# Advanced Raster Visualization (F6.0)

---


### Authors

Gennadii Donchyts, Fedor Baart


---

### Overview

This chapter should help users of Earth Engine to better understand raster data by applying visualization algorithms such as hillshading, hill shadows, and custom colormaps. We will also learn how image collection datasets can be explored by animating them as well as by annotating with text labels, using, for example, attributes of images or values queried from images.

### Learning Outcomes

- Understanding why perceptually uniform colormaps are better to present data and using them efficiently for raster visualization.
- Using palettes with images before and after remapping values.
- Adding text annotations when visualizing images or features.
- Animating image collections in multiple ways (animated GIFs, exporting video clips, interactive animations with UI controls).
- Adding hillshading and shadows to help visualize raster datasets.

### Assumes you know how to:

- Import images and image collections, filter, and visualize (Part F1).
- Write a function and `map` it over an `ImageCollection` (Chap. F4.0).
- Inspect an `Image` and an `ImageCollection`, as well as their properties (Chap. F4.1).

### Introduction to Theory

Visualization is the step to transform data into a visual representation. You make a visualization as soon as you add your first layer to your map in Google Earth Engine. Sometimes you just want to have a first look at a dataset during the exploration phase. But as you move towards the dissemination phase, where you want to spread your

define the code to annotate an image as a function. We will then map that function (as described in Chap. F4.0) over the filtered `ImageCollection`. The code is as follows:

```
var text = require('users/gena/packages:text');

var geometry = ee.Geometry.Polygon(
    [
        [
            [-109.248, 43.3913],
            [-109.248, 33.2689],
            [-86.5283, 33.2689],
            [-86.5283, 43.3913]
        ]
    ], null, false);

Map.centerObject(geometry, 6);

function annotate(image) {
    // Annotates an image by adding outline border and cloudiness
    // Cloudiness is shown as a text string rendered at the image
center.

    // Add an edge around the image.
    var edge = ee.FeatureCollection([image])
        .style({
            color: 'cccc00cc',
            fillColor: '00000000'
        });

    // Draw cloudiness as text.
    var props = {
        textColor: '0000aa',
        outlineColor: 'ffffff',
        outlineWidth: 2,
        outlineOpacity: 0.6,
        fontSize: 24,
        fontType: 'Consolas'
    };
    var center = image.geometry().centroid(1);
    var str = ee.Number(image.get('CLOUD_COVER')).format('%.2f');
```

```
    var scale = Map.getScale();
    var textCloudiness = text.draw(str, center, scale, props);

    // Shift left 25 pixels.
    textCloudiness = textCloudiness
        .translate(-scale * 25, 0, 'meters', 'EPSG:3857');

    // Merge results.
    return ee.ImageCollection([edge, textCloudiness]).mosaic();
}

// Select images.
var images = ee.ImageCollection('LANDSAT/LC08/C02/T1_RT_TOA')
    .select([5, 4, 2])
    .filterBounds(geometry)
    .filterDate('2018-01-01', '2018-01-7');

// dim background.
Map.addLayer(ee.Image(1), {
    palette: ['black']
}, 'black', true, 0.5);

// Show images.
Map.addLayer(images, {
    min: 0.05,
    max: 1,
    gamma: 1.4
}, 'images');

// Show annotations.
var labels = images.map(annotate);
var labelsLayer = ui.Map.Layer(labels, {}, 'annotations');
Map.layers().add(labelsLayer);
```

The result of defining and mapping this function over the filtered set of images is shown in Fig. F6.0.9. Notice that by adding an outline around the text, we can ensure the text is visible for both dark and light images. Earth Engine requires casting properties to their corresponding value type, which is why we've used `ee.Number` (as described in Chap. F1.0) before generating a formatted string. Also, we have shifted the resulting text image 25 pixels to the left. This was necessary to ensure that the text is positioned properly. In

more complex text rendering applications, users may be required to compute the text position in a different way using `ee.Geometry` calls from the Earth Engine API: for example, by positioning text labels somewhere near the corners.

