Inverse light estimation

Estimating the light of outdoor scenes through textures.

Project Report

Mikael Therkelsen

Aalborg University
Department of Architecture, Design, and Media Technology

Department of Architecture, Design, and Media Technology
Aalborg University
https://www.create.aau.dk/

Title:
Inverse light estimation

Theme:
Master thesis

Project Period:
Spring semester 2024

Participants:
Mikael Therkelsen

Supervisors:
Claus Madsen

Page Numbers: 39

Date of Completion:
May 28, 2024

Abstract:

In this project, the focus was on estimating the light settings of an outdoor scene, using textures obtained through the Unity game engine. The implemented system is meant to work with augmented reality, by obtaining information about the scene and estimating the light of that scene. The estimated light can then be used to shade an object that has been inserted into the scene, to make the object blend into the scene. To estimate the light settings, a lighting model was found and implemented using Python. The Python program uses the textures obtained through Unity, to estimate the light settings. The Python program was connected to the Unity project using sockets, and the estimated values were sent through the connection. The system performed mediocre but still showed promise. The estimated values were not accurate when compared with the actual lighting values, but did follow as expected when the lighting changed. The estimated value did make the inserted object blend in, in certain lighting settings. To make a more stable system, more work should be put into the system.

Preface

[illegible]

Mikael Nørlund Therkelsen

<mtherk19@student.aau.dk>

Aalborg University, May 28, 2024

Contents

1 Introduction

Augmented reality exists along the Reality-Virtuality spectrum, spanning from unaltered reality to fully computer-generated virtual environments[1]. Between these extremes, we find mixed realities like augmented reality, where digital objects are inserted into the real world, and augmented virtuality, where real-world objects are inserted into the digital world. Achieving convincing augmented reality requires seamlessly integrating virtual objects into the environment, ensuring they align with both position and lighting.

Augmented reality (AR) is a rising platform, that is used in many fields to aid in understanding topics or aid in performance[2]. And with AR APIs such as Unity's ARFoundation, Google's ARCore, and Apple's ARKit, it has never been easier to develop solid AR applications[3]. These applications can serve multiple functions, such as integrating virtual objects into the real world or modifying the appearance of already existing objects. But these APIs still fall a bit short when lighting an augmented object[3]. And as Gao et. al. [2] discovered, misalignment of lighting between the augmented object and the real world, could impact the perception of the augmented object.

In a fully virtual environment, the shading of objects is a lot easier, due to the lighting conditions being known. This is however not the case when rendering objects in augmented reality[4]. This can lead to differences in the lighting of the real-world scene and the augmented object. Therefore the lighting must be estimated from the scene before shading any augmented objects, for realistic augmented objects.

An example of lighting between the real world and an inserted object can be seen in Figure 1.1. Here the left object is a real-world object, while the right is a 3D model of the left object. The 3D model has baked lighting, which results in misalignment of the lighting. This is of course only one of the many current solutions for lighting in AR there are.

When researching light estimation of a real-world scene, neural networks have been a prominent tool. Neural networks have been used for multiple purposes, such as complete inverse-rendering of a real-world scene[5, 2], calculating Spherical Harmonic(SH) coefficients[6] or just estimating the light direction[7].

There are also other methods for estimating the lighting of real-world scenes. One method could be to use image processing to obtain a shadow color, and use that color to calculate the light contribution from the hemisphere and the sun[8]. Another method could be to obtain specific textures of the scene[9], and then calculate the light contribution using those textures. And with the previously mentioned AR framework, obtaining the necessary textures to estimate the lighting could potentially be possible.

This therefore leaves the following problem statement:

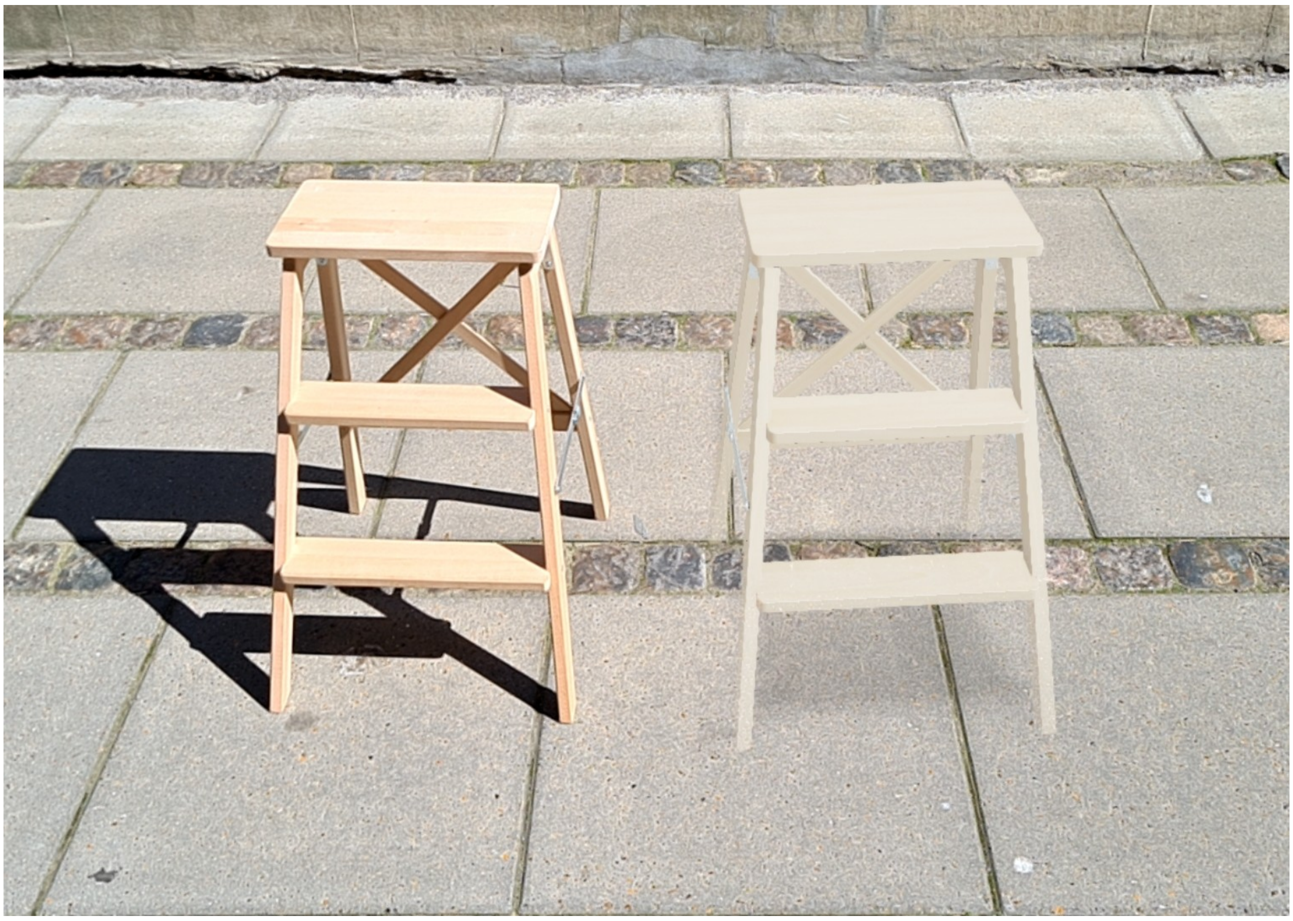


Figure 1.1: An image taken using the Ikea home application. The right-side object is an inserted 3D model of the left-side object. The right-side object has baked-in lighting, which does not align with the lighting of the real-world scene.

“How can the necessary textures of a real-world scene, be collected and used for real-time lighting estimation.“

2 Background research

With tools like Unity's ARFoundation, Google's ARCore, and Apple's ARKit, many challenges relating to augmented reality such as plane detection, object tracking, and object insertion have been overcome. However, challenges regarding accurate estimation of light and shadows for augmented objects still persist[4, 3]. These include issues like obtaining precise sunlight direction data, leading to shadows appearing incorrectly, and accurately light colour contribution estimation, affecting how light colours objects in the scene and the visibility of shadows.

Currently, a lot of work has been put into researching the use of neural networks, for predicting the lighting conditions of a given scene. These neural networks have been used to both inverse render an entire scene or part of it[5, 2] and to predict smaller components of the lighting. These components could be light directions and placement[7], color of lights through spherical harmonics[10, 6] or HDRI maps[11, 12, 13, 14]. These neural networks have, however, only focused on lighting the augmented objects and not on the shadows produced by the objects, which is also an important component when realism is the focus of augmented reality. An unfortunate downside to the use of neural networks is gaining the data that will be used to train the neural network. Not only do the data have to be of a certain quality, but there also have to be enough data points for the neural network to be generalized[9]. Therefore, there could be benefits to pursuing other routes.

For example Wei et. al. [8] used canny edge detection, to find the edges of shadows in the real world. These edges were then translated into the 3D space of the virtual world, and used to create a shadow volume. However, they did not calculate the direction vector of the sun, based on the sun's position in relation to the device, but by estimating the edges of the shadow caster. With the shadow edges and edges of the shadow caster, a shadow volume was created and a depth-test was performed to see if an object was in the real-life shadow.

But to translate the edges of the shadows from the real world's 3D space into the virtual world's 3D space, knowledge of the shadow's position on three axes must be known. To get the shadow's position on all three axes, Wei et. al. [8] used RGBD images. With the ARCore framework[15], it is now possible to get a depth map of the scene no matter if the device in use has a depth sensor, as long as it has the necessary processing power. With the depth map, an approach like the one presented by Wei et. al. [8] could potentially be implemented onto a mobile device.

Another method was also presented by Madsen et. al. [9], where the irradiance of the scene was calculated. This was done through a radiance image of the scene and an albedo map of the scene, along with the ambient light, shadows, and normals of the scene. Now with the information about the scene, Equation 2.1 can be used to calculate the irradiance values of the scene, if the light direction of the sun can be found. The equation system in Equation 2.1 estimates the contributed light from the sunL_d and the contributed light from the hemisphereL_a, so the values can be used when shading objects.

$$\begin{bmatrix} P(\vec{x_1}) \\ P(\vec{x_2}) \\ \ldots \\ P(\vec{x_m}) \end{bmatrix} = \begin{bmatrix} \frac{\rho_d(\vec{x_1})}{\pi} * c(\vec{x_1}) & \frac{\rho_d(\vec{x_1})}{\pi} * s(\vec{x_1}) * (\vec{n}(\vec{x_1}) \cdot \vec{l}(\vec{x_1})) \\ \frac{\rho_d(\vec{x_2})}{\pi} * c(\vec{x_2}) & \frac{\rho_d(\vec{x_2})}{\pi} * s(\vec{x_2}) * (\vec{n}(\vec{x_2}) \cdot \vec{l}(\vec{x_2})) \\ \ldots & \ldots \\ \frac{\rho_d(\vec{x_m})}{\pi} * c(\vec{x_m}) & \frac{\rho_d(\vec{x_m})}{\pi} * s(\vec{x_m}) * (\vec{n}(\vec{x_m}) \cdot \vec{l}(\vec{x_m})) \end{bmatrix} \begin{bmatrix} k & \vec{L_a} \\ k & \vec{L_d} \end{bmatrix} \tag{2.1}$$

With the L_a and L_d scalar values, Equation 2.2 can be used when shading an augmented object[9, 16] and Equation 2.3 can be used when colouring the shadow cast by the augmented object[16]. Besides the L_a and L_d scalar values, the lighting before and after the augmented object is also necessary when shading the shadow cast. The lower part of the fracture in Equation 2.3 is the lighting before the augmented object is inserted and the upper part, is the lighting after the augmented object is inserted. Equation 2.1, Equation 2.2, and Equation 2.3 will be further explained in chapter 3

$$L_o(\vec{x}) = \frac{\rho_d(\vec{x})}{\pi} * (\vec{L_a} * Ambient_a + \vec{L_d} * Sun_a * (\vec{n}(\vec{x}) \cdot \vec{l}(\vec{x}))) \tag{2.2}$$

$$L_{after} = L_{before} * \frac{\sum_1^l P_i * S_{i-after} * (\vec{d_i} \cdot \vec{n})}{\sum_1^l P_i * S_{i-prev} * (\vec{d_i} \cdot \vec{n})} \tag{2.3}$$

The challenge now becomes obtaining the different images necessary to do the calculation in Equation 2.1.

Obtaining the shadow map and ambient light could now be possible with the new Google geospatial API [15] because it can provide 3D models of buildings of a location. With these 3D models, it is possible to obtain a shadow map of the current location, as long as the light source in digital space is representative of the sun. So to get a representative light, a direct light source can be used[17], as long as the direction of the light is correct.

This therefore leaves obtaining the albedo image, as the last hurdle. But because the 3D models the Google geospatial API [15] provides also have textures, these could be used to obtain an albedo map of the scene. However, as it can be seen in Figure 2.1, the textures collected are not without lighting. It would therefore not be possible to calculate accurate lighting with these textures, without processing the textures first, which is outside the scope of this project.

