Optimizing depth perception for toon-shading in stereoscopic 3D

Dennis Korpel

November 25th, 2022
Version: Thesis defence version

Delft University of Technology

EEMCS

Department of Computer Graphics and Visualization

Documentation

Optimizing depth perception for toon-shading in stereoscopic 3D

Dennis Korpel

1. Reviewer Rafael Bidarra
Department of Computer Graphics and Visualization
Delft University of Technology

2. Reviewer Christoph Lofi
Department of Software Technology
Delft University of Technology

Supervisors Elmar Eisemann and Leonardo Scandolo

November 25th, 2022

Dennis Korpel
Optimizing depth perception for toon-shading in stereoscopic 3D
Documentation, November 25th, 2022
Reviewers: Rafael Bidarra and Christoph Lofi
Supervisors: Elmar Eisemann and Leonardo Scandolo

Delft University of Technology
Department of Computer Graphics and Visualization

EEMCS
Mekelweg 5
2628 CC and Delft

Abstract

The human visual system perceives 3D depth by observing how much points shift between the left eye and right eye. Toon-shaded renders of 3D models often have untextured surfaces with discretized shading, and these flat featureless areas give the eyes few reference points to perceive 3D depth. The primary depth cues come from contour lines and the discrete shading lines dividing bright and dark, so they have to be used to their fullest. This research tries to develop a method for automatically optimizing depth-perception for a 3D toon-shaded scene with smart light placement, local editing of shading, and the generation of effective contour lines.

Acknowledgement

I would like to thank my supervisor, Prof. Elmar Eisemann, for his constructive and positive feedback throughout my long journey. Before starting the thesis, I was toying with some ideas for a subject related to toon shading, but Eisemann brought the concrete problem of toon shading combined with stereoscopic 3D to my attention. As I started investigating it, he gave helpful directions and suggestions, and he was always available for questions.

I would also like to thank Dr. Leonardo Scandolo, for also being present during the meetings and contributing to the discussions, and also continuing to meet me through Zoom when I started working more on my own.

Unfortunately, the combination of the COVID-19 pandemic and my difficulty with doing productive work from home caused a significant delay. On that note, I want to thank my family for encouraging me to keep going, with special thanks to my dad for checking in on my progress while at home.

Contents

1 Introduction

1.1 Introduction

Toon-shading is a rendering style commonly used in video games and animated shows, providing an artstyle reminiscent of traditional cell animation without requiring a large amount of manual labor. Instead of drawing animation frames by hand, a 3D model is rendered with discretized lighting. An example of this is shown in Figure 1.1.

Fig. 1.1.: A 3D model rendered with Lambertian diffuse shading (left) and discretized 'toon-shading' (right)

Discretized lighting replicates the 'flatness' of cell animation, having large areas of a single color without high frequency details. While the backgrounds in traditional animation could be painted with shading and texture, the animated cells were colored flat because drawing texture is a significant effort and hard to do in a consistent, time-coherent manner. Nevertheless, this simplified type of representation can also have its benefits. Research has shown that simplified illustrations are understood and memorized more rapidly. [Win06]

The flat colors do have a disadvantage when used on a 3D screen however, as it can result in a bad depth perception. For example, a review on Disney's 3D re-release of The Lion King[1] mentioned: "Compared to what we are now used to seeing in 3D, The Lion King 3D comes across more like a cardboard cut-out pop-up book."

The lack of high-frequency detail is problematic for stereo-correspondence: the human visual system primarily relies on comparing points from the left eye and right eye view to deduce the depth of the point. Figure 1.2 shows a failure of stereo-correspondence on a toon-shaded model.

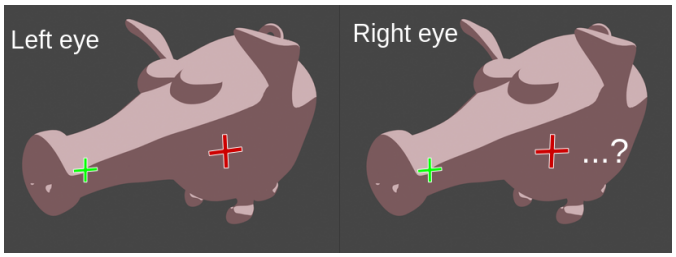

Fig. 1.2.: The point marked in green provides stereo correspondence, since it has a corresponding point in the right-eye view. However, the point marked in red has no contrast with its surrounding, so there is no correspondence.

Previous work on improving depth perception in stereoscopic 3D explores local geometry edits, but in animated and interactive contexts such as Virtual Reality (VR) games, this gives a distorted look when the viewing position changes.

The thesis looks at improving depth perception of toon-shaded renders in stereoscopic 3D by changing the light position, adding local modifications to shading, and using contour lines. A method of finding 'interesting' segments of a 3D model when viewed from a given viewing angle is provided, as well as a vector that moves a point light towards a position that optimizes the depth perception of those segments simultaneously. We also look at a way to edit the shading of segments locally to optimize the light position, at the cost of shading that is not following the rules of physics when it comes to the directionality of the light. Finally, we discuss how to use contour lines and Apparent ridges to improve the depth perception even further, as they allow us to add details on the surface that match the art style but enable an observer to match corresponding points that result in a precise depth localization.

[1] https://www.hollywoodreporter.com/news/how-lion-king-3d-is-236580

1.2 Thesis Structure

In **Chapter 2**, related work on improving depth perception of stereoscopic 3D rendering is discussed.

In **Chapter 3**, we discuss depth perception in stereoscopic 3D and toon-shading in more detail to get a clearer idea of the problem statement. It also discusses curvature, Apparent ridges, and image segmentation, which are tools that are used by our optimization algorithm, which represents a key contribution of this work.

Chapter 4 describes the algorithm and implementation in detail, and shows the results of the separate components briefly.

Chapter 5 goes more in depth evaluating the created method on a wide range of 3D models, and discusses the effect of different parameters.

Chapter 6 concludes the research and provides directions for future work.

Prior art

2

Stereoscopic 3D Several studies have been performed to understand and improve how depth is perceived in stereoscopic 3D renders. This is necessary, since naïve stereo rendering of a CG scene is prone to various issues and has limtations.

For example, there is a limited comfort range in which depth preceived through stereopsis does not conflict with accomodation and vergence (eye focus and rotation). Additionally, in motion picutres, a scene change can also cause discomfort because of the sudden change in depth range. Therefore it can be desireable to modify the depth range of disparity to a comfortable level. Lang et al. [Lan+10] create a method of Nonlinear Disparity Mapping to ease the transition between shots with different depth ranges. Perceptual models for disparity [Did+11][Du+13] can be used to help with this task.

GazeStereo3D [Kel+16] is a method by Kellnhofer et al. for stereoscopic depth adjustment, locally editing disparities based on a focal point obtained from eye tracking or an artist's direction. Scandolo et al. created a method for locally editing vertices to exaggerate depth based on gradients [SBE18], which can help make certain objects stand out and receive a larger share of the comfortable depth range.

Another issue that can arise is conflicting left and right images, 'binocular rivalry'. For example, unless at a large distance from the view point [THA15], specular reflections and refractions can cause contrast differences ('binocular luster') between the left and right eye images. Techniques have been developed to mitigate this issue [Dąb+14] [Tem+12] without resorting to assuming a single viewing point for both eyes, which makes specular reflections appear completely flat.

Other than understanding and modifying perceived stereo depth, prior work has also been done on using light sources to better communicate depth in 2D. Stoppel et al [SEB19] created firefly, a method of using light sources ("fireflies") that move along a closed path to better convey depth on 3D models. The path is automatically created and dynamically updated. This is useful for, e.g., medical visualization, but in toon-shaded scenarios the light source is often the sun, which does not move freely around the subject.

Toon shading The basic toon-shading method of Decaudin [Dec96] has been expanded to allow further fine-tuning. Barla et al. [BTM06] extended 1-dimensional tone-mapping to a 2d map allowing an extra 'tone detail' parameter, which can be used for effects such as levels-of-abstraction, aerial perspective, depth-of-field, backlighting, and specular highlights. They also create a method of interpolating the 'true' vertex normals with an abstracted, modified normal field. Bezerra et al. [Bez+08] created a method for clustering objects in a scene for visual consistency. A method for local edits of two-tone toon-shading has been created by Todo et al. [Tod+07]. They designed a brush tool allowing an artist to modify the shading on toon-shaded 3D models, though only for aesthetic purposes, not in the context of depth perception in stereoscopic 3D.

There are methods to automatically create contour lines for toon-shading, such as Geometric ridges and valleys [OBS04], Suggestive contours [Dec+03], and Apparent ridges [JDA07]. Since line generators can be view-dependent, an additional challenge arises to ensure the line generators are stereo-coherent: each line segment in one view should have a corresponding segment in the other image. This can mostly be mitigated by using a center eye approach, where the line generator uses a point between the left and right eye, but more thorough solutions have been explored [Kim+13] [BSL18] [HWB19]. Other graphics techniques also require special attention to avoid binocular rivalry when used in combination with stereoscopic 3D, such as painting stylization [NAK12] and Ambient Occlusion [SBE22].

In summary, while there is prior art for individual components of this thesis, the combination of toon-shading and depth perception in stereoscopic 3D remains an unexplored topic.

2.0.1 Problem statement

In photo-realistic stereoscopic 3D images, there is usually enough high-frequency detail to create many disparities that help the human visual system preceive depth. When using toon-shading, colors are mostly flat, and the high frequency detail is missing, resulting in poor depth perception. The features that do provide disparities in toon-shading are:

- Natural colors (e.g. spots on a cow)
- The dividing line between bright and dark areas.
- Outlines

While natural colors are usually fixed to the design of the object, there is usually some creative freedom where to place the lights and where to draw the lines. This research aims to find a method of automatically placing a light source and drawing lines in such a way that the 3D depth perception of a scene is enhanced.

3 Background

This section provides background information and theory that will be used and referred back to in the method section (Section 4).

3.1 Stereoscopic 3D

While the human eye only perceives a two-dimensional projection of a three-dimensional world, the human visual system still perceives depth using a variety of depth cues. Even from a two-dimensional image, depth can often be inferred from perspective, occlusion, and known object size among other things. Further cues can be derived from the eye's focus (accommodation and vergence), and the temporal domain (motion of objects yielding perspective changes). The relative strengths of depth cues has been explored by Cutting et al. [CV95], and can be seen in Figure 3.1. One of the strongest cues is stereopsis: the ability to perceive depth from the difference in the image seen by the left and right eye. Because the two eyes are horizontally offset, the difference in position of objects between the two images is used as a measure of depth.

The difference between two such images is called horizontal disparity. The goal of stereoscopy is to give images a 3D depth by showing two images, one per eye, with horizontal disparity similar to what you would get from stereopsis.

When rendering a 3D scene in a computer graphics context, generating the two images with correct horizontal disparities can be relatively simple. Given an existing virtual camera, it simply requires another render pass with a second virtual camera slightly offset, based on the distance between eyes ('Interocular Distance'), to generate an image for the other eye. Displaying the two different images to each eye does require special hardware however, and there are several methods for it, among which the following:

- A stereoscope. This puts the two different images right in front of each eye, either on a screen (like a Virtual Reality Headset) or a printed photo (View-

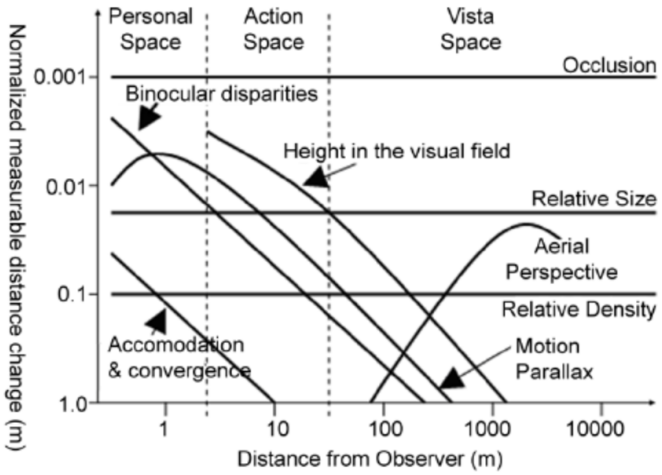

Fig. 3.1.: Relative strength of depth cues, adapted from Cutting et al. [CV95]

Master). A special lens between the eye and the image makes it appear further away.

- Anaglyph glasses. For this technique, the two images are tinted in opposing colors, commonly red and cyan or red and green, and overlayed on top of each other. By wearing red and cyan tinted glasses, the image is separated into the left and right parts for each eye. While this is a simple and cheap way to convey the depth, the resulting image gives unpleasant color perception, and it relies on light-dark contrast to convey depth. For example, a red object on a white background can completely disappear in the red channel / left eye, so it will not appear in 3D then. Since this research focuses on depth perception and not aesthetics, 3D figures in this thesis will use this technique.

- Polarized glasses. The left and right images are projected onto each other using a polarizing filter, and the viewer wears glasses with the same polarizing filter. It requires special screens or projectors, but the polarized glasses are very cheap, making it effective for large screenings such as cinemas.

- Active shutter glasses. An active shutter system requires glasses that are synchronized with the monitor: the monitor altnerates showing the left and right images, and the glasses darken each eye alternatively such that the left and right eye only see the monitor when it is showing the right image for them. This requires a monitor with high refresh rate and more expensive glasses, and the shuttering also results in a darker image. They do not halve the resolution like polarized monitors however.

- Autostereoscopy. This involves no glasses, but the screen having a mechanism to show a different image to each eye, e.g. a parallax barrier. This is very sensitive to the viewing angle, making it adequate for a portable device such as a gaming handheld like the Nintendo 3DS, or a viewfinder on a 3D camera. It does not work on large screens, or on screens viewed by multiple people.

When the stereoscopic images are viewed on a screen instead of a stereoscope, the left and right image generation needs to account for the fact that the eyes accomodate to the screen a short distance away. Like Figure 3.2 shows, the left and right image need to shift a bit inwards depending on the distance between the eyes and the screen.

