Robust Contact Creation for Physics Simulations

Dirk Gregorius – Valve Software

Good morning everybody! Thank you for attending my talk today!
My name is Dirk Gregorius and I am a software engineer at Valve.
My talk today will be about robust contact creation for physics simulations.

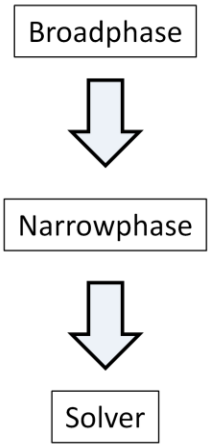
Outline

- **Introduction**
- Basic Strategies
- Solving the Collision Table
- Contact Optimizations
- Code Examples

Let's start with a quick outline of this talk:

- After a short introduction we will define what contact points and contact manifolds are and how to use them to model contact in a physics engine
- We will then define some basic shapes commonly used in physics engines and how to compute robust and stable contact manifolds between them. This will build the major part of this talk!
- The method I will show today can produce many contacts points per frame and we might **not** want to send all these contact points to the solver due to performance reasons.
- The ultimate goal is a fast, stable and plausible simulation and I will show you how to efficiently approximate any number of contact points with a stable manifold.

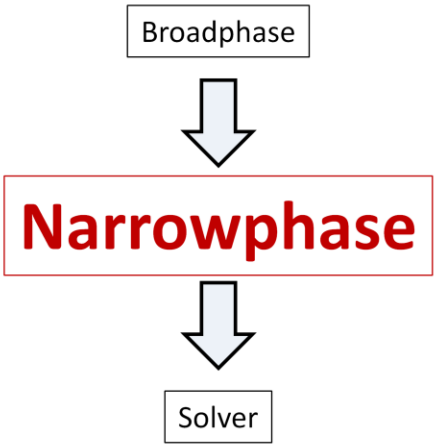
Anatomy of a Physics Tick



Let's quickly look at the anatomy of a physics tick:

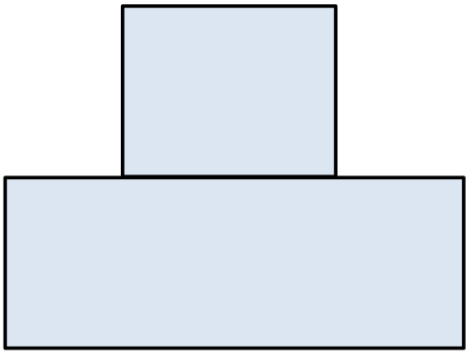
- We use the broadphase to detect pairs of shape proxies that can potentially be in contact
- E.g. in an AABB tree we detect all overlapping AABB pairs
- In the narrowphase we then need to test if the actual collision shapes are touching
- If the shapes are touching we create contact information between the two shapes and send this to the solver
- Finally the solver then advances the rigid bodies and uses the provided contact information to prevent penetration and to simulate friction

Anatomy of a Physics Tick



The topic of this talk today is a complete breakdown of the narrowphase:
- We will cover how to detect whether the two internal shapes of two overlapping proxies are actually touching and how to create the contact information between them that we pass to the solver!

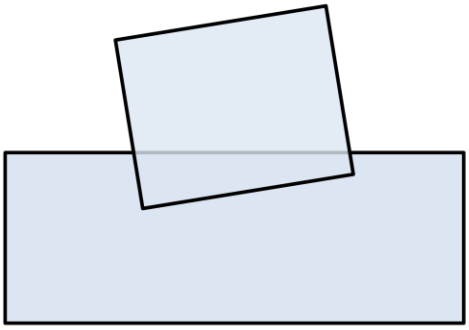
Two Shapes in Touching Contact



Let's get started! What does contact actually mean?

- Obviously contact occurs when two shapes are touching!
- Note that the picture above shows a more or less ideal contact situation

Two Shapes in Overlapping Contact

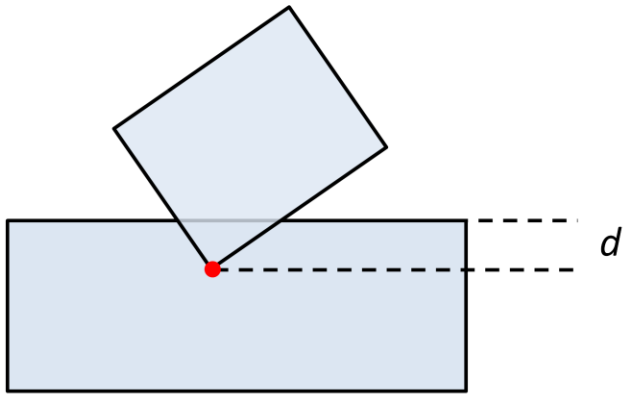


In reality we will most likely deal with overlap and also need to handle penetration!

- It is actual crucial for a decent physics engine that it can handle penetration efficiently
- Ideally there should be no performance penalty in the overlapping case and the rather ideal touching configuration is just a special case with zero penetration

Contact Points

```
struct ContactPoint
{
    Vector3 Position;
    float Penetration;
};
```

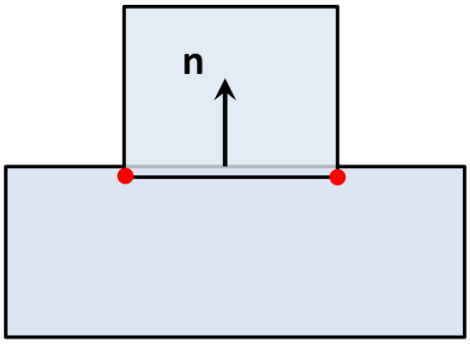


We start with some basic definitions. A contact point is defined by:

- A position (indicated by the red dot)
- And a penetration depth (d)

Contact Manifolds

```
struct ContactManifold
{
   int PointCount;
   ContactPoint Points[ 4 ];
   Vector3 Normal;
};
```

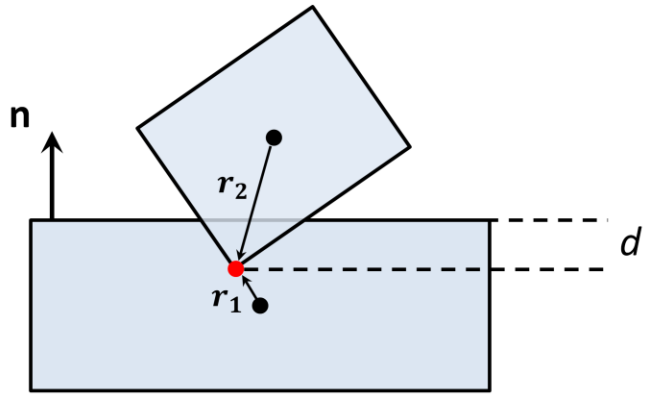


Now we can define a contact manifold simply as a set of contact points that share a **common** normal!

- Note that I assume here a maximum of four contact points in a manifold
- This is an optimization since four points are usually enough for fast, stable and robust contact simulation
- I will show you at the end how we can efficiently reduce larger sets of contact points down to a maximum of four
- Please don't get miss-leaded by the picture here! In 2D two contact points would be indeed sufficient, but in 3D we need at least four contact points!

The solver usually expects the normal to have a specific orientation. E.g. from A to B or from B to A. So you need to make sure to create a consistent orientation when building the manifold!

Recap: Solving Contact (1)

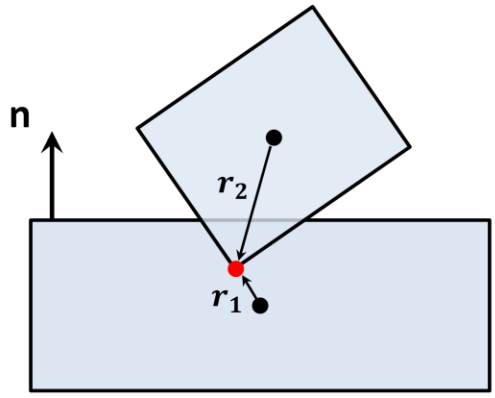


Let's now quickly recap how we solve contact points in a physics engine to bring us all on some common ground here:

- We need to handle contact to prevent penetration and to simulate friction
- In a game physics engine we usually simply solve each contact point individually using some iterative approach (e.g. Sequential Impulse or Projected Gauss-Seidel)

So what do we do with the contact information?

Recap: Solving Contact (2)

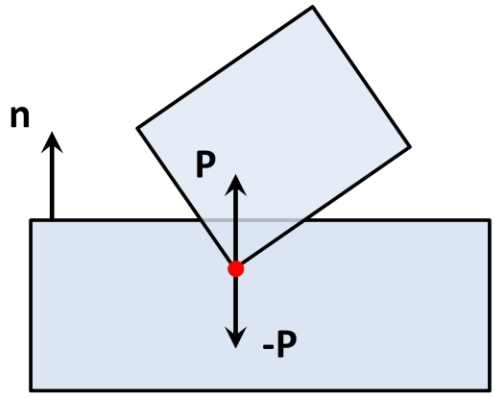


$$v_{rel} = (\boldsymbol{v_2} + \boldsymbol{\omega_2} \times \boldsymbol{r_2} - \boldsymbol{v_1} - \boldsymbol{\omega_2} \times \boldsymbol{r_2}) \cdot \boldsymbol{n} = \boldsymbol{0}$$

First we compute the relative velocity at the contact point in the direction of the normal:

- A negative relative velocity means that the two bodies are penetrating
- A positive relative velocity means that the bodies are separating

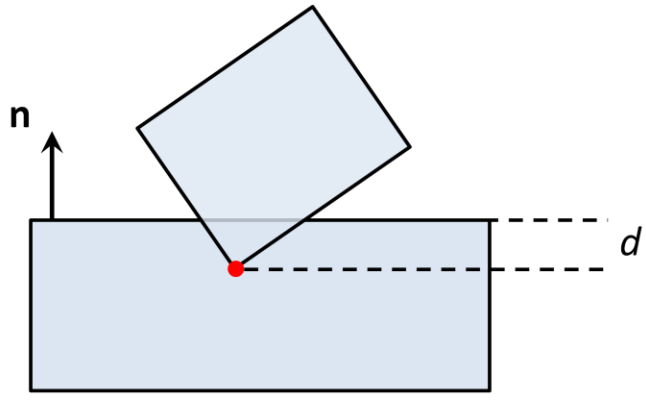
Recap: Solving Contact (3)



Next we apply equal and opposite impulses in the direction of the normal to drive a negative relative velocity to zero:

- Obviously our ultimate goal is to resolve all negative (penetrating) velocities as this will prevent further penetration

Recap: Solving Contact (4)



$$v_{rel} = -\frac{0.1}{\Delta t} \cdot d$$

Finally we also need to resolve the penetration. We have two options here:

- Instead of driving the velocity to zero we can target for a small separating velocity proportional to the penetration depth per tick (exponential decay)
- This is called Baumgarte stabilization

Alternatively we can run a full solver sweep over the contacts again, but now solving the position error directly

- This is called position projection

Contact solving is not the topic of the talk today, but hopefully this gives you an idea how the contact information might be used in the solver.

Outline

- Introduction
- **Basic Strategies**
- Solving the Collision Table
- Contact Optimizations
- Code Examples

Now let’s continue with the basic strategies for creating contact points

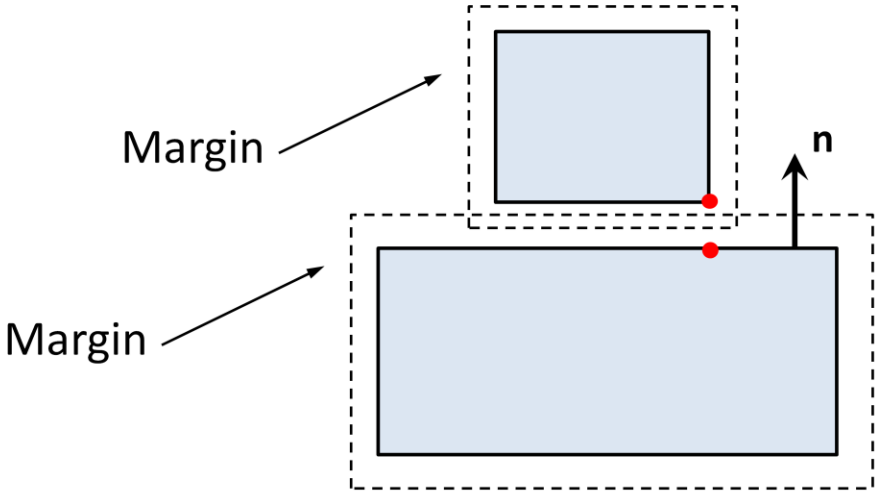
Incremental vs One-Shot Manifold

- **Incremental Manifold**: Find closest points (e.g. using GJK/EPA) to incrementally build a contact manifold point-by-point over several frames
- **One-Shot Manifold**: Find closest features (e.g. using SAT) to compute a contact manifold in one shot using clipping techniques

In order to model contact in our engine we need a contact point location, a contact normal and the penetration depth. There are two basic approaches to find contact points:

1) The incremental approach tries to find one contact point per frame and adds it to a persistent manifold.
2) The one-shot approach detects the closest features and finds all contact points of a manifold in one frame using clipping techniques

Building Incremental Manifolds (1)

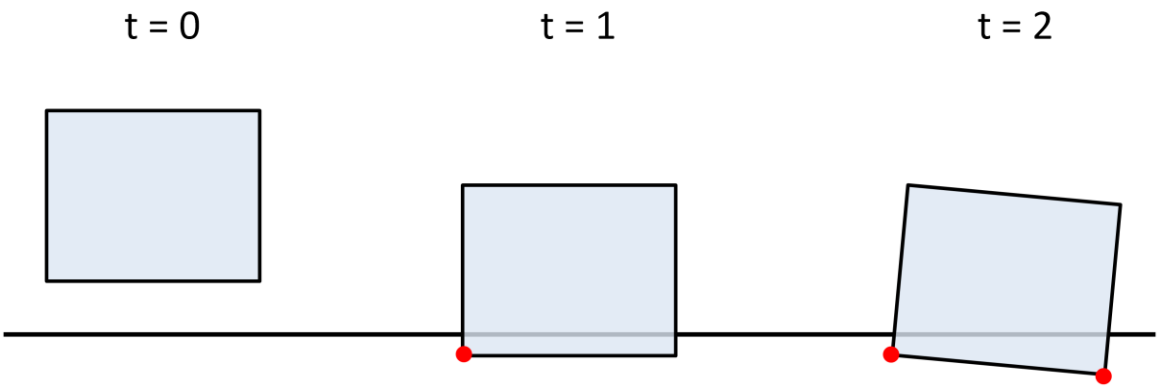


The basic idea is to inflate your collision shapes by a small margin and then use e.g. GJK to compute the closest points between the core shapes.

