한국산학기술학회논문지
Vol. 10, No. 11, pp. 3155-3162, 2009

게임 논리 이식성 향상을 위한 프레임워크 구현

김석현[1], 이면재[2*]
[1]남서울대학교 멀티미디어학과
[2]백석대학교 정보통신학부

Framework Implementation for Improvement of Game Logic Portability

Seok-Hyun Kim[1] and Myoun-Jae Lee[2*]
[1]**Department of Multimedia, Namseoul University**
[2]**Department of Information & Communication, Baekseok University**

요 약 게임 산업이 성장하면서 더욱 새롭고 다채로운 내용을 담고 있는 게임에 대한 요구가 커지고 있다. 그러나 현재 이를 지원하기 위한 소프트웨어 프레임워크는 존재하지 않는다. 3D 엔진이나 물리엔진과 같이 게임을 위한 특정 기술 분야의 미들웨어는 상당히 성숙한 단계에 와있으나 게임논리의 흐름을 위한 프레임워크는 거의 찾아볼 수 없다. 본 논문에서는 게임논리의 설계 및 구현을 위한 소프트웨어 프레임워크로서 게임논리엔진을 제안하고, 프로토타입을 구현한다. 제안된 게임논리엔진에 의해 다양한 장르의 게임을 위한 논리 흐름을 쉽게 기술하고 조합할 수 있다.

Abstract As game industry is growing, the desire for game that has more fresh and various contents is bigger. However software framework for this is not exist. Middlewares for certain game technologies, like 3D engine, physics engine, are in matured step, but it is very hard to see the framework for the flow of game-logic. This paper proposes Game-Logic Engine, the software framework for design and implementation of game-logic, implements a prototype of game logic engine. Game-logic flows for various game genre are easily described and combined by the Game-Logic Engine.

Key Words : Software framework, Game logic, Game story-telling

1. 서론

컴퓨터 게임 산업은 다양한 문화 산업들 중에서 12.86%의 비중을 차지하고 있을 정도로 중요성이 커지고 있다[1]. 이러한 게임 산업의 발전에도 불구하고 게임 제작의 특성상 크게 두 가지의 어려움이 존재하고 있다.

첫번째는 게임 디자이너와 프로그래머간의 효율적인 협업이다. 게임디자이너는 창의력을 바탕으로 다양한 실험을 통해 자신의 아이디어가 우수한지 판단하기를 원하여서 게임을 개발하는 도중에 기획을 변경하기를 원한다. 그러나, 게임 제작 중에 게임 디자이너가 기획 의도를 변경하려는 경우 프로그래머들은 변경된 논리의 구현으로 인한 시간적인 부담과 게임 완성도에 부담을 갖을 수 있어 기획의 변경을 원하지 않는다. 이러한 프로그래머의 부담은 클래스간 상호 의존도가 높은 코드[2]를 작성한 경우에 더욱 심해질 수 있다. 즉, 게임 제작과 같이 코드 작성량이 많은 프로젝트를 개발하려는 경우에 게임 기획의 변경이 요구되면 변경이 요구되는 논리에 영향을 받는 클래스들과 논리들이 동시에 변경되어야 한다. 이는 예정된 시간에 게임을 출시하는 것을 어렵게 할 수 있다.

두번째는 개발된 프로그램 또는 논리의 재사용 문제이다. 이미 개발된 게임 프로그램 또는 논리를 다른 게임에 쉽게 이식하는 경우 게임 개발 시간의 단축을 꾀할 수 있다. 게임 프로그램을 다른 게임에 쉽게 이식하기 위한 연구에는 [4-5]가 있다. 그러나, 이 방법들은 게임 논리 자체의 이식보다 특정 게임 엔진에서 개발된 게임을 다른

*교신저자 : 이면재(davidlee@bu.ac.kr)
접수일 09년 04월 23일 수정일 (1차 09년 05월 27일, 2차 09년 10월 06일, 3차 09년 11월 04일) 게재확정일 09년 11월 12일

게임 엔진에 이식하기 위한 것에 중점을 두었다. 두 방법 모두에서 게임 상태와 게임 행동 등에 관한 처리를 수행하는 게임 모델[4] 부분 또는 이벤트 공간[4]부분과 특정 게임 엔진의 연동을 위하여 어뎁터가 요구되는데, 이 제작으로 인한 개발 시간과 비용이 커질 수 있다.

