UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Spring 2020

PROGRAMMING

LEVEL 3 – Advanced SFX

Maurizio Rigamonti

Particle systems

PARTICLE SYSTEMS (1/2)

- Technique used to simulate many effects
  - Fire, snow, sand, water, sparks, dust
  - Flying birds
  - Abstract effect
- Mathematical formalism to describe phenomena that are
  - Dynamic and time dependent
  - Highly parallel with small individual components
  - Complex

PARTICLE SYSTEMS (2/2)

- **Local** or **global**
  - The particles are all independent
  - Particles interact and react to each other
- Often in 3D games, but useful even in 2D
- 2 main components
  - Particles
  - The engine itself

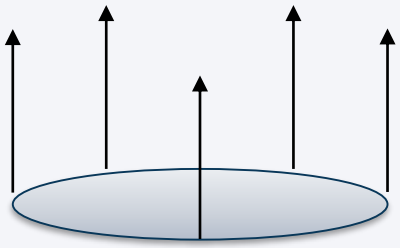
PARTICLE

- Specific properties for each particle
  - Behavior
    - Position, velocity, angle, angular velocity, etc.
  - Appearance
    - Size, shape, color, blending mode, texture, etc.
- Particles have a **lifetime**
- The number of parameters is related with the quality of the simulation

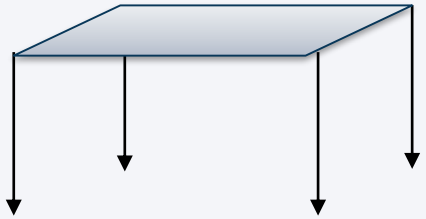
A PARTICLE SYSTEMS

- The system (or engine) defines
  - How particle are born (emitter)
  - What simulation process they undergo
- Emitters

