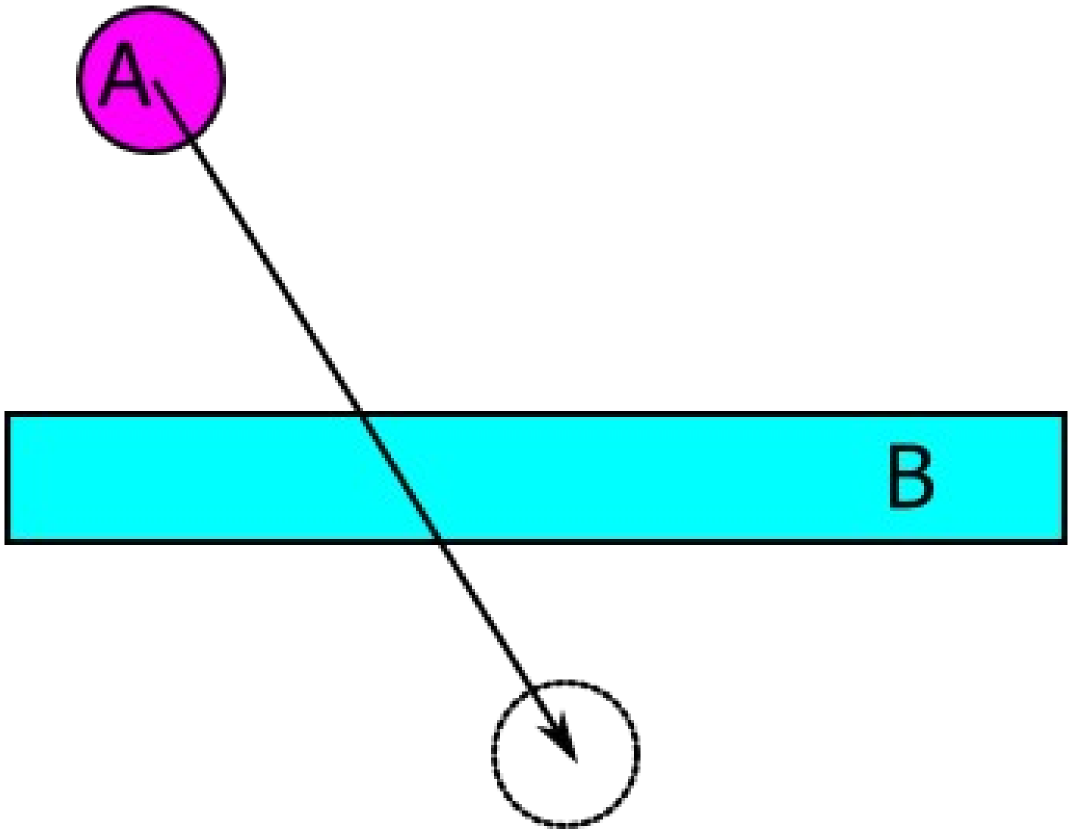
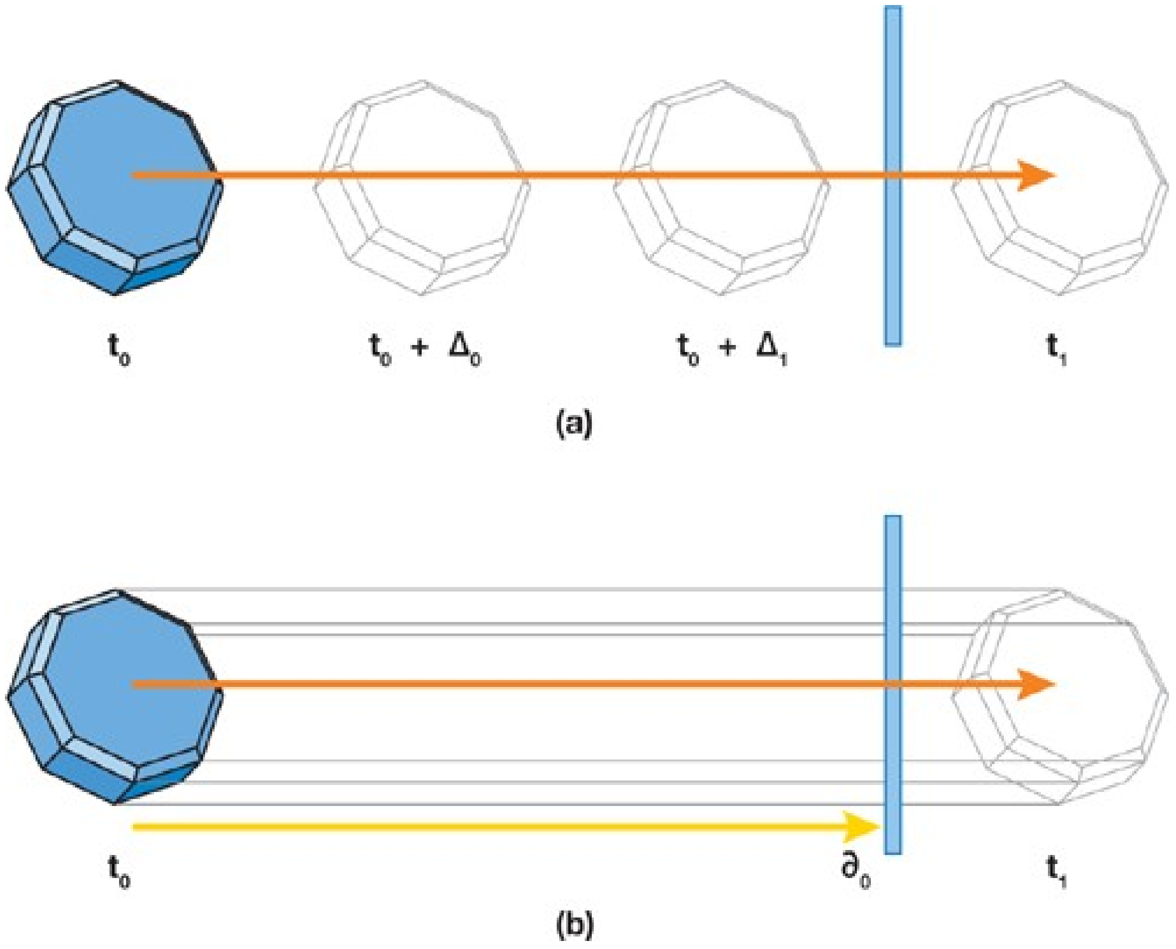
Continuous Collision Detection

