C++프로그래밍및실습[4]

2D 게임 엔진 및 강화 학습 에이전트 개발

최종 보고서

제출일자: 2024-12-22
제출자명: 조영진
제출자학번: 203342

1. 프로젝트 목표

1) 배경 및 필요성

게임 내 에이전트(NPC)는 게임 세계에서 플레이어와 상호작용하며 몰입감을 높이는 주요 요소로, 게임의 꽃이라고 할 만큼 중요한 역할을 맡고 있습니다. 현재 강화학습 기술이 발전함에 따라 에이전트에 강화학습을 적용시키려는 시도가 증가하고 있습니다. 강화학습을 통한 에이전트의 학습 방식은 기존의 규칙 기반 에이전트보다 더 사람 같고, 최적화된 행동을 보여주어, 사용자의 사용 경험을 향상시킬 것으로 기대됩니다.

그러나 기존의 상용화된 강화학습 에이전트는 높은 복잡성과 많은 자원을 필요로 하여, 개발자에게 진입 장벽을 높이는 요인으로 작용합니다. 이에 따라 본 프로젝트는 간단한 2D 게임엔진과 강화학습 알고리즘을 직접 구현하여, 에이전트가 스스로 게임 환경에서 학습하고 적응할 수 있는 프로토타입을 만드는 데 목적이 있습니다.

2) 프로젝트 목표

본 프로젝트의 목표는 강화학습 기반의 에이전트가 간단한 2D 게임 엔진 환경에서 행동을 학습하고, 주어진 목표를 달성할 수 있도록 하는 것입니다. 이를 통해 게임과 에이전트 학습 사이의 연관성을 탐색하고, 엔진과 학습 알고리즘의 상호작용을 탐구하는 것이 주된 목표입니다.

3) 차별점

기존의 게임 에이전트는 보통 규칙 기반으로 설계되거나, 특정 행동 패턴을 미리 정의한 후 이에 따라 작동하도록 설정됩니다. 본 프로젝트의 에이전트는 강화학습을 통해 다양한 게임 내 환경에 따른 최적의 행동을 취할 수 있는 점에서 차별화됩니다.

2. 기능 계획

1) 2D 게임 엔진 구현

- 강화학습 에이전트 학습 환경 제공을 위한 간단한 2D 게임 엔진 구현

(1) 충돌 감지 시스템

- 에이전트가 장애물이나 적에 부딪힐 경우 이를 감지해 행동을 학습하거나 방향을 조정하도록 설정

    - 에이전트와 객체간 충돌 체크 기능

    - 충돌 시 에이전트 학습 진행 및 초기화

(2) 중력 및 이동 시스템

- 게임 내의 중력 효과와 상하좌우 이동 기능을 구현하여 움직임을 구현

    - 중력 구현

    - 키 입력에 따른 이동처리

    - 에이전트의 이동 자동화

(3) 환경 시각화

- 게임 내 환경을 시각적으로 표시해 학습 과정을 볼 수 있도록 함.

    - 시각화를 위한 애니메이션 클래스

2) Q-Learning 강화학습 알고리즘 구현

- Q-Learning 알고리즘을 사용해 에이전트가 2D 게임 환경에서 자율적으로 학습할 수 있도록 구현

(1) 상태 공간 정의

- 에이전트가 속한 공간의 상태를 위치(Vector) 적과의 거리(Vector)로 정의하여 학습을 위한 상태 공간 정의

    - 상태 공간에 속할 변수와 범위 정의

(2) 행동 및 보상 시스템 설정

- 에이전트가 할 수 있는 행동과, 행동에 대한 보상을 정의

    - 8방면 이동을 행동으로 정의

    - 적과 멀어질수록 많은 보상

    - 적과 충돌하면 큰 패널티

(3) Q-Learning 알고리즘 구현

- Q-Learning 알고리즘을 통해 에이전트가 주어진 환경에서 최선의 행동을 학습하게 함.

