C# JellyFarm

지원자 : 김현지

작성일 : 2022년 12월 15일

프로젝트 설명

게임 장르 : 클리커 방치형 게임
엔진 버전 : Unity 2021.3.12f1

게임 설명 :
젤리를 클릭하여 젤라틴을 얻고 매매하여 골드를 얻습니다.
2D 게임을 공부하고자 유튜브의 강의 영상을 보면서 시작,
Bolt 로 구현된 게임을 Script를 작성하여 기능을 구현하였습니다.

프로젝트 소스 링크 : https://github.com/asterism1030/2D-JellyFarm

플레이 화면

5) 젤리 구매 및 해금

6)젤리 수용량 및 생산량 업그레이드

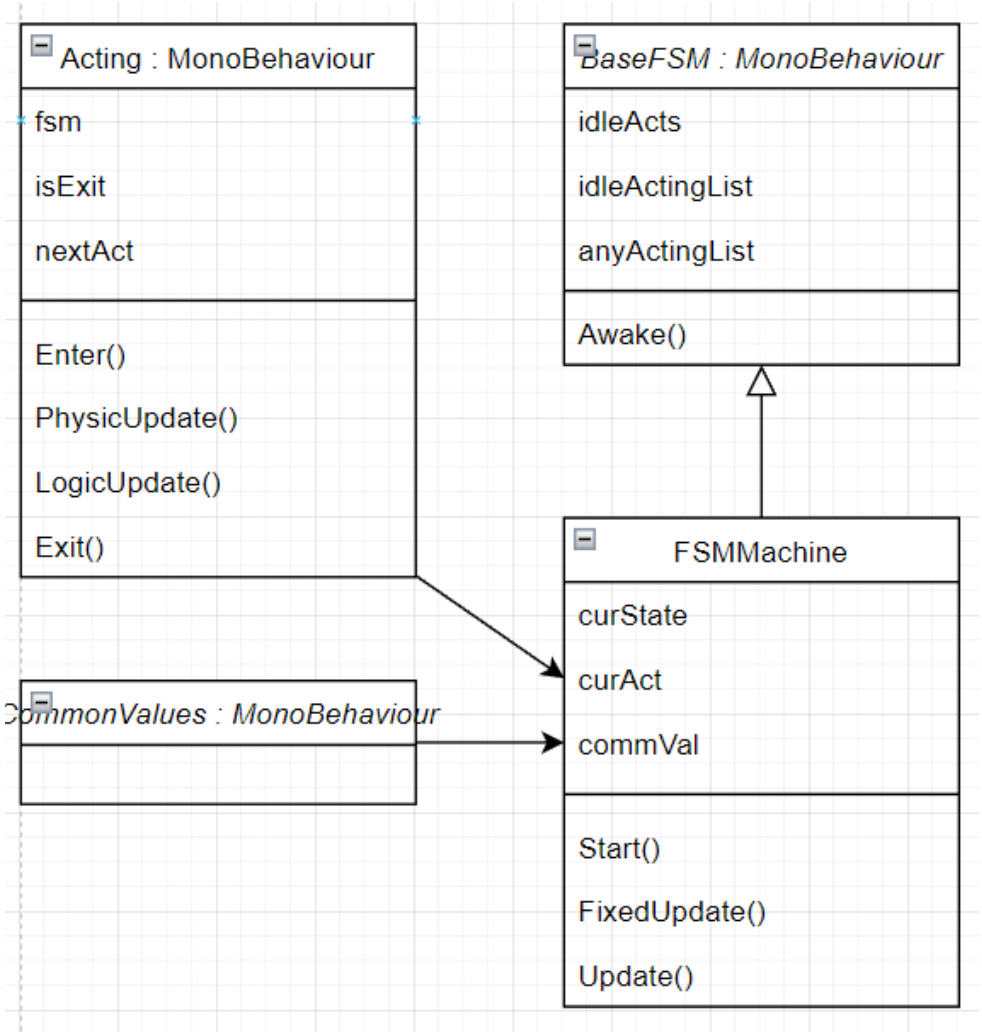
2) 다른 젤리를 해금할 수 있는 젤라틴을 얻음

4) 방치를 해도 골드를 얻음

1) 생성 된 젤리를 누르면

3) 젤리를 아이콘 위로 드래그 드랍을 하면 골드를 얻음

Script 구조

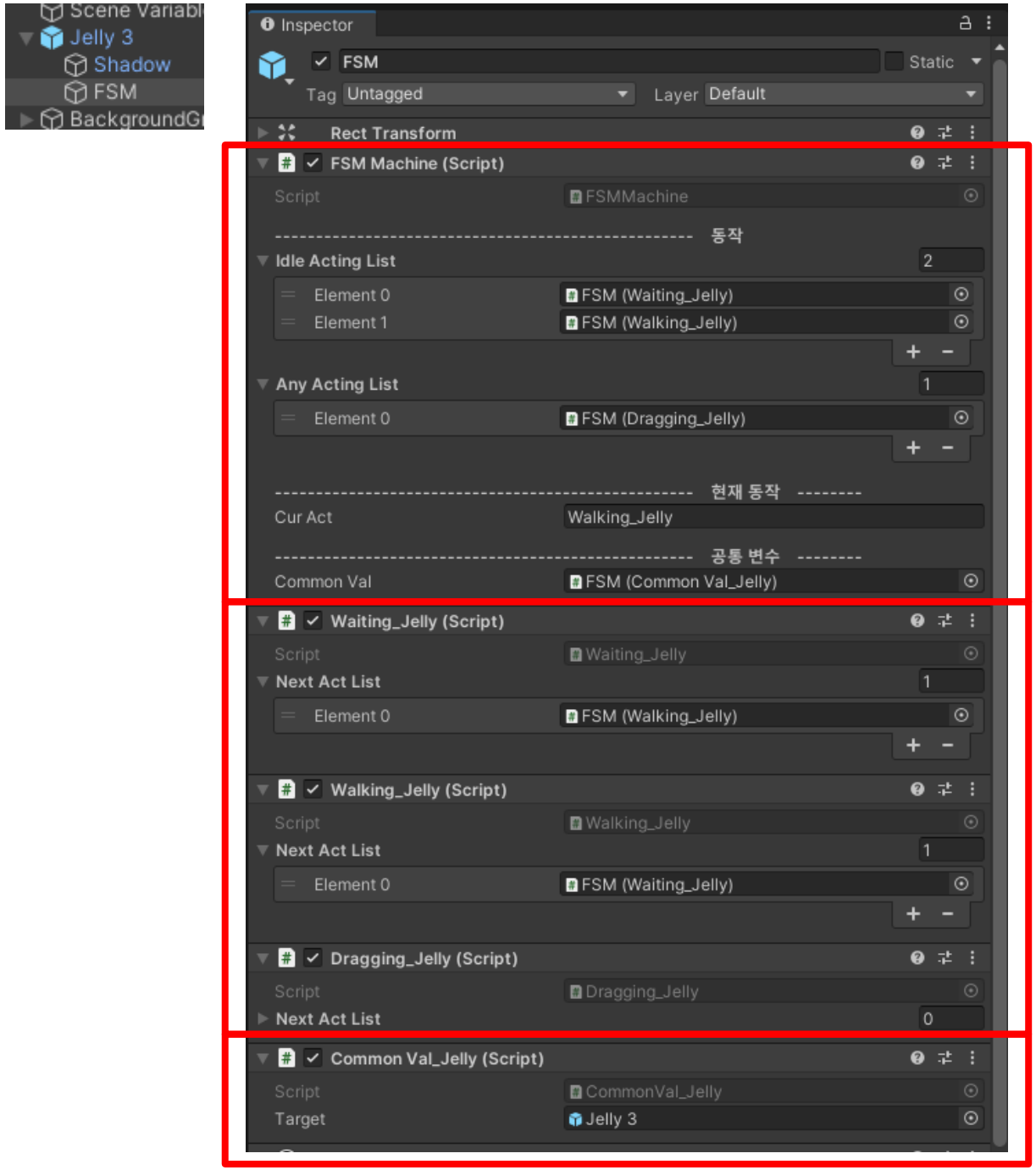


BaseFSM
: Acting 들을 변수로써 관리
- idleActingList : 특정 조건을 만족해야 동작
- anyActingList : 매번 동작

FSMMachine
: Acting 을 상속한 클래스들을 LifeCycle 주기에 맞춰 동작
- FixedUpdate : Enter, PhsyicUpdate 실행
- Update : LogicUpdate, Exit 실행

Acting
: Move, Jump 등 하나의 동작

CommonValues
: 편리성을 위한 클래스로 상속하여 Acting 들이 공통으로 사용하는 변수를 선언함



FSMMachine

Acting

CommonValues

기타 Script 구조

<Manager> : **싱글톤**
- GameManager.cs : 게임 상에서의 동작
- GameInfo.cs : 게임 상에서의 정보
- UserInfo.cs : 플레이 유저에 대한 정보
- SoundManager.cs : 사운드

<Event> : EventTrigger **등에서 사용**
- JellyEvent.cs : 젤리에 대한 전반적 이벤트
- JellyPanelEvent.cs : 특정 판넬 동작 구현
- PanelEvent.cs : 판넬의 기본적인 동작 (숨기기, 보이기)

<Object> : GameObject **의 직접적인 동작에 사용**
- Scrolling.cs : 배경의 구름 이동
- Jelly.cs : Jelly 의 정보

(Project 상에서의 Script 분류는 씬에서의 부모 오브젝트 이름을 따라감)

FSM 개발 고민 흔적

Jelly AI

IEumerator

사용하지 않고?

