An Automated Method for Converting 3D Meshes into Editable 2D Vector Graphics

Vidar Hartveit Ure

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Western Norway University of Applied Sciences

Department of Informatics,
University of Bergen

June 2019

Western Norway
University of
Applied Sciences

Abstract

When computer programs generate and render graphics, the result is typically raster graphics. Raster graphics is described by a 2D grid of a predefined size, e.g. 1000x1000, where each element is a color, called a pixel. This static nature of raster graphics can be a disadvantage in graphic design where artists may have rendered images of 3D scenes as their working material. Furthermore, such raster images lack information about geometry that is occluded from the given viewpoint, making it cumbersome to edit them. Vector graphics is an alternative approach that uses geometry like curves and paths to draw shapes, and the image can contain more than one layer of geometry. Because the size of such geometry is dynamic, vector graphics can be scaled without sacrificing visual quality. This thesis presents an implemented solution for converting the 3D meshes in a scene into layered vector graphics illustrations, including geometry that is occluded from the given viewpoint. All the triangles in a mesh are grouped together to increase their editability when they have been converted into vector graphics. The solution has been made publicly available for use in the form of Unity scripts.

Acknowledgements

First and foremost I would like to thank my three supervisors at Western Norway University of Applied Sciences (HVL), Harald Soleim, Atle Geitung and Daniel Patel, for their support, guidance and frequent feedback during the entire work on this thesis. They always encouraged me to do my best, and this thesis would not have been possible without their help. I also want to thank Marius Giske for taking the time to assist me with using his solutions in my own work, that was crucial for my thesis to get as far as it did.

Contents

Glossary

Application Programming Interface (API) A set of functions or commands that can be used to create software or interact with an external system. iv

Depth layer An individual image in the LDI. It shows the colors of the fragments that are in the same given index of the per-pixel linked list, in every pixel position. iv

Face A set of vertices that are connected to form a part of a mesh. May have three (triangle) or more vertices. iv

Layered Depth Images (LDI) A set of images created by rendering a scene off-screen and storing the pixels for each depth layer into its own separate image. iv

Mesh A set of faces used to define a surface in 3D graphics. iv

Mesh chain A collection of meshes where each mesh both occludes one mesh and is occluded by another (occlusion cycle). iv

Outline A line that identifies the boundaries of an object. Also referred to as "contour". iv

Scalable Vector Graphics (SVG) A file format for vector graphics. iv

Self-occluding mesh A mesh where some faces are occluding other faces of the same mesh, from a given viewpoint. iv

Silhouette A two-dimensional representation of a figure where the figure is identified by its outlines. iv

Vertex A point in 3D space. iv

Chapter 1

Introduction

1.1 Thesis Outline

Chapter 1 will start with explaining the problem description of the thesis, before presenting the goals that should be met in order to solve the explained problem. It then goes on to present the research question. Chapter 1 ends with an in-depth presentation of related work that addresses some of the same issues as this thesis, and how the work differs from this thesis.

Chapter 2 presents all the background knowledge needed to understand the design and implementation of this thesis, as well as its results, that will be explained in the remaining chapters. Chapter 2 also explains the tools and frameworks that the thesis solution builds upon.

Chapter 3 explains the solution of the thesis and how it was implemented, by first presenting the design choices in a general manner and later explaining each stage in depth.

Chapter 4 presents and evaluates the quantitative and visual results that were measured and observed during the implementation and testing of the solution.

Chapter 5 summarizes the key points of the thesis and lessons that were learned during the course of the thesis work.

Chapter 6 lists further work that can be done on the solution in order to improve it.

1.2 Problem Description

When computer programs generate and render graphics, the result is typically raster graphics. Raster graphics is described by a 2D grid of a predefined size, e.g. 1000x1000, where each element is a color, called a pixel. This means that if it is desirable to zoom in and examine certain parts of the image, the image will become unclear, or "pixelated". One can also only see objects that are visible from the angle the image was rendered from, and there exists no information about objects behind the pixels, because all occluded geometry is discarded by the rendering process.

Due to these disadvantages, there are a wide variety of situations where editing a raster graphics representation generated from a 3D scene may be an inconvenience, or even impossible without altering the 3D scene itself. These situations may often arise in graphic design, where frequently the working material is a rendered image of a 3D scene. For example, an architect may need to edit an image of an apartment to make it as informative as possible to a potential buyer, for instance in a brochure, by highlighting different surfaces with text or colors. A geologist may want to edit parts of a terrain model for use in a report, by removing layers in the terrain's geology to expose a certain layer of interest. Similarly, an engineer may need to perform similar editing on a 3D model representation of a car engine, by highlighting certain internal components for use in a technical manual, for instance in the form of exploded views or cutaways [1].

These are all examples where artists often need to edit rasterized images that are the result of rendering 3D scenes through the rendering pipeline. Because the result of such rendering is simply an image consisting of a fixed amount of pixels, the material they work with scales poorly, in regards to zooming in and out on the image. Information about geometry that is obstructed by other geometries is also lost. This means that the artists have a limited ability to edit the images in an intuitive way, and they may spend tedious time editing the 3D scenes themselves before they are rendered onto images.

An example of models that could apply to such scenarios is shown in Figure 1.1 [2]. It shows a 3D geological model of a shoreline along the Hudson River in Newburgh, New York, containing a gas plant and its surrounding terrain. The model includes several structures and layers, like buildings, water, a sewer line and a bedrock.

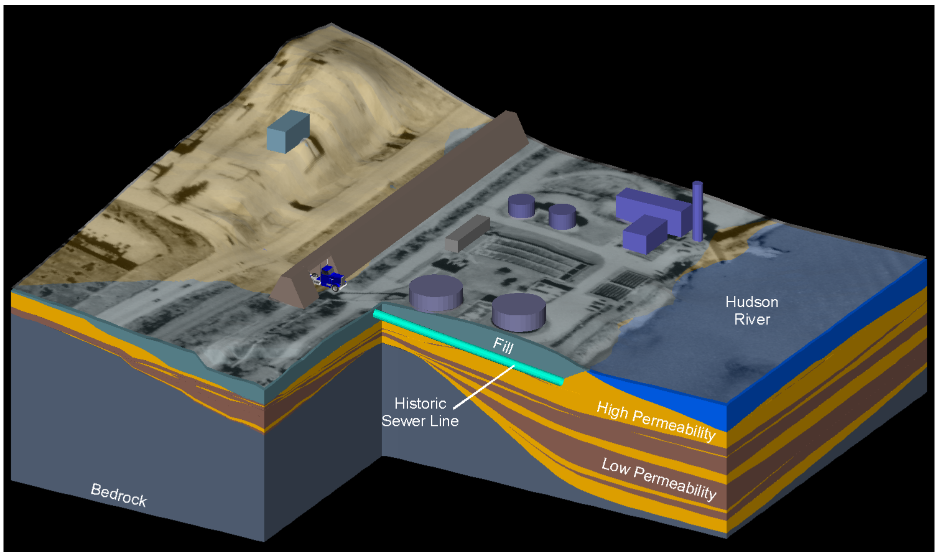


Figure 1.1: A geological model of a gas plant site in Newburgh, New York [2].

Such models often consist of complex layered structures where the most interesting parts could be obstructed by other parts. Furthermore, they may contain large amounts of information where only parts of the information is relevant in the given context. These factors can make examining the models, and the process of making them suitable for presentation through image editing, a cumbersome process. In the example above, it may for example be desirable to remove or highlight the buildings, water or the sewer line, as well as certain geological layers, depending on the purpose of the image. Doing this on the raster image is tedious. If the image had been stored as vector graphics with a group of silhouettes for each layer, this would suddenly be very easy. A silhouette in this context means a two-dimensional, single colored shape identified by the outlines of an object, from a certain viewpoint [3].

1.3 Goal

The issues explained in Section 1.2 are solvable by using vector graphics. Vector graphics is defined by more scalable properties than raster graphics, like paths and geometrical shapes, and is therefore not limited to a fixed pixel amount. Vector graphics also supports multiple layers in a single image, allowing for drawing shapes on top of each other. The differences between raster graphics and vector graphics will be explained more in depth in Section 2.1.

Due to the benefits of vector graphics and limitations of raster graphics, the thesis will attempt to solve the problem introduced in Section 1.2 by implementing a solution that acquires the underlying 3D meshes in a scene, with its colors, and flattens them into a layered structure of 2D silhouettes better suited for editing. The silhouettes are then converted into vector graphics. Because of the advantages of vector graphics, a vector graphics representation of the meshes will allow for zooming into the image without it becoming pixelated. Any hidden structures in the scene will be maintained by organizing the graphics into multiple layers sorted by depth.

The user should at any point be able to render the scene in its current form to the screen, and apply changes to the scene, such as defining new meshes and applying matrix transformations in order to scale, rotate and translate the object in the scene. The user can then generate a layered vector graphics "screenshot" of the scene whenever the user is satisfied with its current form and projection.

Tagging all the triangles that belong together in a single mesh before rendering is important for the user manipulation of the vector graphics to be effective even for large and complex scenes with a high amount of triangles. The user can then simply apply changes on silhouettes that naturally belong together, as opposed to processing each individual triangle as a separate silhouette. For example, editing a house in vector graphics where each wall is divided into several triangles is more cumbersome than simply working with one silhouette for each wall. It is also important that the silhouettes are drawn in the correct order to ensure that each of them knows which other silhouettes they are behind and in front of.

The solution should be implemented as a collection of scripts that are capable of importing a scene, and convert it as it is rendered from a given viewpoint, into a Scalable Vector Graphics (SVG) file. SVG is a file format for drawing vector graphics. The scripts can be used by themselves or be used in other software to facilitate the conversion of 3D meshes into 2D vector graphics, by simply having the user provide a 3D scene along with the needed information about the organization of the meshes. The scripts will be made as a publicly available open source project.

Two important properties of a 3D mesh are its textures as well as shading, where the latter is dependent on the properties of the lights in the scene, and the material properties of the object. However, to limit the problem's domain and make it more feasible for this thesis, textures and shading will not be considered in the solution's convertion of 3D meshes into vector graphics. The fact that textures and shading are not scalable is another argument for excluding them in this thesis, because they would simply be rasterized images that are applied on top of the finished vector graphics. This also means they can be distorted when the vector graphics shapes are edited by the user. Due to these factors, this thesis will instead establish a foundation by focusing on converting meshes with single-colored triangles into flattened vector graphics shapes. This foundation can then be extended further by including textures and shading in the future.

1.4 Research Question

This thesis will address the issue of converting a 3D scene, with all its meshes, into vector graphics. This will require the implementation of a solution that is capable of solving this issue. After thorough research, there appears to be no existing solution available for public use at this point. The research question for this thesis is therefore: **How can a scene of 3D meshes be converted into a vector graphics format from a given viewpoint?**

1.5 Related Work

The advantages of vector graphics and disadvantages of raster graphics are already well known, and the issue of converting raster graphics into vector graphics is not an unexplored field. However, most of the current solutions revolve around converting already rasterized images (including regular photographs and 2D drawings) into vector graphics. Software like Blender [4] and Adobe Illustrator [5] already have ways of exporting images into vector file formats such as SVG. While they do solve the issue of maintaining the image quality while scaling the image, their process is applied after the scene has been rendered into an image, and thus only gives information about the structures that are visible from the given viewpoint.

Although most current solutions appear to only revolve around already rasterized images, there have also been made important contributions with the goal to use all the advantages of vector graphics in the context of 3D computer graphics. These are explained in the following subsections.

A Visibility Algorithm for Converting 3D Meshes into Editable 2D Vector Graphics

"A Visibility Algorithm for Converting 3D Meshes into Editable 2D Vector Graphics" by Eisemann et. al. [1] presents a method for converting 3D meshes into vector graphics. The article's main contribution to the topic is an algorithm capable of splitting a 3D mesh into multiple layers so that each layer can be ordered back to front with respect to visibility. Each stage of this algorithm is explained in detail in 2.3.1. In 3D space, it is possible to have three surfaces that all overlap each other from a given viewing angle. This is what the paper defines as self-occlusion. Self-occlusion is not possible in 2D space. In the paper, they elaborate on the challenges of representing self-occluding meshes and occlusion cycles in a two-dimensional vector graphics format. Self-occluding meshes and occlusion cycles are highly relevant to this thesis, and are explained further in Section 2.3.1.

SVGPU: Real Time 3D Rendering to Vector Graphics Formats

Ellis et. al.'s "SVGPU" [6] is an engine designed with the goal of taking the compact and resolution independent nature of vector graphics, and applying it to real-time rendering of 3D graphics. This can benefit cloud streaming services such as those within the video game industry, where 3D graphical images are rendered on a server and transmitted to a consumer device such as a phone or a PC. By outputting a vector graphics representation of the 3D scene instead of rasterized images, SVGPU may greatly reduce the bottleneck of transmitting such images of ever increasing resolutions over potentially limited communication networks.

Stylized Vector Art from 3D Models with Region Support

"Stylized Vector Art from 3D Models with Region Support" is, as described by the creators Eisemann et. al. [7], "a rendering system that converts a 3D meshed model into the stylized 2D filled-region vector-art commonly found in clip-art libraries". Its goal is to automate and simplify the process of creating illustrations based on 3D models, and the purpose of the regions is to better facilitate stylizing of vector art.

"3D to SVG" by StyleCampaign

StyleCampaign's tool converts 3D graphics into SVG files, and is called "3D to SVG", or "scVector" [8]. This tool lets the user render a 3D model to a window, where the positions of the camera and model can be adjusted. The user can also toggle different render modes to for instance add or remove shadows. Once the user is satisfied with the scene, it can be saved as SVG.

1.5.1 Comparing the thesis with related work

Eisemann et. al.'s visibility algorithm [1] is the most relevant previous work to this thesis, as it solves some of the major challenges associated with converting 3D geometry into 2D, namely self-occluding meshes and mesh chains.

The main difference between Ellis et. al.'s work on the SVGPU [6] and this thesis, is that SVGPU focuses on the fact that vector graphics is inexpensive to store and transmit over a network, while still maintaining a satisfying result to the consumer in terms of image resolution. It appears to focus only on the geometry visible to the viewer from a certain projection, and discards any occluded geometries, making the result less valuable to illustrators who need to edit the different image layers. SVGPU works in real-time, which the resulting solution of this thesis does not.

Eisemann et. al.'s method [7] for creating stylized vector art from 3D models addresses some of the challenges that this thesis also focuses on, namely automating the process of creating vector graphics out of 3D models. However, this challenge is approached with stylizing in mind. Also, as stated in their paper, their focus is on art similar to that of clip-art libraries, and not on larger, more complex scenes. As with SVGPU, it focuses on the visible parts of the models and not on any hidden structures of the models.

The goal of StyleCampaign's "3D to SVG" tool is similar to the goal of this thesis, but the goal is reached with a more simplistic approach. Because it simply takes the triangles from a 3D object and saves them individually as SVG triangles, this approach results in enormous files for larger scenes. This would also make editing the meshes a cumbersome process, as it would only allow the user to operate on individual triangles. Because of the mentioned technique for grouping the triangles, those issues do not apply to the solution of this thesis.

The main problem with all previous work is that neither software nor code appears to be available, which will be addressed in this thesis. The exception is StyleCampaign's "scVector", which is available for use but does not appear to be open source. This thesis will make its code available in a game engine so that it is simple to be improved or integrated into other solutions. Should some of the previous work have lead to better results than those that will be obtained from this thesis, implementing a solution to the same issues in this thesis will still be an important contribution due to the availability of the code.

Chapter 2

Background

2.1 Raster Graphics and Vector Graphics

Raster (or bitmap) graphics is stored as a series of pixels. Pixels are small squares that each are assigned a color, and are arranged in a pattern to form the image [9]. Vector graphics however, is not made up of a grid of pixels. Instead, it consists of a startpoint and an endpoint, that might have other points as well as curves and angles between them. These points, curves and angles together make up a path, which can be a line, square, triangle or curvy shape [10].

