UNIVERSITÀ DEGLI STUDI
DI MILANO

Movement

Part 2 - Dynamic

A.I. for Video Games

Dario Maggiorini (dario@di.unimi.it)

Dynamic Movement aka Steering

- Basically, kinematic is extended taking accelerations (both linear and angular into account)

- Uniform motion: $\vec{s} = \vec{v}.\mathrm{t}$
- Uniformly accelerated motion: $\vec{s} = \vec{v}.\mathrm{t} + \frac{1}{2}\vec{a}\mathrm{t}^2$

- Position += velocity * t + 0.5 * linear acceleration * t * t
- Orientation += rotation * t + 0.5 * angular acceleration * t * t

- Velocity += linear acceleration * t
- Rotation += angular acceleration * t

Moving to a Position (With a RigidBody)

- We have two options here:

1. Move the Rigidbody using its own MovePosition method
   - Once again, do NOT use the transform for this (… it may explode!)
   - This is not really “dynamic” to be honest

2. Push the Rigidbody toward the destination
   - When?
   - How much?
   - How do we stop?

Moving a Rigidbody

Source: DMoveTo
Folder: Movement/Dynamic

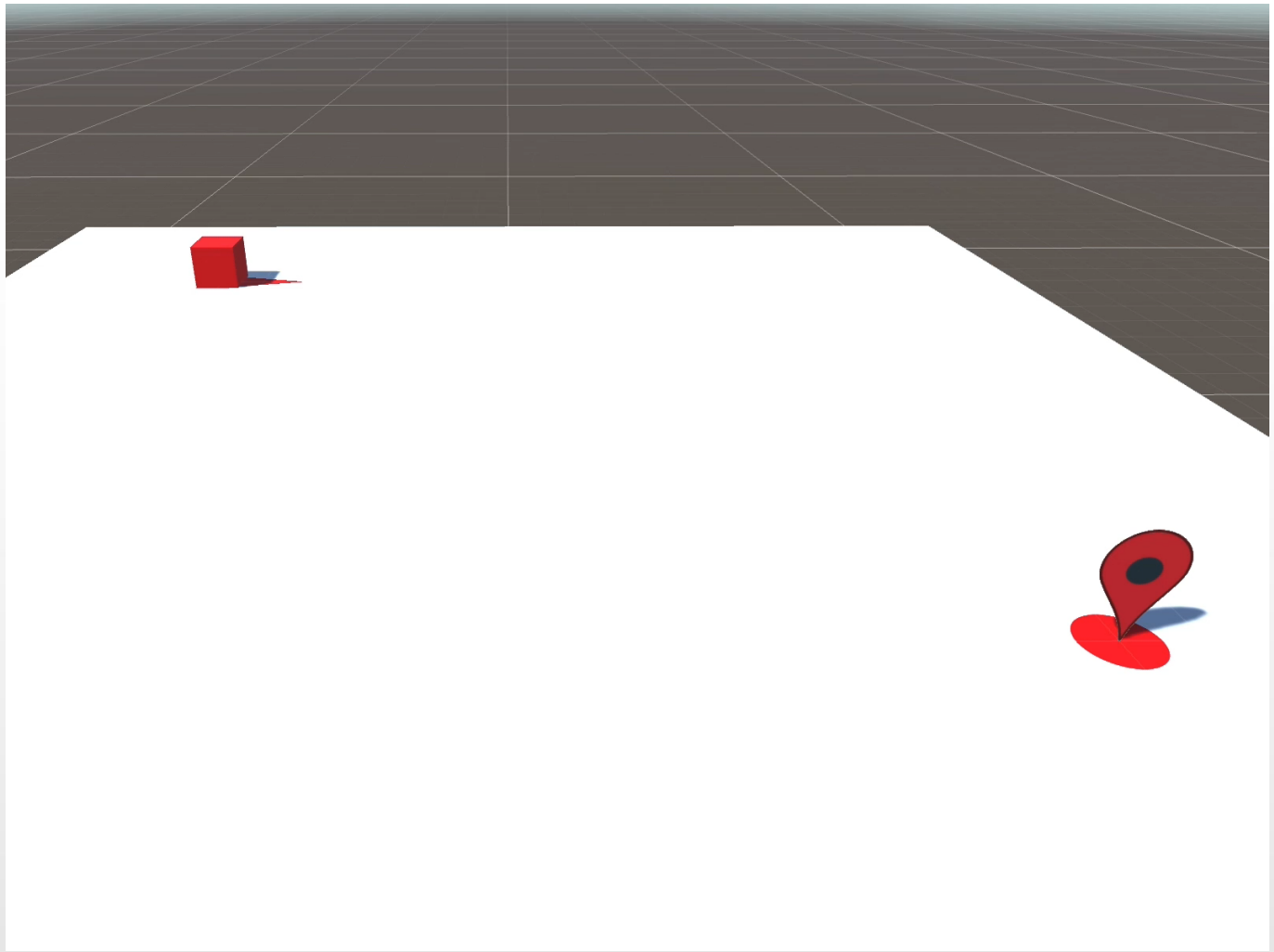
- Same thing as before, but now we have collisions!

```
[RequireComponent(typeof(Rigidbody))]

public class DMoveTo : 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) {
                // we keep only option a
                transform.LookAt (verticalAdj);
                Rigidbody rb = GetComponent<Rigidbody> ();
                rb.MovePosition(transform.position + transform.forward * speed * Time.deltaTime);
            }
        }
    }
}
```

The only change is here

Moving a Rigidbody

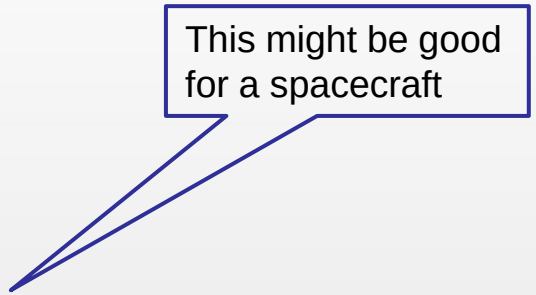
Scene: Move To
Folder: Movement/Dynamic



Pushing a Rigidbody

- Things are getting complicated here

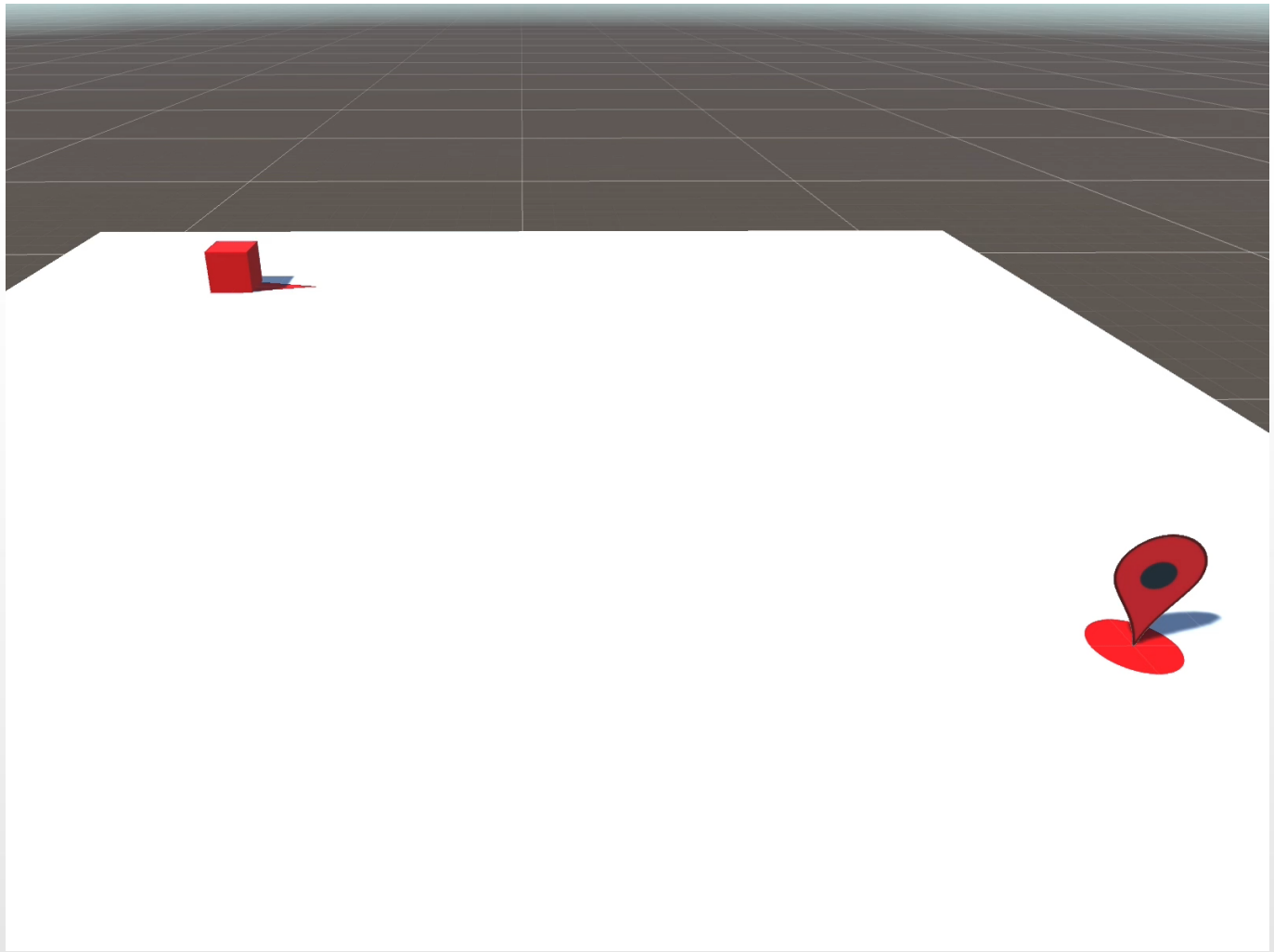
1. You cannot push (AddForce) at every update
   - Because that is going to be an acceleration
     (ask Mr. Newton for details)



2. You cannot push just once
   - Because linear drag will stop you
     Unless you disable gravity … but then you will bounce away!
   - And what about moving destinations?

