테일러 & 프랜시스가 제공하는 저작권 자료

확장성 및 라이브러리 재사용을 위한 6가지 공장 시스템 트릭

케빈 딜

5.1 소개

요즘 많은 게임들이 데이터-구동 AI를 채용한다. 즉, AI에 대한 의사결정 로직은 C++로 하드코딩되지 않고 대신 구성 파일에 배치되고(일반적으로 XML 또는 일부 유사한 포맷을 사용함) 런타임에 로딩된다. 이 파일은 일반적으로 다양한 유형의 다형성 객체의 구성을 지정한다.

물체의 구현과 소유자 사이의 느슨한 결합을 유지하면서 다형성 물체를 인스턴스화하는 표준 솔루션은 공장 패턴이다. 그러나 모든 공장 구현이 동일하지는 않다. 이 장에서는 게임 AI 아키텍처(GAIA)를 위한 공장 시스템에서 배운 트릭과 교훈을 제시하며, 실제 구현 세부 정보를 제공하는 데 중점을 둔다. GAIA는 이 책의 후반 장에서 자세히 논의된다. 이 장은 특히 공장과 재사용할 가치가 있는 공장 측면에 초점을 맞춘다. 특정 주제는 다음과 같다.

49개.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

여기에 포함된 아이디어를 구현한 후, 당신은 다음과 같은 공장 시스템을 갖게 될 것이다.

5.2 배경

GAIA는 (a) 신규 기능이 필요하면 쉽게 추가할 수 있고 (b) 재사용이 가능하도록 설계된 모듈형 AI 아키텍처로, 다른 게임 엔진(고충실도 군사 시뮬레이션과 같이 게임 엔진이 아닌 것까지 포함)을 포함하여 아키텍처를 프로젝트에서 프로젝트로 빠르고 쉽게 옮길 수 있다. 이 두 가지 요구 사항은 공장 구현에 대한 의미를 갖는다. 그러나 이를 이해하기 위해서는 먼저 공장이 무엇인지, 개념적 추상화와 모듈식 객체가 무엇인지, 상호 작용하는 방식을 이해해야 한다.

5.2.1 공장 패턴

공장 패턴은 물체의 유형을 공장 외부 자체에 노출시키지 않고 데이터에서 다형성 물체를 인스턴스화하는 방법을 제공합니다. 이를 분해하기 위해 구성에 트리거, 스파운드 볼륨 및 기타 유사한 목적으로 사용되는 영역이 포함되어 있다고 상상하십시오. 설계자는 어떻게든 영역을 정의하는 두 가지 이상의 방법을 원하므로 "원" 영역(중심점과 반경을 취함), "직사각형" 영역(축 정렬 직사각형의 두 변을 취함), "다각형" 영역(정점의 임의의 순서를 취하고 그것으로부터 닫힌 다각

데이터에서 각 유형의 영역은 서로 다른 주장을 필요로 하기 때문에 다르게 정의되어야 한다. 그러나 코드에서는 사용되는 각 장소에서 각 유형의 영역에 대한 특수 사례 로직을 갖고 싶지 않습니다. 이는 엄청난 양의 중복을 초래할 뿐만 아니라

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

코드, 그러나 설계자가 프로젝트 후반에 새로운 유형의 영역을 요청하면 완전한 악몽이 될 것입니다! 대신, 우리는 모든 유형의 영역에 대한 인터페이스를 제공하는 모든 영역에 대한 기본 클래스를 정의합니다. 영역을 사용하는 코드는 해당 인터페이스와 단순히 상호 작용할 수 있습니다. 따라서 예를 들어, 각 비플레이어 캐릭터(NPC)는 NPC가 산란해야 하는 영역을 지정하는 데이터에 정의된 산란기를 가질 수 있습니다. 해당 산란기는 모든 유형의 영역을 사용할 수 있지만, 산란기 코드는 기본 클래스(즉, 인터페이스)와 함께 작동하여

그러나 이것은 여전히 문제를 남긴다. 영역을 사용하는 코드 내의 모든 장소는 데이터에 명시된 대로 적절한 하위 클래스(직사각형, 원 등)의 영역 객체를 생성할 수 있어야 하지만, 생성되는 특정 유형의 영역을 알아야 하는 것을 원하지 않으며, 올바른 유형의 영역을 생성 및 로딩하기 위한 코드를 복제할 필요가 없다. 우리는 데이터를 보고, 영역의 유형을 결정하고, 적절한 하위 클래스의 객체를 생성하는 영역 팩토리를 생성하여 이를 해결한다. 즉, 해당 영역의 구현을 위한 인스턴스화, 데이터에 저장된 값(원 중심 및

저거.

