British Journal of Mathematics & Computer Science
11(1): 1-15, 2015, Article no.BJMCS.19044
ISSN: 2231-0851

SCIENCEDOMAIN *international*
www.sciencedomain.org

Real-Time Volume Rendering of FCC Datasets Using Box-Splines on the Mobile Platforms

Aaliya Sarfaraz[1] and Muhammad Shaban[2*]

[1]Department of Computer Science, University of Seoul, South Korea.
[2]Department of Electrical and Electronic Engineering, Institut Teknologi Brunei, Brunei.

Article Information

DOI: 10.9734/BJMCS/2015/19044
Editor(s):
(1) Kai-Long Hsiao, Taiwan Shoufu University, Taiwan.
Reviewers:
(1) Anonymous, Suzhou Institute of Biomedical Engineering and Technology, China.
(2) Tayfun Kucukyilmaz, University of Turkish, Turkey.
(3) Anonymous, University of Zilina, Slovakia.
(4) Samir Kumar Bandyopadhyay, University of Calcutta, India.
Complete Peer review History: http://sciencedomain.org/review-history/10602

Original Research Article

Received: 23 May 2015
Accepted: 25 July 2015
Published: 19 August 2015

Abstract

Visualizing volumetric datasets using real-time volume rendering technique involves a large number of interpolation operations that are computationally expensive. This situation used to restrict real-time volume rendering methods to be used only on high-end graphics workstations or special-purpose hardware. This paper presented a real-time direct volume rendering (DVR) implementation of face centered cubic (FCC) datasets with box-spline interpolation on mobile devices. The latest version of OpenGL ES (Open Graphics Library for Embedded System) (3.0) is used for implementation to leverage cutting-edge 3D graphics technology, and it shows interactive performance (2.40 frame per second (FPS)) for moderate-sized volume datasets (64×64×64).

Aims: To present a real-time direct volume rendering (DVR) implementation of face centered cubic (FCC) datasets with box-spline interpolation on mobile devices.

Study Design: Study is based on research conducted in computer lab, University of Seoul, South Korea.

Place and Duration of Study: Computer Science Lab, Department of Computer Science, The graduate College, University of Seoul, South Korea, between June 2014 and April 2015.

Methodology: The latest version of OpenGL ES (Open Graphics Library for Embedded System) (3.0) is used for implementation to leverage cutting-edge 3D graphics technology, and it shows interactive performance (2.40 frame per second (FPS)) for moderate-sized volume datasets (64×64×64).

Results: In our implementation, we calculated the opacity using front–to-back composting whereby the viewing rays are traversed from the eye point into the volume. We also compared different volume sizes, having the same density.

Conclusion: We have presented a real-time volume rendering technique for FCC datasets on mobile

**Corresponding author: Email: m_shaban_khan@yahoo.com;*

devices that efficiently evaluate spline value. Our work has proven that mobile devices constitute a valid program to achieve interactive volume visualization, despite the fact that the rendering capabilities are concentrated in comparison to desktop solutions, due to their inherent autonomy limitations.

Keywords: Volume rendering; ray-casting; FCC lattice; GPU; box-splines.

1 Introduction

Volume rendering is a method of visualizing a three dimensional volumetric data as a two dimensional image with a given view point. Such visualization is significant to gain accurate insight into the tremendous quantity of data. But it is impossible to provide this type of data with conventional rendering techniques due to which, the volume rendering has made its own line of business.

Volume rendering can be employed by any industry or field of research involved with 3D datasets. Till now, the largest area of volume rendering research and its usage is performed by the medical industry. Medical imaging was one of the first applications of volume rendering, and has gone on to be the driving force behind most of the volume rendering research over the past two decades. The two most normally used medical datasets for volume rendering are CT (computed tomography) scans and MRI (magnetic resonance imaging) images. Today's CT scanners and MRI machines typically generate scans of 512 x 512 or 1024 x 1024 pixels. The slices can then be merged into a single 3D representation and used in volume rendering as shown in Fig. 1.

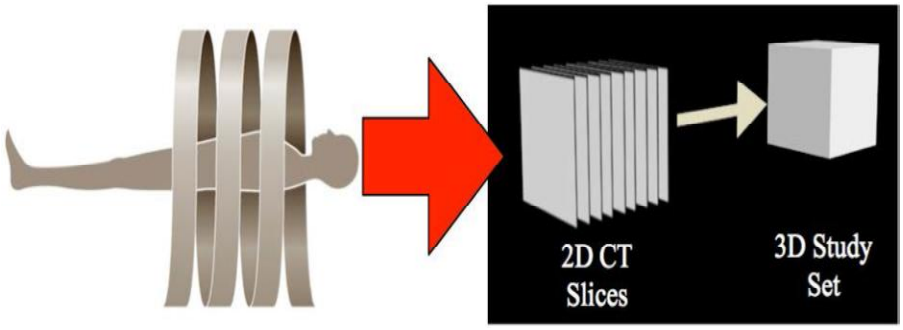


Fig. 1. Illustration of the CT process [25]

It was in the late 2000s that interactive volume visualization became possible in high range desktop and laptops by exploiting the texture functionality [1]. Nowadays, laptops are being replaced with smaller embedded devices like tablets and smartphones. Considering the current trend towards mobile information systems and ubiquitous graphical devices, native volume rendering on mobile devices has become an application with various potential uses; especially last few years have witnessed dramatic improvements that how much computation and communication power can be jammed into such a little gimmick.

