unity Developer Day

MEXICO 2016

Mobile Optimizations

Corey Johnson

Product Manager of Doom

Goal

Arm you with more possibilities for optimizations that you will be able to utilize

Agenda

- Recognizing Your Performance Bottleneck
- Profiling in and out of Unity
- Optimizing Tips

What do we mean, Performance?

Frametime

- CPU usage (Gamecode, Physics, Skinning, Particles, …)
- GPU usage (Drawcalls, Shader usage, Image effects, …)

Stalls

- Spikes in framerate caused by heavy tasks (e.g. GC.Collect)
- Physics world rebuild due to moved static colliders*

Memory

- Optimizing memory is very important on device
- Avoid GC Hickups by reducing memory activity
- Leak detection

Know Your Bottlenecks

Question: Why are we slow?

Know Your Bottlenecks

Question: Why are we slow?

- CPU or GPU Bound?
  - Physics or Rendering?
  - Update() or FixedUpdate() loop?

Know Your Bottlenecks

Answer: Always start in the same place...

- Profile
- Profile
- Profile

CPU-Heavy Tasks

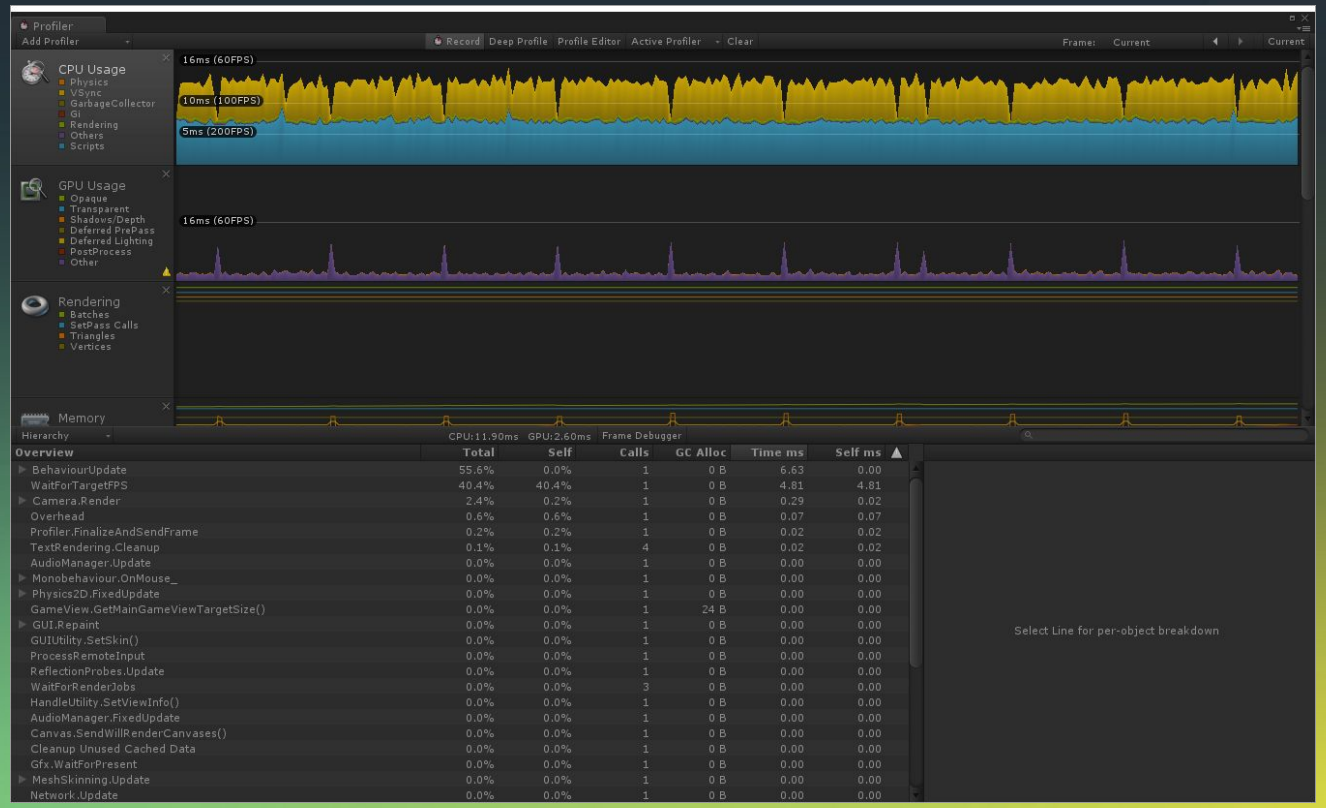
- Physics
- Animation
- Gameplay code
- Runtime GI
- Reflection probes
- Particles
- Creating Batches