AIRRegionFactory::Create(const TiXmlElement\* pElement) {

// 우리가 만들고 싶은 영역의 유형을 얻으십시오

저거.

에어리전 베이스\* pRegion = NULL;

if (nodeType == "Circle") {

pRegion = new AIRegion\_Circle(pElement);

} 기타 if (nodeType == "직사각형") {

pRegion = new AIRegion\_Rect(pElement);

} 기타 if (nodeType == "폴리곤") {

pRegion = new AIRegion\_Poly(pElement);

저거.

저거.

5.2.2 개념적 추상 및 모듈형 구성요소

GAIA는 우리의 AI를 구성하는 다양한 기본 유형의 객체를 나타내는 개념적 추상과 각각의 추상화의 특정 구현인 모듈러 구성 요소를 정의합니다. 즉, 개념적 추상화는 인터페이스이고 모듈러 구성 요소는 해당 인터페이스를 구현하는 하위 클래스입니다. 이전 섹션에서 논의된 영역은 개념적 추상의 한 예이며 직각, 원 및 다각형 영역은 구현하는 서로 다른 유형의 모듈러 구성 요소입니다.

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

그 추상화. GAIA는 다른 많은 개념적 추상화(목표와 행동, 두 가지만 언급하자)를 포함하고 있으며, 각 유형의 개념적 추상화는 다양한 유형의 모듈 구성 요소를 가지고 있다. 각 개념적 추상화는 모든 모듈 구성 요소를 인스턴스화하는 방법을 아는 자체 공장을 가지고 있다. 따라서 지역 공장에서는 다양한 유형의 모든 지역을 인스턴스화하고, 대상 공장에서는 모든 목표를 인스턴스화한다.

5.2.3 신장성

GAIA의 주요 설계 목표(및 모듈러 AI의 주요 이점) 중 하나는 확장 가능하다는 것이다. 즉, 우리는 설계로부터 새로운 요구 사항을 충족하기 위해 새로운 유형의 모듈러 구성 요소를 빠르고 안전하게 추가할 수 있어야 한다. 또한, AI를 구축하는 방식을 개선할 수 있는 새로운 재사용 가능한 개념을 발견함에 따라 새로운 유형의 개념적 추상화를 추가할 수 있어야 한다. 마지막으로, 우리는 상당히 다른 개념적 추상화를 가질 것으로 예상하기 때문에(현재 12개), 그리고 각각의 개념적 추상화는 많은 다른 유형의 모듈러 구성 요소(일부에서는 수십개)에 대한

트릭 #3은 새로운 유형의 모듈러 구성 요소를 추가하고 인스턴스화에서 코드 중복을 최소화하는 방법을 논의하는 반면, 트릭 #5는 새로운 유형의 개념적 추상화를 추가하고 공장 간의 코드 중복을 최소화하는 방법에 대해 논의한다.

5.2.4 재사용

GAIA는 미들웨어처럼 사용하도록 설계되었다. 즉, 어떤 프로젝트에도, 어떤 게임 엔진에도 플러그인되어 사용할 수 있는 독립적인 코드 라이브러리이다. GAIA를 특정 애플리케이션(게임 등)에서 쉽게 분리하여 다른 애플리케이션에서 재사용하기 위해 GAIA는 애플리케이션에 대한 의존성을 가질 수 없다.

