Vampire Survivors

(Event driven programming, Component Based Development)

NAME : 권 성 호

TEL : 010-8874-6452

PART : PROGRAMING

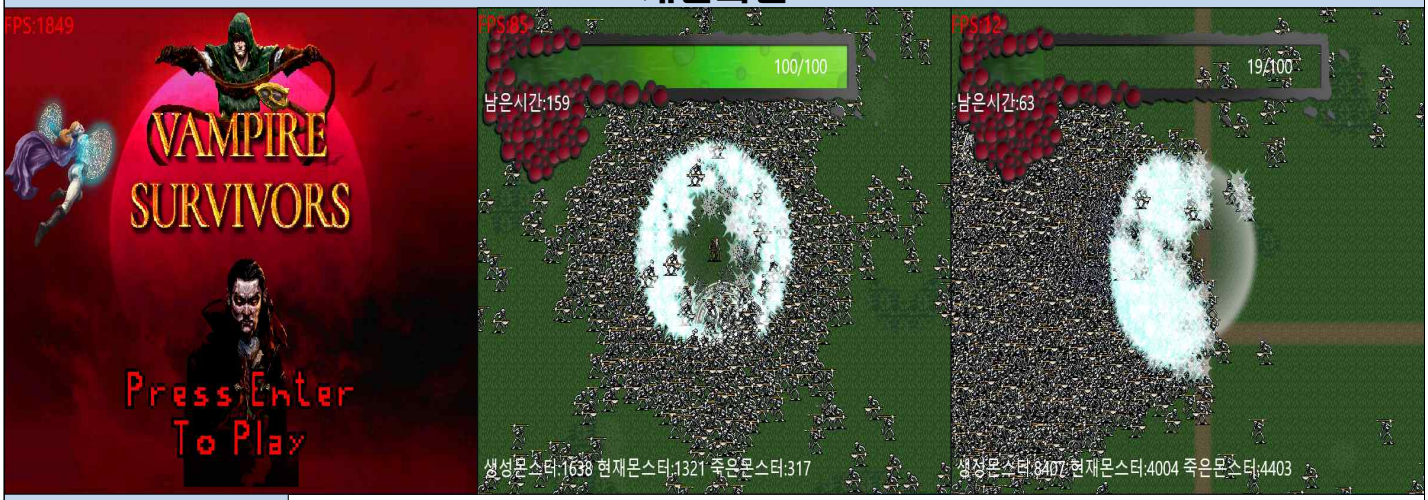
Efforts make all doors open

◎ 목 차 ◎

Ⅰ. 포트폴리오 개요

1. Vampire Survivor

게 임 명	Vampire Survivor
플 렛 폼	PC(Window)
게임화면	



게임 소개	1. 유명 스팀 게임 뱀파이어 서바이벌 모작 2. 이벤트 기반 프로그래밍 3. 컴포넌트 기반 프로그래밍 4. 컴포넌트와 프레임레이트를 이용한 충돌 처리 최적화 5. 렌더링의 최적화
개발 환경	VS2022,FMOD,DXTK
개발 인원	권성호
설명	컴포넌트 기반 프로그래밍 + 이벤트 기반 프로그래밍을 조합하여 객체의 생성, 파괴, 씬의 전환, 충돌을 이벤트화하여 Unity3D의 인터페이스를 구현하였습니다. 사용자 친화적인 게임 제작이 가능하여지도록 하는 것이 목표로 제작하였다.
동영상 링크	https://youtu.be/WgyP4H0j3Wo?si=W2BZg0oxUBsGadu3
GitHub URL	https://github.com/AshOne91/KGCA/tree/2DSample

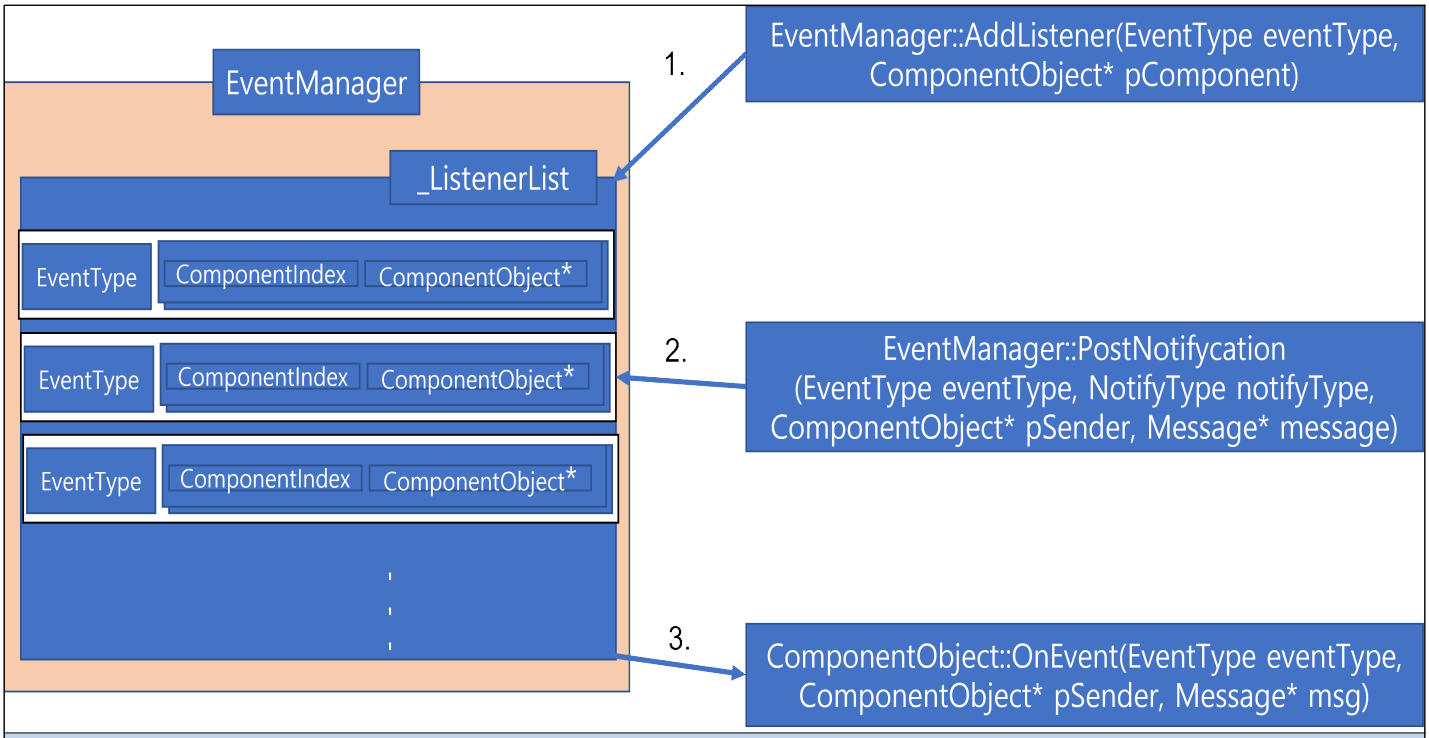
2. 이벤트 기반 프로그래밍

2.1. 이벤트 기반 프로그래밍이란?

이벤트의 발생에 의해 프로그램 흐름이 결정, 해당 포트폴리오에서는 이벤트가 통지되면 해당 이벤트를 구독 중인 객체의 이벤트 콜백 함수를 실행 함 사용자가 원하는 행위를 해당 이벤트에 실행 될 수 있게 하였다.

2.2. 이벤트 기반 프로그래밍

1) Vampire Survivor 이벤트 프로그래밍 도표



이벤트 기반 프로그래밍 도표

EventManager는 이벤트를 관리하는 매니저 객체로 각 이벤트의 발생, 이벤트의 관리, 이벤트의 실행을 담당한다.

순서대로 1.EventManager::AddListener 메소드를 통해 ComponentObject 상속받은 객체가 EventType에 해당하는 이벤트를 구독할 수 있다. 2. EventManager::PostNotifycation은 어떤 이벤트를 발생 해야 되는트리거 역할로 이벤트가 호출되어야 하는 시점에서 호출되어 진다. 마지막으로 이벤트가 트리거가 되면 EventManager의 _ListenerList 해당하는 이벤트를 구독하는 ComponentObject객체의 3.ComponentObject::OnEvent를 호출함으로써 해당 이벤트에 대한 ComponentObject객체의 추상메소드 OnEvent를 호출함으로써 이벤트를 처리할 수 있게 하였다.

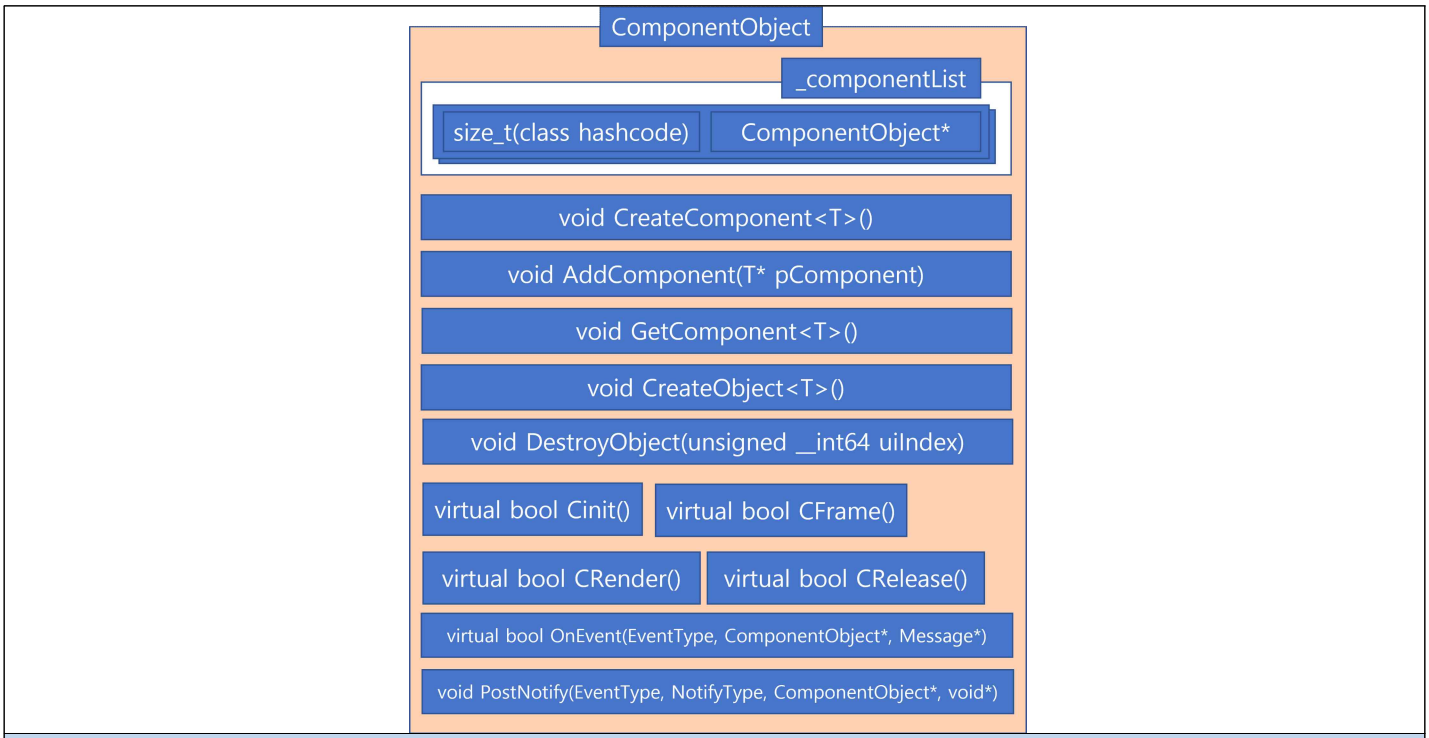
3. 컴포넌트 기반 프로그래밍

3.1. 컴포넌트 기반 프로그래밍이란?

재사용이 가능한 최소 단위의 클래스를 별도로 두어 각 시스템이 독립적인 업무 또는 기능을 수행하는 모듈로서 유지 보수가 가능하게 하는 프로그래밍이다. 하나 이상의 클래스로 이루어질 수 있고 인터페이스를 통해서 만 접근이 가능하다.

3.2. 컴포넌트 기반 프로그래밍

1) Vampire Survivor 컴포넌트 기반 프로그래밍 도표



컴포넌트 객체 도표

해당 프로젝트에서 게임을 구성하는 모듈 및 오브젝트들은 ComponentObject를 상속받는다. 각 Component를 상속받는 게임 객체는 _componentList를 가지고 있다. 해당 리스트는 각 Component의 클래스의 해시코드를 키로 Component의 포인터를 값으로 가지고 있다. ComponentObject 내부에서 Component의 생성 및 관리, 오브젝트의 생성 및 삭제를 할 수 있다. 또한 초기화, 프레임, 출력, 해제 등의 함수를 추상화함으로써 각 분야별로 관리하는 매니저들이 Component를 관리함으로써 단순화되고 명확하게 코드를 작성할 수 있다.
추가로 위에서 설명한 이벤트 기반 프로그래밍을 위해 OnEvent함수와 PostNotify함수를 정의하였다.

4. 이벤트와 컴포넌트의 결합

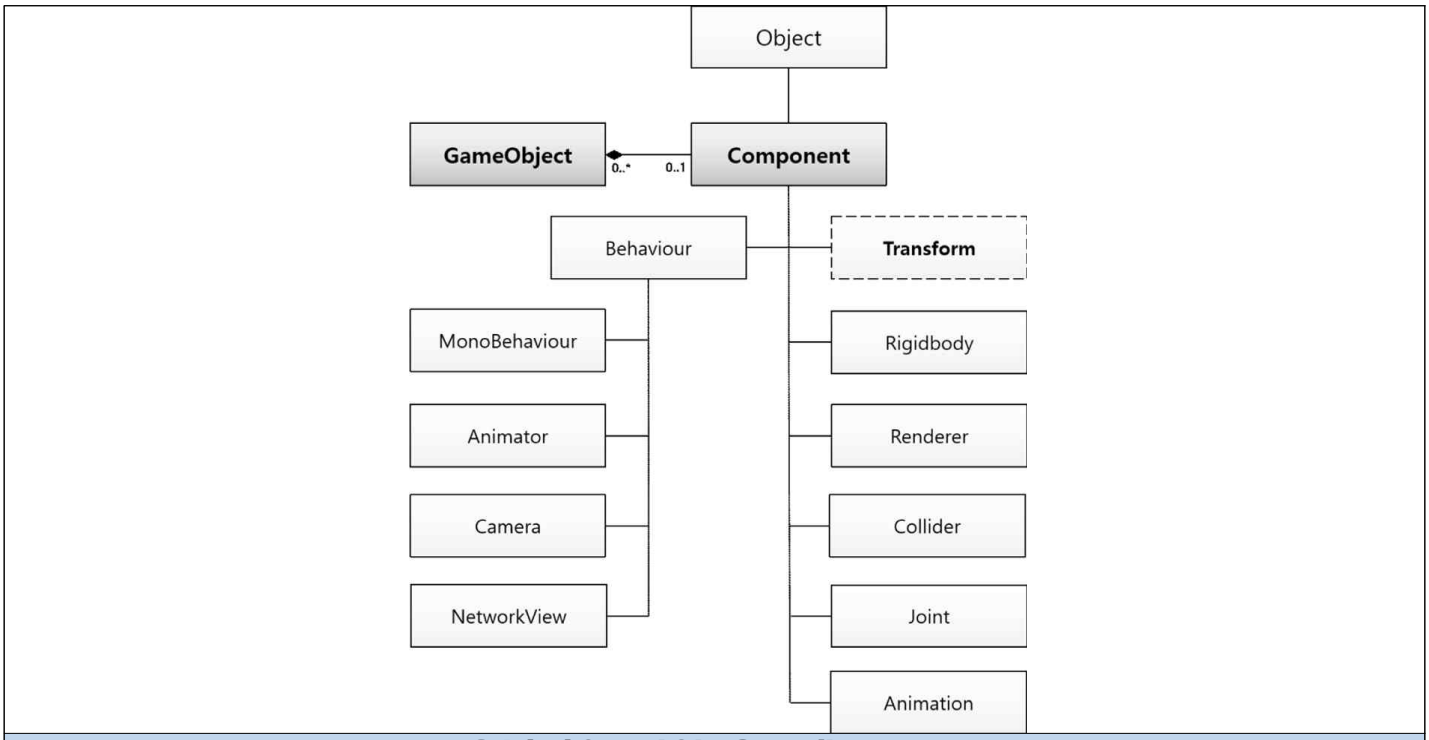
4.1. 이벤트와 컴포넌트의 결합이란?

