VTKUnity - Activiz User's Guide

v2.0.0 - Google Docs Link

Contact: kitware.eu/contact/

Bug tracker: VTKUnity-BugTracker

Table of contents

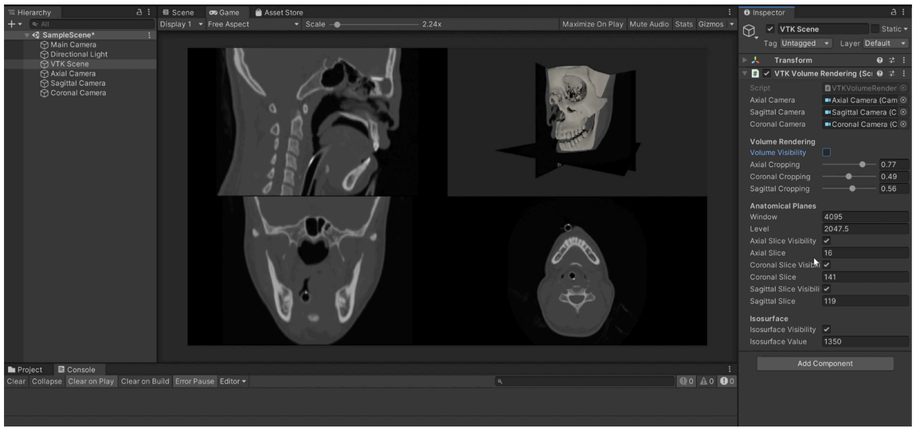
Overview

The role of this asset is to provide access to the [Visualization Toolkit (VTK)]() rendering and processing capabilities inside a Unity application. It is based on [ActiViz]() to offer a complete set of VTK features wrapped for C#. The VTK scene is rendered into Unity's rendering pipeline using the Unity [low-level native plug-in interface]() and leveraging the use of [command buffers]().

The asset is organized with the following directory structure:

- **Plugins**: Native and managed code providing VTK C++ API in Unity.
- **Scripts**: [MonoBehaviour]() scripts used to synchronize and update VTK rendering.
- **StreamingAssets**: Data files necessary for the demo scene.
- **Scenes**: Example scene using files from the StreamingAssets directory to showcase VTK visualization and computation capabilities.

In addition to bringing VTK into the Unity world. This asset also provides advanced visualization algorithms that can be used in a Unity-only scene.

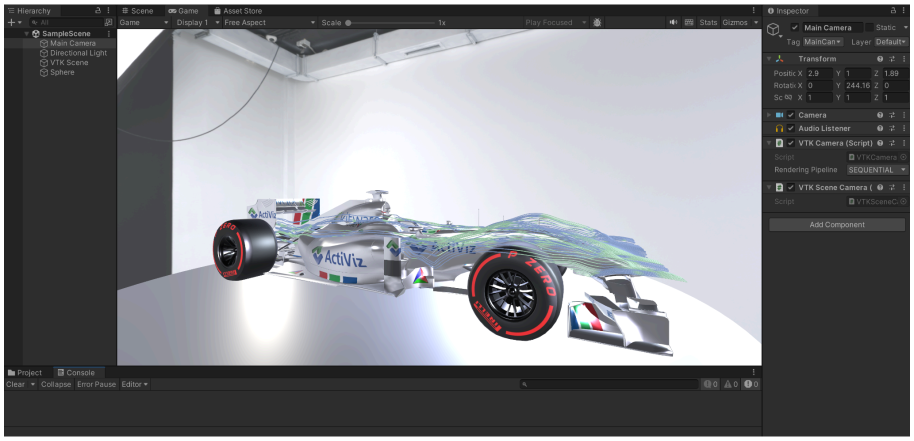


VTK Scene — Tag: Untagged — Layer: Default

Transform

VTK Volume Rendering (Sc

Script	VTKVolumeRender
Axial Camera	Axial Camera (Cam
Sagittal Camera	Sagittal Camera (C
Coronal Camera	Coronal Camera (C
Volume Rendering	
Volume Visibility	☐
Axial Cropping	0.77
Coronal Cropping	0.49
Sagittal Cropping	0.56
Anatomical Planes	
Window	4095
Level	2047.5
Axial Slice Visibility	✓
Axial Slice	16
Coronal Slice Visibi	✓
Coronal Slice	141
Sagittal Slice Visibili	✓
Sagittal Slice	119
Isosurface	
Isosurface Visibility	✓
Isosurface Value	1350



