Collision detection

Marek Trtík

PA199

Outline

- Broad phase
  - Sweep and prune algorithm
- Narrow phase
  - Gilbert-Johnson-Keerthi (GJK) algorithm
- Caching collisions
- Computing collision time

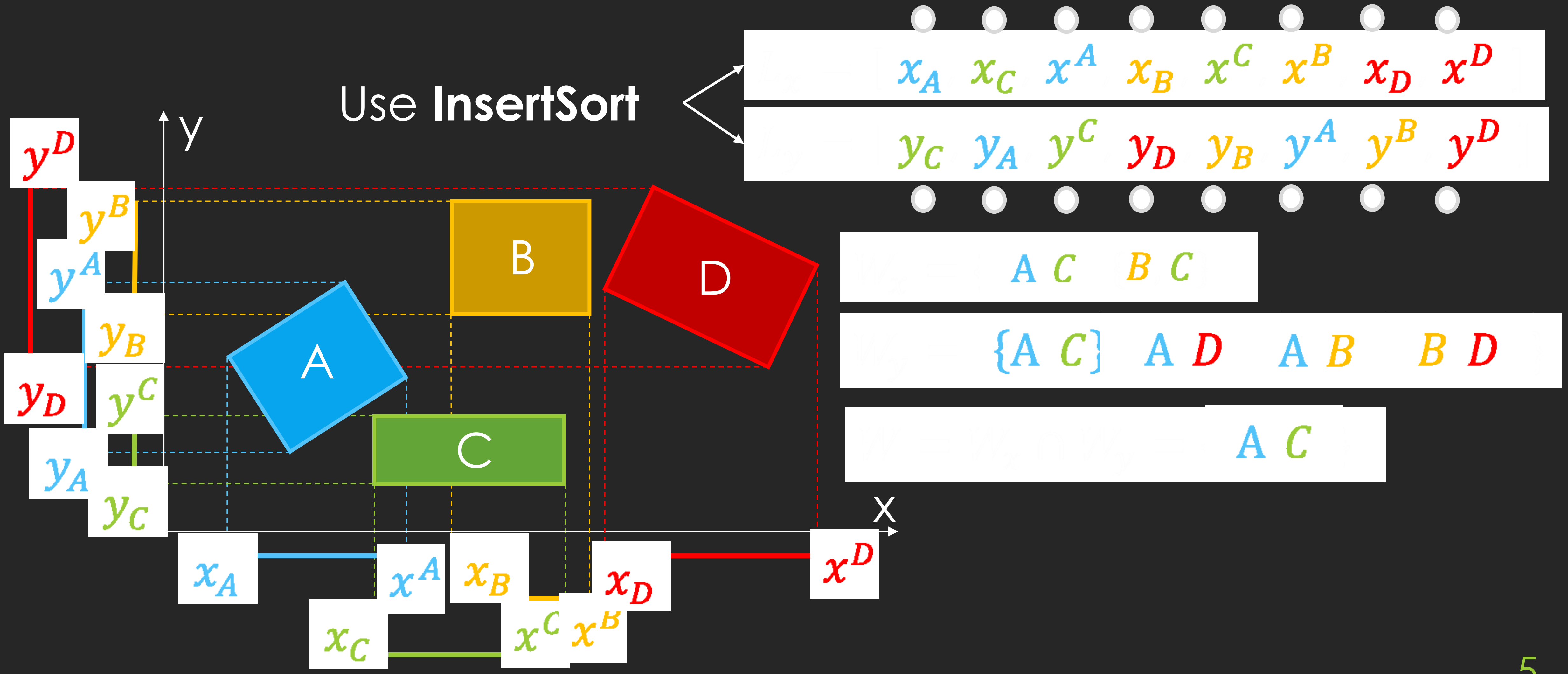
Broad phase

Broad phase

- The goal is to quickly find **pairs** of **potentially colliding** rigid bodies.
  - Used algorithm defines meaning of "potentially colliding". Examples:
    - When AABBs of the bodies are colliding.
    - When both bodies are in the same area of space.
- We can use space partitioning data structures we already know:
  - Octree, k-D tree, BSP
- Rigid bodies change their positions and orientations during simulation.

  => The data structure must be periodically updated.
  - Utilize time coherence of frames (positions of bodies do not change much between adjacent frames) to get an efficient update algorithm.

Sweep and prune algorithm



Sweep and prune algorithm

- The presented version is easy to understand and implement.
- But it wastes time by recomputing W from scratch each time step.
- In practice, we use an improved version:
  - We start with the arrays L_α, $\alpha \in \{x, y, z\}$, and W from the previous frame.
  - We incrementally update each L_α and W for each relocated object A.

foreach axis $\alpha \in \{x, y, z\}$ **do**

  Update α_A in L_α by the new lower bound of A along the axis α

  while $\exists \alpha_Y$ right before α_A in L_α s.t. $\alpha_Y > \alpha_A$ **do**

    Swap α_A with α_Y in L_α.

    if X **is None then** insert (A, Y) to W.

Sweep and prune algorithm

- Update α^A in L_α by the new upper bound of A along the axis α.

  while $\exists \alpha_l^X$ right after α^A in L_α s.t. $\alpha^A > \alpha_l^X$ **do**

  Swap α^A with α_l^X in L_α.

  if X **is None then** Insert (A, X) to W.

  Moving α^A "to the right"

  while $\exists \alpha_l^X$ right after α_A in L_α s.t. $\alpha_A > \alpha_l^X$ **do**

  Swap α_A with α_l^X in L_α.

  if X **is None then** Erase (A, X) from W.

  Moving α_A "to the right"

  while $\exists \alpha_l^X$ right before α^A in L_α s.t. $\alpha_l^X > \alpha^A$ **do**

  Swap α^A with α_l^X in L_α.

  if X **is None then** Erase (A, X) from W.

