МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по производственной практике

Тема: Создание визуально сложных эффектов с помощью шейдеров в среде разработки Unity.

Студент гр. 4303 _____ Коптюг А.Д.

Преподаватель _____ Герасимова Т.В.

Санкт-Петербург

2019

ЗАДАНИЕ

НА ПРОИЗВОДСТВЕННУЮ ПРАКТИКУ

Студент Коптюг А.Д.

Группа 4303

Тема проекта: Создание визуально сложных эффектов с помощью шейдеров в среде разработке Unity.

Исходные данные (технические требования):

Среда для разработки приложений - Unity 2018.4.9f1 (64-bit)

Язык реализации программы - HLSL

Сроки прохождения практики: 01.09.2019 – 25.12.2019

Дата сдачи отчета: 20.12.2019

Дата защиты отчета: 25.12.2019

Студент		Коптюг А.Д.
Преподаватель		Герасимова Т.В.

АННОТАЦИЯ

Воссоздание сложных визуальных эффектов – одна из основных задач любой современной графической симуляции, для которой существуют различные решения. Данная статья является введением в общие принципы написания шейдеров на языке HLSL. На примере создания визуального эффекта “капли дождя” будет продемонстрирована работа шейдера в среде разработки Unity.

SUMMARY

All modern graphic simulations for which there are various solutions. This article is an introduction to the general principles of writing HLSL shaders. Using the example of creating the visual effect of “raindrops”, the shader will be demonstrated in the Unity development environment.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью научной исследовательской работы является:

1. Изучение современных подходов к анализу, проектированию и разработке программного обеспечения. Знание принципов написания шейдерных программ.

2. Формирование умения применять методы проектирования программного обеспечения. Получение навыков использования инструментальных средств, поддерживающих создание программного обеспечения.

3. Освоение подходов к обследованию предметной области выполняемого программного проекта.

1. АКТУАЛЬНОСТЬ

Раньше не существовало способов эффективно писать программы для графических процессоров. Если разработчик приложения хотел включить специальные эффекты для обработки изображения, то ему приходилось либо добавлять обработчики, работающие на CPU, что значительно замедляло скорость выполнения программы, либо довольствоваться сильно ограниченным набором поддерживаемых аппаратно эффектов. Для удобного написания программ разработчикам требовалось создать специализированный язык, который бы позволял создавать простые и быстрые программы для создания и применения различных эффектов. Именно для этой цели была разработана и стандартизирована концепция шейдеров. Благодаря им, теперь практически в любой современной графической симуляции используется код, написанный для видеопроцессора: от реалистичных эффектов освещения в высокотехнологичных AAA-играх до двухмерных эффектов постпроцессинга и симуляции жидкостей.

Однако, бывают случаи, когда программирование шейдеров представляется загадочной чёрной магией и его часто понимают неправильно. Существует множество примеров кода, демонстрирующих создание невероятных эффектов, но в которых практически нет объяснений. Даже опытные программисты шейдеров сталкиваются с проблемой, когда реализация шейдера отнимает уйму времени ввиду сложности его представления.

2. ОПИСАНИЕ ЗАДАЧИ

Основной задачей проекта является создание визуального эффекта, напоминающего капли дождя на окне, с помощью шейдерной программы. В ходе выполнения так же будет произведено сравнение различных способов реализации данного эффекта в среде Unity.

Как было сказано в предыдущем пункте, для воссоздания визуальных эффектов на видеопроцессоре исполняется специальный код, который называется шейдером. По своему типу шейдеры делятся на два основных типа:

- вершинные шейдеры оперируют с различными геометрическими данными вершин объектов в трехмерном пространстве;
- пиксельные или фрагментные шейдеры работают с фрагментами изображения, пикселями, преобразуя их некоторым образом, что может использоваться для постпроцессинга изображения – преобразования уже полученного изображения в новое (например, для применения эффекта черно-белого цвета).

Чтобы понять, как работают шейдеры и в каком порядке они применяются, можно рассмотреть упрощенную схему графического конвейера (рис. 1).

