Lab 8 - AI in Multiplayer Environment

COMP280-Multiplayer Game Programming

Purpose: Using AI in Multiplayer Environment

Contents

1 Introducing AI into a Multiplayer Environment
1.1 Technical requirements
1.2 Setting up the AI system

2 Creating an AI opponent
2.1 Adding the navigation module
2.2 Creating the minion class
2.3 Implementing the minions’ behaviors
2.3.1 Declaring the constructor
2.3.2 Initializing the minion
2.3.3 Handling the delegate functions
2.3.4 Implementing the chase and patrol behaviors
2.3.5 Implementing the Tick() event
2.3.6 Testing the AI opponent

3 Adding opponents to the level
3.1 Creating a spawner class
3.2 Testing the spawner

4 Summary

5 Zip Project and Submit

1. Introducing AI into a Multiplayer Environment

Artificial Intelligence (AI) systems offer an exciting and unique gaming experience for players by providing dynamic challenges that are unpredictable and engaging. This allows developers to create immersive worlds with realistic behavior from non-player characters (NPCs). We will see some of the basics of AI in Unreal Engine. Since we are more interested about multiplayer games, we will not go deep into the details of the system – instead, we'll create simple opponents, which will make our game fully playable from a networked point of view. We will create an enemy Actor that wanders around the level and actively pursues the player’s character once it is detected. This will serve as a starting point for creating more diverse and compelling enemies in the game later. Here are the topics:

- Setting up the AI system
- Creating an AI opponent

- Adding opponents to the level

1.1. Technical requirements

- We are going to use **Unreal Engine 5.4**.
- You have to have finished the **Lab 7**.

1.2. Setting up the AI system

We will not go deep into the Unreal Engine AI system (if you want to make your game enjoyable, it definitely helps to have some know-how on creating a worthy AI opponent). We will do the following:

- **Define NavMesh-like areas**: To make the AI character move around the level, we'll need to define which areas **are allowed** and which **are not** (e.g., we'll need to give the character a **safe place** that the opponents won't dare to step into).
- Then, we'll **create these areas** so that the AI system can manage the minions' walking paths.
- Then, we'll **create undead** (zombies) that walk mindlessly (wander).

Unreal Engine uses a **Navigation System** to let AI Actors navigate a level using **pathfinding algorithms** The Navigation System takes the **collision geometry** in your level and generates a **Navigation Mesh**, which is then split into portions (i.e., polygon geometries) that are used to create a **graph**. This graph is what agents (such as AI characters) use to navigate to their destination. Each portion is given a **cost**, which agents then use to calculate the **most efficient path** (the one with the lowest overall cost).

If you want more information about the Unreal Engine **Navigation System** see the official Epic Games documentation at this link: https://docs.unrealengine.com/5.1/en-US/navigation-system-in-unreal-engine/.

Let's add a **Navigation Mesh** to the level (a process similar to Unity's NavMesh):

1. Open the game level you've been working on so far and, from the **Quickly add to the project button**, select **NavMeshBoundsVolume**. This will add the NavMeshBoundsVolume component to the level and a **RecastNavMesh Actor**.
2. In the Outliner, select NavMeshBoundsVolume and, with the **Scale tool** enabled, resize it so that it covers your desired portion of the level – avoid the **spawn region** for your player characters, as you want to give them a **safe place** to rest or escape to if needed.
3. Hit the **P key** on your keyboard to show the newly created **Navigation Mesh**.

In UE the NavMesh is **green-colored** (as opposed to blue-colored in Unity). It represents the places where the AI character can walk. Notice that **walls** and **doors** create **"holes"** in this mesh, so the AI will be forbidden to step into it. Portions outside of the dungeon are inaccessible – there are no open doors to connect them, so the minions won't be able to reach them.

Let's now create some undead minions and let them walk around the dungeon.

2. Creating an AI opponent

We'll start by creating a class for your hero's **foes**, that can:

- **patrol**, and
- **attack**.