explosion

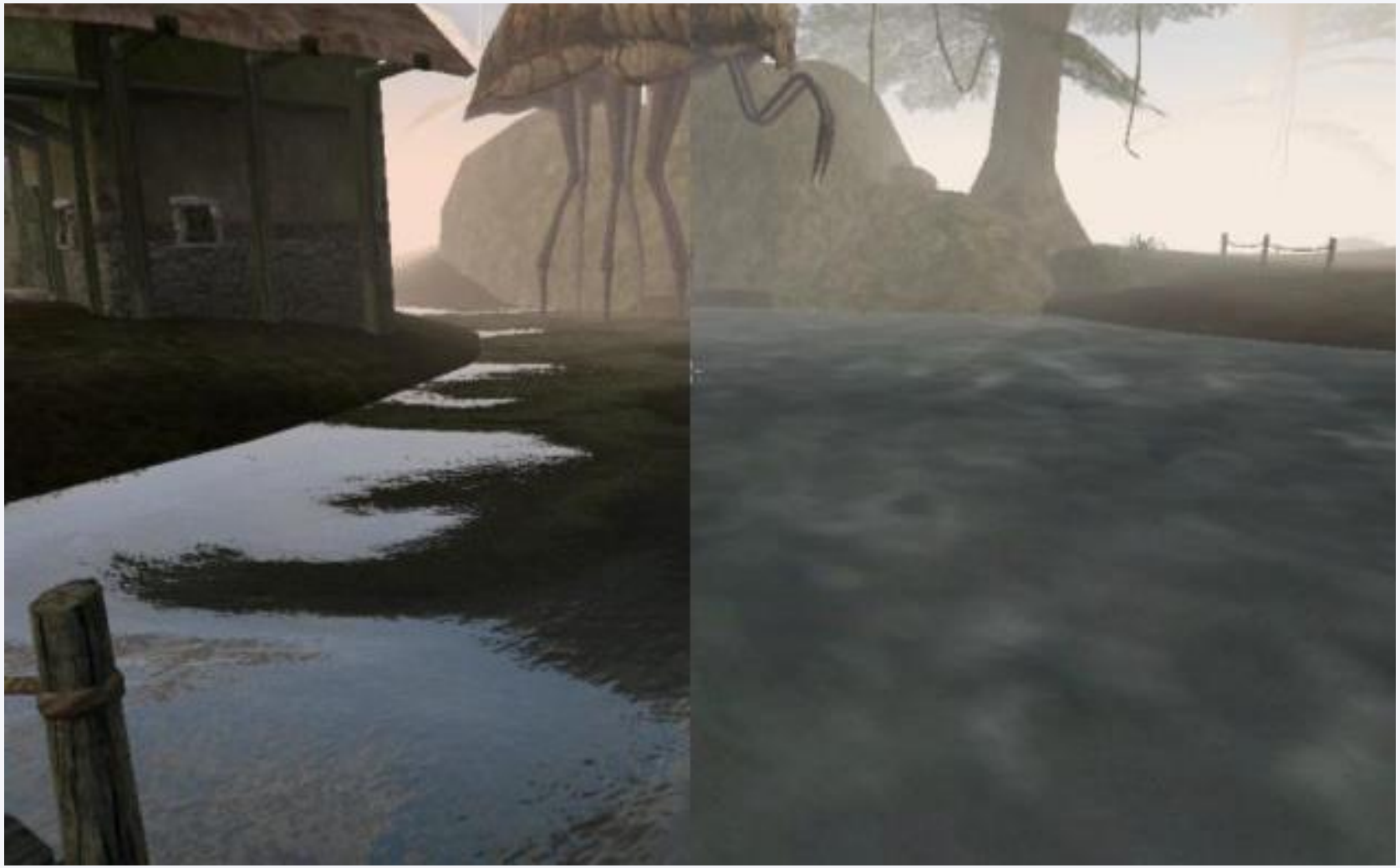


fire



rain

Shaders



SHADERS

- **Programs** to do shading
  - Act on levels of light, colors, etc.
  - Special effects (SFX)
  - Post production
- Most of them for **GPU** (graphical processing unit)
- *Very appreciated skill for programmers in the domain*

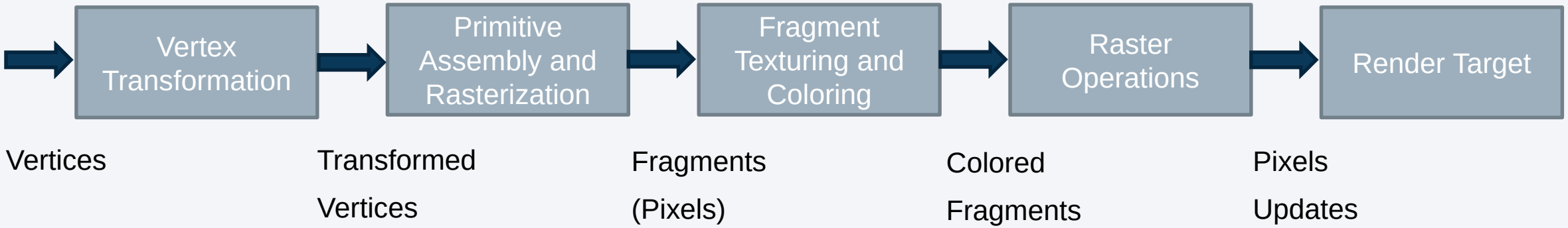
ADVANCED GRAPHIC EFFECTS

- Lightning, texturing, bump mapping, normal mapping, etc.

