Link to Lab Drive

https://drive.google.com/drive/folders/1jgFTWCJ5JD-JnIY7n_L7s2n2vA1F9nNX?usp=sharing

OR

https://tinyurl.com/ta-comp476-daniel

Movement Behaviors

Kinematic and Steering Movement

Kinematic behaviors

- Use position and orientation data (no velocities) and output the desired velocity.
- The output often results in **instant movement** (moving at max speed or being stationary).
- Many games simplify things even further and simply force the orientation of a character to be in the direction it is traveling. If the character is stationary, it usually just faces the last direction it was moving in. If its movement algorithm returns a new velocity, then that is used to set its orientation.

Steering behaviors

- Use kinematic output and current velocities (both character and target) and outputs a steering force used for acceleration.

Rotational behaviors

- Still considered movement behaviors however these usually do not directly affect velocity.
- Kinematic output is an angular component that represents a new rotation
- Steering output is an angular component that represents the difference between the kinematic output and the current rotation.

The target information depends on the behavior. It could be a reference to another agent, Transform, a Vector3, the world geometry, etc. For chasing or evading behaviors, the target is often another moving character. Obstacle avoidance behaviors take a representation of the collision geometry of the world as input. It is also possible to specify a path as the target for a path following behavior.

Rotational Behaviors

Align: The agent looks in the direction that the target is looking in.

```
Quaternion KinematicAlign()
{
    if (transform.forward == target.forward || Mathf.Approximately(target.forward.magnitude, 0))
        return transform.rotation;

    return Quaternion.LookRotation(target.forward);
}
```

Face: Makes the agent look at (face) the target. **FaceAway** is just the opposite of Face.

```
Quaternion KinematicFace()
{
    Vector3 direction = target.position - transform.position;

    if (direction.normalized == transform.forward || Mathf.Approximately(direction.magnitude, 0))
        return transform.rotation;

    return Quaternion.LookRotation(direction);
}
```

LookWhereYouAreGoing: The agent looks in the direction that it is moving.

```
Quaternion KinematicLookWhereYouAreGoing()
{
    if (currentVelocity == Vector3.zero)
        return transform.rotation;

    return Quaternion.LookRotation(currentVelocity);
}
```

Rotational Behaviors

The **steering output** for a rotational behavior is an angular component that represents the difference between the kinematic output's angular component and the current angular component (rotation) of the agent.

```
Quaternion XXX() // XXX is a rotational behavior
{
    return Quaternion.FromToRotation(transform.forward, KinematicXXX() * Vector3.forward);
}
```

The above solution will simply get the rotation directly from the agent's forward vector to the forward vector after the final rotation. However, in some cases we may want to lock the rotation around the Y axis only (ex: a car does not do a backflip when it needs to quickly turn 180 degrees).

The solution here is to remove the y component of the from and to vectors and then get the angle between the two. Because we are working with quaternions in these examples, we will have to do some quaternion math and fortunately for us Unity provides some nice functions for Quaternion and Vector3 that take care of some of the math for us.

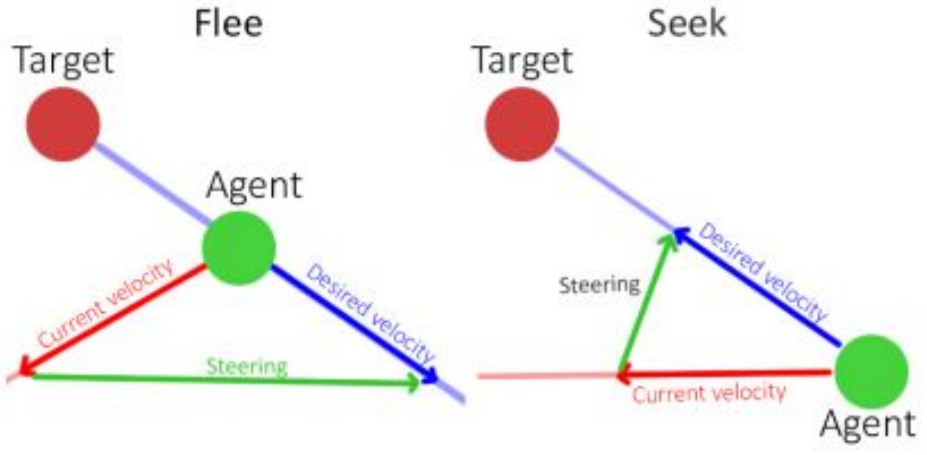
```
Quaternion XXX() // XXX is a rotational behavior
{
    // get the rotation around the y-axis
    Vector3 from = Vector3.ProjectOnPlane(transform.forward, Vector3.up);
    Vector3 to = KinematicXXX() * Vector3.forward;
    float angleY = Vector3.SignedAngle(from, to, Vector3.up);
    return Quaternion.AngleAxis(angleY, Vector3.up);
}
```