- This gives you the contact point and normal
- The penetration depth is the margin minus the distance between the closest points
- The new contact point is then added to a persistent manifold
- Old contact points need to be confirmed (e.g. using some distance heuristic or feature IDs)

Obviously this will only work if the core shapes don't overlap.
If the core shapes are actually overlapping we fallback onto some other algorithm (e.g. EPA, SAT, MPR, or brute force sampling)

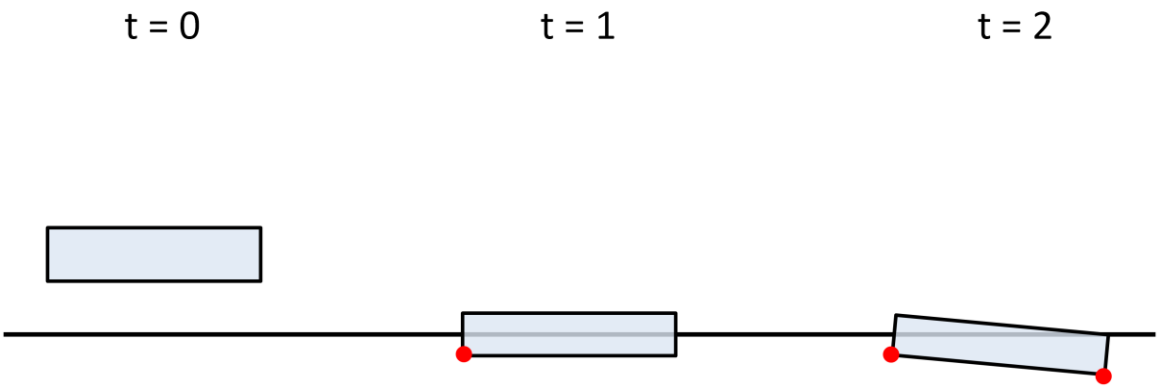
Building Incremental Manifolds (2)



Let's have a quick look how the incremental manifold is constructed over several frames now:

- As you can see there are some potential issues with this approach
- Since we only find one contact point at frame 1 we introduce an artificial torque which can actually be quite noticeable to the player
- Also note that we continue penetrating in the next frame since we need several frames to construct a stable manifold

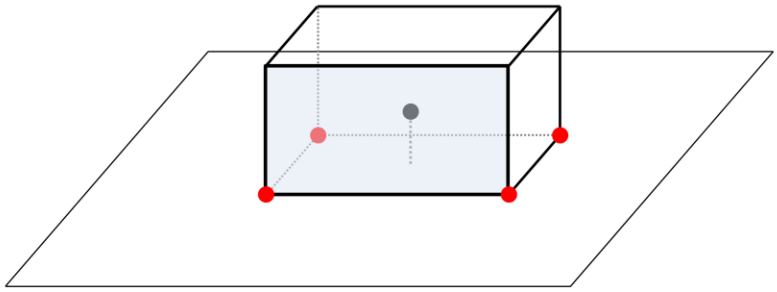
Building Incremental Manifolds (3)



In the worst case (depending on the geometry of our shapes) objects might even rotate out of the world!

- If this is a simple debris object this might not matter
- If it is key to reach the level exit it is a AAA bug

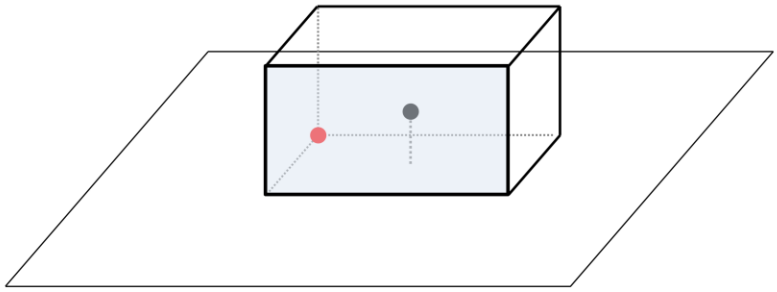
Incremental Manifold Stability



So what is a stable contact?

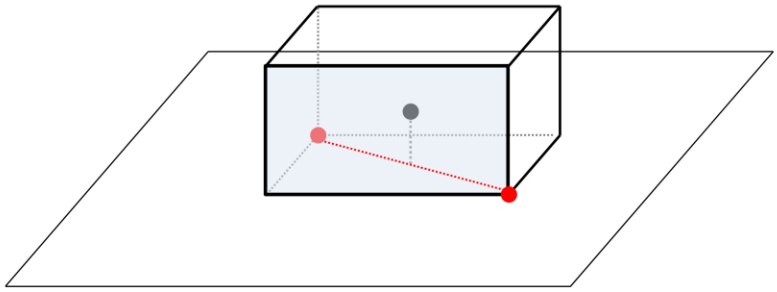
- A contact manifold is stable if the center of mass projects inside the manifold.
- Obviously we need (at least!) up to four frames to construct a stable manifold using the incremental approach
- Let's look into this in a bit more detail

Incremental Manifold Stability (1)



We find the first point:
- This is obviously unstable as we can rotate freely around the first contact point

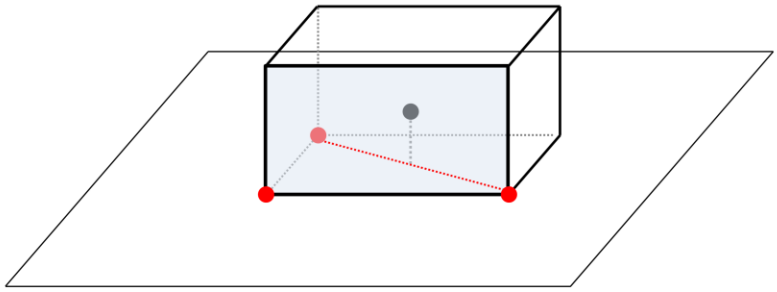
Incremental Manifold Stability (2)



Now we find a second point across the diagonal:

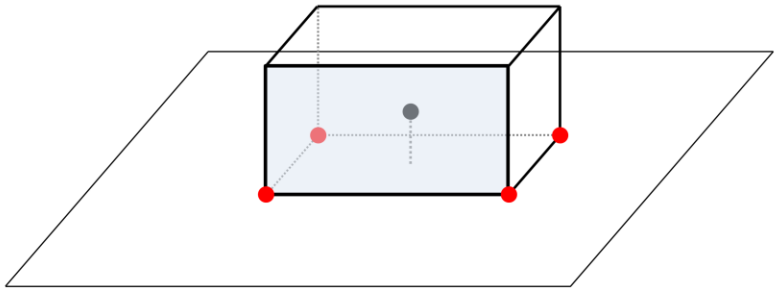
- This is still unstable as we can rotate around the axis through the two contact points across the diagonal
- Think of a hinge here if you like

Incremental Manifold Stability (3)



We find a third point which is still unstable as we can still rotate around the axis through the two contact points across the diagonal
- Note how the center of mass projects onto the edge of the manifold in this case

Incremental Manifold Stability (4)



Finally a stable manifold after a **minimum** of four frames.

Incremental Manifold Summary

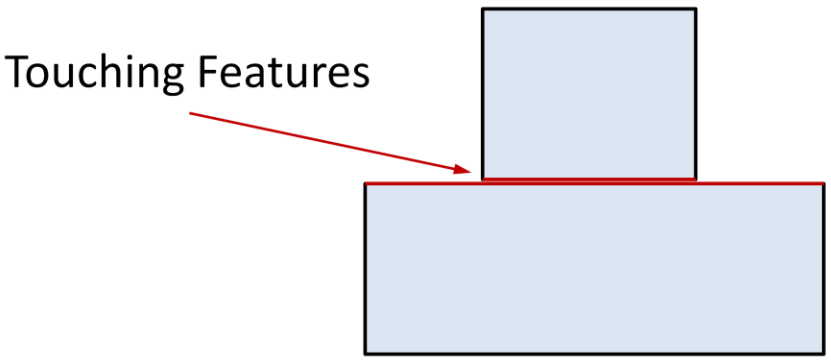
- Simple and fast
- Shipped successfully many titles
- Minor limitations

Summary:

- Incremental manifolds are a great and simply solution.
- Many games have shipped successfully with this approach
- Hopefully this summary helps you to make an educated decision understanding the limitations of this approach and if this is the right solution for your project!

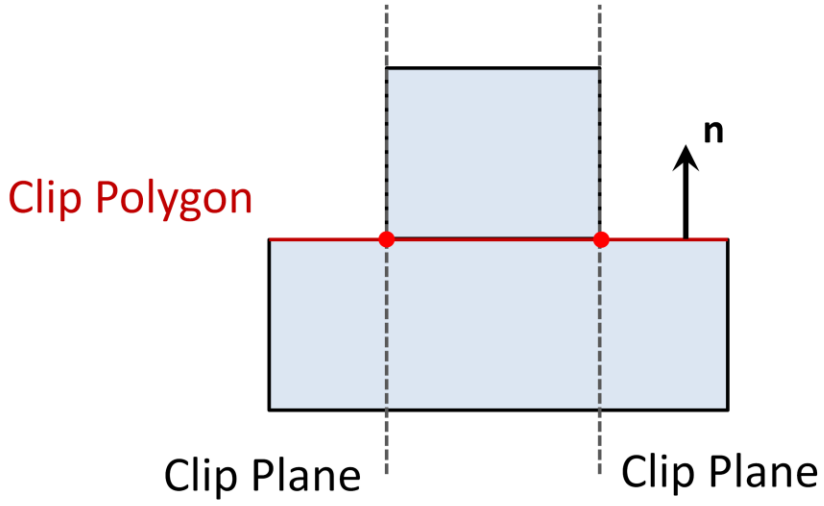
The remainder of the talk will be about the construction of one-shot manifolds.

Building One-Shot Manifolds (1)



To construct a one shot manifold we try to detect the touching features (e.g. using the SAT)

Building One-Shot Manifolds (1)

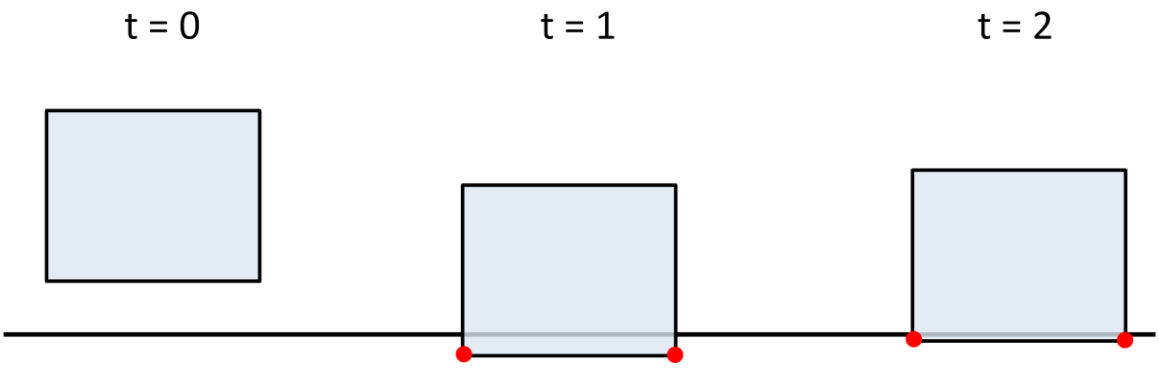


And then we clip one feature against the side planes of the other to find all contact points

- The contact points come obviously from clipping procedure
- And the normal and penetration from the touching features
- This is just the basic idea and we will cover all the fancy details in the talk!

Note that we didn't use any margin here though we definitely could!

Building One-Shot Manifolds (2)



Now let's have a look at the one-shot manifold approach over several frames:

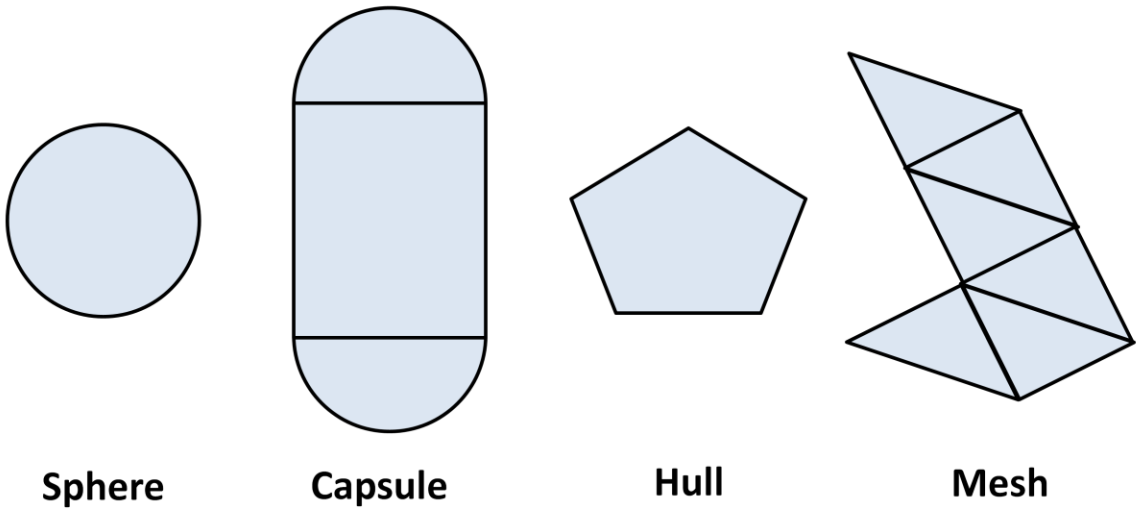
- As you can see the one-shot approach finds both contact points at frame 1 when the two shapes start touching.
- Hence there is no-artificial torque and also no stability issues due to rotations
- Since all contact points are found at frame 1 the object is stopped from penetrating further in the next frame
- Also note how the penetration recovery can start working immediately
- This helps a lot with removing visual artifacts like spongy contact points

Outline

- Introduction
- Basic Strategies
- **Solving the collision table**
- Contact optimizations
- Code examples

So now that we know what a contact is let's look at some common shapes and how to compute stable contact points between them in detail!

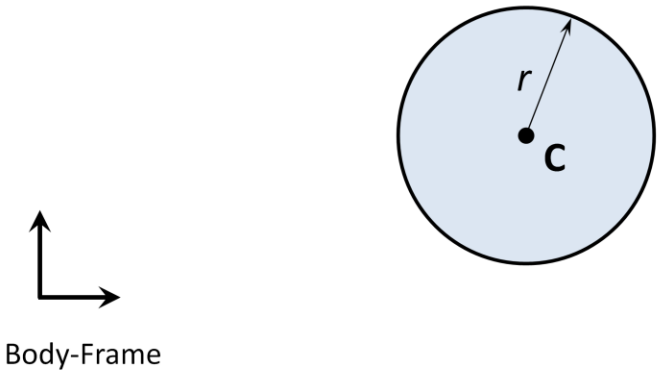
Collision Shapes



In Rubikon we support four different kinds of collision shapes:

- Spheres
- Capsules
- Convex Hulls
- Triangle Meshes

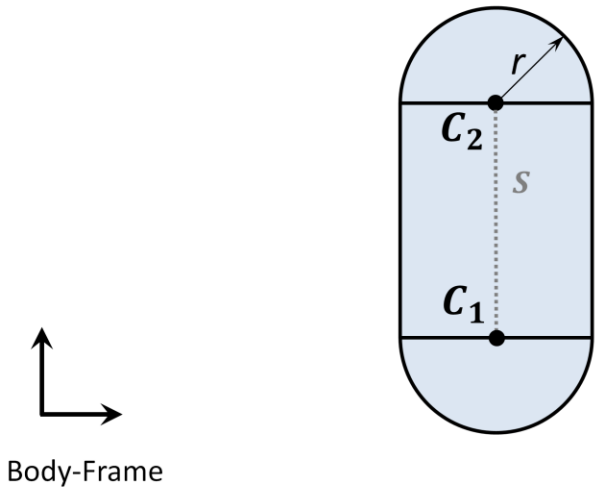
Collision Shapes: Sphere



A sphere in Rubikon is defined by a center point and radius

- There is no shape transform and the center point is considered to be local to the parenting rigid body (or any other parent if used in a different context)

Collision Shapes: Capsule

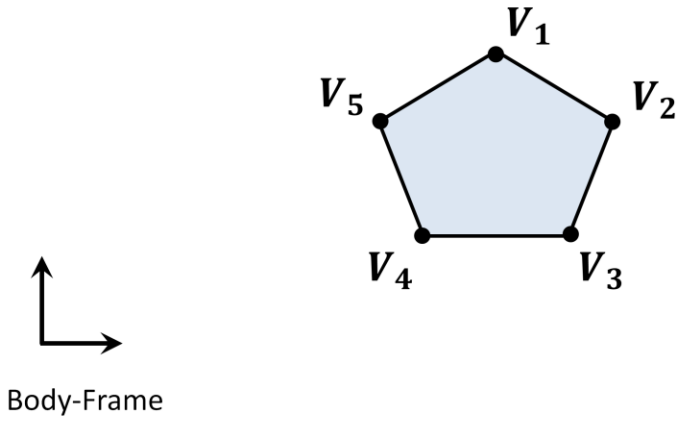


Similarly a capsule is defined by two center points and a radius
- Again there is no shape transform and the center points are considered to be in the local space of the rigid body again
- The nice thing about this definition is that you don't have to deal with any up direction

In the upcoming slides I will quite often refer to the inner segment of the capsule
- The inner segment is defined by the two center points
- Similarly to thinking of a sphere as a point with radius, I like to think of a capsule as a segment with a radius

As a side note:
- It is very easy to setup collision capsules for ragdoll limbs this way.
- For legs and arms you simply snap the two center points to the bone pivots and scale the radius. Done!

