Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

PROJECT

3D Augmented Reality

A.Y. 2022/2023

3D Meshes, Point Clouds & Shaders

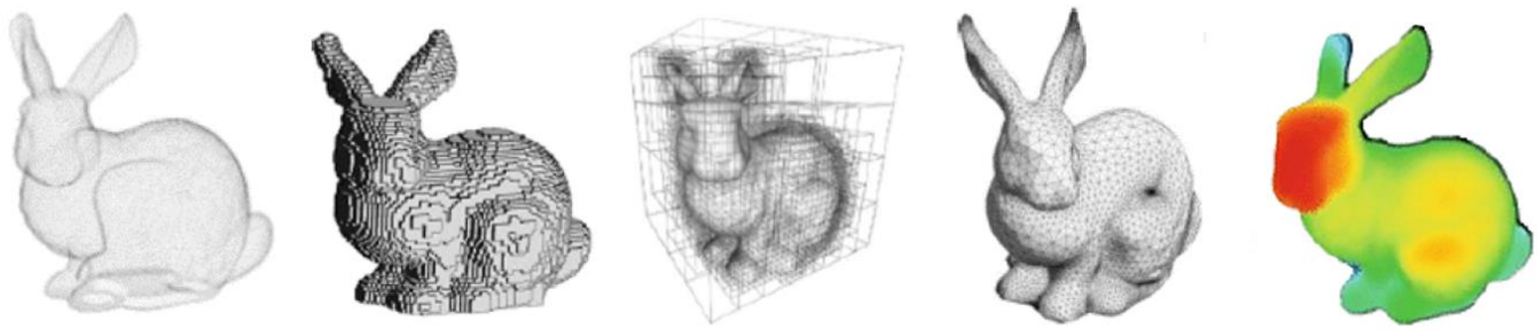
LAB experience 4

Elena Camuffo

elena.camuffo@phd.unipd.it

Represent the Three-Dimensional Data

- Different **3D data structures** have been proposed to represent 3D data in an efficient way.



Point Cloud **Voxels** **Octrees** **Polygon Mesh** **Depth Map**

Volumetric

2.5D

Sparse structures.
Mainly used in Deep Learning.
Obtained from acquisitions.

Connected structures: collection of vertices, edges and faces that defines the shape of a polyhedral object.
Manly used in Computer Graphics, AR/VR.

https://medium.com/@elenacamuffo97/recent-advancements-in-learning-algorithms-for-point-clouds-an-updated-overview-35eabf511183

3D Meshes

- A **mesh** is a collection of *vertices*, *edges* and *faces* that defines the shape of a polyhedral object. Meshes can be of different types, e.g., *Vertex-Vertex (VV), Face-Vertex* (most common) or *Winged-edge,* depending on which elements are explicated inside the mesh.

$$M = \bigcup_{v_i \in R^3} (\boldsymbol{v_i}, \boldsymbol{e_j}, \boldsymbol{f_k}), \qquad v_i = (x_i, y_i, z_i) \qquad f_j = (v_m, v_n, v_p) \qquad e_j = (v_m, v_n)$$