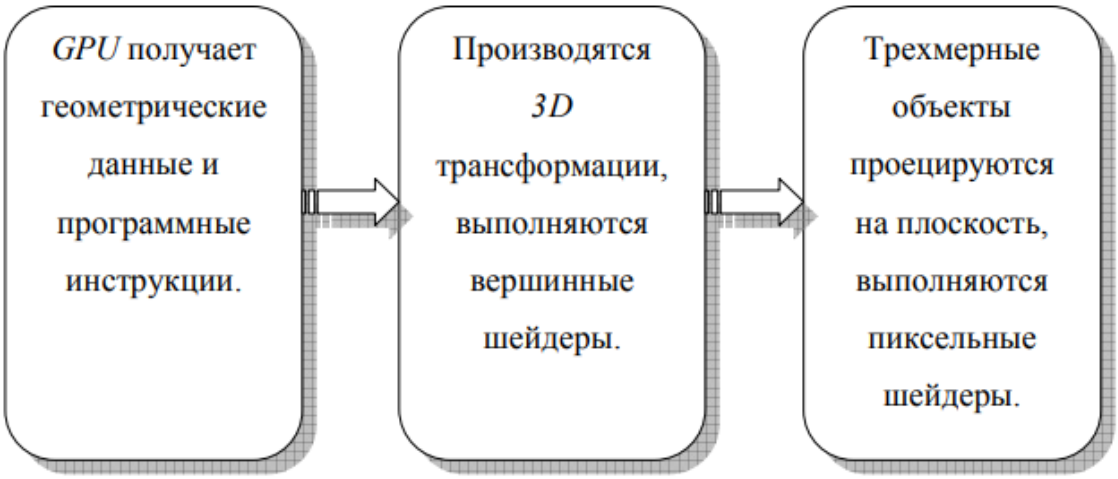

Рис. 1 Упрощенная схема графического конвейера

Обычно для решения задач на графических процессорах чаще используются пиксельные шейдеры. Как показано выше, на последнем этапе происходит проецирование трехмерных объектов на плоскость. На объекты накладывается текстура (двумерное изображение, накладываемое на грани трехмерных

объектов), а для каждого пикселя вычисляется результирующее значение цвета. Далее для каждого пикселя выполняются фрагментные шейдеры. В результате работы последнего из них получается финальный цвет пикселя, который и выводится на экран. Таким образом, каждый фрагментный шейдер будет выполнен N раз, где N – число пикселей результирующего изображения. Именно за счет наличия множества конвейерных процессоров фрагментные шейдеры выполняются быстро.

Модель шейдеров стандартизирует некоторые технические параметры шейдеров. В настоящее время последней версией является Shader Model 6.

Сейчас распространены две графические библиотеки: OpenGL и Direct3D. Для каждой из них были созданы C-подобные языки для написания шейдеров:

- для OpenGL: GLSL (The OpenGL Shading Language);
- для Direct3D: HLSL (High Level Shader Language);
- для OpenGL и Direct3D: Cg (C for Graphics).

Данные языки очень схожи с C, поддерживают различные типы данных, структуры, функции. Начиная с Shader Model 3, поддерживаются операторы ветвления. Отличительной особенностью является наличие встроенных типов для векторов и матриц, а также множества функций, оперирующих с ними. Стоит отметить, что также существуют низкоуровневые, ассемблероподобные языки, например, DirectX ASM.

Среда разработки Unity позволяет писать шейдеры на всех приведенных выше языках, однако, наиболее широко распространена практика написания шейдеров на языке HLSL, это обусловлено более качественной поддержкой языка самим движком Unity.

Далее в проекте будут рассмотрены несколько методов к решению поставленной задачи с использованием возможностей Unity.

