Advanced GPU Programing with Unity3D

What is this course about ?

- Overview of Unity3D
- How to use Unity3D for advanced GPU programing
- Overview of a few CG techniques & implementaion in Unity

Course overview

- Unity3D 101
- Introduction to shader programing
- Custom shaders
- Post processing
- Compute shaders
- Volume rendering

Content may be subject to changes

Assessment

- 100 % project-based
- Reimplementation of a CG paper in Unity

(offered topic available soon)

- Deadline for topics 1st of November
- Lab hours every two weeks

Why Using a Game Engine ?

- Universal
- Ease of use
- High level scripting
- No maintenance costs
- Extensive documentation
- Many out-of-the-box features
- Develop once deploy everywhere (in theory)
- …

Why Unity3D?

- Vanilla OpenGL is too cumbersome
- Other game engines are too high-level
- Right balance between flexibility & ease of use for graphics programing

Caution !

Game engines are not perfect all-in-one solutions.

For developing professional softwares or programs requiring heavy CPU computation, Unity3D might not be the best choice.

Highly recommended for prototyping.

What is Unity3D ?

- Unity is a multi-platform, integrated IDE for developing games, and working with 3D virtual worlds
- WYSWYG editor
- Asset manager
- C# Scripting integrated with Visual Studio

Unity Crash Course

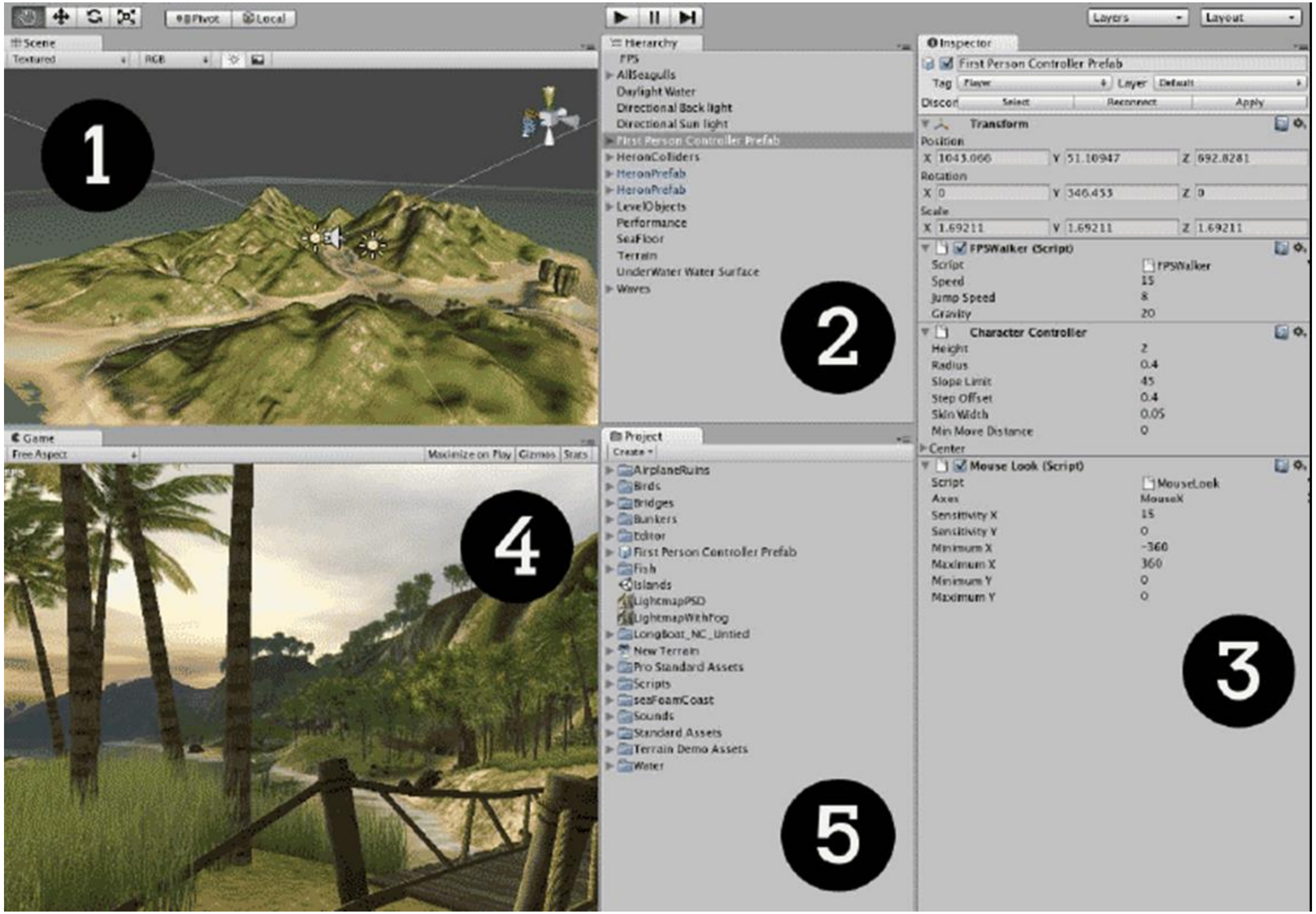
1 – Scene

2 – Hierarchy

3 – Inspector

4 – Game

5 – Project



FPS
AllSeagulls
Daylight Water
Directional Back light
Directional Sun light
First Person Controller Prefab
HeronColliders
HeronPrefab
HeronPrefab
LevelObjects
Performance
SeaFloor
Terrain
UnderWater Water Surface
Waves

☑ First Person Controller Prefab
Tag Player   Layer Default
Discon   Select   Reconnect   Apply

Transform
Position
X 1043.066   Y 51.10947   Z 692.8281
Rotation
X 0   Y 346.453   Z 0
Scale
X 1.69211   Y 1.69211   Z 1.69211

☑ FPSWalker (Script)
Script FPSWalker
Speed 15
Jump Speed 8
Gravity 20

Character Controller
Height 2
Radius 0.4
Slope Limit 45
Step Offset 0.4
Skin Width 0.05
Min Move Distance 0
Center

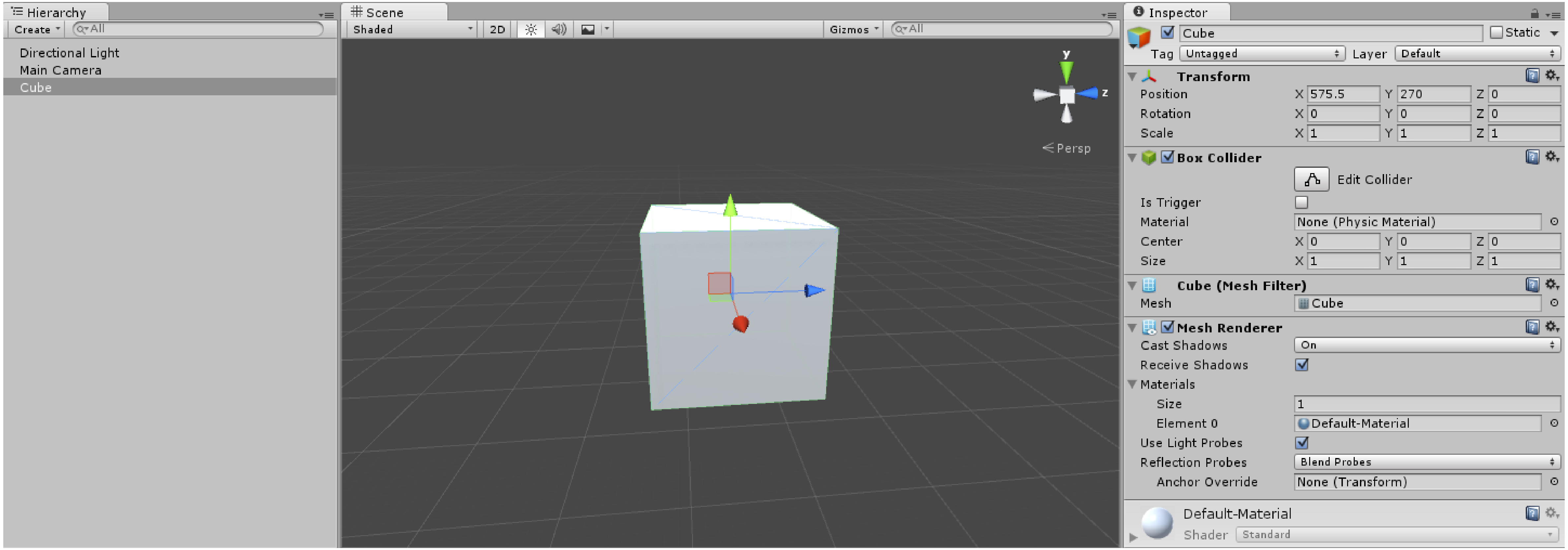
☑ Mouse Look (Script)
Script MouseLook
Axes MouseX
Sensitivity X 15
Sensitivity Y 0
Minimum X -360
Maximum X 360
Minimum Y 0
Maximum Y 0