Main Camera — Tag: MainCan — Layer: Default

Transform

	X	Y	Z
Positic	2.9	1	1.89
Rotatic	0	244.16	0
Sc	1	1	1

Camera

Audio Listener

VTK Camera (Script)

Script	VTKCamera
Rendering Pipeline	SEQUENTIAL

VTK Scene Camera (

Script	VTKSceneC

Limitations

- **OS support**: The native plugin has been implemented and tested for **Windows** only. Support for Linux, MacOS and Android might come in future versions.

- **Unity Graphics API**: VTK relies on the **OpenGL** rendering backend, thus Unity graphics API must be set to OpenGLCore.

- **Virtual and Augmented Reality**: Although VTK and ActiViz natively support rendering in VR and AR headsets such as the Hololens 2, the Unity support for OpenXR does not work with the OpenGLCore rendering backend. For this reason, the plugin only supports rendering in VR headsets using the Unity legacy VR support that is based on OpenVR and available in **Unity 2019.**

Contact

Let us know how to improve the asset so it better fits your needs at kitware@kitware.fr, or use the bug tracker to report issues.

How to use

To use the plugin, create a new Unity 3D project and proceed as follows. Alternatively, open one of the example scenes provided with the asset in order to adapt the existing code.

Setup project:

- Change rendering backend to OpenGLCore
  - Edit->Project Settings->Player->Other Settings
  - Untick "Auto Graphics API for Windows"
  - Remove current API and add "OpenGLCore"
- Add VTKCamera.cs to every camera component in your Unity project
- Add VTKLight.cs to every light component in your Unity project
- The VTKSceneCamera.cs script can also be attached to an empty game object to trigger rendering in the editor without starting the game. This requires all scripts to use the [ExecuteInEditMode] keyword.

Setup VTK scene:

The VTK scene can be managed in a separate script and rendered by multiple cameras. VTK actors must be added to every vtkRenderer object exposed in the VTKCamera scripts added in the previous step.

The VTKSceneObject.cs script implements a MonoBehavior subclass that provides callback functions being called for each camera having a VTKCamera component. Such callbacks allow users to easily add and remove scene objects from the associated vtkRenderer.

- Create an empty GameObject and attach a new script to it
- Edit the new script to make it a VTKSceneObject subclass and implement the methods below:
  - `void Start()`: This is the MonoBehavior start callback, it should be used to initialize data and pipelines
  - `protected override void AddSceneObjects(vtkRenderer renderer)`: Used to add actors to the VTK renderer. Called when enabled and when VTKCamera is enabled.
  - `protected override void RemoveSceneObjects(vtkRenderer renderer)`: Used to remove actors from the VTK renderer. Called when disabled and when VTKCamera is disabled. By default the parent class removes all actors from the renderer.

Here is an example script adding a simple VTK sphere to the Unity scene:

C/C++

```
using Kitware.VTK;
using UnityEngine;

public class VTKExample : VTKSceneObject
{
  vtkActor SphereActor;

  /** MonoBehavior Start callback: Initialize VTK scene */
  void Start()
  {
    vtkSphereSource sphereSource = vtkSphereSource.New();
    sphereSource.Update();

    vtkPolyDataMapper sphereMapper = vtkPolyDataMapper.New();
    sphereMapper.SetInputData(sphereSource.GetOutput());

    this.SphereActor = vtkActor.New();
    this.SphereActor.SetMapper(sphereMapper);
  }

  /** AddSceneObjects callback: Add VTK actors to the renderer */
  protected override void AddSceneObjects(vtkRenderer renderer)
  {
    if (renderer.HasViewProp(this.SphereActor) == 0)
    {
      renderer.AddActor(this.SphereActor);
    }
  }
}
```

Monobehaviour scripts components

The following script components are provided for users to set up a VTK scene within their Unity project.