3. МЕТОДЫ РЕШЕНИЯ ЗАДАЧИ

Перед тем, как перейти к описанию методов, нужно рассказать об общей структуре используемого шейдера т.к. все решения будут опираться именно на его структуру.

```
Shader "MyShaderName"
{
Properties
{
// свойства материала
}
SubShader // сабшейдер для определённого железа (можно определить директивой
компиляции)
{
Pass
{
CGPROGRAM

#pragma vertex vert
#pragma fragment frag

#include "UnityCG.cginc"

// проход шейдера
struct v2f {
float4 pos : SV_POSITION;
fixed3 color : COLOR0;
};

v2f vert (appdata_base v)
{
v2f o;
o.pos = UnityObjectToClipPos(v.vertex);
o.color = v.normal * 0.5 + 0.5;
return o;
}

fixed4 frag (v2f i) : SV_Target
{
return fixed4 (i.color, 1);
}
ENDCG
// для некоторых эффектов может понадобится несколько проходов
}
}
```

Такие директивы компиляции как

`#pragma vertex vert`

`#pragma fragment frag`

определяют какие функции шейдера компилировать в качестве вершинного и фрагментного шейдера соответственно.

Функция *UnityObjectToClipPos* — это вспомогательная функция Unity (из файла *UnityCG.cginc*), которая переводит вертексы объекта в позицию, связанную с камерой. Без неё объект, при попадании в зону видимости камеры,

будет рисоваться в координатах экрана вне зависимости от положения трансформа. Так как первоначально позиции вертексов представлены в координатах объекта.

```
     struct v2f {
float4 pos : SV_POSITION;
fixed3 color : COLOR0;
};
```

Это определение структуры, которая будет обрабатываться в вертексной части и передаваться в фрагментную. В данном случае в ней определено, чтобы из меша забиралось два параметра — позиция вершины и цвет вершины.

3.1 Различные способы реализации эффекта

Далее будут рассмотрены два метода создания эффекта каплей дождя на окне.

Simple Raindrops

Основная идея данного шейдера заключается в циклическом уменьшении/увеличении небольшой псевдослучайной области изображения с последующей рефракцией данной области. Таким образом, создается иллюзия того, что на окне появляется капля, а затем плавно уменьшается и исчезает с окна. Данный эффект описывается функцией зависимости размера капли от времени:

$$SizeDrop = \max(0.0, 0.5 - frac(intensity * t))$$

Где команда frac(), возвращает дробную часть от исходных данных.

Rain Window

Идея этого шейдера состоит в том, чтобы создать динамическую сетку, которая будет содержать в себе набор капель. Каждая капля будет менять свою позицию в пределах своей сетки согласно некоторой функции движения, тем самым создавая ощущение стекания по окну. Для разделения на отдельные зоны для капель используется ранее упомянутая команда frac(uv), которая возвращает дробную часть от исходных uv-координат.

Далее с помощью команды smoothstep создается сама капля: данная операция производит гладкую интерполяцию Эрмита между двумя значениями, передаваемые в качестве параметров.

Теперь требуется, чтобы полученные капли могли двигаться в переделах своей “коробки”. Более того, их движение должно быть более-менее реалистичным и цикличным. Для этой цели используется функция зависимости положения на оси ординат от времени:

$$y = -\sin\left(x + \sin\left(x + \sin(x) \cdot .5\right)\right)$$

График такой функции представлен на рисунке 2.

