Fast Simulation of X-ray Projections of Spline-based Surfaces using an Append Buffer

Andreas Maier
Department of Radiology, Stanford University, Stanford, CA, USA

Hannes G. Hofmann
Pattern Recognition Lab, Department of Computer Science, Friedrich-Alexander University of Erlangen-Nuremberg, Germany

Chris Schwemmer and Joachim Hornegger
Erlangen Graduate School in Advanced Optical Technologies (SAOT), Universität Erlangen-Nürnberg
Pattern Recognition Lab, Department of Computer Science, Friedrich-Alexander University of Erlangen-Nuremberg, Germany

Andreas Keil and Rebecca Fahrig
Department of Radiology, Stanford University, Stanford, CA, USA

Many scientists in the field of x-ray imaging rely on the simulation of x-ray images. As the phantom models become more and more realistic, their projection requires high computational effort. Since x-ray images are based on transmission, many standard graphics acceleration algorithms cannot be applied to this task. However, if adapted properly, simulation speed can be increased dramatically using state-of-the-art graphics hardware.

A custom graphics pipeline that simulates transmission projections for tomographic reconstruction was implemented based on moving spline surface models. All steps from tessellation of the splines, projection onto the detector, and drawing are implemented in OpenCL. We introduced a special append buffer for increased performance in order to store the intersections with the scene for every ray. Intersections are then sorted and resolved to materials. Lastly, an absorption model is evaluated to yield an absorption value for each projection pixel.

Projection of a moving spline structure is fast and accurate. Projections of size 640x480 can be generated within 254 ms. Reconstructions using the projections show errors below 1 HU with a sharp reconstruction kernel. Traditional GPU-based acceleration schemes are not suitable for our reconstruction task. Even in the absence of noise, they result in errors up to 9 HU on average, although projection images appear to be correct under visual examination.

Projections generated with our new method are suitable for the validation of novel CT reconstruction algorithms. For complex simulations, such as the evaluation of motion-compensated reconstruction algorithms, this kind of x-ray simulation will reduce the computation time dramatically. Source code is available at http://conrad.stanford.edu.

Key words: x-ray imaging, simulation, hardware acceleration, GPU, digitally reconstructed radiographs
PACS: Image reconstruction in medical imaging, 87.57.nf
Computer modelling and simulation, 07.05.Tp

I. INTRODUCTION

Significant efforts to reduce x-ray dose to the patient have been made over the last few years, and today's scanners expose patients to only a limited amount of radiation [1][2]. However, x-ray imaging is still based on ionizing radiation, and therefore efforts to reduce patient dose continue [3] [4] [5][6] [7]. Following the ALARA principle, dose to the patient should be as low as reasonably achievable. To facilitate the development of new algorithms while avoiding additional dose to the patient, simulation of the imaging process is required [8][9]. Numerical phantoms are frequently used in this process [10]. However, these numerical phantoms more often than not deviate significantly from realistic anatomy and hence results from phantom data are difficult to generalize to *in vivo* data.

Modern phantoms depict anatomy more realistically [11][12] and model different artifact sources such as breathing motion [13] and heart motion [14]. The simulation of x-ray physics in a Monte Carlo simulation has also been combined with complex numerical phantoms [15].

A major drawback of all these methods is that such detailed simulation is numerically expensive and the simulation of a single scan takes from several hours up to several days. Monte Carlo studies are often only performed if effective doses are to be computed or if correction methods for specific physical effects such as scatter are to be developed. In particular, for motion and its compensation, most studies simulate only mono-energetic x-ray behaviour [16][17][18] as the link between motion and physical effects at the photon level is rather weak.

In recent years, there have been great advances in computer graphics, many of which have found their way into the field of medical imaging. One of the most predominant methods is the use of so-called general-purpose graphics processing units (GPGPUs), which are programmed using e.g. Nvidia's Compute Unified Device Architecture (CUDA) [19][6][20] or the Open Computing Language (OpenCL) [21]. Furthermore, there have also been algorithmic advances that are suitable for x-ray imaging [22][23][24][25]. Especially in the field of digitally rendered radiographs (DRRs), these methods are commonly used to create 2-D projections of voxelized data [26].

