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.

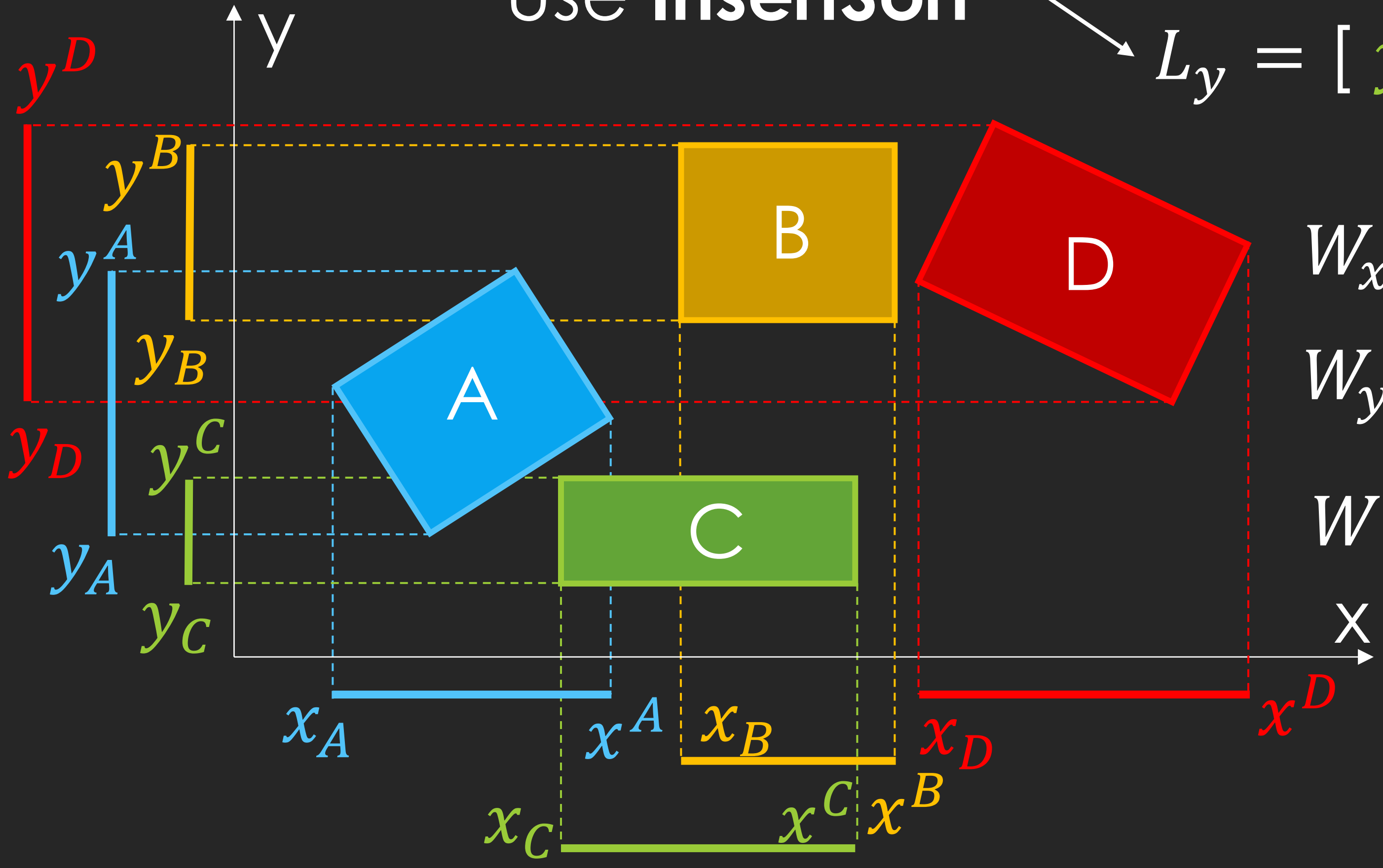
  => The data structure must be periodically updated.
  - Utilize time coherence of frames (positions of bodies do not change much between adjacent frames) to get an efficient update algorithm.

Sweep and prune algorithm

Use **InsertSort**

$$L_x = [\, x_A, x_C, x^A, x_B, x^C, x^B, x_D, x^D \,]$$

$$L_y = [\, y_C, y_A, y^C, y_D, y_B, y^A, y^B, y^D \,]$$



$$W_x = \{\{A, C\}, \{B, C\}\}$$

$$W_y = \{\{A, C\}, \{A, D\}, \{A, B\}, \{B, D\}\}$$

$$W = W_x \cap W_y = \{\{A, C\}\}$$

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 $\exists \alpha_X^Y$ right before α_A in L_α s.t. $\alpha_X^Y > \alpha_A$ do:
        Swap α_A with α_X^Y in L_α.
        if X is None then **Insert** $\{A, Y\}$ to W.

Moving α_A "to the left"

Sweep and prune algorithm

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

while $\exists \alpha_X^Y$ right after α^A in L_α s.t. $\alpha^A > \alpha_X^Y$ do:

    Swap α^A with α_X^Y in L_α.

    if Y is None then **Insert** $\{A, X\}$ to W.

Moving α^A "to the right"

while $\exists \alpha_X^Y$ right after α_A in L_α s.t. $\alpha_A > \alpha_X^Y$ do:

    Swap α_A with α_X^Y in L_α.

    if X is None then **Erase** $\{A, Y\}$ from W.

Moving α_A "to the right"

while $\exists \alpha_X^Y$ right before α^A in L_α s.t. $\alpha_X^Y > \alpha^A$ do:

    Swap α^A with α_X^Y in L_α.

    if Y is None then **Erase** $\{A, X\}$ from W.