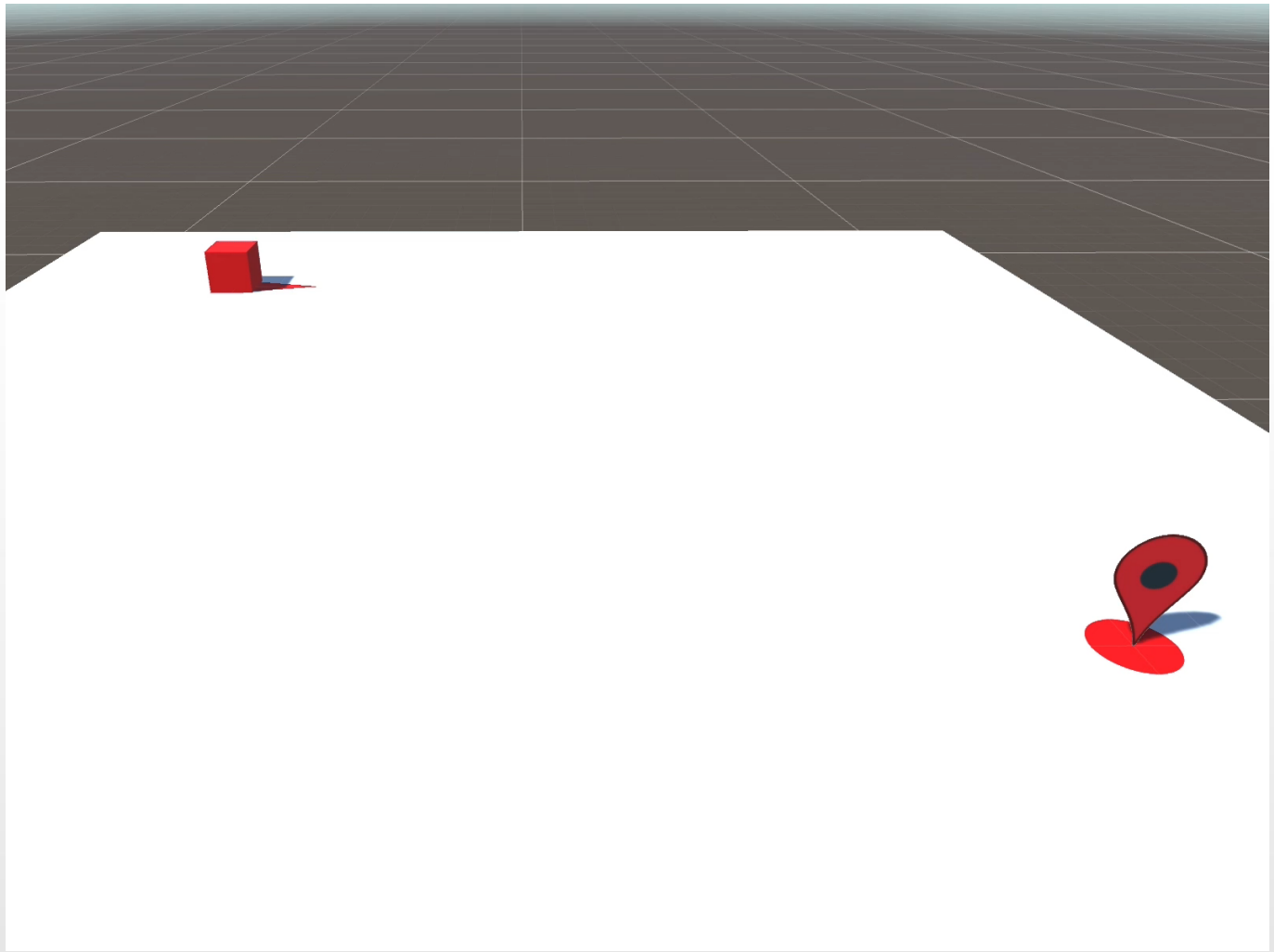
Pushing Too Much

Scene: Push To
Folder: Movement/Dynamic



Pushing Only Once (No Gravity Applied)

Scene: Push Once
Folder: Movement/Dynamic



The Real Problem

- As already said, straight movement and 90 degrees turns are not always what we really need
  - Actually, we **never** want them in a realistic simulation
- Pushing might not be the easiest way to solve this due to surface drag and the “kangaroo effect”
  - But, at the same time, we do not want to take off Rigidbodies and forfeit collisions
- Much easier for us, is to find a way to move the agent around in a believable way

Let's Put Acceleration in the Picture

- What do we need?
  - An acceleration (gas) value
  - A maximum speed

```
public float gas = 2f;
public float maxSpeed = 5f;
public float stopAt = 0.01f;

private float currentSpeed = 0f;

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) {
            transform.LookAt (verticalAdj);

            float t = Time.deltaTime;
            float x = currentSpeed * t + 0.5f * gas * t * t;
            currentSpeed += gas * t;
            currentSpeed = Mathf.Min (currentSpeed, maxSpeed);

            Rigidbody rb = GetComponent<Rigidbody> ();
            rb.MovePosition (transform.position + transform.forward * x);
        }
    }
}
```

This is just an extension of the previous script

Now we update speed based on acceleration, every time the agent will cover more space, up to a maximum speed

Let's Put Acceleration in the Picture

- What do we need?
  - An acceleration (gas) value
  - A maximum speed

```
public float gas = 2f;
public float maxSpeed = 5f;
public float stopAt = 0.01f;

private float currentSpeed = 0f;

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) {
            transform.LookAt (verticalAdj);

            float t = Time.deltaTime;
            float x = currentSpeed * t + 0.5f * gas * t * t;
            currentSpeed += gas * t;
            currentSpeed = Mathf.Min (currentSpeed, maxSpeed);

            Rigidbody rb = GetComponent<Rigidbody> ();
            rb.MovePosition (transform.position + transform.forward * x);
        }
    }
}
```

This is just an extension of the previous script

$x = v\,t + \frac{1}{2}\,a\,t^2$

$v = v_0 + a\,t$

Clamp to maximum speed

But We Also Need Brakes When We Get Close

- Whenever we are below a certain distance (brakeAt) we can apply a negative acceleration (brake) to reduce the speed up to a minimum
  - This minimum could be 0, but we might stop before reaching the destination

```
Vector3 verticalAdj = new Vector3 (destination.position.x, transform.position.y, destination.position.z);
Vector3 toDestination = (verticalAdj - transform.position);

if (toDestination.magnitude > stopAt) {
    transform.LookAt (verticalAdj);

    float t = Time.deltaTime;
    float x;

    if (toDestination.magnitude > brakeAt) {
        x = currentSpeed * t + 0.5f * gas * t * t;
        currentSpeed += gas * t;
        currentSpeed = Mathf.Min (currentSpeed, maxSpeed);
    } else {
        x = currentSpeed * t - 0.5f * brake * t * t;
        currentSpeed -= brake * t;
        currentSpeed = Mathf.Max (currentSpeed, minSpeed);
    }

    Rigidbody rb = GetComponent<Rigidbody> ();
    rb.MovePosition (transform.position + transform.forward * x);
}
```

Exactly same formulas as before, but now we have two cases

Source: DAccelerateTo
Folder: Movement/Dynamic

Little Improvement: Braking is Like Negative Gas

Source: DAccelerateAndBrakeTo
Folder: Movement/Dynamic

```
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) {
            transform.LookAt (verticalAdj);

            float currentGas = toDestination.magnitude > brakeAt ? gas : -brake;

            float t = Time.deltaTime;
            float x = currentSpeed * t + 0.5f * currentGas * t * t;
            currentSpeed += currentGas * t;
            currentSpeed = Mathf.Clamp (currentSpeed, minSpeed, maxSpeed);

            Rigidbody rb = GetComponent<Rigidbody> ();
            rb.MovePosition (transform.position + transform.forward * x);
        }
    }
}
```

