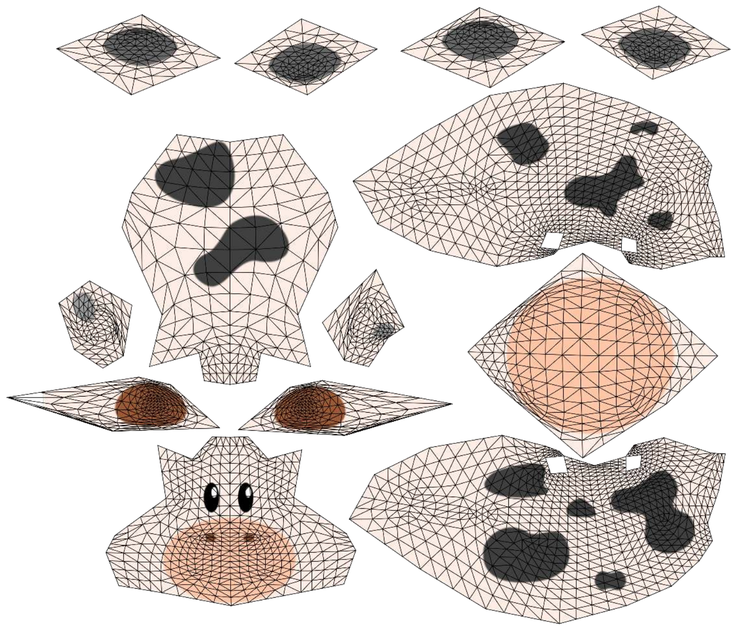
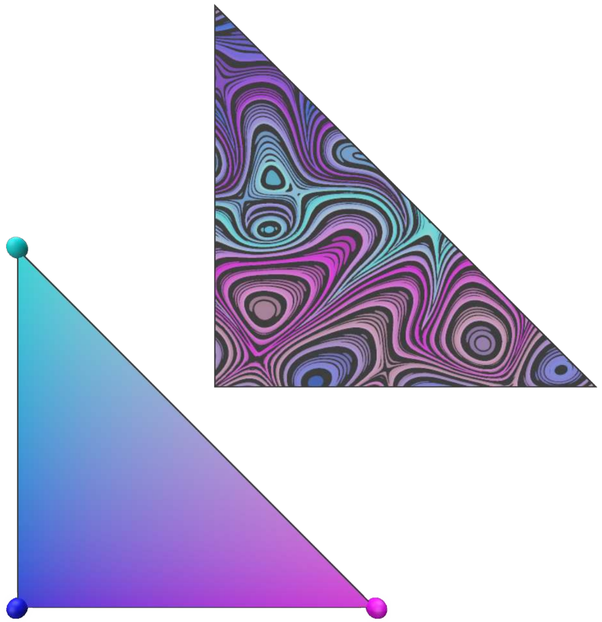
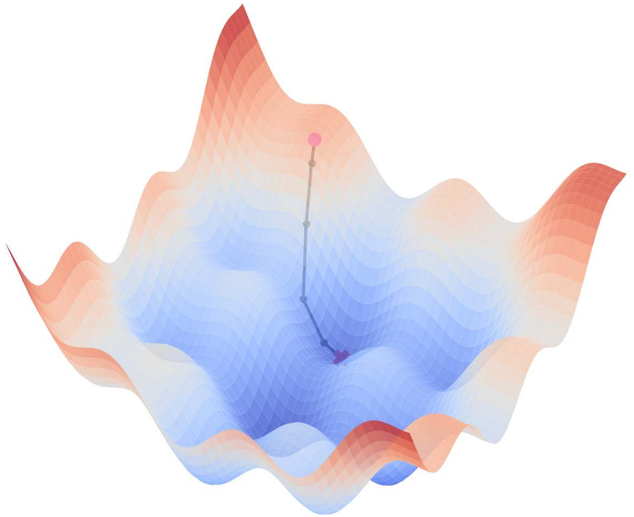
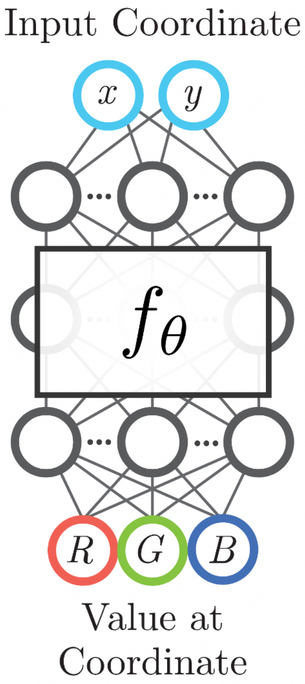
Texture Maps and Optimization









Dale Decatur

SGI 2026

About me

About me

PhD @ UChicago (advised by Rana Hanocka)

About me

PhD @ UChicago (advised by Rana Hanocka)

Research Interests:

About me

PhD @ UChicago (advised by Rana Hanocka)

Research Interests:

- 3D computer vision & geometry processing using 2D supervision

About me

PhD @ UChicago (advised by Rana Hanocka)

Research Interests:

- 3D computer vision & geometry processing using 2D supervision

Outside of Research:

About me

PhD @ UChicago (advised by Rana Hanocka)

Research Interests:

- 3D computer vision & geometry processing using 2D supervision

Outside of Research:

- ultimate frisbee

About me

PhD @ UChicago (advised by Rana Hanocka)

Research Interests:

- 3D computer vision & geometry processing using 2D supervision

Outside of Research:

- ultimate frisbee
- running

About me

PhD @ UChicago (advised by Rana Hanocka)

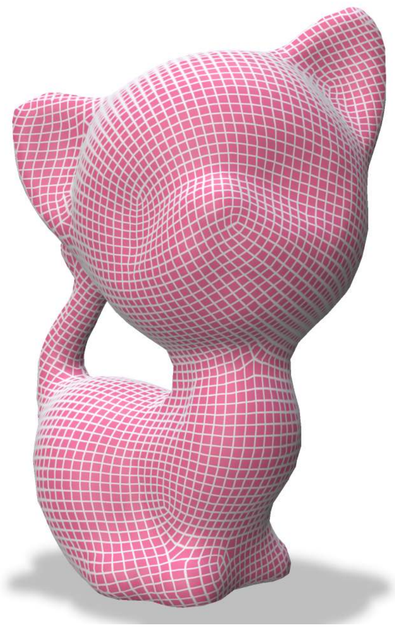
Research Interests:

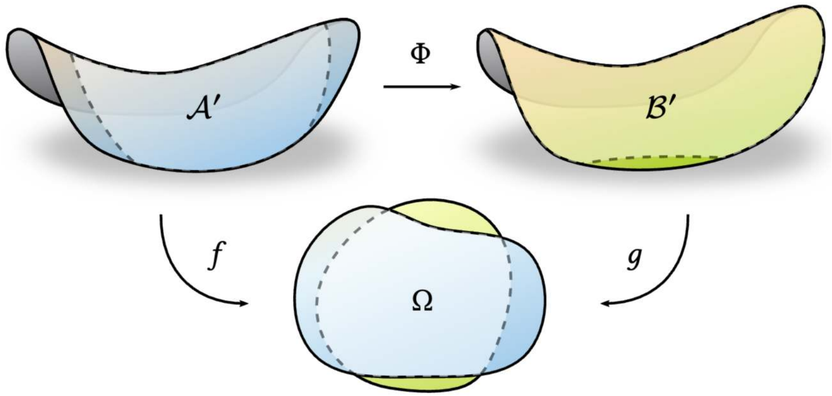
- 3D computer vision & geometry processing using 2D supervision

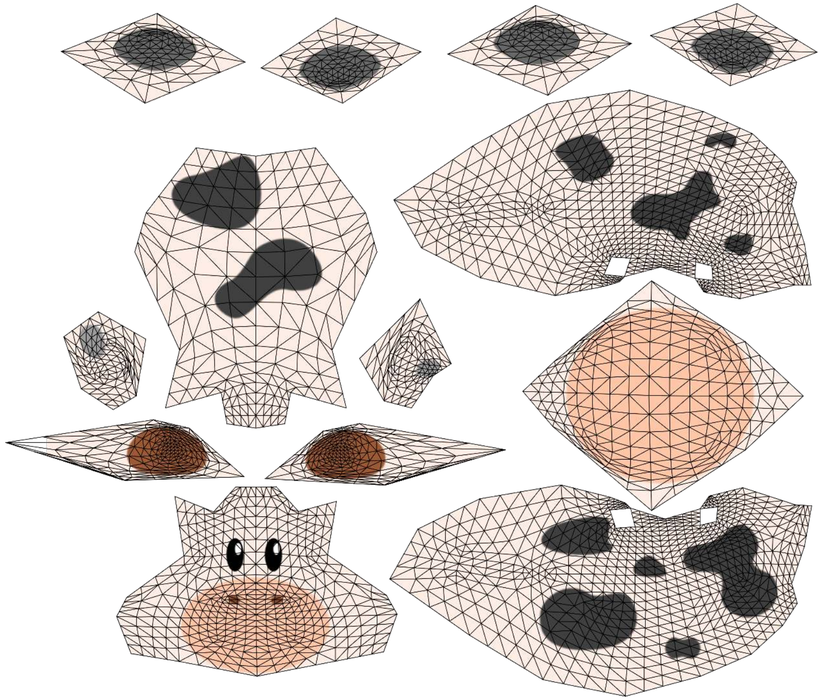
Outside of Research:

- ultimate frisbee
- running
- hiking

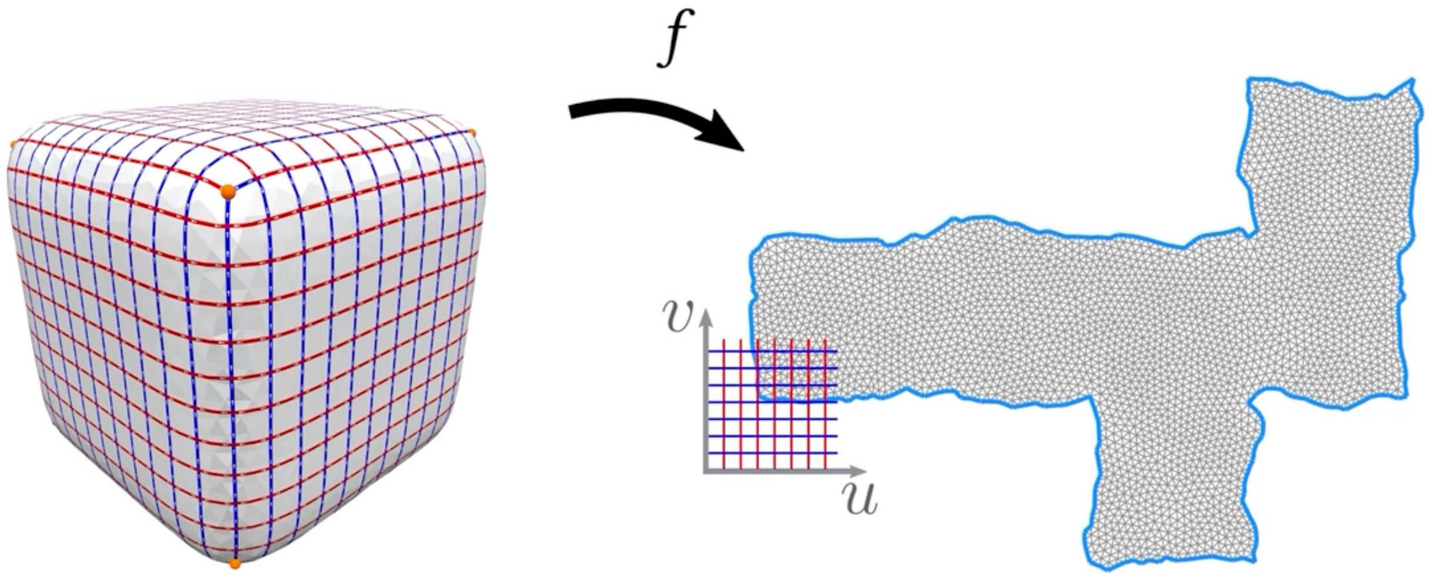
Parameterization Applications



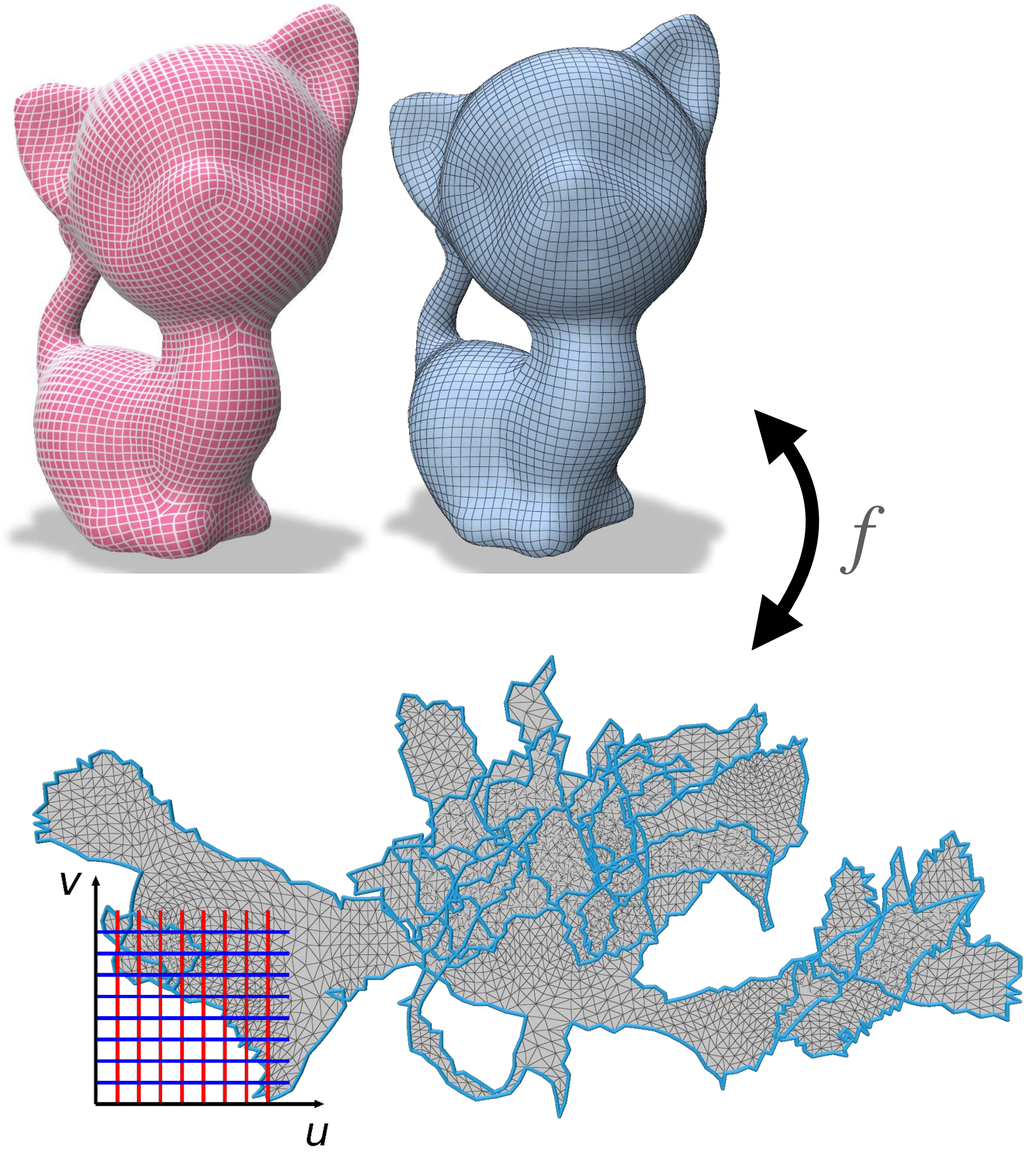




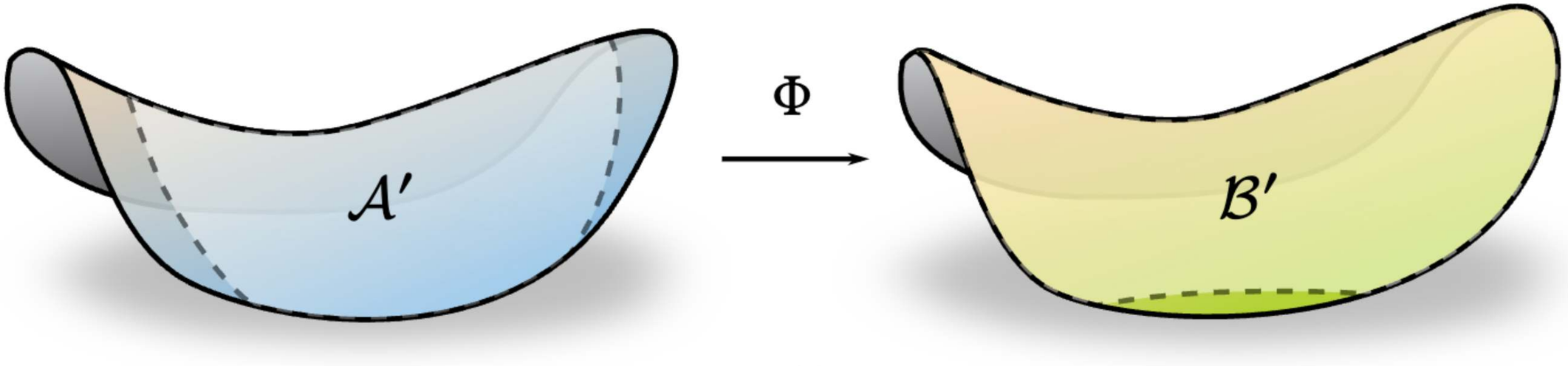
Parameterization Applications



SGP 2021: Maps Between Surfaces, Campen and Schmidt

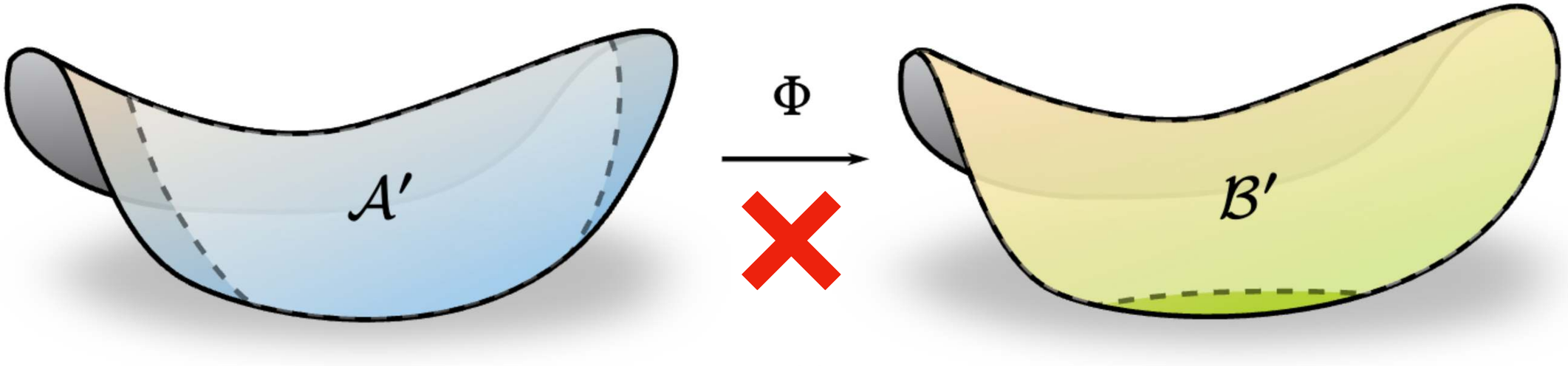


Parameterization Applications



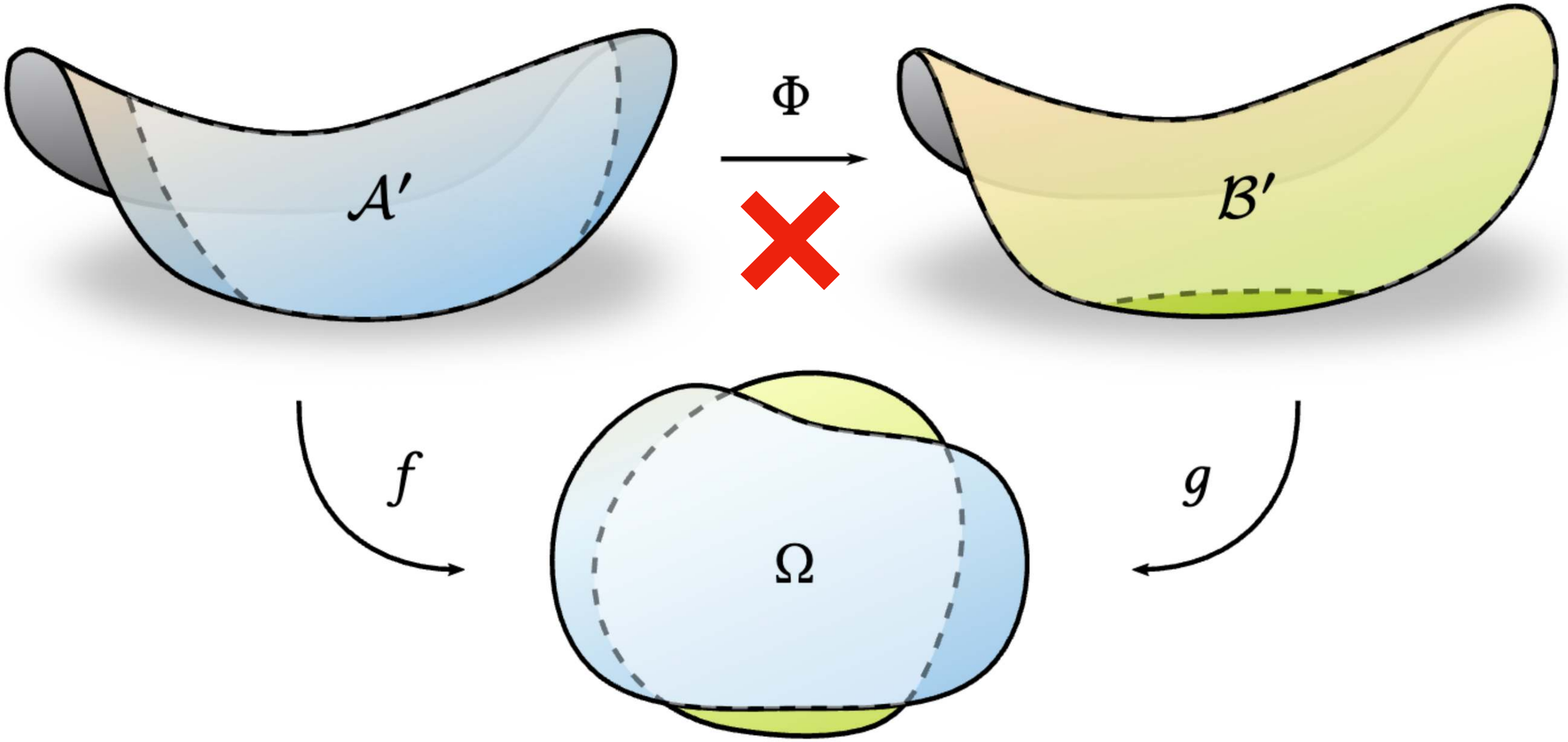
Distortion-Minimizing Injective Maps Between Surfaces, Schmidt et al., 2019

Parameterization Applications



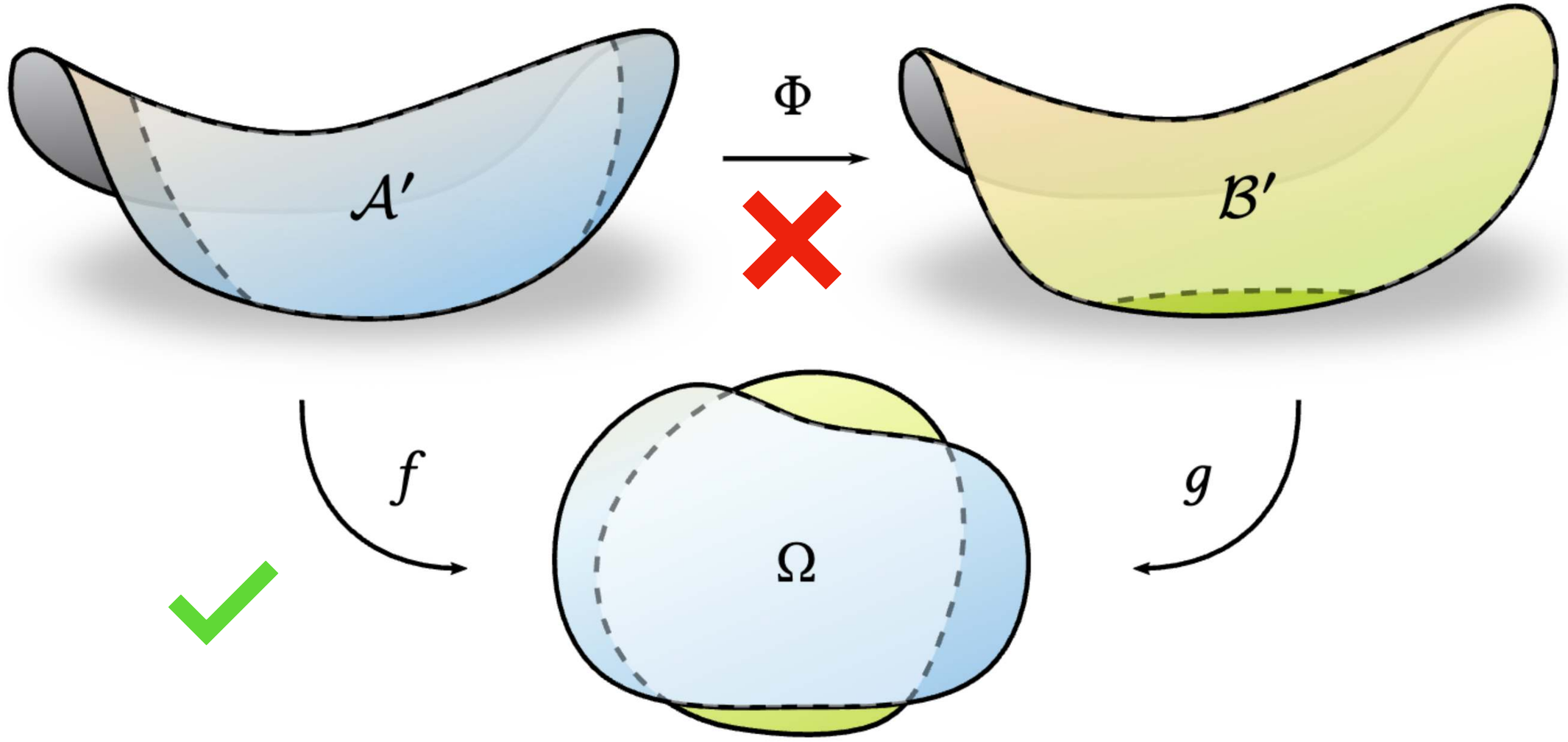
Distortion-Minimizing Injective Maps Between Surfaces, Schmidt et al., 2019

Parameterization Applications



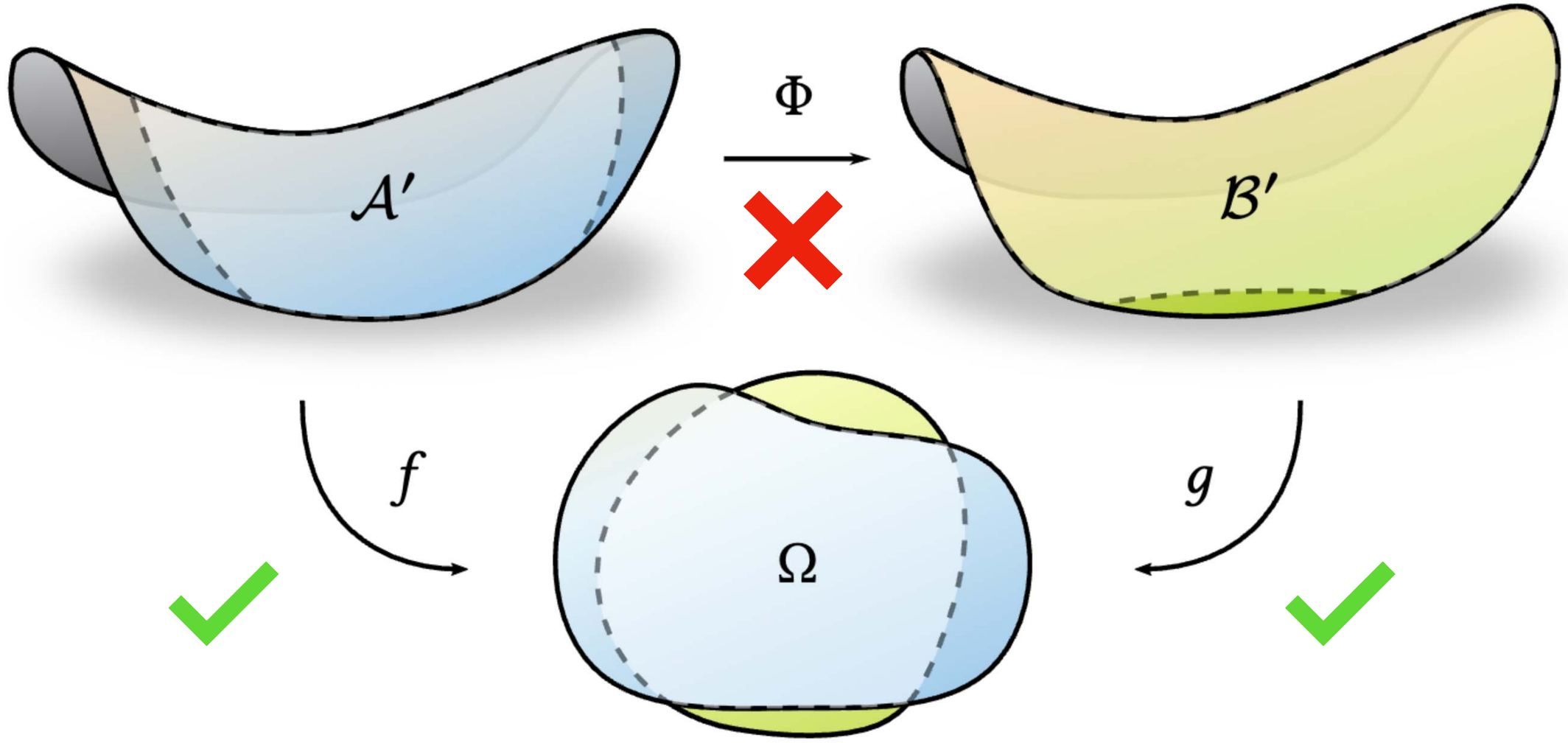
Distortion-Minimizing Injective Maps Between Surfaces, Schmidt et al., 2019

Parameterization Applications



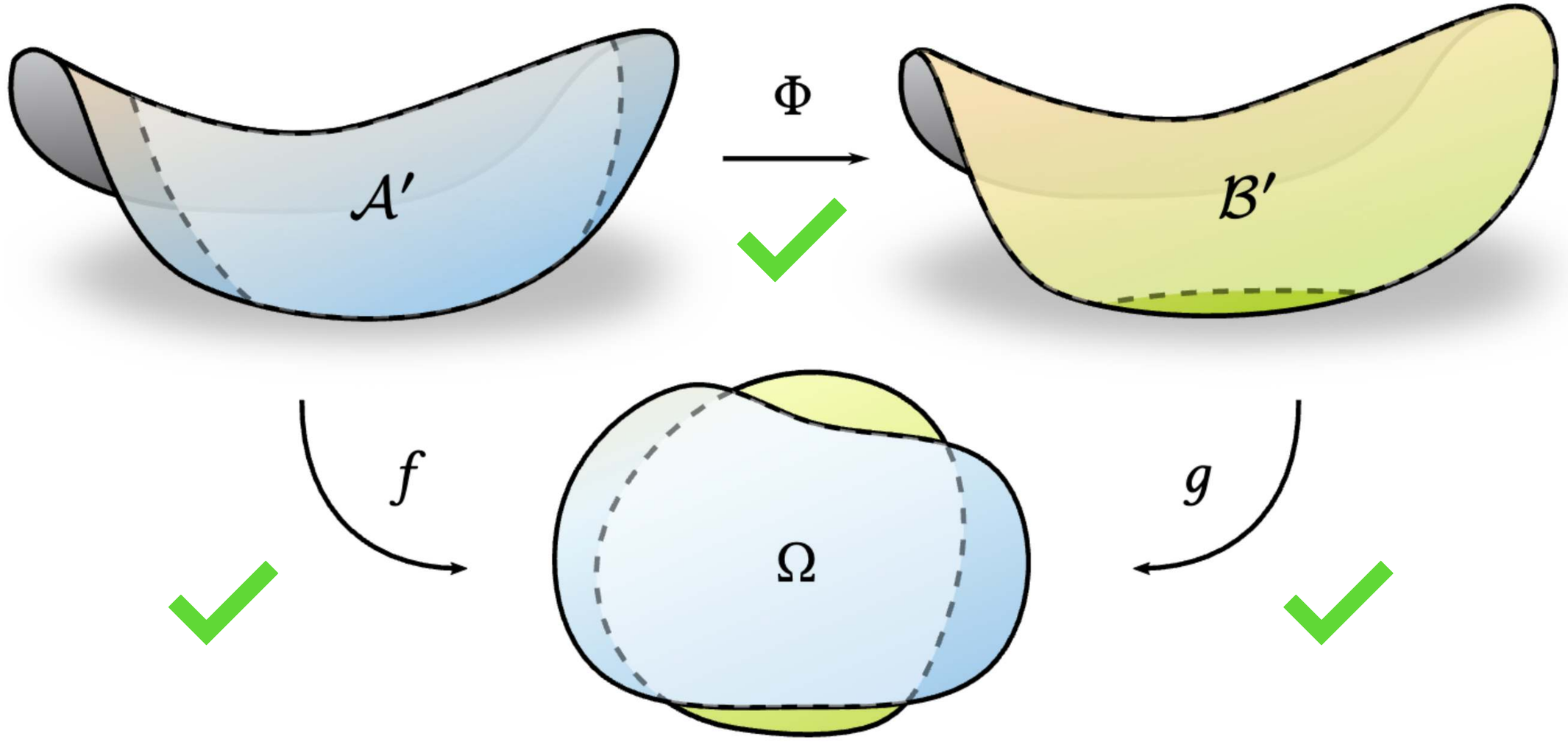
Distortion-Minimizing Injective Maps Between Surfaces, Schmidt et al., 2019

Parameterization Applications



Distortion-Minimizing Injective Maps Between Surfaces, Schmidt et al., 2019

Parameterization Applications



Distortion-Minimizing Injective Maps Between Surfaces, Schmidt et al., 2019

Parameterization Applications

Print Gallery, M.C. Escher, 1956

Artful Mathematics: The Heritage of M. C. Escher

Celebrating Mathematics Awareness Month

In recognition of the 2003 Mathematics Awareness Month theme "Mathematics and Art", this article brings together three different pieces about intersections between mathematics and the artwork of M. C. Escher. For more information about Mathematics Awareness Month, visit the website `http://mathforum.org/mam/03/`*. The site contains materials for organizing local celebrations of Mathematics Awareness Month.*

The Mathematical Structure of Escher's Print Gallery

B. de Smit and H. W. Lenstra Jr.

In 1956 the Dutch graphic artist Maurits Cornelis Escher (1898–1972) made an unusual lithograph with the title *Prentententoonstelling*. It shows a young man standing in an exhibition gallery, viewing a print of a Mediterranean seaport. As his eyes follow the quayside buildings shown on the print from left to right and then down, he discovers among them the very same gallery in which he is standing. A circular white patch in the middle of the lithograph contains Escher's monogram and signature.

What is the mathematics behind *Prentententoonstelling*? Is there a more satisfactory way of filling in the central white hole? We shall see that the lithograph can be viewed as drawn on a certain *elliptic curve* over the field of complex numbers and

M. C. Escher, "Prentententoonstelling." © 2003 Cordon Art B.V.-Baarn-Holland. All rights reserved.

Figure 1. Escher's lithograph "Prentententoonstelling" (1956).

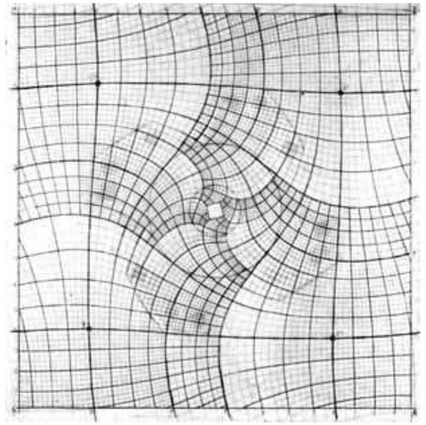
deduce that an idealized version of the picture repeats itself in the middle. More precisely, it contains a copy of itself, rotated clockwise by 157.6255960832… degrees and scaled down by a factor of 22.5836845286….

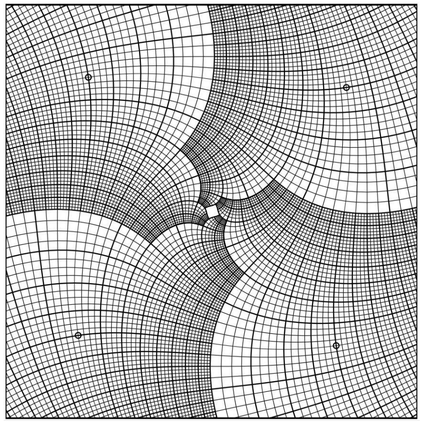
Escher's Method

The best explanation of how *Prentententoonstelling* was made is found in *The Magic Mirror of M. C. Escher* by Bruno Ernst [1], from which the following quotations and all illustrations in this section are taken. Escher started "from the idea that it must…be possible to make an annular bulge," "a cyclic expansion…without beginning or end." The realization of this idea caused him "some almighty headaches." At first, he "tried to put his idea into

B. de Smit and H. W. Lenstra Jr. are at the Mathematisch Instituut, Universiteit Leiden, the Netherlands. H. W. Lenstra also holds a position at the University of California, Berkeley. Their email addresses are `desmit@math.leidenuniv.nl` *and* `hwl@math.leidenuniv.nl`.

de Smit & Lenstra Jr, 2003





Parameterization Applications

Print Gallery, M.C. Escher, 1956

Artful Mathematics: The Heritage of M. C. Escher

Celebrating Mathematics Awareness Month

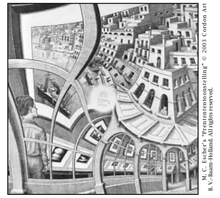
In recognition of the 2003 Mathematics Awareness Month theme "Mathematics and Art", this article brings together three different pieces about intersections between mathematics and the artwork of M. C. Escher. For more information about Mathematics Awareness Month, visit the website `http://mathforum.org/mam/03/`*. The site contains materials for organizing local celebrations of Mathematics Awareness Month.*

The Mathematical Structure of Escher's Print Gallery

B. de Smit and H. W. Lenstra Jr.

In 1956 the Dutch graphic artist Maurits Cornelis Escher (1898–1972) made an unusual lithograph with the title *Prentententoonstelling*. It shows a young man standing in an exhibition gallery, viewing a print of a Mediterranean seaport. As his eyes follow the quayside buildings shown on the print from left to right and then down, he discovers among them the very same gallery in which he is standing. A circular white patch in the middle of the lithograph contains Escher's monogram and signature.

What is the mathematics behind *Prententen-toonstelling*? Is there a more satisfactory way of filling in the central white hole? We shall see that the lithograph can be viewed as drawn on a certain *elliptic curve* over the field of complex numbers and



M. C. Escher, "Prententoonstelling." © 2003 Cordon Art B.V.-Baarn-Holland. All rights reserved.

Figure 1. Escher's lithograph "Prentententoonstelling" (1956).

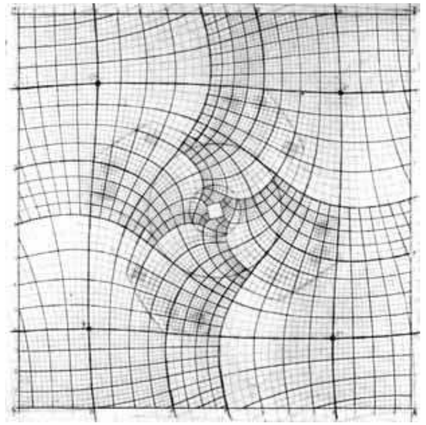
deduce that an idealized version of the picture repeats itself in the middle. More precisely, it contains a copy of itself, rotated clockwise by 157.6255960832… degrees and scaled down by a factor of 22.5836845286….

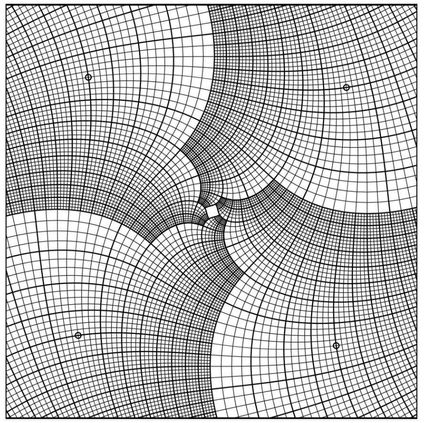
Escher's Method

The best explanation of how *Prentententoonstelling* was made is found in *The Magic Mirror of M. C. Escher* by Bruno Ernst [1], from which the following quotations and all illustrations in this section are taken. Escher started "from the idea that it must…be possible to make an annular bulge," "a cyclic expansion…without beginning or end." The realization of this idea caused him "some almighty headaches." At first, he "tried to put his idea into

B. de Smit and H. W. Lenstra Jr. are at the Mathematisch Instituut, Universiteit Leiden, the Netherlands. H. W. Lenstra also holds a position at the University of California, Berkeley. Their email addresses are `desmit@math.leidenuniv.nl` *and* `hwl@math.leidenuniv.nl`.

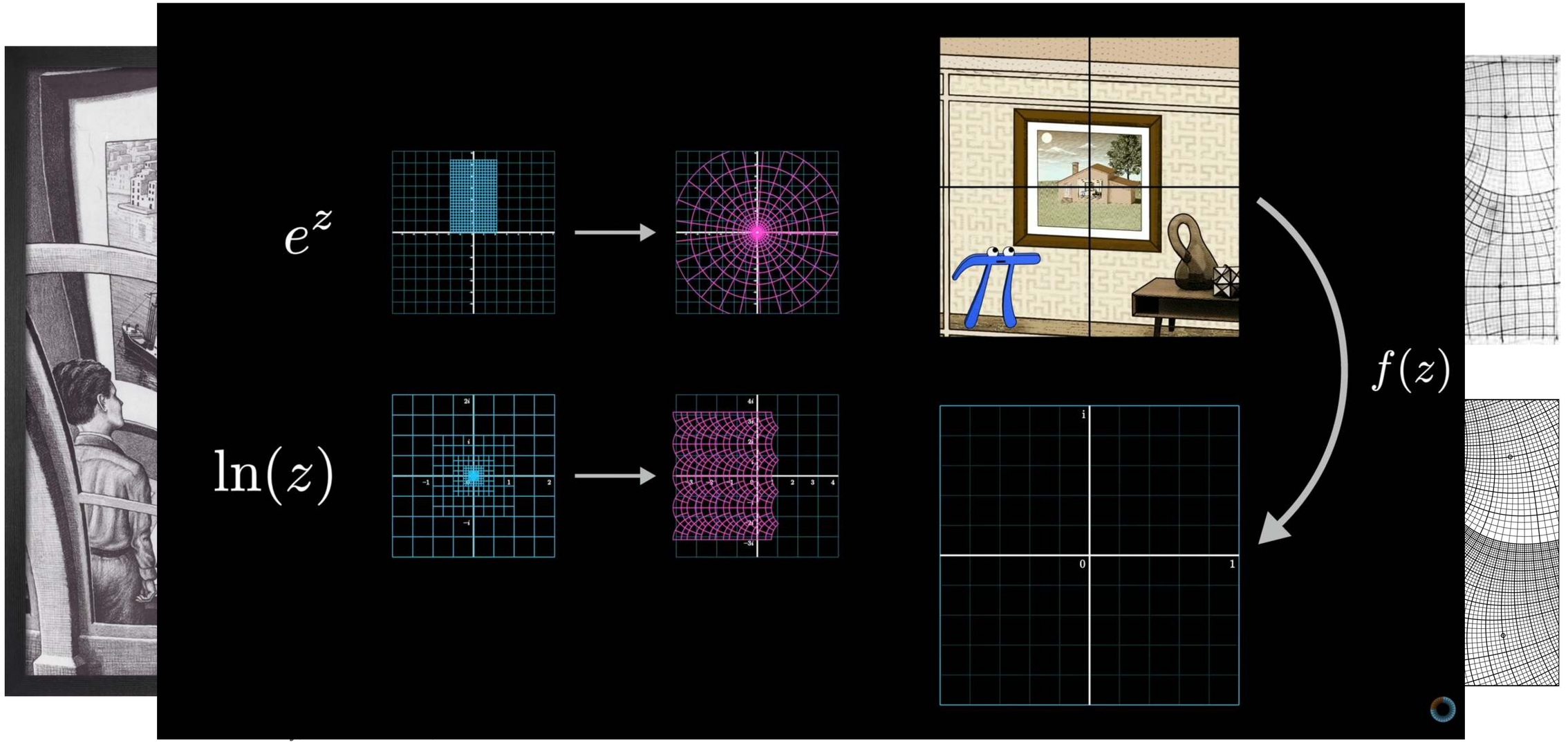
de Smit & Lenstra Jr, 2003





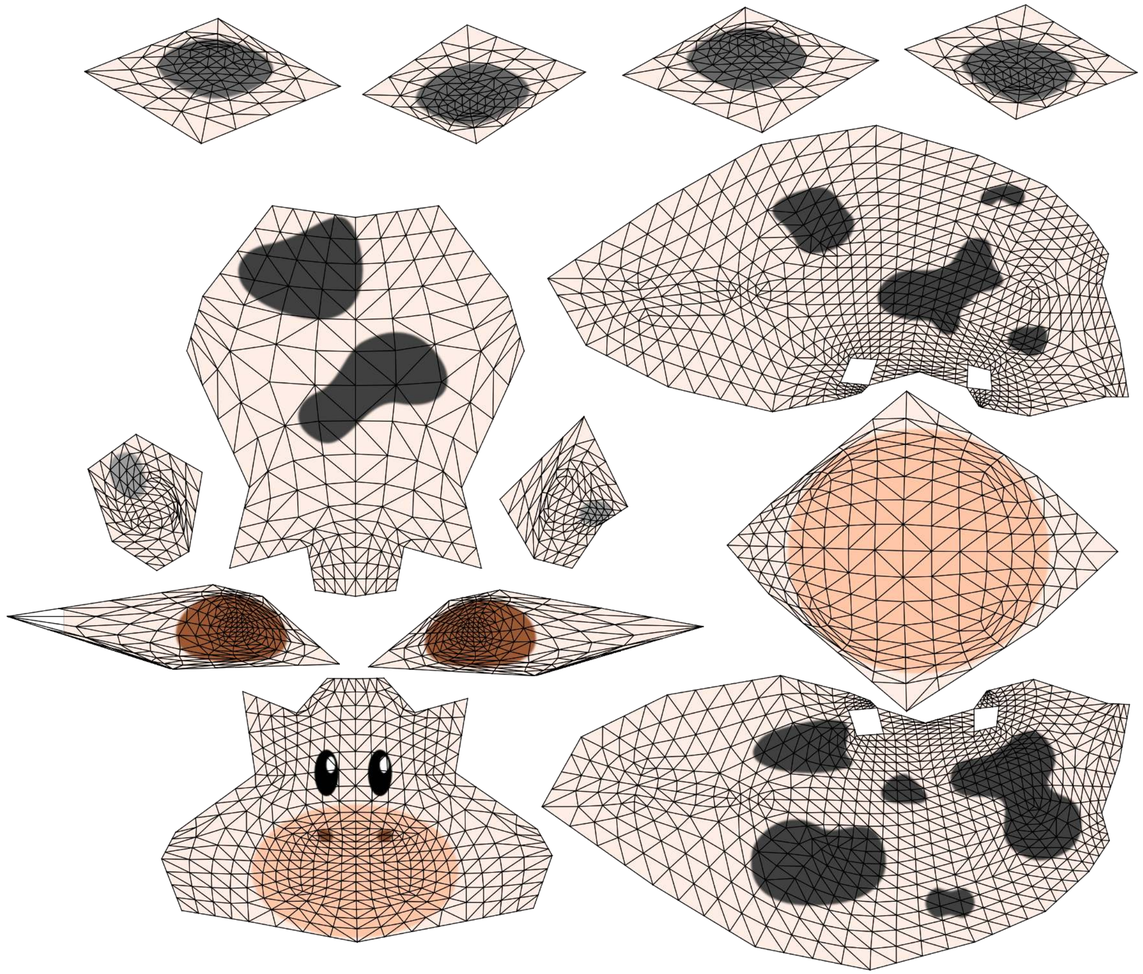
https://www.youtube.com/watch?v=ldxFjLJ3rVY

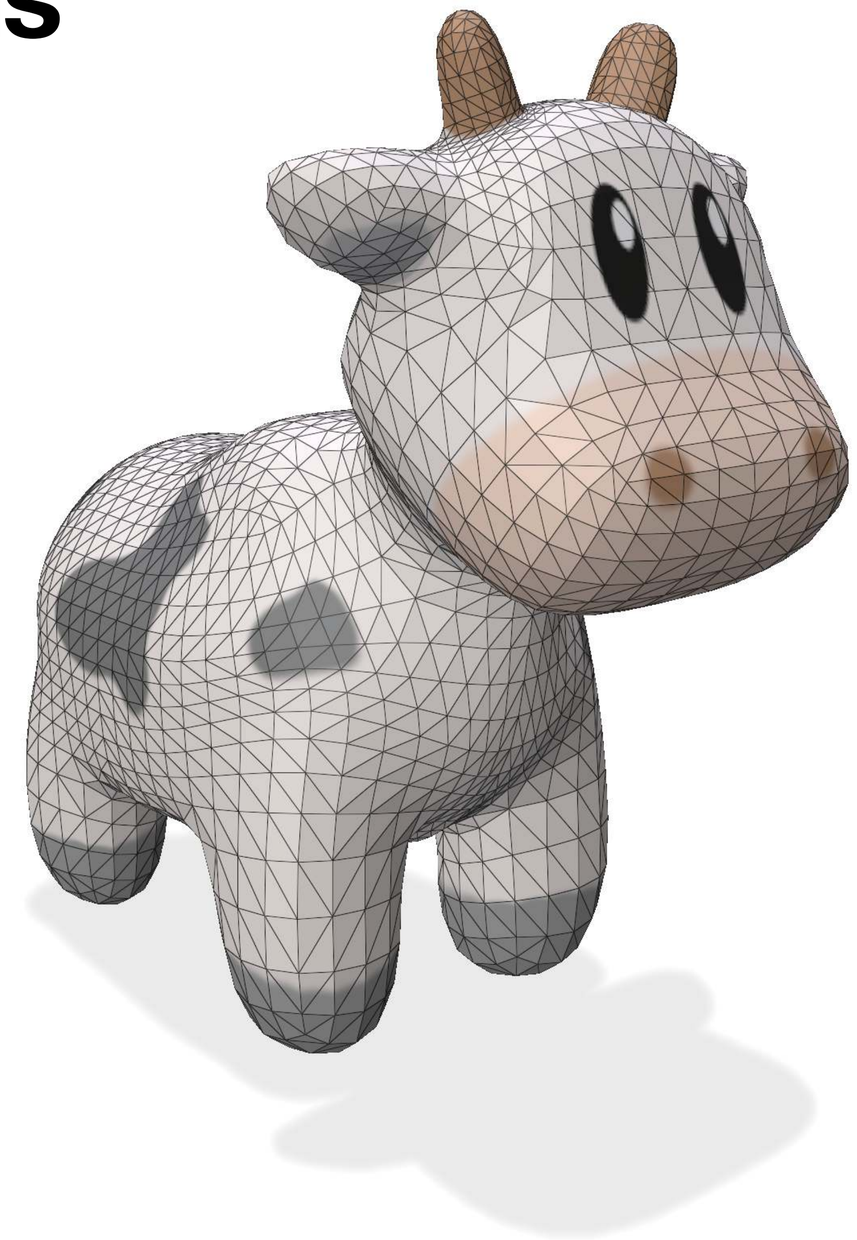
Parameterization Applications



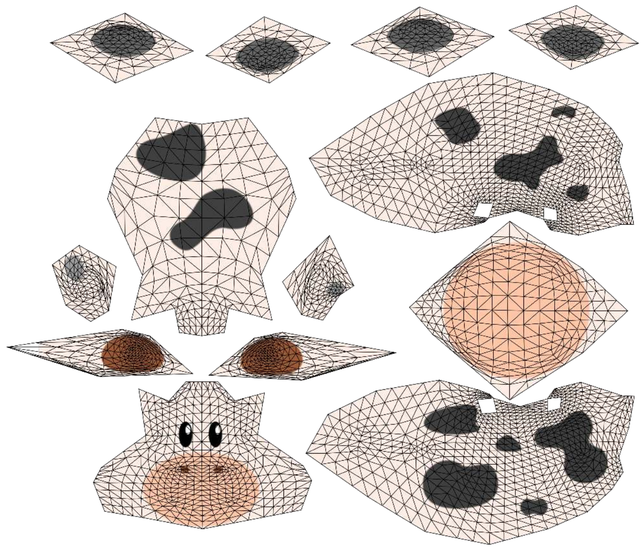
https://www.youtube.com/watch?v=ldxFjLJ3rVY

Parameterization Applications



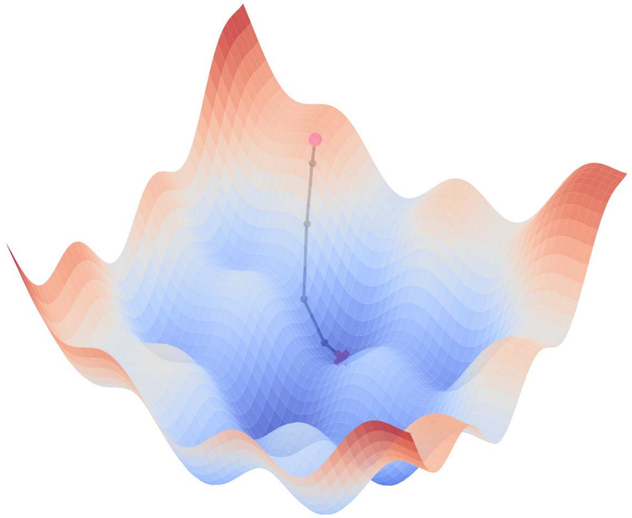


Agenda



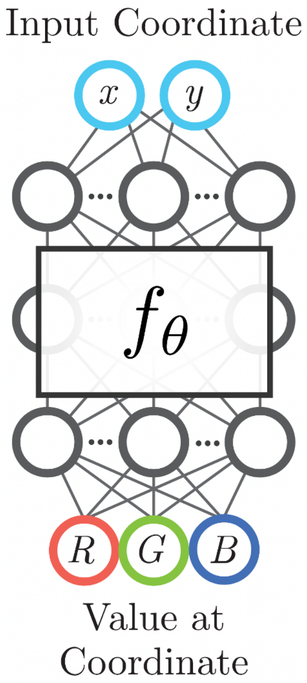
Represent Colors
w/ Texture Maps

Agenda



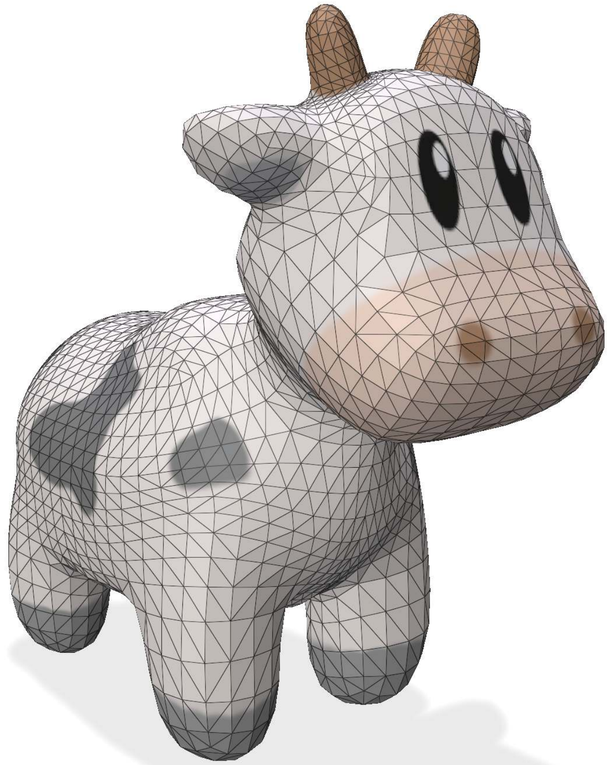
Gradient
Descent

Agenda



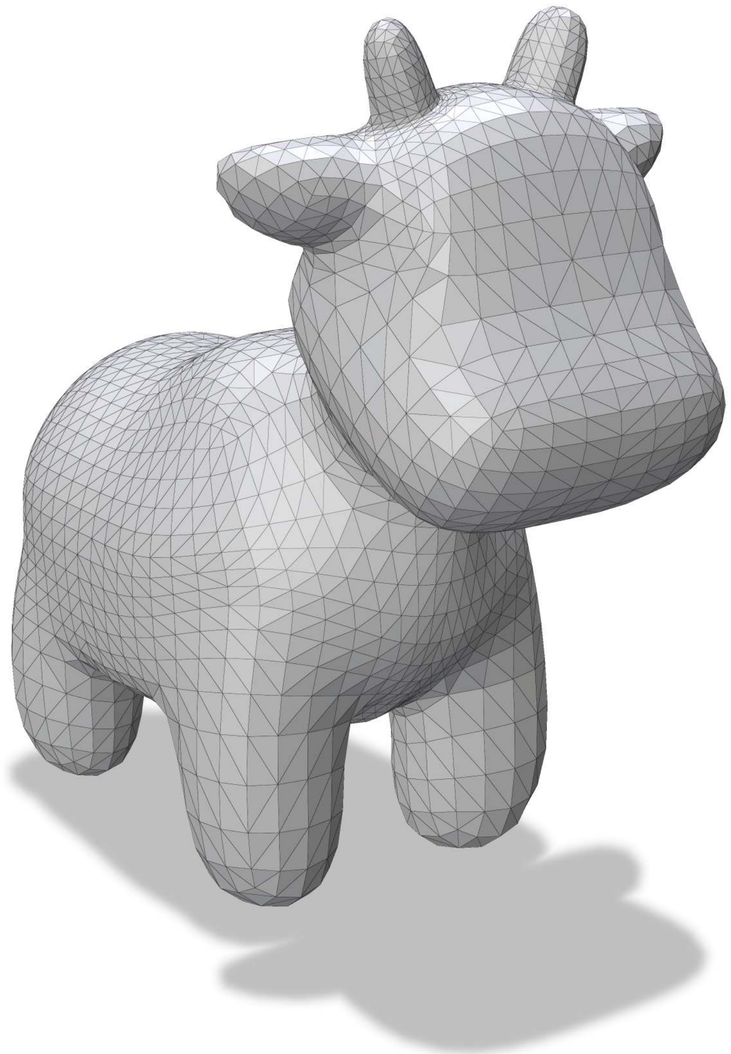
Smooth Representations

Agenda

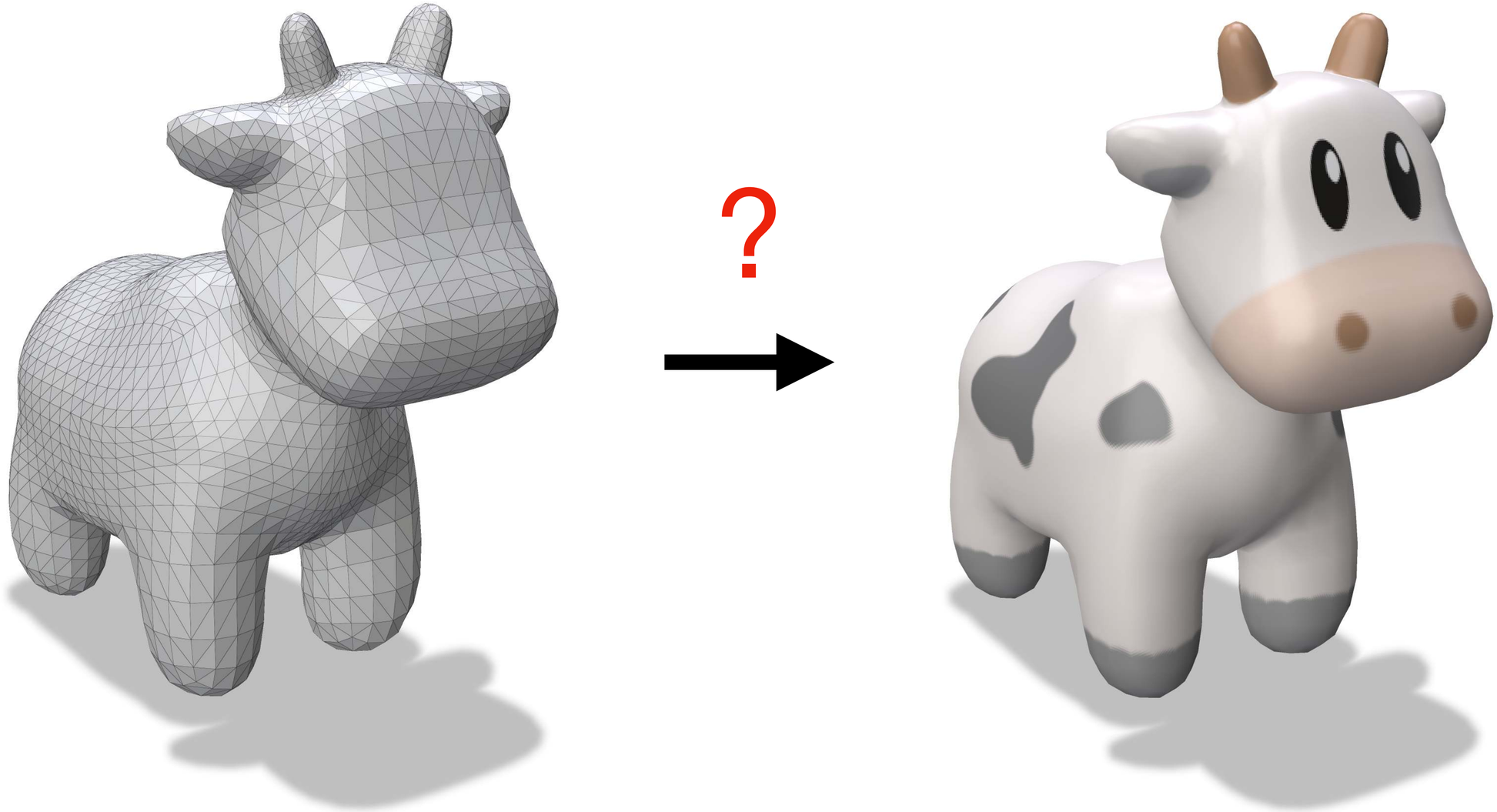


Optimize
Textures

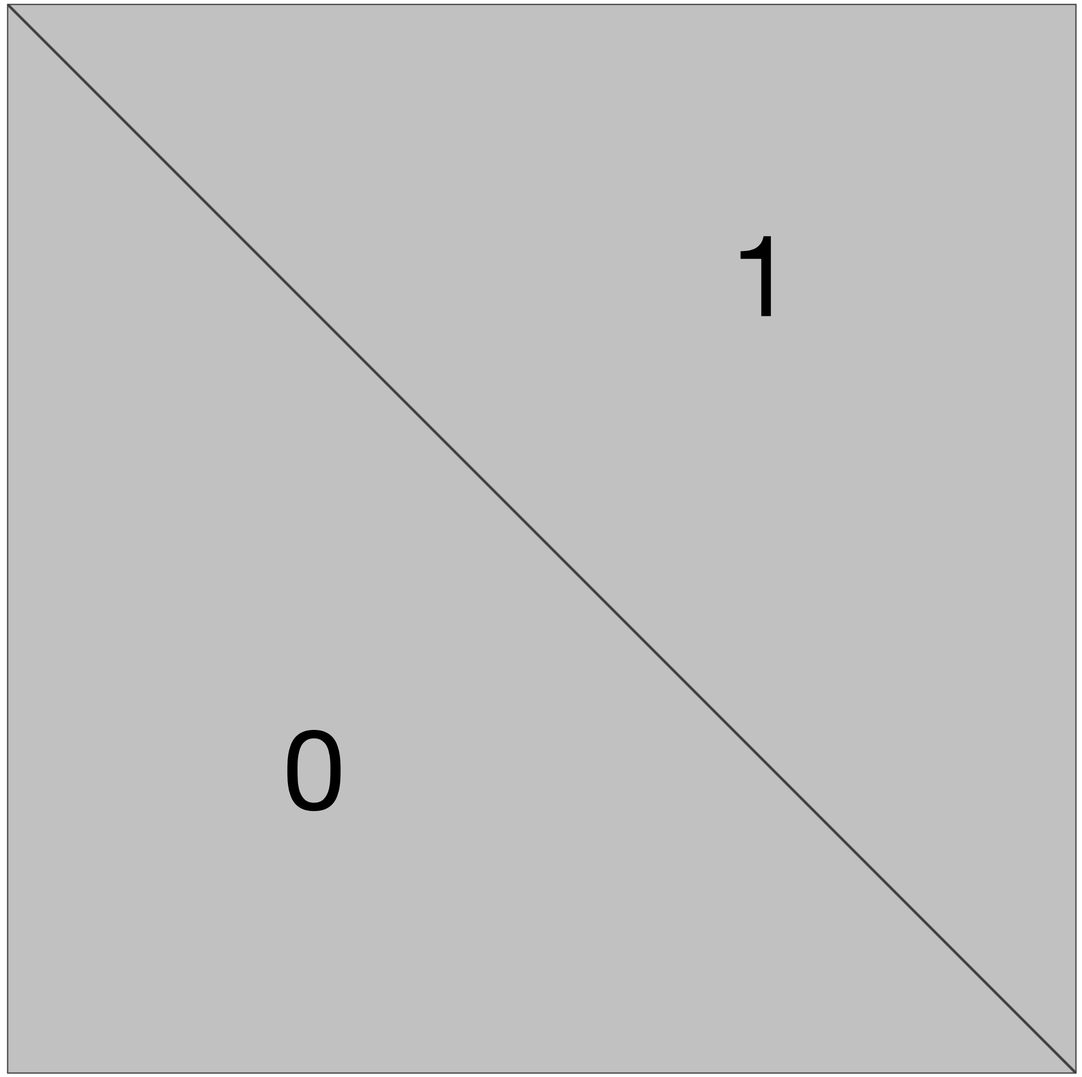
How can we color a 3D shape?

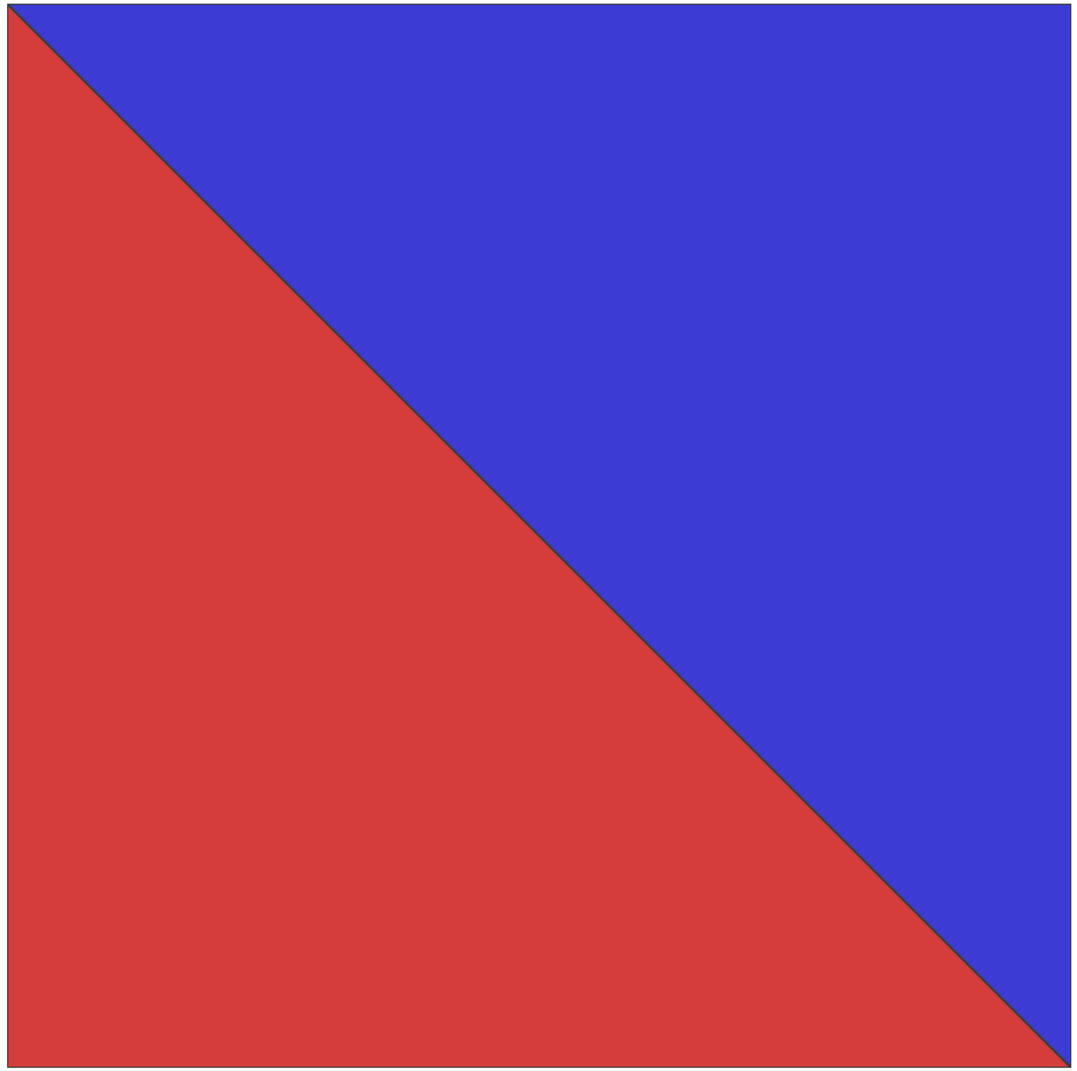


How can we color a 3D shape?



