游戏设计与开发

Shaders

上海交通大学软件学院
数字艺术实验室
Digital ART LAB

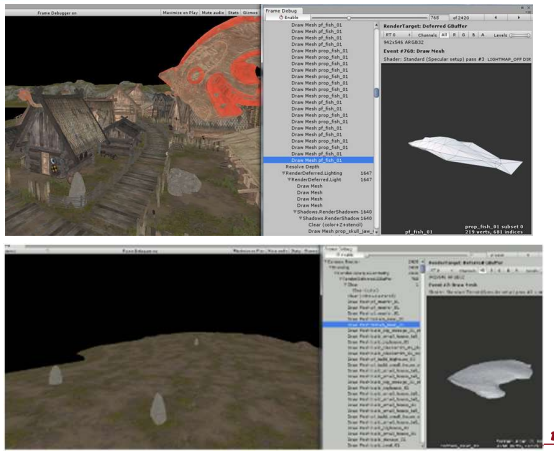
Outline

- Rendering pipelines & Shaders in Unity
- Some Shaders

Rendering Pipelines & Shaders in Unity

上海交通大学软件学院
数字艺术实验室
Digital ART LAB

Unity Frame Debugger



Digital ART Lab,SJTU

Rendering Pipelines in Unity

- **Built-in Render Pipelines**
- **Universal Render Pipelines (URP)**
- **High Definition Render Pipelines (HDRP)**

- **Scriptable Render Pipeline (SRP)**
  - **Allow to write C# scripts to fully control the rendering pipeline**
  - **Two prebuilt SRPs: URP, HDRP (offer extensive customization options)**
  - **You can also build your own custom SRP from scratch**

Built-in Render Pipeline

- Built-in Render Pipeline is a general-purpose render pipeline
- more limited custom extension than SRPs
- can choose between different rendering paths and extend functionality with command buffers and callbacks
- Rendering path
  - is a series of operations related to lighting and shading.
  - Different rendering paths have different capabilities and performance characteristics.
  - Decision depends on the project and the target hardware
  - *Forward Rendering* vs. *Deferred Shading*

Universal Render Pipeline (URP)

- The Universal Render Pipeline (URP) is a prebuilt Scriptable Render Pipeline, made by Unity. URP provides artist-friendly workflows that let you quickly and easily create optimized graphics across a range of platforms, from mobile to high-end consoles and PCs.

URP is supported on the following platforms:

- Windows and UWP
- Mac and iOS
- Android
- XBox One
- PlayStation4
- Nintendo Switch
- WebGL
- All current VR platforms

Note: Projects made using URP are not compatible with the High Definition Render Pipeline (HDRP) or the Built-in Render Pipeline. Before you start development, you must decide which render pipeline to use in your Project. For information on choosing a render pipeline, see the Render Pipelines section of the Unity Manual.

High Definition Render Pipeline (HDRP)

- The High Definition Render Pipeline (HDRP) is a high-fidelity Scriptable Render Pipeline built by Unity to target modern (Compute Shader compatible) platforms.
- HDRP utilizes **Physically-Based Lighting** techniques, linear lighting, HDR lighting, and a configurable hybrid Tile/Cluster deferred/Forward lighting architecture.
- It gives you the tools you need to create applications such as games, technical demos, and animations to a high graphical standard.

HRDP is only supported on the following platforms:

- Windows and Windows Store, with DirectX 11 or DirectX 12 and Shader Model 5.0
- Modern consoles (Sony PS4 and Microsoft Xbox One)
- MacOS using Metal graphics
- Linux and Windows platforms with Vulkan

HDRP does not support OpenGL or OpenGL ES devices.

Note: HDRP only works on these platforms if the device used supports Compute Shaders.