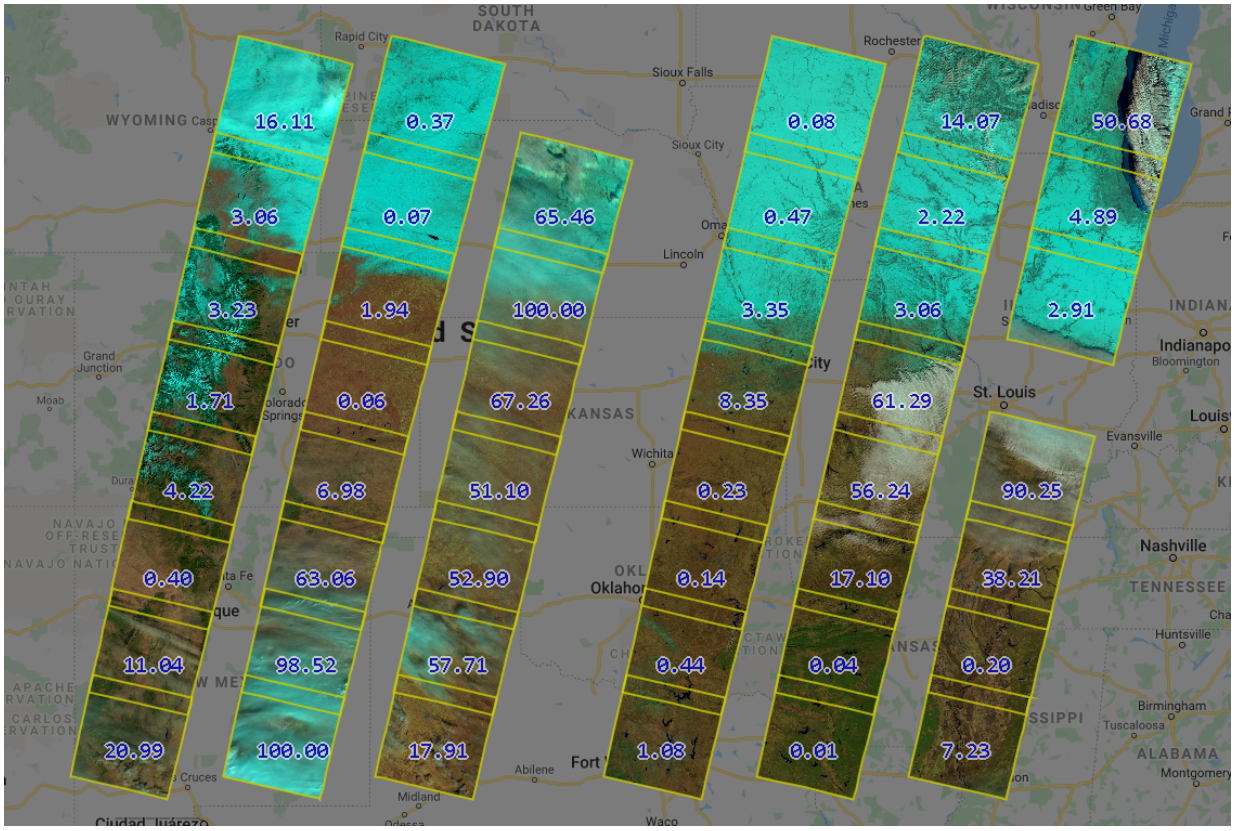


**Fig. F6.0.9** Annotating Landsat 8 images with image boundaries, border, and text strings indicating cloudiness

Because we render text labels using the Earth Engine raster API, they are not automatically scaled depending on map zoom size. This may cause unwanted artifacts; To avoid that, the text labels image needs to be updated every time the map zoom changes. To implement this in a script, we can make use of the Map API—in particular, the `Map.onChangeZoom` event handler. The following code snippet shows how the image containing text annotations can be re-rendered every time the map zoom changes. Add it to the end of your script.

```
// re-render (rescale) annotations when map zoom changes.
```

```
Map.onChangeZoom(function(zoom) {
    labelsLayer.setEeObject(images.map(annotate));
});
```

**Code Checkpoint F60f.** The book's repository contains a script that shows what your code should look like at this point.

Try commenting that event handler and observe how annotation rendering changes when you zoom in or zoom out.

## ***Section 4. Animations***

Visualizing raster images as animations is a useful technique to explore changes in time-dependent datasets, but also, to render short animations to communicate how changing various parameters affects the resulting image—for example, varying thresholds of spectral indices resulting in different binary maps or the changing geometry of vector features.

Animations are very useful when exploring satellite imagery, as they allow viewers to quickly comprehend dynamics of changes of earth surface or atmospheric properties. Animations can also help to decide what steps should be taken next to designing a robust algorithm to extract useful information from satellite image time series. Earth Engine provides two standard ways to generate animations: as animated GIFs, and as AVI video clips. Animation can also be rendered from a sequence of images exported from Earth Engine, using numerous tools such as *ffmpeg* or *moviepy*. However, in many cases it is useful to have a way to quickly explore image collections as animation without requiring extra steps.