  Moving α^A "to the left"

Sweep and prune algorithm

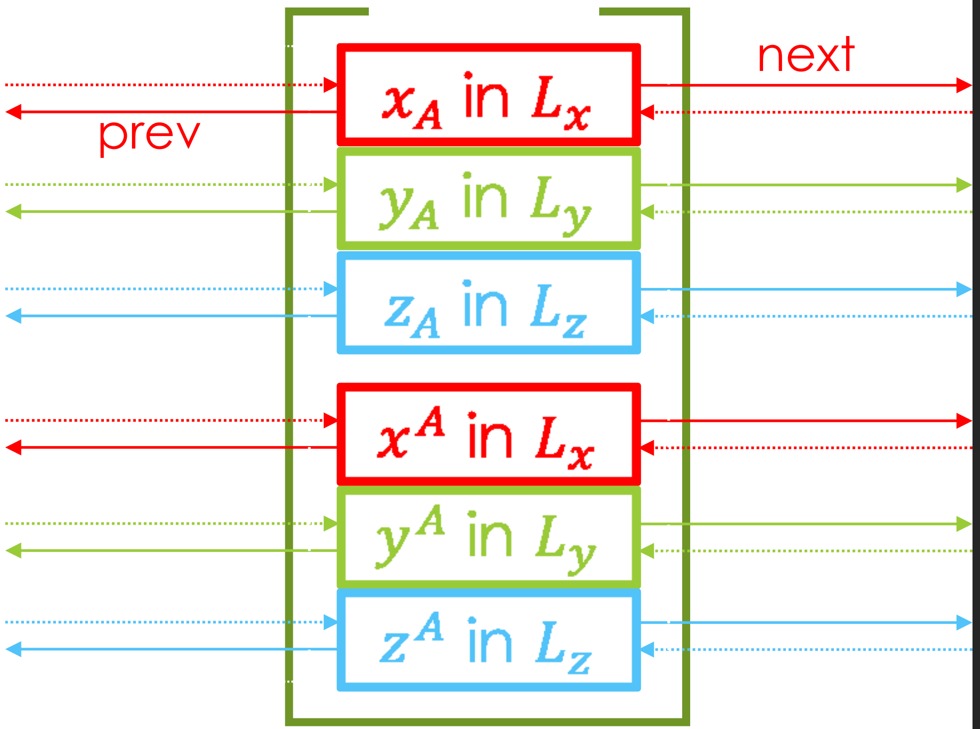
struct

float

"red"

char

using



Sweep and prune algorithm

-

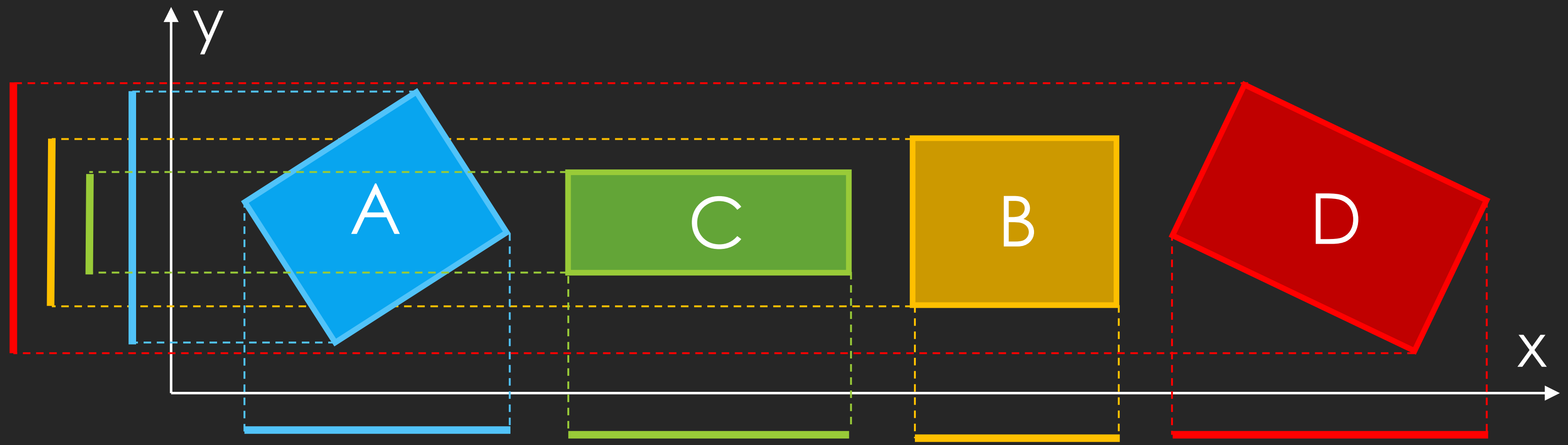
  (α + 3 * (int)p.lohi) sizeof

-
  -

-
  -
  -
  - nullptr

Sweep and prune algorithm

- Performance of the algorithm is sensitive to alignment of objects along coordinate axes:

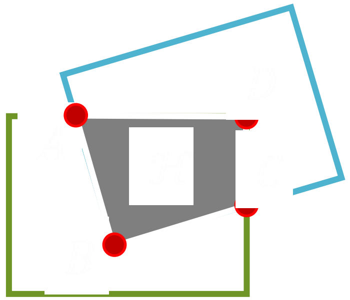


- A relocation of an object leads to a lot of swaps thought the “cluster” in the array.

Narrow phase

Narrow phase

- The goal is for each pair of potentially colliding shapes to:
  - Decide whether the shapes really collide or not.
  - Compute a finite model of the (infinite) set of all collision points.



- Example: Find finite and minimal number of points in $\mathcal{X}$ whose convex hull contains $\mathcal{X}$.

- Requirement: The effect of collision forces computed at points of the model must be equal to collision forces computed at all points in $\mathcal{X}$.

Gilbert-Johnson-Keerthi (GJK) algorithm

- Decides whether two **convex** shapes have **empty intersection** or not.

Convex shapes Concave shapes

  - We can approximate a concave shape by a **set** of convex shapes.
  - For the empty intersection we can obtain a pair of the **closest points**.

