UNIVERSITÀ DEGLI STUDI DI MILANO

Movement

Part 3 – Composition

A.I. for Video Games

Dario Maggiorini (dario@di.unimi.it)

Composing Behavious

- Let’s say we need to implement an agent to chase the player while avoiding obstacles
- We have two behaviours: Chase and Avoid
- Based on the current state of the system, they will provide two acceleration components
- The components will be first blended, and then the result will be used to steer the agent

System status

```
void Update () {
    state.currentDestination = FindDestination (target);

    Vector3[] accelerationComponents = new Vector3[2];
    accelerationComponents[0] = Chase (state);
    accelerationComponents[1] = Avoid (state);

    Vector3 acceleration = Blend (accelerationComponents);

    state = ApplySteering (state, acceleration);
}
```

Behaviours use the current state to suggest accelerations

Components are blended in a global acceleration

The acceleration is applied to the agent. This will update the system state

Blending is Cool!

- So far, blending looks cool, but to use it correctly some refactoring is required

- We will define an architecture made of two parts
  1. A main component to collect directions, blend them together, and apply the steering
  2. Many satellite components to provide directions. Each one will be taking care of a specific behavior

- Refactoring is refactoring
  - Wi will re-organize the previous examples while keeping most of the logic

The Main Component

Source: DDelegatedSteering
Folder: Movement/Dynamic

```
public class DDelegatedSteering : MonoBehaviour {

    public float minLinearSpeed = 0.5f;
    public float maxLinearSpeed = 5f;
    public float maxAngularSpeed = 5f;

    private MovementStatus status;

    private void Start () {
        status = new MovementStatus ();
    }

    void FixedUpdate () {

        status.movementDirection = transform.forward;

        // Contact al behaviours and build a list of directions
        List<Vector3> components = new List<Vector3> ();
        foreach (MovementBehaviour mb in GetComponents<MovementBehaviour> ())
            components.Add (mb.GetAcceleration (status));

        // Blend the list to obtain a single acceleration to apply
        Vector3 blendedAcceleration = Blender.Blend (components);

        // if we have an acceleration, apply it
        if (blendedAcceleration.magnitude != 0f) {
            Driver.Steer (GetComponent<Rigidbody> (), status, blendedAcceleration,
                          minLinearSpeed, maxLinearSpeed, maxAngularSpeed);
        }
    }
```

Clamping is required after speed calculation. So, we need these parameters here

At startup we create a new status information

We set in the status the agent is moving forward

Iterate on all movement behaviours to create a list of directions

Blend the list of directions and calculate a blended acceleration to apply

Apply the blended acceleration to the Rigidbody while taking into account the status information

The Main Component

```
public class DDelegatedSteering : MonoBehaviour {

    public float minLinearSpeed = 0.5f;
    public float maxLinearSpeed = 5f;
    public float maxAngularSpeed = 5f;

    private MovementStatus status;

    private void Start () {
        status = new MovementStatus ();
    }

    void FixedUpdate () {

        status.movementDirection = transform.forward;

        // Contact al behaviours and build a list of directions
        List<Vector3> components = new List<Vector3> ();
        foreach (MovementBehaviour mb in GetComponents<MovementBehaviour> ())
            components.Add (mb.GetAcceleration (status));

        // Blend the list to obtain a single acceleration to apply
        Vector3 blendedAcceleration = Blender.Blend (components);

        // if we have an acceleration, apply it
        if (blendedAcceleration.magnitude != 0f) {
            Driver.Steer (GetComponent<Rigidbody> (), status, blendedAcceleration,
                          minLinearSpeed, maxLinearSpeed, maxAngularSpeed);
        }
    }
```

All service functions can be defined in an external file (or library)

Support to the Main Component

Source: DelegationUtilities
Folder: Movement/Dynamic/Delegates

```
public class MovementStatus {
    public Vector3 movementDirection;
    public float linearSpeed;
    public float angularSpeed;
}

// To be extended by all movement behaviours
public abstract class MovementBehaviour : MonoBehaviour {
    public abstract Vector3 GetAcceleration (MovementStatus status);
}

public class Blender {
    public static Vector3 Blend (List<Vector3> vl) {
        Vector3 result = Vector3.zero;
        foreach (Vector3 v in vl) result += v;
        return result;
    }
}

// The steer function is the same as the FixedUpdate of DGripSteering
public class Driver {
    public static void Steer (Rigidbody body, MovementStatus status, Vector3 acceleration,
                              float minV, float maxV, float maxSigma) {

        Vector3 tangentComponent = Vector3.Project (acceleration, status.movementDirection);
        Vector3 normalComponent = acceleration - tangentComponent;
```

The MovementStatus class holds information about the current system configuration: speeds and movement direction

MovementBehaviour is an abstract class: a template for all satellite components
It is not possible to achieve the same result using an interface, because an interface cannot extend MonoBehaviour

For blending, in this example we use the simplest method available: the sum of all components

The rest of the code is exactly the same as the last example of the previous module (DGripSteering.cs)

Satellite Components

- Each satellite component is specialized in a (possibly simple) behavior like seek, escape, or align
- Each satellite will accept its own working parameters and configuration
- The output of a satellite component is provided by the method GetAcceleration imposed by the superclass
  - The return value is a 3D vector
- Multiple instances of the same satellite component can be present in the gameobject, depending on the global behaviour
  - Think, as an example, if we must avoid two kinds of obstacles: A and B.
    It is much easier for us to implement a behavior “avoid obstacle of type X” and use two copies; one running with parameter A, and one with parameter B

Seeking Component Example

Source: SeekBehaviour
Folder: Movement/Dynamic/Delegates

```
public Transform destination;

public float gas = 3f;
public float steer = 30f;
public float brake = 20f;

public float brakeAt = 5f;
public float stopAt = 0.01f;

public override Vector3 GetAcceleration (MovementStatus status) {
    if (destination != null) {
        Vector3 verticalAdj = new Vector3 (destination.position.x, transform.position.y, destination.position.z);
        Vector3 toDestination = (verticalAdj - transform.position);

        if (toDestination.magnitude > stopAt) {
            Vector3 tangentComponent = Vector3.Project (toDestination.normalized, status.movementDirection);
            Vector3 normalComponent = (toDestination.normalized - tangentComponent);
            return (tangentComponent * (toDestination.magnitude > brakeAt ? gas : -brake)) + (normalComponent * steer);
        } else {
            return Vector3.zero;
        }
    } else {
        return Vector3.zero;
    }
}
```

This is our usual tangent plus normal vectors decomposition and rescaling based on gas/brake and steer (see the linear steering example).

The only difference is that we combine them together to return a single vector to the caller

In the caller (the main component), the vector will be decomposed again (after blending) to devise linear and angular accelerations

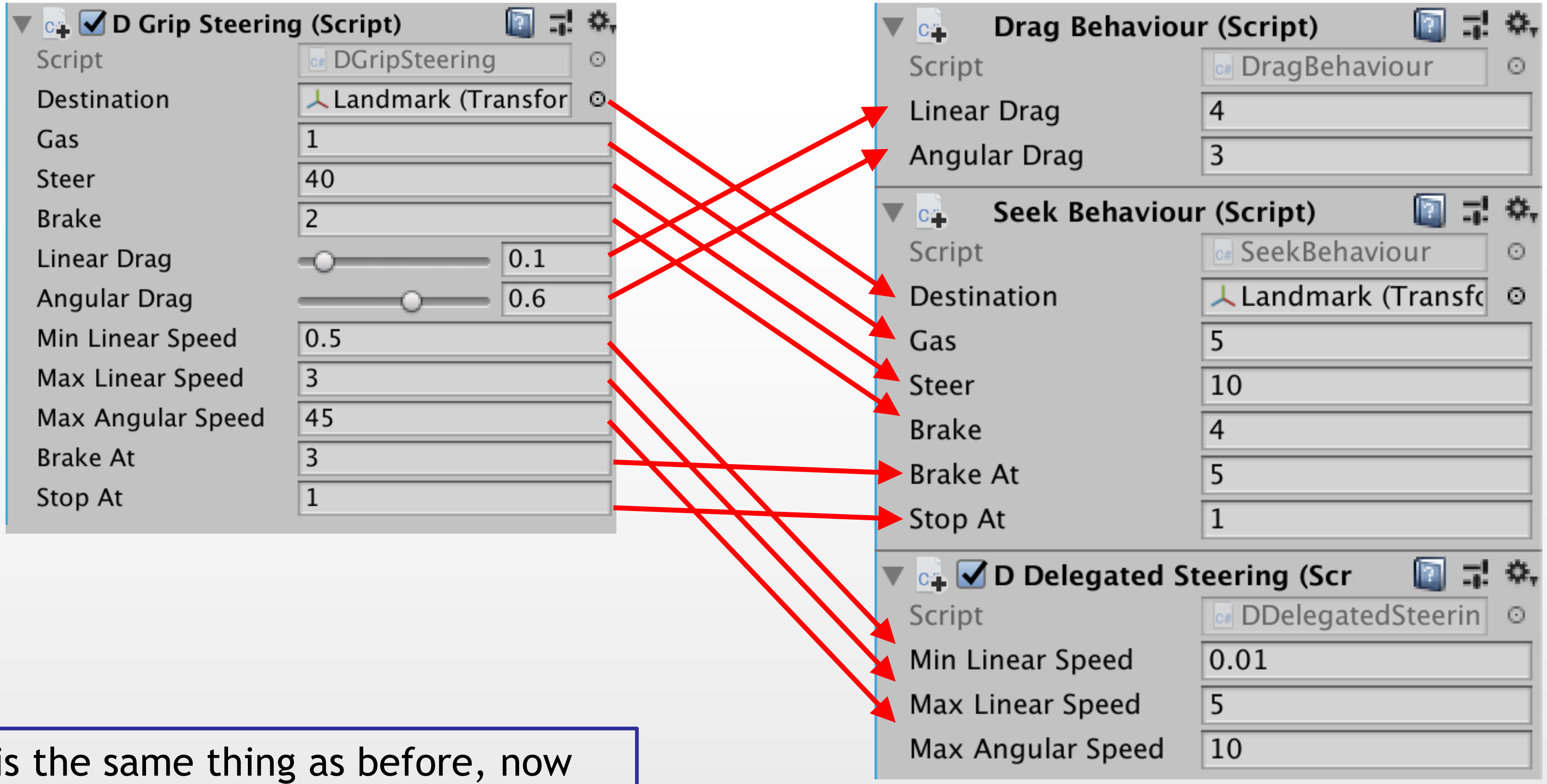
Drag Component Example

Source: DragBehaviour
Folder: Movement/Dynamic/Delegates

Change in semantic for braking.
We do not have the current acceleration; so. we apply a force in the opposite direction of the movement proportional to the current speed.

```
public class DragBehaviour : MovementBehaviour {

    // how long does it take to stop moving by dragging
    public float linearDrag = 3f;
    public float angularDrag = 3f;

    public override Vector3 GetAcceleration (MovementStatus status) {
        return - (status.movementDirection.normalized * status.linearSpeed / linearDrag)
               - ((Quaternion.Euler (0f, 90f, 0f) * status.movementDirection.normalized) * status.angularSpeed / angularDrag);
    }
}
```