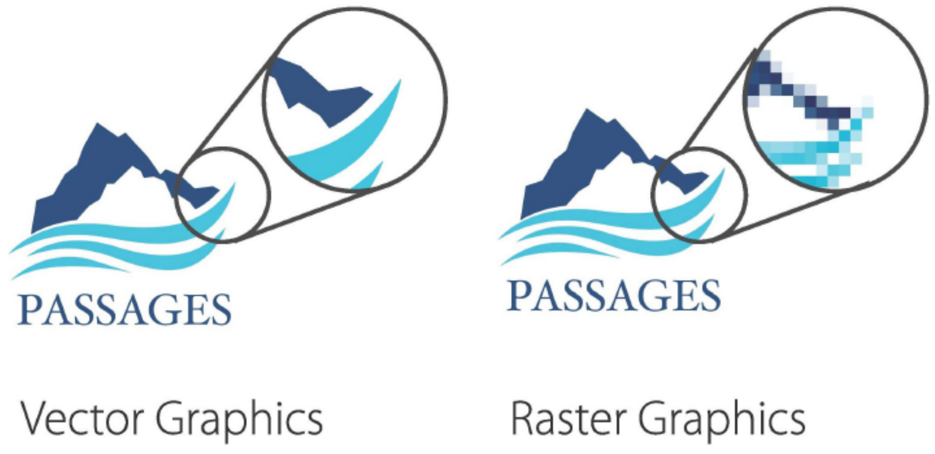


Figure 2.1: Visual example of vector graphics and raster graphics. [11]

Comparing the two ways of rendering graphics, it is clear that scaling vector-based images yields a significantly better result than scaling raster graphics. Since vector graphics is defined by paths, it can be scaled without losing any image quality, while images based on raster graphics will look pixelated due to the fact that they consist of a fixed amount of pixels. This makes vector graphics ideal for logos because scaling of such images often is necessary to display them on anything from a small poster to a large billboard [10]. Raster graphics is usually used for images that need a wide range of color gradations such as photographs [9]. Figure 2.1 shows an example of vector graphics and raster graphics, where the latter becomes pixelated whereas the former maintains its quality after scaling.

2.2 Scalable Vector Graphics (SVG)

SVG (Scalable Vector Graphics) as defined by its creators, The World Wide Web Consortium[12], is "a language for describing two-dimensional graphics in XML. SVG allows for three types of graphic objects: vector graphic shapes (e.g., paths consisting of straight lines and curves), images and text." SVG contains a set of several basic shape elements that can be defined and combined together to make more complex shapes, like rectangles, circles, ellipses, straight lines, polylines and polygons. Figure 2.2 and its corresponding code listing describe an example SVG file which specifies two polygons in the form of a star and a hexagon.[13]

```
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
     xmlns="http://www.w3.org/2000/svg" version="1.1">
  <desc>Example polygon01 - star and hexagon</desc>

  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="1198" height="398"
        fill="none" stroke="blue" stroke-width="2" />

  <polygon fill="red" stroke="blue" stroke-width="10"
           points="350,75  379,161 469,161 397,215
                   423,301 350,250 277,301 303,215
                   231,161 321,161" />
  <polygon fill="lime" stroke="blue" stroke-width="10"
           points="850,75  958,137.5 958,262.5
                   850,325 742,262.6 742,137.5" />
</svg>
```

Listing 2.1: An example of an SVG file. [13]

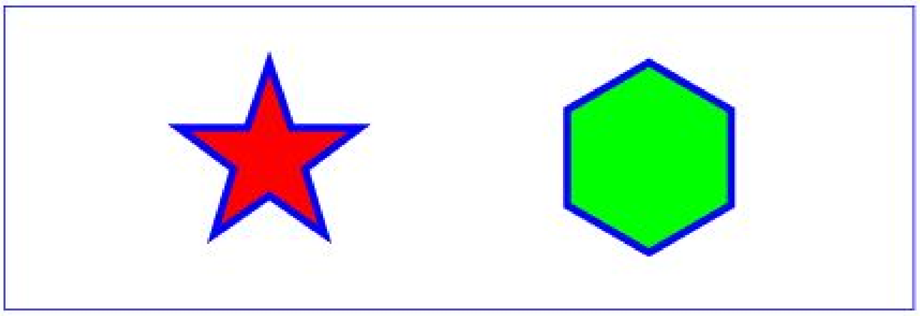


Figure 2.2: An SVG file specifying a star and a hexagon by using polygons. [13]

As shown in this example, every SVG document contains an <svg> tag which specifies various attributes for the canvas in which to draw the vector graphics, such as width and height. In this case the two figures are both defined with the <polygon> tag, which contains a list of points of x and y coordinates that together draw the shape. Various attributes like fill, stroke and stroke-width are used to give the shapes color. Fill defines the fill color of the shape, stroke defines the outline that defines the shape, and stroke-width is the thickness of the outline. In this particular case the star and hexagon are also contained inside a rectangle defined by the <rect> tag which has a thin blue frame.

As has been already explained in Section 2.1 about vector graphics, SVG is not limited to fixed pixel sizes, but is scalable to different display resolutions. It can be magnified or reduced infinitely without any reduced quality. Additionally, "scalable" in the context of SVG also means that, as it runs on the Web, it is compatible with a variety of different web browsers and can therefore be used in a wide range of different applications by a large number of users. The graphics can be implemented as a single stand-alone SVG file or be combined by several files to create more complex graphical structures.[14]

2.2.1 SVG Rendering Order

An important factor of SVG in the context of this thesis is the rendering order of SVG elements. Since the input of the solution will be a three-dimensional scene of meshes, all its structures should be kept in the resulting SVG document. Although vector graphics is two-dimensional, SVG elements are still positioned in three dimensions implicitly according to the order in which each element is drawn. The element that is drawn first is the first element in the SVG document, and each following element will be drawn on top of the previous ones [15]. It is important that the order is taken into account in the solution.

2.3 Triangle meshes

In order to represent three-dimensional geometry in computer graphics, triangle meshes are used. The first component that is needed to define geometry are vertices. A vertex is a point in 3D space defined by x, y and z coordinates. Vertices are then connected by edges, where two vertices connected by a single edge define a line, and three vertices connected by three edges define a triangle. Triangles are the most basic way of defining a face or a polygon. Several triangles can be connected together to define more complex shapes. [16]

2.3.1 Self-occluding meshes and occlusion cycles

There are two major challenges associated with taking a three-dimensional scene of meshes and converting it into 2D silhouettes, namely self-occluding meshes and occlusion cycles [1]. These were briefly introduced in Section 1.5, but are explained further here.

The bent tube in Figure 2.4 and the three overlapping meshes in Figure 2.3 are examples of cases where self-occlusion and occlusion cycles occur, respectively. They both present their own challenges when they are to be converted into a layered structure of silhouettes. Because the bent tube overlaps itself from the given viewpoint (self-occlusion), simply converting it into a single silhouette is not sufficient, as the occluded part will be lost. As for the three meshes that overlap each other from the given viewpoint (mesh chain), it would not be enough to simply make one silhouette per mesh, because it is impossible to draw them in a correct order. Both these challenges show the need for a solution that splits each of the objects into multiple silhouettes across multiple depth layers.

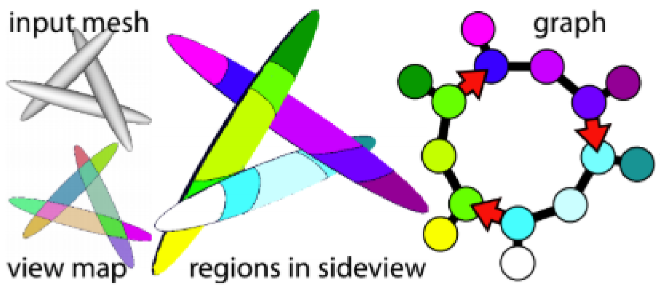


Figure 2.3: A mesh chain resulting in an occlusion cycle [1].

Figure 2.4 [1] explains each stage of their algorithm that solves self-occlusion and occlusion cycles by identifying and removing cycles in the meshes. Their algorithm first takes a self-occluding mesh or a mesh chain and computes its view map by projecting the contours of the mesh. This view map is used to define the regions of constant visibility on the mesh. Each colored part of the mesh corresponds to such a region. A graph is constructed based on these regions, where each region represents a node. Each pair of neighboring regions are connected by an undirected edge. When a region occludes another region, such as region 3 occluding region 1 in the figure, an arc goes from the occluding region to the occluded region. Based on the information in this graph, it is possible to determine where to perform a cut on the mesh. The challenge comes down to eliminating all cycles in the graph. Once a cycle has been eliminated, a layer ID is assigned to the foremost part created by the cut. This part then ready to be handled as an independent layer when converting it to vector graphics. A mesh could potentially have several occlusion cycles or self-occlusions. Therefore the steps of identifying graph cycles, and cutting the mesh based on one of the cycles (steps c) through f) in the figure), are repeated recursively until all cycles have been removed.

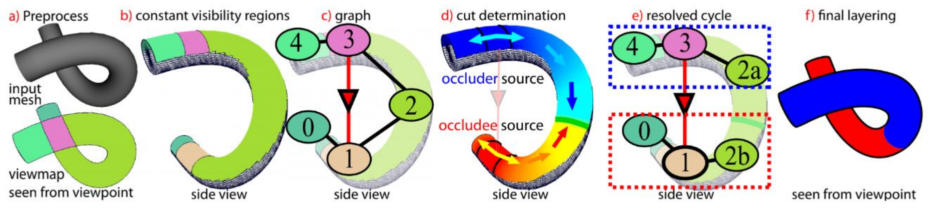


Figure 2.4: Eisemann et. al.'s process of cutting a 3D mesh into multiple layers [1].

Self-occluding meshes and occlusion cycles are two issues that could lead to unsatisfying results in a 2D vector graphics representation of a 3D scene. [1] presents those issues and why they are important in detail, and provides its own approach to solving them, as explained above. These challenges will be taken into account when designing the solution of the thesis.

2.4 The Wavefront .obj File Format

In order to achieve SVG files where the silhouettes are properly grouped based on which triangle mesh they belong to from the original 3D scene, the triangles must be grouped into meshes in the input 3D scene, as explained in 1.3. This will keep the scene organized, and reduce a potentially high amount of triangles into fewer, larger silhouettes. This will in turn facilitate editing of the resulting vector graphics, because the user will not have to treat each triangle in the scene as an individual silhouette. The input 3D scene will consist of Wavefront .obj files, because it supports this grouping with the (g) tag. The Wavefront .obj file format is a common standard for defining 3D objects and is therefore also widely supported. As described by Bourke [17], it is a file format that was initially created by Wavefront for its Advanced Visualizer software, and is used to define the geometry and other properties for graphical objects. Geometry defined in this file format is based upon the general properties of triangle meshes, as explained in Section 2.3. The Wavefront .obj file format is one of several ways of defining such meshes.

2.4.1 Structure

In the Wavefront .obj format, one or more graphical objects are defined by a list of vertices. A list of faces (triangles if the model has been triangulated) will index them together by taking the index of each vertex to create each of the corners of the face. Each face may have three or more vertices. [17]

The code snippet below shows an excerpt of an example .obj file, where each vertex is represented with a "v" followed by three floats specifying its position in 3D space (x- y- and z-coordinate, respectively). An "f" indicates a face, where the used index of the vertex can be followed by the index of its corresponding texture coordinate and vertex normal, separated by slashes. Because texture coordinates and vertex normals are unnecessary in this thesis, they are excluded from this example. The faces may be divided into groups with the "g" prefix. All the faces in this example are triangles, but they can also have more than three vertices.

This example also uses a Material Library (.mtl) file, as indicated by "mtllib" on the first line, and "usemtl" indicating that a specific material from the Material Library file should be applied to a particular group of faces. The material defines the appearance of a mesh, e.g. the mesh color or the texture used on the mesh. Material Library files are explained further in Section 2.4.2.

```
mtllib materialfile.mtl
g GrassPlane
v -3.629571 -0.947346 4.723222
v -3.471317 -0.923668 3.736110
v -3.259937 -0.857212 3.749801
...

usemtl RockMaterial
f 48// 16// 15//
f 1// 32// 65//
f 1// 65// 34//
...
```

Listing 2.2: An example .obj file.

An .obj file may contain several additional fields that are not explained here due to their irrelevance to this thesis, like vertex normals and texture coordinates. Although imported .obj files may contain these fields, they are not necessary to be used in this thesis as they are used for textures and shading.

2.4.2 Material Library (.mtl) files

A Material Library (.mtl) file must be provided with the .obj file for the faces to have colors. As described by Ramey et. al. [18], they contain definitions of one or more materials that are applied to an object's surfaces and vertices. Materials can be defined by colors, textures and reflection maps, but only the color attributes are necessary for this thesis.

The snippet below shows an excerpt from an example .mtl file, where a material is first declared with "newmtl" followed by the material's name. This particular material contains three different attributes that describe the reflectivity of the material with RGB values; "Ka" represents ambient reflectivity, "Kd" represents diffuse reflectivity and "Ks" represents specular reflectivity. The "illum" statement indicates the number of the illumination model (lighting and shading effects) that the material uses. There are 11 different illumination models, and each one determines which lightning and shading settings are turned on or off.[18]

```
newmtl EarthMaterial
Ns 96.078431
Ka 1.000000 1.000000 1.000000
Kd 0.640000 0.640000 0.640000
Ks 0.500000 0.500000 0.500000
Ke 0.584000 0.584000 0.584000
d 1.000000
illum 2
```

Listing 2.3: An example .mtl file.

2.5 Choosing a development tool

The choice of which tool to use for developing the solution comes down to two main categories. Since the topic of the thesis is computer graphics, it can either be implemented with a graphics API like OpenGL, DirectX or Vulkan, or with a game engine like Unity or Unreal Engine. An API is a set of functions or commands that can be used to create software or interact with an external system [19]. The choice of which of the two tools to use comes down to how low-level or high-level the development has to be, and which choice that can reach the thesis goal the fastest. The following subsections explain the two tools of each category that was evaluated, and the reasons why one of them was chosen. The evaluated graphics API and game engine were OpenGL and Unity, respectively.

2.5.1 Graphics API (OpenGL)

OpenGL, or "Open Graphics Library", as described by Segal et. al. [20], is an interface for software to communicate with graphics hardware. It consists of procedures and functions that can be used in software to create computer graphics in either 2D or 3D, and render them to the screen. Graphics are rendered to the screen by first opening a window into the framebuffer, which can be considered the "canvas" on which the graphics are drawn. This window must then be associated with the GL context, in order to have the possibility to issue OpenGL commands. Further, the developer must specify geometry and generate buffers to allocate and initialize memory. Buffers are then filled with data like geometry and colors that is rendered to the framebuffer for presentation on the screen [20].

2.5.2 Game engine (Unity)

Unity is a cross-platform game engine that supports the development of games in 2D and 3D. Its editor lets the developer apply game logics through its scripting API in C#, and build scenes (including user interfaces) with either self made assets or assets downloaded through the Asset Store. Assets may be 2D and 3D models, audio, shaders and tools. The developer can at any time run the scene by pressing the Play button [21].

2.5.3 Choosing Unity

An advantage with using a game engine like Unity is that it provides a framework where parts of the software are already implemented through its scripting API. This can get a project up and running significantly faster than when using a graphics API directly, like OpenGL. Working directly on a graphics API requires the developer to build the project more from the ground up, like programming shaders and storing data in buffer objects. Although game engines may use such graphics APIs themselves, they limit the developer to working within the boundaries of their own scripting API. Working directly with graphics APIs opens up for more freedom as to how the graphics are rendered.

After evaluating the two options, Unity was chosen for implementing the thesis solution. The main reason for this is that a vital component needed for the solution, layered depth images (LDI), was recently implemented in Unity by another master student. Once the layered depth images of the scene have been rendered, the remaining work will revolve around processing these images, and no longer the issue of rendering graphics. Another advantage of using Unity for this thesis is that its editor already provides an interface where the user easily can construct a scene and adjust the camera perspective. This ensures a good user experience, and makes it easier and faster to construct and test different scenes during development of the solution. Lastly, implementing the solution in Unity means that any application and video game that is also implemented in Unity, easily can import this solution's scripts and let the user or player of the software capture SVG screenshots of what they see on the screen.

2.6 Layered Depth Images

When 3D scenes are usually rendered, the rendering pipeline will disregard any surface that is invisible from the given viewpoint with a step called Z-buffering, in order to increase the rendering efficiency. Z-buffering is used to achieve correct occlusion where objects in front are drawn over the objects behind. A two-dimensional array called the Z buffer stores the Z-value of each pixel, overwriting it every time a fragment closer to the camera is found for the given pixel. [22]

With layered depth images (LDI) [23], the stored pixels are not limited to those with the lowest Z-value. Instead, the Z-value of every triangle that is drawn in the pixel is stored. LDIs are created at the same time as the rendering happens. This means that LDIs do not take noteworthy extra time to create, because each triangle has to go through the rendering pipeline regardless, and be checked against the Z buffer to get the correct occlusion. Each pixel is assigned a depth value in addition to its color, allowing the layered depth images to contain multiple pixels per pixel location. This makes it possible to maintain information about all the surfaces in the scene, including those that are hidden by other surfaces. The layered depth images used in this thesis is implemented in the form of a per-pixel linked list.

2.6.1 Per-Pixel Linked Lists

In order to obtain the layered depth images, a per-pixel linked list can be implemented in a fragment shader program to run on the Graphical Processing Unit (GPU). The key concept in a per-pixel linked list is to keep a linked list for every pixel on the screen, and store every fragment that occurs during rendering in each particular pixel. To achieve this, two different buffers are used, namely the Head Pointer Buffer and the Node Buffer. A counter is also needed to keep track of the positions in the Node Buffer each node is stored in.