composing elements

single vertex

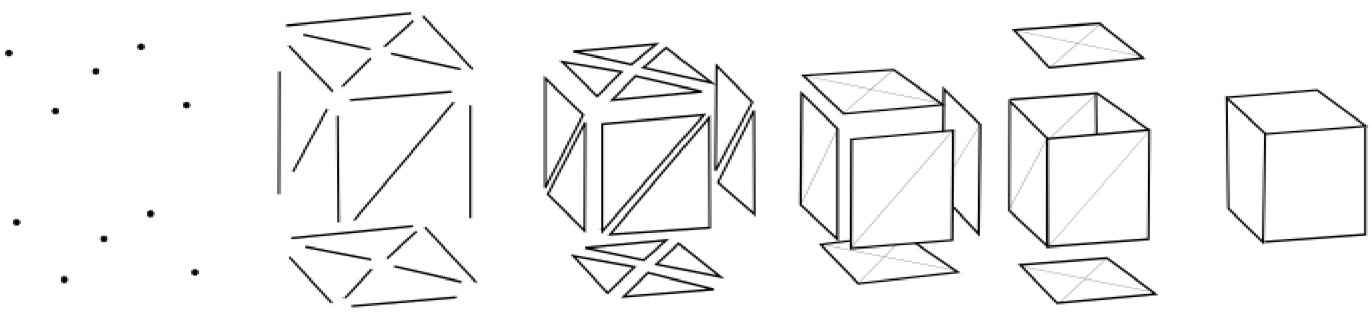
single face

single edge

Vertex-Vertex

Face-Vertex

Winged-Edge



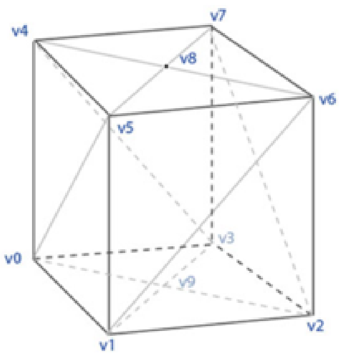
vertices edges faces polygons surfaces

3D Meshes

- Most mesh formats also support some form of **UV coordinates** (2D representation of the mesh *"unfolded"*). It is also possible for meshes to contain **vertex attribute** information such as color, tangent vectors, weight maps to control animation, etc.

Vertex List

v0	0,0,0	v1 v5 v4 v3 v9
v1	1,0,0	v2 v6 v5 v0 v9
v2	1,1,0	v3 v7 v6 v1 v9
v3	0,1,0	v2 v6 v7 v4 v9
v4	0,0,1	v5 v0 v3 v7 v8
v5	1,0,1	v6 v1 v0 v4 v8
v6	1,1,1	v7 v2 v1 v5 v8
v7	0,1,1	v4 v3 v2 v6 v8
v8	.5,.5,1	v4 v5 v6 v7
v9	.5,.5,0	v0 v1 v2 v3

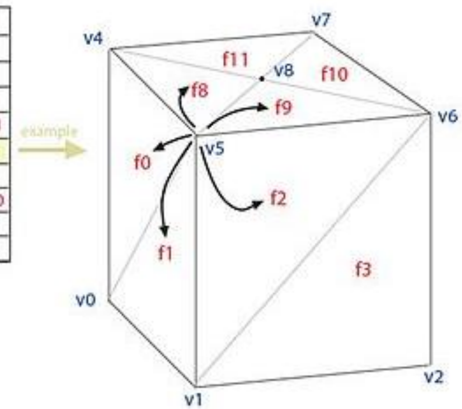


Face List

f0	v0 v4 v5
f1	v0 v5 v1
f2	v1 v5 v6
f3	v1 v6 v2
f4	v2 v6 v7
f5	v2 v7 v3
f6	v3 v7 v4
f7	v3 v4 v0
f8	v8 v5 v4
f9	v8 v6 v5
f10	v8 v7 v6
f11	v8 v4 v7
f12	v9 v5 v4
f13	v9 v6 v5
f14	v9 v7 v6
f15	v9 v4 v7

Vertex List

v0	0,0,0	f0 f1 f12 f15 f7
v1	1,0,0	f2 f3 f13 f12 f1
v2	1,1,0	f4 f5 f14 f13 f3
v3	0,1,0	f6 f7 f15 f14 f5
v4	0,0,1	f6 f7 f0 f8 f11
v5	1,0,1	f0 f1 f2 f9 f8
v6	1,1,1	f2 f3 f4 f10 f9
v7	0,1,1	f4 f5 f6 f11 f10
v8	.5,.5,0	f8 f9 f10 f11
v9	.5,.5,1	f12 13 14 15



Vertex-vertex meshes represent an object as a set of vertices connected to other vertices. This is the *simplest representation*, but not widely used since the face and edge *information is implicit*.

Face-vertex meshes represent an object as a set of faces and a set of vertices. This is the *most widely used mesh representation*, being the input typically accepted by modern graphics hardware.

