CSCI 580 Project Report (Spring 2022)
Renderer 1.9: Advanced Lighting and Stylized Rendering

Team Members: Shravan Kumar, Luiz Cilento, Rashi Sinha, Yaorang Xie

1 INTRODUCTION

The objective of this project is to create a robust renderer that contains advanced lighting features such as ambient occlusion, shadow maps, normal mapping and stylized rendering effects like toon shading, halftone shading, object outlines and wireframes.

Each member of the team contributed to a separated feature. Shravan Kumar worked on ambient occlusion, Luiz Cilento worked on shadow mapping, Rashi Sinha worked on stylized rendering and Yaorong Xie worked on normal mapping.

2 ARCHITECTURE

The renderer implemented in this project is built using Python using the Pillow[7] library for image operations and NumPy[8] for math operations. This renderer implements deferred shading, where the lighting calculation is done in two passes. The first pass involves accumulating all the geometry information of the scene into the geometry buffer (G-Buffer) which stores attributes such as the view space position, view space normal, fragment color, materials, specularity, depth, occlusion and stencil information.

The second pass involves the typical Phong lighting calculation of every fragment using the information stored in the geometry buffer. The deferred rendering architecture was needed to facilitate features like screen space ambient occlusion which requires information of every fragment in the scene in order to work. The features implemented in the project are mostly implemented during the shading of fragments or as an intermediate pass in the rendering pipeline.

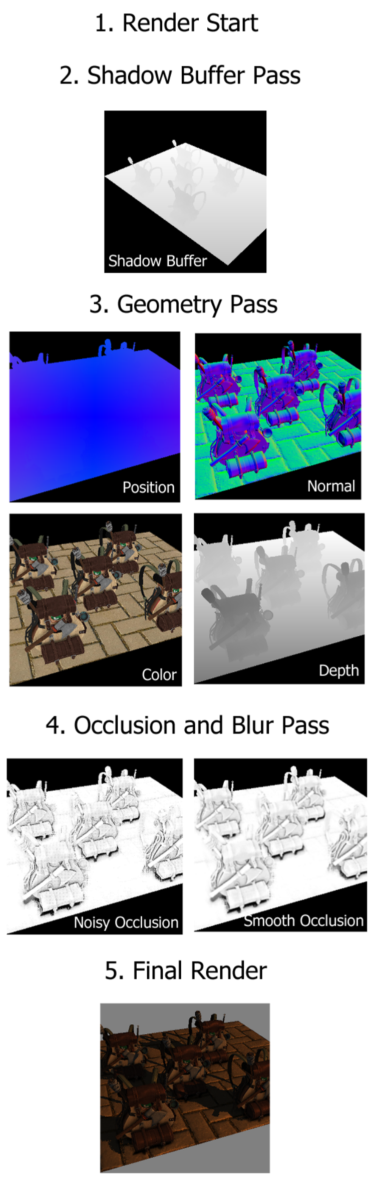

Fig. 2.1 Architecture of the deferred renderer.

3 AMBIENT OCCLUSION

Ambient Occlusion is a method of calculating the degree to which light is exposed to every point in a scene. It is used to increase the contrast in areas where light is occluded due to geometry in close proximity forming crevices, creases and barriers. This causes light to scatter and not escape thus darkening the area. The method of Ambient Occlusion implemented in the project is Screen-Space Ambient Occlusion (SSAO), where the occlusion of light is calculated in screen space by using a geometry and depth buffer.

The algorithm calculates the lighting in 4 total passes. In the first pass, the geometry information (view space position, view space normals, depth, color etc) is calculated for every fragment (pixel) in the screen and cached into the geometry buffer. The second pass calculates the occlusion of light at each fragment by querying the depth and geometry buffers at positions around a fragment in screen space. This is done by creating a kernel of random vectors, each of which are concentrated in a hemisphere in front of the fragment and rotated by a noise vector. The depth buffer is queried using these random vectors and the occlusion at a point is increased whenever a fragment with higher depth value than the queried position's depth is encountered. The total occlusion is then scaled between 0 and 1 and added to an occlusion buffer.

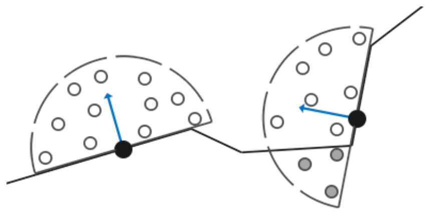

Fig. 3.1 Querying of the depth buffer around a fragment using the SSAO kernel [5].

