Collision Detection

Steve Rotenberg
CSE169: Computer Animation
UCSD
Winter 2021

Collisions

- Collision Detection
  - *Collision detection* is the geometry problem of determining if two (or more) moving objects intersect
- Collision Response
  - *Collision response* is the physics problem of determining the forces resulting from an intersection

Collision Detection

Collision Detection

- *Collision detection* is a geometric intersection problem
- Main subjects:
  - Primitive intersection testing (triangles, spheres, lines…)
  - Optimization data structures (AABB tree, OBB tree…)
  - Pair reduction

Static and Dynamic Testing

- Dynamic collision detection: given two objects and their motion over some small time interval, determine if, where, and when the two objects intersect
- Static collision detection: given two objects in a current configuration, determine if and where they intersect

- Sometimes, we need to find all intersections. Other times, we just need the ‘first’ one. Sometimes, we simply need to know if they intersect but don’t need any details.

Primitives

- Complex collision geometry is built from different types of *primitives*
- Examples of primitives used in collision detection:
  - Triangles
  - Spheres
  - Cylinders (or round capped cylinders called *capsules*)
  - AABB (axis-aligned bounding box)
  - OBB (oriented bounding box)

- At the heart of intersection testing are various primitive-primitive intersection tests

Convex vs. Non-Convex

- There are various geometric collision detection algorithms optimized for convex geometry
- This can put some limitations on generality, so for today, we won’t focus on these methods and stick to more general non-convex cases

Particle Collisions

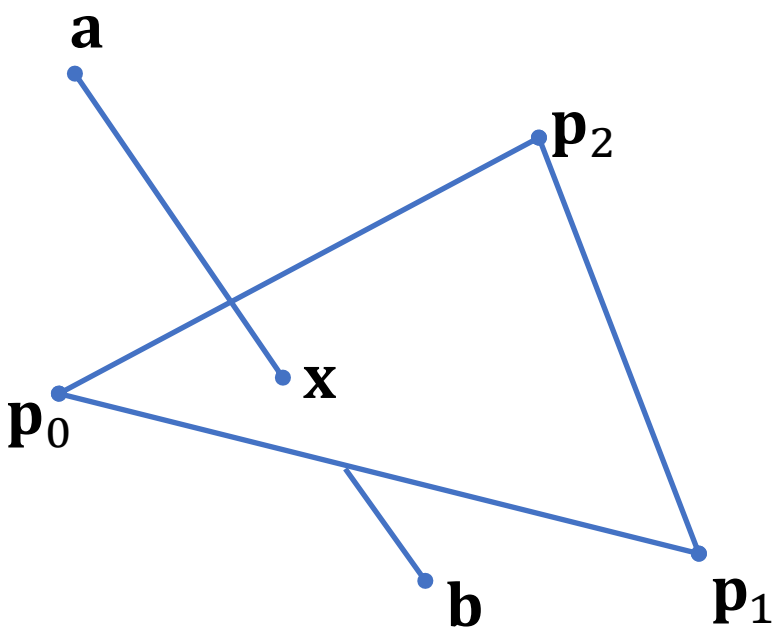
- We'll also mainly focus on the issue of colliding particles (or cloth) with mesh geometry, and won't focus too much on full object-object collisions
- We can treat a moving particle as a line segment going from its previous position to its new (candidate) position
- The candidate position is the result of the forward Euler integration and is where the particle will end up if there are no collisions
- If we find a collision, we have to compute the collision response and a new ending position

Collision Detection Issues

- Performance
- Memory usage
- Accuracy
- Floating point precision

Line Segment vs. Triangle

- Consider the case of testing line segment $\mathbf{ab}$ against triangle $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$:



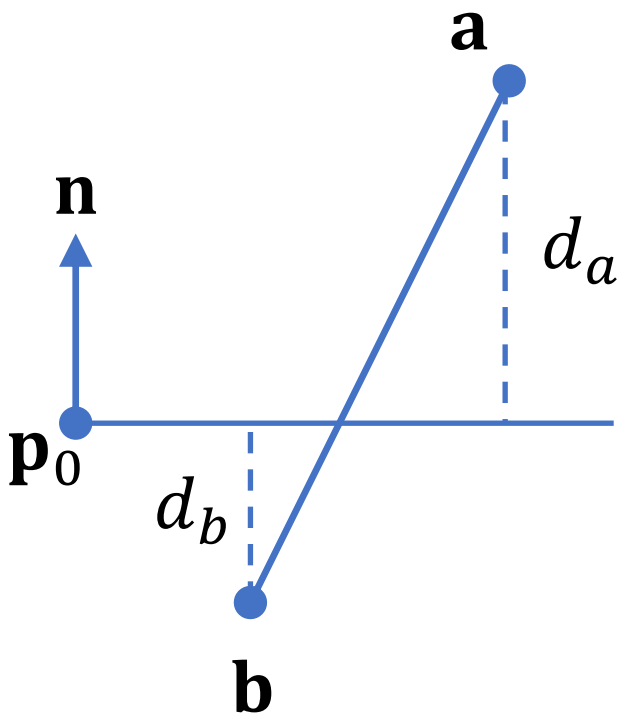
Line Segment vs. Triangle

- First, compute signed distances of $\mathbf{a}$ and $\mathbf{b}$ to the plane:

$$d_a = (\mathbf{a} - \mathbf{p}_0) \cdot \mathbf{n}$$
$$d_b = (\mathbf{b} - \mathbf{p}_0) \cdot \mathbf{n}$$

- Where $\mathbf{n}$ is the unit length normal of the triangle
- Reject if both are above or both are below plane
- Otherwise, find intersection point $\mathbf{x}$:

$$\mathbf{x} = \frac{d_a \mathbf{b} - d_b \mathbf{a}}{d_a - d_b}$$



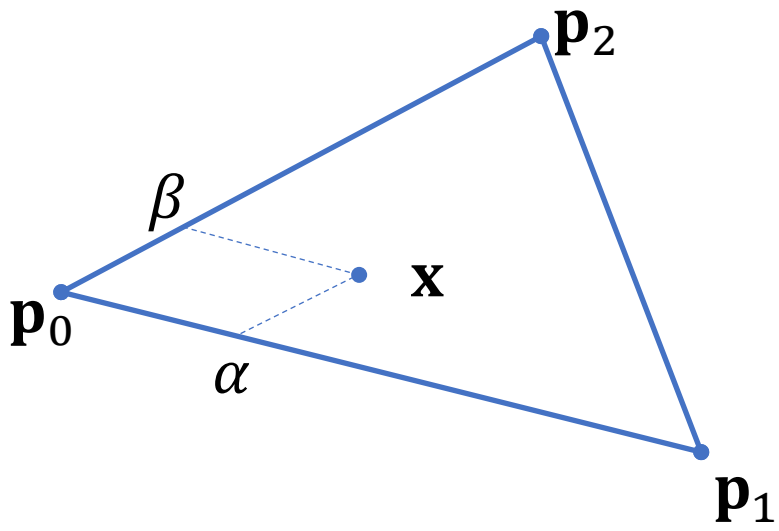
Point Inside Triangle

- To determine if the point $\mathbf{x}$ is inside the triangle, we compute the barycentric coordinates α and β :