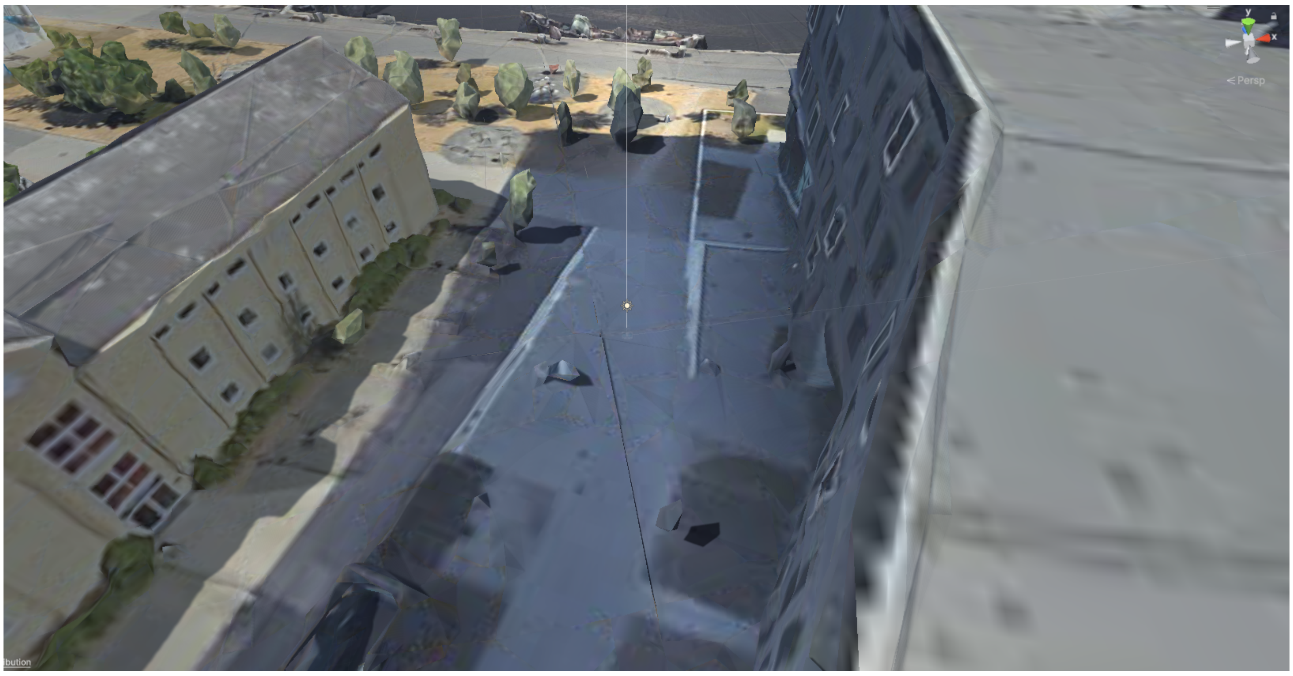


Figure 2.1: A screenshot of the textures obtained from the Google geospatial API.

This project will therefore work with synthetic data, where scenes will be created in Unity to represent an outdoor scene.

3 Lighting model

Understanding the relation between skylight and sunlight is essential when emulating real-world light [18]. This chapter explains how Equation 3.1 estimates outdoor lighting and how the estimated value can be used in rendering augmented objects.

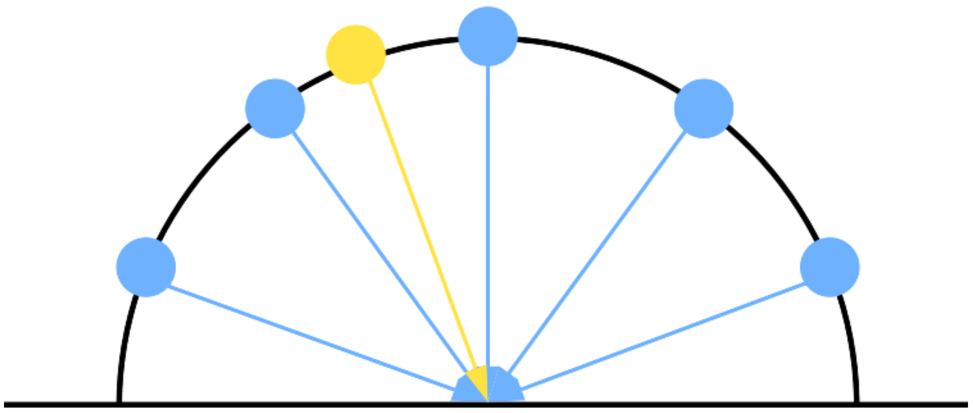


Figure 3.1: A diagram of the lighting model. Here the blue spheres represent points in a hemisphere and the yellow sphere represents the sun. The total amount of light received by a point depends on the total amount of visible sky and the angle between the surface normal and the sun's direction[16].

Figure 3.1 is a visual representation of how light interacts with a given point. A point will receive light from both the sun and the hemisphere if the angle between the light source and the surface normal is less than 90 degrees and the light source is not blocked[16]. When looking in Table 3.1, the textures of a 3D scene can be seen. Here the sun shadow texture shows the points in the 3D scene where the sunlight is blocked by objects. The ambient light textures show how much light a point receives from the hemisphere, by giving the pixels a value between 0 and 1. These two textures therefore represent the amount of light a given point in a 3D scene receives from both the sun and the hemisphere.

Knowing where the light is hitting in a 3D scene is enough to shade an object, if the color of the light is known[18]. This is where Equation 3.1 can be used. By providing the five textures in Table 3.1, Equation 3.1 can be used to calculate L_a and L_d, which is the light contributed by the sun (L_d) and the hemisphere (L_a).

Equation 3.1 shows three matrices, a Nx1 matrix, a Nx2 matrix, and a 2x1 matrix. The Nx2 matrix uses the last four textures in Table 3.1, which are the albedo texture, sun shadow texture, ambient texture, and the normal texture. Here the left column calculates the ambient light contribution, by taking the pixels in the albedo texture ($\frac{\rho_d(\vec{x_n})}{\pi}$) and multiplies the pixel value, with the corresponding ambient light texture ($c(\vec{x_n})$) pixel value. The right column calculates

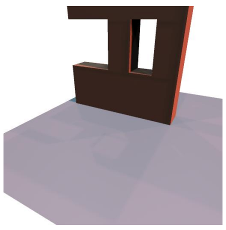
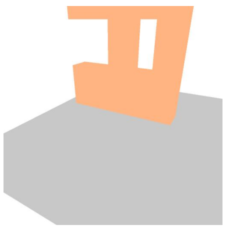
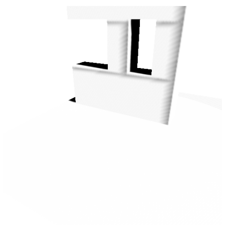
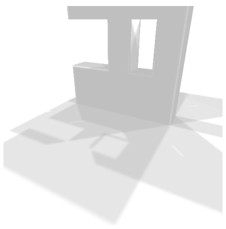
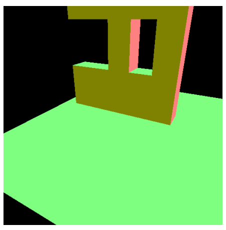
 Radiance	 Albedo	 Sun shadow	 Ambient light	 Normal

Table 3.1: Table of the necessary textures, when estimating the lighting, using Equation 3.1

the light contribution of the sun, by taking the pixels in the albedo texture and multiplying the pixel value, with the corresponding pixel value from the sun shadow texture ($s(\vec{x_1})$) and the dot product between the light direction and the surface normal from the normal texture ($\vec{n}(\vec{x_1}) \cdot \vec{l}(\vec{x_1})$). This will result in the result being 0 if there is no sun at the given pixel, and lowering the result if the angle between the sun and the normal goes towards 90 degrees.

The 2x1 matrix contains the irradiance value from the hemisphere and the sun. Multiplying the Nx2 matrix together, with the 2x1 would result in a 3D scene with light based on the irradiance values. Therefore, if the textures in last four textures in Table 3.1 are being used in Equation 3.1, and the L_a and L_d values are known, the result should look like the radiance texture in Table 3.1.

$$\begin{bmatrix} P(\vec{x_1}) \\ P(\vec{x_2}) \\ \dots \\ P(\vec{x_m}) \end{bmatrix} = \begin{bmatrix} \frac{\rho_d(\vec{x_1})}{\pi} * c(\vec{x_1}) & \frac{\rho_d(\vec{x_1})}{\pi} * s(\vec{x_1}) * (\vec{n}(\vec{x_1}) \cdot \vec{l}) \\ \frac{\rho_d(\vec{x_2})}{\pi} * c(\vec{x_2}) & \frac{\rho_d(\vec{x_2})}{\pi} * s(\vec{x_2}) * (\vec{n}(\vec{x_2}) \cdot \vec{l}) \\ \dots & \dots \\ \frac{\rho_d(\vec{x_m})}{\pi} * c(\vec{x_m}) & \frac{\rho_d(\vec{x_m})}{\pi} * s(\vec{x_m}) * (\vec{n}(\vec{x_m}) \cdot \vec{l}) \end{bmatrix} \begin{bmatrix} k * \vec{L_a} \\ k * \vec{L_d} \end{bmatrix} \tag{3.1}$$

However, because the irradiance values are unknown in AR, these must be estimated. This can be done by taking the inverse of the Nx2 matrix, turning it into a 2xN matrix, and matrix multiplying it with the Nx1 matrix containing the pixels from the radiance texture. This will result in a 2x1 matrix containing the irradiance values, which can be used in rendering augmented objects.

When shading an augmented object, Equation 3.2 can be used for each fragment. The resulting colour of a given fragment $L_o(\vec{x})$, is found by multiplying an ambient result and a diffuse result with the albedo of the object and then adding the results together.

The ambient result is calculated by multiplying the ambient lighting with the estimated $\vec{L_a}$ scalar ($\vec{L_a} * Ambient_a$). The diffuse result is calculated by multiplying the dot product of the light direction and surface normal of the fragment, with the estimated $\vec{L_d}$ scalar ($\vec{L_d} * (\vec{n}(\vec{x}) \cdot \vec{l}(\vec{x}))$). This is then multiplied by either 0 or 1 depending on if the fragment is in shadow or not ($\vec{L_d} * (\vec{n}(\vec{x}) \cdot \vec{l}(\vec{x})) * Sun_a$).

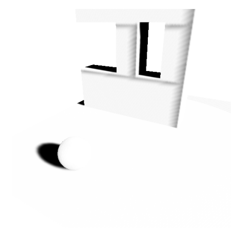
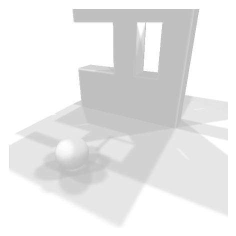
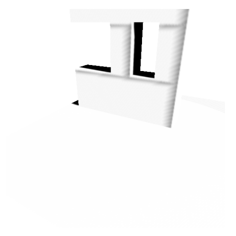
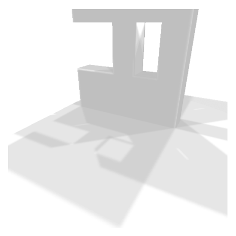
			
After sun	After ambient	Before sun	Before ambient

Table 3.2: Table of the texture containing the before and after lighting a sphere has been inserted into a scene Equation 3.1

$$L_o(\vec{x}) = \frac{\rho_d(\vec{x})}{\pi} * (\vec{L_a} * Ambient_a + \vec{L_d} * (\vec{n}(\vec{x}) \cdot \vec{l}(\vec{x})) * Sun_a) \tag{3.2}$$

When shading the surface that will receive the shadows produced by the augmented object, Equation 3.3 can be used. This equation uses the shadows after an object has been inserted into the scene, and divides it with the shadows before an object was inserted. To see how the shadows look before and after an object has been inserted into a 3D scene, see Table 3.2. Dividing the two textures of shadows with each other will result in a texture only containing the shadow produced by the augmented object.

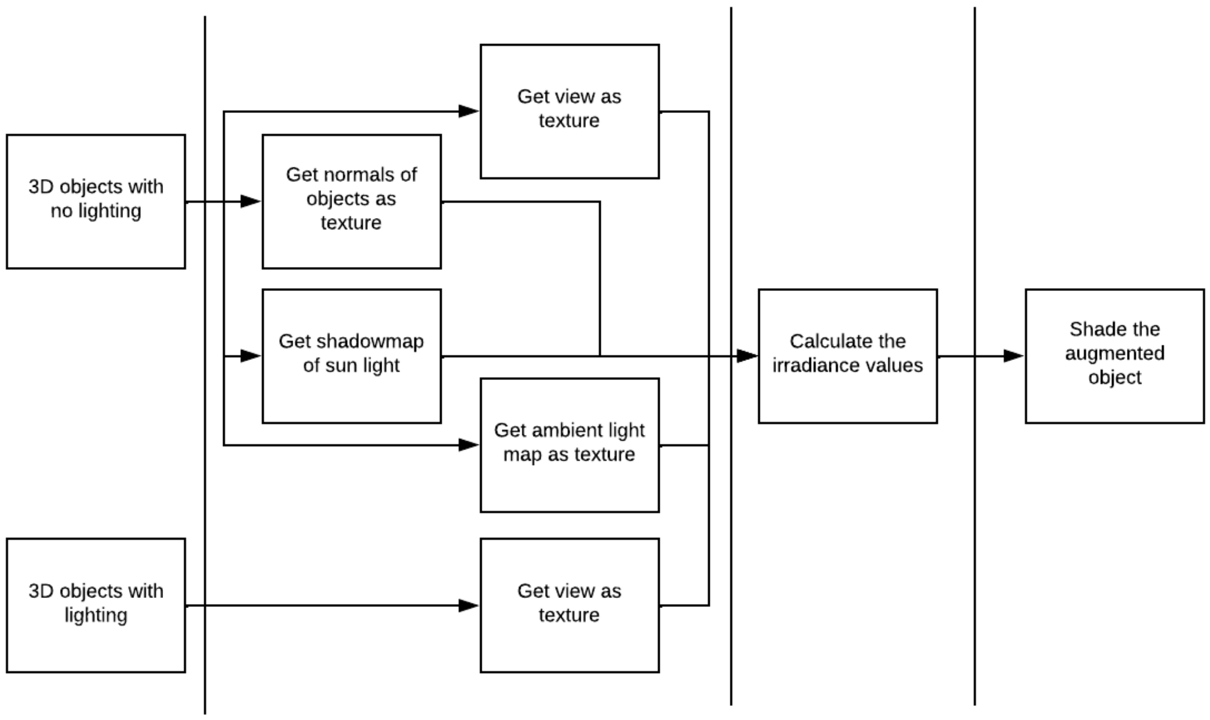
$$L_{after} = L_{before} * \frac{\sum_1^l P_i * S_{i-after} * (\vec{d_i} \cdot \vec{n})}{\sum_1^l P_i * S_{i-prev} * (\vec{d_i} \cdot \vec{n})} \tag{3.3}$$

However, only dividing the textures with each other would only result in a gray colour, and not a shadow with a colour matching the other shadows in the scene. Therefore the textures should be multiplied by the L_a scalar that was estimated using Equation 3.1. But L_a should be divided by the number of direct light sources used in the hemisphere. This therefore means that $\frac{L_a}{l} = P_i$. Multiplying the colour of the surface(L_{before}) with the shadow of the augmented object will then result in the new colour of the surface(L_{after}).

4 Render Implementation

The implemented system is a multi-step system, where the first step is to gather the necessary textures, the second step is to calculate the irradiance values using the gathered textures, and the last step is where the augmented object is shaded with the derived irradiance values. For a diagram of the process see Figure 4.1.