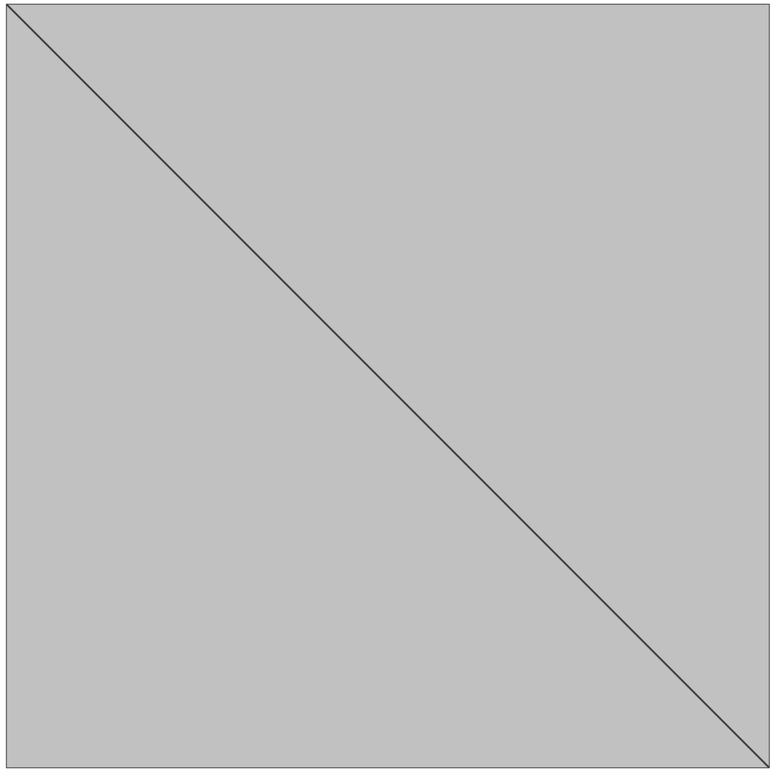
Face Coloring

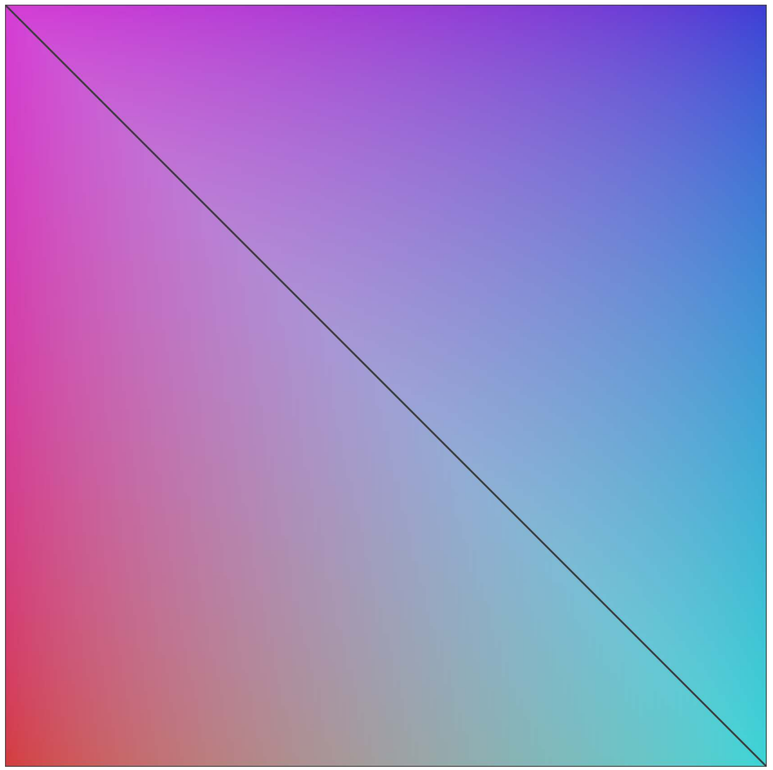




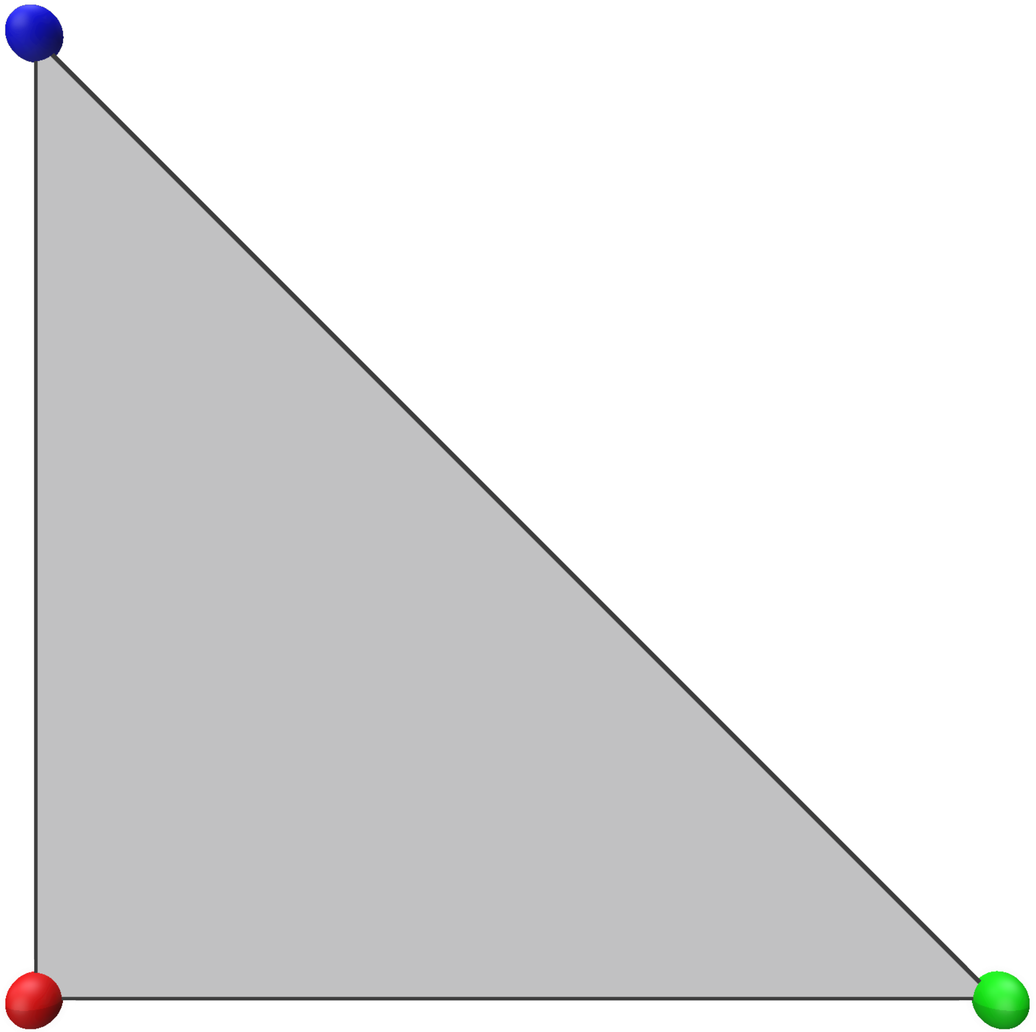
Face Coloring

No sub-triangle variation!

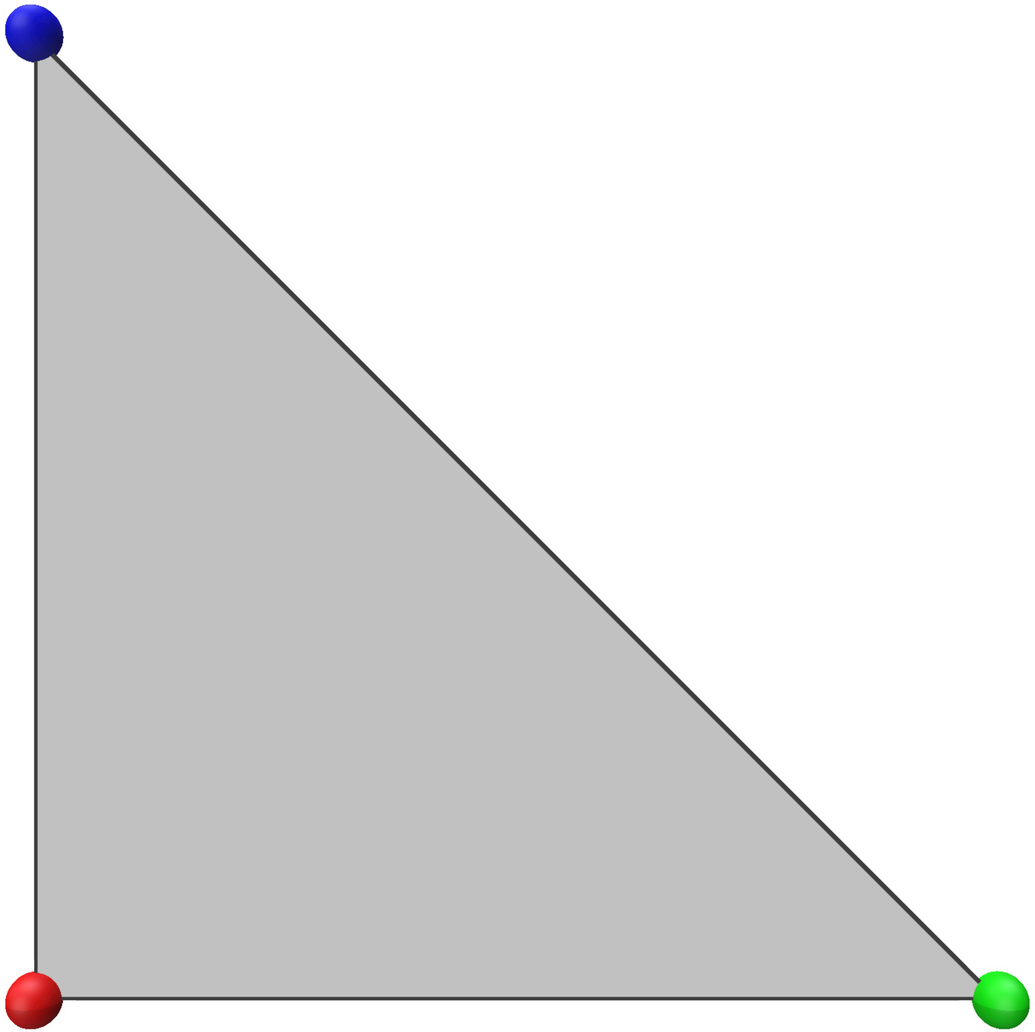


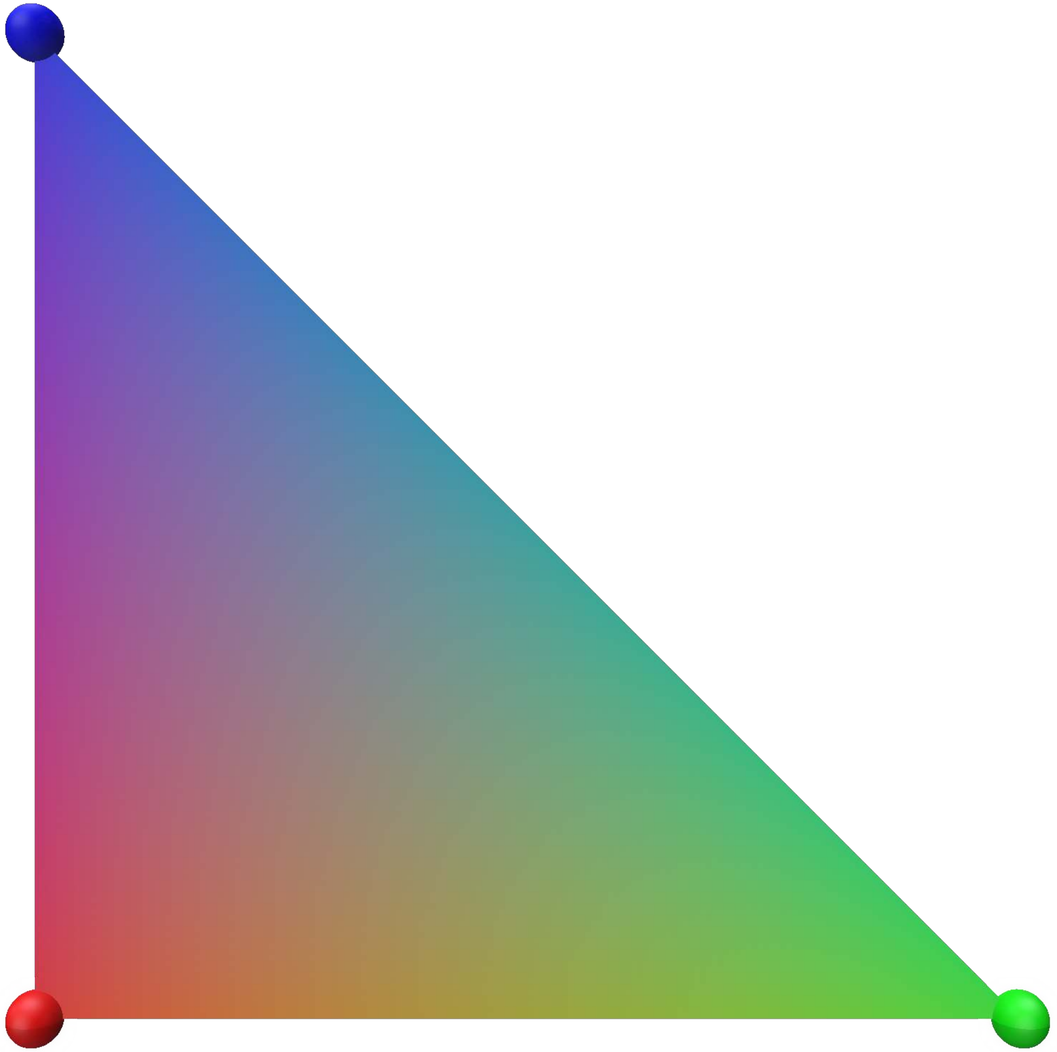


Vertex Coloring



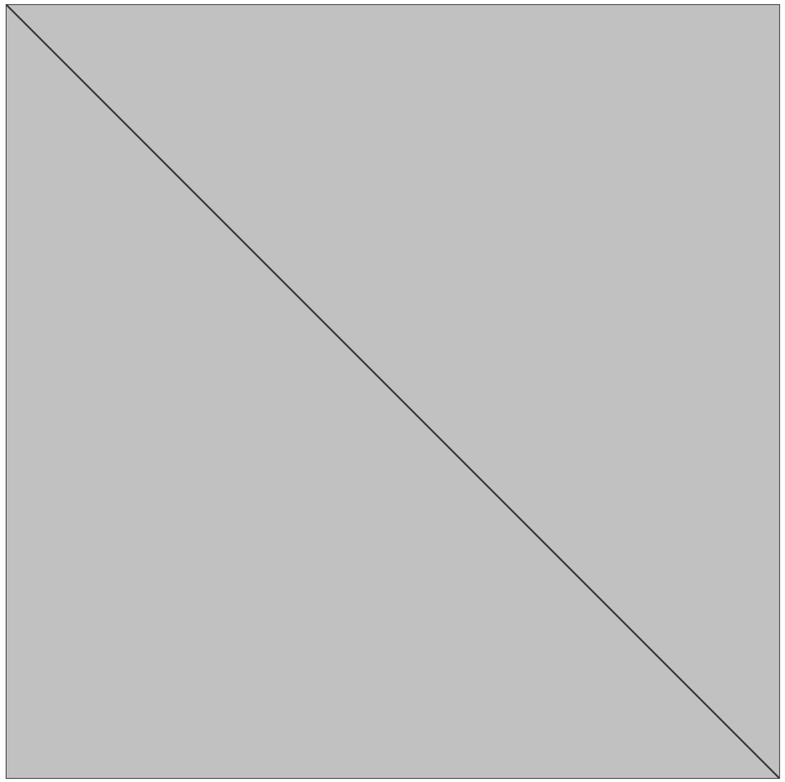
Vertex Coloring





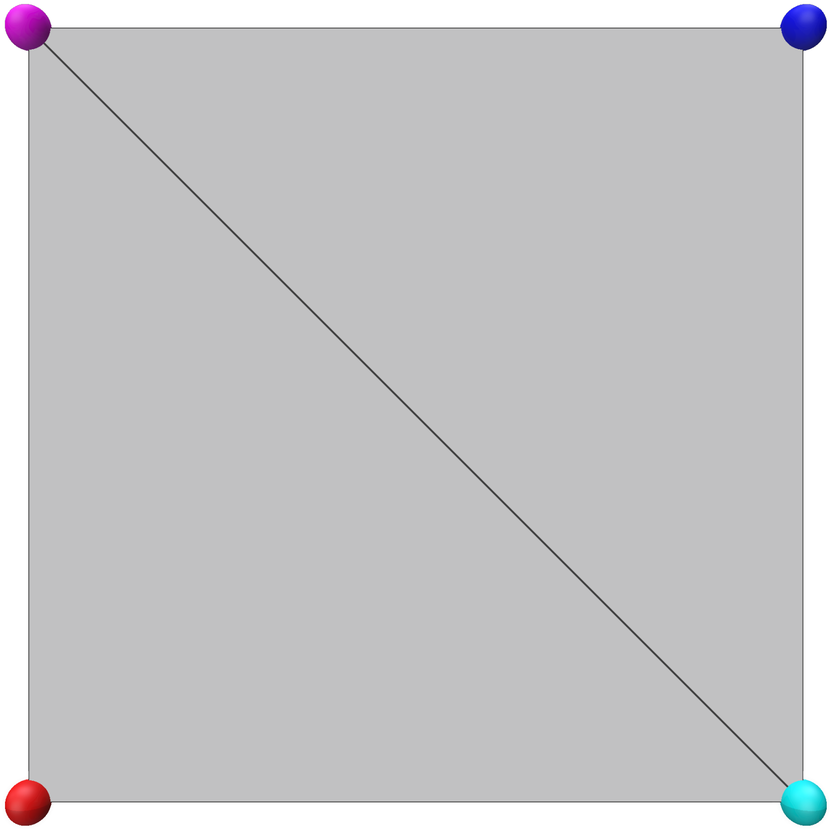
Vertex Coloring

Can produce color variation within the triangle!



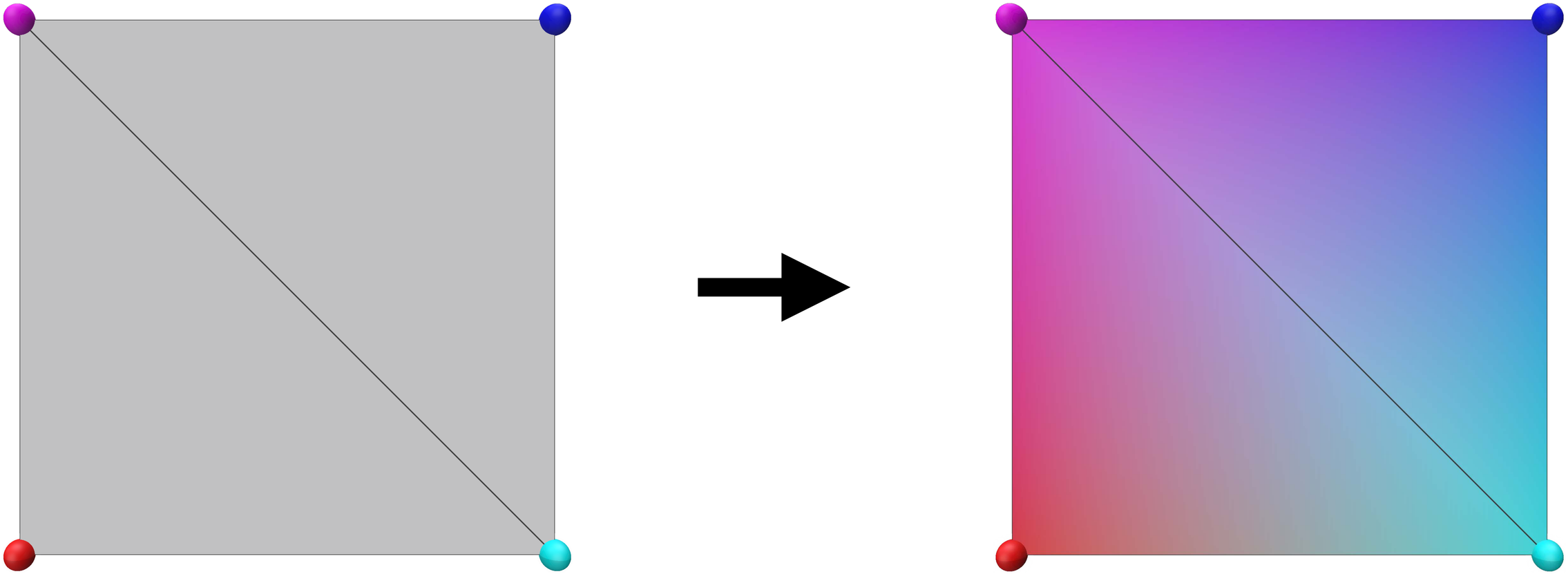
Vertex Coloring

Can produce color variation within the triangle!



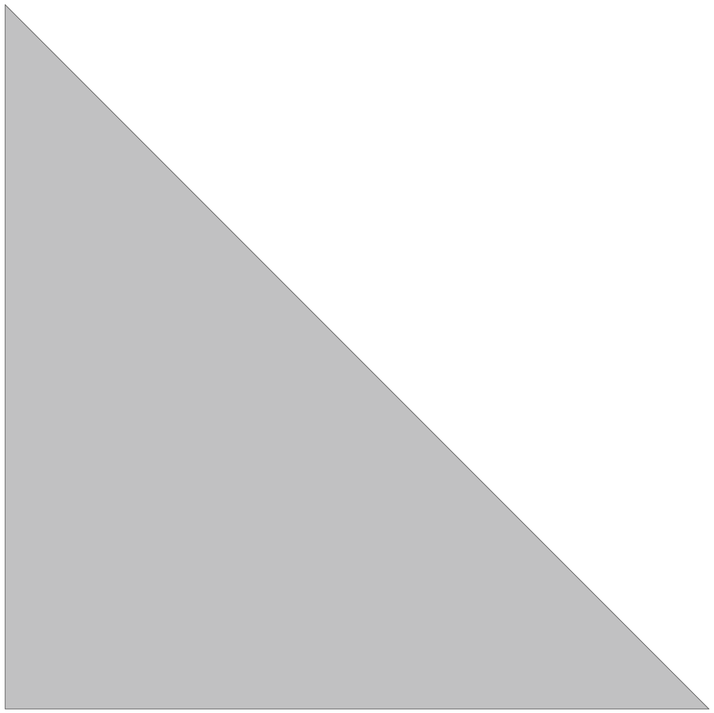
Vertex Coloring

Can produce color variation within the triangle!



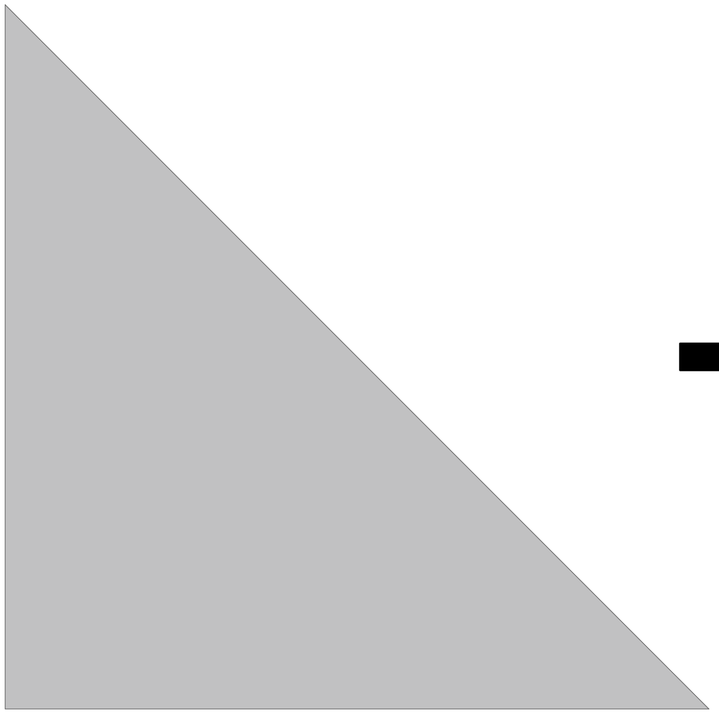
Vertex Coloring

Limited by mesh resolution!

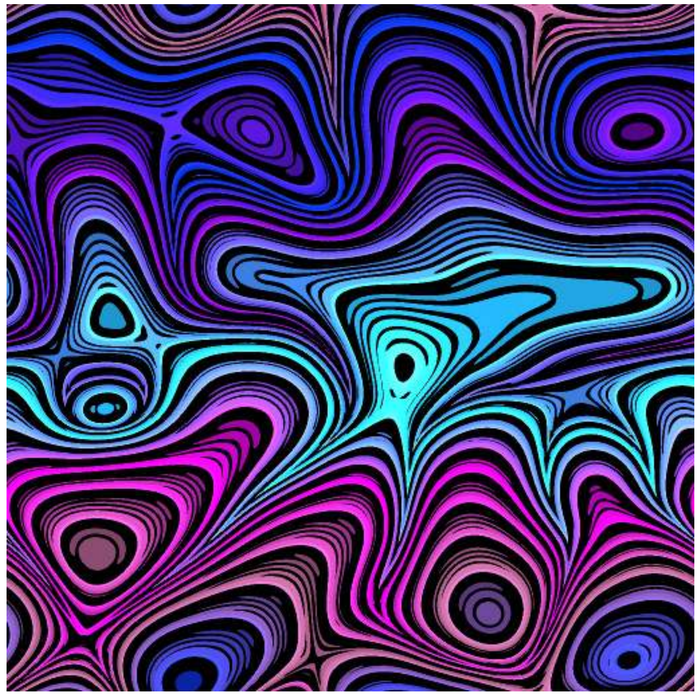


Vertex Coloring

Limited by mesh resolution!

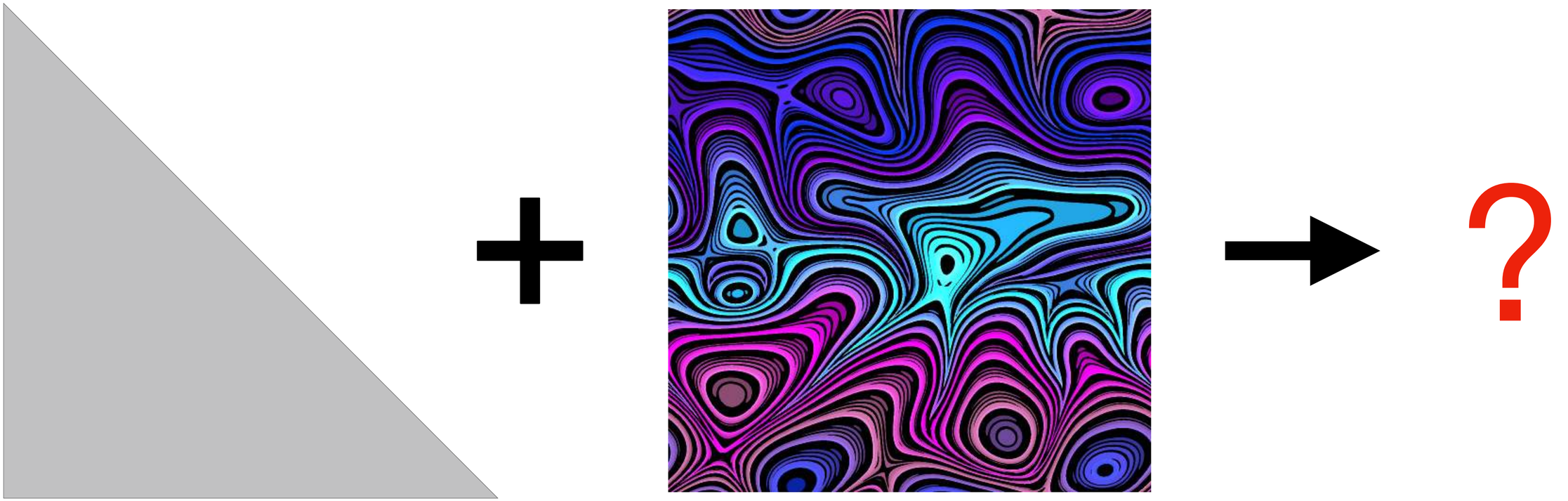


+



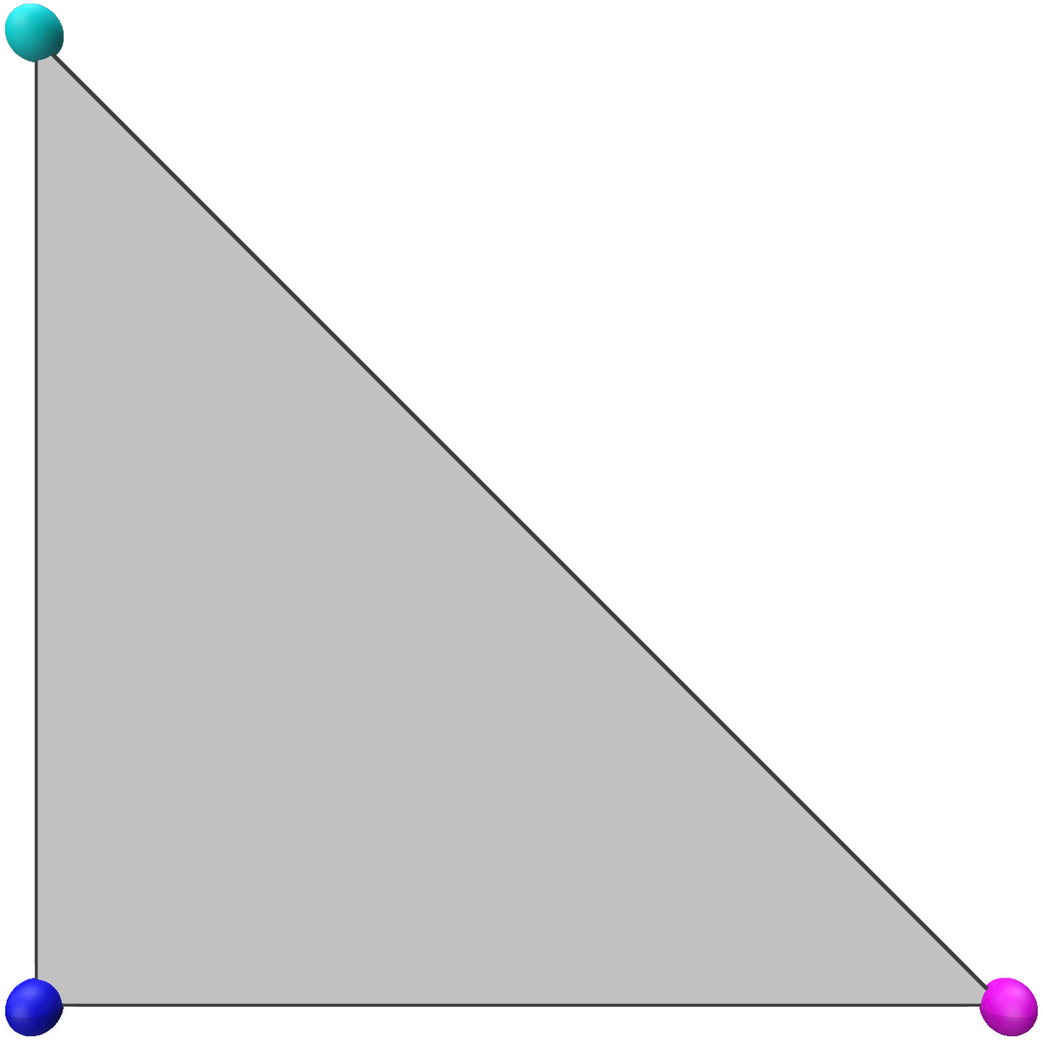
Vertex Coloring

Limited by mesh resolution!



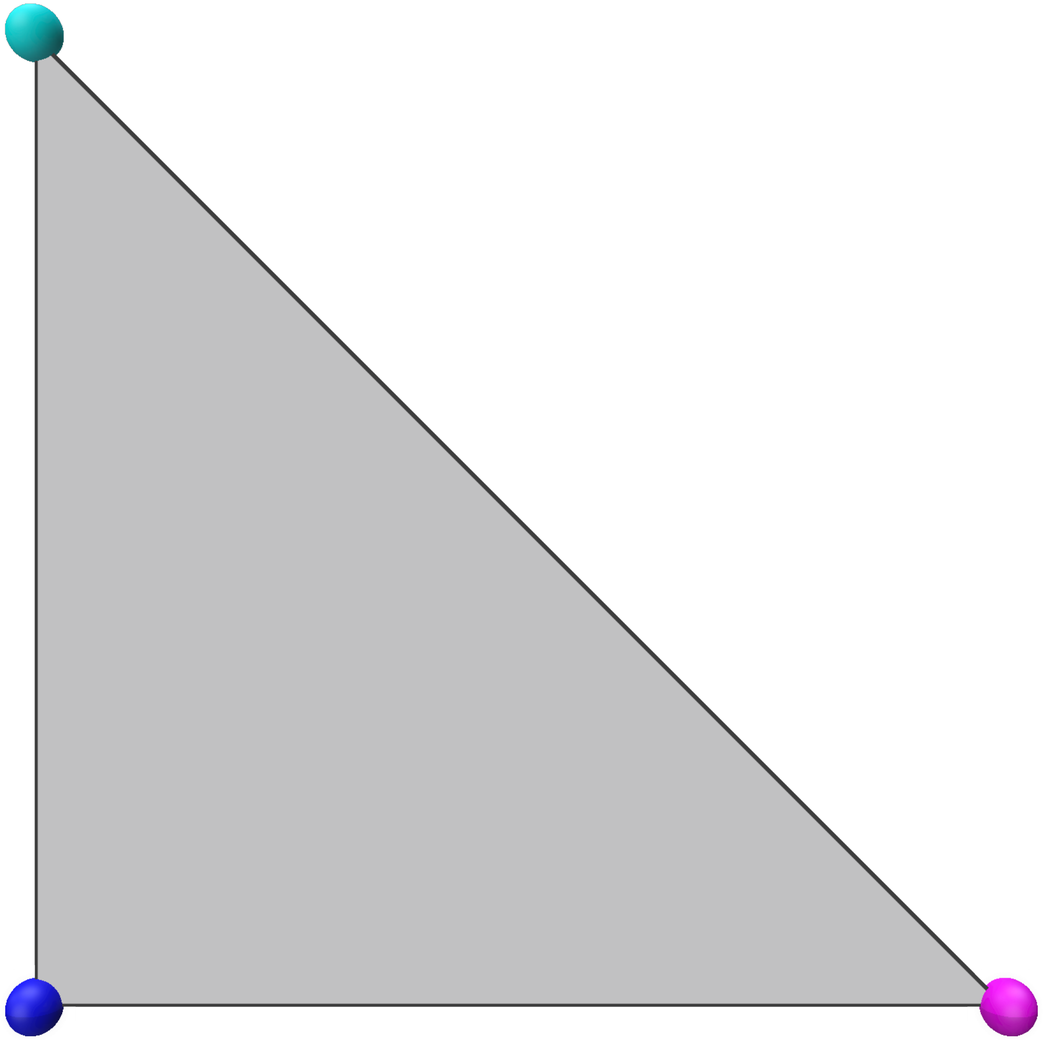
Vertex Coloring

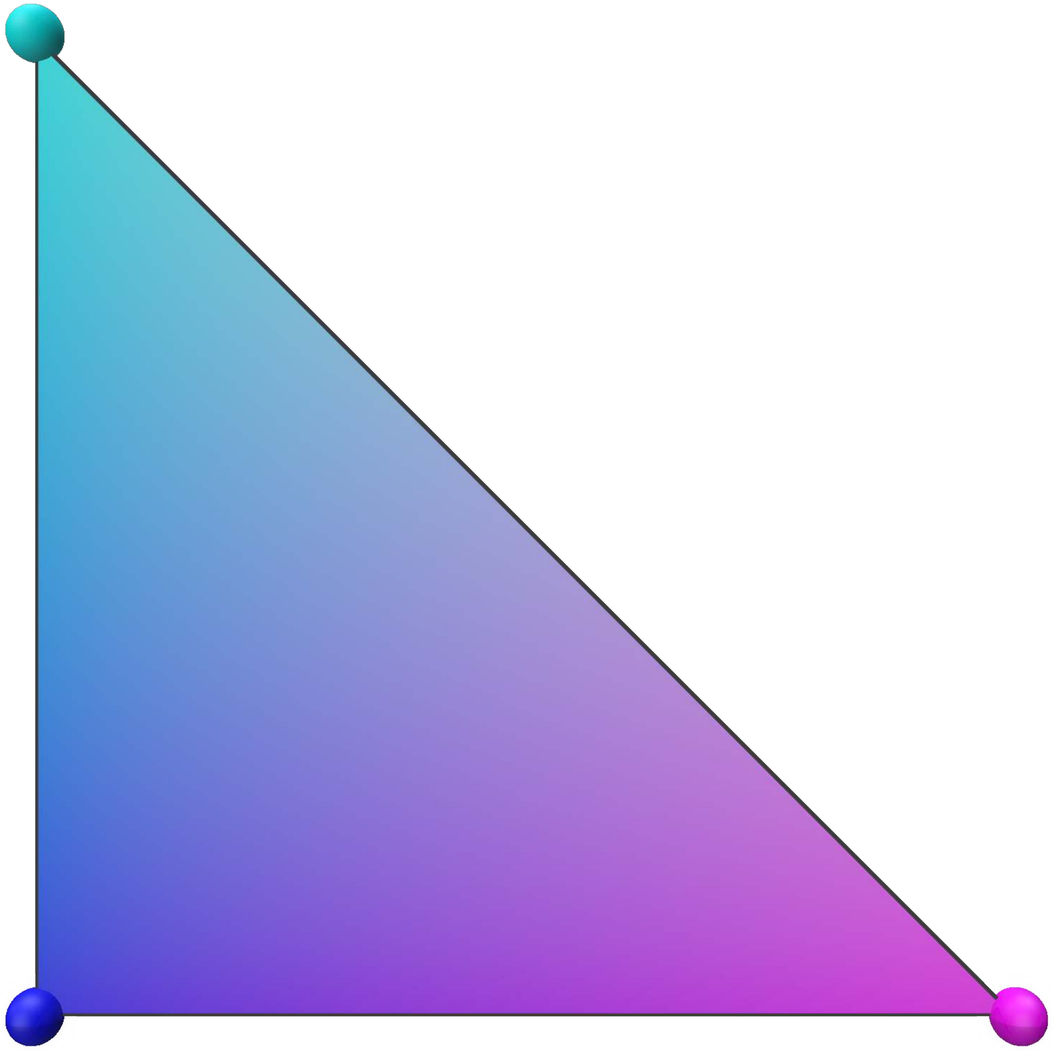
Limited by mesh resolution!



Vertex Coloring

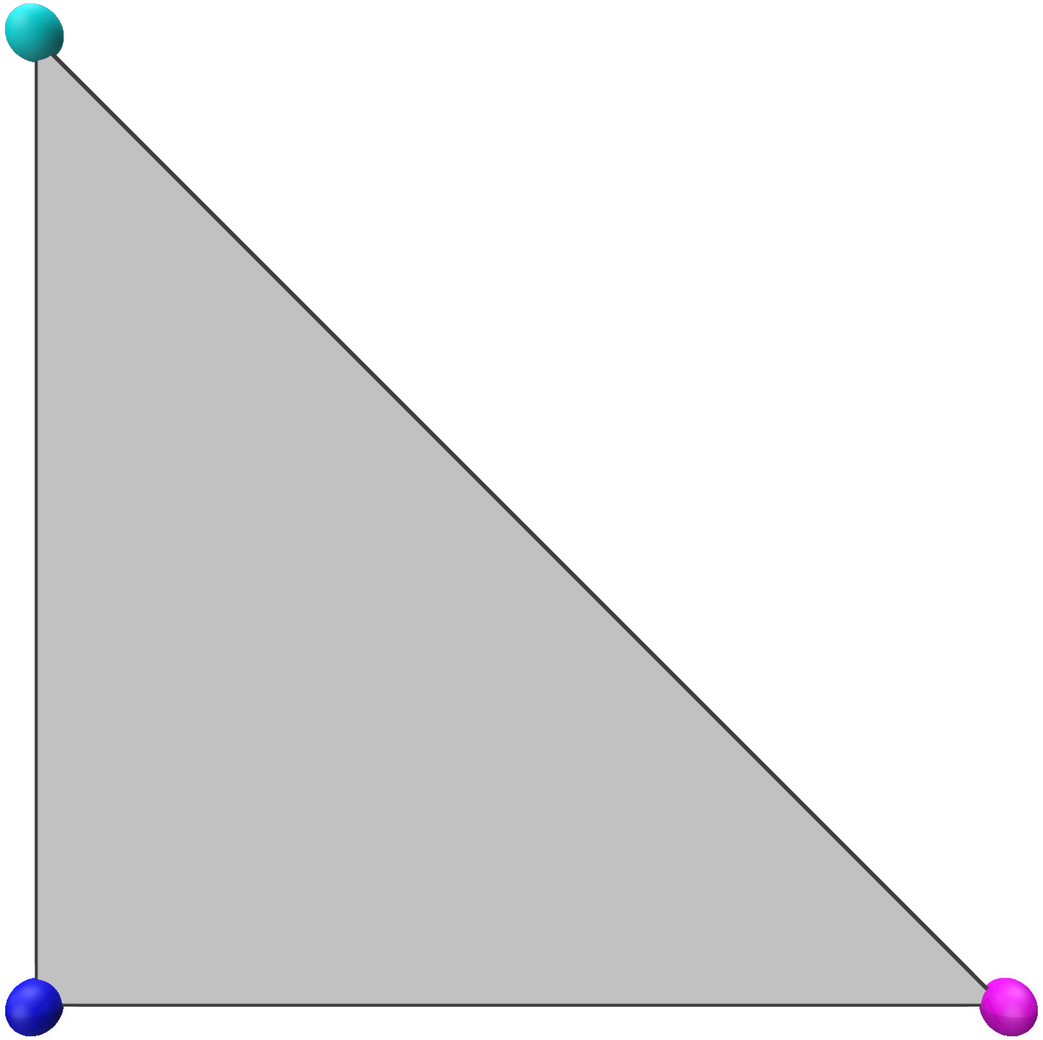
Limited by mesh resolution!

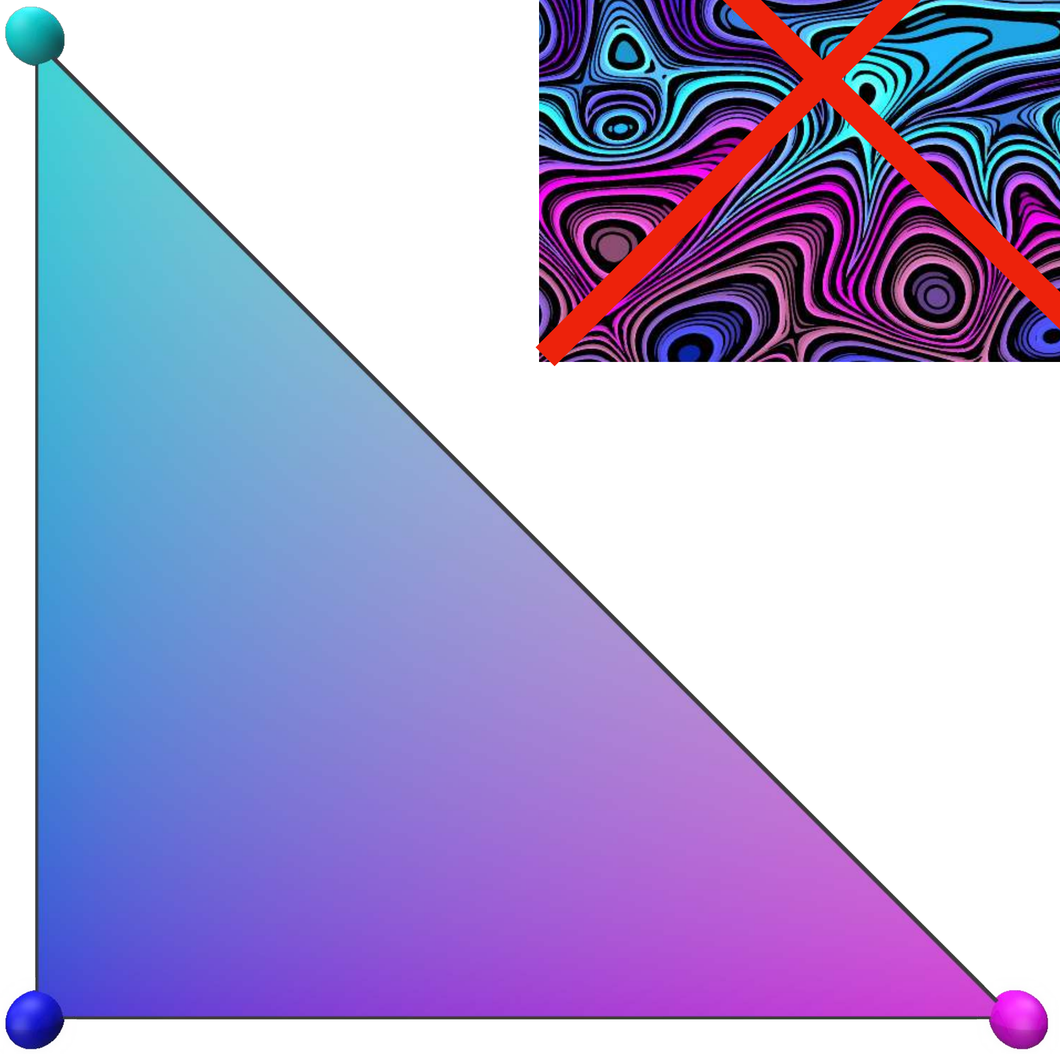




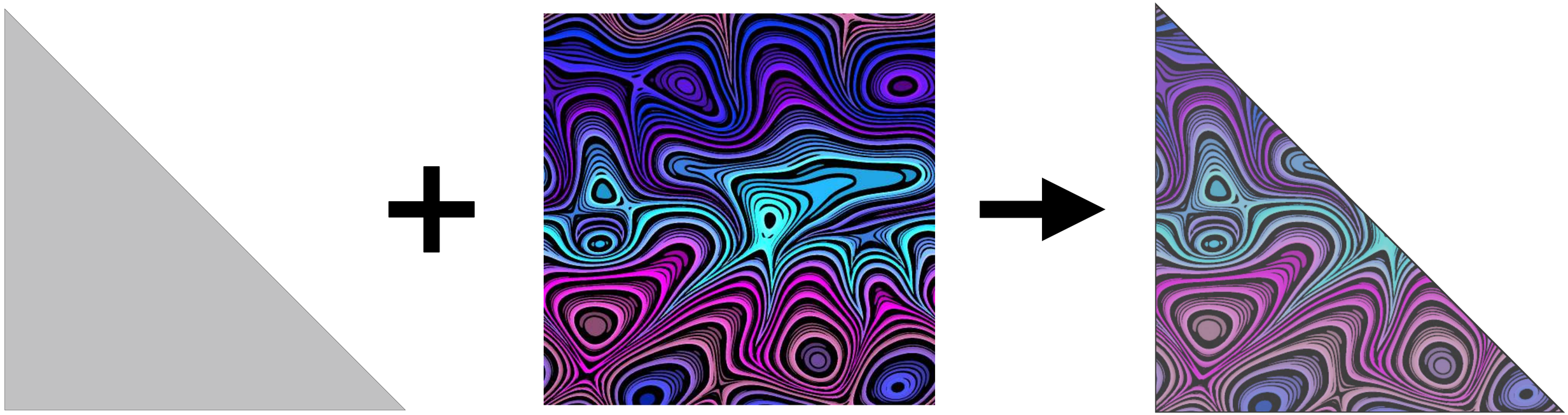
Vertex Coloring

Limited by mesh resolution!

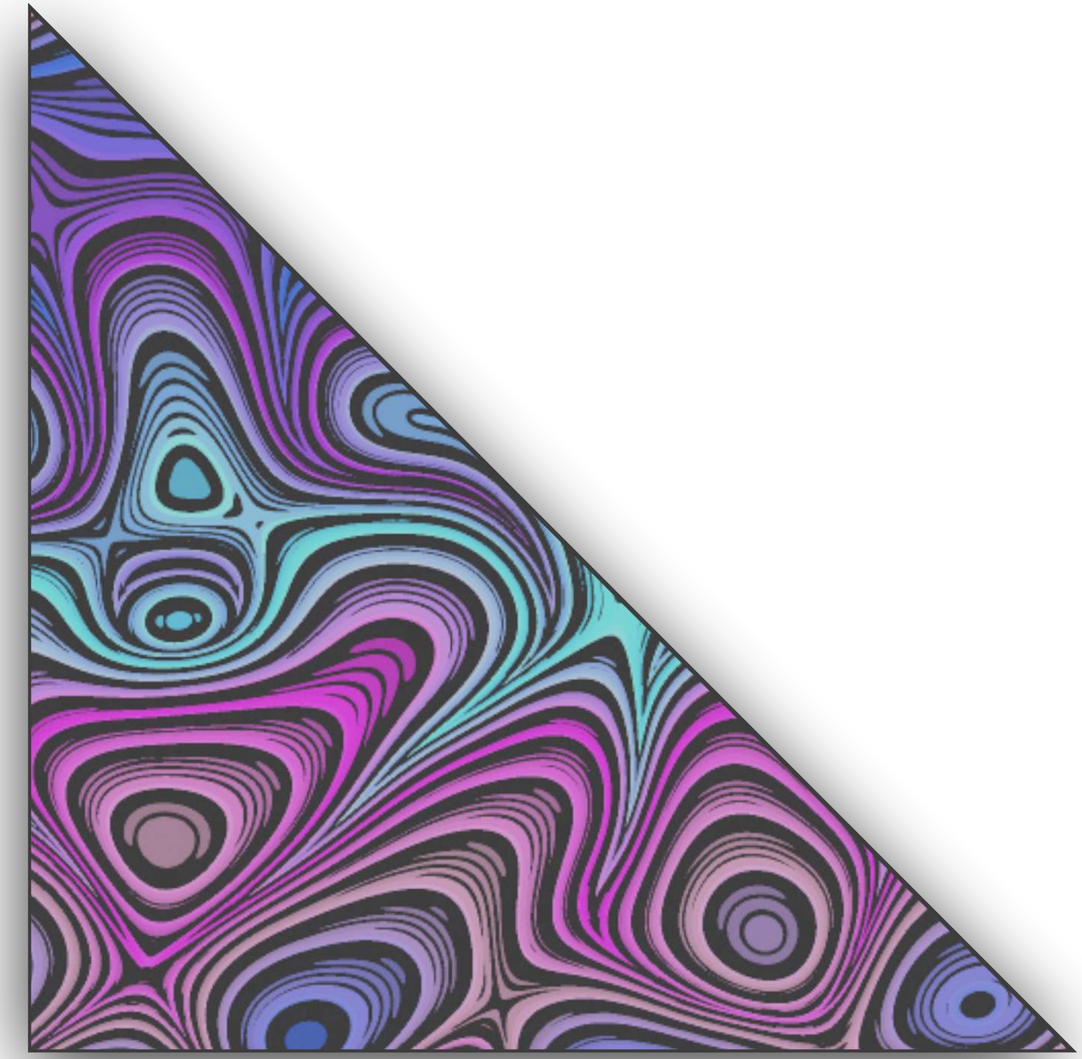




How can we get this?

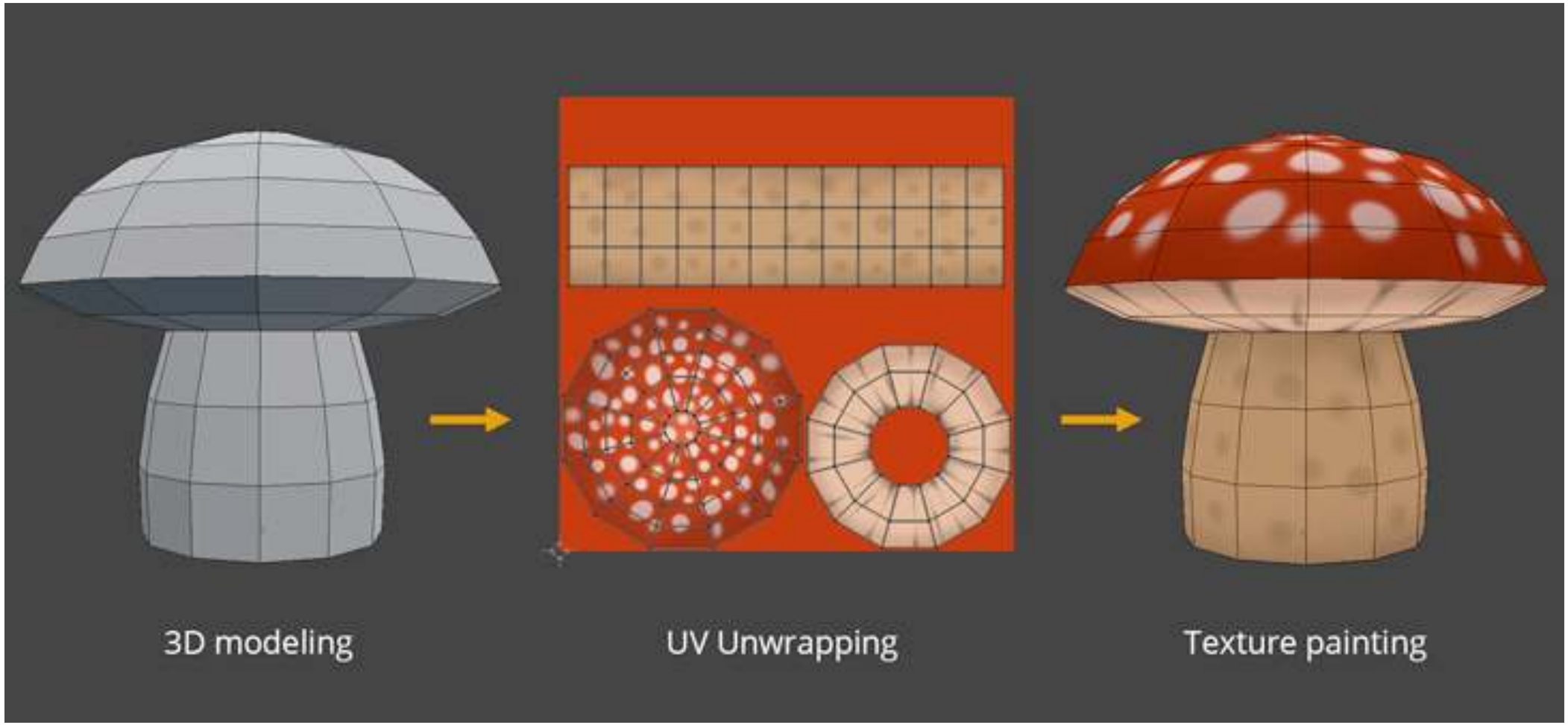


Solution: Texture Mapping



Texture Mapping

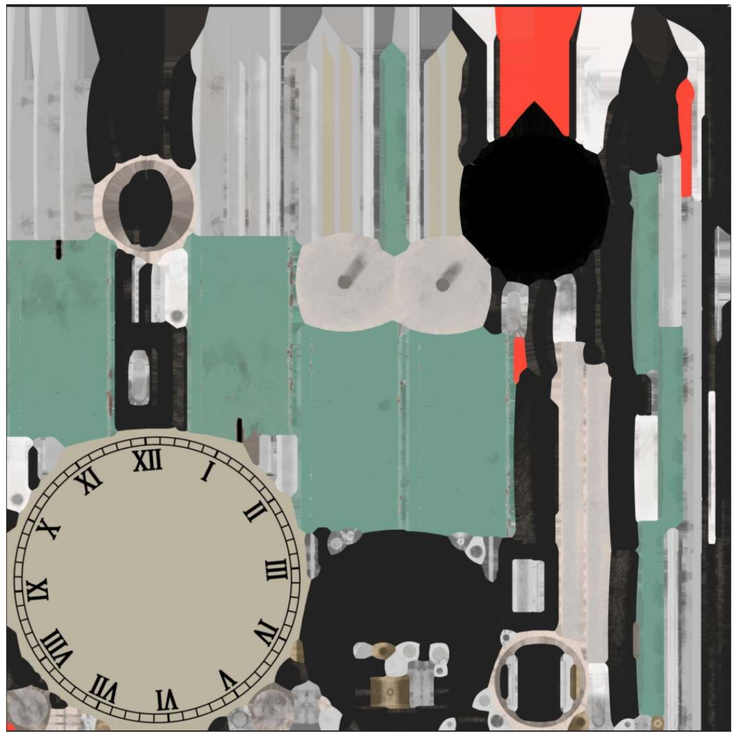
Color resolution is independent from mesh resolution



https://www.blendernation.com/2017/04/22/tutorial-modeling-uv-unwrapping-texturing-mushroom/

Texture Mapping

Color resolution is independent from mesh resolution



https://polyhaven.com/

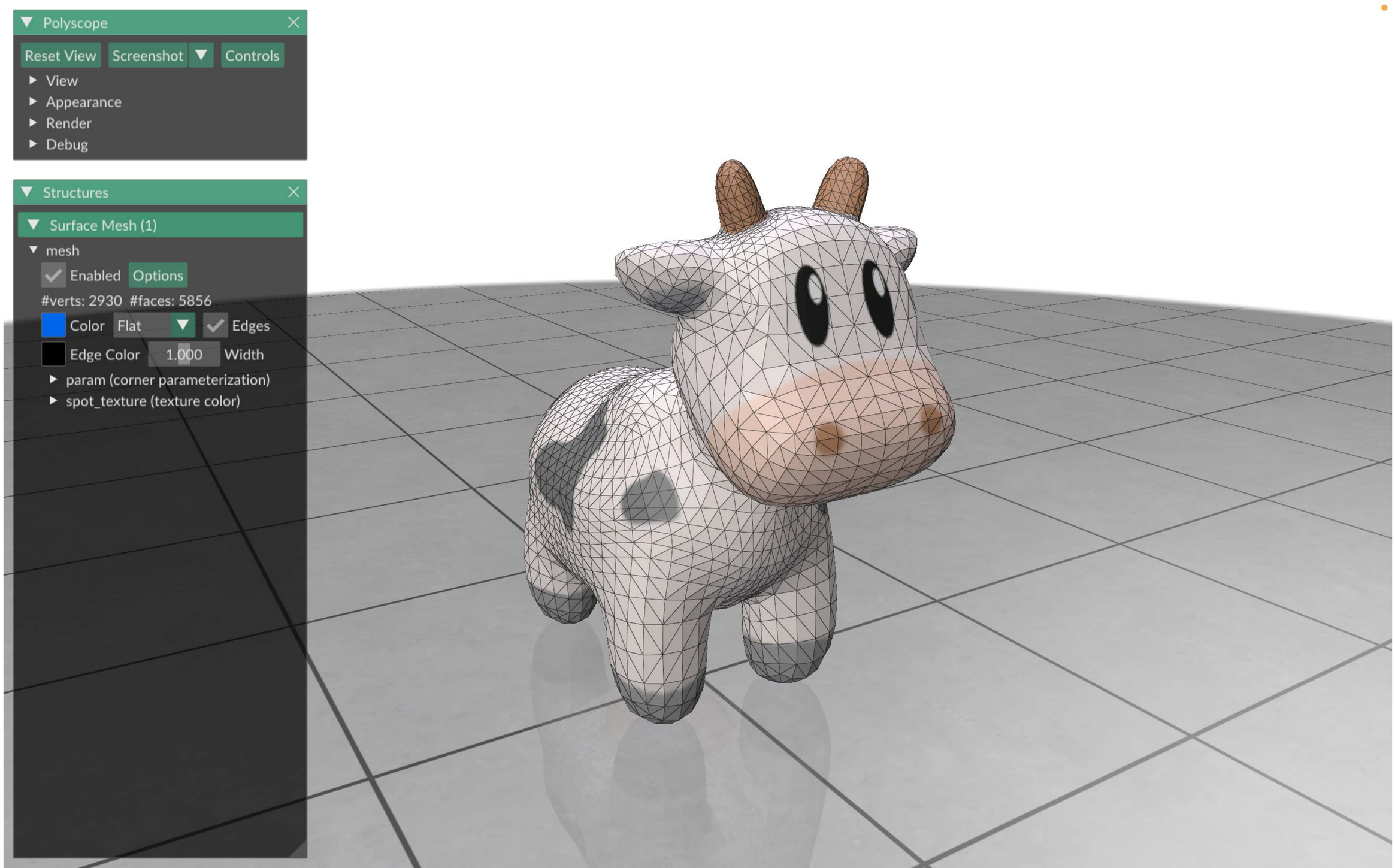
Texture Map Code Demo

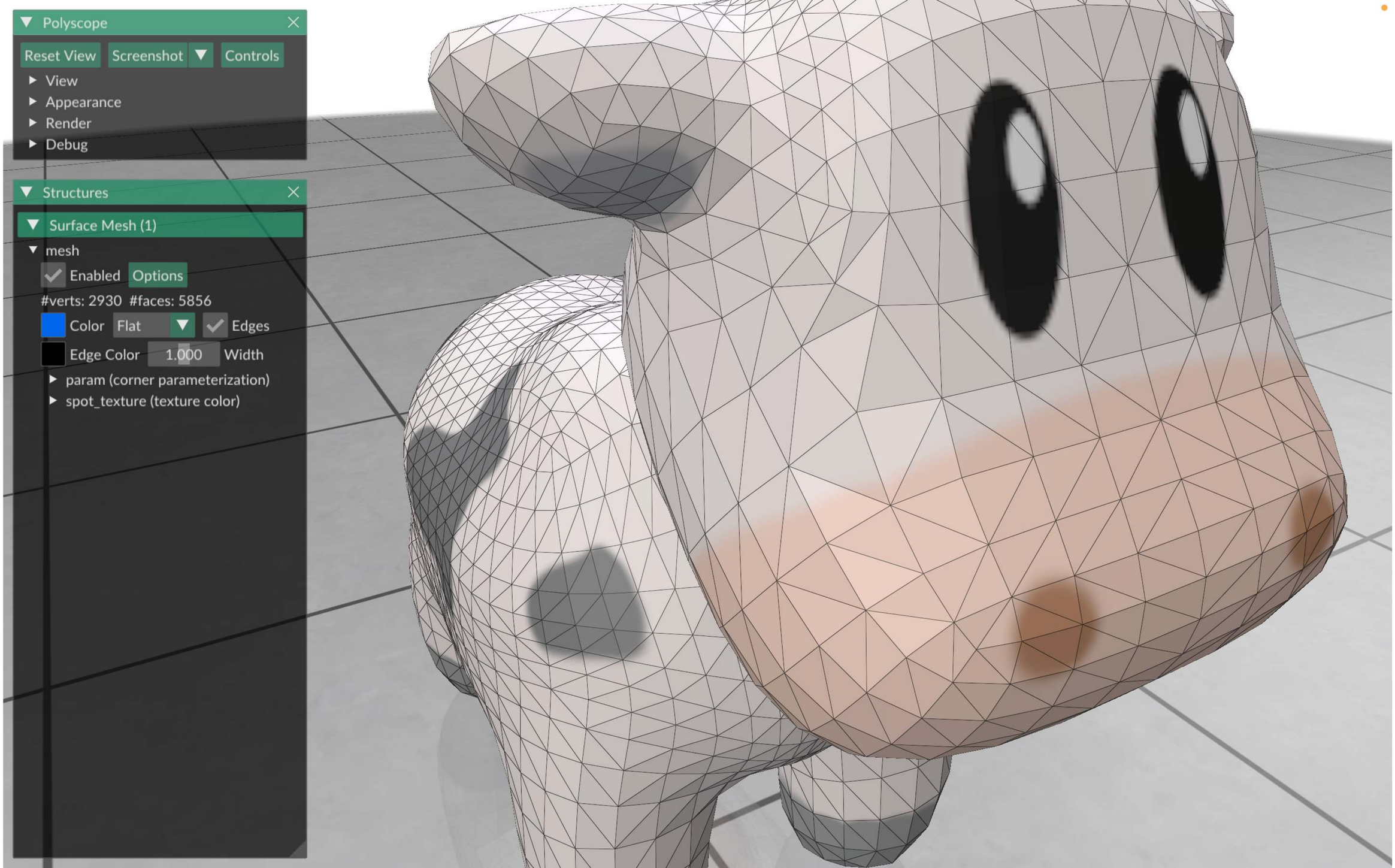
```python
import polyscope as ps
import numpy as np
from src.mesh import Mesh
from PIL import Image

ps.init()
mesh = Mesh("data/spot.obj")
ps_mesh = ps.register_surface_mesh(
    "mesh",
    mesh.vertices,
    mesh.faces,
    edge_width=1,
    material='clay',
)

# Add the parameterization
ps_mesh.add_parameterization_quantity("param", mesh.uv, defined_on='corners')

# Add the texture map
texture_map = Image.open("data/spot_texture.png")
# normalize the texture map values to be between 0 and 1
texture_map = np.asarray(texture_map) / 255
ps_mesh.add_color_quantity("spot_texture", texture_map,
                           defined_on='texture', param_name="param",
                           enabled=True)
ps.show()
```


Polyscope
Reset View
Screenshot
Controls
View
Appearance
Render
Debug
Structures
Surface Mesh (1)
mesh
Enabled
Options
#verts: 2930 #faces: 5856
Color
Flat
Edges
Edge Color
1.000
Width
param (corner parameterization)
spot_texture (texture color)


Polyscope
Reset View
Screenshot
Controls
View
Appearance
Render
Debug
Structures
Surface Mesh (1)
mesh
Enabled
Options
#verts: 2930 #faces: 5856
Color
Flat
Edges
Edge Color
1.000
Width
param (corner parameterization)
spot_texture (texture color)

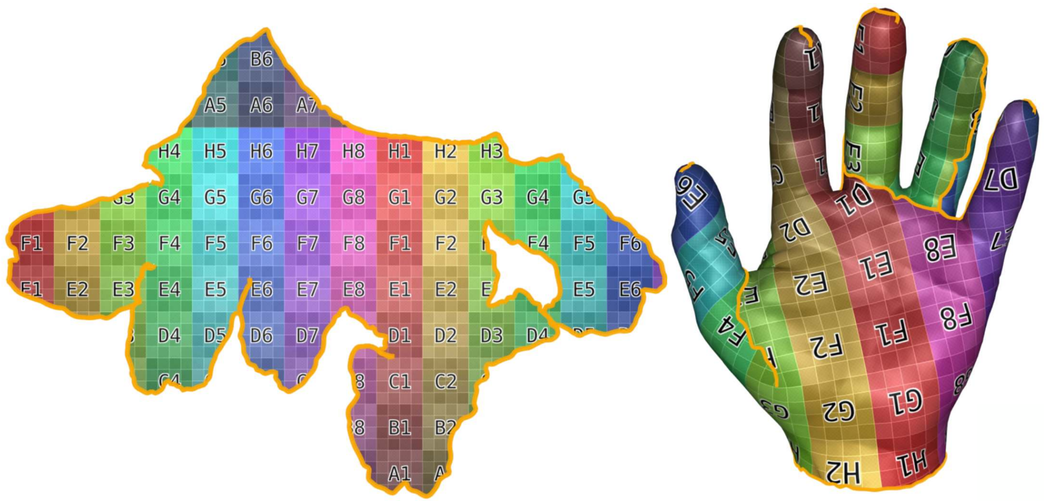
What makes a good parameterization for texture maps?

Recall, we typically want our parameterization to minimize both:

What makes a good parameterization for texture maps?

Recall, we typically want our parameterization to minimize both:

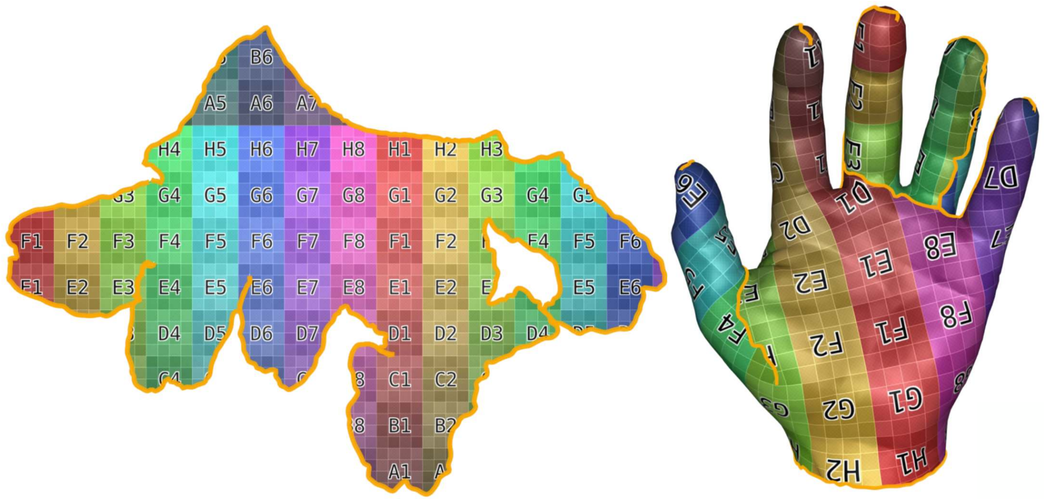
Cuts!



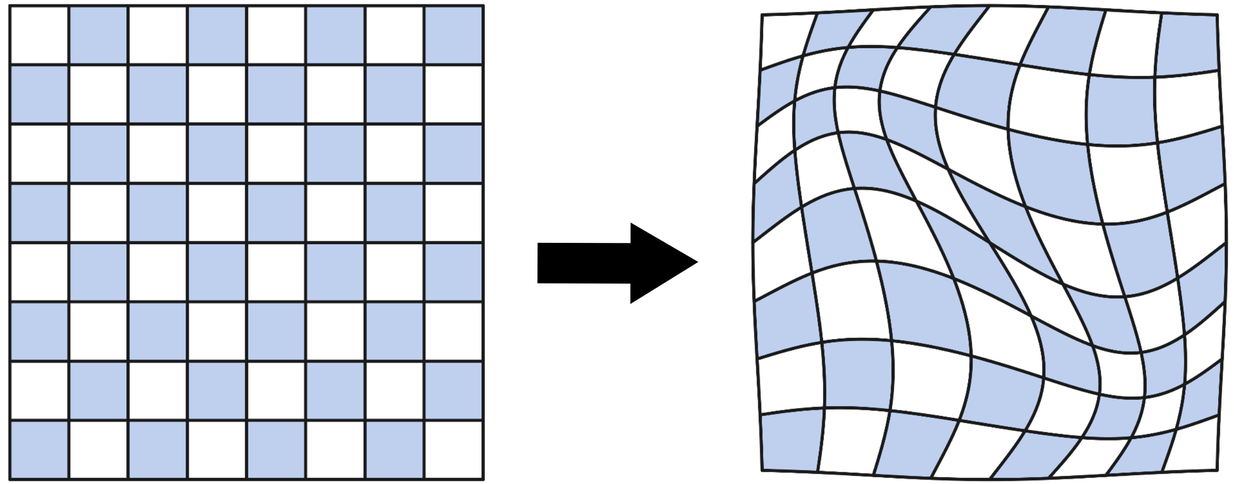
What makes a good parameterization for texture maps?

Recall, we typically want our parameterization to minimize both:

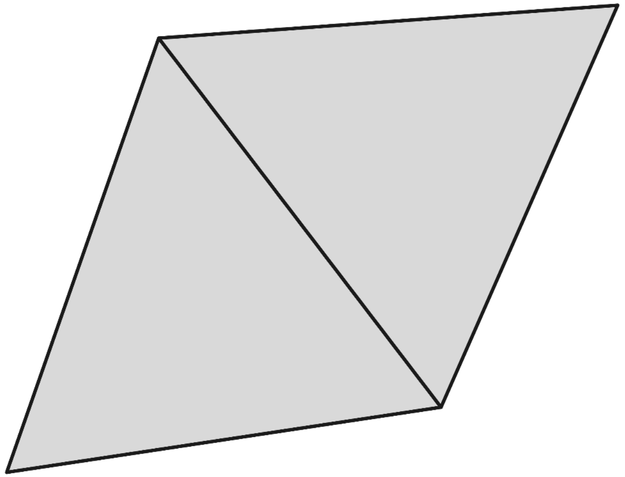
Cuts!



Distortion!

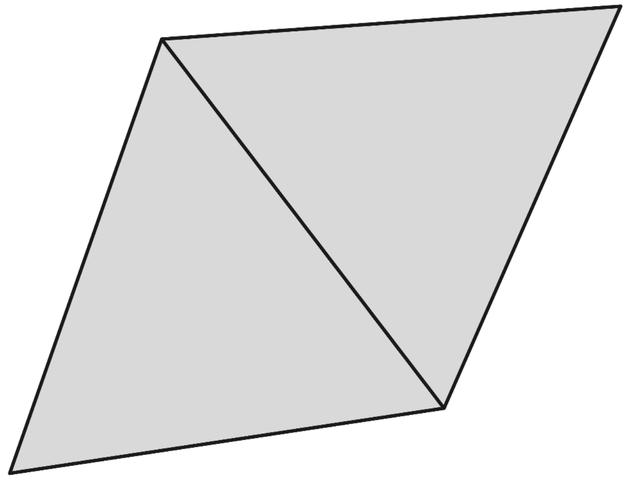


Why do we care about distortion for texture maps?