IEnumerator 함수가 많이 필요한 만큼
너무나 지저분해 보였다..

Jelly AI

Switch~case

아하!

다른 좋은 코드가 있을까?
구글링을 하니
실제로 FSM Pattern 존재..
자신은 FSM Pattern 을 작성하고 있었음
어느 AI 에도 바로 적용할 수 있도록
구조 리팩토링을 하기로 함..

Jelly **FSM**

OOP 화

가독성? 재활용성?
이식성?

Git의 다른 코드들을 보며
좋은 코드에 대해 고민을 거듭

Jelly FSM

FSMMachine

완성!

proj

이름

- AI
- DirectX_Prac_1
- DirectX_Prac_4
- PhysicPuzzle
- Scroll_AngryBird
- TestAndPractice
- Touch_Game
- 개발 고민 흔적

개발 고민 흔적 - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

젤리의 AI 동작
- 대기한다
- 움직인다
- 터치를 당한다

상태머신

위의 동작들을 상태머신(시리얼라이즈드화 해서 동작들을 UI 적으로 드래그 드랍하여 설정할 수 있게 하여 젤리 이외에도 다른 오브젝에 AI 동작을 커스터마이징 할 수 있게 한다)

FSMMachine
: 말그대로 이 기능을 구현하는 것
behavior 에 동작들을 넣고 (번호로 구분, 정해진 동작 이름이 없으므로)
condition 에 동작들의 전환 조건을 입력한다
-> 다양할 수 있으므로 src -> dest 로 인자 2개 전달
condition 전환 메소드는 별도로 구현하여 해당 클래스에 전달하도록 함

