노드 & 컴포넌트 게임엔진 개발 연구

김도현*, 최수빈*
*한국게임과학고등학교 게임엔진연구회
e-mail : lobo_prix@naver.com

A Study on Node & Component Game Engine

Do-Hyun Kim*, Su-Bin Choi*
*Dept of Game Engine Research Team, Korea Game Science High School

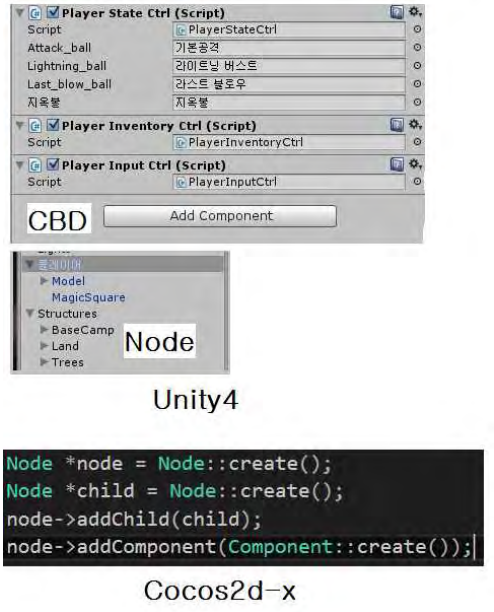
요 약

이 연구에서는 노드 & 컴포넌트 기반 개발(Node & Component Based Development, NCBD)을 지원하는 게임엔진을 직접 설계, 제작하는 내용이다. NCBD를 지원하는 게임엔진은 이미 상용화된 유명게임엔진들(Unity, Unreal, Cocos2d-x)이 공통으로 지원하는 프로그래밍 패러다임이며, 이러한 노드 & 컴포넌트 구조는 깊은 상속구조를 지향하는 구조보다 유연하고 직관적이다.

1. 서론

이 논문은 노드 & 컴포넌트 기반 개발(Node & Component Based Development, NCBD)을 지원하는 게임엔진을 직접 설계, 제작하는 과정과 그 결과물인 Lobo Engine, 그리고 NCBD의 적용결과와 얻게 된 이점들에 대하여 설명한다.

NCBD는 최신 게임엔진들이 공통적으로 지원하는 프로그래밍 패러다임이며, 이 개발방법을 지원하는 상용 게임엔진들(Unity, Unreal, Cocos2d-x 등)이 성공하는 사례가 늘어나고 있다. NCBD는 Node기반 장면구성과 컴포넌트 기반 개발(Component Based Development, CBD)를 통틀어 말하는 것이다. 이 논문에서는 NCBD를 사용자의 입장이 아닌 개발자의 입장에서 설명한다.

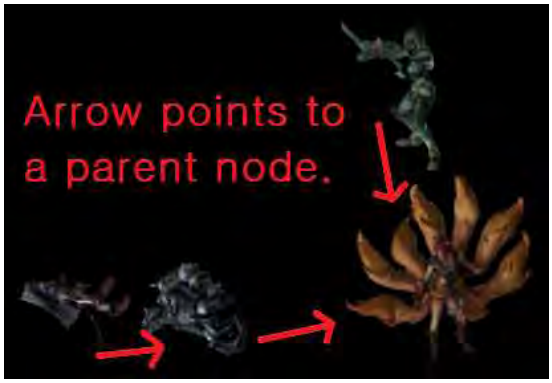


(그림 1) 최신 게임엔진들의 NCBD

2. 정의

2.1 Node

컴퓨터/IT 분야에서 Node는 Tree자료구조에 속하는 하나의 원소를 뜻한다. Node 기반의 장면구조란, 장면을 Tree구조로 조립하는 방법을 말한다. 일반적으로 자식노드는 부모노드에 상대적인 위치, 회전, 크기 값을 가지며, 최상위 노드에서부터 자식노드들을 순회해가며 장면을 그려나간다.



