UNIVERSITÀ DEGLI STUDI DI MILANO

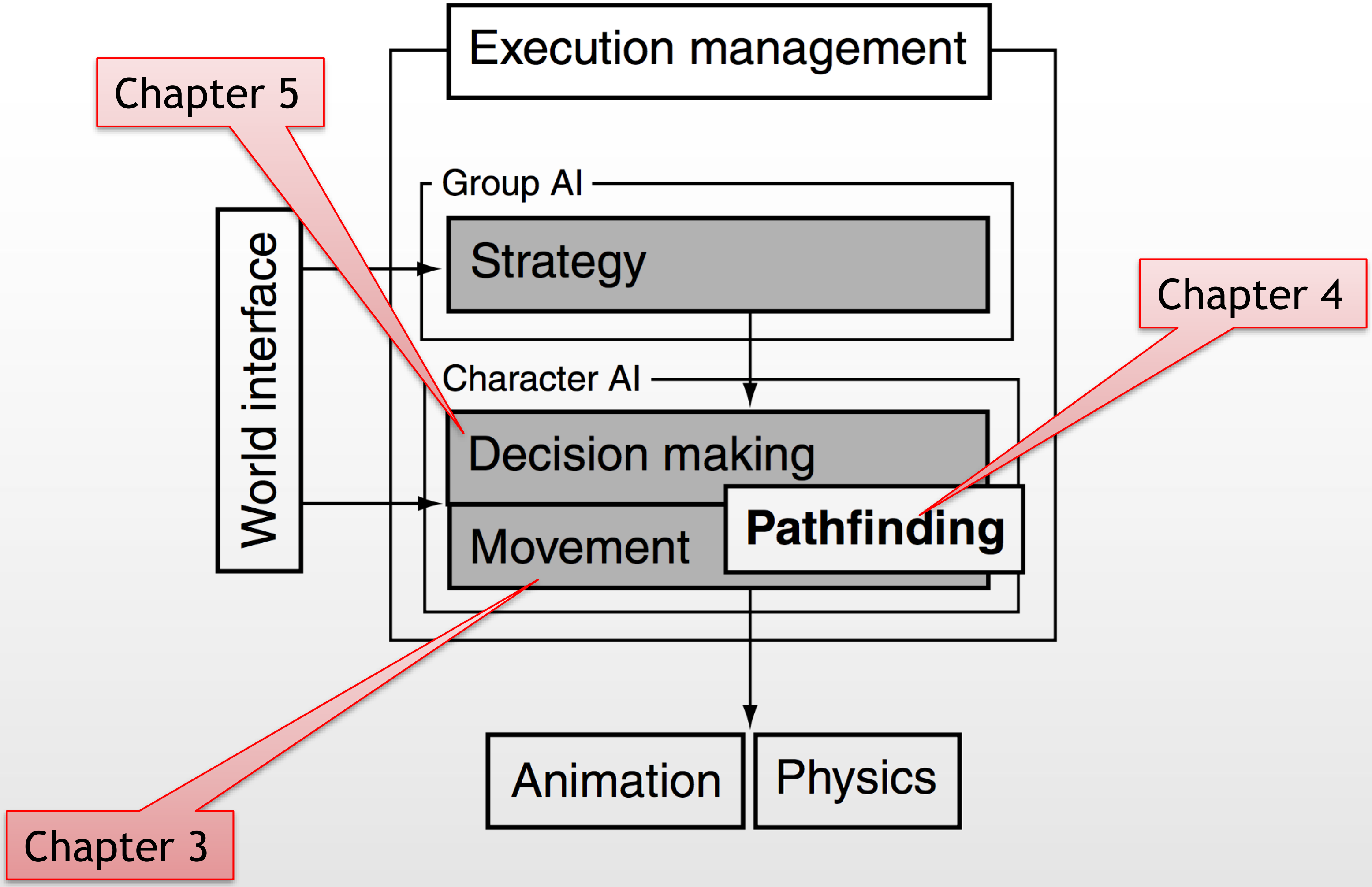
Movement

Part 1 - Kinematic

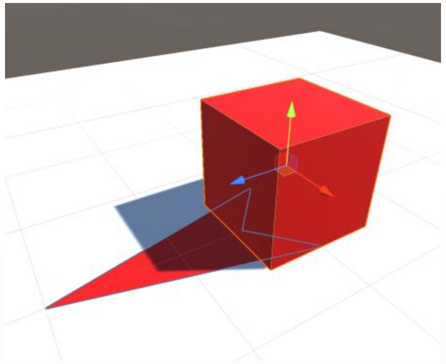
A.I. for Video Games

Dario Maggiorini (dario@di.unimi.it)

We are Rolling Back a Bit



3D vs 2D

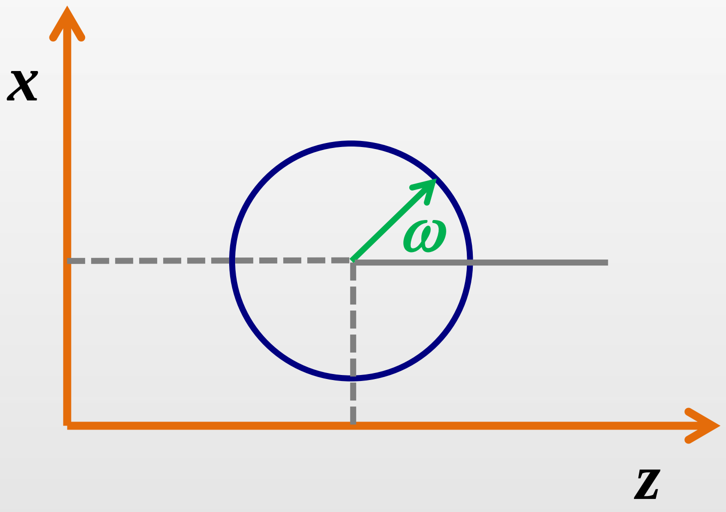


- Character position can be modeled as a point with orientation
- 3D movement is not required in many cases
  - Characters are constrained on a surface
- 2D suffices for most surface-based games
- 2½D (3D position and 2D orientation) suffices for many others
- Full 3D is used mainly for flight simulators & space shooters

