Content

Lessons

Lesson 8d – Tick() method

This lesson introduces reading keyboard input in the `PlayerInput` class – which implemets the `IPlayerInput` interface. PlayerInput is not a MonoBehaviour but needs to check for keyboard input on a regular basis.

Instead of making `PlayerInput` a `MonoBehaviour` it is invoked form the `Update()` method of class `Player`.

See the excerpt from class `Player`:

```
public class Player : MonoBehaviour
{
    …
    public IPlayerInput PlayerInput { get; set; } = new PlayerInput();

    private void Awake()
    {
        …
        PlayerInput.MoveModeTogglePressed += MoveModeTogglePressed;
    }

    private void Update()
    {
        _mover.Tick();
        _rotator.Tick();

        PlayerInput.Tick();
    }
}
```

Where `PlayerInput` looks like this:

```
public class PlayerInput : IPlayerInput
{
    …
    public void Tick()
    {
        CheckHotkeyPressed();
        CheckMoverSwitched();
    }

    private void CheckHotkeyPressed()
    {
        …
    }

    private void CheckMoverSwitched()
    {
        …
    }
}
```

Lesson 10d – Basic Entity Animator

This lesson introduces a very simple Animator. A character has an `Entity` script where entity stands for any kind of NPC.

```
public class Entity : MonoBehaviour, ITakeHits
{
    private int _maxHealth = 5;
    public int Health { get; private set; }
    public event Action OnDied;

    private void OnEnable()
    {
        Health = _maxHealth;
    }

    public void TakeHit(int amount)
    {
        Health -= amount;
        if (Health <= 0)
        {
            Die();
        }
        else
        {
            HandleNonLethalHit();
        }
    }

    private void HandleNonLethalHit()
    {
        Debug.Log("Took non-lethal damage");
    }

    private void Die()
    {
        Debug.Log("Died...");
        OnDied?.Invoke();
    }
}
```

The `Entity` class defines an `event OnDied` which is invoked within its private method `Die()`. The class `EntityAnimator` registers for this event and invokes an anonymous Lambda expression.

```
[RequireComponent(typeof(Entity))]
public class EntityAnimator : MonoBehaviour
{
    private Animator _animator;
    private Entity _entity;

    private static readonly int Die = Animator.StringToHash("Die");

    void Awake()
    {
        _animator = GetComponentInChildren<Animator>();


        _entity = GetComponentInChildren<Entity>();
        _entity.OnDied += () => _animator.SetBool(Die, true);
    }
}
```

Also see the optimization introduced by Rider for handling of the animator parameter.

Lesson 11a – State Machine

A basic State implementation consists of three things

- Constructor
- State transition methods like OnEnter() and OnExit()
- State running methods like Tick() (also see 8d)

As a constructor cannot be part of an interface see the resulting `State` interface:

```
public interface IState
{
    void Tick();

    void OnEnter();
    void OnExit();
}
```

To drive the states a state machine is needed, here is the basic implementation:

```
public class StateMachine
{
    private IState _currentState;

    public void SetState(IState state)
    {
        if (_currentState == state)
            return;

        _currentState.OnExit();

        _currentState = state;
        Debug.Log($"Changed state to {state}");
        _currentState.OnEnter();
    }

    public void Tick() // called from Update() in a MonoBehaviour
    {
        _currentState.Tick();
    }
}
```

Lesson 11e – Enhanced State Machine

Here is the more advanced implementation of the state machine, most relevant change is the addition of transitions:

```
public class StateTransition
{
    public readonly IState From;
    public readonly IState To;
    public readonly Func<bool> Condition;

    public StateTransition(IState from, IState to, Func<bool> condition)
    {
        From = from;
        To = to;
        Condition = condition;
    }
}
```

With the changed state machine like this:

```
public class StateMachine
{
    private List<StateTransition> _stateTransitions = new List<StateTransition>();
    private List<StateTransition> _anyStateTransitions = new
                                                        List<StateTransition>();

    private IState _currentState;
    public IState CurrentState => _currentState;

    public void AddTransition(IState from, IState to, Func<bool> condition)
    {
        var stateTransition = new StateTransition(from, to, condition);
        _stateTransitions.Add(stateTransition);
    }

    public void AddAnyTransition(IState to, Func<bool> condition)
    {
        var stateTransition = new StateTransition(null, to, condition);
        _anyStateTransitions.Add(stateTransition);
    }

    public void SetState(IState state)
    {
        if (_currentState == state)
            return;

        _currentState?.OnExit();

        _currentState = state;
        Debug.Log($"Changed state to {state}");
        _currentState.OnEnter();
    }

    public void Tick()
    {
        StateTransition transition = CheckForTransition();
        if (transition != null)
        {
            SetState(transition.To);
        }

        _currentState.Tick();
    }

    private StateTransition CheckForTransition()
    {
        foreach (StateTransition transition in _anyStateTransitions)
        {
            if (transition.Condition())
            {
                return transition;
            }
        }

        foreach (StateTransition transition in _stateTransitions)
        {
            if (transition.From == _currentState && transition.Condition())
            {
                return transition;
            }
        }

        return null;
    }
}
```

Lesson 12d – Event chaining / propagation

In lesson 12d the `StateMachine` class fires an event on state changes. This state machine is used by class `EntityStateMachine` which is used for NPCs and is a MonoBehaviour driving the state machine via its `Tick()` method.