In contrast to traditional DRRs, we use an analytic model of the scene for DRR generation. This model describes the scene in 3-D and is additionally parameterized with time. Hence, the scene is defined in four dimensions. While this approach seems similar to methods presented by Villard et al. [27], our approach differs from theirs and provides the required high accuracy for subsequent reconstruction. Villard et al. [27] describe a system that generates x-ray projection images of a breathing chest phantom in real-time using OpenGL with the goal of training physicians in an angiographic, i.e. 2-D, environment. Since the data is displayed on a normal computer monitor, a dynamic range of 8 bits is sufficient. Their images are rendered and displayed directly on the computer screen and therefore small errors in the depth values are not visible with this rough quantization. In addition, the subsequent use of the generated images does not require fetching the data from the graphics card back to the host computer's memory. In the context of their application, it would not make sense to use a more accurate (and computationally expensive) quantization. In CT reconstruction, the subtle differences in the absorption values that are lost by a coarse quantization are crucial to obtaining an accurate reconstruction. Furthermore, the approach presented in [27] does not generate correct values for all projection pixels. In some cases, an incorrect value is obtained. These outliers are detected by a subsequent filter and defect pixel interpolation is applied. Again, this is sufficiently accurate for visual interaction, but is another unacceptable source of error for a subsequent tomographic reconstruction. Overall, their approach is very different from ours. In the approach we present here, everything in the graphics pipeline was adjusted including the rasterization, which cannot be altered in OpenGL.

In the following, we will describe a novel simulation pipeline that allows fast generation of x-ray transmission images from an analytic time-variant scene description. Our model is able to generate geometrically accurate x-ray absorption images with respect to motion and physical effects that can be described on a ray-intersection level. Simulation of particle effects such as scattered photons is not yet part of our image formation process.

II. METHODS

The complete processing pipeline is designed to run entirely on graphics card hardware. In this paper, we will mainly focus on the use of spline-based geometrical descriptions, although any geometric shape is suitable for this method, since any shape can be represented by a set of sufficiently small triangles. Figure 1 displays the processing pipeline in more detail. First, a set of 3-D points that sample the surface of the imaged object is generated. Since spline surfaces are an analytic description of the object's shape, any number of sample points can be generated. Connection of neighbouring points with triangles delivers a piecewise-linear approximation of the analytic shape. This process is also known as "tessellation". Next, the points are projected onto the detector plane. As this is an affine transform, triangles are projected onto triangles in detector coordinates. In most applications in computer graphics, these triangles would directly be drawn onto the screen buffer in the graphics card and displayed. In our case, however, we need to perform additional steps to form the final image. For each pixel, we store the intersections of its ray with each triangle in a so-called "append buffer" structure. Then these intersections are sorted according to their depth. For each segment along the ray, the traversed object and its material is identified. Based on these intersections an absorption model is then evaluated to compute the final absorption value per pixel. In the following sections, we will explain each of these steps in more detail and describe necessary adaptations of the algorithms that are commonly used in computer graphics in a similar way.

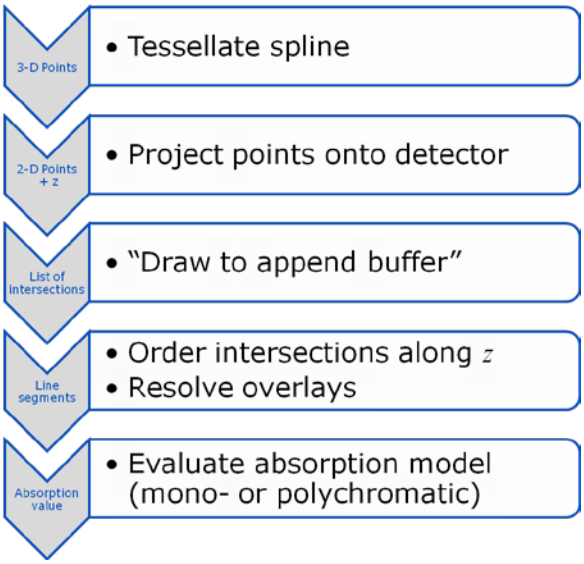

Figure 1: Processing order of the described graphics pipeline: First, the scene is sampled into sets of connected points. These 3-D points are then projected onto detector coordinates and subsequently drawn into an "append buffer", i.e. a screen buffer that is able to store a list of intersections per pixel. For each pixel the line segments that traverse different materials have to be determined. They are used in the absorption model to compute the pixel's observed accumulated mass attenuation.

A. FAST TIME-VARIANT SPLINE TESSELLATION

For tessellation, spline shapes have to be sampled often and therefore should also be sampled quickly. Spline-based surfaces are commonly used in geometric modelling, with the most common being cubic spline surfaces. Although we only describe B-splines in this paper, all the methods can also be applied to other spline-based models such as NURBS [28].

