SSF$_Particle2Fluid_ShaderUtil$

Release 0.0.3

Feb 07, 2020

Contents

1 Introduction **3**
1.1 Basic Setup 6
1.2 Step By Step Usage 6
1.3 Debug Tips 8

2 Advance Topics **11**
2.1 Extend Particle Inputs 11
2.2 Surface Shading 13

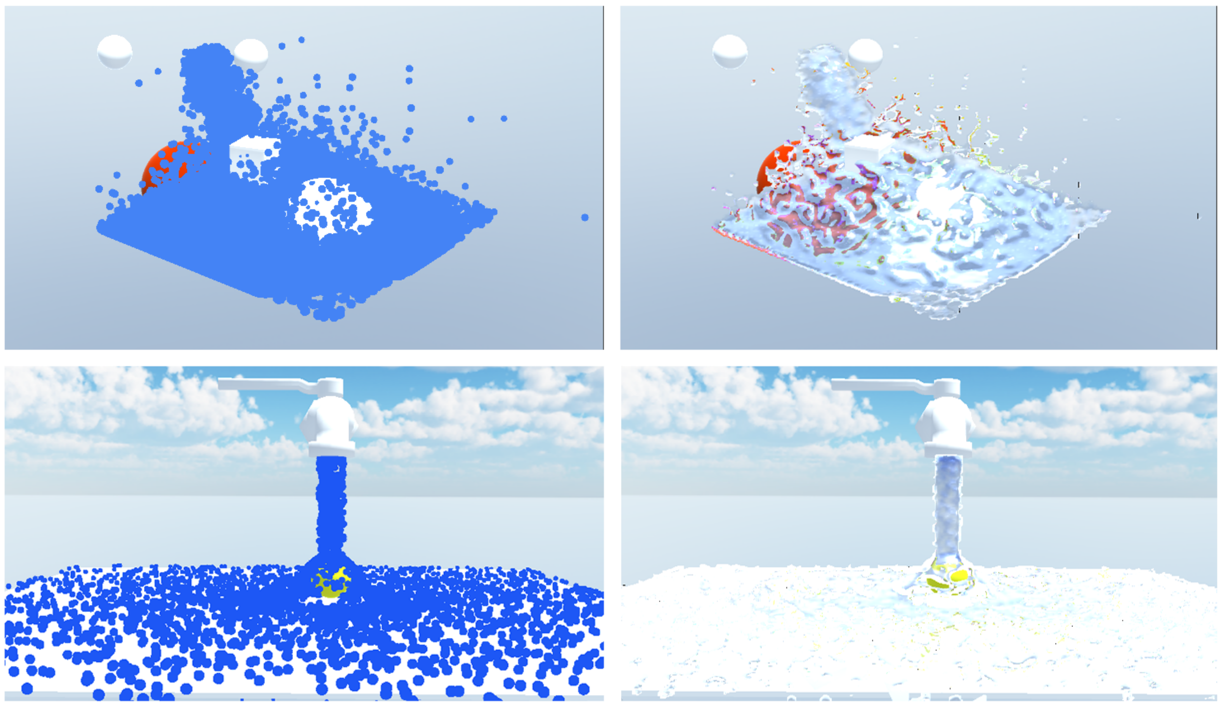
3 API **15**
3.1 SSF_ParticleSource 15
3.2 SSF_TextureGenerator 15
3.3 SSF_RenderSurface 17

4 Indices and tables **19**

This is a Unity shader plugin, not a fluid physics simulation plugin. It is used to render particle data into a smooth liquid surface. It is suitable for rendering simulation systems that use particles as simulation units.

It has the following very good properties:

- Excellent real-time operation efficiency
- Excellent surface effect
- Open data customization interface
- Complete documentation and improvement guidelines



raw particle data smoothed fluid surface with shading

CHAPTER 1

Introduction

This is a Unity shader plugin, not a fluid physics simulation plugin. It is used to render particle data into a smooth liquid surface. It is suitable for rendering simulation systems that use particles as simulation units.

Note: The principle of this plugin is based on the paper `Screen Space Rendering With Curvature Flow`.

Fluid simulation is generally based on grids or particles. In consideration of real-time performance, the SPH-based method (a particle-based method) is still used.

Unity does not have a very suitable fluid rendering plugin, which is the main reason for this plugin. I also noticed that there is indeed an implementation based on the same principle on the Asset Store.

In the process of using, I feel that I can do better, no matter from the efficiency or visual effects or ease of use and scalability, thus this plugin was born.

Note: This plugin is developed on *Unity 2019.3.0f5 (64-bit)* version and supports *Unity Builtin Shader System* and *Unity URP System*. It runs more efficiently on the Unity Builtin Shader System and is not optimized for URP.

Warning: `OnRenderObject` needs to be supported. It cannot run on LWRP.

I prepared several demo scenarios:

Note: By default, the Gameobject named `Renderer` is off on each demo, **enable it** to see the effects. If still not work, **reactive** the *ParticleSource* GameObject and *Renderer* Gameobject.

