Last updated: March 18, 2024

Contents

Introduction

0.1 Software/hardware versions used in this document — warning!

→ Blender examples were generated using at least Blender version 3.0.0.

→ In versions 3.4 and beyond, the function of the Transfer Attribute geometry node has been incorporated into the Sample Index node instead. Many of my examples here still reference the Transfer Attribute node, but be warned that a node by this name no longer exists in the most recent version of Blender!

→ There will be some difference between Mac and PC commands. The main difference vs. Mac is that ctrl becomes ⌘ (command) on Mac. I'm not sure what else changes. Maybe some keyboard shortcuts that involve keys like Alt. Throughout the document I try to describe both types of keyboard commands.

0.2 Notation

→ References to Blender objects, nodes, etc. are rendered as such: Curve, Camera, etc.

→ Menu options that are navigated to primarily using tabs are rendered as such: Properties 〉 Object Properties. I try to use this notation to indicate pages within "main" menus, i.e. the menus that are readily visible on the top and sides of the Blender window. Single tabs are still rendered point-ily: Properties〉, Object〉, etc.

→ I try to indicate secondary menu or object options as such: `Add Modifier ▸ Triangulate`. The `Add Modifier` drop-down menu is located within the Properties 〉 Modifier Properties menu page. The point of using different notation is to distinguish between options which can be reached by clicking on primary tabs/buttons in the main window vs. options/menus located within secondary menu levels.

→ If I'm trying to be concise and indicating a long chain of menu commands, then I will probably just use the "main menu" notation, i.e. Properties 〉 Modifier Properties 〉 Add Modifier 〉 Triangulate.

0.3 Acknowledgements

→ A lot of fundamental tips are from Mark Gillespie, who I pestered a lot when I first attempted to use Blender to render some research results.

→ Silvia Sellán has a good beginner's tutorial on rendering figures for graphics and geometry papers (including steps on setting up lighting and materials.) Her has links to some PBR materials and some nice paper-like materials.

→ Derek Liu has another small Blender tutorial and a bunch of scripts for rendering geometry.

0.4 Sections still being filled out!

This document primarily keeps track of rendering processes I've personally used, mostly so I don't forget how I did them. There are still sections that haven't been filled out, and topics that haven't been added yet. This document will keep growing over time as I get to them. In the meantime, check out the Blender documentation, and other people's tutorials on Stack Exchange, YouTube.

1 Rendering scalar data on surface meshes

We will take advantage of Blender's built-in support for UV maps. Normally, UV maps are used for texture mapping; the uv coordinates of each vertex are the 2d coordinates that indicate the corresponding location in the UV layout (the 2D texture image). We will store our scalar data in one of the uv coordinates; load the UV map into Blender; and extract the uv data to render as a colormap.

1. First write out the OBJ files with the uv coordinates as texture coordinates. In the OBJ file format, texture coordinates are lines that begin with "vt". Geometry-central, the geometry processing library I personally use, has a built-in function to do this: First include the I/O functions in your C++ file:

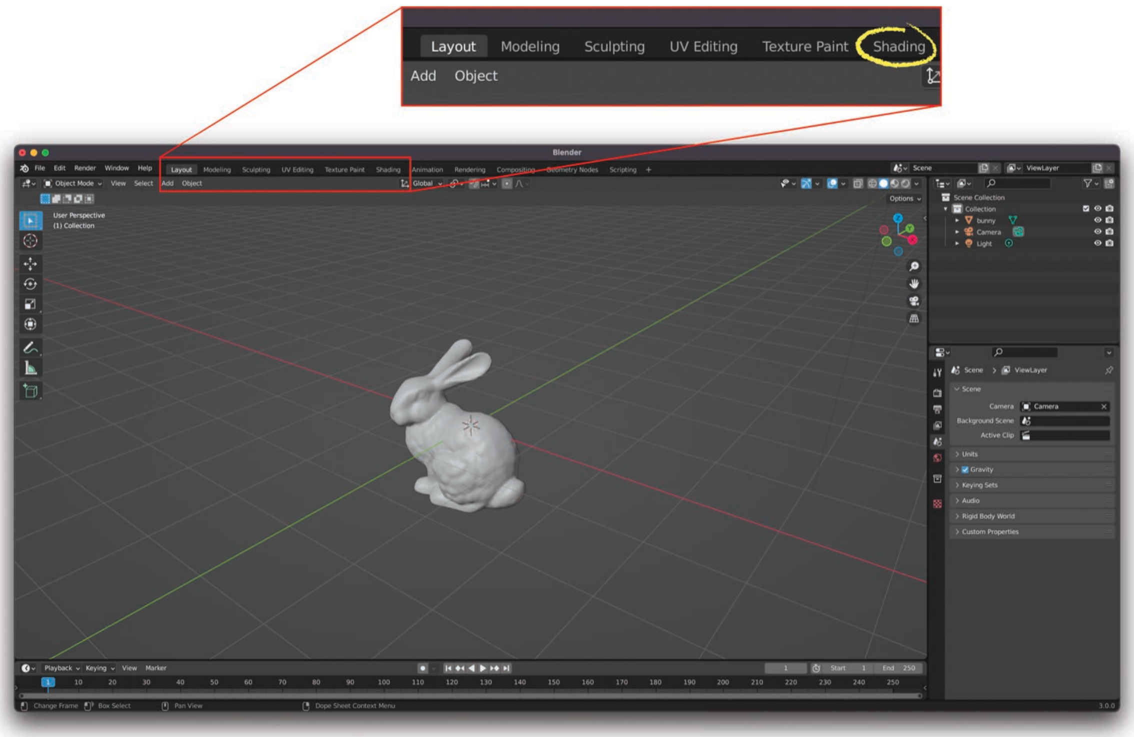
   ```
   #include "geometrycentral/surface/meshio.h"
   ```

   then use one of the `writeSurfaceMesh()` functions that will automatically write your geometry to a supported file type, along with data stored as texture coordinates; see the documentation for mesh output here.

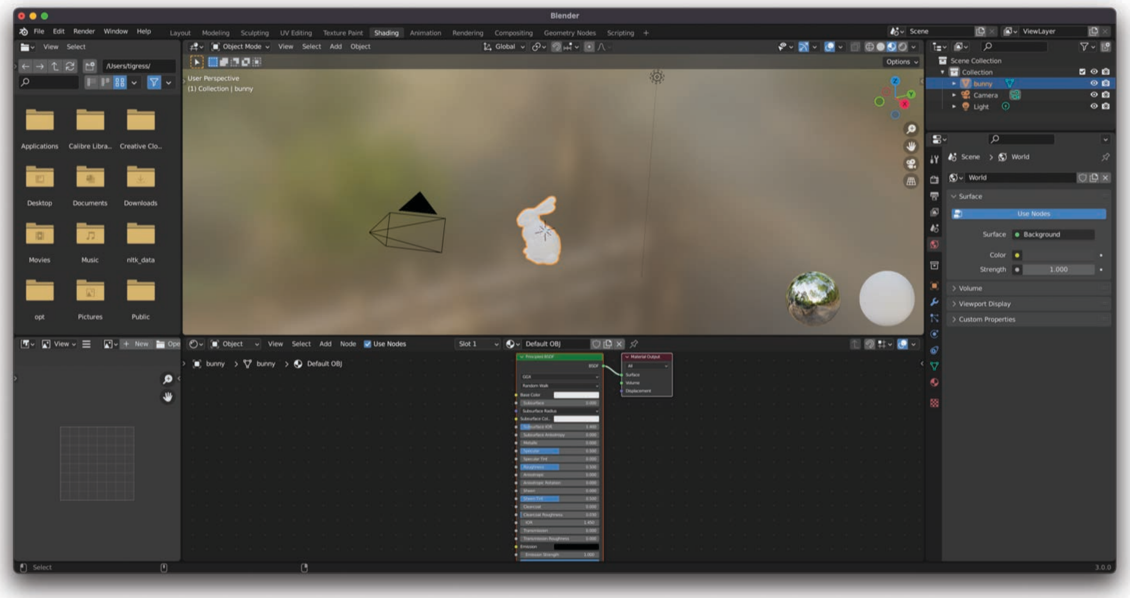
   By default, uv coordinates represent corner data; if you have vertex data, geometry-central has convenience functions called `packToParam()` for converting vertex data to corner data. (Vertex data is really just corner data, where corner values are constant around each vertex.)

   Presumably, other geometry processing libraries such as libigl have similar convenience functions for writing custom texture coordinates. Or you could just write your own I/O functions.

2. In Blender, import your OBJ file. Go to the Shading tab (shown in Figure 1.) Make sure the target mesh is selected – it should have a yellow outline around it vs. the default outline, which is a deeper orange.



(a) The location of the Shading tab.

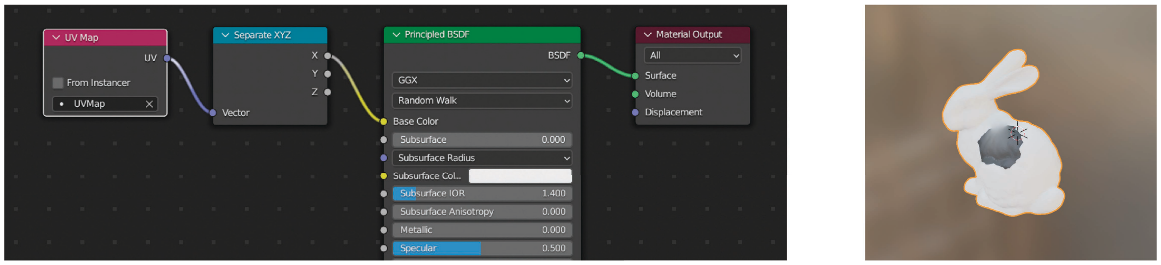


(b) The Shading editor.

Figure 1: Clicking the Shading tab (1a) brings up the Shading editor (1b.)

3. Once the target mesh is selected, you can add shading nodes. shift+A is the keyboard shortcut to `Add` a node. When you press shift+A, a menu will appear containing the different options. You can either manually click through the menu to find the node you want, or type in the search bar (the latter is faster.) Add a UV Map node, which gets the UV map associated with the mesh we imported.

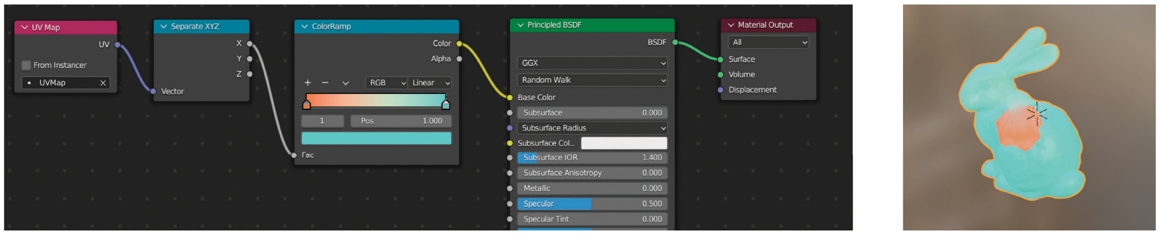
4. By default, Geometry Central stores scalar data in the first uv coordinate. Add a Separate XYZ node, and hook the UV Map node up to Separate XYZ; then hook up the `'X'` field of Separate XYZ to the `'Base color'` field of the Principled BSDF node, which feeds just the first coordinate of the UV map as a color value to the mesh (see Figure 2.) Your editor should look something like Figure 2.



Subsurface 0.000
Subsurface IOR 1.400
Subsurface Anisotropy 0.000
Metallic 0.000
Specular 0.500

Figure 2: The sequence of nodes that gets our scalar data to show on our surface mesh. (The scalar data in this example is just a simple diffusion curve on the bunny.)

5. Add a ColorRamp node after Separate XYZ and feed its color output to the base color of the mesh, to map the scalar data onto a custom color range.



1 Pos 1.000
Subsurface 0.000
Subsurface IOR 1.400
Subsurface Anisotropy 0.000
Metallic 0.000
Specular 0.500
Specular Tint 0.000

Figure 3: Customizing the colormap used to display the scalar data.

6. Currently, the mesh looks polygonal (I mean, because it is, but most of the time we want smooth shading.) In `Object` menu button in the upper left of the viewport, click `Shade smooth` for smooth shading (Figure 4.)

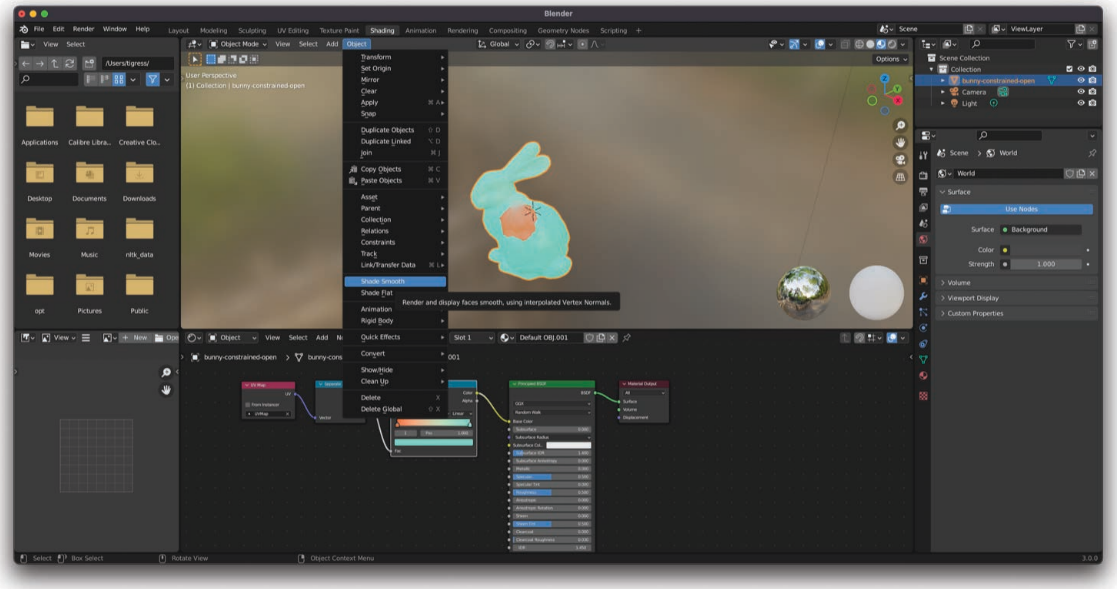


Figure 4: Click `Object`‣`Shade smooth` for smooth shading.

7. For better rendering quality, go to Properties › Render properties (in the right panel menu.) For "Render Engine", select "Cycles" instead of "Eevee". An example render is shown in Figure 5:

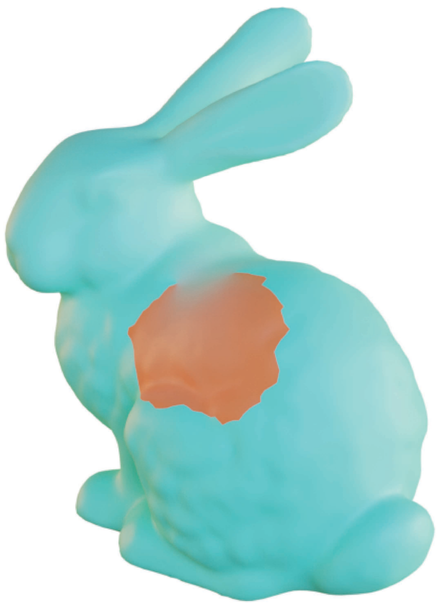


Figure 5: A render, using the "Courtyard" environment texture for lighting. I prefer not to use environment lights when rendering figures for papers, but this is just preference. See Section 6 for lighting methods.

