TNM067 — Scientific Visualization
1. Interpolation, Colormaps and Heightfields

August 30, 2021

1 Introduction

This laboratory exercise will introduce interpolation methods and color maps. At the end of this lab, you will have hands-on experience with implementing and using Heightfield visualizations (Part 1) and Color Mapping for 2D Scalar visualization (Part 2). Finally, you will implement various *interpolations methods* and apply them to up-sample images (Part 3).

1.1 Change log

- 2017-09-01: Initial Version
- 2017-09-04: Improved description of bilinear interpolation, removed duplicates
- 2020-09-02: Update for 2020, Update for distance teaching.

2 Documentation

On the following links you can read more about Inviwo and get help with questions related to Inviwo.

- Inviwo: http://www.inviwo.org
- Inviwo API Documentation: https://inviwo.org/inviwo/doc/
- Inviwo Issue tracker: https://github.com/inviwo/inviwo/issues

3 Part 1 - Heightfield Visualization

A common visualization technique for 2D scalar fields are heightfields, in which the height corresponds to the scalar value of the field at the chosen point. In this lab we will only be using 2D Scalar fields define on a structured grid, hence our 2D scalar fields are analogous to an image with one channel. In Figure 1, this concept is illustrated. On the left we have a box, which corresponds to one pixel. The height of the box is the pixel intensity for that grid cell. On the right, a small grid is shown to illustrate a 2D view of the heightfield.

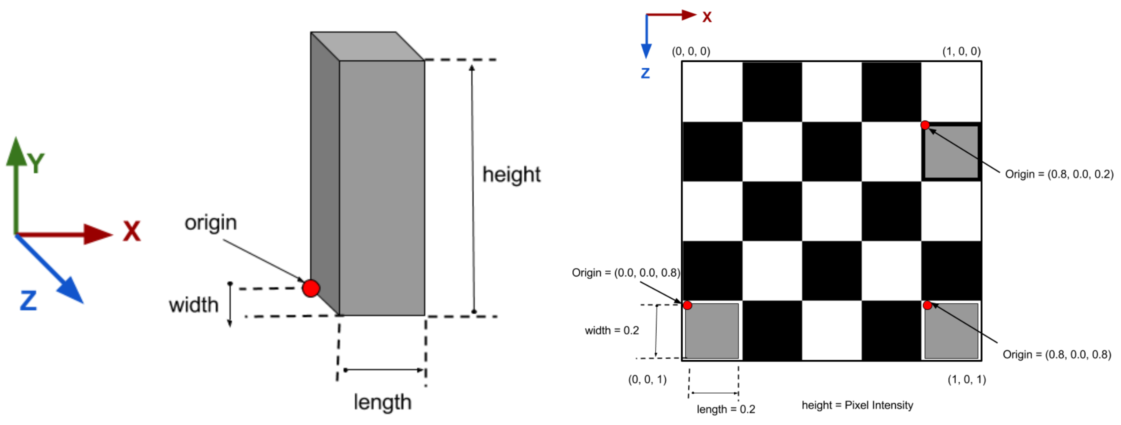


Figure 1: A heightfield is represented using one box per pixel. The intensity of a pixel in the image corresponds to the height of the box placed on that pixel. The dimension of a box is shown on the left, while the placement can be seen on the right.

In the provided lab-code there is a Inviwo processor called *ImageToHeightfield* which takes an Image as an input and outputs a mesh. Your task, described bellow, will be to update the code to map the intensity values of the image to the height of the boxes in the output mesh.

When implementing heightfield visualizations, there are different ways of connecting the neighboring cells. In this lab, we will map each pixel to a box, but another common way is to let each pixel in the input image represent one vertex in the output, mapping the pixel value directly to the height of the vertex. Our way will produce a visualization that is analogous to a 2D histogram or 2D bar chart, while the other approach would use interpolation between vertices to create a smoother visualization.

Task 1 — Modify the height of the height field:

- File : “modules/tnm067lab1/processors/imagetoheightfield”
- Workspace: *Example Workspaces > TNM067Lab1 > lab1part1−heightfield.ivw*

Modify the code in ImageToHeightfield::buildMesh() to not use the default 0 but instead read values from the input image. Build Inviwo again in Visual Studio and look at the results.

Sometimes it might be useful to decrease or increase the height difference between the lowest and highest box in the height field.

Task 2 — Allow for changing the scale of the heightfield:
The processor has a property called heightScaleFactor_. In the function ImageToHeightfield::buildMesh(), scale the height with this property.
Note: This should not change the color.

4 Part 2 - Mapping Scalar Values To Color

The output from the previous task is just an illuminated surface. This can be boring to look at and hard to perceive differences. Color mapping can be used in addition to height to enhance the visualization. In general a color mapping algorithm takes in one or more values and outputs a color. In our case, we use the scalar values from the pixels in the image.