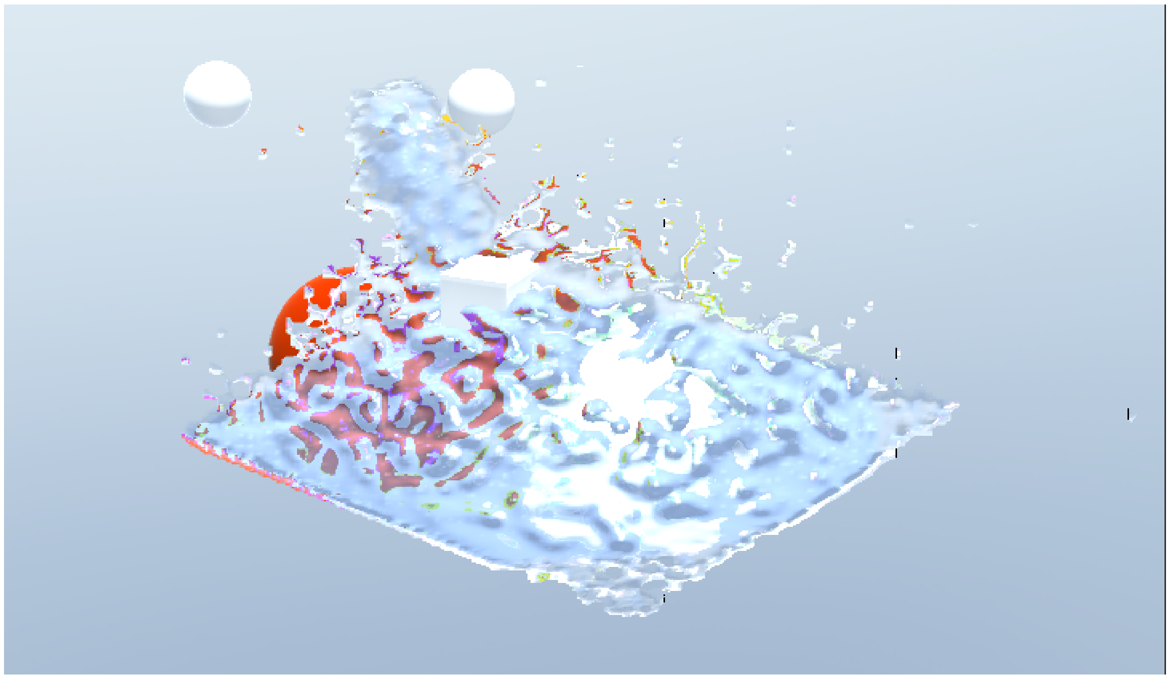


Fig. 1: [DEMO] Load particles from file

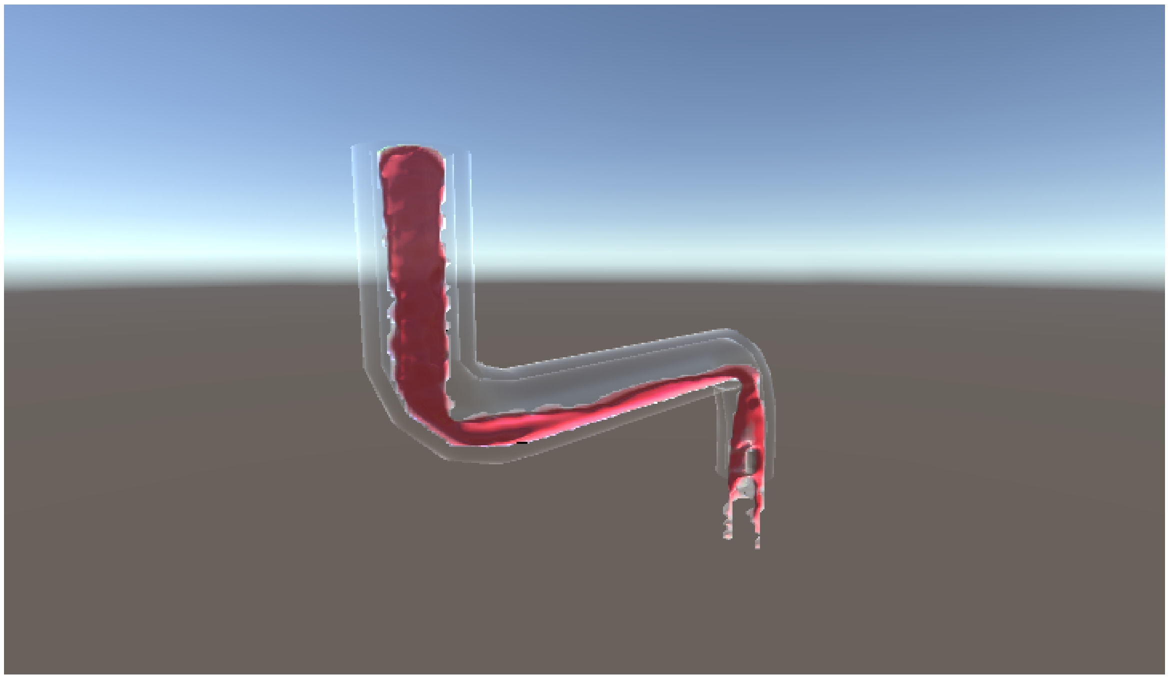


Fig. 2: [DEMO] Blood

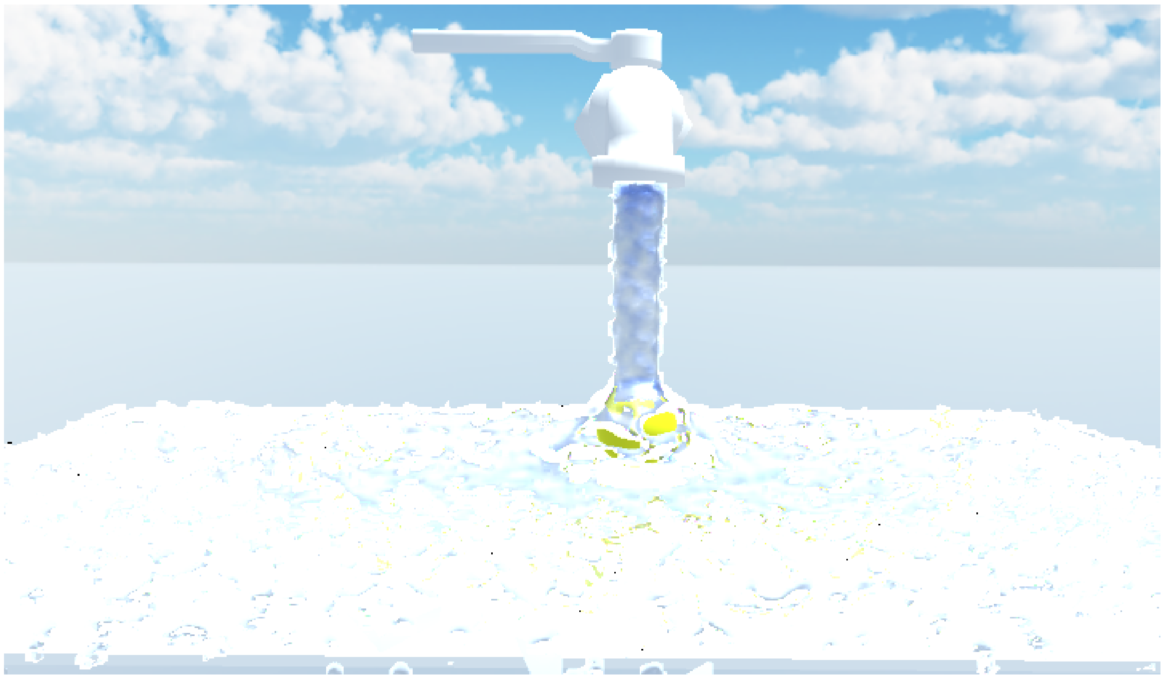


Fig. 3: [DEMO] Single ParticleSystem

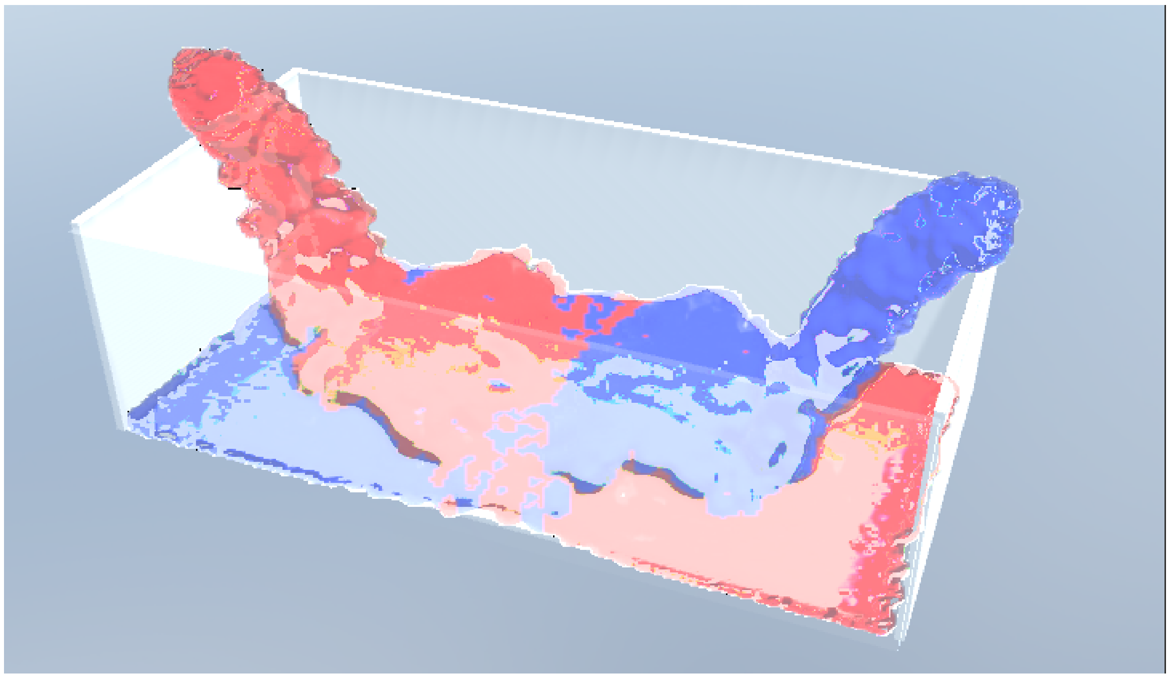


Fig. 4: [DEMO] Multiple ParticleSystems

1.1 Basic Setup

First you need to download this from the Unity Asset Store Plugin.

Then, import this plugin and you will find demos in `Scene Folder` .

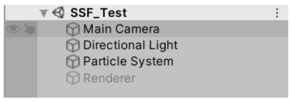
You can choose to open any Demo such as `Demo_File.scene`, and then enable `Renderer`, you can see the effect of the plugin in the scene.

You can also continue to read this article to understand the process of using the plugin from scratch.

1.2 Step By Step Usage