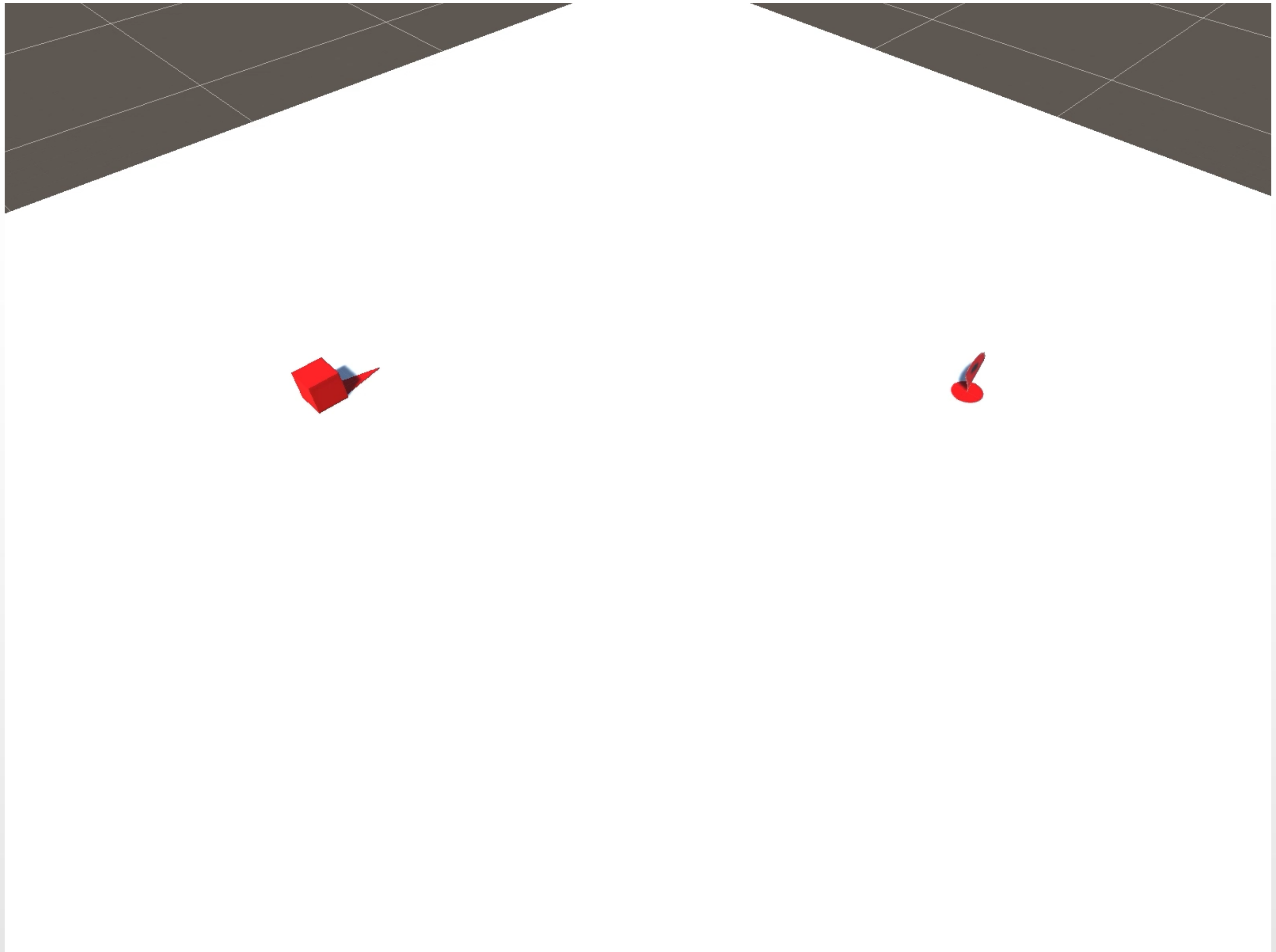
Another Evolution



It is the same thing as before, now distributed over three components

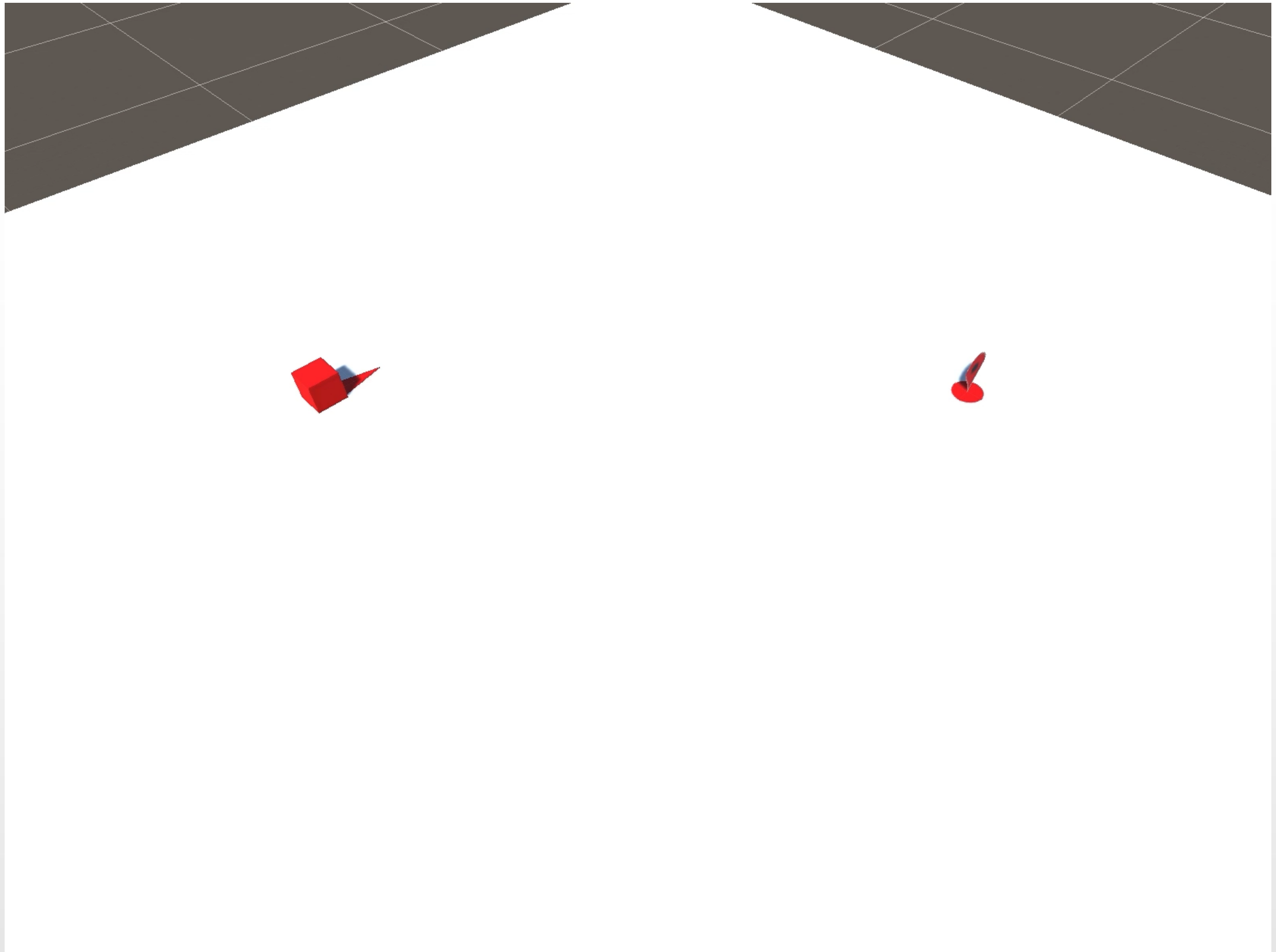
The Behaviour is Predictable

Scene: Delegation
Folder: Movement/Dynamic



... Also With a Moving Target

Scene: Delegation
Folder: Movement/Dynamic



To enable the movement of the landmark you must check the “active” flag for the Rotate script in the pivot

Please Mind

- The proposed blending framework is:

1. Sub-optimal (but code is re-usable)
   - Code is not performance-oriented

2. Not really refined
   - Blending should be weighted
   - Model should be more complex

3. Specific to something resembling a car
   - The movement model (and the blending method) should offer more approaches to select from

