Art styles rendering techniques survey
and making a tool to switch art styles in

Unity 3D

Ge Zhu
Northeastern University
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>4</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>5</td>
</tr>
<tr>
<td>2 BACKGROUND</td>
<td>8</td>
</tr>
<tr>
<td>3 CEL SHADING SURVEY</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Outline</td>
<td>10</td>
</tr>
<tr>
<td>3.1.1 Back-face Method</td>
<td>11</td>
</tr>
<tr>
<td>3.1.2 ZWrite Off Method</td>
<td>11</td>
</tr>
<tr>
<td>3.1.3 Comparison</td>
<td>11</td>
</tr>
<tr>
<td>3.2 Diffuse Ramp</td>
<td>12</td>
</tr>
<tr>
<td>3.2.1 Introduction</td>
<td>12</td>
</tr>
<tr>
<td>3.2.2 Implementation</td>
<td>12</td>
</tr>
<tr>
<td>3.2.3 Halflambert</td>
<td>13</td>
</tr>
<tr>
<td>3.2.4 Diffuse Enhancement</td>
<td>14</td>
</tr>
<tr>
<td>3.3 Rim Light</td>
<td>15</td>
</tr>
<tr>
<td>3.3.1 Introduction</td>
<td>15</td>
</tr>
<tr>
<td>3.3.2 Implementation principle</td>
<td>16</td>
</tr>
<tr>
<td>3.3.3 Improvement</td>
<td>17</td>
</tr>
<tr>
<td>3.3.4 Disable rim light in shadow</td>
<td>18</td>
</tr>
<tr>
<td>3.3.5 Threshold</td>
<td>19</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.3.6 Moving Problem</td>
<td>20</td>
</tr>
<tr>
<td>3.3.7 Other Thoughts</td>
<td>21</td>
</tr>
<tr>
<td><strong>4 LOW-POLY SURVEY</strong></td>
<td>22</td>
</tr>
<tr>
<td>4.1 Flat Shading</td>
<td></td>
</tr>
<tr>
<td>4.1.1 Rewrite Mesh vertices</td>
<td>23</td>
</tr>
<tr>
<td>4.1.2 Use geometry Shader</td>
<td>24</td>
</tr>
<tr>
<td>4.1.3 Use derivatives</td>
<td>24</td>
</tr>
<tr>
<td>4.2 Mesh Simplification</td>
<td></td>
</tr>
<tr>
<td>4.2.1 Introduction</td>
<td>24</td>
</tr>
<tr>
<td>4.2.2 Skinned Mesh</td>
<td>26</td>
</tr>
<tr>
<td>4.2.3 Static Object</td>
<td>26</td>
</tr>
<tr>
<td>4.3 In-game implementation</td>
<td>26</td>
</tr>
<tr>
<td><strong>5 PIXEL ART</strong></td>
<td>27</td>
</tr>
<tr>
<td>5.1 Reduce Color Range</td>
<td>27</td>
</tr>
<tr>
<td>5.2 Reduce Resolution</td>
<td>27</td>
</tr>
<tr>
<td><strong>6 TOOL DESIGN AND USAGE</strong></td>
<td>29</td>
</tr>
<tr>
<td><strong>7 RESULT AND FUTURE WORK</strong></td>
<td>30</td>
</tr>
<tr>
<td><strong>REFERENCES</strong></td>
<td>32</td>
</tr>
<tr>
<td><strong>APPENDIX</strong></td>
<td>34</td>
</tr>
<tr>
<td>Survey Result</td>
<td>34</td>
</tr>
<tr>
<td>Code</td>
<td>35</td>
</tr>
</tbody>
</table>
ABSTRACT

Unique art style can help a game stand out. Many indie games choose to pick a non-photorealistic rendering (NPR) style instead of going for photorealistic rendering. Based on the survey we did, many game developers would like to have a tool that can generate and switch between different art styles for the game to help them in the prototype stage. This thesis discusses three popular art styles: cartoon, low-poly, and pixel about their theories and possible implementation in Unity 3D game. Our contribution in this project is a tool to select between different art styles for developers. In the end, we successfully implemented cartoon and low-poly style in our tool, we could not however implement pixel art style in our tool and the reason was discussed.
1 INTRODUCTION

Every game has an art style. While many AAA games tend to apply photorealistic style, small indie games share more variety in art styles, for example Cuphead [1] has cartoon style, Dead Cells [2] has pixel art style, Poly Bridge [3] has low-poly style. Nowadays, many games use different art styles to show their uniqueness and create special atmosphere.

