Fracture & Contact

Steve Rotenberg
CSE291: Physics Simulation
UCSD
Spring 2019

Eigenvalues & Eigenvectors

Eigenvalue Equations

- Before we get into fracture modeling, let us review the important linear algebra topic of eigenvalues and eigenvectors
- The standard matrix eigenvalue equation is written as:

$$\mathbf{Ax} = \lambda\mathbf{x}$$

- Usually, we have a known matrix $\mathbf{A}$ and we want to find the unit length (eigen) vectors $\mathbf{x}$, such that the transformed version $\mathbf{Ax}$ is in the same direction as the original vector, scaled by nonzero (eigenvalue) λ
- An *n*x*n* matrix $\mathbf{A}$ will have *n* orthogonal eigenvectors, and *n* associated eigenvalues, but they may contain complex numbers
- If the matrix is symmetric, then all of the eigenvectors and eigenvalues will be made of real numbers

Eigenvalues

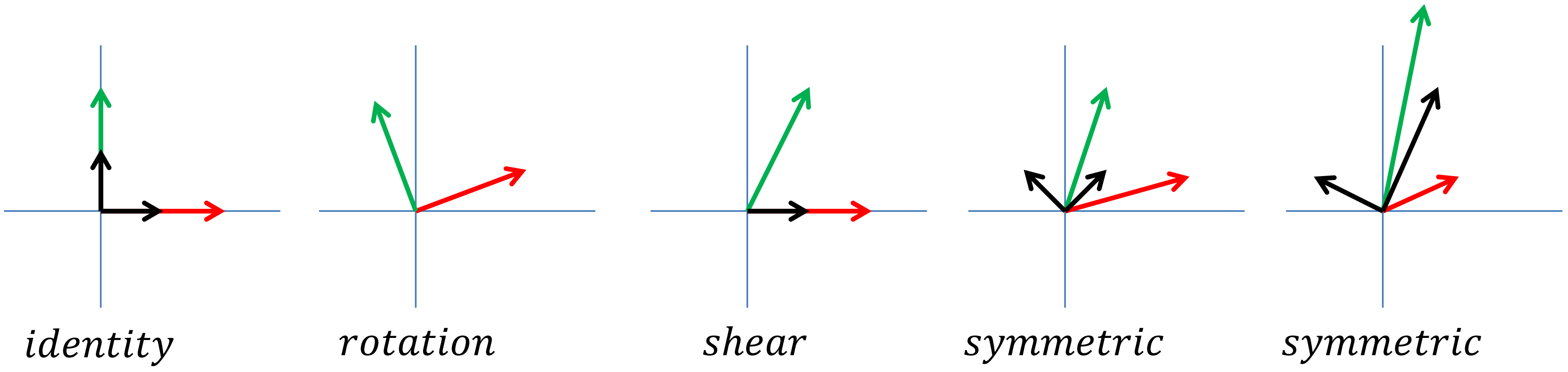
- Consider an orthogonal 3D rotation matrix
- There will be a vector along the axis of rotation that does not change when the rotation is applied
- In this case, there is only one real eigenvector and it’s eigenvalue is 1, since the length does not change
- Consider an *xy*-plane shear matrix

$$M = \begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- It will have two real eigenvectors (**x** & **z**), both with eigenvalues of 1

2D Examples

- Consider a 2D example. The red arrow shows to the first column vector and the green shows the second
- The black vectors are the eigenvectors (scaled down for visibility)



Orthogonal Diagonalization

- We can see that a symmetric matrix can be thought of as a non-uniform scale applied at some orthogonal orientation
- We can represent the eigenvalues as the diagonal of a non-uniform scale matrix:

$$\mathbf{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix}$$

- The orthogonal orientation is the basis formed by the set of eigenvectors. We can group these into an orthogonal basis matrix $\mathbf{Q}$
- The symmetric matrix $\mathbf{A}$ can be decomposed into a rotation *from* basis $\mathbf{Q}$ *into* the identity space (i.e., $\mathbf{Q}^{-1}$) followed by the non-uniform scale, followed by a forward rotation by $\mathbf{Q}$ to align it back in place
- We can write this as:

$$\mathbf{A} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^{-1}$$

Orthogonal Diagonalization

- We see that we can represent a symmetric matrix as an orthonormal matrix $\mathbf{Q}$ times a diagonal matrix $\mathbf{\Lambda}$ times $\mathbf{Q}^{-1}$. We can actually use $\mathbf{Q}^T$ because $\mathbf{Q}^T = \mathbf{Q}^{-1}$for orthonormal matrices

$$\mathbf{A} = \mathbf{Q\Lambda Q}^T$$

★ The columns of $\mathbf{Q}$ are the eigenvectors and the diagonal elements of $\mathbf{\Lambda}$ are the eigenvalues

- This is called an *orthogonal diagonalization*
- It is common in numerical computing (and physics simulation) to have a symmetric matrix and require the eigenvalues and eigenvectors
- Therefore, it would be nice to have an algorithm that could take $\mathbf{A}$ as input and compute $\mathbf{Q}$ and $\mathbf{\Lambda}$ as output

Cyclic Jacobi Algorithm

- Luckily, we have the *cyclic Jacobi algorithm* that does exactly this
- As an added advantage, this is actually a pretty simple algorithm to implement and can be very stable and reliable with minimal effort
- We iteratively compute

$$\mathbf{A} = \mathbf{Q}_i \mathbf{\Lambda}_i {\mathbf{Q}_i}^T$$

- Where $\mathbf{Q}_0 = \mathbf{I}$ and $\mathbf{\Lambda}_0 = \mathbf{A}$
- Each iteration, we will compute improved versions of $\mathbf{Q}_i$ and $\mathbf{\Lambda}_i$ until $\mathbf{\Lambda}_i$ is sufficiently diagonalized (i.e., within floating point tolerances)

Off-Diagonal Elements

- The matrix will have diagonal elements and off-diagonal elements
- Our goal is to gradually remove the off-diagonal elements, reducing them to 0
- Each step of the process, we choose an off-diagonal element and build an orthonormal matrix $\mathbf{J}$ that causes that particular value to go to exactly 0 in the next step of the iteration

Cyclic Jacobi Algorithm

- We can insert an orthonormal matrix $\mathbf{J}$ (Note $\mathbf{J}\mathbf{J}^T = \mathbf{I}$)

$$\mathbf{A} = \mathbf{Q}_i \boldsymbol{\Lambda}_i {\mathbf{Q}_i}^T$$

$$\mathbf{A} = \mathbf{Q}_i \mathbf{J}\mathbf{J}^T \boldsymbol{\Lambda}_i \mathbf{J}\mathbf{J}^T {\mathbf{Q}_i}^T$$

$$\mathbf{A} = \mathbf{Q}_i \mathbf{J} \cdot \mathbf{J}^T \boldsymbol{\Lambda}_i \mathbf{J} \cdot \mathbf{J}^T {\mathbf{Q}_i}^T$$

- And use this to generate a process for iteration

$$\mathbf{A} = \mathbf{Q}_{i+1} \boldsymbol{\Lambda}_{i+1} {\mathbf{Q}_{i+1}}^T$$

$$\mathbf{Q}_{i+1} = \mathbf{Q}_i \mathbf{J}$$
$$\boldsymbol{\Lambda}_{i+1} = \mathbf{J}^T \boldsymbol{\Lambda}_i \mathbf{J}$$

Iteration

$$\mathbf{\Lambda}_{i+1} = \mathbf{J}^T \mathbf{\Lambda}_i \mathbf{J}$$

- We wish to find an orthonormal matrix $\mathbf{J}$ that improves our solution by making $\mathbf{\Lambda}_{i+1}$ more diagonal than $\mathbf{\Lambda}_i$
- We can do this by finding a rotation that causes one symmetric pair of the off-diagonal elements $\mathbf{\Lambda}_{i+1}$ of to be 0 after the iteration
- For a 3D matrix, for example, we can cancel out the *xy* (and *yx*) elements by finding the correct rotation in the *xy* plane
- In higher dimensions, we can cancel out the jk (and kj) elements by finding the correct rotation in the jk plane

