2주차 알파

제목 프로메테우스 미션
분류 시뮬레이션
작성자 김호곤

가. 진행 상황

1. 테크트리 노드는 ScriptableObject로 만들었다.

   테크트리 종류는 크게 시설, 사회, 상용화 연구, 사상 연구가 있고, 각 테크트리 안에 세부적인 테크트리로 나누어져있다. 각 테크트리의 노드 정보는 TechTrees 클래스 안에 Node라는 이름의 내부클래스로 만들어졌다. TechTrees 클래스는 ScriptableObject를 상속받아서 Unity 에디터에서 편집 가능한 파일로 만들 수 있다. 에디터에서 얼마든지 추가, 삭제, 수정이 가능하므로 편집에 용이하다.

```
public class TechTrees : ScriptableObject
{
    [Serializable]
    public class Node
    {
        public string NodeName;
        public TechTreeType Type;
        public Vector2 NodePosition;
        public string Description;
        public float ProgressionValue;
        [Header("비용")]
        public ushort FundCost;
        public byte IronCost;
        public byte NukeCost;
        [Header("요구사항")]
        public SubNode[] Requirments;
    }

    [Serializable]
    public struct SubNode
    {
        public string NodeName;
        public TechTreeType Type;

        public SubNode(string nodeName, TechTreeType type)
        {
            NodeName = nodeName;
            Type = type;
        }
```

```
}
```

NodeName은 노드 이름을 입력할 것으로, 이 문자열로 노드를 식별한다.

TechTreeType은 열거형 타입으로, 해당 노드가 어느 테크트리 종류에 속하는지 식별한다.

NodePosition은 해당 노드가 가로, 세로로 몇 번째 순서로 위치할지 결정한다. 다른 노드와 같은 숫자를 입력하면 노드가 겹쳐져서 보이게 된다.

ProgressionValue는 처음에 만들었을 때는 ProgressionPerMonth라고 이름지어서 게임 상에서 한 달에 얼마만큼 진행도가 올라가는지 결정하는 값으로 할 것이었는데, 그 용도로만 사용하기에는 그 용도가 필요한 테크트리는 전체 4개의 테크트리 중 2개밖에 되지 않는다. 현재는 ProgressionValue라고 이름 바꿔서 다목적용 변수로 쓴다.

Requirments는 요구조건을 담을 배열이다. 테크트리이므로 이전 노드가 해당한다. 이전 노드 정보는 Node 클래스 아래에 정의된 SubNode 구조체로 만들어진다. ScriptableObject를 Unity 에디터에서 편집할 때 다른 객체를 참조하는 것은 안 되고 새로운 객체를 생성하는 것만 되므로 SubNode를 새로 만들어 필요한 정보만 담는다. 요구조건의 정보를 담을 SubNode는 요구조건에 해당하는 노드를 식별할 문자열, 테크트리 종류를 식별할 열거형 변수를 가지고 있다. 이 두 가지 정보만 담는 작은 구조므로 클래스 대신 구조체로 정의했다.

```
[SerializeField] private Node[] _facilityNodes = null;
[SerializeField] private Node[] _techNodes = null;
[SerializeField] private Node[] _thoughtNodes = null;
private Dictionary<string, byte> _indexDictionary = new Dictionary<string, byte>();
private List<SubNode>[][] _nextNodes = new List<SubNode>[(int)TechTreeType.TechTreeEnd][];
```

각 테크트리의 노드는 배열에 담기게 된다. 이름은 '테크트리'인데 실제로는 트리의 구조가 아니다. 첫 번째로는 ScriptableObject로 만들 때 각 노드가 다른 노드의 참조를 가질 수 없고, 두 번째로는 서로 다른 테크트리가 서로 얽혀있어서 구조 자체가 트리의 구조가 아니다. SubNode 구조체로 요구조건 정보를 담을 때 어느 테크트리에 속하는지도 담아야 되는 것에서 볼 수 있듯, 요구조건은 반드시 같은 테크트리에 있어야 되는 것은 아니다. 따라서 모든 노드는 각각의 배열로 묶어서 저장하고, 접근하고 싶은 노드가 어느 인덱스에 존재하는지는 _indexDictionary라는 해쉬테이블에서 찾을 수 있다. 노드는 이름으로 식별하므로 문자열을 '키'로 쓰고, 원하는 '값'은 인덱스 번호이므로 '값'은 byte형이다. byte로 설정한 이유는 각 테크트리의 노드 개수가 255개를 넘을 일이 없기 때문이다. 한 테크트리에 노드 개수가 100개만 돼도 매우 많은 숫자고, 현재 테크트리 중 노드 수가 가장 많은 것은 46개에 불과하다. PC 환경에서는 이런 자료형 절약이 무의미할 것이나, 모바일 환경이면 약간이라도 절약이 필요할 수도 있다. _indexDictionary로 어떤 노드에 접근하고 싶을 때 필요한 것은 노드의 이름과 테크트리 종류로, SubNode에 담긴 두 정보로 원하는 노드로 접근할 수 있다.

