Sperasoft
Your game dev partner

Unity3D Scripting: State Machine

Finite-State Machine

Definition:

- This is a set of five elements:
  $(\Sigma, Q, s \in Q, T \subset Q, \delta : Q \times \Sigma \to Q)$
  where Σ - the alphabet, Q - a set of states, s - the initial (start) state, T - the set of accepting states, δ - the transition function.

- A graph with a finite number of states (nodes) and a finite number of CMV transitions (curves).

Two Questions

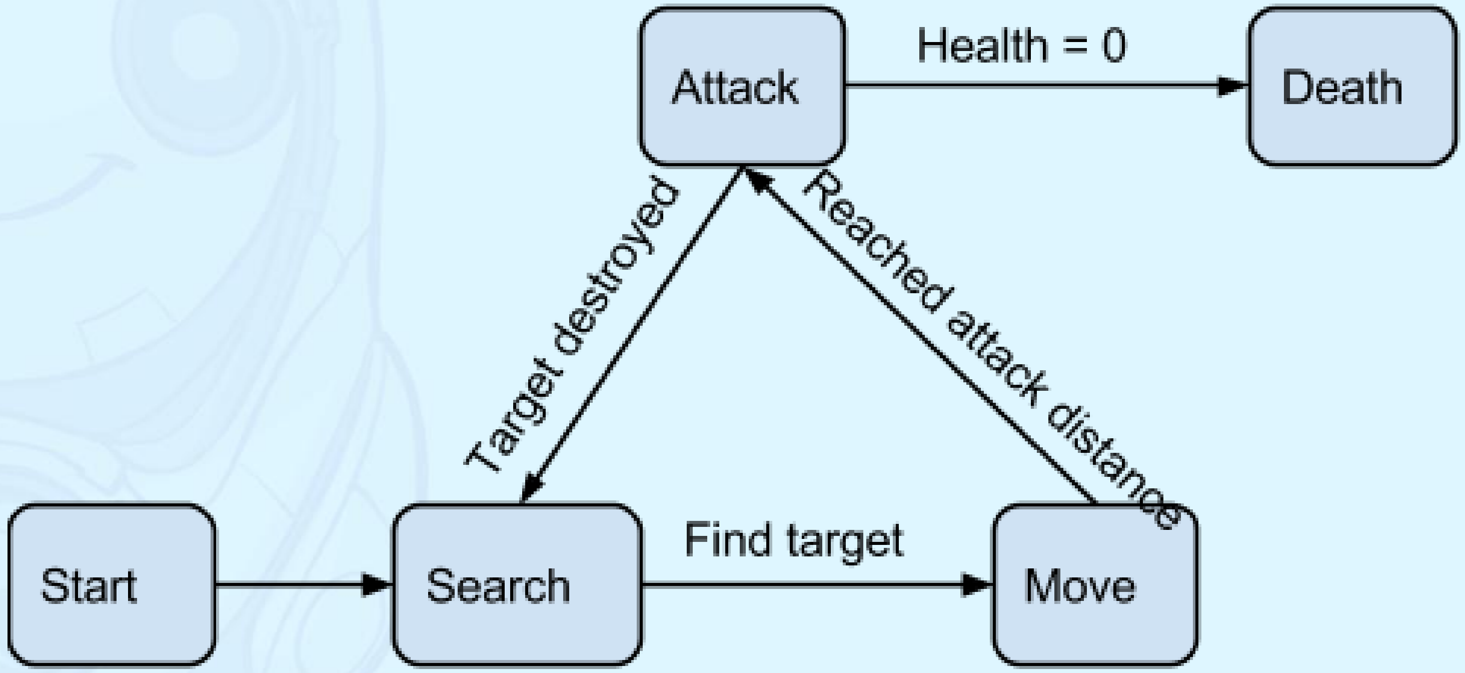
- How this system would help us in programming the game logic?
- How this approach is better than others?

How To Use

Behavior of the game object may be divided into a list of different actions:

- **Starting State**
- **Target Searching**
- **Attack**
- **Death**

Structure


Attack
Health = 0
Death
Target destroyed
Reached attack distance
Start
Search
Find target
Move

Generalization

Any game component, whether a unit, or a menu based or an abstract game as a whole, which uses certain rules can be programmed by using a state machine.