(그림 2) 노드기반으로 구성된 장면. Lobo Engine 사용

2.2 Component

객체지향철학에서 Component는 '객체를 구성하는 객체'라고 하며, 복잡한 작업을 작은 절차(procedure)들로 쪼개는 것과 유사한 개념이다.[1]

2.3 CBD

상속관계는 강한 연관이 생기기 때문에, 깊은 상속구조를 가지는 구조는 상위계층의 변경사항이 하위클래스에 영향이 미치는 경우가 많고 유연하지 못하다. 이 문제의 해결책으로 CBD(Component Based Development)패러다임이 등장했고, 현재까지 게임제작이 널리 쓰이고 있다. CBD의 특징은 깊은 상속구조 대신에 컴포넌트들의 조립

을 통하여 객체를 구성한다.

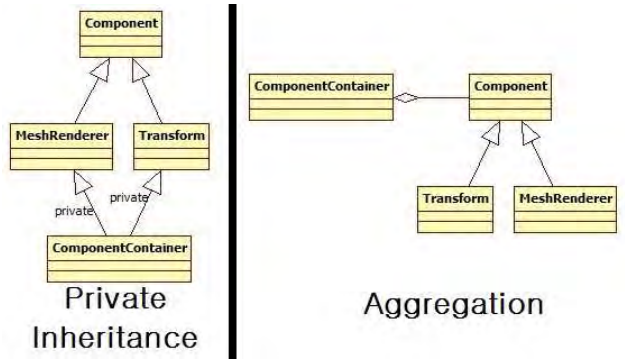
2.4 NCBD

NCBD는 이전에 설명한 CBD에 Node기반 장면구성을 추가한 것으로, 편의상 Node & Component Based Development의 약어로 썼다.

3. 설계 및 구성

3.1 BCD

고품질의 CBD를 지원하려면 우선 BCD(Base Component Development)가 잘 돼있어야 한다. C++에서 BCD의 개발은 private상속과 집약이 있다. 집약을 사용하는 경우 컴포넌트-컨테이너 사이의 결합도가 낮고 실행시간에 컴포넌트들의 조립이 가능하다는 강력한 장점이 있기 때문에 Lobo Engine은 집약관계로 컴포넌트를 디자인하였다.



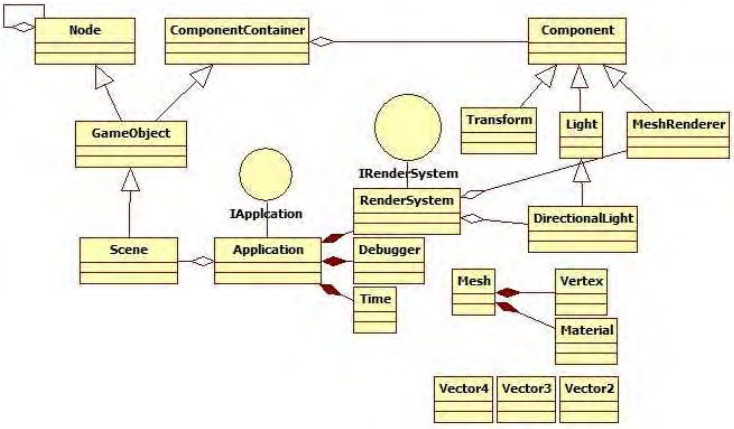
(그림 3) private 상속과 집약

사용자에게 필요한 컴포넌트를 제공해서 조립할 수 있게 하는 것이 CBD의 목표지만, 경우에 따라서는 사용자의 직접제어가 가능해야한다. 그러기 위해서 스크립트 컴포넌트가 있으며, 스크립트 인터프리터기능이 필요하다. 현재는 Python을 엔진에 부착(Embedding)하는 중이다.

3.2 메모리 관리

메모리관리는 모든 곳에 스마트포인터를 사용하는 방법을 사용했으나, 참조카운트를 동기화하는 부분이 오버헤드로 작용한다는 것을 알게 되었고, 메모리관리방식을 Node기반 장면구성의 특징을 활용하는 방식으로 전환하였다. 새롭게 바꾼 메모리관리방식은 부모노드 소멸 시, 자식노드도 함께 소멸하도록 하였다. AttachChild(Node *)로 자식노드를 추가할 수 있다. AttachChild되지 않은 노드의 부모노드는 기본 값 nullptr이며, 이 경우에는 자동으로 메모리해제가 되지 않으므로 수동으로 delete해야 한다. 메모리관리기법은 Qt의 메모리관리를 참고하였다.

