

C++ 프로그래밍 및 실습 [4]

2D 게임 엔진 및 강화 학습 에이전트 개발

진척 보고서 #01

제출일자: 2024-11-17

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 보여 주어, 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 게임 엔진과 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) 간단한 2D 게임 엔진 기본 기능 구현

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

(1) 충돌 감지 시스템

- 에이전트가 장애물이나 벽에 부딪힐 경우, 해당 이벤트를 감지하여 행동을 제안하거나 방향을 조정하도록 설정

- 충돌 이벤트 탐지
- 장애물/벽 객체 및 충돌 체크 기능
- 충돌 시 에이전트 행동 업데이트

(2) 중력 및 이동 시스템

- 캐릭터의 중력 효과와 좌우 이동, 점프 기능을 추가하여 현실적인 움직임을 구현

- 중력 적용을 위한 업데이트 함수 추가
- 키 입력에 따른 좌우 이동 처리
- 점프 기능 및 중력과의 상호작용 구현

(2) 환경 시각화

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

- 기본 배경 및 장애물 시각화 방안 구현
- 에이전트 캐릭터 표시 및 이동 구현

2) 강화학습 알고리즘 구현 및 통합

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

(1) 상태 공간 정의

- 에이전트의 상태를 위치, 속도, 장애물과의 거리 등으로 정의하여 학습을 위한 상태 공간 정의

- 상태 공간의 변수와 범위 정의
- 상태별 식별 코드 생성

(2) 행동 및 보상 시스템 설정

- 에이전트가 할 수 있는 행동과, 특정 행동에 대한 보상을 정의

- 좌우 이동, 점프와 같은 행동 정의
- 목표 도달 시 보상 부여, 충돌 시 패널티 정의

(3) Q-learning 알고리즘 구현

- Q-learning 알고리즘을 통해 에이전트가 목표에 도달하는 행동을 학습하게 함.

- Q 테이블 초기화 및 갱신 함수 구현

- 학습률, 할인률 등의 하이퍼 파라미터 설정
- 학습 루프 구현

3) 학습 데이터 분석 및 시각화

- 학습이 진행됨에 따라 에이전트의 성능을 시각화하여 학습 속도와 정확도를 분석

(1) 보상 기록 및 분석

- 각 반복 별로 획득한 보상을 기록하여 학습의 진행상황을 분석
 - 반복별 보상 저장 기능 구현
 - 보상 데이터를 그래프로 출력

(2) 성공률 시각화

- 에이전트가 목표에 도달한 성공률을 시각화하여 성능을 평가
 - 성공률 계산 함수
 - 시각적 그래프로 결과 출력

3. 진척사항

1) 기능 구현

(1) 입출력 시스템

- 입력: 키보드의 알파벳, 상하좌우 방향키
- 설명
 - Input 객체를 생성해, 누를 수 있는 Key를 배열로 선언
 - vector로 눌린 키의 배열을 만들고, Key클래스를 배열에 할당해 각 키의 상태를 클래스에 직접적으로 저장
 - Update 함수로 vector<Key>배열을 순회하며 키보드 상태를 업데이트 및 게임 오브젝트의 동작 제어.
- 적용된 배운 내용
 - for 구문, if 구문: 입력 상태를 확인하고 상태를 업데이트
 - vector, 배열: 각 상태, 객체를 vector에 저장, 이어지는 정보를 배열에 저장
 - class와 메서드: Input 클래스를 선언하고 메서드로 기능 구현
 - 포인터: 입력 상태를 동적으로 저장하고 관리

