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

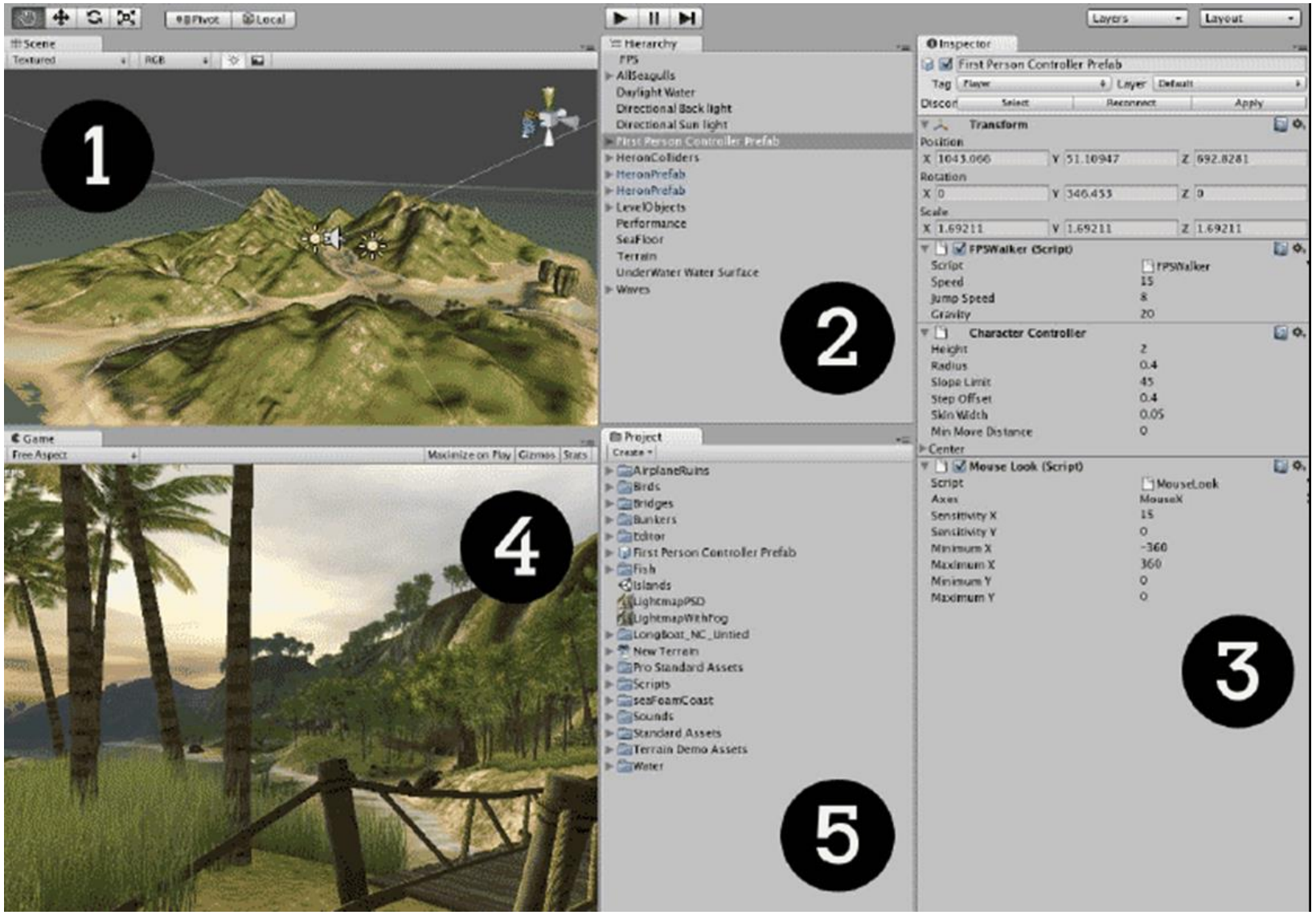
1 – Scene

2 – Hierarchy

3 – Inspector

4 – Game

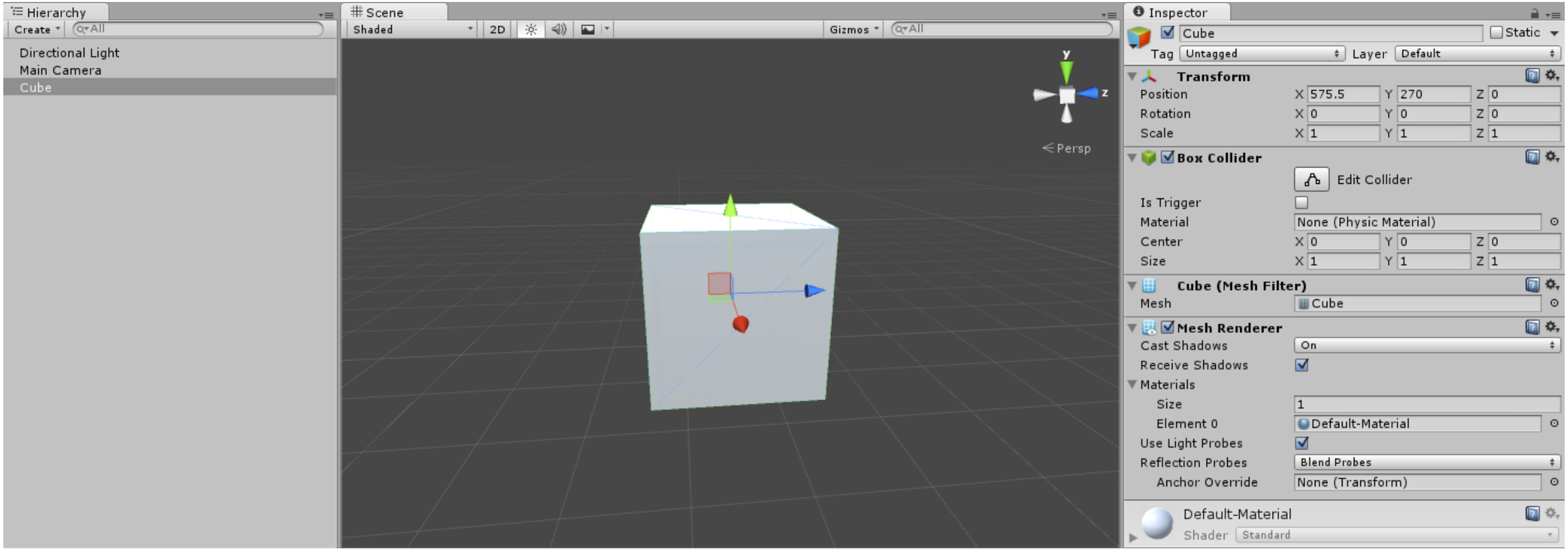
5 – Project



Game Objects

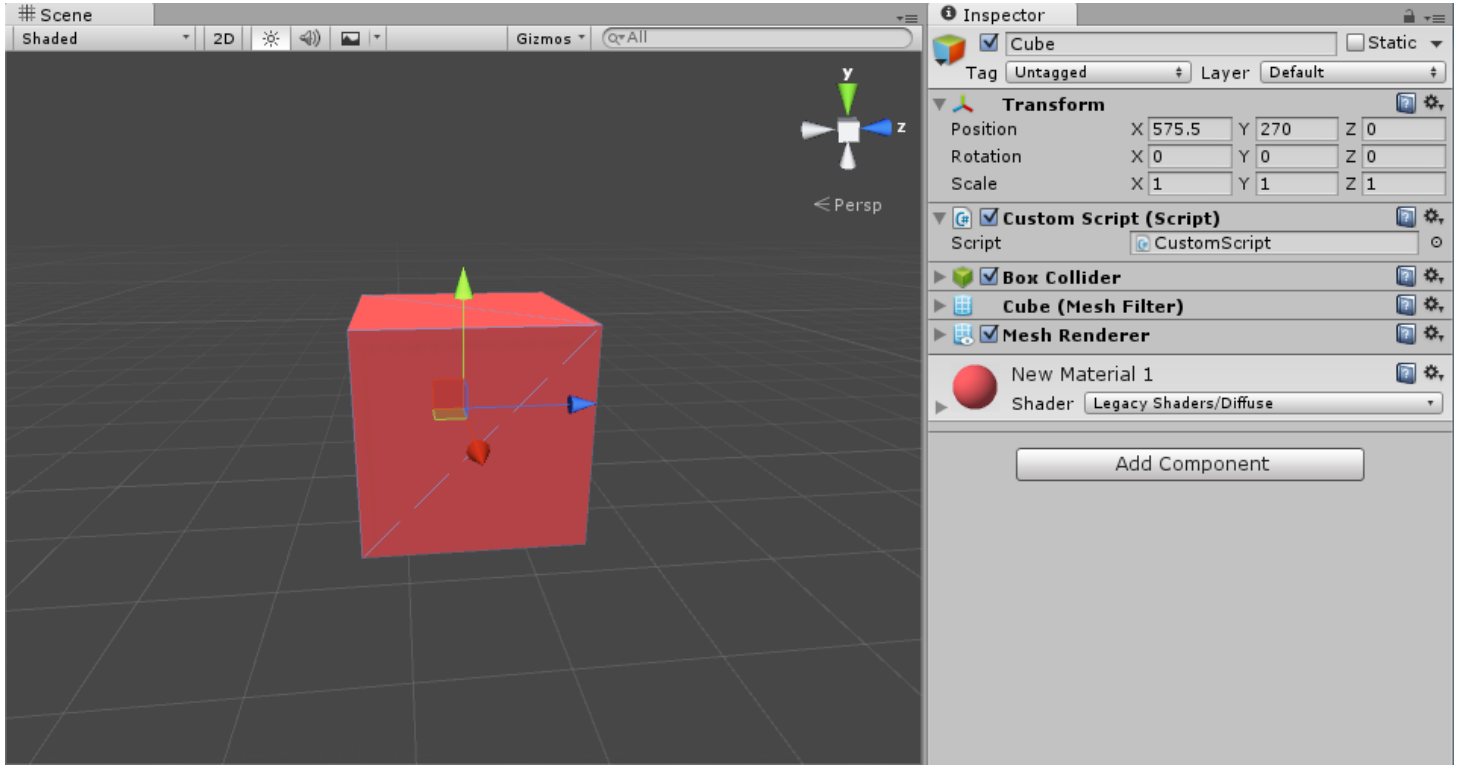
- Everything is a Game Object (lights, cameras, characters,…)
- Contains components (mesh, audio, script, physics, etc.)
- Transform component by default
- Game Objects may contain other game objects (placeholders)

Game Object - Cube Example



Scripting

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



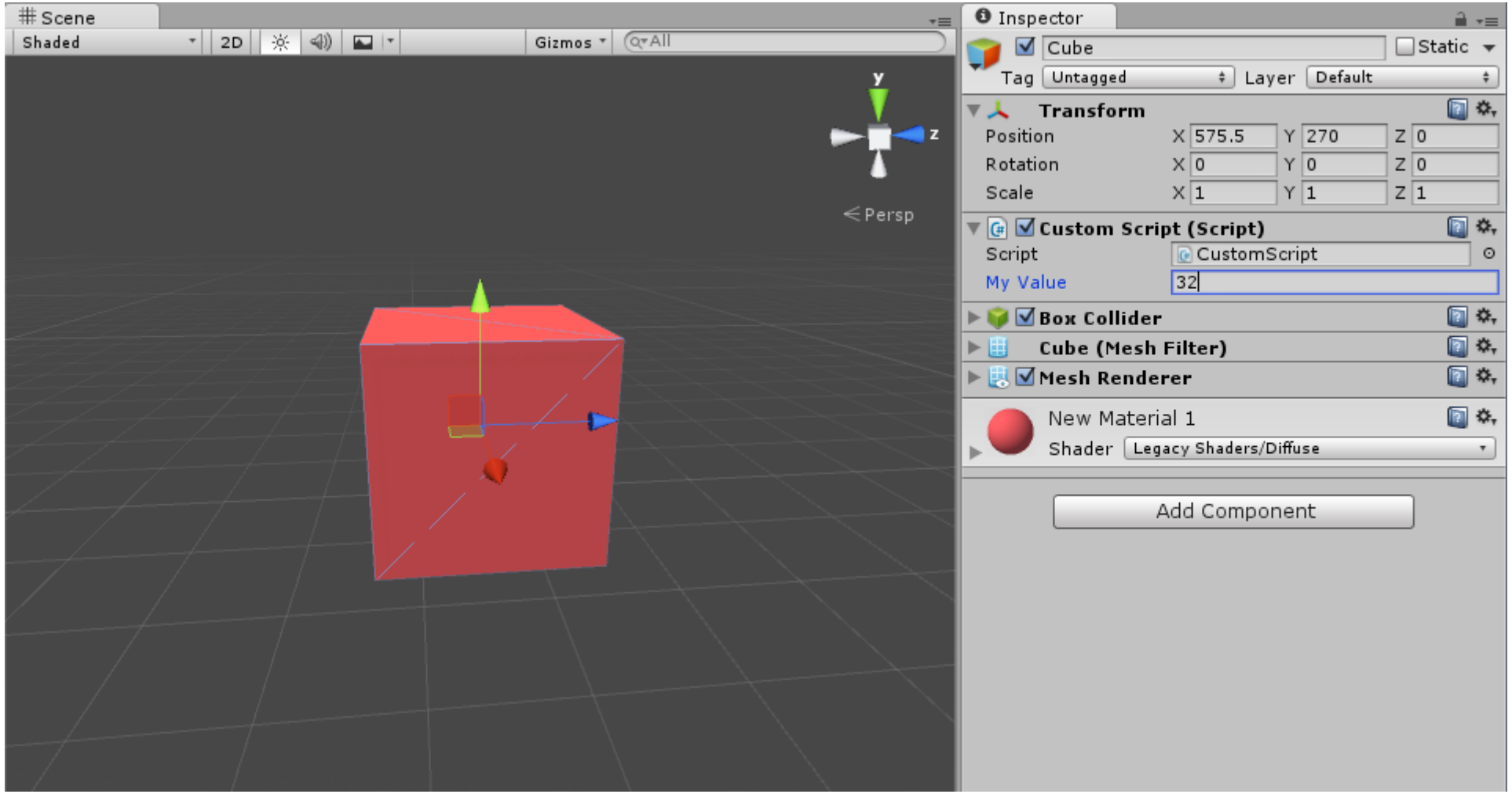
Scripting

```
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


Scene
Shaded
2D
Gizmos
All
y
z
Persp
Inspector
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
```