To speed up the implementation process, the Unity game engine was used. To control the lighting in a scene, all ambient light provided by Unity was turned off. This means that the only contributing light in the scene, are directional lights that are placed in the scene.



3D objects with no lighting
Get view as texture
Get normals of objects as texture
Get shadowmap of sun light
Get ambient light map as texture
3D objects with lighting
Get view as texture
Calculate the irradiance values
Shade the augmented object

Figure 4.1: Diagram of the flow in the implemented system. Textures are gathered and used to calculate the irradiance value. The derived values are then used to shade an object

As Figure 4.2 shows, the scene is built twice. Once where there is no light, representing what Google Geospatial API would give, let us call that digital view. And second, where there is light, representing the camera view, let us call that camera view.

To obtain the five necessary textures, five cameras are being used, one for each texture obtained. The render-texture of each camera is converted to a texture2D object, which allows the render-textures to be saved and used for processing. Examples of textures obtained from the five cameras can be seen in Table 3.1. The diagram in Figure 4.2, shows which camera captures which view.

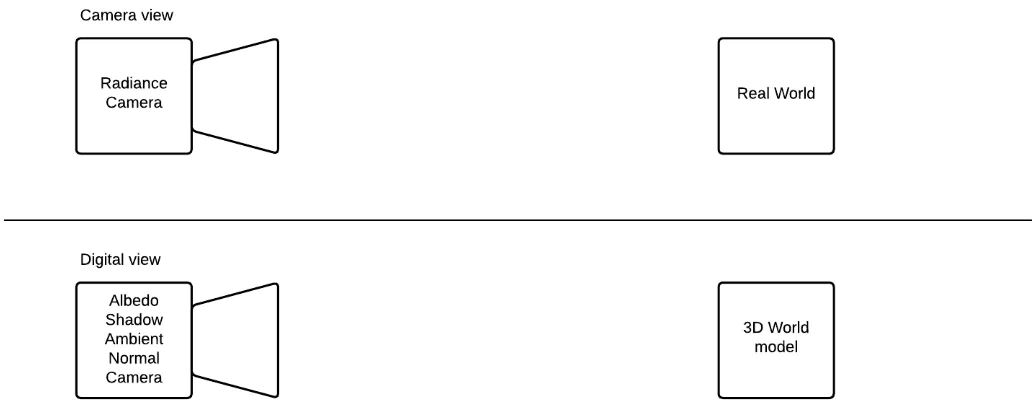


Figure 4.2: Diagram showing what camera captures which view. The radiance camera captures the real world, while the albedo, shadow, ambient, and normal camera captures the digital view.

The radiance texture is simply a rendering of the camera view with synthetic lighting. For synthetic lighting, a directional light is inserted to represent the sun, while multiple directional lights are used to represent the hemisphere. With enough directional light in the hemisphere, a smooth soft shadow can be created. An example can be seen in Figure 4.3. This method is also used by Bertolini et. al, [16], where it was discovered that there were no significant drop in FPS, before 20 directional lights were inserted into the scene.

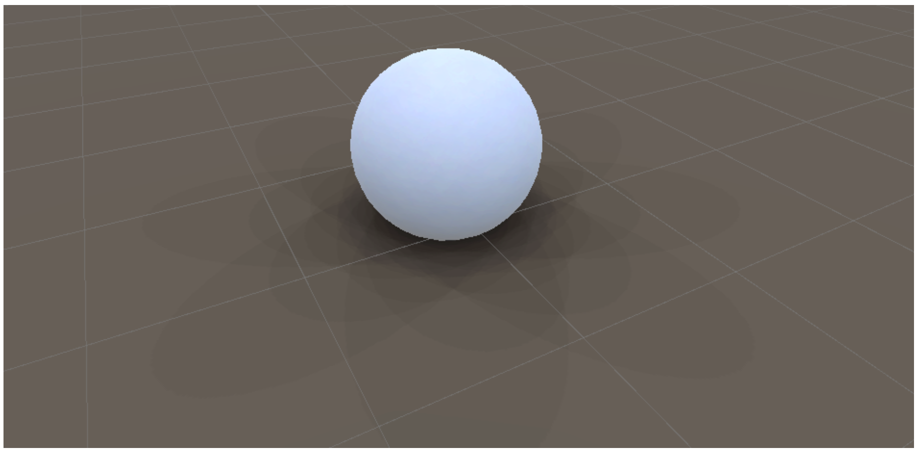


Figure 4.3: Hemisphere lighting emulated in the Unity game engine. The renderer utilises multiple directional light sources, all pointing towards the center of the world space. This results in the light appearing equally bright from all directions, just as the light from the hemisphere.

The albedo texture, like the radiance texture, is simply obtained by rendering the scene, but without the synthetic lighting. This is done by rendering the digital view instead of the camera view. However, all objects in the digital view have a simple custom shader, which takes a texture and applies it to a 3D model without adding light. This ensures that no light creates shadows or alters the color of the texture.

The shadow texture is simply the sun's shadow map when it casts light onto the digital view. The shadow map is then transformed into the view space. To obtain the shadow map in view space, a command buffer was written for the sunlight. The command buffer can be seen in Code snippet 4.1. This command buffer takes the shadow map of the light and sets the shadow map as a global texture. This texture can then be accessed in any shader script, by referring to the texture name provided as the first argument in the SetGlobalTexture method.

```
light = GetComponent<Light>();
if (light)
{
    cb = new CommandBuffer();
    cb.name = "CopyShadowMap";
    cb.SetGlobalTexture("_DirectionalShadowMask", new
 ↪  RenderTargetIdentifier(BuiltinRenderTextureType.CurrentActive));
    light.AddCommandBuffer(UnityEngine.Rendering.LightEvent.AfterScreenspaceMask, cb);
}
```

Code-snippet 4.1: The code that writes to the command buffer to get the shadow map of that light.

To obtain the ambient light texture, the shadow maps from the directional lights in the hemisphere light are used. These shadow maps are also rendered using the digital view. The shadow maps are obtained the same way, the shadow map of the sun is obtained. However, all lights used in the hemisphere must have a command buffer written to it. These shadow maps are divided by the amount of directional light used in the hemisphere and then layered on top of each other. This results in a light shadow where only one shadow is hitting, and a darker shadow where multiple shadows hit.

Lastly, the normal map uses the built-in Unity Normal-Depth-Texture. But because the Unity textures only work with numbers between 0 and 1. This means when the normals are applied directly to the texture all negative numbers, will be rounded to 0 and information is therefore lost. To preserve the information, the normals must be encoded into a color and decoded when used for the irradiance calculation.

To encode the normals, 1 is added to each normal and then divided by 2. This will result in -1 becoming 0, and 1 staying 1. When decoding the normals, 0.5 is subtracted from the normal and multiplied by 2. This will result in 0 becoming -1, and 1 staying 1.

Each texture is saved, so they can be used to calculate the L_a and L_d

After obtaining the textures, they can now be used to calculate L_a and L_d using Equation 3.1. To calculate the L_a and L_d a Python script was created. The Unity project was then connected through sockets to the Python script. The Python code, which calculates the irradiance values can be seen in Code snippet 4.2. The Python script would create two lists, which would contain the pixels that would be used in the calculations. The first list contains the raw pixel value, representing the left side of the equal sign in Equation 3.1. The second list contains two columns of calculated value, representing the right side of the equal sign in Equation 3.1.

This process is done by loading the five necessary textures as NumPy arrays. All five images are then looped over, by only looping over one image, but accessing corresponding pixels by array indexing. Next, each pixel is divided by 255 to make the pixel value between 0 and 1. Before adding a pixel to the list of pixels used for calculating the irradiance values, a pixel must pass a goodness test. The goodness test in this project is a simple depth test to make sure the skybox is not included in the calculation.

If the pixel passes the goodness test, the radiance pixel is added to the list of raw pixels, and the 4 other pixels are used for calculations. Firstly, the albedo pixel is multiplied by the ambient pixels and the result is then added to the first column in the second list. Next, the dot product of the normal pixel and the sunlight direction are calculated. The light direction is provided by Unity, through the socket connection. The albedo pixel is then multiplied by the sun shadow map pixel and the max value of 0 or the dot product. The result is then added to the second column in the second list.

The two lists are then converted into a NumPy $Nx1$ array and a $Nx2$. The two arrays are then multiplied using the numpy.linalg.lstsq, which results in a $2x1$ containing the L_a and L_d values.

```python
albedo = cv.resize(cv.imread(folderName + "/albedoTex." + fileType), dsize)
radiance = cv.resize(cv.imread(folderName + "/radianceTex." + fileType), dsize)
normal = cv.resize(cv.imread(folderName + "/NormalTex." + fileType), dsize)
shadow = cv.resize(cv.imread(folderName + "/ShadowTex." + fileType), dsize)
ambient = cv.resize(cv.imread(folderName + "/AmbientOcclsionTex." + fileType), dsize)
for y in range(albedo.shape[0]):
    for x in range(albedo.shape[1]):
        radianceValue = radiance[y][x] / 255
        albedoValue = albedo[y][x] / 255
        normalVector = normal[y][x] / 255
        ambientValue = ambient[y][x] / 255
        shadowValue = shadow[y][x] / 255

        # "Goodness test"
        if normalVector[0] > 0 and normalVector[1] > 0 and normalVector[2] > 0:
            ambientResult = albedoValue * ambientValue

            diffuse = np.dot((normalVector - 0.5) * 2.0,
↪ np.array([float(receivedData_list[3]), float(receivedData_list[2]),
↪ float(receivedData_list[1])]))
            if shadowValue.all() != 1:
                shadowValue = [0, 0, 0]
            directResult = albedoValue * shadowValue * max(0, diffuse)

            PixelRList.append([radianceValue[2]])
            PixelGList.append([radianceValue[1]])
            PixelBList.append([radianceValue[0]])

            ABRList.append([ambientResult[2], directResult[2]])
            ABGList.append([ambientResult[1], directResult[1]])
            ABBList.append([ambientResult[0], directResult[0]])

PixelRArray = np.asarray(PixelRList)
```

```
ABRArray = np.asarray(ABRList)

PixelGArray = np.asarray(PixelGList)
ABGArray = np.asarray(ABGList)

PixelBArray = np.asarray(PixelBList)
ABBArray = np.asarray(ABBList)

result = str(np.linalg.lstsq(ABRArray, PixelRArray)[0][0]) \
         + ":" + str(np.linalg.lstsq(ABGArray, PixelGArray)[0][0]) \
         + ":" + str(np.linalg.lstsq(ABBArray, PixelBArray)[0][0]) \
         + ":" + str(np.linalg.lstsq(ABRArray, PixelRArray)[0][1]) \
         + ":" + str(np.linalg.lstsq(ABGArray, PixelGArray)[0][1]) \
         + ":" + str(np.linalg.lstsq(ABBArray, PixelBArray)[0][1])
```

Code-snippet 4.2: The implemented code which calculates the irradiance values.

After estimating L_a and L_d they can be used in an object shader using Equation 3.2 and a shadow receiver shader using Equation 3.3, which is explained in chapter 3. The implemented object shader uses two passes. A forward base pass for handling the light from the sun, which is the $\vec{L_d} * (\vec{n}(\vec{x}) \cdot \vec{l}(\vec{x})) * Sun_a$ part of Equation 3.2, and a forward add pass for handling all additional lights in the scene, which is the $\vec{L_a} * Ambient_a$ part of Equation 3.2, and this correspond to the ambient light in the scene.

The shadow receiver is a bit trickier when implemented. Here both shadows before and after the augmented object is inserted are necessary. This therefore means, that by using the textures from when gaining the ambient textures, the shadows before the augmented object is inserted are obtained. To gain the shadows after the augmented object is inserted, the built-in shadow attenuation function was not used. Instead, new shadow maps from the hemisphere lights are created. This is due to when using the forward add pass, the shader loops through each additional light in the scene, and the order is unknown. This therefore means when using the built-in shadow attenuation function it returns a shadow map, but matching it to one of the already obtained shadow maps from the hemisphere, can be very difficult. It is therefore easier to create the new shadow maps and use those instead. With the shadow both before and after inserting an object, Equation 3.3 can now be used. Each shadow map and corresponding shadow map is multiplied by the $\frac{L_a}{l}$ and then divided with each other.

5 Evaluation

To evaluate the system four tests were performed. First, a test was performed with four different scenes, where the hemisphere light stayed the same, but the sunlight's position and colour changed over the course of the scene. This is to simulate a day cycle, with a sunrise and sunset.

In the second test, three out of the four scenes from the previous test were reused. In this test, the strength of the sunlight was changed, while the ambient light remained the same throughout the test

In the third test, the same three scenes were used. Here the hemisphere light stayed the same, and the colour of the sunlight changed. The sunlight in this test remained stationary.

The fourth test, the same three scenes was used. Here the hemisphere light changed while the sun light stayed the same throughout the test.

For an example image of the render, Figure 5.1 can be seen.

