1단계_12. 언리얼 엔진의 메모리 관리

수강일	@2023/04/04
이름	전영재
멘토	Min-Kang Song,현웅 최
작성 상태	In progress
단계	1단계
강의 시청 여부	☑

Contents

강의 제목

💡 **강의 목표**

- 언리얼 엔진의 메모리 관리 시스템의 이해
- 안정적을 언리얼 오브젝트 포인터를 관리하는 방법 학습

---

[1] 언리얼 엔진의 자동 메모리 관리

C++언어 메모리 관리의 문제점

메모리주소에 프로그래머가 직접 접근하여 **할당**(`new`) 과 **해지**(`delete`)를 해줘야한다.

⇒ 실수가 발생하기 쉽고, 한번의 실수로 전체 프로젝트가 뻗을 수 있다.

그래서 C++이후 언어들은 포인터를 버리고 **가비지컬렉션** 시스템을 도입했다.

<실수예시>

메모리 누수 (Leak)	new는 했지만 delete를 하지않음 → 힙에 메모리가 남는다.
허상 포인터 (Dangling)	이미 delete 된 주소를 가르키는 포인터
와일드 포인터	값이 초기화되지 않아 이상한 주소를 가르키는 포이터

가비지 컬렉션 시스템

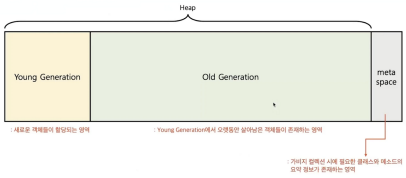
더 이상 사용하지 않는 오브젝트를 자동으로 감지해 메모리를 회수하는 알고리즘이다.

일반적으로 **마크-스윕** 방식을 사용한다.

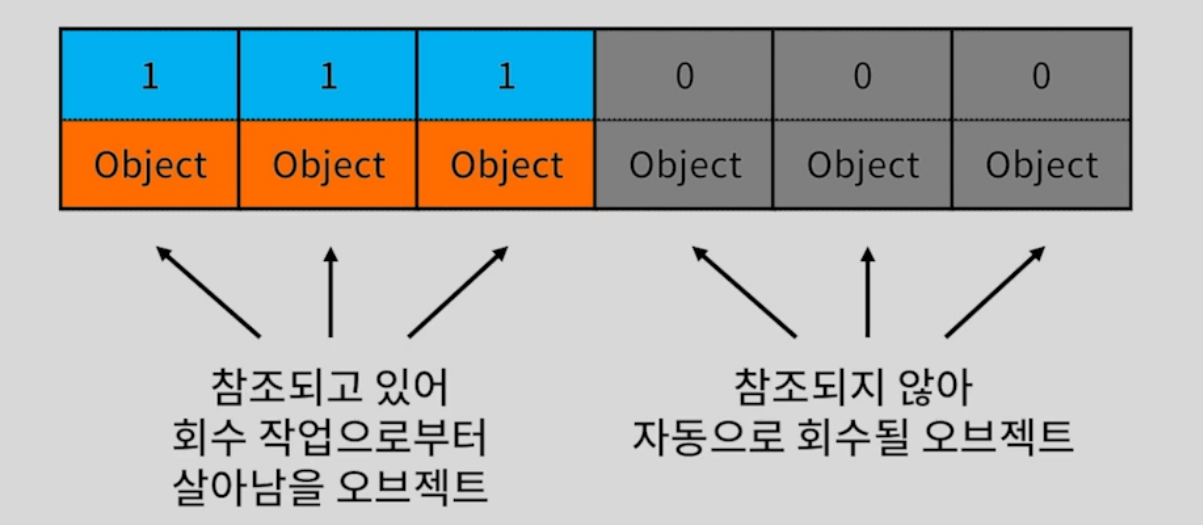
Garbage Collector의 동작 원리

한줄 요약

https://sihyung92.oopy.io/java/garbage-collect/1



1. 저장소에서 최초 검색을 시작하는 **루트 오브젝트** 표기
2. 루트 오브젝트가 참조하는 객체를 **마크**
3. 마크된 객체로부터 **다시 참조하는 객체를 찾아 마크**하고, 이를 **반복**한다.
4. 위 결과, 저장소에는 마크된 객체와 마크되지 않은 객체, **두 그룹으로 분류**된다.
5. 가비지 컬렉터가 마크되지 않은 객체(가비지)들의 메모리를 **회수**한다.