In the third pass, the noisy occlusion buffer is smoothened by blurring it. This is done by iterating over every fragment in the occlusion buffer and averaging the occlusion value with every fragment that is around it. This occlusion buffer is then used in the final lighting pass where the occlusion value is simply multiplied with the final color of a fragment after accumulating color from all the light sources.

The biggest challenge when implementing ambient occlusion was the re-architecturing of the codebase. In order to implement SSAO, we need to know the depth and position values of all the fragments around a given point. This inherently requires multiple passes and would not be possible in the traditional forward rendering pipeline. Hence, the renderer was re-architected to use a deferred shading model where all the geometry information is accumulated in one pass and the lighting is computed in a second pass. The other features were worked on in parallel and implemented into the deferred renderer by separating them into a separate shader module. The other big challenge was to query the depth buffer in the correct way. Given that all the operations need to occur in screen space, we need to orient the random vectors correctly in order to query the correct places in the depth buffer. This was done by converting the vectors from tangent space to view space using a tangent-bitangent-normal (TBN) matrix.

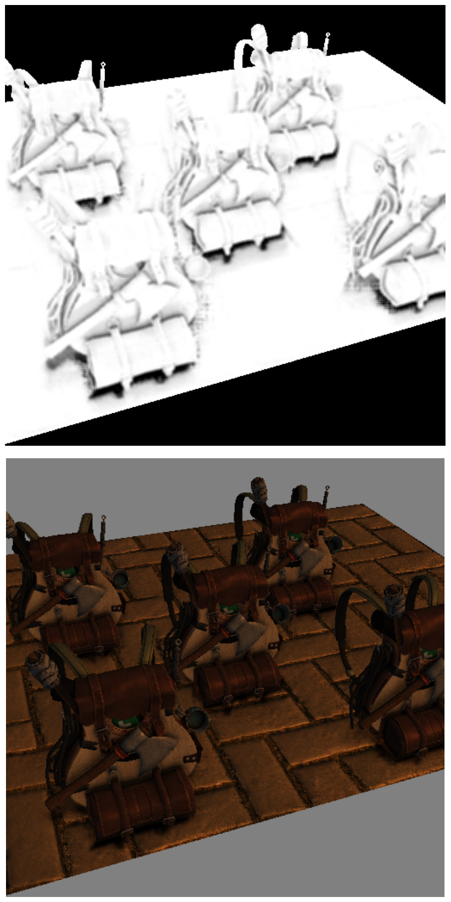

Fig. 3.2 Smoothened occlusion buffer (above) and final render (below) [6].

4 SHADOW MAPS

Shadow mapping is the process of adding shadows to a 3D scene containing at least one light source, one 3D object, and one plane to project the shadows onto. Shadows are created by comparing each object and plane's pixel to a shadow-buffer (depth image from the light's view). Given a scene containing object geometry, camera coordinates, and light coordinates, the objective is to render shadows based on the light source.

The algorithm utilized for this task is based on the idea that everything seen from the light's perspective is in light and everything behind it is in shadow. The first step in the algorithm is to render the scene from the light's perspective and store the depth of the pixels closer to the light in a shadow-buffer. This approach is analogous to how a z-buffer is filled up when rendering 3D objects.

The second step begins by rendering the scene from the camera's perspective and converting each pixel's position to an equivalent position in the light's coordinate space. The pixel's z value is compared to the value stored in the shadow buffer for that pixel's light space position. If the z value is greater than the value in the shadow buffer then the pixel is considered to be behind an occluding object and is in shadow. This can be seen below where the depth of point T(P) (point P transformed to light's coordinate space) is greater than the depth of point C therefore it is in shadow.

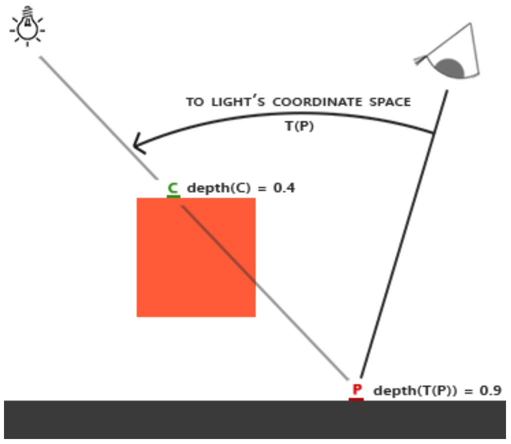

Fig. 4.1 Light space transformation and depth comparison [1]