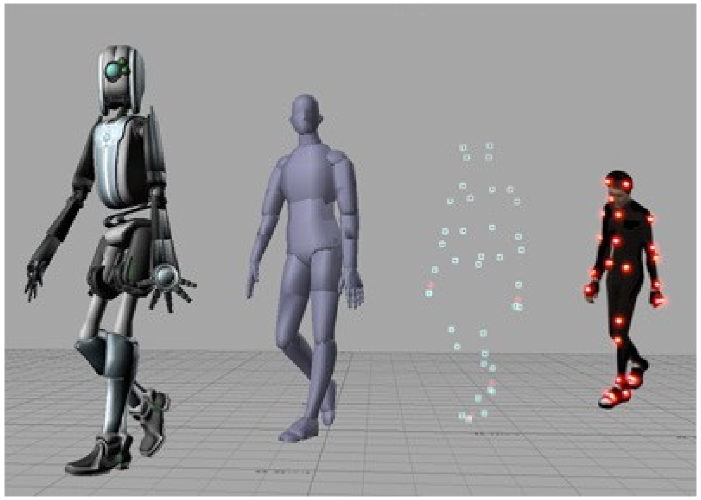


Problem

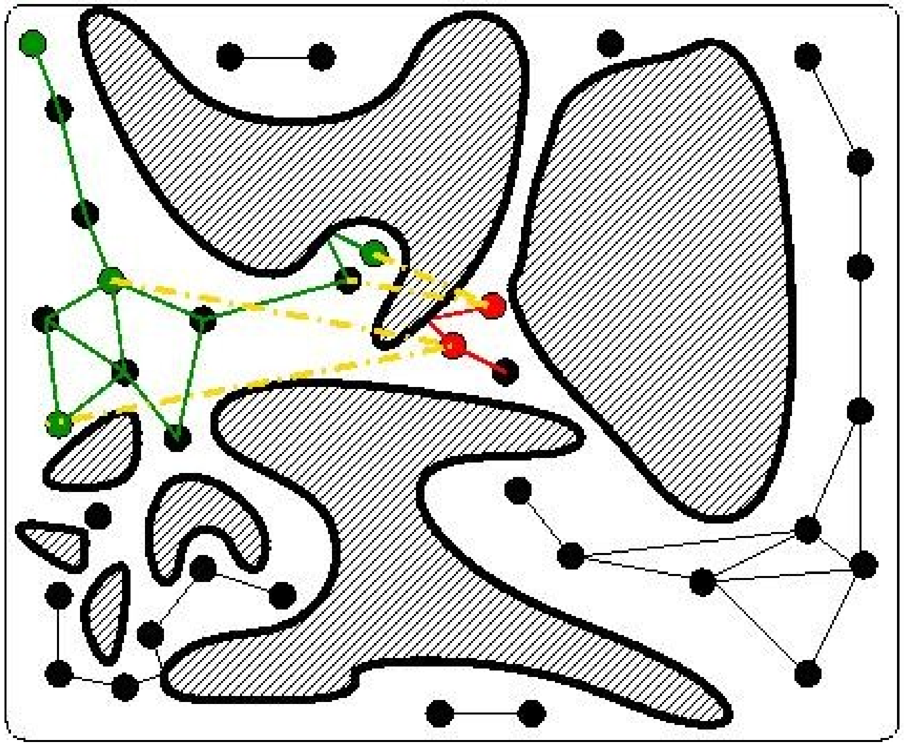
- Interpenetration
- Missed collisions

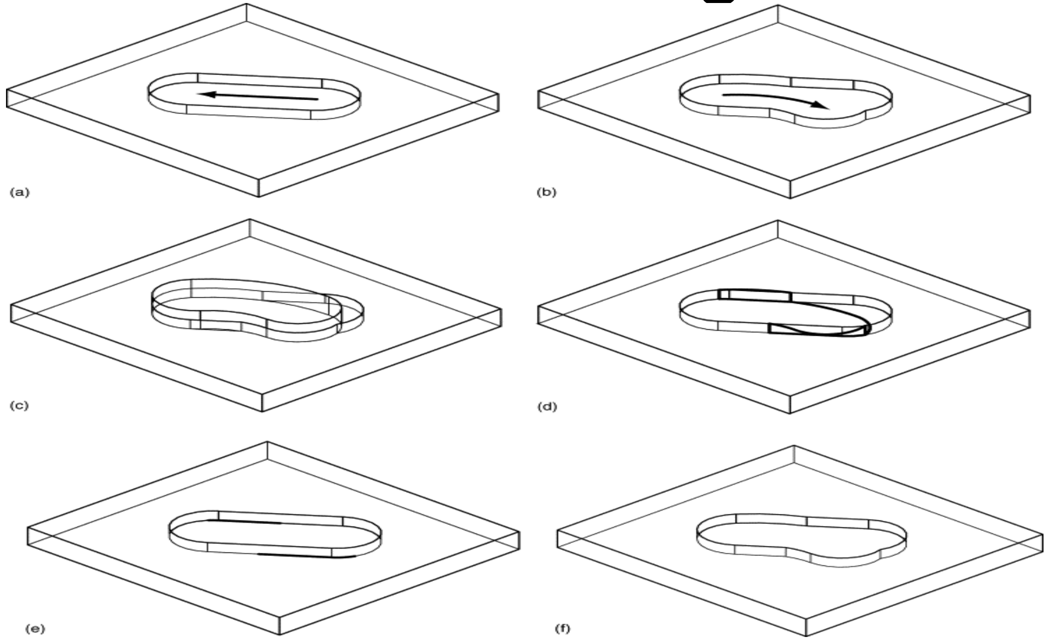


Applications

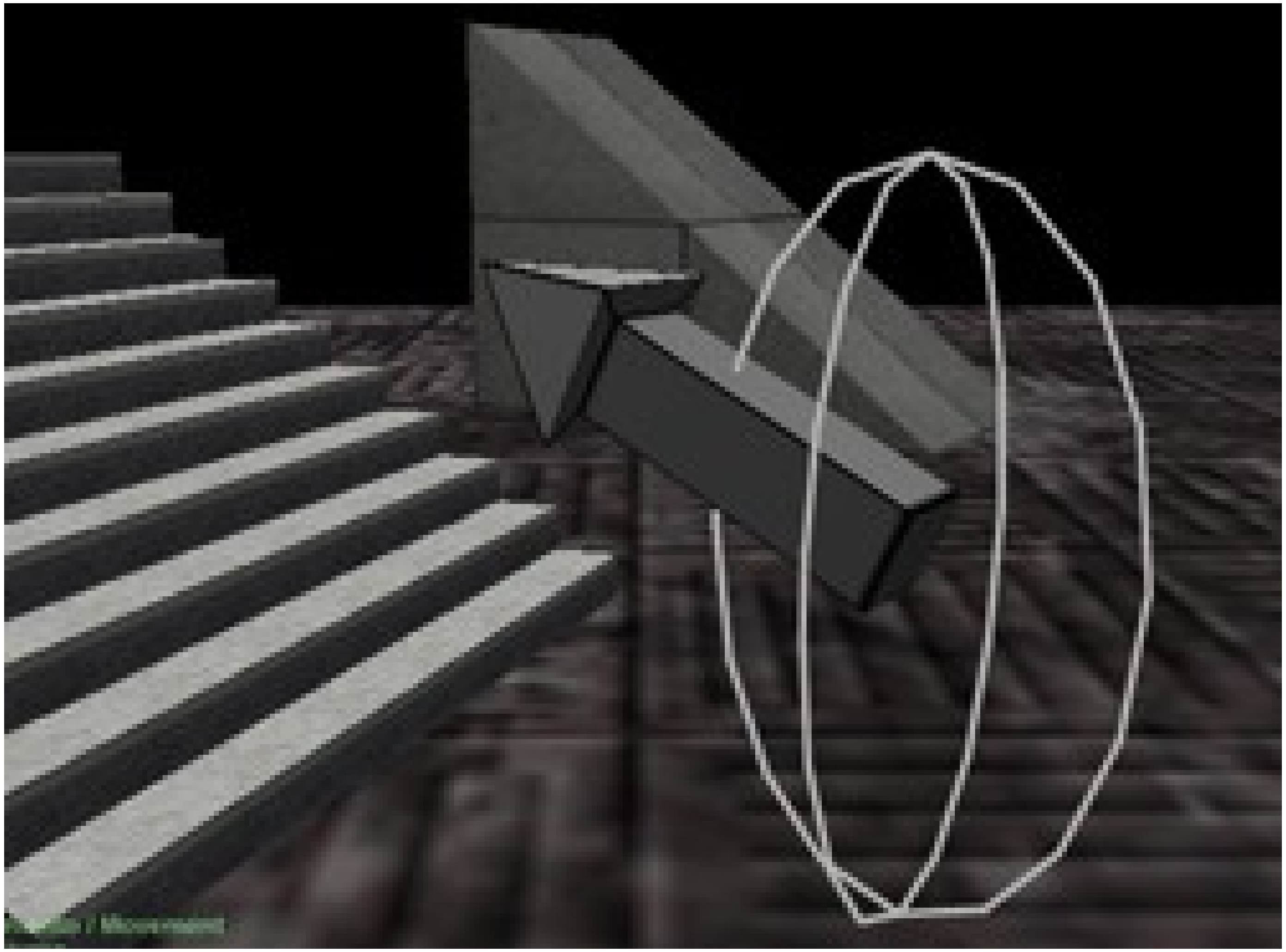


- Dynamics
- VR & games
- Robotics (Probabilistic Road Maps)
- CNC machining

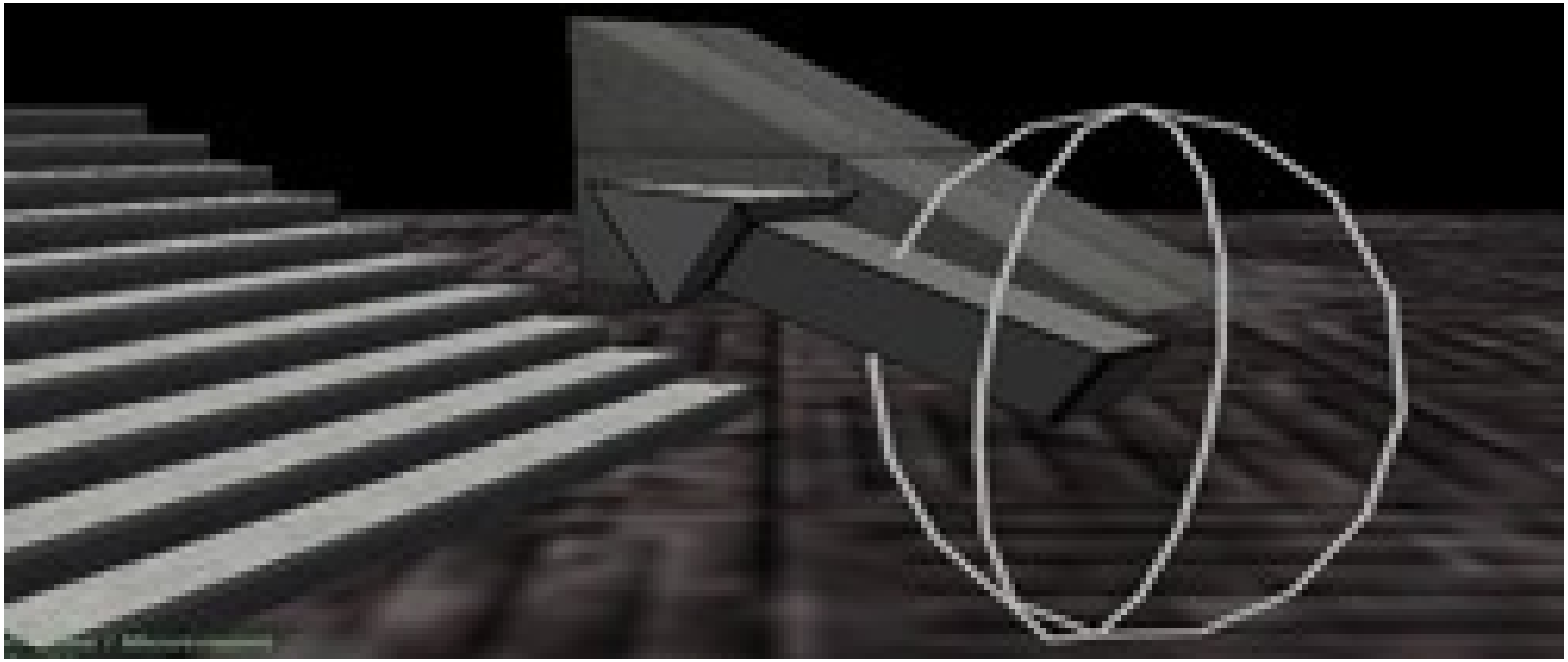




Simple Solution: Swept Ellipsoid



Simple Solution: Swept Ellipsoid



Simple Solution: Swept Ellipsoid

$$C(t) = p_0 + vt, \quad t \in [0, 1]$$

$$d(p) = N \cdot p + D$$

$$d(C(t)) = \pm 1$$

$$t = \frac{\pm 1 - N \cdot p_0 - D}{N \cdot v}$$

Problems for Swept Ellipsoid

- Non-ellipsoids
- Non-linear paths
- Rotations
- Colliding two ellipsoids
- Articulated models

Maths: Canny's Algorithm

- Configuration space: $(\mathbf{x}, \mathbf{Q}) : \mathbb{R}^7$
- Vertex: point **p**
- Edge: point **p** and unit vector **e**
- Face: normal vector **n** and distance to origin *d*

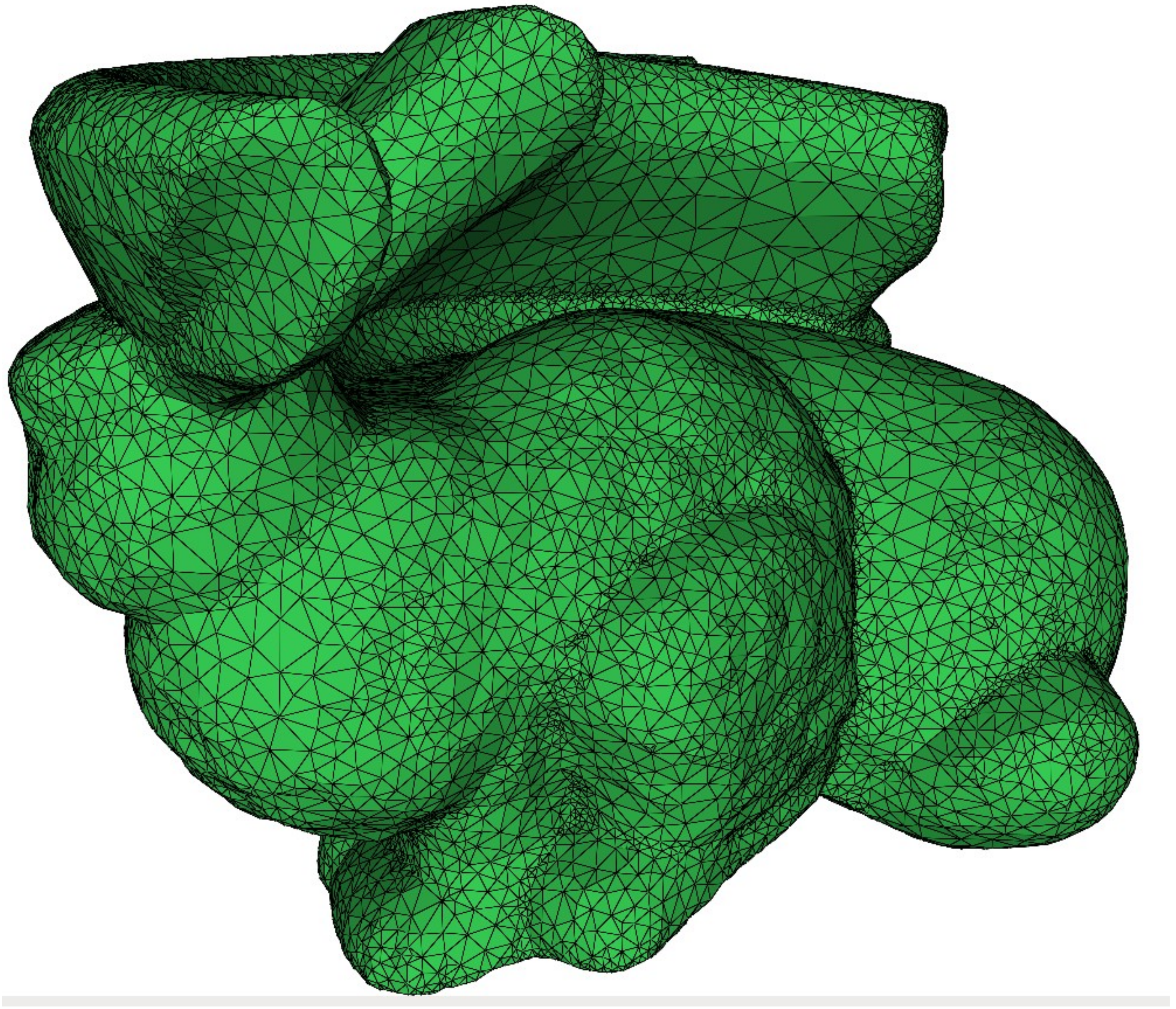
  F = (**n**,d)

Face-Vertex	$\mathbf{Q}\mathbf{F}_A\mathbf{Q}^*(1+\mathbf{x}) = 0$
Vertex-Face	$\mathbf{F}_B(1+\mathbf{Q}\mathbf{p}_A\mathbf{Q}^* + \mathbf{x}) = 0$
Edge-Edge	$(\mathbf{Q}\mathbf{p}_A\mathbf{Q}^* + \mathbf{x} - \mathbf{p}_B)\mathbf{Q}\mathbf{e}_A\mathbf{Q}^*\mathbf{e}_B = 0$