- 스크린샷

```
void Input::Update()
{
    for (size_t i = 0; i < Keys.size(); i++)
    {
        // 키가 눌리면
        if (GetAsyncKeyState(ASCII[i]) & 0x8000)
        {
            // 눌린 키 찾기
            if (Keys[i].bPressed == true) // 과거에도 눌러있었을 때
                Keys[i].state = eKeyState::Pressed;
            else // 새로 누르는 상태
                Keys[i].state = eKeyState::Down;
            Keys[i].bPressed = true;
        }
        else // 키가 안눌림
        {
            if (Keys[i].bPressed == true) // 이전에 누르다가 놓은 상태
                Keys[i].state = eKeyState::Up;
            else // 아예 눌리지 않은 상태
                Keys[i].state = eKeyState::None;
            Keys[i].bPressed = false;
        }
    }
}

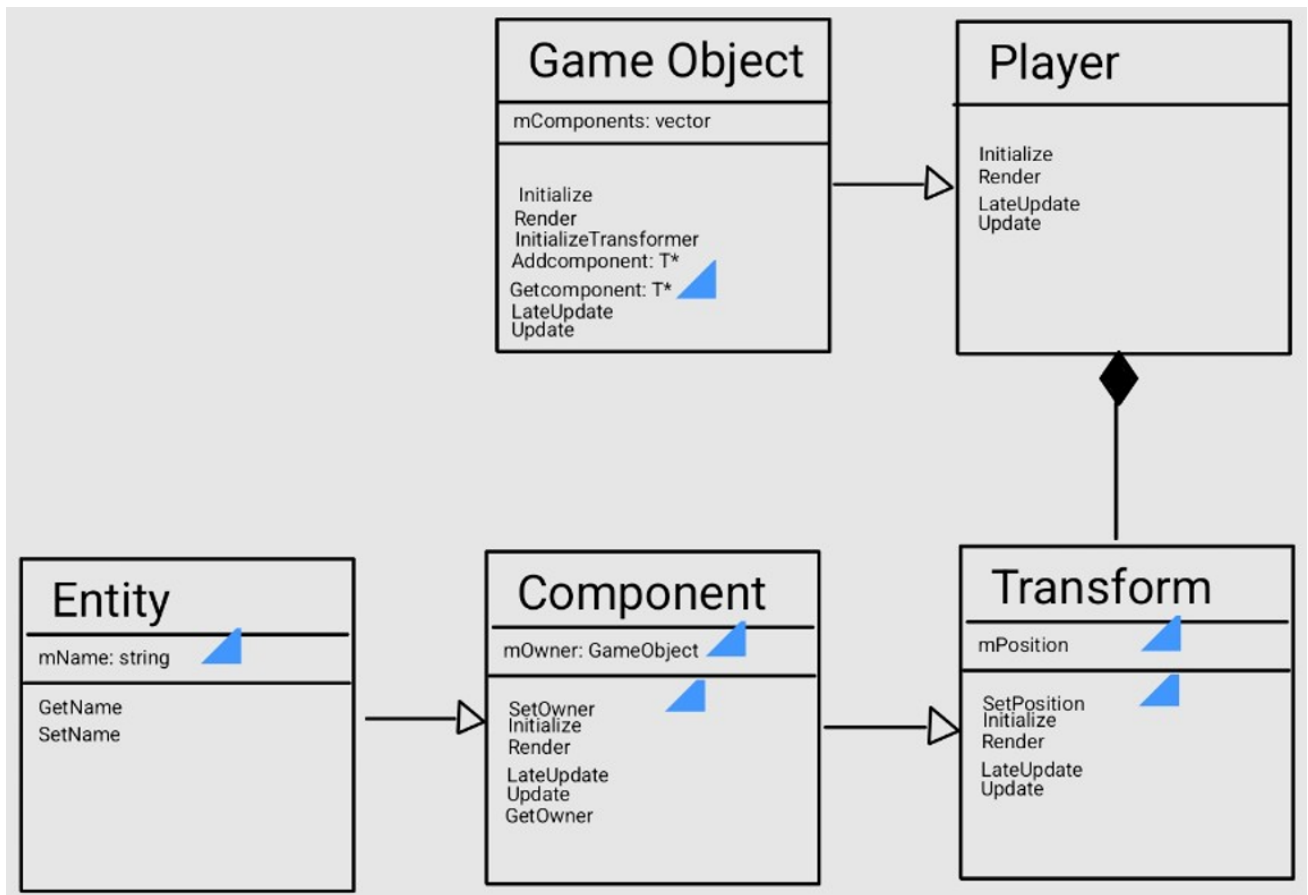
void Input::updateKey(Input::Key& key)
{
    // 키가 눌린 상태라면
    if (isKeyDown(key.keyCode))
    {
        updateKeyDown(key);
    }
    else
    {
        updateKeyUp(key);
    }
}
```

(2) 이동 시스템

- 입력: 상하좌우 방향키

- 설명

- Player 클래스는 사용자의 입력에 따라 캐릭터의 위치를 업데이트하는 이동 시스템을 구현
- 키보드 입력(Input::GetKey)을 통해 상하좌우 이동을 구현
- 이동 속도는 시간에 따라 움직이도록 설정
- Player 클래스는 위치 정보를 저장하는 Transform 객체보유



- Player 클래스의 Update메서드에서 Transform 객체를 수정하며 이동

- 적용된 배운 내용

- 클래스와 상속: Player 클래스는 GameObject를 상속하고 Transform을 보유한다. 이들은 상속관계로 묶여 있어 메서드를 override 하며 다형성의 기초를 쌓는다.

