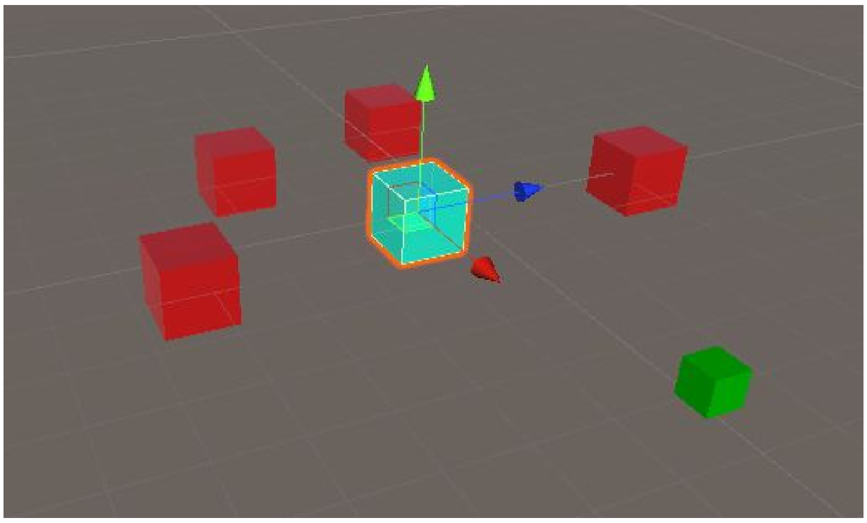
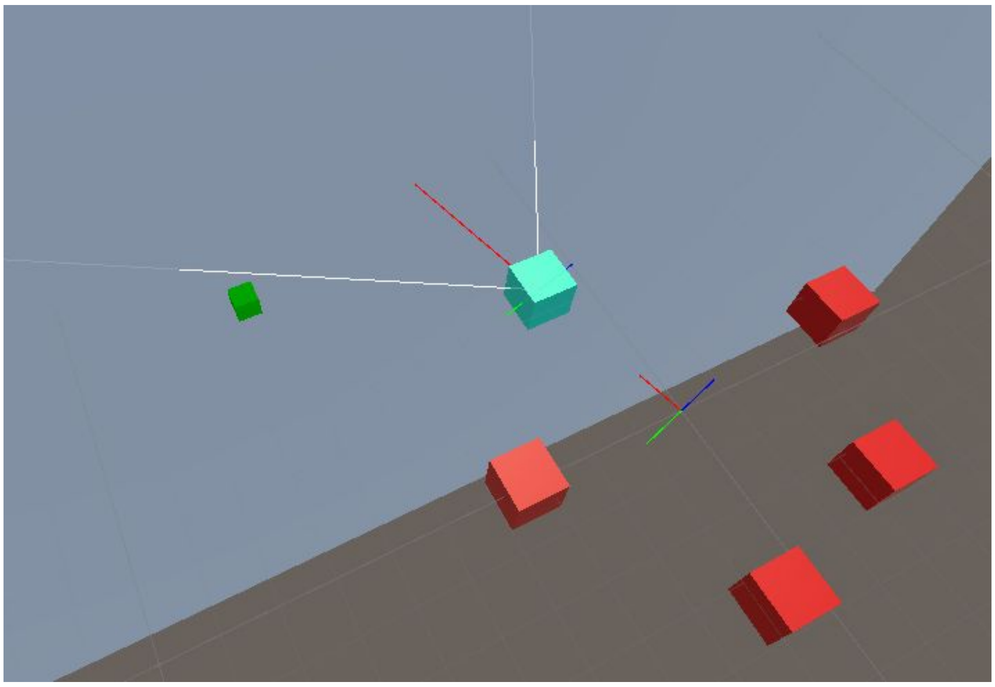
Fleeing behaviour prototype

Introduction:

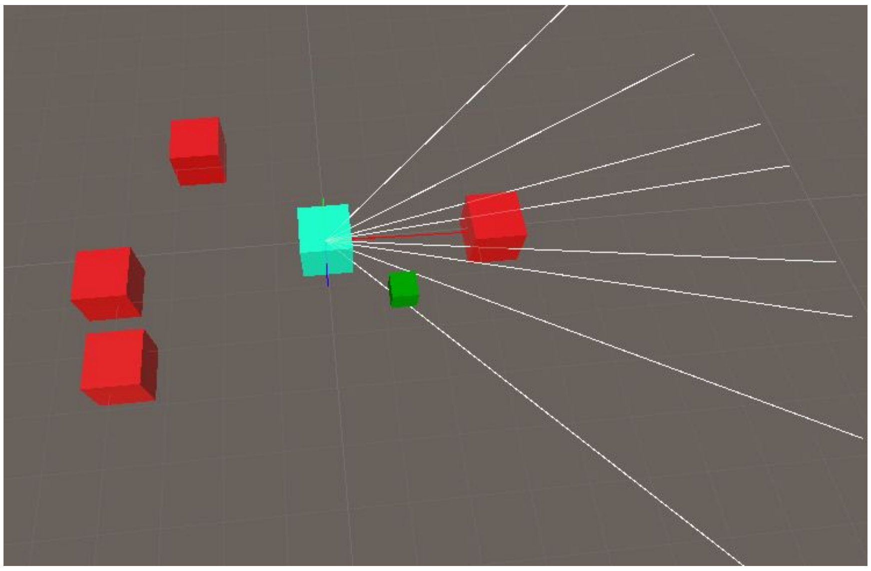
The purpose of this behaviour is to examine the location of all threats, then calculate the most optimal direction to flee in, in order to escape. The found location is visualized by a green cube in the following screenshot:



The location is found by iterating through an array of game objects - in this case the red cubes. The blue cube represents the animal wanting to flee. This operation in itself is simple, but the fleeing behaviour also considers obstacle evasion, to prevent the AI from bumping into walls and getting too easily eaten:



Obstacle evasion is done by projecting a raycast in the fleeing direction, at a fairly limited range. If anything collides with this raycast, the AI will raycast at a 45 degree angle to the left and right, to see which side has the most open space. These additional raycasts will have to connect with the same object, so as to prevent the AI from hitting something off in the background. If nothing connects at the 45 degree angle on either side, the angle is halved and the process is repeated. This happens 5 times or until a result has been found:



This allows for the AI to skirt around corners of obstacles, rather than making big turns each time something gets in the way, which (hopefully) makes for a more believable behavior.

Please note: *Make sure to set the sight range to something other than zero, when setting this system up. Otherwise the result will always be (0,0,0).*

Code:

The code of this prototype is contained in a single class, as the idea is to implement the logic so that the finite state-machine in the final build can access it with a simple function call.

1. FleeBehaviour.cs:

1.0 Fields:

1. Public List<GameObject> **aggressors**

A list of all of the game objects that the animal should consider, when calculating its escape vector.

2. Vector3 **direction**

The direction the animal should move to get away from the aggressors, calculated in global space.

3. RaycastHit **hit**

*A reference to the object hit by the raycast in the **direction** vector. Used for getting the coordinates of the collision, as well as a reference to the game object it collides with, for later comparison.*

4. Public float **sightRange**

*How far ahead the raycast in the **direction** vector will go, determining how close the animal needs to go to an object, before it starts avoiding it.*

5. Vector3 **foundPosition**

*The resulting position, which the animal is supposed to flee to. In this prototype, it is kept as an easily accessible variable, but in the final build, the **Flee( )**-function should simply return this value instead of storing it externally this way.*

1.1 void Update():

Update is only here to call the **Flee( )**-function each frame.

1.2 void Flee():

Flee finds the **direction** vector with a call to **FleeDirection**, passing in the list of **aggressors**. It then raycasts in the **direction** found, to see if anything collides ahead. If a collision is detected, **foundPosition** will be determined by a call to **AlternativeDirection** for obstacle avoidance. This call receives the **direction** vector and a reference to the object that the raycast is colliding with (**hit.transform.gameObject**), for later comparison.