Collision Shapes: Convex Hull



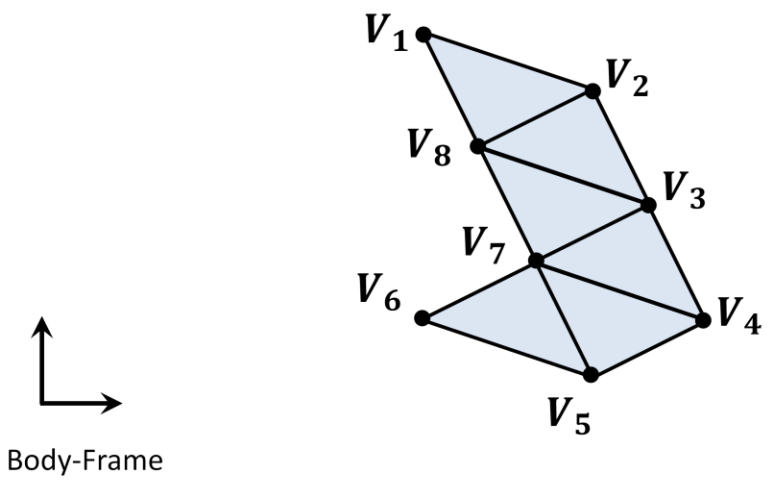
The main collision shape in Rubikon is the convex hull.

- The convex hull is defined by a set of vertices in the local space of the parenting rigid body.
- We also store topological information for the hull like edges and faces (e.g. using the half-edge data structure)

As you might have noticed Rubikon doesn't support cylinders or cones.

- If needed, we describe those as reasonably tessellated convex polyhedra
- Rubikon also has no special box shape as this can be efficiently handled as convex hull with the methods I will show you today
- You should target for your convex hull performance such that a hull with 8 vertices and 6 faces shows similar performance characteristics as a specialized box shape!

Collision Shapes: Triangle Mesh



The final collision shape in Rubikon is the triangle mesh

- We define the triangle mesh as a set of vertices and faces
- Again vertices are defined relative to the parenting rigid body
- Triangle meshes are usually used to describe the static geometry in a game level

The Collision Table

	Sphere	Capsule	Hull	Mesh
Sphere	**YES**	**YES**	**YES**	**YES**
Capsule		**YES**	**YES**	**YES**
Hull			**YES**	**YES**
Mesh				**NO**

Rubikon supports all interactions between the introduced shapes but mesh vs mesh:

- Mesh vs mesh is pretty expensive to solve robustly in **real-time** applications
- Good methods exist to solve this problem (e.g. SDF), but will not be covered in this talk today.

The Collision Table: Tools of the Trade

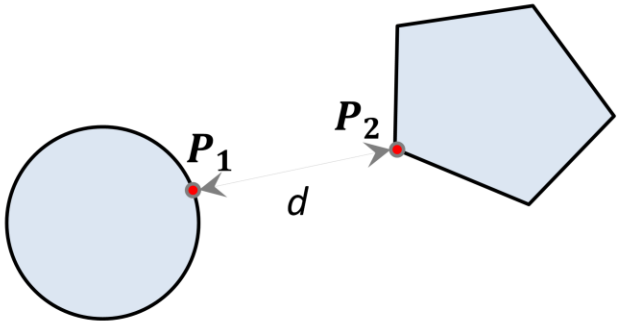
- Contact creation makes heavy use of two well-know algorithms: **GJK** and **SAT**
- Both have been discussed in the physics tutorial exhaustively before
- Check the reference section for some reading suggestions

Contact creation (as presented here) makes heavy use of two well-know algorithms: **GJK** and **SAT**

- Both have been discussed in the physics tutorial exhaustively before and we cannot go into detail here today
- Luckily the essential outcome is easily explained and you can go back to earlier presentations for an in-depth explanation
- I put reading suggestions into the reference section

There is another an algorithm called the Expanding Polytope Algorithm (**EPA**), but we will not discuss it here today!

Recap: GJK

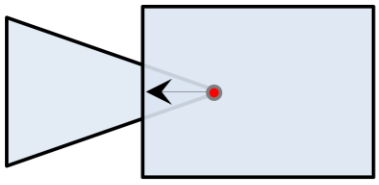


GJK finds the closets points and distance between two **disjoint** convex shapes

Let's quickly recap GJK. GJK is this fancy algorithm which uses support functions, Minkowski Sums, Simplex Solvers and what not else. How this all works is not the topic today and all we **really** need to remember is:

- GJK is an algorithm which computes the closets points between two **disjoint** convex shapes
- **Disjoint** is the important thing to remember here!
- If the two objects are touching or even penetrating GJK gives **no** (immediate) result we can use
- For the remainder of this talk we treat GJK as a black box which can solve the distance problem between two convex shapes for us!

Recap: SAT



The SAT finds the axis of minimum translation between two **overlapping** convex polyhedra, the penetration distance and the touching features

The SAT is pretty fancy as well and uses techniques like Minkowski Sums, Gauss Maps, and many more to find a finite set of planes we need to test for separation. Again we don't care for the details here and treat the SAT as a black box as well. If we **cannot** find a separating axis the SAT will tell us things like:

- The direction in which we need to resolve the penetration (Axis of Minimum Penetration)
- The **minimum** penetration distance
- The touching features

The Collision Table: Convex Contact

	Sphere	Capsule	Hull	Mesh
Sphere	X	X	X	
Capsule		X	X	
Hull			X	
Mesh				

We start with the collision between the convex shapes (e.g. sphere, capsule, and hull)

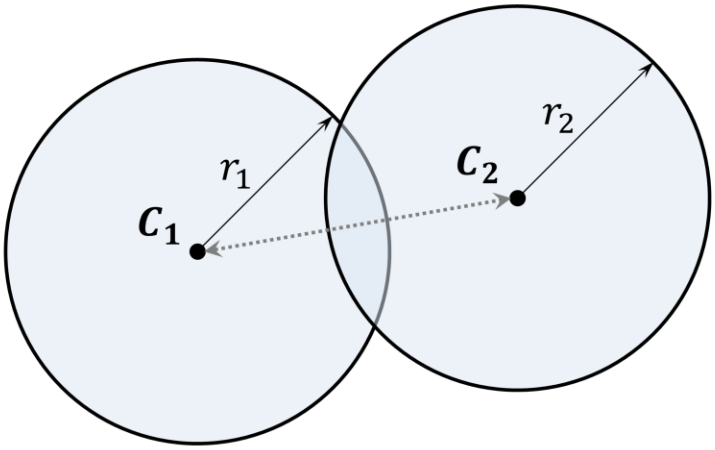
- The contact between these shapes is usually well defined by a single manifold

Sphere vs Sphere

	Sphere	Capsule	Hull	Mesh
Sphere	X	X	X	
Capsule		X	X	
Hull			X	
Mesh				

We start simple with sphere vs sphere!

Sphere vs Sphere

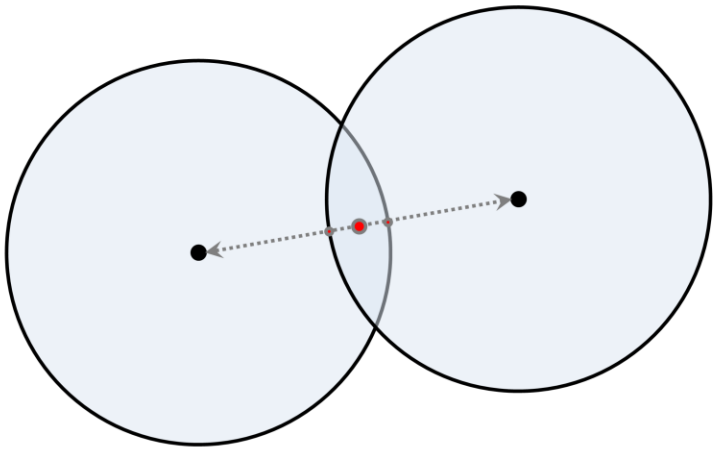


$$Overlap <=> |\boldsymbol{C_2} - \boldsymbol{C_1}| - r_1 - r_2 \leq 0$$

First we need to detect whether the two sphere shapes overlap using a simple distance function:

- Two spheres overlap if the distance between the center points is less or equal than the sum of the radii

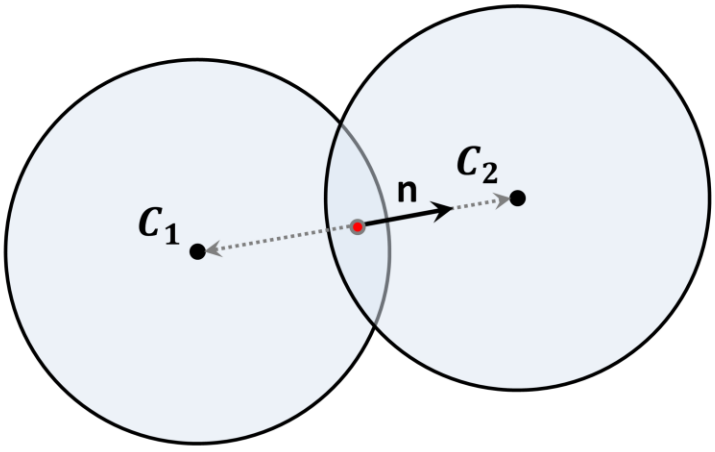
Sphere vs Sphere: Contact Point



If we detected overlap we need to create a contact manifold:

- In the case of sphere vs sphere one contact point is sufficient
- For sphere vs sphere I place the contact into the middle of the two surface points (not the middle of the two center points!)

Sphere vs Sphere: Contact Normal

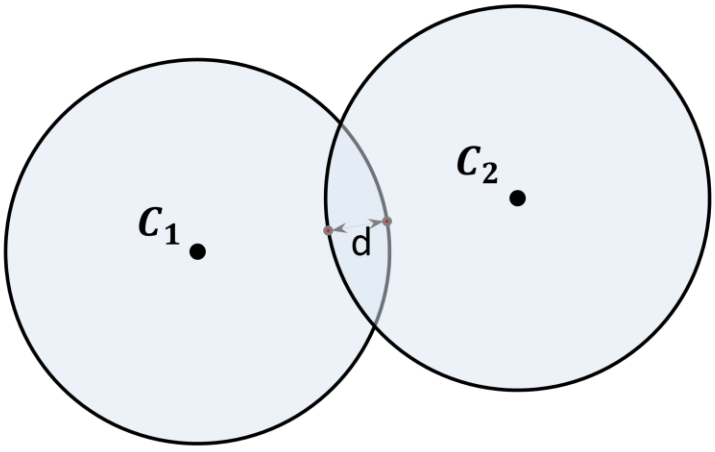


$$n = C_2 - C_1/|C_2 - C_1|$$

We also need a normal direction.

- The normal is simply the normalized difference of the center points

Sphere vs Sphere: Penetration Depth

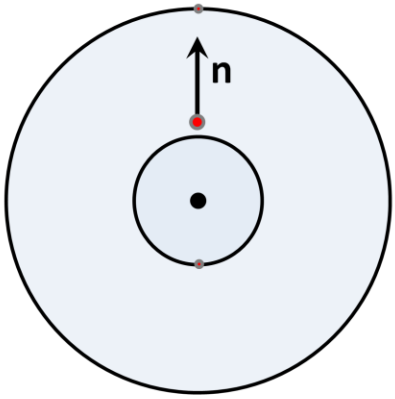


$$d = |C_2 - C_1| - r_1 - r_2$$

Finally we need to compute the penetration depth:

- The penetration depth is simply the distance between center points minus the sum of sphere radii: d = |C2 – C1| - r1 – r2
- We essentially just evaluate our distance function

Sphere vs Sphere: Degenerate Case



We also need to handle one degenerate case:

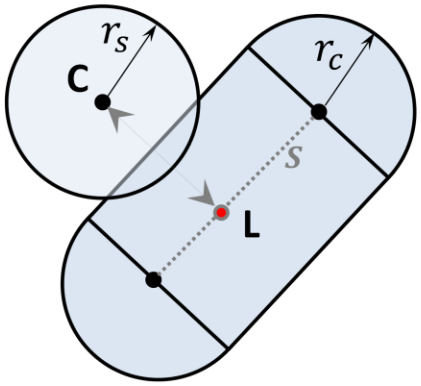
- If the spheres are coincident we cannot compute a normal with the approach described above
- If these happens we simply choose the global up axis as our normal direction (or anything else that makes sense in your context)
- Note that it is also valid to create no contact at all in the degenerate case.
- An arbitrary normal can be the wrong choice and this usually only happens if entities are spawned on top each other in the game!

Sphere vs Capsule

	Sphere	Capsule	Hull	Mesh
Sphere	X	X	X	
Capsule		X	X	
Hull			X	
Mesh				

This was easy! The next pair we will investigate is sphere vs capsule.

Sphere vs Capsule

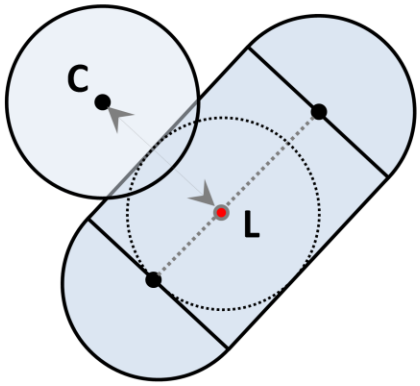


$$Overlap <=> |\boldsymbol{C} - \boldsymbol{L}| - r_S - r_C \leq 0$$

The sphere overlaps the capsule if the distance of the sphere center to the closest point on the inner capsule segment is less or equal than the sum of the radii

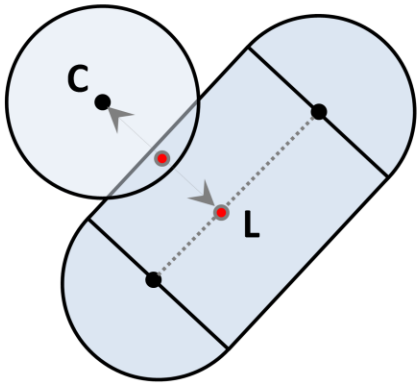
- Note that there is no need to distinguish between different regions of the capsule (e.g. the caps or the cylindrical body of the capsule)

Sphere vs Capsule



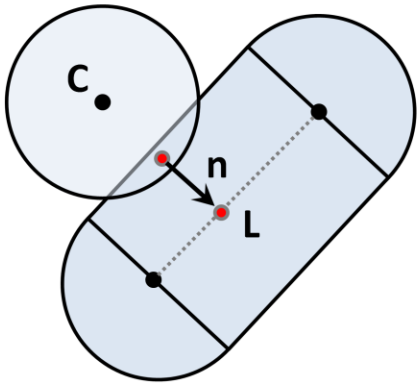
Creating the contact point is now similar to sphere vs sphere:
- We can conceptually create a virtual sphere around the closest point L and build the contact information as before

Sphere vs Capsule: Contact Point



- We put the contact location into the middle of the two surface points (not the middle of the center and closest point on segment!)