We'll extend the Character class. Tha latter can be controlled also by an **AIController**, allowing independent actions during gameplay. At this point, we want the minion to have the following features:

- A **random patrolling** movement around the level
- A **perception system** that will allow it to **see** and **hear** the player's character
- The ability to **seek out** the player once it has been detected

Later, we'll extend the Character class further by adding some more features such as **health** and **spawnable goodies** (when the AI has been defeated) but for now, we'll just focus on the movement and perception system.

2.1. Adding the navigation module

The first thing to do in order to have an agent that can navigate through a Navigation Mesh is to add the corresponding module to your project. To do this, get back to your programming IDE and open your project build file – the one named **UnrealShadows_LOTL.Build.cs** (or similar, if you opted for a different project name). Locate the following line of code:

```
PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore",
"EnhancedInput" });
```

Change it by adding the NavigationSystem declaration, like this:

```
PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore",
"EnhancedInput", "NavigationSystem" });
```

With the project settings updated, we can start working on the minion AI, by creating a dedicated class.

2.2. Creating the minion class

It's time to create the AI minion class, so create a new class derived from Character and name it US_Minion. Once the class has been created, open the US_Minion.h header file and, in the private section, add the following code:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Minion Perception", meta = (AllowPrivateAccess
= "true"))
TObjectPtr<class UPawnSensingComponent> PawnSense;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Minion Perception", meta = (AllowPrivateAccess
= "true"))
TObjectPtr<class USphereComponent> Collision;
UPROPERTY()
FVector PatrolLocation;
```

The Collision property will be used as a trigger for the AI to grab the character, while PatrolLocation will be used to tell the AI where to go if not chasing the character. The PawnSense property is the declaration for PawnSensingComponent, a component that can be used by the AI character to see and hear pawns around the level (i.e., the player characters). This component is quite straightforward to use and is easily configurable, letting you make the opponent more or less "dumb" during gameplay. You'll get mere info on this in a minute or two when you'll be initializing it. Now it's time to add some properties to the public section. Just add the following code:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Minion AI")
float PatrolSpeed = 150.0f;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Minion AI")
float ChaseSpeed = 350.0f;
```

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category="Minion AI")
float PatrolRadius = 50000.0f;
```

We have defined two movement speed properties: PatrolSpeed will be used when the minion is walking around aimlessly, while ChaseSpeed will be used whenever the minion is seeking the character, in order to make it a new pawn for the Lichlord's army! The PatrolRadius property will be used to find a new location in the level for the minion to inspect. After the properties, you will be declaring the public methods needed for the correct behavior of the AI opponent. Still in the public section, add this block of code to declare them:

```
UFUNCTION(BlueprintCallable, Category="Minion AI")
void SetNextPatrolLocation();
UFUNCTION(BlueprintCallable, Category="Minion AI")
void Chase(APawn* Pawn);
virtual void PostInitializeComponents() override;
FORCEINLINE UPawnSensingComponent* GetPawnSense() const { return PawnSense; }
FORCEINLINE USphereComponent* GetCollision() const { return Collision; }
```

The SetNextPatrolLocation() and Chase() methods will be used to let the AI character move around the scene, looking for a new spot or seeking the player character. The PostInitializeComponent() override will be used to register the character events. Lastly, we are declaring the usual getters for the character components that have been added. The last step in the header declaration is to add the event handlers for this character:

```
UFUNCTION()
void OnPawnDetected(APawn* Pawn);
UFUNCTION()
void OnBeginOverlap(AActor* OverlappedActor, AActor* OtherActor);
```

The first one will manage the minion logic once it has detected a pawn with its senses, while the second one will be used to check whether a player character has been captured. The header has finally been declared – please note that, at the moment, we are not taking into consideration the hearing capabilities of the minion; this is something we are going to implement in the next chapter when our thief hero starts to make some noise!

2.3. Implementing the minions' behaviors

You've declared all your functions and properties, so now it's time to put them to good use by implementing some behaviors for your AI minions. Let's make sure everything is running smoothly and get this project rolling! Open the **US_Minion.cpp** file and add the following include statements at the top:

```
#include "AIController.h"
#include "NavigationSystem.h"
#include "US_Character.h"
#include "Components/CapsuleComponent.h"
#include "GameFramework/CharacterMovementComponent.h"
#include "Perception/PawnSensingComponent.h"
#include "Blueprint/AIBlueprintHelperLibrary.h"
#include "Components/SphereComponent.h"
```

As usual, these lines of code will declare the classes we will use from now on. After you have done that, it's time to implement the constructor by adding the needed components and initializing all the properties.

2.3.1 Declaring the constructor

Once the include statements have been properly declared, you can start by locating the AUS_Minion() constructor and inserting the character initialization. Inside the brackets, just after the PrimaryActorTick.bCanEverTick declaration, add the following code:

```
bUseControllerRotationPitch = false;
bUseControllerRotationYaw = false;
bUseControllerRotationRoll = false;
AutoPossessAI = EAutoPossessAI::PlacedInWorldOrSpawned;
AIControllerClass = AAIController::StaticClass();
PawnSense = CreateDefaultSubobject<UPawnSensingComponent>(TEXT("PawnSense"));
PawnSense->SensingInterval = .8f;
PawnSense->SetPeripheralVisionAngle(45.f);
PawnSense->SightRadius = 1500.f;
PawnSense->HearingThreshold = 400.f;
PawnSense->LOSHearingThreshold = 800.f;
Collision = CreateDefaultSubobject<USphereComponent>(TEXT("Collision"));
Collision->SetSphereRadius(100);
Collision->SetupAttachment(RootComponent);
GetCapsuleComponent()->InitCapsuleSize(60.f, 96.0f);
GetCapsuleComponent()->SetGenerateOverlapEvents(true);
GetMesh()->SetRelativeLocation(FVector(0.f, 0.f, -91.f));
static ConstructorHelpers::FObjectFinder<USkeletalMesh>
SkeletalMeshAsset(TEXT("/Game/KayKit/Skeletons/skeleton_minion"));
if (SkeletalMeshAsset.Succeeded())
{
 GetMesh()->SetSkeletalMesh(SkeletalMeshAsset.Object);
}
GetCharacterMovement()->bOrientRotationToMovement = true;
GetCharacterMovement()->RotationRate = FRotator(0.0f, 500.0f, 0.0f);
GetCharacterMovement()->MaxWalkSpeed = 200.f;
GetCharacterMovement()->MinAnalogWalkSpeed = 20.f;
GetCharacterMovement()->BrakingDecelerationWalking = 2000.f;
```

You'll already be familiar with most of the code from the thief character creation, but you will see some noticeable additions. First of all, we are setting the AutoPossessAI property, which lets us define whether the game system will possess the AI character once in the level – we want it to be both in full control when it is spawned at runtime and when it is already in the level when the game starts, so we have opted for a value of PlacedInWorldOrSpawned. Then, we define which controller will be used for the AI system by setting the AIControllerClass property; in this case, we are just using the base AAIController class, but you can obviously implement your own with additional features. The last notable thing is the PawnSense component creation – as you can see, we are initializing the properties that will make the minion see and hear at a certain distance. You should take note of the SensingInterval initialization, which will let us tweak how much time will pass between two sense perceptions. This will make the difference between a very reactive character (i.e., a lower value) or a really dumb one (i.e., a higher one).

2.3.2 Initializing the minion

It's now time to initialize the character when it is added to the game. As you already know, this is usually done from the BeginPlay() method. So, just after the **Super::BeginPlay()** declaration, add the following:

```
SetNextPatrolLocation();
```

This call will simply start the patrolling behavior. Then, add the PostInitializeComponents() implementation by adding this code to the file:

```
void AUS_Minion::PostInitializeComponents()
{
  Super::PostInitializeComponents();
  if(GetLocalRole() != ROLE_Authority) return;
  OnActorBeginOverlap.AddDynamic(this, &AUS_Minion::OnBeginOverlap);
  GetPawnSense()->OnSeePawn.AddDynamic(this, &AUS_Minion::OnPawnDetected);
}
```

As you can see, we are using two delegates to react to an Actor overlap, for checking whether we have reached the player character, and to handle the pawn perception to check whether we can see the player character. Also, notice that they are initialized only if the role of this object is authoritative (i.e., the method is being executed on the server). The next step is to implement these two delegate functions in order to manage the aforementioned events.

2.3.3 Handling the delegate functions

Whenever the minion detects a pawn, it will immediately check whether it is a character and, if the result is successful, it will start chasing it. Let's add the method to handle the delegate in the source file:

```
void AUS_Minion::OnPawnDetected(APawn* Pawn)
{
  if (!Pawn->IsA<aus_character>()) return;
  GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Red, TEXT("Character detected!"));
  if (GetCharacterMovement()->MaxWalkSpeed != ChaseSpeed)
  {
    Chase(Pawn);
  }
}
```

The code here is quite straightforward – we just have added a debug message stating that a character has been detected. The second delegate we need to handle is the overlap, so add the following method implementation:

```
void AUS_Minion::OnBeginOverlap(AActor* OverlappedActor, AActor* OtherActor)
{
  if (!OtherActor->IsA<AUS_Character>()) return;
  GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Yellow, TEXT("Character captured!"));
}
```

As you can see, we check again whether we have found a character, and after that, we simply display a debug message – it looks like our hero got a little too close to the minions and now they've roped them into joining the Lichlord's undead army! Later on, you'll implement a respawn system to let the player restart the game with a brand-new character. The next step will be the actual AI's movement through the Navigation Mesh, both for the patrol and chase behaviors.

2.3.4 Implementing the chase and patrol behaviors

It's now time to start implementing the movement control for your AI character – specifically, you'll be implementing the SetNextPatrolLocation() function, which will find a new reachable point for the minion, and the Chase() function, which will send the minion on a "seek and destroy" mission toward the character. To do so, add the following code to the file:

```
void AUS_Minion::SetNextPatrolLocation()
{
  if(GetLocalRole() != ROLE_Authority) return;
  GetCharacterMovement()->MaxWalkSpeed = PatrolSpeed;
  const auto LocationFound = UNavigationSystemV1::K2_GetRandomReachablePointInRadius(
     this, GetActorLocation(), PatrolLocation, PatrolRadius);
  if(LocationFound)
  {
    UAIBlueprintHelperLibrary::SimpleMoveToLocation(GetController(), PatrolLocation);
  }
}
void AUS_Minion::Chase(APawn* Pawn)
{
  if(GetLocalRole() != ROLE_Authority) return;
  GetCharacterMovement()->MaxWalkSpeed = ChaseSpeed;
  UAIBlueprintHelperLibrary::SimpleMoveToActor(GetController(), Pawn);
  DrawDebugSphere(GetWorld(), Pawn->GetActorLocation(), 25.f, 12, FColor::Red, true, 10.f, 0, 2.f);
}
```

The first function sets the character speed to the patrolling value and uses the UNavigationSystemV1::K2_GetRandomReachablePointInRadius() method to find a reachable point in the Navigation Mesh. Then, the AI is simply commanded to reach that location. The second function does something similar, but the target point will be the character – after all, it is on a mission from the Lichlord to get as many soon-to-be undead heroes as possible!

2.3.5 Implementing the Tick() event

The last thing you need to implement in order to make the patrolling system fully operational is to check whether the AI character has reached its destination; in this case, it will just have to find another point in the Navigation Mesh. As we need to continuously check the distance between the AI and the target point, the best place to write the code is within the Tick() event. Let's find the method and, just after the Super::Tick(DeltaTime) call, add this piece of code:

```
if(GetLocalRole() != ROLE_Authority) return;
if(GetMovementComponent()->GetMaxSpeed() == ChaseSpeed) return;
if((GetActorLocation() - PatrolLocation).Size() < 500.f)
{
  SetNextPatrolLocation();
}
```

As you can see, the first line checks whether the character is patrolling (i.e., the maximum speed should not equal the chase speed). Then, we are checking that we are near enough (about half a meter) to the patrol location in order to look for another reachable point.

2.3.6 Testing the AI opponent

Now that the enemy AI has been created, you can test it out in the game level. To do this, open the Unreal Engine Editor and, from the Content Browser, drag an instance of the US_Minion class (located in the C++ Classes | UnrealShadows_LOTL folder) into the level. The gizmos around the character represent the PawnSense component – its sight and hearing capabilities. The sight area is represented by a green cone that shows how wide and far the AI can see. The hearing sense is represented by two spheres – a yellow one that shows how far the AI will hear a noise if it is not obstructed by any obstacle, and a cyan one that shows how far the AI will sense noise, even if generated behind an obstacle, such as a wall. Go into play mode and the opponent should start wandering around the level and over the Navigation Mesh. Whenever a player character enters the minion's line of sight (i.e., the green cone), the enemy will react and start chasing the player at a higher speed. Once the character has been reached, you will notice that the minion will stop moving – its mission has been completed and it can rest! As an extra exercise, you may want to add a timer that will check whether the AI is staying still for too long; in that case, it will restart its patrolling system by looking for a new reachable location. So, in this section, you have created your AI opponent, ready to roam around the dungeon, seeking its next victim. You have created a simple but effective patrolling system and added a perception sense to the AI so that it can intercept the player when they are not stealthy enough. In the next section, you'll create a spawning system in order to add minions as the game progresses and make things more challenging for the players.

3. Adding opponents to the level

Now that you have an opponent for your thief hero, it is time to let the system spawn a bunch of them at runtime. You'll be doing this by implementing a spawn system similar to the one used in Chapter 3, Testing the Multiplayer System with a Project Prototype – this time, you'll create the spawner in C++ instead. What we want to implement here is an Actor that will have the following features:

- Spawns a few minions at the start of the game.
- Spawns new minions at predefined intervals.
- Spawns the minions in a selected area.
- Randomly selects a minion type every time it spawns. At the moment, we have just one minion type, but in the following chapters, we will add more variations.

Let's get started.

3.1. Creating a spawner class

Start by creating a C++ class that extends from Actor and name it US_MinionSpawner. Once created, open the .h file and, in the private section, add the following declarations:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = "Spawn System", meta = (AllowPrivateAccess =
"true"))
TObjectPtr<class uboxcomponent=""> SpawnArea;
UPROPERTY()
FTimerHandle SpawnTimerHandle;
```

You should be already familiar with the first declaration from Chapter 3, Testing the Multiplayer System with a Project Prototype – we are declaring an area that will be used to randomize the spawned minion location. The second declaration will be used to store a reference of the timer handler used by the spawner to generate new minions at predefined intervals. Now we are going to declare some properties that will make this class customizable in the level. In the public section, add the following property declarations:

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category="Spawn System")
TArray<tsubclassof<class aus_minion="">> SpawnableMinions;
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category="Spawn System")
float SpawnDelay = 10.0f;
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category="Spawn System")
int32 NumMinionsAtStart = 5;
```

The first property will expose an array that will contain all the spawnable minion types. As stated before, at the moment, we have just one type, but we will add some more later on. The other two properties are self-explanatory, letting us define the spawn timing and how many minions should already be in the level when the game starts. The last step is to add a Spawn() method in the protected section of the header:

```
UFUNCTION()
void Spawn();
```

The header is now finished. Now, let's switch to the .cpp and implement some code logic. Implementing the spawner logic It's time to implement the spawner features. To do so, open the .cpp file, find the constructor, and add the required includes at the top of the file:

```
#include "US_Minion.h"
#include "Components/BoxComponent.h"
```

Then, add the following piece of code:

```
SpawnArea = CreateDefaultSubobject<UBoxComponent>(TEXT("Spawn Area"));
SpawnArea->SetupAttachment(RootComponent);
SpawnArea->SetBoxExtent(FVector(1000.0f, 1000.0f, 100.0f));
```

You are already well versed in creating components, so let's dive right into the BeginPlay() method and add this code just after the Super::BeginPlay() declaration:

```
if(SpawnableMinions.IsEmpty()) return;
if(GetLocalRole() != ROLE_Authority) return;
for (int32 i = 0; i < NumMinionsAtStart; i++)
{
  Spawn();
}
GetWorldTimerManager().SetTimer(SpawnTimerHandle, this, &AUS_MinionSpawner::Spawn, SpawnDelay, true, 
SpawnDelay);
```

First of all, we are checking that there is at least one spawnable minion type – if the array is empty, there is no need to go on with the code. Then, we check that the Actor has the authority to spawn something; as usual, we want the server to be in full control of what's happening. After that, we call the Spawn() function in a loop, in order to create a starting pool of enemies. The last step is to create a timer, which will call the Spawn() function at an interval defined by the SpawnDelay value. The last thing to do to have the spawner fully functional is to add the Spawn() function implementation. Let's add it at the end of the file:

```
void AUS_MinionSpawner::Spawn()
{
  FActorSpawnParameters SpawnParams;
  SpawnParams.SpawnCollisionHandlingOverride =
  ESpawnActorCollisionHandlingMethod::AdjustIfPossibleButDont SpawnIfColliding;
  auto Minion =
  SpawnableMinions[FMath::RandRange(0, SpawnableMinions.Num() - 1)];
  const auto Rotation =
  FRotator(0.0f, FMath::RandRange(0.0f, 360.0f), 0.0f);
  const auto Location = SpawnArea->GetComponentLocation() +
    FVector(FMath::RandRange(-SpawnArea->GetScaledBoxExtent().X,SpawnArea->GetScaledBoxExtent().X),
      FMath::RandRange(-SpawnArea->GetScaledBoxExtent().Y, SpawnArea->GetScaledBoxExtent().Y), 0.0f);
  GetWorld()->SpawnActor<AUS_Minion>(Minion, Location, Rotation, SpawnParams);
}
```

As long as it may seem, this code is quite straightforward, and you have already done something similar at the start of this book (do you remember the falling fruits?). We are just taking a random minion type from the array, retrieving a random location in the spawn area, and then we are going to spawn the minion at that location. The only thing worth mentioning is SpawnCollisionHandlingOverride, which is set to spawn the Actor, avoiding any collision with other objects in the level. As an extra exercise, you may add a limit to the number of minions that will be spawned from a single spawner object. This will avoid overcrowding your level and making the game unplayable for your players! The spawn Actor is ready, so it is time to **compile** your project and do some proper **testing**.

3.2. Testing the spawner

- Locate the **US_MinionSpawner** class (found inside the **C++ Classes → UnrealShadows_LOTL_{YI} folder)**
- **Drag it into your level to create an instance of it.**
- **Position the Actor in a suitable place,**

- **Resize the **Box Extent** parameters to set a nice size for the minions to be located within. In my case, I opted to place the spawner in the room labeled **SP3** with the Box Extent property set to **(900, 400, 100)**.
- Then, with the Actor still selected, do the following:
    1. Locate the **Spawn System** category in the Details panel.
    2. Add an element to the **Spawnable Minions** array, which will be labeled as **Index[0]**. From the corresponding drop-down menu, select **US_Minion**.
    3. Tweak **Spawn Delay** and **Num Minions at Start** to suit your needs; in my case, I have left the default values
- Add as many spawner Actors as you feel necessary to balance your game level.
- Play. Notice:
    - the undead minions shall spawn ,
    - their replication and synchronization across all clients should work
- $Take\,Snapshot$

In this lab, we created a fully customizable spawn system that can be used and tweaked for any level in your game.

4. Summary

Here, we dealt with to the basics of AI in a multiplayer game. We:

- Created a Navigation System that will let your AI move independently around the level.
- Then, we created a base minion, which patrols around seeking the player characters, changing its behavior to a more aggressive stance once it finds them.
- Then, we added spawn points around the dungeon in order to populate the area with worthy opponents.

The take away from this lab is that, with your previously acquired knowledge, everything is correctly synchronized over the network. Learning things from the start could be a huge advantage in the future! Putting in the effort and really grasping the basics pays off with the project!

In the next lab, we'll keep on exploring some of the possibilities of implementing worthy opponents for the hero – we will give it a sense of **hearing** and **a health system** in order to make it more engaging and, at the very least, defeatable.

5. Zip Project and Submit

- Click **File → Zip Project**
- Keep the default folder, select the name **US_LTOL_{YourInitials}_Lab8.zip**
- When the zipping finishes click on the provided link **Show in Explorer**
- Upload the zip in the corresponding folder (Lab8).