Complexity: $O(n^2 \log n)$

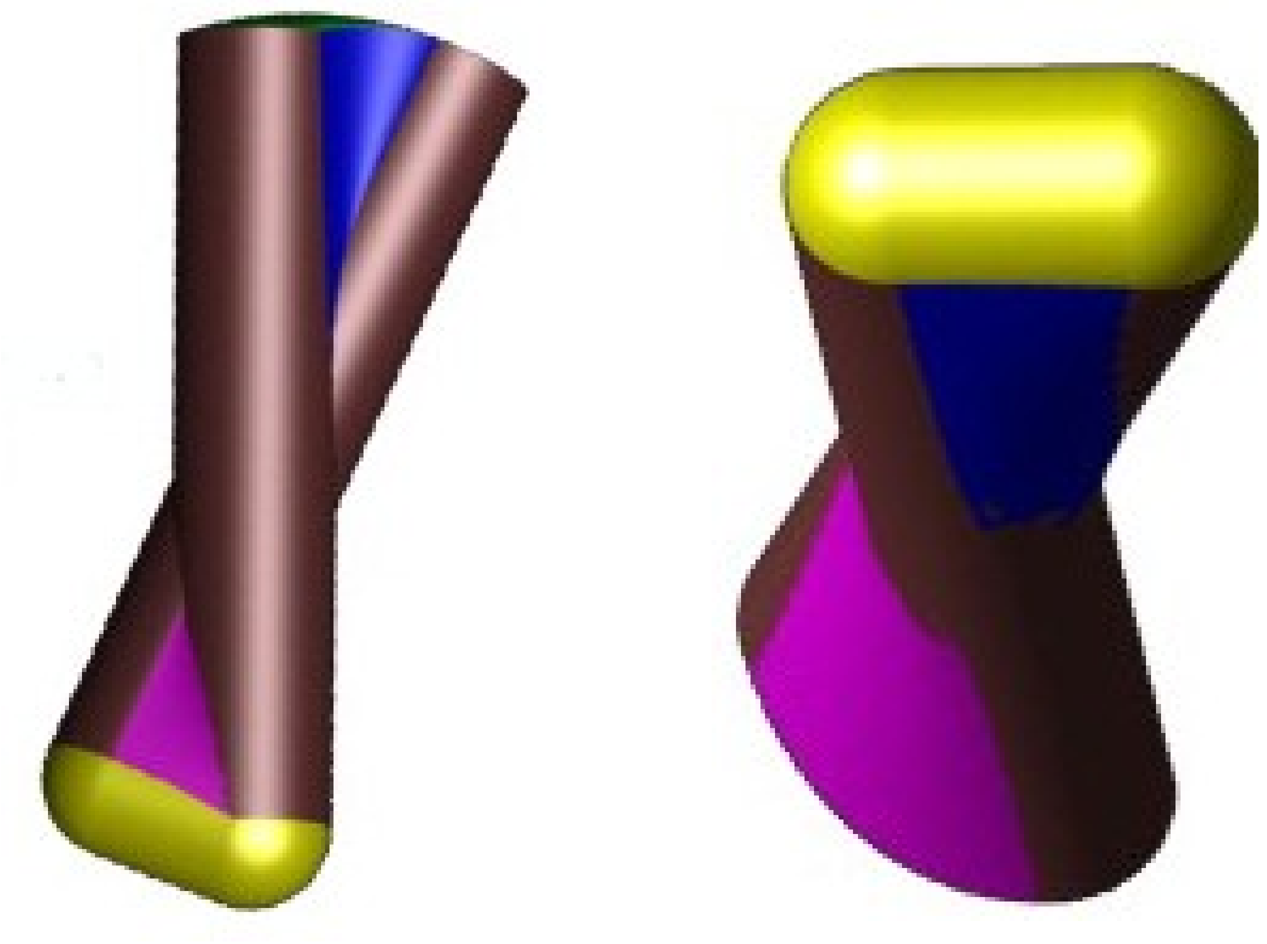
Intuitive Approach: Tessellation

- Voxels
  - Slow!



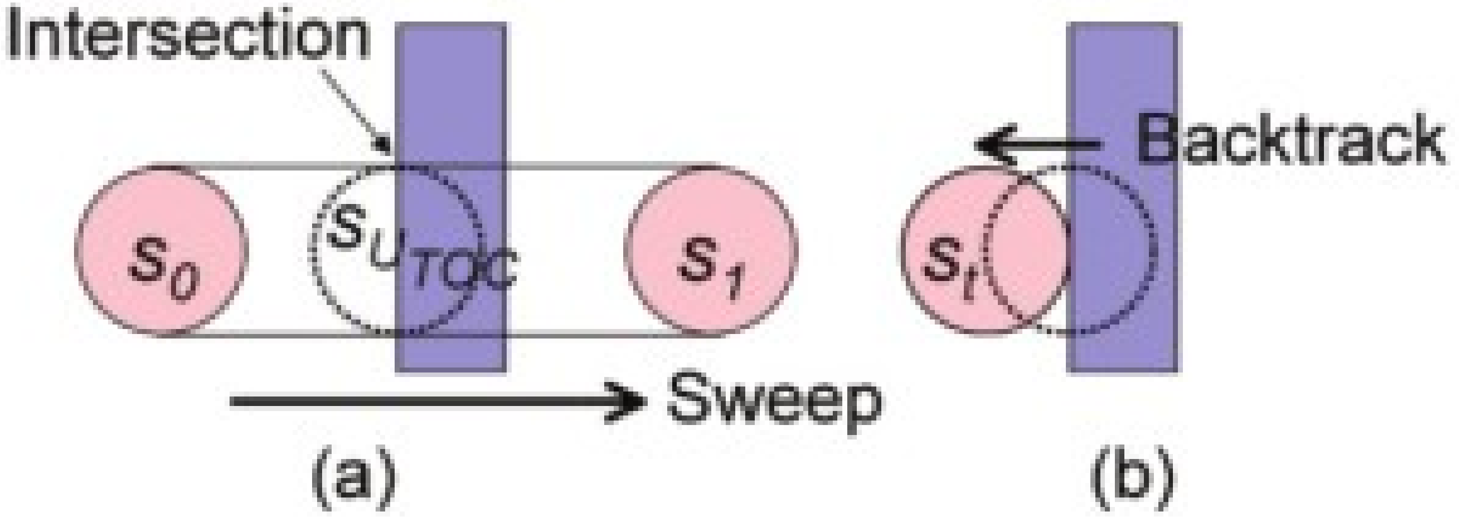
Intuitive Approach: Tessellation

- Swept "Line-Swept Sphere" (LSS)
  - Faster.



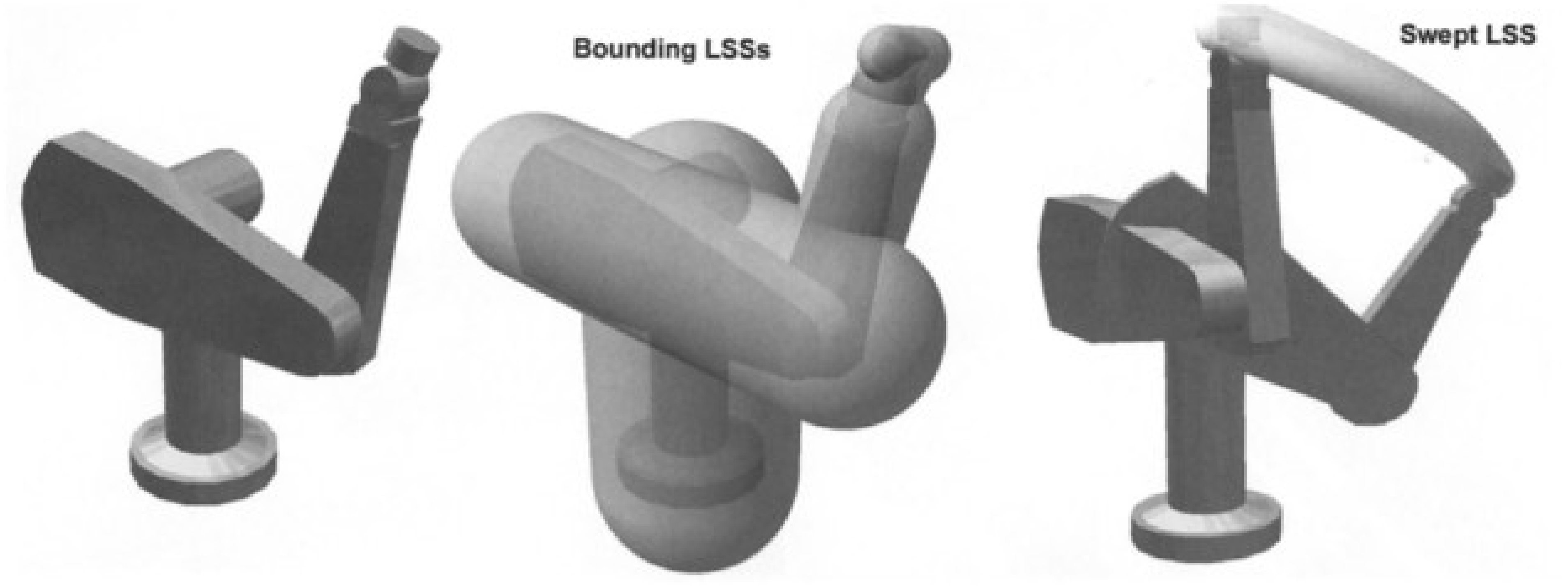
Intuitive Approach: Tessellation

- Backtracking finds time of 1st collision.
- Inexact!
- Fast, and handles articulated motion.



Intuitive Approach: Tessellation

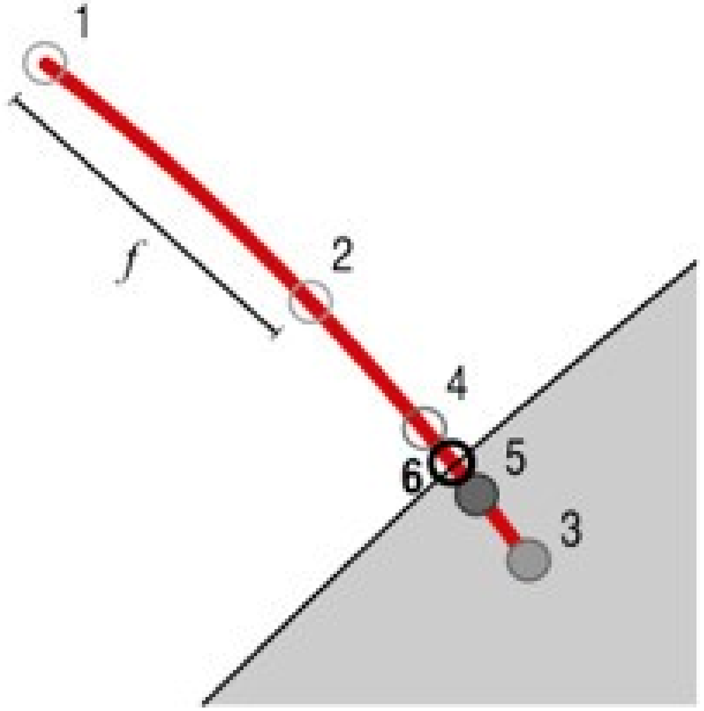
S. Redon et al. (2004). Interactive and Continuous Collision Detection for Avatars in Virtual Environments

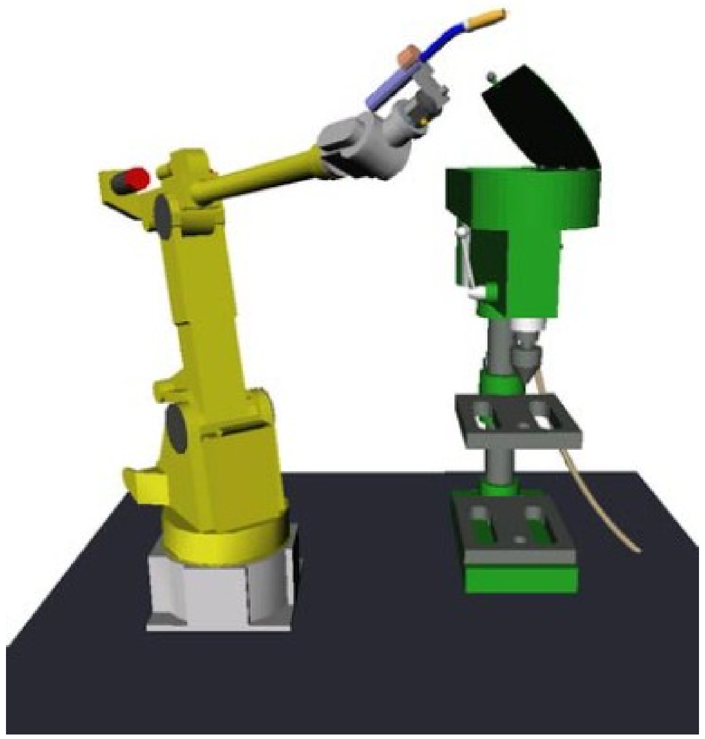


Adaptive Subdivision

$$\rho\, d(\mathbf{q}, \mathbf{q}')$$

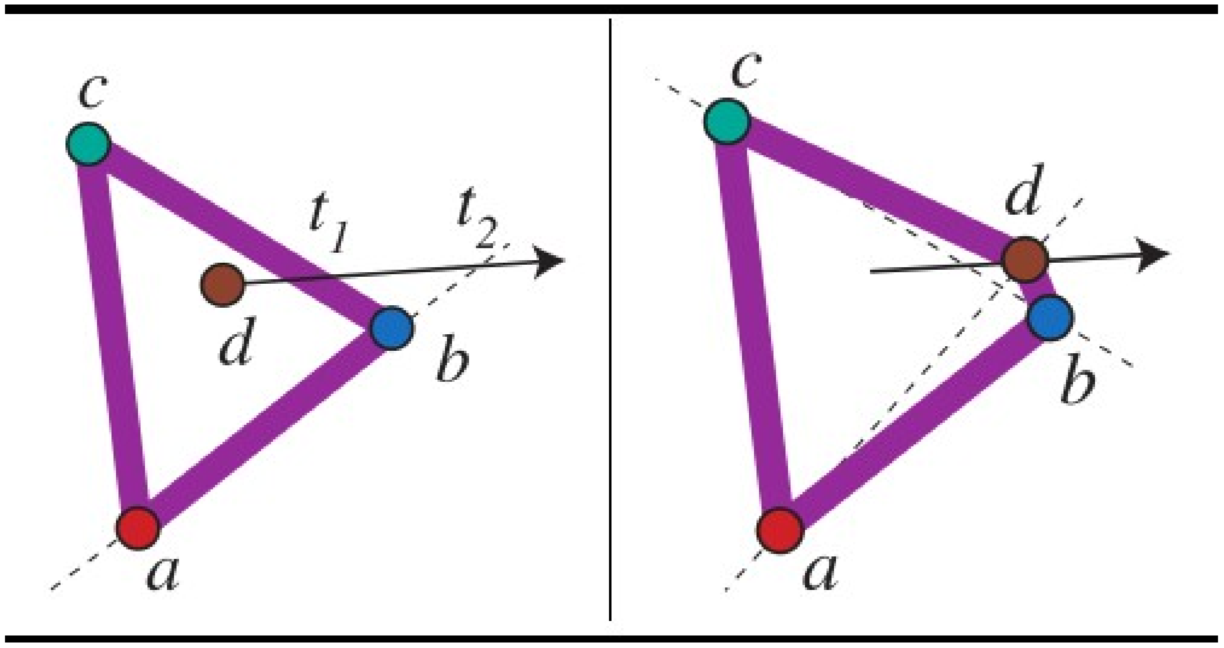
$$\rho\, d(\mathbf{q}, \mathbf{q}') < \eta(\mathbf{q}) + \eta(\mathbf{q}')$$