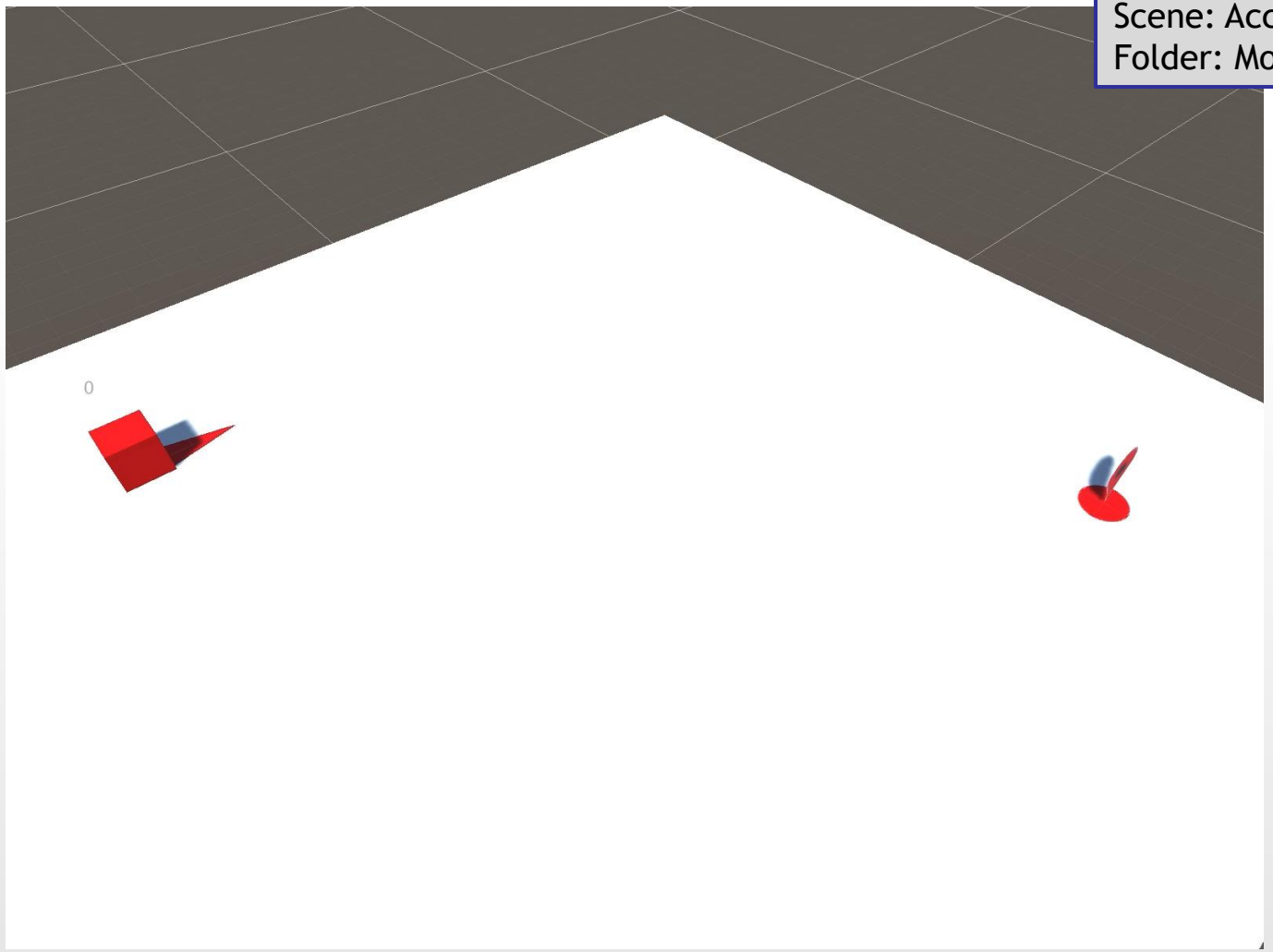
Selection between gas and brake

Signum is inside currentGas

Clamping between two values

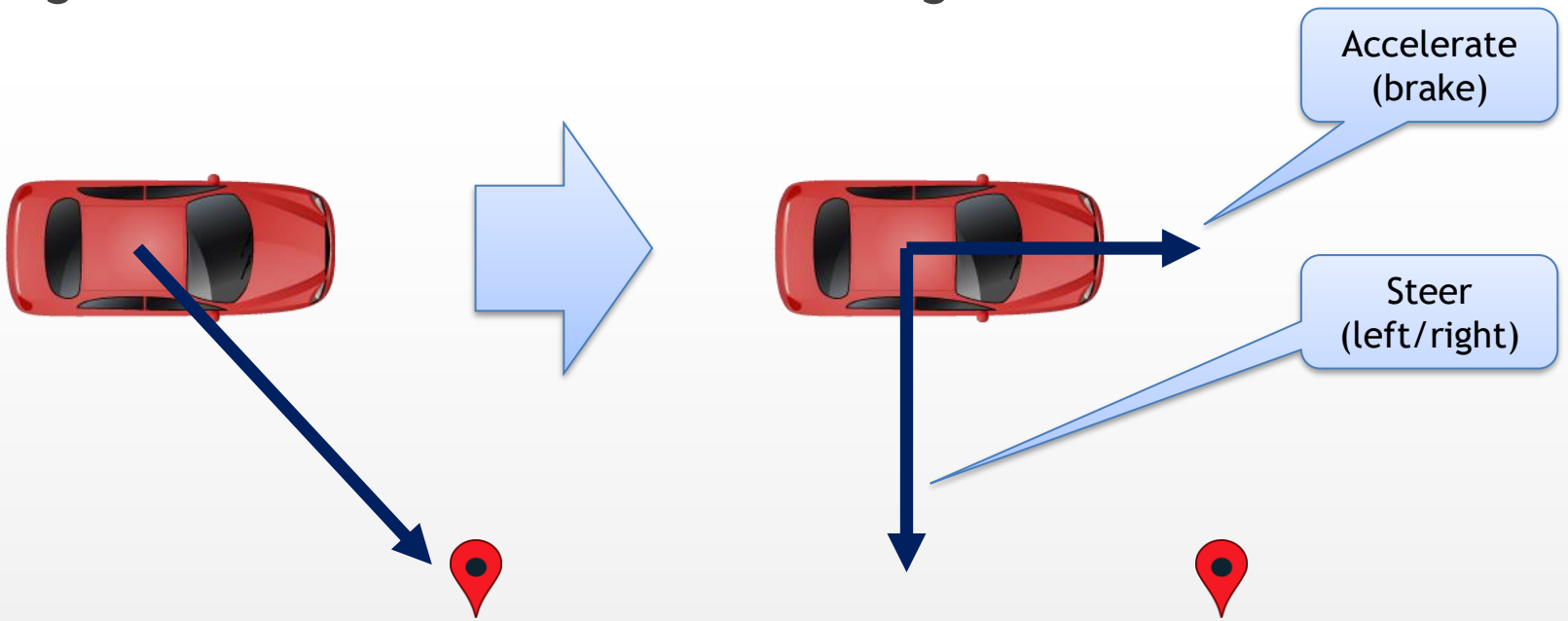
Linear Acceleration: Better But Not Perfect

Scene: Accelerate To
Folder: Movement/Dynamic



Let’s Add Steering

- We want to go in a direction … which is not “straight”



- Vector decomposition can give us a tangent (straight) component and a normal (left/right) component to tell how much to use the gas and the steering well

Steering … by the Book (the Easy Version)

Source: DLinearSteering
Folder: Movement/Dynamic

```
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) {
            Vector3 accVector = toDestination.normalized;

            Vector3 tangentComponent = Vector3.Project (accVector, transform.forward);
            Vector3 normalComponent = accVector - tangentComponent;

            float tangentSpeed = tangentComponent.magnitude * speed;
            float rotationSpeed = normalComponent.magnitude * Vector3.Dot (normalComponent, transform.right) * steer;

            float t = Time.deltaTime;

            Rigidbody rb = GetComponent<Rigidbody> ();
            rb.MovePosition (rb.position + transform.forward * tangentSpeed * t);
            rb.MoveRotation (rb.rotation * Quaternion.Euler (0f, rotationSpeed * t, 0f));
        }
    }
}
```

We are looking only to decompose the direction. Normalizing this avoids changing speed based on distance

The tangential component is the projection along “forward”

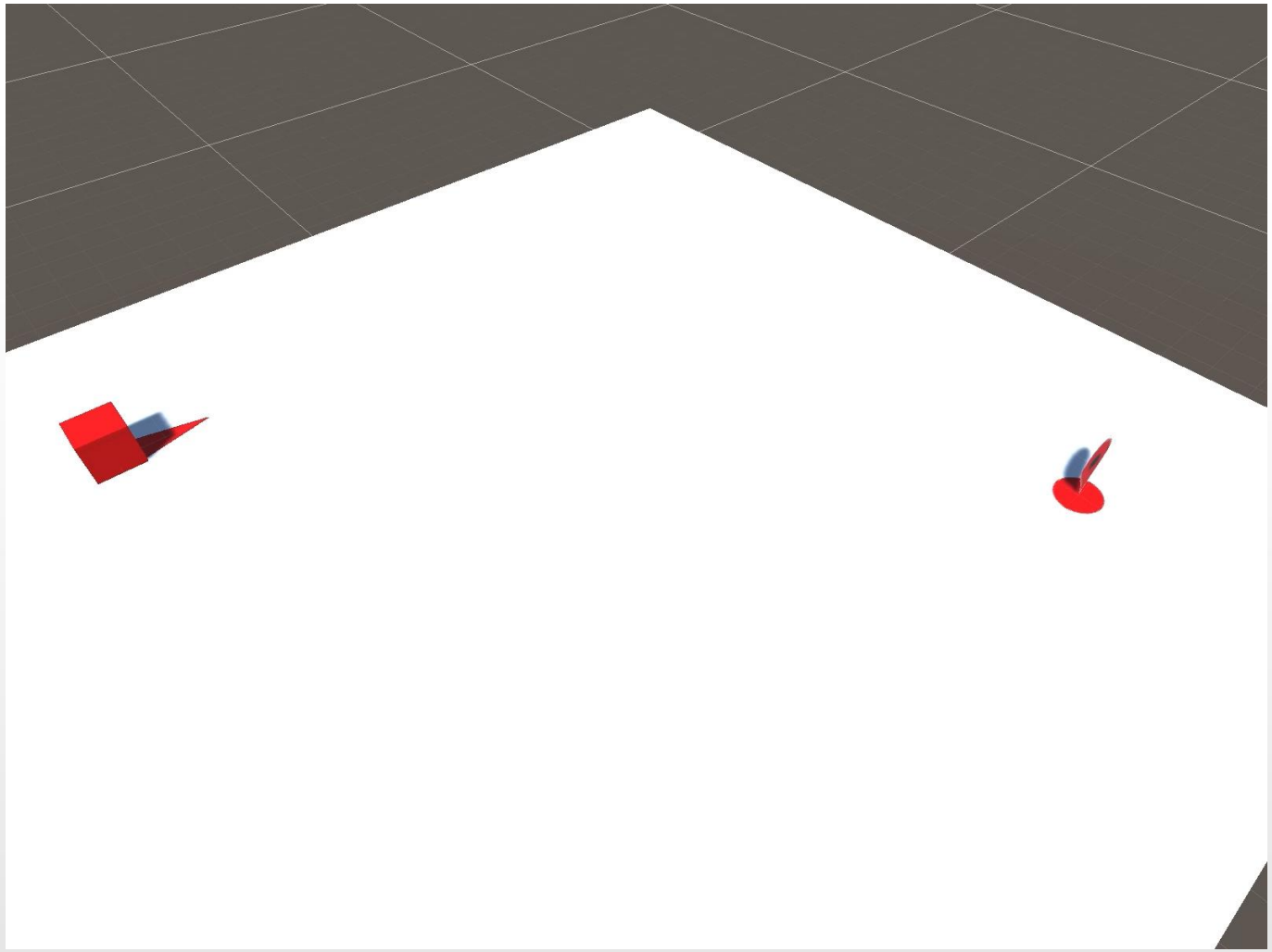
The normal component is whatever is left

This dot product will distinguish between left (-1) and right (+1)

We move the Rigidbody forward (wherever is forward) based on the tangential component and then we rotate based on the normal component.
It is possible to do the opposite as long as we are consistent with all agents

But (Unsurprisingly) …

Scene: Steer To (linear)
Folder: Movement/Dynamic



Then, We Need to Work With Acceleration

- We must consider the two components as multiplicators for linear and angular accelerations, and work with that
  - And good luck about tuning it!

