Journal of Digital Contents Society Vol. 8 No. 4 Dec. 2007(pp. 615-622)

스토리라인 기반의 자유로운 게임 플레이를 위한 게임 엔진 설계

김석현*

요 약

유저에게 다양한 게임 플레이 경험을 제공하기 위해 게임 엔진은 다양한 게임 내 개체 간 상호 작용을 다룰 수 있는 엔진 구조를 가져야 한다. 이를 위해 메시지 기반 게임 엔진 구조가 사용 된다. 그러나 메시지 기반 게임 엔진 구조만으로는 게임 세계를 특정한 스토리라인 기반으로 변화시켜 나가기 어렵다. 이는 메시지 기반 시스템과 같은 event-driven system 자체가 하나하나의 메시지 처리에는 적합하지만 이보다 상위의 보다 큰 논리적인 작업 단위를 처리하기에 적합한 구조는 아니기 때문이다. 이를 위해 storyline 개체를 만들고 지속적으로 storyline의 스토리 진행 함수를 호출함으로써 메시지 기반 시스템의 자유로움은 유지하면서 게임 세계에 특정 storyline의 진행을 추가할 수 있는 게임 엔진 구조를 제안한다.

The Game Engine Architecture for free game experience based on a storyline

Seokhyun Kim*

Abstract

A game engine should have the architecture that can manage various interactions between entities in a game for offering users various game experience. For this purpose, the game engine architecture based on message system is used. But only by this message based system, it is difficult to change game world continuously according to some storylines. The reason of this is event-driven system like message based system is appropriate for processing individual message but is not appropriate for processing more bigger logical work unit. For this purpose this paper proposes the storyline entity. The storyline entity has logical flows for a storyline and is called by engine continuously. By this proposed game engine architecture he or she can maintain free game experience by message based system and can add some progressing of storylines.

Keywords : game engine, architecture, event-driven system

1. 서론

게임 플레이에 있어서 특정한 목적을 설정하고 유저로 하여금 그 목적을 성취하도록 하는 게임과, 특정한 목적이 없이 자유롭게 게임 속 세상을 탐험하며 다양한 경험을 할 수 있도록 하는 게임이 있다. 전자의 방식은 전통적인 2D 횡스크롤 게임에서 많이 찾아볼 수 있다. 정해진 플롯을 따라 게임이 진행되며 유저는 게임에서 제시한 목적을 달성 해야만 다음 단계로 진행한다. 후자의 방식은 비교적 최근에 많이 찾아볼 수 있다. The Sims 시리즈, GTA 시리즈 등의 게임은 특별한 결말이 없이 유저가 만들어 나가는 열려진 결말의 게임이다. 이런 게임 스타일을 sandbox style이라고 부르기도 한다[1].

특히 MMORPG를 비롯한 다양한 온라인 게임은 유저가 즐길 수 있는 공간 속에서 게임 고유의 시스템을 만들어 놓고 이를 토대로 유저의 자유로운 플레이와 상호 관계로 게임 플레이 경험이 형성 되므로 대체로 sandbox style을 가지

※ 제일저자(First Author) : 김석현
접수일자:2007년09월18일, 심사완료:2007년09월28일
* 서울대학교 컴퓨터공학부
shkim@os.snu.ac.kr

고 있다고 말할 수 있다.

이렇게 자유로운 게임 플레이를 지원하기 위해서 게임 엔진은 다양한 개체 간의 상호 작용을 쉽게 지원할 수 있는 구조를 가져야 한다. 그러한 시스템은 여러 개체간의 상호 작용을 가능한 동일한 추상화를 통해 접근할 수 있도록 하는 것이 좋다. 게임 내에서의 여러 활동들을 하나의 추상화를 통한 동일한 접근 지점으로 수행할 수 있게 하는 것이다. 이런 구조를 취할 경우 게임 엔진의 확장성과 재사용성이 증가되며 개체간의 의존성은 줄어들게 된다. 여러 개체간의 다양한 상호작용을 동일한 시스템과 인터페이스를 통해서 수행할 수 있게 함으로서 얻게 되는 이점이다.