    - Q 테이블 초기화 및 갱신 함수 구현

    - 모험률, 패널티 등 하이퍼파라미터 설정

    - 학습 루프 구현

3. 기능 구현

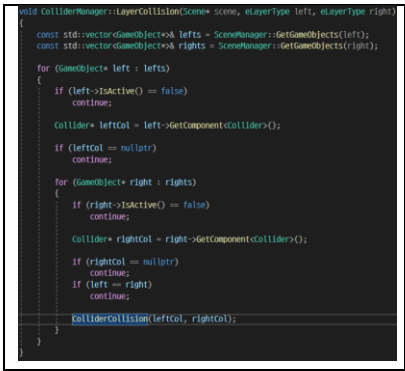
1) 2D 게임 엔진 구현

(1) 충돌 감지 시스템

- Collider 스크립트를 가진 객체끼리 각자의 영역을 충돌했을 때 영역을 계산해 충돌을 감지

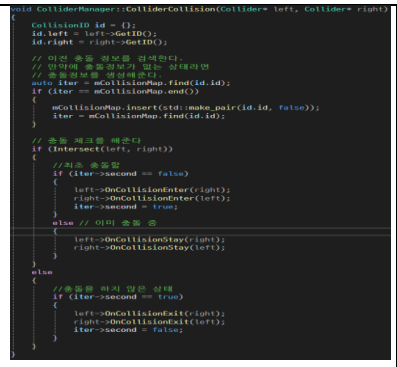
- 적용된 배운 내용: 포인터, 상속, 인터페이스

- 코드 스크린샷 (NewbieEngine_Source/ColliderManager.cpp 내 구현)



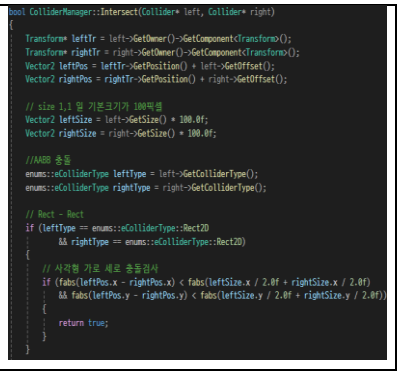
```
void ColliderManager::LayerCollision(Scene* scene, eLayerType left, eLayerType right)
{
    const std::vector<GameObject*>& lefts = SceneManager::GetGameObjects(left);
    const std::vector<GameObject*>& rights = SceneManager::GetGameObjects(right);

    for (GameObject* left : lefts)
    {
        if (left->IsActive() == false)
            continue;

        Collider* leftCol = left->GetComponent<Collider>();

        if (leftCol == nullptr)
            continue;

        for (GameObject* right : rights)
        {
            if (right->IsActive() == false)
                continue;

            Collider* rightCol = right->GetComponent<Collider>();

            if (rightCol == nullptr)
                continue;
            if (left == right)
                continue;

            ColliderCollision(leftCol, rightCol);
        }
    }
}
```

레이어별 충돌을 관리하는



```
void ColliderManager::ColliderCollision(Collider* left, Collider* right)
{
    CollisionID id = {};
    id.left = left->GetID();
    id.right = right->GetID();

    // 이전 충돌 정보를 검색한다.
    // 만약에 충돌정보가 없는 상태라면
    // 충돌정보를 생성해준다.
    auto iter = mCollisionMap.find(id.id);
    if (iter == mCollisionMap.end())
    {
        mCollisionMap.insert(std::make_pair(id.id, false));
        iter = mCollisionMap.find(id.id);
    }

    // 충돌 체크를 해준다
    if (Intersect(left, right))
    {
        //최초 충돌일
        if (iter->second == false)
        {
            left->OnCollisionEnter(right);
            right->OnCollisionEnter(left);
            iter->second = true;
        }
        else // 이미 충돌 중
        {
            left->OnCollisionStay(right);
            right->OnCollisionStay(left);
        }
    }
    else
    {
        //충돌을 하지 않은 상태
        if (iter->second == true)
        {
            left->OnCollisionExit(right);
            right->OnCollisionExit(left);
            iter->second = false;
        }
    }
}
```

