UiO : Department of Informatics
University of Oslo

Compression of 3D media for internet transmission

Atle Nordland

Master's Thesis Autumn 2016

Compression of 3D media for internet transmission

Atle Nordland

Abstract

Large real-life scenes can be captured in 3D efficiently by current capturing devices. The captured 3D data can be transmitted over the internet to a remote viewer, who can experience the captured scene with a display capable of displaying 3D. The 3D data output from a capturing device needs to be represented digitally for storage or internet transmission. Data compression is a vital part of successful internet transmission of data, as the available bandwidth is a limiting factor. In this thesis, we investigate the different ways to represent the 3D data, and how suited the representations are for internet transmission. We select the triangle mesh as the best suited representation, and investigate how to best compress it. We choose two mesh compression algorithms to implement, based on their effective compression rate. One is based on the valences of the vertices, while the other is based on the octree data structure. We evaluate the implementations based on factors important for internet transmission of captured 3D data, like the visual quality and required processing power.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor Carsten Griwodz, who has provided excellent feedback, advice, and support. I would also like to thank my fellow master students on the tenth floor of OJD, who have been invaluable during the last year.

Oslo, August 9, 2016
Atle Nordland

1 Introduction

1.1 Background

Three-dimensional (3D) media has the capability to give a more immersive and accurate experience of media consumed by the viewer, compared to traditional two-dimensional (2D) media. In addition to the 2D data, it also provides depth information. A real-world location captured in 3D gives the viewer the ability to easily understand the scale and the relative distances of the captured objects, much easier than when viewing a regular 2D image. In the recent years there has been an increase in popularity of head-mounted displays (HMDs), like the Oculus Rift or HTC Vive. With a HMD we can move around in a 3D environment, either created manually like the graphics in a computer game, or captured from the real world.

It is easier than ever to capture the geometry of large 3D scenes, like entire rooms. An example of a device capable of capturing this information is the Structure Sensor, which is a mobile 3D sensor created by Occipital. The sensor is attached to a tablet, and by scanning a room with the tablet we can create a 3D map of the room in real-time. Even more portable solutions have been researched as well, by utilizing the increasily powerful smartphone as a capturing device [Tanskanen et al., 2013] [Pribanic et al., 2016].

When we use a capturing device to capture the 3D information in a scene, we need a way to represent the data digitally. Captured 2D data is represented as an image, which is a 2D array of pixels. There is no similar dominant 3D representation, as there are multiple ways to store the information output by capturing devices capable of capturing the 3D information of a scene. Some representations are based on 2D images, adding extra information to describe the depth information in

the captured scene. Others are based on points in a coordinate system, where each point describe a small part of the captured scene.

If a person wants to experience being inside a 3D environment captured in another location, the environment must be transmitted to the viewer. When transmitting the data its size is important, as the transmission time is restricted by the bandwidth and latency of the connection between the sender and receiver. By compressing the data before transmission, and decompressing the data after it has been received by the receiver, the transmission time will be reduced.

1.2 Problem Definition / Statement

In this thesis we want to find the best way to represent and transmit captured 3D data, by finding the most suitable representation and associated compression algorithm. We investiage the best way to represent captured 3D data, based on the representations' visual qualities and suitability for internet transmission. There will be an introduction to the different ways to represent 3D media. We will discuss the representations' abilities to represent 3D scenes, and their suitability for internet transmission. We will look at how a real life scene can be reconstructed in 3D for each of the representations, and how the reconstructed data can be visualized. An exploration of how the representations can be compressed will be done as well, for effective transmission over the internet. A comparison will be made of the different representations, and a representation to investigate further will be chosen.

Next, we will find the best way to compress the representation, based on the compression algorithms' suitability for internet transmission. Two of the state of the art compression algorithms will be selected. We will create a design for each algorithm based on the source research papers, and create implementations based on our design. We will compare the algorithms on factors important for the transmission of 3D media, such as visual quality and required processing power. Compression rate is only one of several important factors when transmitting media, but is the one usually focused on when a new compression algorithm is compared to the existing ones. This is why we do not focus on the compression rate in this thesis, but rather the factors mostly ignored by the mesh compression literature.

1.3 Limitations

Because of the scale of the implementations, we found that it was unfeasible to implement a step done to further compress the output of the compression algorithms, a compression technique called arithmetic coding. The additional step is done by both the chosen compression algorithms, which means that we can evaluate the computational effectiveness of the algorithms correctly. As we explained in 1.2, we do not need to calculate the compression rate of the coders, as it has already been calculated in other research papers.

1.4 Main Contributions

We first provided an overview of the different ways to represent 3D media, and we have concluded that a triangular mesh is the best representation to use when transmitting 3D media over the internet. Compared to the other representations, we found that the triangular mesh can best contain the surface of a captured 3D scene.

Next, we compare two different compression algorithms, both with a very good compression rate. We chose the valence based progressive algorithm from [Alliez and Desbrun, 2001] and the octree based progressive algorithm from [Peng and Kuo, 2005], as they provide effective compression, while using very different ways to create intermediate levels of detail (LODs).

The source research papers provides no source code, so we created designs of the algorithms based on the source research papers, and implemented the compression algorithms based on our design. We compared our implementations on factors important for media transmission, like visual quality, required processing power and data granularity. We found that the visual quality of the intermediate LODs is very different between the two compression algorithms, as the valence based algorithm provides intermediate LODs with an appearance much closer to the original mesh. However, the octree based algorithm provides a higher compression rate, and can more easily compress a scene consisting of several separate meshes.

1.5 Outline

In chapter 2, we present the features of four different ways to represent 3D media. We examine how the representations can be compressed, and give a short introduction to compression theory in chapter 3. In chapter 4, we evaluate the differences between them, and how suited they are to transmit 3D media over the internet. We choose the triangular mesh as our representation, and select two compression algorithms to compare. In chapter 5 and 6, we present our designs based on the algorithms from [Alliez and Desbrun, 2001] and [Peng and Kuo, 2005]. In chapter 7, we choose a mesh library needed to contain the mesh information, and we present the implementations of our designs in chapter 8 and 9. In the end, in chapter 10, we evaluate the implementations, and compare them according to the factors important for internet transmission of meshes.

Part I

3D representations

2

3D media

In this chapter we explain how 3D information can be captured from real life scenes, and we explore the different ways to represent this information.

2.1 Acquiring 3D data

A scene can be captured in 3D by cameras and scanners.

2.1.1 Cameras

A digital camera works by capturing incoming light with a 2D sensor array [Gonzalez and Woods, 2008]. The resolution of the image is determined by the number of sensors in the array, as each sensor captures the information for one pixel. The pixel value is determined from the intensity and color of the light sensed by the sensor.

Dual cameras

We can simulate the binocular vision of humans with two cameras. By placing the cameras side by side, and triggering them simultaneously, we get two images of the scene from slightly different angles. These images do not directly specify the depth information in the captured scene, but lets us perceive the scene in 3D when displaying one image to each eye.

Structured light cameras

A structured light camera consists of a projection system, and a camera system. By projecting a light pattern onto a surface and capturing how the surface distorts it, a depth map can be created.

By using imperceptible light, like infrared light, the depth data can be captured simultaneously with a regular color image of the scene. Since the data is captured simultaneously, we can combine the data into a single image with both depth and colors information. It will not be noticeable when these cameras are capturing, unlike the cameras using patterns made from visible light, as the infrared pattern is invisible to the human eye. An example of such a system is the Microsoft Kinect for the Xbox game console, which can capture 3D video in real-time [Han et al., 2013].

A structured light camera with an infrared projector gets the depth information from a scene by using the infrared projector, and an infrared camera. An infrared pattern is projected onto the scene, invisible to the camera capturing the color data, and is captured by the infrared camera. A limitation of capturing depth by projecting an infrared pattern is that it only works well indoors. The pattern will blend in with the infrared part of the sunlight, and it may not be possible to recognize the pattern in the image captured by the infrared camera.

2.1.2 Scanners

3D scanners work by sampling a scene or an object at defined intervals [Otepka et al., 2013]. One sample is an accurate measurement of the location of the sampled point. The information between two samples will not be captured, requiring the surface to be sampled at a high rate if we want to capture all the small details.

Light Detection And Ranging (LiDAR) is a scanning technology using lasers [Otepka et al., 2013]. The distance to a point is measured by emitting a laser signal, and measuring the time elapsed until the light reflected by the object struck by the laser signal is detected. Lasers have a long range, and can capture the depth of large areas both inside and outside.

Color can be added by capturing a 2D RGB image of the scanned area. The color of each point is the color of the pixel corresponding to the area measured by the laser, requiring the laser and camera to be properly calibrated.

This is similar to the depth information from the structured light cameras, and depth sensors of such a camera could be used to create a point cloud. However, the low resolution of the depth camera and the inability to capture depth information in sunlight makes it unable to be used in many of the applications of the LiDAR technology.

2.2 Representations

We must have a way to represent the captured data. The capturing devices output different types of data, resulting in different ways to store the 3D information. In this section we will describe the most popular ways to represent captured 3D data. Image based representations are manly created directly from cameras, and point clouds are created mainly from scanners. Meshes are not created directly from the data output by capturing devices, but are created from the other representations.

2.2.1 Image based representations

The image based representations are based on regular 2D images, adding additional information to contain the depth information as well.

Stereoscopic images

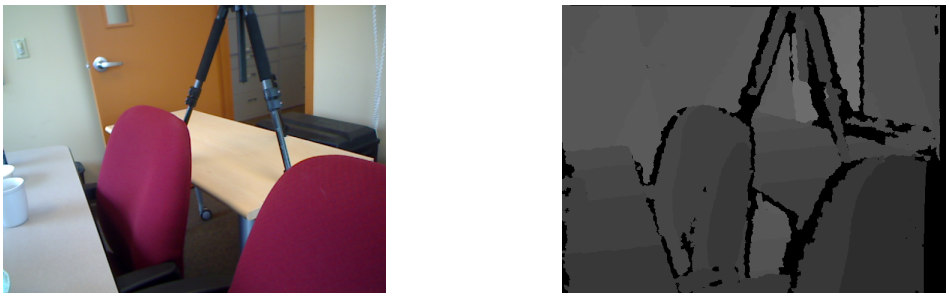
Figure 2.1: A stereoscopic image pair [Digital Collections,].

The image pair produced by the dual cameras is called a stereoscopic image. A stereoscopic image does not explicitly express the depth in the captured scene, but gives us an illusion of depth when viewed. Stereoscopy exploits a property of the human visual system called stereopsis to create an illusion of 3D in images and video. Stereopsis refers to the ability to perceive depth information by processing the visual information captured by the eyes. The space between each eye results in each eye receiving an image slightly different from the image received by the other eye. The part of the brain that processes visual information creates a 3D image from the disparities between the two images, making the viewer capable of viewing the differences in depth between the objects in the image. A stereoscopic image is an image pair showing a scene from slightly different angles, tricking the

brain into combining the images into a single 3D image, when each eye focuses on its respective part of the stereoscopic image.

Figure 2.1 is an old example of a stereoscopic image. The content of the images are mostly the same, but note the differences on the left and right sides of the images. The left image shows more of the wooded hut at the right hand side, but shows less of the area to the left of the tree.

RGB-D



(a) RGB image (b) Depth image

Figure 2.2: An RGB image with its assosiated depth values [Lai et al., 2011].

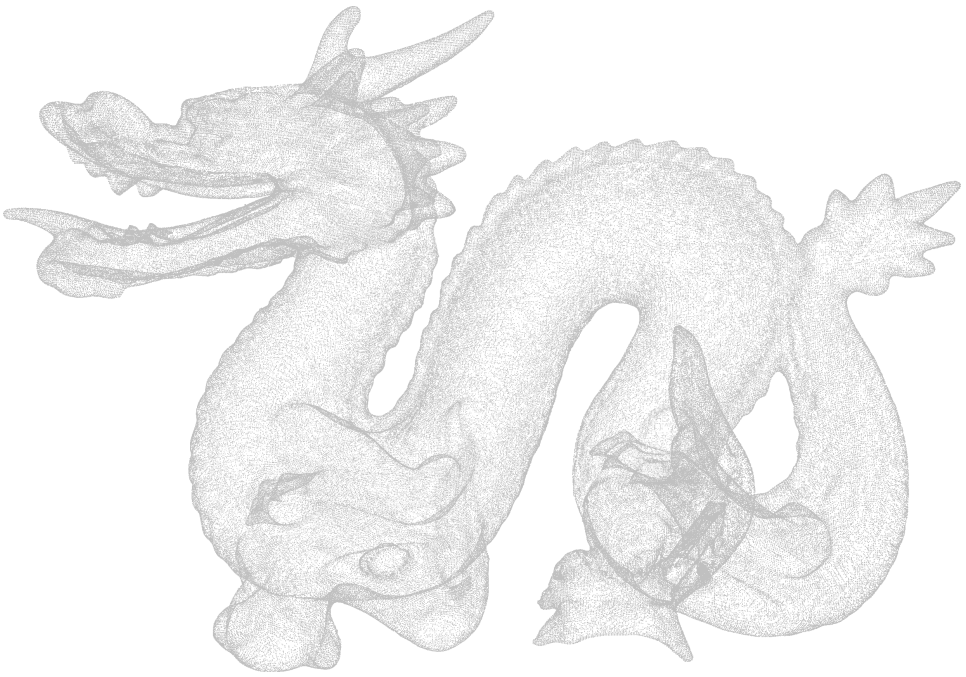
We can incorporate the depth information into an image by associating each pixel with a depth value, called an RGB-D image. Compared to stereoscopic images an RGD-D image contains real depth information. The structured light cameras has the ability to take a picture of a scene, while at the same time capturing the scene's depth information, providing the data needed to create an RGB-D image. It is also possible to obtain the depth data from a scanner, but an RGB-D image can only contain depth data captured from a single angle, while a scanner can capture depth data from multiple angles.

Changing viewpoint in front of an RGB-D image can give the viewer additional information about the 3D scene, unlike a stereoscopic image. Viewing it from an angle makes it easier to see the relative difference in heights, and changing viewpoints can make it easier to see the smaller parts of the image.

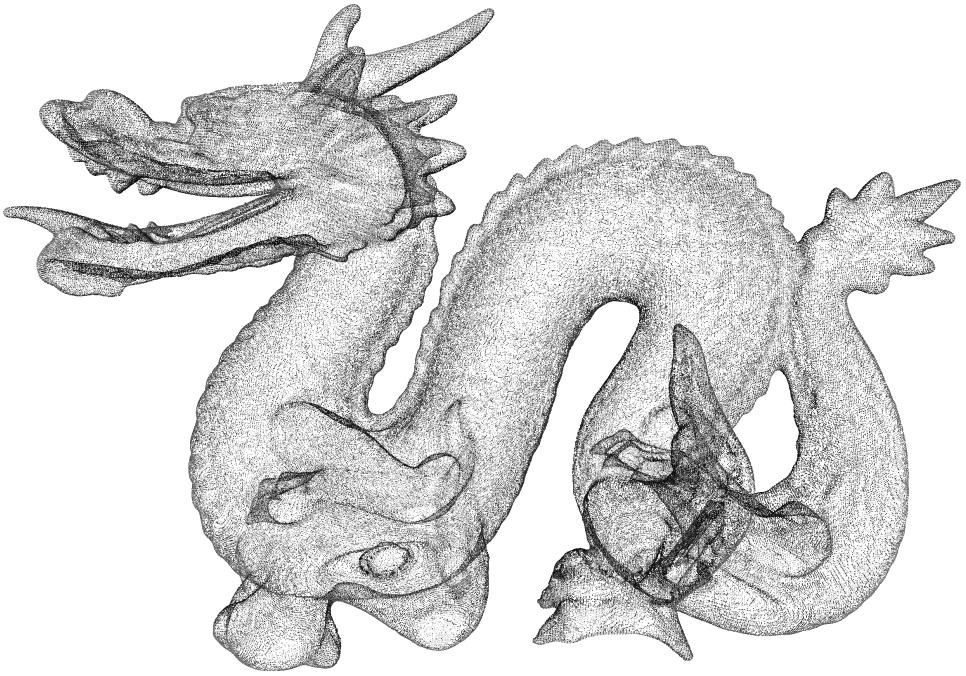
Figure 2.2 shows a frame from an RGB-D video sequence split into two parts, the RGB image and its associated depth image. The dark parts of the image are closer than the light parts.

2.2.2 Point cloud

A point cloud is a set of points in a coordinate system, usually with three dimensions, where each point contains information about the captured scene. A point cloud can contain the sampled data from a scanner. A 3D point is defined by its x, y and z coordinates. A point can contain other information as well, like color



(a) Point cloud



(b) Point cloud with shading

Figure 2.3: A point cloud representation of a dragon model.

or a normal vector. The normal vector is the direction the surface at the sampled point was facing. The points in the set are unorganized, and there are no sharply defined boundaries, as conveyed by the descriptive name point cloud. Together, the set of points can describe a scene, where each point represents a small part of the

sampled surface.

A point cloud can contain an entire scene or object captured by a scanner, unlike the image based representations.

A point cloud can be created from an RGB-D image, having each pixel in the image represent a point in the point cloud. The location of the pixel and the pixel's depth value specifies the coordinates of the point in the cloud.

2.2.3 Polygon mesh

(a) A vertex

(b) An edge connected to vertices

(c) A face connected to vertices and edges

Figure 2.4: The mesh elements.

A polygon mesh consists of three basic elements: vertices, edges and faces. A vertex is a point in a three-dimensional space, and a straight line connecting two vertices is an edge. A face is a closed set of three or more edges. Together, these three elements create a mesh, which can represent any 2D or 3D surface. The simplest mesh possible is a single triangular face, with its three vertices and edges.

Representing an object or an entire scene as a polygon mesh is an approximation process, where the correspondence between the surface and the mesh is dependent on the number of vertices and faces in the mesh. A coarse mesh with few polygons will represent a smooth surface as jagged, and a highly detailed mesh with many polygons can appear as smooth as the surface it is modeled after and make the viewer unable to single out any specific faces.

A mesh can be either manifold or non-manifold. A mesh is manifold if each edge is incident to either one or two faces, and the faces incident to a vertex form a closed or an open fan. In general, a non-manifold mesh represents a surface that cannot exist in the real world. Areas with no thickness, or elements disconnected to the rest of the mesh are possible reasons causing a mesh to be non-manifold.

If a mesh is manifold, it has an additional property as well. It can be either orientable or non-orientable. A face has a front side and a back side, and the orientation of a face is the direction it is facing. The orientation of a face is defined by the order of its incident vertices, which can be either clockwise or counterclockwise. A mesh is orientable if any two adjacent faces have the same orientation. Meshes properly modeled from most real world surfaces will be

orientable. A mesh will not be orientable if it is modeled after an object with a surface with only one side, like the Möbius strip or the Klein bottle.

The information describing the mesh elements is split into geometry and connectivity information. Mesh geometry is each vertex's position in a 3D space. Mesh connectivity, also called topology, describes how the elements fit together. The way the vertices are connected must be specified to get a surface. A mesh without connectivity information is the same as a point cloud, where points are the only elements.

We do not acquire a mesh directly from a scanner or camera, unlike the other representations we have presented. Instead, it is created from other representations.

We can create a mesh from a point cloud by using the points as the mesh's vertices, and creating edges and faces to connect them. Since we can easily create a point cloud from an RGB-D image, we can use RGB-D images to create meshes as well. Reconstructing a polygonal surface from a point cloud generally requires four steps [Remondino, 2003].

1. Pre-processing.
2. Determining global topology.
3. Triangulation.
4. Post-processing.

In the pre-processing step unnecessary point are removed, resulting in fewer polygons to create later. A densely sampled point cloud may have overlapping points, and simple parts of the surface, able to be represented by a single polygon, may be represented by many points in the clouds. These sets of points can be reduced, making the triangulation step easier without compromising the quality of the retriangulated surface. Some erroneous points in the cloud may also result from noise emerged in the scanning process, and these should be removed to get a close approximation to the real surface.

The global topology of the object's surface needs to be determined. The relationship between different parts of the surface needs to be understood, for the preservation of features like edges.

The connectivity of the mesh is created by a process called triangulation. By connecting the point with edges and creating triangular faces, the point cloud is converted to a triangular mesh. Correctly connecting the points is a challenge. Only points adjacent in the surface represented by the point cloud should be connected, or else we might end up with a mesh unrecognizable from the original surface.

In the post-processing steps the polygonal mesh is refined. The mesh surface is improved by removing errors like spikes, by smoothing or removing polygons that can be removed without changing the shape of the surface.

2.3 Visualizing 3D data

2.3.1 Viewing stereoscopic images

Displays capable of showing stereoscopic images are often referred to as 3D displays, even though displays using two images to create an illusion of 3D are not the same as true 3D displays. The scene can only be viewed in 3D from one specific viewpoint. Changing viewpoint in front of the screen will not give us more information about the object being displayed. Commercially available 3D screens are able to show stereoscopic images either with the help of special glasses or without any extra accessories [Holliman et al., 2011].

3D with glasses

Screens requiring glasses from the viewer can display stereoscopic images to an infinite number of viewers at the same time, as long as they wear glasses. Polarized glasses, used in 3D cinemas, filters the light received from the screen and allows different images to be passed through each lens.

Glassless 3D

Screens displaying stereoscopic content without requiring glasses are limited by the number of viewing angles where the 3D effect is properly perceived by the viewer. The screen must be able to simultaneously show different images directly to each eye, without an eye seeing more than one image at the same time. The Nintendo 3DS gaming system solved this problem by displaying a stereoscopic image with the help of a parallax barrier [Emery, 2010]. This barrier is located on top of the regular screen, and works by making each eye capable of only viewing every other column of pixels. The viewer must be located within a small viewing angle for the barrier to block vision of the correct pixel columns, and within a certain distance as well. Figure 2.5 shows how the parallax barrier shows a different set of pixels to each eye. The left eye can see every other pixel, while the barrier blocks the visibility of the pixels designated to the right eye. This is clearly very dependent on the viewing angle. If the viewer is positioned wrongly in front of the screen the barriers will block the visibility of the wrong pixels, and the eyes will receive images created from a combination of pixels from both parts of the stereoscopic image.

Head-mounted dislays

Head-mounted displays (HMDs) can simulate a true 3D display, by using motion tracking and separate image streams for each eye. A HMD is a display strapped to the head of a person, with the screen filling the person's field of view. The motion

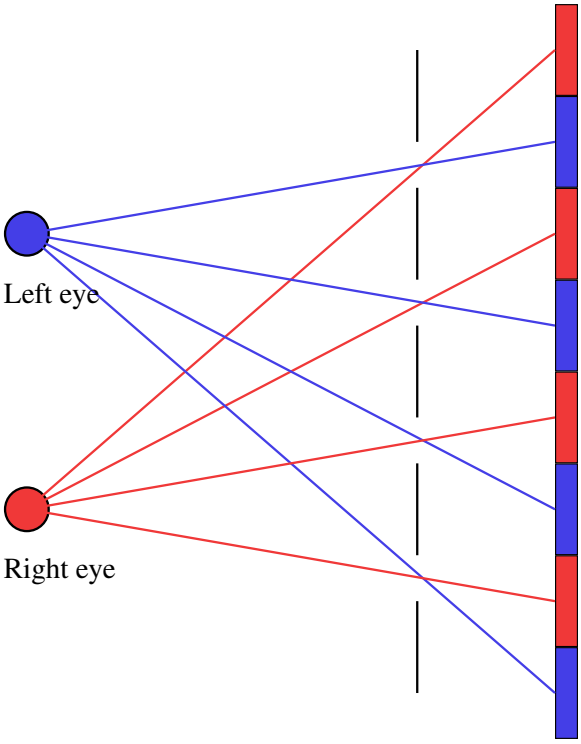


Figure 2.5: Displaying an image to each eye by using a parallax barrier.

sensors can tell the computer the direction the person is viewing, and the location it is being viewed from, enabling the computer to render the correct view. This allows the viewer to turn around and get a full 360 degree view of the scene, and take a step to the side and view it from a different angle as well.

Locally rendered scenes, like in video games, are a good fit for this technology. A new viewpoint can be quickly generated and sent to the HMD through a short cable. Low latency is required to accommodate the quick movements of the head, which makes streaming directly to the HMD from a remote location hard. The time required to send the new position of the headset to the server, generate a new viewpoint and update the stream is too much if the viewer wants a smooth viewing experience while moving around the scene. If the latency is too high, the viewer could already have moved to a different location when the he receives the stream updated for the previous position.

2.3.2 Viewing stereoscopic images

An RGB-D image cannot be displayed directly, as it is not possible to directly show depth data on a 2D screen. This would require the pixels on the screen to move to the position denoted by the depth data. However, an RGB-D image can be seen as a 2D image by projecting the RGB-D image as seen from a specific viewpoint onto a flat 2D surface on a screen.

