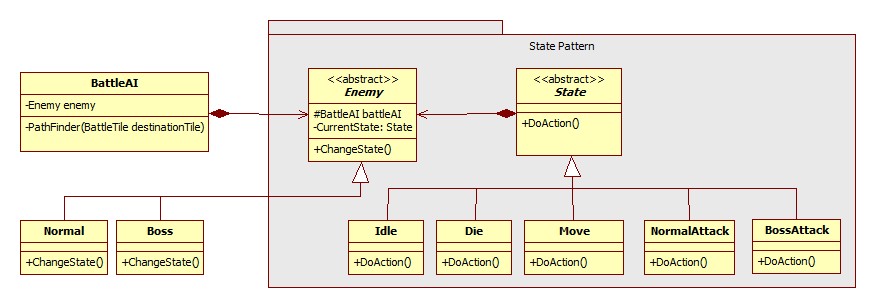
게임프로그래밍 포트폴리오

최대원

|  |  |  |  |
| --- | --- | --- | --- |
| **회사명** | 인하대학교 미래인재개발원 문화컨텐츠 프로그래밍 개발 과정 | | |
| **프로젝트 명** | The RedSettler | **프로젝트 기간** | 18.07 ~ 18.08 (2개월) |
| **프로젝트 상용화** | 아니오 | **담당 역할** | 프레임워크 일부, 몬스터, 몬스터AI |
| **사용 툴 / 언어** | C#, Unity 5.6.5f | | |
| **내용** | 개발 인원 : 9명 (기획1명 + 프로그램 8명)  게임 장르 : 보드 게임  게임 소개 : 보드게임 카탄을 기반으로 턴제 보드 게임과 실시간 전투를 혼합하여 전략적인 운영과 실시간 전투 지원, 형상 관리 툴은 Github와 Sourcetree 사용, 일정 관리 툴은 Jira 사용  담당 업무 :   * 몬스터와 몬스터의 AI설계와 구현 * 프레임워크의 타임 매니저 설계와 구현 | | |

**디자인 패턴 : 상태 패턴(State Pattern)**을 응용하여 몬스터로 사용되는 Enemy Class를 설계하였습니다.

상태 패턴은 하나의 클래스가 하나의 상태만을 갖게 하여 각 상태에 따라 런타임 중에 클래스의 행동을 변경하는 패턴입니다.

  
Enemy : 몬스터들의 추상 클래스입니다. 생성될 때 BattleAI 클래스를 생성시켜 자기 자신을 참조할 수 있게 합니다.

BattleAI : 몬스터의 구체적인 행동 제어를 책임지는 클래스입니다.

Boss : Enemy의 하위 클래스 중에 보스 몬스터 객체입니다.

Normal : Enemy의 하위 클래스 중에 일반 몬스터의 객체입니다.

State : Enemy가 가질 수 있는 상태들의 추상 클래스입니다.

포트폴리오에는 Boss의 ChangeState()와 BattleAI의 PathFinder()를 첨부했습니다.

ChangeState() : 보스 몬스터의 상태를 바꾸는 메서드입니다. 이 메서드는 BattleAI와 유니티의 애니메이션 이벤트를 통해 호출됩니다.

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64 | //상태를 변경하는 메서드  public override void ChangeState(EnemyStateType stateType)  {      switch (stateType)      {          //각 상태로 전이되어 행동을 취하기 위해서는 Enemy의 각종 요소들을 참조할 수 있어야 하는데          //상태 패턴에서는 각 상태가 캡슐화에 의해 객체를 알 수 없습니다.          //객체를 모르는 상황에서 객체의 행동을 제어하기 위해 제어에 필요한 요소들만 생성자를          //이용해 넘겨주는 방법을 선택했습니다.          case EnemyStateType.Idle:              currentState = new Idle(animator, rigidbodyComponent);              break;            case EnemyStateType.Die:              currentState = new Die(animator, DeadTimer, TimeToReturn, new DeadTimerCallback(EndDead));              break;            case EnemyStateType.Damage:              currentState = new Damage(animator);              break;            case EnemyStateType.Attack1:              if (isAttackable[0])              {                  currentState = new Boss.Attack(animator, bossPhase, fireballLifeTimer,                      new TimerCallback(PushFireballTimer), TargetObject, transform, TimeToReturn,                      ObjectPoolManager.Instance.FireballQueue, FireBallSpeed,  LaunchedFireballList,                      (int)Power);                    isAttackable[0] = false;                  //GameTimeManager : 게임의 시간 및 타이머를 관리하는 싱글톤 클래스입니다.                  //공격을 연속으로 하지 못하도록 대기 시간을 타이머를 통해 지정했습니다.                  //타이머 내부 Callback을 통해 시간이 다 되었을 때 호출할 메서드를 지정했습니다.                  Pattern1Timer = GameTimeManager.Instance.PopTimer();                  Pattern1Timer.SetTimer(attack1Tick, false);                  Pattern1Timer.Callback = new TimerCallback(SetAttackable1);                  Pattern1Timer.StartTimer();              }              break;            case EnemyStateType.Attack2:              if (isAttackable[1])              {                  currentState = new Boss.UseSkill(animator, explode, explodeLifeTimer,                      Power, explodeLifeTime, new TimerCallback(UseSkillEnd), bossPhase);                    isAttackable[1] = false;                  Pattern1Timer = GameTimeManager.Instance.PopTimer();                  Pattern1Timer.SetTimer(attack2Tick, false);                  Pattern1Timer.Callback = new TimerCallback(SetAttackable2);                  Pattern1Timer.StartTimer();              }              break;            case EnemyStateType.Move:              currentState = new Move(animator, transform, rigidbodyComponent, destinationPoint,              currentPoint, MoveSpeed, currentTile);              break;            default:              break;          }          base.ChangeState(stateType);  } | [cs](http://colorscripter.com/info#e) |

