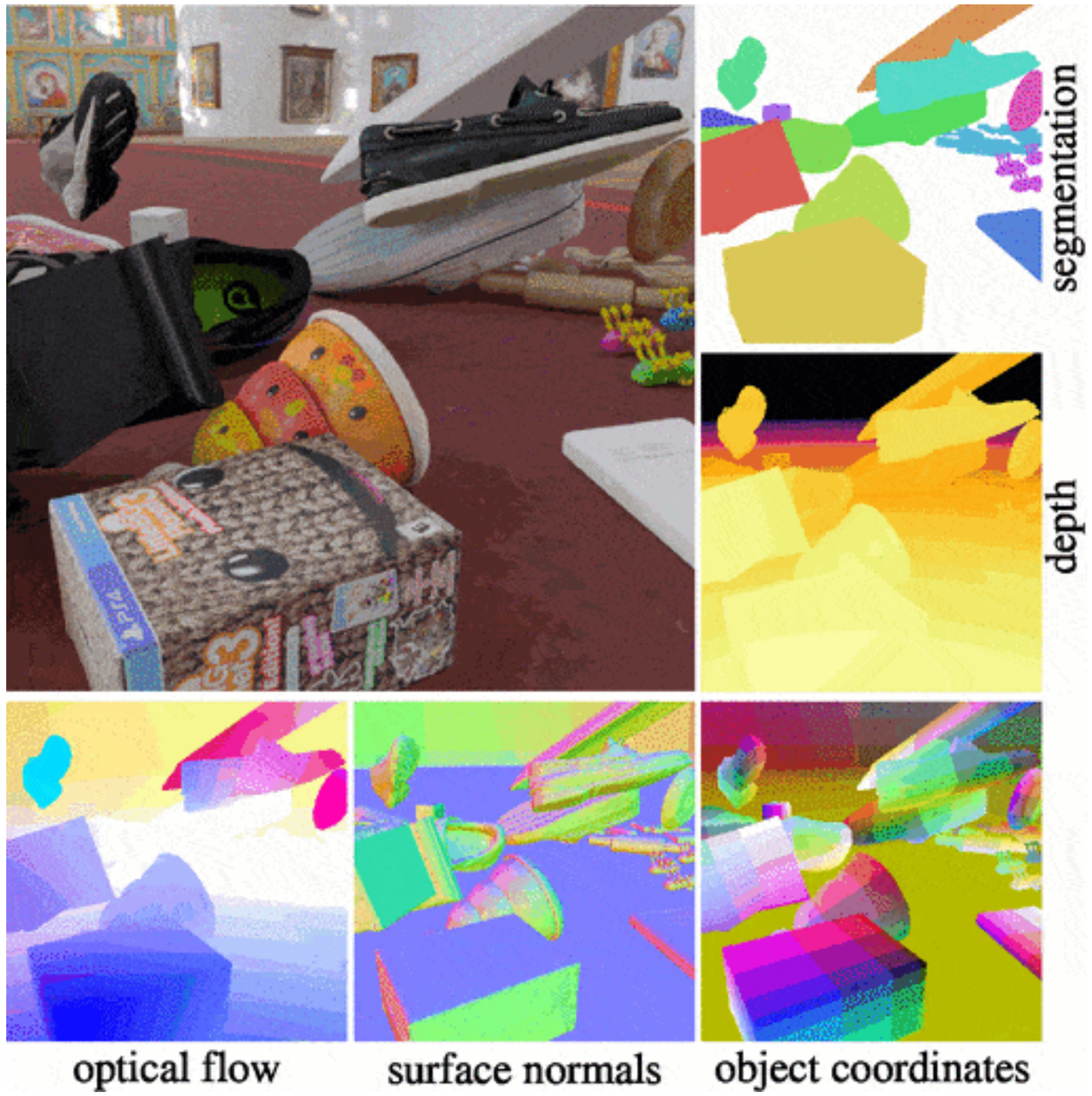
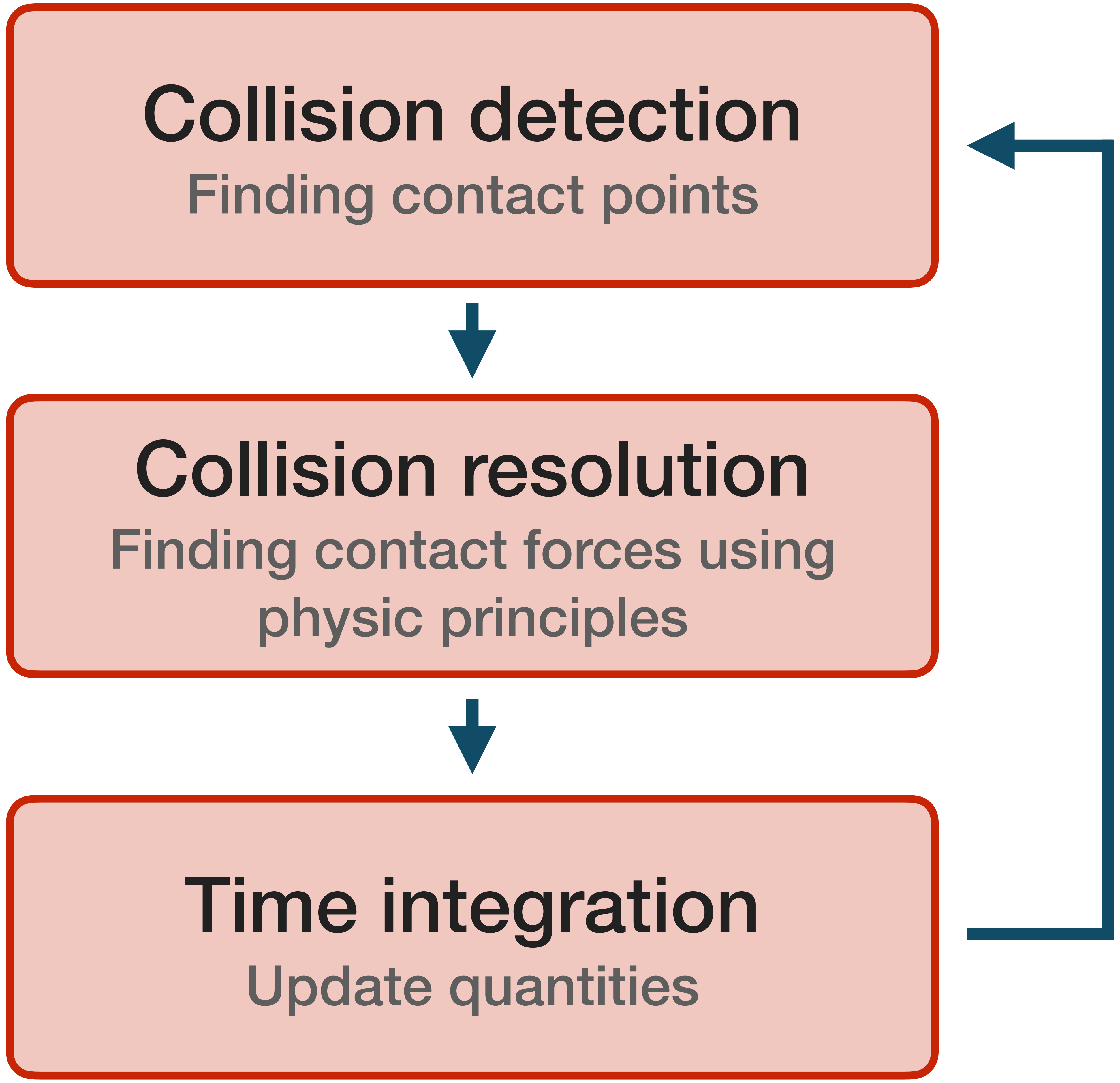
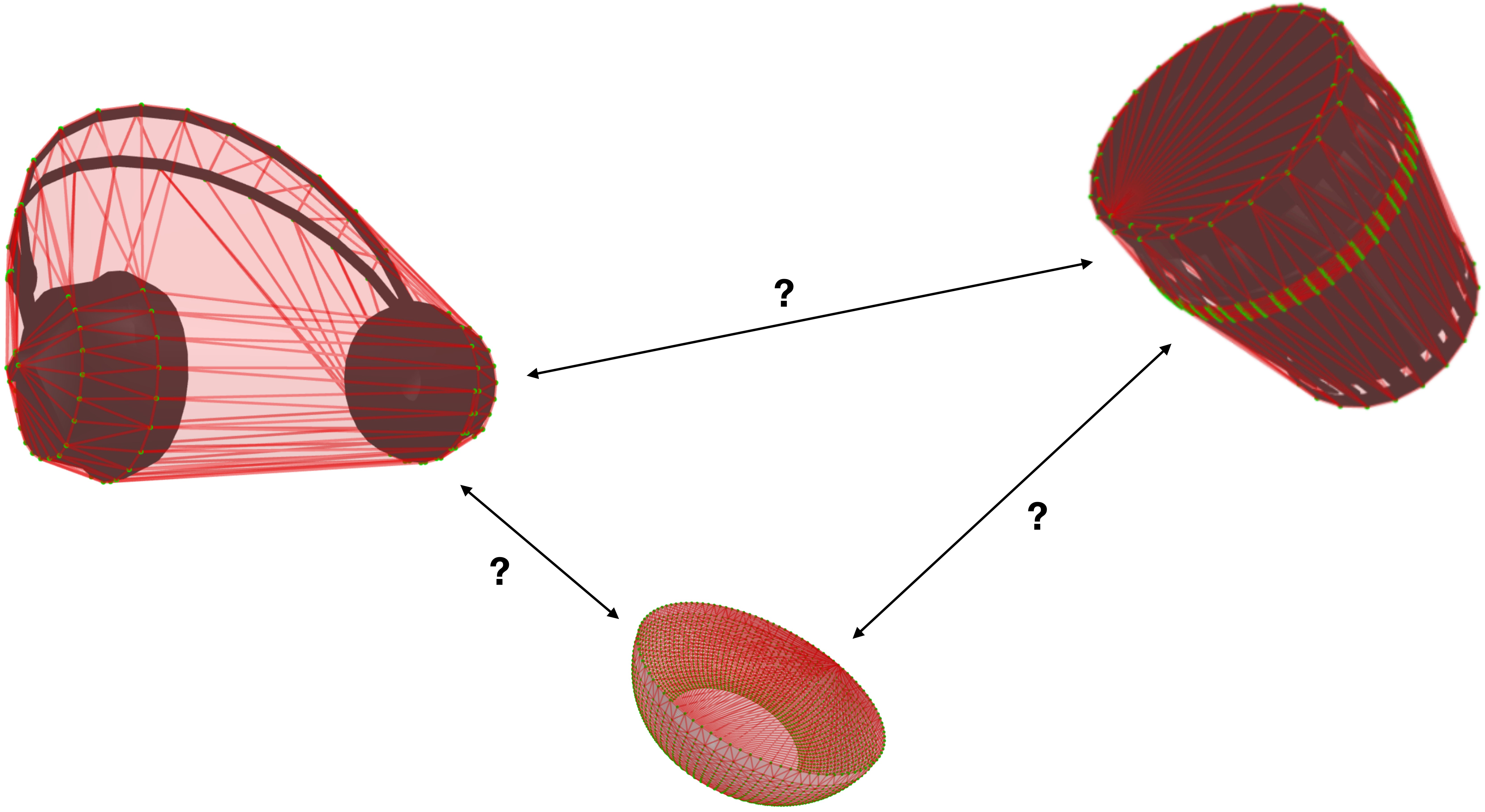
Collision Detection for Rigid-Body Physics Simulation

What is a physics simulator?

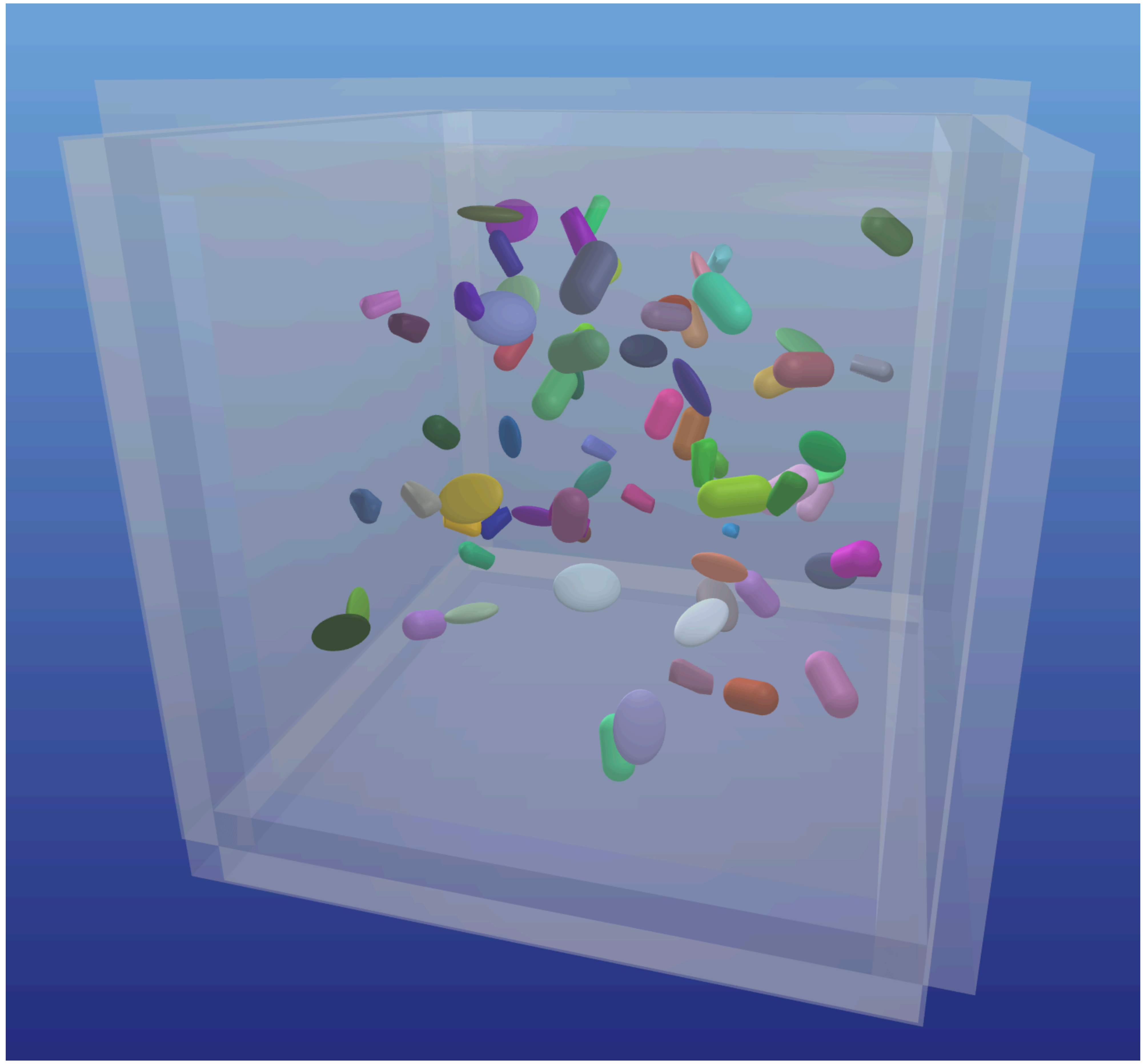


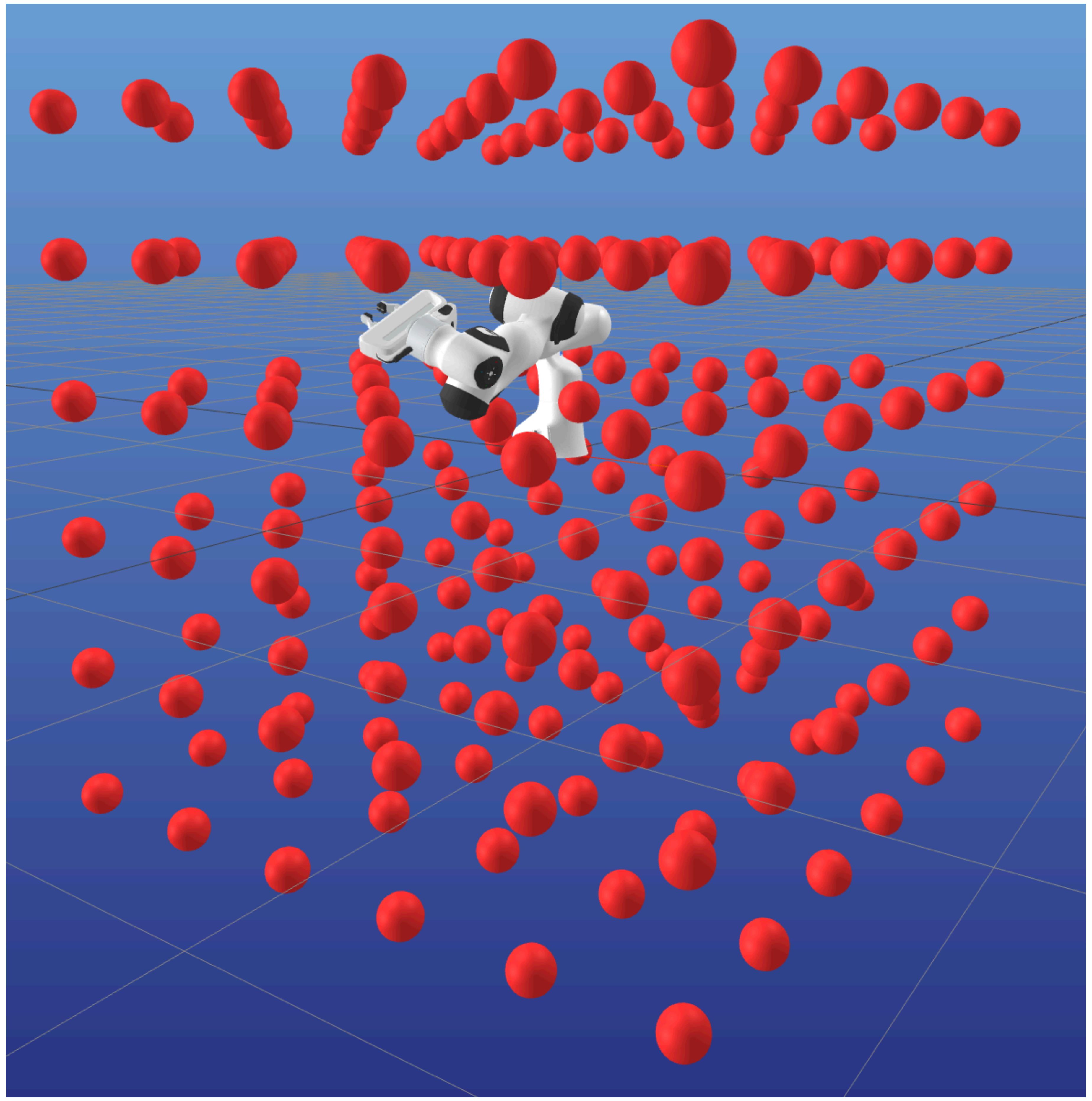


What is collision detection?

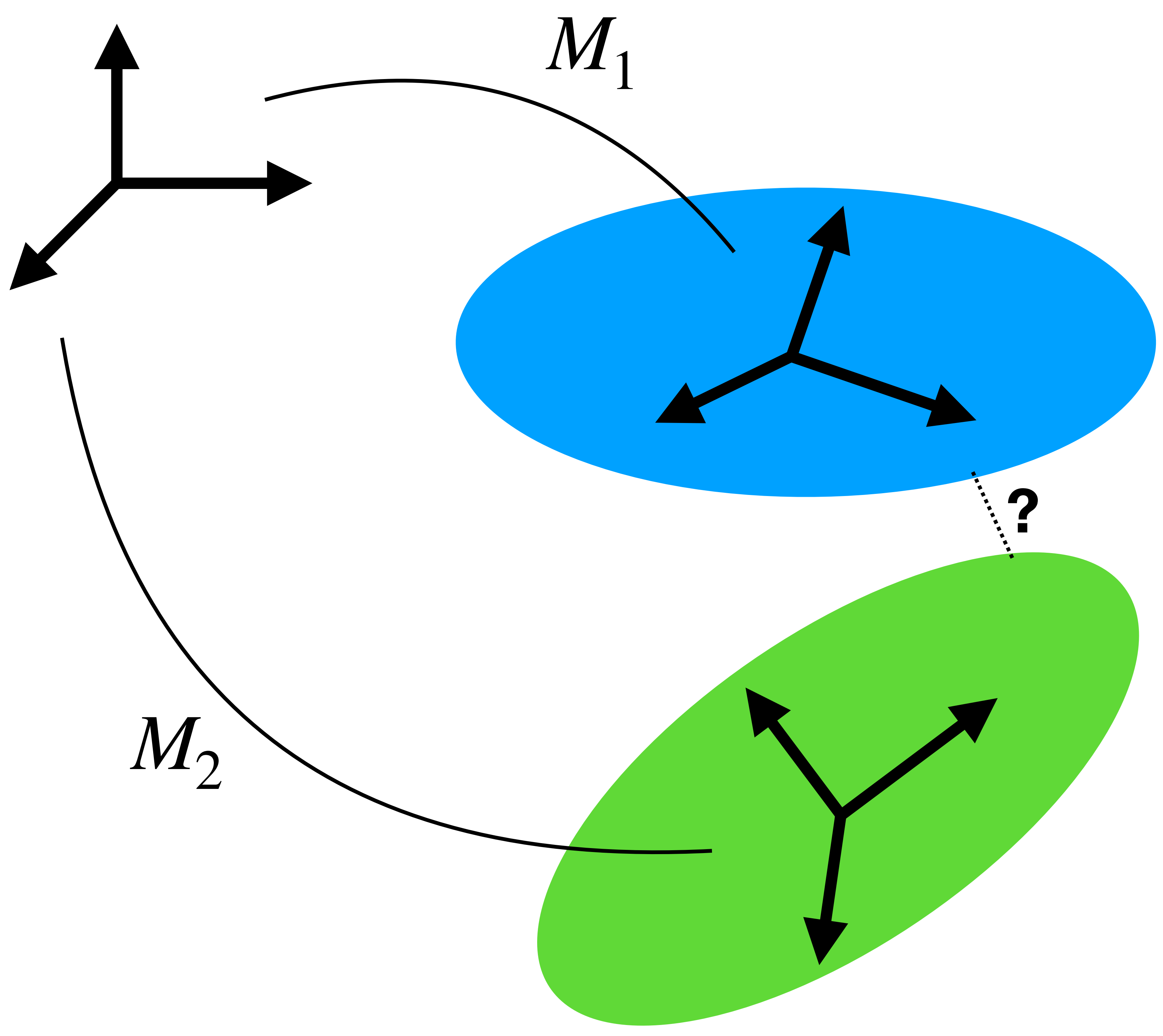


What is collision detection?





HPP-FCL tutorial



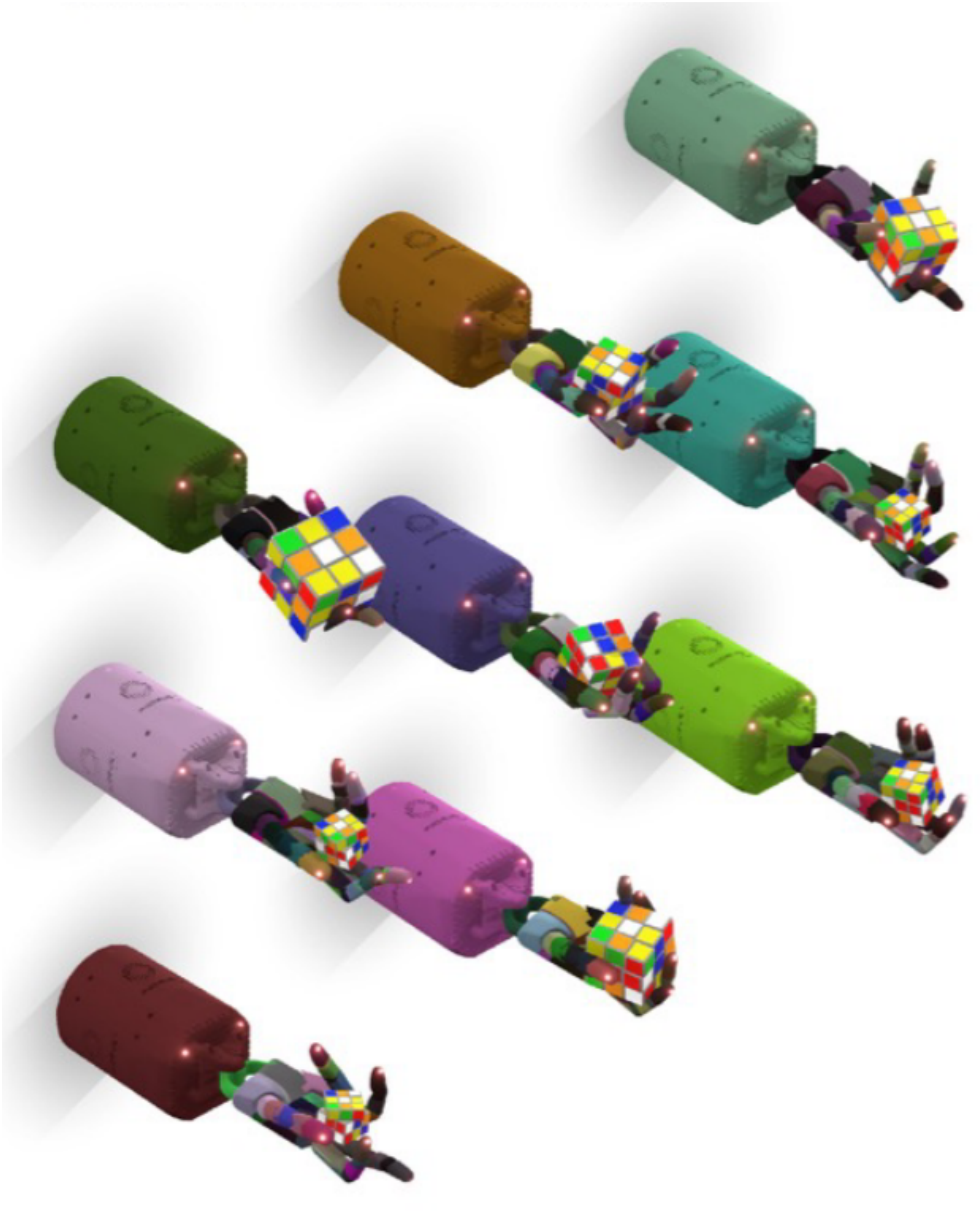
In the terminal:

```
λ conda install hpp-fcl
```

In a python script:

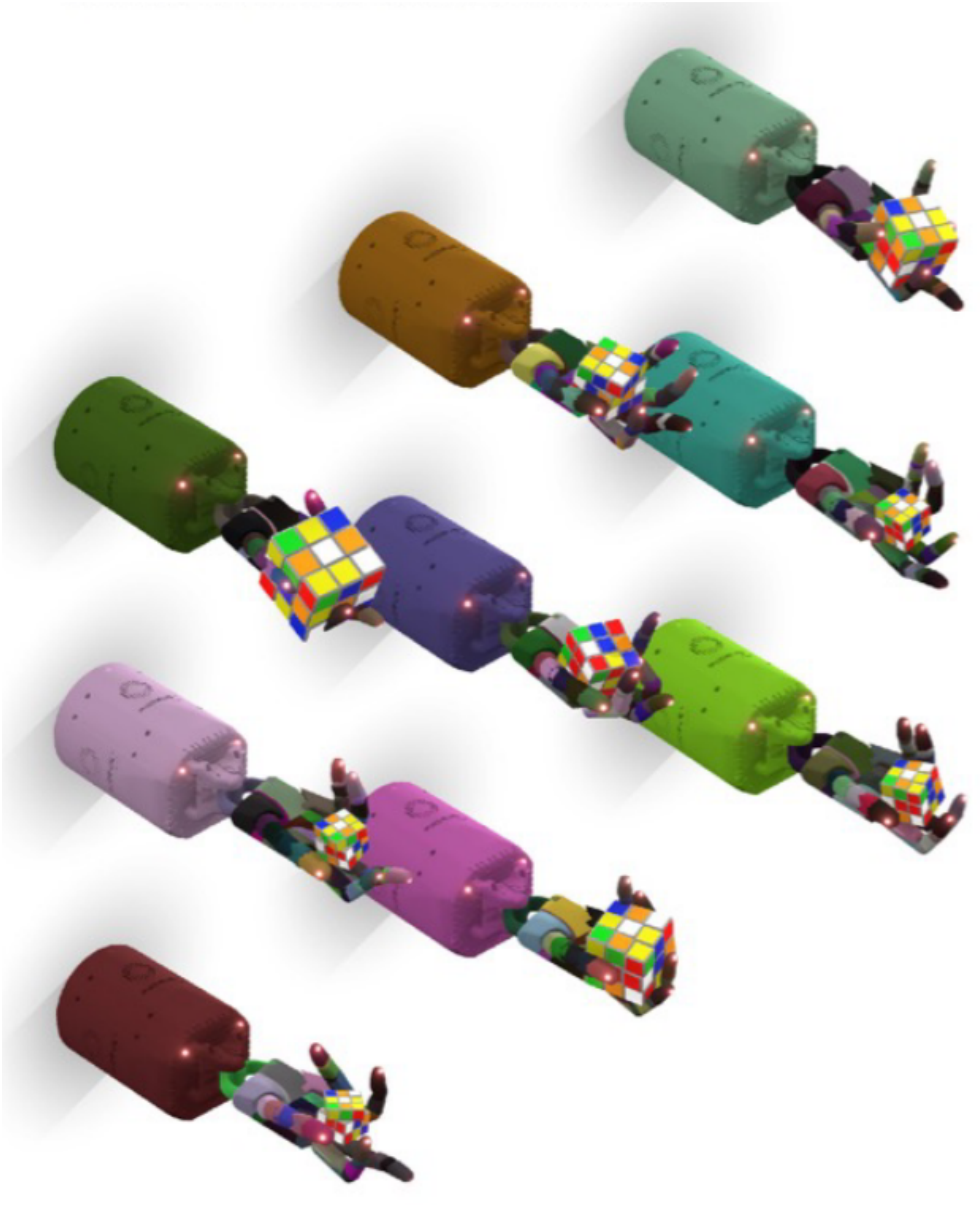
```python
import hppfcl
import pinocchio as pin

shape1 = hppfcl.Ellipsoid(np.array([0.2, 0.3, 0.1]))
M1 = pin.SE3.Random()

shape2 = hppfcl.Ellipsoid(np.array([0.4, 0.2, 0.5]))
M2 = pin.SE3.Random()

req = hppfcl.CollisionRequest()
res = hppfcl.CollisionResult()

is_collision = hppfcl.collide(shape1, M1, shape2, M2, req, res)
```

Collision detection is a computational bottleneck



- ABA: ~ 1-10 micro-seconds
- Collision detection timing for 1 pair of objects: ~ 1-10 micro-seconds
- Contact solving: ~1-10 micro-seconds

Collision detection is a computational bottleneck



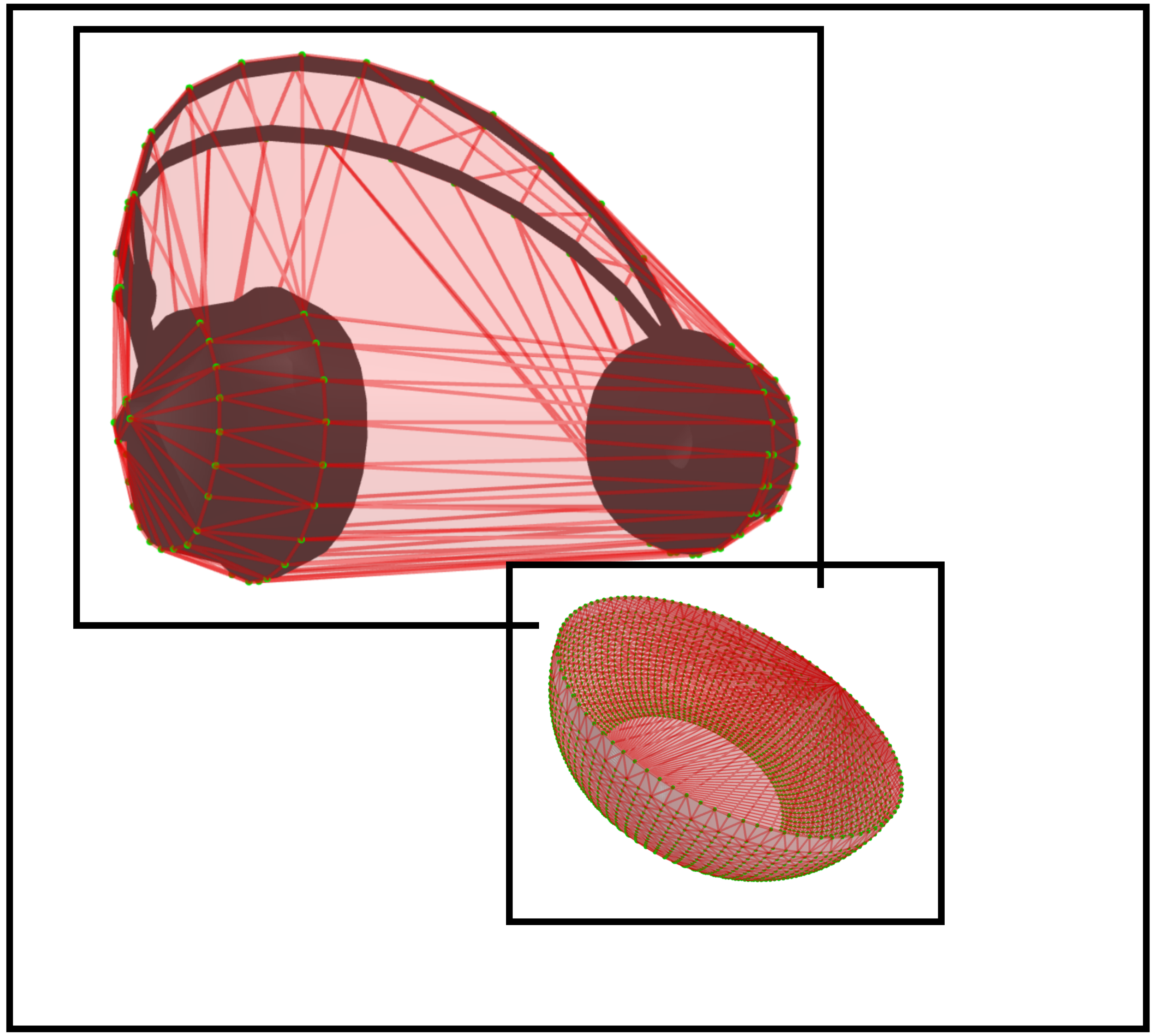
- ABA: ~ 1-10 micro-seconds
- Collision detection timing for 1 pair of objects: ~ 1-10 micro-seconds
- Contact solving: ~1-10 micro-seconds

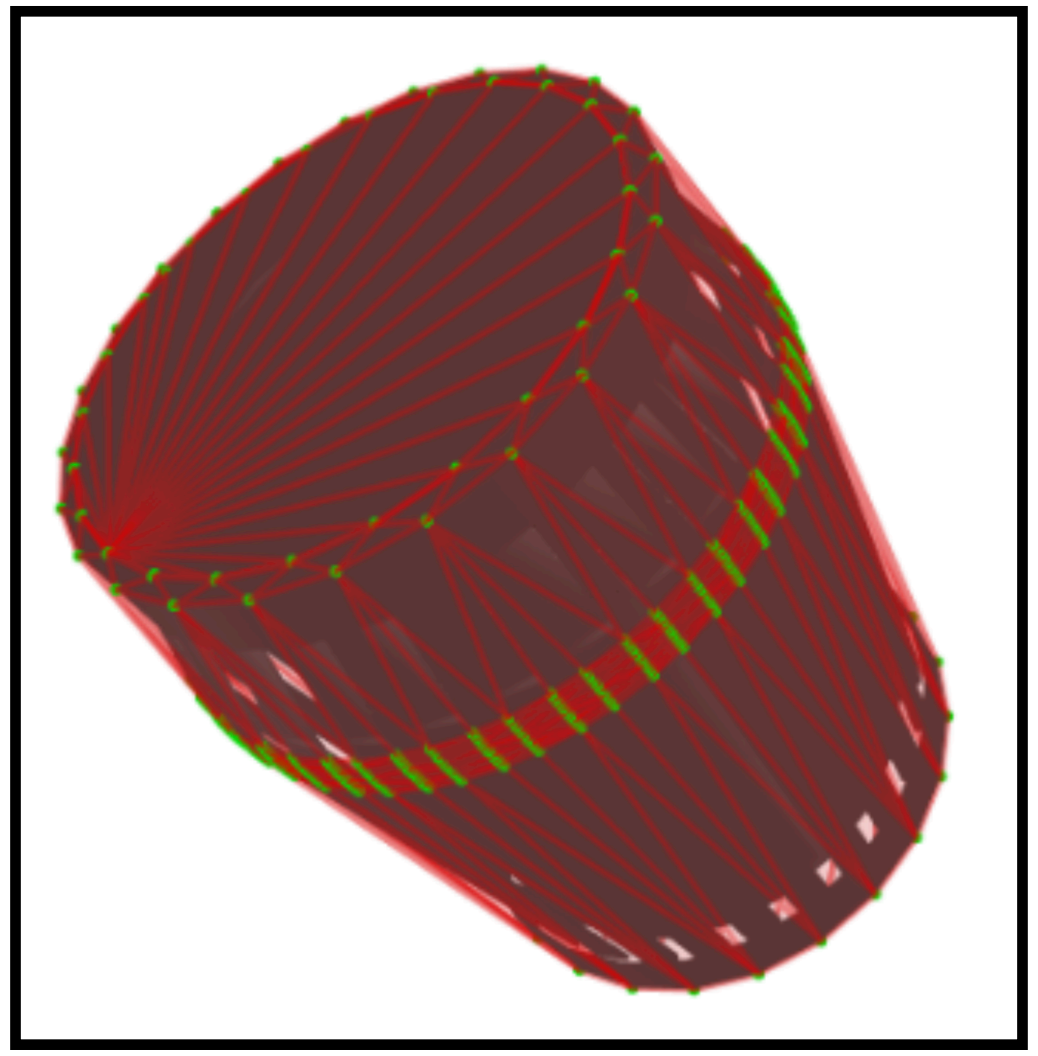
N objects in a scene
-> O(N x N) possible collision pairs!