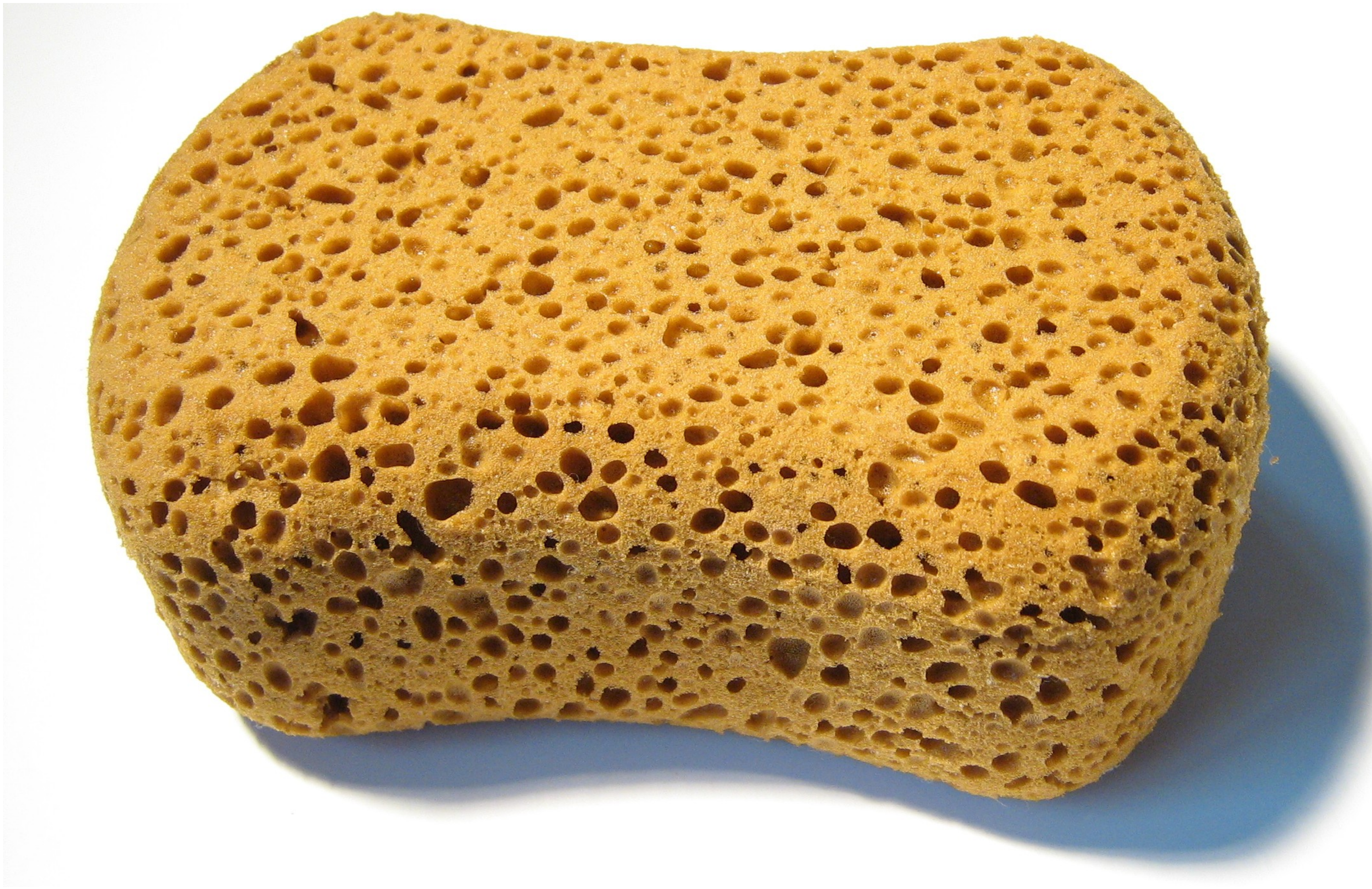


Kinetic Data Structures

Old proof	New proof
$\text{CCW}(a, b, c)$	$\text{CCW}(a, b, c)$
$\text{CCW}(d, b, c)$	$\text{CCW}(c, b, d)$
$\text{CCW}(d, c, a)$	$\text{CCW}(d, c, a)$
$\text{CCW}(d, a, b)$	$\text{CCW}(d, a, b)$

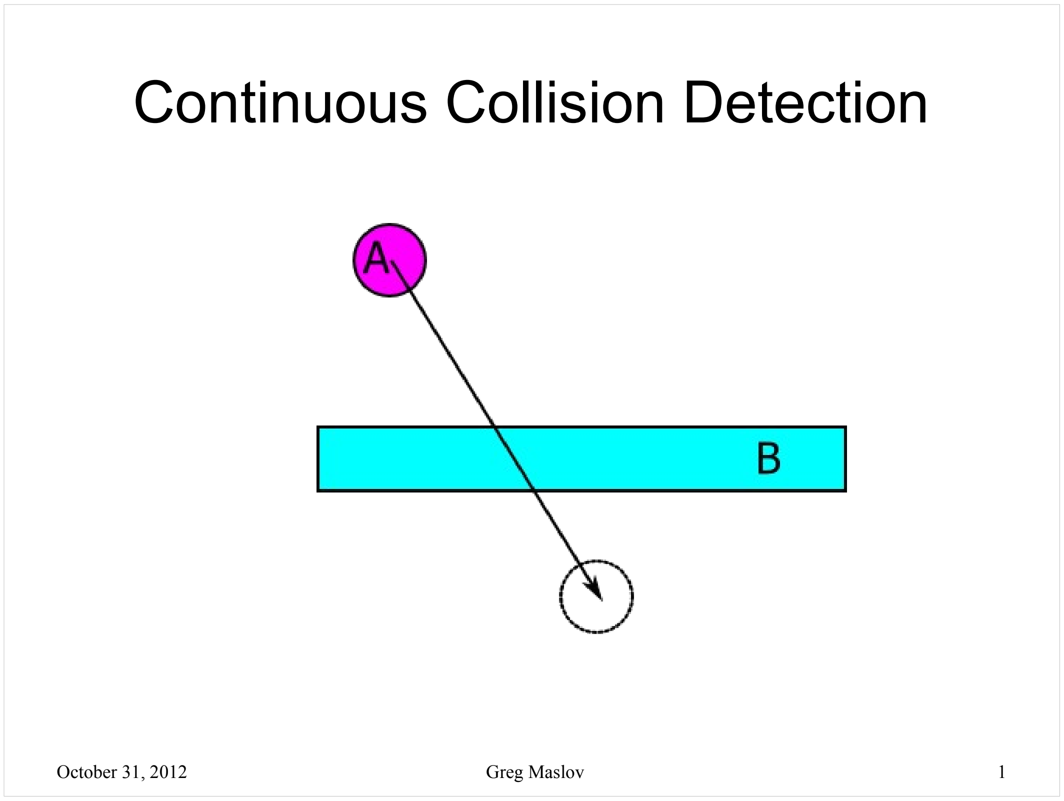


Deformable Models

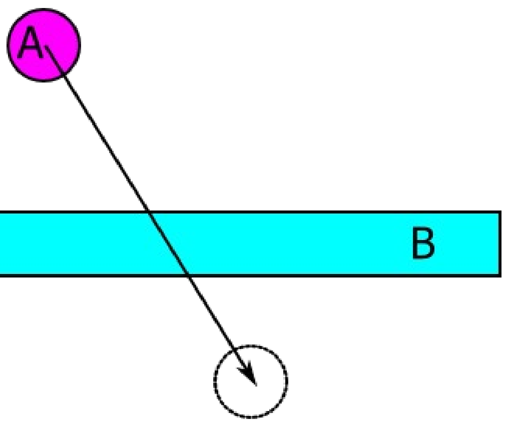


References

- **Swept Ellipsoids:** Fauerby, Kasper. (2003). Improved Collision detection and Response. Online: http://www.peroxide.dk/papers/collision/collision.pdf
- **Tessellation of Swept LSS + BVH Culling:** Redon, S., Kim, Y. J., Lin, M. C., Manocha, D., & Templeman, J. (2004). Interactive and Continuous Collision Detection for Avatars in Virtual Environments, 117. doi:10.1109/VR.2004.46
- **Exact Algebraic CCD:** Canny, J. (1986). Collision Detection for Moving Polyhedra. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-8(2), 200–209. doi:10.1109/TPAMI.1986.4767773
- **Adaptive Subdivision:** Schwarzer, F., Saha, M., & Latombe, J.-C. (2002). Exact Collision Checking Of Robot Paths. WAFR. http://ai.stanford.edu/~latombe/papers/wafr02/collision.pdf
- **Kinetic Data Structures:** Guibas, L. (2001). Kinetic Data Structures. In D. P. Mehta & S. Sahni (Eds.), Handbook of Data Structures and Applications (pp. 23–1--23–18). Chapman and Hall/CRC. http://graphics.stanford.edu/projects/lgl/papers/g-KDS_DS-Handbook-04/g-KDS_DS-Handbook-04.pdf



Continuous Collision Detection

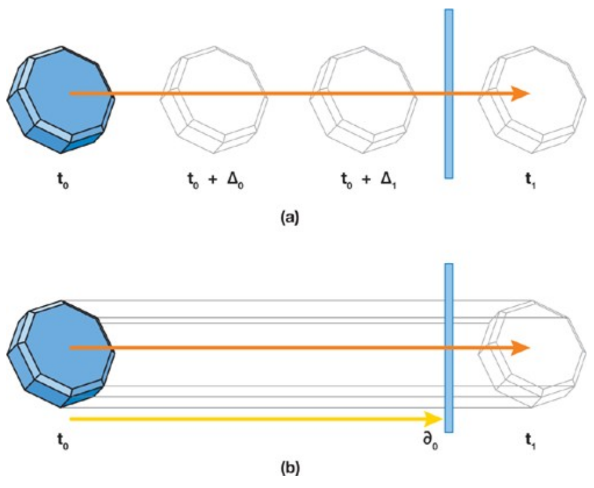


Hello, name, topic.

Please stop me at any time if I say something unclear or if you have a question.

Problem

- Interpenetration
- Missed collisions



In this class we've seen a number of efficient discrete collision detection methods. Most of them are based on some kind of culling, meaning finding a way to reduce the number of collision checks that need to be made between the primitives of two potentially colliding models: their triangles, or edges, planes, vertices, or whatever representation you have.

These work well; you have your rigid body dynamics simulator, it updates the position of every object at each timestep, then checks to see if any have smashed into each other and need to be pulled apart and fixed.

If that sounds really messy and error-prone, it is. Resolution of interpenetrations is a *nontrivial problem*.

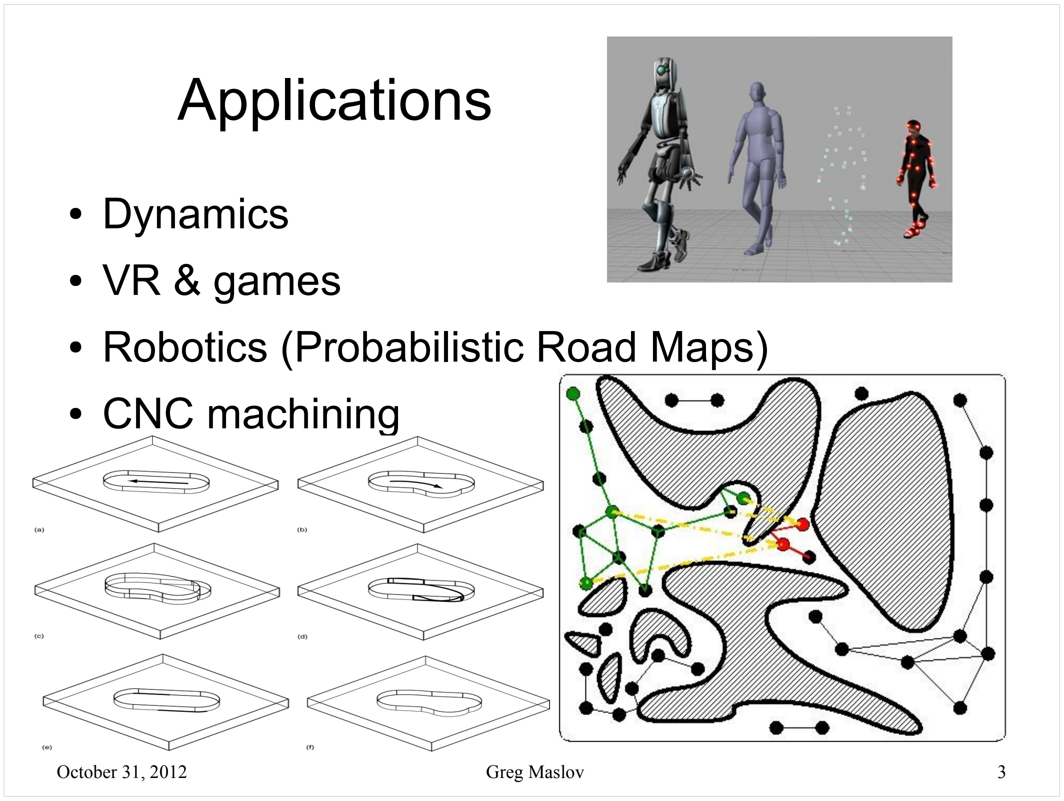
Furthermore, you can have another problem when a collision is missed entirely because of the discrete nature of the time step.