There were two main challenges when implementing this algorithm. The first challenge was coming up with the algorithm for converting each pixel's camera space position to a light space position. Specifically, the issue was figuring out the order of transformations to go from a rasterized point in camera space to a rasterized point in light space. Ultimately, the approach taken was for each pixel position,transform the point back to world space and then transform the world space coordinate to light space for shadow-buffer comparison. The second challenge was the presence of visible issues with the edge of certain shadows when running a scene with multiple teapots. This was caused by an edge case where the pixel position was outside the shadow-buffer (the light could not "see" this position) which was resolved by setting pixels that fell outside the boundaries of the shadow-buffer to be in light.

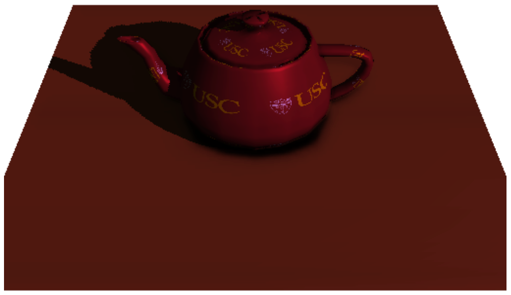

Fig. 4.2 Shadow and ambient occlusion rendered for *Utah Teapot*

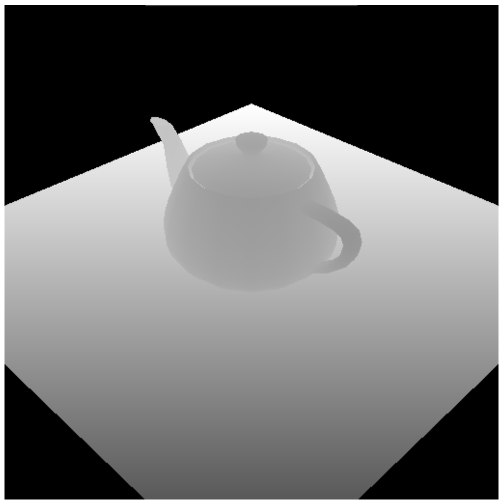

Fig. 4.3 Shadow buffer for *Utah Teapot*

5 WIREFRAME

A "wireframe" is a skeletal representation of a 3D computer graphics model of a real-world object. It represents the primitives used to create the 3D model by their vertices and edges in 3D space. Given an input scene with geometry information for a 3D model, the objective is to render a wireframe of the model.

The approach adopted to render the wireframe for a 3D model is to draw lines to represent each triangle in 2D (raster) space. The rasterizer implements the Bressenham's Line Drawing Algorithm[12] to achieve this. For a line with slope less than 1 between two vertices *(x_1,y_1)* and *(x_2,y_2)*, the algorithm steps from the smaller *x* to larger *x* one-by-one and decides which of *(x+1,y)* or *(x+1,y+1)* pixel to color next. The transformed raster space vertices of each triangle are used as input to the line drawing method and three lines are rendered per triangle.

It was a challenge to come up with a generalized algorithm to include slopes greater than 1. It would be required to step in the *y* direction instead of *x* to be able to draw such lines. Depending on the slope of a line, swapping the *x* and y values with each other can generalize the algorithm.

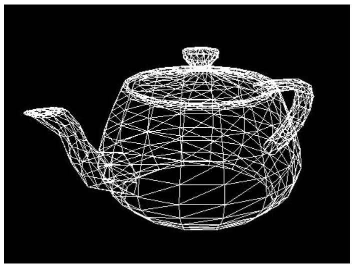

Fig. 5.1 Wireframe rendered for the *Utah Teapot*

6 STYLIZED RENDERING

6.1 Line Art

Line art is a minimalistic representation of complex objects. Artists draw a free-hand outline of objects in 2D that they refer to as line art. The line art stylized rendering feature is a non-photorealistic rendering technique that would render the outline of a 3D model to give an effect of it being hand-drawn.

Back-face culling[13] is the process of removing polygons/faces that are not visible to the viewer (camera) from the render. It essentially means skipping any faces that are facing away from the camera. It can be determined if a polygon is a back face if the *dot product* between the *view vector* and the *surface normal* of the face is positive, in object space.