컴포넌트들은 한 객체를 이루는 부분이기 때문에 상호 간의 조정, 통신이 필요하다. 이러한 문제를 해결하는 방식중 이벤트 즉 메시지를 전달하는 방식이다. 컴포넌트가 이벤트를 발생시키면 해당 이벤트를 구독하는 모든 컴포넌트를 순회하면서 이를 전파한다.

컨테이너 객체를 통해서 통신하기 때문에 컴포넌트끼리의 디커플링 상태를 유지한다. 또한 메시지만을 전달하면 되기 때문에, 특정 분야에 대한 정보를 컨테이너 객체에 노출되지 않고 컴포넌트끼리 주고받을 수 있다.

4.2. 이벤트와 컴포넌트

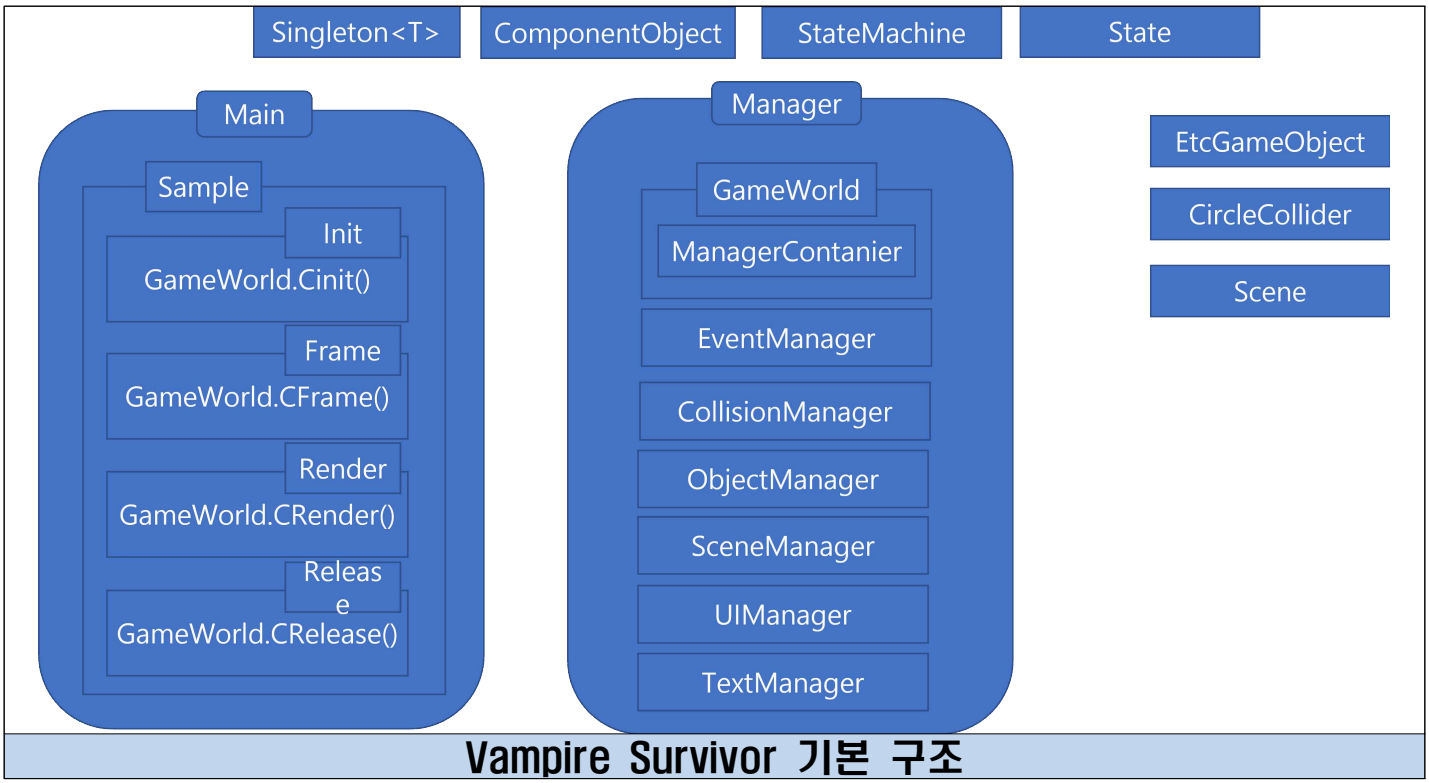
1) 이벤트와 컴포넌트의 결합의 예



유니티의 게임 오브젝트 구조

유니티 프레임워크의 핵심 클래스인 GameObject는 컴포넌트 방식으로 맞춰 설계되었다. 독립적인 컴포넌트를 만들어 서로 영향을 주지 않고 추가 삭제할 수 있다. 물체의 충돌 같은 처리 같은 경우에 컴포넌트 간의 상호처리가 필요하다. 이를 해결하기 위해 메시지처리 방식을 도입했다.

2) Vampire Survivor 이벤트와 컴포넌트의 결합



Vampire Survivor 기본 구조

클래스명	설명
Sample	- 게임의 디바이스, 엔진, 윈도우 기능을 상속받은 클래스 - 메인 문에서 생성 실행 - Sample내부 클래스에서 Init, Frame, Render, Release가 적소에 호출되며 각 함수는 Manager 들의 관리 클래스인 GameWorld의 함수를 호출
Manager	- Manager 영역의 클래스들은 유일 객체이며 Singleton<T> 클래스를 상속받음 - 해당 클래스는 ComponentObject의 추상화된 함수를 재정의하기 위해 해당 인터페이스 재정의 및 상속받음
Singleton<T>	- 싱글톤 객체가 상속받는 템플릿 클래스
ComponentObject	- 위에서 설명한 컴포넌트 객체
StateMachine	- 유한상태머신을 구현하기 위한 객체
State	- 상태를 나타내기 위한 객체

GameWorld	- 게임에서 사용되는 Manager 클래스를 관리하고 Manager들의 Init, Frame, Render, Release를 호출
EventManager	- 이벤트를 관리하는 Manager 클래스, 이벤트의 호출, 객체의 관리, 메소드의 실행을 담당
CollisionManager	- 객체의 충돌을 담당 - 프레임레이트 객체를 통해 최적화를 진행 - 화면 밖을 벗어난 객체(충돌 컴포넌트)는 충돌처리를 하지 않음
ObjectManager	- 객체들의 할당과 해제 관리 - 각 오브젝트의 프레임, 렌더링을 담당함 - 화면 밖을 벗어난 객체는 렌더링에서 제외함
SceneManager	- Scene의 초기화, 실행, 해제를 담당 - Scene에 소속된 오브젝트를 할당 및 해제 - 상태머신으로 구성
UIManager	- 게임의 UI 객체를 담당
TextManager	- 게임의 텍스트를 담당
EtcGameObject	- 게임의 오브젝트는 ComponentObject를 상속 - 게임 오브젝트는 종류에 따라 ObjectManager, UIManager의 관리를 받음
CircleCollider	- ComponentObject를 상속 - 충돌 객체는 생성 시 CollisionManager의 관리를 받음 - 오브젝트 내부에서 생성이 가능 - 충돌 감지 시 OnEvent메소드를 통해 통지 - 오브젝트 삭제 시 자동 삭제
Scene	- State를 상속받음 - 해당 Scene의 object와 text, ui를 컨테이너에 저장 - Scene 내부에서 오브젝트의 생성, 삭제 시 절차적으로 이벤트 전파 - Scene 전환 시 오브젝트의 절차적인 정리

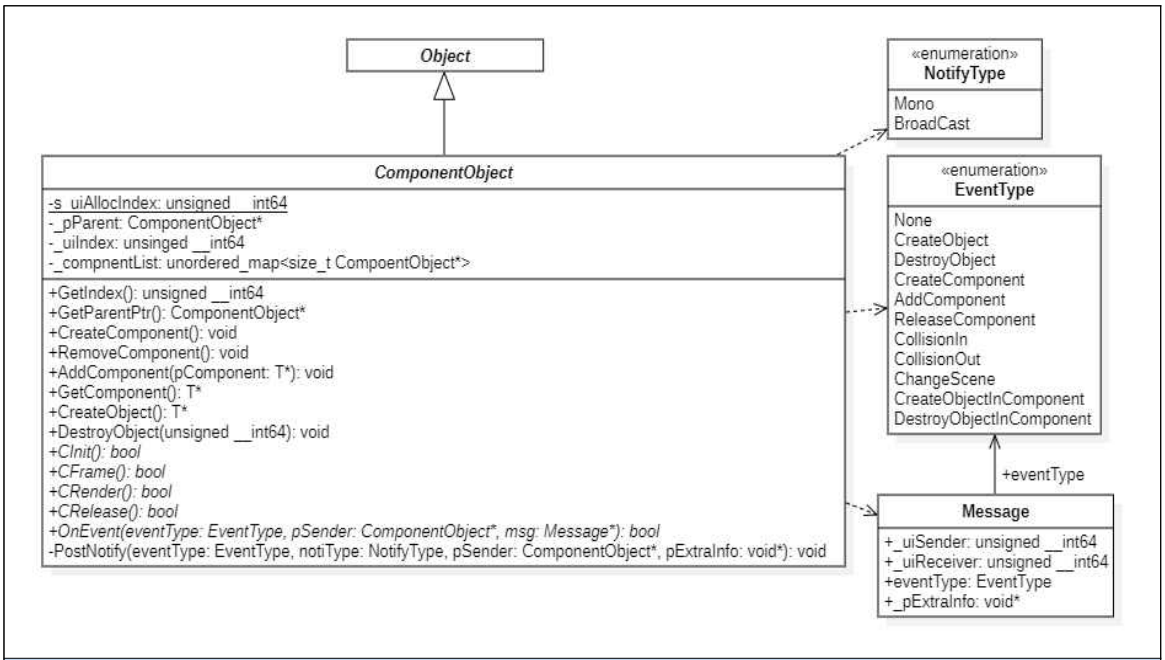
Ⅱ. 기술요약 및 UML

1. 클래스 다이어그램

컴포넌트 기반 프로그래밍과 이벤트 기반 프로그래밍이 본 프로젝트에 어떻게 구성되었는지 설명 클래스다이어그램을 통해 자세히 설명하고자 한다.

1.1. ComponentObject

1) ComponentObject ClassDiagram



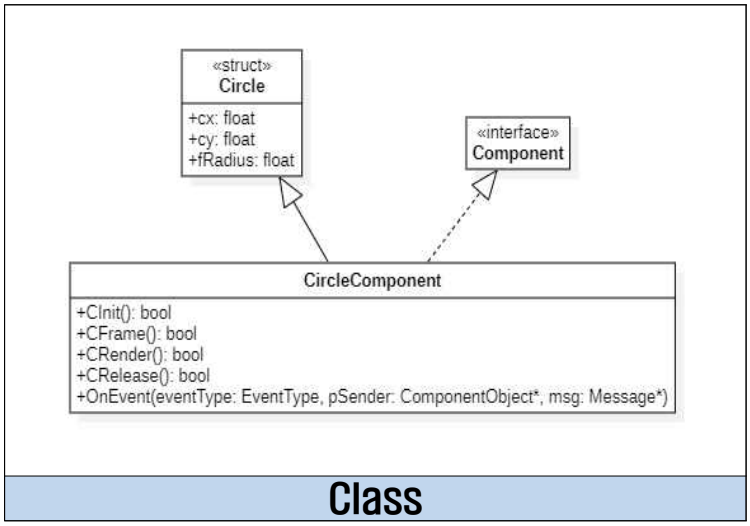
Class

2) ComponentObject 분석

컴포넌트 기반 프로그래밍, 이벤트 기반 프로그래밍에서 중요한 부분을 차지하는 클래스이다. 프로젝트의 객체는 해당 클래스를 상속받아야 한다. 이벤트 타입으로는 Mono(단독), broadcast(전체)가 있으며 Message에는 보내는 이의 고유 아이디, 받는 이의 고유 아이디, 이벤트 타입, 기타 정보 등을 넣는다. 현재 이 프로젝트에서 사용하는 이벤트 타입으로, 객체생성, 객체 삭제, 컴포넌트 생성, 컴포넌트 추가, 컴포넌트 해제, 충돌, 충돌 해제, Scene 전환, 객체 안에서의 객체생성, 객체 안에서의 객체 삭제가 있다.

1.2. CircleComponent

1) CircleComponent ClassDiagram

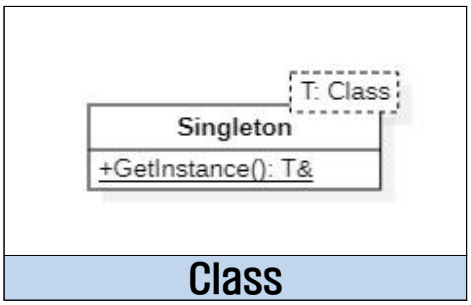


2) CircleComponent 분석

구 충돌을 담당하는 컴포넌트이다. Circle 구조체와 Component의 인터페이스는 상속받는다. CircleComponent가 생성 시 이벤트가 발생하며 해당 Component를 ColiderManager가 충돌 처리에 상용된다.

