

VARIABLE STRETCH TEXTURES
ON THE GPU

by

Emil Geisler

A Senior Honors Thesis Submitted to the Faculty of
The University of Utah
In Partial Fulfillment of the Requirements for the

Honors Degree in Bachelor of Science

In

Computer Science

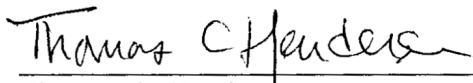
Approved:



Cem Yuksel
Thesis Faculty Supervisor



Mary Hall
Director, School of Computing



Thomas C. Henderson
Honors Faculty Advisor

Sylvia D. Torti, PhD
Dean, Honors College

April 2023

Copyright © 2023

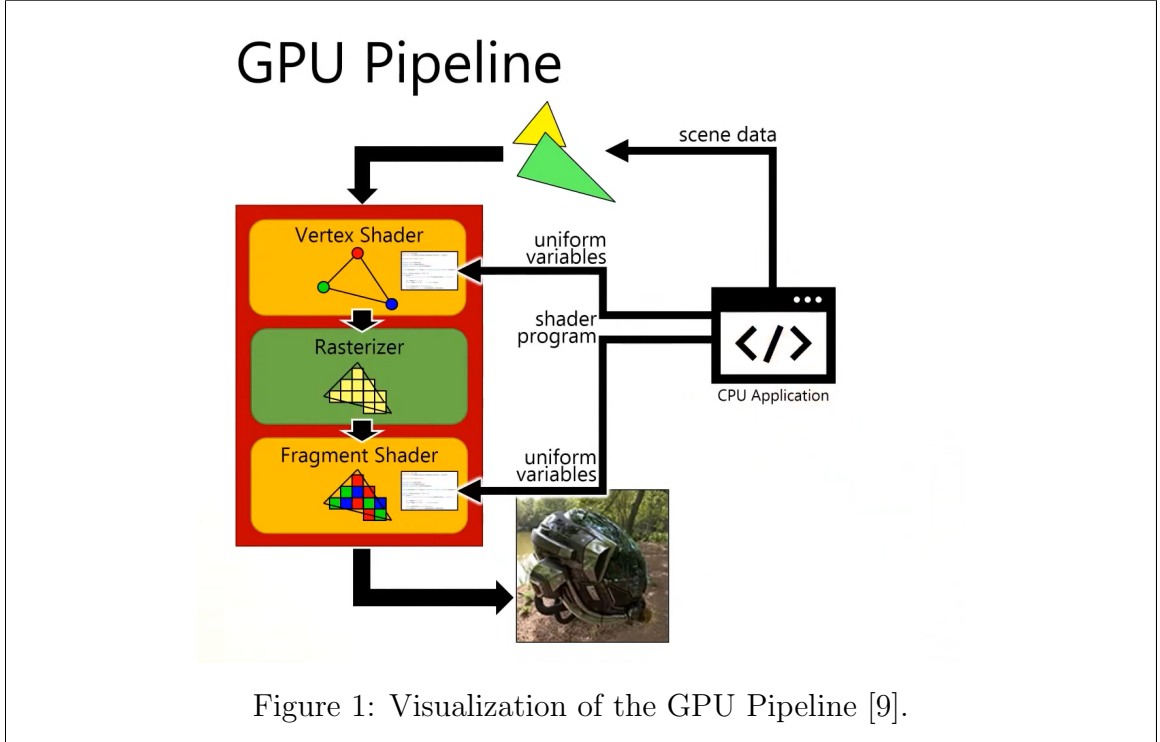
All Rights Reserved

1 Abstract

In computer graphics, a texture is an image that is overlaid on a triangle when the object is rendered. Textures allow for objects to have more complicated appearances than just a single color across the triangle. Usually when a triangle is stretched, the texture stretches uniformly with it. However, when rendering certain objects, such as a lizard with spikes and scales, the uniform stretch of the texture is undesirable because it breaks the illusion of the model. Therefore, the goal of this project is to develop a method to stretch a texture variably across a triangle as the triangle's dimensions are changed. We first propose a method which computes texture stretching according to a linear model. Due to limitations in this model when applied to two dimensions, we propose an alternative model which is designed to preserve two dimensional geometry. We describe the rationale and algorithms defining each model. We analyze the behavior, overall effectiveness, and computational complexity of each model with a variety of examples.

Contents

1	Abstract	ii
2	Background	1
2.1	Graphics Pipeline	1
2.2	Textures	3
3	Spring Model	5
3.1	Model Description	5
3.2	Physical Basis	7
3.3	Linear Model on the GPU	8
3.4	Approximating Inverse Stretch on the GPU	10
3.5	Extension to Two Dimensions	12
3.6	Analysis of Spring Model	12
4	Rigid Region Model	16
4.1	Model Rationale	16
4.2	Model Description	17
4.3	Algorithm	23
4.4	Analysis of Rigid Region Model	25
5	Conclusion	28
5.1	Analysis	28
5.2	Further Research	28
6	References	31



2 Background

2.1 Graphics Pipeline

Computer graphics refers to any use of a computer to create or manipulate images [8]. Graphical interfaces are an essential part of the functionality of modern computers. When using a computer screen, it is necessary to render a new image to the screen roughly 60 times per second. Furthermore, an average computer screen has upwards of one million pixels. Thus, utilizing an interactive graphical interface of a computer requires updating roughly 6×10^7 values *every second*. Factoring in additional computations, such as computing shadows in a video game, and the computational expense of graphics skyrockets. Due to the importance of graphics in modern computers and its high computational cost, GPU's (graphical processing units) were developed to tackle image rendering, and have become ubiquitous in modern machines.

In order to compute the color values of each pixel, the GPU uses extensive parallelization. Most GPUs use a process called *rasterization*, which is meant to convert

a scene consisting of triangles and vectors into an image. Consider figure 1. The input to the GPU is a collection of triangles (the coordinates of their vertices), a *vertex shader* program, and a *fragment shader* program. Using these inputs, the GPU calculates the color of each pixel.

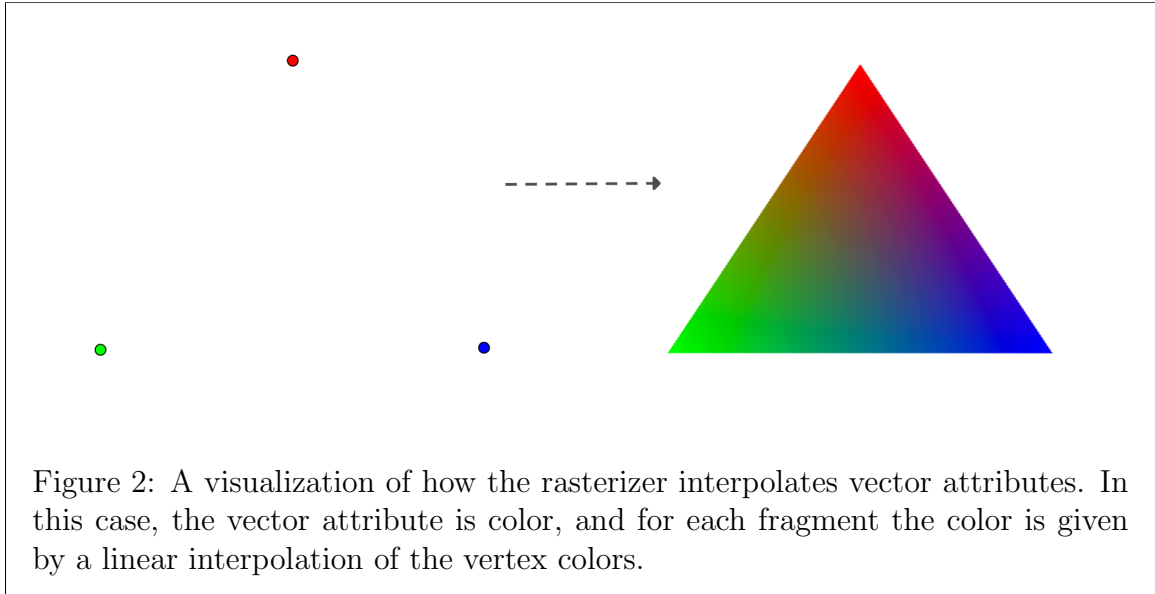
Vertex Shader

The vertex shader is run once for each vertex of each triangle, and each time returns a single vertex, hence the name *vertex* shader. One common use of the vertex shader is to apply linear transformations to the vertices here so that the triangles are in the correct positions relative to the camera. Furthermore, *vertex attributes* can be sent to the GPU as additional input. For instance, we may want to color each vertex. In this case, an array of color values c must be sent to the GPU defining a color for each vertex of each triangle. In the figure, these attributes are denoted “uniform variables”, and are sent from the CPU. For this project, we will be sending *texture coordinates* to each vertex, discussed more in the next section. The output of the vertex shader is a list of vertices and their attributes.

Rasterization

The input to the rasterizer is the output of the vertex shader, which is a list of vertices and their attributes. The rasterizer performs two actions: for each triangle, the rasterizer *enumerates* each of the pixels which that triangle covers, and for each pixel, *interpolates* the vertex attributes from the vertex shader. As output to the rasterizer, each pixel is called a *fragment* (which emphasizes that the fragment is not just the pixel location on the screen, but also each of its interpolated attributes). Figure 2 gives a visual description of interpolation in the case of color. For each fragment of a triangle T , the rasterizer linearly interpolates the color value of the vertices of T by the fragment’s barycentric coordinates.