Figures 2.5, 2.6 and 2.7 and the following explanation and examples of per-pixel linked lists are collected from Giske's master thesis [24] and originate from McKee's presentation on the same topic [25]. The left side of Figure 2.5 shows the pixel grid of the render target. The purpose of the Head Pointer Buffer on the top right is to store the index positions of the list head for each pixel. The Node Buffer contains the nodes themselves, that each contain all the properties of a fragment like color, depth and a pointer to its next node in the given pixel's linked list. A counter keeps track of the total number of nodes, and is used to index each new node with the correct value in the Head Pointer Buffer.

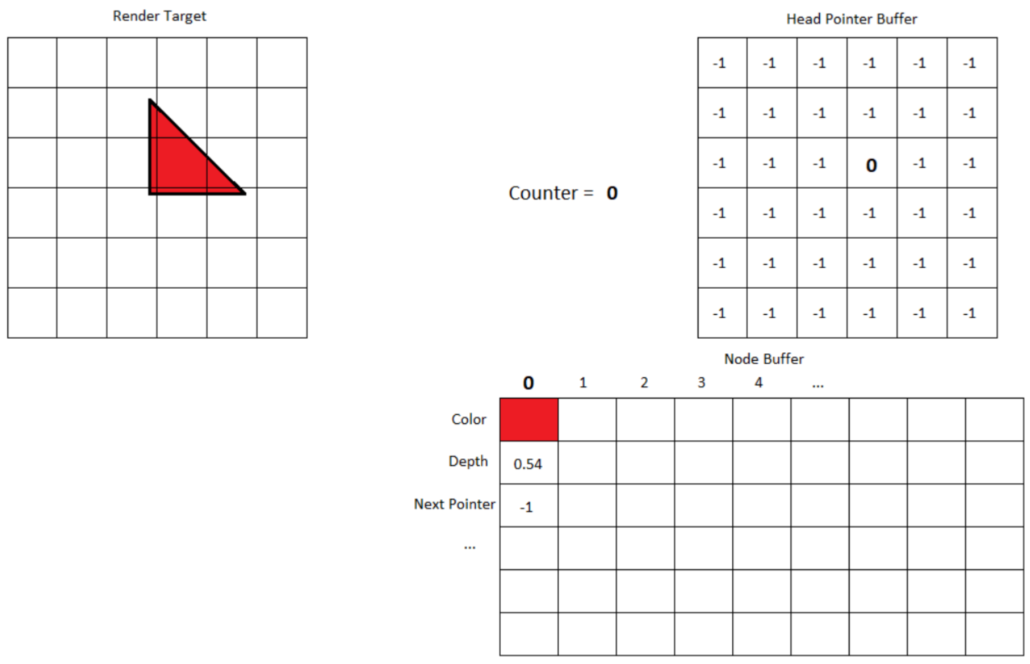
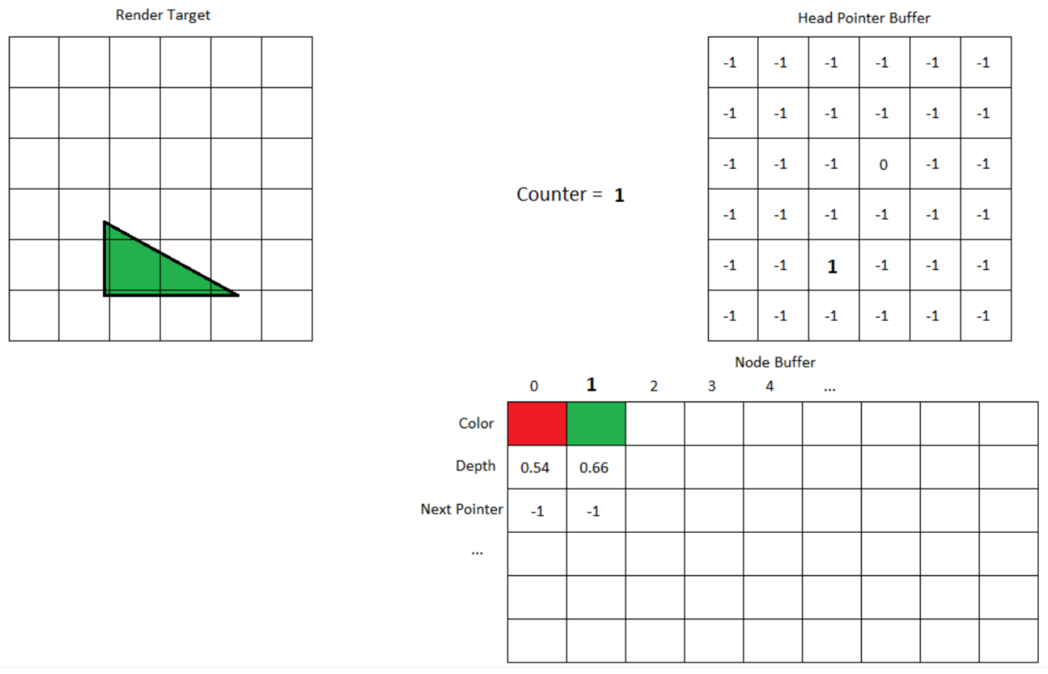


Figure 2.5: The first fragment in the per-pixel linked list [24]

.

Before the per-pixel linked list is filled, the Node Buffer is empty and all the positions in the Head Pointer Buffer have the value -1, which is the end of each linked list. When the first fragment in the scene occurs in a given pixel position during rendering (the red fragment in Figure 2.5), it is placed as a node at index 0 (counter value) in the Node Buffer, containing its color (red), depth (0.54) and a pointer to the next node in the pixel's linked list (-1 as this is currently the only one). The Head Pointer Buffer position of the list is updated to 0, since this new node is now the new head, and the counter is incremented to 1.

The second fragment (Figure 2.6) occurs in a different pixel position than the previous one. It is added as a node with its color (green) and depth (0.66) at position 1 in the Node Buffer. Its next pointer is -1 as there are no other nodes currently in this pixel position. Its position in the Head Pointer Buffer is changed from -1 to the new node's position in the Node Buffer, which is 1. The counter is incremented to 2.



Head Pointer Buffer

-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	0	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	**1**	-1	-1	-1
-1	-1	-1	-1	-1	-1

Node Buffer

	0	**1**	2	3	4	...			
Color	🟥	🟩							
Depth	0.54	0.66							
Next Pointer	-1	-1							
...									

Figure 2.6: The second fragment in the per-pixel linked list [24].

The third fragment (Figure 2.7) occurs in the same pixel position as the first fragment. Its next pointer must therefore be 0, as this is the Node Buffer position of the red fragment in this pixel position. The fragment's color (blue) and depth (0.25) is also stored in the node. The node is stored at position 2 in the Node Buffer, the value in the Head Pointer Buffer is updated from 0 to 2 and the counter is incremented to 3.

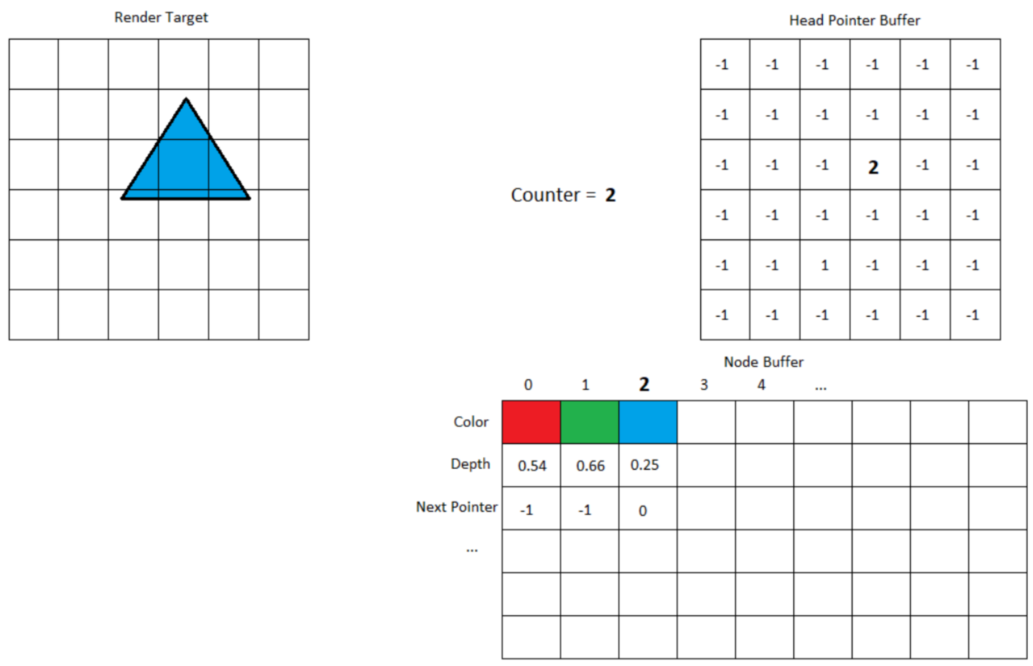


Figure 2.7: The third fragment in the per-pixel linked list [24].

This process continues until all the fragments in the scene are stored in the per-pixel linked list. To get the list for a specific pixel, one must first calculate the pixel position for the given linked list to find its corresponding position in the Head Pointer Buffer. The linked list's head node can be loaded from the Node Buffer by using the index value returned from the Head Pointer Buffer in this position. The list can then be iterated through by loading from the Node Buffer with the current node's next pointer, to get access to the deeper depth layers in the scene. The end of the list is reached when the next pointer of the current node is -1. [24]

Per-pixel linked lists are often used to achieve order independent transparency, where only the transparent fragments are stored and blended together in an order independent way, to appear transparent [26]. However, to maintain a layered structure of all the geometry in the scene, the solution in this thesis will store all the fragments and not only the transparent ones. The linked lists should also be sorted by depth to make sure that the layers in the resulting vector graphics are in the correct order.

Chapter 3

Design and Solution

3.1 General design

The flowchart in Figure 3.1 shows the different stages the finished solution goes through, from acquiring the meshes in the 3D scene to drawing them as vector graphics. Once the 3D meshes have been acquired, an algorithm will flatten the 3D meshes into a collection of 2D silhouettes better suited for a vector graphics representation. This is done by organizing the scene into several depth layers, extracting the outlines of each mesh in each depth layer, and using these outlines to drawing silhouettes of the meshes that can be output as 2D vector graphics. An outline is a line that identifies the boundaries of the mesh, and is hereby referred to as a "contour" [27]. Each depth layer must be in the correct order of depth in the vector graphics illustration.

The main contribution of this thesis is an algorithm that receives a 3D scene, and converts it into vector graphics from a given viewpoint. An important step for being able to obtain all the depth layers in the scene, is rendering the scene into layered depth images. A depth layer in the context of this thesis is an individual image in the LDI. It shows the colors of the fragments that are in the same given index of the per-pixel linked list, in every pixel position. This will be explained in more detail in Section 3.3. Another important factor is to make sure that the scene's triangles are structured into meshes, so that all silhouettes that originate from the same triangle mesh are grouped together. That is further explained in Section 3.2. When the .obj files have been imported into the Unity scene, the import logic uses the group tags in each .obj file to structure the triangles into different meshes. This grouping information needs to be maintained when the scene has been rendered to LDI and the vector graphics are to be drawn. Storing

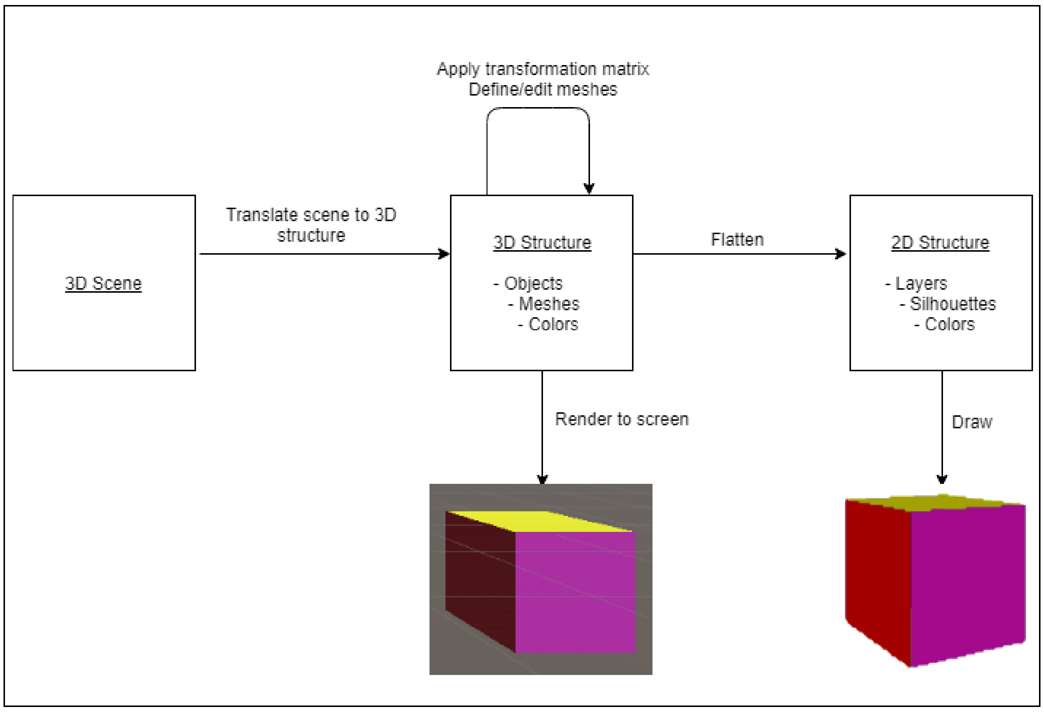


Figure 3.1: A flowchart illustrating the process of converting a 3D scene into vector graphics.

this in the LDI, that in essence is a collection of rendered images, is difficult. This suggests a need to render the scene two times; one for identifying which pixels belong to which mesh by assigning temporary ID tag colors to the triangles (see Subsection 3.5.1), and one for the actual SVG drawing (see Section 3.4). Figure 3.2 describes these main stages of the algorithm that is the main contribution of this thesis.

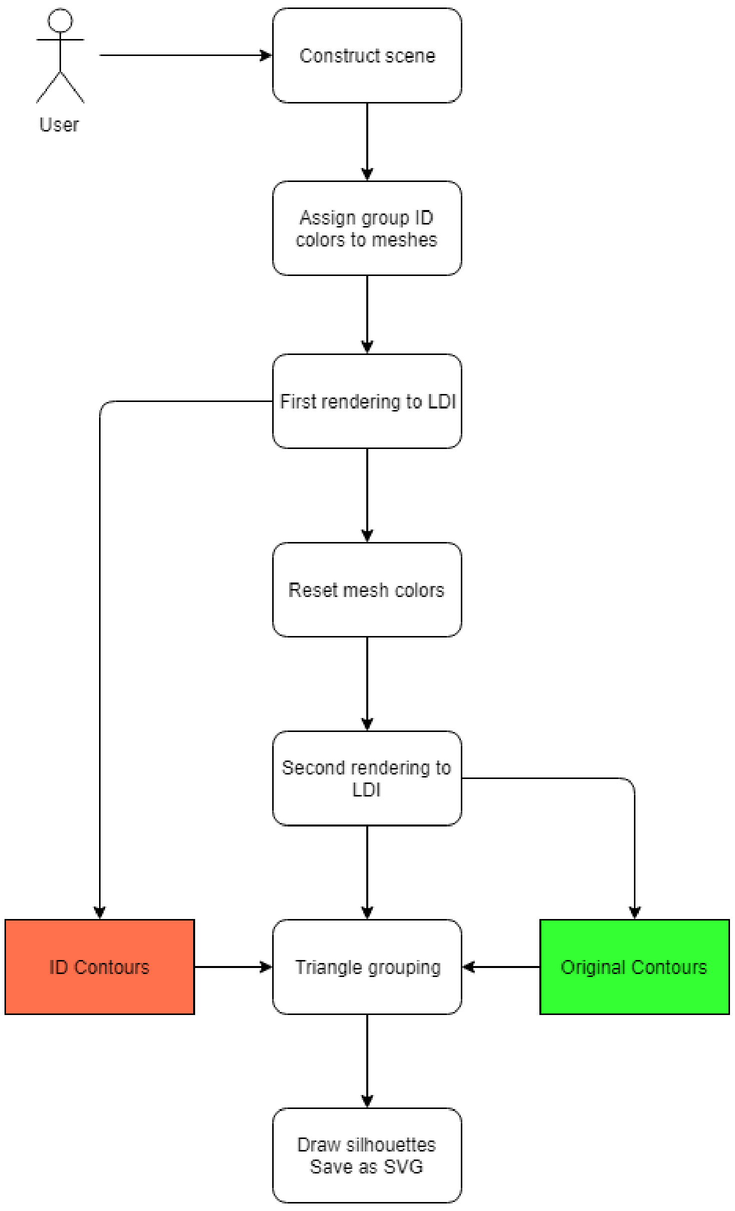


Figure 3.2: A diagram illustrating each step of the designed solution.