The ability to view a RGB-D image from different viewpoints, at least within a small range, makes it possible to create a stereoscopic image pair from a single RGB-D image. By using the depth information, the stereoscopic image pair can be generated locally at the viewer side in real-time, instead of capturing and transmitting two separate streams [Kauff et al., 2007]. Generating the viewpoint locally has several benefits, without requiring extra bandwidth. The stereoscopic images can be adapted to different viewing conditions, like the angle the screen is being viewed from, and the distance from the viewer to the screen. It also enables free viewpoint video, where virtual viewpoints can be selected by the user, by a controller or by head tracking of the viewer.

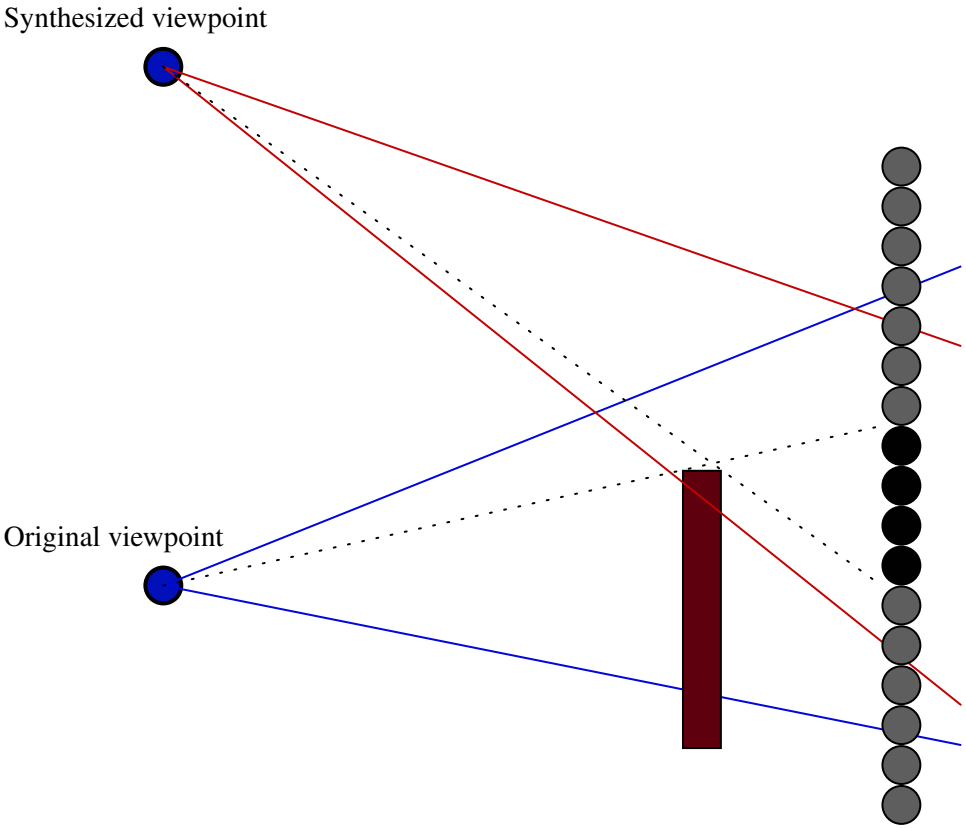


Figure 2.6: The occlusion problem.

The range of possible viewpoint is limited, as an RGB-D image contains only information about the scene from a single direction. Problems with occlusions may arise in the generated viewpoints, also originating from this limitation. Occluded areas in the original image may be visible in the generated viewpoints, and will appear as holes, since the original image contains no information about what is supposed to be in this newly visible area.

Figure 2.6 shows an occlusion resulting in a hole in the synthesized view. The red rectangle occludes some of the gray circles in the background in the original viewpoint. The four black circles are contained in the area which is both visible from the synthesized viewpoint, and should have been visible from the original

viewpoint if not for the red rectangle. Missing the information about this area, it must try its best to fill this area without knowing what is there.

By using multiple RGB-D image streams capturing the scene from different angles, the number of possible viewpoints increases. Through interpolation any possible virtual viewpoint located on the path between two streams can be generated. If the scene is captured from enough directions, it is possible to generate a viewpoint anywhere around the scene, and view the captured scene as a full 3D model. This would, however, require a large number of properly calibrated cameras to provide enough capturing angles.

2.3.3 Viewing point clouds

We cannot view the point cloud in 3D directly on a screen, similar to an RGB-D image. Like an RGB-D image, we can project the point cloud onto a 2D image, and combine two different viewpoints to get a stereoscopic image. The point cloud itself can be visualized in several ways.

Directly viewing a point cloud

It can be easy to get a pleasant viewing experience when directly viewing a cloud if is very densely sampled, but it requires the cloud to have no visible space separating the points. If the points in the cloud are separated enough for us to identify single points, we will not perceive the cloud as a continuous surface. If the point cloud represents the entire 3D surface of an object, we will be able to see the points on the other side of the object through the spaces between each point.

Figure 2.3 shows two images of a point cloud, one with shading and one without. In Figure 2.3(b) each point has been assigned an intensity value, or shade, based on its normal vector and the direction of the lighting. Viewing the nuances of the sampled surface is easier in the shaded point cloud, but it is still hard to get a full understanding of the surface the model represents. As we can see, the hind leg on the opposite side of the model is visible from this side, making it hard to differentiate between the two hind legs.

Conversion to a mesh

As seen in section 2.2.3, we can convert the point cloud into a mesh, where the sampled surface is restored. A mesh can accurately describe a surface, and is a popular way to visualize a point cloud.

It requires a lot of processing power to generate the connectivity between the points in a cloud, especially if it is sampled at a high rate and has a large number of points. If it is not important to visualize the point cloud as a connected surface, a visualization technique like the QSplat system can be used.

QSplat

The QSplat system presents a way to render large models in real-time, without using a polygon mesh [Rusinkiewicz and Levoy, 2000]. It can interactively display models with hundred of millions of points using the computer hardware available in the year 2000. By storing the information hierarchically in a tree structure, the model can be rendered progressively, ensuring smooth frame rate at the expense of visual quality. The amount of data contained in a detailed model can make it difficult to render all the visible points within the time limit of a single frame. Also, attempting to render each visible point in a model is not necessary if the model projects several points onto the space of a single pixel on the screen. It was developed as a visualization algorithm for polygon meshes, but it does not display connectivity data, and visualizes a mesh as if it was a point cloud. The topmost node in the tree contains every point in the model, and it is recursively split in two until every point got its own leaf node. The properties of the intermediate nodes, like colors and normal vectors, are set to be the average of each point contained within the node.

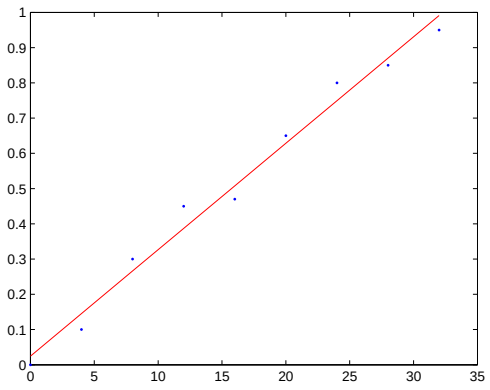


Figure 2.7: A best fitting line. The red line is the straight line best representing the set of blue points.

A tree node is visualized as a splat, shaped like a circular or square flat disk. Its orientation and size is dependent on the normal and size of the node it represents. If it is a leaf node, the normal of the splat is the same as the normal of the contained point. A point in a point cloud may not explicitly contain the normal information, but it can be obtained from the neighborhood around a point. The best-fitting plane given a set of points is the plane that best represents the points in the set. In other words, it is the plane where the points are as close as possible to it. In figure 2.7, we can see a 2D example of a fitting plane, where the line best representing a set of points is shown. By defining the neighborhood as the closest points in a circle around the point we are examining, the best fitting plane gives us the normal vector of the surface sampled at the given point. When all the tree nodes in a chosen depth are visualized, the model can be seen as a series of intersecting splats.

2.3.4 Viewing meshes

Dependent on the application the mesh is used for, there are three major ways to display a mesh [Remondino, 2003].

- Wireframe mode
- Shaded mode
- Textured mode

The easiest way to visualize a mesh is the wireframe mode, where only the edges and vertices are shown. This makes the mesh transparent. It is not suitable in media applications, as it has the same visualization problems as a point clouds, but is mainly used in computer-aided design.

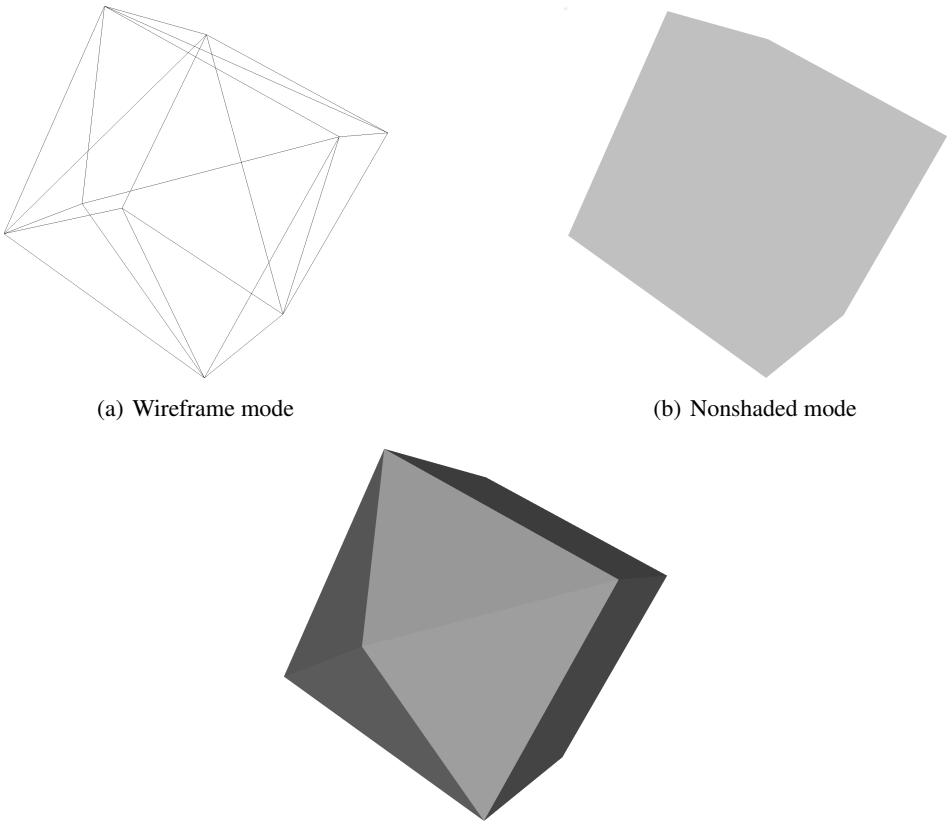
Shaded mode shows faces as well as vertices and edges. In the shaded mode, each face is given a degree of shade, as if a light source shines on the mesh. A face's shade is dependent on its normal vector, which is the direction the face is facing. Without shading it is difficult to see fine details in the mesh, like the difference between two faces on nearly the same plane. With shading each face will get a different degree of shade, as the angle between the normal vector and light angle will be different for the two faces. The wireframe mode and shaded mode is illustrated in figure 2.8.

Textured mode maps a texture, like an image, onto a mesh. Each face will contain a small part of the texture.

2.4 Summary

We can capture 3D information by using multiple cameras or a scanner. The output from the camera systems are mainly used to create stereoscopic and RGB-D image, while the output from scanner are used to create point clouds. Meshes cannot be directly created from a capturing device, but can be created from point clouds and RGB-D images.

Only stereoscopic images can be directly displayed on 3D displays. The other representations use stereoscopic images to display the contained data in 3D, by creating images of the contained data from different viewpoints. The data in a point cloud is often not directly presented to the viewer, as the data cannot directly be visualized as a connected surface. We have explained how the surface the point cloud represents can be visualized with the help of a mesh, or the QSplat system. Unlike a point cloud, we can directly visualize a mesh. There are, however, some variations in how it is done. By applying shading to the faces we can more accurately show the contours of the surface, and by applying a texture to the mesh we can map an image to the mesh's faces.



(a) Wireframe mode

(b) Nonshaded mode

(c) Shaded mode

Figure 2.8: Different visualizations of a simple mesh.

In the next chapter we will explore the different ways to compress the representations mentioned in this chapter.

3

Compression of 3D media

In this chapter we will examine how each of the 3D representations we explored in the last chapter can be compressed. We will also provide an introduction to the different ways to compress data, and provide a short explanation of some general coding techniques.

3.1 Introduction to compression

A compression algorithm works by either representing the data more efficiently, removing unimportant data, or a combination of both. If the decompressed data is identical to the original, we call the compressed lossless. If the decompressed data is missing information contained in the original data, we call the compression lossy. An early example of efficient representation of data is the Morse code, where the most occurring characters are represented by shorter code words. In the English language some characters, like *e* and *t*, occurs more often than the others, and in the morse code these letters are assigned the shortest possible code words. A set of compression algorithm used to compress data by using less data to encode the most occurring symbols are called entropy encoding. The most common entropy compression algorithms are Huffman coding and arithmetic coding, which we explain in section 3.2. These compression algorithms can compress any kind of data, as only the probability distribution of the different symbols is needed.

If we want to remove the unimportant data, we must know the amount of information contained in the different parts of the data for us to separate the important and unimportant pieces. These kinds of compression algorithms must be created specifically for each type of source data. It is normal to remove unimportant

data when we compress media, like images or audio. The hearing range of a person is limited, and an audio recording can contains frequencies beyond the audible frequency range. By removing the imperceptible frequencies we can reduce the amount of data needed to be stored, without making any noticeable changes in the perceived quality of the audio.

When compressing large files, it can take a long time to transmit all the data to the decoder. Dependent on the data being compressed, it can be possible to not wait until the entire data is decoded before presenting it to the user, and use the partially decoded data to show a version of the data with low LOD. These are called progressive compression algorithms, as opposed to the single-rate algorithms. I.e. an image can at first be decoded with a low LOD, and improved as more of the data is decoded. Progressively encoded data can still be lossless, as long as the final result when all the data is decoded is lossless.

The most important factor when evaluating the success of a compression algorithm is the amount of storage space saved, called compressed rate, as it is the main purpose of the operation. When we use lossy compression, the quality of the decompressed data is important as well. If the audio compression removes frequencies within the range of human hearing to achieve a high compression rate, the decompressed audio will be perceived as different to the original.

3.2 Coding techniques

In this section we will take a look at some general coding techniques, which provide lossless compression without needing to be tailored to specifically suit one data type. These are used to further compress the results from some of the compression algorithms explained in this chapter.

3.2.1 Run-length coding

Run-length coding is a basic scheme for reducing redundancy, which represents repeating data as a number followed by the data to be repeated. The character string *zzzzzabbbb* is compressed to *5z1a4b*, using only 6 instead of 10 characters to store the same information.

3.2.2 Huffman coding

Huffman coding uses the estimated probability of each symbol in the source data, and assigns a variable length code to each of the symbols [Huffman, 1952]. The symbols may be the characters in a text file, or other data like strings of numbers or bit sequences. As with the Morse code, the length of the code increases as the probability of the symbols decreases. These codes are prefix-free, meaning that no code is the prefix of another. This removes the necessity of boundaries between

each code, enabling the compressed data to be stored as a sequence of concatenated codes.

If the codes are based on the probabilities of the symbols in a specific dataset, the code table must be stored alongside the compressed data. If the probabilities are based on more general properties, like the probability distribution of the letters in the English alphabet, the table can be known in advance by both the encoder and decoder, and does not need to be transmitted.

3.2.3 Arithmetic coding

Arithmetic coding, like Huffman coding, stores frequently used symbols with fewer bits than the less frequent ones [Witten et al., 1987]. However, it does not convert each input symbol to a code. Instead it converts the entire input, like a text file, into a floating point number greater or equal to zero and less than one.

Before the first symbol in the input is processed the message to be encoded is represented by the interval [0, 1), meaning that 0 is included in the interval but 1 is not. This interval will shrink for each symbol received by the coder, and as we encode more of the message the bits needed to specify this interval will grow. In the end, when the last symbol has been processed, and we have the final interval, the output of the coder is a fraction inside this interval. The entire input message can be reconstructed from this single fraction, without any loss.

The amount the interval shrinks by each added symbol is determined by the probability model. Symbols with higher probabilities shrink the interval by less than the symbols with low probability, and uses therefore fewer bits. The arithmetic coder can either use a probability model with fixed probability values, or it can use an adaptive model. The adaptive model changes the model's probability values during the encoding and decoder, based on the symbol distribution found in the partially encoded or decoded data.

3.3 Image based 3D compression

Compression of image based 3D representations is based on the compression of regular 2D images. There are both lossless and lossy image compression algorithms. The lossless algorithms use compression techniques like the ones mentioned in section 3.2, to better more effectively store the image data. The PNG file format is an example of a popular image format using lossless compression [Boutell et al., 1997]. The lossy compression algorithms remove unimportant information from the image, similar to the audio example in the beginning of this chapter. By removing high frequency information, like sharp transitions between intensity values and color hue, the amount of data can be reduced without necessarily resulting in any visually noticeable effects in the

compressed image. JPEG is an example of such an algorithm, which is widely used to compress images on the internet [Wallace, 1992].

3.3.1 Compression of stereoscopic images

The two main classes of stereoscopic image compression are symmetric and asymmetric compression, both based on methods used to compress regular 2D images. It is also possible to store only one image together with a disparity map of the differences between the image parts, and compress it like an RGB-D image.

Symmetric compression

Symmetric stereoscopic compression applies the same amount of compression to both images in the stereo image pair. Each image can be compressed separately, or joined together as a single image, using the best 2D image compression algorithm available.

Asymmetric compression

Asymmetric stereoscopic compression applies different amounts of compression to each part of the stereoscopic image, and stores one of the images with noticeably lower quality [Gorley and Holliman, 2010]. By exploiting how the human visual system works a stereoscopic image pair can be compressed without storing each image in full resolution, and still be perceived as a high quality 3D image. Each eye has a slightly different view of the world than the other, and these two images are joined to create a 3D representation of the scene registered by the eyes. If the eyes receive clearly different images, the visual system is unable to create a 3D view. It will instead switch from the image on one eye, to the image on the other eye. If a circle is presented to one eye, and a square at the same location is presented to the other eye, they will not appear superimposed to the viewer. Instead, only one of the figures is seen at a time, switching between them.

If the images are similar and one of the images dominates the other, the dominant one will be visible, and the less dominant one will be suppressed. The perceived image is an unbalanced fusion of the two images, where most of the information comes from the dominant image, and the suppressed image only supplements the perceived 3D image.

Based on this suppression theory, one half of the stereoscopic image pair can be highly compressed. The dominant image is a regular image containing all the details of one viewpoint, and the suppressed image is a blurry and low resolution version of the other viewpoint. The human visual system combines the two images into a 3D image of similar quality to the original stereoscopic image.

Disparity map

There is a high correspondence between the data in each image, as both cameras in general capture the same real world objects, from slightly different angles. Instead of storing both of the compressed images, a disparity map can be used [Ahlvers et al., 2004]. A disparity map contains, for each point in one image, the distance to the corresponding point in another image. In other words, the disparity map indicates the way one part of the stereoscopic image must be distorted to generate the other part.

A disparity man can be using in combination with asymmetric coding as well, by using a low resolution disparity map, which will produce a low quality suppressed image [Holliman et al., 1997].

The main image together with the disparity map is very similar to an RGB-D image, as the distances between the points in a stereoscopic image parts depends on their depth in the 3D scene captured by the image. Objects deep in the scene will have a larger distance in the two image parts, compared to objects close to the viewer.

3.3.2 Compression of RGB-D images

RGB-D images are compressed by compressing the RGB and depth data separately, the compression of both types of data based on 2D image compression.

The depth map can be considered a grayscale image where the white pixels are as close as possible in the depth range captured in the image, as seen in Figure 2.2. The two images can be compressed with regular 2D image encoders [Bosc et al., 2010]. Compressing the depth data with a lossy image encoder may, however, not provide the best possible result. Most image encoders are optimized to provide the best visual quality of 2D image data, and are not optimized for depth data [Bosc et al., 2010].

Lossless depth map compression

Some applications need to compress the RGB-D data without any visible errors in the depth data. [Coatsworth et al., 2014] proposed a hybrid lossless and lossy compression scheme for streaming RGB-D data used in urban search and rescue operations. A few pixels encoded with the wrong grayscale intensity may not result in noticeable lower quality in a regular grayscale image, but compression artifacts in the depth data can create unpleasant effects in the decompressed RGB-D image. Errors in the depth data can make parts of objects appear closer, or further away than they really are, and generally make the viewing experience very unpleasant. By lossy compression of the RGB data and lossless compression of the depth data, artifacts in the depth data are avoided.

Locally Adaptive Resolution based compression

[Bosc et al., 2010] attempt to use the Locally Adaptive Resolution (LAR) compression method to preserve the most important parts in the depth map for the generation of virtual viewpoints. Originally a method used to compress regular still images in both a lossy and lossless manner, it focuses on providing high visual quality. It is useful for depth map compression because it accurately preserves the borders of objects, by using more bits to store the area with large differences in the intensity values.

The LAR method assumes an image that consists of two parts, a flat image and a local texture. The flat image contains a set of smooth regions defining the rough features of the image, and the local texture contains the small details for each region. It is assumed that the relevant information needed to compress the depth data is in the flat image.

The flat image is created by quad-tree decomposition, dependent on the local gradient of the pixels in the region. If the gradient is below a certain threshold, the region can be encoded with the average intensity of its pixels. The region is partitioned into four new square regions if the gradient is too high for the region to be representable with a single value. Background areas with small variances in depth can be highly compressed, by only encoding a few large regions. Important parts of the image with a large gradient change, like the edges of objects, use more bits to be correctly stored. [Bosc et al., 2010] compared two virtual viewpoints, one interpolated from two RGB-D streams where the depth map was encoded with the video encoder H.264/AVC, and one interpolated from stream with LAR encoded depth maps. The original RGB image was used, in order to only test the quality of the depth compression. The perceptually based quality evaluation gave the LAR based viewpoint the highest score, in a comparison against the H.264/AVC encoder.

3.3.3 Progressive compression

As we have seen, compression of both stereoscopic and RGB-D images is based on 2D image compression. 2D images can be compressed progressively, which can be used to make the compression of image based 3D progressive.

JPEG supports progressive compression by sending the most important information first [Furht, 1995]. In the early stages of the decoding process an image will look like a very highly compressed image, but it will gain detail as more data is decoded.

Progressive compression of the RGB-D depth data may be undesirable for the same reasons lossy compression of the depth data can be undesirable, as low quality depth data may give unpleasant visual effects.

3.4 Point cloud compression

Uncompressed point clouds can be stored as a list of points in a file, where each line contains the coordinates of the point, and possible additional information.

Most point cloud compression algorithms are progressive [He et al., 2013]. Point cloud can get very large, even when compressed. Progressively compressed point clouds are suitable for internet transmission, so that the viewer do not have to wait until the entire point cloud in received before the visualization of the point cloud is started.

3.4.1 Compression

If we store the points in a specific order, we can predict the location of a point based on the previously stored ones. We can do this by creating a spanning tree over the points, and predict a point based one or more ancestors in the tree branch. A spanning tree of a point cloud is a tree with the points as nodes, containing all the points in the cloud.

3.4.2 Progressive compression

Two possible progressive compression algorithms are the octree algorithm, and the algorithm based on height field approximation [He et al., 2013].

Octree based compression

A point cloud can be organized hierarchically by a process called octree decomposition. An octree is a tree data structure where each internal node has eight children.

The octree is constructed by recursively dividing the cloud's bounding box. A bounding box is the smallest possible imaginary box containing the entire point cloud, with no points outside of the box. The root node of the tree represents the bounding box, and it is divided by partitioning the 3D space into eight cells of equal size. Each nonempty cell is recursively divided until each cell contains only a single point. The octree yields several LODs, one for each level in the tree. By visualizing a cell as a single point, even if several points are contained in the cell, each level above the lowest is an approximation of the point cloud.

The octree is encoded by specifying which cells are nonempty, after a cell subdivision. The configuration of the nonempty cells can be encoded with 8 bits, where each bit corresponds to whether a sub-cell is empty or not. By calculating the probability of the cell configurations based on the configuration of the parent cell and its neighbors, it is possible to encode the most likely configurations with fewer bits.

Height field approximation based compression

A surface can be approximated by representing it as a set of connected height fields. A height field is a 2D data structure representing the elevation of a surface, like the depth part of an RGB-D image. For a 3D surface to be represented by multiple height fields it must be divided into a set of patches, where no points are overlapping when viewed from above. The points in the patch are resampled to a regular grid, where the points are the height above an imaginary plane. The height fields are compressed by a progressive 2D image encoder.