Moving α^A "to the left"

Sweep and prune algorithm

- Possible memory representation of the lists L_α, $\alpha \in \{x, y, z\}$:

```
struct Link {
    Link  *next, *prev;
    float coord;
    char lohi : 1;
};

using AABB = Link[2][3];
```

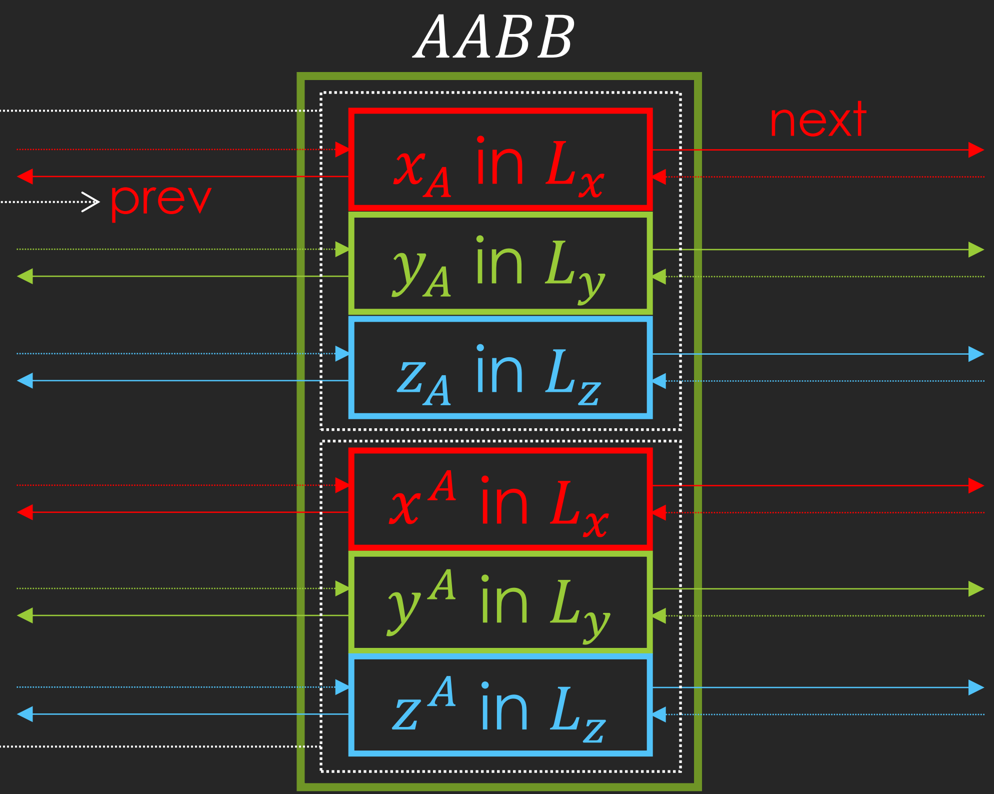
If lohi == 0, then “coord” is α_A and α^A otherwise.

Represents either α_A or α^A.

Points to **some** “red” link in the previous AABB.

Link[0][*]

Link[1][*]



Sweep and prune algorithm

- If "p" is a pointer to a "Link" of the list $L_\alpha, \alpha \in \{0,1,2\}$ (i.e., $\{x, y, z\}$), then we can convert it to a pointer to AABB using the pointer arithmetic:

  (AABB*)(p – (α + 3 * (int)p.lohi) * sizeof(Link))

- Represent the set W as a dictionary of pairs of object IDs.
  - Sort the pair s.t. the lower ID comes first and the other the second.

- Initialize the data structure to contain a single auxiliary AABB s.t:
  - Values in the links are: $x_A = y_A = z_A = -\infty$ and $x^A = y^A = z^A = +\infty$.
  - All 2*3 links are properly interconnected in the lists L_x, L_y, L_z.
  - This auxiliary AABB avoids the "nullptr" check in the algorithm (loops).

Sweep and prune algorithm

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

"Cluster" of bounds y_* and y^* in L_y due to alignment of objects along the x axis.

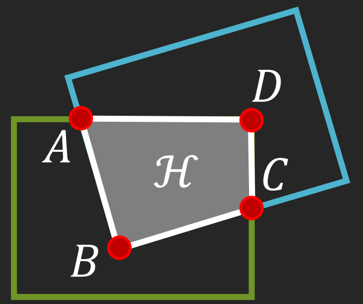
- A relocation of an object leads to a lot of swaps thought the "cluster" in the array.

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**.

Stack of two boxes
(top view)