- Usually used in 3D graphics, but we can use them even for 2D games

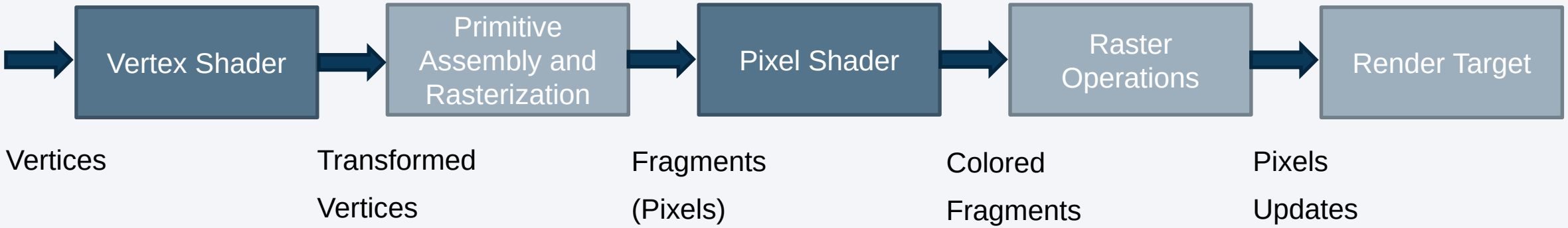
THE 3D PIPELINE: FFP

- In the past, **fixed function pipeline** (FFP)
  - Limited set of function to configure how to draw models and textures on the graphics card
  - Most of games shared the same look and feel



THE 3D PIPELINE

- Today, **programmable graphics pipelines**
  - Special programs compiled and sent over to the graphics card: *shaders*
  - 2000: NVIDIA GeForce 3



LANGUAGES FOR SHADERS

- High Level Shader Language (**HLSL**)
  - From Direct X9
  - Cg, a very similar language (NVIDIA)
- OpenGL Shading Language (**GLSL**)
  - OpenGL
- Unity: **ShaderLab**
  - It uses CG/HLSL

AN OVERVIEW OF HLSL

- Data types
  - **Standard**: int, bool, double, float,…
  - **Vector**: float4, float[], vector,…
  - **Matrices**: float3x3, float 2x2,…
  - **Textures**: sampler2D,..
- Instructions
  - for, if/else/, do/while,…
- Functions
  - cos, sin, mul, cross, dot,…

STRUCTURE OF SHADERLAB SHADERS

- Properties
  - Visible in the Unity's inspector!
- Subshaders (1 or more)
  - They contain 1 or more passes
  - Take care for optimization purposes!

    The execution depends on the hardware
- Fallbacks
  - Alternative when your GPU doesn't support a technique

THE SHADERLAB “EMPTY” CODE

Shader name (visible in dropmenu)

Properties (visible in the inspector)

First shader (if possible for the hardware)