Figure 1: Cuphead [1] with cartoon style

Figure 2: Dead Cells [2] with pixel style
While it is exciting that more and more art styles are being explored by game designers and artists, it also becomes harder for game developers to choose the best art style of their games in the very beginning of the development. They need to find the best art style through the process of prototyping. And as the result of a survey we did shows (see Appendix), most game developers think that unique art style is a way to help a game stand out.

One way is to switch between art style is by changing rendering method. There exist some tools to generate certain art style in this way, but there are few tools yet that can apply multiple art styles and enable switching among them.

This project aims to develop a rendering solution in Unity 3D that can allow game developers to switch between different art styles. The goal of this tool is to make prototyping process faster and more efficient. Game developers only need to build some simple model and put them in Unity, and then after adding our tools, they can start tweak around and test different art style. To evaluate the result, a game is made using the tool developed in this project. The game contains multiple levels, each level is rendered in a unique art style. The evaluation is based on the feedback of players.
We surveyed 13 game developers to see their opinions about this tool before we started. They rated overall 8 out of 10 for desiring this tool, which suggests a need for this type of tool. When answering what art styles they want, 44% mentioned cartoon/cel shading. The response is linked in the Appendix.

The art styles this project aim to cover are: cartoon, pixel, and low-poly. These three styles are commonly used by games nowadays and also look very different either with the realistic style or within themselves. This project is implemented in Unity 3D game engine with Unity Shader system. Unity 3D is a very popular game engine with overall 3 billion devices using it currently [4]. Unity3D has demonstrated the ability to create unique art styles for indie games like Inside [5], Cuphead [1], Ori and the Blind Forest [6]. Unity also has a very powerful Shader system. Our project relies on the programmable shader pipeline. Unity allows us to implement High level shading language (HLSL) shader code with the game and making connection between game codes and shader codes. Based on the above criteria, we choose Unity 3D as the engine of our game.
2 BACKGROUND

We start first with researching about the games that are famous for their art styles. The game Borderlands is famous for its cartoon style, which is done by cel-shading. But this game’s originally developed without this style. In his GDC talk, Randy Pitchford showed how Borderlands developers adapted different rendering techniques to achieve the unique visual effect of this game [7]. As the comparison images below show, the game is much stylized and memorable after adding the art style.

Figure 4: Borderlands without cel-shading(top) and with cel-shading(bottom)
Besides Borderlands, we also looked into other games with unique art style for references, for examples Skullgirl [8] and Dragon Ball Fighter Z [9]. Then we started to do research about the implementation of the three art styles we chose. Since our tool is for game developers to use in a game engine, 3D modeling methods or other artist’s method are not considered.

![Skullgirl gameplay image](image1)

**Figure 5: Skullgirls [8] gameplay image**

![Dragon Ball Fighter Z gameplay image](image2)

**Figure 6: Dragon Ball Fighter Z [9] gameplay image**

Moller in his book *Real-time Rendering Third Edition* provided the basic knowledge for understanding real-time rendering and NPR [10]. It introduced the rasterization rendering pipeline that are used in many modern game engine including Unity 3D. In this pipeline the vertex shader and fragment shader is
programmable, and the majority of our implementation was done within these two parts. We also used the configurable option to control the fixed parts of the pipeline. An example of this is disabling writing to depth buffer when drawing outlines.

Mitchell provided a very general guide for cartoon style rendering (cel shading) [11]. It mentioned many rendering technique of cel shading without going into very detail. We implemented and expended many theories from this article. Some techniques however, require works from artists, like normal simplification, meaning adjust the normal of the object mesh to reduce glitching surface shadow. These techniques are important to cel shading but they require lots of artist’s work and can only be applied for that certain model. Our tool only focused on the techniques that are implemented in shader and can be applied to general models.
3 CEL SHADING SURVEY

Cel Shading is the fundamental technique to implement cartoon art style. It is the most popular NPR technique and has strong connections with fantasy and childhood [11]. Although cel-shading is a very mature topic, there is not a standard way to implement it. It can be seen as a combination of a serial of computer graphics techniques. This part went through the very core techniques of cel shading and implemented them in our tool.

3.1 Outline

Outline, also called silhouette, is a very important part in cel shading. Outlines are often used in black color on objects with bright color to emphasize the shape of those objects. As figure 7 shows, in the game Skullgirls Mobile [12], the outlines of a character not only surround the whole shape of the character, but also some inside parts like fingers. With the help of outlines player can see tell the character and the background more easily.