1.3. Singleton

1) Singleton Diagram

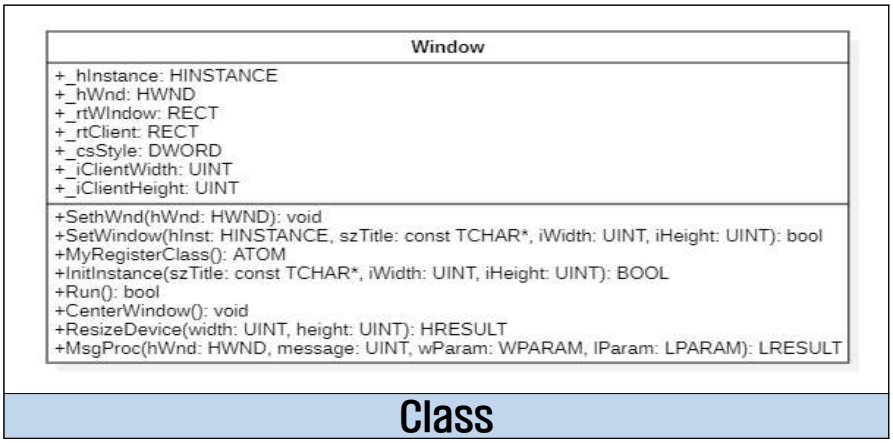


2) Singleton 분석

유일 객체를 위한 클래스 매니저 관련 클래스가 해당 클래스를 상속받는다.

1.4. Window

1) Window Diagram

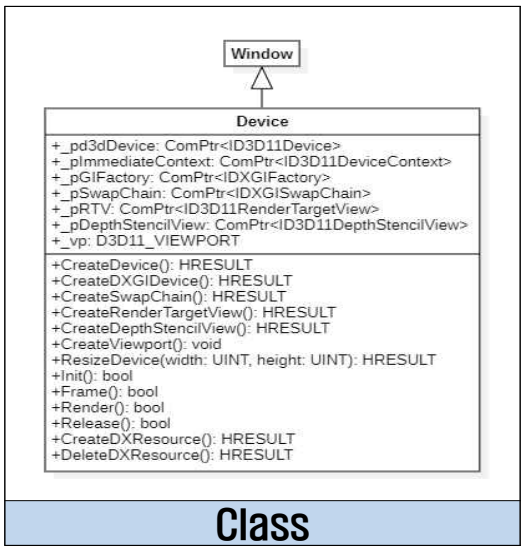


2) Window 분석

윈도우를 생성하기 위한 클래스 윈도우 창의 인스턴스와 핸들, 크기 등을 세팅 및 저장한다. 윈도우 자체의 메시지 루프를 처리 또한 포함하고 있다. 게임의 최상위 클래스로 게임 시작 시 가장 기본이 되는 클래스다.

1.5. Device

1) Device Diagram

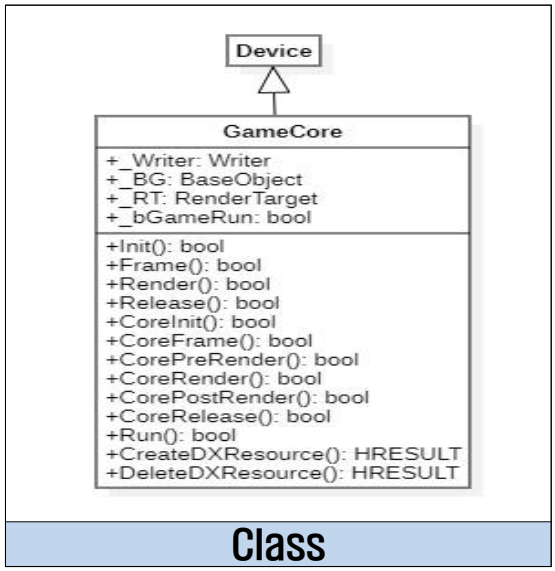


2) Device 분석

Direct3D를 사용하기 위해서 Device(리소스 생성)와 DeviceContext(리소스를 사용 처리 제어)를 획득 및 관리, 윈도우를 상속받아 윈도우의 저장된 핸들과 세팅 값 등을 사용한다.

1.6. GameCore

1) GameCore Diagram

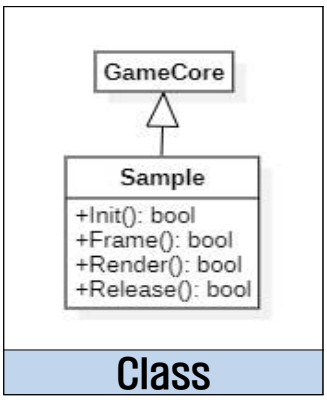


2) GameCore 분석

게임에 필요한 기능을 세팅 및 관리하기 위한 클래스. Device를 상속받아 Window와 Device의 세팅, 루프를 관리하며, 게임에 필요한 Sprite, Texture, Shader, Sound, Timer, Input의 세팅 및 관리한다.

1.7. Sample

1) Sample Diagram

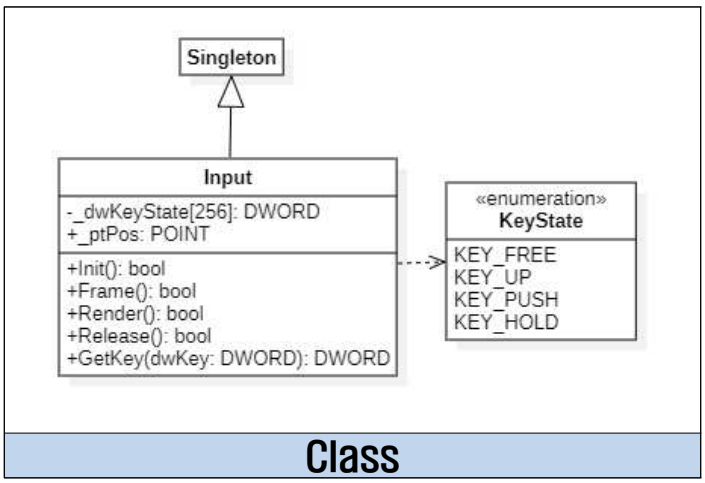


2) Sample 분석

실제 Vampire Survivor 프로젝트의 시작 클래스, GameWorld라는 매니저 클래스의 초기화, 업데이트, 출력, 해제를 담당하고 있다. main 함수에서 Sample 클래스의 생성 및 시작을 호출한다.

1.8. Input

1) Input Diagram

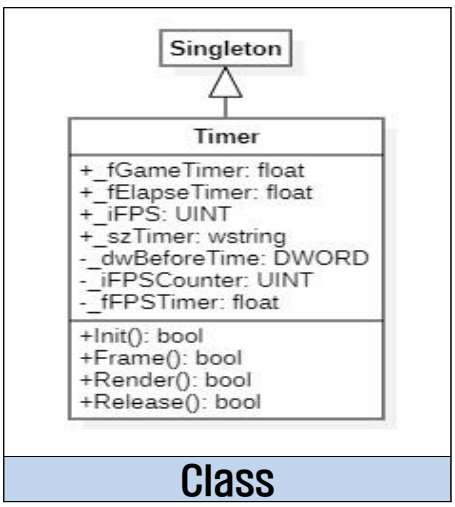


2) Input 분석

게임 코어의 입력을 담당하는 클래스, 입력값 마우스의 클릭 위치, 키보드의 눌림, 누르는 중, 띄어짐을 프레임별로 감지하여 관리한다.

1.9. Timer

1) Timer Diagram

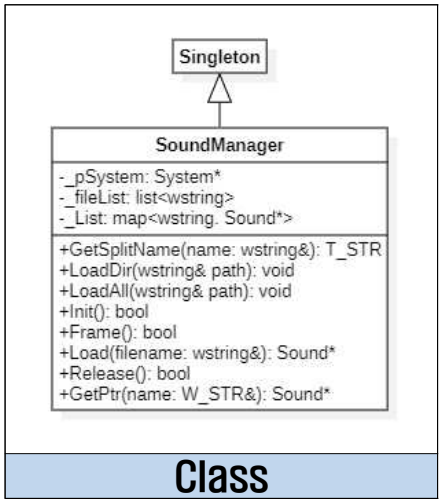


2) Timer 분석

게임 제작 시 필요한 프레임 간의 시간을 측정하기 위한 전역 클래스 게임의 프레임을 측정하고 프레임 간의 시간을 저장해 놓는다.

1.10. SoundManager

1) SoundManager Diagram

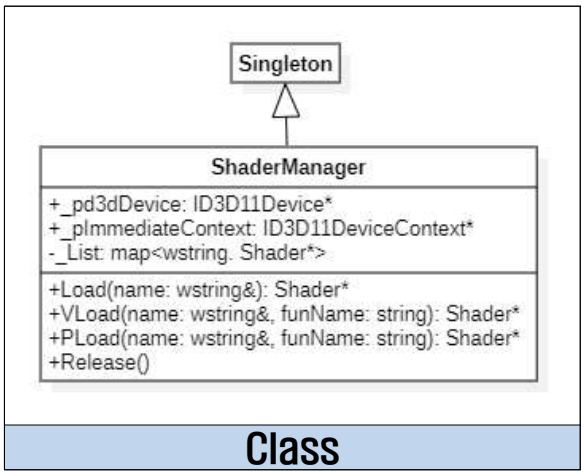


2) SoundManager 분석

게임 코어에서 사운드를 담당하는 싱글톤 클래스이다. FMOD를 이용하여 사운드 파일의 목록과 사운드의 실행, 정지를 관리한다.

1.11. ShaderManager

1) ShaderManager Diagram

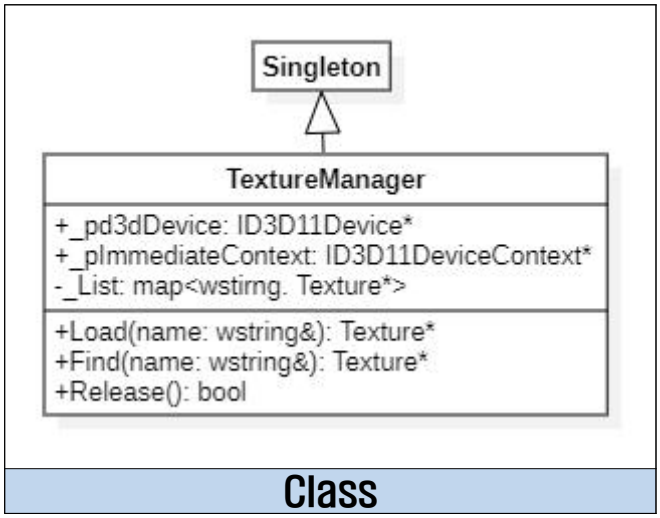


2) ShaderManager 분석

쉐이더파일을 캐싱하고 관리하는 싱글톤 클래스. 리소스를 출력할 때 상황에 맞는 쉐이더를 로드하여 사용하게 해준다.

1.12. TextureManager

1) TextureManager Diagram

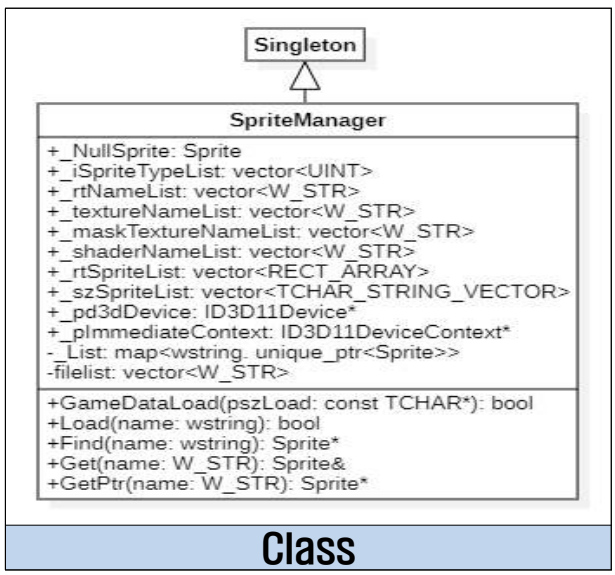


2) TextureManager 분석

게임의 텍스쳐를 캐싱 및 관리하는 싱글톤클래스이다. 해당 클래스를 이용하여 3D 객체는 해당 매니저에게 접근하여 텍스처를 로드한다.

1.13. SpriteManager

1) SpriteManager Diagram

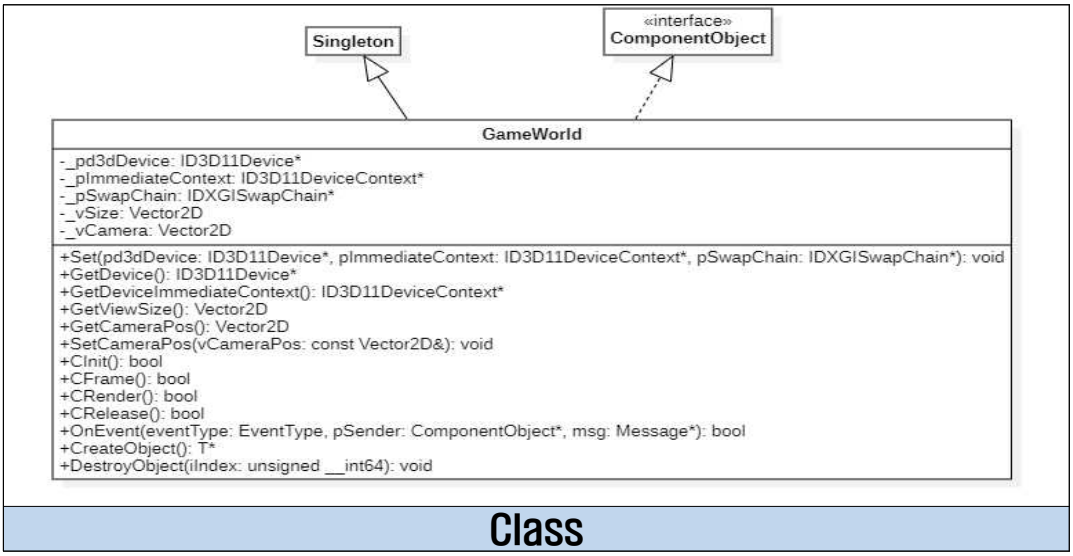


2) SpriteManager 분석

스프라이트 애니메이션을 담당하는 싱글톤 클래스, 텍스쳐의 위치 및 크기를 저장 관리한다.

1.14. GameWorld

1) GameWorld Diagram

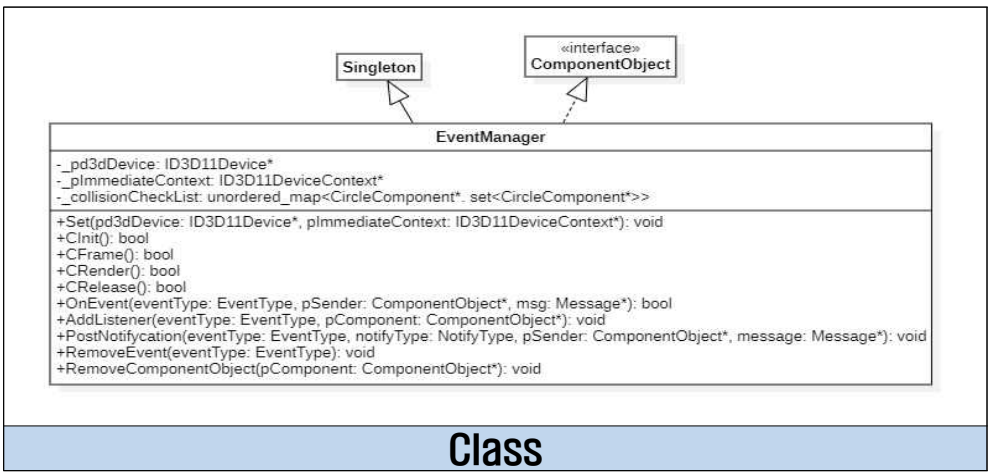


2) GameWorld 분석

게임프로젝트의 매니저 클래스를 관리하는 싱글톤클래스. 매니저 클래스의 생성과 초기화, 로직, 해제를 담당하며 매니저 클래스의 init, Render, Release, Frame을 호출하여 각 매니저가 기능을 수행할 수 있게 해준다.