Advantages of State Machine

1. Divided, limited access to the context of state
2. Encapsulated logic for each state
3. Clarity, ability to upgrade code (source code customization)
4. Reliability

Divided, Limited-Access Context State

Each state controls only its own context and there is no need to check the set of variables for other states

Logic Encapsulation for One State

This allows you to quickly respond to errors that were found during testing, to localize them and make changes. At the same time this greatly reduced the likelihood of introducing errors into other states, because their context is not available, and their logic is elsewhere.

Source Code Customization

Behavior of the object becomes more classification-ruled and understandable.

Behavior modernization takes less time and complexity associated with modifications is greatly reduced.

Reliability

Transition from one state to another occurs only in specific cases, to move from one state to another if it was not planned is not possible.

Property of C #: Operator: “Yield"

Operator: “Yield"

- **"syntactic sugar"**
- **implement pattern: enumerator**

Is being used for:

- **implementing collection**
- **dividing the methods in several steps**

Unity 3D - Specifications

We will use several features Unity3D:

- Operator yield and pattern enumerator.

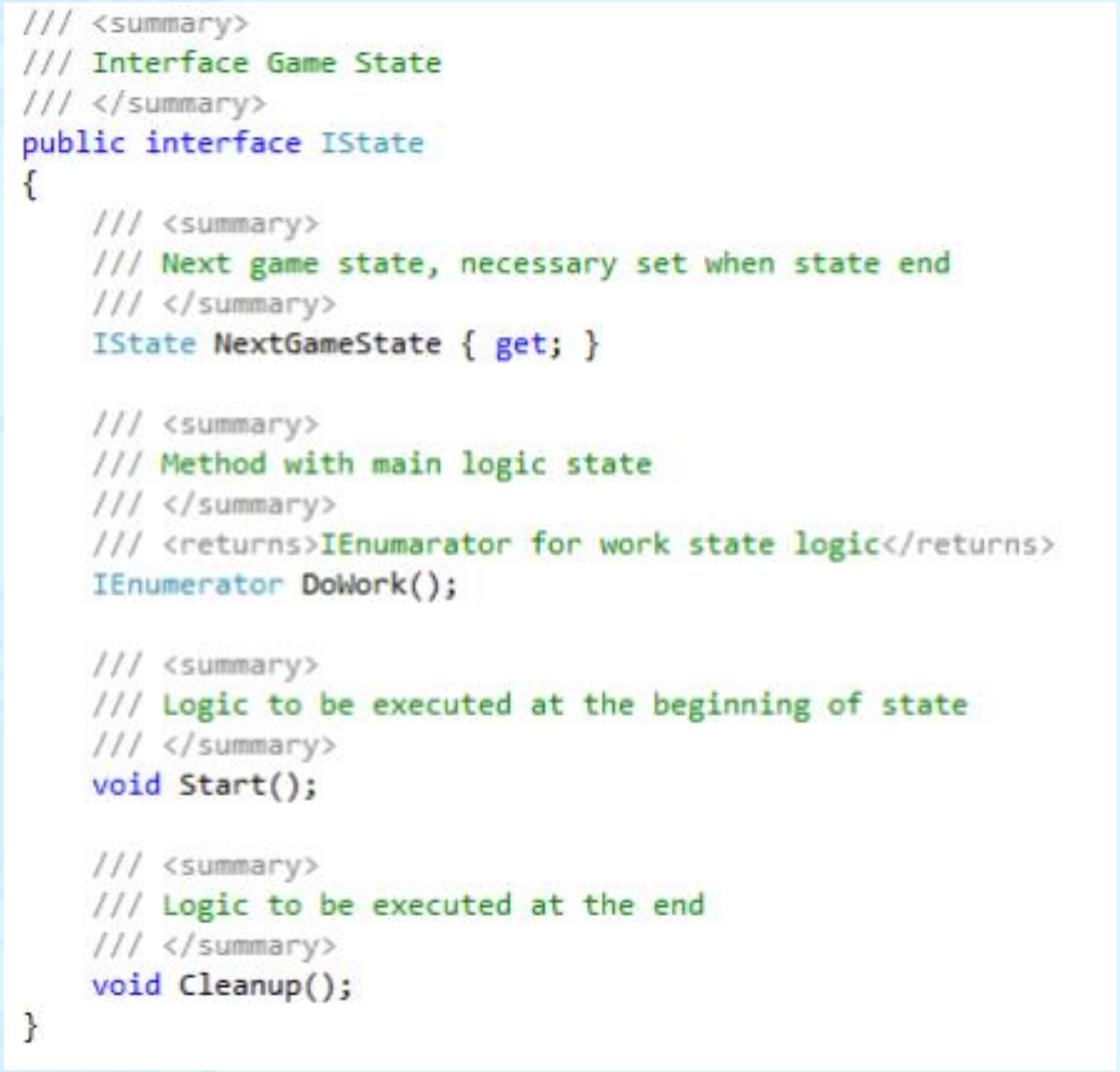
- Call for one of the standard object methods inherited from MonoBehaviour:
  - Update
  - FixedUpdate
  - Start, etc.