Figure 7: The usage of outline in Skullgirls Mobile [16]
According to Wang, the outline drawing methods, can be categorized into image-based methods and object-based methods [13]. Image based methods draw outlines on the frame buffer after the frame is rendered using the edge detection method. In this way it is fast but also hard to control. While the image-based have been implemented in games like Brawl Star, it is not the focus of this paper. The reason is that those methods are implemented using post processing shaders, but our tool only focus object material shaders. In the object-based methods, there are two typical methods that will be discussed below.

### 3.1.1 Back-face Method

The back-face method means that render the object a second time in an additional pass, in the second pass, the object is rendered in black color, slightly larger than the origin object, and only the back faces of the object is rendered.

### 3.1.2 ZWrite Off Method

This method is almost identical to the back-face method, except that it doesn’t render the back face of the additional object. Instead, it rendered the second object behind the origin one.
3.1.3 Comparison

Figure 8: Comparison among no outline (top), Back-face outline (middle), and ZWrite Off Outline (bottom)
As shown in the figure above, the back-face method can present moe details within the shape of the object, but at the same time, some extra detail looks weird like the outline around the character’s eyes. The ZWrite Off method only draw the outline of the shape, so there is no weird outlines, but it also loses some detail.

In practice, it is a trade-off to select which method. Most games like Skullgirl choose the first one because glitching can be solved by modifying the 3D model or adding additional mask, or it is just not obvious when the object is far away. However, the extra details inside the shape do help increase the look of this effect.

3.2 Diffuse Ramp

3.2.1 Introduction

Diffuse Ramp is used in a technology called ramp shading, it is a very important part in cel shading. Almost all the implementation of cel shading are involved with diffuse ramp. Diffuse ramp is a technique that can make the light-dark transaction of the object appears sharper.

3.2.2 Implementation

In reality or photorealistic games, the transaction from light to dark is very smooth. The reason is that in reality, light scatter around at the surface of an object. In photorealistic rendering, this effect is simulated by a term called diffuse light. In the most commonly used Phong Illumination Model, this term is determined by the dot product of surface normal and light direction.
Figure 9: Comparison between without diffuse ramping (top), and with diffuse ramping (bottom)

Cel Shading comes from hand-draw cartoon. Cartoon artists usually don’t draw complex light-dark transaction. Instead, they simply divided the object into several parts and assign a light level for each part. In this way the object turns to have very sharp light-dark transaction. To achieve this effect, we add a ramp texture on the diffuse light calculated by the Phong Illumination Model. After the diffuse value is calculated, we search the ramp texture based on this value to get a new value and use the new value as the real diffuse light.
3.2.3 Halflambert

According to [11], the origin diffuse light equation has several faults such as all the area in shadow remain same level of brightness, which is pure black. An alternative solution is suggested as "Half Lambert":

We also implemented this method in our tool.

Figure 10: Comparison between lambert model (top) and halflambert model (bottom)
3.2.4 Diffuse Enhancement

In real game implementation, it is very hard to get satisfying cartoon look from dark objects. One reason is that the black outlines can barely be seen on dark objects, the other reason is that diffuse values also sit in a very narrow region of the ramp map. To solve this problem, we added a diffuse enhance feature to enhance the diffuse color of dark objects in game.

Figure 11: Comparison between not using diffuse enhancement (top) and using diffuse enhancement (bottom)
3.3 Rim Light

3.3.1 Introduction

Rim light is another cel shading technique that usually can render an effect of bright contour around the object. Roystan defines rim light as "the addition of illumination to the edges of an object to simulate reflected light or backlighting." [14]. Comparing to outline, it is usually brighter, and it is within the object rather than outside of it.

![Outline vs Rim Light](image)

**Figure 12:** The different usage of outline and rim light in Skullgirls Mobile [16]

The concept of rim light comes from a technique called "backlight" in photography. Udi Tirosh illustrated the concept of backlight in his article [15]. Backlight is a light comes from the back of the object to create a contour and separation so it is also referred as "rim light", which is commonly used in cel shading area.
3.3.2 Implementation principle

Although in theory, the rim light is created by putting a light in the back of an object, this method is resource costly and hard to implement in games. Extra real-time lighting for each single object will significantly drop the frame rate. There is another way to achieve similar effect with computer graphics method. The theory is that rim light appears when the view direction is close to be perpendicular to the surface normal.
Figure 13: Comparison between without rim light (top), and with rim light (middle), and their difference image (bottom)
3.3.3 Improvement

Not all the rim light calculated with the formula need to be included in the final image. [11] provide an industrial implementation which uses an artist-painted mask to further filter the rim light. But as the principle of our tool is to provide a general solution to implement special art styles, it is impractical to expect a rim light mask included with every general models. Our tool provides a few alternation for the default rim light calculated to achieve better rim light effect without using a mask.