GPU-Heavy Tasks

- Switching Batches
- Geometry/Pixel shaders
- Compute shaders
- Skinning

Profiling in Unity

- Unity Profiler
  - In-Editor
  - Live Builds on devices
  - Rapid Iteration
  - Memory usage of individual assets



TIPS:
Use Deep Profile to see calls to all methods (including game code)
Use BeginSample() EndSample() to minimize overhead

Custom Profiler Tags

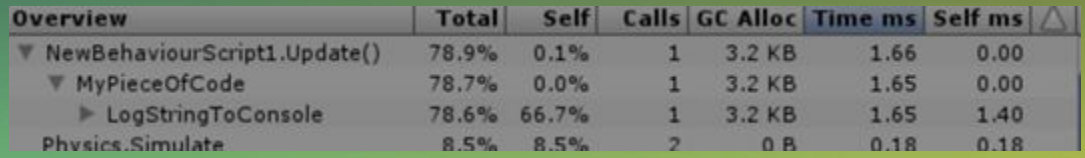
Do this:

```
void Example() {
    Profiler.BeginSample( "MyPieceOfCode" );

    // Do Stuff here

    Profiler.EndSample();
}
```

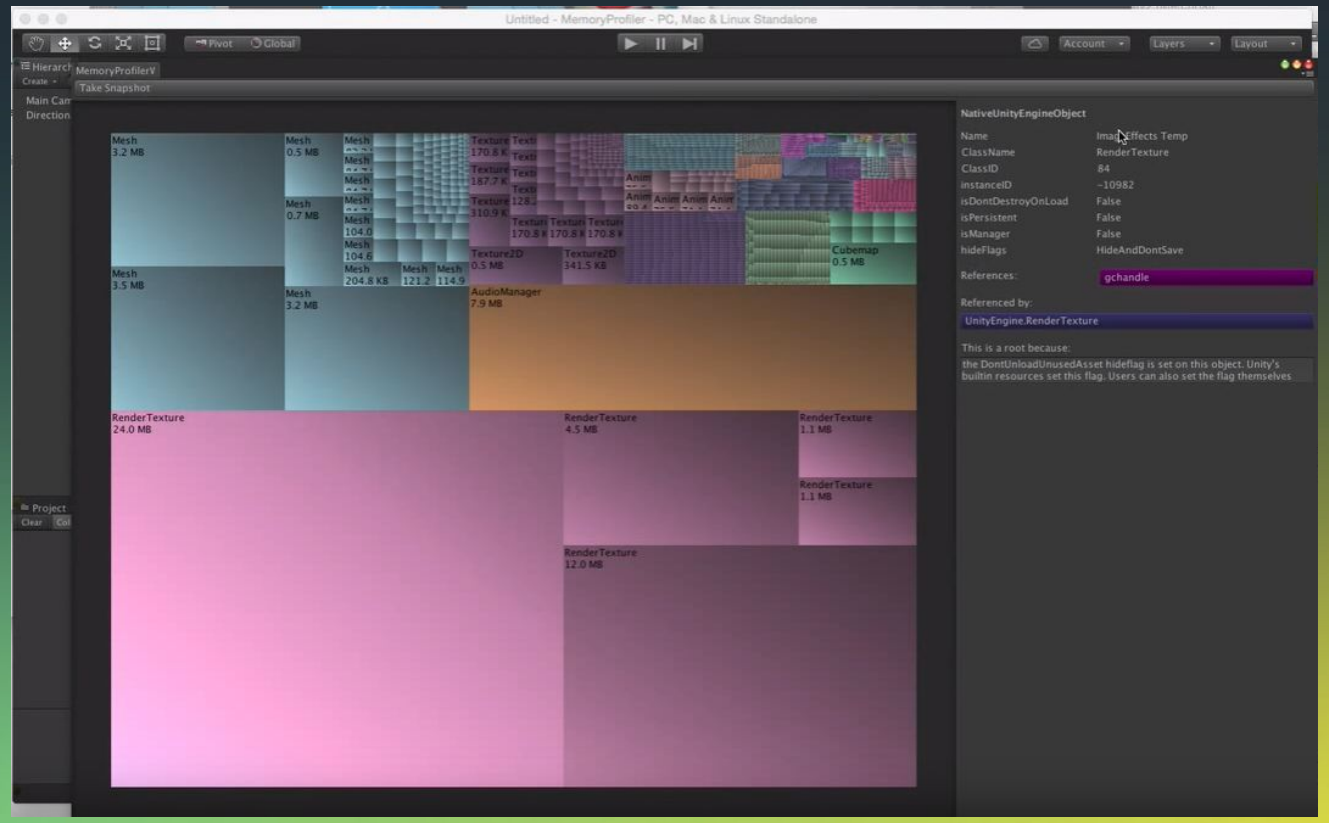
Get This:



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
NewBehaviourScript1.Update()	78.9%	0.1%	1	3.2 KB	1.66	0.00
MyPieceOfCode	78.7%	0.0%	1	3.2 KB	1.65	0.00
LogStringToConsole	78.6%	66.7%	1	3.2 KB	1.65	1.40
Physics.Simulate	8.5%	8.5%	2	0 B	0.18	0.18

Unity Memory Profiler

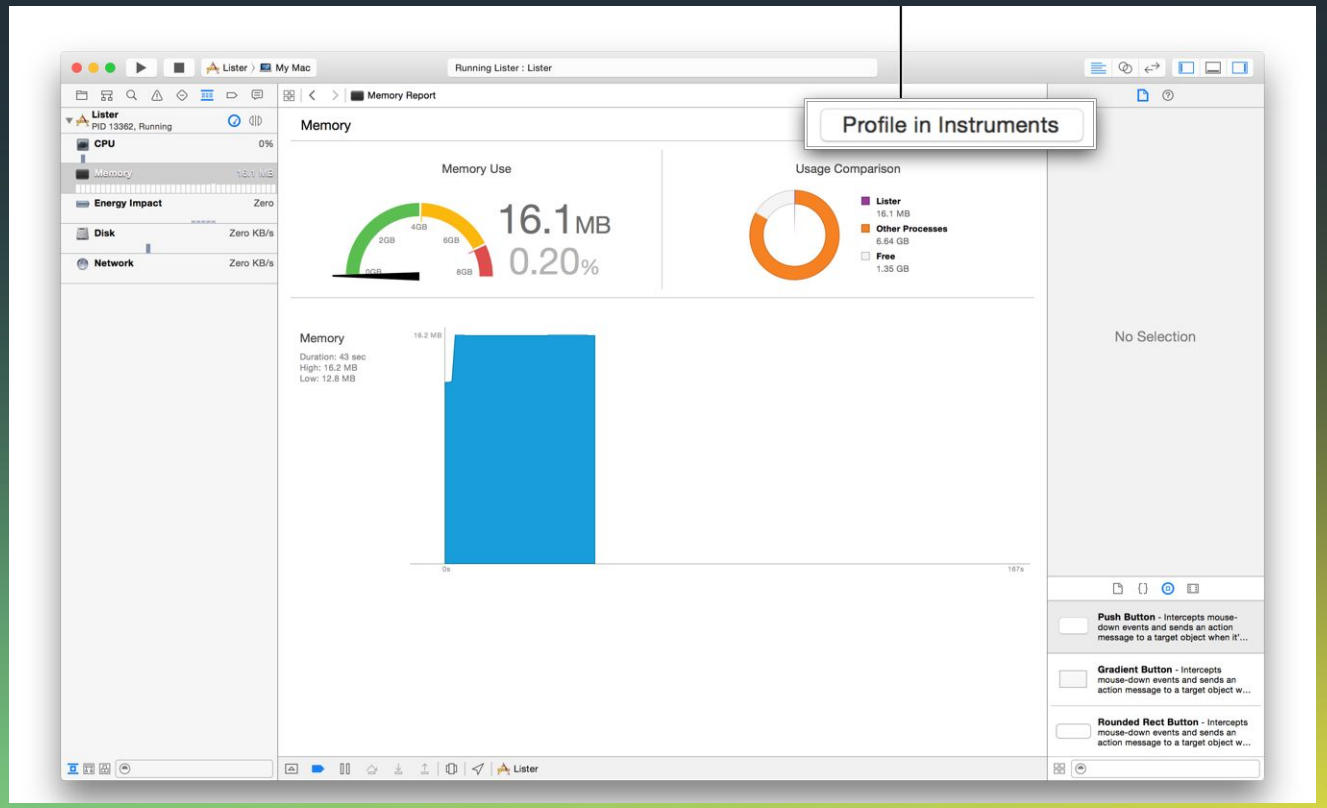
- Open Source https://bitbucket.org/Unity-Technologies/memoryprofiler
  - Profile memory of games running on device