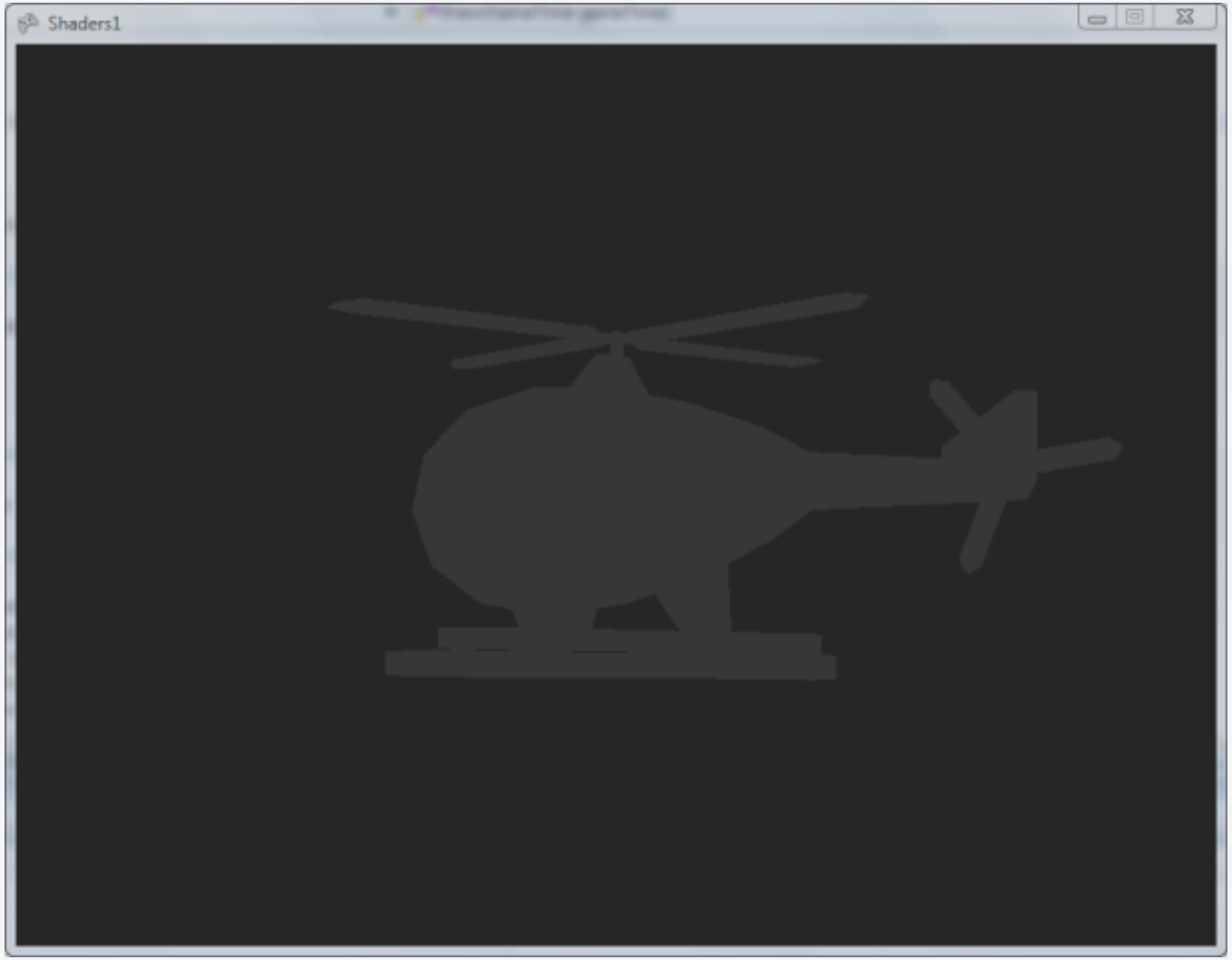
Alternative (if possible)

Fallback (last alternative)

```
Shader "Custom/AmbientLightShader" {

    Properties{
    }

    SubShader{
        Pass{
        }
        Pass{
        }
    }

    SubShader{
    }

    FallBack "Diffuse"
}
```

A SIMPLE EXAMPLE WITH UNITY

Create a shader for ambient lighting



AMBIENT LIGHTNING

- “Basic light in a dark room”
  - The “Hello World” of Shaders ;-)