However, despite big improvements, the mobile terminals are still clearly less capable than desktop computers in many ways. They operate at a lower speed; the display is smaller in size with lower resolution besides there is less memory for running the program and for storing them, and you can use the device for a short time because the battery will finally work away. The interactive 3D rendering on these devices is quite a challenging job, however gradually the public presentation of several mobile devices has been improving day by day. Recently, some novel devices are equipped with 3D dedicated graphics chips and a high resolution color display as well. In addition, many 3D graphics APIs have been developed such as OpenGL ES, and they are backed by almost every latest mobile device. Due to these advancements, the mobile devices become a possible platform for volume visualization and this made it a hot topic for the researches in this fields.

This paper is going to look how to achieve real-time volume visualization on mobile devices that possess fixed imaginations and not viewed as the appropriate platform for visualization. The present techniques of mobile volume visualization leverages tensor-based volume reconstruction methods such as trilinear or tri-cubic interpolations, while for our algorithm, we adopted the 6-direction cubic box-spline with the FCC datasets due to its high quality and reasonable performance. In this paper we presented a real-time volume rendering technique for FCC datasets on mobile devices that efficiently evaluate spline value. This technique has been implemented and systematically compared with other technique in order to evaluate its performance and visual quality.

The work is sequenced as follows. Section 2 describes the previous work, section 3 describes briefly background and section 4 will discuss implementation details, it goes on to trace the results we obtained in section 5. Section 6 concludes the paper with some proposition for future employment.

2 Literature Review

GPU-based direct volume rendering has been used for visualization of medical and scientific datasets. Numerous approaches have been proposed in the literature [2], [3]. Kruger et al. [4] presented one of the first GPU (Graphics Processing Unit) implementations. Their approach used proxy geometry, most often resembling the data set bounding box to specify ray parameters either through an analytical approach or by rendering the proxy geometry into a texture. GPU ray casting has been actively investigated but mostly limited to datasets on the Cartesian lattice. Kim [5] implemented a real time GPU iso-surface ray caster for FCC datasets. By GPU preprocessing, they achieved superior performance, efficient evolution of spline value and its gradient and efficient empty space skipping.

In respect of mobile devices, interactive direct volume rendering is still a largely unexplored field. On mobile devices, 3D textures are supported through OpenGL ES 3.0 feature. Lamberti et al. [6] and Jeong et al. [7] tried to attempt a rendering server that carries out rendering of the volume and streams the resulting image to the mobile client over a network but unfortunately these server-based solutions require a persistent and fast network connection. Moser and Weiskopf [8] introduced a technique for volume rendering on mobile devices that adopts the 2D texture slicing scheme with a rendering speed of 1.5 frames per second. ImageVis3D [9] is an IOS application for the interactive visualization of very large volumetric datasets, that also uses a 2D textures slicing scheme as well as the GPU ray casting scheme . Then Congote et al. [10] implemented a ray-based technique using WebGL, obtaining a frame rate of around 2-3 FPS.

Mensmann et al. [11] demonstrated that CUDA™ programming model is suitable for volume ray casting and more efficient than a shader-based implementation. With the introduction of loops in shaders in the Shader Model™ 3, a single-pass approach was pioneered by Stegmaier et al. [12]. Similar to other fragment shader based approaches, first a full screen quad is rendered on screen in order to invoke the fragment shader. Then, the ray casting fragment shader is applied. Using the assigned texture coordinates, the ray directions for sampling of volume are determined. Finally, the volume is traversed front-to-back. Such a single-pass approach shows great potential although more improvement is needed, especially for mobile devices where every additional texture lookup degrades performance considerably.

Petkov et al. [13] showed the superiority of the lattice-Bolzmann method on the FCC lattice compared to the Cartesian lattice. Leveraging the isotropic structure of the FCC lattice, Qiu et al. [14] proposed an efficient global illumination method on the FCC lattice by discretizing photon tracing. The six-direction box-spline filter on the FCC lattice was first proposed by Entezari [15] and later investigated in detail by Kim et al. [16].

In particular, there is limited work done on real-time volume visualization on mobile devices. One major issue is the technical limitation of mobile devices, which poses challenges for volume rendering visualization methods.

3 Background

In this section we first describe Direct Volume Rendering (DVR) technique, then FCC lattice, and box spline. For a deep insight of volume rendering, refer to the book by Klaus et al [17] and for the perfect theory of box-spline, refer to the book by de Boor et al. [18]. And lastly we will review OpenGL ES 3.0.

3.1 Direct Volume Rendering

Volume rendering is frequently used method for the 3D visualization of medical images; based on transparency and coloration of voxel. A medical image is composed of a set of voxels, each voxel having a grey level that represents a physical property of the tissue. The approaches in DVR are based on the law of physics like emission, absorption and scattering. In that respect several different optical models used for light interaction with volume densities of absorbing, glowing, reflecting, and scattering material [19].

3.2 The FCC Lattice

A three-dimensional lattice L is determined by all of the integer linear combinations of a non- singular generator matrix G as shown in equation (1):

$$L : \{Gj : j \in Z^3, G \in R^{3X3}, detG \neq 0\} \tag{1}$$

The FCC lattice Z_{FCC} is defined as a subset of the Cartesian lattice Z^3 where the total of its constituents is even shown in equation (2)

$$Z_{Fcc} := \{j \in Z^3, G \in R^{3X3}, detG \neq 0\} \tag{2}$$

or by generator matrix as can be seen in equation (3)

$$G_{fcc} := \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{3}$$

Z_{fcc} Can also be defined as follows in equation (4):

$$Z_{fcc} := \{(x, y, z) \in Z^3 : x + y + z \ \ is\ even\} \tag{4}$$

The FCC lattice shows better sampling efficiency than the Cartesian lattice [20]. While the BCC lattice is the optimal 3D sampling Lattice for band-limited and isotropic signals, the FCC lattice is optimal when the signal is sampled at a depressed rate, as indicated by Künsch et al. [21].

