




ARD OPDRACHT



youri mulder
1716390

Inhoud

Introductie	2
Scene.....	3
Sensing	4
LightHandler.....	4
FlagHandler	6
Shader	7
Properties.....	7
SubShader	7
Vertex shader	7
Fragment shader	10
Bijlage FlagShader	11
Bijlage sensing LightHandler	13
Bijlage sensing FlagHandler	15

Introductie

Voor deze opdracht heb ik gekozen om een vlag te laten wapperen. De uitwerking bestaat uit een Nederlandse vlag die op een constante snelheid, omvang en frequentie heen en weer gaat. De vlag wordt dus niet van vorm veranderd op basis van de input van een wind sensor, of wind data.

De applicatie is gemaakt om specifiek op het Android platform te draaien. Het is momenteel niet mogelijk om deze applicatie op een IOS platform te draaien, omdat er Android specifieke code in zit.

In de scene bevindt zich een directional light, waarbij de intensiteit wordt aangepast op basis van een sensor. De sensor die hiervoor gebruikt is, is de light sensor. De light sensor code is specifiek voor een Android platform.

Het licht wordt door de shader gebruikt om de juiste kleur aan de vlag te geven in combinatie met de texture die op de vlag is geplaatst.

De applicatie is een AR applicatie. De AR applicatie maakt gebruik van Vuforia. De vlag wordt geplaatst op een marker die op de foto is te zien onder de vlag. Vuforia maakt gebruik van de camera om de marker te detecteren en te volgen. Voor meer informatie kijk op <https://developer.vuforia.com/>.

Als het apparaat waar de applicatie op draait wordt verplaatst zal de vlag op de marker blijven staan zolang de marker in beeld is. Het is daardoor mogelijk om een rondje om de vlag heen te draaien en de vlag van verschillende perspectieven te bekijken.



Scene

De scene bestaat uit verschillende onderdelen die samen de applicatie vormen. De scene is opgebouwd door onderdelen uit Unity en Vuforia. De onderdelen uit Vuforia worden gebruikt om argumented reality te realiseren. De andere onderdelen vormen het object dat aangepast wordt door de shader en het licht in de scene.

Entity	Afkomst	Details
ARCamera	Vuforia	Wordt gebruikt als camera waar de gebruiker vanuit de scene in kijkt. Camera positie wordt bepaald door de positie van de telefoon.
Directional Light	Unity	De lichtbron van de scene.
ImageTarget Cylinder Plane	Vuforia Unity Unity	Het plaatje waar naar gezocht wordt om de vlag op te plaatsen. Als het plaatje in de echte wereld is gevonden wordt de vlag erop geplaatst. Die bestaat uit een vlaggenstok (cylinder) en de vlag (plane).
EventSystem	Vuforia	Handelt de input voor Vuforia af zodat de vlag op de juiste plek komt en blijft.

Entity	Script	Shader
Directional Light	LightHandler	-
Cylinder (vlaggenstok)	-	Standard (Unity)
Plane (vlag)	FlagHandler	FlagShader (custom)

