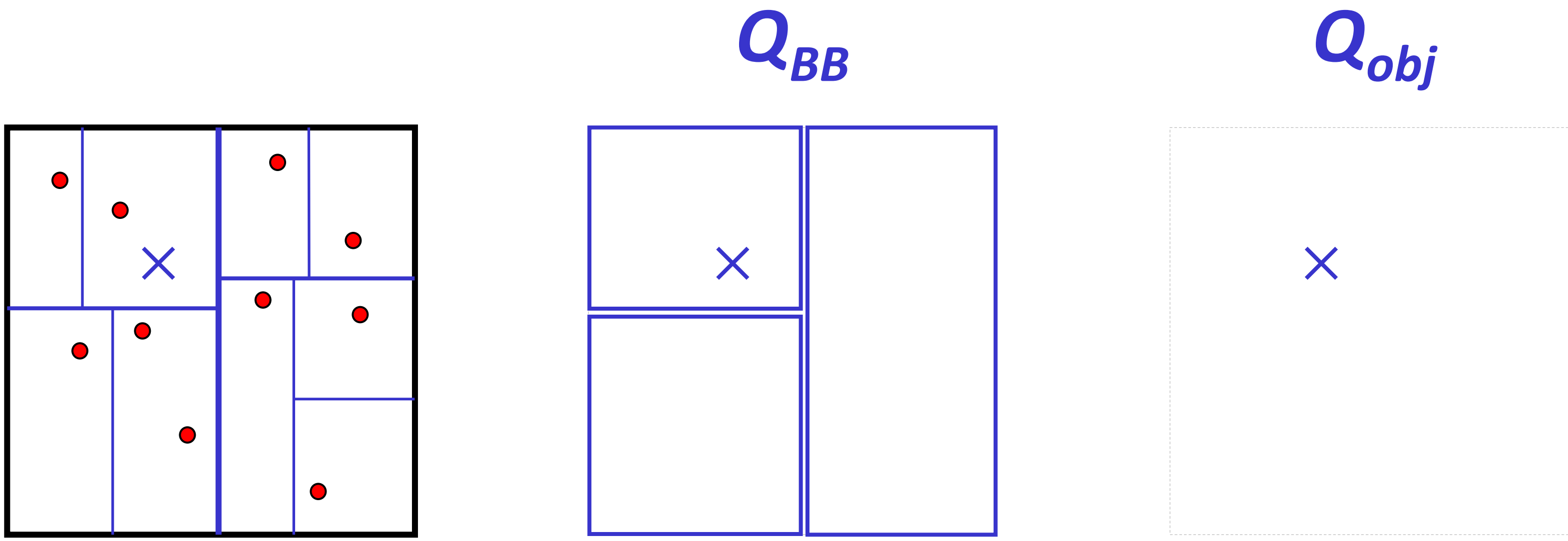


Illustration

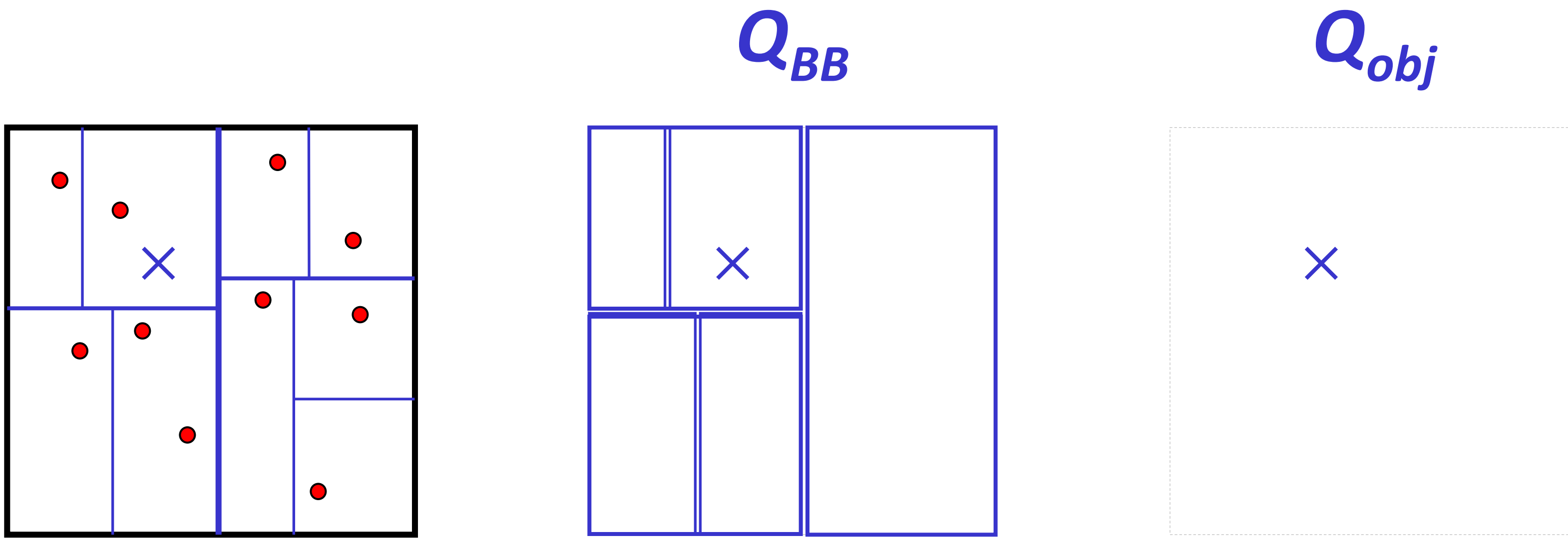
Q_{BB} Q_{obj}



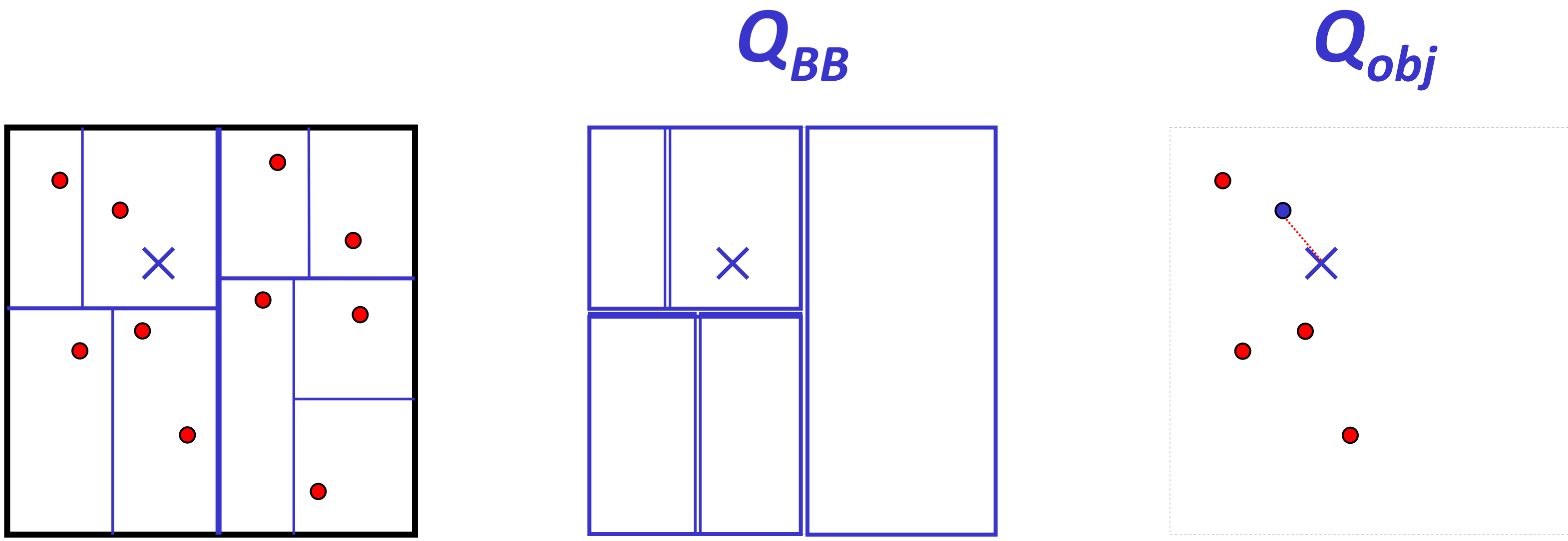
Illustration



Illustration

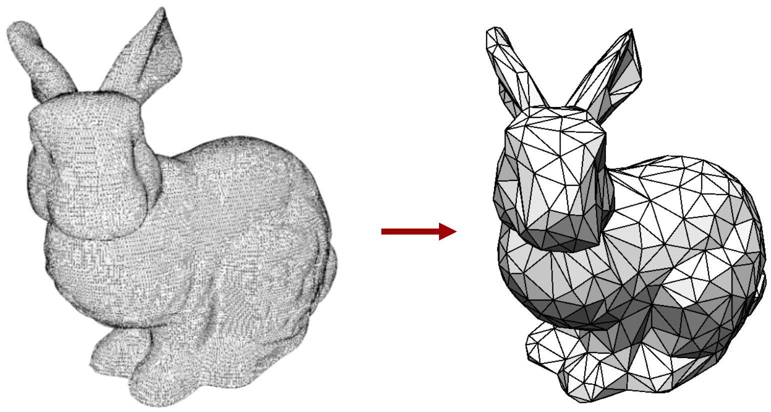


Illustration



Mesh Simplification

Mesh Simplification



Mesh Simplification:

- Triangle meshes are often oversampled
- In particular, meshes from 3D scanners
- We want to decimate the number of triangles such that the shape of the object is roughly maintained
- We want to do this automatically

Variants of the Problem

Problem Variations:

- Mesh simplification
  - Reduce the number of triangles
  - Fixed triangle budget or fixed approximation error
- Multi-resolution models
  - Create a representation that provides many levels of resolution
  - The matching level-of-detail can be extracted at runtime
  - Useful for real-time rendering
    - Choose level of detail for each object in the scene
    - More sophisticated: varying level of detail across one object (the whole scene can be one object)

Curve Simplification

Curve Simplification:

- Compute an approximation of a piecewise linear curve by another piecewise linear curve with fewer segments
- The optimal least-squares solution can be computed in $O(mn^2)$ time using dynamic programming
  - where n = #(*input* line segments)
  - and m = #(*output* line segments)
- Usually, this is still to costly.

Curve Simplification

Curve Simplification:

- Most frequently used heuristic:
  Douglas-Peucker Algorithm.
- Simple Idea:
  - Start with a line connecting the end points
