GD L7 F-L 1:

1. Time Manipulation

Key Concepts:

- **`Time.deltaTime`:** Ensures frame-rate-independent movements by providing the time (in seconds) since the last frame.
- **`Time.timeScale`:** Controls the speed of time in the game:
  - `1`: Normal speed.
  - `< 1`: Slows the game (e.g., `0.5` for slow motion).
  - `0`: Pauses the game.

Practical Usage:

- Use `Invoke("GamePause", 5);` to pause the game after 5 seconds by setting `Time.timeScale = 0;`.

---

2. Enemy Movement

Objective:

- Make enemies dynamically follow the player within a specific range.

Steps:

1. **Find the Player:**
   - Use `GameObject.FindGameObjectWithTag("Player")` to reference the player dynamically.
2. **Calculate Distance:**
   - `Vector3.Distance` computes the distance between two objects.
3. **Move Towards Player:**
   - Use `Vector3.MoveTowards` to move the enemy at a specific speed, only if the player is within a 10-unit range.
4. **Collision Detection:**
   - `OnCollisionEnter` triggers the Game Over state when the enemy collides with the player.

Code Example:

EnemyController

```
public class EnemyController : MonoBehaviour
{
    public GameObject playerObj;
    LectureNew7 obj7;

    void Start()
    {
        obj7 = GameObject.FindAnyObjectByType<LectureNew7>();
        playerObj = GameObject.FindGameObjectWithTag("Player");
    }

    void Update()
    {
        if (Vector3.Distance(this.transform.position,
playerObj.transform.position) < 10f)
        {
            this.transform.position =
Vector3.MoveTowards(this.transform.position,
playerObj.transform.position, 1f * Time.deltaTime);
        }
    }

    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            obj7.GameOver();
        }
    }
}
```

Explanation:

- **Start()**:
  - Finds the player and the `LectureNew7` script for calling `GameOver`.
- **Update()**:
  - Checks if the player is within a 10-unit range.
  - Moves the enemy towards the player using `Vector3.MoveTowards`.
- **OnCollisionEnter()**:
  - Detects collision with the player and triggers the `GameOver()` function.

---

3. Object Interaction: Click to Destroy

Key Concepts:

- **Raycasting:** Cast an invisible ray to detect objects.
- **Mouse Input:** Use `Input.GetMouseButtonDown(0)` to detect left mouse clicks.

Steps:

1. Generate a ray using `Camera.main.ScreenPointToRay(Input.mousePosition)`.
2. Check if the ray intersects a collider using `Physics.Raycast`.
3. Verify the object tag and destroy it using `Destroy()`.

Code Example: Raycasting in `LectureNew7`

```
public class LectureNew7 : MonoBehaviour
{
    RaycastHit hit;
    public Text scoreBox;
    public GameObject GameOverPanel;
    int scoreNumb = 0;

    private void Update()
    {
        scoreBox.text = "Score: " + scoreNumb;

        if (Input.GetMouseButtonDown(0))
        {
            Ray ray =
Camera.main.ScreenPointToRay(Input.mousePosition);
```

```
            if (Physics.Raycast(ray, out hit))
            {
                if (hit.transform.gameObject.CompareTag("Enemy"))
                {
                    scoreNumb++;
                    Destroy(hit.collider.gameObject);
                }
            }
        }
    }
}
```

Explanation:

- **Update():**
  - Updates the score display every frame.
  - Detects left mouse clicks (`Input.GetMouseButtonDown(0)`).
  - Casts a ray from the camera to the mouse position and checks for collisions.
  - If the clicked object is tagged as "Enemy," it increments the score and destroys the enemy.

---

4. Basic UI Implementation

Key Concepts:

- **Unity UI System:** Used for interactive elements like buttons, text, and panels.
- **Components:**
  - **Canvas:** Container for UI elements.
  - **Event System:** Manages interactions with UI elements.
  - **Text:** Displays textual information.
  - **Panel:** A container for organizing UI elements.

Steps:

1. Create a `Canvas` and add UI elements like `Text` and `Panel`.
2. Use `SetActive(true)` to show the Game Over panel.
3. Update the score display dynamically.

Code Example: UI in

LectureNew7

```
public void GameOver()
{
    GameOverPanel.SetActive(true); // Display the Game Over panel
    scoreBox.gameObject.GetComponent<Text>().color = Color.red; //
Change score text color to red
    Time.timeScale = 0; // Pause the game
}
```

Explanation:

- **GameOver():**
  - Activates the `GameOverPanel` to display "Game Over."
  - Change the score text color to red for emphasis.
  - Pauses the game using `Time.timeScale = 0`.
  - GameOver panel is initialised as a public GameObject because we are using the SetActive function and it must be a gameobject for that.