Seek and Flee

```
Vector3 KinematicSeek()
{
    Vector3 desiredVelocity = target.position - transform.position;
    desiredVelocity = desiredVelocity.normalized * maxVelocity;
    return desiredVelocity;

}
Vector3 Seek()
{
    Vector3 desiredVelocity = target.position - transform.position;
    desiredVelocity = desiredVelocity.normalized * maxVelocity;
    Vector3 steering = desiredVelocity - currentVelocity;
    return steering;
}
Vector3 KinematicFlee()
{
    Vector3 desiredVelocity = transform.position - target.position;
    desiredVelocity = desiredVelocity.normalized * maxVelocity;
    return desiredVelocity;
}
Vector3 Flee()
{
    Vector3 desiredVelocity = transform.position - target.position;
    desiredVelocity = desiredVelocity.normalized * maxVelocity;
    Vector3 steering = desiredVelocity - currentVelocity;
    return steering;
}
```

The most simplest behaviors are Seek and Flee. Seek moves the agent towards the target position and Flee moves away from the target (opposite of Seek).

- As we can see, the kinematic behaviors just care about positions and output the new desired velocity, whereas the steering behaviors calculate the desired velocity (same as kinematic) and then output the force needed to go from current velocity towards the new desired velocity.

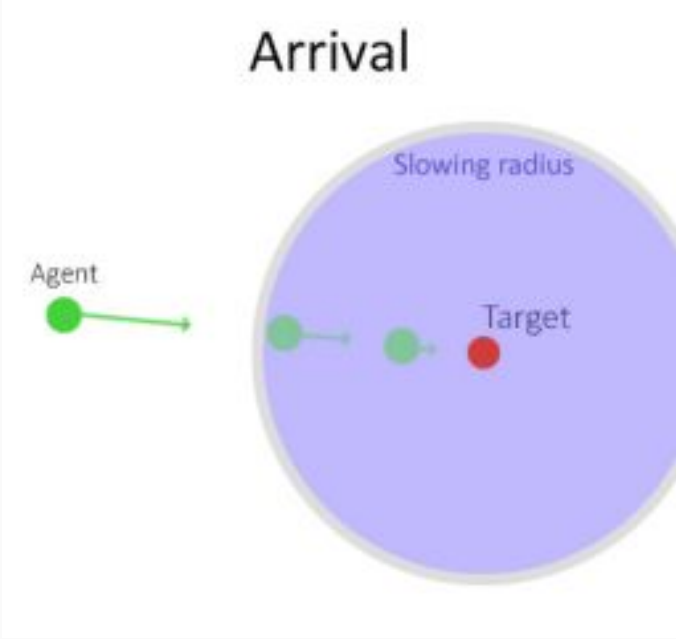


Arrive

Seek behavior, at the moment, has a problem when trying to reach its destination. It will overshoot the destination which will cause a sort of oscillating effect.