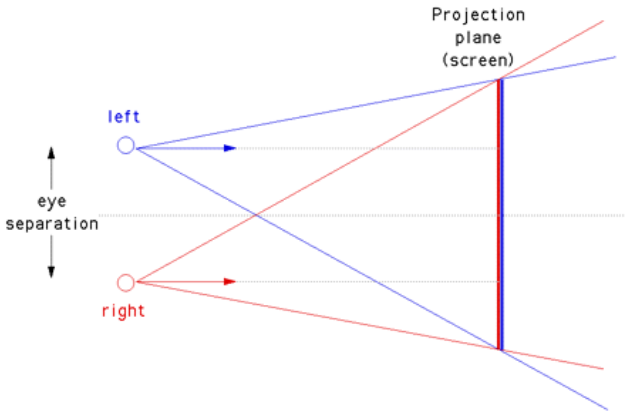

Fig. 3.2.: The off-axis projection method, accounting for the eye's accomodation to a nearby screen

There are two main ways to this: off-axis and toe-in. The off-axis method horizontally shifts both images to reproduce the slanted view frustrum as seen in Figure 3.2. The amount of shift is computed from the distance between eyes ('Interocular Distance') and distance to the screen ('Convergence plane distance').

The 'toe-in' method involves tilting the two virtual cameras inwards a little. However, this results in skewed lines, as shown by Figure 3.3. It should only be used when horizontal shifting is not available in the render pipeline.

3.2 Stereo correspondence

One important aspect of stereopsis is making sure a matching point actually represents the same physical location in the world: The stereo correspondence problem[Tho+16]. If the matching criterion is too strict, insufficient depth cues can

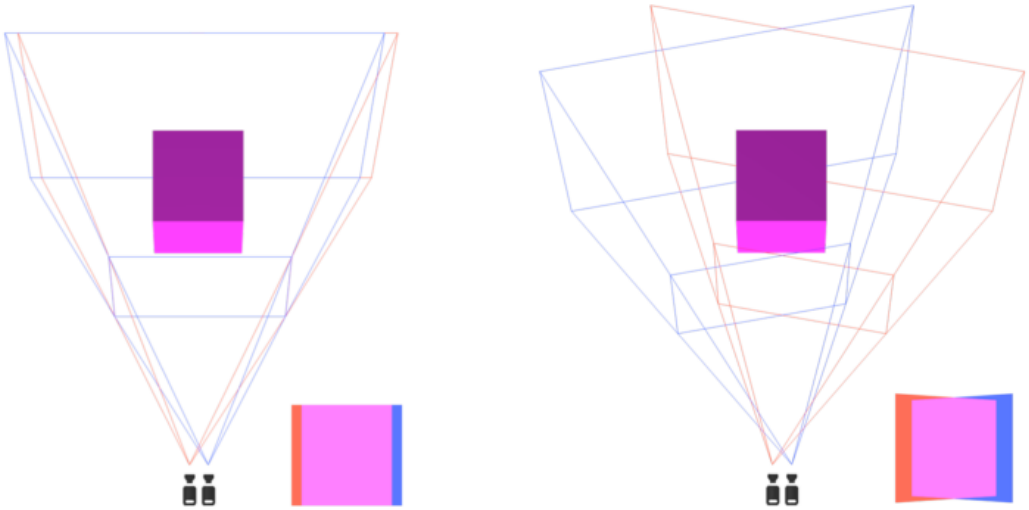

Fig. 3.3.: Comparison between Off-axis stereo (left image) and Toe-in stereo (right image). Respective rendering outcomes are placed side by side.

be obtained. If it is too loose, this will lead to inconsistencies and errors in depth perception. Figure 3.4 shows an example of stereo correspondence.

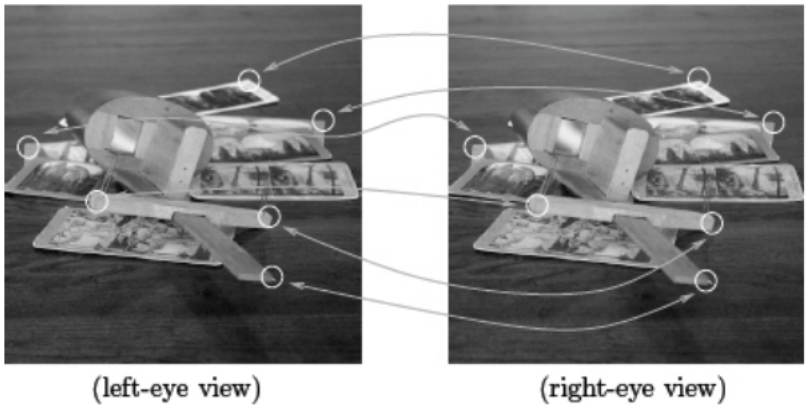

Fig. 3.4.: Example of stereo correspondence: points from the left-eye view are matched with points from the right-eye view to deduce the depth.

It has been shown that luminance is an important matching criterion, but color is not, which is why the anaglyph viewing method works even while having different colors in the left and right image. A lack of luminance difference in the image makes it difficult for stereo correspondences to be found, making stereopsis ineffective.

3.3 Toon shading

'Toon-shading' or 'Cel-shading' refers to using 3D rendering techniques while generating images that appear like a hand-drawn cartoon. It is a form of Non-Photorealistic Rendering (NPR), a branch of Computer Graphics (CG) where the focus is not to produce images of similar quality to photographs, but stylistic imagery instead that shows a simplified view of the world. This can be useful for explanatory purposes since it allows distracting details to be removed, leaving only the essence of the image. It is also used by artists in games and animations as an artistic choice.

A major characteristic of toon-shading is the use of a limited color palette, because traditional animation requires drawing many cells and applying smooth shading manually with paint is hard to do in a time-coherent manner.

In photo-realistic 3D graphics, many colors are used as the result of image textures with high frequency details and applying a shading model. By avoiding such textures and discretizing the shading model, the flat color look of a cartoon can be achieved.

Simple but effective CG shading models use the Lambertian reflectance to model diffuse lighting and Phong or Blinn-Phong shading to model specular lighting. Lambertian diffuse shading is computed as a constant value (based on light and surface properties) multiplied by the cosine of the angle between the surface normal and light direction. This angle is illustrated in Figure 3.5.

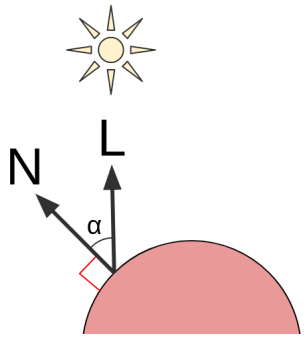

Fig. 3.5.: The key component of diffuse shading is angle α between the surface normal N and light vector L.

This cosine can be computed using a dot product, as shown in Equation 3.1.

$$I_d = k_d \times i_d \times L \cdot N \tag{3.1}$$

- I_d is the illumination of a point on a surface by diffuse shading.
- k_d is a constant based on surface properties

- i_d is the light intensity for diffuse shading
- L is a normalized vector pointing from the surface point that is being illuminated to the light source
- N is the normal vector of the surface point that is being illuminated

For toon-shading, the result of the dot product $L * N$ is discretized. If it is above t, the I_d is 'bright', otherwise I_d is 'dark'. The exact values of the threshold are up to the artist, but they could be for example $t = 0.5$, $bright = 0.75$, $dark = 0.25$. This way of doing toon-shading is popular and has been described by Decaudin [Dec96]. Their result can be seen in Figure 3.6.

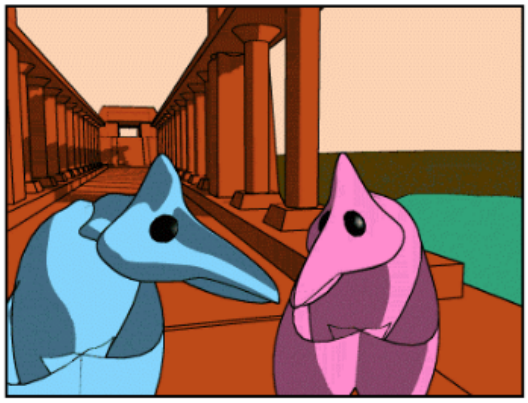

Fig. 3.6.: Early attempt at toon-shading by Decaudin

Note that for aesthetic purposes and to avoid aliasing artefacts, sometimes the dot product $L * N$ is tone-mapped to a one-dimensional texture that almost resembles a harsh threshold, but has been smoothed a little bit. In this thesis, we assume a strict discrete transition between 'bright' and 'dark' for simplicity. A slight easing in the transfer function should not affect the overal result.

Another major characteristic of toon-shading is outlines, reminiscent of pencil strokes or ink. Decaudin's method [Dec96] draws simple contours by looking for discontinuities in image space of each pixel's depth and normal vector. However, several other line-drawing methods have been developed since, such as:

- Geometric ridges and valleys [OBS04]
- Suggestive contours [Dec+03]
- Apparent ridges [JDA07]

It has been shown that existing definitions explain 80% of the lines artists would draw [Col+08]. In this research, Apparent ridges are used as the main basis because they include occluding contours, and adding suggestive contours and geometric ridges and valleys have a tendency to draw more lines than needed for cartoon purposes.

3.4 Curvature

Toon-shaded contours are often drawn based on the curvature of the surface. Curvature describes the rate of change across the tangent of a surface.

The curvature on a point of a two-dimensional curve is computed as the inverse of the radius of the 'Osculating circle', the circle grazing the curve the tightest at that point. A point on a surface of a 3D mesh can have its curvature computed by taking a two-dimensional slice perpendicular to the normal plane. When the product of the curvature of two orthogonal planes is at an extreme, the values are the principal curvature, see Figure 3.7

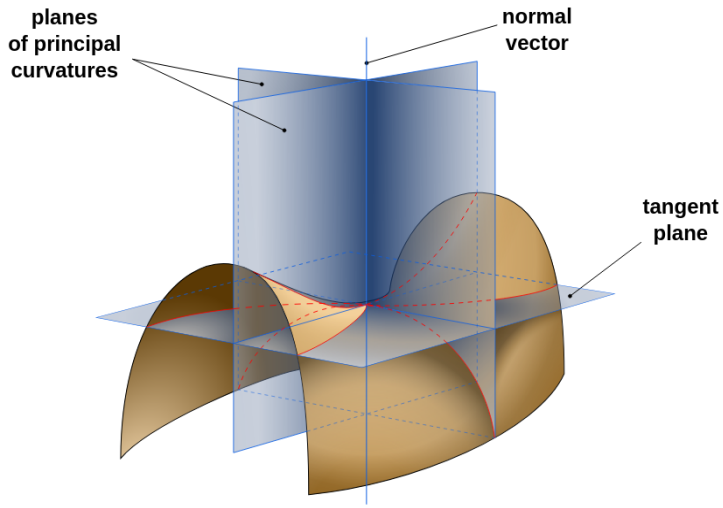

Fig. 3.7.: Saddle surface with normal planes in directions of principal curvatures. Source: Wikimedia

The product of the two principal curvatures $k_1 * k_2$ is the Gaussian curvature. The Gaussian curvature is positive for convex (spherical) shapes, negative for concave (saddle) shapes, and zero for a flat plane.

View-dependent curvature is a measure of curvature not aligned with the surface normal, but instead with the view vector. Apparent ridges use view-dependent cur-

vature: ridges are placed where the view-dependent curvature has a local maximum, and is above a certain threshold. This threshold is to prevent lines being placed at very minor dents, or lines as a result from noise in discrete curvature calculations.

3.5 Image segmentation

When looking for features of a 3D model on an image, it can be useful to segment an image based on curvature. A segment is a group of pixels that are similar to one of their neighbours by some equivalence class.

An image can be seen as a graph where each pixel is a node with edges to its 4 (cardinal) or 8 (cardinal and diagonal) neighbouring pixels, and then graph clustering algorithms can be used.

Clusters can be created and merged efficiently using a union-find data structure[1]. The Hoshen Kopelman algorithm[2] scans an image scanline by scanline and assigns labels to each pixel. At the end, each unique label represents one segment (Figure 3.8), making filtering pixels of a single segment simple.

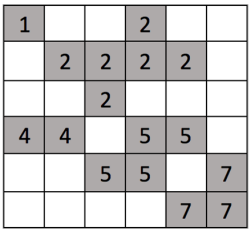

Fig. 3.8.: Example output of Hoshen-Kopelman algorithm. Source: Wikimedia

[1] https://en.wikipedia.org/wiki/Disjoint-set_data_structure
[2] https://en.wikipedia.org/wiki/Hoshen%E2%80%93Kopelman_algorithm

4 Method

In this section the techniques created in order to solve the stereo correspondence problem of toon-shading are described, as well as the motiviation behind their design. Extensive results of the techniques and evaluation can be found in the results section (Section 5). The main contributions are:

- A measure and its derivative for optimizing the position of a point light, resulting in toon shading better conveying the depth of a simple shape
- A way to find 'simple shapes' in image space by segmenting based on gaussian curvature
- An iterative algorithm to optimize the light position for all segments simultaneously
- A method of using local shading edits to further improve the depth perception
- A smoothing step for Apparent ridges making them less noisy with low ridge thresholds
- A distance field to identify places with low stereo correspondence on a toon-shaded image

4.1 Optimization of light position

One element to add depth-revealing disparities is the dividing line between bright and dark sections. For example, by adding a noise function to the shading such that small bright and dark spots cover the surface, more features can be matched. An example of this is shown in Figure 4.1. However, while a small amount of noise can situationally be added to imply rough surfaces, adding noise is generally unacceptable for toon-shading.