3.3.4 Disable rim light in shadow

In practice, it turns out that sometimes the rim light in the dark side (not hit by light) of the object will create some strange effect. This effect may make the object looks over-bloomed when the background is bright. So our tool provides an option to turn off rim light in the shadow area. We still keep the option of rendering rim light in the dark side of the object because in game when the background is dark, rim light can help showing player the contour of the game objects.
Figure 14: Comparison between not disabling rim light in shadow (top) and disabling rim light in shadow (middle), and their difference image (bottom)
3.3.5 Threshold

We made another improvement to the basic formula. We add a threshold to the rim light value. If the value is smaller than the threshold, then ignore the rim light value. This can create a sharper transaction of the rim light area and better visual effect.
3.3.6 Moving Problem

Normally the implementation of view direction is done by using the Unity 3D build-in shader variable \_WorldSpaceCameraPos minus the object’s world space position. This method works well when the object is standing still, but may lead to a more severe problem when the object is moving. When the object is moving, the rim light area changes.
We think that the cause of this problem is the difference in update time of the game and the shader. As the game running, the shader can't take in the proper \_WorldSpaceCameraPos variable. Consider this game is a top-down game which camera angle is fixed, we made a little customization that passing the camera viewing direction from script to the shader to replace \_WorldSpaceCameraPos. The reason we could do this is because that the game camera is orthogonal so the viewing direction is always fixed. And this solved the moving problem.
3.3.7 Other Thoughts

During developing the game, we found that rim light is better on objects with complex mesh than objects with simple meshes. The reason is that the calculation of rim light involves normals. Simple meshes tend to have large faces, which means that all the pixels on this face have same normal value, this may lead to the result of the whole face has rim light, which is not a good effect.
4 LOW-POLY SURVEY

Low-poly is a very widely defined term. Any 3D model with few polygons can be called low-poly. But polygon counts also differ on differ low-poly models. Figure 17 and Figure 18 show two low-poly art works, The Low Ploy Jack Nicholson has much more polygons than Capa Roja. But they also share one similarity: you can see the flat face of those polygons.

Figure 17: Low Poly Jack Nicholson by Paul Douard
Figure 18: Capa Roja by CAZAPAPELES

If a game can be called low-poly depends on the 3D model assets provided. But still there is a technique that can make a 3D model with high polygon counts looks like low-poly by rendering hard-edge instead of soft-edge. This technique is called flat shading. 4.1 will discuss in detail several common methods for doing flat shading. 4.2 will discuss about another technique called mesh simplification that can be used together with flat shading to get better low-poly effect.

4.1 Flat Shading

As images shown below, normal 3D modeling tools will smoothen the mesh before exporting. This can make the model look more natural and also reduce vertex count of the model (by only storing one interpolated vertex information for all triangles sharing it). But to create low-poly effect, we want to reverse this smoothing effect, and this is what flat shading means. The following parts will discuss several ways to do flat shading. Notice that unlike the methods compared in cel shading part, these three methods generate exactly the same visual result, the only difference will be about implementation and efficiency.
4.1.1 Rewrite Mesh vertices

This method rewrites the origin mesh. In the origin mesh, two neighbor triangles share a same vertex, this vertex's normal is the mix of two triangles' normals. So the 3 corner vertices of a triangle have different value. When rendering, each point's normal is interpolated among the three corner vertices' normals and the triangle looks smooth. By rewriting mesh vertices, each triangle’s three corner vertices all have same normal value, after interpolation, every point on the triangle have the same normal value and the triangle looks flat. So if a vertex is shared by 5 triangles like shown in Figure 19, there will be 5 different vertices (which are different in vertex normal) overlapping in the same position, one for each triangle.

This method will dramatically increase the number of vertices of the mesh, leading to increase the mesh data size, which requires more memory to store. This method also require extra code besides shader code, because rewriting mesh data can only be done outside shader. This increases the difficulty of using our tool for game developers.

4.1.2 Use geometry Shader

Using geometry shader can prevent adding more vertices as well as move the work into shader. This method adds one more information, face normal, into vertices in the geometry shader stage. In this way it makes sure each triangle’s three vertices have same face normal.
4.1.3 Use derivatives

Using derivative also prevent adding more vertices and its implemented in shader as well. This method make use of the world space normal information of pixel and recalculate each pixel’s normal in fragment shader. We finally decided to go with this method in our implementation.