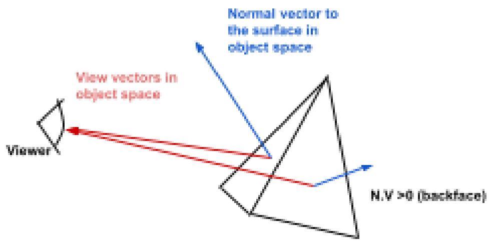
Normal vector to the surface in object space
View vectors in object space
Viewer
N.V >0 (backface)

Fig. 6.1.1 Back-face culling

The algorithm for this stylized rendering method implements inverted back-face culling method to determine back faces and render them in black color. It then renders the front-facing polygons using the z-buffer and colors them the same as the background color of the image. However, in order to actually be able to see the black outline, it increases the thickness by pulling the vertices of the back-facing triangles in their respective normal directions.

One of the key challenges in this implementation was to make the final render look realistic. The stylistic feature of a free-hand line drawing is the variation in the thickness of the lines. To incorporate this variation in the stylized renderer, perlin noise[14][15] was used to smoothly vary how much a vertex of a back face is translated in its normal direction.

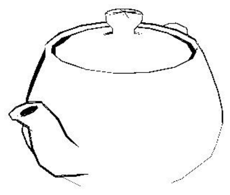

Fig. 6.1.2 Line Art rendered for the *Utah Teapot*

6.2 Toon/Cel Shading

Toon shading[16][17] (or cel shading) is another non-photorealistic rendering technique that mimics the style of 2D drawings in comic books, cartoons or anime. It renders a 3D computer graphics model as a flat 2D object without smooth illumination.

The implementation takes in a 3D model and quantizes the *N dot L* value for each pixel to a range of three values (bright, light and dark). It then uses this new *N dot L* value to calculate the diffused color of the pixel to add to the ambient lighting component. The resulting color for each pixel is one of three shades of the object's base color.

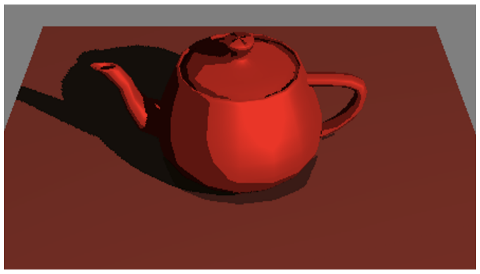

Fig. 6.2.1 Toon (Cel) Shading rendered for the *Utah Teapot* with ambient occlusion and shadows.

It was challenging to draw the outlines for this rendering effect. The first attempt was done to utilize a stencil buffer to render the toon shaded object in the first pass and then a scaled-up version of the object in black in the second pass to represent the outline. This method provided a good silhouette behind the object in the final render. However it was not able to render outlines on the front parts of the object.

The second and final attempt for outlines adopted the inverted back-face culling method explained in *section 6.1.* This method renders the toon shaded front object and the back faces as the black outlines in a single pass.

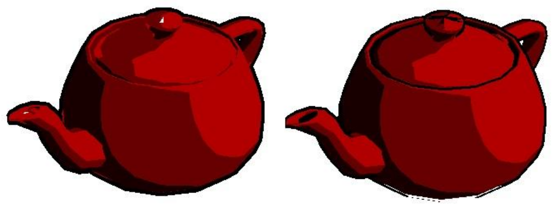

Fig. 6.2.2 Outline results *(a)* Silhouette with two pass stencil-buffer, and *(b)* Outlines using inverted back-face culling

6.3 Halftone Shading

Halftone shading technique[18][19] draws objects using dots with varying sizes or spaces. It represents the lighter areas with smaller, sparsely spaced dots and the darker areas with bigger dots that are closer to each other. The dots are colored with quantized color values just like in toon shading.

The rasterizer implements this technique by computing the corresponding pixel position on a grid of dots of smaller resolution and comparing its distance from the center of the pixel to the radius of a dot, which is calculated using the total lighting value for that pixel using the *Phong Illumination Model.* This makes the dot's radius to be bigger when there is less or no light on that pixel and smaller when there is more or all light on that pixel. The comparison result determines whether the pixel is within a dot (circle) or outside it. If it is within the circle, the pixel is colored with one of the quantized color values. Otherwise it is colored the same as the background color of the image.

It was difficult to determine which value to quantize in order to get the effect. Initially, the *N dot L* value was quantized just like in toon shading. However, since this value contributes to the total lighting value that is used for comparison, this method was producing incorrect results. To rectify this error, the final lighting value was quantized.