Sensing

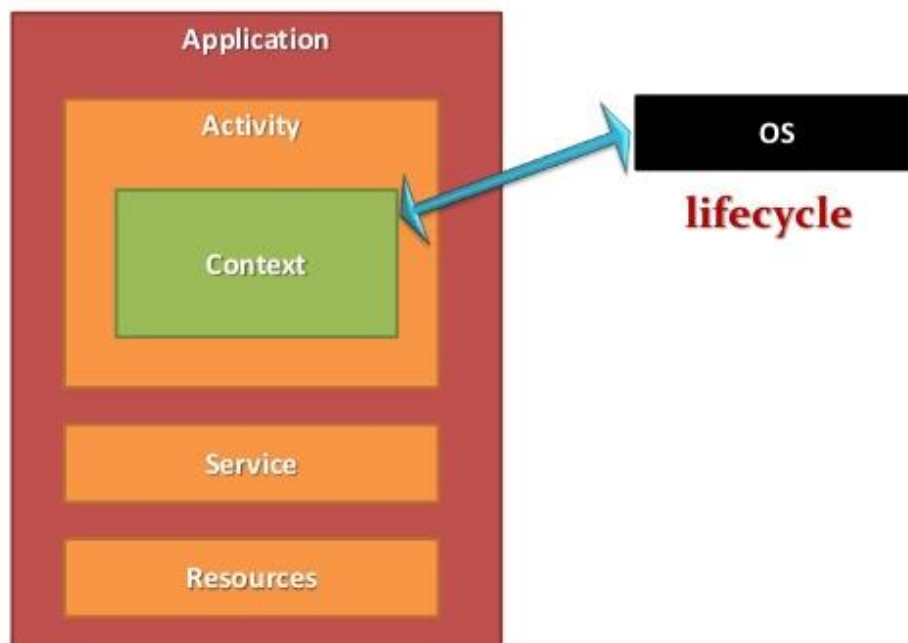
LightHandler

Dit script is gekoppeld aan Directional Light en zorgt er voor dat de lichtsterkte wordt veranderd. De lichtsterkte wordt aangepast op basis van de licht sensor op een Android telefoon. De klasse GetLux is gemaakt om deze sensor uit te lezen. Deze code bestaat uit twee belangrijke onderdelen.

```
public static void Start() {  
    #if UNITY_ANDROID  
        activityClass = new AndroidJavaClass("com.unity3d.player.UnityPlayer");  
        activityContext = activityClass.GetStatic<AndroidJavaObject>("currentActivity");  
  
        jo = new AndroidJavaObject("com.etiennefrank.lightsensorlib.LightSensorLib");  
        jo.Call("init", activityContext);  
    #endif  
}
```

De bovenstaande functie kijkt door middel van de preprocessor of de applicatie voor een Android apparaat wordt gecompileerd. Dan wordt de juiste Activity en Context gemaakt om te kunnen gebruiken voor de licht sensor. Door het gebruik van de activityContext.

Application Structure



Elke Android applicatie start als een activity, een activity is een proces. Elke Android applicatie is zijn eigen proces. De activity wordt overgeërfd van een context. De context is een handle voor systeem resources. De context kan bijvoorbeeld lokale bestanden en databases op de telefoon benaderen. In dit geval wordt de context gebruikt om de licht sensor van de telefoon uit te lezen.

```

public static float GetFloat() {
#if UNITY_ANDROID
    return jo.Call<float>("getLux");
#endif
}

```

Nadat de context is verkregen wordt er gebruik gemaakt van een library om de licht sensor uit te lezen. Unity geeft ons de mogelijkheid om een klasse aan te maken die java methodes kan uitvoeren. Door middel van `AndroidJavaObject.Call` is het mogelijk om het licht op te vragen via de eerder gemaakte context.

De lux wordt hier terug gegeven als float. De maximale waarde van de lux is afhankelijk van het apparaat waar het op wordt gedraaid. De minimale waarde van de lux is 0.

De klasse die gekoppeld is aan de Directional light wordt hieronder besproken. De `GetLux` klasse is een helpende klasse om de code beter te organiseren.

```

public class LightHandler : MonoBehaviour {
    public Camera main_camera;
    public Light light;

    private const float lux_max = 100;
    private const float light_intensity_max = 1;

    // Start is called before the first frame update
    void Start() {
        GetLux.Start();
        main_camera = Camera.main;
        light = GetComponent<Light>();
    }
}

```

De maximale lux waarde wordt op 100 gemaximaliseerd. En de maximale lichtsterkte op 1. Dit om te voorkomen dat de vlag overbelicht raakt.

De `Start()` wordt aangeroepen voor het eerste beeld op het scherm komt. Hier wordt alles in geïnitieerd.

```

void Update() {
    transform.position = main_camera.transform.position;

    float lux = GetLux.GetFloat();
    light.intensity = CalculateNewLightIntensity(lux);
}

```

Elke update cyclus wordt het volgende gedaan -> De licht positie wordt aangepast naar de huidige positie van de ARCamera. Zo lijkt het als het licht uit de telefoon komt. Daarna wordt de nieuwe lux waarde uit `GetLux` gehaald en de nieuwe lichtsterkte berekend en toegepast.

```

float CalculateNewLightIntensity(float lux) {
    lux = ClipLux(lux);
    const float lux_light_intensity_factor = light_intensity_max / lux_max;
    return lux * lux_light_intensity_factor;
}

```

De lux wordt als hij hoger is dan de maximale waarde op de maximale waarde gezet en daarna wordt de nieuwe lichtsterkte berekend. De maximale waarde van de lux is gekoppeld aan de maximale waarde van de lichtbron. Als de lux om zijn maximale waarde is moet de lichtbron dat ook zijn. Dit door middel van de factor te berekenen tussen de maximale waardes en dat te vermenigvuldigen met de huidige lux waarde. Zo krijg je een nieuwe intensiteit tussen de 0-1 in dit geval.