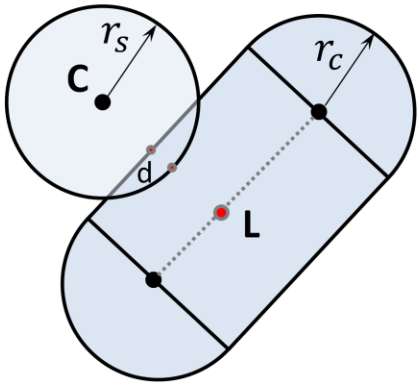
Sphere vs Capsule: Contact Normal



$$\boldsymbol{n} = \boldsymbol{L} - \boldsymbol{C}/|\boldsymbol{L} - \boldsymbol{C}|$$

- The normal is simply the normalized difference vector of the sphere center and the closest point on segment

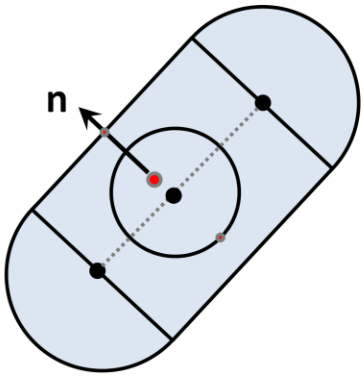
Sphere vs Capsule: Penetration Depth



$$d = |\boldsymbol{C} - \boldsymbol{L}| - r_S - r_C$$

- Finally the penetration depth is simply the distance between the center point and the closest point on the segment minus the sum of the radii: d = |L – C| - rs – rc
- Again we are simple evaluating our distance function

Sphere vs Capsule: Degenerate Case



Again we also need to handle one degenerate case here:

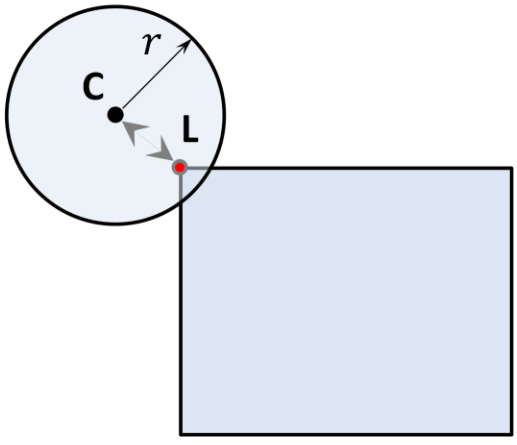
- If the sphere center is exactly on the inner segment we cannot compute a normal
- In this case we can choose any direction which is perpendicular to the inner capsule segment
- Again an arbitrary normal can be the wrong choice and it is also valid to ignore this case.

Sphere vs Hull

	Sphere	Capsule	Hull	Mesh
Sphere	X	X	X	
Capsule		X	X	
Hull			X	
Mesh				

Not to difficult as well. Let’s continue in our collision table with sphere vs convex hull.

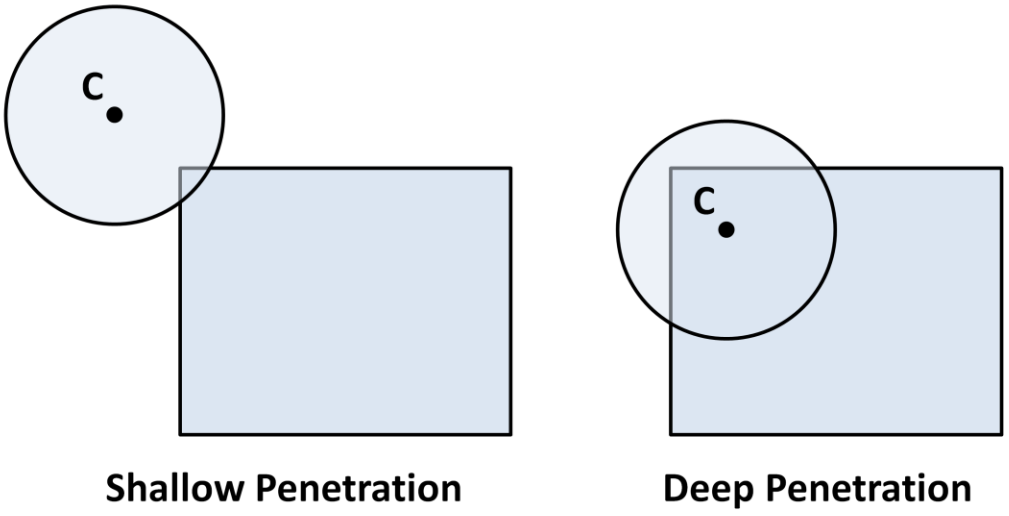
Sphere vs Hull



$$Overlap <=> |\boldsymbol{C} - \boldsymbol{L}| - r \leq 0$$

The sphere overlaps the hull if the distance of the sphere center to the hull is less or equal than the sphere radius

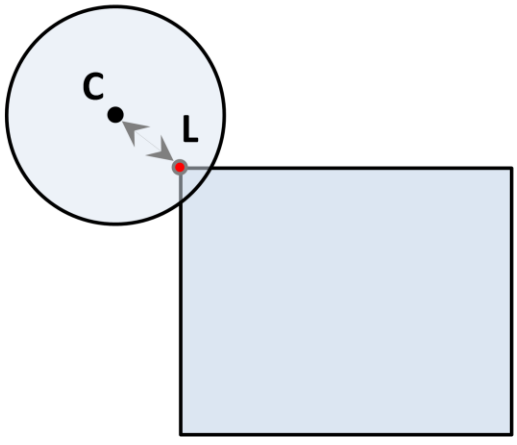
Shallow and Deep Penetration



As mentioned in the beginning we are equipped with two basic algorithms: GJK and SAT

- The GJK can handle the configuration when two convex shapes are disjoint
- The SAT can handle the configuration when two convex shapes are overlapping
- Hence we need to distinguish two situations: Shallow and Deep Penetration
- Shallow penetration is when the sphere center is outside of the hull, but the distance is less than the radius
- Deep penetration is when the sphere center is contained inside the hull

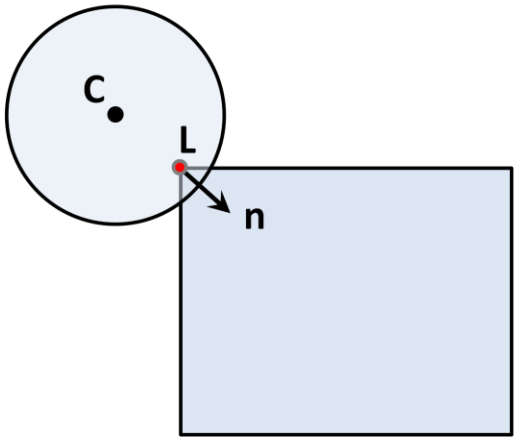
Sphere vs Hull: Shallow Penetration



We start assuming that the sphere center is outside the hull using GJK:

- If GJK succeeds and reports a distance less than the radius we built a shallow contact point
- If GJK fails we assume that the sphere center is inside the convex hull and run a SAT
- Note that we use GJK with the sphere center and not the sphere itself here!

Sphere vs Hull: Shallow Contact

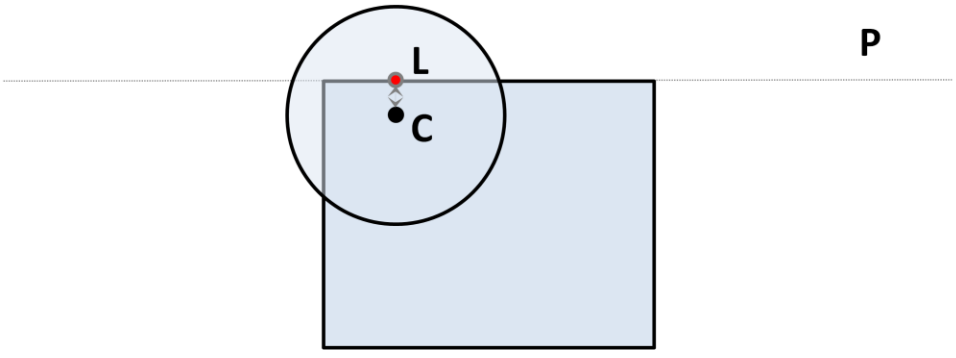


$$d = |\boldsymbol{C} - \boldsymbol{L}| - r$$

Building a shallow contact point is easy:

- The contact point is simply the closest point on the hull (returned from GJK)
- The contact normal is the difference between the closest point on the hull and the sphere center
- The penetration depth is the distance between the closest points minus the sphere radius: d = |C - L| - r

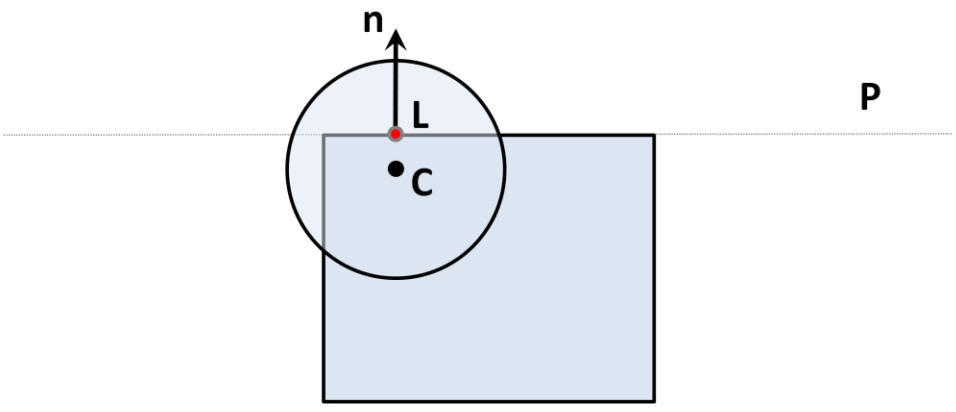
Sphere vs Hull: Deep Penetration



If GJK fails the sphere center is contained inside the hull and we need to run a SAT to find the axis of minimum penetration

- This case is actually quite simple and the possible separating axes are only the face normals of the convex hull
- The face with the smallest distance to the sphere center defines the axis of minimum penetration

Sphere vs Hull: Deep Contact



Building a deep contact point is straight forward:

- The contact point is the projection of the sphere center onto the minimizing face
- The contact normal is simply the normal of the minimizing face
- The penetration depth is the distance of the sphere center to the minimizing face minus the sphere radius: e = d(P, c) - r

Note (if you ae concerned about the correct signs):

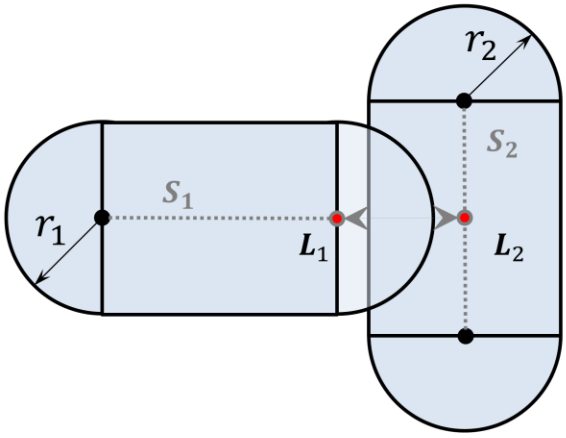
- The distance of the sphere center to the face plane is negative as the center point is behind the plane!

Capsule vs Capsule

	Sphere	Capsule	Hull	Mesh
Sphere	X	X	X	
Capsule		X	X	
Hull			X	
Mesh				

Things get a little bit more exciting, but still everything is pretty straight forward. Let's step down one row in out collision table and look into capsule vs capsule.

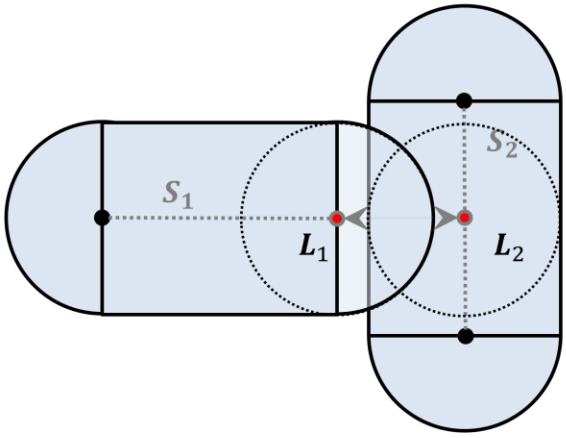
Capsule vs Capsule



$$Overlap <=> |\boldsymbol{L}_2 - \boldsymbol{L}_1| - r_1 - r_2 \leq 0$$

Two capsules overlap if the distance between their inner segments is less or equal than the sum of their radii!

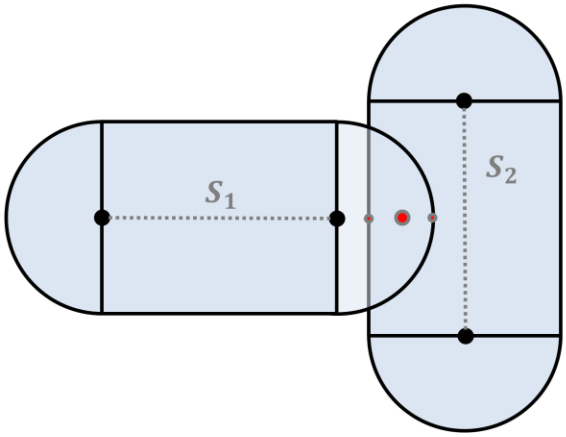
Capsule vs Capsule



$$Overlap <=> |\boldsymbol{L}_2 - \boldsymbol{L}_1| - r_1 - r_2 \leq 0$$

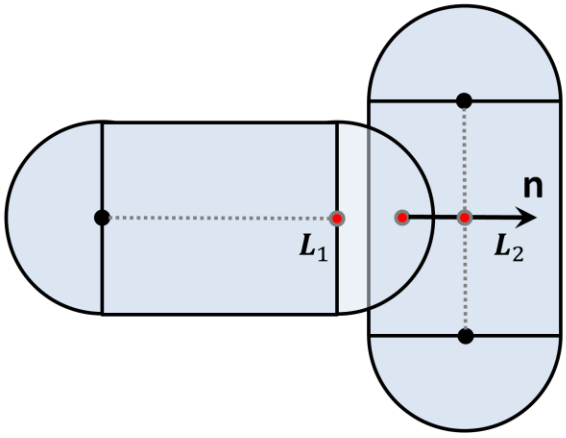
Once we have the closest points we can build virtual spheres around the closest points again and think of this as sphere vs sphere.