개념적 추상화는 이를 위한 한 가지 방법을 제공한다. 우리의 산란 예를 계속하면, 산란 위치의 선택이 GAIA 내부에서 이루어지는 결정의 일부라고 상상한다. 그러나, 설계자들은 게임 특정인 맞춤형 영역을 원한다. 우리의 맞춤형 영역은 게임 엔진에서 구현될 수 있지만, AIRegionBase 인터페이스(GAIA의 일부인)로부터 계승될 수 있다. GAIA의 나머지 GAIA는 게임 특정 코드에 대한 어떠한 의존성 없이 인터페이스를 통해 액세스할 수 있다. 이 예는 게임 특정 영역에 대한 구현이 얼마나 자주 게임

그러나 여전히 문제가 있다. 우리는 응용 특정 모듈러 구성 요소를 인스턴스화할 수 있는 공장에 대한 일부 메커니즘이 필요하지만 공장은 응용 프로그램이 아닌 GAIA의 일부입니다! 따라서 응용 프로그램에 대한 의존성을 추가하거나 GAIA에 내장된 모든 모듈러 구성 요소를 인스턴스화하는 코드를 방해하지 않고 응용 특정 구성 요소를 공장에 추가할 수 있어야 합니다. 이 문제의 해결책은 트릭 #4의 주제이며, 트릭 #1과 #2는 AI 라이브러리의 재사용을 더 쉽게 하는 다른 방법을 논의합니다. 마지막으로 트릭 #6은 코드보다

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

5.3 트릭 #1: 데이터 포맷 추상화

로드 시스템을 설계할 때 가장 먼저 염두에 두어야 할 것은 구성의 형식이 변경될 수 있다는 것이다. 예를 들어, 목록 5.1의 공장은 Tiny XML(인기 오픈 소스 XML 파서[Thomason n.d.]의 객체인 TiXmlElement을 사용한다. 이와 같은 특정 구현에 따라, 또는 심지어 XML과 같은 특정 형식에 따라서는 시간이 지남에 따라 XML이 너무 역전성임을 발견할 수 있거나, 또는 출판사가 이진 데이터 형식을 사용하도록 요구하거나, 또는 설계자가 JSON 또는 YAML에서 작업하는 것이 더

GAIA에서 우리는 애플리케이션이 로드될 때 모든 구성을 파싱하고 AISpecificationNodes로서 내부에 저장한다. 특정 노드는 이름(XML 노드로부터의 라벨), 유형(XML 노드로부터의 유형 속성), 및 속성-부테 및 서브노드(XML로부터의 나머지 속성 및 서브노드)를 갖는다. 이것은 꽤 얇은 래퍼이지만, 예를 들어 JSON 또는 YAML을 지원하기에 충분한 레버리지를 제공한다.

이러한 방식으로 데이터를 랩핑하는 것은 또 다른 주요 이점을 제공한다. 우리는 표현을 제어하기 때문에, 우리는 그 접근 방식을 제어할 수 있다. 따라서, 우리는 데이터를 원시 형식으로 노출시키는 대신, 우리는 당신이 다른 유형의 데이터를 얻을 수 있게 하는 다수의 함수를 제공한다. 이것은 추상적 유형(즉, 영역과 같은 개념적 추상화)뿐만 아니라 간단한 유형(플로트, 정수, 부울, [x, y, z] 위치 등)으로 판독하는 함수를 포함한다. 결과적으로, 우리는 다음과 같은 것들을 할 수 있다.

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

5.4 트릭 #2: 초기화 입력 캡슐화

모듈러 컴포넌트를 초기화하는 것은 종종 단지 그 컴포넌트의 사양 노드 이상의 것을 필요로 한다. 예를 들어, NPC용 AI의 컴포넌트의 경우, 컴포넌트는 그것이 일부인 NPC 또는 그 NPC의 AI 컴포넌트 모두가 사용하는 공유 칠판에 대한 액세스가 필요할 수 있다. 또한, 특정 모듈러 컴포넌트가 이러한 것들을 필요로 하지 않더라도 동일한 개념적 추상화를 공유하는 다른 컴포넌트는 할 수 있거나 포함하는 컴포넌트를 포함할 수 있으므로, 우리는 그것이 필요할 것이라고 생각하든 그렇지 않든 상관없이 모든 객

간단한 해결책은 모든 초기화 입력(명세 노드 포함)을 단일 객체로 랩핑하는 것이다. GAIA에서, 그것은 AICreationData이다. 각 생성 데이터는 AISpecificationNode 및 이 구성 요소가 제어하는 데 도움이 되는 NPC를 재전송하는 AIActor에 대한 포인터뿐만 아니라 다양한 다른 필수 및 선택적 메타데이터를 포함하도록 요구된다.