The **Arrive** behavior extends the Seek behavior and fixes this problem by introducing a “slowing radius”. When the agent is inside this radius, the seeking velocity is decreased until the agent reaches its destination in which the seeking velocity will be zero. We can also add a “stopping radius” if we don’t want the agent to arrive exactly at the target position.

```
Vector3 KinematicArrive()
{
    Vector3 desiredVelocity = target.position - transform.position;
    float distance = desiredVelocity.magnitude;
    desiredVelocity = desiredVelocity.normalized * maxSpeed;

    if (distance <= stopRadius)
        desiredVelocity *= 0;
    else if (distance < slowRadius)
        desiredVelocity *= (distance / slowRadius);

    return desiredVelocity;
}
Vector3 Arrive()
{
    return KinematicArrive() - currentVelocity;
}
```



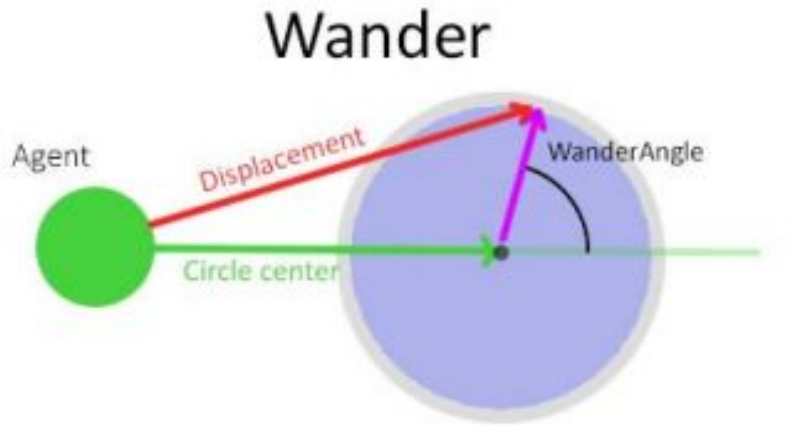
Wander

Wandering is often used in games when the agent does not have a direct task to do and waits for something to happen. A simple implementation is just to have a set interval in where it calculates a random position and seeks to that position. Unfortunately, this can result in some unrealistic looking behavior since every interval, the target position suddenly changes.

Another implementation makes use of a circle that is from a set distance from the agent's position. A random unit vector on the circle is taken and that vector is added to the circle position. This result is now the new target and the agent can delegate to Seek behavior to seek out the new target. Every so often, the wander angle is incremented by a small random amount.

```
Vector3 KinematicWander()
{
    wanderTimer += Time.deltaTime;

    if (lastWanderDirection == Vector3.zero)
        lastWanderDirection = transform.forward.normalized * maxSpeed;

    if (lastDisplacement == Vector3.zero)
        lastDisplacement = transform.forward;

    Vector3 desiredVelocity = lastDisplacement;
    if (wanderTimer > wanderInterval)
    {
        float angle = (Random.value - Random.value) * wanderDegreesDelta;
        Vector3 direction = Quaternion.AngleAxis(angle, Vector3.up) * lastWanderDirection.normalized;
        Vector3 circleCenter = transform.position + lastDisplacement;
        Vector3 destination = circleCenter + direction.normalized;
        desiredVelocity = destination - transform.position; // <-- displacement
        desiredVelocity = desiredVelocity.normalized * maxSpeed;

        lastDisplacement = desiredVelocity;
        lastWanderDirection = direction;
        wanderTimer = 0;
    }

    return desiredVelocity;
}
```