Collider별 충돌을 관리하는



```
bool ColliderManager::Intersect(Collider* left, Collider* right)
{
    Transform* leftTr = left->GetOwner()->GetComponent<Transform>();
    Transform* rightTr = right->GetOwner()->GetComponent<Transform>();
    Vector2 leftPos = leftTr->GetPosition() + left->GetOffset();
    Vector2 rightPos = rightTr->GetPosition() + right->GetOffset();

    // size 1,1 일 기본크기가 100픽셀
    Vector2 leftSize = left->GetSize() * 100.0f;
    Vector2 rightSize = right->GetSize() * 100.0f;

    //AABB 충돌
    enums::eColliderType leftType = left->GetColliderType();
    enums::eColliderType rightType = right->GetColliderType();

    // Rect - Rect
    if (leftType == enums::eColliderType::Rect2D
        && rightType == enums::eColliderType::Rect2D)
    {
        // 사각형 가로 세로 충돌검사
        if (fabs(leftPos.x - rightPos.x) < fabs(leftSize.x / 2.0f + rightSize.x / 2.0f)
            && fabs(leftPos.y - rightPos.y) < fabs(leftSize.y / 2.0f + rightSize.y / 2.0f))
        {
            return true;
        }
    }
}
```

충돌을 수학적으로 계산하는

LayerCollision 함수	ColliderCollision 함수	Intersect 함수

(2) 중력 및 이동 시스템

- 중력 구현을 위한 질량, 가속도, 마찰 구현 Rigidbody 클래스 및 이동을 위한 입력 시스템 Input 클래스

- 적용된 배운 내용: 클래스, 상속, 템플릿, static, vector

- 코드 스크린샷 (NewbieEngine_Source/NewbieRigidbody.cpp, NewbieInput.cpp 내 구현)

<table>
<tr>
<td>

```
void Rigidbody::Update()
{
    // f = m * a
    // a = f / m
    mAccelation = mForce / mMass;

    // v = v.prev + acc
    mVelocity += mAccelation * Time::DeltaTime();

    if (mbGround)
    {
        // 땅 위에 있을 때
        Vector2 gravity = mGravity;
        gravity.normalize();

        float dot = Vector2::Dot(mVelocity, gravity);
        mVelocity -= gravity * dot;
    }
    else
    {
        // 공중에 있을 때
        mVelocity += mGravity * Time::DeltaTime();
    }

    // 최대 속도 제한
    Vector2 gravity = mGravity;
    // 단위벡터(방향)
    gravity.normalize();
    // 중력 방향의 가속도 구하기
    float dot = Vector2::Dot(mVelocity, gravity);
    // 중력 방향으로 속도 계산
    gravity = gravity * dot;

    // 중력 방향이 아닌 방향으로의 속도
    Vector2 sideVelocity = mVelocity - gravity;

    // 속도 제한
    if (mLimitedVelocity.y < gravity.length())
    {
        gravity.normalize();
        gravity *= mLimitedVelocity.y;
    }

    if (mLimitedVelocity.x < gravity.length())
    {
        gravity.normalize();
        gravity *= mLimitedVelocity.x;
    }

    mVelocity = gravity + sideVelocity;
```

</td>
<td>

```
if (!(mVelocity == Vector2::Zero))
{
    // 마찰력 작용 구현
    Vector2 friction = -mVelocity;
    friction = friction.normalize() * mFriction * mMass * Time::DeltaTime();

    // 마찰력으로 인한 속도 감소량 > 현재 속도
    if (mVelocity.length() <= friction.length())
    {
        // 멈추기
        mVelocity = Vector2::Zero;
    }
    else
    {
        mVelocity += friction;
    }

    Transform* tr = GetOwner()->GetComponent<Transform>();
    Vector2 pos = tr->GetPosition();
    pos = pos + mVelocity * Time::DeltaTime();
    tr->SetPosition(pos);

    mForce.clear();
}
```