In this section, we will generate animations in three different ways:

1. Generate animated GIF
2. Export video as an AVI file to Google Drive
3. Animate image collection interactively using UI controls and map layers

We will use an image collection showing sea ice as an input dataset to generate animations with visualization parameters from earlier. However, instead of querying a single Sentinel-1 image, let's generate a filtered image collection with all images intersecting with our area of interest. After importing some packages and palettes and defining a point and rectangle, we'll build the image collection. Here we will use point

geometry to define the location where the image date label will be rendered and the rectangle geometry to indicate the area of interest for the animation. To do this we will build the following logic in a new script. Open a new script and paste the following code into it:

```
// Include packages.
var palettes = require('users/gena/packages:palettes');
var text = require('users/gena/packages:text');

var point = /* color: #98ff00 */ ee.Geometry.Point([-
    106.15944300895228, -74.58262940096245
]);

var rect = /* color: #d63000 */
    ee.Geometry.Polygon(
        [
            [
                [-106.19789515738981, -74.56509549360152],
                [-106.19789515738981, -74.78071448733921],
                [-104.98115931754606, -74.78071448733921],
                [-104.98115931754606, -74.56509549360152]
            ]
        ], null, false);

// Lookup the ice palette.
var palette = palettes.cmocean.Ice[7];

// Show it in the console.
palettes.showPalette('Ice', palette);

// Center map on geometry.
Map.centerObject(point, 9);

// Select S1 images for the Thwaites glacier.
var images = ee.ImageCollection('COPERNICUS/S1_GRD')
    .filterBounds(rect)
    .filterDate('2021-01-01', '2021-03-01')
    .select('HH')
    // Make sure we include only images which fully contain the region
geometry.
```

```
    .filter(ee.Filter.isContained({
        leftValue: rect,
        rightField: '.geo'
    }))
    .sort('system:time_start');

// Print number of images.
print(images.size());
```

As you see from the last last lines of the above code, it is frequently useful to print the number of images in an image collection: an example of what's often known as a "sanity check."

Here we have used two custom geometries to configure animations: the green pin named `point`, used to filter image collection and to position text labels drawn on top of the image, and the blue rectangle `rect`, used to define a bounding box for the exported animations. To make sure that the point and rectangle geometries are shown under the Geometry Imports in the Code Editor, you need to click on these variables in the code and then select the **Convert** link.

Notice that in addition to the bounds and date filter, we have also used a less known `isContained` filter to ensure that we get only images that fully cover our region. To better understand this filter, you could try commenting out the filter and compare the differences, observing images with empty (masked) pixels in the resulting image collection.

**Code Checkpoint F60g.** The book's repository contains a script that shows what your code should look like at this point.

Next, to simplify the animation API calls, we will generate a composite RGB image collection out of satellite images and draw the image's acquisition date as a label on every image, positioned within our region geometry.

```
// Render images.
var vis = {
    palette: palette,
    min: -15,
    max: 1
};
```

```
var scale = Map.getScale();
var textProperties = {
    outlineColor: '000000',
    outlineWidth: 3,
    outlineOpacity: 0.6
};

var imagesRgb = images.map(function(i) {
    // Use the date as the label.
    var label = i.date().format('YYYY-MM-dd');
    var labelImage = text.draw(label, point, scale,
        textProperties);

    return i.visualize(vis)
        .blend(labelImage) // Blend label image on top.
        .set({
            label: label
        }); // Keep the text property.
});
Map.addLayer(imagesRgb.first());
Map.addLayer(rect, {color:'blue'}, 'rect', 1, 0.5);
```

In addition to printing the size of the `ImageCollection`, we also often begin by adding a single image to the map from a mapped collection to see that everything works as expected—another example of a sanity check. The resulting map layer will look like F6.0.10.