One parameter that can be controlled while retaining the toon-shading aesthetic is the position of the light source, and consequently the position of the shading line that constrasts bright and dark parts of the surface. To find an intuition of what

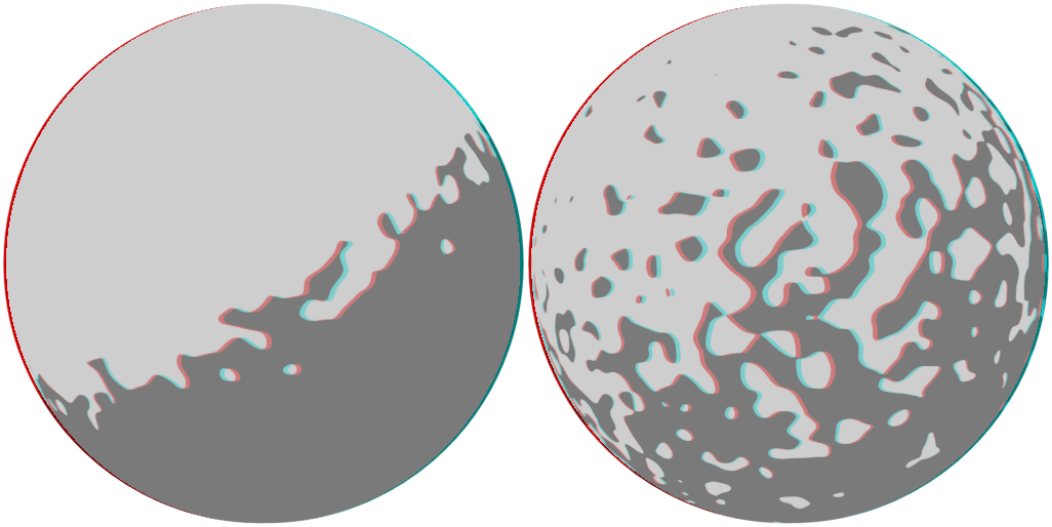

Fig. 4.1.: Effect of noise on diffuse shading. While noise improves depth perception, it results in unphysical shading.

makes a good light placement for depth-perception, we start with a simple case and try to make it more general.

Consider a perfect sphere that is toon-shaded and illuminated by a single light source. The possible light positions can be categorized into these groups based on the resulting shading:

- The light is behind the sphere relative to the camera. This means the sphere is rendered mostly dark and appears as a flat circle on both the left and right images. This gives no cues for depth perception, and consequently the sphere is observed as a flat disc.
- The light is at the camera, illuminating the front side of the sphere. Similar to the previous case, this renders a flat circle giving poor depth perception.
- The light is shining from above or below, casting a perfectly flat shading line across the middle of the sphere. Because a horizontal line aligns with the offset of the two cameras rendering the stereoscopic images, this line will be the same for both the left and right eyes. This removes the ability to distinguish the sphere from a disc with two colors, and consequently the resulting sphere looks like a flat disc with a bright halve and a dark halve.
- The light is illuminating the sphere at an angle, creating a curved line. This gives relatively good depth perception, since there will be a curved shading line indicating the curvature of the sphere, so it will not be perceived as a flat disc.

- The light is close enough to create a bright circle on a dark sphere. This also distinguishes the sphere from a flat disc.

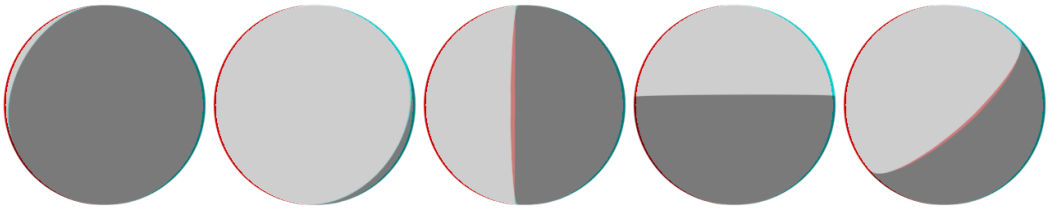

Fig. 4.2.: Light positions: *(a)* behind the sphere, *(b)* in front of the sphere, *(c)* vertical split, *(d)* horizontal split, *(e)* close to the surface

Figure 4.2 illustrates these light positions.

From these observarions, a simple metric can be derived to distinguish good light positions from failure cases: The $bright/dark$ ratio should be near even. When this is the case, there is a long contrast line covering the curvature of the sphere, providing many points of stereo correspondence.

The metric does not exclude case (d), the perfectly horizontal shading line. However, in practice this case rarely comes up and can easily be mitigated by a small adjustment of the light position. We assume this case does not come up, but it could be mitigated in future work if such failure cases are found.

4.1.1 Algorithm

Given the previous examples, we want to create an algorithm that can place the light in such a way that roughly 50% is in light and 50% is in shadow.

A naive way of placing the light to create a near 50:50 ratio is brute-force:

1. Randomize the x, y and z position of the light source
2. Measure the amount of bright and dark pixels
3. Repeat until the bright/dark ratio is inside and acceptable ratio range

Pseudo code for this is in Listing 4.1.

```
1
2  void optimize(desiredRatio, margin)
3  {
4    while abs((brightPixels / totalPixels) - desiredRatio) > margin
5    {
6      light.position = vec3(rand(-1, 1), rand(-1, 1), rand(-1, 1));
7      render() // render with light, compute number of bright pixels
8    }
9  }
```

Listing 4.1: Brute force algorithm pseudo code.

When the acceptable ratio range is very narrow, this can take a long time. On a mesh with large flat surfaces (e.g., a cube) there are many discontinuities: a face can turn from completely dark to bright with a small light position change. On a curved surface, the bright/dark ratio is mostly continuous, so a small adjustment in light position often gives a small adjustment in bright/dark ratio. Therefore, this algorithm can be made more efficient by letting it converge to a ratio via the use of simulated annealing: Initially modify the light position using large random offsets, but reduce the offset size in every iteration, and reject modifications that make the result worse. See Listing 4.2 for a pseudo code example.

```
1  void optimize(desiredRatio, margin)
2  {
3    s = 1 // step size of light position modification
4    oldError = abs((brightPixels / totalPixels) - desiredRatio)
5
6    while (oldError > margin)
7    {
8      oldPos = light.position
9      light.position += vec3(rand(-s, s), rand(-s, s), rand(-s, s))
10     render()
11     error = abs((brightPixels / totalPixels) - desiredRatio)
12
13     if (error > oldError)
14     {
15       light.position = oldPos
16       error = oldError
17     }
18     else
19       s *= 0.95
20   }
21 }
```

Listing 4.2: Simulated annealing pseudo code.

This is effective in giving a good ratio, but the disadvantage is that the artist loses all control and the result can be unpredictable: when lighting a sphere, many solutions are possible, such as placing the lamp above, under, left of or right of the sphere.

Given that applying random offsets can result in unpredictable outcomes, in the next subsections we will look into a method that analyzes the current light configuration and deterministically determines the best local offset towards a desired light/shadow ratio.

4.1.2 Derivative of diffuse shading

At a set point on static geometry, diffuse shading can be seen as a scalar function: the input is a vector from a point on the surface to the light source, and the output is the brightness of the surface at that point. Before clamping the brightness to be non-negative and discretizing it for toon-shading, this function is differentiable: As discussed in Section 3.3, it is the dot product of two vectors: the normal vector n and the normalized light vector.

When we consider a point on the surface with coordinates $(0, 0, 0)$, the light vector is equal to the light position l, and we can analyse the derivative of the brightness with respect of each coordinate of l: see Equation 4.1 for an example with the x-coordinate.

$$\frac{\partial}{\partial l_x}\frac{l_x \cdot n_x}{\sqrt{l_x^2 + l_y^2 + l_z^2}} + \frac{l_y \cdot n_y}{\sqrt{l_x^2 + l_y^2 + l_z^2}} + \frac{l_z \cdot n_z}{\sqrt{l_x^2 + l_y^2 + l_z^2}} \tag{4.1}$$

After doing this for y and z as well, Equation 4.2 can be derived to efficiently calculate the derivative of the brightness with respect to the light position.

$$(\frac{-(l \cdot n)}{l \cdot l} \cdot l + n)/\|l\| \tag{4.2}$$

By moving the light source parallel or antiparallel to this gradient vector, the point achieves its maximum and minimum rate of light increase respectively. In almost all cases, this means that the point gets brighter or darker, respectively. Visually, the gradient vectors represent the overall 'shading direction', as can be seen in Figure 4.3.

By summing the light gradient for each pixel in the rendered image, a 'global gradient vector' can be computed. The direction of this vector can be used to move the light in

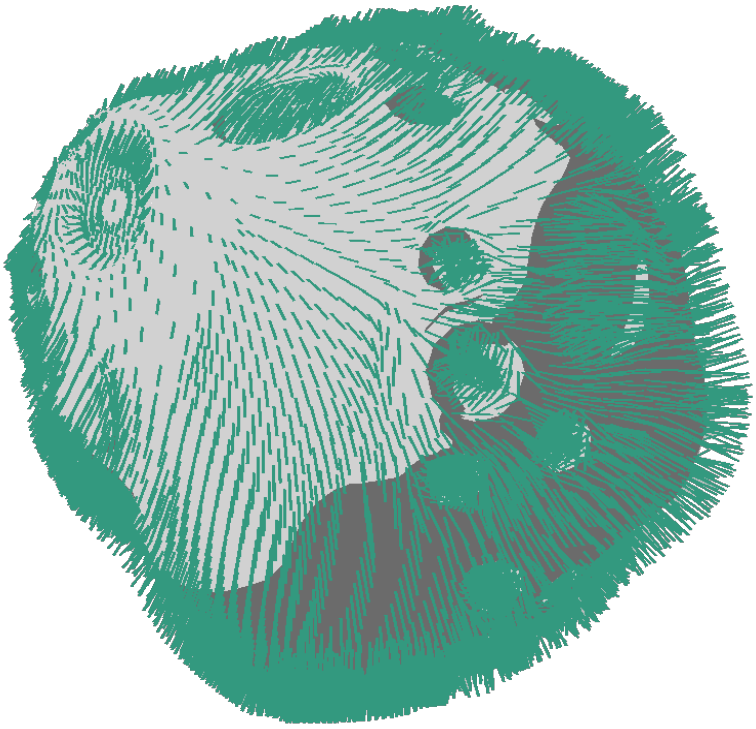

Fig. 4.3.: The diffuse shading gradient plotted at every vertex of a mesh. Intuitively, this describes the overall 'shading direction' going diagonally from the top-left.

a way that changes the brightness ratio while maintaining the overall light direction. By repeatedly moving the light in small steps along the vector, a maximum brightness will be approached. Similarly, moving the light in steps antiparallel to the gradient will approach a minimum brightness.

This can be used in an iterative algorithm to find a light position that gives a desired brightness ratio, while preserving the overall shading direction of an initial light position. This gives more control over the end result to the artist, unlike the global simulated annealing method, which is not sure to find a desired ratio, and the resulting light position can be hard to predict or control. Listing 4.3 shows pseudo code for this algorithm.

```
void optimize(desiredRatio, margin)
{
  s = 0.1 // step size of light position modification
  signedError = (brightPixels / totalPixels) - desiredRatio

  while (abs(signedError) < margin)
  {
    for (pixel in image)
    {
      dSum += brightnessDerivative(
        pixel.normal, normalize(pixel.position - light.position)
      )
    }
    d = normalize(dSum) * s
    render()
    signedError = (brightPixels / totalPixels) - desiredRatio

    if (signedError < margin)
      light.position += d
    else if (signedError > margin)
      light.position -= d
  }
}
```

Listing 4.3: Optimization using brightness derivative of light position.

4.1.3 Segmentation

A limitation of the above method is that it only works well when the scene consists of a single sphere-like mesh, while 3D toon-shading scenes are usually more complex than that. Consider there being a small dent in the sphere. If the dent is not near the shading line, it will be completely invisible.

Contour lines can reveal the location of the dent, but it can still be hard to see whether the dent is convex or concave.

Therefore it is still desirable to find a light position that puts the shading line across the dent (Figure 4.4).

To achieve this, the region with the dent should have a near 50/50 brightness ratio, while the region of the overal sphere should also have this constraint. The previous algorithm can be expanded with two extra steps:

- Segmenting the image into parts with salient shape

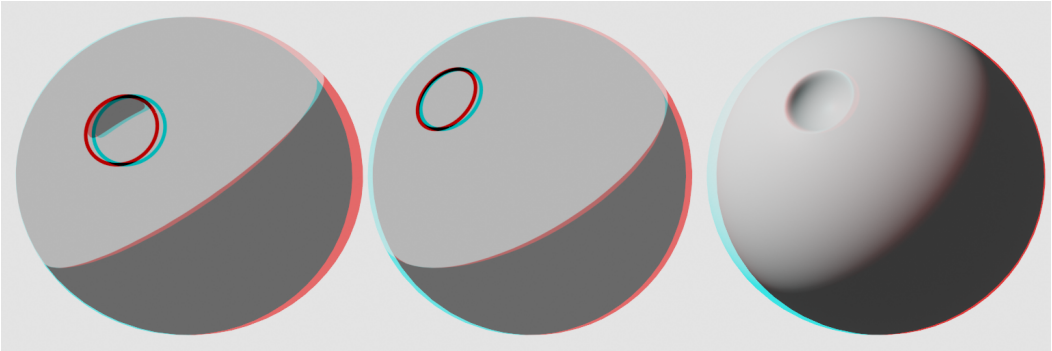

Fig. 4.4.: A sphere with a concave dent in it with toon shading. Left: Using both a contour and a shading line, the shape of the dent is revealed. Middle: with only a contour, the same dent looks flat instead of concave. Right: the same dent with lambertian shading, for reference.

- Optimizing the light source position for all segments simultaneously

Finding the salient parts is done using Gaussian curvature (see Section 3.4).

A sphere-like model has only positive Gaussian curvature while dents create saddle shapes with negative Gaussian curvature, as can be seen in Figure 4.5. Therefore a connected component labelling algorithm can be used based on Gaussian curvature: two neighbouring pixels belong in the same connected component if their Gaussian curvature has the same sign (i.e. both are positive or both are negative).

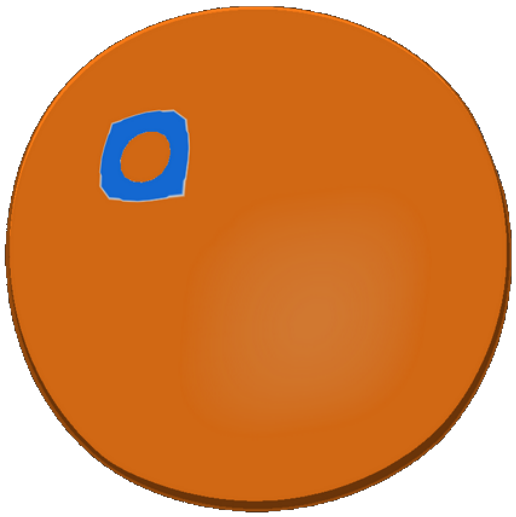

Fig. 4.5.: The Gaussian curvature of a dented sphere. Orange is positive curvature, blue is negative curvature.