$$\mathbf{u} = \mathbf{p}_1 - \mathbf{p}_0$$
$$\mathbf{v} = \mathbf{p}_2 - \mathbf{p}_0$$
$$\mathbf{w} = \mathbf{x} - \mathbf{p}_0$$

$$\alpha = \frac{(\mathbf{u}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{v}) - (\mathbf{v}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{u})}{(\mathbf{u}\cdot\mathbf{v})^2 - (\mathbf{u}\cdot\mathbf{u})(\mathbf{v}\cdot\mathbf{v})}$$

$$\beta = \frac{(\mathbf{u}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{u}) - (\mathbf{u}\cdot\mathbf{u})(\mathbf{w}\cdot\mathbf{v})}{(\mathbf{u}\cdot\mathbf{v})^2 - (\mathbf{u}\cdot\mathbf{u})(\mathbf{v}\cdot\mathbf{v})}$$



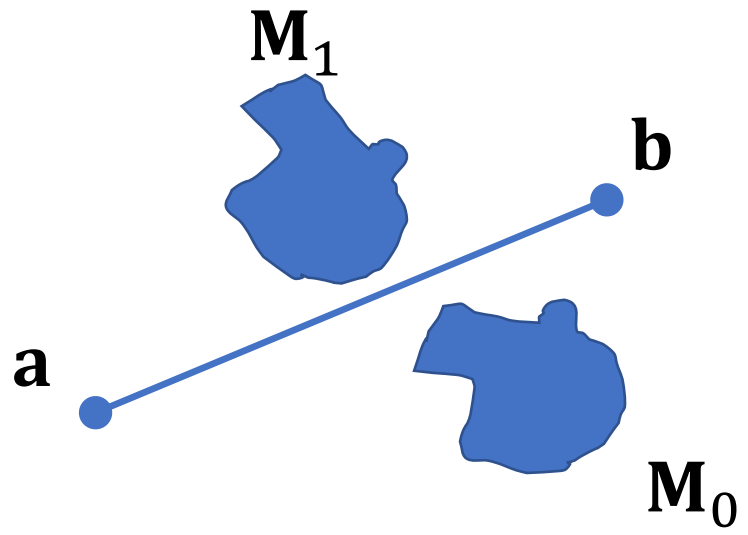
- The point is inside the triangle if $\alpha \geq 0$, $\beta \geq 0$ and $\alpha + \beta \leq 1$

Line Segment vs. Mesh

- To test a line segment against a mesh of triangles, simply test the segment against each triangle (we'll look at faster methods later)
- Sometimes, we are interested in only the first hit along the line segment, other times, we need all intersections, other times maybe just any one of them

Line Segment vs. Moving Mesh

- Say that our line segment represents a particle moving from point $\mathbf{a}$ at time t_0 to point $\mathbf{b}$ at time t_1
- We want to test against a moving object with matrix $\mathbf{M}_0$ at time t_0 and matrix $\mathbf{M}_1$ at time t_1



Segment vs. Moving Mesh

- We compute the *delta* matrix $\mathbf{M}_\Delta$

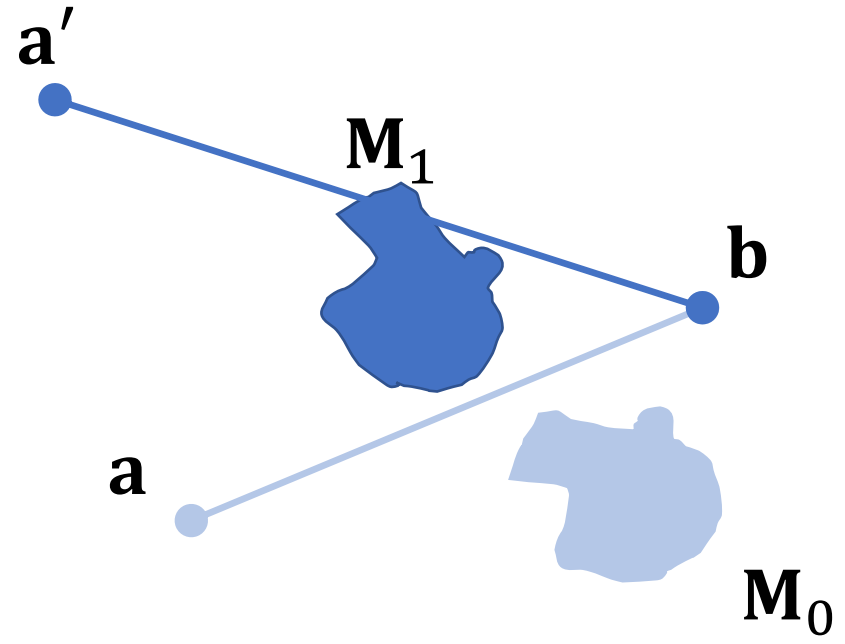
$$\mathbf{M}_1 = \mathbf{M}_0 \cdot \mathbf{M}_\Delta$$