3.5 Mesh compression

A common way to store an uncompressed mesh is as a list of elements, called an indexed face set, either in ASCII or binary. A series of lines with three coordinates each describes the vertices. After the vertices, the faces are described by using the line number of the vertices adjacent to the face. Figure 3.1 shows the indexed face set used to create the model displayed in Figure 2.8. The Object File Format (OFF) is one of several similar data formats, using ASCII characters to define the mesh elements. The first line denotes the numbers of vertices, faces and edges. The next nine lines are the vertex coordinates, and the rest of the lines define the faces. The first number is the number of vertices adjacent to the face, followed by the indices in the list of vertices.

Storing numbers like ASCII values is very inefficient. A decimal number like 1.00005 will require seven bytes, as each digit is stored as a separate character. Using a binary file format will help, where the decimal numbers can be stored as float values, but there is a lot of potential for compression.

In a typical triangular mesh there are roughly twice as many faces as vertices [Ozaktas and Onural, 2008]. This means that for each encoded vertex there will be six vertex references encoded, since a face is defined by its three vertices. Also, the vertex indices only reference existing information, no new information is added like when a vertex's coordinates are added. Many of the compression algorithms try in some way to reduce the amount of vertex indices required, like by changing the order of the stored vertices and faces.

There are many different ways to compress a mesh, and we describe the most popular and well known algorithms in the following sections. Single-rate mesh compression algorithms generally encode connectivity data and geometry data separately. There is a lot of correlation between the variables within the connectivity and geometry data, but there is not much correlation between them. The position of a vertex can be used to predict the position of other vertices nearby, but says little about which nearby vertices it is connected to. Also, the connectivity between vertices says little about the vertex positions. The connectivity and geometry can be stored and processed separately if desired, since the model is

Figure 3.1: Figure 2.8 stored in the uncompressed .off file format.

```
OFF
9 14 21
-1 -1 1
1 -1 1
1 1 1
-1 1 1
-1 -1 -1
1 -1 -1
1 1 -1
-1 1 -1
0 0 -2
3 0 1 2
3 0 2 3
3 4 8 5
3 4 7 8
3 7 6 8
3 5 8 6
3 1 0 4
3 1 4 5
3 2 1 5
3 2 5 6
3 3 2 6
3 3 6 7
3 0 3 7
3 0 7 4
```

not rendered until all the data is received by the application. In section 3.5.1 we present the most popular single-rate connectivity coders, and in section 3.5.2 we explain how the geometry data can be compressed.

Section 3.5.3 presents the most popular progressive coders. We have focused on the compression algorithms with lossless connectivity coding, as they have the most uses. All of the popular first-rate coders have lossless connectivity coding, and only a few progressive coders provide lossy connectivity coding [Caltech et al., 2000] [Gu et al., 2002]. Progressive coders with lossless connectivity coding are more versatile, as the progressive gives us the opportunity to choose between a lossless or lossy result. The decoder would simply need to receive the rest of the data if the original model is needed, and make the decoded model a lossless reconstruction.

3.5.1 Connectivity compression

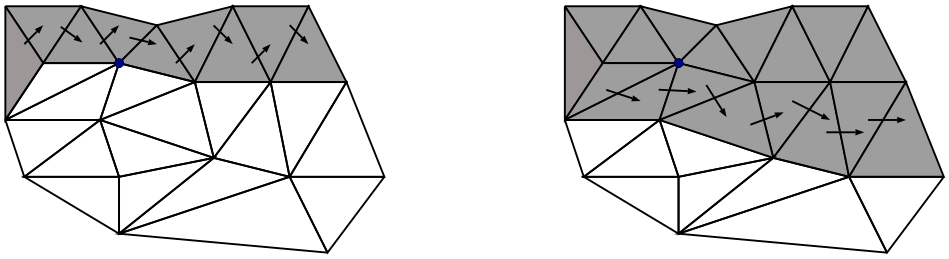
The connectivity compression algorithms try to store the vertices in an order that reduces the amount of redundancy when defining the edges and faces in the mesh.

Triangle strips

A triangle strip is a strip of connected faces with a width of one face. The triangle strip encoders divide a mesh into such strips, and encode them. A triangle strip is a representation supported by most GPUs, and is used for transmitting 3D data

between the CPU and GPU, as an improvement over using an indexed face set. Three vertex indices are needed to specify a triangular face, and many vertex indices will be repeated. The triangle strip methods solve this redundancy by encoding adjacent face in a strip. Two of the vertices in the previously added face can be reused, resulting in a single vertex needed to create a new face. Three vertices are still needed to encode the first face, but the average number of vertex indices encoded per face will be very close to one if the triangle strip is long enough.

Even with this improvement there is still redundant data encoded. A triangle strip will not be able to encode the entire mesh without encoding most vertex indices twice [Deering, 1995]. It would require the strip to circle completely around each vertex before processing the next vertex, which is not possible with a strip-like encoding technique. [Deering, 1995] was the first to introduce a solution to this problem, by using a vertex index buffer. In a large mesh vertex indices may get very large. This buffer contained the last 16 used indices, making it possible to refer to the index in the buffer when possible.



(a) First triangle strip (b) Second triangle strip

Figure 3.2: Creating triangle strips from a mesh

Figure 3.2 shows a possible way to create triangle strips in a mesh. Note that the marked vertex is a part of three faces in the first strip, and three faces in the second strip. The vertex index needs only be referenced once in the first strip, but without the improvement from [Deering, 1995] it must be referenced again in the second strip. With the improvement the vertex index will be contained in the buffer and can be referenced by its buffer index.

Topological surgery

In section3.4.1, we mentioned that a spanning tree could be used to encode a point cloud, which is has the same information as the geometry information in a mesh. [Taubin and Rossignac, 1998] proposed to use a spanning tree to encode a mesh, and use it to encode the connectivity information as well.

The idea of the coder is to use two spanning trees, one for the vertices and one for the faces. At first a vertex spanning tree is created. The mesh is cut along the edges of the spanning tree, hence the name topological surgery. The result is an unfolded

flattened version of the mesh, where each face is situated on the same plane. The triangle spanning tree is generated from the flattened and cut mesh, and the two spanning trees are run-length encoded.

Valence driven approach

[Touma and Gotsman, 1998] proposed an approach based on the valences of the vertices. The valence of a vertex is the number of vertices connected to it by an edge. The valence-driven approach encodes the connectivity of the mesh as a series of vertex valences. The encoding process begins with a seed triangle, its vertices defining an initial borderline. This borderline separates the vertices into the processed vertices behind the borderline, the unprocessed vertices beyond the borderline, and the vertices located on the borderline. These vertices on the borderline are called the active list. The borderline is expanded by adding the unvisited neighbors of the active list's vertices to the active list. The valence of each newly found vertex is output, and the borderline is expanded until every vertex is processed. The series of valences output by the encoder is then coded by an arithmetic coder.

The decoder knows how the encoder finds the seed triangle, and how it expands the borderline. This enables the decoder to replay the role of the encoder and apply the same operations, expanding the mesh a vertex at a time.

The encoding efficiency is dependent on the regularity of the mesh. The arithmetic coder is most effective if the mesh is dominated by vertices with a specific valence, as the high probability of this valence results in few bits needed to encode it.

Triangle traversal

Coders based on triangle traversal are similar to the valence driven approach in the way an initial borderline is created, and the mesh is coded by gradually expanding the borderline a triangle at a time. It differs from the valence driven approach in that it is based on the creation of triangles, like in the methods based on triangle strips, and not on the creation of vertices.

The cut-border machine from [Gumhold and Straßer, 1998] starts with an initial triangle, and iteratively adds new triangles into the processed area within the borderline. The encoded data are symbols denoting how the borderline changed for each added triangle. As with the valence driven approach, the borderline can expand, split or merge with another borderline. An additional index must be coded when the borderline is split, specifying the third index required to do the split operation. In the end the result, which is a small set of symbols and some indices, is encoded by a Huffman encoder.

3.5.2 Geometry compression

A normal way to specify a vertex position is by using a 32-bit floating-point number for each coordinate. This precision is not needed for most applications, and the number of bits per coordinate can easily be reduced without any noticeable quality loss. Instead of using floating points the coordinates can be stored as integers, by a process called scalar quantization. This consists of dividing the bounding box of the mesh into cells, and specifying the coordinate inside the cell by the index of the cell. A bounding box is the smallest possible imaginary box containing the entire mesh, with no vertices outside of the box. The precise position of the coordinate is lost, and is replaced by the center of the representative cell instead. The precision of the quantized coordinates depends on the number of cells created, specified by the number of quantization bits. Using 8 bits results in 256 possible positions in each of the three dimensions. Usually the number of quantization bits is between 8 and 16, which has a large improvement on the storage requirements compared to using 32 bits to specify a coordinate.

The quantization makes the geometry compression algorithm lossy. Even if data is lost, the compression algorithms using coordinate quantization are often called lossless in the mesh compression literature. This is because the quantization creates visually unrecognizable changes, as long as enough quantization bits are used. Nevertheless, the algorithms using coordinate quantization are lossy, as the decoded coordinates are slightly different from the encoded coordinates.

Predictive coding can be used to further compress the geometry of the mesh after the quantization of the coordinates. A correlation exists between adjacent vertices in the mesh, and we can predict the position of a vertex based on the previously added vertices. If a vertex position was correctly predicted, no storage space would be needed for the coordinates. However, this is unlikely, requiring the difference for the predicted position to be encoded. A good prediction scheme results in most of the prediction error values being in a limited range, making some values more likely than others. This makes the prediction errors suitable for compression by an arithmetic coder, the last step of the geometry compression. There are several ways to predict the position of a vertex. Two effective prediction schemes are delta prediction and parallelogram prediction.

Delta prediction

Deering, in his algorithm based on triangle strips [Deering, 1995], used the position of the last added vertex to predict the next one with the method called delta prediction. Adjacent vertices are usually very close, making it possible to use the distance between the two vertices instead of the coordinates of the new vertex. Delta prediction encodes the distance by using the vector between the two positions.

Parallelogram prediction

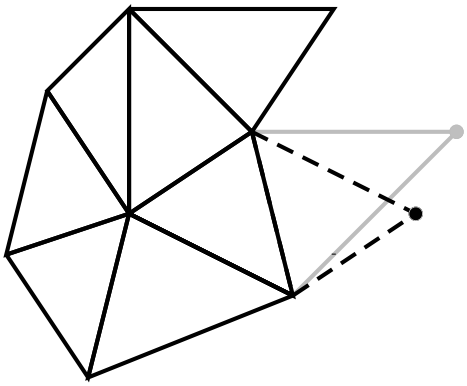


Figure 3.3: Predicting the coordinates of a vertex with parallelogram prediction.

The state-of-the-art connectivity compression scheme proposed in [Touma and Gotsman, 1998] used parallelogram prediction to compress the geometry information. When a new vertex is added it creates a triangle where one edge's vertices are in the active list. This edge is also a part of an existing triangle inside the processed part of the mesh. The parallelogram prediction method predicts that the existing triangle and the triangle created by the new vertex together create the shape of a parallelogram, as shown in figure 3.3. This is an effective prediction scheme if the new vertex is on, or close to, the plane created by the existing triangle.

3.5.3 Progressive compression

Progressive compression of polygonal meshes cannot compress geometry data and connectivity data separately, unlike the first-rate compression methods. Progressively decoding the mesh requires both, since the displayed mesh is improved as soon as more data is received. Progressively displaying the geometry first would be the same as only displaying a point cloud, and the connectivity data alone cannot be used to create a meaningful representation of the mesh. First-rate coders can in general achieve a higher coding gain than progressive coders, because they can exploit the correlation in connectivity and geometry data without having to deal with the other data type.

The progressive compression decoders improve the mesh gradually, from a small base mesh. Each step in the improvement process adds some vertices and faces to the intermediate mesh, and creates a new LOD closer to the original mesh. For the different LODs to be visually pleasing the decoder must know how to change the intermediate mesh to incorporate the newly added mesh elements. This is similar to how progressive compression of point clouds can merge several points into a single new point when displaying the cloud at a lower LOD, but simplifying a mesh must also handle the connectivity changes between the removed vertices and their neighbors.

The encoders based on iteratively simplifying a mesh require the base mesh to be sent to the receiver before the improvements are sent. The size of the base mesh depends of the coder. If it is small, it can be sent as an indexed face set, requiring no additional coders. If it is a larger base mesh, one of the first-rate coders can be used to decrease the time needed to transmit the data before the improvements can begin.

Progressive mesh

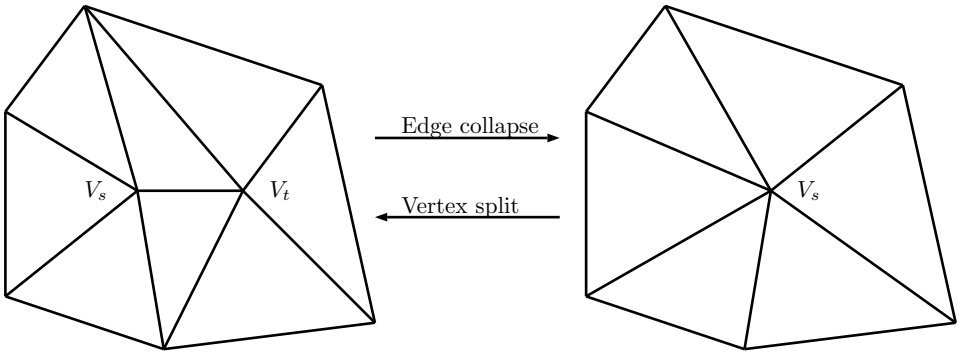


Figure 3.4: Vertex split and edge collapse

[Hoppe, 1996] introduced the first progressive mesh compression algorithm, called progressive mesh (PM). Central to this algorithm are the operations called edge collapse and vertex split.

The edge collapse operator merges two vertices into one, and removes the edge connecting the vertices and the two faces incident to the edge. The remaining edges previously connected to the original vertices are reconnected to the new vertex. The vertex split is the reverse of this operation, which splits a vertex in two and inserts the new edge and its adjacent faces. The neighbors are reconnected to the vertex as they were before the edge collapse changed the connectivity. In Figure 3.4 we illustrate the relationship between the vertex split and edge collapse operators.

The mesh is encoded by doing a series of edge collapse operations. For each operation the quality of the mesh on the encoder side will be a bit worse. In the end a base mesh remains, which can be encoded by a single rate coder. The mesh is decoded by starting with the base mesh, and using the information from the edge collapses to do the vertex splits required to iteratively improve the quality of the mesh. The decoder's operations must be done in reverse order compared to the encoder's operations. The data from the first edge collapse is used to do the last vertex split.

Embedded coding

While PM is based on merging two vertices into one, and removing two faces, others are based on removing a vertex and all its adjacent faces and edges. This is similar to the single-rate valence coder, but the progressive coders require the hole created by the vertex removal to be retriangulated. The vertex with its adjacent faces and edges are often called a patch. [Li and Kuo, 1998] first introduced a coder based on the removal of a vertex, followed by retriangulation of the resulting hole.

The retriangulation requires the connectivity of the vertices on the patch border to be specified. The hole must be filled by new faces and edges after a vertex is removed, and the decoder must know which elements to remove when a vertex is inserted. Before the insertion, all the elements created by the retriangulation must be removed, creating the same hole as created by the vertex removal.

A limited number of possible connectivity configurations exist for a patch. Instead of encoding a list of the patch vertices and the edges connecting them, the index in a table of the possible configurations can be encoded. Two indices are used, a global and a local index. The local index is used to specify the partitioning configuration used to retriangulate a patch. The number of possible configurations increases with the number of vertices in the patch. A triangular patch has only a single configuration while a decagonal patch has 150 different configurations. The global index determines where the patch is located in the mesh, by using the index of one of the new faces in the retriangulated patch.

The position of the vertex is predicted to be the barycenter, also called centroid, of the vertices defining the border of the patch. Obtaining the barycenter of a shape is achieved by averaging the position all the points in the shape.

The mesh is traversed multiple times, and the vertices resulting in the least distortion are removed first. Vertices are removed until no more can be removed without resulting in a non-manifold mesh.

Patch coloring

[Cohen-Or et al., 1999] proposed another compression algorithm based on vertex removal called patch coloring. Instead of removing a vertex at a time ordered by least distortion like the embedded coder, they remove a set of vertices for each iteration of the mesh simplification operation. The set of vertices is independent, which means that no two vertices in the set have an edge joining them. This naturally creates discrete LODs where the vertex removals have occurred evenly across the mesh.

When a patch is retriangulated each face is assigned a color. Each face in a patch has the same color, where the color is different from the colors of the surrounding patches. This enabled the encoder to find the triangles defining the patch, and

remove them before inserting a new vertex. Three, or in some rare cases more, different colors are needed to color the patches, and the color of each face must be encoded. Another more efficient coloring approach uses only two different colors. By re-triangulating the patch in a zigzag way and coloring the inner triangles a different color than the two outer triangles, the patch can be identified by the decoder.

Valence-driven conquest

[Alliez and Desbrun, 2001] proposed a progressive coder based on vertex valences, similar to the single-rate coder by [Touma and Gotsman, 1998]. It shares some similarities with the other compression algorithms based on vertex removals, but manages to achieve a higher coding gain than algorithms like the embedded coders and the patch coloring coder. The vertex locations are encoded with a barycentric error prediction. The connectivity is encoded as sequence of valences from the removed vertices, and a special null code to encode the faces between the patches. It iteratively removes independent sets of vertices, and retriangulates the holes left by the vertex removal. Like the color coder, a complete round of the vertex removal process has evenly degraded the quality across the mesh.

As with the other vertex removal coders, the encoder needs to know the location and shape of the patch to remove before inserting a new vertex. A special mesh traversal method used by both the encoder and decoder enabled the coder to omit the location of a patch, as it is implied by the traversal method. The mesh encoder removes as many vertices with valence less than seven as possible without removing two adjacent vertices in the same iteration. Vertices with a valence higher than six are not removed, to create a statistical concentration around the valence six in the output. This is done to get a better coding gain when using an arithmetic coder on the output of the algorithm. The average valence in a mesh is already six, and an output dominated by a single value is ideal for further compression by an arithmetic coder. Every face not belonging to a patch is encoded as a null patch, and is not removed in this iteration. When a vertex is removed the adjacent vertices are flagged with retriangulation flags. These flags are used to determine the triangulation configuration used when processing the adjacent patches. Since the traversal method is the same for the encoder and decoder, the vertices also get the same retriangulation flags, requiring no information about the shape of the patch to be encoded.

Octree coder

The Octree coder by [Peng and Kuo, 2005] uses the octree data structure to encode a mesh. The vertices in the mesh are partitioned into an octree, and each level in the tree represents the mesh with a different LOD. This is the same idea used in the octree based point cloud compression algorithm in section 3.4.2, but the connectivity between each tree level needs to be encoded as well. Unlike the

previously mentioned coders the octree coder is based on the geometry information in the mesh, as the octree can effectively contain the geometry information. The octree data structure can contain both non-manifold geometry, and polygonal faces. Of the compression algorithms we have discussed in this chapter this is the one providing the highest compression rate, but the octree data structure makes the intermediate LODs look very different to the other coders based on vertex removals.

The first step of the octree coder is to calculate the bounding box of the 3D mesh, and quantize the vertex coordinates. The octree is constructed by having the bounding box as the root of the tree, and the children are created by partitioning the bounding box into eight 3D cells of equal size. The coder traverses the tree from the root and downwards by further partitioning each cell recursively, and encodes the associated local changes in connectivity and geometry.

Empty cells are not partitioned further, so that no storage space is wasted on the empty areas of the mesh. The nonempty cells are recursively partitioned until it is not possible to get a finer resolution, which is defined by the number of quantization bits. In other words, the recursion stops when each nonempty cell only contains a single vertex, and location of the vertex cannot be refined by further partitioning.

The geometry part of the coder encodes the configuration of non-empty child cells when a cell is partitioned. The connectivity coder encodes the connectivity as a series of vertex splits, like the ones used in the PM algorithm. The vertex representing the cell to be partitioned is split until we have a vertex representing each non-empty child cell with the correct connectivity. Two vertices representing cells in an intermediate LOD are connected if there is a vertex contained in one cell that is connected to a vertex contained in the other cell.

3.6 Summary

In this chapter, we have discussed how the 3D presentations we presented in the previous chapter can be compressed. The image based representations are compressed with the help of 2D image compression algorithms. Point clouds and meshes are compressed by ordering the data in specific ways, so that less information needs to be repeated, and data can be predicted from the previously stored data. We have also seen how the representations can be compressed progressively, enabling the decoders to display a low quality version of the data before it is fully decoded.

4

Evaluation of the 3D representations

In this chapter we will compare the 3D representations examined in the previous chapters, and explain why we believe that the triangular mesh is the best representation suited to our needs. We will also select two mesh compression algorithms, which will be the center point of the rest of this thesis, as we will evaluate their suitability for internet transmission.

4.1 Stereoscopic images compared to RGB-D images

Stereoscopic images and RGB-D images are closely related. They share some basic characteristics differing them from the mesh and point cloud representations, which is why we have presented them as two elements in their own class of 3D representations in this thesis. They are both based on the regular 2D image, with the 3D information as an additional effect. Multiple different viewpoints can be generated from a single RGB-D image, and this makes it possible to generate the two viewpoints needed for a stereoscopic image. The reverse is also possible, we can extract the depth information in a stereoscopic image, and create an RGB-D image.

A stereoscopic image can be displayed directly in 3D to a viewer, with the help of a screen capable of showing stereoscopic images. RGB-D images cannot be visualized on such screens, as it would require the screen to show the actual depth information in the image, not just trick the brain into getting a 3D perception of the displayed image.

It can be better to send the stereoscopic image as an RGB-D image, even if it is used as a regular stereoscopic image with a single generated viewpoint-pair. If we

want an accurate 3D perception, we can take into account viewing conditions like the distance to the viewers and the direction from where the viewers are looking at the screen.

4.2 Meshes compared to point clouds

A mesh is a very popular 3D representation, capable of displaying any modeled surface. It is in many ways similar to the point cloud representation, as both are based on representing the shape of a 3D object.

Point clouds can be directly created from the output of a scanner. A mesh's connectivity needs to be generated from the set of sampled points, creating a surface as close as possible to the sampled surface. Meshes are a popular representation in 3D computer graphics, and a mesh can be created manually with 3D modelling software.

As we explained in the previous part a point cloud can be seen as a subset of a mesh. It contains the same data as a mesh without any connectivity information. So, if we have a use case where we need a point cloud representation of a mesh we can simply remove the connectivity part of the mesh data. Generating a mesh from a point cloud is a more expensive operation, requiring the surface to be reconstructed. Using meshes compared to point clouds gives us more options. It can be used in both applications requiring mesh and point clouds, without needing computationally heavy conversion algorithms.

Compression of point clouds is easier than compression of meshes. The progressive point cloud compression algorithms do not have to handle the connectivity changes between the different LODs of the mesh, saving both processing power and storage space. However, for proper visualization of the cloud during the progressive decoding, the visualization process must be done after each new LOD is received. This is an expensive operation, and can create problems if computational efficiency is important, like in real-time applications.

Meshes have the ability to add textures to the mesh surface, while a point cloud has a color value for each point. The color values require less storage than a texture image, but it cannot display the same detail as a textured mesh. If we use a mesh to visualize a colored point cloud, we do not have any information about texture of the faces, and we must colorize the faces based on the colors of the points. For a good visualization of a textures surface, the colored point cloud needs a lot of point to correctly contain the color information, while the texture quality of a mesh is independent of the number of vertices and faces.

4.3 Image based representations compared to model based reprentations

The image based representations differs in an important way from the representations based on proper 3D models. An RGB-D image can be though of as a point cloud where the coordinates of the points have been captured from a single direction only. It is not possible to see the backside of an object captured in a RGB-D image, as it is not a proper 3D representation. The contained depth information only gives the viewer depth perception from one direction. An RGB-D image can generate several different viewpoints, but they are restricted to be within a small angle. Both the mesh and point cloud can show the entire 3D shape of an object, and the two viewpoints needed to get a stereoscopic image can easily be generated from any angle.

The ability to represent an entire 3D model as a single entity is the major benefit of using a mesh or point cloud over the image based representations. A possible use case is that we want to display a 3D model on a server to a viewer wearing a HMD. We could send an RGB-D image capturing the model to the HMD, and update the stream when the viewer changes viewpoint. This would require very low latency for it to handle quick movements of the head, as the server would need to receive the new location, generate a new RGB-D image, and send the updated image back to the viewer. Stereoscopic images needs to be generated from both an RGB-D image and a 3D model if the user wants true 3D perception, and not just a projection of the 3D data onto a 2D screen.

If the 3D object is far away from the viewer, and the viewer's movements are restricted, the viewer may not be able to move to the other side of the object and see the full model. In this case an image based representation can show enough of a models surface, without the viewer noticing any missing information.

It is possible to use free viewpoint video, and send enough viewpoints to cover the entire model. This requires a lot of data, as one RGB-D image only covers a small set of viewpoints. Some viewpoints may contain errors, as synthesized views may not be entirely correct. If accuracy is important another 3D representation is better.