To prevent separate objects from merging together into one segment, an additional condition is that two neighbouring pixels must not have too large absolute difference in depth. Separate objects with the same curvature might touch in image space, but

should still be separate segments. An example of such a configuration is shown in Figure 4.6.

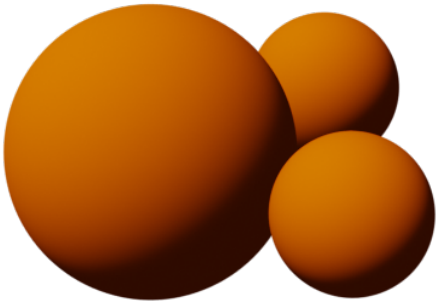

Fig. 4.6.: The three spheres all have positive curvature and touch in image space, but should still get their own segment since they do not form a continuous shape.

To optimize the light position for the found segments, the gradient vector is calculated for each segment. The gradient vectors are normalized and multiplied by the size in pixels of each segment. This way the length of the vector represents the 'importance' of the vector, where smaller segments have smaller importance since they contribute less to the overall image.

Now we want to solve the bright/dark ratio for each segment independently. In order to provide shading lines over all regions, they need to have their own near-even bright/dark ratio. A fully bright region cannot compensate for a fully dark region, since that would not result in shading lines covering each segment.

The weighted gradient vectors are summed to find the direction that the light should move into in order to improve the brightness ratio of all segments at the same time. The light source is moved in this direction until it converges to a fixed point, which is a local optimum. The solution still depends on the initial light position as shown in Figure 4.7.

4.2 Lines

Line drawing is not only part of the aesthetic of cell animation, it is also integral to depth perception of stereoscopic toon shading. Black outlines are very high contrast details that give good stereo correspondence, and since they are not related to the light position, they can complement the shading line very well.

While the shading line can cover a dent on a sphere to show whether it is convex or concave, a line stroke can show its position and shape better.

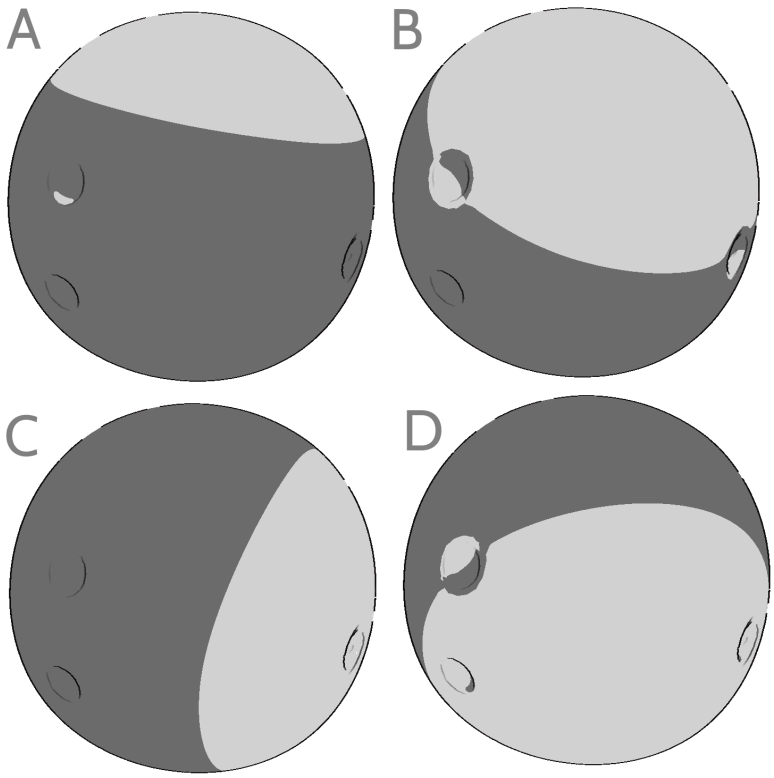

Fig. 4.7.: (A) initial light position, (B) optimized light position from starting point A. (C) Different initial light position, (D) optimized light position from starting point C.

The disadvantage is that there is less control over the placement of lines: While a shading line can be made to cross a curved surface anywhere by adjusting the light position, the placement and direction of line strokes has to make sense with regards to the geometry of the model.

As discussed in the background Section 3.3, existing definitions explain 80% of the lines artists would draw intuitively. In this thesis, Apparent ridges are used for adding line strokes [JDA07]. To make them work in stereoscopic 3D, a center eye approach is used. Instead of generating them for both the left and right image independently, they assume a viewing direction inbetween the two in order to reduce binocular rivalry artefacts.

Apparent ridges are drawn when view-dependent curvature has a maximum, and is above a certain threshold to account for noise. (Section 3.4)

The implementation computes view-dependent curvature and its derivative for each vertex of the model. A zero-crossing of the derivative between two connected vertices means there is a maximum on the edge, so a line vertex is placed on the

edge if the view-dependent curvature is above the threshold. For each triangle, the zero crossings are connected and line segments are generated.

The line segments are rendered as thick lines using a geometry shader that renders each segment as two triangles forming a quad. The lines are given a depth offset in the z-buffer to make them less occluded by the geometry that they rest on.

It is hard to modify line placement by editing the view-dependent curvature in a controllable way, but we can modify the threshold for adding lines based on the need. The threshold does not need to be uniform across the model: areas of the model with low stereo correspondence from lighting can have a lower apparant ridge threshold so more lines are generated that provide points of stereo correspondence.

4.2.1 Line clustering

When the mesh resolution is too low, or the tesselation is irregular, Apparent ridges can result in irregular curves instead of smooth strokes that are expected in toon shading. When the threshold of view-dependent curvature for lines is lowered to increase line output for better depth perception, this becomes particularly noticable, see Figure 4.8.

To mitigate this, an approach of line clustering was tried. Apparent ridges produces separate line segments on a per-triangle basis, based on the view-dependent curvature of their vertices. By clustering separate line segments into line strokes, or ‘snakes’, the outlines can be improved by smoothing ‘snakes’ and discarding short ones, which are likely noise.

To do this, an algorithm was written that keeps track of each snake as a linked list of line segment vertices. The unconnected vertex of the first line segment is the head, and the last vertex of the line segment at the end of the list is the tail. Every line segment is compared against the head and tail of each snake. If the screen-space projected distance is smaller than a threshold, it is joined. After being joined to one snake, it is compared against all heads and tails of existing snakes, so that two snakes can combine into one. When snakes are joined into a closed curve, it is not considered anymore for joining any subsequent snakes or line segments.

Each line segment connects to either 2, 1 or 0 existing snakes. In case of 0 connections, the line segment starts a new snake of length 1. Figure 4.9 shows an example of a line segment connecting to 2 existing snakes.

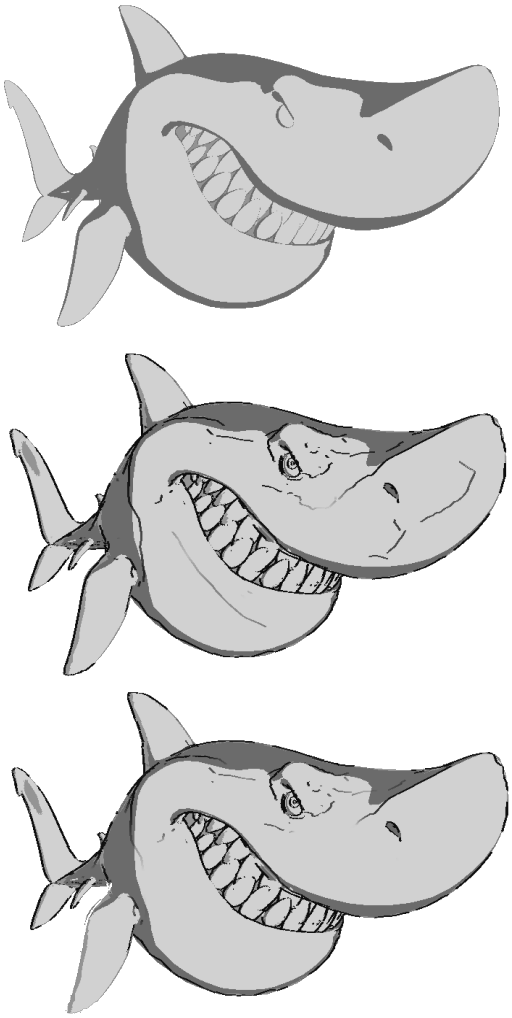

Fig. 4.8.: View dependent curvature threshold of 25 (middle) and 100 (bottom). The lower threshold results in more noise.

To determine whether two vertices are close enough to join, a threshold of screen-space distance is used. Apparent ridges near the edge of the model (contour lines) have a very noisy z-depth (Figure 4.10), but once rendered, they appear continuous.

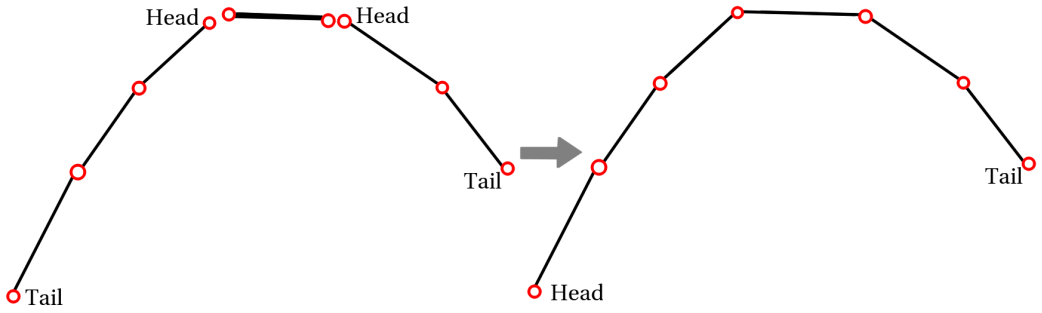

Fig. 4.9.: Example of a new line segment getting connected to two existing snakes, before (left) and after (right).

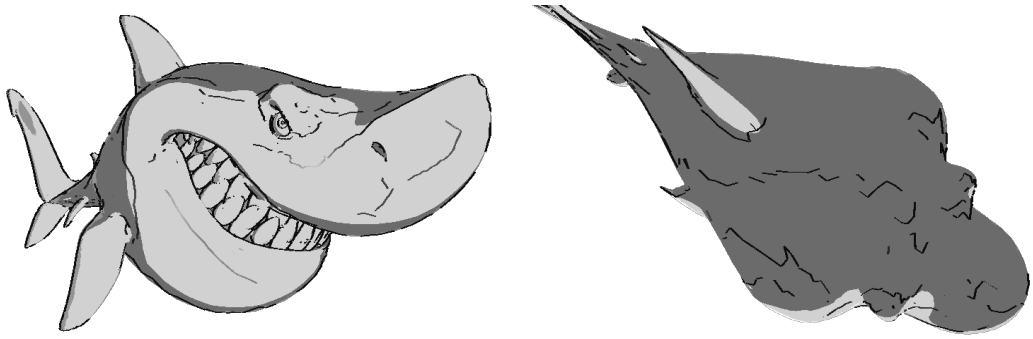

Fig. 4.10.: Top view of shark model with the same ridge lines as front view, where the contour line shows clear discontinuities in depth relative to the camera.

```
foreach (segment in vertexPairs)
{
  // look for an existing snake to attach to
  foreach (snake in snakes) if !snake.isClosed
    foreach (snakePart in {head, tail})
      if (distance(snake.snakePart, segment) < threshold)
        firstMatch = snakePart

  // connect to other snake:
  foreach (snake in snakes) if !snake.isClosed
    foreach (snakePart in {head, tail})
      if (distance(snake.snakePart, segment) < threshold)
        secondMatch = snakePart

  if (not firstMatch and not secondMatch)
    snakes.add(segment);

  if (firstMatch xor secondMatch)
    connectToSnake(firstMatch)

  if (firstMatch and secondMatch)
    combineSnakes(firstMatch, secondMatch)
    if (snake.isClosed)
      snakes.remove(snake);
}
```

Listing 4.4: High-level pseudocode for line clustering algorithm

After all snakes are found, Laplacian smoothing of snakes is done by settings each interior vertex to the average position of its two neighbours before smoothing.

Finally, the snakes are used to remove lines that are considered noise. Since Apparent ridges are using a discrete definition of curvature on a mesh with limited resolution, not all line segments it generates are part of a clear intented line stroke. Since noise line segments do not connect very well, snakes with a length smaller than 10 vertices are removed, since they are likely generated from noisy line segments.

Figure 4.11 shows the result of the line clustering algorithm.

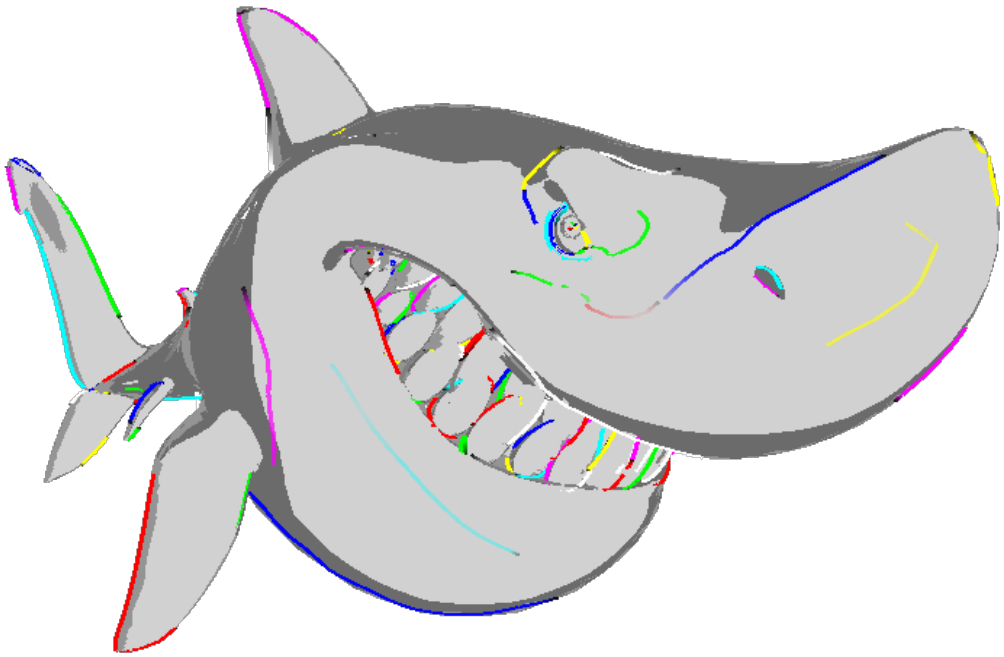

Fig. 4.11.: Result of line clustering. Different snakes are colored differently.