AirplaneRuins
Birds
Bridges
Bunkers
Editor
First Person Controller Prefab
Fish
Islands
LightmapPSD
LightmapWithFog
LongBoat_NC_Untied
New Terrain
Pro Standard Assets
Scripts
seaFoamCoast
Sounds
Standard Assets
Terrain Demo Assets
Water

Game Objects

- Everything is a Game Object (lights, cameras, characters,…)
- Contains components (mesh, audio, script, physics, etc.)
- Transform component by default
- Game Objects may contain other game objects (placeholders)

Game Object - Cube Example



Directional Light
Main Camera
Cube

☑ Cube □ Static
Tag Untagged Layer Default

Transform

Position X 575.5 Y 270 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

☑ **Box Collider**

Is Trigger □
Material None (Physic Material)
Center X 0 Y 0 Z 0
Size X 1 Y 1 Z 1

Cube (Mesh Filter)

Mesh Cube

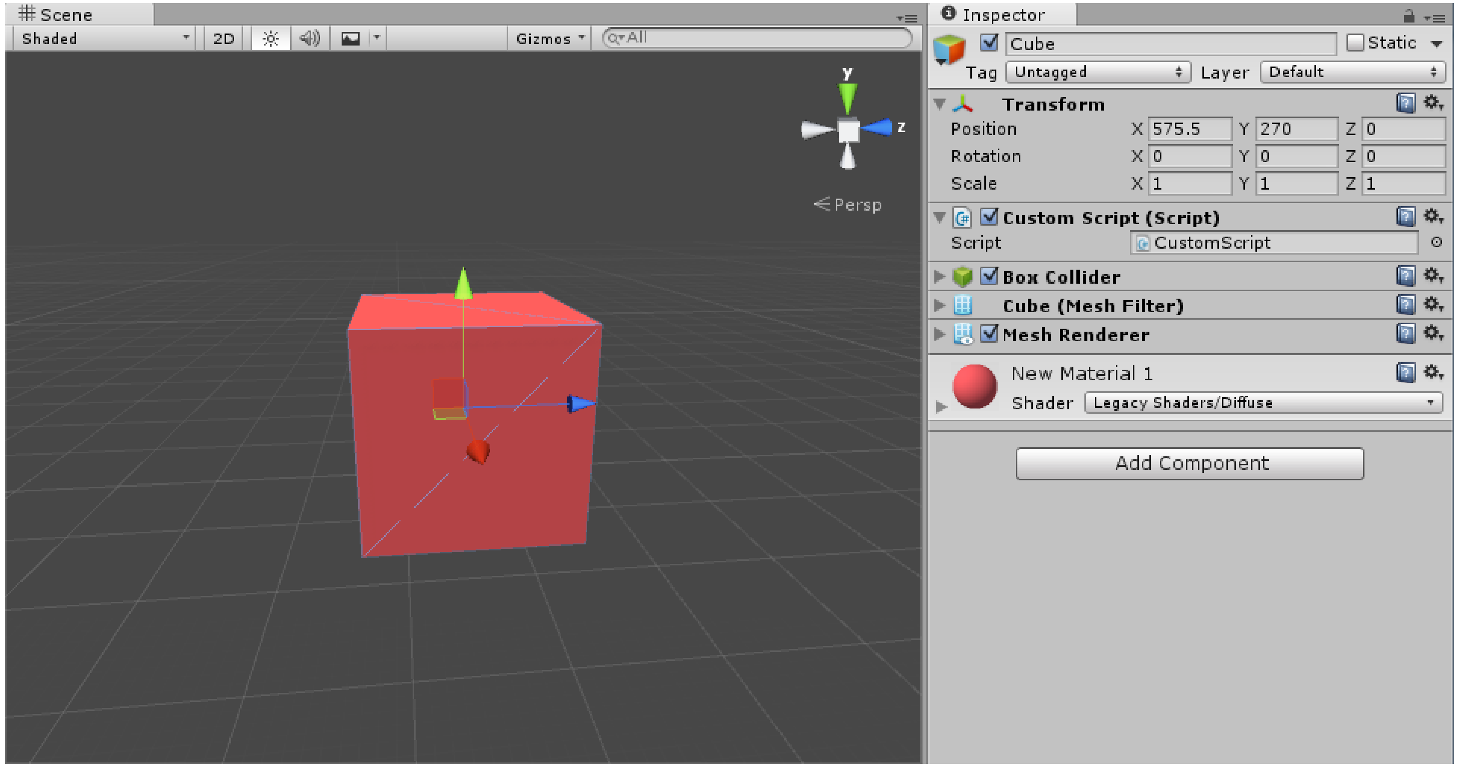
☑ **Mesh Renderer**

Cast Shadows On
Receive Shadows ☑
Materials
Size 1
Element 0 Default-Material
Use Light Probes ☑
Reflection Probes Blend Probes
Anchor Override None (Transform)

Default-Material
Shader Standard

Scripting

- Scripts must be attached to a game object to live
- Some game objects may only contain scripts



☑ Cube □ Static

Tag Untagged Layer Default

Transform

Position	X 575.5	Y 270	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

☑ Custom Script (Script)

Script CustomScript

☑ Box Collider

Cube (Mesh Filter)