Complete Flow: Code Integration

Game Loop:

1. **Enemy Behavior:**
   - `EnemyController` script handles movement and collision detection.
2. **Player Interaction:**
   - `LectureNew7` script manages score updates and raycasting for destroying enemies.
3. **Game Over State:**
   - Triggered by collisions or specific game conditions.
   - Displays the Game Over panel and pauses the game.

Key Functions:

- **Raycasting:** Interactive object destruction.
- **Vector3.Distance & MoveTowards:** Smooth enemy movement.
- **UI Management:** Dynamic score updates and Game Over handling.

GD L8 F-L 2:

Game Development Lecture Notes

Scene Management

- **Types of Scene Management**:
  - **Single Scene**:
    - Suitable for hyper-casual games with simple logic and one level.
    - Difficulty increases within the same scene.
  - **Multi-Scene**:
    - Used for games with multiple levels or environments, where each level is a separate scene.
- **Open World Games**:
  - Environments are divided into chunks, loaded as separate scenes at runtime to optimize performance.
- **Loading Screens**:
  - Used to mask scene loading and improve user experience.
- **Unity's Scene Management API**:
  - Provides developers with control over scene transitions and management.

Code Example for Scene Switching:

```
using UnityEngine.SceneManagement;
using UnityEngine;

public class LevelManager : MonoBehaviour
{
    public void NextButton()
    {
        if (SceneManager.GetActiveScene().buildIndex < 2)
        {

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1,
LoadSceneMode.Single);
        }
        else
        {
            SceneManager.LoadScene(0, LoadSceneMode.Single);
        }
```

```
    }
}
```

Explanation:

- **NextButton()**:
    - Checks the current scene's build index.
    - If the current scene is not the last one, it loads the next scene.
    - If the current scene is the last one, it loops back to the first scene (index 0).
    - `LoadSceneMode.Single` replaces the current scene with the new one.

Unity Scene Management: Load Scene Modes and Best Practices

Unity's scene management system offers two primary modes for loading scenes, each with distinct use cases:

---

1. Single Mode

- **Definition**: In Single Mode, the current scene is completely replaced by the new scene. The original scene is unloaded, and all its resources are released before the new scene loads.
- **Use Case**: Ideal for scenarios like starting a fresh game level where the previous level's data isn't needed.
- **Key Point**: This is the standard mode for scene loading.

---

2. Additive Mode

- **Definition**: Additive Mode allows multiple scenes to coexist simultaneously. The new scene is loaded and added to the currently active scene without unloading the original one.
- **Use Cases**:
    - Maintaining a **persistent game world** while loading new areas or levels.
    - Keeping **UI elements** active while transitioning to a different gameplay scene.
- **Advantages**:
    - Ensures smoother transitions.
    - Enables modular scene management for larger, more complex games.

Performance Considerations

- **Freezing Issues**:
    - Loading large scenes instantly in **Single Mode** (without a loading screen) can cause the game to freeze momentarily, leading to potential "not responding" errors.
    - This can negatively impact the game's discoverability on app stores due to poor user experience.
- **Recommended Solution**:
    - Use a **fake loading screen** to provide visual feedback while the new scene loads in the background.
    - This approach creates the illusion of continuous gameplay and mitigates performance issues.

Unloading Scenes

- **Definition**: Unloading a scene temporarily removes it from memory but retains its state.
- **Comparison**:
    - Different from **closing a scene**, which completely removes it and requires reloading from scratch.
- **Use Case**: Suitable for scenarios where you may need to revisit the scene later without losing its current state.

Choosing the Right Mode

The selection of Single Mode, Additive Mode, or Unloading depends on:

1. The specific requirements of your game.
2. The desired user experience.
3. Performance optimization goals.

Raycasting and Scoring System

Raycasting:

- **Definition**:
  - A technique used to detect objects by casting an invisible ray from the camera or an object.
- **Example Use Case**:
  - Shooting mechanics in games, where a player clicks to "shoot" enemies.

Code Example for Raycasting and Scoring:

```
using System.Collections;
using UnityEngine;
using UnityEngine.UI;

public class LectureNew8 : MonoBehaviour
{
    RaycastHit hit;
    public Text scoreBox;
    public GameObject GameOverPanel;
    int scoreNumb = 0;
    public Text HighScore;

    void Start()
    {
        HighScore.text = "Highscore: " +
PlayerPrefs.GetInt("highscore");
    }

    void Update()
    {
        scoreBox.text = "Score: " + scoreNumb;

        if (Input.GetMouseButtonDown(0))
        {
            Ray ray =
Camera.main.ScreenPointToRay(Input.mousePosition);
            if (Physics.Raycast(ray, out hit))
            {
                if (hit.transform.gameObject.CompareTag("Enemy"))
                {
```

```
                    scoreNumb++;
                    staticnew.score2++;
                    Destroy(hit.collider.gameObject);
                }
            }
        }
    }

    public void GameOver()
    {
        if (PlayerPrefs.GetInt("highscore") < scoreNumb)
        {
            PlayerPrefs.SetInt("highscore", scoreNumb);
        }
        GameOverPanel.SetActive(true);
        scoreBox.color = Color.red;
        Time.timeScale = 0;
    }
}
```