4.2.2 Improved contour lines

While view-dependent curvature reaches a maximum near contours, Apparent ridges sometimes fail to produce contour lines because of noise.

Hence, we derive contour lines independently by considering each edge L R and looking at the other vertex of the two adjacent triangles of the edge, T and B, as drawn in Figure 4.12.

A view vector is calculated by taking the difference of the vertex position and the camera position. To account for stereoscopic 3D, a center-eye camera is used, just like Apparent ridges.

If the dot product of the normal vector of V and the view vector has a different sign than the same dot product for B (i.e. one is positive, the other is negative), then one triangle is front-facing and the other back-facing, so the edge between L and R is considered a contour line and added to the list of line segments.

This results in more reliable contour lines than Apparent ridges provide. They are not considered for the line clustering algorithm, since there is a lot of redundancy in the line segments, making them not connect well due to the tesselation of the mesh, as shown in Figure 4.13. However, from a front view, they do not appear noisy, so they do not need to be smoothed and filtered like Apparent ridges.

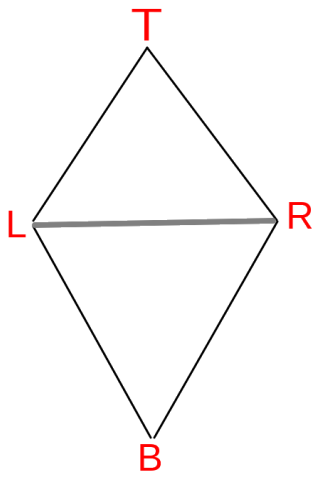

Fig. 4.12.: Vertex names used by the algorithm for contour line generation

Fig. 4.13.: Shark with Apparent ridges (left) and contour lines (right) generated from a front view, but seen from above

4.3 Local shading edits

While the optimization of the position of the light source tries to cover each area of interest, sometimes it is not possible to get a shading line on each of them. Take for example a sphere with dents on multiple sides like in Figure 4.7: the shading line tries to cover as many dents as possible, but because of the topology, this is not possible. Consequently, at least one of the holes does not have a favorable bright/dark ratio.

Without adding an additional light source, it is still possible to make improvements however: Todo et al. [Tod+07] describe a method of making local shading edits for toon-shading. Instead of using a single toon shading threshold for the entire mesh, this threshold can be modified per fragment.

To implement this, each vertex v gets a scalar property representing a value offset O_v, initially set to 0. We redefine our original diffuse equation 3.1 to a modified one (Equation 4.3).

$$I_d = k_d \times i_d \times (L * N + O_v) \tag{4.3}$$

The advantage of doing this on the mesh instead of in image space is that it is time-coherent: when the camera or model moves, each defined offset value remains at the same surface location.

Now that we have a way to locally modify the lighting, what we want to do is adjust the offset of each vertex O_v to improve segments that did not get a desirable bright/dark ratio from the solved light position. Optimal offsets can be obtained from a simple analysis in image space. First, we need to associate each vertex with a segment. As discussed in 4.5.3, we computed a screen-space texture with segment labels for each pixel, and the bright/dark ratio for the segment. Using the screen space coordinates of each vertex, vertices are projected onto this texture and assigned the corresponding segment.

To obtain the desired offset for each segment, the dot products of each fragment belonging to the segment are sorted in ascending order. Then the dot product of a desired percentile can be found, e.g., a desired brightness factor of 0.5 means we look at the 50% percentile, or the median. The offset is then the difference between the threshold used for discretizing the diffuse shading factor I_d and the dot product of the median is used. E.g. when the threshold for bright/dark is 0.6, and the 50% percentile of dot products is 0.4, then setting offset to 0.2 means that half of the fragment will be below the threshold and half will be above, resulting in 50% brightness.

Each segment's vertices are now assigned this optimal offset, which will give the segment its desired bright/dark ratio, unless the segment is not covered by vertices. This should only happen when the mesh resolution is very low, but since Apparent ridges require a dense mesh already, and a denser mesh can be produced with tessellation if needed, this is not problematic.

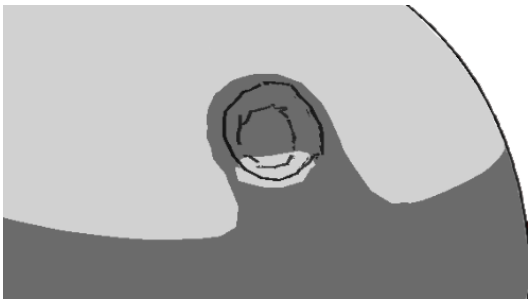

Fig. 4.14.: When local shading edits are pushed too far, it results in notably unphysical shading.

Fig. 4.15.: A harsh transition between shading offsets generates jarring shading lines, but blurring the offsets can mitigate this. Left: no blurring, middle: a little blurring, right: a lot of blurring.

While the added vertex offset perfectly solves the problem from the perspective of brightness constraints, it can result in aesthetic problems:

- A too-large offset will make the lighting look inconsistent (Figure 4.14). This is solved by defining an 'aggressiveness' variable that each offset gets multiplied by in the fragment shader. An aggressiveness of 0 means no effect, 1 means a perfect solution at the cost of artifacts, and anything inbetween is a compromise.

- A large difference in offset between adjacent segments results in jarring shading lines. This is solved by smoothing the offsets in the mesh: the offset of each vertex 'bleeds' to its neighbouring vertices (Figure 4.15). This shows another advantage of making offset a vertex property instead of doing it in

image space: when the offset is smoothed out in image space, it can cause leakage artifacts.

The result of this technique on a dented sphere is shown in Figure 4.16.

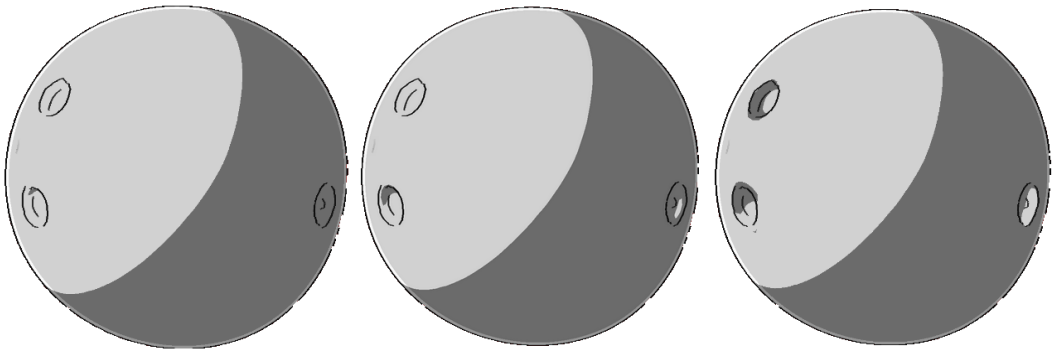

Fig. 4.16.: Demonstration of local shading edits effect of 'aggressiveness' parameter. Left: no local shading edits. Middle: moderate local shading edits (50%). Right: aggressive local shading edits (100%).

4.4 Distance field

With the light position in place, we can still improve depth perception by adding lines at places not covered by shading lines. In order to do this, there needs to be a metric identifying locations that are not covered. As discussed in Section 3.1, high-frequency details result in disparities that give depth cues. When such details are lacking, the depth is estimated from the surrounding context. How this is estimated by the human brain is up to perceptual studies outside the scope of this research, but as a first approximation, it is assumed that the 'perceived depth' of a pixel is equal to the depth of the closest disparity. For each pixel, we need to identify whether it provides a disparity, and if it does not, the closest pixel that does needs to be found.

An edge detection algorithm can be used to find pixels that give visual disparities. For example, the Sobel operator uses a small 3x3 kernel around each pixel to find sharp derivatives in the X or Y direction.

For the application of 3D toon-shading, two simplifications can be made:

- Color changes are discrete because of toon-shading. The pipeline used by the algorithm has no anti-aliasing or blurred shading line, so it suffices to simply look for any difference in RGB values between neighbouring pixels.

- Only sharp derivatives in the X direction are relevant. Since eyes are offset horizontally, perfectly straight horizontal edges do not provide disparities, as discussed in Section 4.1.

We define 'feature pixels' as pixels that have a difference in RGB values with their left or right pixel. We define the 'depth error' of a pixel to be the difference between its perceived and actual Z-coordinate in screen space.

Feature pixels are assumed to give perfect depth perception: their depth error is 0 since the actual depth is equal to the perceived depth. Non-feature pixels (such as the ones in the middle of a flat-colored region) have a known actual depth since they are rendered from a 3D mesh, but have a perceived depth based on the closest feature pixel (in any direction, using Euclidian distance). Computing the 'depth error' as defined above for each pixel results in Figure 4.17.

Fig. 4.17.: Example of depth metric on dented sphere mesh. Blue means the 'perceived depth' is larger than 'actual depth', orange means the 'perceived depth' is smaller than 'actual depth'.

This metric can then be used to add or remove lines in a region. Apparent ridges are ridges placed where view-dependent curvature is at a maximum, but only if it is large than a certain threshold. By making this threshold a function of the depth error, areas with few disparities from shading are more likely to have lines making up for it.

4.4.1 Computing the distance field

We begin by creating a texture which is used as a feature map, where each pixel has a value 1 if it differs in color from its left or right pixel. On this texture, a distance transform is applied with an Euclidian distance metric. A naive distance transform algorithm would do the following for every pixel p:

- If it is a feature pixel, set the depth error of p to 0.
- If it is not a feature pixel, set a 'minimum distance' variable to ∞. Then, for each pixel q, check if it is a feature pixel closer than 'minimum distance'. If so, update the minimum distance of p and set the depth error of p to be d_q - d_p.

For an image of $N * M$ pixels, the naive algorithm has a time complexity of $O(N^2 * M^2)$. While real-time performance is not required for our application, such an algorithm would take a long time to compute on moderately sized images. A faster algorithm for a distance transform has been created by Meijster et al. [MRH02]. This algorithm runs in linear time and is easy to parallelize, so for an image of $N * M$ pixels and for p processors, it runs in $O(N * M/p)$.

Meijster's algorithm has two passes: first a pass on the columns of the image, and secondly a pass on the rows of the image. The row pass is more complex than the column pass, and needs to be adjusted for our purposes: Instead of storing the distance to the closest feature, it needs to store the difference in Z-coordinate.

However, Meijster's algorithm is not straightforward to implement, and some sources [ZF17] suggest it only becomes faster for images exceeding a specific resolution (in their case $2048 * 2048$ pixels).

Since the real-time performance is not a requirement, a simplified version of the row pass is used, resulting in a time complexity of $O(N * M^2/p)$. It is implemented using an OpenGL compute shader.

The metric can also be used to evaluate how well the shape of a toon-shaded render is perceived: The depth error can be aggregated over the whole image using mean squared error. This metric is then to be minimized.

4.5 Implementation details

4.5.1 Test application

An application was made to try out and test the techniques described in the above sections. The implementation uses OpenGL, though it should be adaptable to any flexible graphics pipeline. It renders stereoscopic 3D by rendering two views using the off-axis projection method and combining them into a red-cyan anaglyph image (see Section 3.1).

Basic features of the model are:

- Load a WaveFront .obj model
- Compute curvature data for the mesh
- Display Apparent ridges
- View the mesh by moving the camera and light source
- Apply the discussed algorithms as described below

Input mesh

Triangle meshes are imported from a WaveFront .obj file [1]. The file is expected to contain a manifold triangle mesh with a position and normal vector for each vertex. This is needed because the discrete curvature can only be computed for a manifold mesh.

The discrete curvature for each vertex is calculated using a method by Rusinkiewicz [Rus04]. This computes the two principal curvature constants and the principal curvature directions.

The Gaussian curvature for each vertex is computed by multiplying the two principal curvature values, $k_1 * k_2$. A smoothing pass is applied where each vertex gets assigned the weighted average of the Gaussian curvature of its neighbouring vertices, with the weight based on the Voronoi area. This is done because on certain meshes, it was found that the discrete curvature is very noisy (Figure 4.18). The segmentation algorithm would generate a lot of false segments when it receives noisy input data.

[1] https://en.wikipedia.org/wiki/Wavefront_.obj_file

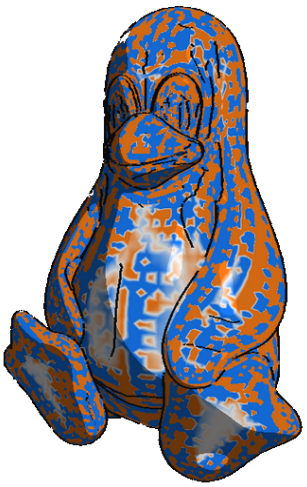

Fig. 4.18.: On certain meshes, the computed discrete Gaussian curvature has a lot of noise

4.5.2 Shaders

Deferred shading [2] is used with three G-buffers: Two G-buffers are used for rendering the scene for the left and right eye, and a third one is used for the light positioning algorithm.

The left and right eye use an off-axis camera transformation matrix.

The preview G-buffer encodes the following pixel information:

- albedo color
- position
- normal

These outputs are put onto textures that are used as inputs for a shader that computes the shading in a second pass. The two images are then combined into an anaglyph image by multiplying the RGB values of the pixels in the left image with $(1, 0, 0)$ (red) and, the pixels in the right image with $(0, 1, 1)$ (cyan).

The third G-buffer encodes the following pixel information:

- Gaussian curvature, interpolated from vertex values.
- depth, the screen-space Z-coordinate of the fragment.
- diffuse shading derivative

[2] https://en.wikipedia.org/wiki/Deferred_shading

- diffuse shading dot product, $L * N$

The first two are used by the connected component labelling algorithm, while the last two are used for computing the light source vector. The camera position when rendering to this G-buffer is in the middle of the two eyes.