이 접근법(즉, AICreationData에 모든 메타데이터를 캡슐화하는 것)의 한 가지 좋은 점은 공유 데이터 및 기능을 공장 시스템에 추가할 때 모자를 걸 수 있는 장소를 제공한다는 것이다. 예를 들어, 응용 프로그램이 응용 프로그램별 데이터를 응용 프로그램별 구성 요소에 전달해야 하는 경우가 종종 있을 것이다. 응용 프로그램은 AICreation에 추가될 응용 프로그램 블랙보드를 가질 수 있다. 데이터 재현은 빈 데이터(데이터가 발생하는 속도를 제어할 수 있도록)와 같은 것을 처리하는 다른 방법이다. 응용 프로그램에서 더 나은 응용 프로그램(AIC) 및 응용 프로그램

AICreationData의 유용성에 대한 다른 예는 아래의 트릭 중 일부에서 찾을 수 있다.

5.5 트릭 #3: 일관된 객체 구성 및 초기화

GAIA를 구현할 때 우리가 저지른 한 가지 실수는 모든 공장이 물체를 초기화하는 데 고유한 접근법을 사용할 수 있도록 한 것이다. 우리의 초기 공장은 목록 5.1에 표시된 것과 같이 작동했는데, 즉, 단순히 TiXmlElement, AISpecificationNode 또는 AICreationData를 생성하는 물체의 구성체에 통과시켰습니다.

그러나 개발이 진행됨에 따라 이 접근법으로는 불가능한 기능을 공장에서 지원하고 싶다는 것이 발견되어 시간이 지남에 따라 다양한 공장에서 사용하거나 단일 공장에서 다양한 유형의 모듈러 구성 요소가 사용하는 다양한 접근법을 사용하기 시작했습니다. 이는 코드를 이해하면 관리할 수 있지만 GAIA를 새로운 응용 프로그램으로 통합하려는 엔지니어에게 매우 혼란스러운 혼란을 초래했다. 결과적으로 우리는 GAIA를 새로운 응용 프로그램으로 통합하려는 엔지니어에게 매우 혼란스러운 문제를 해결했습니다.

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

객체 구성 및 초기화에 대한 강력한 접근법과 이를 사용하기 위해 전체 라이브러리를 천천히 변환합니다. 이 접근법은 다음과 같은 네 가지 기본 단계를 가지고 있습니다.

1. 객체를 인스턴스화합니다.

2. 객체에 사전 로드 기능을 선택적으로 실행시키며, 이는 객체가 초기화되기 전에 객체의 소유자에 의해 디폴트 값이 설정될 수 있게 한다.

3. 생성 데이터에서 객체를 초기화합니다.

4. 초기화에 성공했는지 확인합니다. 그렇다면 객체를 반환합니다. 그렇지 않으면 삭제하고 NULL을 반환합니다.

이 4단계 프로세스가 일관되게 적용되도록 하기 위해 목록 5.2와 같이 AICreationData의 일부로 구현했다.

저거.

공개: // 표시되지 않았습니다. 트릭 // #2의 메타데이터에 대한 액세서리 및 저장소.

아...

// PreLoadCallback는 객체가 인스턴스화되었지만 그것이 초기화되기 전에 ConstructObject()에 의해 호출된다.

typeedef void (\*PreLoadCallback) (AIBase\* 객체); void SetPreLoadFunction(PreLoadCallback pFunction) { m\_pPreLoadFunction = pFunction; }

// 지정된 유형의 객체를 구성하고, 이 // 생성 데이터를 사용하여 이를 초기화한다.

템플릿<클래스 T> T\* ConstructObject() const { T\* pObject = new T;

if(m\_pPreLoadFunction) m\_pPreLoadFunction(pObject);

bool bSuccess = pObject->In //it(\*this); if (!bSuccess) { log에 오류를 인쇄합니다!