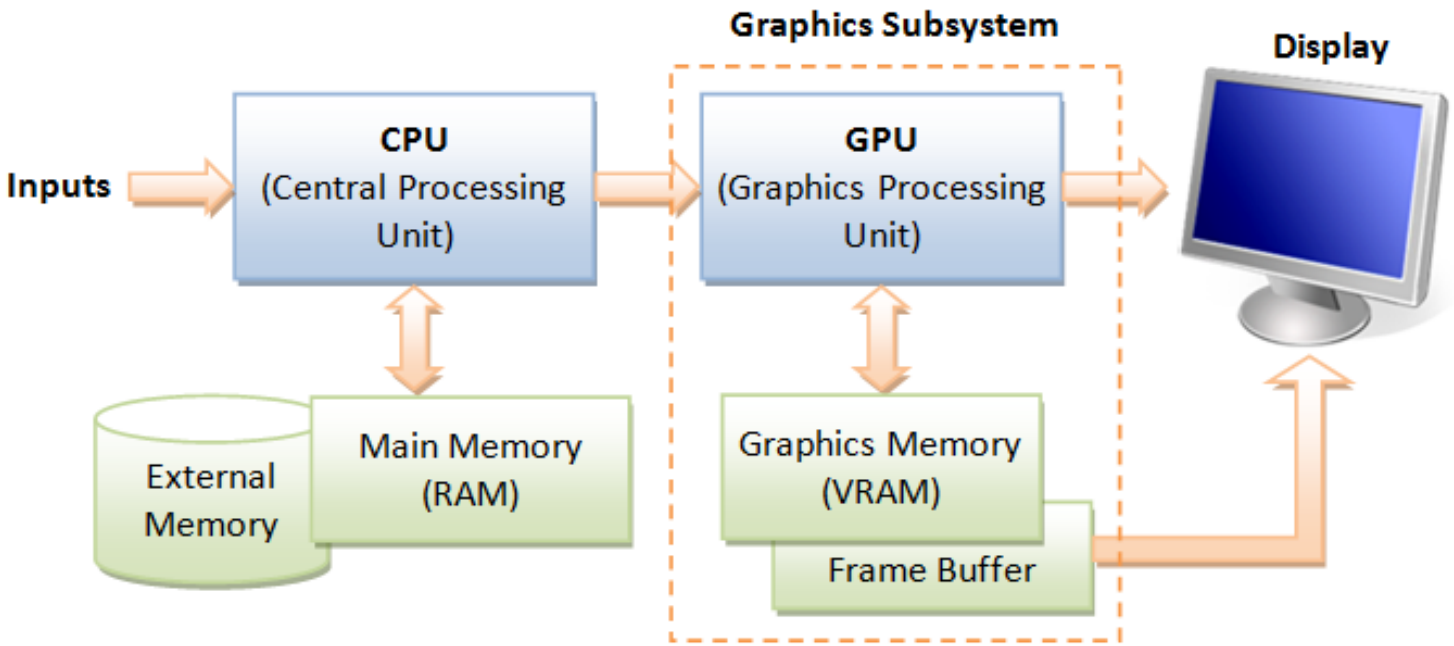
- Useful callbacks

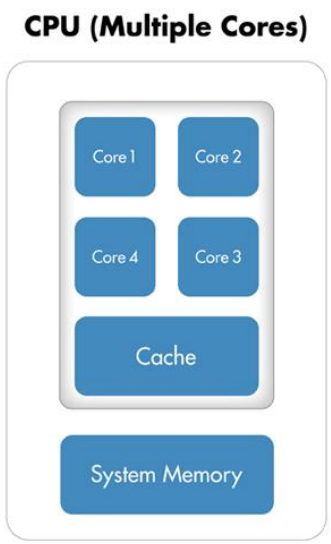
```
void Start () {} // Called when the game starts to play
void Update() {} // Called every frame
void OnDestroy() {} // Called when the game object is destroyed
... many more, check the documentation
```