Fragment Shader

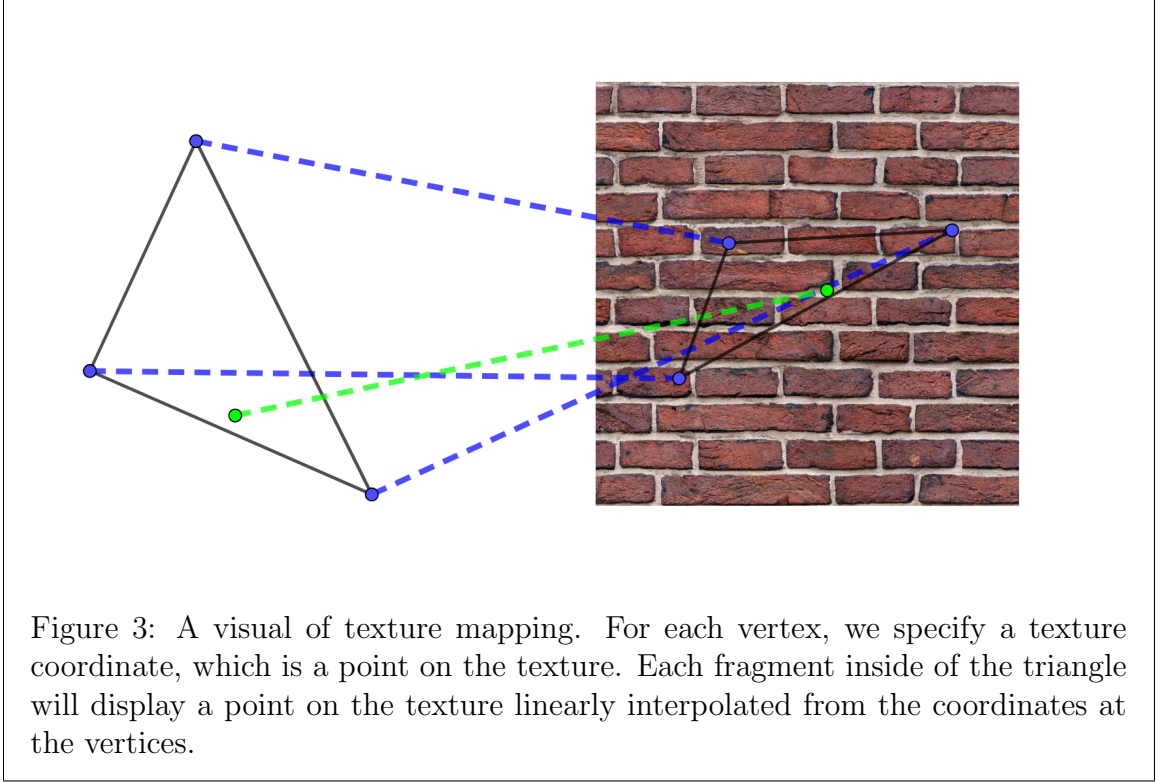


The fragment shader is run once for each fragment. Its input is the fragment's location on the screen as well as any vector attributes which were interpolated by the rasterizer for that fragment. Therefore, the fragment shader runs on a pixel by pixel basis. The only output of the fragment shader is the color of the given pixel, and is sent directly through hardware to the screen or graphical display.

The power of the GPU pipeline is that rasterization has an extremely efficient implementation through hardware on the GPU and the fragment shader is run independently for each pixel, so each of the 10^6 pixel colors can be calculated simultaneously. While there are alternatives to rasterization, namely ray tracing, the sheer computational efficiency of rasterization has proven to be invaluable in many graphics settings.

2.2 Textures

The GPU pipeline is able to draw scenes with triangles which are a single color or a linear interpolation of three colors. Oftentimes, it is desirable to draw a triangle which is not a uniform color, but has an image overlaid on it. In computer graphics,



a *texture* refers to a wide class of objects which specify precomputed data for a scene. For the sake of this project, a *texture* will mean a two dimensional image.

Given an image I and a triangle T , in order to overlay the image on the triangle, it is necessary to specify at what coordinates the image should be overlain. Two dimensional textures on the GPU are normalized to $[0, 1] \times [0, 1]$, so a texture given by an image I can be thought of as a function $g : [0, 1] \times [0, 1] \rightarrow \text{RGB}$ which sends each coordinate to the color of the image at that point. The exact definition of g is more nuanced - for instance, defining the behavior of $g(q)$ if q lies between pixels on the image I . Another problem with textures arises when the resolution of the texture is large compared to the size of the triangle, which causes aliasing in the final image if g is defined naively. There are standard solutions to these problems through more clever sampling techniques and “mip maps” - but we will not need to understand these tools for this project. Therefore for each vertex v of each triangle, a *texture*

coordinate $p_v \in [0, 1] \times [0, 1]$ is sent to the vector shader as a vector attribute. Then, for each fragment f of the triangle T , the rasterizer computes a texture coordinate p linearly interpolated from the vertices. Using this texture coordinate, we return the color $g(q)$ from the fragment shader.

Thus, texture mapping is achieved by the interpolation of texture coordinates by the rasterizer. In particular, after the texture coordinates of each vertex have been specified, the texture coordinates of each fragment are determined. Therefore, in order to simulate non-uniform stretching of a texture on a triangle, it is necessary to modify the texture coordinates inside of the fragment shader. Because the fragment shader is run once for each pixel, whatever computation is performed inside of the fragment shader must have low computational cost. This is the heart of the difficulty of this project - modelling variable stretching can conceivably be achieved through complex physical models on the CPU, but implementing any such model on the GPU would be computationally expensive to the point of being unusable for any real time rendering application.

3 Spring Model

The goal of the project is to develop a technique to modify texture coordinates real time in order to simulate varying elasticity of a texture. The first step is to specify a physical model which can be used to model texture elasticity. We will first describe a physical model for a 1-dimensional texture, and then extend it linearly to 2 dimensions.

3.1 Model Description

Consider a 1-dimensional texture comprised of horizontal line segments of size l_1, l_2, \dots, l_n . For simplicity assume the texture is length 1 so $\sum_i l_i = 1$. Suppose that the elasticity of each line segment is given by a corresponding value e_1, e_2, \dots, e_n (e for *elasticity*).

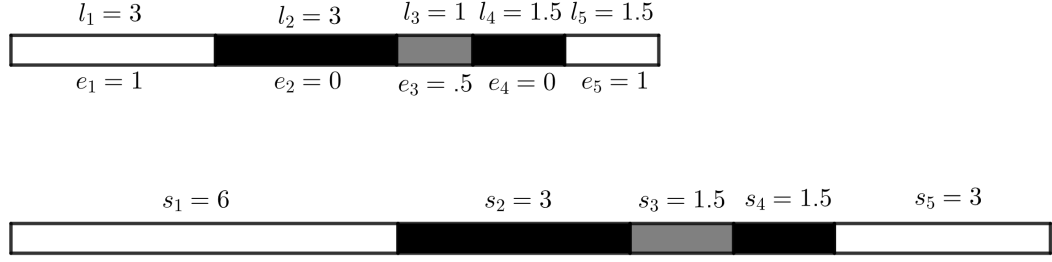


Figure 4: Linear model visual. The color of each segment is proportional to its elasticity constant e_i . Notice that the brighter (higher elasticity) segments stretch more on deformation than the dark segments. If $e_i = 0$, the segment is full rigid and thus does not change length when stretched at all.

The values e_j are in the range $[0, 1]$, so that $e_j = 0$ signifies that the j th segment does not stretch at all, while $e_j = 1$ signifies the j th segment is fully elastic. Suppose that the line segment is then stretched to a length of l . The one dimensional *spring model* stretches each segment proportional to two values:

1. its elasticity constant e_i .
2. its unstretched length l_i .

In particular, suppose that after stretching the texture to a length of l , the line segments of length l_1, l_2, \dots, l_n are stretched to lengths of s_1, \dots, s_n respectively, as shown in figure 4. Then, the stretch values according to the spring model are chosen such that $\frac{e_j l_j}{(s_j - l_j)}$ is the same for each segment. Equivalently, (and more numerically stable to avoid dividing by zero), there is a unique value r such that

$$r e_j l_j = (s_j - l_j) \quad \text{for all } j \in [1, n]$$

Furthermore, we must have that the stretched length is l , so

$$\sum_j s_j = l$$

Combining these two equations, we have that:

$$r \sum_j e_j l_j = \sum_j (s_j - l_j) = \sum_j s_j - \sum_j l_j = l - 1$$

$$r = \frac{l - 1}{\sum_j e_j l_j}$$

Notice that r is not defined if all of the line segments have elasticity of 0 - this corresponds to the fact that if none of the segments can be stretched, then it is impossible to change the length of the texture. Once r is computed, we have a simple way to compute the stretch of any segment s_j :

$$s_j = l_j(r e_j + 1)$$

Furthermore, if $r_0 = \frac{1}{\sum_j e_j l_j}$ is computed once, then it is efficient to compute other values of s_j with a linear equation in l by $r = (l - 1)r_0$:

$$s_j = r'(l - 1)e_j l_j + l_j$$

3.2 Physical Basis

The spring model behaves as if each line segment of the texture is a spring with spring constant k_j equal to $\frac{1}{e_j l_j}$ [3, Chapter 7]. A static system of n springs in series results in an equal amount of force applied through each spring. Thus, by Hooke's law, if Δx_j is the change in length of the j th spring and k_j is the j th spring constant, the following holds:

$$k_i \Delta x_i = k_j \Delta x_j \quad \text{for all } i, j$$

Since $\Delta x_j = s_j - l_j$ and by substituting $k_j = \frac{1}{e_j l_j}$, this corresponds to

$$\frac{s_i - l_i}{e_i l_i} = \frac{s_j - l_j}{e_j l_j} \quad \text{for all } i, j$$

This is exactly the statement that there is some $r \in \mathbb{R}$ such that

$$\frac{s_j - l_j}{e_j l_j} = r \quad \text{for all } j$$

Therefore, the spring model behaves exactly as if each segment is a spring with corresponding spring constants.

There is a practical reason for inverting the spring constant as above. Hooke’s law implies that if two springs with spring constants k_1, k_2 are placed in series, the resulting spring has spring constant k :

$$k = \left(\frac{1}{k_1} + \frac{1}{k_2} \right)^{-1}$$

Therefore, if the spring constants are inverted as $e = k^{-1}$, Hooke’s law yields the computationally simpler equation:

$$e = e_1 + e_2$$

Furthermore, we represent the spring constants as proportional to the lengths l_j of each spring. This is characteristic of elastic materials and is intuitive - if there are two portions of lengths $l_1 < l_2$ with the same elasticity, we expect l_2 to stretch more than l_1 when a force is applied. Ultimately, this detail does not change the implementation, since each “spring” is the length of a pixel and thus each spring is the same length.