AI\_ERROR ("타입 " "'%s", GetNode().GetType().c\_str())의 객체를 초기화하지 못하였다);

pObject를 삭제; pObject = NULL; }

return pObject; }

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

프리로드 함수를 상당히 미묘한 구문을 가진 인포지먼트로 통과시킨다. 또한 우리의 모든 모듈러 컴포넌트가 AIBase(모든 인터페이스의 기본 클래스임)로부터 상속된다는 사실에 의존한다. 우리가 다시 시작한다면, 우리는 아마도 프리로드에 펑커 객체를 사용하여 이 두 가지 문제를 모두 해결할 것이지만, 그 개선의 세부 사항은 이 장의 범위를 벗어난다(특히 아직 구현되지 않았기 때문에!).

목록 5.3은 모든 개선 사항을 가진 Create() 방법을 보여준다. 이 기능은 목록 5.1만큼 간결하다는 점에 유의하십시오. 새로운 유형의 모듈식 객체를 추가하는 것은 단지 두 줄의 코드에 해당하는 if-then-else에 다른 조항을 추가하는 것을 의미합니다. 또한 생성 데이터가 사전 로드 기능을 지원하지만 공장 자체는 이를 사용하지 않습니다. 사용된 경우 사전 로드 기능은 공장이 만들고 있는 객체의 소유자에 의해 설정되며 일반적으로 기본 값을 지정하는 데 사용됩니다.

저거.

AIRRegionFactory::Create(const AICreationData& cd) {

// 우리가 만들고 싶은 영역의 유형을 얻으십시오

const AISTRING&NodeType = cd.GetNode().GetType();

// 지정된 유형의 영역을 생성합니다

에어리전 베이스\* pRegion = NULL;

반환 지역;

5.6 트릭 #4: 외부 코드를 AI에 주입하는 방법

이 시점까지의 트릭들은 사물들을 인스턴스화하는 과정에 초점을 맞추어 왔으며, 나머지 트릭들은 공장 시스템의 다른 측면들에 초점을 맞출 것이다.

우리가 다루는 다음 문제는 애플리케이션에 대한 AI 의존성을 생성하지 않고 AI 라이브러리에 애플리케이션별 코드를 주입해야 하는 것이다. 이것은 많은 프로젝트에서 재사용될 것으로 예상되는 라이브러리에 특히 중요하지만 단일 프로젝트 내에서도 AI가 게임에서 깨끗하게 분리되는 것이 매우 좋을 수 있다.

우리는 게임이 팩토리에 맞춤형 객체 생성자를 추가할 수 있게 함으로써 이것을 달성한다. 이러한 객체 생성자를 구성자라고 부르지만, 클래스의 구성자 기능과 혼동해서는 안 된다. 구성자는 특정 개념 추상화를 위해 모듈러 구성요소의 일부 하위 집합을 인스턴스화하는 방법을 아는 일종의 미니 공장이다. GAIA 라이브러리에는 각 개념 추상에 대한 기본 구성자가 포함되어 있다. 해당 기본 구성자는 모듈러 구성요소를 인스턴스화하는 역할을 한다.

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

핵심 라이브러리에 내장되어 있다. 애플리케이션은 또한 자체 구성자를 구현하여 공장에 추가할 수 있다. GAIA의 대부분의 것과 마찬가지로 구성자는 모두 인터페이스를 정의하는 공통 기본 클래스를 공유하고 공장은 해당 인터페이스와만 상호 작용하므로 구성자의 응용 특정 의존성은 GAIA에 어떠한 요구 사항도 두지 않는다.

구체적인 예를 들어, 우리 지역 공장으로 돌아가 보자. 우리는 AI 라이브러리에 원, 직사각형, 다각형 영역의 세 가지 유형의 영역을 구축했습니다. 우리 게임이 특정 탐색 노드에 묶인 맞춤형 영역을 필요로 하거나, 우리 게임이 수중 이동을 위한 맞춤형 역학을 가지고 있고 이러한 역학을 반영하는 영역을 가져야 한다고 상상해 보거나, 우리 게임이 주차 차고와 같은 다층 구조를 지원한다고 가정해 보아라. 우리는 지역의 위치와 범위를 정의하는 게임별 개념을 미리 사용하는 방법을 아는 게임별 영역을 추가할 수 있어야 한다.