  - Find the input point farthest away from the straight line
  - Insert a new vertex there. We obtain two new segments
  - Apply the algorithm recursively to the parts (a number of times)
- Usually gives (visually) good results

Mesh Simplification

Mesh Simplification:

- We need to find an approximating mesh to a given mesh

Optimal solution?

- It can be shown that finding an L_∞-norm best approximation to a mesh is NP-hard
- For other cases (e.g., least-squares) no efficient optimal techniques are known.

Mesh Simplification

Approximation algorithms:

- Polynomial time approximation algorithms with strict error guarantees are known, but they are too slow for practical applications

Michelangelo's St. Matthew
386,488,573 triangles
[Stanford Digital Michelangelo Project]

Parametric Simplification

If we have a parametric representation

- Spline surface
- Trimmed NURBS
- or the similar

we can just retessellate the original. No need for mesh-based simplification.

In the following: Input is a mesh (no side information)

Mesh Simplification

Three classes of techniques:

- Mesh refinement
  - Start with a simple base mesh, refine to approximate the object
  - "Gift-wrapping"
  - Complicated to implement (need to adjust topology)
- Mesh decimation
  - Start with full mesh
  - Keep on throwing away triangles until precision is met
  - This is the current standard technique
- Other approaches
  - Transform into implicit function and retessellate
  - Vertex clustering on a regular grid (useful for out-of-core impl.)