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

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

Gilbert-Johnson-Keerthi (GJK) algorithm

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

Convex shapes

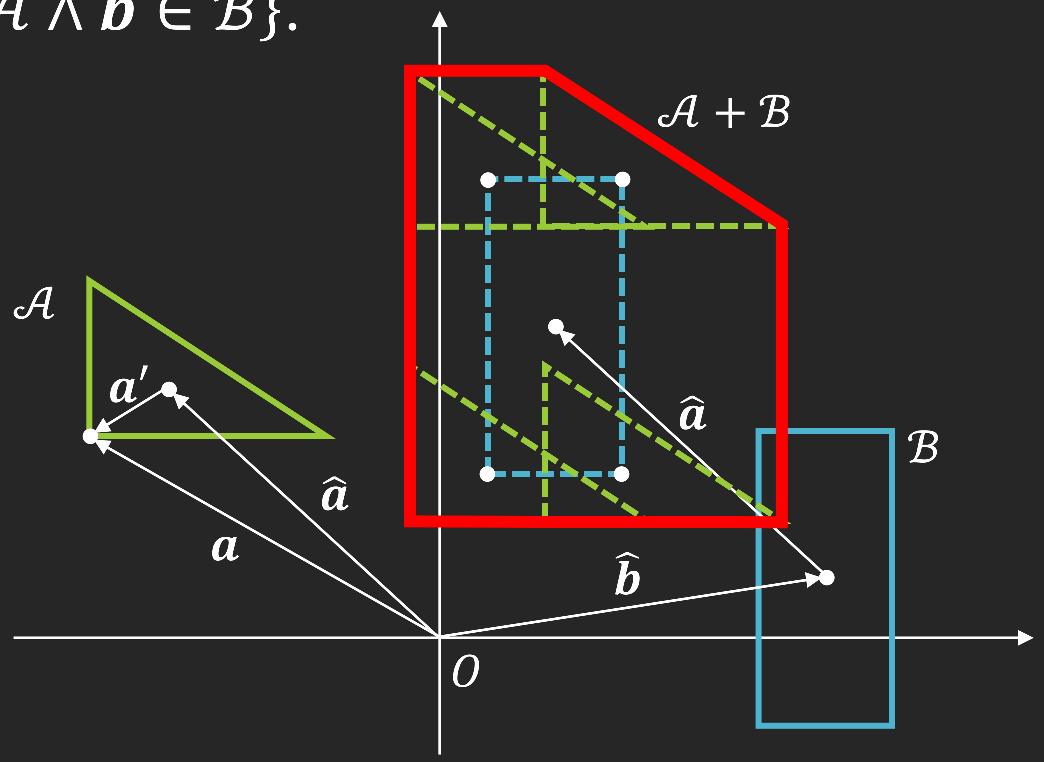
Concave shapes

  - 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**: $\mathcal{A} + \mathcal{B} = \{\boldsymbol{a} + \boldsymbol{b};\, \boldsymbol{a} \in \mathcal{A} \wedge \boldsymbol{b} \in \mathcal{B}\}$.
- How to draw Minkowski sum?
  - Choose some points $\hat{\boldsymbol{a}} \in \mathcal{A} \wedge \hat{\boldsymbol{b}} \in \mathcal{B}$.
  - Then, $\forall \boldsymbol{a} \in \mathcal{A}\ \exists \boldsymbol{a}'$ s.t. $\boldsymbol{a} = \hat{\boldsymbol{a}} + \boldsymbol{a}'$.
  - Therefore, for each $\boldsymbol{a} \in \mathcal{A} \wedge \boldsymbol{b} \in \mathcal{B}$
    $$\boldsymbol{a} + \boldsymbol{b} = \hat{\boldsymbol{a}} + \hat{\boldsymbol{b}} + \boldsymbol{a}' + \boldsymbol{b}'$$
  - So, we draw $\mathcal{A} + \mathcal{B}$ around $\hat{\boldsymbol{a}} + \hat{\boldsymbol{b}}$:
    - Draw $\mathcal{B}$ around $\hat{\boldsymbol{a}} + \hat{\boldsymbol{b}}$.
    - Draw $\mathcal{A}$ around $\mathcal{B}$'s perimeter.
    - $\mathcal{A} + \mathcal{B}$ is the convex hull.



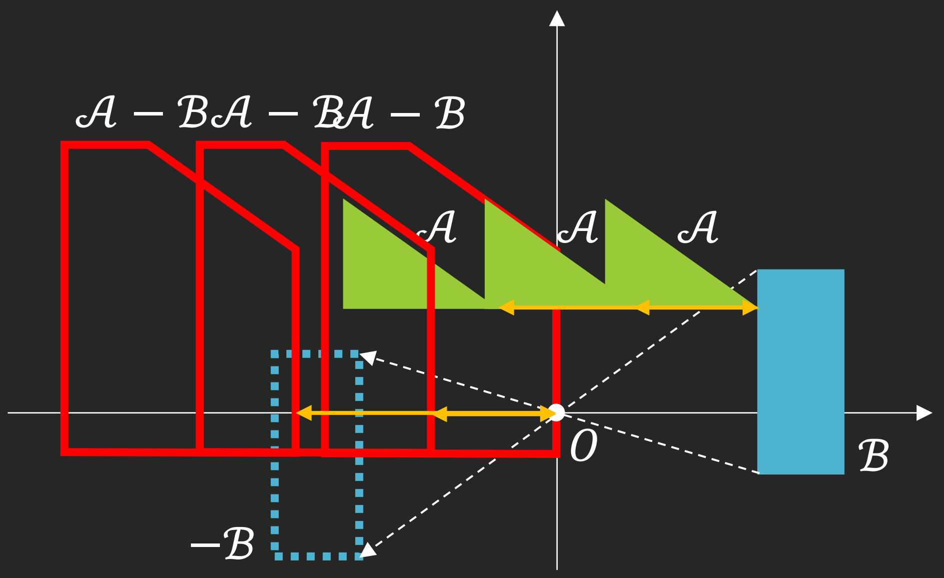
GJK: Minkowski difference

- **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 $\hat{\boldsymbol{a}} - \hat{\boldsymbol{b}}$, s.t. $\hat{\boldsymbol{a}} \in \mathcal{A} \wedge \hat{\boldsymbol{b}} \in \mathcal{B}$. But $\hat{\boldsymbol{a}} - \hat{\boldsymbol{b}} \in \mathcal{A} - \mathcal{B}$.

- **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 $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{A} - \mathcal{B}$ there exist $\boldsymbol{a}_u, \boldsymbol{a}_v \in \mathcal{A}$ and $\boldsymbol{b}_u, \boldsymbol{b}_v \in \mathcal{B}$ s.t.

  $\boldsymbol{u} = \boldsymbol{a}_u - \boldsymbol{b}_u$ and $\boldsymbol{v} = \boldsymbol{a}_v - \boldsymbol{b}_v$. Then, for $t \in \langle 0,1 \rangle$, we get

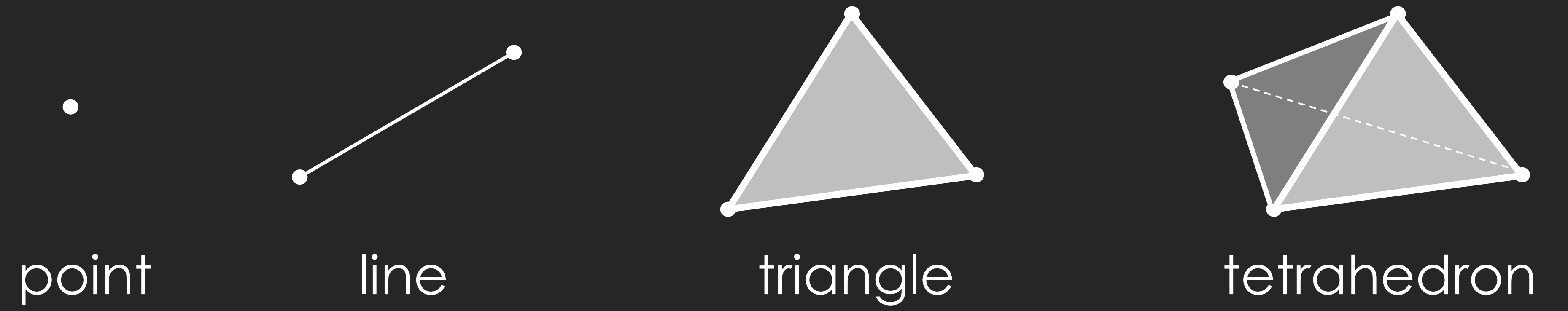
  $$\begin{aligned} \boldsymbol{u} + t(\boldsymbol{v} - \boldsymbol{u}) = (\boldsymbol{a}_u - \boldsymbol{b}_u) + t\big((\boldsymbol{a}_v - \boldsymbol{b}_v) - (\boldsymbol{a}_u - \boldsymbol{b}_u)\big) &= \\ \boldsymbol{a}_u - \boldsymbol{b}_u + t\boldsymbol{a}_v - t\boldsymbol{b}_v - t\boldsymbol{a}_u + t\boldsymbol{b}_u &= \\ \boldsymbol{a}_u + t(\boldsymbol{a}_v - \boldsymbol{a}_u) - \big(\boldsymbol{b}_u + t(\boldsymbol{b}_v - \boldsymbol{b}_u)\big). \end{aligned}$$