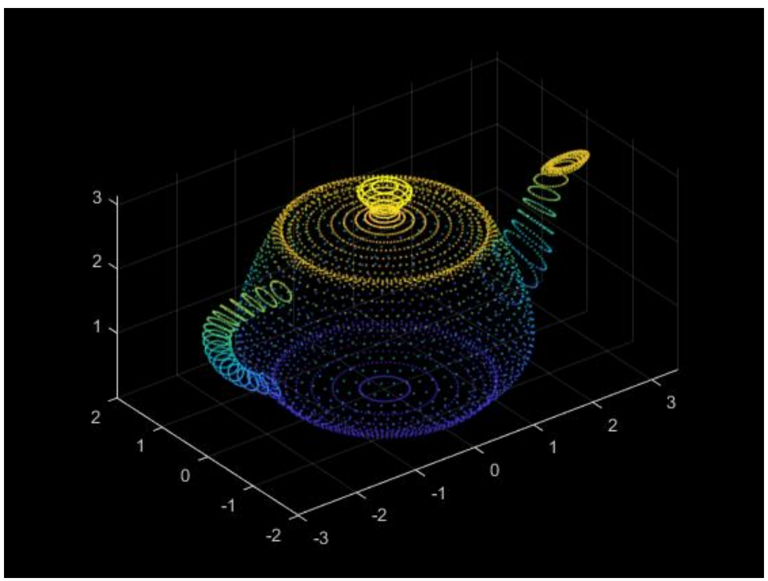
Point Clouds

- A **point cloud** is a *set of data points in the 3D space*. Each point is spatially defined by a triplet of coordinates and a combination of such points can be used to describe the **geometry** of an object or the complete scene.

$$P = \bigcup_{p_i \in R^3} p_i\,, \qquad p_i = (x_i, y_i, z_i)$$

single point

point cloud



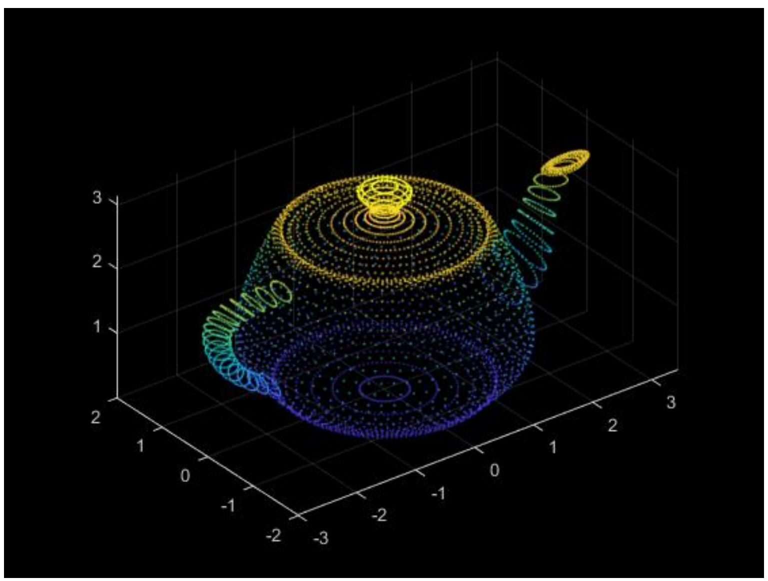
- They are easily obtained as a *map of the external surface of objects* through reality capture devices.
- Depending on the device they can assume different shapes.

Point Clouds

- The properties relative to each of the point of a point cloud can be specified adding a series of **attributes**, in addition to the 3D coordinates.

$$P = \bigcup_{p_i \in R^3} p_i\,, \qquad \underbrace{p_i}_{\text{single point}} = (x_i, y_i, z_i, \overbrace{r, g, b}^{\text{color}}, \underbrace{a}_{\text{intensity}}, \overbrace{l}^{\text{label}}, \dots)$$

point cloud



- Attributes can define:
  - Properties related to the object's **surface** like *color or reflectivity*
  - Properties related to the **content** like *labels* (a number defining a specific object or class of objects to which the point belongs).