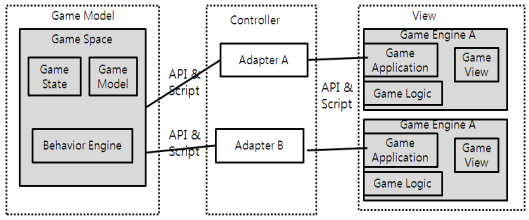
본 논문에서는 기존 클래스를 이용한 함수 호출 방법에서 발생될 수 있는 클래스간 강한 의존도를 줄여서 클래스의 독립성을 높이는 동시에 게임 논리의 이식을 향상시키기 위하여 게임 논리와 게임 데이터들을 분리하여 효율적으로 게임을 제작할 수 있는 메시지(message) 기반의 게임 논리 엔진(game-logic engine)을 제안한다. 제안된 게임 논리엔진은 게임 데이터를 저장하는 엔티티, 논리를 기술하는 이벤트, 엔티티와 이벤트간의 통신 수단인 메시지로 구성된다. 이러한 방법으로 게임 데이터와 논리를 분리하는 동시에 메시지를 사용하여서 클래스간 강한 의존도를 감소시킨다.

본 논문의 구성은 다음과 같다. 먼저 2장에서 관련 연구를 정리한다. 3장에서 게임논리엔진을 제안하고, 4장에서 제안된 게임논리엔진 프로토타입 구현에 대하여 기술하고 이를 실험한다. 그리고, 5장에서 결론 및 향후 연구 방향에 대하여 논한다.

2. 관련연구

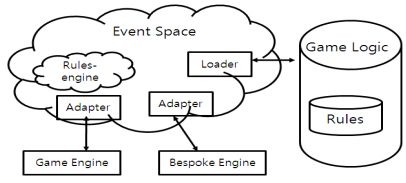
게임논리를 비롯한 여러 게임내부 상태를 G-factor라 명명한 [3]에서는 40개의 게임을 예로 사용하여G-factor의 이식성(portability)에 대한 연구를 수행하였다. 이식성을 비교하기 위하여 게임 내부 변수들을 읽어들이는 위치에 따라 data-driven/hard-corded로 나누고, object model에 따라 dynamic/static으로 나누고, 사용하는 스크립트의 컴파일 시점에 따라 pre-compiled/on-the-fly로 나누었다. 이러한 구분을 통해 40여가지 게임을 총 6가지 단계의 이식성으로 나누어 평가하였다. 이 논문은 게임 논리의 이식성 평가 요구사항을 중점으로 기술하고 평가한 것이다.

게임 엔진들간의 프로그램 이식성을 높이기 위한 연구로는 [4]와 [5]가 있다. 그림 1은 [4]에서 제시된 구조를 보여준다. 이 구조는 게임 모델, 컨트롤러, 게임 엔진으로 구성된다. 게임 모델은 게임 상태, 게임 모델, 게임 행동 엔진으로 구성된다. 게임 엔진에서는 게임 응용프로그램, 게임 논리등으로 구성된다. 컨트롤러는 게임 모델과 게임 엔진간의 연결을 수행하는 부분으로 각 게임 엔진당 어뎁터가 있어서 해당 게임 엔진에 적합하게 변환하는 역할을 수행한다. 이 어뎁터를 사용하여 이미 제작된 게임 프로그램을 쉽게 다른 게임 엔진에 이식할 수 있다.



[그림 1] [4]에서 제시된 구조

그림 2는 [5]에서 제시된 구조를 보여준다. 이 구조는 이벤트 공간(Event Space)과 게임 논리 부분으로 구성된다. 이벤트 공간은 게임 행동을 컨트롤하는 룰 엔진(rule engine), 게임 엔진과 룰 엔진과의 게임 상태 정보를 동기화하는 어뎁터, Character 또는 NPC와 같은 오브젝트의 행동과 사건을 관리하거나 오브젝트 애트리뷰트를 표현하기 위한 템플릿들을 가진 룰-엔진, 이를 초기화하는 로더로 구성된다. 이 방법에서도 [4]에서 제시된 방법에서와 같이 어뎁터를 사용하여 이미 제작된 게임 프로그램을 다른 게임 엔진에 쉽게 이식할 수 있다. [4,5] 연구에서는 게임 논리보다는 게임 프로그램 전체를 다른 게임 엔진에 이식하는 것에 대한 연구들이다. 이 방법들은 개발자에게 호환성이 우수한 어뎁터 제작에 대한 부담감을 준다.