  $\mathcal{A}$ and $\mathcal{B}$ are convex => $\boldsymbol{a}_u + t(\boldsymbol{a}_v - \boldsymbol{a}_u) \in \mathcal{A}$, $\boldsymbol{b}_u + t(\boldsymbol{b}_v - \boldsymbol{b}_u) \in \mathcal{B}$ => $\boldsymbol{a}_u + t(\boldsymbol{a}_v - \boldsymbol{a}_u) - \big(\boldsymbol{b}_u + t(\boldsymbol{b}_v - \boldsymbol{b}_u)\big) \in \mathcal{A} - \mathcal{B}$ => $\mathcal{A} - \mathcal{B}$ is convex.

- **Consequence**: If the origin lies in the convex hull of points $\boldsymbol{a}_1, \dots, \boldsymbol{a}_n \in \mathcal{A} - \mathcal{B}$, then convex shapes $\mathcal{A}$ and $\mathcal{B}$ have non-empty intersection.

GJK: Simplex

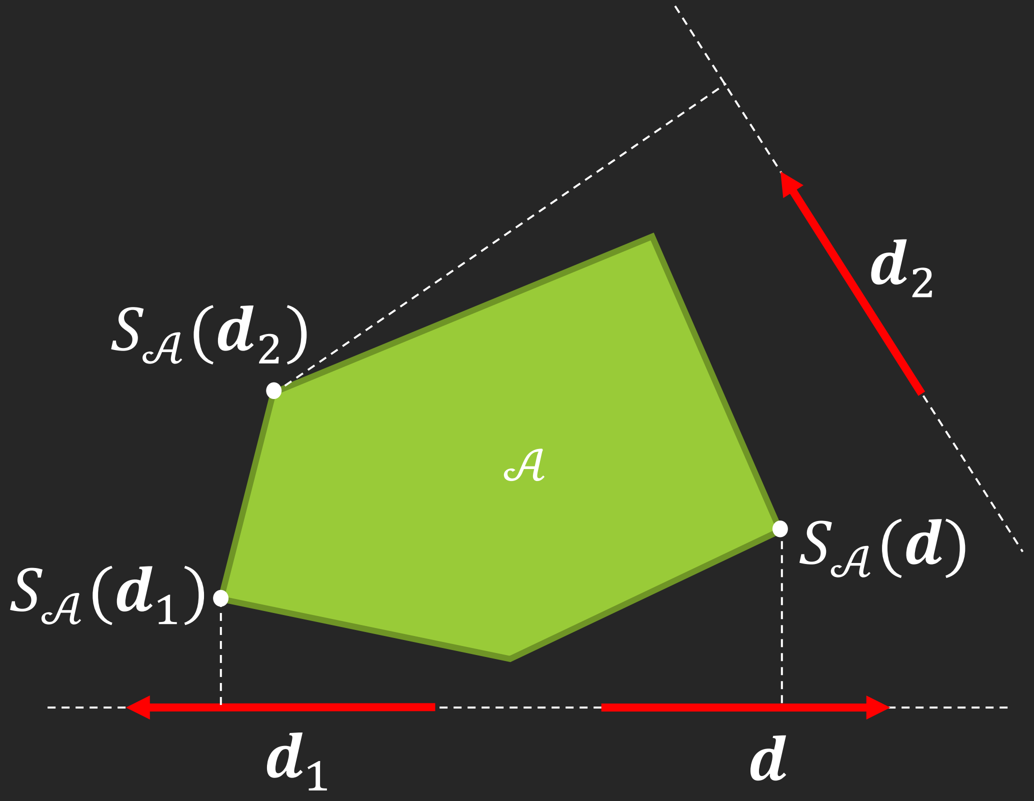
- A **simplex** is a convex hull of an affinely independent points.



- 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.

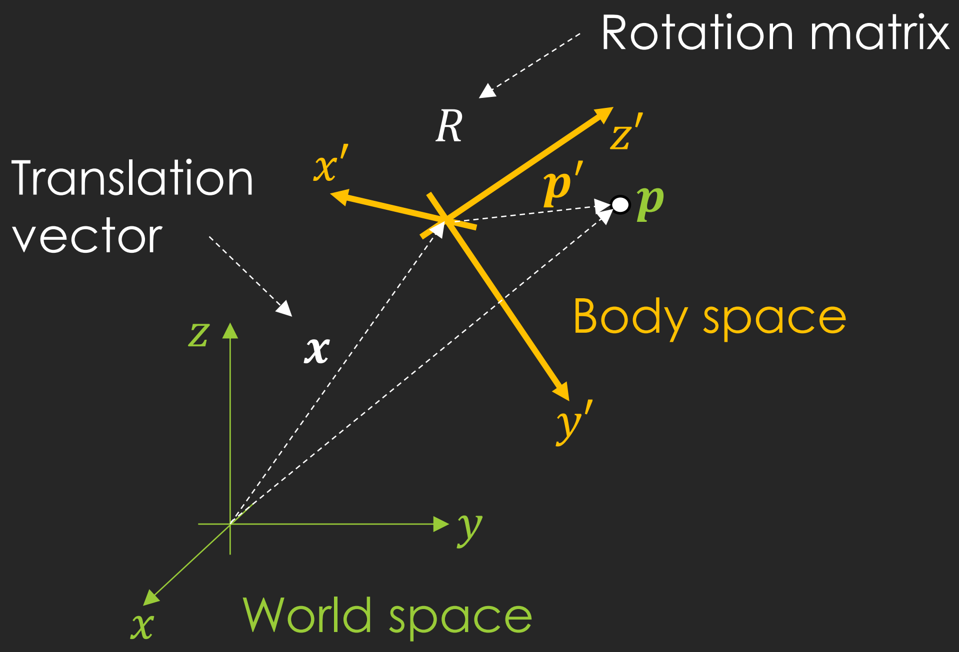
GJK: Support function

- Given a shape $\mathcal{A}$ and a non-zero vector $\boldsymbol{d}$, a **support function** $S_{\mathcal{A}}$ returns a point $S_{\mathcal{A}}(\boldsymbol{d}) \in \mathcal{A}$ s.t. $S_{\mathcal{A}}(\boldsymbol{d}) \cdot \boldsymbol{d} = \max\{\boldsymbol{x} \cdot \boldsymbol{d}; \boldsymbol{x} \in \mathcal{A}\}$.