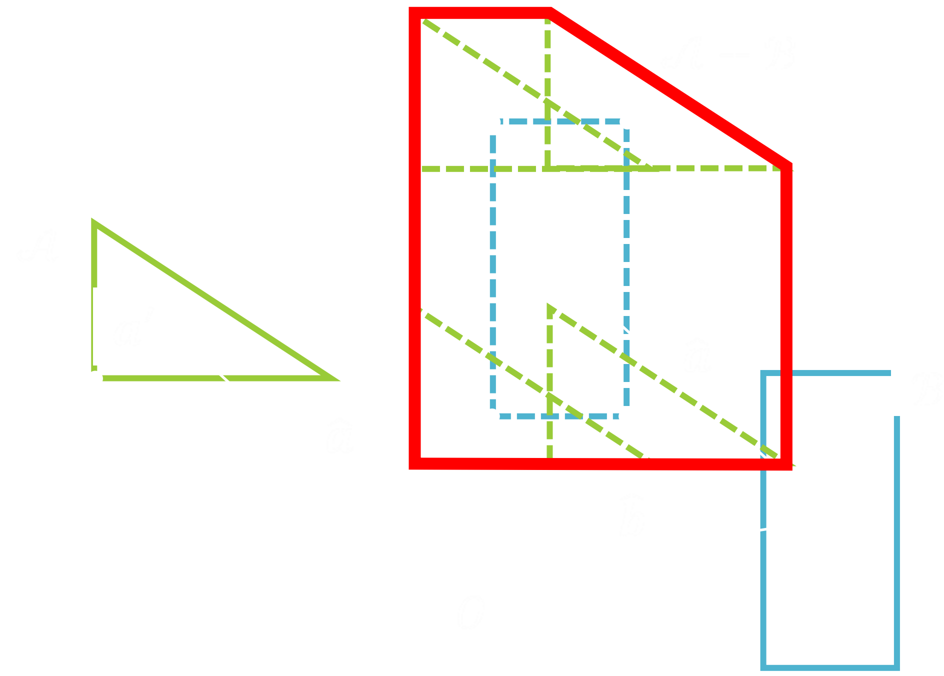
- We must first build a terminology:
  - Minkowski sum and difference
  - Simplex
  - Support function

GJK: Minkowski sum

- Minkowski sum: $\mathcal{A} + \mathcal{B} = \{a + b; a \in \mathcal{A} \wedge b \in \mathcal{B}\}$.
- How to draw Minkowski sum?
  - Choose some points $\hat{a} \in \mathcal{A} \wedge \hat{b} \in \mathcal{B}$.
  - Then, $\forall a \in \mathcal{A}\ \exists a'$ s.t. $a = \hat{a} + a'$.
  - Therefore, for each $a \in \mathcal{A} \wedge b \in \mathcal{B}$

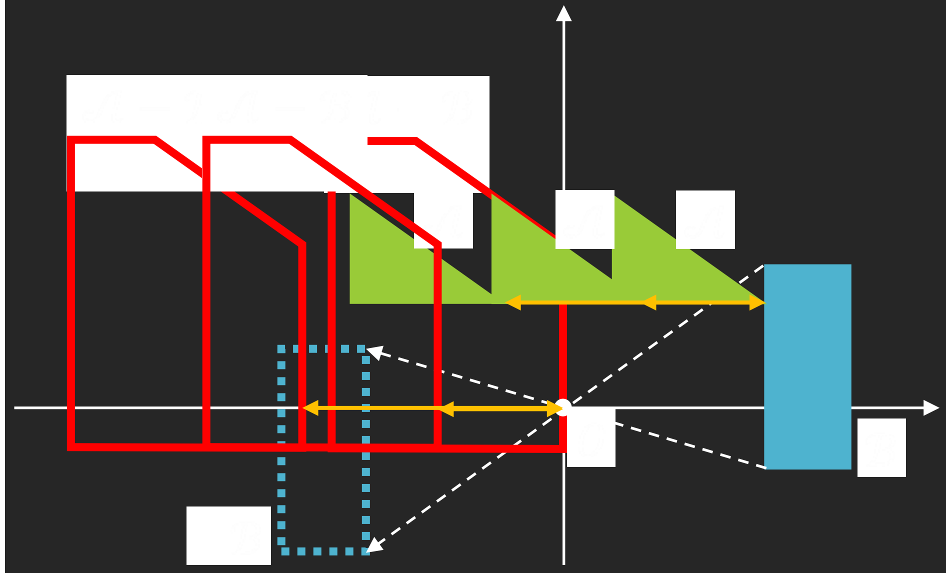
$$a + b = \hat{a} + \hat{b} + a' + b'$$

  - So, we draw $\mathcal{A} + \mathcal{B}$ around $\hat{a} + \hat{b}$.
    - Draw $\mathcal{B}$ around $\hat{a} + \hat{b}$.
    - Draw $\mathcal{A}$ around $\mathcal{B}$'s perimeter.
    - $\mathcal{A} + \mathcal{B}$ is the convex hull.



GJK: Minkowski difference

- **Minkowski difference:** $\mathcal{A} - \mathcal{B} = \mathcal{A} + (-\mathcal{B})$, where $-\mathcal{B} = \{-\boldsymbol{b}; \boldsymbol{b} \in \mathcal{B}\}$
- **Lemma:** The shortest distance between $\mathcal{A}$ and $\mathcal{B}$ is equal to the distance of $\mathcal{A} - \mathcal{B}$ to the origin.
  Proof: It is a length of the shortest $\widehat{\boldsymbol{a}} - \widehat{\boldsymbol{b}}$, s.t. $\widehat{\boldsymbol{a}} \in \mathcal{A} \wedge \widehat{\boldsymbol{b}} \in \mathcal{B}$. But $\widehat{\boldsymbol{a}} - \widehat{\boldsymbol{b}} \in \mathcal{A} - \mathcal{B}$.
- **Consequence:** Shapes $\mathcal{A}$ and $\mathcal{B}$ collide if and only if $\mathcal{A} - \mathcal{B}$ contains the origin.