8. For material options, see Section 9.

9. In this example, we only visualized a gradient between two values – for tips on visualizing more complicated scalar functions, see Section 8. Important for visualizing functions on surfaces: Lights and materials can interact in unexpected ways to affect the saturation and lightness of the colors in your texture. If you are planning to display colormapped

data on your surface next to a colorbar legend, you should probably add additional items to your shader to ensure accurate colors; see Section 8.

2 Rendering points

TODO: You can write an OBJ file with vertex positions, then render the points by instancing a sphere, cube, etc. at each vertex.

3 Rendering vector data

TODO: Vectors on vertices, edges, faces. Little vectors with arrows, or streamlines.

When your data gets more complicated, there is typically more information you want to specify than can be stored in an OBJ file. For example, for a vector field you want to specify (1) the 3D positions of each vector; (2) RGB colors for each vector; (3) possible more information for each vector...

A common approach to store the extra information is to co-opt the texture coordinates and normal information in an OBJ file. Personally, I find it's easier to simply store all this info as the vertex positions of a *separate* OBJ file – this approach has the benefit that you don't have to re-write your OBJ file representing your mesh every time you simply make a change in how you want the mesh to be rendered. It also seems easier to me to explicitly keep track of each type of quantity you're using to render the mesh, though of course it's possible to update the instructions that follow to take different types of input.

3.1 Vector fields

To render a vector field, typically you want to display a little arrow for sampled point in a domain. We will do this in Blender via *instancing*, which copies an object to multiple points. Once arrows have been instanced to each sample point, we can display our vector field by controlling the local rotation of each arrow. One way to do that is to write out each arrow's direction in world coordinates (meaning as a vector in $\mathbb{R}^3$), read data from this file via a script in Blender, and use the Align Euler to Vector node to align each arrow to the corresponding direction. In what follows below, I will describe the process of rendering tangent vectors on a surface mesh — though this process can also be adapted to the (simpler) process of rendering a collection of vectors in a 3D volume.

I save the vectors encoding rotation information in an OBJ file, with the vectors as vertex positions — I personally find this the most straightforward way to get collections of 3-tuples into Blender. A straightforward way to display a vector field is to have one vector per vertex or face, although for finely-tessellated meshes this will produce a vector field that is too densely sampled and makes for a poor visualization. A better approach is to use some method for Poisson Disk Sampling, which was used, for instance, in the Vector Heat Method paper — I've implemented Poisson disk sampling in the C++ geometry processing library geometry-central, though similar variants are implemented other software (MeshLab, for one.)

1. For better control of geometry in Blender, we can use Geometry Nodes. The Geometry Nodes editor can be accessed by clicking the Geometry Nodes tab (shown in Figure 6.)

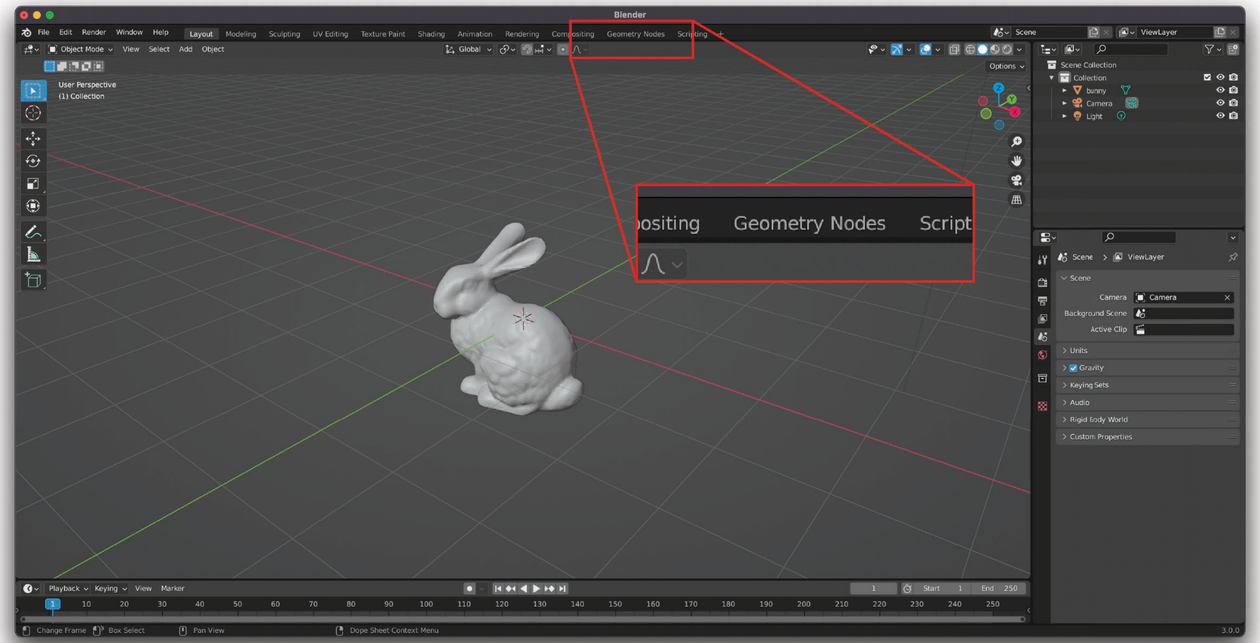


Figure 6: The location of the Geometry Nodes tab.

2. Import two items: the mesh, and the arrow object you want instanced. You probably want to choose a simplified arrow representation to make the visualization more clear, such as the compass-needle-like arrow used in the Vector Heat Method paper.

   In this example, I've imported a kitten mesh and a teardrop-needle shape similar to the one used in the Vector Heat Method (Figure 7.)

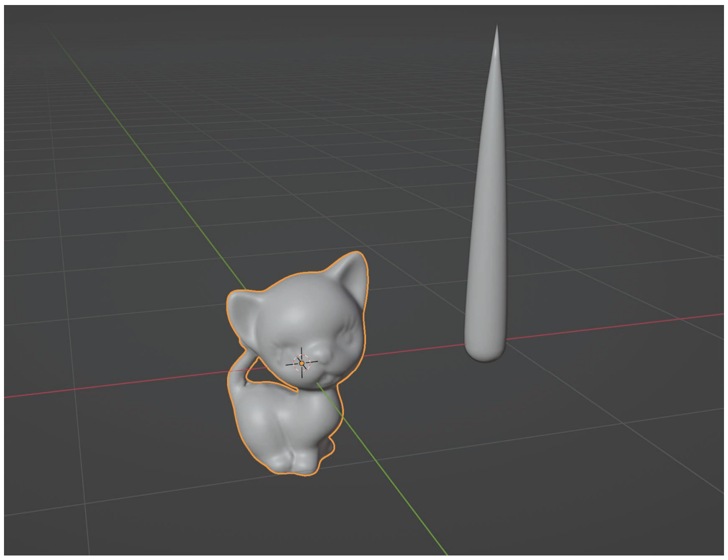


Figure 7: The surface mesh used in this example (kitten) and the mesh used to represent the arrows.

You can toggle the visibility of each object in the View Layers menu in the upper right (see Figure 8.) Toggline the eye icon controls the visibility of the item in the viewport, and the camera icon controls the item's visibility in the render. We want to set the arrow object to non-visible in the final render, and also in the viewport to avoid being distracted by it.

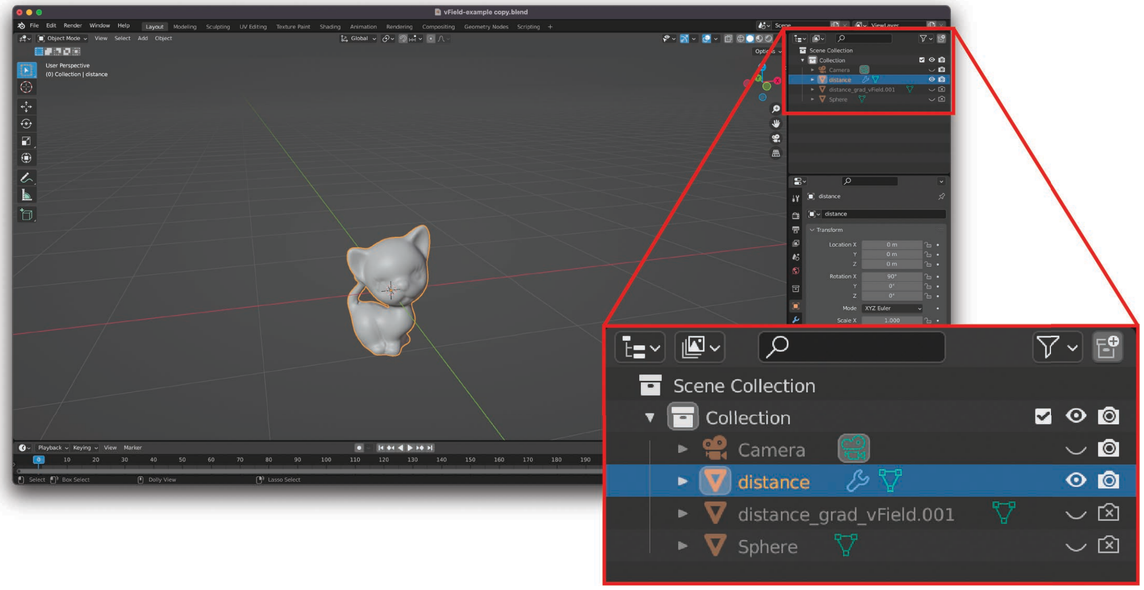


Figure 8: The default position of the View Layers menu.

3. Select the mesh representing your surface. In the Geometry Nodes editor, click "New". You will see two initial nodes: Group Input and Group Output. The "Geometry" slot of Group Input contains the currently-selected geometry, and any geometry that you want rendered must be eventually fed into Group Output.
4. Add two nodes: Mesh to Points and Instance on Points. (Beware: There is a very similarly-named node called "Instance to Points". You want "Instance **on** Points.") In the Mesh to Points, select the mesh element you want to instance to (e.g. faces.) Feed its "Points" output into the "Instances" output of Instance on Points, which will instance a yet-to-specified object to each Point. Finally connect both the instancing output and Group Input to a Join Geometry node to ensure that both the mesh and the instances get rendered. See Figure 9.

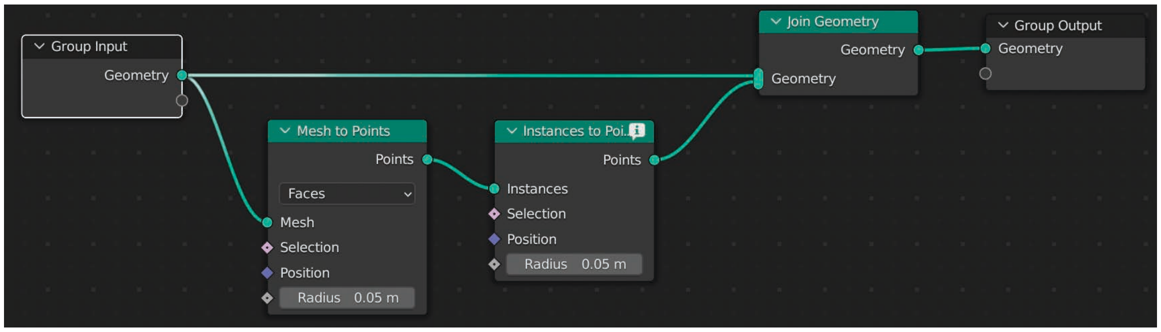


Figure 9: Preparing to instance geometry.

5. Assuming the object you want instanced is already in the viewport, add an Object Info node and select your object within this node. You can also add any other nodes modifying this object if you want; in this example, I've added a Shade smooth node so I don't have to do it via the GUI. Your geometry nodes and viewport should now look something like this:

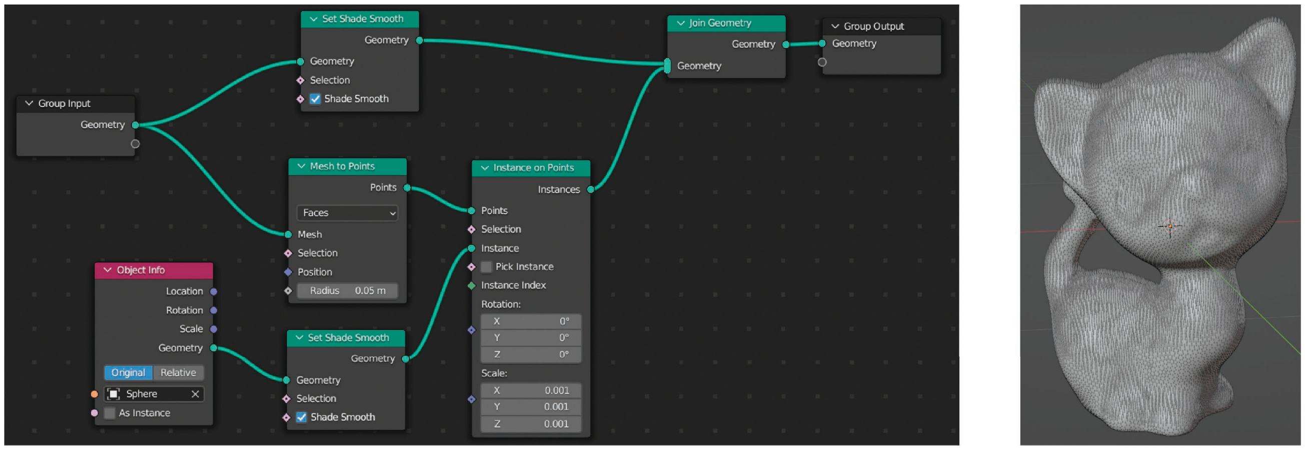


Figure 10: Instancing the geometry. The arrow object is now instanced to every face in the kitten mesh. Their orientations haven't been set yet, so they're aligned to an arbitrary direction right now.

Since the arrow-needle object was so large to begin with, I scaled all instances down via the "Scale" fields in Instance on Points.

6. We could use a Python script to load in the rotation data – and more generally, use a Python script to do all the Geometry Node manipulation – but for a single example, it's fine to avoid scripting.

   In this example, I've saved the rotation data for each face vector in a separate OBJ file, where the vectors are encoded as vertex positions. This is just an OBJ file with only vertex positions; there's no need to define a face, line, etc., and in fact adding extra connectivity info will slow Blender down a huge amount.

   See Figure 11 for the node configuration.

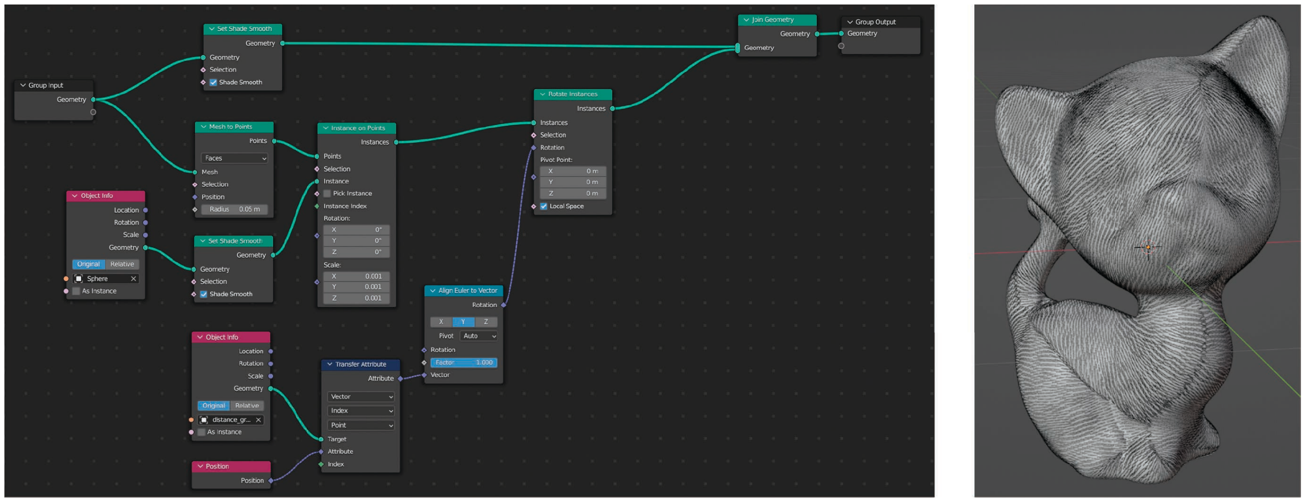


Figure 11: Assigning directions to each vector. Now it's starting to look like our vector field!