In general, a spline shape always has an internal and an external dimension. The internal dimension describes the dimension of the manifold that the spline represents. A spline of internal dimension one is a curve, while a spline of internal dimension two describes a surface. Splines with internal dimension three will describe a surface that is moving over time. The external dimension is the dimension in which the spline's control points are defined, that is, splines with external dimension two are defined in a 2-D space and splines with external dimension three are defined in a 3-D space. In the experimental part of this paper we deal with moving surface models in 3-D. Therefore, the spline models used have internal and

external dimension three. In the following, we will follow the notation of Ruijters et al. [29] closely.

A spline $\boldsymbol{s}(u)$ with internal dimension one is defined as a weighted sum of its control points $\boldsymbol{c}(l)$

$$\boldsymbol{s}(u) = \sum_l \boldsymbol{c}(l)\beta(u-l), \tag{1}$$

where u indexes the internal dimension and l iterates over all control points. The term $\beta(u-l)$ describes the weighting function for each control point. A possible definition of $\beta(u)$ for a cubic spline is

$$\beta(u) = \begin{cases} 0, & |u| \geq 2 \\ \frac{1}{6}(2-|u|)^3, & 1 \leq |u| < 2 \\ \frac{2}{3} - \frac{1}{2}|u|^2(2-|u|), & |u| < 1 \end{cases}. \tag{2}$$

Note that with this definition of $\beta(u)$, only the four control points with non-zero weights have to be evaluated in Eq. 1. With this definition, a spline with internal dimension three is evaluated as

$$\boldsymbol{s}(u,v,t) = \sum_i \sum_k \sum_l \boldsymbol{c}(l,k,i)\beta(u-l)\beta(v-k)\beta(t-i). \tag{3}$$

Each sum evaluates four control points. Hence, a 4x4x4 loop has to be performed for each spline evaluation and global memory has to be accessed 64 times.

In [22] it is shown that this operation can be executed more quickly on graphics hardware. By means of linear interpolation, the computation can be mapped onto the texture units of the graphics card:

$$f(l+\lambda) = (1-\lambda)f(l) + \lambda f(l+1), \tag{4}$$

where l is the integer part of the interpolation position and λ its decimal part. Sigg et al. [22] exploit the fact that any weighted addition of neighbouring elements can be described by a weighted linear interpolation. This is given by

$$af(l) + bf(l+1) = (a+b)f\left(l + \frac{b}{a+b}\right). \tag{5}$$

Using the fact that u can be split into $l+\lambda$ and that there are only four relevant points, Eq. 1 can be rewritten as

$$\boldsymbol{s}(u) = \boldsymbol{s}(l+\lambda) = \boldsymbol{c}(l-1)\beta(\lambda+1) + \boldsymbol{c}(l)\beta(\lambda) + \boldsymbol{c}(l+1)\beta(\lambda-1) + \boldsymbol{c}(l+2)\beta(\lambda-2).$$

Now Eq. 5 can be applied which yields

$$\boldsymbol{s}(u) = \boldsymbol{s}(l+\lambda) = g_{l0}\boldsymbol{c}(l+h_{l0}) + g_{l1}\boldsymbol{c}(l+h_{l1}) \tag{6}$$

with

$$g_{l0} = \beta(\lambda + 1) + \beta(\lambda), \qquad g_{l1} = \beta(\lambda - 1) + \beta(\lambda - 2) \tag{7}$$

$$h_{l0} = \frac{\beta(\lambda)}{g_{l0}} - 1, \qquad h_{l1} = \frac{\beta(\lambda - 2)}{g_{l1}} + 1$$

Using the same procedure, this process can also be extended to splines with higher internal dimensions. With internal dimension three, weights for the decomposition of the internal dimensions $v = k + \kappa$ and $t = i + \iota$ can be defined in a manner analogous to that of Eq. 7. Using this process, Eq. 3 is reformulated to

$$\begin{aligned}\boldsymbol{s}(u,v,t) = g_{i0}\{g_{l0}[g_{k0}\boldsymbol{c}(l + h_{l0}, k + h_{k0}, i + h_{i0}) + g_{k1}\boldsymbol{c}(l + h_{l0}, k + h_{k1}, i + h_{i0})] \\ + g_{l1}\,[g_{k0}\boldsymbol{c}(l + h_{l1}, k + h_{k0}, i + h_{i0}) + g_{k1}\boldsymbol{c}(l + h_{l1}, k + h_{k1}, i + h_{i0})]\} \\ + g_{i1}\{g_{l0}[g_{k0}\boldsymbol{c}(l + h_{l0}, k + h_{k0}, i + h_{i1}) + g_{k1}\boldsymbol{c}(l + h_{l0}, k + h_{k1}, i + h_{i1})] \\ + g_{l1}\,[g_{k0}\boldsymbol{c}(l + h_{l1}, k + h_{k0}, i + h_{i1}) + g_{k1}\boldsymbol{c}(l + h_{l1}, k + h_{k1}, i + h_{i1})]\}.\end{aligned}$$

