LatticeLibrary and BccFccRaycaster: Software for processing and viewing 3D data on optimal sampling lattices

Elisabeth Schold Linner*, Max Morén, Karl-Oskar Smed, Johan Nysjö, Robin Strand

Div. of Visual Information and Interaction, Department of Information Technology, Uppsala University, Box 337, 751 05 Uppsala, Sweden

Received 15 October 2015; received in revised form 21 January 2016; accepted 22 January 2016

Abstract

In this paper, we present LatticeLibrary, a C++ library for general processing of 2D and 3D images sampled on arbitrary lattices. The current implementation supports the Cartesian Cubic (CC), Body-Centered Cubic (BCC) and Face-Centered Cubic (FCC) lattices, and is designed to facilitate addition of other sampling lattices. We also introduce BccFccRaycaster, a plugin for the existing volume renderer Voreen, making it possible to view CC, BCC and FCC data, using different interpolation methods, with the same application. The plugin supports nearest neighbor and trilinear interpolation at interactive frame rates. These tools will enable further studies of the possible advantages of non-Cartesian lattices in a wide range of research areas.

© 2016 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

Keywords: Optimal sampling lattices; Image processing; Volume raycasting; C++; GLSL

1. Motivation and significance

1.1. Scientific background

Since the introduction of the digital camera, digital images have become a very valuable medium for studies in a wide range of fields, such as medicine, quality control, surveillance,
robotics, and more. A digital image consists of so-called spatial elements (spels), usually referred to as pixels in 2D, and voxels in 3D. Each spel has the characteristics of position and intensity, and approximates a part of the imaged object. From these characteristics, we may derive information about the object from the collection of spels that makes up its image. It is important to keep in mind that it is usually the actual object, and not the image, that we are interested in. Thus, it is also important to understand what kind of information we can expect to derive from a digital image.

Real life objects can be assumed to be continuous, and of infinite resolution, while digital images are discrete, and of finite resolution. A digital image may thus not capture all the details of an object, sometimes resulting in amplification or suppression of some object details, as well as other image artifacts. This effect is referred to as aliasing, and is, for non-bandlimited signals, inversely proportional to the number of spels per unit area or volume; the denser the object is sampled, the smaller is the aliasing effect. In two or more dimensions, sampling density is direction dependent. As a consequence, the resolution, as well as the amount of aliasing, is different in different directions in the image. In practice, this means that for objects of a size that is distinguishable at the resolution in one direction, but not in another, it is their orientation, rather than their actual size, that determines their visibility in the image. The actual level of detail in the image, meaning the size of the smallest object that is visible independent of its orientation, is determined by the direction with the lowest resolution. However, smaller objects may still be visible, if their orientation corresponds to a direction of higher resolution. This kind of small details are therefore unreliable, and redundant, but the data representing them must be processed all the same for a certain general minimum resolution to be guaranteed in the image. The redundancy $\eta_{\text{lattice type}}$ for a specific sampling lattice is calculated in frequency domain as

$$\eta_{\text{lattice type}} = \frac{1}{\mathbb{R}_{\text{lattice type}}},$$

where $\mathbb{R}_{\text{lattice type}}$ is the packing density in frequency domain of the replicated image spectra, derived in, for example, [1]. This redundancy is $\sim 21\%$ for square spels, and $\sim 48\%$ for cubic spels. It can be minimized by using a sampling lattice with as little direction dependency as possible. In 2D, this is the hexagonal lattice, with a redundancy of $\sim 9\%$, while in 3D, it is the body-centered cubic (BCC) lattice, with a redundancy of $\sim 26\%$. The reduction of redundancy comes at a cost, though, as the optimal lattices lack the dimensional separability possessed by the square and cubic lattices, resulting in less intuitive, although not necessarily less efficient, data structures and algorithms, making the learning threshold higher for new users. It is this threshold that we wish to lower by introducing software for processing, analyzing and viewing data sampled using these lattices.

1.2. Related work

The first volume renderer for data sampled on BCC and FCC lattices is the raycaster introduced in [2]. It applies an edge detector, described in [3,4], to extract object surfaces with sub-spel precision, in order to produce more accurate surface gradients for, for example, shading. The raycasting is performed by tracing discrete lines through the volume. Code outlining the functionality of this renderer is available through the authors.

Another volume rendering method, called splatting, explained in [5], is adapted to BCC data in [1]. This paper also introduces methods for gradient reconstruction on a BCC lattice, seemingly comparable to central differences on a CC lattice. A similar method is applied in [6]. Unfortunately, none of the renderers described in these publications is available today.