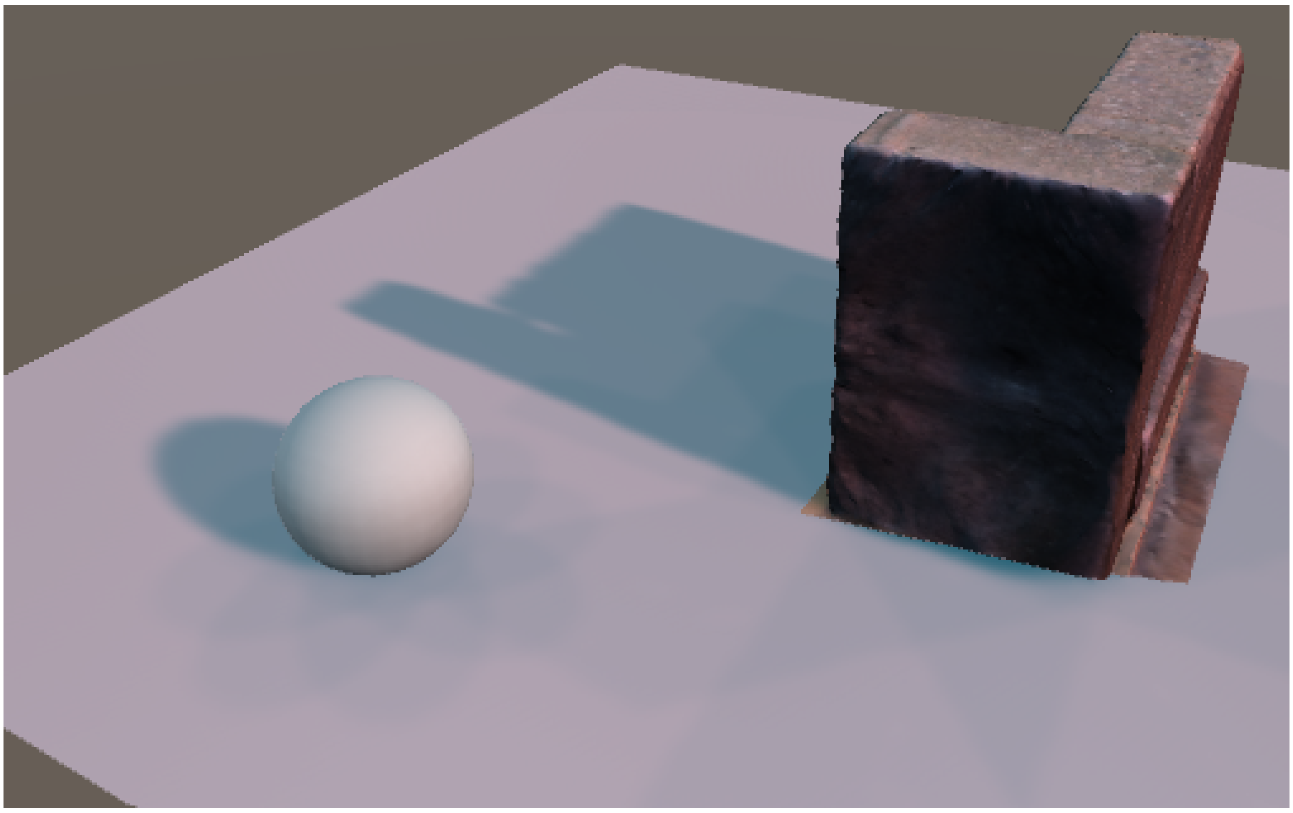


Figure 5.1: The unity scene where the sphere and the shadow are rendered using the implemented diffuse shader and shadow receiver.

5.1 Day cycle evaluation

This section covers the first evaluation, where the day cycles were used. Images of each scene and the estimated values will be covered. Hemisphere light stays the same bluish colour(RGB 186, 226, 255) throughout the day cycle, while the sunlight goes from a red colour(RGB 183, 27, 27) to a more light white-yellow colour(RGB 255, 244, 214). For an image of the colours in use, see Figure 5.2.

The hemisphere contains 9 directional lights, where the strength of each light equals $\frac{1}{l}$ where l equals the amount of light. The sunlight is a directional light with a strength of 1.

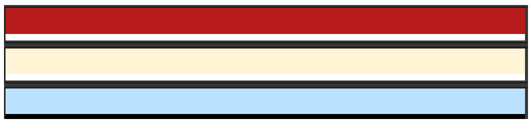


Figure 5.2: The colours used when lighting the scene. The blue colour is used for the hemisphere lights, while the red and white-yellow colours are used for the sunlight.

In this evaluation, due to the light of the hemisphere staying the same and only the sunlight changing, only the estimated L_d value should change over time, no matter the scene in use. The estimated L_d value should change due to the light colour change, and the estimated value should therefore reflect the change. The changes to L_d should be reflected by having the red channel estimated higher, when the light is red, and as the light turns more white-yellow the estimated Ld channels should get closer to each other.

For the full video of the day cycles, please refer to the provided videos or the following link `https://drive.google.com/drive/folders/1Sl3JGIUI4WZ7trNPyBjyVsT795FmqQcm?usp=sharing`.

5.1.1 Scene 1

In the first scene, a simple wall with a hole was put in the scene. The sphere and the shadow cast by it were shaded using the estimated L_a and L_d values. The scene can be seen in Figure 5.3. A graph of the estimated L_a values can be seen in Figure 5.4 and a graph of the estimated L_d values can be seen in Figure 5.5.

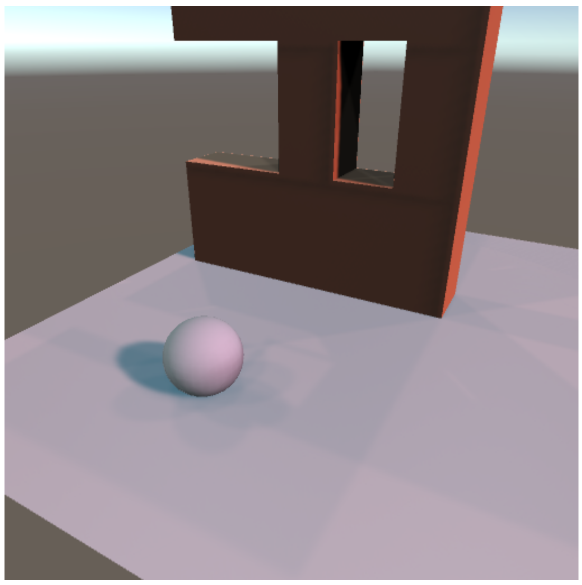


Figure 5.3: The first scene the implementation was tested on.

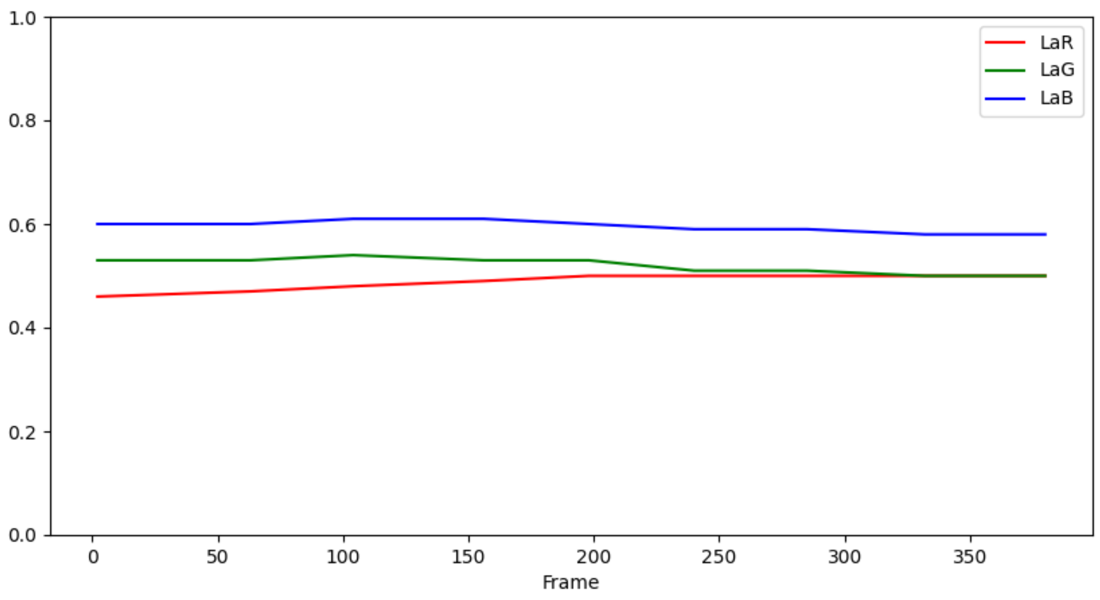


Figure 5.4: The estimated ambient lighting value over 400 frames, when scene1 was used.

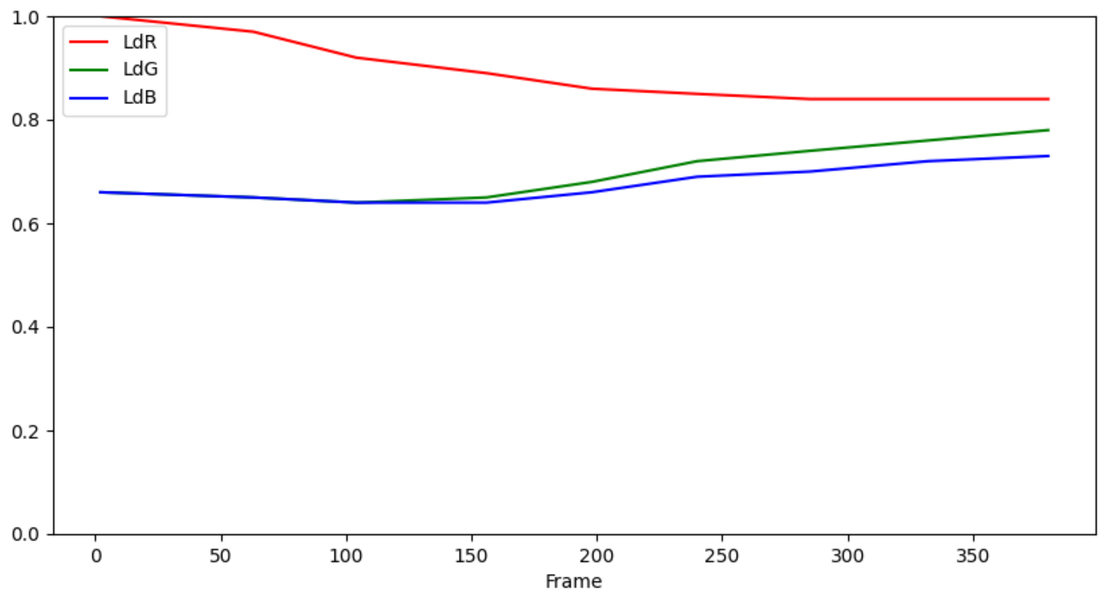


Figure 5.5: The estimated light from the sun over 400 frames, when scene1 was used.

As it can be seen in Figure 5.4, the L_a value remains relatively stable throughout the day cycle, where the difference between the maximum and minimum L_a values are 0.0457172, 0.039789, and 0.02541. It can also be seen, that the hemisphere is estimated to have more blue, than red and green. This is also expected due to the colour of the hemisphere being a blue light source.

As for the L_d values, they also act as expected. In Figure 5.5 the estimated values shows, that the sunlight starts out being a more reddish color, and as the day cycle goes on, the light has less of a red tint.

The maximum, minimum, and the difference between the maximum and minimum for both the L_a and the L_d values for scene 1 can be seen in Table 5.1

	La R	La G	La B	Ld R	Ld G	Ld B
Max	0.502336	0.535783	0.606686	1.004243	0.775147	0.726304
Min	0.457172	0.495994	0.581276	0.841424	0.641238	0.641714
difference	0.045164	0,039789	0.02541	0,162819	0,133909	0,08459

Table 5.1: Table containing the maximum, minimum L_a and L_d values and the difference between the maximum and minimum for scene 1.

5.1.2 Scene 2

The second scene is also a simple scene where a statue is inserted. Again the sphere and the shadow cast by it were shaded using the estimated L_a and L_d values. The scene can be seen in

Figure 5.6. A graph of the estimated L_a values can be seen in Figure 5.7 and a graph of the estimated L_d values can be seen in Figure 5.8.

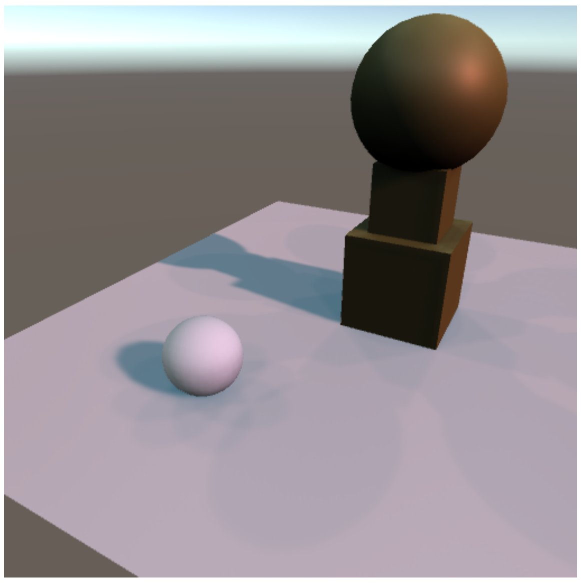


Figure 5.6: The first scene the implementation was tested on.

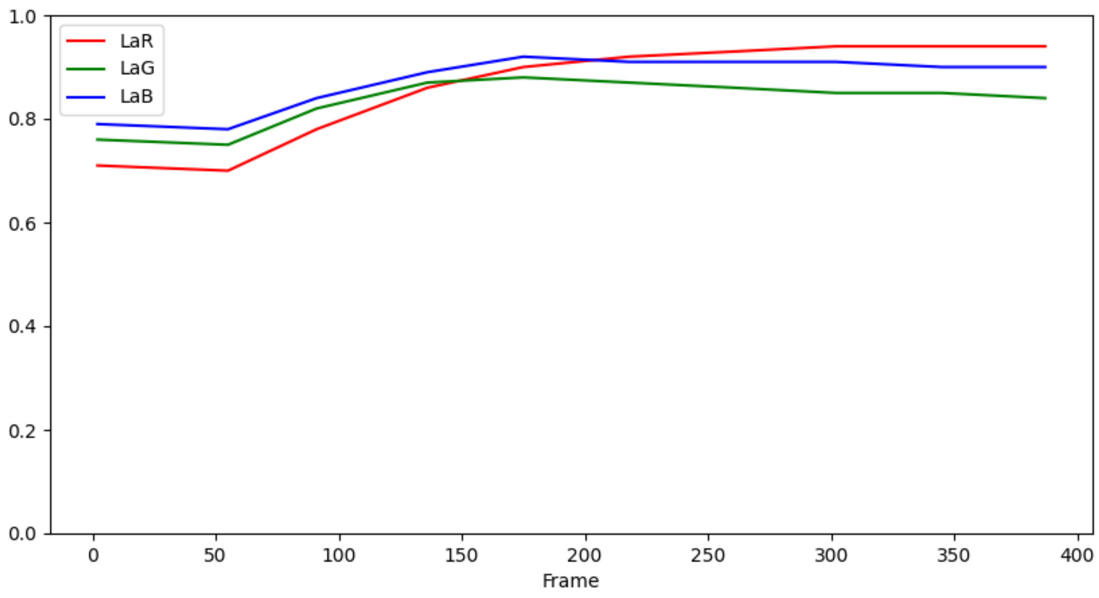


Figure 5.7: The estimated ambient lighting value over 400 frames, when scene2 was used.