3.3 클래스 다이어그램



(그림 4) 클래스 다이어그램

지면관계상 중요한 클래스 몇 개만 추려서 설명하겠다.

1) Application

응용프로그램의 흐름을 담당한다. 플랫폼 종속적인 초기화코드를 Application 내부로 캡슐화하여 크로스플랫폼 지원이 용이하다.

2) RenderSystem

렌더링 API를 캡슐화하여 API 종속적인 부분을 감춰서 추후에 OpenGL이나 다른 렌더링API로의 포팅이 용이하다.

3) Node

자식노드를 추가, 제거할 수 있어서 트리구조의 장면을 구성할 수 있다.

4) ComponentContainer

Component를 map<K,V>로 저장한다. Component 부착 시, 컴포넌트의 OnAttached()를 호출하고, 제거 시 OnDetached()를 호출해주는 역할도 담당한다.

5) GameObject

Node클래스와 ComponentContainer클래스를 상속받는 이 클래스로는 Node기반 장면구성과 CBD를 지원할 수 있다. GameObject에는 기본적으로 Transform컴포넌트가 부착된다.

6) Component

Component들의 일반화 클래스다. 자신을 소유하는 ComponentContainer *owner와 자기 자신이 부착/제거될 때 호출되는 OnAttached(), OnDetached() 가상함수가 선언되어있다.

a) Transform

Transform은 Position, Rotation, Scale값과 월드행렬을 관리하는 컴포넌트다. Transform의 값들은 부모노드에 상대적이기 때문에 트리구조를 알아야하므로, owner를 ComponentContainer * 에서 GameObject *로 다운캐스팅하여 사용하고 있다. TreeComponentContainer같은 인터페이스를 만들어 owner으로 쓰거나, Node를 컴포넌트로 분리하여 참조하는 방식 등으로 개선할 수 있을 것으로 생각한다.

b) Light

Light는 Ambient, Diffuse, Specular 값을 관리하며, 서브클래스인 DirectionalLight에서는 방향이 추가된다. Light컴포넌트는 RenderSystem에 빛을 등록/제거한다.

c) MeshRenderer

기하구조와 재질의 저장 및 RenderSystem에 렌더링 대상으로 등록하는 책임을 가진다. MeshRenderer에서 직접 그린다면 Rendering API가 RenderSystem밖으로 노출되므로 RenderSystem의 캡슐화가 실패한 것이 된다.

4. 엔진내부구조

4.1 main함수

```
int main()
{
    int ret = Application::Create()->Execute();
    Application::Destroy();
    return ret;
}
```

(그림 5) Lobo Engine의 main함수

main함수는 굉장히 간단한 구조다. Application::Create로 Application 객체를 생성한 다음, Execute()를 호출한다. Execute의 메인루프가 끝나면 Application의 자원해제 후, 종료코드를 반환하며 종료한다.

4.2 게임루프

```
int Application::Execute()
{
    MSG msg{};
    while (!end)
    {
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
        else
        {
            Time::Instance()->Tick();
            if (Time::Instance()->IsPaused())
            {
                Sleep(10);
            }
            else
            {
                Update();
                Draw();
            }
        }
    }
    return 0;
}
```

(그림 6) Execute()내부의 게임루프

전형적인 게임엔진 루프의 모습이다. 조금 설명하자면, 매 프레임마다 Time::Tick()을 호출하여 프레임 사이의 dt값을 갱신하고, 일시정지(IsPaused)시 Sleep을 호출하여 CPU자원을 낭비하지 않는다는 것이다.

Update에서는 감시자패턴을 사용하여 UpdateList에 등록된 GameObject들의 Update()함수를 호출한다.