1.15. EventManager

1) EventManager Diagram

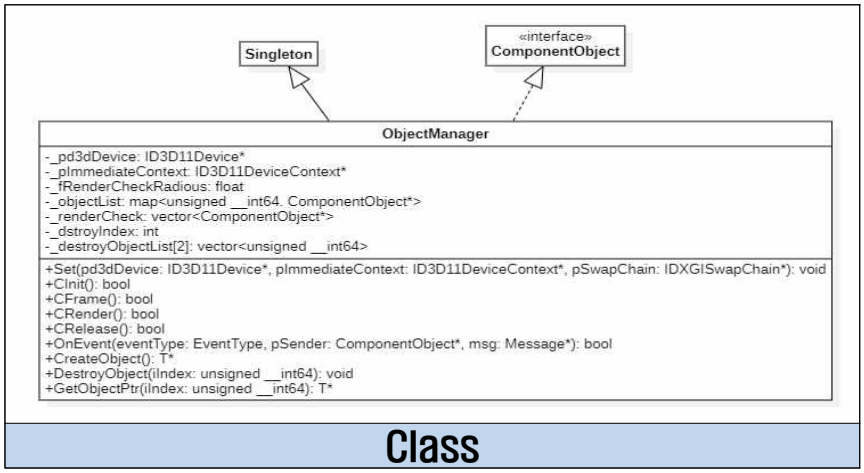


2) EventManager 분석

이벤트를 관리하는 싱글톤 클래스. 이벤트의 등록, 발생, 실행을 담당한다.

1.16. ObjectManager

1) ObjectManager Diagram

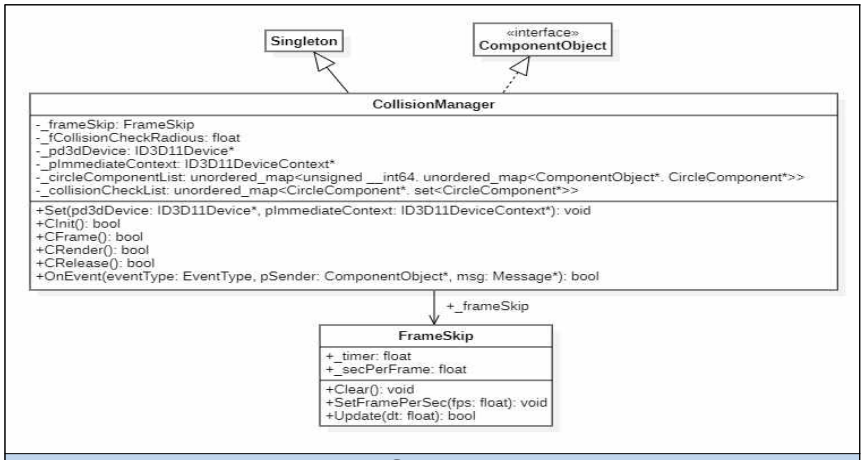


2) ObjectManager 분석

오브젝트의 생성과 해제, 렌더에 관여한다. Scene에서 생성 요청된 오브젝트를 생성하고 Scene이 전환 시 이벤트를 받아 오브젝트를 해제한다. 또한 화면의 크기를 기준으로 반지름으로 잡아 원 영역을 만들어 원 영역의 객체만 렌더링하도록 객체를 관리하고 있다.

1.17. CollisionManager

1) CollisionManager Diagram



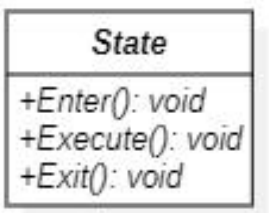
Class

2) CollisionManager 분석

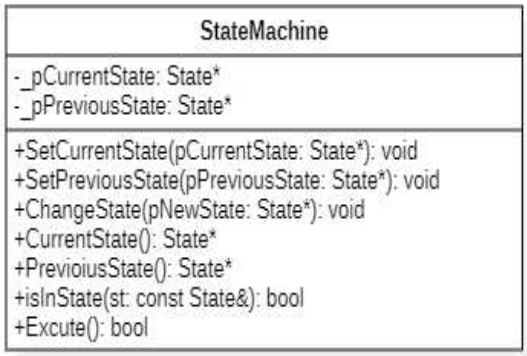
충돌 컴포넌트 생성, 해제 시 이벤트를 받아 충돌 컴포넌트의 충돌 처리를 담당한다. 충돌 처리는 게임 로직 중 가장 부하가 큰 작업으로 최적화 작업이 들어갔다. 출력 화면을 기준으로 화면을 벗어난 충돌 컴포넌트는 건너뛴다. 또한 매 프레임 충돌 처리는 많은 작업량을 요구하므로 Frame Skip 클래스를 통해 지정된 FPS만 충돌 처리를 하도록 설정하였다. 예로 10으로 지정한다면 1초당 10번의 충돌 처리만 한다는 뜻이다.

1.18. SceneManager

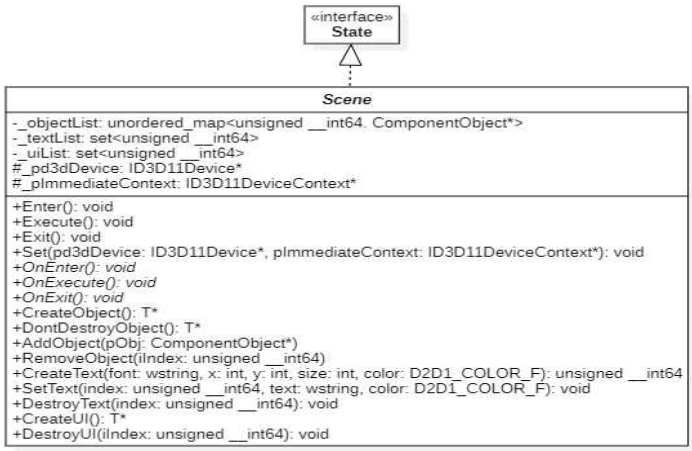
1) SceneManager Diagram



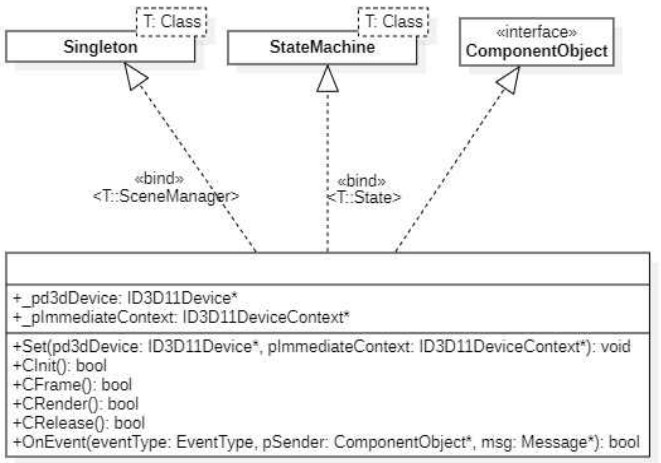
State



StateMachine



Scene



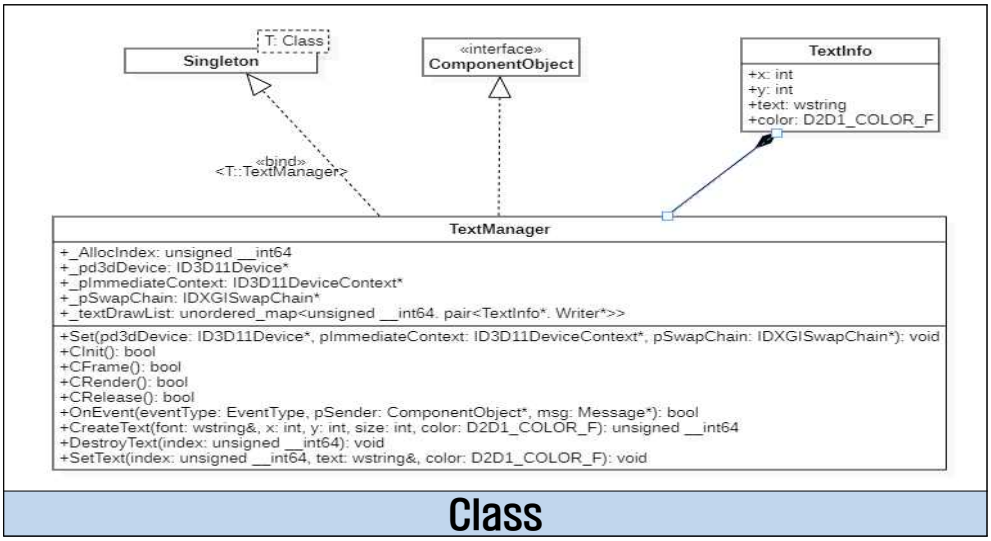
SceneManager

2) SceneManager 분석

SceneManager는 StateMachine을 이용하여 State를 상속받은 Scene을 이용하여 Scene의 입장과 Scene의 퇴장을 구현하였다. Scene에서는 Scene에 사용되는 UI, Texture, Colider를 관리하며 Scene의 퇴장 시 관리하는 객체들의 정보를 Event Manager에 통지하여 각 객체를 전담하는 Manager에서 해당 객체를 정리할 수 있도록 브릿지 역할을 하고 있다.

1.19. TextManager

1) TextManager Diagram

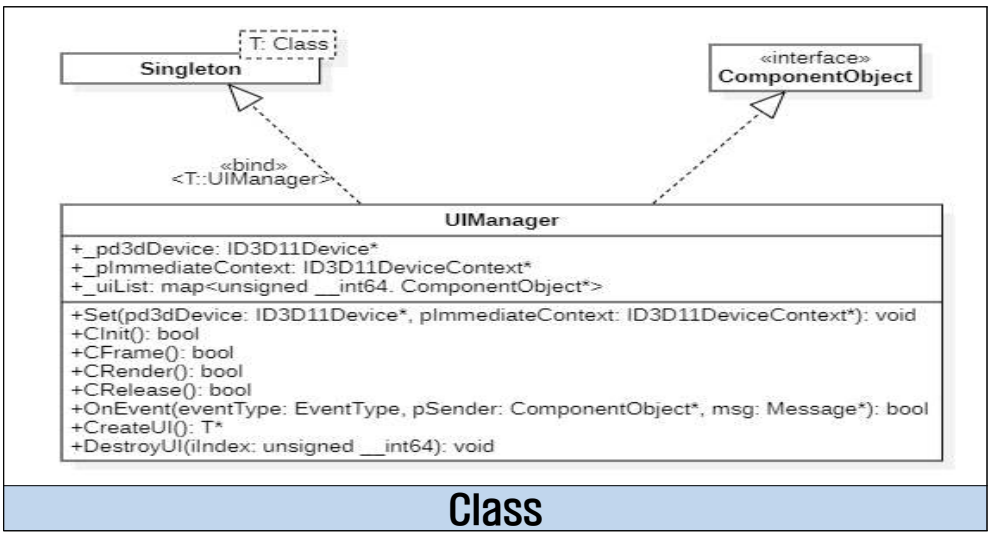


2) TextManager 분석

일반 객체와는 다르게 Text는 TextManager에서 별도로 관리한다. 화면좌표에 바로 렌더를 하고 모든 오브젝트와 UI가 그려진 후 맨 마지막에 텍스트를 렌더링한다.

1.20. UIManager

1) UIManager Diagram



2) UIManager 분석

텍스트와 마찬가지로 화면좌표를 사용하며 일반 오브젝트와 다르게 별도로 관리한다. 모든 오브젝트가 화면상에 그려진 후 바로 화면좌표에 관리하는 UI 객체를 렌더링한다.

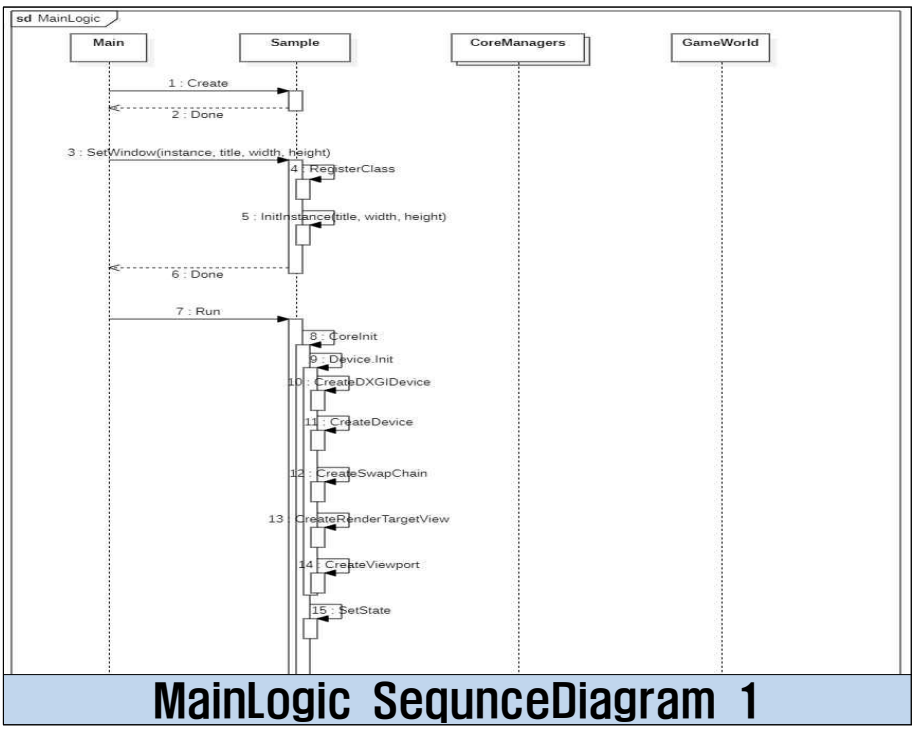
2. 시퀀스다이어그램

현 프로젝트에서 객체의 생성과 Scene의 전환, 충돌 처리, 렌더링 처리를 컴포넌트, 이벤트화하여 어떻게 처리했는지 시퀀스다이어그램을 통해 파악하고자 한다.

2.1. MainLogic

현 프로젝트의 시작과 끝을 시간 흐름으로 분석하고자 한다. 프로젝트 구동에 필요한 사전 작업과 구성요소를 파악하고자 한다.