These problems crop up very often when simulating fast-moving objects, or very thin objects. If you're trying to model a bullet colliding with a piece of paper, then you're REALLY in trouble.

It would be better if you could find the exact point and time when the objects WOULD have collided, if you had been simulating every infinitesimal point in between.

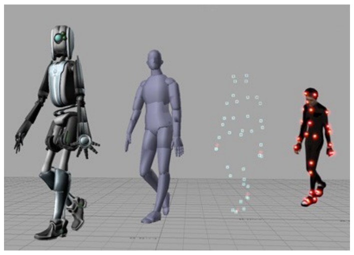
Well, you can do a *little* better with discrete collision detection. Just decrease the time step! In the limit as dt goes to zero, you're fine! Unfortunately, that solution doesn't always result in good performance, and it still provides no guarantee that you haven't missed any collisions.

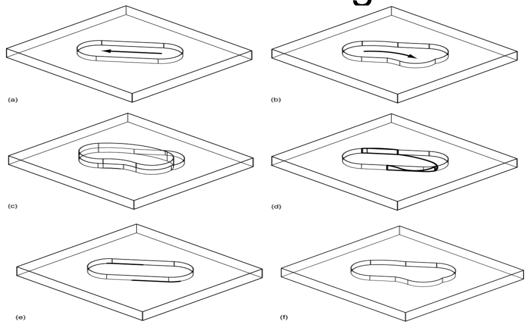
That's where continuous collision detection comes in. :-)

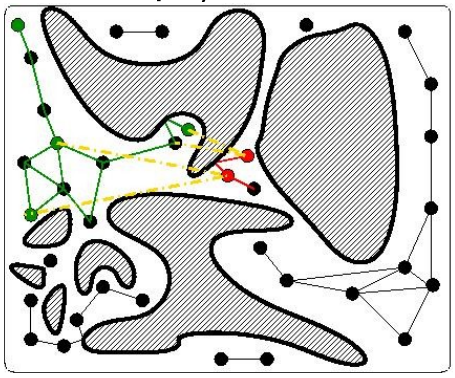


Applications

- Dynamics
- VR & games
- Robotics (Probabilistic Road Maps)
- CNC machining







October 31, 2012 Greg Maslov 3

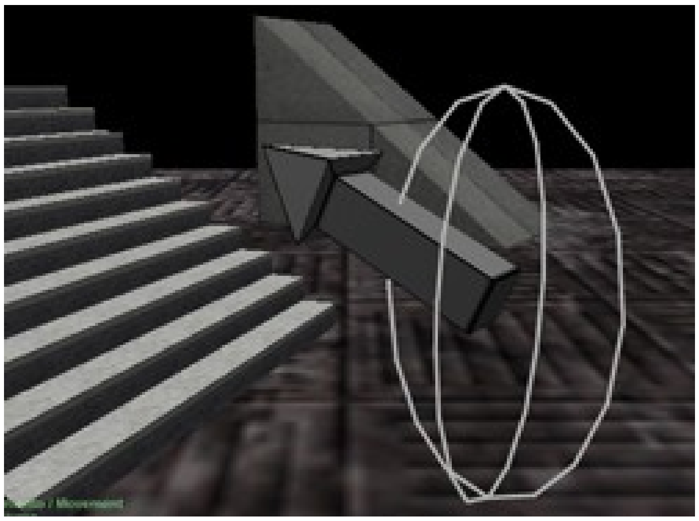
So where is continuous collision detection useful? Where does it provide an advantage over discrete collision detection?

First is dynamics; that's the bullet example. Virtual reality and video games also benefit from having precise collision information. In VR systems particularly, the application is to interpolate between two configurations of someone wearing a motion-capture suit; here your simulation might get position updates only a few times per second, so decreasing the time step is simply not an option.

Another area is robotics. I know a few of us here are in Dr. Alterovitz's robotics class this semester, so you may have heard of the Probabilistic Road Maps algorithm. This is a motion planning algorithm that requires a particular primitive query, LINK, which asks if there is a clear path for the robot to move from one configuration to another. These configurations may be quite far apart in terms of where the tip of your robot arm is, so you can't just check both endpoints; you have to look at all points in between.

Finally we have CNC machining, which is a nice area to work in because all your computation of toolpaths is done offline, ahead of time. You can use as slow an algorithm as you can stand, and the more precision, the better.

Simple Solution: Swept Ellipsoid



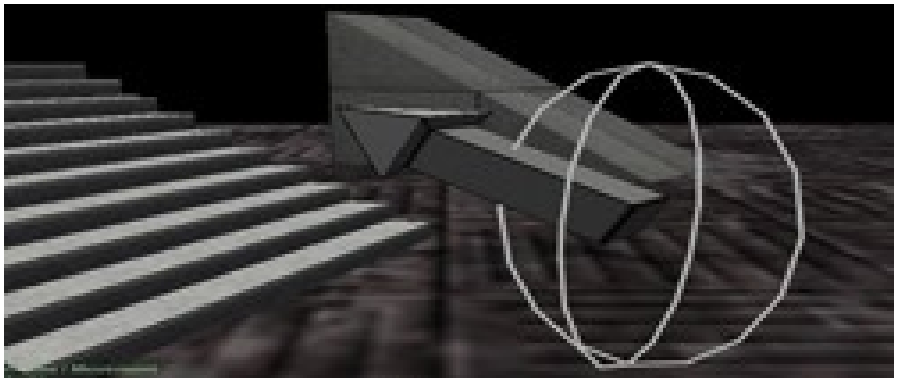
October 31, 2012 Greg Maslov 4

Let's start with something simple and concrete. I'm going to present a very simple example of the class of continuous collision detection algorithms called “algebraic methods”.

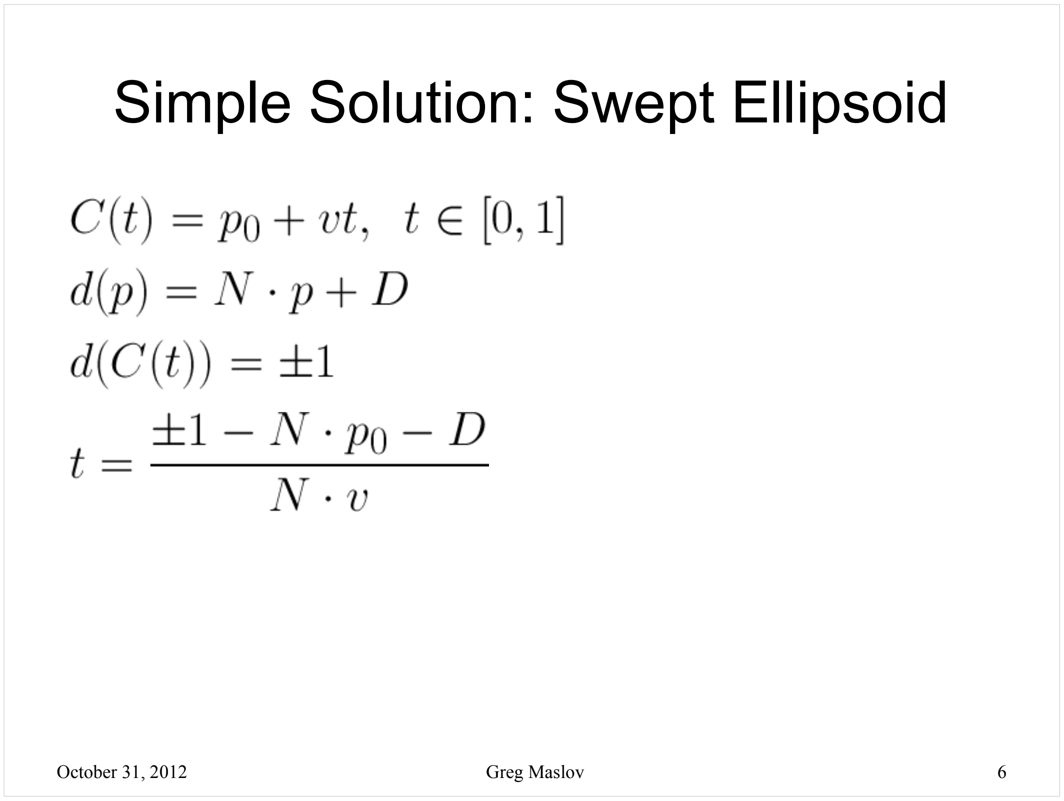
Consider an ellipsoid moving around in an environment composed of triangles. This is a good model for a lot of things. In fact most first-person shooter games use an ellipsoid to model the entire player character.

What we'd like to do is find the first triangle in the environment that this ellipsoid will hit as it moves along the velocity vector shown.

Simple Solution: Swept Ellipsoid



The first thing to note is that you can perform a simple linear transformation on the environment and on the ellipsoid to transform it into a unit sphere. This immediately makes things much simpler, reducing the problem to finding the time of collision between a swept unit sphere and a triangle. This is now simple enough to solve algebraically.



Simple Solution: Swept Ellipsoid

$$C(t) = p_0 + vt, \quad t \in [0, 1]$$
$$d(p) = N \cdot p + D$$
$$d(C(t)) = \pm 1$$
$$t = \frac{\pm 1 - N \cdot p_0 - D}{N \cdot v}$$

I won't go into the full algorithm here; like any collision detection algorithm, there are a lot of special cases and branching decision trees to go down.

For now I'll just derive one case, finding the point of collision between a sphere and the plane containing the triangle.

We parameterize the sphere's motion by tracking its center point over time. This is just linear interpolation from a starting position p_0 along the object's scaled velocity vector “v”.

The signed distance between a point and a plane is just the plane's normal, dot the point, plus the plane's constant.

And the time we're interested in, t, is when the signed distance between the plane and the unit sphere is plus or minus one.

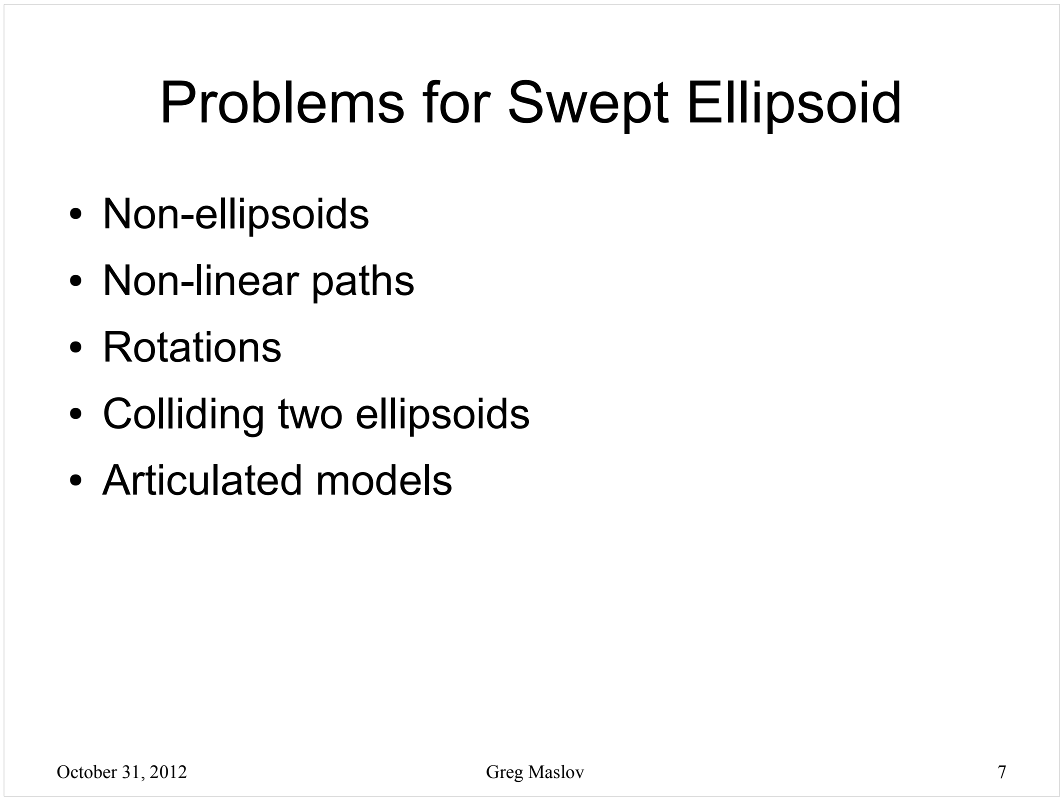
Solve it.