4.2 Mesh Simplification

4.2.1 Introduction

As shown in the previous images, the character model after applying flat shading still looks not very “low-poly”. The reason is that the model itself contains too many vertices and flat shading can not reduce vertices count. So we then thought about reducing the vertices count before doing flat shading. And the widely used vertices reduction method is mesh simplification.

Mesh simplification is a method defined in [16] as “approximating a given input mesh with a less complex but geometrically faithful representation”. This technique is mostly used to generate level of detail (LOD). With LOD, objects far away from the camera have their mesh simplified to save rendering cost. Notice that LOD is based on the fact that the model is far away from camera so player can not see it clearly, but there is no guarantee that all objects in low-poly games are far from player. So we need to be very careful about the level of simplification. If the level is too low, the simplification is not noticable, if the level is too high, the object will not preserve its shape.

We used the mesh simplification method based on edge contraction by iteratively contracting edges [17]. The mesh simplification code is by Whinarn [18]. We further added the function to allow user to control the level of simplification by a coefficient from 0 to 1.
Figure 20: The same object without flat shading (top), with flat shading (middle), and with flat shading plus mesh simplification (bottom)
As shown in Figure 20 above, after applying flat shading to an object, it changes from smooth surface to wireframe surface. Mesh simplification further reduces the number of vertices of the mesh.

As mesh simplification method actually achieve better result than just using flat shading. But it is not working with skinned mesh or static objects.

4.2.2 Skinned Mesh

Skinned mesh is a special mesh that is combined with bone animation. It is normally used for character animation with bending joints in games. If we did a mesh simplification, the animation information is destroyed when the vertices are modified and the character can no longer perform any animation.

4.2.3 Static Object

Static object also can not work with mesh simplification. An object marked as static means that it will not change its shape nor position during the whole game. Static is often used on environment objects that guaranteed not changing in the game. Objects are required to be static in many game techniques like mesh collision and global illumination. Mesh simplification changes the mesh data which means that it modifies the object. So objects must be non-static to be used for mesh simplification.

Figure 21: static tag in Unity 3D game engine
4.3 In-game implementation

Like rewriting mesh vertices, mesh simplification also can not be implemented in shader. This creates some inconvenience for using our tool. Sadly it is impossible with current technology to modify mesh information in shader because in modern GPU, every vertex is processed without knowing any information about it neighbor vertex. We implemented mesh simplification as an add-on function to flat shading. Users can just stay with flat shading by default or choose to add mesh simplification if possible.

Based on using our tool in game, we found out that low-poly style looks more obvious on round objects like trees and the character than square objects like the blocks. Game developers should take this into consideration before trying to add low-poly style into their game.
5 PIXEL ART

Pixel art is a simulation of the art style of old 8-bit and 16-bit video games. Those games decades ago did not have very large range of colors or high resolutions. So what pixel art focus on is reducing color range and resolution.

As shown in Figure 22, Tales of the Neon Sea is a typical pixel art game. There are lots of aliasing like at the leg of the character to create an effect of low resolution. And the usage of color is also restricted in a limited range, for example, there are only four to five degree of brown at the coat of the character.

5.1 Reduce Color Range

The diffuse ramp technique is also used here to reduce color range. But instead of mapping the normal vector dot light direction to a new value and calculate the diffuse color based on the new value, here it is directly mapped to a diffuse color. Which color to map is decided by a palette created by the the artists.
5.2 Reduce Resolution

To reduce the resolution of an object, the first step is to render this object onto a low resolution sprite. Then the sprite is enlarged to the normal size and rendered onto screen. This means that the character in general pixel art games are not 3D models, even if once they were, instead they are just moving sprites. Anti-aliasing is disable to stop the image from being blurry and thus preserving the true pixel colors as the artist intended.

After researching multiple pixel art implementation methods nowadays, we found it impossible to add this art style to our tool. Because applying shader (diffuse ramp part) is only a small part of implementing a pixel art style, even if we added this shader into our tool, developers still need spend lots of time implementing other parts. They won’t be saving time, instead they are restricted by the setting of our tool. Another reason is that pixel art is not suitable for many 3D or 2.5D games where objects can not be drawn on a sprite. Considering the implementation difficulty and the general usability, we decided not implementing this style.
6 TOOL DESIGN AND USAGE

We implemented each style in one shader and wrote a small script to enable switching among them with as simple as a key press. The implementation is based on the shader replacement function provided by Unity 3D, which can replace shaders of all the objects seen by the camera.