$$\mathbf{M}_\Delta = \mathbf{M}_0{}^{-1} \cdot \mathbf{M}_1$$

- Then transform $\mathbf{a}$ by $\mathbf{M}_\Delta$

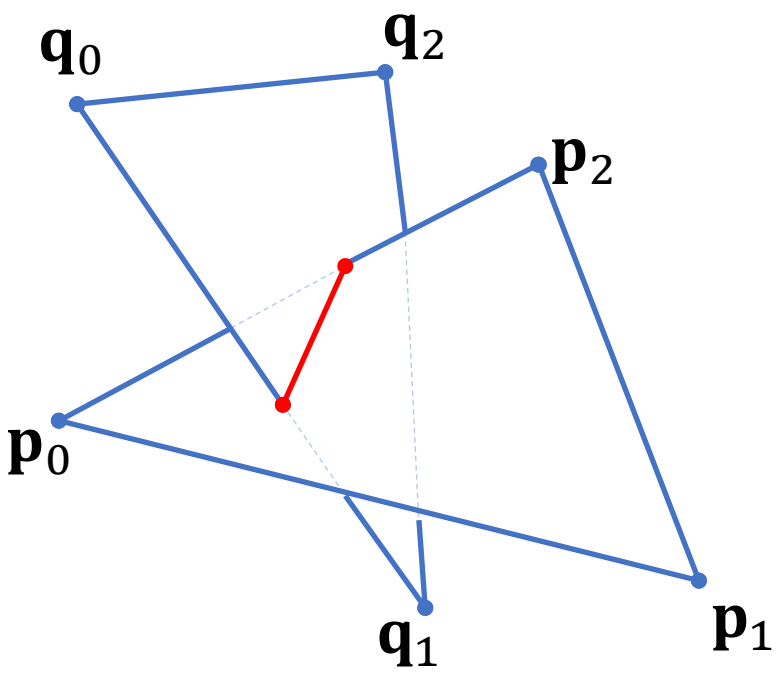
$$\mathbf{a}' = \mathbf{a} \cdot \mathbf{M}_\Delta$$

- And then test segment $\mathbf{a}'\mathbf{b}$ against the object at matrix $\mathbf{M}_1$



Triangle vs. Triangle

- Consider testing triangle $\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$ against triangle $\mathbf{q}_0\mathbf{q}_1\mathbf{q}_2$:



Triangle vs. Triangle

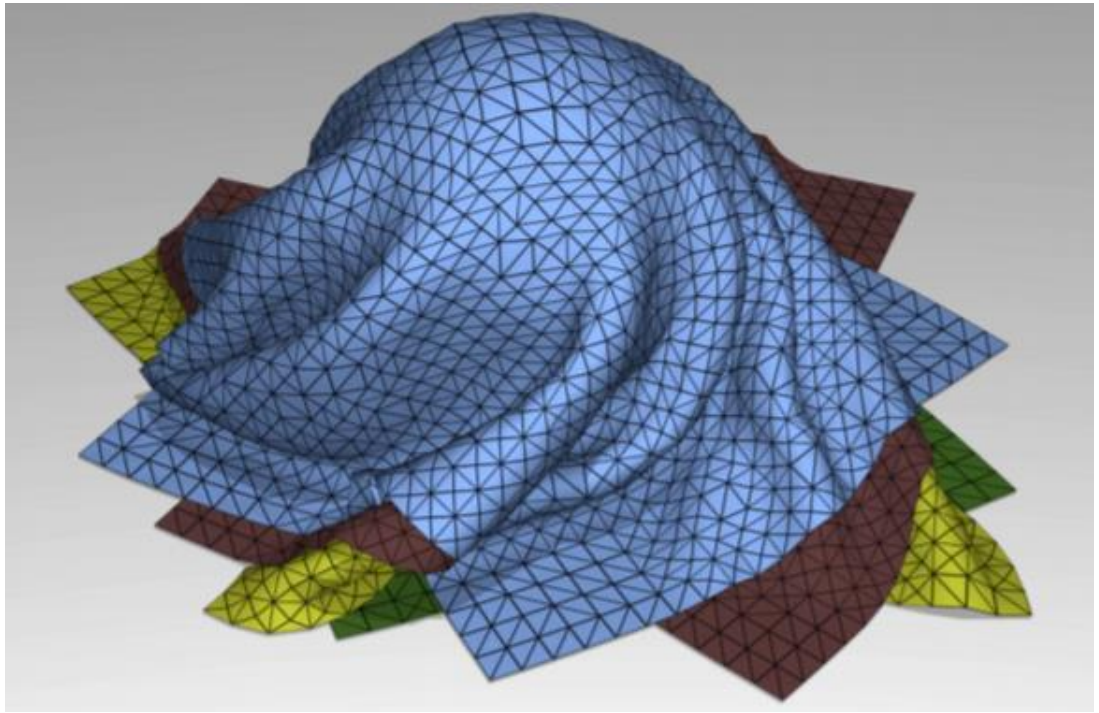
- To test two triangles, we start by computing the signed distance of the verts of each one to the plane of the other
- If they are all on one side, there is no intersection
- Otherwise, we intersect the two edges of one with the plane of the other in the same fashion as we did for a single segment
- We then clip the line segment connecting the two points to the triangle in the 2D plane
- If any part of the line segment remains, it represents the intersection of the two triangles

Mesh vs. Mesh

- To test two static triangle meshes for intersection, we can test all triangles of each against all triangles of the other (we'll talk about faster ways later)
- If each colliding pair produces a line segment, we can connect up all of those line segments into a closed loop called a *collision manifold*
- Note that this assumes that meshes are clean boundary meshes without self-intersections
- Collision manifolds can be useful for classifying the collision type as point-like, line-like, or plane-like, or determining cases of multiple disjoint overlaps

Continuous Collision Detection (CCD)

- A variety of *continuous collision detection* methods have been devised over the years
- These methods use starting and ending configurations for each object to test if any collisions happened in the time between
- Related approaches are also useful for handling highly deformable objects like cloth



Continuous Collision Detection

- “Fast Continuous Collision Detection Between Rigid Bodies”, Redon, Kheddar, Coquillart, 2002
- This paper introduced a popular method that has been extended in various ways
- The basic approach assumes that each object moves along a straight path while rotating at a constant angular velocity. This implies that any point on the objects will follow a helical path (sometimes called *screw motion*)
- Algebraic equations are set up to test the motion trajectories of points and edges of the objects and these are evaluated using *interval arithmetic* to compute bounded results

CCD References

- Some good CCD papers:
- “Fast and Exact Continuous Collision Detection with Bernstein Sign Classification”, Tang, Tong, Wang, Manocha, 2014
- “Efficient Geometrically Exact Continuous Collision Detection”, Brochu, Edwards, Bridson, 2012
- “Continuous Collision Detection for Articulated Models using Taylor Models and Temporal Culling”, Zhang, Redon, Lee, Kim, 2007
- “Interactive Continuous Collision Detection for Non-Convex Polyhedra”, Zhang, Lee, Kim, 2006

Collision Response

Impact vs. Contact

- It can be useful to distinguish between momentary (instantaneous) *impacts* and finite time *contacts*
- With impacts, the closing velocity is positive before the collision, and with contacts, the closing velocity is zero (or very close)
- Some collision response methods handle impacts differently from contacts, while other methods don’t distinguish between the two

