유니티 최적화, 로딩, 관리, 리소스

**로딩시간 최적화**

프로파일러를 적극 활용하는 것이 정답

* 에디터의 프로파일러 탭을 보면 함수들의 수행시간이 표시된다.

시간을 많이 잡아먹는 부분을 집중적으로 수정하면 됨.

* + 불필요한 물리 오브젝트 제거하기

Find류의 함수를 주의

* Update 등 매프레임마다 호출되는 곳에서는 Find같은 검색함수는 사용하지 않는 것이 좋다. 초기화 할 때 미리 레퍼런스를 갖고 있는 것을 추천

릴리즈 전 Debug.Log 지우기

* 로그를 남기는 것 자체가 프레임에 영향을 준다.

유니티 로딩 시스템 이해

* 아이폰이나 안드로이드용 유니티 어플을 실행해보면 다음과 같은 순서로 진행된다.

어플 실행 -> 유니티 로고 출력 ->게임 화면

“유니티 로고 출력” 시 씬에 올라와 있는 오브젝트들을 모두 로딩 한다.

이때 로고 이미지 한장만 랜더링 할 수 있는 상태이기 때문에 로딩 진행상황등을 표시하기가 어렵다. 따라서 이부분을 빨리 지나가게 한 뒤 게임 화면에 들어왔을 때 별도의 로딩화면을 만들어 동적으로 로딩 하도록 하는 시스템을 갖추는 것이 좋다.

[ 최적화의 시작은 병목 파악부터 ]

CPU

너무 많은 DP CALL

복잡할 스크립트나 물리 연산

Vertex Processing

너무 많은 버텍스들

버텍스당 너무 많은 연산

Fragment Processing

너무 많은 픽셀, 오버 드로우

프래그먼트 당 너무 많은 연산

BandWidth

크고, 압축되지 않는 텍스쳐

고해상도 프레임 버퍼

[ 스크립트 최적화 ]

**# 유니티의 핵심 기능은 모두 C++로 제작되어 있다.**

**# 예) Transform.position 에서 Transform은 C# 속성, Position은 C++ 영역**

**# 유니티 객체들을 멤버 변수에 저장해서 캐싱하여 사용하는 것이 좋다.**

**# FindObject 계열 함수들은 매우 느리다. (미리 찾아서 캐싱)**

 - 예를 들어 Find 라는 검색이 붙은 것들은 웬만하면 지속적으로 사용하지 않는 것이 좋습니다. Find는 프로젝트 안의 모든 오브젝트를 순환하며 그 안의 클래스의 스트링을 비교하기 때문입니다.

**# Instanitate 와 Destory 함수를 이용한 프리팹의 생성/해제는 비용이 크다**

 - 활성화/비활성화를 활용한 오브젝트 풀을 사용하는 것이 좋습니다.

**# Update 함수보다는 Coroutine을 활용한다.**

**# 박싱과 언박싱은 부하가 큰 작업이다.**

 - C# 박싱과 언박싱에 관한 자세한 내용은 <http://vallista.tistory.com/entry/C-%EB%B0%95%EC%8B%B1%EA%B3%BC-%EC%96%B8%EB%B0%95%EC%8B%B1> 아주 간단한 설명은 제 블로그 내용인 <http://loadofprogrammer.tistory.com/79> 을 참조하시면 됩니다.

**# 나눗셈보다는 곱셈이 몇십 배 빠르다.**

 - 나눗셈은 곱셈보다 연산 속도가 월등히 느립니다. 100 / 10 이런 식이 아닌 100 \* 0.1을 사용

**# magnitude 보다는 sqrMagnitude를 사용해서 비교한다. (제곱근 계산 x)**

 - Unity 공식 문서에서 쓰여있습니다.

**# 삼각함수의 값은 상수로 저장하고 사용하는 것이 좋다.**

**# 문자열은 readonly 혹은 const 키워드를 사용해 가비지 컬렉션으로부터 벗어나도록 한다.**

출처: <https://202psj.tistory.com/1263> [알레폰드의 IT 이모저모]

**[만흉의 원인 : 가비지 컬렉터]**

가비지 컬렉터(GC)는 언제 일어날지 모릅니다.

**# Mono의 동적 메모리 관리 때문에 메모리 해제를 위해 GC가 자동 호출된다.**

**# GC는 언제 일어날지 모른다.**

**# GC가 일어나면 게임이 멈추는 현상이 발생하게 된다.**

**# 동적 메모리 해제가 가능한 일어나지 않도록 하는 것이 GC 관리의 핵심**

- 우리가 쓰고 있는 MonoBehavior는 메모리 관리에 GC가 자동 호출되도록 설계되어 있습니다. 이 GC가 프로그래머의 입장에서 좋을 수도 있고 그렇지 않을 때도 있습니다. GC가 실행되는 동안 많은 양을 처리하게 되면 렉 현상을 겪게되며 그렇지 않기 위해서는 게임을 만들 때 GC를 고려하여 만들어야합니다. 가비지 컬렉터의 할 일을 줄여주기 위한 방법에 대해 알아보겠습니다.