Scripts within the Core folder are provided to synchronize the two scenes:

- *VTKCamera.cs*: This script should be attached to Unity's Camera to trigger VTK rendering. It uses Unity's command buffers to call the VTK rendering callback implemented in the native plugin. It exposes a vtkRenderer object in which the VTK scene objects can be rendered.

- *VTKLight.cs*: This script should be added to a Unity light object to set up the corresponding VTK scene light and synchronize light parameters. VTK supports illumination with Directional, Point and Spot light types. Shadows, indirect lighting and cookies are unsupported for now.

- *VTKSceneObject.cs*: The VTKSceneObject.cs script implements a MonoBehavior subclass that provides callback functions being called for each camera having a VTKCamera component. Such callbacks allow users to easily add and remove scene objects from the associated vtkRenderer.

- *VTKSceneCamera.cs*: This script can be attached to an empty game object to trigger rendering in the editor without starting the game. This requires all scripts to use the [ExecuteInEditMode] keyword.

- *VTKNativePlugin.cs*: This script provides helper functions called internally to synchronize the VTK render passes with the Unity render pipeline.

- Other scripts related to depth peeling are explained in the "Order Independent Transparency" section below.

Scripts within the Scene folder provide example classes to design and update a VTK scene. Best practice is to attach each script to an empty GameObject in Unity:

- *VTKStreamlines.cs*: Instantiates and updates a VTK scene to display streamlines in a vector field.

- *VTKVolumeRendering.cs*: Read a 3D image to perform volume rendering, display anatomical planes, and extract polygonal surfaces using VTK filters.

- *VTK.cs*: Basic VTK scene showing how to use multiple cameras

- *VTKSampleScene.cs*: Basic VTK scene used to demonstrate the Order Independent Transparency technique presented in the next section

Order Independent Transparency

Overview

When rendering transparent objects, blending of fragments can result in rendering artifacts if the geometry is not sorted.

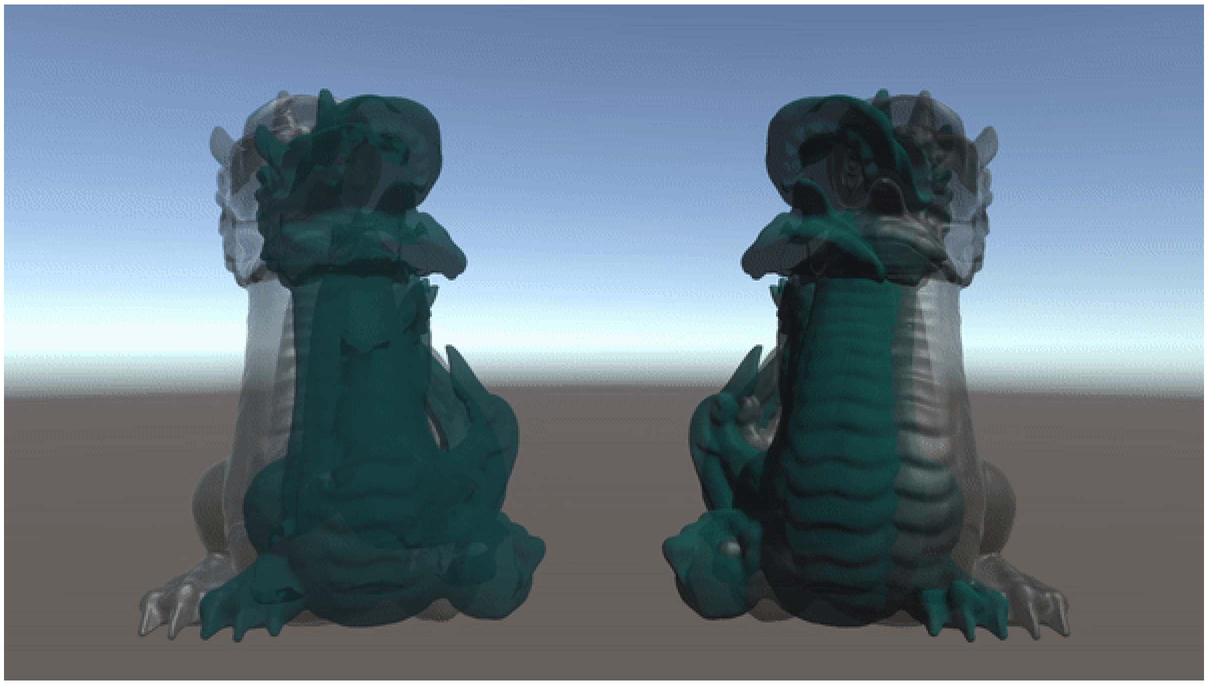
Order independent transparency techniques aim to produce a correct alpha compositing, while avoiding sorting of the geometry.