GJK: Support function

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



$$\boldsymbol{p} = R\boldsymbol{p}' + \boldsymbol{x}$$

$$\boldsymbol{p}' = R^{\top}(\boldsymbol{p} - \boldsymbol{x})$$

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

GJK: Support function

- When a convex shape $\mathcal{A}$ is defined in body space $(R, \boldsymbol{x})$, then we denote $R\mathcal{A} + \boldsymbol{x}$ the corresponding convex shape in the world space.
  - More precisely: $R\mathcal{A} + \boldsymbol{x} = \{R\boldsymbol{p}' + \boldsymbol{x}\,; \boldsymbol{p}' \in \mathcal{A}\}$.
- **Lemma**: $S_{R\mathcal{A}+\boldsymbol{x}}(\boldsymbol{d}) = RS_{\mathcal{A}}(R^\top \boldsymbol{d}) + \boldsymbol{x}$, for each world-space vector $\boldsymbol{d} \neq \boldsymbol{0}$.

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

$$
\begin{aligned}
(R\boldsymbol{p}' + \boldsymbol{x}) \cdot \boldsymbol{d} = (R\boldsymbol{p}') \cdot \boldsymbol{d} + \boldsymbol{x} \cdot \boldsymbol{d} \\
&= \boldsymbol{d}^\top (R\boldsymbol{p}') + \boldsymbol{x} \cdot \boldsymbol{d} \\
&= (\boldsymbol{d}^\top R)\boldsymbol{p}' + \boldsymbol{x} \cdot \boldsymbol{d} \\
&= (R^\top \boldsymbol{d})^\top \boldsymbol{p}' + \boldsymbol{x} \cdot \boldsymbol{d} \\
&= \boldsymbol{p}' \cdot (R^\top \boldsymbol{d}) + \boldsymbol{x} \cdot \boldsymbol{d}.
\end{aligned}
$$

GJK: Support function

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

$= \max\{(R\boldsymbol{p}' + \boldsymbol{x}) \cdot \boldsymbol{d}; \boldsymbol{p}' \in \mathcal{A}\}$

$= \max\{\boldsymbol{p}' \cdot (R^\top \boldsymbol{d}) + \boldsymbol{x} \cdot \boldsymbol{d}; \boldsymbol{p}' \in \mathcal{A}\}$ according to (*)

$= \max\{\boldsymbol{p}' \cdot (R^\top \boldsymbol{d})\,; \boldsymbol{p}' \in \mathcal{A}\} + \boldsymbol{x} \cdot \boldsymbol{d}$

$= S_{\mathcal{A}}(R^\top \boldsymbol{d}) \cdot R^\top \boldsymbol{d} + \boldsymbol{x} \cdot \boldsymbol{d}$

$= (R S_{\mathcal{A}}(R^\top \boldsymbol{d}) + \boldsymbol{x}) \cdot \boldsymbol{d}$ according to (*)

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

GJK: Support function examples

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

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

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

$$S_{\mathcal{A}}(\boldsymbol{d}) = \left(\operatorname{sgn}(\boldsymbol{d}_x)\, s_x, \operatorname{sgn}\left(\boldsymbol{d}_y\right) s_y, \operatorname{sgn}(\boldsymbol{d}_z)\, s_z\right)^{\mathsf{T}}$$

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:

$$S_{\mathcal{A}}(\boldsymbol{d}) = \begin{cases} \left(\frac{r}{\sigma}\boldsymbol{d}_x, \frac{r}{\sigma}\boldsymbol{d}_y, \operatorname{sgn}(\boldsymbol{d}_z)\, h\right)^{\top} & \text{if } \sigma > 0 \\ (0,0, \operatorname{sgn}(\boldsymbol{d}_z)\, h)^{\top} & \text{otherwise} \end{cases}$$

where $\sigma = \sqrt{\boldsymbol{d}_x^2 + \boldsymbol{d}_y^2}$, and $\operatorname{sgn}(a)$ was defined earlier.

- $\mathcal{A}$ is any convex **polytope** (e.g., point, line, triangle, convex polygon, tetrahedron, box, …) with vertices $\mathrm{V} = \{\boldsymbol{v}_1, \dots, \boldsymbol{v}_n\}$:

$$S_{\mathcal{A}}(\boldsymbol{d}) = \boldsymbol{v}_k \quad \text{s.t.} \quad \boldsymbol{v}_k \cdot \boldsymbol{d} = \max\{\boldsymbol{v}_i \cdot \boldsymbol{d}\,; \boldsymbol{v}_i \in \mathrm{V}\}$$