TIPS:
IL2CPP memory info is better than Mono
Under active development

Profiling outside of Unity (iOS)

- Instruments
  - Profile game running on iOS device
  - Mono & IL2CPP Builds



TIPS:
Best for profiling on-device memory usage
Best for determining method CPU usage

Profiling outside of Unity (Android)

- Unity Profiler
  - adb
  - logcat
- GPU
  - Adreno (Qualcomm)
  - PVRTune, PVRUniSCo (PowerVR)
  - Intel GPA

Garbage Collection

- Managed Memory
  - Size doubles when limit is hit
  - NEVER SHRINKS
  - Can stall when collected

- Can explicitly call System.GC.Collect() during breaks in game

Garbage Collection - Stack vs Heap

- Heap Objects
  - Memory block allocated on the Heap and must be Garbage Collected when no longer in use
  - As Heap expands and contains more objects it takes longer for the GC to scan & clean
  - Classes, Strings, Arrays, Lists
- Stack Objects
  - Only live within their scope and memory is freed when it goes out of scope
  - Structs, primitive types

Data Layout Matters

```
struct Stuff {
        int a;
        float b;
        bool c;
        string name;
        }
Stuff[] arrayOfStuff;                   //<< Everything is scanned. GC takes more time
```

VS

```
int[] As;
float[] Bs;
bool[] Cs;
string[] names;                         //<< Only this is scanned. GC takes less time.
```

Object Pooling

- Create a pool of objects to reuse
  - Instantiate as many objects as you'll need before you need them
  - Enable as-needed
  - Disable, Reset when they're done
- No more Instantiate/Destroy cycle (expensive)
- Saves GC from having to run as often
  - No new memory allocated
- Allocate a sensible number of objects
  - Don't allocate TOO many objects as they do take up their own memory in the Heap that can't be reused

Use System.Text.StringBuilder over string

```
string str = “1 allocation” + “ 2 allocations”;
```

- Each string concatenation allocates multiple objects
  - Plus a 3rd for the actual result
  - Problematic if called in loops, Update(), FixedUpdate(), etc
- Use System.Text.StringBuilder
  - .Append() is faster in loops
  - Starts with a capacity, increases when it is surpassed in an Append() call. Then it allocates more memory
- Mecanim: Use Animator.StringToHash() for release
  - Can be used for custom code

More Memory Optimizations

Reuse temporary buffers

- If buffers for data processing are needed every frame, allocate the buffer once and reuse

Don’t use OnGUI

- Even empty OnGUI calls are very memory intensive

Don’t .tag == .tag

- Use CompareTag()

Other GC Optimizations

- for(;;) instead of foreach
  - foreach on anything but arrays allocates an Enumerator (due to old Mono implementation)
- Avoid LINQ functions
  - Allocates memory for Enumerators, essentially a foreach
- Avoid anonymous functions and lambda expressions
  - Allocates memory if needing to access variables outside its scope
- Avoid Boxing value types
  - Converts them to reference types allocated on the Heap

Marshalling Cost

You can write native plugins

- Can be super fast!
- Can be expensive!
- Design plugins carefully to avoid marshalling cost

Can sneak up on you

- gameObject.GetComponent<...>()
- Cache your components

Case Study - Caching

```
public static void ApplyTransform(Matrix4x4[] outputMatrices, Matrix4x4[] inputMatrices)
{
    for(int i = 0; i < inputMatrices.Length; ++i) {
        outputMatrices[i] = Camera.main.worldToCameraMatrix*inputMatrices[i];
    }
}
```

Getting 20k matrices which transform object from local to camera space

Naive implementation: **125 ms**

Case Study - Caching

```
Matrix4x4    worldToCameraMatrix = Camera.main.worldToCameraMatrix;
for(int i = 0; i < inputMatrices.Length; ++i) {
    outputMatrices[i] = worldToCameraMatrix*inputMatrices[i];
}
```