In order to identify the meshes in the scene, the algorithm implements a "hack" that uses the color attributes stored in the first LDI as an ID tag for the meshes, called "ID tag colors". The algorithm assigns a temporary ID tag color to every triangle in every mesh, so that each mesh has one single color to separate them from each other. It then renders the scene to the first LDI to create the "ID LDI" in the scene, where each mesh can be identified by its own color. "ID contours" are obtained by extracting the outlines of each mesh in this step, and are stored in a single collection for the entire ID LDI. See Section 2.6 for an explanation of LDIs.

In the next step, the rendering of what actually becomes the vector graphics is performed. All the triangles of each mesh are first assigned their original colors, removing the "ID tag colors" they were assigned in the previous step. The scene is rendered again into a new LDI, and a collection of contours is extracted as in the previous rendering, called "original contours". The difference is that the result will be separate contours for each region in a mesh based on color. If a mesh consists of multiple different colors, one contour for each of the colored regions in the mesh will be obtained, resulting in multiple contours for that mesh.

Once the ID contours and original contours have been extracted from all the layers in the two LDIs, the original contours can be grouped together so that all the original contours that belong to the same mesh are regarded as a single group. This grouping process is explained in Subsection 3.5.2. With every original contour organized into groups, the algorithm can go through each group in each depth layer and draw them as silhouettes (see Section 3.6), that can in turn be written into an SVG document (see Section 3.7).

Figure 3.3 shows an example of all the key stages of a scene being converted from 3D to 2D. The 3D scene containing a cube (upper left) is rendered to an LDI, which has several depth layers like the one in the figure (upper right). Contours (lower left) are extracted from each such depth layer, and is drawn into a silhouette in 2D (lower right).

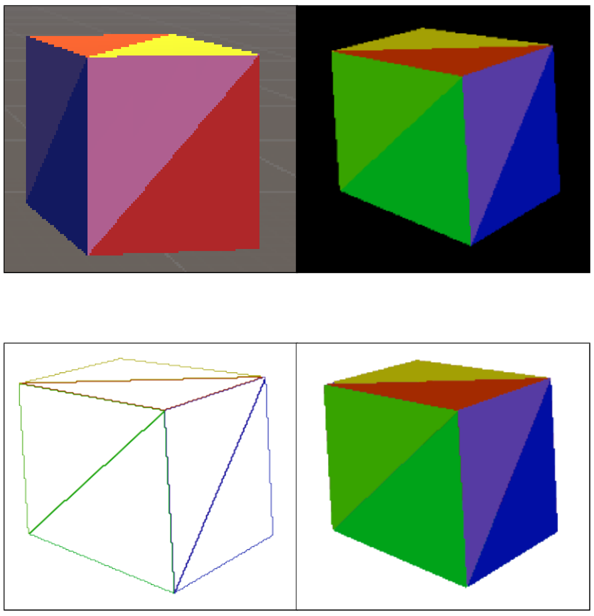


Figure 3.3: Upper left: A 3D scene. Upper right: A depth layer of the 3D scene. Lower left: Contours extracted from the depth layer. Lower right: A silhouette drawn from the contours.

This section has presented the general design of the algorithm that converts a 3D scene into editable 2D vector graphics. Each of the steps introduced here are explained in more detail in the following sections.

3.2 Constructing the scene

The user first imports a collection of .obj files along with their corresponding .mtl files into the scene. Every .obj file is converted into a Unity GameObject, which is the base class for all entities in Unity scenes [28]. These components are assigned their correct material colors according to their corresponding .mtl files. The .obj file is organized into groups of faces with the group (g) tag before import, so that the import logic of Unity can organize all the triangles of each group into a mesh. In this way, a GameObject will have each of its meshes as a child.

Figure 3.4 shows how the input .obj files and .mtl files are organized in order to have the necessary structure for grouping the meshes. The scene can contain one or more parent 3D objects (.obj files), each having one or more meshes as children. Each mesh contains one or several faces, and each face has a collection of vertices that define its position and shape. Each mesh also has one or more materials obtained from the parent .obj file's .mtl file, containing a color.

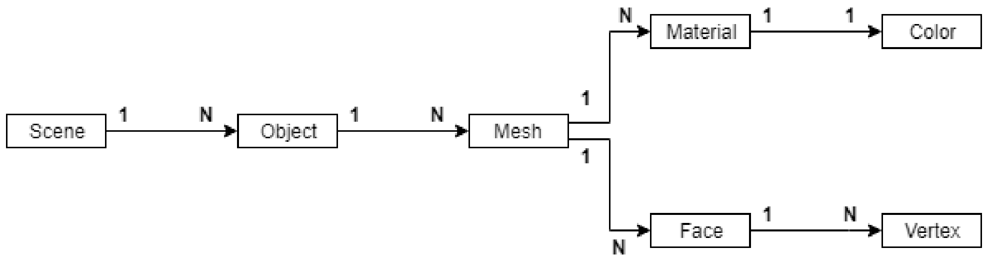


Figure 3.4: The structure of the input 3D scene.

Once the input .obj and .mtl files have been imported with the structure as described in Figure 3.4, every object in the scene can be positioned, rotated and scaled in the Unity editor by the user. Once the user is satisfied with the scene, the user can adjust the camera in the Unity editor. It is also possible to adjust the rendering resolution by entering "Game" mode and clicking "Aspect", allowing the user to choose an aspect ratio or resolution from a drop down menu, or creating their custom one. Although a high resolution results in a smooth SVG result, it is important to keep in mind that the higher the resolution, the more time is required to process the scene. This is because the silhouette drawing algorithm has to process more pixels. The resulting SVG files will also be larger if the resolution is high, because in the solution's current state, more pixels means more points to draw.

Figure 3.5 shows an example of a scene that has been constructed in Unity. For simplicity the scene only contains a cube that was made in Blender and exported as an .obj file. The cube has six sides where each side consists of two triangles, each with its unique color. In its .obj file, each side of the cube (or pair of triangles) is regarded as a separate mesh. It is important that, to facilitate editability in the finished vector graphics result, the user should not have to pick and edit the individual triangles for each side. The two triangles of each side should therefore be grouped together also in the later stages of the solution.

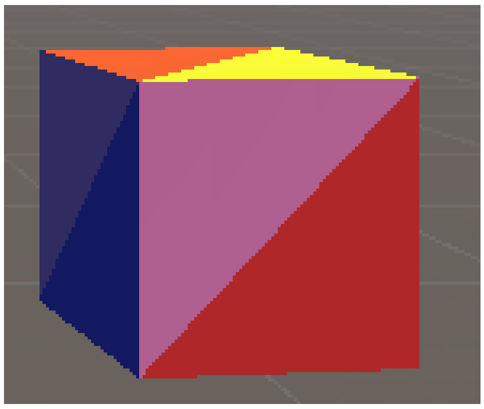


Figure 3.5: A 3D model of a cube loaded into Unity, where each of its triangles have different colors and the six sides are separate meshes.

In the Unity editor, the user can add, remove, position, rotate and scale any object as well as changing the viewpoint of the camera. Once the user is done with constructing the scene, the solution can be run by clicking the "Play" button in the Unity editor. The scripts will then start the process of converting the scene into vector graphics from the perspective of which it is shown on the screen.

3.3 Acquiring the scene's depth layers

When the user has constructed the scene and pressed the Play button in the Unity Editor, the scene is first rendered into a per-pixel linked list as described in Section 2.6.1. The implementation of the per-pixel linked list has been provided by Giske [29]. The solution then iterates through each layer in the per-pixel linked list, starting with the foremost layer and working its way through until there are no more nodes to visit (indicated by -1 in each linked list). The color of the current node is returned by a fragment shader for output into an image texture, resulting in one image texture per depth layer. Two examples of such image textures are shown in Figure 3.6 showing the two depth layers of the cube introduced in Figure 3.5.

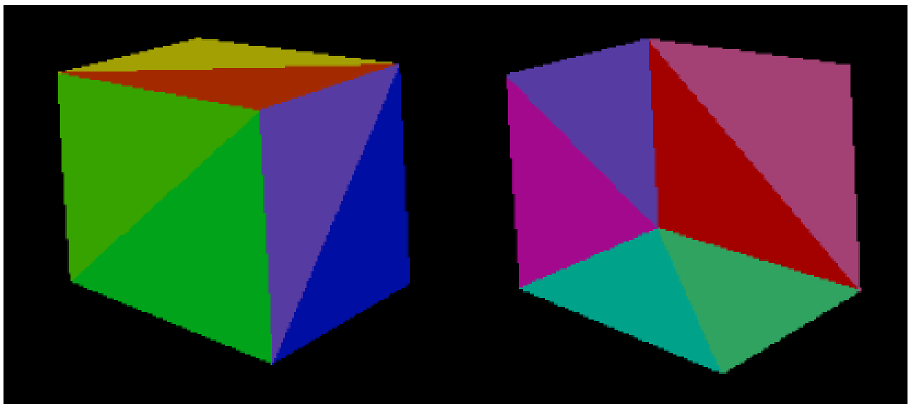


Figure 3.6: The two resulting depth layers when rendering a cube to LDI, with depth layer 1 (front side) to the left and depth layer 2 (back side) to the right.

An image texture for each depth layer is stored as a PNG file in the Unity project's Asset folder. The solution will later extract the mesh contours of these image files and draw their silhouettes.

3.3.1 Sorting the per-pixel linked lists

Although the previous thesis by Giske [24] already provides a finished solution for rendering a scene to LDI through a per-pixel linked list, these linked lists are not sorted by depth. This lack of sorting can in many cases lead to unexpected and flawed results, because the fragments are added to the linked lists in the order of which they occur during rendering. This order is not always correct in regards to the fragments being sorted by depth, from lowest to highest, which is what is needed for the finished vector graphics layers to be drawn in the correct order (back to front). This is particularly true for self-occluding meshes and mesh chains, described in Section 2.3.1, where the order of their fragments could appear in many different orders regardless of the order of depth.

An example of a self-occluding mesh is shown in Figure 3.7, which shows a simple cube. This cube is different from the other cube used in this chapter, because its sides are not divided into separate meshes. Instead the entire cube is regarded as a single mesh, making it self-occluding.

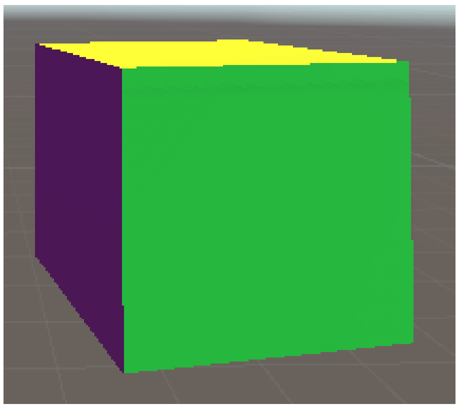


Figure 3.7: A simple cube where all its sides are regarded as a single mesh, making it self-occluding.

The result of rendering this cube to an LDI is shown in Figure 3.8. As the result shows, this rendering does not give a correct order of the fragments of the two depth layers with regards to their depth values. This suggests that a sorting of the nodes by depth should be implemented for the per-pixel linked list, either by sorting the lists as they are created, or after. Another solution is to let the lists be created as usual, but traversing the fragments in each list in the correct order of depth. The latter method is the one that is implemented in this solution.

This way of traversing the fragments in each list in the correct order, is as follows. In the fragment shader that outputs the color of a node to an image texture, the correct head node of the linked list is loaded as usual by

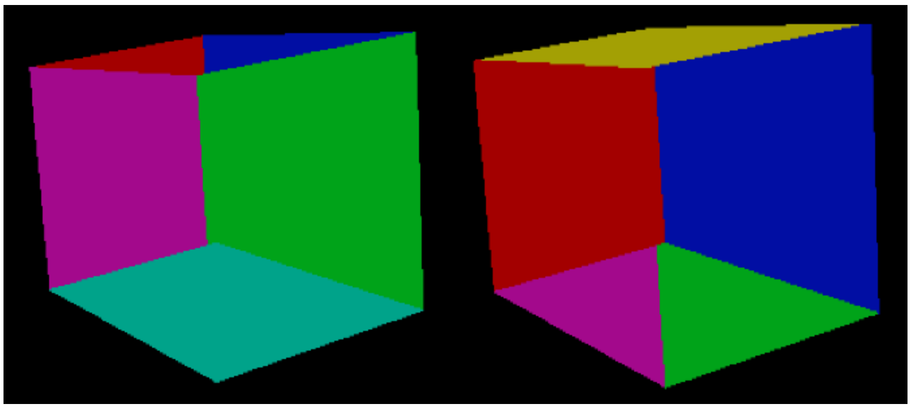


Figure 3.8: The result of rendering a self-occluding cube to an LDI. The left shape is the foremost depth layer while the right shape is the hindmost depth layer.

first using the pixel position to calculate the correct position in the Head Pointer Buffer, and then loading from the Node Buffer with this position as an argument [24]. However, instead of simply iterating through the nodes in the list in the order of which they were created, the fragment shader should iterate through the list of nodes and return the color of the node with the lowest depth value. This node is then marked as visited by using an added boolean attribute. In the next fragment shader call in the same pixel (the next depth layer), the visited node is ignored and the color of the second closest node is returned.

Because the head node could have been marked as visited in an earlier fragment shader call, the algorithm cannot always use this as a start node. Instead, it must keep the head as a temporary start, and iterate through the linked list until the first unvisited node has been found. Starting from this node, the algorithm iterates through the rest of the linked list, always keeping the currently closest node and updating it every time an unvisited node with lower depth value occurs. After going through the linked list in this way, the fragment shader can return the color of the correct node.

Figure 3.9 shows the result of using the above steps to always find the unvisited fragment that is closest to the camera. The depth layers now display the colors in the correct order. However, as can be seen in the figure, a few pixels around the edges of the cube sides in the left depth layer show the color of the pixel in the next depth layer instead of the current one. This is likely due to the depth values being so close to each other where two cube sides are intersecting, that they output the wrong color.

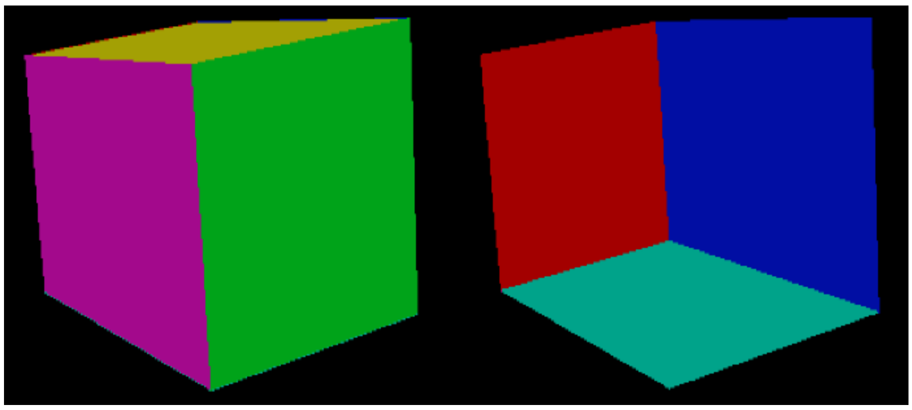


Figure 3.9: The result of rendering a self-occluding cube to an LDI after sorting the order of the fragments in each pixel's linked list.

3.4 Extracting mesh contours

Once the LDI of the scene has been obtained, the solution can start extracting the contours of each depth layer for the purpose of creating silhouettes in 2D. For each depth layer, the algorithm creates an image texture (Texture2D in Unity) of the depth layer, as mentioned in 3.3. For each of these image textures, the algorithm iterates through each pixel, checking if the given pixel is a part of an outline. This is the case if the color value of any neighboring pixel is either empty or different from that of the current pixel, indicating that the current pixel is a part of the outline. In any such case, the pixel color is stored together with its x and y coordinates to create a separate texture with all the contours from that depth layer.

The implementation of this step is shown in the code listing below. The function "IsEdgePixel" takes the image texture and the color of the current pixel, and returns "true" if any of its neighboring pixels has a different color than that of the current pixel, indicating that the pixel is at the edge of a mesh (or a part of a mesh if the mesh has multiple colors). Every time a new edge pixel is found, a new contour pixel is created, consisting of the color and coordinates of the current pixel. It is also marked as unvisited, which is necessary for the vector graphics drawing step (see Section 3.6). The function "FoundContour" searches through the list of contours that have currently been found, and returns it if it exists. In the case that it exists, it means that this is the contour that the new contour pixel belongs to, and the new contour pixel is added to this contour. If the function returns "null", it means that the new contour pixel is a part of a new contour, and a new contour object with that pixel is created and added to the list of found contours. The total count of pixels in a contour is incremented when a new pixel is added, which is necessary for the drawing of the vector graphics later. As mentioned in Section 3.1, this method for extracting contours is performed two times in the solution; first for extracting ID contours for grouping, and then once more for extracting the original contours that will be used for drawing the vector graphics.

```
Color c = texture.GetPixel(pX, pY);
if (c != Color.black)
{
    if (IsEdgePixel(texture, c, pX, pY))
    {
        Contour foundContour = FoundContour(contours, c);
        ContourPixel contourPixel = new ContourPixel
        {
            Color = c,
            XCoord = pX,
            YCoord = pY,
            Visited = false
        };

        if (foundContour == null)
        {
            foundContour = new Contour(Width, Height);

            foundContour.Pixels[pX, pY] = contourPixel;
            foundContour.Color = c;
            foundContour.NumberOfPixels++;
            contours.Add(foundContour);
        }

        foundContour.Pixels[pX, pY] = contourPixel;
        foundContour.Color = c;
        foundContour.NumberOfPixels++;
    }
}
```

Listing 3.1: Code for extracting contours out of an image texture.