이러한 엔진 설계를 위해 유용한 방식으로 메시지 기반 시스템을 생각할 수 있다. [2]는 이러한 메시지 기반 게임 개체 관리 기법의 좋은 예를 보여준다. 이 방식을 이용하면 다양한 게임 개체의 상호 작용을 메시지를 통해서 매우 유연하면서도 통일성 있게 작성할 수 있다.

그러나 메시지 기반 시스템이 제공할 수 있는 게임 플레이 경험에는 한계가 있다. 메시지 기반 시스템은 일종의 event-driven system으로 바라볼 수 있다. Windows API 등의 여러 시스템에서 event-driven 방식을 이용해 메시지를 발생시키고 이를 처리하는 함수를 통해 이벤트의 발생에 따라 전체 시스템의 행동 방식이 결정되도록 한다. 게임 엔진에서 이러한 event-driven 방식만을 사용하여 게임이 진행되도록 한다면 유저는 게임 상의 세계가 수동적으로 느껴질 수도 있다. Event-driven 방식으로 게임을 만들면 유저가 게임 속에서 어떤 행동을 하고 이러한 행동의 응답으로써 여러 사건이 연쇄적으로 발생하게 된다. 일련의 사건의 시작은 언제나 유저의 어떤 행동이다. 유저의 플레이와 상관없이 게임 속의 세계가 어떠한 방향으로 흘러가고, 이에 따라 능동적으로 유저에게 먼저 '말을 거는' 게임 세계를 구축한다면 유저는 더욱 인상 깊은 경험을 할 수 있을 것이다.

이 논문은 어떤 storyline을 기반으로 게임 세계가 자율적인 방향으로 움직여 나가면서 동시에 sandbox style로 유저가 게임 세계 속에서 자유롭게 플레이를 하며 유저와 게임세계가 보다 능동적이고 평행한 관계로서 상호작용할 수 있는 게임 엔진 구조를 제시한다. 2장에서 [2]에 제시된 메시지 기반 게임 개체 관리 기법을 살펴보고 다른 메시지 시스템과 비교해 본다. 그리고 storyline을 게임에 추가하기 위해 메시지 시스템이 갖는 한계를 살펴본다. 3장에서 storyline 개체 및 전체 엔진 설계에 대해 서술하고 4장에서 엔진 프로토타입 구현을 서술하고 평가한다. 5장에서 향후 연구 방향에 대해 언급하고 6장에서 결론을 내린다.

2. 메시지 기반 게임 개체 관리

2.1 메시지 시스템

먼저 [2]에서 제시된 메시지 시스템에 대해 간략히 정리한다.

2.1.1 메시지 구조

메시지는 단순한 열거형 상수이다. EM_CLSINIT, EM_START, EM_UDATE 등의 메시지를 정의하고 각 메시지에 의미를 부여한다. 위에서 언급한 메시지는 차례대로 클래스 초기화, 개체 활성화, 개체 상태 update의 의미를 가지고 있다.

이렇게 다양한 메시지들은 다양한 parameter들을 필요로 한다. 그러나 공통된 메시지 인터페이스를 위해 두 개의 int 형 변수를 메시지와 함께 전달되는 매개변수로 사용한다. 예를 들어 클래스 초기화 메시지의 경우 매개 변수는 초기화에 필요한 어떤 정보가 될 것이다. 하지만 위치를 지정하는 메시지의 경우 매개 변수는 개체의 위치를 나타내는 3차원 좌표가 될 것이다.

2.1.2 메시지 전송 및 처리 함수

메시지 전송은 EntSendMessageg함수를 이용한다. EntSendMessage 함수는 다음과 같다.

<표 1> EntSendMessage

```
int EntSendMessage( ENTITY   *ent,
                    EM       message,
                    int      var1,
                    int      var2 );
```

ent 개체에 message를 var1, var2의 매개 변수와 함께 보내준다. 이렇게 메시지를 보내면 해

당 ent 개체의 타입을 확인하고 미리 그러한 타입에 대해 등록된 메시지 처리함수가 호출되게 한다. 같은 타입이라도 인스턴스에 따라서 상태가 다르므로 메시지 처리 함수는 ent를 첫 번째 인자로 받아서 같은 메시지 처리 코드로 ent 개체에 대해서 메시지 처리 작업을 하게 된다. 그러한 메시지 처리 함수는 다음과 같은 모습이다.