Dealing with Point Cloud Data

- The example provides Point Clouds in simple **txt** format. However, Point Clouds can be stored in many other different formats, e.g., **CSV**, and the syntax depends on the specific format.
- The most common formats are **OFF, OBJ, PLY, FBX** and **binary ASCII.**

- In many cases Point Clouds are difficult data structures to deal with, and also not suitable for some purposes, so that they must be converted to **meshes** and/or **voxels.**

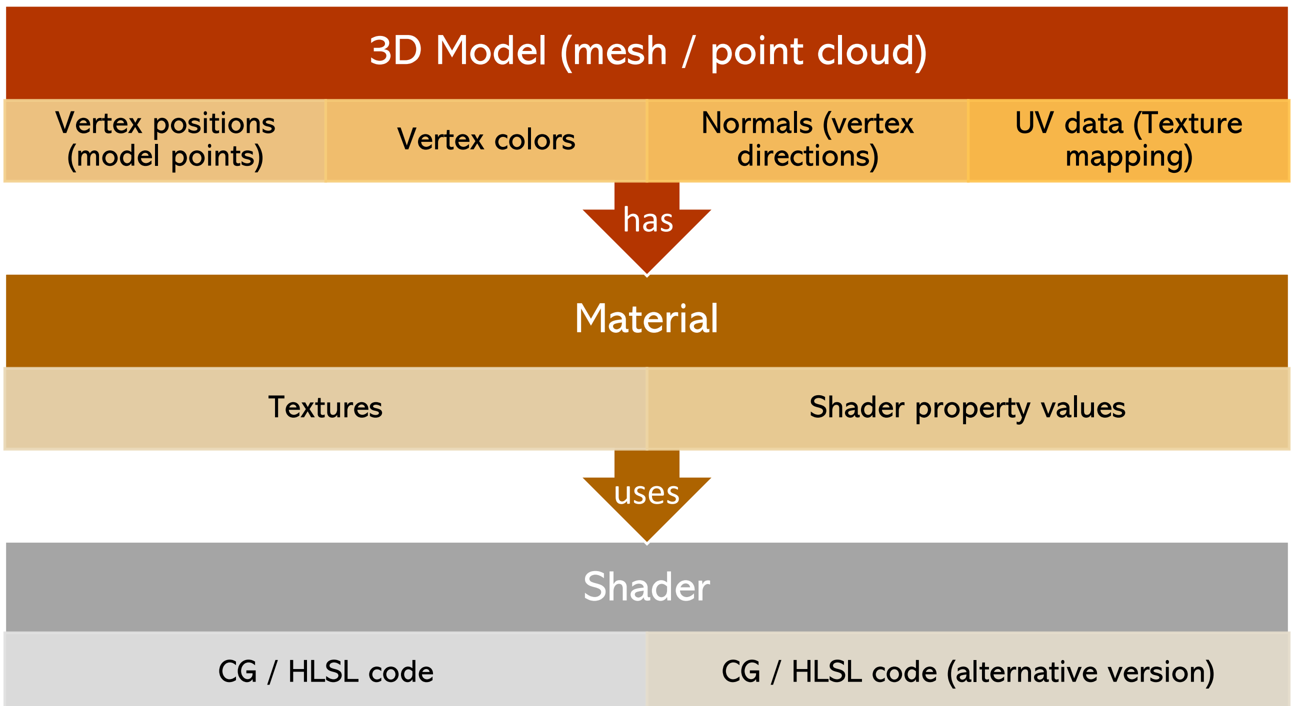
	Voxel	Point cloud	Polygon mesh
Memory efficiency	Poor	Not good	Good
Textures	Not good	No	Yes
For neural networks	Easy	Not easy	Not easy

- They are complex to be rendered. To produce any visual effect, **Shaders** can be employed.

Useful repository that provides tools for Point Cloud processing in **Python**: https://github.com/Dan8991/PCutils ***[Credits: Daniele Mari]***

Rendering Process

A general 3D model has some properties and can be associated with a GameObject with a ***MeshRenderer*** component that allows assigning it a material. The Material uses the Shader that takes data from the model and material to draw pixels to the screen based on it’s CG/HLSL code.



Shaders in Unity

In Unity, shaders are divided into three broad categories. You use each category for different things, and work with them differently:

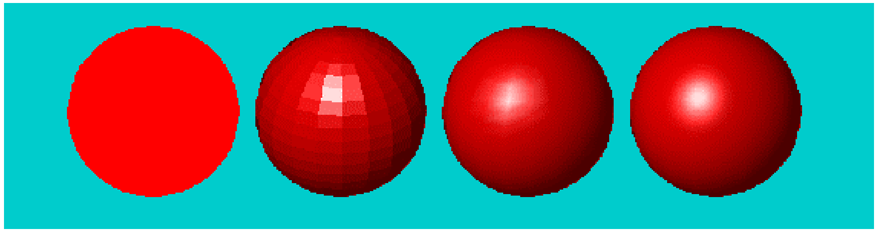
- **Shaders** that are **part of the graphics pipeline** are the most common type of shader. They perform calculations that *determine the color of pixels on the screen*. In Unity, you usually work with this type of shader by using Shader objects.
- **Compute shaders** perform calculations parallelizable *on the GPUs*, outside of the regular graphics pipeline.
- **Ray tracing** shaders perform calculations related to ray tracing.

