부록

단위 기능과 시스템

이름 : 조대훈

1. ‘단위 기능’이 모여 ‘시스템’, ‘시스템’이 모여 게임

‘단위 기능’들이 모여서 하나의 ‘시스템’을 만들고, ‘시스템’이 모여 게임을 만든다는 말이 있습니다. 게임 개발자들은 이러한 말의 의미를 잘 이해했다고 생각합니다. 정말로 잘 이해한 분들이 있을지 모르지만, 저처럼 ‘유니티’로 게임 개발에 접한 사람은 부분적으로 잘못 이해했을 수도 있습니다.

1.1 잘못된 접근

잘못된 접근의 가장 큰 요인은, 유니티의 편의성입니다. 유니티는 매우 간단하게 게임 객체 생성하고, 해당 객체에 ‘클래스로 작성된 기능’을 컴포넌트 시켜 게임을 동작시킬 수 있습니다. 또한 ‘기능 클래스’와 ‘데이터 클래스’에 대한 구별이 따로 없습니다. 이러한 접근 방법은 다음과 같은 사고를 갖도록 합니다.

- **단위 기능구현 설계 -> 단위 기능 구현**
- **시스템 설계 (단위 기능들의 연동 방법 구상) -> 시스템 구현**

이러한 사고는 이후 기능이 추가되면 다음과 같은 설계 및 구현 순서를 갖게 됩니다.

- **추가할 단위 기능 발생. -> 단위 기능 구현**
- **시스템 설계 -> 기존 단위 기능 수정 -> 시스템 구현 [.....반복.....]**

추가할 단위 기능이 발생할 때마다, 기존에 존재하는 모든 클래스를 수정해야 하는 경우가 발생하는 것입니다.

1.2 올바른 접근

과거로 돌아가 봅시다. OpenGL 과 같은 그래픽스 엔진을 사용하여 게임 개발을 하던 시기, 선배 개발자들은 필요한 '시스템'과 '단위 기능'을 어떤 순서대로 구현했는지 생각해 봅시다.

우리가 사용하는 유니티와 달리 OpenGL 은 Main 함수에서부터 코드가 시작됩니다. 여기서 '단위 기능'을 바로 구현하는 것도 가능은 하겠지만, 더러운 코드를 기피하는 개발자 특성상 연관된 '단위 기능' 클래스들을 하나의 '시스템' 클래스에 넣어서 관리하였을 것입니다. 우리는 여기서 다음과 같은 규칙을 찾을 수 있습니다.

- '모든 기능 클래스'의 생명 주기는 '시스템' 클래스의 생명 주기보다 작거나 같다.
-> 시스템 클래스가 선언 및 정의된 후에, 각 기능 클래스가 선언 및 정의된다.
-> 시스템 클래스가 종료되면, 모든 기능 클래스는 종료된다.

선배들은 하나의 '시스템 클래스' 내에서 **연관된** '단위 기능' 클래스들을 관리하였을 것입니다. **연관된** '단위 기능' 클래스들은 **동일하거나 비슷한 데이터**를 사용하는 경우가 많았을 것입니다. 선배들은 '단위 기능' 클래스들이 **동일한 '데이터 클래스'를 참조**하여 사용하도록 하였을 것입니다.

- '모든 단위 기능 클래스'들은 '동일한 데이터 클래스'를 참조하게 하면 편리하다.

이러한 코드 구성을 갖게 된 선배들은 다음과 같은 생각을 하게 되었을 것입니다.

- '모든 단위 기능 클래스'가 하나의 '데이터 클래스'에 의해 연동하여 동작하도록 설계하자.

이러한 접근 방식은 다음과 같은 사고를 갖도록 합니다.

- 시스템 분류 및 설계 -> 시스템 클래스 선언 및 정의 + 데이터 클래스 선언 및 정의.

- 단위 기능 설계 -> 데이터 클래스 수정 -> 단위 기능 구현.

이러한 사고는 이후 기능이 추가되면 다음과 같은 설계 및 구현 순서를 갖게 됩니다.

- 추가 단위 기능 설계 -> 데이터 클래스 수정 -> 단위 기능 구현.

새로운 기능이 추가되어도, '데이터 클래스'에 새로운 필드 멤버만 추가하면 됩니다. 이는 기능의 확장성에 많은 이점을 제공할 것입니다.