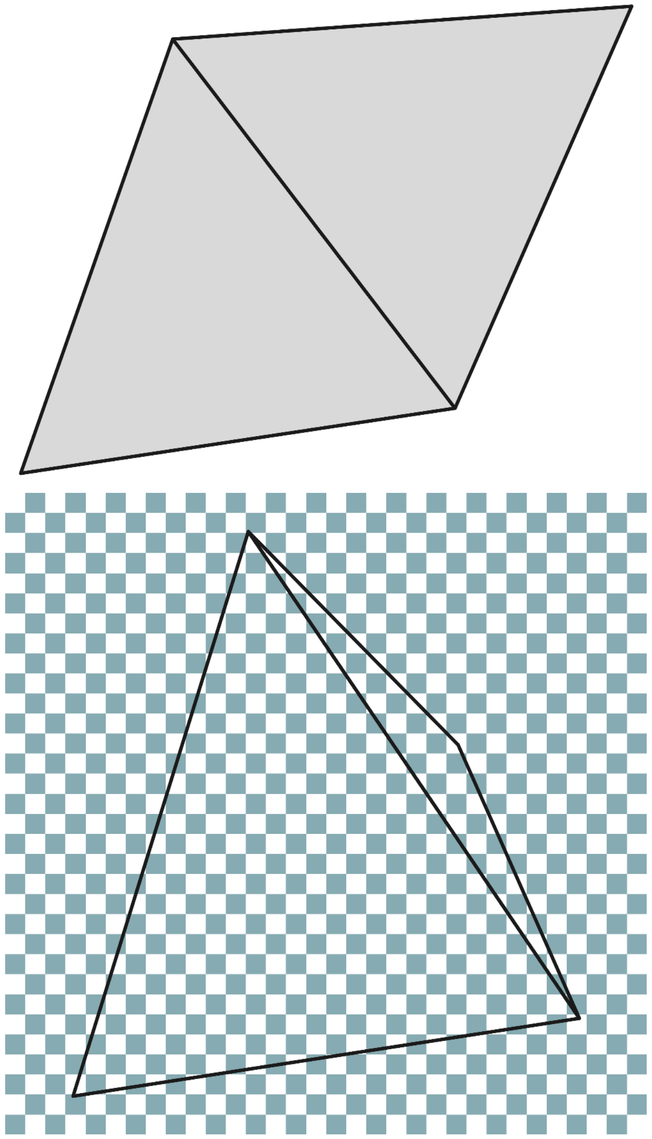
Why do we care about distortion for texture maps?

- When applying our texture maps, we want our texels to uniformly sample the surface



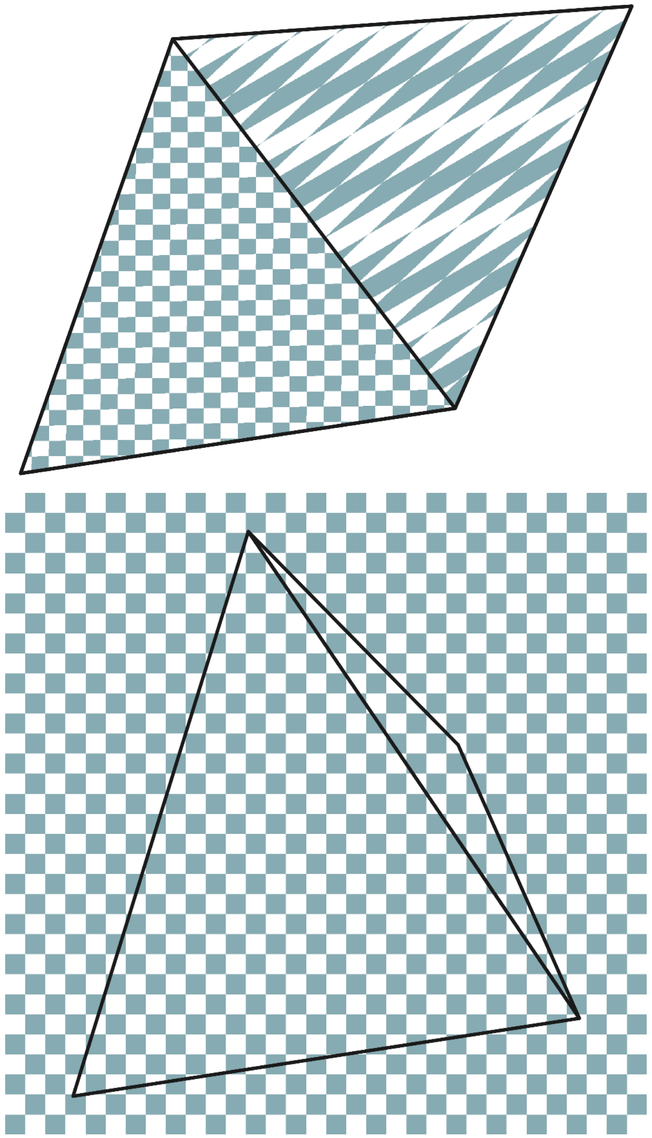
Why do we care about distortion for texture maps?

- When applying our texture maps, we want our texels to uniformly sample the surface
- Parameterization w/ distortion → uneven texel sampling



Why do we care about distortion for texture maps?

- When applying our texture maps, we want our texels to uniformly sample the surface
- Parameterization w/ distortion → uneven texel sampling
- This can result in inconsistent texture resolution on the mesh



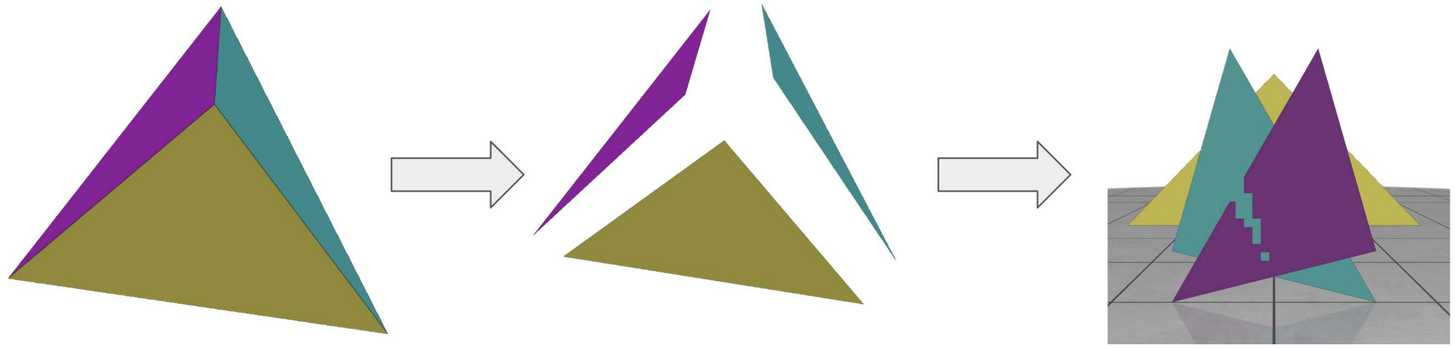
Triangle Soup Parameterization

How can we do this?

Triangle Soup Parameterization

How can we do this?

- Trivial parameterization (as shown w/ Richard)

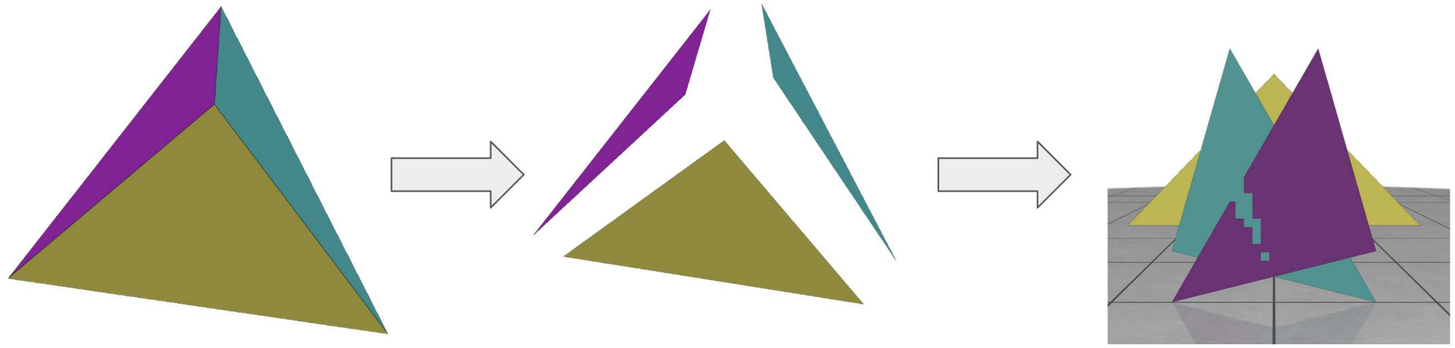


Triangle Soup Parameterization

How can we do this?

- Trivial parameterization (as shown w/ Richard)

triangles overlap!

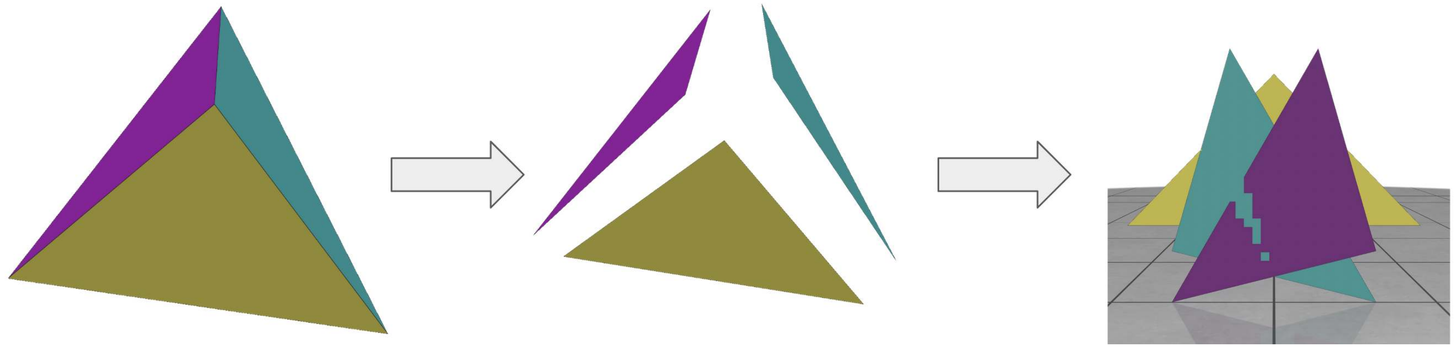


Triangle Soup Parameterization

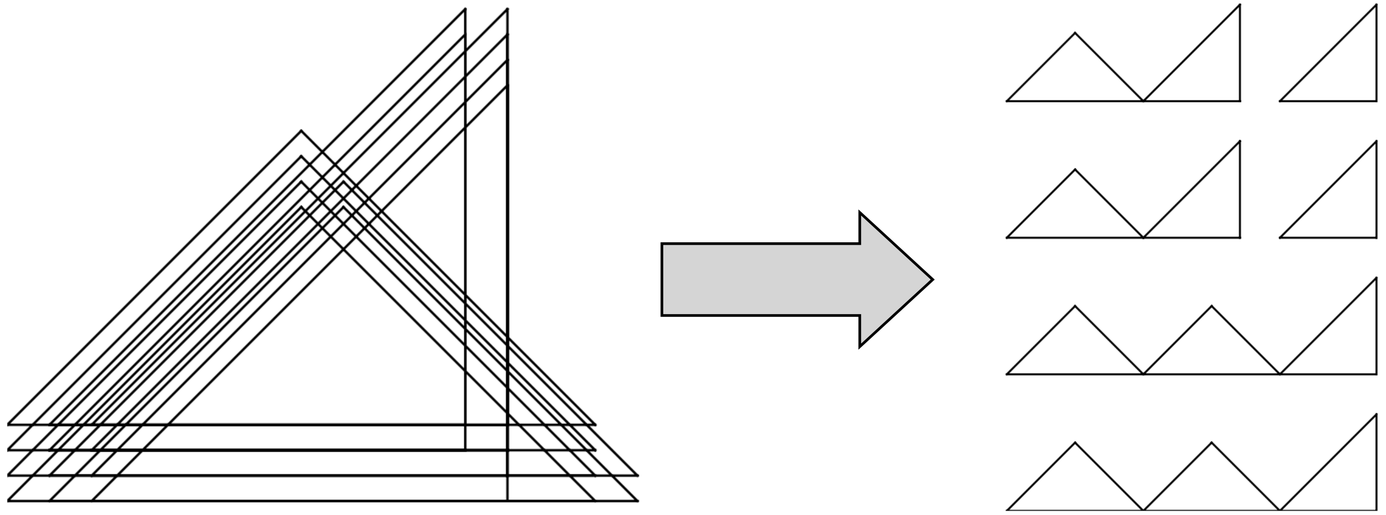
How can we do this?

triangles overlap!

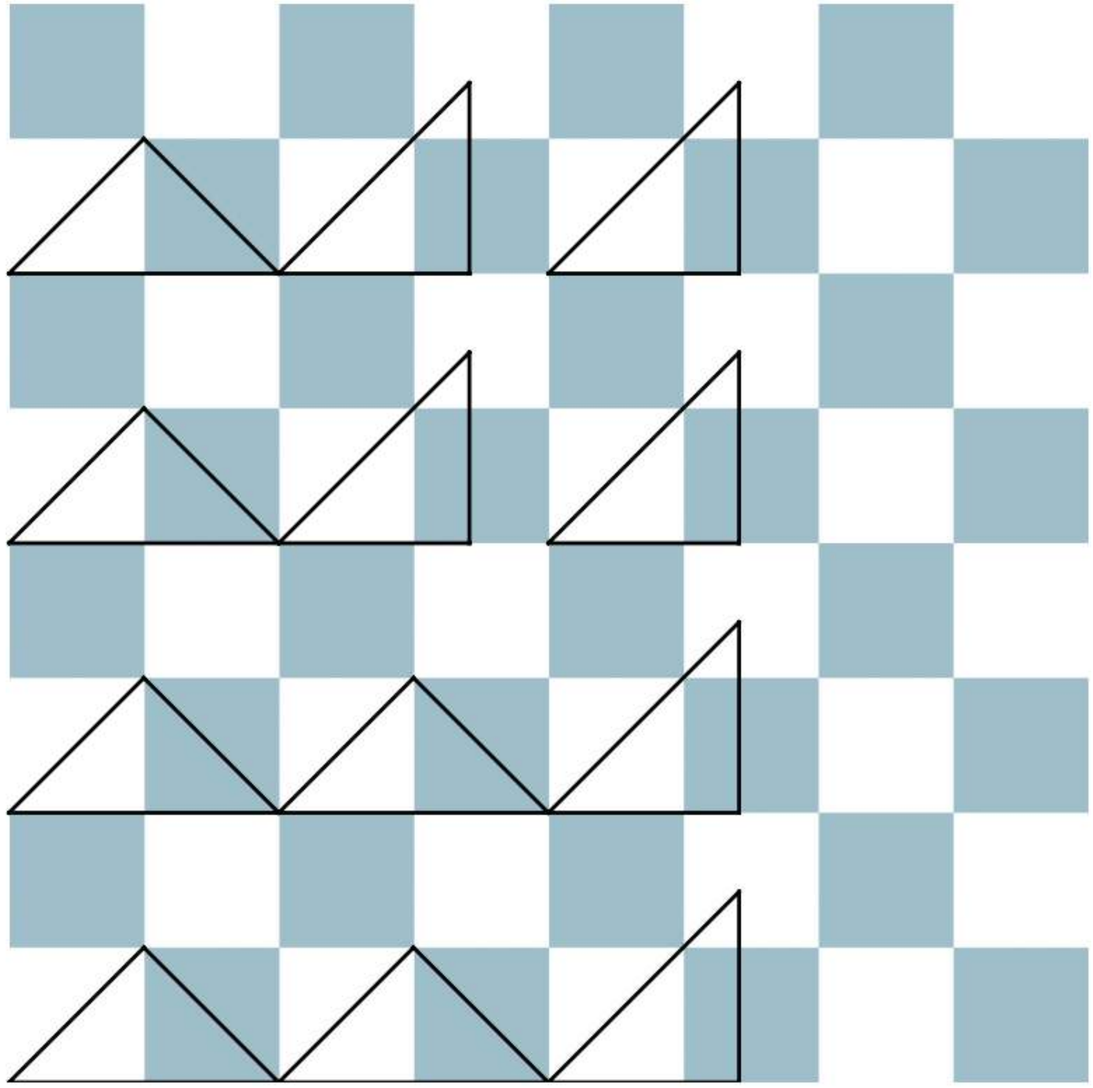
- Trivial parameterization (as shown w/ Richard)



- Then translate and scale all triangles to be packed into the unit square

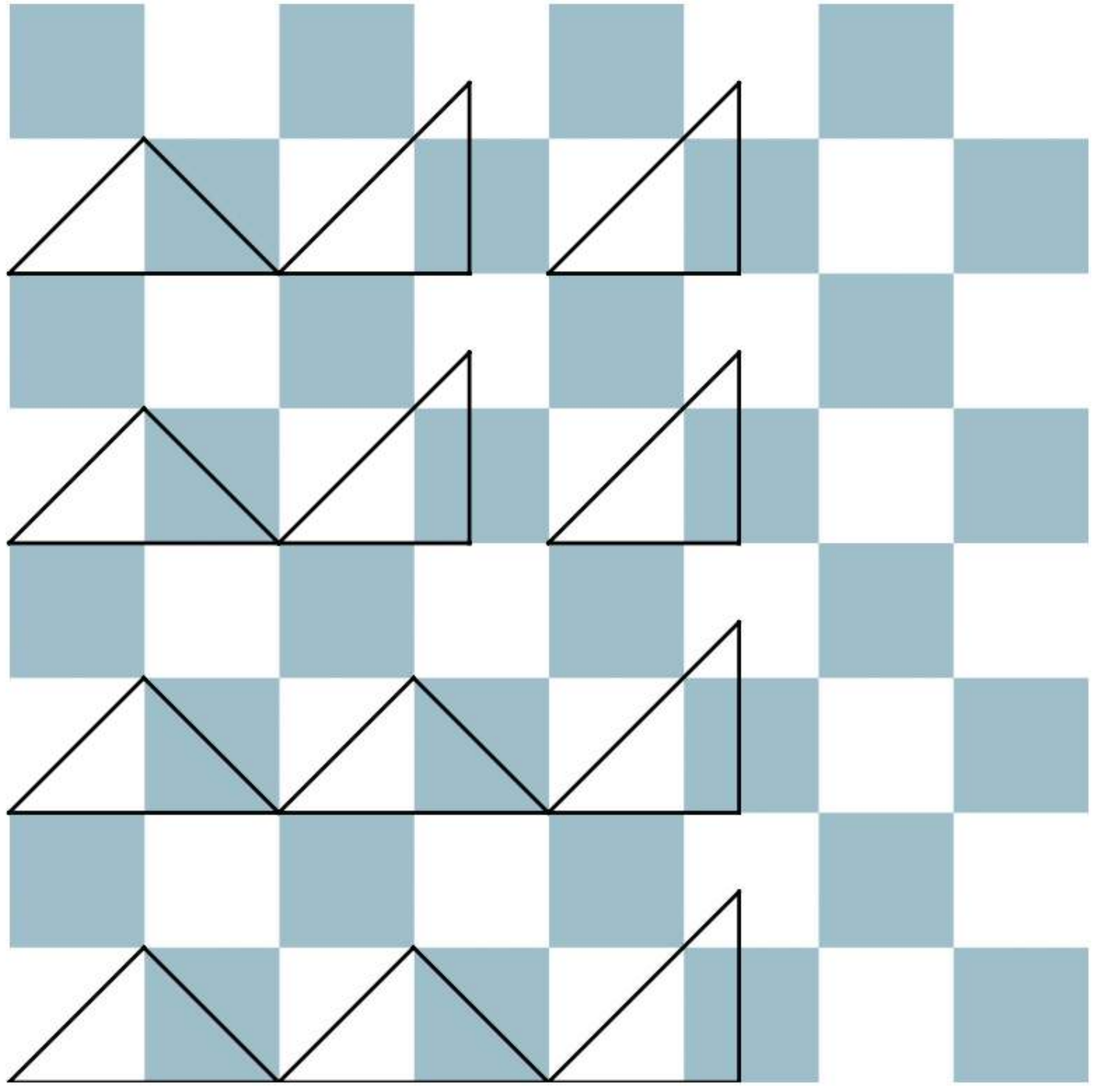


Triangle Soup Parameterization

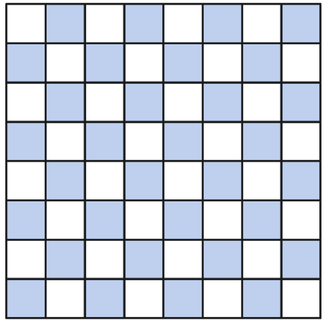


Triangle Soup Parameterization

Is there distortion here?

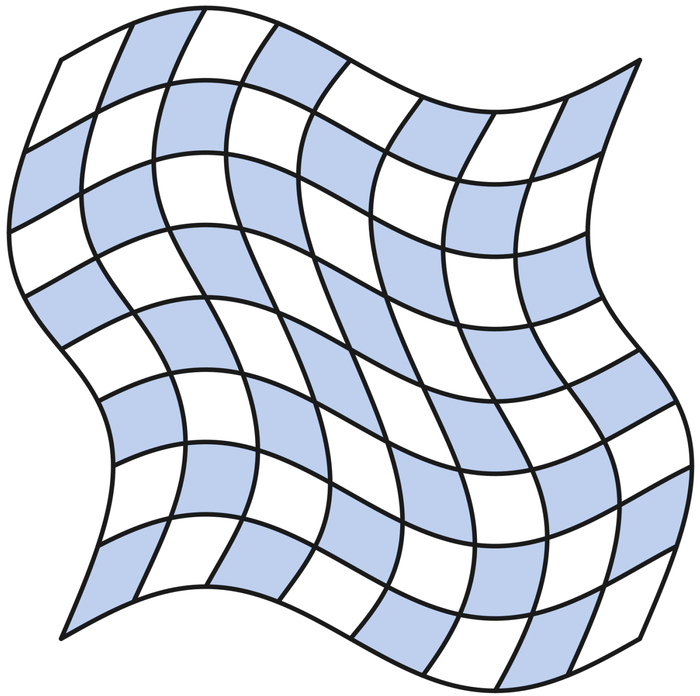


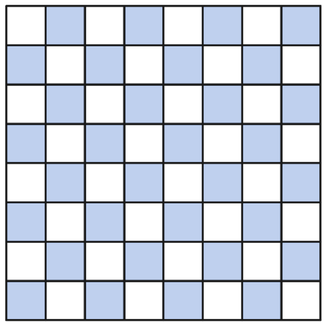
Recall: 2 types of distortion



Recall: 2 types of distortion

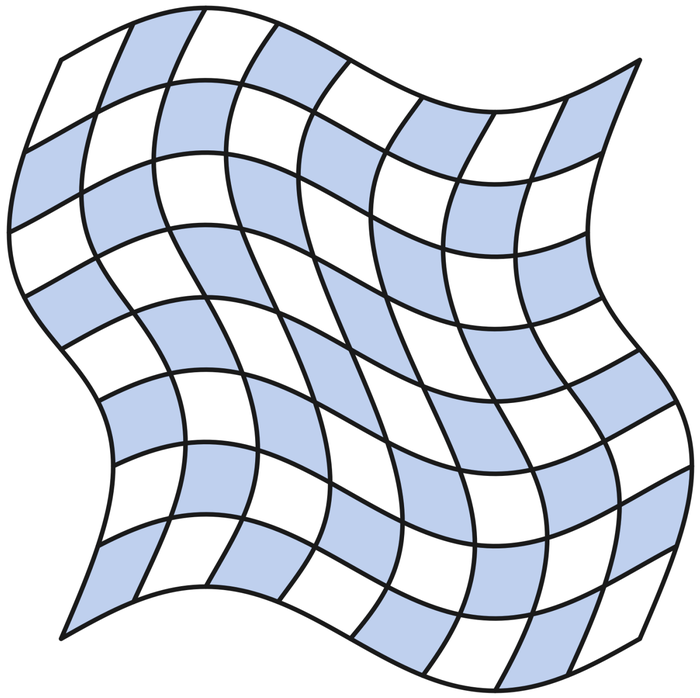
angle distortion (not conformal)

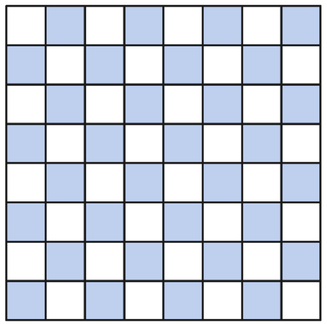




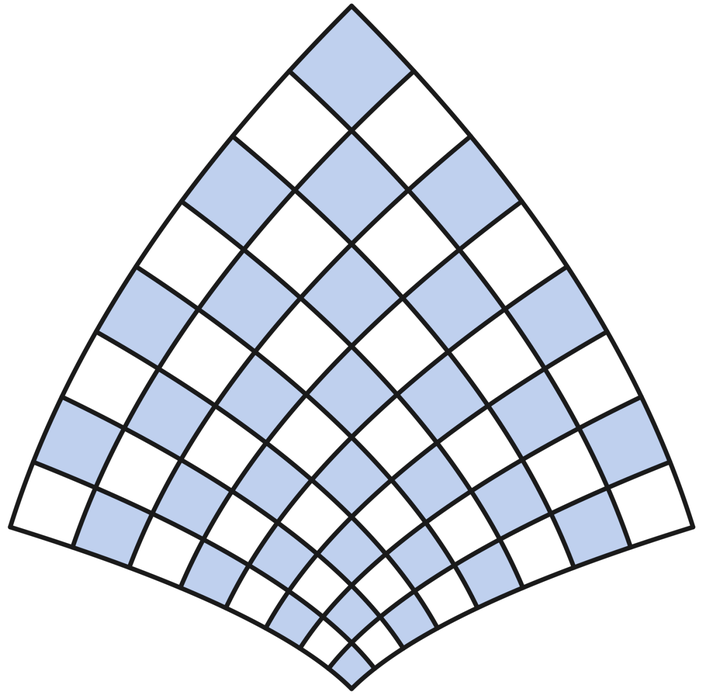
Recall: 2 types of distortion

angle distortion (not conformal)





area distortion (not authalic)



Recall: singular values & distortion

The singular values of the Jacobian tell us everything:

$$J_f = U\Sigma V^T = U \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{pmatrix} V^T$$

Recall: singular values & distortion

The singular values of the Jacobian tell us everything:

$$J_f = U\Sigma V^T = U \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{pmatrix} V^T$$

f is *conformal* or *angle-preserving* $\iff \sigma_1 = \sigma_2$

Recall: singular values & distortion

The singular values of the Jacobian tell us everything:

$$J_f = U\Sigma V^T = U \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{pmatrix} V^T$$

f is *conformal* or *angle-preserving* $\iff \sigma_1 = \sigma_2$

f is *authalic* or *area-preserving* $\iff \sigma_1\sigma_2 = 1$

Recall: singular values & distortion

The singular values of the Jacobian tell us everything:

$$J_f = U\Sigma V^T = U \begin{pmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \\ 0 & 0 \end{pmatrix} V^T$$

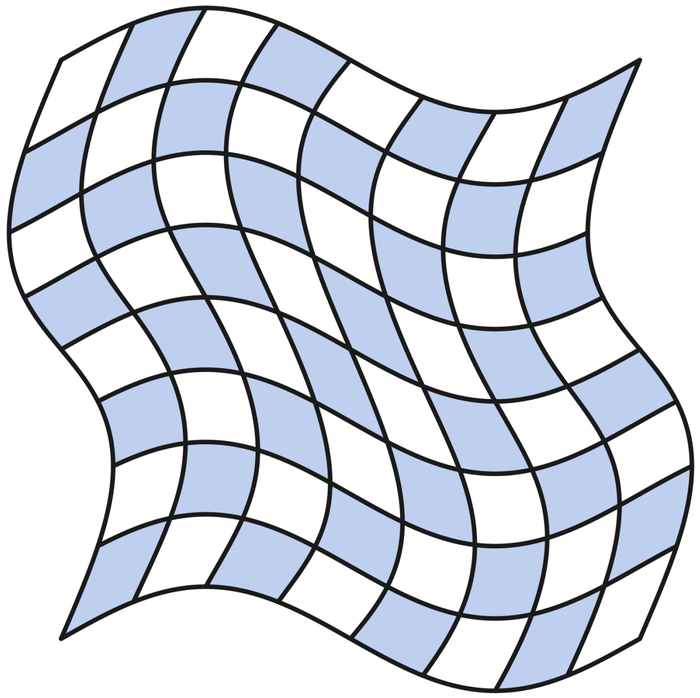
f is *conformal* or *angle-preserving* $\iff \sigma_1 = \sigma_2$

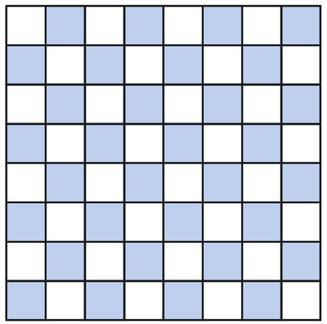
f is *authalic* or *area-preserving* $\iff \sigma_1\sigma_2 = 1$

f is *isometric* or *length-preserving* $\iff \sigma_1 = \sigma_2 = 1$

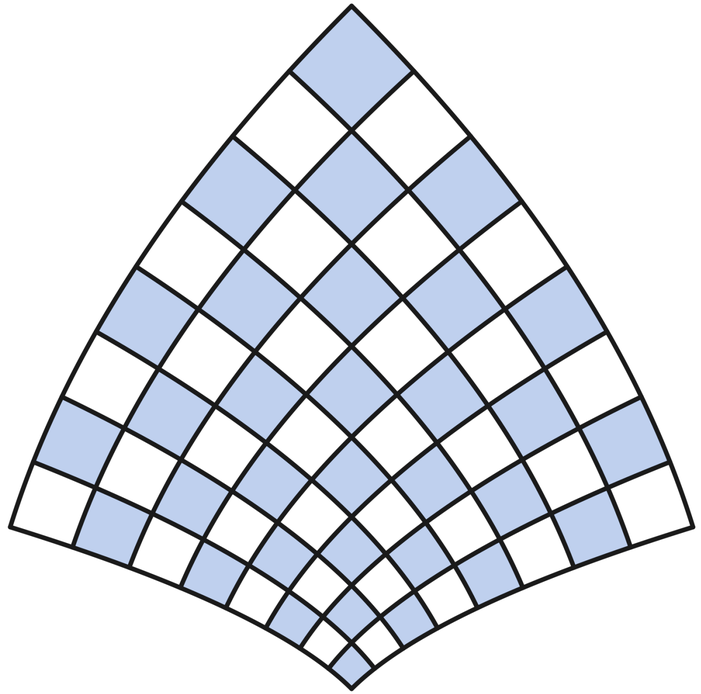
Recall: 2 types of distortion

angle distortion (not conformal)



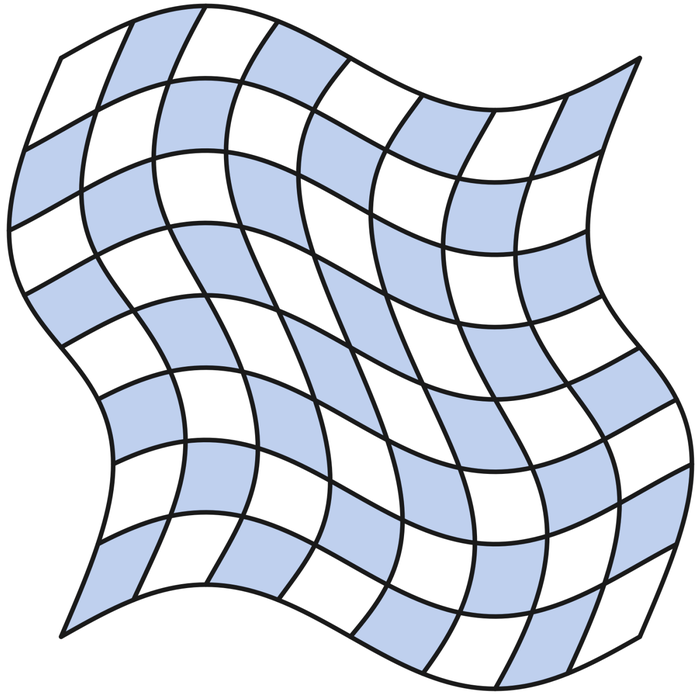


area distortion (not authalic)

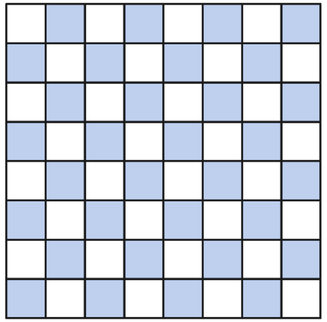


Recall: 2 types of distortion

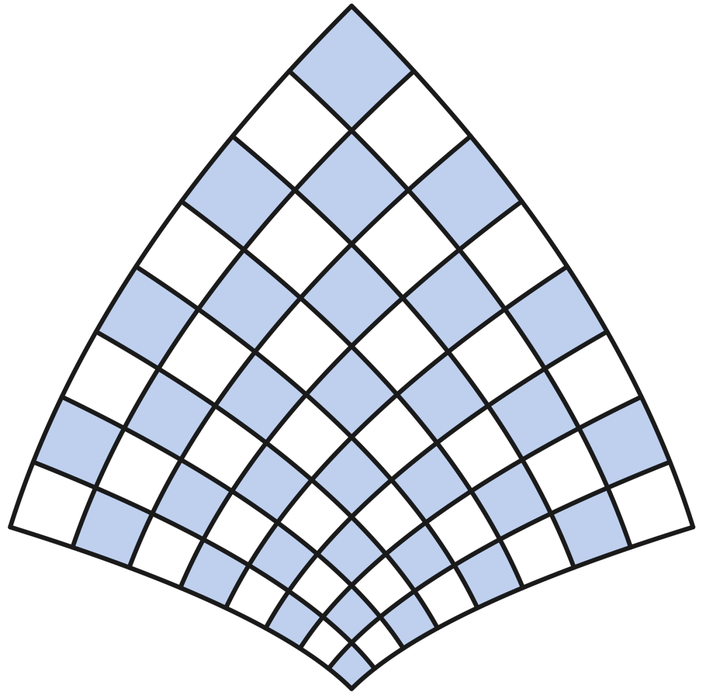
angle distortion (not conformal)



$\sigma_1 \neq \sigma_2$

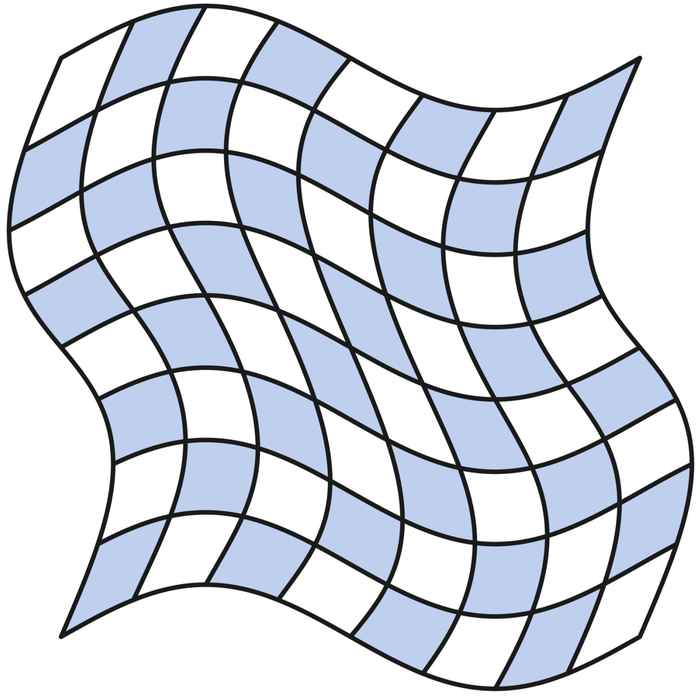


area distortion (not authalic)

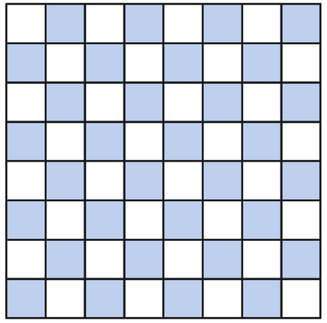


Recall: 2 types of distortion

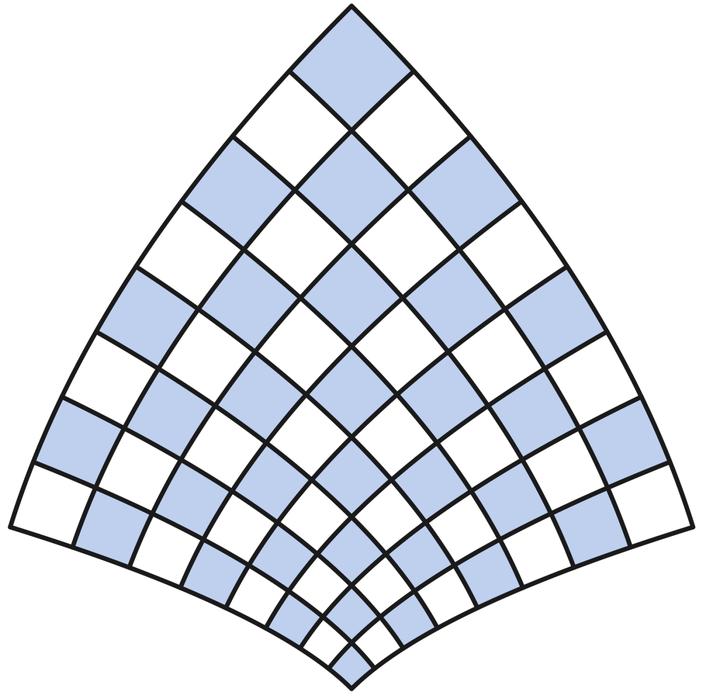
angle distortion (not conformal)



$\sigma_1 \neq \sigma_2$



area distortion (not authalic)



$\sigma_1 \sigma_2 \neq 1$

Triangle Soup Parameterization

Triangle Soup Parameterization

Angle distortion?

Triangle Soup Parameterization

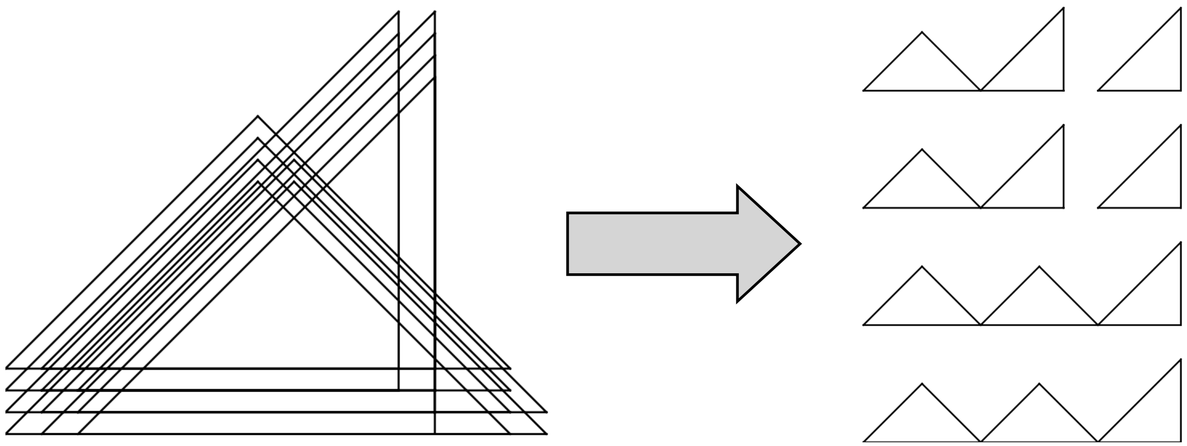
Angle distortion?

- All triangles mapped to plane w/ trivial parameterization (isometric!)

Triangle Soup Parameterization

Angle distortion?

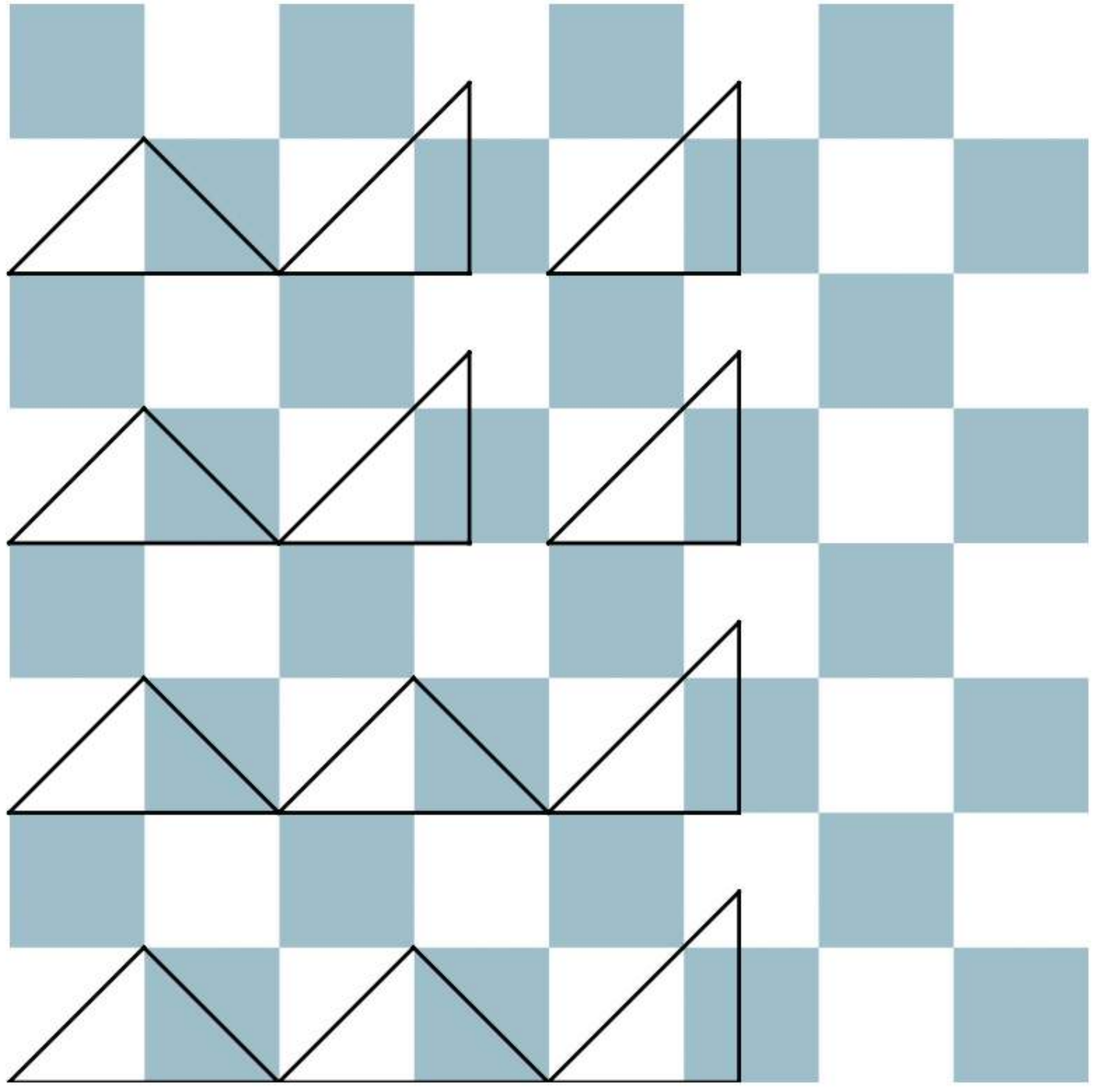
- All triangles mapped to plane w/ trivial parameterization (isometric!)
- Translation and uniform scaling does not affect the angles



Triangle Soup Parameterization

Angle distortion?

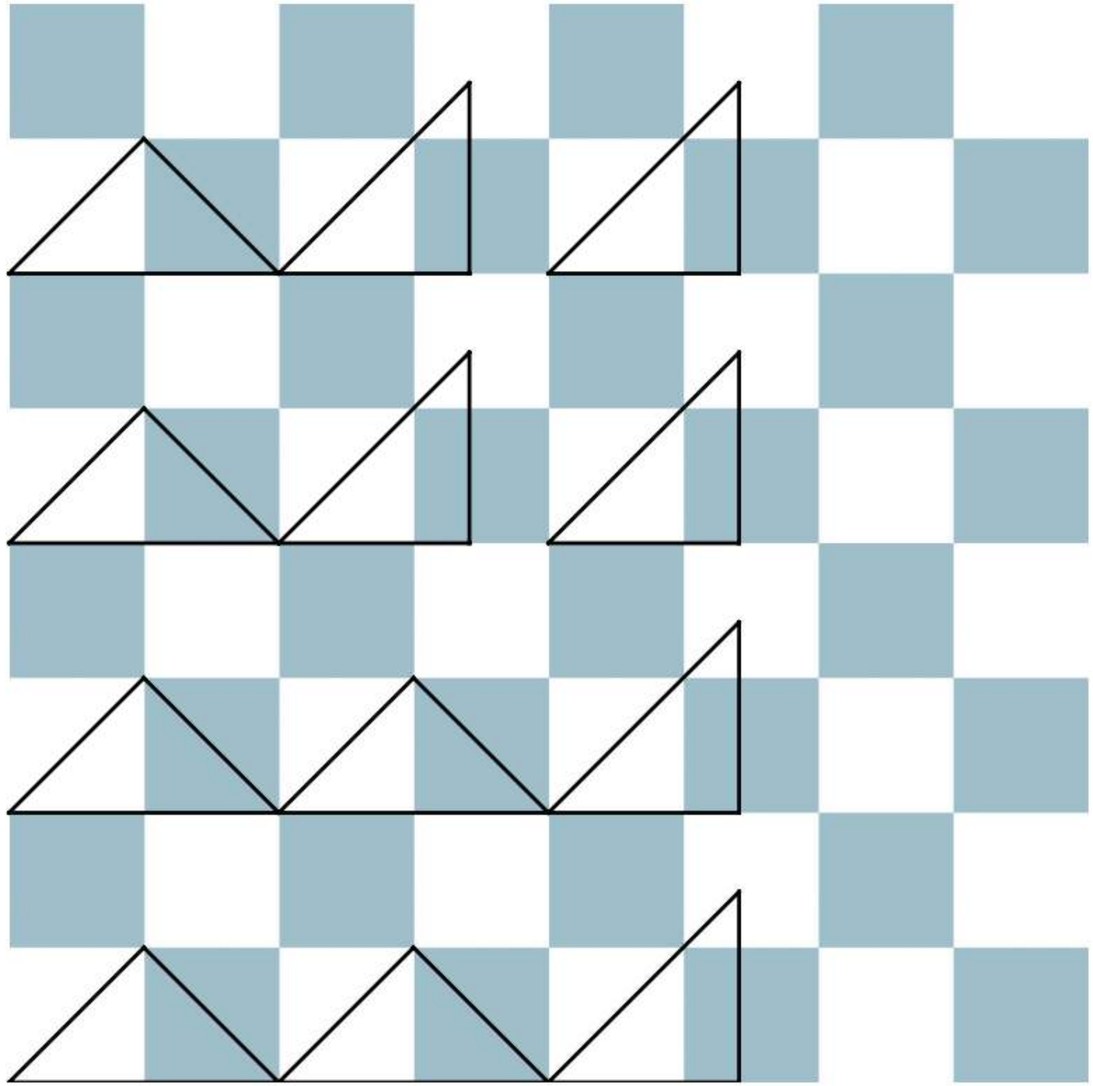
- All triangles mapped to plane w/ trivial parameterization (isometric!)
- Translation and uniform scaling does not affect the angles



Triangle Soup Parameterization

Angle distortion?

- All triangles mapped to plane w/ trivial parameterization (isometric!)
- Translation and uniform scaling does not affect the angles
- Conclusion: no angle distortion!

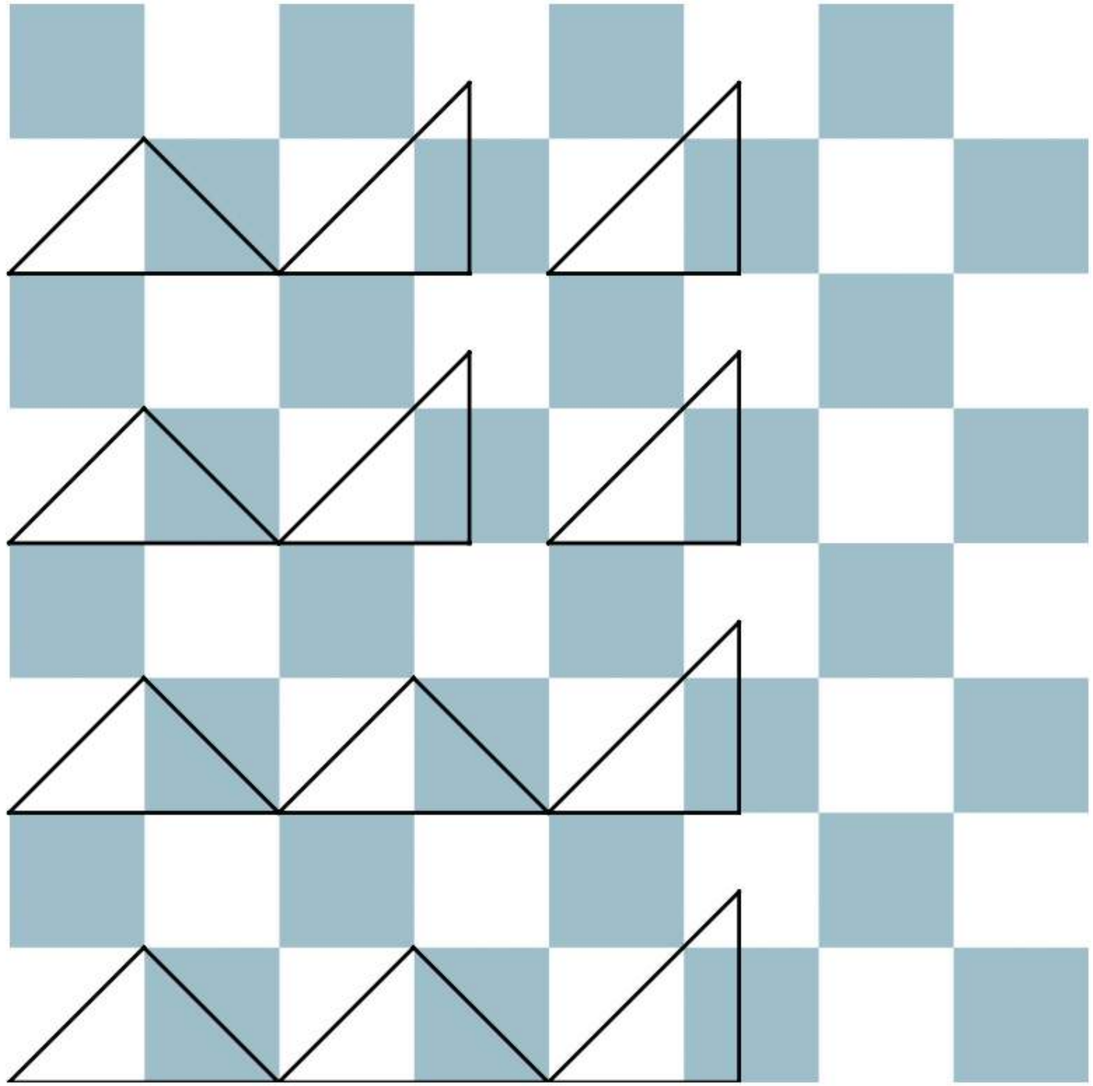


Triangle Soup Parameterization

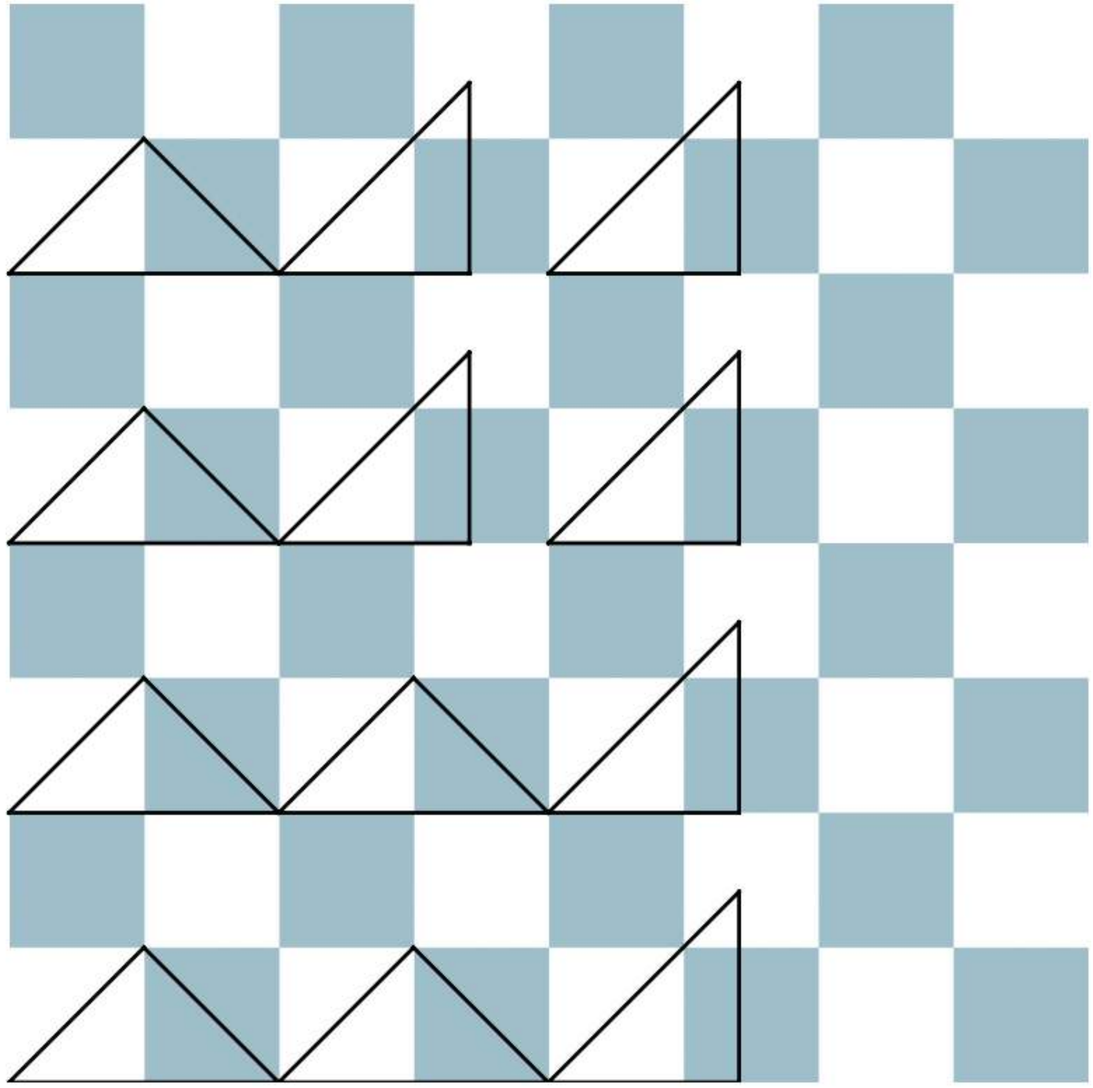
Angle distortion?

- All triangles mapped to plane w/ trivial parameterization (isometric!)
- Translation and uniform scaling does not affect the angles
- Conclusion: no angle distortion!
- Can verify this by checking the SVD of the Jacobian:

$$\sigma_1 = \sigma_2$$

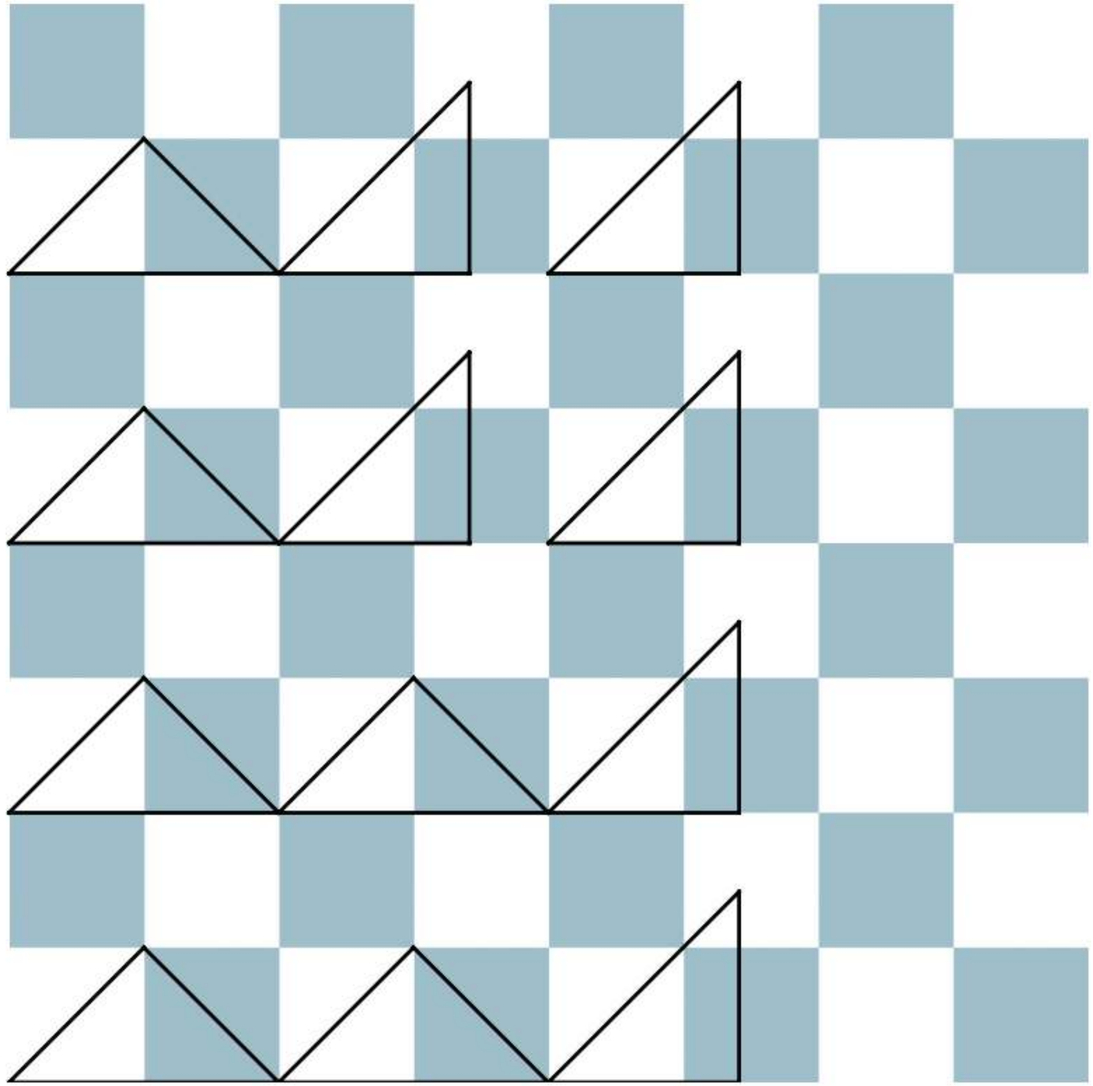


Triangle Soup Parameterization



Triangle Soup Parameterization

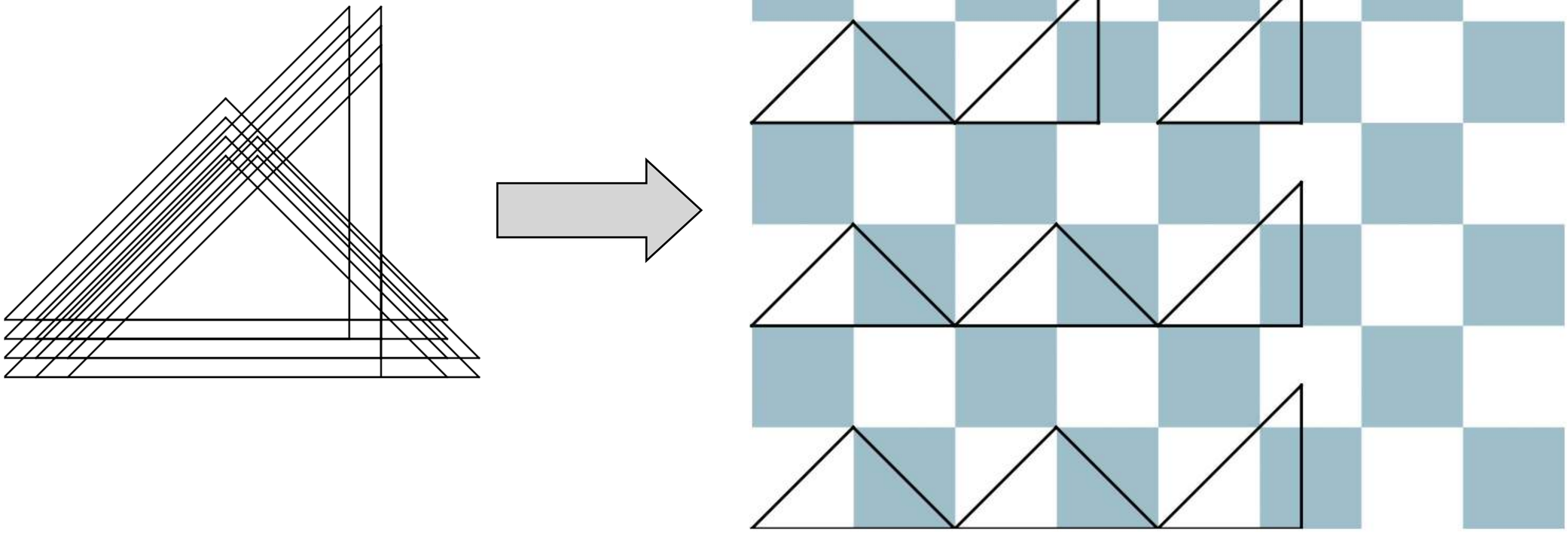
Area distortion?



Triangle Soup Parameterization

Area distortion?

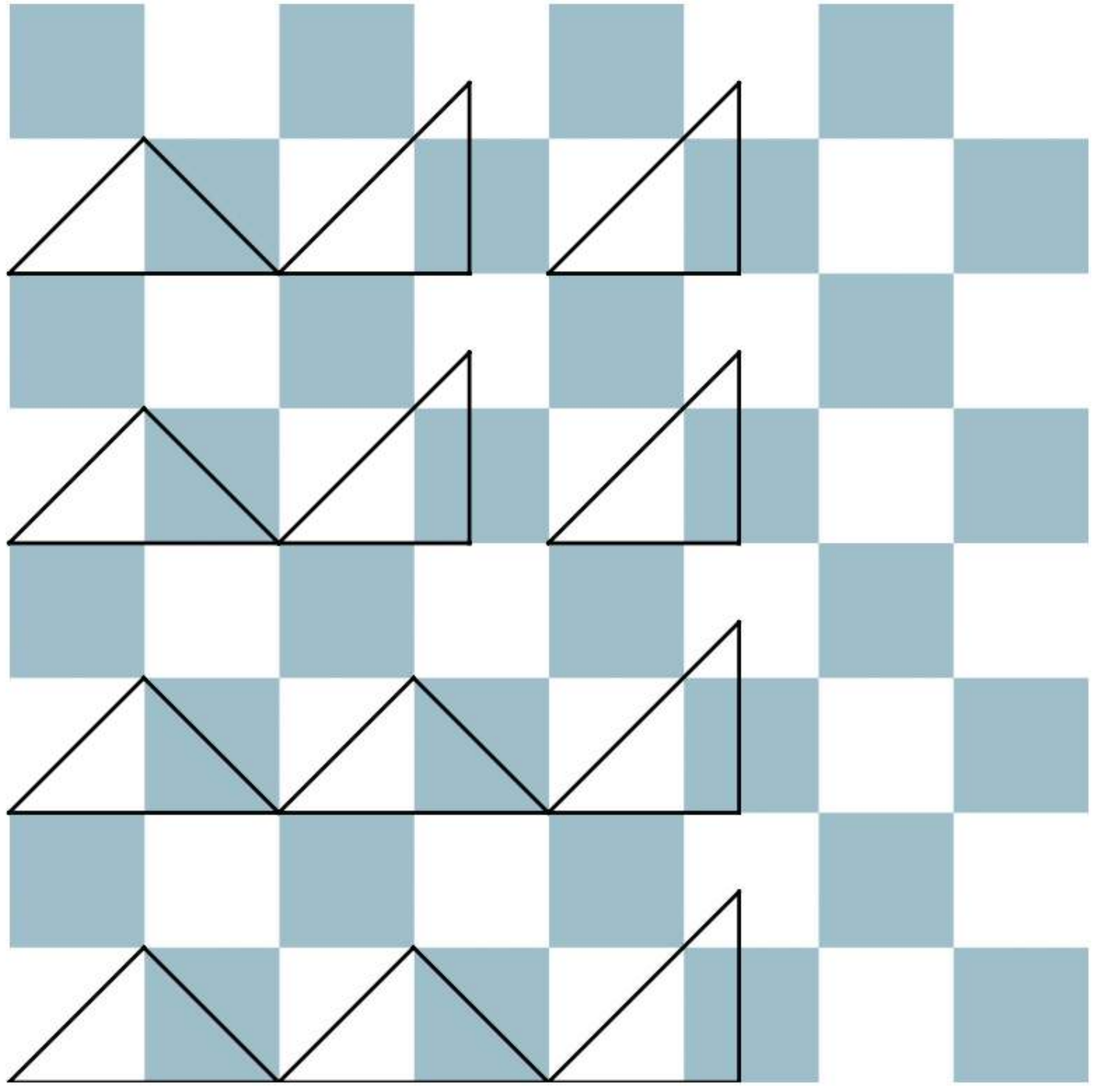
- When packing the triangles into the texture image, we might need to resale them so that they fit…



Triangle Soup Parameterization

Area distortion?

- When packing the triangles into the texture image, we might need to resale them so that they fit…
- This means that the areas are not guaranteed to be the same

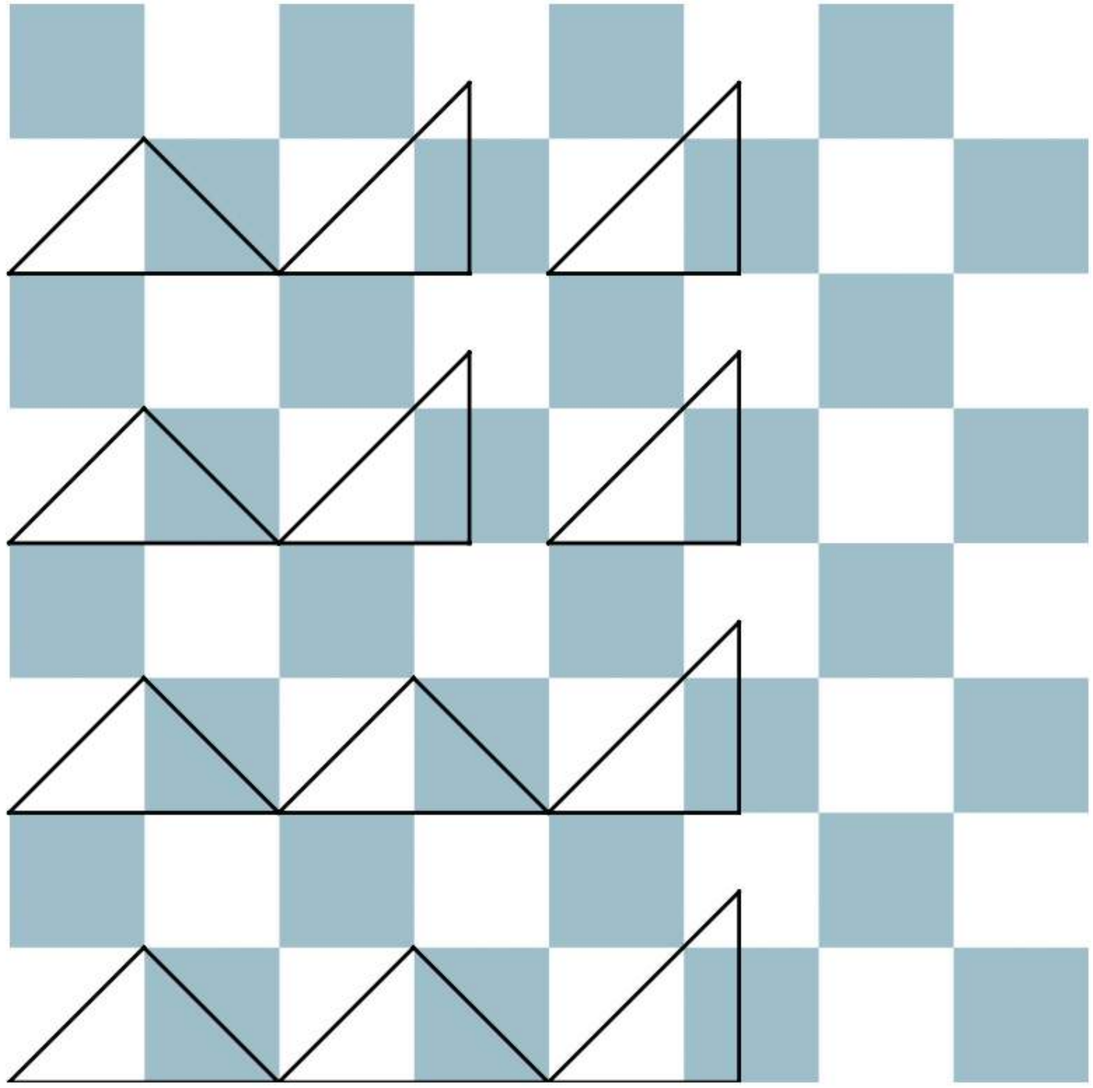


Triangle Soup Parameterization

Area distortion?

- When packing the triangles into the texture image, we might need to resale them so that they fit…
- This means that the areas are not guaranteed to be the same
- We can check the SVD values again to verify this:

$$\sigma_1 \sigma_2 \neq 1$$



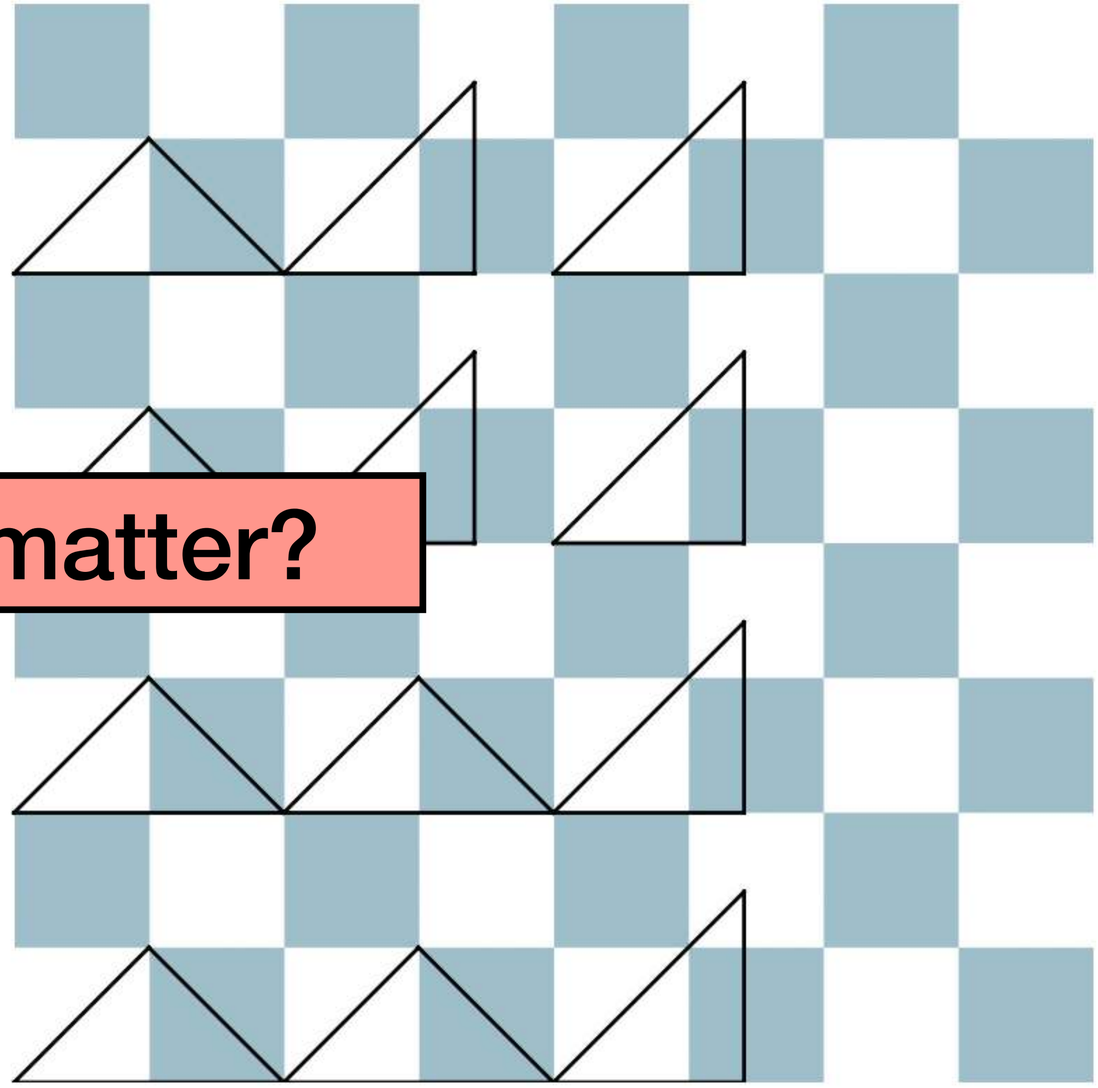
Triangle Soup Parameterization

Area distortion?