Givens Rotation

- In numerical computing, a *Givens rotation* is an *n*-dimensional rotation about a plane formed by two principal axes
- For example, we can write $\mathbf{G}_{jk}(\theta)$ as the rotation in the plane of the jth and kth axis by angle θ

$$\mathbf{G}_{jk}(\theta) = \begin{bmatrix}
1 & & & & & & 0 \\
& \ddots & & & & & \\
& & \cos(\theta) & \cdots & -\sin(\theta) & & \\
& & \vdots & \ddots & \vdots & & \\
& & \sin(\theta) & \cdots & \cos(\theta) & & \\
& & & & & \ddots & \\
0 & & & & & & 1
\end{bmatrix}$$

Rotation Solution

$$\mathbf{\Lambda}_{i+1} = \mathbf{J}^T \mathbf{\Lambda}_i \mathbf{J}$$

- We would like to find the jk Givens rotation that causes element jk to be 0 after the iteration

$$\mathbf{\Lambda}_{i+1} = \mathbf{G}_{jk}(\theta)^T \, \mathbf{\Lambda}_i \mathbf{G}_{jk}(\theta)$$

- We can multiply this out for the jk element of $\mathbf{\Lambda}_{i+1}$, using $c = \cos(\theta)$ and $s = \sin(\theta)$

$$\Lambda'_{jk} = (c^2 - s^2)\Lambda_{jk} + sc\left(\Lambda_{jj} - \Lambda_{kk}\right)$$

- We wish to find the angle θ that causes Λ'_{jk} to be 0 which can be solved as:

$$\theta = \frac{\Lambda_{kk} - \Lambda_{jj}}{2\Lambda_{jk}}$$

Iteration

- We cycle through the off-diagonal elements
- For each element jk, we compute:

$$\theta = \frac{\Lambda_{kk} - \Lambda_{jj}}{2\Lambda_{jk}}$$

- To get $\mathbf{G}_{jk}(\theta)$ and then update

$$\mathbf{Q}_{i+1} = \mathbf{Q}_i \mathbf{G}_{jk}(\theta)$$
$$\mathbf{\Lambda}_{i+1} = \mathbf{G}_{jk}(\theta)^T \, \mathbf{\Lambda}_i \mathbf{G}_{jk}(\theta)$$

- Note: these should use matrix multiplication that is optimized to take advantage of the fact that $\mathbf{G}_{jk}(\theta)$ is mostly 1's and 0's. Very few elements need to be updated and this can be performed in linear time with respect to *n* for an *n*x*n* matrix

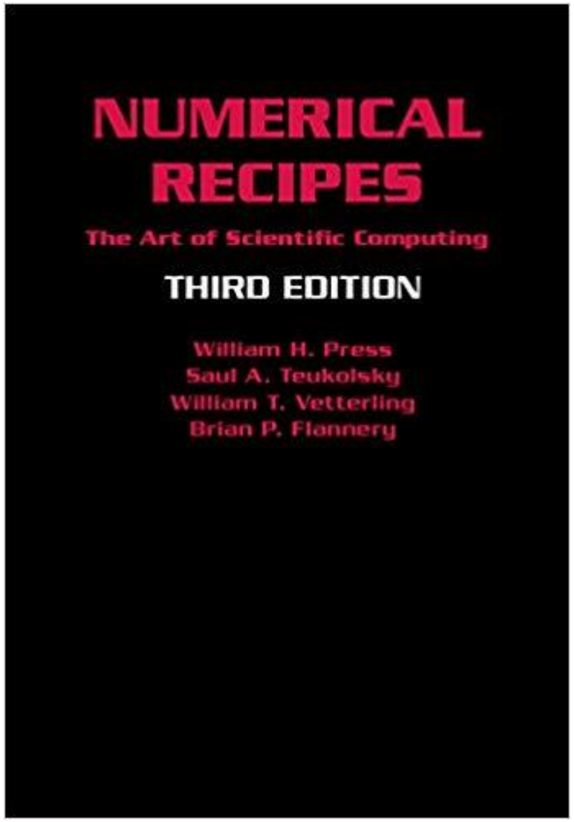
- We stop iterating when all of the off-diagonal elements are close enough to 0

Cyclic Jacobi Algorithm

- At each iteration, we choose a rotation that zeros out one of the off-diagonal element pairs of $\mathbf{\Lambda}$
- The overall algorithm can just cycle through the off-diagonal values and zero them out one by one
- The catch is that each new rotation can take previous zeros and make them non-zero again, but the effect grows smaller as the process proceeds
- The algorithm will consistently improve the result with each iteration, and will proceed until all of the off-diagonal elements are within some small tolerance near zero
- It is a good method for finding the eigenvalues and eigenvectors of a symmetric matrix and is included in many common numerical computation toolkits

Numerical Recipes

- “Numerical Recipes Third Edition”, Press, Teukolsky, Vetterling, Flannery, 2007
- This is a fantastic book that covers a very wide range of practical numerical algorithms. Every serious computer scientist should be familiar with this book



Fracture Mechanics

Fracture Mechanics

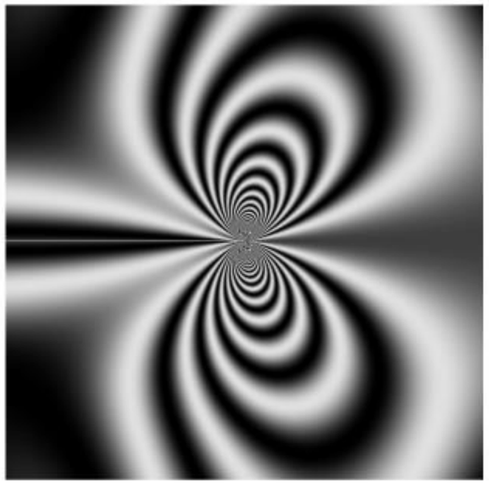
- Fracture is a complex subject that is ultimately dependent material properties at a variety of scales from macroscopic down to the molecular level

Fracture Theories

- An early fracture theory is the *maximum normal strain theory* (or St. Venant theory) which says that fracture occurs when the largest normal strain exceeds the *uniaxial fracture strain*
- A slight variation is the *maximum principal stress theory* (or Rankine theory) which says fracture occurs when the magnitude of the largest principal stress exceeds the *uniaxial fracture stress*
- Most materials can tolerate much larger compression forces compared to tension. *Coulomb-Mohr theory* is designed to address these different extremes and handle combinations where one axis is under compression while another is under tension
- They can all be looked at from the same point of view as they all define a boundary in strain/stress tensor space that if crossed, leads to a material failure (fracture). The exact shape of the boundary is defined differently with each model

Crack Tip

- If one applies the linear elasticity theory to an idealized representation of a crack tip under tension, the math leads to a stress that approaches infinity in the immediate vicinity of the tip
- This would lead to a crack always propagating once it starts, if the material is under tension. Highly elastic materials like glass can exhibit *brittle fracture* like this
- In real materials, the crack propagation can be stopped by plastic deformation taking place near the tip (even if it is very slight). If there is a noticeable amount of plastic deformation, we refer to it as *ductile fracture*



Fracture Modeling

Fracture Analysis

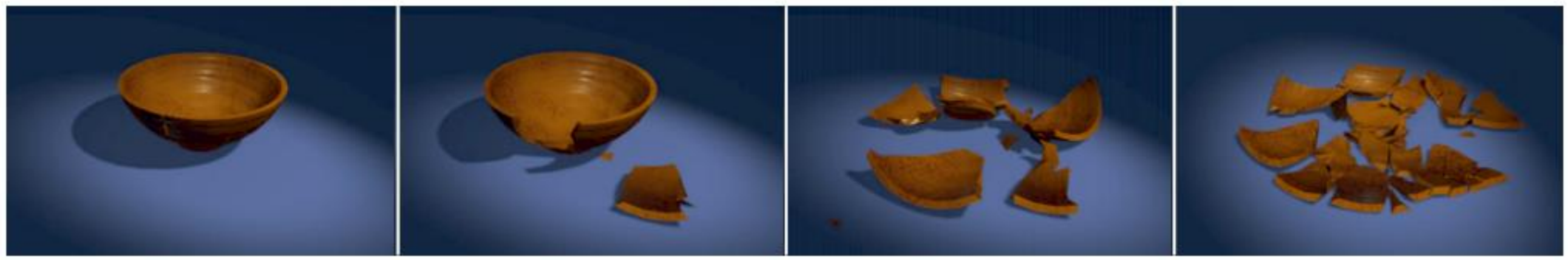
- For engineering applications, methods have been developed for analyzing the properties of fracture and crack propagation
- Most of these methods involve close-up study of crack tips and require some type of way to handle the fact that stresses approach extreme levels in very small areas
- For these reasons, most engineering grade fracture analysis is done using specialized finite element types that are designed to handle these cases
- One catch is that they tend to require the placement of these special elements in the place where you are expecting the extreme stresses
- This makes them useful for applications like analyzing CAD models for potential fracture sites as well as various theoretical applications
- For more general applications, we would like to allow any element to fracture if necessary, rather than requiring specific element types