Impacts

- We’ll just look at impacts, and allow contacts to just be treated as lots of repeated small impacts
- This will work OK, but might lead to some vibration

Impulse

- Force is the derivative of momentum, and finite changes in momentum require forces acting over time
- An impulse $\mathbf{j}$ is a large force applied over a short time, such that it can be treated as an instantaneous change in momentum, rather than a rate of change over time

$$\mathbf{j} = \int \mathbf{f} dt = \Delta \mathbf{p}$$

- Impulses operate in a very similar fashion as forces, except they affect the velocity instead of the acceleration
- As with forces, impulses obey Newton's Third Law (action/reaction)
- An impulse can be applied to a particle resulting in an instantaneous change in momentum $\mathbf{p}$ and therefore velocity $\mathbf{v}$

$$\Delta \mathbf{p} = \mathbf{j} = m \Delta \mathbf{v}$$
$$\Delta \mathbf{v} = \frac{1}{m} \mathbf{j}$$

Compression vs. Restitution

- Even if we treat our collisions as instantaneous, we can still break that into two hypothetical phases of *compression* and *restitution*
- At the start of the *compression* phase, the closing velocity is positive (i.e., the objects are approaching each other at the collision site)
- During compression, both objects deform at the collision site until the closing velocity reaches zero
- Then during *restitution*, the objects return to their original shape and release the stored deformation energy
- If energy is lost during compression, the restitution will only restore a portion of the original energy
- In reality, this can be complex, but we can model the process with a single constant ε called the *coefficient of restitution*

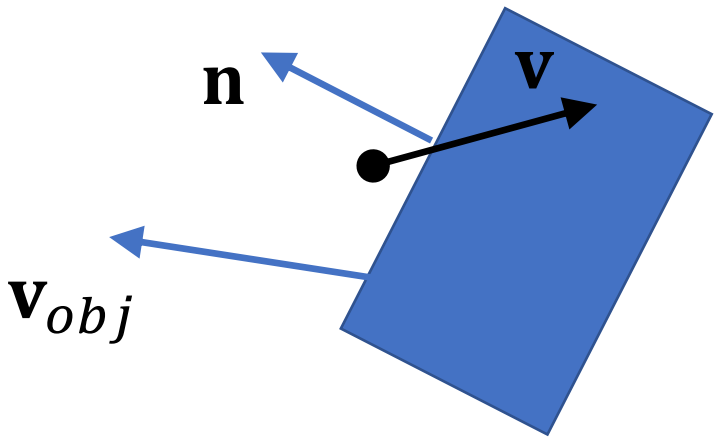
Collision Restitution

- The *coefficient of restitution* ε will range from 0 (perfectly inelastic) to 1 (perfectly elastic)

- Note: there are different ways to define the exact meaning of this constant, and it is ultimately just an approximation to a far more complex process that is dependent on the material properties and shape of the object
- Note: sometimes called coefficient of elasticity

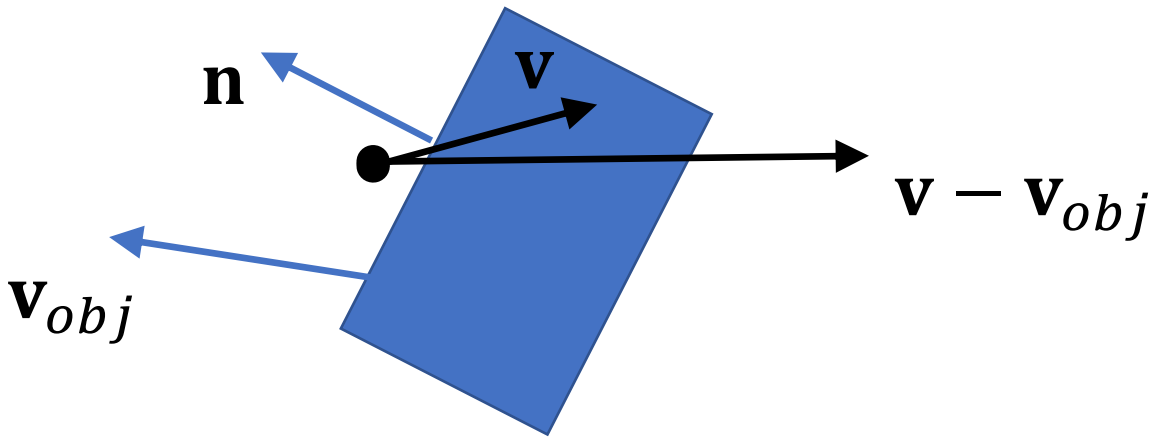
Particle Collision with a Moving Surface

- Consider a frictionless particle colliding with a heavy moving object
- The particle has mass m and velocity $\mathbf{v}$ before the collision
- The object is moving with velocity $\mathbf{v}_{obj}$
- The normal at the collision point is $\mathbf{n}$
- We want to find the impulse $\mathbf{j}$ applied to the particle



Particle Collision with a Moving Surface

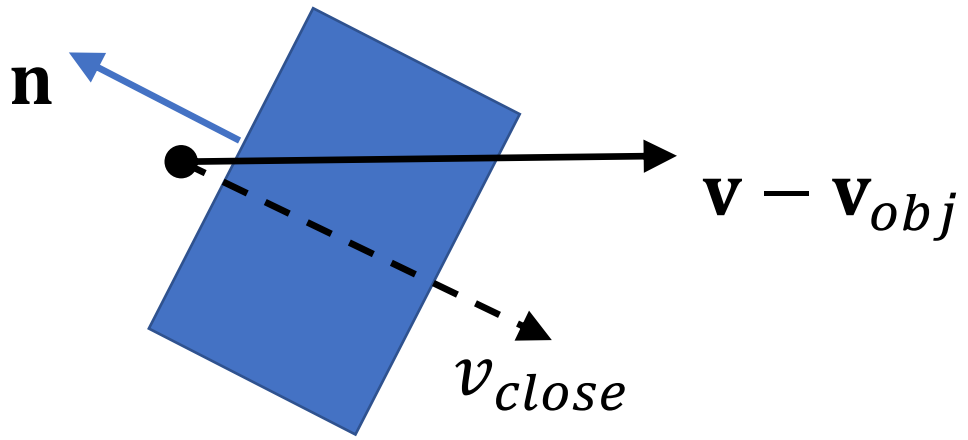
- We can look at things from the moving frame of reference of the object and assume the object is still and the particle is moving with velocity $\mathbf{v} - \mathbf{v}_{obj}$