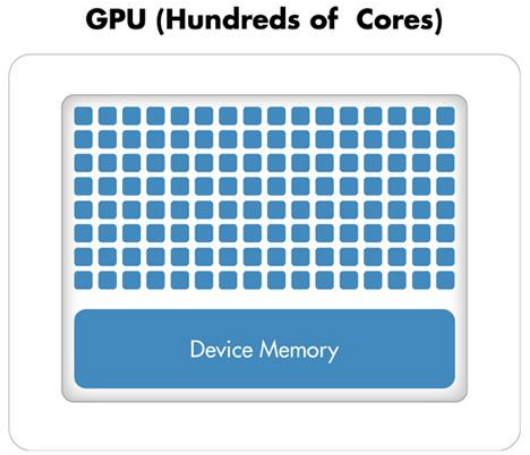
Scripting Demo

Rendering with Unity

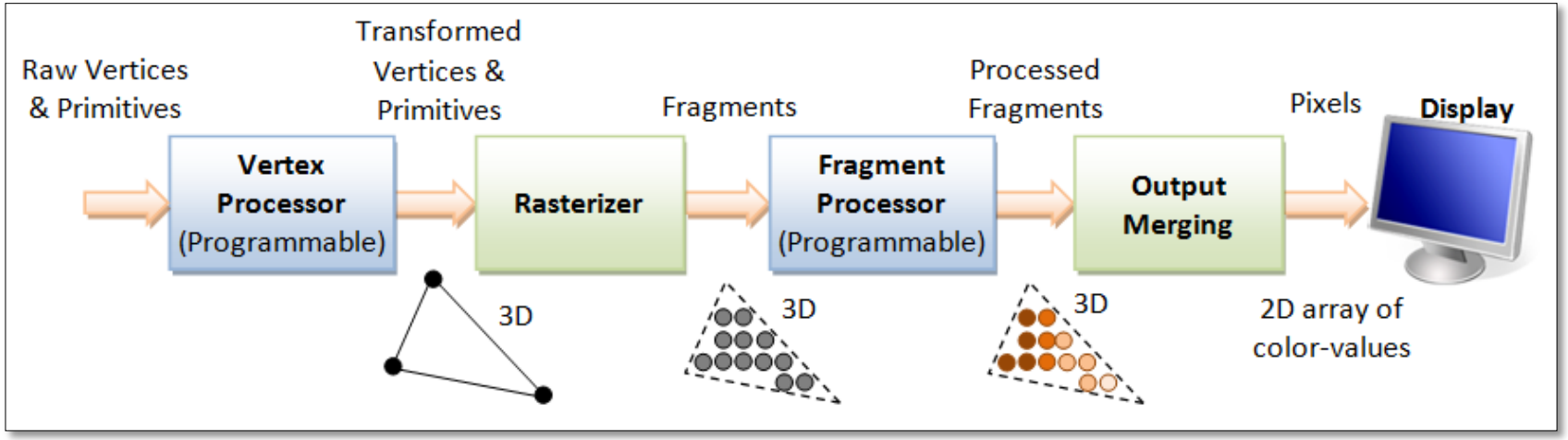
Introduction to Shader Programing

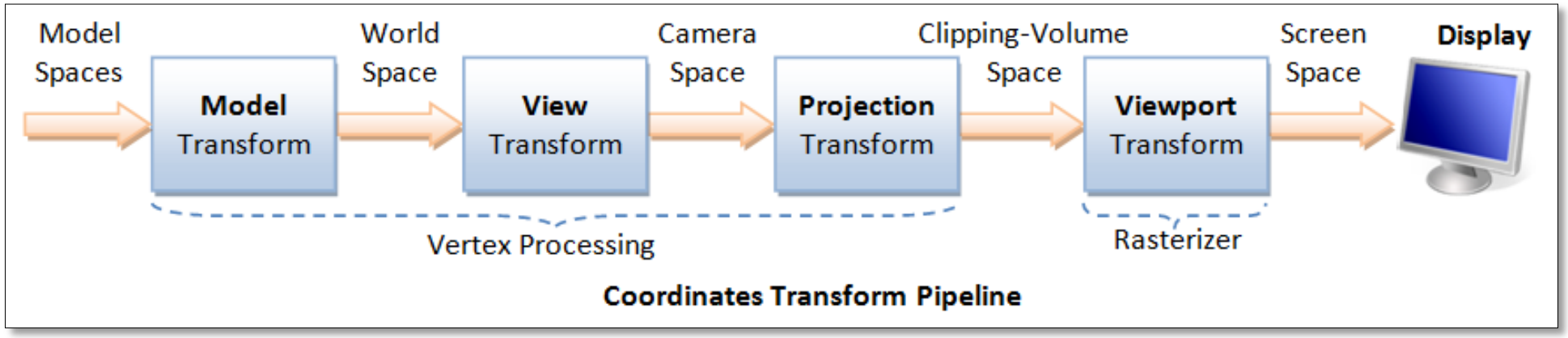






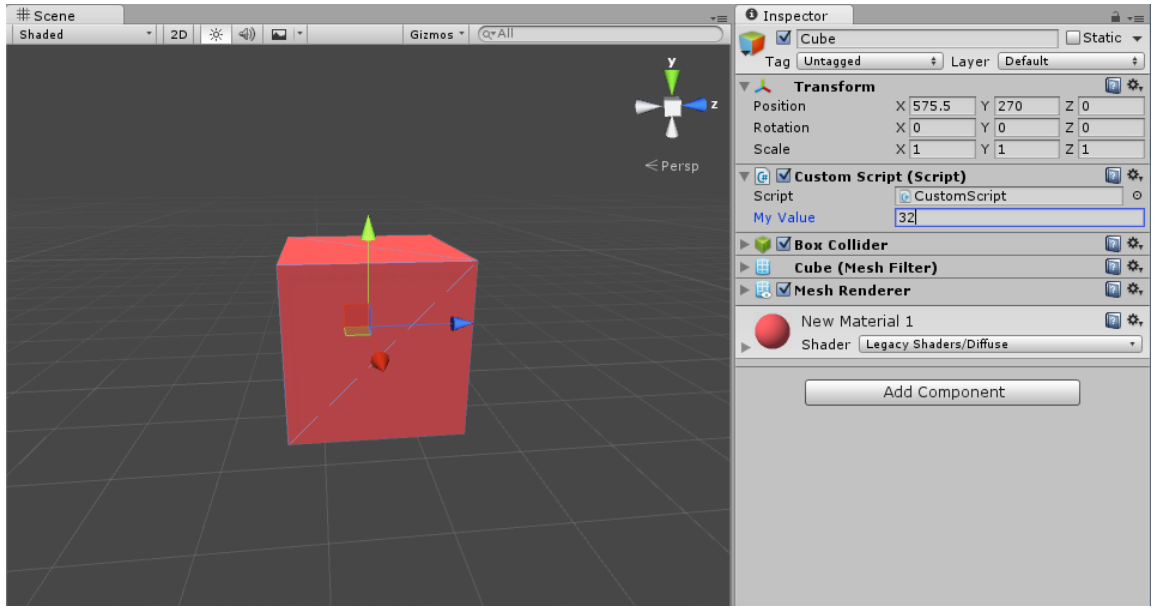
Introduction to Shader Programing

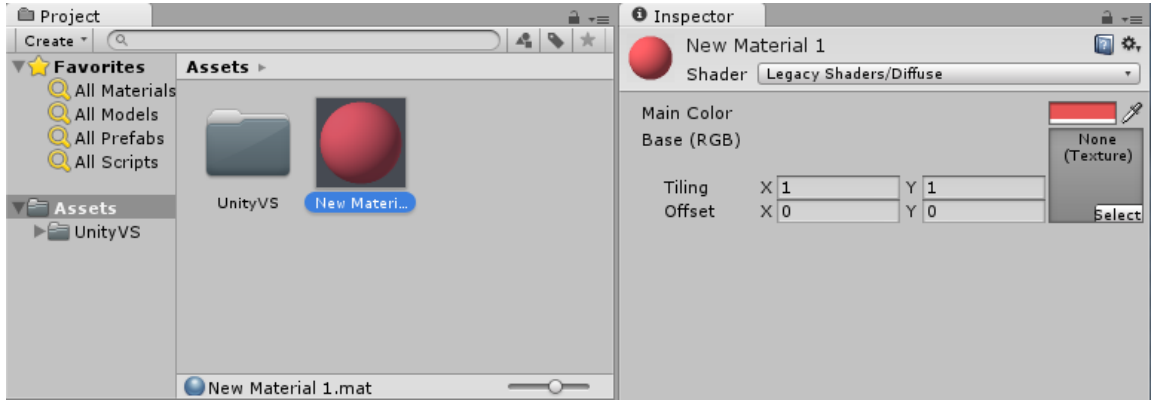




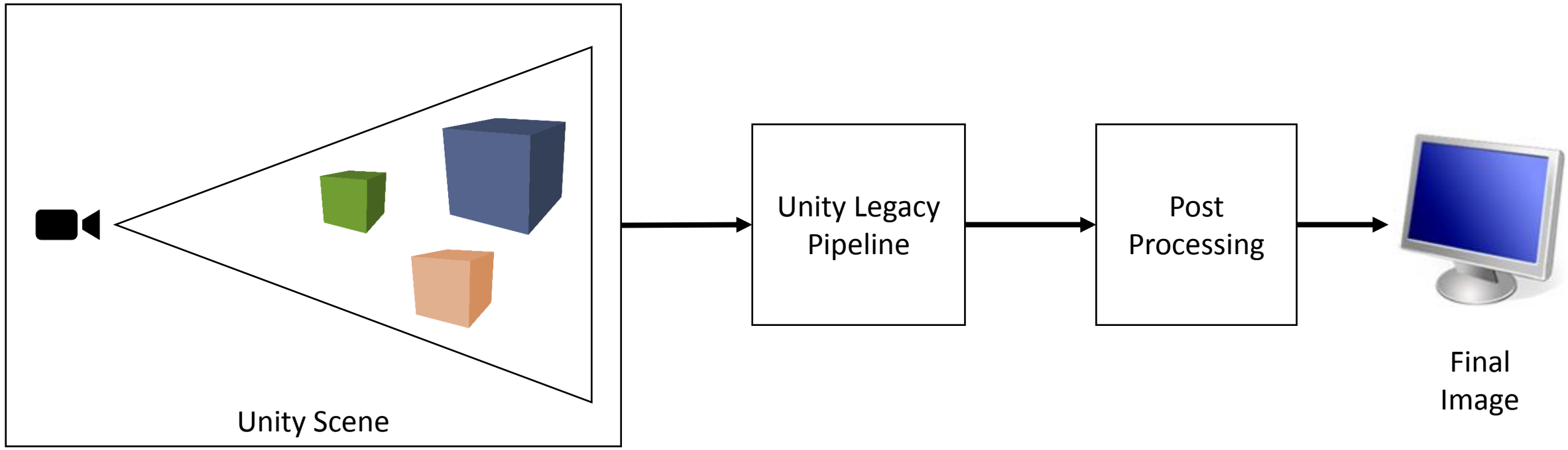
Unity Rendering

- 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



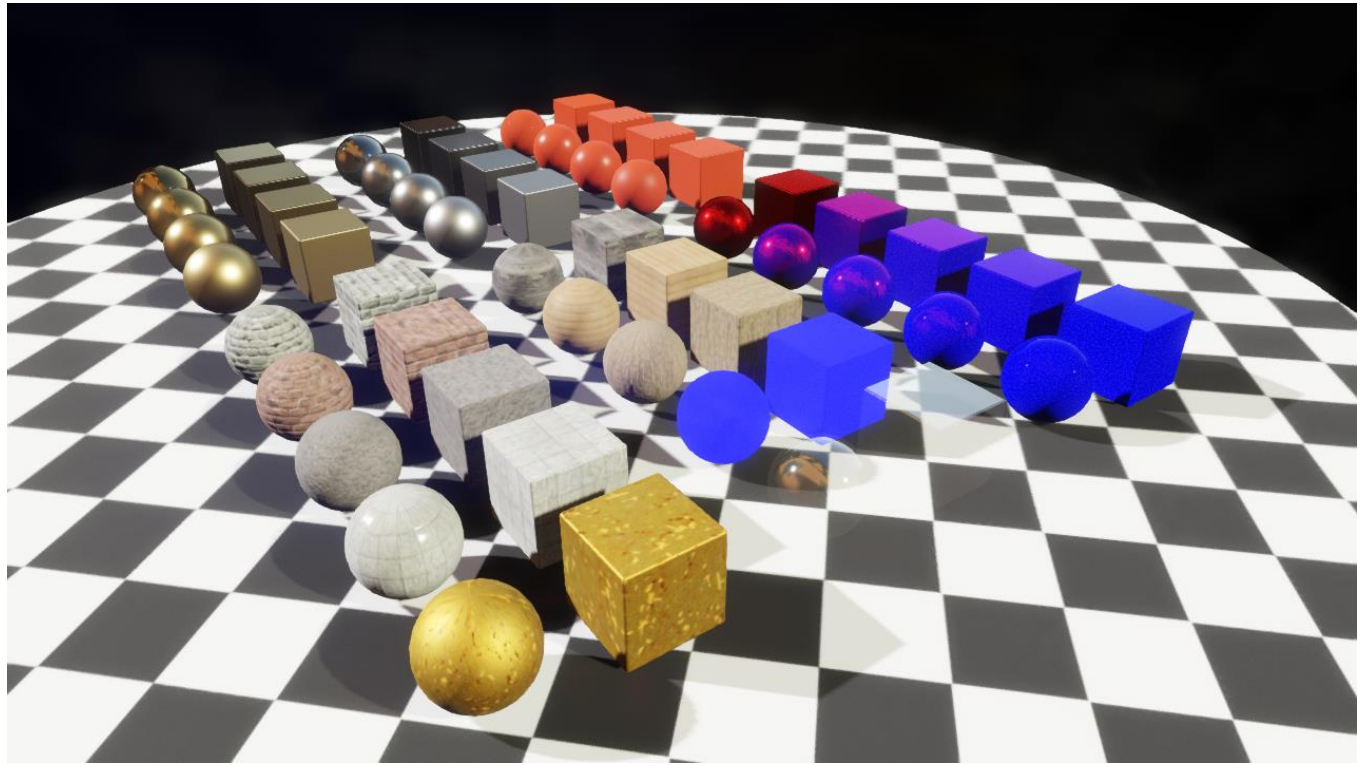


Unity Rendering Pipeline



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

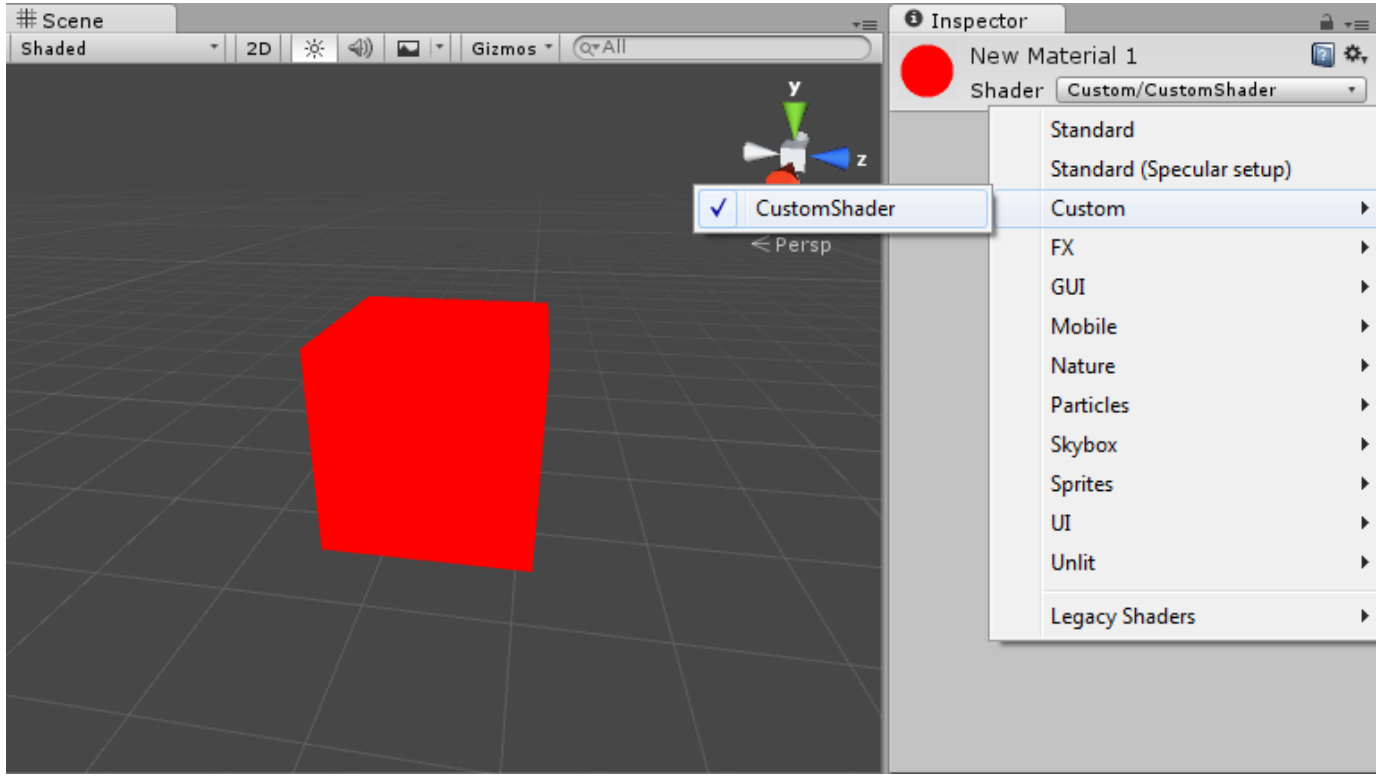
https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx

- Unity Shader Reference

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

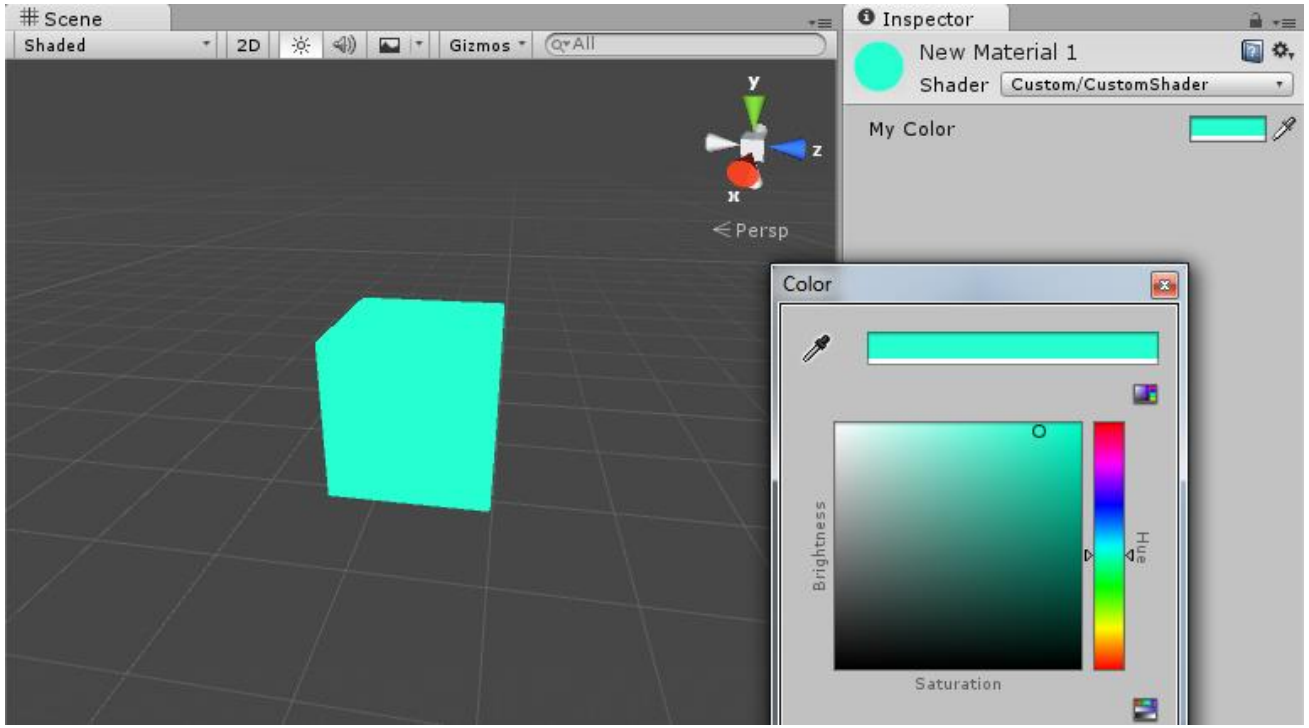
Simple Color Shader

```
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
        }
    }
}
```



Shader Scripting Demo

Advanced GPU Programing with Unity3D

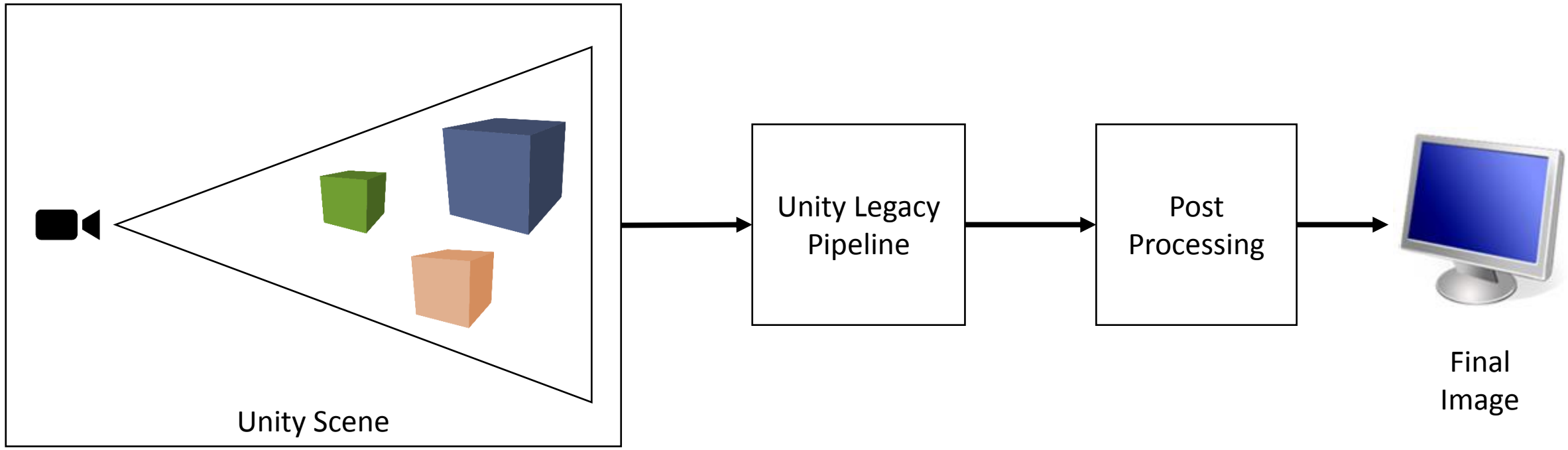
-

Part 2

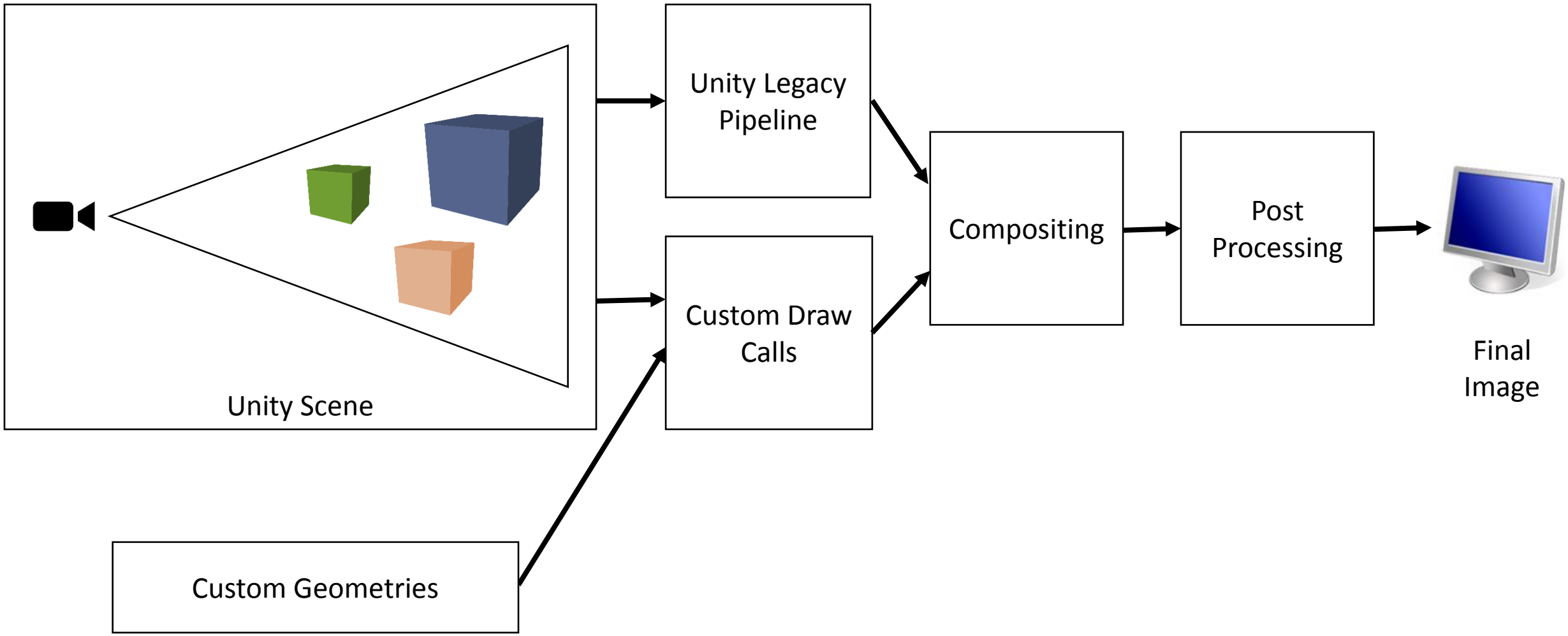
Overview

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

Unity Rendering Pipeline



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

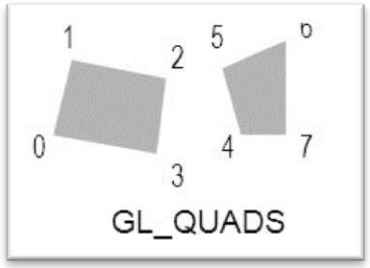
- DrawMesh.cs

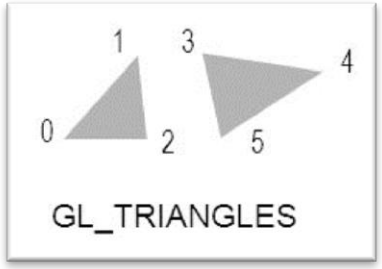
Compute Buffers

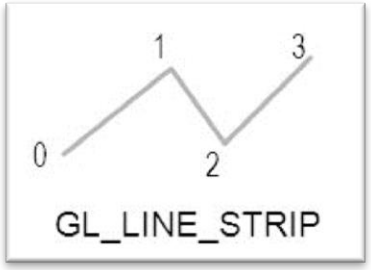
- GPU buffer to store generic information, ints, floats, vectors, matrices, custom types
- Easy setup of the CPU side
  - public **ComputeBuffer**(int **count**, int **stride**);
  - public void **SetData**(Array **data**);
  - public void **SetBuffer**(string **propertyName**, ComputeBuffer **buffer**);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