Fracture Modeling

- To include fracture into our previously discussed model of tetrahedral elastic/plastic elements:
  - We will have to choose a material failure model to determine when a fracture starts
  - We will also need some way to evaluate this model at some location of choice (i.e., at node points, element centers…)
  - We will need a way to modify the mesh to account for the fracture. Generally, this will involve cutting the mesh at some local fracture plane

Fracture Paper

- We will look at the paper “Graphical Modeling and Animation of Brittle Fracture”, O’Brien, Hodgins, 1999
- This presents a general purpose fracture algorithm for complex geometry
- They start with a tetrahedral mesh and a linear elastic solid model
- They introduce a process for determining the initiation of fracture at a node and propagating the fracture throughout the solid
- They also discuss collision detection and response



Fracture Paper

- The paper uses a tensile stress measure to determine when fracture occurs
- They evaluate this at the nodes of the mesh
- If fracture occurs at a node, the fracture plane is found and used to split neighboring tetrahedra
- The fracture does not propagate any further in one simulation step, but will very possibly continue in the next frame

Algorithm Overview

- "Our fracture algorithm is as follows:
  - After every time step, the system resolves the internal forces acting on all nodes into their tensile and compressive components, discarding any unbalanced portions
  - At each node, the resulting forces are then used to form a tensor that describes how the internal forces are acting to separate that node
  - If the action is sufficiently large, the node is split into two distinct nodes and a fracture plane is computed
  - All elements attached to the node are divided along the plane with the resulting tetrahedra assigned to one or the other incarnations of the split node, thus creating a discontinuity in the material
  - Any cached values, such as the node mass or the element shape functions, are recomputed for the affected elements and nodes
  - With this technique, the location of a fracture or crack tip need not be explicitly recorded unless this information is useful for some other purpose, such as rendering."

$source: O'Brien, et\ al., 1999$

Force Decomposition

- The paper uses a linear elastic model with Green's strain $\boldsymbol{\epsilon}$ tensor and the Cauchy stress tensor $\boldsymbol{\sigma}$
- They also include a linear damping model and so the stress tensor is the sum of elastic and viscous components
- After computing the stress tensor $\boldsymbol{\sigma}$ for an element, it is decomposed into tensile $\boldsymbol{\sigma}^+$ and compressive $\boldsymbol{\sigma}^-$ components such that $\boldsymbol{\sigma} = \boldsymbol{\sigma}^+ + \boldsymbol{\sigma}^-$
- This is done by taking the eigenvalues & eigenvectors of the stress tensor
- Positive eigenvalues correspond to tensile stresses and negative ones to compressive stresses
- They separate these out and add them to the appropriate tensor ($\boldsymbol{\sigma}^+$ or $\boldsymbol{\sigma}^-$)
- We can use these to compute the forces at each node that result from tensile and compressive stresses

Separation Tensor

- Each node is connected to several tetrahedra, and each tetrahedron computes the node forces resulting from tensile and compressive stresses
- These are summed up into a symmetric 3x3 stress variant called the *separation tensor* $\boldsymbol{\varsigma}$
- The positive eigenvalues of this tensor represent the overall tensile stresses acting on the node and the largest of these can be compared to the material toughness τ to determine if fracture occurs at the node

Fracture Plane

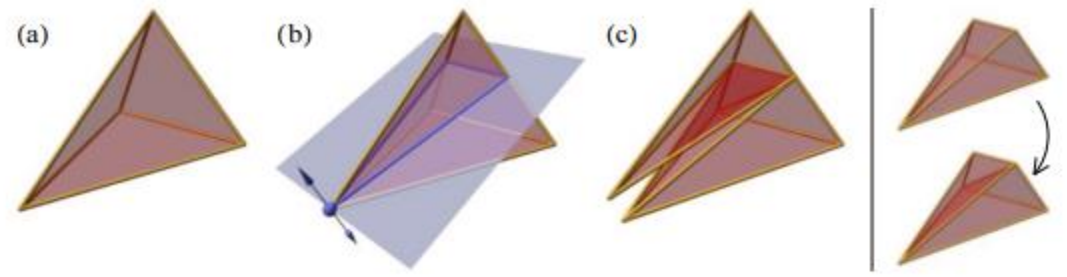
- The eigenvector corresponding to the largest eigenvalue of **ς** will indicate the direction of maximum tensile stress, and thus will be normal to the fracture plane

Node Splitting

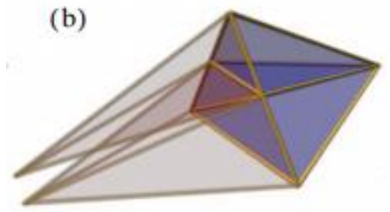
- If we determine that a fracture takes place at a node, we duplicate the node and re-assign the local tetrahedra to one of the two nodes based on whether they are above or below the fracture plane
- A simple implementation could just loop through the elements connected to the node and assign them based on where the element center is relative to the fracture plane
- A more elaborate implementation (like the one in the paper) would actually split elements that intersect the plane
- After any splitting and re-assigning is finished, element and node properties (volumes, masses, etc.) are recomputed as needed

Element Splitting

- Given the where the fracture occurs and the normal to the fracture plane, we can determine how to split up a tetrahedron that intersects the fracture plane



- We will also have to split a few neighboring tetrahedra



images: O'Brien, et al., 1999

Related Papers

- “Graphical Modeling and Animation of Ductile Fracture”, O’Brien, Bargteil, Hodgins, 2002
  - Adds plasticity to model elastic->plastic->fracture progression

- “Adaptive Tearing and Cracking of Thin Sheets”, Pfaff, Narain, de Joya, O’Brien, 2014
  - Uses similar approaches on 2D shell elements combined with adaptive triangle meshing and bending stresses/plasticity to model high resolution cracking and plastic deformation behavior for thin materials

Contact Modeling

Contact Modeling

- Collision and contact modeling is a complex subject, important to both deformable and rigid body dynamics
- It is generally split into two separate problems, namely the geometric problem of *contact detection* and the physics problem of *contact response*

- Incidentally, we will use the words *collision* and *contact* interchangeably today. In rigid body dynamics, there is more of a need to distinguish between instantaneous collisions and finite time contacts, but for deformable solids, we will assume finite time contact

Contact Detection

- Contact detection is the geometric problem of determining which elements are touching or intersecting others and determining specific properties of the contact area
- At the level of tetrahedra, we would want to be able to determine if tetrahedron *A* is intersecting tetrahedron *B*
- This is a limited form of the more general case of finding the intersection of two convex polyhedra, which is a well studied problem in computational geometry
- Testing for potential collisions in a set of *n* tetrahedra is effectively an $O(n^2)$ problem, but we can reduce this with some spatial data structures
- We will come back to some contact detection issues later

Contact Response

- Contact response refers to the computation of forces resulting from contact
- This includes forces normal to the contact plane responsible for keeping the two objects from going through each other as well as friction forces in the contact plane that oppose sliding motion

Contact Discontinuities

- High quality contact modeling produces discontinuities that can be problematic for some integrators:
  - Velocities normal to the contact plane are discontinuous when initial impacts occur
  - Tangential velocities can be discontinuous due to stick-slip transitions in friction model
- This makes it difficult to use integrators like Runge-Kutta or any Adams methods, as they rely on some assumption of smoothness to achieve their benefits