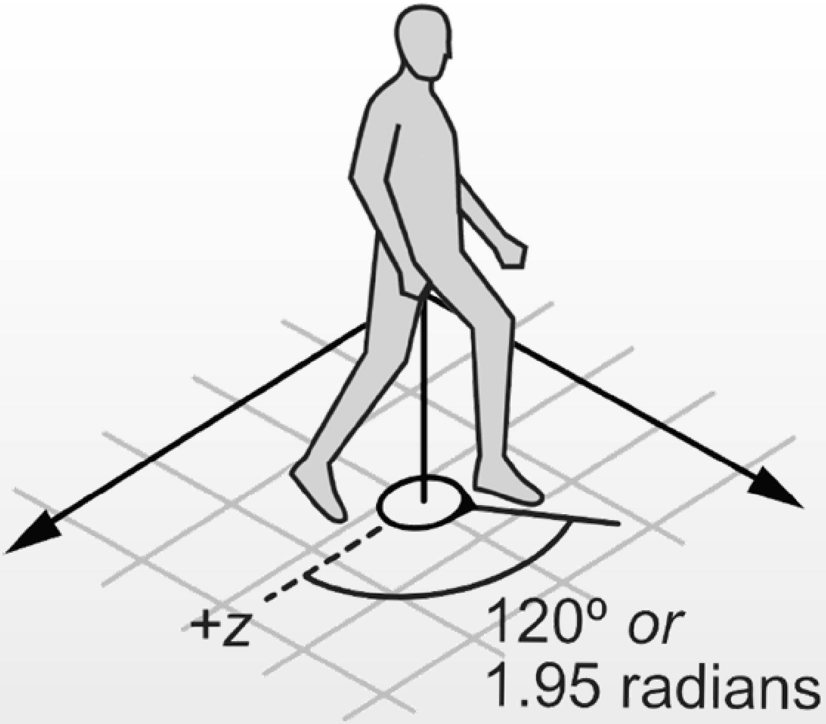
Statics

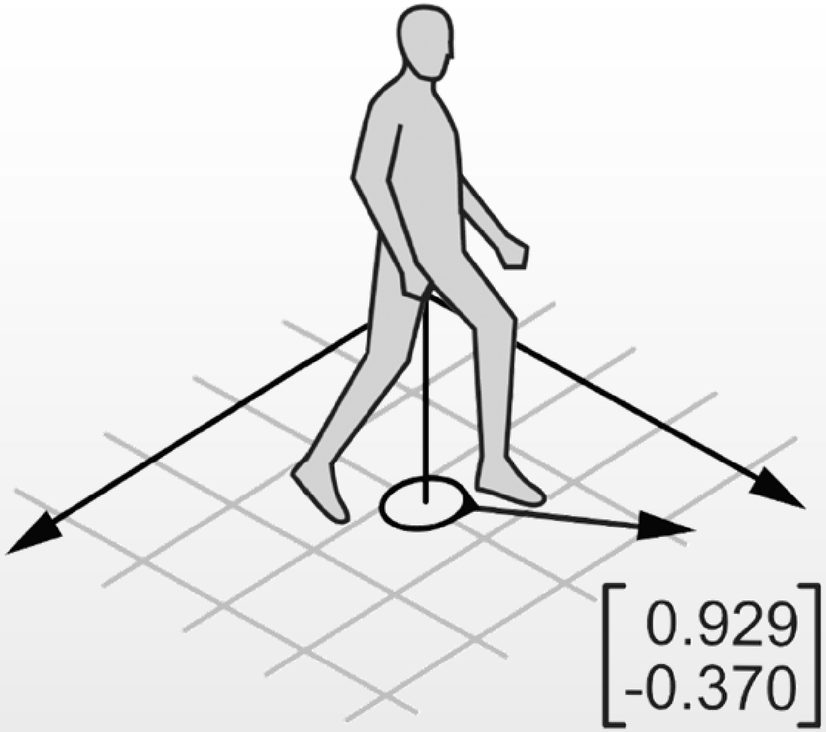
- Character position is a point
  - In gaming this is usually a vector in (x, z) coordinates

- Character orientation is given as $\vec{\omega_v} = \begin{bmatrix} \sin \omega_s \\ \cos \omega_s \end{bmatrix}$