Unity 3D - Specifications

We need the following entities:

- **States**
- **Manager, which manages the transition from one state to another**
- **Model - context of state machine, to which the state will call to perform a variety of actions.**

State. Implementation Example

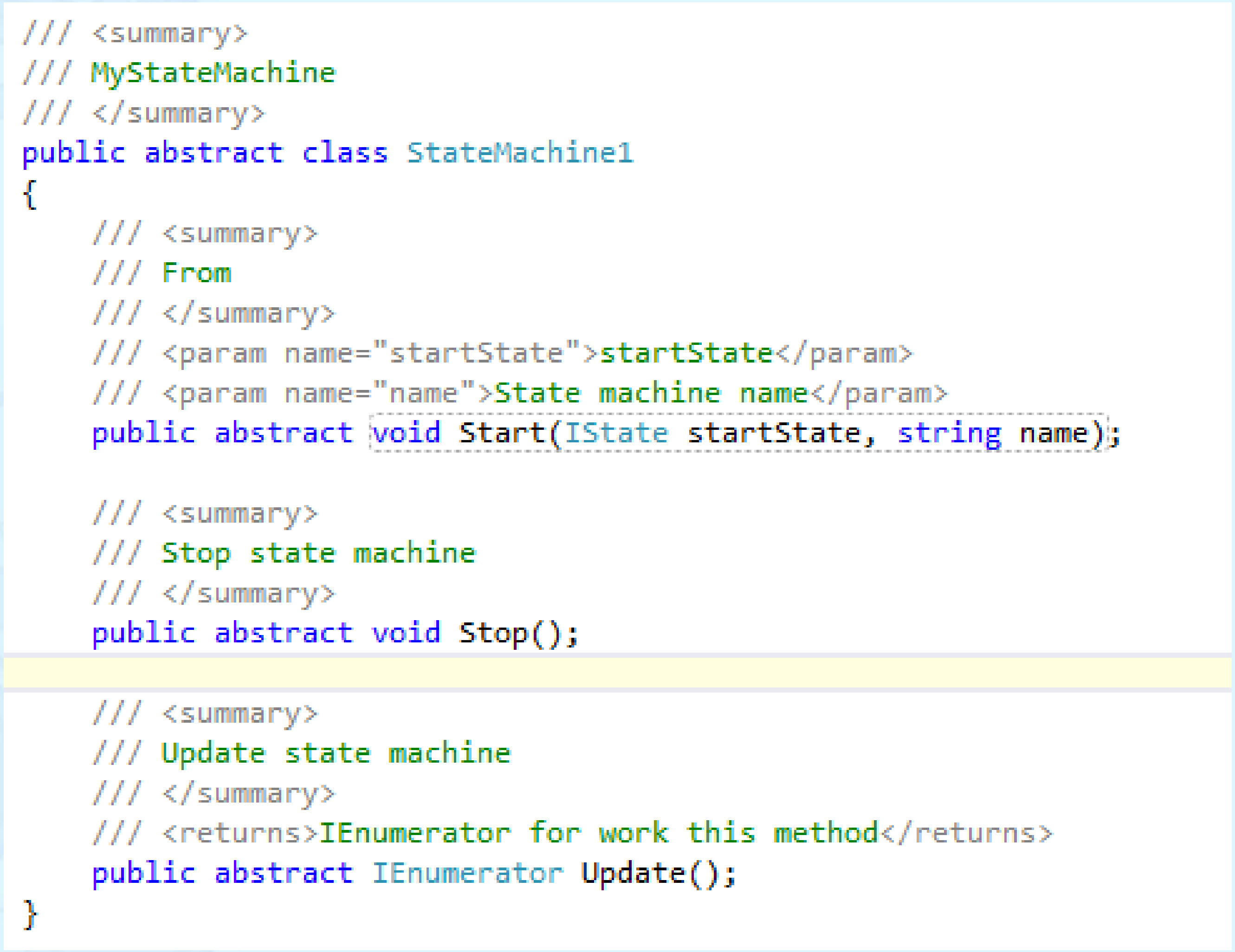


```
/// <summary>
/// Interface Game State
/// </summary>
public interface IState
{
    /// <summary>
    /// Next game state, necessary set when state end
    /// </summary>
    IState NextGameState { get; }

    /// <summary>
    /// Method with main logic state
    /// </summary>
    /// <returns>IEnumarator for work state logic</returns>
    IEnumerator DoWork();

    /// <summary>
    /// Logic to be executed at the beginning of state
    /// </summary>
    void Start();

    /// <summary>
    /// Logic to be executed at the end
    /// </summary>
    void Cleanup();
}
```