</td>
</tr>
<tr>
<td>Rigidbody 가속도, 최대속도, 중력, 마찰력</td>
<td>구현 및 실제구현</td>
</tr>
</table>

```cpp
int ASCII[(UINT)eKeyCode::End] = {
    'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P',
    'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L',
    'Z', 'X', 'C', 'V', 'B', 'N', 'M',
    VK_LEFT, VK_RIGHT, VK_DOWN, VK_UP,
    VK_LBUTTON, VK_MBUTTON, VK_RBUTTON,
};

void Input::Initialize()
{
    createKeys();
}

void Input::Update()
{
    updateKeys();
}

void Input::createKeys() {
    // vector로 mKeys의 배열을 만들고 각 자판마다의 key클래스를 만들어
    // mKeys에 push함으로 각 키를 객체화해 관리를 쉽게 만듦
    for (size_t i = 0; i < (UINT)eKeyCode::End; i++) {
        Key key = {};
        key.bPressed = false;
        key.state = eKeyState::None;
        key.keyCode = (eKeyCode)i;

        Keys.push_back(key);
    }
}
```

키를 배열로 선언하는 함수

```cpp
void Input::updateKeys()
{
    std::for_each(Keys.begin(), Keys.end(),
        [](Key& key) -> void
        {
            updateKey(key);
        });
}

void Input::updateKey(Input::Key& key)
{
    // 창이 포커싱 되었을 때
    if (GetFocus())
    {
        // 키 눌렸는지 확인 및 업데이트
        if (isKeyDown(key.keyCode))
            updateKeyDown(key);
        else
            updateKeyUp(key);

        // 마우스 포지션 가져오기
        getMousePositionByWindow();
    }
    else
    {
        clearKeys();
    }
}

bool Input::isKeyDown(eKeyCode code)
{
    // 키가 눌렸는지 확인
    return GetAsyncKeyState(ASCII[(UINT)code]) & 0x8000;
}
```

키를 매 프레임마다 update하는 함수

(3) 환경 시각화

- 게임 환경을 시각화하는 Render메서드와 애니메이션, 애니메이터 클래스

- 적용된 배운 내용: 포인터, 인터페이스, 상속, for구문, vector

- 코드 스크린샷 (NewbieEngine_Source/NewbieAnimation, Animator, Application.cpp 내)

```cpp
void Application::Render()
{
    // Rendering 되는 함수
    clearRenderTarget();

    // Time Rendering
    Time::Render(mBackHdc);

    ColliderManager::Render(mBackHdc);


    // GameObject Render도 SceneManager가
    SceneManager::Render(mBackHdc);

    // BackBuffer에 있는걸 원본 Buffer에 복사
    copyRenderTarget(mBackHdc, mHdc);
}
```

프로그램의 High-Level Interface Render 함수 보여줘야 하는 모든 것을 순서대로 렌더링

```cpp
void Application::Run()
{
    Updata();
    LateUpdate();
    Render();
    Destroy();
}
```

Application을 실행하면 반복적으로 호출되는 Run 메서드.

```cpp
void Animation::CreateAnimation(const std::wstring& name, graphics::Texture* spriteSheet
    , Vector2 leftTop, Vector2 size, Vector2 offset, UINT spriteLegth, float duration)
{
    mTexture = spriteSheet;
    for (size_t i = 0; i < spriteLegth; i++)
    {
        Sprite sprite = {};
        sprite.leftTop.x = leftTop.x + (size.x * i);
        sprite.leftTop.y = leftTop.y;
        sprite.size = size;
        sprite.offset = offset;
        sprite.duration = duration;

        mAnimationSheet.push_back(sprite);
    }
}
```

애니메이션을 생성하는 CreateAnimation 메서드

```cpp
void Animation::Update()
{
    if (mbComplete)
        return;

    mTime += Time::DeltaTime();

    if (mAnimationSheet[mIndex].duration < mTime)
    {
        mTime = 0.0f;
        if (mIndex < mAnimationSheet.size() - 1)
            mIndex++;
        else
            mbComplete = true;
    }
}
```