3.3 Linear Model on the GPU

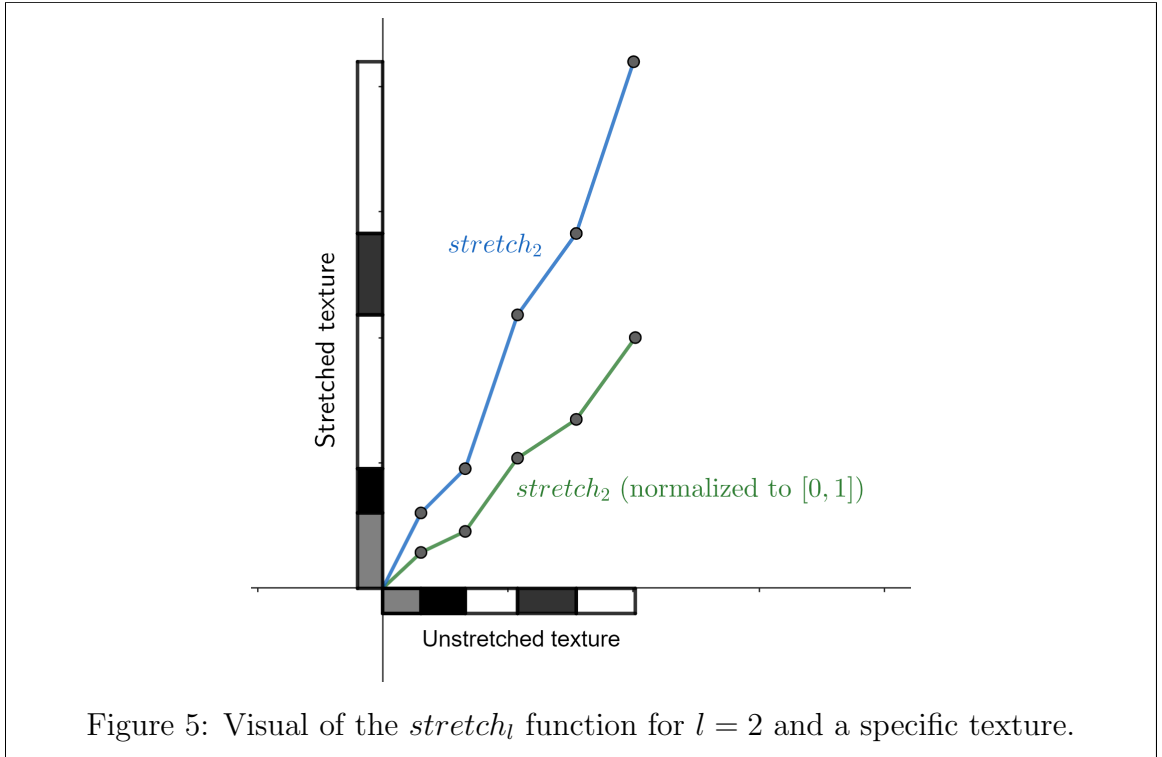
Now that a model has been specified for stretching segments of a texture, it must be implemented on the GPU. In particular, for any one dimensional texture specifying elasticity and any stretch length l , there is an associated function $stretch_l : [0, 1] \rightarrow [0, l]$ defined by the spring model which sends each point in $[0, 1]$ to its stretched position. Thus at a fragment f in the fragment shader with texture coordinate

$p \in [0, 1]$, we must determine the point $q \in [0, 1]$ such that $stretch_l(q)/l = p$. Thus, in order to compute q , we must efficiently compute the inverse $stretch_l^{-1}(lp)$.

In the spring model, notice that each segment of the texture is stretched proportionally to its length and elasticity. Therefore, the function $stretch_l$ has derivative proportional to the elasticity constant at each point, so $stretch_l$ is a piecewise linear function, as seen in figure 5.

Therefore, it is possible to explicitly compute the inverse of $stretch_l$ by reflecting it across $y = x$. This amounts to inverting the slopes of each linear portion and computing the reflection of each vertex across $y = x$. However, explicitly computing the inverse of $stretch$ is not an effective solution for the following reasons:

1. Computing this inverse explicitly is computationally inefficient even in one dimension due to the casework required to invert each vertex of the function.
2. If the texture is compressed short enough that the $stretch$ function is not injective, the inverse does not exist at all and thus cannot be computed. In this



situation, it is better to compute an approximation of an inverse than to fail completely.

3. Computing the inverse does not extend to two dimensions.

The third bullet is the most important - computing an explicit inverse is simply unattainable in two dimensions, partially because the *stretch* function is not guaranteed to be injective in most situations. Therefore, we should aim for an effective way of approximating the inverse on the GPU rather than computing it directly.

3.4 Approximating Inverse Stretch on the GPU

In order to compute the inverse of $stretch_l$ in the fragment shader, we first describe how the function $stretch_l$ can be defined on the fragment shader. The first step is to compute the function $stretch_2 : [0, 1] \rightarrow [0, 2]$ on the CPU. Notice that in the spring model, the stretch lengths of each segment are linear in l where l is the stretch amount. Therefore, since the following equation holds for $l = 1$ and 2, it holds for all l :

$$stretch_l(x) = (l - 1)(stretch_2(x) - x) + x \quad (1)$$

The function $stretch_2(x)$ can be sent to the fragment shader as a *texture*, and the stretch length l can be sent to the fragment shader as a uniform variable. Then, the function $stretch_l : [0, 1] \rightarrow [0, l]$ can be defined on the fragment shader by equation (1). Notice that the linearity of the spring model means that the $stretch_l$ function is extremely efficient to compute on the fragment shader, since it amounts to a texture lookup combined with a few arithmetic operations.

Thus, our problem is to compute $stretch_l^{-1}(ly)$ given a stretch length l and a texture coordinate $y \in [0, 1]$ in the fragment shader. Define $f(x) = stretch_l(x)/l$, so the goal is to compute $f^{-1}(y)$. Notice that if the stretch value l is not too small, f is injective, so the value $x = f^{-1}(y)$ is unique. Notice that x satisfying $f(x) = y$ is equivalent to x being a fixed point of the function $h(x) = f(x) - y + x$. Therefore, our

general strategy will be to start by guessing $x = y$, and then iteratively modifying x to closer approximate a fixed point of h .

“Step Method”. In this method, we assume that f is monotone increasing, which

Algorithm 1 StepMethod ($f : [0, 1] \rightarrow [0, 1]$, $y \in [0, 1]$)

```

 $x = y$ 
 $c = .3$ 
while  $|f(x) - y| > \epsilon$  do                                 $\triangleright \epsilon$  small or fixed number of iterations
     $x = x + c(y - f(x))$ 
end while
return  $x$ 

```

ensures that the updated guess $x + c(y - f(x))$ always pushes x in the right direction. This is true so long as l is not too small. Furthermore, if c is smaller than the largest derivative of f , these two facts imply that $x \rightarrow x + c(y - f(x))$ is a contraction of $[0, 1]$, and therefore will converge to a fixed point. A fixed point x satisfies $y = f(x)$, and thus under modest requirements **StepMethod** will converge to an inverse $x = f^{-1}(y)$.

Newton’s Method. Newton’s method is a more specific version of **StepMethod** -

Algorithm 2 NewtonsMethod ($f : [0, 1] \rightarrow [0, 1]$, $y \in [0, 1]$)

```

 $x = y$ 
while  $|f(x) - y| > \epsilon$  do                                 $\triangleright \epsilon$  small or fixed number of iterations
     $x = x + \frac{1}{f'(x)}(y - f(x))$ 
end while
return  $x$ 

```

instead of a generic constant c , we specify the coefficient c as $1/(f'(x))$. This tends to be a more effective method to achieve convergence, partly because it is not necessary that f is monotone. There a variety of guarantees about the convergence of Newton’s method [1]. One nice feature of Newton’s method is that if f is linear, Newton’s

method outputs the desired fixed point of h in a single iteration. One potential issue is that most of the guarantees about Newton’s method convergence apply when f has a continuous second derivative. However, because the second derivative of f is 0 wherever it is defined (and f is usually monotonic), Newton’s method should converge. In practice, Newton’s method is effective at determining the stretch inverse in the fragment shader, as discussed in the next section. Notice that the derivative $f'(x)$ at a point x is equal to the value re_j where x has elasticity e_j and r is the constant described in section 3.1. Thus, the values re_j can be computed while determining the $stretch_2$ function and can be sent to the fragment shader as an additional texture.