Particle Collision with a Moving Surface

- For a frictionless collision, the impulse will act along the direction of the normal and will oppose closing velocity along the normal

$$v_{close} = \left(\mathbf{v} - \mathbf{v}_{obj}\right) \cdot \mathbf{n}$$

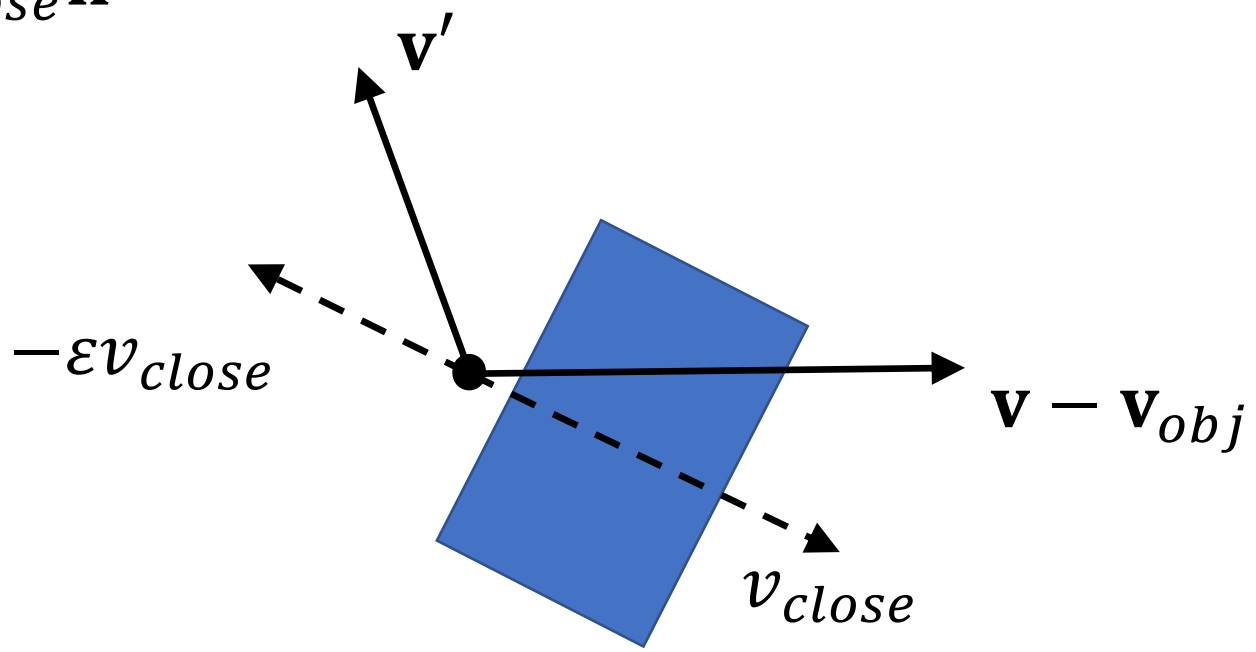


Particle Collision with a Moving Surface

- We find the impulse **j** that causes the closing velocity after the collision to be flipped and scaled by the restitution ε

$$\mathbf{v}' \cdot \mathbf{n} = -\varepsilon v_{close}$$

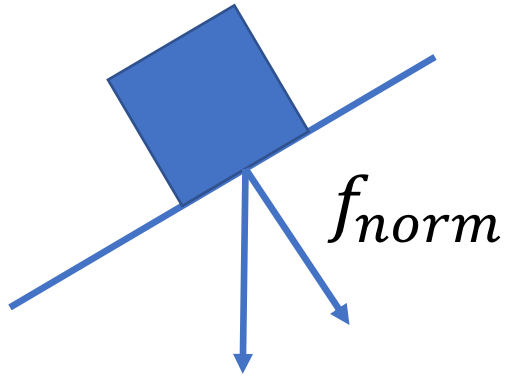
$$\mathbf{j} = -(1+\varepsilon) m v_{close} \mathbf{n}$$



Coulomb Friction

- The simple but effective Coulomb friction model says:

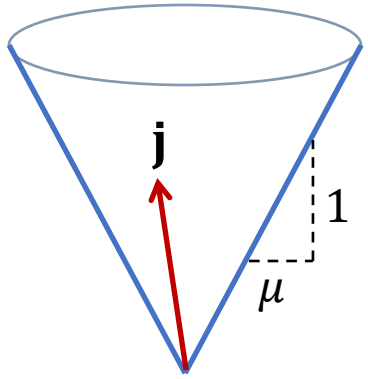
$$f_{stat} \leq \mu_s f_{norm}$$
$$f_{dyn} = \mu_d f_{norm}$$



- 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
- The coefficients range from 0 (perfectly frictionless) to around 1.5 or so, but there really isn't a strict maximum, as materials could conceivably have adhesive (sticky) properties
- μ_s is typically a bit larger than μ_d

Friction Cone

- Consider a block sliding on a flat surface
- The ground applies a normal force and a friction force
- According to the Coulomb friction model, the two forces must add up to a vector that lies within the *friction cone*
- The same can be said about impulses resulting from impacts



Tangential Velocity

- We need to find the *tangential velocity* of the impact in order to determine the direction the friction will act
- If $\mathbf{n}$ is the surface normal of the contact point, we can project the velocity to get *normal* and *tangential* velocity vectors:

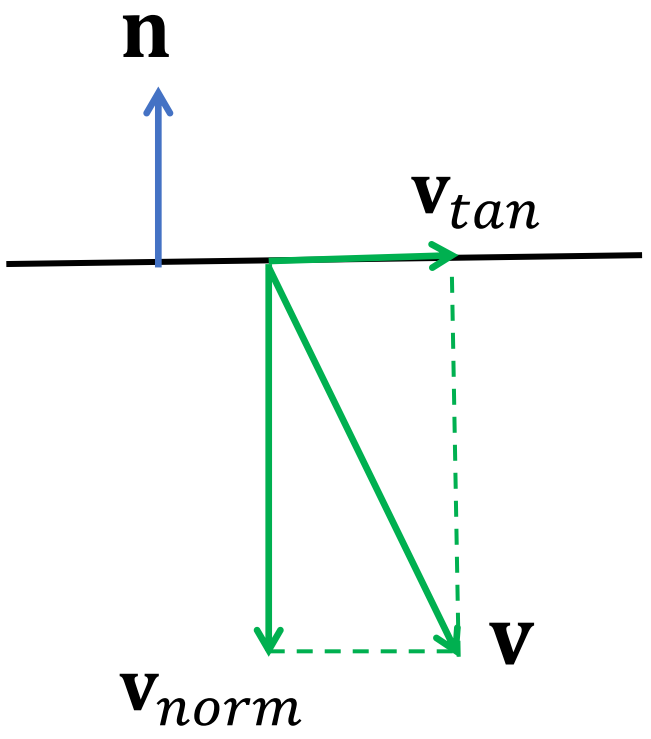
$$\mathbf{v}_{norm} = (\mathbf{v} \cdot \mathbf{n})\mathbf{n}$$