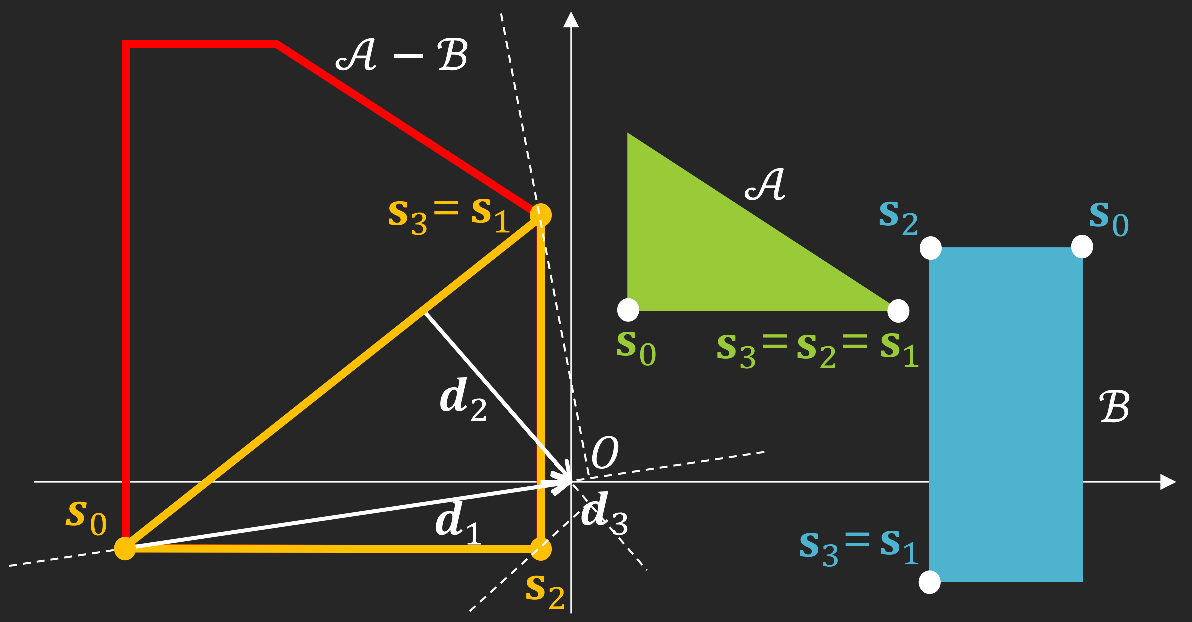
GJK: Support function

- **Lemma**: $S_{\mathcal{A}-\mathcal{B}}(\boldsymbol{d}) = S_{\mathcal{A}}(\boldsymbol{d}) - S_{\mathcal{B}}(-\boldsymbol{d})$.

  Proof:
  $$\begin{aligned} S_{\mathcal{A}-\mathcal{B}}(\boldsymbol{d}) \cdot \boldsymbol{d} &= \max\{(\boldsymbol{a}-\boldsymbol{b}) \cdot \boldsymbol{d}; \boldsymbol{a} \in \mathcal{A} \wedge \boldsymbol{b} \in \mathcal{B}\} \\ &= \max\{\boldsymbol{a} \cdot \boldsymbol{d}; \boldsymbol{a} \in \mathcal{A}\} - \min\{\boldsymbol{b} \cdot \boldsymbol{d}; \boldsymbol{b} \in \mathcal{B}\} \\ &= S_{\mathcal{A}}(\boldsymbol{d}) \cdot \boldsymbol{d} + \max\{\boldsymbol{b} \cdot (-\boldsymbol{d}); \boldsymbol{b} \in \mathcal{B}\} \\ &= S_{\mathcal{A}}(\boldsymbol{d}) \cdot \boldsymbol{d} + S_{\mathcal{B}}(-\boldsymbol{d}) \cdot (-\boldsymbol{d}) \\ &= (S_{\mathcal{A}}(\boldsymbol{d}) - S_{\mathcal{B}}(-\boldsymbol{d})) \cdot \boldsymbol{d}. \end{aligned}$$

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

GJK: The algorithm – intuition (2D case)



$\mathbf{s}_0, \mathbf{s}_0$ – closest points from the previous round (or random)

$$S = \{\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2\}$$

$$\mathbf{s}_0 = \mathbf{s}_0 - \mathbf{s}_0$$

$$\mathbf{s}_1 = S_{\mathcal{A}-\mathcal{B}}(\boldsymbol{d}_1) = S_{\mathcal{A}}(\boldsymbol{d}_1) - S_{\mathcal{B}}(-\boldsymbol{d}_1) = \mathbf{s}_1 - \mathbf{s}_1$$

$\mathbf{s}_1 \cdot \boldsymbol{d}_1 \geq 0$ => continue

$$\mathbf{s}_2 = S_{\mathcal{A}-\mathcal{B}}(\boldsymbol{d}_2) = S_{\mathcal{A}}(\boldsymbol{d}_2) - S_{\mathcal{B}}(-\boldsymbol{d}_2) = \mathbf{s}_2 - \mathbf{s}_2$$

$\mathbf{s}_2 \cdot \boldsymbol{d}_2 \geq 0$ => continue

$$\mathbf{s}_3 = S_{\mathcal{A}-\mathcal{B}}(\boldsymbol{d}_3) = S_{\mathcal{A}}(\boldsymbol{d}_3) - S_{\mathcal{B}}(-\boldsymbol{d}_3) = \mathbf{s}_3 - \mathbf{s}_3$$

$\mathbf{s}_3 \cdot \boldsymbol{d}_3 < 0$ => 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 $t \in \langle 0,1 \rangle$ s.t. $|X - O|^2 = |\mathbf{s}_1 + t(\mathbf{s}_2 - \mathbf{s}_1)|^2$ is minimal. So, solve the equation:

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

$$\frac{d}{dt}(\mathbf{s}_1 \cdot \mathbf{s}_1 + t2\mathbf{s}_1 \cdot (\mathbf{s}_2 - \mathbf{s}_1) + t^2(\mathbf{s}_2 - \mathbf{s}_1)^2) = 0$$

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

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

Also clip t to $\langle 0,1 \rangle$.

GJK: The algorithm – intuition (2D case)

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

$$\mathbf{s}_1 + t(\mathbf{s}_2 - \mathbf{s}_1) = (\mathbf{s}_1 - \mathbf{s}_1) + t((\mathbf{s}_2 - \mathbf{s}_2) - (\mathbf{s}_1 - \mathbf{s}_1))$$
$$= \underbrace{\mathbf{s}_1 + t(\mathbf{s}_2 - \mathbf{s}_1)}_{\in \mathcal{A}} - \underbrace{\left(\mathbf{s}_1 + t(\mathbf{s}_2 - \mathbf{s}_1)\right)}_{\in \mathcal{B}}$$

GJK: The algorithm

Choose some $\boldsymbol{p} \in \mathcal{A} - \mathcal{B}$. // Usually, $\boldsymbol{p}$ comes from the previous frame.

$S = \emptyset$ // We start with the empty simplex.

$\boldsymbol{s} = S_{\mathcal{A}-\mathcal{B}}(-\boldsymbol{p})$ // NOTE: Our direction vector $\boldsymbol{d}$ to the origin is just $-\boldsymbol{p}$.

while $|\boldsymbol{p}|^2 - \boldsymbol{p} \cdot \boldsymbol{s} > \epsilon^2$ do: // Proving termination condition: see [4].

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

  $\boldsymbol{p} = \boxed{\mathrm{closest\_to\_origin}}(\boxed{\mathrm{convex\_hull}(S \cup \{\boldsymbol{s}\})})$

Point, line, triangle, or tetrahedron.

Can be computed quickly for shapes

  $S =$ smallest $X \subseteq S \cup \{\boldsymbol{s}\}$ s.t. $\boldsymbol{p} \in \mathrm{convex\_hull}(X)$ // Reduce the simplex.