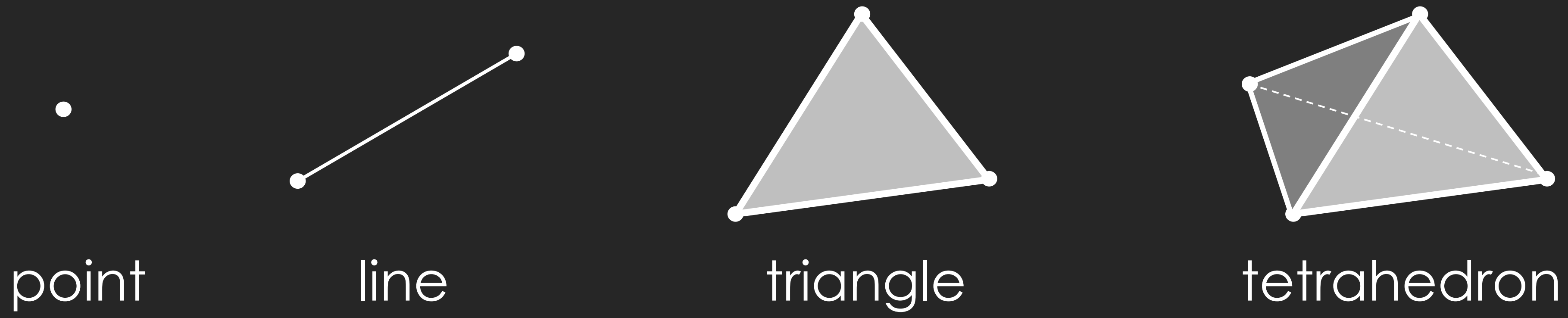


GJK: Minkowski difference

- Lemma: If shapes $\mathcal{A}$ and $\mathcal{B}$ are convex, then $\mathcal{A} - \mathcal{B}$ is also convex.
  Proof: For each $u, v \in \mathcal{A} - \mathcal{B}$ there exist $a_u, a_v \in \mathcal{A}$ and $b_u, b_v \in \mathcal{B}$ s.t.
  $u = a_u - b_u$ and $v = a_v - b_v$. Then, for $t \in (0,1)$, we get
  $$u + t(v - u) = (a_u - b_u) + t\big((a_v - b_v) - (a_u - b_u)\big) =$$
  $$a_u - b_u + ta_v - tb_v - ta_u + tb_u =$$
  $$a_u + t(a_v - a_u) - \big(b_u + t(b_v - b_u)\big).$$
  $\mathcal{A}$ and $\mathcal{B}$ are convex => $a_u + t(a_v - a_u) \in \mathcal{A}$, $b_u + t(b_v - b_u) \in \mathcal{B}$ =>
  $a_u + t(a_v - a_u) - \big(b_u + t(b_v - b_u)\big) \in \mathcal{A} - \mathcal{B}$ => $\mathcal{A} - \mathcal{B}$ is convex.
- Consequence: If the origin lies in the convex hull of points $x_1, \ldots, x_n \in \mathcal{A} - \mathcal{B}$, then convex shapes $\mathcal{A}$ and $\mathcal{B}$ have non-empty intersection.

GJK: Simplex

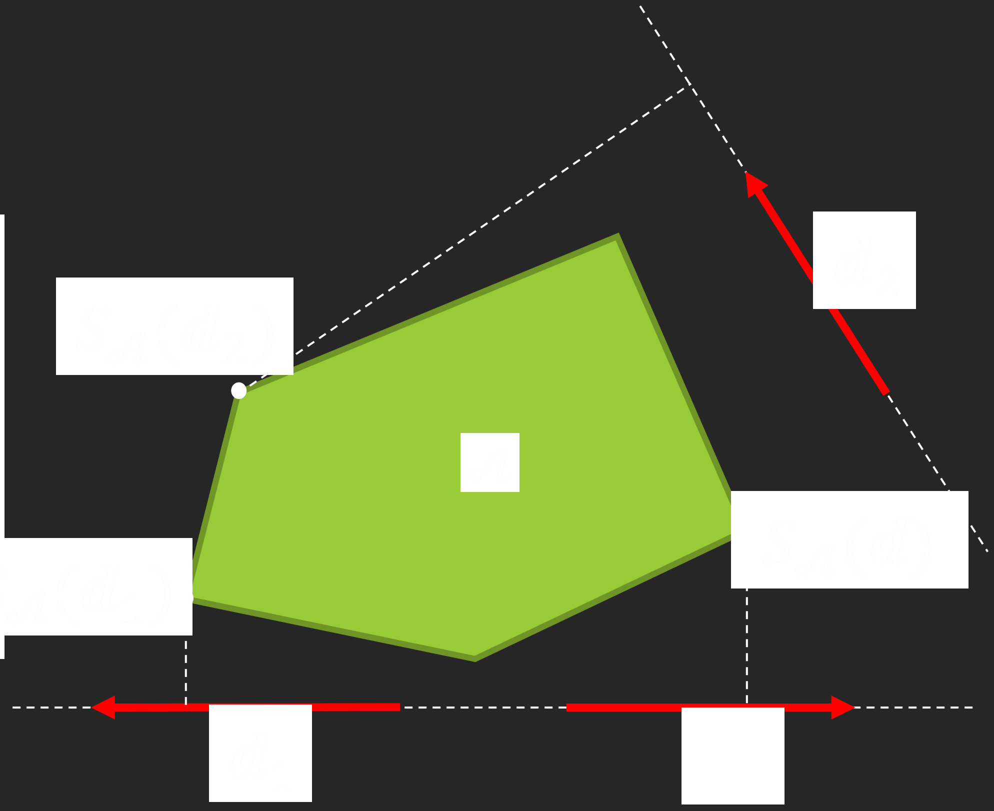
- A **simplex** is a convex hull of an affinely independent points.



- GJK searches for a simplex s.t. origin lies inside or prove that no such simplex exists.
- Note: In 2D case we only need point, line and triangle.