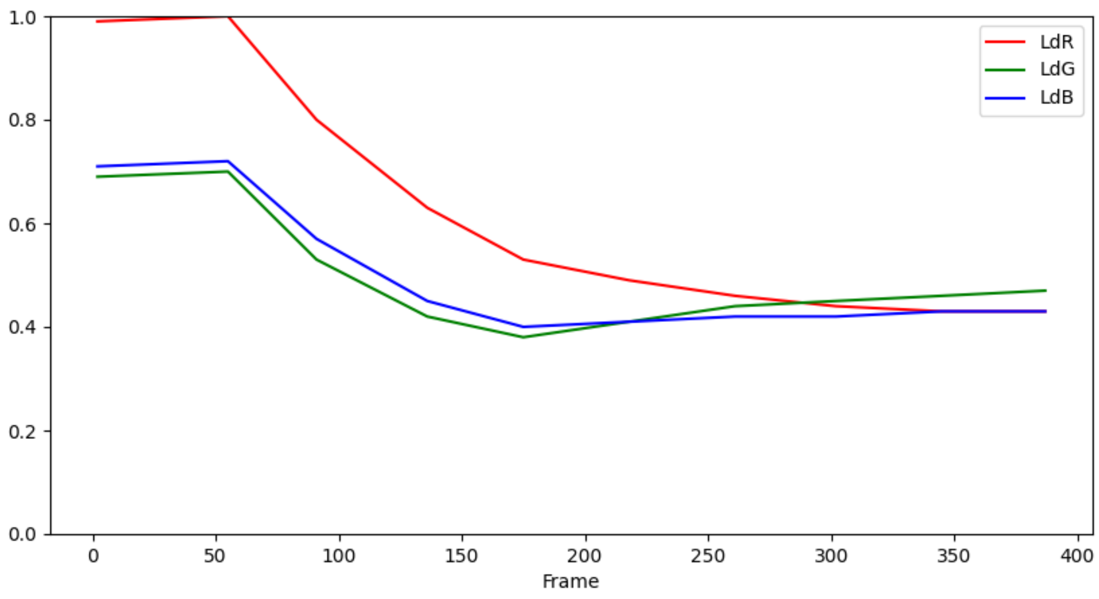


Figure 5.8: The estimated light from the sun over 400 frames, when scene2 was used.

When looking at Figure 5.7, the L_d still acts somewhat as expected. The L_d value is estimated to contain more red in the beginning, and as the light turns more white-yellow the amount of red falls and gets closer to the green and blue. However, as it can be seen the estimated blue and green colours also falls, but later stabilised towards the end of the day cycle.

The fall in L_d matches the rise in the L_a. When looking at Figure 5.8, it can be seen that the values rise about the 100 frame, which is where the L_d values start to fall. It can also be seen that the L_a value is estimated to contain more red at the end of the cycle, which should not be the case. The exact maximum, minimum, and the difference between them, can be seen in Table 5.2

Skrive lidt om hvorfor dette kan være tilfældet, når jeg har snakket/hørt lidt fra claus

	La R	La G	La B	Ld R	Ld G	Ld B
Max	0.943305	0.884905	0.915652	0.999304	0.700152	0.724198
Min	0.699046	0.751017	0.778627	0.427935	0.384828	0.402957
difference	0.244259	0.133888	0.137025	0.571369	0.315324	0.321241

Table 5.2: Table containing the maximum, minimum L_a and L_d values and the difference between the maximum and minimum for scene 2.

5.1.3 Scene 3

The third scene is also a simple scene where a wall with a small roof is added to the scene. The sphere is now placed a bit under the roof, so the roof will lower the amount of light from the hemisphere hitting the sphere. The roof will also cast a shadow on the sphere, toward the end of the day cycle. The sphere and the shadow cast are again shaded using the estimated L_a and L_d. The scene can be seen in Figure 5.9. A graph of the estimated L_a values can be seen in Figure 5.10 and a graph of the estimated L_d values can be seen in Figure 5.11.

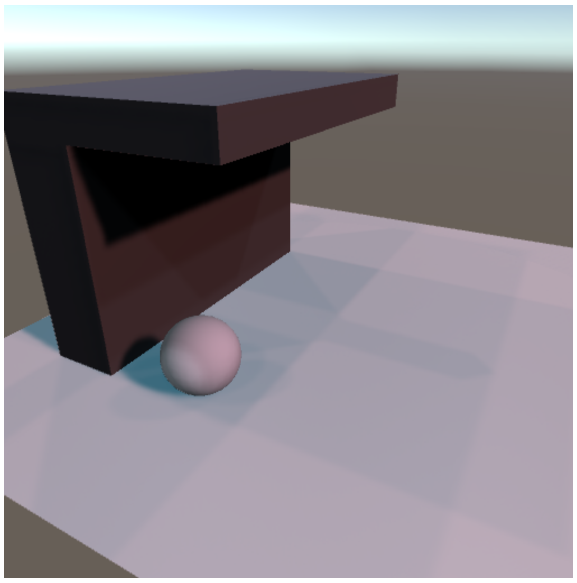


Figure 5.9: The first scene the implementation was tested on.

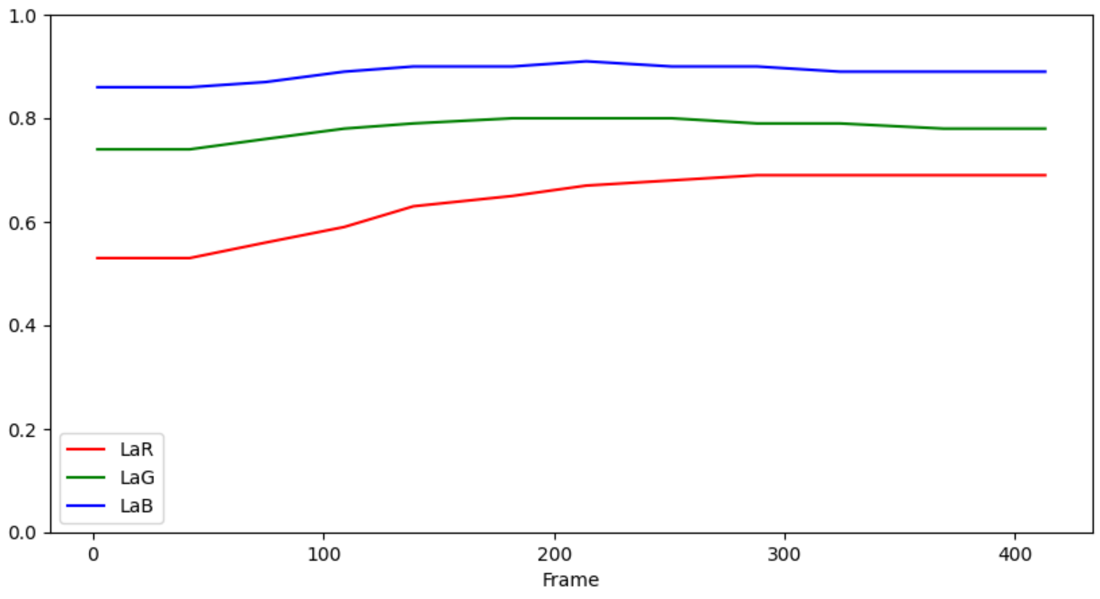


Figure 5.10: The estimated ambient lighting value over 400 frames, when scene3 was used.

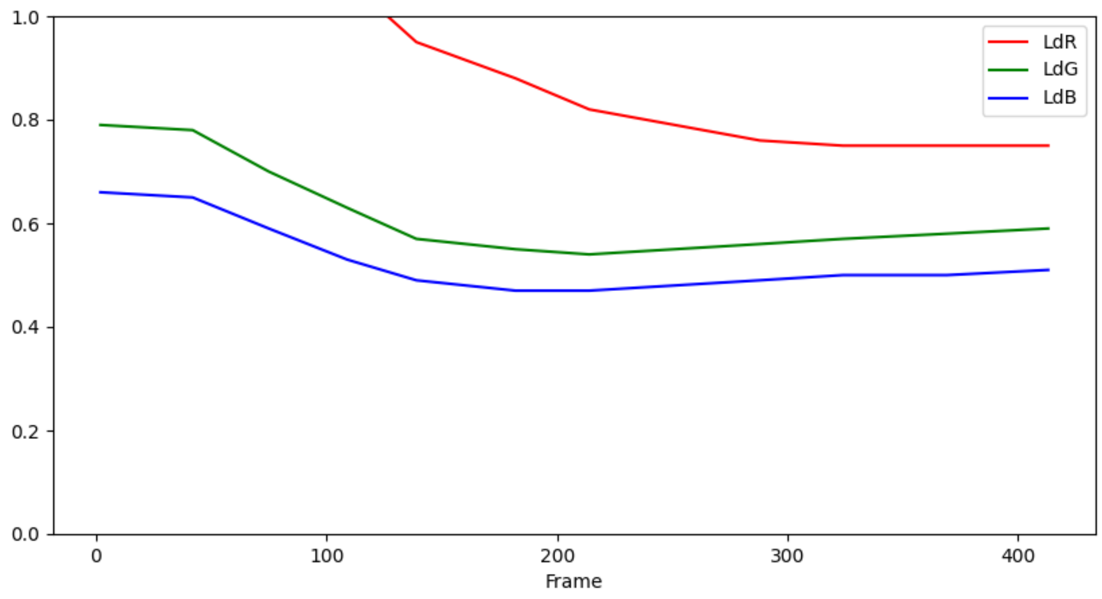


Figure 5.11: The estimated light from the sun over 400 frames, when scene3 was used.

When looking at Figure 5.10 and Figure 5.11 it can be seen the estimated values are back to acting more as expected. The hemisphere is estimated to contain most blue and less red throughout the whole day cycle. It can also be seen that the sunlight is estimated to contain more red in the beginning of the day cycle and less red at the end of the cycle. There is however still a small fall for the green and blue colour at around the 100 frame mark. It can also be seen that there is still a small rise in the red colour in the estimated L_a value. But the estimated green and blue colour remain stable throughout the day cycle. For the maximum, minimum and the difference between them, see Table 5.3.

	La R	La G	La B	Ld R	Ld G	Ld B
Max	0.693179	0.801701	0.905399	1.331048	0.788903	0.658013
Min	0.531184	0.738352	0.859704	0.745862	0.537469	0.465685
difference	0.290222	0.063349	0.045695	0.585186	0.251334	0.192328

Table 5.3: Table containing the maximum, minimum L_a and L_d values and the difference between the maximum and minimum for scene 3.

5.1.4 Brick scene

The fourth scene is another simple scene, where a 3D model of some brick has been placed in the middle of the scene. Again the sphere and the shadow cast by it were shaded using the estimated L_a and L_d values. The scene can be seen in Figure 5.12. A graph of the estimated L_a values can be seen in Figure 5.13 and a graph of the estimated L_d values can be seen in Figure 5.14.

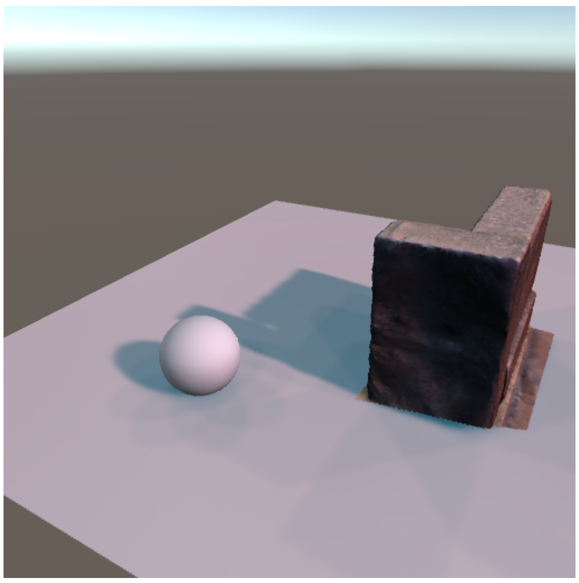


Figure 5.12: The first scene the implementation was tested on.

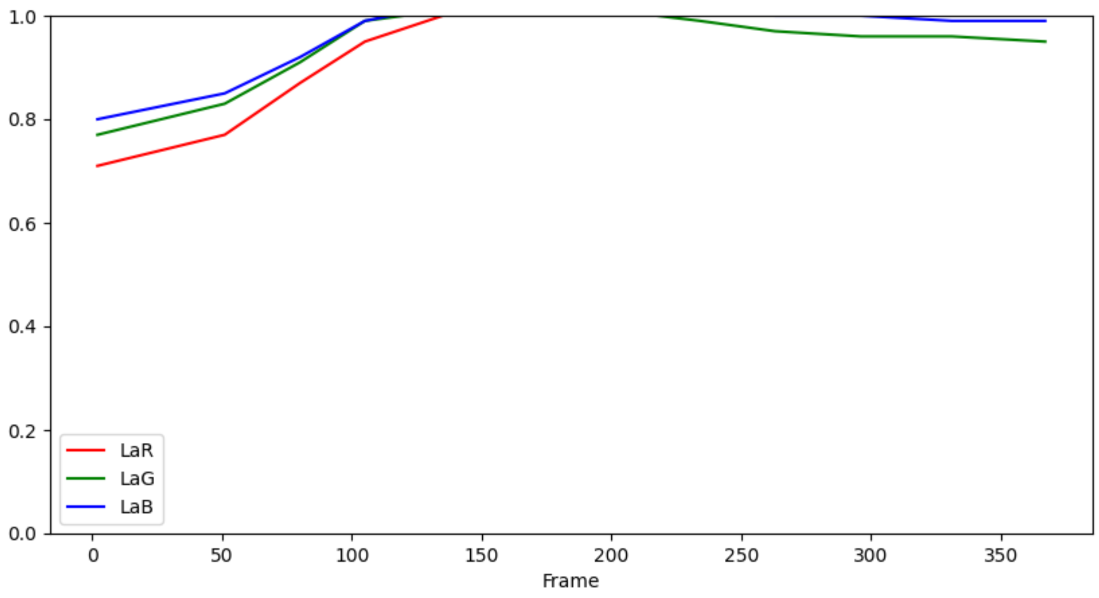


Figure 5.13: The estimated ambient lighting value over 400 frames, when the brick scene was used.