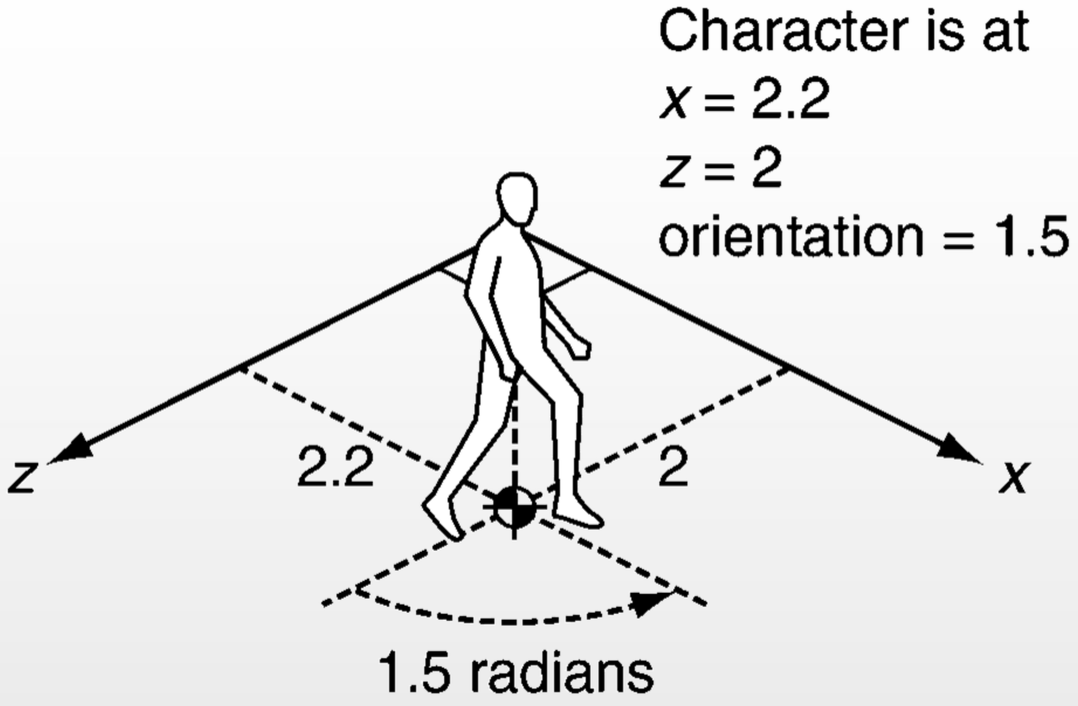
Statics



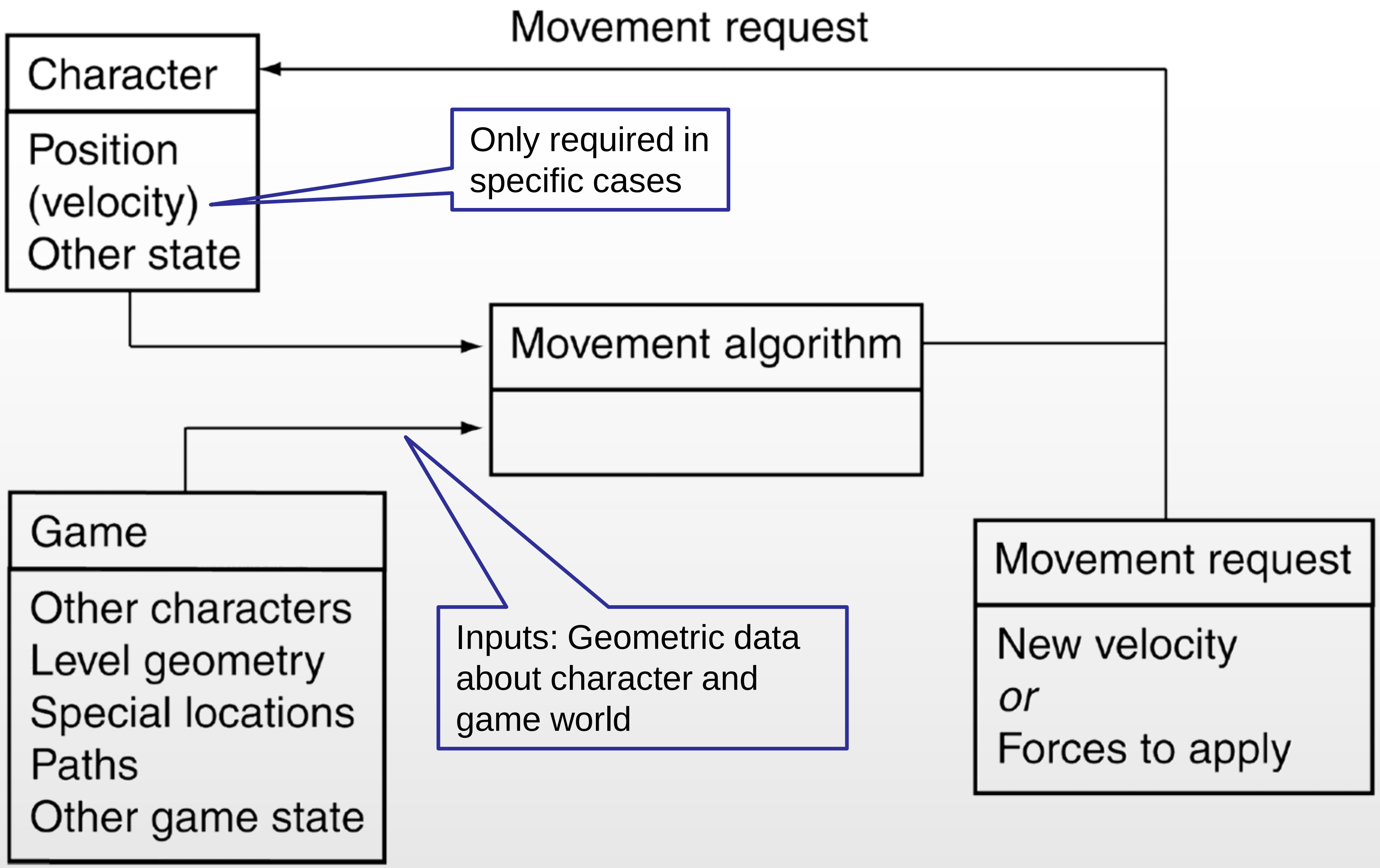


Coordinate System

- Right-hand coordinate system (typical of many game engines)
- xz is the orthonormal basis of the 2D space
- y axis (when used) direction is opposite the gravity («up»)
- Orientation is measured (counterclockwise) in radians