Lux	Lichtsterkte
100	1
50	0.5
25	0.25
0	0

Let op: Dit is niet gebaseerd op natuurkundige wetten.

FlagHandler

Deze code zorgt ervoor dat de vlag geïnitieerd wordt met de juiste begin waardes.

```
float speed_current = 10f;
float frequency_current = 1f;
float amplitude_current = 0.2f;

protected void UpdateValues() {
    GetComponent<Renderer>().material.SetFloat("_Speed", speed_current);
    GetComponent<Renderer>().material.SetFloat("_Frequency", frequency_current);
    GetComponent<Renderer>().material.SetFloat("_Amplitude", amplitude_current);
}

protected void Start() {
    UpdateValues();
}
```

Deze waardes zijn tot stand gekomen door te testen welke de mooiste uitput gaven. Er zijn verschillende waardes geprobeerd en uiteindelijk bleken deze waardes de mooist bewegende vlag te geven. In eerste instantie is er geprobeerd om de snelheid, frequentie en sterkte te veranderen door een gyroscoop. Dit bleek geen mooi effect te geven, want deze waardes worden doorgegeven aan een sinus functie. Als de snelheid veranderd komt er een andere uitkomst uit deze sinus functie. Daarom lijkt het heel schokkerig als de snelheid of een andere waarde wordt aangepast.

Hierdoor is er gekozen om de sensor code te schrijven voor de lichtbron.

Shader

Properties

Om invoer te hebben voor de shader zijn er een aantal properties aangemaakt. De eerste is `_MainTex`. `_MainTex` is de texture die op het object wordt geprojecteerd. In dit geval is dat de Nederlandse vlag. Daarna moet er een kleur gespecificeerd zijn voor de diffuse reflectie. Hiervoor is gekozen om standaard wit te hanteren. De laatste drie waardes zijn besproken in [Sensing FlagHandler](#).

```
Properties {  
    _MainTex("Texture", 2D) = "white" {}  
    _Color("Overall Diffuse Color Filter", Color) = (1,1,1,1)  
    _Speed("Speed", Range(0,20)) = 1  
    _Frequency("Frequency", Range(0,3)) = 1  
    _Amplitude("Amplitude", Range(0,0.5)) = 1  
}
```

SubShader

```
Tags { "RenderType" = "Opaque" }  
Cull Back
```

Opaque is de standard instelling voor `RenderType`. Dit is een standaard instelling en dat heb ik zo gelaten.

Voor de culling is gekozen om deze off te zetten. Zodat alleen de achterkant van het object zichtbaar is. Hier is niet specifiek voor gekozen om het zo mooi mogelijk er uit te laten zien, maar om te testen wat de culling doet. Als je aan de achterkant van het object kijkt zie je nu het object niet verschijnen en kijk je er door heen.

Vertex shader

```
struct AppData {  
    float4 vertex : POSITION;  
    float3 normal : NORMAL;  
    float2 uv : TEXCOORD0;  
};
```

De input van de vertex shader zijn de bovenstaande variabelen binnen de `AppData` struct. De vertex shader ontvangt een vertex, de normaal op deze vertex en de positie waar het eerste punt van de texture moet komen.

De inhoud van de vertex shader bestaat uit twee delen. Het eerste deel wordt gebruikt om de animatie te berekenen en het tweede deel om de licht reflectie te berekenen.

```
v2f vert(AppData input) {  
    v2f output;  
  
    float3 newNormal = input.normal * -1;  
    // animation  
    output.vertex = UnityObjectToClipPos(input.vertex);  
    float worldPos = mul(unity_ObjectToWorld, input.vertex).xyz;  
  
    float y_movement = sin(  
        (worldPos.x + _Time.y * _Speed) * _Frequency
```



```

    ) * _Amplitude * (input.vertex.x - 5);
    output.vertex.y += y_movement;

    // light
    float3 normalDirection = UnityObjectToWorldNormal(newNormal);
    float3 lightDirection = normalize(_WorldSpaceLightPos0.xyz);

    float3 diffuseReflection = _LightColor0.rgb * _Color.rgb
        * max(0.0, dot(normalDirection, lightDirection));

    output.color = float4(diffuseReflection, 1.0);
    output.uv = TRANSFORM_TEX(input.uv, _MainTex);

    return output;
}

```

Als eerste wordt de normaal vector omgedraaid. Dit is nodig, omdat ik de plane die de vlag maakt omgedraaid is.