_nextNodes는 다음 노드 정보를 담을 가변배열의 이중배열이다. 노드에 요구조건만 있고 다음 노드 정보가 없는 이유는 요구조건과 다음 노드 정보는 반드시 서로 일치해야 되는데, 첫 번째로 개발 중 실수가 발생할 수도 있고, 두 번째로 요구조건이 정해졌으면 다음 노드도 이미 정해진 것이기 때문에 일일이 수동으로 입력하는 것은 불필요하기 때문이다. 이 가변배열의 이중배열 중 이중배열의 첫 번째 인덱스는 테크트리의 종류를 담고, 두 번째 인덱스는 노드의 인덱스에 해당한다. 예를 들어 _nextNodes[시설][5]이면 시설의 5번 노드와 대응하는 것이다. 즉, 시설 5번 노드의 다음 노드 정보를 담는 가변배열에 접근하는 것이다. 각 노드의 요구조건은 몇 개가 있을지 모르므로, 다음 노드 정보는 가변배열에 저장된다.

```
private void SetNextNodes(Node[] techTreeNodes)
```

```
    {
        // 노드 등록
        for (byte i = 0; i < techTreeNodes.Length; ++i)
        {
            Node node = techTreeNodes[i];

            for (byte j = 0; j < node.Requirments.Length; ++j)
            {
                SubNode subNode = node.Requirments[j];
                byte type = (byte)subNode.Type;
                byte index = _indexDictionary[subNode.NodeName];

                // 가변 배열 생성한 적 없으면 새로 생성
                if (null == _nextNodes[type][index])
                {
                    _nextNodes[type][index] = new List<SubNode>();
                }

                // 다음 노드로 등록
                _nextNodes[type][index].Add(new SubNode(node.NodeName, node.Type));
            }
        }
    }
```

위는 한 테크트리의 노드 배열 전체를 순화하면서 '다음 노드'를 등록하는 함수다. 어떤 한 노드가 선택됐을 때 그 노드의 요구조건에 해당하는 노드 인덱스를 찾아가 그곳에 자신을 '다음 노드'로 등록하는 방식이다. 요구조건에 해당하는 노드 인덱스를 찾아갈 때 필요한 것은 위에서 서술한대로 테크트리 종류와 노드의 이름을 '키'로 해서 찾은 인덱스 '값'이다. 따라서 이 함수가 호출되기 전에 _indexDictionary 해쉬테이블에 이미 노드가 등록되어있어야 된다.

2. 여러 테크트리에서 공통된 기능은 추상 클래스에 모았다.