Part I - The Broad Phase

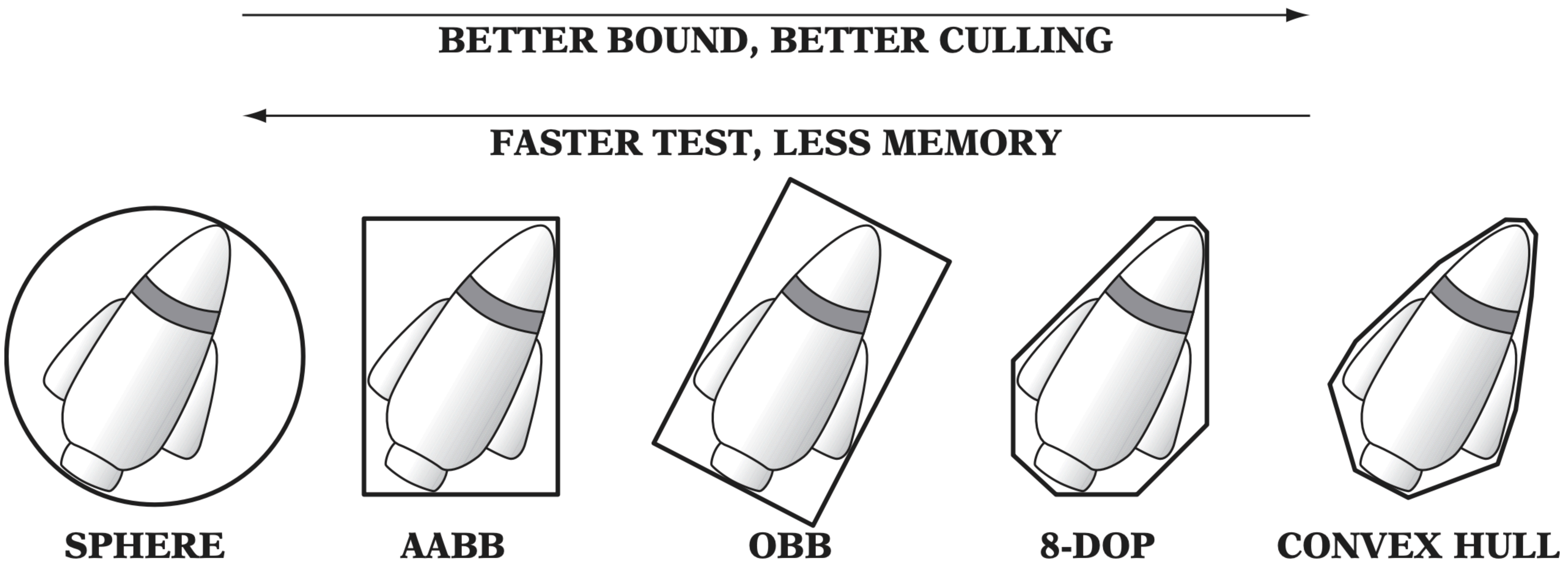
Bounding volumes

- Use bounding volumes (BVs) to prune collisions
- Only check overlapping BVs



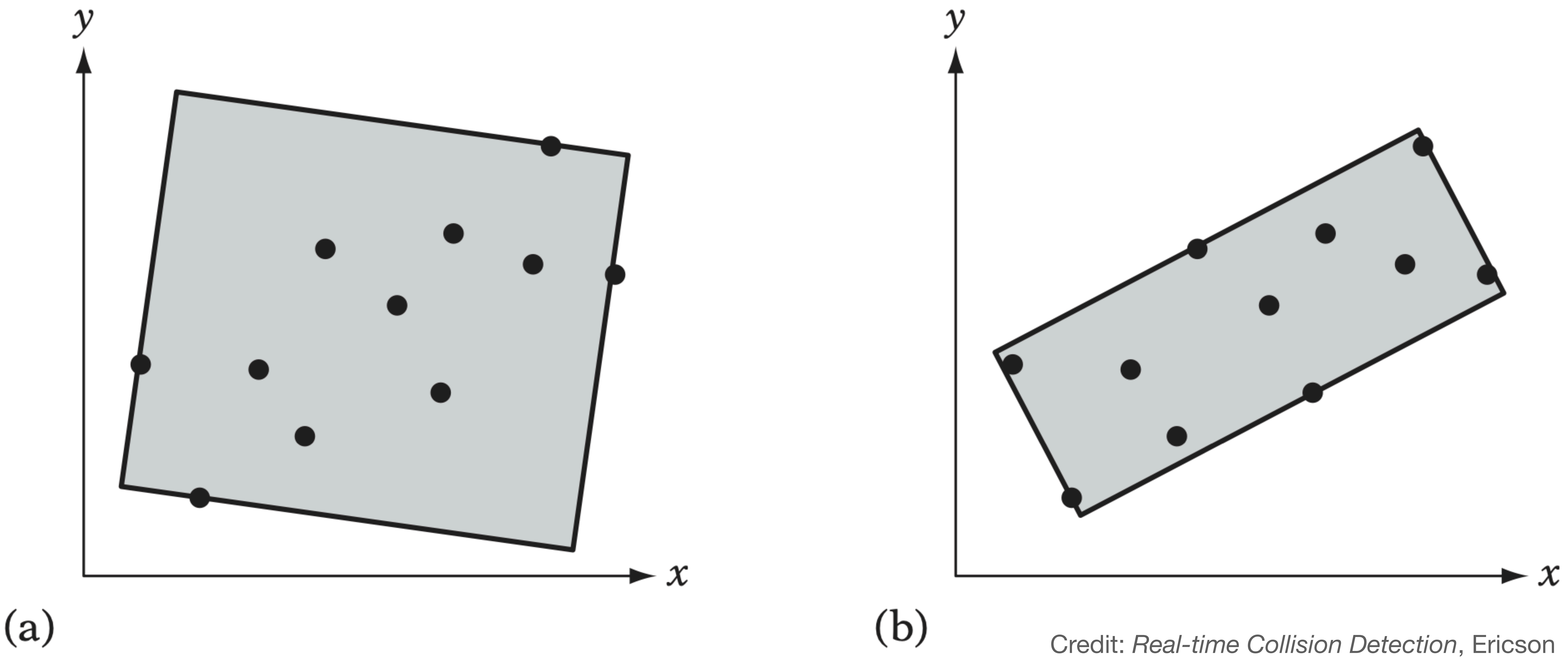


Bounding volumes



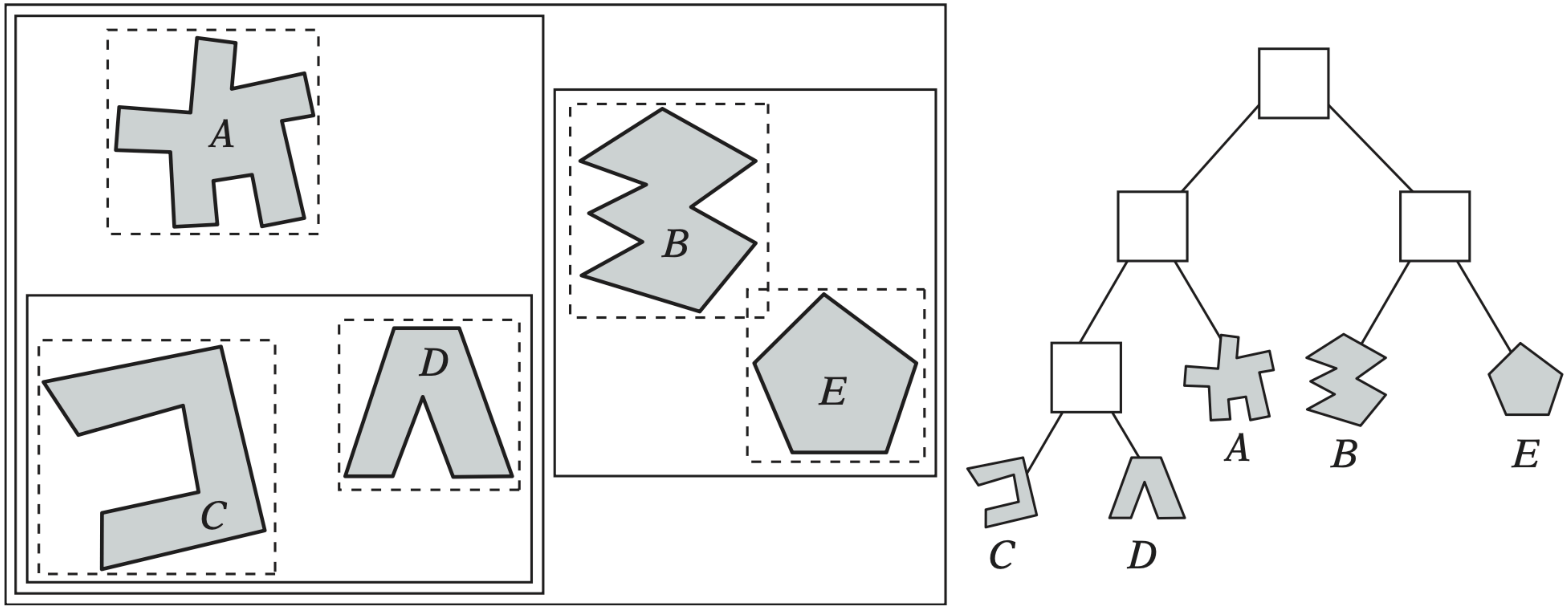
Credit: *Real-time Collision Detection*, Ericson

Bounding volumes



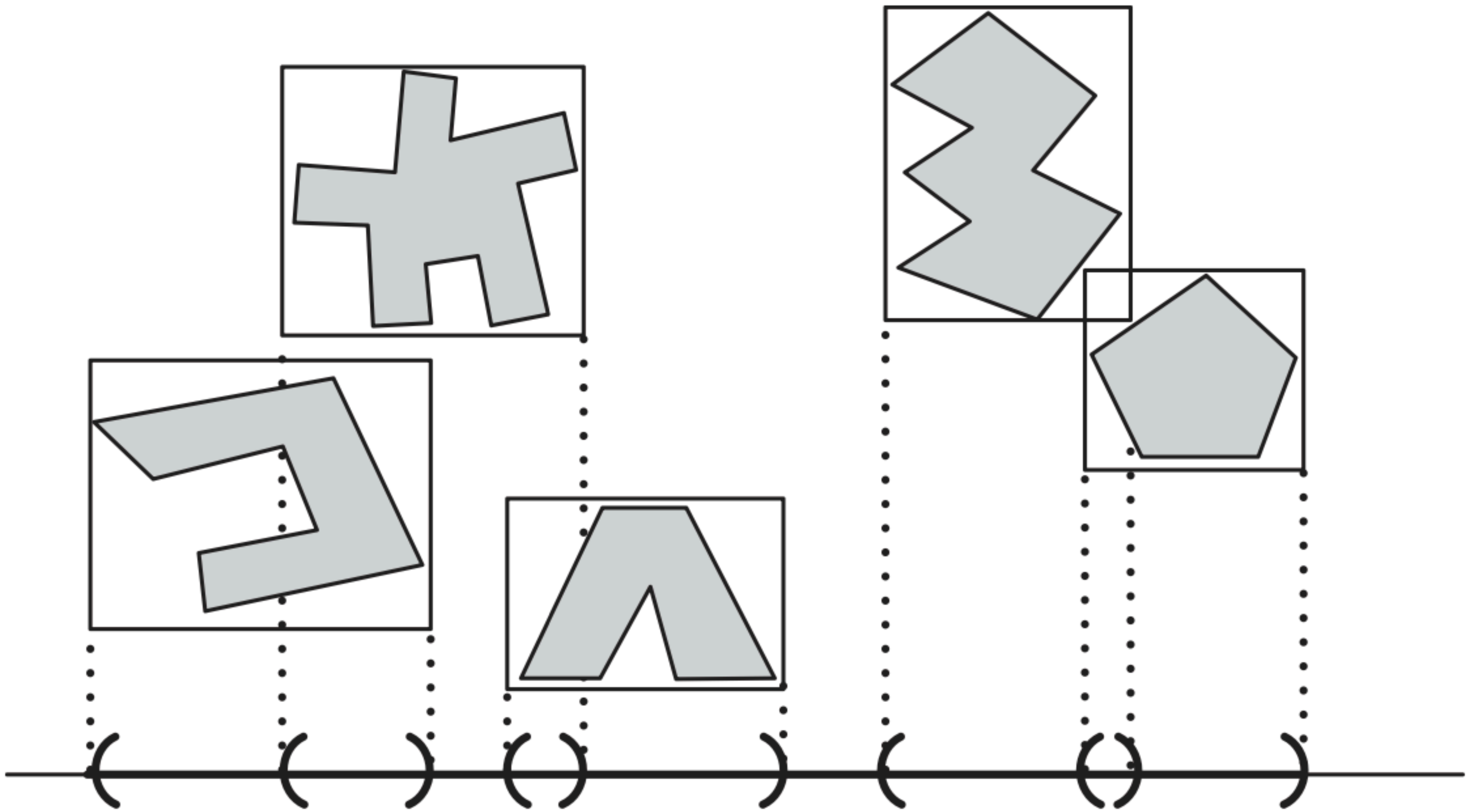
Credit: *Real-time Collision Detection*, Ericson

Broad phase with a dynamic tree of any BVs



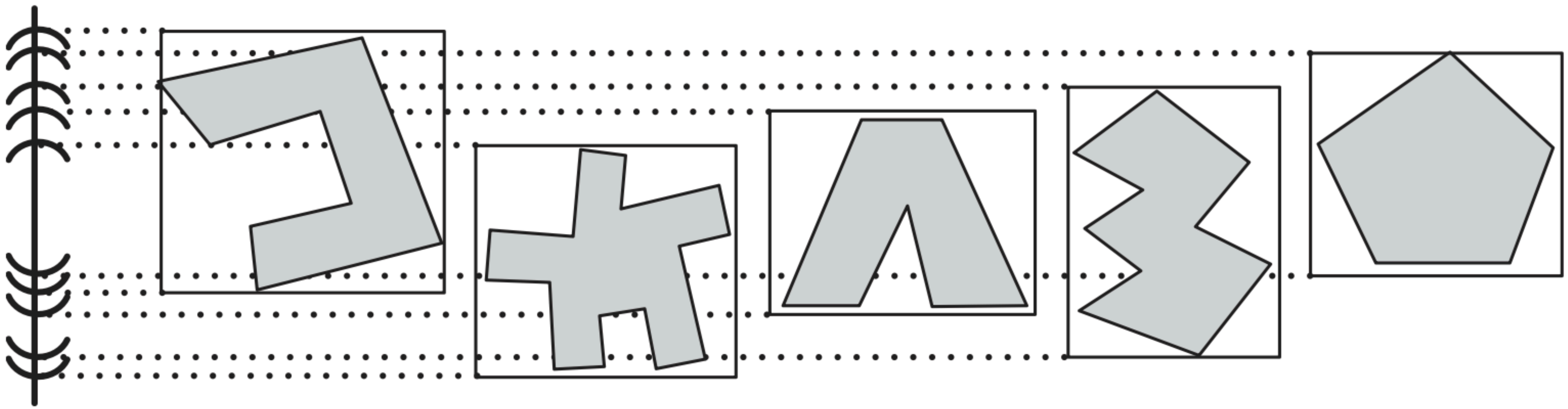
Credit: *Real-time Collision Detection*, Ericson

Broad phase with AABBs - Sweep and Prune (SaP)



Credit: *Real-time Collision Detection*, Ericson

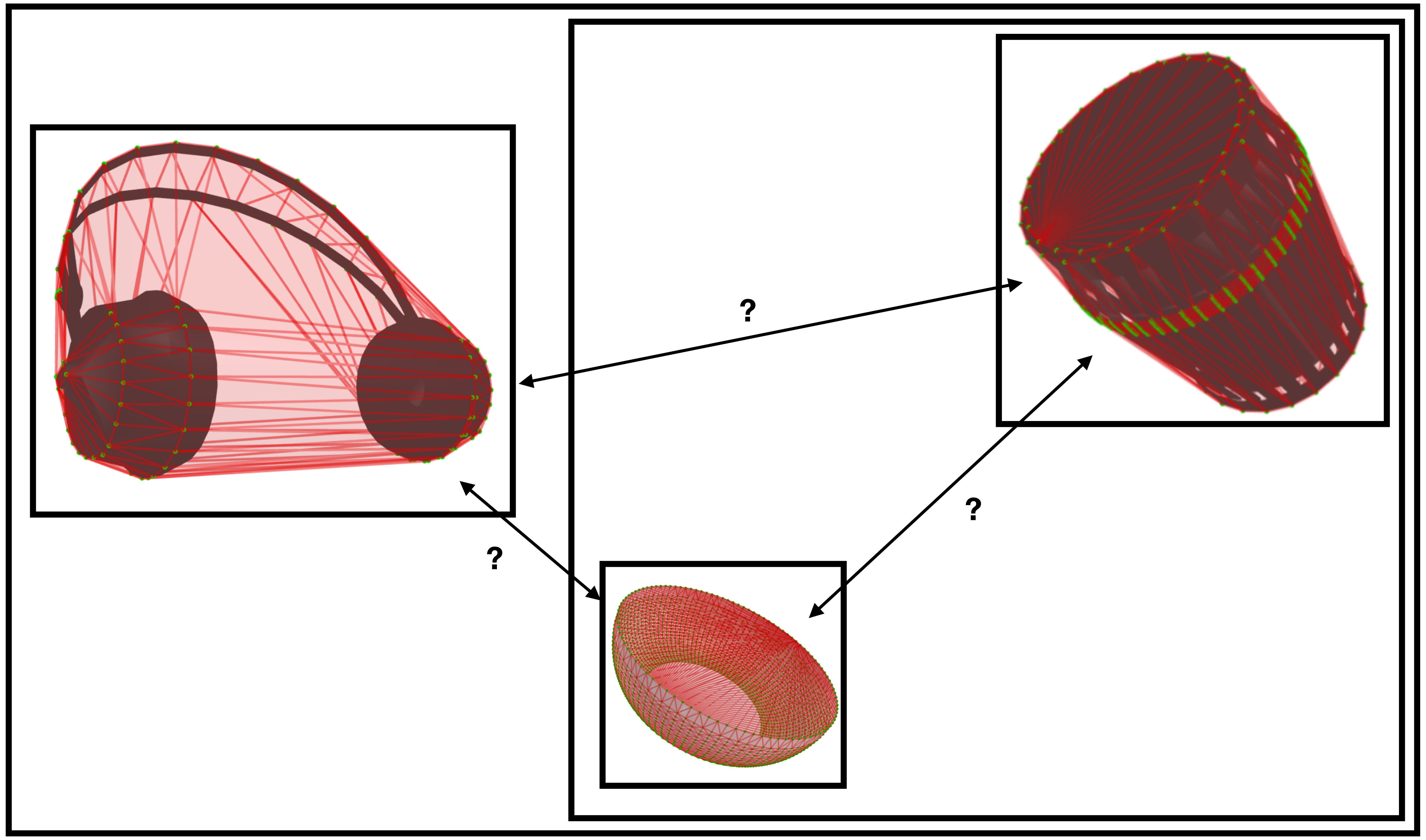
Broad phase with AABBs - Sweep and Prune (SaP)



Credit: *Real-time Collision Detection*, Ericson

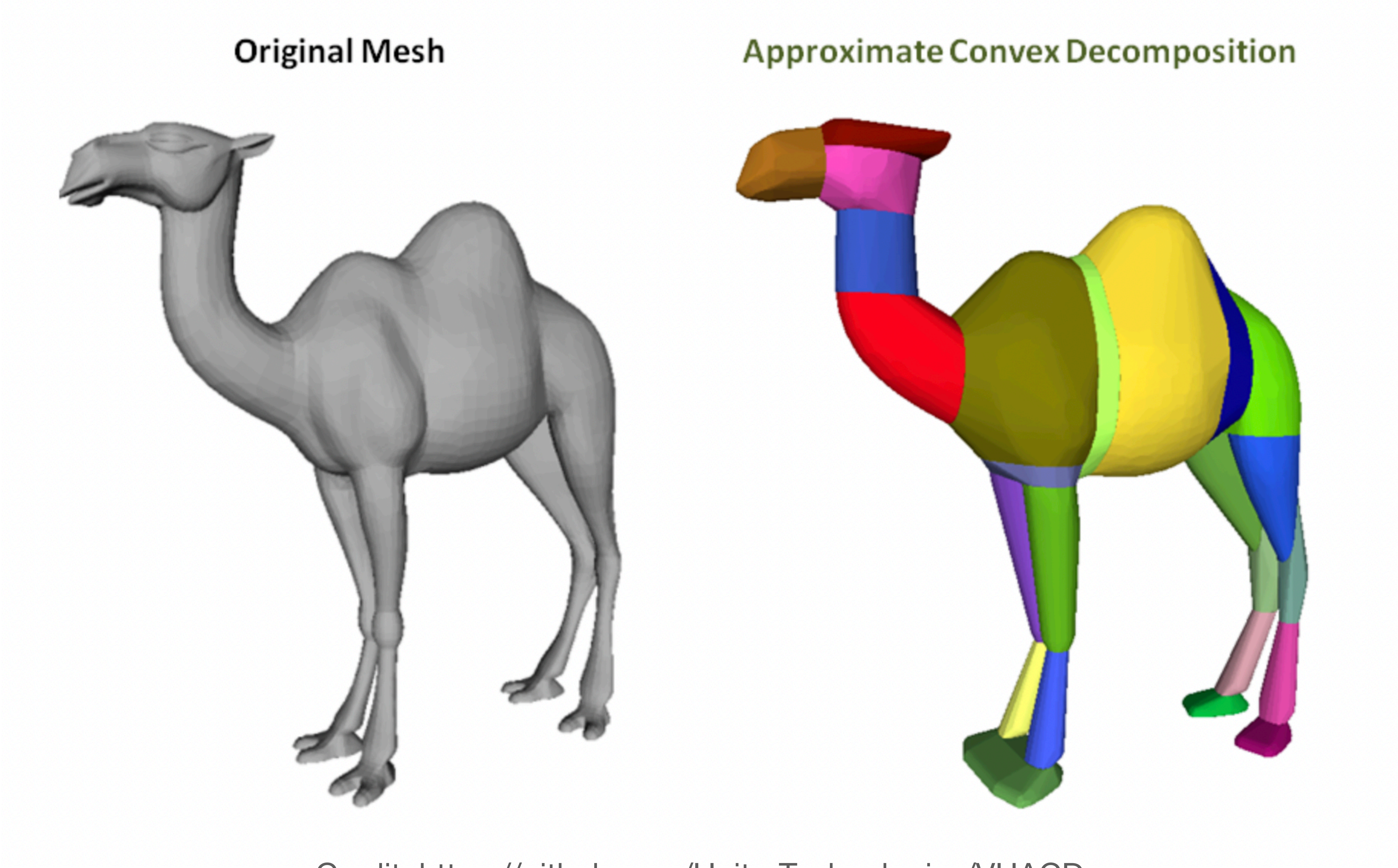
Part I conclusion - What is collision detection?

Avoid computing collisions as much as possible



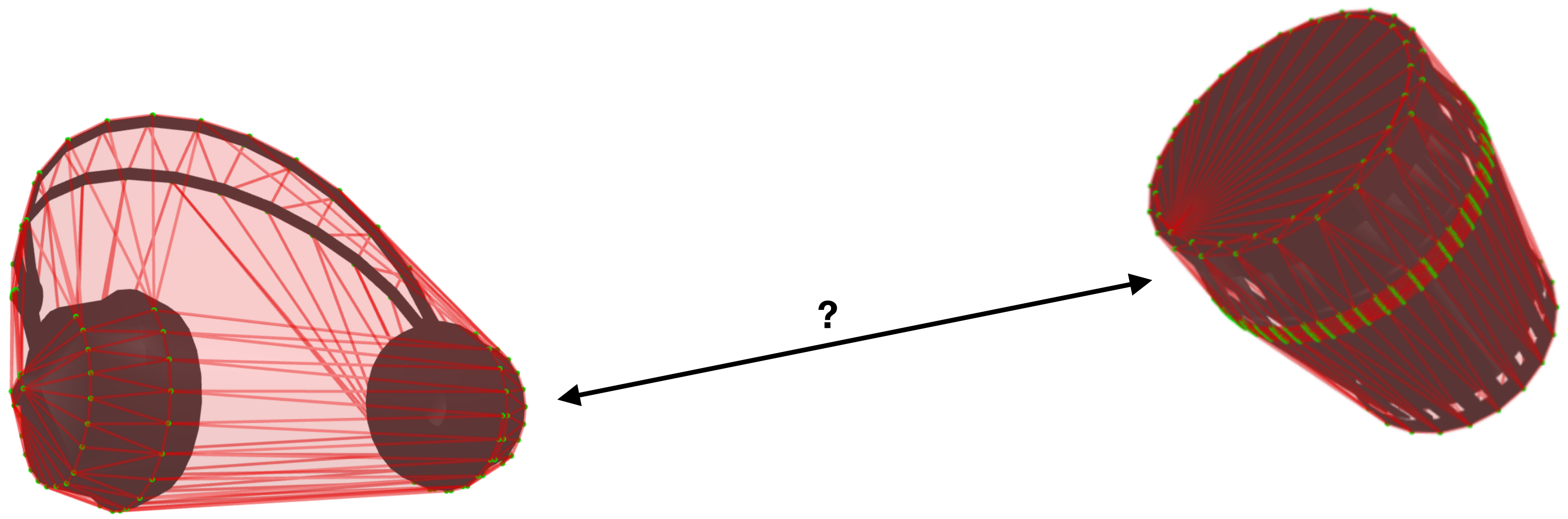
Part II - The Narrow Phase

Collision detection: convex shapes decomposition

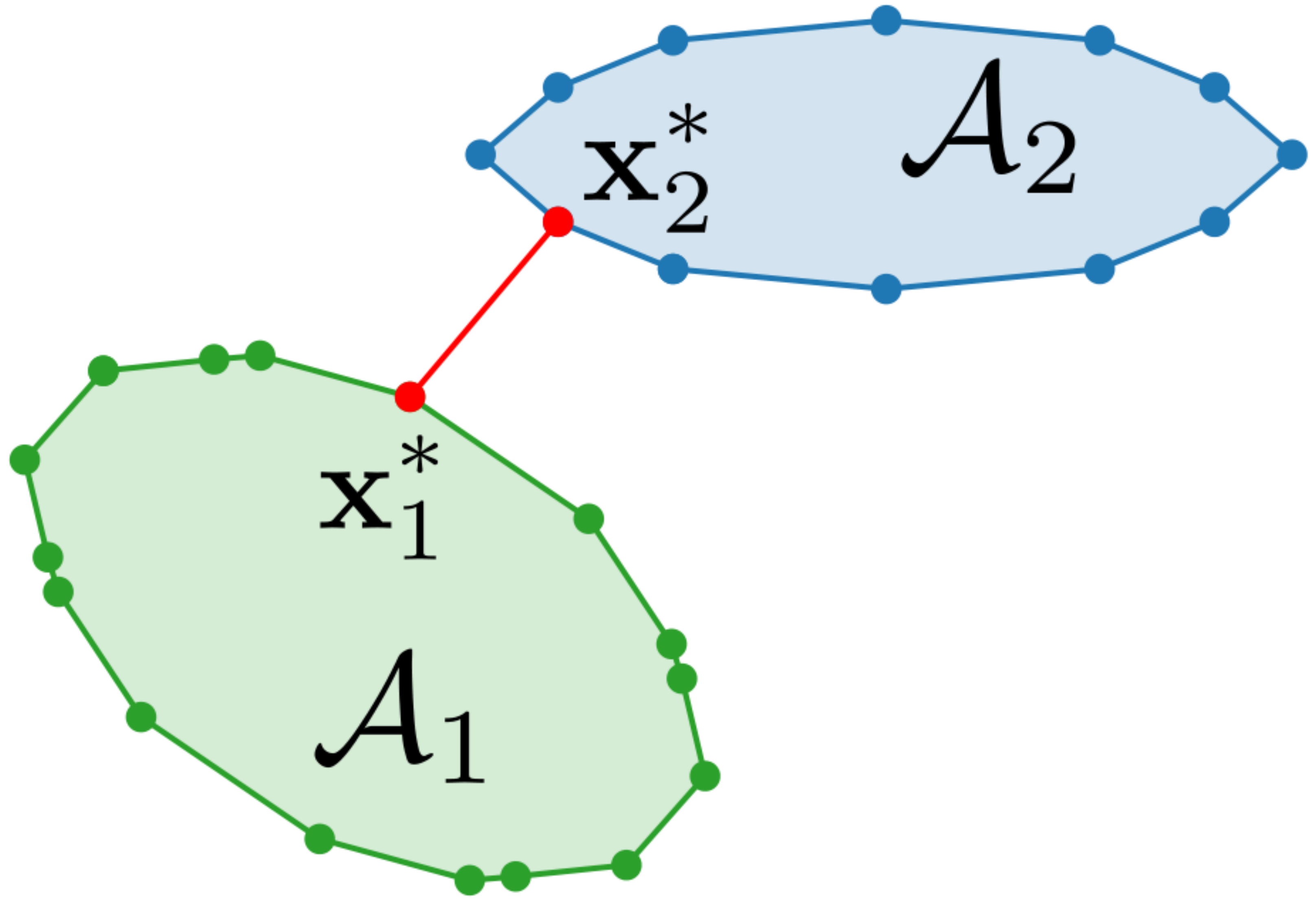


Credit: https://github.com/Unity-Technologies/VHACD

Narrow Phase Collision detection



$$\min_{x_1 \in \mathcal{A}_1, x_2 \in \mathcal{A}_2} \frac{1}{2} || x_1 - x_2 ||^2$$