3.5 Extension to Two Dimensions

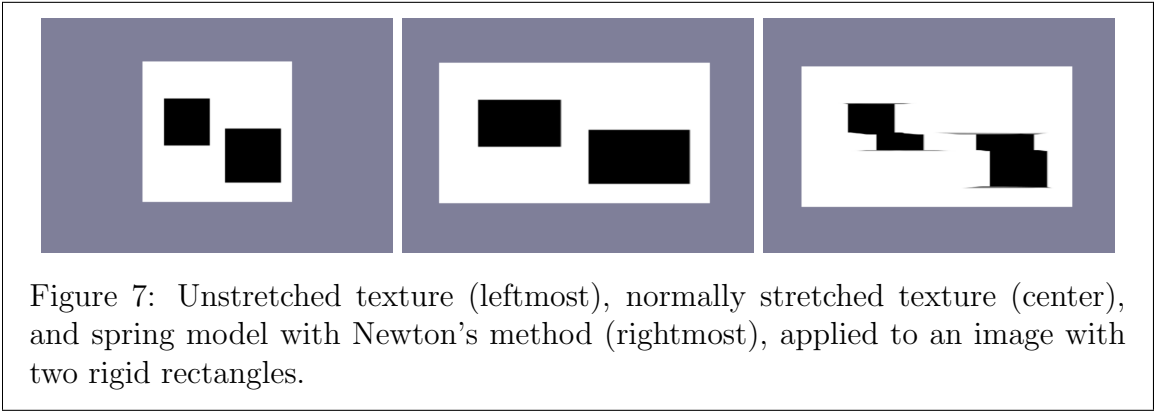
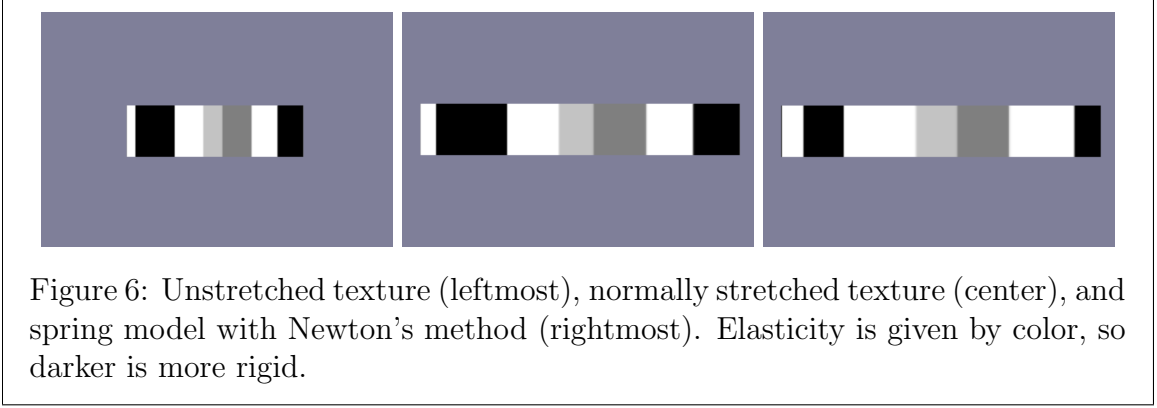
We extend the spring model to two dimensions so that it behaves in the same way when restricted to one dimension - we calculate the stretch functions for the x and y directions separately to form a function $stretch_l : [0, 1]^2 \rightarrow [0, 1]^2$ defined by $stretch(a, b) = (stretch_l^x(a), stretch_l^y(b))$. We modify the `NewtonsMethod` algorithm by replacing $1/f'(x)$ with the multiplicative inverse of the gradient of f [1], i.e. the matrix

$$\begin{bmatrix} \frac{1}{\partial f / \partial x} & 0 \\ 0 & \frac{1}{\partial f / \partial y} \end{bmatrix}$$

3.6 Analysis of Spring Model

In figure 6, we show the results of the algorithm applied to a “one dimensional texture” which is widened along the y axis for clarity of the results. Notice that when the texture is stretched uniformly (without any spring model adjusting), the rigid portions of the texture (the black pieces) stretch proportionally to the rest of the texture. When using the spring model, the size of each rigid portion is kept constant as the texture is stretched. This is the expected and desired behavior.

Now consider figure 7 of a texture with 2 rigid rectangles. The spring model preserves



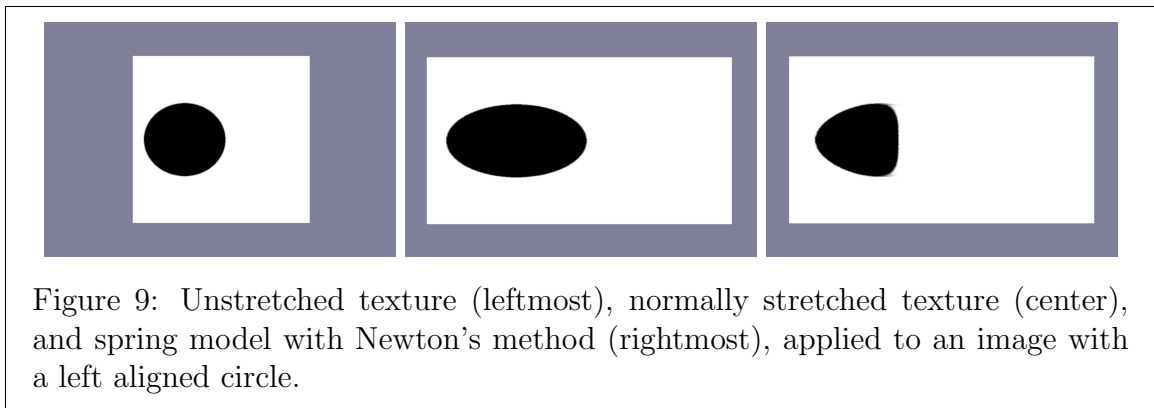
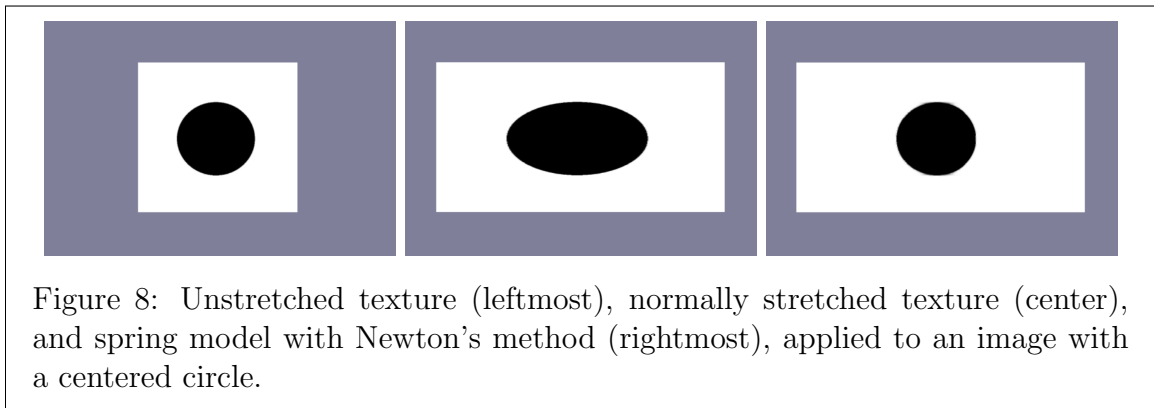
the sizes of the rectangles when stretched along the x axis, as desired. However, the rectangles split apart as the x -stretch is applied. This can be explained explained by considering how the spring model is extended to two dimensions. When an x -dimensional stretch is applied to the texture, the stretch values of each row are calculated independently, so two adjacent rows of pixels can have completely different x stretch behavior despite being next to one another. Therefore, the $stretch_t^x$ function can be highly discontinuous along a column, as exemplified here.

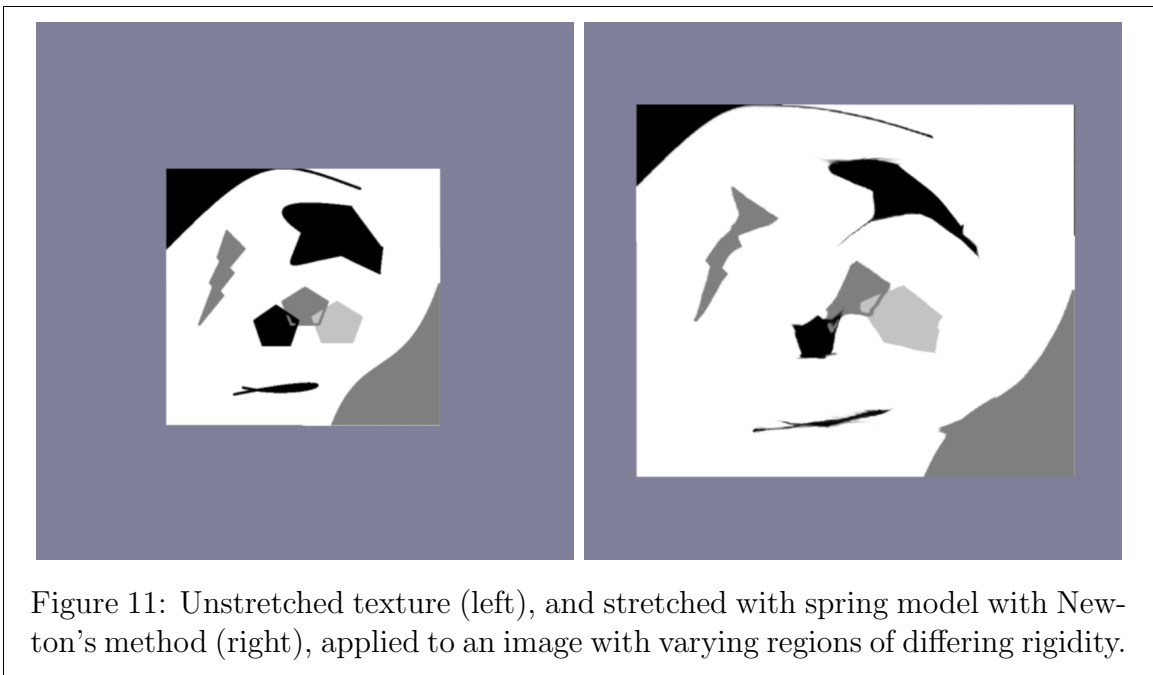
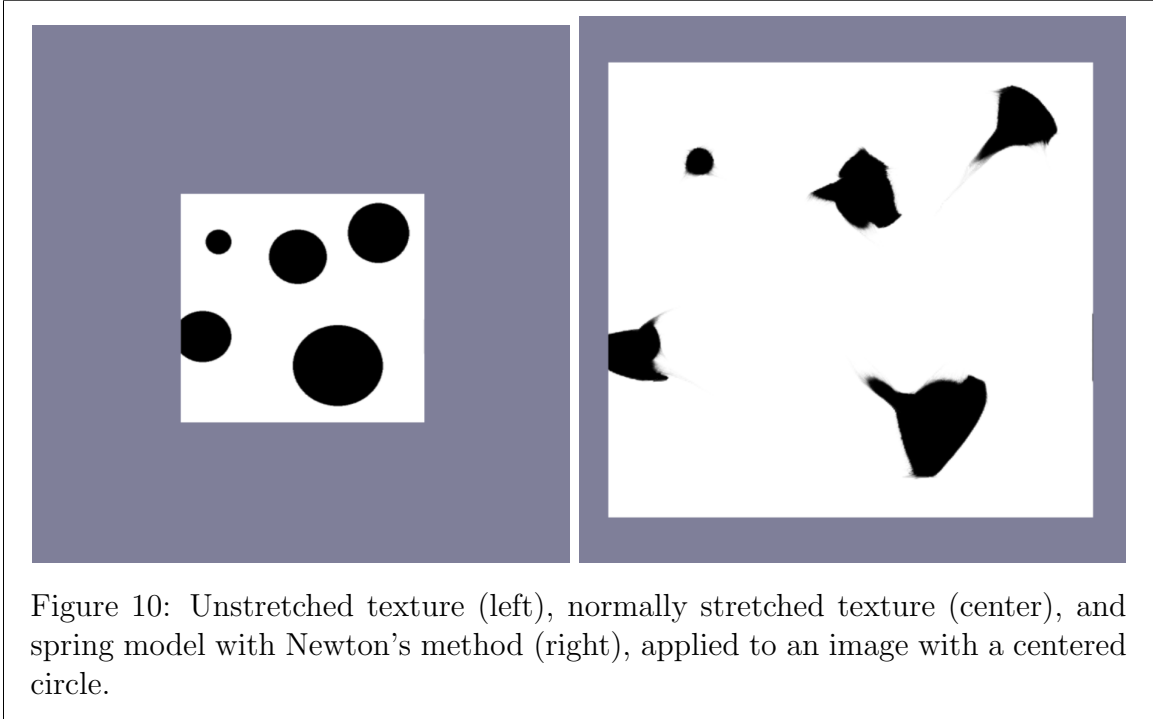
Furthermore, the brushed “tails” in the image are due to the stretch function being highly discontinuous around these points, so any convergence method (like Newton's method) will generally fail to find the inverse of the stretch.