<표 2> ENT_PROC

```
typedef int (*ENT_PROC) (
                    ENTITY      *ent,
                    EM          message,
                    int         var1,
                    int         var2   );
```

[2]는 C 기반의 구현을 제공하고 있다. C++를 사용하면 메시지를 보내고 이를 해당 객체에 맞는 처리 함수로 처리하는 과정을 가상함수의 호출을 통해 대신할 수 있다. 메시지 처리 방식과 가상함수 방식은 비슷한 일을 한다고 볼 수 있지만 장단점에 있어서 많은 차이가 존재한다.

EntSendMessage로 보내진 메시지는 메시지를 받는 entity class에 등록된 ENT_PROC이 호출되게 함으로써 처리된다.

2.2 다른 메시지 시스템과의 비교

2.2.1 윈도우즈 메시지 시스템

2.1에서 설명한 방법은 윈도우즈의 메시지 시스템과 유사하다. 윈도우에서 사용되는 struct MSG는 메시지를 받는 윈도우 핸들, 메시지 identifier, 두 개의 4byte parameter, 메시지가 보내진 시간, 메시지가 보내질 때의 화면상에서의 커서 위치가 들어간다.

윈도우는 시스템 전체에 하나가 존재하는 시스템 메시지 큐와 GUI 쓰레드 당 하나씩 존재하는 쓰레드 메시지 큐를 가지고 있다. 발생된 메시지는 일단 시스템 메시지 큐에 들어갔다가 이후 해당 윈도우에 분배 된다. 모든 메시지가 언제나 반드시 메시지 큐를 거치는 것은 아니다. 메시지의 종류에 따라서, 프로그래머의 선택에 따라서 메시지가 메시지 큐를 거치지 않고 직접 해당 윈도우에게 전달되기도 한다[3].

2.1의 경우 해당 개체에게 메시지가 직접 전달되는 EntSendMessage만 있지만 윈도우는 PostMessage 함수가 있다. PostMessage를 사용하면 메시지가 바로 전달되지 않고 위에서 설명한대로 메시지 큐를 거쳐서 해당 윈도우에게 전달된다.

PostMessage를 통해 메시지 큐로 들어간 메시지들은 메시지의 종류에 따라서 메시지 큐 안에서 적절한 관리를 받기도 한다. 예를 들어 큐 안에 다수의 WM_PAINT 메시지가 존재하면 이들은 하나로 합쳐진다.

2.1의 메시지 시스템과 비교해 보면 기본적으로 메시지를 해당 개체에게 전달하는 개념은 유사하다. 하지만 GUI 시스템의 경우 SendMessage 뿐만 아니라 PostMessage도 존재한다. 그리고 메시지 큐에 쌓인 메시지가 윈도우에 전달되기 전에 적절한 처리를 메시지들에 수행할 수 있다.

2.2.2 Qt의 signal/slot 메커니즘

Qt는 Trolltech 사에서 개발한 cross platform GUI toolkit이다. C++기반이며 메시지 처리에 있어서 signal/slot이라는 독특한 메커니즘을 제공한다.

2.2.1의 윈도우 메시지 시스템을 사용할 때 한가지 단점으로 지적할 수 있는 것은 메시지의 parameter를 전달함에 있어서 프로그래머가 실수를 하면 이를 찾는 것이 매우 어려울 수 있다는 점이다.

Qt는 GUI widget이 특정 signal을 발생 시킬 때 parameter의 개수와 type을 지정해 줄 수 있다. 따라서 프로그래머가 잘못된 parameter를 집어넣으면 컴파일러가 이를 찾아주기 때문에 윈도우 메시지 시스템과 같은 문제를 피할 수 있다.

그러나 이러한 signal/slot 메커니즘을 위해 Qt는 표준 C++ keyword가 아닌 자신들만의 키워드를 제공하며 이를 moc(meta object compiler)를 이용해서 표준 C++ 키워드를 기반한 코드로 자동 변환한다. 즉, Qt의 signal/slot과 같은 방식은 2.1, 2.2.1의 방식에 비해 메시지의 parameter의 type 및 개수를 명확히 하여 보낼 수 있는 장점이 있지만 moc와 같은 보조적인 tool을 만드는 overhead가 존재하다[4].