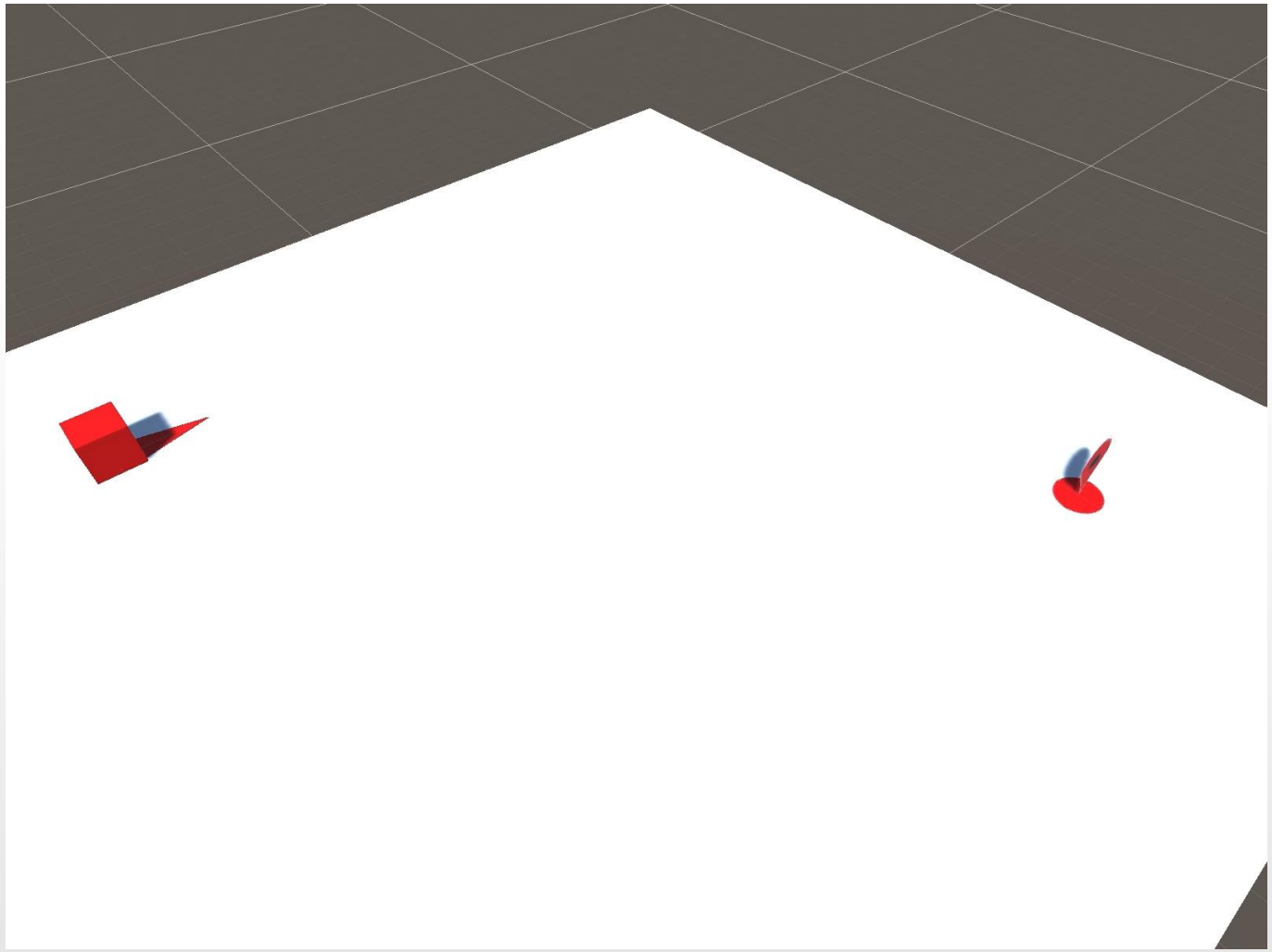
```
void FixedUpdate () {
    if (destination) {

        Vector3 verticalAdj = new Vector3 (destination.position.x, transform.position.y, destination.position.z);
        Vector3 toDestination = (verticalAdj - transform.position);

        Vector3 accVector = toDestination.normalized;

        Vector3 tangentComponent = Vector3.Project (accVector, transform.forward);
        Vector3 normalComponent = accVector - tangentComponent;

        float tangentAcc = tangentComponent.magnitude * gas;
        float rotationAcc = normalComponent.magnitude * Vector3.Dot (normalComponent, transform.right) * steer;

        float t = Time.deltaTime;

        float tangentDelta = currentLinearSpeed * t + 0.5f * tangentAcc * t * t;
        float rotationDelta = currentAngularSpeed * t + 0.5f * rotationAcc * t * t;

        currentLinearSpeed += tangentAcc * t;
        currentAngularSpeed += rotationAcc * t;

        Rigidbody rb = GetComponent<Rigidbody> ();
        rb.MovePosition (rb.position + transform.forward * tangentDelta);
        rb.MoveRotation (rb.rotation * Quaternion.Euler (0f, rotationDelta, 0f));
    }
}
```

$x = v\,t + \frac{1}{2}\,a\,t^2$

$\theta = \omega\,t + \frac{1}{2}\,\alpha\,t^2$

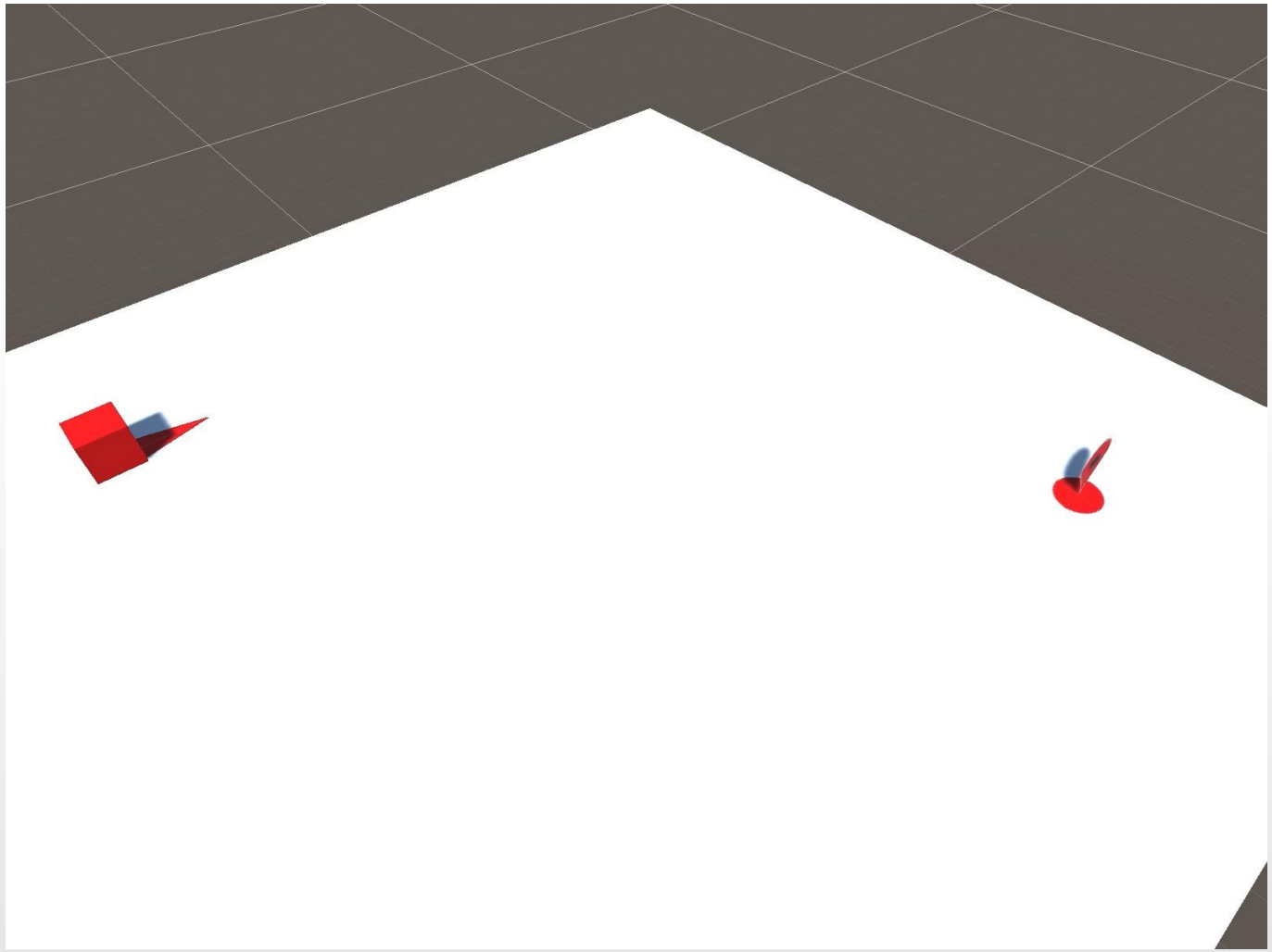
Too Fast! (gas = 3, steer = 8)

Scene: Steer To (accelerate)
Folder: Movement/Dynamic



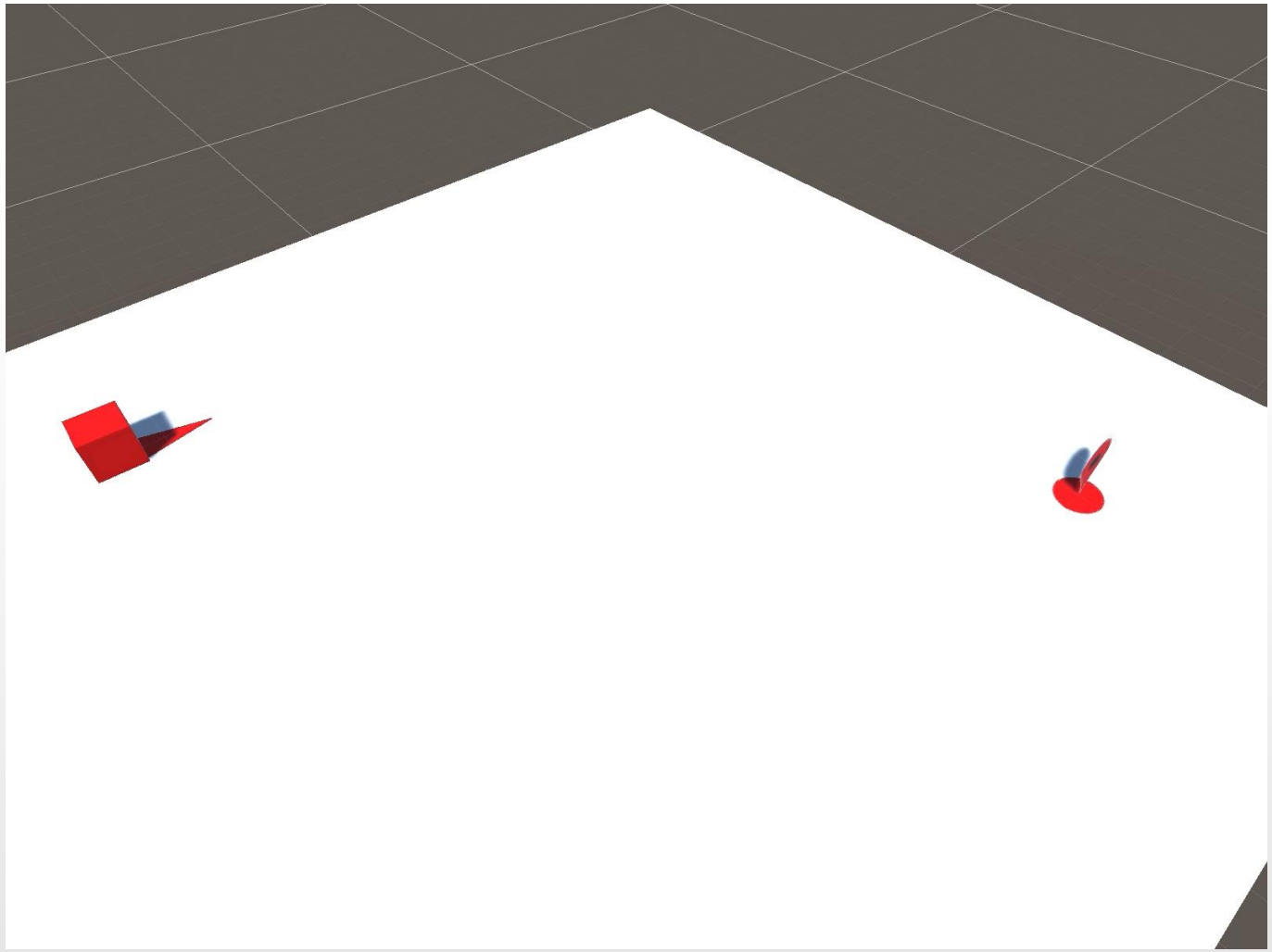
Sheer Luck (gas = 2, steer = 10)