Problem formulation

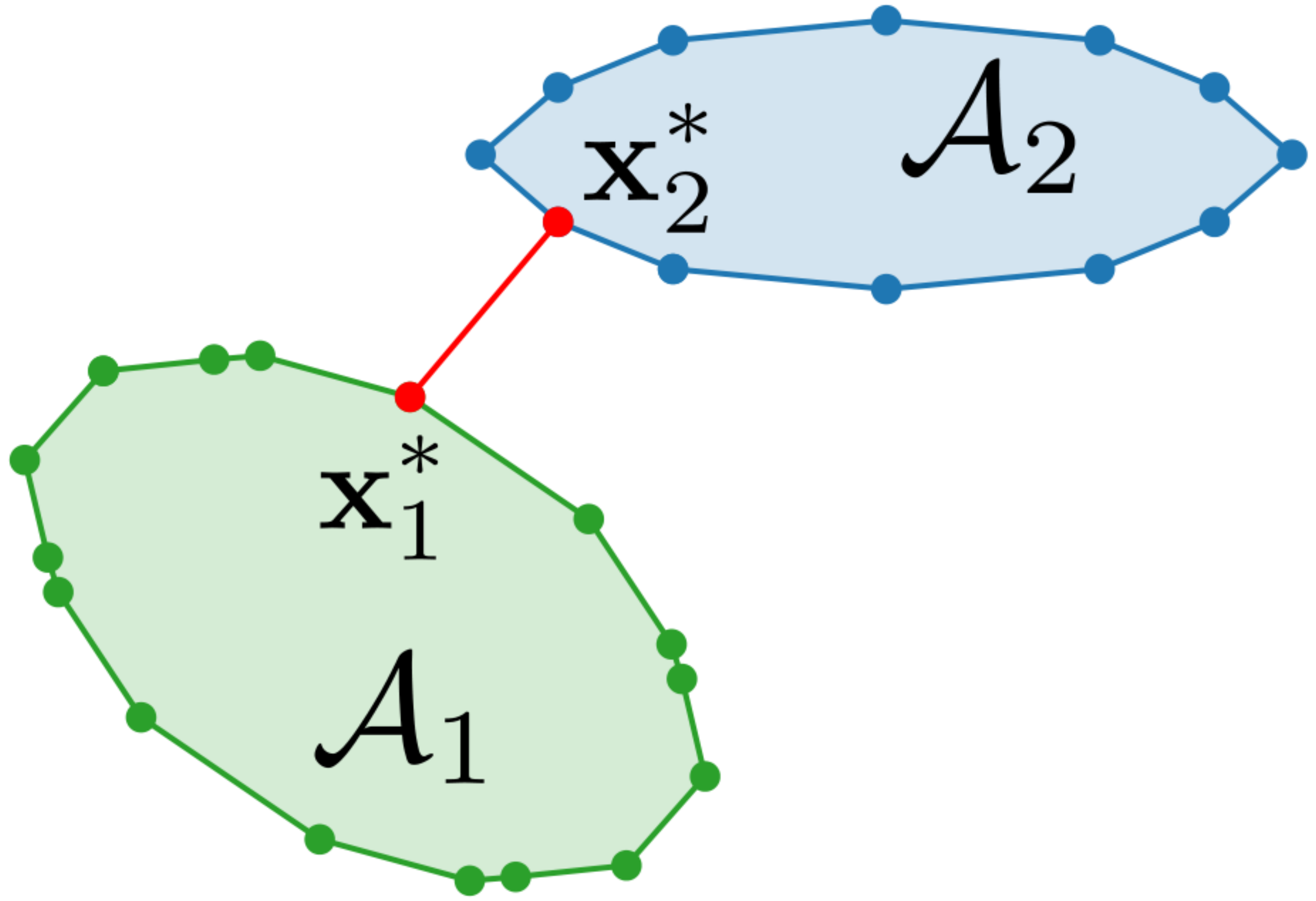
$$\min_{x_1 \in \mathcal{A}_1, x_2 \in \mathcal{A}_2} \frac{1}{2} || x_1 - x_2 ||^2$$

If the shapes are meshes →

$$\min_{x_1, x_2} \frac{1}{2} || x_1 - x_2 ||^2$$

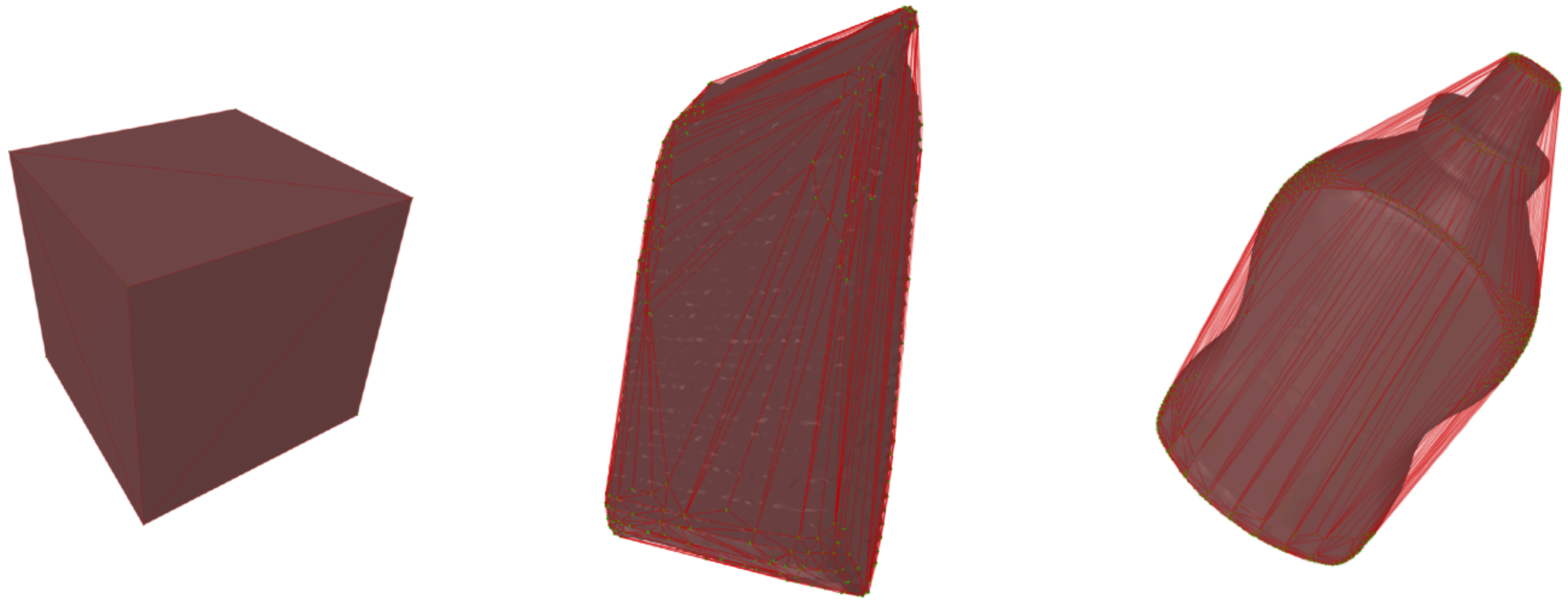
s.t. $A_1 x_1 \leq b_1$

$A_2 x_2 \leq b_2$



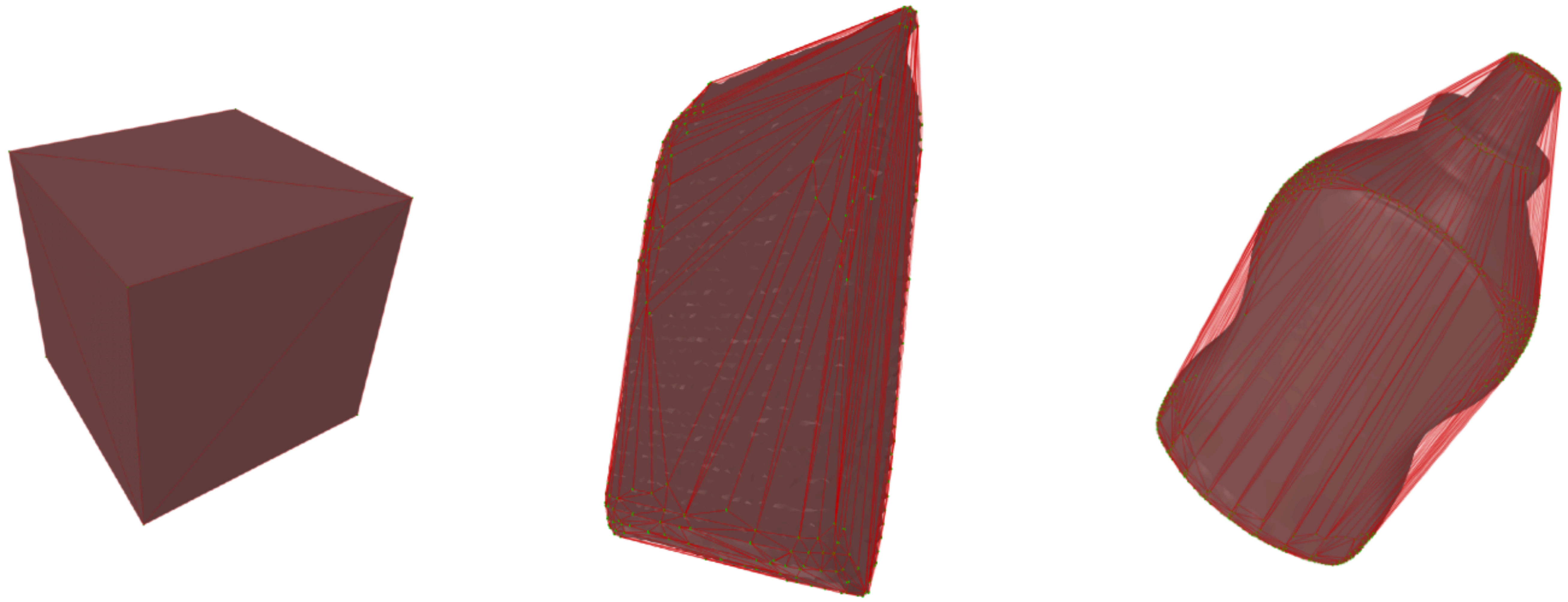
As many constraints as the number of faces in each polytope!

ProxQP vs. GJK



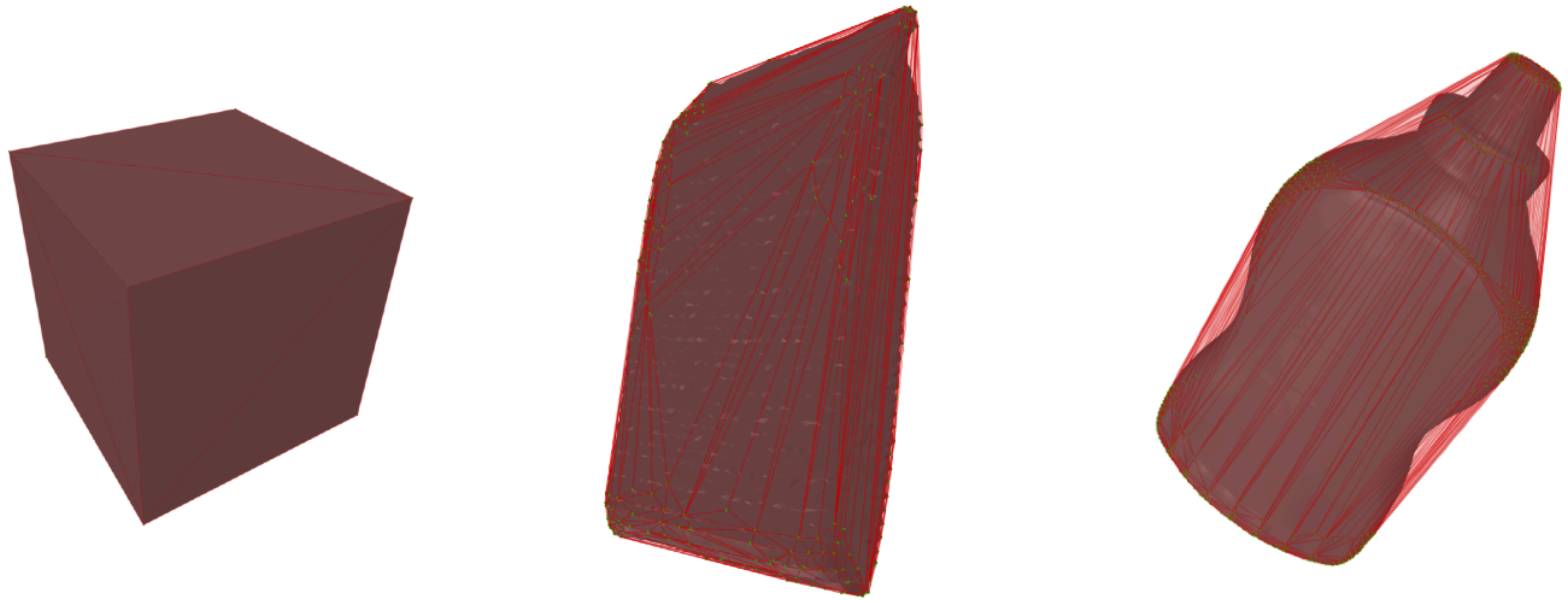
	$N_v = 8$, $N_f = 6$	$N_v = 250$, $N_f = 496$	$N_v = 940$, $N_f = 1876$
ProxQP	5.3 ± 2.7 μs	$(2 \pm 0.6) \cdot 10^3$ μs	$(20 \pm 14) \cdot 10^3$ μs

ProxQP vs. GJK



	$N_v = 8$, $N_f = 6$	$N_v = 250$, $N_f = 496$	$N_v = 940$, $N_f = 1876$
ProxQP	$5.3 \pm 2.7\ \mu s$	$(2 \pm 0.6) \cdot 10^3\ \mu s$	$(20 \pm 14) \cdot 10^3\ \mu s$
GJK	$\mathbf{0.2 \pm 0.03}\ \mu s$	$0.8 \pm 0.3\ \mu s$	$2.1 \pm 0.5\ \mu s$

ProxQP vs. GJK



	$N_v = 8$ $N_f = 6$	$N_v = 250$ $N_f = 496$	$N_v = 940$ $N_f = 1876$
ProxQP	$5.3 \pm 2.7\ \mu s$	$(2 \pm 0.6) \cdot 10^3\ \mu s$	$(20 \pm 14) \cdot 10^3\ \mu s$
GJK	$\mathbf{0.2 \pm 0.03}\ \mu s$	$0.8 \pm 0.3\ \mu s$	$2.1 \pm 0.5\ \mu s$
Ours	$\mathbf{0.2 \pm 0.05}\ \mu s$	$\mathbf{0.7 \pm 0.2}\ \mu s$	$\mathbf{1.4 \pm 0.3}\ \mu s$

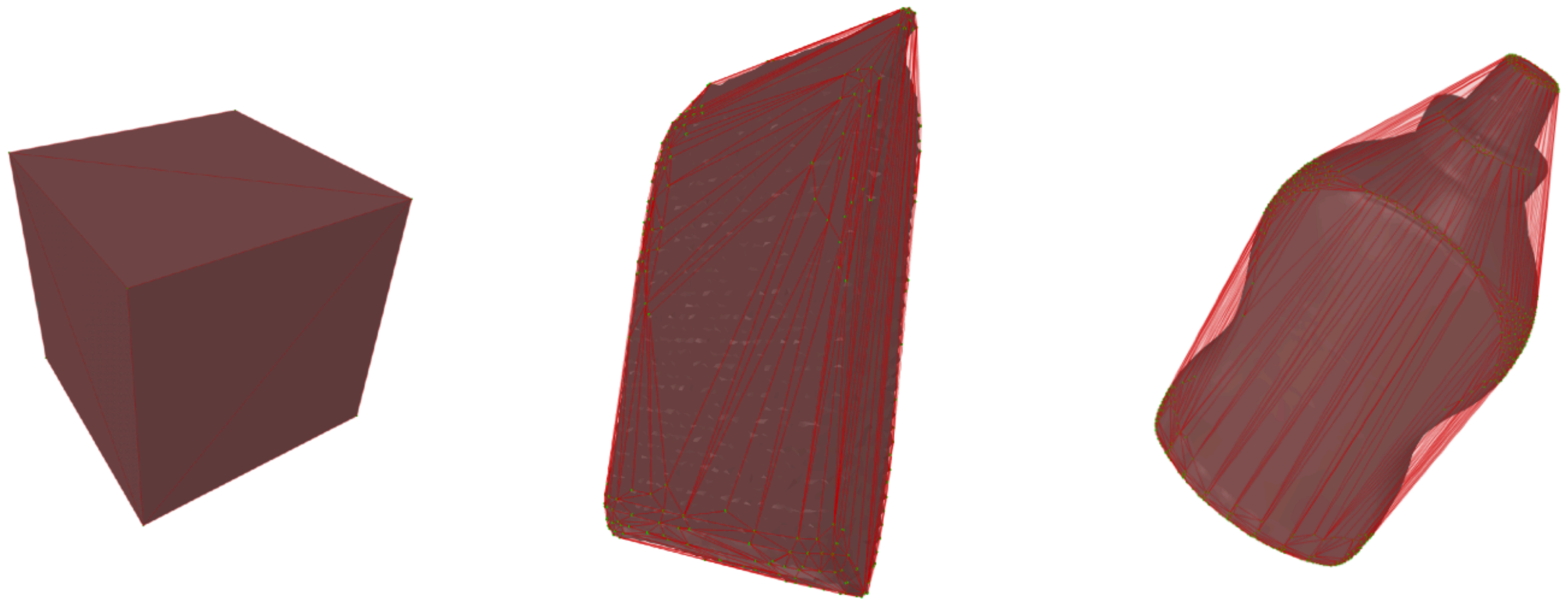
GJK - Gilbert, Johnson & Keerthi


Elmer G. Gilbert

Daniel W. Johnson


S. Sathiya Keerthi

GJK - Gilbert, Johnson & Keerthi



	$N_v = 8$, $N_f = 6$	$N_v = 250$, $N_f = 496$	$N_v = 940$, $N_f = 1876$
ProxQP	$5.3 \pm 2.7\ \mu s$	$(2 \pm 0.6) \cdot 10^3\ \mu s$	$(20 \pm 14) \cdot 10^3\ \mu s$
GJK	$\mathbf{0.2 \pm 0.03}\ \mu s$	$0.8 \pm 0.3\ \mu s$	$2.1 \pm 0.5\ \mu s$
Ours	$\mathbf{0.2 \pm 0.05}\ \mu s$	$\mathbf{0.7 \pm 0.2}\ \mu s$	$\mathbf{1.4 \pm 0.3}\ \mu s$

What is GJK?
Why is it so fast?

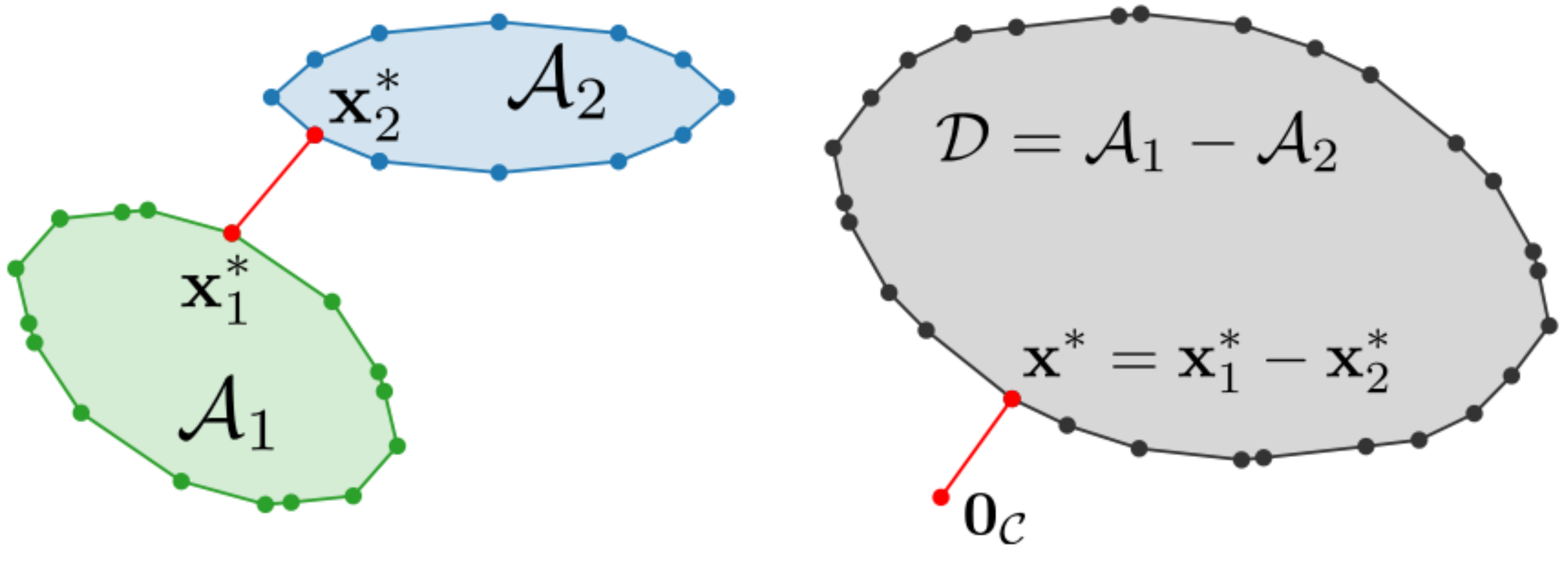
GJK = Acceleration of Frank-Wolfe applied to a Minimum Norm Point problem (MNP)

- MNP?
- Frank-Wolfe?
- Acceleration?

Recasting the collision problem to a MNP

The Minkowski difference:

$$\mathscr{D} = \mathscr{A}_1 - \mathscr{A}_2 = \{x = x_1 - x_2,\ x_1 \in \mathscr{A}_1\ x_2 \in \mathscr{A}_2\}$$

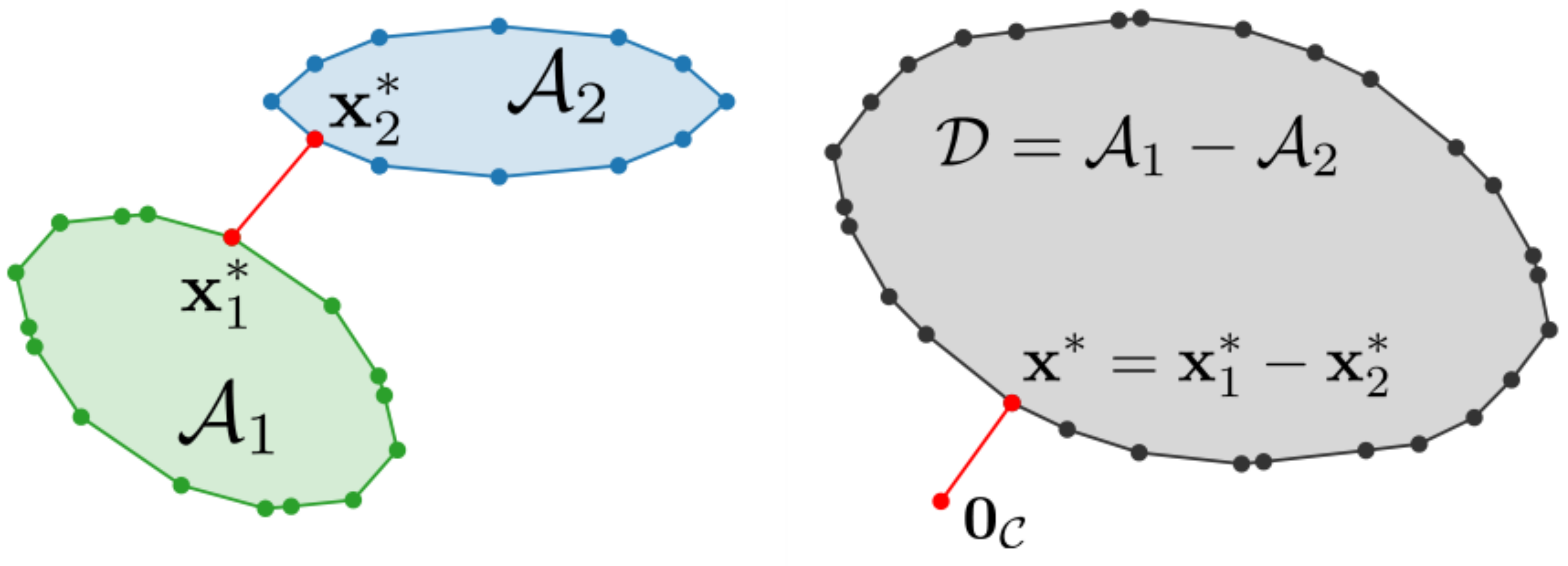


No collision

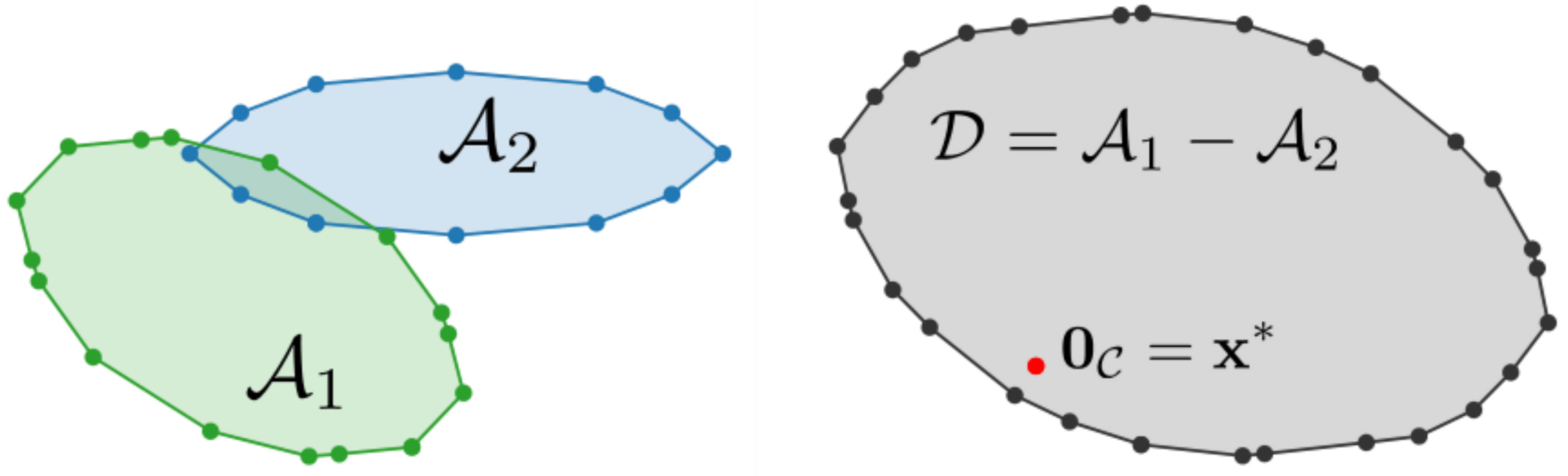
Recasting the collision problem to a MNP

The Minkowski difference:

$$\mathscr{D} = \mathscr{A}_1 - \mathscr{A}_2 = \{x = x_1 - x_2,\ x_1 \in \mathscr{A}_1\ x_2 \in \mathscr{A}_2\}$$



No collision

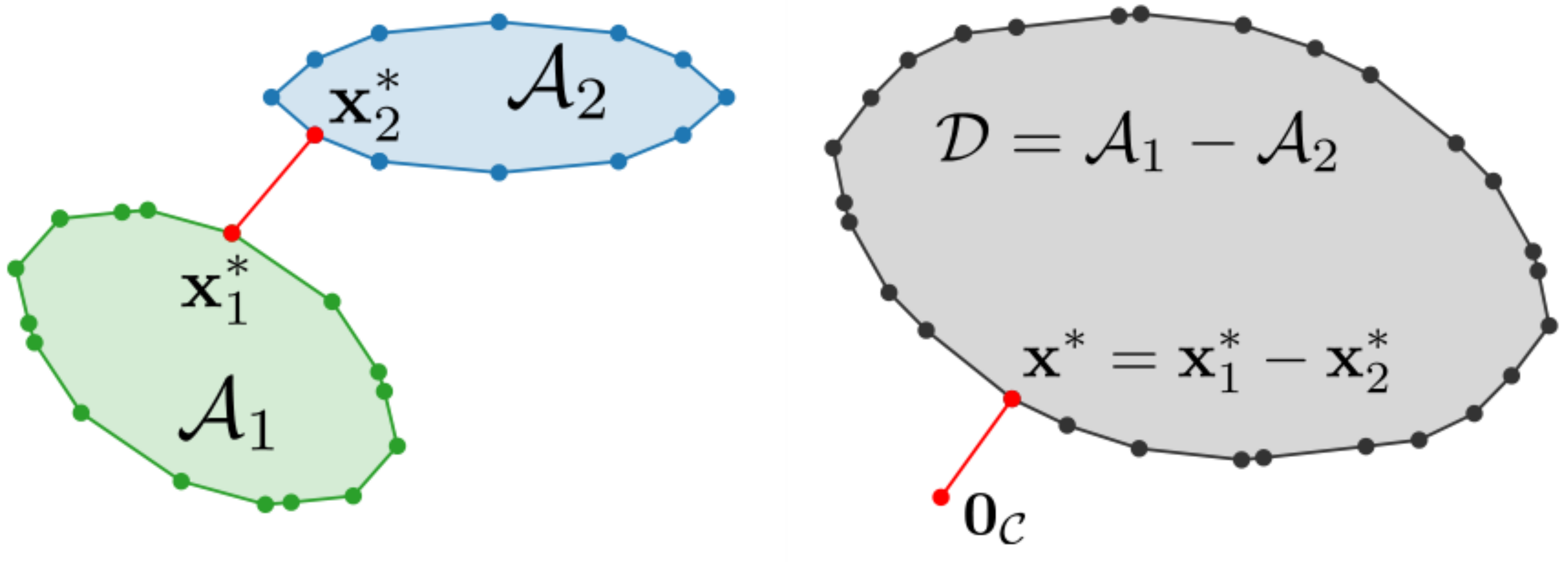


Collision

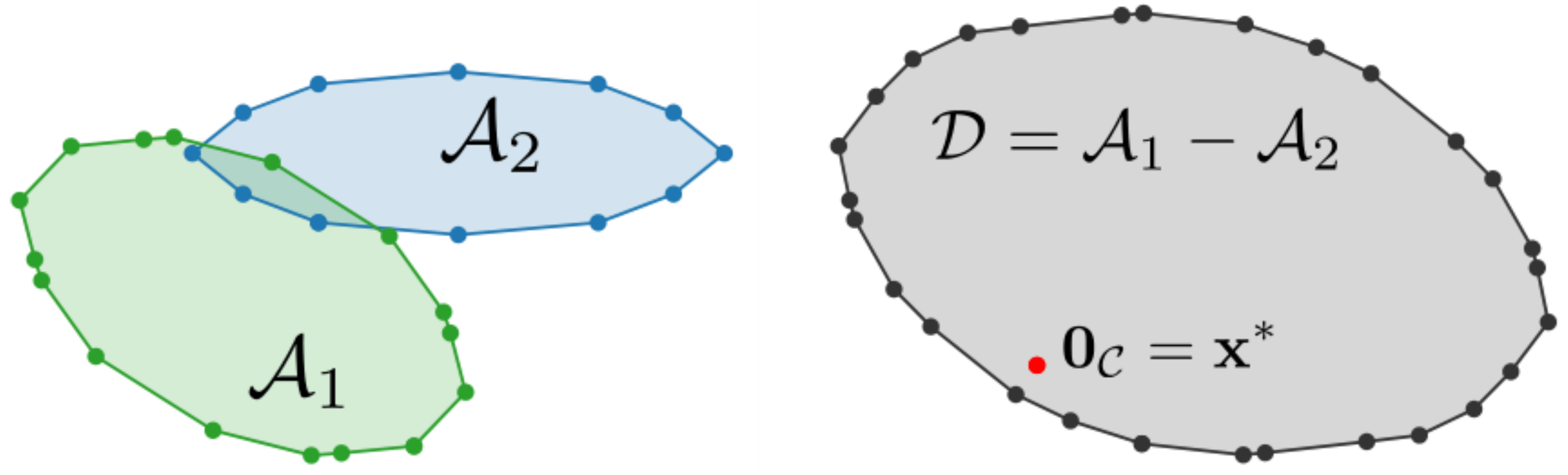
Recasting the collision problem to a MNP

The Minkowski difference:

$$\mathscr{D} = \mathscr{A}_1 - \mathscr{A}_2 = \{x = x_1 - x_2,\ x_1 \in \mathscr{A}_1\ x_2 \in \mathscr{A}_2\}$$



No collision



Collision

$$\min_{x_1 \in \mathscr{A}_1, x_2 \in \mathscr{A}_2} \frac{1}{2} ||x_1 - x_2||^2 \longrightarrow \boxed{\min_{x \in \mathscr{D}} \frac{1}{2} ||x||^2}$$

MNP

Recasting the collision problem to a MNP

The Minkowski difference:

$$\mathcal{D} = \mathcal{A}_1 - \mathcal{A}_2 = \{x = x_1 - x_2, \, x_1 \in \mathcal{A}_1 \, x_2 \in \mathcal{A}_2\}$$

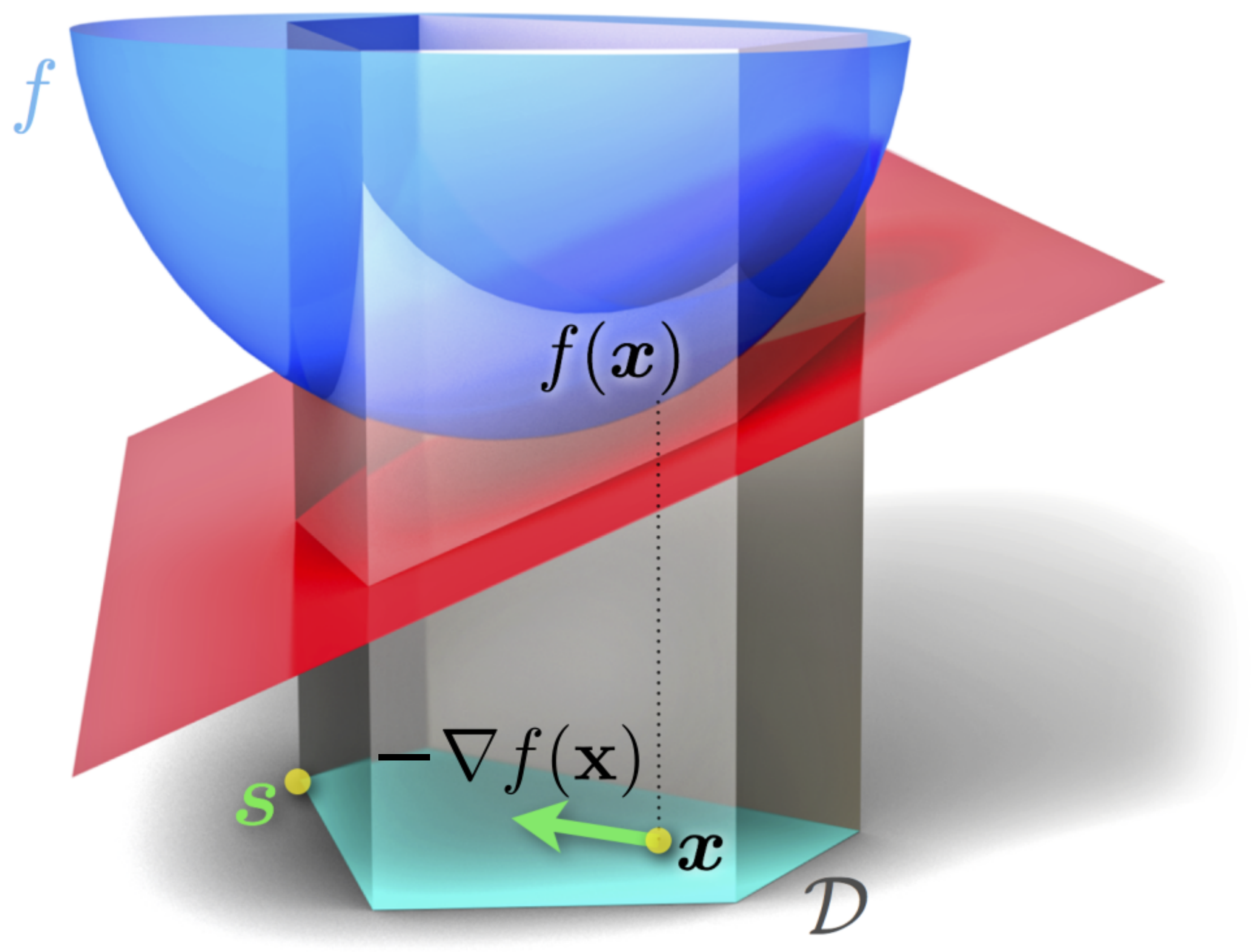
Problem: the Minkowski difference is intractable.
Solution: work implicitly with the Minkowski difference
Algorithm: Frank-Wolfe

$$\min_{x_1 \in \mathcal{A}_1, x_2 \in \mathcal{A}_2} \frac{1}{2} ||x_1 - x_2||^2 \longrightarrow \min_{x \in \mathcal{D}} \frac{1}{2} ||x||^2$$