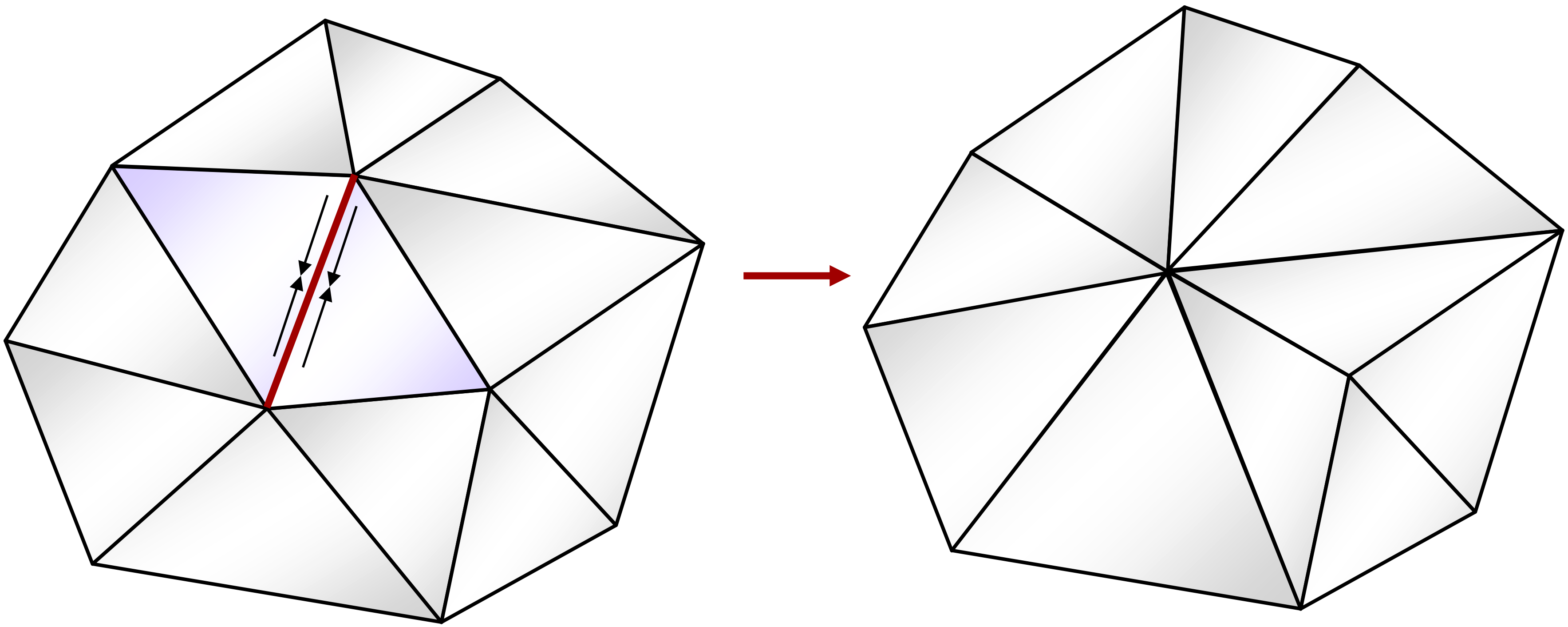
Mesh Decimation

Mesh decimation – basic idea:

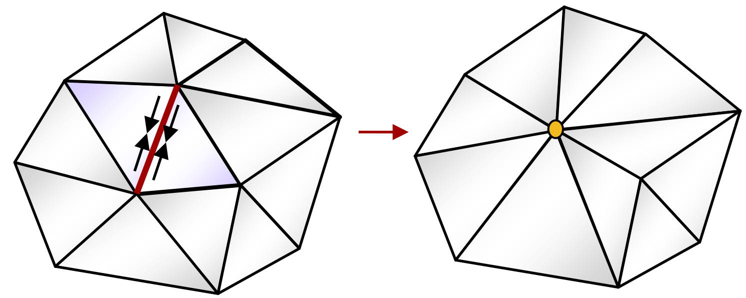
- Start with the full mesh
- Then, subsequently remove
  - Triangles (fill hole)
  - Vertices (retriangulate hole)
  - Edges (kills two triangles)
- Edge contraction (“edge collapse”) algorithms are nowadays the most common technique
- Robust and simple to implement

Edge Contraction

Edge contraction:



Edge Contraction



Edge contraction algorithm:

- Questions:
  - Which edges can be collapsed?
  - What error does this cause?
  - Edges collapse into points –
    where should we place the new point?
  - What is the best order for edge collapses?
- Standard algorithm:
  - Greedy algorithm
  - Put edges in priority queue
  - Pick the "cheapest" edge and remove it
  - Recompute costs

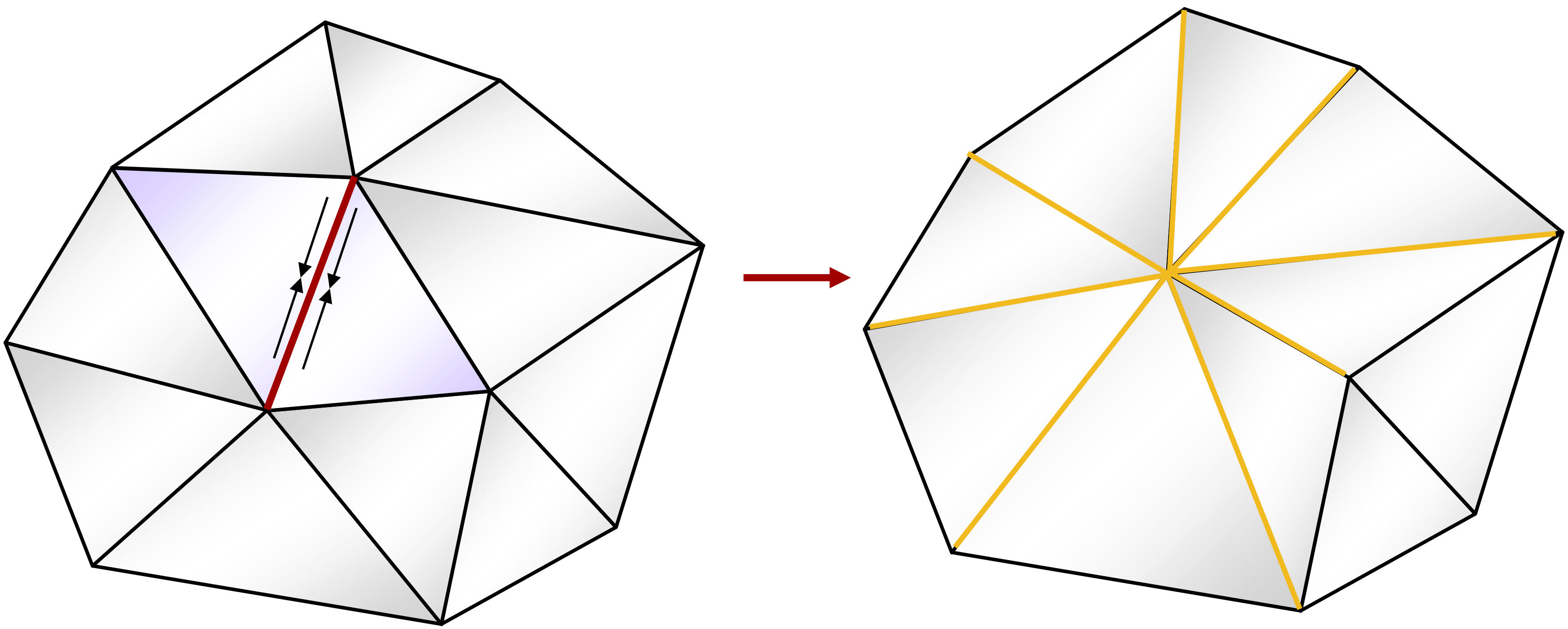
Edge Contraction

Algorithm:

- For each edge in the mesh, compute the costs of collapsing the edge
  - If an edge collapse changes the topology, set costs to $+\infty$
  - Put all (finite cost) edges in priority queue sorted by cost
- **While** queue not empty **and** result not simple enough
  - Remove min-cost edge
  - Collapse the edge
  - Recompute costs of all affected edges (incl. topology check)
  - Update the priority queue accordingly

Edge Contraction

Affected edges:



edge contraction

affected edges

Components

The algorithm needs the following components:

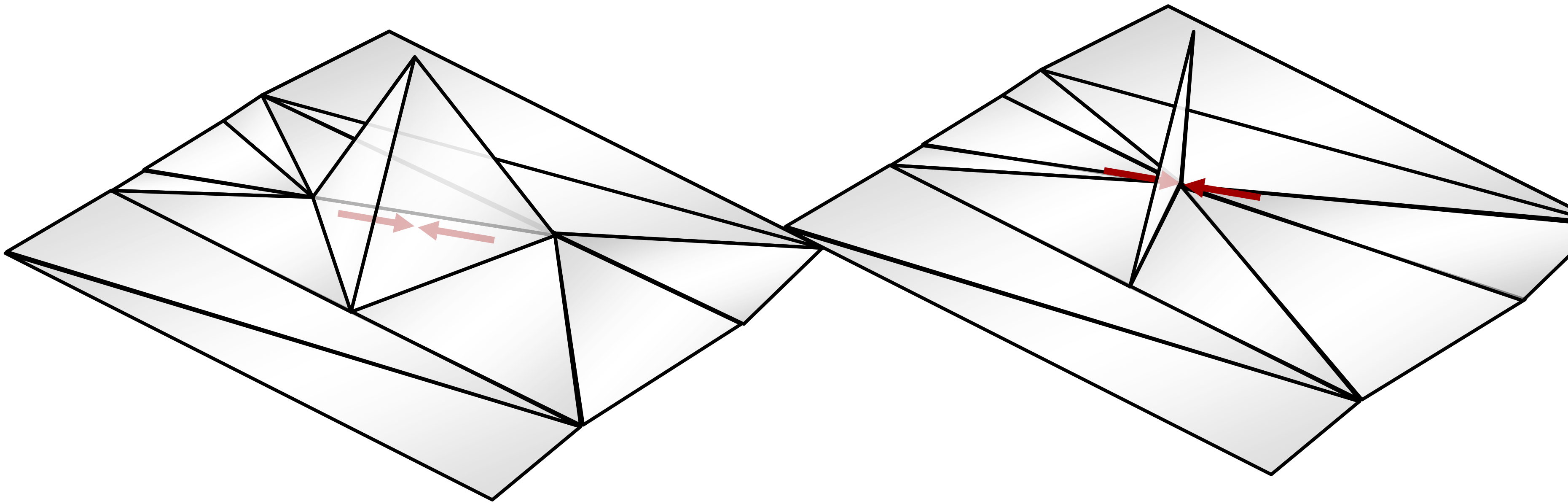
- Topology check (mostly fixed)
- Error metric (lots of choices)
- Placement of new vertices (lots of choices)

Topology Check

We do not want to change the topology of the mesh

- Input is a triangulated two-manifold, probably with boundary
- This means:
  - Every edge is adjacent to *one* or *two* triangles (*boundary* / *interior*)
  - Triangles do not intersect
  - The mesh is conforming – no vertices in the middle of edges (fortunately, edge collapsing cannot change this)

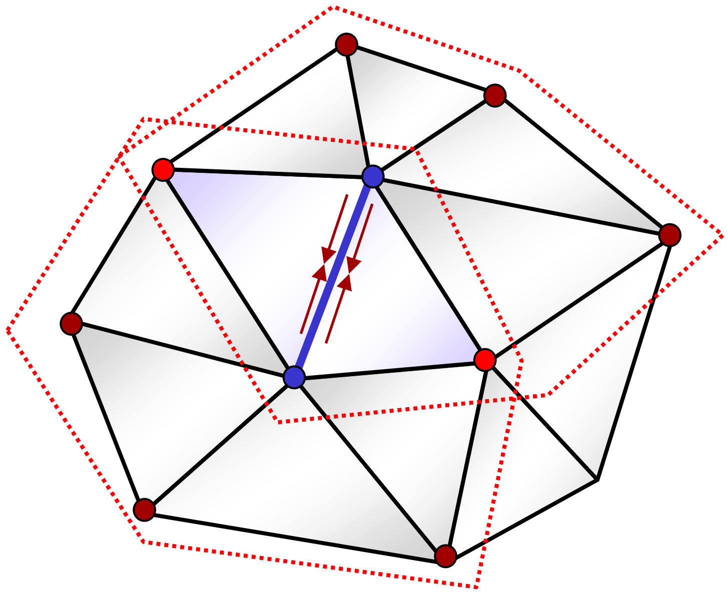
Problem #1: Folds



Problem #1:

- Edge collapses can cause topological folds in meshes
- We need a criterion to prevent this