Import the OBJ file containing the rotation info. Add the following nodes: Object Info, Position, Transfer Attribute. In Object Info, select the object corresponding to the rotation info. Feed this node and Position into Transfer Attribute, and set the data type to "Vector", mapping type to "Index", and domain to "Point". This transfers the position of vertices from the rotation object's geometry, which should have the same number of points as there are instances of the arrow geometry. (Note: In future versions of Blender (3.4+), the function of the Transfer Attribute node has been incorporated into the Sample Index node instead.)

Add the nodes Align Euler to Vector and Rotate Instances. These two nodes in combination takes in the tangent vectors encoded by our fake rotation object, and produces a rotation that is aligned to the given direction, and aligns each instance to this direction. The “Axis” to align (X, Y, or Z) depends on the local axes of the original arrow object; in this example, it happens to be Y.

7. We also want to scale the arrows depending on their magnitude. Add a Vector Math node, and set it to “Length”. Feed in the attribute from Transfer Attribute into Length, and feed this length value into a Scale Instances node as one of the components in “Scale” (Figure 12.)

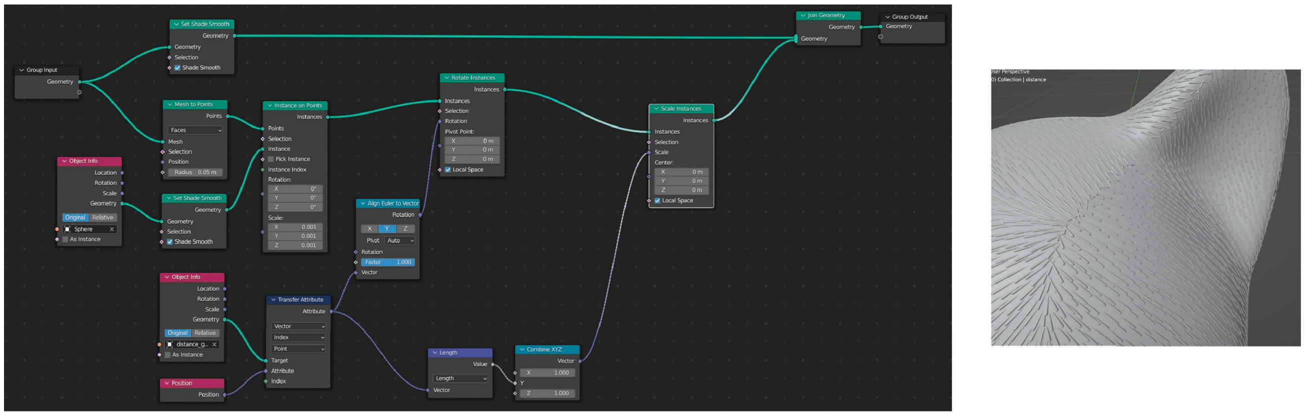


Figure 12: Scaling the instanced arrows according to magnitude. In this example, not much changes because the vector field represents the gradient of an approximate distance field, so the magnitude is more or less unit everywhere; you would really have to zoom in to notice any changes. On the other hand, depending on the range of magnitudes in your vector field, you may need to impose another uniform scaling to make sure the arrows end up visible.

8. In this example, the arrow object was radially symmetric. However, if our arrow object was flattened, then simply aligning its longitudinal direction to the direction we input doesn’t guarantee that the arrow also “lies flat” on the surface the way we want:

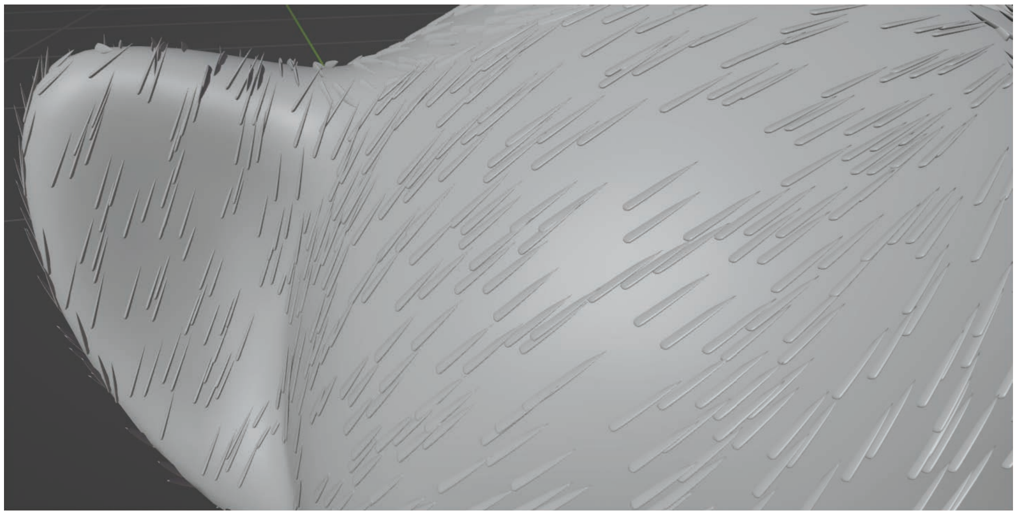


Figure 13: For a non-symmetric object, the rotation that aligns one dimension to a given direction is non-unique. In particular, specifying that the longitudinal axis of a flattened arrow object should align with the given direction doesn’t guarantee that the arrow lies flat along the surface – this is particularly apparent on the ear of the kitten.

The solution is to add another Align Euler to Vector node; this time, we want to align one of the transverse axes of the arrow to align with the normal of the surface. To get the normal of the surface, we use a Transfer Attribute node like before, feeding in the surface geometry as input and a Normal node. Again, make sure to select "Vector" as the data type in Transfer Attribute. In this case, selecting "Nearest Face Interpolated" produces the same result as selecting "Index" and "Face" as the mapping and domain types, respectively. Finally, select the appropriate local axis of the arrow object to align in Align Euler to Vector, and feed in its output rotation into the input rotation of the original Align Euler to Vector node we used to rotate our arrows.

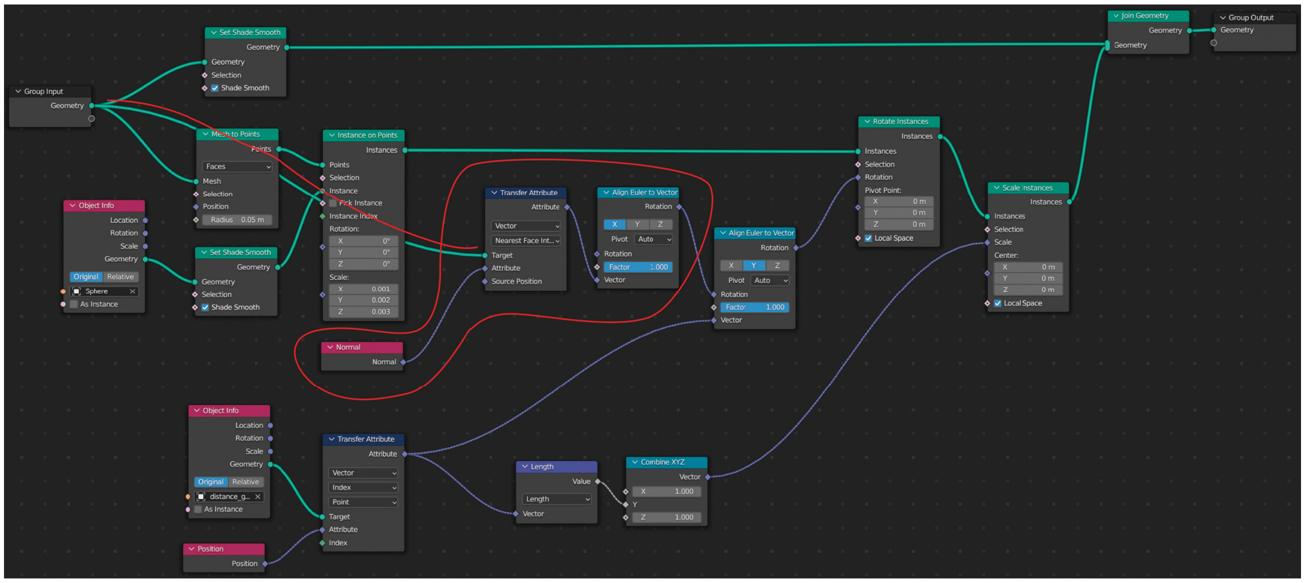


Figure 14: Adding nodes to additionally align our arrows to lie flat along the surface. New nodes and connections added in this step are indicated in red.

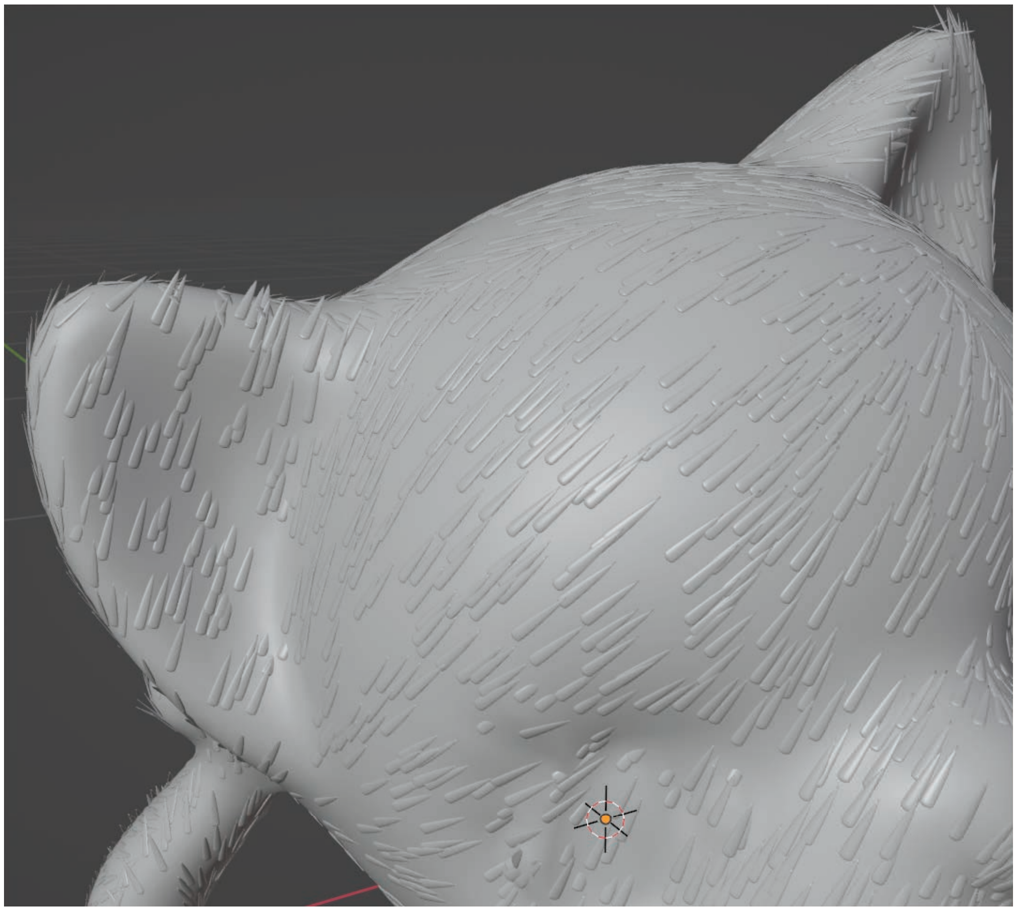
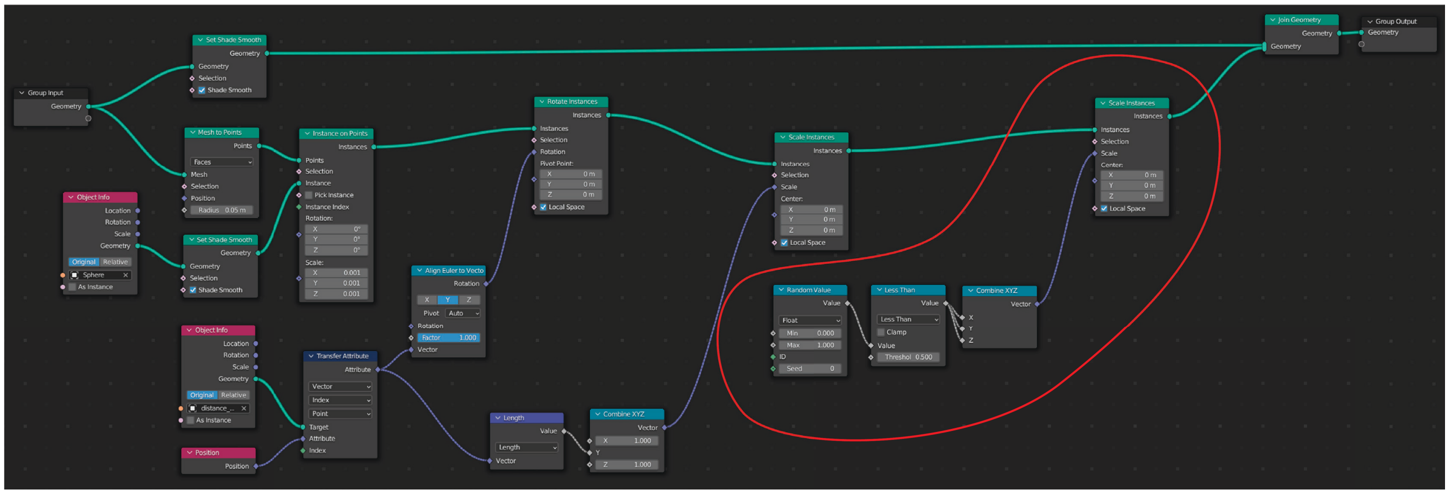
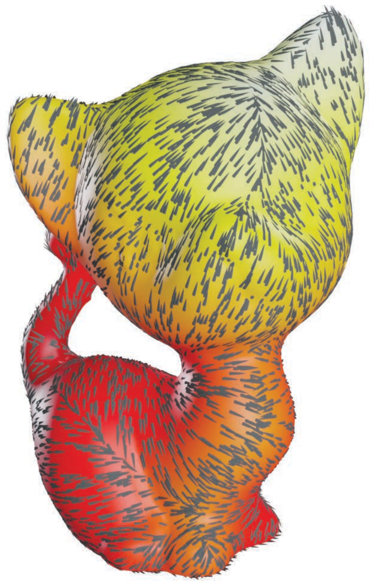


Figure 15: Now our flattened arrows lie flat along the surface!