And Then … Here it Comes an Obstacle

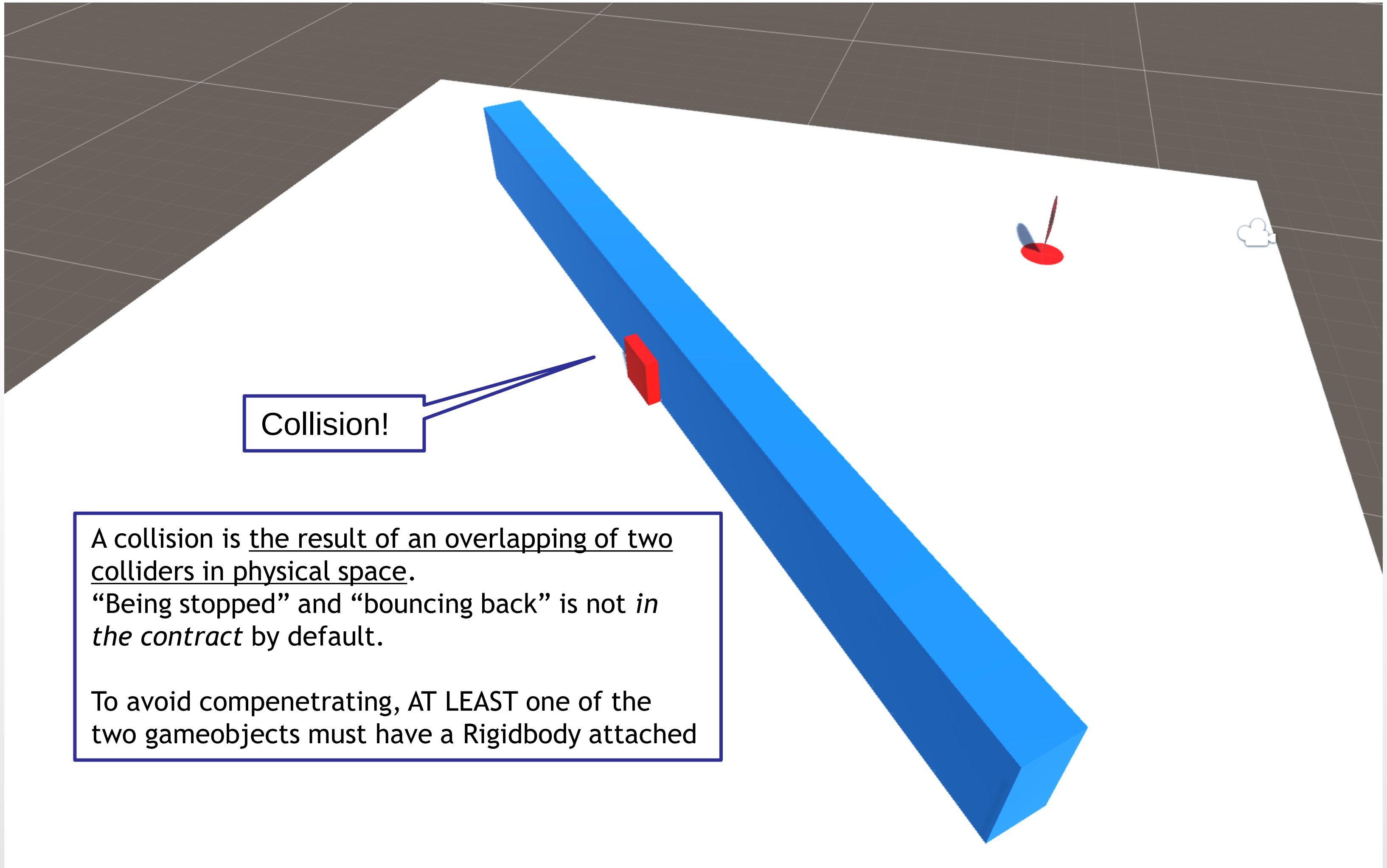
- Anything can get in your trajectory

- Then, we have two problems:

1. To correctly simulate a “bump”
   - It is a physics-related problem
   - There are issues about speed and geometry
   - We must be careful with kinematic flags combinations

2. To avoid the obstacle
   - An algorithm-related problem

Remember: Collisions are "Just” Triggers!



A collision is <u>the result of an overlapping of two colliders in physical space</u>.
“Being stopped” and “bouncing back” is not *in the contract* by default.

To avoid compenetrating, AT LEAST one of the two gameobjects must have a Rigidbody attached

Speed is Bad for Collisions

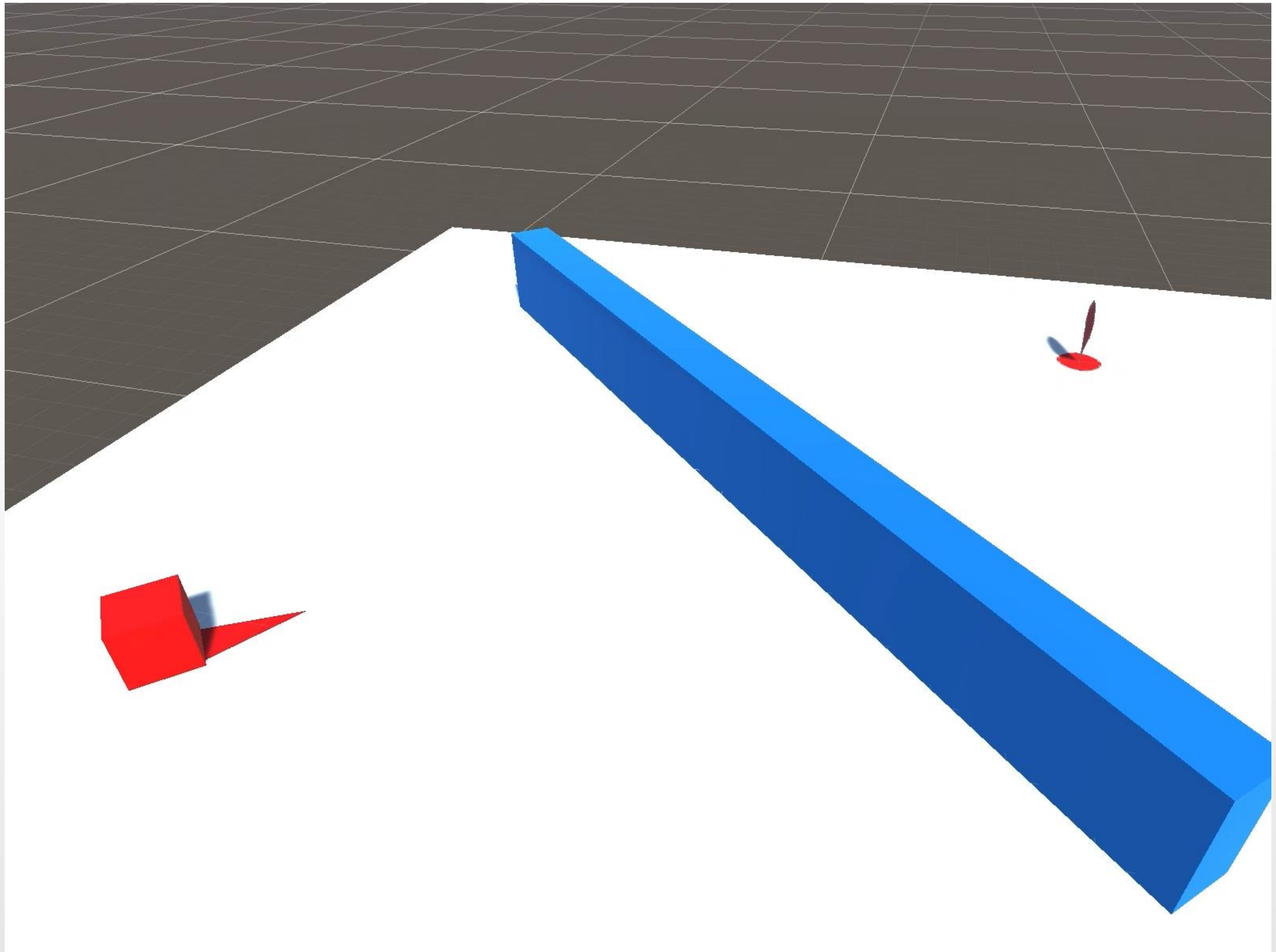
- Moving gameobjects with a Rigidbody is like teleporting them over a small distance at every update
  - The illusion of movement is given by the fact that each position is very close to the previous one

- If a gameobject is moving very fast, it could just teleport past of an obstacle
  - Or inside the obstacle, making the physics subsystem behave in an unpredictable way

- For blazing-fast objects, it is much better to apply forces or set a speed, and let the physics work