☑ Mesh Renderer

New Material 1

Shader Legacy Shaders/Diffuse

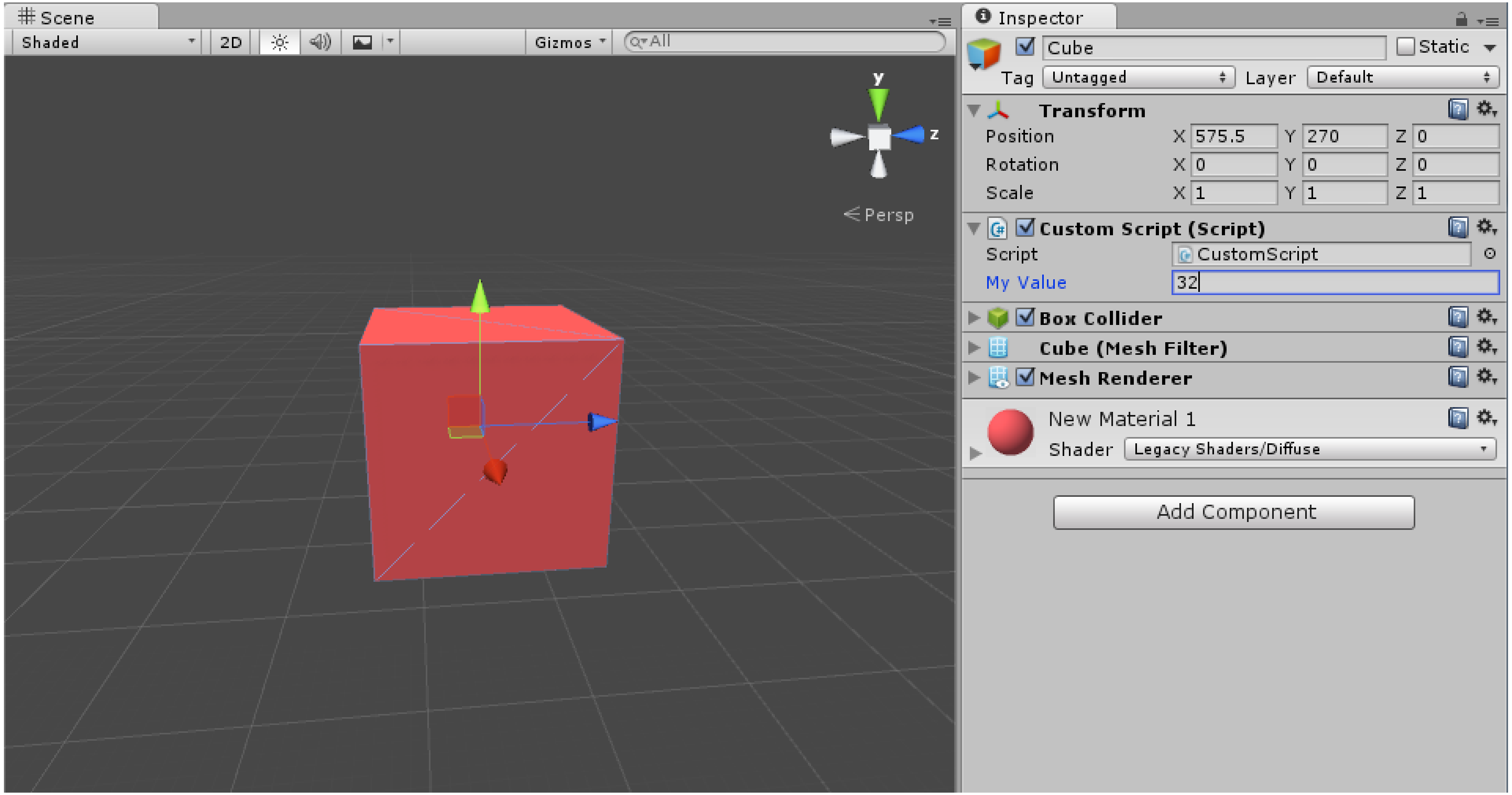
Scripting

```
using UnityEngine;
using System.Collections;

public class CustomComponent: MonoBehaviour // The base class of all components
{
   // Use this for initialization
   void Start ()
   {
   }

   // Update is called once per frame
   void Update ()
   {
   }
}
```

Scripting

```
using UnityEngine;
using System.Collections;

public class CustomComponent: MonoBehaviour // The base class of all components
{
   public int myValue;

   void Start () // Use this for initialization
   {
   }

   // Update is called once per frame
   void Update ()
   {
   }
}
```

Scripting



☑ Cube □ Static

Tag Untagged Layer Default

☑ Transform

Position X 575.5 Y 270 Z 0

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

☑ Custom Script (Script)

Script CustomScript

My Value 32

☑ Box Collider

Cube (Mesh Filter)

☑ Mesh Renderer

New Material 1

Shader Legacy Shaders/Diffuse

Add Component

Scripting

- Useful stuffs

```
this.gameObject; // The reference to the game object
this.transform; // Position, rotation, scale of the game object
this.GetComponent<Type>(); // Get component attached to game object
GameObject.Find(string name); // Find another game object in the scene
```

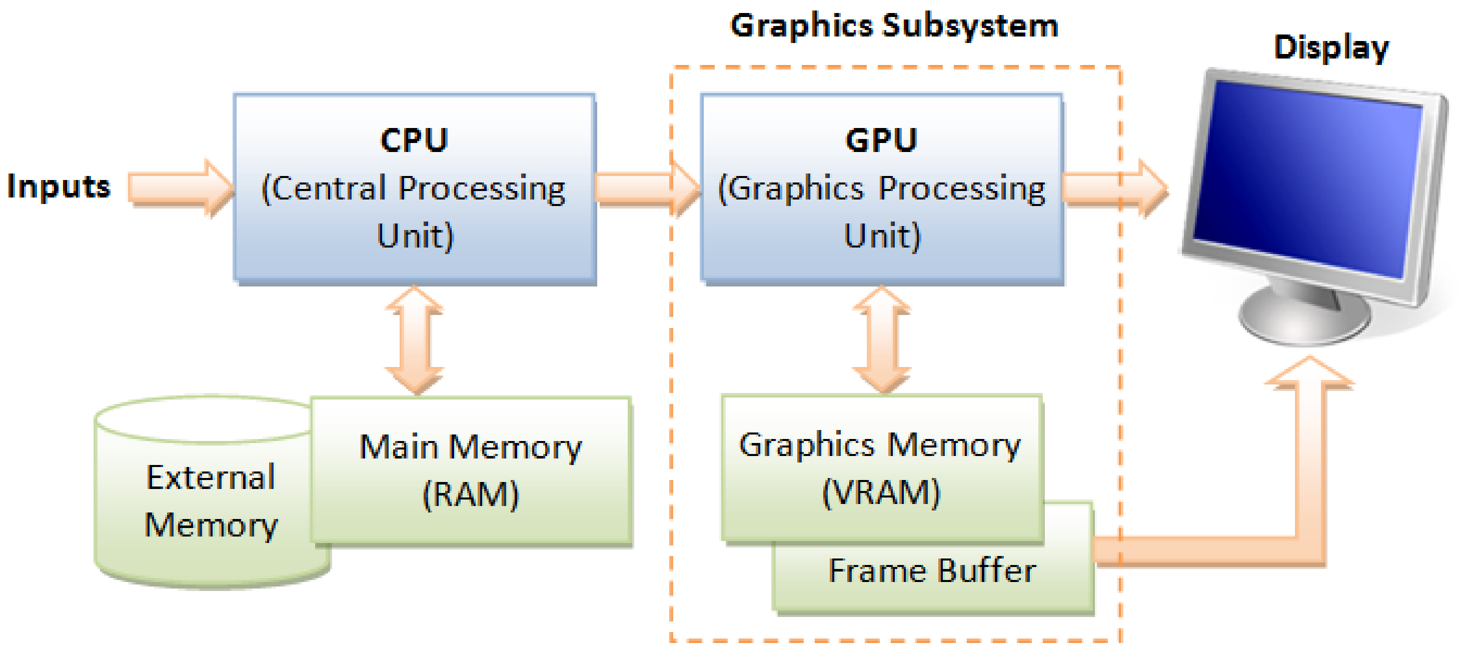
- Useful callbacks

```
void Start () {} // Called when the game starts to play
void Update() {} // Called every frame
void OnDestroy() {} // Called when the game object is destroyed
... many more, check the documentation
```