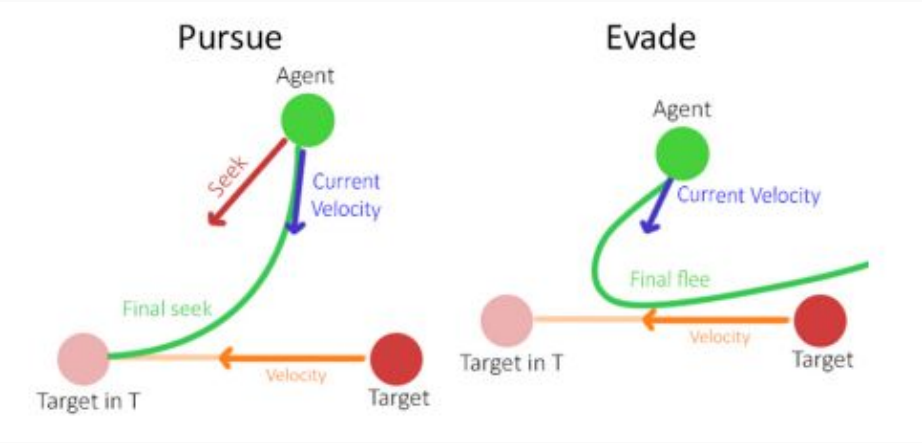
```
Vector3 Wander()
{
    return KinematicWander() - currentVelocity;
}
```



Pursue and Evade

Pursue is similar to Seek except that Pursue predicts where the target will be in time T so that it can intercept the target. For the implementation, the agent must know the velocity of the target to predict where the target will be. To improve the result, the pursue acceleration decreases depending on how close the target is. Evade is just the opposite of Pursue, It considers the velocity of the target and flees from the position where the target will be in time T.

```
Vector3 Pursue()
{
    float distance = Vector3.Distance(target.position, transform.position);
    float ahead = distance / 10;
    Vector3 futurePosition = target.position + targetVelocity * ahead;

    // Seek()
    Vector3 steering = KinematicSeek(futurePosition) - currentVelocity;

    return steering;
}
Vector3 Evade()
{
    float distance = Vector3.Distance(transform.position, target.position);
    float ahead = distance / 10;
    Vector3 futurePosition = target.position + targetVelocity * ahead;

    // Flee()
    Vector3 steering = KinematicFlee(futurePosition) - currentVelocity;

    return steering;
}
```



Steering Behaviors

Combining Steering Behaviors

One of the biggest advantages of steering behaviors is that they can be easily combined by just adding all the forces together.

- After adding them all, you add it to your current velocity and truncate the result so it does not exceed the agent's max speed and you now have applied your combined steering force.
- Adding a Quaternion to another is done by multiplication ($C = A * B$)
- Subtracting a Quaternion from another is done by multiplication with its inverse ($C = A * B^{-1}$)
- Recall that if you want to apply a quaternion rotation to a vector, you can do so by multiplication ($v_2 = Q * v$)

```
Vector3 steeringForce = GetSteeringForceSum();
Vector3 acceleration = steeringForce / mass; // optional
Velocity += acceleration * Time.deltaTime;
Velocity = Vector3.ClampMagnitude(Velocity, maxSpeed);

transform.position += Velocity * Time.deltaTime;
```

Using Weights

Another advantage is that every steering force can have its own weight. By just multiplying a certain behavior's output with a weight, that specific behavior can influence the resulting force more or less to achieve many different results.

Tasks

Implement Kinematic Behaviors

- Modify the GetKinematic() methods in provided scripts located in the folder: `Assets/_Runtime/_Scripts/AI/`. You must implement the code for the following behaviors:
  - Seek and Flee
  - Arrive
  - Wander
  - Face Away
  - LookWhereYouAreGoing
- Modify the AIAgent script : Average the kinematic output from all behaviors attached to each agent and apply it.

Implement Steering Behaviors

- Modify the GetSteering() methods in provided scripts located in the folder: `Assets/_Runtime/_Scripts/AI/`.
  - Seek and Flee
  - Arrive
  - Wander
  - Face Away
  - LookWhereYouAreGoing
- Modify the AIAgent script : Sum up the steering output from all behaviors attached to each agent and apply it.

Thank You

Link to Lab Drive

https://drive.google.com/drive/folders/1jgFTWCJ5JD-JnIY7n_L7s2n2vA1F9nNX?usp=sharing

OR

https://tinyurl.com/ta-comp476-daniel

Other Links

https://docs.unity3d.com/2020.3/Documentation/Manual/index.html

https://docs.unity3d.com/2020.3/Documentation/Manual/ExecutionOrder.html