Capsule vs Capsule: Contact Point



As for sphere vs sphere we put the contact into the middle of the two surface points

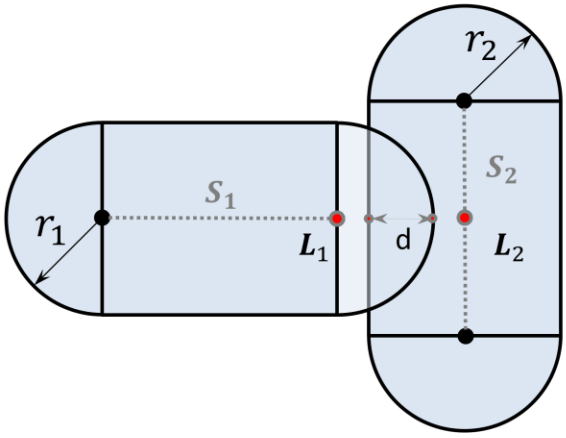
Capsule vs Capsule: Contact Normal



$$n = L_2 - L_1/|L_2 - L_1|$$

The normal is simply the normalized difference of the two closest points

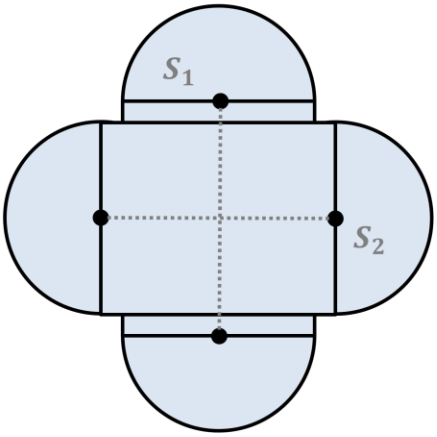
Capsule vs Capsule: Penetration Depth



$$d = |L_2 - L_1| - r_1 - r_2$$

Finally we need to compute the penetration depth that we need to resolve the overlap:
- The penetration depth is simply the distance between the closest points minus the sum of the radii: d = |L2 – L1| - r1 – r2

Capsule vs Capsule: Degenerate Case



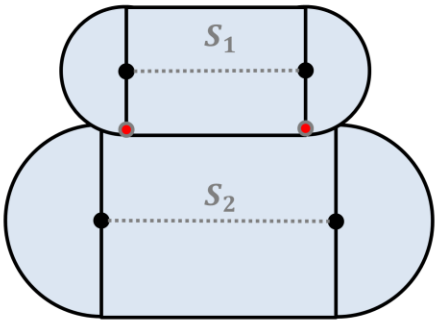
We need to also handle one degenerate case:

- If the inner segments overlap we cannot compute a normal

We need to distinguish two cases here:

- If the segments are not parallel we can use the cross product to find a normal
- Otherwise we can choose any direction which is perpendicular to any of the two inner capsule segments
- Again skipping this case is a valid option as an arbitrary normal can be the wrong choice!

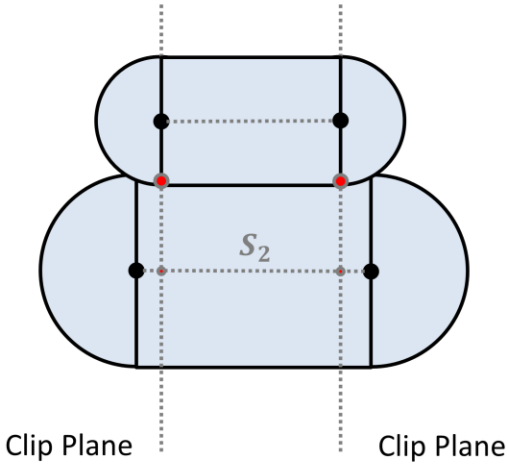
Capsule vs Capsule: Stacking



Now assume we actually would like to stack capsules in our game:

- In the parallel case we might want to consider to create two contact points
- So before we run the test shown earlier we test whether the two capsules can be considered parallel

Capsule vs Capsule: Clipping



If we detect two parallel capsules we try to create two contact points using clipping:

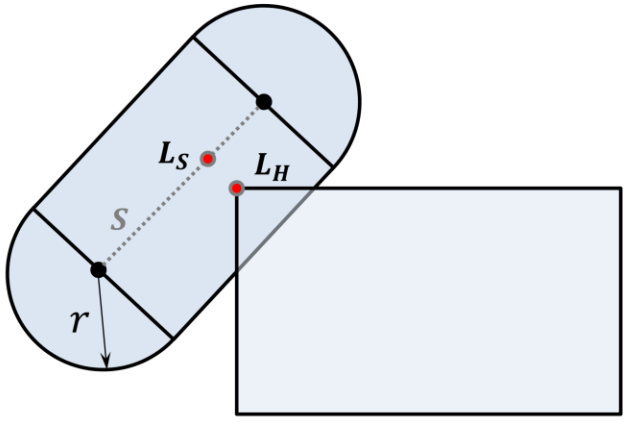
- We clip the inner segment of the second capsule against the side planes of the inner segment of the first capsule
- Then we keep the points whose distance to the first segment is less that the sum of the radii
- Finally we shift the clipped points to the middle of the associated surface points

Capsule vs Hull

	Sphere	Capsule	Hull	Mesh
Sphere	X	X	X	
Capsule		X	X	
Hull			X	
Mesh				

Finally we created more than one contact point for this first time. So things start getting to be a bit more involved. Let's move on with capsule vs convex hull!

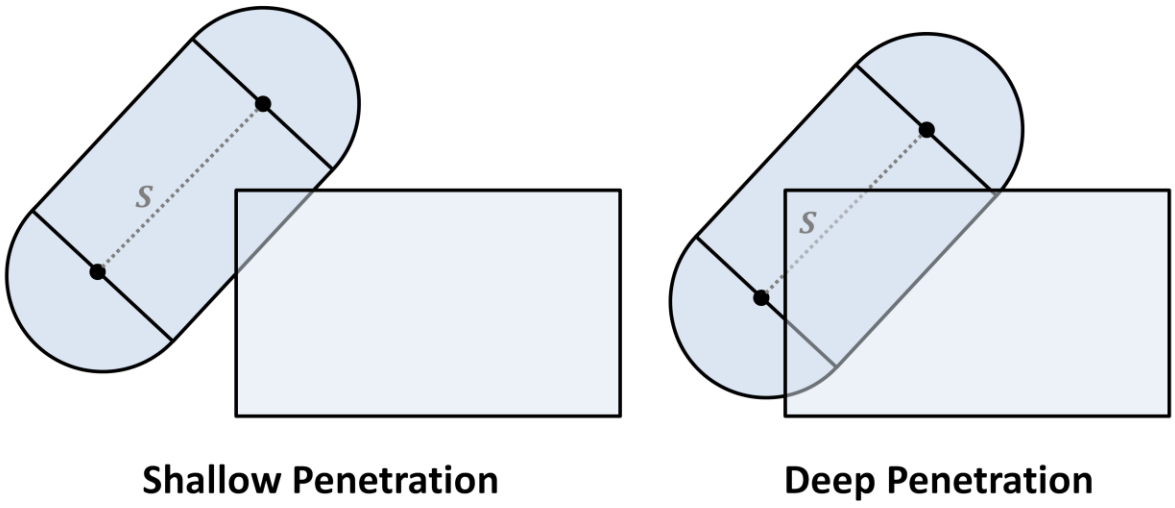
Capsule vs Hull



$$Overlap <=> |\boldsymbol{L_H} - \boldsymbol{L_S}| - r \leq 0$$

The capsule overlaps the hull if the distance of the inner capsule segment to the hull is less or equal than the capsule radius

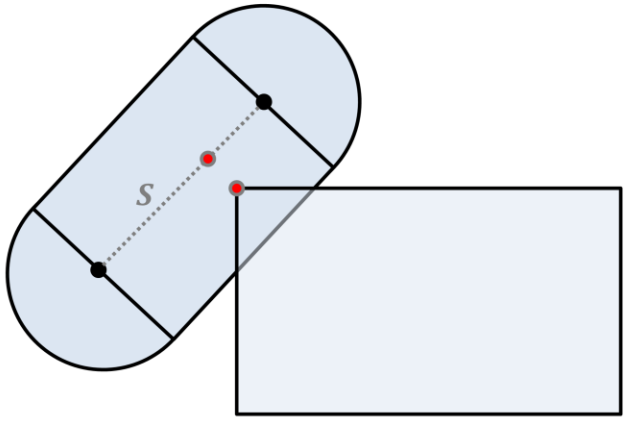
Shallow and Deep Penetration



As already learned in the sphere vs hull case we need to distinguish between shallow and deep penetration:

- We need to handle shallow penetration if the inner capsule segment is outside of the hull, but the distance is less than the radius
- We need to handle deep penetration if the inner capsule segment is intersecting the hull

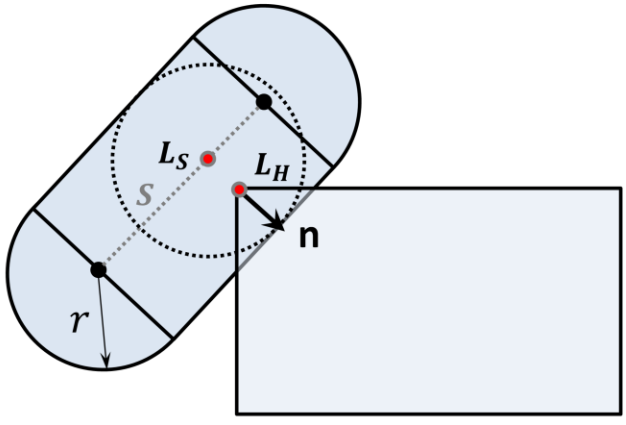
Capsule vs Hull: Shallow Penetration



We start assuming that the inner capsule segment is outside the hull using GJK:

- If GJK succeeds and reports a distance less than the radius we built a shallow contact point
- If GJK fails we assume that the inner capsule segment is inside the convex hull and run a SAT
- Note again that we use GJK with the inner capsule segment and not the capsule itself here!

Capsule vs Hull: Shallow Contact

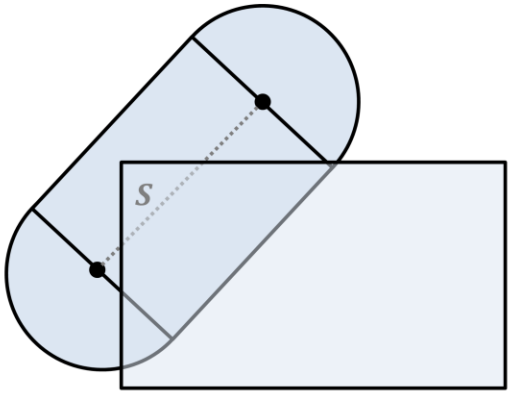


$$d = |\boldsymbol{L_H} - \boldsymbol{L_S}| - r$$

Building a shallow contact point is easy:

- Once we detected the closest point we can again build a virtual sphere around it and use the same ideas as before for sphere vs hull
- The contact point is simply the closest point on hull (returned from GJK)
- The contact normal is the normalized difference between the closest points
- The penetration depth is the distance between the closest points minus the capsule radius: d = |C - L| - r

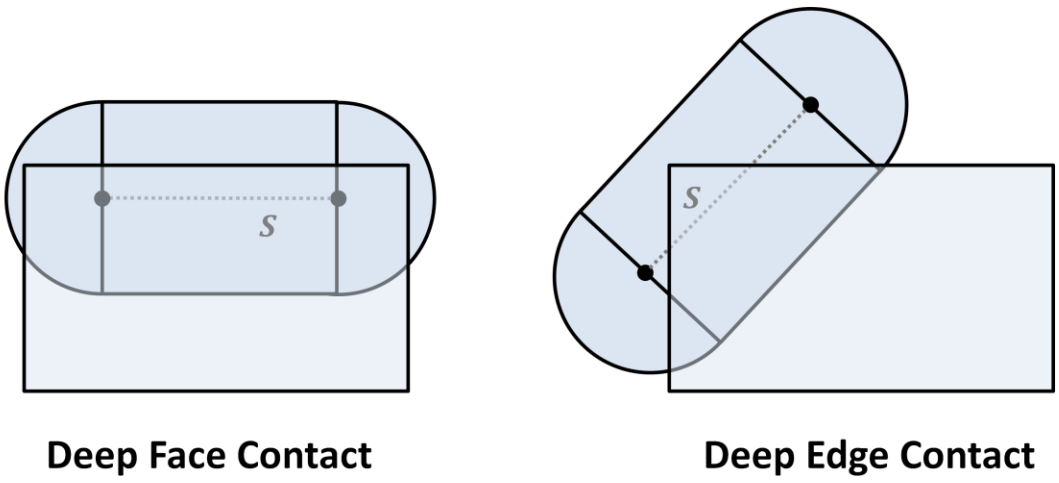
Capsule vs Hull: Deep Penetration



If GJK fails the inner capsule segment is contained inside the hull and we need to run a SAT to find the axis of minimum penetration

- The possible separating axes are the face normals of the convex hull and the cross products between the inner capsule segment and the edges of the hull
- This is where things get a bit more involved now, but luckily this case should not happen too often in practice!

Capsule vs Hull: Deep Contact

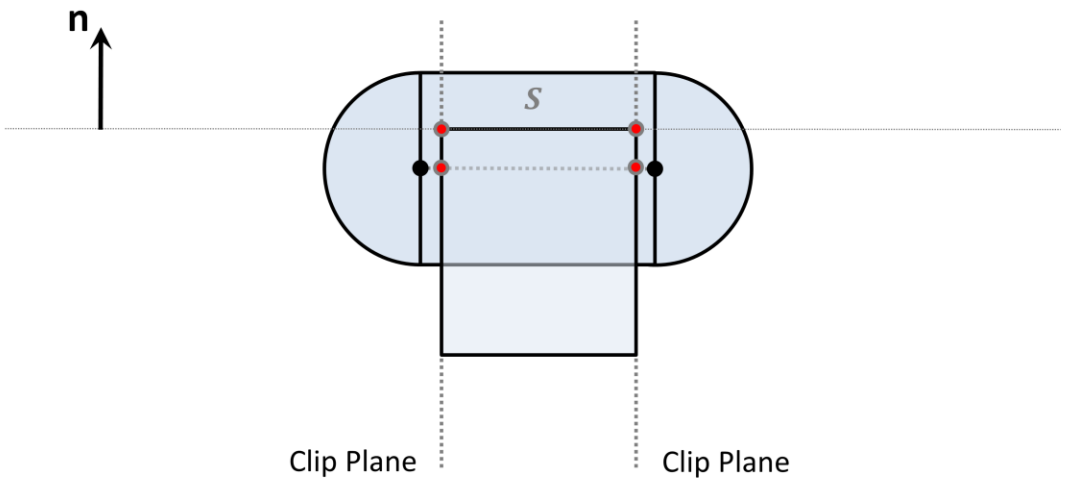


Deep Face Contact **Deep Edge Contact**

For building a deep contact we need to distinguish two cases:

- The first case is if the axis of minimum penetration is a face normal
- The second case is if the axis of minimum penetration is the cross product of the inner capsule segment and one of the hull edges

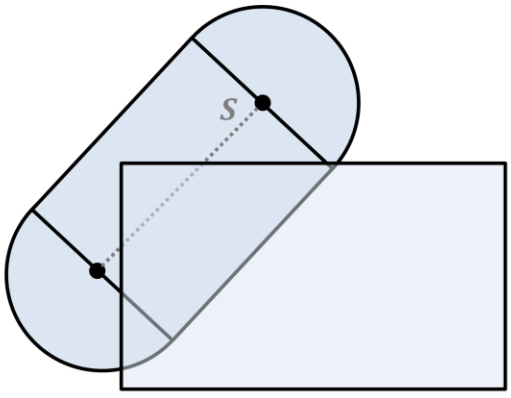
Capsule vs Hull: Deep Face Contact



Let's start with the simple case if the axis of minimum penetration is associated with a face of the convex hull:

- In this case we simply clip inner capsule segment against the side planes of the minimizing face
- The normal comes from the plane of the minimizing face
- And the penetration depth is the distance of the clipped points to the minimizing face minus the capsule radius
- Finally we project the clip-points onto the minimizing face