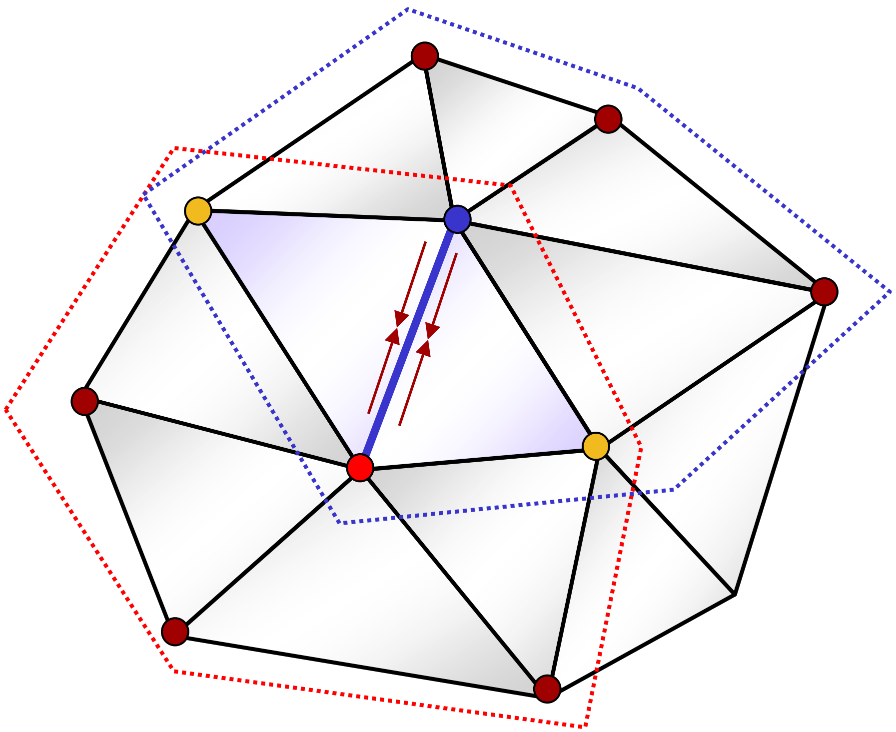
Criterion



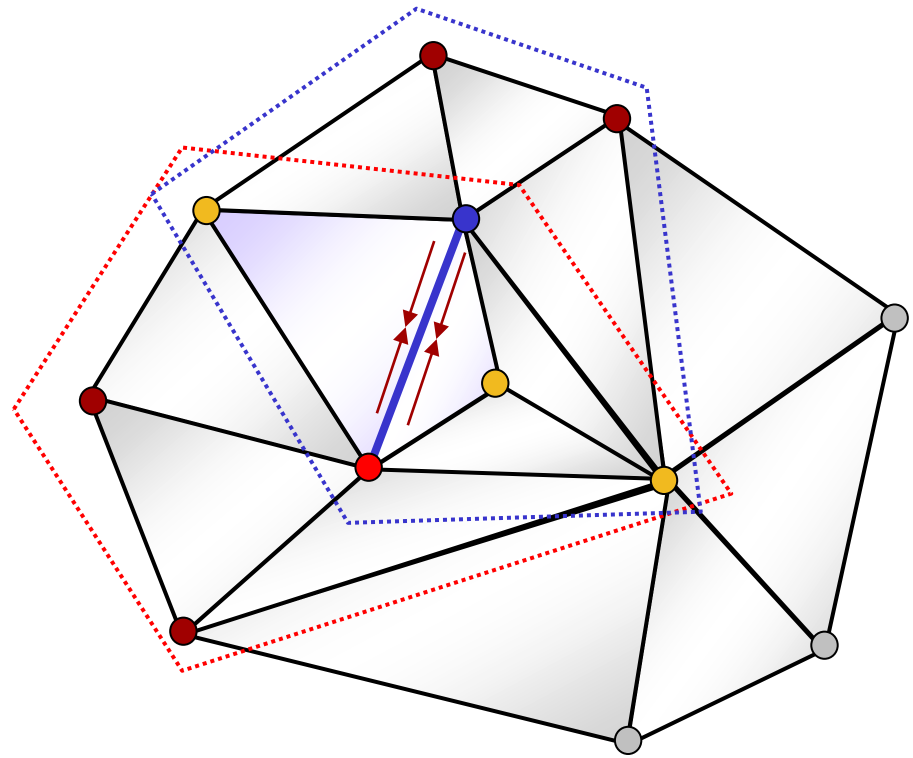
Criterion:

- Consider the two vertices of the edge $\mathbf{v}_1$, $\mathbf{v}_2$
- Let $R^{(1)}(\mathbf{v})$ be the on-ring neighborhood of $\mathbf{v}$, *excluding* $\mathbf{v}_1$, $\mathbf{v}_2$
- If $\#(R^{(1)}(\mathbf{v}_1) \cap R^{(1)}(\mathbf{v}_2)) = 2$, the collapse is permitted
- For boundary points: $\#(R^{(1)}(\mathbf{v}_1) \cap R^{(1)}(\mathbf{v}_2)) = 1$

Illustration



this works



this folds

Intersections

Preventing Intersections

- The previous criterion only guarantees topologically correct meshes
- The embedding into space (read: vertex placement in $\mathbb{R}^3$) can still cause self intersections
- We need to check this separately:
  - Do the newly created triangles intersect with the shape
    - (Hierarchical intersection test with dynamic hierarchy)
  - If so, avoid the collapse operation
- Often, people omit this check (hard to implement, does not happen frequently in practice)

Components

The algorithm needs the following components:

- Topology check (mostly fixed) ✓
- Error metric (lots of choices)
- Placement of new vertices (lots of choices)

Error Metrics

Various potential error metrics:

- S = original, S' = approximation, dist(·,·) = smallest distance
- L_2-error: $\int_S dist(S',x)^2 dx$
- L_1-error: $\int_S |dist(S',x)| dx$
- L_∞-error: $\max_{x \in S} |dist(S',x)|$
- Hausdorff error: $\max\left(\max_{x \in S} |dist(S',x)|, \max_{x \in S'} |dist(S,x)| \right)$

  (two sided maximum distance, symmetric measure)

Complexity Problem

Evaluating the error metric can be expensive:

- Compute the distance between two objects in $\Omega(n+m)$
- Naive computation takes $O(nm)$
- Doing this for each edge collapse is expensive

Solutions:

- Compute distance to previous level of detail only (works well in practice, but no guarantees)
- Use an approximate distance measure.