Figures 8 and 9 contain a texture stretched with a single rigid circle. In figure 8, the model behaves as we would hope - the circle has the same size and center when the texture is stretched. In 9, we have the exact same image, but with the rigid circle

translated to the left. In this case, our model fails to preserve the shape of the circle. Once again, this is because the spring model treats each row independently. Because the middle row of the rigid circle is close to the left side of the image, it is pulled to the left when stretched. However, the other rows of the circle have more whitespace between them and the left edge of the image, so when the image is stretched, they are pulled to the right.

Compared to the situations required in a more realistic graphics setting, like rendering realistic reptilian skin, these toy examples of a single circle or a few rectangles are extremely simple. Let us consider more complicated examples representing multiple regions of varying rigidity, as shown in figures 10 and 11. In figure 10, we have a collection of rigid circles. When the texture is stretched vertically and horizontally, the area of each circle stays roughly the same, but the shapes are significantly de-





formed. In 11, we have a collection of shapes of varying rigidity, which when stretched become unrecognizable.

Even in these simple cases, it is clear that the spring model deforms the structure

of the texture far too much to be effective. Even though it preserves the size of regions when stretched and behaves well in one dimension, two dimensional shapes become unrecognizable with the spring model under even modest conditions.

4 Rigid Region Model

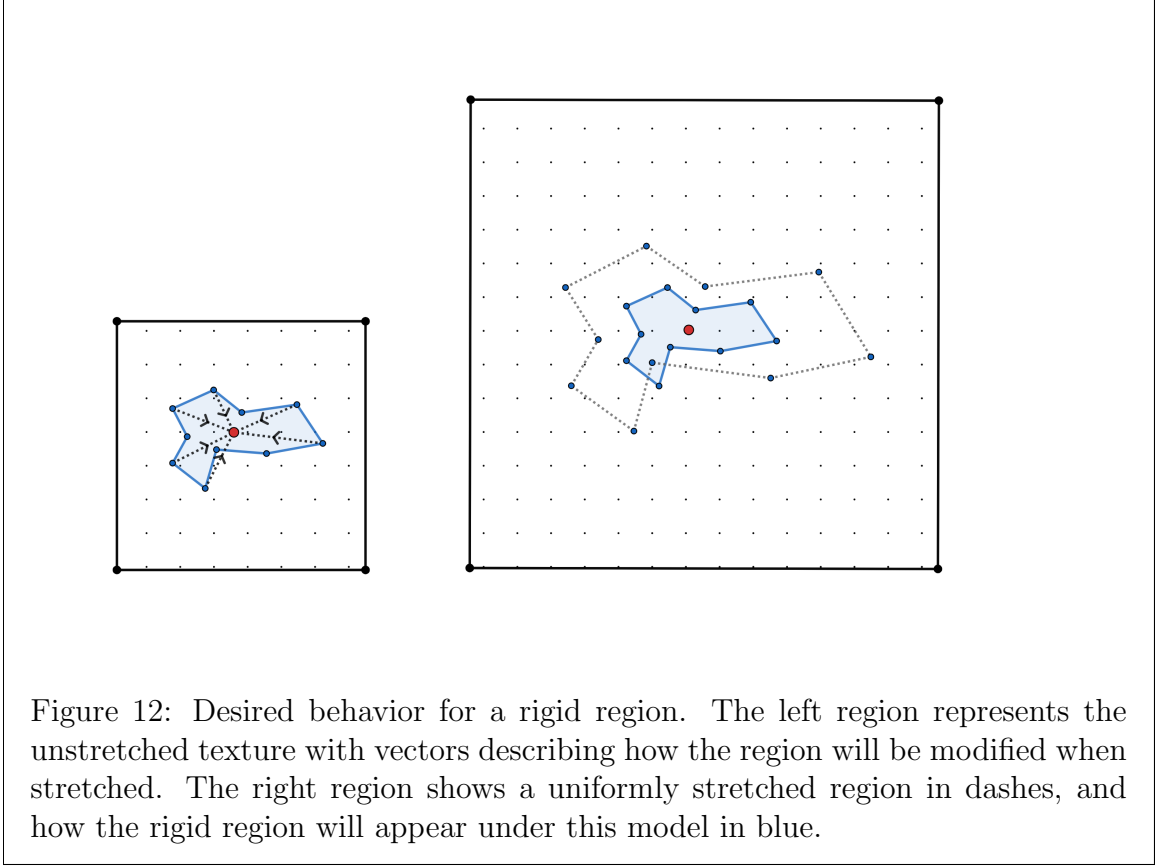
The spring model described in the previous section was designed for one dimensional textures and was naively extended to two dimensions by treating each dimension of stretch independently. This approach ignored the interaction between the geometry of the x and y axes. This failure of the spring model is the motivation for our next model, which simplifies the problem by only allowing completely rigid or non-rigid regions, but is designed to treat two dimensional geometry with more nuance.

4.1 Model Rationale

Consider a texture with a single rigid region R such that the rest of the texture is elastic. When the texture is stretched, the goal is to preserve the size, shape, and location of the rigid region. The basis of the *rigid region model* is to compute the centroid c of the region R (the weighted average of its points), and then stretch each point x in the rigid region linearly towards c , as shown in figure 12.

Notice that this approach ensures that the size of R will be modified correctly while maintaining its geometric shape. Furthermore, sending each point towards the centroid will ensure that the center of the rigid region will remain constant while stretching.

What about the other points on the texture which are not in a rigid region? Under this model, we only have two possible values of elasticity for each point: fully rigid, or fully elastic. Therefore, all of the other points are fully elastic and thus do not have any specific constraints. We aim to make the stretch function on the texture as smooth as possible so that the stretching appears realistic. Therefore, we will choose



the behavior of stretching on non-rigid points to maximize the smoothness of the stretching function.

4.2 Model Description

Suppose that there is a texture I of size $[0, 1] \times [0, 1]$ and it is stretched from $[0, 1] \times [0, 1]$ to $[0, l_x] \times [0, l_y]$. Let us describe a function $stretch_{l_x, l_y} : [0, 1] \times [0, 1] \rightarrow [0, l_x] \times [0, l_y]$ which specifies where each point is stretched to. The rigid region model assures that pixels in a rigid region are stretched linearly towards the centroid. In particular, for a point p in a rigid region with centroid c , we have (where $p.x, p.y$ are the x and y coordinates of p respectively):

$$stretch_{l_x, l_y}(p) = \left(p.x + (l_x - 1)c.x, p.y + (l_y - 1)c.y \right)$$

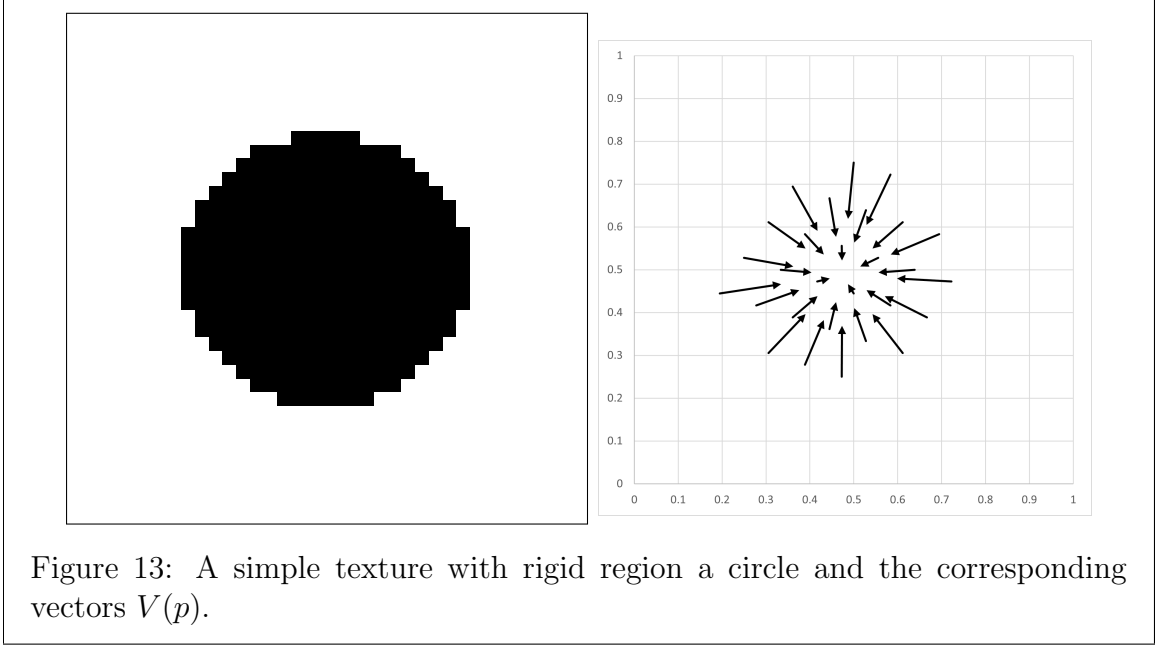


Figure 13: A simple texture with rigid region a circle and the corresponding vectors $V(p)$.