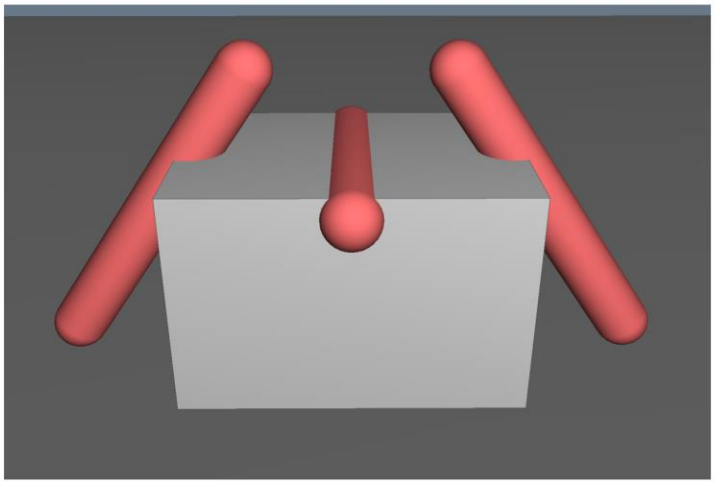
Capsule vs Hull: Deep Edge Contact



Now let's investigate the case if the axis of minimum penetration is realized by the cross product of the inner capsule segment and an edge of the convex hull:

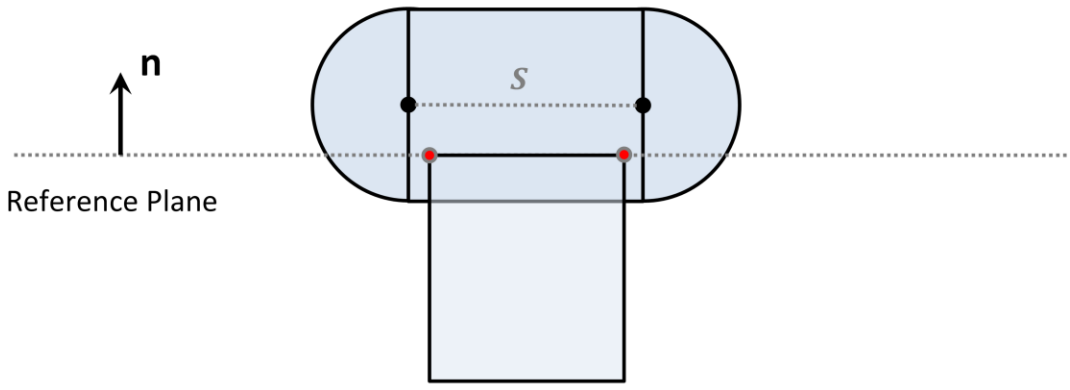
- We compute the closest points between the minimizing edges and choose the contact point in the middle
- The normal and penetration depth come directly from the SAT

Capsule vs Hull: Deep Contact Example



I created a screenshot of deep capsule penetration as this might be difficult to imagine from the 2D sketches:
- The slide shows two capsules (side) in deep **edge** contact and one capsule (middle) in deep **face** contact

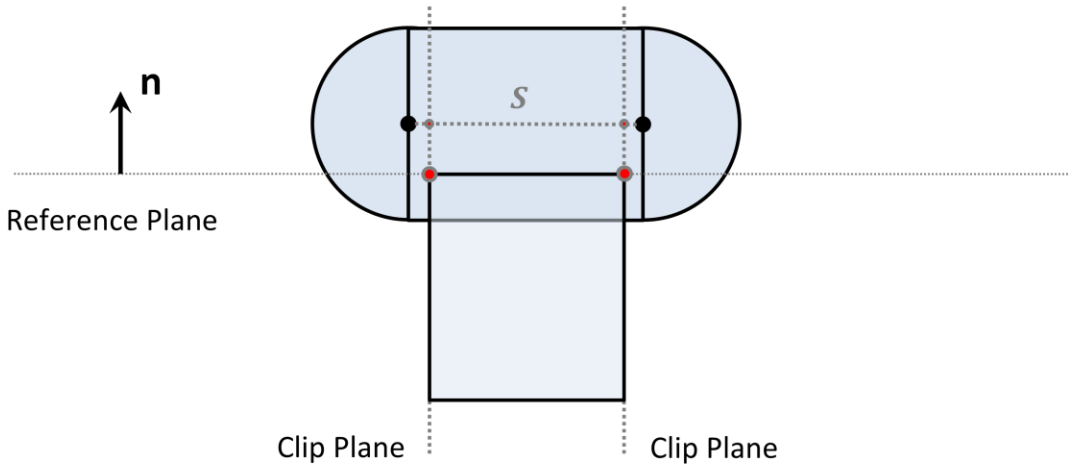
Capsule vs Hull: Stacking



In the case of a deep face contact our algorithm produces two contact points. Since deep penetration should only occur very seldom we also like to create two contact points in a similar shallow contact configuration

- In order to detect this case we need to iterate over all faces of the convex hull and check if our contact normal is parallel to a face normal.
- If we succeed we call this the **reference** face!

Capsule vs Hull: Clipping



In order to create two contact points we need to clip again:

- We simply clip the inner segment of the capsule against the side planes the **reference** face as in the deep penetration case
- The normal comes from the plane of the minimizing face
- And the penetration depth is the distance of the clipped points to the minimizing face minus the capsule radius
- Finally we project the clip-points onto the minimizing face again

Note (if you ae concerned about the correct signs):

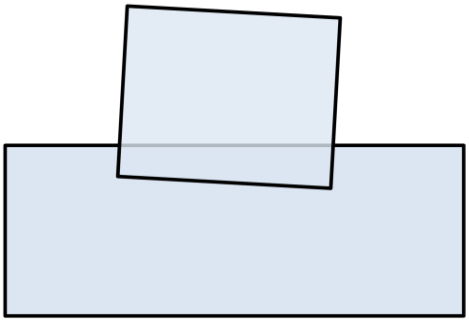
- The distance of the clipped points to the reference plane is now positive as the inner segment is in front of the plane!

Hull vs Hull

	Sphere	Capsule	Hull	Mesh
Sphere	X	X	X	
Capsule		X	X	
Hull			X	
Mesh				

Finally we arrived at the last convex contact pair which will also be the most complicated one for today!

Hull vs Hull



$Overlap <=> No\ separating\ axis$

So far we detected overlap by defining a distance function

- In the case of the two convex hulls we use a SAT based approach to detect overlap
- Hence the two hulls overlap if there is no separating axis

Hull vs Hull: The Separating Axis Test

- The possible separating axes between two convex polyhedra are:
  - The face normals of polyhedron A (2D & 3D)
  - The face normals of polyhedron B (2D & 3D)
  - **The cross product between all edge combinations of A and B (3D only)**

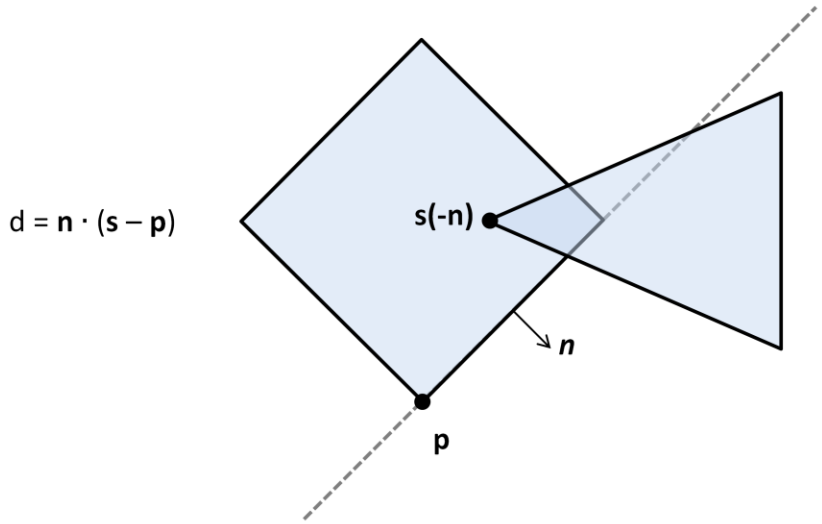
Let's recap the possible separating axes between two convex polyhedra:
- The face normals of hull A
- The face normals of hull B
- The cross products between all edges of A and all edges of B

Overlap Test (3D)

```
void SAT( Manifold& manifold, const Hull& hullA, const Hull& hullB )
{
   FaceQuery faceQueryA = QueryFaceDirections( hullA, hullB ); // Faces A
   if ( faceQueryA.Separation > 0.0f ) // We found a separating axis

   FaceQuery faceQueryB = QueryFaceDirections( hullB, hullA ); // Faces B
   if ( faceQueryB.Separation > 0.0f ) // We found a separating axis

   EdgeQuery edgeQuery = QueryEdgeDirections( hullA, hullB ); // Edges A & B
   if ( edgeQuery.Separation > 0.0f ) // We found a separating axis

   // No separating axis found, the hull much overlap…
};
```

A simple SAT implantation then could the look like this:
- Note that this is ‘open’ function and we continue it in a second!

Hull vs Hull: Testing Face Separation



For the face tests we build a plane for each face and find the support point in the opposite normal direction on the other hull

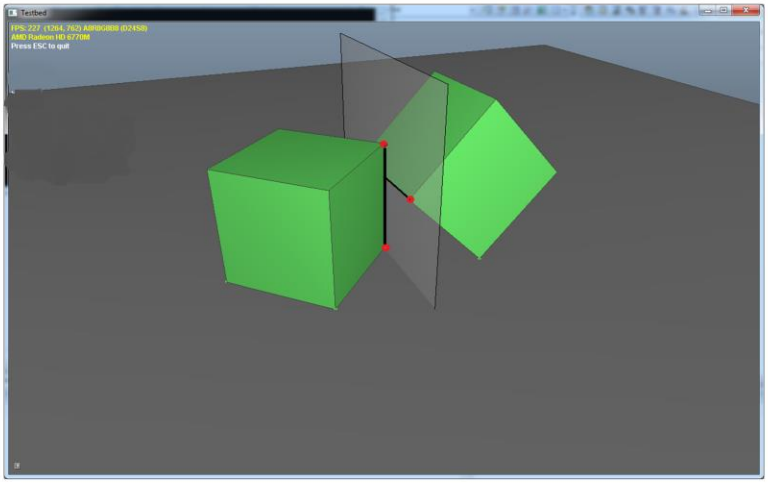
- The distance of that support point to the plane is the separation or penetration for this axis
- We perform this test for each face and keep track of the largest distance

Hull vs Hull: Testing Face Separation

```
Query QueryFaceDirections( const Hull& hullA, const Hull& hullB )
{
    for ( int index = 0; index < hullA .FaceCount; ++index )
      Plane planeA = hullA .GetPlane( index );
      Vector vertexB = hullB .GetSupport( -planeA.Normal );
      float distance= Distance( planeA, vertexB );

      // Keep track of largest distance and face index

   return largest distance and associated index of face;
};
```

In code this could look like this

Hull vs Hull: Testing Edge Separation

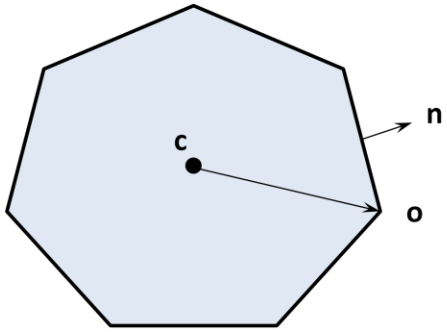


Then for each edge pair we build the cross product between the two witness edges to get the possible separating axis

- We build a plane through the edge on Hull A with the normal of the cross product
- We need to make sure that we have a consistent normal orientation pointing away from A
- Finally we check the distance as before finding the support point in the negative normal direction and compute its distance to the plane

Hull vs Hull: Normal Orientation

if *(dot*(**n**, **o** – **c**) < 0) **n** = **-n**



We need a well defined normal direction to get the sign of the distance right

- The normal must point away from hull A
- We simply compare the normal against a reference vector from the center of the hull to any of the edge vertices
- We flip the axis orientation if necessary

Hull vs Hull: Testing Edge Separation

```
EdgeQuery QueryEdgeDirection( const Hull& hullA, const Hull& hullB )
{
  for_each( Edge* pEdgeA in hullA.Edges )
    for_each ( Edge* pEdgeB in hullB.Edges )
      Vector axis = Cross( pEdgeA->GetDirection(), pEdgeB->GetDirection() );
      if ( Dot( axis, pEdgeA->Origin - hullA.Center ) < 0.0f )
        axis = -axis;

      Plane planeA = Plane( axis, pEdgeA->Origin );
      Vector vertexB = hullB .GetSupport( -planeA.Normal );
      float distance = Distance( planeA, vertexB );

      // Keep track of largest distance and edge indices

  return largest distance and associated edge indices;
};
```

In code this could look like this

Hull vs Hull: Contact Creation

- Minimizing feature defines contact type(e.g. Faces A/B or Edges)
- If the minimizing feature is a face we create a face contact
- If the minimizing feature is an edge combination we create an edge contact

If we didn't find a separating axis we can now check which feature realizes the axis of minimum penetration (e.g. Faces A/B or Edges)

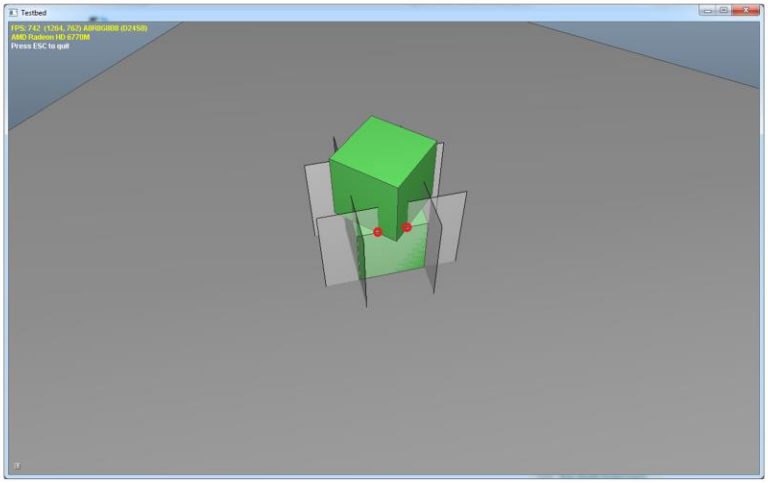
- If the minimizing feature is a face we create a face contact
- If the minimizing feature is an edge combination we create an edge contact

Hull vs Hull: Contact Creation

```
void SAT(Manifold& manifold, const Hull& hullA, const Hull& hullB )
{
   // …No separating axis found, the hulls much overlap
   bool bIsFaceContactA = faceQueryA.Separation > edgeQuery.Separation;
   bool bIsFaceContactB = faceQueryB.Separation > edgeQuery.Separation;
   if ( bIsFaceContactA && bIsFaceContactB )
      CreateFaceContact( manifold, faceQueryA, hullA, faceQueryB, hullB );
   else
      CreateEdgeContact( manifold, edgeQuery, hullA, hullB );
};
```

We continue the implementation of our SAT function. On termination all query results are negative.
- The axis of minimum penetration is realized by the feature with the smallest penetration!

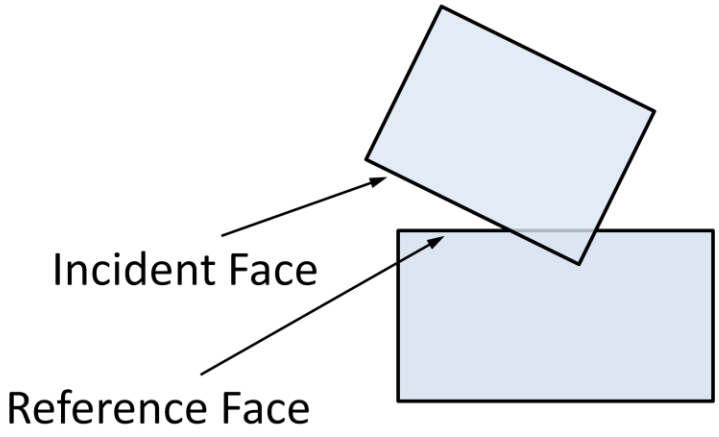
Hull vs Hull: Face Contact



If the axis of minimum penetration is a face normal we build a face contact:

- Conceptually we clip the incident face on the other hull against the side planes of the minimizing face.