에니메이션을 재생하는 Update 메서드

```cpp
void Animator::CreateAnimation(const std::wstring& name
    , graphics::Texture* spriteSheet
    , Vector2 leftTop
    , Vector2 size, Vector2 offset
    , UINT spriteLegth, float duration)
{
    Animation* animation = nullptr;
    animation = FindAnimation(name);
    if (animation != nullptr)
        return;

    animation = new Animation();
    animation->SetName(name);
    animation->CreateAnimation(name, spriteSheet
        , leftTop, size, offset, spriteLegth, duration);

    animation->SetAnimator(this);

    Events* events = new Events();
    mEvents.insert(std::make_pair(name, events));

    mAnimations.insert(std::make_pair(name, animation));
}
```

애니메이션을 생성하는 CreateAnimation 메서드

```cpp
void Animator::PlayAnimation(const std::wstring& name, bool loop)
{
    Animation* animation = FindAnimation(name);
    if (animation == nullptr)
        return;


    if (mActiveAnimation)
    {
        Events* currentEvents
            = FindEvents(mActiveAnimation->GetName());

        if (currentEvents)
            currentEvents->endEvent();
    }


    Events* nextEvents
        = FindEvents(animation->GetName());

    if (nextEvents)
        nextEvents->startEvent();

    mActiveAnimation = animation;
    mActiveAnimation->Reset();
    mbLoop = loop;
}
```

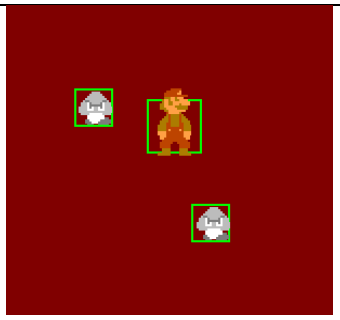
애니메이션 재생하는 PlayAnimation 메서드

2) Q-Learning 강화학습 알고리즘 구현

(1) 상태 공간 정의

- 에이전트 주위에 생성되는 적을 피해 오래 생존하는 것이 목표인 공간

- 스크린샷



상태 공간 예시 스크린샷

```cpp
struct State
{
    Vector2 playerPosition; // 플레이어 위치 [x, y]
    std::vector<Vector2> enemyPositions; // 적 위치 [[x1, y1], [x2, y2], ...]
};

class Environment : public GameObject
{
public:
    Environment();

    void Reset(); // 환경 초기화
    double Step(int action); // 행동 수행 및 보상 반환

    void Initialize() override;
    void Update() override;
    void LateUpdate() override;
    void Render(HDC hdc) override;

    State GetLearnState() { return learnState; }

    // 페널티 관리
    void ApplyPenalty(double penalty); // 페널티 기록
    double GetLastPenalty() const;    // 마지막 페널티 반환

    void RestartGame();

    // 적 삭제 함수
    void DeleteEnemy(GameObject* enemy = nullptr);

private:
    float mTime;
    // 적 생성 함수
    void spawnEnemy();
    State learnState;

    double lastPenalty; // 마지막 페널티 값
};
```