3.3 Box Splines

Box splines were introduced by de Boor at al. [22] as multivariate generalizations of uniform B-splines and have turned out to be remarkably useful.

A box-spline is a piecewise polynomial with a finite support and certain continuity and is uniquely defined by a direction matrix. In equation (5), given an $n \times m$ (usually $n < m$) direction matrix Ξ, a box-spline M_{Ξ} can be constructed by taking consecutive directional convolutions along each column direction (Fig. 2). In other words, starting from the base case (n = m).

$$M_{\Xi}(x) := \frac{1}{|\det \Xi|} X_{\Xi}(X),\ X \in R^n \tag{5}$$

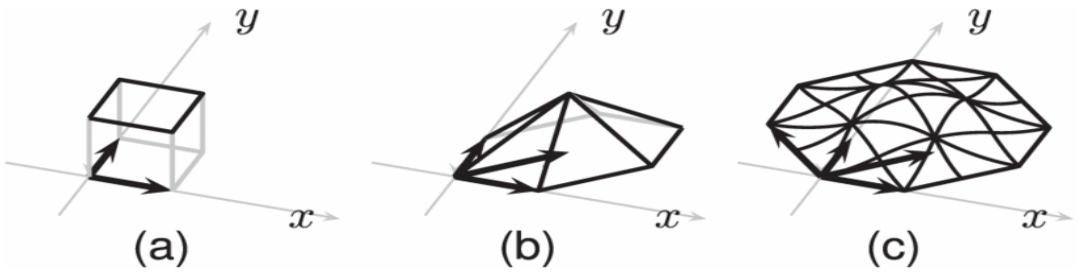


Fig. 2. Construction of box-splines with direction matrices

(a) $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, (b) $\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$ and (c) $\begin{bmatrix} 1 & 0 & 1 & -1 \\ 0 & 1 & 1 & 1 \end{bmatrix}$ via consecutive directional convolutions.

Where Ξ is invertible and $X_{\Xi}(X)$ is the characteristic function on the half-open parallelepiped $\Xi[0,1]^m$. A box-spline defined by the direction matrix $\Xi \cup \{\xi\}$can be recursively defined as in equation (6)

$$M_{\Xi \cup \{\xi\}}(x) := \int_0^1 M_{\Xi}(X - t\xi)dt, \xi \in R^n. \tag{6}$$

Given a discrete dataset on the Cartesian lattice, we can reconstruct a continuous spline s(x) by a convolution of the dataset $V: Z^n \to R$ and a box-spline filterM_{Ξ} shown in equation (7)

$$s(x) := V * M_{\Xi} := \sum_{J \in Z^n} V(j) M_{\Xi}(X - J) \tag{7}$$

While the theory put forth by de Boor et al. [23] is based on shifts on the Cartesian lattice, box-splines on non-Cartesian lattices can be easily obtained by change-of-variables [24].

3.4 OpenGL ES 3.0

OpenGL ES (Embedded system) is an application programming interface (API) for an advanced 3D graphics targeted at handheld and embedded devices. Open GL ES (3.0) was released in 2012 August and is back-ward compatible with OpenGL ES 2.0, while OpenGL ES 2.0 was successful but significant features that enabled techniques such as shadow mapping, volume rendering, GPU based particle animation, geometry instancing, texture compression, and gamma correction were missing. OpenGL ES 3.0 brought these features to mobile devices while taking care of the constraints of embedded system. OpenGL ES 3.0 introduced many new features related to texturing i.e., 3D textures, 2D texture arrays, seamless cube maps and many more. There is likewise a major update in shading language. For brief description, refer to the book by Ginsburg at al. [27].

4 Implementation

This part identifies the components of the real-time volume rendering algorithm we adopted for an efficient evaluation for the spline on the FCC lattice, that's been introduced by Kim at al. [5]. They introduced a ray casting algorithm that is fully implemented as a shader program.

4.1 Implementation Setting

The implementation of the real-time volume rendering is based on android NDK (Native Development Kit) using C language. OpenGL ES 3.0 and EGL 1.4 are used as graphics API's as provided in Intel's Multimedia Accelerator Software Development Kit. Many older devices do not support OpenGL ES 3.0 so for the test program we chose Google Nexus7 tablet with android version 4.4 KitKat.

4.2 Evaluation of Spline on the GPU

For the evaluation of a spline, it is significant to know the polynomial structure induced by the knot planes of M_{cc} since all the stages in the same polynomial piece share the stencil, which is the relative position of finite data on Z_{fcc} required for valuation. Kim et al. [5] already analyzed the spline structure. For the complete theory of GPU evaluation refer to the Kim et al. [5].