2.3 메시지 시스템의 한계

2.3.1 Event-driven vs Multithreaded

앞에서 살펴본 시스템들은 메시지 기반의

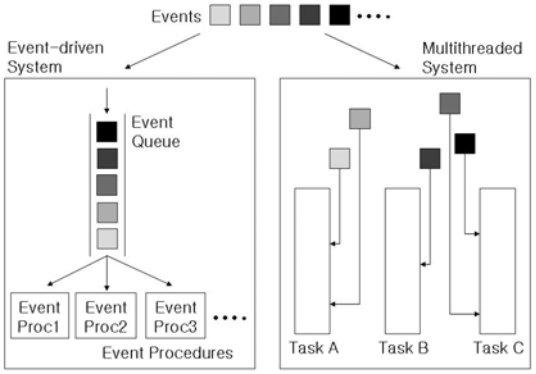
event-driven system이다. 이러한 시스템은 개별 메시지들을 적절한 처리 함수에서 처리하게 함으로써 전체 시스템을 구동 시킨다. 이러한 방식으로도 필요한 모든 작업을 할 수 있지만 여러 메시지들이 보다 큰 하나의 논리적인 단위의 작업을 위해 협업해야 하는 경우 전체 시스템의 복잡도가 크게 증가하는 단점이 있다.

예를 들어 게임에서 다양한 조건에 따라 여러 갈래로 뻗어 나갈 수 있는 quest를 설계하는 경우를 생각해 보자. Event-driven 방식으로 프로그램을 짜게 되면 quest와 관련된 여러 가지 현재 상태를 어떤 개체에 저장해 놓고 여러 메시지 처리 함수에서 이 상태를 참고하면서 quest를 진행하는 코드를 작성하게 된다. 이렇게 되면 해당 quest와 관련된 코드는 게임 코드 전체로 분산되고 코드의 유지보수 및 디버깅이 매우 어려워진다.

이상과 같이 개별 이벤트로 처리하기 곤란한 하나의 논리적인 작업 단위를 처리하기 유용한 모델로써 Multithreaded system을 생각할 수 있다. Event-driven system과 Multithreaded system의 비교는 다른 분야에서도 종종 이루어진다[5]. Event-driven system의 장점은 효율성에 있다. 모든 이벤트 프로시저가 하나의 이벤트 큐를 공유하므로 메모리 사용에 있어서 효율적이다. 또한 다양한 이벤트 처리에 있어서 좋은 성능을 보여준다. 그러나 단점은 여러 이벤트가 다양하고 복잡하게 사용되며 상호 연관될 경우 코드의 이해나 디버깅 작업이 매우 어려워진다는 것이다. Multithreaded system의 장점은 코드의 이해가 쉽고 디버깅이 용이하지만 동기화 overhead가 있다는 단점이 있다. 또한 coding을 하는데 있어서 보다 숙련된 기술이 필요하다.

(그림 1)은 여러 이벤트들을 처리하는데 있어서 event-driven system과 multithreaded system은 어떤 차이를 가지고 있는지 보여준다. 전자는 이벤트 큐에 여러 이벤트를 쌓아 놓고 이를 이벤트 처리 함수로 분배하는 구조이다. 후자는 다양한 이벤트를 이벤트 단위로 처리하는 것이 아니라 보다 큰 논리적인 작업 단위를 처리하는 처리 함수들이 존재하고 이 함수들이 multithread 방식으로 동작 하면서 쓰레드 간 context switch 및 여러 동기화 매커니즘을 통해 다양한 이벤트들이 처리 된다.

2.3.2 게임 엔진에서의 대안적 시스템



(그림 1) Event-driven vs Multithreaded

Event-driven system의 약점에도 불구하고 게임에서 multithreaded system을 본격적으로 사용하기는 어렵다. 그러한 모델로 엔진을 설계하게 되면 게임 엔진에는 OS의 요소가 들어오고 quest, storyline 등의 개체는 이러한 OS적인 요소를 지니고 있는 게임 엔진 상에서 작동하는 프로그램과 같은 위상을 지니게 될 것이다. 그러나 이렇게까지 무겁고 복잡한 시스템이 게임 엔진에 본격적으로 필요한 시점은 아니다.