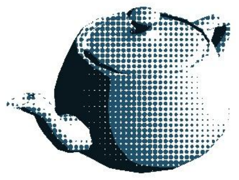

Fig. 6.3.1 Halftone Shading rendered for the *Utah Teapot*

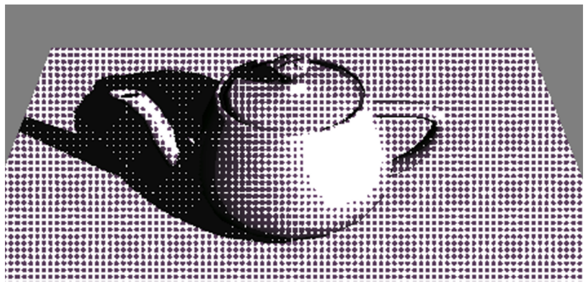

Fig. 6.3.2 Halftone Shading with ambient occlusion and shadows

7 NORMAL MAPS

Normal mapping is an advanced implementation of bump mapping. It is a texture mapping technique to add surface details to the model without adding more polygons to the original model, such as bumps, bents, grooves, scratches, etc. These surface details are used to fake the lighting of the model as if it were represented by the real geometry. A normal map is usually generated from a high polygon model and is used to enhance the appearance and details of a low polygon model.

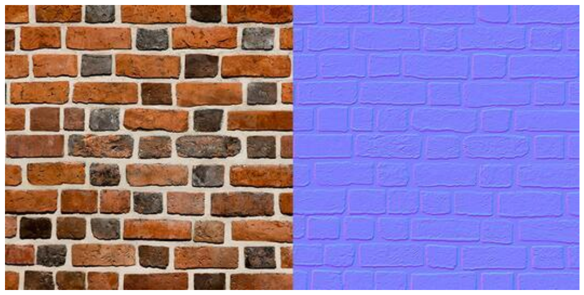

Fig. 7.1 Example of normal map (The left figure is the original model and the right figure is the normal map)

There are three steps in my implementation of normal mapping. Firstly, I generate a depth map by using the z-values of the original model. The resulting image is a grayscale image showing the depth of the model in the raster space. I put the depth of a specific point into the depth map only when it passes the triangle test of the barycentric coordinate system and z-buffer testing. Finally, I map the depth to the range [0, 255] and store it in an image format. The following figures show an example of the original model and the resulting depth map image.

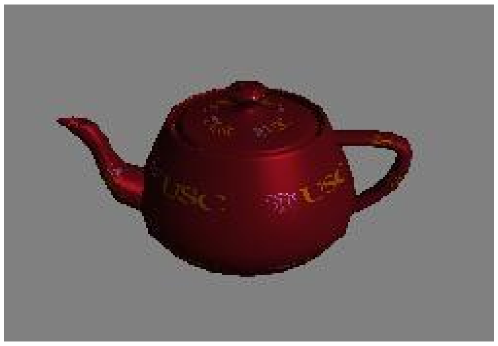

Fig. 7.2 Original *Utah Teapot*

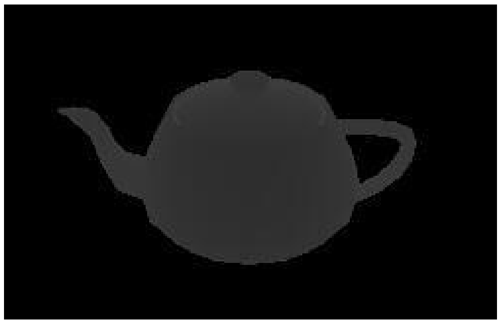

Fig. 7.3 Depth map of *Utah Teapot*

The second step is to construct a normal map from the depth map. To calculate the normal of a specific direction (x, y), I first compute the tangent vector of the plane that is parallel to the x-axis and the tangent vector of the plane that is parallel to the y-axis. The tangent vectors are calculated based on the rate of change of z with x and the rate of change of z with y. After I have the two tangent vectors, the normal will be obtained by performing a cross product of the two tangent vectors, because the normal is perpendicular to them. Finally, I normalize the resulting normal and convert it from the range [-1, 1] to [0, 255] to store it in an image format.