환경을 정의한 클래스

(2) 행동, 보상, 재시작 시스템 설정

- 에이전트가 할 수 있는 8방면 이동과 보상 시스템

- 적용된 배운 내용: if문, 클래스, 상속

- 코드 스크린샷 (NewbieEngine_Window/NewbieEnvironment.cpp, EnemyScript)

```cpp
double Environment::Step(int action) {
    // 플레이어 이동 처리
    if (action == 0) learnState.playerPosition.y -= 100.0f * Time::DeltaTime(); // 위
    else if (action == 1) learnState.playerPosition.y += 100.0f * Time::DeltaTime(); // 아래
    else if (action == 2) learnState.playerPosition.x -= 100.0f * Time::DeltaTime(); // 왼쪽
    else if (action == 3) learnState.playerPosition.x += 100.0f * Time::DeltaTime(); // 오른쪽
    else if (action == 4) { // 좌상
        learnState.playerPosition.x -= 70.7f * Time::DeltaTime();
        learnState.playerPosition.y -= 700.7f * Time::DeltaTime();
    }
    else if (action == 5) { // 우상
        learnState.playerPosition.x += 700.7f * Time::DeltaTime();
        learnState.playerPosition.y -= 700.7f * Time::DeltaTime();
    }
    else if (action == 6) { // 좌하
        learnState.playerPosition.x -= 70.7f * Time::DeltaTime();
        learnState.playerPosition.y += 70.7f * Time::DeltaTime();
    }
    else if (action == 7) { // 우하
        learnState.playerPosition.x += 70.7f * Time::DeltaTime();
        learnState.playerPosition.y += 70.7f * Time::DeltaTime();
    }


    double reward = 1.0; // 기본 생존 보상
    const double distanceScale = 0.1; // 거리 보상 스케일링 계수

    // 적과의 거리 계산 및 보상
    for (auto& enemy : learnState.enemyPositions) {
        double distance = sqrt(pow(learnState.playerPosition.x - enemy.x, 2) + pow(learnState.pl
        reward += distance * distanceScale; // 거리가 멀수록 보상 증가
    }

    // 페널티를 보상에 포함
    reward += lastPenalty;

    // 페널티 초기화 (1회 적용)
    lastPenalty = 0.0;

    return reward;
}
```

에이전트 이동, 보상, 패널티를 관장하는 Environment 클래스의 Step 메서드

```cpp
void Environment::RestartGame()
{
    Scene* activeScene = SceneManager::GetActiveScene();

    // 플레이어 위치 초기화
    Layer* playerLayer = activeScene->GetLayer(enums::eLayerType::Player);
    if (!playerLayer->GetGameObjects().empty()) {
        GameObject* player = playerLayer->GetGameObjects().front();
        Transform* playerTransform = player->GetComponent<Transform>();
        playerTransform->SetPosition(Vector2(500.0f, 500.0f)); // 초기 위치
    }

    DeleteEnemy();

    // 학습 환경 초기화
    Layer* layer = activeScene->GetLayer(enums::eLayerType::BackGround);
    Environment* env = layer->GetEnvironment();
    if (env) {
        env->Reset(); // 환경 상태 초기화
    }
}
```

게임을 재시작하는 Environment클래스의 RestartGame 메서드

(3) Q-Learning 알고리즘 구현

- Q-Table은 map<상태(pair<Vector, action>), reward>로 설정하여 현재 위치에서 최선의 행동을 찾도록 설정. 탐험(랜덤) 행동은 1에서 시작해 0.3까지 0.999씩 지수적으로 감소하도록 설정

- 적용된 배운 내용: 상속, vector, map, for문, if문

- 코드 스크린샷 (NewbieEngine_Window/NewbieQLearningAgent.cpp 내 구현)

```cpp
// 행동 결정
int QLearningAgent::chooseAction(const State& state, double epsilon)
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> dis(0.0, 1.0);

    if (dis(gen) < epsilon) {
        // 탐험: 랜덤 행동
        return rand() % numActions;
    }
    else {
        // 활용: 최적 행동 선택
        int bestAction = 0;
        double maxQ = std::numeric_limits<double>::lowest();
        for (int action = 0; action < numActions; ++action) {
            double qValue = Q[{state.playerPosition, action}];
            if (qValue > maxQ) {
                maxQ = qValue;
                bestAction = action;
            }
        }
        return bestAction;
    }
}
```

에이전트의 행동을 결정하는 chooseAction 메서드

```cpp
void QLearningAgent::updateQValue(const State& currentState, int action, double reward, const State& nextState, doub
{
    double maxNextQ = std::numeric_limits<double>::lowest();
    for (int nextAction = 0; nextAction < numActions; ++nextAction) {
        double qval = getQValue(nextState.playerPosition, nextAction);
        if (maxNextQ < qval)
        {
            maxNextQ = qval;
        }
    }

    double currentQ = getQValue(currentState.playerPosition, action);
    setQValue(currentState.playerPosition, action, currentQ + alpha * (reward + gamma * maxNextQ - currentQ));
}
```

