# 11-8 Predataor

Last time we made a “Creature” who will serve as prey for predators. Now we’ll build the predator.

### Position the Fish GameObject

Drag the Prefab Player Container into the game and put it in front of the FPC

* Open the Player (FPC) gameObject
* Drop Player Container on to the FPC / Main Camera
* Set the coordinates to 0, 0, 10 (put it 10 meters in front)
* Set the Y rotation to 90 degrees

### Align the Scene Camera

* Select the Main Camera
* Go to the menu GameObject and you’ll see near the bottom “Align view to selected.”

### The Fish Game Object

* Body is a sphere: scale 0.5, 0.5, 1, which makes it longer in the z (forward) axis
* Nose is an elongated red sphere positioned to make it easier to see the front
* Tail is an empty gameObject with a trail renderer attached
* AddComponent / Physics / RigidBody
  + To imitate behavior in water:
    - Set use gravity to false
    - Set mass to 0.1
    - Set drag to 10

### Making the Fish Move

We want to move the predator by applying force in a particular direction. So, to begin we need the following:

* rb –a reference to the rigidBody attached to the fish
* targetDirection – a Vector3 representing the position we want to move toward. We want that to reference our current position and we want it always to have an absolute magnitude of less than one.
* AddForce – a call that applies a force toward a particular point
* Quaternion.LookRotation() – rotates the object to point at a particular point
* Debug.DrawRay – a function that will draw a line toward a particular point. This is only visible in the Scene window and will tell us if we are pointing where we intend to point.
* FixedUpdate – a version of the Update function that fires at fixed intervals

So, to make all this happen, create a C# script called P0 and add the following:

    Rigidbody rb;  
    Vector3 targetDirection;  
  
    void Start () {  
        rb = GetComponent<Rigidbody>();  
    }  
  
  
    void FixedUpdate () {  
       targetDirection = (Random.onUnitSphere + transform.position).normalized;       rb.AddForce(targetDirection \* 5f);  
       transform.rotation = Quaternion.LookRotation(targetDirection);  
       Debug.DrawRay(transform.position, moveToward, Color.green, .2f);  
    }  
}

Clearly that is only part of what needs to happen. We can make serious progress simply by reorganizing our code. Notice that it doesn’t do anything new. It does, how ever, only change direction once..

public class P0 : MonoBehaviour {  
  
    Rigidbody rb;  
    Vector3 targetDirection;  
  
    void Start () {  
        rb = GetComponent<Rigidbody>();  
        SetTargetDirection();  
    }  
          
    void FixedUpdate () {  
        Move();  
    }  
  
    void SetTargetDirection(){  
       targetDirection = (Random.onUnitSphere + transform.position).normalized;  
    }  
  
    void Move(){  
        rb.AddForce(targetDirection \* 5f);  
        transform.rotation = Quaternion.LookRotation(targetDirection);  
        Debug.DrawRay(transform.position, targetDirection, Color.green, .2f);  
    }  
}

The obvious problem here is that the fish never changes direction. We can fix that by using a call to InvokeRepeating, which will call a function a given amount of time.

void Start () {  
        rb = GetComponent<Rigidbody>();  
        SetTargetDirection();  
    }

That changes the start function to this:

void Start () {  
        rb = GetComponent<Rigidbody>();  
        InvokeRepeating ("SetTargetDirection", 0f, 4f);  
    }

Also, notice the suddenness with which it changes direction. We’d like it to rotate from one direction to another. We can do that with new a function that will give us incremental rotations and a new Vector3 that will keep track of the incremental changes. We’ll also add a debug.DrawRay call to show us the current direction.

The variables now look like this:

using System.Collections;  
using System.Collections.Generic;  
using UnityEngine;  
  
public class P0 : MonoBehaviour {  
  
    Rigidbody rb;  
    Vector3 targetDirection;  
**Vector3 currentDirection;**  
  
    void Start () {  
        rb = GetComponent<Rigidbody>();  
**InvokeRepeating("SetTargetDirection", 0f, 3f);**  
    }  
          
    void FixedUpdate () {  
        Move();  
    }  
  
    void SetTargetDirection(){  
        targetDirection = Random.onUnitSphere;  
    }  
  
    void Move(){  
  
        currentDirection = Vector3.RotateTowards(transform.forward, targetDirection, .05f, 0.1F);  
        rb.AddForce(currentDirection \* 5f);  
        transform.rotation = Quaternion.LookRotation(currentDirection);  
  
        Debug.DrawRay(transform.position, targetDirection \* 2, Color.green, .2f);  
        Debug.DrawRay(transform.position, currentDirection \* 2, Color.red, .2f);  
    }  
}

And the Move function looks like this:

void Move(){  
 **currentDirection**=

Vector3.RotateTowards(transform.forward, **targetDirection**, .05f, 0.1F);

rb.AddForce(**currentDirection**\* 5f);  
 transform.rotation = Quaternion.LookRotation(**currentDirection**);  
  
 Debug.DrawRay(transform.position, **targetDirection**\* 2, Color.green, .2f);  
 Debug.DrawRay(transform.position, **currentDirection**\* 2, Color.red, .2f);  
}

### Script P1

The script we just wrote is a good start. I’m going to keep it in its current form. I’ll

* Duplicate P0.
* Rename the new script P1.
* Open it and rename the class P1.
* Attach the new script to the Fish object
* Deactivate the P0 script that is on the fish object.

