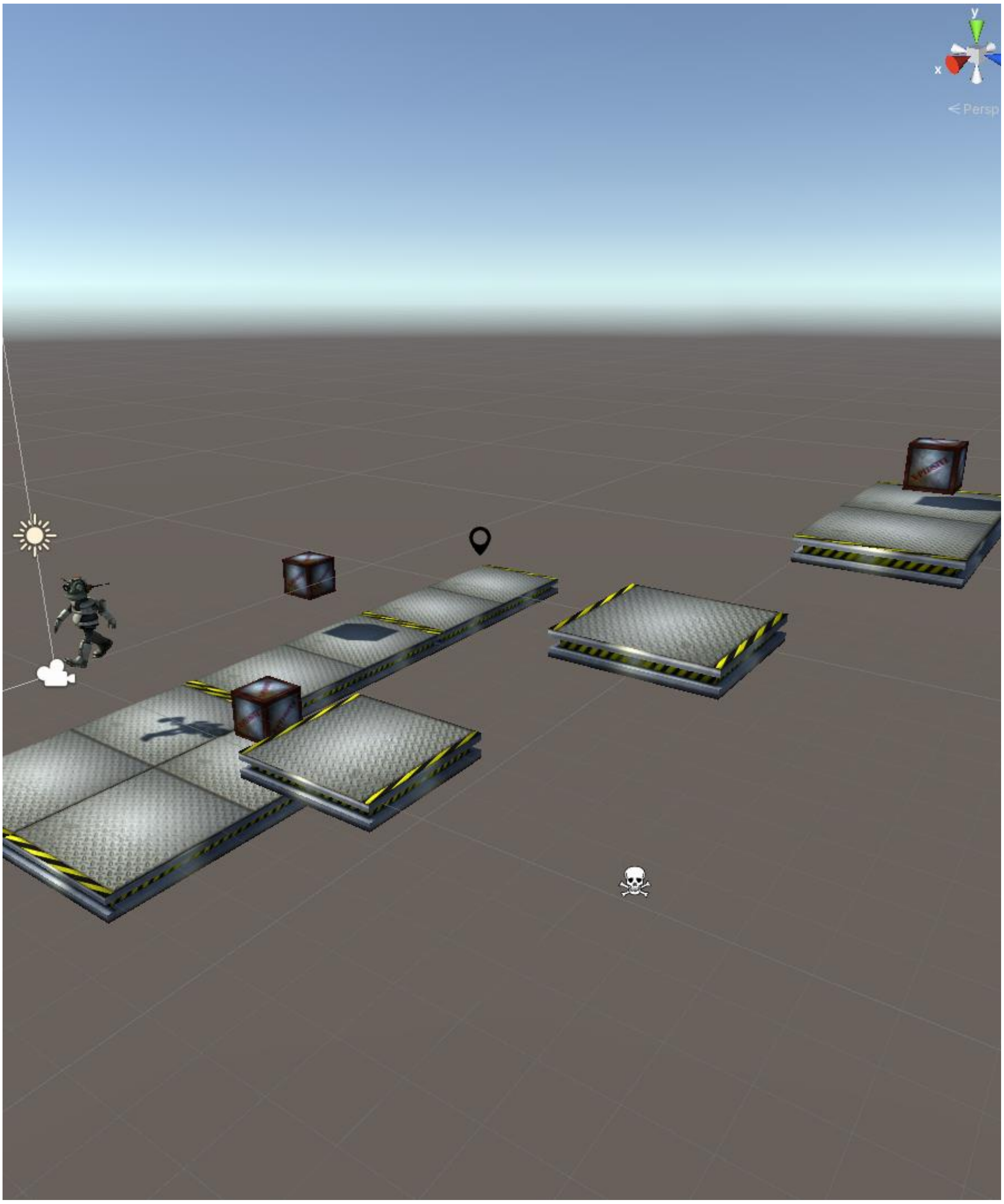
Professur für Erweiterte Realität
Fakultät für Informatik
Technische Universität München

Tutorial & Exercises – Tutorial 9

Platformer



Enemy behavior

Every game needs some form of enemies. In this tutorial, we will implement a basic form of enemy behavior.

Preparation

Before we can begin with enemy behavior, we need to prepare some scripts.