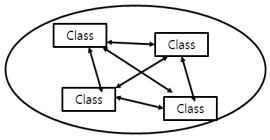


[그림 2] [5]에서 제시된 구조

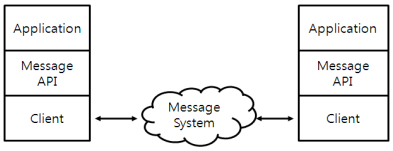
그림 3은 기존 게임 제작 방법에서 많이 사용되는 클래스간 강하게 결속된(tightly coupled) 형태의 정보 교환 방법을 보여준다[2]. 특정 클래스의 구조가 변경되는 경우 이와 연관된 클래스의 소스를 모두 변경시켜야 한다. 특히 게임 제작 초기에 기획이 자주 변경되어 클래스의 구조가 자주 변경되는 경우 예정된 출시시간에 완성된 게임의 제작이 어려울 수 있으며, 소스 코드의 완성도 또한 부족해 질수 있다.

그림 4는 상대 클래스에 대한 호출 방법과 내부 구현 사항에 대한 방법을 인식할 필요가 없는 메시지 기반 통신 방법을 보여준다[2]. 특정 클래스의 구조가 변경되더라도 메시지 인터페이스로 구성되는 메시지 시스템을 통하여 정보를 교환하므로 상대 클래스에서는 내용을 변경할 필요가 없다. 이러한 메시지 기반 통신 방법은 클래스

의 구조 변경으로 인한 추가적인 소스 개발에 대한 부담을 감소시킨다.



[그림 3] 클래스간 결속 구조



[그림 4] 메시지 기반 통신

3. 제안된 게임 논리 엔진

본 논문에서는 게임 논리의 이식을 수월하게 하기 위하여 메시지 통신 방법을 게임 논리 엔진에 적용하여 클래스간 의존도가 높은 기존 게임 제작방법에서의 문제점을 개선한다.

3.1 게임 논리엔진의 요구사항

게임논리엔진은 게임논리의 흐름을 기술하고 그대로 작동되게 만든다. 이를 위한 요구사항을 게임 디자이너의 측면, 게임 프로그래머의 측면에서 살펴본다.

게임 디자이너의 목표는 자신이 고안한 핵심적인 게임 요소들을 게임 프로그램을 통하여 충실하게 구현하는 것이다. 이를 위해 먼저 자신이 생각한 모든 것을 제대로 표현할 수 있는 수단이 필요하다. 그러므로, 제안 게임 논리 엔진은 풍부한 표현력을 가지면서 쉽고 직관적으로 게임의 내부 논리 흐름을 기술할 수 있는 도구가 되어야 한다.

게임프로그래머는 게임디자이너가 만든 게임기획 의도를 프로그램에 충실히 반영하여 완성할 수 있어야 한다. 이를 위해 게임디자이너가 논리적인 표현 방식을 따른다면 프로그래머는 훨씬 쉽게 실제 게임 코드에 적용할 수 있다. 또한 게임 프로그래머는 3D 렌더링 엔진, 물리엔진 등의 다양한 모듈들을 다루어야 하므로 게임 논리엔진은 다양한 모듈들의 상호 작용을 쉽게 담을 수 있어야 한다.

3.2 게임논리모델

게임이 만드는 가상 세계의 흐름은 여러 인과로 표현할 수 있다. 다양한 인과들을 만들어 놓고 현재 게임 내의 상태에 따라 해당 규칙들이 적용되게 하면 다양하고 자유롭게 게임 논리의 흐름을 기술할 수 있다.

이를 위해, 게임 내 객체의 특성을 기술하는 엔티티들을 정의한다. 각각의 엔티티는 게임 객체의 속성들로 구성된다. 엔티티는 자신의 속성 값에 따라 여러 상태를 가지는 것으로 볼 수 있다. 엔티티 E_i의 상태 갯수를 $|E_i|$로 나타내기로 하자.

게임 내에서 실제 객체들은 엔티티의 인스턴스이다. 인스턴스들은 엔티티에서 정의된 규칙을 따라 실행된다. 특정 엔티티의 인스턴스들은 모두 같은 방식으로 작동하므로 논의의 편의를 위해 모든 엔티티의 인스턴스가 1개만 존재하는 경우만 생각한다. 이 경우 게임논리상태 $\overline{S}$는 식(1)과 같이 표현될 수 있다. 여기에서 $S(E_i)$는 E_i의 상태이다.

$$\overline{S} = [S(E_1)\ S(E_2)\ \ldots\ S(E_n)] \quad (1)$$

게임논리상태를 $\overline{S_i}$에서 $\overline{S_j}$로 변화 시키는 규칙 R_{ij}는 식(2)와 같이 기술된다.

$$R_{ij} : \overline{S_i} \rightarrow \overline{S_j} \quad (2)$$

게임논리프레임워크는 시점 t의 게임논리상태, 즉 $\overline{S}(t)$와 정의된 모든 규칙을 비교하여 규칙이 성립하면 이를 정의하여 게임논리상태를 변화시키는 작업을 매 시점 t에 시행한다.