- When packing the triangles into the texture image, we might need to resale them so that they fit…
- This means that the areas are not guaranteed to be the
- We can check the SVD values again to verify this:

$$\sigma_1 \sigma_2 \neq 1$$

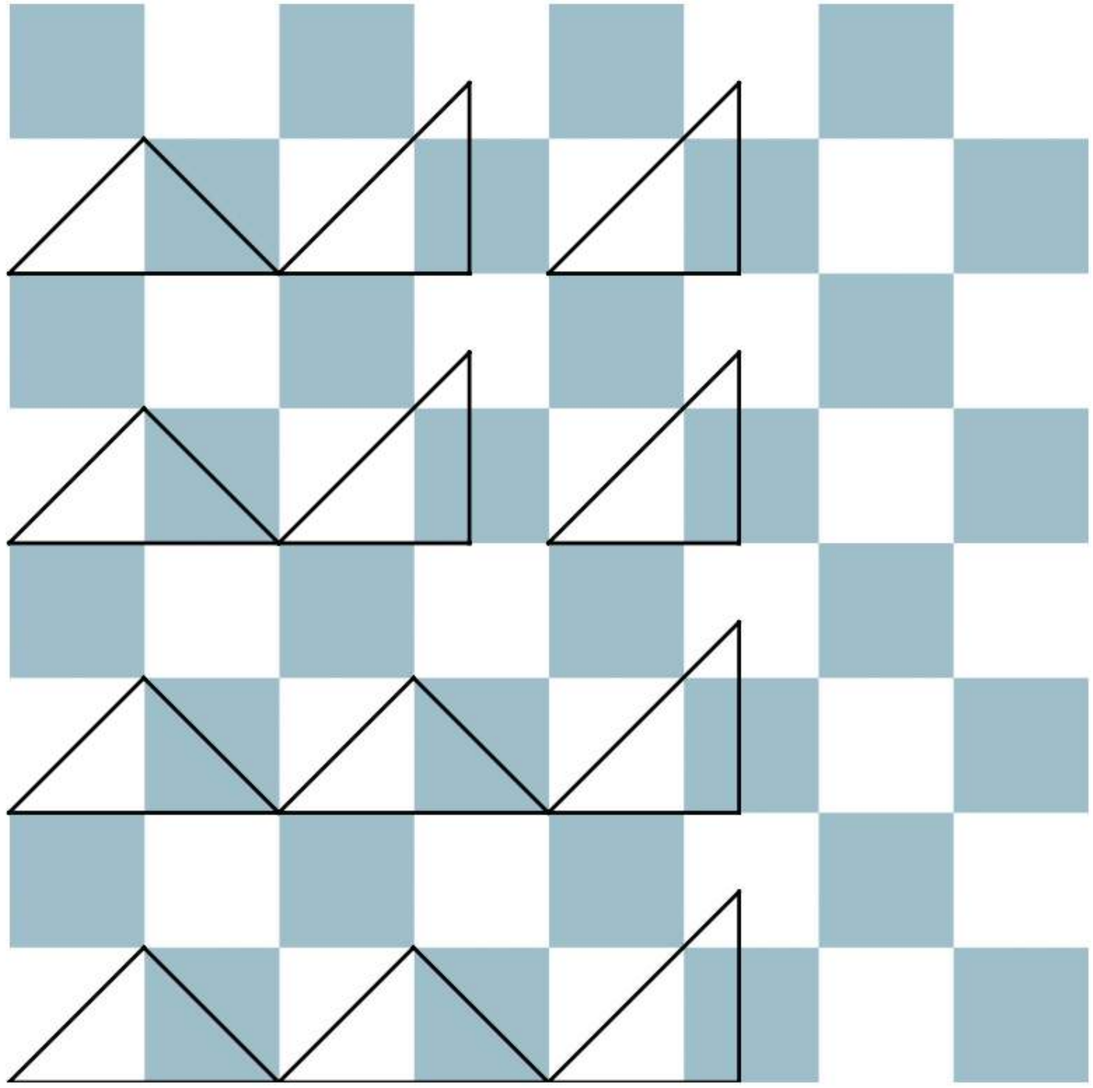
Does this matter?



Triangle Soup Parameterization

Area distortion?

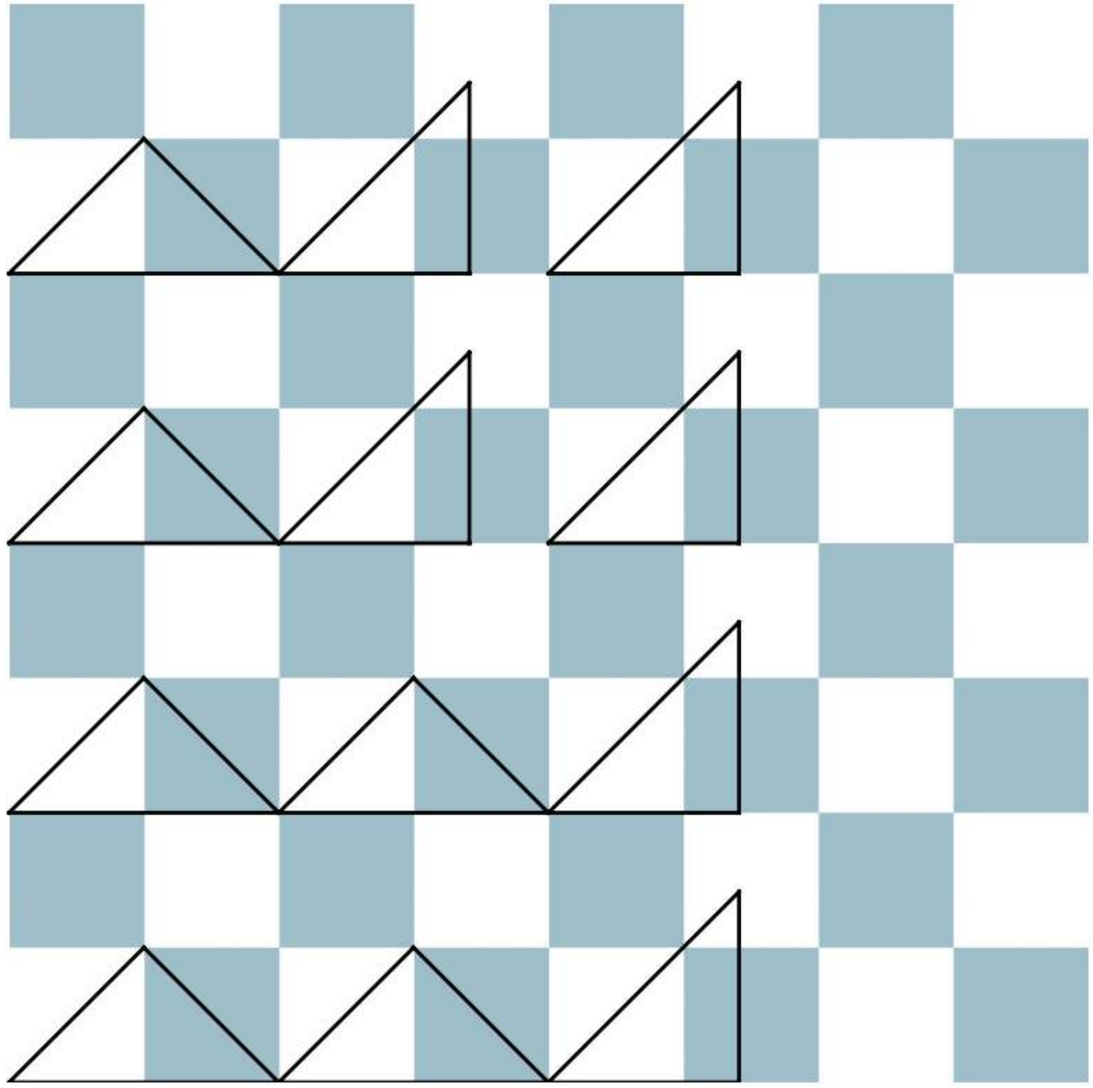
- Intuitively, what we care about for our texture map is that the surface is sampled evenly



Triangle Soup Parameterization

Area distortion?

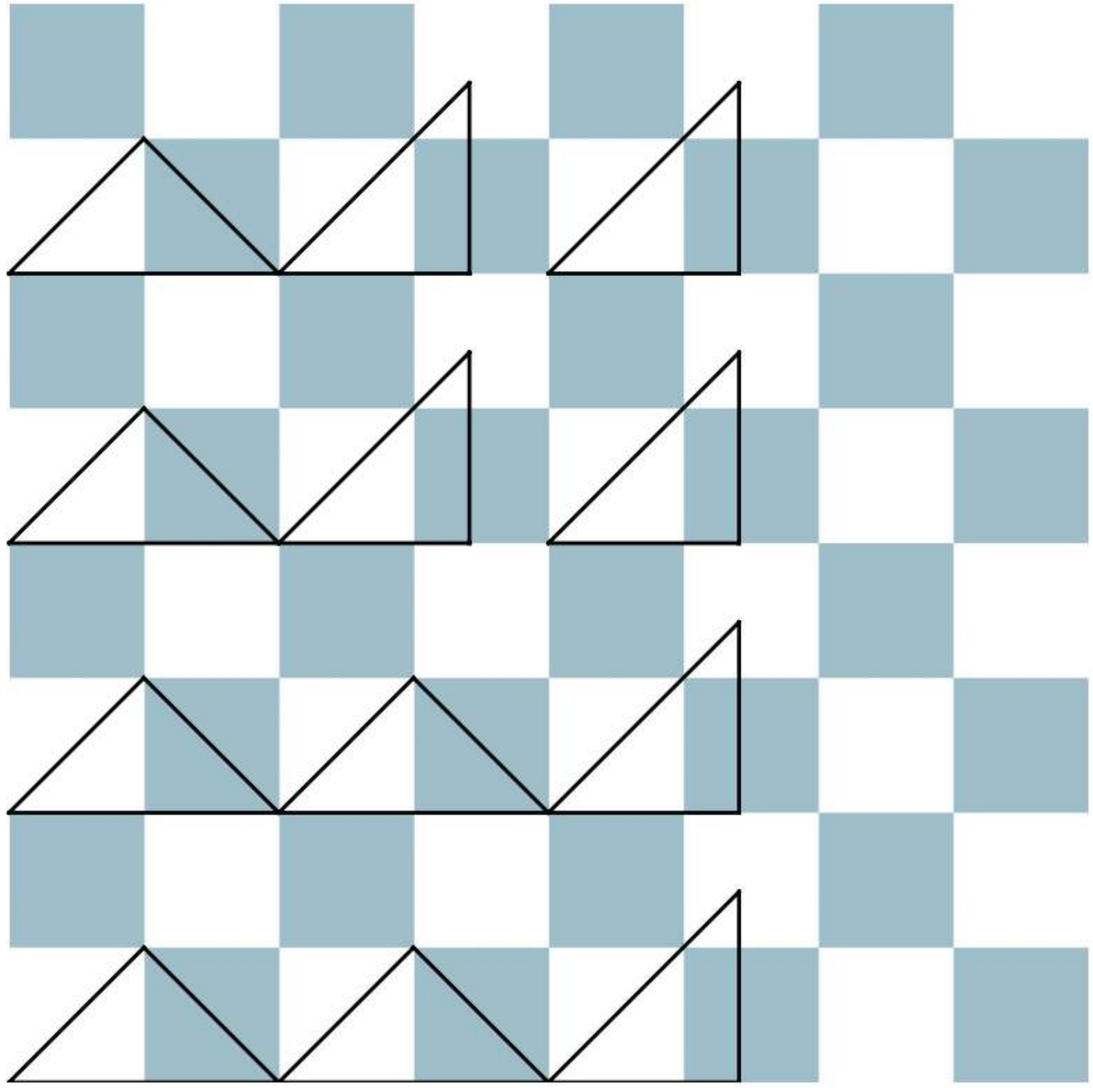
- Intuitively, what we care about for our texture map is that the surface is sampled evenly
- All triangles get scaled by the same factor!



Triangle Soup Parameterization

Area distortion?

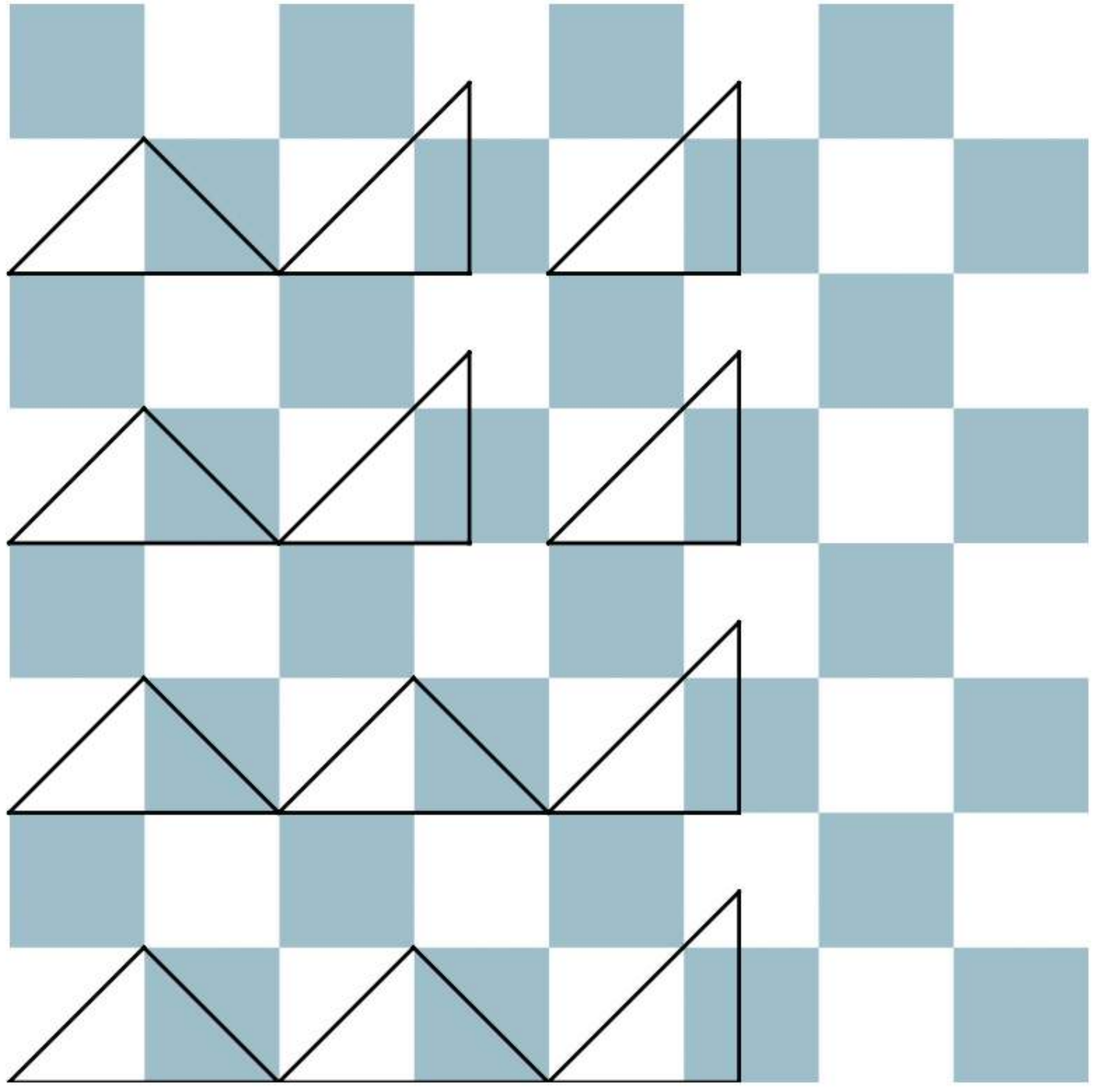
- Intuitively, what we care about for our texture map is that the surface is sampled evenly
- All triangles get scaled by the same factor!
- The area of each triangle is preserved up to some global scaler



Triangle Soup Parameterization

Area distortion?

- Intuitively, what we care about for our texture map is that the surface is sampled evenly
- All triangles get scaled by the same factor!
- The area of each triangle is preserved up to some global scaler
- This means that we will still get an even sampling of our surface!



Can we connect this (relative) area preserving parameterization with the SVD?

Can we connect this (relative) area preserving parameterization with the SVD?

- Since our parameterization is conformal, $\sigma_1 = \sigma_2$ for all triangles

Can we connect this (relative) area preserving parameterization with the SVD?

- Since our parameterization is conformal, $\sigma_1 = \sigma_2$ for all triangles
- Since we scale by the same factor for all triangles, σ_1 and σ_2 are constant across all triangles.

Can we connect this (relative) area preserving parameterization with the SVD?

- Since our parameterization is conformal, $\sigma_1 = \sigma_2$ for all triangles
- Since we scale by the same factor for all triangles, σ_1 and σ_2 are constant across all triangles.
- Thus, $\sigma_1 = \sigma_2 = c$ for all triangles for some constant c

Can we connect this (relative) area preserving parameterization with the SVD?

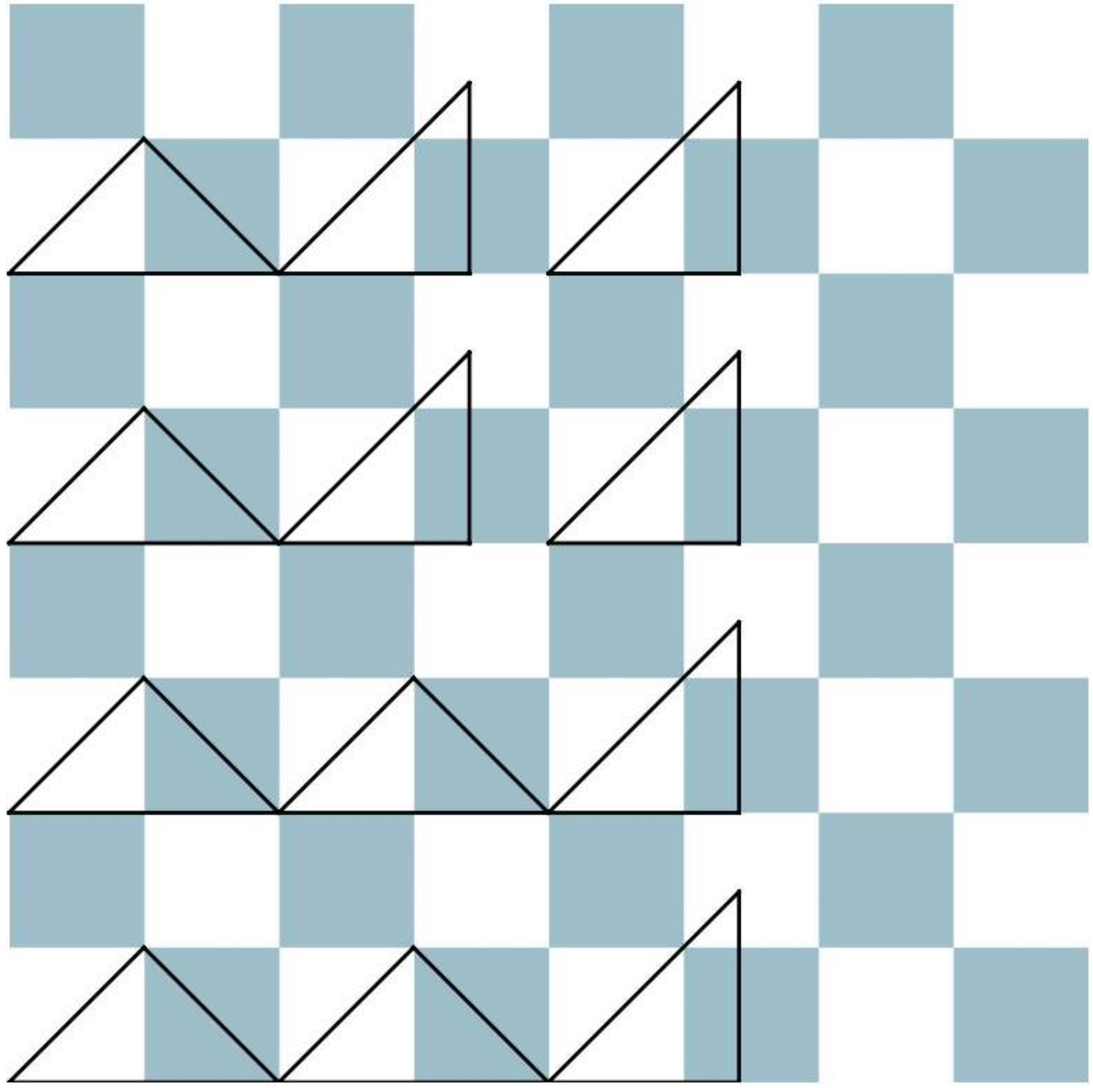
- Since our parameterization is conformal, $\sigma_1 = \sigma_2$ for all triangles
- Since we scale by the same factor for all triangles, σ_1 and σ_2 are constant across all triangles.
- Thus, $\sigma_1 = \sigma_2 = c$ for all triangles for some constant c
- Can verify this by checking the SVD! (See exercise 104)

Can we connect this (relative) area preserving parameterization with the SVD?

- Since our parameterization is conformal, $\sigma_1 = \sigma_2$ for all triangles
- Since we scale by the same factor for all triangles, σ_1 and σ_2 are constant across all triangles.
- Thus, $\sigma_1 = \sigma_2 = c$ for all triangles for some constant c
- Can verify this by checking the SVD! (See exercise 104)
- This constant c is our scale factor!

Triangle Soup Parameterization

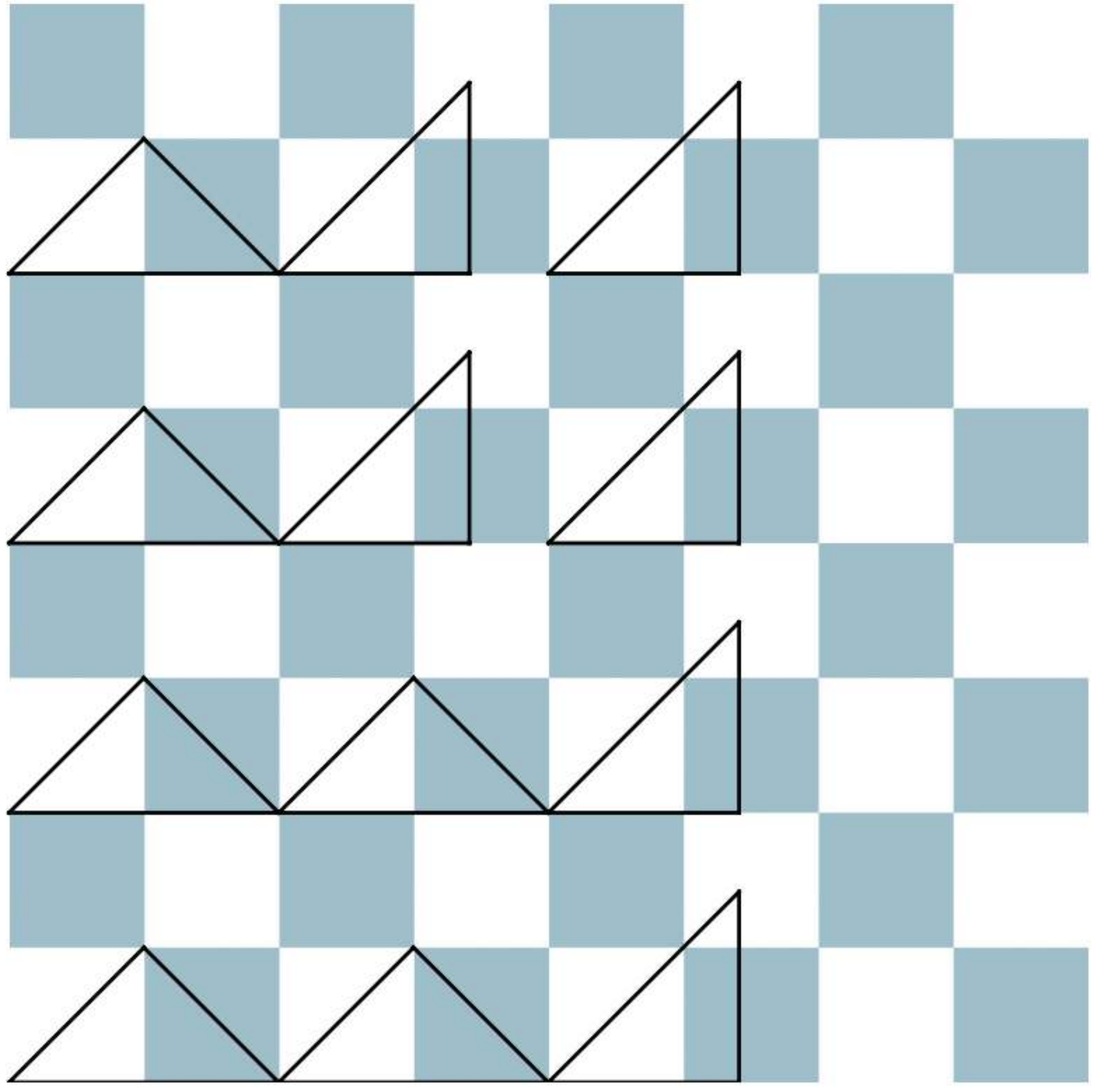
Drawbacks



Triangle Soup Parameterization

Drawbacks

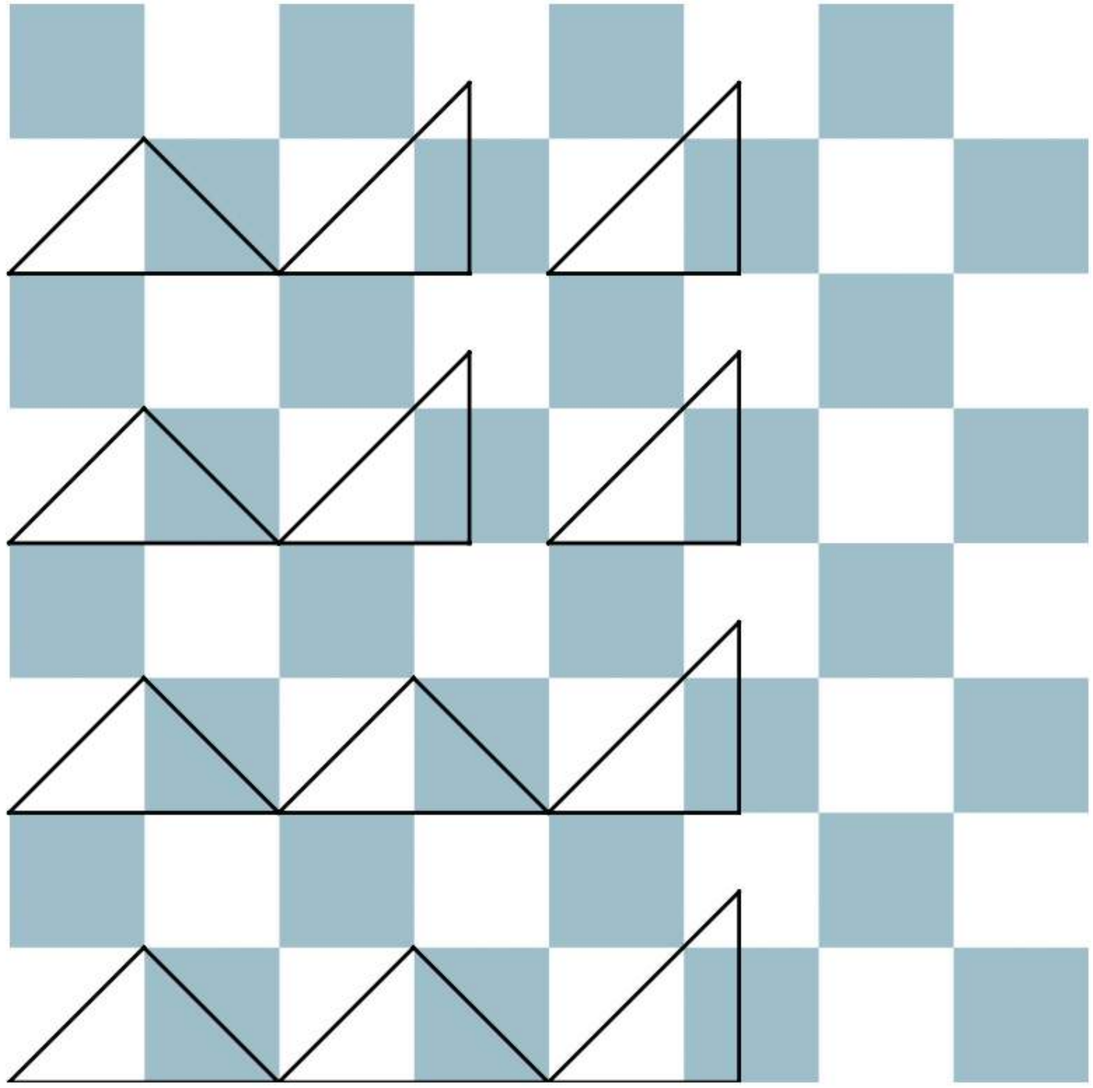
- Maximum amount of cuts (texture discontinuities)



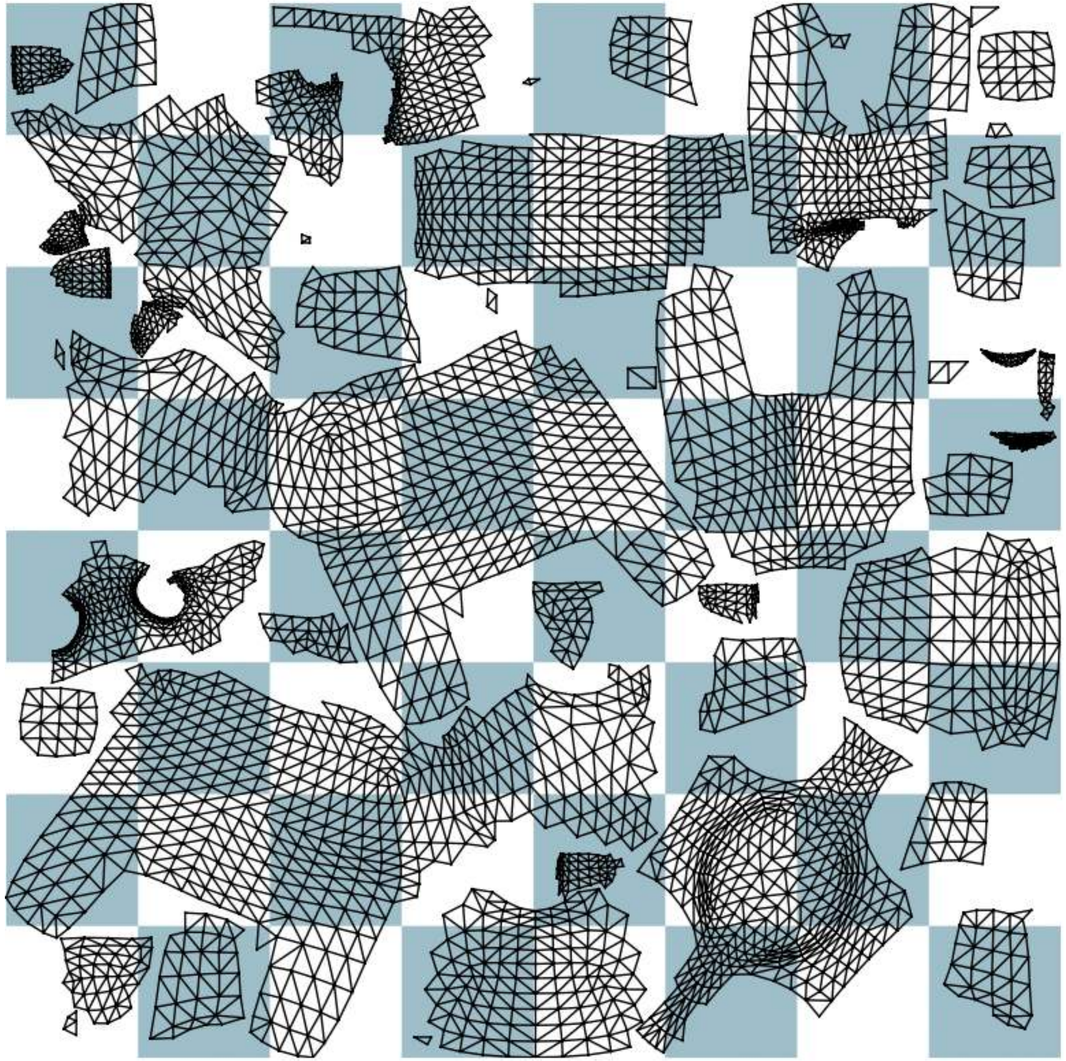
Triangle Soup Parameterization

Drawbacks

- Maximum amount of cuts
  (texture discontinuities)
- Lots of gaps
  (inefficient use of texture space)

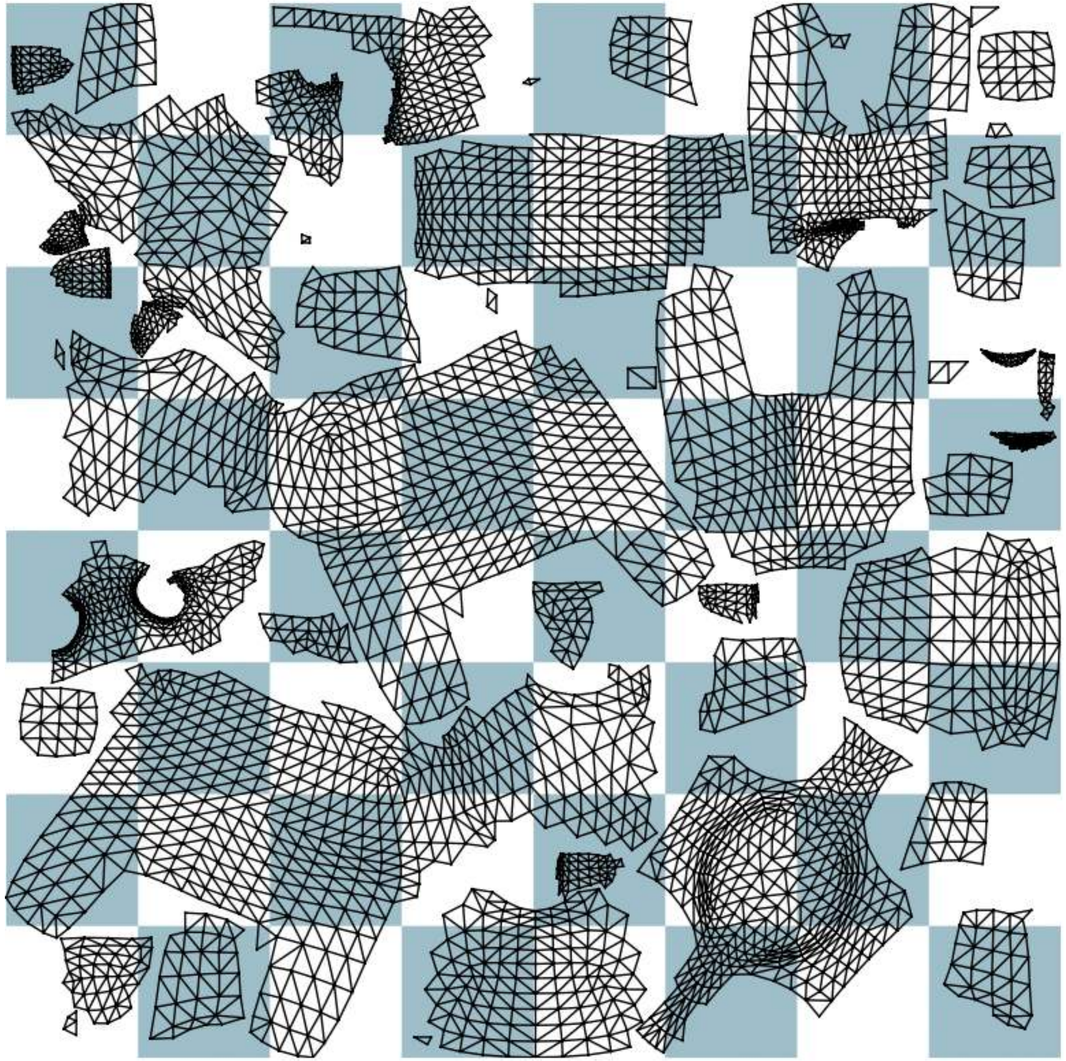


XAtlas Parameterization



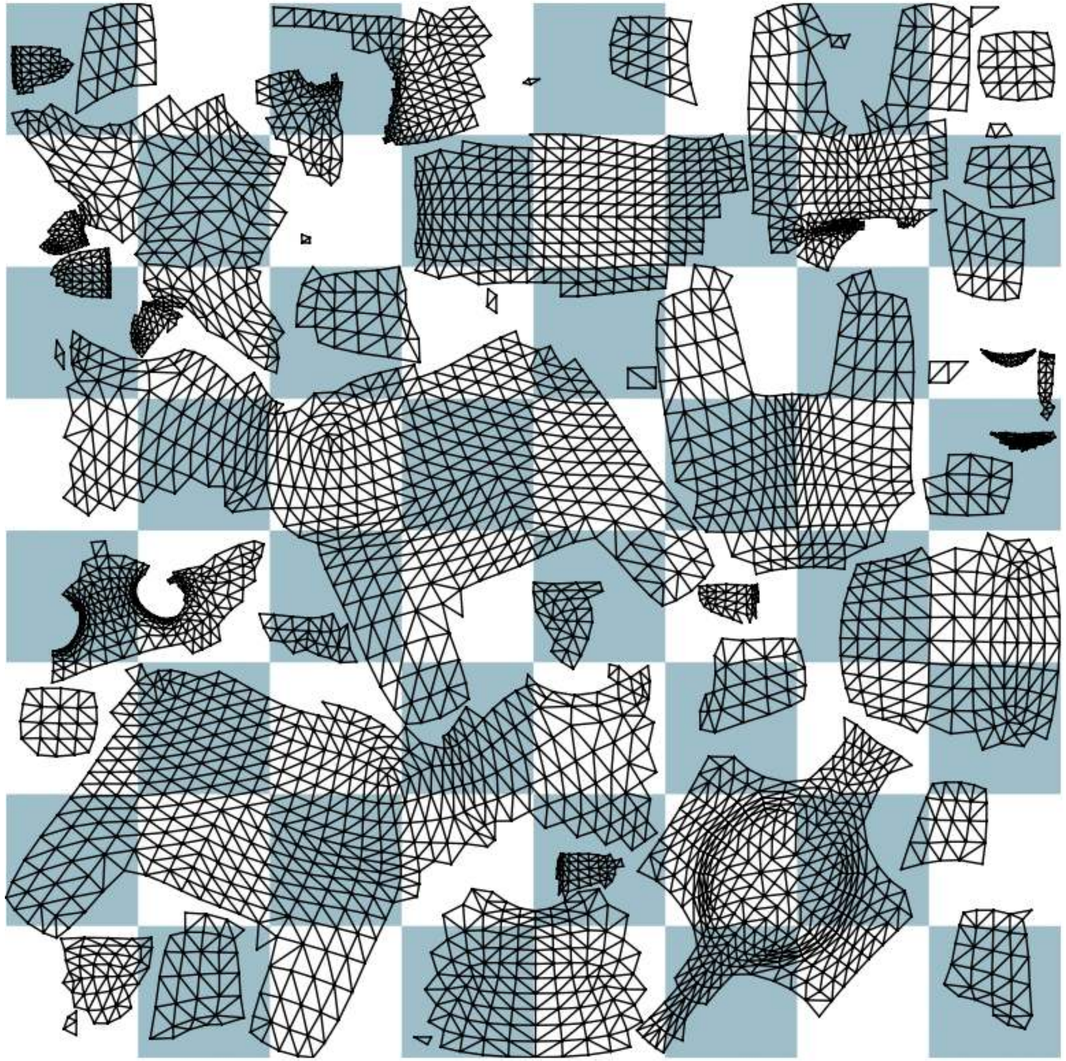
XAtlas Parameterization

- Tries to tradeoff between distortion and cuts



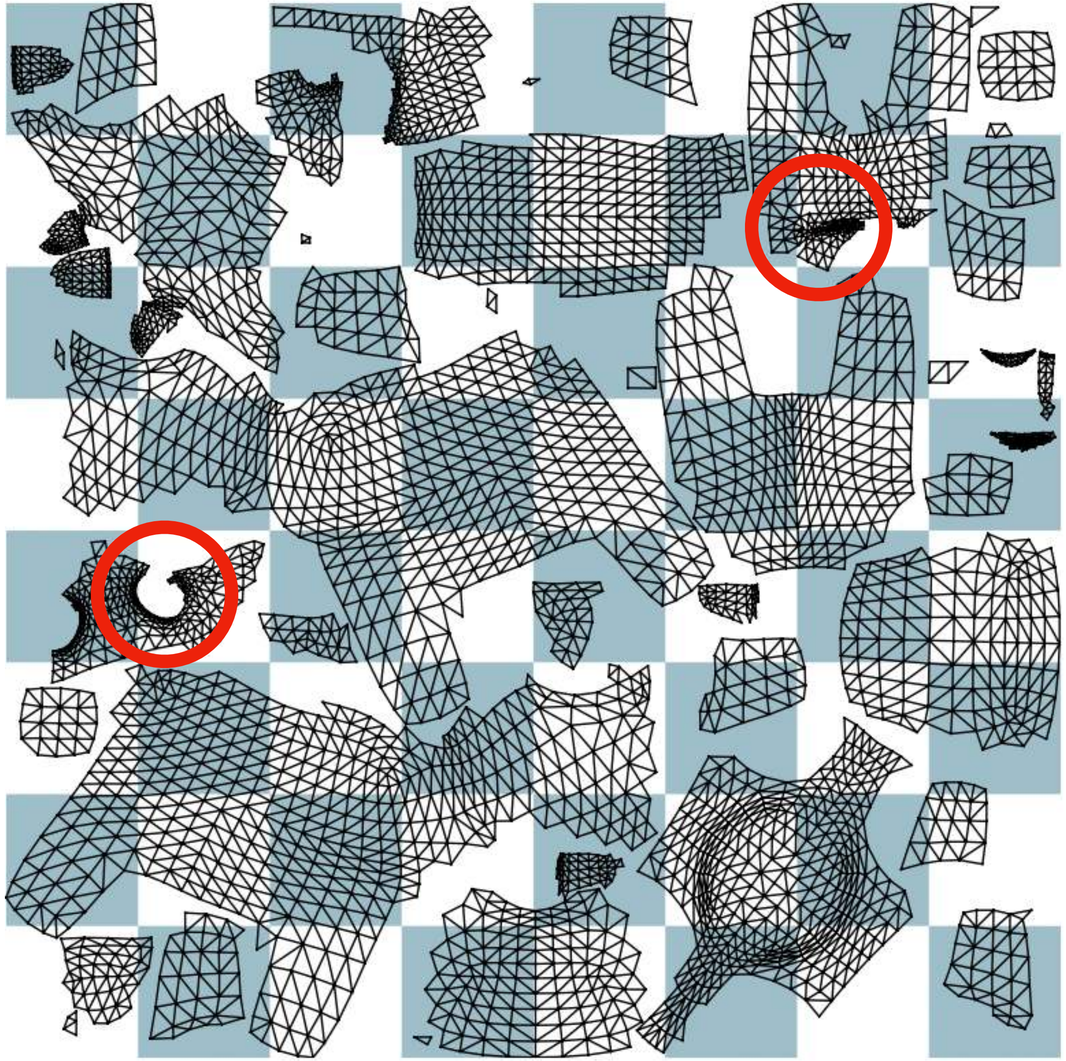
XAtlas Parameterization

- Tries to tradeoff between distortion and cuts
- Much fewer cuts!

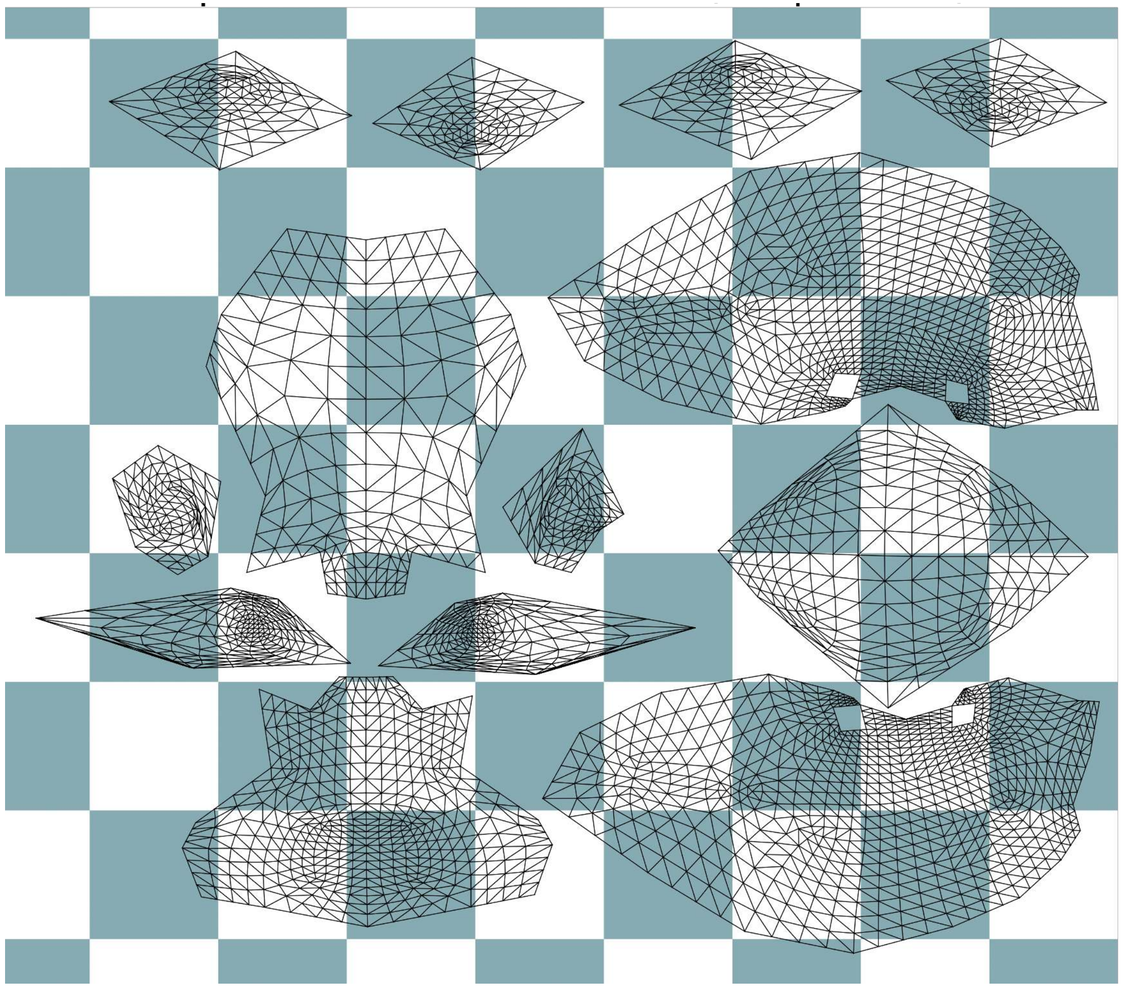


XAtlas Parameterization

- Tries to tradeoff between distortion and cuts
- Much fewer cuts!
- Low distortion, except in a few places…



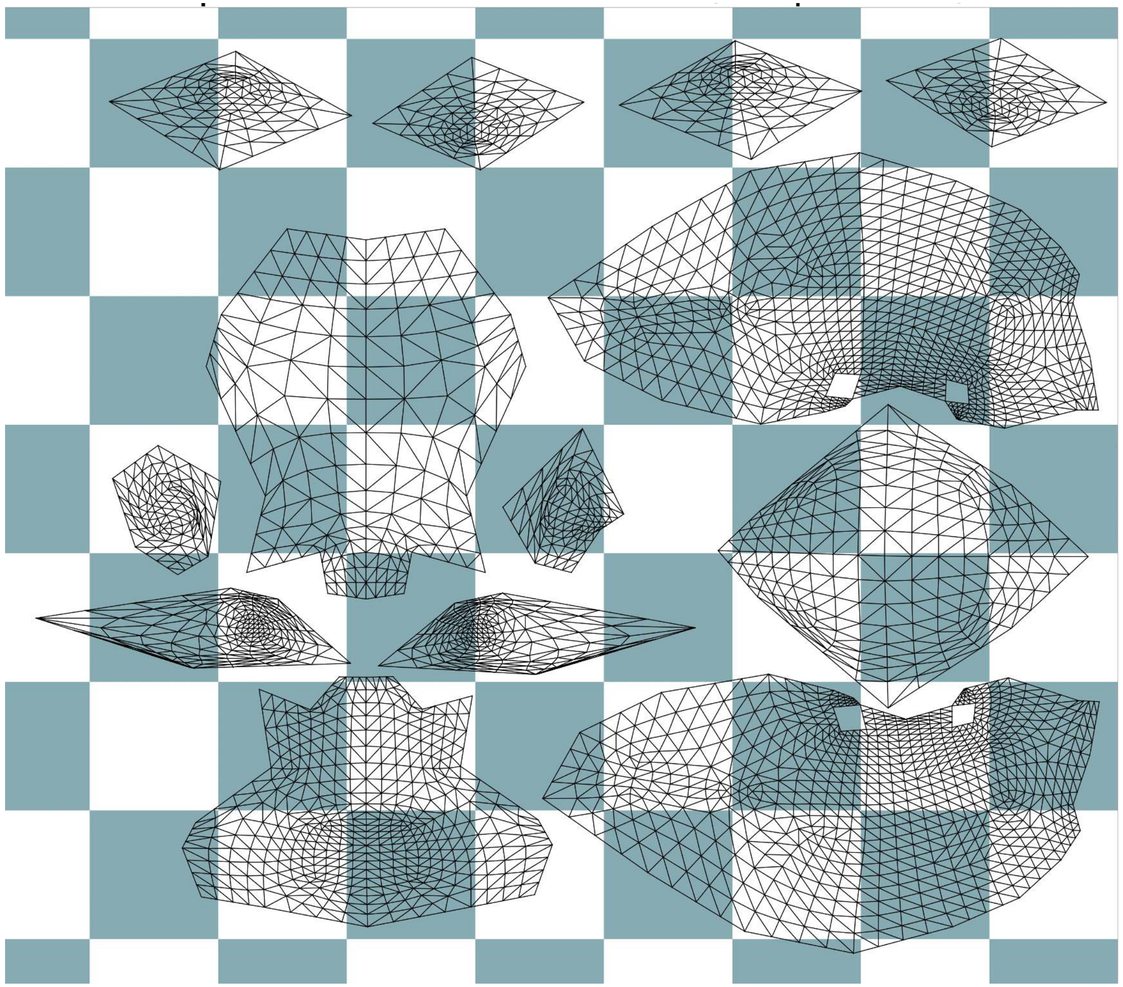
Manual Parameterization



Spot the Cow, Keenan Crane

Manual Parameterization

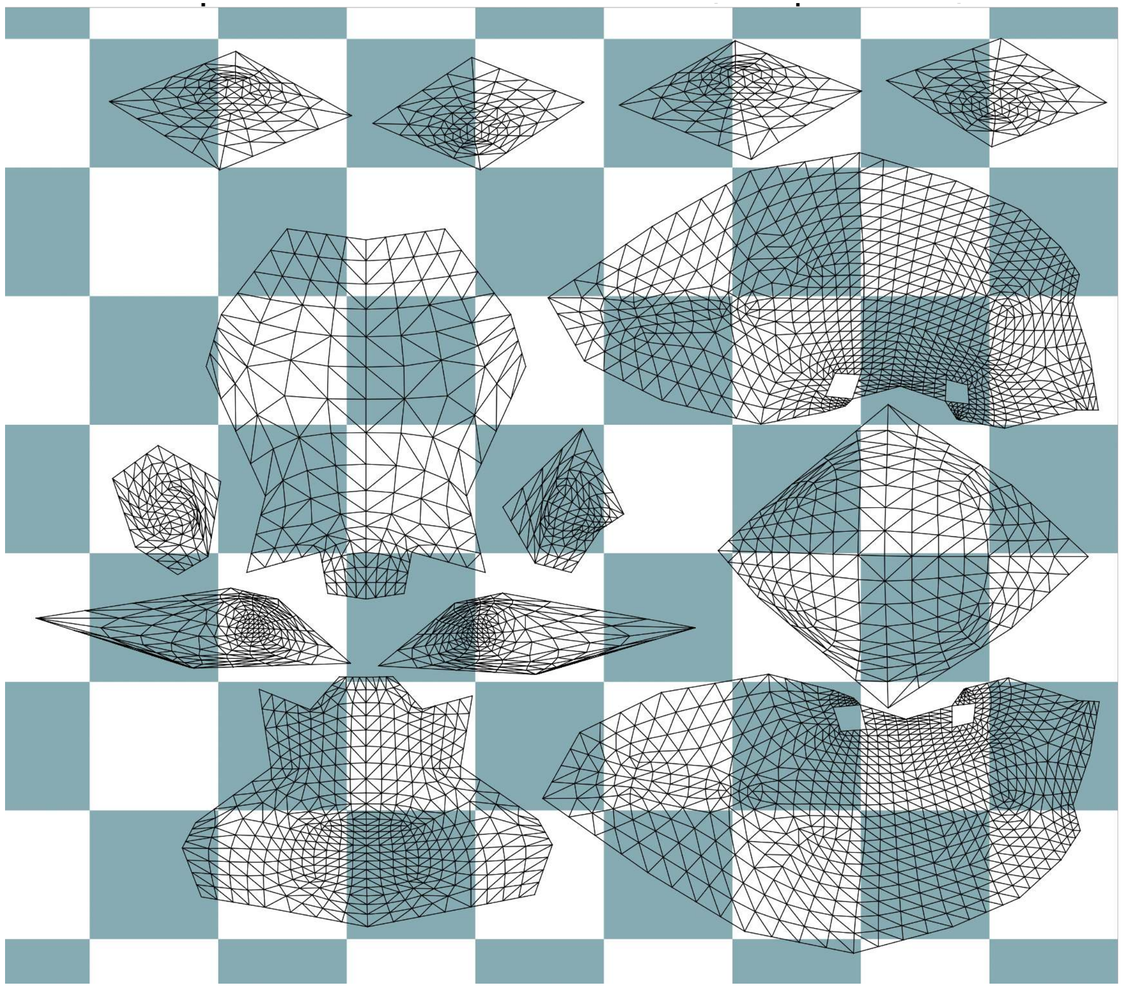
- Manually place cuts to mostly lie on semantic boundaries



Spot the Cow, Keenan Crane

Manual Parameterization

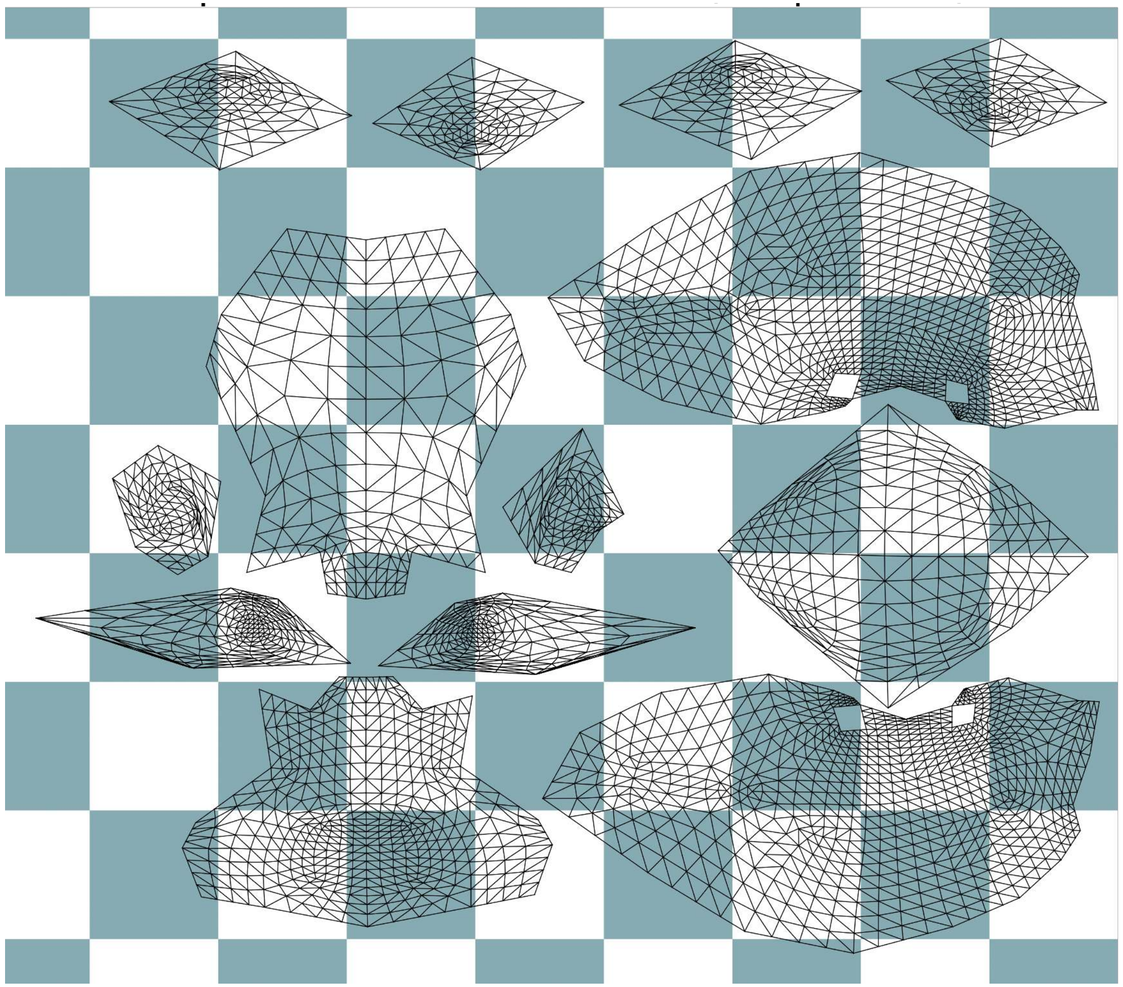
- Manually place cuts to mostly lie on semantic boundaries
- Hide distortion in non-visible or unimportant regions



Spot the Cow, Keenan Crane

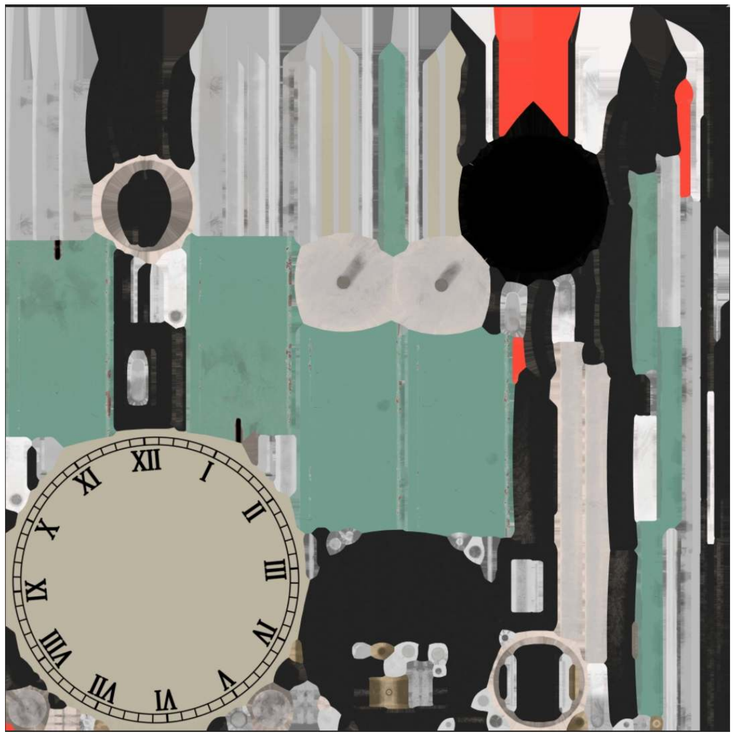
Manual Parameterization

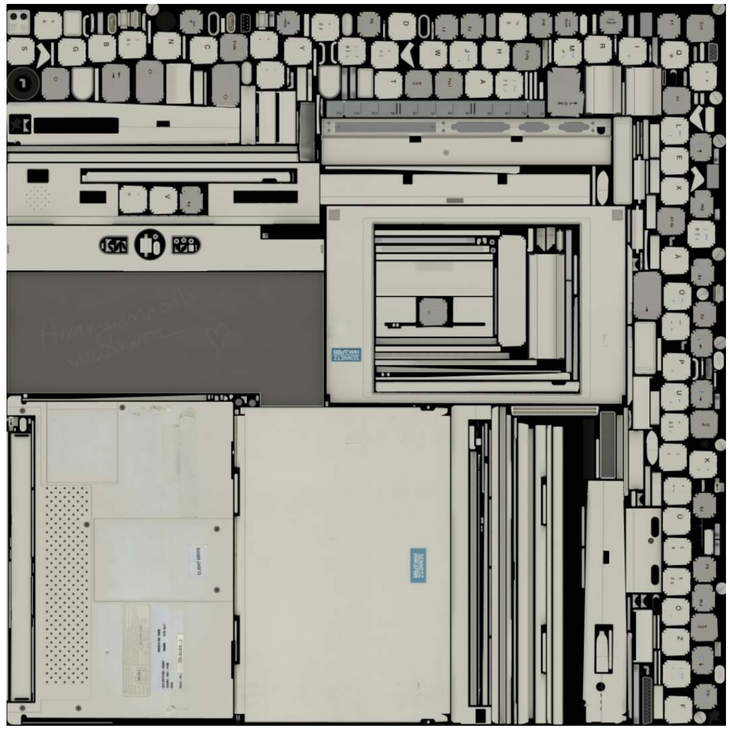
- Manually place cuts to mostly lie on semantic boundaries
- Hide distortion in non-visible or unimportant regions
- Not scalable…



Spot the Cow, Keenan Crane

Where do these textures come from?

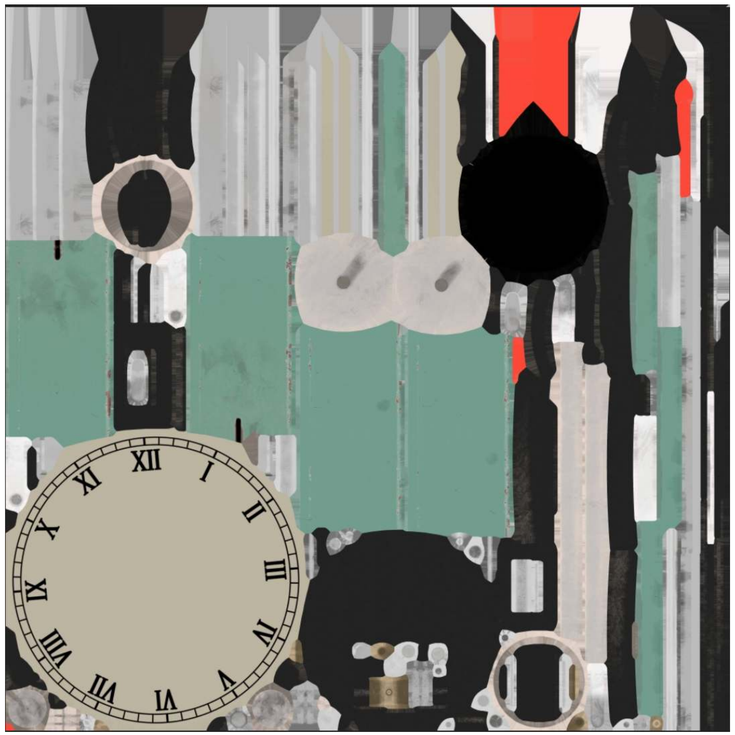


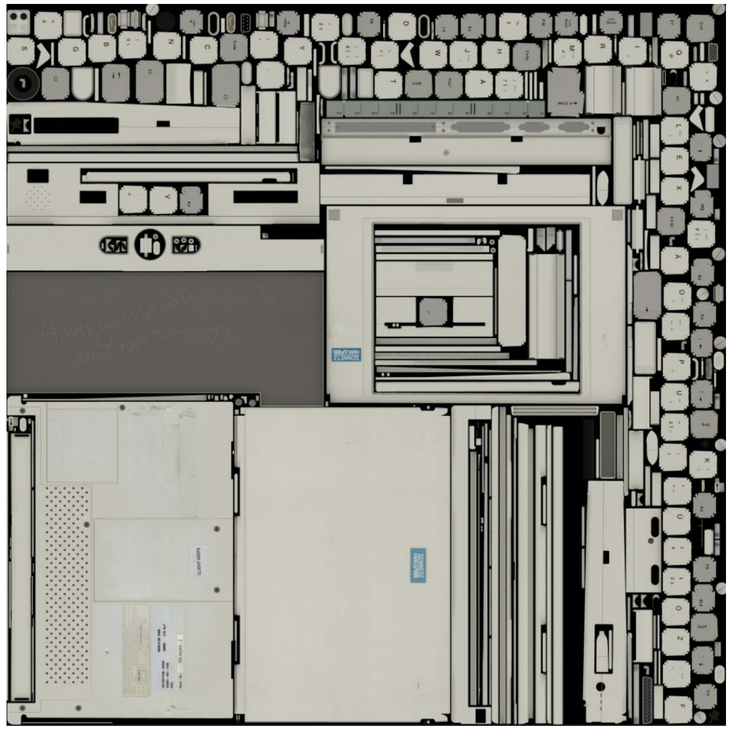


James Ray https://polyhaven.com/ Arrangemonk

Where do these textures come from?

Usually hand-painted by an artist



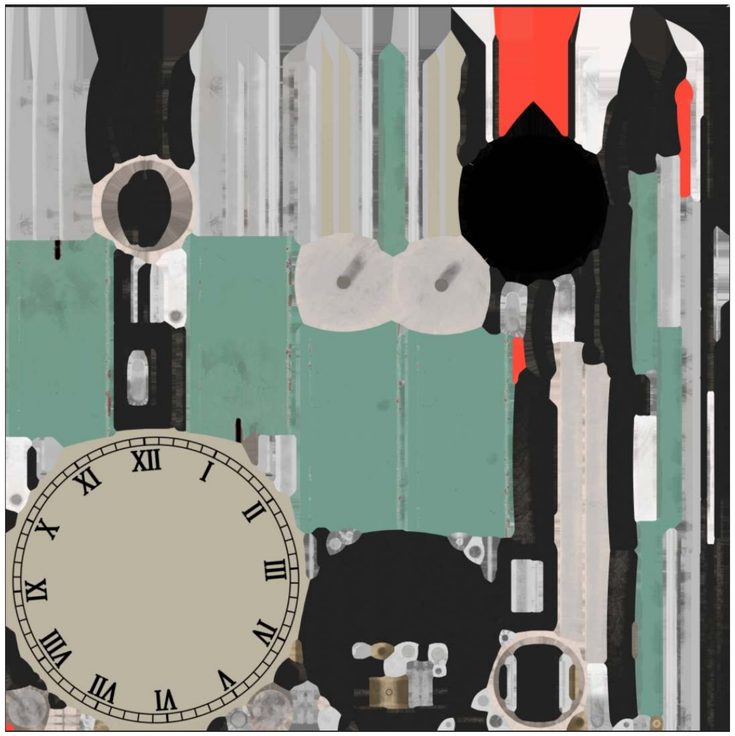


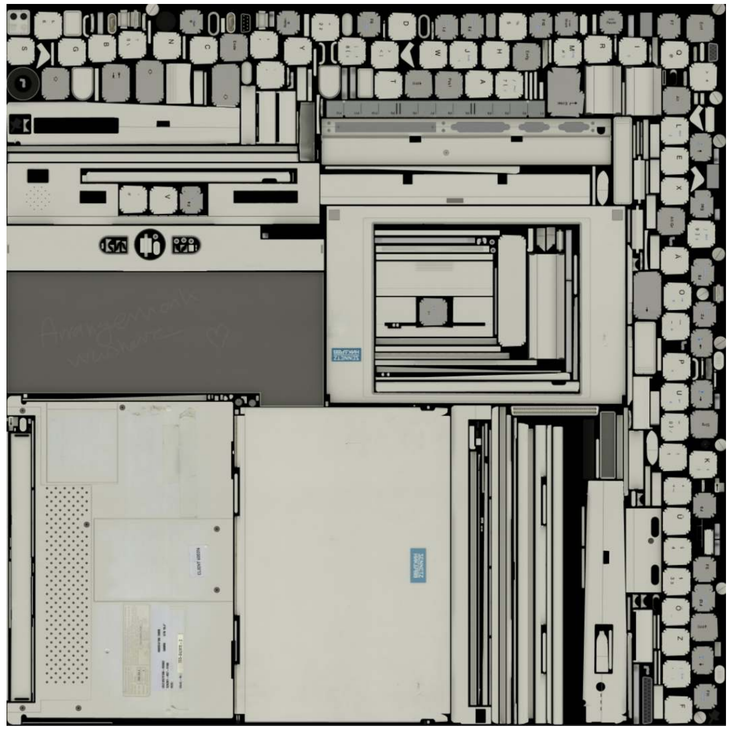
James Ray https://polyhaven.com/ Arrangemonk

Where do these textures come from?

Usually hand-painted by an artist

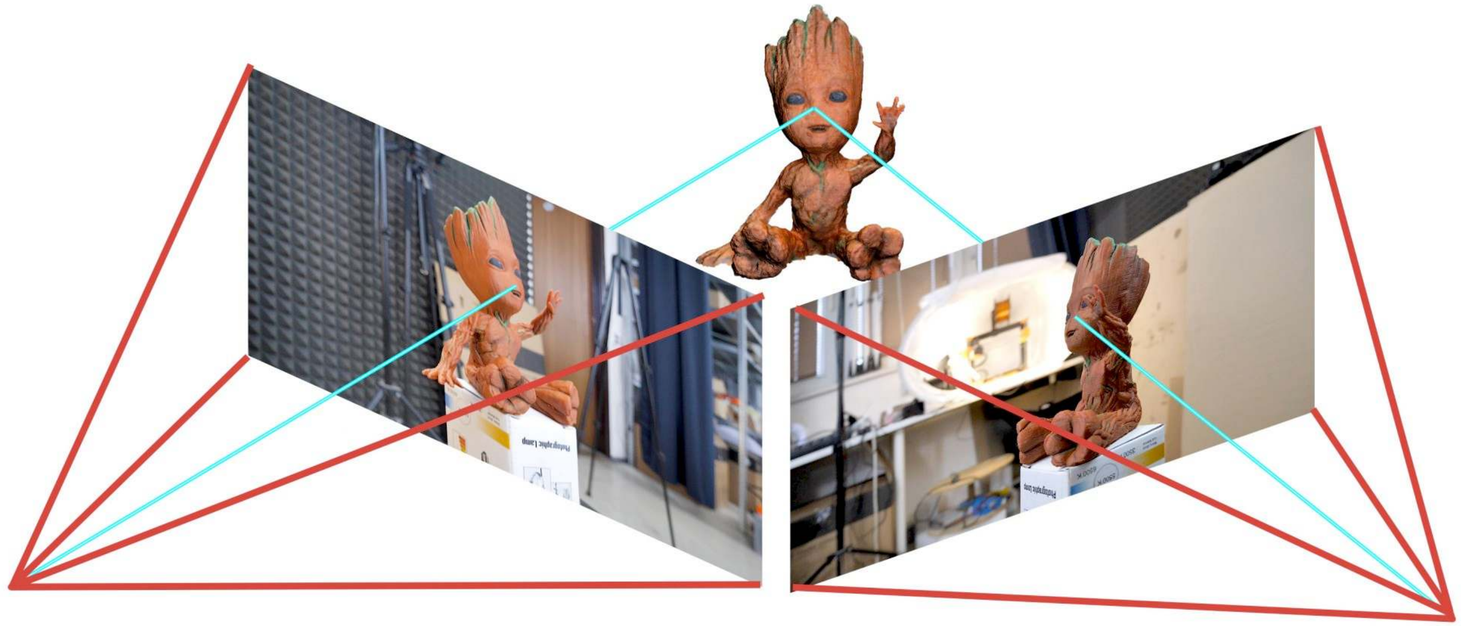
What if we want to replicate a texture seen on image of some existing object?