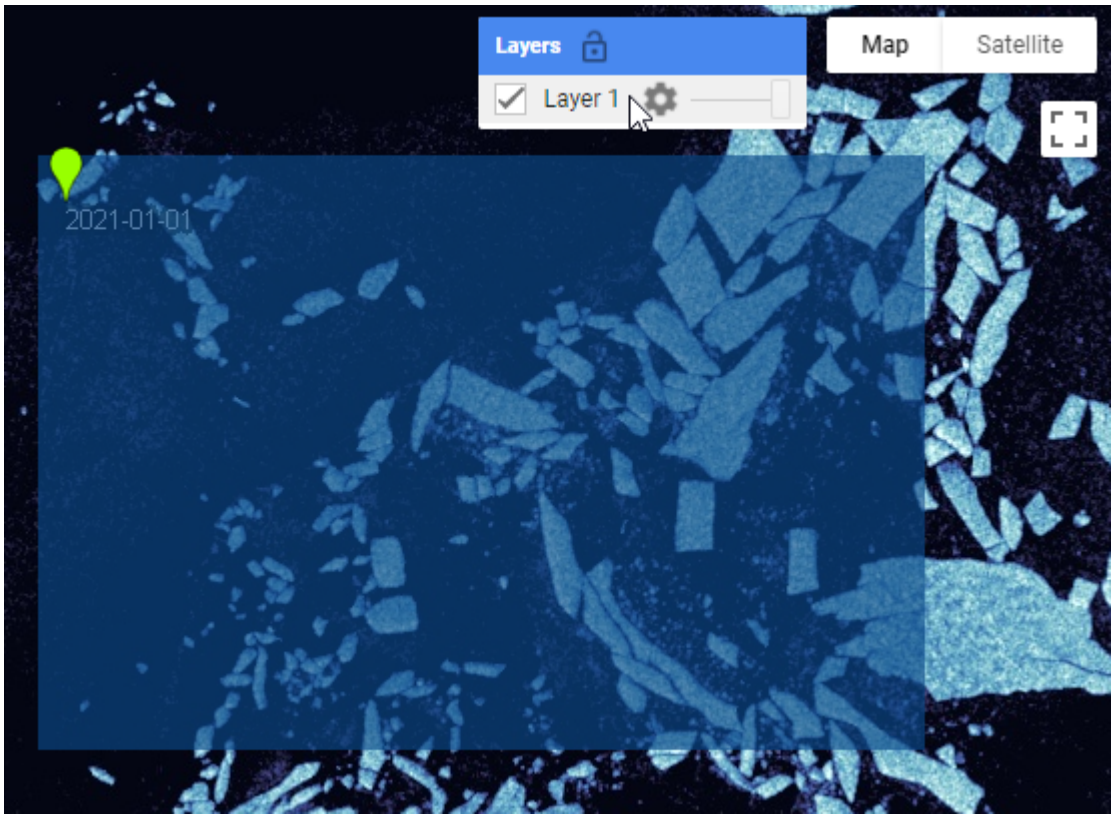


**Fig. F6.0.10** The results of adding the first layer from the RGB composite image collection showing Sentinel-1 images with a label blended on top at a specified location. The blue geometry is used to define the bounds for the animation to be exported.

**Code Checkpoint F60h.** The book's repository contains a script that shows what your code should look like at this point.

**Animation 1: Animated GIF with ui.Thumbnail**

The quickest way to generate an animation in Earth Engine is to use the animated GIF API and either print it to the **Console** or print the URL to download the generated GIF. The following code snippet will result in an animated GIF as well as the URL to the animated GIF printed to **Console**. This is as shown in Fig. F6.0.11:

```
// Define GIF visualization parameters.
var gifParams = {
    region: rect,
    dimensions: 600,
```

```
    crs: 'EPSG:3857',
    framesPerSecond: 10
};

// Print the GIF URL to the console.
print(imagesRgb.getVideoThumbURL(gifParams));

// Render the GIF animation in the console.
print(ui.Thumbnail(imagesRgb, gifParams));
```

Earth Engine provides multiple options to specify the size of the resulting video. In this example we specify 600 as the size of the maximum dimension. We also specify the number of frames per second for the resulting animated GIF as well as the target projected coordinate system to be used (EPSG:3857 here, which is the projection used in web maps such as Google Maps and the Code Editor background).

https://earthengine.googleapis.com/v1alpha/projects/earthengine-legacy/videoThu… JSON