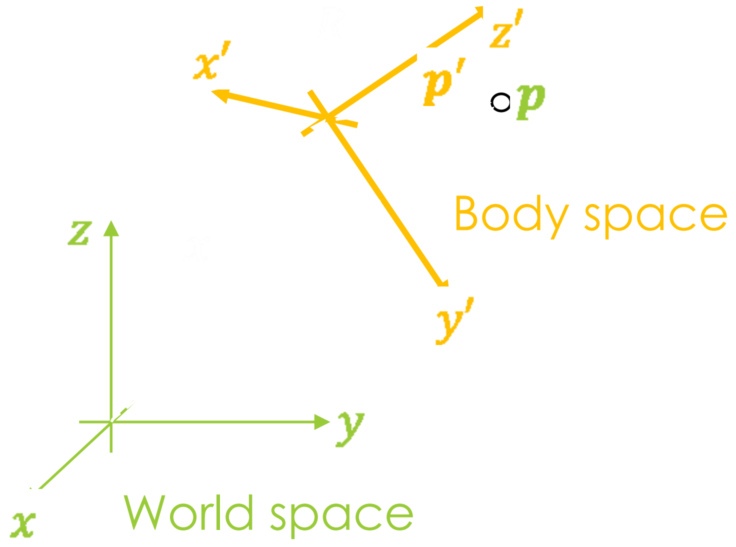
GJK: Support function

- Given a shape $\mathcal{A}$ and a nonzero vector d, a support function $s_{\mathcal{A}}$ returns a point $s_{\mathcal{A}}(d) \in \mathcal{A}$ s.t. $s_{\mathcal{A}}(d) \cdot d = \max\{x \cdot d, x \in \mathcal{A}\}$.



GJK: Support function

- A shape $\mathcal{A}$ can be defined in a local system – the body/model space.



- Therefore, this must be reflected in the computation of $S_{\mathcal{A}}(\mathbf{d})$.

GJK: Support function

- When a convex shape $\mathcal{A}$ is defined in body space (B-space), then we denote $R\mathcal{A} + x$ the corresponding convex shape in the world space.
  - More precisely: $R\mathcal{A} + x = \{Rp' + x : p' \in \mathcal{A}\}$.
- Lemma: $S_{R\mathcal{A}+x}(d) = RS_{\mathcal{A}}(R^T d) + x$ for each world-space vector d.

Proof: First, we show that $\forall p' \in \mathcal{A}$ the following equality (*) holds true:

$$
\begin{aligned}
(Rp' + x) \cdot d &= (Rp') \cdot d + x \cdot d \\
&= d^T(Rp') + x \cdot d \\
&= (d^T R)p' + x \cdot d \\
&= (R^T d)^T p' + x \cdot d \\
&= p' \cdot (R^T d) + x \cdot d.
\end{aligned}
$$

GJK: Support function

- Now, $S_{R\mathcal{A}+x}(d)\cdot d = \max\{p\cdot d; p\in R\mathcal{A}+x\}$

$$\begin{aligned}
S_{R\mathcal{A}+x}(d)\cdot d &= \max\{p\cdot d; p\in R\mathcal{A}+x\}\\
&= \max\{(Rp'+x)\cdot d; p'\in\mathcal{A}\}\\
&= \max\{p'\cdot(R^\top d)+x\cdot d; p'\in\mathcal{A}\} && \text{according to (*)}\\
&= \max\{p'\cdot(R^\top d); p'\in\mathcal{A}\}+x\cdot d\\
&= S_{\mathcal{A}}(R^\top d)\cdot R^\top d + x\cdot d\\
&= (RS_{\mathcal{A}}(R^\top d)+x)\cdot d && \text{according to (*)}
\end{aligned}$$

Therefore, $S_{R\mathcal{A}+x}(d) = RS_{\mathcal{A}}(R^\top d)+x$.

GJK: Support function examples

- $\mathcal{A}$ is a sphere at the origin with the radius r:

$$S_{\mathcal{A}}(\vec{d}) = r\frac{\vec{d}}{|\vec{d}|}$$

- $\mathcal{A}$ is an axis aligned bounding box (AABB) at the origin with sizes $2s_x, 2s_y, 2s_z$ along corresponding coordinate axes:

$$S_{\mathcal{A}}(\vec{d}) = \left(\operatorname{sgn}(d_x)\, s_x, \operatorname{sgn}(d_y)\, s_y, \operatorname{sgn}(d_z)\, s_z\right)^T$$

where $\operatorname{sgn}(a) = \begin{cases} -1 & \text{if } a < 0 \\ 1 & \text{otherwise} \end{cases}$

GJK: Support function examples

- $\mathcal{A}$ is a cylinder at the origin with the central axis aligned with the z-coordinate axis, with the radius r and with the top and bottom base at z-coordinate h and $-h$, respectively:

$$S_{\mathcal{A}}(\vec{d}) = \begin{cases} \left(\frac{r}{\sigma} d_x, \frac{r}{\sigma} d_y, \operatorname{sgn}(d_z)\, h\right)^T & \text{if } \sigma > 0 \\ (0, 0, \operatorname{sgn}(d_z)\, h)^T & \text{otherwise} \end{cases}$$

where $\sigma = \sqrt{d_x^2 + d_y^2}$, and $\operatorname{sgn}(x)$ was defined earlier.

- $\mathcal{A}$ is any convex polytope (e.g., point, line, triangle, convex polygon, tetrahedron, box, …) with vertices $V = \{v_1, \dots, v_n\}$:

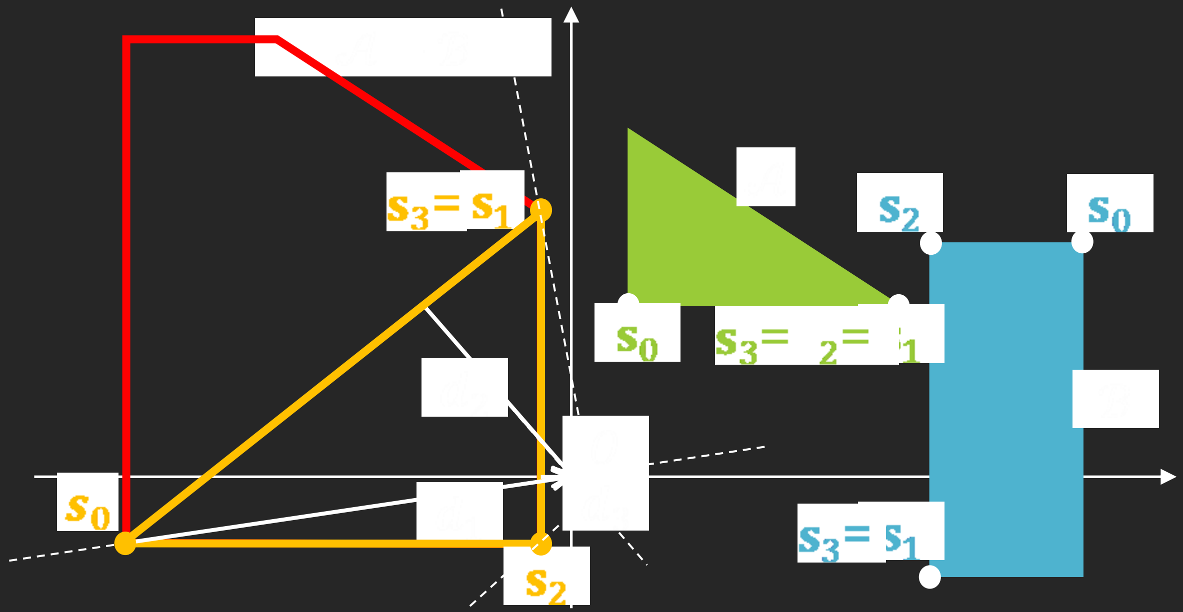
$$S_{\mathcal{A}}(\vec{d}) = v_k \quad \text{s.t.} \quad v_k \cdot \vec{d} = \max\{v_i \cdot \vec{d} : v_i \in V\}$$

GJK: Support function

- Lemma: $S_{A-B}(\vec{d}) = S_A(\vec{d}) - S_B(-\vec{d})$
  Proof: $S_{A-B}(\vec{d}) \cdot \vec{d} = \max\{(a-b) \cdot \vec{d};\ a \in A \wedge b \in B\}$
  $$= \max\{a \cdot \vec{d};\ a \in A\} - \min\{b \cdot \vec{d};\ b \in B\}$$
  $$= S_A(\vec{d}) \cdot \vec{d} + \max\{b \cdot (-\vec{d});\ b \in B\}$$
  $$= S_A(\vec{d}) \cdot \vec{d} + S_B(-\vec{d}) \cdot (-\vec{d})$$
  $$= (S_A(\vec{d}) - S_B(-\vec{d})) \cdot \vec{d}$$

- We therefore do not have to construct $A - B$ and S_{A-B}. We work with the given shapes A and B and their support functions.

GJK: The algorithm – intuition (2D case)



s_0 s_0

$S = \{ s_0, s_1, s_2 \}$

s_0 s_0 s_0

s_1 s_1 s_1

s_1

s_2 s_2 s_2

s_2

s_3 s_3 s_3

s_3 $<$ NO INTERSECTION!

GJK: The algorithm – intuition (2D case)

- Since $\mathcal{A}$ and $\mathcal{B}$ have empty intersection, we can compute a pair of closest points:
  - First, we find the closest point X of the simplex $S = \{\mathbf{s}_1, \mathbf{s}_2\}$ to the origin, that is $X = \mathbf{s}_1 + t(\mathbf{s}_2 - \mathbf{s}_1)$ for some $t \in (0,1)$ s.t. $\|X - O\|^2 = \|\mathbf{s}_1 + t(\mathbf{s}_2 - \mathbf{s}_1)\|^2$ is minimal. So, solve the equation:

$$\frac{d}{dt}\|\mathbf{s}_1 + t(\mathbf{s}_2 - \mathbf{s}_1)\|^2 = 0$$

$$\frac{d}{dt}\left(\mathbf{s}_1 \cdot \mathbf{s}_1 + 2t\,\mathbf{s}_1 \cdot (\mathbf{s}_2 - \mathbf{s}_1) + t^2(\mathbf{s}_2 - \mathbf{s}_1)^2\right) = 0$$

$$2\mathbf{s}_1 \cdot (\mathbf{s}_2 - \mathbf{s}_1) + 2t(\mathbf{s}_2 - \mathbf{s}_1)^2 = 0$$

$$t = -\frac{\mathbf{s}_1 \cdot (\mathbf{s}_2 - \mathbf{s}_1)}{(\mathbf{s}_2 - \mathbf{s}_1)^2}$$

Also clip to (0,1)

GJK: The algorithm – intuition (2D case)

- Then, find the corresponding points in the shapes $\mathcal{A}$ and $\mathcal{B}$.

$$s_1 + r(s_2 - s_1) = (s_1 - s_1) + r((s_2 - s_2) - (s_1 - s_1))$$
$$= \underbrace{(s_1 + r(s_2 - s_1))}_{\in \mathcal{A}} - \underbrace{(s_1 + r(s_2 - s_1))}$$

GJK: The algorithm

// Usually, $\boldsymbol{p}$ comes from the previous frame.

// We start with the empty simplex.

// NOTE: Our direction vector $\boldsymbol{d}$ to the origin is just $-\boldsymbol{p}$.

while **do** // Proving termination condition: see [4].

// $\boldsymbol{p}$ is still far from the closest point of $\mathcal{A} - \mathcal{B}$ to the origin.

Point, line, triangle, or tetrahedron.

Can be computed quickly for shapes

// Reduce the simplex.

return // $|\boldsymbol{p}|$ is the closest distance.

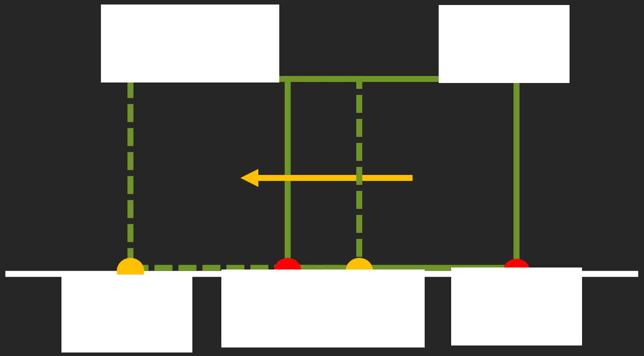
Caching collisions

Caching collisions

- Efficiency of the PGS algorithm for a constraint system depends on the initial value λ^0.
- It's likely that λ computed for a collision constraint at current frame would be "almost valid" for the next frame (if the collision persists).
- Therefore, caching λ values for collision (and other types of) constraints amongst frames can bring considerable speed boost.
- How to match collisions computed in different frames?