Animatie

Daarna wordt de input vertex gebruikt om de output vertex te berekenen. De output vertex wordt door middel van UnityObjectToClipPos omgerekend naar een homogeen coördinaat. Eerst wordt de vertex van object naar view space getransformeerd door middel van unity_ObjectToWorld. Dit wordt opgeslagen in worldPos. Deze informatie wordt later gebruikt om de animatie af te doen.

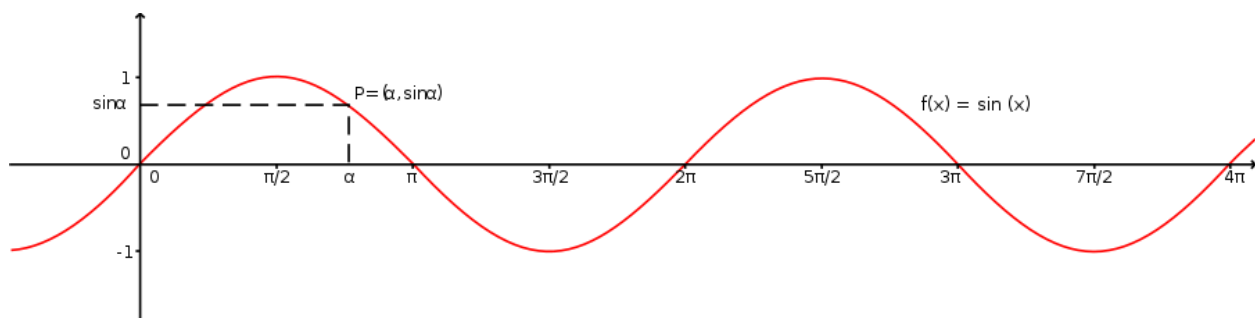
Het belangrijkste deel is de berekening van y_movement. Dit is de relatieve beweging vanaf het begin punt van de vertex in de vlag. Op basis hiervan wordt de nieuwe positie bepaald die de animatie van de vlag vormt.

In eerste instantie is er geprobeerd de animatie op de z as te doen. Dit was echter niet mogelijk met deze configuratie. Zodra de z positie van de output vertex werd veranderd, werd de vlag transparant op die plek. Verschillende oplossingen zijn geprobeerd. Uit onderzoek blijkt dit te liggen aan z-buffer in combinatie met z-depth. Door middel van de z-buffer worden alleen de dichtstbijzijnde pixels weergegeven. Waardoor er een transparante uitput komt.

```

float y_movement = sin(
    (worldPos.x + _Time.y * _Speed) * _Frequency
) * _Amplitude * (input.vertex.x - 5);

```



De beweging van een vlag lijkt op een sinus/cosinus. De eerste paar vertices bewegen niet door de laatste toevoeging aan de formule ($\text{input.vertex.x} - 5$). Daarnaast wordt de sinus versterkt door middel van

de `_Amplitude`. De uitkomst van de sinus is altijd tussen de 0-1, door middel van de `_Amplitude` kan je dit vermeerderen of verminderen. De input van de sinus is een combinatie van de huidige wereld positie in de wereld, de huidige tijd en de snelheid. De positie en snelheid zijn in dit geval constant. Hoe hoger de snelheid des te meer de tijd wordt meegenomen in de functie. `_Time` is de tijd sinds het level is geladen. Dit zal dus met dezelfde snelheid oplopen, namelijk 1 per seconde. Als laatste de `_Frequency`, de `_Frequency` is een constante wordt gebruikt om de sinus sneller of minder snel van 0-1 te laten gaan. Hoe hoger het getal, des te groter de volgende input wordt van de sinus wordt. Waardoor hij sneller van 0 naar 1 zou gaan of andersom.