3.3 게임논리엔진의 구성요소

3.2의 게임논리모델을 기초로 게임논리엔진의 구성 요소를 설계한다. 게임논리엔진은 세 가지 객체타입으로 게임 내부의 논리적 흐름을 표현한다. 첫 번째 객체타입은 게임논리모델의 엔티티를 구현하는 엔티티이다. 엔티티는 게임 내부에 존재하는 모든 객체의 속성을 저장한다. 엔티티가 객체의 정보를 저장하는 방식은 데이터베이스에서 스키마 설계에 따라 다양한 정보가 저장되는 것과 유사하다. 엔티티의 형태는 테이블 방식으로 설계되고 이 테이블에 엔티티의 인스턴스 정보들이 저장된다. 게임논리엔진은 엔티티를 통해 게임 내부의 다양한 정보를 얻을 수 있다. 즉, 엔티티는 게임 상태를 저장하는 데이터베이스와 같은 역할을 수행한다.

두 번째 객체타입은 게임논리모델의 규칙을 구현하는 이벤트(event)이다. 구현에서 이름을 이벤트로 한 이유는,

수학적 모델에서는 간략하게 규칙이라는 표현이 적합하지만 실제 구현을 하다 보면 비교적 긴 시간동안 다양한 엔티티의 상호작용을 다루기 때문에 이벤트라는 이름을 사용한다.

이벤트는 게임 내부의 다양한 동적인 상호작용을 표현한다. 예를 들어 엔티티 A의 한 인스턴스에서 공격이라는 이벤트를 생성한다고 하자. 공격 이벤트는 A가 다른 엔티티를 공격하기 위한 모든 과정을 처리한다. 공격 목표의 유효성 검사, 공격과정의 특수효과 발생, 공격 이후의 과정 등 A가 다른 객체와 공격이라는 방식으로 상호작용하는데 필요한 모든 과정을 순차적으로 처리한다. 공격과 관련된 모든 과정이 끝나면 공격 이벤트는 자동적으로 사라진다. 이러한 일련의 과정은 게임 내부 상태를 바꾸게 된다. 그러므로 이벤트는 게임논리모델의 규칙과 대응된다.

마지막 객체타입은 메시지로서 함수호출과 같은 역할을 수행한다. 기존의 게임 제작방법에서는 게임논리모듈이 작동할 때 내부적으로 클래스 인스턴스들 간에 다양한 상호작용이 발생하며 이러한 상호작용에는 주로 함수호출을 사용한다. 이 방식은 게임의 설계가 바뀌면 코드를 수정하고 컴파일을 다시 해야만 변경된 설계를 확인할 수 있다. 그러나, 제안된 게임논리엔진은 인스턴스 사이의 상호작용을 메시지 전달 및 처리로 표현한다. 이렇게 하면 게임코드 자체를 재컴파일하지 않고도 새로운 메시지를 추가함으로써 새로운 인스턴스 사이의 상호작용을 표현할 수 있다.

이와같이 게임논리엔진에서는 엔티티, 이벤트, 메시지 세 가지 객체타입을 통해 다양한 게임의 논리적인 흐름을 구현한다. 게임논리엔진은 이러한 객체들의 인스턴스 정의 및 상호작용 방식을 프로그램 코드 외적인 요소로 변경할 수 있고 실시간에 바꿀 수 있으므로 게임의 변경 및 유지보수에 커다란 유연성을 부여할 수 있다.

3.3 게임논리엔진 설계

3.3.1 엔티티

게임논리엔진은 게임 내부 상태에 대한 데이터베이스적인 성격을 가지고 있다. 아바타의 위치, 적 몹(mob)의 위치 및 성격 등 게임 내에 존재하는 다양한 인스턴스들에 대한 정보를 저장하고 필요할 때 이 정보를 볼 수 있도록 한다. 하지만 본격적으로 데이터베이스 기술을 도입하려는 것은 아니다. 관계형 데이터베이스는 스키마를 정의하고 데이터를 넣고 SQL을 통해 다양한 정보를 조합하고 검색할 수 있지만 게임논리엔진의 엔티티들은 단순한 정보의 테이블로서 게임 내부 객체의 속성을 정의하고 저장할 수 있는 구조이다. 게임디자이너와 게임프로그래머 모두 다양한 게임 내부 정보를 저장하는 용도로 엔티티 객체를 사용할 수 있다. 이 엔티티는 애트리뷰트와 실제 값으로 구성되어 있다.