The diffuse shading derivative is computed in GLSL according to Listing 4.5.

```
vec3 brightnessGradient(vec3 l, vec3 n)
{
  return (-dot(l, n) / dot(l, l) * l + n) / sqrt(dot(l, l));
}
```

Listing 4.5: GLSL shader code computing of diffuse shading derivative.

4.5.3 Connected component labelling

Connected component labelling is done using the Hoshen-Kopelman Algorithm [HK76].

An image is created where each pixel has an integer identifying the component it belongs to. The integer is an index of a pixel, and we assume the image has row-major ordering and the top-left pixel has index 0. A union-find data structure is used to efficiently create and merge components. Each component has a 'root' pixel, which is the pixel with the lowest index in that component, and it is representative of it. Every other pixel either points directly to the root pixel, or to another pixel that transitively points to the root. The 'find' function finds the root pixel of any pixel by visiting its parent until a pixel is its own parent, and updates links it traverse along the way for efficiency. The 'union' function combines two segments by making one segment's root be a child of the other segment's root. Pseudocode for these functions is in Listing 4.6.

```
int find(start) {
  root = start
  while (labels[root] != root) {
    root = labels[root]
  }
  while (labels[start] != start) {
    swap(labels[start], root)
  }
  return start
}

int union(i0, i1) {
  rootHi = find(i0)
  rootLo = find(i1)
  // the root has the lowest index of the whole set it represents.
  if (rootHi < rootLo) {
    swap(rootHi, rootLo)
  }
  labels[rootHi] = rootLo
  return rootLo
}
```

Listing 4.6: Pseudocode of 'union' and 'find' operations in the disjount set data structure

With these helper functions, the image can now be scanned for segments. All pixels are traversed from top to bottom and left to right, and assigned a label based on its neighbours up and to the left, which have already been traversed. When the pixel is unique from its neighbours, a new group is created by making its index equal to itself. When it is similar only to its neighbour to the left or its neighbour above, it joins that segment using the 'find' function. When it is similar to both neighbours, both are merged into one segment using the 'union' function. See Listing 4.7 for pseudo code.

```
left = the pixel belongs to the same group as the one on the left
up = the pixel belongs to the same group as the one above
if (!up && !left) {
  labels[i] = i;
} else if (up && !left) {
  labels[i] = find(labels, upIndex)
} else if (!up && left) {
  labels[i] = find(labels, leftIndex)
} else if (up && left) {
  labels[i] = union(leftIndex, upIndex)
}
```

Listing 4.7: Core of Hoshen-Kopelman algorithm

Unlike the original Hoshen-Kopelman algorithm, diagonal neighbouring pixels are also considered. This is done to reduce the number of 1x1 pixel components that appear because of aliasing. Additionally, segments with a size under a threshold of 64 pixels will be deleted and considered transparent, since such segments are likely noise instead of interesting areas.

There are two edge cases:

- Pixels in the top-most row or left-most column have no pixel above or to the left respectively. Pixels out of bounds are assumed to be not part of the same component.
- Transparent pixels are skipped over, and a transparent pixel as a neighbour is not considered as part of the same component.

After running the algorithm, we now have labels for each segment, but because each label corresponds to the index of a pixel, the indices are sparse. To make them easier to use and store in a texture, each label is renamed to the lowest number that is not yet used as a label. For example, instead of 3 segments with labels $(201, 434, 640)$, we now have labels $(0, 1, 2)$.

The algorithm is implemented on the CPU, so the depth and curve data of the fragment are downloaded from the GPU texture and then used to distinguish segments. The output is a texture of labels: for each fragment, the segment it belongs to is stored as an index. It also outputs the sizes of each segment.

Using the obtained indices, the diffuse shading derivative is summed per segment and multipled by a weight based on the size of the segment.

Also, the bright/dark ratio of each segment is calculated using the 'diffuse shading dot product' output texture. With these two results combined, a vector displacing the light source for each segment is computed.

These vectors are than averaged, and the resulting vector is used to move the light each iteration in the light optimization algorithm (Section 4.1).

5 Results

The algorithms as described in Section 4.1 have been applied to a range of test-models. The test models were obtained from blendswap [1] and have a 'Creative Commons Attribution 3.0' license. The individual attribution can be found in the appendix A.

The selection of models was based on whether they are suitable for toon-shading, and whether they are varied among each other. The models all have curved surfaces, because a model with only flat planes (such as a box) does not provide a shading line that can be smoothly influenced by light position.

The models have been scaled to roughly the same size, such that they roughly fit in a sphere of radius 1. Subdivision modifiers (Catmull-Clark subdivision) have been applied for sufficient resolution, resulting in triangles counts between 10000 and 500000. Vertex positions and normal vectors were not manually tweaked.

5.0.1 Models

The asteroid model (Figure 5.1) has many bumps and holes, resulting in many different segments that the algorithm needs to cover.

The 'mike' model (Figure 5.2) is a typical stylized human character.

The Moped is a vehicle with both small parts (such as the handle bars) and large parts (such as the wheels).

The robot (Figure 5.4) has many cylindrical parts, unlike 'Mike' who has limbs with a more organic shape.

The snake model (Figure 5.5) is mostly a single curve, with some finer details on the head.

The dented sphere model (Figure 5.6) highlights the effect of local shading edits, with a few distinct far apart dents on an otherwise perfect sphere.

[1] https://blendswap.com

Model	Triangles	Segments	Time to optimize
asteroid	43360	49	0.7s
mike	338208	39	1.7s
moped	65616	30	0.8s
robot	110768	27	0.9s
snake	13760	19	0.7s
dented sphere	13056	3	0.8s
shark	30796	20	0.7s

Tab. 5.1.: Statistics about the used test models

The shark model (Figure 5.7) has a large body with distinct curvature and a few smaller details like fins and teeth.

Statistics about each model can be found in Table 5.1.

5.0.2 Convergence

The combined dot product derivative for all segment has a direction, but not a meaningful magnitude. A small step size gives high accuracy but requires many iterations to reach an optimum point, while a large step size will overshoot its final target when it gets near. Therefore, the step size used to adjust the light position is given a relatively large magnitude to start with, and each iteration it gets reduced by multiplying it with a factor below 1.

Based on experimentation, an initial step size of $1/8$ that gets multiplied by $31/32$ each iteration for 64 iterations turned out to make the light position converge to an optimum on the test models.

As discussed, the size of each segment influences the weight of its desired light direction vector. On top of that, the bright/dark ratio also influences it: a segment that already has a desired bright/dark ratio does not need to move the light, resulting in a weight of 0. While the segment size remains constant throughout iterations, the bright/dark ratio will vary, so the weight can oscillate between 0 and a constant based on segment size.

Figure 5.8 shows how this results in oscillations in the optimization, because moving the light in favor of one segment's light/dark ratio can cause another segment to lose its wanted light/dark ratio.

Instead of a discrete cut-off between a weight of 0 and a constant, the weight can also be interpolated based on how far the bright/dark ratio is from the desired range.

The transfer function with fall-off mitigates this mostly, and even results in certain segments ending up closer to the desired ratio than the discrete transfer function, as shown in the same figure.

The results could be further improved with a transfer function that is not piecewise linear but e.g. quadratic, though this has not been explored yet.

5.0.3 Parameters

As a starting position, the light source has been placed on the camera, making the models front-lit and mostly bright. This has been done to make the results reproducible. An artist can choose a starting light position to guide the optimization to a lighting direction to their liking. However, the final light position is also dependent on the parameters of the optimization algorithm. The two major parameters will be discussed in this section.

Segment weight

The weight that each segment contributes to the light position each iteration is a function of the segment size in pixels. Each segment could be considered equally important (constant weight), but it can also be reasoned that larger segments are more important since they take up more of the image. The result of using different functions is shown in Figure 5.9.

On many models, the difference is negligible, but the asteroid and dented sphere are exceptions. In those cases, a proportional weight puts more emphasis on solving the larger sphere as a whole than the smaller features on top of the sphere, which have a small segment size. The dent on the dented sphere is poorly lit with proportional weight, but with constant weight it is lit well since it is considerd just as important as the rest of the sphere.

Acceptable brightness ratio range

Another parameter is the range of tolerated brightness ratio. Instead of optimizing for an exact 50/50 bright/dark ratio, a wider window can be used, where a segment is allowed to have any ratio that is more than 20% and less than 80% bright. Figure 5.10 shows the impact of the wideness of this interval on the asteroid model.

The interval has a lot of influence on the resulting shading direction: With the most lenient interval, 20-80% the initial light position is already sufficient for most segments, so little modification of the light position is needed. For the stricter inverval, 35-65%, the initial light position is far from sufficient, so it finds an optimum by moving the light to the left. For the strictest ratio, desiring an exact 50%, the optimum on the left is not good enough, so it converges on a better optimum where the light comes from above.

The lenient 20-80% interval yields a subobtimal result, where many dents are less pronounced since they are allowed to stay at a near 80% brightness from the front lighting, but it does respect the initial lighting position more. Contrast this with the strict 50% interval, which yields a better result but gives less control to the artist: different initial positions can easily converge to a single point where the strict criteria are met. The medium interval is the recommended compromise where the found optimum gives decent stereo correspondence, while still being flexible in allowing multiple light positions as solutions based on the starting position chosen by the artist.

5.0.4 Ridges

Figure 5.12 shows the shark model with contour lines and Apparent ridges. Most lines are contour lines, but on the snout an apparent ridge is drawn. The natural color of the shark complements the shading line very well, since it also provides a contour in a different place than the shading line. This is not something the algorithm takes into account when optimizing the light position however, so as a future improvement, the algorithm could ensure the resulting shading and natural color of the model complement each other in all cases.

Figure 5.11 shows ‘mike’ with contour lines and Apparent ridges. In this case the Apparent ridges do not add much more than contour lines, since most body parts are simple cylinder-like shape with consistent curvature, giving few significant maxima in view-dependent curvature. Unfortunately, there are only few places where lines can reasonably be added without making artistic choices.

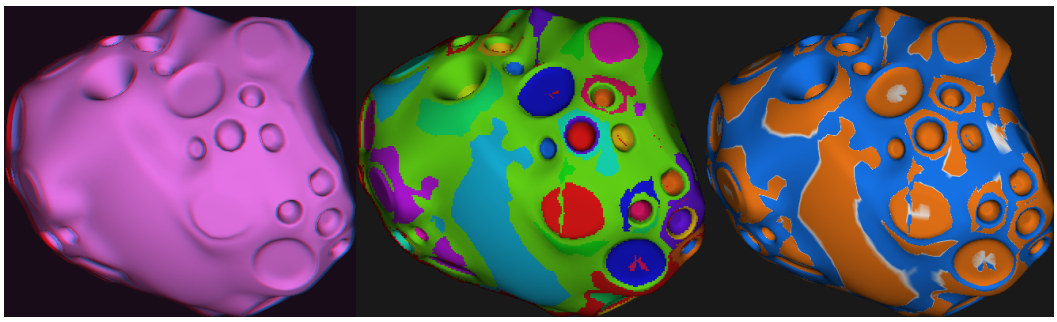

Fig. 5.1.: Model ‘asteroid’. Left: rendered with phong shading, for reference. Middle: segmentation, 49 segments. Right: Gaussian curvature, orange is positive, blue is negative.

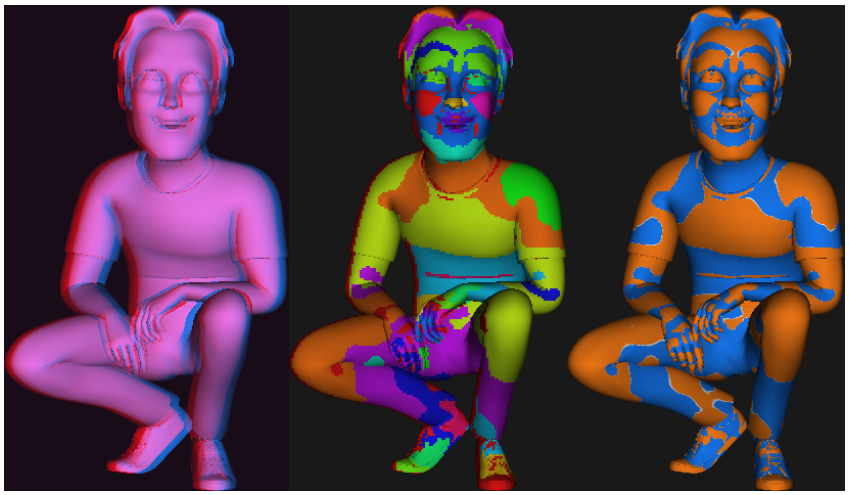

Fig. 5.2.: Model ‘mike’. Left: rendered with phong shading, for reference. Middle: segmentation, 39 segments. Right: Gaussian curvature, orange is positive, blue is negative.

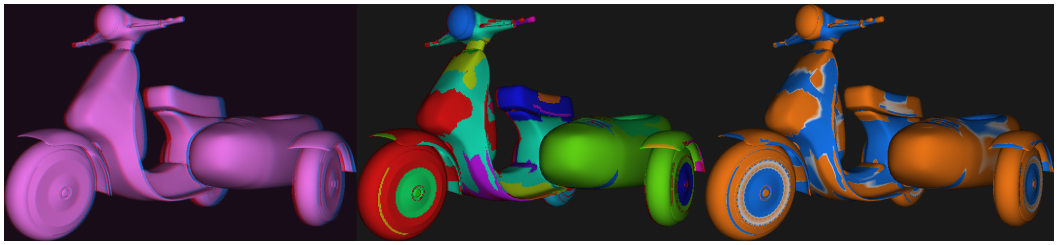

Fig. 5.3.: Model ‘moped’. Left: rendered with phong shading, for reference. Middle: segmentation, 30 segments. Right: Gaussian curvature, orange is positive, blue is negative.

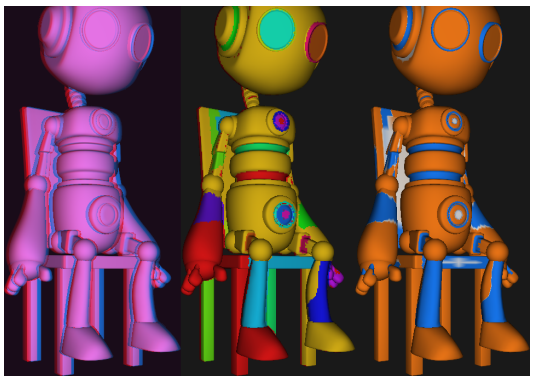

Fig. 5.4.: Model ‘robot’. Left: rendered with phong shading, for reference. Middle: segmentation, 27 segments. Right: Gaussian curvature, orange is positive, blue is negative.