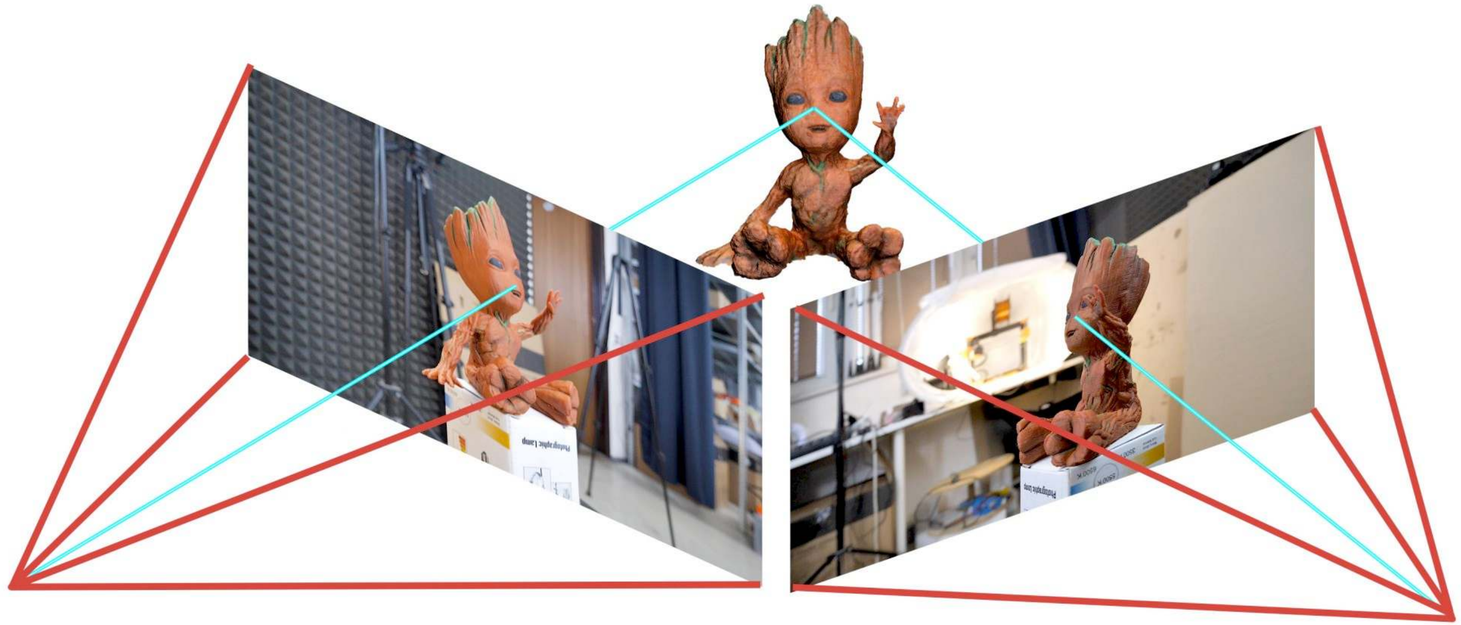
James Ray https://polyhaven.com/ Arrangemonk

Inverse Problem — photogrammetry



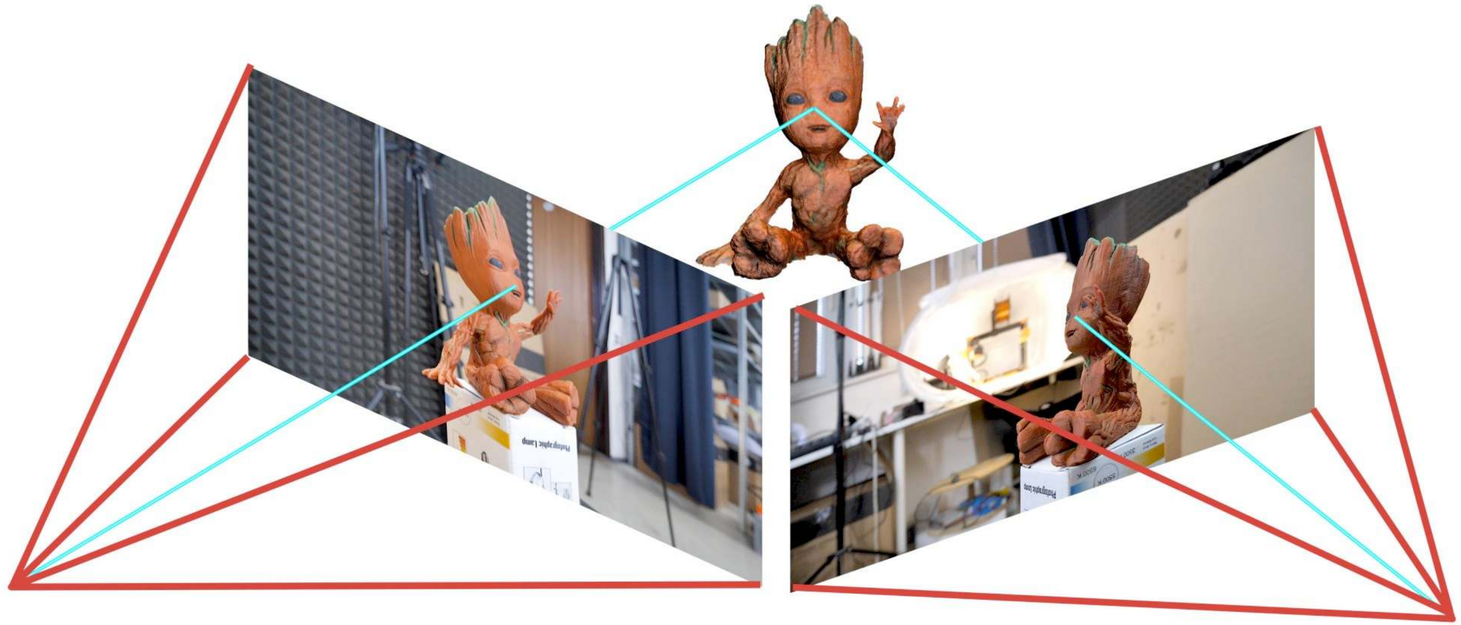
Inverse Problem — photogrammetry

- Want to learn a texture given 2D images of a 3D object



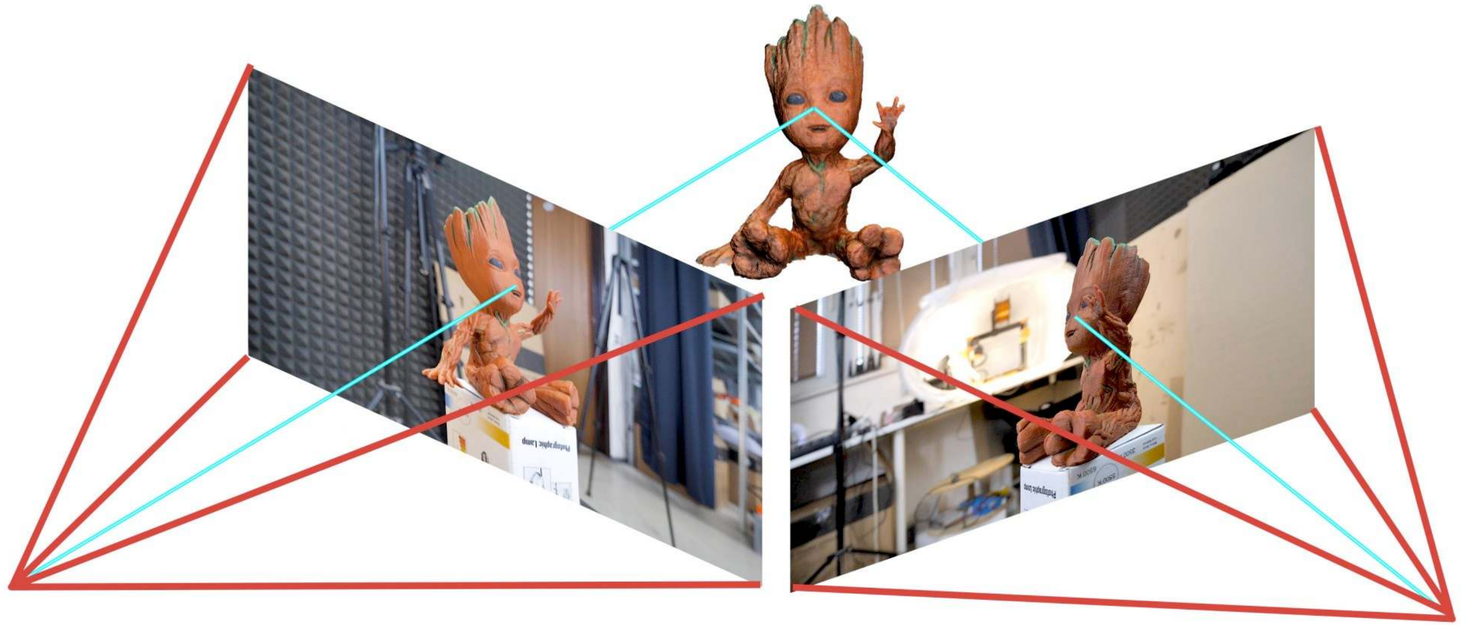
Inverse Problem — photogrammetry

- Want to learn a texture given 2D images of a 3D object
- Need to accumulate information from multiple (*possibly inconsistent*) views



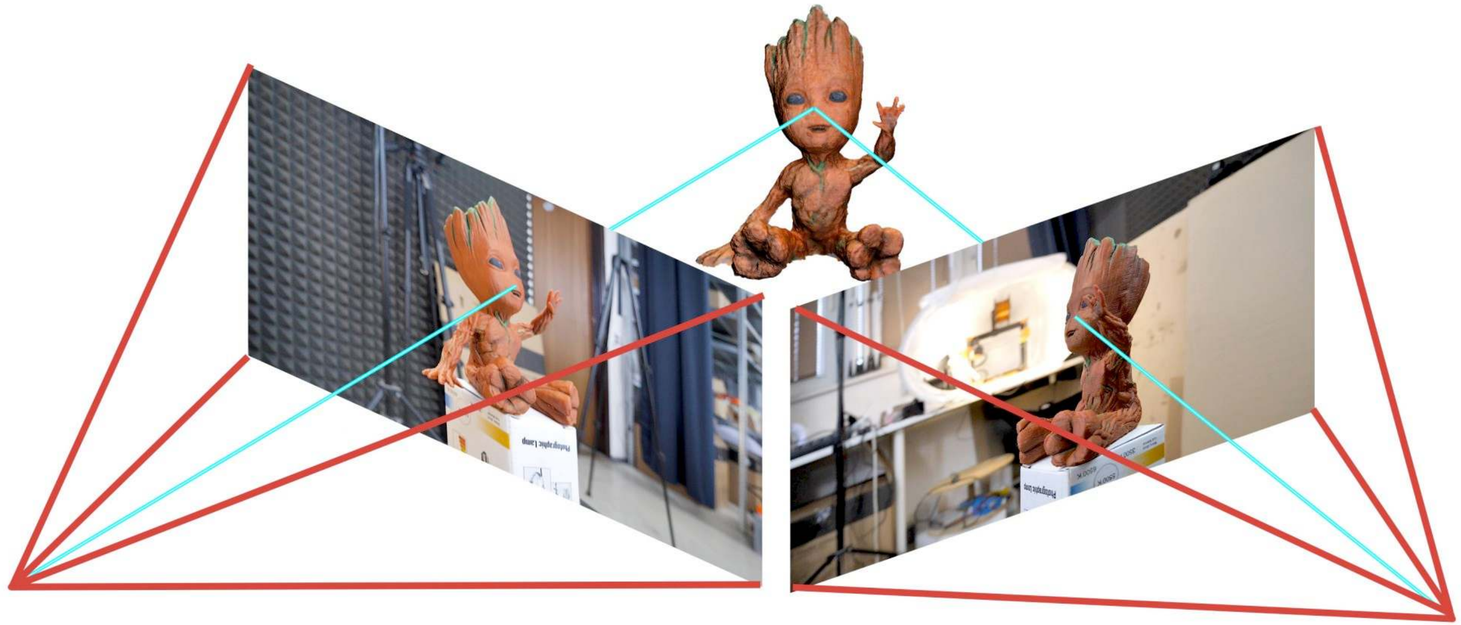
Inverse Problem — photogrammetry

- Want to learn a texture given 2D images of a 3D object
- Need to accumulate information from multiple (*possibly inconsistent*) views
- Want to produce a single *consistent* texture over the mesh



Inverse Problem — photogrammetry

- Want to learn a texture given 2D images of a 3D object
- Need to accumulate information from multiple (*possibly inconsistent*) views
- Want to produce a single *consistent* texture over the mesh
- How can we do this?



Texture Optimization

Optimization

Optimization

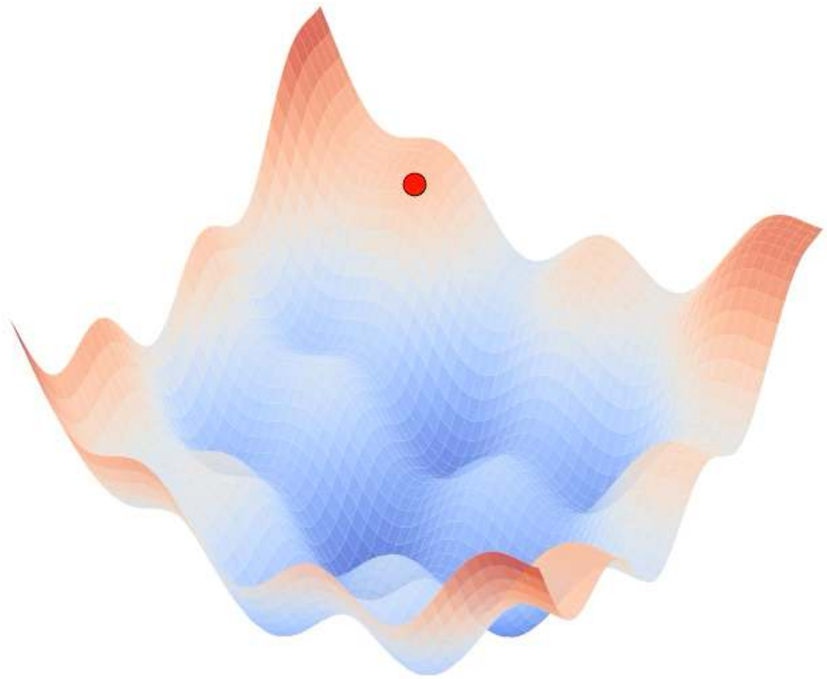
Assume we have some parameterized function and some target value

Optimization

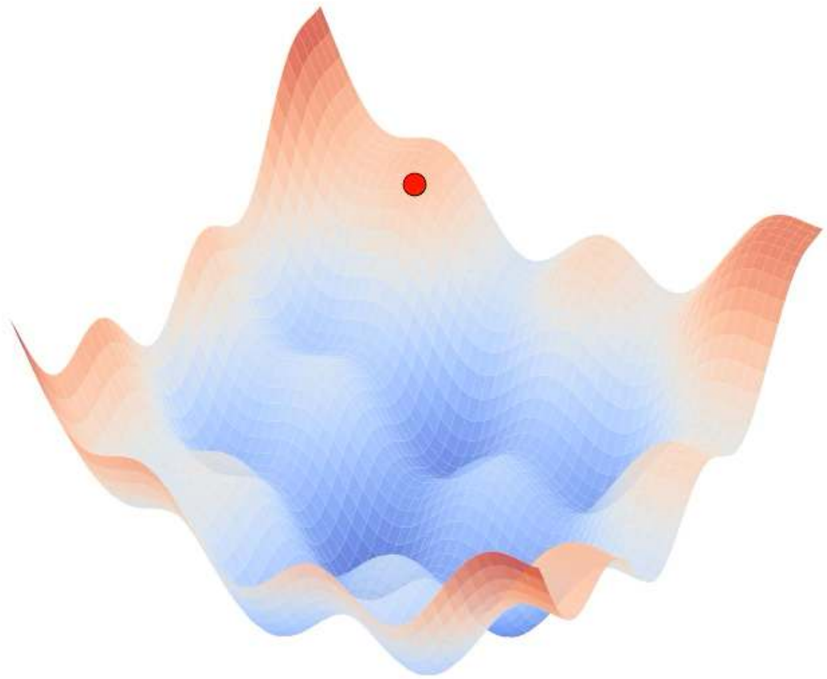
Assume we have some parameterized function and some target value

How can we update the parameters of our function to get the function value to match the target?

Gradient Descent



Gradient Descent



Gradient Descent

Gradient Descent

1. Determine the direction in which to change our parameters so that the function value moves closer to our target

Gradient Descent

1. Determine the direction in which to change our parameters so that the function value moves closer to our target

    - Compute some loss or measure of “closeness” between our current function evaluation and our target value

Gradient Descent

1. Determine the direction in which to change our parameters so that the function value moves closer to our target
    - Compute some loss or measure of “closeness” between our current function evaluation and our target value
    - Compute the gradient of this loss w.r.t. our parameter(s)

Gradient Descent

1. Determine the direction in which to change our parameters so that the function value moves closer to our target
    - Compute some loss or measure of “closeness” between our current function evaluation and our target value
    - Compute the gradient of this loss w.r.t. our parameter(s)
2. Take a small step in that direction

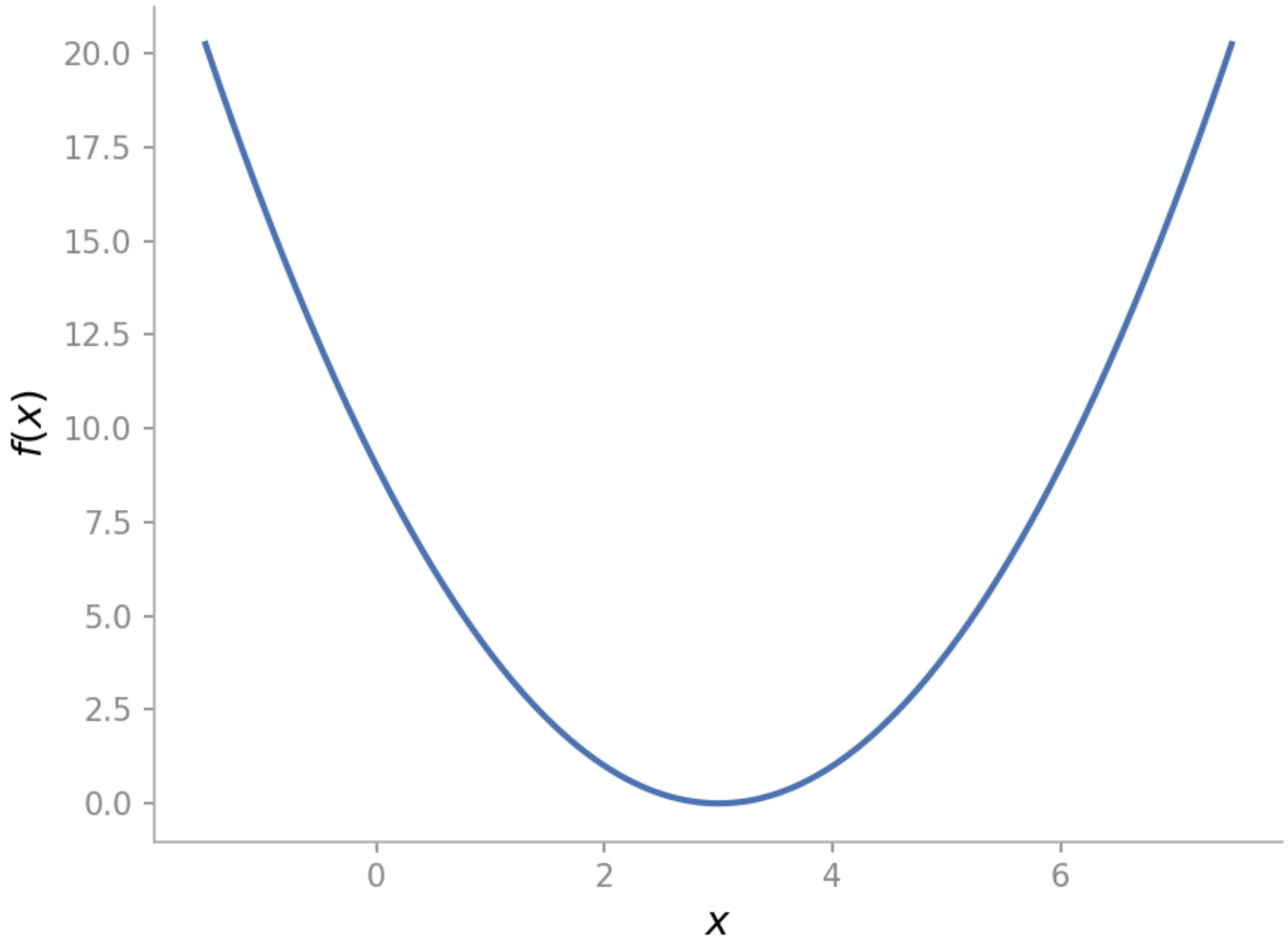
Gradient Descent

1. Determine the direction in which to change our parameters so that the function value moves closer to our target
    - Compute some loss or measure of “closeness” between our current function evaluation and our target value
    - Compute the gradient of this loss w.r.t. our parameter(s)
2. Take a small step in that direction
3. Repeat this process over and over until we get “close enough” to our target value

Example — setup

Example — setup

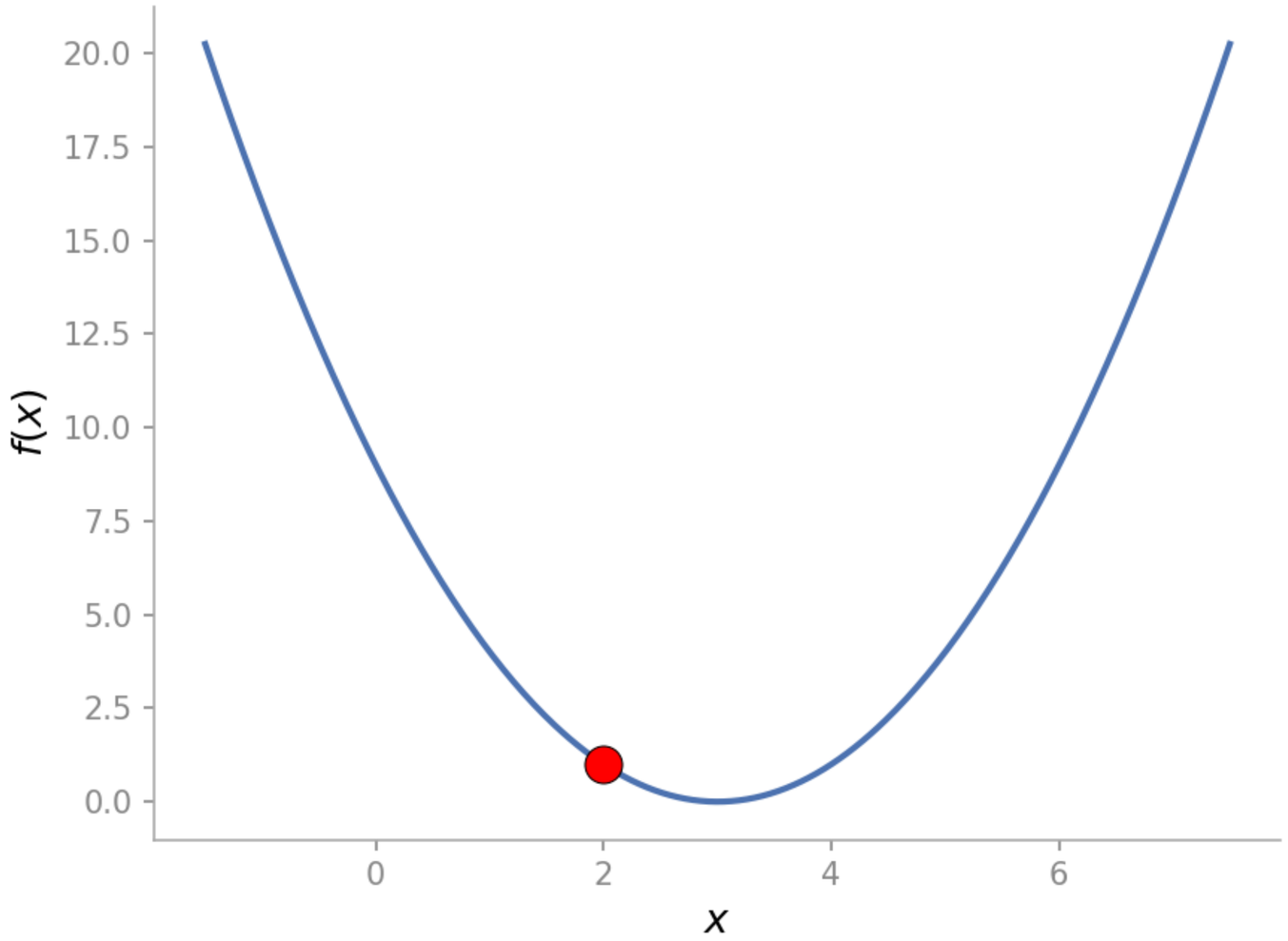
$$f(x) = x^2 - 6x + 9$$



Example — setup

$f(x) = x^2 - 6x + 9$

$x_0 = 2$

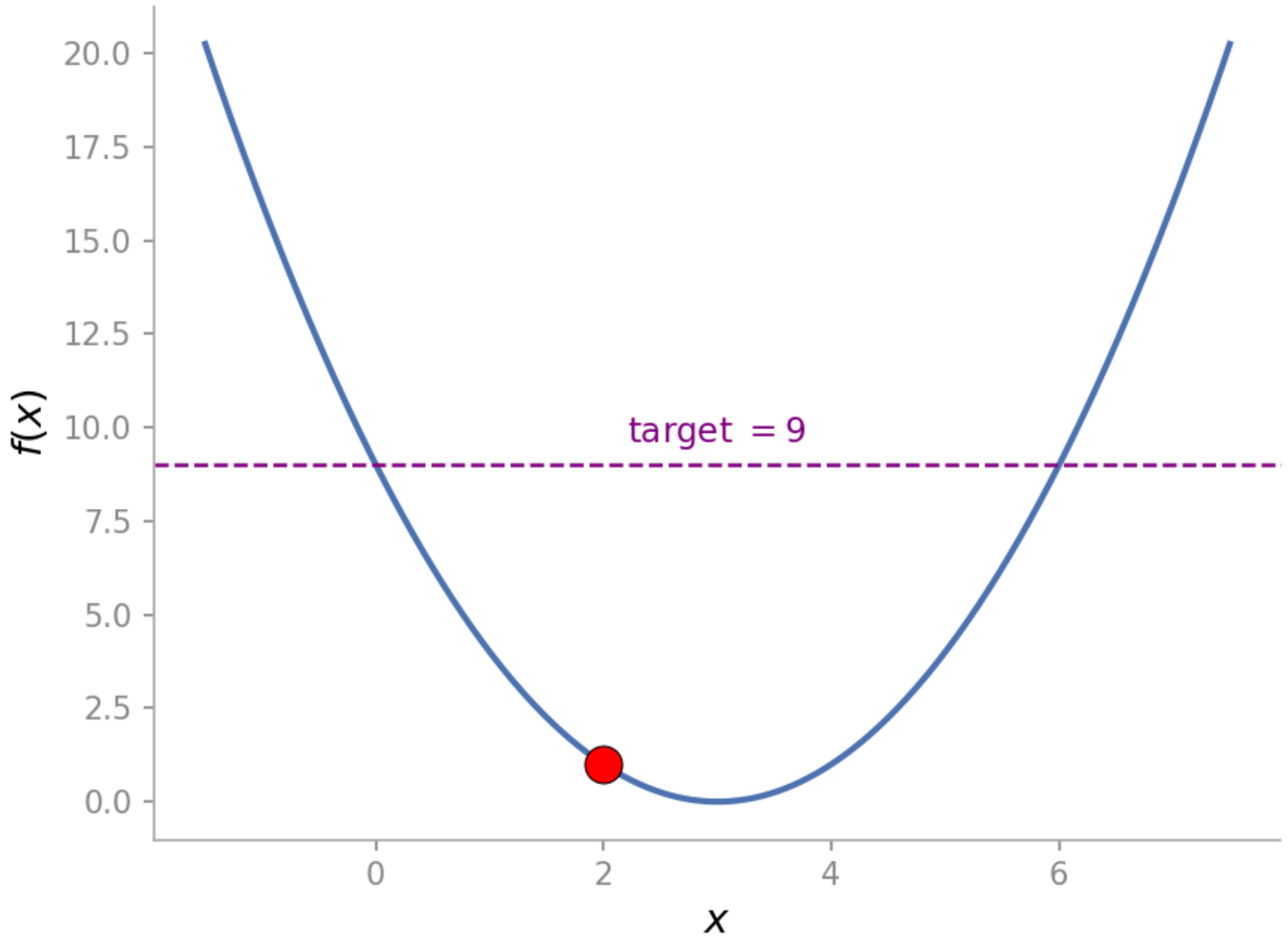


Example — setup

$f(x) = x^2 - 6x + 9$

$x_0 = 2$

Target $= 9$



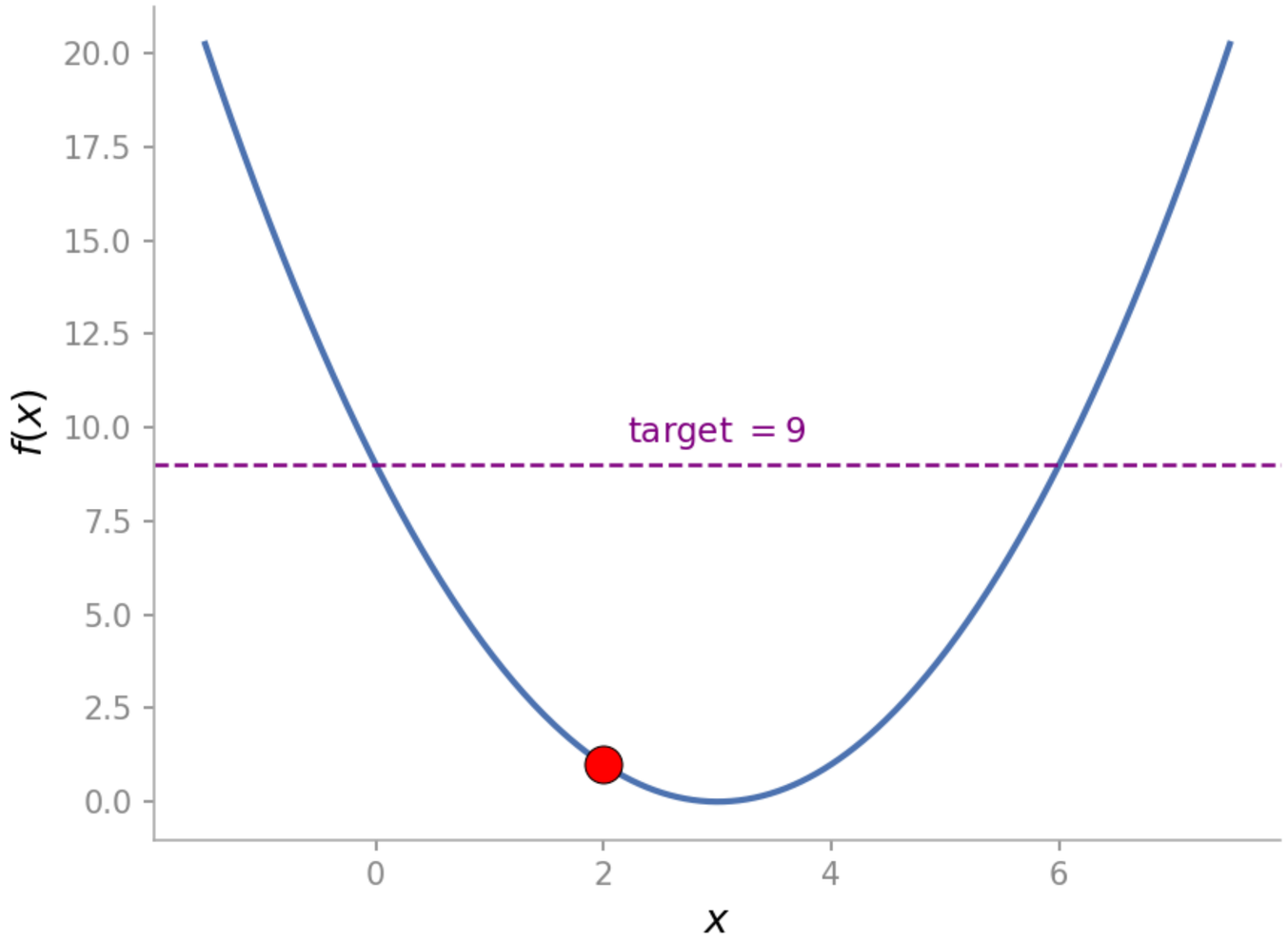
Example — setup

$f(x) = x^2 - 6x + 9$

$x_0 = 2$

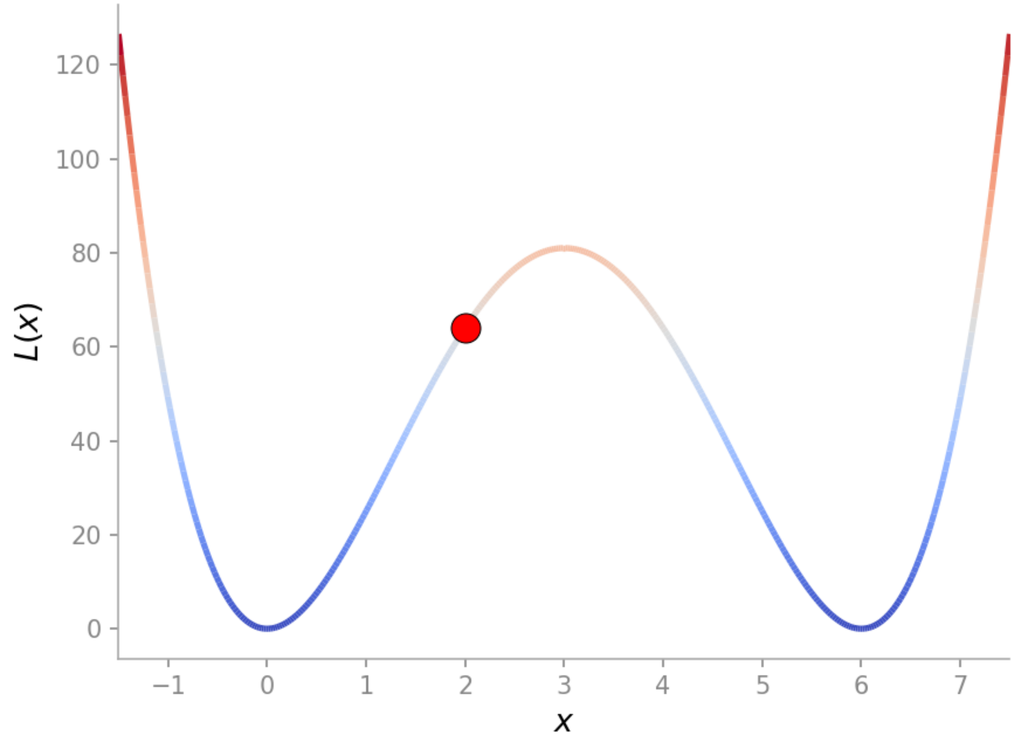
Target $= 9$

$L = (f(x) - \text{Target})^2$



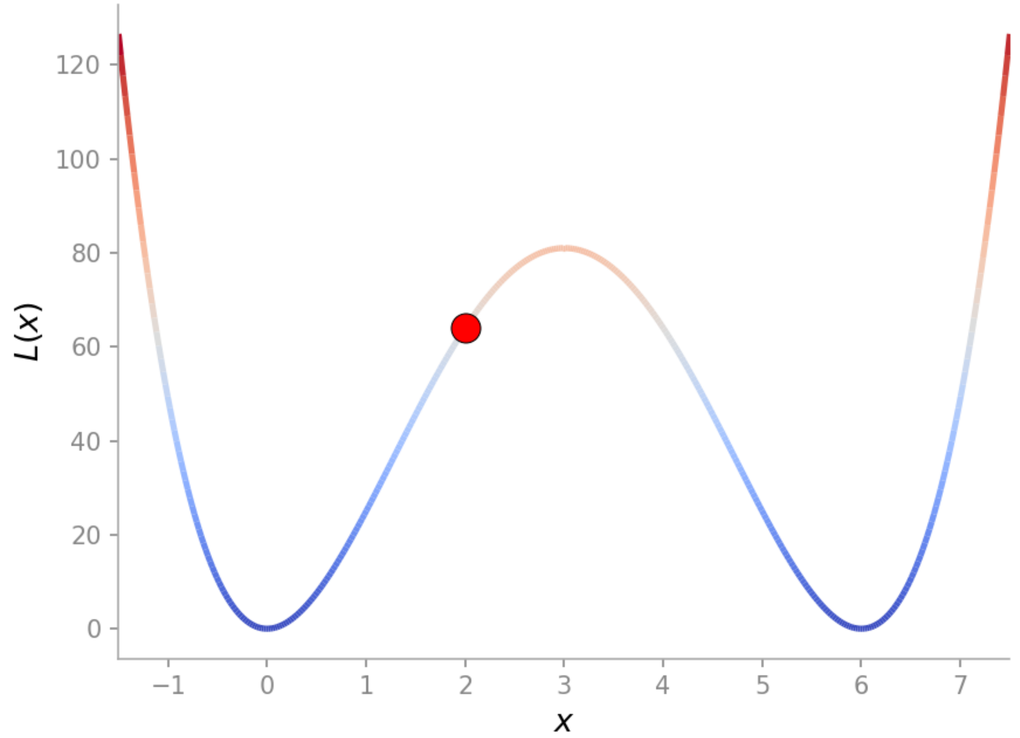
Example — gradient calculation

Example — gradient calculation



Example — gradient calculation

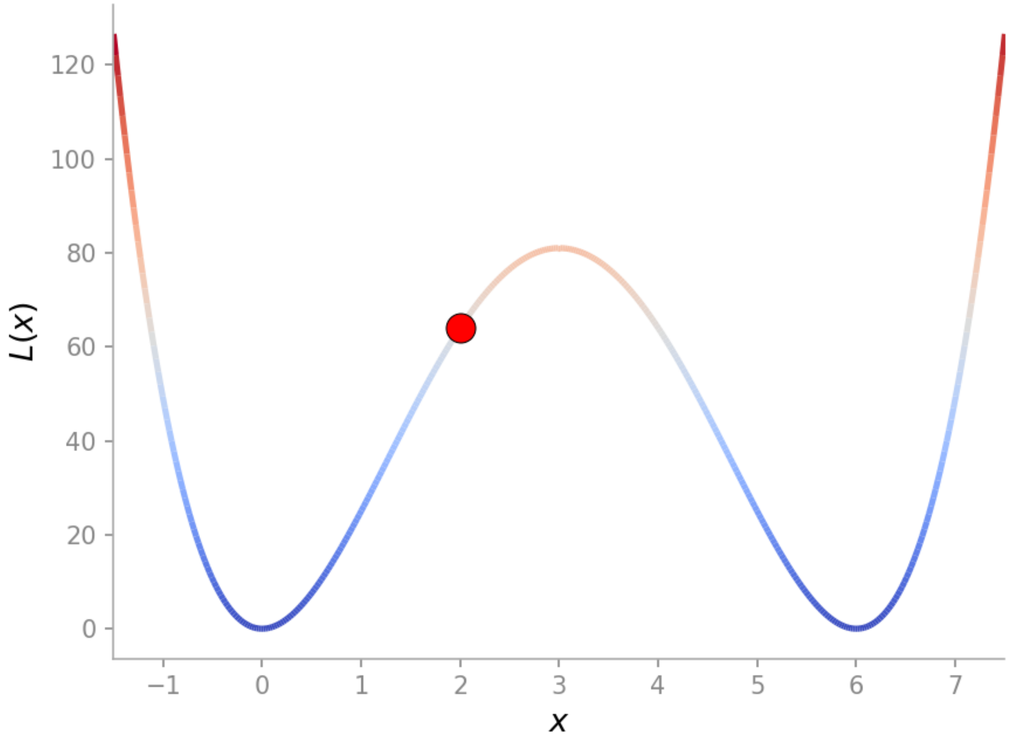
Start by taking the derivative:



Example — gradient calculation

Start by taking the derivative:

$$\frac{dL}{dx} = \frac{dL}{df}\frac{df}{dx} = 2(f(x) - 9)(2x - 6)$$

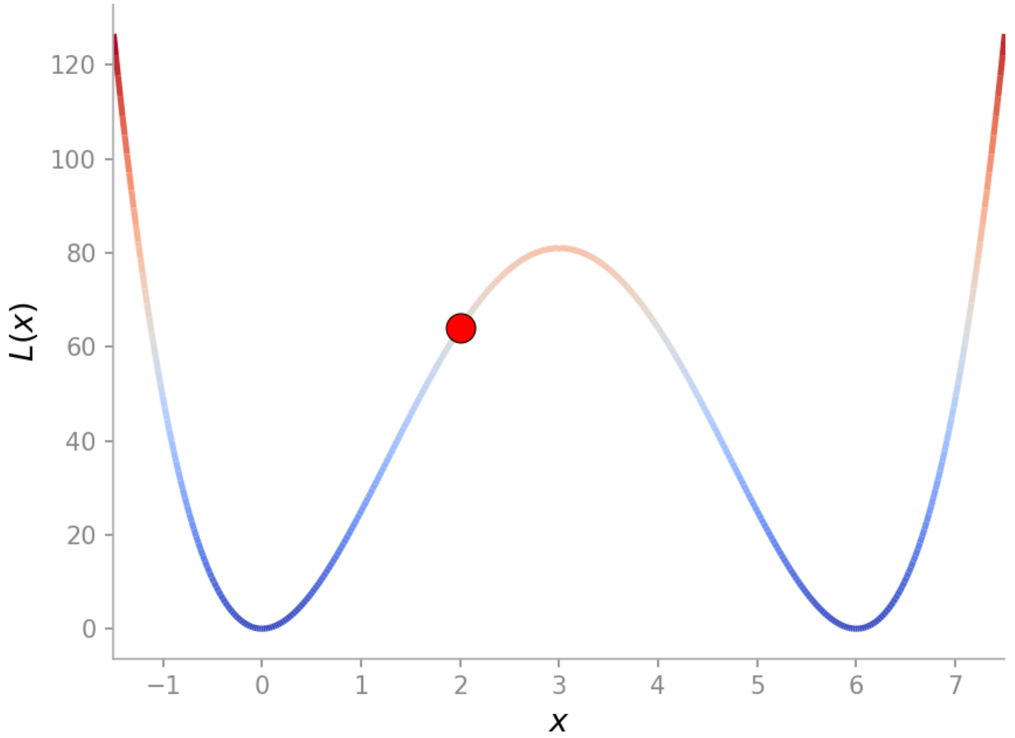


Example — gradient calculation

Start by taking the derivative:

$$\frac{dL}{dx} = \frac{dL}{df}\frac{df}{dx} = 2(f(x) - 9)(2x - 6)$$

Evaluate at the initial parameter value:



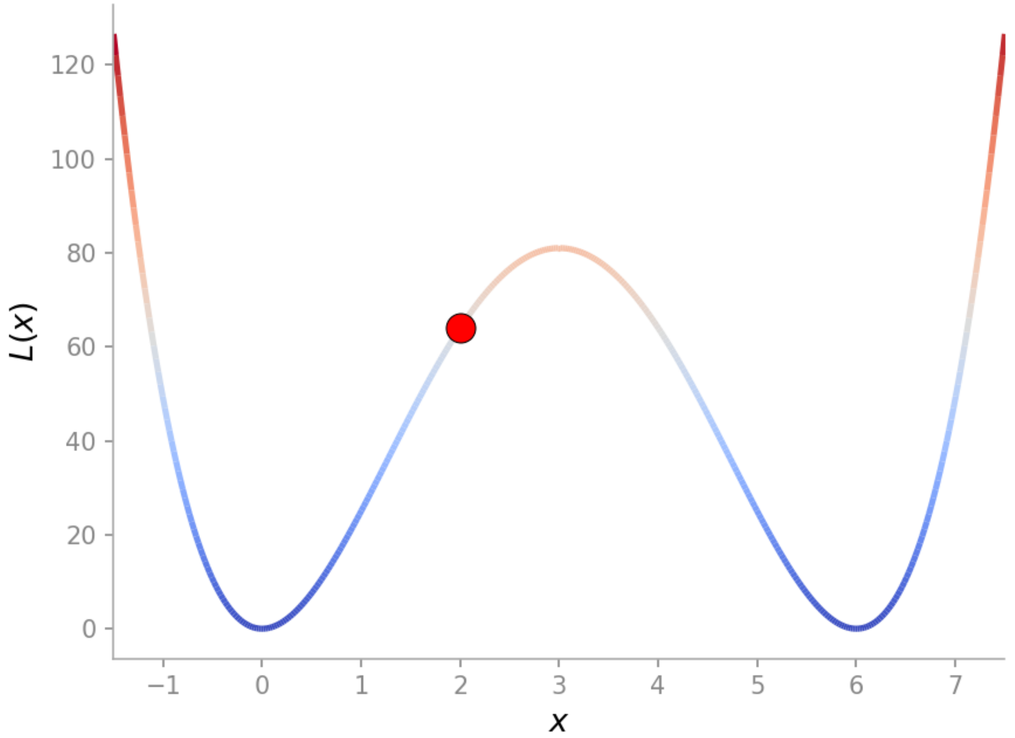
Example — gradient calculation

Start by taking the derivative:

$$\frac{dL}{dx} = \frac{dL}{df}\frac{df}{dx} = 2(f(x) - 9)(2x - 6)$$

Evaluate at the initial parameter value:

$$f(2) = 2^2 - (6 \cdot 2) + 9 = 1$$



Example — gradient calculation

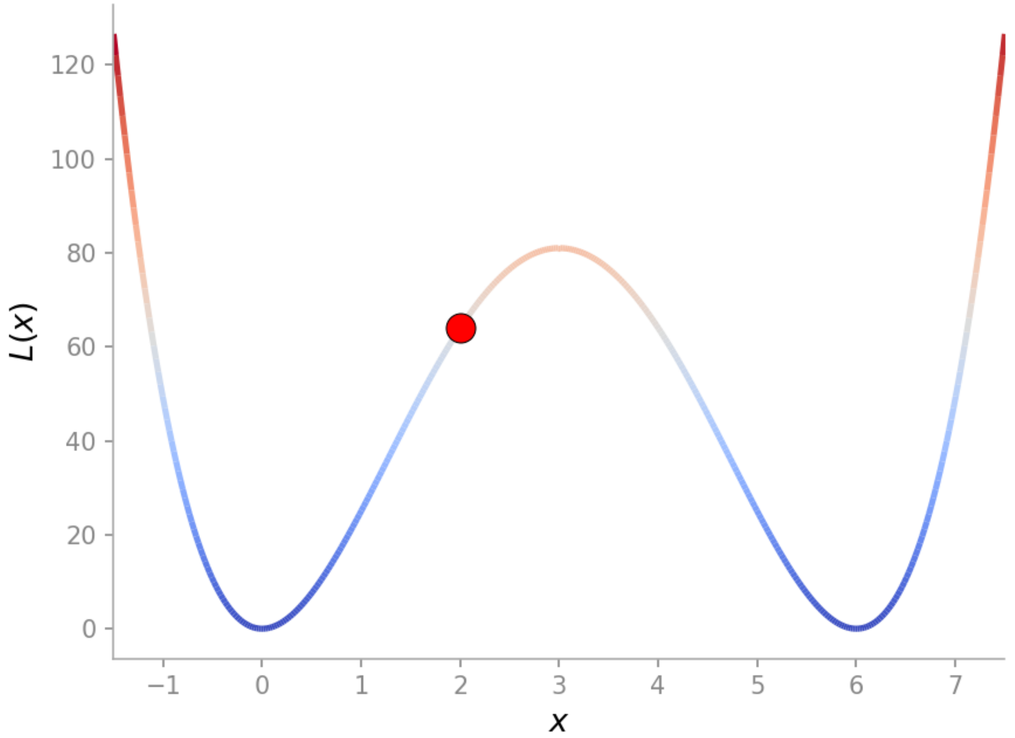
Start by taking the derivative:

$$\frac{dL}{dx} = \frac{dL}{df}\frac{df}{dx} = 2(f(x) - 9)(2x - 6)$$

Evaluate at the initial parameter value:

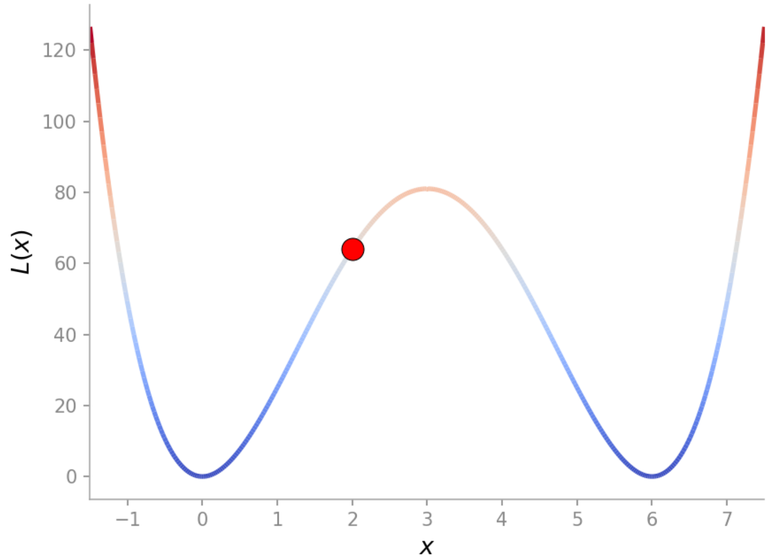
$$f(2) = 2^2 - (6 \cdot 2) + 9 = 1$$

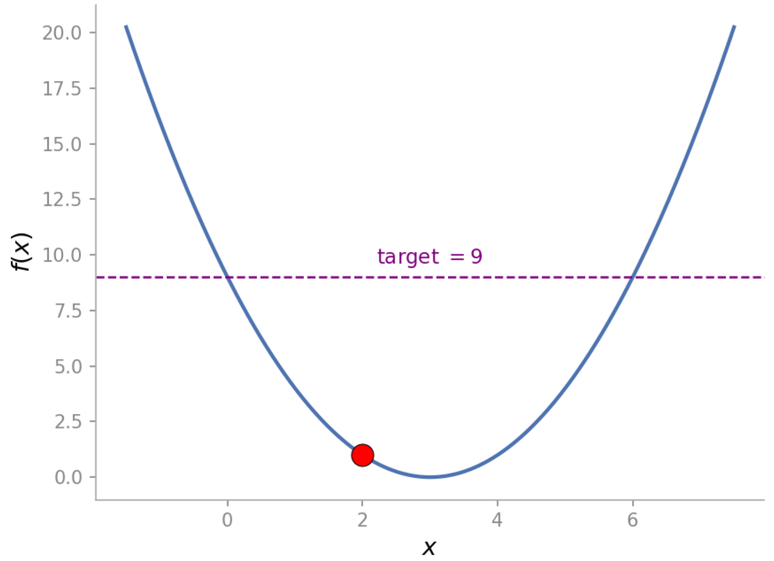
$$\left.\frac{dL}{dx}\right|_{x=2} = 2(1 - 9)(2 \cdot 2 - 6) = (-16)(-2) = 32$$



Example — update step

Now we can use this gradient along with an update rule to update our parameter!

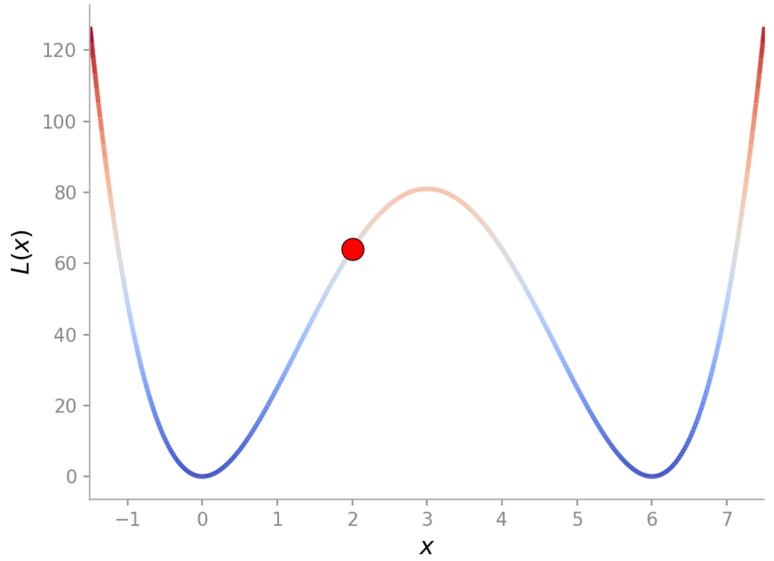


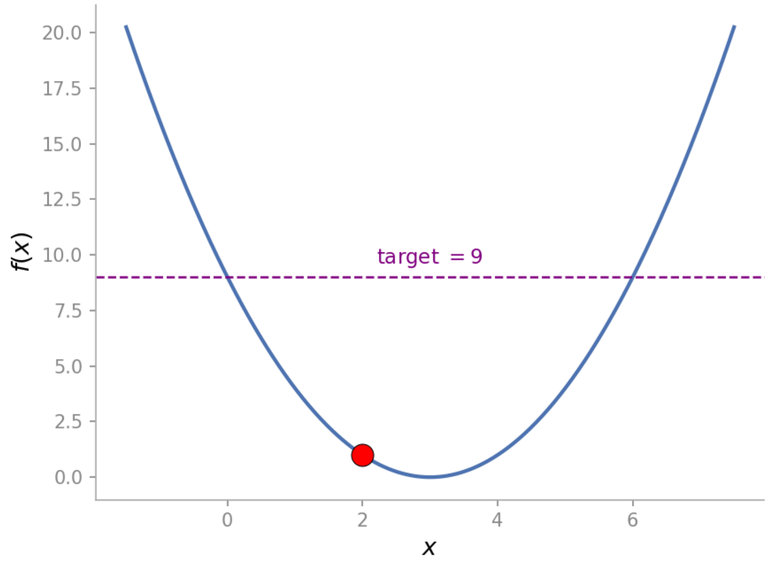


Example — update step

Now we can use this gradient along with an update rule to update our parameter!

$$x_{n+1} = x_n - \eta \frac{dL}{dx}$$



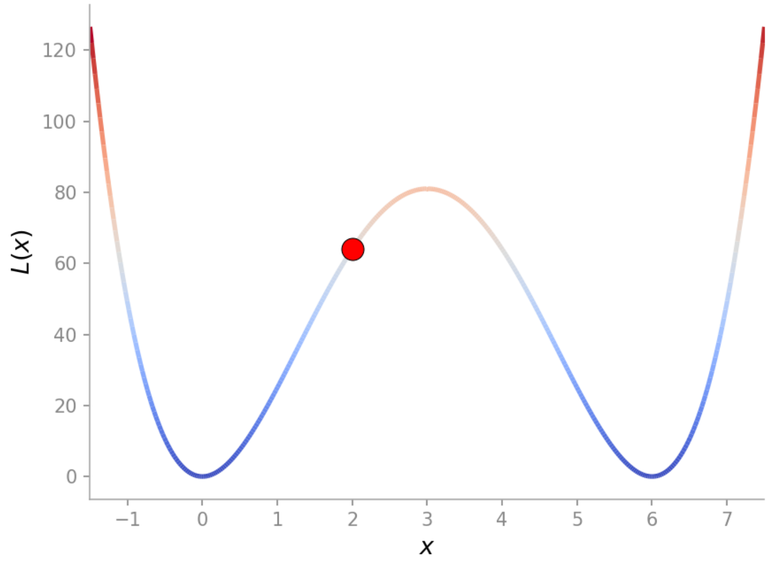


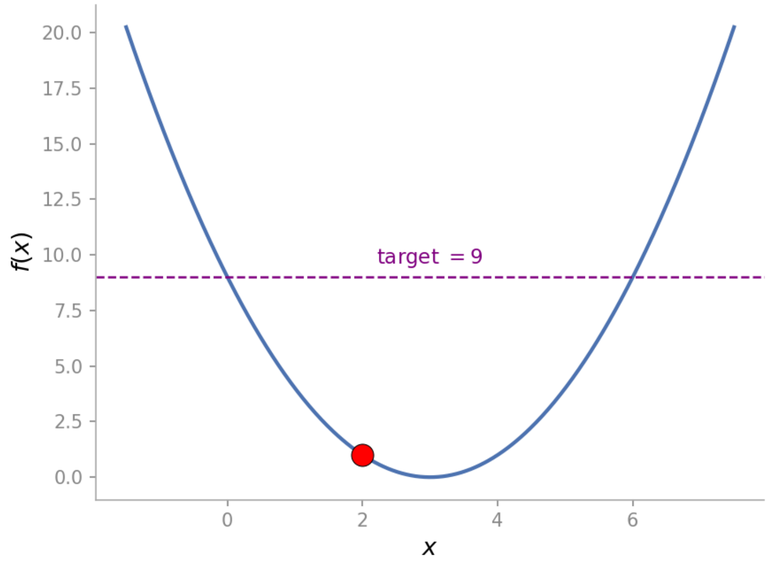
Example — update step

Now we can use this gradient along with an update rule to update our parameter!

$$x_{n+1} = x_n - \eta \frac{dL}{dx}$$

$$x_1 = x_0 - \eta \frac{dL}{dx} = 2 - 0.01 \cdot 32 = 1.68$$





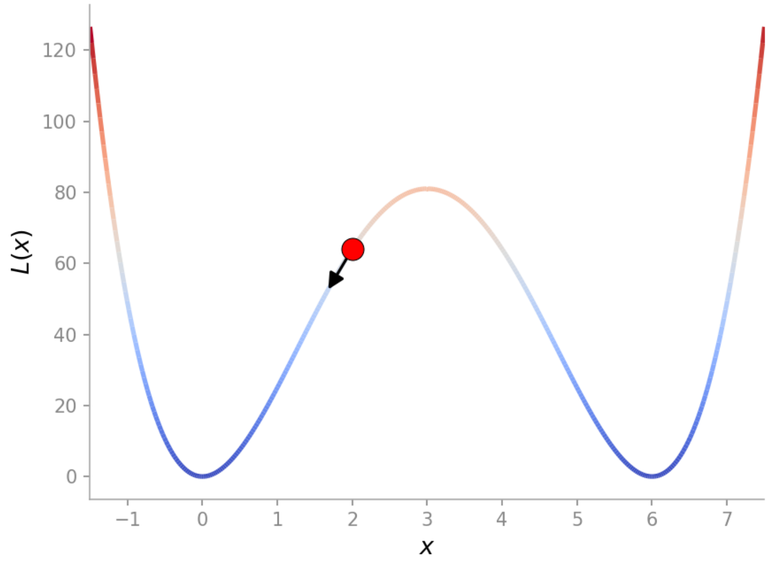
Example — update step

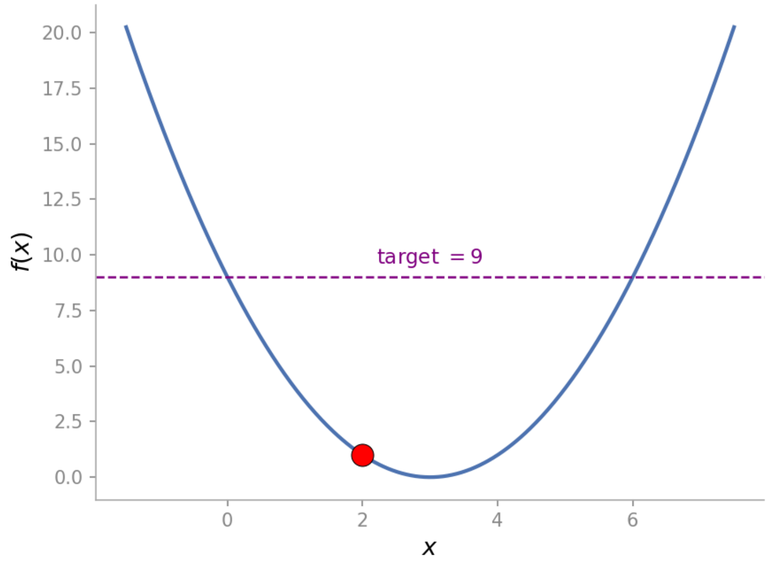
Now we can use this gradient along with an update rule to update our parameter!

$$x_{n+1} = x_n - \eta \frac{dL}{dx}$$

$$x_1 = x_0 - \eta \frac{dL}{dx} = 2 - 0.01 \cdot 32 = 1.68$$

$x_1 = 1.68$ is closer to a solution at $x = 0$





Example — update step

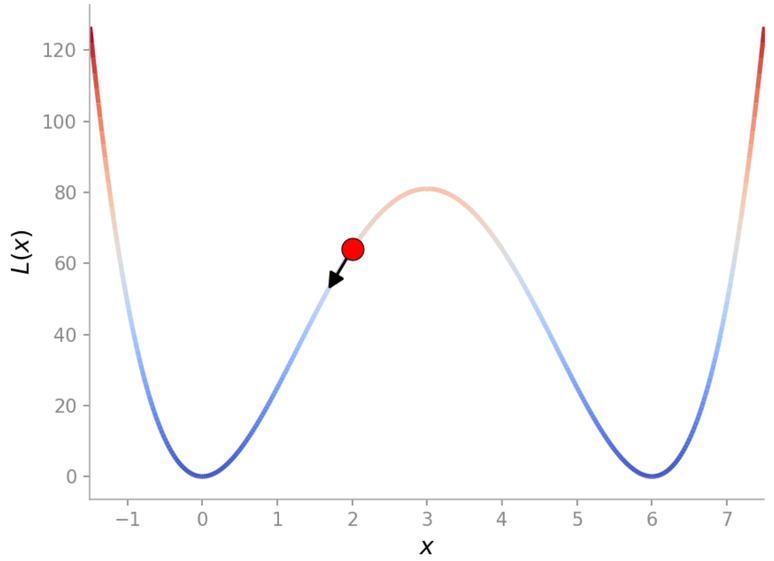
Now we can use this gradient along with an update rule to update our parameter!