State. Description

No need to restrict the user with inheritance. System is fairly abstract, to restrict certain properties of the state through the interface.

States Manager: Implementation Example



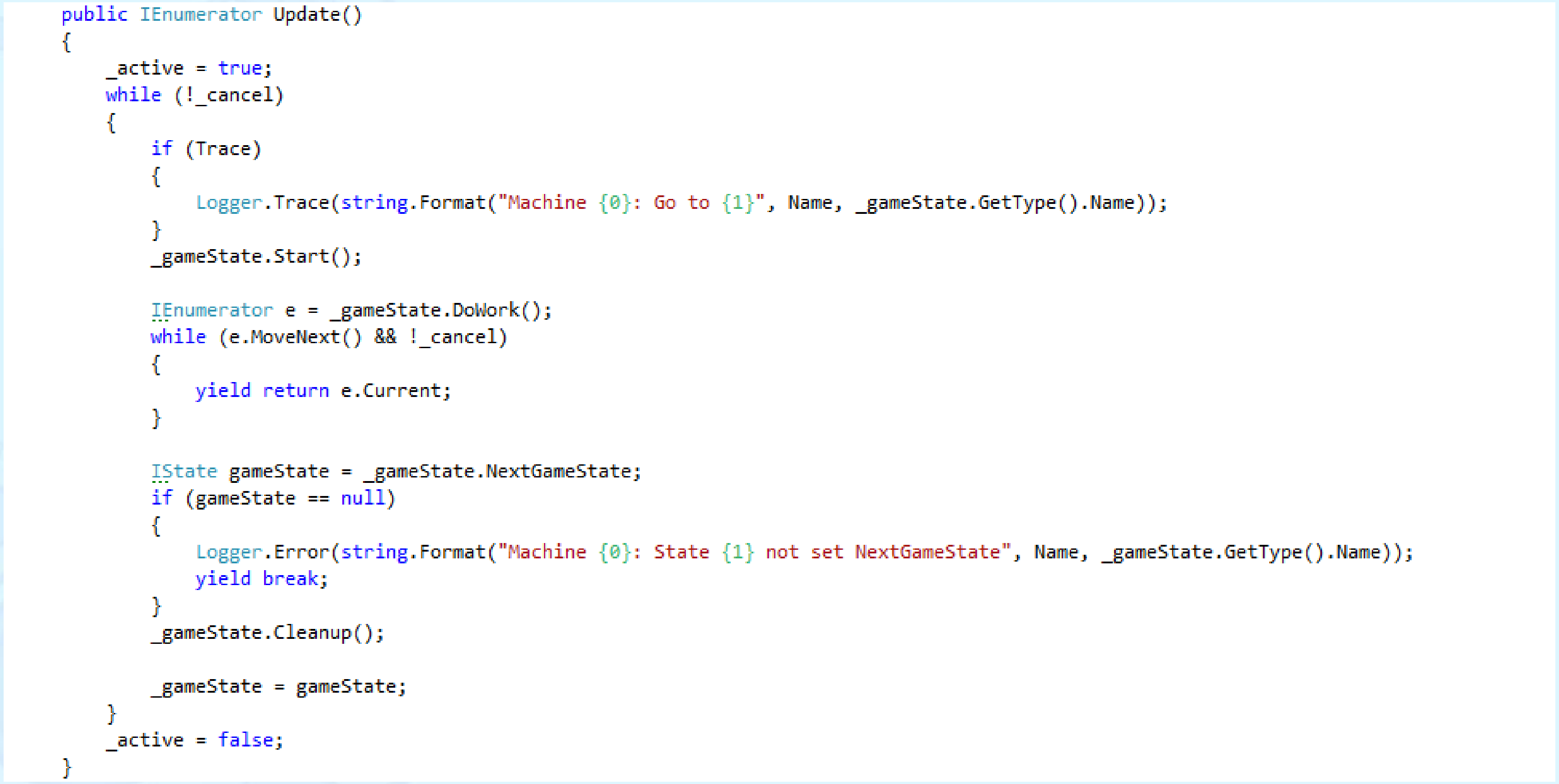
```
/// <summary>
/// MyStateMachine
/// </summary>
public abstract class StateMachine1
{
    /// <summary>
    /// From
    /// </summary>
    /// <param name="startState">startState</param>
    /// <param name="name">State machine name</param>
    public abstract void Start(IState startState, string name);

    /// <summary>
    /// Stop state machine
    /// </summary>
    public abstract void Stop();

    /// <summary>
    /// Update state machine
    /// </summary>
    /// <returns>IEnumerator for work this method</returns>
    public abstract IEnumerator Update();
}
```