테크트리는 한두 가지가 아니고 여러 가지를 만들 거을 생각하기 때문에 생산성 높은 코드를 만드는 것을 목표로 했다. 따라서 추상 클래스를 부모 클래스로 만들어 공통된 기능은 부모 클래스에서 정의하고 개별적인 기능은 추상함수로 만들어서 자식 클래스에서 정의한다. 그러면 공통된 기능을 두 번 이상 반복해서 작성할 필요가 없어져서 개발 시간을 단축할 수 있을 것으로 기대한다.

```
public abstract class TechTreeViewBase : MonoBehaviour, IState
{
    /* ====================== Variables ====================== */

    [Header("부모클래스")]
    [Header("노드 간격")]
    [SerializeField] private float _width = 100.0f;
    [SerializeField] private float _height = 100.0f;
    [SerializeField] private bool _yCenterize = false;
```

```
[Header("참조")]
[SerializeField] protected TMP_Text AdoptBtn = null;
[SerializeField] protected TMP_Text GainsText = null;
[SerializeField] protected TMP_Text StatusText = null;
[SerializeField] protected GameObject NodeObject = null;
[SerializeField] private TMP_Text _costsText = null;
[SerializeField] private TMP_Text _backBtn = null;
[SerializeField] private TMP_Text _descriptionText = null;
[SerializeField] private Image _progressionImage = null;
[SerializeField] private Transform _techTreeContentArea = null;
[SerializeField] private GameObject _cursor = null;

protected List<TechTrees.SubNode>[] NextNodes = null;
protected Dictionary<string, byte> NodeIndex = null;
protected TechTrees.Node[] NodeData = null;
protected GameObject[] NodeBtnObjects = null;
protected TechTreeNode[] NodeBtns = null;
protected TechTrees TechTreeData = null;
protected byte CurrentNode = 0;
protected float[][] Adopted = null;
protected bool IsAdoptAvailable = false;
protected bool IsBackAvailable = true;
private Transform _cursorTransform = null;
private float _supportRate = 0.0f;
private float _timer = 0.0f;
private bool _runAdoptProgression = false;
```

공통으로 필요한 변수는 위와 같다. [SerializeField]로 변수를 에디터 인스펙터 창에 노출시킬 때 무슨 목적의 변수인지 구분하기 위해 위에 [Header]를 추가해서 구분했다. protected 변수는 private 변수와 이름 규칙을 다르게 해서 자식 클래스에서 변수를 쓸 때 부모 클래스로부터 상속받은 변수인지 구분할 수 있게 했다.

```
public void SetCurrentNode(byte current, Vector3 position)
{
    // 애니메이션 진행 중에는 작동 불가
    if (_runAdoptProgression)
    {
        return;
    }

    // 현재 노드 정보
    CurrentNode = current;

    // 커서 위치
    _cursorTransform.position = position;
    if (!_cursor.activeSelf)
```

```
        {
            _cursor.SetActive(true);
        }

        // 비용 확인
        SetAdoptButtonAvailable(IsUnadopted() && CostAvailable());

        // 설명 텍스트 업데이트
        _descriptionText.text                                                    =
$"[{NodeData[CurrentNode].NodeName}]\n{NodeData[CurrentNode].Description}";

        // 비용 텍스트 업데이트
        _costsText.text = GetCostText();

        // 획득 텍스트 업데이트
        GainsText.text = GetGainText();

        // 상태 메세지 제거
        StatusText.text = null;
    }
```

위는 테크트리 화면에서 한 노드를 클릭하면 호출될 함수다. 매개변수 중 current는 클릭된 노드의 인덱스 번호를 받을 것이고, position은 클릭된 노드의 위치를 받을 것이다.

_runAdoptProgression은 어떤 노드를 승인했을 때의 애니메이션을 진행할지 여부를 나타내는 bool형 변수다. 이 변수가 true면 '승인 중'에 해당하므로 노드 클릭이 동작하지 않게 하기 위해 함수를 바로 반환한다.