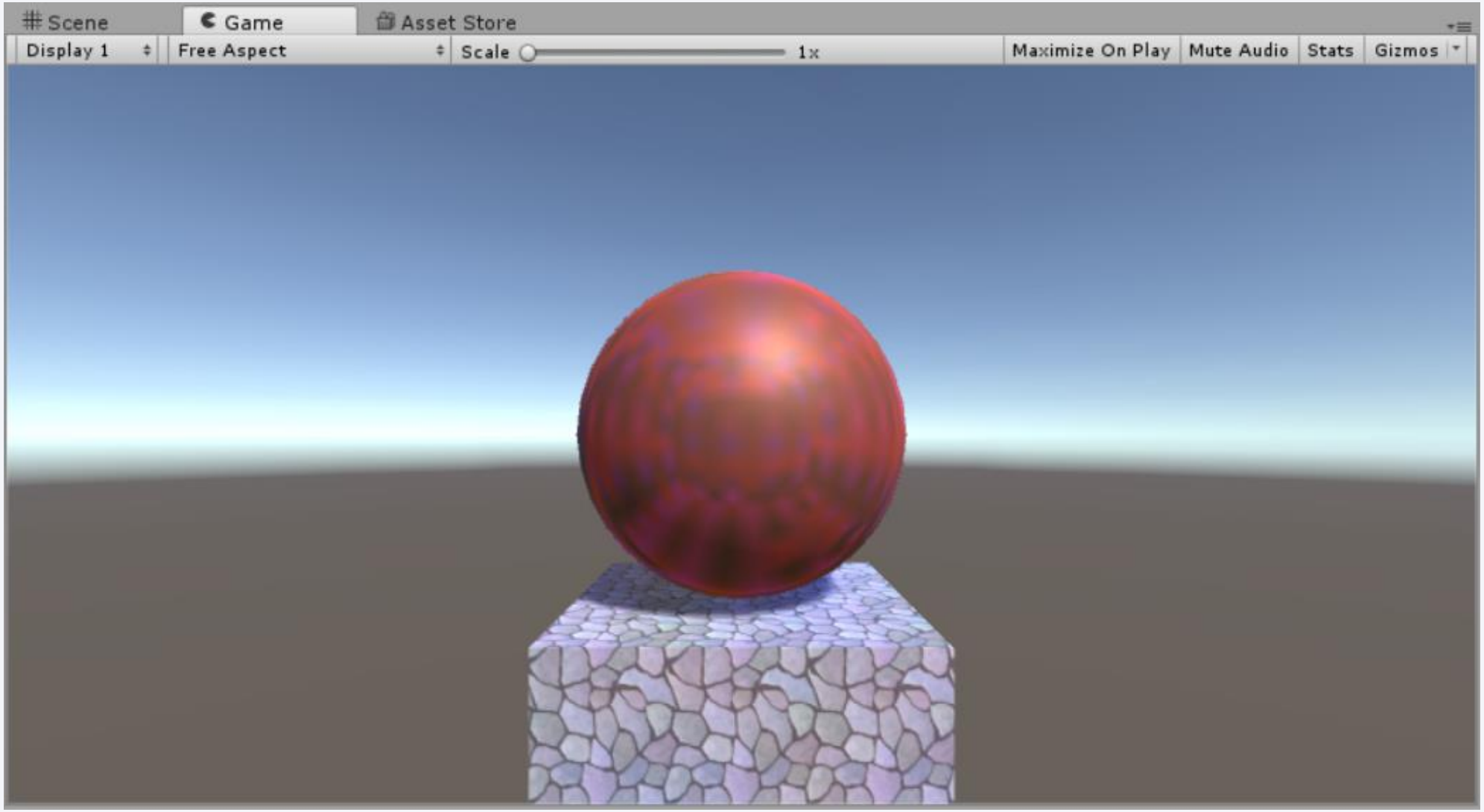
- AL = ALIntensity * ALColor

- Goals of the example
  1. Calculate the standard transform for vertices
  2. Calculate ambient light for the object