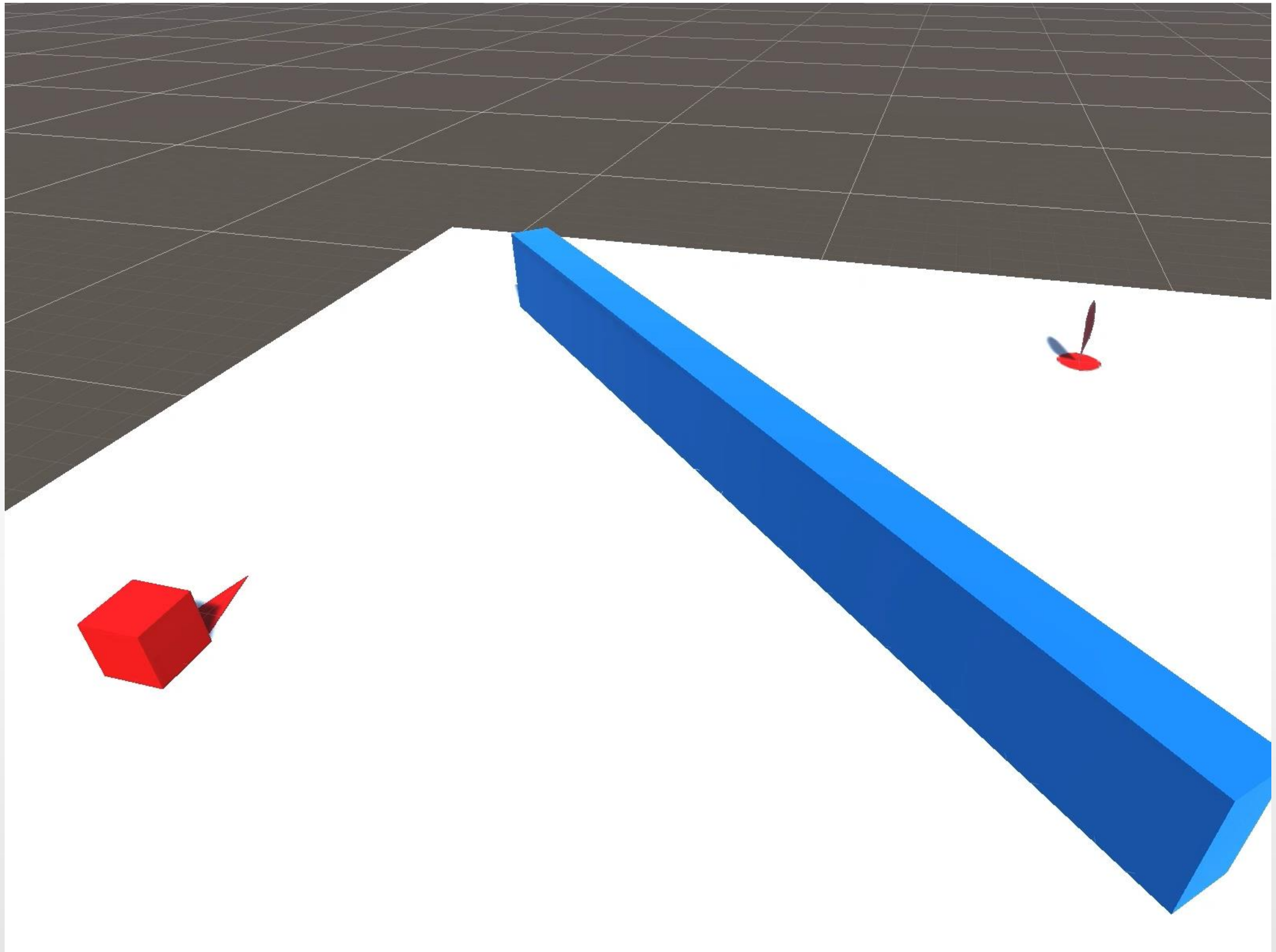
Moving With Speed Set to 2

Scene: Bump
Folder: Movement/Dynamic



Moving With Speed Set to 200

Scene: Bump
Folder: Movement/Dynamic



Detecting and Avoiding Obstacles

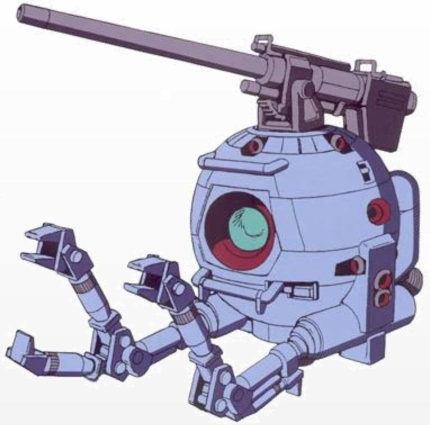
- Physics.Raycast() is your best friend here
- Beware! raycasting from trasform.position may detect colliders of other objects attached to you
  - An easy way to avoid this is to use LayerMasks

```
private IEnumerator GoChasing() {
    while (true) {
        Vector3 ray = destination.position - transform.position;
        RaycastHit hit;
        if (Physics.Raycast (transform.position, ray, out hit)) {
            if (hit.transform == destination) {
                GetComponent<NavMeshAgent> ().destination = destination.position;
            }
        }
        yield return new WaitForSeconds (resampleTime);
    }
}
```

This is an old example from the navigation meshes lecture

Using Raycasting in Unity

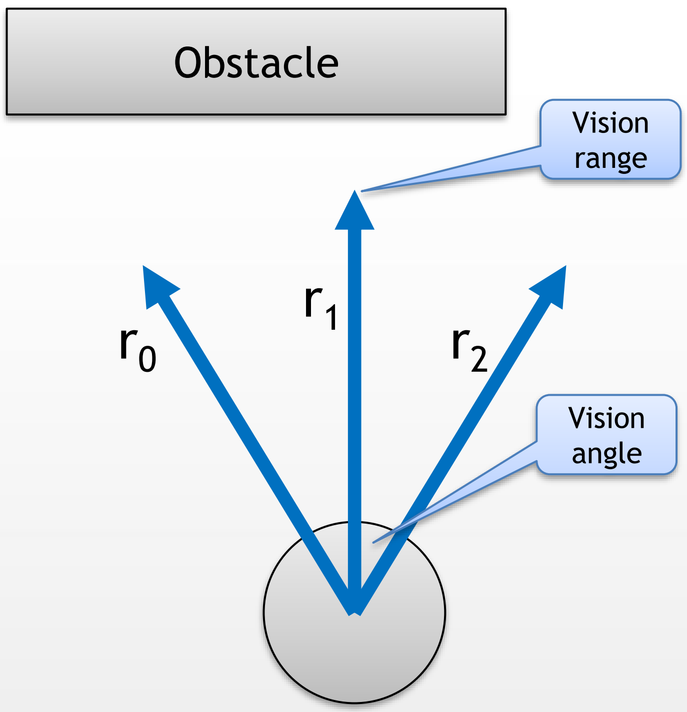
- Actually … there are many of them available
  - You project “something” forward and check if it bumps into an obstacle



- RayCast
  - The plain old “line of sight” approach
- LineCast
  - Is there anything from point A to point B?
- BoxCast
  - Use your box collider and check if you fit
- SphereCast
  - If you are moving a ball or something round
- CapsuleCast
  - For a more human-like collider

Easy Obstacle Avoidance

1. Cast three rays
   - r_0, r_1, r_2
2. Analyze hit results
3. Take a decision



r_0 hit	r_1 hit	r_2 hit	decision
no	no	no	Keep going
yes	yes	yes	Brake and backpedal
yes	no	no	Right turn
yes	yes	no	Sharp right turn

Of course, this table is NOT exhaustive

How to Implement Obstacle Avoidance

- The easy way: just add a new behaviour to blend
  - The new satellite component will give directions to steer away from obstacles

Obstacle Avoidance Component

Source: AvoidBehaviour
Folder: Movement/Dynamic/Delegates

```
public class AvoidBehaviour : MovementBehaviour {

    public float sightRange = 5f;
    public float sightAngle = 45f;

    public float steer = 15f;
    public float backpedal = 10f;

    public override Vector3 GetAcceleration (MovementStatus status) {

        bool leftHit = Physics.Raycast (transform.position, Quaternion.Euler (0f, -sightAngle, 0f) * status.movementDirection, sightRange);
        bool centerHit = Physics.Raycast (transform.position, status.movementDirection, sightRange);
        bool rightHit = Physics.Raycast (transform.position, Quaternion.Euler (0f, sightAngle, 0f) * status.movementDirection, sightRange);

        Vector3 right = Quaternion.Euler (0f, 90f, 0f) * status.movementDirection.normalized;

        if (leftHit && !centerHit && !rightHit) {
            return right * steer;
        } else if (leftHit && centerHit && !rightHit) {
            return right * steer * 2f;
        } else if (leftHit && centerHit && rightHit) {
            return -status.movementDirection.normalized * backpedal;
        } else if (!leftHit && centerHit && rightHit) {
            return -right * steer * 2f;
        } else if (!leftHit && !centerHit && rightHit) {
            return -right * steer;
        } else if (!leftHit && centerHit && !rightHit) {
            return right * steer;
        }

        return Vector3.zero;
    }
}
```

