Making of Delta Building Visualization with Compute Shaders

Kristo Männa
Who am I

Hobbyist solo game developer

Unity3D

Project POMPEII

Delta building visualization project
Who am I

Hobbyist solo game developer

Unity3D
Who am I

Hobbyist solo game developer

Unity3D

Project POMPEII
Who am I

Hobbyist solo game developer

Unity3D

Project POMPEII

Delta building visualization project
Topics

What are compute shaders

Introduction to compute shaders (using Unity)

Breakdown of Delta Building Visualization effects
Topics

What are compute shaders

Introduction to compute shaders (using Unity)

Breakdown of Delta Building Visualization effects
Topics

What are compute shaders

Introduction to compute shaders (using Unity)

Breakdown of Delta Building Visualization effects
Topics

What are compute shaders

Introduction to compute shaders (using Unity)

Breakdown of Delta Building Visualization effects
What are compute shaders?

GPU is very good at parallel processing (like really good)

This power is underutilized
What are compute shaders?

GPU is very good at parallel processing (like really good)

This power is underutilized

That's where Compute Shaders come in
What are compute shaders?

GPU is very good at parallel processing (like really good)

This power is underutilized

That's where Compute Shaders come in

Use GPU for other things besides graphics
What are compute shaders?

GPU is very good at parallel processing. This power is underutilized. That's where Compute Shaders come in. Use GPU for other things besides graphics.
How to use a **Compute Shader**

```csharp
struct v2f
{
    float2 uv : TEXCOORD0;
    UNITY_FOG_COORDS(1)
    float4 vertex : SVPOSITION;
};
sampler2D _MainTex;
float4 _MainTex_ST;

v2f vert (appdata v)
{
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);
    UNITY_TRANSFER_FOG(o,o.vertex);
    return o;
}

fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = tex2D(_MainTex, i.uv);
    // apply fog
    UNITY_APPLY_FOG(i.fogCoord, col);
    return col;
}
```
How to use a **Compute Shader**

*a Shader with properties*:

```c
#include "UnityCG.cginc"

struct appdata
{
    float4 vertex : POSITION;
    float2 uv : TEXCOORD0;
};

struct structExample {
    uint a;
    float b;
    float3 c;
};

RWTexture2D<float4> WritableTexture;
RWStructuredBuffer<uint> WritableBuffer;
AppendStructuredBuffer<structExample> AppendableBuffer;
Texture2D<float4> ReadOnlyTexture;
float justAFloat;

[numthreads(1,1,1)]
void CSMain (uint3 id : SV_DispatchThreadID, uint3 SV_GroupThreadID, uint3 SV_GroupID, uint SV_GroupIndex) {
    //Code goes here

    //Save the result somewhere
    WritableBuffer[SV_GroupID.x] = WritableBuffer[SV_GroupID.x] + 1;
}
```

*Note: The code snippet is a placeholder and is not intended for actual execution.*
How to use a **Compute Shader**

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Example : MonoBehaviour {

    public ComputeShader cs;
    ComputeBuffer buffer;

    void SetupShader () {
        elementType = 10;
        elementTypeSize = 4; // Each uint is 4 bytes
        buffer = new ComputeBuffer(elementType, elementTypeSize);

        var array = new uint[elementType];
        for (int i = 0; i < elementType; i++) array[i] = i;
        buffer.SetData(array);
    
    cs.SetBuffer("WritableBuffer", buffer);
    cs.Dispatch(0, array.Length, 1, 1); // Kernel index, dimensions
}

#pragma kernel CSMain

struct structExample {
    uint a;
    float b;
    float3 c;
};

RWTexture2D<float4> WritableTexture;
RWStructuredBuffer<uint> WritableBuffer;
AppendStructuredBuffer<structExample> AppendableBuffer;
Texture2D<float4> ReadOnlyTexture;
float justAFloat;

[numthreads(1,1,1)]
void CSMain (uint3 id : SV_DispatchThreadID, uint3 SV_GroupThreadID, uint3 SV_GroupID, uint SV_GroupIndex) {

    // Code goes here

    // Save the result somewhere
    WritableBuffer[SV_GroupID.x] = WritableBuffer[SV_GroupID.x] + 1;
```
How to use a Compute Shader
How to use a Compute Shader

cs.Dispatch(0, 1, 1, 1);

[numthreads(1, 1, 1)]
How to use a Compute Shader

cs.Dispatch(0,1,1,1);

[numthreads(1,1,1)]
How to use a Compute Shader