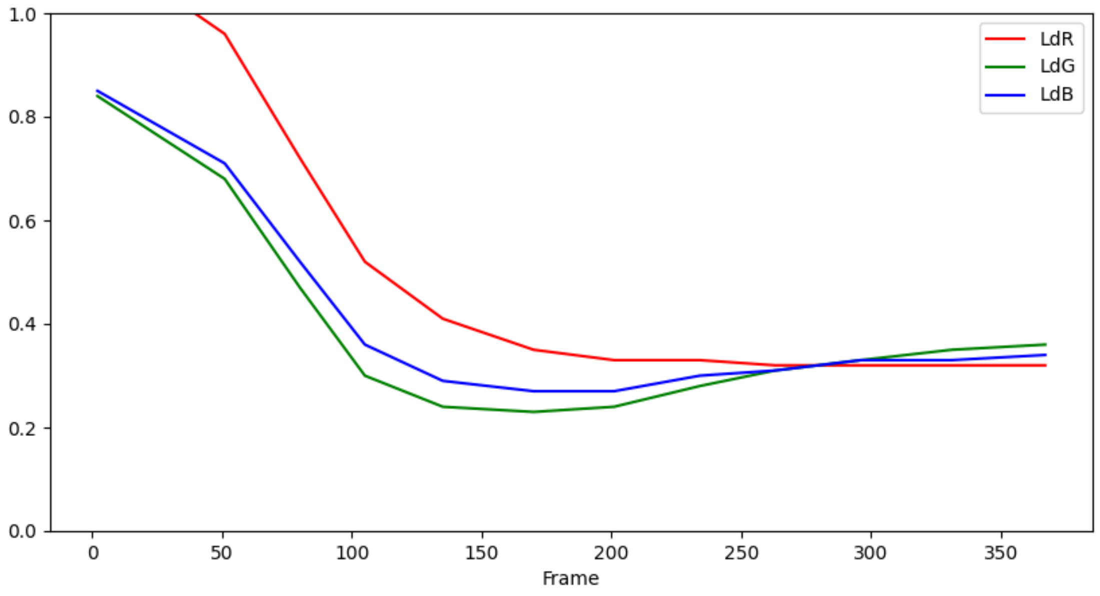


Figure 5.14: The estimated light from the sun over 400 frames, when the brick was used.

	La R	La G	La B	Ld R	Ld G	Ld B
Max	1.033038	1.016158	1.028082	1.129385	0.843282	0.849750
Min	0.711607	0.772404	0.798172	0.322417	0.227578	0.267017
difference	0.321431	0.243754	0.22991	0.806968	0.615704	0.582733

Table 5.4: Table containing the maximum, minimum L_a and L_d values and the difference between the maximum and minimum for the brick scene.

Again the estimated values are not as expected. The estimated L_d values start as expected, with the light estimated to contain more red than green and blue. But all three colours fall when hitting the 50-frame mark. It is also at the 50-frame mark, the L_a values start to rise and is also estimated to contain more red than both green and blue.

5.1.5 Overall method results

When looking overall at the result of this test, it can be seen that the system is able to estimate some scenes as expected. In Scene 1 and Scene 3, the Hemisphere lighting was estimated to contain more blue than both red and green throughout the day cycle. It should, however, be noted that the values of the hemisphere light, were estimated to be higher in Scene 3 when compared to Scene 1. This should not be the case due to the light settings in the two scenes are the same. However, when looking at the estimated L_a values in Scene 2 and Scene 4, the values are estimated at approximately the same height. The L_a values of Scene 2 and Scene 4 rise during the day cycle, which is unexpected.

In Scene 1 and Scene 3, the sunlight was estimated to contain more red in the beginning and end with less red, which was expected. This is also the case when looking at Scene 2 and Scene 4, but the green and blue colour also drops, which is unexpected. This drop in green and blue happens at the same time the L_a value increases.

5.2 Light strength evaluation

In this section, the second evaluation will be covered. This evaluation focuses on the light strength of the sun and how the strength impacts the rendering. Three of the previous four scenes are used in this test as well. The L_a value will start at (0.3, 0,35, 0.4) and remain the same throughout the test. The L_d value starts at (1, 0.85, 0.8) and will fall four times in intervals of 0.2 and therefore ends at (0.4, 0.25, 0.2). Images and the estimated values are provided in each subsection. In this test, it is expected that only the L_d value will fall and the L_a value will remain steady throughout the test.

5.2.1 Scene 1

Scene 1 of this test, is the simple scene with a wall with a hole. The sphere and the shadow cast by the sphere were shaded using the estimated L_a and L_d. Images of the scene and the estimated L_a and L_d values, with the different sunlight strengths, can be seen in Table 5.5. The estimated L_a and L_d values can also be seen in Table 5.6

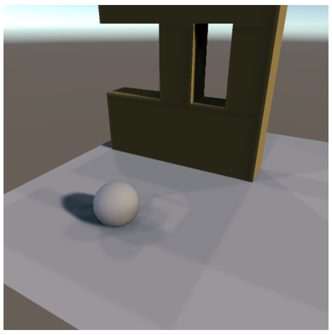
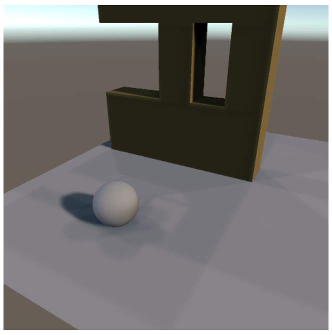
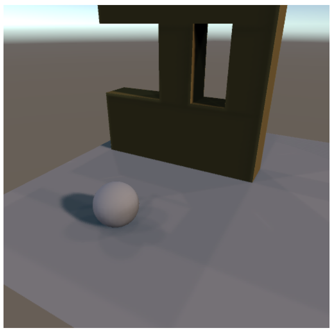
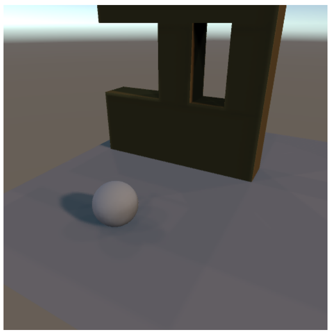
	
L_d (0.32, 0.26, 0.15)	L_d (0.27, 0.21, 0.11)
L_a (0.64, 0.66, 0.74)	L_a (0.62, 0.64, 0.71)
	
L_d (0.22, 0.16, 0.08)	L_d (0.19, 0.13, 0.052)
L_a (0.6, 0.61, 0.69)	L_a (0.57, 0.58, 0.65)

Table 5.5: Table containing images of scene 1, as the light strength fall.

	Light setting 1	Light setting 2	Light settings 3	Light settings 4
Real Ld	(1, 0.85, 0.8)	(0.8, 0.65, 0.6)	(0.6, 0.45, 0.4)	(0.4, 0.25, 0.2)
Real La	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)
Estimated Ld	(0.32, 0.26, 0.15)	(0.27, 0.21, 0.11)	(0.22, 0.16, 0.08)	(0.19, 0.13, 0.052)
Estimated La	(0.64, 0.66, 0.74)	(0.62, 0.64, 0.71)	(0.6, 0.61, 0.69)	(0.57, 0.58, 0.65)

Table 5.6: Table containing the real L_a and L_d values and the estimated L_a and L_d values for scene 1

5.2.2 Scene 2

Scene 2 are the scene where the statue is inserted. The images of the scene with different light strength can be seen in Table 5.7, and the estimated and real L_a and L_d can be seen in Table 5.8

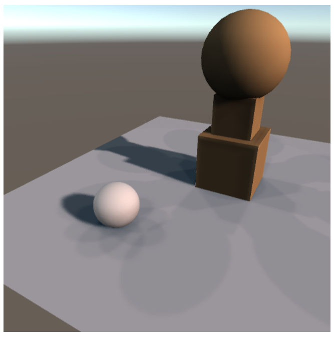
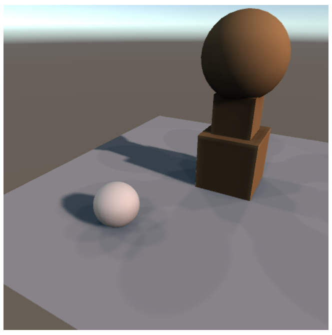
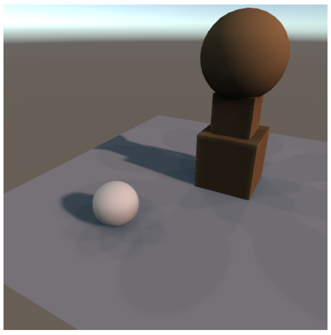
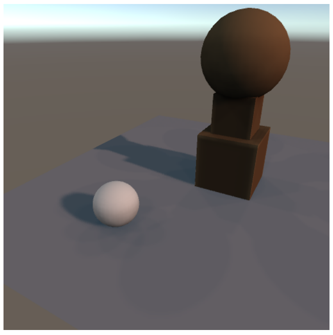
	
L_d (0.59, 0.44, 0.39) L_a (0.70, 0.70, 0.71)	L_d (0.52, 0.38, 0.33) L_a (0.69, 0.69, 0.70)
	
L_d (0.44, 0.31, 0.26) L_a (0.68, 0.67, 0.68)	L_d (0.37, 0.25, 0.20) L_a (0.66, 0.66, 0.67)

Table 5.7: Table containing images of scene 2, as the light strength fall.

	Light setting 1	Light setting 2	Light settings 3	Light settings 4
Real Ld	(1, 0.85, 0.8)	(0.8, 0.65, 0.6)	(0.6, 0.45, 0.4)	(0.4, 0.25, 0.2)
Real La	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)
Estimated Ld	(0.59, 0.44, 0.39)	(0.52, 0.38, 0.33)	(0.44, 0.31, 0.26)	(0.37, 0.25, 0.20)
Estimated La	(0.70, 0.70, 0.71)	(0.69, 0.69, 0.70)	(0.68, 0.67, 0.68)	(0.66, 0.66, 0.67)

Table 5.8: Table containing the real L_a and L_d values and the estimated L_a and L_d values for scene 2

5.2.3 Scene 3

The last scene is the scene with a wall with a roof on, that covers the inserted sphere. Images of the scene can be seen in Table 5.9 and the the estimated and real L_a and L_d values can be seen in Table 5.10.

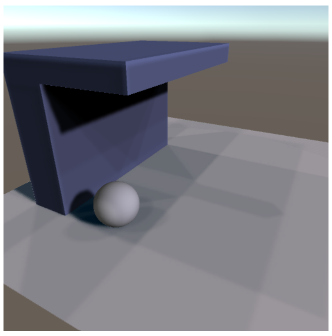
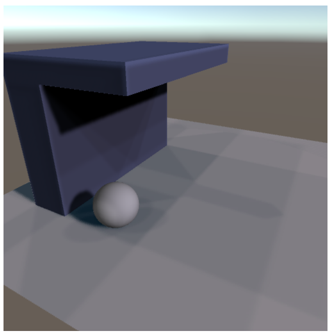
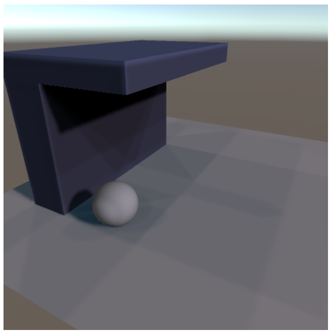
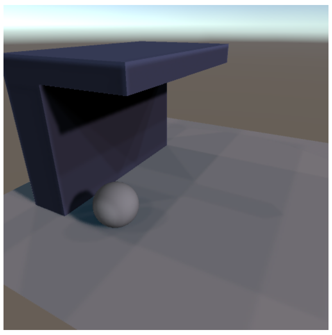
 L_d (0.60, 0.59, 0.67) L_a (0.50, 0.50, 0.49)	 L_d (0.54, 0.52, 0.58) L_a (0.49, 0.49, 0.49)
 L_d (0.47, 0.45, 0.50) L_a (0.48, 0.48, 0.49)	 L_d (0.41, 0.39, 0.42) L_a (0.47, 0.47, 0.49)

Table 5.9: Table containing images of scene 3, as the light strength fall.

	Light setting 1	Light setting 2	Light settings 3	Light settings 4
Real Ld	(1, 0.85, 0.8)	(0.8, 0.65, 0.6)	(0.6, 0.45, 0.4)	(0.4, 0.25, 0.2)
Real La	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)
Estimated Ld	(0.60, 0.59, 0.67)	(0.54, 0.52, 0.58)	(0.47, 0.45, 0.50)	(0.41, 0.39, 0.42)
Estimated La	(0.50, 0.50, 0.49)	(0.49, 0.49, 0.49)	(0.48, 0.48, 0.49)	(0.47, 0.47, 0.49)

Table 5.10: Table containing the real L_a and L_d values and the estimated L_a and L_d values for scene 3

5.2.4 Overall method results

When looking over the results of the tests, the estimated values do act somewhat as expected. The estimated L_a remains almost steady throughout all three tests, which is expected because the hemisphere light did not change. The estimated L_d fell throughout all three tests, which is also expected. But when comparing the estimated L_a and L_d values with the real L_a and L_d, they are a lot different. In scene 1 and 2 the L_a value was estimated to be the strongest light contributor, while the L_d value is the actual strongest light contributor in the scene. The L_a is also estimated to be a lot stronger than what the real L_a value is. And the L_d value is also estimated to being lower than what the real L_d is. However to determine whether the estimated L_a and L_d values is fitting to the scene, the images in Table 5.5, Table 5.7, and Table 5.9 should be taken into consideration. When looking at scene 1 and 3, the sphere seems to blend in quite well. However, looking at scene 2 the sphere seems to look bright when compared to the rest of the scene.

5.3 Sunlight Colour change

In this section, the third test will be covered. This test focuses on the impact of sunlight colour change. For this three scenes were used and during each scene, the sunlight would change colour 3 times. This means that L_a and L_d will be estimated 4 time. The position of the sunlight will remain the same and the L_a will also remain the same at (0.3, 0.35, 0.4). In this test, it is expected that only the L_d value is changing over time, due to the sunlight changing and not the hemisphere. This therefore also means, that the estimated L_a should remain steady throughout the tests.

5.3.1 Scene 1

The first scene is the wall with holes in it. Images of the scene can be seen in Table 5.11 and the estimated L_a and L_d and the real L_a and L_d values can be seen in Table 5.12.