  $\boldsymbol{s} = S_{\mathcal{A}-\mathcal{B}}(-\boldsymbol{p})$

return $\boxed{\boldsymbol{a} \in \mathcal{A}, \boldsymbol{b} \in \mathcal{B} \text{ s.t. } \boldsymbol{p} = \boldsymbol{a} - \boldsymbol{b}.}$ // $|\boldsymbol{p}|$ is the closest distance.

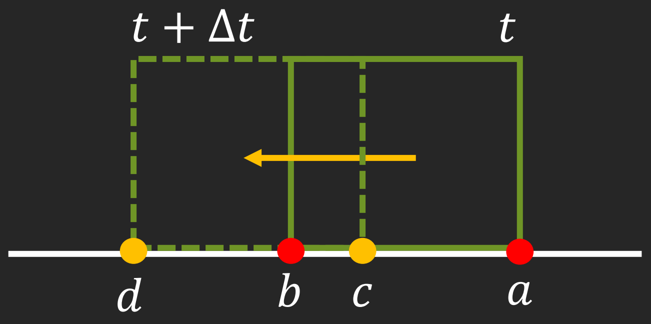
Caching collisions

Caching collisions

- Efficiency of the PGS algorithm for a constraint system depends on the initial value $\boldsymbol{\lambda}^0$.
- It is 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:

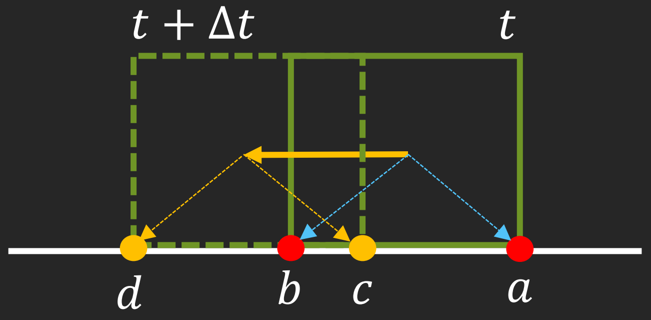
$a \rightarrow c, b \rightarrow d$

Word distance mapping:

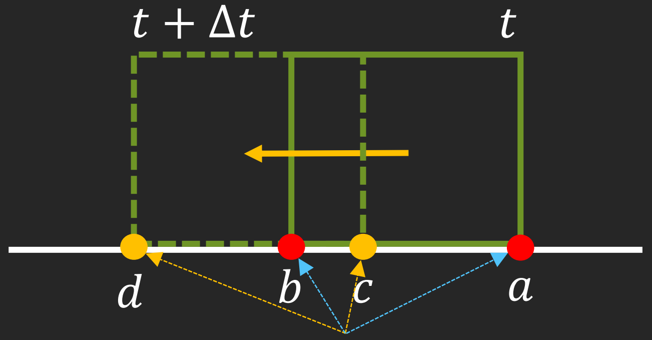
$b \rightarrow c$ (wrong), $a \rightarrow ?, ? \rightarrow d$

Imprecise.

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



Precise in space of the **box**



Imprecise in space of **ground**

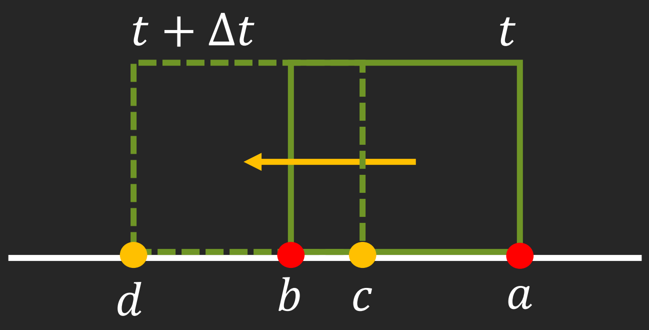
Caching collisions

- 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 also **comparison** and **hashing** of CollisionID instances.

Caching collisions



We get precise mapping:

$\mathrm{id}(a) = \mathrm{id}(c) \Rightarrow a \rightarrow c$

$\mathrm{id}(b) = \mathrm{id}(d) \Rightarrow b \rightarrow d$

Recommended approach

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

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

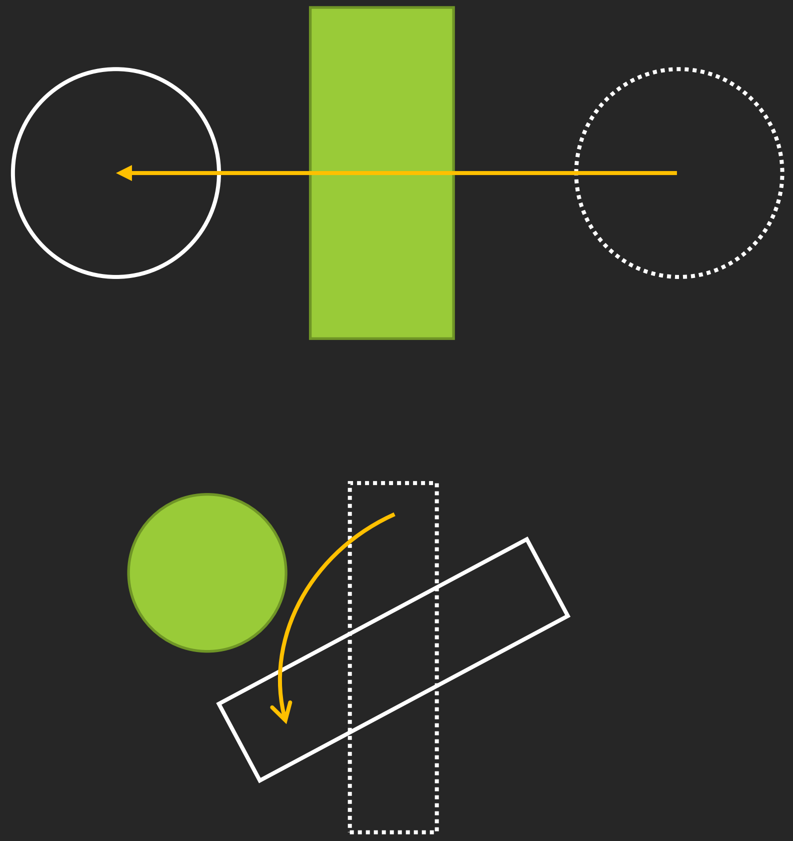
- And how to use the cache?