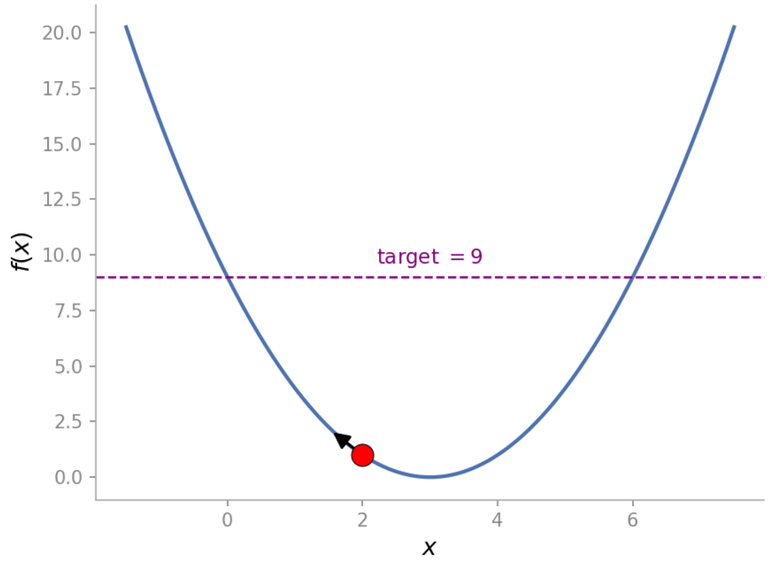
$$x_{n+1} = x_n - \eta \frac{dL}{dx}$$

$$x_1 = x_0 - \eta \frac{dL}{dx} = 2 - 0.01 \cdot 32 = 1.68$$

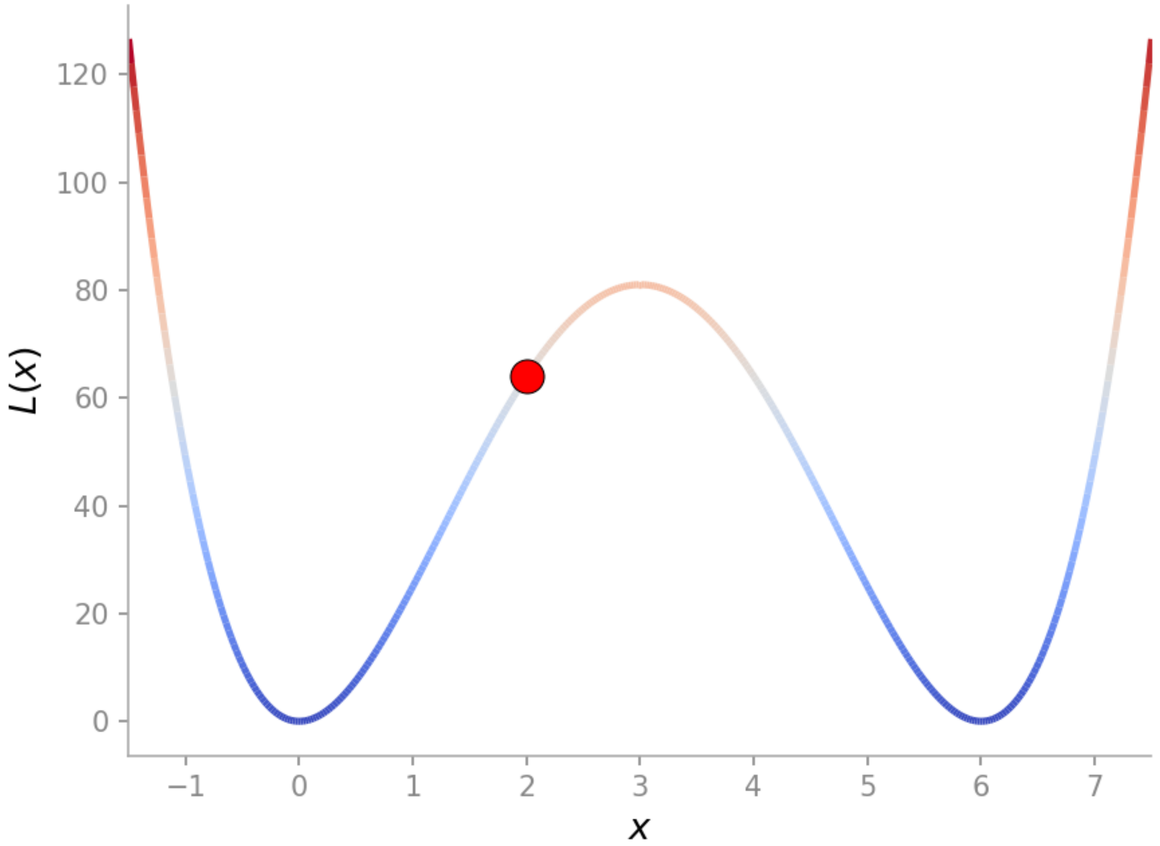
$x_1 = 1.68$ is closer to a solution at $x = 0$

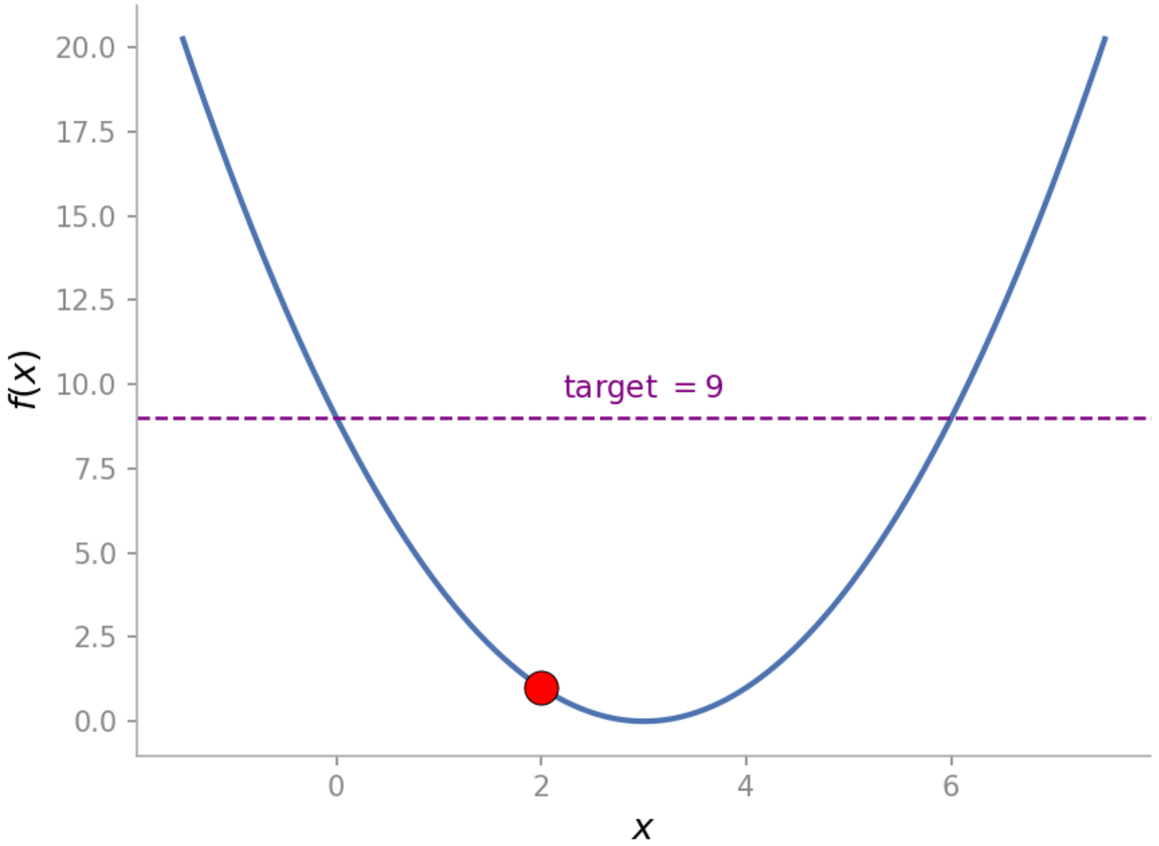
$f(x_1) = 1.7424$ is closer to our target 9



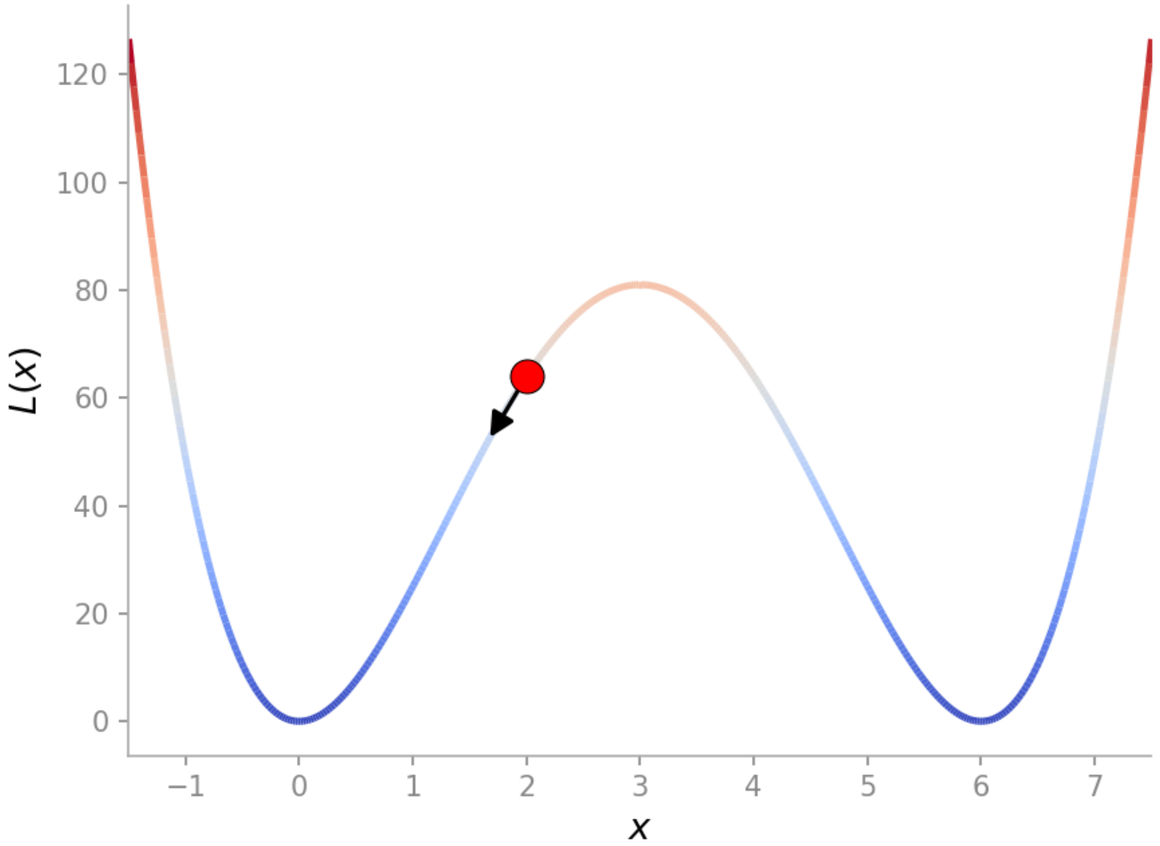


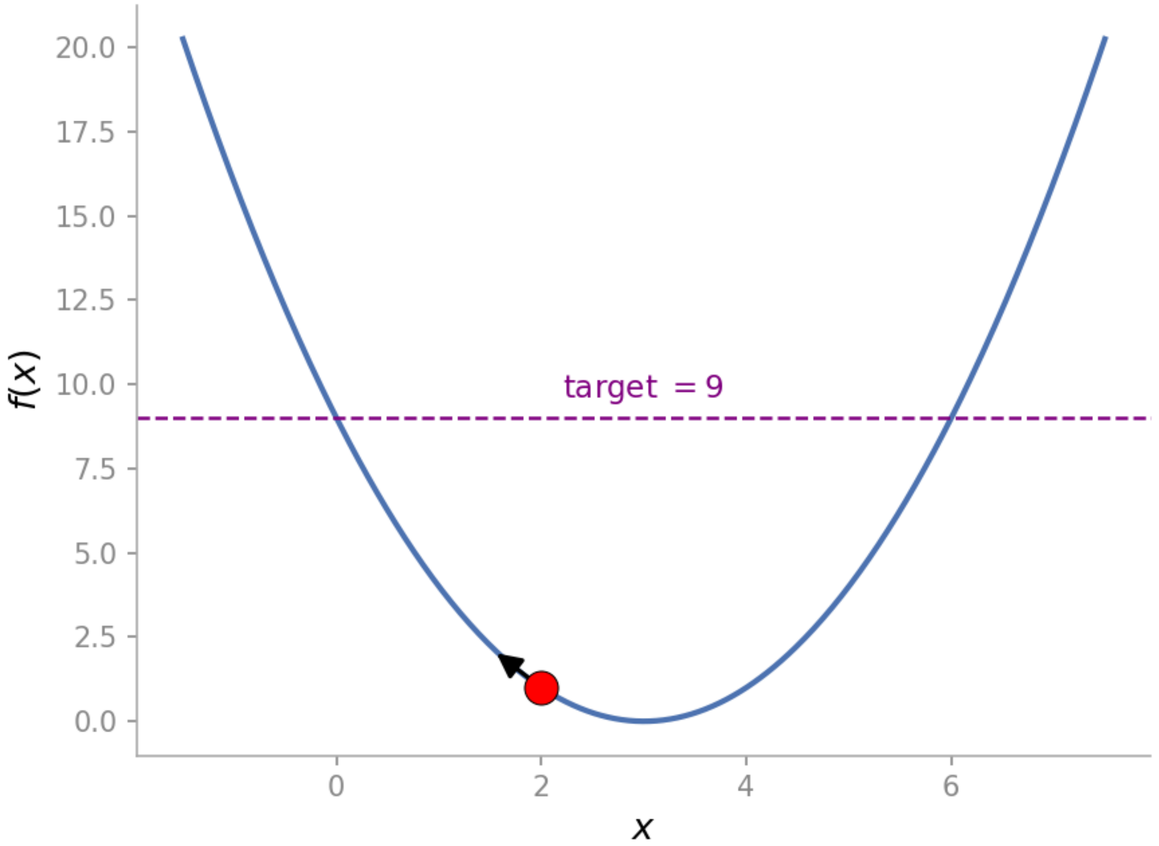
Example — visualization



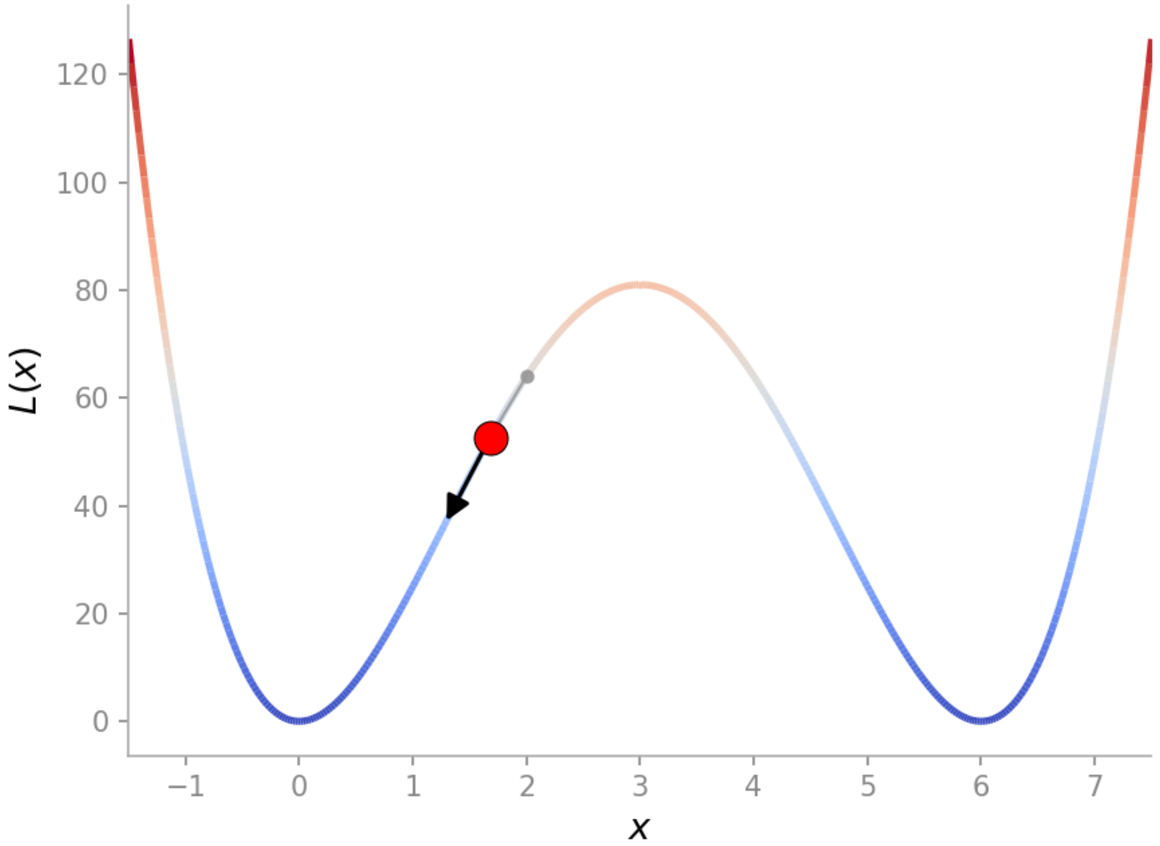


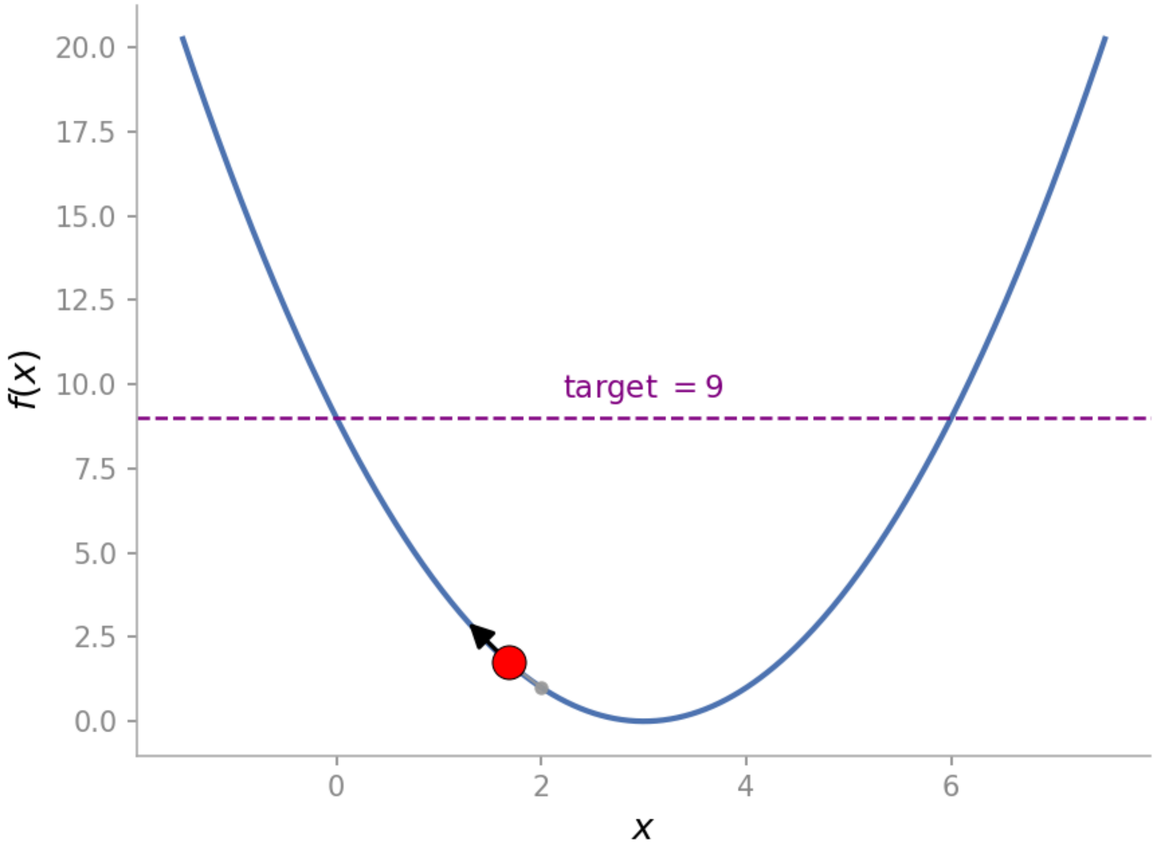
Example — visualization



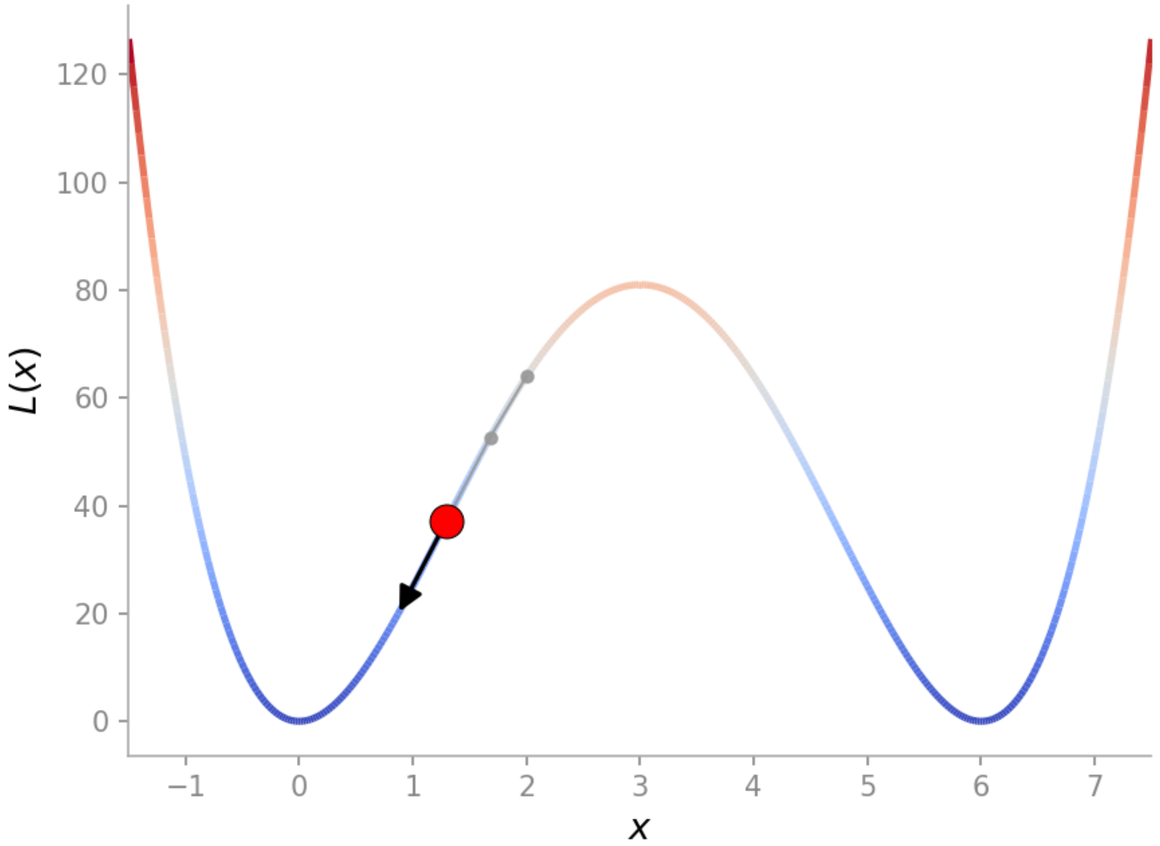


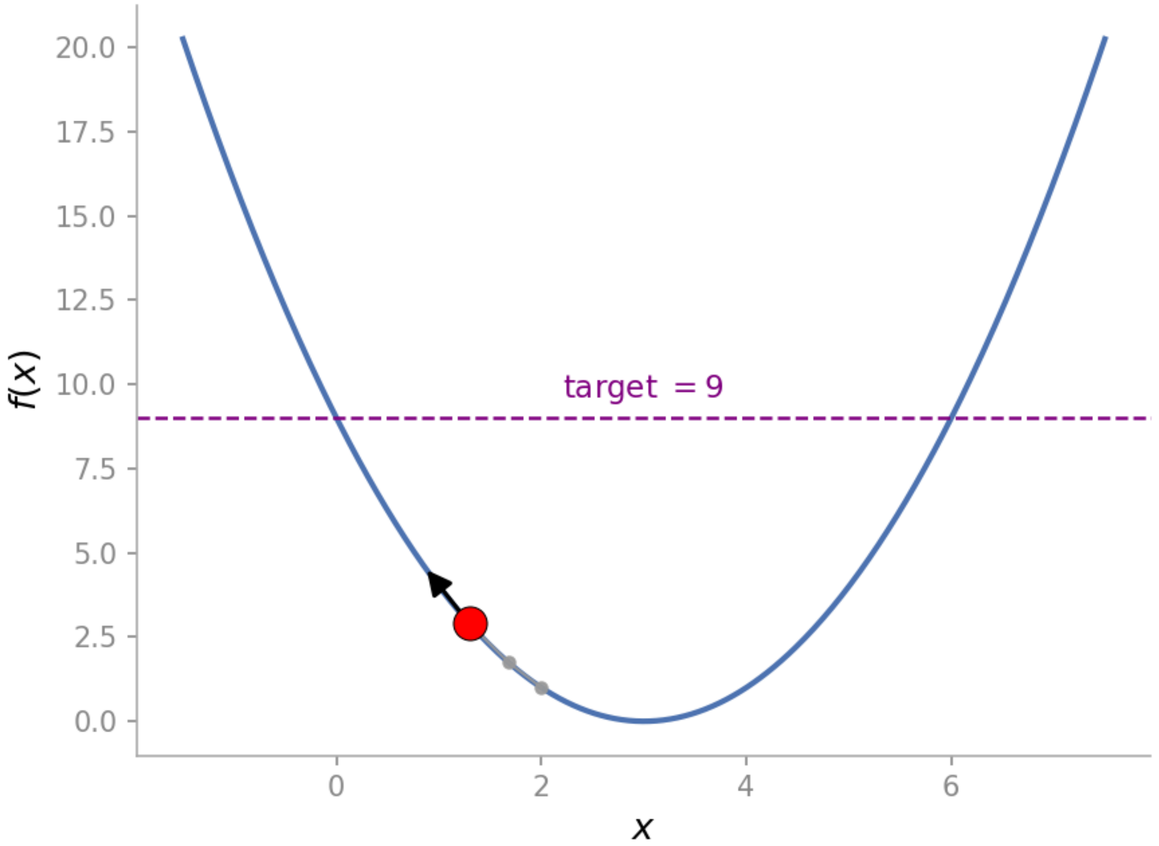
Example — visualization



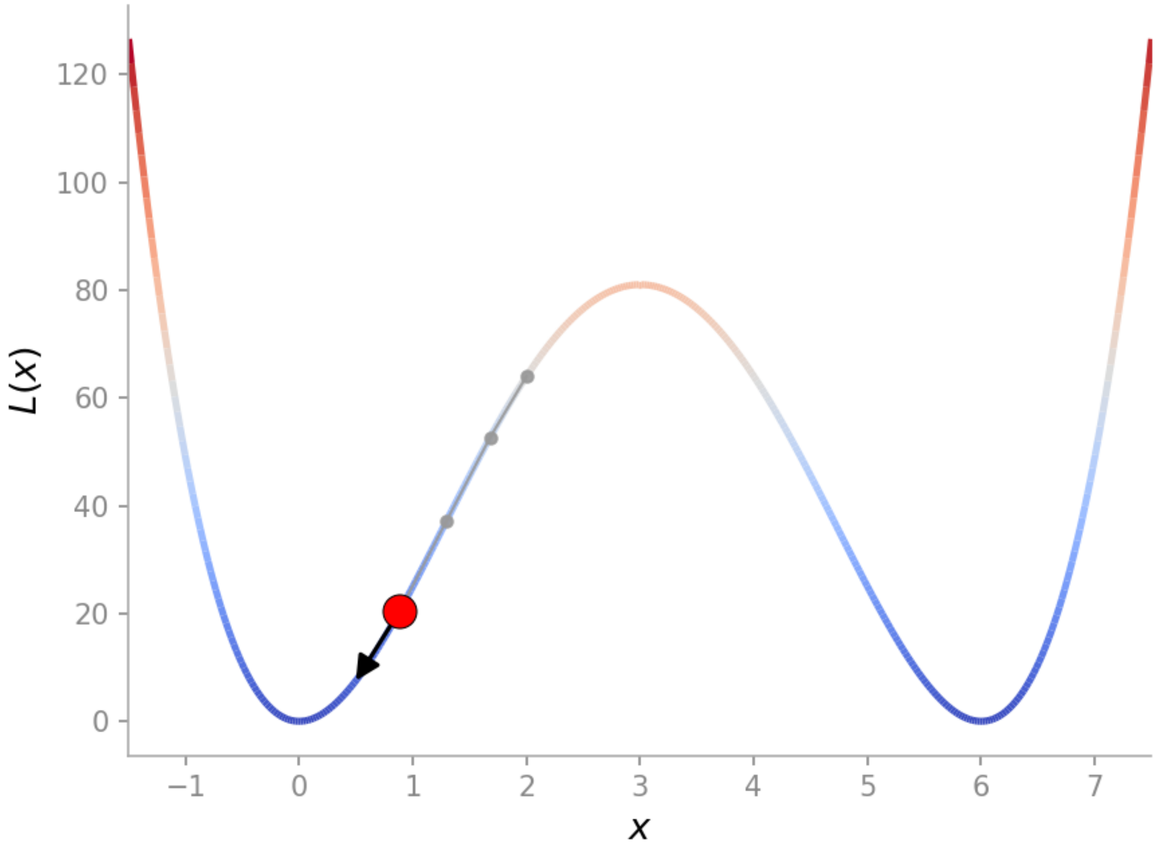


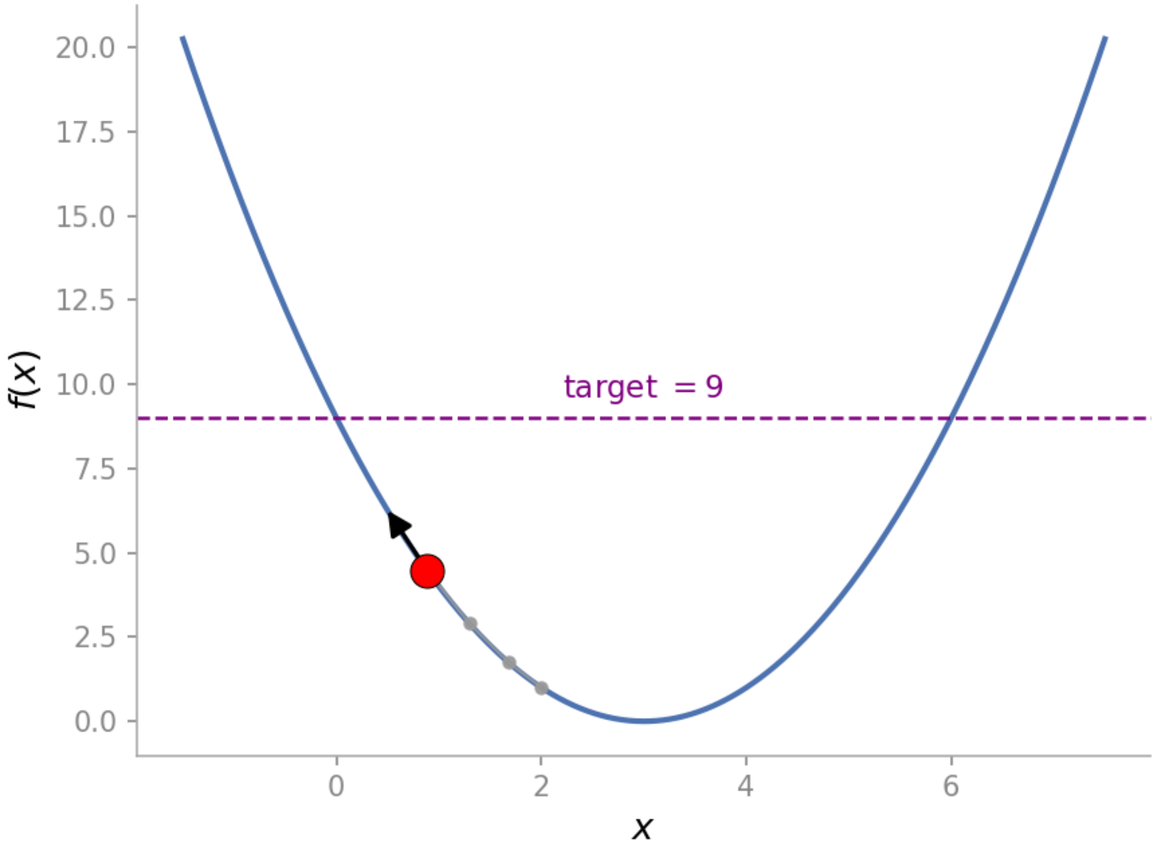
Example — visualization



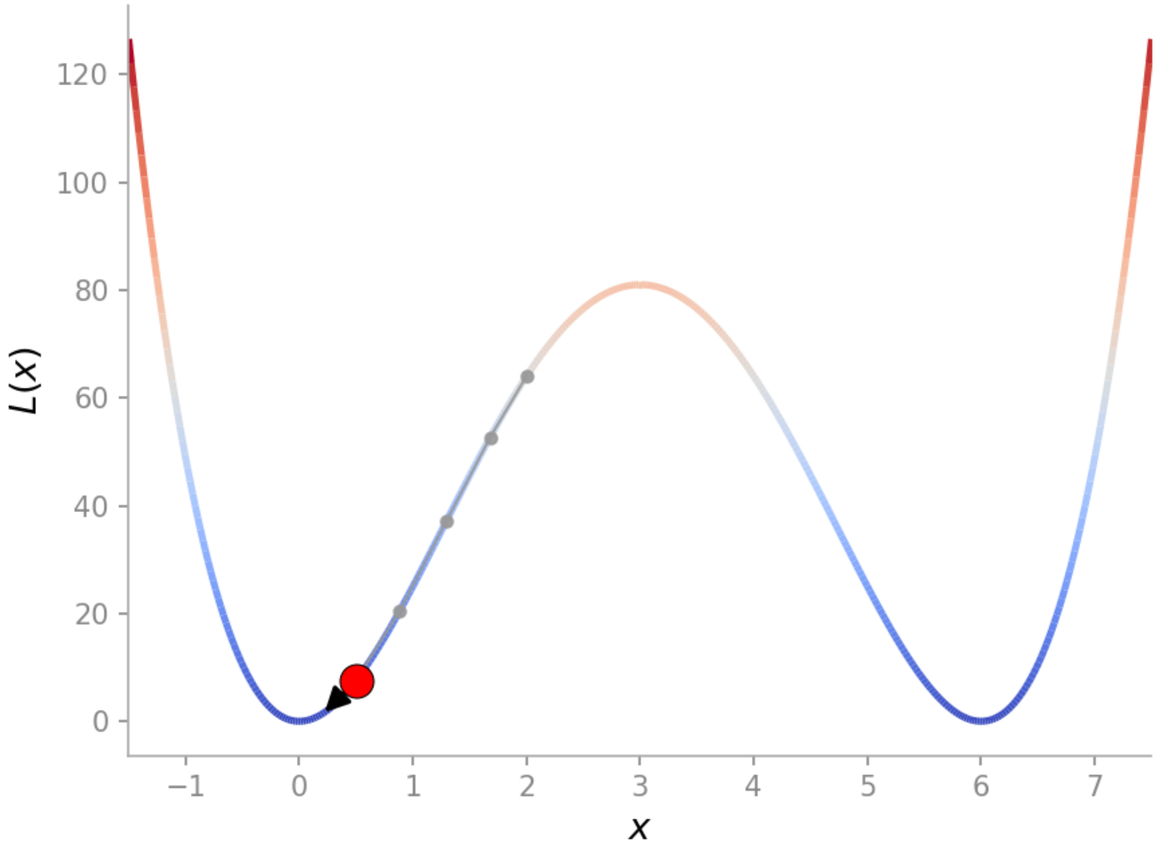


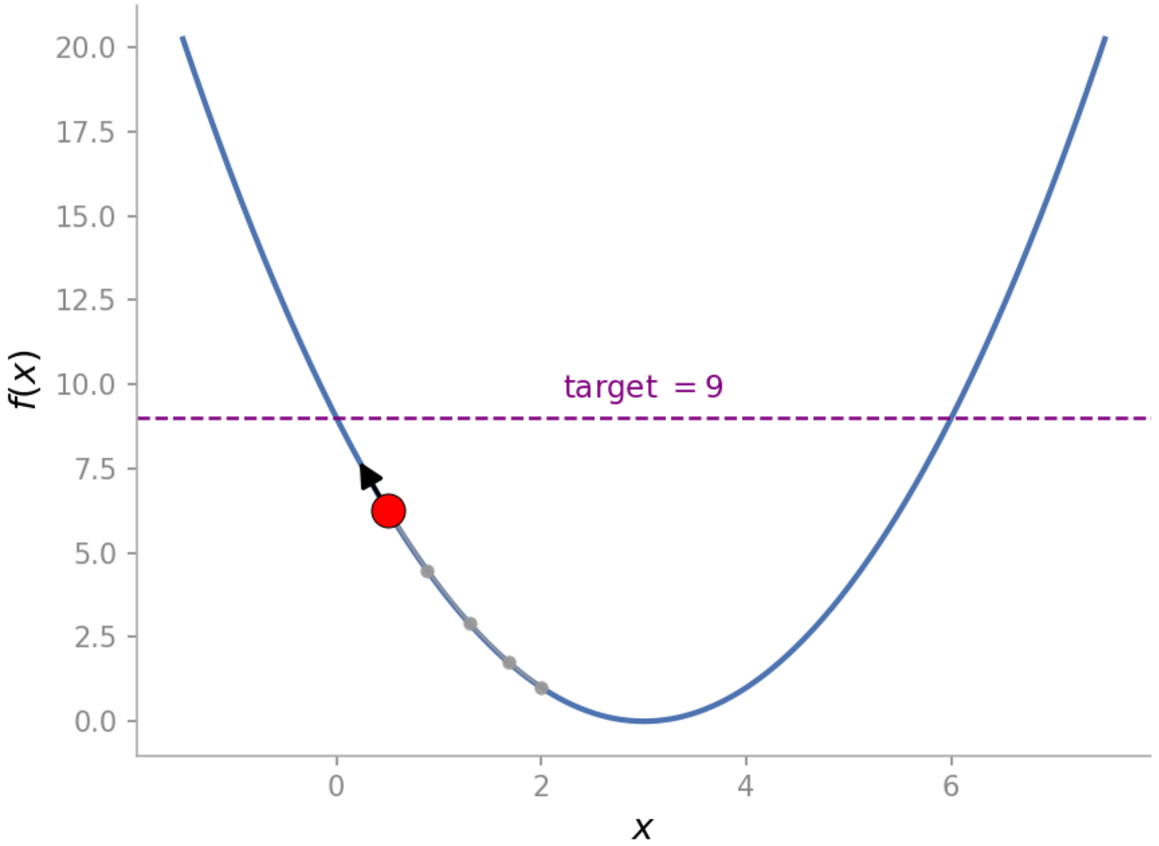
Example — visualization



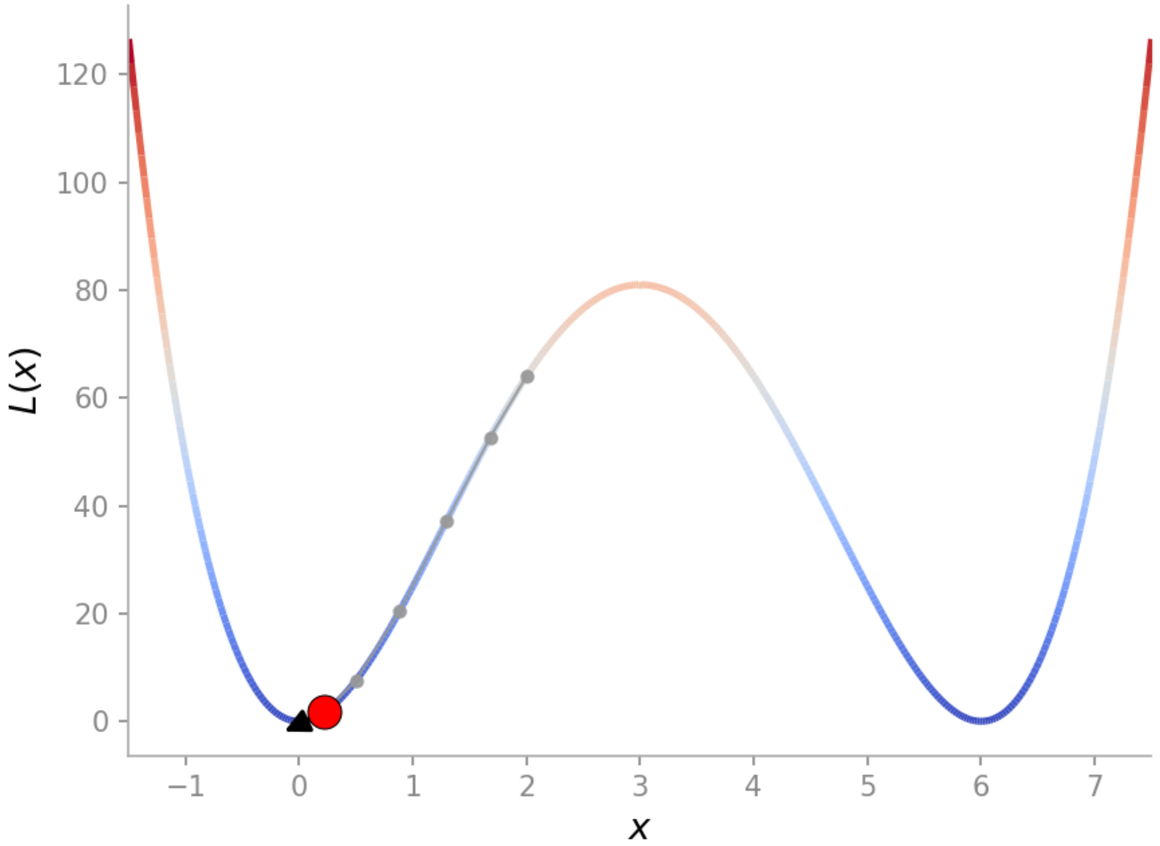


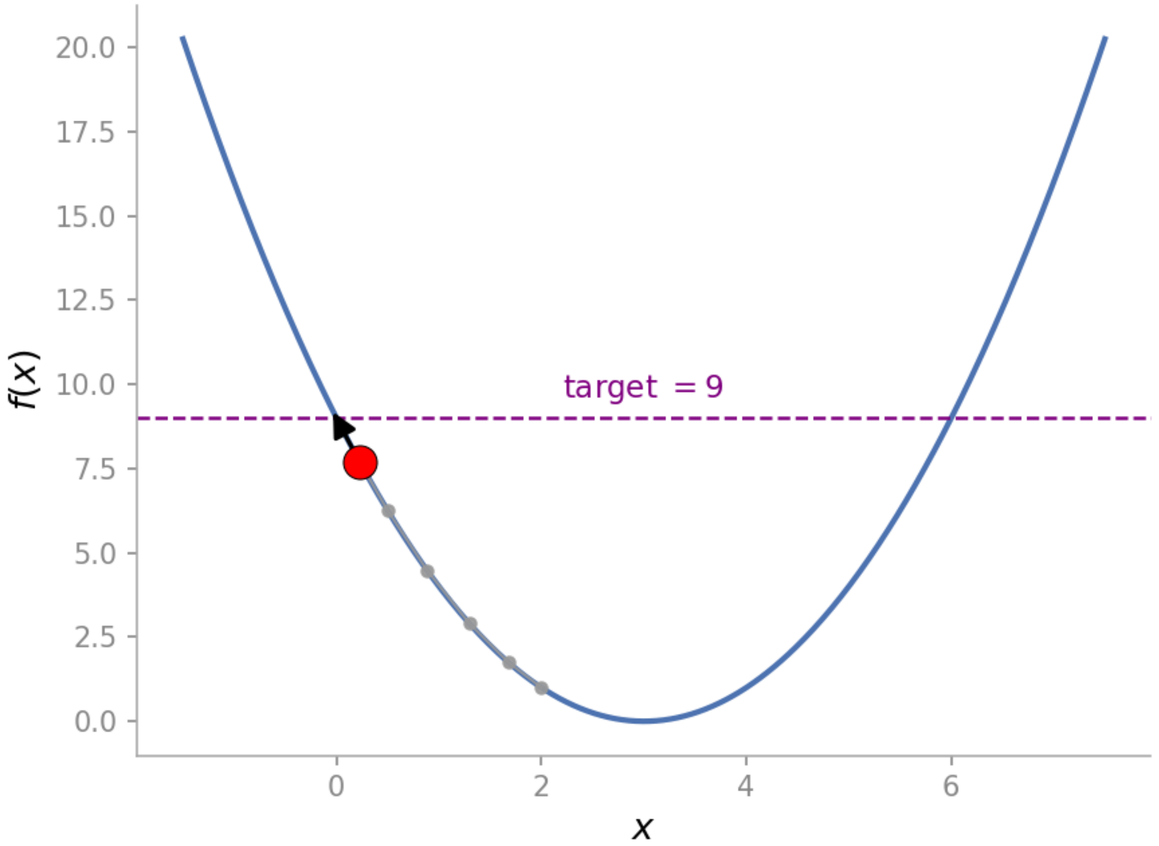
Example — visualization



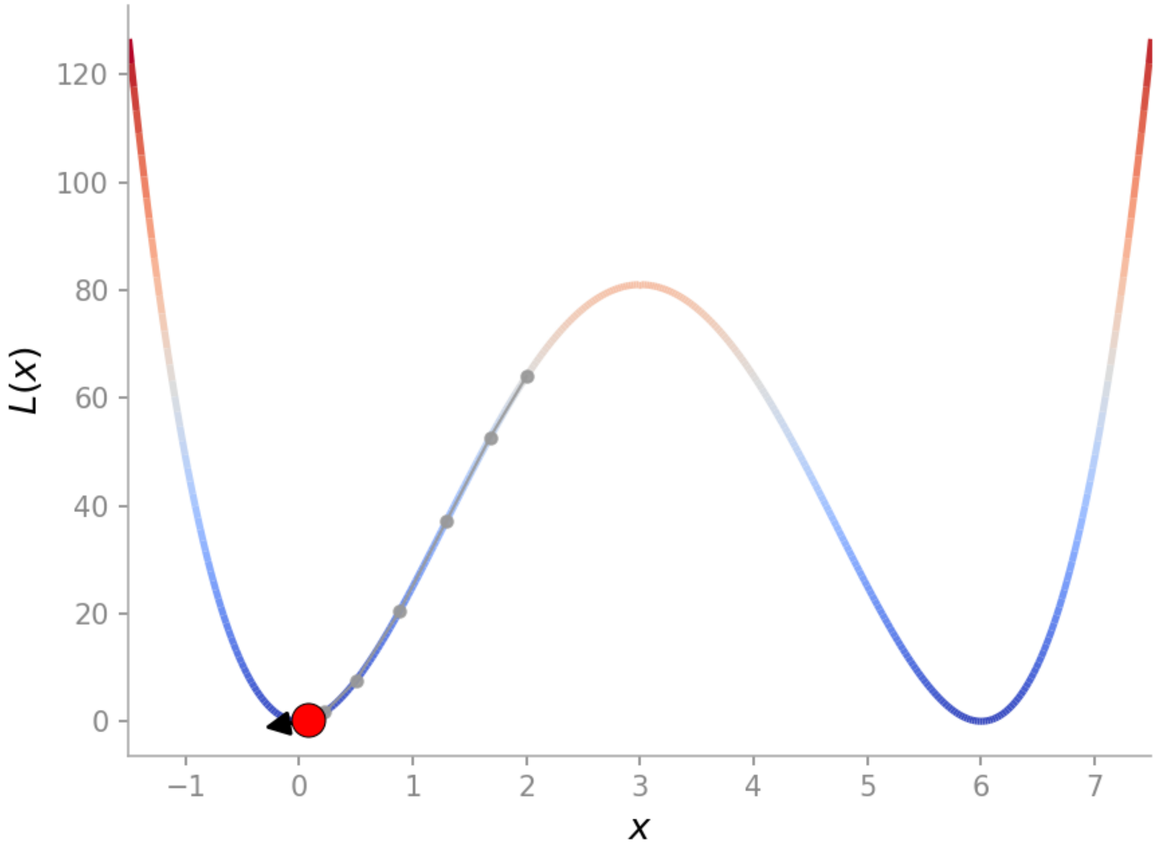


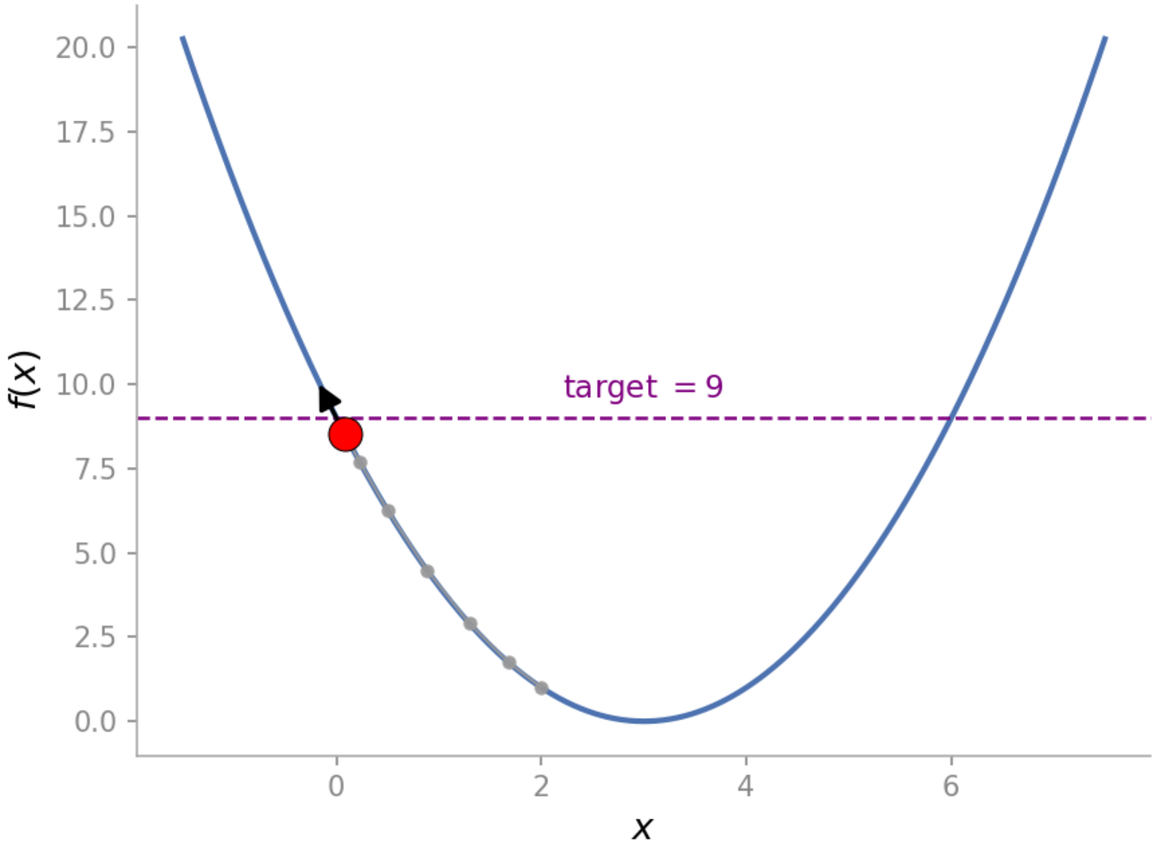
Example — visualization



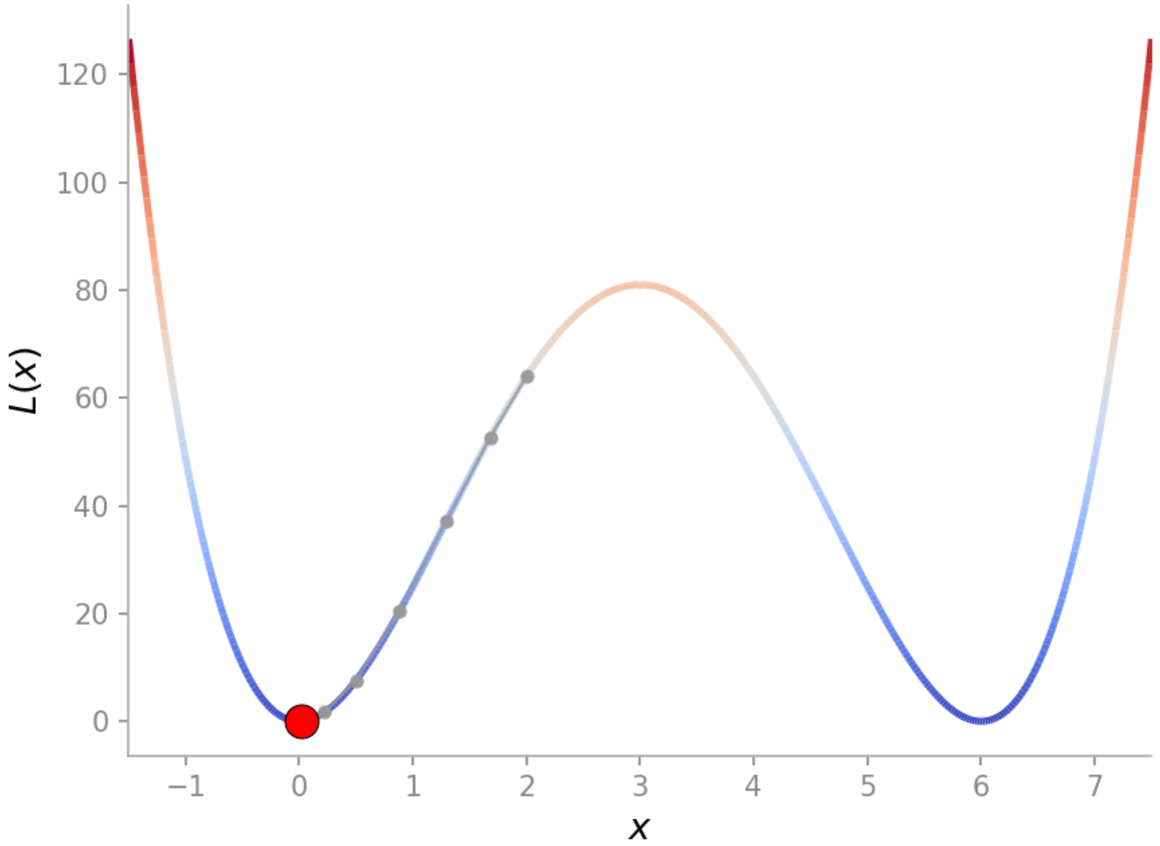


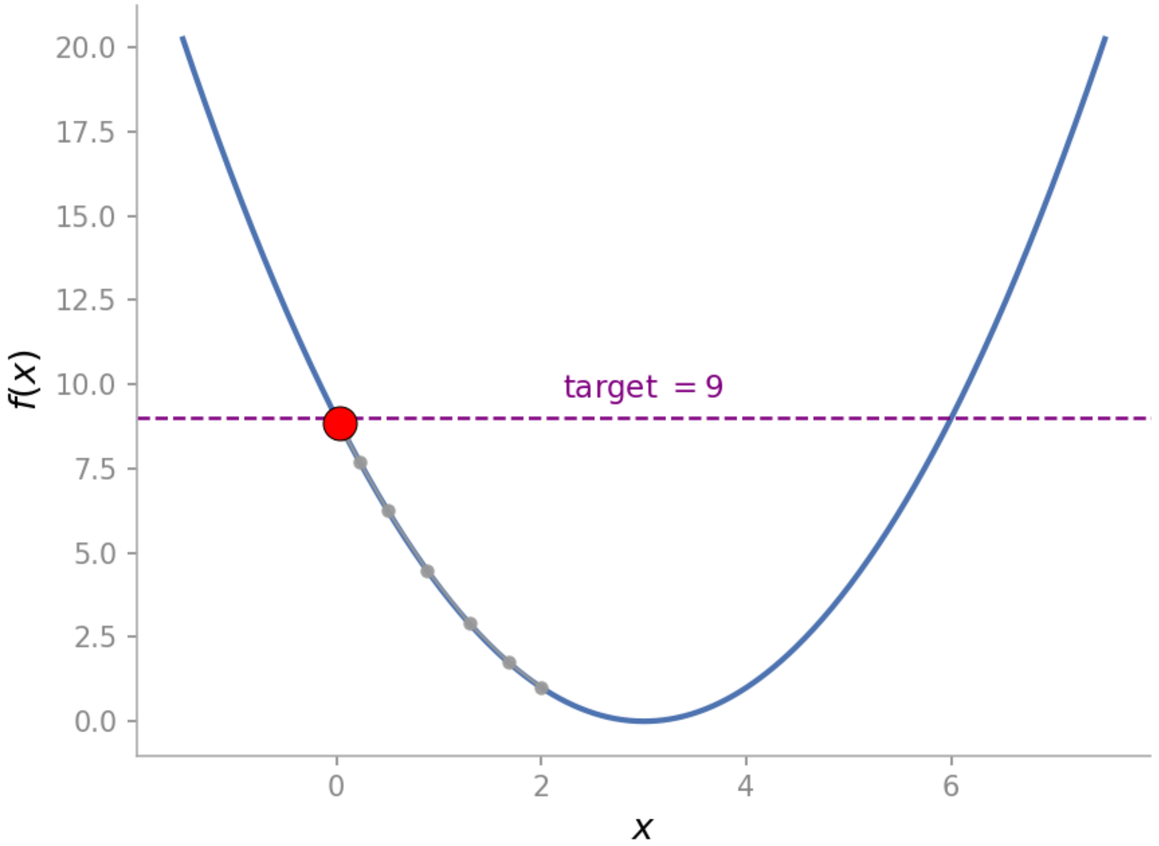
Example — visualization





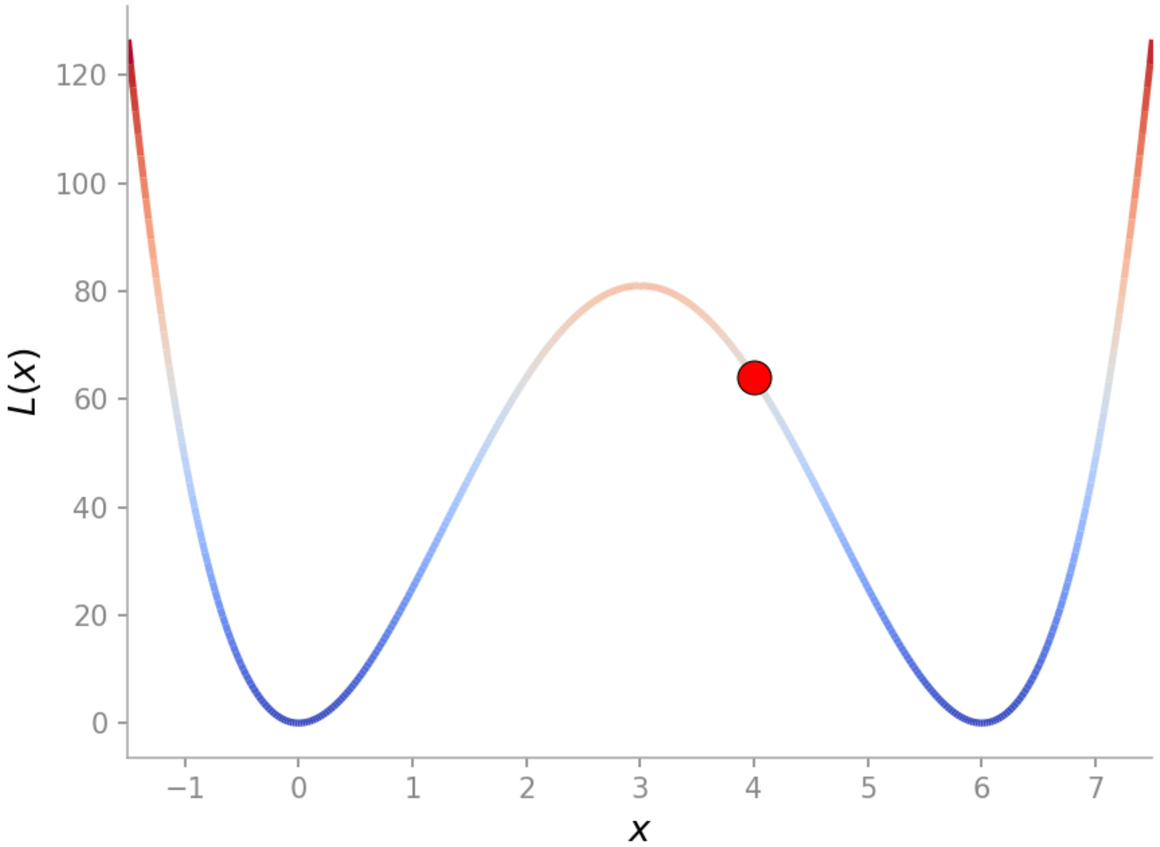
Example — visualization

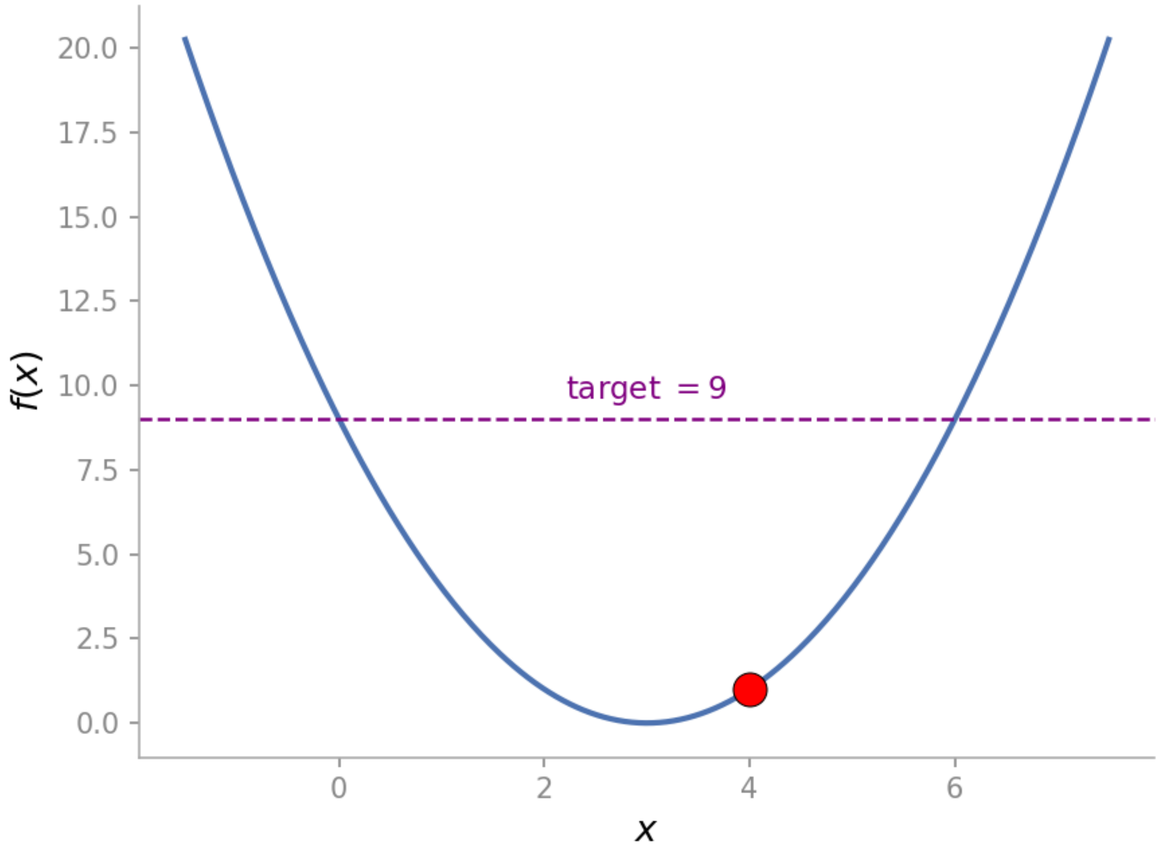




Example — different starting point

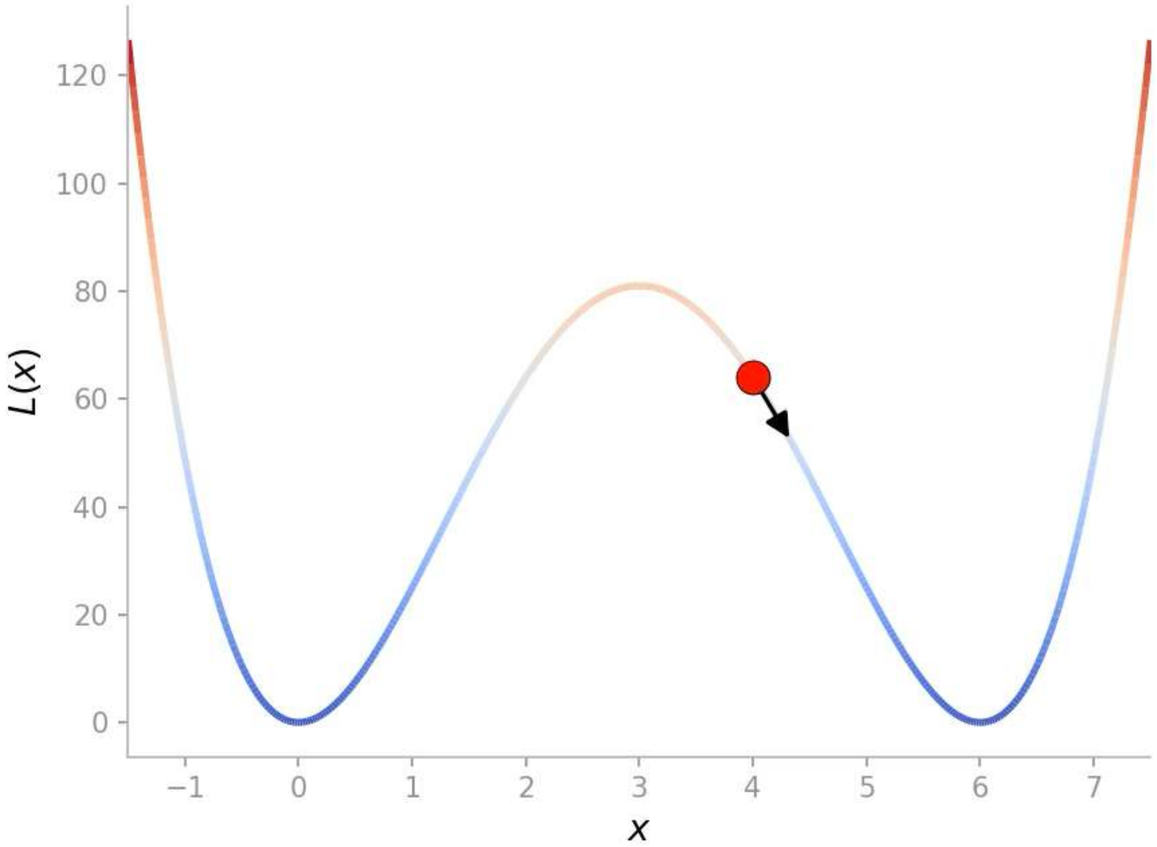
What if we initialize our parameter to $x = 4$?

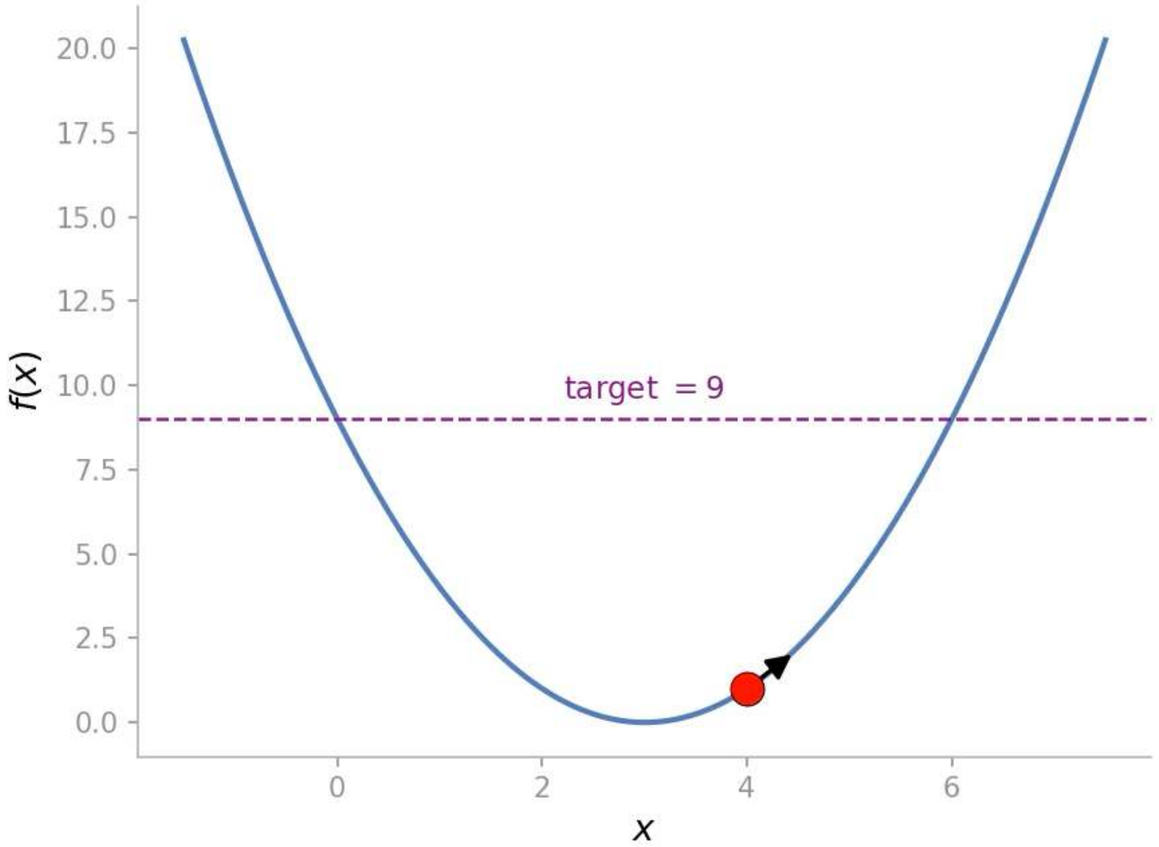




Example — different starting point

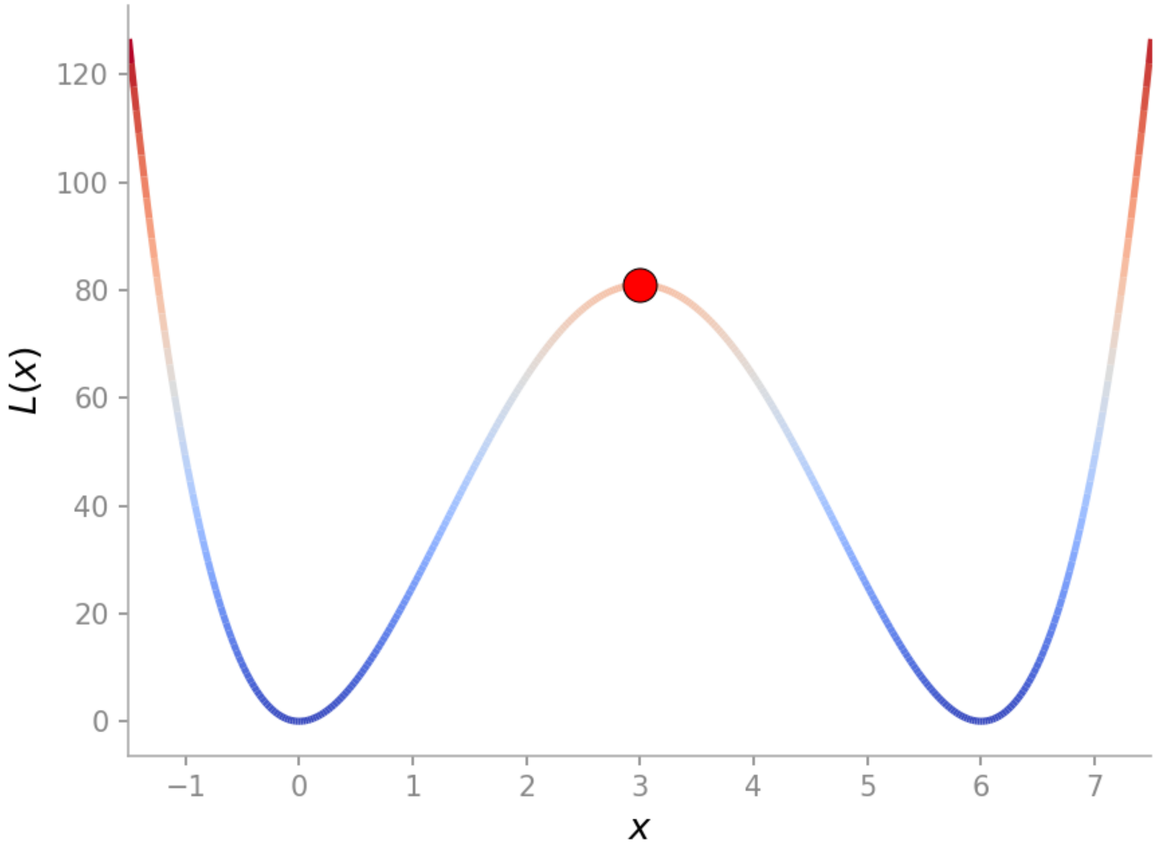
What if we initialize our parameter to $x = 4$?

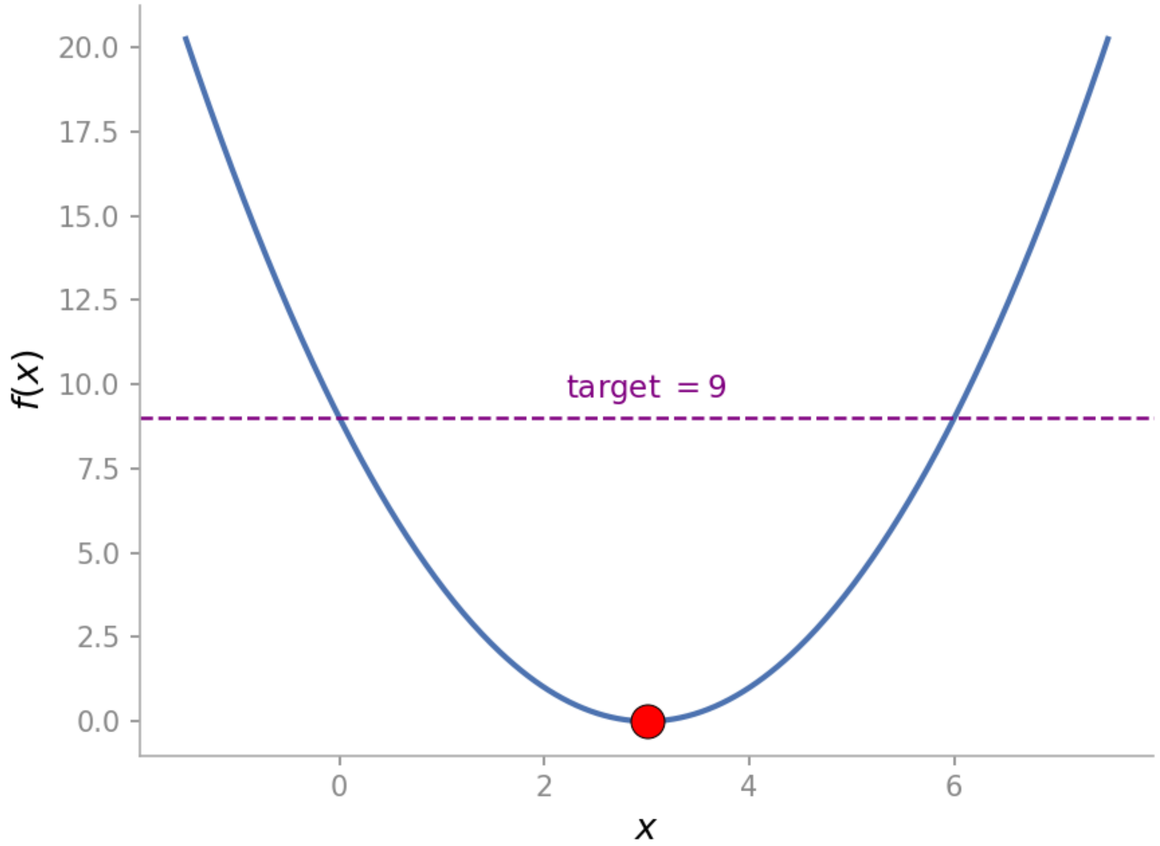




Example — different starting point

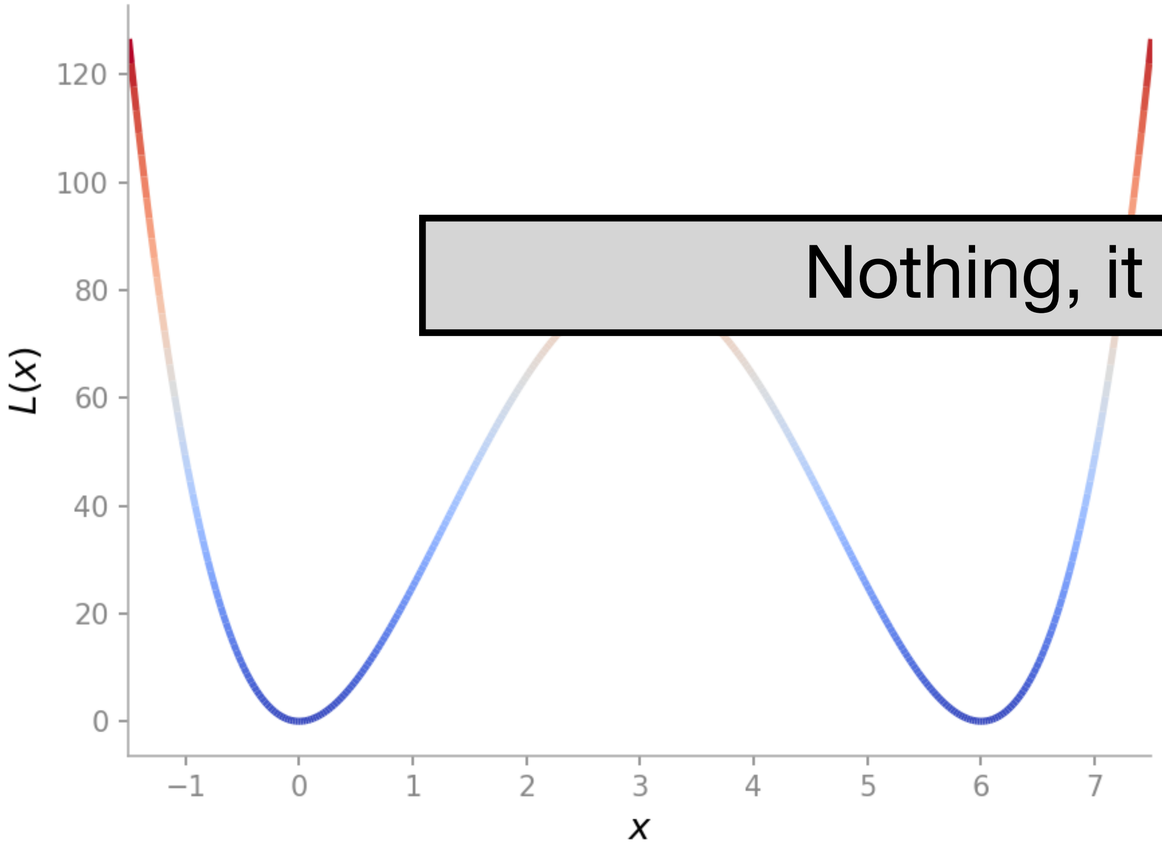
What if would happen if we initialize our parameter to $x = 3$?



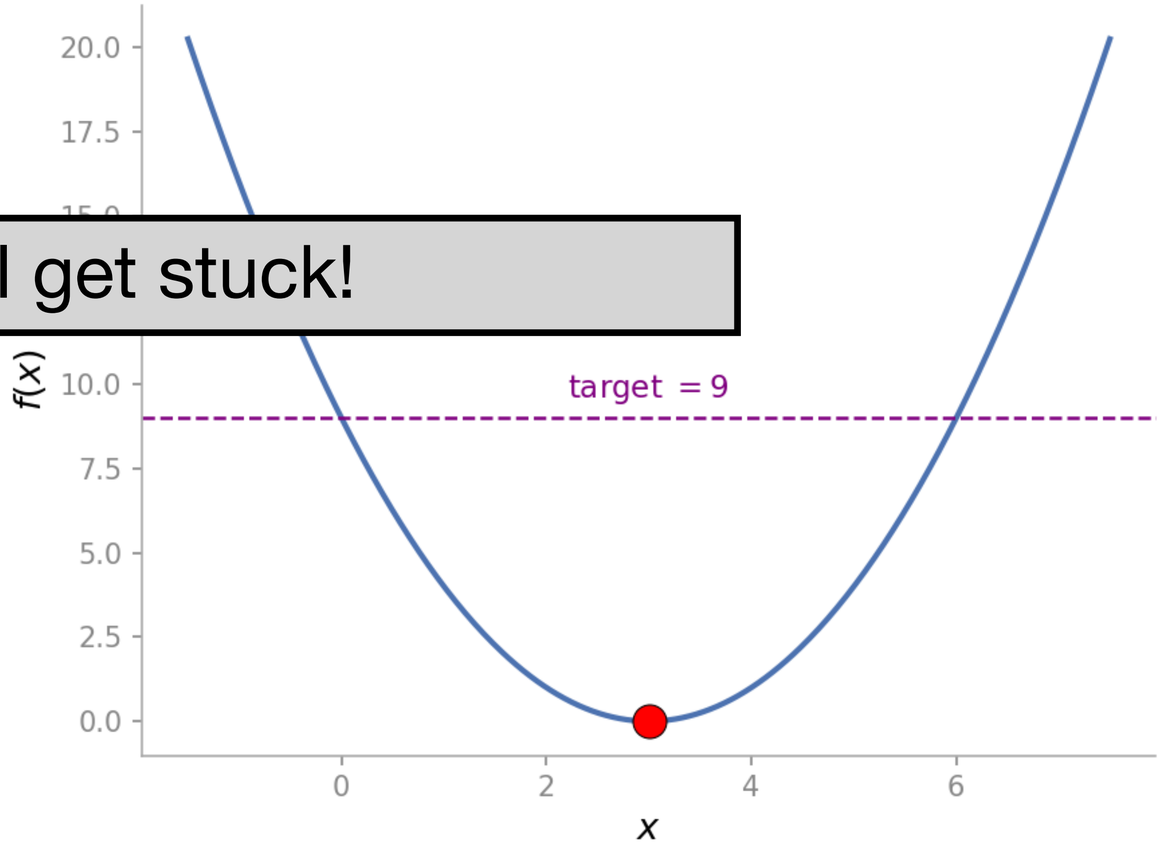


Example — different starting point

What if would happen if we initialize our parameter to $x = 3$?