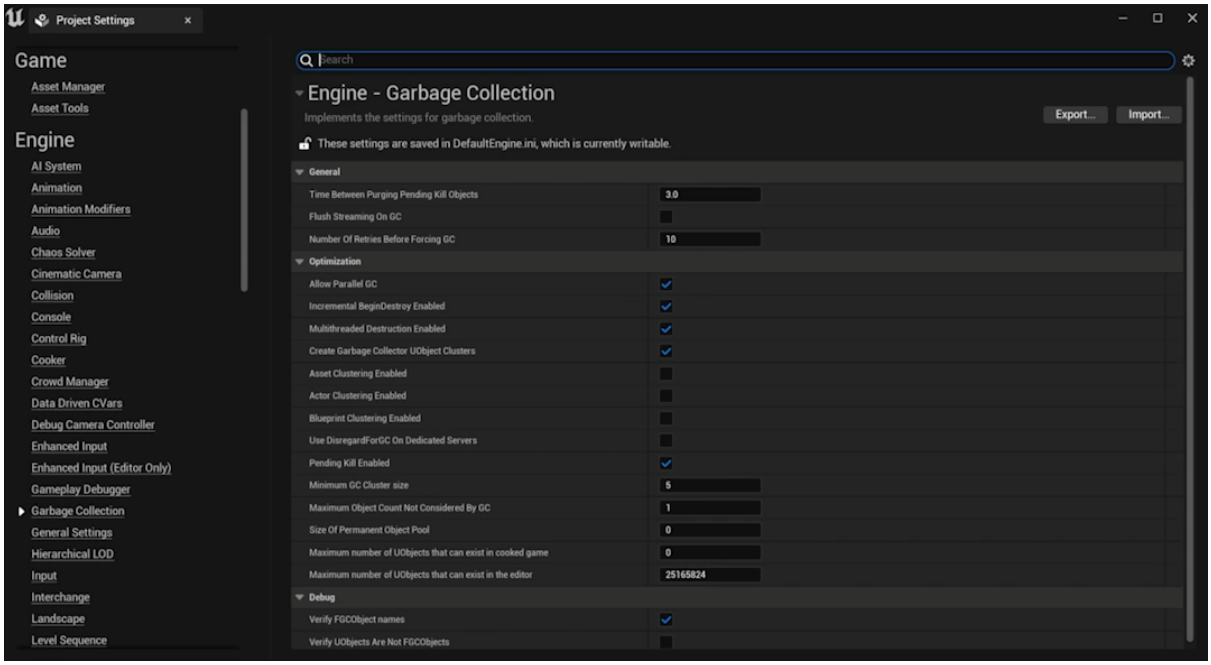


참조되고 있다=1 / 참조되지 않는다=0. 후에 0번인 데이터만 싹 스윕하면 된다.

언리얼 엔진의 가비지 컬렉션

자체적으로 마크-스윕 방식의 가비지 컬렉션을 구축했으며, 에디터 내의 **Project Settings**에서 설정을 확인할 수 있다.

이런 가비지 컬렉터가 돌아가는 것만으로도 적지않은 리소스 소모가 드는데, 언리얼은 성능향상을 위해 **병렬처리**, **클러스터링** 같은 기능을 탑재했다.