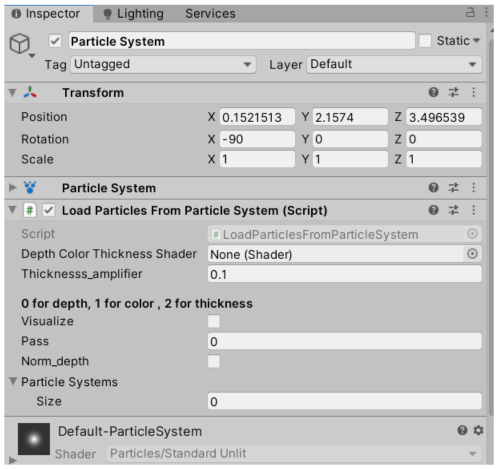
1.2.1 Setup Scene

1. Create an empty Scene named `SSF_Test`
2. Create a *ParticleSystem* and deactivate its `Renderer` function
3. Create an empty *Object* named `Renderer`

The Inspector should looks like: 

SSF_Test
Main Camera
Directional Light
Particle System
Renderer

Add `SSF_LoadParticlesFromParticleSystem`. The Inspector should appears as follows:



Inspector | Lighting | Services

Particle System — Static
Tag: Untagged — Layer: Default

Transform
Position: X 0.1521513, Y 2.1574, Z 3.496539
Rotation: X -90, Y 0, Z 0
Scale: X 1, Y 1, Z 1

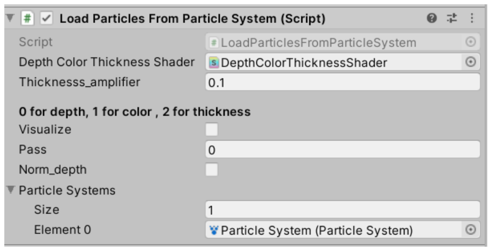
Particle System

Load Particles From Particle System (Script)
Script: LoadParticlesFromParticleSystem
Depth Color Thickness Shader: None (Shader)