And if “t” is in the range zero to one, we have the time of collision and can easily find the exact point of collision.

Now even if the sphere doesn't collide this way, it may go on to hit an edge or a vertex later on in its path. Those collisions require solving a quadratic equation.

So you can see that this idea is pretty simple to implement and fast to execute.

So what are the problems?



Problems for Swept Ellipsoid

- Non-ellipsoids
- Non-linear paths
- Rotations
- Colliding two ellipsoids
- Articulated models

Well, obviously it only works for ellipsoids. Surprisingly not everything can be sufficiently well approximated by an ellipsoid.

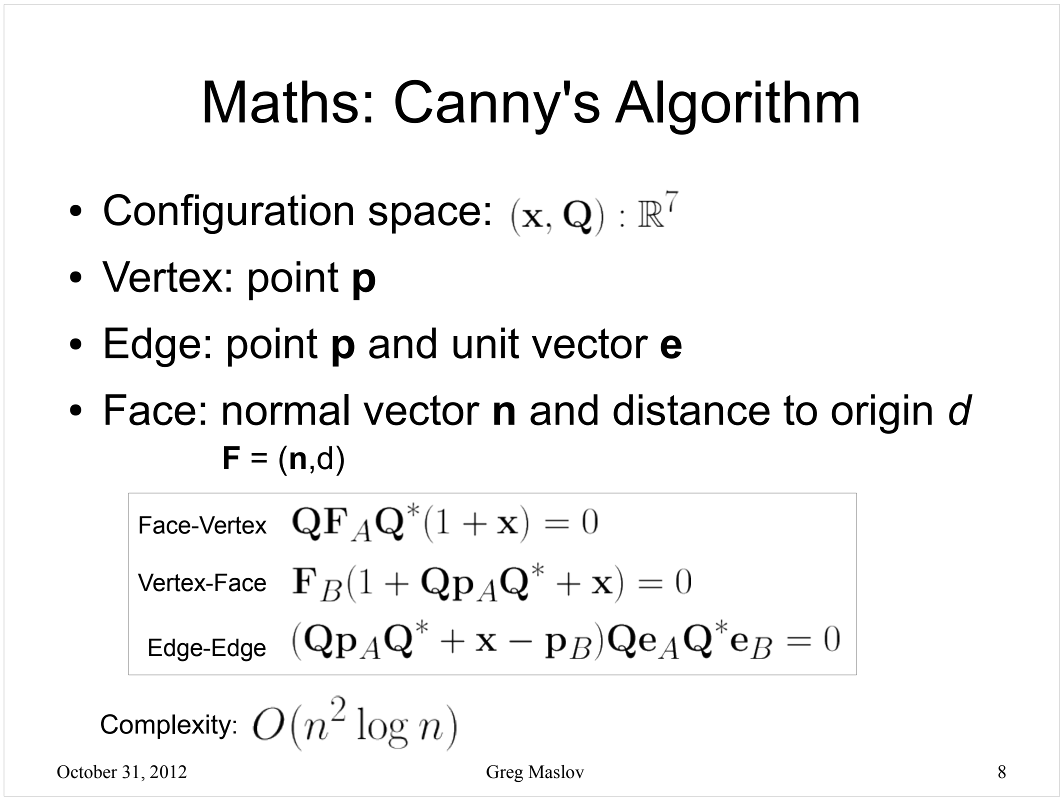
Second, we only consider linear paths. If your ellipsoid is doing something like, say, moving at the end of an articulated model <swing arm>, the motion may be a lot harder to parameterize, or result in difficult equations to solve.

Another point is that our linear transformation trick won't be applicable anymore if we allow the ellipsoid to rotate.

Analytically finding the intersection between a line segment and a rotating ellipsoid moving on a nonlinear path? Good luck!

And then we have the problem of finding the collision between two moving objects, not just between an object and a static environment.

Finally, if you have an articulated model like a robot arm or a manikin, it neatly combines all of the above problems into one big mess.



Maths: Canny's Algorithm

- Configuration space: $(\mathbf{x}, \mathbf{Q}) : \mathbb{R}^7$
- Vertex: point **p**
- Edge: point **p** and unit vector **e**
- Face: normal vector **n** and distance to origin *d*

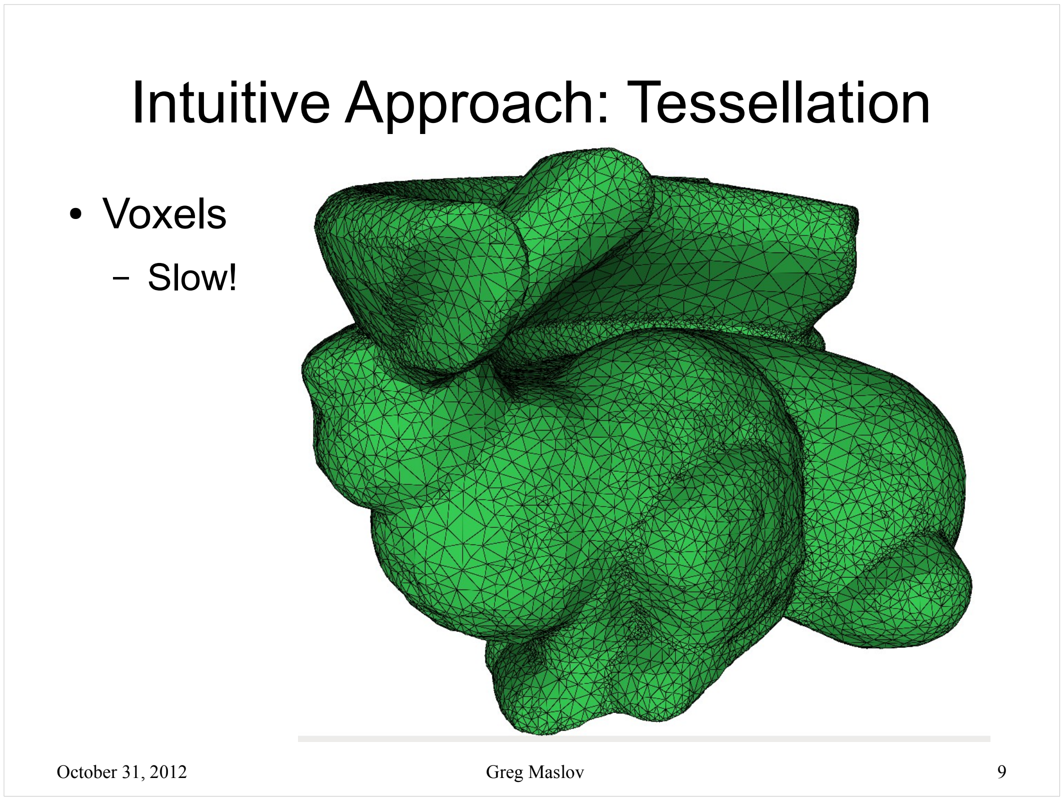
  F = (**n**,d)

Face-Vertex	$\mathbf{Q}\mathbf{F}_A\mathbf{Q}^*(1+\mathbf{x}) = 0$
Vertex-Face	$\mathbf{F}_B(1+\mathbf{Q}\mathbf{p}_A\mathbf{Q}^*+\mathbf{x}) = 0$
Edge-Edge	$(\mathbf{Q}\mathbf{p}_A\mathbf{Q}^*+\mathbf{x}-\mathbf{p}_B)\mathbf{Q}\mathbf{e}_A\mathbf{Q}^*\mathbf{e}_B = 0$

Complexity: $O(n^2 \log n)$

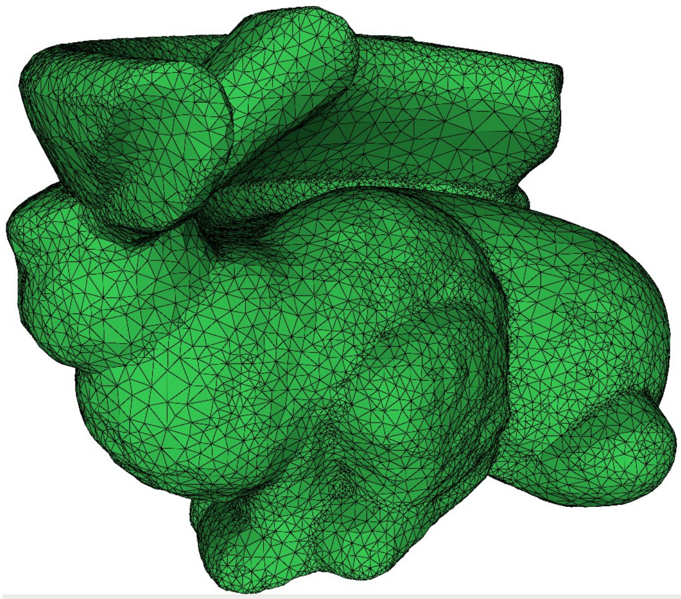
October 31, 2012 Greg Maslov 8

So let's briefly look at a more complicated algebraic algorithm for exact continuous rigid body collision detection.
This algorithm was developed in 1986 by John Canny, whose name you may recognize as being attached to the Canny edge detection algorithm.
It handles a lot of the problems mentioned on the previous slide: it allows arbitrary polyhedral geometry, rotations, and colliding two objects which are both moving.
Consider the configuration space of a rigid body. It has a position X and a rotation, represented by the quaternion Q, making a 7-dimensional vector in total.
Straight lines in this vector space correspond to arbitrary straight-line translations coupled with smoothly interpolated rotations. Actually, if you've heard of SLERP, Spherical Linear Interpolation, it depends on this fact: linear interpolation between two quaternions, representing rotations, corresponds to motion along the great circle arc connecting two points on a sphere.
So in this configuration space you can easily represent all of the features of a rigid body: its vertices, edges, and faces.
As it turns out, you can then work out these algebraic constraint equations that represent the three types of collisions that may occur. These multiplications by Q and Q-star are quaternion rotations, putting the vector into the reference frame of the rigid body.
Finally, and I'm glossing over a LOT of important details, you can parameterize these constraints over time, and find the time intervals of collision among all pairs of swept features in your two rigid bodies. This requires the solution of a cubic equation, when the motion is linear. The time intervals are then recursively merged to find the earliest time of collision.
The whole algorithm takes O(n^2 log n) time, where n is the number of features (vertices, edges, and faces) in both objects.
Unfortunately, although it's elegant and exact, it's too slow to work in real-time on models with a significant number of triangles. It could be used as the primitive collision test in a larger framework which performs some kind of culling.



Intuitive Approach: Tessellation

- Voxels
  - Slow!



October 31, 2012 Greg Maslov 9

What about a faster method, which can take advantage of all the existing highly optimized code out there for discrete collision detection?

Well, the first thing I thought when reading about Swept Volumes is, why not just SWEEP THE VOLUME, and see if it collides?

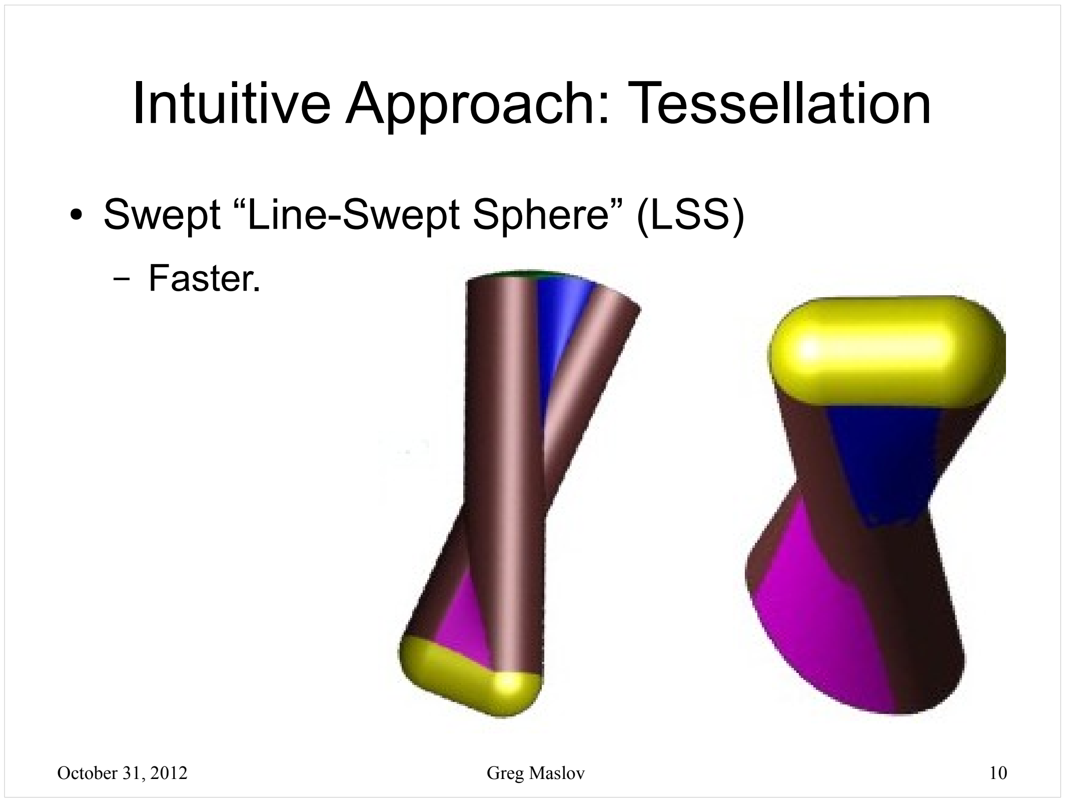
You can do that. The process is called, in general, tessellation.