- Cache complex expressions
- Properties can hide expensive operations

Optimized implementation: **33.5ms**

Case study - Copying

```
static void MultiplyMatrices(ref Matrix4x4 result, ref Matrix4x4 lhs, ref Matrix4x4 rhs)
```

Create a method using references

- We had 3 redundant copies (2 inputs, 1 output)
- Matrix4x4 is a value-type

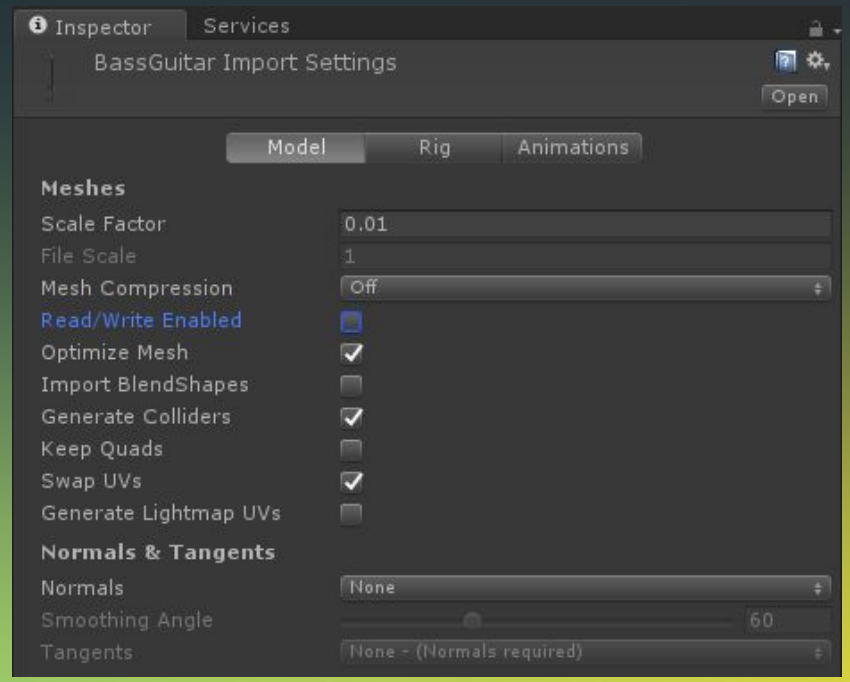
Optimized implementation: **21.5ms**

Optimizing Graphics

- Bake what can be baked
  - Lighting
  - Shadows
- Batch what can be batched
  - Static Meshes
  - Materials
  - UI Canvas elements

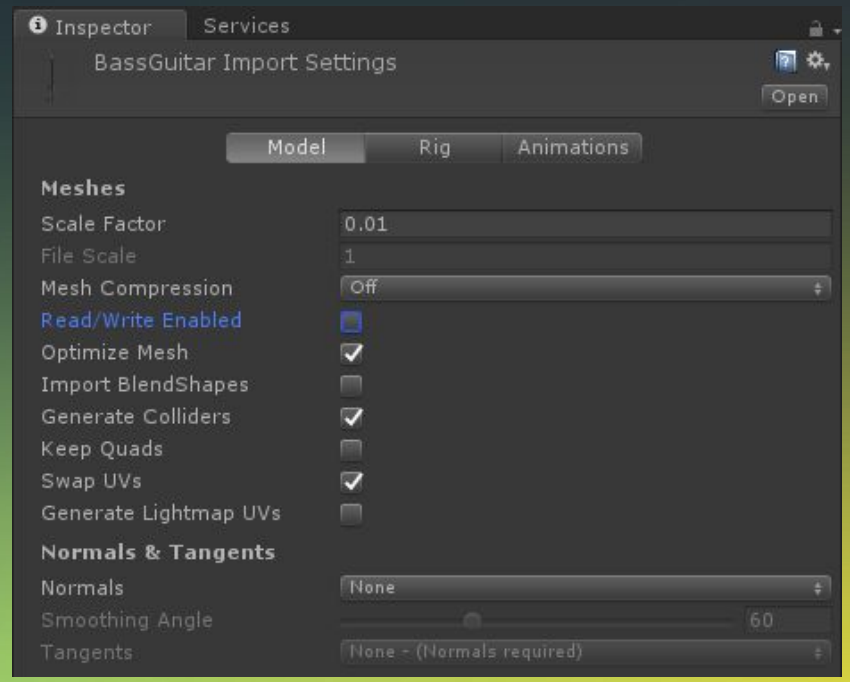
Optimizing Meshes

- Only use as many vertices as you need
- Set “Read/Write” to false if not accessing vertices in script
  - Enabled = extra copy in memory
  - Non-uniform scaling requires read/write
- Enable “Optimize Mesh”
  - Reorder vertex info for fast reading
- Always enable ‘**Optimize Mesh Data**’ in ‘**Player Settings->Other Settings**’
  - Removes redundant vertex attributes (tangents, normal, color, etc)