행동 후 Q-Table을 업데이트 하는 메서드 updateQValue

```cpp
void QLearningAgent::Update()
{
    // 매 업데이트마다 초기화
    Scene* scene = SceneManager::GetActiveScene();
    Layer* layer = scene->GetLayer(enums::eLayerType::BackGround);
    Environment* env = layer->GetEnvironment();

    Layer* playerLayer = scene->GetLayer(enums::eLayerType::Player);
    GameObject* player = playerLayer->GetGameObjects().front();

    // 환경 상태 가져오기
    const State& currentState = env->GetLearnState();

    action = chooseAction(learnState, epsilon);

    // epsilon을 지수적으로 감소
    epsilon *= 0.999;  // 0.995씩 곱해서 점차 감소
    if (epsilon < 0.3) {
        epsilon = 0.3;  // 최소 epsilon 값 설정
    }
    double reward = 0.0;

    // reward 계산
    reward = env->Step(action);
    reward += env->GetLastPenalty(); // 페널티를 추가로 반영

    Transform* tr = player->GetComponent<Transform>();

    // 다음 상태 가져오기
    const State& nextState = env->GetLearnState();

    tr->SetPosition(nextState.playerPosition);

    // Q-값 업데이트
    updateQValue(currentState, action, reward, nextState, alpha, gamma);

    learnState = nextState; // 다음 상태로 갱신
}
```

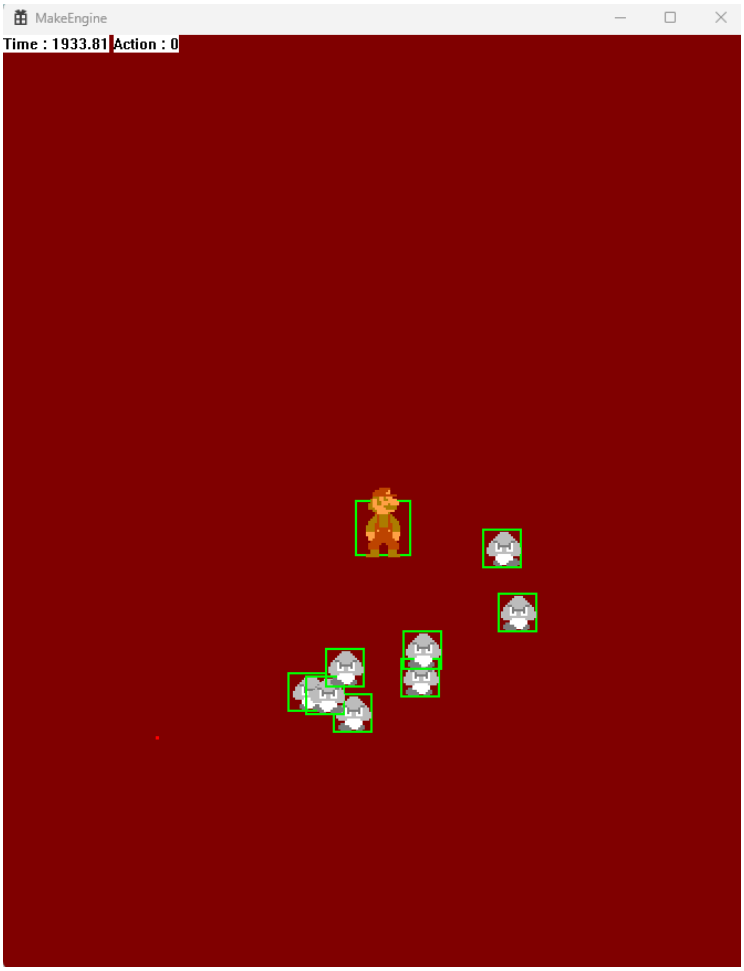
프레임마다 상태를 갱신하고 학습하는 Update 메서드

행동을 정하고 보상을 계산하고 행동하고, Q-table을 업데이트

4. 테스트 결과

1) 2D 게임 엔진 구현

- 충돌 감지, 이동, 환경 시각화 모두 동시 테스트.
    - 충돌 감지: 마리오의 초록 선과 회색 버섯의 초록 선이 만나면 게임이 재시작
    - 중력: 구현은 완료했으나 게임 환경 상 필요한 기능은 아니라 테스트 진행 X
    - 이동: 마리오는 자동으로 Q-Learning에 의해 이동
    - 환경 시각화: 각 오브젝트들이 이동하는 모습을 관찰할 수 있음.
- 테스트 결과 스크린샷
    - 스크린샷만으로 설명이 불가해 영상자료 링크 첨부
    - https://youtube.com/shorts/FnyRvSznltQ?feature=share