MNP

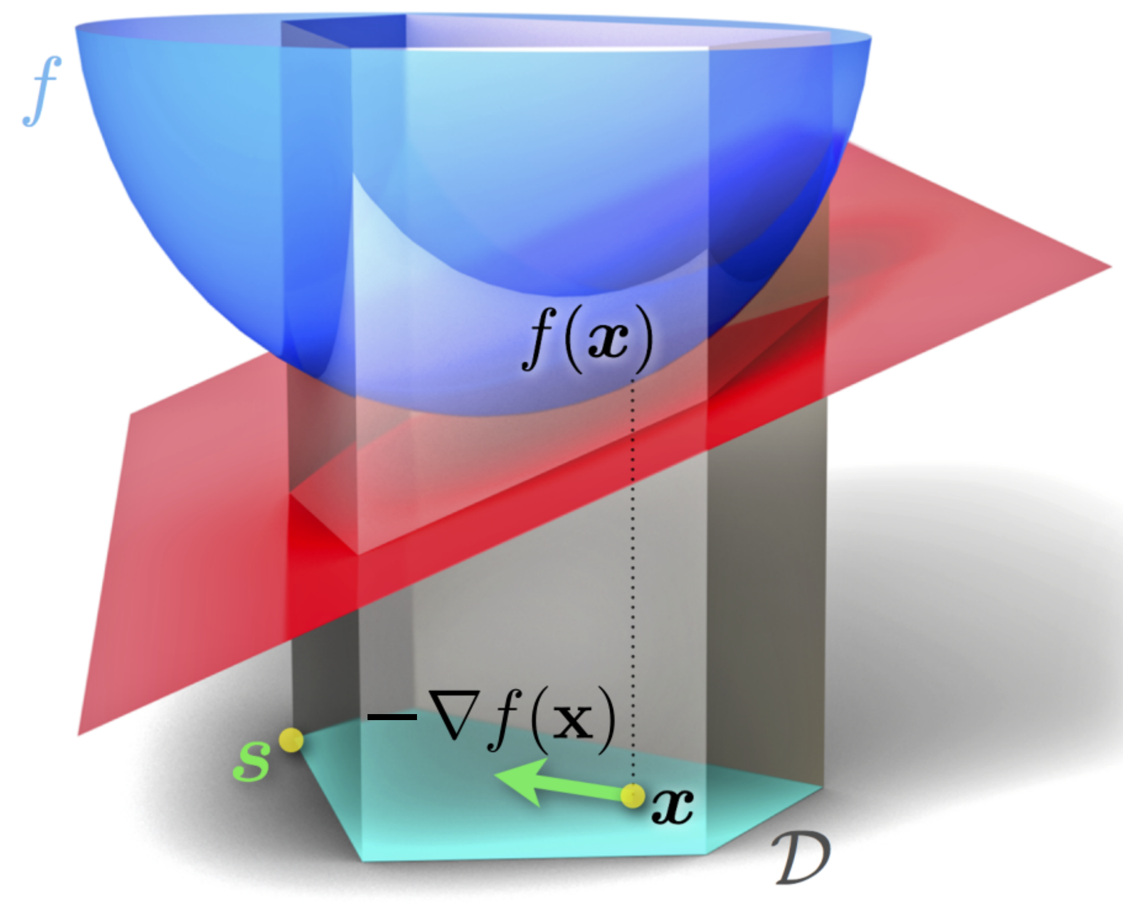
The Frank-Wolfe algorithm

$$\min_{x \in \mathscr{D}} f(x)$$ f convex, $\mathscr{D}$ convex



The Frank-Wolfe algorithm

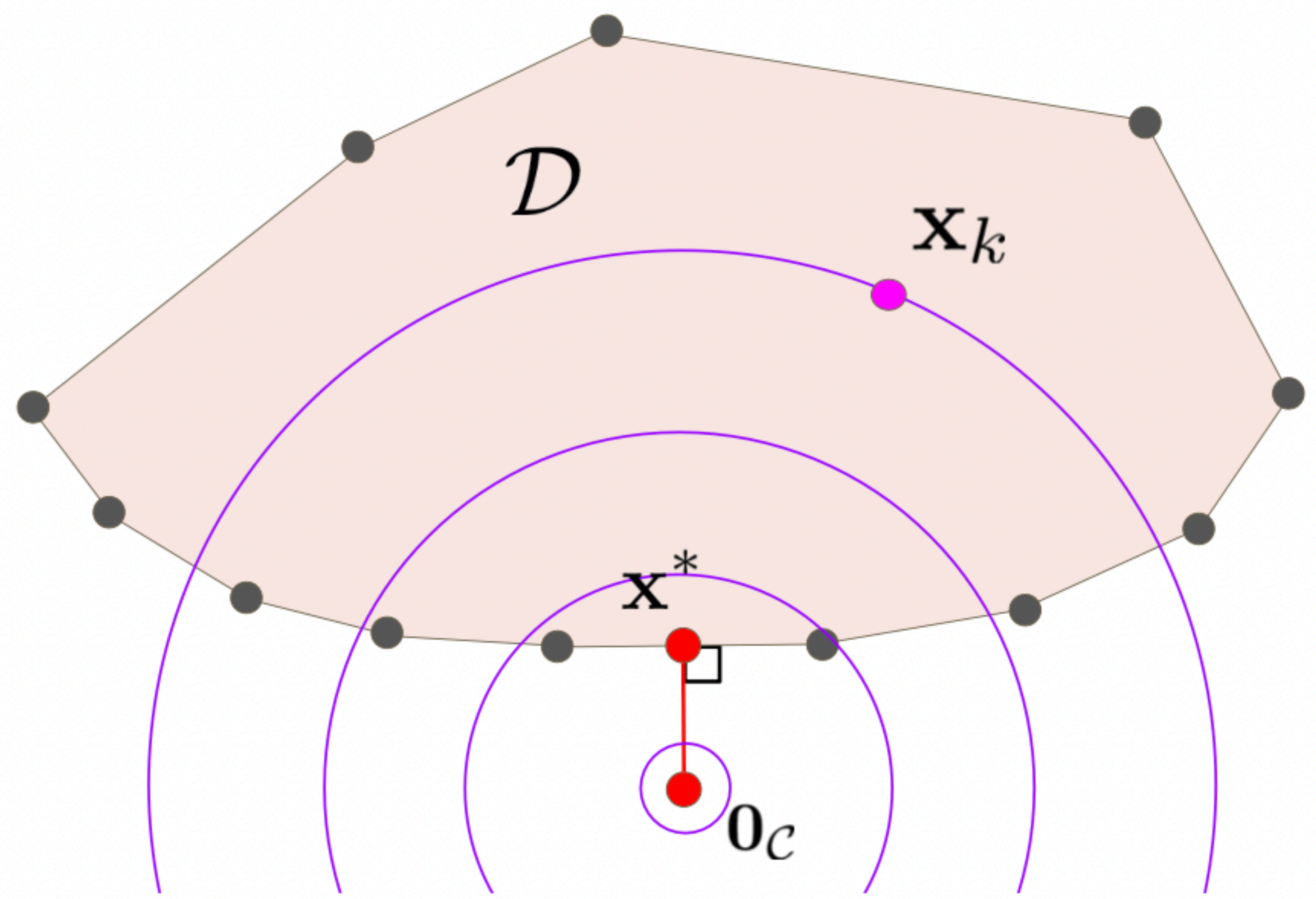
$$\min_{x \in \mathcal{D}} f(x)$$ f convex, $\mathcal{D}$ convex



Collision detection:

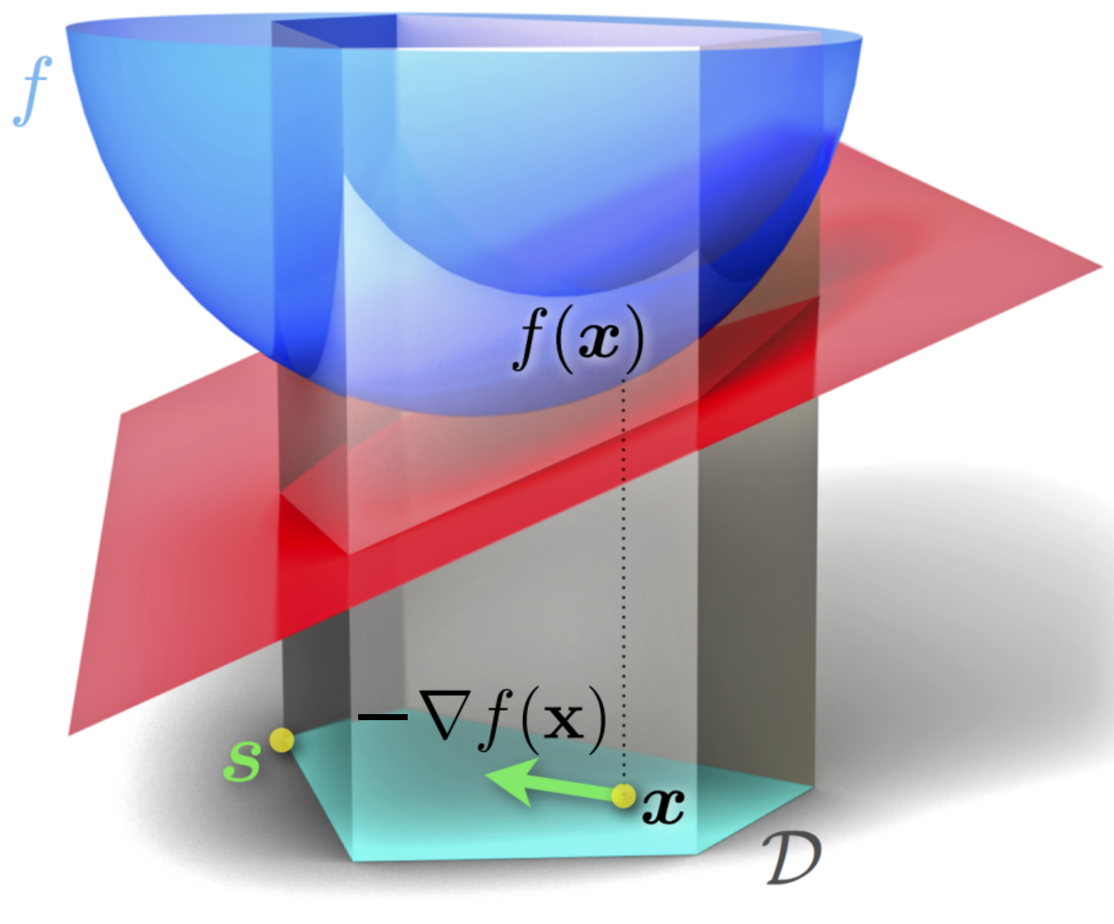
$$f(x) = \frac{1}{2} ||x||^2$$

$\mathcal{D}$ Minkowski difference of two shapes

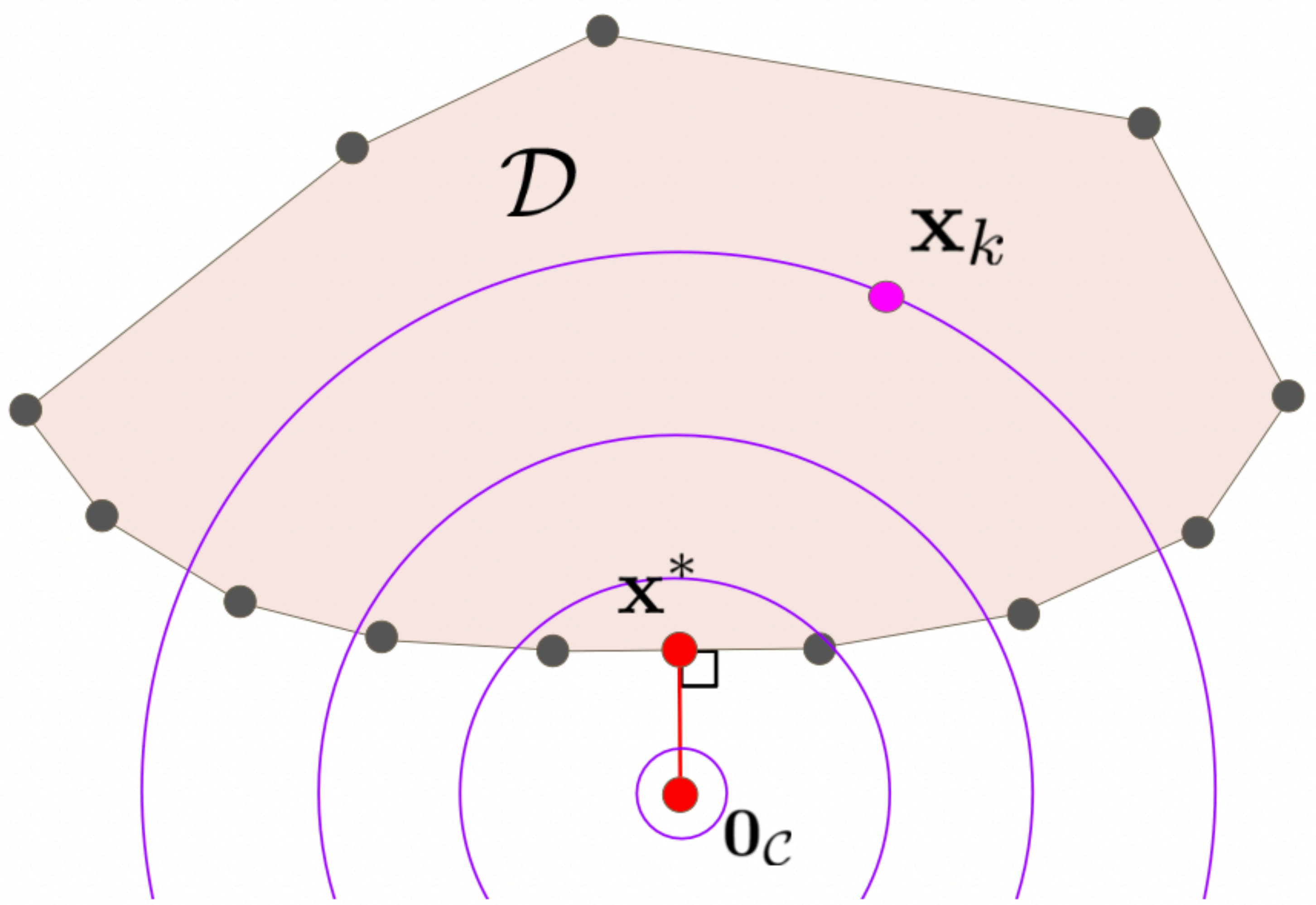


The Frank-Wolfe algorithm

$$\min_{x \in \mathscr{D}} f(x) \quad f \text{ convex, } \mathscr{D} \text{ convex}$$

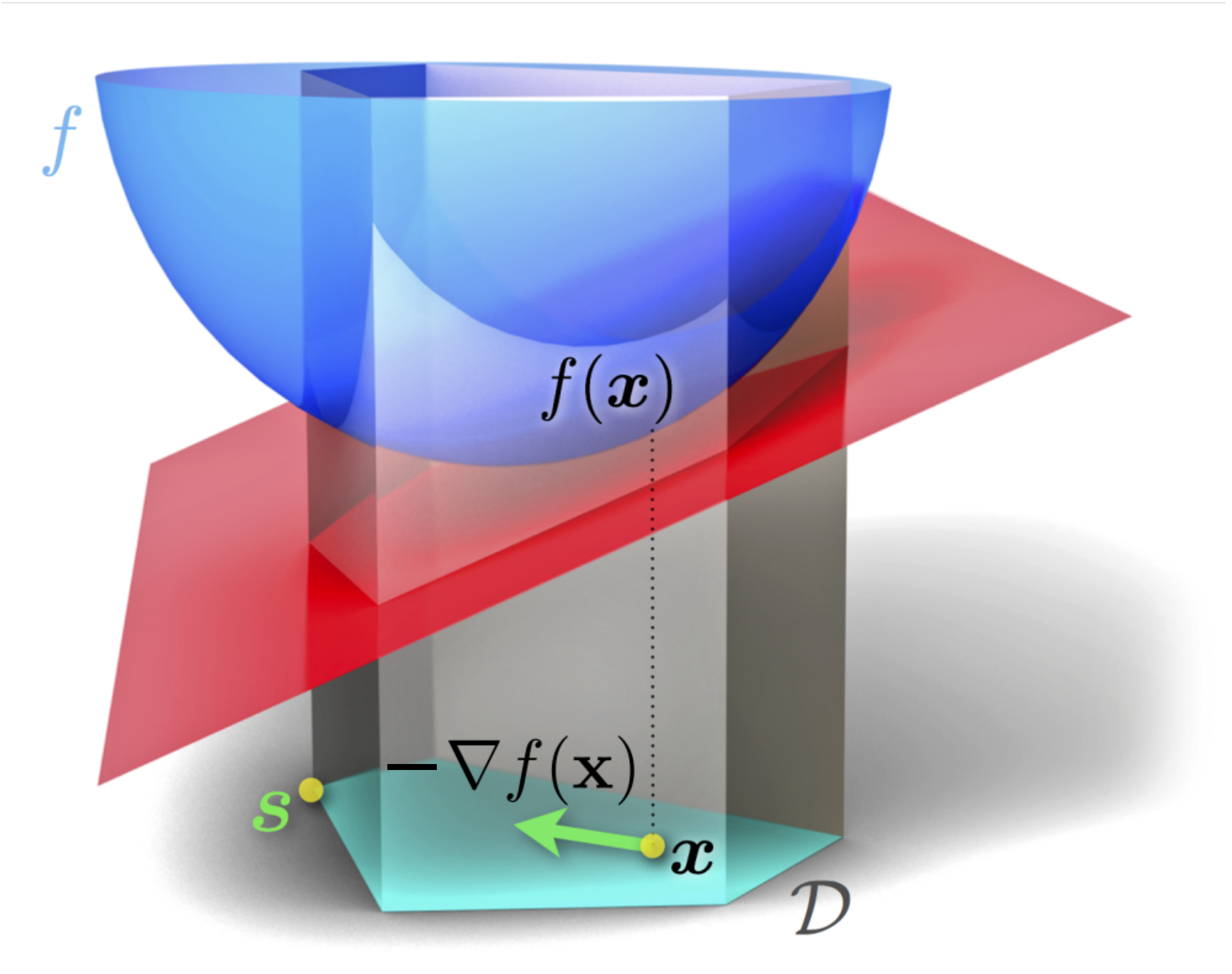


Frank-Wolfe = “constrained gradient descent”



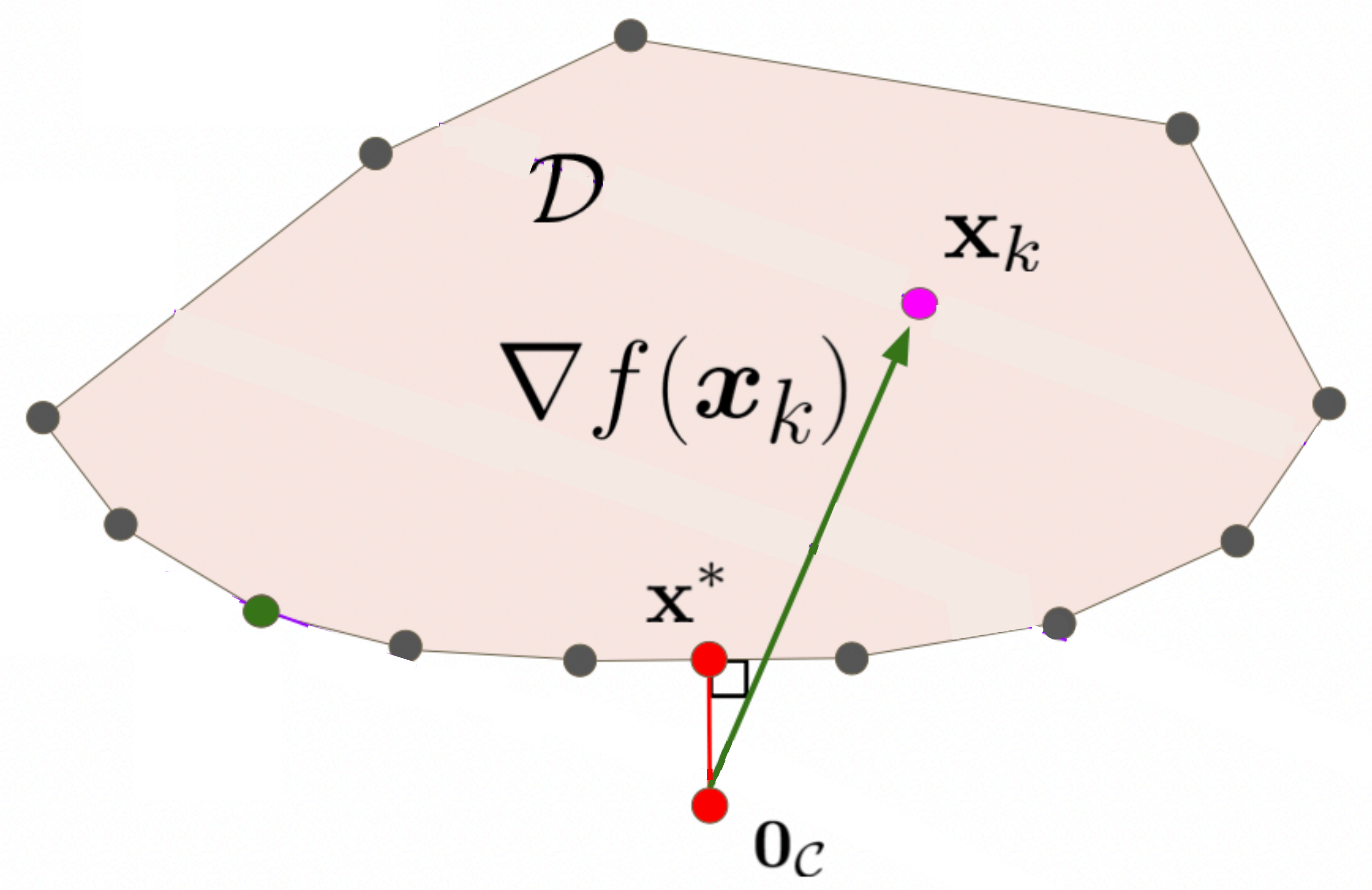
The Frank-Wolfe algorithm

$$\min_{x \in \mathcal{D}} f(x) \quad f \text{ convex, } \mathcal{D} \text{ convex}$$



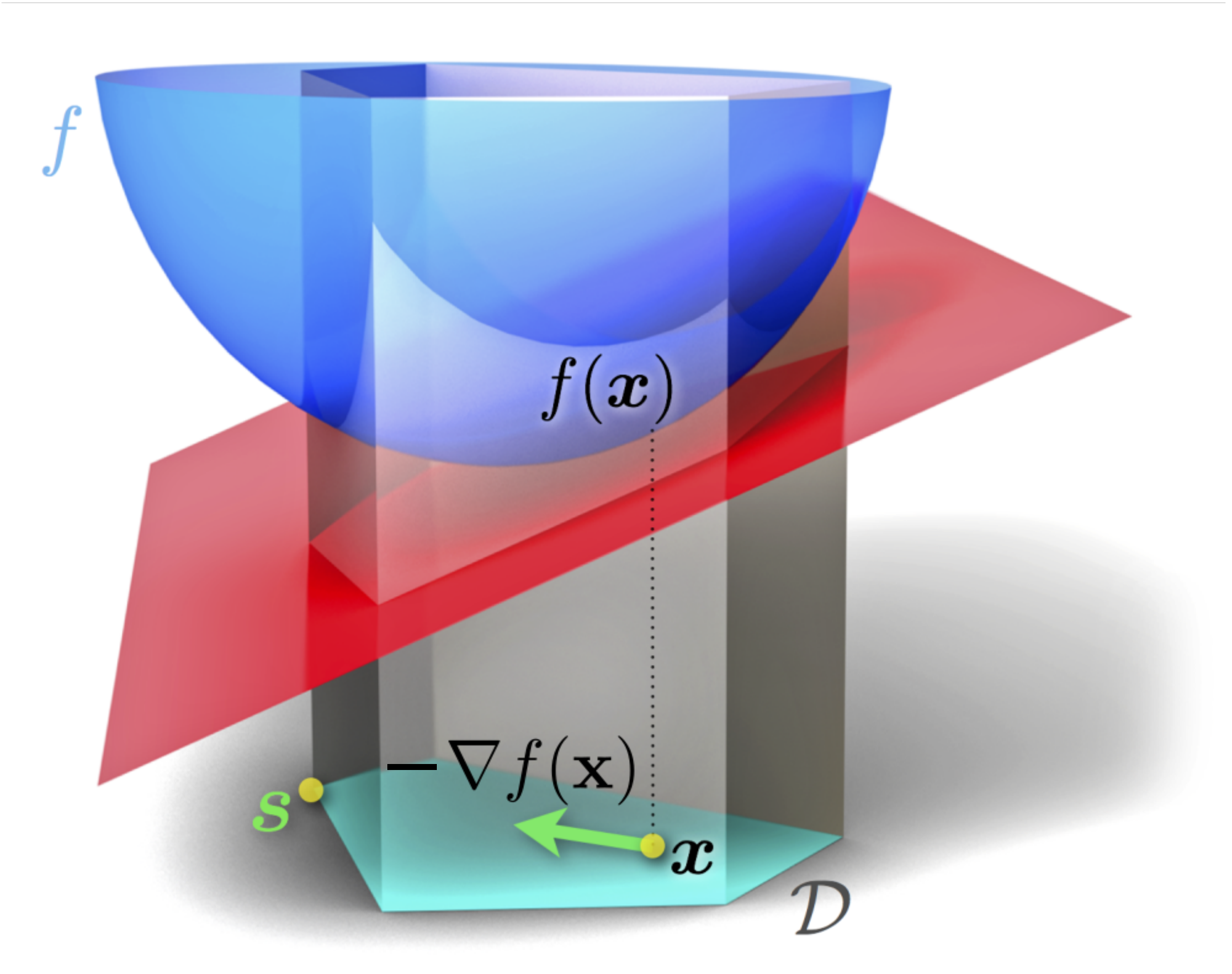
Frank-Wolfe = "constrained gradient descent":

<u>Step 1:</u> Compute gradient $\nabla f(x_k)$ at current iterate x_k



The Frank-Wolfe algorithm

$\min_{x \in \mathcal{D}} f(x)$ f convex, $\mathcal{D}$ convex

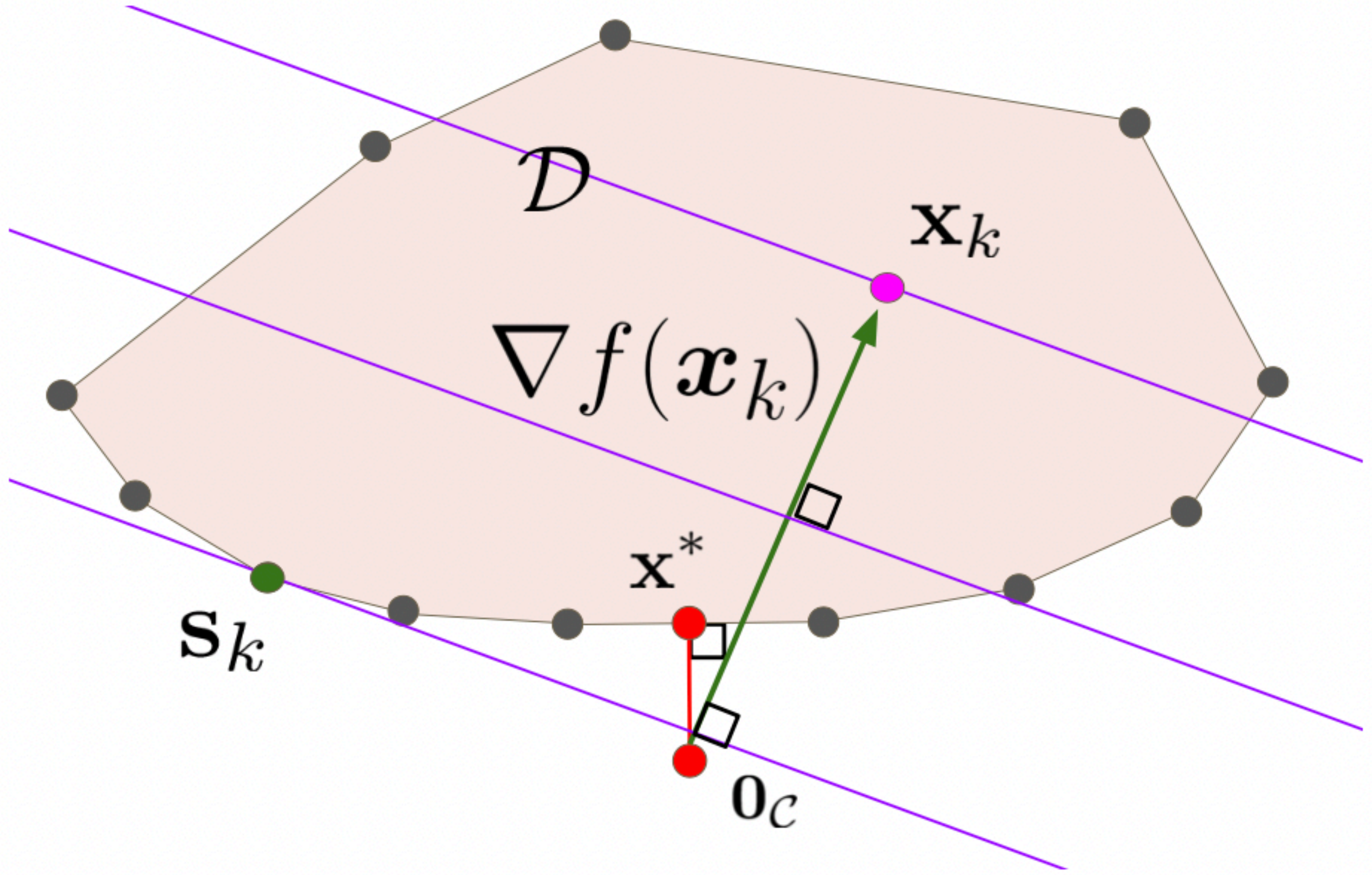


Frank-Wolfe = "constrained gradient descent":

<u>Step 1:</u> Compute gradient $\nabla f(x_k)$ at current iterate x_k

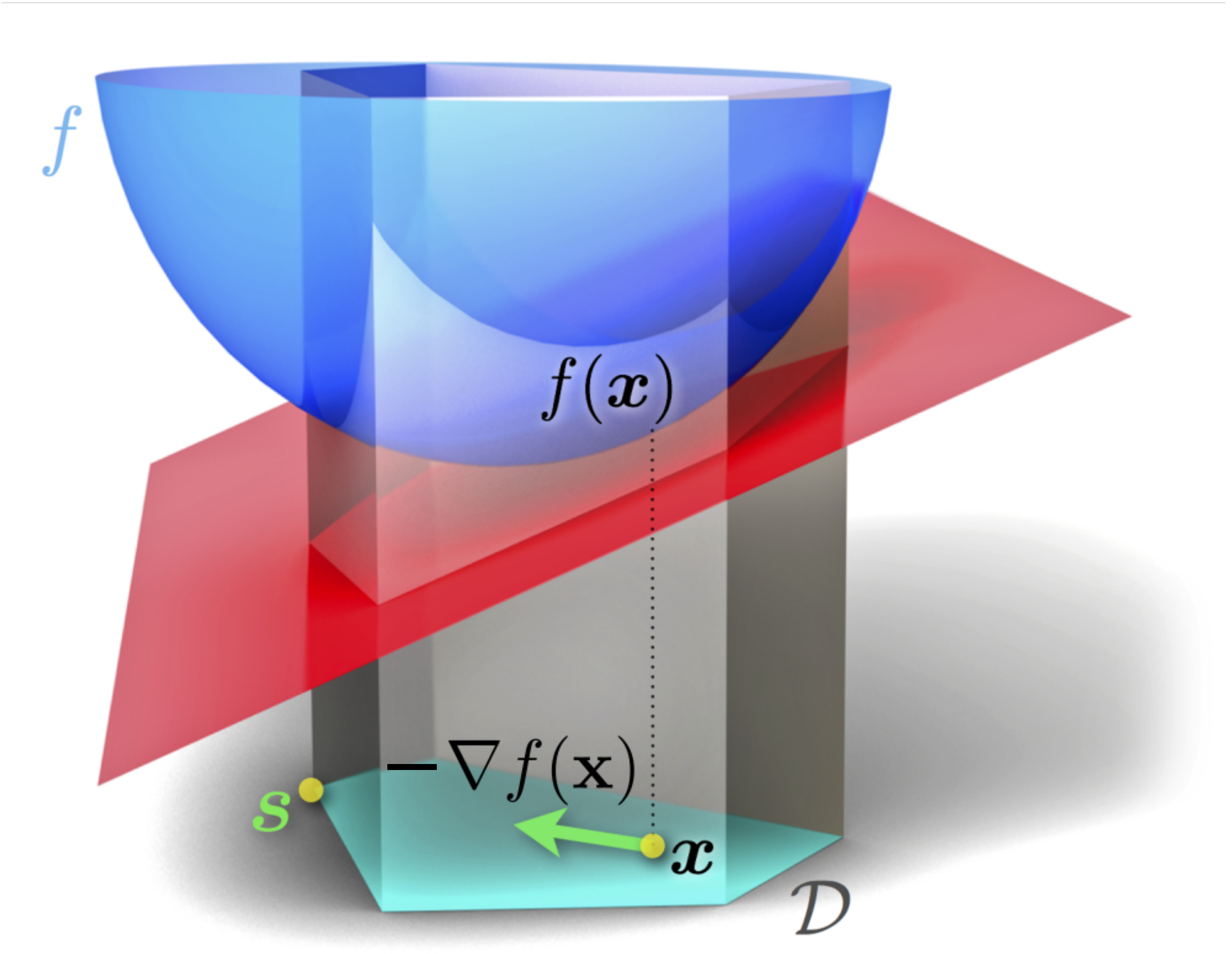
<u>Step 2:</u> Compute point $s_k \in \mathcal{D}$ "most" in direction $-\nabla f(x_k)$