Scene: Steer To (accelerate)
Folder: Movement/Dynamic



Drunk Driver (gas = 0.5, steer = 30)

Scene: Steer To (accelerate)
Folder: Movement/Dynamic



Why is It THAT Bad?

- Simple: Because the model is not refined enough

- You want a car, but you simulate a soap bar on the water
  - A car is not a pointy mass (while a soap bar is a pointy mass)
    - We have a pivot somewhere in the back
  - Wheels are revolving around only one axis
    - They exert an opposing force to sideway movements

- Moreover, a constant acceleration is not a natural behaviour
  - We need to introduce more dynamic movement i.e., acceleration should change over time

(Not So) Obvious Improvements

1. Use brakes

2. Clamp speed
   - Whenever reasonable

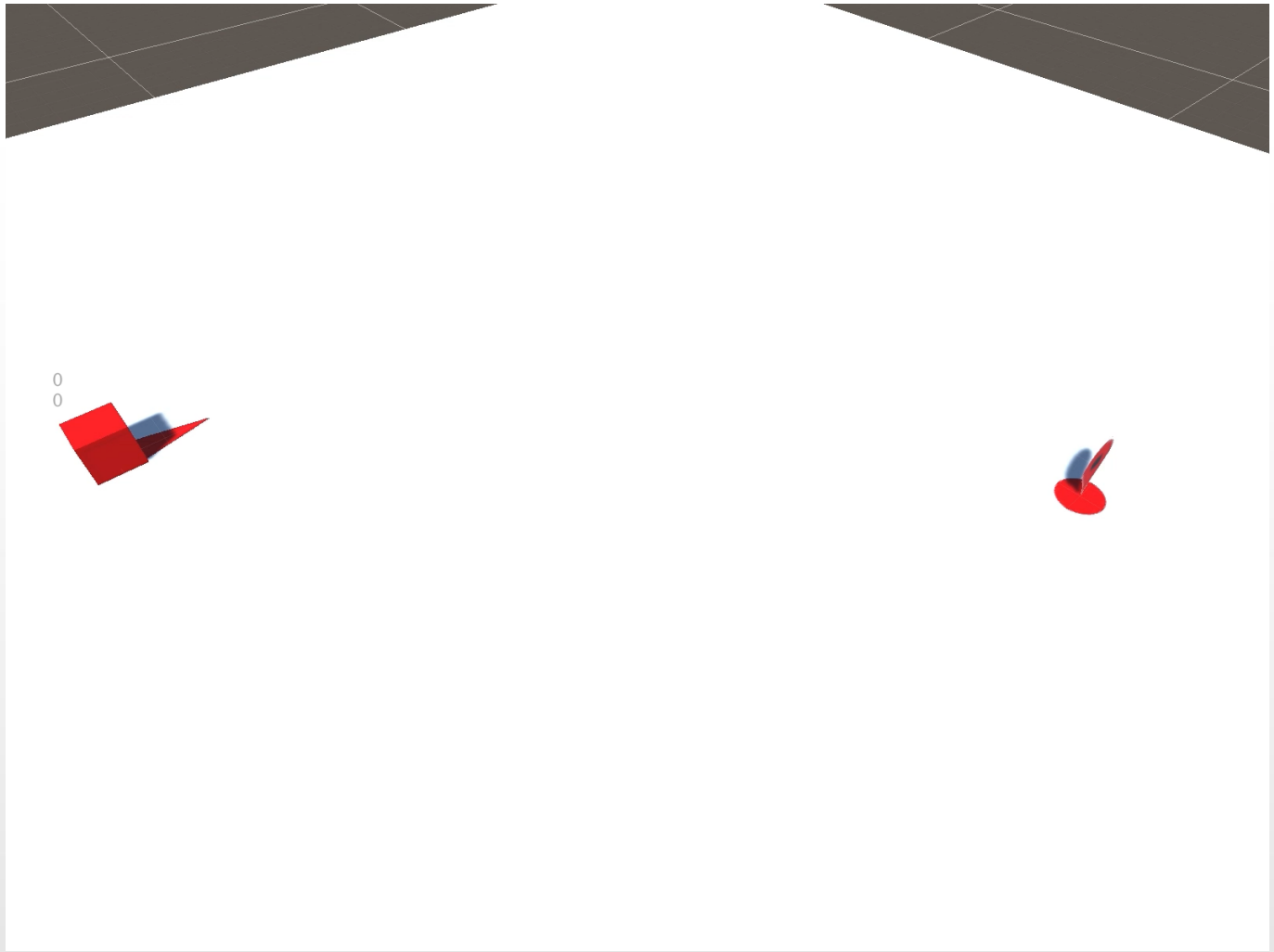
3. Add drag
   - Both linear and angular
   - You will be surprised how well it works

4. Consider higher order derivatives

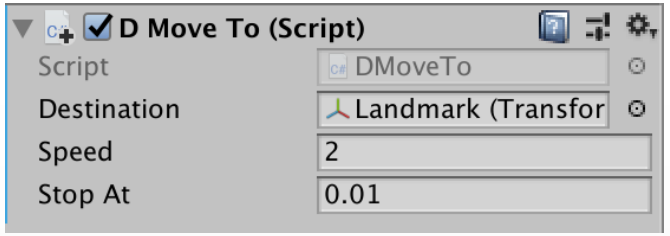
```
currentLinearSpeed = (currentLinearSpeed * (1f - linearDrag * t)) + tangentAcc * t;
currentAngularSpeed = (currentAngularSpeed * (1f - angularDrag * t)) + rotationAcc * t;

currentLinearSpeed = Mathf.Clamp (currentLinearSpeed, minLinearSpeed, maxLinearSpeed);
currentAngularSpeed = Mathf.Clamp (currentAngularSpeed, -maxAngularSpeed, maxAngularSpeed);
```

And Finally ...

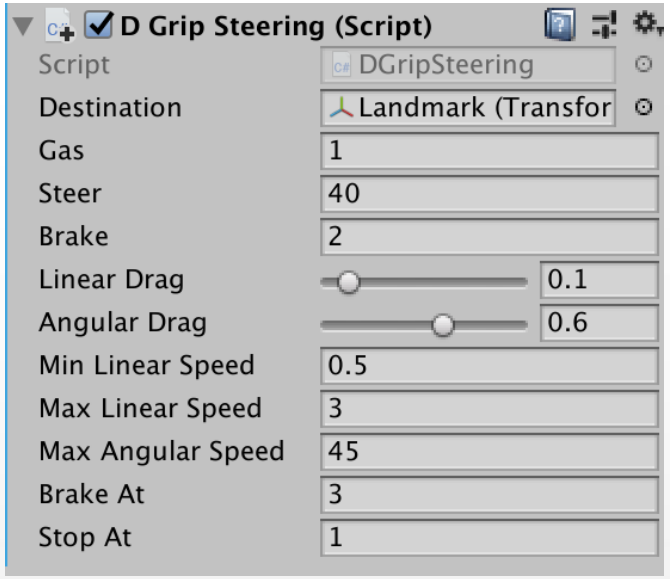
Scene: Steer To (full)
Folder: Movement/Dynamic



Evolution



From this ...



... to this

And this is still a very simplified version of many games!

Dynamic Algorithms

- Dynamic movement algorithms use kinematic data (include velocities) and output an acceleration
  - Read: they tell you where to push the agent

- A steering behavior is not designed to do everything
  - Each algorithm does a single thing and only takes the input needed to do that
  - To get more complicated behaviors, we need to combine the steering behaviors and make them work together

- Variable matching
- Seek/Flee
- Align
- Velocity matching

Variable Matching

- Simplest SB family

- Tries to match one or more elements of the character kinematics to a target kinematics
  - Position: accelleration toward target and deceleration when approaching
  - Orientation: apply rotation
  - Velocity: following target on a parallel path while copying its movement