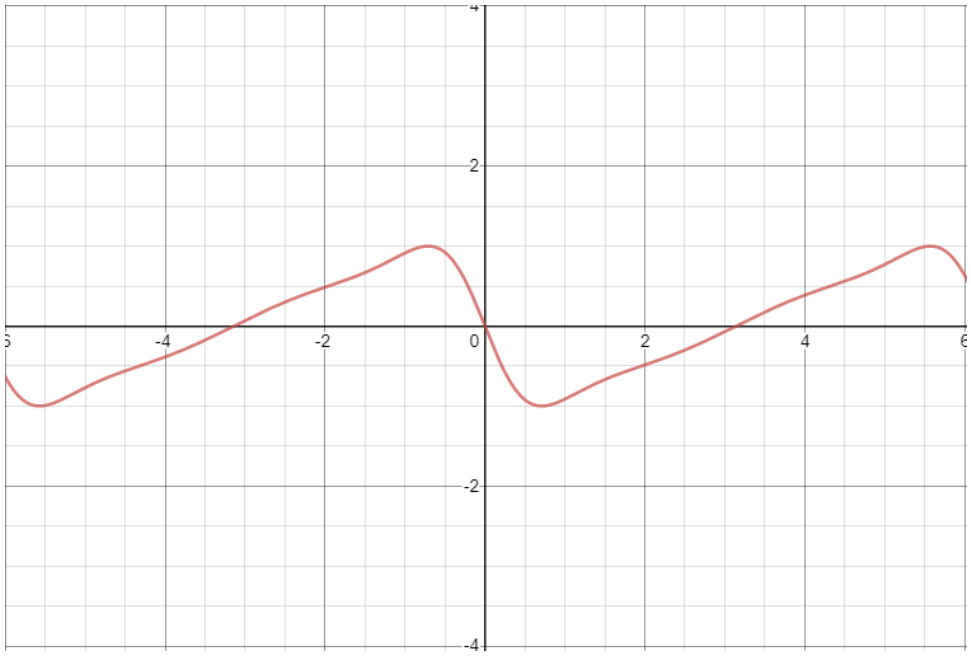

Рис. 2

Исходя из приведенного графика, движение капли можно описать следующим образом: капля быстро стекает вниз, а затем медленно поднимается вверх.

Теперь осталось добавить скорость смещения самой сетки по оси ординат. Для обеспечения правдоподобности, скорость смещения должна быть равна скорости капли в момент её подъема наверх, это создаст эффект прилипания капли к окну.

Помимо самой капли, можно сделать “хвост”, который следует за её движениями. С помощью ранее описанных команд smoothstep и frac() создаются несколько маленьких капель позади основной.

3.2 Сравнение методов

Краткое сравнение предложенных методов создания визуального эффекта: (таблица)

Вид шейдера	Дополнительные улучшения	Основная идея алгоритма	Причина использования
Simple Raindrops	Нет	Циклическое сжатие/расширение капли	Простота использования
Rain Window	Размытие окна	Циклическое движение капли вместе с содержащей её сеткой	Реалистичность эффекта

Таблица 1

Как видно из Таблицы 1, у методов есть как преимущества, так и недостатки.

4. ПРАКТИЧЕСКАЯ РЕАЛИЗАЦИЯ ОПИСАННЫХ МЕТОДОВ

Основываясь, на исследовательской работе из предыдущего раздела необходимо реализовать вышеизложенные шейдеры, и провести визуальное сравнение, чтобы сделать выводы о применимости этих методов.

В качестве фонового изображения использовалось изображение ночного города (Рис.3)

Рис.3 Ночной город

Исходный код полученных шейдеров представлен в Приложении А и Б.

4.1 Simple Raindrops

Демонстрация результата:

\

Рис.4

4.2 Rain Window

Демонстрация результата:

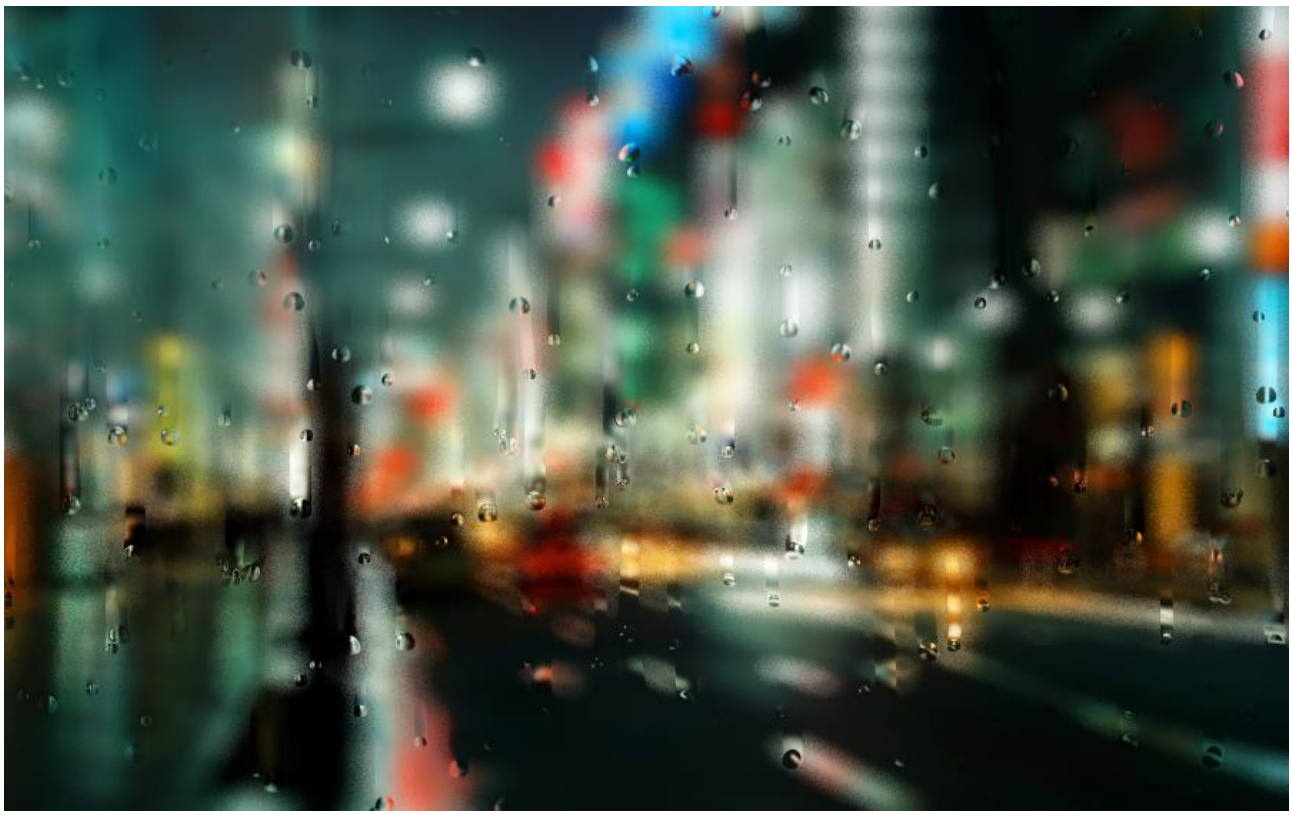

Рис.5

4.3 Результаты

Основываясь на полученных результатах, сложно сделать вывод о том, какой метод показал себя наилучшим образом. Каждый метод имеет как преимущества, так и недостатки. В первом случае, метод достаточно легко реализуем и не требует особых познаний, однако сам результат нельзя назвать достаточно реалистичным. Второй метод, наоборот, требует более глубоких знаний в работе с шейдерами, но при этом создаваемый им визуальный эффект выглядит наиболее правдоподобно.

ЗАКЛЮЧЕНИЕ

В результате выполненной научной исследовательской работы была проделана работа в области воссоздания визуального эффекта “капли дождя на окне” с использованием шейдерных программ и среды разработки Unity. В результате тестирования было выяснено, что каждый из представленных методов имеет как преимущества, так и недостатки. Если выбирать самый качественный метод, то шейдер Rain Window даёт более реалистичный эффект.

В целом, написание шейдеров — это математика и понимание физики эффектов. Это в чём-то похоже на смешивание ингредиентов, если не решается конкретная задача физического моделирования. Много любопытного можно сделать, смешивая между собой шум, преломление, подповерхностное рассеивание света, реакцию диффузии и прочие физические свойства объектов. Шейдерное программирование — это безусловно не элементарно, и там есть куда копать в глубину.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Hohn Kessenich, Dave Baldwin, and Randi Rost. 2014. The OpenGL© Shading Language (Version 4.50). https://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf. (дата обращения: 10.12.2019).

2. Khronos Group, Inc. 2009. ARB_shader_subroutine. https://www.opengl.org/registry/ specs/ARB/shader_subroutine.txt (дата обращения: 10.12.2019).

3. Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. 2002. Shader Metaprogramming, 2010.

4. Natalya Tatarchuk. 2006. Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows. In Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games (I3D ’06). ACM, New York, NY, USA, 63–69

5. Ivan Selesnick .Least Squares with Examples in Signal Processing// NYU-Poly, 7 марта, 2013.

6. Gabriel Peyre´. The Numerical Tours of Signal Processing- Advanced Computational Signal and Image Processing//IEEE Computing in Science and Engineering, 13 (4), стр.94-97, 2011.