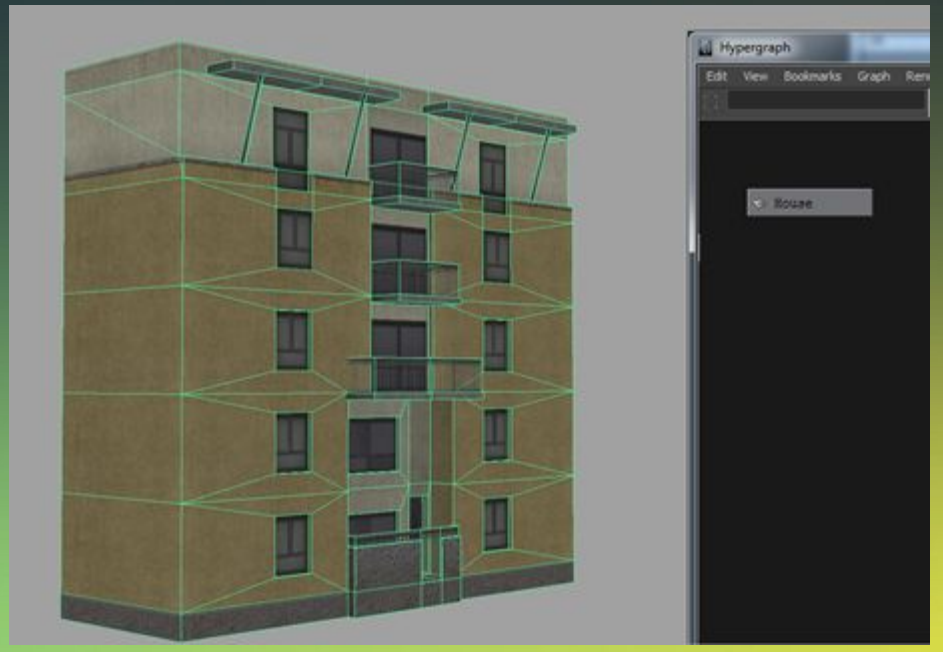


Optimizing Meshes

- Disable “Import Blend Shapes” if none are used
- Disable “Normals and Tangents” if they won’t be used by materials
- Pre-transform static geometry to world space
- Enable Static and Dynamic batching