```cpp
[ numthreads(8,4,1) ] / [ numthreads(8,8,1) ]

cs.Dispatch(0,1,1,1);
```
When to use compute shaders

A lot of operations that can be done in parallel
When to use compute shaders

A lot of operations that can be done in parallel
When to use compute shaders

A lot of operations that can be done in parallel
When to use **compute shaders**

A lot of operations that can be done in parallel
When to use compute shaders

A lot of operations that can be done in parallel
Quick demo

Batch rendering in unity
Quick demo - Batch rendering in unity

- 100 x 100 = 100 000 objects
- GPU: GTX 1070 Ti
Quick demo - Batch rendering in Unity

- 100k Capsule GameObjects
- 30-40 fps
- Can use Unity’s static batching
- Best approach for lower end GPUs

Standard rendering

- 100k Capsule GameObjects
- 30-40 fps
- Can use Unity’s static batching
- Best approach for lower end GPUs

Statistics

Audio:
- Level: -74.8 dB
- Clipping: 0.0%
- DSP load: 0.2%
- Stream load: 0.0%

Graphics:
- 35.9 FPS (27.9ms)
- CPU: main 27.9ms, render thread 16.4ms
- Batches: 33587
- Saved by batching: 0
- Tris: 27.9M, Verts: 18.5M
- Screen: 1080x805 - 9.8 MB
- SetPass calls: 8, Shadow casters: 13584
- Visible skinned meshes: 0, Animations: 0
GPU Instanced rendering

- 100k Capsule GameObjects
- GPU Instancing enabled shaders
- 50-60 fps
- For lower end about the same performance as Standard
- For Mid-Higher end meaningfully better
Manual Instanced rendering

- 100k Capsule meshes
- Custom shaders
- must call draw from code
- 300-600 fps
- For lower end can be worse than previous methods
- For Mid-Higher end way better
Manual Instanced rendering

- 1 Million quads
- Simpler, smaller mesh improves performance
- Lighter shader improves performance
Delta Building Visualization project (DBV)
DBV effects

- Individually animated crowd rendering
  - Bonus: Bubbles
- 1 million grass object rendering
- Cloud shadows
- Dynamic snow tracks
- Falling snow
DBV effects

- Individually animated crowd rendering
  - *Bonus: Bubbles*
- 1 million grass object rendering
- Cloud shadows
- Dynamic snow tracks
- Falling snow
DBV effects

- Individually animated crowd rendering
  - Bonus: Bubbles
- 1 million grass object rendering
- Cloud shadows
- Dynamic snow tracks
- Falling snow
DBV effects

- Individually animated crowd rendering
  - Bonus: Bubbles
- 1 million grass object rendering
- Cloud shadows
- Dynamic snow tracks
- Falling snow
**DBV effects**

- Individually animated crowd rendering
  - *Bonus: Bubbles*
- 1 million grass object rendering
- Cloud shadows
- Dynamic snow tracks
- Falling snow
**DBV effects**

- Individually animated crowd rendering
  - *Bonus: Bubbles*
- 1 million grass object rendering
- Cloud shadows
- Dynamic snow tracks
- Falling snow
2000 Animated Agent rendering

Requirements:

- max 2000 actors → max visible actors ~1500
- Every actor has animations
- Need to render them all
Naive approach:

1. Have two gameObject pools
   a. Far away actors
   b. Close animated actors
2. Based on distance to camera render actors
   with objects from first or second pool

- Very slow for 50+ animators
2000 Animated Agent rendering

Used approach:

1. Bake animations to a texture
2. Only have one pool for all actors
3. Render visible actors with that pool
Bubbles

Requirements:
- Actors sometimes create bubbles
- Bubbles are animated
- max 500 bubbles at once

Used approach:
- GPU instanced billboard shader
- Bubble pool
- Simple procedural scale animations
1 Million grass object rendering

Requirements:

- There is some grass outside the building
- Would be nice if it looked like real grass
- Need to handle 1 million grass objects
1 Million grass object rendering

Naive approach:

1. Place down GPU instanced grass GameObjects
2. Let unity handle culling and batching

Works okay for sub 10 000 grass objects
...but we need 1 million
1 Million grass object rendering

Used approach:

1. Bake grass object positions
2. Using compute shader apply:
   a. Frustum culling
   b. Distance culling
   c. 3D dither distance culling
3. Render visible with Graphics.DrawMeshInstanced()
1 Million grass object rendering
1 Million grass object rendering
1 Million grass object rendering
Cloud shadows

Requirements:

- Would be nice if could visualize cloudiness somehow

Naive approach:

- Use Unity’s projector component to render cloud shadows onto ground
- Will have to affected objects twice

Used approach:

- Same as naive
- Ok since only one object to re-render
Dynamic snow tracks

Requirements:

- Would be nice if could visualize snow tracks for actors
- Will need to create tracks for 2000 actors every frame
Dynamic snow tracks

Used approach:

1. Send actor positions to compute shader
2. Draw to splatmap
3. Tessellate a plane
4. Use vertex displacement
Dynamic snow tracks

Requirements:
- Would be nice if could visualize snow tracks for actors
- Will need to create tracks for 2000 actors every frame
Dynamic snow tracks
Dynamic snow tracks
Snowflakes

Naive approach:

- Use Unity’s particle system
- Very slow for 1000+ particles since CPU based
Snowflakes

Used approach

- Using compute shader cull and apply velocity to particles
- Use DrawMeshInstanced() to draw visible snowflakes
Conclusion

Learned about:

1. **Compute shaders**
2. How and where to use them
3. Breakdown of some simple DBV effects
Any final questions?
Peace!