-> support point $s_k = \text{argmin}_{y \in \mathcal{D}} < y, \nabla f(x_k) >$



The Frank-Wolfe algorithm

$\min_{x \in \mathscr{D}} f(x)$ $\quad f$ convex, $\mathscr{D}$ convex



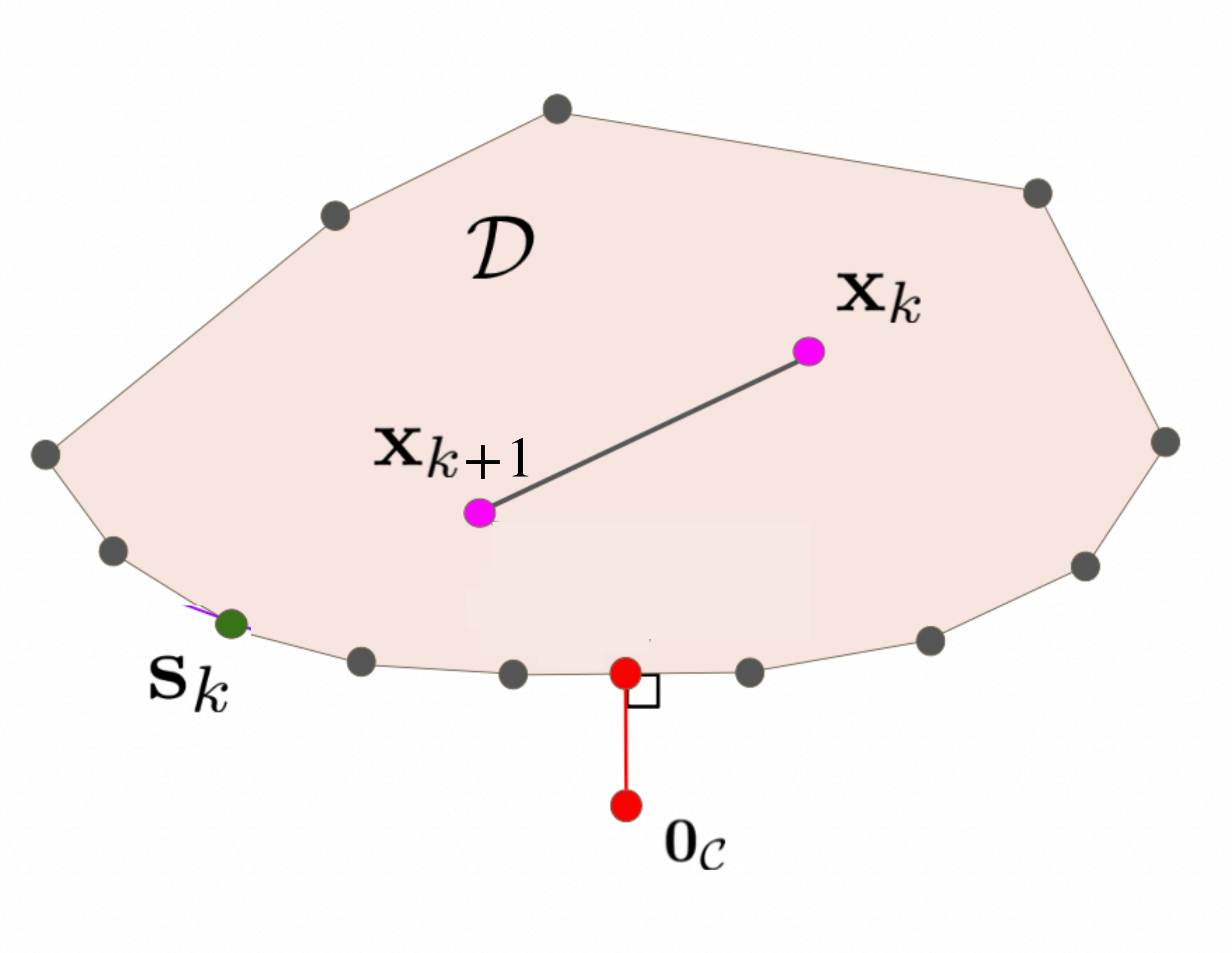
Frank-Wolfe = "constrained gradient descent":

Step 1: Compute gradient $\nabla f(x_k)$ at current iterate x_k

Step 2: Compute point $s_k \in \mathscr{D}$ "most" in direction $-\nabla f(x_k)$

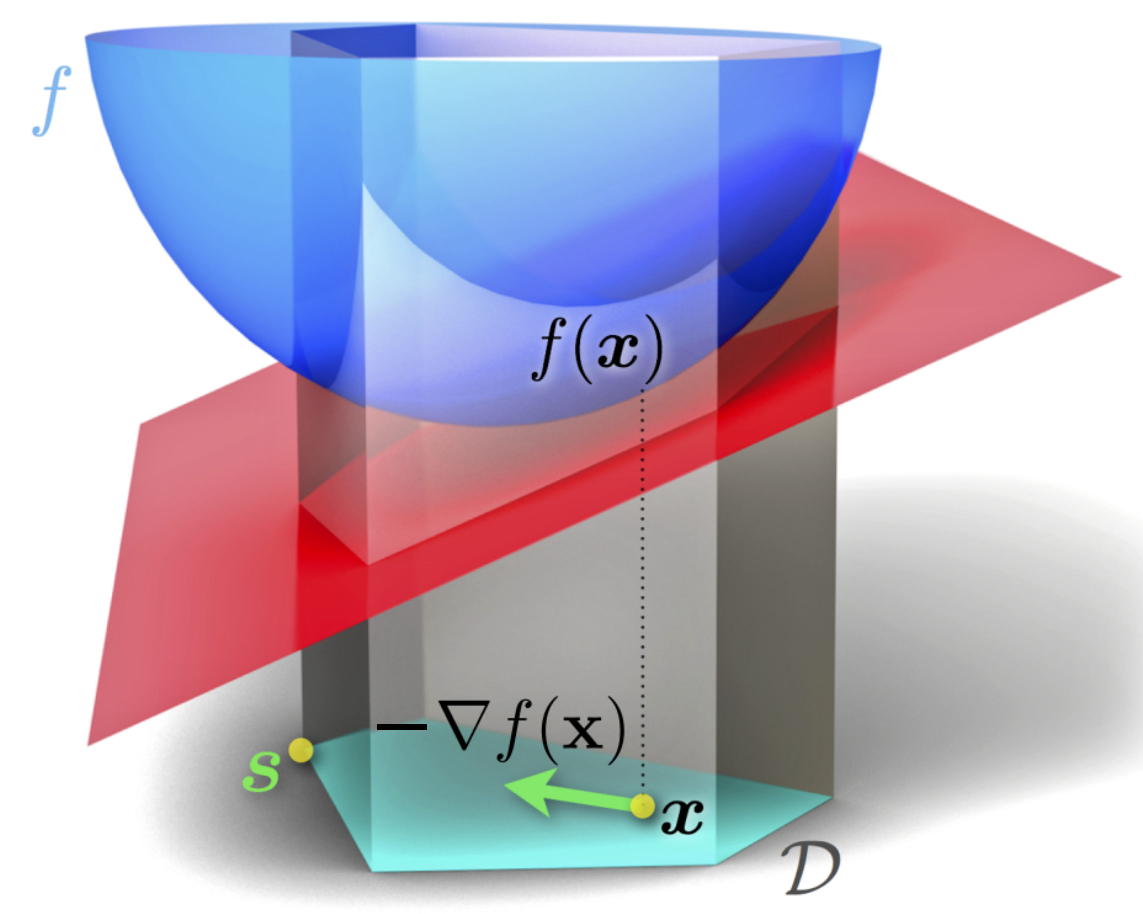
-> support point $s_k = \text{argmin}_{y \in \mathscr{D}} < y, \nabla f(x_k) >$

Step 3: Move towards s_k and repeat steps 1-3



The Frank-Wolfe algorithm

$\min_{x \in \mathcal{D}} f(x)$ f convex, $\mathcal{D}$ convex



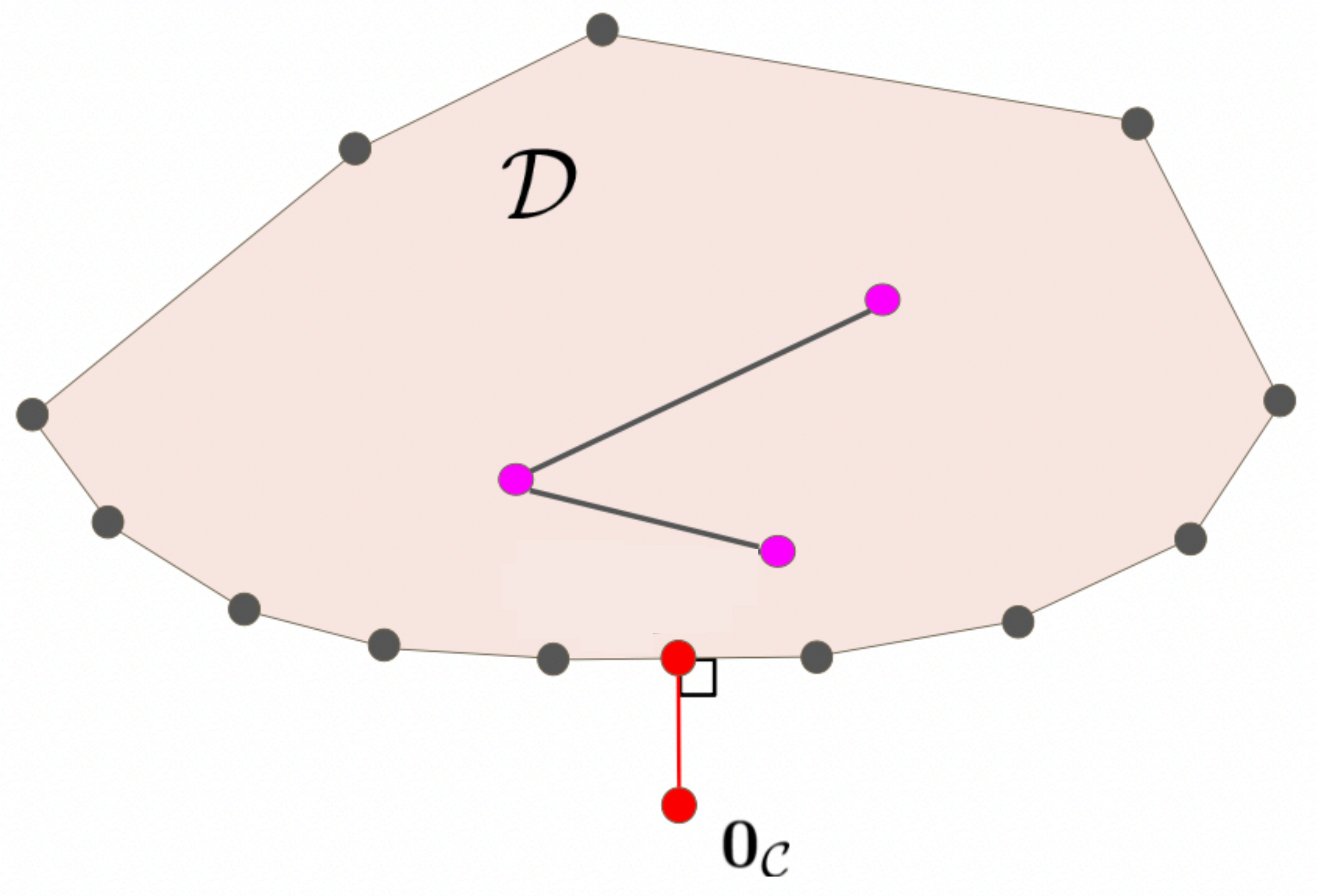
Frank-Wolfe = “constrained gradient descent”:

Step 1: Compute gradient $\nabla f(x_k)$ at current iterate x_k

Step 2: Compute point $s_k \in \mathcal{D}$ “most” in direction $-\nabla f(x_k)$

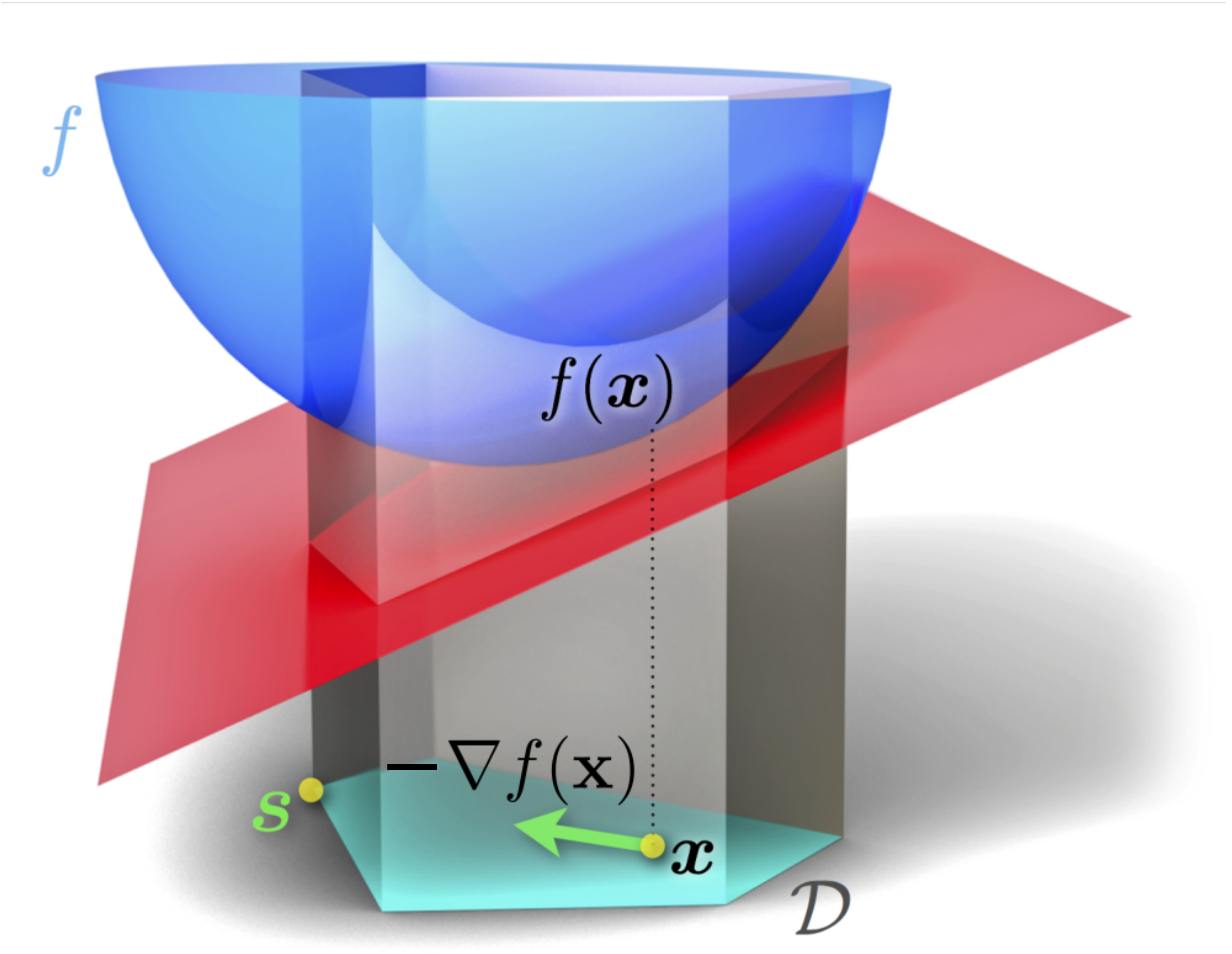
-> support point $s_k = \text{argmin}_{y \in \mathcal{D}} < y, \nabla f(x_k) >$

Step 3: Move towards s_k and repeat steps 1-3



The Frank-Wolfe algorithm

$\min_{x \in \mathcal{D}} f(x)$ $\quad f$ convex, $\mathcal{D}$ convex



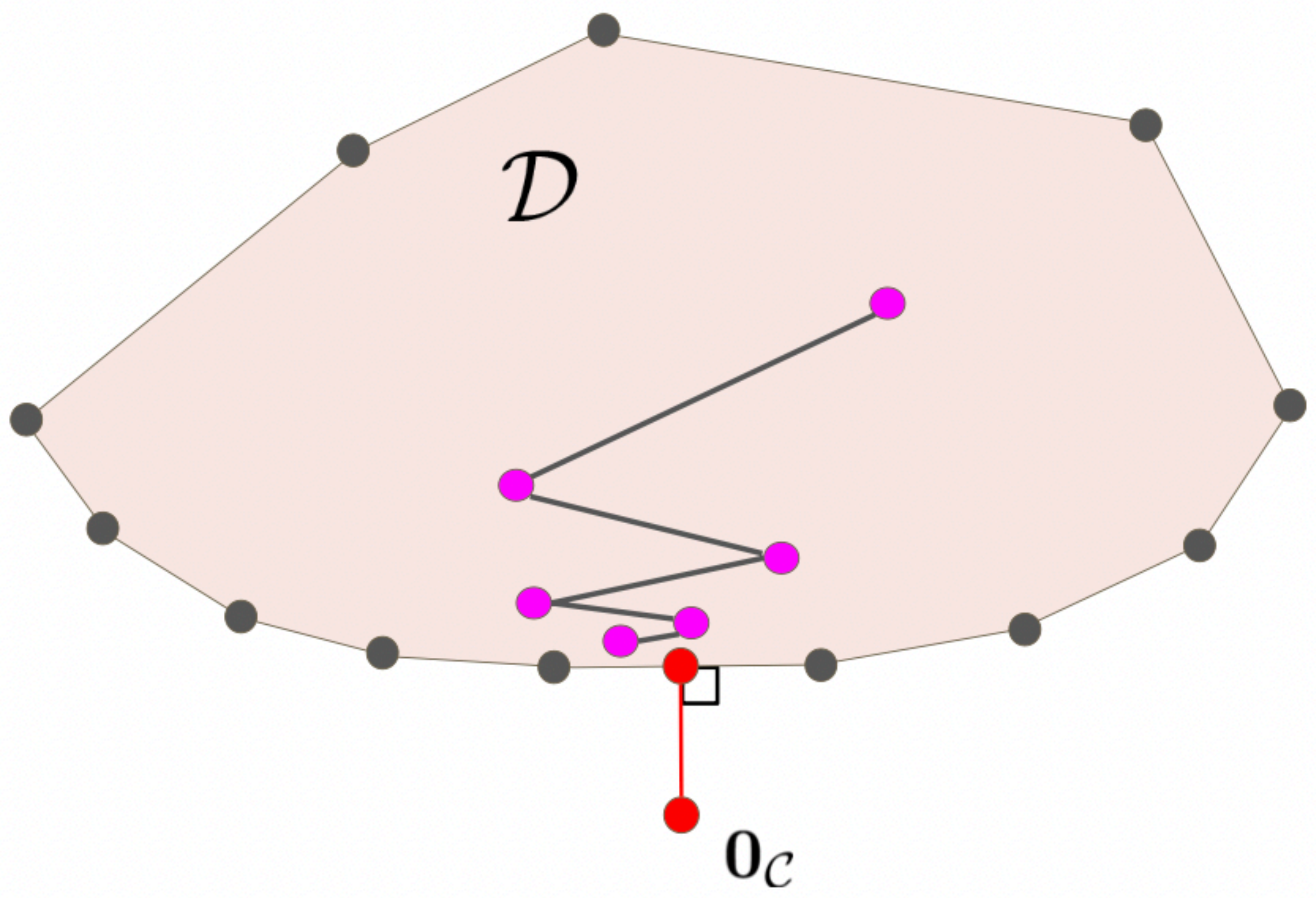
Frank-Wolfe = “constrained gradient descent”:

<u>Step 1:</u> Compute gradient $\nabla f(x_k)$ at current iterate x_k

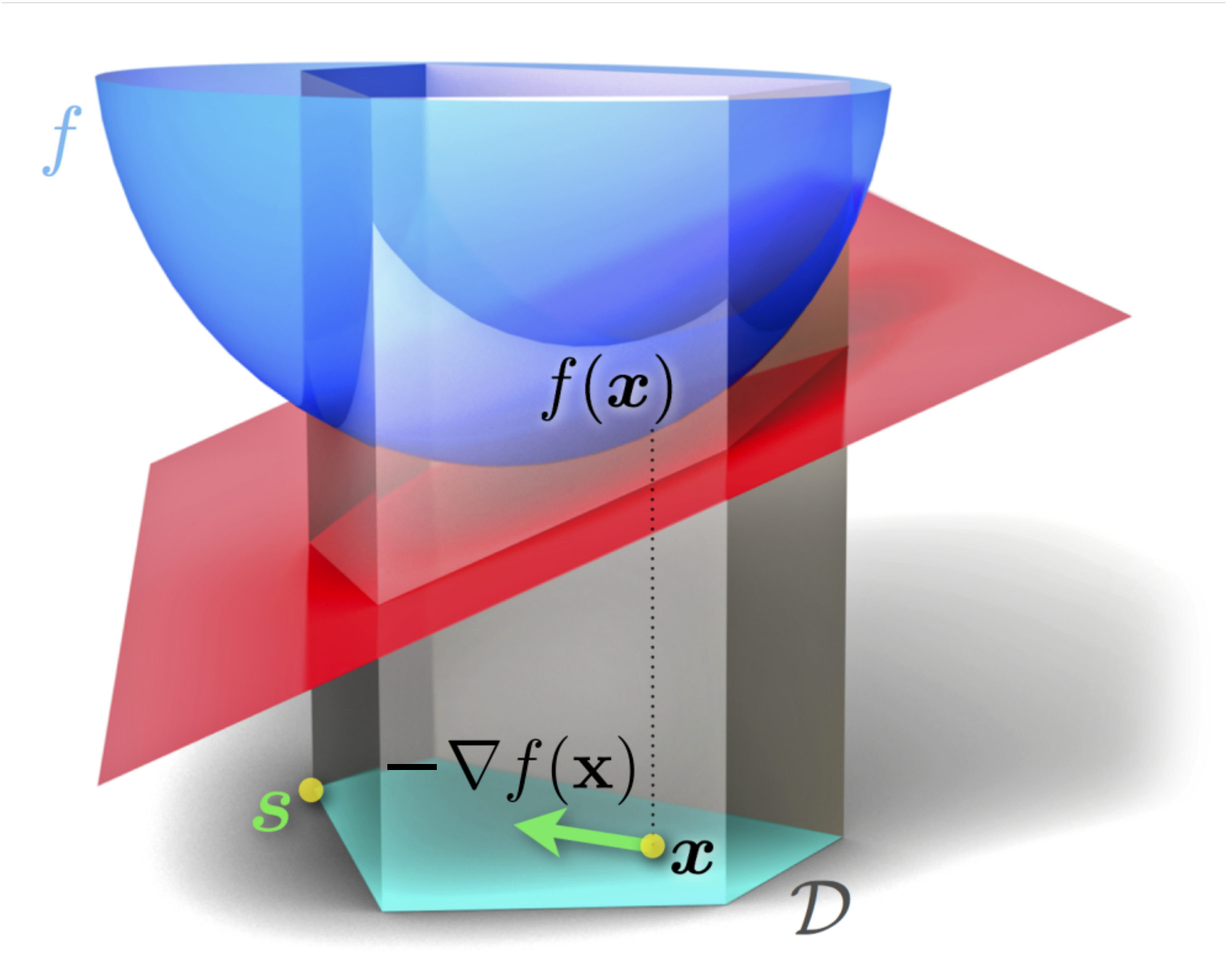
<u>Step 2:</u> Compute point $s_k \in \mathcal{D}$ “most” in direction $-\nabla f(x_k)$

-> support point $s_k = \text{argmin}_{y \in \mathcal{D}} < y, \nabla f(x_k) >$

<u>Step 3:</u> Move towards s_k and repeat steps 1-3



The Frank-Wolfe algorithm



$\min_{x \in \mathcal{D}} f(x)$ f convex, $\mathcal{D}$ convex

Frank-Wolfe = “constrained gradient descent”:

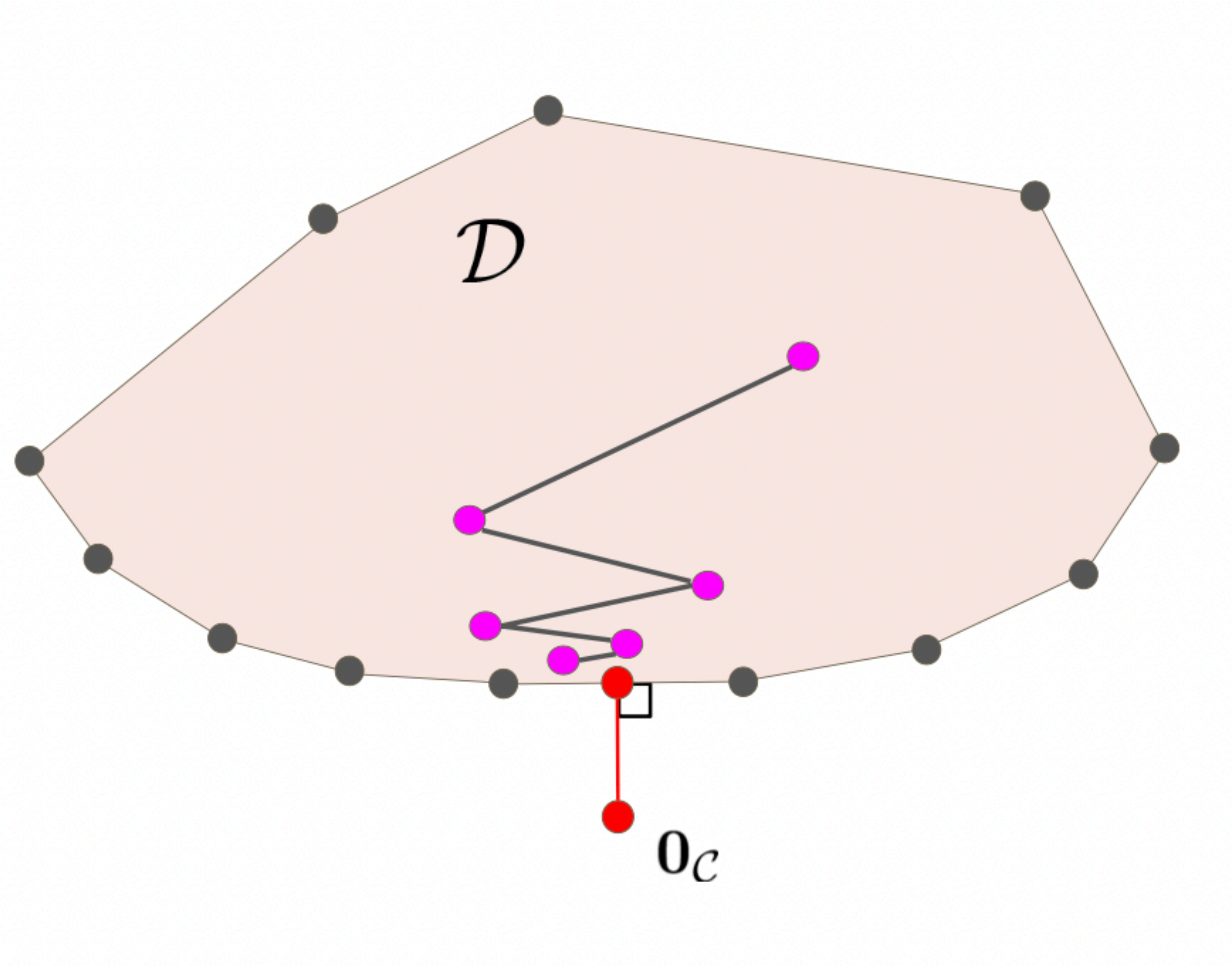
Step 1: Compute gradient $\nabla f(x_k)$ at current iterate x_k

Step 2: Compute point $s_k \in \mathcal{D}$ “most” in direction $-\nabla f(x_k)$

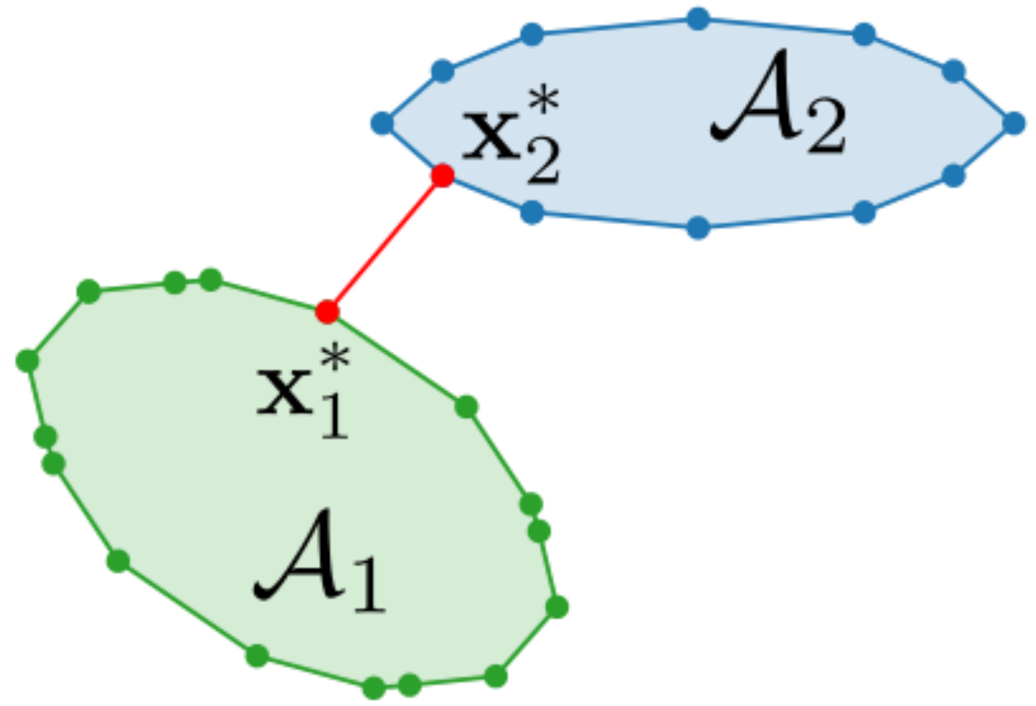
-> support point $s_k = \text{argmin}_{y \in \mathcal{D}} < y, \nabla f(x_k) >$

Step 3: Move towards s_k and repeat steps 1-3

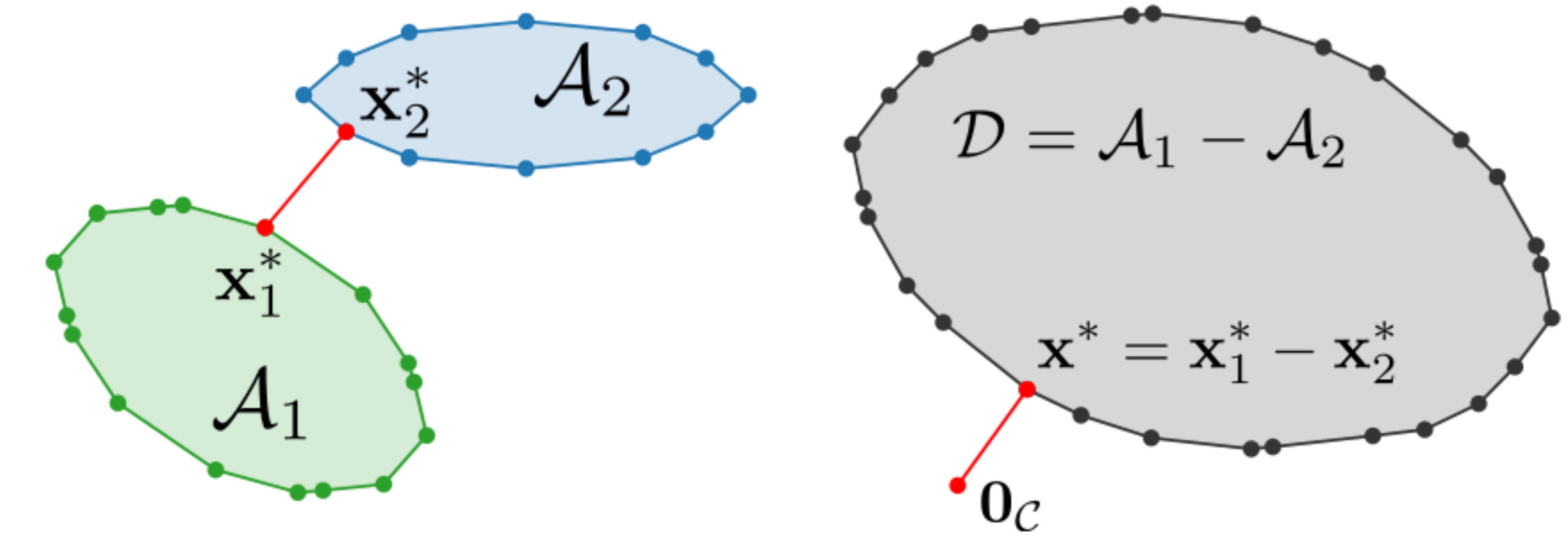
?