There are seven knot planes generated byM_{fcc}, three of which are axis-aligned and decompose the whole space into cubes; $\{\boldsymbol{j}+[0,1)^3 : \boldsymbol{j} \in \mathbb{Z}^3\}$. Depending on the lower corner *j* of each cube, we can split the cubes into two groups:

$$\{\boldsymbol{j}+[0,1)^3 : \boldsymbol{j} \in \mathbb{Z}_{\text{fcc}}\} \quad \text{and} \quad \{\boldsymbol{j}+[0,1)^3 : \boldsymbol{j} \in (\mathbb{Z}^3 \setminus \mathbb{Z}_{\text{fcc}})\}.$$

Notice that each group can be identified by calculating the parity of j; j (1) + j (2) + j (3). Then by the remaining four knot planes, cubes in each group are decomposed into five tetrahedral $\{\tau_0, \tau_1, \tau_2, \tau_3, \tau_4\}$ and $\{\tau_5, \tau_6, \tau_7, \tau_8, \tau_9\}$ in Fig. 5. Notice that each tetrahedron τ_j in the second group (Figs. 3 (f) – (j)) can be transformed to τ_{J-5} with a reflection with regard to the origin followed by a translation by (1, 1, 1):

$$\tau_{j-5} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tau_j + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \quad \text{for } j \in \{5, 6, 7, 8, 9\}.$$

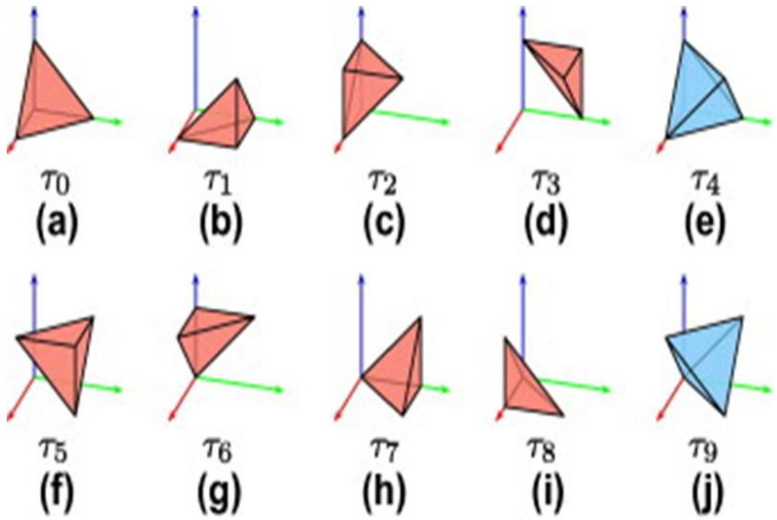


Fig. 3. Shift-invariant tetrahedral polynomial pieces induced by M_{fcc} for (top) even and (bottom) odd parity cubes

4.3 Challenges

During this algorithm, we came across a number of problems because real-time rendering on a mobile device is subject to a number of limitations.

1. In the fragment shader, we came across with a floating point texture problem. We can accept two options either to use texelFetch() or texture() and both are quite different in functionality. texture is the usual texture access function, which handles filtering and normalized ([0,1]) texture coordinates whereas the texelFetch() directly access a texel in the texture without using un-normalized coordinates .In our implementation we were not able to fork out using texelFetch () so we decided to use texture().
2. The limited size of graphics memory is another major problem for volume rendering: 16 MB is typically very small for volume datasets. Simply here we cannot render with a 16MB dataset hence the datasets we used is about 32KB~622KB.
3. With 3D textures we were unable to employ a floating point texture (GL_R32F, GL_R32UI etc.). Floating point textures have a special designated range of internal formats (RGBA_16F, RGBA_32F, etc.). Regular textures store fixed-point data, so reading from them gives you [0, 1] range values. Contrary, floating point textures give you floating point numbers as a result. Also not all hardware supports filtering of floating point textures so is the case here, our hardware doesn't support it as well so we decided to use unsigned-byte (GL_R8) and it worked. Fig. 4 shows the GLSL code piece that fetches FCC data coefficients for evaluation and Fig. 5 shows the GLSL code piece that evaluates the coefficients [5].

```
ivec3      nearest;
      vec3          plocal;
      ivec3         unit;
origin = floor(p_in);
      p_cube = p_in-origin;
      parity = int(dot(origin,vec3(1,1,1)))&1;
      plocal = p_cube;
p_cube += float(parity)*(1.0-2.0*p_cube);
      vitet = vec4(
                              dot(p_cube,vec3(-1,-1,-1))>-1.0,
                              dot(p_cube,vec3( 1, 1,-1))> 1.0,
                              dot(p_cube,vec3( 1,-1, 1))> 1.0,
                              dot(p_cube,vec3(-1, 1, 1))> 1.0
                              );
      type = 1.0-dot(vitet,vec4(1.0,1.0,1.0,1.0));
      itet = int(dot(vitet.yzw,vec3(1,2,3))) + 4*int(type) + parity*5;
      ivec3           offset = int(vitet.y+vitet.z+vitet.w)*(1-ivec3(vitet.wzy));
      offset += parity*(1 - 2*offset);
      unit = 1-2*offset;
      plocal = vec3(offset) + vec3(unit)*plocal;
      nearest = ivec3(origin) + offset;
#define   FETCH_COEFF(i)            c[i] = texture(volume_tex,

      vec3(fcc)*scale_position_inv).r;
      unit *= 2;
      ivec3           fcc = ivec3(nearest);
            FETCH_COEFF(0)
      fcc[0] += unit[0];
            FETCH_COEFF(1)
      fcc[0] -= unit[0];        fcc[1] += unit[1];   FETCH_COEFF(2)
      fcc[1] -= unit[1];        fcc[2] += unit[2];   FETCH_COEFF(3)
      fcc[0] += unit[0];        fcc[1] +=(unit[1]>>1);
fcc[2] =(unit[2]>>1)FETCH_COEFF(4)
```

```
fcc[0] -= unit[0];                                  FETCH_COEFF(5)
      fcc[2] -= unit[2];                            FETCH_COEFF(6)
      fcc[1] -= unit[1];                            FETCH_COEFF(7)
      fcc[2] += unit[2];                            FETCH_COEFF(8)
      fcc[0] += (unit[0]>>1);fcc[1] += ((3*unit[1])>>1);
FETCH_COEFF(9)
      fcc[1] -= unit[1];                            FETCH_COEFF(10)
      fcc[0] -= unit[0];                            FETCH_COEFF(11)
      fcc[2] -= unit[2];                            FETCH_COEFF(12)
      fcc[0] += unit[0];                            FETCH_COEFF(13)
      fcc[1] += (unit[1]>>1);   fcc[2] += ((3*unit[2])>>1);
FETCH_COEFF(14)
      fcc[2] -= unit[2];                            FETCH_COEFF(15)
      fcc[0] -= unit[0];                            FETCH_COEFF(16)
      fcc[1] -= unit[1];                            FETCH_COEFF(17)
      fcc[0] += unit[0];                            FETCH_COEFF(18)

#undef    FETCH_COEFF
```