4.4 Choosing the representation and algorithms

In this section we explain our reasoning behind using a triangle mesh as out chosen 3D representation. Each representation has a set of distinct qualities, and the choice of a representation will depend on the intended use cases. We have chosen to decide with the perspective of 3D media transmission over the internet. In section 4.4.1 we explain why we feel that the triangle mesh is the representation best suited to out needs. In section 4.4.2 we decide on the octree coder and the valence-driven conquest as the chosen compression algorithms.

4.4.1 Choosing the triangle mesh representation

Based on the comparison between the 3D representations we find that meshes are the most interesting choice. The image based representations can easily show the surface of an object, but only from a limited angle. For us to view more angles we need to use multiple images, while the mesh representation only need one connected model.

A point cloud can represent a full 3D model, but its lack of connected surface has its drawbacks. We cannot properly model a surface with a point cloud. As we have seen, point cloud visualizations use mesh-like techniques to visualize its modeled surface.

Most mesh compression algorithms are based on triangular meshes. Of the algorithm we saw in section 3.5, only the octree based compression algorithm supports compression of meshes with faces of a higher degree than three.

Triangular meshes are the most versatile option, as a set of triangles can recreate any geometric shape. This means that it is easy to convert meshes containing faces of a higher degree than three to pure triangular meshes. We have decided to work with triangular meshes so that we do not restrict us needlessly in the selection of compression algorithms.

4.4.2 Selecting two compression algorithms

Algorithm	Bits per vertex (bpv)
Progressive mesh [Hoppe, 1996]	37 [Maglo et al., 2015]
Embedded coding [Li and Kuo, 1998]	20* [Peng et al., 2005]
Patch coloring [Cohen-Or et al., 1999]	25 [Peng et al., 2005]
Valence-driven conquest [Alliez and Desbrun, 2001]	21 [Maglo et al., 2015]
Octree coder [Peng and Kuo, 2005]	15 [Maglo et al., 2015]

Table 4.1: Compression efficiency of progressive mesh compression algorithms. **The bpv value for the embedded coder is the bpv required to reconstruct the first third of the vertices and faces only. In the later improvements a significant part of the bits is used to improve the vertex coordinates, and not used to add more vertices.*

As with the other representations, the mesh compression coders can be split into single-rate and progressive coders. Single-rate coders are more effective as they do not need to spend any data on the connectivity of the intermediate LODs, requiring the whole mesh to be sent before the receiver can create the mesh. When sending a large model over the internet the transmission time may be too long for an impatient viewer, making the progressive coders more useful. The total transmission time is longer, but the progressive model can early on display the mesh at a lower LOD and update it when more information arrives. This makes the progressive algorithms a clear choice for finding compression algorithms suitable for mesh transmission.

The progressive coders can be further split in to the coders based on the connectivity and the geometry. We believe it is interesting to see the differences in the implementations of a connectivity based coder and a geometry based coder.

In table 4.1 we list how efficient the progressive compression algorithms are related to each other. We have chosen the Valence-driven conquest from [Alliez and Desbrun, 2001] as our connectivity based progressive compression algorithm, as it has the highest compression rate of the connectivity based compression algorithms. The geometry based compression algorithm we discussed was the octree coder by [Peng and Kuo, 2005]. Of the all of the progressive compression algorithms we discussed it has the highest compression rate, but it needs a separate data structure to create the different LODs, unlike the valence based algorithm which works directly on the mesh. We will explain the design and implementation of the valence based algorithm in chapter 5 and 8, and we will explain the design and implementation of the octree based compression algorithm in chapter 6 and 9.

4.5 Summary

In this chapter we have compared the different representations, and chosen the triangle mesh as our 3D representation. The triangle mesh can best contain the entire surface of a 3D object using only a single mesh. After we chose a representation we chose two compression algorithms to compare. We chose the valence based algorithm by [Alliez and Desbrun, 2001] and the octree based algorithm by [Peng and Kuo, 2005], because of their high compression rate and their different ways to create the intermediate LODs.

Part II

Designs and implementations

5

Design of the valence coder

5.1 Introduction

The valence based compression algorithm is based on the removal of vertices, where the encoder gradually decreases the quality of a mesh. The decoder reverses the operations by gradually refining a low quality mesh until it is identical to the original. We briefly described this progressive compression algorithm in section 3.5.3. We have not included the arithmetic coding of the data output by the encoder, to lessen the scope of the implementation.

The encoder creates several layers of mesh improvements, where each layer provides the decoder with the improvements needed to get the next LOD and closer to the original mesh. The encoder creates a layer by removing an independent set of vertices from the mesh, and retriangulates the holes created by the missing vertices. An independent set of vertices is a set where none of the vertices is adjacent to another vertex in the set. A single vertex is removed at a time, and the information needed for the decoder to insert the vertex back again is encoded, enabling granular improvement of the decoded mesh. The decoder applies the improvements onto a base mesh received from the encoder, which is the lowly detailed mesh remaining after the improvement layers are generated.

The first layer created by the encoder contains the information the decoder needs to get the decoded mesh from an approximation of the original mesh, to an accurate and lossless reconstruction. This means that the first layer created by the encoder must be the last layer received by the decoder. When the encoder can no longer create any more layers, the remaining base mesh is compressed by a first-rate compression algorithm of choice. The layers of improvements are sent in the

reverse order from the order they were output by the encoder, after the compressed base mesh.

The first removed vertex in a layer is the first vertex to be added by the decoder. This means that the encoder can be interrupted in the middle of a layer, and send the data encoded so far together with the base mesh. The decoder can then add the vertices removed before the encoder was interrupted, and apply any layers created earlier by the encoder.

The output of the encoder is a stream of valence codes, together with the geometry information of the removed vertices. The algorithm uses two different mesh traversals, called conquests, to remove a set of vertices. The general conquest removes an independent set of vertices from the mesh, but the resulting mesh is speckled with vertices of valence three. This happens because the algorithm tries to maintain the regularity of a mesh between the conquests. A mesh is regular if the vertex valences are centered around one value. Regular meshes are usually centered around the value six [Alliez and Desbrun, 2001]. If the coder created irregular intermediate LODs from a regular mesh, the visual quality of the LODs will not be as good as possible, as the irregular surface may seem cluttered and uneven compared to a smooth and regular mesh surface. It may seem counterproductive to end up with a mesh with many vertices of valence three if we want to maintain the regularity. To solve the problem of the valence three vertices we use another type of conquest called a cleaning conquest. It removes these vertices, resulting in a regular mesh, as long as the mesh was regular before the encoder started the general conquest. The decoder must first reverse the cleaning conquest, and then reverse the general conquest.

Section 5.2 contains a set of definitions used when explaining the design. Section 5.3 explains how the general conquest is used to encode the connectivity, and section 5.4 explains how it is used to decode the connectivity. Section 5.5 explains how the cleaning conquest differs from the general conquest. At the end, in section 5.6, we explain how the coordinates of the vertices are encoded.

Since this algorithm is based on an expanding borderline, it will not compress any disjoint pieces of the mesh. If we want to compress a mesh consisting of multiple pieces, we must start the compression process anew for each of them.

In this chapter we do not explicitly mention the creation of edges, and assume they are implicitly created and removed together with the creation and deletion of faces.

5.2 Definitions

Null patch A single face not belonging to any patches.

Patch A vertex, and its incident faces and edges. It has a degree, which is the valence of the vertex.

Vertex removal A vertex of valence n is removed, along with its incident faces and edges. The resulting hole is filled with $n - 2$ new faces. This operation is used by the encoder.

Vertex insertion n-2 faces are removed, and the resulting hole is filled with a vertex and n new faces. This operation is used by the decoder.

Retriangulated patch A patch after its vertex has been removed.

Gate An oriented edge, with a reference to its front face and front vertex, as illustrated in figure 5.1. We traverse the mesh with the help of a queue of gates. We use a gate to access a patch, and after a patch is processed we find new gated pointing to new patches. This is shown in figure 5.4.

State flags Flags used to set the states of faces and vertices. The state can be either free or conquered.

Retriangulation flags Each vertex is flagged plus or minus, to either maximize or minimize its valence when a patch is retriangulated.

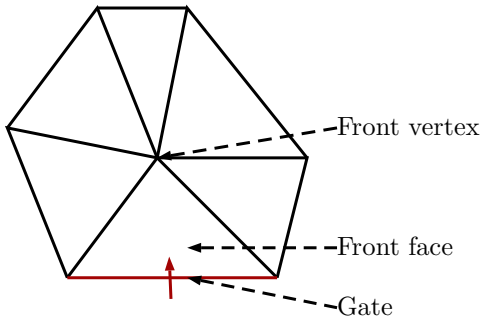
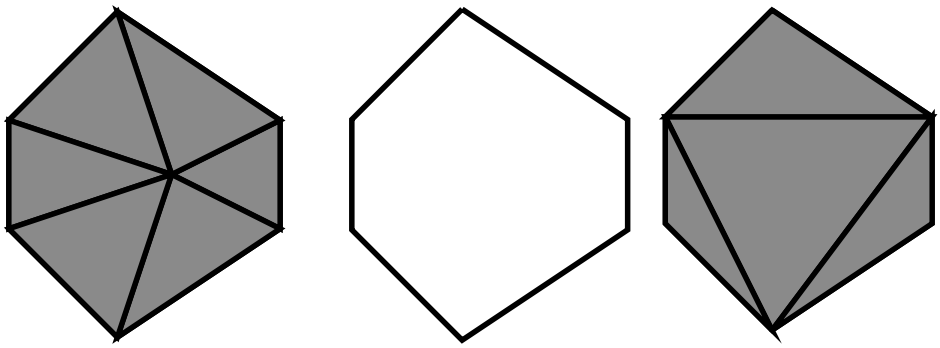


Figure 5.1: A gate with its front face and front vertex.



(a) Original patch (b) Emptied patch (c) Retriangulated patch

Figure 5.2: The three stages of a path during a vertex removal, from left to right. A vertex insertion is the opposite operation, starting with the figure on the right hand side, and ending up with the figure on the left hand side.

5.3 Encoder's general conquest

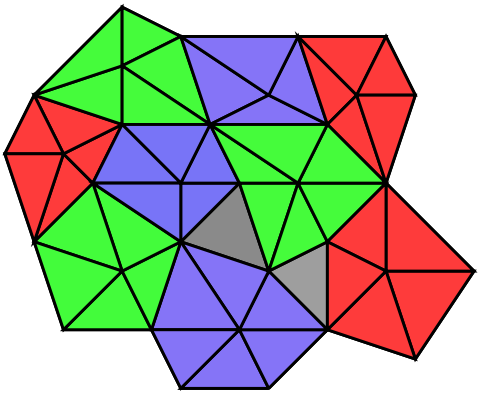


Figure 5.3: A mesh divided into patches. The colored patches are regular patches, while the two gray faces are null patches.

The removal of an independent set of vertices is called a conquest. We remove one vertex at a time, and when it is removed, we need to remove its adjacent faces as well. We call the vertex together with the set of faces a patch. After the patch has been emptied, we need to retriangulate the resulting hole. The empty patch must be filled with faces for the intermediate LODs to be visually appealing. Figure 5.2 shows the three stages of a patch during the removal of a vertex.

The conquest of a mesh is done by viewing the mesh as a set of connected patches. Ideally, we would be able to divide the mesh so that every face is contained in a patch, and do a vertex removal on each patch. Unfortunately, it is often not possible to perfectly partition the mesh into patches. The null patches are created when an optimal set of patches cannot be found, encoding the space between the regular patches. Each face belongs to either a regular patch or a null patch, as shown in figure 5.3. Each colored patch will be emptied and retriangulated during a conquest. The two gray faces are the null patches, and will be untouched by the encoder and decoder, at least until the next conquest.

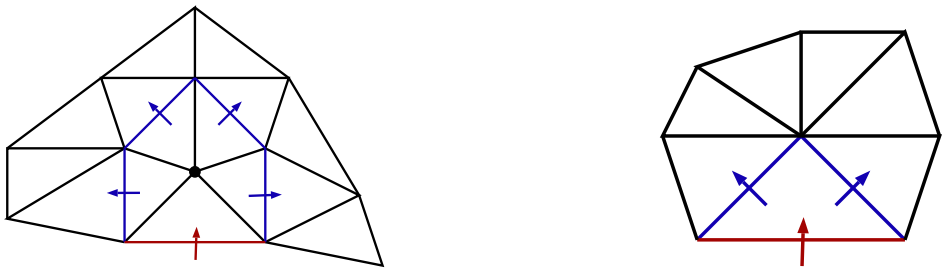
The encoder outputs a stream of codes, which either specifies the valence of the removed vertex, or the occurrence of a null patch. If a vertex valence is output, it is followed by the coordinates of the removed vertex. An excerpt from the stream of values output by the encoder might look like this: $valence\_number - vertex\_coordinates - null\_patch - null\_patch - valence\_number - vertex\_coordinates$. In this example two vertices are removed, and two null patches are processed.

Only vertices with valence of six or less are removed. Removing a vertex with a valence less than six results in a retriangulation where the sum of the neighborhood vertices' valences is less or equal to the original sum [Alliez and Desbrun, 2001]. If we retriangulate a patch with a higher degree than six, the sum of valences is more than the original sum. If we create more edges, there is more connectivity

data to encode, and it will adversely affect the compression rate.

5.3.1 Traversing the mesh

The conquest is done with the help of a queue of gates. A gate points to a new patch, and is the driving force behind the mesh traversal. After a gate is pulled from the queue, the patch belonging to the edge is processed. When a patch of degree n is processed, $n-1$ gates are added to the queue, as seen in Figure 5.4(a). When a null patch is processed, two gates are added. The gates are added in counterclockwise order, starting with the gate after the red incoming gate.



(a) Regular patch (b) Null patch

Figure 5.4: The patches and their gates

The gates are processed until there are no more gates in the queue, which means that every face and vertex in the mesh has been visited.

The conquest starts with an initial gate. The initial gate's vertices are flagged conquered, and the gate is added to the queue. The retriangulation flags of the gate's vertices are also set to an initial value, whose purpose will be explained in section 5.3.4.

5.3.2 Processing a gate

There are three possible ways to proceed when a gate is pulled from the queue, dependent on the gate's front vertex and front face.

1. The front face is flagged as conquered.
2. The front vertex is flagged free and its valence is six or less.
3. The front vertex is flagged conquered, or it is both flagged free and has a valence of more than six.

The first option means that the patch this gate belongs to has already been processed, and nothing more needs to be done with this gate.

The second option means that the front vertex can be removed. The vertex removal process is explained in section 5.3.3. Afterwards the patch is retriangulated, as explained in section 5.3.4

The last option means that we cannot remove the front vertex because is on the borderline of another patch, or the valence of the vertex is too high. A zero is output, the front face is flagged conquered, and two new gates are added to the queue.

5.3.3 Vertex removal

When a vertex is removed, the patch is emptied, and the information needed to recreate the patch is encoded. The valence of the front vertex is output, together with the vertex's geometry information. The geometry encoding is explained in section 5.6.

5.3.4 Patch retriangulation

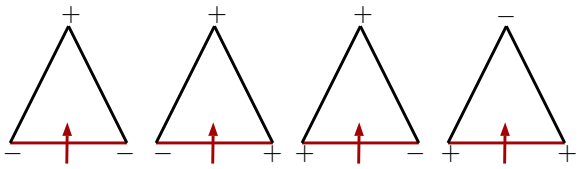
The next step after the encoder empties a patch is to retriangulate the resulting hole. The important part of the retriangulation step is to create a configuration of faces so that the decoder can find the patch. The encoder can easily get the patch faces by finding the faces adjacent to the front vertex, but the decoder must find the faces created by the encoder's retriangulation function.

The decoder needs two pieces of information to find a retriangulated patch. It needs the degree of the patch, which is provided by the encoded valence number. It also needs to know the configuration of the faces, which is found without requiring any more data.

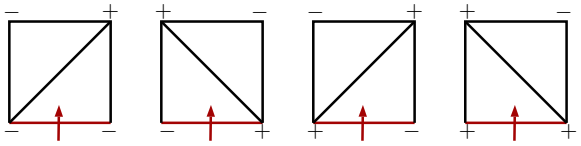
By flagging each vertex with a retriangulation flag during the conquest, we can use these flags to know the configuration of the retriangulated faces. The deterministic conquest enables the encoder and decoder to process each patch in the same order, meaning that each vertex will have the same retriangulation flag on the encoder and decoder side. The encoder uses the flags to know which of the vertices on the patch borderline defines each new face, and the decoder uses the flags to traverse the retriangulated faces defining the patch.

Figure 5.5 shows how to retriangulate the patches. The $-$-sign and the $+$-sign adjacent to each vertex are the retriangulation flags. They signify if the valence of a vertex should be minimized or maximized during the retriangulation. This is done so that we can maintain the regular property of a mesh, as mentioned in the introduction. The configuration of faces is dependent on the flags of the vertices incident to the gate, and the figure shows each of the four different possibilities for the different patch degrees. We set the retriangulation flags for the neighboring vertices according to the flags beside each vertex in the illustration.

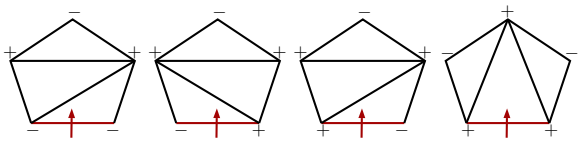
Some of the neighboring vertices may be part of other patch borderlines, and already have their retriangulation flags set, conflicting with our configuration. We do not change the retriangulation flags of these vertices, but the triangulation is still done the way illustrated in figure 5.5.



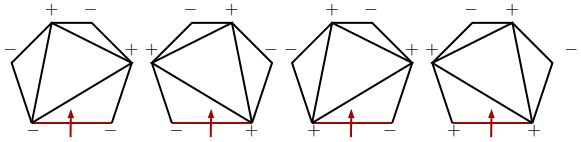
(a) Patch degree 3



(b) Patch degree 4



(c) Patch degree 5



(d) Patch degree 6

Figure 5.5: Retriangulation configurations

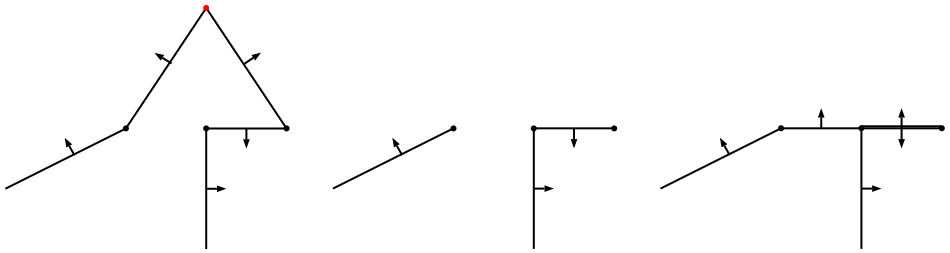
The vertices of the new faces are ordered counterclockwise, maintaining a consistent orientation of the mesh faces.

It is not always possible to retriangulate a patch. Emptying the patch will always work, as you cannot create non-manifold or non-orientable geometry by simply creating a hole in the mesh. However, attempting to retriangulate the hole may result in these errors. We can either end up with non-manifold or non-orientable geometry straight away in the retriangulated patch, or an error may occur later in the conquest. We do not want to loose the manifold and orientable properties of the intermediate LODs, as it will create clearly visible errors.

Non-orientable or non-manifold retriangulation

In figure 5.6 we show a vertex removal resulting in both a non-orientable and non-manifold mesh. The hole is filled by two faces, one on the left side, and one on the right side. A face already exists where the right hand side face is added, and is adjacent to the left hand side face.

The result is that two adjacent faces have opposite orientations, making the mesh non-orientable. One edge is incident more than two faces, making the mesh non-manifold as well.



(a) Before the vertex removal (b) Patch emptied (c) Attempted retriangulation

Figure 5.6: A vertical cross section of a mesh during a vertex removal resulting in non-manifold and non-orientable geometry. Each line is a face viewed from the side, and the dots are the vertices. The arrows show the orientation of each face, and the red vertex shown in (a) is the vertex to be removed.

Fold-overs

(a) Convex patch (b) Concave patch

Figure 5.7: Retriangulation of convex and concave patches. The patches are empty, and the first retriangulated face is shown by the stippled line. The numbers show the order of the vertices used to create the face, defining the orientation of the face.

The shape of a patch can be either convex or concave. In a convex shape all the inner angles are less than 180 degrees, illustrated in figure 5.7. A retriangulated concave patch can give us an undesirable result without creating non-manifold or non-orientable geometry. A face created during the retriangulation of a concave patch can cross the patch borderline. As we can see in figure 5.7(b), the retriangulated face will be created outside of the patch borderline. As we have previously explained, the orientation of a face is defined by the order of the vertices used to create it. In the convex patch in figure 5.7, the new face's vertices are in counterclockwise order. In the concave patch, however, the vertices are in clockwise order, resulting in the opposite orientation. The result is that the face is folded on top of the faces outside of the patch, with the front of the face pointing the opposite direction of the surrounding faces. A face may connect with this upside down face in a later stage of the conquest, and create a situation similar to the non-orientable and non-manifold geometry shown in figure 5.6.

How to avoid non-manifold and non-orientable geometry

The retriangulation problems can be solved by aborting the vertex removal if the retriangulation cannot be done without creating non-manifold or non-orientable geometry. This requires the vertex removal to be reverted, so that the state of the patch is the same as before. We avoid the vertex removal by processing the front face as a null patch instead.

The fold-over problem requires us to detect when a fold-over has happened, even when it keeps the manifold and orientable properties of the mesh.

Our solution is to check if the patch is concave before we process it, and if so, process the front face as a null patch. We can find out if a patch is concave by checking if the clockwise angle of any three adjacent vertices on the patch borderline is less than 180 degrees. This is an easy operation in 2D, as we only see the angles from one side. However, in 3D, we can see the patch from both the front and the back. From one direction we will find the clockwise angle, while from the other side we will find the counterclockwise angle. Each angle in a patch is defined by three vertices, and we need to check each angle in the patch. We also need a vector specifying our viewing direction, for us to find the correct angle. We cannot use the normal vector of one of the retriangulated faces as a reference. If the face is folded over, its normal vector will point in the wrong direction, and we will falsely detect a concave patch as convex. Therefore, we use the surface normals of the two faces in the original mesh adjacent to the edges defining the angle as the reference viewpoint.

Another solution could be to not detect the cause of the problem, but to detect the effect. We can check the surface normals of each face after the path has been retriangulation. If one of the normals differs largely from the others, it can indicate the face being upside down. The problem with this solution is that patch configurations without fold-overs can have faces with almost opposite normals, which is why we did not chose this solution for our implementation in chapter 8.

5.3.5 Traversing the patch borderline

After a successful patch retriangulation, we must find the new gates, and set the flags for the vertices neighboring the front vertex. A simple solution would be to do this before the patch is removed. Then, if the retriangulation did not work, we would need to revert the neighboring vertices' state and retriangulation flags to their original values, and remove the added gates from the queue. Therefore, we traverse the neighborhood after the retriangulation by using an array containing the neighboring vertices, obtained before the patch was removed.

5.4 Decoder's general conquest

The decoder uses the same gate system as the encoder to traverse the mesh. An edge used as a gate by the encoder is not removed later in a conquest, since the set of removed vertices is independent. This means that the decoder can use the same edges as gates, and reverse the vertex removals. The retriangulation flags are applied to the vertices the same way as done by the encoder. After a gate is pulled from the queue, the decoder must retrieve a new valence code from the encoded data, for it to figure out how to process the gate.

As with the encoder, the gate is ignored if the front face is conquered. If the patch is not ignored, a valence code must be retrieved from the encoded data. There are two possible options for the next step, dependent on the value of the valence code.

1. The valence code is between three and six:
   The gate belongs to a regular patch. We first find the faces defining the patch, empty the patch, and insert a vertex. How to find the patch is explained in section 5.4.1, and how to insert the vertex is explained in section 5.4.2.

2. The valence code is zero:
   The gate belongs to a null patch. The mesh remains unchanged, and the gates and flags are handled the same way as by the decoder.

5.4.1 Discovering the retriangulated patches

The decoder finds the faces defining a retriangulated patch by using the retriangulation flags of the gate's vertices. We know the face configuration used in the retriangulation, as the gate flags are the same for the encoder and decoder.

We can traverse the faces in the patch by starting with the face adjacent to the gate, and find the next faces according to the configuration table in figure 5.5. Say that we want to find the faces in a patch of degree five, and the gate vertex flags are $+$ and $+$. Then we look up the correct configuration in the table, and find that the two next faces to add are the faces of both sides of the first face.

After the patch is discovered, we add new gates from the patch borderline in the same way as done by the encoder, and we remove the contained faces.