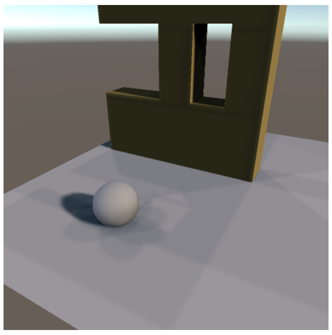
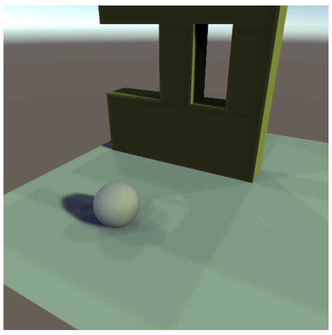
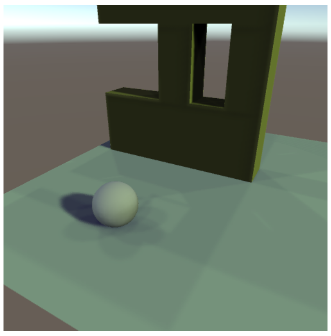
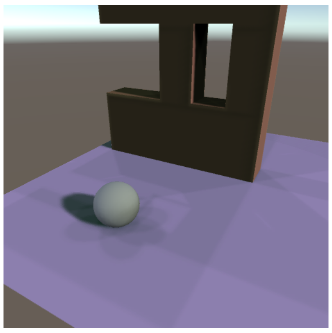
	
L_d 0.32, 0.26, 0.15	L_d 0.27, 0.29, 0.12
L_a 0.64, 0.66, 0.74	L_a 0.62, 0.68, 0.71
	
L_d 0.23, 0.27, 0.09	L_d 0.23, 0.23, 0.13
L_a 0.6, 0.67, 0.69	L_a 0.6, 0.65, 0.71

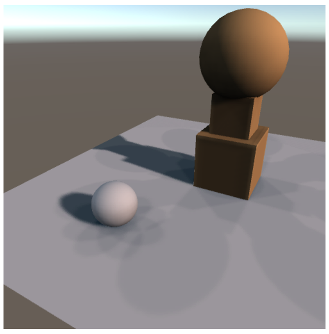
Table 5.11: Table containing images of scene 1, as the sun light colour changes.

	Light setting 1	Light setting 2	Light settings 3	Light settings 4
Real Ld	(1, 0.85, 0.8)	(0.8, 1, 0.6)	(0.6, 0.8, 0.45)	(0.85, 0.5, 1)
Real La	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)
Estimated Ld	(0.32, 0.26, 0.15)	(0.27, 0.29, 0.12)	(0.23, 0.27, 0.09)	(0.23, 0.23, 0.13)
Estimated La	(0.64, 0.66, 0.74)	(0.62, 0.68, 0.71)	(0.6, 0.67, 0.69)	(0.6, 0.65, 0.71)

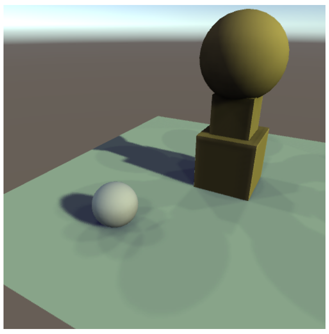
Table 5.12: Table containing the real L_a and L_d values and the estimated L_a and L_d values for scene 1

5.3.2 Scene 2

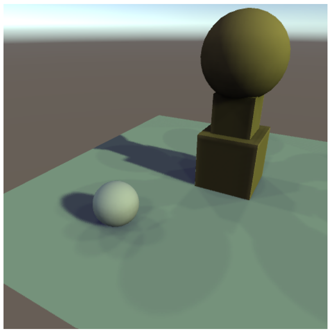
The second scene is a simple plane with a statue on it. Images of the scene can be seen in Table 5.13 and the estimated L_a and L_d and the real L_a and L_d values can be seen in Table 5.14



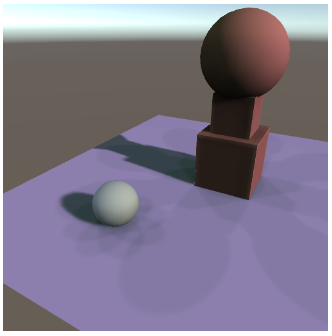
L_d 0.56, 0.42, 0.35
L_a 0.52, 0.56, 0.62



L_d 0.49, 0.47, 0.29
L_a 0.50, 0.57, 0.61



L_d 0.42, 0.44, 0.24
L_a 0.49, 0.57, 0.59



L_d 0.43, 0.39, 0.29
L_a 0.49, 0.56, 0.61

Table 5.13: Table containing images of scene 2, as the sun light colour changes.

	Light setting 1	Light setting 2	Light settings 3	Light settings 4
Real Ld	(1, 0.85, 0.8)	(0.8, 1, 0.6)	(0.6, 0.8, 0.45)	(0.85, 0.5, 1)
Real La	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)
Estimated Ld	(0.56, 0.42, 0.35)	(0.49, 0.47, 0.29)	(0.42, 0.44, 0.24)	(0.43, 0.39, 0.29)
Estimated La	(0.52, 0.56, 0.62)	(0.50, 0.57, 0.61)	(0.49, 0.57, 0.59)	(0.49, 0.56, 0.61)

Table 5.14: Table containing the real L_a and L_d values and the estimated L_a and L_d values for scene 2

5.3.3 Scene 3

The last scene is a simple plane, with a wall with a roof on it. This roof can cast a shadow on the sphere. Images of the scene can be seen in Table 5.15 and the estimated L_a and L_d and the real L_a and L_d values can be seen in Table 5.16

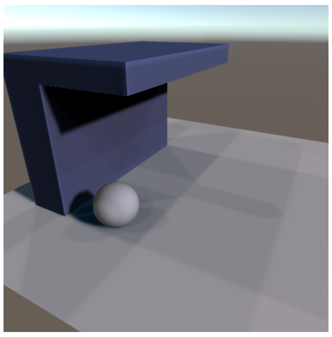
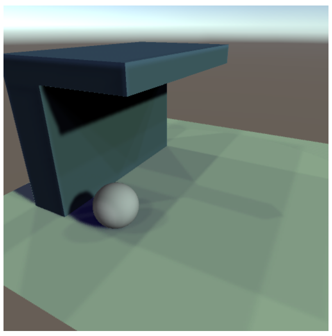
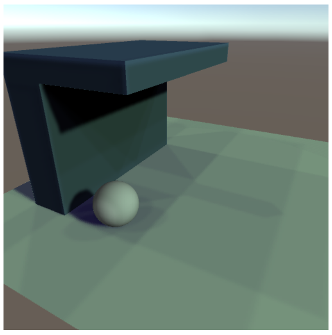
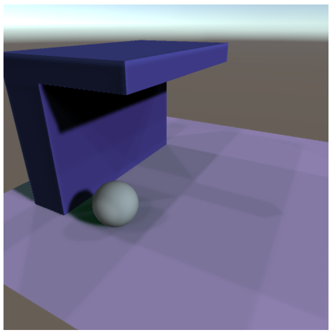
 L_d 0.6, 0.59, 0.67 L_a 0.5, 0.5, 0.49	 L_d 0.54, 0.63, 0.58 L_a 0.49, 0.51, 0.49
 L_d 0.48, 0.61, 0.51 L_a 0.48, 0.5, 0.49	 L_d 0.49, 0.56, 0.59 L_a 0.48, 0.5, 0.5

Table 5.15: Table containing images of scene 3, as the sun light colour changes.

	Light setting 1	Light setting 2	Light settings 3	Light settings 4
Real Ld	(1, 0.85, 0.8)	(0.8, 1, 0.6)	(0.6, 0.8, 0.45)	(0.85, 0.5, 1)
Real La	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)	(0.3, 0.35, 0.4)
Estimated Ld	(0.6, 0.59, 0.67)	(0.54, 0.63, 0.58)	(0.48, 0.61, 0.51)	(0.49, 0.56, 0.59)
Estimated La	(0.5, 0.5, 0.49)	(0.49, 0.51, 0.49)	(0.48, 0.5, 0.49)	(0.48, 0.5, 0.5)

Table 5.16: Table containing the real L_a and L_d values and the estimated L_a and L_d values for scene 3

5.3.4 Overall method results

When testing for sunlight colour change, the system performed mediocre. In the first three images in Table 5.11, Table 5.13, and Table 5.15 the sphere blends fairly well into the scene, but in the fourth image the sphere no longer fits into the scene. The light is estimated to be too green for the scene. When looking at the raw estimated values, the values are again wrongly estimated. The L_a and L_d values are estimated to be almost equal in contributing light to the scene, which is not the case. However, the estimated L_a value did remain somewhat steady throughout all three tests.

5.4 Hemisphere light Colour change

In this test, the system was tested on scenes where the L_a value changed and the L_d stayed the same throughout the test at (1, 0.85, 0.8). The L_a starts at (0.3, 0.35, 0.4) and changes 3 times. Images of the scenes along with tables of the estimated and real L_a and L_d can be seen in the corresponding subsections. therefore in this test, it is expected the L_d value stays the same throughout the test, and only the L_a changes.

5.4.1 Scene 1

The first scene is a simple plane with a wall on it, and the wall contains holes. Images of the scene can be seen in Table 5.17 and the estimated L_a and L_d and the real L_a and L_d values can be seen in Table 5.18.

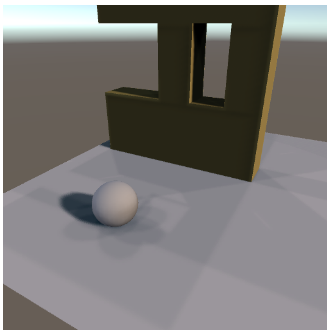
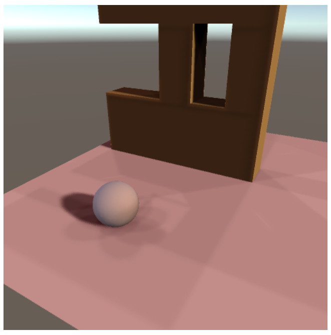
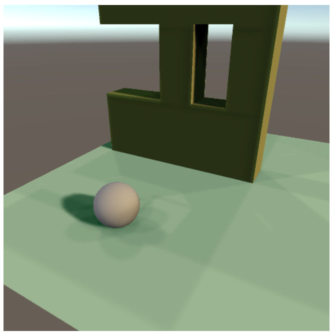
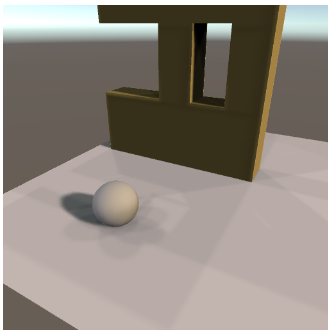
	
L_d 0.32, 0.26, 0.15	L_d 0.39, 0.24, 0.10
L_a 0.64, 0.66, 0.74	L_a 0.72, 0.64, 0.71
	
L_d 0.37, 0.28, 0.10	L_d 0.39, 0.3, 0.14
L_a 0.69, 0.68, 0.71	L_a 0.72, 0.71, 0.74

Table 5.17: Table containing images of scene 1, as the hemisphere light colour changes.

	Light setting 1	Light setting 2	Light settings 3	Light settings 4
Real Ld	(1, 0.85, 0.8)	(1, 0.85, 0.8)	(1, 0.85, 0.8)	(1, 0.85, 0.8)
Real La	(0.3, 0.35, 0.4)	(0.3, 0.3, 0.3)	(0.3, 0.5, 0.35)	(0.5, 0.5, 0.5)
Estimated Ld	(0.32, 0.26, 0.15)	(0.39, 0.24, 0.10)	(0.37, 0.28, 0.10)	(0.39, 0.3, 0.14)
Estimated La	(0.64, 0.66, 0.74)	(0.72, 0.64, 0.715)	(0.69, 0.68, 0.71)	(0.72, 0.71, 0.74)

Table 5.18: Table containing the real L_a and L_d values and the estimated L_a and L_d values for scene 1

5.4.2 Scene 2

The second scene is a simple plane with a statue on it. Images of the scene can be seen in Table 5.19 and the estimated L_a and L_d and the real L_a and L_d values can be seen in Table 5.20.

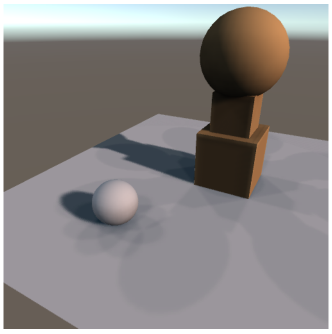
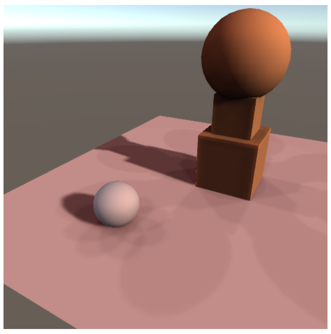
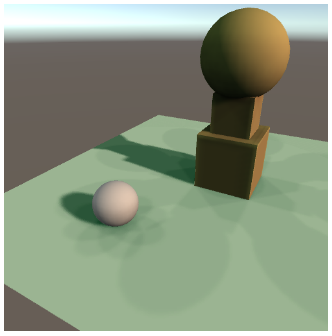
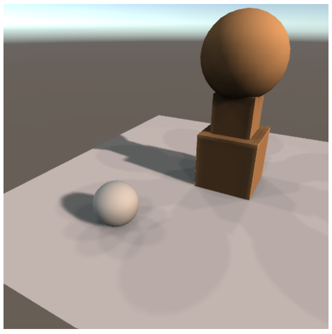
	
L_d 0.56, 0.42, 0.35 L_a 0.52, 0.56, 0.62	L_d 0.59, 0.41, 0.32 L_a 0.60, 0.54, 0.58
	
L_d 0.58, 0.43, 0.33 L_a 0.57, 0.59, 0.58	L_d 0.59, 0.44, 0.34 L_a 0.60, 0.62, 0.62

Table 5.19: Table containing images of scene 2, as the hemisphere light colour changes.

	Light setting 1	Light setting 2	Light settings 3	Light settings 4
Real Ld	(1, 0.85, 0.8)	(1, 0.85, 0.8)	(1, 0.85, 0.8)	(1, 0.85, 0.8)
Real La	(0.3, 0.35, 0.4)	(0.3, 0.3, 0.3)	(0.3, 0.5, 0.35)	(0.5, 0.5, 0.5)
Estimated Ld	(0.56, 0.42, 0.35)	(0.59, 0.41, 0.32)	(0.58, 0.43, 0.33)	(0.59, 0.44, 0.34)
Estimated La	(0.52, 0.56, 0.62)	(0.60, 0.54, 0.58)	(0.57, 0.59, 0.58)	(0.60, 0.62, 0.62)

Table 5.20: Table containing the real L_a and L_d values and the estimated L_a and L_d values for scene 2

5.4.3 Scene 3

The last scene is a plane with a wall and roof, that can cast shadow onto the sphere. Images of the scene can be seen in Table 5.21 and the estimated L_a and L_d and the real L_a and L_d values can be seen in Table 5.22.