이를 처리하기 위해 먼저 AIRegionBase(원, 직사각형 및 다각형 영역과 마찬가지로)에서 모두 계승되는 하나 이상의 게임별 영역 클래스를 구현합니다. GAIA는 게임에 의존할 수 없지만 게임이 GAIA에 의존하는 것은 무방하므로 게임별 코드에 AIRegionBase.h를 포함하는 데 문제가 없습니다. (이것이 사실이 아니라면 게임이 GAIA를 전혀 사용할 수 없습니다. 마지막으로 게임이 이미 개시되어 있을 때 게임 내의 특정 영역에 대한 특정 구성 요소를 추가하거나, 특정 구성 요소를 검색할 때까지 게임 내의 특정 영역에 대한

매우 드물지만 구성체로 지원하기 쉬운 또 다른 트릭은 프로젝트가 때때로 일부 모듈 구성 요소의 내장 구현을 맞춤형 버전으로 교체하기를 원한다는 것이다. 예를 들어, AI 라이브러리가 (x,y,z) 삼중선 이외의 것을 사용하여 위치를 지정하는 프로젝트에 의해 사용된다고 상상해 보세요. (이는 완전히 조작된 예가 아닙니다. GAIA를 사용하는 프로젝트 중 하나는 이 문제가 있지만 이것이 우리가 해결한 방법이 아닙니다. 따라서 우리는 여전히 직사각형, 원형 및 다각형 영역을 갖고 싶지만, 내장 영역이 작동하지 않을 것입니다. 프로젝트

이 장에서 중복을 줄이기 위해(우리 코드에서와 마찬가지로) 이 트릭의 지역 공장을 보여주는 코드 목록을 제공하는 것을 자제할 것이다. 이러한 아이디어의 구현은 목록 5.4 및 5.5의 다음 트릭의 아이디어와 함께 찾을 수 있다.

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

저거.

공공: 가상 ~AIC구성자베이스() {}

// 생성 데이터에서 객체를 생성하려고 시도한다. // 순수 가상하여 아동 수업이 강제적으로 // 구현할 것이다.

가상 T\* Create(const AICreationData& cd) = 0; };

템플릿<클래스 T>클래스 AIFactoryBase { 공개: 가상 ~AIFactoryBase();

// 맞춤형 시공자를 추가합니다. 소유권을 취합니다.

void AddConstructor(AIConstructorBase<T>\* pCnstr) { m\_Constructors.push\_back(pCnstr); }

// 영역을 생성할 수 있는 구성체를 모두 살펴본다. 생성 데이터의 유형인 객체를 처리하는 방법을 모르는 구성체는 NULL을 반환해야 한다.

T\* Create(AICreationData& cd);

프라이빗: std::벡터<AIC구축기<T>\*>m\_구축기; };

템플릿<클래스 T>T\*AIFactoryBase<T>::Create(AICreationData& cd) {T\* pRetVal = NULL;

// NOTE : 정지 조건에 주의를 기울여 - 우리는 // 이 생성 데이터를 처리할 수 있는 구성자를 찾자마자 터져 나간다. 우리는 그것들을 추가된 // 역순으로 시도하고 싶으므로 루프 // 역순으로 시도한다.

(!pRetVal) AI\_ERROR\_CONFIG("Factory"가 "%s" 타입의 객체를 생성하지 못하였다면, cd.GetNode().GetType());

return pRetVal;

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

저거.

저거.

가상 AIRregionBase\* Create(const AICreationData& cd);

};

에어리전 팩토리() {

AddConstructor(새로운 AIRegionConstructor\_Default);

}

5.7 트릭 #5: 템플릿과 매크로를 사용하여 공장 정의를 표준화합니다.