Quadric Error Metric

Quadric error metric: [Garland and Heckbert 1997]

- Very efficient solution to the error quantification problem
- However, the estimates might be too pessimistic

Idea:

- Measure distance to planes, rather than original triangles
- The error is represented as a 3D quadric

Quadric Error Metric

Implicit plane equation:

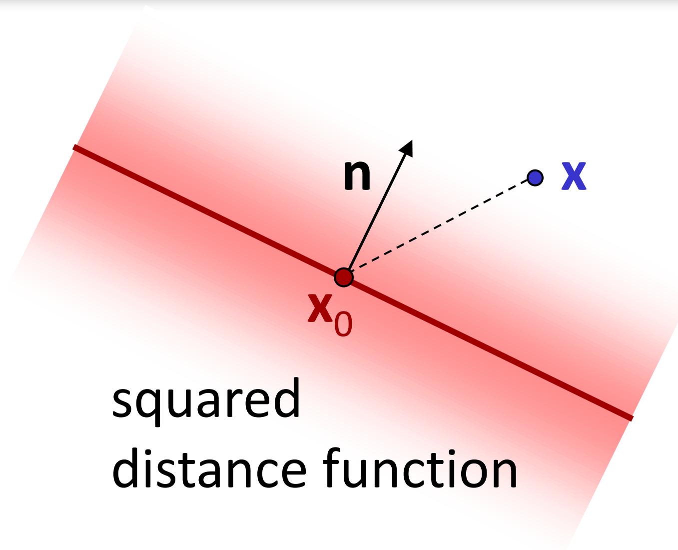
$$\langle \mathbf{n}, \mathbf{x} - \mathbf{x}_0 \rangle = 0$$

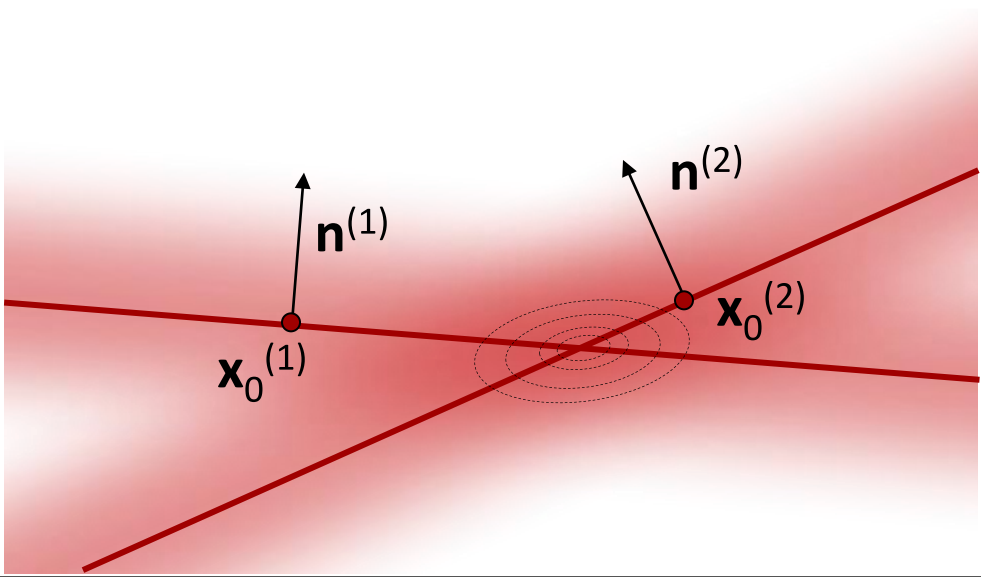
Quadratic error function:

$$\langle \mathbf{n}, \underbrace{\boxed{\mathbf{x}}}_{\text{variable}} - \mathbf{x}_0 \rangle^2$$

Minimum distance to several planes:

$$\sum_{i=1}^{n} \langle \mathbf{n}^{(i)}, \underbrace{\boxed{\mathbf{x}}}_{\text{variable}} - \mathbf{x}_0^{(i)} \rangle^2$$





Quadric Error Metrics

Use in mesh simplification:

- Assign an initial error quadric to each vertex
  - Formed by summing up the plane error functions of the planes of all adjacent triangles
  - Weight components by triangle area
  - Error will be zero for the vertex itself (intersection of all planes)
- For each possible edge contraction:
  - Just add the error quadrics of both vertices involved
  - This means, the new, contracted vertex should approximate the planes of all triangles involved so far as well as possible

Quadric Error Metrics

Use in mesh simplification:

- For each possible edge contraction:
  - Compute the optimum vertex position according to the summed error metric
  - Evaluate the quadric to determine the error
  - This is the candidate move (error, position) that is stored in the edge contraction queue
- When an edge contraction occurs:
  - Use the computed position
  - To recompute neighborhood error quadrics, add the error matrix of the new vertex to each neighboring vertex
  - This gives new edge contraction costs

Extension

Meshes also have attributes, such as:

- Color
- Texture coordinates

This can be handled using quadric error metrics as well:

- Just store additional columns in the x-vectors
- Treat color values (etc.) as additional dimensions of the vertex position, weighted by relative importance to preserve them

How well does this work?

Advantage:

- Very fast: Evaluating the error metric and finding a new vertex position is O(1)

Disadvantage:

- For noisy meshes, the error approximation is bad:

- Possible solutions:
  - Mesh smoothing (normals from larger neighborhoods)
  - Reset quadrics after a few computation steps

Components

The algorithm needs the following components:

- Topology check (mostly fixed) ✓
- Error metric (lots of choices) ✓
- Placement of new vertices (lots of choices) ✓

Conclusion:

- Quadric error metrics are a very popular choice due to their simplicity and performance.
- More accurate alternatives exist (at higher costs).

Multi-Resolution Meshes

Multi-resolution version:

- We want to store multiple levels of detail in one representation
- Simple, but effective approach: Progressive meshes [Hoppe 1996]

Progressive meshes:

- Simplify as strongly as possible (we get a *base mesh*)
- Record all edge contractions in a list

Progressive Meshes

Adjusting the level of detail:

- Start with the base mesh
- Perform *inverse edge contractions*, which are *vertex splits*, to increase the level of detail
- Perform edge contractions to reduce the level of detail
- The index in the list of edge contractions controls the level of detail:
  - Index up: Level of detail increases
  - Index down: Level of detail decreases

Example

[H. Hoppe, Microsoft Research, 1996]

Hardware Friendly Implementation

Progressive meshes are expensive:

- Graphics hardware can render billions of triangles
- Performing precomputed edge collapses / vertex splits still takes a lot of computational resources

Hardware Friendly approach:

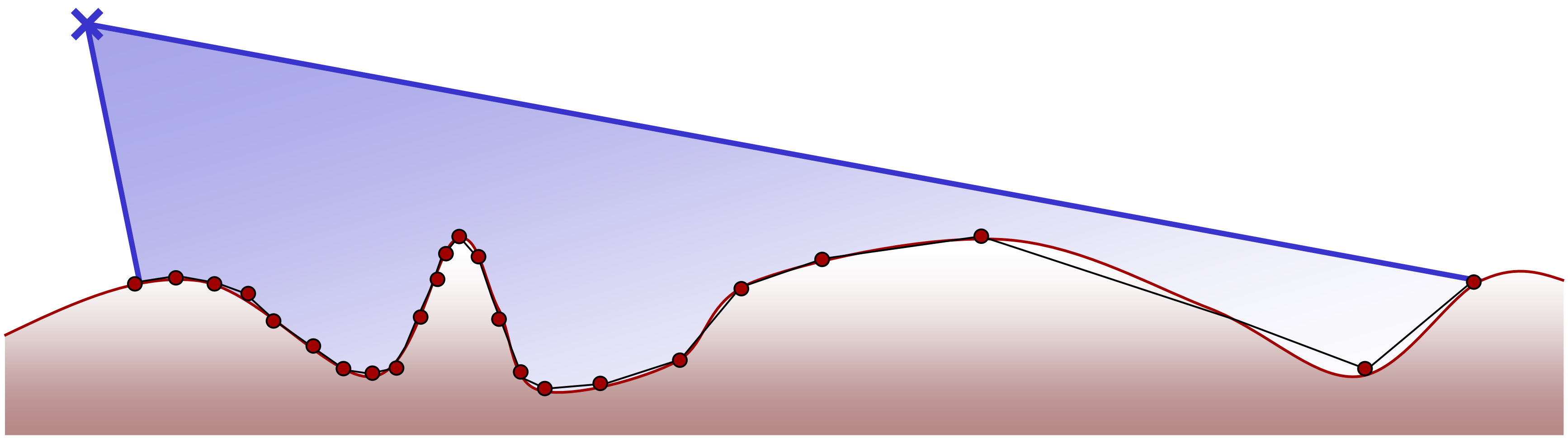
- Precompute a number of levels of detail
- Just render them as needed
- Use linear interpolation to smoothly blend in the new vertices (avoid popping)

Adaptive Rendering

Problem:

- Assume we are handling a very large object
- For example a terrain model of the globe (Google earth)
- Progressive levels of detail are not helpful
  - Either too coarse or too much geometry
- We need adaptive extraction of details
  - Level-of-detail varying across the object
  - How can this be done with a progressive mesh representation?

Adaptive LOD Extraction



Adaptive / non-uniform level of detail extraction:

- Assumption:
  - We are given a camera position
  - and a geometric error messure $g(x, lod)$.
  - We want to extract geometry such that $g(x, lod) / z(camera) < \varepsilon$.

Adaptive LOD Extraction

Adaptive / non-uniform level of detail extraction:

- Simple idea:
  - Start with base mesh
  - Test for each vertex if adjacent triangles are accurate enough
    - Conservative test (minimum depth)
  - If accuracy is not sufficient: perform vertex split
- Problem: *Vertex splits are not independent*
  - We can only perform splits if the vertex already exists
  - Vertices might have been created by previous vertex splits
  - Need to take into account the *dependence hierarchy*.

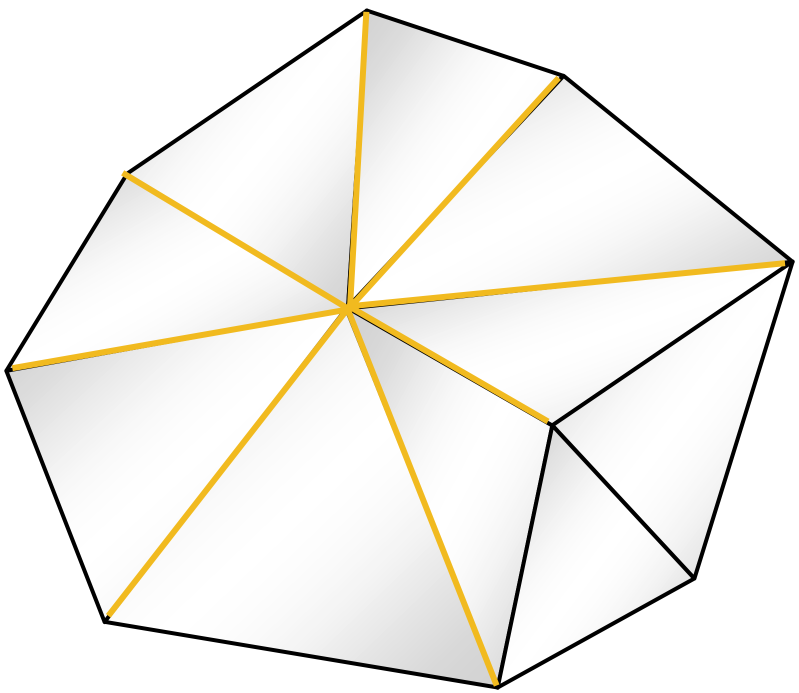
Multi-Triangulation

Formal Framework: Multi-Triangulation

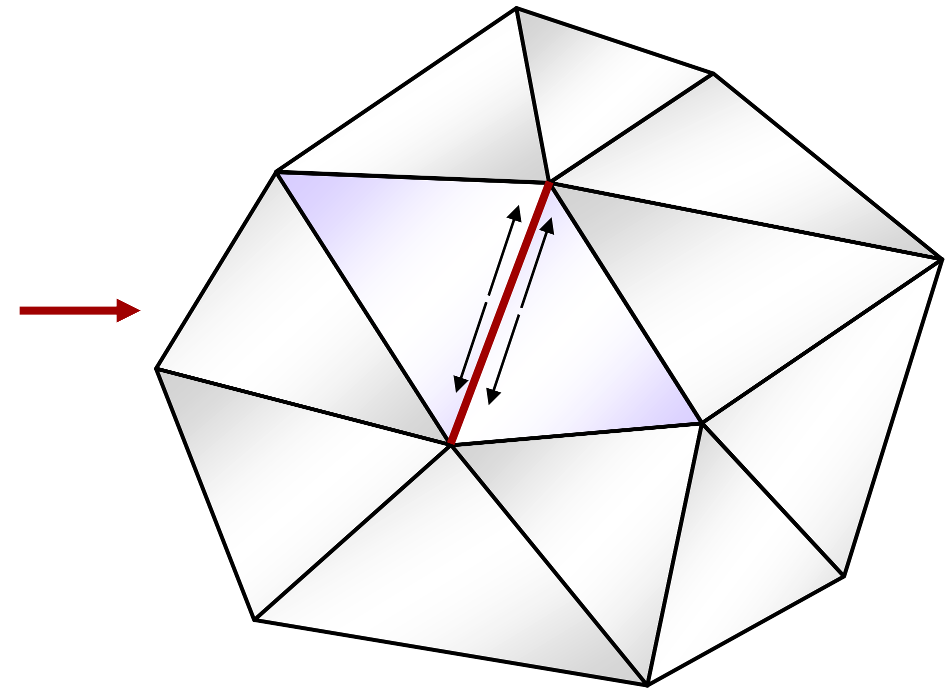
- During construction of the progressive mesh:
  - An edge contraction *depends* on a previous contraction if one of its vertices is the result of a previous edge contraction.
    - Correspondingly, a vertex split depends on previous splits if its vertex is the result of a previous split
  - One edge contraction might depend on up to two other contractions, which each might depend on up to two others
  - This yields a acyclic directed graph (DAG)