Figure 3.10 shows an example of the contours extracted from the front-most depth layer that is the result of rendering the cube to an LDI.

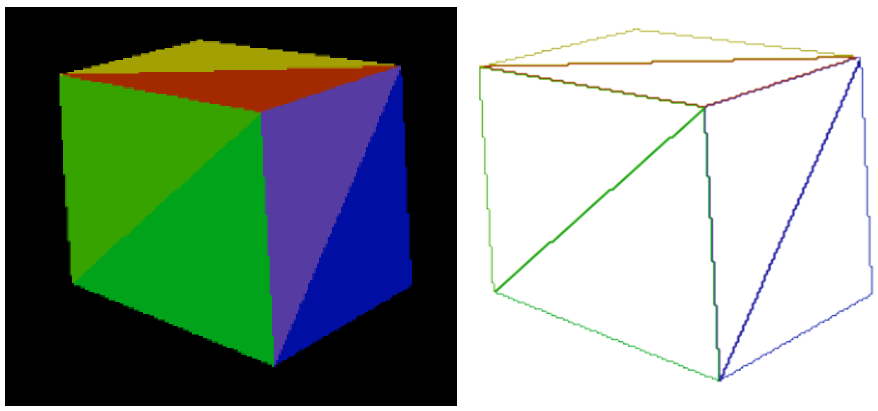


Figure 3.10: Left: The frontmost depth layer of a cube. Right: The contours extracted from the depth layer.

3.5 Identifying meshes and grouping mesh triangles

3.5.1 Identifying mesh IDs in the LDI

As mentioned in Section 3.1, the solution must first tag all the triangles of each mesh in the scene with an ID tag color, so that the meshes can be distinguished from each other, and triangles belonging to the same mesh can be grouped together. The ID tag colors are determined by creating an RGBA color with random float values between 0 and 1 for each of the four values. Figure 3.11 shows a depth layer of the cube with its temporary ID tag colors for each of its three visible sides to the left. To the right is a contour image texture that is the result of rendering this cube and extracting the ID contours out of its image texture. Notice how both triangles for each side of the cube has the same color, and how this is reflected in its contours.

3.5.2 Grouping the mesh contours

The next step is to render the scene again, but with its original colors instead of the temporary ID tag colors, like the cube in Figure 3.12. The contours are then extracted from the depth layers in the same way that they were extracted in Subsection 3.5.1. However, the resulting contours are different from the previous step as they are now based on the original colors of the scene. For the cube example in Figure 3.12, this means that each of its three

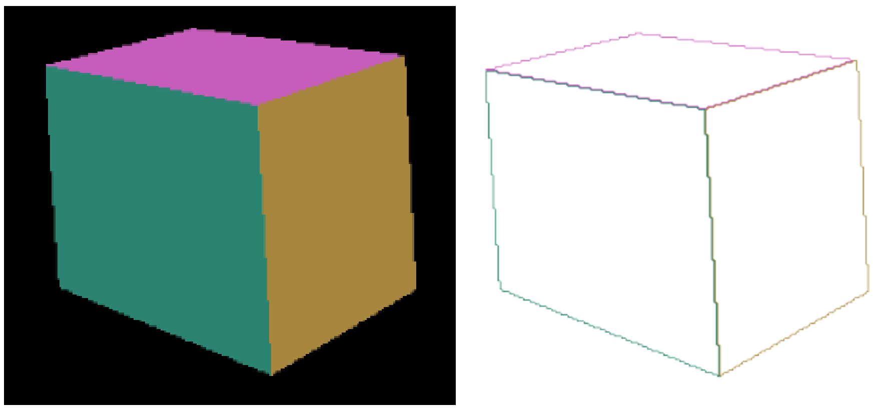


Figure 3.11: A depth layer with ID tag colors (left) and its corresponding ID contours (right).

visible sides are divided into separate contours, two for each side, as each side has two triangles of different colors.

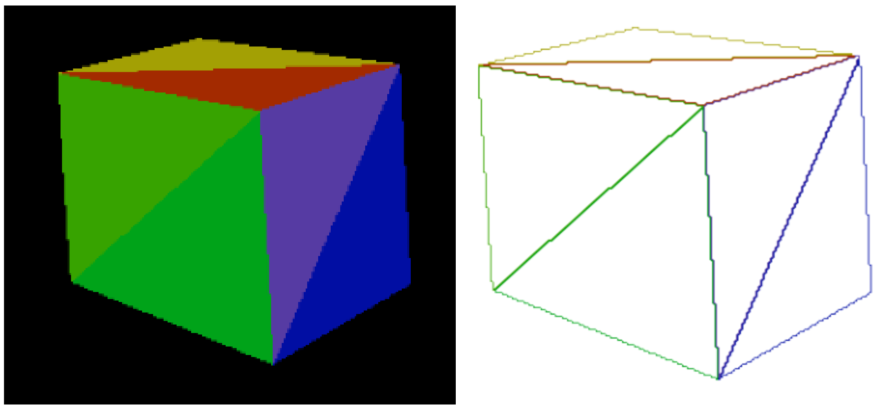


Figure 3.12: A depth layer (left) and its corresponding original contours (right).

Now that all the ID contours and original contours have been extracted from the scene, the ID contours must be used to group the original contours together. For a set of original contours to be grouped together, they must belong to the same mesh as indicated by the ID contours. To match an original contour with an ID contour, the algorithm takes a pixel sample from the original contour and iterates through all the ID contours in attempt to find the correct contour based on the pixel's coordinates. Figure 3.13 shows an example ID contour (marked in green) and the three different cases that can occur when checking the pixel in an original contour. In case A, the original contour pixel's coordinates are exactly on top of the ID contour, meaning that the original contour that the pixel belongs to is in fact part of this ID contour. The same also applies in case B, where the pixel is placed inside the ID contour. If the pixel is situated outside the ID contour as in case C, the algorithm concludes that this is the wrong ID contour, and goes on to check the next one. This check is performed for each ID contour with the same pixel sample from the original contour until the correct ID contour is found.

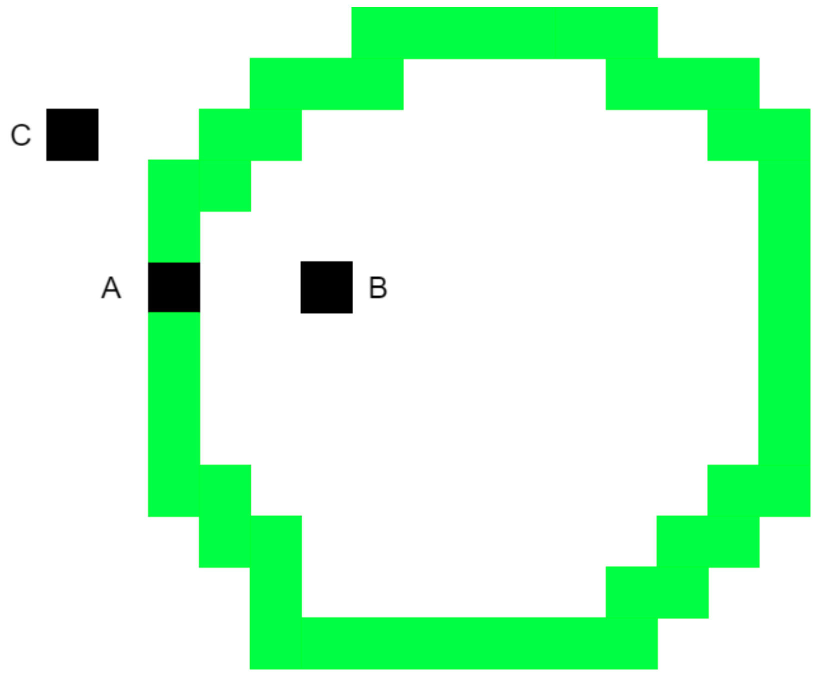


Figure 3.13: The three different cases that can occur when checking a pixel sample from an original (black) contour against an ID contour (green).

In the case where the pixel is placed exactly on top of an ID contour (case A), the algorithm can immediately conclude that the pixel's original contour should be grouped based on this particular ID contour. However, to conclude with either case B or C, a slightly more complicated procedure is needed. To safely determine that a pixel is in fact outside or inside an ID contour, four different conditions must be true. They are illustrated in Figure 3.14. The four conditions are checked as the algorithm iterates through all the pixels in the given ID contour. In Figure 3.14, (x2, y2) are the coordinates of the ID contour pixel that is checked. For a pixel with coordinates (x, y) to be inside an ID contour, the ID contour must have pixels that are above (y2 > y), below (y2 < y), to the left (x2 < x) and to the right (x2 > x) of the pixel. In other words, these four conditions combined being true means that the pixel is surrounded by the ID contour, and that the given original contour is within this particular ID contour. If all these four conditions are met, the algorithm concludes that the correct ID contour has been found. If one or more of them are not met, this means that the pixel is outside of the given ID contour.

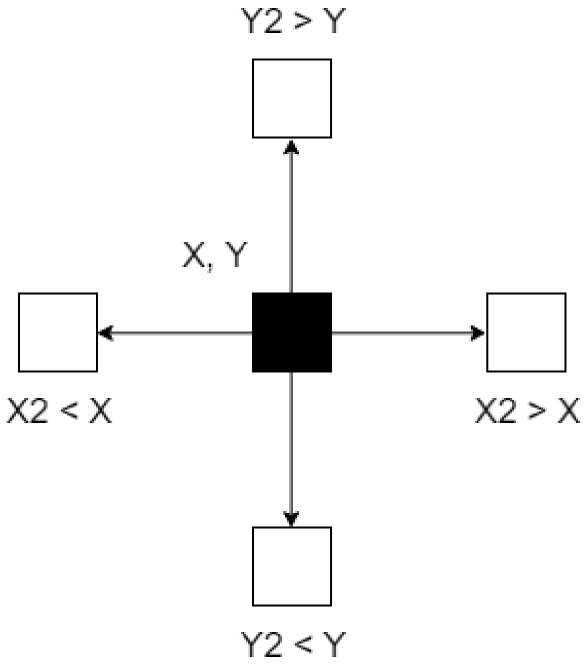


Figure 3.14: The four conditions that must be true for the pixel sample (x, y) to be inside a certain ID contour.

3.6 Drawing silhouettes

All the original contours in every depth layer have now been extracted and grouped based on the ID contours, and the algorithm can use these groups of original contours as input to draw the final silhouettes that are to be used as the final vector graphics output. To draw a silhouette from a contour, the solution first checks the color of the contour, so that the silhouette is drawn with the correct color. It then traces along the pixels in the contour, copying the coordinates of each pixel to a point that is added to its silhouette. Before the algorithm can start drawing a silhouette of the contour, a start pixel for the contour must first be found. This pixel represents the first point in the vector graphics silhouette, and is where the drawing should start on the SVG canvas. Starting from this pixel, every neighboring pixel around it is checked in order to find out where the next point in the silhouette should be. An illustration of the drawing procedure is shown in 3.15, where the algorithm starts from a start point on the corner of the light green triangle's contour and draws a silhouette along it. The dotted lines show the remaining part of the silhouette to draw at that certain point in time.

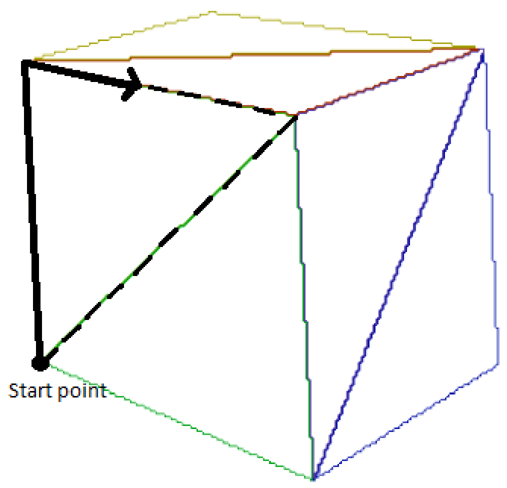


Figure 3.15: The drawing algorithm drawing a silhouette of the light green triangle from the cube's contour.

Figure 3.16 shows all the eight neighboring pixels around the currently visited pixel in a contour, all of which are possible next steps for the drawing of the silhouette at the given point in time. The current pixel is in position (x, y) at the center. A pixel is a neighbor of the current pixel if either or both of its coordinates are equal to the current pixel's coordinates, minus or plus 1. For instance, the neighboring left pixel is (x-1, y) relative to the current pixel. Similarly, the neighboring bottom right pixel is (x+1, y-1) relative to the current pixel.

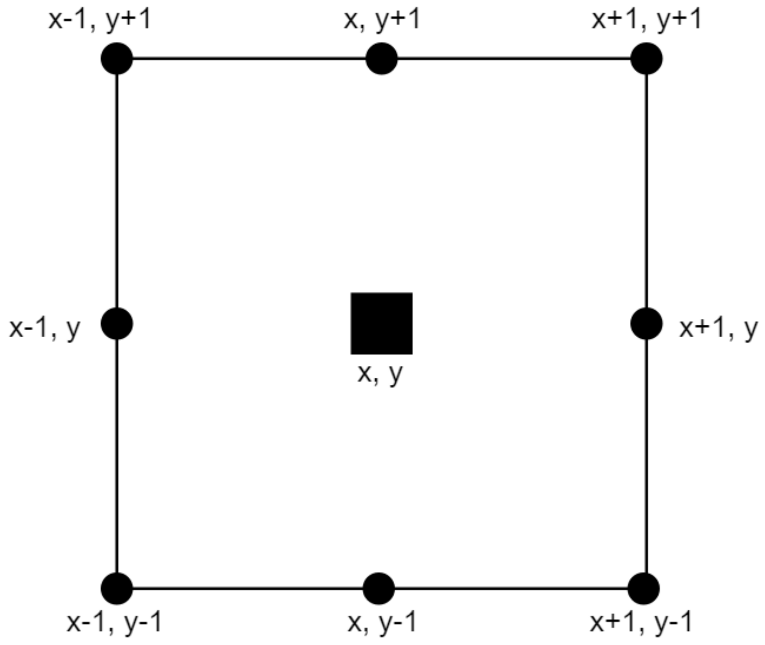


Figure 3.16: The eight possible next pixels to visit for the drawing algorithm at any point, from the current pixel (x, y).

If the given neighboring pixel has the same color as the silhouette and has not already been visited, a new silhouette point is created with the pixel's x and y coordinates and the pixel is marked as visited. This process continues until there are no more unvisited pixels to visit from the given current pixel. The algorithm will then find a new start pixel in the current contour and repeat the given stages as long as there are more unvisited pixels. The algorithm knows that there are more unvisited pixels if the number of visited pixels is less than the total amount of pixels in that contour. The reason for this repetition is that the contour of a mesh could be split into several pieces and is not necessarily connected. The drawing algorithm can also get "stuck" if it starts going on a wrong path due to there being more than one possible path to follow on the contour. This limitation in the drawing algorithm can result in silhouette holes, as explained in Subsection 3.6.2.

All the silhouettes of each depth layer have now been drawn. Although this solution starts with processing the frontmost depth layer, the order of the vector graphics layers must now be flipped so that the hindmost layer is processed first. The reason for this is that, as mentioned in 2.2.1, shapes in SVG are drawn on top of each other, implying that the vector graphics layers must be drawn back to front, starting with the hindmost layer. This solution however starts with processing the frontmost depth layer.

3.6.1 Handling screen boundaries

An important situation to consider when traversing along pixel paths in an image texture, that essentially is a 2D array, is that parts of the object whose contours are to be traversed along, can be outside of the view of the image. One example of this situation is Figure 3.17, where the lower part of the cube is outside the bounds of the image. If situations like these are left unhandled by the silhouette drawing algorithm, this leads to an IndexOutofRange exception. The reason for this is that the algorithm will attempt to check pixels that are outside of the image (array of pixels), when the edge of the image has been reached.