PREPARE YOUR SHADER

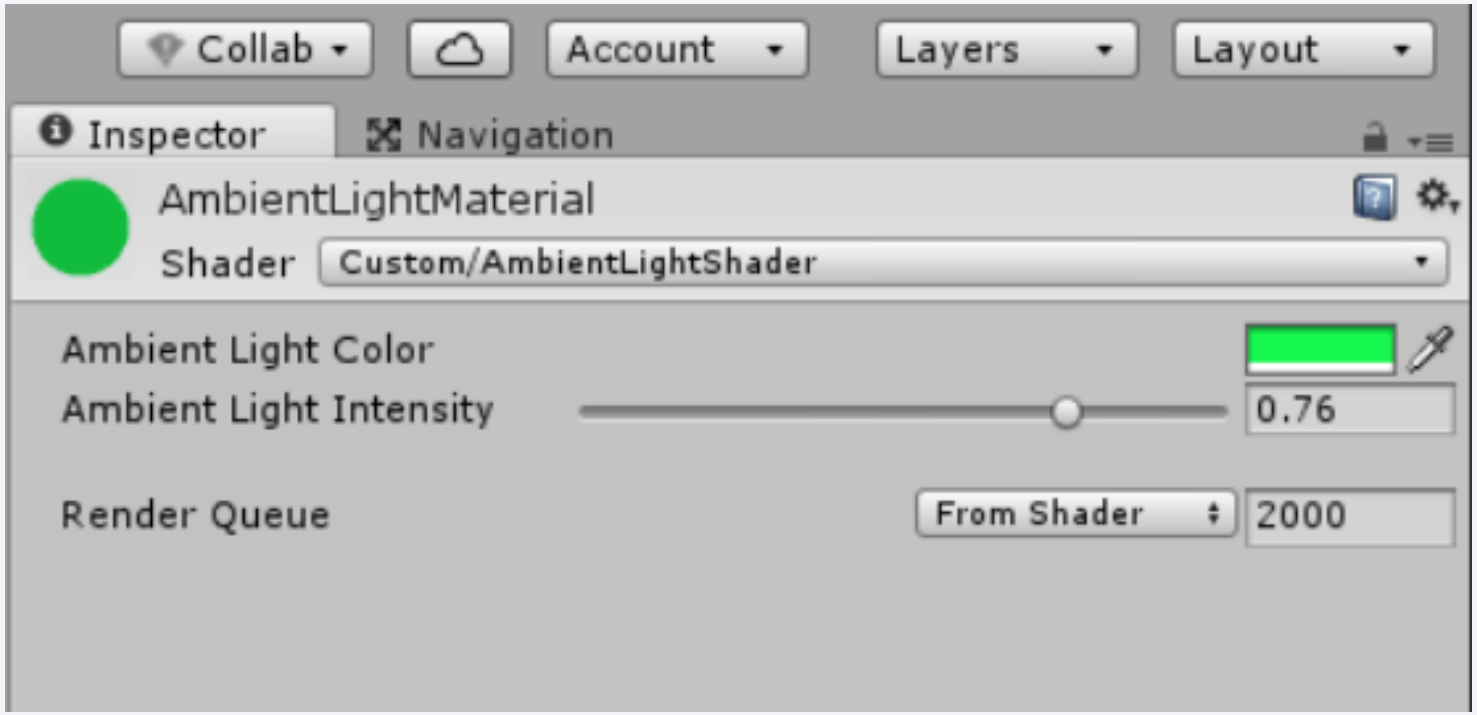
1. Create a 3D (or 2D) object
   - **Hierarchy**
2. Create a material
   - **Asset**
3. Create a shader
   - **Asset (e.g. the Standard Surface Shader)**
4. Assign the material to the 3D object
5. Assign the shader to the material
6. Clean and edit the code of your shader

EXAMPLE SCENE



DEFINE PROPERTIES

```
Properties{
   _AmbientLightColor ("Ambient Light Color",Color)=(1,1,1,1)
   _AmbientLightIntensity("Ambient Light Intensity",Range(0.0,1.0))=0.5
}
```



DEFINE #PRAGMA

```
SubShader {
   Pass{
        CGPROGRAM
        #pragma target 2.5
        #pragma vertex vertexShader
        #pragma fragment fragmentShader
        ENDCG
   }
}
```

- **target**: the target release of the hardware (max 5.0)
- **vertexShader**: declare the name of the vertex shader function
- **fragmentShader**: declare the name of the fragment (a.k.a. pixel) shader function

DEFINE VARIABLES

```
SubShader {
   Pass{
        CGPROGRAM
        [#pragma…]

        fixed4 _AmbientLightColor;
        float _AmbientLightIntensity;
        ENDCG
   }
}
```

- The variables containing the color components (RGBA values) and the intensity
- Properties are pointing at them

THE VERTEX SHADER FUNCTION

```
SubShader {
   Pass{
        CGPROGRAM
        #pragma vertex vertexShader
        [#pragma…]
        [#variables]

        float4 vertexShader(float4 v:POSITION) : SV_POSITION{
               return mul(UNITY_MATRIX_MVP, v);
        }
        ENDCG
   }
}
```