9. Currently there are too many arrows, and all the arrows are too tiny to see well. Also, the arrows are somewhat clumped. The vector field visualization would be better if the sampling rate was less than one vector per face. We *could* cull arrows by randomly selecting a proportion of them and setting their lengths to zero, but this provides an uneven sampling:



(a) The updated node configuration for randomly culling instances, in case you wanted to do something similar. New nodes added in this step are indicated in red.

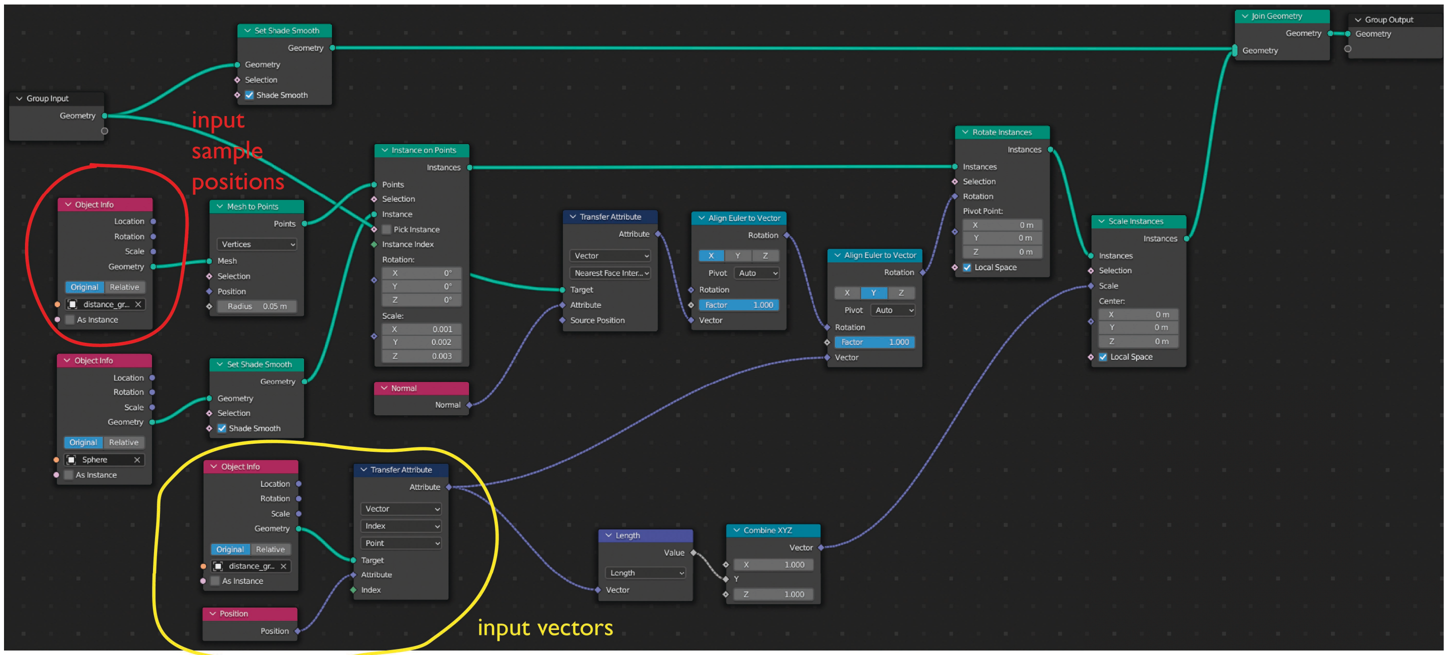


(b) The result, which doesn't look great.

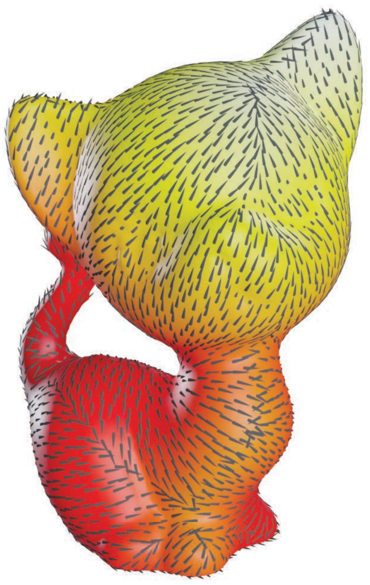
Figure 16: Randomly culling arrows provides an uneven sampling. Do NOT do this.

A better method is to use something like Poisson Disk Sampling; the Vector Heat Method paper used this method by Bridson. It's easiest to implement the sampling algorithm in the code you used to generate the file containing the vector information, so probably not via Blender. As of writing this section (July 2022), I've added a `PoissonDiskSampler` class to geometry-central, though implementations of Poisson disk sampling for surface meshes also exist in MeshLab, among other software.

10. Now I've implemented the Poisson disk sampling algorithm mentioned in the previous step, and re-written the OBJ file containing rotation information. Specifically, I've written one OBJ file that contains the locations at which vectors are sampled (as vertex positions), and a second OBJ file that contains the corresponding vectors at those sampled locations (also as vertex positions.) Ideally I would have stored the vectors as vertex normals within the first OBJ file, but I had trouble getting Blender to recognize them. Now our vector field looks like this:



(a) The updated node configuration.



(b)

Figure 17: A Poisson-sampled surface – much better!

11. As an optional visual tweak, you may want to adjust the darkness of your arrows to be inversely proportional to the darkness of the underlying mesh, for optimal contrast between the arrows and the surface mesh.

    One way to do this is save yet another file containing the scalar function evaluated at each vector of your vector field, and import this data via nodes. Again, this may be a non-optimal way to import attribute data; scripting may be a much better "one-shot" way to import data (you could write a Python script that reads any data you'd like, in any way you'd like, from any file(s)), but so far this method has been reasonably efficient for me.

The following commands assume Blender version 3.2.1 or higher. Go to the Geometry Shading editor where you have all the geometry nodes for your vector field set up. Add a Store Named Attribute node. Feed the last of nodes controlling the instanced geometry into the "Value" field of Store Named Attribute. Input the scalar data you want to store on the instances into Store Named Attribute, and set the attribute domain to "Instance", and set the data type appropriately (in this case, float.) Give a name to your attribute; here I've just named it "col". Node configurations are shown in Fig. 18.

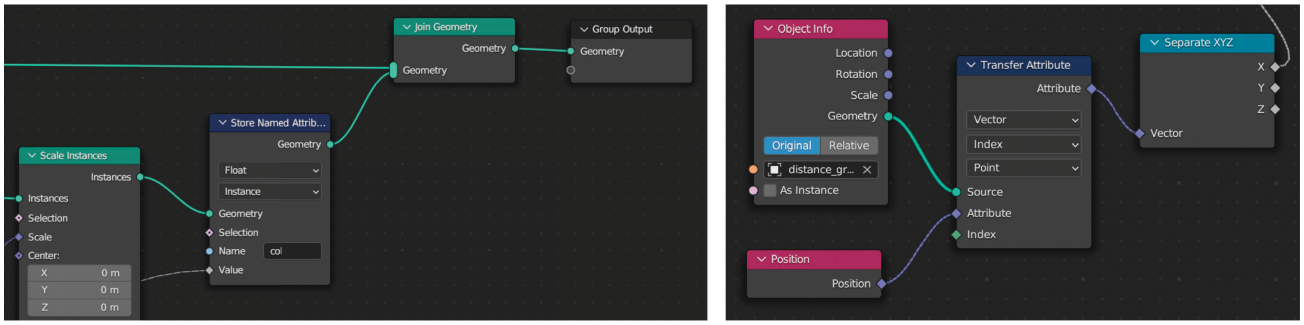


Figure 18: Left: Adding the nodes that will allow us to define an attribute on instanced geometry. Right: The nodes used to extract the scalar field values at the instance samples; in this example, I've stored the values as the x-coordinate of vertex positions in a separate OBJ file. *Note:* In newer versions of Blender (3.4+), the function of the Transfer Attribute node has been incorporated into the Sample Index node instead.

Go to the Shading editor and select the arrow object used for instancing. Add an Attribute node, and specify which attribute you want to access – in this example, I called the attribute "col". Hook this up to the same colormap you used to visualize the scalar function on the surface – now at each instance, we have the color of the mesh directly underneath the instance. We want to use this color to determine the gray level that optimally contrasts with this color. This corresponds to computing the relative luminance (contrast ratio) between colors.

Instead of computing luminance ourselves, we can simply use the RGB to BW node. I've added nodes to compute $(1 - \text{luminance})$, and also re-mapped the range so that there's less white in this particular example. On the other hand, it's probably easier to use an Invert node instead. The updated Shader node configuration is shown in Figure 19. The arrows will be rendered in shades of gray, using a lighter shade of gray in areas with darker colors, and vice versa.

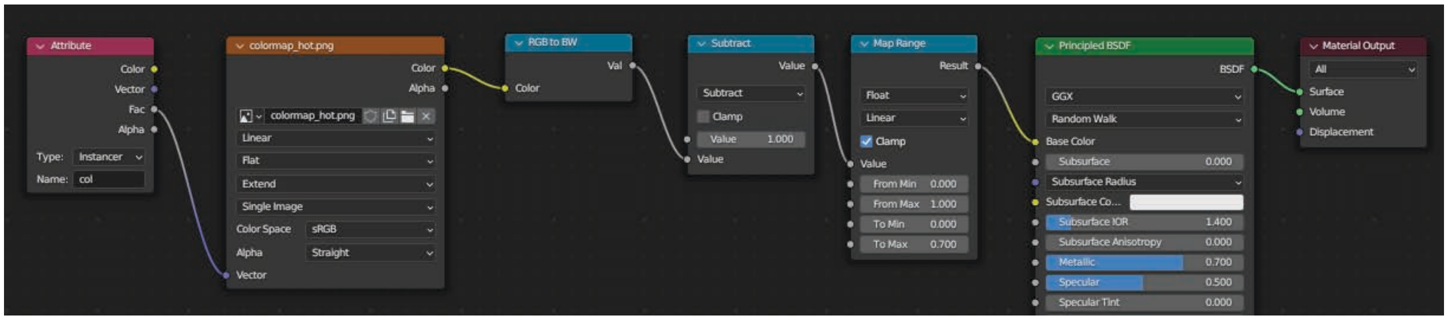


Figure 19: The updated node configuration in the Shader editor. Maybe a better way to do the inverse mapping (instead of the math nodes I used), is to simply use an Invert node.

12. Here's the final render. Throughout this example, I've prevented the arrows from having shadows rendered on them because I find them distracting in this particular example (Properties » Object Properties » Visibility » Ray Visibility and uncheck "Shadow".)

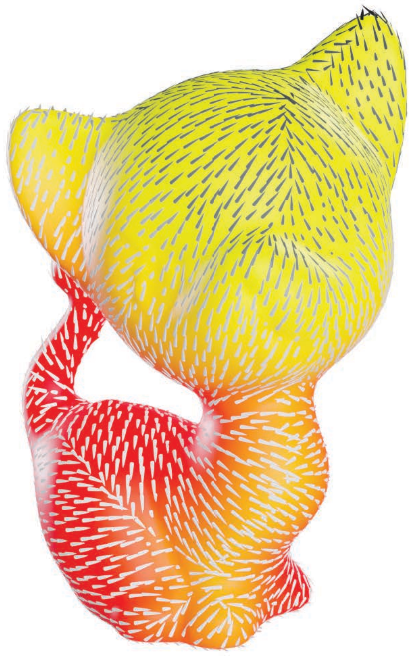


Figure 20: The "final" render. I'd personally still want to tweak the color gradient of the arrows, since the white arrows against the yellow portion of the surface are a bit hard to see. We would probably still want to make some tweaks to make the render look even better; see the sections on materials, lighting, and Section 10 for more tips.

3.2 Streamlines

TODO:

4 Rendering data with more than 3 dimensions

TODO: Probably easiest to do by using more than one file as input (and combining inputs appropriately), or co-opting other data-types in the same file. Or perhaps a Python script; see answers like this one.

5 Rendering curves

5.1 Basics

1. OBJ files can contain "line" elements. A line element is specified in an OBJ file with a line starting with the letter "l", followed by a sequence of vertex indices which build a polyline.
2. Load your OBJ into Blender. We will convert the polyline to a Curve object, for which there are nice rendering options in Blender. To convert to a Curve, select Object › Convert to › Curve.
3. Currently the curve will have zero width. To thicken the curve, go to Properties › Object Data Properties (icon in the right menu panel.) Go the `Geometry▸Bevel` within this tab, and up the "Depth" value. You may need to also up the "Resolution" value to maintain a smooth appearance. If this doesn't have an effect, make sure that `Shape▸Fill Mode` is set to "Full".
4. By default, the profile of the curve is circular.

5. You can set the material properties of the Curve like you would any other object, in Properties › Material Properties (the icon in the right menu panel.)

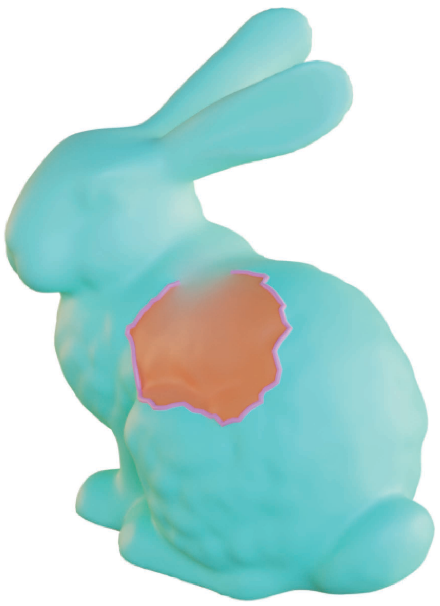


Figure 21: A bunny mesh with a round purple-colored Curve on it.

Additional options:

→ For more complex cross-sections, you can use a more complicated bevel (in Properties › Object Data Properties › Geometry › Bevel) or using a Bevel object (I have never tried the latter.)

→ To prevent the curves from casting a shadow on other objects, you must have selected "Cycles" as the render engine (Section 11.3). Make sure the curve object is selected in the viewport, then go to Properties › Object properties and `Visibility ▸ Ray visibility` and uncheck the "Shadow" box. You might want this feature, for example, if you're rendering curves that lie along a surface, and you want the curves to have volume but not cast distracting shadows on the surface.

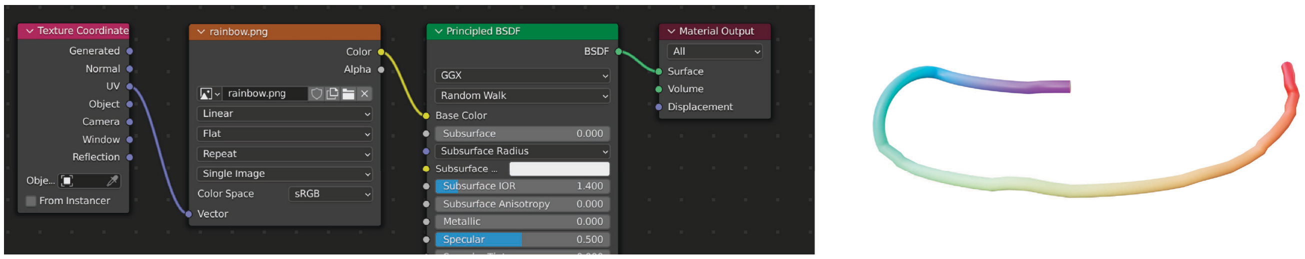
5.2 Making curves lie flat against a surface

TODO: This section under heavy construction — not safe for reading. I've found a way to do this using Geometry Nodes, but it's still not perfect. If you're rendering curves that lie along a surface and want the curves to lie flat on the surface, I have no idea how to do this. I tried for ages fiddling with the Bevel options and constructing my own Bevel (which is an option in Blender 3.0+), but it seems like the beveled curve will always have thickness. I suppose a solution is to construct your own polygon strip from the curve, and export that as a mesh.