Look left and right by sightAngle degrees

Sharp turn.
Double the steering

For all other combinations, let the other satellite components decide

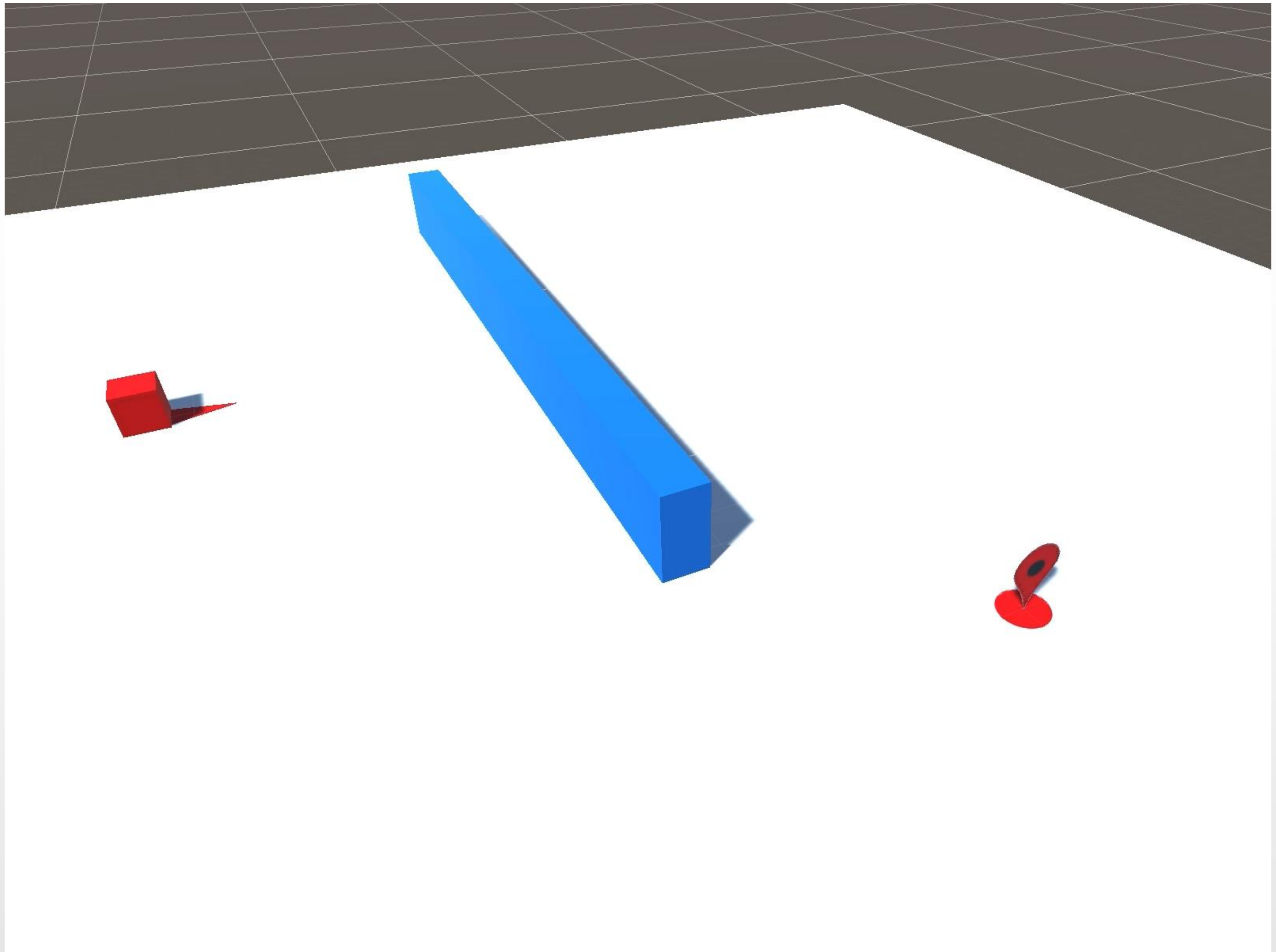
How to Implement Obstacle Avoidance

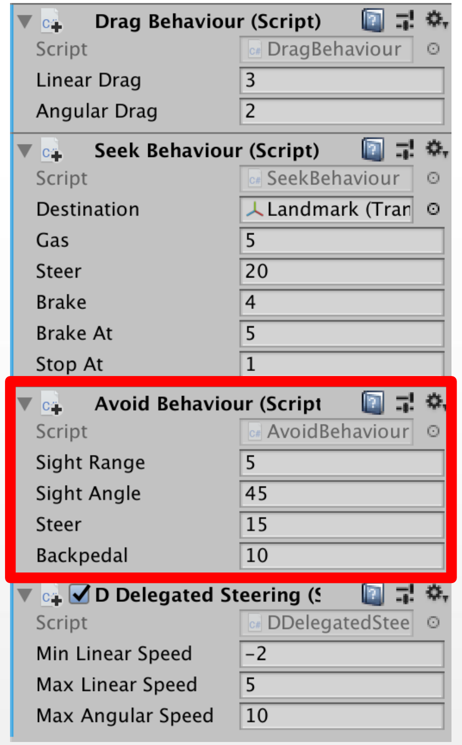
- The easy way: just add a new behaviour to blend in
  - The new satellite component will give directions to steer away from obstacles

- But this is creating us a problem!
  To make backpedaling possible, we must set the minimum linear speed to a negative value
  - When reaching the target, all three RayCast will score a hit and we will start to backpedal
  - This problem can be easily fixed by using LayerMasks, and ignoring the target
  - Another option is to strip the collider from the target. This might not be always possible, depending on your game mechanics

Obstacle Avoided

Scene: Obstacle Avoidance
Folder: Movement/Dynamic





All we did in this scene is add this component to the agent

Internal Knowledge Does Matter

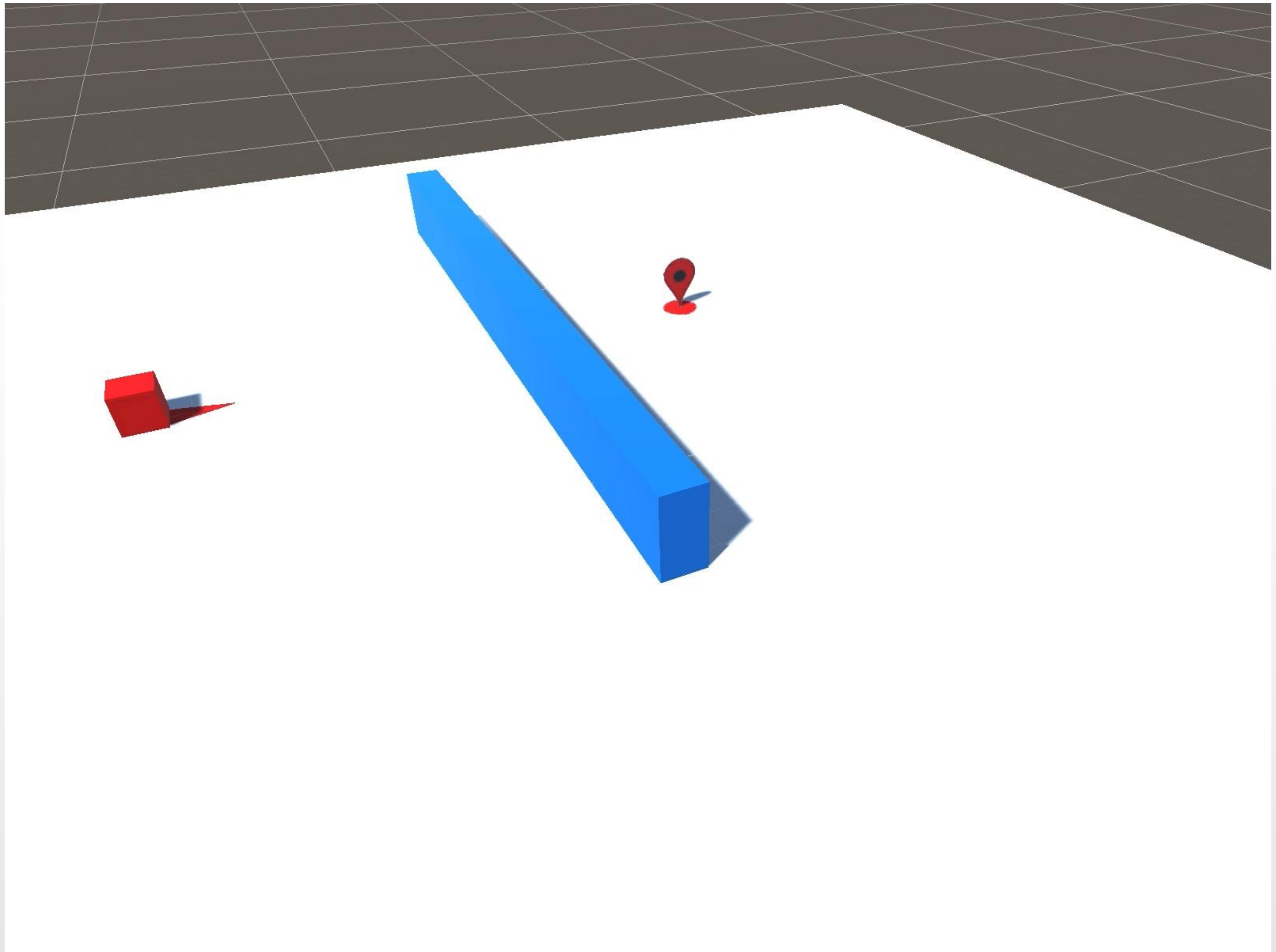
- We already discussed about ”what your agent knows” and “what your agent pretends to know to take a decision”

- In our example we are not considering this, and it is an error
  - We always know where the target is located, and go toward it

- If the landmark is located well behind the obstacle, the obstacle avoidance component might not be able to push the agent far enough to go around the wall
  - The right way to do this, should be to seek for the target only when in line of sight or, at least, set a reasonable rally point (e.g., using a point of visibility technique)

Obstacle Not Avoided

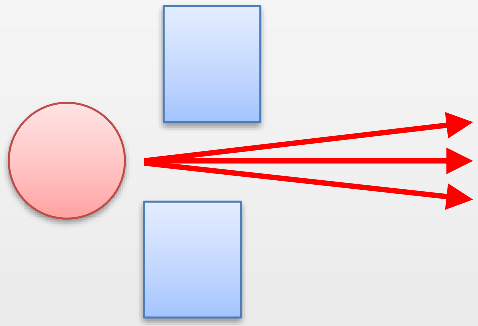
Scene: Obstacle Avoidance
Folder: Movement/Dynamic



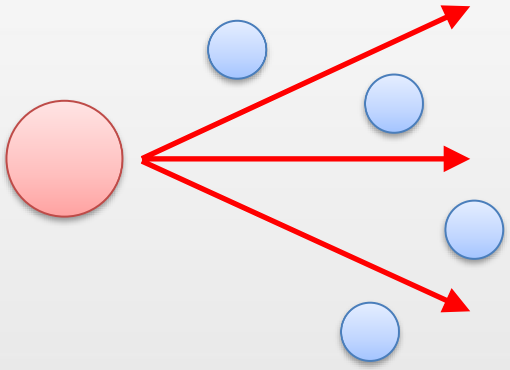
Even More Problems

- The solution to obstacle avoidance depends on the level layout

- It might be possible for a given pattern of obstacles to get undetected
  - This is especially true when using RayCast
  - Changing the vision angle will just switch the problem to another pattern
  - Using BoxCast or SphereCast is always a preferable solution