The interpolation methods for BCC and FCC lattices described in [7–10] are implemented in the volume raycaster VRay [11]. This volume renderer features both a command line interface (CLI) and a graphical user interface (GUI). It is, however, a CPU implementation, and does not reach interactive frame rates. Moreover, due to dependencies on third party libraries that are no longer maintained, it does not compile in a modern environment. Some papers, like [12], features code snippets that can be used to implement the proposed methods, but do not reference a complete volume renderer.

Further interpolation methods are introduced and compared to the former ones in [13–15]. The implementations used in [14,15] reach interactive frame rates, but do not seem to be available online or through the authors.

To our knowledge, there is no software, other than LatticeLibrary, available for image processing or analysis on the BCC and FCC lattices.

1.3. Our contribution

1.3.1. LatticeLibrary

The main purpose of LatticeLibrary is to provide a framework where any processing or analysis method can be applied to data sampled on any lattice. The software architecture is designed to facilitate the addition of new methods as well as new lattices. As our research interest is the behavior of various image processing methods applied to data sampled on the BCC and FCC lattices, we use this area to demonstrate the usage of LatticeLibrary, and include these methods in the release. However, LatticeLibrary may serve as a foundation for other applications as well.

LatticeLibrary is written by the first author of this paper, under supervision by the last author.

1.3.2. BccFccRaycaster

To visualize the data being processed using the BCC and FCC support in LatticeLibrary, we introduce a plugin for the volume raycaster Voreen 3.0.1 [16]. Our implementation supports nearest neighbor interpolation and trilinear interpolation at interactive frame rates. To the best of our knowledge, it is the first open-source rendering tool that supports interactive frame rates for BCC and FCC volumes.

BccFccRaycaster is written by the second and third authors of this paper, under supervision by the other three.
2. Software description

2.1. LatticeLibrary

2.1.1. Software description

LatticeLibrary is a C++ framework for processing sampled data. The data is treated as an undirected mathematical graph, as explained in, for example, [17]. The connectivity of the graph is easily defined using the given interface, enabling the application of all implemented processing methods to data sampled on any lattice. As our field of research is image processing, the functionality featured in this release is focused in this area.

2.1.2. Software functionality

The core of LatticeLibrary is the Lattice abstract base class. A Lattice object contains the dimensionality and spatial distribution of a dataset, stored in the form of a one-dimensional data array. It does not, however, contain the actual data. If the user wishes to study the behavior of a particular lattice, a corresponding class is derived from the Lattice base class. The derived class implements methods for conversion between data indices and point coordinates, extraction of neighbors of a lattice point, and approximations of physical measurements based on the shape of the associated Voronoi cell. All other LatticeLibrary classes are required to access the data using this interface, making all functionality implemented in LatticeLibrary applicable to any lattice.

The Image class is an example of how the Lattice class can be applied. An Image object contains a reference to a Lattice object, defining how to interpret the data in a featured one-dimensional data array. To simplify usage, it supports wrappers of some of the Lattice methods, along with some functionality specific to Image objects. Our release features numerous other classes, demonstrating how the Lattice interface can be used to implement various image processing methods. At the point of publication, LatticeLibrary supports the following methods:

- Basic image arithmetic, for example addition and subtraction of two images.
- Spatial template filtering, that can be used for weighted average filters (for example, mean, Gaussian and Laplacian filters), as well as sequence based filters (for example, median filters).
- Seeded intensity-weighted distance transforms, more specifically, the minimum barrier distance [18], the approximate minimum barrier distance [19,20], fuzzy connectedness [21,22], fuzzy distance transform [23,24], and geodesic distance transform [25–27], the implementations of which are used to produce the results published in [28].
- Crisp and fuzzy segmentation, based on the output of seeded intensity-weighted distance transforms, also used in [28]. The fuzzy segmentation is an implementation of the algorithm introduced in [29].
- Anti-aliased Euclidean distance transform [30], used to produce the results of [31] and [32]. Applicable to, for example, fuzzy segmented images.

For further details, please see the Doxygen documentation for LatticeLibrary.

2.1.3. Sample code snippets

Listing 1, featured in the Appendix A, shows a minimal example of how to initialize Image objects, and Listing 2, also in Appendix A shows an example of how fuzzy image segmentation can be applied using LatticeLibrary.

2.2. BccFccRaycaster

2.2.1. Software description

BccFccRaycaster is an extension of the volume raycaster Voreen 3.0.1 [16], that enables viewing of volume data sampled on BCC and FCC lattices using nearest neighbor and trilinear interpolation. For image data sampled on BCC lattices, BccFccRaycaster supports DC-splines [33], linear box-splines [34,12], and cosine-weighted B-splines [15] for trilinear interpolation. For data sampled on FCC lattices, DC-spline interpolation [33] is supported. Nearest neighbor interpolation is supported for data sampled on CC, BCC, and FCC lattices. Renderings of the Marschner–Lobb test volume, introduced in [35], are shown in Fig. 1 as examples of nearest neighbor and trilinear interpolation on the CC, BCC, and FCC sampling lattices. Fig. 2(a) shows the frame rates achieved using nearest neighbor interpolation, and Fig. 2(b) those achieved using trilinear interpolation. The results are obtained using an Intel® CoreTM i5-3570K CPU @ 3.40 GHz, and GeForce GTX 650 Ti Boost (GK106) 2048 MB. While it is difficult to define a border between interactive and non-interactive frame rates, BccFccRaycaster is evidently able to render objects of up to 107 voxels at ~24 frames per second, which is a common frame rate for video display.