그래서 3장에서 앞에서 언급한 multithreaded system에서 힌트를 얻어 간단하지만 특정 quest나 storyline의 흐름을 하나의 코드에 집중 시키고, 이렇게 가독성을 높임으로써 storyline이 보다 능동적으로 유저의 게임 경험에 영향을 줄 수 있는 게임 엔진 설계에 서술한다.

3. 엔진 설계 제안

3.1 StoryLine 개체

StoryLine 개체는 여러 개체가 참여하는 story의 진행과 같은 큰 흐름을 coding하기 위해 전체적인 logic의 흐름을 개발자가 쉽게 확인할 수 있도록 가독성을 향상시킴으로써 엔진 개발의 정확성과 효율성을 높이는데 첫 번째 목적이 있다. 또한 StoryLine 개체는 실시간에 storyline에 변화를 줌으로써 유저의 게임 내에서의 다양한 행동에 보다 적극적으로 대응할 수 있는 구

조를 지향한다.

이를 위해 StoryLine 개체는 내부적으로 다음과 같은 구조체의 배열을 갖는다. 구조체는 이벤트의 발생 시간과 이벤트 처리 함수 포인터를 갖고 있다.

<표 3> EVENT 구조체

```
struct EVENT {
        DWORD    time;
        int          (*evt_proc)( StoryLine* );
};
```

StoryLine 개체에 여러 이벤트를 설정하면 다음 그림과 같은 내부 자료 구조가 형성 된다.

(그림 2) Storyline 개체의 EVENT 배열

StoryLine 개체는 마치 program counter와 유사하게 다음에 실행될 이벤트가 무엇인지 저장해 놓는다. executeStory 함수가 호출 되면 현재 게임 시간과 다음에 실행될 이벤트의 발생 시간을 비교하고 만약 게임 시간이 다음에 실행될 이벤트의 시간보다 크거나 같으면 해당 이벤트를 실행하고 다음에 실행될 이벤트를 방금 실행된 이벤트의 다음 이벤트로 변경한다.

호출되는 이벤트 처리 함수 내에 게임 내의 다양한 개체들에게 메시지를 보내고 개체들의 여러 상태를 조정함으로써 미리 정해 놓은 이벤트의 흐름 즉, storyline이 실행 되도록 한다. StoryLine 개체 안에 유저의 반응에 따라 동적으로 story 흐름을 바꿀 수 있도록 A.I. 등을 집어넣을 수도 있을 것이다.

StoryLine 개체는 multithreaded system의 task와 유사한 면을 가지고 있다. 마치 프로그램처럼 StoryLine 개체에 등록된 story 전개 함수들을 시간순서대로 호출해 나간다. 또한 여러 조건에 따라서 실시간에 특정 시간대의 story 처리 함수에 대해 교체, 삽입, 삭제 연산을 수행함으로써 동적인 story 전개의 변화도 가능하다.

3.2 StoryLine 개체를 위한 main loop

StoryLine 개체는 자신의 logic을 가지고 시간 흐름에 따라 그 logic을 전체 게임에 적용 시켜 나간다. 따라서 게임의 main loop에서 StoryLine 개체의 executeStory 함수를 지속적으로 호출해야 한다.

StoryLine 개체들의 executeStory 함수를 모두 호출한 다음에 게임 엔진 내부 메시지 큐에 저장되어 있는 메시지들을 처리하게 된다. 이를 pseudo code로 표현하면 다음과 같다.

<표 4> Main loop pseudo code

```
for each storyline entity
        call executeStory of storyline entity
while message queue is not empty
        call DispatchMessage
```

3.3 메시지 큐에서의 메시지 처리

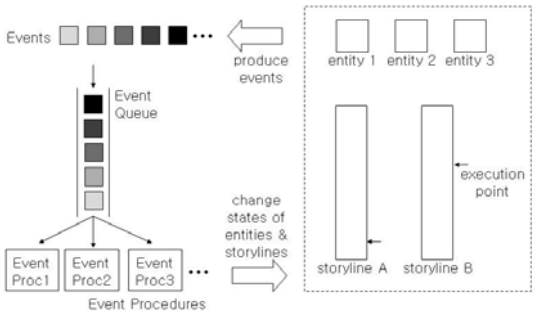
2.1에서 서술한 윈도우 GUI에서 시스템 메시지 큐에 있는 메시지 중 특정 메시지에 대해서 시스템의 목적에 맞는 처리가 들어갔던 것처럼 게임 엔진 메시지 큐에도 메시지에 대한 처리를 할 수 있다.