Movement Algorithm Structure

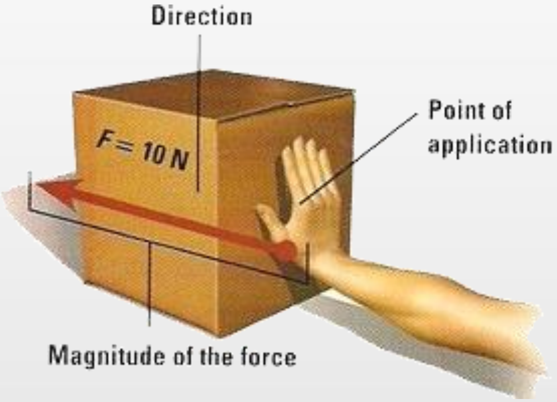


Moving Stuff Around

- Kinematic (or static)
  - Set a different position at every frame
  - A transform is enough

1. $v = v_0 + at$
2. $\Delta x = (\frac{v + v_0}{2})t$
3. $\Delta x = v_0 t + \frac{1}{2}at^2$
4. $v^2 = v_0^2 + 2a\Delta x$

- Dynamic
  - Using a RigidBody and push it around
  - Requires physics



NOTE: These are different concepts from the *isKinematic* flag you can set in a RigidBody

Kinematic Movement

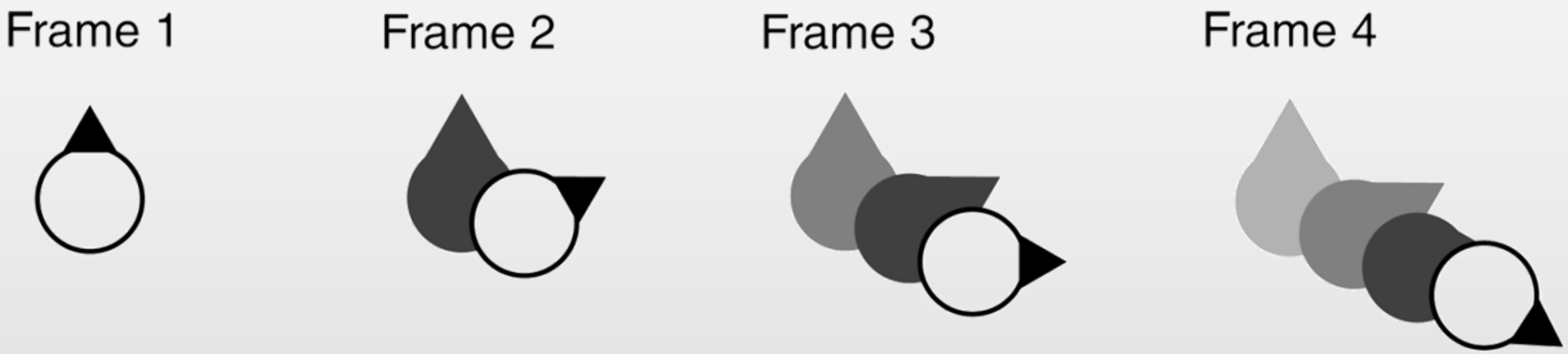
We can describe a moving character by:

1. Position
   - A 2- or 3-dimensional vector
2. Orientation
   - A 2-dimensional unit vector given by an angle, a single real value between 0 and 2π, or ω

3. Velocity (linear)
   - Same dimension of position
4. Rotation (angular velocity -> rad/s)
   - Same dimension of orientation

Updating Position and Orientation

- This is just about applying velocity to position and rotation to orientation
- Nevertheless, abrupt speed and rotation is not realistic so, in kinematic movement we smooth the effect over various frames
  - NOTE: this is not the same as applying a dynamic movement



Moving to a Position (Kinematic)

Set a new position at every update

```
public class KMoveTo : MonoBehaviour {

    public Transform destination;
    public float speed = 1f;

    void FixedUpdate () {
        if (destination) {
            transform.position += (destination.position - transform.position).normalized * speed * Time.deltaTime;
        }
    }
}
```

A gameobject in the position where we want to go

Not your usual friendly neighborhood update

Calculates a unit vector from position to destination

FixedUpdate is (tentatively) run by unity every fixed, and configurable, amount of milliseconds. It is advisable to put all movement operations inside FixedUpdate for a smoother result

DO NOT (For Your Own Safety)