Explanation:

- **Raycasting in Update()**:
  - Checks if the player clicks the left mouse button (`Input.GetMouseButtonDown(0)`).
  - Casts a ray from the camera to the mouse position (`Camera.main.ScreenPointToRay`).
  - Detects if the ray hits an object tagged as "Enemy".
  - Increments the score and destroys the enemy game object.
- **Scoring System**:
  - Uses a `scoreNumb` variable to keep track of the current score.
  - Displays the score in a UI `Text` element (`scoreBox`).
- **Game Over**:
  - Compares the current score with the stored high score in `PlayerPrefs`.
  - Updates the high score if the current score is higher.
  - Activates the "Game Over" UI panel and stops the game with `Time.timeScale = 0`.

What are PlayerPrefs?

PlayerPrefs is a simple data storage system in Unity that allows developers to save and load basic game data across play sessions. It helps retain player progress even after the game is closed.

How to Use PlayerPrefs

1. **Saving Data**:
    - Use `PlayerPrefs.SetInt`, `PlayerPrefs.SetFloat`, or `PlayerPrefs.SetString` to save data.
    - Requires a **key** (unique identifier) and a **value** to store.

Example:
```
PlayerPrefs.SetInt("highscore", scoreNumb);
```

2. **Retrieving Data**:
    - Use `PlayerPrefs.GetInt`, `PlayerPrefs.GetFloat`, or `PlayerPrefs.GetString` to retrieve saved data.
    - Provide the **key** to fetch the associated value.

Example:
```
int highScore = PlayerPrefs.GetInt("highscore");
```

Key Features of PlayerPrefs

- **Supported Data Types**:
    - PlayerPrefs can store **integers**, **floats**, and **strings** only.
    - Not suitable for saving complex data structures.
- **Granular Control**:
    - Effective for saving individual values, but managing a large volume of data can be challenging.
    - Use structures or combine PlayerPrefs values for better data organization in such cases.
- **Global Accessibility**:

- Any script in your Unity project can access and modify PlayerPrefs data.

Example: Saving and Displaying High Scores

1. **Saving the High Score**:
   - At the end of the game, compare the current score with the stored high score.

If the current score is higher, update the "highscore" PlayerPref:

```
if (scoreNumb > PlayerPrefs.GetInt("highscore", 0)) {

    PlayerPrefs.SetInt("highscore", scoreNumb);

}
```

2. **Displaying the High Score**:

At the start of the game, retrieve the high score and display it in the UI:
`highScoreText.text = PlayerPrefs.GetInt("highscore", 0).ToString();`

Limitations of PlayerPrefs

1. **Limited Data Types**:
   - Cannot handle complex objects like arrays, lists, or dictionaries.
2. **Scalability Issues**:
   - Not ideal for large-scale games or complex data storage needs.

3. **Alternatives for Advanced Use**:
    - For extensive games (e.g., open-world RPGs), consider advanced data management techniques like:
        - **Data Serialization** (e.g., JSON, XML).
        - **Databases** (e.g., SQLite).

Conclusion

PlayerPrefs is a simple and effective tool for small-scale data persistence, such as saving high scores or basic settings. For complex game projects, it is recommended to explore more robust solutions.

1. What are Static Classes?

A **static class** in C# is a special class that:

- Cannot be instantiated (i.e., no objects can be created using the `new` keyword).
- Serves as a container for **static members** (variables and methods) that are globally accessible without needing an instance of the class.

Syntax Example:

```
public static class StaticClass

{

    // Static members go here

}
```

---

2. Characteristics of Static Classes

- **Non-instantiable**: Objects cannot be created from static classes.
- **Inheritance Restriction**: Static classes cannot inherit from or be inherited by other classes.
- **Static Members Only**: All members (variables and functions) must be declared as `static`.

---

3. What are Static Members?

Static members (variables and methods) are associated with the class itself, not with any object instance.

Key Features:

Global Accessibility: Can be accessed directly using the class name.
Example:
```
StaticClass.SomeStaticVariable;
```

- **Single Instance**: Only one copy of a static member exists in memory, shared across the application.
- **Persistence**: Static members retain their values throughout the application's lifecycle.

---

4. Why Use Static Classes and Members?

1. **Centralized Data and Functions**:
    - Provide a unified location for globally accessible data or methods.
2. **Mathematical Calculations**:
    - Ideal for grouping constants or utility functions that don't depend on object states.
3. **Game-Wide Settings**:
    - Useful for global configurations like sound settings, difficulty levels, or player preferences.