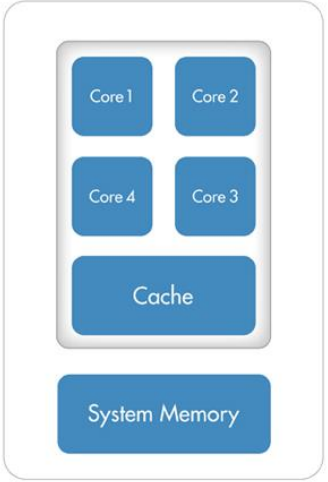
Scripting Demo

Rendering with Unity

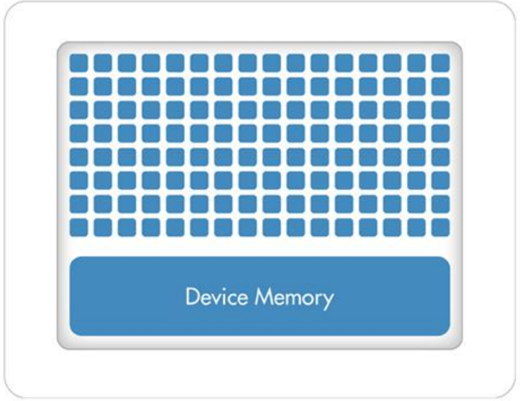
Introduction to Shader Programing



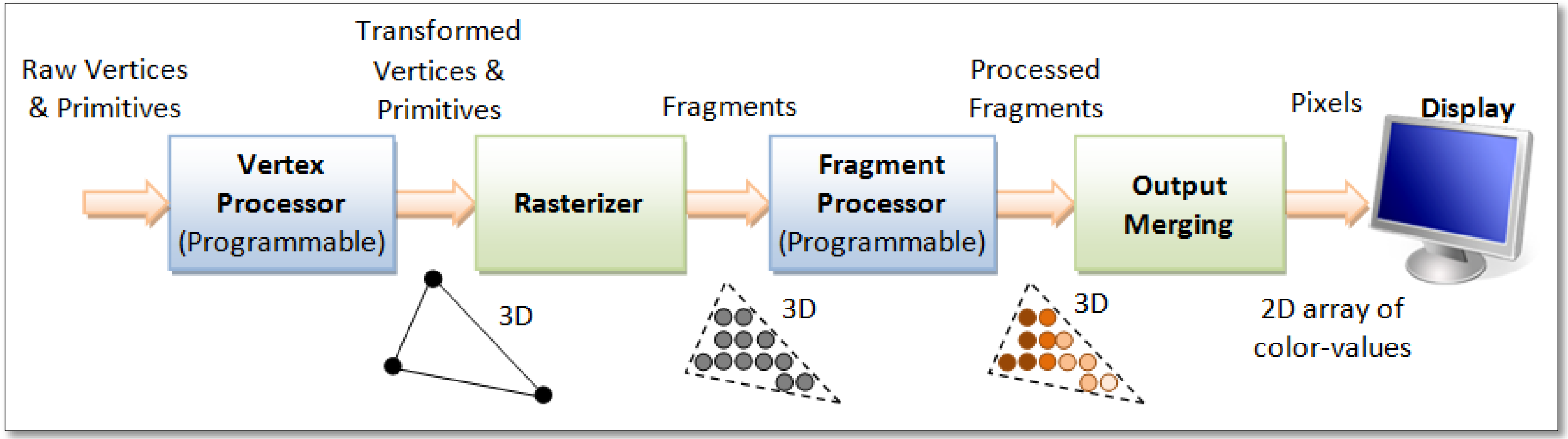
CPU (Multiple Cores)

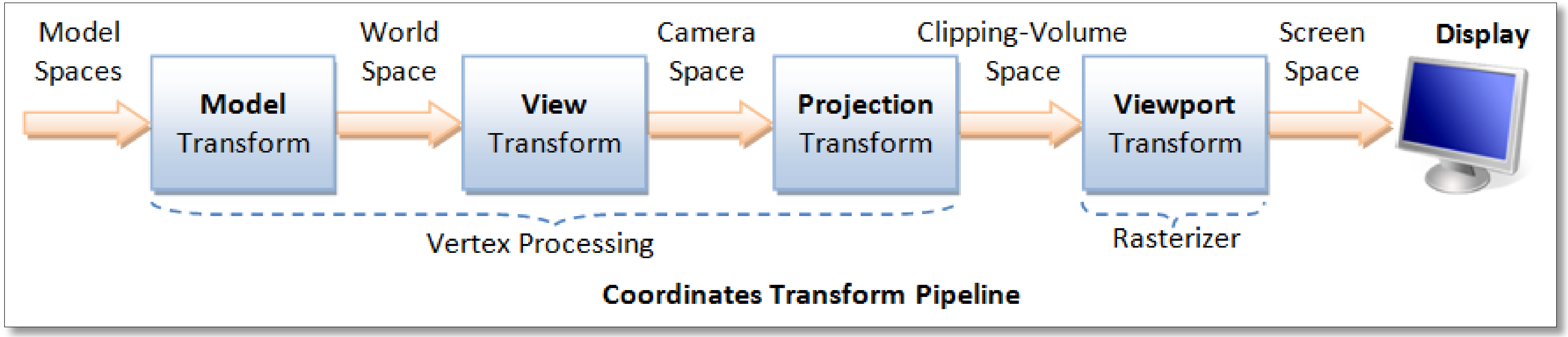


GPU (Hundreds of Cores)



Introduction to Shader Programing

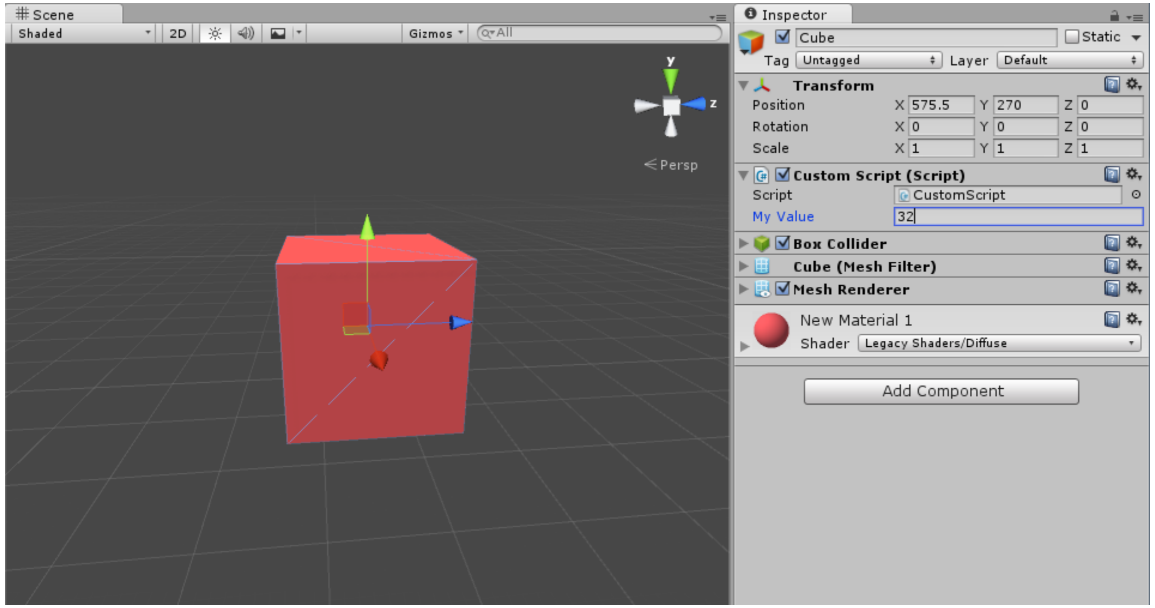

Raw Vertices & Primitives
Vertex Processor (Programmable)
Transformed Vertices & Primitives
3D
Rasterizer
Fragments
3D
Fragment Processor (Programmable)
Processed Fragments
3D
Output Merging
Pixels
Display
2D array of color-values



Coordinates Transform Pipeline

Unity Rendering

- In order to be rendered Game Objects need:
  - Mesh Filter
  - Mesh Renderer
- Mesh Renderer contains a reference to a material
- Materials are simply an interface to the shader program



Cube ☐ Static

Tag Untagged Layer Default

Transform

	X	Y	Z
Position	575.5	270	0
Rotation	0	0	0
Scale	1	1	1

☑ Custom Script (Script)