Another class (`NpcLoot`) which is also a MonoBehaviour and a component on NPCs as well (and thus can easily retrieve `EntityStateMachine` via `GetComponent<>` calls) now needs to be informed of state changes. Instead of exposing the inner state machine in `EntityStateMachine`, the latter adds an event as well for state changes and thus propagates the event from the inner state machine.

Very cool! See StateMachine first (for full code see lesson 11e):

```
public class StateMachine
{
    …
    public event Action<IState> OnStateChanged;

    public void SetState(IState state)
    {
        if (_currentState == state)
            return;

        _currentState?.OnExit();

        _currentState = state;
        Debug.Log($"Changed state to {state}");
        _currentState.OnEnter();

        OnStateChanged?.Invoke(_currentState); // invoke event on state change
    }

    // Tick() is called from EntityStateMachine.Update()
    public void Tick()
    {
        StateTransition transition = CheckForTransition();
        if (transition != null)
        {
            SetState(transition.To);
        }

        _currentState.Tick();
    }
        …
}
```

Next up, `EntityStateMachine` registers for the event and from `StateMachine` and adds an event of its own. If the first event is fired, the latter is fired as well.

```
public class EntityStateMachine : MonoBehaviour
{
    private StateMachine _stateMachine; // StateMachine class from above

    …
    public event Action<IState> OnEntityStateChanged; // event in this class


    private void Awake()
    {
        // set up references
        Player _player = FindObjectOfType<Player>();
        NavMeshAgent _navMeshAgent = GetComponent<NavMeshAgent>();
        Entity _entity = GetComponent<Entity>();

        // event propagation
        _stateMachine = new StateMachine();
        _stateMachine.OnStateChanged += state =>
              OnEntityStateChanged?.Invoke(state);

        // states for entities
        Idle idle = new Idle();
        ChasePlayer chasePlayer = new ChasePlayer(_navMeshAgent, _player);
        … more states
        … add transitions etc.

        _stateMachine.SetState(idle);
    }


    // drive state machine!
    private void Update()
    {
        _stateMachine.Tick();
    }
}
```

Finally NpcLoot can register for the event in EntityStateMachine and will receive state changes from StateMachine!

```
public class NpcLoot : MonoBehaviour
{
    …
    private Inventory _inventory;
    private EntityStateMachine _entityStateMachine;

    private void Awake()
    {
        _inventory = GetComponent<Inventory>();

        _entityStateMachine = GetComponent<EntityStateMachine>();
        _entityStateMachine.OnEntityStateChanged += HandleEntityStateChanged;
    }
     …

}
```

Lesson 12d addendum – Event chaining / propagation

Here is what it would look like, if not only the new state was propagated but the previous state as well.

Changes to `StateMachine`:

```
public class StateMachine
{
    …
    public event Action<Istate, IState> OnStateChanged;

    public void SetState(IState state)
    {
        if (_currentState == state)
            return;

        _previousState = _currentState; // new field
        _currentState?.OnExit();

        _currentState = state;
        Debug.Log($"Changed state to {state}");
        _currentState.OnEnter();

        OnStateChanged?.Invoke(_currentState, _previousState);
    }
```

Changes to EntityStateMachine:

```
public class EntityStateMachine : MonoBehaviour
{
    private StateMachine _stateMachine; // StateMachine class from above

    …
    public event Action<Istate, IState> OnEntityStateChanged;

    private void Awake()
    {
        // set up references
        Player _player = FindObjectOfType<Player>();
        NavMeshAgent _navMeshAgent = GetComponent<NavMeshAgent>();
        Entity _entity = GetComponent<Entity>();

        // event propagation
        _stateMachine = new StateMachine();
        _stateMachine.OnStateChanged += (state, previousState) =>
              OnEntityStateChanged?.Invoke(state, previousState);
```

Finally `NpcLoot` will have to register on the event providing a method that will have to parameters, one for the new state and one for the previous state.

Lesson 14 – GameStateMachine

Lesson 14 introduces a game state machine. This state machine builds on the same state machine as the entity state machine and except for other states and transitions of course.

There are two notable exceptions:

- the game state machine is a single instance game machine (though not a Singleton)
- the game state machine's event for state changes is static

Single instance

```
private static bool _initialized;
private StateMachine _stateMachine;

private void Awake()
{
   // single instance
   if (_initialized)
    {
        Destroy(gameObject);
        return;
    }

    _initialized = true;
    DontDestroyOnLoad(gameObject);

    // state machine and event propagation (see lesson 12d)
   _stateMachine = new StateMachine();
    _stateMachine.OnStateChanged += (state, previousState) =>
                        OnGameStateChanged?.Invoke(state, previousState);

    // … next init states and transitions
}
```

The idea here is to be able to add the game state machine to multiple scenes for testing the scenes without having to run through multiple menu or initialization levels first and to still always only have one state machine in the game.