메시지 간에는 우선순위가 존재할 수 있다. 논리적으로 어떤 메시지는 반드시 다른 메시지에 선행해서 처리되는 경우가 있다. 예를 들어 개체를 랜더링하는 경우 반투명한 개체는 불투명한 개체들 보다 나중에 랜더링 되어야 한다. 이런 경우 메시지의 우선순위를 미리 지정해 놓고 이 우선순위에 따라 메시지들을 큐에서 sorting한 다음 각 개체에게 보내주면 메시지 간의 우선순위 문제를 해결할 수 있다.

또한 여러 시스템의 필요에 따라 메시지 큐 안에 있는 메시지들에 대해 여러 가지 처리가 추가될 수 있다.

3.4 전체 구조도

이상에서 설명한 엔진 구조에 대한 전체 구조도는 (그림 3)과 같다.



(그림 3) 게임 엔진 전체 구조도

4. 구현 및 평가

4.1 구현 내용

논문에서 제안한 엔진의 프로토타입을 Windows XP, Visual Studio 2005 환경에서 C++로 구현하였다. 아직 3D 엔진과 연동시키지는 않고 텍스트 환경에서 각 개체의 메시지 발생과 처리 과정을 살펴보았다.

(그림 4)는 VS2005에서 본 엔진 프로토타입의 주요 클래스들이다. 이 클래스들은 sege라는 네임스페이스 안에 들어 있다.

(그림 4) 주요 Class들

Entity class는 게임 엔진 내부의 개체를 표현한다. processMsg라는 가상 함수가 메시지 처리를 담당한다. EntityFactory는 게임 엔진에 Entity class를 상속한 게임 개체들을 종류에 맞게 생성하는 객체이다. Entity를 상속하여 새로운 개체 클래스를 만들고 클래스의 이름과 클래스 생성 함수의 포인터를 등록한다. 예를 들어 Entity를 상속한 Character라는 클래스를 등록하기 위해서 다음 <표 5>와 같은 코드가 필요하다.

<표 5> 클래스 등록 코드

```
// Character class 정의
class Character  { .... };
// Character 객체 생성 함수
Entity* CreateChar()
{ return new Character; }
// 클래스 이름과 생성 함수 등록
RegisterEntity("Character", CreateChar);
```

이러한 방식으로 각 게임에 특화된 개체들을 Entity 클래스를 상속하여 정의하고 등록함으로써 다양한 게임에 엔진의 시스템을 재사용할 수 있다.

MsgSystem 개체는 (그림 3)의 게임 엔진 전체 구조도에서 메시지 큐에 들어온 메시지들을 여러 Entity 개체 및 StoryLine 개체의 메시지 처리 함수로 분배시켜 주는 역할을 하는 메시지 시스템의 핵심 클래스이다. 그리고 3장에서 제안한 StoryLine 개체는 Entity class를 상속하여 다른 개체와 마찬가지로 메시지를 받을 수 있다.

엔진 프로토타입이 외부로 노출하는 엔진 API는 다음 <표 6>과 같다.

<표 6> 엔진 API

함수 이름	역할
InitSege	엔진 초기화
FinishSege	엔진 종료
RegisterEntity	Entity를 (이름, 생성함수) pair로 등록
CreateEntity	Entity를 생성
SendMsg	메시지를 바로 처리. 함수를 호출하고 메시지 처리 완료 후 리턴
PostMsg	메시지를 나중에 처리. 메시지 큐에 쌓여 있다가 추후 처리됨
ExecuteStoryLines	StoryLine 개체들의 스토리 실행 함수들을 모두 호출해줌
ProcessPostedMsgs	PostMsg로 메시지 큐에 쌓여있는 메시지들을 실행함