You use Shader objects with materials to determine the **appearance of your scene**

Shaders in Unity

- The most common shaders are computer programs **used to do shading during the rendering processing pipeline**: the production of appropriate levels of light, darkness, and color within an image, or, in the modern era, also to produce special effects or do video post-processing (like image effects).

- The main categories of Shaders are:
  - **Surface Shaders** are a code generation approach that makes it much easier to write lit shaders than using low level vertex/pixel shader programs.
  - **Unlit Shaders** do not interact with Unity Lights, useful for special effects.
  - **Image Effect Shaders** are typically a post-processing effect that read the source image, do some calculations on it, and render the result into the provided destination.



none flat gouraud phong

Shaders in Unity

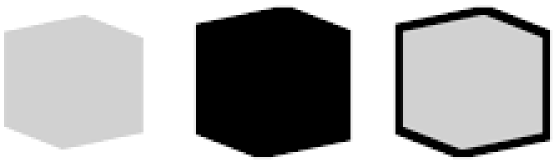
- A *shader asset* is an asset in your Unity project that defines a **Shader object**. It is a text file with a *.shader extension*. It contains *shader code*. There are two parts of shaders: *vertex* and *fragment*.

```
// Shader for Vertex Color in the point
// cloud data
Shader "Custom/VertexColor"
{
Properties
    {
     _PointSize("PointSize", Float) = 10
    }
    SubShader
    {
    Pass
        {
        LOD 200 // level of detail

        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        // vertex shader inputs
        struct VertexInput
        {
            float4 v : POSITION;
            float4 color: COLOR;
        };

        // vertex shader outputs
        struct VertexOutput
        {
            float4 pos : SV_POSITION;
            float size : PSIZE;
            float4 col : COLOR;
        };
```

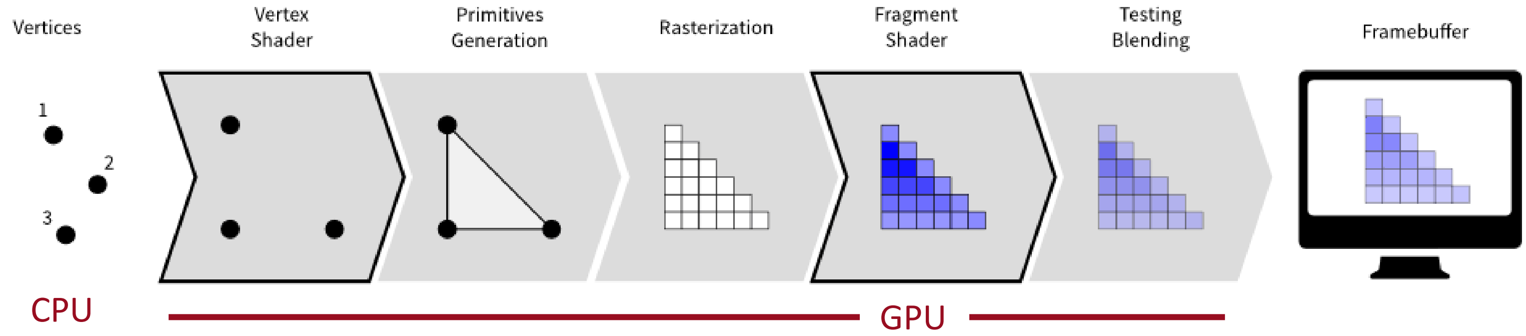
Shader
contains the name of the shader as a string and the whole shader.

Properties
contains **shader variables** (textures, colors etc.) that will be saved as **part of the Material** and displayed in the material inspector.

Pass
represents an execution of the vertex and fragment code for the same object rendered with the material of the shader.