Hull vs Hull: Face Contact



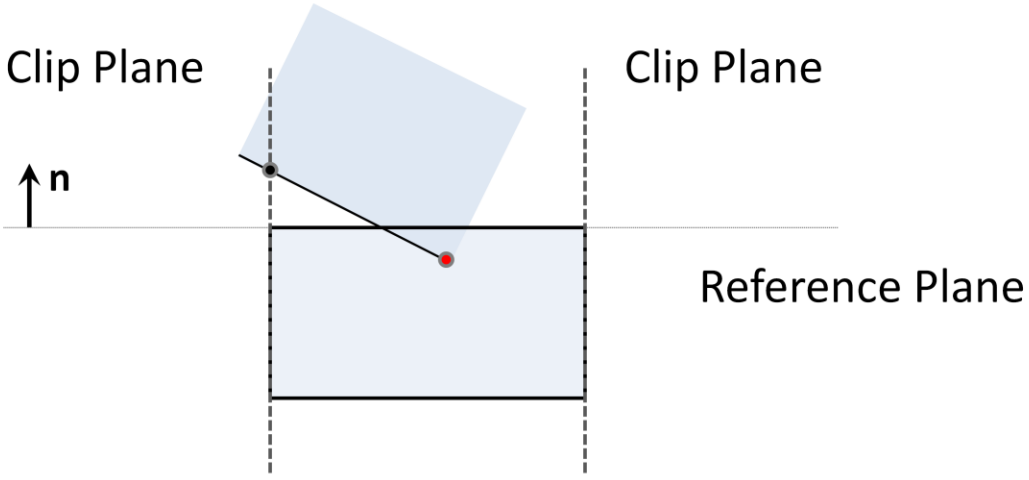
What does this mean?

- We used the SAT to find the axis of minimum penetration
- We call the minimizing face the **reference** face
- Next we identify the **incident** face
- The incident face is simply the most anti-parallel face on the other hull

Implementation hint:

- To find the incident face simply iterate all faces on the other hull and compute the dot product of each face normal with the normal of the reference face. The face with the smallest dot product defines the incident face!

Hull vs Hull: Face Contact Clipping

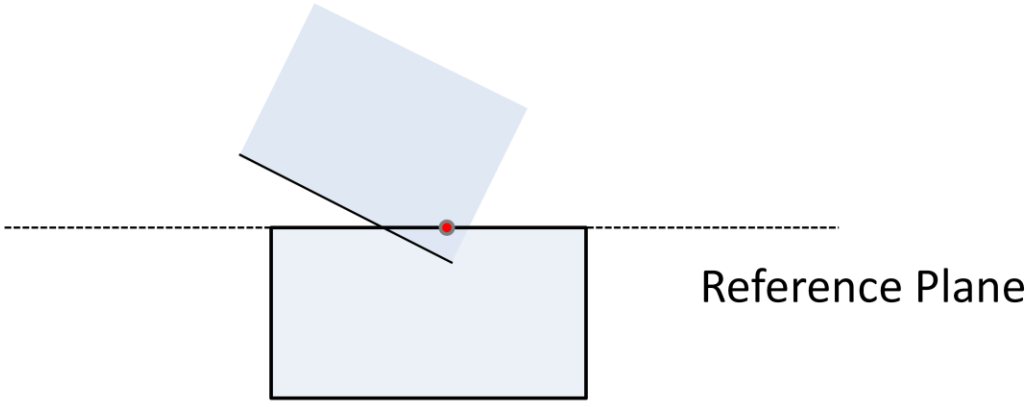


Next we setup ourselves for clipping:

- We clip the incident face polygon against the side planes of the reference face
- Then we keep all points below the reference face
- We can essentially use any polygon clipping algorithm for this (e.g. I use Sutherland Hodgman clipping)

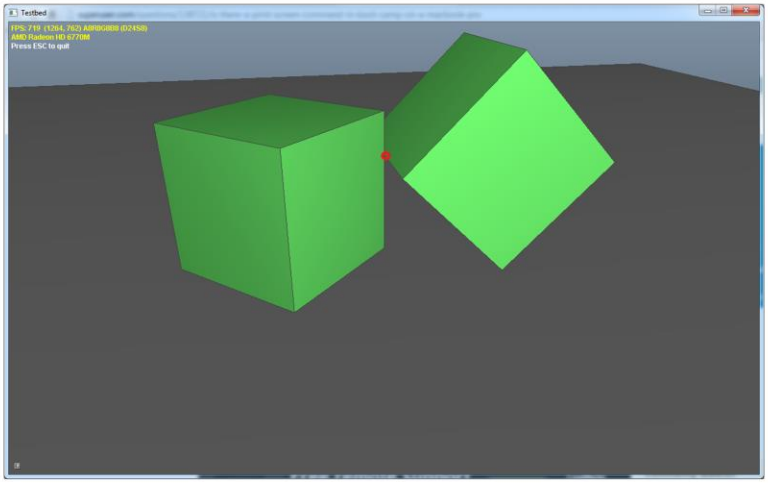
Note that we don't clip against the reference face since the additional points will not add to the stability of the manifold.

Hull vs Hull: Coherent Points



Finally we move the contact points onto the reference face:
- This helps with coherence and (as we will see later) with contact point reduction where we assume that the contact points are in common plane

Hull vs Hull: Edge Contact



If the axis of minimum penetration is realized by an edge pair the contact point must be the closest points between the minimizing edges!

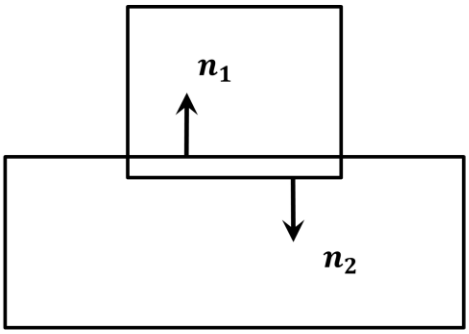
Hull vs Hull: Edge Contact

- For an edge contact we simply compute the closest points between the two witness edges
- Then we choose the contact point in the middle of the closest points
- We construct the normal the same way as we did for testing the edge separation

Creating an edge contact is rather trivial:

- For an edge contact we simply compute the closest points between the two witness edges
- Then we choose the contact point in the middle of the closest points
- We construct the normal the same way as we did for testing the edge separation (e.g. build the cross product between the edges and check the orientation)

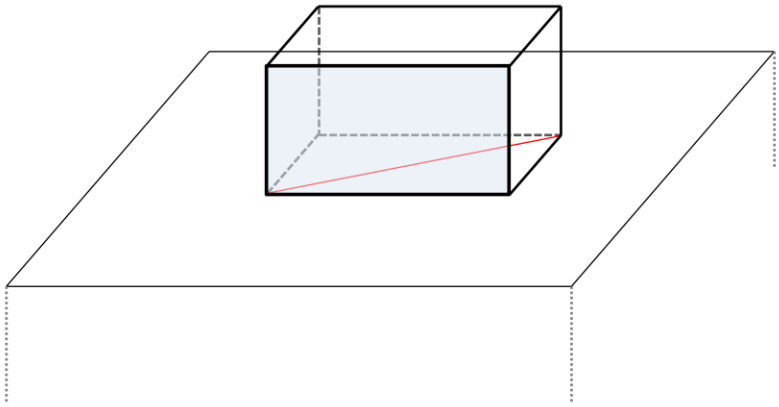
Hull vs Hull: Feature Flip-Flop



Often many axes return the same penetration. Which normal is the axis of minimum penetration in the above example?

- The solver likes coherent contact points as this improves stability, so we like to build our contact points from the same features between frames
- The solution is simple and we apply a bias (or weighting) to prefer face contacts over edge contacts and one face axis over another

Hull vs Hull: Coplanar Faces



The clipping algorithm I showed you will not find the full manifold if coplanar faces are not merged correctly:

- Hence coplanar faces will introduce instability into your solver
- In the above example we would get an unstable triangle manifold since the clipping code only sees a part of the incident face
- In the worst case you could even get an empty manifold (e.g. for more complicated examples with many smaller coplanar faces)

As a rule of thumb we want large faces and aim for sharp edges between adjacent faces (e.g. minimum angles of 10 degrees between faces)

- Nearly coplanar faces will introduce fuzziness into your solver since it is not clear which face to rest on!

Hull vs Hull: Temporal Coherence

- An important optimization is to store the result from the last frame
- If we detected a separating axis we try this first. Chances are high it will still be a separating axis in the current frame and we can exit early!
- If we detected overlap we try to rebuild the contact from the last touching features again and only run the full algorithm if this fails!

An important optimization of this technique is to utilize temporal coherence (e.g. store the result from the last frame)

- If we detected a separating axis in the last frame we try this first in the current frame. Chances are high it will still be a separating axis and we have an early out!
- If we detected overlap in the last frame we try to rebuild the contact from the last features again. Chance are high they still realize the contact manifold and we can skip the whole SAT!

As a side note:

- You should expect a speed-up by an order of magnitude or more if implemented correctly! So this is significant in practice and not just some theoretical idea!
- Contact performance is not about SAT vs GJK, but to not call any of those geometric algorithms at all if possible!
- Sergiy will talk about this later in the optimization talk, but the general idea is: If you don’t need to do it, don’t do it!

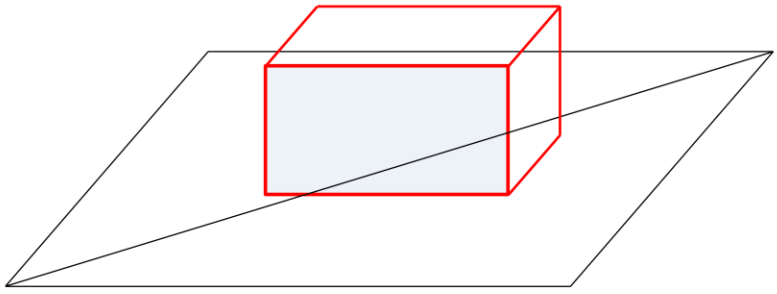
The Collision Table: Mesh Contact

	Sphere	Capsule	Hull	Mesh
Sphere				X
Capsule				X
Hull				X
Mesh				

Finally we look into mesh contacts

- As opposed to convex contacts there can now be many manifolds for each triangle the convex shape is touching
- Luckily we can reuse all of the ideas we used to build the manifolds between the convex shapes earlier

Mesh Contact: Collecting Triangles

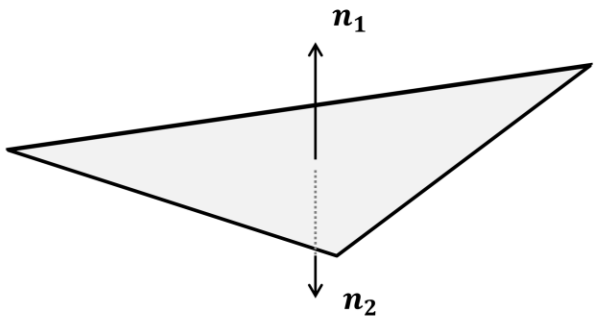


The first step for mesh contacts is a mid-phase to collect all triangles the convex shapes can potentially overlap.

- The can be easily achieved by a box query against the triangle mesh
- Triangles are usually sorted in a tree for fast AABB queries so this is a fast test
- Then for each potential pair (convex, triangle) we need to compute a contact manifold

Luckily there is really nothing new here and we will reuse all the ideas we developed for the convex contacts!

Mesh Contact: Triangles as Hulls



The simple trick for the mesh contacts is to realize that we can treat each triangle as a convex hull

- In order to make this work we treat every triangle as double sided
- Now we can reuse all the functions we developed before and no special code is needed

Is this really it?

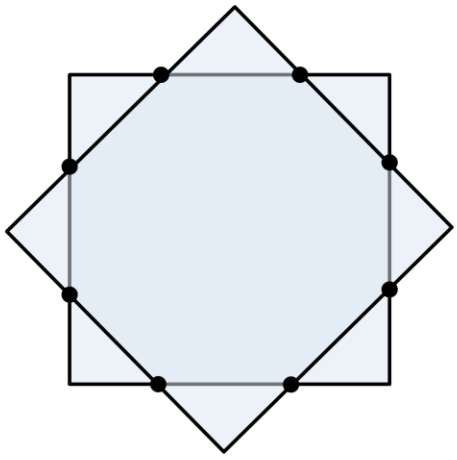
- Well, this talk will get you started and put you on the right track, but mesh contacts are really a talk on their own and there are many nifty details to consider
- P. Terdiman actually just published a nice paper on mesh contacts which is a great introduction into this problem
- I added a link to the reference section of this talk for your convenience

Outline

- Introduction
- Basic Strategies
- Solving the collision table
- **Contact optimizations**
- Code examples

So now that we know how to compute stable and robust manifolds let's have a look how to bring the contact point count down if a manifold is too complex!

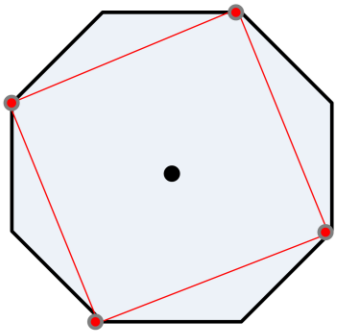
Contact Point Reduction



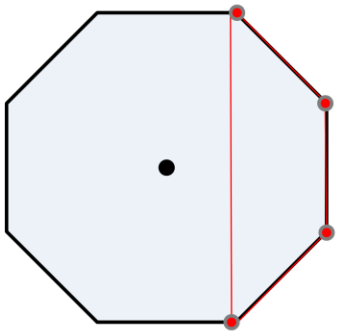
The above slide shows the top-down view of two boxes resting on each other

- The methods you learned today can produce many contact points (e.g. 8 in the above example)
- For performance reasons we don’t want to send all contact points to the solver and limit ourselves to a maximum of four points!
- Usually four contact points are efficient for fast, stable and robust contact simulation

Contact Point Reduction



4 Good Points

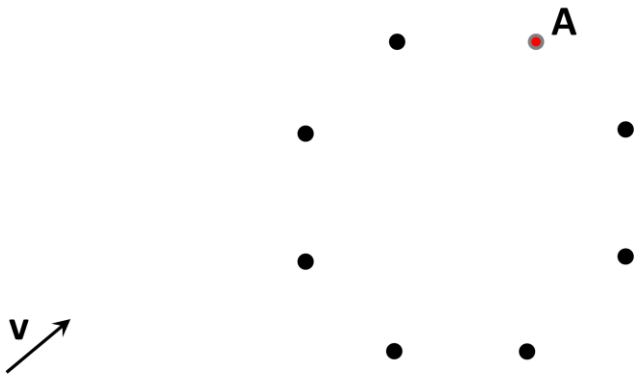


4 Bad Points

Our goal is to find the combination of four contact points with the largest area that doesn't destabilize the manifold!

- The slide shows a set of four good and four bad points to illustrate the idea!

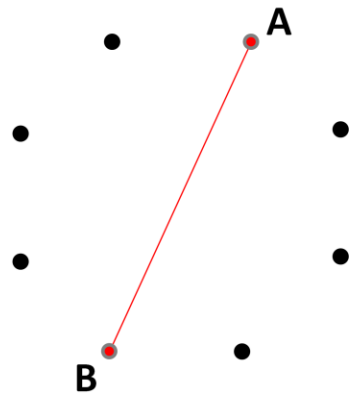
Contact Point Reduction: 1st Point



In order to reduce our manifold we will use an iterative approach.