게임디자이너가 설계한 엔티티 구조는 게임 소프트웨어가 구동될 때 외부 파일에서부터 로딩되어 생성된다. 엔티티 구조의 장점은 실시간에 얼마든지 테이블을 추가하고 테이블에 데이터를 넣을 수 있다는 것이다. 예를 들어 온라인 게임의 경우 네트워크 패킷을 통해 새로운 몬스터 속성을 정의하고 추가하는 일이 가능하다.

엔티티에서는 게임 진행에 필요한 다양한 내부 상태 정보를 저장하기도 한다. 게임프로그래머는 엔티티를 활용하여 다양한 정보를 저장하는 구조를 활용할 수도 있다.

3.3.2 이벤트

이벤트는 엔티티를 상속 받는다. 이벤트도 엔티티처럼 다양한 속성을 가질 수 있기 때문이다. 이벤트와 엔티티의 가장 큰 차이는 이벤트는 게임의 동적인 상호작용을 관장하는 객체라는 점이다. 모든 이벤트는 기본적으로 게임논리엔진이 지속적으로 호출하는 처리함수를 가지고 있다. 활성화된 이벤트는 주기적으로 호출되는 처리함수 내부에서 자신의 논리적 흐름을 만든다.

모든 이벤트는 시작조건과 종료조건을 가진다. 시작조건이 만족되는 경우 이벤트는 활성화되고 종료조건이 만족되기 전까지 작동한다. 시작조건과 종료조건은 이벤트 간의 의존관계를 처리한다는 점에서 매우 중요하다. 예를 들어 A, B, C 세 이벤트에 대해 A->B 의 의존 관계가 형성되는 경우, A 이벤트가 활성화 된 상황에서만 B가 발생할 수 있다. 만약 ((A AND C)->B)로 시작 조건을 설정한다면 A와 C가 활성화된 상황에서만 B가 활성화 될 수 있다.

이벤트는 다양한 엔티티 사이의 상호작용을 표현하여 게임논리의 흐름을 기술하며 다양한 이벤트를 사용하여 복잡한 게임논리의 흐름이 기술된다.

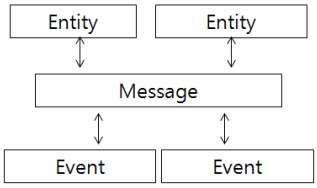
3.3.3 메시지

엔티티와 이벤트는 서로 통신할 수 있는 수단이 필요하다. 메시지는 이를 위한 객체이다. 엔티티와 이벤트는 서로에게 메시지를 보내는 방식으로 정보를 전달한다. 이는 함수 호출을 대체하는 것과 같다. 함수를 호출할 때 함수이름과 호출인자가 필요했던 것처럼 메시지를 보낼 때는 메시지 수신 대상, 메시지 종류, 메시지 인자가 요구된다. 함수 호출과 메세지의 차이점은 메시지는 수신자를 다수의 객체로 할 수 있다는 것이다. 명시적으로 n개의

메시지 수신 객체를 설정할 수도 있고, 특정 엔티티에 들어있는 모든 객체들은 특정 메시지를 받을 수 있다.

3.3.4 전체 구조

그림 5는 엔티티, 이벤트, 메시지의 구조를 보여준다. 이러한 구조를 통해 게임논리엔진은 게임논리모듈의 프레임워크로써 다양한 장르의 게임논리를 기술하면서, 게임 개발 공정에서 게임디자이너와 게임프로그래머의 협업을 단순하고 효율적으로 만들어 준다.



[그림 5] 게임 논리 엔진 메시지 전달 과정

4. 게임논리모델 분석및 프로토타입 구현

4.1 게임논리모델 복잡도 분석

게임논리모델을 토대로 게임논리를 구현할 때 복잡도는 게임논리상태 및 게임논리규칙의 개수와 관련되어 있다. 게임논리상태는 각 엔티티 상태들의 벡터로써 표현된다. 따라서 가능한 게임논리상태의 총 수는 각 엔티티 상태 수의 곱이다.