- 조건문: 특정 키 입력에 따라 이동 처리

- 캡슐화: private으로 멤버를 선언 후 Get, Set 메서드로 멤버에 접근

- 포인터: 포인터로 객체에 직접 접근하여 값을 조정

- 사진

```
// 가변형으로 다형성 구현
template <typename T>
T* GetComponent()
{
    T* component = nullptr;
    for (Component* comp : mComponents)
    {
        // dynamic_cast는 캐스팅 가능시 캐스트 된 결과를, 불가능시 nullptr 반환
        component = dynamic_cast<T*>(comp);
        if (component)
            break;
    }
    return component;
}
```

```
void SetPosition(Vector2 pos) { mPosition.x = pos.x; mPosition.y = pos.y; }
Vector2 GetPosition() { return mPosition; }
```

```

void Player::Update()
{
    GameObject::Update();
    if (Input::GetKey(eKeyCode::Right))
    {
        Transform* tr = GetComponent<Transform>();
        Vector2 pos = tr->GetPosition();
        pos.x += 100.0f * Time::DeltaTime();
        tr->SetPosition(pos);
    }

    if (Input::GetKey(eKeyCode::Left))
    {
        Transform* tr = GetComponent<Transform>();
        Vector2 pos = tr->GetPosition();
        pos.x -= 100.0f * Time::DeltaTime();
        tr->SetPosition(pos);
    }

    if (Input::GetKey(eKeyCode::Up))
    {
        Transform* tr = GetComponent<Transform>();
        Vector2 pos = tr->GetPosition();
        pos.y -= 100.0f * Time::DeltaTime();
        tr->SetPosition(pos);
    }

    if (Input::GetKey(eKeyCode::Down))
    {
        Transform* tr = GetComponent<Transform>();
        Vector2 pos = tr->GetPosition();
        pos.y += 100.0f * Time::DeltaTime();
        tr->SetPosition(pos);
    }
}

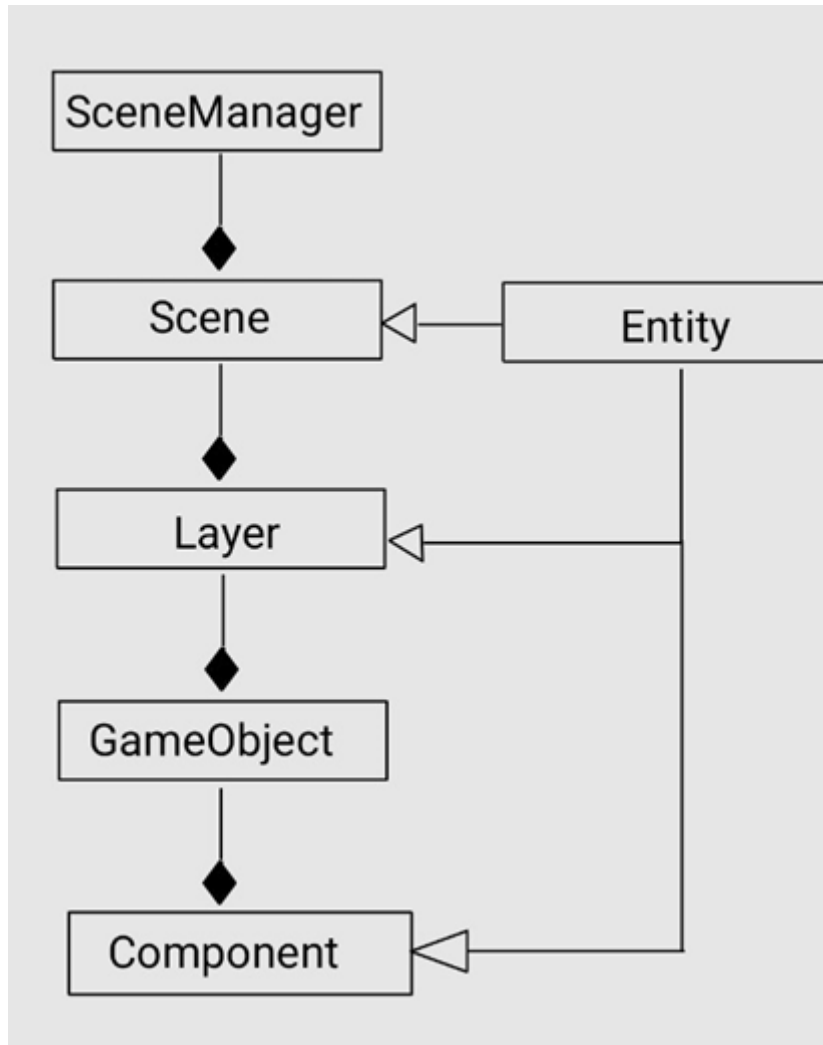
```