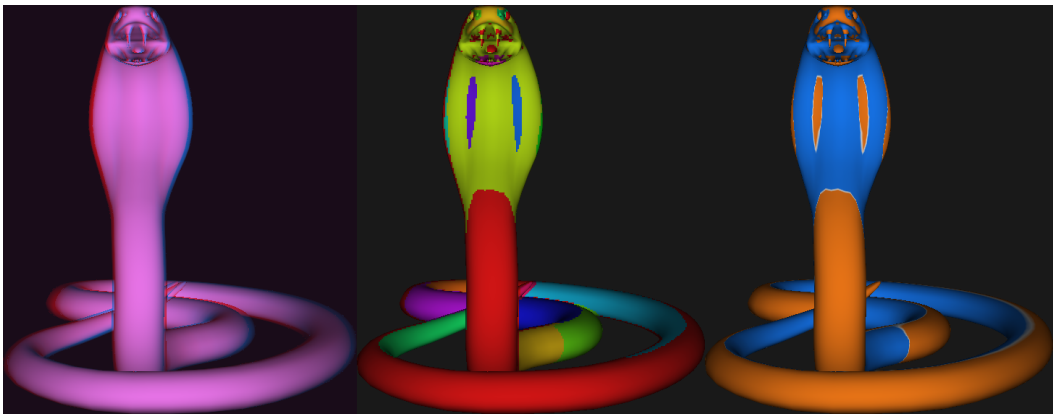

Fig. 5.5.: Model ‘snake’. Left: rendered with phong shading, for reference. Middle: segmentation, 19 segments. Right: Gaussian curvature, orange is positive, blue is negative.

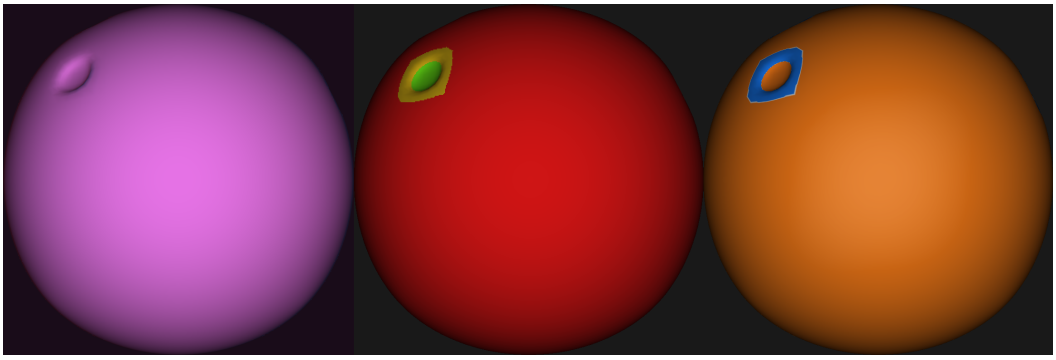

Fig. 5.6.: Model ‘dented sphere’. Left: rendered with phong shading, for reference. Middle: segmentation, 3 segments. Right: Gaussian curvature, orange is positive, blue is negative.

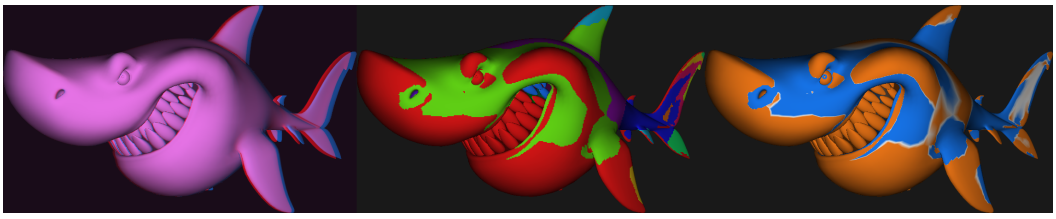

Fig. 5.7.: Model ‘shark’. Left: rendered with phong shading, for reference. Middle: segmentation, 20 segments. Right: Gaussian curvature, orange is positive, blue is negative.

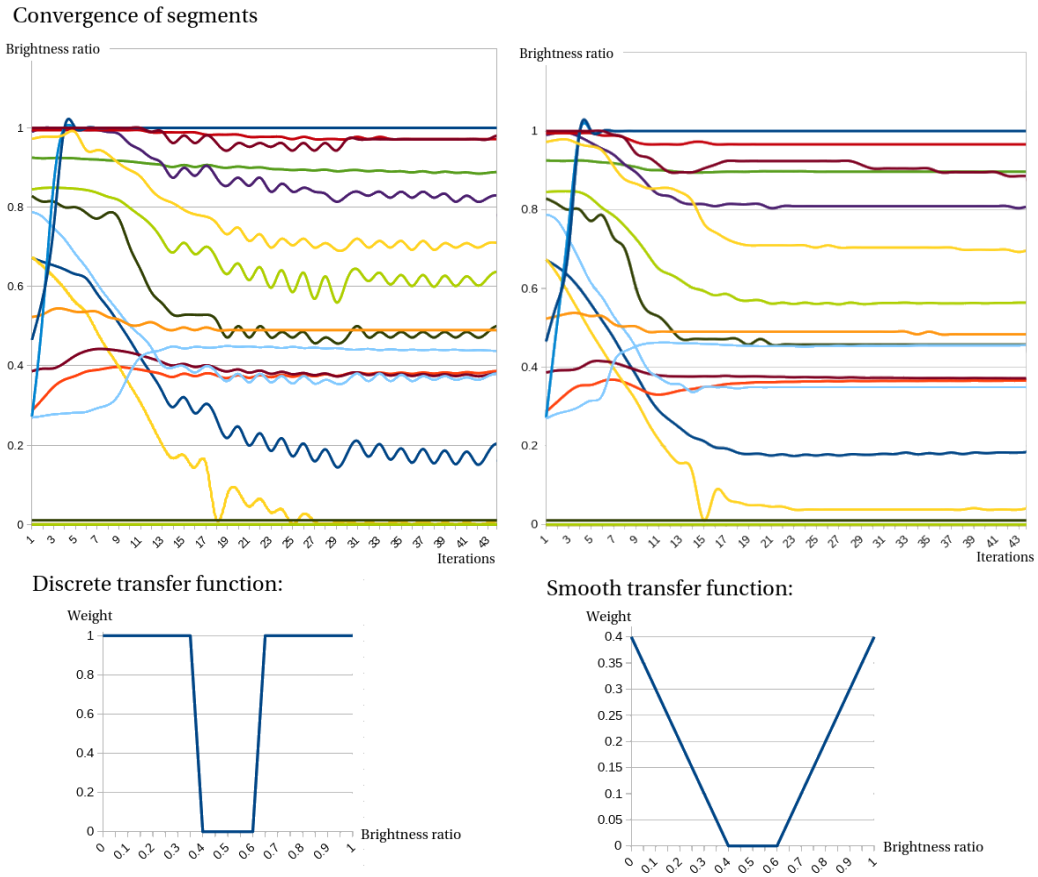

Fig. 5.8.: Comparison of convergence of segments to their optimal brightness ratio. Left: a discrete transfer function between bright/dark ratio and segment weight results in oscillations as segments switch between 0% and 100% influence between iterations. Right: a linear fall-off based on distance to the segment's desired bright-dark ratio results in smoother convergence.

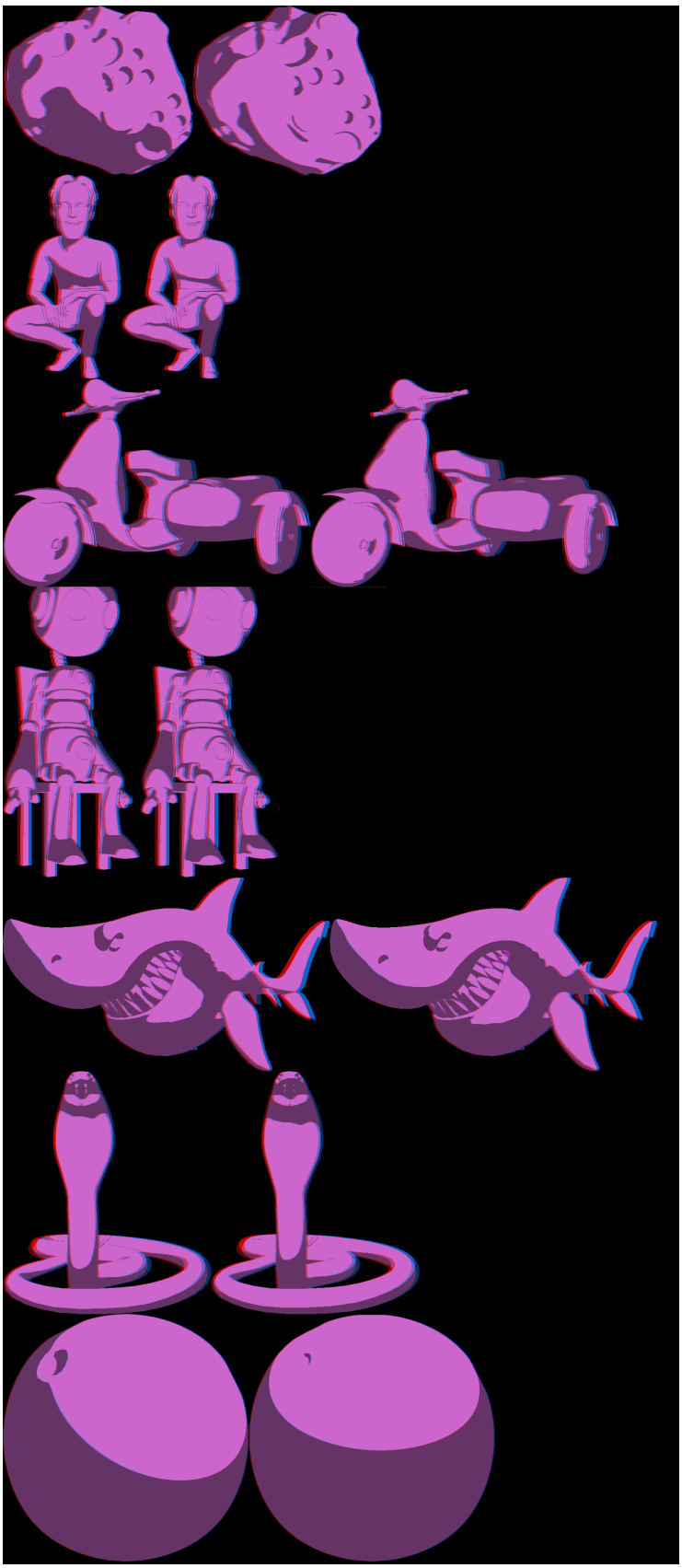

Fig. 5.9.: Effect of segment weight on final light position. Left: Constant weight, Right: Weight proportional to segment area.

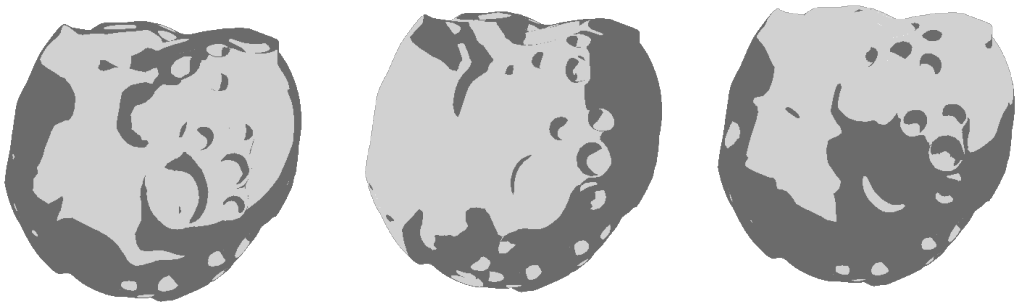

Fig. 5.10.: Comparison of the effect of varying the acceptable light-dark ratio range. Left: lenient (0.2, 0.8), Middle: medium (0.375, 6.125), High: strict (0.5, 0.5).

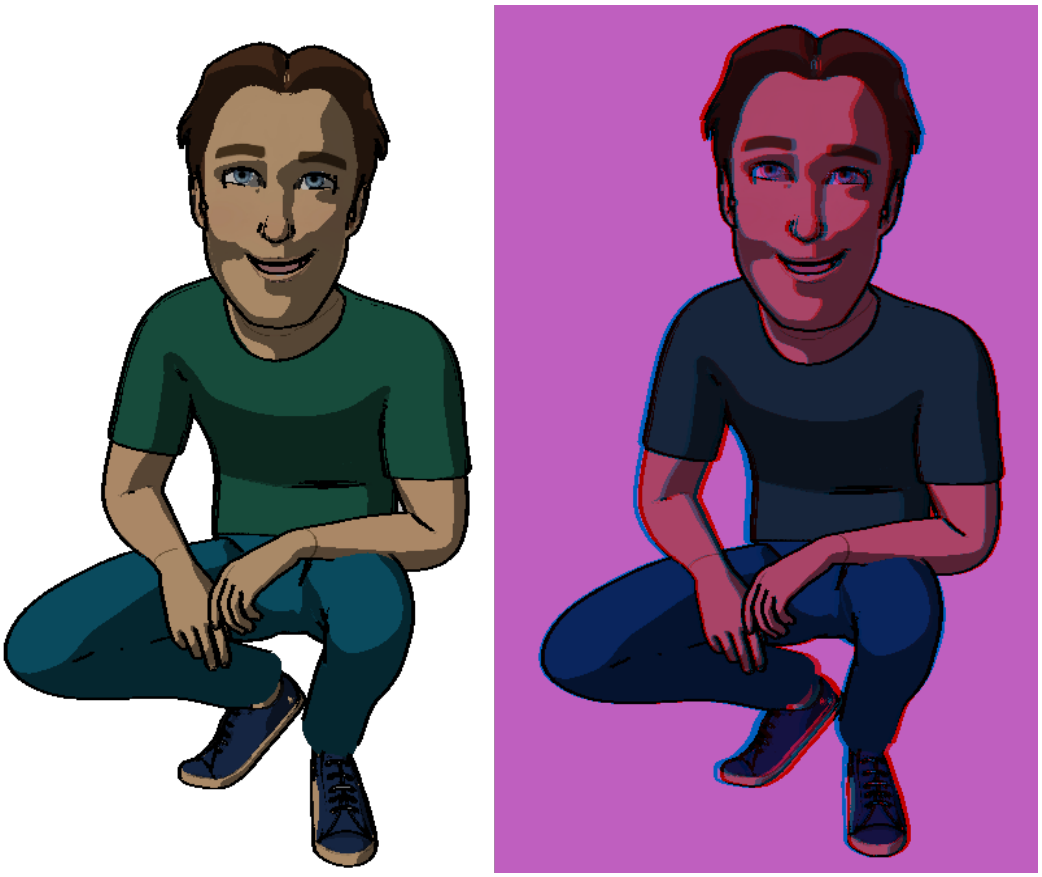

Fig. 5.11.: Mike model with color, 2D (left) and anaglyph 3D (right).