**A\*알고리즘 응용 : 길찾기 구현**

PathFinder() : BattleAI의 길 찾기 메서드입니다. 육각 타일 기반의 레벨에서 작동할 수 있는 A\*알고리즘을 적용했습니다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84 | //A\* 알고리즘  private void PathFinder(BattleTile destinationTile)  {      //openSet : 이동 가능한 노드들의 후보들이 들어있는 리스트      //closedSet : 이동 불가능한 노드들의 후보들이 들어있는 리스트      openSet.Clear();      closedSet.Clear();        //시작 타일과 목적지 타일을 설정      currnetTile = enemy.GetCurrentTile(enemyPosition);      startTile = currnetTile;      startTile.ParentTileXCoord = currnetTile.TileCoordinate.x;      startTile.ParentTileZCoord = currnetTile.TileCoordinate.z;      endTile = destinationTile;        do      {          closedSet.Add(currnetTile);            for (int i = 0; i < coordX.Length; i++)          {              //인근 타일을 검색합니다.              BattleTile battleTile = SearchAdjacentTiles(i);                if (battleTile != null)              {                  if (battleTile.isWall)                  {                      closedSet.Add(battleTile);                      continue;                  }                    //인근 타일의 경로 비용(g)과 휴리스틱(h)를 멘해튼 거리로 계산합니다.                  battleTile.g = Mathf.Abs(startTile.TileCoordinate.x - battleTile.TileCoordinate.x)                       + Mathf.Abs(startTile.TileCoordinate.z - battleTile.TileCoordinate.z);                  battleTile.h = Mathf.Abs(battleTile.TileCoordinate.x - endTile.TileCoordinate.x)                       + Mathf.Abs(battleTile.TileCoordinate.z - endTile.TileCoordinate.z);                    //해당 타일의 경로 비용이 현재 타일의 경로 비용보다 적은 경우                  //이 타일이 반대 방향으로 가는 길일수도 있기 때문에                  //가중치(현재 노드의 g비용)를 더해 줍니다.                  int tempG = 0;                  if (battleTile.g < currnetTile.g)                  {                      tempG = currnetTile.g;                  }                  battleTile.f = battleTile.g + battleTile.h + tempG;                    //인접 타일 중 이동 가능한 후보가 되는 타일들은 오픈셋에 추가합니다.                  //이때 현재 타일이 이 인접 타일들의 부모 노드가 됩니다.                  if (!closedSet.Contains(battleTile) && !openSet.Contains(battleTile))                  {                      battleTile.ParentTileXCoord = currnetTile.TileCoordinate.x;                      battleTile.ParentTileZCoord = currnetTile.TileCoordinate.z;                      openSet.Add(battleTile);                  }                    //오픈셋에 추가함과 동시에 바로 비용 합계(f)를 기준으로 오름차순 정렬합니다.                  if (openSet.Count > 1)                  {                      openSet.Sort(delegate (BattleTile a, BattleTile b)                      {                          if (a.f > b.f) return 1;                          else if (a.f < b.f) return -1;                          return 0;                      });                  }              }          }            //인접 타일들 모두 탐색했다면 오픈셋에서 가장 적은 비용을 가진 노드를 골라냅니다.          //이 노드는 다음 이동 할 타일이 됩니다.          if (openSet.Count > 0)          {              currnetTile = openSet[0];              openSet.Remove(currnetTile);          }          else return;      }      while (currnetTile != endTile);       //목적지에 도착하면 while문이 종료됩니다.       //마지막 노드로부터 각 노드의 부모 노드를 따라 최단 경로를 구성합니다.      pathTile = CreateParh(startTile);  } |