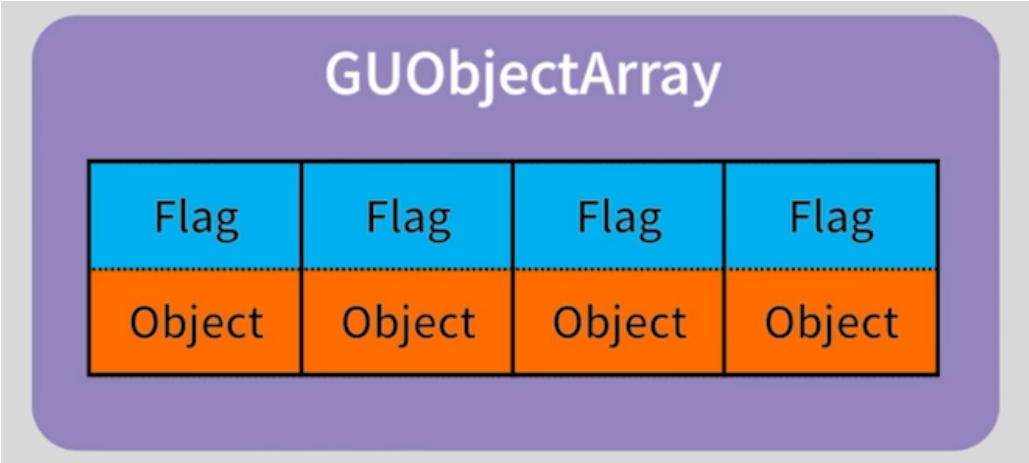


가비지컬렉션을 위한 객체저장소

언리얼엔진에는 관리하는 **모든 언리얼 오브젝트의 정보를 저장**하는 전역변수인 `GUObjectArray`가 존재하며, 위 설명에서의 저장소라고 생각하면 되겠다.

이 Array의 각 요소에는 플래그가 설정되어 있다.

- Garbage플래그 : 다른 오브젝트로부터의 참조가 없어 **회수 예정**인 오브젝트
- RootSet플래그 : 다른 오브젝트로부터의 참조가 없어도 **회수하지 않는** 특별한 오브젝트



언리얼엔진의 가비지 컬렉터는GUObjectArray의 플래그를 확인해 빠르게 회수목표를 파악하고 제거한다.

또한 언리얼엔진의 가비지 컬렉터는 지정된 시간마다 **주기적**으로 메모리 회수한다.

위 GUObjectArray에서 회수 판단의 척도가 되는 Garbage플래그는 프로그래머가 수동으로 설정하는게 아니라 시스템에서 **자동으로** 설정하기에 더욱 안전하다.

> ⇒ 그렇기에 오브젝트를 삭제할때는, delete로 직접삭제하는게 아니라 **참조를 없앰으로써 가비지컬렉터가 삭제하게 냅두는 것**이다.

💡 중요한 오브젝트가 있어 따로 관리해야 한다면 `AddToRoot`를 통해 RootSet플래그를 설정해 보호 설정을 할 수 있다. 나중에 필요 없으면 `RemoveFromRoot`로 없애면된다.

언리얼 오브젝트를 통한 포인터 문제의 해결

1. 가비지 컬렉터를 통해 **자동으로** 메모리 문제를 해결하기에 메모리 누수문제가 없다.

   ⇒ 단, C++오브젝트는 직접 신경써야한다. (스마트포인터 라이브러리를 써도 된다.)

2. 댕글링 포인터문제 해결가능
   - 사용여부 판단을 위해, `IsValid()` 함수를 제공한다.
   - 단, C++오브젝트는 직접 신경써야한다.
3. 와일드 포인터문제
   - 오브젝트를 `UPROPERTY` 속성을 지정해주면 자동으로 초기값을 nullptr로 초기화 해준다.
   - 마찬가지로 C++오브젝트는 직접 nullptr로 초기화 해줘야 한다.

회수되지 않는 언리얼 오브젝트

- `UPROPERTY`로 참조된 오브젝트는 가비지컬렉터가 회수하지 않는다.
- 또는 `AddReferencedObject` 함수를 통해 참조 설정
  - 근데 자주 안쓴다.
- RootSet으로 지정한 오브젝트도 회수하지 않는다.
  - 진짜 중요한데만 쓰는건데, 잘안쓴다.

> ⇒그래서 가급적 오브젝트 포인터는 **UPROPERTY로 선언**하고 메모리는 **가비지컬렉터가 관리하도록 위임**하는게 좋다.

💡 UPROPERTY를 사용하지 못하는 경우에서 언리얼 오브젝트를 관리해야 하는경우, FGCObject 클래스를 상속받은후, AddReferencedObjects 함수를 구현하면 된다.
⇒ 컨텐츠 제작에서 잘쓰진 않는다.