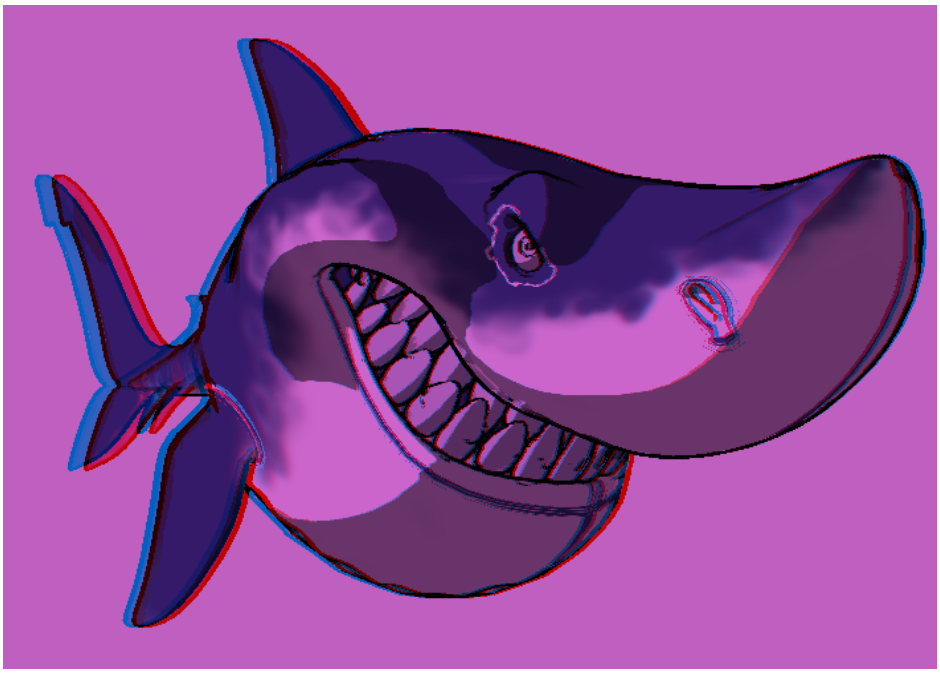

Fig. 5.12.: Shark with contour lines, Apparent ridges, and a color texture

Conclusion

6

6.1 Conclusion

We looked at ways to mitigate the lack of stereo-correspondence in toon shaded renders, resulting in poor depth perception when viewing in stereoscopic 3D. The flat aesthetic of toon shading gives the human visual system few points of contrast which are needed to estimate depth by finding the disparity between the left image and right image. The shading line created by the light source, dividing the model in 'bright' and 'dark' parts does provide a stereo-correspondence, as do the contour lines.

Instead of changing the geometry of the scene, we focused on adjusting the shading generated by a light source. We identified that on a sphere-like surface, the best depth perception is achieved when the shading line divides the sphere in roughly equal 'bright' and 'dark' parts. The sign of Gaussian curvature can be used to segment the image into parts that are sphere-like.

The derivative of the brightness (assuming Lambertian diffuse shading) with respect to the light position given a camera position can be computed for each pixel, and averaged per segment. The vectors of segments can be combined into a single vector that moves the light in a direction optimizing the depth perception of all segments simultaneously.

When the light cannot cover all segments, a segment can have a constant offset to the brightness to locally edit the shading line. This can result in jarring shading disrepancies, but when the offset is not too large and is smooth over the mesh, the result can be an improvement. By projecting the offset onto mesh vertices, the mesh and camera can freely be rotated maintaining a time-coherent brightness offset on the model. Segmentation and optimization is not time-coherent, but can be performed at a start- and end point and then interpolated.

An implementation is provided that uses the GPU for real-time performance for most components of the algorithm. Only the light optimization is not real-time, taking a few seconds per scene.

6.2 Future work

Multiple light sources

This work always assumed a single point light source and Lambertian diffuse shading. While multiple light sources is very uncommon in toon-shading, since it goes against the simplistic nature, it is possible to have different light sources that affect different parts of the scene differently, as an alternative to the local brightness offset per vertex.

Temporal domain

The brightness derivative can be used to create 'light sweeps', which give every part of the model a stereo correspondence at at least one point at a time. Other than this, the temporal domain remains unexplored in this thesis, but can possibly be used in more contexts.

Segmentation

The segmentation based on Gaussian curvature, while simple and effective, is not as good as manual segment selection. Sometimes two separate segments are accidentally connected, or segments are separated based on noise in the tesselation or curvature computation. This can be improved by using a more clever segmentation algorithm, which could split segments that have a large circumference to area ratio for example. Alternatively, a brush tool can be provided that allows artists to manually select segments.

Additionally, the assumption is made that all segments are equally important. Usually a scene has a focal point that the artists deliberately choses, and accentuates using composition techniques. Letting an artist also specify focal points could guide the optimization to more pleasant results.

Depth metric

The perceived depth metric makes several assumptions that have not been thoroughly tested. A perceptual study can help clarify how accurate this is. An improved depth metric can make it easier to evaluate the effectiveness of light optimization techniques without needing to manually test them.

Specular lighting

The specular highlight is a point of contrast with points that provide stereo correspondence, thus beneficial for depth perception. This has not been taken into consideration by the algorithm yet. An interesting improvement could be to locally modify specular highlights for better depth perception.

Bibliography

[BTM06] Pascal Barla, Joëlle Thollot, and Lee Markosian. “X-Toon: An Extended Toon Shader”. In: *Proceedings of the 4th International Symposium on Non-Photorealistic Animation and Rendering*. NPAR ’06. Annecy, France: Association for Computing Machinery, 2006, pp. 127–132 (cit. on p. 6).

[Bez+08] Hedlena Bezerra, Elmar Eisemann, Xavier Decoret, and Joelle Thollot. “3D Dynamic Grouping for Guided Stylization”. In: *NPAR 2008: Proceedings of the 6th International Symposium on Non-photorealistic Animation and Rendering*. ACM, 2008, pp. 89–95 (cit. on p. 6).

[BSL18] Dennis R. Bukenberger, Katharina Schwarz, and Hendrik P. A. Lensch. “Stereo-Consistent Contours in Object Space”. In: *Computer Graphics Forum* 37.1 (2018), pp. 301–312. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13291` (cit. on p. 6).

[Col+08] Forrester Cole, Aleksey Golovinskiy, Alex Limpaecher, et al. “Where Do People Draw Lines?” In: *ACM Transactions on Graphics (Proc. SIGGRAPH)* 27.3 (Aug. 2008) (cit. on p. 14).

[CV95] James Cutting and Peter Vishton. “Perceiving layout and knowing distances: The interaction, relative potency, and contextual use of different information about depth”. In: vol. 5. Jan. 1995, pp. 69–177 (cit. on pp. 8, 9).

[Dąb+14] Łukasz Dąbała, Petr Kellnhofer, T. Ritschel, et al. “Manipulating refractive and reflective binocular disparity”. In: *Computer Graphics Forum* 33 (May 2014) (cit. on p. 5).

[Dec+03] Doug Decarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. “Suggestive contours for conveying shape”. In: *ACM Transactions on Graphics* 22.3 (2003), pp. 848–855 (cit. on pp. 6, 13).

[Dec96] Philippe Decaudin. *Cartoon-Looking Rendering of 3D-Scenes*. Tech. rep. 1996 (cit. on pp. 6, 13).

[Did+11] Piotr Didyk, Tobias Ritschel, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel. “A Perceptual Model for Disparity”. In: *ACM Trans. Graph.* 30 (July 2011), p. 96 (cit. on p. 5).

[Du+13] Song-Pei Du, Belen Masia, Shi-Min Hu, and Diego Gutierrez. “A Metric of Visual Comfort for Stereoscopic Motion”. In: *ACM Trans. Graph.* 32.6 (Nov. 2013) (cit. on p. 5).

[HWB19] Dejing He, Rui Wang, and Hujun Bao. “Real-Time Rendering of Stereo-Consistent Contours”. In: *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*. 2019, pp. 81–87 (cit. on p. 6).

[HK76] J. Hoshen and R. Kopelman. “Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm”. In: *Phys. Rev. B* 14 (8 Oct. 1976), pp. 3438–3445 (cit. on p. 38).

[JDA07] Tilke Judd, Frédo Durand, and Edward H. Adelson. “Apparent ridges for line drawing”. In: *ACM Trans. Graph.* 26.3 (2007), p. 19 (cit. on pp. 6, 13, 25).

[Kel+16] Petr Kellnhofer, Piotr Didyk, Karol Myszkowski, et al. “GazeStereo3D: Seamless Disparity Manipulations”. In: *ACM Transactions on Graphics (Proc. SIGGRAPH)* 35.4 (2016) (cit. on p. 5).

[Kim+13] Yongjin Kim, Yunjin Lee, Henry Kang, and Seungyong Lee. “Stereoscopic 3D Line Drawing”. In: *ACM Trans. Graph.* 32.4 (July 2013) (cit. on p. 6).

[Lan+10] Manuel Lang, Alexander Sorkine-Hornung, Oliver Wang, et al. “Nonlinear Disparity Mapping for Stereoscopic 3D”. In: *ACM Trans. Graph.* 29 (July 2010) (cit. on p. 5).

[MRH02] A. Meijster, Jos Roerdink, and Wim Hesselink. “A General Algorithm For Computing Distance Transforms In Linear Time”. In: Feb. 2002, pp. 331–340 (cit. on p. 35).

[NAK12] Lesley Northam, Paul Asente, and Craig S. Kaplan. “Consistent Stylization and Painterly Rendering of Stereoscopic 3D Images”. In: *International Symposium on Non-Photorealistic Animation and Rendering*. Ed. by Paul Asente and Cindy Grimm. The Eurographics Association, 2012 (cit. on p. 6).

[OBS04] Yutaka Ohtake, Alexander Belyaev, and Hans-Peter Seidel. “Ridge-Valley Lines on Meshes via Implicit Surface Fitting”. In: *ACM SIGGRAPH 2004 Papers*. SIGGRAPH ’04. Los Angeles, California: Association for Computing Machinery, 2004, pp. 609–612 (cit. on pp. 6, 13).

[Rus04] Szymon Rusinkiewicz. “Estimating Curvatures and Their Derivatives on Triangle Meshes”. In: *Symposium on 3D Data Processing, Visualization, and Transmission*. Sept. 2004 (cit. on p. 36).

[SBE18] Leonardo Scandolo, Pablo Bauszat, and Elmar Eisemann. “Gradient-Guided Local Disparity Editing”. In: *Computer Graphics Forum* 37.7 (2018) (cit. on p. 5).

[SBE22] Peiteng Shi, Markus Billeter, and Elmar Eisemann. “Stereo-consistent screen-space ambient occlusion”. In: *I3D 2022: ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. 2022 (cit. on p. 6).

[SEB19] Sergej Stoppel, Magnus Paulson Erga, and Stefan Bruckner. “Firefly: Virtual Illumination Drones for Interactive Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 25 (2019), pp. 1204–1213 (cit. on p. 5).

[Tem+12] Krzysztof Templin, Piotr Didyk, Tobias Ritschel, Karol Myszkowski, and Hans-Peter Seidel. “Highlight Microdisparity for Improved Gloss Depiction”. In: *ACM Transactions on Graphics - TOG* 31 (July 2012) (cit. on p. 5).

[Tho+16] William Thompson, Roland Fleming, Sarah Creem-Regehr, and Jeanine Stefanucci. *Visual Perception from a Computer Graphics Perspective*. Apr. 2016, pp. 1–518 (cit. on p. 10).

[Tod+07] Hideki Todo, Ken Ichi Anjyo, William Baxter, and Takeo Igarashi. "Locally controllable stylized shading". In: *ACM Transactions on Graphics* 26.3 (2007) (cit. on pp. 6, 31).

[THA15] Robert Toth, Jon Hasselgren, and Tomas Akenine-Möller. "Perception of Highlight Disparity at a Distance in Consumer Head-Mounted Displays". In: *Proceedings of the 7th Conference on High-Performance Graphics*. HPG '15. Los Angeles, California: Association for Computing Machinery, 2015, pp. 61–66 (cit. on p. 5).

[Win06] Holger Winnemoller. "Perceptually-motivated non-photorealistic graphics". In: *ProQuest Dissertations and Theses* December (2006), p. 208 (cit. on p. 2).

[ZF17] Francisco Zampirolli and Leonardo Filipe. "A Fast CUDA-Based Implementation for the Euclidean Distance Transform". In: July 2017, pp. 815–818 (cit. on p. 35).

List of Figures

List of Tables

List of Listings

Appendix A

A.1 Model credits

Models used to test the algorithm are:

- "Rigged Cartoon Moped - NPR" by JonasDichelle, https://blendswap.com/blend/11614
- "Monster Cobra" by Calore, https://blendswap.com/blend/10648
- "Mike Rig" by lucky3d1, https://blendswap.com/blend/22596
- "Toon Asteroid" by MASALART, https://blendswap.com/blend/18750
- "Linux mascot Tux" by Vido89, https://blendswap.com/blend/23774
- "Shark Cartoon - Animation Ready" by mortysanchez, https://blendswap.com/blend/22094