While this computation is analytically equivalent to the formulation in Eq. 3, it requires only 8 3-D texture interpolations instead of 64 accesses to the global memory. When memory bandwidth is an issue, computations can be performed without time penalty. Attention has to be paid to the accuracy of the computation when the interpolation units are used. Therefore, numerical stability will be evaluated in the results section.

Sigg and Hadwiger [22] further suggest replacing the computation of the weights in Eq. 7 by a texture look-up. This method again requires access to the global memory. Depending on the generation of the graphics card and its ratio of computational power to global memory access rates, this can even lead to decreased performance, since the computational power of graphics cards is increasing more rapidly than their global memory access rates [30]. Since the local computation also allows higher numerical accuracy, we chose to compute the interpolation weights locally in our implementation, as proposed in reference [29].

In order to get a mesh of triangles, $\boldsymbol{s}(u,v,t)$ is now sampled for every point in time t, on a $U \times V$ grid, where U is the number of points in the u direction and V is the number of points in the v direction. Every cell in this grid generates two triangles, T_1 and T_2. T_1 is described by the points $\boldsymbol{s}(u,v,t)$, $\boldsymbol{s}(u+1,v,t)$,

and $\boldsymbol{s}(u+1, v+1, t)$ and T_2 by the points $\boldsymbol{s}(u, v, t)$, $\boldsymbol{s}(u, v+1, t)$, and $\boldsymbol{s}(u+1, v+1, t)$.

B. PROJECTION

In our framework, the projection of 3-D points onto 2-D points is performed using a perspective projection. We formulate this using a 3x4 projection matrix $\boldsymbol{P}$ in homogeneous coordinates [31]

$$\boldsymbol{s}'(u, v, t) = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \boldsymbol{P} \cdot \begin{pmatrix} \boldsymbol{s}(u,v,t) \\ 1 \end{pmatrix}. \tag{8}$$

Because this process is affine, lines are projected onto lines and thus triangles onto triangles.

The homogeneous detector coordinates (also called screen coordinates in computer graphics) have two interesting properties which will be used in the following: First, the coordinate system is now aligned with the detector orientation. And second, the last coordinate of the projected 2-D point represents the depth of the respective 3-D point, i.e., its distance from the detector pixel. In this manner, one can easily check whether the normal of a triangle is pointing towards the detector or not. The normal $\boldsymbol{n}$ of triangle T_1 is computed as

$$\boldsymbol{n} = [\boldsymbol{s}'(u+1, v, t) - \boldsymbol{s}'(u, v, t)] \times [\boldsymbol{s}'(u+1, v+1, t) - \boldsymbol{s}'(u, v, t)], \tag{9}$$

where $\times$ denotes the vector product. The direction d is then computed as

$$d = \operatorname{sign}\left(\boldsymbol{n} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}\right) = \operatorname{sign}(n_3) \tag{10}$$

where $\cdot$ denotes the inner product.

In order to determine the coordinates (x, y, z) of $\boldsymbol{s}'(u, v, t)$ on the detector, we set

$$x := \frac{x'}{z'}$$

$$y := \frac{y'}{z'} \quad \text{and}$$

$$z := \sqrt{z'^2 + x^2 + y^2}\,.$$

Note that z' contains depth information in the direction that is parallel to the principal ray. Hence z' is

only correct for the principal ray. Here we assume that the principal ray hits the detector at (0, 0). We are, however, interested in the actual path length of the ray. Thus, we need to correct this with the orientation of the ray. Note that in a scenario without noise and the assumption of a single x-ray energy, we can directly compute a cosine-weighted projection image using

$$z := z'.$$

C. DRAWING ONTO THE APPEND BUFFER

In normal rendering, the triangles would be drawn into the screen buffer of the graphics card after they were transformed to detector coordinates (also called screen coordinates in computer graphics). Usually, a scan line algorithm is applied for this process. The algorithm determines the pixels affected by each triangle efficiently and calls a program to draw each pixel onto the screen buffer which represents the x-ray detector in our case. The depth values of the drawn pixels are interpolated from the vertices of the triangle [32]. In computer graphics, the algorithm also determines the colour that has to be drawn onto the screen. In our case, we replace this by a more general term, $data$, which reflects the ID of the drawn object, its depth z, and its direction d (cf. Eq. 10).