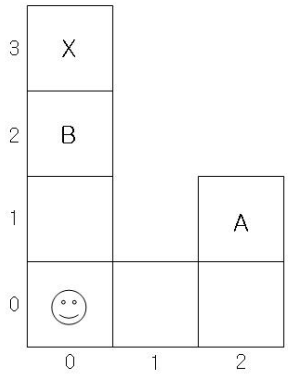
4.2 테스트 시나리오

제안한 엔진 프로토타입 설계 방식은 메시지 기반 시스템에 StoryLine 개체를 추가한 것이다. 이러한 방식을 평가하기 위해 하나의 간단한 테스트 시나리오를 두 가지 방식으로 구현하였다. 우선 메시지 기반 시스템만을 이용하였고 다음

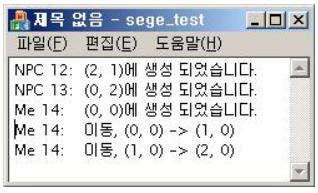
으로 메시지 기반 시스템과 StoryLine 개체를 이용 하였다.

테스트 시나리오에 사용한 게임맵은 (그림 5)와 같다. 게임맵은 모두 7개의 방으로 구성 되어 있다. 방의 배치는 그림과 같다. 먼저 스마일 표시가 있는 위치에 자신의 아바타가 위치한다. A, B는 NPC이다. 아바타가 A와 B를 모두 먼저 만난 다음에 X 표시가 있는 방에 갈 수 있다.

(그림 6)은 게임 엔진을 간단한 텍스트 기반 프로그램에 연결하여 NPC A, B와 아바타가 생성되었음을 표시해주는 모습이다. 나의 아바타는 키보드의 방향키를 누르면 다른 방으로 이동할 수 있다. 게임 멥의 구조에 따라 현재 아바타의 위치에서 이동 가능한 방향으로 방향키를 누르면 이동하게 된다. 이러한 이동의 결과 역시 (그림 6)에 표시 되어 있다.



(그림 5) 테스트 게임맵



(그림 6) 초기화 및 아바타 이동 결과

(그림 6)에서 A, B 및 아바타의 엔진 내부 아이디가 각각 12, 13, 14 이기 때문에 이 번호로 표시 되어 있다.

평가는 이상과 같은 테스트 시나리오를 얼마나 유연하고 효율적으로 작성할 수 있는 가에 초점을 맞추었다.

4.3 두 가지 구현 방식의 비교

4.3.1 메시지 시스템만을 이용한 구현

테스트 시나리오를 구현하기 위해 먼저 (그림 5)와 같은 상태로 맵을 설정하는 초기화 함수를 만들었다.

다음으로 여러 개체들에서 공통으로 접근할 수 있는 상태의 모임을 하나의 구조체로 만들었다. 이 구조체에는 Map, NPC, 아바타의 포인터와 아바타가 A, B 각각을 만났는지의 여부, X로 표시된 방으로 아바타가 접근 가능한지의 여부가 들어 있다. 즉 이 구조체에는 테스트 시나리오 구현을 위한 자료구조가 집중 되어 있다.

아바타가 NPC와 만났는지를 확인하는 코드는 Map 클래스에 구현 하였다. Map은 모든 방을 알고 있으므로 각 개체의 충돌을 확인하기에 적당하다. 충돌 확인은 게임의 메인 loop에서 매번 확인해야 하므로 이 충돌 확인 함수를 메인 loop에서 호출하도록 하였다.

NPC와 아바타의 충돌이 발생 하면 시나리오의 전체 상황이 기록되어 있는 구조체의 내용을 변경한다. 아바타를 이동 시키는 함수에서 이 구조체의 내용을 확인하여 X로 표시된 방으로의 접근 권한이 바뀌면 그 방에 들어갈 수 있게 한다.

4.3.2 메시지 시스템과 StoryLine을 이용한 구현

StoryLine을 이용한 구현에서는 맵 초기화 함수를 <표 3>의 이벤트 구조체를 이용하여 StoryLine의 첫 번째 이벤트 처리 함수에서 처리하게 하였다.

시나리오를 구성하는 Map, NPC, 아바타의 포인터 및 기타 시나리오를 위한 상태 정보들은 StoryLine 개체의 멤버 변수로 선언 하였다.

충돌을 체크하는 함수는 4.3.1의 방식과 마찬가지로 Map 클래스에 구현하여 main loop에서 매번 호출하도록 하였다.

또한 StoryLine을 이용하여 추가적인 NPC의 움직임을 구현 하였다. NPC를 이동할 수 있게 하는 이벤트 처리 함수를 만들어 놓고 일정한

확률로 StoryLine 개체에 NPC 이동 이벤트를 추가 하였다.