5.3 Assigning UV coordinates

TODO: This section under heavy construction — not safe for reading Unfortunately, Blender doesn't have a way to store info along a Curve – you're only able to store UV coordinates if the Curve has been converted into Mesh, from which there doesn't appear to be a way to determine the coordinates corresponding to the control points of the original curve. This stinks, because a lot of the time we want to store per-vertex or per-segment attributes for a Curve, and this prevents us from doing so.

What's kind of silly is that Curves *do* have UV maps in Blender – it's just that there's no way to change it. The auto-generated UV map for a Curve divides the UV space more-or-less linearly between the points of the curve. So for example, you can use these UV coordinates to render simple gradients that interpolate from 0 at one end to 1 at the other end of the curve:



Linear
Flat
Repeat
Single Image
Color Space sRGB
Vector

GGX
Random Walk
Subsurface 0.000
Subsurface Radius
Subsurface ...
Subsurface IOR 1.400
Subsurface Anisotropy 0.000
Metallic 0.000
Specular 0.500

Figure 22: Using Blender's auto-generated UV coordinates to make a simple color gradient along a curve.

Therefore one possible workaround is to take advantage of these built-in UV coordinates to map a custom texture onto the Curve:

1. Parameterize your curve onto $[0, 1]$. Create an image texture encoding the values of your function at the corresponding x location of the image.
2. Load in this image texture; use the built-in UV coordinates to index the texture; feed the values into the shader.

More specifically, the auto-generated UV map maps to a square. What this means is that the UV coordinates are the result of distributing UV space *equally* among segments; segments are demarcated by edge loops in the beveled curve. Edge loops are displayed in Figure 23. Within each segment, UV space is distributed approximately *linearly* (up to some stretching induced by the beveling.) This is different than distributing UV space linearly along the whole curve; if you were to assign a uniform texture, you would see that the mapped texture would appear squashed on shorter segments. So you have to keep this in mind, depending on how you're trying to render your curve.

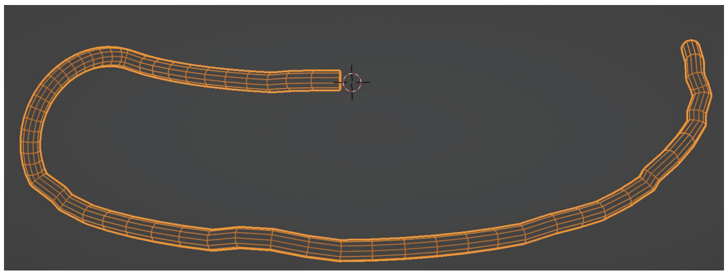


Figure 23: The edge loops on a beveled Curve.

When you load in your curve as a line element from an OBJ file, the beveled Curve that results will have one edge loop for each vertex (every point in the line element.)

TODO: It appears that Blender does indeed UV map in this way, although it's doing other unexpected things...

Warning: How you bevel the Curve might also stretch the UV map in unexpected ways. I also have only worked with piecewise linear curves; I'm not sure how much changes with Bezier curves.

6 Lighting

First delete any default lights that come with the Blender file. The default lighting will almost certainly not be able to provide the best lighting for your scene.

6.1 World lighting

Typically I don't use environment (world) lighting; usually I find that adding a few area lights works better. (See Section 6.2). But occasionally I find it useful — especially for larger scenes that contain more than a few objects, such as the "Planet Cow" scenes from my talk on winding numbers — so here are some steps on the process.

Blender comes with a bunch of default materials and light sources. You can view these resources by navigating to the Blender application, and on Mac OS, right-clicking and selecting "Show Package Contents". The directory `Contents`▸`Resources`▸`[version]`▸`datafiles`▸`studiolights`▸`world` contains some nice default lights that will illuminate the scene from all angles. In Linux, you also navigate to where your Blender application is located, and go to `[version]`▸`datafiles`▸`studiolights`.

1. Enter the Shading editor (see Figure 1.) In the upper left corner of the Shader editor, select "World" data from the drop-down menu:

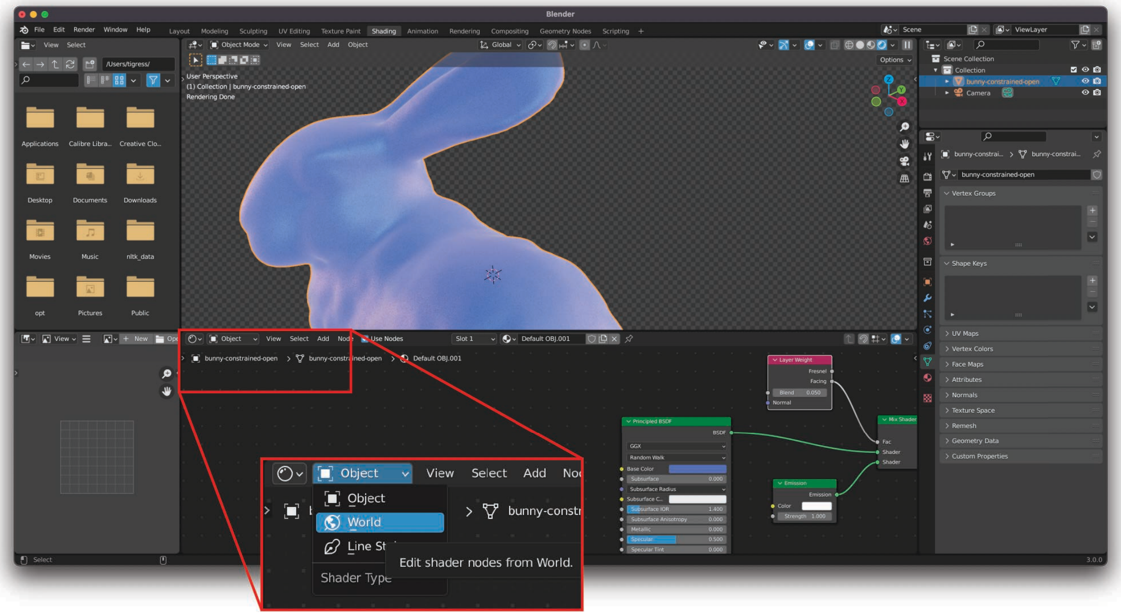


Figure 24: Navigating to world lighting in the Shading editor.

2. Add an Environment Texture node and connect it to the "Color" input of the Background node (see Figure 25.) The Environment Texture node has a field for a filename: click on this field and navigate to the default lighting textures in `Contents`▸`Resources`▸`[version]`▸`datafiles`▸`studiolights`▸`world`. So far I haven't found an easy way to navigate to this directory directly from the file browser that this opens up; a solution is to have the location of the lights open in a different file browser and drag in the filepath, or save the default lights in a more accessible location (i.e. not in the Blender package contents.)

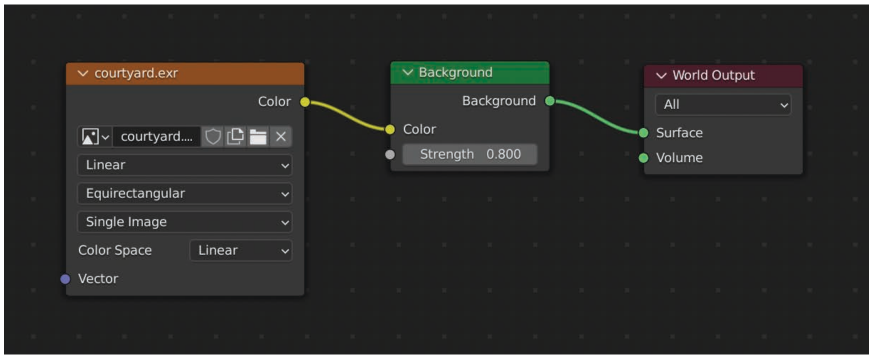
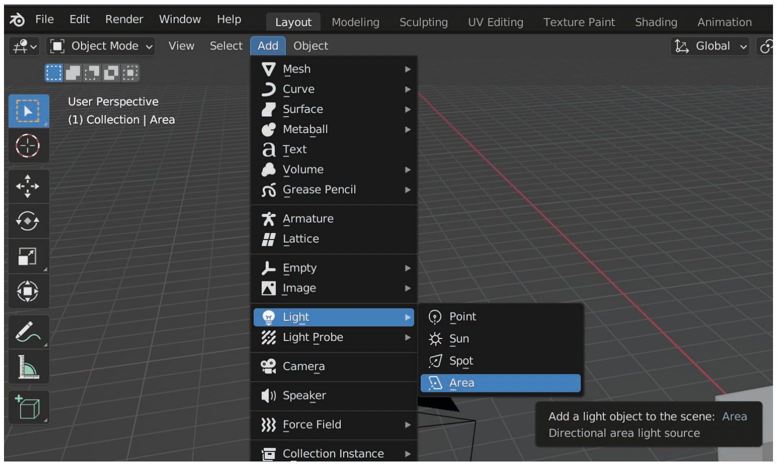


Figure 25: The nodes.

3. After selecting the light, you can play around with the strength and adjust it to taste. Hit F12 to get a render preview. As mentioned in Section 16.1, you will probably need more light than you think, so also test out how your render will look on a white background. To do the latter, you can either paste your rendered image into a separate document, or use the Compositing editor in Blender; see Section 11.5.

6.2 Directional lighting

For scenes focused on only one or a few items, instead of adding environment lights I almost always add specific directional lights to create aesthetically-pleasing self-shadows on the object, which I find gives the object a better sense of "3D-ness" and "3D placement".



Add a light object to the scene: Area
Directional area light source

Figure 26: How to add an area light.

6.2.1 The area lights method

I rarely use this method, but I've listed it for completeness (I usually use Sun lights instead, described in Section 6.2.2.)

I add a few area lights to illuminate the object from multiple sides. A quick way to add more lights is to duplicate an existing light (Shift+D.) I fiddle with the angle and power of the lights, toggling the camera view and using the "Rendered" viewport shading to see how the object looks (FAQ 15.3), until I'm reasonably satisfied that the object is well-lit (Figure 27.)

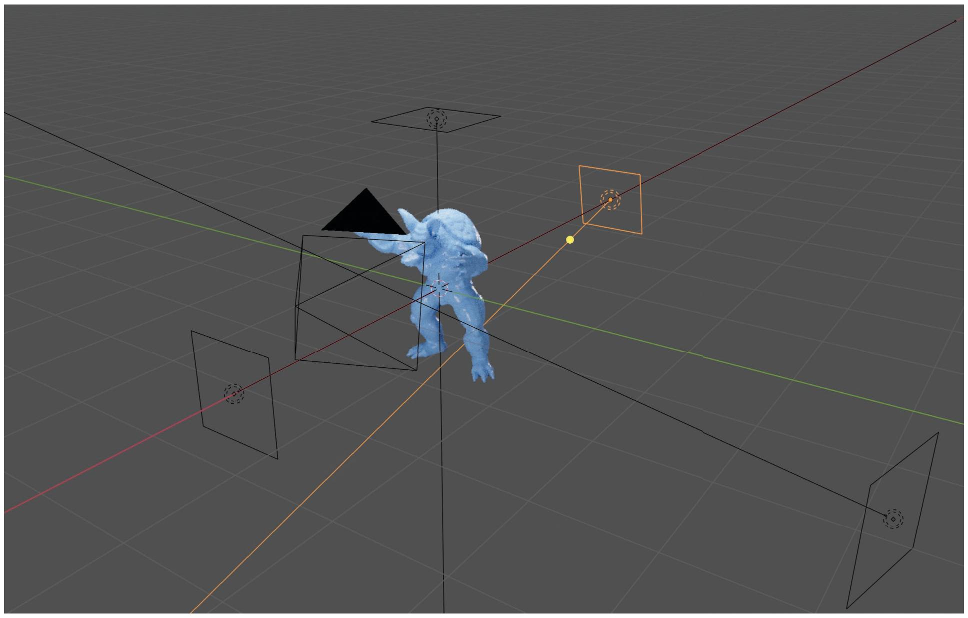
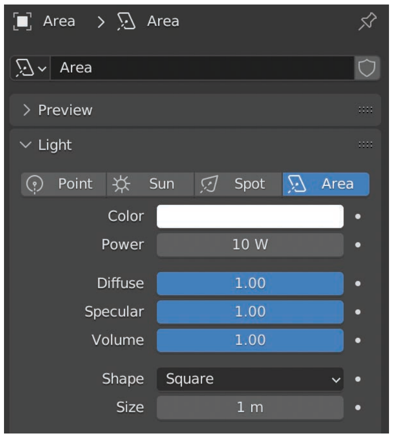


Figure 27: An example of a scene with several directional lights added.

The power of the lights can be adjusted under Properties › Object Data Properties (the icon in the right panel menu) – see Figure 28. The positioning and angles of the lights can be adjusted, just like any other object, under Properties › Object Properties (.)

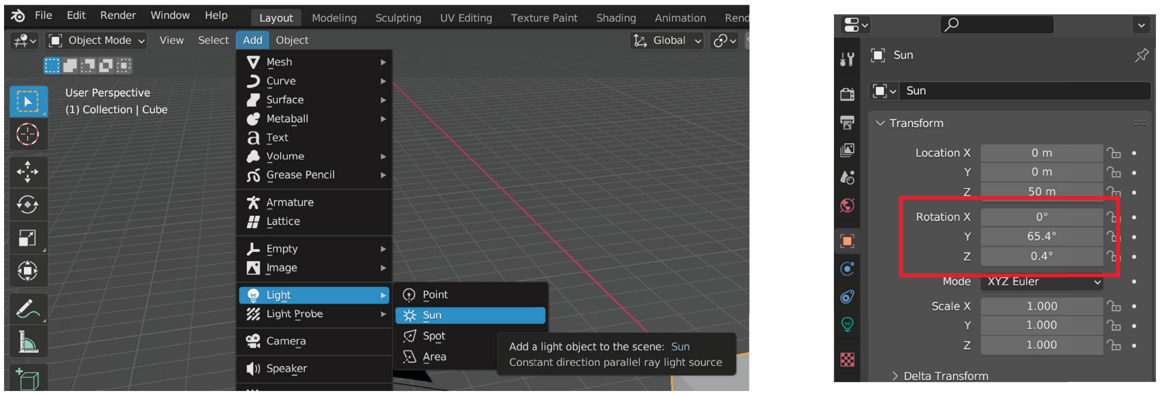


Color
Power 10 W
Diffuse 1.00
Specular 1.00
Volume 1.00
Shape Square
Size 1 m

Figure 28: The light settings underneath Properties › Object Data Properties (the icon.) Here, you can change the light's strength, or switch to a different type of light if needed.

6.2.2 The Sun light method

For typical single-object scenes, usually I only add one single "Sun"-type light, which I usually find is enough to both illuminate the object sufficiently, and result in visually-pleasing shadows that are not too complicated. After adding a Sun light, I simply play with the X-, Y-, and Z-"Rotation" parameters (by dragging the corresponding input boxes left and right) until the scene's objects have self-shadows that look good to me. For most of my scenes, I find that I usually have to increase the light's power ∼4000 watts (Figure 28).