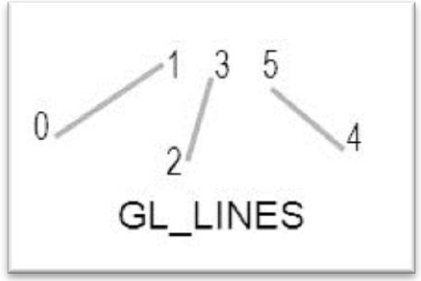
Procedural Drawing

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







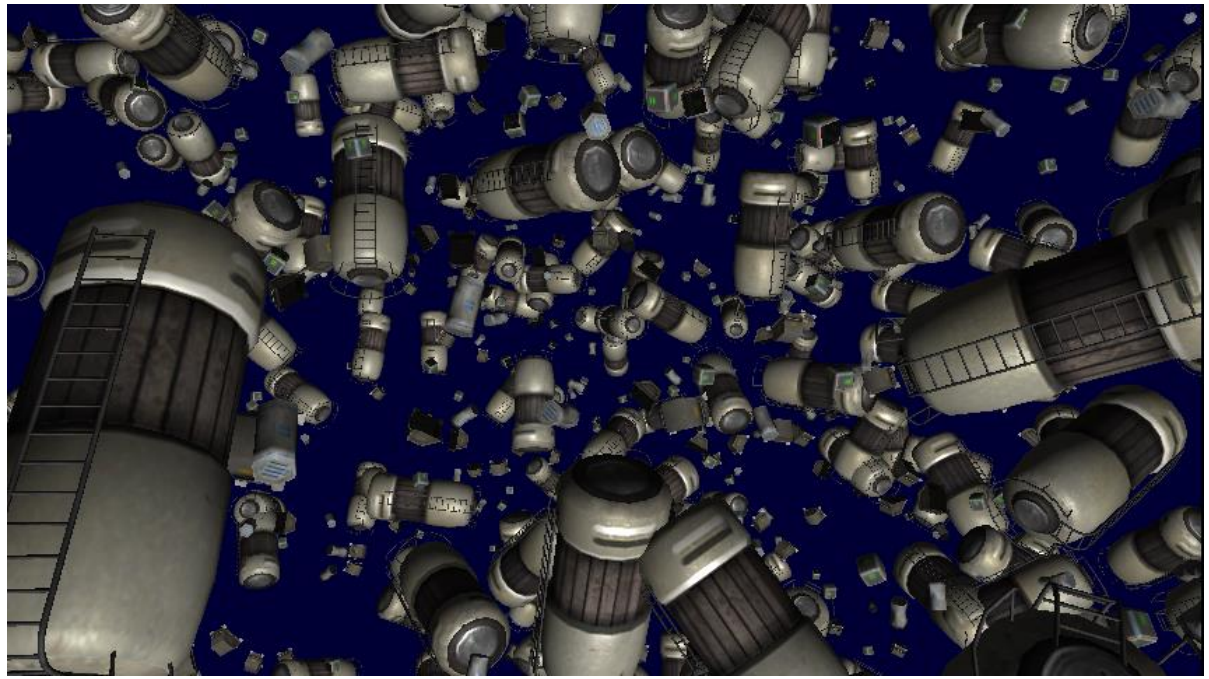


Code + Demo

- 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

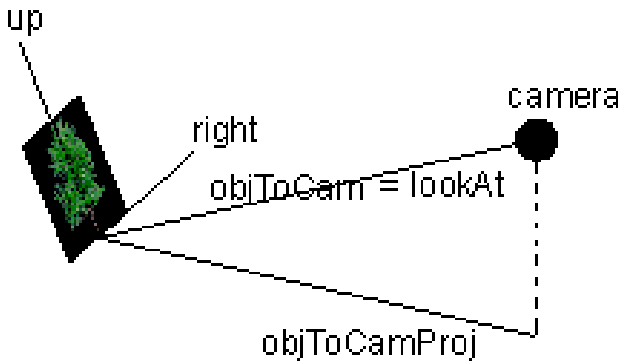


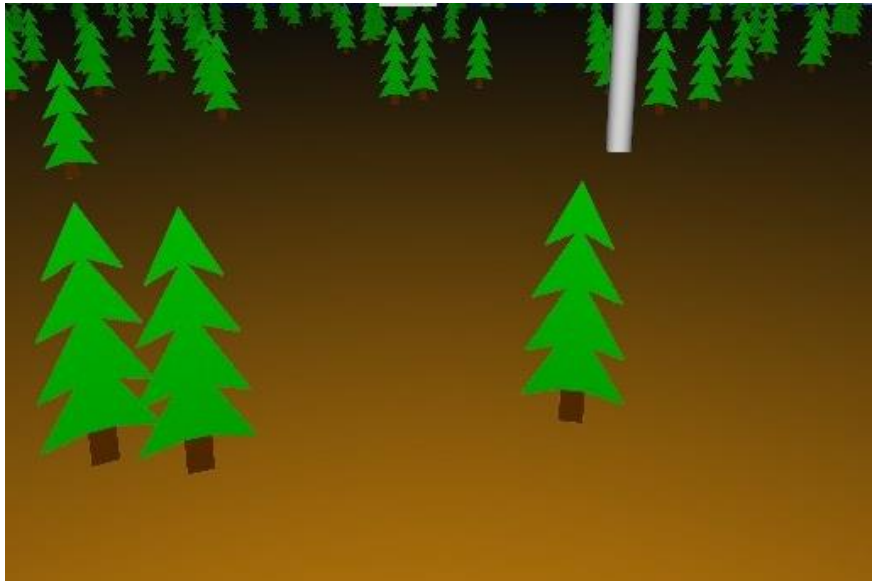
Code + Demo

- 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

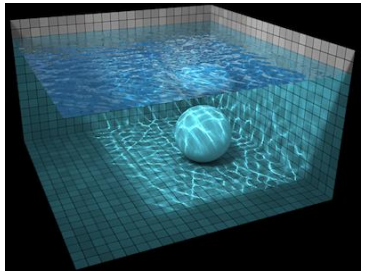
Advanced GPU Programing with Unity3D

-

Part 3

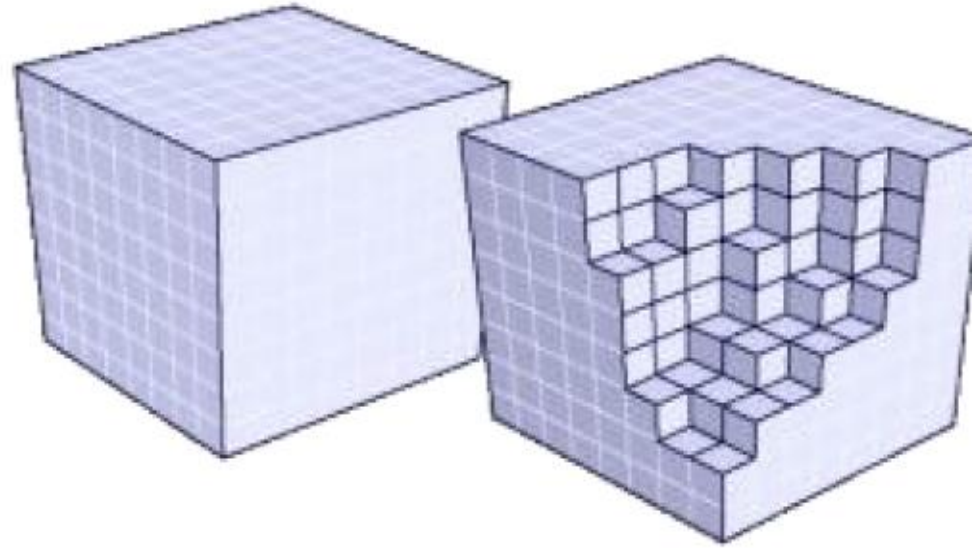
Suggested Topics

- Caustics
- Referactions
- Sub-surface scattering
- Ambient occlusion
- Translucent object with depth-peeling
- Relief/parallax mapping
- Non-photorealistic Rendering
- Physics-particle system



- Water simulation
- Hair simulation
- Cloth
- Shadow Mapping
- Smoke effects
- Terrain real-time tessalation
- Any reseach paper from SIGGRAPH, EGSR, EG, SIGGRAPH-ASIA, I3D, GDC, SCA, IEEE Vis

Voxels



- Analagous to pixels (picture elements), voxels (volume elements) are a discretised representation of 3D space
  - Spatial subdivision of 3D environment
  - Traditionally: environment discretised into homogeneous regular cubes i.e. discrete scalar field
  - Some extensions: object space discretisation, vector/tensor fields

Advantages

- Volumetric representation is arguably “real” 3D
  - Physically more accurate e.g. For simulation, physics: destruction, finite elements, fluids
  - More structural information in models
    - Interior details
    - Transparency
    - Fuzzy boundaries
    - Participating media
  - Illumination is not only a function of surface (e.g. Sub surface scattering)
- Potentially more appropriate discretization for rasterization:
  - Voxel to pixel mapping better than triangle to pixel mapping or texel to pixel
  - Can account for effects generated by parallax, displacement, bump-mapping
- Data more uniform – potentially more parallel

Volume Effects

Translucency and sub-surface detail

http://http.developer.nvidia.com/GPUGems/gpugems_ch39.html

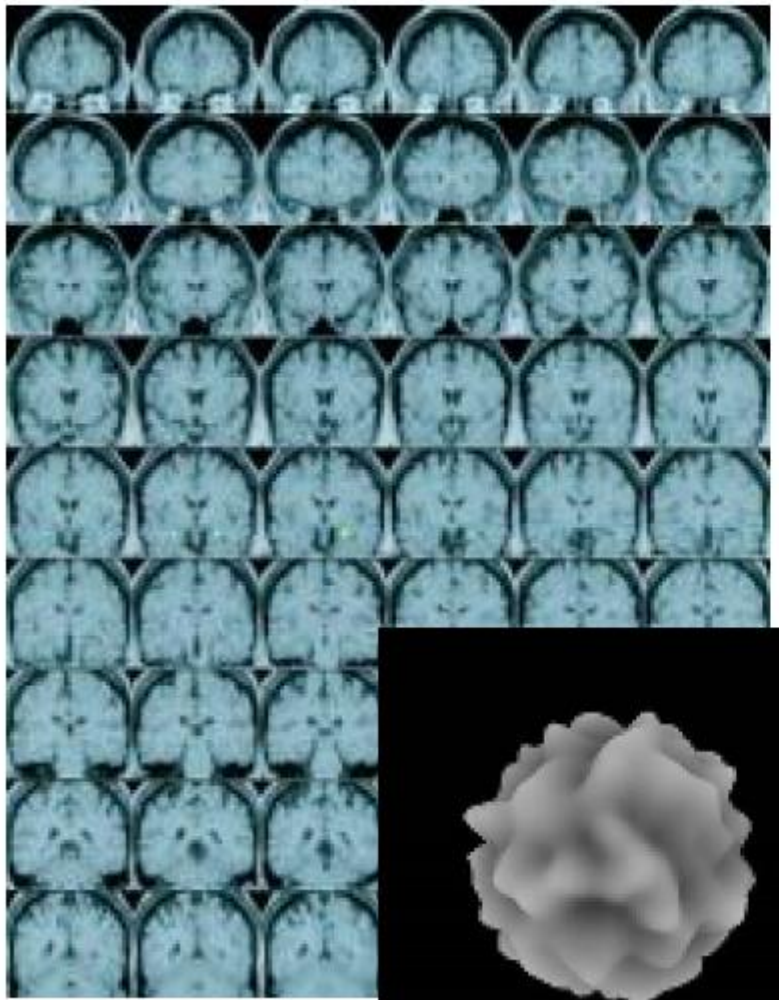
Volume Effects

Volume Effects

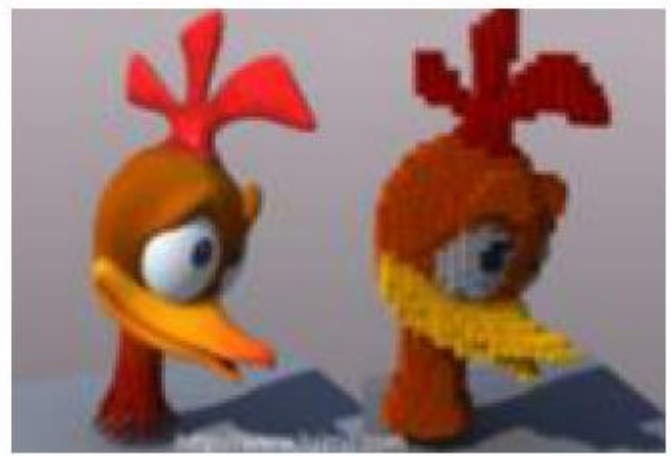
Challenges

- More data (some of it redundant)
  - $256^3 \approx 16$Mb. What about animation?
  - Large resolutions required to avoid looking blocky
- More complex operations for rendering equation
- Traditional graphics hardware driven more towards accelerating surface & texture models
- Difficult to manually model, edit
- Difficult to understand if not rendered carefully