4.3.3 두 구현 방식의 비교 평가

StoryLine 개체의 첫 번째 특성은 엔진에 하나하나의 메시지 처리 이상의 보다 큰 단위의 논리적 작업을 위한 개체를 만드는 것에 있다. StoryLine 개체라는 틀이 있으면 개발자는 자연스럽게 이 개체에 해당 스토리와 관련된 자료 구조 및 함수를 모으려고 할 것이다. 하지만 이 점을 특별한 장점이라고 보기는 어렵다. 메시지 시스템만을 위한 구조에서도 스토리를 위한 클래스를 하나 선언하면 되기 때문이다.

StoryLine 개체의 중요한 특성은 이벤트 구조체를 이용하여 게임의 진행 방식을 쉽게 바꿀 수 있다는 것이다. 선행 이벤트의 결과와 게임 내부 여러 개체들의 상황을 보면서 이후 이벤트의 구성을 변화시킴으로써 StoryLine 개체는 보다 능동적이고 다양한 게임의 흐름을 가능하게 할 수 있다.

5. 향후 연구

StoryLine 개체는 각 이벤트를 시간 순서에 따라서 발생 시킨다. 즉, 이벤트를 발생시킬 수 있는 조건이 시간에 한정 되어 있다.

하지만 게임 내에서는 훨씬 다양한 조건들을 비교 평가할 수 있다. 여러 가지 게임 내의 상태에 따라 특정 이벤트를 발생 시킬 수 있게 함으로써 StoryLine 개체의 응용성을 보다 확장할 것이다.

StoryLine 개체는 Linux의 데몬과 같은 역할을 게임 내에서 수행할 수 있는 방법을 찾으며 설계 되었다. 즉, 게임 내부에서 특정한 서비스 내지 활동을 독자적으로 수행하는 개체로서의 역할을 수행할 수 있는 구조를 연구할 것이다. 이러한 생각은 게임 엔진을 일종의 가상 머신이나 OS로써 바라보고 그 위에서 작동하는 프로그램과 유사한 모델을 만들고자 하는 전망과 맞닿아 있다.

6. 결론

유저에게 다양한 게임 플레이 경험을 제공하기 위해 개임 개체 간의 다양한 상호 작용을 표현할 수 있는 엔진 구조가 필요하다. 이를 위해 메시지 기반의 개체 관리 구조가 제안 되었다.

본 논문에서는 이 구조를 더욱 발전 시켜서 StoryLine 개체를 포함한 게임 엔진 구조를 제시하였다. StoryLine 개체는 게임 세계 속에서 능동적으로 특정 story를 진행 시켜 나감으로써 유저로 하여금 보다 다양한 플레이 경험을 얻게 할 수 있는 가능성을 보여준다.

또한 StoryLine 개체의 동적인 이벤트 설정 구조를 통해 게임 내의 여러 상태에 맞추어 게임의 흐름을 동적으로 바꿀 수 있다. 이러한 특성을 통해 메시지 기반 게임 엔진 구조의 미덕인 자유로운 게임 플레이를 유지 하면서 특정한 storyline의 흐름을 게임 속에 부여할 수 있다.

참 고 문 헌

[1] http://en.wikipedia.org/wiki/Sandbox_%28video_games%29

[2] Matthew Harmon, 류광 역, “게임 개체 관리를 위한 시스템”, Game Programming Gems4, ISBN 89-5674-238-3, 정보문화사, pp. 145-163.

[3] http://msdn2.microsoft.com/en-us/library/ms644927.aspx

[4] http://doc.trolltech.com/4.3/signalsandslots.html

[5] B Gu, Y Kim, J Heo, Y Cho. Shared-stack cooperative threads. In Proc. of the 2007 ACM symposium on Applied computing, pp. 1181-1186, New York, NY, USA, 2007

김 석 현

2001년 : 서울대 재료공학부(학사)
2001년～2004년 : 엔틱스소프트(현 레드덕)
2004년～2005년 : 이네트
2006년～현 재: 서울대학교 컴퓨터공학부 대학원 석사과정
관심분야 : 컴퓨터 게임, 3D Graphics, OS, System Software, 정보 보안 등