Add a light object to the scene: Sun
Constant direction parallel ray light source

Location X	0 m
Y	0 m
Z	50 m
Rotation X	0°
Y	65.4°
Z	0.4°
Mode	XYZ Euler
Scale X	1.000
Y	1.000
Z	1.000

Figure 29: *Left:* Adding a Sun light. *Right:* Under Properties 》Object Properties (the icon), I simply play with the X-, Y-, and Z-"Rotation" parameters (by dragging the corresponding input boxes left and right) until the scene's objects have self-shadows that look good to me.

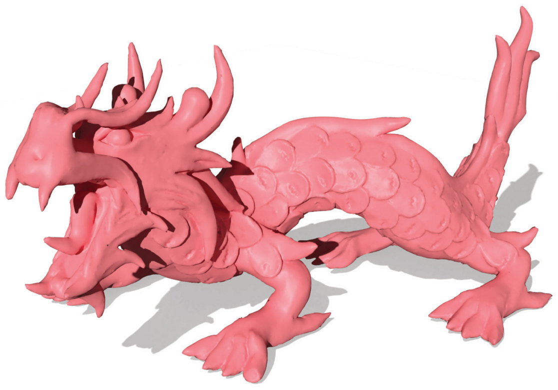


Figure 30: An example of an object lit using a single Sun light — the next section details the steps used to create the shadow. For more examples, see this paper where I lit every 3D scene with this method.

7 Shadows

7.1 Cast shadows

Rendered images are often lent a greater degree of "realness" and sense of 3D placement in the context of a paper when rendered with a cast shadow. This is usually done by adding an "invisible" plane onto which the shadow from the object is cast. To add a plane, go to Add 》Mesh 》Plane.

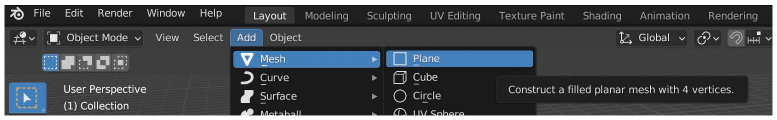
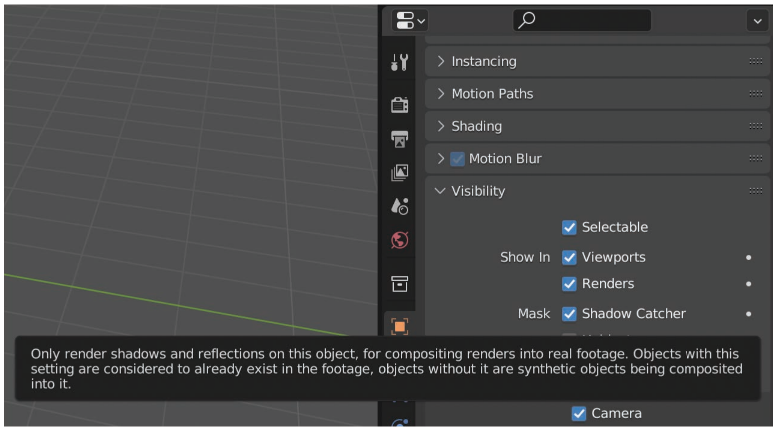

File Edit Render Window Help
Layout Modeling Sculpting UV Editing Texture Paint Shading Animation Rendering
Object Mode View Select Add Object
Global
User Perspective
(1) Collection
Mesh
Curve
Surface
Metaball
Plane
Cube
Circle
UV Sphere
Construct a filled planar mesh with 4 vertices.

Figure 31: How to add a plane mesh.

Make the plane big enough so that it captures all of the object's shadow by scaling it under Object Properties (.) Then apply the transforms by hitting Ctrl + A .

To have the plane catch shadows but not actually be rendered in the final render, select the plane and turn on the "Shadow Catcher" option: Go to Properties » Object Data Properties » Visibility » Shadow Catcher. You must be using the Cycles render engine, not Eevee, to use Shadow Catcher.



Only render shadows and reflections on this object, for compositing renders into real footage. Objects with this setting are considered to already exist in the footage, objects without it are synthetic objects being composited into it.

Figure 32: Turning on the Shadow Catcher option. This option is often used for ground planes that are used to render shadows on an invisible "ground".

Usually, the light settings that best illuminate the scene objects well, such as those shown in Figure 27, are *not* simultaneously ideal for generating clean-cut, well-positioned, aesthetically-pleasing cast shadows. I usually therefore create a second set of lights specifically for creating the shadow. To keep these lights logically separate from the original scene items, I add a new scene "collection" by right-clicking on "Scene Collection" in the Outliner menu in the upper-right (Figure 33.)

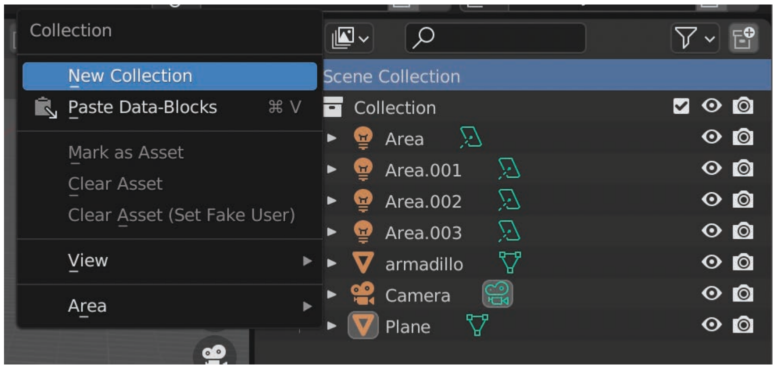
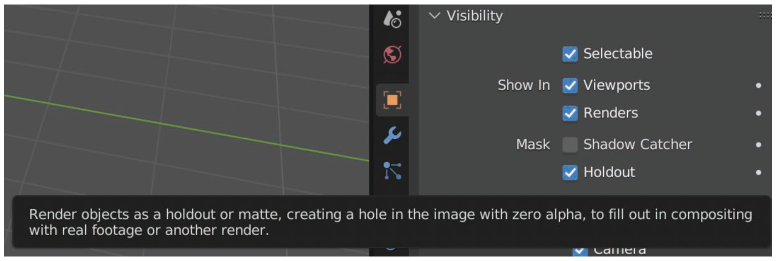


Figure 33: Adding a new collection.

To this new collection, I add all the objects that are needed to make the shadow: the ground plane; a copy of the scene object(s), and additional light(s) that I've fiddled around with and deemed to make a good shadow. If the scene object(s) are particularly high-resolution and adding copies of them would slow the Blender and its rendering process down considerably, I just add lower-resolution versions of the objects (you don't need high-res objects to generate the shadows, which are fuzzy anyway!) Next, you want to select the "Holdout" option under Object Properties » Visibility, which takes an object's occlusion into account in the render, without rendering the object itself:



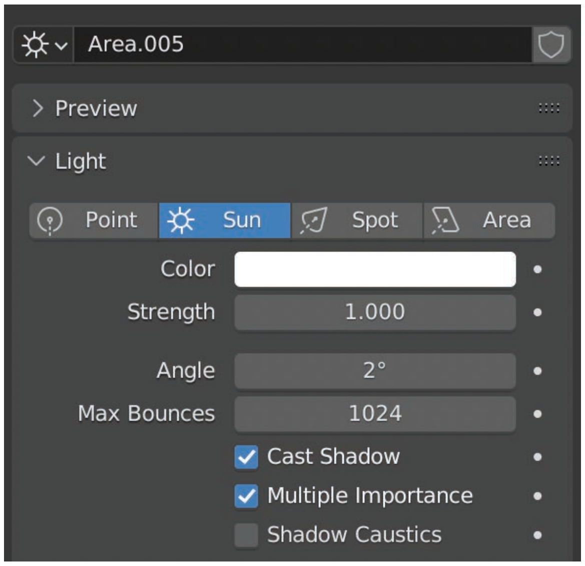
Visibility

	✓ Selectable
Show In	✓ Viewports
	✓ Renders
Mask	Shadow Catcher
	✓ Holdout

Render objects as a holdout or matte, creating a hole in the image with zero alpha, to fill out in compositing with real footage or another render.

Figure 34: Setting an object to rendered as "holdout."

For the light that generates the shadow, I usually add a "Sun"-type light to make the shadow, as it tends to make the most clean-cut shadow whose positioning is easiest to control. I usually just place the sun somewhere above the scene objects; then fiddle around with its rotation under Object Properties » Rotation ( ) until the shadow looks okay to me in the viewport's render preview. Further, you can adjust the fuzziness of the shadow's outlines by increasing the "Angle" option in the light's Object Data Properties ( ). An angle of 0° will create a crisp shadow, while 2° — 4° is usually enough to make a nice-looking "fuzzy" shadow. (I usually just use 0°.)



Area.005

Preview

Light

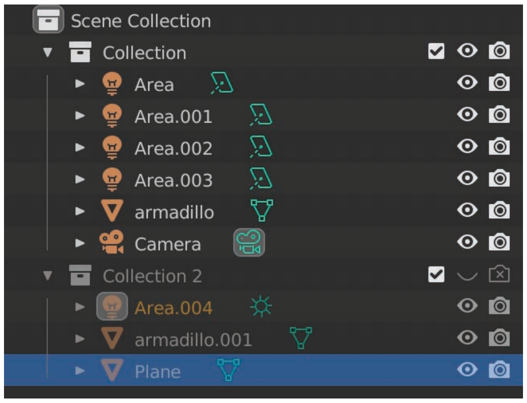
Point | Sun | Spot | Area

Color	
Strength	1.000
Angle	2°
Max Bounces	1024
	✓ Cast Shadow
	✓ Multiple Importance
	Shadow Caustics

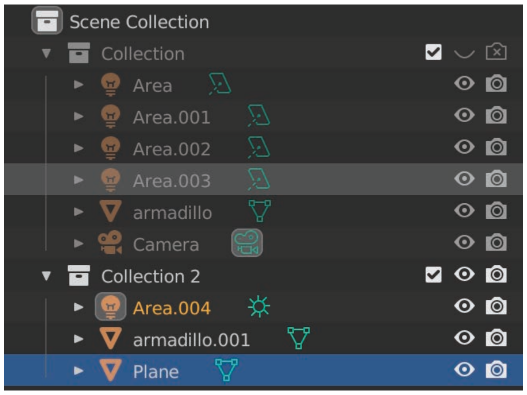
Figure 35: Adjusting the options of the "Sun" light to tune the resulting shadow. Increasing the angle will create a shadow with a fuzzier outline; I usually just use 0°. For most of my scenes, I usually increase the light's power ∼4000 watts, and set the ground plane's alpha value in the Shading editor to 0.600, though I imagine these parameters will be somewhat scene-dependent.

Sometimes the shadow is initially too dark. The easiest way to remedy this is to change the alpha value of the ground plane: Go to the Shading editor, select the plane, and set its alpha value to something less than 1. For most of my scenes, I usually set the ground plane's alpha value 0.600.

We are now able to render two images separately: One containing just the lit and shaded object (no ground shadow, only self-shadows), and a second containing just the ground shadow. To render the two scene collections separately, toggle the camera icon next to the collection ("Disable in renders", .) To avoid confusing myself, I always also toggle the "Hide in viewport" option as well (the eye icon) at the same time.



(a) The lit and shaded objects will be rendered; the shadows are disabled.



(b) The lit and shaded objects are disabled from the render; the shadows will be rendered.

Figure 36: Rendering the scene objects separately from their shadows.

After rendering these two images separately, I composite these two images together for the final figure (I use Adobe Illustrator.)

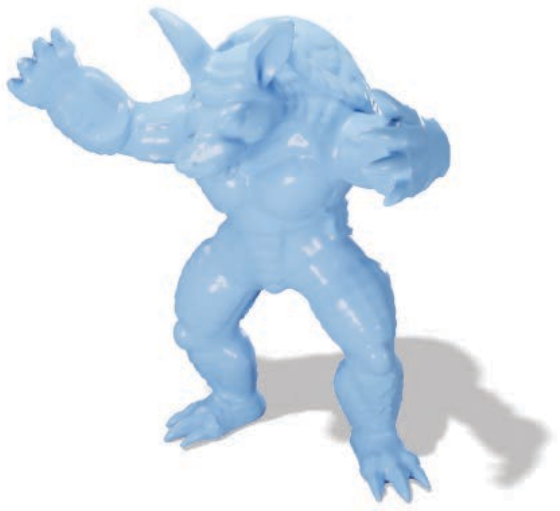


Figure 37: A rendered object composited with its shadow. See also Figure ??.

7.2 Non-physically-based shadows

Alternatively, you can add a "shadow" in a 2D graphics editor like Adobe Illustrator that only appears physically correct. Creating a shadow in this manner will be easier with regards to controlling the extent of the shadow, its shape, its crispness, etc. However, the shadow will almost inevitably have a more cartoonish/cel-shaded aesthetic to it. This might fit exactly into your aesthetic goals for the figure — or it might make your render look fake and funny, so you should exercise your own judgement!

8 Function visualization

8.1 Setting colormaps

You can either set the control points of a Color Ramp node using a Python script (which I personally find confusing), or use an image as a color map via a Image Texture node (which I find much easier.)

Matplotlib has some examples of good color maps that are designed to be perceptually aligned with how the data might change. Many colormaps in Polyscope come from Matplotlib. Additionally, Mark has some code that samples from these colormaps and outputs a PNG; he has some examples here.

Once you have the colormap image, simply add a Image Texture node, select your image file, and hook the node up to the base color of your mesh (Figure 38).

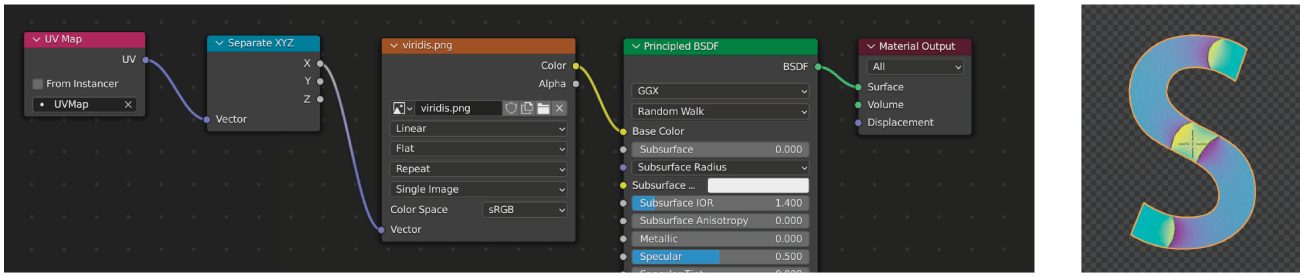


Figure 38: Using an external colormap to visualize scalar data. Left: The node configuration. Right: A viewport preview of the mesh, which is using the “viridis” colormap from Matplotlib.