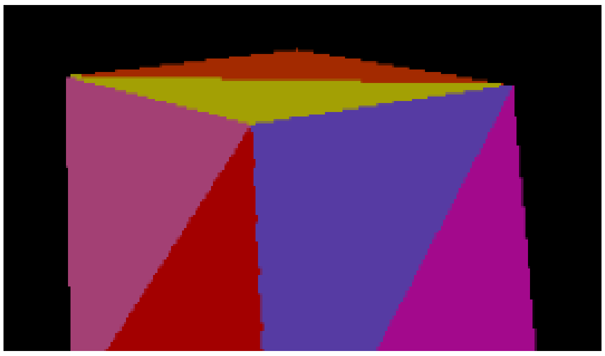


Figure 3.17: An example of an object causing an out of bounds error during silhouette drawing. Notice how its lower part is outside the boundaries of the image.

Since the screen has four sides, there are four different ways that reaching the boundaries of an image can happen. They are illustrated in an example in Figure 3.18. Given a screen of width 13 and height 7, the drawing algorithm can for instance attempt to reach a pixel with coordinates (-1,5) outside the left image edge, coordinates (14, 3) outside the right image edge, coordinates (5, 8) above the top image edge or coordinates (8, -1) below the bottom image edge.

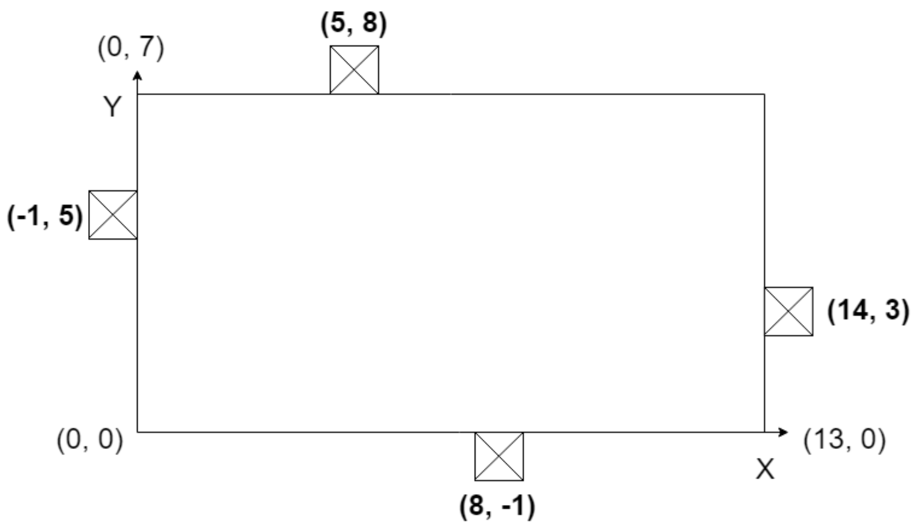


Figure 3.18: The four different cases where the silhouette drawing algorithm can go outside the bounds of the screen.

All these four cases will have to be checked and eventually handled every time the drawing algorithm has reached a new pixel and has to pick the next pixel to visit. The pixel position that the drawing algorithm is currently in has to be adjusted accordingly, depending on which of the four cases is true. Additionally, instead of checking all of the eight possible next pixels as described in Section 3.6, they are reduced to a certain few depending on which screen edge is reached. If for instance the y coordinate of the current index is larger than the screen pixel height, the left and right pixels should be checked instead. If one of them has a contour pixel that is not null and it is unvisited, the coordinate of the current pixel is adjusted accordingly. The same applies if the screen edge is below, left or right, but with different coordinates and adjustments. The three different actions are specified in Table 3.1.

Pixel position (relative to screen edge)	**Required action**
Above (y >screen pixel height)	Check left or right
Below (y = -1)	Check left or right
Left (x = -1)	Check above or right
Right (x >screen pixel width)	Check above or left

Table 3.1: The three different actions that can be taken when a shape is outside the screen edge.

3.6.2 Avoiding holes

A limitation with the algorithm that draws the silhouettes is that the silhouettes can get multiple holes in them, as shown in the orange and yellow plane in Figure 3.19. A "hole" in this context means that a silhouette is not drawn in a single, connected line. Instead it consists of several disconnected silhouette pieces, resulting in gaps between each piece. This is due to a limitation in the silhouette drawing algorithm which can cause the drawing to stop before the entire silhouette has been finished. When the drawing gets "stuck", meaning there are no more unvisited pixels to traverse to from the current pixel, the algorithm restarts the drawing at the first unvisited pixel it finds. While this does ensure that all necessary points in the silhouette are included, the drawn silhouettes are disconnected into several pieces, resulting in holes between them.

As shown in Figure 3.19, holes can lead to the fill color of the shape behaving unexpectedly when the shape is drawn as SVG, because the shapes that are filled are not closed. In this figure specifically, the orange triangle is divided into two pieces with two holes separating them. The holes are highlighted with black rings at the left and right edges of the plane. Figure 3.20 shows the highlighted areas in Figure 3.19 in more detail, where the silhouette pieces are disconnected. The fill color has been removed from the figure in order to expose the holes more clearly.

To reduce the number of such occurrences, an extra step to the drawing algorithm is implemented, that takes place after the silhouette is drawn, and before the silhouette is added to its correct group. This step is defined as "merging", and is done by using a threshold value (given by the user) to connect start or end points of two silhouette pieces if their distance from each other is within that threshold. More specifically, the step iterates through all possible combinations of the silhouette's pieces, looking for neighboring start or end points in each of the two pieces. Two start or end points are

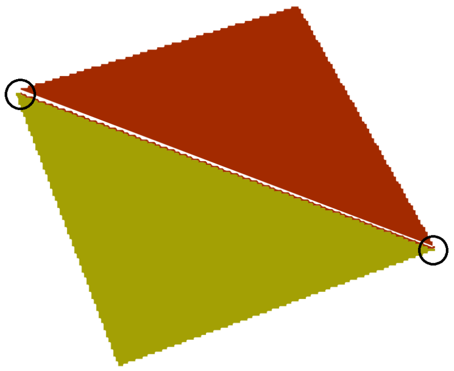


Figure 3.19: An SVG plane containing holes, with the holes highlighted in black circles.

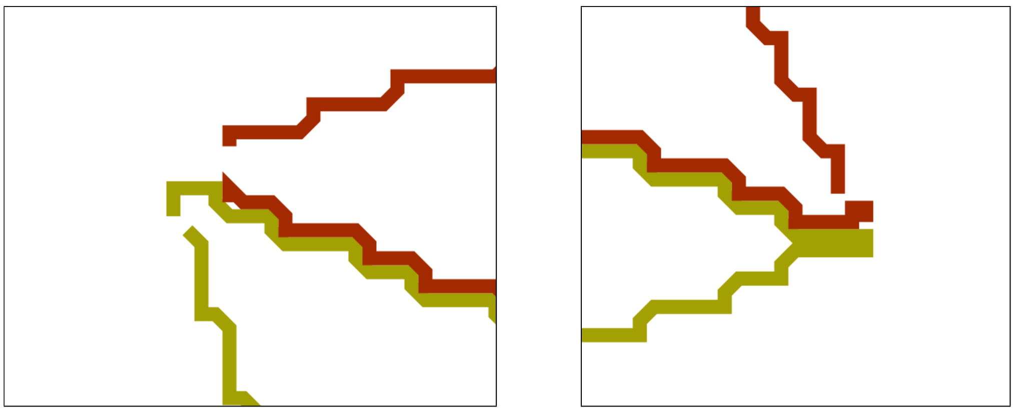


Figure 3.20: The left and right holes as highlighted in Figure 3.19.

neighbors if the distance between them is smaller or equal to the threshold. Threshold means the distance in number of pixels between two points. If they are neighbors, the two pieces are replaced with a new connected piece containing all points of the two old pieces, but with the neighboring points connected together.

The result of merging the pieces of the orange triangle with threshold 3 is shown in Figure 3.21. The merging occurs in the left corner of the plane, highlighted in Figure 3.20 A higher threshold might be necessary in other cases to achieve properly connected silhouettes.

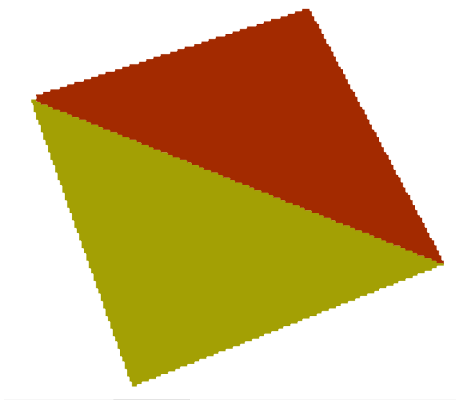


Figure 3.21: The orange and yellow plane after the orange triangle's pieces have been merged into a single connected piece.

It should be clarified that all SVG shapes that this solution creates always contain at least one hole, located between the start and end point. Although this does theoretically leave every shape open, it is of little significance because the start and end points are so close to each other that the color filling is not affected.

3.7 Converting silhouettes into SVG

This is the final stage of the algorithm, where the algorithm has traced around every group of contours and drawn them as groups of silhouettes. Figure 3.22 shows how each depth layer has been organized into groups of silhouettes, where each group corresponds to a mesh in the 3D structure. Each silhouette in a group corresponds to a subset of connected triangles in the mesh where all the triangles share the same color. The shape of each silhouette is defined by its two-dimensional points, and each silhouette has its own color.

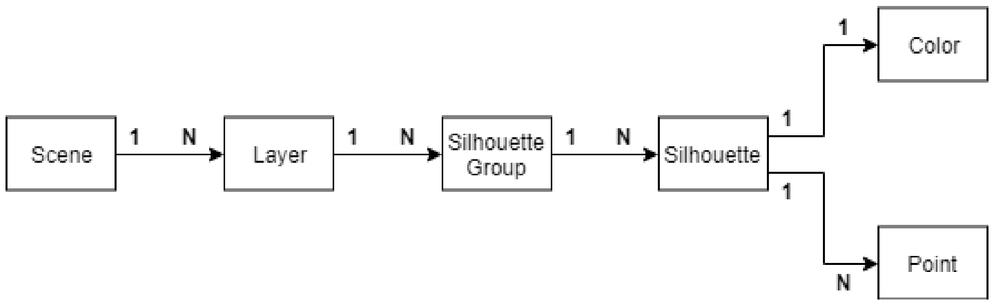


Figure 3.22: The structure of the scene after it has been converted to 2D.

The solution can now traverse the 2D data structure described in 3.22 and convert it into the SVG format. This is done by first creating an empty SVG document, adding a new group of silhouettes with a "<g>" tag, and adding a new "polyline" shape for each silhouette into their correct group. Each silhouette is added to the document by iterating through its points and writing them with their x and y coordinates, starting with the first point and writing them in the order in which they were added by the algorithm.

The listing below shows the SVG file that is the result of rendering the cube in Figure 3.5. The first two lines are the SVG file's metadata, including the width and height of the SVG canvas. It is the same as the render resolution, as given by the user in the Unity editor. <svg> indicates the start of the SVG file. <g> indicates a group, and in this case each group is one of the six sides of the cube, consisting of two triangles made with two polylines in the SVG. The "scale" and "translate" values are included to ensure that the vector graphics are not inverted, because the drawing algorithm (Section 3.6) draws the silhouettes in a reversed manner. After all the points have been added for the polyline with the "points" attribute, the polyline is given the correct line color and fill color according to the triangle's silhouette inside the "style" attribute, with the "stroke" and "fill" attributes, respectively.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
    width="1920" height="1080">
<g>
<polyline transform="scale(1, -1) translate(0, -1080)"
    points = "672,173 672,174 672,175 672,176 (...) "
    style="fill:rgb(55, 163, 0);
        stroke:rgb(55, 163, 0);" />
<polyline transform="scale(1, -1) translate(0, -1080)"
    points = "678,73 679,73 679,72 680,72 (...) "
    style="fill:rgb(0, 163, 25);
        stroke:rgb(0, 163, 25);" />
</g>
<g>
<polyline transform="scale(1, -1) translate(0, -1080)"
    points = "674,201 675,201 675,200 676,200 (...) "
    style="fill:rgb(163, 160, 4);
        stroke:rgb(163, 160, 4);" />
<polyline transform="scale(1, -1) translate(0, -1080)"
    points = "678,199 678,200 678,201 679,201 (...) "
    style="fill:rgb(163, 42, 0);
        stroke:rgb(163, 42, 0);" />
</g>
<g>
<polyline transform="scale(1, -1) translate(0, -1080)"
    points = "792,145 792,146 792,147 792,148 (...) "
    style="fill:rgb(86, 59, 163);
        stroke:rgb(86, 59, 163);" />
<polyline transform="scale(1, -1) translate(0, -1080)"
    points = "798,50 798,51 798,52 798,53 (...) "
    style="fill:rgb(1, 14, 163);
        stroke:rgb(1, 14, 163);" />
</g>

(Three groups for layer 2)