Meshes, Materials, Shaders and Textures

- **Meshes:** the main graphics primitive, define shapes
- **Materials:** define how a surface should be rendered, including refs to Textures, Tiling, Color tints and etc. Options for Material: one specific **Shader** to be used
- **Shaders:** small scripts that contain the math and algorithms for calculating the Color of each pixel, based on lighting input and **Material configuration**
- **Textures:** bitmap images. A Material can contain refers to one or more textures; the Material's Shader can use the textures to calculate surface color of a GameObject
  - basic Color (Albedo) using texture
  - reflectivity or roughness using texture

Standard Shader

- **A built-in shader with a comprehensive set of features**
  - **Hard surface: stone, wood, glass, plastic and metal (even decent for non-hard materials: skin, hair, cloth)**
  - **supports a wide range of shader types and combinations (as Diffuse, Specular, Bumped Specular, Reflective)**
  - **Its features are enabled/disabled by using/not using the various texture slots and parameters in the material editor**
- **Physically Based Shading (PBS)**
  - **principles of physics in shading**
  - **energy conserve, Fresnel reflections, self occlusion, HDR**
  - **the Disney model for diffuse component**
  - **GGX model for specular**
  - **Smith Joint GGX visibility term**
  - **Schlick Fresnel appromixation.**

Writing Shaders in Unity

- **Three different ways**
  - **Surface Shaders *(for lights and shadows; higher level of abstraction for interaction with Unity's lighting pipeline; Cg/HLSL + autogenerated code; do not use if no lighting)***
  - **Vertex and Fragment Shaders *(the most flexible; but more code, harder to interact with lighting; using* Cg/HLSL)**
  - **Fixed Function Shaders *(legacy Shader syntax, written in ShaderLab language)***
- **All wrapped in ShaderLab**
- **All internally converted to vertex and fragment shaders**

ShaderLab

- **ShaderLab: a declarative language**

  Syntax

```
Shader "name" { [Properties] Subshaders [Fallback] [CustomEditor] }
```

- **Example**
- **Shader**
  - properties
  - SubShader
  - pass

```
// colored vertex lighting
Shader "Simple colored lighting"
{
    // a single color property
    Properties {
        _Color ("Main Color", Color) = (1,.5,.5,1)
    }
    // define one subshader
    SubShader
    {
        // a single pass in our subshader
        Pass
        {
            // use fixed function per-vertex lighting
            Material
            {
                Diffuse [_Color]
            }
            Lighting On
        }
    }
}
```

