UNITY 3D
-성능 최적화-

SOUL SEEK

메모리와 리소스 최적화

메모리와 리스소 최적화

최적화 이유

- 용량이 50MB를 넘게 되면 다운로드 하는 유저 수가 절반으로 줄어든다
- 메모리를 많이 쓰면 게임유저 수에 제약이 생긴다(최소 사양이 있는 이유)
- 최적화를 하지 않을 시 발생하는 오버헤드는 일반 APP와 비교가 되지 않을 정도로 크다

1. 스크립트 연산 최적화

- 유니티 GameObject를 찾지 말고 캐싱하라.
  - FindObject 계열 함수들은 매우 연산이 느리다. 미리 캐싱 후 사용하자.
- Object Fooling 기법을 사용하라.
  - Instanitate와 Destroy함수를 이용한 생성/해제를 최소화 하자.
- Update Func 보다는 Coroutine을 활용하자.
  - 매 프레임 마다 호출되는 Update보다 Coroutine을 사용하자.
- 문자열을 연결할 땐 StringBuilder를 쓰자.
  - String + String 대신 StringBuilder.Append()함수를 사용하자.
- 나누기 10보단 곱하기 0.1을 사용하자.
  - 나눗셈보다 곱셈의 연산 속도가 몇 십 배 빠르다.
- 가비지 컬렉션에서 벗어나자.
  - 문자열은 readonly와 const를 사용하여 최대한 발생 안되게 하자.
- 객체 캐싱을 활용하자.
  - Component 참조 GetComponent() 함수는 한번만 호출하여 객체를 Cashing해서 사용하자.
- 빈 Callback 함수는 제거
  - 사용하지 않는 Start(), Update()같은 Callback Func는 비어있어도, 성능에 영향을 끼친다.

메모리와 리스소 최적화

2. 리소스 최적화

- 플랫폼 별 텍스처 압축 방식에 따른 메모리 관리.

메모리 사용량 = 가로 픽셀 * 세로 픽셀 * 압축방식의 메모리 사용량(bpp)

- 눈에 보이는 것 혹은 중요 리소스의 경우 압축하지 않고, 잘 보이지 않거나 비중이 낮은 것은 압축을 많이 한다.

[standalone & WebGL]

압축 방식	메모리 사용량 (bytes/pixel)
RGB crunched DXT1	variable
RGBA Crunched DXT5	variable
RGB Compressed DXT1	0.5 bpp
RGBA Compressed DXT5	1 bpp
RGB 16bit	2 bpp
RGB 24bit	3 bpp
Alpha 8bit	1 bpp
RGBA 16bit	2 bpp
RGBA 32bit	4 bpp

[iOS]

압축 방식	메모리 사용량 (bytes/pixel)
RGB Compressed PVRTC 2 bits	0.25 bpp (bytes/pixel)
RGBA Compressed PVRTC 2 bits	0.25 bpp
RGB Compressed PVRTC 4 bits	0.5 bpp
RGBA Compressed PVRTC 4 bits	0.5 bpp
RGB 16bit	2 bpp
RGB 24bit	3 bpp
Alpha 8bit	1 bpp
RGBA 16bit	2 bpp
RGBA 32bit	4 bpp

[Android]

압축 방식	메모리 사용량 (bytes/pixel)
RGB Compressed DXT1	0.5 bpp (bytes/pixel)
RGBA Compressed DXT5	1 bpp
RGB Compressed ETC1	0.5 bpp
RGB Compressed PVRTC 2 bits	0.25 bpp
RGBA Compressed PVRTC 2 bits	0.25 bpp
RGB Compressed PVRTC 4 bits	0.5 bpp
RGBA Compressed PVRTC 4 bits	0.5 bpp
RGB 16bit	2 bpp
RGB 24bit	3 bpp
Alpha 8bit	1 bpp
RGBA 16bit	2 bpp
RGBA 32bit	4 bpp