</svg>
```

Listing 3.2: The resulting SVG file.

Most of the polyline points in the listing have been replaced with "(...)". The listing also only shows the groups for layer 1 and not layer 2 to save space. Layer 2 is structured the same way as layer 1, but with different values. Figure 3.23 shows the result of opening this SVG file in the SVG editor software Inkscape [30], which is a free and open source vector graphics editor. The left image shows the cube as it appears initially, with only the frontmost layer (layer 1) visible. In the right image, the top and right sides of the cube have been moved to expose parts of the hindmost layer (layer 2). This movement of the two sides was done in Inkscape after rendering it to SVG.

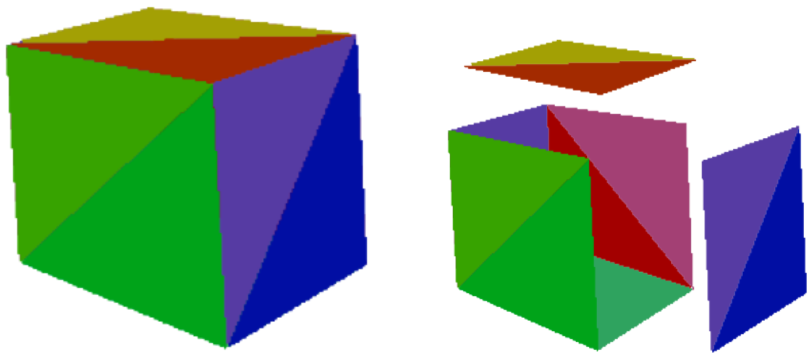


Figure 3.23: Left: The resulting SVG drawing of a cube. Right: The same cube with its front and top sides removed. Notice how the backsides are revealed when moving the two front faces.

Chapter 4

Results and Evaluation

4.1 Visual results

The solution successfully converts meshes of single-colored triangles into SVG files. All the layers in the scene are maintained and are in the correct order of depth. The solution does not convert textured meshes into SVG, so any eventual textures on the meshes are removed prior to rendering. The resulting SVG does not include shading, so any lights in the scene are only for the purpose of helping the user see the meshes clearly in the editor, and are otherwise ignored.

Potential errors in the results can be silhouette holes (3.6.2), incorrect grouping of the silhouettes and incorrect order of fragments when sorting the depth layers (3.3.1). The incorrect grouping happens in only a few cases, and can be due to a small chance of some of the random ID tag color values being identical between two or more meshes. If none of these errors happen, the solution gives satisfying results and all information in the scene is maintained according to the thesis goal. Some experimentation with silhouette merging thresholds might be necessary from the user in order to achieve the smallest possible amount of holes. The user may also have to manually regroup the silhouettes in the cases where the automatic grouping provided by the solution is incorrect.

4.1.1 Comparison with previous work

A major limitation in the solution of this thesis is that it splits meshes into multiple silhouettes more than what is necessary for the all layers to be maintained. While it does manage to group the parts of a mesh located in the same depth layer, meshes that span multiple depth layers are cut in such a way that each part is cut based on occlusions and order of depth. [1]'s method on the other hand, only cuts a mesh when it is necessary. The layering method of this thesis is similar to that of depth peeling. Figure 4.1 shows their comparison of their own method and depth peeling and illustrates these differences between the two approaches to layering. In the figure, "Depth Peeling" causes the visible parts to be regarded as a single layer, whereas the rest of the red circle that is occluded by the blue circle appears in a separate layer. In their solution however, the two circles are regarded as their own independent layers, and are not cut.

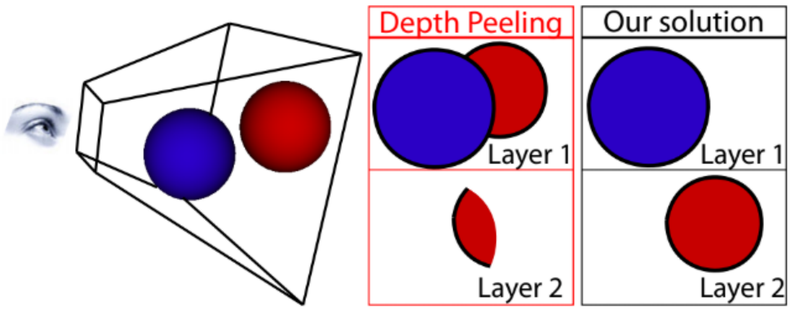


Figure 4.1: [1]'s comparison of depth peeling to their own solution.

There could be found no way of using the strengths of [1]'s approach to only split the meshes when necessary in this thesis. They present a high level description of their method, and the implementation details surrounding their method are unclear. Finding a way to group the silhouettes across depth layers, while also respecting the order of the vector graphics layers, proved to be difficult considering the design choices that were made in this thesis. As for the issues of self-occlusions and mesh chains, there were made no specific methods of solving these in this thesis. Both issues are automatically handled by the fact that the meshes are split up according to the principles of depth peeling.

4.1.2 Comparisons with the original scenes

This subsection shows the result of converting three different scenes of varying complexity into vector graphics, and compares the result with the original 3D scene.

Color grid

The mesh in this scene (Figure 4.2) is a large square built up by multiple smaller squares, each having their own single color. Because the entire mesh is visible from the given viewpoint, the resulting SVG has only one layer. This scene shows that the solution can successfully group a collection of triangles in a single mesh into a single silhouette.

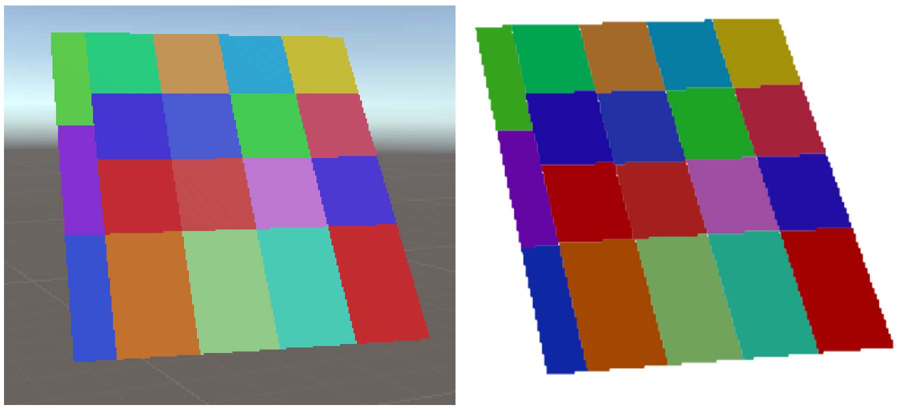


Figure 4.2: Left: A scene containing a colored grid. Right: The result of rendering this scene to SVG.

When experimenting with this scene from several different viewpoints, occasional holes in the silhouettes were encountered. However, because of the simplicity of this scene, there were no other major errors or limitations in its SVG file compared to the original scene, and the mentioned holes occurred only rarely. All the information about the original scene like the grid's shape, colors and the grouping of the triangles, was maintained in the SVG file.

Cube

This scene contains the same cube that was used as an example in the illustrations in Chapter 3. It consists of six meshes, one for each side. Each side has two triangles of different colors. From the viewpoint in Figure 4.3, three of the sides are initially visible, and the remaining three are in the next layer. As shown in Figure 3.23, removing any of the foremost sides reveals the sides in the layer behind them.

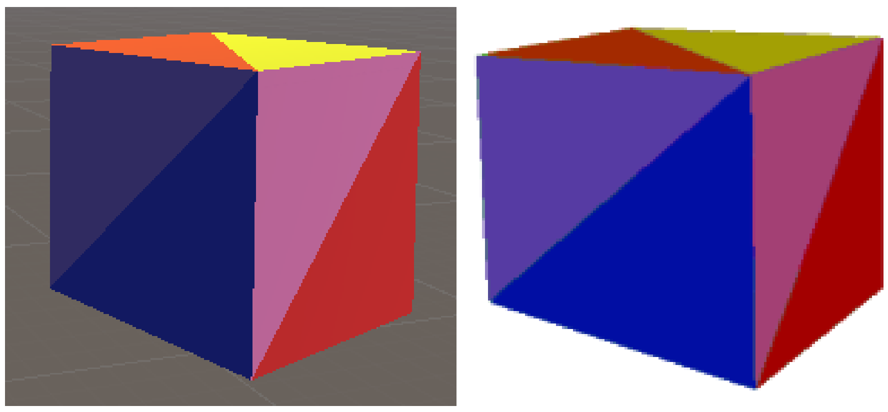


Figure 4.3: Left: A scene containing a cube. Right: The result of rendering this scene to SVG.

As shown in Chapter 3, there were several cases where holes occurred in the cube's silhouettes. All of these cases were solved by optimizing the merging threshold value prior to rendering the cube. The most notable downside of the results from experimenting with this cube, was that the error rate of grouping the triangles was higher than in the other scenes. Mostly this resulted in three and three of the triangles being grouped together instead of the two triangles for each side of the cube. This in turn also lead to leftover triangles not being grouped with any other triangles at all. It can be concluded that all the information about the original scene was maintained in the SVG file in the cases where the triangle grouping was correct and there were no silhouette holes.

Terrain

Like the cube scene, the scene in Figure 4.4 also contains multiple meshes. Each mesh is a geological layer in a terrain model, where the green, brown, black and red meshes represent grass, dirt, rock and magma, respectively. The terrain in this scene aims to replicate some of the same challenges presented in Section 1.2 and Figure 1.1, and is a practical example of scenes that can be valuable to edit in a vector graphics format. The resulting SVG file contains eight depth layers in total.

Due to the curved and uneven shapes of the meshes, they can occlude each other and themselves in many ways and from many different perspectives. Section 4.1.1 mentioned a limitation in the solution where the meshes

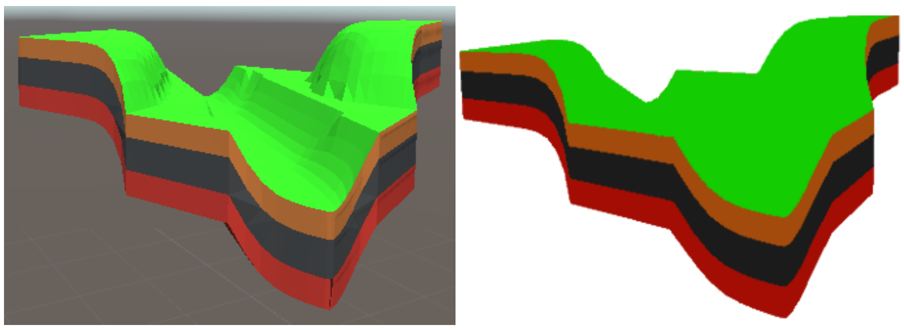


Figure 4.4: Left: A scene containing a terrain. Right: The result of rendering this scene to SVG.

are split even when not necessary, due to the nature of depth peeling. For the terrain scene, this became more and more apparent the deeper one went in the layers in the SVG file. Figure 4.5 shows that the parts of the grass mesh that were initially occluded from the given viewpoint, were separated into multiple silhouettes. The visible part of the grass was one single silhouette, and the occluded parts were divided into several silhouettes. When the foremost silhouette was removed, the remaining silhouettes of the same mesh were exposed.

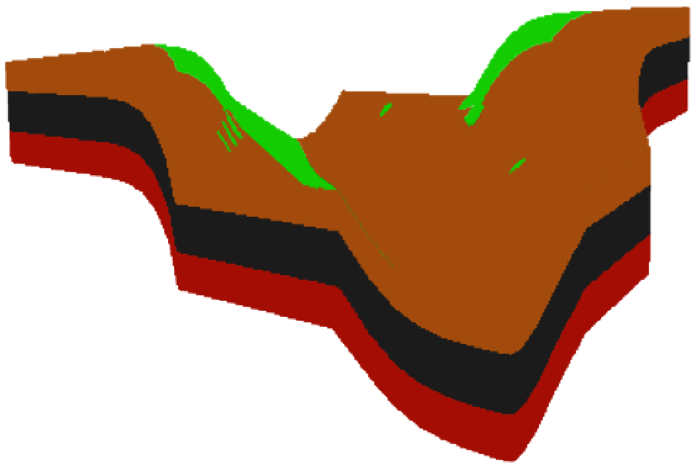


Figure 4.5: The terrain scene's SVG file with most of the grass (green) silhouettes removed.

This negative aspect of depth peeling was even more apparent in the deeper layers, because of the complexity of the scene, and therefore also a high amount of self-occlusions and occlusions between different meshes. In Figure 4.6, almost all the grass silhouettes are removed, as are most of the dirt silhouettes. As the figure shows, some of the green and brown silhouettes still remained in the deeper layers. These silhouettes were very small in size and would likely provide little value to the user. This particular SVG file also contained many silhouette holes, particularly in the rock (black) and magma (red) silhouettes, although the amount of holes varied greatly between different experiments and resulting SVG files.

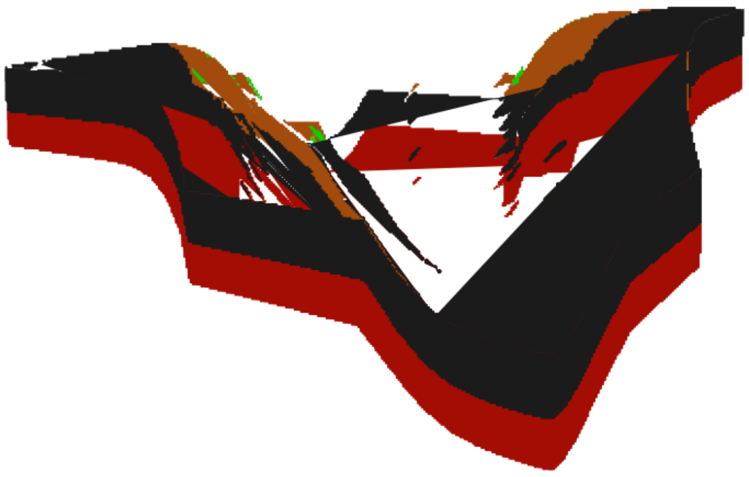


Figure 4.6: The terrain scene's SVG file with the grass (green) silhouettes and most of the dirt (brown) silhouettes removed.

Summary

Based on the results of the three scenes that were tested, one can conclude that the grid and cube scenes resulted in SVG files with few to no errors. In the cases where the grouping of the silhouettes was incorrect, the user could manually split the groups and regroup them correctly by using the tools provided in Inkscape. The result of converting the terrain scene into SVG proved to be less ideal, where the holes were more frequent and difficult to remove by merging the silhouettes. Incorrect grouping also happened here, although in fewer cases than with the cube. Testing the solution with the terrain scene also made the solution's limitations related to depth peeling very apparent, with the meshes being split into a high amount of silhouettes that were larger in numbers the deeper one went through the depth layers, and often very small in size. The simplest way for the user to fix this would be to manually remove the smaller silhouettes, if they proved to be of little use to them.

4.1.3 The torus problem

One situation that causes an issue when converting meshes into SVG is when a single colored mesh contains a hole, like the torus in Figure 4.7.

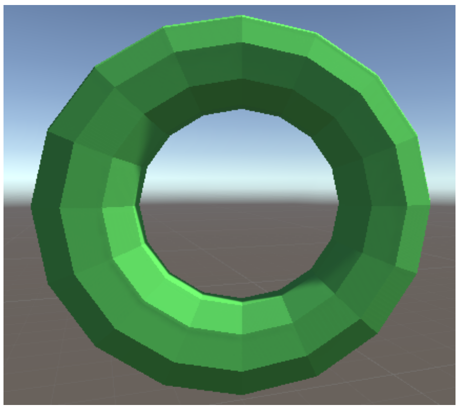


Figure 4.7: A scene containing a single colored torus.

This torus would ideally result in one silhouette that includes its hole through the center. However, converting the torus from 3D to 2D resulted in an outer outline that defines its round shape, and an inner outline defining the shape of its hole. This inner outline was regarded by Inkscape as the outline of a separate circle, and was therefore automatically filled with its color as a separate silhouette. This error is shown in Figure 4.8.

The right side shows the two outlines that were drawn by the drawing algorithm, and the left side shows them both filled with color in Inkscape. While this error can be solved easily by the user by toggling off the hole's fill color, it still presents a challenge that arises when silhouettes are drawn from the outlines of three-dimensional objects with holes. It is worth mentioning that this issue would not appear if one half of the torus had a different color, because the inner circle of the torus would be regarded by Inkscape as outlines of two separate silhouettes.

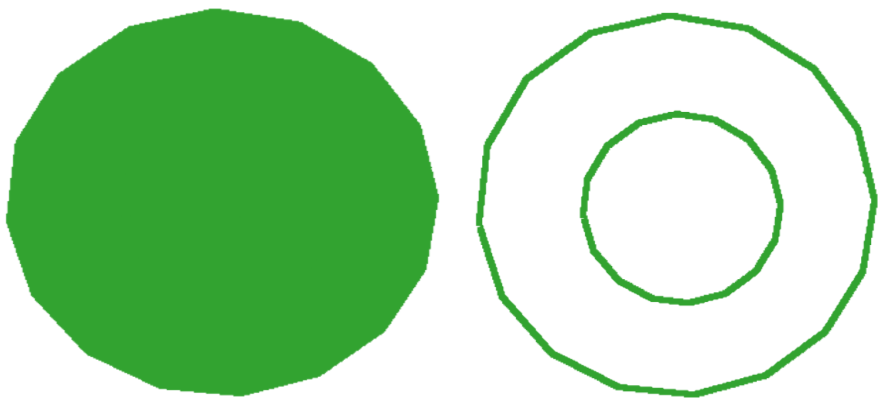


Figure 4.8: Left: The result of rendering a single colored torus to SVG. Notice how its hole is also filled with color. Right: The same SVG but with the fill colors toggled off.

4.2 Numerical performance

The performance of the implemented solution is an important factor. The solution automates the process of converting 3D meshes into vector graphics for the purpose of letting the user avoid doing this work manually. However, this automation would be of little contribution if it meant long waiting times for the user. Being a faster alternative to the manual procedures is therefore an important aspect of the solution. In order to verify that it is indeed faster, the solution's running time was tested on the three scenes shown in Section 4.1.2. The specifications of the testing machine is shown in Table 4.1. As most of the solution runs on the CPU, the solution's speed is most dependent on the CPU's performance. Rendering the Unity scene to LDIs is the only step that runs on the GPU. This will be apparent in the running times, because the running time for generating LDIs will always be much lower than the other steps.

For each of the three scenes, the running times for the five main steps of the solution were measured individually. Those five steps are generating LDI, extracting contours from the depth layers obtained from the LDI, grouping the contours based on ID tag colors, drawing silhouettes from the contours and writing the silhouettes to SVG files. The total running time for the

CPU	Intel Core i5-4670K
GPU	MSI GeForce GTX 970 4GB
RAM	8GB

Table 4.1: Hardware specifications of the machine that the solution was tested on.

scenes was also measured, which includes rendering the meshes to LDIs with their ID tag colors and extracting the contours from the depth layers. All scenes were tested three times, each time with a different resolution. The tested resolutions were 720x480, 1280x720 and 1920x1080.

In addition to the running times, the size of the finished 2D structure for the three scenes was also analyzed. This included counting the number of depth layers. Further, the number of silhouette groups, silhouettes, silhouette pieces and total number of points were counted. The ideal number of silhouette pieces is equal to the number of silhouettes, because this indicates that the silhouettes contain no holes. On the other hand, if the number of silhouette pieces is higher the number of silhouettes, this indicates silhouette holes.

4.2.1 Color grid

The running times and size of the vector graphics for the color grid scene are shown in Table 4.2 and Table 4.3, respectively. The first table shows that none of the resolutions proved to be problematic for this scene in regards to performance. The running time at 720x480 was around 2 seconds, increasing with 102% to around 4 seconds at 1280x720, and increasing further with 124% to a little under 10 seconds at the highest resolution. Although the running times well over doubled when going to a higher resolution, around 10 seconds for the highest resolution can be considered a satisfying result. However, considering the simplicity of the scene and the fact that it only contained one depth layer, this is not a sufficient indication of how well the solution would perform in practice.

The scene contained only one depth layer because all the surfaces were visible from the given viewpoint. It also contained only one silhouette group, proving that all the triangles of different colors were correctly grouped together, as the input scene was a single mesh. The grid consisted of 19 squares, all of which were drawn as 19 silhouettes. The number of silhouette pieces

Resolution	**720x480**	**1280x720**	**1920x1080**
Generating LDI	0.012	0.029	0.06
Extracting contours	0.142	0.431	0.883
Grouping contours	0.528	1.289	2.942
Drawing silhouettes	0.765	1.952	4.364
Writing SVG file	0.03	0.091	0.124
Total time	2.192	4.419	9.898

Table 4.2: Running times for the color grid (seconds).

Resolution	**720x480**	**1280x720**	**1920x1080**
Depth layers	1	1	1
Silhouette groups	1	1	1
Silhouettes	19	19	19
Silhouette pieces	19	21	19
Points	2114	3211	4863

Table 4.3: Color grid's number of entities in the vector graphics.

also reflected this in two of three cases. One of the silhouettes were drawn in three pieces on the middle resolution (as indicated by the 21 pieces instead of 19), although this did not result in any visible holes. Lastly, the increase in points reflects the increasing resolution, as a higher resolution means more pixels to trace and add to the silhouettes.

4.2.2 Cube

The running times and size of the vector graphics for the cube scene are shown in Table 4.4 and Table 4.5, respectively. Although this scene contained two depth layers as opposed to the color grid's single depth layer, the running times were similar to those of that scene. This could be due to the high number of silhouettes in the color grid scene that made up for its single depth layer. The cube scene's vector graphics only contained 12 silhouettes, one for each triangle. Table 4.5 shows that some of these silhouettes were drawn in multiple pieces, being the least apparent at the lowest resolution (15) and slightly more apparent at the middle resolution (17). However, as in the color grid scene's case, this did not result in visible holes in the SVG file. The resulting SVG file had a higher number of points the higher the resolution, and this number was always slightly higher than the color grid scene's SVG file, which was likely due to the cube being closer to the camera than the color grid, and the cube's higher depth layer count.

Resolution	**720x480**	**1280x720**	**1920x1080**
Generating LDI	0.012	0.044	0.114
Extracting contours	0.289	0.737	1.826
Grouping contours	0.327	0.852	1.931
Drawing silhouettes	0.475	1.231	2.756
Writing SVG file	0.077	0.226	0.517
Total time	2.24	4.779	10.795

Table 4.4: Running times for the cube (seconds).

Resolution	**720x480**	**1280x720**	**1920x1080**
Depth layers	2	2	2
Silhouette groups	6	6	6
Silhouettes	12	12	12
Silhouette pieces	15	17	16
Points	3157	4839	7307

Table 4.5: Cube's number of entities in the vector graphics.

4.2.3 Terrain