Vertex Split

Affected edges:

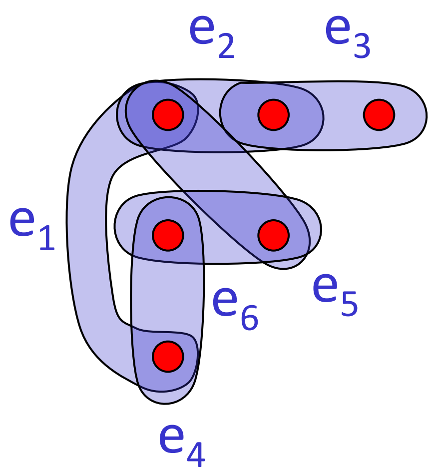


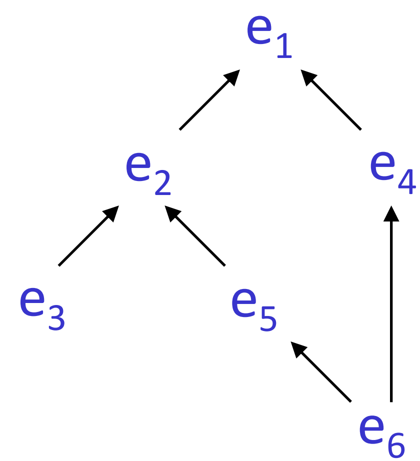
base mesh

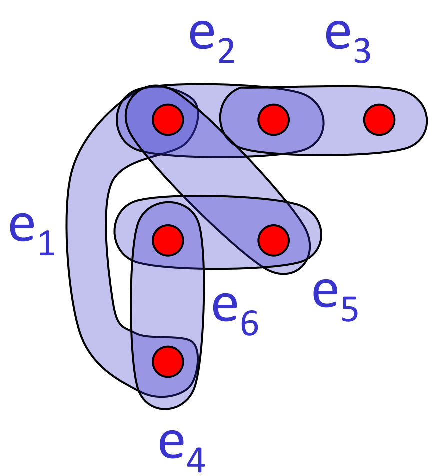


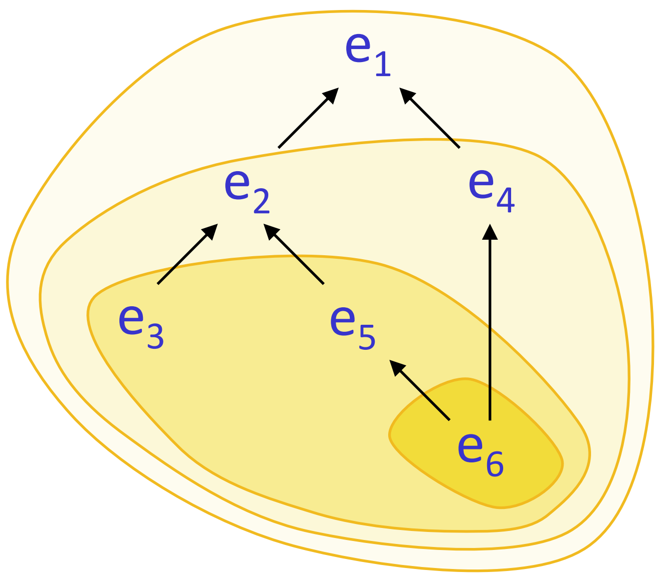
vertex split

Dependencies









Optimizing the Hierarchy

Need to take care of the dependencies:

- Need to store dependencies (DAG)
- When building the hierarchy:
  - Minimizing dependencies maximizes adaptivity, but might reduce quality
  - Possible strategy:
    - Only collapse non-dependent edges
    - When no edges are left, start new round of collapsing
    - Creates hierarchy with several levels

Hardware Friendly Version

Same problems again:

- The representation might be to costly to extract
- Executing a single vertex split / edge collapse from a precomputed hierarchy might still be more expensive than rendering (processing) many triangles
- Solution:
  - Clustered simplification with “large nodes”
  - Same idea as for the adaptive grids, but with edge collapses

Large Node Hierarchies

Idea for a hardware friendly algorithm (sketch):

- Divide the object into hierarchy of clusters
- For example:
  - Octree decomposition
  - Binary splitting along principal axis
  - Or the similar
- Hierarchy:
  - Leaf nodes store original triangles, at least $k \geq$ *a few thousand* triangles per node
  - Inner nodes:
    - Union of child node triangles
    - Simplification to reduce complexity to 1/4 of input (octree)

Large Node Hierarchies

Problem: Boundaries

- Triangulations might be non-conforming at boundaries
- Possible solution:
  - For each edge: Compute two triangulations
    - Neighbor with the same resolution
    - Neighbor with resolution one level lower
  - During rendering:
    - Extract balanced cut of the hierarchy
    - Choose appropriate adaptor triangulation
- Alternative solution: [Klein & Guthe]
  - Bounded Hausdorff error approximation
  - Triangles overlap at the boundaries (“fat boarders”)

Appearance Simplification

(for Large Scene Rendering)

Problems with Mesh Simplification

Problems:

- Mesh simplification cannot perform arbitrarily strong simplifications without destroying object appearance completely
- We need an alternative approach for rendering really large scenes
- As an example: Hierarchical point-based simplification (extra slides set)

Announcement

Written Exam:

- If someone cannot participate in the first of the two exams:
  - In the case of not pass the second (and only) try, we would offer an optional, additional oral exam.
  - If the student passes the oral exam, she/he would pass the lecture.
- This applies only if...
  - ...you need to have an important reason for not being able to take the first exam (for example, collision with another exam on the same day)
  - ...you need to notify us (by email) at least one week before the first exam.