Some Shaders

上海交通大学软件学院
数字艺术实验室
Digital ART LAB

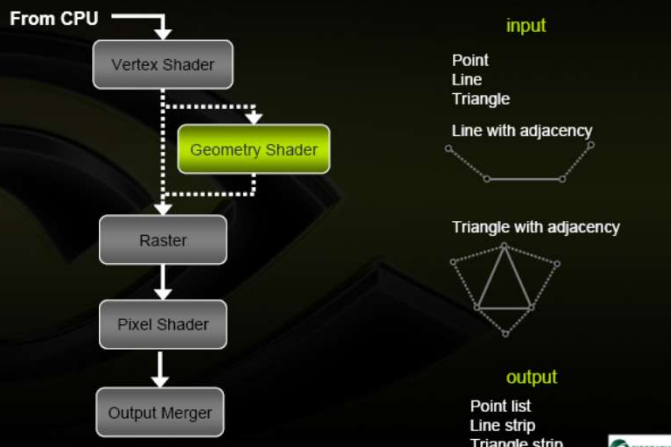
Example Shaders

Assassin's Creed Odyssey

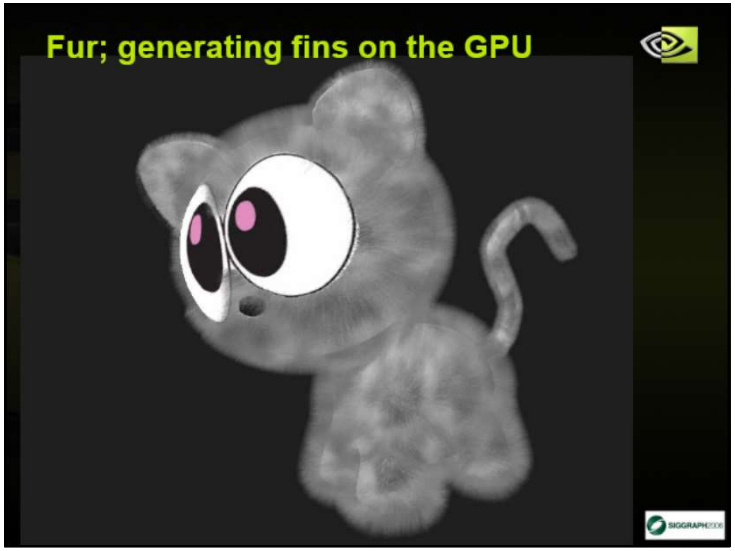
The Legend of Zelda: Breath of the Wild

Digital ART Lab, SJTU

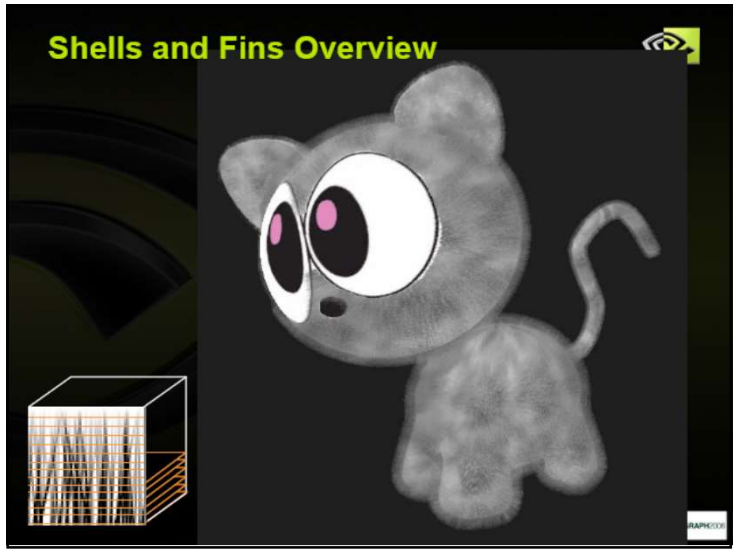
Geometry Shader

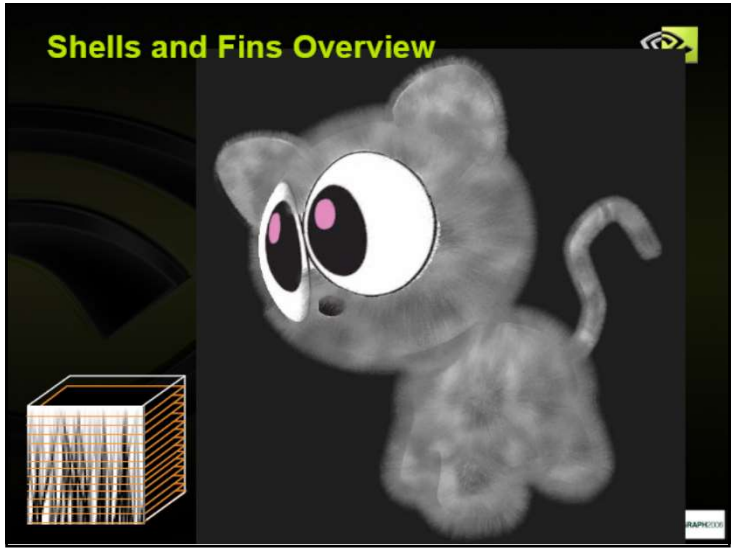


Fur; generating fins on the GPU