- Remember that we assume here that all contact points are in the same plane!
- First we need to choose an initial point.
- We can essentially start with any point, but we want to make sure that it is relatively coherent between frames
- So a better solution than just using the point at index zero (which can change between frames) we can query a support point in fixed direction in the contact plane
- If we are concerned about CCD and continuous contact solving we want to start with the deepest point and guarantee it is in our reduced contact set
- For the sake of simplicity assume we start with the red point in our example here!

Contact Point Reduction: 2nd Point

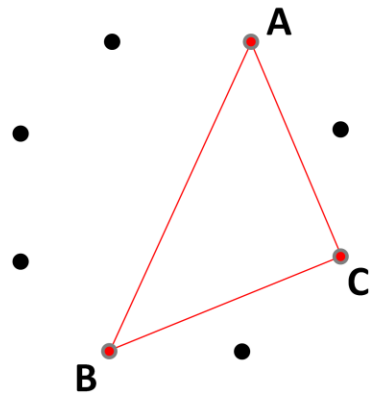


$$Distance = |\boldsymbol{B} - \boldsymbol{A}|$$

After we identified our initial point we simply search the farthest point from the starting point

- In practice we compute the squared distance of course

Contact Point Reduction: 3rd Point

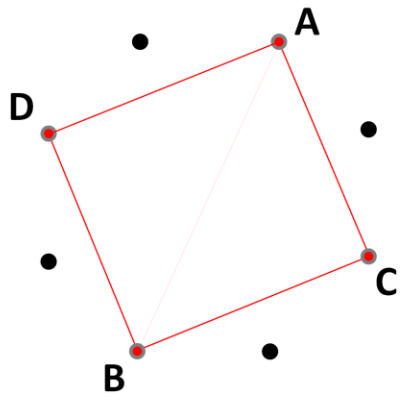


$$Area = 0.5 \cdot (\boldsymbol{CA} \times \boldsymbol{CB}) \cdot \boldsymbol{n}$$

For the third point we find the point which maximizes the area

- For each remaining point we build a triangle with our current edge and compute the area
- We can use the cross product to compute the area
- Note that we use the dot product with the face normal here and not the absolute value
- This saves us a sqrt() and also returns a signed distance which we can use to detect the winding of the triangle

Contact Point Reduction: 4th Point (1)



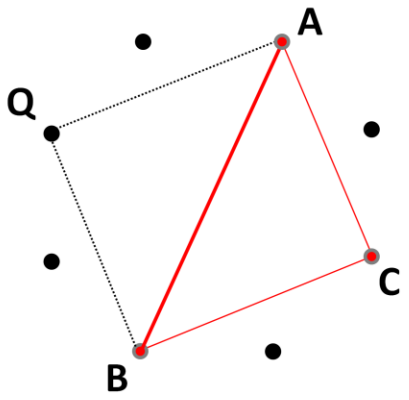
Finally we want to add a fourth point which adds the largest area to our triangle

- We use the same formula as before and check the remaining points with each edge of the triangle.
- Note that we only consider triangles with a negative area as we are only interested in points on the outside of the edge in consideration

Note:

- Conceptually we are computing barycentric coordinates and keep the point with the smallest barycentric coordinate!

Contact Point Reduction: 4th Point (2)

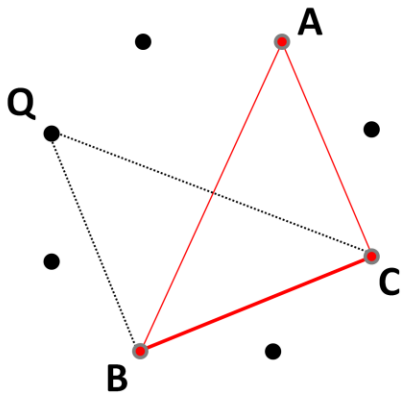


$$Area = 0.5 \cdot (\boldsymbol{QA} \times \boldsymbol{QB}) \cdot \boldsymbol{n}$$

Let's quickly check why this works. We test point Q with edge AB:

- The area of the triangle ABQ is negative (CW winding) and we keep track of it
- The negative sign simply means that Q is on the other side of AB as C which is what we want!

Contact Point Reduction: 4th Point (3)

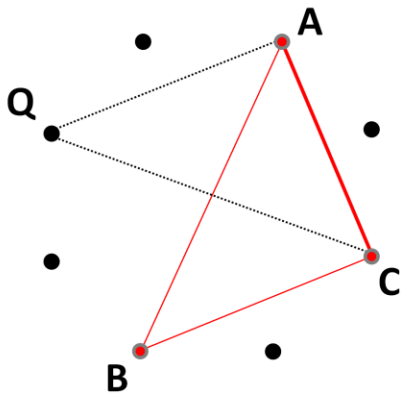


$$Area = 0.5 \cdot (\boldsymbol{QB} \times \boldsymbol{QC}) \cdot \boldsymbol{n}$$

We continue with the next edge and test point Q with edge BC

- The area of the triangle BCQ is positive (CCW winding) and we skip it
- The positive sign means that Q is on the same side of BC as A which we are not interested in!

Contact Point Reduction: 4th Point (4)



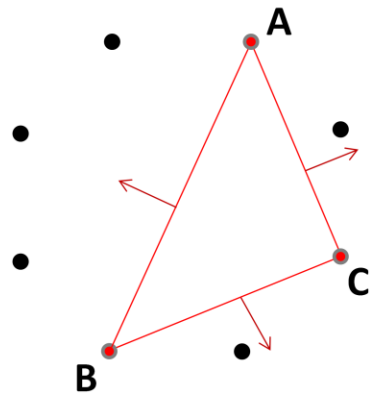
$$Area = 0.5 \cdot (\boldsymbol{QC} \times \boldsymbol{QA}) \cdot \boldsymbol{n}$$

We continue to the final edge and test point Q with edge CA

- The area of the triangle CAQ is positive again (CCW winding) and we skip it as well
- Again the positive sign means that Q is on the same side of AC as B which we ignore!

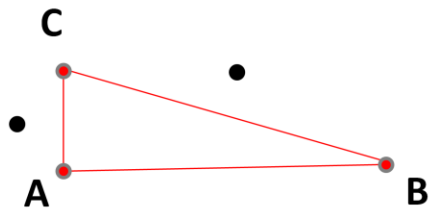
We repeat this algorithm with each remaining point and keep the point which adds the largest negative area to the current triangle!

Contact Point Reduction: 4th Point (5)



When computing the fourth point your intuition might suggest to use the edge normals to find the furthest point from the triangle
- In the above example this would indeed return the right answer, but let's look at another example

Contact Point Reduction: 4th Point (6)



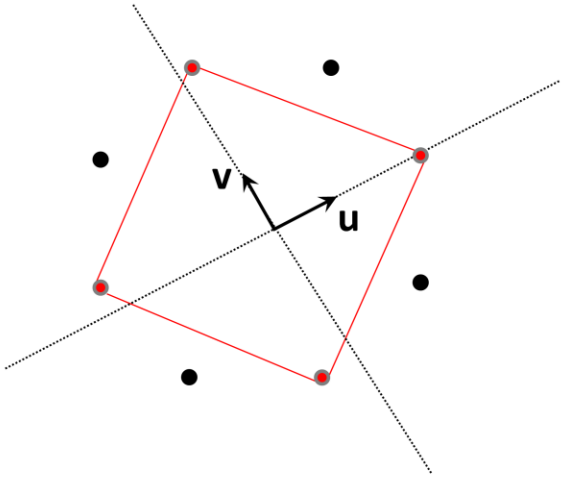
In this example the remaining points have about the same distance from their closest edges

- Finding the furthest point from an edge does not take the edge length into account

So the simple rule is:

- Maximize area, not distance!

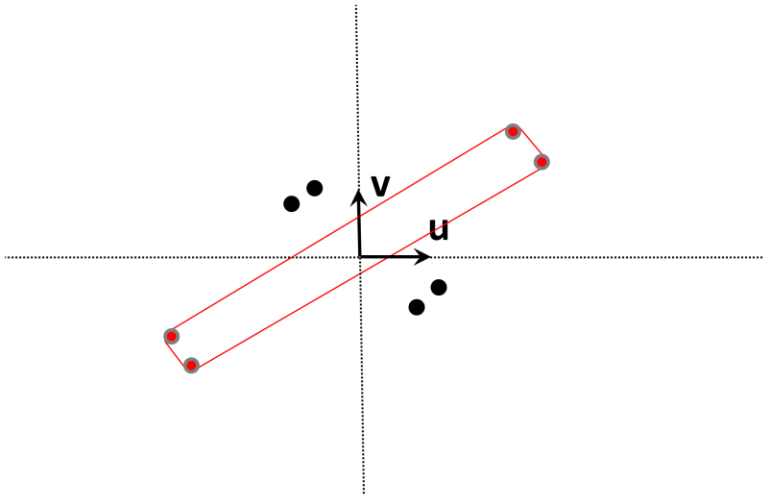
Contact Point Reduction: AABB (1)



An alternative to the algorithm presented before is to simply find the extreme points along two orthonormal axes in the contact plane (think AABB):

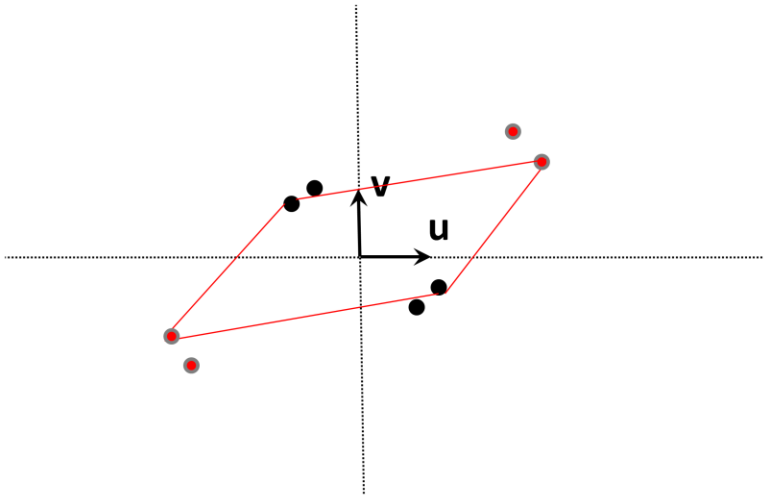
- We first build an orthonormal basis (u,v) with the manifold normal
- And then we find the extreme points along these axes
- Note that the result is now dependent on the choice of your axes!

Contact Point Reduction: AABB (2)



This approach can create wrong reduced manifolds though as the above example can hopefully illustrate.

Contact Point Reduction: AABB (32)



What we really want in this case is something more along the lines like this!

Outline

- Introduction
- Basic Strategies
- Solving the collision table
- Contact optimizations
- **Code examples**

So before we close let’s look at some code examples how a basic collision pipeline could be implemented.

Examples: Pair Management

```
void rnWorld::BuildNewContacts( void )
{
   if ( Shape1->GetType() != RN_MESH_SHAPE && Shape2->GetType() != RN_MESH_SHAPE )
   {
      rnConvexContact* Contact = new rnConvexContact( this, Shape1, Shape2 );
      mContactList.PushBack( Contact );
   }
   else
   {
      if ( Shape1->GetType() != RN_MESH_SHAPE || Shape2->GetType() != RN_MESH_SHAPE )
      {
         rnMeshContact* Contact = new rnMeshContact( this, Shape1, Shape2 );
         mContactList.PushBack( Contact );
      }
      else
         // No contact point is created if both shapes are meshes!
   }
}
```

When we detect a new pair we check the shape types

- If both shapes are convex (e.g. sphere, capsule, or hull) we create a convex contact
- If any of the two shapes is a mesh we create a mesh contact
- If both shapes are a mesh we ignore the pair, but report a warning to the user

Examples: Convex Contact

```
void rnCollide( rnManifold& Manifold, rnTransform& XForm1, rnShape* Shape1, rnTransform& XForm2, rnShape* Shape2 )
{
  typedef void (*rnCollideFunc)( rnManifold&, rnTransform&, rnShape*, rnTransform&, rnShape* );
  static const rnCollideFunc CollisionMatrix[ RN_SHAPE_COUNT ][ RN_SHAPE_COUNT ] =
  {
    { rnCollideSpheres, rnCollideSphereCapsule,  rnCollideSphereHull,   NULL },
    { NULL,             rnCollideCapsules,       rnCollideCapsuleHull,  NULL },
    { NULL,             NULL,                    rnCollideHulls,        NULL },
    { NULL,             NULL,                    NULL,                  NULL }
  };

  RN_ASSERT( Shape1->GetType() <= Shape2->GetType() );
  rnCollideFunc Collide = CollisionMatrix[ Shape1->GetType() ][ Shape2->GetType() ];
  Collide( Manifold, XForm1, Shape1, XForm2, Shape2 );
}
```

In order to invoke the correct collision routine based on the shape type we simply use a two dimensional array of function pointers

- Note that the lower collision table does not need to be implemented due to symmetry!
- As the order of the shapes in a pair is arbitrary we simply make sure that the first shape type is always larger than the second (e.g. in the constructor of the convex contact)

Examples: Mesh Contact

```
void rnCollide( rnManifold& Manifold, rnTransform& XForm1, rnShape* Shape1, rnTransform& XForm2, rnShape* Shape2 )
{
  RN_ASSERT( Shape1->GetType()  != RN_MESH_SHAPE && Shape2->GetType()  == RN_MESH_SHAPE );
  switch ( Shape1->GetType() )
  {
    case RN_SPHERE_SHAPE:
      rnCollideSphereMesh( Manifold, XForm1, Shape1, XForm2, Shape2 ); break;
    case RN_CAPSULE_SHAPE:
      rnCollideCapsuleMesh( Manifold, XForm1, Shape1, XForm2, Shape2 ); break;
    case RN_HULL_SHAPE:
      rnCollideHullMesh( Manifold, XForm1, Shape1, XForm2, Shape2 ); break;
    default:
      break;
  }
}
```

Dispatching the correct collision function for a mesh contact is even easier!

- The 2D matrix becomes a simple linear array in this case which we simple handle in a switch block
- Note that this is really easy and you don't want to implement difficult dispatch mechanisms with unnecessary indirections for these problems here!!!

Thank You!

This closes the talk. Thank you!

Questions?

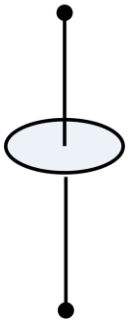
Hopefully we have still some time left for questions! Please step forward to the microphones and tell us your affiliation!

REMINDER:
- Repeat questions for audience!

References

- C. Ericson: “Real-Time Collision Detection”
- G. v. d. Bergen: “Collision Detection in Interactive 3D Environments”
- S. Migdalsky: “SAT in Narrow Phase and Contact Manifold Construction”
- E. Catto: “Contact Manifolds” (GDC 2007)
- E. Catto: “Computing Distance with GJK” (GDC 2010)
- E. Coumans: “Contact Generation” (GDC 2010)
- V. Robert: “A Different Approach for Continuous Physics” (GDC 2012)
- D. Gregorius: “The Separating Axis Test between Convex Polyhedra” (GDC 2013)
- D. Gregorius: “Implementing Quickhull” (GDC 2014)
- P. Terdiman: “Contact Generation for Meshes” (www.codercorner.com/blog)

Appendix: Gauss Map for Capsule



The Gauss Map for the capsule is simply a ring:

- The IsMinkowskiFace() test becomes a simple plane test qualifying the two hull normals against a plane of the ring
- Please refer to my GDC 2013 presentation for details!