Depth Peeling is an approach using multiple render passes in order to peel the objects from front to back until there is no more geometry to render. The order of draw calls in a render pass does not matter as fragments are discarded based on their depth value. It has the advantage of working not only for blending the front and back faces of the same object, but also for transparent objects intersecting each other.

Dual depth peeling is an enhancement of the Depth Peeling algorithm, allowing both the front and back depth layers to be resolved in a single render pass. Hence the number of required render passes to complete the resolution of all fragments is drastically reduced.

Unity rendering pipeline sorts the translucent objects from back to front, resulting in a correct alpha blending of game objects. However, this has some limitations when objects are intersecting each other.

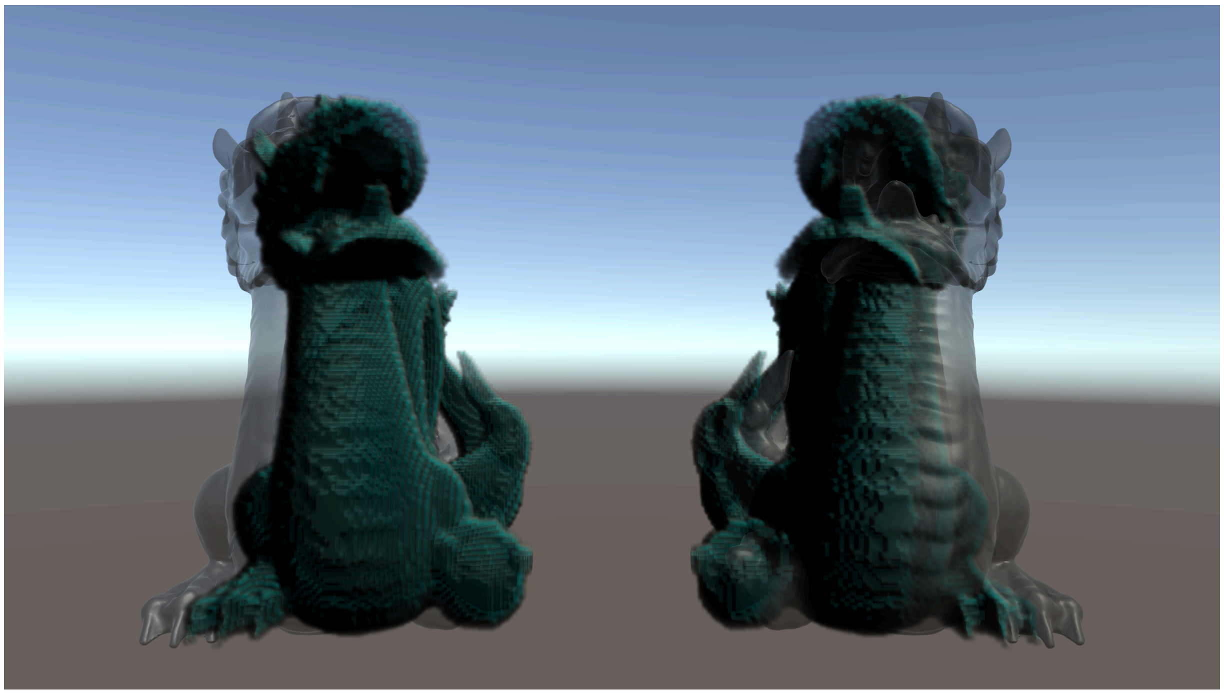
On the following image on the left, Unity game objects cannot be correctly sorted because sorting relies on the center of the object. In opposition, dual depth peeling produces the expected interlaced geometries.



Default geometry sorting in Unity

Dual Depth Peeling in unity

Another big advantage of depth peeling is that it works with VTK Volume rendering, allowing a volume to be correctly composed with a Unity transparent game object.