Nadat de verandering op de y positie aan de vertex is toegevoegd wordt ook aan de normaal vector deze verandering toegevoegd. Dit is om later het licht op de juiste locatie toe te passen.

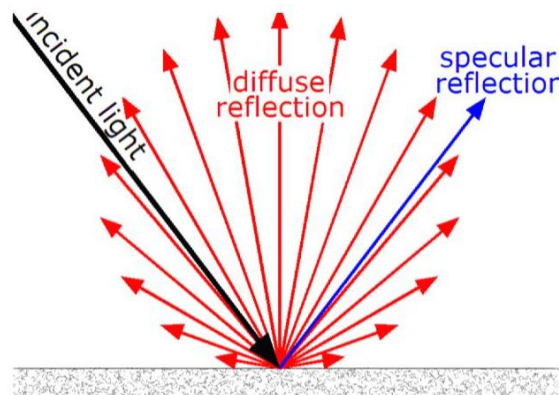
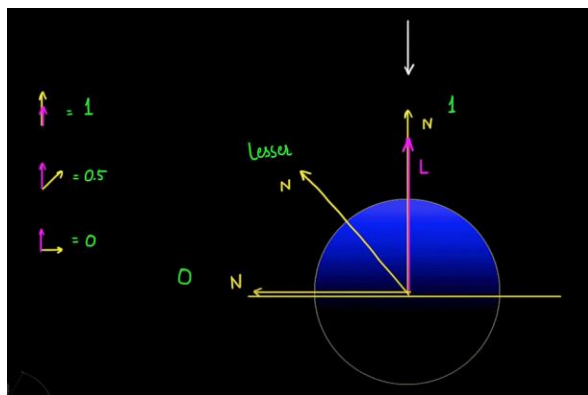
Licht

Er is gekozen om diffuse reflectie toe te passen op de vlag. Diffuse reflectie is gerealiseerd door de richting van de normaal vector van de vertex te berekenen in world space, en de richting van de normaal vector van licht in world space. Deze kunnen worden gebruikt om de diffuse reflectie te berekenen. Voor de diffuse reflectie zijn er nog twee andere waarden nodig. Namelijk de kleur van de diffuse reflectie. Die is standaard ingesteld op wit in de shader en de kleur van de lichtbron. Door deze te vermenigvuldigen krijg je de uitkomst van de kleur van de diffuse reflectie.

```
float3 diffuseReflection = _LightColor0.rgb * _Color.rgb
    * max(0.0, dot(normalDirection, lightDirection));
```

Er wordt het inproduct van de twee richtingsvectoren genomen. Als de uitkomst onder 0 komt zal hij altijd op 0 gezet worden. Dit betekent dat er geen reflectie plaats zal vinden. Als de twee vectoren loodrecht op elkaar staan zal de diffuse reflectie 0 zijn. Als de hoek groter is dan 90° dan zal er een min getal uit komen.

De diffuse reflectie zal als 4 dimensionale vector in de output color gestopt worden, waarbij de alpha waarde op 1 gezet wordt. De output color wordt later gebruikt in de fragment shader. Die de kleur per pixel bepaald.



Fragment shader

```
fixed4 frag(v2f input) : SV_Target {  
    return tex2D(_MainTex, input.uv) * input.color;  
}
```

De fragment shader pakt de juiste pixel uit de shader door middel van de tex2D functie. Die de texture meekrijgt en de texture coordinate. Zo kan via tex2D bepaald worden welke pixel op de texture op het object geprojecteerd wordt.

Door de kleur van de vertex shader hiermee te vermenigvuldigen wordt de diffuse reflectie toegepast op de vlag.