1.3 Vector3 FleeDirection(List<GameObject> **_aggressors**):

FleeDirection iterates through the list of **_aggressors**, finding the delta vector between the animal and the aggressor, normalizing that vector, then storing the result by incrementing another Vector3 called **result**. This will gradually build a direction that is the accumulated delta vectors between the animal and all aggressors, as the optimal “go this way to survive” pointer. The returned **result** is also normalized, to make it easier to work with (since only the direction matters, not the magnitude).

1.4 Vector3 AlternativeDirection(Vector3 **_localFleeDirection**, GameObject **_obstacle**):

AlternativeDirection runs the bulk of the logic in the prototype. What happens first when this function is called, is that a bunch of vectors are calculated, using **_localFleeDirection** as the baseline for these calculations. Before going into the logic of how these vectors are then applied, let’s go through a quick summary of what each of the calculated vectors represent:

1. **Result**

This vector is used for a few different things. To begin with, it is simply the cross product between the “up” vector (0,1,0) and ***_localFleeDirection****. This gives a* ***result*** *that is perpendicular (at a 90 degree angle) to the* ***_localFleeDirection*** *vector. Later in the code, it will be updated to house the actual result of the whole computation, which is actually a bit confusing (there’s no reason to reuse this variable other than for memory optimization) and may be refactored in the final build.*

2. **Right**

This is the global version of the ***result*** *vector, set relative to the animal, where it is on its left side. This vector is for illustrating the principles of how the calculations work, and can be cut out of the final build.*

3. **Left**

Same as the ***right*** *vector, but on the left side of the animal by subtracting it from the animals’ position instead. Can likewise be refactored out in the final build.*

4. **LocalResult**

This is the same as ***_localFleeDirection****, but renamed for readability. Can potentially also be refactored out from the final build.*

5. **LocalRight**

This is the local, workable version of the ***right*** *vector, found by subtracting the position of the animal. This is necessary when performing any vector calculations, or the result would be different depending on where in the world the animal is located.*

6. **LocalLeft**

Same as the ***localRight*** *vector, but inverted.*

7. **MiddleRight**

*This vector will be iterated through, but its initial location is at a 45 degree angle from the **_localFleeDirection**, on the right side of the animal. This is done by simply adding **localRight** and **localResult** together, to find the average between them. It is used to determine the direction of any sideways raycasts from the animal, on the right side.*

8. **MiddleLeft**

*Same as with the **middleRight** vector, but on the left side.*

Once all of these vectors have been calculated, a for-loop is started, which is used to first check on the left and right side of the animal at a 45 degree angle. If both of these raycasts impact the same object as the one there is right in front of the animal (checked by comparing with the **_obstacle** parameter), the raycast returning the longest distance to its point of impact will be chosen. The value of **result** will be updated to now hold the point midway between the animal and the impact point.
Should both of the impact distances be exactly the same (unlikely though not impossible), either of the two sides will be chosen at random.
Should only one of the two raycasts connect with the object, the one that does not impact anything will be chosen for calculating the new point, as it is an open space.
Should none of the raycasts connect with anything, that means the object is too small to be detected at the 45 degree angle raycasts and the angle will be halved before the process is repeated.

Once a result has been found, the workable coordinate is returned by adding it to the transform of the animal. Additionally, depending on whether it is the left or right side that was chosen, **localLeft** or **localRight** is added to the returned value, to make the animal turn just a little sharper.

1.5 bool VerifyHit(GameObject **_go**, RaycastHit **_hit**):

VerifyHit simply compares **_go** with the gameObject stored in **_hit**. If these two are the same, the function returns true. If they are two different objects, it returns false.

1.6 void OnDrawGizmos():

This is a debugging function used to display a green cube, at the location that the animal wants to flee to.