2) Q-Learning 강화학습 알고리즘 구현

- 게임 환경, 학습, 보상, 재시작 동시에 테스트
    - 게임 환경: 위의 게임 엔진에서 테스트 완료
    - 학습 및 보상: 처음에 위로만 가던 에이전트가 점차 좌상단으로 이동
    - 재시작 시스템: 적과 닿으면 환경을 초기화하고 재시작
- 테스트 결과 영상
    - 영상 자료 첨부
    - https://youtube.com/shorts/oe7gzirFSBE?feature=share

5. 계획 대비 변경 사항

1) 학습 데이터 분석 및 시각화

- 보상 기록 및 분석, 성공률 시각화
- 프로젝트에서 제외
- 개발 지체 및 시간 부족

6. 배운 점 & 느낀 점

1) 배운 점

- 디자인 패턴의 아름다움

이번 프로젝트를 통해 디자인 패턴이 코드의 가독성과 이해도를 높이는 데 얼마나 중요한 역할을 하는지 알게 되었습니다. 그동안 주먹구구식으로 프로젝트를 진행하며 큰 틀에서의 설계보다는 구현에만 치중했던 제 방식에 대해 돌아볼 수 있었습니다. 게임 엔진을 개발하면서 참고 자료를 활용하긴 했지만, 이를 통해 코드를 체계적으로 추상화하고 디자인하는 것이 얼마나 중요한지 배웠습니다. 개발은 단순히 코드를 잘 작성하는 것이 아니라 전체적인 구조와 설계가 뒷받침되어야 한다는 점을 깨달았습니다.

- 해내고자 하는 마음

이 프로젝트는 제게 도전적인 과제였으며, 처음부터 끝까지 포기하지 않고 진행해 온 점이 특히 의미 있었습니다. 진행 중에 "내가 과연 이걸 할 수 있을까?"라는 의문이 들 정도로 부담감이 컸지만, 하루하루 꾸준히 문제를 해결해나가며 결국 목표한 결과를 이뤄냈습니다. 이번 경험을 통해 포기하지 않는 자세와 꾸준함이 어떤 일이든 해낼 수 있다는 자신감을 심어주었습니다.

2) 느낀 점

- 자기객관화

1.5개월이라는 기간 동안 도전적인 프로젝트를 찾던 중 게임 엔진과 강화학습 주제가 흥미로워 보였습니다. 하지만 학업과 병행하며 이 프로젝트를 진행하는 것이 얼마나 어려운 일인지 과소평가했던 것 같습니다. 매일 학교를 마치고 책상에 앉아도 30분 이상 집중하기 어려웠고, 쉬고 싶은 마음이 들면서 계획에 차질이 생겼습니다. 프로젝트 중간에는 "차라리 더 쉬운 프로젝트를 선택할 걸"이라는 후회도 들었습니다.

이 경험을 통해, 이후 프로젝트를 계획할 때는 저의 게으른 습관까지 고려하여 최악의 상황을 가정한 현실적인 계획을 세워야겠다고 느꼈습니다.

- 문서 작업의 어려움

보고서를 작성하는 일이 쉽지 않았습니다. 문서 작성 경험이 부족한 탓에 예시를 보고도 시작하기까지 몇 시간을 허비했습니다. 특히 2주마다 제출해야 하는 진척 보고서는 큰 부담으로 다가왔습니다. 지난 작업 내용을 정확히 기억하지 못해 코드를 다시 분석하며 보고서를 작성해야 했습니다. 이번 경험을 통해, 매일 3분이라도 업무 내용을 정리해 보고서 작성 시간과 노력을 크게 절감해야겠다고 생각했습니다.