In most cases in computer graphics, objects are opaque and hence the viewing ray cannot penetrate more than the first intersection with the scene. Only one intersection with the scene is relevant for the image formation process. As all triangles are drawn in parallel into the screen buffer, the correct observation for each pixel (x, y), i.e. only the closest triangle, has to be drawn onto the screen buffer $S(x, y)$ at the respective position. Therefore, a so-called z-buffer $Z(x, y)$ is introduced. $Z(x, y)$ is initialized to a maximum value. Then, all triangles are drawn onto the screen buffer using Algorithm A1 shown below. The respective pixel is only updated if the currently considered pixel/triangle is closer to the screen than the one drawn before. Note that the comparison and update of the z-buffer must be performed as an atomic operation, i.e., the operations must be executed without interruption by another thread. Otherwise, the outcome of the algorithm is non-deterministic.

Algorithm A1

```
draw_z_buffer (x, y, z, data) {
  if (Z(x, y) > z){
    Z(x, y) := z
    S(x, y) := data
  }
}
```

Instead of only one intersection per pixel we have to store a list of all intersections per pixel which is dynamically created during the painting process. If the number of intersections is uniformly distributed over the image, the use of a k-buffer is useful [24]. The k-buffer uses memory usually used for anti-aliasing for the storage of these lists. A drawback of the k-buffer method is that a maximum number of intersections per pixel has to be chosen in advance. If this maximum is exceeded, compensation strategies have to be developed in order to create a correct image. For each pixel an array is allocated that is large enough to contain the maximum number of intersections. Therefore, the algorithm is capable of drawing only a limited number of objects in the same field of view. In contrast, the use of an append buffer [25] allows arbitrary distribution of the number of intersections across the detector. Our algorithm uses a global array – the append buffer – in which a linked list per pixel is stored. Hence, the algorithm is only limited by the total number of intersections, i.e., the size of the append buffer. The algorithm requires three buffers:

1. the append buffer $A(i)$,
2. a global pointer $tail$ which stores the position of the last entry to the append buffer,
3. and a field $F(x, y)$ that points to the last entry for the pixel at location (x, y). This field has $U \times V$ entries.

At the initialization of the projection, all fields are initialized with 0. Then Algorithm A2 is executed for every processed pixel: First the next free index in the append buffer is retrieved by reading the $position$ from $tail$ and incrementing it in an atomic operation. Next, the index of the last entry for pixel (x, y) is read from $F(x, y)$ while $F(x, y)$ is updated to $position$ in an atomic operation. Then the data can be

written into the append buffer. Note that we store two elements for each intersection. In the first element (at $A(2 * position)$), we store the data, while in the second element (at $A(2 * position + 1)$) we store the index of the previous buffer entry for this pixel to create a linked list.

```
Algorithm A2

draw_append_buffer (x, y, z, data) {
  atomic increment {
    position: = tail
    tail + +
  }
  atomic exchange {
    lastpos := F(x, y)
    F(x, y): = position
  }
  A(2 * position): = data
  A(2 * position + 1): = lastpos
}
```

After the drawing is finished, the data can then be read from the append buffer into a single array $q(i)$ per pixel, using Algorithm A3: The variable $position$ is initialized with the last element in the field $F(x, y)$. Then each intersection of the ray through the pixel (x, y) is read from the append buffer successively until the end at $position = 0$ is reached.

```
Algorithm A3

read_append_buffer (x, y, q) {
  position: = F(x, y)
  i: = 0
  while (position > 0) {
    q(i): = A(2 * position)
    position := A(2 * position + 1)
    i + +
  }
}
```

In this manner, a complete list of all intersections with the scene is obtained for every pixel.

As the atomic access to the global memory is quite slow, we also implemented a second variant of the

algorithm that uses local buffering in each thread that draws the triangles. Global memory access is only performed for the access to $F(x, y)$ and at the end of a thread when data for a complete chunk of intersections is written to the append buffer. Note that the local buffering strategy is only suitable for rather small triangles, as the number of drawn pixels per triangle must not exceed the size of the local buffer. In our case the local buffer could store only 1000 intersections per triangle.