Shaders in Unity

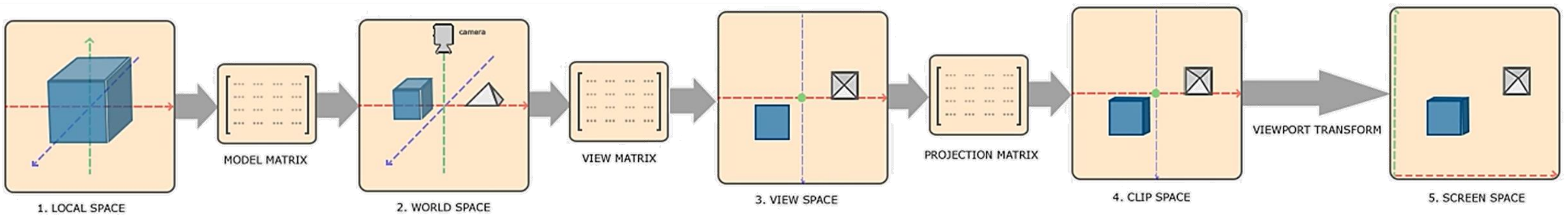



```
        float _PointSize;

        // vertex shader
        VertexOutput vert(VertexInput v)
        {

            VertexOutput o;
            o.pos = UnityObjectToClipPos(v.v);
            o.size = _PointSize;
            o.col = v.color;

            return o;
        }

        // fragment shader
        float4 frag(VertexOutput o) : COLOR
        {
            return o.col;
        }

        ENDCG
        }
    }
}
```

The **Vertex Shader** is a program that runs on each vertex of the 3D model. Quite often it does not do anything particularly interesting, e.g., transform vertex position to rasterize the object on screen.

The **Fragment Shader** is a program that runs on every pixel that object occupies on-screen and is usually used to *calculate and output the color* of each pixel.

Shaders in Unity




```
        float _PointSize;

        // vertex shader
        VertexOutput vert(VertexInput v)
        {

            VertexOutput o;
            o.pos = UnityObjectToClipPos(v.v);
            o.size = _PointSize;
            o.col = v.color;

            return o;
        }

        // fragment shader
        float4 frag(VertexOutput o) : COLOR
        {
            return o.col;
        }

        ENDCG
        }
    }
}
```

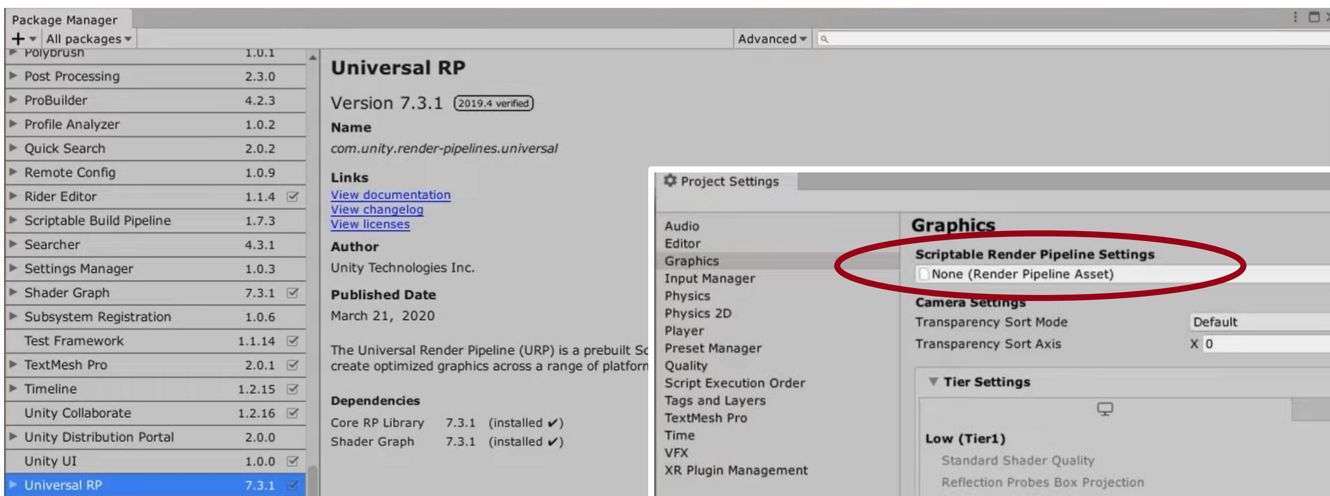
_PointSize
Variable defining size of points (rendered as spherical meshes).