In this part of the lab you will implement color mapping schemes using linear interpolation. Linear interpolation is commonly used in visualization, and besides color mapping can be used to approximate values for points in-between grid points and to help build smooth surfaces, and more.

You will mainly be working with the file "modules/tnm067lab1/utils/scalartocolormapping.cpp". Before you begin the task try to understand the member variables and member functions in the header file of the class *ScalarToColorMapping*. The *sample* method which you will implement in *ScalarToColorMapping*, can be used by two processors in our lab files: *ImageMappingCPU* and *ImageToHeightField*. The functionality of the first processor is to take a gray-scale image and output a RGBA image of the colors selected in the Property panel of Inviwo. This is done by using the *sample* method, but right now it only provides blank results. The second processor, as you know, creates a heightfield from a gray-scale image. The color of the heightfield can be set by using the *sample* method, but this processor requires some more modifications.

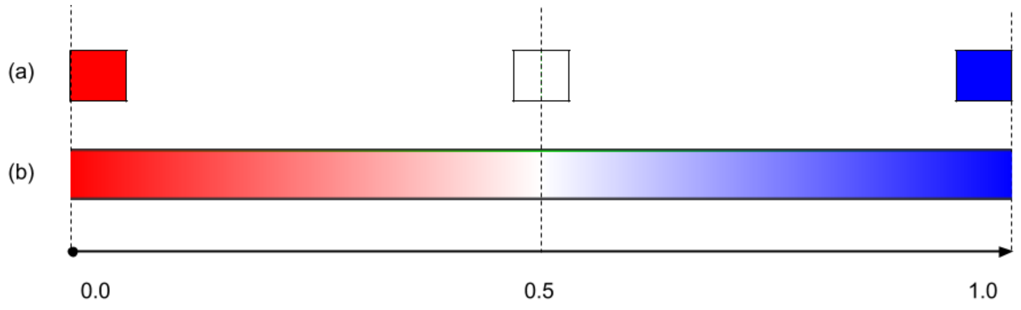


Figure 2: Scalar Color map illustration. (a) Colormap with base colors red, white and blue. (b) Linear Interpolation between base colors red, white, blue.

Task 3 — Linear interpolation of colors:

- Workspace: *Example Workspaces* > *TNM067Lab1* > *lab1part1−colormapping.ivw*

Implement *ScalarToColorMapping::sample* method, which takes a scalar t as input and provides an RGBA-vector as output.

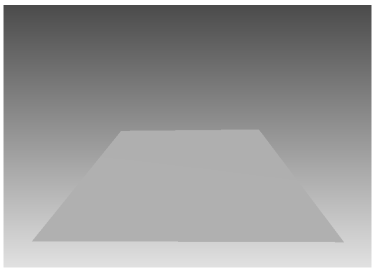
- Given a scalar value $t \in [0, 1]$, calculate which two colors from the baseColors_ vector should be used in the interpolation. An illustration is seen in Figure 2, where $t < 0.5$ means interpolating between red and white, while $t > 0.5$ results in interpolating between white and blue.
- Calculate the interpolation factor and use linear interpolation between the two selected base colors. Note that t cannot be used directly as an interpolation factor.

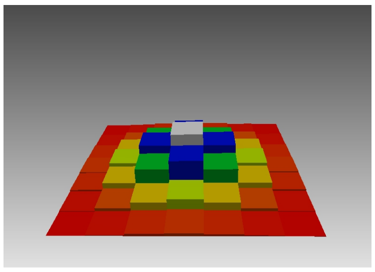
  Note: Your implementation should work for any number of given base colors. After completing this task you should be able to see the results in the workspace by looking at the canvas connected to the *ImageMappingCPU* processor.

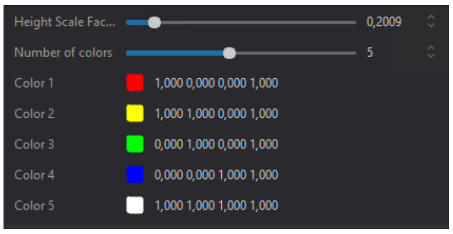
Task 4 — Color the heightfield:

Update the code in *ImageToHeightfield* to use the *ScalarToColorMapping* to change the color of the output mesh. There is a local object called *map* in the *buildMesh* function, which contains the color map of the colors defined in the Property panel of the processor in Inviwo. This object can be used to sample the color map.
Note: Change the scaling-slider should not change the color.

After all tasks up until Task 4 have been implemented correctly, you should have the results in Figure 3 using the same colors.