The new script looks exactly like P0, except for the name change.

### Keeping the Fish from wandering

It would be nice if the fish didn’t just wander off. In fact, we’d like to give the fish behaviors so it can do different things under different circumstances. For example, if it gets too far from its starting position, we’d like it to turn around and head back.

Of course, if it is going to go back to where it started, we need to keep track of that, with a Vector3. We’ll call it startPosition and, because we can declare variables of the same type in the same line, we can do it like this:

    Rigidbody rb;  
    Vector3 targetDirection, currentDirection, startPosition;

There are also lots of ways to keep track of different behavior. One of them would be to create

a new data type just to do the job. It looks like this:

enum Fish {swimming, returning, hunting};  
    Fish action;

In this case, Fish is the new data type and action is an instance of Fish that can hold one of three values: swimming, returning, or hunting.

We’ll also add two lines to the Start function

    void Start () {  
        rb = GetComponent<Rigidbody>();  
        InvokeRepeating ("SetTargetDirection", 0f, 2f);  
**startPosition = transform.position;  
        action = Fish.swimming;**  
    }

The real changes come in the SetTargetDirection() function. First we want to know the distance from our current position to the start position.

First, if the fish is swimming, then if it is more than a certain distance from where it started, we want to change its action to returning. If it is not too far from where it started, we’ll just ask it to turn in a random direction.

If the fish is returning then, if the distance is less than a certain amount, we’ll tell it to start swimming again.

    void SetTargetDirection(){  
        float distance = (transform.position - startPosition).magnitude;  
  
        if(action == Fish.swimming){  
            targetDirection = Random.onUnitSphere;  
            if(distance > 5f){  
                action = Fish.returning;  
            }  
        } else if(action == Fish.returning) {  
            targetDirection = (startPosition - transform.position).normalized;  
            if(distance < 1f){  
                action = Fish.swimming;  
            }  
        }  
    }

### Script P2

Once this works, we’ll create a new script and go from there.

* Duplicate P1.
* Rename the new script P2.
* Open it and rename the class P2.
* Attach the new script to the Fish object
* Deactivate the P1 script that is on the fish object.

The new script looks exactly like P1, except for the name change.

### Making the Fish Hunt

To make the fish hunt, we use colliders. First, the trigger

* Select the Fish object.
* In the inspector, look at the sphere collider and check is trigger
* Make the radius 2
* Set the center to 0, 0. 2.3 (so the trigger is in front of the fish.

With the fish object still selected go to AddComponent / Physics / Sphere Collider. We want a small collider to sit right at its nose.

* Set the radius to .2
* Set the position to 0, 0, .6

Next we need to add a variable of type transform to keep track of the prey. Then our variable list will look like this:

    Rigidbody rb;  
    Vector3 targetDirection, currentDirection, startPosition;  
**Transform prey;**  
  
    enum Fish {swimming, returning, hunting};  
    Fish action;

We’ll add one more behavior to SetTargetDirection so it looks like this:

        void SetTargetDirection(){  
        float distance = (transform.position - startPosition).magnitude;  
  
        if(action == Fish.swimming){  
            targetDirection = Random.onUnitSphere;  
            if(distance > 5f){  
                action = Fish.returning;  
            }  
        } else if(action == Fish.returning) {  
            targetDirection = (startPosition - transform.position).normalized;  
            if(distance < 1f){  
                action = Fish.swimming;  
            } **else if(action == Fish.hunting){  
                targetDirection = (prey.position - transform.position).normalized;  
            }**        }  
    }

But the new parts will involve the colliders. When something comes into the trigger zone, want to think about two things. If the thing we’ve detected is the terrain, we want to turn away from it. Setting the action to returning will do just that.

But, if what we’ve run into is edible, we want to chase it. Then we set action to hunting and put the transform of the other object into prey so we can keep swimming toward it.

    void OnTriggerEnter(Collider otherObj){  
        if(otherObj.gameObject.tag == "Terrain"){  
            action = Fish.returning;  
            SetTargetDirection();  
        } else if(otherObj.gameObject.tag == "Prey"){  
            action = Fish.hunting;  
            prey = otherObj.transform;  
            SetTargetDirection();  
        }  
    }

Of course, the prey might get away, so we watch for it in an OnTriggerExit function.

    void OnTriggerExit(Collider otherObj){  
        if(otherObj.gameObject.transform = prey){  
            action = Fish.swimming;  
            SetTargetDirection();  
            print("missed it");

        }  
    }

But, we are swimming at the thing, and if we get close enough to hit it with the collider, we can destroy it.

    void OnCollisionEnter(Collision otherObj){  
        if(otherObj.gameObject.transform = prey){  
            Destroy(otherObj.gameObject);  
            action = Fish.swimming;  
            SetTargetDirection();  
            print("got it");  
        }  
    }

Last Wednesday we worked on objects tagged prey. If this fish comes across anything with that tag, it will chase it and destroy it.

But there are important things it does not do. It doesn’t change speeds while hunting. It doesn’t reproduce and it doesn’t die. It also has no real strategy, other than swim randomly in a fixed volume.