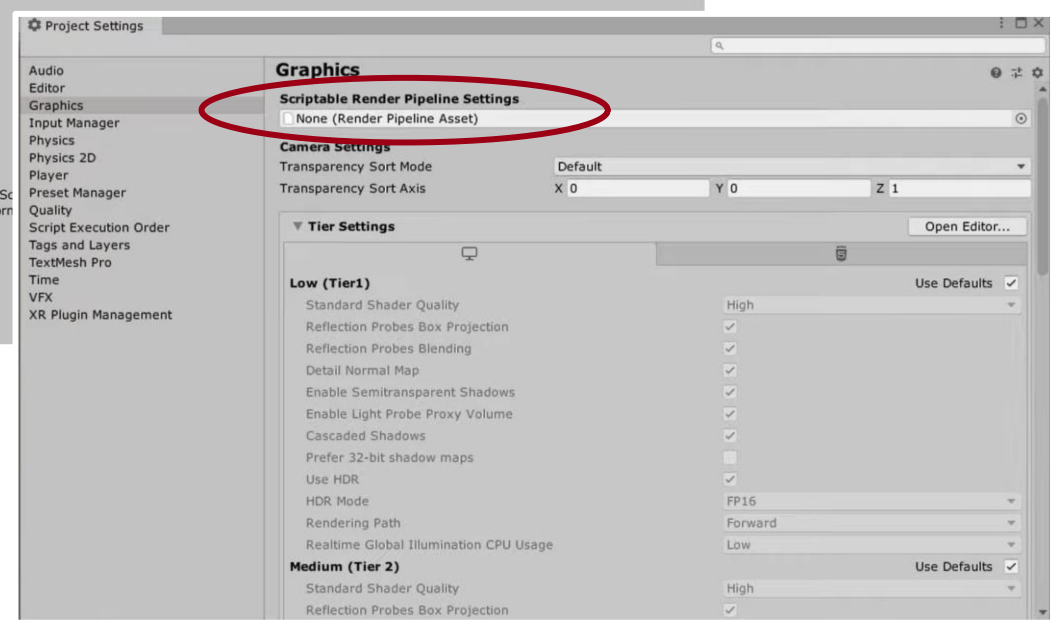
UnityObjectToClipPose()
defines the set of transformations that are needed to map the 3D object from its local space to the screen space.

There are also tools to build shaders in a simpler way, based on nodes:
https://assetstore.unity.com/packages/vfx/shaders/shader-graph-easing-nodes-193427

Shader Graph tool

- Beforehand you need to install **Universal Rendering Pipeline** package and set it in the *Graphics setting.* Then you are ready to download the **Shader Graph** package.