We build an array of shaders, one for each art style. Since we developed majority of our functions in shaders, so just switching shaders would switch the effect. We assign numbers on the keyboard to the effects, “1” for no effect, “2” for the cartoon effect, “3” for the low-poly effect, etc, people can select the art effects they want by press the corresponding key.
7 RESULT AND FUTURE WORK

Although it is unfortunate that we could not implement all three art styles, we did finished the tool that can switch between no style, cartoon style, and low-poly style. We also implemented this tool into a demo game based on Unity Survival Shooter Tutorials. The result of the final game is sown in the figure below:
Figure 23: Final game with no effect (top), cartoon style (middle), and low-poly style (bottom)

The game is playable at:


In the future, we plan to test this tool on more games, and ask game developers to try our tool and give feedbacks about the looking of each style and the experience of using the tool. And based on the feedback we will refine our tool. The mesh simplification is not working correctly with skinned mesh, we want to do further research to solve this problem. Lee provides a potential solution for animation mesh simplification [20], this is one direction for further research. Since pixel art style is not achieved using pure shader method, we want to make a different tool that can achieve pixel art style using other methods and provide this option to our users.
REFERENCES


https://roystan.net/articles/toon-shader.html

https://www.diyphotography.net/3-point-lighting-technique/


https://github.com/Whinarn/UnityMeshSimplifier


APPENDIX

SURVEY RESULT

**Do you agree that unique art style can help a game stands out?**
13 responses

**What is the game that you think has the most stunning art style?**
11 responses
When developing your games, what is your attitude about art style?

13 responses

- I have to going for this specific art style! I will figure out the game play... (30.8%)
- I already have game story and mechanics designed, and I think this... (7.7%)
- I don't have preference, but I will figure it out through prototype stage. (7.7%)
- I don't worry too much about this. I will just go with the assets that are avail... (38.5%)
- No opinion yet, just learning about... (7.7%)
- Art is important, but the game shoul... (7.7%)

If there is an asset that allows you to add different art styles for your 3D models in Unity, how would you rate your desire for it.

13 responses

- 0 (0%)
- 1 (7.7%)
- 2 (15.4%)
- 3 (23.1%)
- 4 (0%)
- 5 (7.7%)
- 6 (46.2%)
- 7 (0%)
- 8 (15.4%)
- 9 (0%)
- 10 (0%)
Cel shading code:

```csharp
Shader "Unlit/bf+diffuse+rim"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
        _OutlineWidth ("Outline Width", Range(0.0,0.1)) = 0.05
        _OutlineColor ("Outline Color", COLOR) = (0,0,0,1)
        _DiffuseRamp("Diffuse Ramp", 2D) = "white" {}
        _DiffuseEnhance("Diffuse Enhance", Range(1.0,10.0)) = 1.0
        _ZOffSet("Z Off Set", Range(0.0,1.0)) = 0.0
        [MaterialToggle]_UseRimLight("Use Rim Light", Float) = 0
        _RimPow("Rim Light Power", Float) = 1
        _RimThreshold("Rim Light Threshold", Range(0.0,1.0)) = 0.3
        [MaterialToggle]_RimInShadow("Enable Rim Light In Shadow", Float) = 0
        _Tint("Tint", Color) = (1, 1, 1, 1)
        [Gamma]_Metallic("Metallic", Range(0, 1)) = 0
        _Smoothness("Smoothness", Range(0, 1)) = 0.5
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            Cull Front
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
                float3 normal : NORMAL;
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
                float4 vsNormal : TEXCOORD1;
            };

            float _OutlineWidth;
        }
    }
}
```
fixed4 _OutlineColor;
float _ZOffSet;

v2f vert (appdata v)
{
    v2f o;
    //position in view space
    float4 vsPos = mul(UNITY_MATRIX_MV, v.vertex);
    o.vsNormal = float4(mul((float3x3)UNITY_MATRIX_IT_MV, v.normal), 0);
    //o.vsNormal.z = -0.5;
    o.vsNormal = normalize(o.vsNormal);
    vsPos = vsPos + o.vsNormal * _OutlineWidth * min(1,sqrt(-vsPos.z));
    o.vertex = mul(UNITY_MATRIX_P, vsPos);
    return o;
}

fixed4 frag (v2f i) : SV_Target
{
    i.vsNormal = normalize(i.vsNormal);
    //clip(0.9 + i.vsNormal.y);
    return _OutlineColor;
}
ENDCG
}
Pass
{
    Tags{ "LightMode" = "ForwardBase" }

CGPROGRAM
#pragma vertex vert
#pragma fragment frag
#pragma multi_compile_fwdbase