Important: Lights and materials will often interact with your colors in unexpected ways, leading to colors that are too washed out or too dark. This will look bad in a paper figure (Figure 39), because your color bar legend will look like it’s a different color than what’s rendered on your surface:

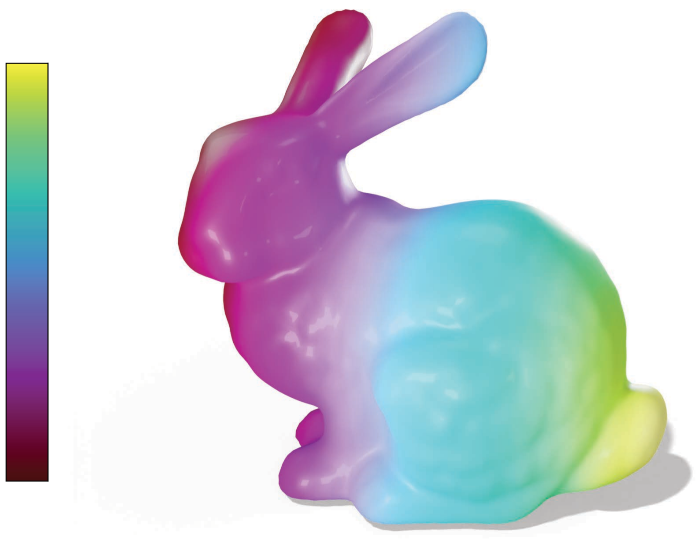
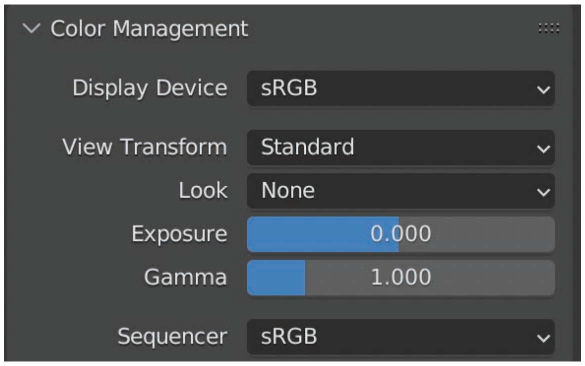


Figure 39: Bad color management! The texture-mapped colors look completely different than what they did originally.

There are two main things you must do to get more accurate color-mapping. First is go to Properties › Render Properties › Color Management (.) There is something in the Render Properties called the “View Transform” whose default is set to a non-standard option for some reason. You must switch to "Standard", which will then convert colors to the display the "standard" way. There are also contrast settings there ("Look") although I find that leaving the default option to "None" is sufficient.


Color Management

Display Device: sRGB
View Transform: Standard
Look: None
Exposure: 0.000
Gamma: 1.000
Sequencer: sRGB

Figure 40: The Color Management settings.

Further, to avoid the color of textures being washed out by the lights in your scene, you must add a separate "Emission" node in your shader. Hook up the output of the texture map to the input of Emission. Add a Mix Shader node to mix the output of Emission with that of BSDF. Pure emission would shade the object perfectly according to the colors in the texture, but would ignore the effects of the BSDF shader; to strike a good balance, leave the strength of Emission at 1, and usually a value of 0.5 in Mix Shader is good enough. After adding an emission shader, you may have to re-adjust some of the specular/roughness settings in BSDF to retain the material appearance from before — for this reason, I usually adjust emission settings before finalizing my material parameters.

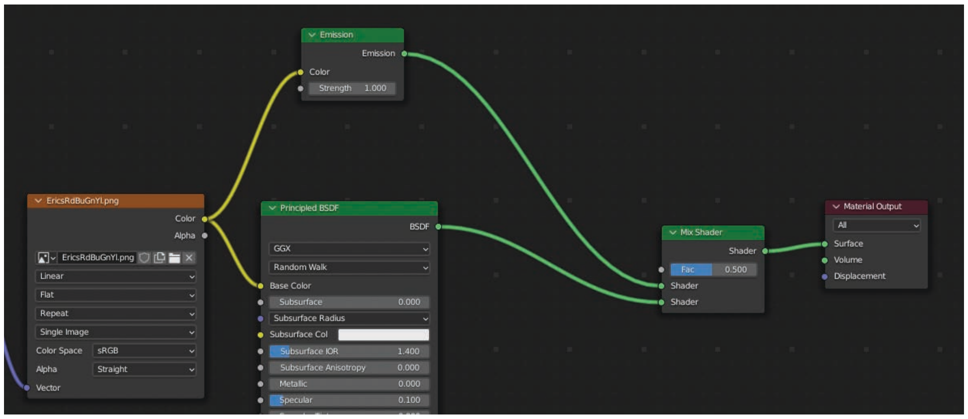


Figure 41: Adding an emission shader.

Here is a better color-mapped surface:

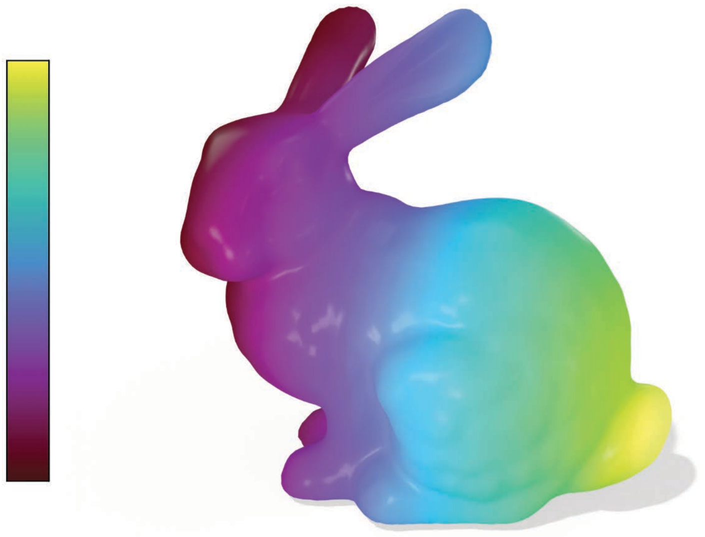


Figure 42: Better color management — the colors now match better between the colorbar and the texture-mapped color on the surface.

8.2 Rendering isolines

Sometimes you want to render a scalar function on a mesh, and also display its isolines. A classic example is a distance function.

8.2.1 Using trigonometric functions

To render isolines as darkened bands, I use the following method, which is based on tricks I picked up from looking at Inigo Quilez's shaders on Shadertoy. (There are also simpler methods, described below.) Essentially, the scalar function ϕ is color-mapped as usual. But ϕ also gets mapped via a periodic function, which darkens the color quantity at regularly-spaced intervals (regularly-spaced numerically, that is. Whether the isolines actually end up being regularly-spaced spatially depends on whether ϕ is a true distance function.)

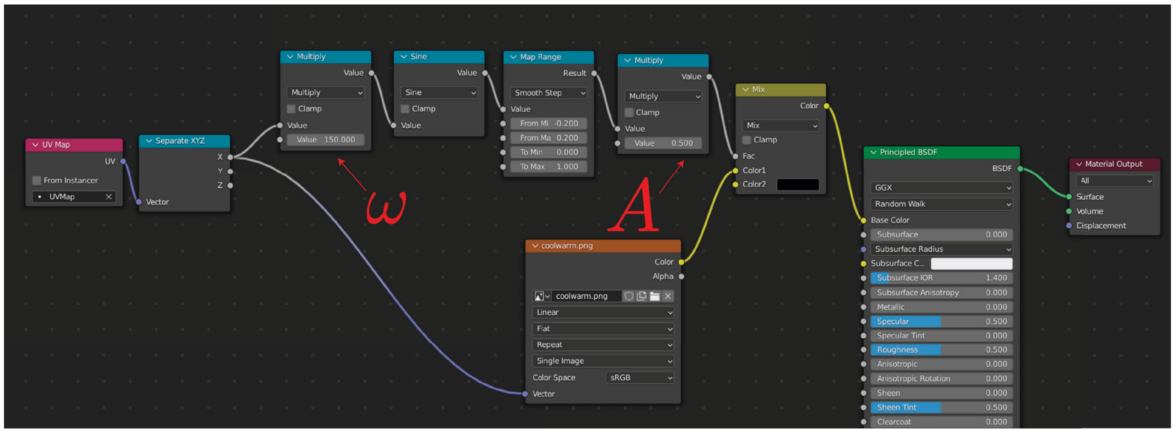
For the periodic function, a sinusoid is perfect the job; its range will eventually get mapped to $[0, 1]$, representing the darkening factor. The sine function is especially nice because $\sin(0) = 0$, which avoids having an isoband straddle the zero-set of the function. The period length controls the width of the isobands, and the amplitude controls the darkness. The function that computes the darkening factor is as follows:

$$\text{darkFactor}(\phi,\ A,\ b) = A * \text{smoothstep}(\text{-}b,\ b,\ \sin(\omega\phi))$$

This computes the factor by which the isobands are darkened. The amplitude A controls the darkness of the isobands ($A \propto$ darkness), ω controls the width of the isobands ($\omega \propto \text{width}^{-1}$), and b controls how much the function `smoothstep()` smooths out the edges of the bands, which avoids artifacts ($b \propto$ smoothness.)

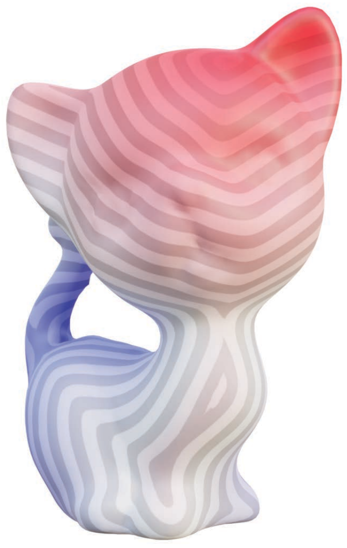
Annoyingly, Blender doesn't have nodes that allow you to directly write your own script defining how to transform a quantity, like Houdini. Instead we build the darkening function by hooking up individual Math nodes, which only allow for one atomic operation at a time. The smoothstep function is implemented in the Map Range node; just select the "Smooth step" option. See Figure 43.

The output of this chain of Math nodes is a scalar between 0 and 1 that indicates the degree to which a color should be darkened. Hook the output of the last of the Math nodes to the '`Fac`' input of a Mix RGB node, and hook the Color Ramp output to the first '`Color`' input of Mix RGB. Set the second color of Mix RGB to be black. This will allow us to blend between the original color map output, and a color that transitions to black, with the degree of transition controlled by the darkening factor we compute. The final node configuration is shown in Figure 43.



UV Map: From Instancer ☐, UVMap
Multiply: Multiply, Clamp ☐, Value 150.000
Sine: Sine, Clamp ☐
Map Range: Smooth Step, From Mi -0.200, From Ma 0.200, To Min 0.000, To Max 1.000
Multiply: Multiply, Clamp ☐, Value 0.500
Mix: Mix, Clamp ☐
coolwarm.png: coolwarm.png, Linear, Flat, Repeat, Single Image, Color Space sRGB
Principled BSDF: GGX, Random Walk, Subsurface 0.000, Subsurface Radius, Subsurface C..., Subsurface IOR 1.400, Subsurface Anisotropy 0.000, Metallic 0.000, Specular 0.500, Specular Tint 0.000, Roughness 0.500, Anisotropic 0.000, Anisotropic Rotation 0.000, Sheen 0.000, Sheen Tint 0.500, Clearcoat 0.000
Material Output: All

(a) The node configuration.

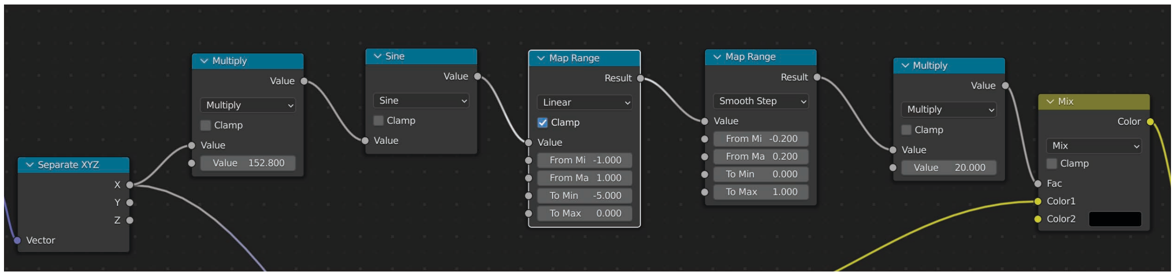


(b) The final render.

Figure 43: Rendering isobands of a distance function on a kitty.

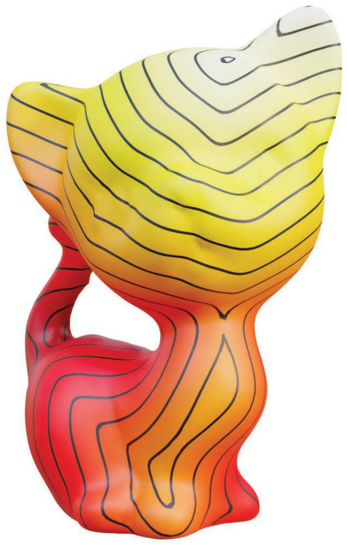
Instead of blending with black, you can experiment with other colors or types of quantities (color, brightness, etc.) although I've found that simply blending with black or white yields best results.

A possible downside of this method is that it's not so straightforward to control the ratio of [width of dark bands]:[width of light bands]. You can kind of control the widths of the isobands by fiddling with the min/max values of the pre-image of smoothstep(), but this also affects the level of smoothness, so tends to result in overly-fuzzy bands. A better way to adjust the band width is to add another Map Range node just before the Map Range node that implements `smoothstep()`. By re-mapping the output of sin from $[-1, 1]$ to an interval that, for example, is mostly negative like $[-5, 0]$, this forces the subsequent Map Range, which is set up to map values onto $[0, 1]$, to clamp all values less than its "From minimum" value to 0. Hence a narrower band of values receive darkening, resulting in thinner isobands. See Figure 44.



```
Multiply: Multiply, Clamp [ ], Value 152.800
Sine: Sine, Clamp [ ]
Map Range: Linear, Clamp [✓], From Mi -1.000, From Ma 1.000, To Min -5.000, To Max 0.000
Map Range: Smooth Step, From Mi -0.200, From Ma 0.200, To Min 0.000, To Max 1.000
Multiply: Multiply, Clamp [ ], Value 20.000
Mix: Mix, Clamp [ ]
```

(a) Adding another Map Range node (highlighted in white) to control the width of the isobands.



(b) The final render.

Figure 44: Decreasing the isoband width and increasing the amplitude A results in thin black lines.

8.2.2 Using modular arithmetic

TODO: Alternatively, you can also take mod 1 using a math node.

9 Materials

For paper figures, I almost always just adjust the Specular, Roughness, and Clearcoat parameters of the Principal BSDF node in the Shader until I'm happy with it.

TODO: PBR materials

10 Additional visual effects

10.1 Adjust saturation/lightness

TODO

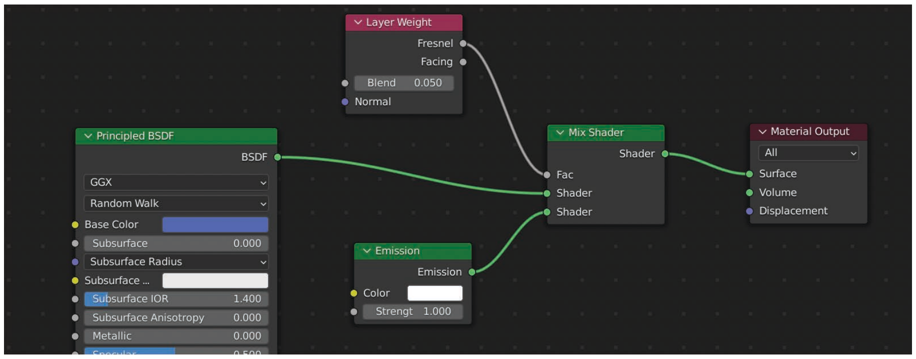
10.2 Rim light effect

This tip is courtesy of Mark, to get a "glow" effect around the silhouettes of the mesh. In photography and animation, rim lights / a rim light effect is used to increase contrast between the foreground object and background.