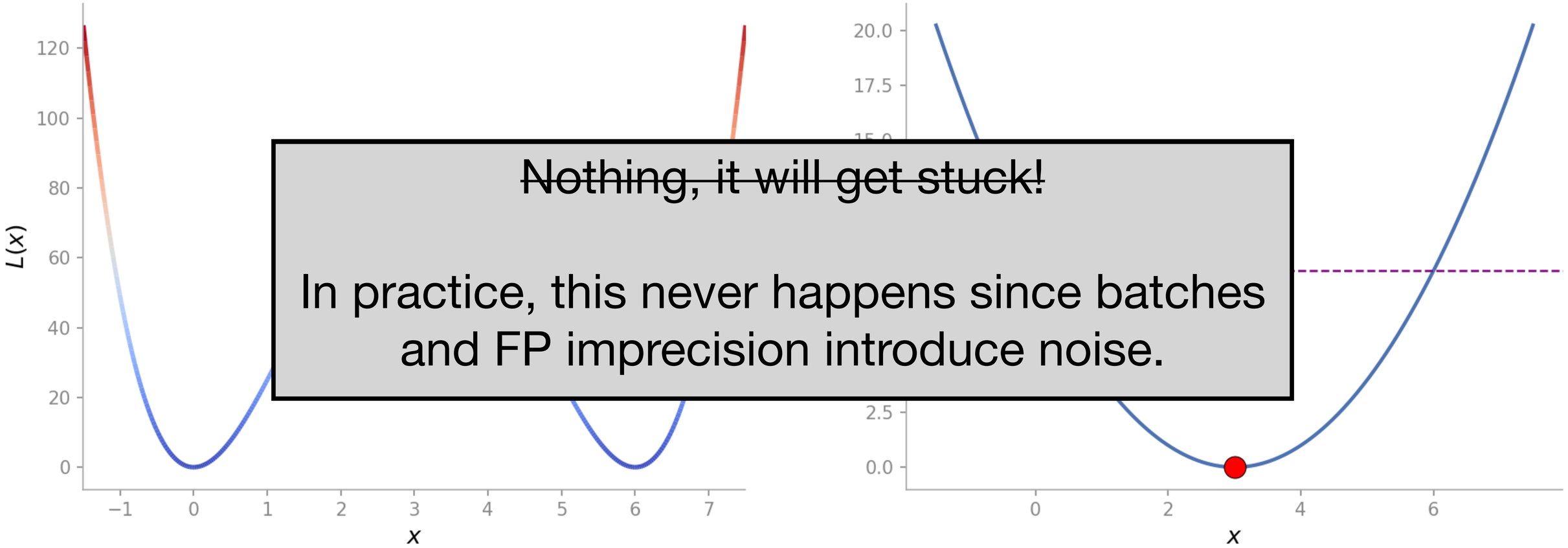


Nothing, it will get stuck!



Example — different starting point

What if would happen if we initialize our parameter to $x = 3$?



~~Nothing, it will get stuck!~~

In practice, this never happens since batches and FP imprecision introduce noise.

We can use this same approach to optimize our mesh texture!

We can use this same approach to optimize our mesh texture!

How? What is our function, parameters, target, loss, etc.?

We can use this same approach to optimize our mesh texture!

How? What is our function, parameters, target, loss, etc.?

Here is one example:

We can use this same approach to optimize our mesh texture!

How? What is our function, parameters, target, loss, etc.?

Here is one example:

- Parametric function = rendering function (how we visualize 2D images of our textured 3D shape)

We can use this same approach to optimize our mesh texture!

How? What is our function, parameters, target, loss, etc.?

Here is one example:

- Parametric function = rendering function (how we visualize 2D images of our textured 3D shape)
- Parameters = texture map pixel values, aka texel (texture pixel) values

We can use this same approach to optimize our mesh texture!

How? What is our function, parameters, target, loss, etc.?

Here is one example:

- Parametric function = rendering function (how we visualize 2D images of our textured 3D shape)
- Parameters = texture map pixel values, aka texel (texture pixel) values
- Target = some existing images of our object with the desired colors

We can use this same approach to optimize our mesh texture!

How? What is our function, parameters, target, loss, etc.?

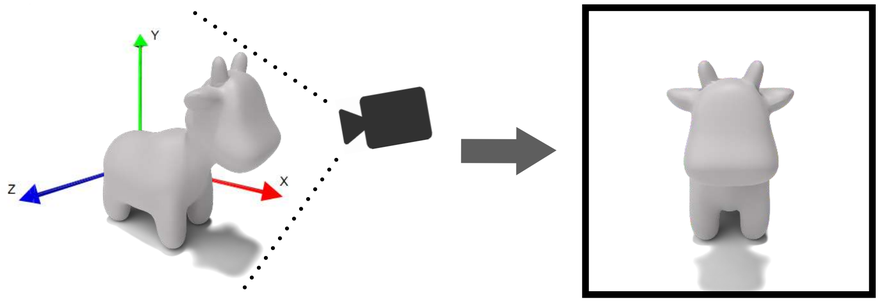
Here is one example:

- Parametric function = rendering function (how we visualize 2D images of our textured 3D shape)
- Parameters = texture map pixel values, aka texel (texture pixel) values
- Target = some existing images of our object with the desired colors
- Loss = pixel differences between our rendered images and target images

We can use this same approach to optimize our mesh texture!

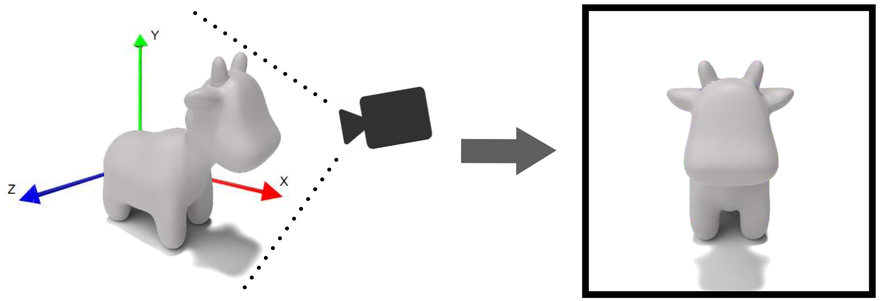
We can use this same approach to optimize our mesh texture!

- Parametric function

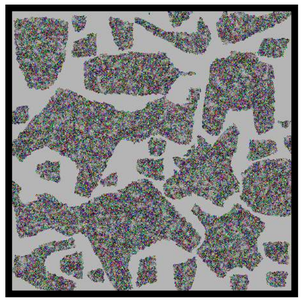


We can use this same approach to optimize our mesh texture!

- Parametric function

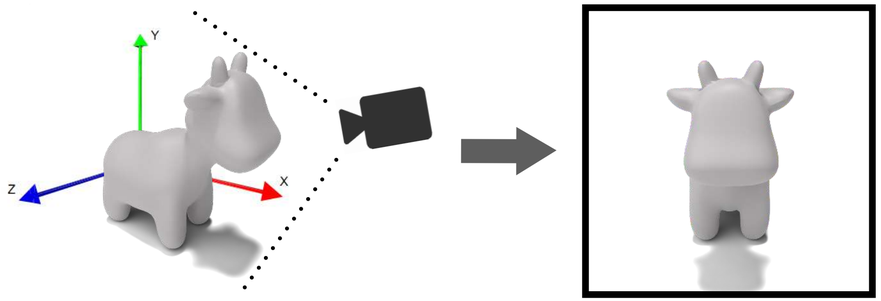


- Parameters

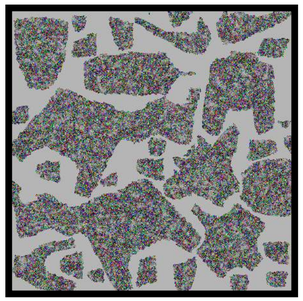


We can use this same approach to optimize our mesh texture!

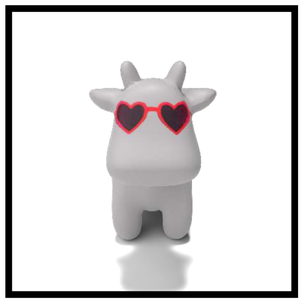
- Parametric function



- Parameters

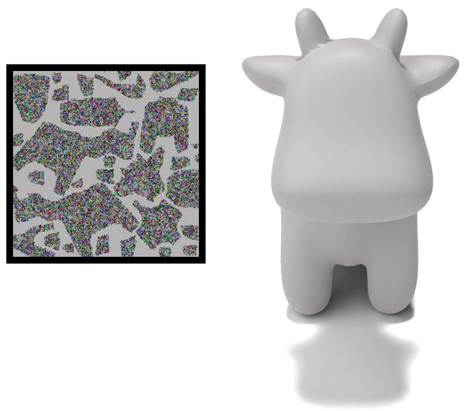


- Target

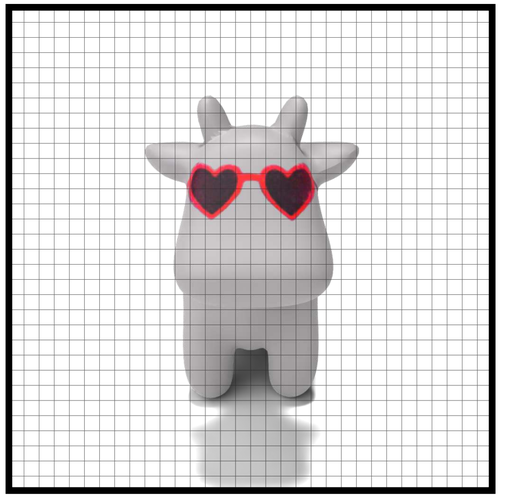


We can use this same approach to optimize our mesh texture!

Parameter(s)



Target

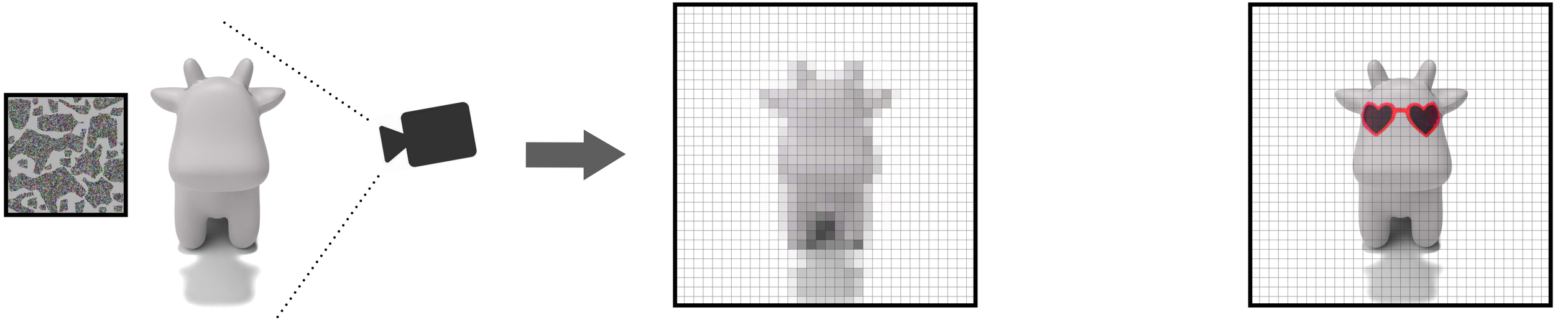


We can use this same approach to optimize our mesh texture!

Parameter(s)

Parametric Function

Target

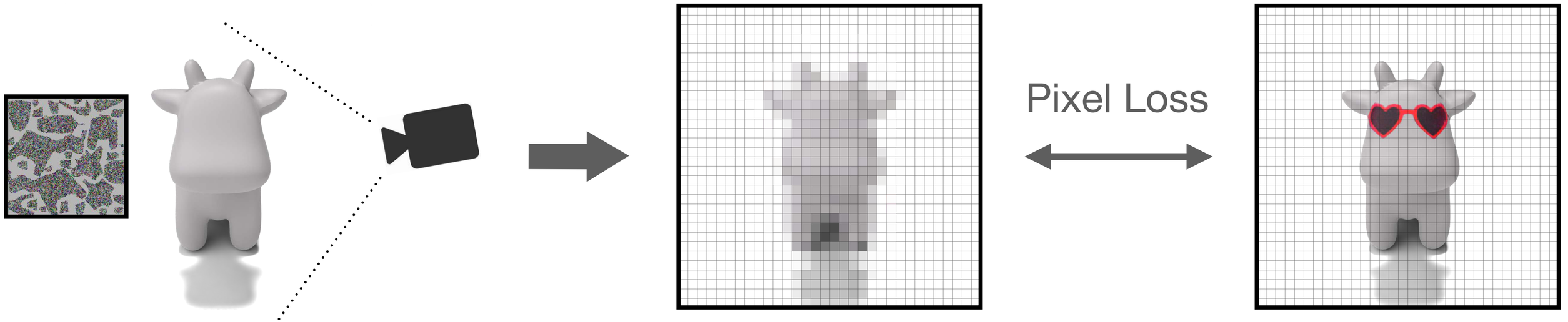


We can use this same approach to optimize our mesh texture!

Parameter(s)

Parametric Function

Target

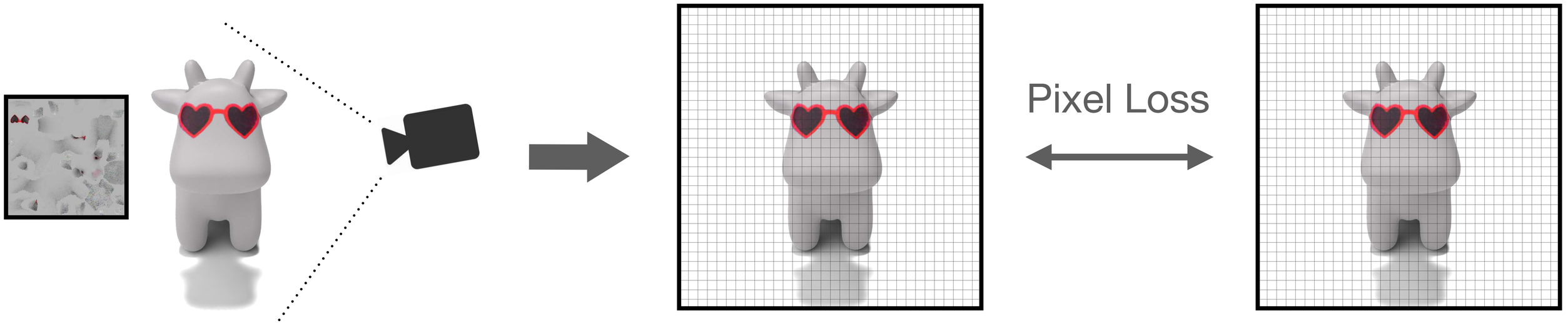


We can use this same approach to optimize our mesh texture!

Parameter(s)

Parametric Function

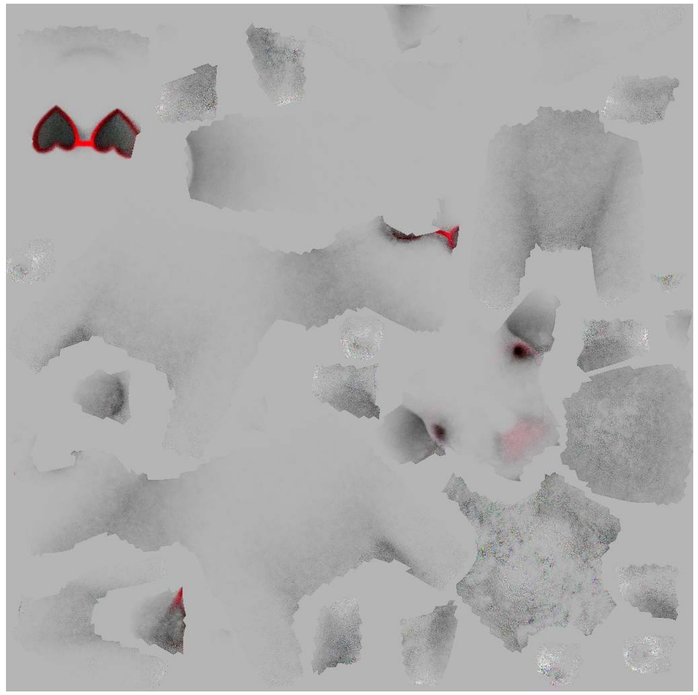
Target

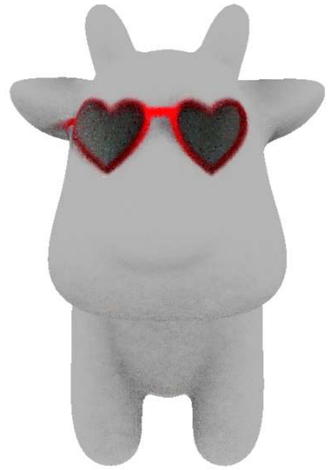


What happens if we do this?

What happens if we do this?

It works!

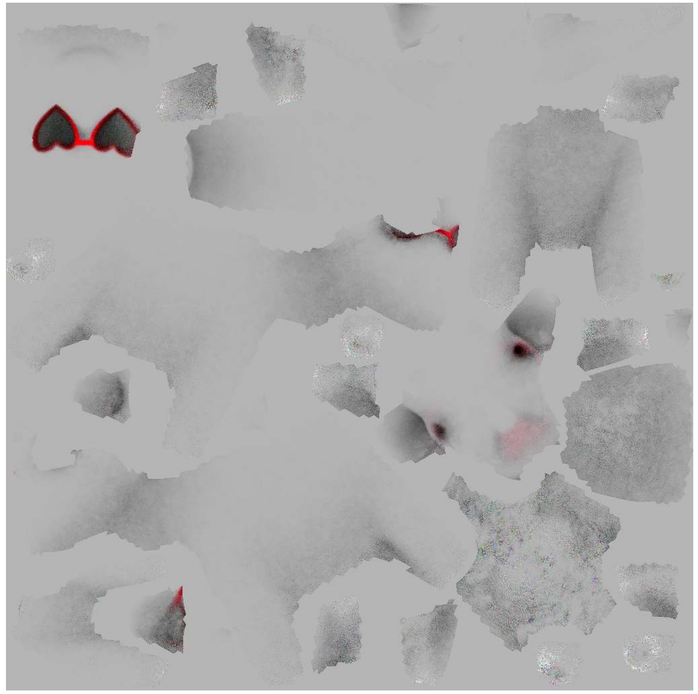


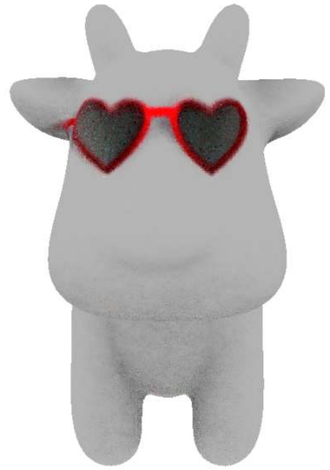


What happens if we do this?

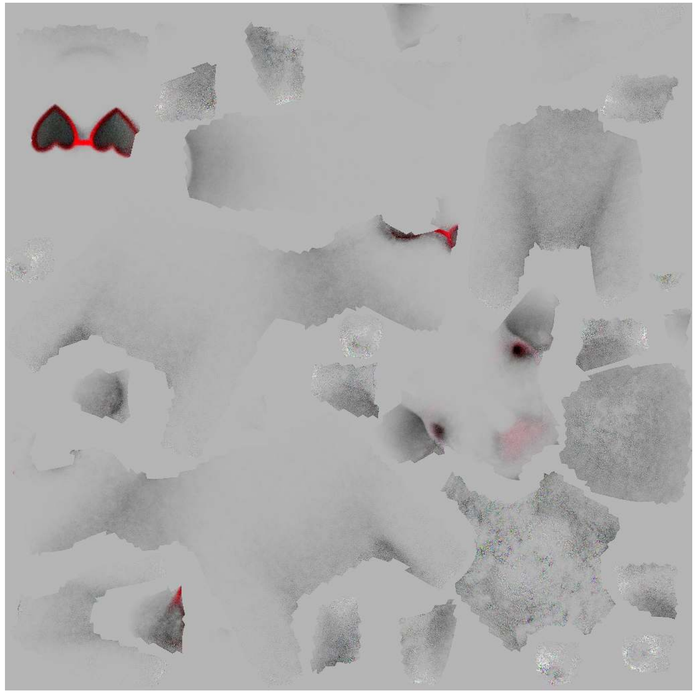
It works!

However…





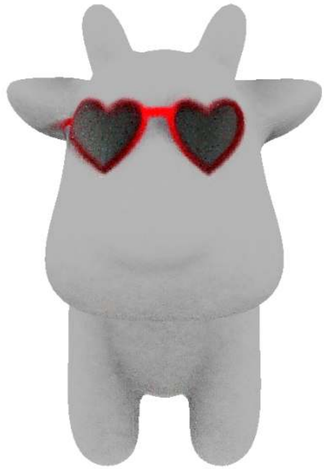
What happens if we do this?



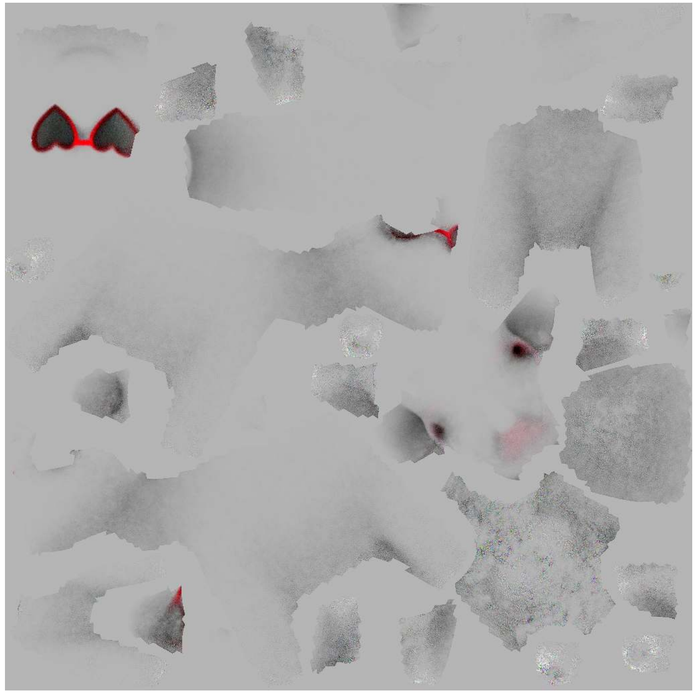
It works!

However…

- Each texel is optimized (mostly) independently



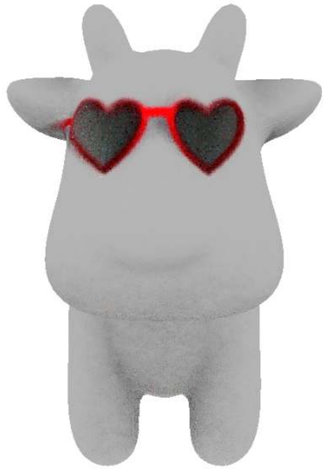
What happens if we do this?



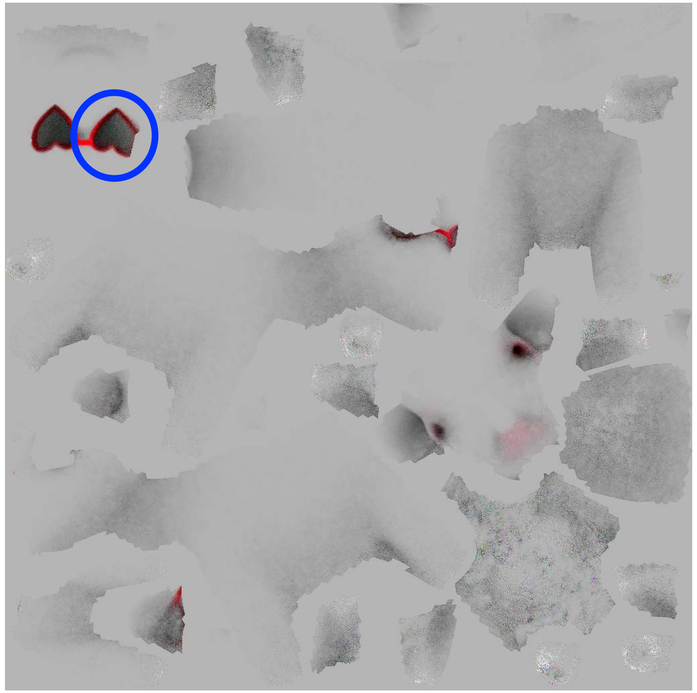
It works!

However…

- Each texel is optimized (mostly) independently
- As a result, the texture is not smooth and ends up a bit grainy/noisy



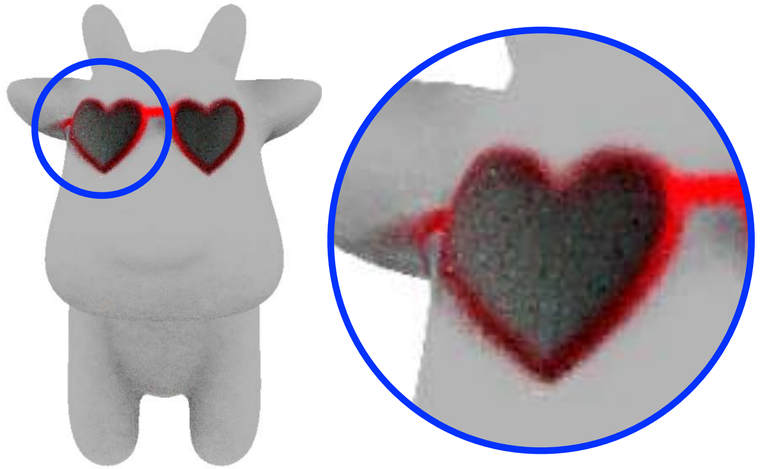
What happens if we do this?



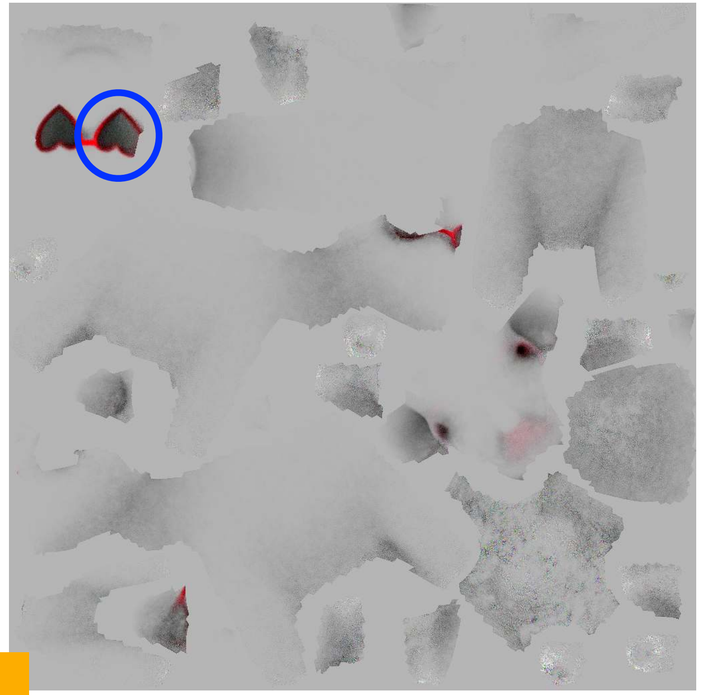
It works!

However…

- Each texel is optimized (mostly) independently
- As a result, the texture is not smooth and ends up a bit grainy/noisy
- This leads to less-than-ideal textures in 3D



What happens if we do this?

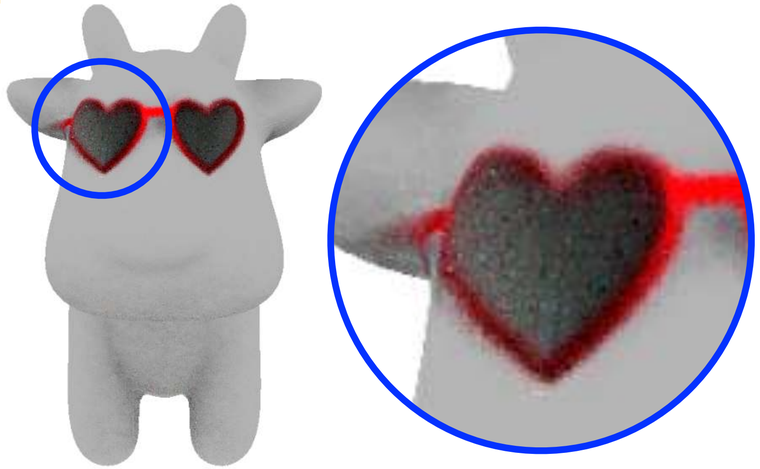


It works!

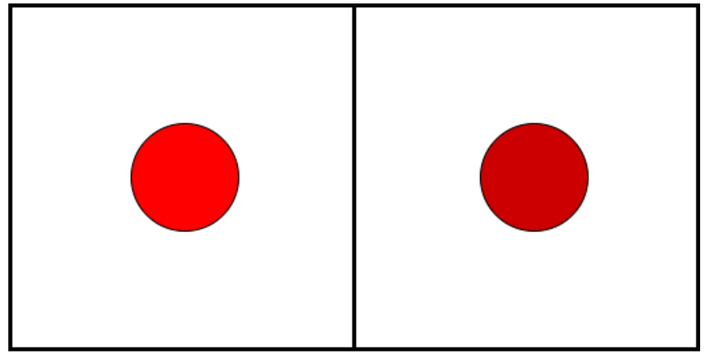
However…

- Each texel is optimized (mostly) independently
- As a result, the texture is not smooth and ends up a bit grainy/noisy
- This leads to less-than-ideal textures in 3D

How can we fix this?

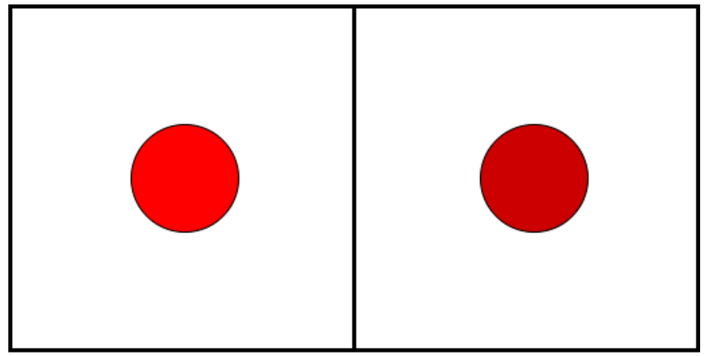


Smooth Color Function



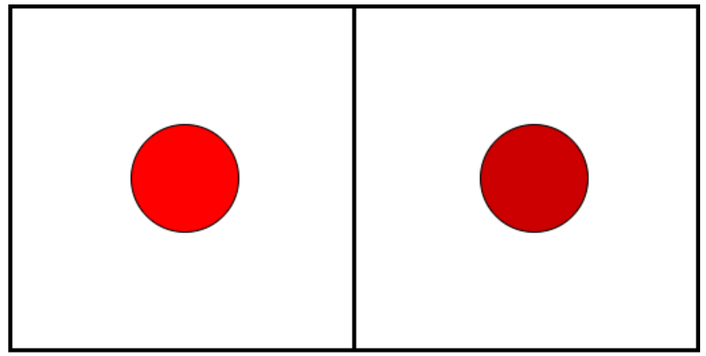
Smooth Color Function

- We can learn (optimize) a function $\phi(u, v) : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ that maps the texels to RGB color values



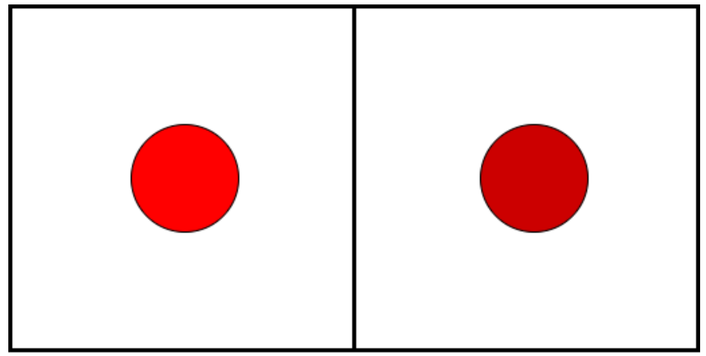
Smooth Color Function

- We can learn (optimize) a function $\phi(u, v) : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ that maps the texels to RGB color values
- To get the color at any point in the 2D texture map, just query ϕ at that UV coordinate! $RGB(u, v) = \phi(u, v)$



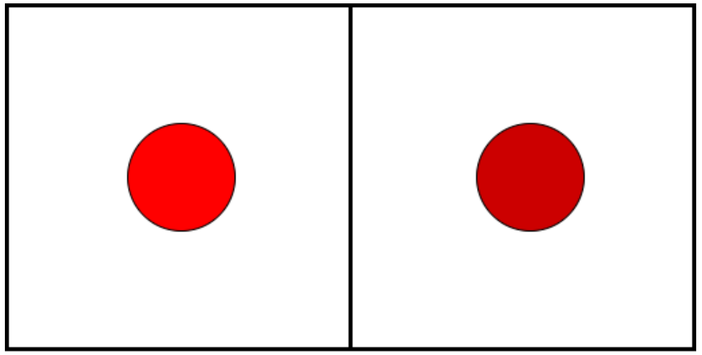
Smooth Color Function

- We can learn (optimize) a function $\phi(u, v) : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ that maps the texels to RGB color values
- To get the color at any point in the 2D texture map, just query ϕ at that UV coordinate! $RGB(u, v) = \phi(u, v)$
- If this function is “smooth”, then we are set!

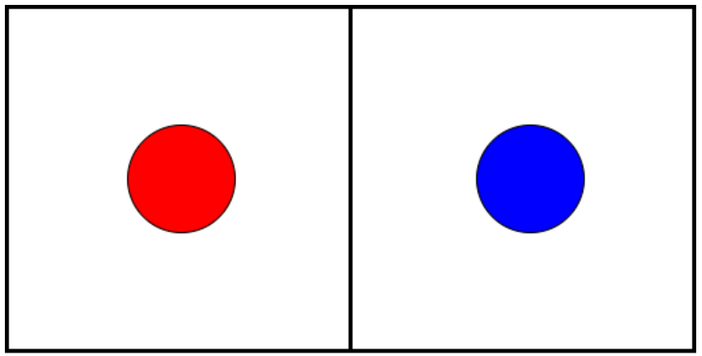


Smooth Mapping

What does smoothness mean here?



“Smooth”

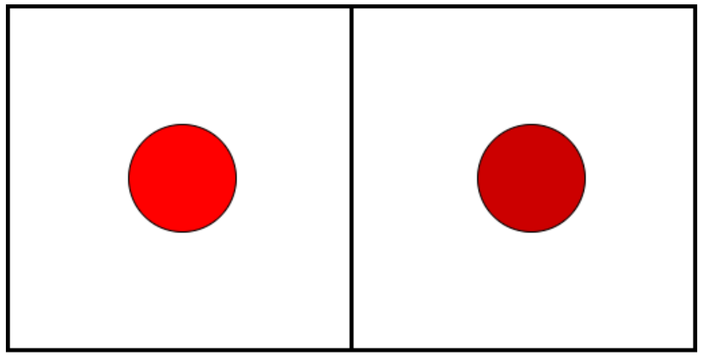


Not “Smooth”

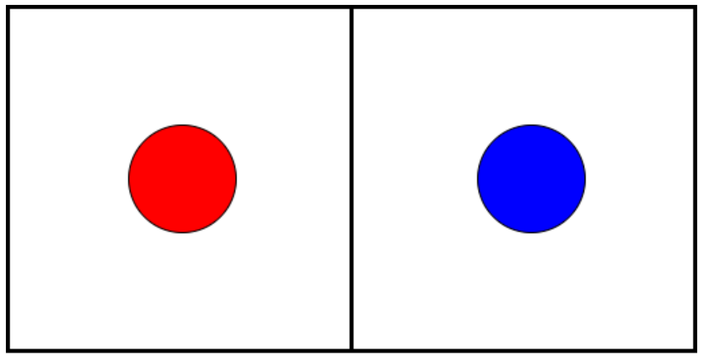
Smooth Mapping

What does smoothness mean here?

- Does this mean ϕ needs to be $C^0, C^1, \ldots, C^\infty$?



"Smooth"

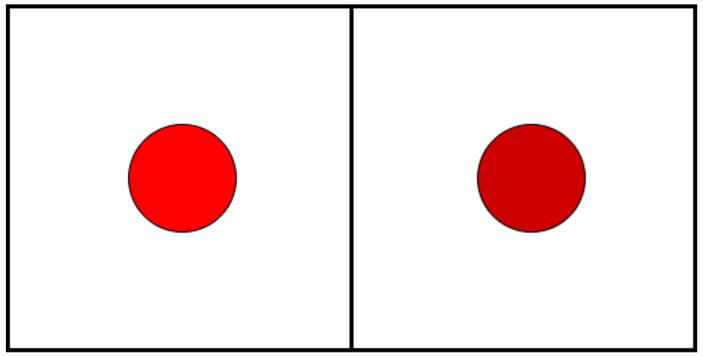


Not "Smooth"

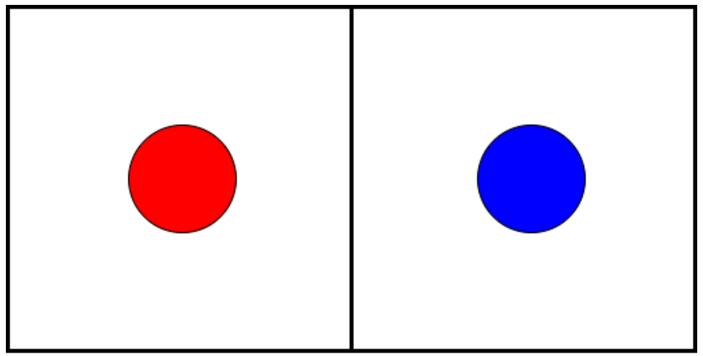
Smooth Mapping

What does smoothness mean here?

- ~~Does this mean ϕ needs to be $C^0, C^1, \ldots, C^\infty$?~~



“Smooth”

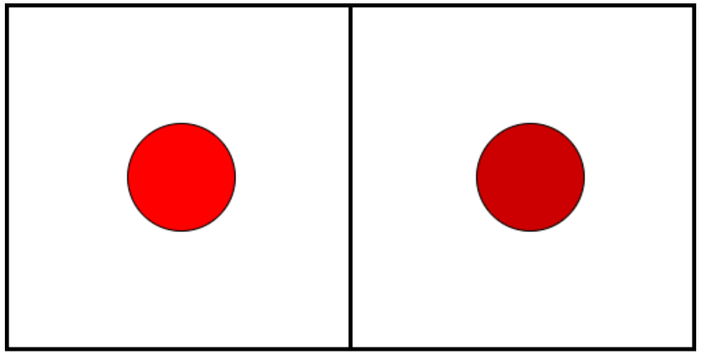


Not “Smooth”

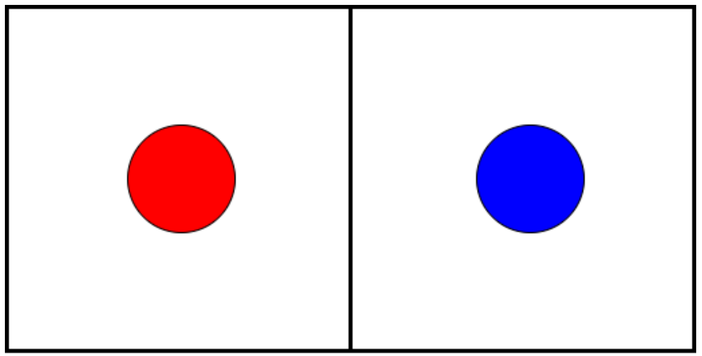
Smooth Mapping

What does smoothness mean here?

- ~~Does this mean ϕ needs to be $C^0, C^1, \ldots, C^\infty$?~~
- In practice, we want ϕ to produce similar outputs for similar inputs ***at the scales we care about***



"Smooth"

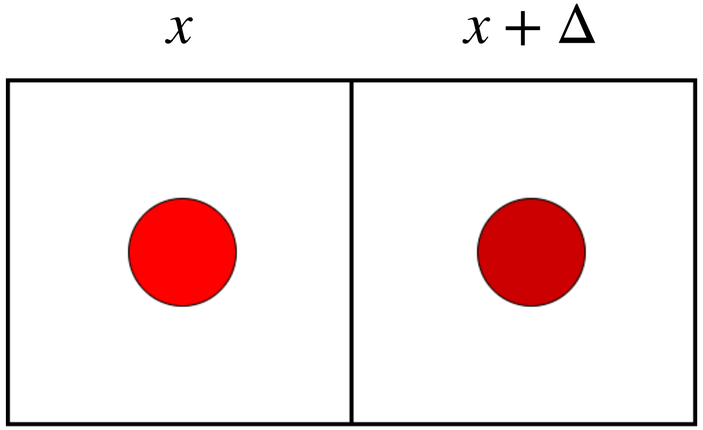


Not "Smooth"

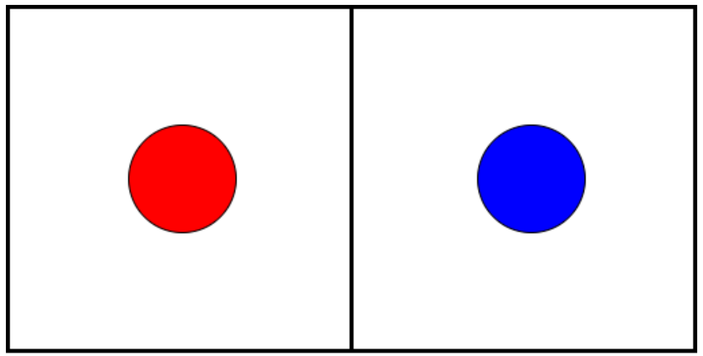
Smooth Mapping

What does smoothness mean here?

- ~~Does this mean ϕ needs to be $C^0, C^1, \ldots, C^\infty$?~~
- In practice, we want ϕ to produce similar outputs for similar inputs ***at the scales we care about***
- Want $|\phi(x + \Delta) - \phi(x)|$ to be small for Δ at the texel scale



"Smooth"

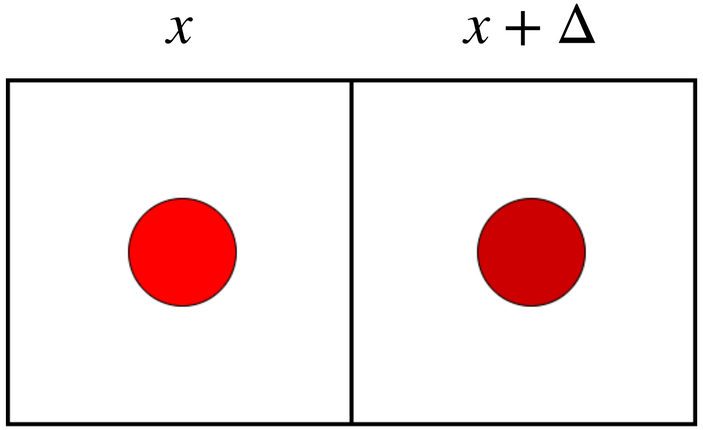


Not "Smooth"

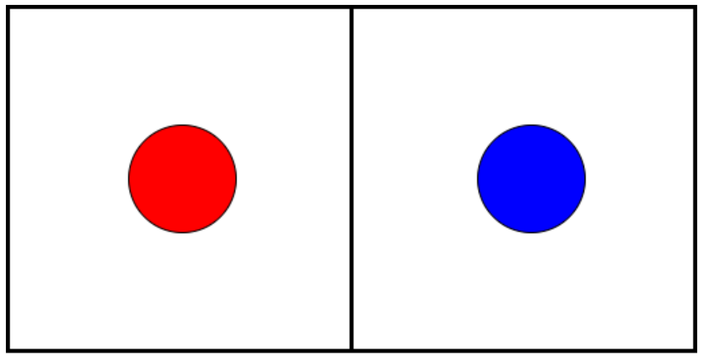
Smooth Mapping

What does smoothness mean here?

- ~~Does this mean ϕ needs to be $C^0, C^1, \ldots, C^\infty$?~~
- In practice, we want ϕ to produce similar outputs for similar inputs ***at the scales we care about***
- Want $|\phi(x + \Delta) - \phi(x)|$ to be small for Δ at the texel scale
- This way two nearby texel centers are mapped to similar RGB colors so our texture will look “smooth” and not grainy!



“Smooth”

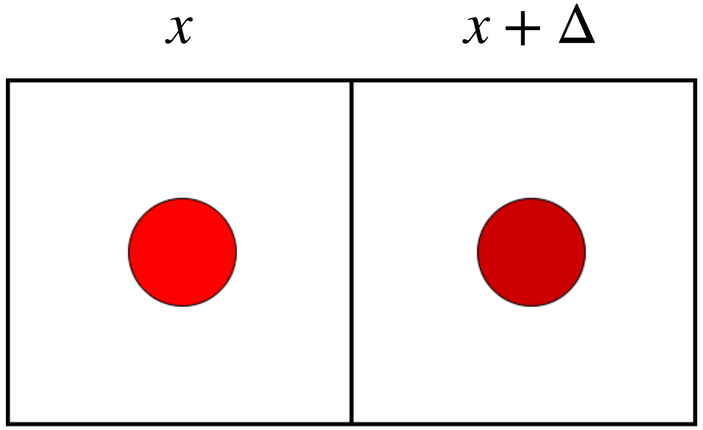


Not “Smooth”

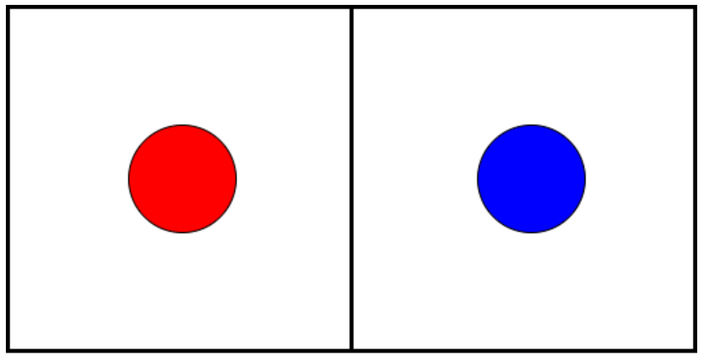
Smooth Mapping

What does smoothness mean here?

- ~~Does this mean ϕ needs to be $C^0, C^1, \ldots, C^\infty$?~~
- In practice, we want ϕ to produce similar outputs for similar inputs ***at the scales we care about***
- Want $|\phi(x + \Delta) - \phi(x)|$ to be small for Δ at the texel scale
- This way two nearby texel centers are mapped to similar RGB colors so our texture will look “smooth” and not grainy!
- How can we achieve this?



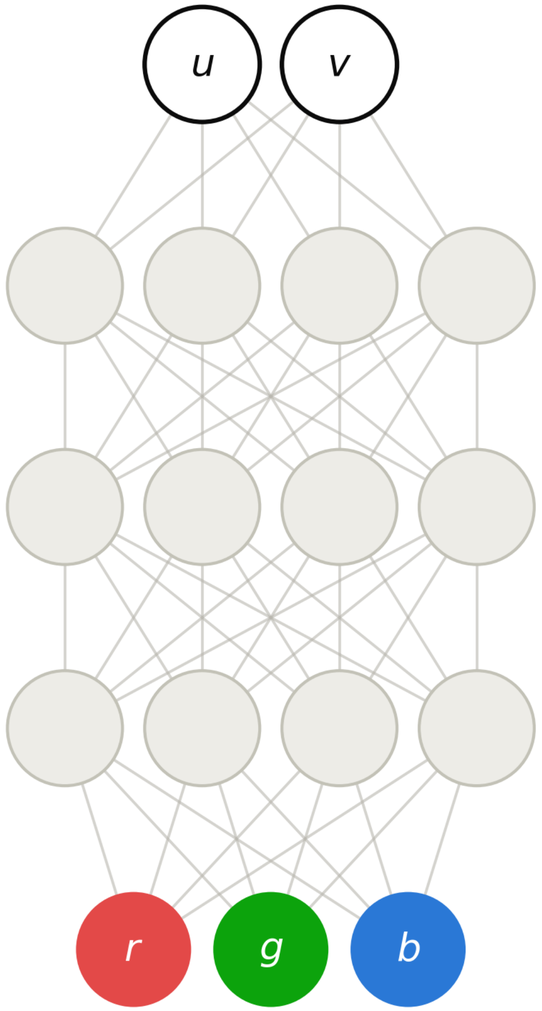
“Smooth”



Not “Smooth”

Smooth Functions: Coordinate Networks

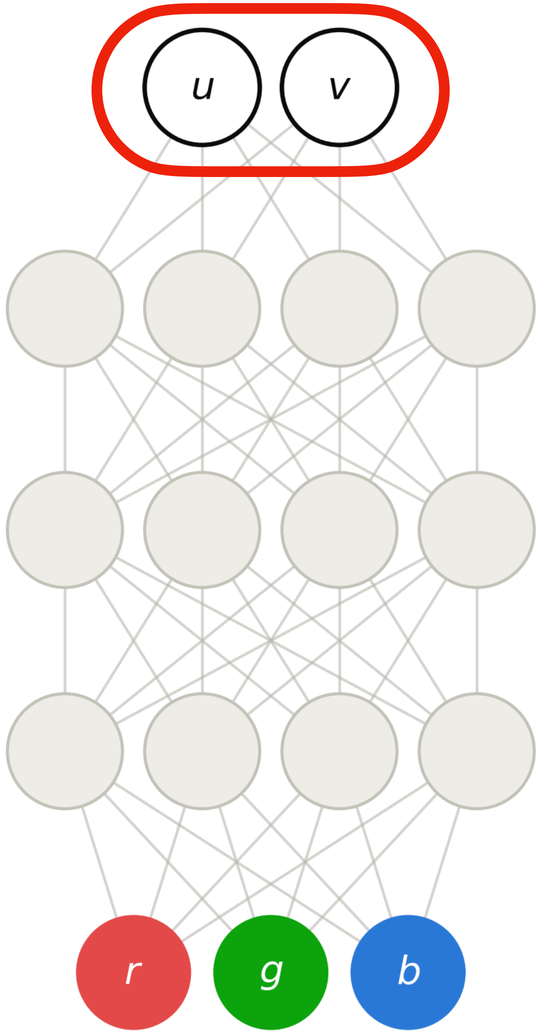
Using neural networks as our mapping ϕ



Smooth Functions: Coordinate Networks

Using neural networks as our mapping ϕ

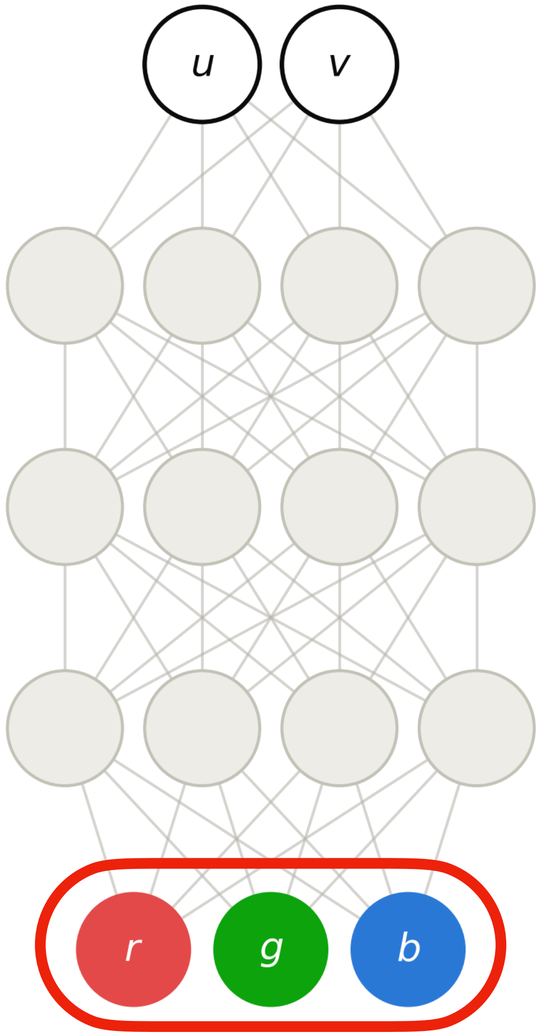
- Input: 2-dimensional UV coordinate (u,v)



Smooth Functions: Coordinate Networks

Using neural networks as our mapping ϕ

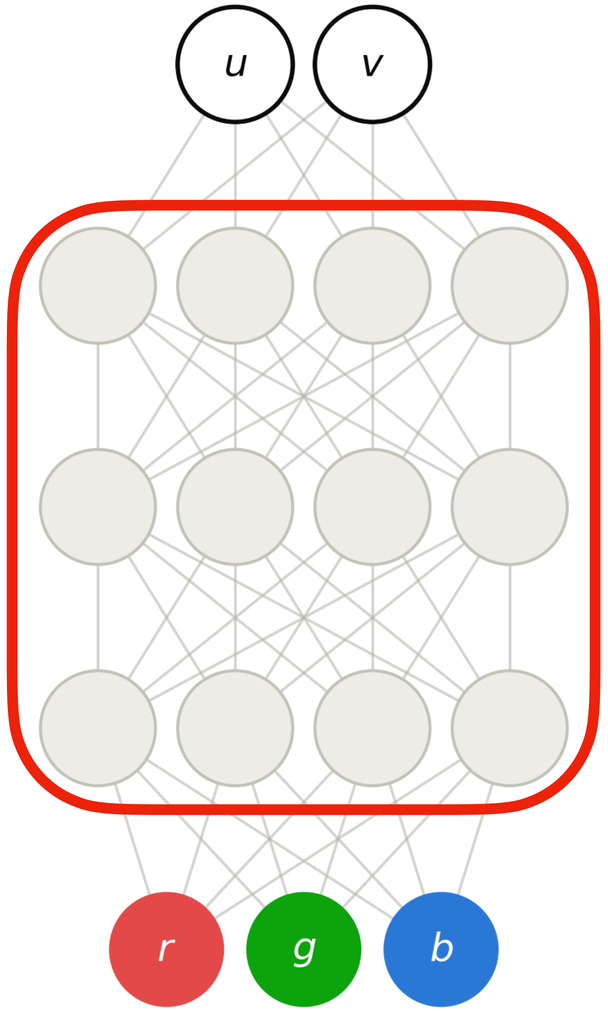
- Input: 2-dimensional UV coordinate (u,v)
- Output: 3-dimensional RGB color (r,g,b)



Smooth Functions: Coordinate Networks

Using neural networks as our mapping ϕ

- Input: 2-dimensional UV coordinate (u,v)
- Output: 3-dimensional RGB color (r,g,b)
- Through optimization, learn weights that give us the mapping we want

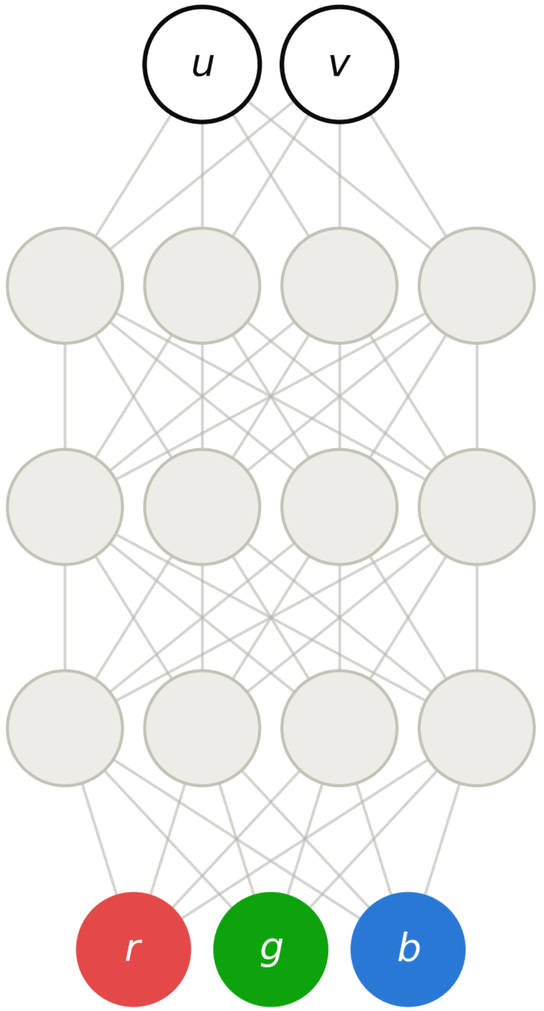


Smooth Functions: Coordinate Networks

Using neural networks as our mapping ϕ

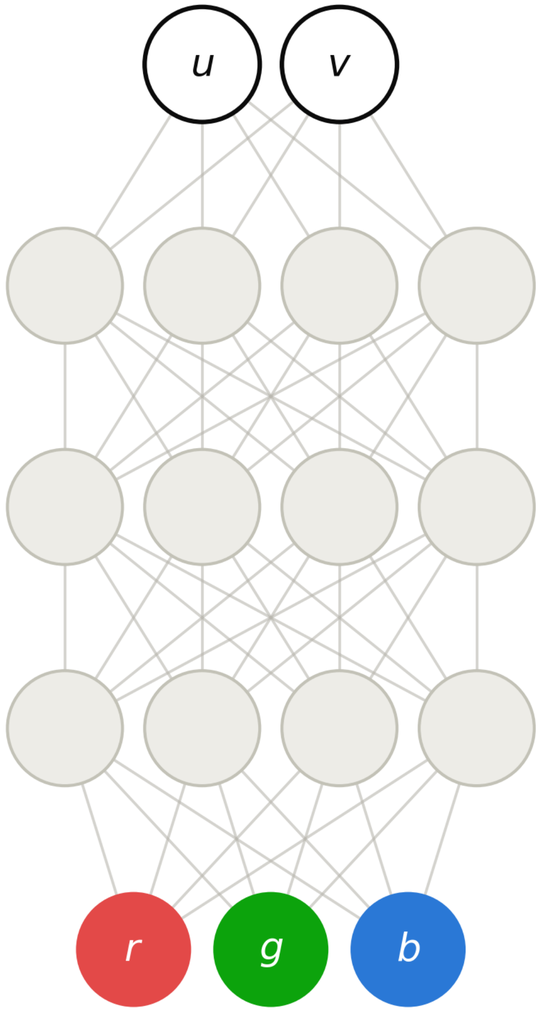
- Input: 2-dimensional UV coordinate (u,v)
- Output: 3-dimensional RGB color (r,g,b)
- Through optimization, learn weights that give us the mapping we want

Can query this network just like a function!



Smooth Functions: Coordinate Networks

But are they “smooth”?



Smooth Functions: Coordinate Networks

But are they “smooth”?

- Coordinate networks are shown to have a bias towards smoothness

arXiv:1806.08734v3 [stat.ML] 31 May 2019

On the Spectral Bias of Neural Networks

Nasim Rahaman [*1 2] Aristide Baratin [*1] Devansh Arpit [1] Felix Draxler [2] Min Lin [1] Fred A. Hamprecht [2] Yoshua Bengio [1] Aaron Courville [1]

Abstract

Neural networks are known to be a class of highly expressive functions able to fit even random input-output mappings with 100% accuracy. In this work we present properties of neural networks that complement this aspect of expressivity. By using tools from Fourier analysis, we highlight a learning bias of deep networks towards low frequency functions – i.e. functions that vary globally without local fluctuations – which manifests itself as a frequency-dependent learning speed. Intuitively, this property is in line with the observation that over-parameterized networks prioritize learning simple patterns that generalize across data samples. We also investigate the role of the shape of the data manifold by presenting empirical and theoretical evidence that, somewhat counter-intuitively, learning higher frequencies gets *easier* with increasing manifold complexity.

1. Introduction

The remarkable success of deep neural networks at generalizing to natural data is at odds with the traditional notions of model complexity and their empirically demonstrated ability to fit arbitrary random data to perfect accuracy (Zhang et al., 2017a; Arpit et al., 2017). This has prompted recent investigations of possible implicit regularization mechanisms inherent in the learning process which induce a bias towards low complexity solutions (Neyshabur et al., 2014; Soudry et al., 2017; Poggio et al., 2018; Neyshabur et al., 2017).

In this work, we take a slightly shifted view on implicit regularization by suggesting that low-complexity functions are *learned faster* during training by gradient descent. We expose this bias by taking a closer look at neural networks through the lens of Fourier analysis. While they can approximate arbitrary functions, we find that these networks prioritize learning the low frequency modes, a phenomenon we call the *spectral bias*. This bias manifests itself not just in the process of learning, but also in the parameterization of the model itself: in fact, we show that the lower frequency components of trained networks are more robust to random parameter perturbations. Finally, we also expose and analyze the rather intricate interplay between the spectral bias and the geometry of the data manifold by showing that high frequencies get easier to learn when the data lies on a lower-dimensional manifold of complex shape embedded in the input space of the model. We focus the discussion on networks with rectified linear unit (ReLU) activations, whose continuous piece-wise linear structure enables an analytic treatment.

Contributions[1]

1. We exploit the continuous piecewise-linear structure of ReLU networks to evaluate its Fourier spectrum (Section 2).
2. We find empirical evidence of a *spectral bias*: i.e. lower frequencies are learned first. We also show that lower frequencies are more robust to random perturbations of the network parameters (Section 3).
3. We study the role of the shape of the data manifold: we show how complex manifold shapes can facilitate the learning of higher frequencies and develop a theoretical understanding of this behavior (Section 4).

2. Fourier analysis of ReLU networks

2.1. Preliminaries

Throughout the paper we call ‘ReLU network’ a scalar function $f : \mathbb{R}^d \mapsto \mathbb{R}$ defined by a neural network with L hidden layers of widths $d_1, \cdots d_L$ and a single output neuron:

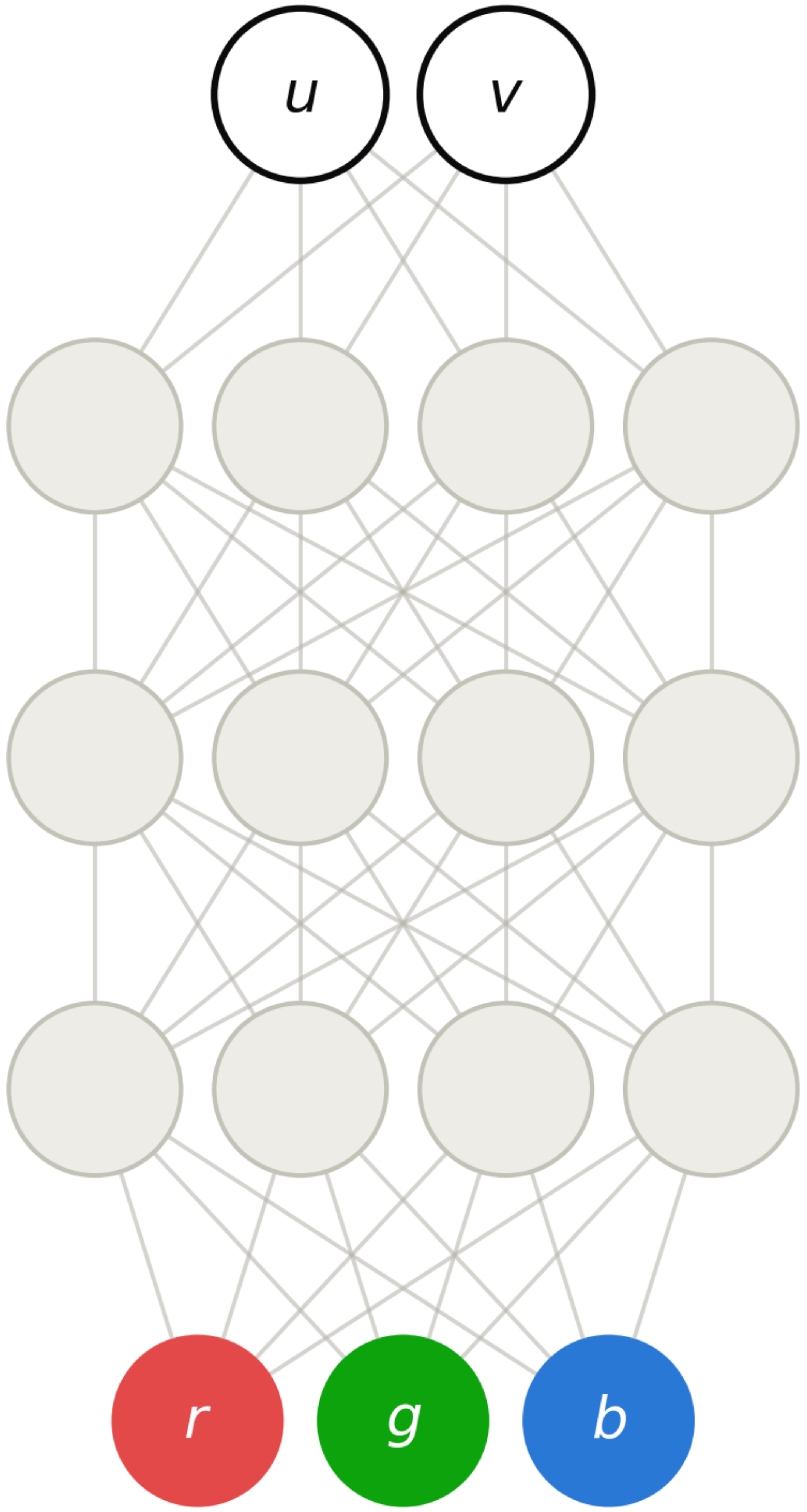
$$f(\mathbf{x}) = \left(T^{(L+1)} \circ \sigma \circ T^{(L)} \circ \cdots \circ \sigma \circ T^{(1)}\right)(\mathbf{x}) \quad (1)$$

*Equal contribution [1]Mila, Quebec, Canada [2]Image Analysis and Learning Lab, Ruprecht-Karls-Universität Heidelberg, Germany. Correspondence to: Nasim Rahaman <nasim.rahaman@live.com>, Aristide Baratin <aristide.baratin@umontreal.ca>, Devansh Arpit <devansharpit@gmail.com>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

[1]Code: https://github.com/nasimrahaman/SpectralBias

Rahaman et al, ICML 2019



Smooth Functions: Coordinate Networks

But are they “smooth”?

- Coordinate networks are shown to have a bias towards smoothness
- Low frequency features are faster and cheaper to fit

On the Spectral Bias of Neural Networks

Nasim Rahaman [*1 2] Aristide Baratin [*1] Devansh Arpit [1] Felix Draxler [1] Min Lin [1] Fred A. Hamprecht [1] Yoshua Bengio [3] Aaron Courville [3]

Abstract

Neural networks are known to be a class of highly expressive functions able to fit even random input-output mappings with 100% accuracy. In this work we present properties of neural networks that complement this aspect of expressivity. By using tools from Fourier analysis, we highlight a learning bias of deep networks towards low frequency functions – i.e. functions that vary globally without local fluctuations – which manifests itself as a frequency-dependent learning speed. Intuitively, this property is in line with the observation that over-parameterized networks prioritize learning simple patterns that generalize across data samples. We also investigate the role of the shape of the data manifold by presenting empirical and theoretical evidence that, somewhat counter-intuitively, learning higher frequencies gets easier with increasing manifold complexity.

1. Introduction

The remarkable success of deep neural networks at generalizing to natural data is at odds with the traditional notions of model complexity and their empirically demonstrated ability to fit arbitrary random data to perfect accuracy (Zhang et al., 2017a; Arpit et al., 2017). This has prompted recent investigations of possible implicit regularization mechanisms inherent in the learning process which induce a bias towards low complexity solutions (Neyshabur et al., 2014; Soudry et al., 2017; Poggio et al., 2018; Neyshabur et al., 2017).

In this work, we take a slightly shifted view on implicit regularization by suggesting that low-complexity functions are *learned faster* during training by gradient descent. We expose this bias by taking a *closer* look at neural networks through the lens of Fourier analysis. While they can approximate arbitrary functions, we find that these networks prioritize learning the low frequency modes, a phenomenon we call the *spectral bias*. This bias manifests itself not just in the process of learning, but also in the parameterization of the model itself: in fact, we show that the lower frequency components of trained networks are more robust to random parameter perturbations. Finally, we also expose and analyze the rather intricate interplay between the spectral bias and the geometry of the data manifold by showing that high frequencies get easier to learn when the data lies on a lower-dimensional manifold, or more generally, when there is more input space of the model. We focus the discussion on networks with rectified linear unit (ReLU) activations, whose continuous piece-wise linear structure enables an analytic treatment.

Contributions[1]

1. We exploit the continuous piecewise-linear structure of ReLU networks to evaluate its Fourier spectrum (Section 2).
2. We find empirical evidence of a *spectral bias*: i.e. lower frequencies are learned first. We also show that lower frequencies are more robust to random perturbations of the network parameters (Section 3).
3. We study the role of the shape of the data manifold: we show how complex manifold shapes can facilitate the learning of higher frequencies and develop a theoretical understanding of this behavior (Section 4).

2. Fourier analysis of ReLU networks

2.1. Preliminaries

Throughout the paper we call 'ReLU network' a scalar function $f : \mathbb{R}^d \to \mathbb{R}$ defined by a neural network with L hidden layers of widths $d_1, \cdots, d_L$ and a single output neuron:

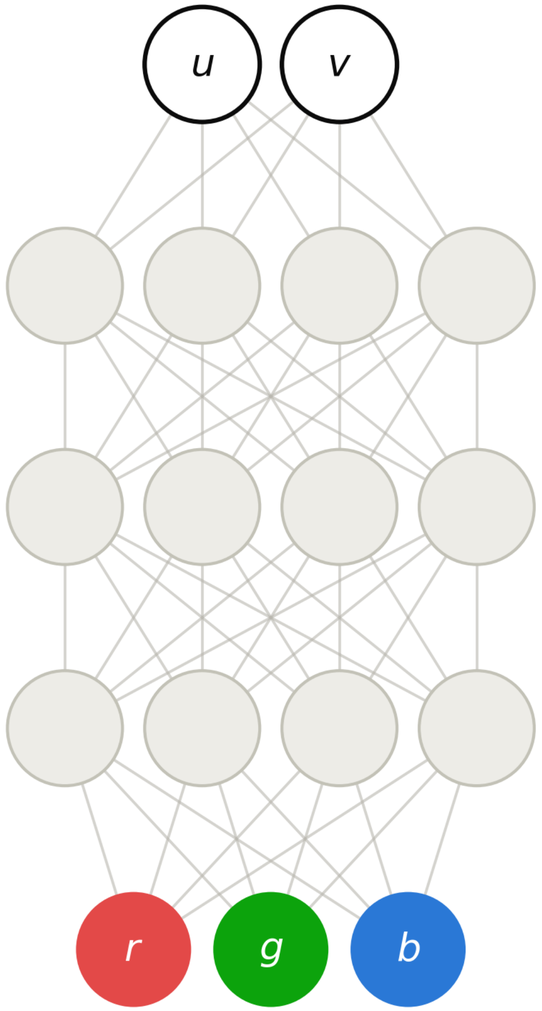
$$f(\mathbf{x}) = \left(T^{(L+1)} \circ \sigma \circ T^{(L)} \circ \cdots \circ \sigma \circ T^{(1)}\right)(\mathbf{x}) \quad (1)$$

[1]Code: https://github.com/nasimrahaman/SpectralBias

arXiv:1806.08734v3 [stat.ML] 31 May 2019

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

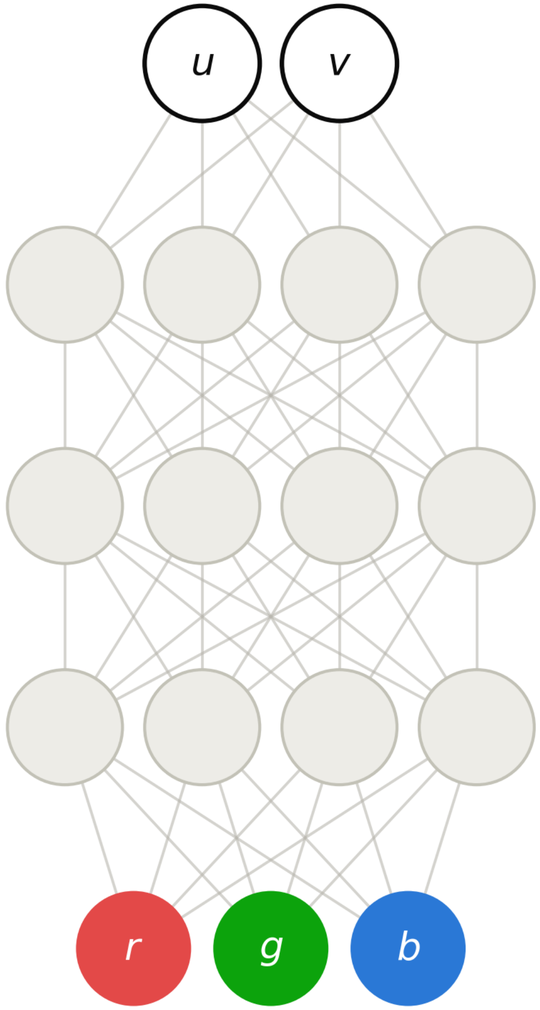
Rahaman et al, ICML 2019



Smooth Functions: Coordinate Networks

But are they “smooth”?