Volume data

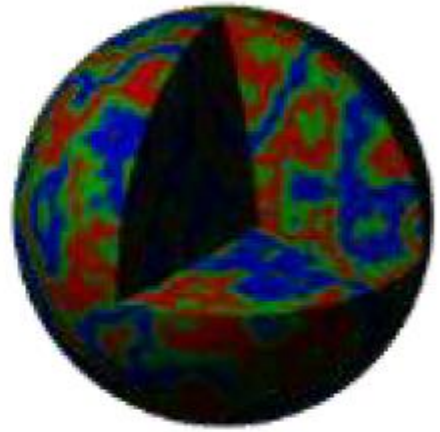


- Sources of volume data:
  - Scanned: e.g. CT, MRI
  - Procedural or simulated
  - Computed from surface: e.g. voxelised (baked)
  - Artist generated: simple volumes e.g. voxel games: minecraft, voxatron

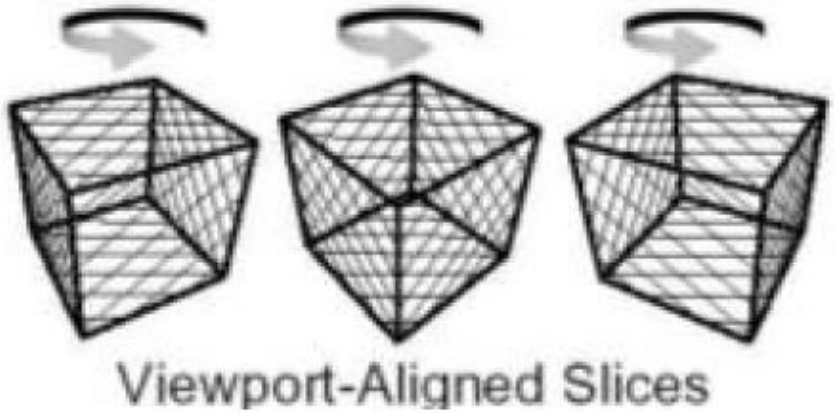


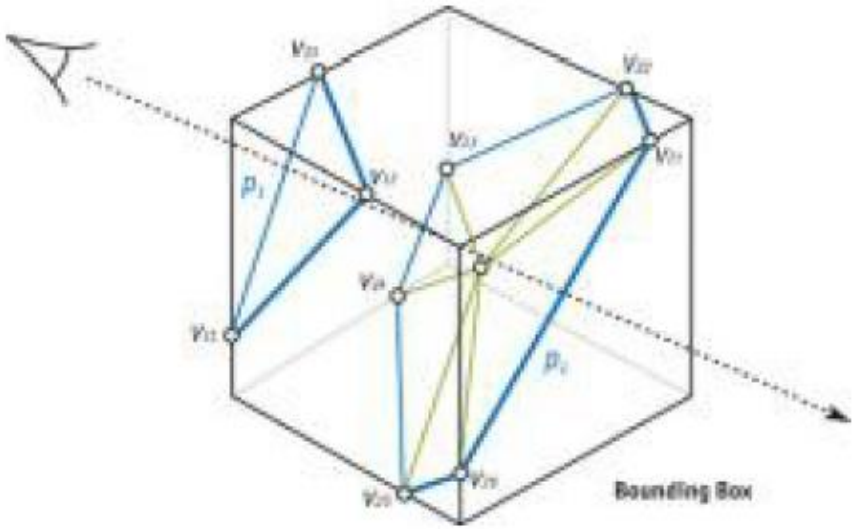
Volumetric Textures

- Texture mapping may be applied not only on the surface
- Volumetric textures define mapping 3D
  - Mostly procedural
    - Generators e.g. turbulence hlsl, noise glsl
  - More commonly used as textures in off-line renderings
  - For real-time, hardware support available. Several hardware related advantages:
    - direct 3D addressing
    - tri-linear interpolation
    - 3D coherent texture caching
  - N.B. Memory limitations!
    - 512^3 3D texture with 1 byte values takes over 128MB



View Aligned Slices



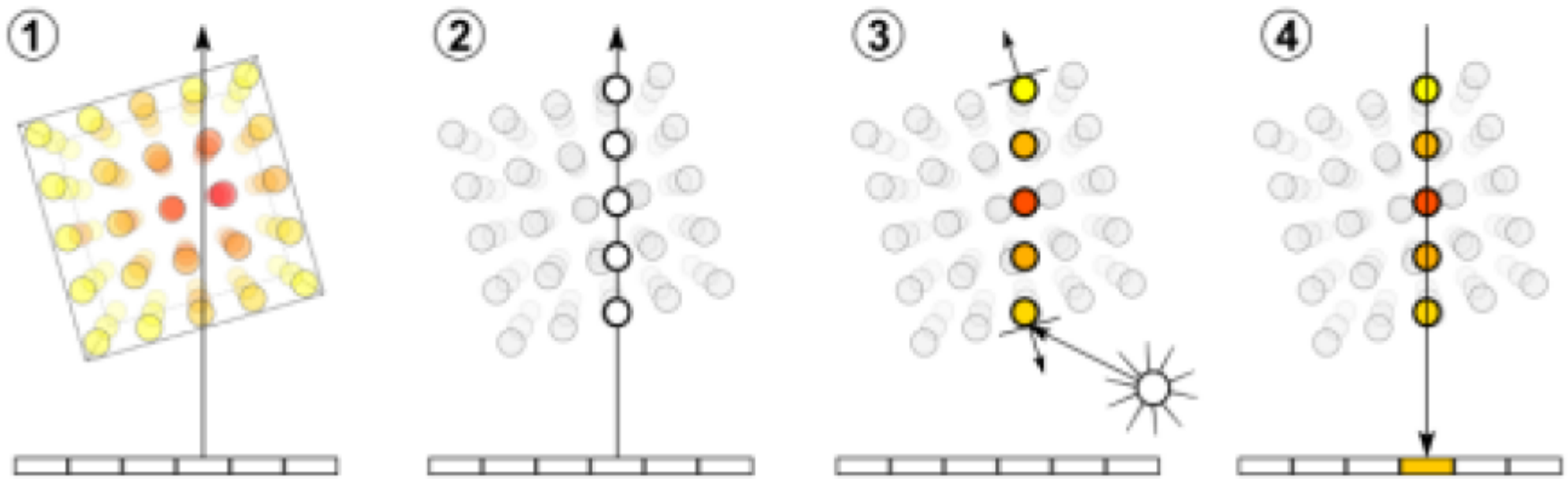


1. Transform volume bounding box vertices using the modelview matrix.
2. Compute view orthogonal sampling planes, based on:
   - Distance between min and max *z* of bounding box verts
   - Equidistant spacing scaled by voxel size and sampling rate.
3. For each plane
   a) Test for intersections with bounding box. Generate a **proxy polygon** (upto 6 sides).
   b) Tessellate proxy polygon into triangles and add the resulting vertices to the output vertex array
   c) Generate texture coordinates for each triangle vertex

Volume Ray Casting

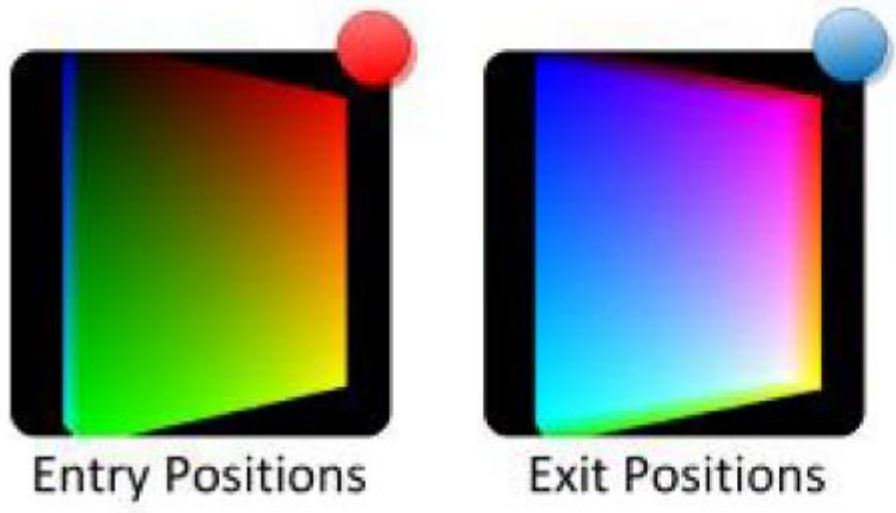
- **Ray casting.** For each pixel of the final image, cast eye ray through the volume (usually enclosed within a *bounding box* used to intersect the ray and volume).
- **Point Sampling.** equidistant *sampling points* or *samples* are selected along ray. Sampling points usually will be located in between voxels so tri-linearly interpolate values from surrounding voxels.
- **Point Shading.** For each sampling point either:
  - Apply some colour based on sampled value and a transfer function: classical Direct Volume Rendering (DVR)

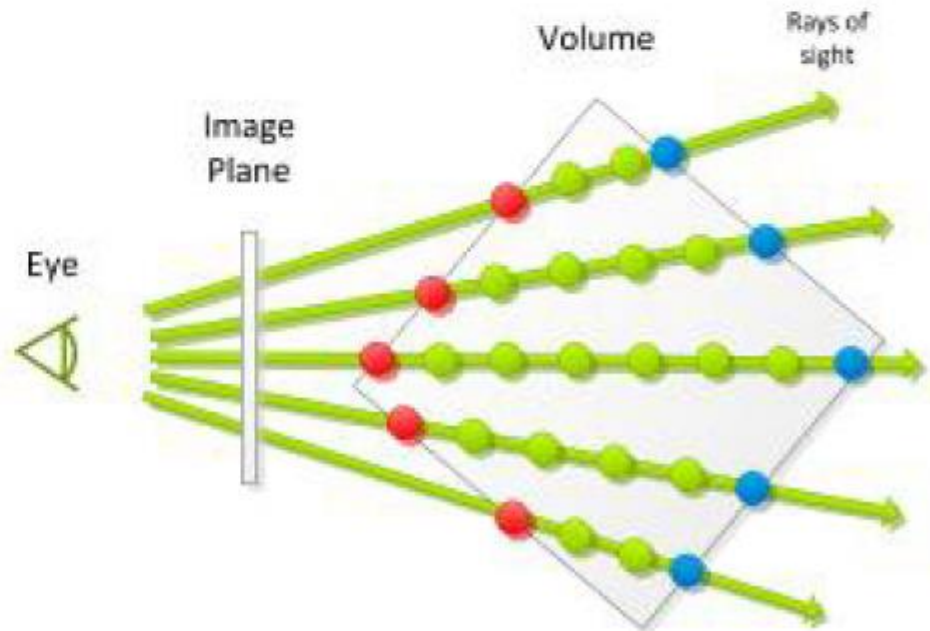
  OR
  - Calculate gradient (orientation of local surfaces) and calculate illumination using e.g. Phong
- **Compositing.** Combine shaded samples to get the final colour value for the ray.



GPU Ray Marching

- Compute volume Entry Position
- Compute ray of sight direction
- While in Volume
  - Lookup data value at ray position
  - Accumulate Colour and Opacity





High-Level Drawing Functions

```
// Create a render texure in which it is possible to render
RenderTexture(int width, int height,int depth, RenderTextureFormat format);
RenderTexture.GetTemporary(int width, int height, int depth)

// Static - Sets current render target for rendering to an offline texture
Graphics.SetRenderTarget(RenderTexture rt);

// Static - Copies source texture into destination render texture with a shader
Graphics.Blit(Texture source, RenderTexture dest,  Material mat, int pass = -1);
```

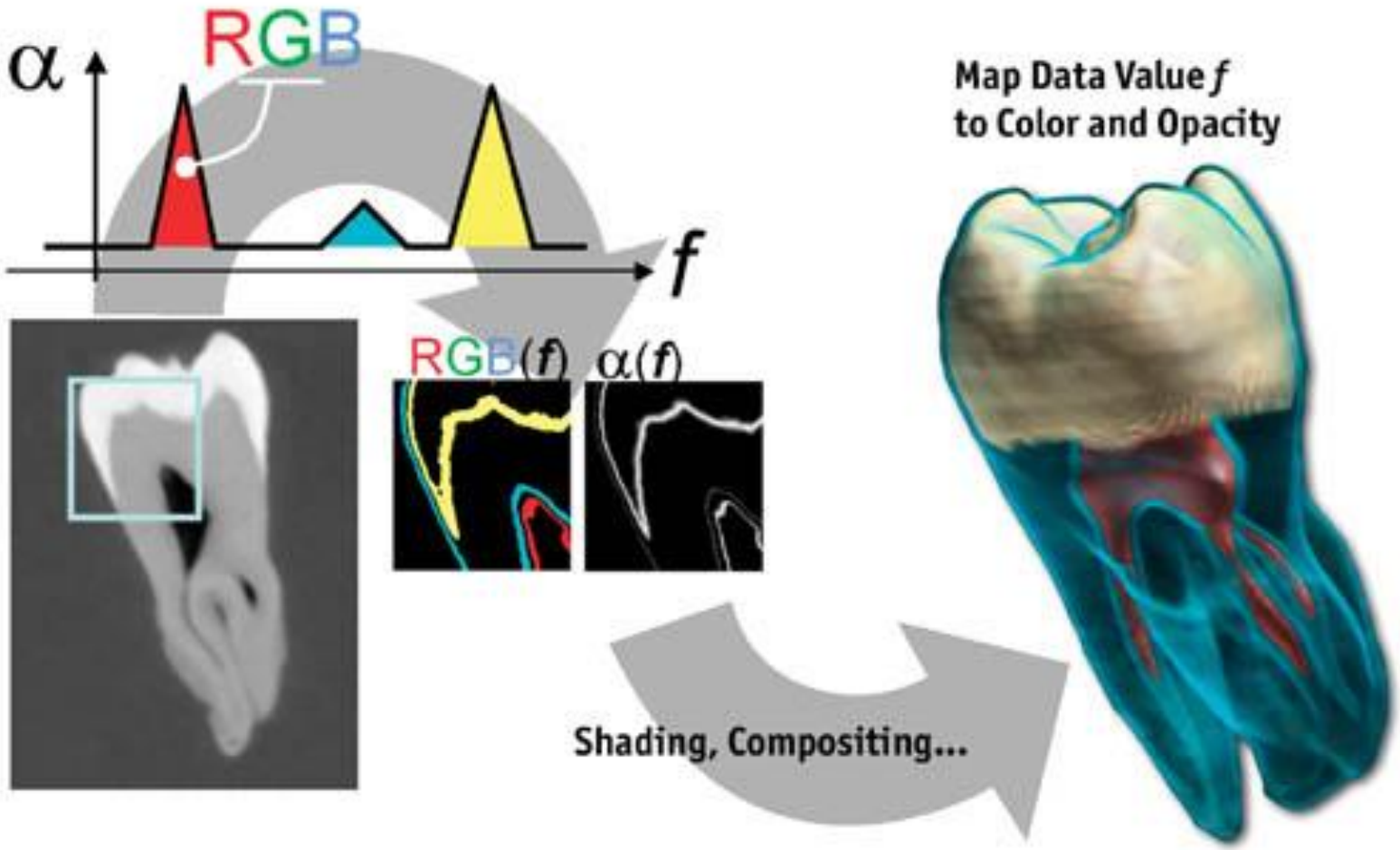
GPU Ray Marching in Unity

- Demo alpha blending

Transfer Function

- Transparent scalar field is difficult to understand (information overload)
- Map scalar values to colours or opacity
  - Interpretive rendering
  - Allows user to choose which levels are more visible OR attach colors/alpha to specific voxel levels
  - Visualisation: make visual data easier to understand
  - Less important for games