---

5. How Are They Used?

Accessing Static Members:

```
int retrievedValue = StaticClass.SomeStaticVariable;
```

-

Calling Static Methods:

```
StaticClass.SomeStaticFunction();
```

Static Members in Non-Static Classes:
Static members can exist within non-static classes, retaining their global accessibility and single-instance behavior.
Example:

```
public class NonStaticClass

{

    public static int staticMember;

    public int instanceMember;

}
```

---

6. Considerations and Limitations

1. **Overuse**:
    - Over-relying on static classes can lead to tight coupling, reducing code flexibility and maintainability.
2. **Unity's Recommendation**:
    - Use static classes for utility functions and constants.
    - For more complex game logic, adopt object-oriented programming with instance-based classes.

---

7. Alternatives for Complex Data

For large-scale or complex games (e.g., RPGs), static classes may not be sufficient. Instead, consider:

- **Structures**:
  Group related variables into custom data types for better organization.
- **Data Serialization**:
  Convert data into formats like JSON or XML for efficient storage and retrieval.
- **Databases**:
  Use databases for managing large datasets efficiently.

Conclusion

Static classes and members provide a simple and effective way to manage globally accessible data and utility functions in Unity. However, their limitations make them less suitable for complex projects. For scalable and maintainable code, explore alternative solutions like data serialization and databases.

Key Takeaways

1. **Scene Management**: Use `SceneManager` methods to switch scenes dynamically.
2. **Raycasting**: Essential for shooting mechanics and interaction detection.
3. **Scoring**: Manage scores with variables and display them using UI elements.
4. **Game Over**: Use `PlayerPrefs` to store high scores and manage game state.
5. **Static Members**: Helpful for sharing data across the game.

GD L9 F-L 3: (Arranged Class Notes)

Introduction to Animations in Unity

Unity provides a robust animation system for creating animations for objects and characters. Animations in Unity can be recorded and saved as clips, which can then be reused without consuming CPU resources since the calculations are pre-recorded. This system is efficient compared to using functions for animations, which require runtime calculations.

Key Points:

1. **Using Unity's Animation Component:**
    - Pre-recorded calculations ensure no CPU overhead.
    - Efficient for simple animations such as moving platforms or aesthetic changes.
2. **Recording and Reusing Animations:**
    - Animations can be saved as clips.
    - Clips are reusable and efficient for both simple and complex objects.
3. **Recommendations for Complex Animations:**
    - Use Unity for simple animations.
    - For complex animations (e.g., humanoid characters), create models and animations in 3D modeling software like Blender. Blender and Unity concepts for animation are similar, enabling seamless switching.

Steps to Create Animations in Unity:

1. **Enable the Animation Tab:**
    - Navigate to **Window > Animation**.
    - Unity provides three windows:
        - **Animation** (independent window for creating animations).
        - **Animator** and **Animator Parameters** (used for managing and controlling animations).
2. **Creating an Animation Clip:**
    - Select the object in the scene.
    - Open the **Animation Window**.
    - Click **Create** and name the clip.
    - A timeline will appear for recording animations.
3. **Recording Animation:**
    - In the **Animation Window**, click the **Record** button.
    - Move the timeline cursor to the desired time.

    - Modify object properties (position, rotation, scale, etc.).
    - Keyframes are automatically added to the timeline.
    - Click **Stop Recording** and preview the animation.
4. **Using Animator Component:**
    - An **Animator Component** is automatically added to the object.
    - The Animator manages applied animations.

Creating Multiple Animation Clips:

- Each clip is independent of others.
- Use the **Animator** to manage the order of playing clips using the concept of a Finite State Machine (FSM).
- **Default State:** The animation state active when the game starts (indicated by orange color).

Animator Transitions and Conditions:

1. **Making Transitions:**
    - Right-click on the default state and select **Make Transition**.
    - Connect transitions between states.
2. **Adding Conditions:**
    - Open the **Animator Parameters** tab.
    - Add parameters (**Float, Integer, Boolean, Trigger**).
    - Use the added parameters to define conditions for transitions.
3. **Understanding Has Exit Time:**
    - **Checked:** Transition occurs only after the first animation completes.
    - **Unchecked:** Immediate transition to the next animation.

Controlling Animations via Code:

1. **Setting Triggers:**
   Anim.SetTrigger("Switch");
2. **Boolean Parameters:**
    - **SetBool():** Used to toggle animation states.
    - Example: Control walking and idle states based on conditions.

Concepts of Root Motion vs. Non-Root Motion:

1. **Root Motion:**
    - Animation ignores recorded coordinates.
    - Used for rigid body objects or humanoid animations.
    - Requires the **Apply Root Motion** checkbox in the Animator Component.
2. **Non-Root Motion:**
    - Animation plays at recorded coordinates.
    - Ideal for static or non-rigid objects.