Thicknesss_amplifier: 0.1

0 for depth, 1 for color , 2 for thickness
Visualize:
Pass: 0
Norm_depth:
Particle Systems
Size: 0

Default-ParticleSystem
Shader: Particles/Standard Unlit

1. Assign shader and the ParticleSystem just as follows:



Load Particles From Particle System (Script)
Script: LoadParticlesFromParticleSystem
Depth Color Thickness Shader: DepthColorThicknessShader
Thicknesss_amplifier: 0.1

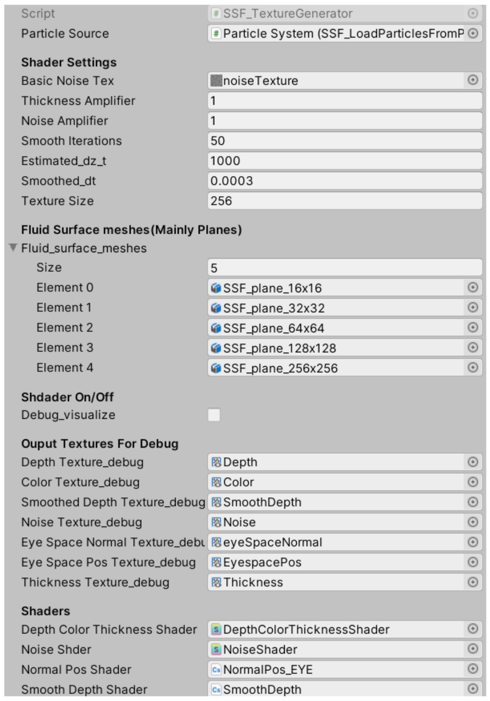
0 for depth, 1 for color , 2 for thickness
Visualize:
Pass: 0
Norm_depth:
Particle Systems
Size: 1
Element 0: Particle System (Particle System)

2. Disable and then enable the Component to take effect.
3. Now Toggle on `Visualize`, black spheres can be viewed in the *Scene* Window and *Game* Window.