#include "UnityCG.cginc"
#include "Lighting.cginc"
#include "AutoLight.cginc"
#include "UnityShaderVariables.cginc"

sampler2D _MainTex;
float4 _MainTex_ST;
sampler2D _DiffuseRamp;
float _DiffuseEnhance;
half _UseRimLight;
float _RimPow;

float4 _WSView;
half _RimInShadow;
float _RimThreshold;

struct a2v {
    float4 vertex : POSITION;
    float3 normal : NORMAL;
    float2 uv : TEXCOORD0;
};

struct v2f {
    float4 vertex : SV_POSITION;
    float2 uv : TEXCOORD0;
    float4 wsNormal : TEXCOORD1;
    float4 wsView : TEXCOORD2;
    float4 wsLight : TEXCOORD3;
};

v2f vert(a2v v) {
    v2f o;
    o.vertex = UnityObjectToClipPos(v.vertex);
    o.uv = TRANSFORM_TEX(v.uv, _MainTex);

    float4 wsPos = mul(UNITY_MATRIX_M, v.vertex);
    o.wsNormal = float4(UnityObjectToWorldNormal(v.normal), 0);
    o.wsNormal = normalize(o.wsNormal);
    o.wsView = float4(_WorldSpaceCameraPos0) - wsPos;
    o.wsLight = _WorldSpaceLightPos0;
    return o;
}

float4 frag(v2f i) : SV_Target {
    i.wsNormal = normalize(i.wsNormal);
    i.wsLight = normalize(i.wsLight);
    fixed4 color = tex2D(_MainTex, i.uv);
    float ndotl = dot(i.wsNormal, i.wsLight);
    float lambert = saturate(ndotl);
    float halflambert = pow((0.5 * ndotl + 0.5), 2);
    float3 directDiffuse = diff * _LightColor0;
    float3 diffuseColor = color.rgb * directDiffuse;
diffuseColor *= _DiffuseEnhance;

    //rim
    if (_UseRimLight == 1) {
        i.wsView = normalize(i.wsView);
        //traditional rim light calculate method, generate wrong effect when
        object is moving
float ndotv = saturate(dot(i.wsView, i.wsNormal));
//my method, use script defined view vector, perfect solve
float ndotv2 = saturate(dot(_WSView, i.wsNormal));
//method in reference, use up vector instead of view vector, can handle view not from top situation
float ndotu = saturate(dot(float4(0, 1, 0, 0), i.wsNormal));
float rimColor = pow((1 - ndotv2), _RimPow);
if (diff.x != 1 && _RimInShadow == 0)
{
    rimColor = 0;
}
//rim threshold
if (rimColor < _RimThreshold)
{
    rimColor = 0;
}
return float4(diffuseColor, 1) + rimColor;
}
return float4(diffuseColor, 1);

LODCG

}

}

Fallback "Diffuse"

Low-poly shader code:

Shader "Unlit/lowpoly"
{
    Properties
    {
        _Tint("Tint", Color) = (1, 1, 1, 1)
        _MainTex ("Texture", 2D) = "white" {}
        [Gamma] Metallic("Metallic", Range(0, 1)) = 0
        _Smoothness("Smoothness", Range(0, 1)) = 0.5
    }
    SubShader LOD 100
}
Pass
{
    Tags{
        "LightMode" = "ForwardBase"
    }
} CGPROGRAM
{
    Tags { "RenderType"="Opaque" }

    #pragma vertex vert
    #pragma fragment frag

    #include "UnityCG.cginc"
    #include "UnityPBSLighting.cginc"

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

    struct v2f
    {
        float2 uv : TEXCOORD0;
        float4 vertex : SV_POSITION;
        float3 wsNormal : TEXCOORD1;
        float3 wsPos : TEXCOORD2;
    };

    sampler2D _MainTex;
    float4 _MainTex_ST;
    float4 _Tint;
    float _Metallic;
    float _Smoothness;

    v2f vert (appdata v)
    {
        v2f o;
        o.vertex = UnityObjectToClipPos(v.vertex);
        o.uv = TRANSFORM_TEX(v.uv, _MainTex);
        o.wsNormal = UnityObjectToWorldNormal(v.normal);
        o.wsPos = mul(UNITY_MATRIX_M, v.vertex);
        return o;
    }

    fixed4 frag (v2f i) : SV_Target
    {
        float3 dpdx = ddx(i.wsPos);
float3 dpdy = ddy(i.wsPos);
i.wsNormal = normalize(cross(dpdy, dpdx));
float3 lightDir = _WorldSpaceLightPos0.xyz;
float3 viewDir = normalize(_WorldSpaceCameraPos - i.wsPos);

float3 lightColor = _LightColor0.rgb;
float3 albedo = tex2D(_MainTex, i.uv).rgb * _Tint.rgb;

float3 specularTint;
float oneMinusReflectivity;
albedo = DiffuseAndSpecularFromMetallic(
    albedo, _Metallic, specularTint, oneMinusReflectivity
);