Mixamo Integration:

1. **Downloading Assets:**
    - Characters: Download with or without skin as Unity FBX.
    - Animations: Download as Unity FBX without skin.
2. **Importing in Unity:**
    - Drag and drop the files into the **Assets** folder.
    - Configure settings under the **Rig Tab**:
        - Set **Animation Type** to Humanoid.
        - Select **Create From This Model** under Avatar Definition.
        - Extract materials and textures.
3. **Animator Controller Setup:**
    - Create an Animator Controller in the Assets folder.
    - Assign animations to states (e.g., Idle, Walk).
    - Manage transitions using FSM and parameters.

Blend Trees:

- Blend multiple animations based on float parameters.
- Example: Blend walking and running animations using a **Speed** parameter.
- Allows smooth transitions between animations.

Sample Codes Explanation:

PlayerController.cs:

- Switch animation triggered via space bar.

Example:
```
if (Input.GetKeyDown(KeyCode.Space))
{
   Anim.SetTrigger("Switch");
```

- }

NinjaController11.cs (NPC):

- Uses NavMeshAgent for navigation.
- Animates walking and idle states based on distance to the destination.

Example:
```
if (Vector3.Distance(this.transform.position, Destination.position) < 1.5f)
{
   Agent.SetDestination(this.transform.position);
   Anim.SetBool("IsWalk", false);
}
else
{
   Agent.SetDestination(Destination.position);
   Anim.SetBool("IsWalk", true);
```

- }

NinjaController11.cs (Player):

- Controls walking animations based on key presses (Up Arrow).

Example:
```
if (Input.GetKeyDown(KeyCode.UpArrow))
{
   Anim.SetBool("IsWalk", true);
```

- }

Recommended Practices:

- Use Unity's animation system for simple objects and Blender for complex models.
- Leverage Mixamo for quick humanoid animations.
- Prefer non-root motion for controlled movement animations like walking.
- Utilize Animator Parameters to manage transitions dynamically via code.

PlayerController.cs

```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public Animator Anim;

    // Start is called before the first frame update
    void Start()
    {
        if (Anim == null)
        {
            Anim = gameObject.GetComponent<Animator>();
        }
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            Anim.SetTrigger("Switch");
        }
    }

    // References
    public void MessageTest(string message)
    {
```

```
        // Debug.Log("Message Received: " + message);
        // Destroy(this.gameObject, 1f);

        // Commented out for future use
    }
}
```

NinjaController11 (Version 1).cs

```
using UnityEngine;
using UnityEngine.AI;

public class NinjaController11 : MonoBehaviour
{
    public bool IsNPC = true;
    public bool IsPlayer = false;
    public Transform Destination;
    public Animator Anim;
    public NavMeshAgent Agent;

    // Start is called before the first frame update
    void Start()
    {
        Anim = GetComponent<Animator>();
        Agent = GetComponent<NavMeshAgent>();
    }

    // Update is called once per frame
    void Update()
    {
        if (Destination != null && IsNPC)
        {
            if (Vector3.Distance(this.transform.position,
Destination.position) < 1.5f)
            {
                Agent.SetDestination(this.transform.position);
                Anim.SetBool("IsWalk", false);
            }
```

```
            else
            {
                Agent.SetDestination(Destination.position);
                Anim.SetBool("IsWalk", true);
            }
        }

        if (Input.GetKeyDown(KeyCode.UpArrow) && IsPlayer)
        {
            Anim.SetBool("IsWalk", true);
        }

        if (Input.GetKeyUp(KeyCode.UpArrow) && IsPlayer)
        {
            Anim.SetBool("IsWalk", false);
        }
    }
}
```

NinjaController11 (Version 2).cs

```
using UnityEngine;
using UnityEngine.AI;

public class NinjaController11 : MonoBehaviour
{
    public bool IsNPC = true;
    public Transform Destination;
    public Animator Anim;
    public NavMeshAgent Agent;

    // Start is called before the first frame update
    void Start()
    {
        Anim = GetComponent<Animator>();
        Agent = GetComponent<NavMeshAgent>();
    }
```

```
    // Update is called once per frame
    void Update()
    {
        if (Destination != null)
        {
            if (Vector3.Distance(this.transform.position,
Destination.position) < 1.5f)
            {
                Agent.SetDestination(this.transform.position);
                Anim.SetBool("IsWalk", false);
            }
            else
            {
                Agent.SetDestination(Destination.position);
                Anim.SetBool("IsWalk", true);
            }
        }
    }
}
```

GD L10,11 F-L 4,5:

Navigation Mesh (NavMesh)

The lecture begins by exploring how to implement AI-controlled movement using Unity's Navigation Mesh (NavMesh) system. Traditionally, coding AI movement for scenarios like avoiding obstacles or chasing players involves complex logic for each case. NavMesh simplifies this by pre-calculating walkable areas in the game environment.