Transfer Function

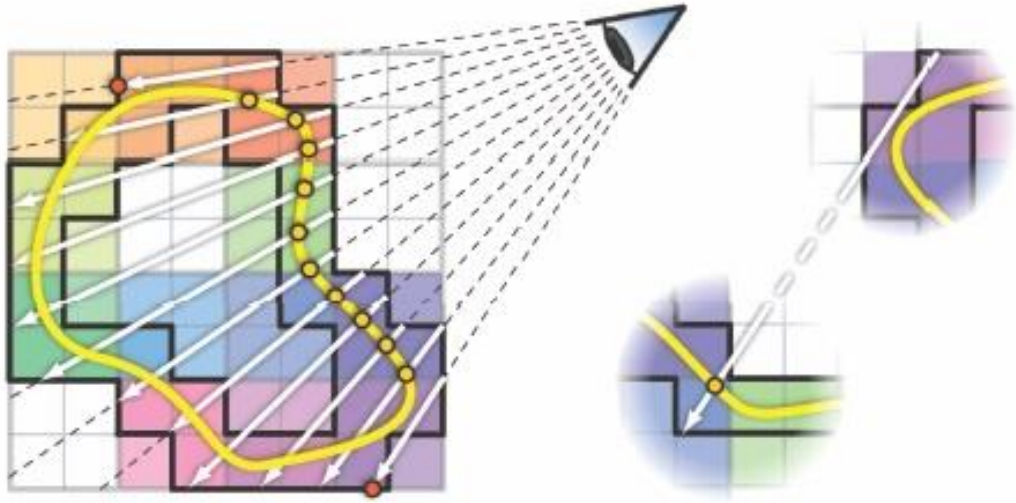


GPU Ray Marching in Unity

- Demo transfer function

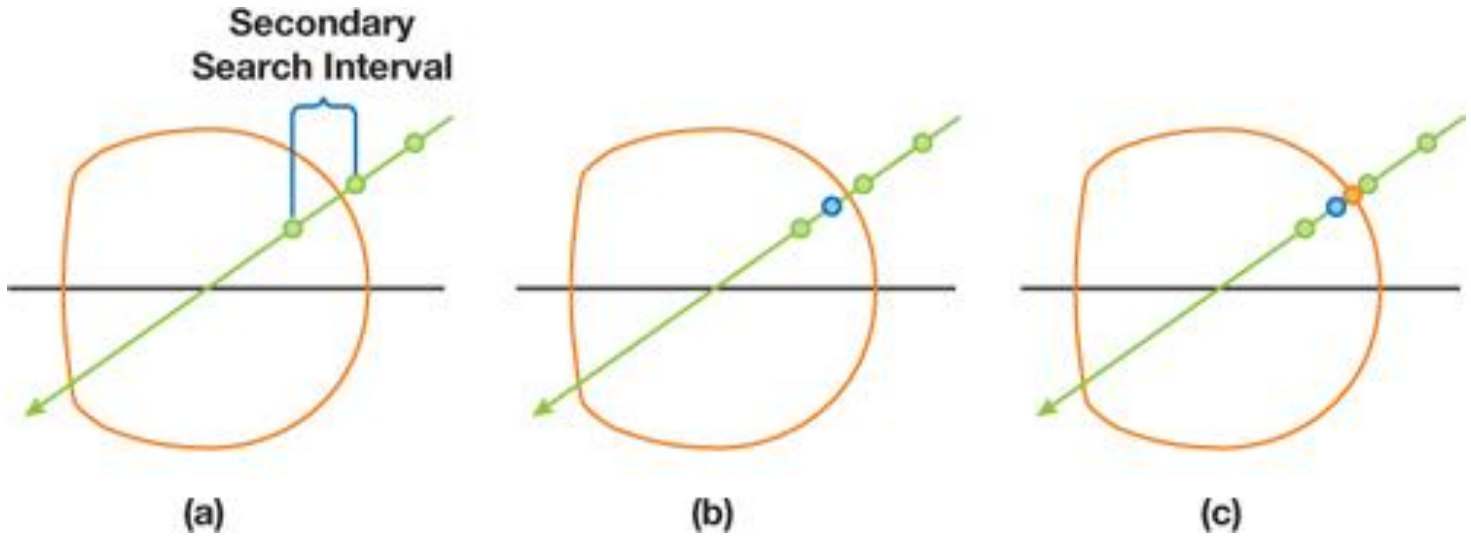
Iso-surface rendering

- Do not aggregate value along the ray
- Stop ray marching at a given intensity value (iso)
- Faster than full traveersal & allows optimizations
- Also used to simulate fluid effects



Linear/Binary ray marching

- Too much small steps = too much time
- Split the ray casting in to parts
  - Linear sampling first, with large steps
  - Binary sampling afterwards to find the exact surface position

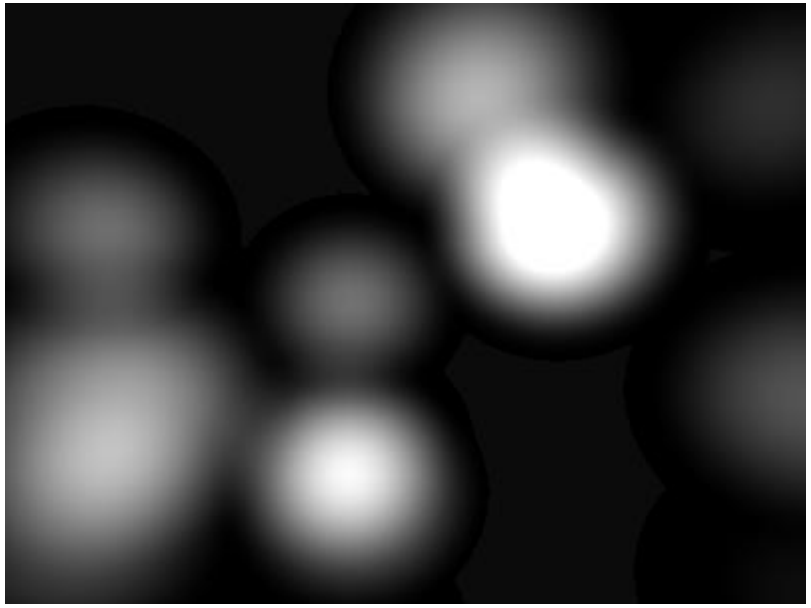


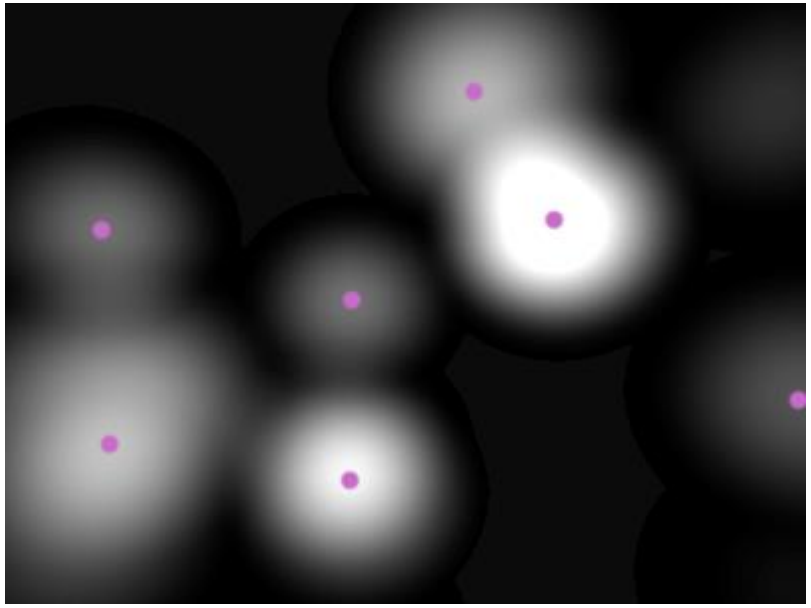
GPU Ray Marching in Unity

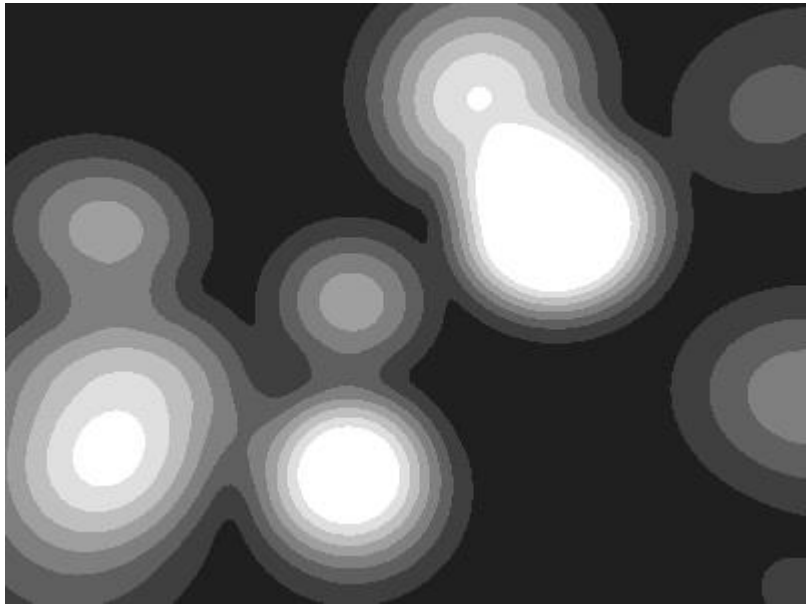
- Demo isosurface

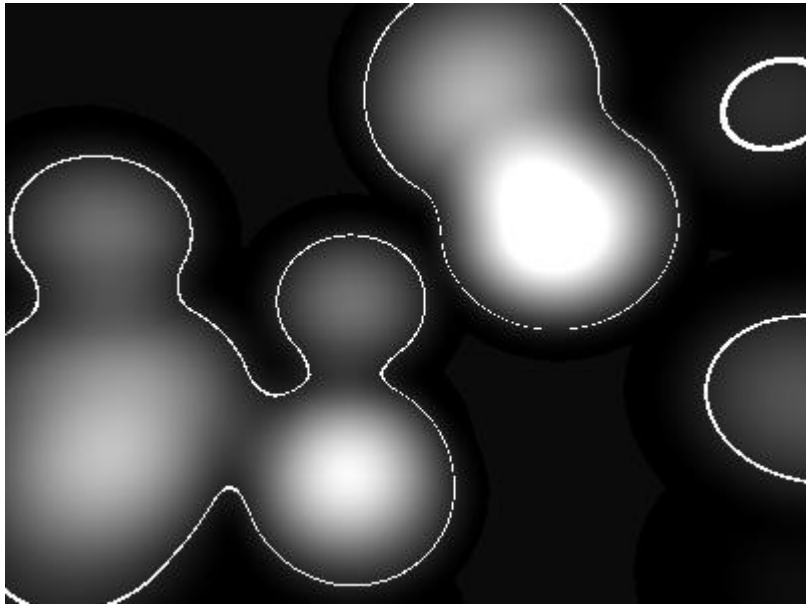
Metaballs

- Iso-surface rendering of density field
- Used in visualization and games
- Density field can be discretized in a texture
- Density threshold arbitrarily chosen









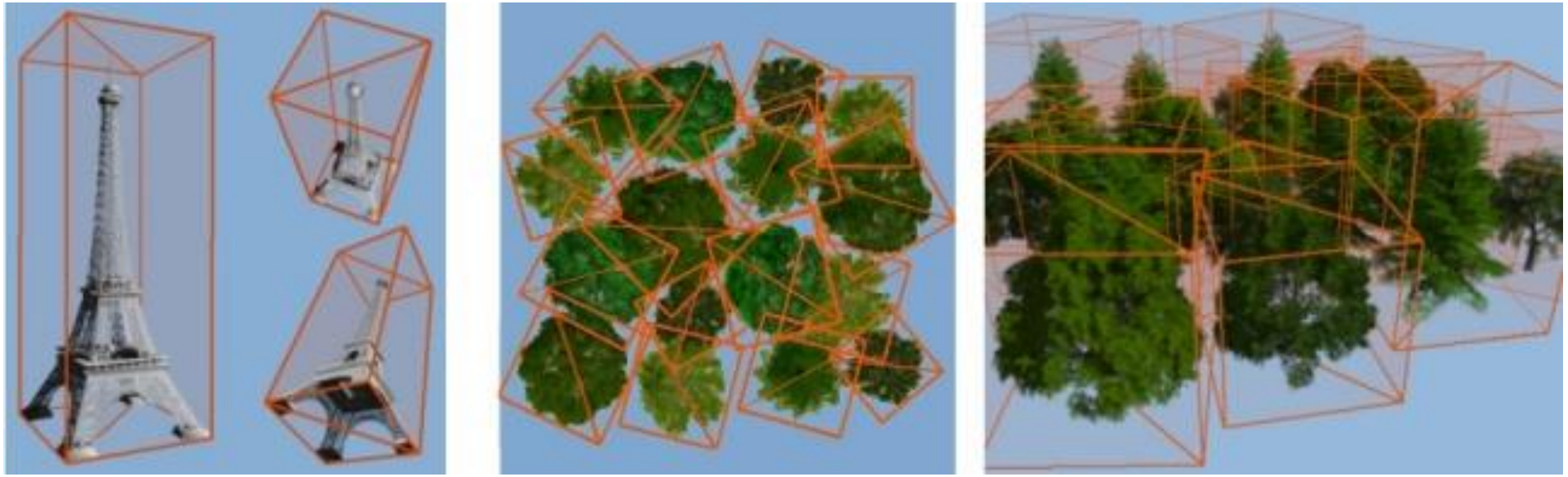
[Metaballs and Isosurfaces with OpenGL - YouTube]