Script CustomScript

My Value 32

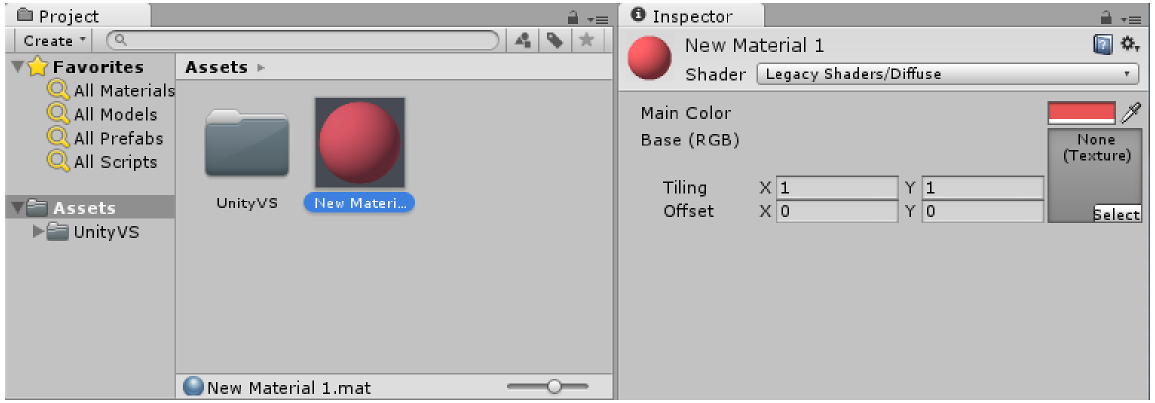
☑ Box Collider

Cube (Mesh Filter)

☑ Mesh Renderer

New Material 1

Shader Legacy Shaders/Diffuse



New Material 1

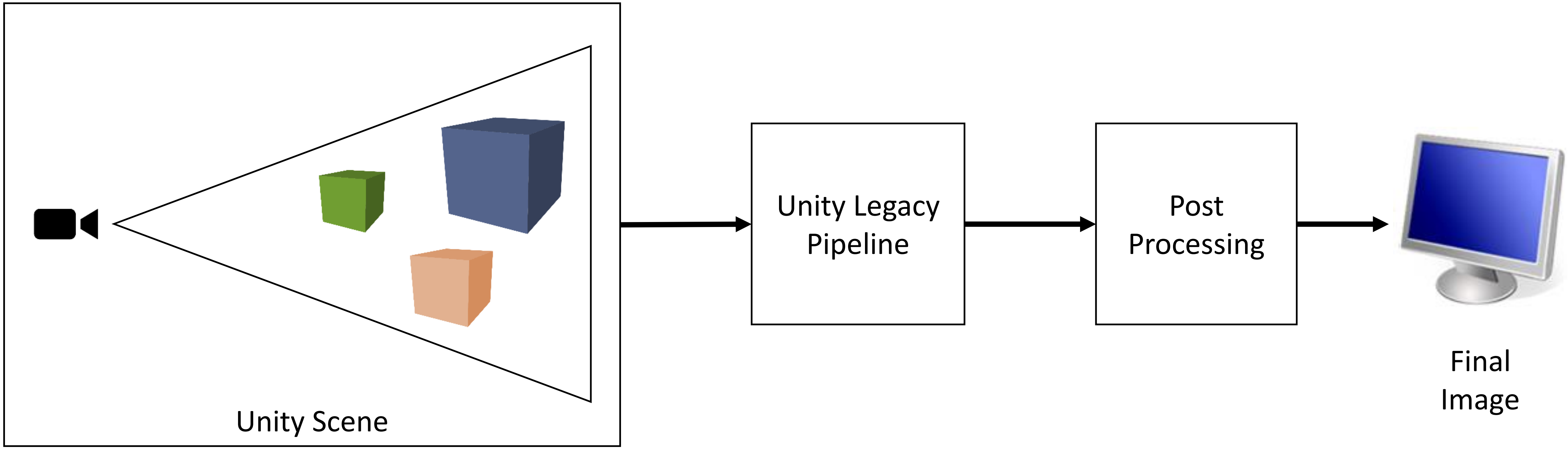
Shader Legacy Shaders/Diffuse

Main Color

Base (RGB)

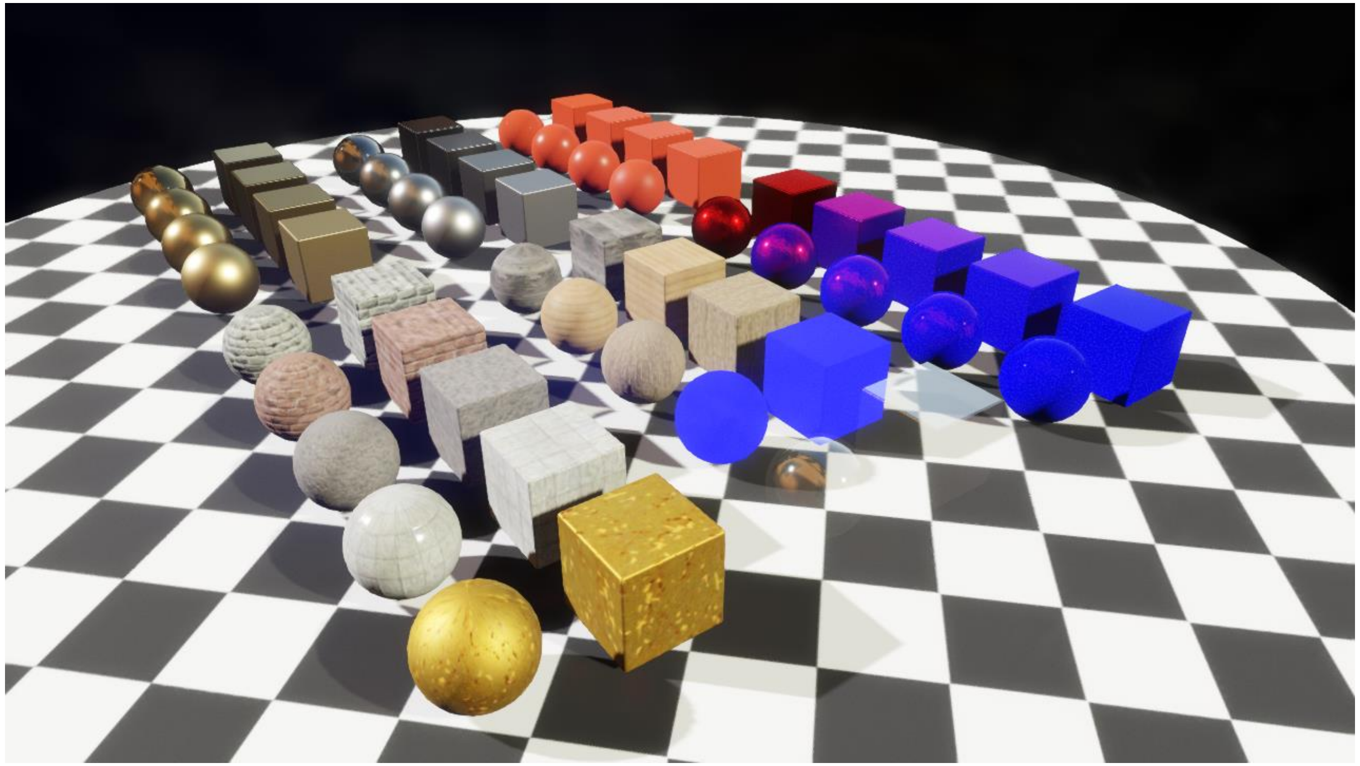
	X	Y
Tiling	1	1
Offset	0	0

Unity Rendering Pipeline



Unity Legacy Shaders

- Wide variety of legacy shaders
- Work out of the box, no need to script them
- Interface with shader parameters via the material properties