커서 오브젝트는 테크트리 창을 비활성화할 때마다 같이 비활성화할 것이므로 노드 선택 시 커서의 활성화 여부를 확인하고, 커서가 비활성화면 활성화한다. _cursor는 커서의 GameObject를 참조하고 _cursorTransform은 커서의 Transform을 참조한다. GameObject와 Transform를 참조하는 변수를 따로 만든 것은 참조 회수를 줄이기 위함이다.

```
    protected void BasicInitialize(byte length)
    {
        // 참조
        NodeIndex = TechTreeData.GetIndexDictionary();
        Adopted = PlayManager.Instance.GetAdoptedData();
        _cursorTransform = _cursor.transform;

        // 승인 버튼 사용 불가
        AdoptBtn.color = Constants.TEXT_BUTTON_DISABLE;

        // 배열 생성
        NodeBtns = new TechTreeNode[length];
        NodeBtnObjects = new GameObject[length];

        // 노드 배치
```

```
        float sizeX = 0.0f;
        float sizeY = 0.0f;
        for (byte i = 0; i < length; ++i)
        {
            // 위치 계산
            float posX = (NodeData[i].NodePosition.x + 0.5f) * _width;
            float posY = (NodeData[i].NodePosition.y + 0.5f) * _height;

            // 노드 생성 후 위치 조정
            NodeBtnObjects[i] = Instantiate(NodeObject, _techTreeContentArea);
            NodeBtnObjects[i].transform.localPosition = new Vector3(posX, posY, 0.0f);

            // 노드 참조
            NodeBtns[i] = NodeBtnObjects[i].GetComponent<TechTreeNode>();

            // 노드 초기화
            NodeBtns[i].SetTechTree(this, i);

            // x, y 최대 값
            if (posX > sizeX)
            {
                sizeX = posX;
            }
            if (posY > sizeY)
            {
                sizeY = posY;
            }
        }

        // 전체 크기
        float areaWidth = sizeX + _width * 0.5f;
        float areaHeight = sizeY + _height * 0.5f;
        RectTransform contentArea = _techTreeContentArea.GetComponent<RectTransform>();
        contentArea.SetSizeWithCurrentAnchors(RectTransform.Axis.Horizontal, areaWidth);
        contentArea.SetSizeWithCurrentAnchors(RectTransform.Axis.Vertical, areaHeight);

        // 가운데 정렬
        float pivotX = 0.0f;
        float pivotY = 0.0f;
        if (Constants.TECHTREE_AREA_WIDTH > areaWidth)
        {
            pivotX = (Constants.TECHTREE_AREA_WIDTH_CENTER * Constants.TECHTREE_AREA_WIDTH - areaWidth * 0.5f) / (Constants.TECHTREE_AREA_WIDTH - areaWidth);
        }
```

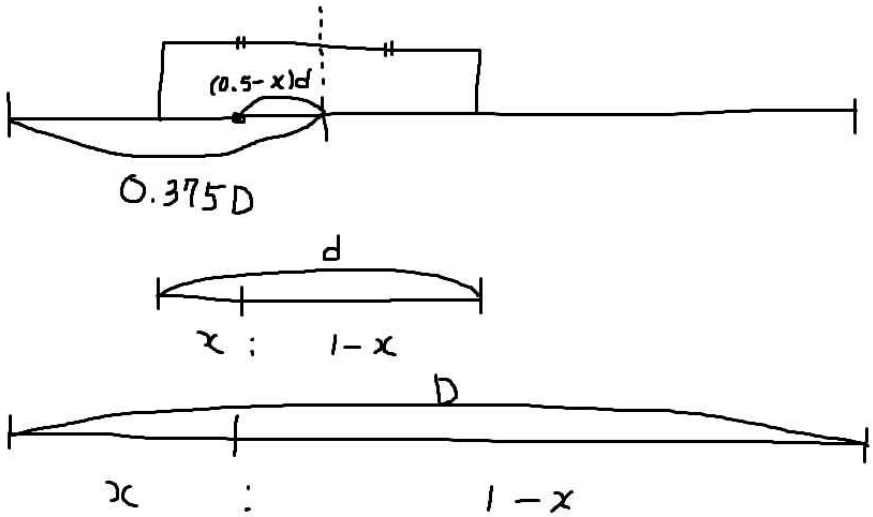
```
        if (_yCenterize && Constants.TECHTREE_AREA_HEIGHT > areaHeight)
        {
            pivotY = (Constants.TECHTREE_AREA_HEIGHT_CENTER * Constants.TECHTREE_AREA_HEIGHT - areaHeight * 0.5f) / (Constants.TECHTREE_AREA_HEIGHT - areaHeight);
        }
        contentArea.pivot = new Vector2(pivotX, pivotY);
    }
```

위는 자식 클래스가 Awake 함수에서 호출할 함수다. 매개변수 length는 해당 테크트리의 노드 개수다.

노드를 배치할 때 등록된 노드 개수만큼 반복문을 실행하는데, 반복문의 한 주기에서는 먼저 노드 오브젝트를 생성하고 위치를 조정한다. 노드 간격은 노드 정보로 입력했던 가로, 세로 위치 순서와 위에서 설정된 _width, _height를 적용하는데, 노드 오브젝트의 pivot은 중앙에 위치하기 때문에 노드 간격의 0.5배만큼을 더해 보기 좋은 위치를 계산한다.