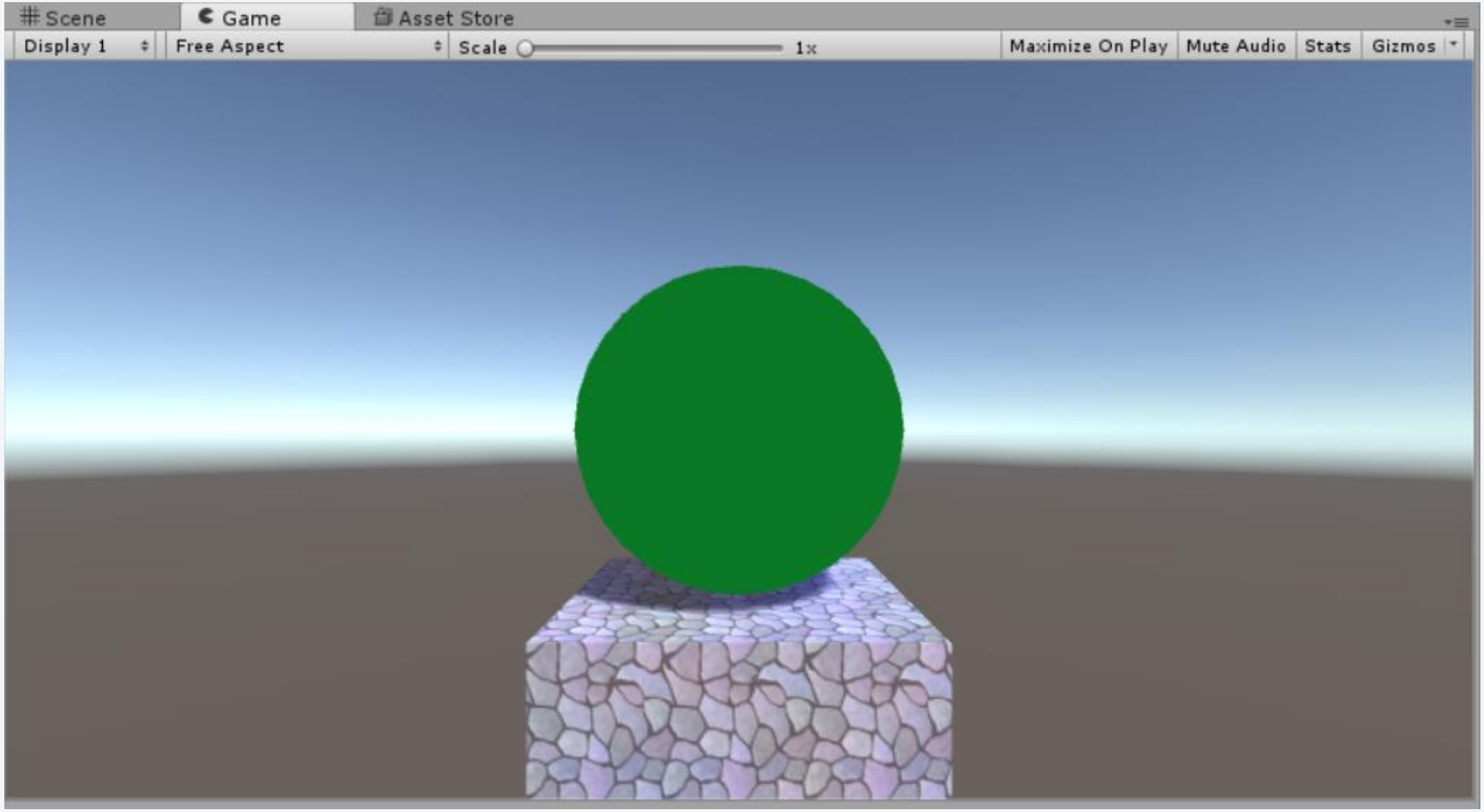
- UNITY_MATRIX_MVP are the model, view and perspective matrices
- The SV_POSITION output is the transformed position of the vertex
- SV_POSITION is a semantic in the programmable pipeline

THE PIXEL SHADER FUNCTION

```
SubShader {
   Pass{
        CGPROGRAM
        #pragma vertex fragmentShader
        [#pragma…]
        [#variables]

        fixed4 fragmentShader() : SV_Target{
               return _AmbientLightColor * _AmbientLightIntensity;
        }
        ENDCG
   }
}
```

- The SV_Target contains the RGBA value of the pixel after the ambient light calculation