Writing Custom Unity Shaders

- HLSL/CG – cross compiled to GLSL for certain platforms
- Out of date OpenGL version (update announced soon)
- Advanced GPU stuffs only with DX11
- Windows platforms (7,8,10) with recent GPU is prefered

Writing Custom Unity Shaders

- Surface Shaders
  - Custom to Unity‘s pipeline
  - Specific syntax
  - Desgined to interact with complex ligtings setups (deffered lighting, shadows, global illumination)
- Vanilla Shaders
  - Shaders as we know it, vertex, fragment, etc.
  - More freedom, but no out-of-the box lighting
- Compute Shaders
  - GPGPU computation made easy
  - Simple interoperability with DX11

Useful Links

- HLSL Documentation

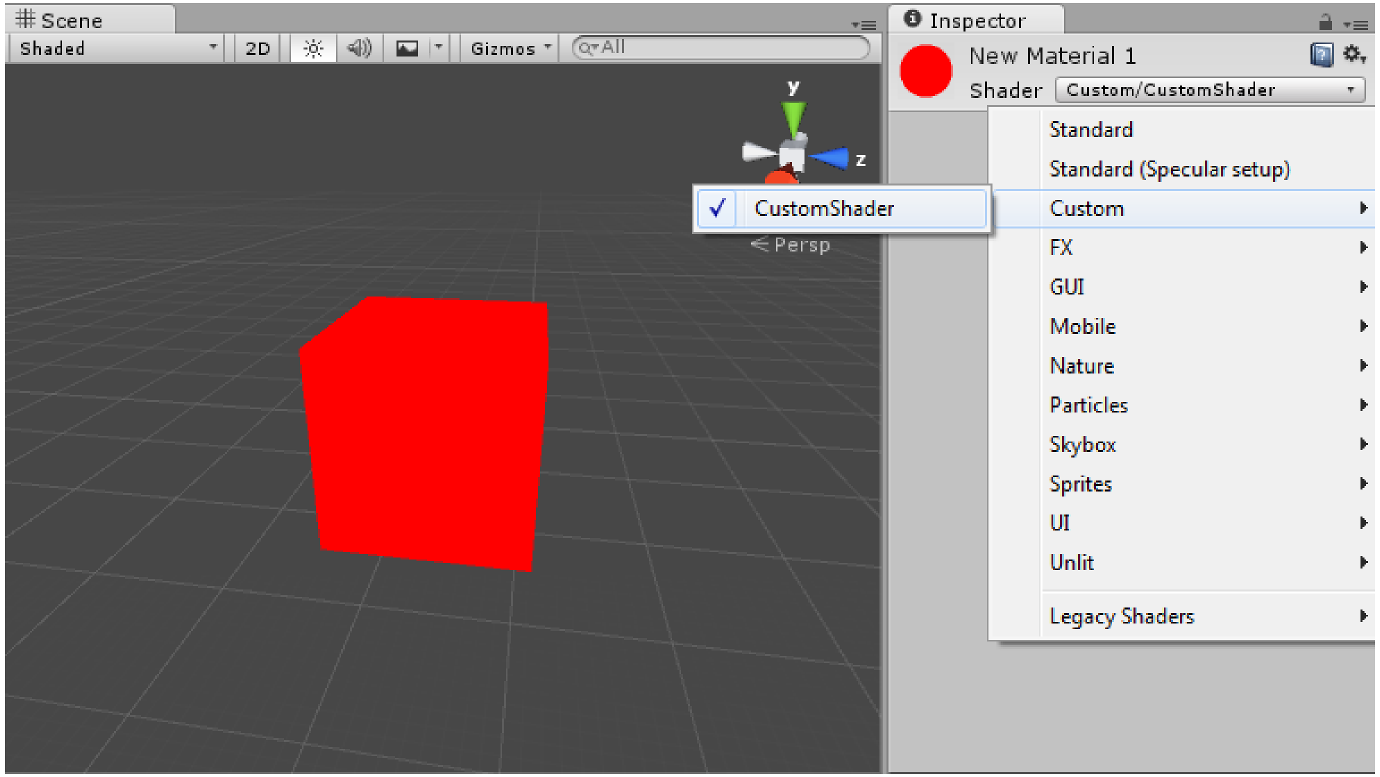
https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx

- Unity Shader Reference

http://docs.unity3d.com/Manual/SL-ShaderPrograms.html

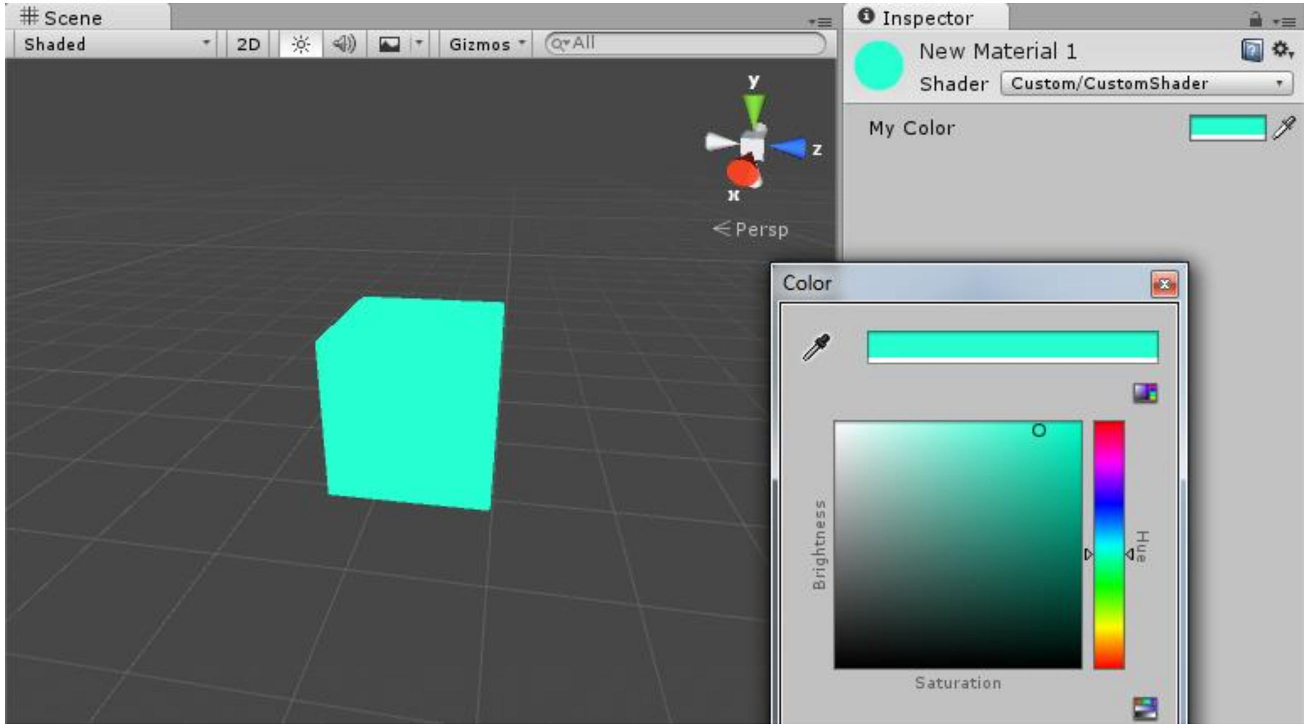
Simple Color Shader

```
Shader "Custom/ColorShader"
{
    SubShader
    {
        Pass
        {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            float4 vert(appdata_base v) : POSITION
            {
                return mul(UNITY_MATRIX_MVP, v.vertex);
            }

            float4 frag(float4 position:POSITION) : COLOR
            {
                return float4(1,0,0,1);
            }

            ENDCG
        }
    }
}
```