Note: `Visualize` works only for debug purpose, it will not affect the proper workflow functionality.

1.2.2 Cofigure Renderer

1. Move on to the *Inspector* of the `Renderer` in hierachy
2. Click Add Component, Add `SSF_TextureGenerator`. This should be many missing values in the inpsector. Assign as follows:



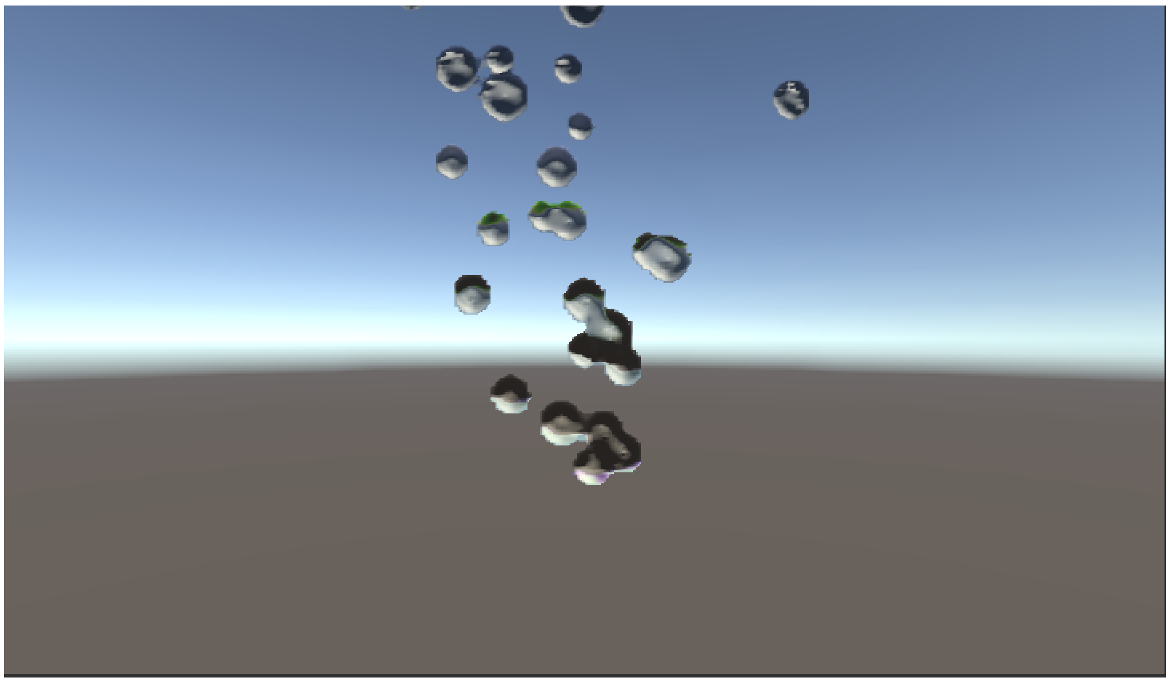
Script	SSF_TextureGenerator
Particle Source	Particle System (SSF_LoadParticlesFromP
Shader Settings	
Basic Noise Tex	noiseTexture
Thickness Amplifier	1
Noise Amplifier	1
Smooth Iterations	50
Estimated_dz_t	1000
Smoothed_dt	0.0003
Texture Size	256
Fluid Surface meshes(Mainly Planes)	
▼ Fluid_surface_meshes	
Size	5
Element 0	SSF_plane_16x16
Element 1	SSF_plane_32x32
Element 2	SSF_plane_64x64
Element 3	SSF_plane_128x128
Element 4	SSF_plane_256x256
Shdader On/Off	
Debug_visualize	☐
Ouput Textures For Debug	
Depth Texture_debug	Depth
Color Texture_debug	Color
Smoothed Depth Texture_debug	SmoothDepth
Noise Texture_debug	Noise
Eye Space Normal Texture_debu	eyeSpaceNormal
Eye Space Pos Texture_debug	EyespacePos
Thickness Texture_debug	Thickness
Shaders	
Depth Color Thickness Shader	DepthColorThicknessShader
Noise Shder	NoiseShader
Normal Pos Shader	NormalPos_EYE
Smooth Depth Shader	SmoothDepth

3. Disable and then enable the Component to take effect. Component of type `SSF_RenderSurface` should be automatically added.

The meaning and effect of parameters can be checked in *API*

1.2.3 Congratulations!

From *Scene* View, fluid-like shape can already be viewed .