[1] 무엇이든 동적 생성 및 해제는 굉장히 비용이 큰 작업입니다.

 위에 언급된 바가 있는 오브젝트 풀링 기법을 사용해 메모리를 관리하는 것이 좋습니다.

[2] 오브젝트가 해제되면 다음 과정으로는 GC가 동작되어 렉이 걸릴 수 밖에 없습니다.

 즉 오브젝트를 만들어둔 후 활성화 또는 비활성화를 이용해 사용하도록 하는 것이 좋습니다.

[3] 문자열 병합은 StringBuilder의 Append를 사용하면 좋습니다.

 왜냐하면 string + string은 임시 문자열을 뱉기 때문에 가비지 컬렉션이 일어나는 환경을 제공하기 때문입니다.

[4] foreach 대신에 for를 이용하도록 합니다.

 foreach는 한번 돌리면 24byte의 가비지 메모리를 생성시키게 되며 수많이 돌면 더 많은 메모리를 생성시키게 되므로 for 문을 이용하도록 하는 편이 좋습니다.

[5] 태그 비교에서는 CompareTag()를 사용하도록 합니다.

 객체의 tag 프로퍼티를 호출하는 것은 추가 메모리를 할당하며 복사를 하게됩니다.

[6] 모든 비교문에서 .equals()를 사용하도록 합니다.

 "==" 구문으로 사용하게되면 임시적인 메모리가 남게 되며 가비지 컬렉션이 할 일이 늘게 됩니다.

[7] 데이터 타입은 Class 대신 Struct를 사용하여 만들어 주면 메모리 관리가 된다.

 구조체는 메모리 관리를 Stack에서 하므로 GC에 들어가지 않게 됩니다.

[8] 즉시 해제시에는 Dispose를 수동으로 호출하게 되면 즉시 클린업됩니다.

[9] 임시 객체를 만들어내는 API를 조심해야합니다.

 GetComponents<T>, Mesh, Vertices, Camera.allCameras 등등..

[10] 객체의 변경 사항에 대해 캐싱합니다.

 객체의 이동과 변형에 대한 처리를 캐싱해서 매 프레임당 한번만 처리합니다.

[11] 컴포넌트 참조를 캐싱한다.

 GetComponent()는 한번만 호출하며 객체를 캐싱해서 사용한다.

[12] 콜백 함수중 쓰지 않는 함수는 제거합니다.

 Start(), Update(), OnDestroy() 등.. 비어있어도 성능에 영향을 끼치므로 지워주도록 합니다.

출처: <https://202psj.tistory.com/1263> [알레폰드의 IT 이모저모]

**[리소스 최적화]**

**★ 권장 압축 텍스쳐 사용하기**

**# 아이폰(PowerVR) : PVRCT**

**# 안드로이드(Tegra) : DXT**

**# 안드로이드(Adreno) : ATC**

**# 안드로이드(공통) : ETC1**

**★ 텍스쳐**

**# 텍스쳐 사이즈는 무조건 2의 제곱이어야합니다.**

 - POT(Power of Two)

 - POT가 아닌 경우 POT 텍스쳐로 변환되어 로딩된다.

 - 900 x 900 -> 실제로는 1024 X 1024로 변환

 - 화면 해상도에 맞추어 1280으로 만드는 경우엔 실제 메모리에 2048로 생성됩니다.

**# 텍스쳐 아틀라스를 활용하라.**

 - 텍스쳐 아틀라스로 최대한 묶음

 - UI만이 아니라 같은 재질의 오브젝트들을 묶음

 - 한 화면에 나오는 텍스쳐끼리(UI)

 - 알파가 있는 텍스쳐끼리, 알파가 없는 텍스쳐끼리 묶어야합니다.

 - 압축된 텍스쳐와 밉맵을 사용하자. (대역폭 최적화)

 - 32 bit가 아닌 16 bit 텍스쳐 사용도 상황에 맞게 고려합니다.

**# 텍스쳐 메모리 사용량 프로파일링**

 - 메모리 사용량 로그를 서버에 남기면 프로파일링에 도움이 됩니다.

**★ Mesh**

**# Import시에 언제나 "Optimize Mesh" 옵션 사용**

 - 변환 전, 후 버텍스 캐쉬를 최적화 해줍니다.