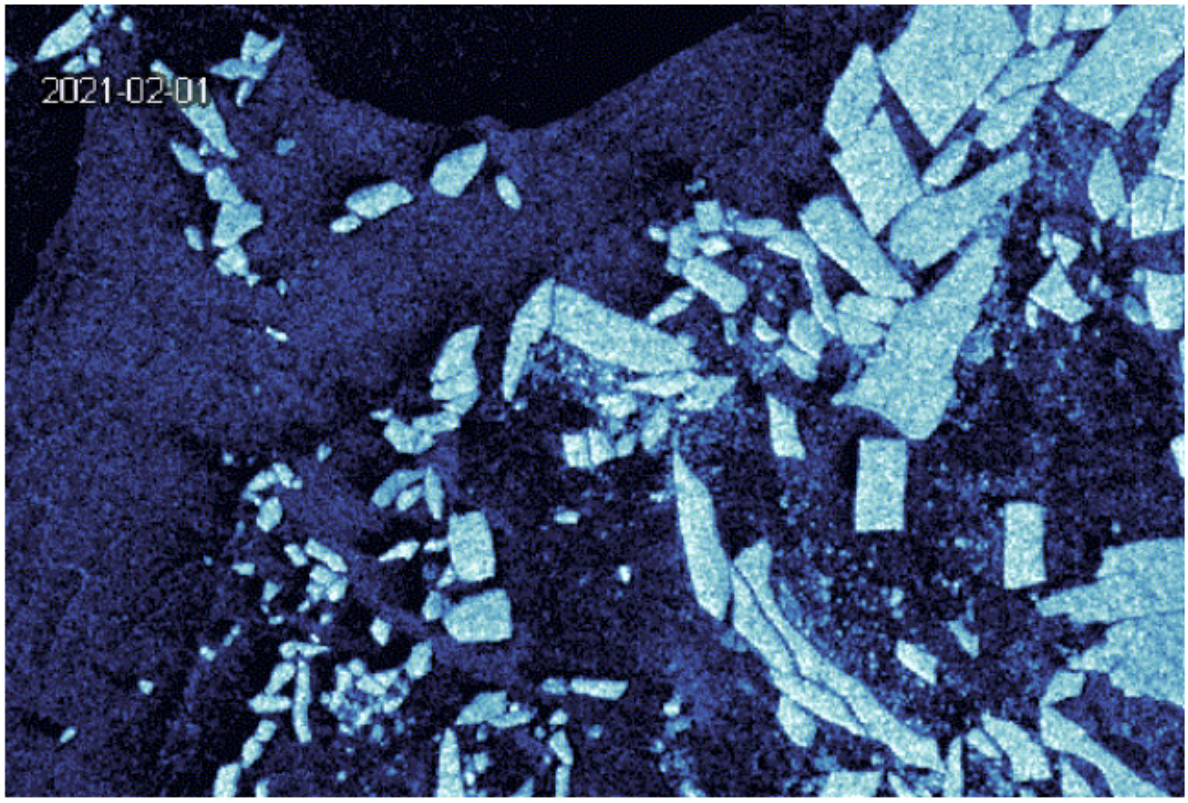

**Fig. F6.0.11 Console** output after running the animated GIF code snippet, showing the GIF URL and an animation shown directly in the **Console**

**Animation 2: Exporting an Animation with Export.video.toDrive**

Animated GIFs can be useful to generate animations quickly. However, they have several limitations. In particular, they are limited to 256 colors, become large for larger animations, and most web players do not provide play controls when playing animated GIFs. To overcome these limitations, Earth Engine provides export of animations as video files in MP4 format. Let's use the same RGB image collection we have used for the animated GIF to generate a short video. We can ask Earth Engine to export the video to the Google Drive using the following code snippet:

```
Export.video.toDrive({
    collection: imagesRgb,
    description: 'ice-animation',
    fileNamePrefix: 'ice-animation',
    framesPerSecond: 10,
    dimensions: 600,
    region: rect,
    crs: 'EPSG:3857'
});
```

Here, many arguments to the `Export.video.toDrive` function resemble the ones we've used in the `ee.Image.getVideoThumbURL` code above. Additional arguments include description and fileNamePrefix, which are required to configure the name of the task and the target file of the video file to be saved to Google Drive. Running the above code will result in a new task created under the Tasks tab in the Code Editor. Starting the export video task (F6.0.12) will result in a video file saved in the Google Drive once completed.

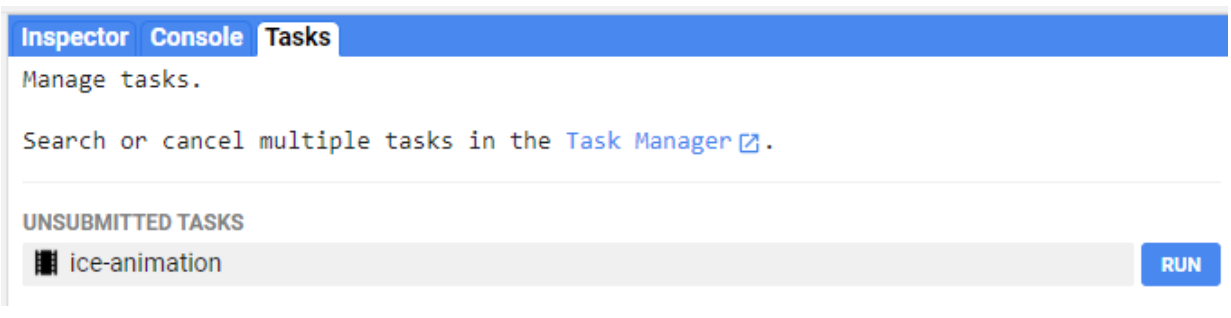

**Fig. F6.0.12** A new export video tasks in the Tasks panel of the Code Editor

**Animation 3: The Custom Animation Package**

For the last animation example, we will use the custom package `'users/gena/packages:animation'`, built using the Earth Engine User Interface API. The main difference between this package and the above examples is that it generates an interactive animation by adding Map layers individually to the layer set, and providing UI controls that allow users to play animations or interactively switch between frames. The `animate` function in that package generates an interactive animation of an `ImageCollection`, as described below. This function has a number of optional arguments allowing you to configure, for example, the number of frames to be animated, the number of frames to be preloaded, or a few others. The optional parameters to control the function are the following:

- *maxFrames*: maximum number of frames to show (default: 30)
- *vis*: visualization parameters for every frame (default: {})
- *Label*: text property of images to show in the animation controls (default: undefined)
- *width*: width of the animation panel (default: '600px')
- *compact*: show only play control and frame slider (default: false)
- *position*: position of the animation panel (default: 'top-center')
- *timeStep*: time step (ms) used when playing animation (default: 100)
- *preloadCount*: number of frames (map layers) to preload (default: all)

Let's call this function to add interactive animation controls to the current Map:

```
// include the animation package
var animation = require('users/gena/packages:animation');

// show animation controls
animation.animate(imagesRgb, {
  label: 'label',
  maxFrames: 50
});
```

Before using the interactive animation API, we need to include the corresponding package using `require`. Here we provide our pre-rendered image collection and two optional parameters (`label` and `maxFrames`). The first optional parameter label indicates that every image in our image collection has the `'label'` text property. The `animate`

function uses this property to name map layers as well as to visualize in the animation UI controls when switching between frames. This can be useful when inspecting image collections. The second optional parameter, `maxFrames`, indicates that the maximum number of animation frames that we would like to visualize is 50. To prevent the Code Editor from crashing, this parameter should not be too large: it is best to keep it below 100. For a much larger number of frames, it is better to use the Export video or animated GIF API. Running this code snippet will result in the animation control panel added to the map as shown in Fig. F6.0.13.

It is important to note that the animation API uses asynchronous UI calls to make sure that the Code Editor does not hang when running the script. The drawback of this is that for complex image collections, a large amount of processing is required. Hence, it may take some time to process all images and to visualize the interactive animation panel. The same is true for map layer names: they are updated once the animation panel is visualized. Also, map layers used to visualize individual images in the provided image collection may require some time to be rendered.


**Fig. F6.0.13** Interactive animation controls when using custom animation API

The main advantage of the interactive animation API is that it provides a way to explore image collections at frame-by-frame basis, which can greatly improve our visual understanding of the changes captured in sets of images.

**Code Checkpoint F60i.** The book's repository contains a script that shows what your code should look like at this point.

## ***Section 5. Terrain Visualization***

This section introduces several raster visualization techniques useful to visualize terrain data such as:

- Basic hillshading and parameters (light azimuth, elevation)
- Combining elevation data and colors using HSV transform (Wikipedia: HSL and HSV)
- Adding shadows

One special type of raster data is data that represents height. Elevation data can include topography, bathymetry, but also other forms of height, such as sea surface height can be presented as a terrain.

Height is often visualized using the concept of directional light with a technique called hillshading. Because height is such a common feature in our environment, we also have an expectancy of how height is visualized. If height is visualized using a simple grayscale colormap, it looks very unnatural (Fig. F6.0.14, top left). By using hillshading, data immediately looks more natural (Fig. F6.0.14, top middle).

We can further improve the visualization by including shadows (Fig. F6.0.14, top right). A final step is to replace the simple grayscale colormap with a **perceptual uniform topographic** colormap and mix this with the hillshading and shadows (Fig. F6.0.14, bottom). This section explains how to apply these techniques.

We'll focus on elevation data stored in raster form. Elevation data is not always stored in raster formats. Other data formats include Triangulated Irregular Network (TIN), which allows storing information at varying resolutions and as 3D objects. This format allows one to have overlapping geometries, such as bridges with a road below it. In raster-based digital elevation models, in contrast, there can only be one height recorded for each pixel.

Let's start by loading data from a digital elevation model. This loads a topographic dataset from the Netherlands (Algemeen Hoogtebestand Nederland). It is a Digital Surface Model, based on airborne LIDAR measurements regridded to 0.5 m resolution. Enter the following code in a new script.

```
var dem = ee.Image('AHN/AHN2_05M_RUW');
```

We can visualize this dataset using a sequential gradient colormap from black to white. This results in Fig. F6.0.14. One can infer which areas are lower and which are higher, but the visualization does not quite “feel” like a terrain.

```
// Change map style to HYBRID and center map on the Netherlands
Map.setOptions('HYBRID');
Map.setCenter(4.4082, 52.1775, 18);

// Visualize DEM using black-white color palette
var palette = ['black', 'white'];
var demRGB = dem.visualize({
    min: -5,
    max: 5,
    palette: palette
});
Map.addLayer(demRGB, {},'DEM');
```