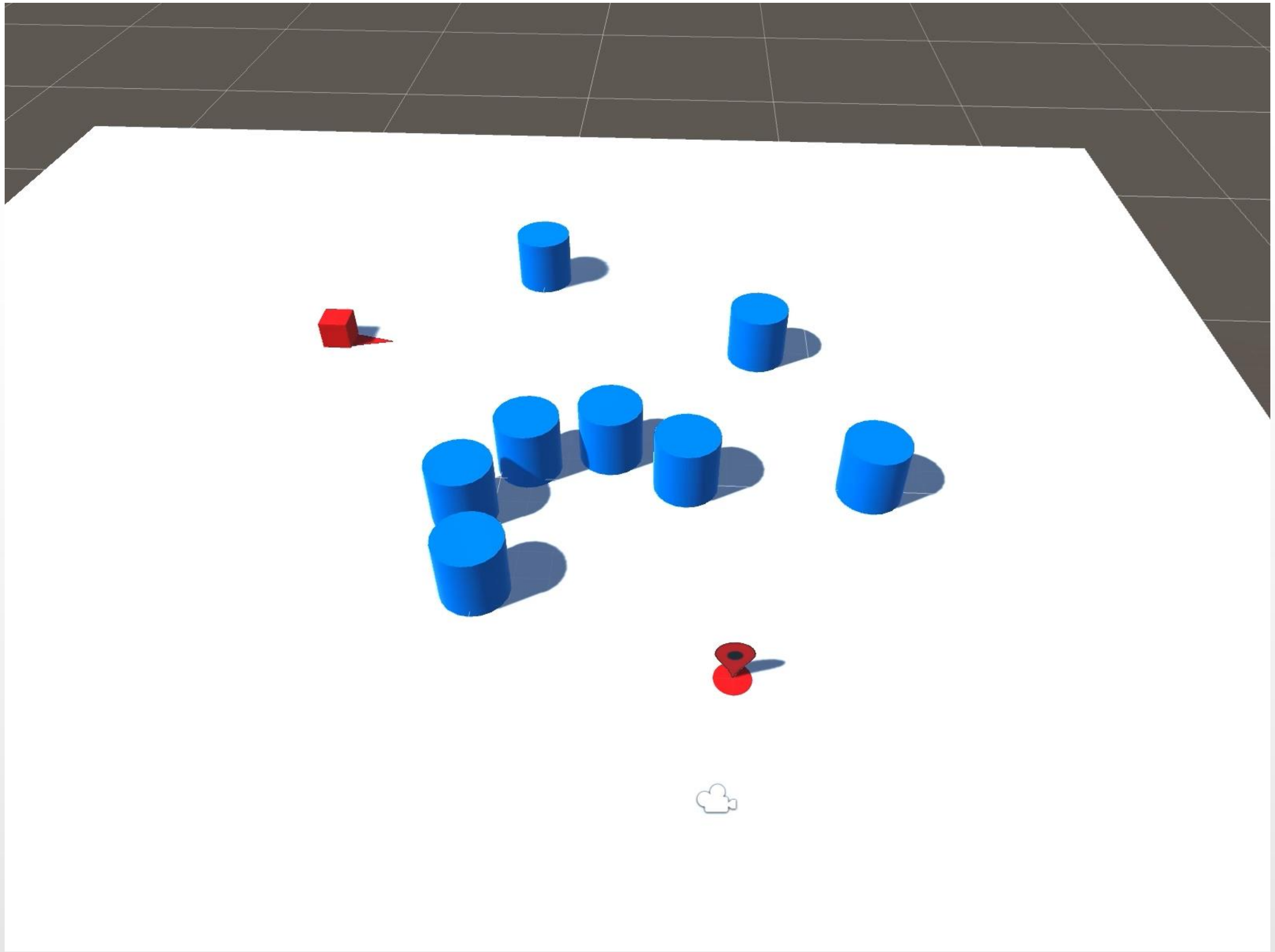
Vision angle too small



Vision angle too large

Fail Due to RayCast

Scene: RayCast Fail
Folder: Movement/Dynamic



Introducing BoxCast

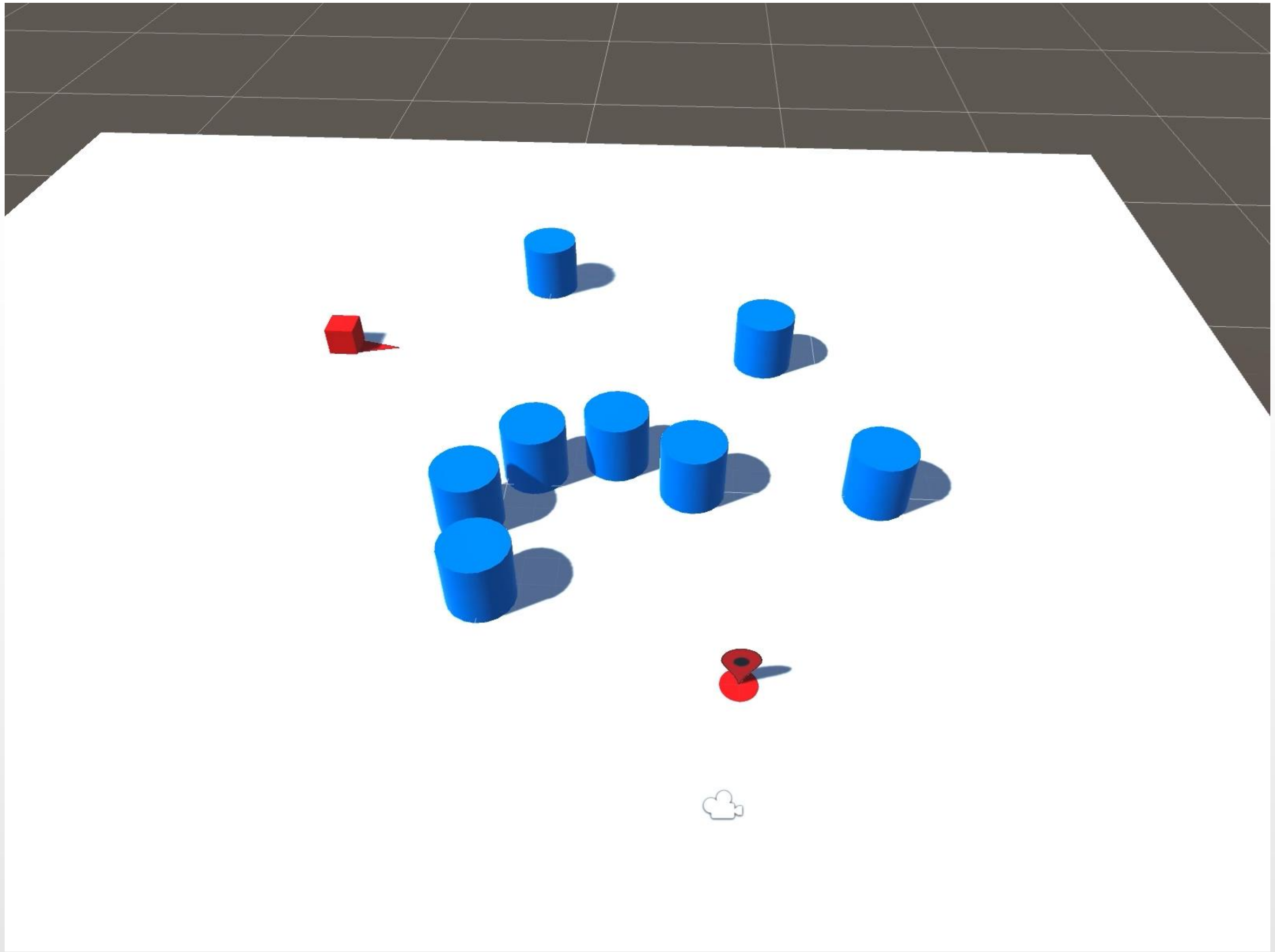
Source: AvoidBehaviourVolume
Folder: Movement/Dynamic/Delegates

All we need to to is to change the RayCast calls into BoxCast ones.

```
Collider collider = GetComponent<Collider> ();

bool leftHit = Physics.BoxCast (transform.position,
                                collider.bounds.extents,
                                Quaternion.Euler (0f, - sightAngle, 0f) * status.movementDirection,
                                transform.rotation,
                                sightRange);

bool centerHit = Physics.BoxCast (transform.position,
                                  collider.bounds.extents,
                                  status.movementDirection,
                                  transform.rotation,
                                  sightRange);

bool rightHit = Physics.BoxCast (transform.position,
                                 collider.bounds.extents,
                                 Quaternion.Euler (0f, sightAngle, 0f) * status.movementDirection,
                                 transform.rotation,
                                 sightRange);
```

Using BoxCast

Scene: BoxCast
Folder: Movement/Dynamic



And now ... the BoxCast is preventing us to stop at the the landmark, making the agent spinning around it

UNIVERSITÀ DEGLI STUDI DI MILANO

About Compound Gameobjects

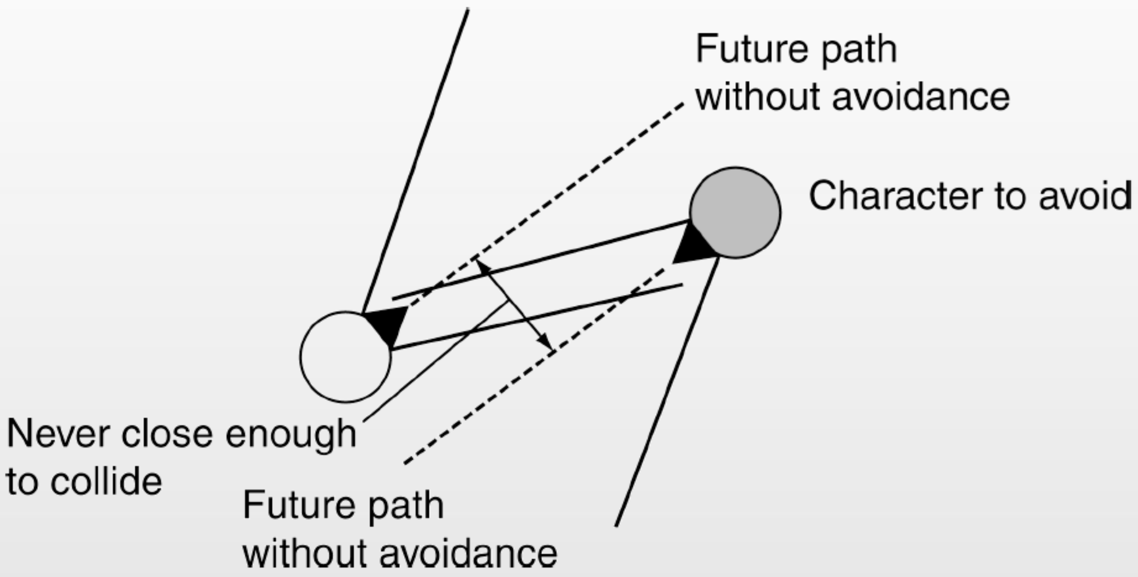
- If you have a compound gameobject, there is an easy way to calculate the global bounding box
  - We just need to iterate over all the colliders of the children objects and add them to a bounding box. At the end, the bounding box will encapsulate the global object volume