언리얼 오브젝트의 관리원칙

1. 생성된 언리얼 오브젝트를 유지하기 위한 레퍼런스 방법을 설계할것
   - 언리얼 오브젝트 내의 언리얼 오브젝트 : UPROPERTY 사용
   - C++ 오브젝트 내의 언리얼 오브젝트 : FGCObject 상속 후 구현
2. 생성된 언리얼 오브젝트는 강제로 지우려 하지 말 것
   - 참조를 끊는다는 생각으로 설계할 것.
   - 콘텐츠 제작에서 Destroy함수를 사용할 수 있으나, 결국 내부 동작은 동일하다.
     (이거도 플래그 설정하고 나중에 가비지컬렉터가 가져가는 방식이거든)

---

[2] 실습

`UPROPERTY`가 붙은 오브젝트, 안붙은 오브젝트를 하나씩 만든다. 3초 후에 가비지 컬렉터가 스윕한 후에, 각 포인터들은 어떤 형태로 남아있을지 확인하는 예제다

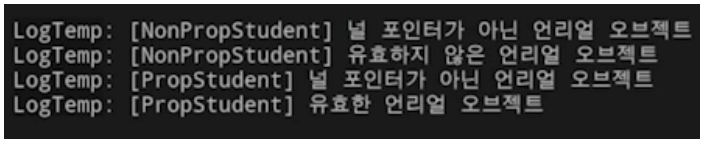
```
#include "CoreMinimal.h"
#include "Engine/GameInstance.h"
#include "MyGameInstance.generated.h"

UCLASS()
class UNREALMEMORY_API UMyGameInstance : public UGameInstance
{
  GENERATED_BODY()

public:
  virtual void Init() override;
  virtual void Shutdown() override;

private:
  TObjectPtr<class UStudent> NonPropStudent;
  UPROPERTY()
  TObjectPtr<class UStudent> PropStudent;

  TArray<TObjectPtr<class UStudent>> NonPropStudents;
  UPROPERTY()
  TArray<TObjectPtr<class UStudent>> PropStudents;

  class FStudentManager* StudentManager = nullptr; //c++클래스이기에 UPROPERTY를 사용할 수 없고 그래서 초기화를 직접 해줘야한다.
};
```

```
include "MyGameInstance.h"
#include "Student.h"
#include "StudentManager.h"

void CheckUObjectIsNull(const UObject* InObject, const FString& InTag)
{
  if (nullptr == InObject)
  {
    UE_LOG(LogTemp, Log, TEXT("[%s] 널 포인터 언리얼 오브젝트"), *InTag);
  }
  else
  {
    UE_LOG(LogTemp, Log, TEXT("[%s] 널 포인터가 아닌 언리얼 오브젝트"), *InTag);
  }
}
void CheckUObjectIsValid(const UObject* InObject, const FString& InTag)
{
  if (InObject->IsValidLowLevel())
  {
    UE_LOG(LogTemp, Log, TEXT("[%s] 유효한 언리얼 오브젝트"), *InTag);
  }
  else
  {
    UE_LOG(LogTemp, Log, TEXT("[%s] 유효하지 않은 언리얼 오브젝트"), *InTag);
  }
}


void UMyGameInstance::Init()
{
  Super::Init();

  NonPropStudent = NewObject<UStudent>();
  PropStudent = NewObject<UStudent>();

  NonPropStudents.Add(NewObject<UStudent>());
  PropStudents.Add(NewObject<UStudent>());

  StudentManager = new FStudentManager(NewObject<UStudent>());
```

```
}

void UMyGameInstance::Shutdown()
{
  Super::Shutdown();

  //check UObject
  CheckUObjectIsNull(NonPropStudent, TEXT("NonPropStudent"));
  CheckUObjectIsValid(NonPropStudent, TEXT("NonPropStudent"));
  CheckUObjectIsNull(PropStudent, TEXT("PropStudent"));
  CheckUObjectIsValid(PropStudent, TEXT("PropStudent"));

  //check TArray UObject
  CheckUObjectIsNull(NonPropStudents[0], TEXT("NonPropStudents"));
  CheckUObjectIsValid(NonPropStudents[0], TEXT("NonPropStudents"));
  CheckUObjectIsNull(PropStudents[0], TEXT("PropStudents"));
  CheckUObjectIsValid(PropStudents[0], TEXT("PropStudents"));

  //check c++ object
  const UStudent* StudentInManager = StudentManager->GetStudent();
  delete StudentManager;
  StudentManager = nullptr;
  CheckUObjectIsNull(StudentInManager, TEXT("StudentInManager"));
  CheckUObjectIsValid(StudentInManager, TEXT("StudentInManager"));
}
```

💡 `IsValidLowLevel`: nullptr의 여부도 확인한다. `IsValid`보다 좀더 깊게 파악한다고 보면된다.