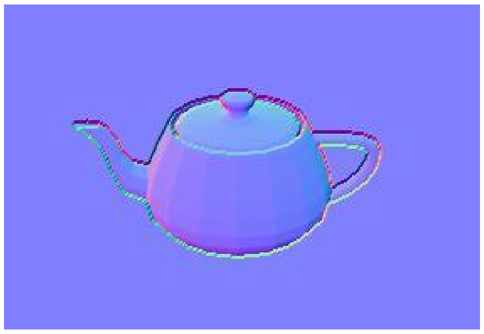

Fig. 7.4 Normal map of *Utah Teapot*

After applying the normal map to the model, the Utah Teapot looks like this:

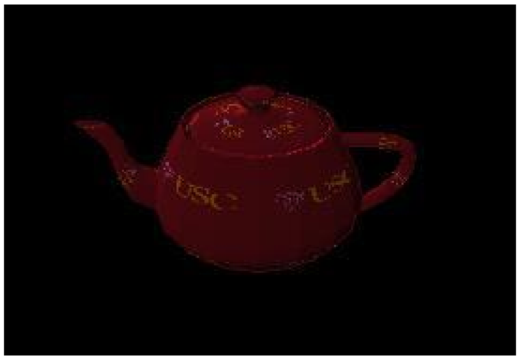

Fig. 7.5 *Utah Teapot* with normal map

The overall geometry of the model still looks good with some lighting details lost. It is probably because the normals are essentially fake and are generated only from the depth. Another possible issue is that some precision may be lost during the process of mapping the depth and normal to an image format.

8 ADDITIONAL RENDERS

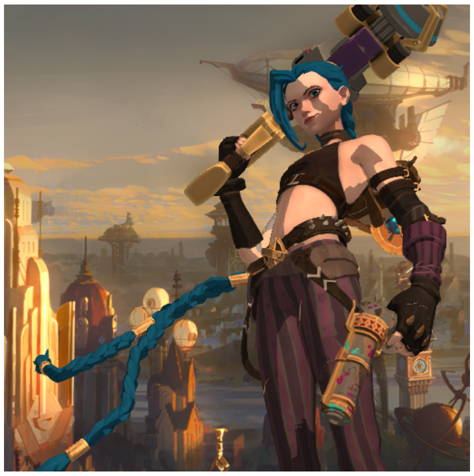
Fig. 8.1 Render of Jinx from Arcane, League of Legends (Riot) [9][10].

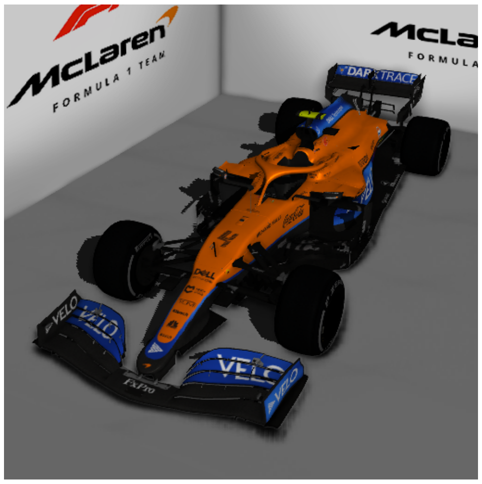

Fig. 8.2 Render of a McLaren F1 car [11].

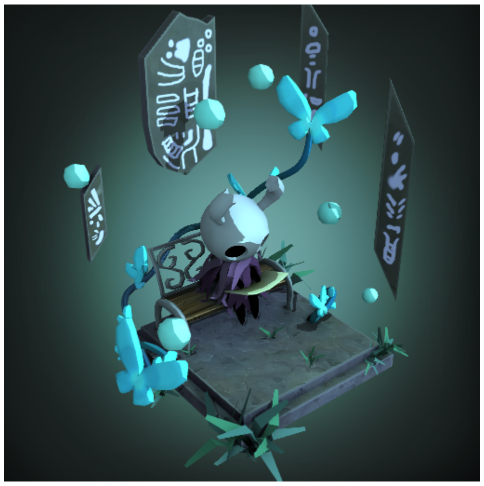
Fig. 8.3 The Knight from Hollow Knight [20][21].