2. 개선된 접근

과거의 접근 방식과 유니티의 장점을 잘 합쳐봅니다. 유니티의 큰 장점은 '단위 기능' 구현이 매우 편리하고 가시적이라는 것입니다. 유니티 이벤트 함수들과 컴포넌트 방식으로 인해, '단위 기능 클래스' 구현이 너무나도 편리합니다. 또한, 유니티 이벤트 함수는 각 클래스가 개별적으로 실행될 수 있도록 해주어, '단위 기능 클래스'들이 '시스템 클래스' 안에서 선언 및 정의 될 필요가 없어졌습니다.

따라서, 저는 유니티에서 다음과 같은 설계 방식을 갖게 되었습니다.

- 시스템 분류 및 설계 -> 데이터 클래스 선언 및 정의 (데이터 초기화)

- 단위 기능 설계 -> 데이터 클래스 수정 -> 단위 기능 구현.

과거와는 달리 시스템을 분류만 할 뿐, 시스템 클래스는 존재하지 않습니다. 이럴 경우 데이터 클래스가 선언 및 정의될 부분이 필요합니다.

2.1 독립된 데이터 클래스

이러한 문제점은 제가 앞서 말한 데이터 클래스를 독립하여 존재하게 하면 해결됩니다. 이러한 방식은 '데이터 클래스'에 대한 접근이 수월하게 할 뿐만 아니라, 프로젝트 내 존재하는 모든 클래스 간의 의존성 줄여주는 데 효과적입니다.

※ 'Initialization(초기화) 단위 기능'

시스템 클래스가 없기 때문에, 데이터 클래스의 초기화 시점을 신경 써야 합니다. '시스템이 사용하는 데이터 클래스'는 '시스템 클래스'의 필드 멤버이므로 생명주기가 동일합니다. 그러니, '시스템 클래스'가 없어도 '데이터 클래스'의 생명주기는 '시스템 클래스'와 동일하게 취급해 주어야 합니다.

- 해당 데이터 클래스가 처음으로 사용되는 시점들을 확인합니다.
- 해당 데이터 클래스들에 대한 '초기화 기능을 수행하는 단위 기능'을 만듭니다.
- '초기화 단위 기능'을 제외한 '단위 기능'의 Controller 클래스들을 비활성화 시켜 놓습니다.
- '해당 데이터 클래스'가 초기화됨과 동시에, 관련된 '단위 기능'의 Controller 클래스들을 활성화 시켜 해당 기능들을 사용하도록 합니다.

2.2 MVC 방식으로 나뉜 단위 기능 클래스

'단위 기능' 클래스를 MVC 패턴의 분류 방식으로 나누면, 각 클래스의 책임이 명확해집니다.

2.2.1 Data 클래스

- Data 의 선언 및 정의, Data 의 올바른 Get 과 Set 을 제공해야 합니다.
- Data 의 변동 여부를 Data 를 사용하는 클래스들에게 알려주기 위해서,
  Observer 의 Subject 역할을 수행합니다.

2.2.2 InteractionView 클래스

- 다양한 방식의 사용자 입력에 반응하여 작동해야 합니다.
- '단위 기능 로직'을 수행시킬 'Controller 클래스'를 미리 참조하고 있어야 합니다.

2.2.3 Controller 클래스

- 다양한 InteractionView 클래스가 접근할 수 있는 메소드를 제공해야 합니다.
- 간혹, Data 클래스의 변경에 반응하여 Observer 의 Subscriber 역할을 수행해야 합니다.

2.2.4 Model 클래스

- '단위 기능' 클래스의 로직 구현을 수행합니다.
- 'Data 클래스'의 Data 를 Set 하였을 경우, 관련된 Observer-Subject 의 Notify 메소드를
  호출해줘야 합니다.

2.2.5 View 클래스

- 다양한 'Data 클래스'를 참조하여, UI / UX 를 구성하면 됩니다.
- Data 가 동적으로 변동하는 걸 인식하기 위해 Observer 의 Subscriber 역할을 수행합니다.

2.3 MVC 방식으로 나뉜 단위 기능 클래스의 장점

이 처럼 '단위 기능'을 보다 세밀하게 나누는 방식은, 설계와 더불어 생각해 볼 때 큰 장점을 가져옵니다.

1) 설계 초반

- 시스템의 주요 기능 로직이 고정됩니다.
- 개발자는 Model 과 Data 를 클래스 부분을 완성 시킬 수 있습니다.

2) 설계 중반

- 사용자와 상호작용을 하는 방식이 고정됩니다.
- 개발자는 InteractionView 와 Controller 클래스 부분을 완성 시킬 수 있습니다.

3) 설계 후반

- 사용자에게 보여주는 UI / UX 가 고정됩니다.
- 개발자는 엔진을 통해 UI / UX 를 구성하고, View 클래스 부분을 완성 시킬 수 있습니다.