5.4.2 Vertex insertion

After we have discovered a retriangulated patch, we do a vertex insertion. We delete the retriangulated patch faces, and insert a vertex with the coordinates stored in the encoded data. The patch is filled with faces by adding a face for each pair of adjacent faces on the patch borderline, connected with the newly created vertex. The decoder does not need to consider the possibility of non-manifold or non-

orientable geometry when filling the patch. The vertex insertion recreates a patch configuration which successfully existed in the encoder's mesh.

5.5 Cleaning conquest

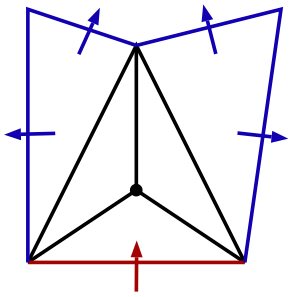


Figure 5.8: A cleaning patch and its gates

A general conquest is followed by a cleaning conquest. It is very similar to the general conquest for both the encoder and decoder, except for a key difference. While the encoder's general conquest removes any vertices with a valence less than six, the cleaning conquest only removes vertices with the valence three. Any patch with a degree other than three will result in a null patch. The retriangulation in the general conquest produces a mesh containing many vertices with valence three, if the mesh is regular. Figure 10.4(b) in the evaluation chapter shows how a regular mesh looks like before the cleaning conquest removes the valence three vertices. The mesh is traversed a bit differently compared to the general conquest, so that the correct vertices can be found without creating more null patches than necessary. Instead of adding the edges on the patch borderline as new gates, the edges surrounding the two adjacent faces are added, as shown in figure 5.8. The decoder uses the same traversal method as the encoder when receiving the output of the encoder's cleaning conquest, but can otherwise process the encoded data like during a general conquest.

5.6 Geometry encoding and decoding

Before the encoding begins, the coordinates are quantized, which reduces the number of bits used to store the coordinates without creating any noticeable changes in the quality of the mesh.

5.6.1 Coordinate quantization

Quantization is the process of mapping continuous values into a set of discrete values, where the set has a restricted size. Uncompressed vertex coordinates are

usually stored with three float values, one for each axis. The coordinates are quantized with the help of the mesh's bounding box. The bounding box of a mesh can be found by iterating though each vertex, and finding the minimum and maximum values of the x, y and z coordinates. These values define the location of the corners of the bounding box.

By dividing the bounding box into cells, we can assign each vertex a cell, and use the cell index instead of the vertex's coordinates to denote its location. The number of cells depends on the number of bits used to quantize the mesh. Ten quantization bits gives us a range of 1024 different coordinates.

The number of bits needed to store a coordinate is the number of quantization bits, which is normally markedly less than a 32 bit float value. It is important that the number of quantization bits is high enough so that no two vertices are assigned the same index. Then these two vertices would have the same coordinates, and create problems for the faces surrounding them.

5.6.2 Barycentric prediction

The location of a vertex is encoded using barycentric prediction. A barycenter is the center of a set of geometric objects. Each removed vertex is surrounded by a ring of neighbors that will remain through the current conquest. This enables both the encoder and decoder to use the neighborhood to calculate the barycenter. The location of a vertex can be encoded by storing the difference between the vertex's location and the barycenter.

5.7 Summary

In this chapter we described the valence based progressive mesh compression algorithm, based on the paper by [Alliez and Desbrun, 2001]. The algorithm is based on the removal of independent sets of vertices, called conquests. A conquest removes a vertex at a time, and retriangulates the resulting hole, until an independent set is removed. Both the encoder and decoder traverse the mesh in the same way, but the operations done by the encoder is based on the state of the mesh, while the decoder base the operations on the data received by the encoder. The encoder must do a check to see if each vertex can be safely removed as well, so that the retriangulation does not create non-manifold or non-orientable geometry. The conquests are done until a small base mesh remains. The decoder receives the base mesh and reverses the vertex removals done by the conquests.

6

Design of the octree coder

6.1 Introduction

In this chapter we explain how the octree based compression algorithm works. The octree coder is a compression algorithm based on the octree data structure, as we briefly explained in section 3.5.3. As with our design of the valence coder, we have omitted the arithmetic coding of the data output by the encoder.

Unlike the valence coder, we do not work directly on the mesh when generating the different LODs. Instead, we build an octree data structure based on the mesh, and iteratively expand this tree. We generate a mesh from the tree, its LOD dependent on the tree's depth. A mesh can be created from the octree by representing each leaf cell as a vertex, positioned in the center of the cell's bounding box. The cell's neighbor relations define which vertices are connected by edges. Two cells are neighbors if a vertex in one cell is connected by an edge to a vertex in the other cell.

After the decoder has processed all the encoded data, the decoder has the exact same data structure as the encoder had. From this data structure we create a mesh, its LOD dependent on the number of tree levels created by the encoder. The vertex coordinates are quantized before the encoding begins, the same way as done by the valence coder.

In the first section we take a look at how the encoder and decoder create the octree data structure. In the following sections we explain how the geometry information is encoded and decoded, and how the connectivity information is encoded and decoded.

6.2 Generating the octree

The first step when creating the octree data structure is to create the root cell of the tree. The encoder's root cell contains every vertex and is the same size as the bounding box of the quantized mesh. The bounding box of the decoder's root cell is the same size as the encoder's, its size derived from the number of quantization bits. The lowest LOD of the mesh is represented as a single vertex, consisting only of the root cell. The next LOD after the root cell is split contains up to eight vertices.

One tree cell is divided at a time. The order of cell divisions is based on an importance value, where the most important cells are divided first. The importance value I is defined by the following formula, where s is the size of the cell's bounding box, v is the number of neighbors, and d is the average distance to the cell's neighbors.

$$I = s \times v \times d$$

When a cell is split, the cell's bounding box is partitioned into eight parts of equal size. Each part is represented by a new octree cell. If a part contains no vertices we do not need an octree cell to contain it, since further partitioning an empty cell gives us no more information about the mesh. This partitioning replaces the parent cell's representative vertex with up to eight new ones. We need to both encode the location of the new cells, and their neighbor relations.

The encoder and the decoder do the same split operation. The way the encoder splits a cell is dependent on the contained vertices and their relationships to vertices in neighboring cells. The decoder splits a cell based on the data output by the encoder.

6.2.1 On the encoder

The cell's bounding box is partitioned into eight pieces, and each child cell is assigned one part and the contained vertices. After the partitioning, the encoder sets the neighbor relations for each new child-cell.

The location of the non-empty child cells must be encoded, together with the child cells' neighbor relations. The parent cell's neighbor relations are removed, since the parent cell has been replaced by the new child cells.

6.2.2 On the decoder

On the decoder, the cell's bounding box is partitioned into eight pieces, but only the non-empty child cells are created and assigned parts of the bounding box.

The configuration of the non-empty child cells is obtained from the encoded data. After the cells are created, the child cells' neighbor relations are obtained from

the encoded data. As with the encoder, the parent cell's neighbor relations are removed. The result is an octree a step closer to representing the original mesh.

6.3 Encoding the geometry

When the decoder divides a cell it needs to know which child-cells contains vertices, and which are empty. The encoder stores this information by outputting the number of non-empty child cells, and their configuration. A prediction technique is used to compress the child cell configuration, which is explained later in this section.

6.3.1 Encoding number of child cells

The encoder finds the configuration of non-empty child cells by creating eight new child cells, and partitioning the parent cell's vertices into these cells. After all the vertices are assigned to a child-cell we can remove the empty child cells. The encoder outputs the number of remaining child cells, which is between one and eight. There will always be at least one non-empty child cell, since we do not try to divide empty cells.

The next step is to find the configuration of the cells. We can skip this step if the number of cells is eight, as there is only one possible configuration when every cell is non-empty.

6.3.2 Encoding configuration of child-cells

The configuration of non-empty child cells can be stored as eight bits representing the parent cell's array of child cells, where each bit indicates whether a cell in empty or not. From these bits the decoder knows the locations of the new cells, as the array of child cells are ordered based on their position in the parent cell's partitioned bounding box.

The bit sequence can be stored more effectively by considering that some configurations are more probable to appear than others. By sorting the list of possible sequences by probability we can encode the configuration by using the index in the list. Both the encoder and decoder know the number of non-empty cells, so the list of configurations needs only contain configurations with the specified number of non-empty cells. The use of an index enables us to use fewer bits per configuration.

The probability calculation must give the same result on both the encoder and decoder, so that the index used to access the list returns the same configuration. We calculate the probability by assigning each child-cell a priority value which

estimates the probability for it to be non-empty. After the probability values are found the list of configurations can be sorted.

Calculating the cell priority values

The estimated probability value used to set a cell's priority is based on the observation that non-empty child cells tend to be close to the center of the parent cell's neighbors. The priority value of a child cell is found from the number of neighbors close to the cell, and their distances to it. More specifically we find the distance between the child cell and neighbor cells contained in the same part of the mesh.

Bipartitions of the neighbors along each axis are used to find the neighbors in the vicinity of each child cell. The bipartitions are found by dividing the set of cells along the centroid of the parent cell, once for each axis. The three bipartitions creates eight unique combinations of bipartition indices, one for each child-cell.

Each bipartition part is assigned a distance value, and the distance values of a cell's three associated bipartition parts are used to find the cell's priority value. The distance value of a bipartition part is the sum of distances to the parent cell's center along the partitioned axis. Each cell distance is multiplied with the octree level of the cell as well. Cells at the lower levels of the tree provide more accurate information, and therefore get a higher weight in the sum of distances.

The following is the formula for finding the distance value to a bipartition:

$$D_{i,j} = \sum_{k=1}^{n_{i,j}} l_{i,j,k} \times d_{i,j,k}$$

$D_{i,j}$ is a distance value for a bipartition part, where i is the axis and j is the bipartition index. $d_{i,j,k}$ is the distance along axis i to the cell k in the bipartition, and $l_{i,j,k}$ is the octree level of the same cell.

The priority value of a child cell is the sum of the three associated bipartition parts' distance values multiplied by weighing values. The weighting value is based on the unbalance of a bipartition compared to the other two. The priority value P_k is found by the formula

$$P_k = \sum_{i=1}^{3} w_i \times D_{i,b}$$

,

where w_i is the weighting value, and $D_{i,b}$ is the cell's associated bipartition parts. The weighting value $w_i \in \{1, 2, 3\}$, is set by assigning the most unbalanced bipartition the weight $w_i = 3$. A bipartition is unbalanced if one of the two parts has a larger number of octree cell than the other.

The amount of unbalance in a bipartition is found with the following formula:

$$u_i = \left| \frac{D_i, 1}{D_i, 1 + D_i, 2} - 0.5 \right|$$

Creating the list of configuration

After each child cell is assigned a priority value, we find the probability of each possible configuration. The probability of a configuration is the sum of priority values for the configuration's non-empty child cells. We sort the list of configurations according to the probability values, and find the configuration we want to encode in the list. The index of the correct configuration is output, which concludes the encoding of the geometry.

6.4 Decoding the geometry

The decoder calculates the cell priorities the same way as the encoder. After the decoder has read the number of non-empty child cells from the encoded file it can sort the relevant configurations by probability, and use the encoded configuration index to find the correct one. The non-empty child-cells can be created after the configuration is retrieved.

6.5 Encoding the connectivity

We encode the connectivity between the child cells, and between the child cells and the neighbors, as a series of operations similar to the vertex split first used in the PM compression algorithm [Hoppe, 1996].

A k-d tree is a binary tree, where k-d is short for k-dimensional. In this case the tree is a three-dimensional binary tree. Unlike the octree, which splits a cell into eight, the k-d tree splits a cell into two equally sized parts. By containing the octree child cells in a k-d cell we do a series of binary splits, and encode the changes in connectivity after each split.

The root of the k-d tree contains all of the non-empty octree child cells, and is connected to the same neighbors as its contained octree cells. When we split a k-d cell we create two new k-d cells, and divide the octree cells into the new k-d cells. As with the root k-d cell, we set the neighbors of the new k-d cells to be the neighbors of the contained octree cells. The cells of the tree are split until each octree cell is contained in its own k-d leaf cell. If the there are n nonempty octree cells we need $n - 1$ k-d cell splits.

There is a change in the connectivity when a k-d cell is split, and we need to encode how it changes. The decoder does not know about the child cells' neighbor

relations, so it has to change the connectivity based on the information output by the encoder. We call the set of cells having a neighbor relation to the cell to be split the cell's neighborhood. Three pieces of information is needed to store the neighborhood's connectivity changes:

1. The pivot cells, which are the cells connected to both new center cells.
2. The non-pivot cells, which are the cells connected to one of the new center cells.
3. The adjecency between the two new center cells.

6.5.1 Encoding the pivot-cells

The pivot-cells can be represented as a bit-sequence, where each bit denotes whether a cell in the neighborhood is a pivot cell or not. We estimate the probability for each cell being a pivot cell, sort the different possible configurations by probability, and encode the index in the list of configurations. This is very similar to the encoding of the non-empty child cell configurations.

As with the non-empty child cell encoding, we find a priority value for each cell, based on its probability to be a pivot cell. The cell is more likely to be a pivot cell if the triangle created by the cell and the two center cells is regular, i.e. the sides are of equal length. We find the regularity r of a triangle with the following formula:

$$r = \frac{\sigma}{2s} = \frac{\sqrt{s(s-a)(s-b)(s-c)}}{2s}$$

σ is the area of the triangle created by the two center cells and the neighborhood cell, and a, b and c are the length of the triangle sides. s is definded as $s = \frac{a+b+c}{2}$.

6.5.2 Encoding the non-pivot-cells

The neighboring cells are divided into segments, where the pivot-cells belong to their own separate segments. The neighbors between two pivot-cells belong to the same segment. If any of the neighbors are connected to more than two other cells in the neighborhood, they get their own segment as well.

The segments are encoded one at a time. The encoder first outputs a bit sequence, where a bit per segment is used to tell if all the cells in the segment are connected to the same center cell.

The next step is to encode which center cell each segment is connected to. If the cells in a segment are connected to different center cells, the segment is processed by treating each cell as a separate segment. The center cell connected to a segment can be predicted by predicting that the segment is connected to the closest of

the two center cells. For segments consisting of more than one cell we find the segment's centroid, and compare its distance to the center cells.

A bit is used to encode whether the prediction is correct or not. By it self, this is not more efficient than using a bit to denote which center cell the segment is connected to. The prediction bit can, however, be efficiently encoded by an arithmetic coder, as the prediction scheme will make one bit value more likely than the other.

6.5.3 Encoding the adjacency

We need to encode whether the two new cells are connected or not. A single bit is output, denoting if the cells are connected.

6.6 Decoding the connectivity

The creation of a k-d tree is done on the decoder as well, and the k-d cells are split in the same way. We know the coordinates of the non-empty child cells, so we know when a k-d split results in two non-empty k-d cells.

Both sides of the coder can find the same segments, since the decoder receives the configuration of pivot-cells before the segments are created.

6.7 Creating the mesh

At any time during the decoding process we can create a mesh from the information contained in the octree. We start by creating the vertices, the centroid of the leaf cells defining their coordinates. From the vertices' representative octree cells we know which vertices they should be connected to by edges. From an intial vertex the first face is be created, connecting the neighboring vertices. After the intial face is created, we expand the mesh one face at a time by adding faces to the border of the mesh. The octree data stucture does not contain any information about the orientation of the faces, so each face must be added adjecent to another face. This way we do not create faces with different orientations, and maintain the orientable property of the mesh. We create faces until each vertex is adjacent to as many faces as its representative cell has neighbors. If the mesh consists of multiple disjoint mesh pieces, the mesh creation method must start anew with each piece.

We first tried another method, based how the mesh creation is described in the source paper by [Peng and Kuo, 2005]. We tried to update the mesh each time a cell is split, by removing the original vertex and adding the new ones, and connecting the vertices to the rest of the mesh.

We had to use our method because of the mesh library we chose, called OpenMesh, which is presented in chapter 7. The original mesh creation does not create

faces with a specific orientation. During the implementation of the mesh creation we found out that OpenMesh does not have to ability to support non-orientable meshes, a fact that is not mentioned in the official OpenMesh documentation.

This original mesh creation method can create areas where two adjacent faces has opposite orientation. If OpenMesh notifies us that the face we are trying the create has opposite orientation to an adjacent face, we can simply re-add the face with the correct orientation. The problem arises when mesh elements are created from two disconnected octree cells, and we end up with two disjoint meshes. Later, when the decoder created a face connecting the two, we may find that the two mesh pieces has opposite orientation.

The required orientable property actually helps us to preserve the surface normals of the original mesh, even if the octree coder does not store the face orientations. It does, however, require the first face to have the correct orientation. If the first face is created upside down the every face with have the wrong orientation. The encoder can get a face's orientation from the original mesh, and by sending the normal of the first face in eacn mesh piece we can maintain the orientation of the mesh surface, but with a small addition to the storage required.

6.8 Summary

In this chapter we described our design of the compression algorithm based on the octree datastructure, created by [Peng and Kuo, 2005]. The algorithm use the octree data structure to contain the information in the mesh, using cell splits to increase the detail of the octree. Each cell represents a vertex at an LOD dependent on the cell's level in the tree. We can create a lossless representation of the original mesh from the leaf cells in a fully expanded octree. The encoder splits a cell based on the vertices contained within, and encodes the status of the new cells, which is used by the decoder to repeat the split operations. The decoder creates the mesh from the leaf cells, and their neighbor relations to other cells. The octree contains no information about the faces or their orientation, as the original algorithm assumes that the faces are not oriented. We created a mesh creation method that works with orientable meshes, as long as the first created face has the correct orientation. The rest of the faces will have an orientation dependent on the first face.

7

Data structure and library

In this chapter we will chose a mesh data structure and library, which we will use in the following implementations.

In section 7.1 we present some of the most popular data structures used to contain the information in a triangular mesh, and explain why we chose the half-edge data structure as the one best suited to our needs. In section 7.2 we explain how the OpenMesh library works, which is a mesh library based on the half-edge data structure.

7.1 Choosing a data structure

A data structure containing the information in a mesh must contain the mesh elements, and the connectivity between them. Several different ways to represent the data exists. The most popular data structures are based on the following different concepts:

- Vertex and face lists
- Face relations
- Winged-edges, bi-directional edges with left and right pointers.
- Half-edges, directed edges with previous and next pointers.

In this section we take a close look at each of these data structures, and conclude with the best data structure suited to our needs. There are some operations we will have to do numerous times when implementing the chosen compression algorithms. It would be beneficial to use a data structure enabling

easy implementations of these operations. Both the octree coder and the valence coder need to find a vertex's incident faces and neighboring vertices, as this is needed each time a vertex is removed from the mesh. The valence coder must traverse the mesh as well, when finding new vertex to remove.

7.1.1 List of vertices and faces

A simple data structure is a list with only the position of the vertices, and the vertices defining each face. This is the same data contained in the uncompressed mesh file formats explained in section 3.5, without any additional data stored. Listing 7.1 describes an implementation of this data structure.

```
1  class Point
2  {
3      float x, y, z;
4  }
5
6  class Vertex
7  {
8      Point *p;
9  }
10
11 class Face
12 {
13     Vertex[] *vertices; //The three vertices defining the face
14 }
```

Listing 7.1: Faces with pointers to vertices

This is a simple data structure, requiring little memory. It can easily be drawn by iterating though a list of all the faces, and accessing each face's vertices. Modifying the mesh is more problematic. Since the edges between the faces and vertices are implicit, there is no easy way to find the incident faces to a vertex, or a vertex's neighboring vertices.

7.1.2 Face-based data structure

A more complex data structure is the face-based data structure, where the face contains pointers to the neighboring faces, as well as the vertices defining the face. Each vertex has also a pointer to an incident face. Listing 7.2 describes an implementation of this data structure.

```
1  class Point
2  {
3      float x, y, z;
4  }
5
6  class Vertex
7  {
8      Point *p;
9      Face *incident_face;
10 }
11
```

```
12 class Face
13 {
14     Face[] *adjacent_faces //The three adjacent faces
15     Vertex[] *vertices ////The three vertices defining the face
16 }
```

Listing 7.2: Faces with pointers to vertices and adjacent faces

There are of course usually several faces incident to a vertex, but only one is needed to gain access to the topology surrounding the vertex. Through the incident face, the rest of the incident faces and neighboring vertices can be reached. The incident faces can be found by starting with the vertex's face pointer, and finding all adjacent faces also connected to the vertex until every incident face is found. The pointers to the adjacent faces make it easy to traverse the faces of the mesh, which is important for the coders based on an expanding borderline.

However, like the list of vertices and face, this data structure does not contain any edge related information. This is not beneficial for operations based on edges like the edge collapse operator.

7.1.3 Winged-edge based data structure

The winged-edge based data structure is based on edges, unlike the two previous data structures we have discussed, and contains a lot of adjacency information. The edges allows for easy access of the faces and vertices adjacent to an edge. Listing 7.3 describes an implementation of this data structure.

```
1  class Point
2  {
3      float x, y, z;
4  }
5
6  class Vertex
7  {
8      Point *p;
9      Wingededge *we;
10 }
11
12 class Face
13 {
14     Wingededge *we;
15 }
16
17 class Wingededge
18 {
19     Wingededge *left_previous, *left_next;
20     Wingededge *right_previous, *right_next;
21     Vertex *from, *to;
22     Face *left, *right;
23 }
```

Listing 7.3: Faces with pointers to vertices and adjacent faces

This data structure contains explicit information about the edges, which makes it easy to traverse the adjacent elements of the mesh. The mesh can be traversed by following the edges' previous and next pointers.

7.1.4 Half-edge based data structure

The half-edge data structure is similar to the winged-edge data structure, but contains oriented edges. While the winged-edge data structure use a single bidirectional edge to express the connection between faces or vertices, the half-edge data structure use two edges to express the same connection, one in each direction. This requires twice the number of edges, but makes some operations less computationally heavy. Listing 7.4 describes an implementation of this data structure.

```
1  class Point
2  {
3      float x, y, z;
4  }
5
6  class Vertex
7  {
8      Point *p;
9      Halfedge *he;
10 }
11
12 class Face
13 {
14     Halfedge *he;
15 }
16
17 class Halfedge
18 {
19     Halfedge *opposite, *previous, *next;
20     Vertex *v;
21     Face *f;
22 }
```

Listing 7.4: Half-edge data structure

The neighboring vertices of a vertex can be found with very few steps compared to the data structures with implicit edges, by starting with the half-edge pointed to by the vertex. The opposite of this half-edge will points back towards the vertex, and the next half-edge will point to the next vertex in the neighborhood. This can be repeated until all the neighbors are found.

7.1.5 Conclusion

We have chosen the half-edge data structure for the implementation of the two compression algorithms. The data structure based on lists of vertices and faces does not provide easy removal of mesh elements. This is an integral part of both compression algorithms, and makes it a bad choice of data structure. The face-based data structure contains more information, but there is no explicit information about the edges.

Both the half-edge and winged-edge data structured would work for our purposes. The winged-edge stores more data per edge, but the half-edge data structured needs

twice the amount of edges. For each winged-edge two half-edges are needed, one for each direction.

The doubled-sided edges in the winged-edge data structures have arbitrary orientations. If we want to get the vertices incident to a face in counter-clockwise order, we have to find out if we need to go forward or backwards from the first edge. The half-edges incident to a face are in counter-clockwise order, enabling us to simply use the next-pointer when traversing the edges surrounding a face.

We have found that the half-edge data structure is best suited to out needs, compared to the other data structures. The data structure both enables easy implementations of our needed operations. Even if it needs more memory than the other data structures, less computational power is needed to traverse the mesh. In the next section we will provide a brief explanation of the OpenMesh library, which is based on the half-edge data structure.

7.2 The OpenMesh library

We have chosen to use the OpenMesh library to represent the triangle meshes in the chosen compression algorithms. OpenMesh is an efficient library for working on polygonal meshes, developed at the Computer Graphics Group, RWTH Aachen [Botsch et al., 2002].

OpenMesh is based on the half-edge data structure, and designed to be flexible, efficient and easy to use. It can handle both polygonal and triangular meshes, and provides more efficient operations to the triangle meshes. The vertices, edges and faces are stored internally in arrays, but these arrays are not directly accessible to the user.

We gain access to the mesh elements with the help of a collection of handles. In listing 7.5 we create a vertex with coordinates $(-1, -1, 1)$, and use the returned vertex handle to retrieve the Point object defining its location. Similarly named handles exists for faces and half-edges.

```
1 //Define MyMesh as a triangle mesh class
2 typedef OpenMesh::TriMesh_ArrayKernelT<> MyMesh;
3 MyMesh mesh;
4 MyMesh::VertexHandle vertex = mesh.add_vertex(MyMesh::Point(-1, -1,  1));
5 MyMesh::Point vertex_point = mesh.point(vertex);
```

Listing 7.5: Create a vertex in OpenMesh, and retrieve its coordinates

When a mesh element is deleted from the mesh is it only marked for deletion, to save processing power. For the elements to be properly deleted a garbage collector function must be run, which removes any elements marked for deletion, and reorganizes the underlying data structure. The garbage collector function should not be run during a traversal of the mesh, as any handle created before the function is run may be invalid.