1) MainLogic SequnceDiagram 1



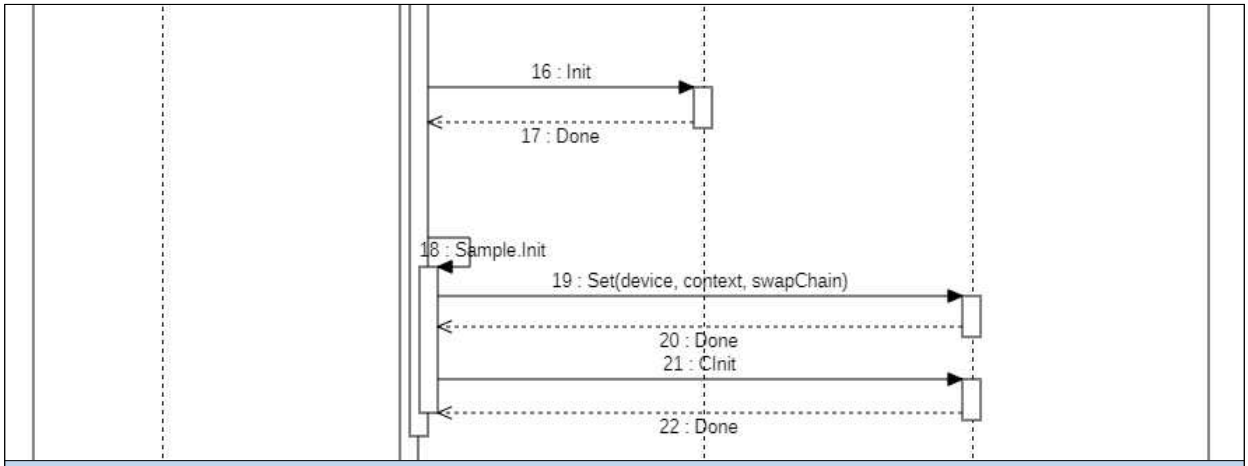
MainLogic SequnceDiagram 1

2) MainLogic 분석

Sample 클래스는 Window, Device, Core를 상속받고 있다. 순서대로 각 요소를 초기화하고 있다.

1. Window의 SetWindow에서는 윈도우 창을 등록하고 세팅된 윈도우의 핸들과 크기를 저장한다.
2. GameCore::CoreInit에서는 Device의 초기화와 게임 제작에 필요한 매니저들의 초기화를 진행한다. Direct3D를 사용하기 위한 디바이스(리소스 생성)와 디바이스 컨텍스트(렌더링)를 얻어온다.
3. 윈도우 생성 시 얻어온 윈도우의 크기를 이용하여 SwapChain을 생성한다.
4. 렌더링 될 백 버퍼를 생성한다.
5. 스크린의 어떤 위치에 작업을 할지 뷰포트를 지정한다.
6. 기타 렌더링의 기본속성값을 지정한다.

1) MainLogic SequnceDiagram 2

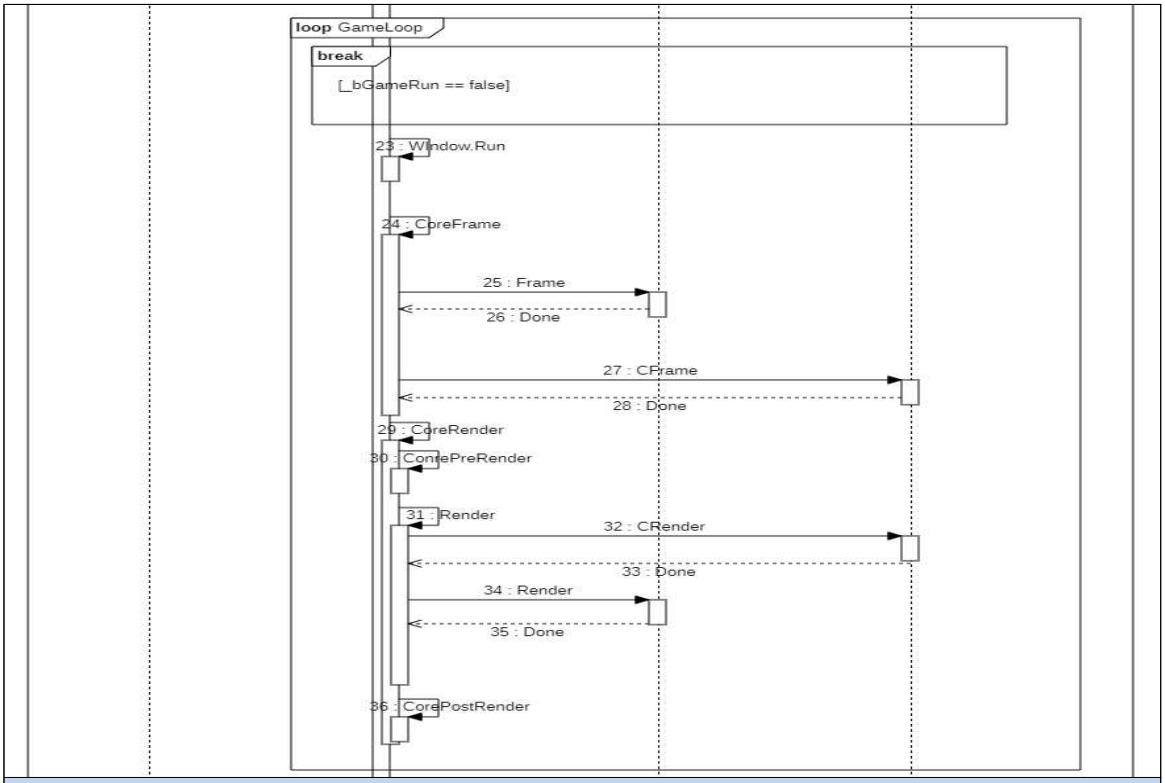


MainLogic SequnceDiagram 2

2) MainLogic 분석

1. Core에서 사용되는 매니저를 초기화한다.
2. GameWorld(게임 제작 클래스)의 초기화를 진행한다.
3. GameWorld의 초기화에서는 사용자가 만든 게임 매니저를 초기화하며 각 매니저가 이벤트구독을 진행한다.

1) MainLogic SequnceDiagram 3

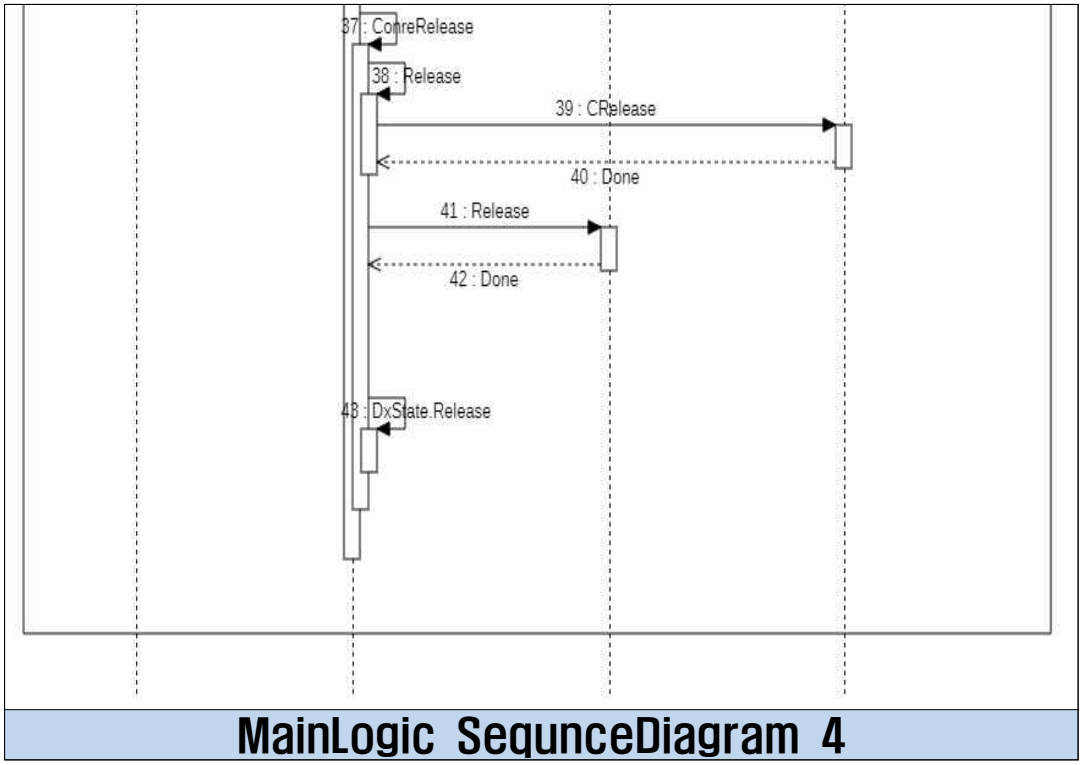


MainLogic SequnceDiagram 3

2) MainLogic 분석

1. Window의 기본 메시지 루프를 처리한다.
2. GameCore의 매니저 클래스의 Frame과 GameWorld의 사용자 매니저 클래스의 Frame을 실행한다.
3. CoreRender를 실행한다. 객체를 출력하기 전 랜 더 타깃 뷰를 초기화하고 객체를 그리기 그래픽 파이프라인을 세팅한다. 이후 Render에서 GameWorld의 매니저의 Render를 호출한다. ObjectManager, UIManager, TextManager에서 객체를 랜 더 타깃 뷰에 출력한다.
4. CorePostRender를 실행한다. 백 버퍼의 내용을 버퍼에 출력한다.

1) MainLogic SequnceDiagram 4



MainLogic SequnceDiagram 4

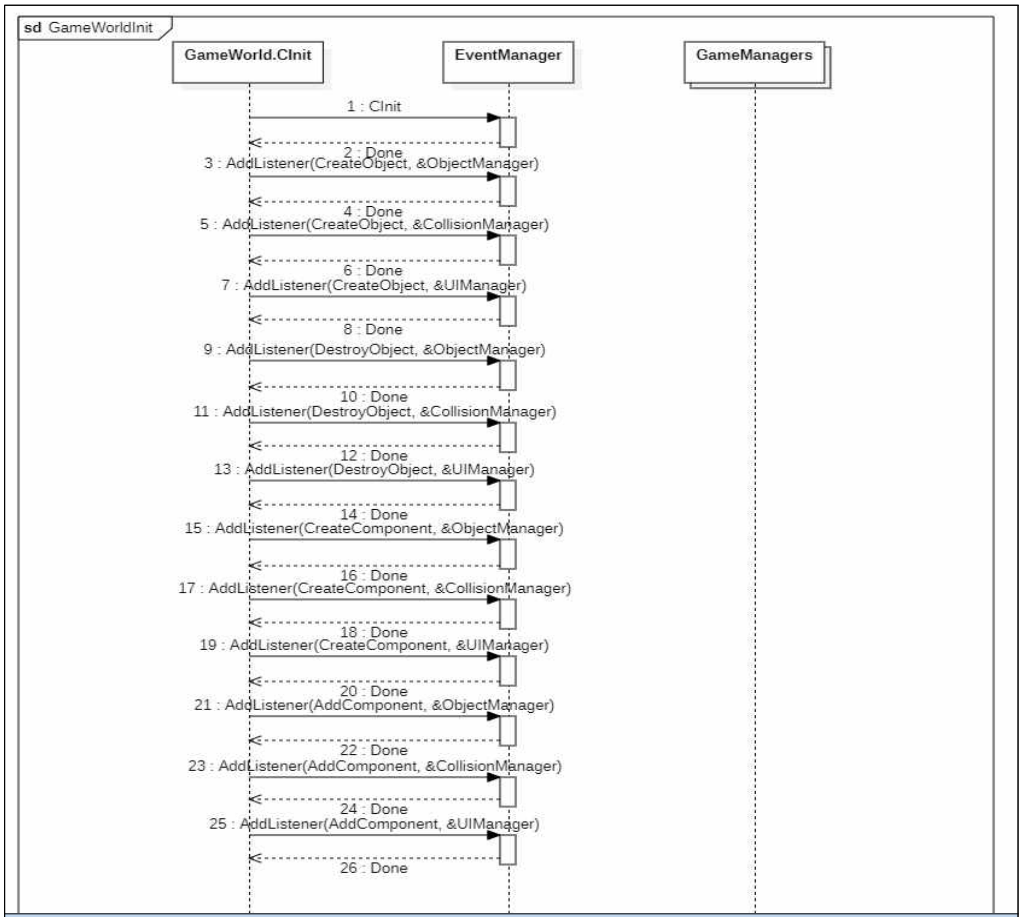
2) MainLogic 분석

1. Window 메시지 루프에서 종료메시지가 감지되면 루프를 빠져나와 CoreRelease를 호출한다.
2. 오버라이딩된 Release를 호출하여 GameWorld의 CRelease를 호출한다. CRelease에서는 GameWorld에서 사용한 매니저들을 해제한다. 서로 연관된 기능이 있을 수 있으니, 순서에 유의하여 해제한다.
3. Core에서 사용하는 매니저를 해제한다.
4. Direct3D 관련 리소스를 해제한다.

2.2. GameWorldInit

해당 프로젝트에서 GameWorld의 초기화는 이벤트 기반 프로그래밍을 어떻게 활용되는지 파악에 많은 도움을 주는 부분이다. GameWorld 초기화 과정을 분석하면서 해당 프로젝트에서 이벤트 기반 프로그래밍이 어떻게 활용되었는지 확인할 수 있다.