Recap of collision detection with Frank-Wolfe



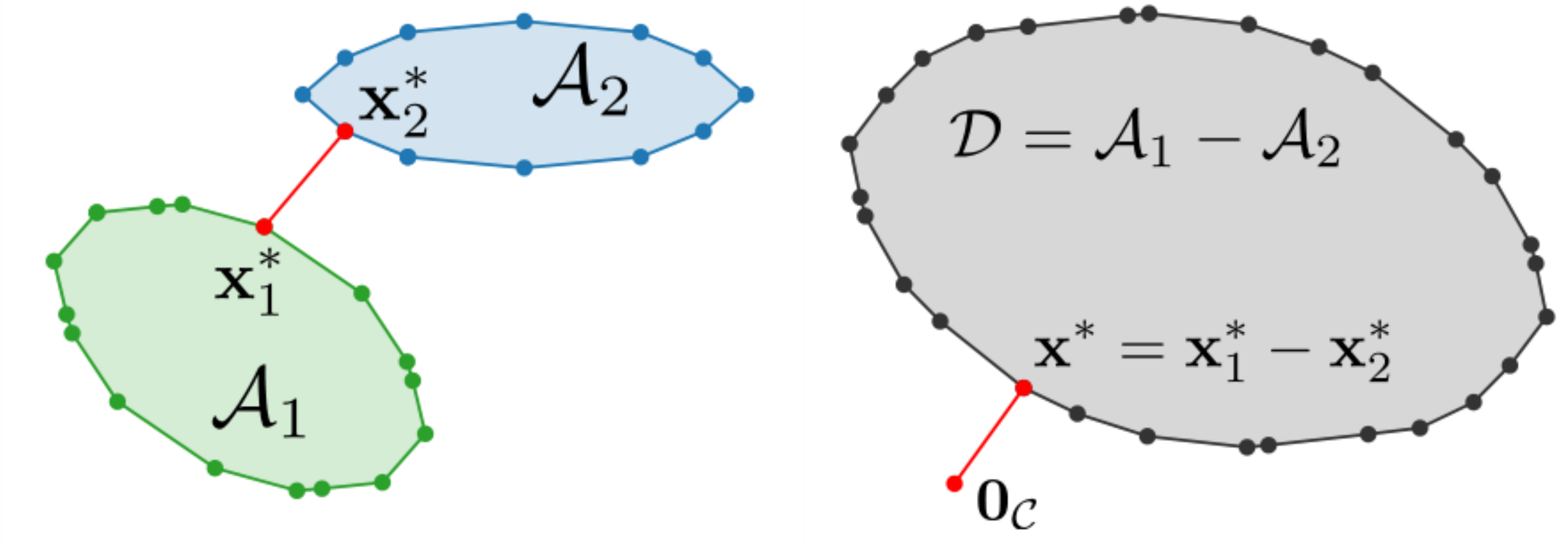
Recap of collision detection with Frank-Wolfe



$$\min_{x_1 \in \mathscr{A}_1, x_2 \in \mathscr{A}_2} \frac{1}{2} || x_1 - x_2 ||^2 \longrightarrow \min_{x \in \mathscr{D}} \frac{1}{2} || x ||^2$$

MNP

Recap of collision detection with Frank-Wolfe



$$\min_{x_1 \in \mathcal{A}_1, x_2 \in \mathcal{A}_2} \frac{1}{2} ||x_1 - x_2||^2 \longrightarrow \boxed{\min_{x \in \mathcal{D}} \frac{1}{2} ||x||^2}$$

MNP

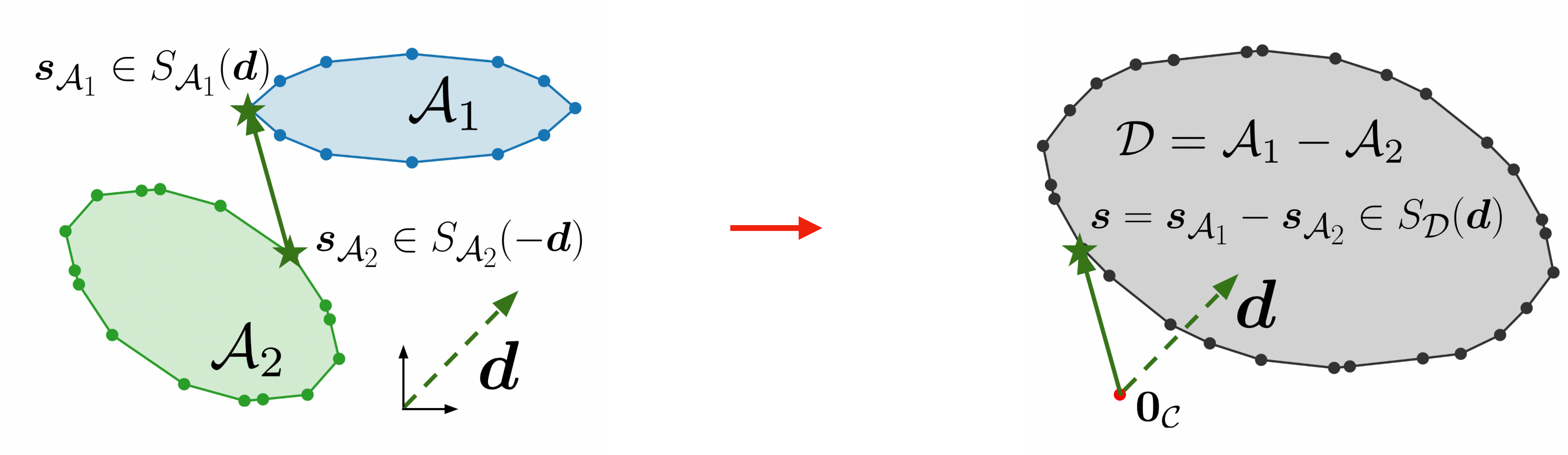
Frank-Wolfe = "constrained gradient descent", needs to compute support points:

$$\boxed{s = \text{argmin}_{y \in \mathcal{D}} < y, \nabla f(x) >}$$

Computing support points on a Minkowski difference

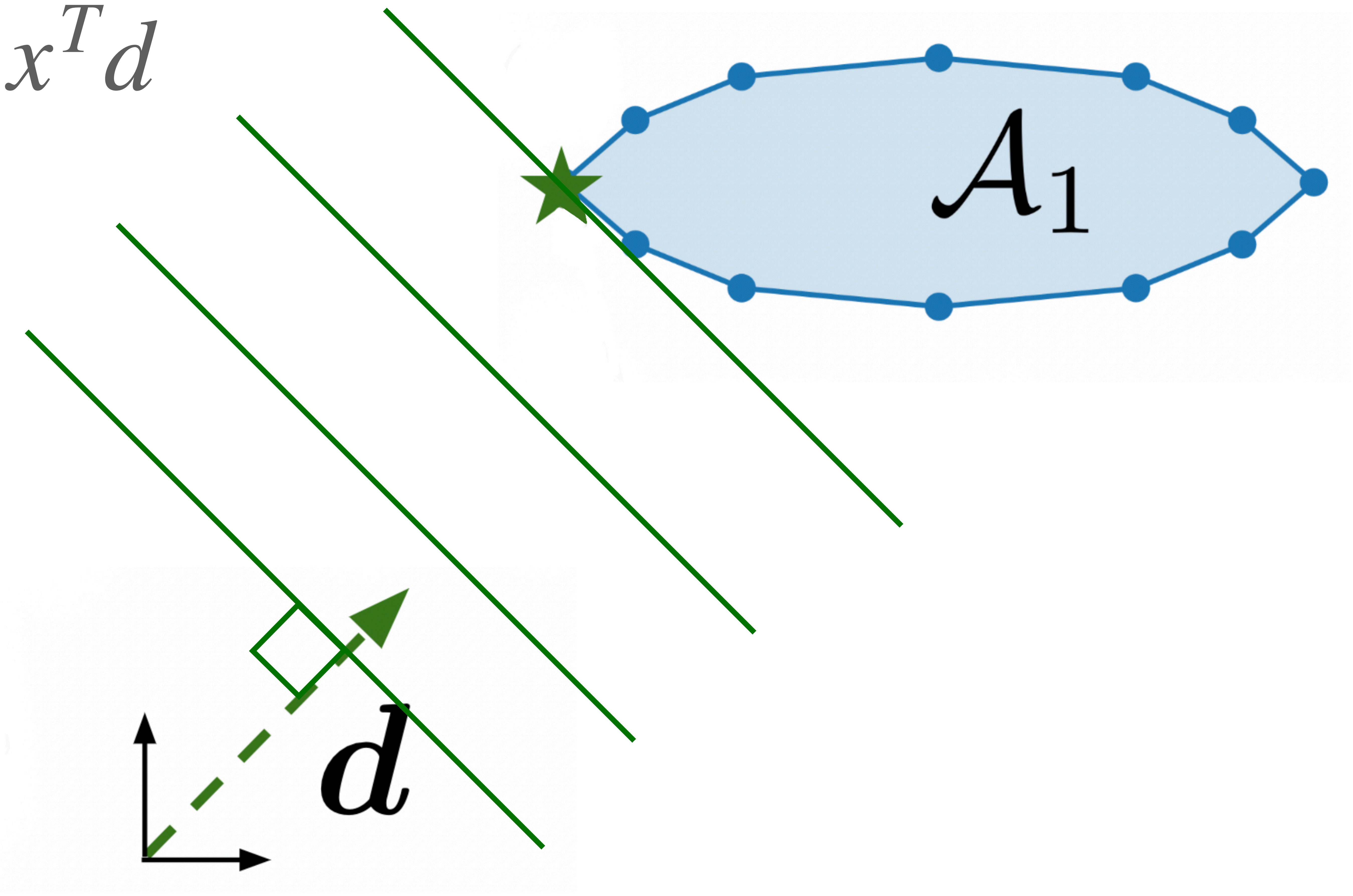
$$S_{\mathscr{A}}(d) = \underset{x \in \mathscr{A}}{\text{argmin}}\ x^T d$$

$$\begin{aligned} s_1 &\in S_{\mathscr{A}_1}(d) \\ s_2 &\in S_{\mathscr{A}_2}(-d) \end{aligned} \longrightarrow s = s_1 - s_2 \in S_{\mathscr{D}}(d)$$



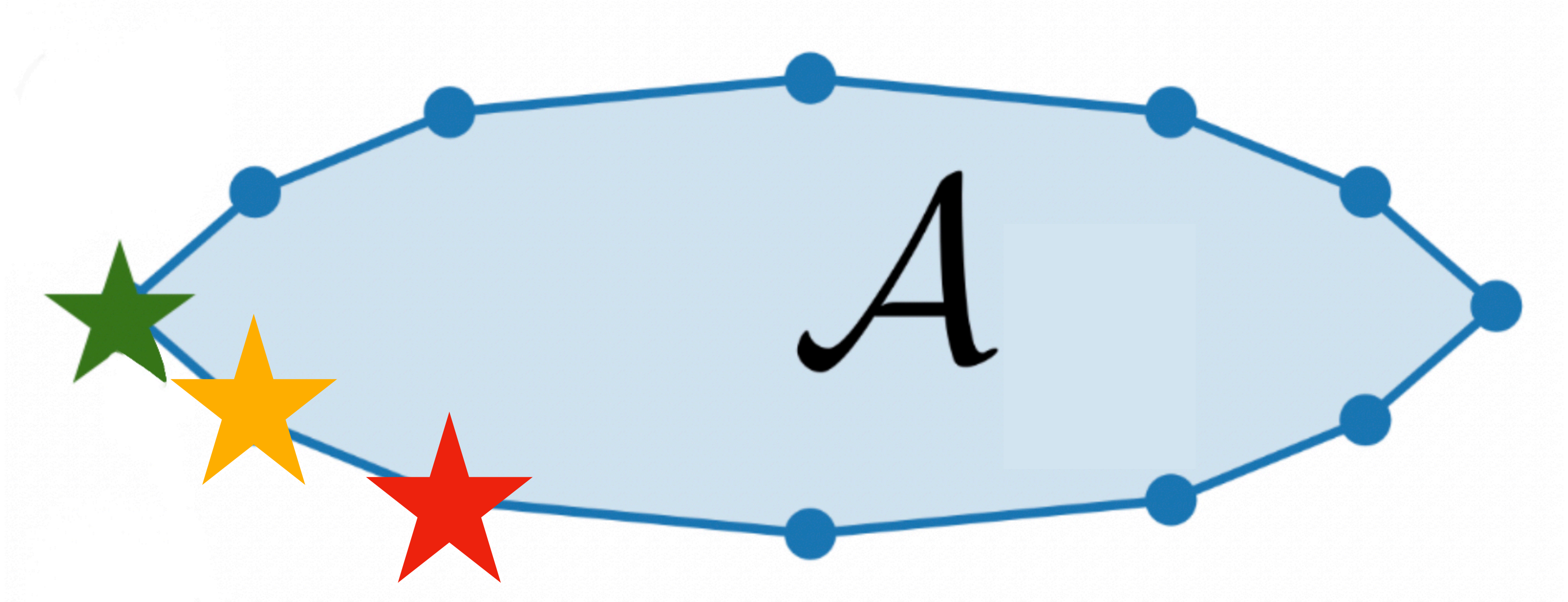
Computing support points on shapes

$$S_{\mathscr{A}}(d) = \underset{x \in \mathscr{A}}{\text{argmin}}\ x^T d$$

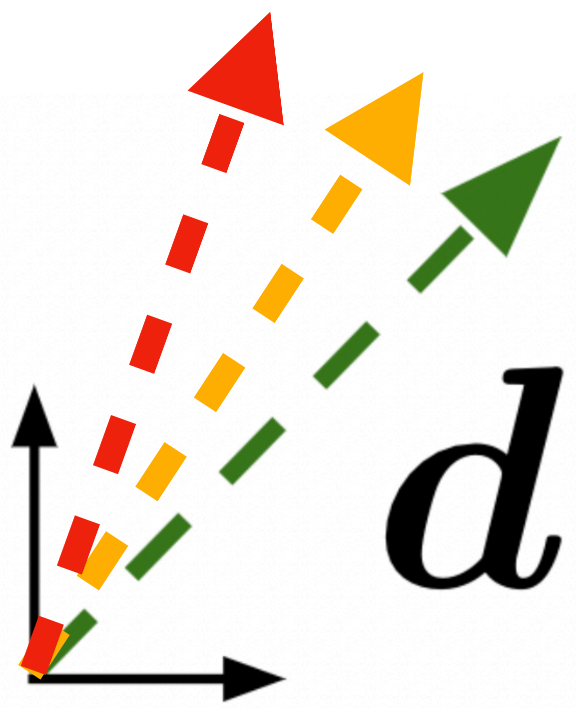


Computing support points on shapes

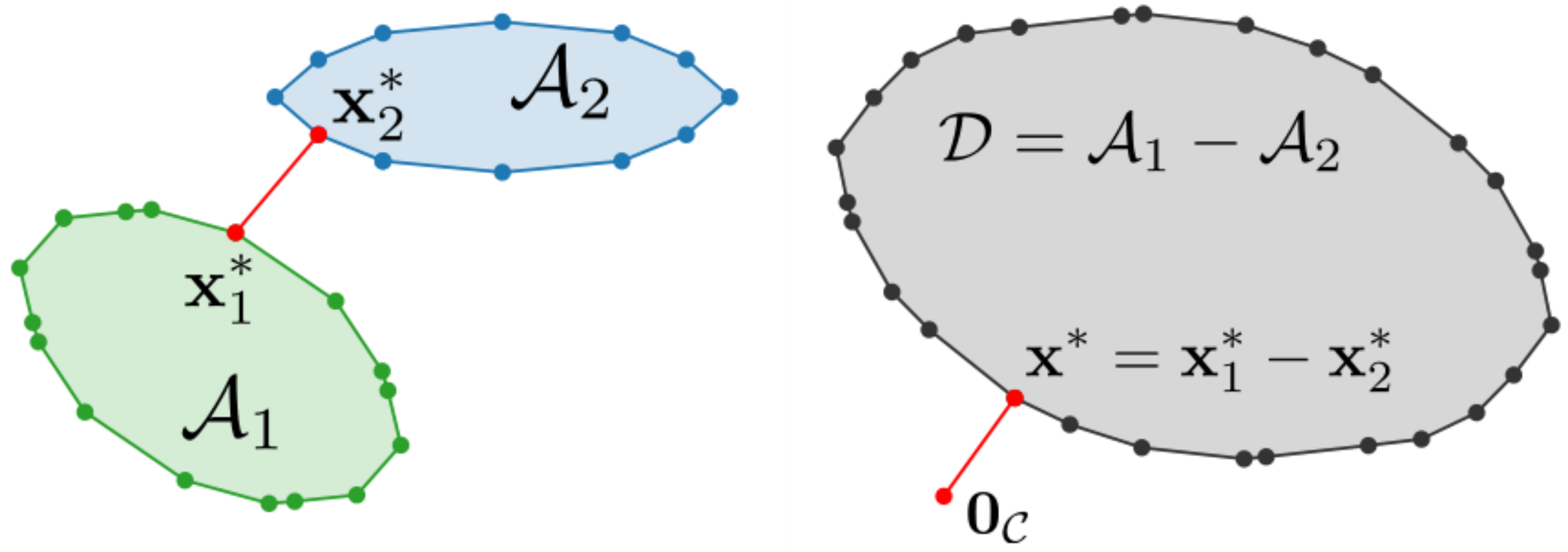
$$S_{\mathscr{A}}(d) = \underset{x \in \mathscr{A}}{\text{argmin}}\ x^T d$$



Can be computed very efficiently for most shapes



Recap of collision detection with Frank-Wolfe



$$\min_{x_1 \in \mathcal{A}_1, x_2 \in \mathcal{A}_2} \frac{1}{2} ||x_1 - x_2||^2 \longrightarrow \min_{x \in \mathcal{D}} \frac{1}{2} ||x||^2$$

MNP

Frank-Wolfe = "constrained gradient descent", needs to compute support points:

$$s = \text{argmin}_{y \in \mathcal{D}} < y, \nabla f(x) >$$

Frank-Wolfe zigzags

Algorithm Frank-Wolfe

Let $\boldsymbol{x}_0 \in \mathcal{D}$, $\epsilon > 0$

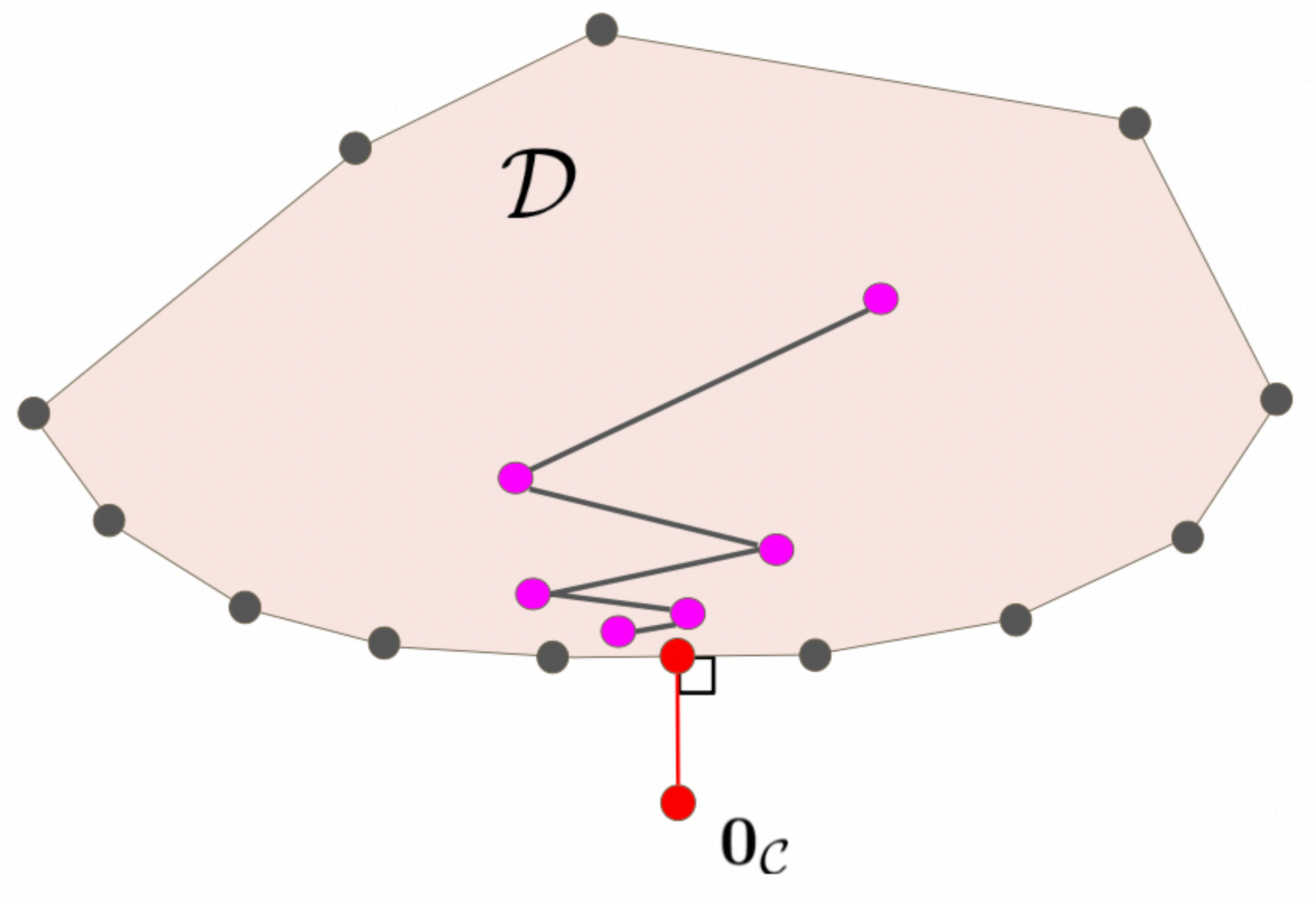
For `k=0, 1, ...` **do**

1: $\boldsymbol{s}_k \in \arg\min_{\boldsymbol{s}\in\mathcal{D}} \langle \boldsymbol{\nabla} f(\boldsymbol{x}_k), \boldsymbol{s}\rangle$ ▷ Support

2: **If** $g_{FW}(\boldsymbol{x}_k) \leq \epsilon$, **return** $f(\boldsymbol{x}_k)$ ▷ Duality gap

3: $\gamma_k = \arg\min_{\gamma\in[0,1]} f(\gamma \boldsymbol{x}_k + (1-\gamma)\boldsymbol{s}_k)$ ▷ Linesearch

4: $\boldsymbol{x}_{k+1} = \gamma_k \boldsymbol{x}_k + (1-\gamma_k)\boldsymbol{s}_k$ ▷ Update iterate



From Frank-Wolfe to GJK

Algorithm Frank-Wolfe

Let $x_0 \in \mathcal{D}$, $\epsilon > 0$

For `k=0, 1, ...` **do**

1: $\boldsymbol{s}_k \in \arg\min_{\boldsymbol{s} \in \mathcal{D}} \langle \nabla f(\boldsymbol{x}_k), \boldsymbol{s} \rangle$ ▷ Support

2: **If** $g_{FW}(\boldsymbol{x}_k) \leq \epsilon$, **return** $f(\boldsymbol{x}_k)$ ▷ Duality gap

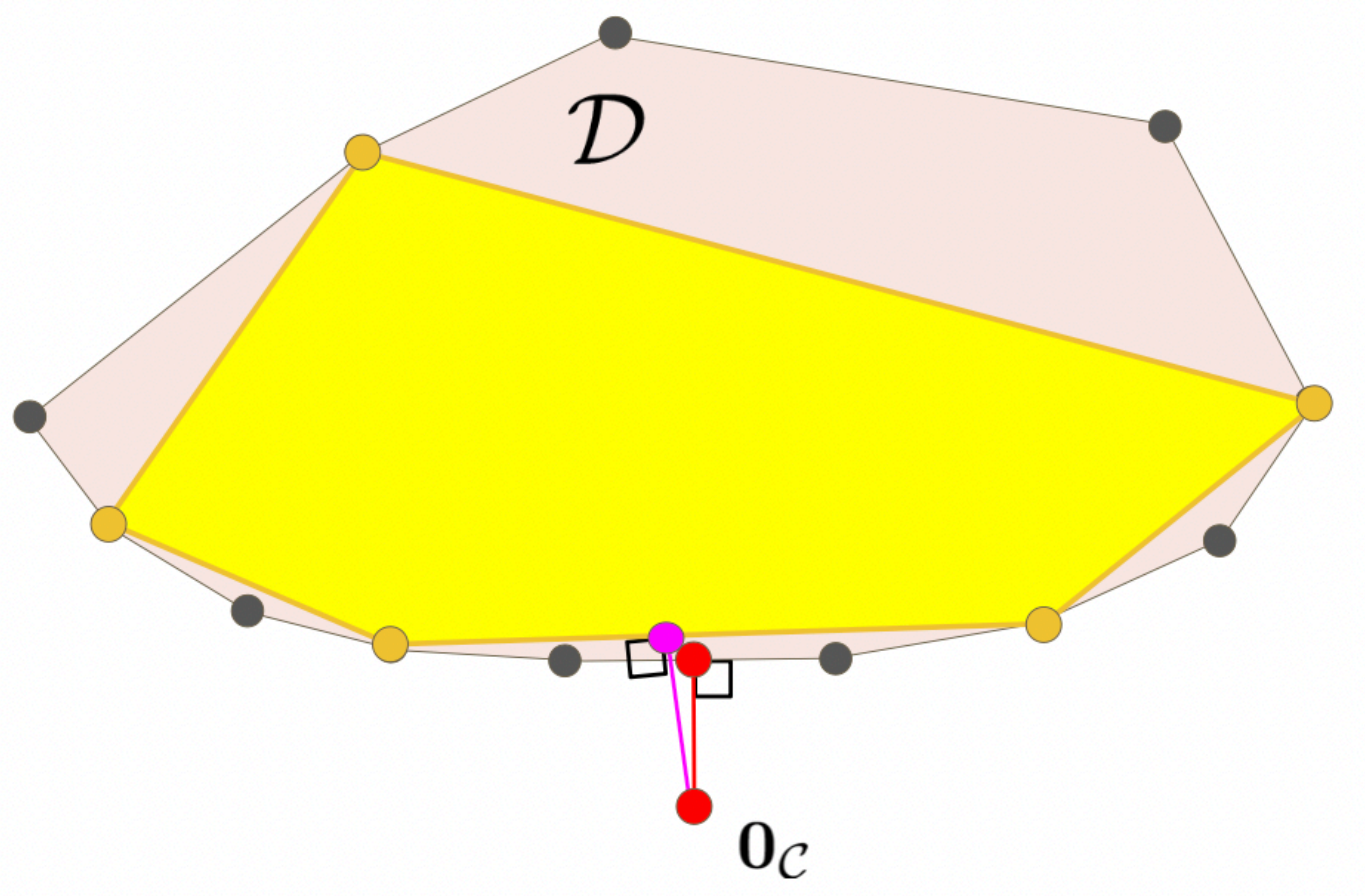
3: $\gamma_k = \arg\min_{\gamma \in [0,1]} f(\gamma \boldsymbol{x}_k + (1-\gamma)\boldsymbol{s}_k)$ ▷ Linesearch

4: $\boldsymbol{x}_{k+1} = \gamma_k \boldsymbol{x}_k + (1-\gamma_k)\boldsymbol{s}_k$ ▷ Update iterate

Algorithm Fully-Corrective Frank-Wolfe

In Frank-Wolfe, replace line 3 and 4 by:

1: $\boldsymbol{x}_{k+1} = \arg\min_{\boldsymbol{x} \in \text{conv}(\boldsymbol{s}_0, \ldots, \boldsymbol{s}_{k-1})} f(\boldsymbol{x})$



From Frank-Wolfe to GJK

Algorithm Frank-Wolfe

Let $x_0 \in \mathcal{D}$, $\epsilon > 0$

For `k=0, 1, ...` **do**

1: $\boldsymbol{s}_k \in \arg\min_{\boldsymbol{s} \in \mathcal{D}} \langle \nabla f(\boldsymbol{x}_k), \boldsymbol{s} \rangle$ ▷ Support

2: **If** $g_{FW}(\boldsymbol{x}_k) \leq \epsilon$, **return** $f(\boldsymbol{x}_k)$ ▷ Duality gap

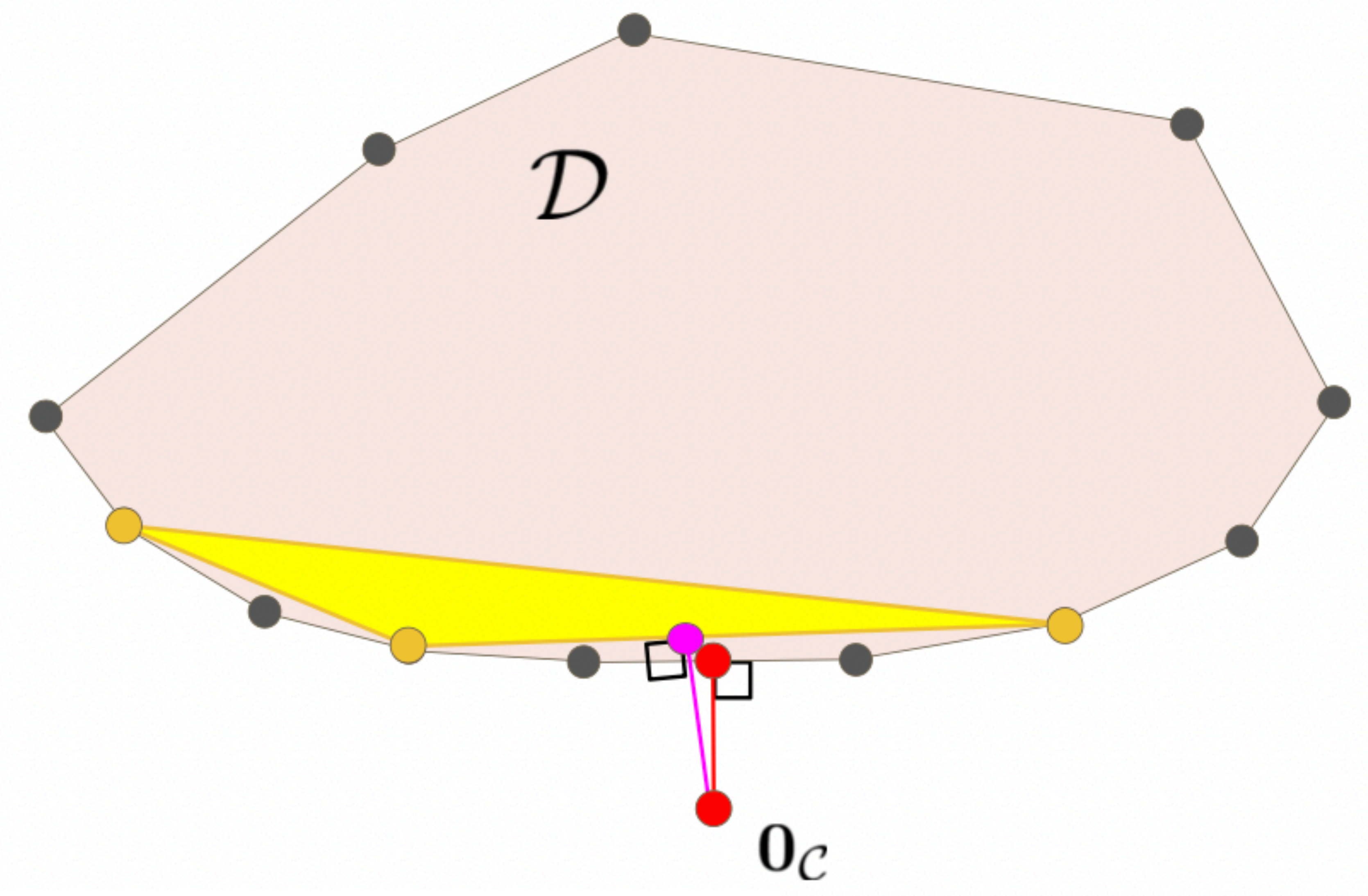
3: $\gamma_k = \arg\min_{\gamma \in [0,1]} f(\gamma \boldsymbol{x}_k + (1-\gamma)\boldsymbol{s}_k)$ ▷ Linesearch

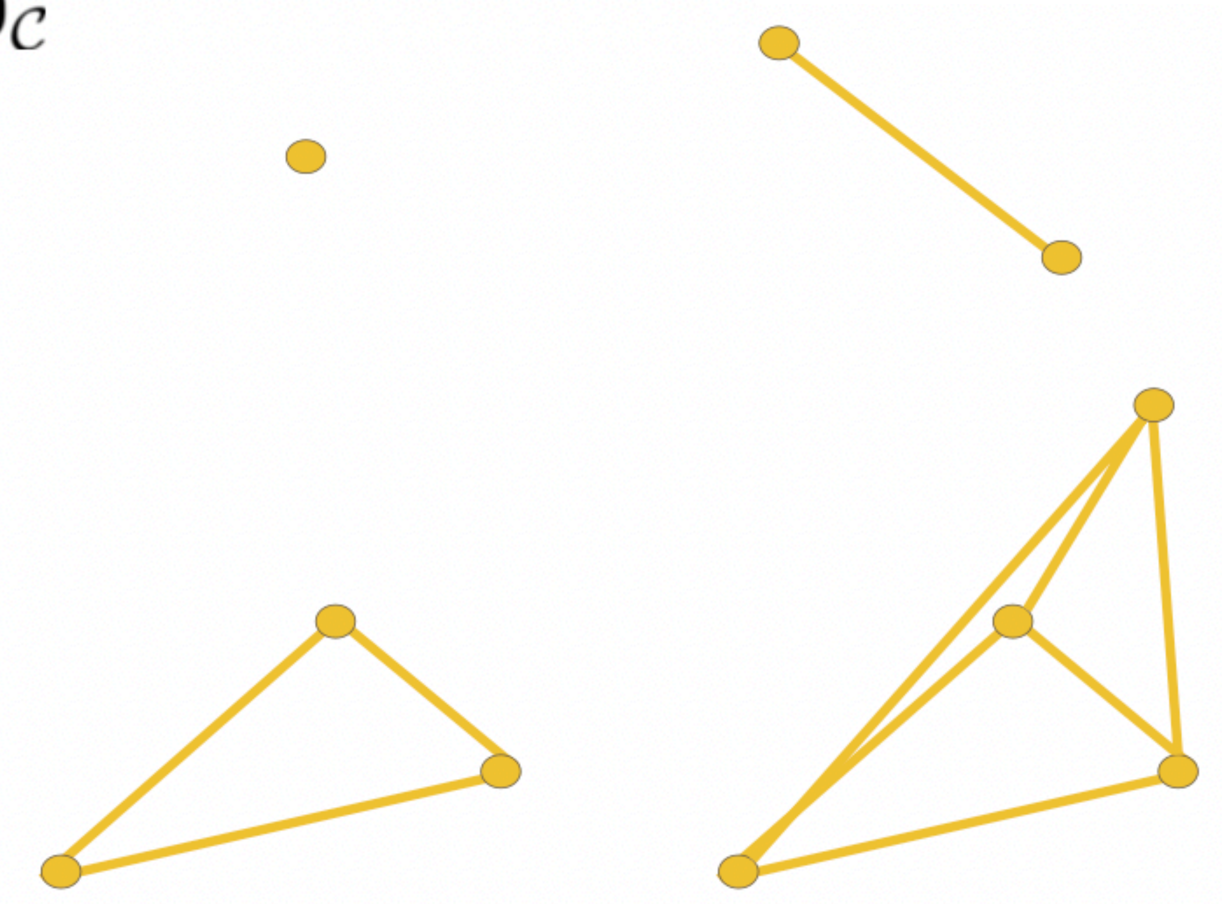
4: $\boldsymbol{x}_{k+1} = \gamma_k \boldsymbol{x}_k + (1-\gamma_k)\boldsymbol{s}_k$ ▷ Update iterate