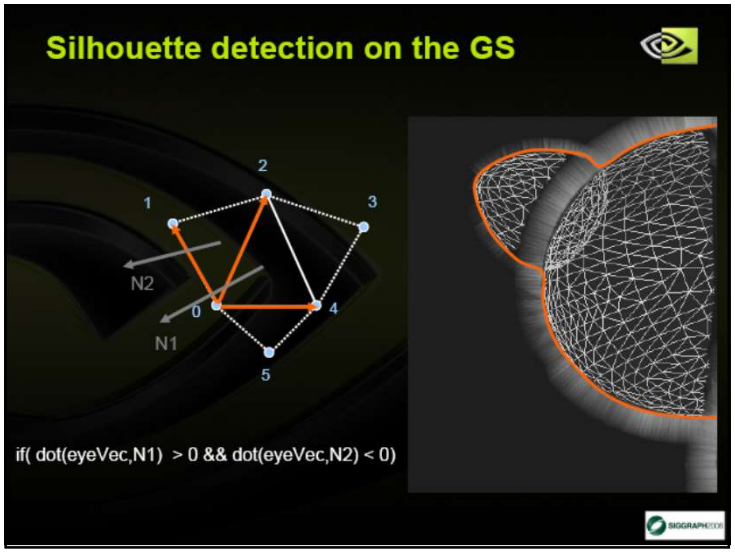

Shells and Fins Overview

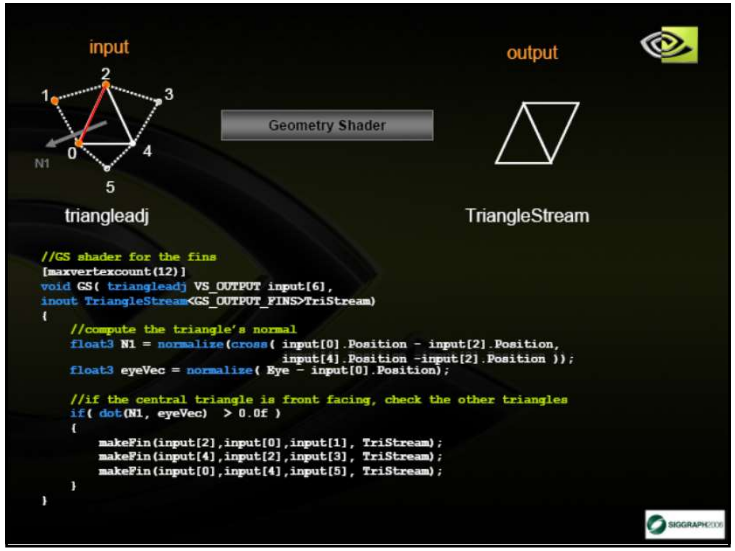

Shells and Fins Overview

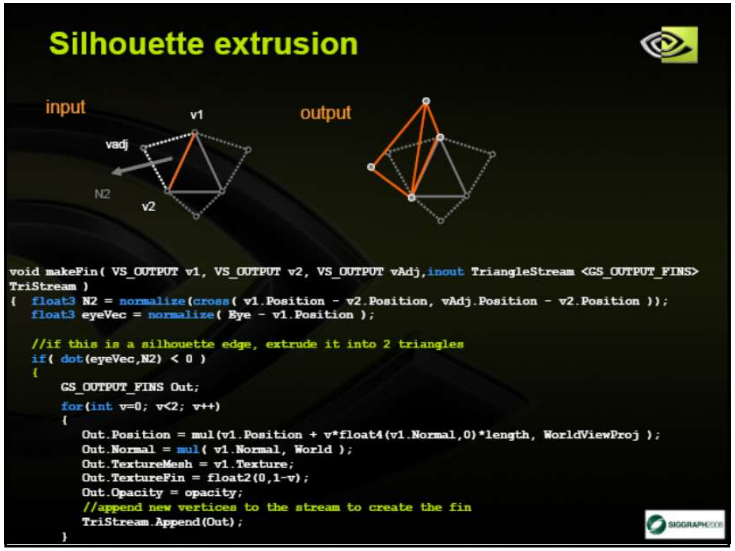

Shells and Fins Overview


Shells: problems at silhouettes
8 shells
+
fins


fins


Silhouette detection on the GS
1
2
3
4
5
0
N1
N2
if(dot(eyeVec,N1) > 0 && dot(eyeVec,N2) < 0)