The running times and size of the vector graphics for the terrain scene are shown in Table 4.6 and Table 4.7, respectively. This scene was the largest of the three scenes that were tested, and was also the most complex one, which is reflected in the tables. The running time started at around 6.8 seconds for the lowest resolution, increasing with around 171% to 18.4 seconds for the middle resolution. This in turn increased with around 128 % to a little over 42 seconds for the highest resolution.

This scene contained eight depth layers in total. Due to its complexity, it is difficult to evaluate the number of entities in the vector graphics for

Resolution	**720x480**	**1280x720**	**1920x1080**
Generating LDI	0.029	0.118	0.251
Extracting contours	1.09	2.927	6.572
Grouping contours	1.442	3.841	8.503
Drawing silhouettes	1.251	3.455	7.378
Writing SVG file	1.592	4.245	11.710
Total time	6.798	18.449	42.193

Table 4.6: Running times for the terrain (seconds).

Resolution	**720x480**	**1280x720**	**1920x1080**
Depth layers	8	8	8
Silhouette groups	26	27	28
Silhouettes	31	31	31
Silhouette pieces	316	457	727
Points	17128	27967	45105

Table 4.7: Terrain's number of entities in the vector graphics.

this scene. The number of silhouette groups varied slightly, which indicates that incorrect grouping occurred at some places in this scene. There was also a very high number of silhouette pieces as opposed to the number of silhouettes. While the silhouette count remained constant at 31 for all three resolutions, the silhouette piece count started at 316 at the lowest resolution, growing to 457 at the middle resolution and 727 at the highest resolution. As Figure 4.6 showed, this did indeed result in holes that severely affected the quality of the vector graphics in the deeper layers. Not surprisingly, the terrain scene's SVG files had the largest points count by far, containing 17 128 points at the lowest resolution and 45 105 points at the highest resolution. The terrain scene had some empty groups that had to be manually removed from the SVG, which could be related to the limitation where silhouettes were grouped incorrectly. These empty groups are not included in the result tables.

Summary

Setting up the results of all three scenes side by side, the varying complexity of each scene is largely reflected in the running times and the size of the vector graphics that were the result of converting the scenes from 3D to 2D. As mentioned beforehand, rendering the scenes to LDIs always took a very short amount of time due to the fact that it is done one the GPU, and always took well under a second. Drawing the silhouettes was the step that took the longest time for the color grid and the cube, but in the terrain's case feeding the 2D structure into an SVG file was the most time consuming step. The total times varied from roughly 2 seconds for the simplest scene at the lowest resolution, to roughly 42 seconds for the most complex scene at the highest resolution. The running times varied greatly depending on the scene and resolution. These running times are likely to be most dependent on CPU performance, implying that a fast CPU should be used to run this solution for the lowest possible running time. Judging from the results of the terrain scene specifically, one can conclude that using this solution in practice on "real world" cases like a geological model should not result in long waiting times for the user. Most importantly, using this automated solution should never take any more time than converting a 3D scene into layered and editable vector graphics through manual procedures.

Future iterations of this solution could reduce the number of points that define each silhouette by finding smarter and more effective methods than tracing along a contour pixel by pixel. The solution in its current state also shows potential for improvement regarding the high number of silhouette pieces as opposed to silhouettes, specifically in the terrain scene's case (Table 4.7.

Chapter 5

Conclusion

This thesis has introduced an automated method for converting a scene of 3D meshes into editable 2D vector graphics. It has been implemented in the form of Unity scripts that can be used in other Unity projects like video games and other graphics software. There could be found no other open source solutions that can be used for this purpose, and no other solutions that are available for public use by anyone. Although it is implemented in Unity, the base principles should be portable to other game engines or graphics APIs. The solution has managed to maintain occluded geometry that otherwise would be lost in the standard rendering process, by rendering the scene to LDIs. The solution groups all triangles in the same mesh in the vector graphics, greatly increasing their editability. Textures and shading has been left out as future work. [1] introduced a method for converting 3D meshes into editable 2D vector graphics by only splitting meshes when necessary. Because they only give a high-level description of their solution and because its code does not appear to be available, this thesis did not manage to create a solution in the same way. The solution of this thesis instead works similarly to depth peeling.

The intended workflow for the user is to first load .obj files and their .mtl files into the scene, and edit the scene's meshes and camera viewpoint in the Unity editor. The triangles must be grouped into meshes in the .obj files beforehand. Adjustments to the number of depth layers, silhouette merging thresholds and file storage path can also be made by changing the attributes of the scene's camera. When the user presses the editor's Play button, the scene is converted into an SVG file and stored on the machine on the given file path. The running time depends on the complexity and size of the scene, and the resolution that is set in the Unity editor. Judging from the scenes that were tested, the solution is fast enough to not make the running time an inconvenience to the user. The results from these scenes varied between approximately 2 - 9 seconds for the grid scene, and 6 - 42 seconds for the terrain scene, depending on the resolution. The terrain scene is most akin to the scenes that would be used in practice. This proves that the solution's running time is always faster than achieving similar vector graphics through manual procedures. A high resolution is recommended in order to get the sharpest looking SVG.

The research question in this thesis asked how a scene of 3D meshes can be converted into a vector graphics format from a given viewpoint. The thesis has answered this question through the solution's implementation. This includes how the scene must first be constructed by the user, how the scene's depth layers are acquired through LDIs and how the mesh contours are extracted from the depth layers. It also includes how the silhouettes of each mesh are grouped by tagging the meshes with ID tag colors in a separate LDI. Lastly, it includes how silhouettes are drawn based on the mesh contours for each depth layer, and stored in the SVG format.

One of the strengths of vector graphics is that zooming in or out on a vector graphics image does not affect its quality. However, the solution in its current state does not take full advantage of this, because the silhouettes are a result of tracing along the contours of a mesh pixel by pixel, which results in a stair case effect and a high amount of points that define the vector graphics. The former is especially apparent if the scene is set to a low resolution. Future work could take more advantage of the nature of vector graphics by identifying straight lines and curves in the mesh contours and avoid drawing them pixel by pixel. Another limitation in the solution is that silhouettes sometimes are drawn piecewise instead of in a single stroke, leading to unexpected behavior when they are filled with color. In most cases the user can fix this by experimenting with threshold values that are used for merging the pieces. The silhouettes are sometimes grouped incorrectly, but this can be solved by the user by manually regrouping them in an SVG editor.

An experience that was made during the work on this thesis was that working with shaders in Unity can be a challenge due to their poor shader documentation. The implementation for rendering the scene into LDIs was already implemented by Giske [24], which helped the project get a significant head start. However, a lot of time was still spent on improving this implementation by sorting the per-pixel linked list, due to the limited documentation that was available.

Chapter 6

Further Work

6.1 Textures and shading

A useful next step would be to take the textures and shading of the original scene and apply it to the SVG result. The reason behind this is that although textures and shading are not scalable as mentioned in 1.3, they contain a lot of information about the scene that still could be useful in many cases. Furthermore, it appears that there are no solutions for applying textures and shading to vector graphics currently in existence, making it an important contribution to this topic in general. One possible solution is to store a normal rendering of the scene to a render texture and somehow cutting out each piece of textures and shading according to the silhouette points of the SVG, and then applying the pieces as fill to their corresponding silhouettes with the <pattern> tag in SVG [31]. Attempts were made to implement an algorithm like this for the foremost depth layer, but a major challenge was to effectively figure out which silhouette each texture and shading piece belonged to, without comparing each individual pixel of the texture and shading piece with all the points in every silhouette. This would be similar to the method for grouping mesh contours, presented in Subsection 3.5.2. While including textures and shading for the frontmost depth layer can be possible, it is harder to know for sure if it is possible for the other depth layers.

6.2 Lines and curves

In its current state, when this solution converts the 3D meshes into vector graphics, it traces around the contours of each mesh, copying the x and y coordinates of every contour pixel into a SVG polyline point (see Section 3.6

and Section 3.7). While this method ensures correctness of the silhouettes with regards to the original meshes, it does not use the vector graphics format to its full advantage when it comes to scalability. Scaling the SVG results in a "staircase effect" that clearly shows the individual points. Improving the solution of this thesis by implicitly recognizing areas of the silhouettes that can be simplified, and replacing a large number of points in the silhouettes with lines and curves, would lead to a significantly more scalable result and greatly reduce the size of the SVG files. This would also improve the SVG's readability for the user. An example case is the cube presented in Chapter 3, which could reduce each triangle from a large number of points (creating a staircase effect) to three points representing each corner of the triangle.

6.3 Improvements to the current implementation

Drawing the silhouettes in a single stroke

As mentioned in Section 3.6.2, the silhouettes that are drawn may include holes that cause unpredictable behavior for their fill colors. This is due to a limitation in the drawing algorithm which causes the drawing to end prematurely. The way that this limitation is currently handled is to keep count of the number of pixels in the contour and restart the drawing of the silhouette at a new location, as long as the number of visited pixels is lower than the total number of contour pixels. This results in a silhouette possibly consisting of multiple disconnected pieces, causing the holes. An improvement on the drawing algorithm that ensures that a silhouette is drawn in a single stroke instead of restarting multiple times, would likely remove the holes entirely.

Sorting close fragments

Section 3.3.1 mentioned that some pixels in a depth layer could display the colors of the fragments in the same pixel positions in the next depth layer instead of the current depth layer, which is likely due to them being so close to each other that their depths cannot be as easily distinguished. Further work on examining and solving this limitation could improve the correctness of the depth layers and therefore also reduce noise in the SVG output.

Reducing the mesh identification error rate

Although the process of identifying which pixels in a depth layer belong to which mesh usually gives the correct result, this is not always the case. When it does not happen correctly, the silhouettes are put into wrong groups, requiring the user to manually ungroup and regroup them correctly in the SVG file. As mentioned in Section 4.1, this could be due to the random ID tag color values sometimes being identical between two or more meshes.

Bibliography

[1] E. Eisemann, P. Sylvain, and F. Durand, *A visibility algorithm for converting 3d meshes into editable 2d vector graphics*, `https://people.csail.mit.edu/sparis/publi/2009/siggraph/Eisemann_09_A_Visibility_Algorithm.pdf`, Accessed 2018-02-16, 2009.

[2] C Tech Development Corporation, *Geological model of a gas plant site in newburgh, new york*, `https://www.ctech.com/3d-geologic-modeling/`, Accessed 2018-10-26, n.d.

[3] Dictionary.com, LLC, *Silhouette definition*, `https://www.dictionary.com/browse/silhouette`, Accessed 2019-04-26, n.d.

[4] Blender Foundation, *Blender homepage*, `https://www.blender.org/`, Accessed 2018-10-26, n.d.

[5] Adobe, *Adobe illustrator homepage*, `https://www.adobe.com/products/illustrator.html`, Accessed 2018-10-26, 2018.

[6] A. Ellis, W. Hunt, and J. Hart, *Svgpu: Real time rendering to vector graphics formats*, `https://pdfs.semanticscholar.org/34a1/f3cce5ba5a1a6736e9b3b75964f71eb08da2.pdf`, Accessed 2018-02-16, 2016.

[7] E. Eisemann, H. Winnemöller, J. Hart, and D. Salesin, *Stylized vector art from 3d models with region support*, `http://citeseerx.ist.psu.edu/viewdoc/download?rep=rep1&type=pdf&doi=10.1.1.212.3060`, Accessed 2018-02-16, 2008.

[8] StyleCampaign, *3d to svg*, `http://stylecampaign.com/blog/2014/03/3d-to-svg/`, Accessed 2018-04-06, 2014.

[9] Tech-Ease, *What is the difference between bitmap and vector images?* `http://etc.usf.edu/techease/win/images/what-is-the-difference-between-bitmap-and-vector-images`, Accessed 2018-02-16, n.d.

[10] P. Christensson, *Vector graphic definition*, `https://techterms.com/definition/vectorgraphic`, Accessed 2018-02-16, 2006.

[11] WhiteboardStudio, *The difference between raster graphics vector graphics*, `http://whiteboardstudio.com/the-difference-between-raster-graphics-vector-graphics/`, Accessed 2018-03-13, 2013.

[12] World Wide Web Consortium, *Scalable vector graphics (svg) 1.1 (second edition) - introduction*, `https://www.w3.org/TR/2011/REC-SVG11-20110816/intro.html`, Accessed 2018-09-11, 2011.

[13] ——, *Scalable vector graphics (svg) 1.1 (second edition) - basic shapes*, `https://www.w3.org/TR/2011/REC-SVG11-20110816/shapes.html`, Accessed 2018-09-17, 2011.

[14] ——, *Scalable vector graphics (svg) 1.1 (second edition) - concepts*, `https://www.w3.org/TR/2011/REC-SVG11-20110816/concepts.html`, Accessed 2018-09-17, 2011.

[15] ——, *Scalable vector graphics (svg) 1.1 (second edition) - rendering model*, `https://www.w3.org/TR/2011/REC-SVG11-20110816/render.html`, Accessed 2018-09-17, 2011.

[16] Scratchapixel 2.0, *Introduction to polygon meshes*, `https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-polygon-mesh/introduction`, Accessed 2018-11-20, n.d.

[17] P. Bourke, *Object files (.obj)*, `http://paulbourke.net/dataformats/obj/`, Accessed 2018-10-12, n.d.

[18] D. Ramey, L. Rose, and L. Tyerman, *Mtl material format (lightwave, obj)*, `http://paulbourke.net/dataformats/mtl/`, Accessed 2018-10-12, 1995.

[19] Tech Terms, *Api definition*, `https://techterms.com/definition/api`, Accessed 2019-02-15, 2016.

[20] M. Segal and K. Akeley, *The opengl graphics system: A specification*, `https://www.khronos.org/registry/OpenGL/specs/gl/glspec40.core.pdf`, Accessed 2018-12-10, 2010.

[21] Unity Technologies, *Unity editor*, `https://unity3d.com/unity/editor`, Accessed 2018-11-21, 2018.

[22] Computer Hope, *Z-buffering*, `https://www.computerhope.com/jargon/z/zbuffering.htm`, Accessed 2018-06-12, 2017.

[23] J. Shade, S. Gortler, L. He, and R. Szeliski, *Layered depth images*, `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/Shade-SG98.pdf`, Accessed 2018-06-12, 1998.

[24] M. E. Giske, *Interactive hybrid ray tracing in unity*, Western Norway University of Applied Sciences, University of Bergen, 2018.

[25] J. McKee, *Real-time concurrent linked list construction on the gpu*, https://developer.amd.com/wordpress/media/2013/06/2041_final.pdf, Accessed 2018-10-19, 2011.

[26] Nvidia, *Transparency (or translucency) rendering*, https://developer.nvidia.com/content/transparency-or-translucency-rendering, Accessed 2019-02-20, 2014.

[27] Dictionary.com, *Outline definition*, https://www.dictionary.com/browse/outline?s=t, Accessed 2019-02-15, n.d.

[28] Unity Technologies, *Gameobject*, https://docs.unity3d.com/ScriptReference/GameObject.html, Accessed 2019-05-03, 2019.

[29] M. E. Giske, *Hybridraytracing-unity*, https://github.com/Nepetre/HybridRayTracing-Unity, Accessed 2018-10-19, 2018.

[30] Inkscape, *Overview*, https://inkscape.org/about/overview/, Accessed 2019-03-05, 2019.

[31] Mozilla, *<pattern>*, https://developer.mozilla.org/en-US/docs/Web/SVG/Element/pattern, Accessed 2019-04-26, 2019.

List of Figures

Listings

List of Tables