Caching collisions

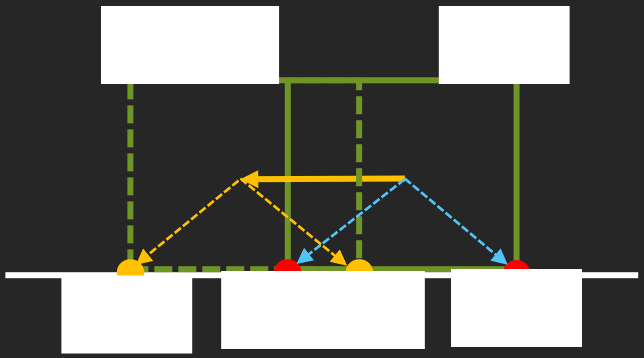
- There are several possibilities:
  - **Distance** between collision points in **world space**:



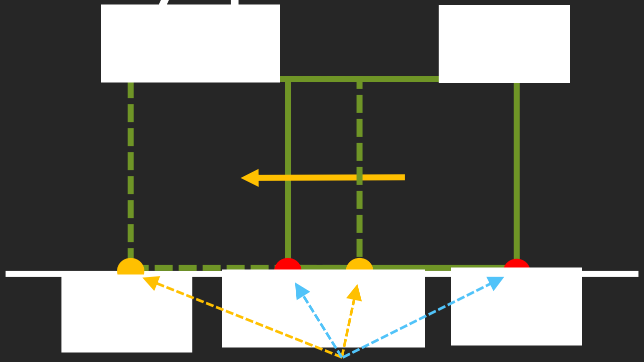
Correct mapping:
$a \to c$, $b \to d$
World distance mapping:
$b \to c$ (wrong), $a \to ?$, $? \to d$

Imprecise.

  - **Distance** between collision points in **body space**:



Precise in space of the **box**



Imprecise in space of **ground**

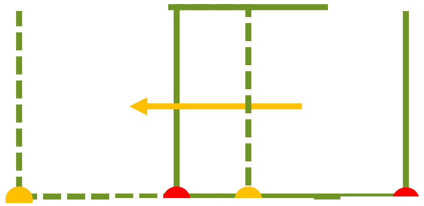
Caching collisions

- Identify collisions by geometrical properties of collision shapes

```
enum GTYPE { VERTEX, EDGE, FACE };
struct CollisionID {
    int body_index_1;           // The index of R_i: i
    GTYPE feature_type_1;       // The type of colliding geometry in R_i
    int feature_index_1;        // Index of the colliding geometry in R_i
    int body_index_2;           // The index of R_j: j
    GTYPE feature_type_2;       // The type of colliding geometry in R_i
    int feature_index_2;        // Index of the colliding geometry in R_i
};
```

- Define data comparison and hashing of CollisionID instances

Caching collisions



We get **precise** mapping:
$id(a) = id(c) \rightarrow a > c$
$id(b) = id(d) \rightarrow b > d$

Recommended approach

- The cache should be a map from CollisionID instance to values α:

```
using collision_cache = std::unordered_map<CollisionID, float>;
```

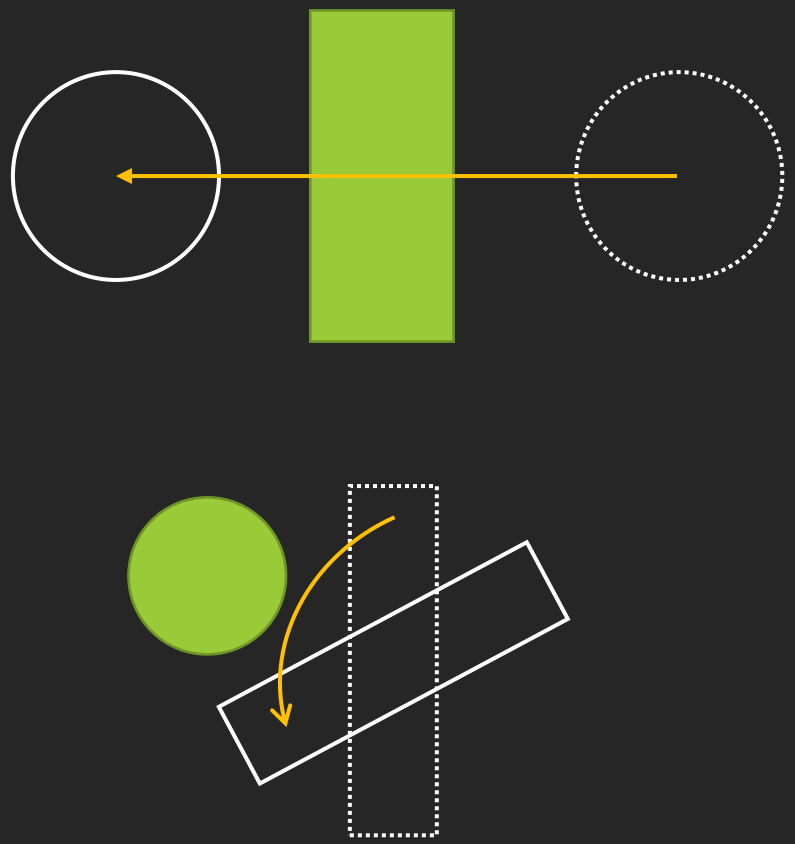
- And how to use the cache?