Fig. 4. GLSL code piece that fetches FCC data coefficients

```
#define   CUBE(x) ((x)*(x)*(x))
#define   SQR(x)  ((x)*(x))
#define   x       plocal[0]
#define   y       plocal[1]
#define   z       plocal[2]


      float u0, u1, u2, u3, val0, val1;

      u0 = (1.0-x-y-z);
      u1 = x;
      u2 = y;
      u3 = z;
      val0 =
      0.0416666667*
      (
        SQR(u0)*(
            u0*(-4.0* c[0] + c[5] + c[10] + c[15] + c[11] + c[16] + c[8] + c[6]
          + c[18] +c[13] + c[7] + c[12] + c[17]) +
            3.0*(
                    -4.0*(c[5]*(u2 + u3) + c[10]*(u1 + u3) + c[15]*(u1 + u2)) +
                     u1*(2.0*(c[10] + c[15] + c[18] + c[13]) + c[5] + c[8] + c[6]
          + c[7]) + u2*(2.0*(c[5] + c[15] + c[16] + c[6]) + c[10] +
            c[11] + c[13] + c[12]) +u3*(2.0*(c[5] + c[10] + c[11] +
c[8]) + c[15] + c[16] + c[18] + c[17]) )
          ) +
          4.0*(
            (
                    c[0] +
                     c[5]*CUBE(u2 + u3) +c[10]*CUBE(u1 + u3) +c[15]*CUBE(u1 + u2)
            +SQR(u1)*(u1*(c[18] + c[13] + c[1]) + 3.0*(c[13]*u2 +
            c[18]*u3)) +SQR(u2)*(u2*(c[16] + c[6] + c[2]) + 3.0*(c[16]*u3
            + c[6]*u1)) +SQR(u3)*(u3*(c[11] + c[8] + c[3]) +
3.0*(c[11]*u2 + c[8]*u1))
                    ) +
            3.0*(
                     c[0]*u1*u2*u3 +(u1*u2 + u2*u3 + u3*u1)*(c[0]*(1.0 + u0) +
            c[5]*(u2 + u3) +c[10]*(u1 + u3) + c[15]*(u1 + u2)) +u0*(
              c[0] + (u1 + u2)*(c[15] + c[6]*u2 + c[13]*u1) +(u2 +
```

```
u3)*(c[5] + c[11]*u3 + c[16]*u2) +(u1 + u3)*(c[10] + c[8]*u3
                + c[18]*u1))
                        )
            )
        );
    u0 =  x+y+z-1.0;
```

```
u1 = -x+y-z+1.0;
    u2 =  x-y-z+1.0;
    u3 = -x-y+z+1.0;
    val1 =
    0.0208333333*
    (

        SQR(u0)*
        (
                3.0*(2.0*(c[5]*(u1 + u3) + c[10]*(u2 + u3) + c[15]*(u1 + u2))
+ c[14]*u3 + c[9]*u1 + c[4]*u2) +u0*(c[5] + c[10] + c[15] +
c[14] + c[9] + c[4]))+
        SQR(u1)*
        (
                3.0*(2.0*(c[0]*(u2 + u3) + c[5]*(u0 + u3) + c[15]*(u0 + u2)) +
c[2]*u0 + c[16]*u3 + c[6]*u2) +
                u1*(c[0] + c[5] + c[15] + c[2] + c[16] + c[6])
        )+
        SQR(u2)*
        (
                3.0*(2.0*(c[0]*(u1 + u3) + c[10]*(u0 + u3) + c[15]*(u0 + u1))
+ c[1]*u0 + c[18]*u3 + c[13]*u1) +
                u2*(c[0] + c[10] + c[15] + c[1] + c[18] + c[13])
        )+
        SQR(u3)*
        (
                3.0*(2.0*(c[0]*(u1 + u2) + c[5]*(u0 + u1) + c[10]*(u0 + u2)) +
c[3]*u0 + c[11]*u1 + c[8]*u2) +
                u3*(c[0] + c[5] + c[10] + c[3] + c[11] + c[8])
        )+
        3.0*
        (
                c[0]*(u0*SQR(2.0- u0) + 6.0*u1*u2*u3) +
                c[5]*(u2*SQR(2.0- u2) + 6.0*u0*u1*u3) +
                c[10]*(u1*SQR(2.0- u1) + 6.0*u0*u2*u3) +
                c[15]*(u3*SQR(2.0- u3) + 6.0*u0*u1*u2)
      )
    );

    return val0*(1.0-type)+val1*type;

#undef    CUBE
#undef    SQR
#undef    x
#undef    y
#undef    z
}
```

Fig. 5. GLSL code for coefficient evaluation

5 Results

The main drawback we have faced is the limited texture memory hence our implementation failed to compute the dataset having more than 86x86x86 size of voxels. So all the datasets had to be resampled in order to achieve more meaningful comparison, to serve this purpose we used MATLAB [26]. The datasets are provided by S. Roettger et al. [27]. All the tests are done on Windows 7 Professional (64 bit) with Intel

R
Xeon

R CPU X5550 @2.67 GHz and 12GB memory.