input
output
Geometry Shader
triangleadj
TriangleStream
//GS shader for the fins
[maxvertexcount(12)]
void GS(triangleadj VS_OUTPUT input[6],
inout TriangleStream<GS_OUTPUT_FINS>TriStream)
{
//compute the triangle's normal
float3 N1 = normalize(cross(input[0].Position - input[2].Position,
input[4].Position -input[2].Position));
float3 eyeVec = normalize(Eye - input[0].Position);
//if the central triangle is front facing, check the other triangles
if(dot(N1, eyeVec) > 0.0f)
{
makePin(input[2],input[0],input[1], TriStream);
makePin(input[4],input[2],input[3], TriStream);
makePin(input[0],input[4],input[5], TriStream);
}
}


Silhouette extrusion
input
output
v1
vadj
v2
N2
void makePin(VS_OUTPUT v1, VS_OUTPUT v2, VS_OUTPUT vAdj,inout TriangleStream <GS_OUTPUT_FINS>
TriStream)
{ float3 N2 = normalize(cross(v1.Position - v2.Position, vAdj.Position - v2.Position));
float3 eyeVec = normalize(Eye - v1.Position);
//if this is a silhouette edge, extrude it into 2 triangles
if(dot(eyeVec,N2) < 0)
{
GS_OUTPUT_FINS Out;
for(int v=0; v<2; v++)
{
Out.Position = mul(v1.Position + v*float4(v1.Normal,0)*length, WorldViewProj);
Out.Normal = mul(v1.Normal, World);
Out.TextureMesh = v1.Texture;
Out.TextureFin = float2(0,1-v);
Out.Opacity = opacity;
//append new vertices to the stream to create the fin
TriStream.Append(Out);
}

Silhouette extrusion


```
for(int v=0; v<2; v++)
{
   Out.Position = mul(v2.Position + v*float4(v2.Normal,0)*length, WorldViewProj );
   Out.Normal = mul( v2.Normal, World );
   Out.TextureMesh = v2.Texture;
   Out.TexturePin = float2(1,1-v);
   Out.Opacity = opacity;

   TriStream.Append(Out);
}

TriStream.RestartStrip();
}}
```

Some more Geometry Shader applications

- Silhouette detection and extrusion for:
  - Shadow volume generation
  - NPR
- Access to topology for calculating things like curvature
- Render to cube map in single pass
  - In conjunction with Render Target arrays
- GPGPU
  - enables variable number of outputs from shader

Cel Shader with Outline（描边）

- Cel – Shaded Lighting
- $dot\ (\vec{N},\ \vec{L})$ → the ramp texture (2d, only two colors)

```
// convert light direction to world space & normalize
// _WorldSpaceLightPos0 provided by Unity
float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);

// finds location on ramp texture that we should sample
// based on angle between surface normal and light direction
float ramp = clamp(dot(input.normal, lightDir), 0, 1.0);
float3 lighting = tex2D(_RampTex, float2(ramp, 0.5)).rgb;

// sample main texture for color
// for ex, in the image above, the main texture has the dog's
// black and white color and eyes
float4 albedo = tex2D(_MainTex, input.texCoord.xy);

// get final color
float3 rgb = albedo.rgb * lighting * _Color.rgb;
return float4(rgb, 1.0);
```

Ref: https://lindenreid.wordpress.com/2017/12/19/cel-shader-with-outline-in-unity/

Digital ART Lab,SJTU

Cel Shader with Outline（描边）

- Outline Effect
  - —Z-buffer + stencil buffer
  - —drawing a scaled version (along the vertex normal) of the original mesh *after* the original mesh
  - —using the stencil buffer to not draw the outline where the original has already been drawn.