The library provides many functions making the mesh traversal easier. Iterators allow us to navigate though the vertices, faces or half-edges. Circulators allow us to iterate over the mesh elements adjacent to an element of the same or different type. E.g., we can get the vertices adjacent to a vertex, or the edges incident to a face. Table 7.1 shows a list of possible circulators. We can also use a circulator going in a specified direction. Table 7.2 shows two additional circulators used to specify the direction of the VertexVertexIter. The same can be done with the other circulators as well.

Circulator	Iterating over
VertexVertexIter	A vertex's neighboring vertices
VertexIHalfedgeIter	A vertex's incoming halfedges
VertexOHalfedgeIter	A vertex's outgoing halfedges
VertexFaceIter	A vertex's adjacent faces
FaceVertexIter	A face's vertices
FaceHalfedgeIter	A face's halfedges
FaceFaceIter	A face's edge-neighboring faces

Table 7.1: OpenMesh circulators

Circulator	Direction
VertexVertexIter	Unspecified direction
VertexVertexCwIter	Clockwise direction
VertexVertexCCWIter	Counterclockwise direction

Table 7.2: OpenMesh circulator directions

We can easily add custom properties to the mesh elements. Listing 7.6 shows how we can add a Boolean value called *visited* to a vertex, which can useful if we want to traverse the mesh and keep track of where we have already been. The *property* function accepts a reference to a property, and a handle to a mesh element.

```
1 OpenMesh::VPropHandleT<bool> visited; //Define a vertex property
2 mesh.add_property(visited); //Add property to mesh
3 mesh.property(visited, vertex_handle) = true;
4 bool vertex_is_visited = mesh.property(visited, vertex_handle);
```

Listing 7.6: OpenMesh custom property

OpenMesh only supports orientable manifold meshes. This fact is not documented in the official OpenMesh documentation, but it is something we were made aware of during the implementation of the octree based compression algorithm. This created some problems with the implementation of the octree based compression algorithms, which originally does not store the orientations of the faces. We get an error if we try to add a mesh element that makes the mesh non-manifold or non-orientable, and the creation of the mesh element is aborted, leaving the mesh is the same state as before the attempted element creation.

7.3 Summary

In this chapter we have presented the popular data structures used to contain the mesh data. We chose the half-edge based data structure, as it supports efficient implementations of the mesh traversal operations needed by the algorithms we will implement. It does, however, require more storage space compared to the other representations. For the implementation of the compression algorithms we decided to use the half-edge based OpenMesh library, which is an effective and flexible library.

8

Implementation of the valence coder

8.1 Introduction

In this chapter, we explain our implementation of the valence based compression algorithm. We focus only on the most interesting parts, and explain how we implemented the mesh traversal, the encoder's vertex removal and the decoder's vertex insertion.

We present our implementation of the general conquest in section 8.2. As we explained in section 5.5, the cleaning conquest is very similar to the general conquest, and we will therefore not describe our implementation of the cleaning conquest.

8.2 Conquest

The integral part of the conquest is the gate system. We use a First-In-First-Out queue to keep track of the half-edges we designated to be gates. The conquest must have a starting patch, which means we must find an initial gate, described in section 8.2.1.

Before we remove a vertex we must chech if the patch is concave, and we explain how we do this in section 8.2.2. Section 8.2.3 details our implementation of the vertex removal, while section 8.2.4 details our vertex insertion implementation.

8.2.1 Finding the initial gate

The deterministic property of the conquest requires the encoder and decoder to always find the same initial gate. We find this gate by finding the vertex with the lowest index, which is the first vertex returned by OpenMesh's vertex iterator. The next step is to find this vertex's neighbor with the lowest index. The initial gate is set to be the half-edge going from the first to the second vertex. This is a simple solution, and we are guaranteed to find the same half-edge on the encoder and decoder, as we know that this edge will remain throughout the conquest. This edge will be on the borderline on a patch, and consequently not removed, since the set of removed vertices is independent.

8.2.2 Detecting a concave patch

Before we attempt a vertex removal, we must check if the patch is concave. We do this by checking if any of the angles in the patch are more than 180 degrees. Listing 8.1 describes how we check the angle between two adjacent edges on the patch borderline.

```
MyMesh::Point normal_1 = mesh.normal(patch_faces[i]);
MyMesh::Point normal_2 = mesh.normal(patch_faces[i+1]);
MyMesh::Point p1 = mesh.point(neighborhood[i]);
MyMesh::Point p2 = mesh.point(neighborhood[i+1]);
MyMesh::Point p3 = mesh.point(neighborhood[i+2]);
MyMesh::Point vec1 = p2-p1;
MyMesh::Point vec2 = p3-p2;

//Find dot product
float dot = vec1[0]*vec2[0] + vec1[1]*vec2[1] + vec1[2]*vec2[2];

//Find determinant
float det_1 = vec1[0] * vec2[1] * normal_1[2] +
              vec1[2] * vec2[0] * normal_1[1] +
              vec1[1] * vec2[2] * normal_1[0] -
              vec1[2] * vec2[1] * normal_1[0] -
              vec1[0] * vec2[2] * normal_1[1] -
              vec1[1] * vec2[0] * normal_1[2];
float angle_1 = atan2(det_1, dot);

float det_2 = vec1[0] * vec2[1] * normal_2[2] +
              vec1[2] * vec2[0] * normal_2[1] +
              vec1[1] * vec2[2] * normal_2[0] -
              vec1[2] * vec2[1] * normal_2[0] -
              vec1[0] * vec2[2] * normal_2[1] -
              vec1[1] * vec2[0] * normal_2[2];
float angle_2 = atan2(det_2, dot);

//Value 0.0 equals 180 degrees
if(angle_1 <  0.005 || angle_2 <  0.005) {
    is_concave = true;
}
```

Listing 8.1: Checking if an angle makes the patch concave. The patch_faces and neighborhood arrays contain the patch's faces and vertices in counterclockwise order.

When we find an angle between two edges in 3D, we must provide a reference viewpoint. We use the two patch faces adjacent to the edges, and check the angle from directly above the faces. It was not enough to use only one of the faces as a reference. We needed to check if the patch looks concave from above either of the two faces, in case one face had a very different normal vector compared to the other.

Two edges making a straight line creates problems during the retriangulations, as a new edge will be created atop of it. Therefore, we cannot allow angles very close to 180 degrees, and use a limit slightly larger than zero.

8.2.3 Vertex removal

Emptying patch

We need to remove the front vertex and its surrounding faces before the retriangulation can begin. We do not need to delete the edges between the front vertex and the vertices on the patch borderline, as these are automatically deleted by OpenMesh. With the front vertex gone, the faces would later be deleted by OpenMesh's garbage collector function, but we need to delete them manually to be able to retriangulate the hole with new faces.

Retriangulating the patch

The next step is to fill the empty patch, according to figure 5.5. The information in the retriangulation tables can be stored in two series of arrays. The first array type specifies the retriangulation flags for the vertices in the neighborhood and the second array contains the information needed to create the new faces.

```
1 #define TRISUB 0
2 #define TRIADD 1
3 #define TRIINIT 2
4 int retri_flag_pent[2][2][3] = {
5     {{TRIADD, TRISUB, TRIADD}, //--
6      {TRIADD, TRISUB, TRIADD}}, //-+
7     {{TRIADD, TRISUB, TRIADD}, //+-
8      {TRISUB, TRIADD, TRISUB}}}; //++
```

Listing 8.2: Retriangulation flags for patch with degree five

Listing 8.2 shows one of the four arrays specifying the retriangulation flags. One array is needed for each of the possible patch degrees, from three to six. The two first dimensions in the array specify the relevant configuration, which depends on the retriangulation flags of the gate's two vertices. The first index is the flag of the vertex the gate is originating from, where index zero is the - flag, and index one is the + flag. The second index is the flag of the vertex the gate is pointing to. The three inner values are the flags the vertices in the neighborhood will get,

in counterclockwise order. There are only three flags, and not five, since the gate's vertices already have their retriangulation flags set.

```
//Contains indices of vertices starting with the left vertex in the gate
    as 0.
int retri_conf_pent[2][2][3][3] = {
    {{{0, 1, 2}, {0, 2, 4}, {2, 3, 4}}, //--
     {{0, 1, 4}, {1, 2, 4}, {2, 3, 4}}}, //-+
    {{{0, 1, 2}, {0, 2, 4}, {2, 3, 4}}, //+-
     {{0, 1, 3}, {1, 2, 3}, {0, 3, 4}}}}; //++
```

Listing 8.3: Retriangulation configurations for patch with degree five

Listing 8.3 shows one of three arrays specifying the configuration of the new faces defining the retriangulated patch. There is one less array compared to the retriangulation flag arrays, as a patch with degree three has only one possible configuration. The two first dimensions are the retriangulation flags deciding the configuration, like for the array in listing 8.2. The third dimension decides which triangle in the retriangulated patch to define. The innermost array dimension denotes the vertices defining a face, where each value is a vertex's index in the neighborhood, going counterclockwise from the gate's starting vertex.

```
if(patch_degree == 5) {
    for(i = 0, j = 0; i < 3; i++) {
        std::vector<MyMesh::VertexHandle> face_vhandles;
        face_vhandles.push_back(
            patch_vertices[retri_conf_pent[from_flag][to_flag][i][0]]);
        face_vhandles.push_back(
            patch_vertices[retri_conf_pent[from_flag][to_flag][i][1]]);
        face_vhandles.push_back(
            patch_vertices[retri_conf_pent[from_flag][to_flag][i][2]]);
        MyMesh::FaceHandle new_face = mesh.add_face(face_vhandles);

        if(new_face != MyMesh::PolyConnectivity::InvalidFaceHandle) {
            mesh.property(f_state_flag, new_face) = CONQUERED;
            added_faces.push_back(new_face);
        }
        else { //Encoder cannot handle topology
            for(i = i-1; i >= 0; i--) {
                mesh.delete_face(added_faces[i]);
            }
            reverse_retriangulation = true; //Skip borderline traversal
            break;
        }
    }
}
```

Listing 8.4: Creating faces to fill a patch with degree five. The implementation is similar for the patches of other degrees. patch_vertices is an array containing the patch vertices in counterclockwise order

We can see how the configuration tables are used in listing 8.4. The retri_conf_pent array makes it possible to create the different face configurations without any duplicate code or conditional statements dependent on the retriangulation flags. The OpenMesh function add_face returns an error value if the face creation results in an invalid mesh, which means we must abort the vertex insertion. If some faces have already been added in this patch retriangulation, we remove them and tell the

encoder to not traverse the patch borderline.

If the reverse_retriangulation variable is set to true, we create a new vertex with the same position as the original vertex. We recreate the original faces by iterating through the neighbors array and use the adjacent neighbor pairs and the new vertex to create each face.

Traversing the patch borderline

```
1  //Set neighborhood state flags
2  for(i = 0; i < patch_vertices.size()-1; i++) {
3      mesh.property(v_state_flag, patch_vertices[i]) = CONQUERED;
4  }
5  //Set retriangulation flags, and find new gates.
6  for(i = 0; i < patch_vertices.size()-1; i++) {
7      if(mesh.property(v_retri_flag, patch_vertices[i]) != TRIINIT) {
8          //Vertex retriangulation has already been set, do nothing.
9      } else if (patch_degree == 3) {
10         mesh.property(v_retri_flag, patch_vertices[i]) =
11             retri_flag_tri[from_flag][to_flag];
12     } else if (patch_degree == 4) {
13         mesh.property(v_retri_flag, patch_vertices[i]) =
14             retri_flag_quad[from_flag][to_flag][i-1];
15     } else if (patch_degree == 5) {
16         mesh.property(v_retri_flag, patch_vertices[i]) =
17             retri_flag_pent[from_flag][to_flag][i-1];
18     } else if (patch_degree == 6) {
19         mesh.property(v_retri_flag, patch_vertices[i]) =
20             retri_flag_hex[from_flag][to_flag][i-1];
21     }
22     // Find half-edge pointing to next vertex in patch_vertices
23     MyMesh::VertexOHalfedgeIter voh_it;
24     for (voh_it=mesh.voh_iter(patch_vertices[i]); voh_it.is_valid(); ++
       voh_it) {
25         if(mesh.to_vertex_handle(*voh_it) == patch_vertices[i+1]) {
26             MyMesh::HalfedgeHandle next_gate;
27             next_gate = mesh.opposite_halfedge_handle(*voh_it);
28             gate_queue.push_back(next_gate);
29             break;
30         }
31     }
32 }
```

Listing 8.5: Retriangulation configurations for patch with degree five

Listing 8.5 shows how we traverse the neighborhood and set the vertices' retriangulation flags, by using the retri_flag tables. If any vertex has a previously defined retriangulation flag, like the gate's vertices and any vertex belonging to another patch, we ignore them.

Since the vertices in patch_vertices are ordered, we can add the new gates by finding the edge connecting a vertex with the next vertex. The edges surrounding a face in a half-edge data structure has a counterclockwise direction. This means that the edge going from a vertex to the next vertex in the neighborhood, which is also stored counterclockwise, is an edge on the inner side of the patch. The opposite edge will be outside of the patch, and is a new gate.

8.2.4 Vertex insertion

We insert a vertex into the mesh when the decoder retrieves a valence number from the encoded data.

Discovering the patch

```
//Vertex indices in patch_vertices, used to
int repatch_conf_pent[2][2][3][2] = {{{{0,1}, {0, 2}, {2, 3}}, //--
                            {{0, 1}, {1, 2}, {2, 3}}}, //-+
                            {{{0,1}, {0, 2}, {2, 3}}, //+-
                            {{0, 1}, {0, 2}, {1, 2}}}}; //++
```

```
MyMesh::FaceHandle patch_faces[valence_code-2];
for(i = 0; i < valence_code-2; i++) {
    if (valence_code == 5) {
        from_vertex = patch_vertices[repatch_conf_pent[from_flag][to_flag
    ][i][0]];
        to_vertex = patch_vertices[repatch_conf_pent[from_flag][to_flag][i
    ][1]];
    }
    MyMesh::HalfedgeHandle gate;
    MyMesh::VertexOHalfedgeIter voh_it;
    for (voh_it=mesh.voh_iter(from_vertex); voh_it.is_valid(); ++voh_it) {
        if(mesh.to_vertex_handle(*voh_it) == to_vertex) {
            gate = *voh_it;

            int face_flag = mesh.property(f_state_flag,
                                          mesh.face_handle(gate));
            if(face_flag == REMOVABLE) {
                gate = mesh.opposite_halfedge_handle(gate);
            }
            break;
        }
    }
    MyMesh::FaceHandle face = mesh.face_handle(gate);
    patch_faces[i] = face;
    mesh.property(f_state_flag, face) = REMOVABLE;

    //Set next vertex to be processed in for loop
    patch_vertices[i+2] = mesh.opposite_vh(gate);
}
```

Listing 8.6: Discovering the patch faces. Note that the code for the patches with a different degree than five has been removed

Before we can insert a vertex, we must discover the retriangulated patch to replace. For us to discover the retriangulated patch, we need to traverse the faces defining it. We start with the face adjacent to the gate, and find one face at a time. Two of the vertices defining the first face will also be a part of another face in the patch. We use the patch_vertices array to store the vertices defining the patch borderline, as shown in listing 8.6. It starts with the gate's vertices, and is filled as the patch is discovered. The repatch_conf_pent array in listing 8.2.4 gives us the indices of two vertices both belonging to an existing face, and a new undiscovered face.

We use an additional face state flag to keep track of the discovered faces. We do not know if the edge belongs to the new triangle, or to a previously discovered face.

Recreating the original patch

```
//Contains the indices used to get vertices in patch_vertices in
//insert_general_vertex() in counter clockwise order.
int ccw_vertices_pent[2][2][5] = {{{0, 1, 2, 4, 3}, //--
                                   {0, 1, 3, 4, 2}}, //-+
                                   {{0, 1, 2, 4, 3}, //+-
                                   {0, 1, 4, 2, 3}}}; //++
```

Listing 8.7: The counterclockwise order of the discovered vertices.

The patch discovery leaves us with an unsorted array of vertices, but for the recreation of the original faces we need the vertices sorted in counterclockwise order. The array in listing 8.7 contains the indices of the vertices in patch_vertices in correct order. This allows the patch_vertices array to be traversed as if it was ordered, by substituting the patch_vertices index with the value in ccw_vertices_pent. This is shown in listing 8.8

```
for(i = 0; i < valence_code; i++) {
    if (valence_code == 5) {
        from_vertex = patch_vertices[ccw_vertices_pent[from_flag][to_flag
    ][i]];
        if(i == valence_code-1) {
            to_vertex = patch_vertices[ccw_vertices_pent[from_flag][
    to_flag][0]];
        }
        else {
            to_vertex = patch_vertices[ccw_vertices_pent[from_flag][
    to_flag][i+1]];
        }
    }
}
```

Listing 8.8: Finding the vertices used to create each original face

8.3 Summary

In this chapter we explained how we implemented the general conquest, on both the encoder and decoder. We showed how we check if a patch is concave, by calculating the angles of the edges in the patch borderline. We showed how we use a set of configuration arrays to contain the retriangulation information, which can effectively give the information needed to achieve the deterministic traversal of the mesh. Similar tables are used by the vertex removal and vertex insertion operations as well, so that the patches retriangulated by the encoder can be found by the decoder.

9 Implementation of the octree coder

9.1 Introduction

We implemented this encoder after the valence coder, and we decided to continue using the OpenMesh library. We were satisfied with its easy of use, and wanted to take advantage of the opportunity to use a familiar library. This however, created some problems during the implementation of the mesh creation, which we discuss in section 9.6.

We use a class called OTCell to represent the cells in the octree. Listing 9.1 shows the OTCell's constructor. We send the number of quantization bits with each new octree cell so that it can compare it with the octree level, and know when to stop dividing cells. The parent_id is used to create a unique identifier for each cell, dependent on the cell's position in the tree. An OTCell object contains a set of neighboring Cells, and a collection of the vertices contained within the cell's bounding box, alongside some values used to encode and decode the cell divisions.

The KdCell class, its constructor shown in listing 9.2, is used to represent the k-d cells. It contains a set of neighbors like the OTCell class, and a list of contained OTCells.

```
1 OTCell(int bounding_box[], //The cell's bounding box
2        int quantization_bits, //The number of quantization bits
3        int position, //The position in the parent's list of child cells
4        int OTlevel, //The tree level the cell is located in.
5        std::string parent_id //Identifier, for debugging
6        )
```

Listing 9.1: OTCell constructor

```
1 KdCell(std::vector<int> cell_bb, //The cell's bounding box
```

```
2         std::string identifier //Identifier, for debugging
3         )
```

Listing 9.2: KdCell constructor

The two classes are subclasses of the same superclass Cell. We use a set to contain a cell's neighbors, and during the connectivity encoding the elements in a neighbor set can be both octree cells and k-d cells. Therefore we use the Cell superclass to refer to both types of cells.

In this chapter we explain our implementation of the octree coder, with a focus on the encoder. The decoder is very similar to the encoder. It does the same division operations of the octree cells and k-d cells, the difference lies in the way the information used to divide the cells is found. The encoder finds the information from the input mesh, and the decoder gets the information from the encoded data.

In section 9.3, we explain how an octree cell is divided by the encoder. The encoder encodes the resulting geometry and connectivity changes. The encoding of the child cells' locations is explained in section 9.4. In section 9.5, we demonstrate how we encode the connectivity information, which is the neighbor relations of the child cells. Lastly, in section 9.6, we explain how the decoder creates a mesh from the octree data structure.

9.2 Queue of cells

The queue of cells is implemented by using a priority queue with the cell with the highest importance value first. After a new cell is created and assigned neighbors, its importance is calculated and it is added to the queue. In listing 9.3 we show how we generate a cell's importance value. We use the length along one axis as the size of the bounding box. It represents its size well since the bounding box is a cube, and it does not unnecessarily dominate the importance value, like e.g. the volume would do. Nevertheless, it is still usually the largest value of the three, when the mesh requires a high number of quantization bits. The average distance between the neighbors will be low if the mesh is fairly detailed, and a cell's number of neighbors will be rather small as long as the largest cells are divided first.

```
1  float distance_sum = 1;
2  std::set<Cell*>::iterator it;
3  for (it = neighbors.begin(); it != neighbors.end(); ++it) {
4      //Add the distance between the centroids of the two cells
5      distance_sum += point_distance(centroid, (*it)->get_centroid());
6  }
7  float avg_distance = distance_sum/neighbors.size();
8
9  //Get bounding box size
10 int bb_size = abs(cell_bb[3]-cell_bb[0]);
11
12 importance = avg_distance*bb_size*neighbors.size();
```

Listing 9.3: Calculating a cell's importance value

9.3 Octree cell division

When we divide an octree cell we need to create the new cells, and partition the parent cell's bounding box. On the encoder side we must assign the vertices to octree cells dependent on the partitioning of the bounding box, and we must set the neighbor relations of the new child cells.

9.3.1 Partitioning

Two new bounding boxes will be created along each axis, splitting the length along each axis in two. We find the coordinates of the new bounding boxes by finding the midpoints of the sides of the current bounding box.

We create a new octree cell for each new bounding box, and create an array of new octree child cells. The array is ordered based on the position of the child cells' bounding boxes in the parent's partitioned bounding box. We use one byte to store the cell configuration, where each bit denotes if the corresponding cell in the ordered array of child cells exists. In section 9.4 we explain how we encode this value, by using its index in a list of configurations sorted by probability.

9.3.2 Vertex-cell assignment

The parent cell's contained vertices must be assigned to the child cells, and each vertex must have a pointer to its cell as well. This pointer is needed when we set the neighbor relations of the new cells. Since we store the child cells in an array ordered by location we can easily find the associated child cell for each vertex, as we can see in listing 9.4.

```
1  int axis_index[3] = {0,0,0};
2  if(vertex_point[0] > x_midpoint) {
3      axis_index[0] = 1;
4  }
5  if(vertex_point[1] > y_midpoint) {
6      axis_index[1] = 1;
7  }
8  if(vertex_point[2] > z_midpoint) {
9      axis_index[2] = 1;
10 }
11 int child_index = (axis_index[0]*4+axis_index[1]*2+axis_index[2]);
12 children[child_index]->add_vertex(vertices[i]);
13 mesh.property(v_cell, vertices[i]) = children[child_index];
```

Listing 9.4: Assigning vertices to cells, and vice versa

The vertex-cell assignment must be done for every vertex contained in the parent cell. After the vertices have been assigned to the octree cells, the cells without any vertices can be removed.

9.3.3 Setting the neighbor relations

After a cell is partitioned and the vertices are assigned to the child cells we must set the child cells' neighbor relations. The neighbor relations of the parent cell are removed, so that only the leaf cells are visualized when a mesh is generated from the tree.

We find out if a vertex is adjacent to a vertex in another cell by circulating its adjacent vertices. If any adjacent vertex belongs to another cell, we add the cells to each others neighbor sets. Storing a cell's neighbors in a set enables us to add already existing neighbor relations without any duplication problems.

We cannot do this in the same loop used to go the vertex-cell assignment, as every vertex must have the cell pointer correctly stored. Therefore, we must iterate over the parent cell's vertices twice during the partition phase.

In addition to the vertex-cell assignment, finding the neighbor relations is a computationally heavy operation. During the creation of the tree each vertex is accessed several times. Combined, the set of vertices in the mesh is iterated over once per tree level, and for each vertex all of its neighbors are accessed as well.

9.4 Geometry encoding

We need to encode the configuration of non-empty child cells for the decoder to be able to recreate the octree child cells. We must first find the priority values of the cells, and then sort the possible configurations by priority.

9.4.1 Calculating cell priority

The cell priority values are calculated from the distances to the cells in the child cells' three bipartition parts. We use a 2D vector to organize the cells belonging to each bipartition part. For each bipartition we add the cells to a part dependent on the cells' location along the relevant axis. When we have an overview over the cells in each bipartition parts, we can find the distance value belonging to each part.

```
1  int bp_distance[3][2]; //bipartition part's distances values
2  for(i = 0; i < 3; i++) {
3      for(j = 0; j < 2; j++) {
4          int d = 0; //Weighted distance to cells in bipartition part
5          //For cells in bipartition part:
6          for(k = 0; k < bp[i][j].size(); k++) {
7              //Distance along the i-th axis
8              int distance = abs(centroid[i] - bp[i][j][k]->get_centroid()[i
       ]);
9              d += bp[i][j][k]->get_OTlevel() * distance;
10         }
11         bp_distance[i][j] = d;
12     }
13 }
```

Listing 9.5: Finding the distance values

In listing 9.5, we find the distance value for each of the six bipartition parts. The bp array contains the cells belonging to each bipartitioning part, where i is the axis index, j is the part index, and k is the index of the cell in the part.