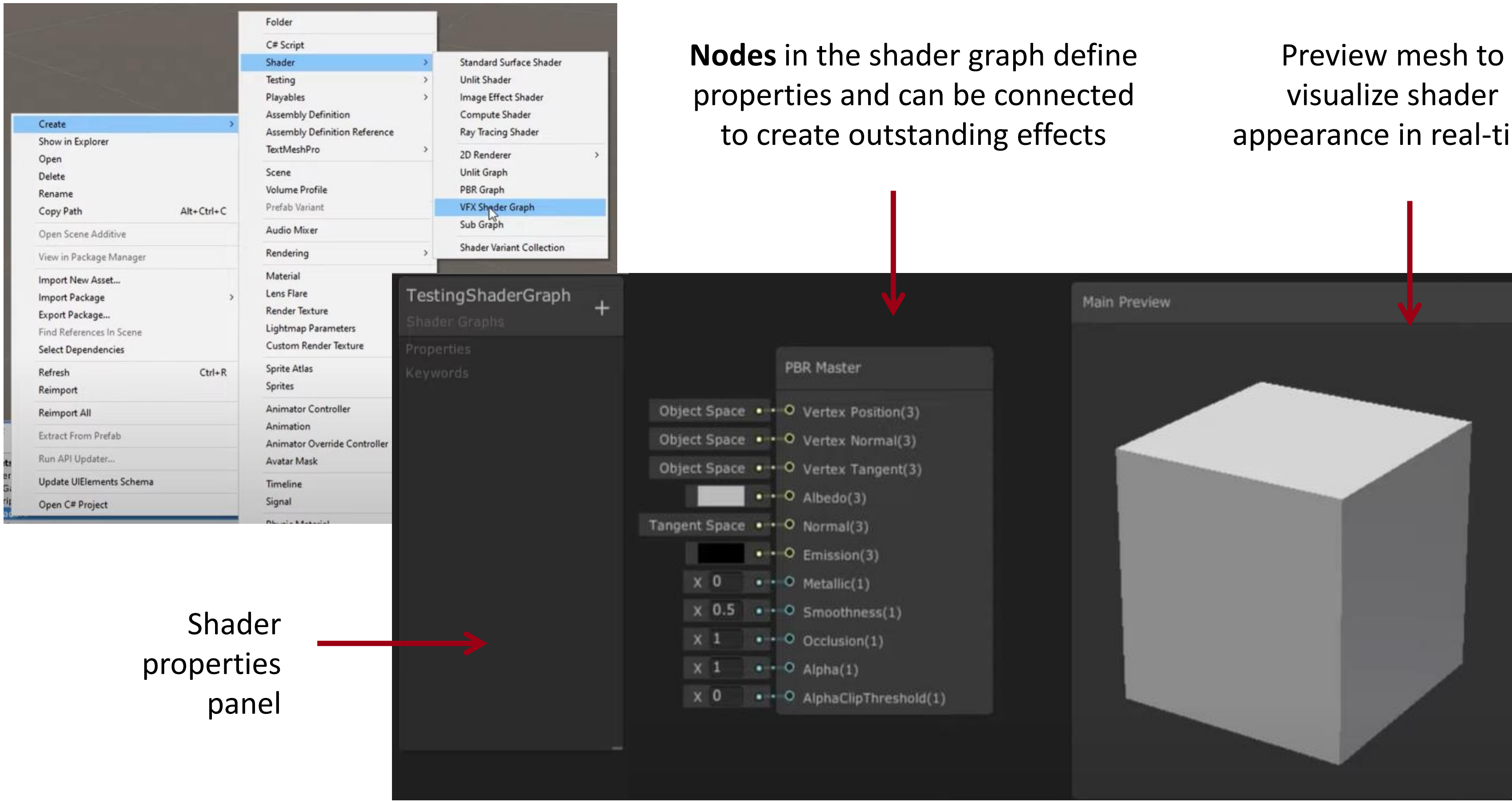
You can find the scriptable rendering pipeline under *Assets > Create > Rendering* and select *URP Asset* with *Universal Renderer*

Shader Graph tool

Nodes in the shader graph define properties and can be connected to create outstanding effects

Preview mesh to visualize shader appearance in real-time

Shader properties panel



Find more on: https://www.youtube.com/watch?v=VsUK9K6UbY4

EXERCISE: Visualization Tool

PROJECT

1. **Set Universal Rendering Pipeline and test Point Cloud meshing tool**

   In the *Point Cloud* folder, try the tool to load point clouds and have a look of the code. Then set the *Universal Rendering Pipeline* renderer.

2. **Build *point cloud shader* using Shader Graph**

   Convert the rose mesh to point cloud using the tool. Then, download and use the *Shader Graph* package tools to build a Shader, changing point clouds' appearance in such a way that the points look like *Neon.*

3. **Build *mesh shader* from scratch**

   Write a Shader for meshes to make it look like a *Hologram*. The material should be transparent, moving with regular disturb paths like in *Star Wars*.

4. **(ADVANCED) Change Point Clouds format**

   The only supported format for parsing is .OFF format, but our Point Clouds. Starting from that script, parse also the .txt data provided by the LiDAR (lidar.txt example file). Once parsed the .txt file, render it as a mesh, like it is performed for the .OFF files. Otherwise, you can write a script to convert your .txt point cloud to a .OFF file.

5. **(OPTIONAL) LiDAR point cloud visualization**

   Try the visualization tool using LiDAR point clouds, using the LiDAR Simulator (you can also use LAB5). You can assign colors depending on the labels given or on the distance.