Contact Modeling

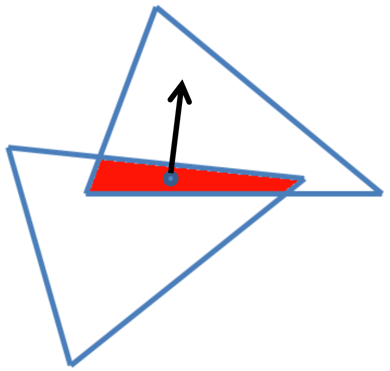
- There are several computational approaches to computing contact forces
- The blue Belytschko, 2014 book covers four different approaches of varying complexity and capabilities
- We will examine a simple method known as the *penalty method* which just applies large displacement dependent forces to counteract any interpenetrations
- More sophisticated methods set up large systems of equations and solve exact contact forces in order to maintain certain kinematic constraints

Penalty Methods

- We can apply penalties in different ways
- For example, we could just consider the vertices of a tetrahedron intersecting the volume of another (which is simple to compute) and generate a function similar to a spring repulsion on the vertex
- A more sophisticated approach would be to use the volume of the intersection itself to determine the force

Volumetric Penalties

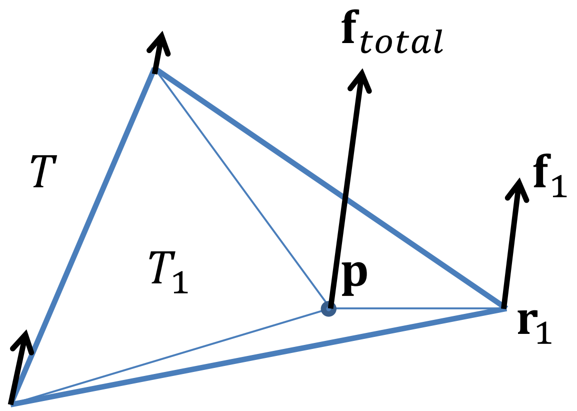
- Assuming we have two interpenetrating tetrahedra, we want to compute a single force (& it's equal and opposite) and the single location where it will apply in order to push the two apart
- To do this, we require a convex clipping routine that clips one tetrahedron to the 4 planes of the other, producing a convex volume for the intersection
- We then need to compute the volume and center of mass of this intersection polyhedron to determine the magnitude of the force (which can be a linear function of the volume) and the location (center of mass)
- To determine the direction of the force, one can use an area-weighted average of the normals of the faces of the polyhedron that came from one of the tetrahedra (or some variant of this)



Barycentric Weights

- We want to apply a force at a point inside a tetrahedron and have this transfer to a force on each node
- We can use a barycentric weighting for this
- The force on each node is equal to the total force times the ratio of the volume of the tetrahedron connecting the point to the opposite face over the total volume
- For example, if we apply a force at position $\mathbf{p}$, we turn this into a force $\mathbf{f}_1$ at point $\mathbf{r}_1$ (and similar for $\mathbf{f}_2$ and $\mathbf{f}_3$)

$$\mathbf{f}_1 = \frac{T_1}{T}\mathbf{f}_{total}$$



Friction Forces

- We can use the estimated contact point and contact plane to compute friction forces
- The simple but effective Coulomb friction model says:

$$f_{stat} \leq \mu_s f_{norm}$$
$$f_{dyn} = \mu_d f_{norm}$$

- Where f_{stat} is the force due to static friction, f_{dyn} is the force due to dynamic friction, f_{norm} is the normal force, and μ_s and μ_d are the coefficients of static and dynamic friction
- The direction of the force will be opposite of the tangential velocity at the contact point
- Dynamic friction isn't hard to add to the existing penalty force system, but to properly model static forces requires more sophisticated constraint handling and does not fit well in to a penalty based scheme
- The blue book covers some fancier options

Contact Summary

- With the penalty method, we compute instantaneous forces based on the current configuration of the system (just like with elastic forces, etc.)
- We can therefore include them in the same loop that we compute all other system forces
- To determine the contact forces on an element, we first need to identify the elements it is overlapping with
- For each overlapping pair of tetrahedra, we compute the convex polyhedron representing the intersection of the two using a polygon-plane clipping algorithm
- We then compute the volume, center of mass, and weighted average normals to determine the force vector and application point
- We then apply equal and opposite penalty forces to the two elements at the contact point using a barycentric weighting

Contact Optimization

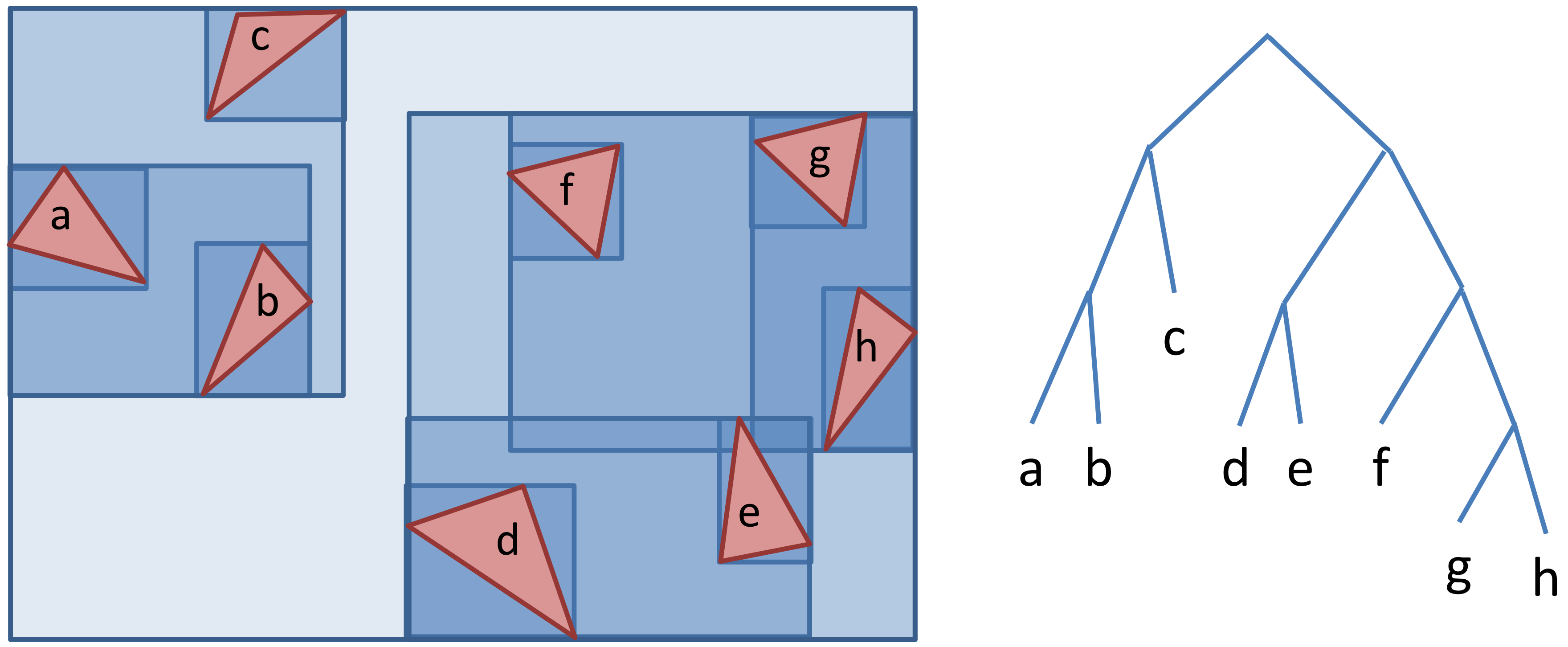
Contact Detection Performance

- If we have *n* tetrahedra in our system and any one could potentially contact any other, we have an O(n^2) problem
- We should be able to do better with some intelligent data structures
- With an axis-aligned bounding box (AABB) tree, we should be able to achieve O(*n* log *n*)
- In some cases, we can use spatial hash tables to achieve linear O(*n*) performance
- We will discuss spatial hash tables when we talk about particle fluids, so today we'll look at AABB trees

Axis Aligned Bounding Box

- A single AABB is represented with two 3D vectors - one for the min and one for the max corner
- Testing box-box overlap is trivial
- The boxes can be grouped into a tree structure. It is easier to make this a binary tree, but not strictly necessary
- At the leaf nodes, boxes contain either one or a small number of tetrahedra