UObject 체크 코드 작동 결과



둘다 `nullptr`은 아니라고 나오지만, Nonprop은 유효하지 않고 Prop은 유효하다고 나왔다.

> ⇒ 즉, `nullptr`만 보고 섣불리 판단하면 **댕글링포인터** 문제가 생긴다는 걸 보여주는 예시다.

자료구조 안의 오브젝트는 어떠한지 알기 위해 TArray에 넣은 형태도 실험해보았지만, `UPROPERTY`의 여부에 따라 위와 같은 결과가 나온다.

C++오브젝트 체크코드

다음은 일반 C++코드에서 언리얼 오브젝트를 만드려면 어떻게 해야되는지 알아보자.

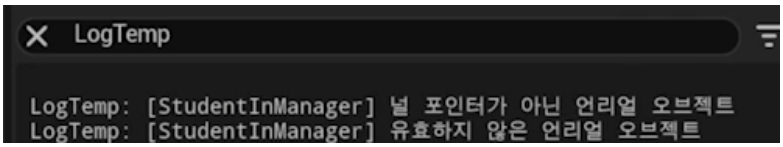
```
#include "CoreMinimal.h"

class UNREALMEMORY_API FStudentManager : public FGCObject
{
public:
  FStudentManager(class UStudent* InStudent) : SafeStudent(InStudent) {}
  const class UStudent* GetStudent() const { return SafeStudent; }

  // 오브젝트 등록을 위한 추가코드
  virtual FString GetReferencerName() const override
  {
    return TEXT("FStudentManager");
  }
  virtual void AddReferencedObjects(FReferenceCollector& Collector) override;
```

```
private:
  class UStudent* SafeStudent = nullptr;
};
```

```
#include "StudentManager.h"
#include "Student.h"

void FStudentManager::AddReferencedObjects(FReferenceCollector& Collector)
{
  if (SafeStudent->IsValidLowLevel())
  {
    Collector.AddReferencedObject(SafeStudent);
  }
}
```

1. 소멸자는 일부러 삭제한다.
2. 오브젝트명에 F를 붙여 언리얼 규칙과 같이 만들어주고, 생성자를 만들어준다.

   ⇒ FStudentManager
3. 만든 C++클래스를 MyGameInstance에서 NewObject로 생성한다.
4. shutdown 단계에서 delete를 통해 직접 삭제해주고, StudentManager의 내부 오브젝트인 SafeStudent는 어떻게 되는지 확인해보았다.



> ⇒ Nonprop과 같은 결과가 나왔다. C++클래스에는 `UPROPERTY`를 사용할 수 없었기에 내부 오브젝트를 회수로부터 **방지할 능력이 없었고**, 그 결과 C++클래스의 회수와 **함께 내부 오브젝트도 회수**되었지만 포인터는 여전히 해당 주소를 가르키는 댕글링 포인터 문제가 발생한 것이다.

💡 위 문제를 해결하기 위해선 `FGCObject`를 상속받아서 `AddReferebcedIbhect()`와 `GetReferenceName()`을 사용해 리플렉션 시스템에 등록함으로써 해결할 수 있다.

📌 Summary

- C++언어의 고질적인 포인터 문제의 이해
- 이를 해결하기 위한 가비지 콜렉션의 동작원리 이해와 설정방법
- 다양한 상황에서 언리얼 오브젝트를 생성하고 메모리에 유지하는 방법 이해
- 언리얼 오브젝트 포인터를 선언하는 코딩 규칙의 이해

  ⇒ 왠만하면 UPROPERTY쓰면된다

0n. 언리얼 엔진의 메모리 관리 강의 과제

❓ **Q1. C++언어에서 포인터 사용시 발생하는 문제점, 실수에 대해 정리하시오.**

❓ **Q2. C++ STL에서 제공하는 스마트 포인터를 조사하고 이들의 장점에 대해 정리하시오.**