States Manager: Explanation

Key methods:

- Set the initial state
- Infinite state that calls the logic of the current status
- Cleanup

Infinite cycle. Implementation Example

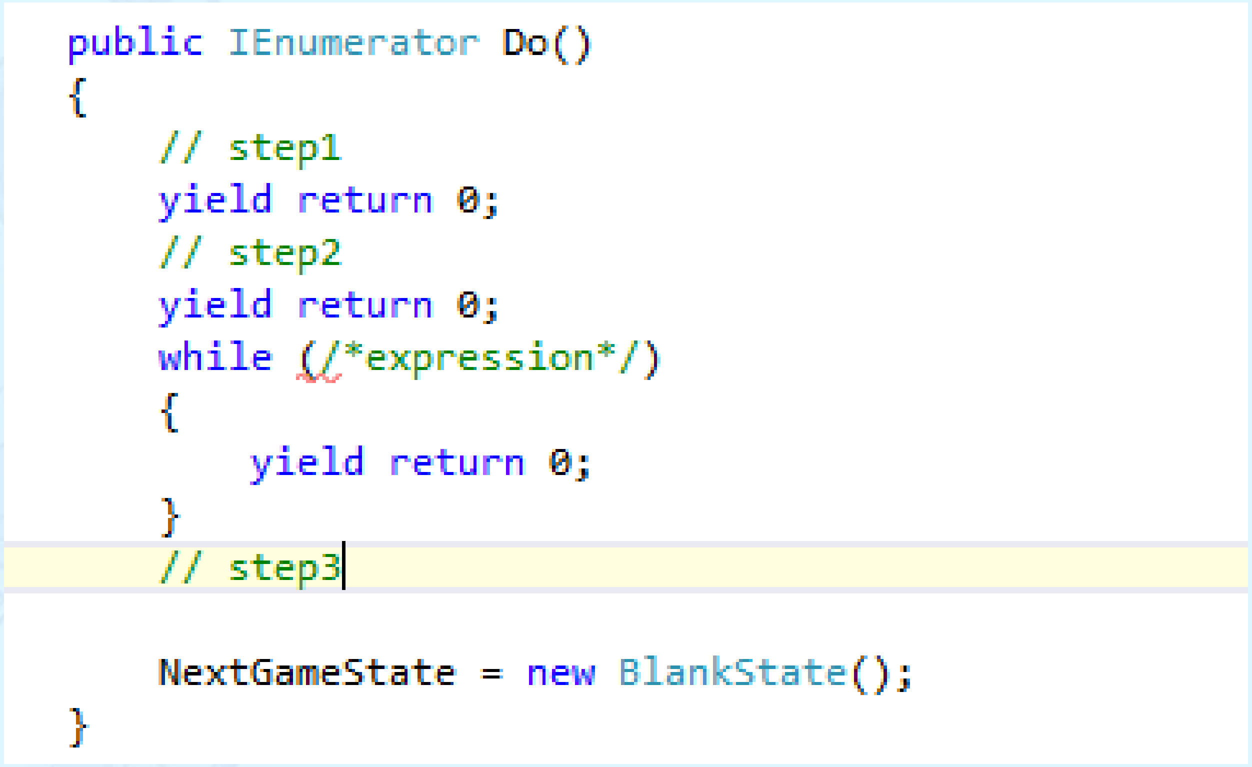


```
public IEnumerator Update()
{
    _active = true;
    while (!_cancel)
    {
        if (Trace)
        {
            Logger.Trace(string.Format("Machine {0}: Go to {1}", Name, _gameState.GetType().Name));
        }
        _gameState.Start();

        IEnumerator e = _gameState.DoWork();
        while (e.MoveNext() && !_cancel)
        {
            yield return e.Current;
        }

        IState gameState = _gameState.NextGameState;
        if (gameState == null)
        {
            Logger.Error(string.Format("Machine {0}: State {1} not set NextGameState", Name, _gameState.GetType().Name));
            yield break;
        }
        _gameState.Cleanup();

        _gameState = gameState;
    }
    _active = false;
}
```