Digital ART Lab,SJTU

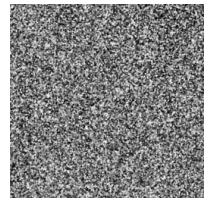
Simple Water

- Foam Line
  - Reading depth at every vertex
  - Transform depth into distance from water surface
  - Using the value as a gradient

Ref: https://lindenreid.wordpress.com/2017/12/15/simple-water-shader-in-unity/

Simple Water with Waves

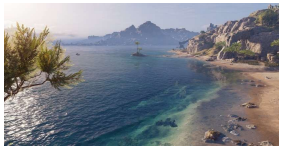
- Sample a noise texture (random value)
- waveValue = Sin(time * noise_value)
- Vertex position += waveValue



- Vertex Shader

Ocean Simulation

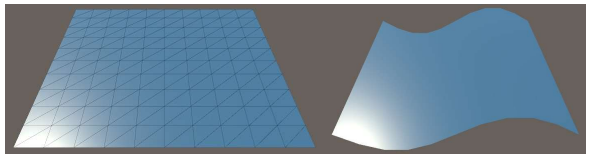
- Gerstner wave
- FFT wave
- Flow Map
- Bump Mapping



- Wave Particle
- Physically-based method
- …

Ocean Simulation

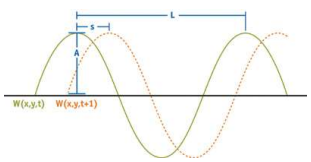
- Model the ocean using a mesh

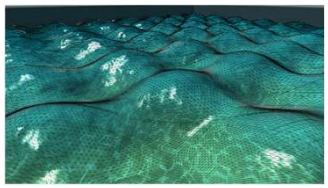


Ocean Simulation

- Model the ocean using a mesh
- Model the wave using wave function
  - Vertices move according to a sine function

$$h(x,z,t) = -y_0 + \sum_{i=1}^{N_w} A_i cos(k_{i_x} x + k_{i_z} z - w_i t)$$



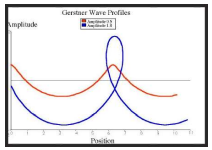


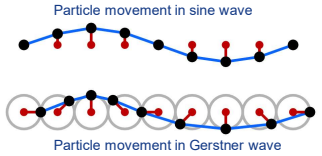
Ocean Simulation

- Model the ocean using a mesh
- Model the wave using wave function
- Model the "peak" effect using Gerstner wave
  - Gerstner wave = solution of periodic surface gravity waves
  -

$$x = x_0 - \sum_{i=1}^{N} (\vec{k}_i / k_i) A_i sin(\vec{k}_i \cdot x_0 - w_i t + \phi_i)$$

$$y = \sum_{i=1}^{N} A_i cos(\vec{k}_i \cdot x_0 - w_i t + \phi_i)$$



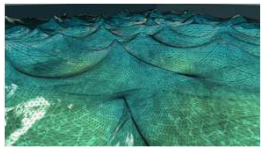


Ocean Simulation

- Model the ocean using a mesh
- Model the wave using wave function
- Model the "peak" effect using Gerstner wave
  - Gerstner wave = solution of periodic surface gravity waves
  -

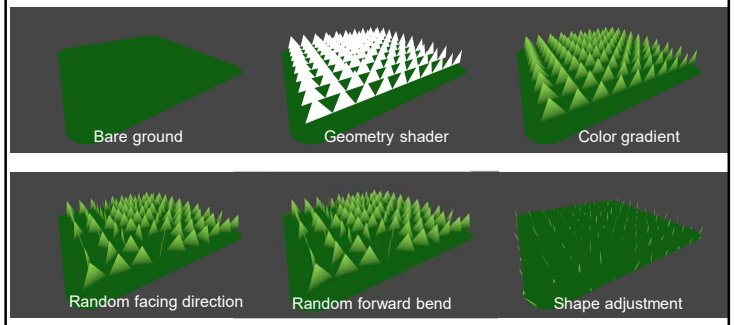
$$x = x_0 - \sum_{i=1}^{N} (\vec{k}_i / k_i) A_i sin(\vec{k}_i \cdot x_0 - w_i t + \phi_i)$$