2.2.2. Software functionality

As the purpose of the software presented in this paper is to facilitate application of alternative sampling lattices to a wide range of research areas, the target group is diverse with respect to computer experience, and the learning threshold should be low. Thus, we choose to base BccFccRaycaster on an existing renderer with a GUI, as GUIs are considered to be more usable for inexperienced and infrequent users. For tasks related to computer graphics, [36] indicates that a GUI may also be beneficial for experienced programmers, further justifying our choice. However, a CLI, or equivalent, may be necessary for time consuming or repetitive tasks. Voreen 3.0.1 has a CLI in the sense that it can be used as a library in combination with GLUT. It also incorporates a Python interpreter.

BccFccRaycaster supports several different storage formats for volumes sampled on BCC and FCC lattices. The separate storage model is based on dividing the BCC and FCC lattices into two and four CC sublattices, respectively, as shown in Fig. 3. Each sublattice is stored in a separate file. At rendering, the intensity value of a spel is stored in the A channel of a texture element, and the RGB-channels are used for gradient information. The interleaved storage model allows for a BCC or FCC volume to be stored in a single file, but does not support inclusion of gradient information. For BCC volumes, the intensity values of the first sublattice are stored in the R channel, and those of the second sublattice are stored in the G channel.
(a) CC lattice, $80 \times 80 \times 80$ spels, nearest neighbor interpolation.

(b) CC lattice, $80 \times 80 \times 80$ spels, trilinear interpolation.

(c) BCC lattice, $2 \times 63 \times 63 \times 63$ spels, nearest neighbor interpolation.

(d) BCC lattice, $2 \times 63 \times 63 \times 63$ spels, cosine-weighted B-splines with $\lambda = 0.1$.

(e) FCC lattice, $4 \times 50 \times 50 \times 50$ spels, nearest neighbor interpolation.

(f) FCC lattice, $4 \times 50 \times 50 \times 50$ spels, DC-spline interpolation.

Fig. 1. Rendering examples, produced by BccFccRaycaster.

(a) Frame rates, nearest neighbor. The difference between the two storage models is explained in Section 2.2.2.

(b) Frame rates, trilinear interpolation. The difference between the two storage models is explained in Section 2.2.2.

Fig. 2. Frame rates achieved by BccFccRaycaster, using different kinds of interpolation. These frame rates are obtained using an Intel® Core™ i5-3570K CPU @ 3.40 GHz and GeForce GTX 650 Ti Boost (GK106) 2048 MB.
2.2.3. Examples of rendering setups

BccFccRaycaster makes use of the GUI provided by Voreen for rendering setups. Fig. 4 shows a basic setup for a CC volume, using modules, so-called processors, native to Voreen. The VolumeSource processor contains the volume data, the CubeMeshProxyGeometry and MeshEntryExitPoints processors performs pre-processing, and the SingleVolumeRaycaster processor carries out the raycasting process. The Background processor defines the background of the rendered image, which is plotted by the Canvas processor, named CC in this particular example. Examples of rendering setups for BCC and FCC data are shown in Figs. 5 and 6, respectively. Each VolumeSource processor in the BCC and FCC setups represents a sublattice in the data, as defined in the references describing the interpolation methods.
3. Impact

As explained in Section 1.1, the hexagonal, BCC and FCC lattices yield less redundancy, when used as sampling lattices, than the two- and three-dimensional Cartesian lattices. This can be exploited in two obvious ways:

- Keeping the number of sample points constant, better resolution can be reached using the same amount of resources. This approach is explored in [39], where it is shown that phantoms of small tumors or lesions are indeed more visible in an image sampled on a BCC lattice, than on a CC lattice with the same number of points.

- A given resolution may be achieved using less resources, as in [40,6,41]. This could mean, for example, lower radiation dosage in computer tomography, and shorter acquisition time in magnetic resonance imaging. Moreover, fewer data points correspond to shorter processing times and less memory usage for volume images.

Despite these advantages, there is very little active research in image processing and analysis on optimal sampling lattices. LatticeLibrary has already been used to produce the results presented in [31,32,28]. We hope that by making this software framework public, we will lower the threshold for other researchers to explore the advantages of alternative sampling lattices and their applications. Improving and optimizing existing data processing methods may save time, financial and natural resources, and, ultimately, even lives.