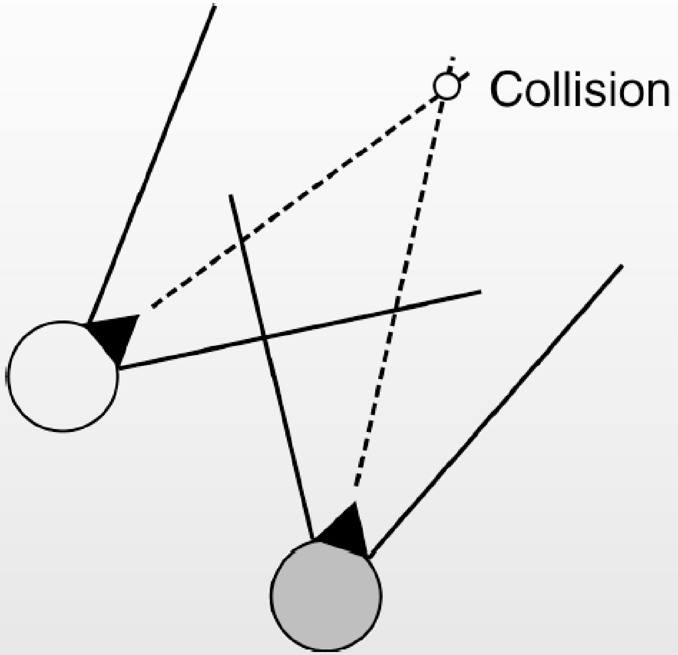
```
private Vector3 ObjectSize (GameObject go) {
    Bounds b = new Bounds (go.transform.position, Vector3.zero);
    foreach (Collider c in go.GetComponentsInChildren<Collider> ()) {
        b.Encapsulate (c.bounds);
    }
    return b.size;
}
```

- You can use (half of) the return value as that most obscure (second) parameter of the BoxCast method

Advanced Collision Avoidance

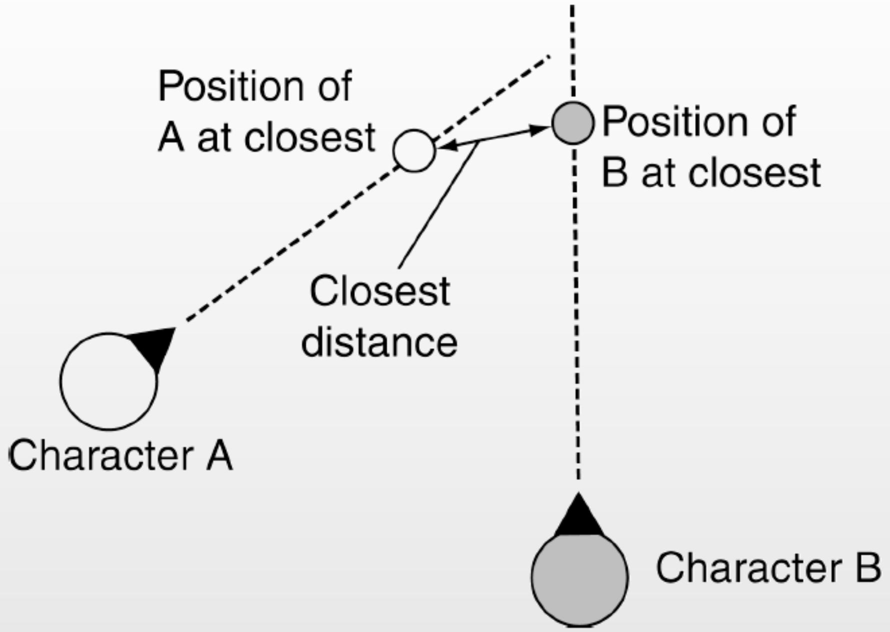
- Naive avoidance solution implies that:
  1. Characters evade even if they won't collide
  2. Collision will take place anyway





Advanced Collision Avoidance

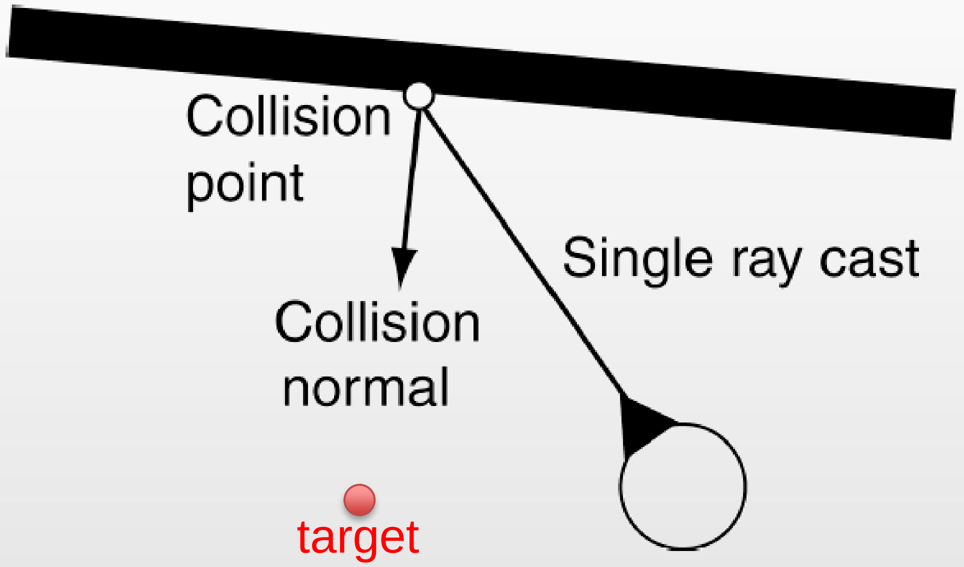
- Check whether characters will collide if they keep current velocity
  - Work out future closest point between agents and verify if their distance will be less than a certain radius, if yes: delegate to evade or separate
  - Closest points are not generally at trajectory cross



Obstacle Avoidance with Delegate Behavior

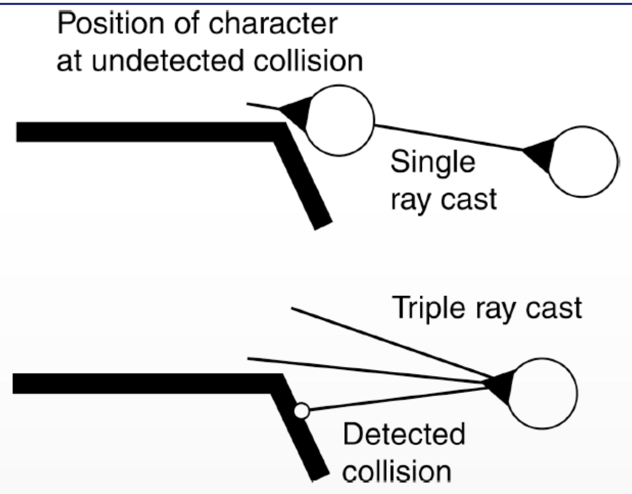
- In collision avoidance targets are inside a bounding area, this cannot be assumed for other type of obstacles
  - E.g., the bounding area of a wall occupies a whole room, keeping the agent outside of it

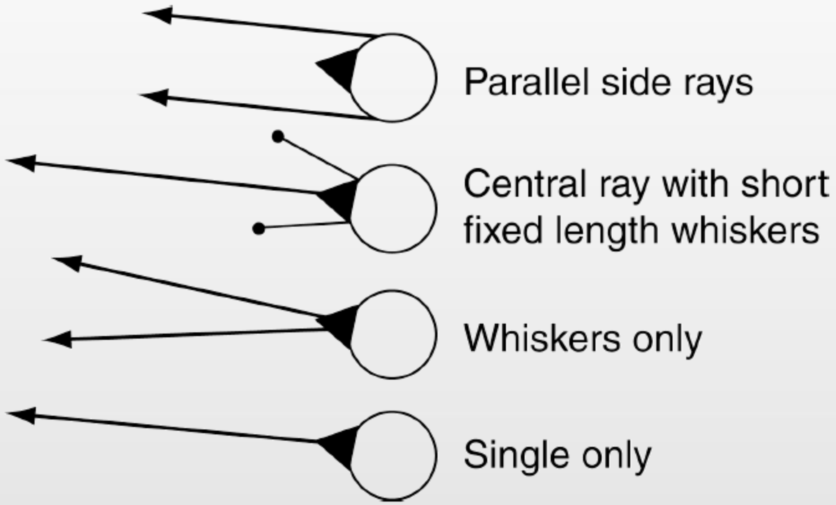
1. casts a (limited) ray in the direction it is moving
2. Check for collisions: if yes, identifies a target on the normal
3. <u>Delegates to SEEK</u> for avoidance



Obstacle Avoidance Problems

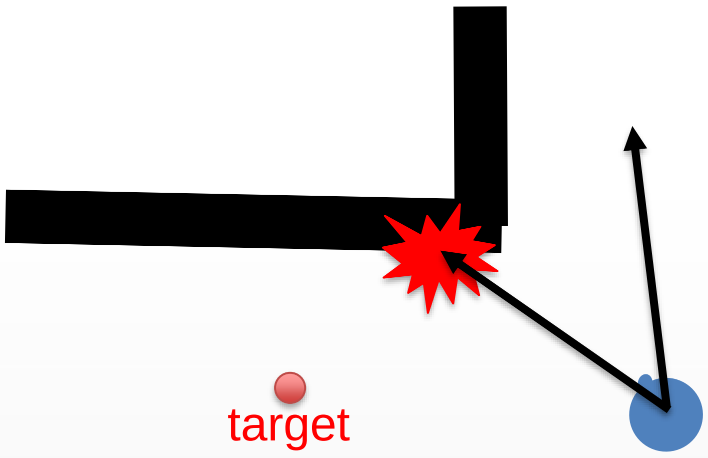
- A single ray cast is not enough in many cases
- Advanced solutions use more than 1 ray
  - No silver bullet here, it depends on the game geometry
  - Beware of not constraining your character to avoid tight passages

