**Module: Graphics Programming**

**Module Code: M3I622944**

**Name: Kristiyan Nenov**

**Student ID: S1512113**

*I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award*.

*Signature*: Kristiyan Nenov

Contents

[**Rim Lighting** 3](#_Toc521672878)

[**Toon Shading** 3](#_Toc521672879)

[**RimToon** 4](#_Toc521672880)

[**ADS Lighting** 4](#_Toc521672881)

[**Blinn-Phong Lighting** 5](#_Toc521672882)

[**Skybox** 5](#_Toc521672883)

[**Reflection** 6](#_Toc521672884)

[**Fog** 6](#_Toc521672885)

[**ADS + Fog / RimToon + Fog** 7](#_Toc521672886)

In this report, we’re going to discuss the theoretical underpinnings, practical implementation principles and possible application areas of several special effects.

# **Rim Lighting**

Rim lighting, also known as back lighting, is an effect that simulates the bleeding of light around an object from sources that are behind it or otherwise have no effect on the shaded surfaces of the model. We can simulate the effect by determining how closely the view direction comes to glancing the surface. To implement this, all we need is the surface normal and the view direction. When the view direction is face on to the surface, the view vector will be collinear to the surface normal and so the effect of rim lighting will be the least. When it glances the surface, the normal and the view vectors will be almost perpendicular to one another and the rim light effect will be the greatest. We can use the dot product between the surface normal and the view direction to calculate the light intensity. The way the fragment shader works is it firstly converts normal to view space, then it calculates the position in clip space. It normalizes the eye vector and then calculates the rim contribution by using the dot product as discussed above. It finally applies it by constraining it in a range using a smoothstep function.

# **Toon Shading**

The basic idea of toon shading is to use the diffuse lighting intensity (the dot product between the eye space surface normal and light direction vector) and set it to a couple different values so the color doesn’t transition smoothly, but gives a cartoonish effect. The way it works is, it gets the diffuse lighting intensity by calculating the dot product between the light direction and the normal and then checks the value of the result. The result can only be between 0 and 1. It then has preset values (such as 0.95/0.5/0.25) which set the color intensity to a preset value. Then that value is used to display to the screen.

Toon or cell shading is a process designed to make 3D computer graphics appear to be flat. As mentioned above the effect of this specific technique is a fully visual so it finds most of its use in altering the overall appearance of games when utilised. This effectively allows creators to take a different approach on the visual side of their games. A great example of the use of toon shading is The Legend of Zelda: Breath of the Wild. A very successful title that although has a good base still attracts an enormous amount of people with its gorgeous art style that utilises cell shading.

# **RimToon**

This is basically a combination of both previously discussed effects. It works in exactly the same way, it first does the Rim lighting and then applies toon shading and combines the results.

The combination of the two lighting models allows for the creation of a lighting effect created by a specific light source that resembles a close approximate of a real-life example. Again, an example of this can be found in the above-mentioned game The Legend of Zelda: Breath of the Wild. As it was said above this game utilizes these specific lighting models to create a very varied art style and differentiate itself from other titles. This type of lighting model can very well serve in specific genres of games such as educational games or games for an younger audience due its realistic yet somewhat animated nature.

# **ADS Lighting**

ADS Lighting combines the three types of light – Ambient, Diffuse and Specular. In order to understand how ADS lighting works, we have to delve deeper into each of the three types of light. Ambient light doesn’t come from any particular direction. It has an original source somewhere, but the rays bounce around the scene and become directionless. Objects lit by it are evenly lit on all surfaces in all directions. The diffuse light model is a distant directional light source such as the sun light. The light reflected off the surface is scattered equally in all directions. Specular light, like diffuse light, is directional, but it interacts more sharply with the surface and in one particular direction. Reflected light is concentrated along the direction of the perfect reflector. What a viewer sees depends on the angle. It tends to cause a bright spot on the surface it shines on, which is called the specular highlight. The way ADS lighting works is by firstly calculating the ambient light. That’s done by multiplying the strength of the effect by the light color. Moving on to diffuse light, firstly the normal and the resulting direction vector are normalized in order to work with unit vectors. Then the actual diffuse light impact is calculated by taking the dot product of the normalized values mentioned above. Finally, the resulting value is multiplied with the light’s color. This finalizes the diffuse component. After that, the specular light is done. It starts off by setting the specular strength. Then the view direction vector and the corresponding reflect vector are calculated. Then the dot product between the view direction and reflect direction is calculated and it is raised to the power of 32, which is a shininess value of the highlight. The result of all three calculations for ambient, diffuse and specular lighting is then combined and multiplied by the color of the object to make the final result of ADS lighting.