Static event

Compare the game state machine event with the entity state machine's and there is one notable change in the public fields.

```
public class GameStateMachine : MonoBehaviour
{
    public Type CurrentStateType => _stateMachine.CurrentState.GetType();
    public static event Action<IState, IState> OnGameStateChanged;

    private StateMachine _stateMachine;
```

versus:

```
public class EntityStateMachine : MonoBehaviour
{
    public Type CurrentStateType => _stateMachine.CurrentState.GetType();
    public event Action<IState, IState> OnEntityStateChanged;

    private StateMachine _stateMachine;
```

The reason for the game state machine's static event is, that the entity state machine is attached to entities within the scene and its state changes will only be relevant within the scene whereas the game state machine's state will be relevant also outside of the scene – for example to UI elements in a separate UI scene.

The current use case as of lesson 14 is the class `PauseCanvas` where on state change to state Pause the canvas' game object is set to active:

```
public class PauseCanvas : MonoBehaviour
{
    [SerializeField] private GameObject _pausePanel;

    private void Awake()
    {
        _pausePanel.SetActive(false);
        GameStateMachine.OnGameStateChanged += HandleGameStateChanged;
    }

    private void OnDestroy()
    {
        GameStateMachine.OnGameStateChanged -= HandleGameStateChanged;
    }

    private void HandleGameStateChanged(IState state, IState previousState)
    {
        _pausePanel.SetActive(state is Pause);
    }
}
```

The pause canvas is as stated above part of the UI scene.

Rider Tips and Shortcuts

Lesson 11b @02:55 – Multiselect

Select square text area without line selection by holding down Alt and selecting an area in the editor.

Lesson 11d @12:55 – Pause before Play

Press Pause button before pressing Play so game starts in paused mode and pressing the step button will jump a frame per klick.

Lesson 11d @12:55 to 14:55 – Attach debugger while game is already playing

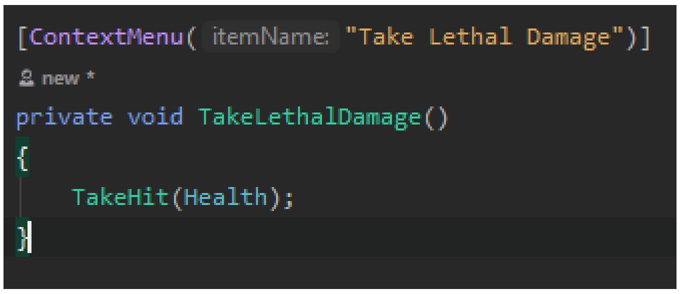
Combinig this with the previous step this will work best. You have a game in paused mode and will jump ahead one step per klick. Now attach debugger as demonstrated around 14:05 and you can debug for example Update() easily.

Unity Tips and Shortcuts

Lesson 11e @08:00 – Context Menus for MonoBehaviours

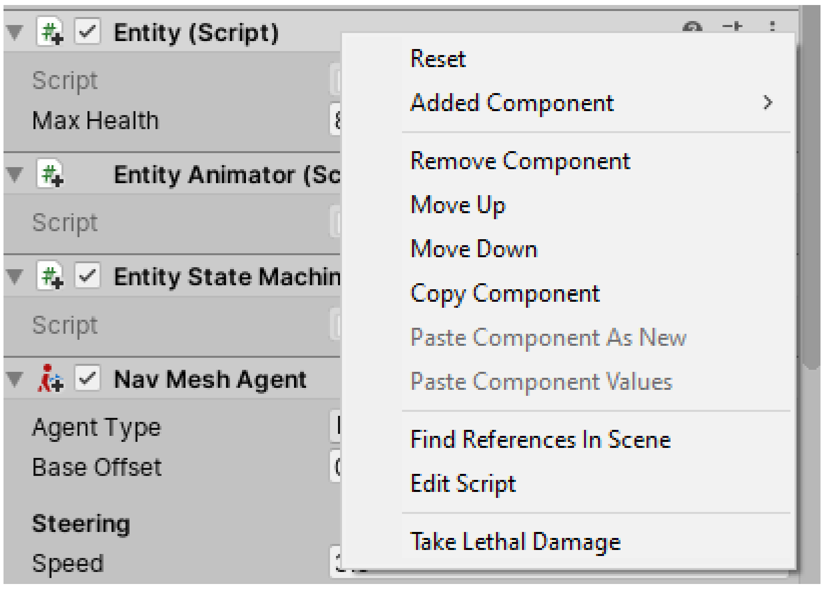
For MonoBehaviour components (i.e. scripts) methods can be annotated with: [ContextMenu(string itemname).

In this case a private method TakeLethalHit() is added to class Entity like this:



```
[ContextMenu( itemName: "Take Lethal Damage")]
new *
private void TakeLethalDamage()
{
    TakeHit(Health);
}
```

This will result in the method being invokable in the Unity Editor (also during play mode) on right clicking the component:



In the menu item „Take Lethal Damage“ is clicked the method will be invoked, killing the entity in the game.