1) GameWorldInit SequnceDiagram 1

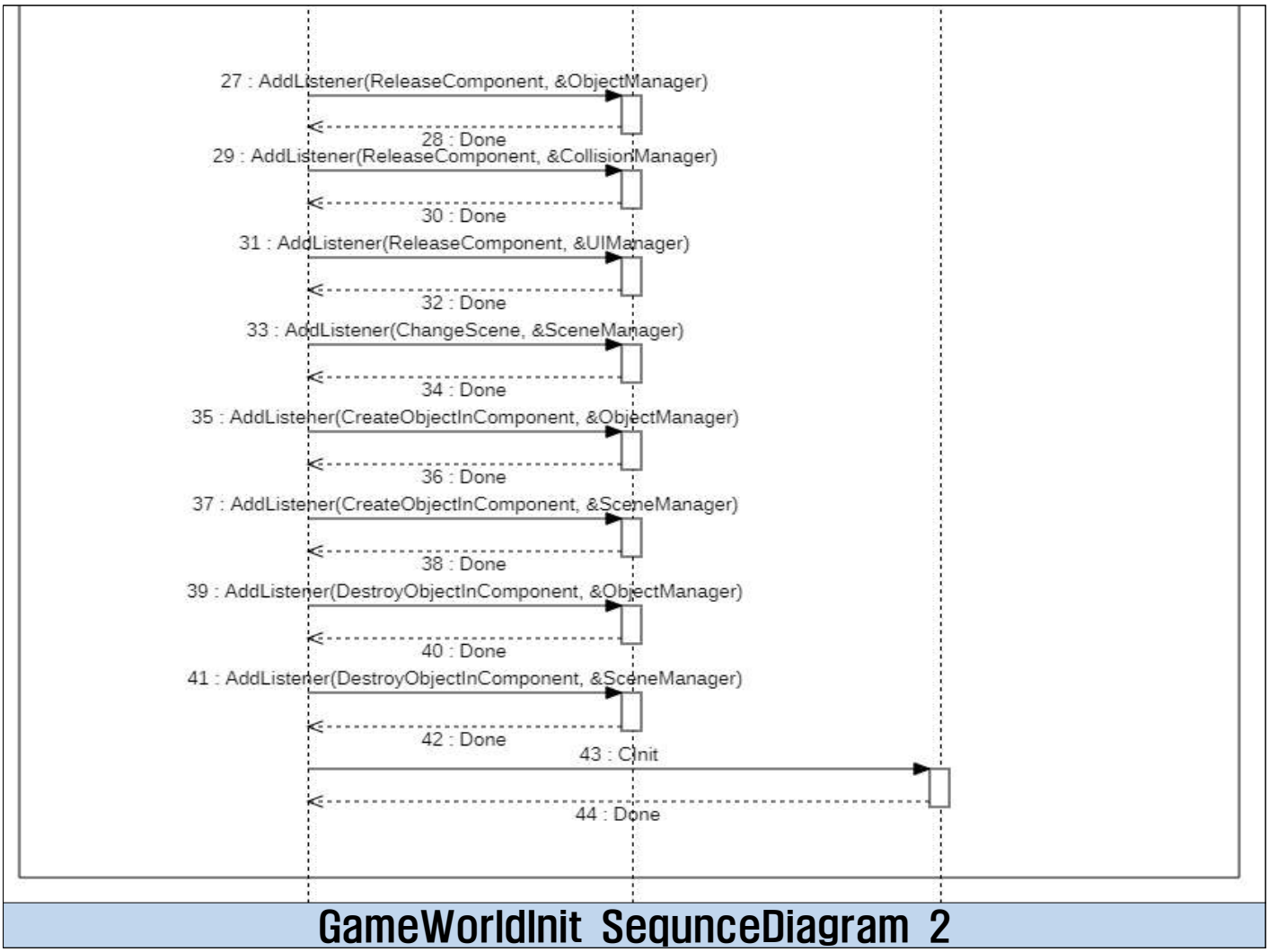


GameWorldInit SequnceDiagram 1

2) GameWorldInit 분석

1. CreateObject 메시지는 ObjectManager에서 CreateObjectInComponent메시지를 수신 시 객체를 등록이 완료된 후 각 매니저에게 통지하는 메시지다.
2. DestroyObject 메시지는 ObjectManager에서 DestroyObjectInComponent메시지를 수신 시 객체를 삭제될 객체 버퍼에 삽입 후 각 매니저에게 통지하는 메시지다.
3. CreateComponent메시지는 Component 생성 시 통지된다. 통지를 받은 매니저는 객체를 검사하여 전담 Component를 파악하여 리스트화 후 관리한다.
4. AddComponent메시지는 CreateComponent와 거의 동일 하다. 호출하는 템플릿 메서드(생성방식)의 따라 분리되어 있다.

1) GameWorldInit SequnceDiagram 2



GameWorldInit SequnceDiagram 2

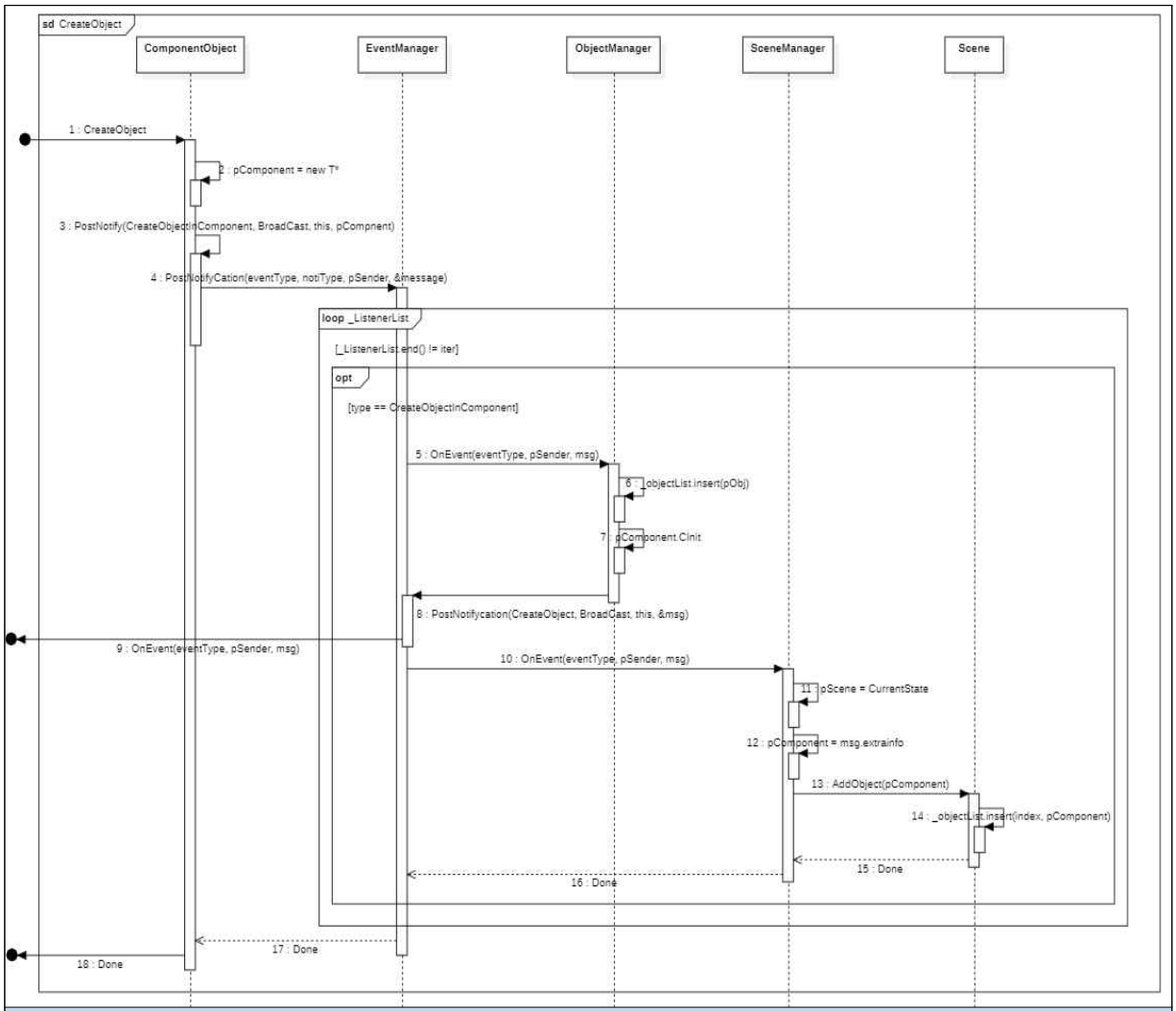
2) GameWorldInit 분석

1. 1. ReleaseComponent메시지는 Component 객체가 삭제될 시 리스트로 관리하는 자식 Component를 순회하여 각 매니저에게 통지한다. 담당 매니저가 수신 시 부모 Component의 고유 아이디로 검사하여 해당 Component가 존재 시 매니저의 리스트에서 삭제해 준다.
2. 2. ChangeScene 메시지는 Scene을 변경할 때 통지한다. 해당 메시지를 수신받은 SceneManager는 현재 Scene에서 관리하는 객체를 삭제하고 Scene을 변경한다.
3. 3. CreateObjectInComponent메시지는 Component 객체에서 Object를 생성했을 때 통지한다. 해당 Object를 관리해야 되는 매니저는 메시지를 수신하고 리스트화 하여 관리한다.
4. 4. DestroyObjectInComponent메시지는 Component 객체에서 Object를 삭제했을 때 통지한다. 해당 Object를 관리하는 매니저는 메시지를 수신하여 리스트에서 삭제한다.

2.3. CreateObject

현 프로젝트에서 객체의 생성을 특정 메소드(Scene, ComponentObject, ObjectManager)으로 한정하여 객체 생성에 대한 개발자의 실수를 줄이고 생성자, 소멸자를 사용하지 않도록 하여 개발자의 실수를 최대한 줄이도록 하였다.

1) CreateObject SequnceDiagram



CreateObject SequnceDiagram

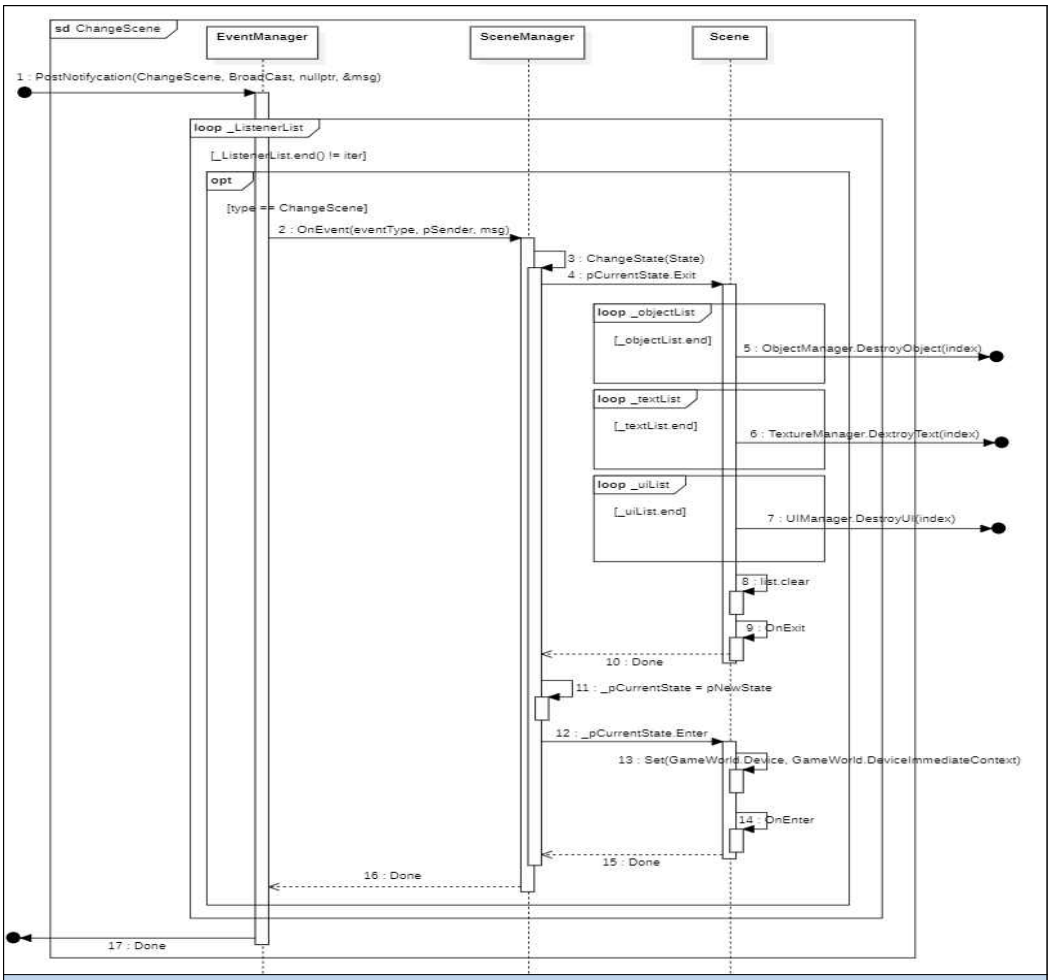
2) CreateObject 분석

1. Component의 CreateObject 템플릿 메소드를 호출 시 객체를 생성한다. 객체를 생성 후 EventManager에 CreateObjectInComponent메시지를 통지한다.
2. EventManager는 해당 메시지를 구독 중인 리스트를 순회하여 ObjectManager와 SceneManager에 해당 메시지를 통보한다.
3. ObjectManager, SceneManager는 메시지에 첨부된 Object를 관리 리스트에 추가한다.

2.4. ChangeScene

Scene이라는 개념을 도입하여 게임의 한 장면을 Scene 내부에서만 게임을 구현할 수 있게 강제하였다. Scene과 Scene 사이에는 객체를 공유하지 않는다. Scene을 넘어갈 시 Scene에서 사용한 객체들을 개발자가 아닌 Scene 자체에서 정리하도록 하여 개발의 편의성을 올렸다.

1) ChangeScene SequnceDiagram



ChangeScene SequnceDiagram

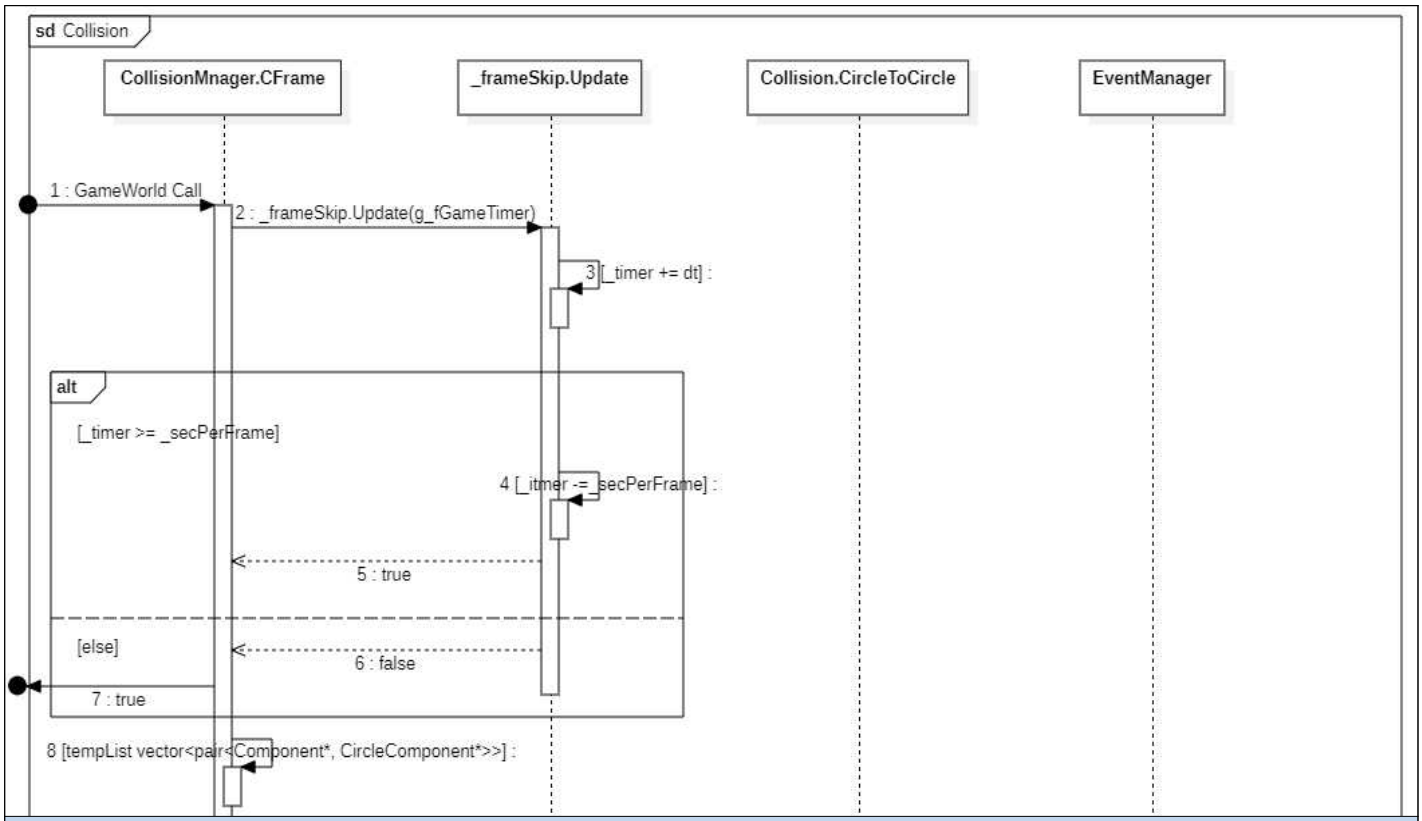
2) ChangeScene 분석

1. Scene에서 다음 Scene으로 넘어갈 시 EventManager에 ChangeScene 메시지를 통지한다.
2. EventManager는 ChangeScene 메시지를 구독하고 있는 매니저들을 순회하며 메시지를 호출한다.
3. SceneManager는 ChangeScene 메시지를 수신하면 현재 Scene의 Exit를 호출한다.
4. Exit 내부에서 현재 Scene에서 관리되는 객체들을 삭제한다. 이후 OnExit를 호출한다.
5. 현재 Scene을 다음 Scene으로 바꾼 후 Enter, OnEnter를 호출하며 Scene 전환을 마무리한다.