Tesselation is computing an explicit polygonal mesh for the swept volume.

The picture here used a voxel-based method to compute the boundary of the volume that this Stanford bunny traces out as it moves.

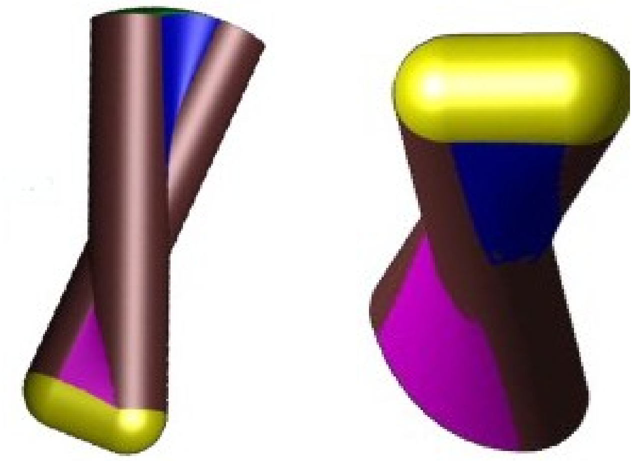
This can then be collided with the world using any old discrete collision detection algorithm.

Unfortunately, using voxels and extracting a boundary representation from them is very slow and uses a lot of RAM.

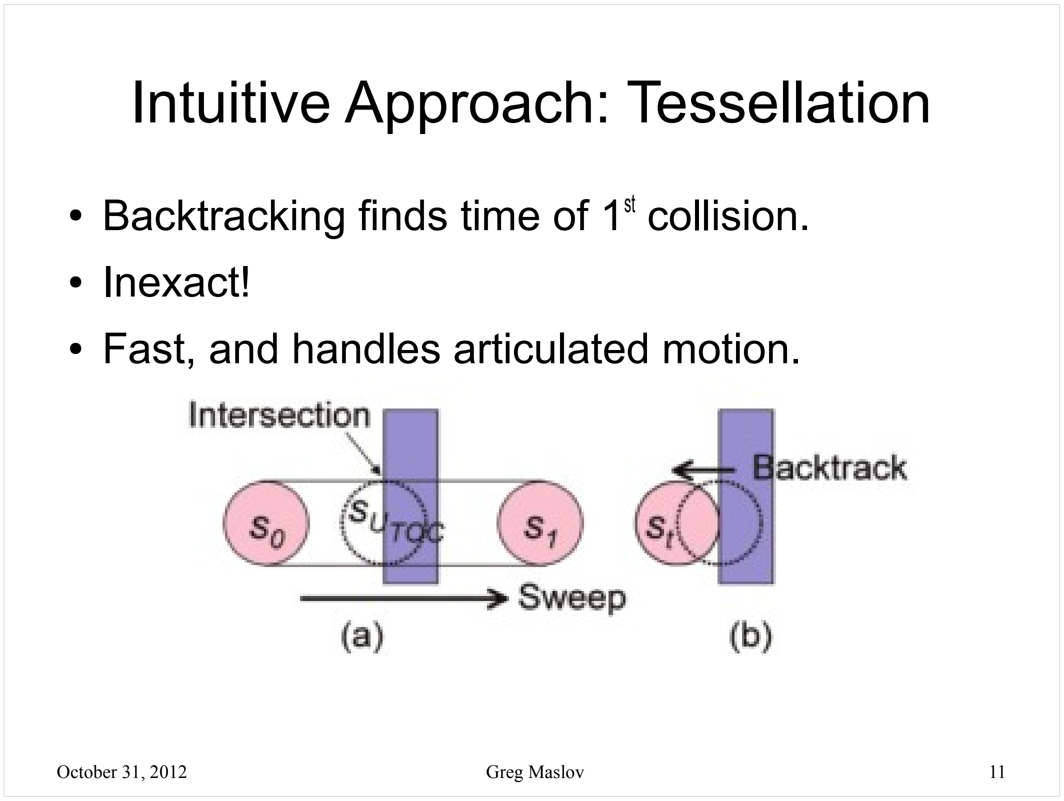


Intuitive Approach: Tessellation

- Swept "Line-Swept Sphere" (LSS)
  - Faster.

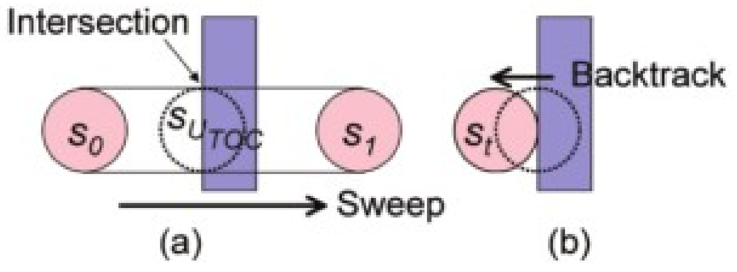


If your objects are simple enough, like ellipsoids or these Line-Swept Spheres, then computing their swept-volume tessellation can be made much faster, by taking advantage of an exact algebraic description of the swept surface, which is a little too involved for me to go into here.



Intuitive Approach: Tessellation

- Backtracking finds time of 1st collision.
- Inexact!
- Fast, and handles articulated motion.



October 31, 2012 Greg Maslov 11

Colliding the tessellation only tells you where in the interior the collision occurred, not when in time, and not even where the original object was in the sweep when it first started to collide.

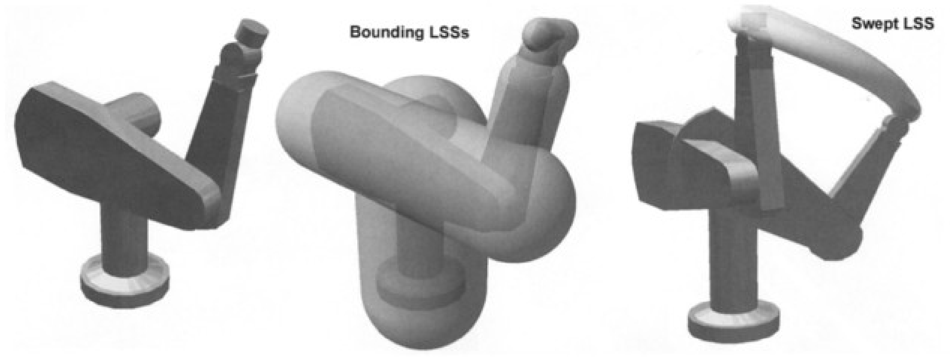
The discrete collision detection algorithm can only report the position of first collision inside the swept volume, going from time s_0 to time s_1. What you have to do is backtrack in time starting from that position, checking for a discrete collision of the object itself at each step.

Both the tesselation and the backtracking are necessarily inexact. This introduces errors in both space and time.

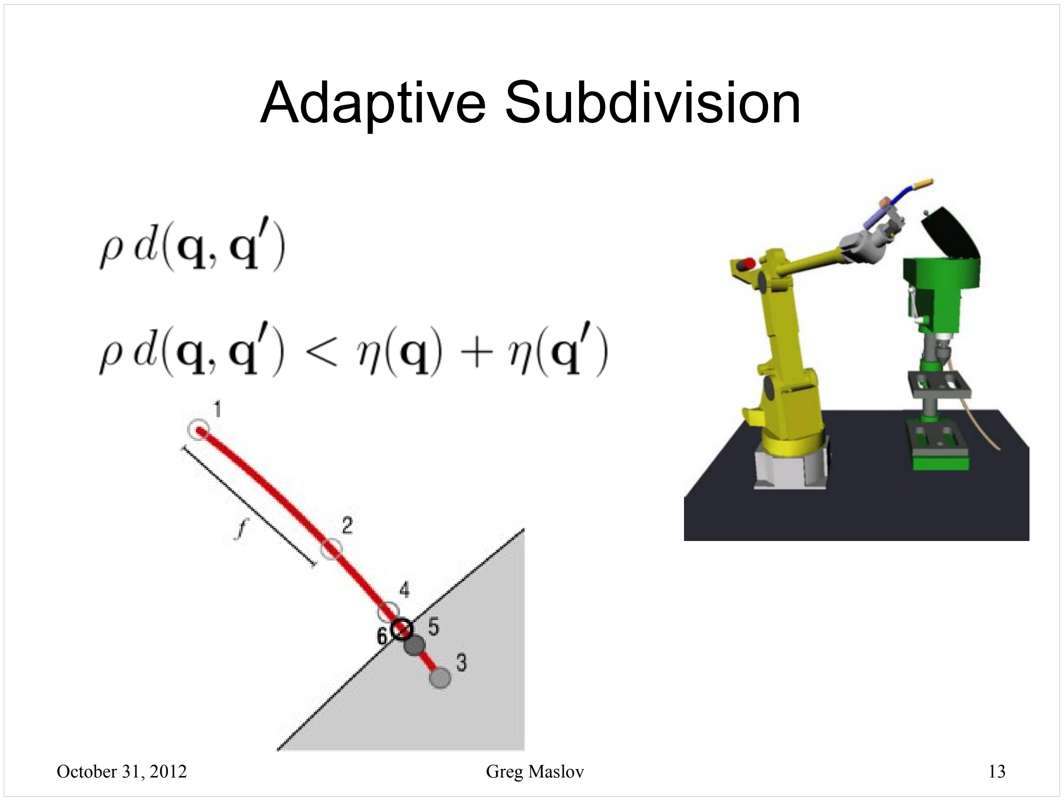
However, this approach can be fast enough for interactive applications, with a caveat. Note that the tessellated mesh of the swept volume will often contain a lot of triangles, a good deal more than the original model. This is expensive to collide, and since it's different at every time step, you can't precompute a BVH or similar structure to speed things up. You need a slightly more clever culling approach.

Intuitive Approach: Tessellation

S. Redon et al. (2004). Interactive and Continuous Collision Detection for Avatars in Virtual Environments



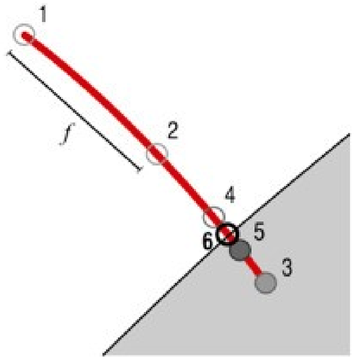
... which is what this paper describes.
Notice that an LSS is a nice shape for constructing the Bounding Volume Hierarchy (BVH) of a complex articulated model.
Here for example is a robot arm, the bounding LSS of each component, and one swept LSS during a particular motion.
You can easily imagine sweeping each LSS in turn, hierarchically, to efficiently find potential collisions between this model and a cluttered environment. Once you have a small set of potential collisions, you can then use some other CCD algorithm to locate the exact collision among the primitive triangles of the robot and the environment.
That general approach is the subject of the paper cited here. Layers and layers of culling is the name of the game, and at the bottom the primitive collision test is essentially an improved version of Canny's algorithm, which I described earlier.
The overall algorithm is still inexact due to the tessellation culling step, meaning it could produce false negatives, but it can be tuned to any desired tolerance.

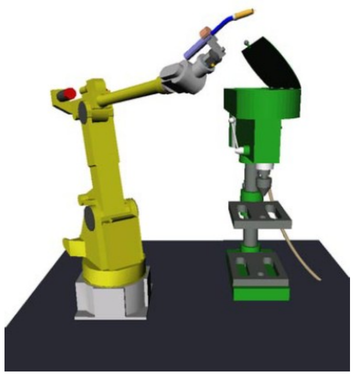


Adaptive Subdivision

$$\rho\, d(\mathbf{q}, \mathbf{q}')$$

$$\rho\, d(\mathbf{q}, \mathbf{q}') < \eta(\mathbf{q}) + \eta(\mathbf{q}')$$





There is another, simpler approach to the continuous collision detection problem, which originated from robotics.

The other algorithms I've described so far all work in Euclidean 3D space, where the location of all your objects and obstacles are known precisely. In robotics though, you're usually working rather in the N-dimensional Configuration Space of a robot, and this has the curious property that it's usually extremely difficult to determine the shape of an obstacle. All you can do is sample discrete points and see if they are or aren't within an obstacle.