- Coordinate networks are shown to have a bias towards smoothness
- Low frequency features are faster and cheaper to fit
- In practice, can be too smooth! Use positional encodings to align learned feature frequency w/ texel scale



On the Spectral Bias of Neural Networks

Nasim Rahaman * 1 2 Aristide Baratin * 1 Devansh Arpit 1 Felix Draxler 1 Min Lin 1 Fred A. Hamprecht 1 Yoshua Bengio 1 Aaron Courville 1 2

Abstract

Neural networks are known to be a class of highly expressive functions able to fit even random input-output mappings with 100% accuracy. In this work we present properties of neural networks that complement this aspect of expressivity. By using tools from Fourier analysis, we highlight a learning bias of deep networks towards low frequency functions – i.e. functions that vary globally without local fluctuations – which manifests itself as a frequency-dependent learning speed. Intuitively, this property is in line with the observation that over-parameterized networks prioritize learning simple patterns that generalize across data samples. We also investigate the role of the shape of the data manifold by presenting empirical and theoretical evidence that, somewhat counter-intuitively, learning higher frequencies gets easier with increasing manifold complexity, and expose this bias by taking a closer look at neural networks through the lens of Fourier analysis. While they can approximate arbitrary functions, we find that these networks prioritize learning the low frequency modes, a phenomenon we call the *spectral bias*. This bias manifests itself not just in the process of learning, but also in the parameterization of the model itself: in fact, we show that the lower frequency components of trained networks are more robust to random parameter perturbations. Finally, we also expose and analyze the rather intricate interplay between the spectral bias and the geometry of the data manifold by showing that high frequencies get easier to learn when the data lies on a lower-dimensional manifold, and validate this empirically with a synthetic handwidth. We support an approach for selecting problem-specific Fourier features that greatly improves the performance of MLPs for low-dimensional regression tasks relevant to the computer vision and graphics communities.

1. Introduction

The remarkable success of deep neural networks at generalizing to natural data is at odds with the traditional notions of model complexity and their empirically demonstrated ability to fit arbitrary random data to perfect accuracy (Zhang et al., 2017a; Arpit et al., 2017). This has prompted recent investigations of possible implicit regularization mechanisms inherent in the learning process which induce a bias towards low complexity solutions (Neyshabur et al., 2014; Soudry et al., 2017; Poggio et al., 2018; Neyshabur et al., 2017).

In this work, we take a slightly shifted view on implicit regularization by suggesting that low-complexity functions are *learned faster* during training by gradient descent. We

*Equal contribution 1Mila, Quebec, Canada 2Image Analysis and Learning Lab, Ruprecht-Karls-University Heidelberg, Germany. Correspondence to: Nasim Rahaman <nasim.rahaman@live.com>, Aristide Baratin <aristide.baratin@umontreal.ca>, Devansh Arpit <devansharpit@gmail.com>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

1. We exploit the continuous piecewise-linear structure of ReLU networks to evaluate its Fourier spectrum (Section 2).
2. We find empirical evidence of a *spectral bias*: i.e. lower frequencies are learned first. We also show that lower frequencies are more robust to random perturbations of the network parameters (Section 3).
3. We study the role of the shape of the data manifold: we show how complex manifold shapes can facilitate the learning of higher frequencies and develop a theoretical understanding of this behavior (Section 4).

2. Fourier analysis of ReLU networks

2.1. Preliminaries

Throughout the paper we call ‘ReLU network’ a scalar function $f : \mathbb{R}^d \to \mathbb{R}$ defined by a neural network with L hidden layers of widths $d_1, \cdots, d_L$ and a single output neuron:

$$f(\mathbf{x}) = \left(T^{(L+1)} \circ \sigma \circ T^{(L)} \circ \cdots \circ \sigma \circ T^{(1)}\right)(\mathbf{x}) \quad (1)$$

[1]Code: https://github.com/nasimrahaman/SpectralBias

Rahaman et al, ICML 2019

Fourier Features Let Networks Learn
High Frequency Functions in Low Dimensional Domains

Matthew Tancik[1]* Pratul P. Srinivasan[1]* Ben Mildenhall[1]* Sara Fridovich-Keil[1]
Nithin Raghavan[1] Utkarsh Singhal[1] Ravi Ramamoorthi[2] Jonathan T. Barron[2] Ren Ng[1]

[1]University of California, Berkeley [2]Google Research [3]University of California, San Diego

Abstract

We show that passing input points through a simple Fourier feature mapping enables a multilayer perceptron (MLP) to learn high-frequency functions in low-dimensional problem domains. These results shed light on recent advances in computer vision and graphics that achieve state-of-the-art results by using MLPs to represent complex 3D objects and scenes. Using tools from the neural tangent kernel (NTK) literature, we show that a standard MLP fails to learn high frequencies both in theory and in practice. To overcome this spectral bias, we use a Fourier feature mapping to transform the effective NTK into a stationary kernel with a tunable bandwidth. We suggest an approach for selecting problem-specific Fourier features that greatly improves the performance of MLPs for low-dimensional regression tasks relevant to the computer vision and graphics communities.

1 Introduction

A recent line of research in computer vision and graphics replaces traditional discrete representations of objects (e.g. voxel grids, meshes, point clouds, and octrees) with continuous functions that are parameterized by deep fully-connected networks (also called multilayer perceptrons or MLPs). These MLPs, which we will call "coordinate-based" MLPs, take low-dimensional coordinates as inputs (typically points in $\mathbb{R}^3$) and are trained to output a representation of shape, density, and/or color at each input location (see Figure 1). This strategy is compelling since coordinate-based MLPs are amenable to gradient-based optimization and machine learning, and can be orders of magnitude more compact than grid-sampled representations. Coordinate-based MLPs have been used to represent shapes [29, 7, 22], to represent scenes containing multiple objects [24], to represent both shape and occupancy [24], and signed distance [32], and have achieved state-of-the-art results across a variety of tasks such as shape representation [9, 10, 12, 13, 17, 26, 32], texture synthesis [15, 31], shape inference from images [22, 23], and novel-view synthesis [27, 29, 35, 37].

We leverage recent progress in modeling the behavior of deep networks using kernel regression with a neural tangent kernel (NTK) [16] to theoretically and experimentally show that standard MLPs are poorly suited for these low-dimensional coordinate-based vision and graphics tasks. In particular, MLPs have difficulty learning high frequency functions, a phenomenon referred to in the literature as "spectral bias" [3, 33]. NTK theory suggests that this is because standard coordinate-based MLPs correspond to kernels with a rapid frequency falloff, which effectively prevents them from being able to represent the high-frequency content present in natural images and scenes.

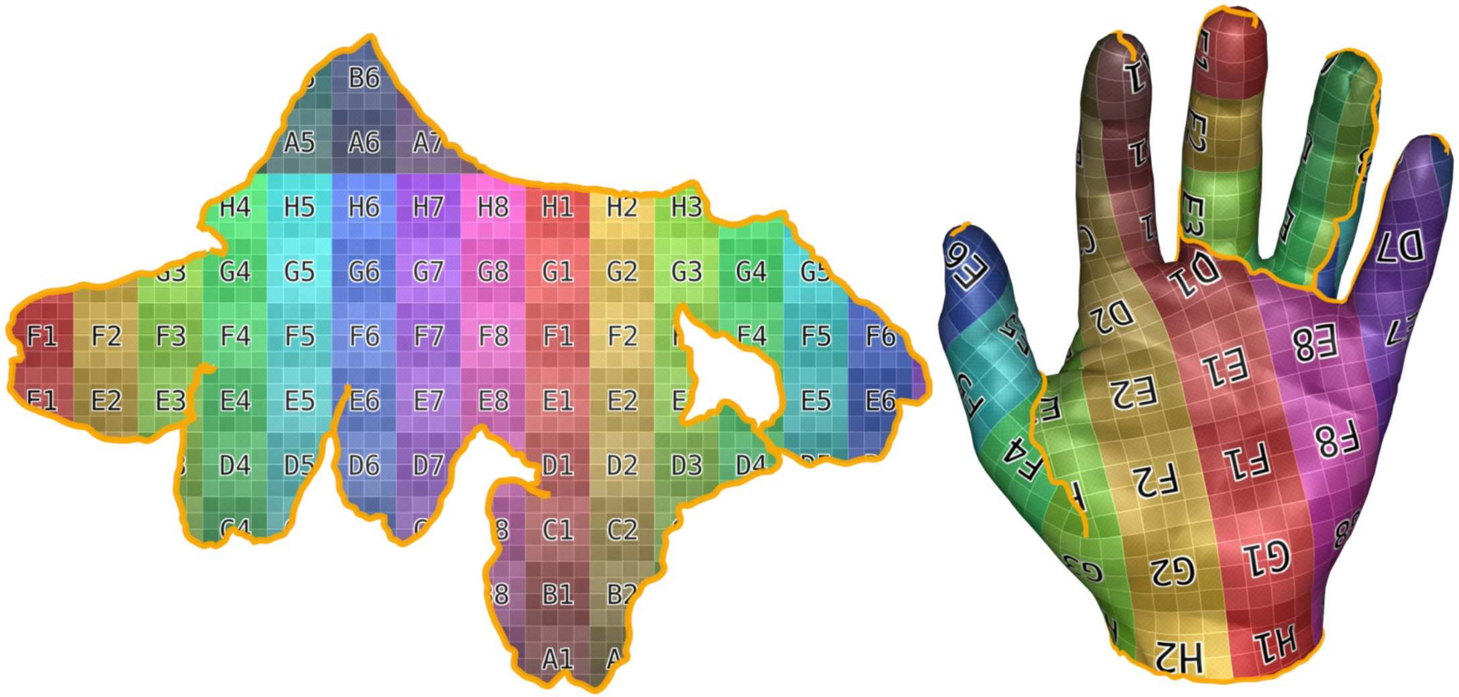
A few recent works [27, 44] have experimentally found that a heuristic sinusoidal mapping of input coordinates (called a "positional encoding") allows MLPs to represent higher frequency content.

* Authors contributed equally to this work. Preprint. Under review.

Tancik et al, NeurIPS 2020

Does our smoothness translate to 3D?

Nearby 3D points might not be close in 2D input space and thus might not benefit from the smoothness of our function!



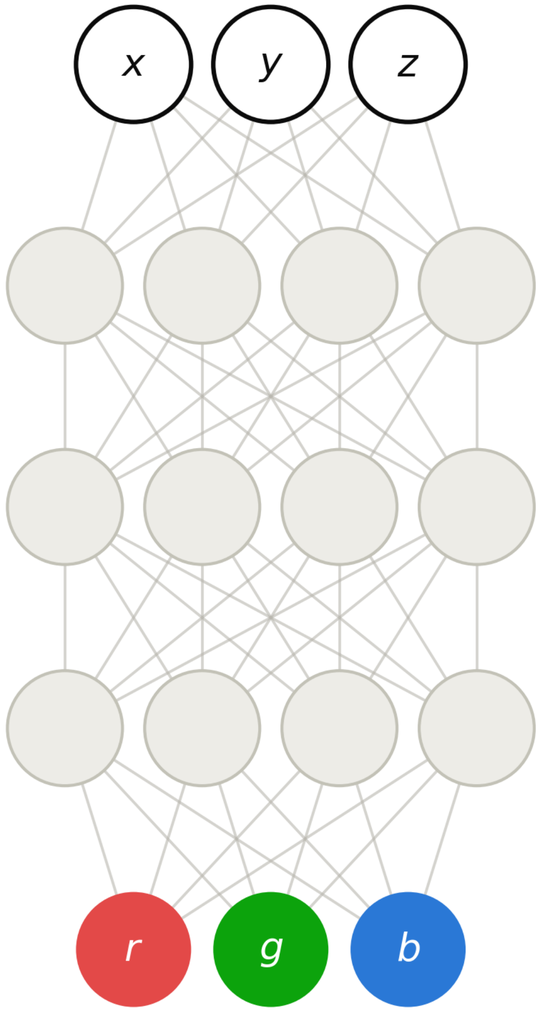
Optcuts, SIGGRAPH Asia 2018

Does our smoothness translate to 3D?

Nearby 3D points might not be close in 2D input space and thus might not benefit from the smoothness of our function!

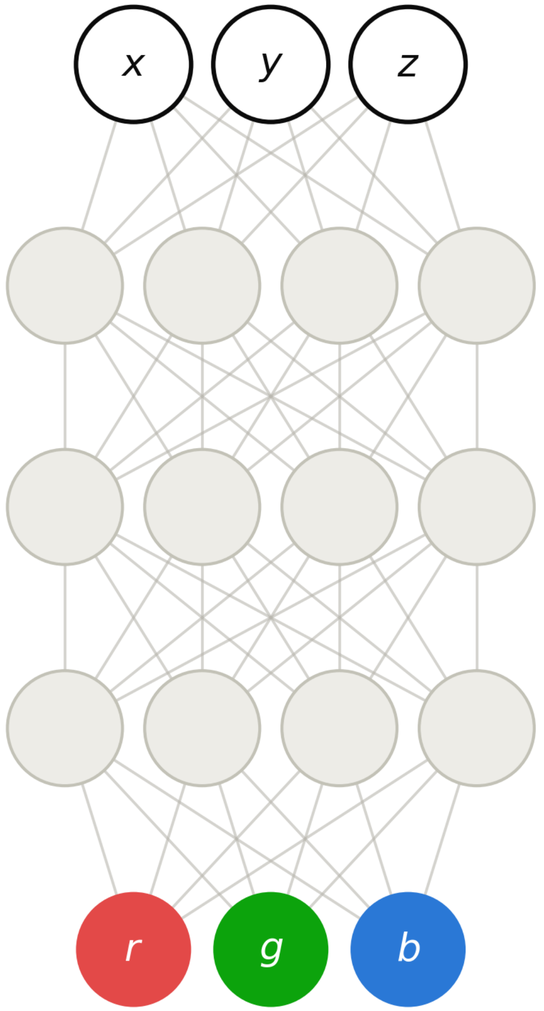
Optcuts, SIGGRAPH Asia 2018

3D Coordinate Network



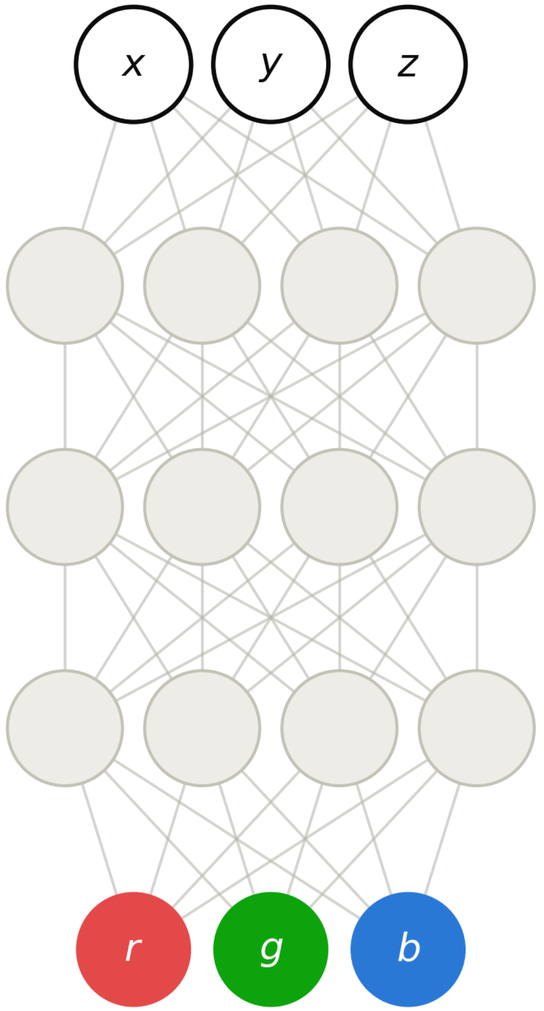
3D Coordinate Network

- Learn a **3D** coordinate network $\phi : \mathbb{R}^3 \to \mathbb{R}^3$ that maps ambient 3D space → RGB colors: $\phi(x, y, z) = (r, g, b)$



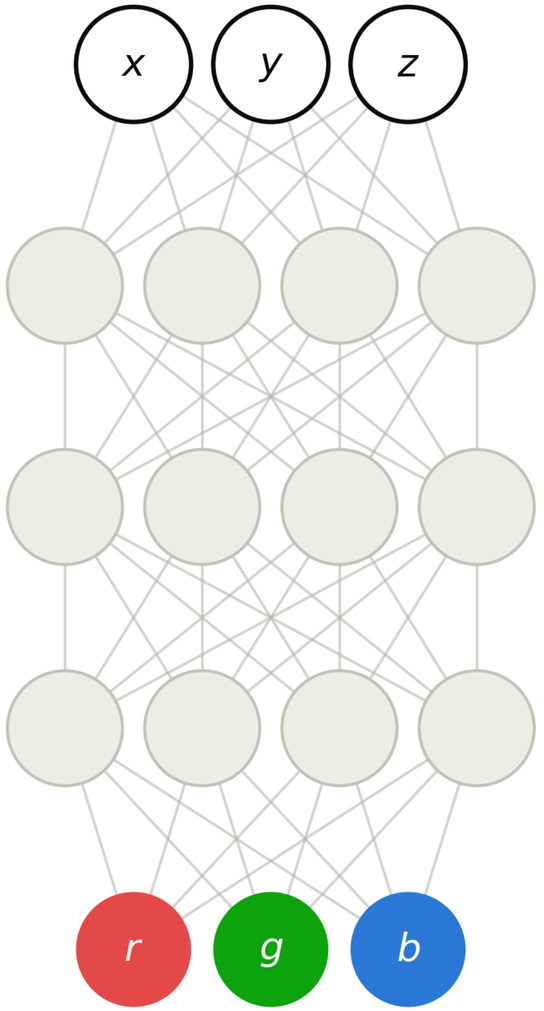
3D Coordinate Network

- Learn a **3D** coordinate network $\phi : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ that maps ambient 3D space → RGB colors: $\phi(x, y, z) = (r, g, b)$
- In practice, we only care about the function's behavior for points on the surface of the 3D object



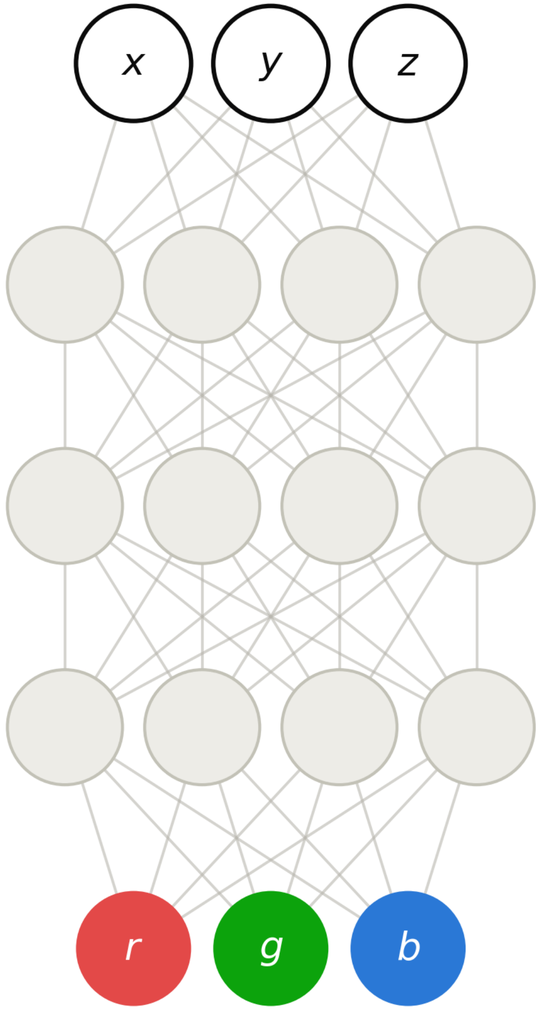
3D Coordinate Network

- Learn a **3D** coordinate network $\phi : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ that maps ambient 3D space → RGB colors: $\phi(x, y, z) = (r, g, b)$
- In practice, we only care about the function's behavior for points on the surface of the 3D object
- But how do we optimize this network? Need a way to map between texels and 3D surface points so we know where to query it…



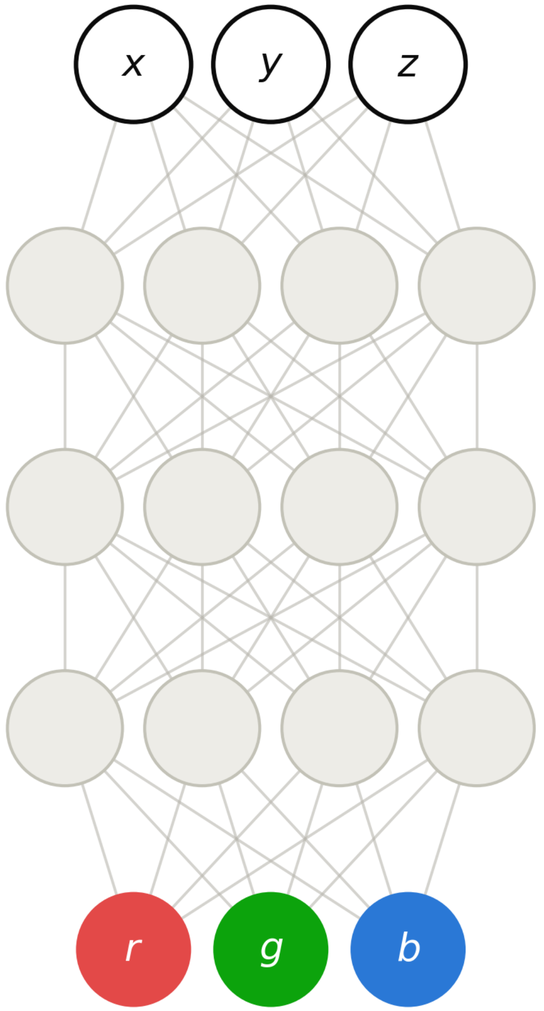
3D Coordinate Network

- Learn a **3D** coordinate network $\phi : \mathbb{R}^3 \rightarrow \mathbb{R}^3$ that maps ambient 3D space → RGB colors: $\phi(x, y, z) = (r, g, b)$
- In practice, we only care about the function's behavior for points on the surface of the 3D object
- But how do we optimize this network? Need a way to map between texels and 3D surface points so we know where to query it…
- Sounds familiar :)



3D Coordinate Network

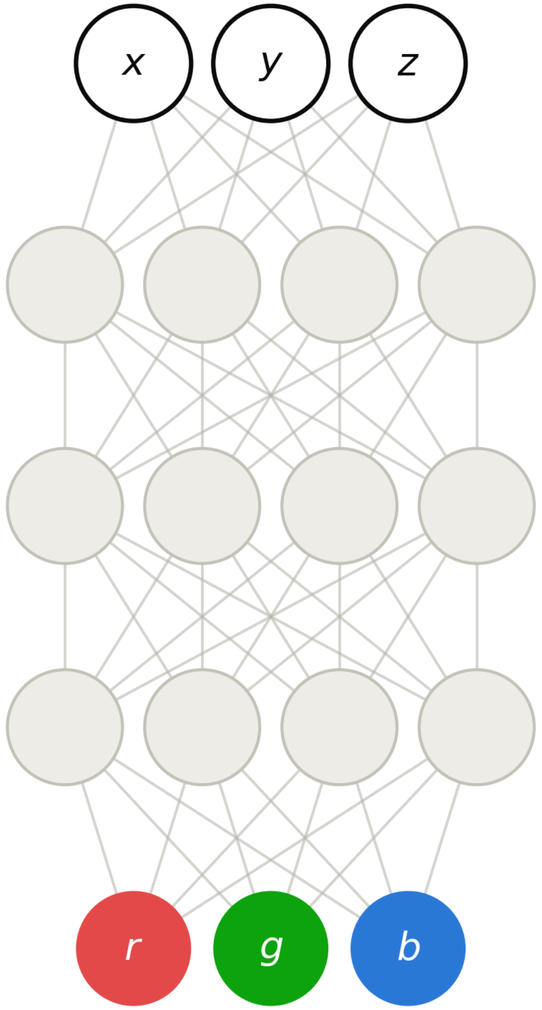
Inverse Map



3D Coordinate Network

Inverse Map

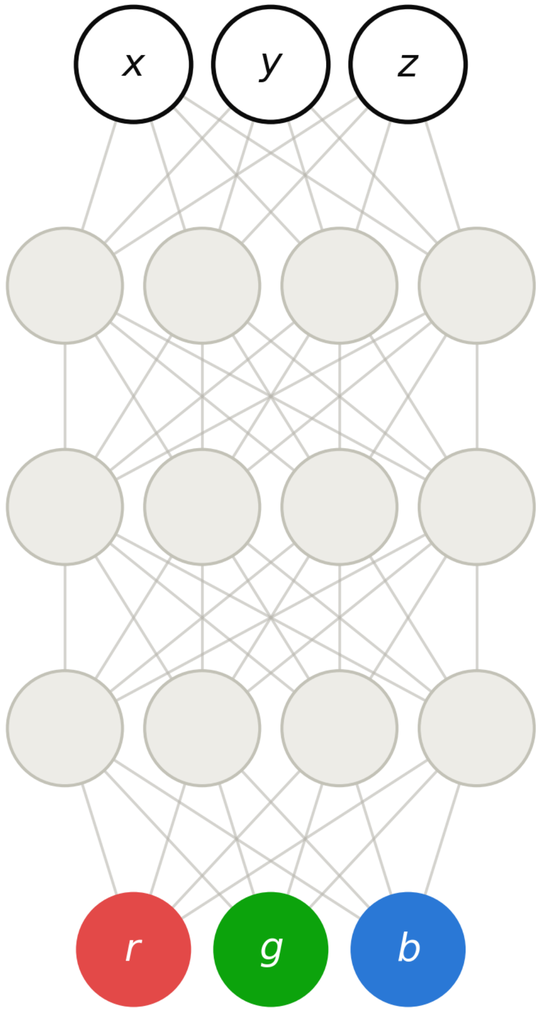
- Parameterization: 3D surface → 2D texture



3D Coordinate Network

Inverse Map

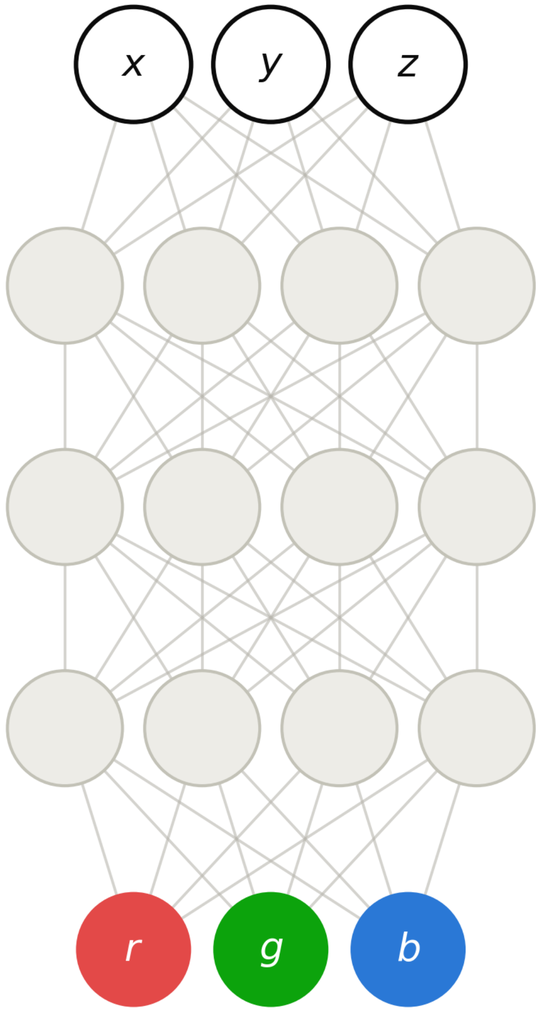
- Parameterization: 3D surface → 2D texture
- Want: 2D texture → 3D surface



3D Coordinate Network

Inverse Map

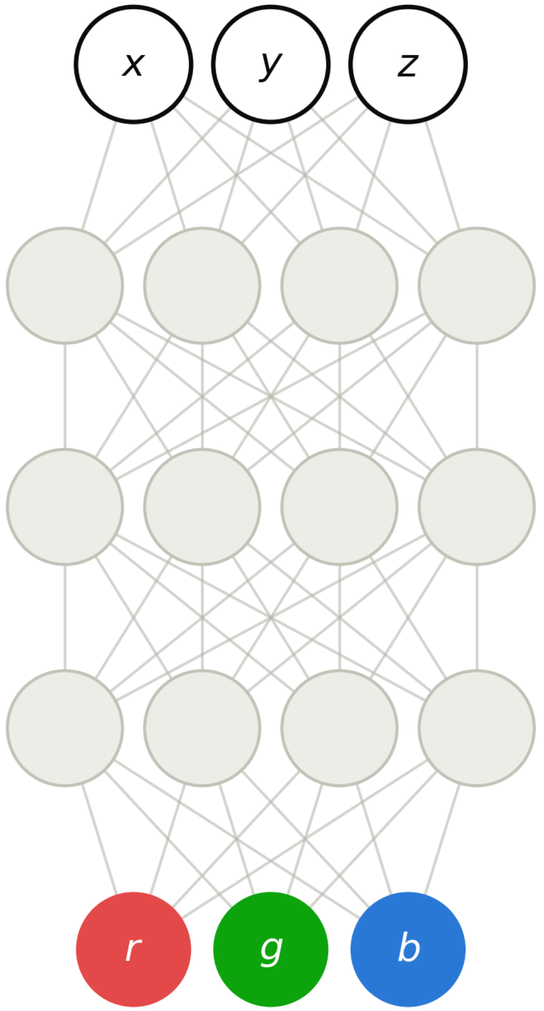
- Parameterization: 3D surface → 2D texture
- Want: 2D texture → 3D surface
- Use the inverse of the parameterization!



3D Coordinate Network

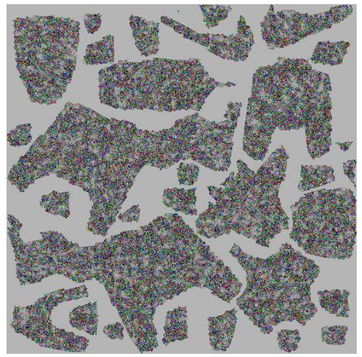
Inverse Map

- Parameterization: 3D surface → 2D texture
- Want: 2D texture → 3D surface
- Use the inverse of the parameterization!
- 2D texel point p_t → 3D surface point p_s → ϕ →(r,g,b) → store (r,g,b) at p_t



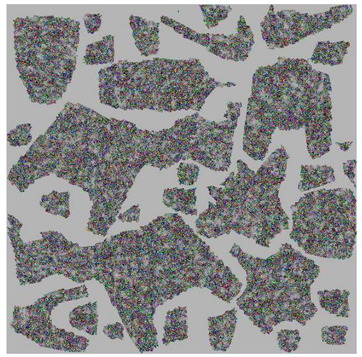
What does this give us?

Recap of the overall pipeline



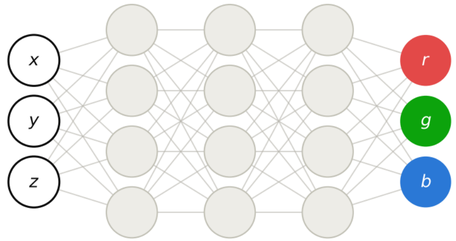
What does this give us?

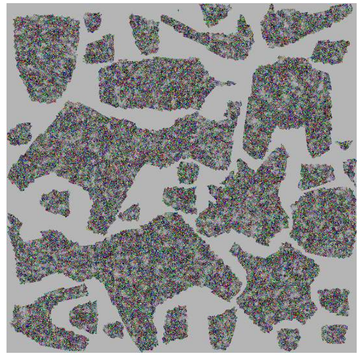
Recap of the overall pipeline



What does this give us?

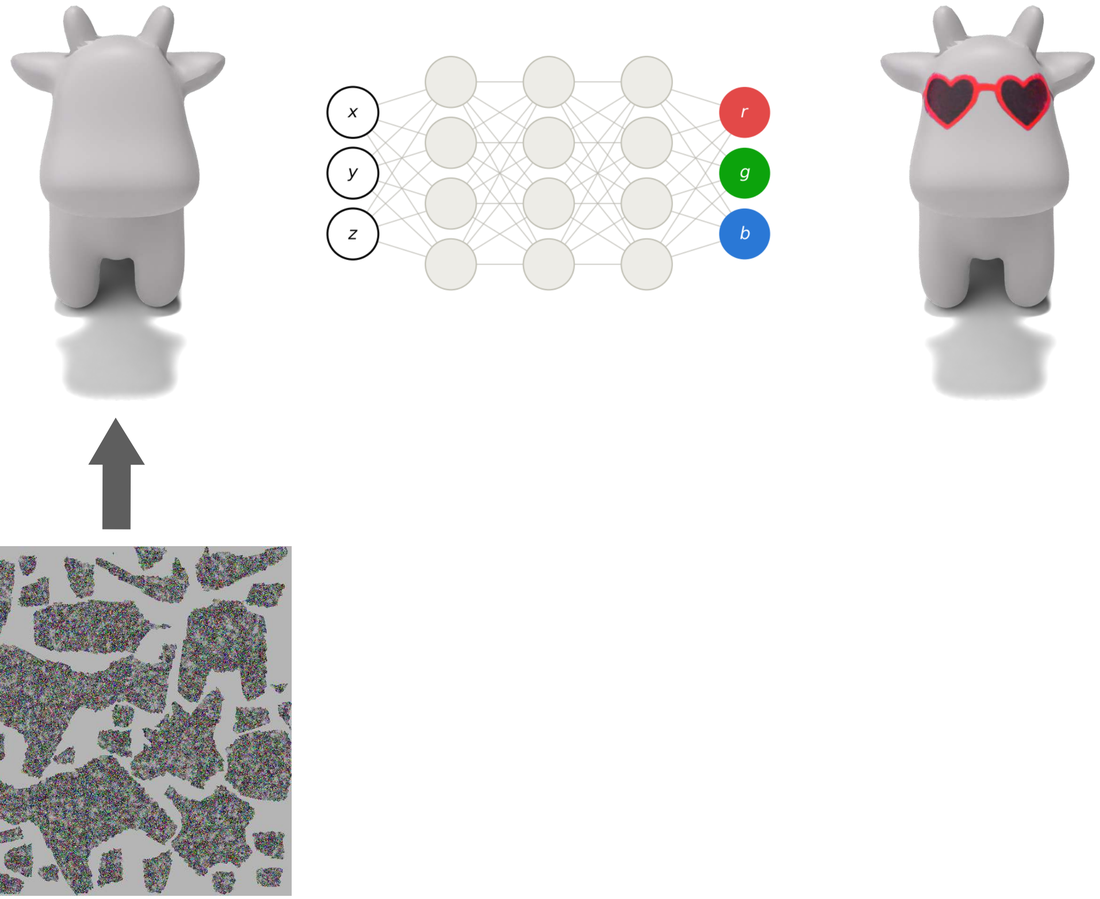
Recap of the overall pipeline





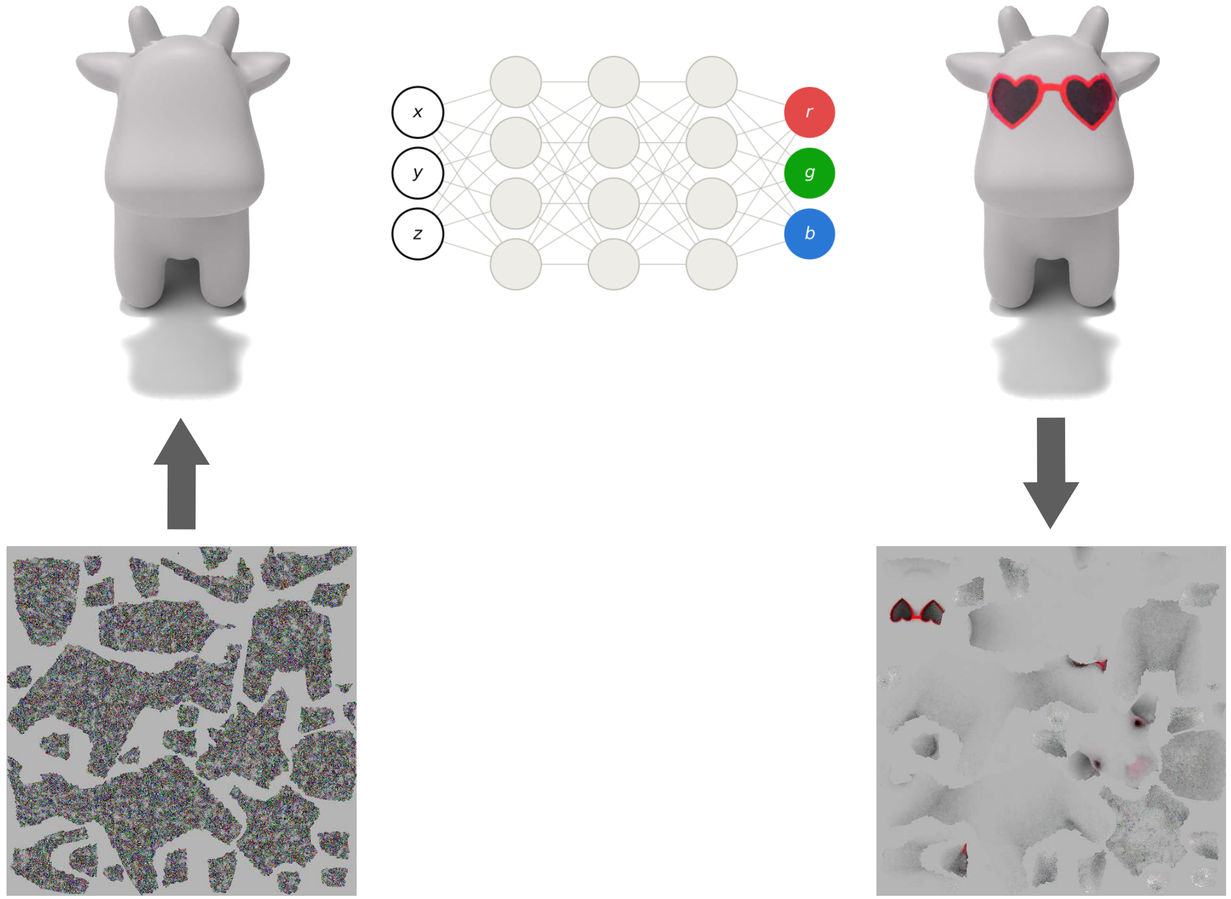
What does this give us?

Recap of the overall pipeline



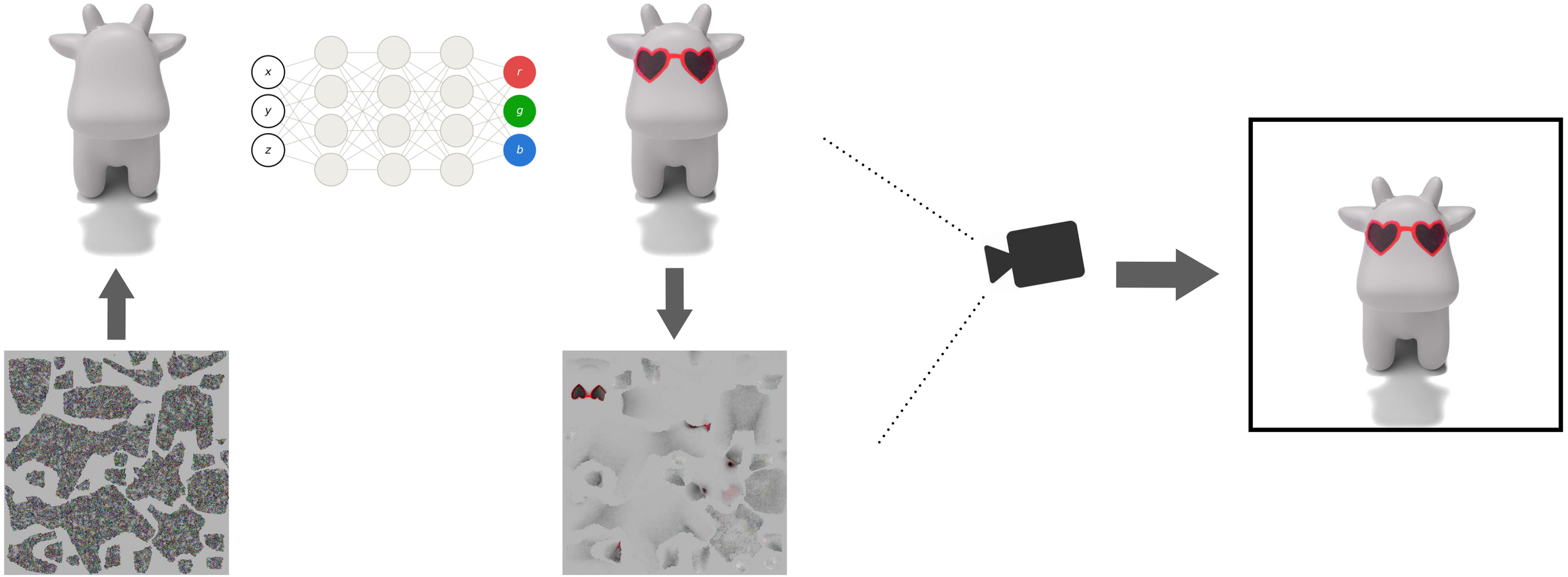
What does this give us?

Recap of the overall pipeline



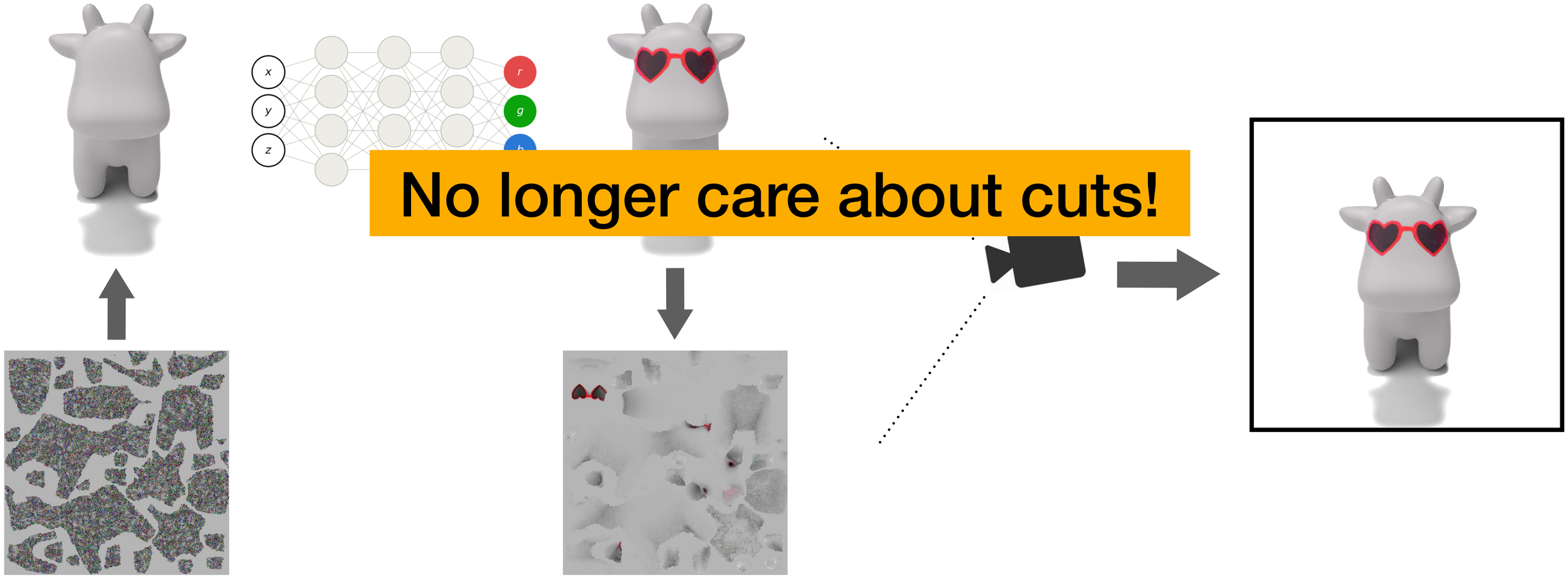
What does this give us?

Recap of the overall pipeline



What does this give us?

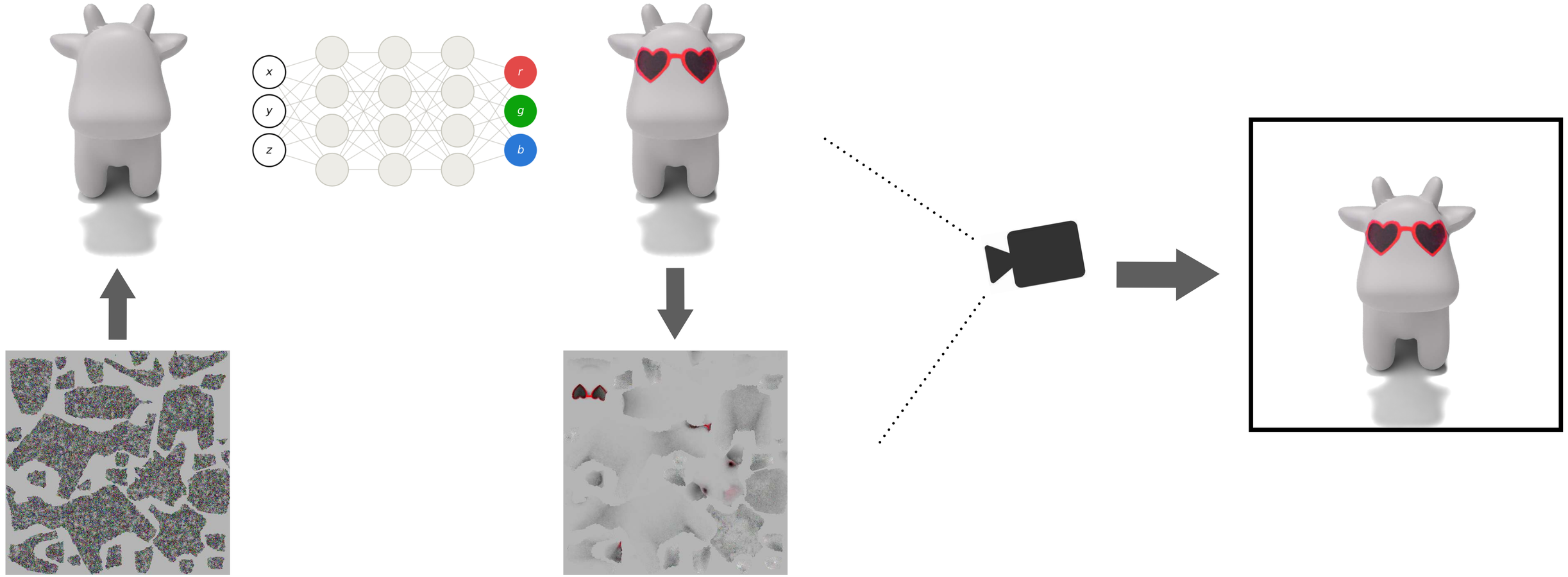
Recap of the overall pipeline



What does this give us?

Recap of the overall pipeline

3D $\phi \rightarrow$ cuts don't matter

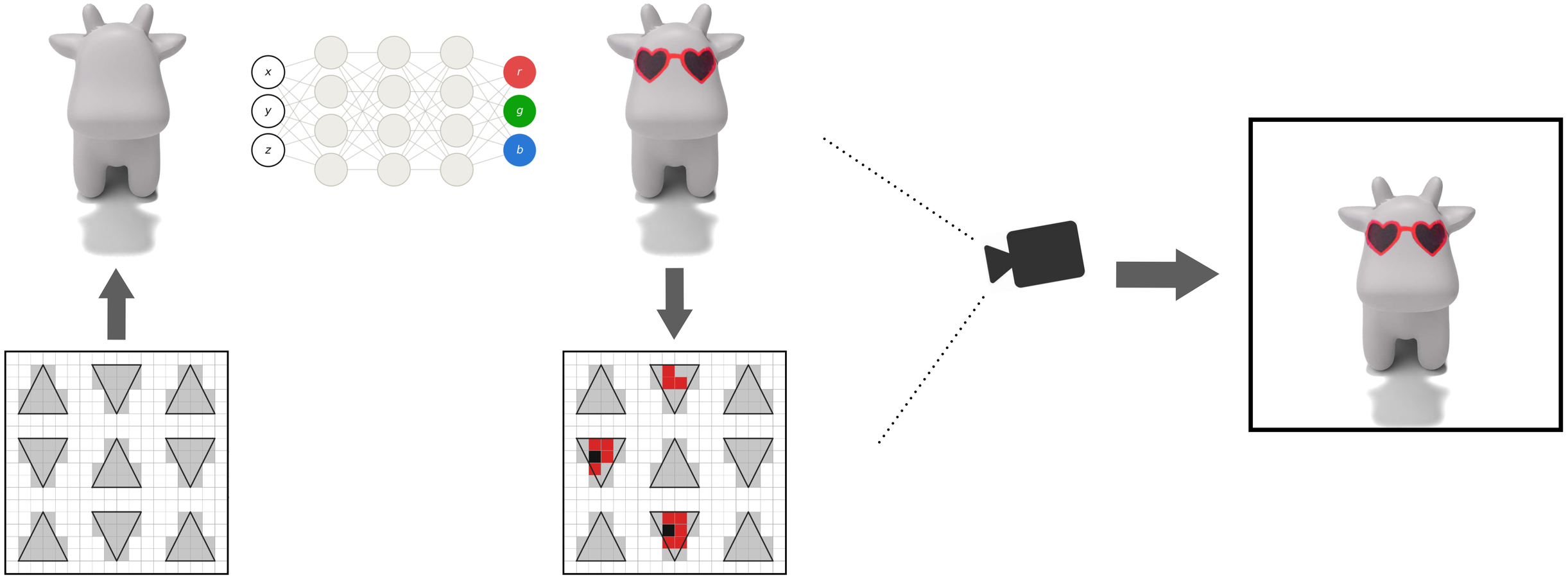


What does this give us?

Recap of the overall pipeline

3D ϕ → cuts don't matter

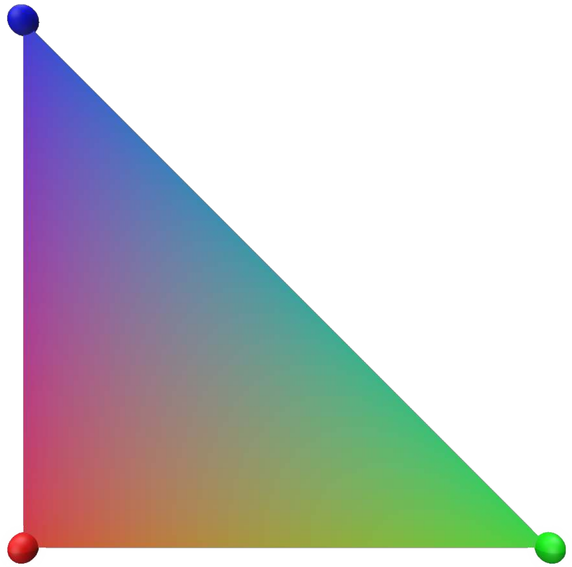
soup param → no distortion



Conclusion: Recap Mesh Coloring Methods

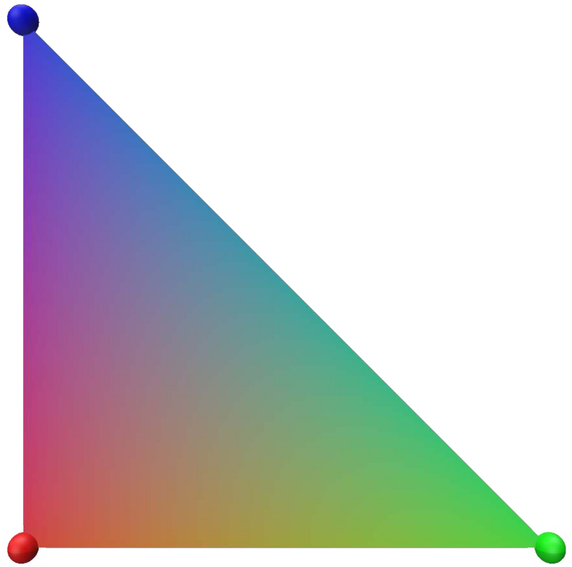
Conclusion: Recap Mesh Coloring Methods

Vertex/Face Coloring



Conclusion: Recap Mesh Coloring Methods

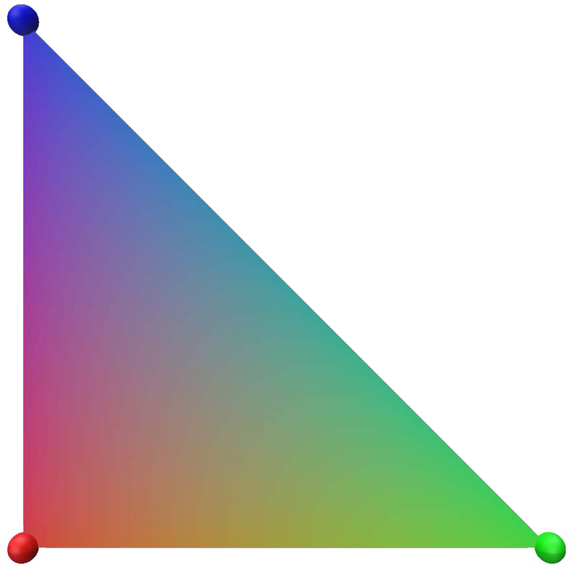
Vertex/Face Coloring



mesh resolution

Conclusion: Recap Mesh Coloring Methods

Vertex/Face Coloring

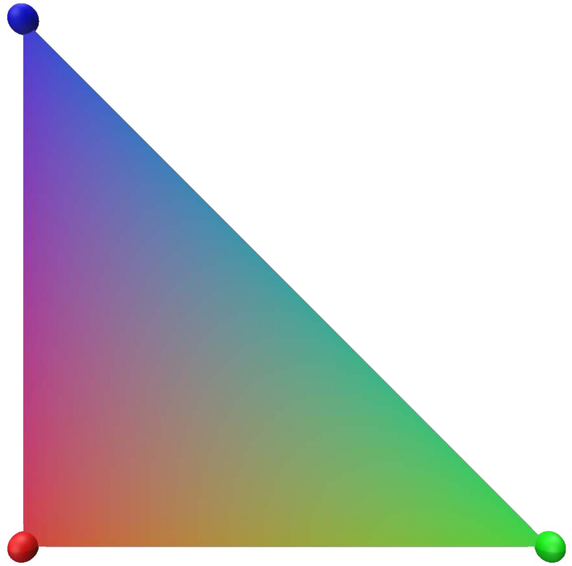


mesh resolution

parameterization

Conclusion: Recap Mesh Coloring Methods

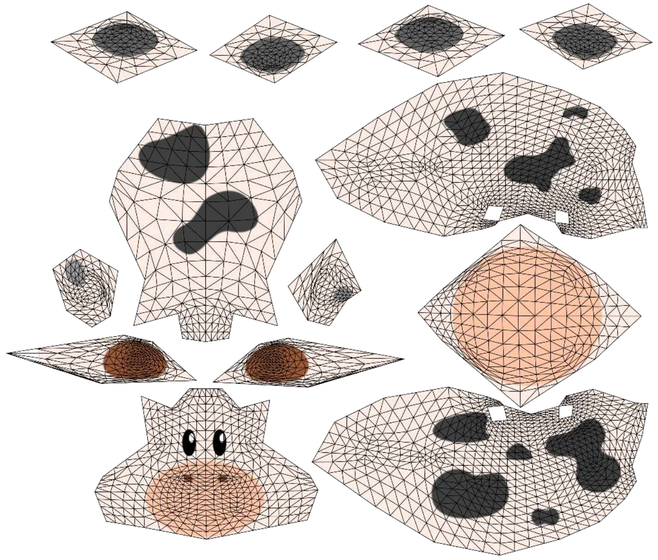
Vertex/Face Coloring



mesh resolution

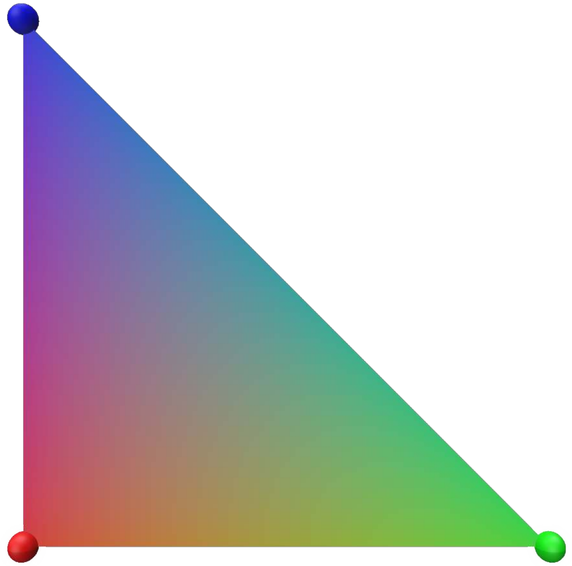
parameterization

Texture Maps



Conclusion: Recap Mesh Coloring Methods

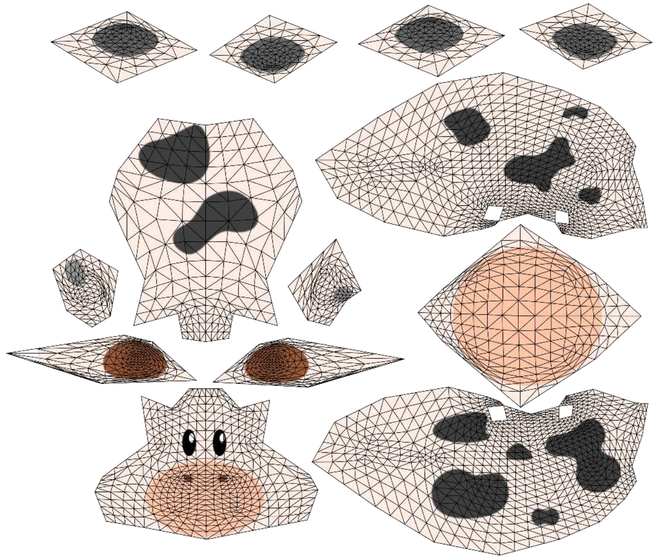
Vertex/Face Coloring



mesh resolution

parameterization

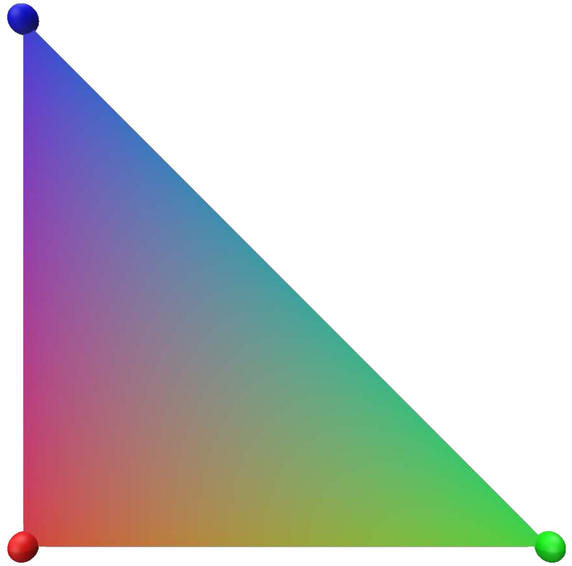
Texture Maps



mesh resolution

Conclusion: Recap Mesh Coloring Methods

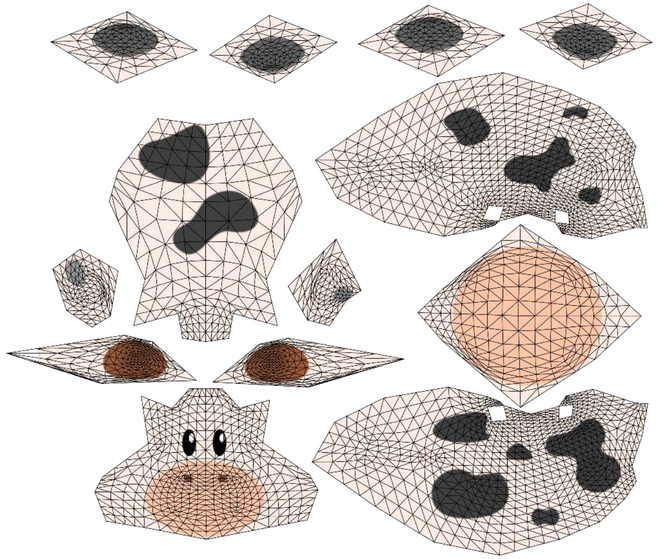
Vertex/Face Coloring



mesh resolution

parameterization

Texture Maps

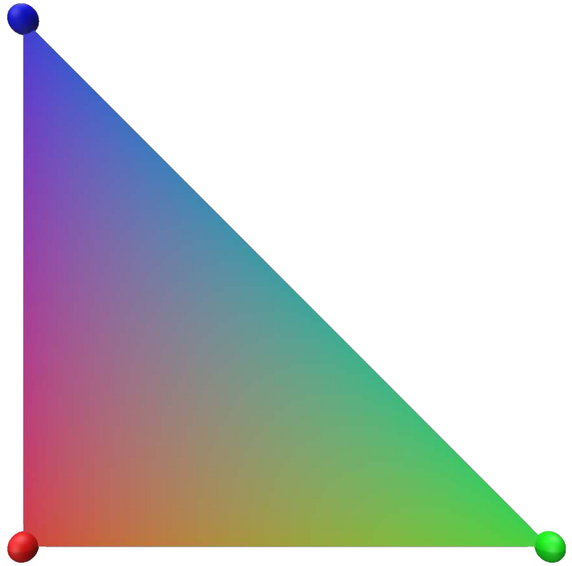


mesh resolution

parameterization

Conclusion: Recap Mesh Coloring Methods

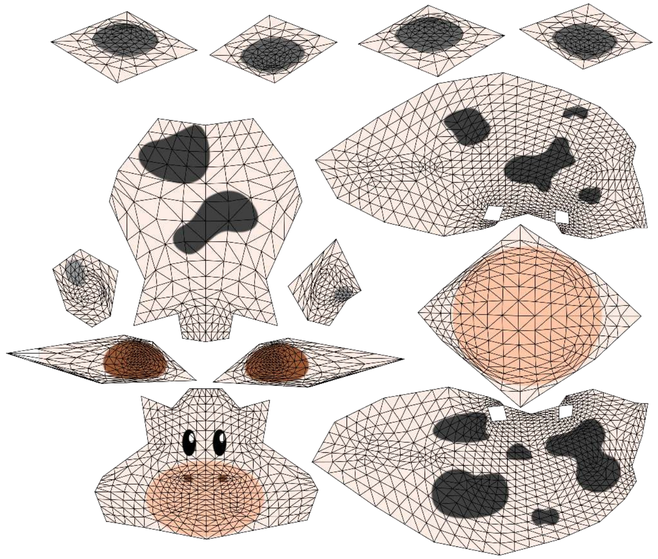
Vertex/Face Coloring



mesh resolution

parameterization

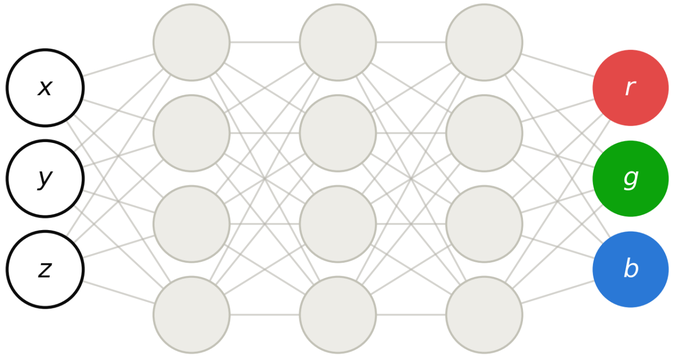
Texture Maps



mesh resolution

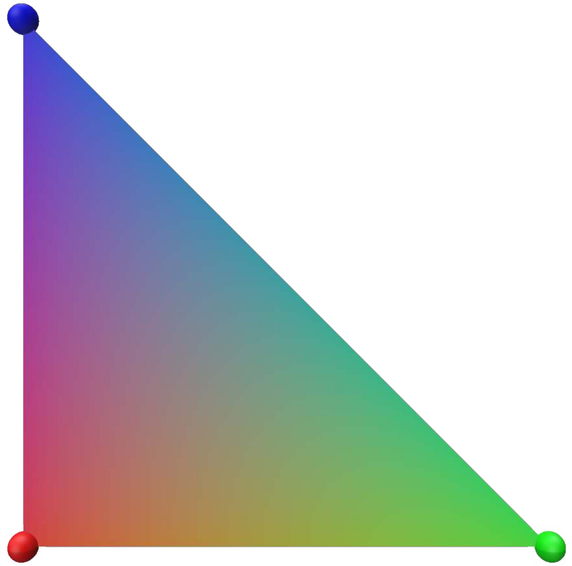
parameterization

Coordinate Networks



Conclusion: Recap Mesh Coloring Methods

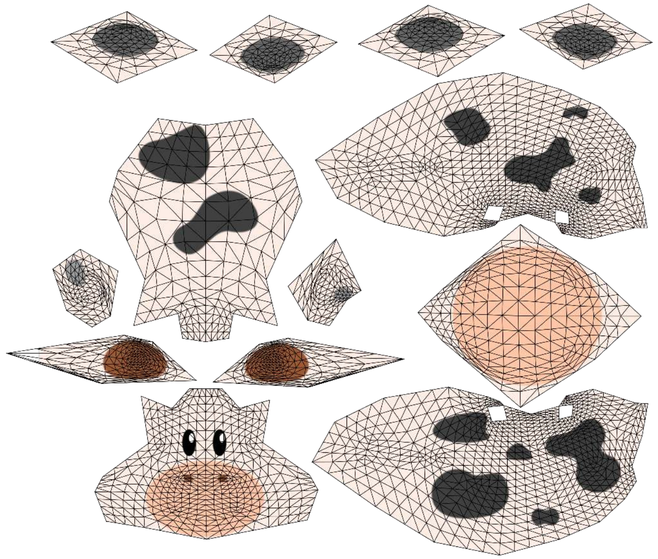
Vertex/Face Coloring



mesh resolution

parameterization

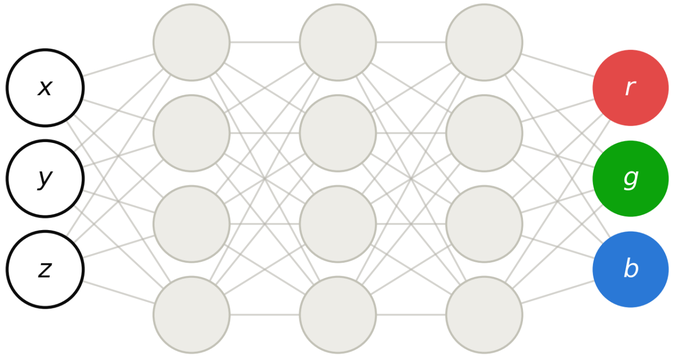
Texture Maps



mesh resolution

parameterization

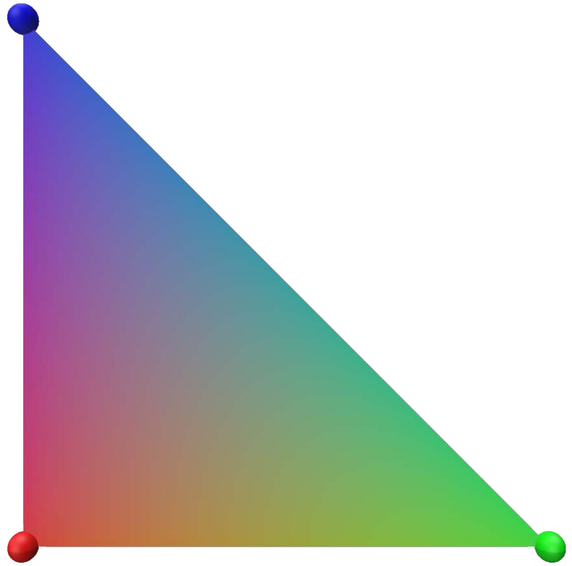
Coordinate Networks



mesh resolution

Conclusion: Recap Mesh Coloring Methods

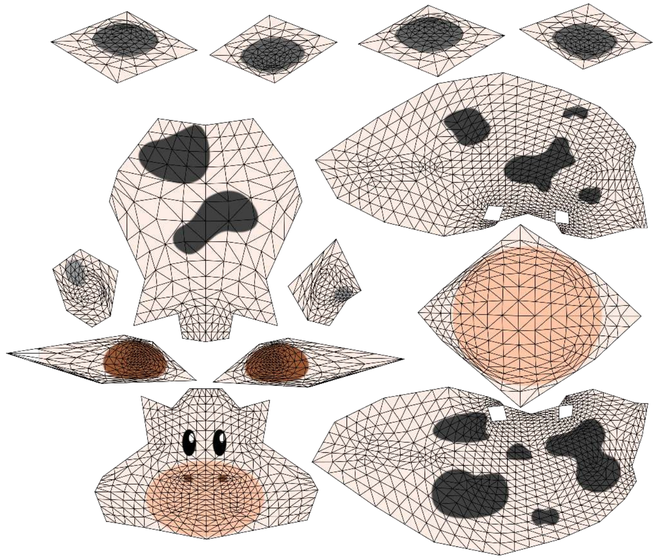
Vertex/Face Coloring



mesh resolution

parameterization

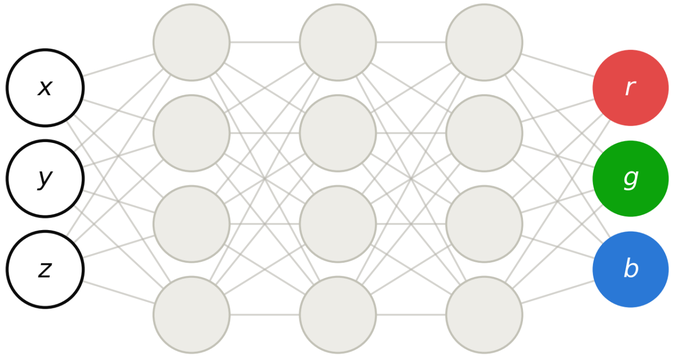
Texture Maps



mesh resolution

parameterization

Coordinate Networks



mesh resolution

parameterization

Exercises

103_pytorch — Overview of a basic optimization using PyTorch

104_texture_maps — Code a parameterization and visualize a texture map

105_gradient_descent — Write and run your own gradient descent in python

106_texture_optimization (extra credit!) — Optimize a texture map

Exercises: 103_pytorch

Goals:

- Install PyTorch
- Learn how to use basic PyTorch commands
- Optimize a simple parameter according to a loss function

Exercises: 104_texture_maps

Goals:

- Understand how to visualize texture maps in Polyscope
- Write code to compute the “trivial” parameterization that maps each triangle to the plane without overlap and visualize it

Note: for this exercise, as long as you are able to map all triangles to the unit square without overlaps, angle distortion, or area distortion beyond a constant scaler then that is a valid solution. You don’t necessarily have to follow the pseudocode / approach in the solution.

Exercises: 105_gradient_descent

Goals:

- Write your own NumPy code to implement back propagation from scratch!
- Implement both forward() and backward() methods for a few simple functions
- Fill in a training loop to chain together multiple functions to optimize a parameter
- Better understand how PyTorch auto-differentiation works

Exercises: 106_texture_optimization

Goals:

- Write code to determine the surface points that correspond to the texel centers
- Use the identified surface points to query the coordinate network responsible for learning the mapping between 3D points and colors to get the predicted surface colors
- Optimize the coordinate network that predicts surface colors to match some target images

Thank you!

Please reach out with any questions you might have or if you just want to chat!

email: ddecatur@uchicago.edu

or

Message me on slack!

Stick around for Colab Tutorial