Simple Color Shader

```
Shader "Custom/CustomShader"
{
    Properties
    {
        _MyColor("My Color", Color) = (1, 1, 1, 1)
    }

    SubShader
    {
        Pass
        {
            CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            float4 _MyColor;

            float4 vert(appdata_base v) : POSITION
            {
                return mul(UNITY_MATRIX_MVP, v.vertex);
            }

            float4 frag(float4 position:POSITION) : COLOR
            {
                return _MyColor;
            }

            ENDCG
        }
    }
}
```



New Material 1

Shader Custom/CustomShader

My Color

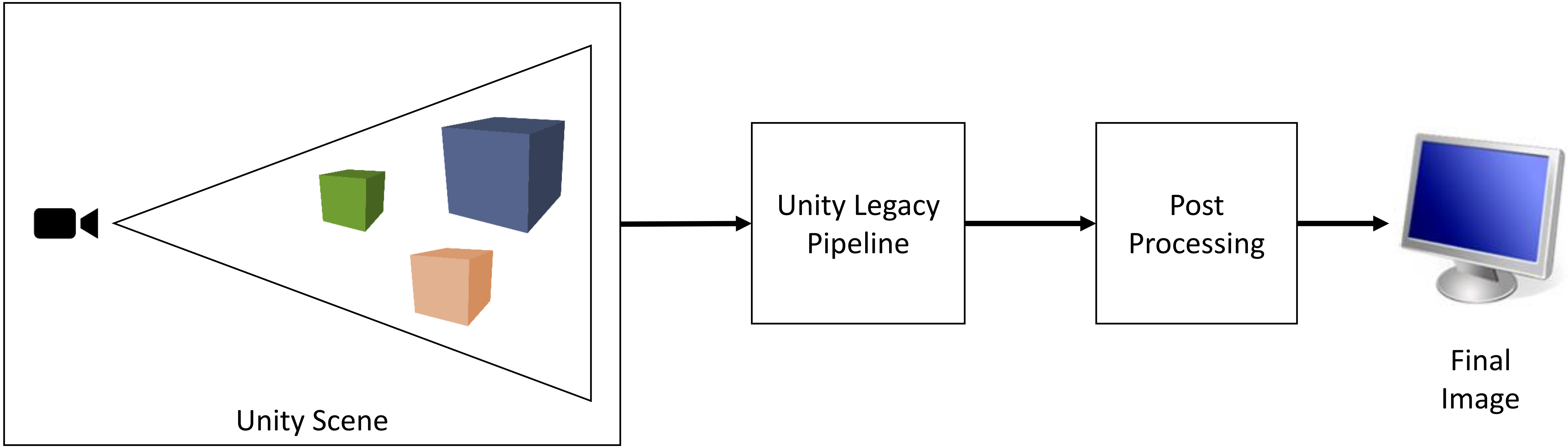
Shader Scripting Demo

Advanced GPU Programing with Unity3D - Part 2

Overview

- High level drawing
- Compute Buffers
- Procedural Drawing
- Instancing
- Billboards
- Compute Shaders

Unity Rendering Pipeline



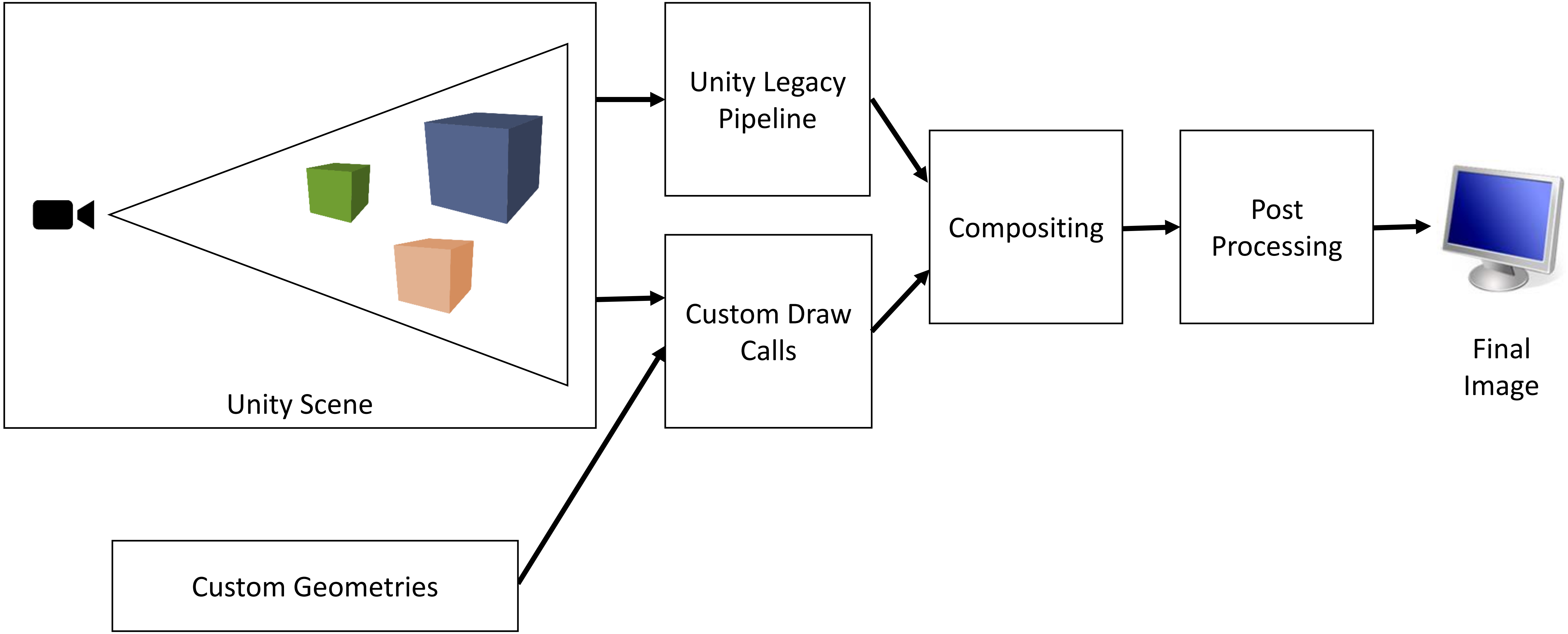
Unity Scene

Unity Legacy Pipeline

Post Processing

Final Image

Hacking Unity‘s Pipeline



High-Level Drawing Functions

- Important Game Object Callbacks:
  - OnRenderObject() // To draw stuffs
  - OnRenderImage() // For post-processing
- Important Drawing Functions
  - Graphics.DrawMeshNow() // For drawing meshes stored in the project
  - Graphics.DrawProcedural() // For drawing custom geometries, procedurals meshes, lines, particles...
- Bind shader via Material.SetPass()

Code + Demo

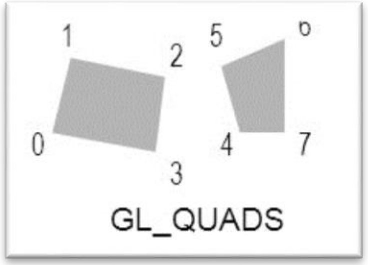
- DrawMesh.cs

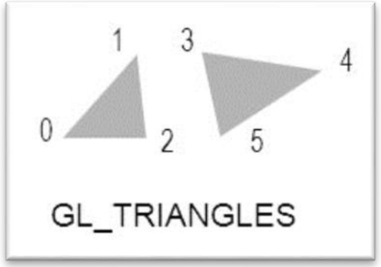
Compute Buffers

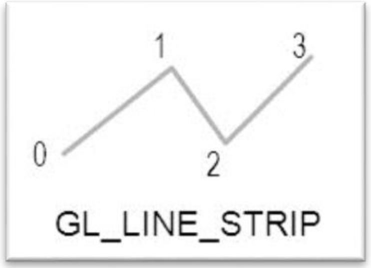
- GPU buffer to store generic information, ints, floats, vectors, matrices, custom types
- Easy setup of the CPU side
  - public **ComputeBuffer**(int **count**, int **stride**);
  - public void **SetData**(Array **data**);
  - public void **SetBuffer**(string **propertyName**, ComputeBuffer **buffer**);
- Easy setup on the GPU side
  - StructuredBuffer<float> myBuffer;
- Must be cleared when terminating the program