In our implementation, we calculated the opacity using front–to-back composting whereby the viewing rays are traversed from the eye point into the volume. Table 1 shows the performance comparison for the same size datasets. Fig. 6 shows the rendering results of linear interpolation and our implementation side by side. For reference images we included desktop generated images using box-spline interpolation, shown on the left hand side. The rendering results seen on the right hand side are rendered with linear interpolation and the images in the center are rendered with our box-spline interpolation. It is seen that our results looks more blurry here, it is because we are using a very small dataset with a low resolution, to claim this we carried out our test on desktop with a high resolution dataset. The result we got in desktop is shown in Fig. 8, where we used the same density datasets. Now it is clearer that with a high resolution dataset FCC box-spline interpolation has better performance than linear interpolation. Also the volume rendered images by the Box-spline interpolation seem smaller than the ones by the linear interpolation, the difference is because of the number of values required to evaluate a spline [18].

In the comparison of rendering with the FCC box-spline interpolation, it is conspicuous that our algorithm has good image quality, even though it employs only half of the dataset but still getting along with linear interpolation. Hence spline evaluation is an expensive operation and have limited support, we get a low frame rate. Further, it should be brought into amount that linear interpolation is hardwired on the graphics hardware according to what, it is much more time efficient. Table 1 shows the performance comparison (fps) of both techniques.

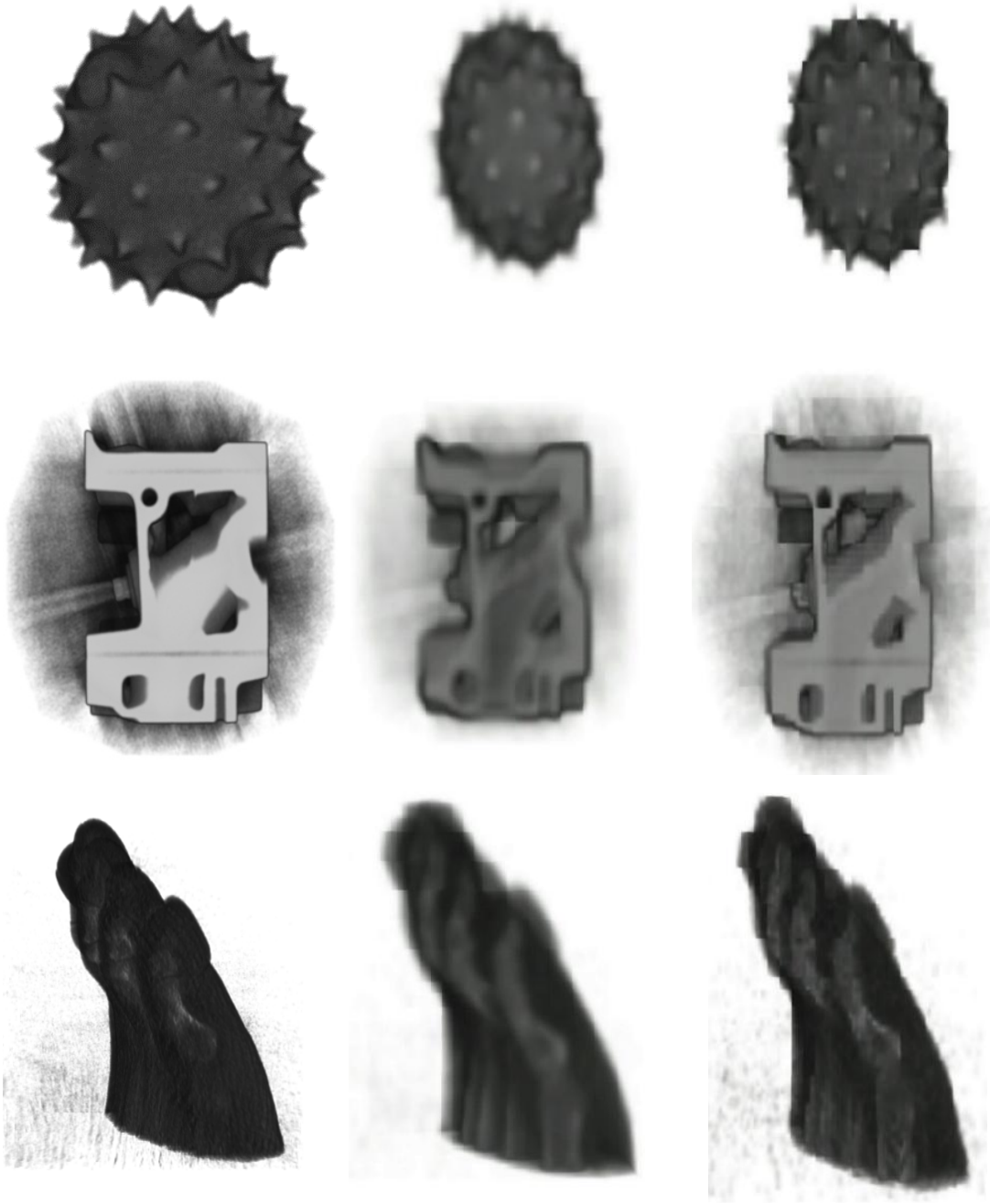
We also compared different volume sizes, having the same density. Table 2 shows the performance comparison. Fig. 7 compared the rendering results, it can be seen easily that linear interpolation has more jiggling whereas our method has more sooth and clear results. Conclusively our algorithm showed better quality due to the diminished number of data fetches, and thus the lower complexity of the spline pattern.