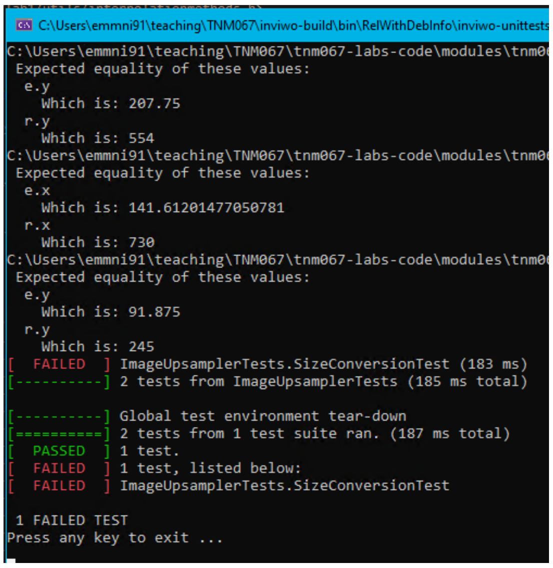

(a)


(b)


Height Scale Fac... 0,2009
Number of colors 5
Color 1 🟥 1,000 0,000 0,000 1,000
Color 2 🟨 1,000 1,000 0,000 1,000
Color 3 🟩 0,000 1,000 0,000 1,000
Color 4 🟦 0,000 0,000 1,000 1,000
Color 5 ⬜ 1,000 1,000 1,000 1,000

(c)

Figure 3: Initial view and possible results after task 4 using the colors in c).

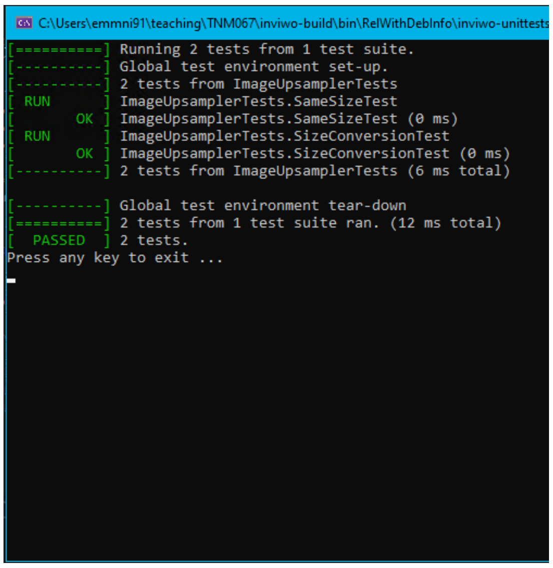


```
C:\Users\emmni91\teaching\TNM067\inviwo-build\bin\RelWithDebInfo\inviwo-unittests
C:\Users\emmni91\teaching\TNM067\tnm067-labs-code\modules\tnm0
 Expected equality of these values:
   e.y
     Which is: 207.75
   r.y
     Which is: 554
C:\Users\emmni91\teaching\TNM067\tnm067-labs-code\modules\tnm0
 Expected equality of these values:
   e.x
     Which is: 141.61201477050781
   r.x
     Which is: 730
C:\Users\emmni91\teaching\TNM067\tnm067-labs-code\modules\tnm0
 Expected equality of these values:
   e.y
     Which is: 91.875
   r.y
     Which is: 245
[  FAILED  ] ImageUpsamplerTests.SizeConversionTest (183 ms)
[----------] 2 tests from ImageUpsamplerTests (185 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (187 ms total)
[  PASSED  ] 1 test.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] ImageUpsamplerTests.SizeConversionTest

 1 FAILED TEST
Press any key to exit ...
```

(a)



```
C:\Users\emmni91\teaching\TNM067\inviwo-build\bin\RelWithDebInfo\inviwo-unittests
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from ImageUpsamplerTests
[ RUN      ] ImageUpsamplerTests.SameSizeTest
[       OK ] ImageUpsamplerTests.SameSizeTest (0 ms)
[ RUN      ] ImageUpsamplerTests.SizeConversionTest
[       OK ] ImageUpsamplerTests.SizeConversionTest (0 ms)
[----------] 2 tests from ImageUpsamplerTests (6 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (12 ms total)
[  PASSED  ] 2 tests.
Press any key to exit ...
```

(b)

Figure 4: Example of the output from the unittests before and after Task 5 has been correctly implemented.

5 Part 3 - Data Interpolation

In this part of the lab you will implement various interpolation schemes such as piecewise constant, bilinear, quadratic and barycentric interpolation on grayscale image data. Your task is to up-sample the grayscale image. For this task, you will be implementing your code in "modules/tnm067lab1/processors/imageupsampler.cpp".