An important step to visualize terrain is to add shadows created by a distant point source of light. This is referred to as hillshading or a shaded relief map. This type of map became popular in the 1940s through the work of Edward Imhoff, who also used grayscale colormaps (Imhoff, 2015). Here we’ll use the `'gena/packages:utils'` library to combine the colormap image with the shadows. That Earth Engine package implements a `hillshadeRGB` function to simplify rendering of images enhanced with hillshading and shadow effects. One important argument this function takes is the light azimuth—an angle from the image plane upward to the light source (the Sun). This should always be set to the top left to avoid bistable perception artifacts, in which the DEM can be misperceived as inverted.

```
var utils = require('users/gena/packages:utils');

var weight =
    0.4; // Weight of Hillshade vs RGB (0 - flat, 1 - hillshaded).
var exaggeration = 5; // Vertical exaggeration.
var azimuth = 315; // Sun azimuth.
var zenith = 20; // Sun elevation.
var brightness = -0.05; // 0 - default.
var contrast = 0.05; // 0 - default.
```

```
var saturation = 0.8; // 1 - default.
var castShadows = false;

var rgb = utils.hillshadeRGB(
    demRGB, dem, weight, exaggeration, azimuth, zenith,
    contrast, brightness, saturation, castShadows);

Map.addLayer(rgb, {}, 'DEM (no shadows)');
```

Standard hillshading only determines per pixel if it will be directed to the light or not. One can also project shadows on the map. That is done using the `ee.Algorithms.HillShadow` algorithm. Here we'll turn on `castShadows` in the `hillshadeRGB` function. This results in a more realistic map, as can be seen in Figure F6.0.14.

```
var castShadows = true;

var rgb = utils.hillshadeRGB(
    demRGB, dem, weight, exaggeration, azimuth, zenith,
    contrast, brightness, saturation, castShadows);

Map.addLayer(rgb, {}, 'DEM (with shadows)');
```

The final step is to add a topographic colormap. To visualize topographic information, one often uses special topographic colormaps. Here we'll use the `oleron` colormap from `crameri`. The colors get mixed with the shadows using the `hillshadeRGB` function. As you can see in Fig. F6.0.14, this gives a nice overview of the terrain. The area colored in blue is located below sea level.

```
var palettes = require('users/gena/packages:palettes');
var palette = palettes.crameri.oleron[50];

var demRGB = dem.visualize({
    min: -5,
    max: 5,
    palette: palette
});

var castShadows = true;
```

```
var rgb = utils.hillshadeRGB(
    demRGB, dem, weight, exaggeration, azimuth, zenith,
    contrast, brightness, saturation, castShadows);

Map.addLayer(rgb, {}, 'DEM colormap');
```

Steps to further improve a terrain visualization include using light sources from multiple directions. This allows the user to render terrain to appear more natural. In the real world light is often scattered by clouds and other reflections.

One can also use lights to emphasize certain regions. To use even more advanced lighting techniques one can use a raytracing engine, such as the R *rayshader* library, as discussed earlier in this chapter. The raytracing engine in the Blender 3D program is also capable of producing stunning terrain visualizations using physical-based rendering, mist, environment lights, and camera effects such as depth of field.