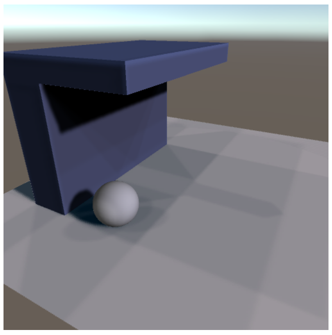
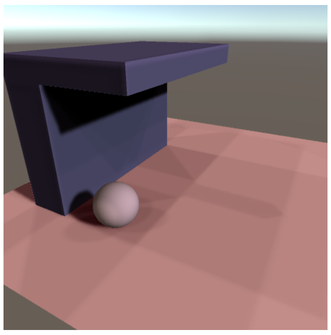
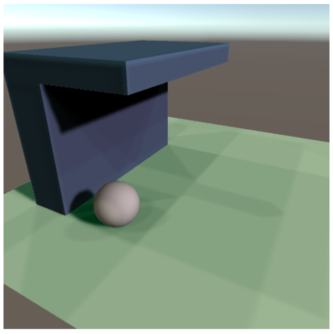
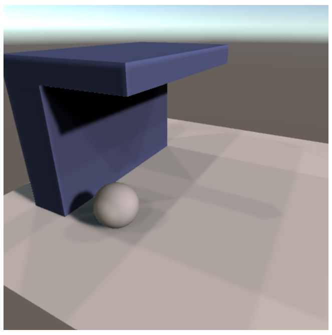
 L_d 0.6, 0.59, 0.67 L_a 0.5, 0.5, 0.49	 L_d 0.67, 0.57, 0.65 L_a 0.59, 0.48, 0.44
 L_d 0.64, 0.6, 0.65 L_a 0.56, 0.53, 0.45	 L_d 0.67, 0.62, 0.66 L_a 0.59, 0.56, 0.48

Table 5.21: Table containing images of scene 3, as the hemisphere light colour changes.

	Light setting 1	Light setting 2	Light settings 3	Light settings 4
Real Ld	(1, 0.85, 0.8)	(1, 0.85, 0.8)	(1, 0.85, 0.8)	(1, 0.85, 0.8)
Real La	(0.3, 0.35, 0.4)	(0.3, 0.3, 0.3)	(0.3, 0.5, 0.35)	(0.5, 0.5, 0.5)
Estimated Ld	(0.6, 0.59, 0.67)	(0.67, 0.57, 0.65)	(0.64, 0.6, 0.65)	(0.67, 0.62, 0.66)
Estimated La	(0.5, 0.5, 0.49)	(0.59, 0.48, 0.44)	(0.56, 0.53, 0.45)	(0.59, 0.56, 0.48)

Table 5.22: Table containing the real L_a and L_d values and the estimated L_a and L_d values for scene 3

5.4.4 Overall method results

During the test, the system managed to make the sphere blend into the scene, with the lighting in images 1, 2, and 4 in Table 5.17, Table 5.19, and Table 5.21. However, in the third image in Table 5.17, Table 5.19, and Table 5.21 the sphere seems to gain more of a yellow tint, instead of a green is tint. When comparing the estimated L_a and L_d values with the real L_a and L_d values, the estimated values are again wrong. But when looking at the estimated L_d, it remained steady in all three tests, and the estimated L_a changes through the tests.

5.5 System evaluation

When running the system, some issues came up regarding the performance. Firstly the system showed to have high memory usage, which led to system crashes if the system ran for too long. This is due to the textures taking up memory, and if these textures are not properly handled, they can end up taking up a lot of memory and even crash the system. The system also ran with a low FPS of about 4-5, if every frame is saved as textures. However, if the texture was only saved every 1 second the FPS would remain at about 27-45 FPS. This is of cause to be expected, due to processing power can now be used at rendering, instead of saving the textures.

6 Discussion

The system was implemented in two parts. One part was developed in Unity, which collected textures and rendered the graphics. And the second part was developed in Python, which calculated the lighting, using the provided textures. This led to having data being sent back and forth using either a socket connection, API calls, or other types of connection. This could introduce a delay between the lighting updates, depending on how strong the connection is and the amount of data sent. However, it could be argued that because the weather conditions do not change from one extreme to another extreme in an instance, the delay could be okay as long the delay is not too big.

When testing the system, there was a frame freeze, when textures were saved so the Python script could use the textures, which is not optimal. Therefore, other methods for the Python script to obtain the textures should be developed.

One method could be to send the textures through the socket connection, instead of saved in a directory and Python obtained them from the directory. This would result in the step of saving textures being unnecessary and the frame freeze would disappear.

Another method could be to implement the lighting calculation in the unity project. This would result in a direct connection between the script collecting textures and the one doing the lighting calculation. This way it is not necessary to save the textures. It would also result in connecting to the Python program unnecessarily and therefore the delay that could be created by the connection would disappear.

The method used to obtain the necessary textures, resulted in high memory usage. This could be due to improper handling of textures and improper clean-up of already used, and therefore not necessary textures. This led to crashes of the system if it ran for too long. Therefore it is necessary to further develop the way textures are obtained in Unity before this method can be functional on a mobile device.

Besides the high memory usage, the FPS was also low if every frame was used for light estimation. However, when only every 1 second the textures were updated, the frame rate raised again. This is due to processing power can be used for rendering instead of collecting the textures. This, therefore, means it needs to be discussed if collecting every single frame is necessary, or if updating the light once a second is enough. It is also possible to use Unity coroutines when collecting the textures. Using coroutines will not allow for usage of every single frame, but it will allow for highest amount of frames to be used, without the collecting impacting the FPS. This is due to coroutines allow code to run across multiple frames, instead of blocking a frame until the code have executed. But as stated before, the weather do not change from one instance to another, so only updating the textures every 1 second could potentially be enough.

Besides the high memory usage and low FPS, the socket connection potentially also gave issues. Sometimes the inserted object would not receive the updated irradiance value, and therefore looked the same, even when the light had changed. But when taking the values from the Python program, and giving them to Unity manually, the inserted object blended into the scene better. An example can be seen in Table 6.1.

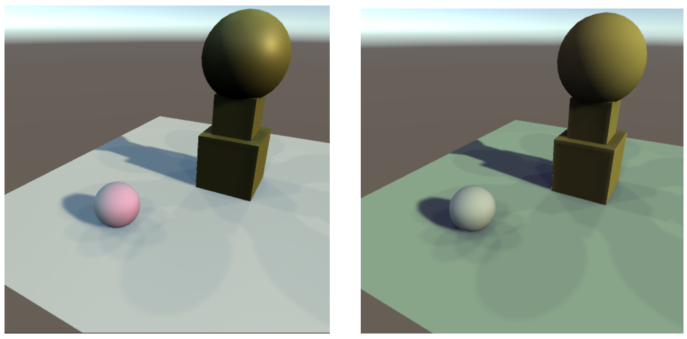


Table 6.1: Table containing a image, where the render on the left have received irradiance values through the socket connection, while the right render was manually given to the render.

To create the ambient lighting, 9 directional lights were used. This was due to the limitation of obtaining texture of the shadow maps if more than 9 directional lights were used. This resulted in very prominent edges which do not look very pleasant when used as ambient light. Therefore, if this method is used for hemisphere lighting, more directional lights should be used. This would make the shadow appear smoother and therefore have less prominent edges. However, this method would lower the performance of the program, due to the additional shadow map rendering. Bertolini el. at. [16] saw a significant decrease in frame per second when using 20 directional lights or above. Therefore, it should be considered if performance is more important than the accuracy of the shadows.

When comparing this method with the method presented by both Bertolini et. al. [16] and Wei et. al. [8], this project differs by using both the camera view and a digital view to gain information about the light, instead of only using the camera view. When only working with the camera view, it results in only the augmented object will add shadows to the shadow maps. But when working with both the digital view and camera view, the models in the digital view will also add to shadow maps. This will lead to two shadow maps having to be rendered for each directional light in the hemisphere. One shadow map without the inserted object, and another with the inserted object. This leads to this method only being able to use half the number of directional lights as Bertolini et. al. can, for the same amount of processing power.

7 Conclusion

In this project, a system to estimate the light contribution was implemented. The system was developed using the Unity game engine for computer graphics rendering and Python to calculate the lighting of a scene.

Solutions for estimating light were researched, to establish already light estimation solutions. Here both the area of neural network and none AI solutions were researched, to find the weaknesses and strengths of each area. The implemented system used the Unity game engine to create scenes and obtain textures of those scenes, while the Python program received the textures and estimated the lighting, based on the lighting model presented in chapter 3.

When evaluating it was discovered that when obtaining every frame for light estimation resulted in a low frame rate. But when only collecting texture once a second, the frame rate raised back up. Also due to improper texture handling and clean op, the system had a high memory usage which caused system crashes if the system ran for too long.

Besides the high memory usage, the socket connection implemented to connect the Python program to the Unity project also gave issues. When sending the estimated lighting values through the socket connection resulted in a different result, than when the values were given manually to the Unity program. This could indicate a bug i present in the system.

Overall the implemented system performed mediocre, where with some light settings, the object shaded using the estimated lighting blended in. While with other light settings the object did not blend in. Therefore, the system should be further developed, before it can be put into use

Bibliography

[1] Richard Skarbez *et al.* “Revisiting Milgram and Kishino’s Reality-Virtuality Continuum”. In: *Frontiers in Virtual Reality* 2 (2021). ISSN: 2673-4192. DOI: 10.3389/frvir.2021.647997. URL: https://www.frontiersin.org/articles/10.3389/frvir.2021.647997.

[2] DUAN GAO *et al.* “Deep Inverse Rendering for High-Resolution SVBRDF Estimation from an Arbitrary Number of Images”. In: *ACM Trans. Graph.* 38.4 (July 2019). ISSN: 0730-0301. DOI: 10.1145/3306346.3323042. URL: https://doi.org/10.1145/3306346.3323042.

[3] Claus Brøndgaard Madsen. “Challenges of Visually Realistic Augmented Reality”. English. In: *Proceedings: GRAPP 2020 - 15th International Conference on Computer Graphics Theory and Applications*. Ed. by Kadi Bouatouch *et al.* Vol. 1. Position paper; GRAPP 2020 - 15th International Conference on Computer Graphics Theory and Applications, GRAPP 2020 ; Conference date: 27-02-2020 Through 29-02-2020. Institute for Systems, Technologies of Information, Control, and Communication, Feb. 2020, pp. 376–383. DOI: 10.5220/0009171303760383.

[4] Miika Aittala. “Inverse lighting and photorealistic rendering for augmented reality”. In: *The Visual Computer* 26.6 (June 2010), pp. 669–678. ISSN: 1432-2315. DOI: 10.1007/s00371-010-0501-7. URL: https://doi.org/10.1007/s00371-010-0501-7.

[5] Ye Yu *et al.* *InverseRenderNet: Learning single image inverse rendering.* 2018. arXiv: 1811.12328 [cs.CV].

[6] Dachuan Cheng *et al.* “Learning Scene Illumination by Pairwise Photos from Rear and Front Mobile Cameras”. In: *Computer Graphics Forum* 37.7 (2018), pp. 213–221. DOI: https://doi.org/10.1111/cgf.13561. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13561. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13561.

[7] Peter Kán *et al.* “DeepLight: light source estimation for augmented reality using deep learning”. In: *The Visual Computer* 35.6 (June 2019), pp. 873–883. ISSN: 1432-2315. DOI: 10.1007/s00371-019-01666-x. URL: https://doi.org/10.1007/s00371-019-01666-x.

[8] Housheng Wei *et al.* “Simulating Shadow Interactions for Outdoor Augmented Reality With RGBD Data”. In: *IEEE Access* 7 (2019), pp. 75292–75304. DOI: 10.1109/ACCESS.2019.2920950.

[9] Claus B. Madsen *et al.* “Estimation of Dynamic Light Changes in Outdoor Scenes Without the Use of Calibration Objects”. English. In: *Proceedings: International Conference on Pattern Recognition, Hong Kong*. International Conference on Pattern

Recognition ; Conference date: 21-08-2006 Through 24-08-2006. United States: IEEE Computer Society Press, 2006. ISBN: 0769525210.

[10] Yu-ke Sun *et al.* “Learning Illumination from a Limited Field-of-View Image”. In: *2020 IEEE International Conference on Multimedia Expo Workshops (ICMEW).* 2020, pp. 1–6. DOI: `10.1109/ICMEW46912.2020.9105957`.

[11] Marc-André Gardner *et al. Learning to Predict Indoor Illumination from a Single Image.* 2017. arXiv: `1704.00090 [cs.CV]`.

[12] Mathieu Garon *et al.* “Fast Spatially-Varying Indoor Lighting Estimation”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* June 2019.

[13] Yongjie Zhu *et al. Spatially-Varying Outdoor Lighting Estimation from Intrinsics.* 2021. arXiv: `2104.04160 [cs.CV]`.

[14] Chloe LeGendre *et al.* “DeepLight: Learning Illumination for Unconstrained Mobile Mixed Reality”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* 2019, pp. 5911–5921. DOI: `10.1109/CVPR.2019.00607`.

[15] URL: `https://developers.google.com/ar/develop`.

[16] Fulvio Bertolini *et al.* “Outdoor Illumination Estimation for Mobile Augmented Reality: Real-time Analysis of Shadow and Lit Surfaces to Measure the Daylight Illumination”. In: *Proceedings of the 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications.* Vol. 1. 15th International Conference on Computer Graphics Theory and Applications. SCITEPRESS - Science and Technology Publications, 2020-2-27, pp. 227–234. ISBN: 978-989-758-402-2.

[17] Unity. *Types of light.* 2022. URL: `https://docs.unity3d.com/Manual/Lighting.html` (visited on 10/19/2023).

[18] John M Snyder. “Area light sources for real-time graphics”. In: *Microsoft Research, Redmond, WA, USA, Tech. Rep. MSR-TR-96–11* (1996).