Why Use NavMesh?

- **Simplifies AI Movement**: Eliminates the need for complex logic to handle various movement scenarios.
- **Efficiency**: Offloads pathfinding calculations to Unity, enhancing performance.
- **Flexibility**: Works seamlessly with complex terrains and level designs.

Steps for Implementing NavMesh:

1. **Mark Static Objects**:
    - Identify and mark all non-moving objects in the scene as "Static." This ensures Unity knows these objects won't move, enabling accurate NavMesh calculations.
    - Example: The plane representing the ground is marked as "Static."
2. **Bake NavMesh**:
    - Initiate the NavMesh baking process to analyze the scene's static geometry and generate NavMesh data, defining walkable areas.
    - Baking settings include slope, step height, and agent radius, which can be adjusted to meet game requirements.
3. **Add NavMesh Agent**:
    - Attach the "NavMesh Agent" component to any game object you want to control with NavMesh. This component allows the object to navigate the baked NavMesh.
    - Configure settings such as:
        - **Agent Type**: Specifies behavior, e.g., humanoid agents avoid collisions and move naturally.
        - **Agent Settings**: Adjust parameters like speed, acceleration, and obstacle avoidance quality.
4. **Set Destination**:

    - Programmatically set the agent's destination using `agent.SetDestination(player.transform.position)` in the `Update()` function. This ensures the AI continuously follows the player as their position updates.

Key Considerations:

- **Performance**: Complex NavMesh settings and multiple agents can affect performance. Optimize baking settings to maintain efficiency.
- **Dynamic Obstacles**: Use the "NavMesh Obstacle" component for moving objects, allowing the NavMesh to update dynamically at runtime. Note that this impacts performance slightly but is necessary for dynamic levels.

Send Messages

Send Message is a Unity mechanism for communication between game objects without requiring direct script references.

Why Use Send Messages?

- **Loose Coupling**: Keeps scripts modular and maintainable by avoiding tight interconnections.
- **Simplicity**: Easier to use compared to managing explicit script references.

How It Works:

- A script calls `SendMessage()` on a target game object, specifying a method name and optional parameters.
- Unity searches for the method in all scripts attached to the target object and executes it.

Code Example:

- In the `EnemyController11` script, `OnCollisionEnter()` sends a "MessageTest" message to the collided object (player) if the tag matches.
- The `PlayerController11` script contains a `MessageTest()` method to receive and print the message.

Benefits:

- **Reduced Dependencies**: Enables script reusability and minimizes cascading changes.
- **Broadcast Messages**: Allows sending messages to multiple objects simultaneously.

Limitations:

- **Performance**: Less efficient than direct method calls.
- **Debugging**: Harder to track message flow in complex interactions.

Coroutines in Unity: A Superior Alternative to Invoke

Coroutines in Unity are C# methods that allow pausing and resuming execution of functions, enabling time-based events and effects like timers and animations. Unlike the `Invoke` method, coroutines offer more flexibility and better control over execution.

Implementation of Coroutines

1. **Declaration**
   - Coroutine functions are declared using the `IEnumerator` return type, signaling to the compiler that the function uses the coroutine concept.
2. **Yielding Control**
   - Coroutines use the `yield return` statement to temporarily pause execution at a specific point, handing over control to other tasks while the coroutine waits in the background.
3. **Types of `yield return`**
   - `WaitForSeconds`: Pauses execution for a specified duration.
   - `yield return null`: Pauses execution for a single frame. This is particularly useful for animations requiring precise timing, such as gradually adjusting a UI element's alpha value.
4. **Resuming Execution**
   - Once the specified delay or condition is met, the coroutine resumes execution from where it paused.
5. **Calling a Coroutine**
   - Coroutines are initiated using the `StartCoroutine(functionName)` method, typically within other functions like `Start`.

Benefits of Coroutines

- **Performance**: Coroutines are performance-friendly as they don't block the main thread, allowing the game to run smoothly during their execution.
- **Control**: Coroutines can be stopped using the `StopCoroutine` method and restarted as needed, unlike the `Invoke` method.
- **Arguments**: Unlike `Invoke`, coroutines can accept arguments, providing greater flexibility in their use.

Examples of Coroutines

Timer Coroutine

```
IEnumerator Timer(float duration)
{
    float startTime = 0f;
    while (startTime < duration)
    {
        startTime += Time.deltaTime;
        yield return null; // Pauses for a single frame
    }
}
```

1.
    - This coroutine accepts a `float` parameter (`duration`) and increments a `startTime` variable until it matches the duration, pausing for a frame after each increment.

Test Coroutine

```
IEnumerator Test()
{
    Debug.Log("Hello");
    yield return new WaitForSeconds(5); // Pauses for 5 seconds
    Debug.Log("World!");
}
```

2.
    - This coroutine prints "Hello," waits for 5 seconds, and then prints "World!".