7. Fikret Isık Karahanoglu, Ilker Bayram, Dimitri Van De Ville. A Signal Processing Approach to Generalized 1D Total Variation// IEEE Transactions on Signal Processing (Volume: 59, Issue: 11, Nov. 2011), стр. 5265-5274

ПРИЛОЖЕНИЕ А

ШЕЙДЕР SIMPLE RAINDROPS

```
Shader "Unlit/SimpleRaindrops"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _SecondTex("Texture", 2D) = "white" {}
        _Intensity("Intensity",Range(0,5)) = 1
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" "Queue" = "Transparent"}
        LOD 100
        GrabPass {"_GrabTexture"}

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // make fog work
            #pragma multi_compile_fog

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                float4 grabUv : TEXCOORD1;
                UNITY_FOG_COORDS(1)
                float4 vertex : SV_POSITION;
            };
```

```
            sampler2D _MainTex, _GrabTexture;
            sampler2D _SecondTex;
            float _Intensity;
            float4 _MainTex_ST;

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                UNITY_TRANSFER_FOG(o,o.vertex);
                o.grabUv = UNITY_PROJ_COORD(ComputeGrabScreenPos(o.vertex));
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {

            fixed2 u = i.uv,
            n = tex2D(_SecondTex, u).rg;

            float4 col = tex2Dproj(_GrabTexture, i.grabUv);

            for (fixed r = 4; r > 0.; r--) {
                fixed2 x = _ScreenParams.xy * r * .055,
                p = 6.28 * u * x + (n - .5) * 2.,
                s = sin(p);

                 fixed4 d = tex2D(_SecondTex, round(u * x - 0.25) / x);

                fixed t = (s.x + s.y) * max(0., 0.5 - frac(_Intensity *_Time.y
* (d.b + .1) + d.g) * 2.);

              float2 projUv = i.grabUv.xy / i.grabUv.w;

               if (d.r < (5. - r)*.08 && t > .5) {
                 fixed3 v = normalize(-fixed3(cos(p), lerp(.2, 2., t - .5)));
                 return tex2D(_GrabTexture, projUv - v.xy * .1); //преломление
света
                }
```

```
            }
            return col;


        }
      ENDCG
    }
  }
}
```