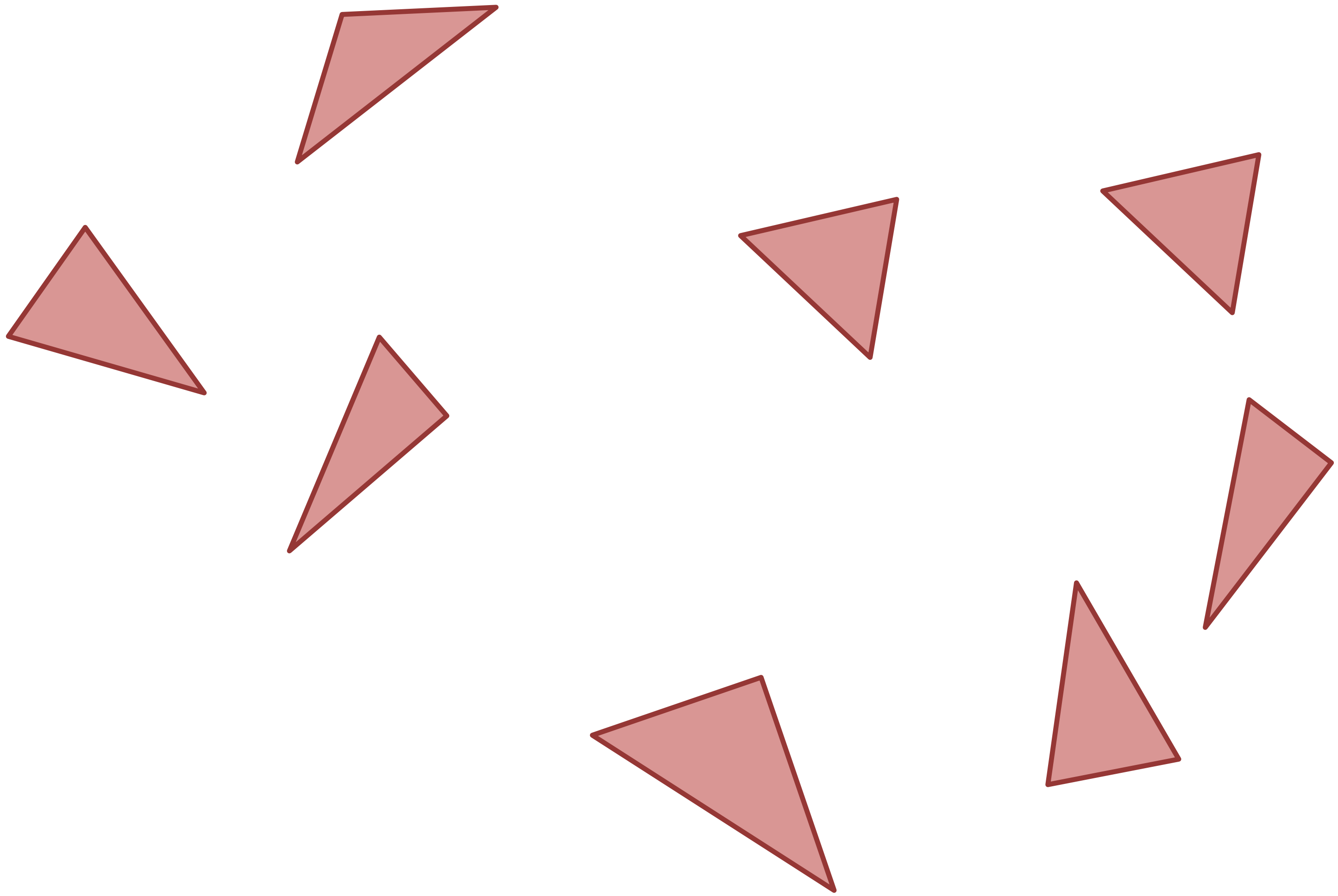
Axis Aligned Bounding Box Tree



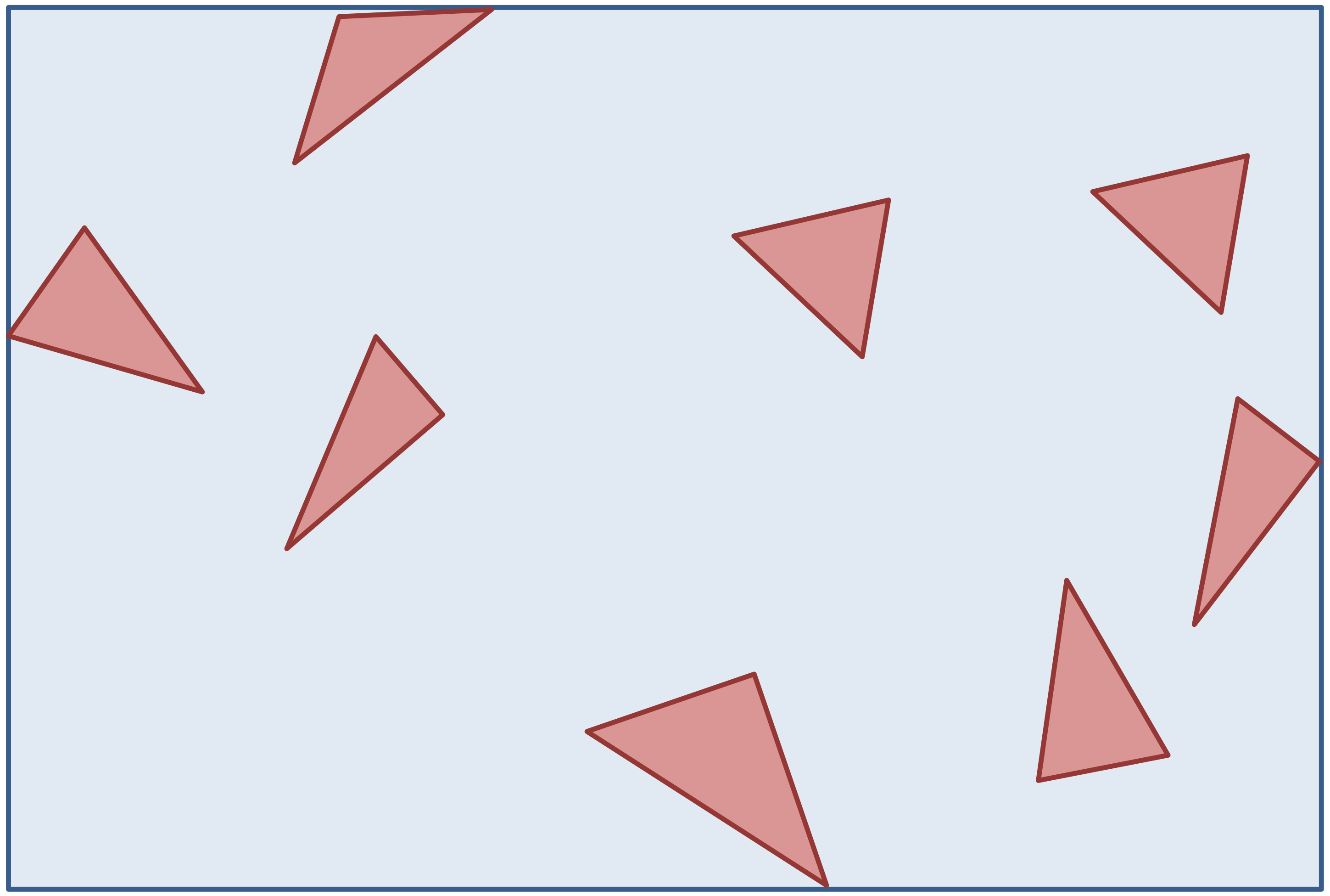
Tree Construction

- Typical tree construction algorithms operate in either a top-down or bottom-up fashion
- Top-down approaches start with the bounding box of the entire set and recursively split it into smaller boxes
- Bottom-up approaches start by building a box around each tetrahedron and them combine nearby boxes to form an agglomerated tree
- Both approaches are reasonable. Bottom-up methods can tend to produce more efficient tree structures leading to faster collision detection, but top-down methods tend to be faster to construct