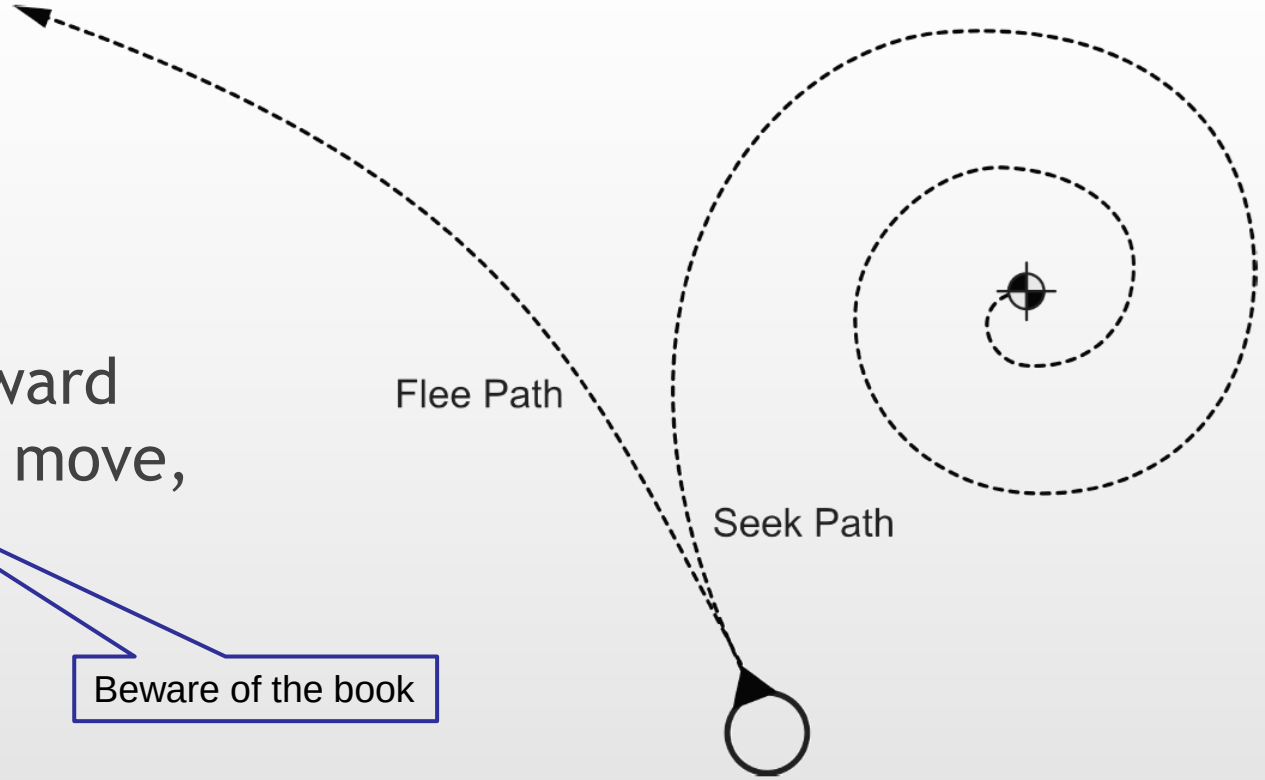
- When more than one element is matched there may be conflicts
  e.g., velocity and position

Seek (and Flee)

- SEEK matches a position:
  - Finds direction and accelerates toward target
  - Since speed cannot be infinite, we explicitly define a maximum value
  - Apply drag (if any)

- It’s good for chasing and evading

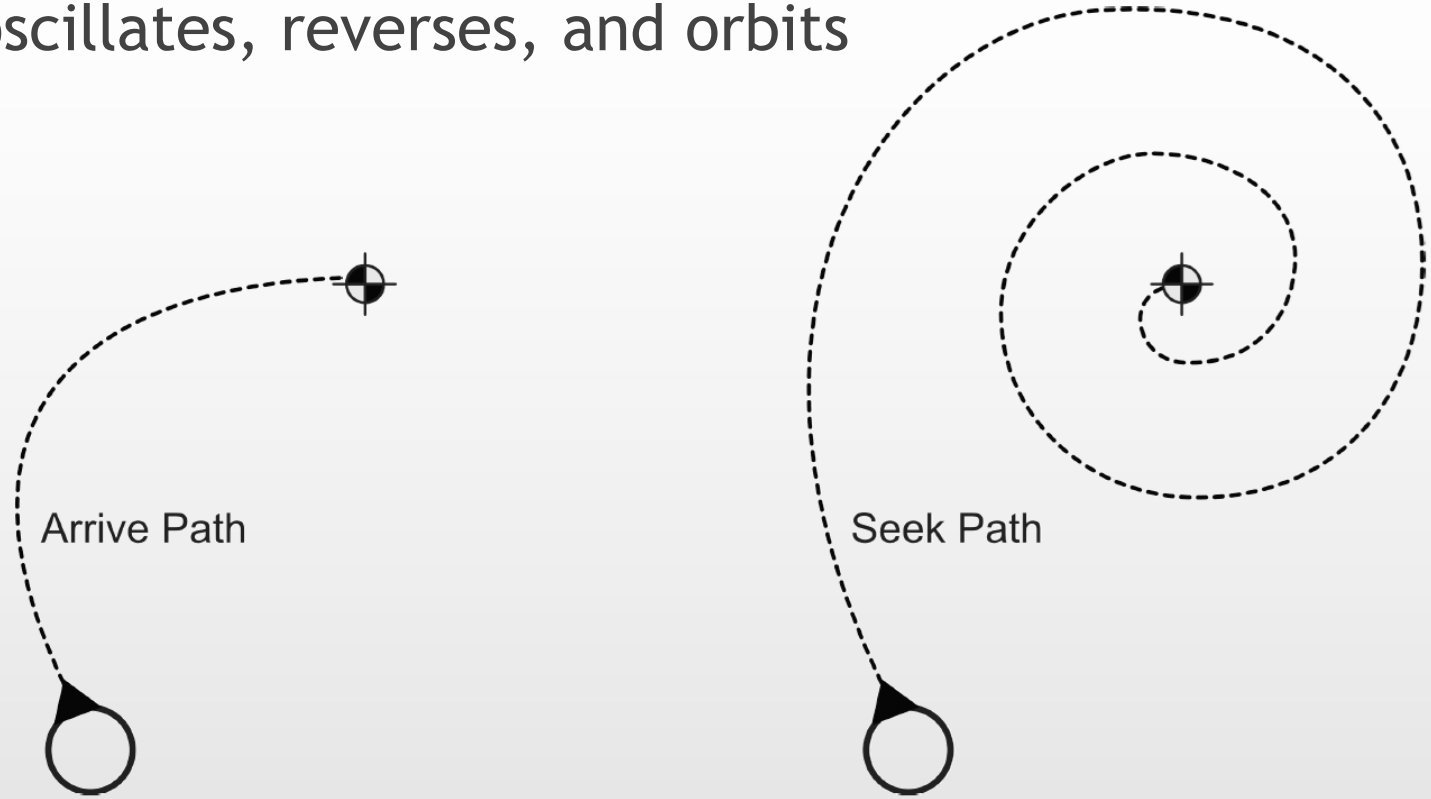
- Acceleration is always directed toward the target. If the target DOES NOT move, agent will orbit around it
  - Drag can trasform orbit into a spiral: if enough big, it goes unnoticed



Arrive

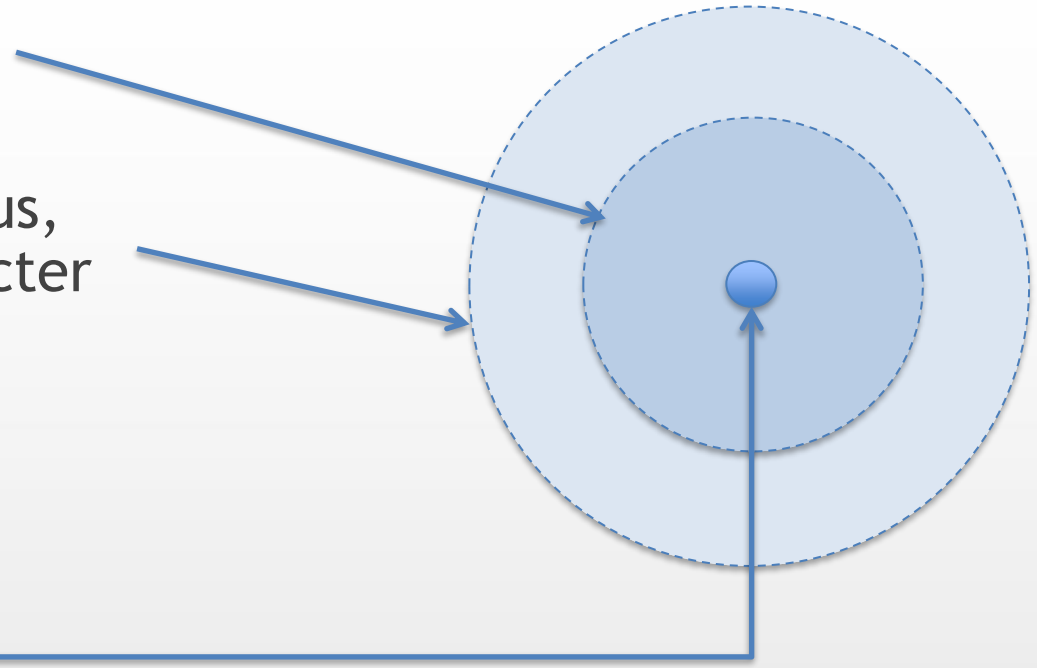
- Seek is ok for chasing when the target is moving, when it does not, we have problems
  - Character overshoots, oscillates, reverses, and orbits

- Arrive is designed for reaching a target



Arrive

- The dynamic version of arrive uses two radii:
  - Arrival radius as in the kinematic case
  - Slowing-down radius: larger than previous, when reached (at max speed) the character starts to slow down



- Velocity:
  - Max speed till slowing-down radius
  - Zero in target
  - Interpolated in between

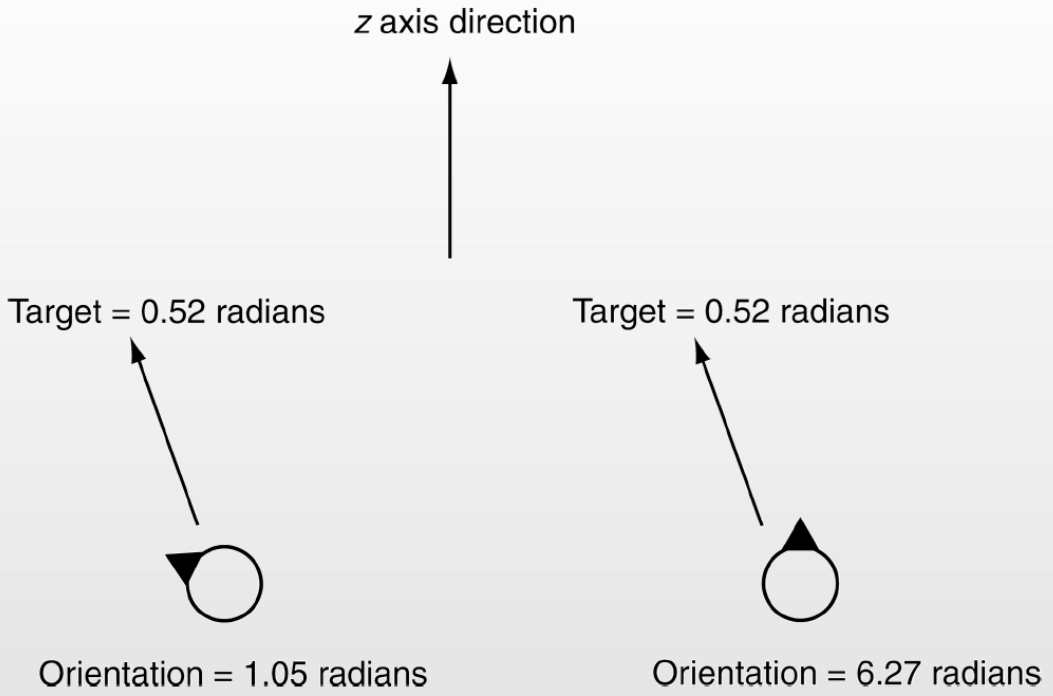
Align

- Tries to match agent orientation with that of the target
  - Position and velocity are not considered
- Output: is angular acceleration (NO linear accelleration)
- The behavior is similar to arrive
  - Tries to get target orientation and to sets there with rotation = 0