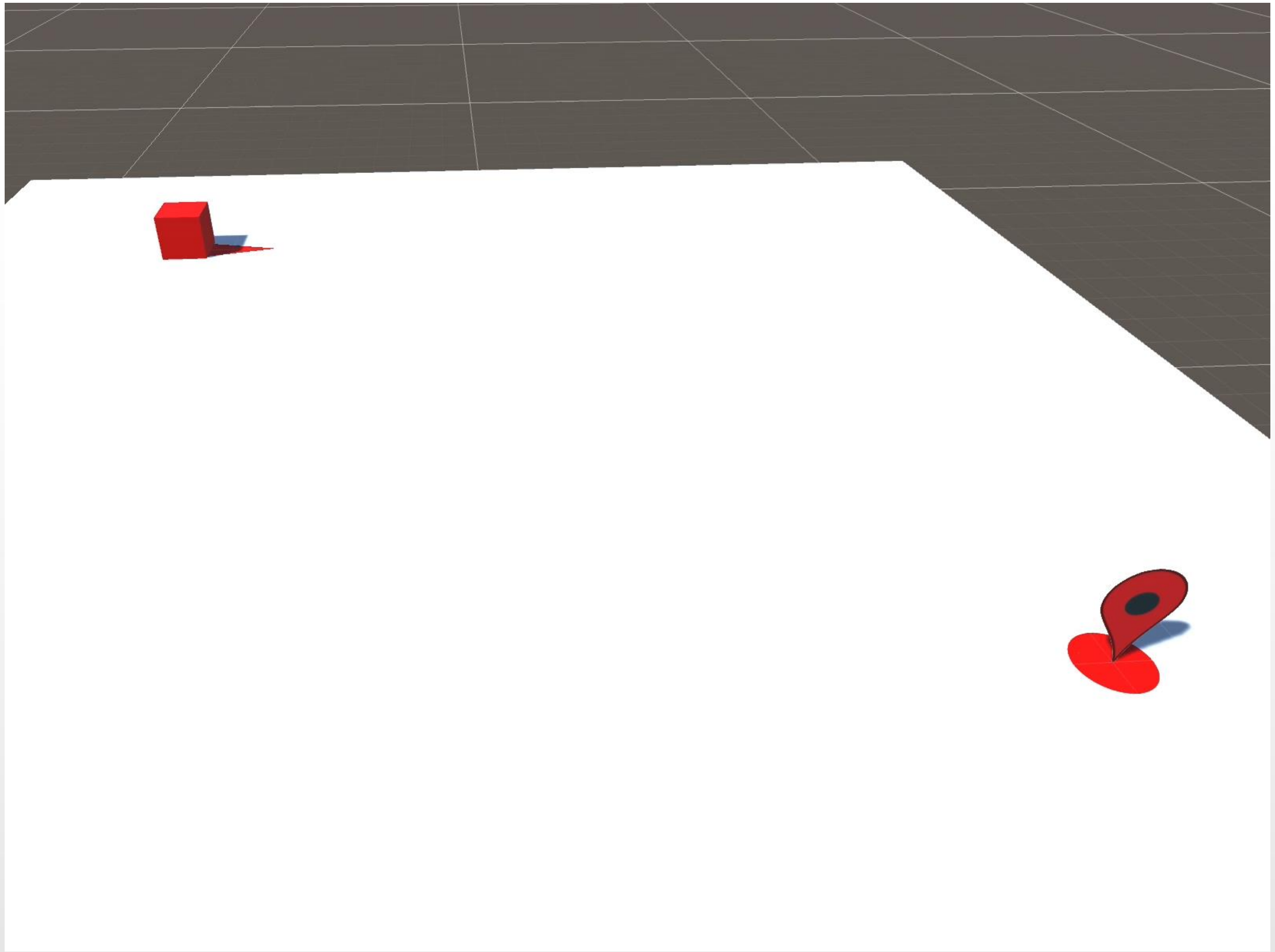
- **NEVER Mix kinematic and dynamic approaches**

–i.e., **NEVER** brutalize the position of a *GameObject* with a *RigidBody* using its *Transform*

Because you will get unpredictable results

… and thank me later for this

Nice Try



Moving to a Position (Kinematic)

- Set a new position at every update

```
public class KMoveTo : MonoBehaviour {

    public Transform destination;
    public float speed = 1f;

    void FixedUpdate () {
        if (destination) {
            transform.position += (destination.position - transform.position).normalized * speed * Time.deltaTime;
        }
    }
}
```

- Way too many errors here!
  - Not “looking” in the right direction
  - Not being able to stop
  - Sinking into the ground

Why These Errors?

- It is extremely context dependent
  - We are moving in 3D but using a 2D plane
  - Target and destination have different sizes and centers
  - Approximation errors are in the way

- (Very) easy solution: set all gameobjects geometric reference on the ground (plane)
  - Changing assets because of software constraints can be done but your 3D artist will not ne happy
  - We should try to work around that

Looking in the Right Direction

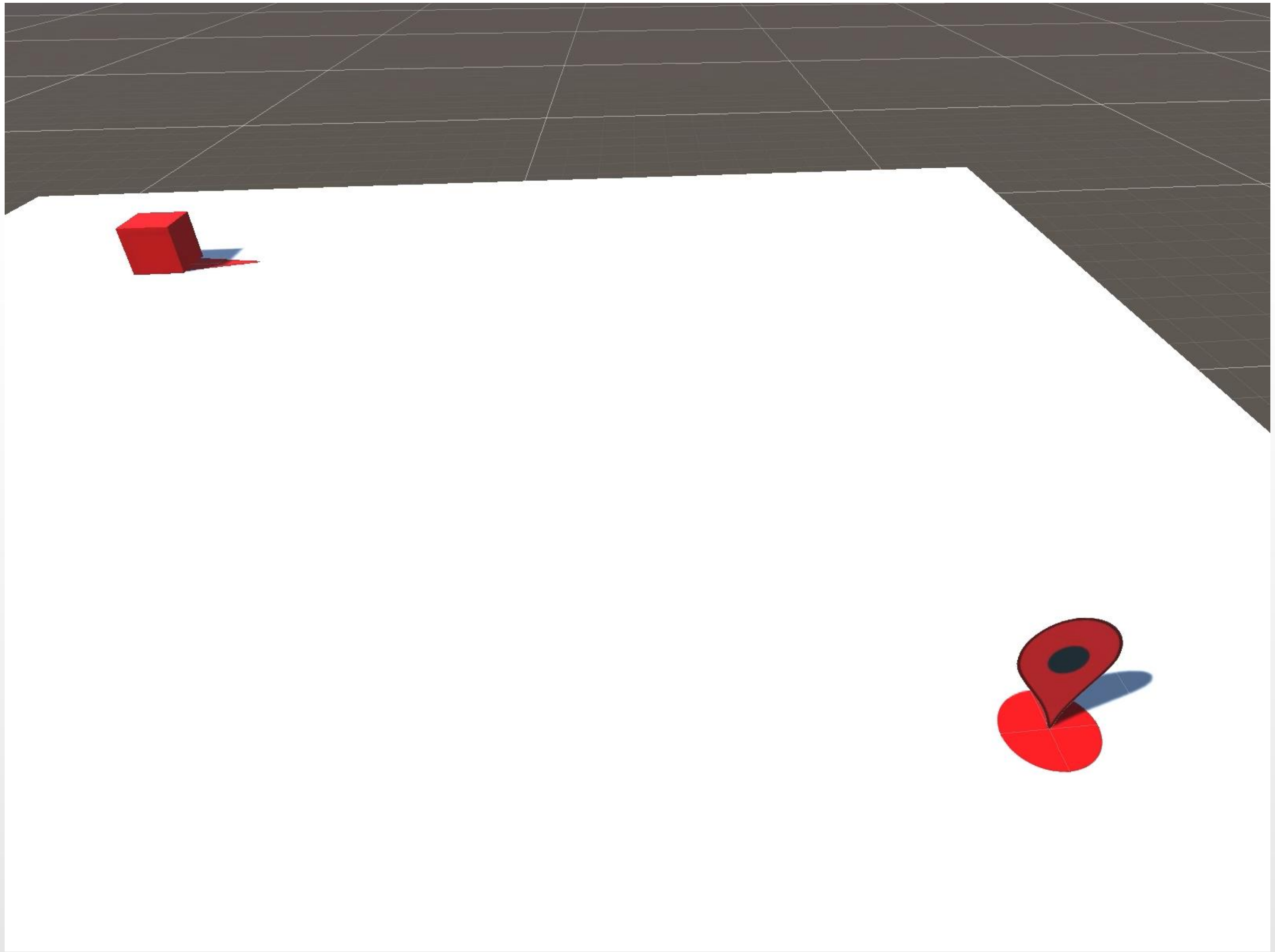
- We can use the LookAt Method

```
public class KMoveTo : MonoBehaviour {

    public Transform destination;
    public float speed = 1f;

    void FixedUpdate () {
        if (destination) {
            transform.LookAt (destination.position);
            transform.position += (destination.position - transform.position).normalized * speed * Time.deltaTime;
        }
    }
}
```