Before implementing the tasks below read the following:

- Make sure you understand the interpolation techniques that you are about to implement. Read the lecture notes on "Data Representation and Interpolation" that are available on Lisam.
- The output image size is controlled by the size of the outport, which is, in most cases controlled by the canvas.
- Your upscaling implementation should work for any output size, but you can assume the output size is always equal or larger than the input size.

The tasks

The following task will be implemented in two different files. The *ImageUpsampler* class in *processors/imageupsampler.cpp* contains the code for creating the new image and reading/writing pixel values. The namespace *TNM067::Interpolation* in file *utils/interpolationmethods.h* will contain the code for the interpolation logic.

You will be switching between **two start-up projects** in Visual Studio: inviwo and inviwo-unittests-tnm067lab1. The target inviwo is the default target which you have been using up until now, and is used for inspecting visual results. The second target runs pre-implemented unittests which tests the implementation of the methods in the following tasks. The tests can be run by changing the start-up target in Visual Studio [1]. When running the unittests target you will see whether your code is correctly implemented or not in the output console windows, an example of which you can see in Figure 4.

Task 5 — Coordinate conversion:

- Startup target : unittests/inviwo-uittests-tnm067lab1

Implement the method *ImageUpsampler::convertCoordinates* in *processors/imageupsampler.cpp*. The method takes a set of pixel coordinates in the output image and should rescale them to be in the input image coordinate system. Since the output image is larger than the input image, the returned coordinates may be between pixels in the input image, consequently the calculated coordinates must be of the type double.

Task 6 — Piecewise constant interpolation:

- Startup target : inviwo
- Workspace: *Example Workspaces > TNM067Lab1 > lab1part1-upsampling.ivw*

Implement piecewise constant interpolation in the method *upsample*. Carefully read the method first to understand its functionality and variables. Note that there is a function called inIndex in the method, which you can use to calculate pixel index from image coordinates. You should implement the interpolation in the correct switch case in *ImageUpsampler.cpp*, around line 48. Look at the top-left corner of the canvas connected to the *Image Layout* processor in the workspace to inspect your results.

Bilinear Interpolation

Bilinear interpolation can be seen as a box filter moving over the image. Given a certain pixel location we look at the color of the four nearest pixels and weigh according to distance from current pixel. See Figure 5 as an example, the black dot represents the center of the pixel in the upscaled image and the four corners represent the colors to be used in the interpolation. The area of the color under the new pixel is proportional to its contribution, in this case red will contribute the most while yellow, blue and green will contribute less. Using bilinear interpolation, the colors can be mixed together along the box lines to get the final ooutput col.

[1]To change start-up project, Right-click the project in the Solution Explorer and set it as Start-up project.

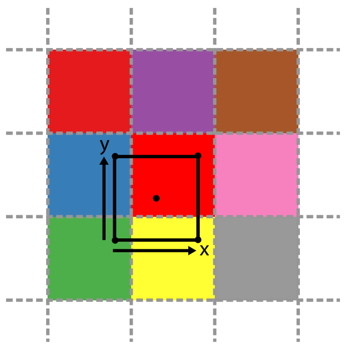


Figure 5: An illustration to exemplify bilinear interpolation. The underlying grid is the input image, while the black box represents a pixel in the output image. Using bilinear interpolation, the colors of the underlying image is interpolated along the box edges to get the color output of the upsampled image. In the example, the center point is located so that red will contribute the most to the output color.

Task 7 — Bilinear Interpolation:
A:

- Startup target : unittests/inviwo-uittests-tnm067lab1

First implement the linear interpolation function in *utils/interpolationmethods.h* then use that function to implement bilinear interpolation.
To enable the unittest for each of these methods change the define above the methods from 0 to 1.
B:

- Startup target : inviwo-cli

Update the code in *imageupsampler.cpp* to also support Bilinear upsampling.

Task 8 — Quadratic interpolation:
A:

- Startup target : unittests/inviwo-uittests-tnm067lab1

First implement the quadratic interpolation function in *utils/interpolationmethods.h* then use that function to implement biquadratic interpolation.
To enable the unittest for each of these methods change the define in the code from 0 to 1.
B:

- Startup target : inviwo

Update the code in *imageupsampler.cpp* to also support biquadratic upsampling.

Task 9 — Barycentric Interpolation:
A:

- Startup target : unittests/inviwo-uittests-tnm067lab1

Implement the barycentric interpolation function in *utils/interpolationmethods.h*
In contrast to the biLinear and biQuadratic functions the barycentric can't be written as a combination of the 1d version.
To enable the unittest for each of these methods change the define in the code from 0 to 1.
B:

- Startup target : inviwo

Update the code in *imageupsampler.cpp* to also support Barycentric upsampling.
See: https://en.wikipedia.org/wiki/Barycentric_coordinate_system