트릭 #4의 접근법은 공장에 대한 강력한 기반을 제공하지만, 우리는 모든 개념을 추상화하기 위해 모든 코드를 복사하고 붙여넣을 필요가 없으므로, 만약 그렇게 한다면, 공장이 시간이 지남에 따라 분기될 것이라는 것이 사실상 보장된다(그리고 실제로 GAIA에서는 그랬다). 이것은 각 공장이 대부분 동일하지만 다른 모든 공장과 약간 다르기 때문에 빠르게 유지 보수 악몽이 된다.

좀 더 자세히 살펴보면, 지역 공장과 다른 공장(행위 공장이나 대상 공장 등)의 주요 차이점은 다음과 같다.

1. 그들이 생성하는 객체의 유형(AIRregion vs. AITarget 또는 AIAction). 2. 디폴트 구성자의 Create() 기능의 구현(적절한 유형의 객체를 실제로 인스턴스화해야 한다).

첫 번째 단계로 생성되는 객체의 유형을 취하고 대부분의 복제를 처리하는 C++ 템플릿을 사용할 수 있다. 결과는 목록 5.4에 나와 있다.

이렇게 되면 목록 5.5와 같이 지역 공장의 신고가 훨씬 짧아진다. 참고로 이 목록에서 알 수 있듯이 GAIA의 공장도 싱글톤이다. sin-gleton 패턴은 이 장의 범위를 벗어났지만 잘 알려져 있으며 GAIA는 그것에 대해 끔찍하게 특이한 일을 하지 않는다.

목록 5.5는 꽤 좋지만 여전히 많은 코드가 있습니다. 우리는 수시로 새로운 개념적 추상화를 추가할 것이며 이것이 가능한 한 간단하기를 원합니다. C++ 매크로로는 할 수 있습니다. 매크로 프로그래밍은 고통스럽지만 우리가 필요로 하는 매크로는 상당히 간단하며 우리는 한 번만 구현하면 됩니다. (운이 있는 독자 여러분, 우리의 예에서 이익을 얻을 수 있습니다).

첫 번째 단계는 목록 5.5의 코드를 구성할 수 있지만 "지역" 대신 다른 단어를 대체할 수 있는 매크로를 작성하는 것입니다. 따라서 우리는 "액션" 또는 "타겟"을 매크로에 전달하여 액션 공장 또는 대상 공장을 얻을 수 있습니다. 그 매크로는 목록 5.6에 나와 있습니다.

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

저거.

저거.

저거.

저거.

저거.

다음으로 다른 매크로를 호출하고 각 개념적 추상화의 이름으로 각 매크로를 정의한다. 목록 5.7은 목록 5.6의 매크로를 사용하여 실제로 공장을 만드는 호출과 함께 이 매크로가 지역, 행동 및 목표 개념적 추상화만 있으면 어떻게 보일지 보여준다.

저거.

저거.

GAIA\_EXECUTE\_FACTORY\_MACRO는 또한 각 공장에 대한 싱글톤 객체를 정의하고 개념적 추상화를 글로벌 객체 관리자에 추가하는 데 사용된다. 따라서, 우리가 GAIA에 새로운 유형의 개념적 추상화를 추가하고자 할 때, 우리가 해야 할 일은 GAIA\_EXECUTE\_FACTORY\_MACRO에 추상화의 이름을 추가하는 것뿐이다. 특히 이 시점에서 두 번은 매우 드물게(혹은 이 시점에서) 새로운 개념적 추상화를 추가하기 때문에 매년 우리가 해야 할 모든

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

5.8 트릭 #6: 글로벌 오브젝트 구성

복제는 코드에서와 마찬가지로 구성에서 많은 문제가 될 수 있습니다. 예를 들어, 우리는 여러 다른 NPC에 대한 산란 볼륨을 정의하는 영역이 있다고 상상해 보십시오. 우리는 이 영역을 모든 NPC 구성에 복사하고 붙여넣을 필요가 없으며, 다른 것이 아니라면 설계자가 게임을 조정함에 따라 변경될 가능성이 있으며, 이를 변경하기 위해 모든 복사본을 추적할 필요가 없다고 생각합니다!