This is the only change

UNIVERSITÀ DEGLI STUDI DI MILANO

Now Facing the Destination, But Still Bad



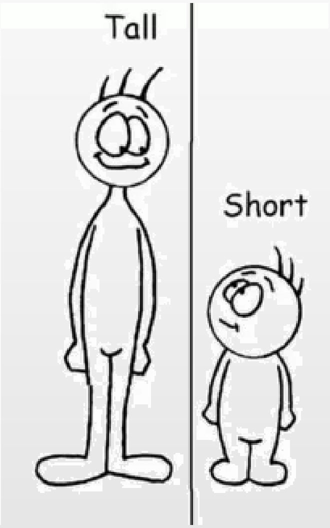
Looking in the Right Direction

- We can use the LookAt Method

```
public class KMoveTo : MonoBehaviour {

    public Transform destination;
    public float speed = 1f;

    void FixedUpdate () {
        if (destination) {
            transform.LookAt (destination.position);
            transform.position += (destination.position - transform.position).normalized * speed * Time.deltaTime;
        }
    }
}
```

Since we are looking toward the destination, this is the same as transform.forward

- The LookAt method is making my geometric center looking (orienting the forward direction) toward the destination
  - Since we have a transform as a parameter, the destination will be another geometric center
  - What if the two centers are at different heights?
    - Most probably, our agent will start to fly or sink



Stopping at the Right Distance

- Just add some tolerance

```
public class KMoveTo : MonoBehaviour {

    public Transform destination;
    public float speed = 2f;
    public float stopAt = 0.01f;

    void FixedUpdate () {
        if (destination) {
            Vector3 toDestination = (destination.position - transform.position);
            if (toDestination.magnitude > stopAt) {
                transform.LookAt (destination.position);
                transform.position += toDestination.normalized * speed * Time.deltaTime;
            }
        }
    }
}
```

As small as you need

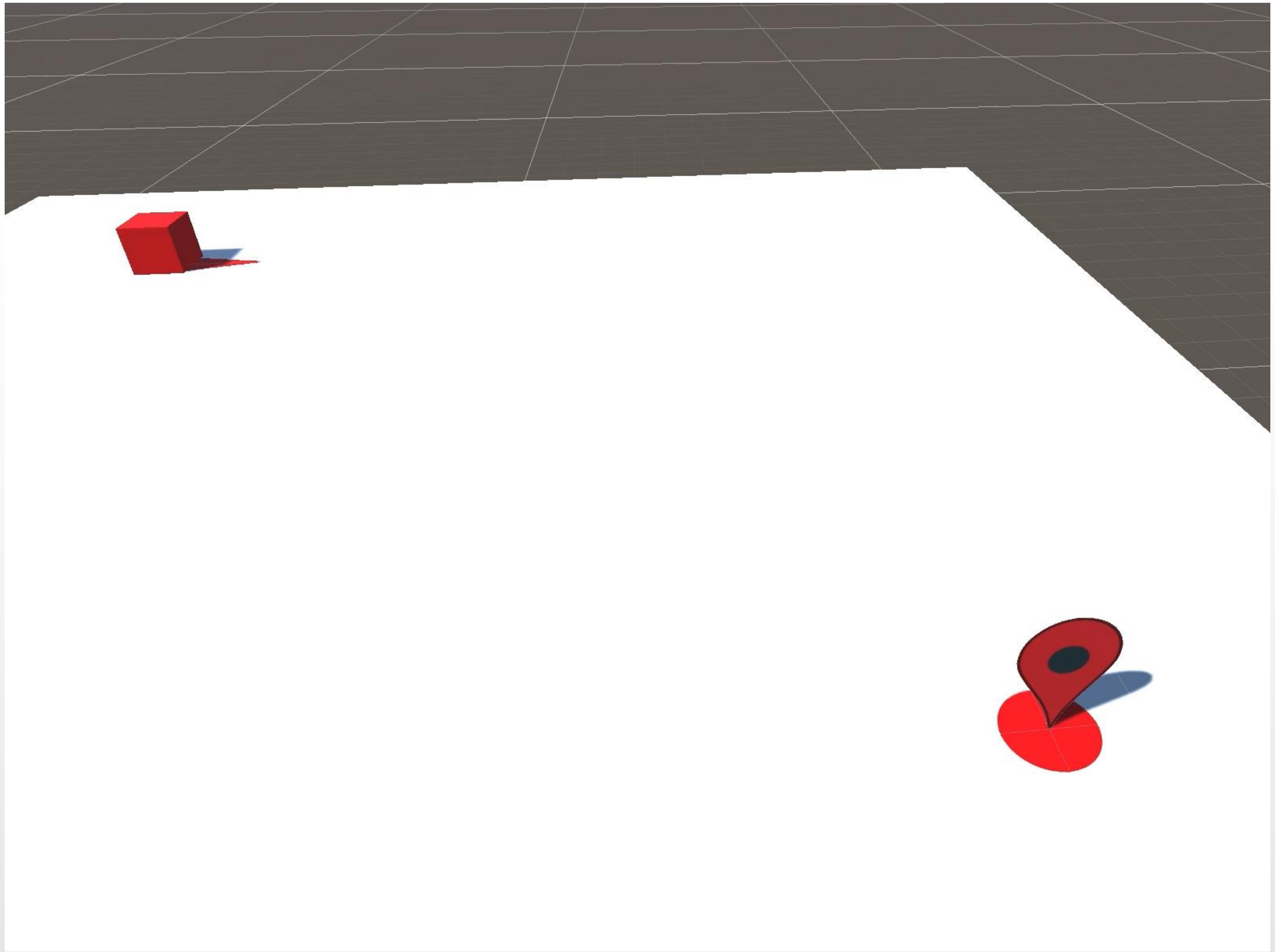
Vector from source to destination.
Its magnitude is the distance we must cover