**# 언제나 Optimize Mesh Data 옵션을 사용한다.**

 - Player Setting -> Other Settings

 - 사용하지 않는 버텍스 정보들을 줄여준다.

**★ 오디오**

**# 모바일에서 스트레오는 의미 없다.**

 - 모두 92kb, 모노로 인코딩

**# 사운드 파일을 임포트하면 디폴트로 3D 사운드로 설정**

 - 2D 사운드로 변경

**# 압축 사운드 (mp3, ogg), 비압축 사운드 (wav) 구별**

 - 비압축 사운드 : 순간적인 효과음, 이펙트 등..

 - 압축 사운드 : 배경 음악

**★ 폰트 리소스 최적화**

**# Packed Font를 사용**

 - R, G, B, A 채널에 저장하는 기법으로 메모리 용량을 1/4로 절약하도록 합니다.

 - Packed Font는 단점이 너무 많습니다. 일반적으로 글씨에 그림자도 못 넣고 알파도 적용이 안됩니다. NGUI Atlas 적용도 되지 않습니다.

**★ 리소스 기타**

**# ResourceLoadAsync() 함수는 엄청 느리다.**

 - 게임 레벨 로드시에 사용했을 경우, 일반 함수에 비해 수십 배나 더 느립니다.

**[그래픽스 최적화]**

**# Draw Call (DP Call)**

 - "적절한 DP Call은 얼마 정도 인가요?" 일반적으로 100 이하를 추천합니다. 보통 70 ~ 100 정도가 일반적입니다.

 - 하지만 실제로는 케이스 바이 케이스

**# Culling**

 - 가장 빠르게 그리는 방법은 아무것도 그리지 않는 것입니다.

**# 프러스텀(Frustum) 컬링**

 - 각 Layer 별로 컬링 거리를 설정하는 것이 가능합니다.

 - 멀리 보이는 중요한 오브젝트(ex. 성, 산맥..)는 거리를 멀게 설정하고 중요도가 낮은 풀이나 나무 등은 컬링 거리를 짧게 설정합니다.

**# 오클루젼(Occlusion) 컬링**

 - Window -> Occlusion Culling 메뉴에서 설정 가능합니다.

 - 카메라에 보이는 각도의 오브젝트들만 렌더링하는 기법.

**# 오브젝트 통합 (Combine)**

 - 드로우 콜은 오브젝트에 설정된 재질의 셰이더 패스당 하나씩 일어납니다.

 - 렌더러에 사용된 재질의 수만큼 드로우 콜이 발생합니다.

 - Combine (통합)

   [1] 성질이 동일한 오브젝트들은 하나의 메쉬와 재질을 사용하도록 통합

   [2] Script 패키지 - CombineChildren 컴포넌트 제공 (하위 오브젝트를 모두 하나로 통합)

   [3] 통합하는 경우 텍스쳐는 하나로 합쳐서, Texture Atlas를 사용해야된다.

**# Batch**

 - Static Batch

   [1] Edit -> Project Setting -> Player에서 설정합니다.

   [2] 움직이지 않는 오브젝트들은 static으로 설정해서 배칭이 되게 합니다.

   [3] Static으로 설정된 게임 오브젝트에서 동일한 재질을 사용할 경우 자동으로 통합합니다.

   [4] 통합되는 오브젝트를 모두 하나의 커다란 메쉬로 만들어서 따로 저장합니다.(메모리 사용량 증가)

 - Dynamic Batch

   [1] 움직이는 물체를 대상으로 동일한 재질을 사용하는 경우 자동으로 통합합니다.

   [2] 동적 배칭은 계산량이 많으므로 정점이 900개 미만인 오브젝트만 대상이 됩니다.

**# Lighting (라이팅)**

 - 라이트 맵을 사용하자

   [1] 고정된 라이트와 오브젝트의 경우(배경) 라이트 맵을 최대한 활용한다.

   [2] 아주 빠르게 실행됩니다. (Per-Pixel Light 보다 2~3배)

   [3] 더 좋은 결과를 얻을 수 있는 GI와 Light Mapper를 사용할 수 있습니다.

 - 라이트 렌더 모드

   [1] 라이팅 별로 Render Mode : Important / Not Important 설저어 가능

   [2] 게임에서 중요한 동적 라이팅만 Important로 설정 (Per-Pixel Light)

   [3] 그렇지 않은 라이트들은 Not Important로 설정

**# Overdraw**

 - 화면의 한 픽셀에 두 번 이상 그리게 되는 경우 (Fill rate)

   [1] DP Call의 문제만큼이나 Overdraw로 인한 프레임 저하도 중요한 문제

   [2] 특히 2D 게임에서는 DP Call보다 더욱 큰 문제가 됩니다.