이 때 실제 구현되는 것은 개별 엔티티이기 때문에 게임논리상태 구현 비용은 개별 엔티티의 상태수의 합에 비례한다. 규칙의 구현 비용은 규칙의 전체 수에 비례한다. 따라서 전체 구현 비용은 개별 엔티티의 상태 수와 규칙의 수의 합으로 표현할 수 있다. 식(3)은 이 비용을 나타낸다.

$$COST_{GameLogicEngine} = O(|R| + \sum_{i=1}^{N} |E_i|) \qquad (3)$$

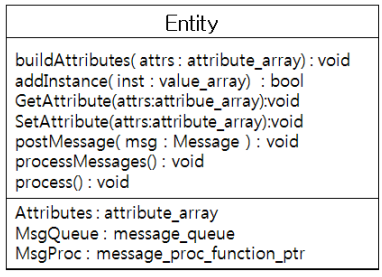
게임 논리엔진에서와 동일한 논리를 일반적인 클래스 설계 방식으로 구현하는 경우 게임논리엔진에서의 규칙 집합 R은 규칙과 관련된 클래스들의 멤버함수로써 구현되어야 한다. 따라서 최악의 경우 r∈R인 r과 관련된 각각의 클래스들에 멤버함수가 구현되어야 한다. 일반 클래스 구현 방식의 비용은 식(4)와 같으며, 이 비용은 각 클래스의 구현 비용의 합과 게임 논리엔진에서의 규칙에 대응하는 논리를 구현함으로써 영향을 받는 클래스 갯수의 합으로 표현된다. 여기에서 c는 각 규칙과 관련된 평균 클래스의 개수이며 C_i는 i번째 클래스이며, c|R|은 규칙 R에 대응되는 논리를 구현함으로써 영향을 받는 클래스의 수이다.

$$COST_{GeneralClass} = O(c|R| + \sum_{i=1}^{N} |C_i|) \qquad (4)$$

제안 논리 엔진의 프로토타입을 구현하여 제안 논리 엔진의 구현 비용과 클래스 구현 방식의 구현 비용을 비교한다.

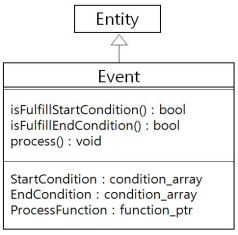
4.2 프로토타입 구현

게임논리엔진의 검증을 위해 게임논리엔진 프로토타입을 구현하였다. 그림 6은 엔티티 구조를 보여준다. 게임디자이너는 게임을 설계할 때 다양한 엔티티들을 정의하면서 게임에 등장하는 요소들의 특성을 설계한다. 엔터티 요소에는 애트리뷰트를 지정할 수 있는 buildAttribute(), 인스턴스를 추가할 수 있는 addInstance(), 엔터티에 관한 메시지를 처리하는 ProcessMessage() 등으로 구성된다.

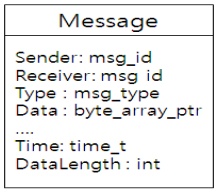


[그림 6] 엔티티 구조

그림 7은 이벤트 구조를 보여준다. 이벤트 클래스는 엔티티 클래스를 상속받는다. 그리고, 이벤트 클래스에서는 이벤트의 시작 조건을 나타내는 isFullFillStartCondition()과 종료 조건을 나타내는 isFullFillEndCondition(), 그리고 이벤트 처리 함수를 등록하는 SetRunFunction() 등으로 구성된다. 그림 8은 메시지 구조체를 보여준다. 메시지 타입, 메시지 내용과 크기를 나타내는 필드로 구성된다.



[그림 7] 이벤트 구조



[그림 8] 메시지구조

4.3 실험

게임논리엔진 방식과 일반적인 클래스 구현 방식의 구현 비용 차이 검증을 위하여 4.1에서 제시한 식을 바탕으로 실험을 하였다. 엔티티와 클래스의 수를 일정하게 유지하면서 규칙의 수를 증가시킨다. 구현 비용은 게임논리의 추가로 인해 구현이 변경되어야 하는 클래스의 수로 측정한다. 특정 논리의 추가로 영향을 받는 클래스가 많다면 클래스들이 보다 강하게 연관되어 있는 유연성이 부족한 구조라 할 수 있다.

실험에 사용된 게임논리는 사용자가 게임 맵 상에서 제시된 조건을 만족하면 다음 스테이지로 진행하기 위하여 문이 열리는 것이다. 이 조건을 점점 복잡하게 하면서 규칙의 수를 증가시킨다.