(3) 환경 시각화

- 설명

- 게임 환경 시각화를 위해 크게 시간, 씬(Scene), 레이어, 렌더링으로 나누어 계층적 구조를 구성함
- 시간(Time)
 - 프레임 단위의 시간 간격(DeltaTime)을 계산해 프레임과 독립적인 렌더링 구현
- 씬 관리
 - 씬 클래스를 선언하고 씬을 여러 개 보유하여 관리하는 SceneManager 클래스 생성

- SceneManager 클래스는 `map<SceneName, Scene*> mScene`을 보유
- 레이어(Layer) 관리
 - 씬을 구성하는 레이어를 관리하며, 레이어별로 오브젝트를 업데이트 및 렌더링
- 렌더링
 - SceneManager->Render() => Vector<Scene>[i]->Render() => Vector<Layer>[i]->Render() => Vector<GameObject>->Render() 순으로 호출



- 적용된 배운 내용
 - 상속: 계층 구조를 만들어 다형성을 활용
 - 포인터: 포인터를 사용해 객체에 직접 접근
- 스크린샷
 - 1. 프레임에 따른 균일한 이동

```

void Time::Render(HDC hdc)
{
    static float time = 0.0f;

    time += DeltaTimeValue;

    // 초당 프레임 계산
    float fps = 1.0f / DeltaTimeValue;

    wchar_t str[50] = L"";
    swprintf_s(str, 50, L"Time : %d", (int)fps);
    int len = wcslen_s(str, 50);

    TextOut(hdc, 0, 0, str, len);
}

```

- 2. Scene 내의 레이어 생성 및 렌더링

```

void Scene::Render(HDC hdc)
{
    for (Layer* layer : mLayers)
    {
        if (layer == nullptr)
            continue;
        layer->Render(hdc);
    }
}

```

- 3. Layer 내의 GameObject 렌더링

```

void Layer::Render(HDC hdc)
{
    for (GameObject* gameObj : mGameObjects)
    {
        if (gameObj == nullptr)
            continue;
        gameObj->Render(hdc);
    }
}

```

- 4. GameObject의 Component 렌더링(Component는 여러 개 보유할 수 있음)

```

void GameObject::Render(HDC hdc)
{
    for (Component* comp : mComponents)
    {
        comp->Render(hdc);
    }
}

```

- 5. SceneManager의 Scene Load 과정

```
Scene* SceneManager::LoadScene(const std::wstring& name)
{
    // mActiveScene이 존재할 때
    if (mActiveScene)
        mActiveScene->OnExit();

    // Scene을 검색
    std::map<std::wstring, Scene*>::iterator iter = mScene.find(name);

    // 찾는 Scene이 없을 때 nullptr 반환
    if (iter == mScene.end())
        return nullptr;

    // 검색된 씬을 현재 활성화된 씬으로 설정
    mActiveScene = iter->second;
    mActiveScene->OnEnter();

    // 활성화된 씬 객체를 반환
    return iter->second;
}
```

(4) 실행 방법

1. CPP2409-P/MakeEngine.exe 응용프로그램 실행

1-1. 1번이 불가한 경우 CPP2409-P/MakeEngine/x64/Debug/MakeEngine.exe 응용프로그램을 실행시키시면 됩니다.

2. 키보드 "N"을 눌러 Scene 전환

3. 키보드 상하좌우를 눌러 사진(캐릭터)이동

2) 테스트 결과

(1) 입출력 시스템

- 설명

- 키보드 "N"을 클릭 시 Scene이 변경되도록 구현

- 스크린샷

(전)



(후)