UnityLight light;
light.color = lightColor;
light.dir = lightDir;
light.ndotl = DotClamped(i.wsNormal, lightDir);
UnityIndirect indirectLight;
indirectLight.diffuse = 0.1;
indirectLight.specular = 0.3;

return UNITY_BRDF_PBS(
    albedo, specularTint,
    oneMinusReflectivity, _Smoothness,
    i.wsNormal, viewDir,
    light, indirectLight
);
}

ENDCG

Code for switching style:

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

public class SwitchStyle : MonoBehaviour {

    //all the art style Shaders
    public Shader[] effectShaders;
}
//moving objects
public Material[] mat;

//if do mesh simplification
public bool doMeshSimplification = true;

//objects that will be performed mesh simplification
public GameObject[] meshObjs;

//mesh simplification quality
public float quality = 0.5f;

//store the origin meshes
private List<Mesh> originMeshes;
private bool isLowPoly = false;

void Start () {
    //define world space view direction outside shader to avoid moving
    //problem of rim light.
    mat[0].SetVector("_WSView", new Vector4(-transform.forward.x, -transform.forward.y, -transform.forward.z, 0));
    mat[1].SetVector("_WSView", new Vector4(-transform.forward.x, -transform.forward.y, -transform.forward.z, 0));

    originMeshes = new List<Mesh>();
}

void Update () {
    //switch among art styles using number keys.
    if (Input.GetKeyDown(KeyCode.Alpha1))
    {
        GetComponent<Camera>().SetReplacementShader(effectShaders[0], "RenderType");
        if (doMeshSimplification && isLowPoly)
        {
            RestoreMesh();
            isLowPoly = false;
        }
    }
    if (Input.GetKeyDown(KeyCode.Alpha2))
    {
        GetComponent<Camera>().SetReplacementShader(effectShaders[1], "RenderType");
        if (doMeshSimplification && isLowPoly)
        {
            RestoreMesh();
            isLowPoly = false;
        }
    }
}
if (Input.GetKeyDown(KeyCode.Alpha3))
{
  GetComponent<Camera>().SetReplacementShader(effectShaders[2], "RenderType");
  if (doMeshSimplification && !isLowPoly)
  {
    SimplifyMesh();
    isLowPoly = true;
  }
}

void SimplifyMesh()
{
  for (int i = 0; i < meshObjs.Length; i++)
  {
    var meshSimplifier = new UnityMeshSimplifier.MeshSimplifier();
    if (meshObjs[i].GetComponent<SkinnedMeshRenderer>() != null)
    {
      meshSimplifier.Initialize(meshObjs[i].GetComponent<SkinnedMeshRenderer>().sharedMesh);
      originMeshes.Insert(i, meshObjs[i].GetComponent<SkinnedMeshRenderer>().sharedMesh);
    }
    if (meshObjs[i].GetComponent<MeshFilter>() != null)
    {
      meshSimplifier.Initialize(meshObjs[i].GetComponent<MeshFilter>().sharedMesh);
      originMeshes.Insert(i, meshObjs[i].GetComponent<MeshFilter>().sharedMesh);
    }
    meshSimplifier.SimplifyMesh(quality);
    var destMesh = meshSimplifier.ToMesh();
    //destMesh.bindposes = meshObjs[i].GetComponent<SkinnedMeshRenderer>().sharedMesh.bindposes;
    if (meshObjs[i].GetComponent<SkinnedMeshRenderer>() != null)
    {
      meshObjs[i].GetComponent<SkinnedMeshRenderer>().sharedMesh = destMesh;
    }
    if (meshObjs[i].GetComponent<MeshFilter>() != null)
    {
      meshObjs[i].GetComponent<MeshFilter>().sharedMesh = destMesh;
    }
  }
}

void RestoreMesh()
if (isLowPoly)
{
    for (int i = 0; i < meshObjs.Length; i++)
    {
        if (meshObjs[i].GetComponent<SkinnedMeshRenderer>() != null)
        {
            meshObjs[i].GetComponent<SkinnedMeshRenderer>().sharedMesh = originMeshes[i];
        }
        if (meshObjs[i].GetComponent<MeshFilter>() != null)
        {
            meshObjs[i].GetComponent<MeshFilter>().sharedMesh = originMeshes[i];
        }
    }
}