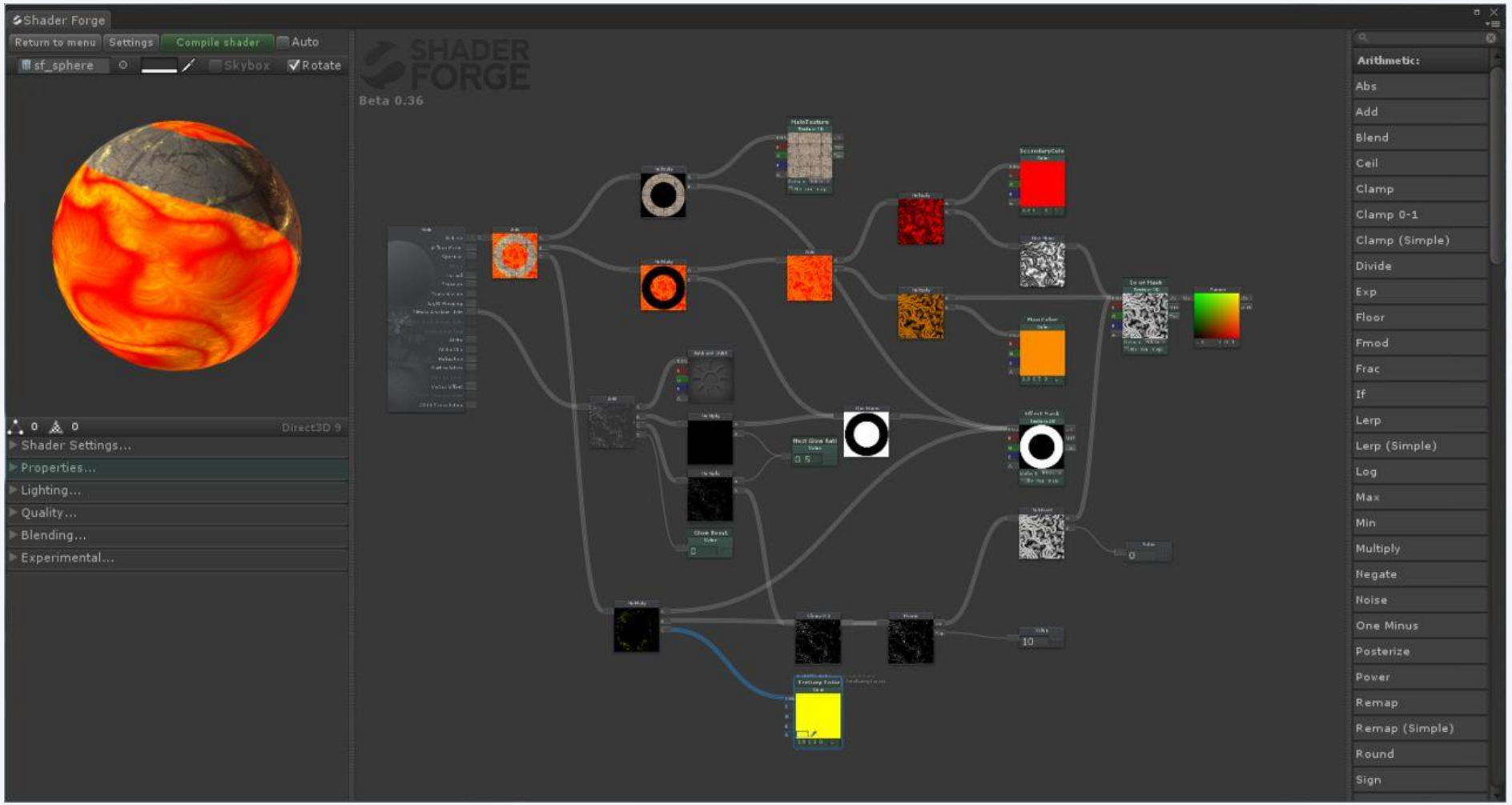
THE RESULT OF OUR SHADER



OTHERWISE…

- … it’s possible to use plugins with already programmed shaders.

THE SHADERFORGE PLUGIN



QUESTIONS?