GPU Ray Marching in Unity

- Demo metaballs + volume construction

Volumetric Billboards

- Uses 3D textures instead of traditional 2D for billboards
  - Full-parallax effect, without artifacts
  - Combine with mip-mapping for Level-of-Detail
  - Low amount of vertex processing

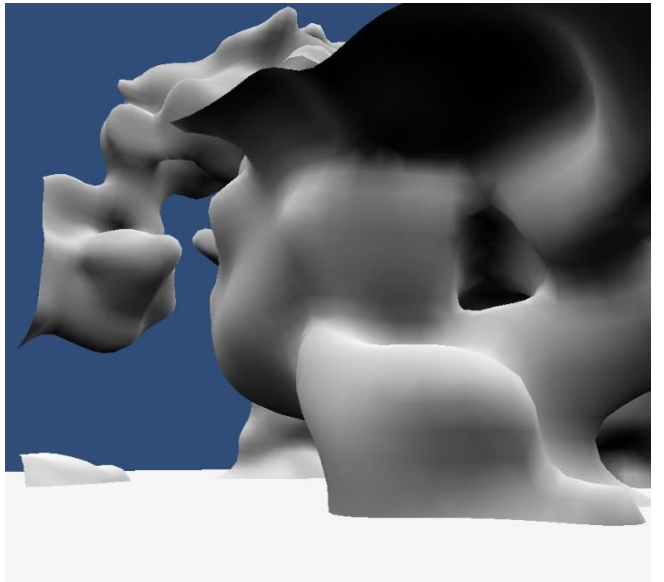


GPU Ray Marching in Unity

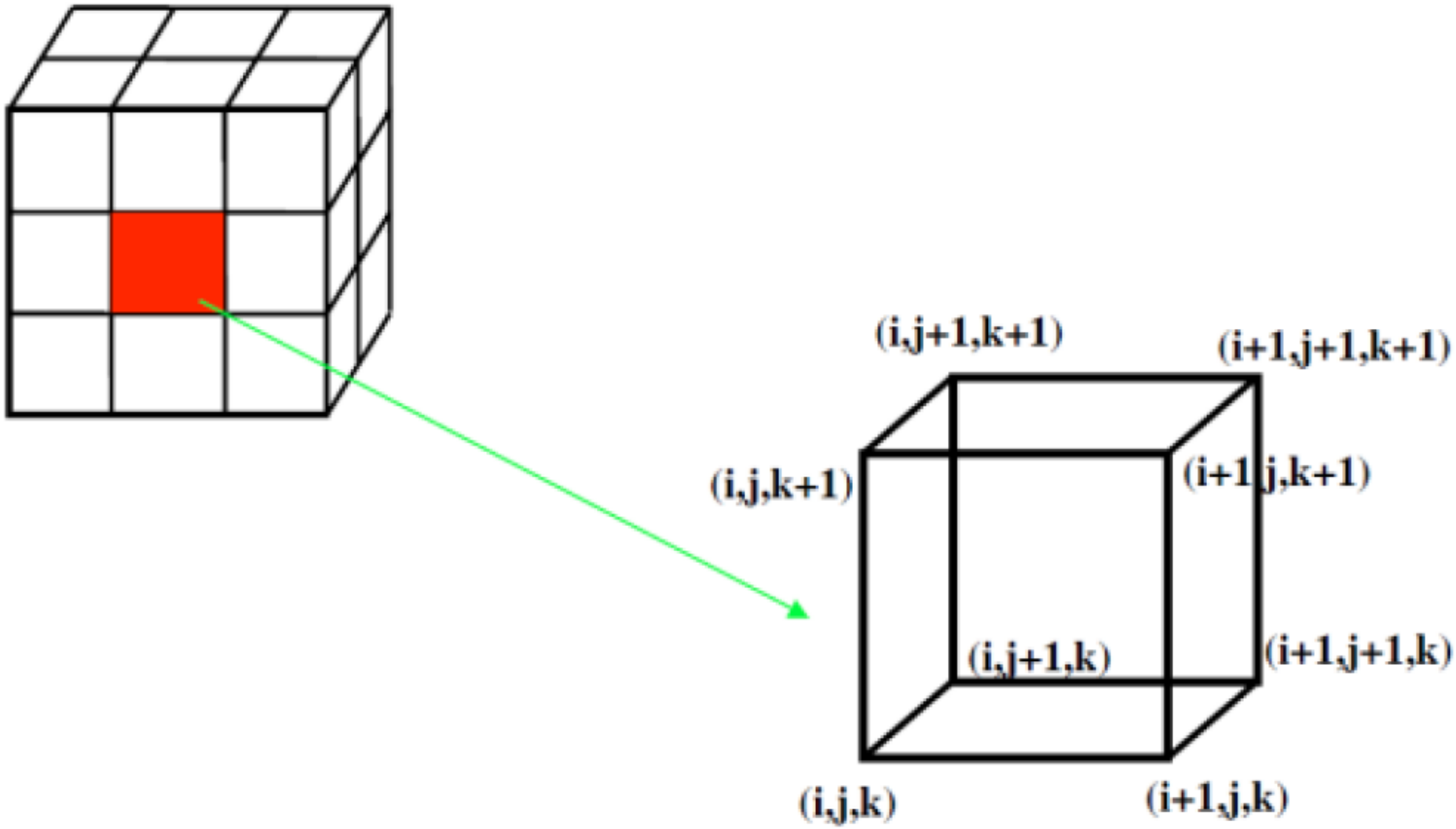
- Demo instanced volumes

Indirect Volume Rendering

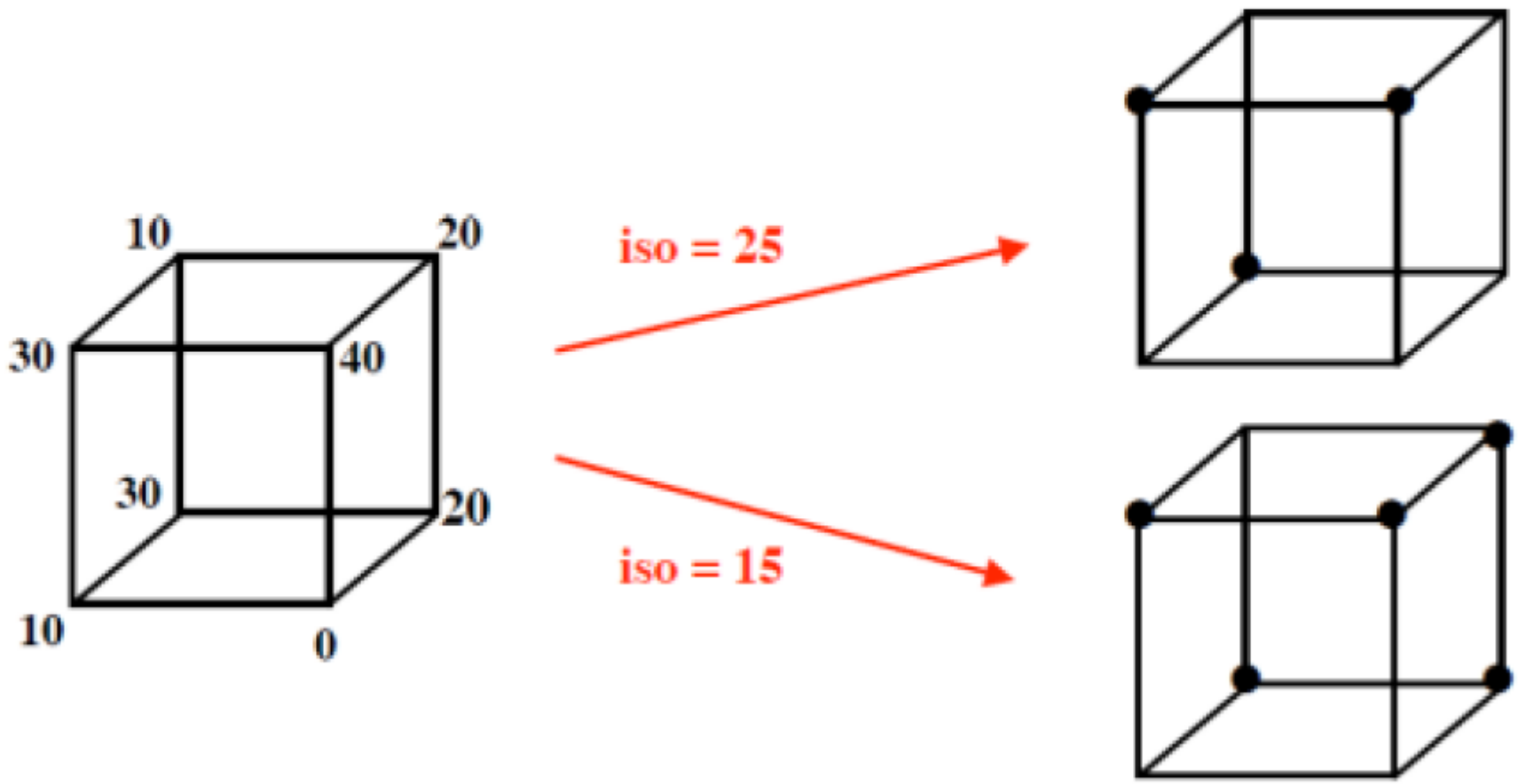
- There are some benefits to surface based techniques when it comes to rendering
  - More traditional pipeline optimizations
  - Accurate reflections
  - Clear boundary representation
- Indirect Volume Rendering techniques first extract one or more iso-surfaces from the volume data
  - Alternatively render one iso-surface and blend it with DVR
- Either:
  - Implicitly/on-the fly
  - Iso-surface mesh extraction



Marching Cubes



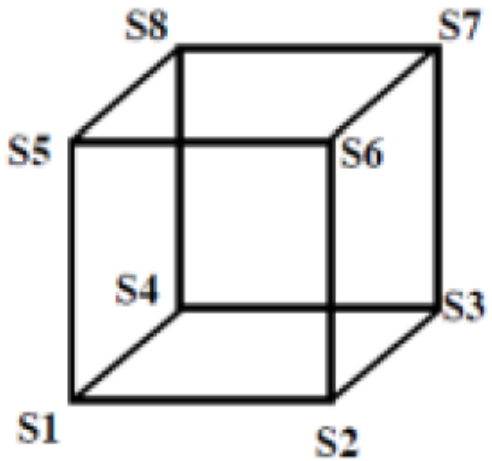
Marching Cubes

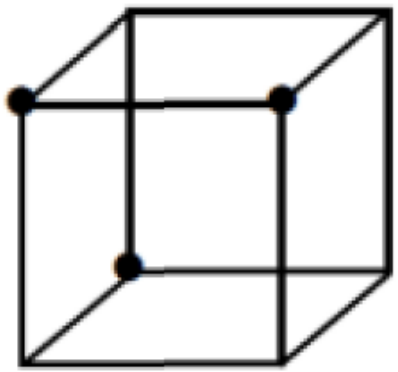


Marching Cubes

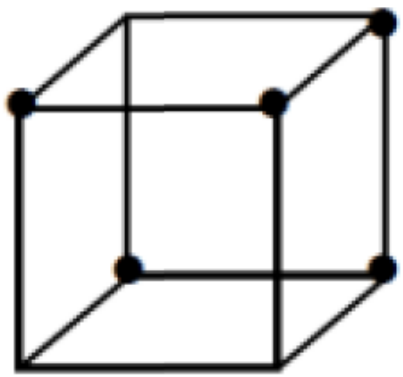
Marked vertex by ● = inside = 1

Unmarked vertex = outside = 0





00011100



?

index | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8

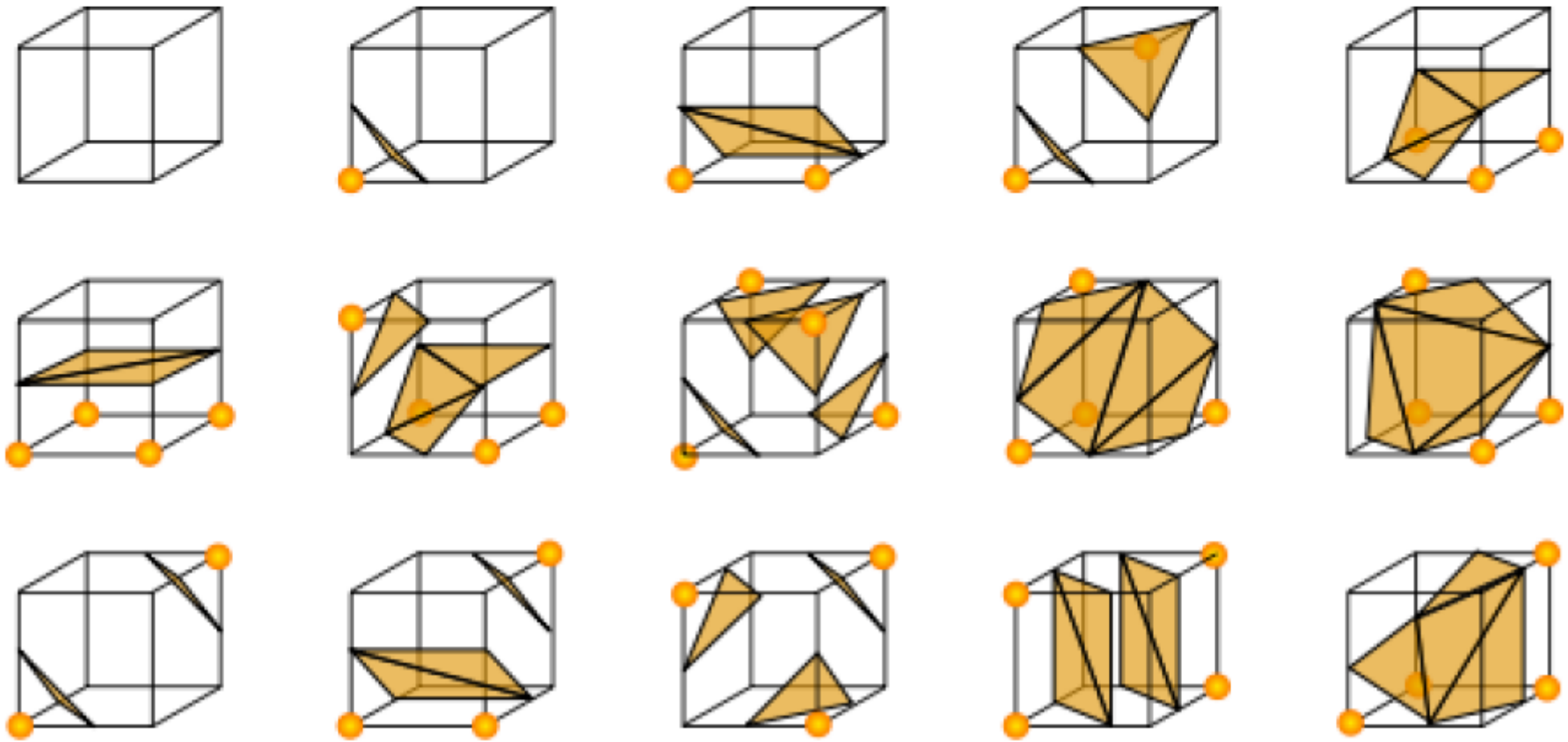
or

index | S8 | S7 | S6 | S5 | S4 | S3 | S2 | S1

Forms the bits of a binary number between 0 and 255 for an 8-vertex cube

Marching Cubes

- Removing redundant cases e.g. completementary and rotational symmetries: each voxel is identified as one of 15 cases:

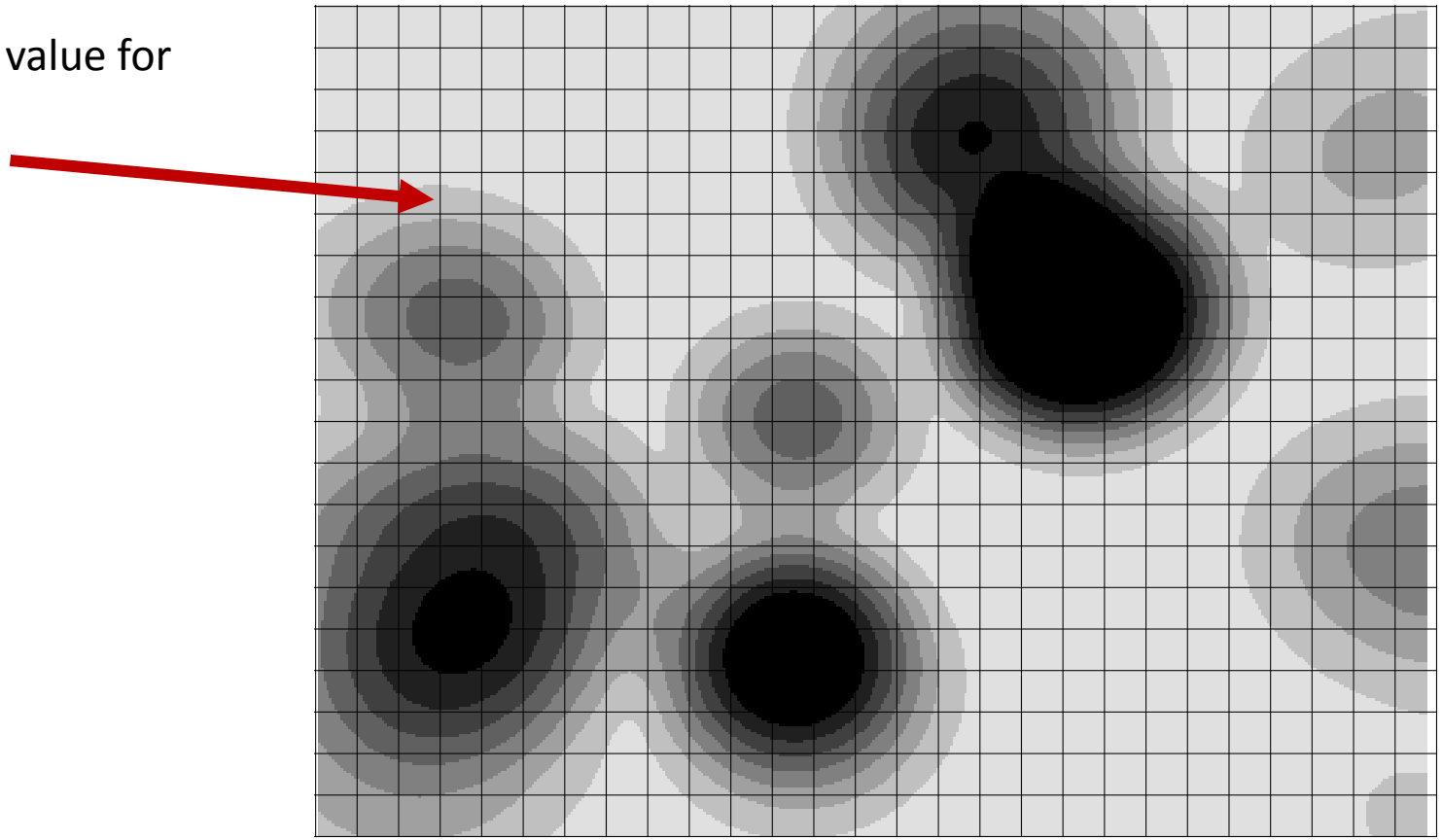


Marching Cubes on the GPU

- http://scrawkblog.com/2014/10/16/marching-cubes-on-the-gpu-in-unity/

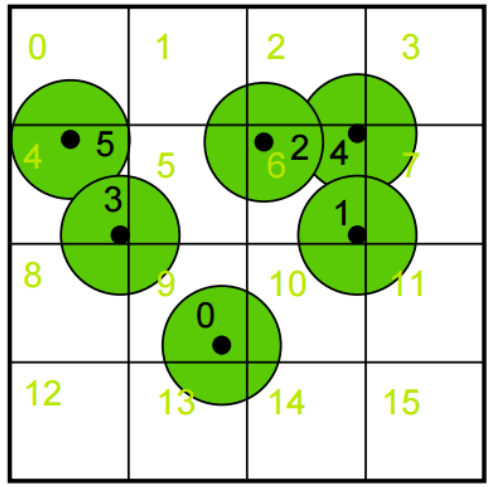
Real-time Metaballs

Find density value for each cell



Real-time Metaballs – Option 1

- For each voxel, evaluate each atoms and see if the density distance field overlaps with the cell
- Complexity increases greatly with the number of atoms
- Possible to use efficient neighbouring search to only sample atoms in neighboorhood of the cell, drawbacks complex to implement and to somewhat the most efficient approach

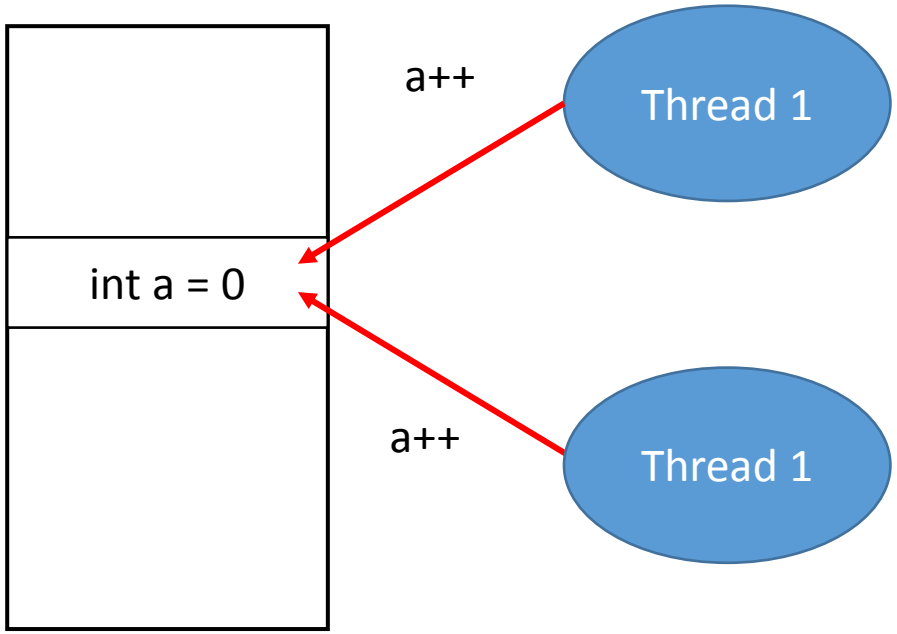


Real-Time Metaballs – Option 2

- For each atoms, update the density in the neighbouring grid cells
- Advantage: no need for neighbooring search
- Drawback: we must prevent concurrency issues

GPGPU Computing - Atomics

- In the parallel world, instructions are computed simulateneously



?

GPGPU Computing - Atomics

- Use atomics to prevent threads to access resources simultaneously

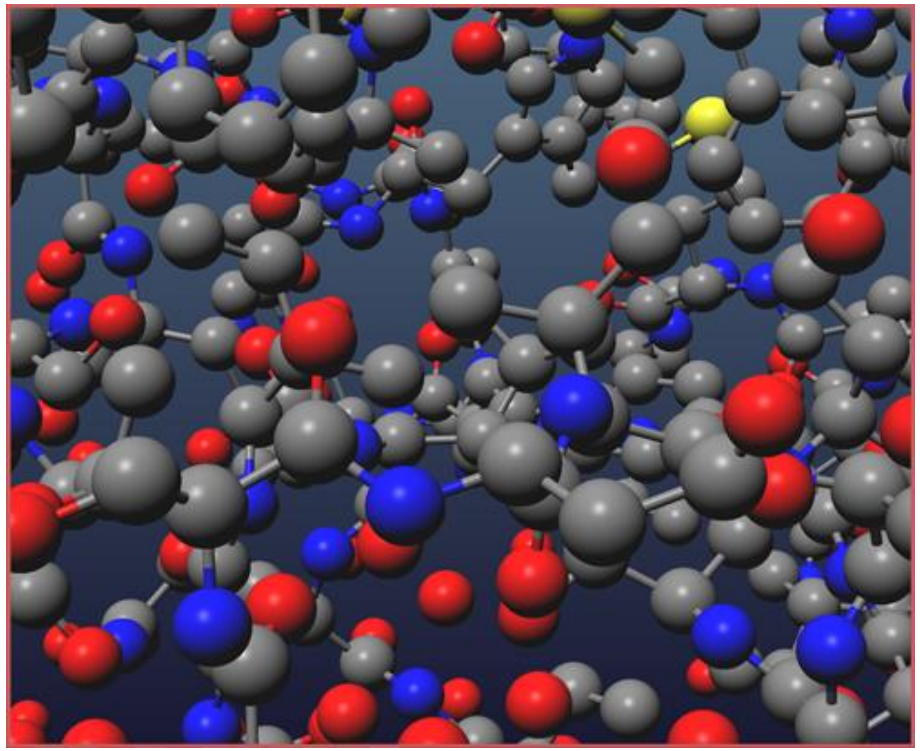
```
void InterlockedAdd( in  R dest, in  T value, out T original_value );

void InterlockedMin( in  R dest, in  T value, out T original_value );

void InterlockedCompareStore( in R dest, in T compare_value, in T value );

void InterlockedMax( in  R dest, in  T value, out T original_value );

…
```

Other problems

- Atomic is only available for integers
- Density sampling required floats
- Solution: use two textures
  - One for gathering density
  - One for the rendering
  - Copy density from int to floats
- Nasty solution (increase memory footprint)
- Alternative solution: implement custom atomic addition for floats

Demo

Sphere impostors

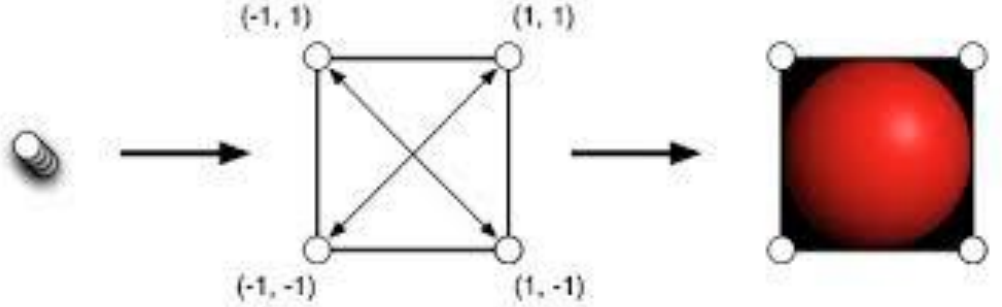


- Commonly used in molecular rendering
- Also use for other shapes like „sticks“
- „Standard“ billboards are simply 2D textures facing the camera
- No interaction with light & flat feeling

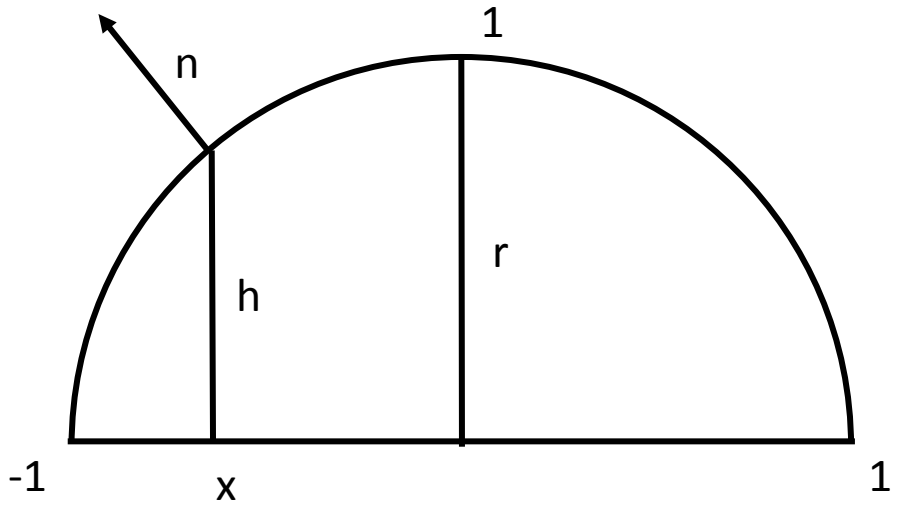
Sphere impostors

- An alternative is to raytrace the sphere in the fragment shader
- Custom lighting + custom depth output

Principle



- Use custom UV to solve the sphere equation
- First step is to cut a hole in the billboard via discard
- Second is to compute the normal vector of the sphere
- Finally output correct depth for blending with other spheres



Demo

References

- Simon Green “Volume Rendering for Games” nVidia GDC 2005 presentation
- http://developer.nvidia.com/object/gdc_2005_presentations.html
- Ikits et al “Volume Rendering Techniques” GPU Gems 2. Chapter 39
- http://http.developer.nvidia.com/GPUGems/gpugems_ch39.html

Follow up topics

- Realtime volume convolution
- Realtime volume update (metaballs)
- Realtime marching cubes
- Post processing