$$\mathbf{v}_{tan} = \mathbf{v} - \mathbf{v}_{norm}$$

Tangential Velocity

$$\mathbf{v}_{norm} = (\mathbf{v} \cdot \mathbf{n})\mathbf{n}$$

$$\mathbf{v}_{tan} = \mathbf{v} - \mathbf{v}_{norm}$$



Friction Handling

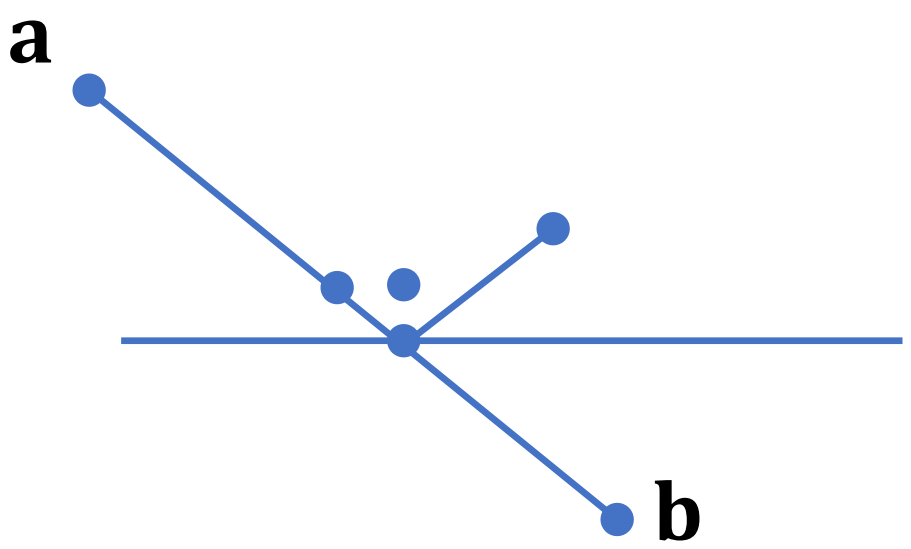
- Correct friction handling can get tricky, especially with potential switching between static & dynamic friction modes
- A simple way to handle it is to just compute the frictionless impulse $\mathbf{j}$ as before, and then assume that an additional friction impulse is applied in the opposite direction as the tangential velocity, and of magnitude $\mu_d|\mathbf{j}|$
- It should also be capped to prevent the tangential velocity from switching direction

Collision Handling

- For handling particles (& cloth) colliding with static & moving objects, we treat the moving particles as line segments
- We handle moving objects with the delta matrix technique discussed earlier
- For multiple moving objects in the scene, we examine all objects individually and find the closest intersection. As we may change the length of the line segment for moving objects, we can use the proportion along the line segment that the collision occurred to find the first collision
- When a collision is detected, we compute the impulse and immediately change the particle velocity as a result
- Even after this, the particle is still in an illegal position and something must be done to correct the final position

Position Adjustment

- There are various options for how to handle this, each with its own impact on performance, accuracy, robustness, etc.



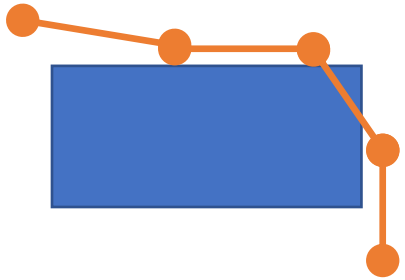
Bouncing

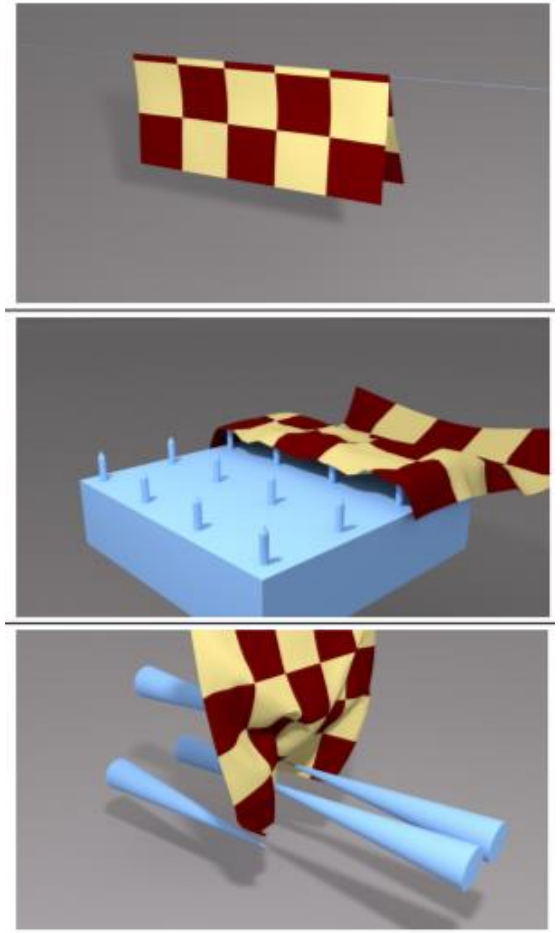
- Computing the bounced position is the best approach, as it is consistent with the rest of the physics model
- We need to determine when exactly the collision happened (we can just assume that the particle traveled at a constant velocity within the frame)
- We then compute the impulse and adjust the velocity
- Then, we move the particle forward by the amount of time remaining within the frame
- Ideally, we should then check collisions on this new path
- A particle getting stuck in a narrow crack might bounce several times, so we should put a cap on the maximum number of bounces allowed, then just stop the particle at some point if it exceeds the limit