Define vectors $V(p)$ by subtracting by p and dividing coordinate-wise by l_x, l_y :

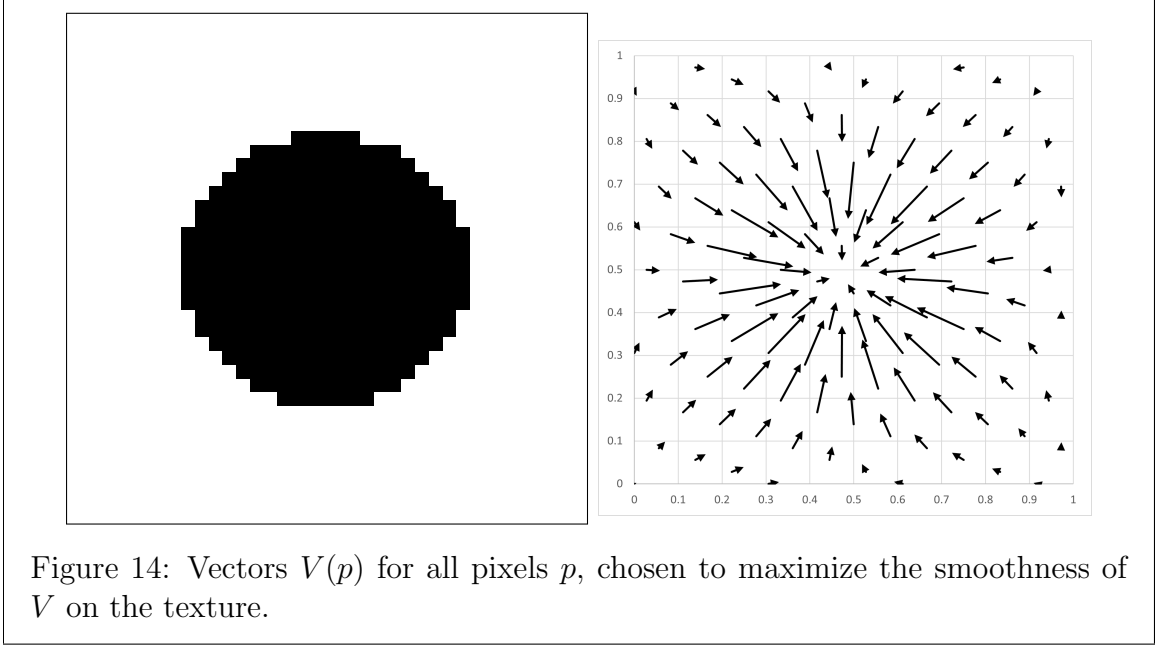
$$V(p) = \left(\frac{(1 - l_x)p.x + (l_x - 1)c.x}{l_x}, \frac{(1 - l_y)p.y + (l_y - 1)c.y}{l_y} \right) \quad (2)$$

The vectors $V(p)$ are shown in figure 13. Notice that by normalizing, the vectors for p in the rigid region point towards the centroid with length proportional to the distance from the centroid.

Now, the goal is to define $V(p)$ for the remaining pixels p in the texture to minimize the sum of the square difference of adjacent vectors, which is one numerically convenient way of phrasing continuity or smoothness. In particular, we select the remaining vectors $V(p)$ to minimize the following sum:

$$\min_V \sum_p \sum_q |V(p) - V(q)|^2 \quad (3)$$

where p is summed over pixels not in a rigid region, and q is summed over points neighboring p . Furthermore, this sum is minimized with the constraint that $V(p)$ is fixed for p in a rigid region as defined in equation (2). p and q are *neighbors* if p



and q are horizontally or vertically adjacent, including wrapping around the image - for instance, a point on the far left side of the image has a neighbor on the far right side, so every point has exactly 4 neighbors. A solution to this minimization for V is shown in figure 14, where the remaining vectors are chosen so V is smooth.

Thus, choosing the stretch values V on the elastic coordinates (points where the texture is elastic) amounts to solving a least squares problem. For simplicity, let us solely solve for the x components $V_x(p)$ of the vectors $V(p)$. The same method can be used to solve for the y components, since the sum in equation (3) splits into two least squares problems in V_x and V_y :

$$\begin{aligned} \min_V \sum_p \sum_q |V(p) - V(q)|^2 &= \min_V \sum_p \sum_q (V_x(p) - V_x(q))^2 + (V_y(p) - V_y(q))^2 \\ &= \min_{V_x} \sum_p \sum_q (V_x(p) - V_x(q))^2 + \min_{V_y} \sum_p \sum_q (V_y(p) - V_y(q))^2 \end{aligned}$$

Thus, let us consider the equation in x :

$$\min_{V_x} \sum_p \sum_q (V_x(p) - V_x(q))^2 \quad (4)$$

One effective method of solving least squares problems is *conjugate gradient*, which solves $Ax = \mathbf{b}$ for skew-symmetric positive semi definite matrices A . Let us show that we can write the equation in (4) as

$$f(x) = \frac{1}{2} x^T A x - b x + c \quad (5)$$

for a skew-symmetric positive semi definite matrix A . Since equation (5) is convex if A is positive definite, its minimum can be found by setting the gradient equal to 0:

$$\Delta f(x) = Ax - b = 0$$

Let

$$x = \begin{bmatrix} V_x(p_1) \\ V_x(p_2) \\ \vdots \\ V_x(p_N) \end{bmatrix}$$

where p_1, p_2, \dots, p_N are the pixels which are *not* inside of a rigid region, since the remaining values of V are already fixed. Let us consider a term $|V(p) - V(q)|^2$ in equation (3) for p in a rigid region and q a neighbor of p . There are two cases: q is in a rigid region or it is not. First suppose q is not in a rigid region, so $p = p_i$ and $q = p_j$ for some $i \neq j$. Then, let A^{ij} be the matrix with coefficients a such that

$$a_{ij} = a_{ji} = -2 \quad a_{ii} = a_{jj} = 2$$

and the other terms are 0. Then,

$$\frac{1}{2}x^T A^{ij}x = \frac{1}{2}\left(2V_x(p_i)^2 + 2V_x(p_j)^2 - 2V_x(p_i)V_x(p_j) - 2V_x(p_j)V_x(p_i)\right) = (V_x(p_i) - V_x(p_j))^2$$

Thus, A^{ij} encodes the desired term for p_i, p_j . Now consider the case when q is in a rigid region, so there is a precomputed vector $V_x(q) = \frac{(1-l_x)p.x + (l_x-1)c.x}{l_x}$ for q . Let $p = p_i$. Then, let b^{iq} be a vector with $2V_x(q)$ in the i th entry and 0 in remaining terms. Let A^i have a 2 in the i th diagonal entry and let $c = V_x(q)^2$. Then,

$$\frac{1}{2}x^T A^i x - b^{iq}x + c = \frac{1}{2}2V_x(p)^2 - 2V_x(q)V_x(p) + V_x(q)^2 = (V_x(p) - V_x(q))^2$$

Thus by linearity, by adding the symmetric matrices A^i, A^{ij} , vectors b^{iq} , and constants c for each pair of points p, q with p rigid and q a neighbor, there is a symmetric matrix A , vector b , and constant c such that

$$\frac{1}{2}x^T Ax - bx + c = \sum_p \sum_q (V_x(p) - V_x(q))^2$$

as in equation (4). Furthermore, A is positive *semi* definite since for any choices of vectors $V_x(p_1), \dots, V_x(p_N)$, we have:

$$\frac{1}{2}x^T Ax = \sum_p \sum_{q \text{ rigid}} (V_x(p) - V_x(q))^2 + \sum_p \sum_{q \text{ not rigid}} V_x(p)^2 \geq 0$$

where q is summed over the neighbors of p . Furthermore, the $x^T Ax$ is only zero if $V_x(p) = V_x(q)$ for all neighbors p, q , and $V_x(p) = 0$ for all p adjacent to a rigid region. So long as there is at least one rigid region, it is impossible for these requirements to be satisfied by any vector other than the zero vector, so A is positive semi definite unless the texture is fully elastic. Thus, applying conjugate gradient obtains a solution vector $V_x(p_1), \dots, V_x(p_N)$. Combining these steps describes a complete algorithm for determining the vectors $V(p)$. After renormalizing by multiplying by l_x, l_y and adding

p , a stretch function $stretch_{l_x, l_y}$ is determined.