규칙집합 1: 아이템 발견 → 문 열림

규칙집합 2: 아이템 발견 → NPC 출현, NPC 만남 → 문 열림

규칙집합 3: 아이템 발견 → NPC와 적 그리고 장애물 출현, NPC 만나고 적을 제압하고 장애물 제거 → 문 열림

규칙 집합 1, 2, 3을 모두 구현하기 위해 필요한 엔티티 및 클래스는 Item, Gate, NPC, Enemy, Barrier, 총 5 가지이다.

표 1은 게임 논리 엔진을 사용하여 구현한 Item, Gate, NPC, Enemy 엔티티를 나타낸다.

[표 1] 실험 엔티티 구현

엔티티 이름	멤버변수
Item	mLocation : Vector3
	mFound : boolean
Gate	mOpened : boolean
NPC	mVisit : boolean
	mLocation : Vector3
Enemy	mLocation : Vector3
	mHP : int
Barrier	mDestroyed : boolean
	mLocation : Vector3

게임 논리엔진에서의 이벤트는 규칙집합 1, 2, 3을 구현하기 위한 것으로 이벤트의 수는 각 집합에 속해있는 규칙의 수와 일치한다. 표 2는 규칙집합 3의 'NPC 만나고 적 제압하고 장애물 제거 → 문 열림' 규칙을 스크립트를 이용하여 이벤트로 구현한 예이다. 이때 규칙은 NPC 만나는 것과 장애물 제거이다.

[표 2] 규칙집합 3의 이벤트 구현

Event Rule3-3

```
bool isFullfillStartCondition() {
  if NPC.mVisit && Enemy.mHP<0 && Barrier.mDestroyed then
    return true
  return false
}
bool isFullfillEndCondition() {
  if Gate.mOpened then
    return true
  return false
}
void process() {
  if | my_location - NPC.mLocation | < T then
    NPC.mVisited = true
  if | my_location - Enemy.mLocation | < T then
    Enemy.mHP = -1
  if | my_location - Barrier.mLocation | < T then
    Barrier.mDestroyed = true
}
```

게임 논리엔진에서는 규칙집합들을 스크립트 형태로 기술하기 때문에 클래스 방식으로 구현하는 경우에서의 클래스 멤버의 변경 또는 추가가 요구되지 않는다. 이는, 게임 논리 엔진에서는 표 1에서의 엔티티와 동일한 상황에서, 규칙집합 3을 구현하는 경우에 클래스 구현방식에서 멤버함수로 구현되는 논리를 스크립트 형태로 구현하기 때문이다.

클래스 방식으로 규칙집합 1을 구현하는 경우 아이템 클래스에 아이템 발견을 확인하는 checkItem() 함수가 추가가 필요하다. 규칙집합 2의 구현을 위해서는 아이템 클래스에는 아이템 발견과 NPC 만남을 처리하는 checkItem(), NPC 클래스에는 checkMeeting() 함수가 추가된다. 이와같은 방법으로 규칙집합 3에서는 아이템 발견과 NPC 만남, 적 제거, 장애물 제거를 표현하는 함수가 각각 추가되어야 한다. 표 3은 클래스로 구현한 경우

각 규칙집합의 구현을 위해 추가된 멤버함수들이다.

[표 3] 클래스 구현 방식에서 추가된 멤버함수

규칙집합	추가된 멤버함수	변경된 클래스 수
1	Item::checkItem()	1
2	Item::checkItem(), NPC::checkMeeting()	2
3	Item::checkItem(), NPC::checkMeeting(), Enemy::combat() Barrier::checkValidity()	4

표 4는 게임 논리 엔진과 클래스 구현 방식에서 논리 표현을 제외했을 때 필요한 클래스의 개수를 비교한 결과이다. 두 방법 모두 표 1의 엔티티 수에 해당하는 클래스들이 필요하다. 차이점은 클래스 구현 방식은 논리를 멤버함수들의 상호작용으로써 표현하는데 반하여, 게임 논리엔진은 표 2와 같이 이벤트 클래스로 논리를 기술한다는 것이다. 즉, 클래스 구현 방식에서는 클래스들 간의 n:n 의존되지만 게임 논리 엔진 구현방식에서는 이벤트와 클래스들 사이의 관계가 1:n 이 됨으로써 클래스간 의존도가 감소한다.