노드는 ScrollView에 생성되는데, 노드가 들어있는 오브젝트의 크기 조정이 필요하다. 가장 처음에 오는 노드는 가장자리에서부터 노드 간격의 0.5배만큼 떨어져있으므로 가장 끝 노드와 끝쪽 가장자리 간격도 노드 간격의 0.5배로 통일하는 것으로 계산했다. 이는 가로, 세로 모두 적용한다.

전체 크기를 계산했으면 가운데 정렬이 필요하다. 화면 상에서 ScrollView는 한쪽으로 치우쳐져있고, 전체 크기가 SrollView보다 크면 한쪽으로 치우쳐진 ScrollView에 꽉 채워서 표시하면 되는데, ScrollView보다 작으면 한쪽으로 치우쳐진 것보다 가운데로 정렬하는 것이 보기 좋다. ScrollView안에 표시되는 내용물은 pivot에 따라 정렬이 달라지는 것을 발견했다. 이를 이용해서 pivot을 조절해 내용물을 가운데 정렬한다.



ScrollView 크기를 D라고 하고 노드 전체 크기를 d라고 할 때, ScrollView 크기는 정해져있고 ScrollView의 화면상 가운에 위치 비율 또한 정해져있다. 따라서 ScrollView의 좌측 가장자리에서부터 화면상 가운데 위치까지 거리는 0.375D다. 전체 노드는 화면 중앙으로 정렬할 것이기 때문에 화면 중앙을 기준으로 전체 노드의 좌측, 우측 길이는 같다. 여기서 pivot 값을 x라고 할 때, 다음과 같은 식이 성립한다.

0.375 * D - (0.5 - x) * d = x * D

이 식을 x에 대해서 나타내면 아래 식이 된다.

x = (0.375 * D - 0.5 * d) / (D - d)

이 식과 위의 코드를 대응시키면 다음과 같다.

D = Constants.TECHTREE_AREA_HEIGHT

0.375 = Constants.TECHTREE_AREA_WIDTH_CENTER

d = areaWidth

세로로 가운데 정렬할 때도 같은 식을 이용하면 된다. 다만, 세로로 가운데 정렬하는 것은 필요한 경우가 있고 필요 없는 경우가 있기 때문에 _yCenterize라는 bool 값으로 세로를 가운데 정렬할 것인지 결정한다. _yCenterize는 [SerializeField]를 통해 Unity 에디터의 인스펙터 창에서 필요하면 체크할 수 있다.

```
/// <summary>
/// 획득 텍스트 생성한다.
/// </summary>
protected abstract string GetGainText();


/// <summary>
/// 승인 성공 시
/// </summary>
protected abstract void OnAdopt();


/// <summary>
/// 승인 실패 시
/// </summary>
protected abstract void OnFail();


/// <summary>
/// 승인 대기 상태인지 확인
/// </summary>
protected abstract bool IsUnadopted();
```

위 함수는 자식 클래스마다 동작이 상이하므로 추상 함수로 만들었다. 위 함수의 동작은 자식 클래스에서 정의된다.

```
private void Update()
{
    if (_runAdoptProgression)
    {
        // 애니메이션 진행
        _timer += Time.deltaTime;
        _progressionImage.fillAmount = _timer;

        // 애니메이션 완료
        if (1.0f <= _timer)
        {
            if (_supportRate >= Random.Range(0.0f, 100.0f))
            {
```

```
                    // 소리 재생
                    AudioManager.Instance.PlayAuido(AudioType.Select);

                    // 성공 시 동작
                    OnAdopt();
                }
                else
                {
                    // 소리 재생
                    AudioManager.Instance.PlayAuido(AudioType.Failed);

                    // 실패 시 동작
                    OnFail();

                    // 상태 메세지
                    StatusText.color = Constants.FAIL_TEXT;
                    StatusText.text = Language.Instance["정책 실패"];

                    // 비용 확인 후 승인 버튼 다시 활성화
                    SetAdoptButtonAvailable(CostAvailable());
                }

                // 복귀
                _progressionImage.fillAmount = 0.0f;
                _runAdoptProgression = false;
                _timer = 0.0f;

                // 뒤로가기 가능
                IsBackAvailable = true;
                _backBtn.color = Constants.WHITE;
            }
        }
    }
```