This algorithm will determine the optimal stretch vectors $V_x(p)$ for a specific x -stretch l_x . If this algorithm has to be repeated for each new stretch value l_x , the algorithm would be too computationally expensive to be usable in any real time setting. Let us show that if the vectors V_x are found for $l_x = 2$ (denoted $V_x^2(p)$), then for any stretch l_x , the new stretch vectors V_x can be found by linearly interpolating with V_x^2 :

$$V_x(p) = \frac{2(l_x - 1)}{l_x} V_x^2(p)$$

First notice that this holds for points q in a fixed region with centroid c since we have

$$V_x(q) = \frac{(1 - l_x)q.x + (l_x - 1)c.x}{l_x}$$

and

$$V_x^2(p) = \frac{c.x - q.x}{2}$$

so therefore

$$V_x(q) = \frac{l_x - 1}{l_x} (c.x - q.x) = \frac{2(l_x - 1)}{l_x} V_x^2(p)$$

Now let us show that the same equality holds for points q not in a region. Let b_2 be the vector defining the least squares minimization in equation (4) when the stretch is $l_x = 2$. After scaling by l_x , each of the vectors in b_2 will be scaled by $\frac{2(l_x - 1)}{l_x}$ by the calculation above. Furthermore, A is independent of the stretch l_x . Therefore, by linearity, the solution to $Ax = b$ will be given by exactly $x = \frac{2(l_x - 1)}{l_x} x_2$, where x_2 is the solution for $l_x = 2$. Therefore, it suffices to compute the stretch vectors V^2 once for $l_x = 2$ and then scale by $\frac{2(l_x - 1)}{l_x}$ for successive stretching. Therefore, the function $stretch_{l_x, l_y}$ can be defined on the fragment shader by sending the vectors V to the fragment shader as a texture and then multiplying and dividing by the required coefficients. Therefore, the same method of calculating an inverse as in section 3.4 can be used to apply the stretching of the texture coordinates in the fragment shader.

If Newton's method is to be used, one can compute approximate derivatives at each point by using local values of V and sending these as an additional texture to the fragment shader.

4.3 Algorithm

Algorithm 3 RigidRegionModel (bool texture[width, height], rigidRegions[])

```

for  $R \in \text{rigidRegions}$  do
  compute the centroid  $c_R$  of  $R$ 
  for  $q \in R$  do
    compute the stretch vectors  $V(q) = (c_R - q)/2$ 
  end for
end for

```

Construct the matrix A based on pixel adjacency.

Construct b by inserting vectors $2V(q)$ for q in a rigid region.

Solve $Ax = b$ using conjugate gradient.

Fill remaining vectors in V with solution values x .

Normalize V and send to GPU.

NewtonsMethod or StepMethod in fragment shader to compute inverse.

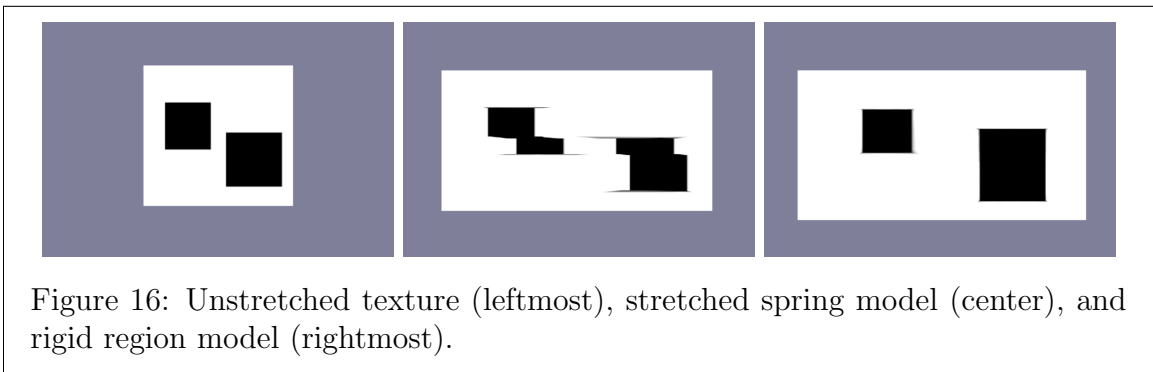
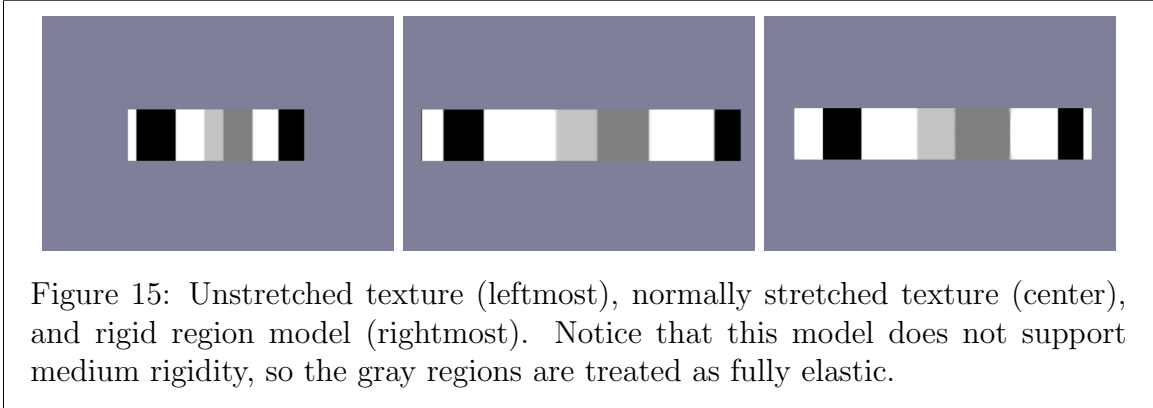
Use modified texture coordinates to draw stretched image.

The algorithmic complexity of **RigidRegionModel** is dominated by the conjugate gradient algorithm. Each of the steps other than conjugate gradient are linear in n where $n = \text{width} * \text{height}$ of the provided image. The matrix A has $O(5n)$ non-zero entries - one for each pixel and four for each of its neighbors.

However, the conjugate gradient step has potentially large computational time. In general, the algorithmic complexity of conjugate gradient is $O(n\sqrt{\kappa})$ where κ is the condition number of A [7]. Consider the trivial case when there is no rigid region. Let A' be the matrix defining the least squares problem in equation (5). Then, the vector $x = [1, 1, \dots, 1]^T$ satisfies $A'x = 0$ since

$$A'x = \sum_{i,j} (x_i - x_j)^2 = \sum_{i,j} 0 = 0$$

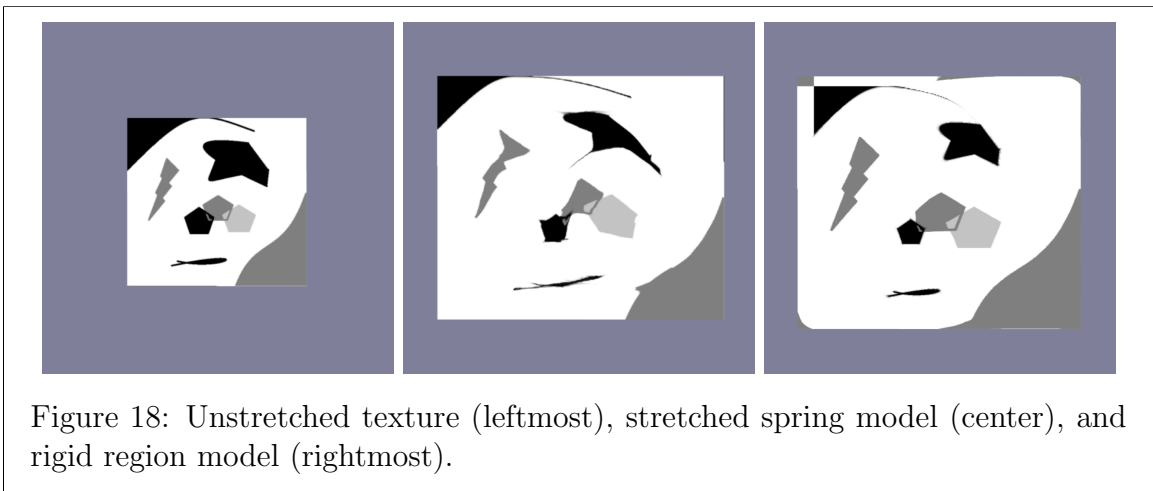
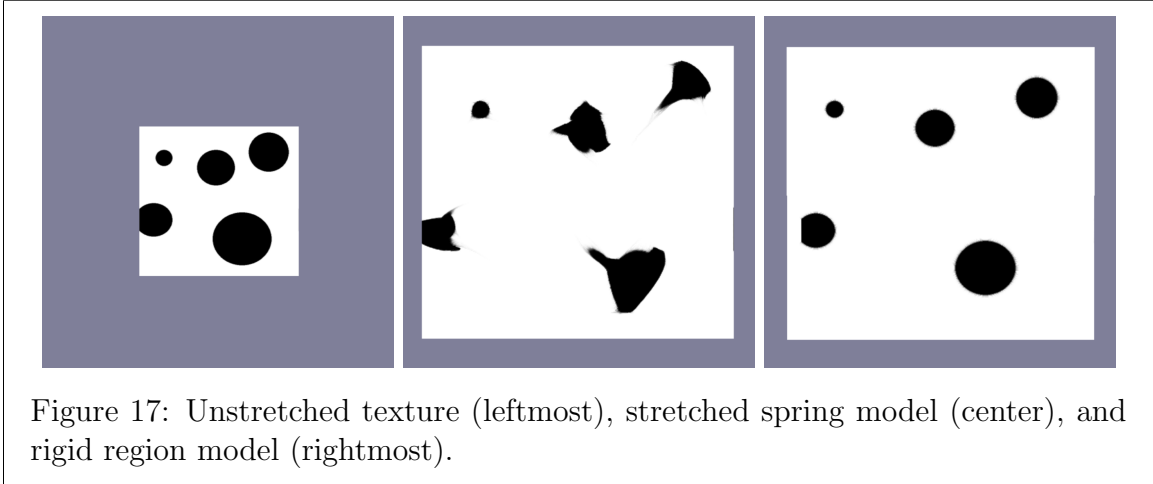
where x_i, x_j are neighbors. Therefore, A is not positive definite, and therefore has infinite condition number [7]. This means that conjugate gradient is not guaranteed to converge in a finite number of steps. Of course if there is no rigid region, then we can immediately return $x = [0, 0, \dots, 0]$ as the solution since there should be no stretching of any vector. However if the rigid region is small, then A will be approximately A' and thus have a large condition number [5]. Therefore, the algorithmic complexity of conjugate gradient in `RigidRegionModel` can be large. Based on the definition of A , it is possible to compute bounds on $\sqrt{\kappa}$, but these will not provide a good understanding of the effective running time of the algorithm. Since conjugate gradient is descent based, it is not necessary to run it until complete convergence. When computing the examples in this project, running conjugate gradient for 300 steps was sufficient for a highly accurate solution. Therefore, the algorithm is effectively $O(n)$, but with a potentially large coefficient of n depending on the number of steps needed for accurate convergence. A better understanding of the practical algorithmic complexity of the algorithm would require more testing, since the complexity bounds for conjugate gradient do not apply well when the matrix A is barely positive definite.