Infinite Cycle. Clarification

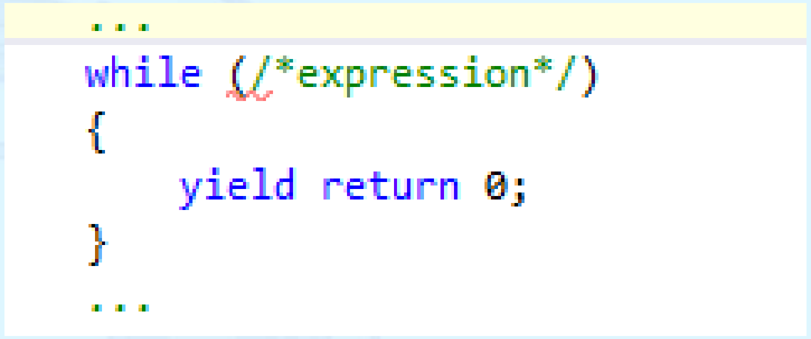
- Possibility of correct completion
- Debug messages output
- Calling IState.Start (), IState.Cleanup (), IState.DoWork () methods
- Switch state with the help of Field NextGameState

IState.DoWork (). Implementation Example



```
public IEnumerator Do()
{
    // step1
    yield return 0;
    // step2
    yield return 0;
    while (/*expression*/)
    {
        yield return 0;
    }
    // step3

    NextGameState = new BlankState();
}
```

Awaiting Something



```
...
while (/*expression*/)
{
    yield return 0;
}
...
```

Check one of the conditions, and if it does not suffice you pass control higher up and go back to check conditions at other times when the manager would transfer control method.

This way you give a chance to any other logic to be executed in a single game flow (GAME THREAD), avoiding a whole bunch of problems related to multithreading.

Methods

For any application, we need a method to invoke the script cyclically inherited from MonoBehaviour: Update, FixedUpdate, LaterUpdate, etc.

Depending on the task you should choose the right method and call it for update the state machine.

Before you update the state machine it is necessary to perform the initial state install. This can be done easily in the Start or Awake methods.

Model (context)

Often there is a need preserve some information between the states was preserved some information - the context of state machine. This context could be a script that renews the state machine. It is possible to store certain flags and objects necessary for the states in this script.

Also, this context may contain links to game resources and other game objects. So you pass a reference from state to state to this script.

This approach will help to avoid a large number of singletons in the game.