2.5. Collision

현 프로젝트에서 모든 게임 객체가 충돌 연산을 수행하기에 원활한 게임플레이를 위해서는 충돌 연산의 최적화가 필수였다. 현 프로젝트에서 충돌 처리와 최적화를 어떻게 하였는지 시퀀스다이어그램을 통해 설명하고자 한다.

1) Collision SequnceDiagram 1

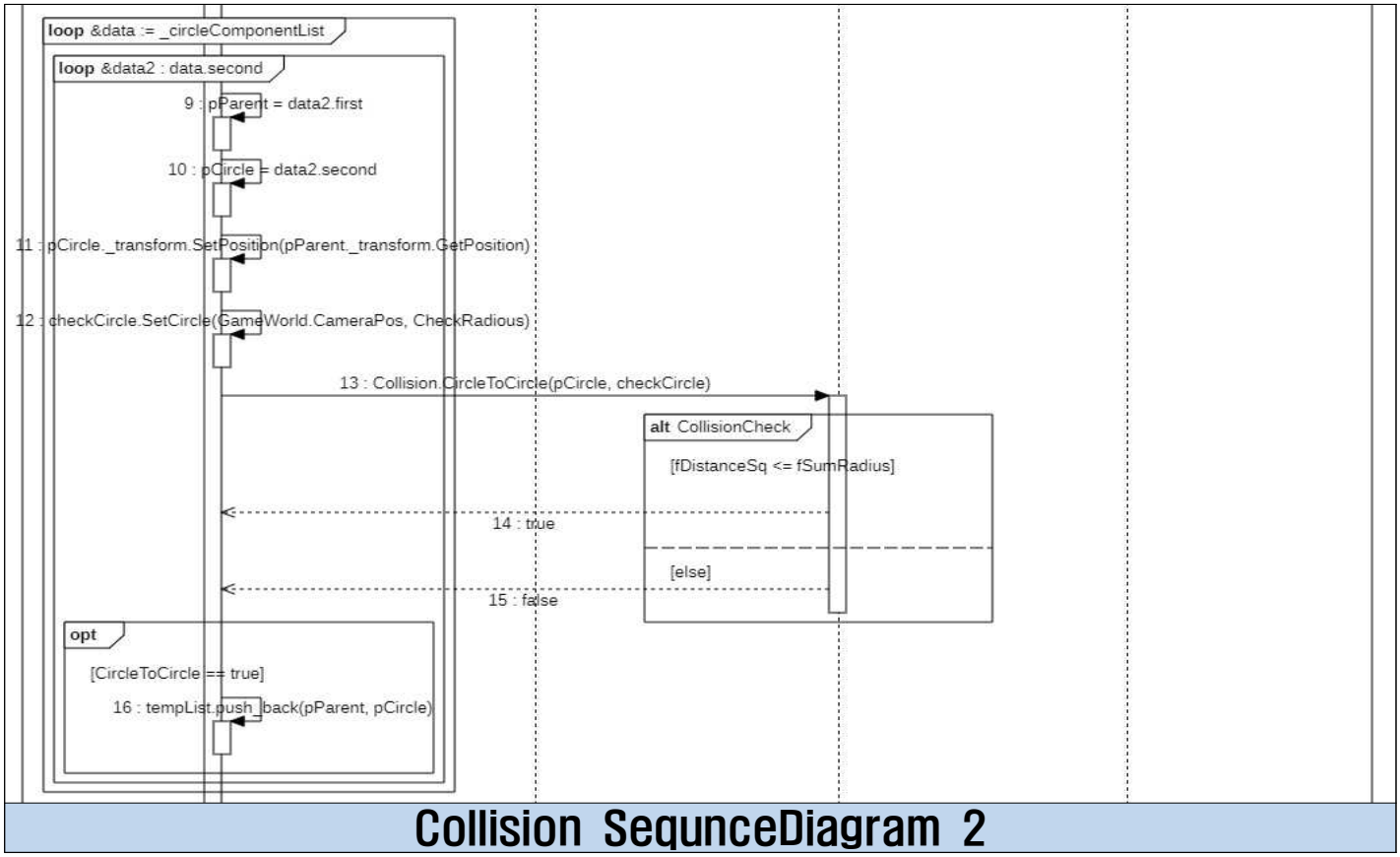


Collision SequnceDiagram 1

2) Collision 분석

1. CollisionManager의 CFrame은 현 프로젝트의 프레임마다 호출된다. CollisionManager의 FrameSkip 객체의 Update를 CFrame의 상단에서 같이 호출한다.
2. FrameSkip은 프레임 간의 간격 시간을 누적하여 해당 프레임에 업데이트를 진행할지 스킵할지를 결정한다. 누적된 시간이 프레임레이트에 도달했다면 업데이트를 진행한다. (1초당 5번의 실행을 해야 한다면 200밀리세컨드(1/5)에 도달할 때까지 델타 타임(프레임 간격 시간)을 누적하여 200밀리세컨드에 도달 시 업데이트를 진행 시킨다.)

1) Collision SequnceDiagram 2

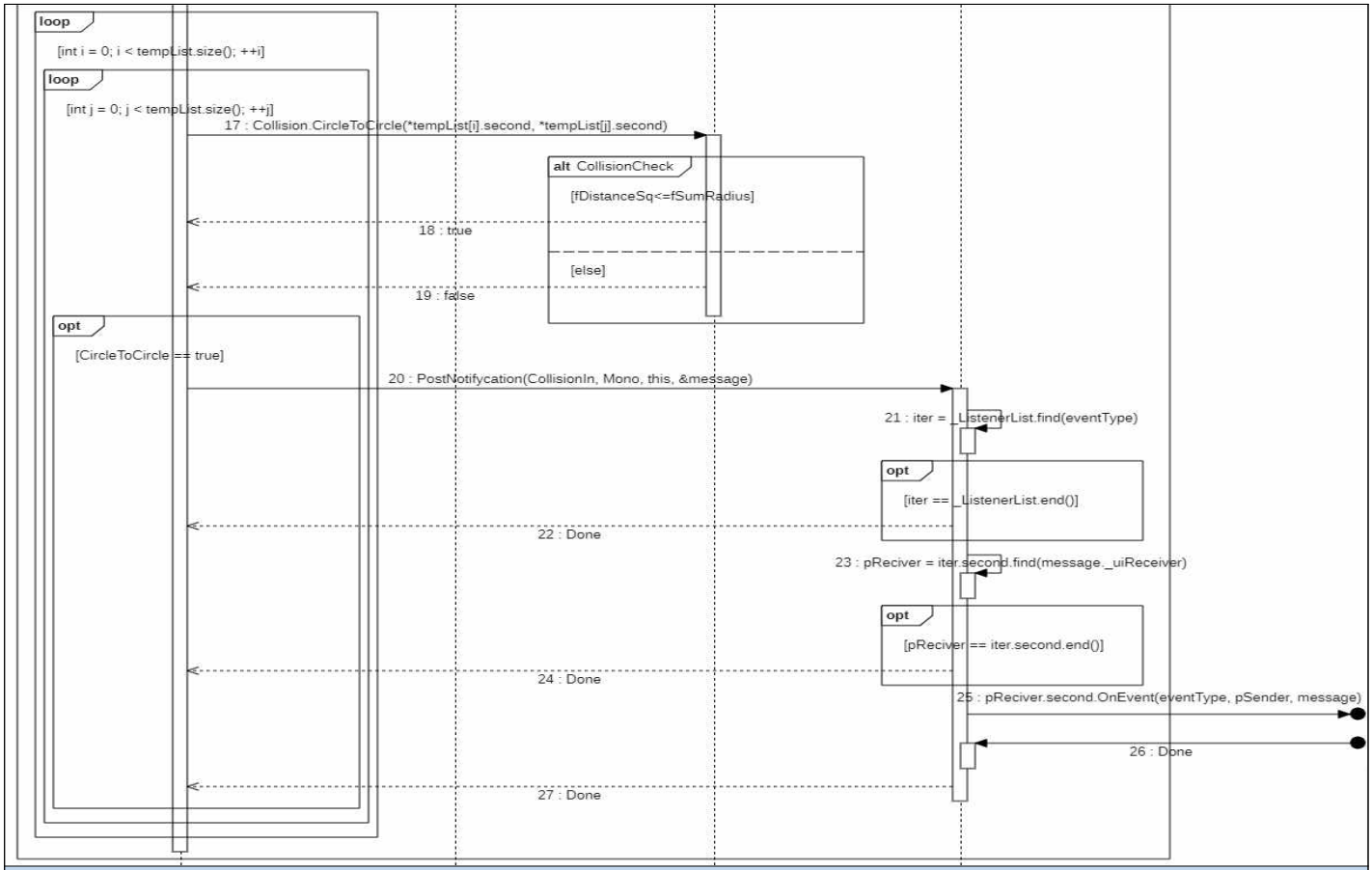


Collision SequnceDiagram 2

2) Collision 분석

1. CircleComponent의 리스트를 순회한다. 카메라 좌표(화면 중앙)를 기준으로 화면 영역의 대각선의 절반을 기준으로 checkCircle를 생성한다.
2. CircleToCircle 함수를 통해 CircleComponent의 원과 checkCircle의 반지름 합산의 제곱과 두 원의 원점 거리의 제곱을 비교하여 두 원이 충돌하는지를 판단한다.
3. 만약 두 원의 원점 거리의 제곱이 반지름 합산의 제곱보다 작거나 같으면 두 원은 충돌한 것이므로 화면 안에 존재하는 것이다. 충돌 처리를 하기 위해 CircleComponent를 임시리스트에 넣는다.

1) Collision SequnceDiagram 3



Collision SequnceDiagram 3

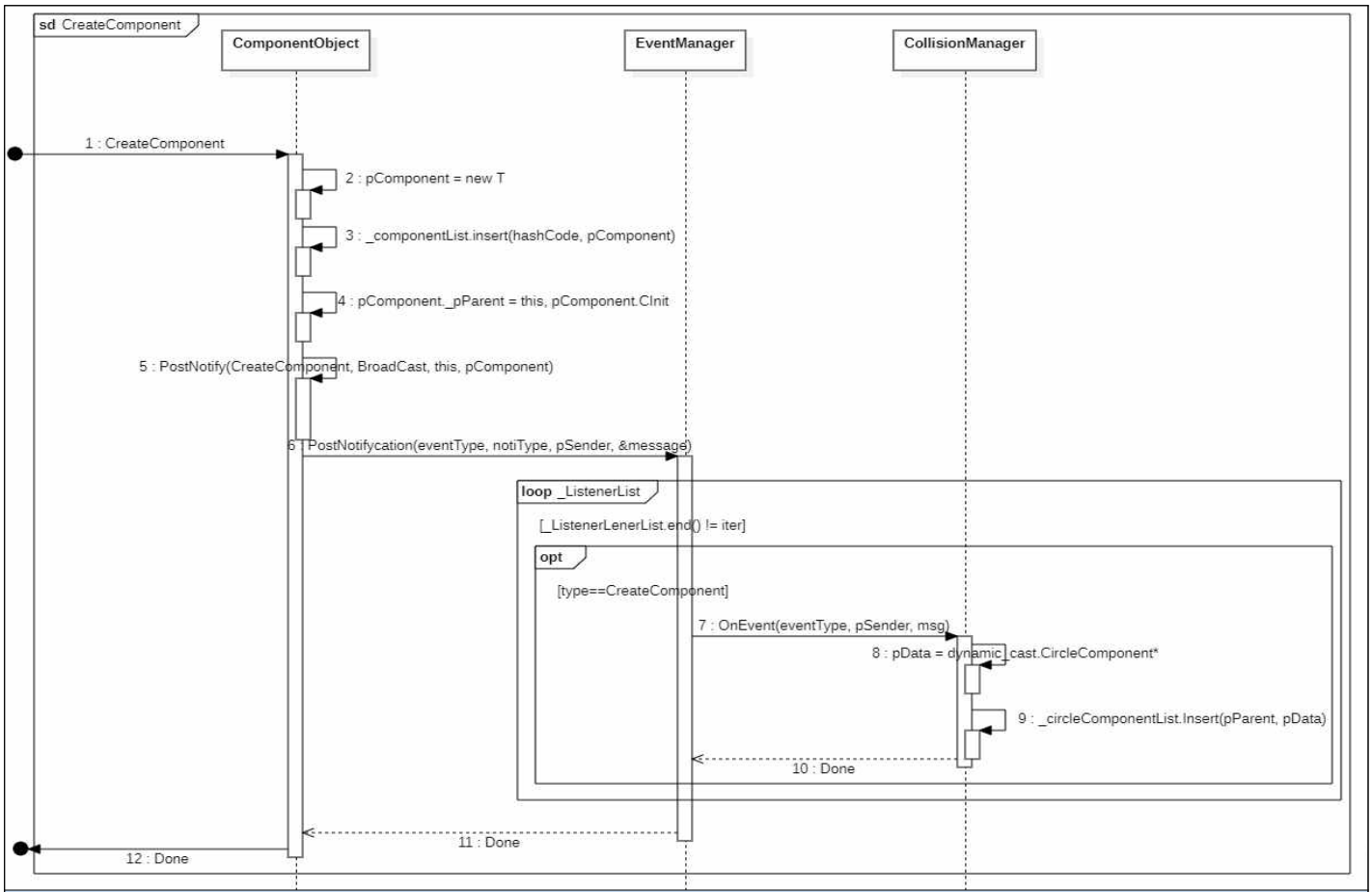
2) Collision 분석

1. 충돌검사를 진행할 임시 CircleComponent리스트를 순회하며 충돌검사를 진행한다.
2. CircleComponent가 추가될 시 CollisionIn이벤트에 부모 오브젝트를 구독시켜 놓는다.
3. 충돌된 CircleComponent[i]의 부모 오브젝트의 고유 아이디를 수신자로 ExtraInfo로 CircleComponent[j]를 담아 EventManager에 CollisionIn이벤트로 통지한다.
4. EventManager는 수신자의 고유 아이디를 검색하여 해당 메시지 내용을 구독 중인 객체에 메시지를 전달한다.