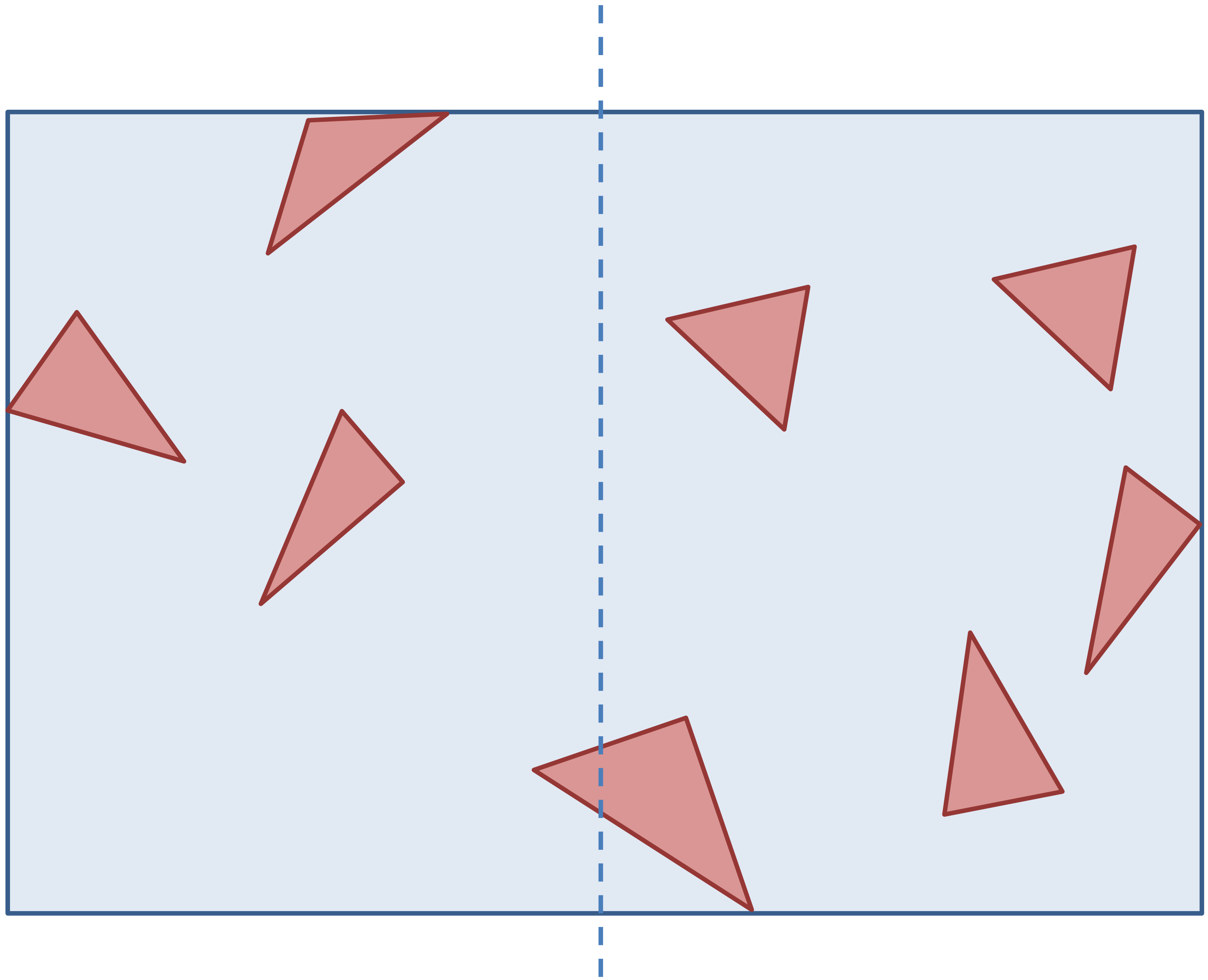
AABB-Tree



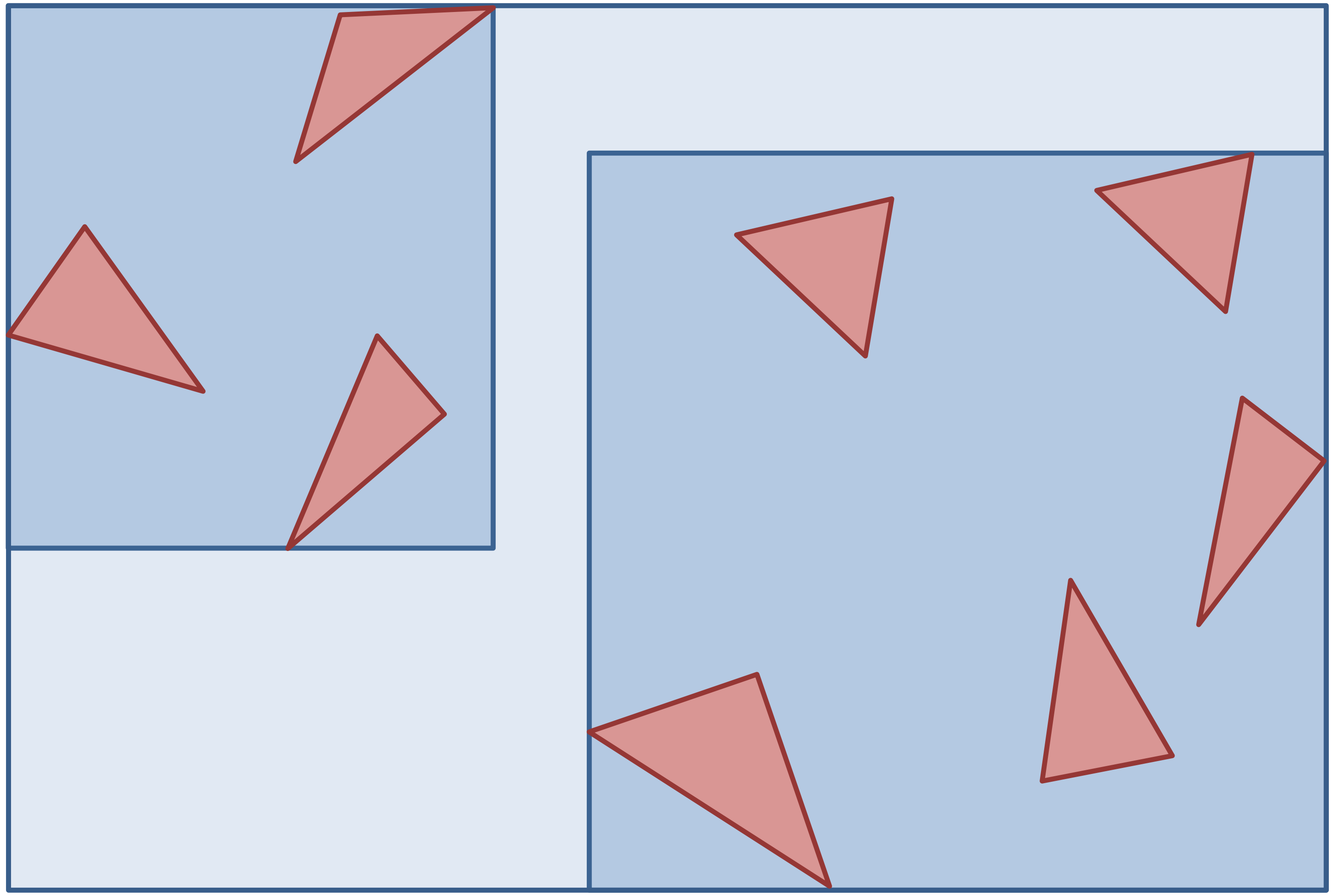
AABB-Tree



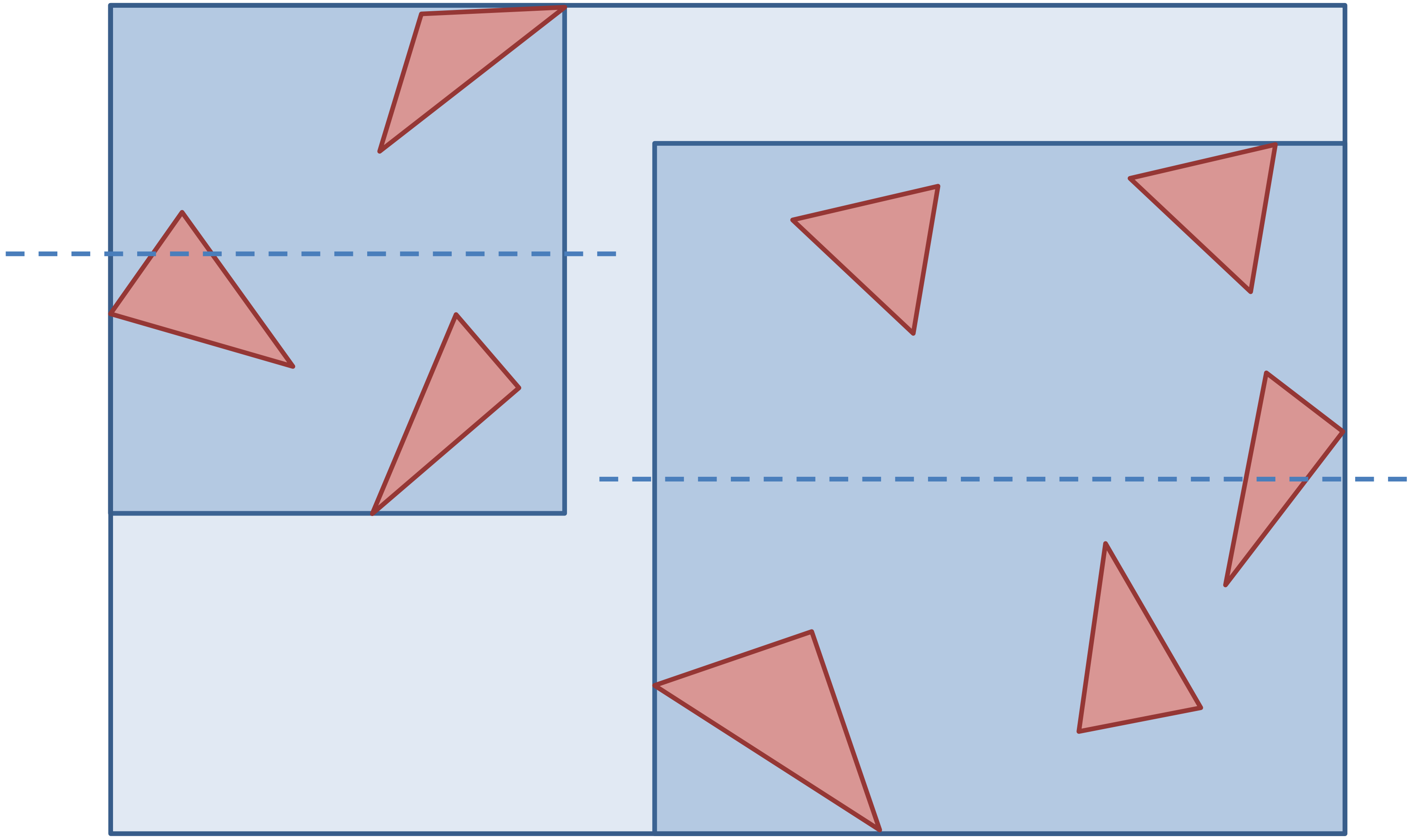
AABB-Tree



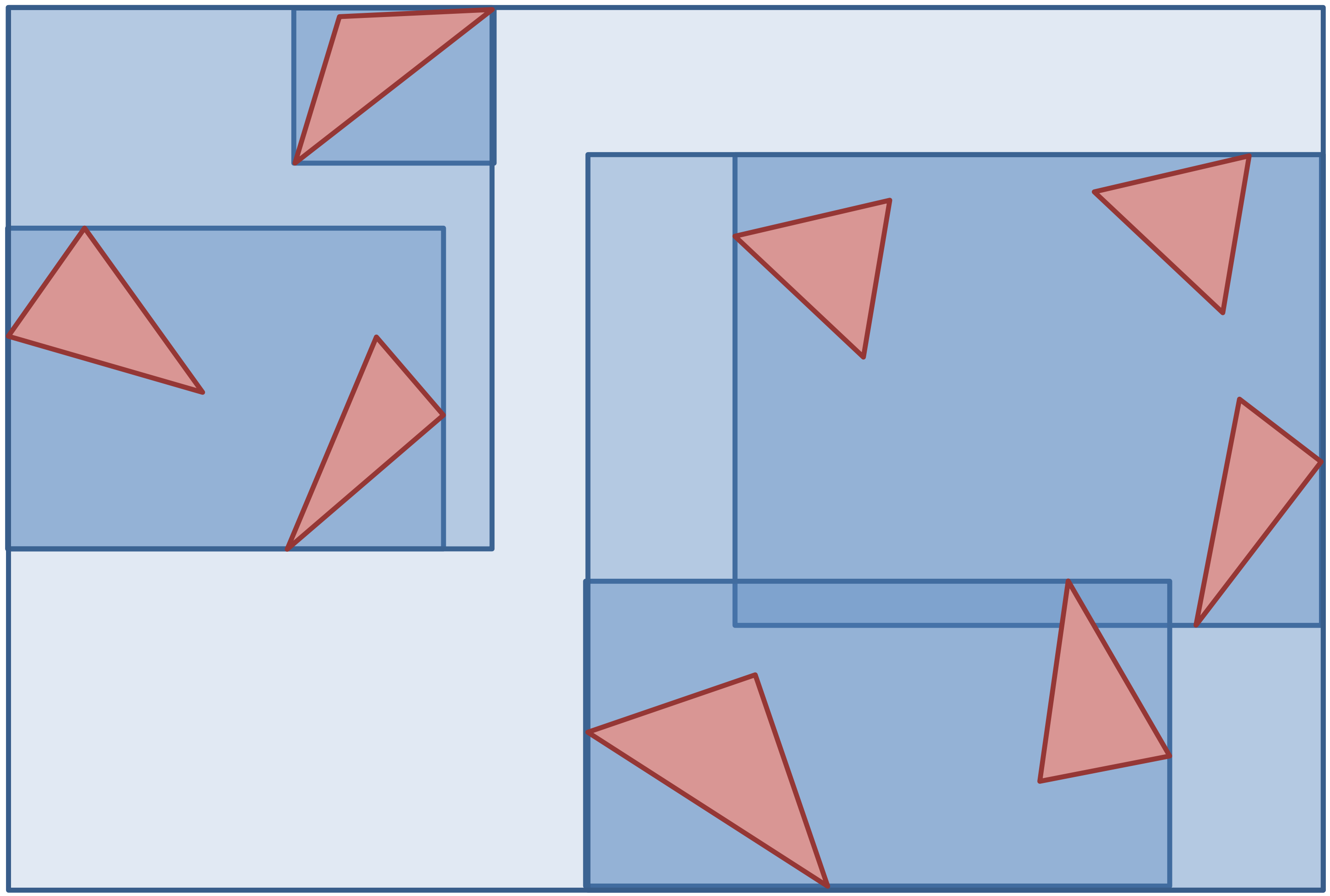
AABB-Tree



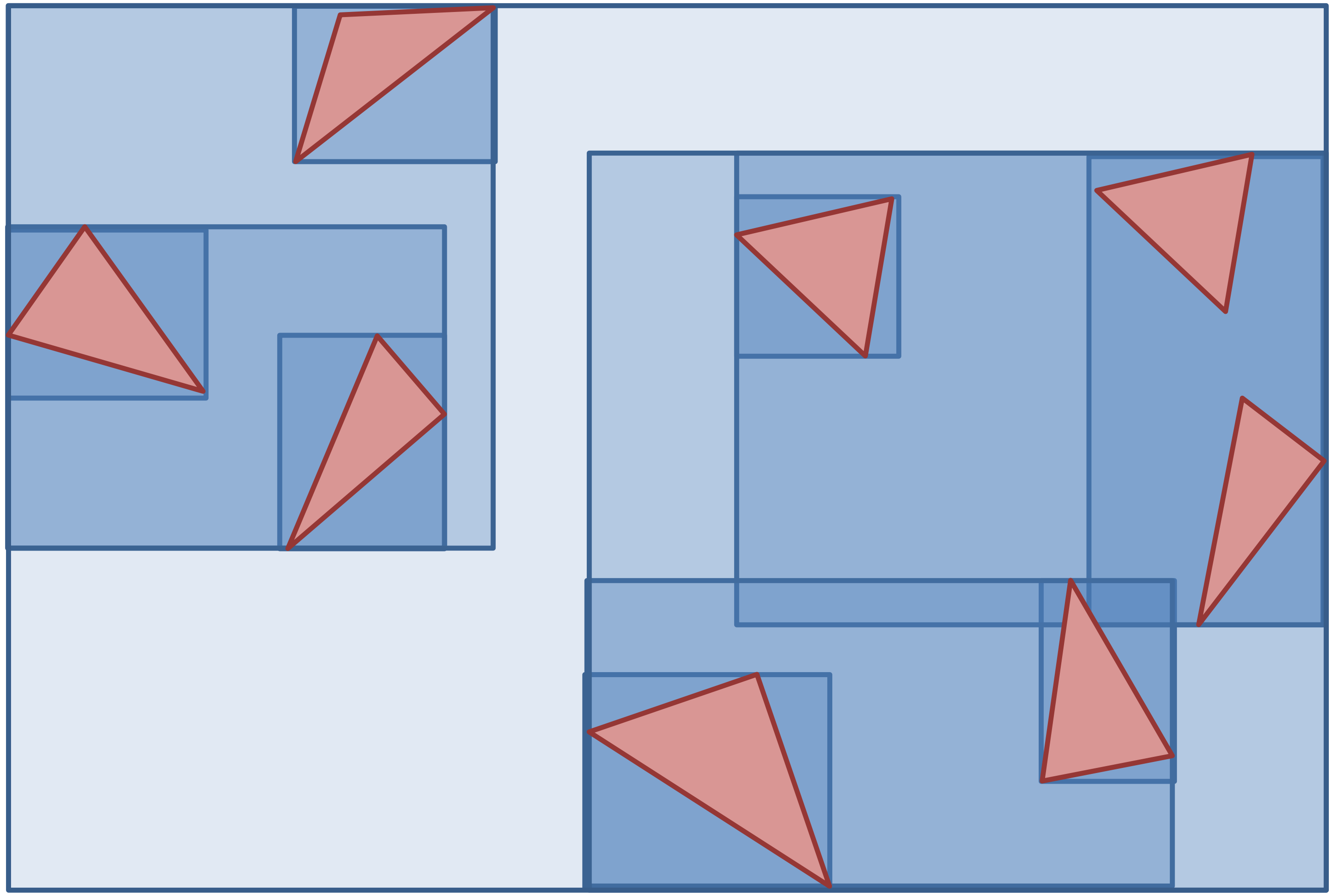
AABB-Tree



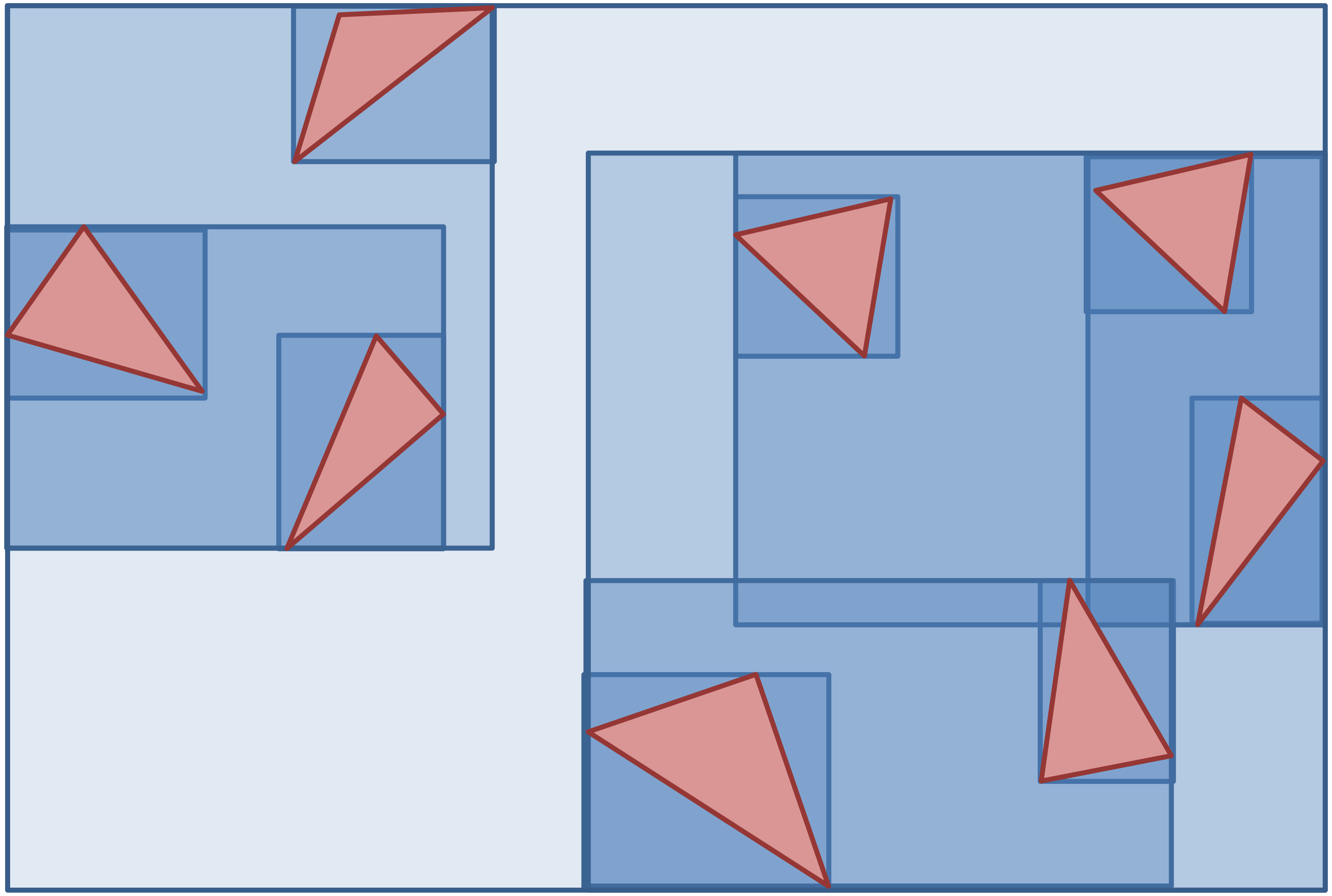
AABB-Tree



AABB-Tree



AABB-Tree



Intersection Testing

- Once the tree is constructed, we can do a recursive tree-tree test to determine all overlapping pairs
- There are some versions of this approach that keep track of information from frame to frame to cache the most likely areas of intersections to improve performance in situations with good temporal coherence

Tree Motion Update