Cloth Collisions

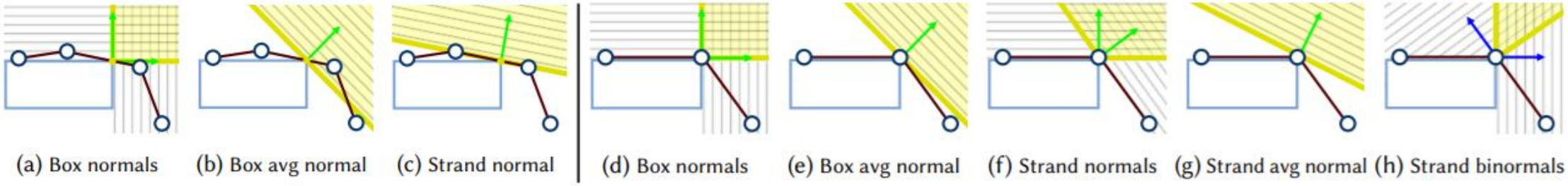
- For the cloth project, you only need to do collisions with a ground plane
- These are very easy to detect… just check if the final position (after Euler integration) is below the ground level
- You can compute the impulse and correct the velocity, and then push the particle back above the ground level to a legal position
- Cloth would have a very low restitution $\varepsilon \approx 0.05$ and a reasonably high friction, perhaps $0.5 \leq \mu \leq 0.75$

Cloth Collisions





- For more complex cases, our simple method of just testing collisions for the particles won't work very well for cloth
- There are various approaches to dealing with this, including recent methods…
- "Eulerian-on-Lagrangian Cloth Simulation", Macklin, Erleben, Muller, Chentanez, Jeschke, Corse, 2018



Collision Optimization

Multi-Level Collision Detection

- Scene level (broad phase)
  - At the scene level, we typically have a large number of objects moving around in a large static environment object, leading to potentially N^2 pair tests
  - We want to reduce this so we only check close pairs of objects for collisions
- Object level (mid phase)
  - At the object level, we have a complex geometry made up from possibly thousands of primitives (usually fixed relative to each other), leading to another potential N^2 type of test for each object pair
  - We need a way to efficiently check only potentially close pairs of primitives
- Primitive level (narrow phase)
  - At the primitive level we need to efficiently and robustly perform many individual primitive-primitive intersection tests

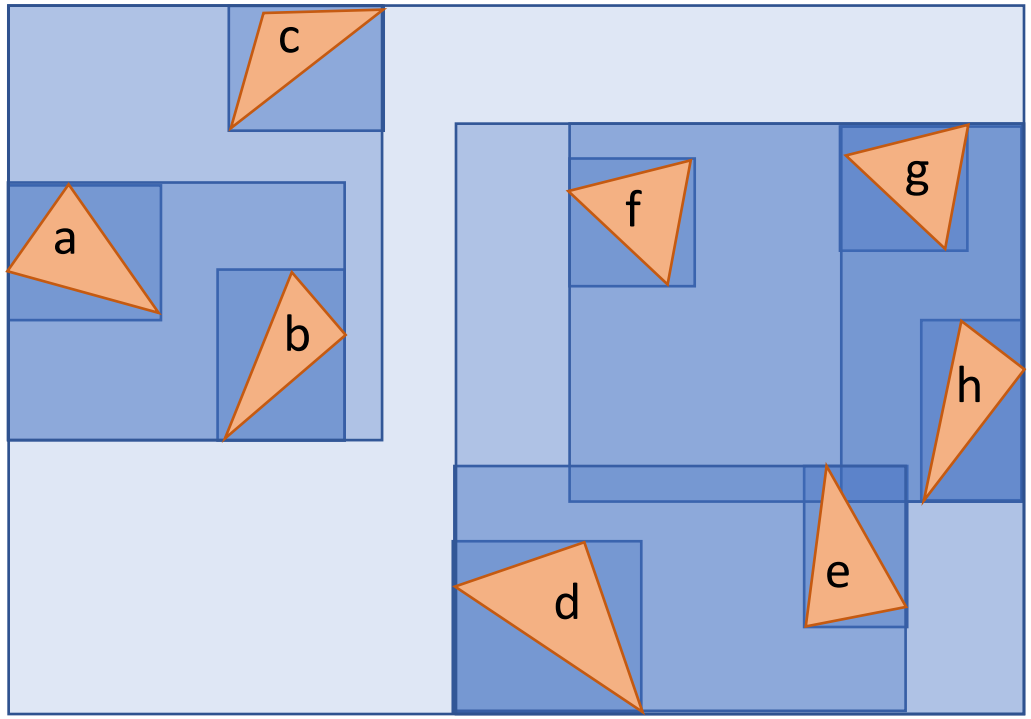
Spatial Data Structures

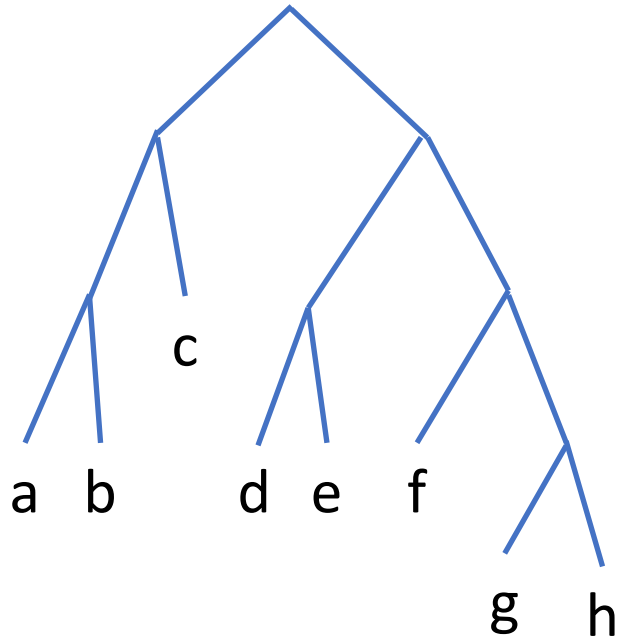
- Bounding volume hierarchies (BVH)
  - AABB tree (axis-aligned bounding box)
  - OBB tree (oriented bounding box)
  - Sphere tree
  - k-dop tree
- Spatial partitions
  - Octree
  - KD tree
  - BSP tree (binary space partition)
- Uniform grid
- Spatial hash

Bounding Volume Hierarchies

- *Bounding volume hierarchies* (BVH) are tree-like data structures that group sub-trees into simple bounding volumes like spheres or boxes
- The volumes can overlap, so that a point may be contained in several leaf volumes
- Common examples include:
  - Sphere tree
  - AABB tree (axis-aligned bounding box)
  - OBB tree (oriented bounding box)
  - k-DOP tree (discreet oriented polytope)