- Create a new script called **AdvancedEnemy.cs**.
- Create a new capsule and assign the script AdvancedEnemy to it.
- Make sure that your Enemy has a Rigidbody component attached.
- Create the following variables in your Class AdvancedEnemy. (Use [SerializeField] to make the variables accessible through the inspector.)
- Initialise the field enemyRigidbody in Awake()

```
private float ChaseSpeed;
private float NormalSpeed;
private GameObject Prey;
private Rigidbody enemyRigidbody;

void Awake(){
    enemyRigidbody = GetComponent<Rigidbody>();
}
```

Create an enum called **Behaviour** to switch the behavior of your enemy. Create a variable of type **Behaviour**. You can set this value in the inspector.

```
public enum Behaviour
{
    LineOfSight,
    Intercept,
    PatternMovement,
    ChasePatternMovement,
    Hide
}
public Behaviour behaviour;
```

In the FixedUpdate method create a switch statement to allow different behavior with the same script.

```
switch (behaviour)
{
    case Behaviour.LineOfSight: //Exercise 1
      break;
    case Behaviour.Intercept: //Exercise 2
      break;
    case Behaviour.PatternMovement: //Exercise 3
      break;
    case Behaviour.ChasePatternMovement: //Exercise 4
      break;
    case Behaviour.Hide: //Exercise 5
      break;
    default:
      break;
}
```

Exercise 1: Line of sight chasing

Your enemy will chase the player in a direct line.

Create a new method called ChaseLineOfSight.

```
private void ChaseLineOfSight(Vector3 targetPosition, float Speed)
```

In this method, we calculate the vector between the enemy and the target that we are chasing. Calculate that vector and save it in a new variable called direction.

```
Vector3 direction = targetPosition – transform.position;
direction.Normalize();
```

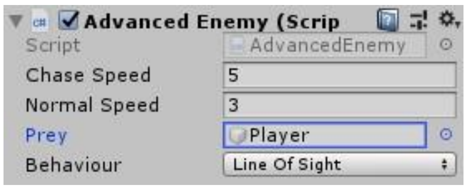
Use the velocity of the enemies' rigidbody to move the enemy towards the target.

```
enemyRigidbody.velocity = new Vector3(direction.x * Speed, enemyRigidbody.velocity.y, direction.z * Speed);
```

Finally, call the just created method in FixedUpdate in the first case of the switch statement. Use Prey.transform.position as the targetPosition and pass the variable ChaseSpeed to the method.

```
void FixedUpdate()
{
    switch (behaviour)
    {
    ….
        case Behaviour.LineOfSight: //Exercise 1
            ChaseLineOfSight(Prey.transform.position, ChaseSpeed);
            break;
        default:
            break;
}
```

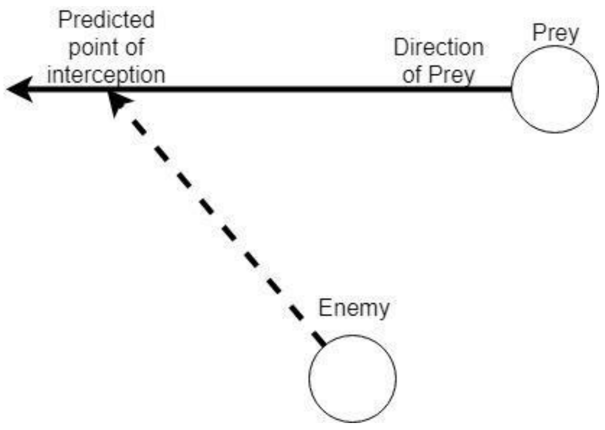
In the inspector assign the corresponding variables.



Advanced Enemy (Scrip

Script	AdvancedEnemy
Chase Speed	5
Normal Speed	3
Prey	Player
Behaviour	Line Of Sight

Exercise 2: Intercepting



The enemy now has knowledge of the speed and direction of the player. Instead of moving directly towards the player, the enemy now predicts the player's movement and tries to intercept the player.

The algorithm predicts a point of interception. The algorithm calculates a **time to close**. The time that it would take the enemy to get to the prey, based on their current velocities. Let's assume it would take the enemy 1s to travel to the prey. After 1s the prey will already have moved further. Therefore, the algorithm predicts where the prey will be in 1s. This is called the **Point of Interception.**

Go ahead and create a new function Intercept.

```
private void Intercept(Vector3 targetPosition){
}
```

To predict the **point of interception** you need to compute the relative velocity between player and enemy, as well as the time to close.

- Create the following variables in your just created method Intercept()
- Compute the relative velocity between prey and enemy
- Compute the Distance between prey and enemy

```
Vector3 enemyPosition =
    gameObject.transform.position;
Vector3 PreyPosition =
    Prey.transform.position;
Vector3 VelocityRelative, Distance,
    PredictedInterceptionPoint;
float timeToClose;
```

```
VelocityRelative = Prey.GetComponent<Rigidbody>().velocity - enemyRigidbody.velocity;
Distance = targetPosition - enemyPosition;
```

- Compute the **time to close**. This is the distance between prey and enemy divided by the relative velocity.

```
timeToClose = Distance.magnitude / VelocityRelative.magnitude;
```

- You can now calculate the **interception point**

```
PredictedInterceptionPoint = targetPosition +
    (timeToClose * Prey.GetComponent<Rigidbody>().velocity );
```

- Finally, you can now calculate the direction and move your enemy.

```
Vector3 direction = PredictedInterceptionPoint – enemyPosition;
direction.Normalize();

enemyRigidbody.velocity = new Vector3(direction.x * ChaseSpeed, enemyRigidbody.velocity.y, direction.z * ChaseSpeed);
```

Time to test it

In the FixedUpdate() method add the just created Intercept() method to the corresponding switch case.

```
switch (behaviour)
{
    ….
    case Behaviour.Intercept:
        Intercept(Prey.transform.position);
        break;
```

In the inspector change the Behaviour to Intercept.