Applications of Coroutines

1. **Fade-in/Fade-out Effects**
    - Gradually change a UI element's alpha value over time using `yield return null` for smooth animations.
2. **Timers**
    - Ideal for creating precise timers, as demonstrated in the Timer example.
3. **Game Logic**
    - Useful for timed events, such as delaying the start of a game or triggering specific actions after a set duration.

Event Triggers in Unity

Event Triggers in Unity are a versatile way to add custom events and functionalities to UI elements, especially when the built-in button functionalities are insufficient. They are particularly useful for touch screen buttons and complex interactions, making them essential for game development.

Implementation of Event Triggers

1. **Add Event Trigger Component**
    - Attach the **EventTrigger** component to the desired UI element.
    - This can be done via the Unity editor (drag-and-drop) or programmatically in code. For example, an EventTrigger component is added to a `BlankImage` game object.
2. **Create an Event Trigger Entry**
    - An **EventTrigger.Entry** defines the specific event to listen for, such as `PointerClick`.

- Example:

```
EventTrigger.Entry customEvent = new EventTrigger.Entry();

customEvent.eventID = EventTriggerType.PointerClick;
```

    - The `eventID` property specifies the type of event (e.g., `PointerClick`, `PointerDown`, `PointerUp`) to monitor.
3. **Add Listener to the Callback**
    - A callback function executes when the defined event occurs.
    - Use `AddListener` to link a function to the callback of the created entry.

- Example:
  `customEvent.callback.AddListener(Display);`
    - This line ensures that the `Display` function is called when the `PointerClick` event occurs on the UI element.
4. **Add the Entry to the Event Trigger**
    - Add the created entry to the list of triggers for the EventTrigger component.
- Example:
  `eventTrigger.triggers.Add(customEvent);`

Benefits of Event Triggers

- **Flexibility**:
  Event Triggers enable custom interactions beyond the limitations of standard button `onClick` events, which is particularly valuable in mobile game development.
- **Customization**:
  Developers can precisely define responses to various UI interactions, such as swiping, holding, or dragging.

Comparison with Standard Buttons

- **Standard Buttons**:
  Suitable for simple interactions like a pause button using `onClick` events.
- **Event Triggers**:
  Offer greater flexibility and control, making them ideal for more complex interactions, such as:
    - Attack buttons in MOBA games, where multiple key combinations trigger specific functions.
    - Binding functions to specific key combinations, avoiding the need for nested `if` statements.

Use Cases of Event Triggers

1. **Touch Screen Buttons**
    - Create interactive and responsive buttons for touch screen devices.
2. **Complex Interactions**
    - Implement advanced interactions such as swiping, holding, dragging, and dropping.
3. **In-Game Events**

- Link UI interactions to in-game events by triggering specific functions based on user input.

Delegates in C#: A Powerful Tool for Method References

Delegates in C# are a feature that allows you to store references to methods and invoke them later. They are similar to "function pointers" in C++, but with added benefits such as type safety and multicasting, which enables a single delegate to invoke multiple methods simultaneously. Delegates play a crucial role in event-driven programming and game development.

Implementation of Delegates

1. **Declaration**
   - Use the `delegate` keyword to declare a delegate with a specific return type and parameter list.

Example:
```
public delegate void SimpleDelegate(string message);
```

   - This declares a delegate named `SimpleDelegate` that references methods returning `void` and accepting a `string` parameter.
2. **Creating a Delegate Object**
   - Instantiate the delegate and assign it a method reference that matches its signature.

Example:
```
SimpleDelegate mydel = SimpleDelegate(PrintOne);
```

   - Here, `mydel` is a delegate object of type `SimpleDelegate` that references the `PrintOne` method.
3. **Adding Methods (Multicasting)**
   - Delegates support multicasting, allowing you to add references to multiple methods using the `+=` operator.

Example:
```
mydel += PrintTwo;
```

   - This adds the `PrintTwo` method to `mydel`. Now, `mydel` references both `PrintOne` and `PrintTwo`.

4. **Invoking the Delegate**
    - Calling the delegate executes all the methods it references in the order they were added.

Example:
```
mydel("Multicast Example");
```

    - This invokes both `PrintOne("Multicast Example")` and `PrintTwo("Multicast Example")` sequentially.

Use of Delegates in Games

1. **Complex Input Handling in MOBA Games**
    - **Scenario**: Players press combinations of keys (e.g., Spacebar + Q) for specific attacks.
    - **Solution**: Create delegates for each attack and bind them to the respective key combinations. When the combination is pressed, the delegate invokes the associated attack function.
    - **Benefit**: Simplifies input handling and avoids deeply nested `if` statements.
2. **Special Move Execution in Fighting Games**
    - **Scenario**: Special moves require specific input sequences.
    - **Solution**: Use delegates to manage input sequences by listening for correct key presses and triggering the corresponding move when the sequence is complete.
    - **Benefit**: Improves performance by reducing repetitive checks in update loops and simplifies code logic.