ПРИЛОЖЕНИЕ Б

ШЕЙДЕР RAIN WINDOW

```
Shader "Unlit/RainWindow"
{
    Properties
    {
        _MainTex("Texture", 2D) = "white" {}
        _Aspect("Aspect: x / y", range(0.1,10)) = 2
        _Size("Size", int) = 5
        _CircleIn("Circle in", float) = 0.05
        _CircleOut("Circle out", float) = 0.03
        _Speed("Drop Speed", range(0,10)) = 1
        _Distortion("Distortion", range(0,1)) = 1
        _Blur("Blur", range(0,1)) = 1
        _GlassTex("GlassMask", 2D) = "white" {}
        _HorFactor("HOriontal", float) = 1
        _VertFactor("Vert", float) = 1
    }
        SubShader
    {
        Tags { "RenderType" = "Opaque" "Queue" = "Transparent"}
        LOD 100
        GrabPass {"_GrabTexture"}
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float2 uv : TEXCOORD0;
            };

            struct v2f
            {
                float2 uv : TEXCOORD0;
                float4 grabUv : TEXCOORD1;
                float4 vertex : SV_POSITION;
            };

            sampler2D _MainTex, _GrabTexture, _GlassTex;
            float4 _MainTex_ST;
```

```
            float _HorFactor, _VertFactor;

            v2f vert(appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                o.uv = TRANSFORM_TEX(v.uv, _MainTex);
                o.grabUv = UNITY_PROJ_COORD(ComputeGrabScreenPos(o.vertex));
                return o;
            }

            float N21(float2 p) {
                p = frac(p*float2(242.12, 25.123));
                p += dot(p, p + 24.01);
                return frac(p.x*p.y);
            }


            // calc one drop layer
            int _Size;
            float _CircleIn;
            float _CircleOut;
            float3  Layer(float2 UV, float t, float2 aspect, float argOne,
float argTwo) {
                float2 asp = float2(2, 1);
                float2 uv = UV * _Size * asp;
                float4 data = tex2D(_GlassTex, UV);
                // uv
                uv.y += t * 0.25;
                //if (data.r > .4) uv.x -= t * .25;
                //else if (data.r < .3) uv.x += t * .25;

                float2 gv = frac(uv) - 0.5;
                float2 id = floor(uv);
                float n = N21(id);
                t += n * 6.28631;

                // position of time
                float w = UV.y * 10;
                float x = (n - .5) * .8;

                x += (.4 - abs(x)) * sin(3 * w) * pow(sin(w), 6) * .45;
                float y = -sin(t + sin(t + sin(t) * 0.5)) * .45;
                    y -= (gv.x - x)*(gv.x - x);
                    // rain point
                    float2 drop_pos = (gv - float2(x,y))/ aspect;
                    float drop = smoothstep(0.05, 0.03, length(drop_pos));

                    // rain trail
                    float2 drop_trail_pos = (gv - float2(x,t*0.25)) / aspect;
                    //drop_trail_pos.y = (frac(drop_trail_pos.y * 8) / 8) -
_CircleOut;
                    drop_trail_pos.y = (frac(drop_trail_pos.y * 8) - 0.5) / 8;

                    float drop_trail = smoothstep(.03, .01,
length(drop_trail_pos));
                    float fog_trail = smoothstep(-.05, .05, drop_pos.y);
                    fog_trail *= smoothstep(0.5, y, gv.y);
                    drop_trail *= fog_trail;// *0.22;

                    // blur mod
                    fog_trail *= smoothstep(.05, .04, abs(drop_pos.x));
```

```
                    float2 offs = drop * drop_pos;// +drop_trail *
drop_trail_pos;
                    //offs.x += data.x * _HorFactor;
                    //offs.y += data.y * _VertFactor;

                    return float3(offs, fog_trail);
                }



                float timed_angle(float2 pos, float rawtime) {
                    float anglePercent = atan2(pos.y, pos.x) / 6.2831 + 0.5;
                    float time = fmod(rawtime, 3.0) - 1.0;
                    return fmod(anglePercent + time, 1.0);
                }

                sampler2D _FogTex;
                // merge texture with drop
                float _Distortion;
                float _Blur;
                int _Aspect;
                float _Speed;
                fixed4 frag(v2f i) : SV_Target{
                    float t = fmod(_Time.y, 7200) * _Speed;

                    float4 col = 0;
                    float2 aspect = float2(_Aspect, 1);
                    float3 drops = Layer(i.uv, t, aspect, 1 , 0);
                    //drops += Layer(i.uv * 1.35 + 7.51, t, aspect, 1, 0);
                    drops += Layer(i.uv * 0.95 + 1.54, t, aspect, 1, 0);
                    drops += Layer(i.uv * 1.57 + 6.54, t, aspect, 1, 0);

                    float fade = 1 - saturate(fwidth(i.uv) * 50);
                    float blur = _Blur * 7 * (1 - drops.z * fade);


                    blur *= .1;

                    //col = tex2Dlod(_MainTex, float4(uv +
drops.xy*_Distortion, 0, blur));

                    float WiperUv = i.uv;
                    float2 projUv = i.grabUv.xy / i.grabUv.w;
                    projUv += drops.xy * _Distortion * fade;

                    const float numSamples = 16;
                    float a = N21(i.uv) * 6.2831;
                    for (float i = 0; i < numSamples; i++) {
                        float2 offs = float2(sin(a), cos(a))*blur;
                        float d = frac(sin((i + 1)* 546.)*5424.);
                        d = sqrt(d);
                        offs *= d;
                        col += tex2D(_GrabTexture, projUv + offs);
                        a++;
                    }
                    col /= numSamples;
                    return col;
                }

                ENDCG
            }
    }
}
```