Algorithm Fully-Corrective Frank-Wolfe

In Frank-Wolfe, replace line 3 and 4 by:

1: $\boldsymbol{x}_{k+1} = \arg\min_{\boldsymbol{x} \in \text{conv}(\boldsymbol{s}_0, \ldots, \boldsymbol{s}_{k-1})} f(\boldsymbol{x})$





Optimal solution can be described by a simplex

Nesterov accelerated Frank-Wolfe (or GJK)

Algorithm Frank-Wolfe

Let $x_0 \in \mathcal{D}$, $\epsilon > 0$

For `k=0, 1, ...` **do**

1: $\boldsymbol{s}_k \in \arg\min_{\boldsymbol{s} \in \mathcal{D}} \langle \nabla f(\boldsymbol{x}_k), \boldsymbol{s} \rangle$ ▷ Support

2: **If** $g_{FW}(\boldsymbol{x}_k) \leq \epsilon$, **return** $f(\boldsymbol{x}_k)$ ▷ Duality gap

3: $\gamma_k = \arg\min_{\gamma \in [0,1]} f(\gamma \boldsymbol{x}_k + (1-\gamma)\boldsymbol{s}_k)$ ▷ Linesearch

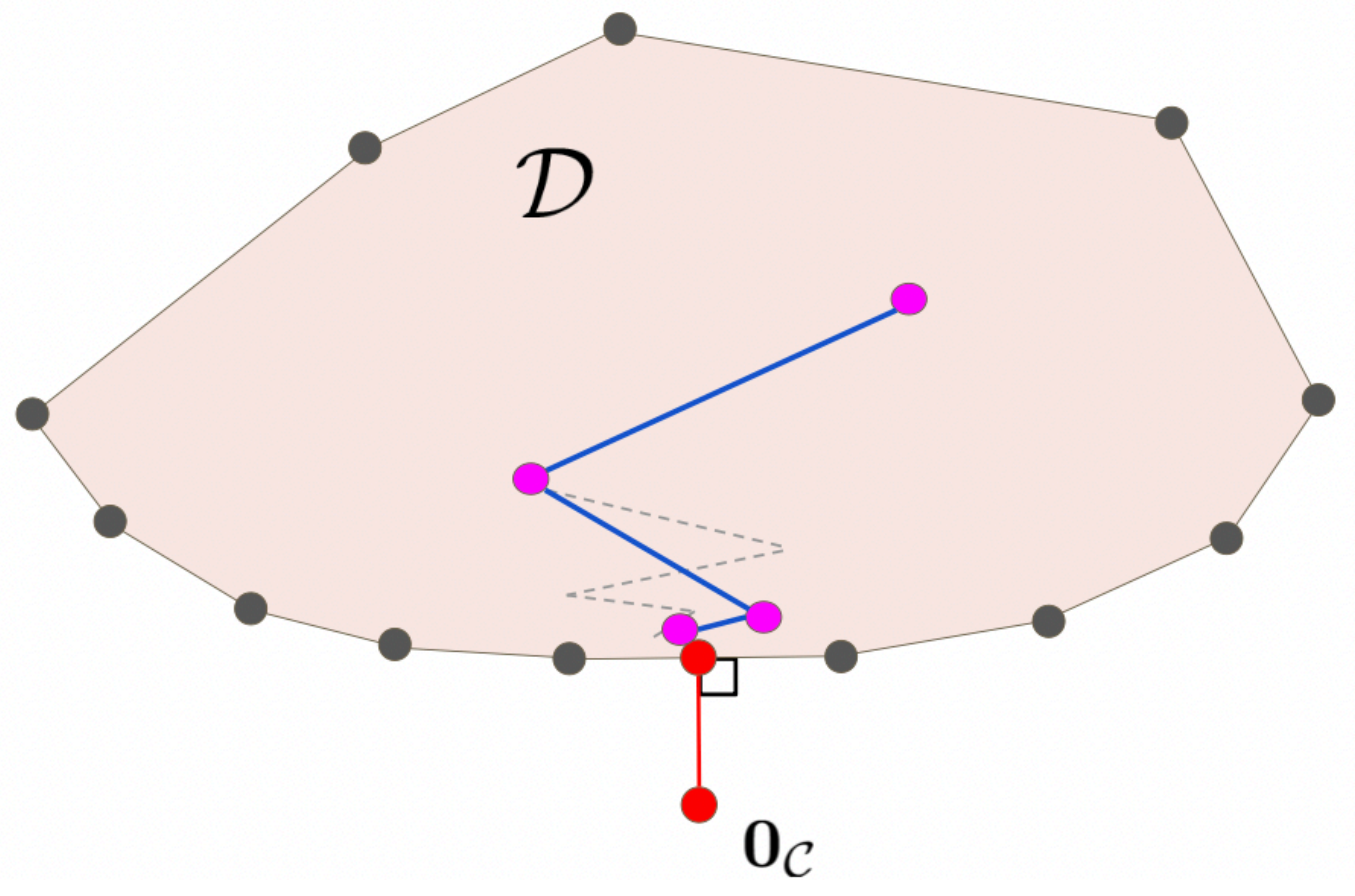
4: $\boldsymbol{x}_{k+1} = \gamma_k \boldsymbol{x}_k + (1-\gamma_k)\boldsymbol{s}_k$ ▷ Update iterate

Algorithm Nesterov-accelerated Frank-Wolfe

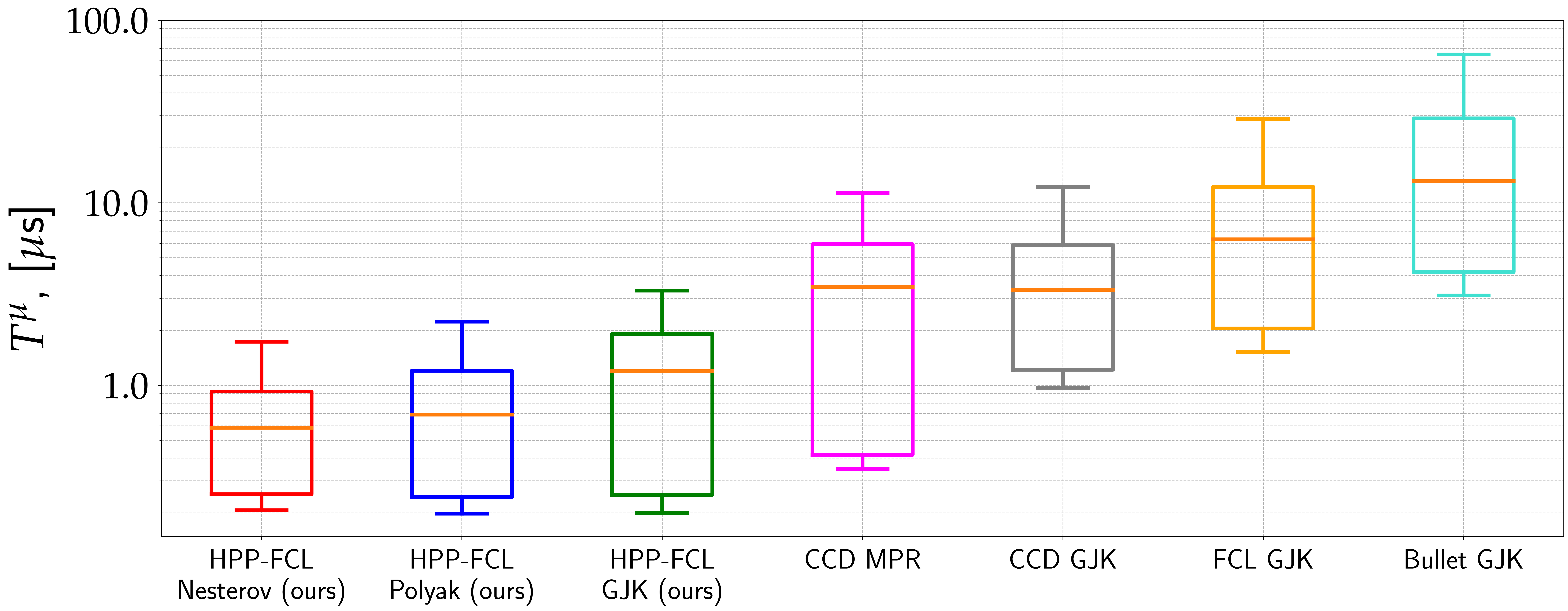
In Frank-Wolfe, let $\boldsymbol{d}_{-1} = \boldsymbol{s}_{-1} = \boldsymbol{x}_0$, $\delta_k = \frac{k+1}{k+3}$ *and replace line 1 by:*

1: $\boldsymbol{y}_k = \delta_k \boldsymbol{x}_k + (1-\delta_k)\boldsymbol{s}_{k-1}$

2: $\boldsymbol{d}_k = \delta_k \boldsymbol{d}_{k-1} + (1-\delta_k)\nabla f(\boldsymbol{y}_k)$

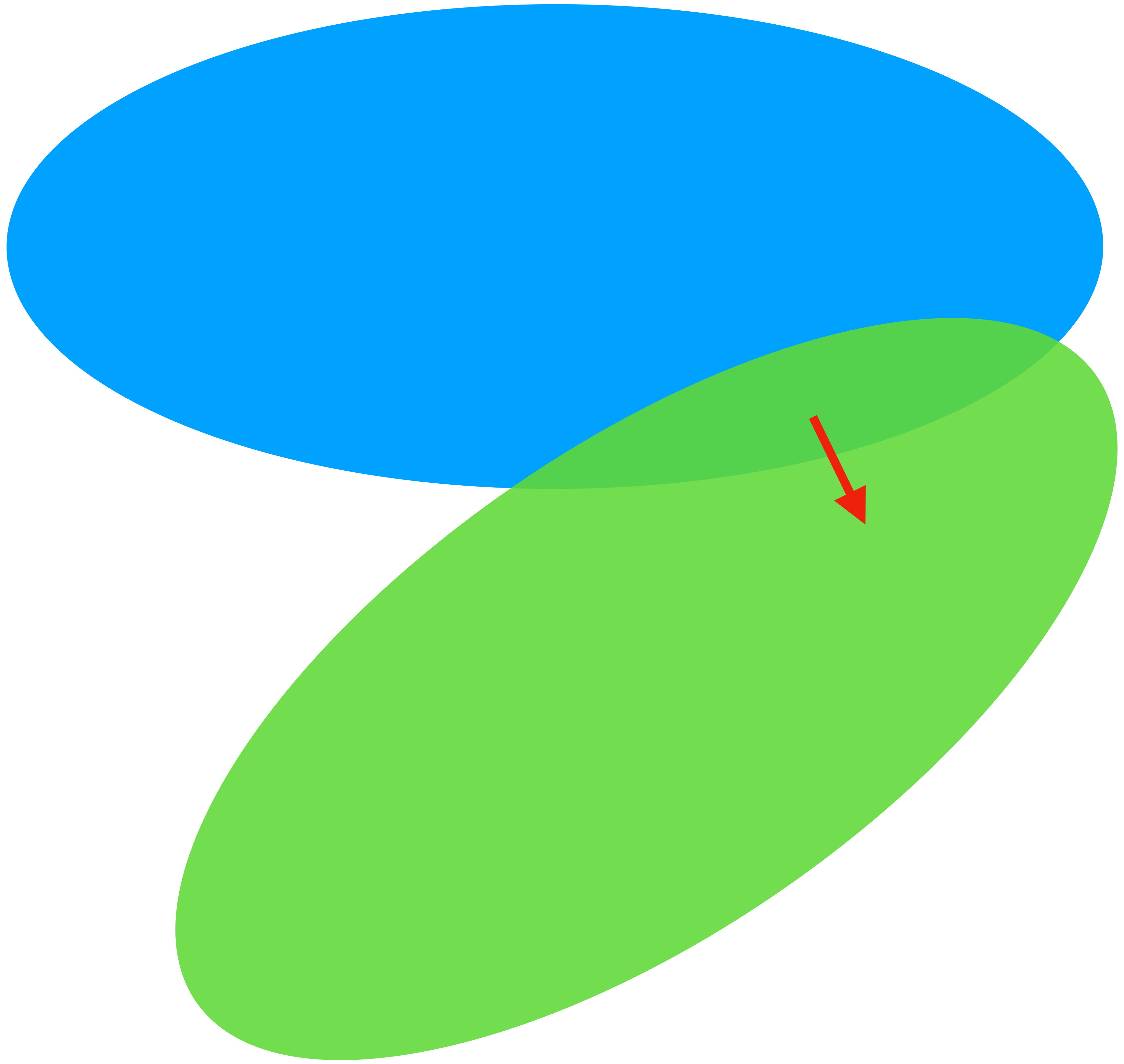


HPP-FCL vs. The Rest of The World



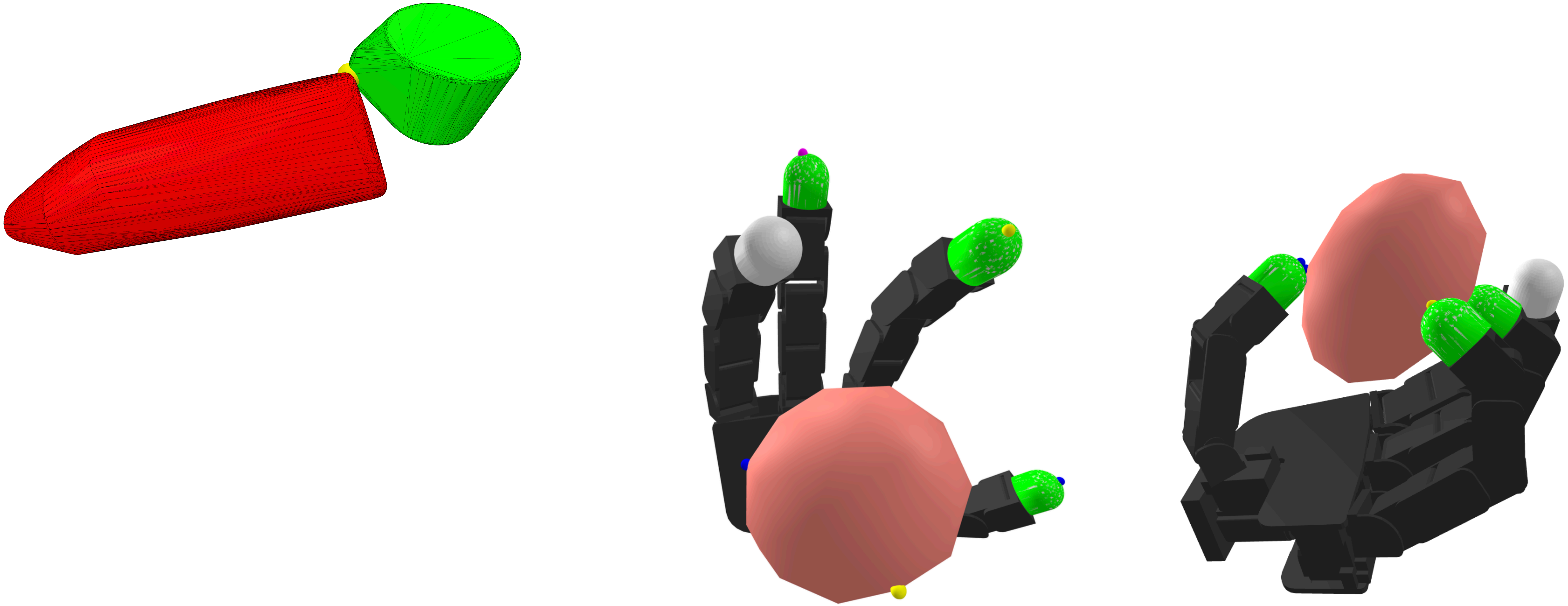
Expanding Polytope Algorithm - an extension of GJK

Separation vector:
Vector of smallest norm such that if shapes are translated by it, they don't overlap



Part III - Beyond Collision Detection: Differentiable Collision Detection

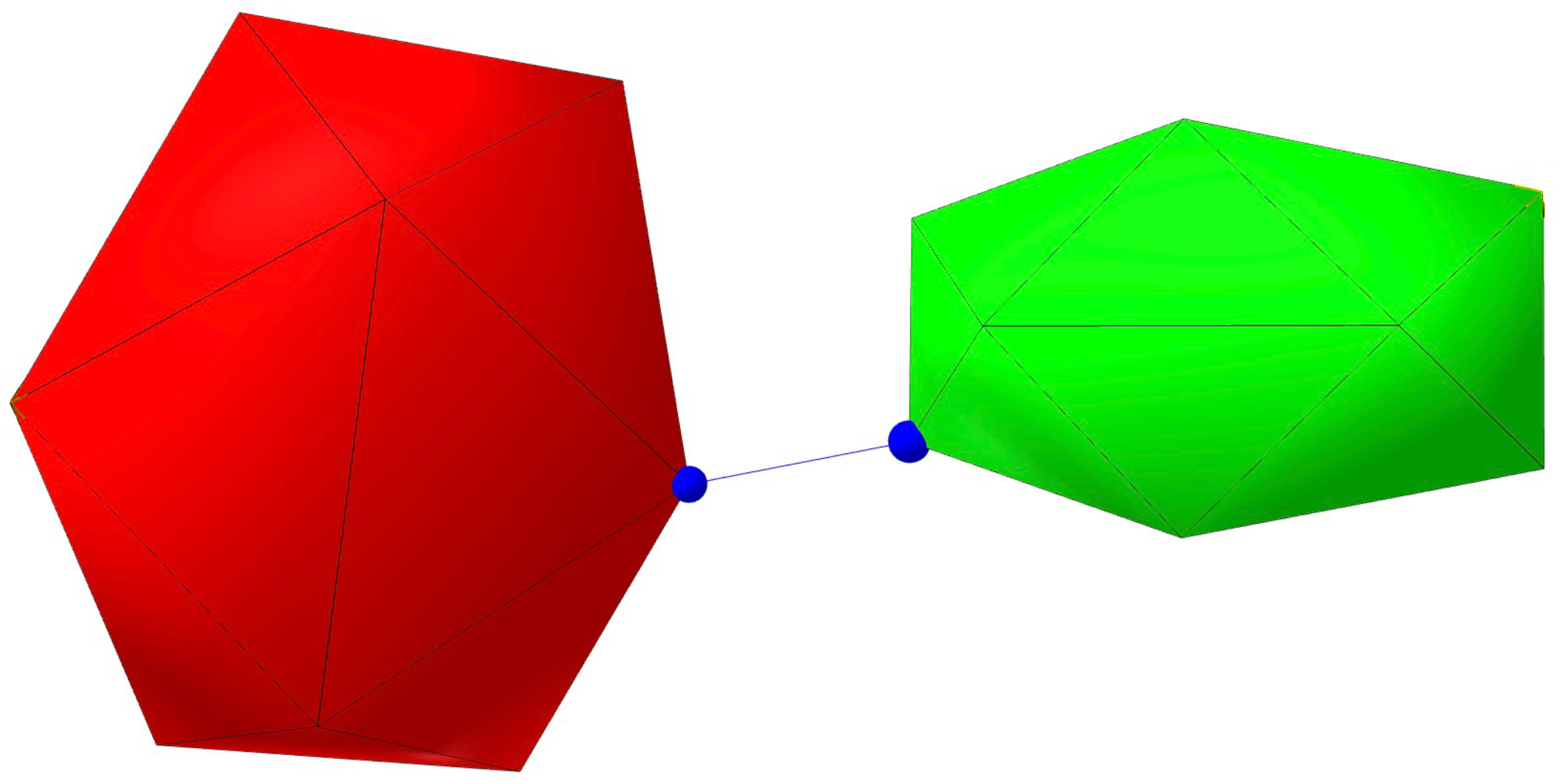
Witness points are a function of the shapes placements



Witness points are a function of the shapes placements

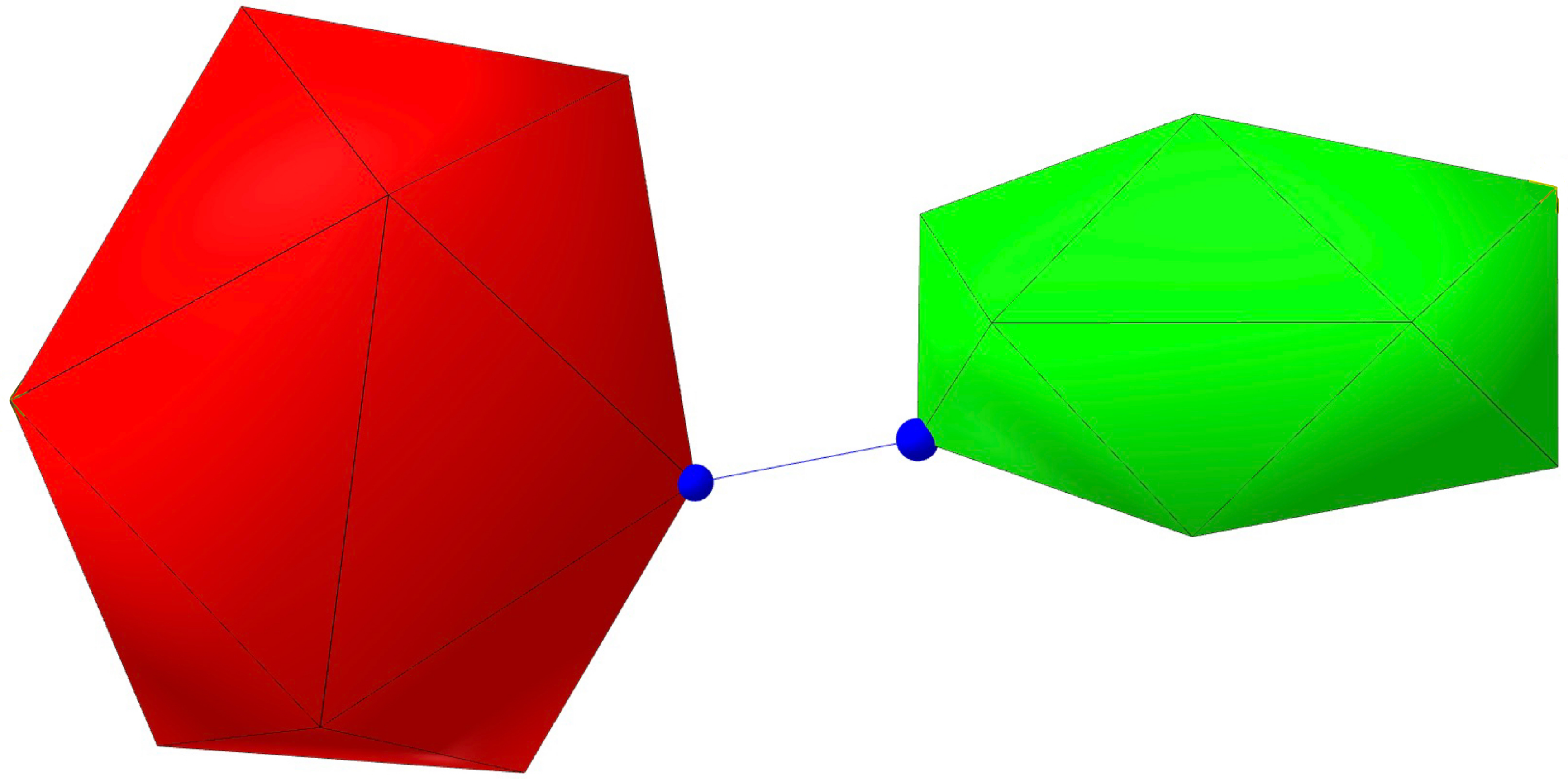
$$x_1^*(T), x_2^*(T) = \text{argmin} \, || x_1 - x_2 ||_2^2$$
$$\text{s.t. } x_1 \in \mathcal{A}_1, x_2 \in \mathcal{A}_2(T)$$

SOTA algos:
GJK + EPA



Witness points are a function of the shapes placements

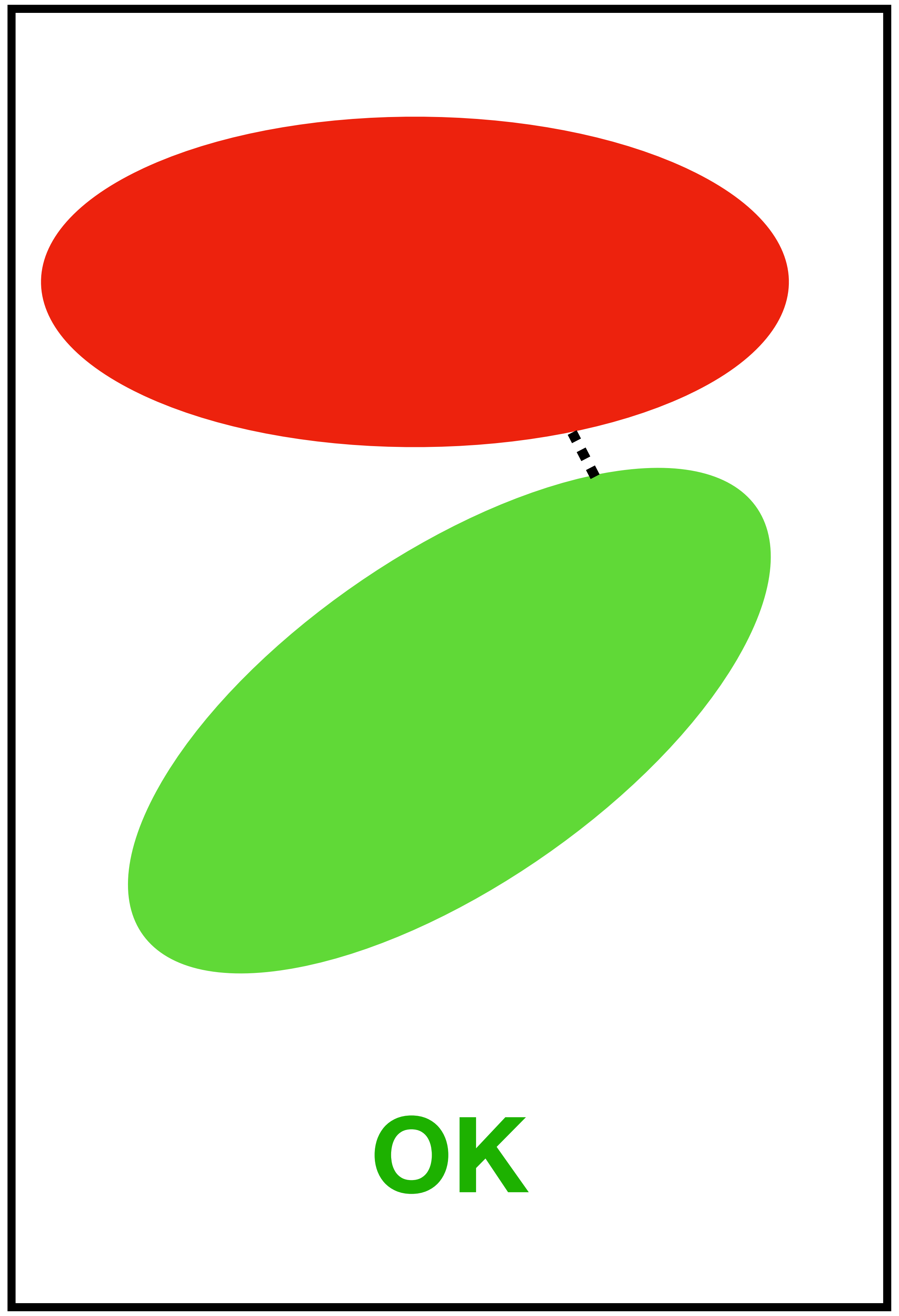
$$x_1^*(T), x_2^*(T) = \text{argmin}\,||x_1 - x_2||_2^2 \quad \text{s.t. } x_1 \in \mathcal{A}_1, x_2 \in \mathcal{A}_2(T) \longrightarrow \frac{\partial x_1^*(T)}{\partial T}, \frac{\partial x_2^*(T)}{\partial T}$$

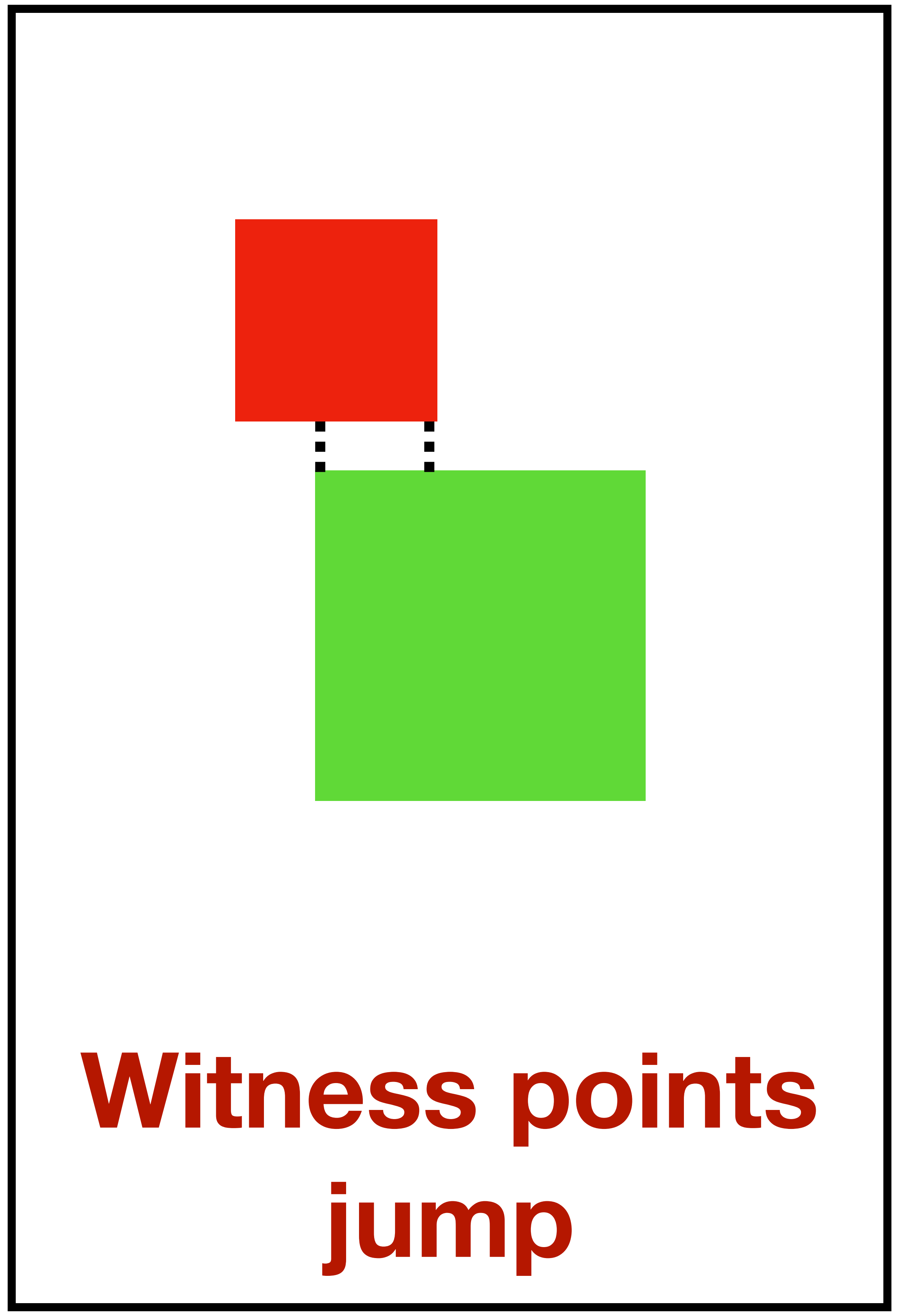


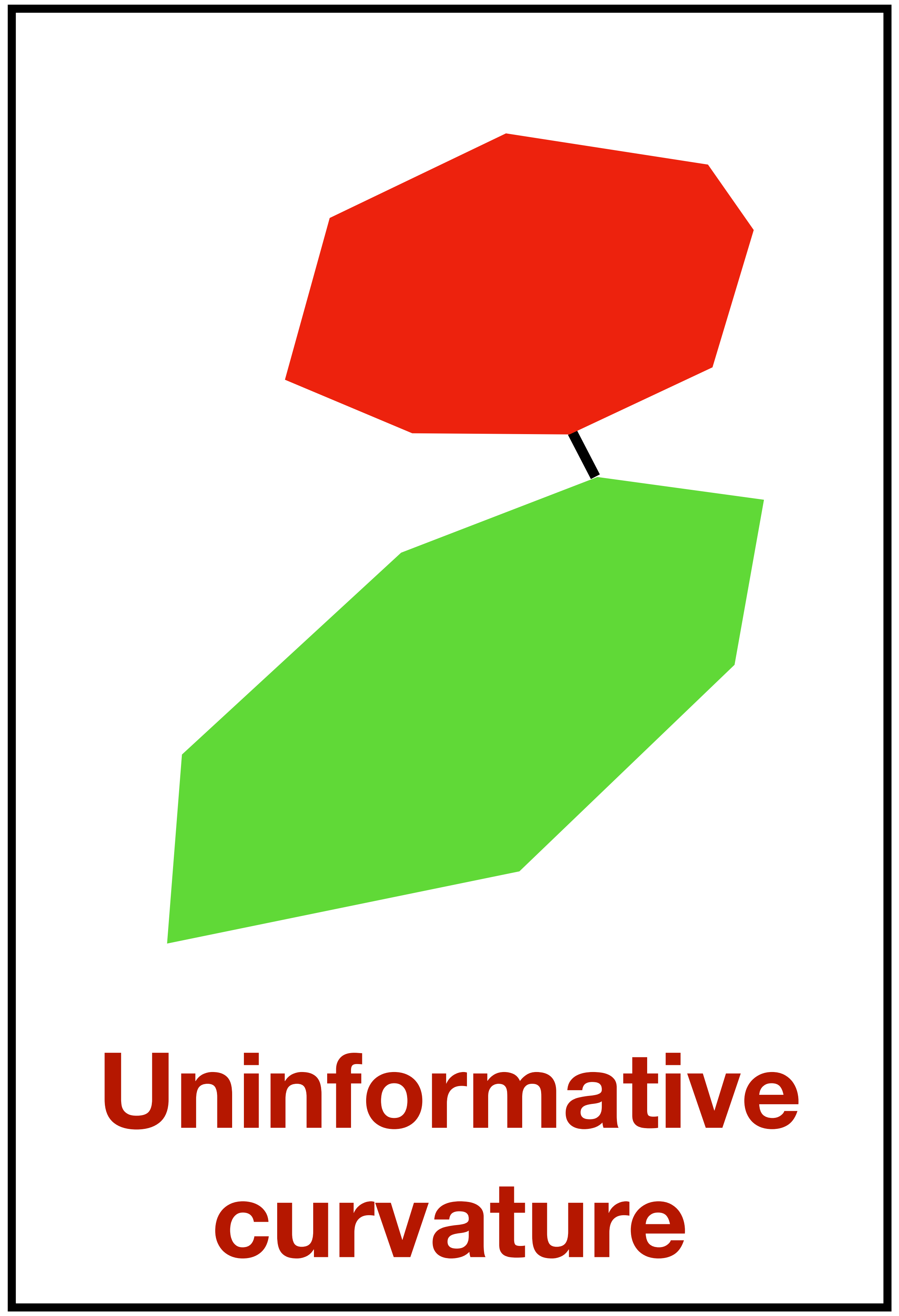
If we move the shapes,
how do the blue points move?