The priority value of a cell is found by multiplying each relevant distance with a weighting value, and adding the products together. For each bipartition we find the unbalance, calculated from the values in the relevant row in array bp_distance. An array of weights is assigned the weighting values, from one to three, dependent on the size of the relevant bipartition's unbalance value. We see how we used the array of distance values and weights to find the priority value for each cell in listing 9.6. The resulting array is ordered the same way as the array of child cells.

```
1 cell_priority.clear();
2 for(i = 0; i < 8; i++) {
3     int priority = 0;
4     priority += bp_distance[0][i/4] * weights[0];
5     priority += bp_distance[1][(i/2) % 2] * weights[1];
6     priority += bp_distance[2][i % 2] * weights[2];
7     cell_priority.push_back(priority);
8 }
```

Listing 9.6: Calculating the cell priorities

9.4.2 Creating the configuration tables

The probability of a child cell configuration is calculated by adding the priority values for each non-empty cell. We store a configuration as an 8-bit integer. We create the list of possible configurations by finding every possible 8-bit permutation with the same amount of ones as our configuration. The list of configurations is sorted based on the configurations' priority value. Lastly, we iterate through the list until we find our configuration, and encode its index.

9.5 Connectivity encoding

We encode the connectivity as a series of binary splits, with the help of a k-d tree. The first step after splitting a k-d cell is to set the two new k-d cells' neighbors according to their contained cells. The layout of the updated neighborhood must be encoded, for the decoder to replicate this division. The neighborhood around a cell is circular, without a defined start or end point. Therefore we must find a deterministic ordering of the neighborhood, so that we can use the indices of the cells to encode the connectivity. Without a deterministic ordering a neighborhood index may refer to different vertices on the encoder and decoder side. The ordering is explained in the first subsection, followed by explanations of how we encode the pivot and non-pivot cells.

We remove a k-d cell's neighbor relations after it has been split in two, like we do when we split an octree cell. After all the k-d splits are done, we remove the neighbor relations to the k-d cells, and then remove the k-d cells. The k-d cells are only used to find the vertex split information, and will not be stored in the tree representing the mesh. The decoder must transfer the neighbor relations from the k-d cells to the representative octree cells, as the k-d cells will contain the connectivity information obtained from the encoded data.

9.5.1 A deterministic ordering of the neighborhood

A simple solution for a deterministic ordering of the neighborhood would be to find a starting cell based on location, like the cell nearest origin. We would choose a starting neighbor, and find the next neighbors until we end up with the starting cell again. This works well for most neighborhoods, but gets problematic in some cases. If a cell is connected to more than two other neighbors we get two possible outgoing paths from the cell. This does not happen often in a manifold mesh, but the intermediate k-d splits may create temporary non-manifold neighborhoods.

Out solution is to use a set to keep track of the added cells, and use a deterministic traversal of the neighborhood. The first cell to be processed is the cell nearest origin. When we process a cell we add it to the vector of neighbors, and add it to the set of added cells. We recursively process its neighboring cells also connected to the center cell, ordered by their distance to the origin. We sort them by the distance along one axis at a time, so that no two cells are sorted on the same value. If a cell is already in the set of added cells we ignore it, and do not process its neighbors. This enables both the encoder and decoder to end up with the same neighborhood vector. In listing 9.7, we see the function used to traverse the neighborhood, where cmp_cell_pos is our custom distance comparator.

```
1  void KdCell::get_next_neighbor(std::vector<Cell*>& v_neighborhood,
2                          std::set<Cell*>& cells_added,
3                          Cell* c) {
4      //Get neighbors connected to both c and the center cell
5      std::vector<Cell*> adjecent_n;
6      std::set<Cell*>::iterator it;
7      for (it = neighbors.begin(); it != neighbors.end(); ++it) {
8          if(c->has_neighbor(*it)) {
9              adjecent_n.push_back(*it);
10         }
11     }
12     //Sort them by position
13     sort(adjecent_n.begin(), adjecent_n.end(), cmp_cell_pos);
14
15     //Process all unvisited cells in adjecent_n.
16     for(i = 0; i < adjecent_n.size(); i++) {
17         if(!cells_added.count(adjecent_n[i])) {
18             v_neighborhood.push_back(adjecent_n[i]);
19             cells_added.insert(adjecent_n[i]);
20             get_next_neighbor(v_neighborhood, cells_added, adjecent_n[i]);
21         }
22     }
23 }
```

Listing 9.7: Recursive code finding the next neighbor

Since the intermediate k-d splits can result in non-manifold geometry, we can end up with a neighborhood consisting of multiple disjoint neighborhood parts. This means that some cells will be unreachable from the starting cell. We solve this problem by iterating though the neighborhood, and find any unvisited cells. We sort them by location, and use each unvisited cell as the starting point for a new neighborhood discovery. The additional parts of the neighborhood are added to the end of the neighborhood vector.

9.5.2 Encoding the pivot cells

We can easily find the configuration of pivot cells, as we store the neighbor relations as sets. The pivot cells are the cells contained in both k-d cells neighbor sets, found by a set intersection. We find the configuration of pivot vertices by iterating through the neighborhood vector, and use a bit to store if a cell is a pivot vertex or not.

We find a priority value for each cell by applying the formula from section 6.5. The sorting of the possible configurations is done the same way as the sorting of the non-empty child cell configurations.

9.5.3 Encoding the non-pivot cells

To encode the non-pivot cells we must divide the neighborhood into segments. For an easier generation of segments we rotate the neighborhood vector so that the first cell is a pivot cell, as cells in the beginning of the neighborhood vector may be part of the same segment as the cell at the end of the vector. If there are no pivot cells we do nothing, as each cell in a neighborhood without any pivot cells will belong to the same segment. This can also be done by the decoder, since we decode the configuration of pivot cells before we decode the non-pivot cells.

The code in listing 9.8 describes how we add the cells into segments. The result is a vector of vectors, the vectors containing the segment's cells. We ignore the pivot vertices, since we have already encoded their connectivity information.

```
1  std::vector<std::vector<Cell*> > segments;
2  int last_piv = -1;
3  for(i = 0; i < v_neighborhood.size(); i++)  {
4      if(pivot_config & (1 << i)) {//Pivot segment, is ignored
5          last_piv = i;
6      }
7      else {
8          if(segments.size() == 0 || i == last_piv+1) {
9              segments.push_back(std::vector<Cell*>());
10         }
11         segments.back().push_back(v_neighborhood[i]);
12     }
```

```
13 }
```

Listing 9.8: Segment creation, used by both encoder and decoder

We use two bit sequences to encode the segment information, one to encode whether the segments are connected to the same cell or not, and one to encode is the prediction is correct. In listing 9.9, we compare the distances between the centroid of the segment and the center vertices. We predict that the segment is connected to the closest center cell, and use a bit to denote whether the prediction is correct or not.

```
1  MyMesh::Point sc = segment_centroid(segments[i]);
2  float d0 = point_distance(sc, c0);
3  float d1 = point_distance(sc, c1);
4  if((d0 > d1 && neighbors_0.count(segments[i][0])) ||
5     (d0 >= d1 && !neighbors_0.count(segments[i][0]))) {
6      //Prediction is correct
7      connected_to = (1 << encoding_index) | connected_to;
8  }
9  else {
10     //Prediction is false
11     connected_to = (0 << encoding_index) | connected_to;
12 }
13 encoding_index++;
```

Listing 9.9: Encoding a segment

9.6 Creating the mesh

The octree is only an intermediate data structure used to represent the different LODs of the mesh, and we have to create a mesh to visualize the information contained in the tree.

The first step is to create the vertices. Then we start with a vertex, draw its adjacent faces, and expand the mesh a vertex at a time until the entire mesh is created. If the creation of a face returns an error from OpenMesh we add it again with the opposite orientation. Creating the faces without ending up with non-manifold geometry require us to only add faces adjacent to at least one other face. When a vertex is processed we add the next vertices to be processed to a vertex queue. We add the unvisited vertices belonging to the neighbors of the associated cell.

We describe how we create the vertices in section 9.6.1, and we describe how we process a vertex and create its adjacent faces in section 9.6.2.

9.6.1 Creating the vertices

The vertices are created by finding all the leaf cells, and creating each cell's representative vertex. We start with the root cell, and create the vertices for any leaf child cells. We add the cells as properties to the new representative vertices. We need this when we want to know how to connect the vertices with faces and

edges. We recursively continue the search for leaf cells within any of the non-leaf child cells.

```
void set_tree_vertices(OTCell* otcell) {
    OTCell** children = otcell->get_children();
    MyMesh::VertexHandle vertices[8];
    int i;
    for(i = 0; i < 8; i++) {
        if(children[i] != NULL && children[i]->is_leaf()) {
            vertices[i] = mesh.add_vertex(children[i]->get_centroid());
            children[i]->set_representative_vertex(vertices[i]);

            mesh.property(v_cell, vertices[i]) = children[i];
            mesh.property(v_visited, vertices[i]) = false;
        }
        else if(children[i] != NULL) {
            set_tree_vertices(children[i]);
        }
    }
}
```

Listing 9.10: Creating the vertices

9.6.2 Processing a vertex

When we process a vertex we find an adjacent face and use it as a starting point, since each new face must be added adjacent to another face

Finding the vertex's initial adjacent face

If the vertex is adjacent to a face, we can use it as the initial face. If the vertex is not adjacent to any faces, we must create an initial face, which must be adjacent to an existing face.

For each of the vertices in the list of vertices belonging to the associated cell's neighbors we try to find an edge going to another vertex in the list. If this edge is found, we know that it is adjacent to a face, since the half-edges are created together with the faces in OpenMesh. Then we can use these two vertices to create the vertex's initial adjacent face. In listing 9.10 we show how we create the first face, if the vertex has no adjacent faces. The create_face function tries to add to add the face with both orientations, to maintain the manifold and orientable property of the mesh.

```
std::set<MyMesh::VertexHandle>::iterator it;
for (it = neighbor_vertices.begin(); it != neighbor_vertices.end(); ++it){
    MyMesh::VertexOHalfedgeIter voh_it;
    for (voh_it = mesh.voh_iter(*it); voh_it.is_valid(); ++voh_it) {
        if(neighbor_vertices.count(mesh.to_vertex_handle(*voh_it))) {
            first_face = create_face(vertex,
                                     mesh.from_vertex_handle(*voh_it),
                                     mesh.to_vertex_handle(*voh_it));
        }
    }
}
```

Listing 9.11: Creating the first face. neighbor_vertices is the vertices belonging to the cells neighboring the assosiated cell

Creating the patch faces

We create the faces adjacent to the vertex by starting with an edge incident to the initial face, going from the center vertex to a vertex on the borderline. We find another vertex which has neighbor relations to both vertices, and is not the third vertex of the initial face, as we do not want to recreate an existing face. The three vertices define a face adjacent to at least one other face. If the edge between the two vertices has a face on both sides, this face already exists, and we find the next edge by circulating around the center vertex. We end the circulation after the number of adjacent faces is the same as the number of vertices in the list of neighboring vertices.

9.7 Summary

In this chapter, we explained how we implemented the octree cell division, and how the geometry and connectivity information is encoded, so that the decoder can do the same operations. We explained how the possible non-empty child cells can be ordered by connectivity, and how a configuration can be encoded by storing its index in the list of configurations. The connectivity information is encoded by with a series of k-d splits. The encoder and decoder must have a deterministic ordering of the neighbors to a cell, which we found by traversing the cells based on their distance to the origin. We explained how we encoded the information in the neighborhood, by encoding which cells are connected to both new center cells, and which are only connected to one. The decoder must generate a mesh from the octree. We invented a mesh created method that creates a mesh from the tree while also making the mesh orientable, which is required by the OpenMesh library.

10 Evaluation and discussion

10.1 Introduction

In this chapter we evaluate the results of our implementations of the compression algorithms we chose.

The source papers do not directly assess the visual quality or the computational complexity of the algorithms, which can be just as important as the compression rate when transmitting a 3D model over the internet.

When displaying media to a viewer the visual quality is important, and the progressive compression algorithms try to create visually pleasing intermediate LODs.

10.2 Execution times

In table 10.2 we see the time used to encode and decode figure 10.1 with the LOD shown in figure 10.2. The hardware used is listed in table 10.1. The execution time for the valence encoder is the time it takes to do six full conquests of the mesh. The time for the decoder is the time it takes to apply the six layers of improvements to the base mesh with 800 vertices. The valence coder and octree coder works very differently, so a direct comparison of the execution times is difficult, but we get a good estimate of the differences. The valence coder requires a single-rate encoder to encode the base mesh, while the octree coder can start the improvement from a single vertex. Since we want to compare the two algorithms, we found the time used by the octree encoder and decoder to expand the tree from 800 leaf cells to the

final LOD. The times are calculated by subtracting the time spent expanding the tree up to 800 leaf cells from the total time used to encode and decode the entire mesh.

Motherboard	Notebook W35xSS_370SS
CPU	i7-4710MQ, 2.50GHz
RAM	8 GB, 1600 MHz

Table 10.1: The hardware used to test the implementations.

Algorithm	Encoder	Decoder
Valence coder	2.7	0.19
Octree coder	1.49	1.46

Table 10.2: Execution times in seconds required to encode and decode figure 10.2.

The execution times of the octree encoder and decoder are very similar. Most of the expensive operations we discussed in chapter 9 were done by both the encoder and decoder, except the neighbor finding operation done by the encoder when a new cell is created. Unlike the octree coder the execution times of the valence encoder and decoder differs a lot. The encoder uses a lot of processing power to check if each patch can be decimated without error, which the decoder do not have to do.

The execution times gives us an indication of the differences in our implementations' effectiveness, but it is not a very relevant criteria for the coders' eligibility to be chosen as the best algorithm to transmit a mesh over the internet. If the original mesh will be encoded and transmitted in real-time the encoded data must be sent as soon as it is created by the encoder. In this case the octree encoder will always be more effective than the valence encoder, as the octree encoder can immediately start sending the information needed to split the first cells. The valence encoder needs to encode the entire mesh before sending the base mesh and the layers of improvements, since the encoder generated the highest LODs first.

10.3 Visual quality

In this section we discuss the differences in the visual quality of the meshes created by the two compression algorithms. We compress the figure shown in 10.1, which has 2904 vertices.

10.3.1 Visual quality of the intermediate LODs

Figure 10.2 shows the cow model with a LOD containing 800 vertices, as created by the valence coder and the octree coder. The mesh in figure 10.2(a) was created by stopping the octree encoder after the octree contained over 800 leaf cells, and decoding the resulting data. The mesh in figure 10.2(b) is the result of the encoder

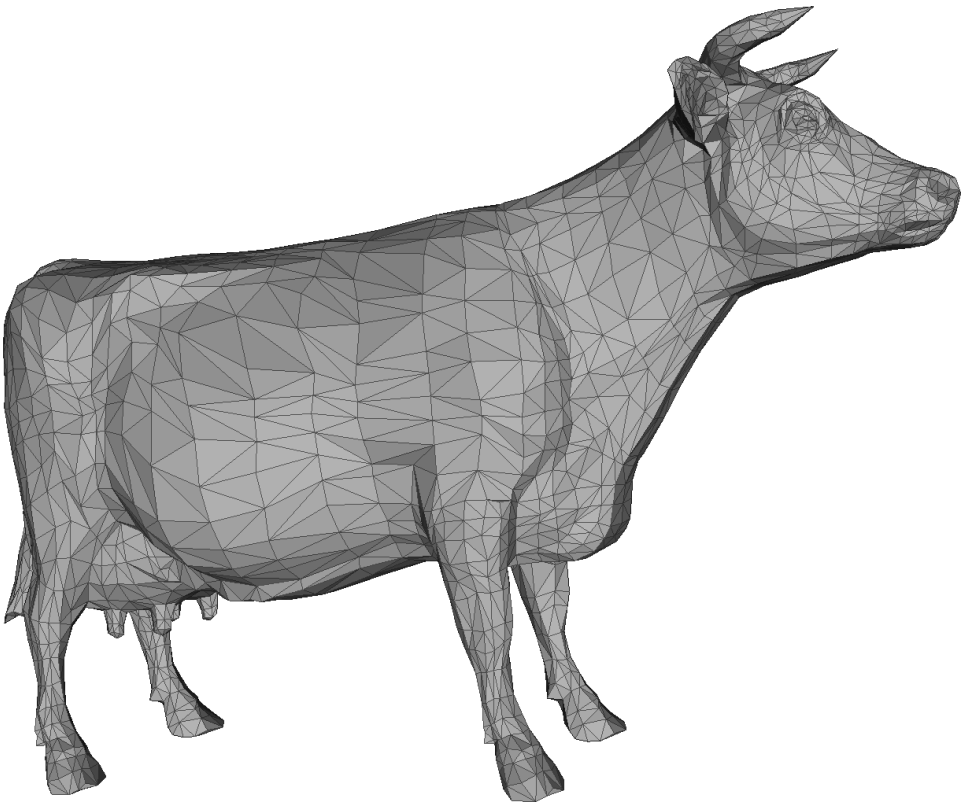


Figure 10.1: A cow figure with 2904 vertices

doing six conquests of the original mesh. This mesh could either be used as the base mesh and sent together with the output of the conquests, or a coarser mesh could be created by doing additional conquests.

As we can see from the images in figure 10.2, the meshes are very different, even if they have the same number of vertices. Of the two meshes, the mesh created by the valence coder is clearly the closest to the original uncompressed mesh.

The valence encoder applies the vertex removals across the mesh, ensuring that the mesh's vertex density is proportionally equal in the different LODs. In figure 10.1 we see that the head of the cow is more frequently populated by vertices and faces compared to the body of the cow. This is still true for the meshes created by the valence coder in figure 10.2 and 10.3. As we also can see in these figures, this is not true for the meshes created by the octree decoder. The octree encoder will split the cells likely to provide the most new information first, based on the cells' size and neighbor relations. In a relatively small mesh like the cow model, the size of the cell's bounding box and neighbor distance will dominate the importance value calculation. During the encoding of the cow model the largest valence of any cell is 15, which is a small value compared to the size of cell's bounding boxes in the upper levels of the tree. Therefore, the large cells will be prioritized in the earlier stages of the encoding, even if some of the smaller cells contain more vertices. Preferably the cells containing the most number of vertices would be

split first, but this is information limited to the encoder, and cannot be done by the decoder. The mesh created from such an octree with low detail uses approximately the same vertex density to display the different areas of the model, as we can see in figure 10.2(b).

Each point added by the valence decoder is correctly placed, and will remain throughout the improvement process. This makes the intermediate meshes good approximations of the original shape. The vertices created by the octree coder are temporary, and will be replaced as long as the cell it represents can be split further.

10.3.2 Errors in the intermediate LODs

The octree coder's intermediate LODs are independent from the final mesh. There can be errors in the intermediate meshes that do not occur in the final one. The valence coder creates a new LOD by building upon the previous LOD, so any potential errors will be present throughout the improvement process. The octree coder is based on progressive improvement of a mesh like the valence coder, but unlike the valence coder the octree coder does not work directly on the mesh.

The valence encoder's decimation step does not allow us to do a vertex removal resulting in non-manifold or non-orientable geometry. The attempted retriangulation after the vertex removal would fail, and we would end up with a hole in the mesh. Its size ranges from one face up to an entire patch. This hole would remain throughout the rest of the conquests, and the decoder would need to receive additional data on how to fill this hole. The octree encoder splits a cell according to the original mesh's vertices located within the bounding box of the cell, and does not take into accord how this changes the mesh representation of the octree. Therefore, the mesh generated from an octree with an intermediate LOD can contain effects like holes. Since the subsequent splits are based on the original mesh and not the current LOD, the hole will be filled when more vertices are created.

The octree decoder creates several disjoint meshes if it is not possible to create a mesh representing all the connectivity data in the octree. If two cells are neighbors, their representative vertices should share an edge. However, in OpenMesh it is not possible to simply create an edge, as it is indirectly created when a face is added to the mesh. That is why two disjoint pieces of the mesh will only be connected when at least two cells in one of the pieces have a neighbor relation to one cell in the other piece. In figure 10.2(a) we see a clear example of this, where a piece of the front leg is missing. When the cells containing the cells are splits later on, enough vertices will be present to create a connected surface.

As we explained in chapter 9, the original octree based compression algorithm is based on meshes without oriented faces. We changed the way the mesh is created so that it works with OpenMesh, which also enables our final mesh to have a defined inside and outside, like the mesh created by the valence decoder. This does however require the first face created to have the correct orientation. Otherwise

each face in the mesh will have the wrong orientation, resulting in a completely black model without any differences in shading.

10.3.3 Errors in the final LOD

If the coders were allowed to compress and decompress a mesh in its entirety, both the results would in most cases be visually identical to the original mesh. The octree coder's decoded mesh can have faces not present in the original mesh. No information about the faces is stored in the octree data structure. We only store information about the edges, and this can result in some wrongly placed faces. If a mesh has a hole the size of one face, the three vertices defining the border of the hole will be connected by edges. A face will fill the hole when this mesh is recreated by the decoder, since their representative cells are connected in the octree. The only other differences are the errors created by the coordinate quantization, which is done by both the encoders.

10.3.4 Quality inbetween the LODs

The octree coder decides on a cell to split by using an importance value, largely influenced by the size of the cells. There are no times when the decoded mesh looks especially good. The improvement of the quality happens by dividing one of the larger cells, improving the quality a little bit each time.

Compression algorithms based on an expanding borderline, like the valence coder, will show a clear difference between the visited and unvisited part of the mesh. The mesh will look the best directly after a full conquest has finished. In figure 10.4, we see a regular mesh being improved by the decoder. The first step the valence decoder does when improving a mesh to the next LOD is to apply the vertices removed by the encoder's cleaning conquest, visualized in figure 10.4(b). Afterwards, the vertices from the general conquest can be added, resulting in a detailed and regular mesh. The cleaning layer only contains vertices with valence three, and creates an irregular mesh. The result is a mesh with a surface visually more unlike the original mesh's surface than the base mesh. In this case it is best to only improve the mesh when enough data is receive to apply both the cleaning layer and general layer, to better preserve the shape of the mesh's surface.

10.4 Internet transmission

The visual quality of the data and the time required to process it is important, but how well the compressed data is suited for internet transmission must be considered as well. We must fit the encoded data into packets when we want to transmit it over the internet, and possibly use a data link with low latency and high packet loss.

10.4.1 Fitting data in an ethernet packet

The main way to assess the success of a compression algorithm is its bit-rate, and this is also very important when we want to transmit the compressed data over the internet. When the bit-rate is high we can fit more improvements in a single packet, and transmit the model using few packets. Since we chose not to implement the arithmetic coding of the encoded data, we use the numbers from table 4.1. The octree coder needs 15 bpv to store a mesh, while the valence coder needs 21 bpv.

The valence encoder stores information needed to add a single vertex, while the octree encoder stores information needed to replace a vertex with up to eight new ones. The differences in the granularity of the improvements do not matter, as many improvements fits in a single packet. However, the valence coder's mesh looks the best after a conquest and cleaning conquest has finished, while the octree coder has an even rate of visual improvement. The maximum size of the payload in an Ethernet packet is 1500 bytes, which is the maximum possible number of improvement information received at once. If we assume we can fill the entire packet with mesh improvement data, the valence encoder can send improvement data for 571 new vertices. Note that the size per vertex will vary, dependent on how effectively the value can be encoded by the arithmetic coder. The number is only an average, so the number of improvements per packet will not be the same, but it is an indicator for the amount of improvements which can be sent at once. This means that we need multiple packets to store a conquest of any larger meshes. The first conquest and cleaning conquest of the cow mesh shown in figure 10.3 removes approximately 800 vertices. This requires the decoder to process two packets to packets go from second highest LOD to the highest LOD.

10.4.2 Transmitting a scene

A 3D reconstruction of a scene will often consist of many different objects. Disconnected objects will be represented as separate meshes, which are handled differently by our coders. The octree data structure can represent any connectivity, both non-manifold meshes and disjoint mesh pieces. Our method for creating a mesh from the data structure only creates one mesh piece at a time. The creation process is repeated until every leaf cell and its connectivity is represented in the mesh. Separate pieces can end up with differently oriented faces, but this can be easily be solved by providing the orientation for the first face in each piece to the decoder.

The borderline of the valence coder only expands to adjacent vertices, and cannot span several mesh pieces. If we want to encode a scene with the valence coder we must either encode and decode one piece of the mesh at a time, or we must use several instances of the encoder and decoder to process the pieces simultaneously.

10.4.3 Packet loss

Neither the valence coder nor the octree coder can handle packet loss. The improvements received by the decoders assume that the decoder has the correct state. An improvement may reference a vertex or cell created during a previous improvement, which is not possible if the packet containing the previous improvement got lost.

Since the octree coder can encode several disjoint meshes simultaneously, it is possible to modify the algorithm to better handle lost packets when transmitting a scene. If a packet only contains improvement for one disjoint piece of the mesh the encoder could send a packet for each separate mesh piece. Then the decoder could process this set of packets in any order, as long as each packet contains a piece identifier. If a packet is lost it still needs to be sent again, but the other packets can be processed by the decoder while it waits.