It's not cool enough, right?

1.2.4 Check Other Cool Demos

Now it's time too check other cool demos!

1.3 Debug Tips

The overall workflow of this plugin can be separated into 3 parts:

- Particles Data Input
- Texture Generating
- Surface Shading

Here are some useful tips for users when using this plugin:

1. On anything regarding Graphics Changes (e.g. Saving/Exiting Scene, Saving Shader…), the *ComputeBuffer* used to generate textures will be discarded.
2. Under all situations, the first step to debug is to check if `ParticleSource` was assigned on `SSF_TextureGenerator`
3. If assigned, toggle On `checkVisualize` of `SSF_TextureGenerator` and check TextureOutputs.
4. If there's colored output on *EyeSpaceNormalTex*, then problems exist on the surface shading part.
5. If none, it could be two possible reasons during *Texture Generating*:
    1. `ParticleSource` is not providing data properly.
    2. `ComputeBuffer` is lost for some reasons (may due to scene saving and loading).

This first reason may due to users' buggy coding.

To tackle down the second reason, you have to first reactive `ParticleSource`, then reactive `SSF_TextureGenerator`.

Note: Here, **reactive** means exactly *Disable and then Enable*

CHAPTER 2

Advance Topics

In this chapter, guidance on modifying this plugin will be demonstrated. Besides, customizing surface shading will also be covered. A little bit knowledge about parameter tuning may be included.

2.1 Extend Particle Inputs

Considering that users may have their own source of particle data, such as a particle solution system running in parallel with the GPU, or imported pre-made particle data, here we will explain how to extend the input of particles.

In `SSF Particle2Fluid ShaderUtil (SSF)` ,the input of particles is implemented by the base class `SSF_ParticleSource`.

2.1.1 Particle Data Struct

The structure of particle data in SSF is as follows:

```
public struct SSF_particle
{
    public Vector3 position;
    public Color color;
    public float radius;
}
```

Note: If you modify the particle's data structure, you need to pay attention to replacing 32 in `particleBuffer = new ComputeBuffer (getParticleNum (), 32);` in **SSF_ParticleSource.cs** with the number of bytes of particle data. At the same time, corresponding changes should be made in `DepthColorThickness.shader` and `NoiseShader.shader`.

2.1.2 Explain SSF_ParticleSource

The input and update of extended data is to create a new class inheriting from `SSF_ParticleSource` and implement the corresponding virtual function.

The following two member variables exist in `SSF_ParticleSource`:

```
protected ComputeBuffer particleBuffer;// Buffer sent to GPU
protected SSF_particle[] particlesData;// Particle Data for above buffer
```

Where `particleBuffer` is used to provide data to `SSF_TextureGenerator` to generate related textures for rendering.

You can notice that the following member function modifiers in `SSF_ParticleSource` are *public virtual*:

- setupParticleBufferData ()
- updateParticleBufferData ()

In `setupParticleBufferData` , `particlesData` needs to be created and assigned, and updated in `updateParticleBufferData ()` , neither of these operations need to involve `particleBuffer`.

2.1.3 Example

The simplest example is `SSF_LoadParticlesFromFile.cs`.

```
public class SSF_LoadParticlesFromFile : SSF_ParticleSource
   {
       public UnityEngine.Object particleFile;
       public float particleRadius;
       public Color particleColor;
       [Range(0,2)]
       public int positionOrder_0=0;
       [Range(0,2)]
       public int positionOrder_1=2;
       [Range(0,2)]
       public int positionOrder_2=1;

       public override void setupParticleBufferData()
       {
           base.setupParticleBufferData();
           if (particleFile != null)
           {
               TextAsset asset = particleFile as TextAsset;
               string[] striparr = asset.text.Split(new string[] { "\r\n", " " },␣
↪StringSplitOptions.RemoveEmptyEntries);
               particle_num = striparr.Length / 3;
               print("Loaded particles : " + particle_num);
               particlesData = new SSF_particle[particle_num];
               for (int i = 0; i < particle_num; i++)
               {
                   particlesData[i].position = new Vector3(Convert.
↪ToSingle(striparr[3 * i+positionOrder_0]),
                   Convert.ToSingle(striparr[3 * i + positionOrder_1]), Convert.
↪ToSingle(striparr[3 * i + positionOrder_2]));
                   particlesData[i].radius = particleRadius;
                   particlesData[i].color = particleColor;
               }
```

(continues on next page)

(continued from previous page)

```
            }
        }
        public override void updateParticleBufferData()
        {
            base.updateParticleBufferData();
        }

    }
```

You can also refer to `SSF_LoadParticlesFromParticleSystem`, this is a bit complicated and tedious.

2.2 Surface Shading

In previous asset, Surface Shading has lots of limitations:

- achieved by ImageEffects on Camera, which is no longer supported in *URP*.
- do not work well when there are transparent objects in scene.
- do not support mulitiple lights and global illumination.

In a word, it limits as it's just some sort of imageEffects.