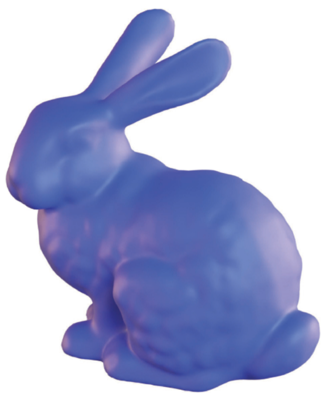
Enter the Shader editor (Shading.)

1. Add a Layer weight node, and an Emission node.
2. Set the color of the emission to something lighter-colored, like white.
3. Add a Mix Shader node and drop it between Principled BSDF and Material Output. Hook up Emission to the other "Shader" input of Mix Shader. Finally, hook up the "Fresnel" output of Layer weight to the "Fac" input of Mix Shader, and set its value to something relatively small like 0.05.

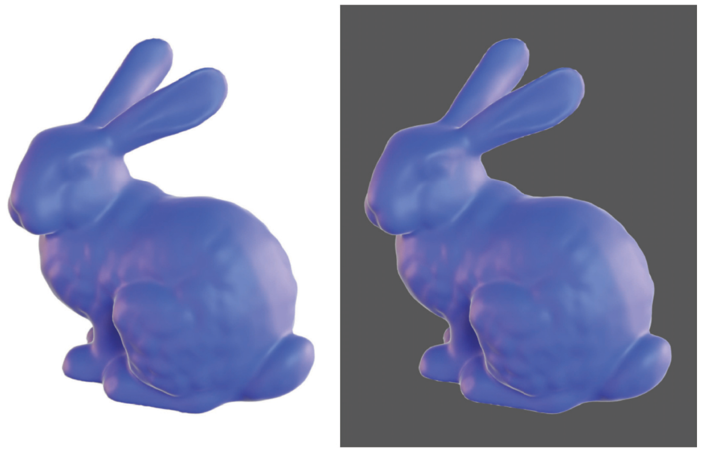
This method produces a white-ish glow around the silhouette of the mesh (Figure 45.)



(a) The nodes.



(b) Without glow effect.



(c) With glow effect.

Figure 45: The glow effect (45c) can be more easily seen against a dark background. On the other hand, the glow around the outer silhouettes tends to be jagged.

11 Common rendering settings

11.1 Rendering the final image

To render the image, hit F12. A separate render window will pop up.

11.2 Smooth shading

Most of the time we want our polygonal models to be smoothly shaded. In Object menu button in the upper left of the viewport (Figure 46), click 'Shade smooth' for smooth shading.

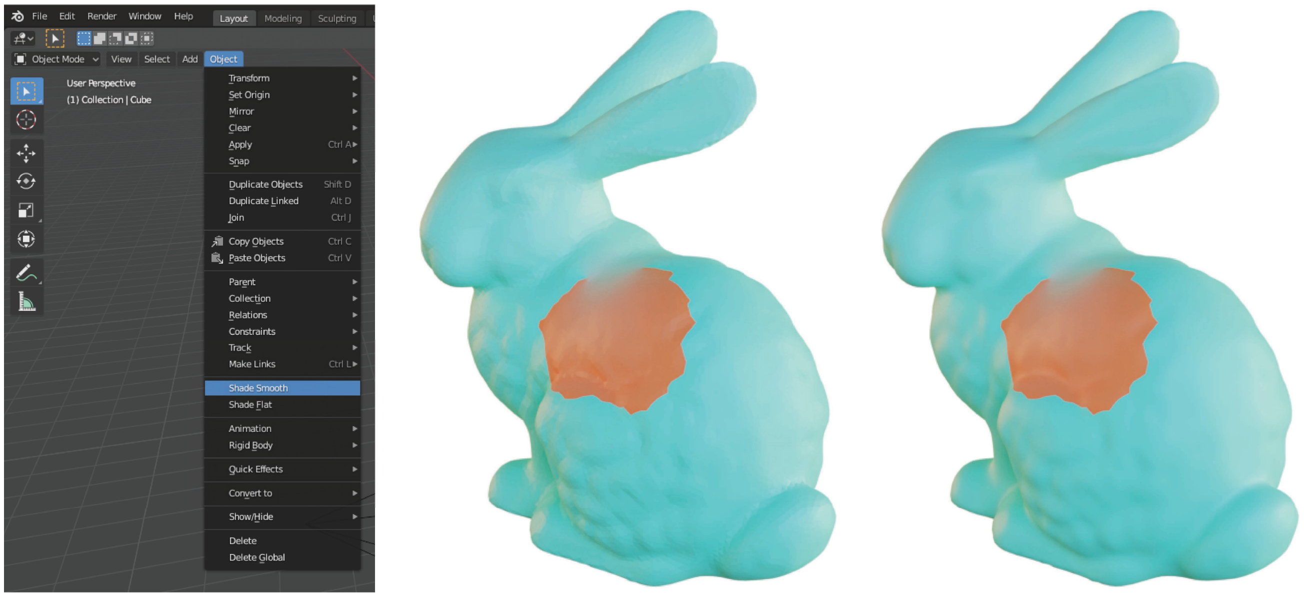


Figure 46: The Object » Shade smooth option (left.) The difference between 'Shade flat' (middle) and 'Shade smooth' (right) is shown (may need to zoom in to see the difference.)

You can also use the Set Shade Smooth node in Geometry Nodes:

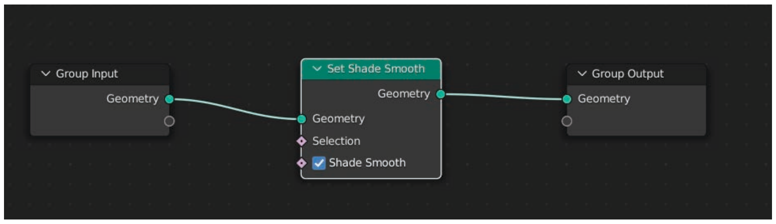


Figure 47

11.3 Eevee vs. Cycles

For better rendering quality, go to Properties » Render properties (in the right panel menu.) For "Render Engine", select "Cycles" instead of "Eevee". "Cycles" uses ray-tracing.

On the other hand, rendering using "Cycles" takes a lot longer. While iterating on your scene, it may be wiser to use "Eevee"; or at the very least, use "Material Preview" as your viewport shading, rather than "Rendered" (see Section 12.1.)

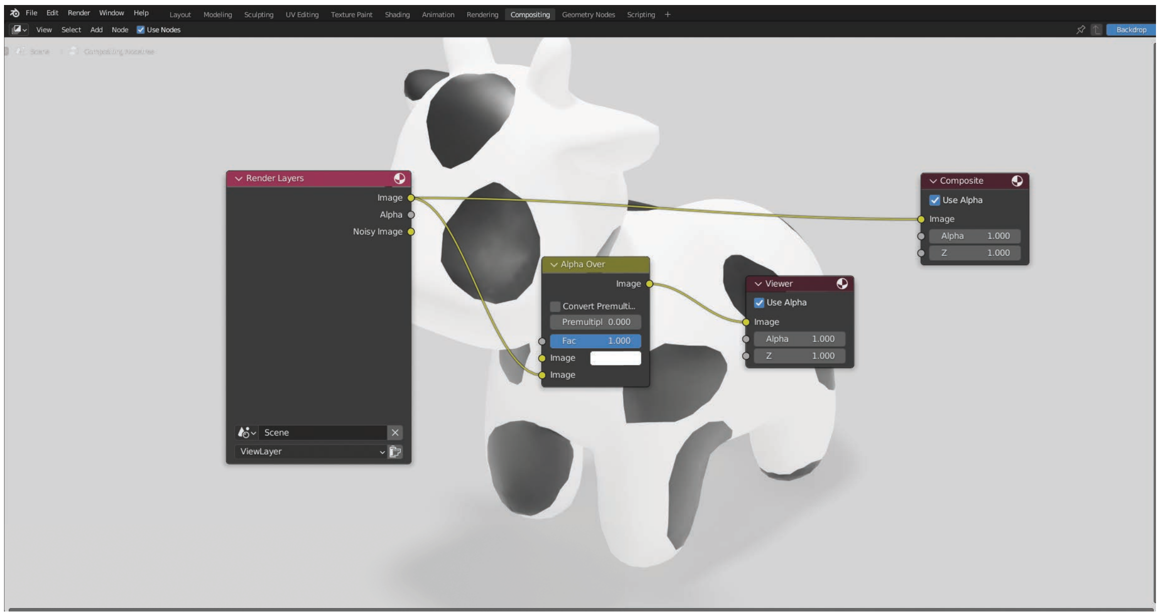
When using "Cycles", typically a value of 50 for "max samples" during rendering is enough to produce a crisp picture. For more complicated geometry, transparency, etc. you may need more. You can also turn down the number of max samples to about 30 or so for viewport rendering.

11.4 Rendering images with transparency

To render images with transparent background, go to Properties » Render properties (in the right panel menu.) Under the `Film` drop-down menu, check the 'Transparent" box.

11.5 View render with a transparent background against a white background

It can be hard to tell if the lighting is correct in your scene just from the final render (which usually has a transparent background); you'll want to preview it against a white background, to see how the image will look in a paper document, slides, etc. You can do this either by pasting your rendered image into a separate document; or using the Compositing editor with the following nodes:



Alpha Over: Convert Premulti... (unchecked), Premultipl 0.000, Fac 1.000

Viewer: Use Alpha (checked), Alpha 1.000, Z 1.000

Composite: Use Alpha (checked), Alpha 1.000, Z 1.000

Figure 48: Viewing a render with a transparent background against a white background in the Compositing editor.

TODO: There's some weird gamma correction going on, so that the background isn't perfectly white — I don't really use this method myself at the moment.

12 Viewport options

12.1 Viewport shading

In the upper right of the viewport, there are a few options for viewport shading.

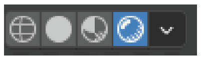


Figure 49: The viewport shading icons in the upper right of the viewport.

Usually "Rendered" is the most useful, since it is the closest to what your final render will look like. The "Rendered" shading can be slow/noisy if you're using Cycles however, so it may be more efficient to toggle "Material Preview" instead, or use Eevee.

12.2 Changing viewport appearance

To change the default background color of the viewport, go to Edit › Preferences › Themes › 3D Viewport › Theme Space › Gradient Colors and change the values of "Gradient High/Low". (The Edit › menu is in the upper left of the main window.) This will change the default background color, i.e. the viewport background will now be this color every time you open a new Blender file.

You can also customize the vignetting in the viewport in Edit › Preferences › Themes › 3D Viewport › Gradient Colors › Background Type.

13 Scripting

In theory, everything done using the Blender GUI can also be achieved via a Python script. Using a script is useful when you need to render many objects, and/or set similar rendering properties for each of them. You can look up documentation online about the Blender Python API. Personally, I found it takes a definite effort to get the hang of scripting in Blender. It's helpful to start by trying to re-create a simple Blender example you have already done using the GUI – for example, one of the examples in this document.

Also, Python scripting is essentially a programmatic way to manipulate the Blender GUI. Therefore, I expect it will be easier to learn to use Blender through the GUI first, before trying scripting. It's hard to know what to script without understanding how Blender expects objects to be manipulated.

A tip mentioned by "Linus" in the 2022 SGP Discord server: Enable Preferences » Interface » Python tooltips, and then hovering over UI elements will tell you how to access it with Python.

Note: You must launch Blender from the command line to receive output to terminal.

14 Useful keyboard shortcuts

14.1 Change view in viewport

You can quickly snap to different views using 1, 3, 7, and 9 on the numberpad. Each of them snaps to an axis-aligned view. Additionally, 4 and 6 rotate the view in increments around the z-axis; 2 and 8 incrementally rotate the view around the x-axis. Hitting 5 toggles perspective vs. orthographic view.

Unfortunately these shortcuts only work on keyboards with a numberpad (unless you do some fancier keyboard re-mapping.)

15 FAQs

15.1 How do I delete _____?

Select the object and hit the x key.

15.2 How do I disconnect a node?

There are a few ways to disconnect nodes:

1. Hold down ctrl or ⌘, and drag the right mouse button to "cut" links with a knife (see Figure 50.)

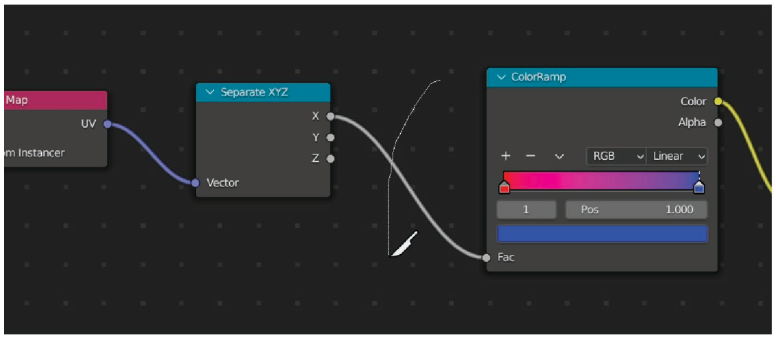
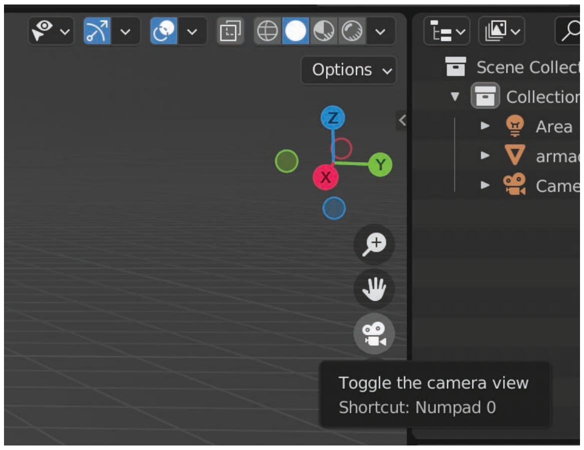


Figure 50: Disconnecting nodes.

2. Alternatively, hold down option/alt and use the left mouse button to drag a node "out" of a connection.
3. Alternatively, you can simply grab the input connection with your mouse, and drag it off.

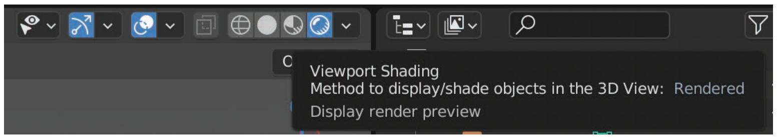
15.3 How do I preview the object before rendering?

Toggle the camera view (Figure 51) to view the object in the viewport through the currently-active camera. Turn on the "Rendered" option under Viewport Shading (Figure 52) to display in the viewport a preview of how your object will be lit and shaded.



Toggle the camera view
Shortcut: Numpad 0

Figure 51: Toggling the camera view.



Viewport Shading
Method to display/shade objects in the 3D View: Rendered
Display render preview

Figure 52: Turn on render preview in the viewport.

15.4 How do I use a different camera to render images, without affecting existing cameras?

16 Gotchas

16.1 Your scene probably needs a lot more light than you think

Even if you set the background color of the viewport to white, objects will look brighter in Blender than they will on the white background of a paper. So you will have to add a lot more light than you think in order for the images to stand out on a white page. See Section 11.5 for how to view your (transparent background) image against a light-colored background.

16.2 Color management

Go to Properties » Render Properties » Color Management and set "View Transform" to "Standard" to get non-washed out colors. To get accurate texture colors, see Section 8 for more detail.

16.3 Receiving terminal output

You must launch Blender from the command line to receive output to terminal, i.e. print statements and other useful error messages (helpful when debugging.)

Part I

Animations