Benefits of Delegates

1. **Code Clarity and Organization**
    - Delegates abstract the logic of calling multiple methods into a single invocation, improving readability and simplifying complex event handling.
2. **Flexibility and Reusability**
    - You can dynamically change the methods associated with a delegate, allowing for easy modification of game behavior without altering core logic.
3. **Performance Optimization**
    - Delegates streamline event-driven code by eliminating repetitive condition checks and focusing on specific tasks, resulting in smoother gameplay and better performance.

Key Example: Multicasting Delegates

```
public delegate void SimpleDelegate(string message);

void PrintOne(string message)
{
    Debug.Log("Method One: " + message);
}

void PrintTwo(string message)
{
    Debug.Log("Method Two: " + message);
}

// Implementation
SimpleDelegate mydel = new SimpleDelegate(PrintOne);
mydel += PrintTwo; // Multicasting
mydel("Multicast Example"); // Invokes both PrintOne and PrintTwo
```

Output:

```
Method One: Multicast Example
Method Two: Multicast Example
```

Conclusion

The lecturer emphasizes that mastering delegates is crucial for C# programming and game development in Unity. Delegates offer powerful capabilities for managing method references, handling complex user inputs, and optimizing event-driven logic. Students are encouraged to practice using delegates, as they are foundational to writing efficient and maintainable game co

Practical Applications:

- **Input Handling**: Link actions to key presses or controller inputs.
- **Event Systems**: Notify subscribers when specific events occur.
- **AI Behavior**: Dynamically switch AI behaviors based on conditions.

Additional Concepts:

- **Touchscreen Buttons**: Use event triggers for implementing interactive touchscreen buttons.
- **Performance Profiling**: Utilize tools like Unity Profiler and Android Debug Bridge (ADB) to identify performance bottlenecks.
- **Canvas Size in Photoshop**: Use power-of-two dimensions (e.g., 256x256) for UI elements in Photoshop for better performance in Unity.

GameController11.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;

public class GameController11 : MonoBehaviour
{
    public GameObject BlankImage;
    public Button NormalButton;

    public delegate void SimpleDelegate(string message);

    public void PrintOne(string msg)
    {
        Debug.Log("1st message: " + msg);
    }
```

```
    public void PrintTwo(string msg)
    {
        Debug.Log("2nd message: " + msg);
    }

    void Start()
    {
        SimpleDelegate myDelegate = PrintOne;
        myDelegate("Hello World!");

        myDelegate += PrintTwo;
        myDelegate("Multicast example");

        EventTrigger eventTrigger =
BlankImage.AddComponent<EventTrigger>();
        EventTrigger.Entry customEvent = new EventTrigger.Entry
        {
            eventID = EventTriggerType.PointerClick
        };
        customEvent.callback.AddListener(Display);
        eventTrigger.triggers.Add(customEvent);
    }

    void Display(BaseEventData eventData)
    {
        Debug.Log("Button Test");
    }

    void DisplayTwo()
    {
        Debug.Log("Button Test");
    }
}
```

EnemyController11.cs

```
using System.Collections;
using System.Collections.Generic;
```

```
using UnityEngine;
using UnityEngine.AI;

public class EnemyController11 : MonoBehaviour
{
    public NavMeshAgent Agent;
    public GameObject Player;

    void Start()
    {
        StartCoroutine(Timer(5));

        Agent = GetComponent<NavMeshAgent>();
        Player = GameObject.FindGameObjectWithTag("Player");

        if (Player != null)
        {
            Agent.SetDestination(Player.transform.position);
        }
    }

    void Update()
    {
        if (Player != null)
        {
            Agent.SetDestination(Player.transform.position);
        }
    }

    public IEnumerator Test()
    {
        Debug.Log("Hello");
        yield return new WaitForSeconds(5);
        Debug.Log("World!");
    }

    public IEnumerator Timer(float seconds)
    {
```

```
        float elapsedTime = 0;

        while (elapsedTime < seconds)
        {
            elapsedTime += Time.deltaTime;
            yield return null;
        }
    }

    void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.CompareTag("Player"))
        {
            collision.gameObject.SendMessage("MessageTest", "Hello");
        }
    }
}
```

PlayerController11.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController11 : MonoBehaviour
{
    void Start()
    {
    }

    void Update()
    {
    }

    public void MessageTest(string message)
    {
        Debug.Log("Message Received: " + message);
```

```
        // Uncomment if you want the object to be destroyed after 1
second
        // Destroy(gameObject, 1f);
    }
}
```