 - 기본적으로 앞에서 뒤로 그린다

   [1] Depth testing으로 인해 오버드로우를 방지합니다.

   [2] 하지만 알파 블렌딩이 있는 오브젝트의 경우에는 알파 소팅 문제가 발생합니다.

 - 반투명 오브젝트의 개수의 제한을 건다

   [1] 반투명 오브젝트는 뒤에서부터 앞으로 그려야합니다. -> Overdraw 증가

   [2] 반투명 오브젝트의 지나친 사용에는 주의해야합니다.

 - 유니티 Render Mode를 통해 overdraw 확인이 가능합니다.

**# 유니티 셰이더**

 - 기본 셰이더는 모바일용 셰이더 사용

   [1] 기본 셰이더를 사용할 경우에 모바일용 셰이더를 사용합니다. Mobile -> VertexLit은 가장 빠른 셰이더

 - 복잡한 수학 연산

   [1] pow, exp, log, cos, sin, tan 같은 수학 함수들은 고비용입니다.

   [2] 픽셀별 그런 연산을 하나 이상 사용하지 않는 것이 좋습니다.

   [3] 텍스쳐 룩업 테이블을 만들어 사용하는 방법도 좋습니다.

   [4] 알파 테스트 연산(discard)은 느립니다.

   [5] 기본적인 연산보다는 최적화시키고 간략화시킨 공식들을 찾아서 사용할 수 있습니다.

 - 실수 연산

   [1] float : 32bit - 버텍스 변환에 사용. 아주 느린 성능 (픽셀 셰이더에서 사용은 피함)

   [2] Half : 16bit - 텍스쳐 uv에 적합하며 대략 2배 빠릅니다.

   [3] fixed : 10bit - 컬러, 라이트 계산과 같은 고성능 연산에 적합. 대략 4배 빠릅니다.

 - 라이트 맵을 사용하자

   [1] 위의 라이팅에서 설명한 라이트 맵을 사용합니다.

**[물리엔진 최적화]**

**# Fixed Update 주기 조절**

 - FixedUpdate()는 Update와 별도로 주기적으로 불리며, 주로 물리 엔진 처리

 - 디폴트는 0.02초, 즉 1초에 50번 호출합니다.

 - TimeManager에서 수정이 가능합니다.

 - 게임에 따라 0.2초 정도(혹은 이상)로 수정해도 문제 없습니다.

**# 물리 엔진 설정**

 - Static Object

   [1] 움직이지 않는 배경 물체는 Static으로 설정합니다.

 - 충돌체의 이동

   [1] 리지드 바디가 없는 고정 충돌체를 움직이면 CPU 부하가 발생합니다 - 물리 월드 재구성

   [2] 이럴 경우에 리지드 바디를 추가하고 IsKinematic 옵션을 사용.

 - Maximum Allowed timestep 조정

   [1] 시스템에 부하가 걸려 지정된 시간보다 오래 걸릴 경우, 물리 계산을 건너 뛰는 설정

 - Solver Iteration Count 조정

   [1] 물리 관련 계산을 얼마나 정교하게 할지를 지정합니다. (높을수록 정교함)

   [2] Edit -> Project Setting -> Physics

 - Sleep 조절

   [1] 리지드바디의 속력이 설정된 값보다 작을 경우 휴면 상태에 들어갑니다.

   [2] Physics.Sleep() 함수를 이용하면 강제 휴면 상태를 만들 수 있습니다.

**# 2D 물리 vs 3D 물리**

 - 2D 게임에는 2D 물리를 (RigidBody2D), 3D 게임에는 3D 물리를(RigidBody)

**# 물리 엔진 스크립트**

 - 래그돌 사용을 최소화합니다.

   [1] 랙돌은 물리 시뮬레이션 루프의 영역이 아니기 때문에 꼭 필요할 때만 활성화합니다.

 - 태그 대신 레이어 사용

   [1] 물리 처리에서 레이어가 훨씬 유리합니다. 성능과 메모리에서 장점을 가집니다.

 - 메쉬 콜라이더는 절대 사용하지 않는다.

 - 레이캐스트와 Sphere Check 같은 충돌 감지 요소를 최소화합니다.

**# Tilemap Collision Mesh**

 - 2D 게임에서 타일 맵의 Collision Mesh를 최적화하라

   [1] Tilemap을 디폴트로 사용해서 각 타일별로 충돌 메쉬가 있는 경우 물리 부하가 커집니다.

   [2] 연결된 Tilemap을 하나의 Collision Mesh로 물리 연산을 최적화 합니다.

출처: <https://202psj.tistory.com/1263> [알레폰드의 IT 이모저모]