- A big advantage to using the AABB tree for collision detection is that it can be re-used from frame to frame with minimal effort
- We assume that all of the elements will be moving, and so the entire tree will need to be updated
- However, we assume the movement will be small in any one frame, and so we can keep the structure of the tree the same and only update the boxes
- We perform a bottom-up traversal starting at the leaf nodes
- For each leaf node, we recompute the bounding box around the elements
- Each interior node of the tree recomputes its bounding box around the box of its child nodes
- When we get back to the top, we've updated the entire tree
- This operation is linear in performance O(*n*)

Tree Reconstruction

- We can use the motion update algorithm to adjust the tree from frame to frame
- However, over time, large motions will cause the need for the tree structure to be adjusted or rebuilt from scratch
- The most basic approach would just be to periodically rebuild the tree, maybe every 100 time steps
- A smarter approach would be based on some measure of quality degradation, for example by looking at the change in the ratio of parent volumes to child volumes and rebuilding (possibly only a subtree) as the quality degrades beyond some tolerance

Collision References

- A couple references on collision detection with deformable models:
  - “Efficient BVH-based Collision Detection Scheme with Ordering and Restructuring”, Wang, Tang, Manocha, Tong, 2018
  - “Collision Detection for Deformable Objects”, Teschner, et al., 2004