Without Depth Peeling, VTK volumes are rendered on the top of Unity translucent objects

VTK Volume Rendering mixed with a Unity transparent game object.

Using Dual Depth Peeling in a Unity scene

Depth peeling is performed by adding command buffers to the Unity camera rendering the translucent game objects. The mesh renderer of those game objects is automatically disabled to delegate the rendering to the depth peeling pass performed by the camera command buffers.

Depth Peeling requires modifications to the fragment shader code of transparent materials. A detailed presentation of the steps required to adapt an existing material is given in the next section. An adaptation of the Unity Standard material is also provided within this package.

The following MonoBehavior scripts are provided to setup depth peeling in a Unity scene:

- *DualDepthPeelingCamera.cs*:

This script should be attached to Unity's Camera to setup depth peeling. It internally manages a rendering system in a singleton to ease the addition and removal of cameras and objects to the render passes.
The following parameters are publicly exposed:

uint NumberOfLayers = 4;
Defines the maximum number of depth peeling render passes.

- *DepthPeelingObject.cs*:

This script should be attached to a translucent object to add it to the depth peeling rendering system. Any mesh renderer attached to the parent game object will be disabled. A depth peeling material must be used by the parent game object.

Current Editor limitations:

- The preview of depth peeling materials in the Unity editor is not supported yet.

Adapt an existing material for depth peeling

Using an existing material for Dual Depth Peeling requires modifications to the original fragment shader code, as well as the render pass state of this material.

The steps required to adapt an existing shader are presented below. A modified version of the Unity standard shader is also shipped with the plugin and can be used as an example for adapting a more complex shader.

- Add the following properties to the shader **Properties** block to control the blend state of the different depth peeling render passes:

```
[HideInInspector] _BlendOperation("BlendOperation", Float) = 4.0
[HideInInspector] _SrcBlend ("__src", Float) = 1.0
[HideInInspector] _DstBlend ("__dst", Float) = 0.0
```

- Override the shader Pass state in the **Pass** block as follow:

```
Blend [_SrcBlend] [_DstBlend]
BlendOp [_BlendOperation]
ZWrite Off
ZTest LEqual
Cull Off
```

  Disabling culling makes sense for fully transparent objects where we want to see the backfaces. Some translucent shaders effects require backface culling. In this case the Cull state can stay unchanged.

- Add the following directive inside the **CGPROGRAM** block:

```
#include "DualDepthPeelingDeclaration.cginc"
```

  The path to the include file is relative to the current shader directory.

- The **vertex** shader main function can remain unchanged

- The **fragment** shader main function should be modified to use the following:

- The declaration should be changed to return a **FragmentOutput** struct. This struct is defined in the *DualDepthPeelingDeclaration.cginc* file included above. It stores the required fragment information for depth peeling.

  At the beginning of the fragment function, a depth peeling **pre-color** step is performed in order to discard useless fragments. Add the following lines at the very beginning of the fragment shader main function, before computing any color. Note the instantiation of the FragmentOutput object that will be returned later.

  ```
  FragmentOutput output;
  DEPTH_PEELING_PRECOLOR(i.vertex, output)
  ```

  The only variable name you have to adapt to your use case is the vertex position passed to the depth peeling pre-color macro. In the above snippet it is referenced as *i.vertex* but you should adapt this to use the name of your shader "vertex to fragment" struct.

- After this pre-color step should come the original fragment color computation code. For instance, here is a very simple fragment computation sampling from a texture and setting the opacity to 0.5.

  ```
  fixed4 col = tex2D(_MainTex, i.uv);
  col.a = 0.5;
  ```

  Most fragment shaders return a single color targeting SV_TARGET (or COLOR0). This corresponds to the **Color0** attributes of the **FragmentOutput** struct.

- Finally copy the above computed color to the FragmentOutput instance and proceed to depth peeling by adding the following block at the very end of the shader pass:

  ```
  output.Color0 = col;
  DEPTH_PEELING_PEEL(output)
  return output;
  ```

License

VTK

VTK is an open-source toolkit distributed under the OSI-approved BSD 3-clause License. See the following Copyright.txt for details.

Activiz

Activiz is a closed-source software licensed under the BSD license. Read the Activiz License Agreement for more information.