Align

- Orientation wraps around 2π rad
  - Required rotation cannot be calculated subtracting orientations
  - Calculate subtraction and then convert result in $[-\pi, \pi]$

- When approaching the target value it does like arrival
  - Two radii, which in this case, are intervals



Velocity Matching

- Tries to match the velocity of a target
- Seldom useful when alone, is critical for implementing group-based behaviors
- Can be easily derived from the arriving algorithm
  - Arrive must already match velocity of a moving target

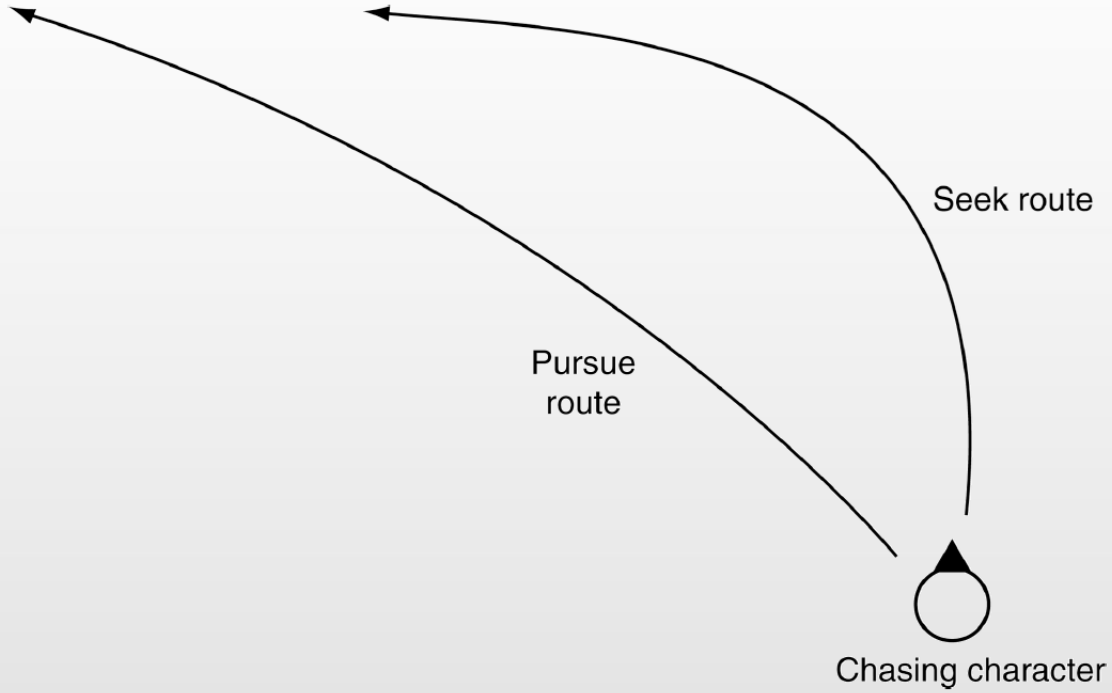
Delegated Behaviors

- Seek (flee), align, and velocity matching are the only fundamental behaviors
- All other behaviors pursue a target and then delegate to them to calculate the steering
- Complex behaviors:
  1. Pursue and evade
  2. Face
  3. Looking where you are going
  4. Wander
  5. Path following
  6. Separation
  7. Collision avoidance
  8. Obstacle and wall avoidance

Pursue and Evade

- If chasing a far moving target, seeking behaviour produces «unnatural» output
  - Moving toward the target position is not enough

- We need to predict target future position and move toward it
  - Let's assume target will continue moving with the same velocity



Dynamic Movement: 3. PURSUE and EVADE

- Algorithm:
  1. Works out current distance between agent and target
  2. Calculate the time to get there (at max speed)
     - Calculated time becomes the predictive lookahead
  3. Calculates target position after given time assuming target continues moving with current velocity
  4. New position becomes the agent target
  5. Delegates to seek to perform steering calculation

- If the target is too far away, a pre-defined time limit is used to perform calculation

Face

- It makes a character look at its target(s)

  1. Performs calculation based on target orientation

  2. Delegates to align to perform rotation

Looking Where You are Going

- We want an agent to be looking in the direction it is moving

1. Calculate target orientation using current velocity

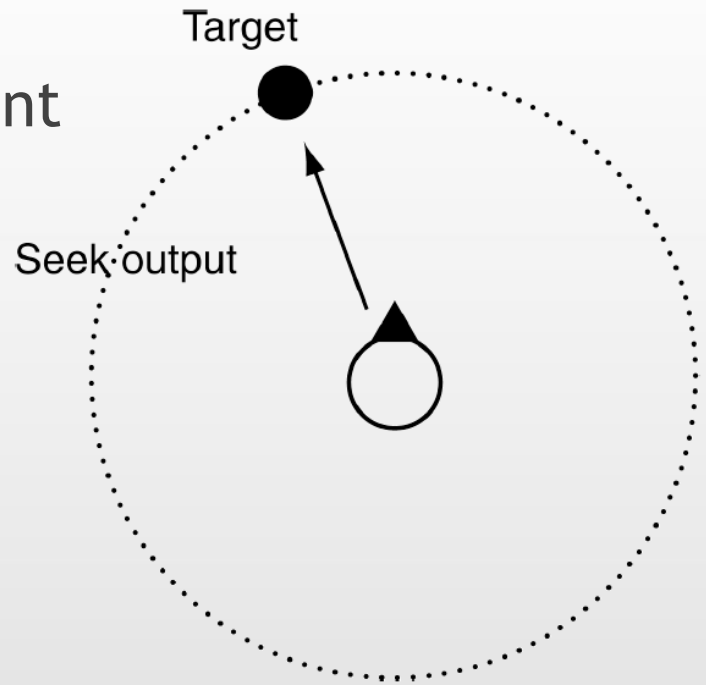
2. Delegate to align to calculate the angular acceleration
   - The agent rotates gradually

Wander

- Kinematic wandering gives linear jerkiness and controls a character moving aimlessly about

- Possible implementation
  1. Constrain a target on a circle surrounding the agent
  2. Move the target randomly at each iteration
  3. Delegate to seek

… quite nice, but can be improved



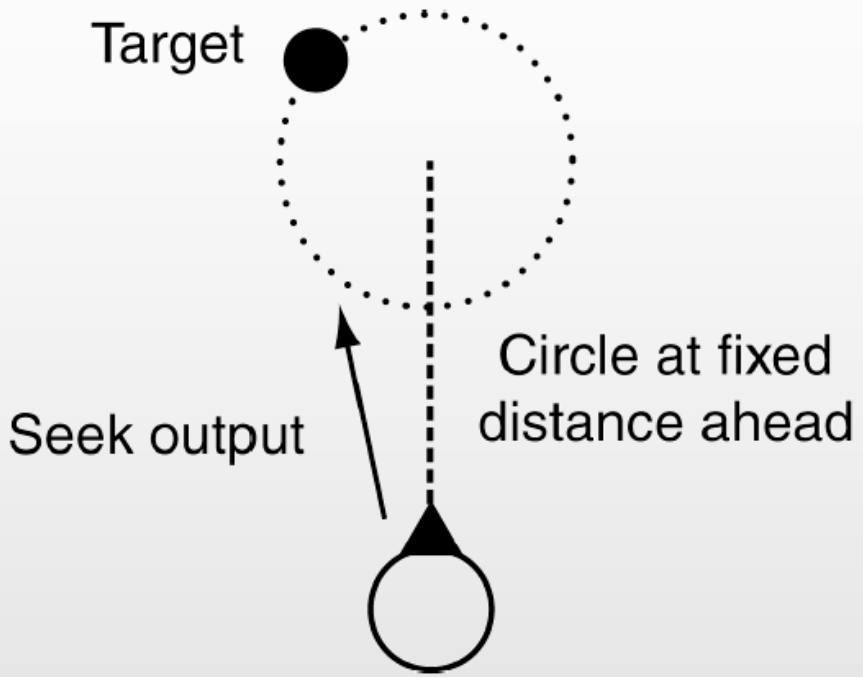
Wander

- Improved solution: shrink circle and move it in front of the character, then either:

1. Delegate to face for the orientation and then move with max acceleration

OR

2. Delegate to seek and look where you are going



Path Following

- The target is a whole path, not a single position

  1. Calculates the position of a target based on character current position and path shape
  2. Delegates to SEEK

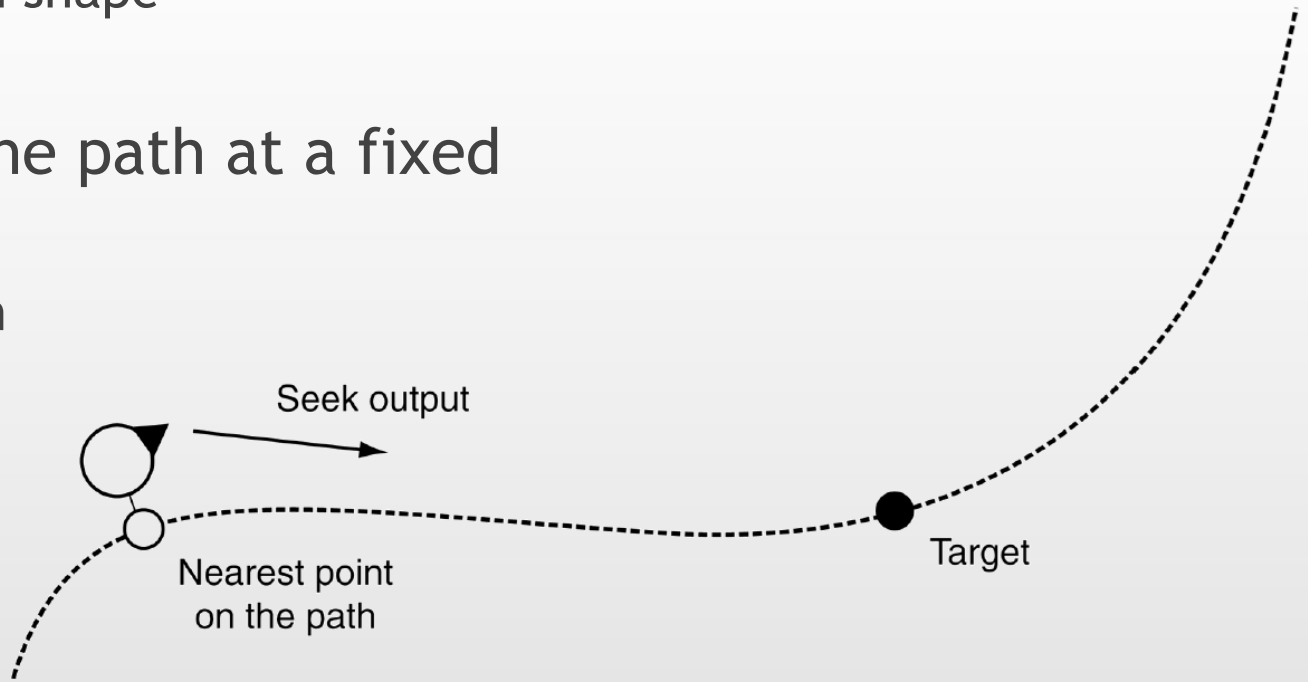
- Two variants exists:
  - Chase the rabbit
  - Predictive path following

Path Following (Chase the Rabbit)

To calculate target position:

1. Map current position to the nearest point in the path
   May be complex, it depends on path shape

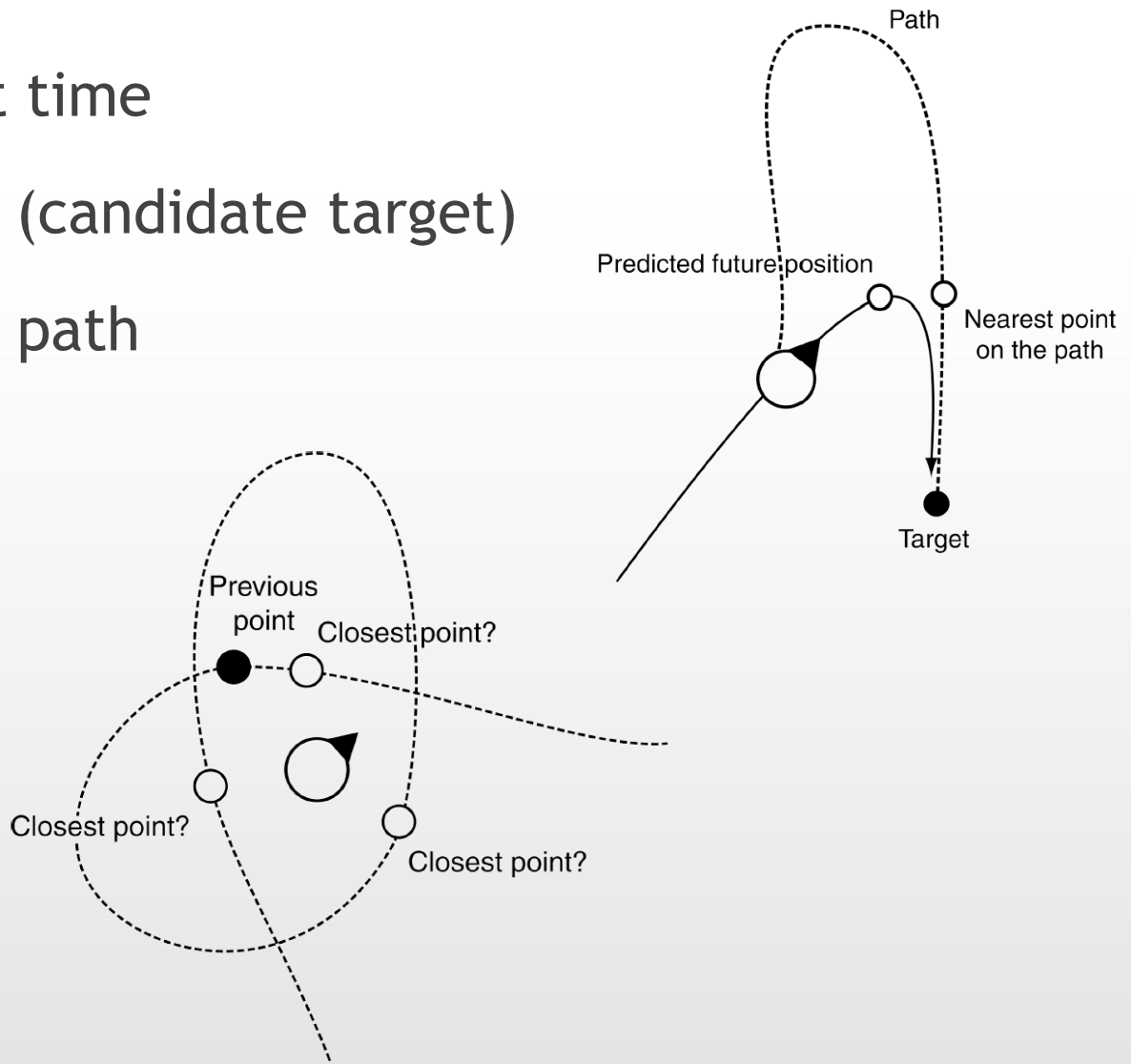
2. Select target: a point along the path at a fixed distance from character
   Distance is measured along the path



Path Following (Predictive Path Following)

1. Predict location of character in a short time
2. Map it to the nearest point in the path (candidate target)
3. Place target ahead of nearest point on path
4. Delegate to seek

The result is smoother for complex paths,
but the agent may cut corners
or lack coherence



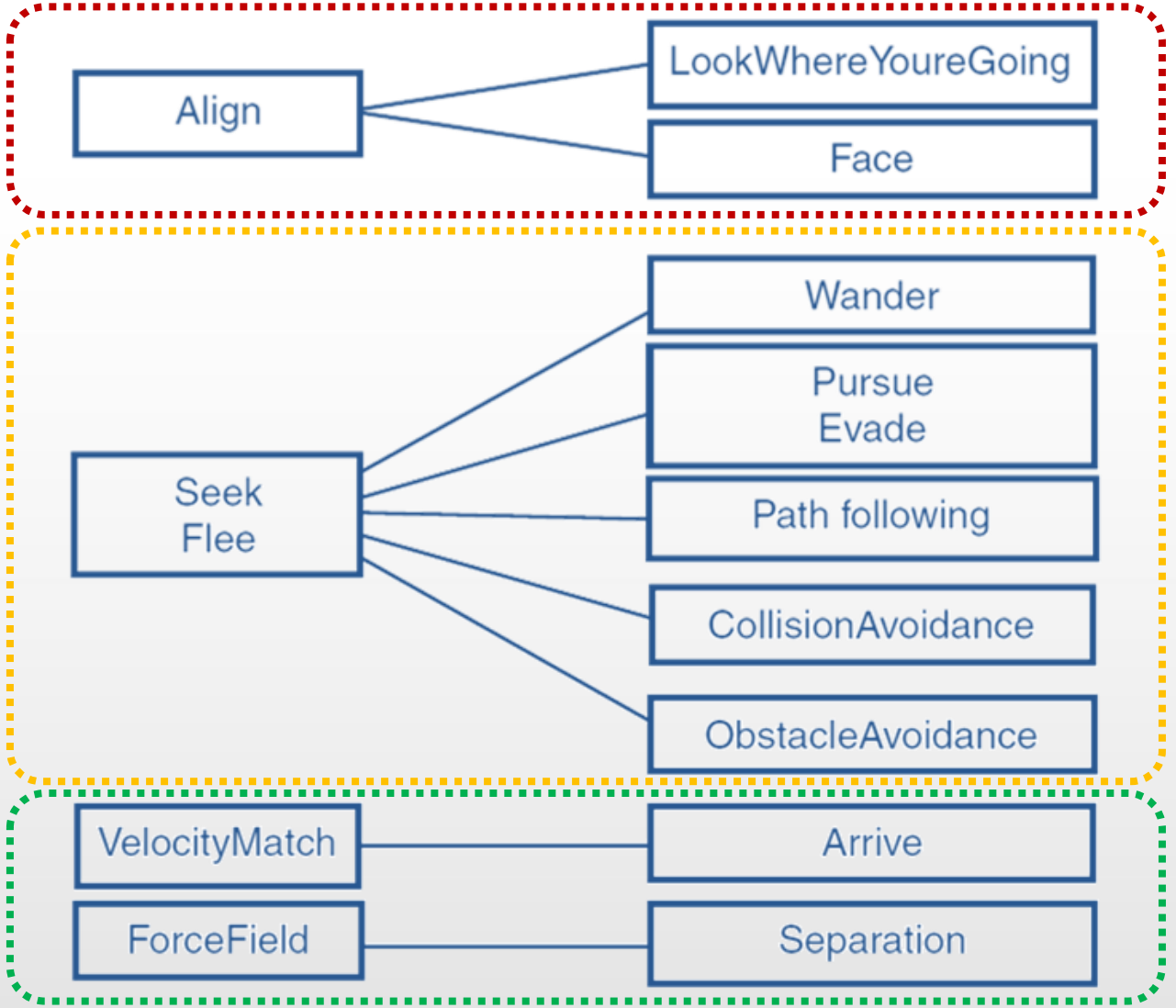
Separation (Repulsion)

- Used for crowd simulation
  avoids collisions and getting too close

- No good when characters' paths cross
  Use collision avoidance in such cases

- In the majority of cases output is null (no movement)
  if a proximity threshold reached, delegates to evade, but
  - Strength of movement depends on distance from neighbor
  - Separation strength decreases linearly or by inverse square

Summary of Delegate Behavior



References

- On the textbook
  - § 3.3 excluding 3.3.14