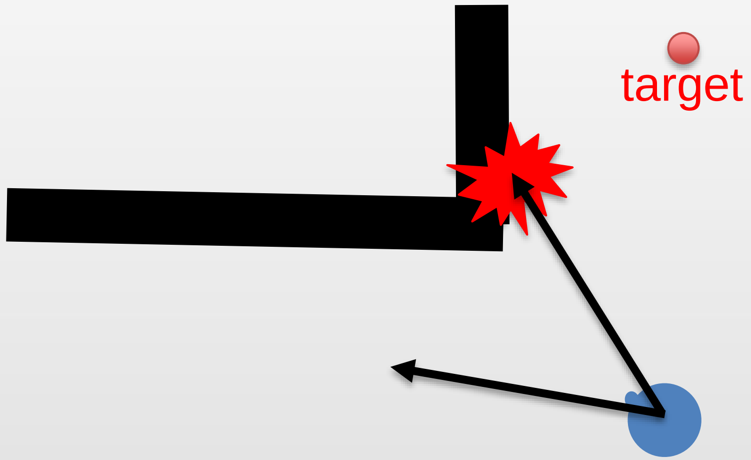




Obstacle Avoidance - Corner Trap

- Problem with acute angled corners:
  1. Left ray detects collision
  2. Right ray detects no collision
  3. Whisker based algorithm sets target to the left
  4. Now right ray detects collision
  5. Left ray detects no collision
  6. Algorithm sets target to the right

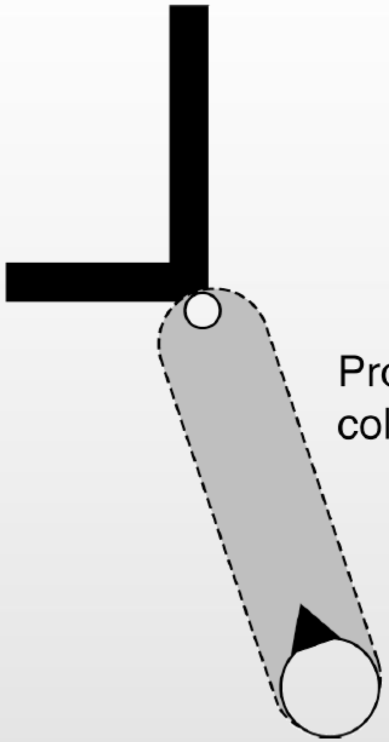




Obstacle Avoidance – Corner Trap

- Wide enough fan angles can avoid the problem
  - But a wide fan angle does not allow the agent to walk through a door
  - Possible solutions:
    1. Get level designers to make doors wide enough for AI characters
    2. Use adaptive fan angles:
       - Increase fan angles if collisions are detected
       - Decrease fan angles if collisions are not detected

- Other solutions
  - Project a volume (such as a cube)
  - Extrude a collider as a cone of vision



Moving In 3D Space

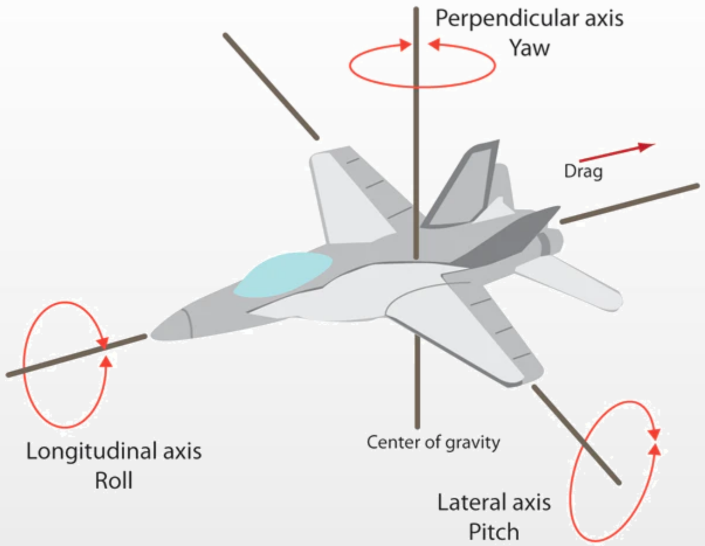
- We can still leverage on the delegate system, but the movement model must be extended

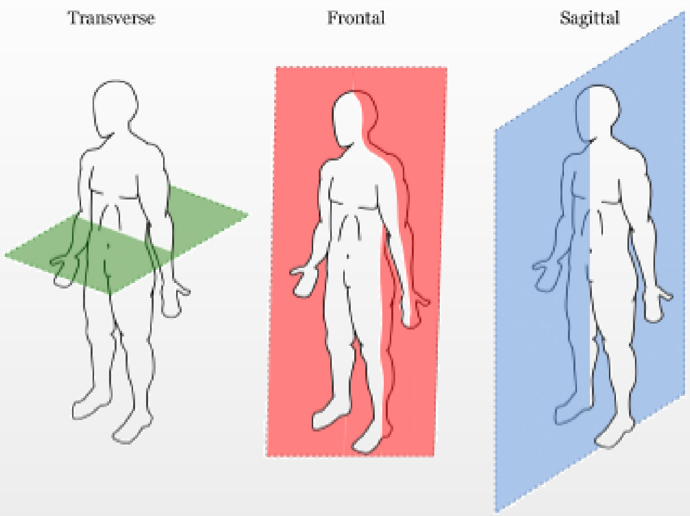
- Status configuration is more complex
  - A specific angular speed can be applied on every axis

- Position calculation
  - The main component needs to calculate new positions and orientations based on multiple angular accelerations

Easy Movement Model

- Following the same logic as before, we can describe the movement of a flying object as a linear acceleration on the longitudinal axis and a steering vector on the frontal plane
  - The vector in the frontal plane will make the agent pitch and jaw, but not roll

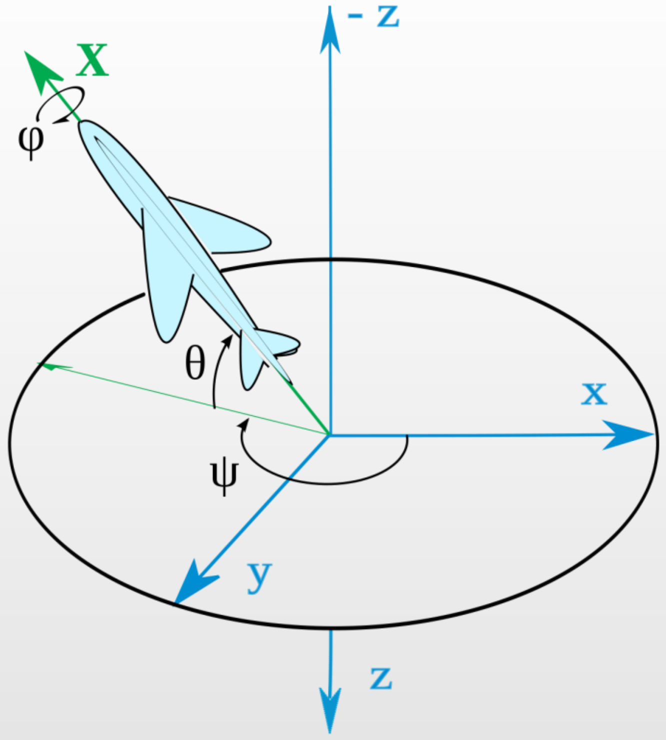




This will NOT simulate accurately an airplane or a spaceship

Complex Movement Model

- We need a mathematical model to covert a vector into a triplet of rotations (angular accelerations)
  - This is another version of the problem on converting from Cartesian Coordinates to Spherical Coordinates
  - Unfortunately, like in the case of coordinates conversion, the solution to this problem is not unique



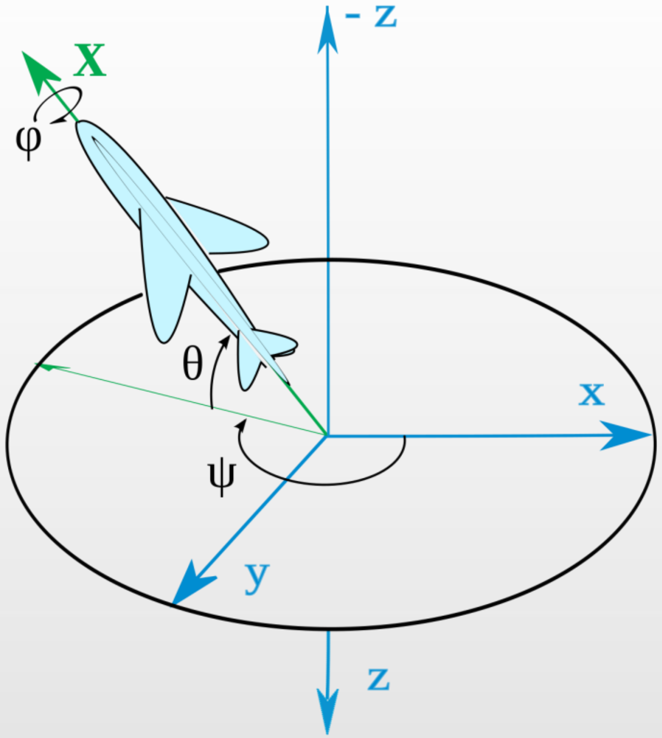
- To make the solution unique, we must state a vector AND a target plane
  - Usually, the transverse plane is used

Complex Movement Model

- We need a mathematical model to covert a vector into a triplet of rotations (angular accelerations)
  - This is another version of the problem on converting from Cartesian Coordinates to Spherical Coordinates
  - Unfortunately, like in the case of coordinates conversion, the solution to this problem is not unique
- To make the solution unique, we must state a vector AND a target plane
  - Usually, the transverse plane is used

If you think about it, the 2D model we defined is a simplification of this model: we assumed that the transverse plane is fixed. This way, it is possible to find a unique solution to the problem.

Moreover, setting the acceleration vector on the transverse plane results in roll and pitch to be always 0. Thus, yaw was enough to describe the system status.



Throwing & Jumping

- Even if you are using a 2D movement model, situations exist where you need something to move in a 3D space
  - Usually, when you throw something and when you jump

- Luckily, for this two cases we can find easy workarounds

1. Throwing something
   - Instantiate the object
   - Push it
   - Let the physics subsystem do the rest

2. Jumping
   - Set the Rigidbody velocity as the one you are using in your calculation
   - “push” the Rigidbody up
   - Do not use your movement model until you hit the ground again

Blending While Jumping

Is it reasonable to steer while jumping?

NO

YES

References

- On the textbook
  - § 3.3.14