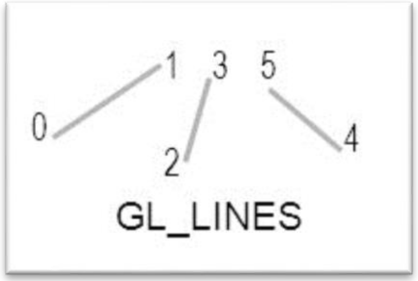
Procedural Drawing

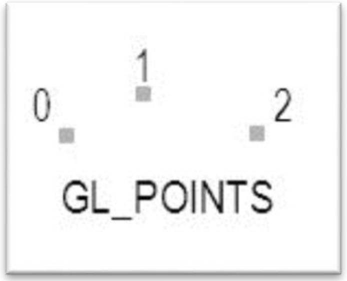
- Draws arbitrary geometries on the GPU
- Data must be uploaded on the GPU memory first
- Must specify topology before hand









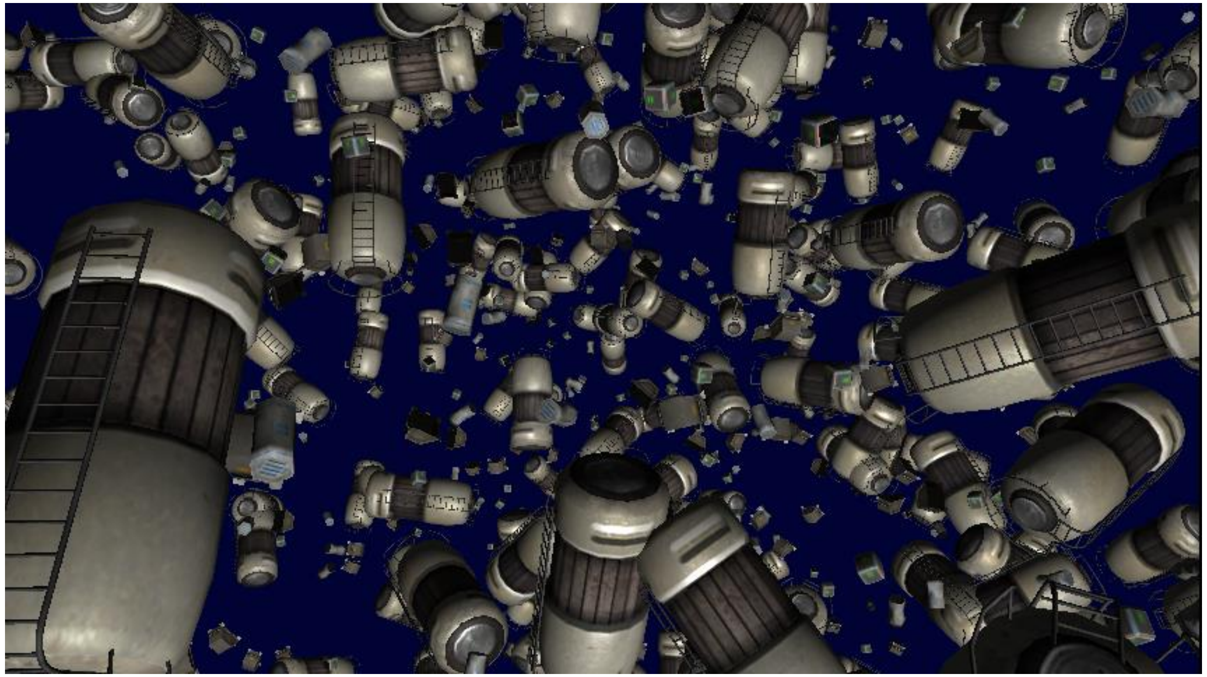


Code + Demo

- DrawRandomProcedural.cs
- DrawMeshProcedural.cs

Instancing

- Store all information on the GPU
- Reuse same geometry to draw multiple times
- Position / rotation differ for each instance
- Drawing can be issued in a single draw call
- Much faster than issuing one draw call per instance

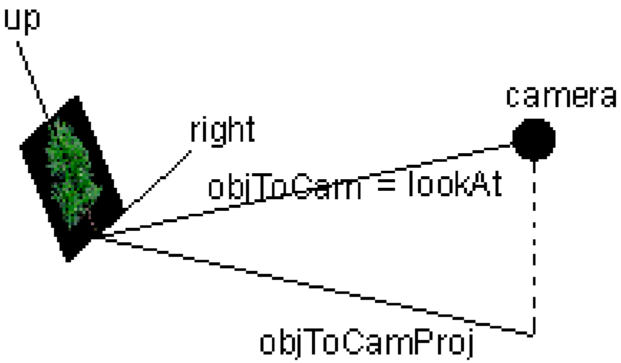


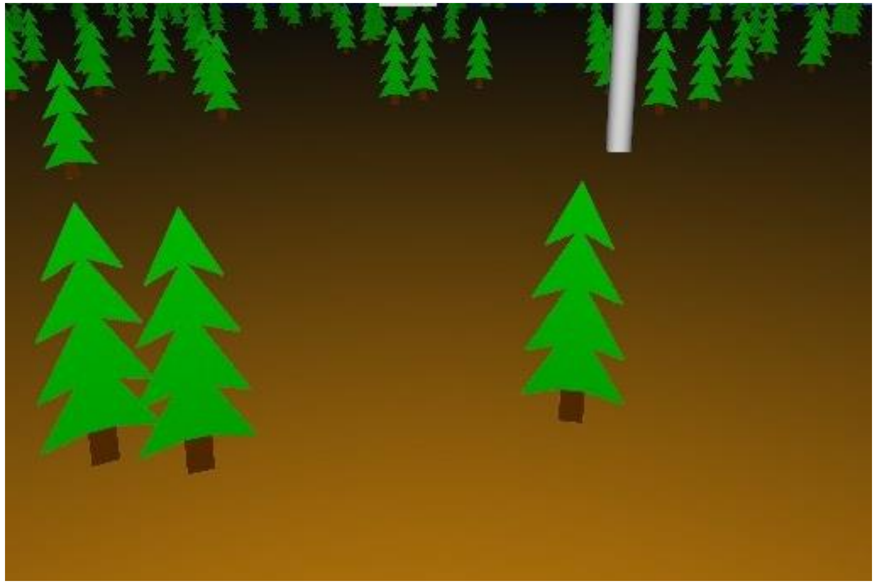
Code + Demo

- DrawMeshInstanced.cs
- DrawInstanced.cs

Textured Billboards

- Billboards are 2D elements incrusted in a 3D world
- Camera facing textured quads
- Usfeful in game for populating backgroud elements
- Must faster to render than meshes





Code + Demo

- DrawBillboards.cs

Compute Shaders

- GPU parallel computing for generic purposes
- Computation is done outside the rendering pipeline
- Similar to CUDA, OpenCL
- nteroperability with DX11
- Same HLSL syntax as shaders

Compute Shader Example

```
// test.compute

#pragma kernel FillWithRed                    // Kernel declaration (entry point)

RWTexture2D<float4> res;                      // Read-write Buffer

[numthreads(1,1,1)]
void FillWithRed (uint3 id: SV_DispatchThreadID)
{
  res[id.xy] = float4(1,0,0,1);
}
```

Code + Demo

- DrawBillboardCompute.cs