In our implementation, we **reconstruct the fluid surface from textures** using quads of different resolutions (or dimensions).

Based on that, *Amplify Shader Editor* was used to write a surface shader `Fluid Surface.shader` for that surface. **Therefore the rendering process of fluid surface can be integrated into Unity's Rendering Pipeline.**

2.2.1 Textures Description

To understand how to change surface shading, textures generated from `SSF_TextureGenerator` should be understood.

Textures Name	Texture Format	Description
`DepthTexture`	R	origin depth of particles in ViewSpace
`ThicknessTexture`	R	describes how thick the fluid is from ViewSpace
`NoiseTexture`	R	used to peturb surface normal and add Foam effect, ViewSpace
`SmoothedDepthTexture`	R	smoothed depth of particles in ViewSpace
`EyeSpaceNormalTexture`	RGBA	fluid surface normal generated from `SmoothedDepthTexture` in ViewSpace
`EyeSpacePosTexture`	RGBA	fluid surface position generated from `SmoothedDepthTexture` in ViewSpace

2.2.2 Surface Shader

It's already described that surface is created from a quad mesh.

On enabling the `SSF_TextureGenerator`, two things happen simultaneously.

- a script called `SSF_RenderSurface` will be attached.

- a GameObject which is the Surface Mesh will be attached as **the child** of *ParticleSource* of `SSF_TextureGenerator`

Note: Tuning the parameters of `SSF_RenderSurface` and `SSF_TextureGenerator` to ajust surface appearance .

Through opening the `Fluid Surface.shader`, the graph flow can be viewed.

`advance_topics/../images/Shader_Graph.png`

We mainly do following things:

1. render mesh as transparent object
2. replace the quad's vertices' positions and normals with fluid surface normal and vertices.
3. sample from `ThickenessTexture` to set opacity
4. sample from `ThickenessTexture` and use *Lambert-Beer Law* to set `Specular`
5. take fluid's *Index of Refraction* into Consideration and set `Refraction`
6. sample from `NoiseTexture` to peturb normal and add Foam Effect
7. sample from `ColorTexture` to set `Albedo` port

Note: It's recommended to open the `Fluid Surface.shader` using *Amplify Shader Editor*.

For customization purposes, you can copy this shader and make your customization.

Then assign the shader as the *Shader* Input to `SSF_RenderSurface`.

CHAPTER 3

API

Code Logics are clear when viewing project codes. API parts of Docs seen to be unnecessary.

However this chapter is about parameter tuning which should also cover some part of API.

Thus let us start script by script.

3.1 SSF_ParticleSource

This class has been explained clearly in *Extend Particle Inputs*

This class provides data to `SSF_TextureGenerator` for generating textures.

Note: `SSF_TextureGenerator` uses the transform of `SSF_ParticleSource` as **the model matrix** for particles, please ensure it's your expected model matrix.

Note: Thus, if using multiple particleSystem, the `simulationSpace` should be setted to `World` and this script should be attached to a gameObject with Identity transform.

3.2 SSF_TextureGenerator

SSF_TextureGenerator forks `particleBuffer` from *ParticleSource* and then generate Textures for further *surface reconstruction and shading*.

Its working logic can be summarized as follow:

```
void OnEnable()
{
```

(continues on next page)

(continued from previous page)

```
    print("[SSF] Enabled TextureGenerator "+ gameObject.name);
    setupTextures();
    setupMaterials();
    // Add Surface Mesh and shading if not exists.
    if(GetComponent<SSF_RenderSurface>()==null){
        gameObject.AddComponent<SSF_RenderSurface>();
    }
    GetComponent<SSF_RenderSurface>().enabled = true;
    // Set shading's texsource from this
    GetComponent<SSF_RenderSurface>().tex_source = this;
}
void OnDisable()
{
    print("[SSF] Disabled TextureGenerator "+ gameObject.name);
    releaseTextures();
    releaseBuffers();
    DestroyImmediate(material_depthColorThickness);
    DestroyImmediate(material_noise);
    //Disable Surface Shading
    if(GetComponent<SSF_RenderSurface>()!=null){
    GetComponent<SSF_RenderSurface>().enabled = false;
    }
}
```

Then Draw Textures On Each Frame:

```
void OnRenderObject()
{
    if (particleSource != null)
    {
        particleSource.updateParticleBuffer();
        setParams();
        check_debugVisualize();
        drawColorTexture();
        drawDepthTexture();
        drawThicknessTexture();
        smoothDepthTexture();
        drawNoiseTexture();
        drawNormalViewDirTexture();
    }
}
```

3.2.1 Param Tuning

smoothIterations Describes how many smoothing operations each frame, normally 50-120 is suitable

smothed_dt Describes the timestep dt for each smothing operation, normally 5e-4 is suitable

estimated_dz_t From some sense, it amplifies the smoothing effect of above params, normally 1000

thicknessAmplifier Controls *thicknessTexture*'s output magnitude

basicNoiseTex Render each Particle with a basicNoiseTex, generate a *noiseTexture* for fluid shading

noiseAmplifier Controls *noiseTexture*'s output magnitude

textureSize Controls the Texture Size of Texture ouput with `textureSize*textureSize`

debug_visualize If toggle on, it will copy `*Texture_ouput` to `*Texture_debug` for debug purposes. This requires two kinds of textures share the same format and dimension.