Earlier on I mentioned the LINK query, and the Probabilistic Road Maps algorithm. The question here is to determine if, and where, the straight segment from point 1 to point 3 here collides with an obstacle. Note that these are obstacles in C-space.

The usual approach is to just sample a few points in between, and stop when you've had enough - that is, reached a certain tolerance for minimum width of an obstacle that you might miss.

A better approach would be to know exactly when to stop, so that you're GUARANTEED to have no collisions in between the points you've checked. That's adaptive subdivision.

An interesting fact is that it's possible to relate distance in configuration space to distance in the workspace. A single constant, rho, can be determined such that, when considering the straight-line motion between two configurations q and q', no point on the robot moves farther in the workspace than rho times the distance in configuration space.

What does that mean in practice? A Cartesian robot, which consists of a single rigid body moving along the x,y,z axes without rotating, would have a rho of 1: distance in the configuration space is equivalent to distance in the workspace. A robot with a manipulator arm would have a large rho, because if the arm is stretched out and you rotate the base even a little bit, you'll produce a huge motion down at the tip.

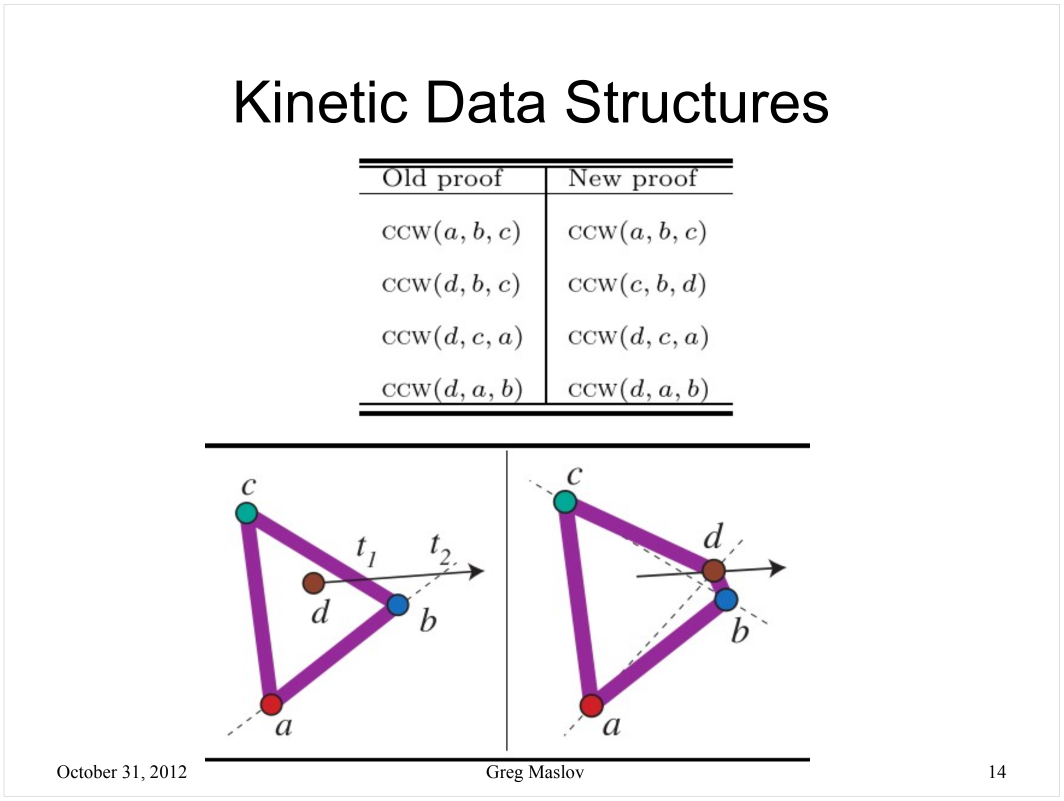
Now assume that you have a function eta, which gives the closest Euclidean distance that the robot approaches any obstacles in the workspace when it's in a given configuration q. If the robot is just touching an obstacle, eta will be zero.

Now you can prove that if this inequality holds, then there CANNOT be any collisions between configurations q and q'.

Proof: Assume there is a configuration on the line between q and q', where the robot touches an obstacle, and that the above inequality does hold. Some point on the robot would have to move a distance of at least eta(q) to reach the obstacle, then at least eta(q') to reach its position at q'. But the earlier result was that NO point on the robot can move farther than rho-d-q-q'.

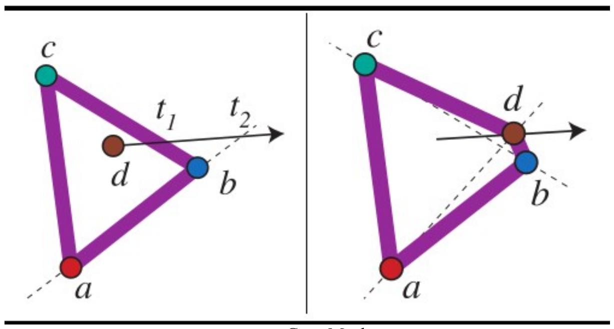
Now you don't have to check all of the infinitely many configurations between your first and second position for collisions. You just have to recursively subdivide and check the midpoints, and this inequality will tell you when it's guaranteed to be safe to stop.

You can read the paper for details on how to efficiently calculate rho and eta ;-)



Kinetic Data Structures

Old proof	New proof
$\mathrm{ccw}(a,b,c)$	$\mathrm{ccw}(a,b,c)$
$\mathrm{ccw}(d,b,c)$	$\mathrm{ccw}(c,b,d)$
$\mathrm{ccw}(d,c,a)$	$\mathrm{ccw}(d,c,a)$
$\mathrm{ccw}(d,a,b)$	$\mathrm{ccw}(d,a,b)$



Kinetic data structures are a more general theory that can be applied to collision detection in particular.

The basic idea is that the motion of objects within a system can be predicted well within a short time horizon, but occasionally discrete events occur to upset things. Such events may be internal to the system, or external user input.

Consider first the problem of keeping the convex hull of a set of points updated as they move around in the plane. We can calculate the convex hull of a,b,c,d here, but would rather not re-calculate it at every time step. What we do is choose a set of so-called *certificates* -- predicates that, if they are all true, guarantee that the convex hull of a,b,c,d is still a,b,c.

The converse does NOT have to be true, in general. You could choose an overly-sensitive certificate that fails without the convex hull being invalid; it'll only result in a less efficient algorithm.

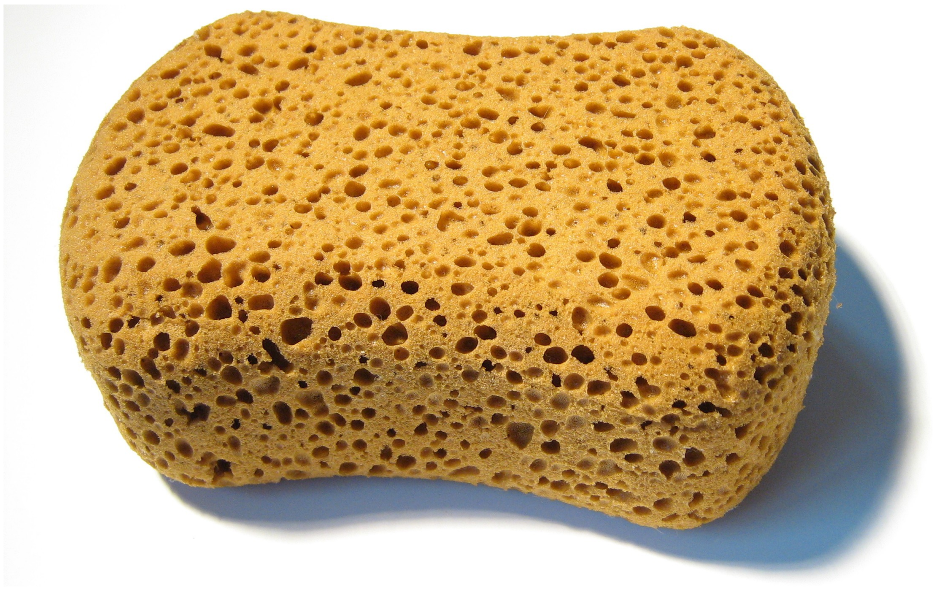
Now if d moves outside of the triangle, you can see that at least one of these certificates must fail. Then the convex hull has to be updated and new certificates chosen.

Note that if we know something about how the points are moving, it's also possible to predict exactly when any one of the certificates will fail.

The task of the algorithm designer working with kinetic data structures is to choose a good set of certificates, which guarantee the structure's validity while being small enough to check efficiently.

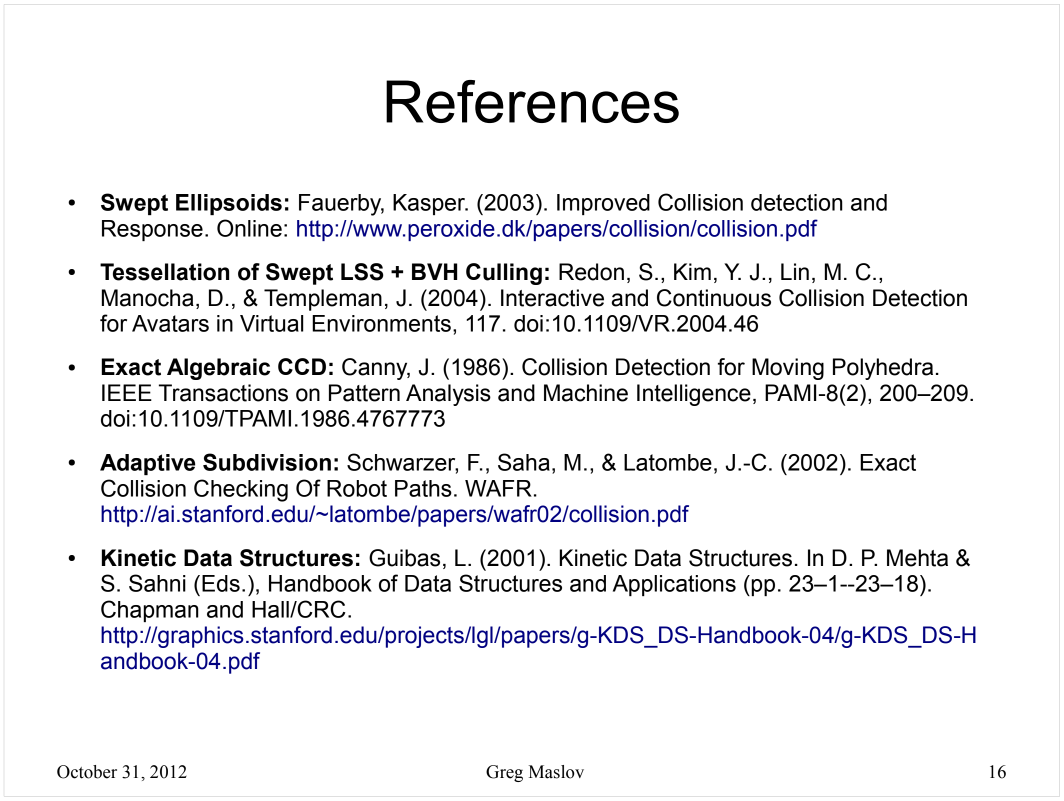
Hopefully you can now see how the collision detection problem can be attacked in this way; instead of the predicate “the convex hull of a,b,c,d is a,b,c”, we might have the predicate “a,b,c does not collide with d,e,f”; and a well-chosen set of certificates.

Deformable Models



And now we come to the topic of deformable models. The algorithm which I've just described makes use of a good amount of preprocessing and tree-building, all of which becomes rather moot when you have to recompute it every time step due to your model changing shape.

Fortunately, I believe the next speaker today will cover exactly these issues: Deformable Models.



References

- **Swept Ellipsoids:** Fauerby, Kasper. (2003). Improved Collision detection and Response. Online: http://www.peroxide.dk/papers/collision/collision.pdf
- **Tessellation of Swept LSS + BVH Culling:** Redon, S., Kim, Y. J., Lin, M. C., Manocha, D., & Templeman, J. (2004). Interactive and Continuous Collision Detection for Avatars in Virtual Environments, 117. doi:10.1109/VR.2004.46
- **Exact Algebraic CCD:** Canny, J. (1986). Collision Detection for Moving Polyhedra. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-8(2), 200–209. doi:10.1109/TPAMI.1986.4767773
- **Adaptive Subdivision:** Schwarzer, F., Saha, M., & Latombe, J.-C. (2002). Exact Collision Checking Of Robot Paths. WAFR. http://ai.stanford.edu/~latombe/papers/wafr02/collision.pdf
- **Kinetic Data Structures:** Guibas, L. (2001). Kinetic Data Structures. In D. P. Mehta & S. Sahni (Eds.), Handbook of Data Structures and Applications (pp. 23–1--23–18). Chapman and Hall/CRC. http://graphics.stanford.edu/projects/lgl/papers/g-KDS_DS-Handbook-04/g-KDS_DS-Handbook-04.pdf

Hopefully you've now seen a good overview of the different algorithms that are available for continuous collision detection.

If you're interested in more details, you should look at one of these papers. These slides should be made available on the course website later today.