In the gaming industry ADS lighting serves a specific function as mentioned above it utilizes several different lighting models to produce its effect. For example, in certain games that feature an active day/night cycle the transition between night and day needs to be clearly evident and have an impact on the player in order to recreate the actual event of sunrise/dawn. While utilizing only the suns light model which is only diffuse light can recreate this specific game aspect this would without a doubt be a very flat and unrealistic representation of the event. Therefore, by using the diffuse light and combining it with the other two lighting models (ambient and specular) a realistic scene transition can be created to better immerse the player and create a more atmospheric scene. This allows for developers to really underline the two different stages of their game in the face of day and night time.

# **Blinn-Phong Lighting**

This lighting method is an extension to the ADS lighting. It’s very similar to it, with a small difference in calculating the specular model. Instead of using the reflection vector, it uses the halfway vector that is a unit vector exactly halfway between the view and light directions. The closer this vector aligns with the surface’s normal vector, the higher specular contribution it’s going to have. The specular vector is calculated by a clamped dot product between the normal and halfway vectors.

Even though both methods utilise all the matrices, model, view and projection to create the effect of lighting the Blinn-Phong method allows for a more accurate model representation that leads to a more detailed object and a clearer image. An area where this can be useful could be the Serious game scene and more specifically in close view scenarios such as medical trainings. Brain surgery, eye surgery and many others require an extremely accurate representation of the human body to ensure a near perfect example for what the trainees to expect in a real world scenario. By making sure that all models in the game are illustrated and defined as well as possible the realism of the games increases dramatically and therefore their effectiveness with it.

# **Skybox**

A skybox is a large cube that goes around the entire scene. It contains six images. OpenGL has the functionality to implement a skybox, which makes it a lot easier. The skybox is still quite complex in the way it works. The first thing needed is a way to load the skybox. Since it’s just a cubemap, a function is made that accepts a vector of 6 texture locations. This function simply loads the cubemap though, displaying it is done through shader work. In the vertex shader, the incoming position vectors are set as the outgoing texture coordinates for the fragment shader. The fragment shader samples the texture values from the cubemap. Rendering the skybox is done by firstly disabling depth writing in order to make sure the skybox is always in the background. Then we bind the cubemap texture and the skybox sampler is automatically filled with the skybox cubemap.

Skyboxes are traditionally used to create backgrounds (scenery, actual sky or both) in order to make the game world look bigger than it actually is. This is done for several reasons such as immersion, a more complete look of the game world and others. Although skyboxes are a visual addition to games they can also serve another function. Skyboxes can reduce the load on games as they are drawn static objects that only recreate the effect of a generated background when in fact they keep the player inside a pre-created textured cube and recreate the effect of a background and sky. This can not only save processing power as previously mentioned but also smaller and for certain situations large development teams can save a lot of time by utilising well textured skyboxes instead of having to either hand make the environment or even make algorithms for randomly generated terrain.

# **Reflection**

Reflection, otherwise known as environment mapping, is the effect that an object reflects its surrounding environment. In order to reflect the skybox, two vectors are needed. Those are the view direction vector and the reflection vector. The view direction vector is calculated using the position of the object and the camera’s position. The result is then used to calculate the reflection vector.

The main function that a reflection function serves in a game is to increase the realism and immersion of players. Weather it being a reflection of a scenery onto water or a full reflection recreating the effect of a mirror, it is clear that reflections are things that we encounter on an everyday basis and imagining games without them is to say the least, not ideal. However, in modern games reflections can also be used as a game mechanic and not only as a cosmetic addition to the look of the game. For example, in the game Tom Clancy's Rainbow Six Siege players rely on many different aspects to play to their full potential. One of these aspects is reflections from other players on surfaces. Being able to see the reflection of the shiny weapon or headgear your opponent is carrying can give you an advantage in the future to come brawl allowing you to pre-fire and thus win the duel. This is a way of looking as reflections as part of the mechanics of the game and not only as a graphic feature.

# **Fog**

The fog effect simulates the effect of a fog. The further away the camera is from an object, the higher the fog effect will be. In order to create the fog effect, firstly the distance to the object is determined. Then the fog factor is calculated using the distance and the fog factor is set between 0 and 1. After that the final color of the object is mixed with the fog color and the fog factor.

# **ADS + Fog / RimToon + Fog**

Both ADS and RimToon with added fog effect are done in the same way. After the calculations for the ADS and RimToon effects are done, their result is then used to mix the color with the fog effect. It is done in exactly the same way as the Fog effect itself, but instead of using a fixed color, the result of the effects is used to mix with the fog effect.

Fog and ADS Fog are effectively two similar ways of achieving the same effect which is of course the simulation of fog. The use of fogs in games has been a technique utilized in many different games and is almost a standard in the gaming industry. It can easily be associated with horror games as its main function is to limit the visibility of the player and to disguise certain objects. This leads to increased tension in the player to bring a more immersive experience. However, setting the mood of the game is not the only use for fogs. A very useful aspect of the fog is its real-world effect which is limiting the vision of people located in it and this translates very well to either driving or flying simulators. A dense fog can easily increase the difficulty of a player looking for more of a challenge when learning to fly a plane in a simulator.