Axis Aligned Bounding Box Tree





BVH Intersections

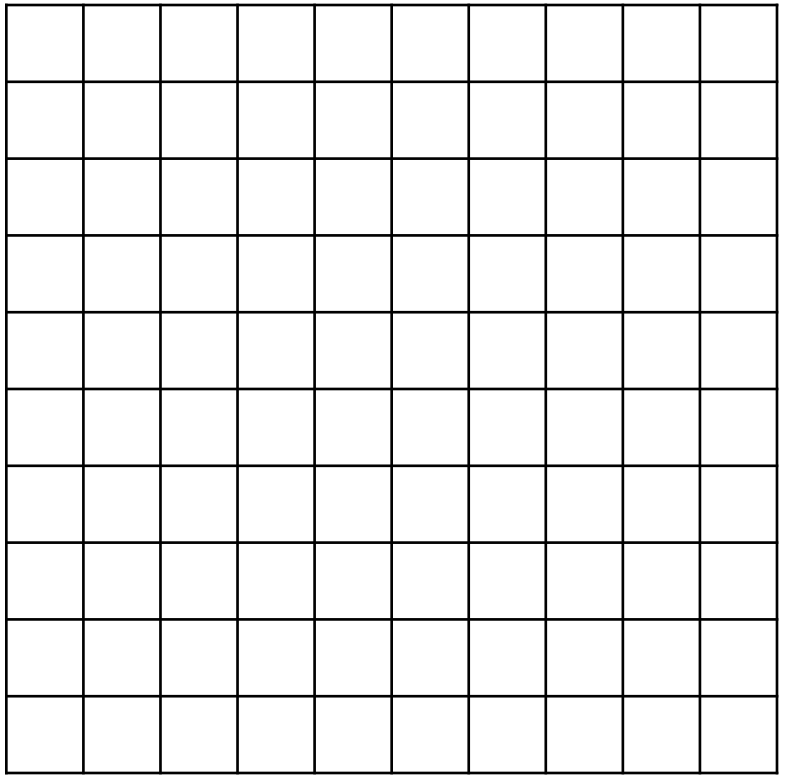
- The test the intersection of a primitive (like a line segment) with a BVH, we start at the top of the tree and determine if the primitive intersects the top volume
- If so, we examine the sub-volumes contained within
- If a sub-volume is intersected, we proceed to test its sub-volumes
- We eventually get down to the leaf nodes and test against the triangles contained
- For BVH-BVH intersections, a double hierarchical traversal algorithm is used (starting with top-level to top-level comparison and then proceeding down both trees)

Spatial Partitions

- *Spatial partitions* divide up space into volumes such that a point within the top level volume will end up in exactly one leaf volume
- Hierarchical examples (similar to BVH's):
  - Octree
    - Top level volume is a cube
    - Each level splits into 8 equal cubes
  - KD-Tree (k-dimensional)
    - Top level volume is a box or infinite
    - Each level splits into 2 at arbitrary point along x, y, or z
  - BSP-Tree (binary separating plane)
    - Top level volume is a convex polyhedron or infinite
    - Each level splits into 2 along an arbitrary plane
- Non-hierarchical examples:
  - Uniform grid
  - Spatial hash table

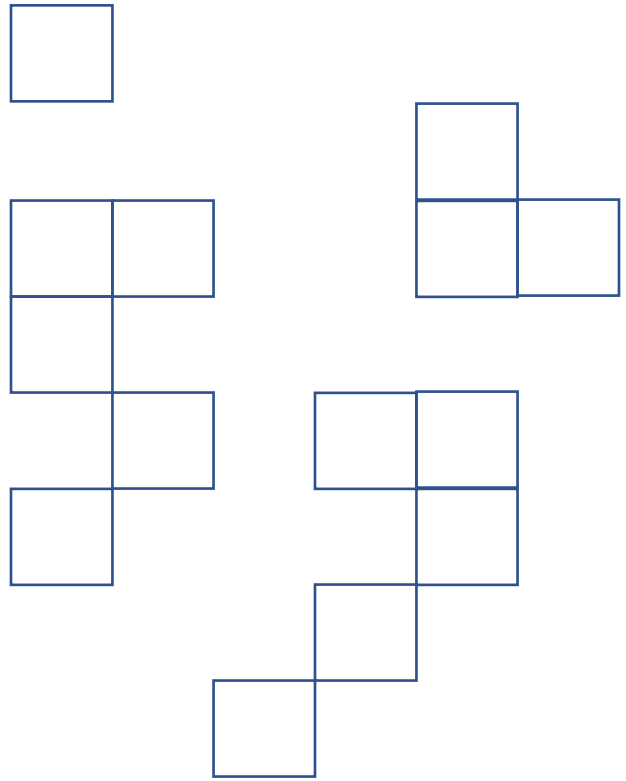
Uniform Grid

- We can use a uniform grid to speed up collisions
- We precompute which primitives overlap which cells, and store this information in the cells
- When a colliding primitive is near, we find which cells it overlaps and just test against the primitives in those cells
- This can be very fast, but doesn’t scale well for complex, detailed geometry



Spatial Hash

- A spatial hash table is similar to a uniform grid, except we don’t store a fixed sized block of cells – instead, we can have an (effectively) infinite grid of cells, but a cell only exists if it is occupied
- In situations with many similarly sized primitives, it is both fast and memory efficient
- This is implemented using the same mechanism as a standard hash table, and we’ll look closer at it in a later lecture



Pair Reduction

- When we have a handful of moving objects in a scene, we can quickly test all of them against each other by comparing bounding spheres
- For scenes with 1000's of objects, even quick bounding sphere tests end up taking a long time, as we have to test each pair of objects leading to the order of N^2 tests
- The subjects of *pair reduction* examines how to reduce this number

- Spatial hash tables are very effective for pair reduction on scenes where all objects are similarly sized
- The *sweep and prune* algorithm is designed for more general cases

Sweep and Prune

- The basic *sweep and prune* algorithm takes an axis (such as *x*-axis) and projects all objects to it to find their start and end interval along the axis
- Next, the min and max values for all objects are sorted into a list
- Then, we sweep down the list, maintaining an ‘active list’ that starts out empty
  - When we get to a new min value, we pair it with all objects on the active list and add these pairs to another list of potentially colliding pairs. We then add the new object to the active list
  - When we get to a new max value, we remove the associated object from the active list
- Once we finish with the axis, we can do the same thing for the y- and z-axes. Once we finish that, we find any pairs that overlap on all 3 axes and do further collision testing on them (mid or narrow phase)
- The basic algorithm is simple and fast and can be improved by using incremental sorting algorithms, as the sorted list changes very little from frame to frame
- There are many other enhancements to the algorithm, such as adaptations to parallel processors or GPUs