표 5는 게임 논리 엔진과 클래스 구현 방식에서 규칙이 변경됨에 따라 변경이 요구되는 클래스의 수에 대한 비교이다. 게임 논리엔진 방식은 항상 규칙과 같은 수의 이벤트를 사용하여 논리를 구현하는데 반해, 일반적인 클래스 방식을 사용하면 대체로 각 규칙에 관여하고 있는 클래스 대부분에 멤버함수를 추가해야 한다.

[표 4] 논리 표현을 제외한 경우 클래스 수

게임논리엔진	클래스 구현
5	5

[표 5] 변경이 요구되는 클래스의 개수 비교(괄호는 변경이 요구되는 클래스 종류)

규칙집합	게임논리엔진	클래스 구현
1	1(Event)	1(Item)
2	2(Event)	2(Item, NPC)
3	3(Event)	4(Item, NPC, Enemy, Barrier)

규칙이 간단한 규칙집합 1과 2의 경우 두 방식 모두 변경되는 클래스의 수는 같지만 종류가 다르다. 게임 논리엔진 방식에서는 이벤트 클래스를 상속한 클래스들을 변경하고 확장해 나간다. 따라서 규칙이 변경되면 이벤트 계열의 클래스들이 자동적으로 변경된다. 그러나 일반적인 클래스 구현방식은 논리의 특성에 따라 서로 영향을 주고 받는 다양한 클래스들이 변경되어야 하고 이는 단순히 변경이 요구되는 클래스의 수로 표현하기 어려운 복잡도의 증가를 야기한다.

이러한 현상은 다소 규칙이 복잡한 규칙집합 3의 경우 더욱 심해져서 클래스 구현방식에서 변경 클래스의 수도 게임 논리엔진 방식보다 증가한다. 규칙이 많아지고 규칙에 관여하는 클래스가 많아질수록 이러한 격차는 더욱 커질 것이다.

5. 결론 및 향후 연구 방향

게임논리엔진은 게임의 이야기흐름을 구성하는 다양한 게임논리들을 손쉽게 구현하고 조합할 수 있는 프레임워크이다. 제안된 게임논리엔진 방식은 기존의 클래스 제작 방식에 비해 훨씬 효율적이고 간단하게 이러한 논리의 조합을 표현할 수 있다. 게임논리엔진을 사용함으로써 게임논리모듈에서는 쉽게 구현할 수 없었던 복잡한 게임논리의 전개가 가능해진다.

현재 프로토타입 구현 및 테스트는 텍스트 방식으로 진행되었으며 외부 스트립트와의 연동을 구현하지 않은 상황이다. 추후 연구에서는 제안된 게임 논리 엔진을 다양한 유연성 평가 요소로 실험하고 3D 엔진과의 연동 및 스크립트를 구현하여 보다 실제적인 상황에서 게임논리 엔진 구조에 대한 평가를 진행할 예정이다.

참고문헌

[1] 통계청, 문화산업 매출규모, e-나라지표, www.index.go.kr.

[2] Mark Deloura, et.al, Game Programming gems, CHARLES RIVER MEDIA, 2001.

[3] A. BinSubaih, S. C. Maddock, and D. Romano, A survey of ‘game’ portability, Tech. Rep. CS-07-05, University of Sheffeld, Sheffield, UK, 2007.

[4] Ahmed BinSubaih, Steve Maddock, Using ATAM to Evaluate a Game-based Architecture, 20th European Conference on Object-Oriented Programming ECOOP 2006 July 3-7, 2006, Nantes, France.

[5] Ahmed BinSubaih, Steve maddock, Daniela Romando,

Game logic portability, Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology, Valencia, Spain, June 15-17th, 2005.

김 석 현(Seok-Hyun Kim) **[정회원]**

- 2001년 2월 : 서울대학교 재료공학부 (공학학사)
- 2008년 2월 : 서울대학교 컴퓨터공학부 (공학석사)
- 2008년 3월 ~ 현재 : 서울대학교 컴퓨터공학부 박사과정
- 2008년 3월 ~ 현재 : 남서울대학교 멀티미디어학과 전임강사

<관심분야>
운영체제, 시스템소프트웨어, 보안, 컴퓨터그래픽

이 면 재(Myoun-Jae Lee) **[종신회원]**

- 1992년 2월 : 홍익대학교 전자계산학과 졸업 (이학사)
- 1994년 2월 : 홍익대학교 전자계산학과 대학원 졸업(이학 석사)
- 2006년 8월 : 홍익대학교 전자계산학과 대학원 졸업(이학 박사)
- 2009년 3월 ~ 현재 : 백석대학교 정보통신학부 교수

<관심분야>
게임 인공지능, 게임 엔진