$$y = \sum_{i=1}^{N} A_i cos(\vec{k}_i \cdot x_0 - w_i t + \phi_i)$$

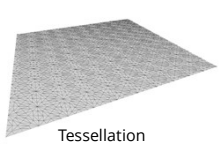


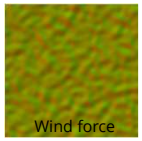
Grass Simulation

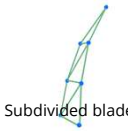
- Grass

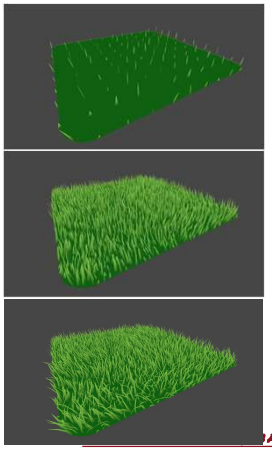


Grass Simulation



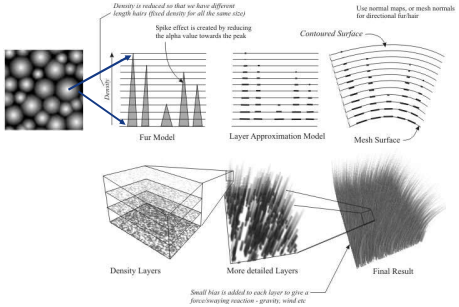






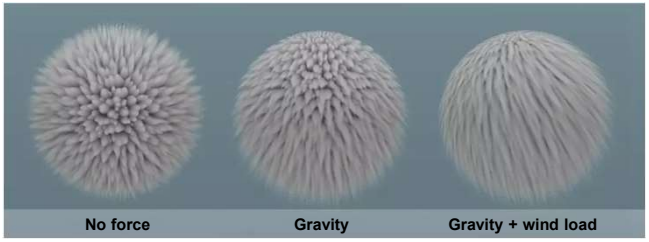
Fur Simulation

- Model
  - Model fur using layer approximation model
  - Render one layer in each pass



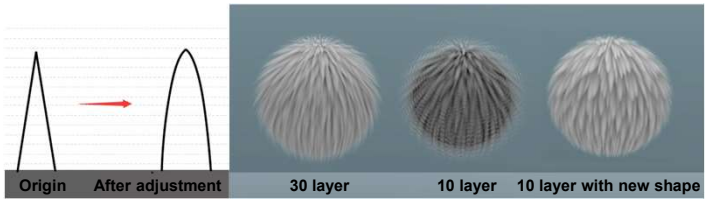
Fur Simulation

- Model
  - Model fur using layer approximation model
  - Render one layer in each pass
  - Apply gravity and wind force



Fur Simulation

- Model
  - Model fur using layer approximation model
  - Render one layer in each pass
  - Apply gravity and wind force
  - Adjust the fur shape

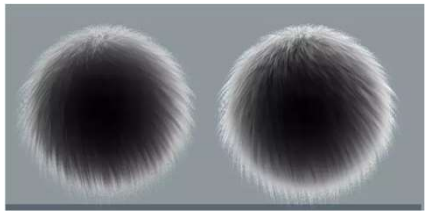


Fur Simulation

- Lighting
  - Ambient occlusion

Fur Simulation

- Lighting
  - Ambient occlusion
  - Rim light



Fresnel Fresnel + Occlusion

Fur Simulation

- Lighting
  - Ambient occlusion
  - Rim light
  - Sun light

Sun light at rim

With additional rim light

Fur Simulation

- Final result