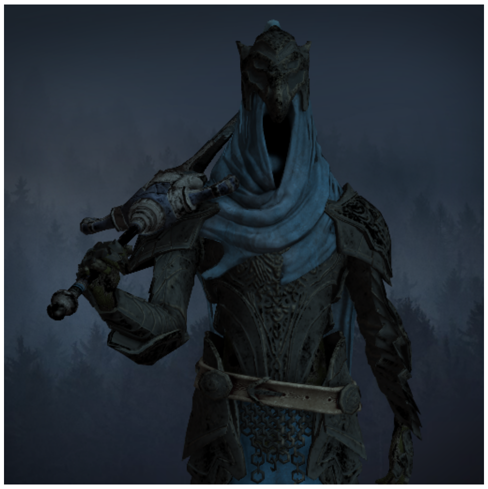
Fig. 8.4 Artorias from Dark Souls [22][23].

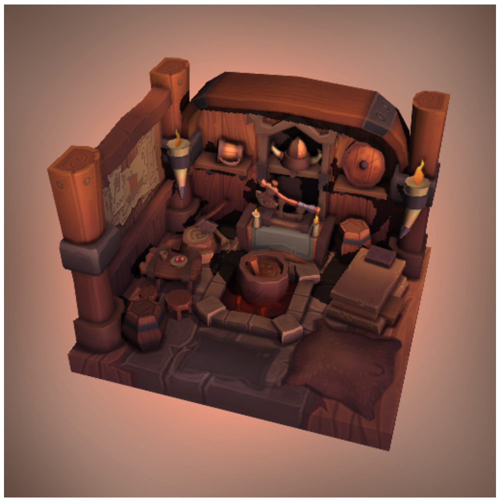
Fig 8.5 Viking room [24].

9 REFERENCES

[1] “Shadow Mapping.” *LearnOpenGL*, https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping.

[2] “Shadow Mapping.” *Wikipedia*, Wikimedia Foundation, 1 Dec. 2021, https://en.wikipedia.org/wiki/Shadow_mapping.

[3] Raghavachary, Saty. “Shadows.” CSCI 580: 3D Graphics and Rendering. https://bytes.usc.edu/cs580/s22_CG-012-Ren/lectures/Lect_Shadows/Shadows.pdf. Accessed 21 Apr. 2022.

[4] Wikimedia Foundation. (2022, April 14). Normal mapping. Wikipedia. Retrieved April 22, 2022, from https://en.wikipedia.org/wiki/Normal_mapping

[5] “Screen Space Ambient Occlusion”, LearnOpenGL, https://learnopengl.com/Advanced-Lighting/SSAO

[6] “Survival Guitar Backpack”, Berk Gedik, Sketchfab, https://sketchfab.com/3d-models/survival-guitar-backpack-low-poly-799f8c4511f84fab8c3f12887f7e6b36

[7] Pillow, https://pillow.readthedocs.io/en/stable/

[8] NumPy, https://numpy.org/

[9] Arcane, https://arcane.com/

[10] Arcane - Jinx Model, Craft Tama, https://sketchfab.com/3d-models/arcane-jinx-b74f25a5ee6e43efbe9766b9fbebc705

[11] F1 2020 McLaren MCL35, Excalibur, https://sketchfab.com/3d-models/f1-2020-mclaren-mcl35-428df471f2ec489fa914f5f517b3ef64

[12] Bressenham’s Line Drawing Algorithm, https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html

[13] Back-face culling, http://medialab.di.unipi.it/web/IUM/Waterloo/node66.html

[14] Perlin-noise, https://pypi.org/project/perlin-noise/

[15] Perlin noise, Khan Academy, https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise

[16] Cel Shading, https://danielilett.com/2019-06-01-tut2-0-lighting-models/

[17] The Cel Shading Technique, Raul Reyes Luque Dec 2012, https://raulreyesfinalproject.files.wordpress.com/2012/12/dissertation_cell-shading-raul_reyes_luque.pdf

[18] Halftone, https://en.wikipedia.org/wiki/Halftone

[19] Halftoning Techniques, https://www.youtube.com/watch?v=d8gBnnjC6Jk

[20] “Hollow Knight”, Dasha Klyusova, Sketchfab, https://sketchfab.com/3d-models/hollow-knight-5a76d93e39984f829abd6f406562265b

[21] Hollow Knight, Team Cherry, https://www.hollowknight.com/

[22] “Knight Artorias”, Samize, Sketchfab, https://sketchfab.com/3d-models/knight-artorias-0affb3436519401db2bad31cfced95c1

[23] Dark Souls, From Software, https://www.fromsoftware.jp/ww/

[24] “Viking Room”, nigelgoh, https://sketchfab.com/3d-models/viking-room-a49f1b8e4f5c4ecf9e1fe7d81915ad38