글로벌 객체 구성은 특정 객체에 대한 구성을 한 번 정의한 다음, 우리가 그것을 제자리에 복사하고 붙여 넣은 것처럼 다른 곳에서 사용할 수 있도록 한다. 우리는 모든 개념적 추상화를 위해 이것을 할 수 있으므로 글로벌 지역, 글로벌 목표, 글로벌 행동 등을 가질 수 있다. 구성 파일에서 우리는 개념적 추상화의 유형(지역 정의, 목표 정의, 행동 정의 등)에 따라 명명된 특수 노드에 글로벌을 배치한다. 각 글로벌 구성에는 다른 곳에서 검색하는 데 사용되는 고유한 Circle 이름이 주어져야 한다. 예를 들어, 우리는 우군과 적군의 산란 구역

<지역 정의> <지역명="친절하게 산란하는 지역" 유형="원" 중심="(0,0,0)" 반경="100"/> <지역명="적개산하는 지역" 유형="원" 중심="(300,0,0)" 반경="100"/>

</지역 정의>

그런 다음 다음과 같이 특수 유형 '글로벌'과 고유 이름을 사용하여 이러한 지역을 참조할 수 있다.

<지역 유형="글로벌" 이름="EnemySpawn지역"/>

이 작업을 수행하기 위해, 우리는 두 가지가 필요하다. 첫째, 우리는 싱글톤인 글로벌 객체 관리자가 필요하다. 구성이 파싱될 때, 글로벌 객체 관리자는 모든 글로벌 정의 노드(예: RegionDefinitions node)에서 판독하고 모든 글로벌 구성을 저장하는 데 다시 책임이 있다. 둘째, 목록 5.4의 AIFactoryBase 클래스에서 템플릿화된 Create() 함수가 확장되어야 구성자를 호출하기 전에 모든 글로벌을 해결한다. 수정된 함수 정의는 목록 5.8에 나와 있다.

저거.

저거.

// 사양 노드는 글로벌 관리자에 저장되어 있습니다.

const AISpecificationNode& node = cd.GetNode();

const AISTRING&NodeType = node.GetType();

저거.

저거.

저거.

저거.

테일러 & 프랜시스가 제공하는 저작권 자료

노드.에트리뷰트 스트링("이름");

const AISpecificationNode\* PActualNode =

AIGlobalManager::Get().Find(globalName);

if (!p액추얼노드) {

} 기타 {

// 생성 데이터의 노드를 설정합니다.

// 이 글로벌에 대한 실제 노드, 생성

// 객체, 그런 다음 생성 상에 노드를 설정

// 데이터를 이전 값으로 되돌립니다.

CD.SetNode(\*PacttualNode);

pRetVal = Create(cd);

CD. 셋노드(node);

return pRetVal;

}

}

// 나머지는 목록 4와 동일합니다.

아...

5.9 결론

이 장에서는 재사용 가능한 AI 라이브러리인 GAIA를 위한 공장 시스템의 오류 수정 및 재팩터링의 힘든 작업을 통해 배운 여러 가지 다른 트릭을 다뤘다. 함께, 이러한 트릭은 모든 공장 전반에 걸쳐 객체를 지정하고 초기화하는 일관된 접근법을 제공하고, AI 라이브러리를 나머지 게임으로부터 분리할 수 있게 하고, 공장 코드를 통합하여 새로운 공장(모든 서포트 구조와 함께)이 단일 코드 라인으로 생성될 수 있도록 하며, 공장 코드 내에서 그리고 구성 자체 내에서 중복을 크게 줄인다. 결과는 확장성(새로운 모듈

참조

딜, K. 2016. 모듈형 AI. 게임 AI 프로 3, 에디. S. 라빈. 보카 라톤, FL: CRC 프레스, pp. 87-114

감마, E., R. 헬름, R. 존슨, J. 블리시데스 1995. 디자인 패턴: 재사용 가능한 객체 지향 소프트웨어 요소. 보스턴, MA: 애디슨-웨슬리, pp. 107~116.

토마슨, L. n.d. TinyXML 메인 페이지. http://www.grinninglizard.com/tinyxml/(2016년 6월 10일에 액세스됨).

저거.