We move the agent only if it is not closer to the destination that the tolerance

NOTE: in this example we are giving for granted that the destination is reachable.

What if it is flying and we want to go to the closest point on the ground?

Still Sinking, But Now Stops Without Oscillations



Avoid Sinking

- Option A: adjust target height to our height
- Option B: adjust rotation after LookAt

```
public class KMoveTo : MonoBehaviour {

    public Transform destination;
    public float speed = 2f;
    public float stopAt = 0.01f;

    void FixedUpdate () {
        if (destination) {

            Vector3 toDestination = (destination.position - transform.position);
            if (toDestination.magnitude > stopAt) {

A               // option a : we look at the destination position but with "our" height
                // Vector3 verticalAdj = new Vector3 (destination.position.x, transform.position.y, destination.position.z);
                // transform.LookAt (verticalAdj);

B               // option b : we care only about rotation on vertical axis
                transform.LookAt (destination.position);
                Vector3 rotationAdj = new Vector3 (0f, transform.rotation.eulerAngles.y, 0f);
                transform.rotation = Quaternion.Euler (rotationAdj);

                transform.position += transform.forward * speed * Time.deltaTime;
            }
        }
    }
}
```

Calculates an adjusted destination, and look there

Calculates a quaternion considering only the rotation on the y axis after the LookAt

A Really Working Solution

Source: KMoveTo
Folder: Movement/Kinematic

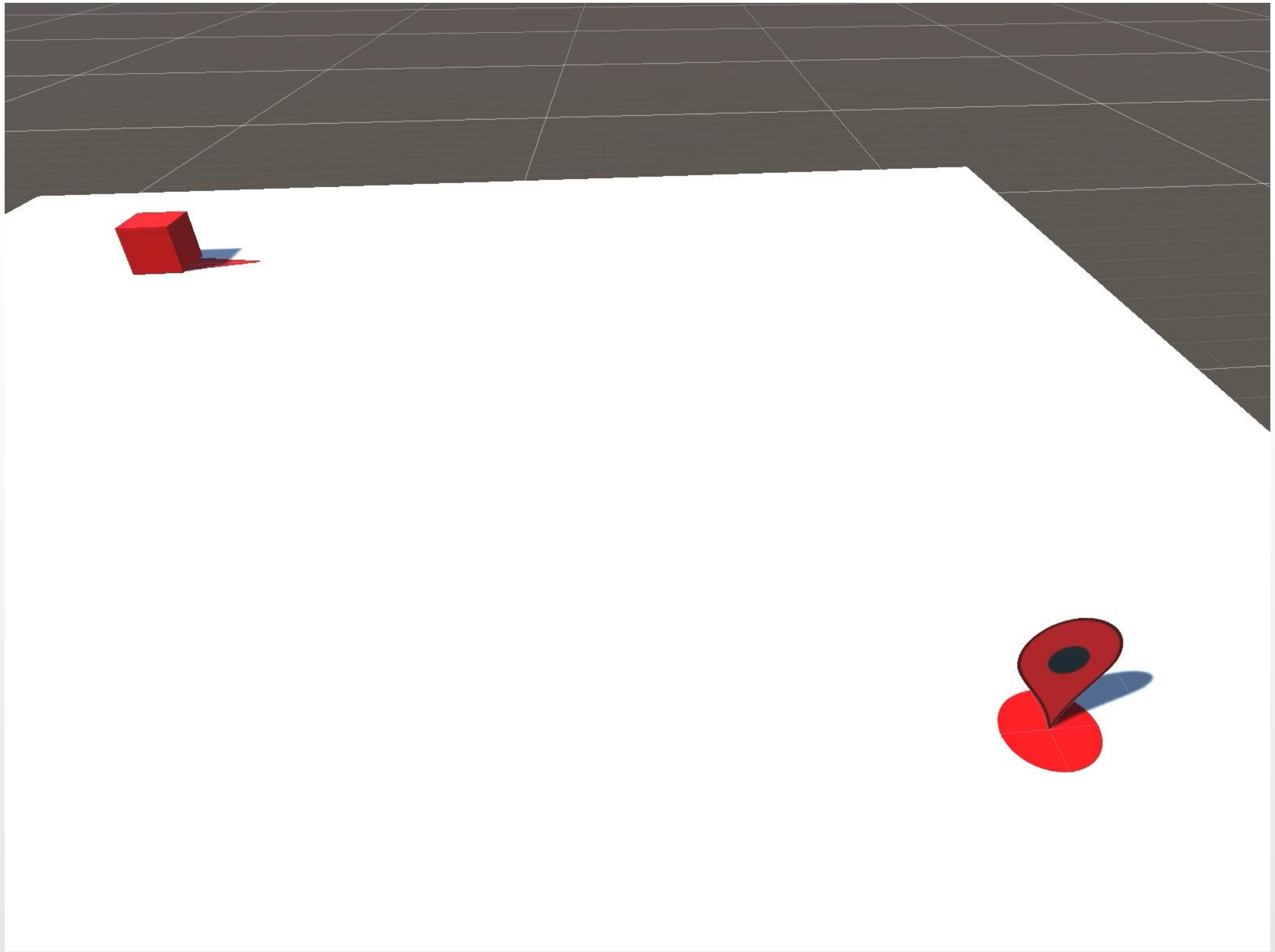
- We can leverage on the height adjustment of option A to also solve the problem of unreachable destinations

```
public class KMoveTo : MonoBehaviour {

    public Transform destination;
    public float speed = 2f;
    public float stopAt = 0.01f;

    void FixedUpdate () {
        if (destination) {

            Vector3 verticalAdj = new Vector3 (destination.position.x, transform.position.y, destination.position.z);
            Vector3 toDestination = (verticalAdj - transform.position);
            if (toDestination.magnitude > stopAt) {

                // option a : we look at the destination position but with "our" height
                // transform.LookAt (verticalAdj);

                // option b : we care only about rotation on vertical axis
                transform.LookAt (destination.position);
                Vector3 rotationAdj = new Vector3 (0f, transform.rotation.eulerAngles.y, 0f);
                transform.rotation = Quaternion.Euler (rotationAdj);

                transform.position += transform.forward * speed * Time.deltaTime;
            }
        }
    }
}
```