Draw에서는 RenderSystem::Draw를 호출하고, RenderSystem::Draw는 등록된 MeshRenderer리스트를 순회하며 MeshRenderer에 저장된 Mesh를 그린다.

5. 엔진사용자 코드

5.1 MyScene.h

```
#pragma once

#include "Scene.h"

namespace lobo
{
    class MyScene :public Scene
    {
        LE_CREATE(MyScene);
    private:
    private:
        MyScene(){};
        virtual ~MyScene(){};
        void Init();
    protected:
    public:
    };
```

(그림 7) 사용자 정의 클래스 - MyScene.h

LE_CREATE(Type) 매크로는 Type인스턴스의 정적팩토리함수 Create()를 정의해준다. Create()는 가변인자를 Init()에게 넘겨주는데, 이 정적팩토리는 C++11의 가변인자템플릿(Variadic Template)으로 구현되어 Type Safe하다. LE_CREATE가 정적팩토리를 정의하므로 MyScene의 생성자와 소멸자는 private로 선언하였다.

5.2 MyScene.cpp

```
void MyScene::Init()
{
    Scene::Init();

    GameObject * ahri = GameObject::Create(this);
    Transform *tr = LE_GET_COMPONENT(ahri, Transform);
    tr->Position({ 0, -3, 0 });
    MeshRenderer *renderer = MeshRenderer::Create("res/ahri_firefox.obj");
    renderer->Wireframe(false);
    ahri->AttachComponent(renderer);
    async([ahri,tr](){
        while (Time::Instance()->Total() < 20)
            tr->Rotation({ 0, Time::Instance()->Total(), 0 });
    });
```

(그림 8) 사용자 정의 클래스 - MyScene.cpp

MyScene은 Scene을 상속받으므로 Scene의 Init()을 호출한다. 아직 실패하는 경우는 없지만, 실패하는 경우(메모리 할당 실패, 잘못된 인자전달 등)에는 예외를 던지도록 개선할 수 있겠다.

<(그림 3) 클래스 다이어그램>을 보면, Scene도 Node를 상속받으므로 장면을 구성할 수 있다. 그래서 GameObject::Create에 this를 부모노드로 넘겨줬다.

GameObject는 기본적으로 Transform 컴포넌트를 가지므로 Transfrom 컴포넌트를 가져올 수 있다. 매크로 LE_GET_COMPONENT(game_object, Component)는

static_cast<Component *>(game_object->GetComponent("Component Name")))와 같은 의미다.

MeshRenderer은 "res/ahri_firefox.obj"리소스를 읽어들여 Mesh데이터를 생성한다음 GameObject *ahri에 AttachComponent()된다.

다음 줄은 C++11에 추가된 async로 함수비동기호출을 하며, GameObject *ahri가 시간에 따라 y축을 기준으로 회전하도록 구현되어있다.

아래에 더 많은 코드가 GameObject들이 트리형태로 장면을 구성하도록 짜여있지만, 위에서 설명한 초기화방법과 동일한 패턴이기 때문에 지면상 생략하였다.

6. NCBD 적용결과

엔진사용자는 새로운 클래스를 작성하거나 엔진 내부에 접근하지 않고, GameObject에 필요한 기능의 컴포넌트를 부착하는 것만으로 장면을 쉽게 제작할 수 있었다.

게임엔진 또는 그와 비슷한 구조의 프로그램을 제작할 때, NCBD 구조를 사용하는 것이 구조나 활용, 유연성 등에서 더 좋다는 결과를 얻을 수 있었다.

7. 실제 구동 영상

(그림 9) 노드&컴포넌트 기반으로 구성한 장면 영상(.gif)

자식노드들이 부모노드의 Transform에 영향을 받고 있는 것을 확인할 수 있다. 영상을 볼 수 없다면 다음의 링크에서도 확인할 수 있다.

http://blog.naver.com/lobo_prix/220296414880

8. 소스코드

Lobo Engine의 Visual Studio 2013 프로젝트 전체를 아래에서 다운로드 받을 수 있다.

http://blog.naver.com/lobo_prix/220296566314

참고문헌

[1] Marc Gregoire "Professional C++" Wrox