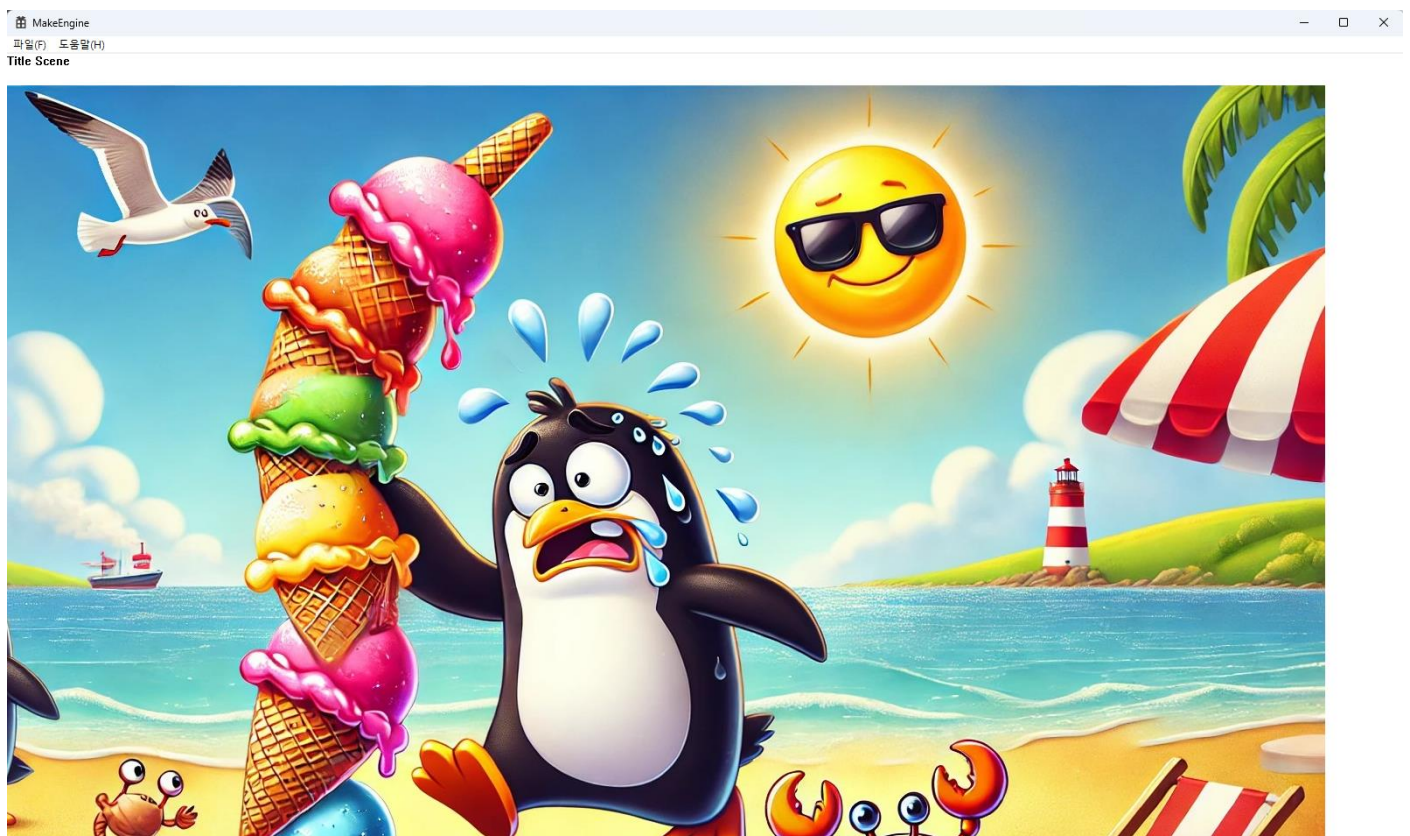


(2) 이동 시스템

- 설명

- 방향키를 움직일 시 사진이 상하좌우로 움직이게 설계

- 스크린샷





(3) 환경 시각화

- 설명

- 각 Scene별로 별도의 상태(Transform, 위치)를 가지므로 "N"을 눌러 Scene을 전환해도 그 전의 움직임던 상태들은 그대로 유지



4. 계획 대비 변경 사항

1) 2D 게임 엔진 개발 일정 지체

- 이전

게임 엔진 구현: 11.03~11.17 (2주)

- 이후

게임 엔진 구현: 11.03~12.1 (4주)

- 사유

1. 게임 엔진 구현에서 다형성에 기초한 구조 설계로 시간이 지체
2. WinApi를 사용하는데 있어 미숙함이 존재
3. 게임 엔진 구현 표준을 학습하는 과정에서 다형성에 기초한 구조 설계로 인한 어려움이 존재

5. 프로젝트 일정

업무	11/3	11/10	11/17	11/24	12/1	12/8	12/15	12/22
제안서 작성	---->							
기능 1: 게임 엔진 구현								
충돌 감지 시스템				---->				
중력 및 이동 시스템		1/2		---->				
환경 시각화			0	---->				
기능 2: 강화 학습 알고리즘								
상태 공간 정의					---->			
행동 및 보상 시스템 설정					---->			
Q-learning 구현					----->	---->	---->	
기능 3: 학습								

데이터 분석 및 시각화								
성공률 시각화							---→	
보상 기록 분 석 및 분석								---→
최종 점검 및 보고서 작성								---→