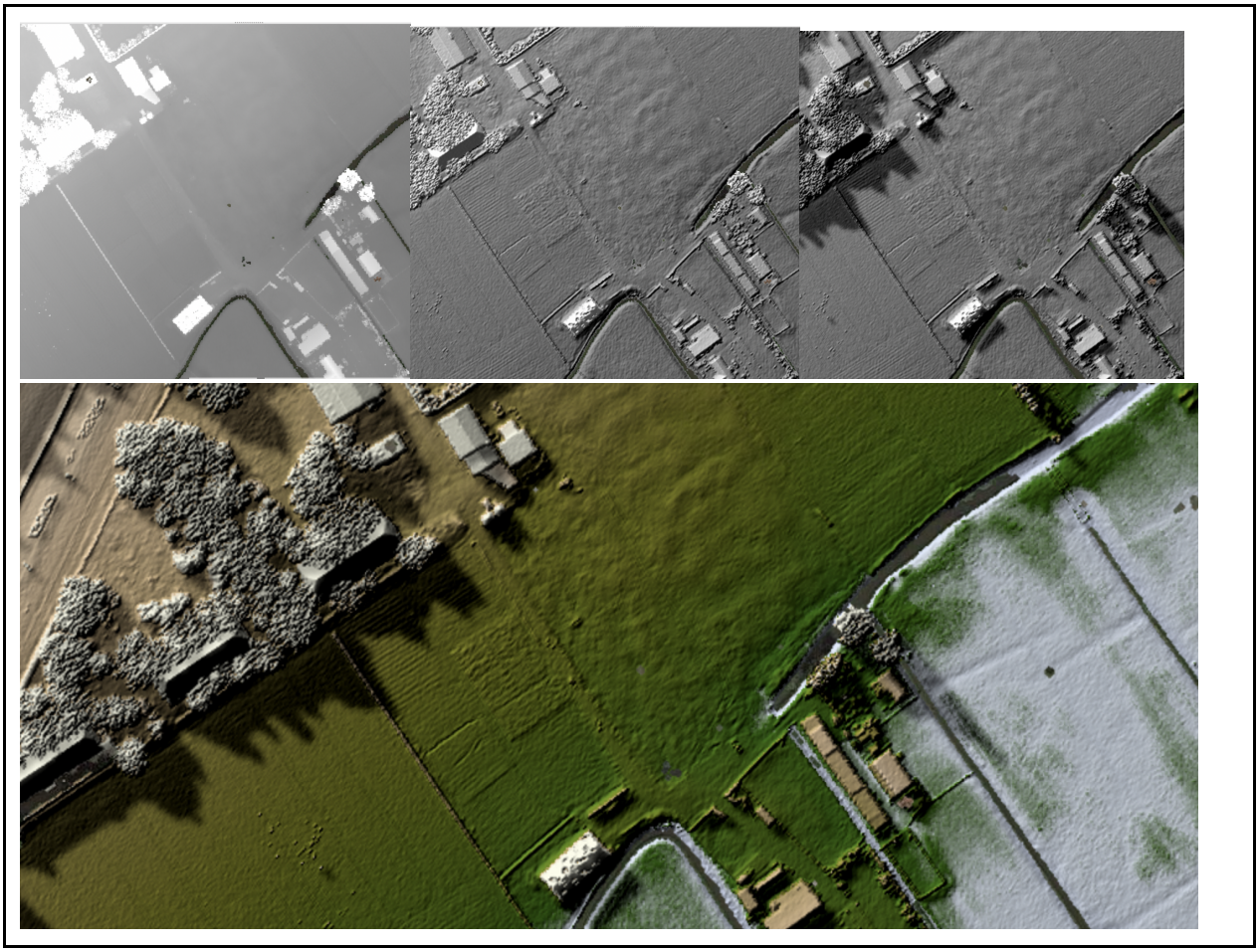

**Figure F6.0.14** Hillshading with shadows

Steps in visualizing a topographic dataset:
1) Top left, topography with grayscale colormap
2) Top middle, topography with grayscale colormap and hillshading
3) Top right, topography with grayscale colormap, hillshading, and shadows
4) Bottom, topography with topographic colormap, hillshading, and shadows

**Code Checkpoint F60j.** The book's repository contains a script that shows what your code should look like at this point.

## Synthesis

To synthesize what you have learned in this chapter, you can do the following assignments.

**Assignment 1.** Experiment with different color palettes from the `palettes` library. Try combining palettes with image opacity (using `ee.Image.updateMask` call) to visualize different physical features (for example, hot or cold areas using temperature and elevation).

**Assignment 2.** Render multiple text annotations when generating animations using image collection. For example, show other image properties in addition to date or image statistics generated using regional reducers for every image.

**Assignment 3.** In addition to text annotations, try blending geometry elements (lines, polygons) to highlight specific areas of rendered images.

## Conclusion

In this chapter we have learned about several techniques that can greatly improve visualization and analysis of images and image collections. Using predefined palettes can help to better comprehend and communicate Earth observation data, and combining with other visualization techniques such as hillshading and annotations can help to better understand processes studied with Earth Engine. When working with image collections, it is often very helpful to analyze their properties through time by visualizing them as animations. Usually, this step helps to better understand dynamics of the changes that are stored in image collections and to develop a proper algorithm to study these changes.

## Feedback

To review this chapter and make suggestions or note any problems, please go now to bit.ly/EEFA-review. You can find summary statistics from past reviews at bit.ly/EEFA-reviews-stats.

## References


Burrough PA, McDonnell RA, Lloyd CD (2015) Principles of Geographical Information Systems. Oxford University Press

Crameri F, Shephard GE, Heron PJ (2020) The misuse of colour in science communication. Nat Commun 11:1–10. https://doi.org/10.1038/s41467-020-19160-7

Imhof E (2015) Cartographic Relief Presentation. Walter de Gruyter GmbH & Co KG

Lhermitte S, Sun S, Shuman C, et al (2020) Damage accelerates ice shelf instability and mass loss in Amundsen Sea Embayment. Proc Natl Acad Sci USA 117:24735–24741. https://doi.org/10.1073/pnas.1912890117

Thyng KM, Greene CA, Hetland RD, et al (2016) True colors of oceanography. Oceanography 29:9–13

Wikipedia (2022) Terrain cartography. https://en.wikipedia.org/wiki/Terrain_cartography#Shaded_relief. Accessed 1 Apr 2022

Wikipedia (2022) HSL and HSV. https://en.wikipedia.org/wiki/HSL_and_HSV. Accessed 1 Apr 2022

Wild CT, Alley KE, Muto A, et al (2022) Weakening of the pinning point buttressing Thwaites Glacier, West Antarctica. Cryosphere 16:397–417. https://doi.org/10.5194/tc-16-397-2022

Wilkinson L (2005) The Grammar of Graphics. Springer Verlag