Table 1. Performance (in fps: frames per second) comparison

Dataset	Size	Linear interpolation (fps)	FCC box-spline interpolation (fps)
Daisy	64×60×56	50.40	2.40
Engine	64×64×64	46.79	2.40
Foot	64×64×64	48.40	2.40

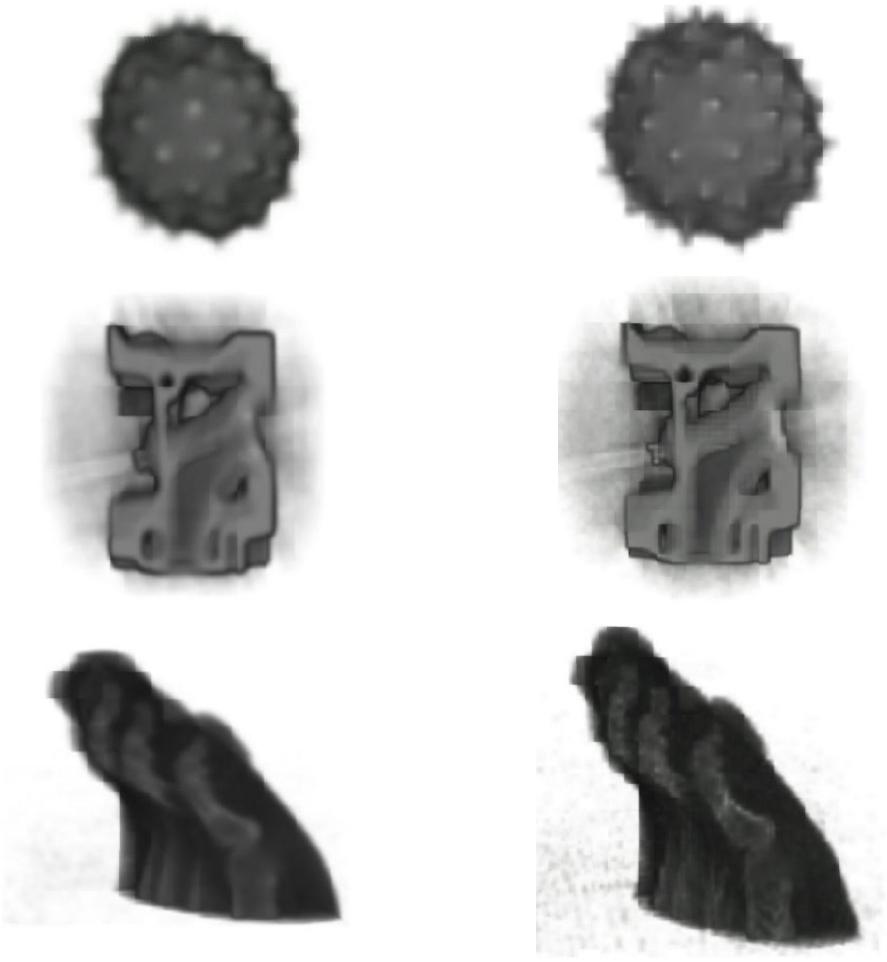
Table 2. Performance comparison (same density of different volume sizes)

Dataset	Size linear interpolation	Speed linear interpolation (fps)	Size FCC box-spline interpolation	Speed FCC box-spline interpolation (fps)
Engine	65×65×65	45.20	82×82×82	2.40
Daisy	49×46×43	64.80	62×58×54	2.80
Foot	65×65×65	47.59	82×82×82	2.40



(a) **(b)** **(c)**

Fig. 6. Rendered images: (a) Desktop generated; (b) FCC datasets with box-spline interpolation and (c) linear interpolation



(a) **(b)**

Fig. 7. Rendered images: performance comparisons for table 2; (a) FCC datasets with box-spline interpolation and (b) linear interpolation

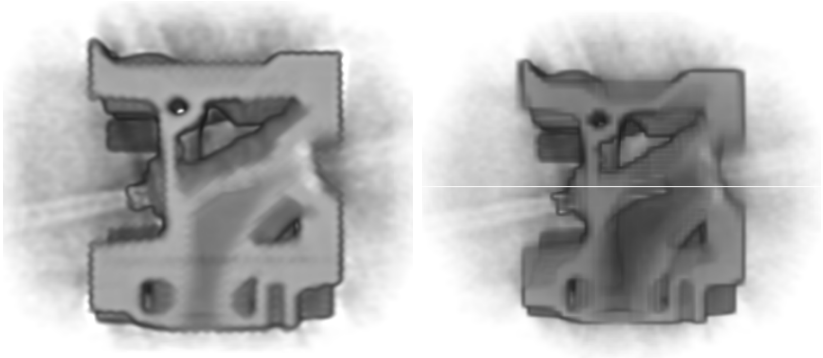


Fig. 8. Desktop generated images: Linear interpolation and Box-spline interpolation

6 Conclusion

We have presented a real-time volume rendering technique for FCC datasets on mobile devices that efficiently evaluate spline value. Our work has proven that mobile devices constitute a valid program to achieve interactive volume visualization, despite the fact that the rendering capabilities are concentrated in comparison to desktop solutions, due to their inherent autonomy limitations. We compared our results with the linear interpolation, our experiments show that the FCC box-spline interpolation provides a slightly higher quality image, even though using only half of the datasets. This work is a clear contribution to the literature. However, there comes a challenge for speed and storage, though it paves a new research topic for future.