Caching collisions

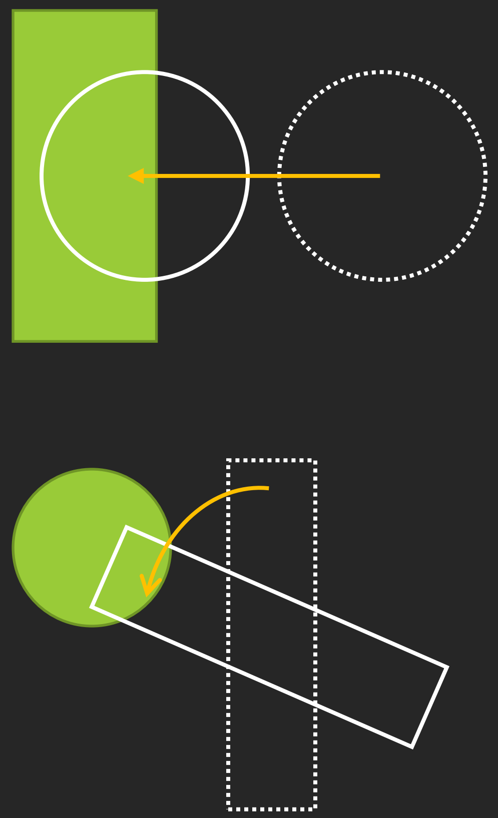
- Before solving the constraint system initialize λ^0 s.t.
  - For each computed collision c and the corresponding element λ_i^0:
    - Build the CollisionID instance id from c.
    - If id is present in the cache, then set λ_i^0 to the value λ in the cache.
    - Otherwise, set λ_i^0 to 0.
- Once new solution λ is computed updated the cache as follows:
  - Clear the cache.
  - For each collision c and the corresponding computed value λ_i:
    - Build the CollisionID instance id from c.
    - Insert the mapping $id \to \lambda_i$ to the cache.

Computing collision time

Tunnelling and penetration



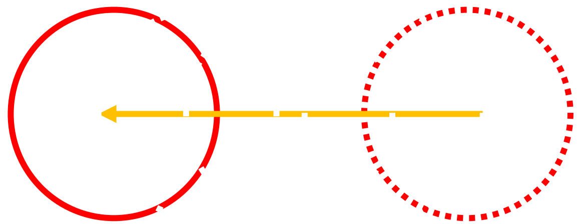
Tunnelling

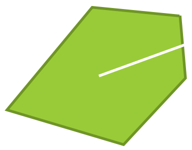


Penetration

Dealing with tunnelling and penetration

- The simplest approach is to subdivide the game time step Δt into several small "internal" time steps.
- For broad phase:
  - Approximate collision shapes of bodies by "moving spheres":

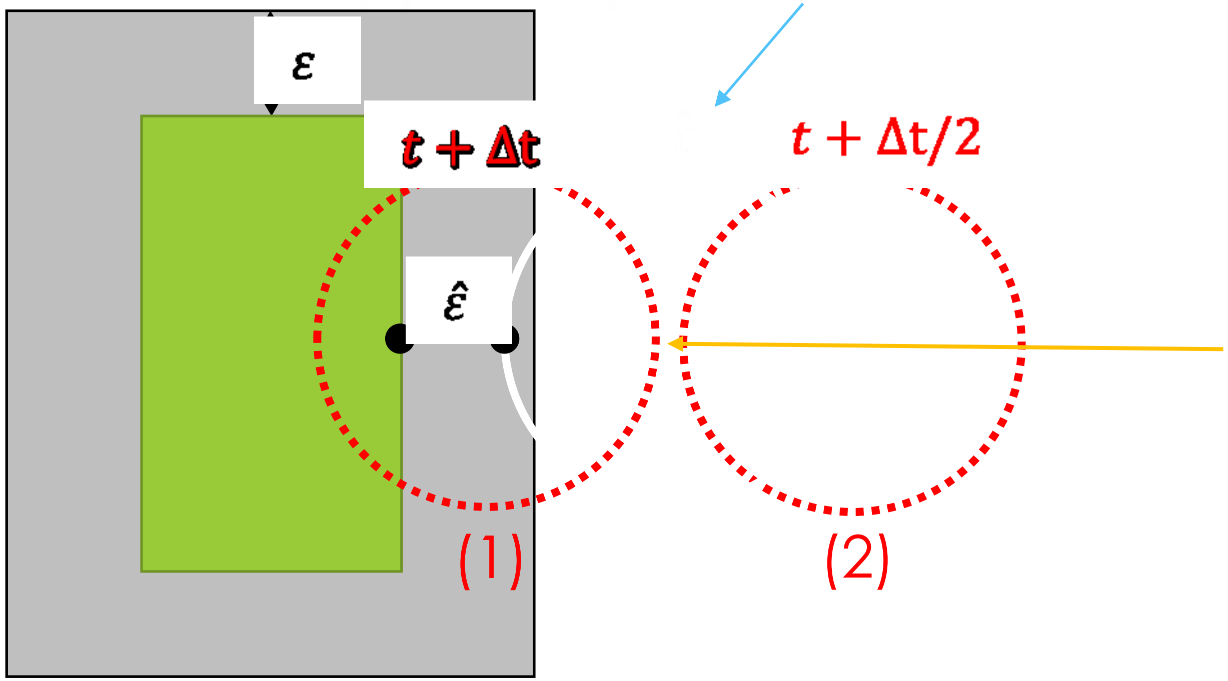




- Use the adaptive time step:
  - For each pair of potentially colliding shapes compute the nearest collision time;
  - Move the bodies only to the minimum of all nearest collision times.

Computing collision time

- Binary search for $\hat{t} \in (t, t + \Delta t)$



- There are 4D collision algorithms – they consider translations and rotations of tested objects.

References

[1] *Erin Catto;* Iterative Dynamics with Temporal Coherence; Crystal Dynamics, Menlo Park, California, 2005
[2] E. G. Gilbert, D. W. Johnson and *S. S.* Keerthi; A fast procedure for computing the distance between complex objects in three-dimensional space; Journal on Robotics and Automation, vol. 4, no. 2, pp. 193-203, April 1988
[3] *G. Bergen*; A Fast and Robust GJK Implementation for Collision Detection of Convex Objects; Eindhoven University of Technology. 1999
[4] G.v.d. Bergen; Collision detection in interactive 3D environments; ISBN: 1-55860-801-X, Elsevier, 2004.