Exercise 3: Pattern Movement

In this exercise, we will make the enemy move between predefined points in your level. Thereby creating the illusion of intelligent behavior.

We are going to create a list of points that make our movement pattern. Create a new Script **WayPoint**.

- Delete the Start and Update method.
- Use Gizmos to visualize your waypoints.
- Use the OnDrawGizmos method to draw a sphere at the waypoint's position.

```
public void OnDrawGizmos(){
    Gizmos.DrawSphere(gameObject.transform.position, 1f);
}
```

Back in your script AdvancedEnemy:

- At the top of your class create a new list for storing the waypoints.

```
[SerializeField]
private List<WayPoint> wayPoints;
```

- Create an integer variable to store the index of the current waypoint.

```
private int currentWayPoint = 0;
```

- Create a float variable distanceThreshold.

```
[SerializeField] private float distanceThreshold;
```

- Create a new function PatternMovement()
- The enemy will move to the current waypoint in a straight line. We can reuse the code defined in ChaseLineOfSight().

```
//Move towards the current waypoint.
ChaseLineOfSight(wayPoints[currentWayPoint].position, NormalSpeed);
```

- When the enemy has reached the waypoint we need to increment the current waypoint. Calculate the distance between the enemy and the waypoint to check if it has reached the waypoint. Rather than checking if the Distance is zero, check if the distance is under a small threshold (distanceThreshold).

- Once the enemy has traveled to all waypoints in the list, it should start at the beginning again. You can use the modulo operator for this.

```
private void PatternMovement(){
  //Move towards the current waypoint.
  ChaseLineOfSight(wayPoints[currentWayPoint].transform.position, NormalSpeed);
  //Check if we are close to the next waypoint and incerement to the next waypoint.
  if(Vector3.Distance(gameObject.transform.position, wayPoints[currentWayPoint].transform.position) < dis-
  tanceThreshold){
    currentWayPoint = (currentWayPoint + 1) % wayPoints.Count; //modulo to restart at the beginning.
  }
}
```

Time to test it

In your FixedUpdate() append the code.

```
switch (behaviour)
{
   …
   case Behaviour.PatternMovement:
     PatternMovement();
     break;
…
}
```

In the inspector change the Behaviour to PatternMovement.

Create some waypoints. Create an empty object, place it in your level and assign the WayPoint script. You can save this object as a prefab. Create a few waypoints and assign them in the inspector to your AdvancedEnemy. Set the DistanceThreshold to e.g. 1.5.

Optional: Change between pattern movement and chasing

To dynamically change between pattern movement and chasing you need to calculate the Distance between enemy and player and switch between your methods accordingly. At the top of AdvancedEnemy, create a new variable ChaseEvadeDistance.

```
[SerializeField]
private float ChaseEvadeDistance;
```

Add:

```
case Behaviour.ChasePatternMovement:
   if(Vector3.Distance(gameObject.transform.position, Prey.transform.position) <
     ChaseEvadeDistance){
     ChaseLineOfSight(Prey.transform.position, ChaseSpeed);
   } else {
     PatternMovement();
   }
   break;
```

In the inspector change the Behaviour to ChasePatternMovement. Set the ChaseEvadeDistance to e.g. 10.

Exercise 4: Hiding from Enemy

So far, the enemy will attack the player based only on the distance between the two.

You will now extend the ChaseLineOfSight method. The enemy will only attack the player if it can see the player.

Use a raycast from the enemy to the player to determine if an object is occluding the player. You can use RaycastHit to determine which collider was hit by the raycast.

Implement a new function PlayerVisible(Vector3 targetPosition) that returns a boolean.

```
private bool PlayerVisible(Vector3 targetPosition)
```

- Calculate the direction vector from enemy to the player
- Create a variable RaycastHit hit;
- Do a raycast from enemy to the player
- Check if the raycast hits the player. If so return true.
- If a different object was hit, return false

```
Vector3 directionToTarget = targetPosition - gameObject.transform.position;
directionToTarget.Normalize();

RaycastHit hit;
Physics.Raycast(gameObject.transform.position, directionToTarget, out hit);

return hit.collider.gameObject.tag.Equals("Player");
```

- In FixedUpdate, add to the switch statement:

```
case Behaviour.Hide:
    if (PlayerVisible(Prey.transform.position))
    {
        ChaseLineOfSight(Prey.transform.position, ChaseSpeed);
    } else {
        PatternMovement();
    }
    break;
```

In the inspector change the Behaviour to Hide.

Technische Universität München
Fakultät für Informatik
Professur für Erweiterte Realität

Boltzmannstr. 3
85748 Garching bei München

https://far.in.tum.de