Collision detection is non-smooth

$$x_1^*(T), x_2^*(T) = \text{argmin}\,||x_1 - x_2||_2^2$$
$$\text{s.t } x_1 \in \mathcal{A}_1, x_2 \in \mathcal{A}_2(T)$$

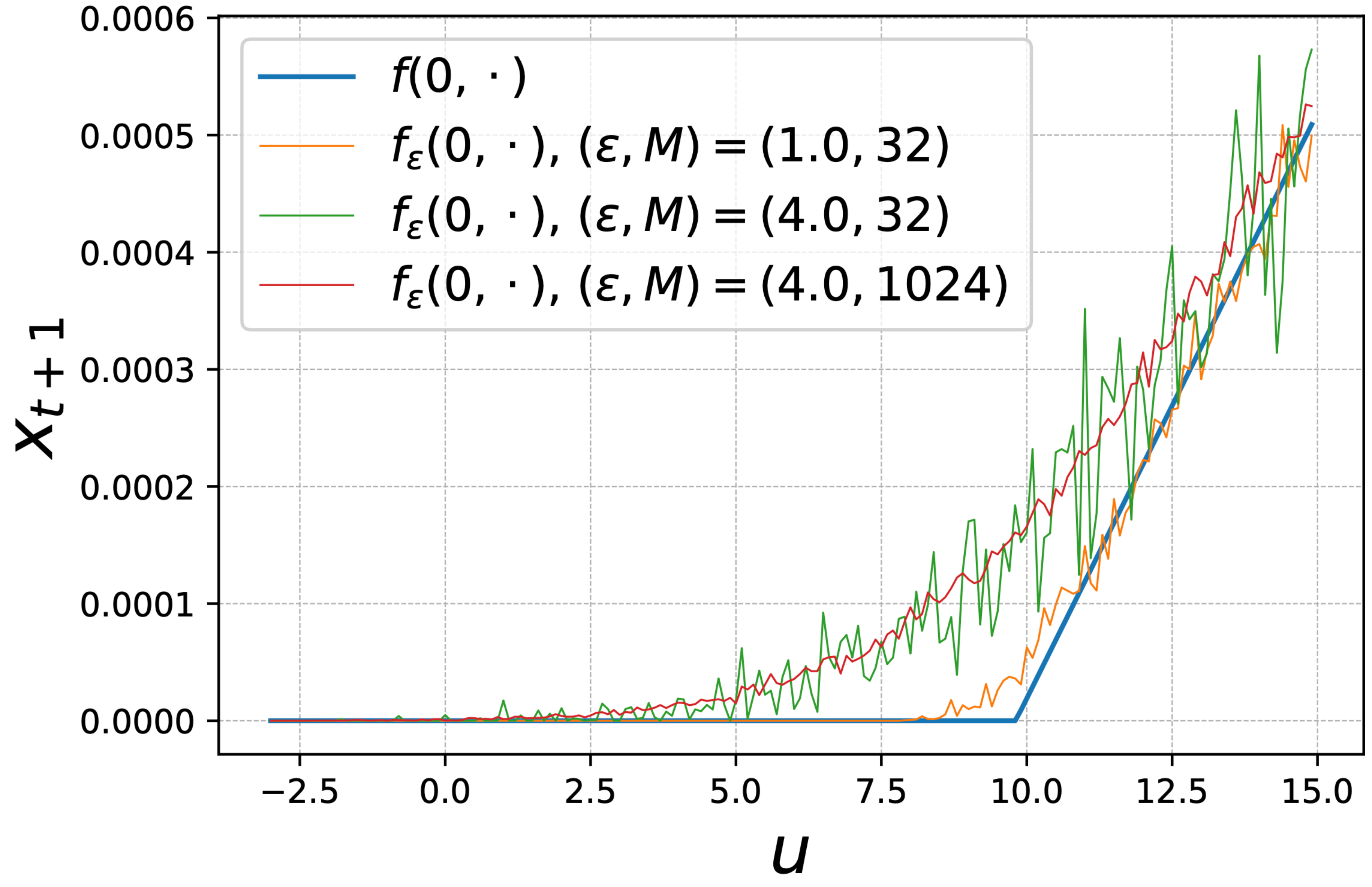






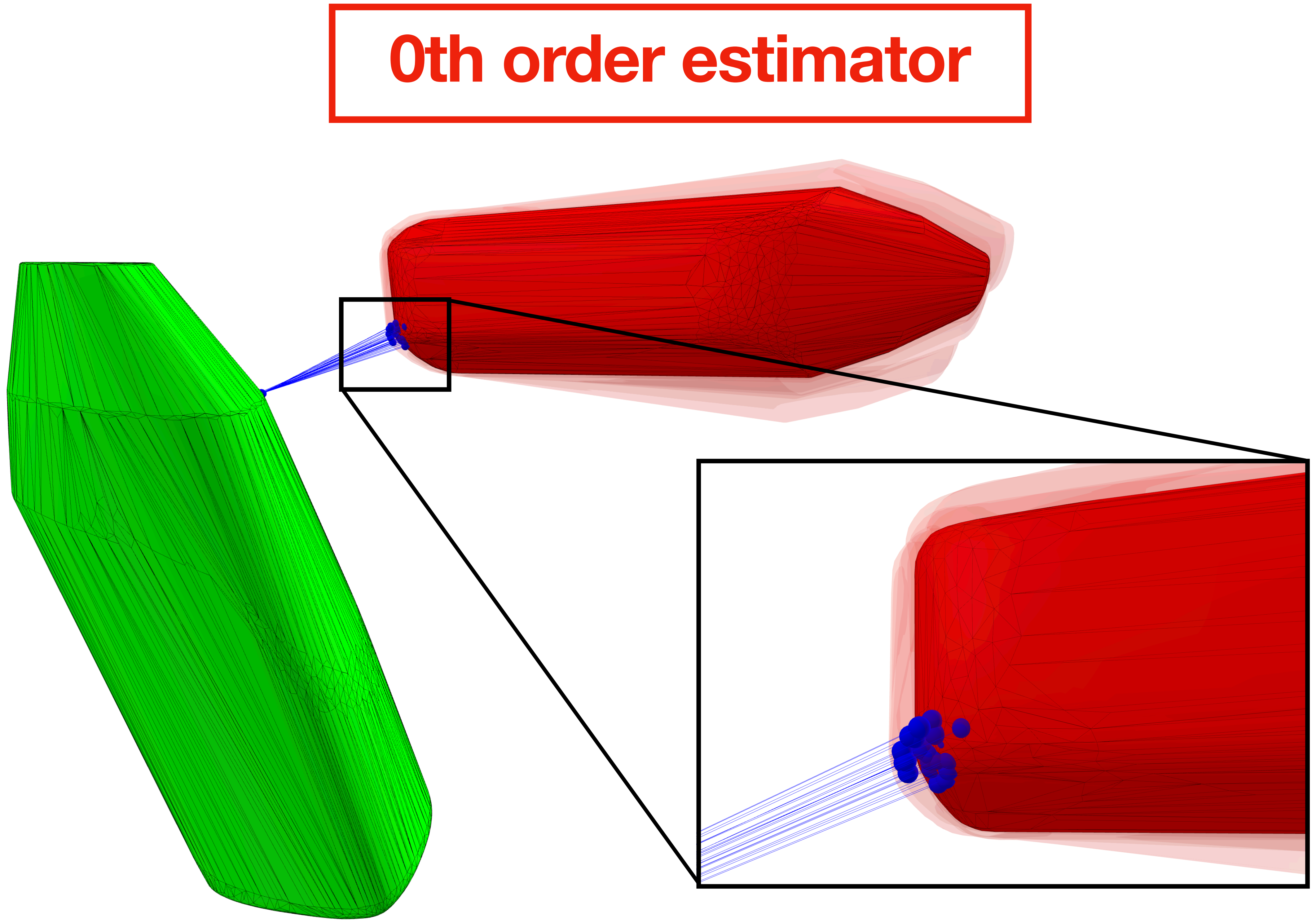
Randomized smoothing

$$g_\epsilon(x) = \mathbb{E}_{Z\sim\mu}\left[g(x+\epsilon Z)\right] \longrightarrow \nabla_x^{(0)} g_\epsilon(x) = \frac{1}{M}\sum_{j=0}^{M} -g(x+\epsilon z^{(j)})\frac{\nabla \log \mu(z^{(j)})}{\epsilon}$$

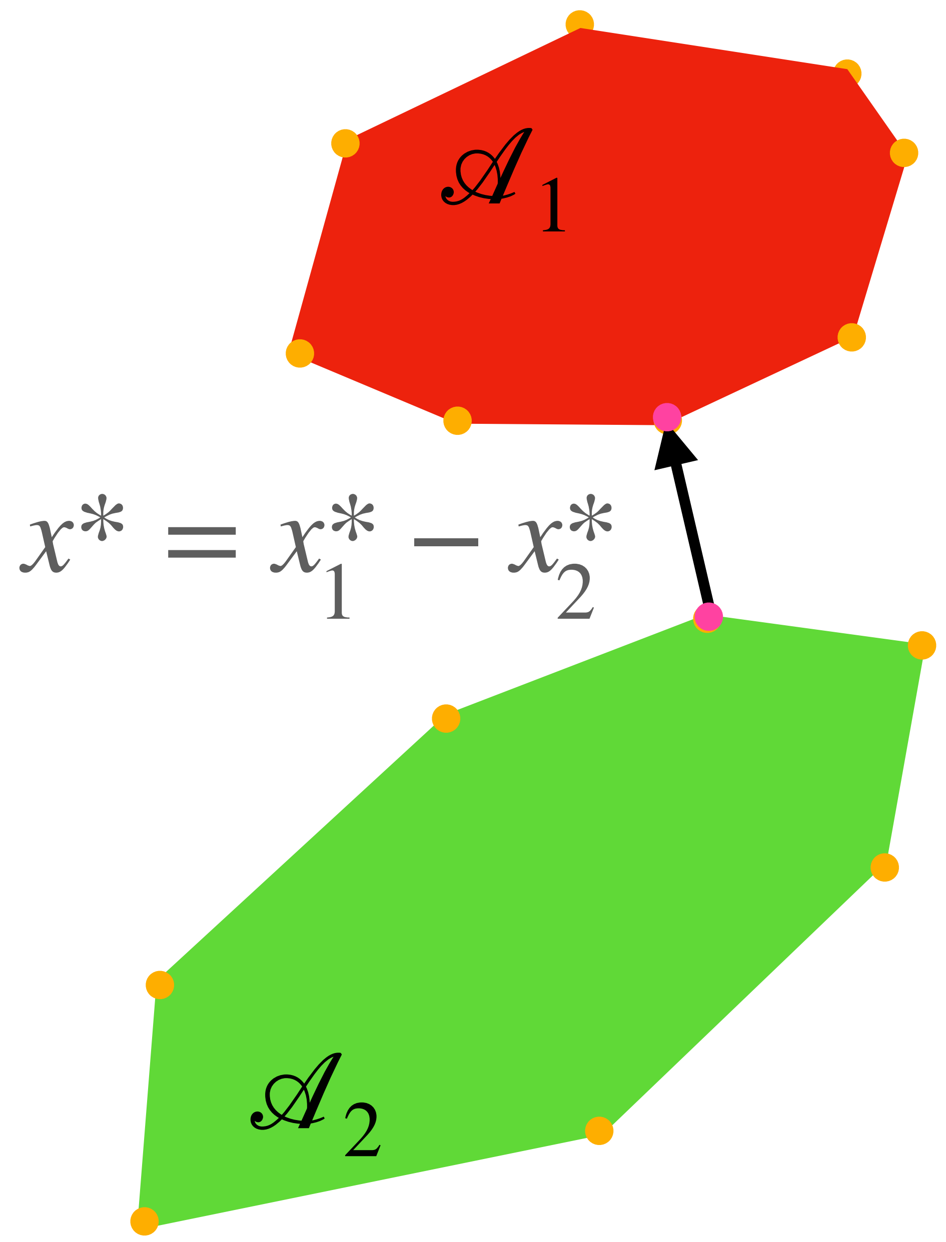


Randomized smoothing - 0th order

$$x^*_{1,\epsilon}(T), x^*_{2,\epsilon}(T) = \mathbb{E}_z \left[\text{argmin} ||x_1 - x_2||^2 \right]$$
$$\text{s.t } x_1 \in \mathcal{A}_1, x_2 \in \mathcal{A}_2(T \oplus \epsilon z)$$



Collision detection optimality conditions & the implicit function theorem



$$f(x^*, T) = 0$$

$$\frac{\partial x^*}{\partial T} = -\left[\frac{\partial f(x^*, T)}{\partial x^*}\right]^{-1} \frac{\partial f(x^*, T)}{\partial T}$$

Need the **Hessian** of support function, usually **null/undefined.**
We use **Randomized Smoothing** again.

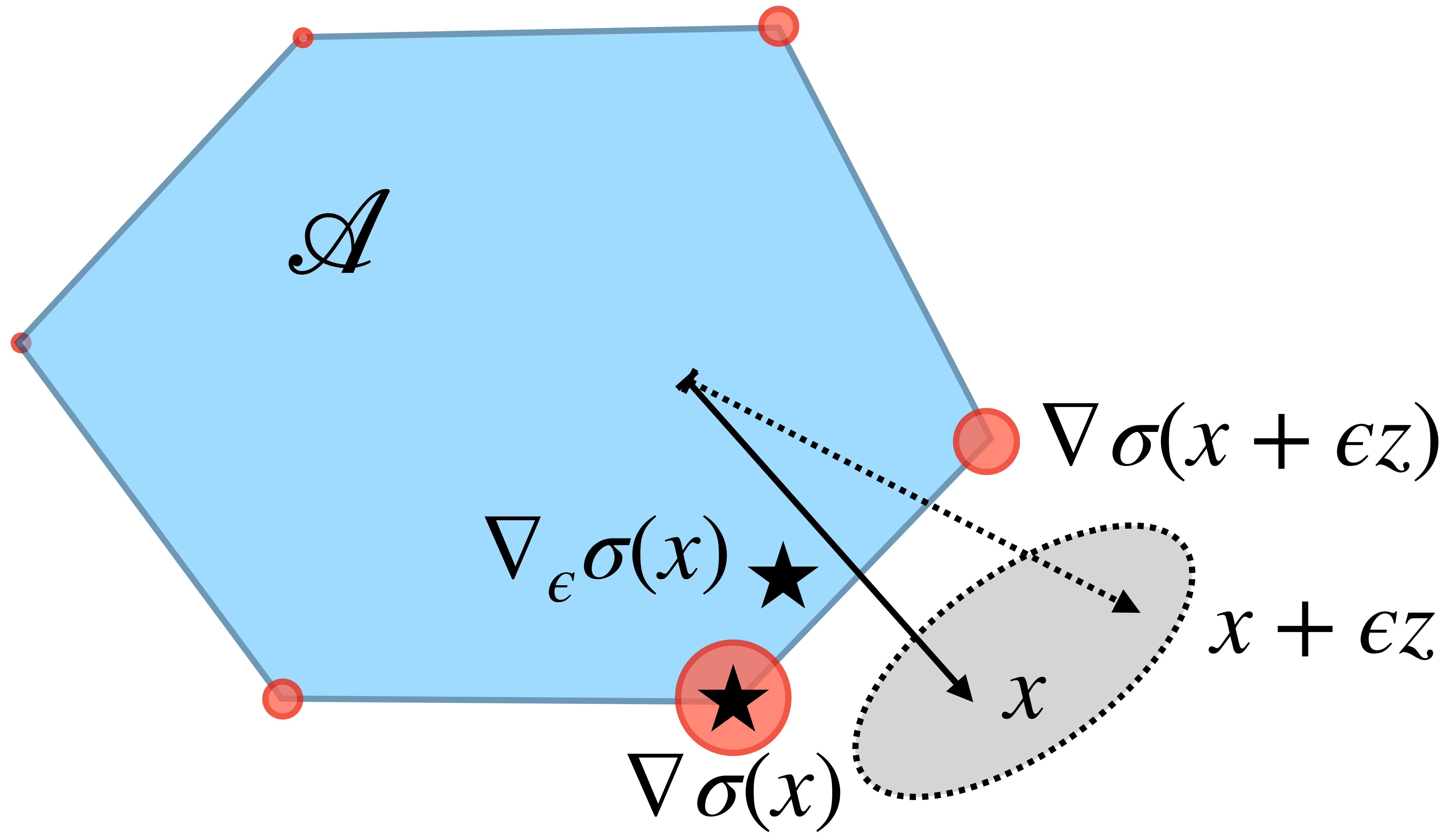
Randomized smoothing - 1st order

Need the **Hessian** of support function, usually **null/undefined.**
We use **Randomized Smoothing** again:

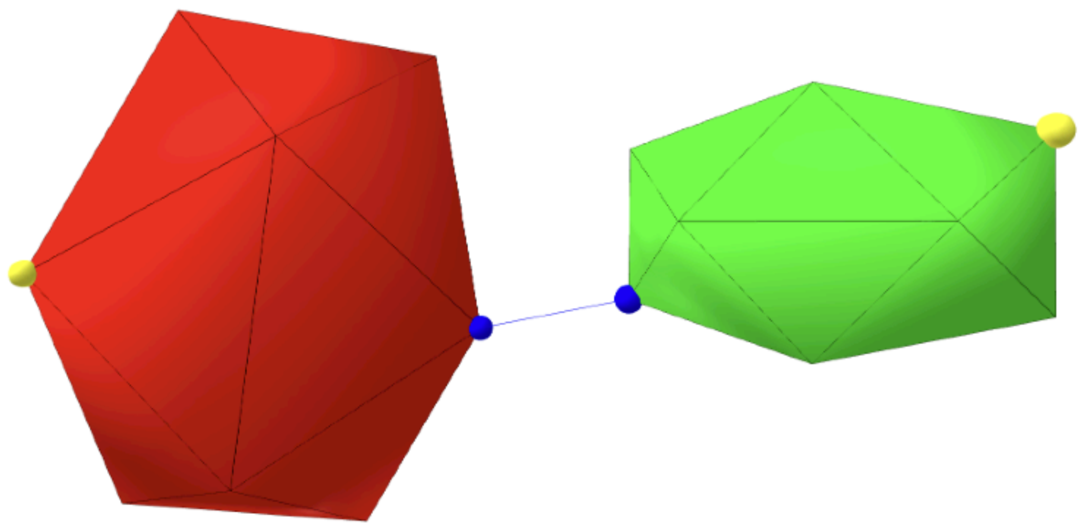
$$\sigma_{\mathscr{A}}(x) = \max_{y \in \mathscr{A}} y^T x$$

$$\nabla \sigma_{\mathscr{A}}(x) = S_{\mathscr{A}}(x) = \text{argmax}_{y \in \mathscr{A}} \, y^T x$$

$$\frac{\partial^2 \sigma_{\mathscr{A},\epsilon}(x)}{\partial x^2} = \frac{1}{M} \sum_{j=0}^{M} - \nabla \sigma_{\mathscr{A}}(x + \epsilon z^{(j)}) \frac{\log \mu(z^{(j)})}{\epsilon}$$

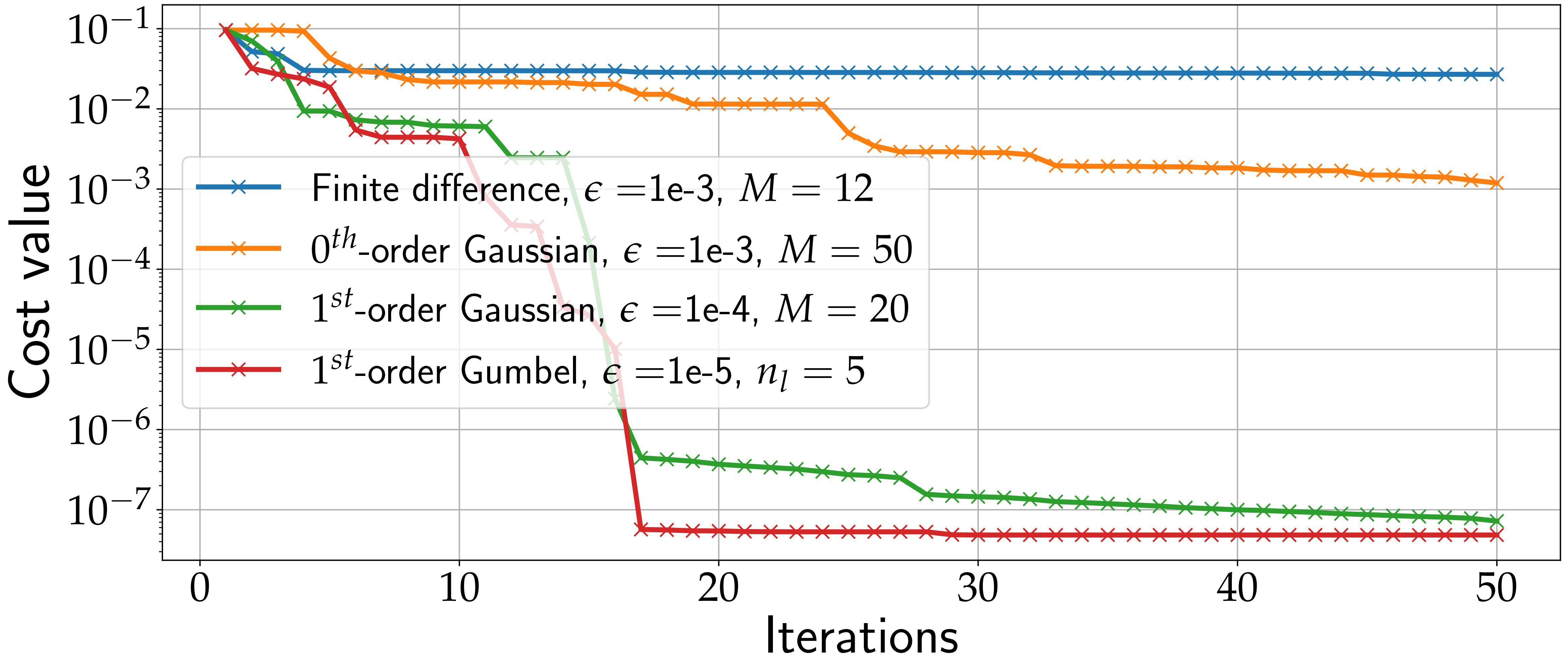


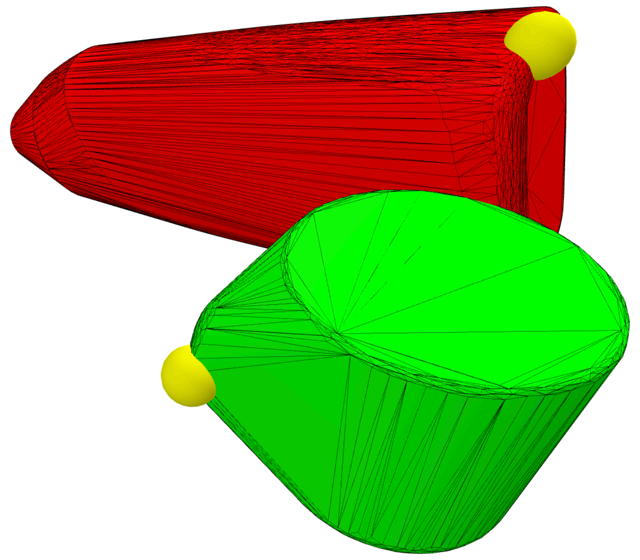
Solving an optimization problem with collision detection derivatives



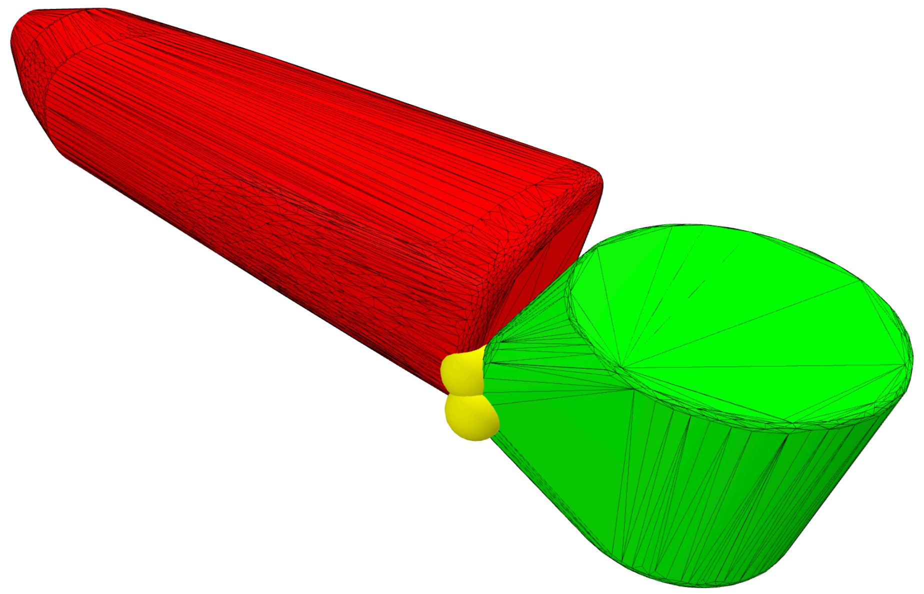
$$\min_T \sum_{i=1,2} ||x_i^*(T) - x_{i,des}^*||^2 + ||x_1^*(T) - x_2^*(T)||^2$$

$$= C(T)$$

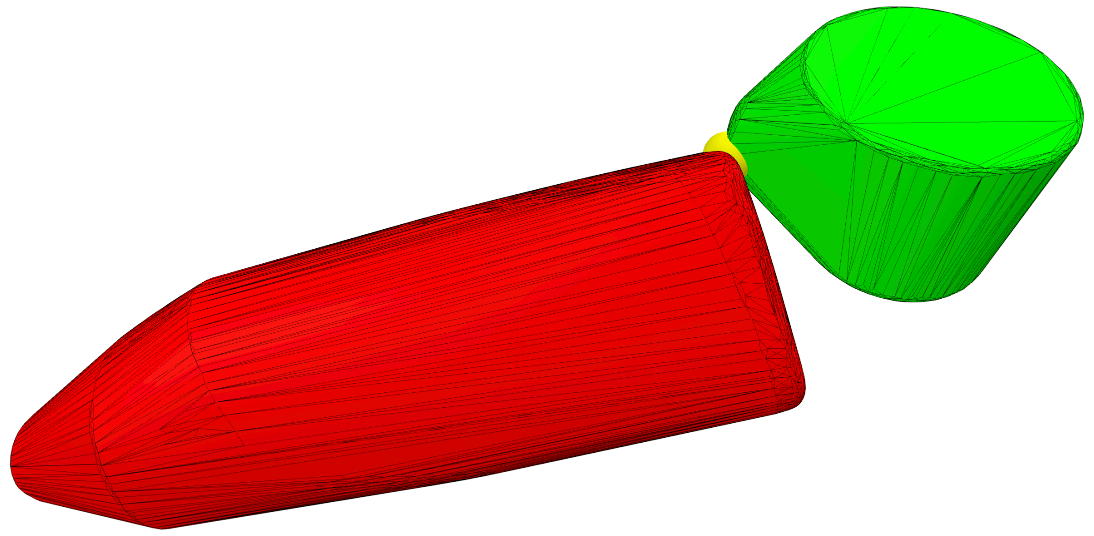




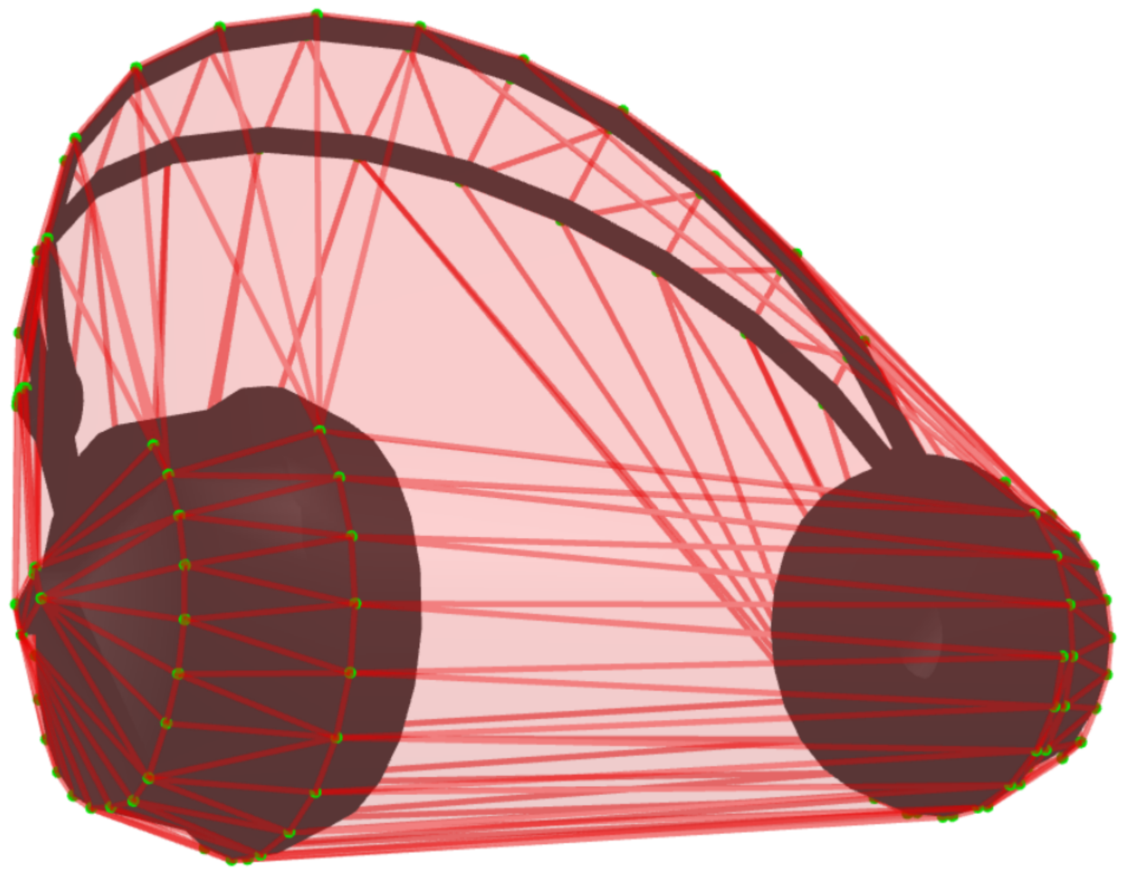
Finite differences

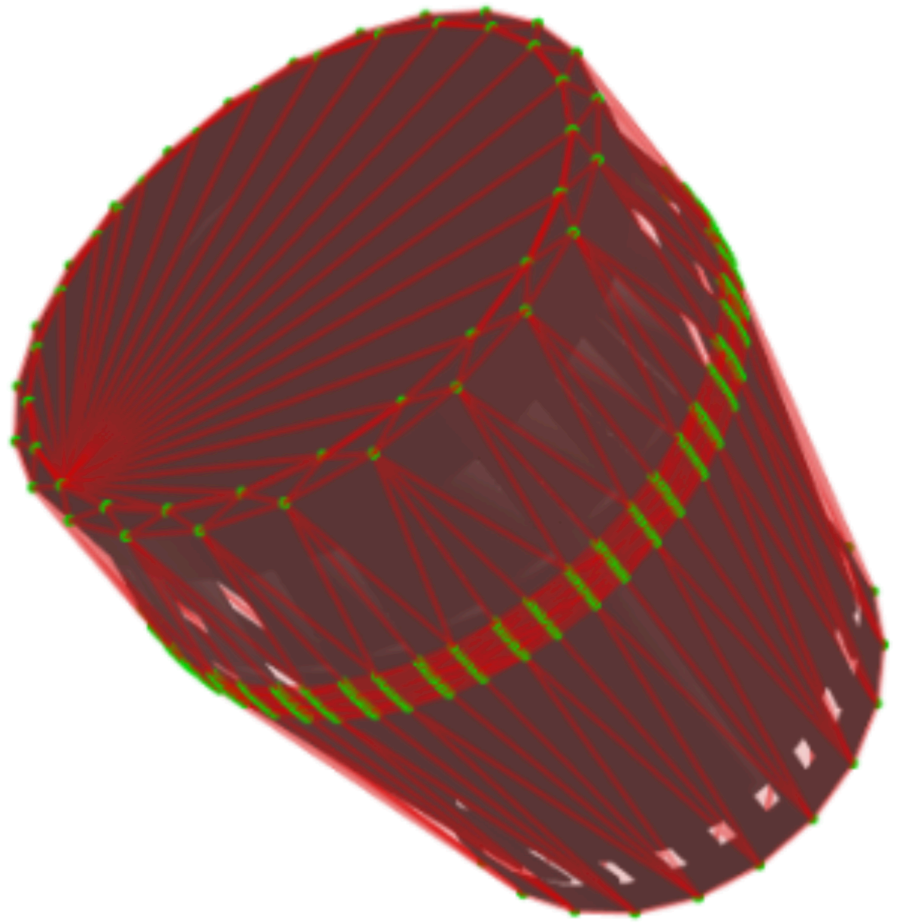


Zero-order



First-order





Conclusion