2.6. CreateComponent

게임 오브젝트는 ComponentObject를 리스트로 가지고 있다. 게임 오브젝트 또한 ComponentObject를 상속받은 클래스이며 하나의 타입만 가지고 있을 수 있다. 기능을 모듈화하고 더 나아가 이벤트 기반 프로그래밍과의 결합으로 더욱 쉽게 게임을 제작할 수 있게 하였다. ComponentComponent의 생성 과정을 보며 설명하고자 한다.

1) CreateComponent SequnceDiagram



CreateComponent SequnceDiagram

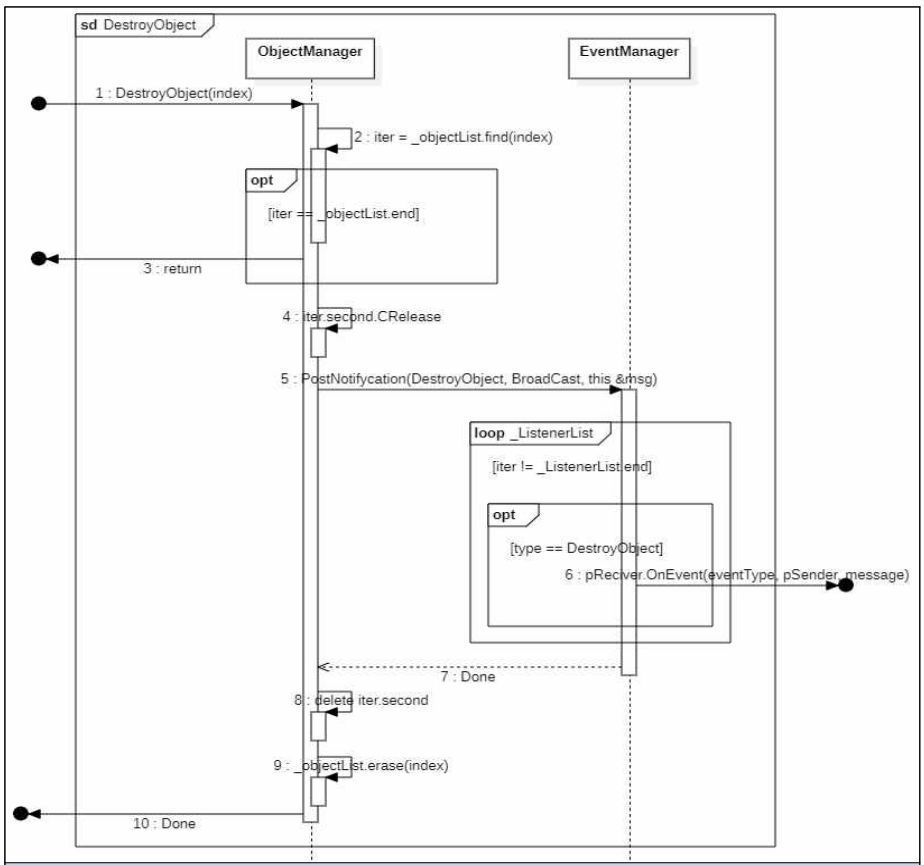
2) CreateComponent 분석

1. ComponentObject의 CreateComponent<T>() 템플릿 메소드에 생성할 Component타입을 기입하여 호출한다.
2. 메소드 내부에서 해당 타입을 이용하여 Component를 생성한다. 타입에 대한 해시코드를 키로 하여 리스트에 추가하고, 부모 지정과 Component에 대한 초기화를 진행한다.
3. Component의 생성 내용을 EventManager에 메시지로 통지한다.
4. EventManager는 메시지 타입을 구독하고 있는 매니저에게 해당 메시지를 통지하고 각 매니저는 해당 타입을 검사하여 각 Component를 관리하는 매니저는 관리 리스트에 추가한다.

2.7. DestroyObject

C++ 프로그래밍에서 객체의 힙 영역에 객체를 생성 시 사용이 끝난 객체는 반드시 삭제해야 한다. 삭제되지 않은 객체는 메모리 누수로 이어지며 메모리 사용량 증가로 인한 많은 문제를 야기할 수 있다. 본 프로젝트에서는 오브젝트의 메모리 생성과 삭제를 ObjectManager에서 전담하도록 하고 오브젝트의 생성을 특정 메소드로 강제하여 개발자의 실수를 줄일 수 있도록 제작하였다. 더 나아가 오브젝트 풀의 사용도 쉽게 구현할 수 있는 구조라 생각한다.

1) DestroyObject SequnceDiagram



DestroyObject SequnceDiagram

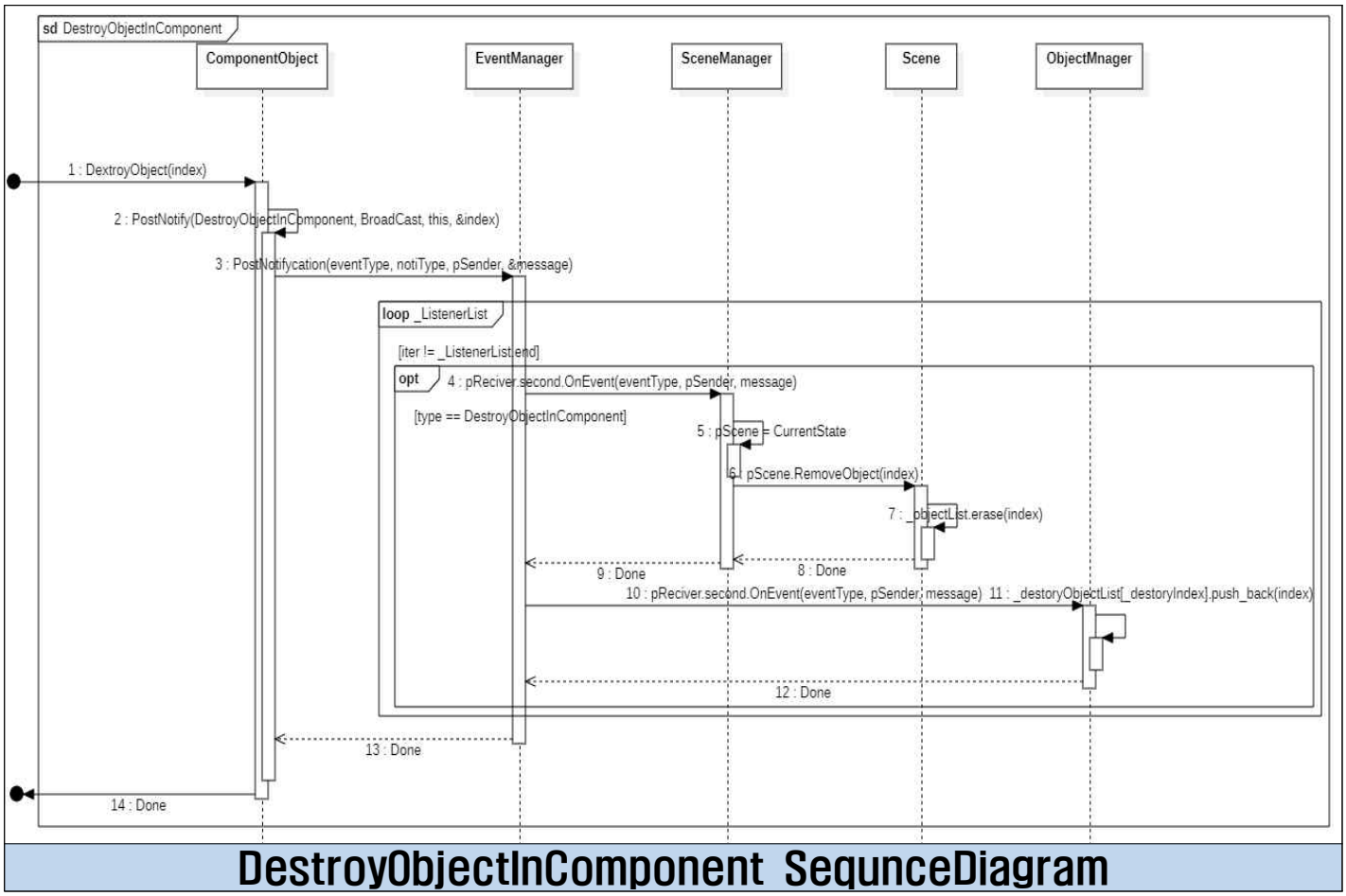
2) DestroyObject 분석

1. 오브젝트 삭제 시 ObjectManager의 DestroyObject메소드에 오브젝트의 고유 아이디를 파라미터로 호출한다.
2. ObjectManager는 해당 고유 아이디로 관리리스트에서 객체를 찾는다. 찾았다면 오브젝트의 CRelease를 호출하여 재귀적으로 객체가 해제될 수 있게 유도한다.
3. EventManager의 DestroyObject메시지를 통보한다. EventManager는 해당 메시지를 구독하고 있는 객체에 해당 메시지를 통지한다.
4. ObjectManager는 완전히 정리된 오브젝트를 delete하고 관리 목록에서 삭제한다.

2.8. DestroyObjectInComponent

객체의 삭제는 매우 중요한 작업이다. 잘못된 메모리 삭제는 메모리 누수와 미정 행동을 유발한다. 객체의 삭제 과정을 보면서 메모리의 누수 없이 객체를 어떻게 해제하였는지 알아보고자 한다.

1) DestroyObjectInComponent SequnceDiagram



DestroyObjectInComponent SequnceDiagram

2) DestroyObjectInComponent 분석

1. 충돌 이벤트 발생 시 적에게 데미지를 주고 HP가 0이면 DestroyObject에 삭제할 오브젝트의 고유 아이디 파라미터로 함수를 호출한다.

2. EventManager는 DestroyObjectInComponent 메시지를 구독하고 있는 SceneManager, ObjectManager 삭제될 오브젝트의 정보를 넘겨준다.

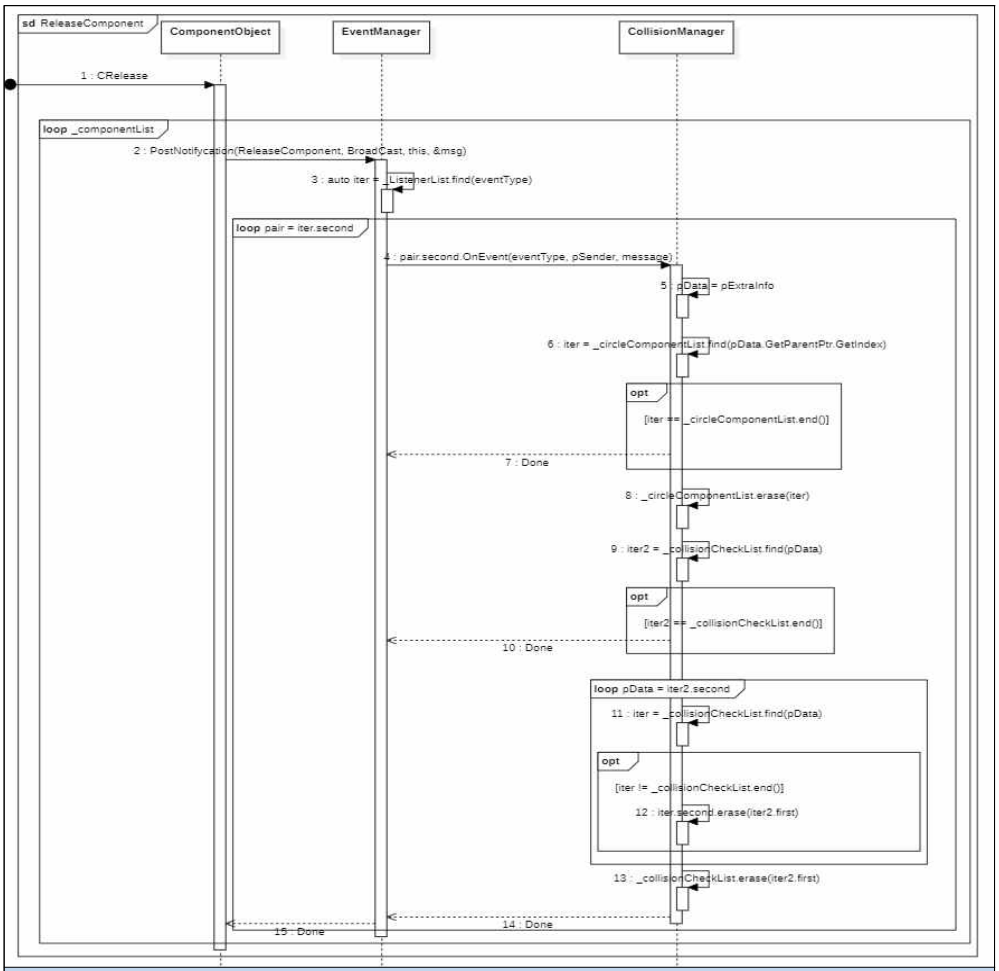
3. SceneManager는 현재 Scene에 접근하여 Scene에서 관리하는 오브젝트를 리스트에서 삭제한다.

4. ObjectManager는 삭제리스트에 삽입한다. 삭제리스트는 다음 CFrame에서 삭제가 된다. 바로 삭제하지 않은 이유는 삭제되는 위치가 루프가 도는 중에 삭제가 될 경우, 많은 연관 클래스에 영향이 가는 경우가 있기에 안전하게 삭제하기 위하여 임시버퍼를 두었다.

2.9. ReleaseComponent

Component는 해제가 될 시 재귀적으로 해제가 되어야 한다. 이 과정에 문제가 있다면 메모리의 누수가 발생할 여지가 있다. Component가 해제되는 과정을 통해 현 프로젝트에서의 메모리 해제 방법을 파악하고자 한다.

1) ReleaseComponent SequnceDiagram



ReleaseComponent SequnceDiagram

2) ReleaseComponent 분석

1. 게임 오브젝트 혹은 어떤 ComponentObject의 CRelease가 호출되면 해당 객체가 관리 중인 자식 Component 리스트를 순회하며 관련 정보를 메시지의 담아 EventManager에 통지한다.
2. 해당 Component를 관리 중인 매니저는 해당 Component에 관련된 내용을 정리한다.
3. ReleaseComponent메시지에 대한 작업이 마무리된다면 통지하였던 자식 Component의 CRelease를 호출하여 자식의 자식 Component 또한 정리될 수 있게 한다.
4. 이후 자식 Component를 delete 한다.