Caching collisions

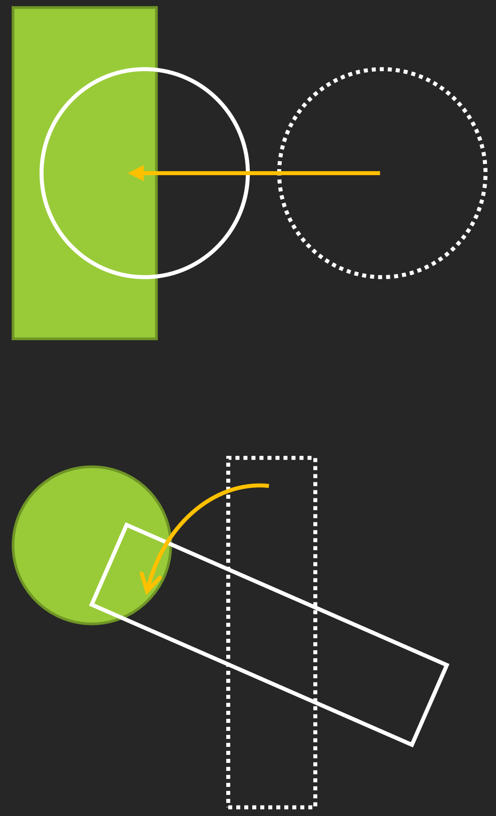
- Before solving the constraint system initialize $\boldsymbol{\lambda}^0$ s.t.
  - For each computed collision c and the corresponding element $\boldsymbol{\lambda}_i^0$:
    - Build the CollisionID instance id from c.
    - If id is present in the cache, then set $\boldsymbol{\lambda}_i^0$ to the value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

Computing collision time

Tunnelling and penetration



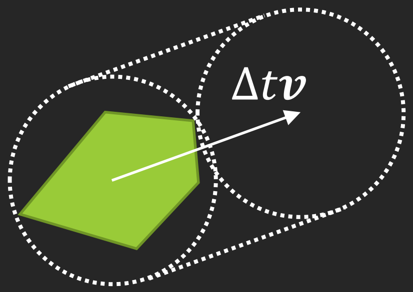
Tunnelling

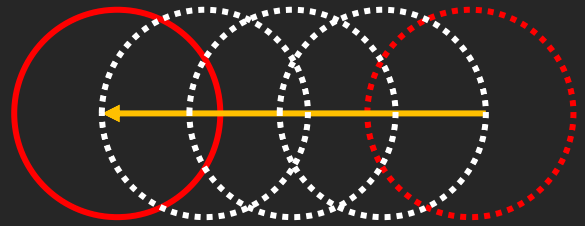


Penetration

Dealing with tunnelling and penetration

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



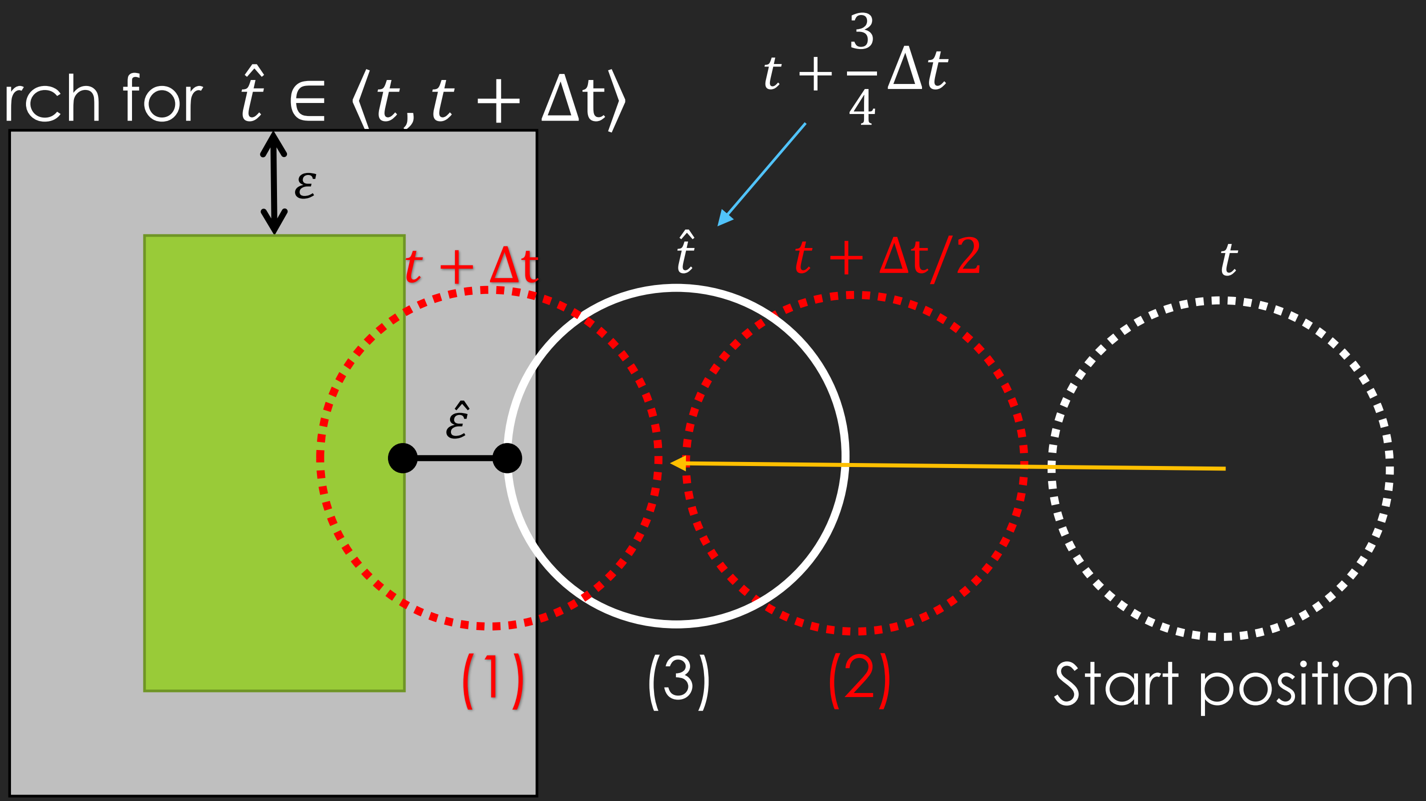


We move all bodies in each internal time step.

- 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 $\hat{t} \in \langle t, t + \Delta t \rangle$



- 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.