메모리와 리스소 최적화

2. 리소스 최적화

1. 디바이스별로 권장하는 압축 텍스쳐 포맷
   - iOS(powerVR) : PVRCT, Android(Tegra) : DXT, Android(Adreno) : ATC, Android(공통) : ETC

2. 이미지 가로세로 사이즈는 무조건 2의 제곱
   - 원래 이미지의 크기보다 큰 사이즈의 2의 제곱크기로 변환해서 다시 메모리에 저장하기 과정을 거치기 때문에 UI버튼이나 특정 사이즈와 비율이 유지되어야 하는 이미지가 아니라면 지키자

3. 압축된 텍스처와 MipMap 활용
   - Mipmap을 사용하여 랜더링 속도를 향상시키자.

4. 오디오는 92kb Mono Incoding, 압축형식은 .WAV
   - 모바일에서 스테레오는 의미가 없으므로 92k의 Mono로 Incodeing하자.
   - 억지로 압축해서 손실을 가질 필요없이 유니티 내부 인코더를 활용하자.

메모리와 리스소 최적화

3. 캐싱 활용법

- 캐싱은 자주 사용하는 데이터를 디스크나 메모리에 저장해두는 기법.
- 페이스북이나 카카오톡과 게임을 연동하게 되면 프로필 이미지를 자주 불러오게 된다.
- 다운로드 할 이미지가 디스크에 있는지 체크 (void LoadProfile)
- 디스크에 없으면 인터넷에서 이미지를 받아서 디스크에 저장(IEnumerator DownloadProfile)
- 예제 첨부 – 이런식으로 쓸 수도 있다.

4. Game-Stats로 항목체크

- **Time per frame and FPS**
  - 하나의 프레임을 처리 및 렌더링 하는데 걸린 시간 및 상호 관계에 있는 프레임 / 초
- **Batches(배칭)**
  - 여러 오브젝트를 메모리 덩어리로 결합시키는 작업
- **Saved by baching(결합된 배칭)**
  - 결합된 배칭의 수, 가능한 많은 매터리얼을 공유 하도록 유도하는 것이 좋다.
- **This and Verts(삼각형의 정점 수)**
  - 그려진 삼각형과 정점의 수
- **Screen**
  - 화면 크기, 안티 앨리어싱 레벨 및 메모리 사용량
- **SetPass**
  - 드로우 콜 과 같은 용도, 랜더링 패스의 수, 각 패스에 대해서 유니티 런타임은 CPU오버헤드를 가져올 수 있는 새로운 셰이더를 바인딩 한다.
- **Visible Skinned Meshes**
  - 렌더링 스킨 메시 수
- **Animations**
  - 재생 애니메이션 수

메모리와 리스소 최적화

5. Baching

- 3D Object들을 하나의 메모리 덩어리로 만들어서 한 번에 그리도록 도와주는 작업.

1. **Static Baching**
   - 변하지 않는 오브젝트의 인스펙터 탭에서 static 체크박스를 선택함으로써 스태틱 배칭이 일어나도록 설정.
2. **Dinamic Baching**
   - 동적으로 비슷한 재질의 오브젝트를 하나의 연산 단위로 묶는 방식, 3D 오브젝트의 정점수에 영향을 받는다.
   - 비슷한 메테리얼을 사용하는 경우 채택하는 것이 좋다.

6. Overdraw & Shader

1. Overdraw
   - 한 픽셀에 여러 번 그리는 행위
   - 한 픽셀에 여러 번 그리는 행위를 가 많은 만큼 그래픽 부하가 생긴다.
   - 반투명이 필요한 이미지에만 옵션을 줘서 2D 오브젝트 제작을 할 때, Overdraw가 발생하지 않도록 한다.
2. Shader
   - 모바일 빌드를 할 경우 모바일용 Shader를 사용하는 것이 좋다.

PROFILER

- App의 성능을 체크하는데 사용한다.