위 Update 함수에서는 승인 애니메이션을 진행한다. 승인의 성공, 실패 확률은 지지율에 따라 결정되게 할 것이므로 Unity에서 제공하는 Random을 이용해서 무작위 값보다 지지율 값이 더 크면 승인에 성공하게 했다. 애니메이션 동작은 앞서 언급했던 _runAdoptProgression가 참이기만 하면 애니메이션을 시작할 수 있다. 따라서 애니메이션 시작은 아래 함수로 간단하게 시작할 수 있다.

```
    protected void AdoptAnimation(float supportRate)
    {
        _runAdoptProgression = true;
        _supportRate = supportRate;
    }
```

_runAdoptProgression에 참을 저장하고 _supportRate에 지지율 값만 전달하면 애니메이션을 시작할 수 있다.

3. 테크트리 화면의 동작은 다형성을 시도했다.

테크트리 화면은 ScrollView를 제외하면 모두 같은 형태다. 따라서 ScrollView만 다수로 만들고 나머지 UI는 같은 것을 공유한다. 그 때문에 다형성이 필요하다. 여기서는 테크트리에 상태 패턴을 적용하기 위한 IState 인터페이스를 상속받았다. 그 덕분에 전체 창에 해당하는 PopUpScreenTechTree의 동작은 같으면서 다른 화면 결과를 보여줄 수 있다.

```
public class PopUpScreenTechTree : MonoBehaviour
{
    /* ====================== Variables ====================== */

    [SerializeField] private TechTreeViewBase[] _techTreeView = new TechTreeViewBase[(int)TechTreeType.TechTreeEnd];

    private byte _currentTechTree = 0;



    /* ====================== Public Methods ====================== */

    /// <summary>
    /// 테크트리 화면 활성화
    /// </summary>
    public void ActiveThis(TechTreeType techTreeType)
    {
        // 현재 테크트리 변경
        _currentTechTree = (byte)techTreeType;

        // 현재 테크트리 실행
        _techTreeView[_currentTechTree].Execute();

        // 전체 창 연다.
        gameObject.SetActive(true);
    }


    public void BtnAdopt()
    {
        // 다형성
        _techTreeView[_currentTechTree].BtnAdopt();
    }


    public void BtnBack()
    {
        // 다형성
```

```
        _techTreeView[_currentTechTree].ChangeState();

        // 전체 창 닫는다.
        gameObject.SetActive(false);
    }
```

_techTreeView에는 테크트리를 부모 클래스 형식으로 담는다. 그중 몇 번째 인덱스를 사용할 것인지는 _currentTechTree에 저장한다. 이 클래스에서 하는 동작은 Excute, ChangeSate, BtnAdopt를 호출하는 것뿐인데, _currentTechTree가 무엇이냐에 따라 나타나는 화면이 달라진다.

나. 개선할 점

테크트리 제작 목적은 생산성 높은 코드를 작성하는 것이다. 결과적으로 전체적인 틀이 만들어지고, 새로운 테크트리를 추가하더라도 필요한 것만 작성하면 된다. 그러나 전체적인 틀을 만드는데까지 상당한 시간이 결렸기 때문에 사실상 '생산성 높은 코드'라는 의미가 없어졌다. 첫 번째로 기획 단계에서 테크트리 UI나 노드 형태, 노드의 동작 등 구체적인 작동 방식을 정하지 않아 작성 중 즉흥적으로 구현한 것이 대다수였고, 두 번째로는 대부분 즉흥적으로 구현하다보니 테크트리 간 공통 동작이 생각했던 것만큼 많지 않았다. 특히 사회 테크트리는 다른 모든 테크트리와는 매우 상이해서 공통 동작을 모아둔 추상 클래스를 상속받지 못했다. 다음에는 어떤 기능을 구현할 때 구체적으로 무엇이 공통 기능이고 개별 동작인지 구분할 필요가 있다.

다. 다음 계획

1. 테크트리 제작이 예상보다 오래 걸렸다. 사회 테크트리는 미완성이므로 다음 주 초중으로 완성이 목표다.
2. 기획이 구체적이지 않아 대부분 기능은 즉흥적으로 지어내게 될 것으로 보인다. 생산성 높은 코드를 작성하는 것은 무엇을 구현할지 확실할 때 빠르게 작성할 수 있을 것으로 생각되므로 지금은 기능에 집중하고 생산성은 마지막 주차에 시도하는 것으로 전환한다.