Combine Meshes



Combine Textures

Texture Atlases can be made by artists too…

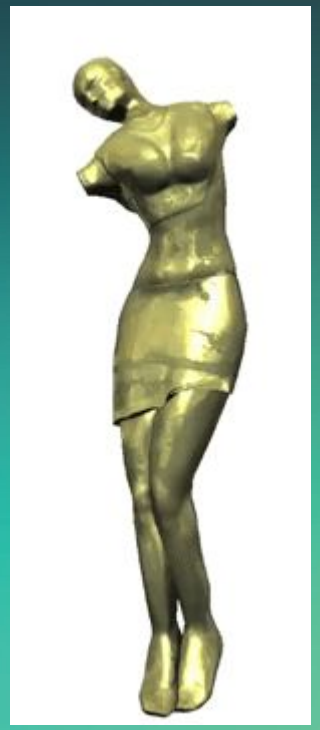
Optimizing Textures

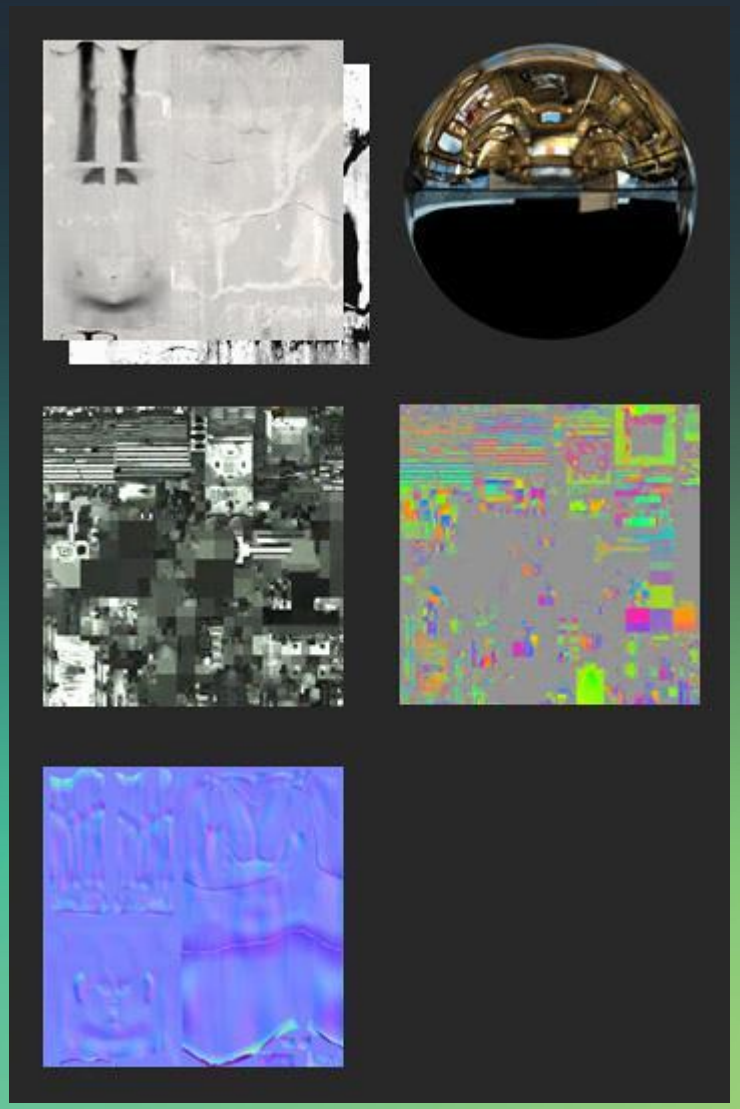
- iOS - Use PVRTC
- Android
  - OpenGLES 2.0 devices: ETC1
  - OpenGLES 3.0 devices: ETC2
  - Specific GPUs might handle other formats more efficiently
- UI - for textures that can’t be compressed without fidelity loss use 16-bit texture instead of 32
- 16-bit Texture Formats
  - Gradient alpha - RGBA4444
  - Only cutout alpha - RGBA5551
  - No alpha - RGB565

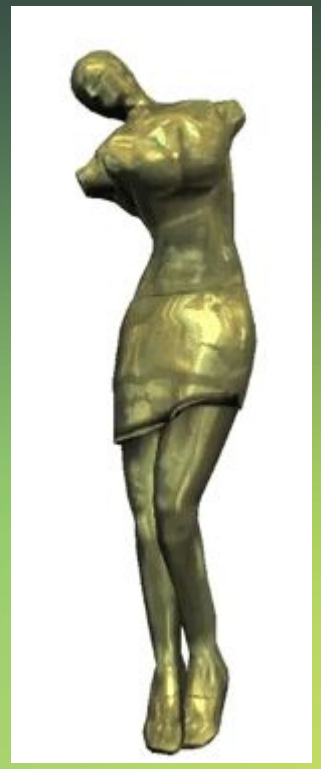
Optimizing Textures - Example

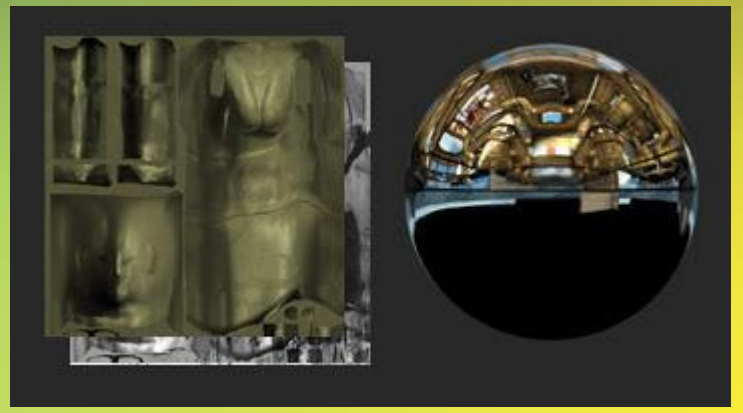
- Shadowgun
- Used “Render to Texel” tool to bake normal-mapped lighting into textures
  - https://www.assetstore.unity3d.com/en/#!/content/4153
  - Saved massive run time calculations