fluid_surface_meshes Array of Plane meshes with different resolutions, corresponding to *surfaceQuality* in `SSF_RenderSurface`

3.3 SSF_RenderSurface

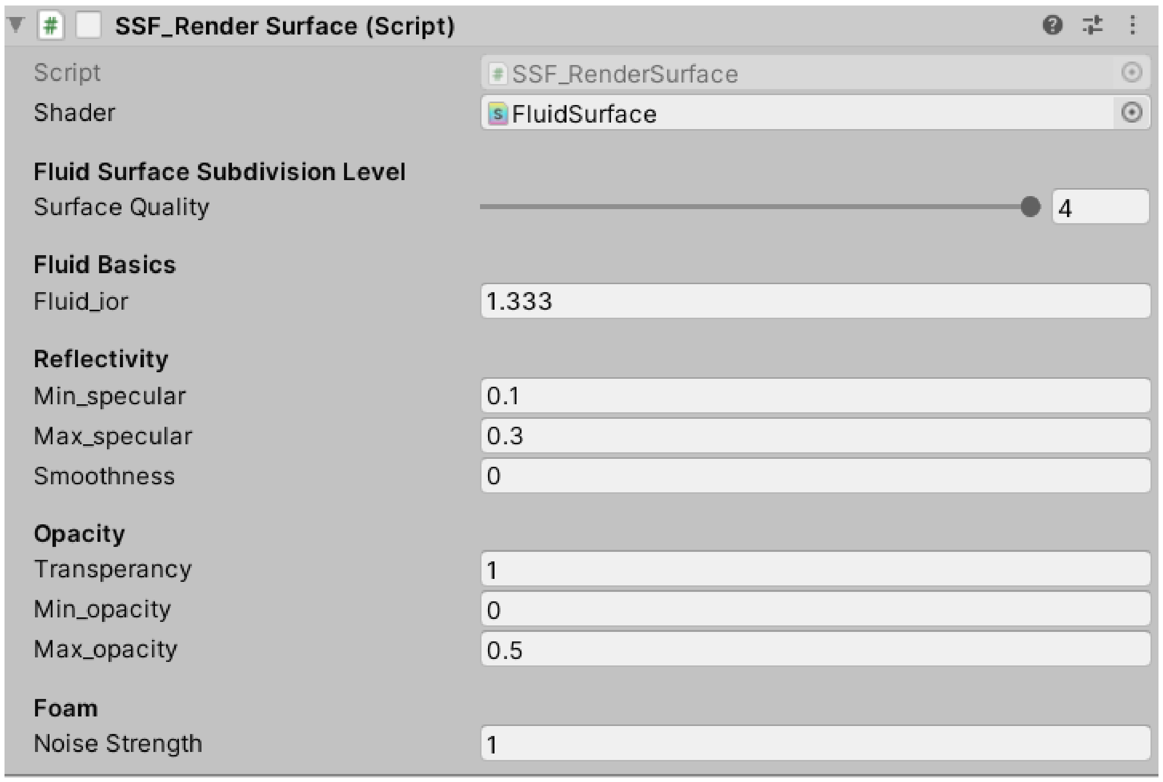
This script is used to ajust the surface shading appearance and quality.

The logic is straight forward.

When enabled:

1. it creates a plane mesh (according to *surfaceQuality*)
2. set the mesh as the child of the ParticleSource, which ensures visablity.
3. attach `FluidSurface` Shader to that mesh and ajust params as setted.

3.3.1 Param Tuning



SSF_Render Surface (Script)

Script	SSF_RenderSurface
Shader	FluidSurface
Fluid Surface Subdivision Level	
Surface Quality	4
Fluid Basics	
Fluid_ior	1.333
Reflectivity	
Min_specular	0.1
Max_specular	0.3
Smoothness	0
Opacity	
Transperancy	1
Min_opacity	0
Max_opacity	0.5
Foam	
Noise Strength	1

shader shader used to reconstruct surface and rendering.

fluid_ior controls the Index of Refration of fluid, water is 1.333

surfaceQuality Fluid Surface Subdivision Level,corresponding to `fluid_surface_meshes` in `SSF_TextureGenerator`.

min/max_specular: control specular of fluid surface, as the specular is generated based on *Lambert-Beer Law*

smoothness controls Reflectivity,0 is roughest,1 is smoothest

transperancy controls opacity based on *thickness*, 0 is transparent, 1 is fully non-transparent

min/max_opacity: controls the bounds of opacity, 0 is transparent, 1 is fully non-transparent

noiseStrength controls the significance of Foam Effect and Normal Peturbition.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search