4. Future work

While our aim at present is to provide readable, maintainable, and well documented software for handling data sampled on different lattices, we are also interested in improving the computational efficiency of the software, to further encourage its being used. The filtering and morphological operations in LatticeLibrary are easily parallelized, and may even be moved to the GPU to optimize performance. Apart from that, more image processing and analysis methods can be added, image registration being an obvious application of the already implemented distance transforms. We also look forward to seeing LatticeLibrary being used in fields other than image processing.

BccFccRaycaster should be ported to Voreen 4.0 [42], as all work on Voreen 3.0 seem to have ceased. We also wish to add higher order interpolation methods, such as [43], and [44], to facilitate separation of pre-aliasing errors in the data and post-aliasing caused by the rendering process. Naturally, as new interpolation methods are being developed for BCC and FCC lattices, these should be added to the software.

Acknowledgments

The authors would like to thank Fredrik Nysjö, for reviewing part of the BccFccRaycaster code, and Luis Ibáñes, for providing code samples for volume rendering on optimal sampling lattices. We would also like to thank the reviewers for their efforts and insightful comments.
Appendix. Code snippets

Listing 1: Minimal example of image initialization on the supported sampling lattices.

```cpp
#include "cclattice.h"
#include "bcclattice.h"
#include "fcclattice.h"
#include "image.h"

using namespace LatticeLib;

int main(int argc, char *argv[]) {
    int exitValue = 0;
    double *imageDataCC, *imageDataBCC, *imageDataFCC;
    int nRowsCC, nColumnsCC, nLayersCC,
        nRowsBCC, nColumnsBCC, nLayersBCC,
        nRowsFCC, nColumnsFCC, nLayersFCC, nBands;
    double samplingDensity;

    /* Load image data with method of choice,
    and set the variables declared above. */

    CCLattice latticeCC(nRowsCC, nColumnsCC, nLayersCC, samplingDensity);
    BCCLattice latticeBCC(nRowsBCC, nColumnsBCC, nLayersBCC, samplingDensity);
    FCCLattice latticeFCC(nRowsFCC, nColumnsFCC, nLayersFCC, samplingDensity);

    Image<double> volumeImageCC(imageDataCC, latticeCC, nBands);
    Image<double> volumeImageBCC(imageDataBCC, latticeBCC, nBands);
    Image<double> volumeImageFCC(imageDataFCC, latticeFCC, nBands);

    return exitValue;
}
```

Listing 2: Example of image segmentation applied on the BCC sampling lattice.

```cpp
#include "image.h"
#include "bcclattice.h"
#include "intensityworkset.h"
#include "approximateminimumbarrierdistance.h"
#include "pnorm.h"
#include "seededdistancetransform.h"
#include "seed.h"
#include "segmentation.h"
#include <vector>

using namespace LatticeLib;

int main(int argc, char *argv[]) {
    int exitValue = 0;
```
// Initialize the input image.
int nRows, nColumns, nLayers, nBands;
double samplingDensity;
double *volumeData;
/* Load image data with method of choice,
and set the variables declared above. */
BCClattice lattice(
    nRows, nColumns, nLayers, inputDensity);
Image<double> inputImage(
    volumeData, lattice, nBands);
int neighborhoodSize = 14;

// Set the seed points for the distance transform.
int nSegmentationClasses;
vector<vector<Seed> > seedPoints;
/* For each segmentation class, seedPoints
contains a vector of Seed objects, providing
seed points for that class.
Initialize by method of choice. */

// Distance transform.
PNorm<double> norm(2);
ApproximateMinimumBarrierDistance<double>
distanceMeasure(norm);
SeededDistanceTransform distanceTransform;
double *distanceTransformData = new double[
    nRows * nColumns * nLayers * nLabels];
Image<double> distanceImage(
    distanceTransformData, lattice, nLabels);
int *rootData = new int[
    nRows * nColumns * nLayers * nLabels];
Image<int> rootImage(
    rootData, lattice, nLabels);
seededDistanceTransform.apply(
    inputImage, seedPoints, distanceMeasure,
    neighborhoodSize, distanceImage, rootImage);

// Segmentation
Segmentation segmentation;
double *fuzzySegmentationData = new double[
    nRows * nColumns * nLayers * nLabels];
Image<double> fuzzySegmentationImage(
    fuzzySegmentationData, lattice, nLabels);
IntensityWorkset<double> fuzzySegmentation(
    fuzzySegmentationImage, 0, 1);
segmentation.fuzzy(
    distanceImage, neighborhoodSize,
    fuzzySegmentation);

delete volumeData;
delete distanceTransformData;
delete rootData;
delete fuzzySegmentationData;
return exitValue;