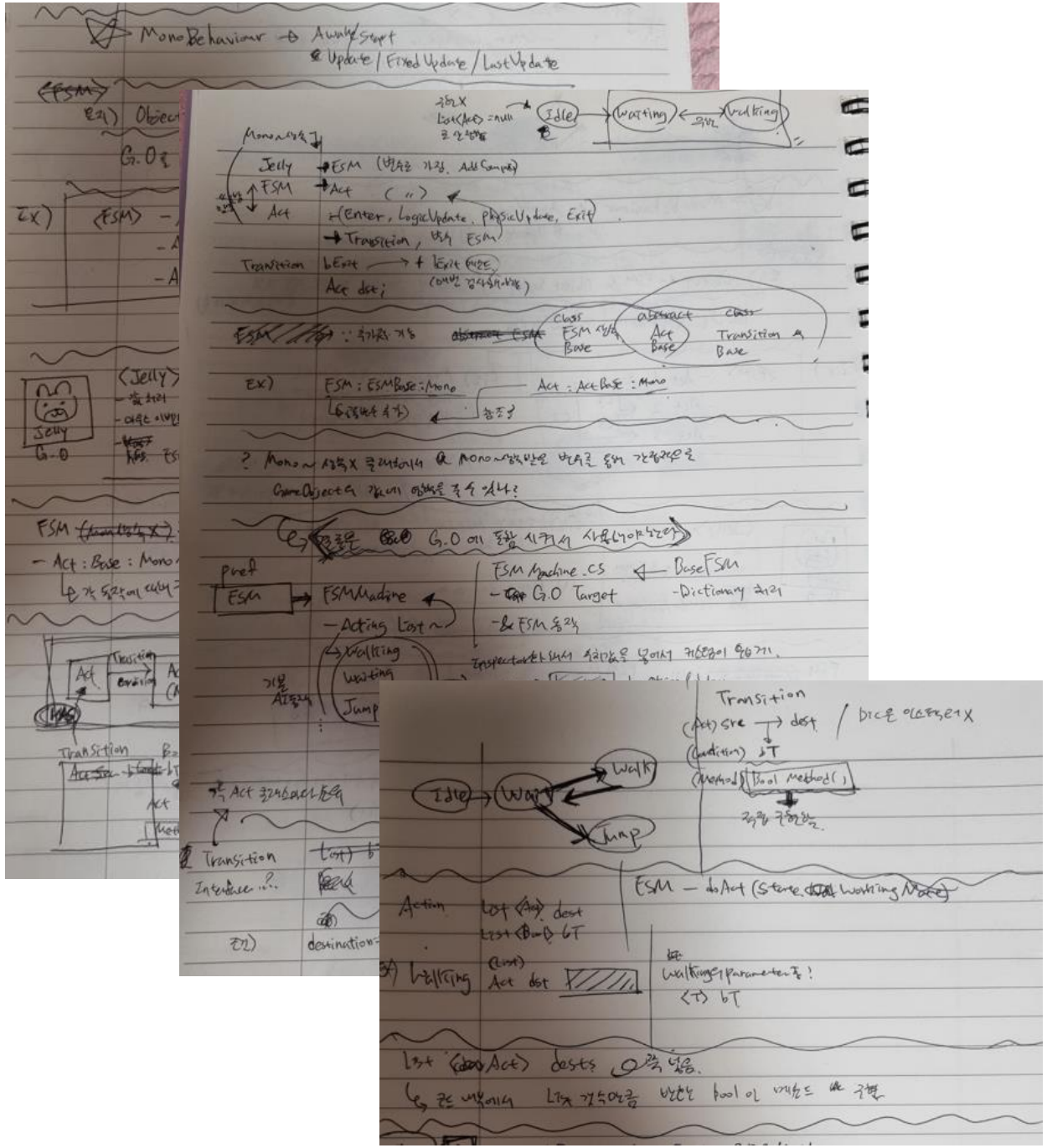
사실상 동작들을 관할하는 manager 격의 클래스는 필요함

번호로 구분은 헷갈릴 수 있으므로(동작이 많아지면) 클래스 명으로 구분하도록 한다
인스펙터 창에 동작들을 드래그 드랍하여 넣을 예정이었으나 되지가 않고(검색 해야겠지만 배고파서 머리 안돌아간다) 생각해보니 꽤 귀찮은 작업이므로 FSM머신에 코드로 넣도록 한다
코드 -> 인스펙터 순으로 초기화 되므로 그 방법이 동작 순서상 더 나을듯

AI 들은 다양할수 있음
해당 오브젝트마다 FSM 머신을 갖고 있도록 함

구조 정리 :
Object 굵직한 클래스
-> has FSM
-> FSM 에 동작 넣음
-> 동작 전환을 위한 조건 넣음

타겟 오브젝트 변수는 어떻게 할것인가?



FSMMachine.cs

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

[Serializable]
public class FSMMachine : BaseFSM
{
    enum State { NONE, ENTER, LOGICUPDATE, PHYSICUPDATE,
EXIT };

    State curState = State.NONE;

    [Header("----------------------------------------  현재 동작  --------")]
    public string curAct = null;

    [Header("----------------------------------------  공통 변수  --------")]
    public CommonValues commonVal;

    void Start()
    {
        foreach(Acting act in idleActs.Values) {
            act.FSM = this;
        }
        foreach(Acting act in anyActingList) {
            act.FSM = this;
        }
    }

    void FixedUpdate()
    {
        // AnyState 동작
        foreach(Acting act in anyActingList) {
            act.Enter();
            act.PhysicUpdate();
        }

        // Entry -> Idle 을 거치는 동작
        if(curAct == null || idleActs.ContainsKey(curAct) == false) {
            return;
        }

        switch(curState) {
            case State.NONE:                    // = idle
            {
                idleActs[curAct].Init();
                curState = State.ENTER;
            }
            break;
            case State.ENTER:                   // Enter to Node
            {
                idleActs[curAct].Enter();
                curState = State.PHYSICUPDATE;
            }
            break;
            case State.PHYSICUPDATE:
            {

                idleActs[curAct].PhysicUpdate();
                curState = State.LOGICUPDATE;
            }
            break;
        }
    }

    void Update()
    {
        // AnyState 동작
        foreach(Acting act in anyActingList) {
            act.LogicUpdate();
            act.Exit();
        }

        // Entry -> Idle 을 거치는 동작
        if(curAct == null || !idleActs.ContainsKey(curAct)) {
            return;
        }

        switch(curState) {
            case State.LOGICUPDATE:
            {
                idleActs[curAct].LogicUpdate();

                // isExit ?
                if(idleActs[curAct].isExit == true) {
                    curState = State.EXIT;
                }
                else {
                    curState = State.PHYSICUPDATE;
                }

            }
            break;
            case State.EXIT:
            {
                idleActs[curAct].Exit();

                if(idleActs[curAct].NextAct != null) {
                    curAct = idleActs[curAct].NextAct;
                    curState = State.NONE;
                } }
            break;
        }
    }
}
```

후기

Git이나 구글링을 통해 외국의 정말 좋은! 코드들을 많이 접할 수 있었고 구조 설계의 롤모델이 되었음

클린한 코드를 위해서 다양한 패턴을 알아두는 것도 좋다고 생각했음

(상태머신 패턴을 알았다면 시간이 많이 단축되었을 것)

개인적으론 1인 플젝(TimeAttack)을 하던 때에 비해 게임의 장르와 기능이 명확하고 에셋이 제공되어 있어서 개발하는데 더 집중할 수 있어서 좋았음