4.4 Analysis of Rigid Region Model

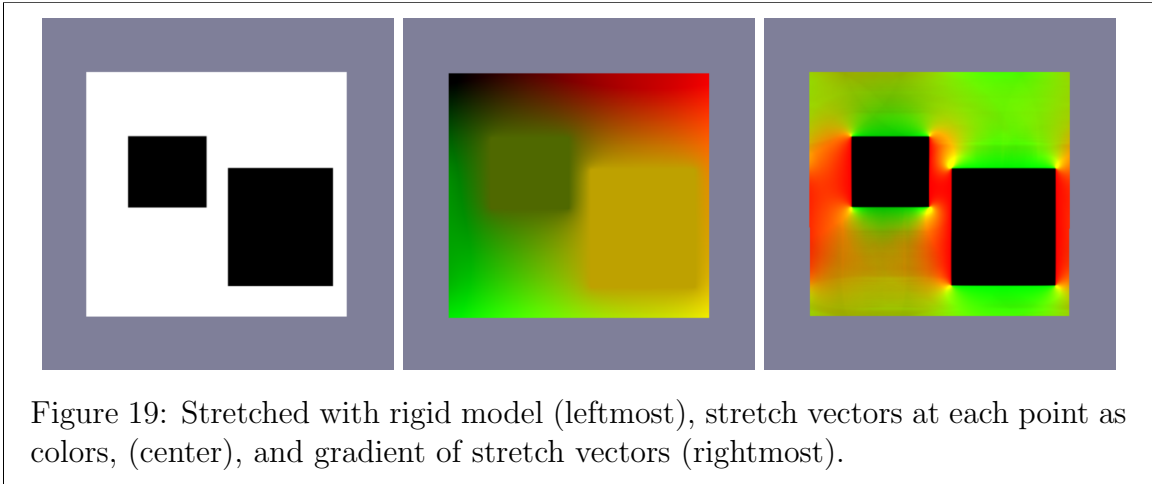
Figure 15, illuminates one limitation of the rigid region model compared to the spring one: the rigid region model is only able to support fully elastic and fully non-elastic regions. Therefore, the gray (moderately rigid) areas of the stretch texture are treated as fully elastic. Notice that the black (rigid) regions are the same size in each of the unstretched, spring, and rigid region examples. However, while the white regions are visibly stretched more than the dark gray and light gray regions in the spring model, they are stretched proportionally to the gray regions in the rigid region model (since both are treated as fully elastic).

Now consider the comparison in figure 16 of the spring and rigid region models applied to a texture with two rigid rectangles. Because the rigid region model explicitly preserves the structure of connected rigid regions, each rectangle's shape and relative position is preserved. The difference between the two models on this image empha-



sizes the inability for the spring model to “see” two dimensional geometry, while the rigid region model encodes geometry by explicitly preserving two dimensional regions when stretched.

The texture with multiple rigid circles in figure 17 gives a more complex example showing how the rigid region model improves upon the spring model. Each circle is preserved in its shape and kept the same size as the texture stretches. In figure 18, we see how while the rigid region model preserves the shape of rigid portions of the texture (in black), no guarantees are made about the shape of the remainder of the texture. The gray shapes illuminate this lack of structure, in particular, the warping of the pentagons. Another feature illuminated in this example is how adjacency is



defined while computing the stretch vectors. As the stretch is applied, the gray area in the lower right hand side “spills over” to the upper left hand corner, since in the matrix A we specify that points on opposite sides of the texture are adjacent. Thus, the rigid region model behaves best when the texture is continuous along opposite sides.

Consider figure 19 which shows the behavior of the rigid model on the example of two rigid rectangles. In the second image, the stretch vectors $V(p)$ are plotted at each point by color. In particular, the red value at a point p represents the x component $V_x(p)$ and the green value represents the y component $V_y(p)$. In the rightmost image, the same color scheme is used for $\Delta V(p)$, so the red color represents $\partial V/\partial x$ and the green color represents $\partial V/\partial y$. Notice that inside each of the rigid regions, the stretch vectors are constant. This is also shown by the fact that the rigid regions are black in the figure on the right showing the derivatives of the stretch vectors. Notice that in order to preserve the rigid regions and continuity, the x derivatives of the vectors are large (red) between the rectangles, and similarly in the y direction.

5 Conclusion

5.1 Analysis

The rigid region model is far more effective than the spring model. Despite being limited to only fully elastic or fully rigid regions, the rigid region model preserves the geometric structure of a texture, which the spring model fails to do even in simple cases. While the preprocessing of the rigid region model is more computationally expensive, this only needs to be computed once for each texture, and can be precomputed as a texture for any practical use. In addition to behaving well on rigid regions, the rigid region model also preserves the continuity of the remainder of the texture by minimizing the difference between adjacent stretch vectors.

5.2 Further Research

A first step for future research is to conduct more testing on realistic examples and understand how the current model performs. For the sake of this project, we have focused on the technical aspect of introducing varying stretching on a texture, rather than how the model we have developed performs on realistic textures. Therefore, in order to more accurately assess how the current model fails on practical examples, it would be necessary to first test the current model extensively.

One main limitation of the current model is that it does not necessarily extend well to complex triangular meshes. In particular, if two adjacent triangles are stretched different amounts, then their stretched texture will not necessarily be continuous along their boundary. This will make the texture look choppy and unrealistic at their intersection.

Coordinating stretch values across different triangles is difficult because it cannot be solely accomplished in the fragment and vertex shaders, which can only see a single triangle and pixel at a time, respectively. Therefore, solving this problem would require a more sophisticated approach which utilizes more complicated preprocessing

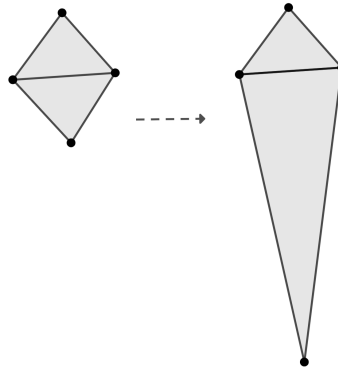


Figure 20: When adjacent triangles are stretched different amounts, the texture coordinates on each can be stretched differently along their boundary.

or other shaders on the GPU pipeline.

Parallax mapping is a technique to create the illusion of 3D geometry on a flat triangle by modifying texture coordinates inside of the fragment shader [2]. While parallax mapping has a different end goal from creating the illusion of variable stretch-

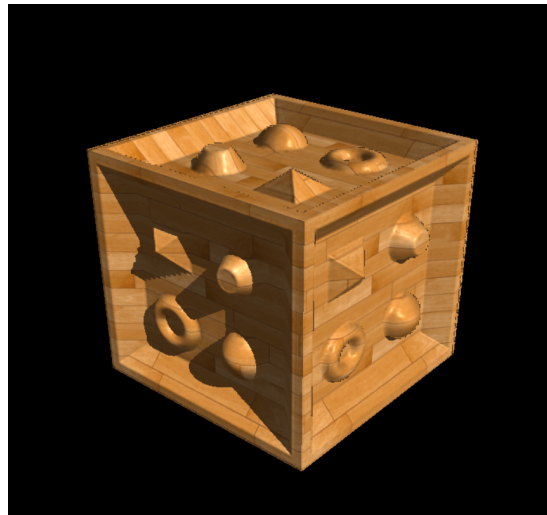


Figure 21: An example of parallax mapping implemented for the final project of CS 5610, Interactive Computer Graphics. The only triangles rendered are two per face of the cube. The illusion of depth is achieved by modifying texture coordinates inside the fragment shader.

ing, the methods are similar. The original parallax algorithm has been improved in a number of novel ways, including a more effective approximation algorithm [4] as well as using more nuanced geometry to reduce artifacts caused by the approximation [6]. In each of these cases, it is possible that the implementation of rigid region model in the fragment shader can be improved upon in similar ways. A more in depth literature review of existing techniques for parallax mapping could yield useful techniques which can be applied to variable stretch textures.

6 References

- [1] A. Galántai. The theory of newton’s method. *Journal of Computational and Applied Mathematics*, 124(1-2):25–44, 2000.
- [2] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. In *Proceedings of ICAT*, volume 2001, pages 205–208, 2001.
- [3] Samuel J. Ling, Jeff Sanny, and William Moebs. *Work and Kinetic Energy*. OpenStax, Rice University, 2017.
- [4] Morgan McGuire and Max McGuire. Steep parallax mapping. *I3D 2005 Poster*, pages 23–24, 2005.
- [5] Cleve Moler. What is the condition number of a matrix?, Jul 2017.
- [6] Fabio Policarpo and Manuel M Oliveira. Relaxed cone stepping for relief mapping. *GPU gems*, 3:409–428, 2007.
- [7] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.
- [8] Peter Shirley, Michael Ashikhmin, and Steve Marschner. *Fundamentals of computer graphics*. AK Peters/CRC Press, 2009.
- [9] Cem Yuksel. Intro to graphics 07 - gpu pipeline, Feb 2021.

Name of Candidate: Emil Geisler

Date of Submission: May 1, 2023