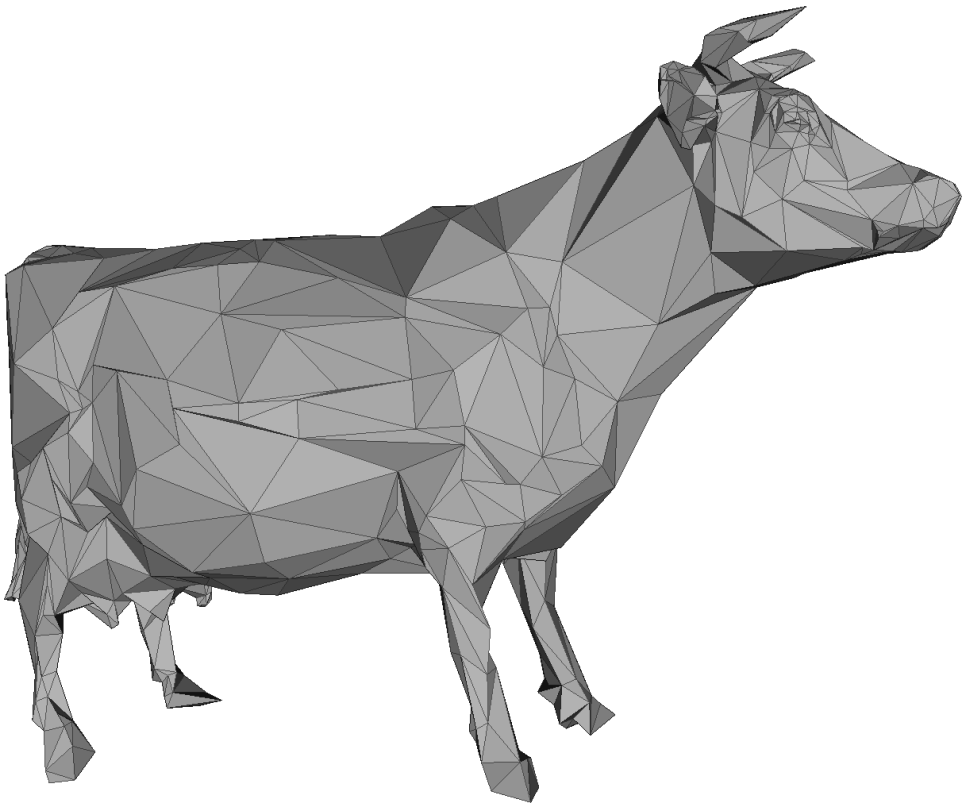
10.4.4 Transmitting a base mesh

The valence decoder requires a base mesh to be used as a starting point, while the octree decoder starts improving the mesh from a single vertex. The ability to not use a base mesh is does not have a large effect on the visual quality of the mesh during the decoding.

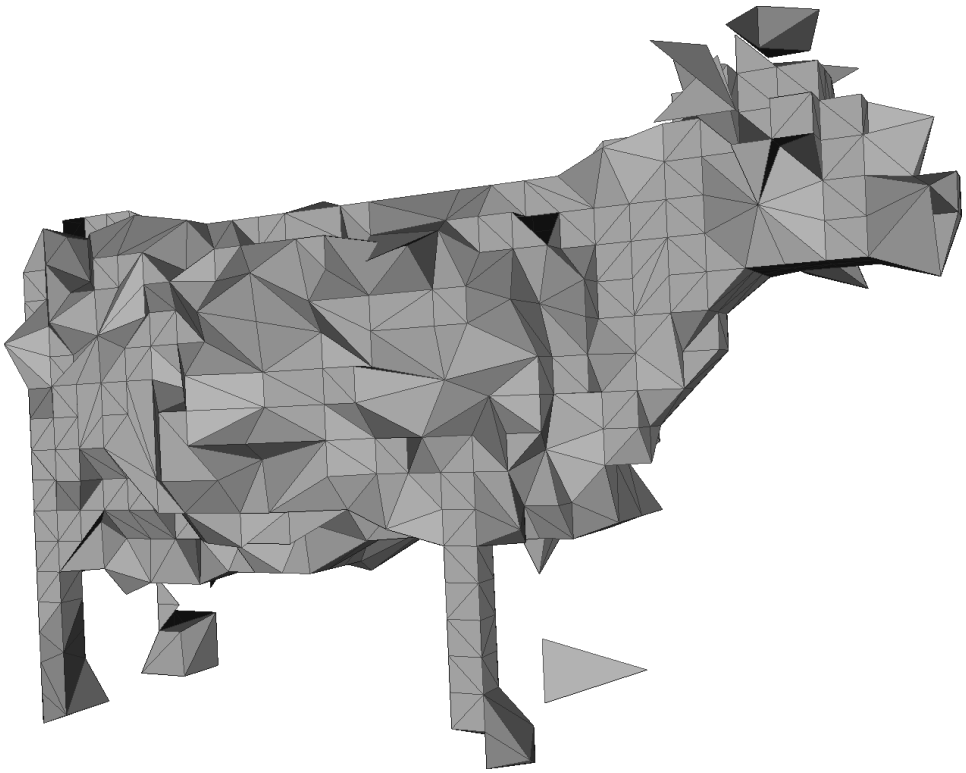
If the output of the octree encoder was stored locally we would be able to visualize the mesh with a single vertex at first, and create more detailed meshes as often we want while the octree is expanded. When the encoded data is received from a remote location we receive data to split a lot more than one cell, and we can create a basic version of the mesh after the first packet.

10.5 Summary

The octree coder is robust, as its data structure supports non-manifold meshes and can contain several disjoint meshes in the same octree. This does, however, require a mesh library that supports non-manifold meshes. The octree coder can more efficiently compress a mesh than the valence coder, using only 71% of the space. However, as we discussed in section 10.3, the valence coders' intermediate meshes have a markedly higher visual quality.

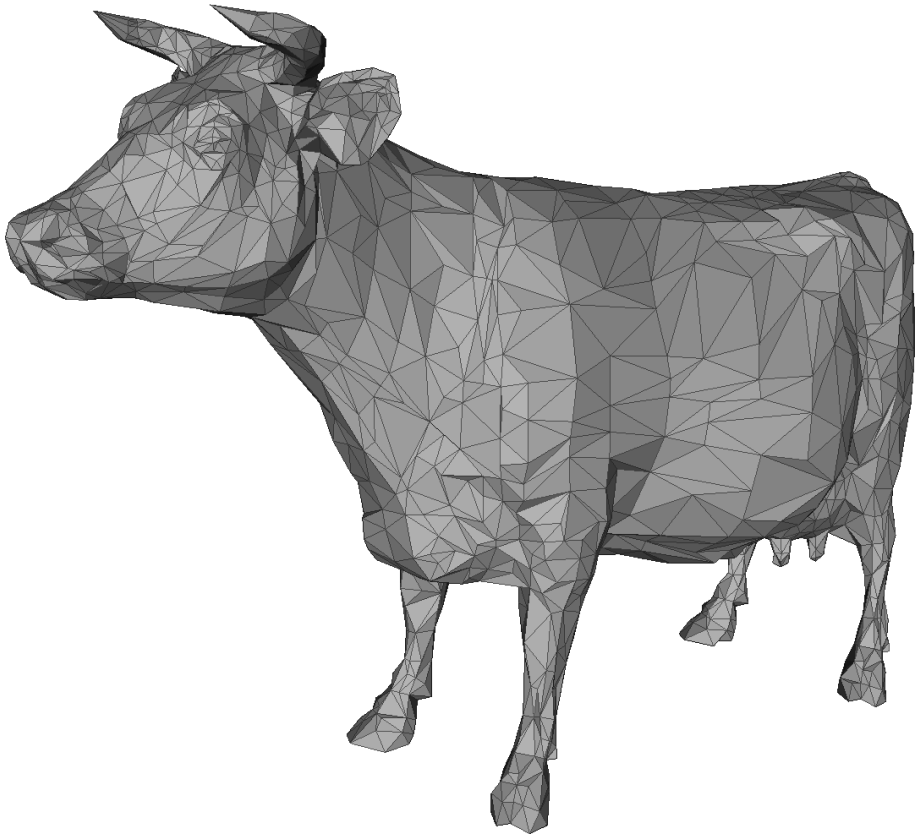


(a) Valence coder

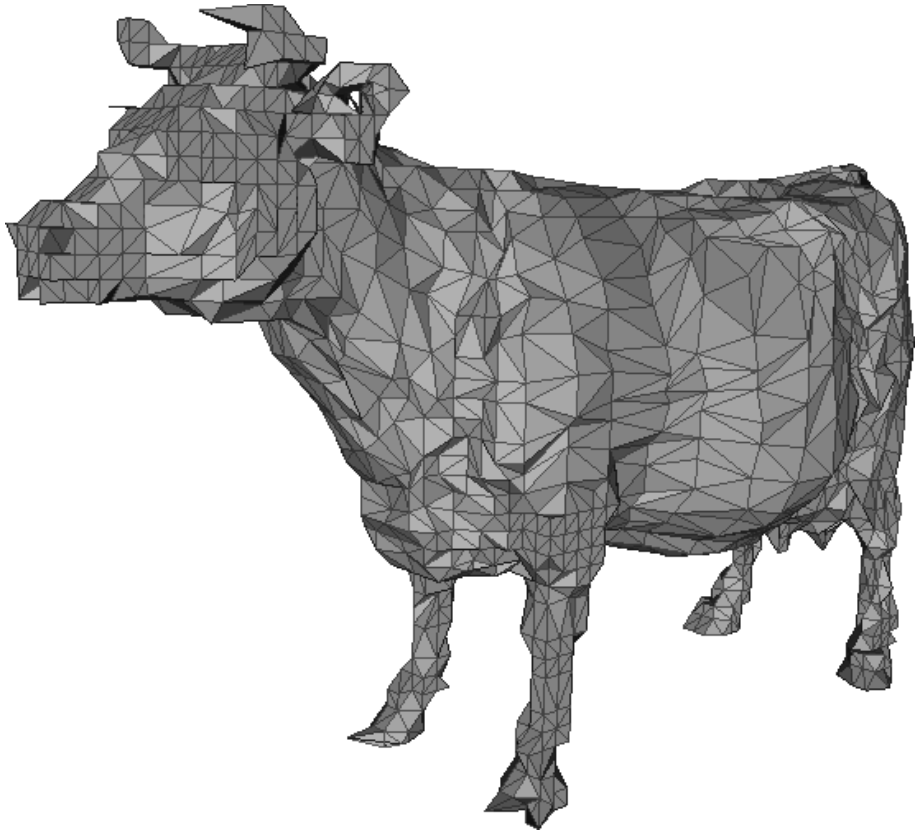


(b) Octree coder

Figure 10.2: The cow model with 800 vertices.

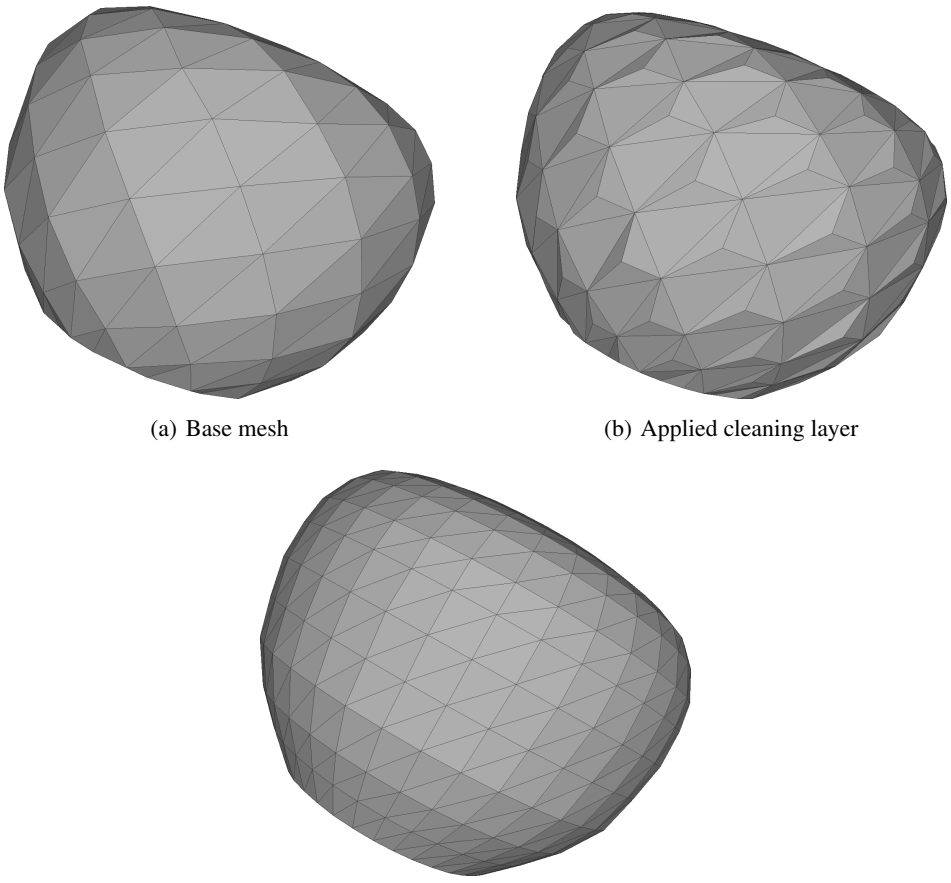


(a) Valence coder



(b) Octree coder

Figure 10.3: The cow model with 2100 vertices.



(a) Base mesh

(b) Applied cleaning layer

(c) Applied general layer

Figure 10.4: Valence coder's improvement of a regular mesh.

11 Conclusion

11.1 Summary

In this thesis we have investigated the different ways to represent 3D data captured from real life, and implemented two mesh compression algorithms.

In the first part of the thesis we investigate the different ways to represent 3D media. A scene can be captured in 3D by capturing systems using cameras or scanners, and we discuss the different ways to represent this data in chapter 2. Image based 3D representations are based on 2D images, adding additional information to contain the depth information. The point cloud and mesh representations are based on points in a coordinate system, where each point represents a sample of the captured 3D scene. A mesh contains additional information to contain the relationship between the points, by connecting them with edges and faces. In chapter 3, we describe how the 3D representations can be compressed.

We discuss how the 3D representations differ when representing captured 3D data in chapter 4, and we chose the triangle mesh as our representation. We also chose two progressive mesh compression algorithms to study further.

The second part of the thesis details our designs and implementations of the chosen compression algorithms. In chapter 5 and 6, we detail our designs of the valence coder and octree coder respectively, which are based on the source paper. We implement our designs in chapter 8 and 9.

We chose the half-edge based library OpenMesh to contain the mesh data, explained in chapter 7. OpenMesh can only support orientable meshes, which we were not aware of until long into the implementation phase, as it is not mentioned

in OpenMesh's documentation. The octree based compression does not originally maintain the normal vectors of the faces when compressing a mesh, but we had to make the mesh orientable for it to be contained by OpenMesh. We created a mesh creation method which preserves the orientation of the faces, as long as the first face is correctly oriented. This method requires more computational power compared to the original method, but it a visual improvement over the original algorithm as we can separate the inside and outside with shading.

We chose the compression algorithms mostly based on the compression rate, but there are other factors than the compression rate that are important as well when transmitting 3D media. We evaluate how well the coders do according to factors important for internet transmission in chapter 10.

11.2 Main Contributions

We wanted to find the best way to transmit captured 3D media over the internet. To do this we needed to find the best suited 3D representation, and how to best compress the data.

We compared the different representations based on their suitability for internet transmission of captured 3D data. We compared image based representations, point clouds and meshes. We found that the triangular mesh is the best 3D representation, as it can best represent the surface of a scene.

We implemented the two algorithms so that we could test them on the factors important for internet transmission. We chose the valence based algorithm by [Alliez and Desbrun, 2001] and the octree based algorithm by [Peng and Kuo, 2005]. They were the most effective progressive algorithms we found that provides lossless compression of the connectivity information.

Most mesh compression literature revolves around the compression rate of the different compression algorithms. When a new compression algorithm is proposed it is compared to the compression rate of the current state-of-the art compression algorithm, and usually pays little heed to the other factors important when compressing media. In this thesis we focused on the factor not generally mentioned by the mesh compression literature, but are still important for internet transmission of 3D media, which are the following:

- Execution time
- Visual quality
- Suitability for internet transmission

We found that the valence coder is the best coder when transmitting meshes, even though the octree coder requires less space, and can easier transmit scenes with multiple disjoint mesh pieces. The valence coder's low LODs have a superior quality compared to the octree coder when the meshes have the same number

of vertices. This means that the valence decoder requires fewer vertices than the deoctree coder to create an intermediate mesh with the same visual quality. It also maintains the normal vector of each face, without needing many more bpvs than the octree coder.

11.3 Future work

Our implementations of the compression algorithms does not contain the arithmetic coding of the output. We omitted it to lessen the scope of the thesis, and it is not needed as we do not measure the compression rate of our implementations. It is, however, needed if we want to have fully functioning coders.

The progressive mesh compression algorithms mentioned in this thesis do not mention textures. The addition of textures to a mesh can be a huge improvement, and it would be interesting to see how to best compress it together with the progressively compressed mesh.

A

Source code

The source code for the implementations of the compression algorithms is licensed under GNU General Public License version 3, and is available on Bitbucket.

The source code of the valence coder is located at:
`https://bitbucket.org/atleno/valencecoder`

The source code of the octree coder is located at:
`https://bitbucket.org/atleno/octreecoder`

Bibliography

[Ahlvers et al., 2004] Ahlvers, U., Zoelzer, U., and Heinrich, G. (2004). Efficient adaptive lossless stereo image coding. In *Proceedings of the 11th Int. Workshop on Systems, Signals and Image Processing*.

[Alliez and Desbrun, 2001] Alliez, P. and Desbrun, M. (2001). Valence-driven connectivity encoding for 3d meshes. In *ACM SIGGRAPH*, pages 198–205.

[Bosc et al., 2010] Bosc, E., Pressigout, M., and Morin, L. (2010). Focus on visual rendering quality through content-based depth map coding. *28th Picture Coding Symposium*, pages 158–161.

[Botsch et al., 2002] Botsch, M., Steinberg, S., Bischoff, S., and Kobbelt, L. (2002). Openmesh - a generic and efficient polygon mesh data structure. In *1st OpenSG Symposium*.

[Boutell et al., 1997] Boutell, T. et al. (1997). Png (portable network graphics) specification version 1.0. https://tools.ietf.org/html/rfc2083 (visited 2016-07-28).

[Caltech et al., 2000] Caltech, A. K., Schröder, P., and Sweldens, W. (2000). Progressive geometry compression. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 271–278.

[Coatsworth et al., 2014] Coatsworth, M., Tran, J., and Ferworn, A. (2014). A hybrid lossless and lossy compression scheme for streaming rgb-d data in real time. *2014 IEEE International Symposium on Safety, Security, and Rescue Robotics*.

[Cohen-Or et al., 1999] Cohen-Or, D., Levin, D., and Remez, O. (1999). Progressive compression of arbitrary triangular meshes. In *Proceedings of the conference on Visualization '99*, pages 67–72.

[Deering, 1995] Deering, M. (1995). Geometry compression. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 13–20.

[Digital Collections,] Digital Collections, T. N. Y. P. L. A klondike camp. http://digitalcollections.nypl.org/items/510d47e0-2bf9-a3d9-e040-e00a18064a99 (visited 2016-05-04).

[Emery, 2010] Emery, D. (2010). Nintendo unveils 3ds handheld games console. http://www.bbc.com/news/10323971 (visited 2016-05-04).

[Furht, 1995] Furht, B. (1995). A survey of multimedia compression techniques and standards. part i: Jpeg standard. *Real-Time Imaging*, 1(1).

[Gonzalez and Woods, 2008] Gonzalez, R. C. and Woods, R. E. (2008). *Digital Image Processing*. Pearson Education, 3 edition.

[Gorley and Holliman, 2010] Gorley, P. W. and Holliman, N. S. (2010). Investigating symmetric and asymmetric stereoscopic compression using the psnr image quality metric. *44th Annual Conference on Information Sciences and Systems (CISS), 2010, US.*

[Gu et al., 2002] Gu, X., Gortler, S. J., and Hoppe, H. (2002). Geometry images. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361.

[Gumhold and Straßer, 1998] Gumhold, S. and Straßer, W. (1998). Real time compression of triangle mesh connectivity. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 133–140.

[Han et al., 2013] Han, J., Shao, L., Xu, D., and Shotton, J. (2013). Enhanced computer vision with microsoft kinect sensor: A review. *IEEE Transactions on Cybernetics*, 43(5):1318–1334.

[He et al., 2013] He, C., Wang, D., and Li, X. (2013). Compression of point set surfaces: a survey. *Computer Modelling and New Technologies*, 17(4):236–243.

[Holliman et al., 1997] Holliman, N. S., Dodgson, N. A., Favalora, G. E., and Pockett, L. (1997). Compression and interpolation of 3d stereoscopic and multiview video. *Proceedings of SPIE - The International Society for Optical Engineering*, 3012:227–238.

[Holliman et al., 2011] Holliman, N. S., Dodgson, N. A., Favalora, G. E., and Pockett, L. (2011). Three-dimensional displays: A review and applications analysis. *IEEE Transactions on Broadcasting*, 57(2):362–370.

[Hoppe, 1996] Hoppe, H. (1996). Triangle mesh compression. In *ACM SIGGRAPH*, pages 99–108.

[Huffman, 1952] Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. In *Proceedings of the IRE Vol.40*, pages 1098–1101.

[Kauff et al., 2007] Kauff, P., Atzpadin, N., Fehn, C., Muller, M., Schreer, O., Smolic, A., and Tanger, R. (2007). Depth map creation and image-based rendering for advanced 3dtv services providing interoperability and scalability. *Signal Processing: Image Communication. Special Issue on 3DTV*, 22(2):217–234.

[Lai et al., 2011] Lai, K., Bo, L., Ren, X., and Fox, D. (2011). A large-scale hierarchical multi-view rgb-d object dataset. *IEEE International Conference on Robotics and Automation (ICRA), May 2011.*

[Li and Kuo, 1998] Li, J. and Kuo, C. (1998). Progressive compression of 3-d graphic models. In *Proc. IEEE*, pages 1052–1063.

[Maglo et al., 2015] Maglo, A., Lavoué, G., Dupont, F., and Hudelot, C. (2015). 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Computing Surveys (CSUR)*, 47(3).

[Otepka et al., 2013] Otepka, J., Ghuffar, S., Waldhauser, C., Hochreiter, R., and Pfeifer, N. (2013). Georeferenced point clouds: A survey of features and point cloud management. *ISPRS International Journal of Geo-Information*, 2(4):1038–1065.

[Ozaktas and Onural, 2008] Ozaktas, H. M. and Onural, L. (2008). *Three-Dimensional Television: Capture, Transmission, Display*. Springer-Verlag Berlin Heidelberg.

[Peng et al., 2005] Peng, J., Kim, C.-S., and Kuo, C.-C. J. (2005). Technologies for 3d mesh compression: A survey. *Journal of Visual Communication and Image Representation*, 16(6):688–733.

[Peng and Kuo, 2005] Peng, J. and Kuo, C.-C. J. (2005). Geometry-guided progressive lossless 3d mesh coding with octree (ot) decomposition. In *ACM SIGGRAPH*, pages 609–616.

[Pribanic et al., 2016] Pribanic, T., Petkovic, T., Donlic, M., Angladon, V., and Gasparini, S. (2016). 3d structured light scanner on the smartphone. In *ICIAR*, pages 443–450.

[Remondino, 2003] Remondino, F. (2003). From point cloud to surface: the modelling and visualization problem. In *Proceedings of ISPRS International Workshop on Visualization and Animation of Reality-based 3D Models*, pages 236–243, ETH Zurich, Switzerland.

[Rusinkiewicz and Levoy, 2000] Rusinkiewicz, S. and Levoy, M. (2000). Qsplat: A multiresolution point rendering system for large meshes. *Proceedings of SIGGRAPH*.

[Tanskanen et al., 2013] Tanskanen, P., Kolev, K., Meier, L., Camposeco, F., Saurer, O., and Pollefeys, M. (2013). Live metric 3d reconstruction on mobile phones. In *IEEE International Conference on Computer Vision*, pages 65–72.

[Taubin and Rossignac, 1998] Taubin, G. and Rossignac, J. (1998). Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115.

[Touma and Gotsman, 1998] Touma, C. and Gotsman, C. (1998). Triangle mesh compression. In *Proceedings of the Graphics Interface*, pages 26–34.

[Wallace, 1992] Wallace, G. K. (1992). The jpeg still picture compression standard. *IEEE Transactions on Consumer Electronics*, 38(1).

[Witten et al., 1987] Witten, I. H., Neal, R. M., and Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6).