By considering the distance to the adjusted destination we will be stopping at the closest location on the supporting plane (ground)

It Seems Fine now

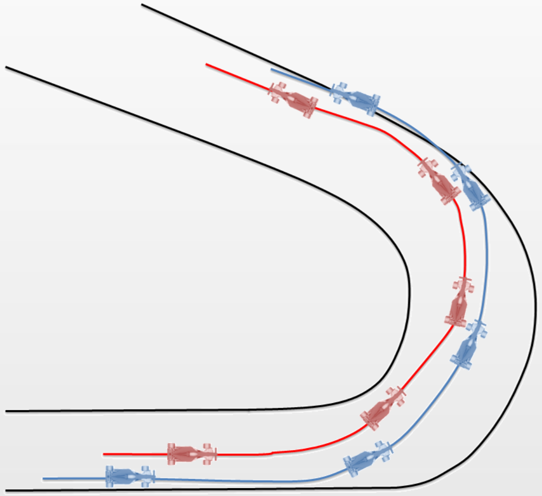
Scene: Move To
Folder: Movement/Kinematic



Nobody is Perfect

- We are supposing gravity is along the vertical axis
(or that the supporting surface is horizontal, which is the same thing)
  - Maybe, sometimes, it is not working like that ...

- Moreover, straight movement and instant turns are not always what you really need



Kinematic Algorithms

- Kinematic movement algorithms use static data (position and orientation, no velocities) and output a desired velocity
  - Read: they tell you where to go “at full speed”

- Seek
- Flee
- Arriving
- Wandering
- … and many more

Seek

- Inputs:
  - static data of character (position and orientation)
  - target position
- Calculates:
  - Direction from character to target
  - Requests a velocity along this line (orientation value is ignored)
- This is ok to chase something
  - It never reaches the target, but continues to seek. Otherwise, we need a more complicated approach

Flee

- Same as seek … but in the other direction

- Reverse the seek velocity vector to calculate next position

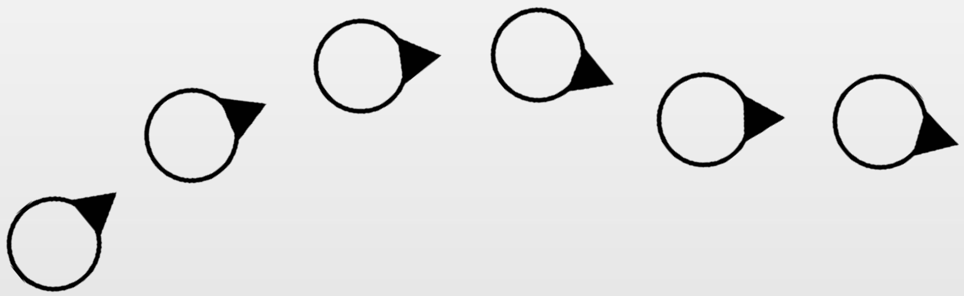
Arriving

- Seek/flee moves character at max speed:
  - No good when character needs to reach a target point, since it overruns and oscillates

- Possible solutions
  1. Arriving circle: stop if inside. High imprecision
  2. Define a range of velocities and decelerates when approaching target
     Smaller oscillations
  3. Arriving circle + range of velocities. Best solution with kinematic

Wandering

- Kinematic wander: move in the direction of current orientation at max speed
- Improvement: randomly modify orientation to obtain a more natural movement
  - Character moves forward in the facing direction at each frame



References

- On the textbook
  - § 3.1
  - § 3.2