Bijlage FlagShader

// Upgrade NOTE: replaced 'mul(UNITY_MATRIX_MVP,*)' with 'UnityObjectToClipPos(*)'

```
Shader "CustomShaders/FlagShader" {
    Properties {
        _MainTex("Texture", 2D) = "white" {}
        _Color("Overall Diffuse Color Filter", Color) = (1,1,1,1)
        _Speed("Speed", Range(0,20)) = 0
        _Frequency("Frequency", Range(0,3)) = 0
        _Amplitude("Amplitude", Range(0,0.5)) = 0
    }

    SubShader {
        Tags { "RenderType" = "Opaque" }
        Cull Back

        Pass {
            Tags { "LightMode" = "ForwardBase" }
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            uniform float4 _LightColor0;

            struct AppData {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
                float2 uv : TEXCOORD0;
            };

            struct v2f {
                float4 vertex : SV_POSITION;
                float4 color : COLOR0;
                float2 uv : TEXCOORD0;
            };

            sampler2D _MainTex;
            float4 _MainTex_ST;
            uniform float4 _Color;

            float _Speed;
            float _Frequency;
            float _Amplitude;

            v2f vert(AppData input) {
                v2f output;

                float3 newNormal = input.normal * -1;
                // animation
                output.vertex = UnityObjectToClipPos(input.vertex);
                float worldPos = mul(unity_ObjectToWorld, input.vertex).xyz;

                float y_movement = sin(
                    (worldPos.x + _Time.y * _Speed) * _Frequency
                ) * _Amplitude * (input.vertex.x - 5);
```

```

        output.vertex.y += y_movement;

        // light
        float3 normalDirection = UnityObjectToWorldNormal(newNormal);
        float3 lightDirection = normalize(_WorldSpaceLightPos0.xyz);

        float3 diffuseReflection = _LightColor0.rgb * _Color.rgb
            * max(0.0, dot(normalDirection, lightDirection));

        output.color = float4(diffuseReflection, 1.0);

        output.uv = TRANSFORM_TEX(input.uv, _MainTex);

        return output;
    }

    fixed4 frag(v2f input) : SV_Target {
        return tex2D(_MainTex, input.uv) * input.color;
    }

    ENDCG
}
Fallback "Diffuse"
}

```

Bijlage sensing LightHandler

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public static class GetLux {

    private static AndroidJavaObject activityContext = null;
    private static AndroidJavaObject jo = null;
    private static AndroidJavaClass activityClass = null;

    public static void Start() {
        #if UNITY_ANDROID
            activityClass = new AndroidJavaClass("com.unity3d.player.UnityPlayer");
            activityContext = activityClass.GetStatic<AndroidJavaObject>("currentActivity");

            jo = new AndroidJavaObject("com.etiennefrank.lightsensorlib.LightSensorLib");
            jo.Call("init", activityContext);
        #endif
    }

    public static float GetFloat() {
        #if UNITY_ANDROID
            return jo.Call<float>("getLux");
        #endif
    }
}

public class LightHandler : MonoBehaviour {
    public Camera main_camera;
    public Light light;

    private const float lux_max = 100;
    private const float light_intensity_max = 1;

    // Start is called before the first frame update
    void Start() {
        GetLux.Start();
        main_camera = Camera.main;
        light = GetComponent<Light>();
    }

    float ClipLux(float lux) {
        lux = lux > lux_max ? lux_max : lux;
        return lux;
    }

    float CalculateNewLightIntensity(float lux) {
        lux = ClipLux(lux);
        const float lux_light_intensity_factor = light_intensity_max / lux_max;
        return lux * lux_light_intensity_factor;
    }

    void Update() {
```

```
transform.position = main_camera.transform.position;

float lux = GetLux.GetFloat();
light.intensity = CalculateNewLightIntensity(lux);
}

void OnGUI() {
    GUI.Label(new Rect(500, 300, 200, 40), "Lux: " + GetLux.GetFloat().ToString());
}
}
```

Bijlage sensing FlagHandler

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FlagHandler : MonoBehaviour
{
    float speed_current = 10f;
    float frequency_current = 1f;
    float amplitude_current = 0.2f;

    // Update is called once per frame
    protected void UpdateValues() {
        GetComponent<Renderer>().material.SetFloat("_Speed", speed_current);
        GetComponent<Renderer>().material.SetFloat("_Frequency", frequency_current);
        GetComponent<Renderer>().material.SetFloat("_Amplitude", amplitude_current);
    }

    protected void Start() {
        UpdateValues();
    }

    void OnGUI() {
        GUI.Label(new Rect(500, 500, 200, 40), "speed: " +
        GetComponent<Renderer>().material.GetFloat("_Speed").ToString());
        GUI.Label(new Rect(500, 600, 200, 40), "freq: " +
        GetComponent<Renderer>().material.GetFloat("_Frequency").ToString());
        GUI.Label(new Rect(500, 700, 200, 40), "amp: " +
        GetComponent<Renderer>().material.GetFloat("_Amplitude").ToString());
    }
}
```