Optimizing Textures - Example



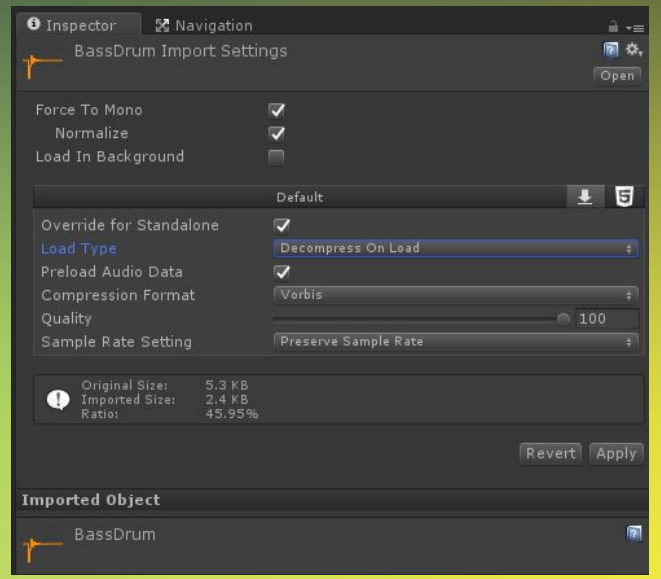






Optimizing Audio

- “Force to Mono” if sounds don’t require 3D/Stereo
- Load Type “Decompress on Load” if clip smaller than 200kb
  - Unity uses 200kb playback buffer when decompressing audio so leave it decompressed. Saves memory when playing the sound.
- “Stream to Disk” for long audio clips
  - Only 1 clip at a time
  - Buffers compressed data
  - Decodes on the fly
  - Uses minimal memory
- “Compressed in Memory” for other clips



Optimizing UI

- Keep UI elements at the same z-depth
  - Different z-depths breaks batching
- Use Sprite Packer
  - Fewer draw calls for Sprites
- Separate UI into several Canvases (but not too many)
  - Batch time grows more than linearly by # of elements to sort, analyze
- Combine UI that doesn’t change
  - Canvas won’t need to be rebatched
- Reduce switching between overlapping Text and Sprites
- Reduce text in UI if possible
  - Text is batched separately from Sprites

Other Optimizations

- Limiting Rigidbodies to 2 dimensions in a 2D game
  - Use Box2D or roll your own
  - Doesn’t pull in whole physics system(s)
- Rigidbodies on projectiles
  - Calculate collision on your own
- Lots of individual 3D objects for collectables or characters
  - Use animated sprites on particles to represent simple objects
- Perform expensive calculations every few frames and cache the results
  - Coroutines (maybe)

Script Optimizations

- Avoid Find…() methods
  - Cache a reference instead
  - FindWithTag() is more optimized but still not as fast
- Use Non-allocating functions
  - i.e. pass array as parameter to fill instead of allocating and returning a new one
  - Unity’s Physics system has examples of non-allocating functions

i.e. Physics2D.RaycastNonAlloc()

public static int RaycastNonAlloc(Vector2 origin, Vector2 direction, RaycastHit2D[] results)

Vector Math Optimizations

- Normalize a vector once if used over and over
  - Normalization function takes longer than just storing and accessing it
- v.normalized slower than v * 1.0/v.length
- Use Vector’s .sqrMagnitude to compare distances instead of getting the actual distance
  - Saves some calculations

Shader Optimization

- In general, less instructions is better*
- Move calculations to Vertex Shader
  - High DPI devices make every pixel count
- Simplify math
  - Trig functions are super expensive
  - Bake into lookup textures
- Reduce temporary registers used
  - Number of shader threads that can work simultaneously depends on this

10000 Objects Update() vs Update() 10000 Objects

- Blog Post - http://blogs.unity3d.com/2015/12/23/1k-update-calls/
  With Sample Project - https://github.com/valyard/Unity-Updates/commits/master
  By Unity's Valentin Simonov
  - Much faster to run a function on 10000 objects from a single manager GameObject's Update() method
    - Due to remaining on the Managed side. Native → Managed call to Update and various safety checks Unity does internally makes Update() on 10000 objects slow

unity Developer Day

GRACIAS