As future work, our current research is centered on the betterment of the rendering performance and quality based on a continuous search of new techniques as well suited to this kind of devices.

Acknowledgements

Authors declare that this work was done in Department of Computer science, University of Seoul and Electrical and Electronic Engineering, Institut Teknologi Brunei. No funding was provided by any government or semi government authority as this is purely research based work.

Competing Interests

Authors have declared that no competing interests exist.

References

[1] Engel K, Kraus M, Ertl T. High-quality pre-integrated volume rendering using Hardware-accelerated pixel shading. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics hardware. HWWS '01, ACM. 2001;9(16).

[2] Engel K, Hadwiger M, Kniss JM, Lefohn A, Salama CR, Weiskopf D. Real-time volume graphics. A.K. Peters Publisher; 2005.

[3] Preim B, Bartz D. Visualization in Medicine. Elsevier Inc. Publisher; 2007.

[4] Kr¨uger J, Westermann R. Acceleration techniques for GPU-based volume rendering. Proceedings of IEEE Visualization. 2003;287–292.

[5] Kim M. GPU isosurface raycasting of FCC datasets. Graphical Models. 2013;75(2):90–101.

[6] Lamberti F, Sanna A. A solution for displaying medical data models on mobile devices. In SEPADS'05, pp. 1–7, Stevens Point, Wisconsin, USA. (WSEAS); 2005.

[7] Jeong S, Kaufman AE. Interactive wireless virtual colonoscopy. The Visual Computer. 2007;23(8): 545–557.

[8] Moser M, Weiskopf D. Interactive Volume Rendering on Mobile Devices. In Workshop on Vision, Modelling, and Visualization VMV '08. 2008;217–226.

[9] ImageVis3D. Imagevis3d: A real-time volume rendering tool for large data. Scientific computing and imaging institute (sci); 2011.

[10] Congote J, Segura A, Kabongo L, Moreno A, Posada J, Ruiz O. Interactive visualization of volumetric data with Webgl in real-time. Proceedings of the Web3D '11. 2011;137–146. ACM.

[11] Jörg Mensmann, Timo Ropinski, Klaus H. Hinrichs. An Advanced Volume Raycasting Technique using GPU Stream Processing", GRAPP: International Conference on Computer Graphics Theory and Applications. 2010;190-198.

[12] Stegmaier S, Strengert M, Klein T, Ertl T. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. Proceedings of The Fourth International Workshop on Volume Graphics. 2005;187 – 241.

[13] Kaloian Petkov, Feng Qiu, Zhe Fan, Arie E. Kaufman, Klaus Mueller. Efficient LBM visual simulation on face-centered cubic lattices. IEEE Transactions on Visualization and Computer Graphics 15, ISSN 1077-2626. 2009;802–814.

[14] Feng Qiu, Fang Xu, Zhe Fan, Neophytou Neophytos, Arie Kaufman, Klaus Mueller. Lattice-based volumetric global illumination. IEEE Transactions on Visualization and Computer Graphics. 2007;13(6):1576–1583.

[15] Alireza Entezari, Optimal, "Sampling Lattices and Trivariate Box Splines", PhD Thesis, Simon Fraser University; 2007.

[16] Minho Kim, Alireza Entezari, Jörg Peters. Box spline reconstruction on the face-centered cubic lattice. IEEE Transactions on Visualization and Computer Graphics. 2008;14(6):1523–1530.

[17] Klaus Engel, Markus Hadwiger, Joe M.Kniss, Christ of Rezk-Salama, Daniel Weiskopf, Real-Time Volume Graphics.

[18] Carl de Boor, Klaus Höllig, Sherman Riemenschneider, Box Splines, Springer Verlag New York, Inc.; 1993.

[19] Nelson Max. Optical Models for Direct Volume Rendering. IEEE transection on Visualization and Computer Graphics. 1995;1077-2626.

[20] John Horton Conway, Neil J.A. Sloane, Sphere Packings, Lattices and Groups, third ed., Springer-Verlag New York, Inc., New York, NY, USA; 1998.

[21] Hans R. Künsch, Erik Agrell, Fred A. Hamprecht, "Optimal lattices for sampling", IEEE Transactions on Information Theory. 2005;51(2):634–647.

[22] de Boor C, Hollig KA, Riemenschneider S. Box Splines. Applied Mathematical Sciences, Springer-Verlag. 1998;159–174.

[23] Carl de Boor, Klaus Höllig, Sherman Riemenschneider, Box Splines, Springer Verlag New York, Inc.; 1993.

[24] Minho Kim, Jörg Peters. Symmetric box-splines on root lattices. Journal of Computational and Applied Mathematics. 2011;235(14):3972–3989.

[25] Christian John Noon. A Volume Rendering Engine for Desktops, Laptops, Mobile Devices and Immersive Virtual Reality Systems using GPU-Based Volume Recasting. PhD thesis, Iowa State University; 2012.

[26] MATLAB, version 8.10.604 (R2013a). The Math Works Inc. Natick, Massachusetts; 2013.

[27] Roettger S. Volume library (online). January 2012.
Available: http://www9.informatik.uni-erlangen.de/External/vollib/

Peer-review history:
The peer review history for this paper can be accessed here (Please copy paste the total link in your browser address bar)
http://sciencedomain.org/review-history/10602