Another modification that is required is the drawing algorithm itself. First of all, we store the depth values in 32 bit floating-point values. If not scaled correctly, integer-valued depth information may introduce artefacts in the resulting reconstruction.

Additional parts of the drawing routine also require alteration. In computer graphics (e.g. OpenGL), triangles are usually drawn into the screen buffer by determination of the three closest raster points. During this process the floating point values are transformed into integer pixel coordinates. Then, a 2-D drawing routine is applied that is optimized for drawing speed based on a raster grid. As this approach is approximate, it is often combined with so-called anti aliasing where the projection is actually generated in a higher resolution which is then reduced to the desired detector grid. In the following, we will refer to this as the traditional triangle drawing technique.

In our case, this traditional procedure is sub-optimal, since the 3-D position of the triangle is affected by this procedure. In 3-D space, the deletion of the floating point information is equivalent to moving the original point less than a detector cell. This change is invisible in a single projection image. The cropping of the floating point information is dependent on the position of the detector grid in 3-D as the points are projected onto their nearest neighbours with respect to the grid. This operation is dependent on the viewing angle. Thus, the triangle's location is different in every view which will introduce artefacts in the subsequent reconstruction that are not visible in the DRRs. For this work, we chose to check every raster point within the 2-D bounding box of the triangle to be drawn. Only those raster points that were inside the true 3-D triangle corners, as represented by floating-point values, were drawn onto the detector.

As we draw onto a raster image, the correct depth-value has to be computed for each raster point. We do this by estimation of a plane that fits the three corners of the triangle to the respective z^* coordinate:

$$z^* = m_0 + m_1 x^* + m_2 y^*$$

In this case, x^*, y^*, and z^* denote the coordinates on the raster grid, while $m_0 \ldots m_2$ are the plane parameters that are estimated for each triangle. Each raster point is then drawn with the correct depth value into the append buffer.

While the last two modifications would seem to cause additional computational effort, we did not measure an appreciable increase in runtime during testing. This effect is likely to be related to the append buffer drawing procedure that relies on atomic access to the memory. Thus additional computations do not increase runtime as the problem is memory-bandwidth bound.

D. SCENE RESOLUTION

Once the intersections for each pixel have been determined, the segments that the ray passes through have to be computed, requiring several additional steps. First, the intersections have to be sorted according to their depth along the ray.

Due to the way that the image was created, the points at the boundary between two triangles may be drawn twice, hence these duplicates have to be eliminated. Another problem in the procedure is occurrence of multiple intersections with the same object along the ray (cf. Figure 2). The figure shows two rays with four intersections each. While Ray A enters and leaves the object only once, Ray B enters and leaves the same object twice. Although all four intersections are at the same relative position, Ray A has a much longer path through the material. Fortunately, the direction of the face normals can be used to resolve this contradiction: An entry point is found if the direction d (cf. Eq. 10) is negative and an exit point is found if it is positive. Only alternating entry and exit points of the same object are considered.

In some cases objects do not completely overlap other objects. Hence, it may occur that a ray enters Object A, then Object B, and exits Object A before Object B. In these cases, it is unclear at what point the ray is inside Object A and at what point it is inside Object B (cf. Figure 3). The only way to resolve this is to assign a priority to each object. Then the contradiction can be resolved: If the ray is inside multiple objects, only the object of highest priority is relevant.

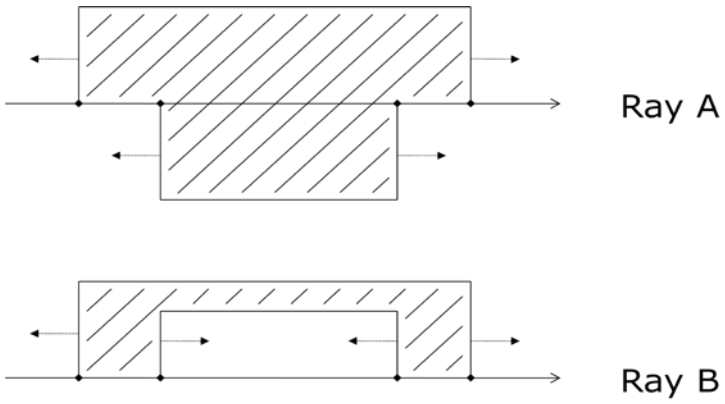

Figure 2: The direction of the face normals is used to resolve multiple intersections along the same ray.

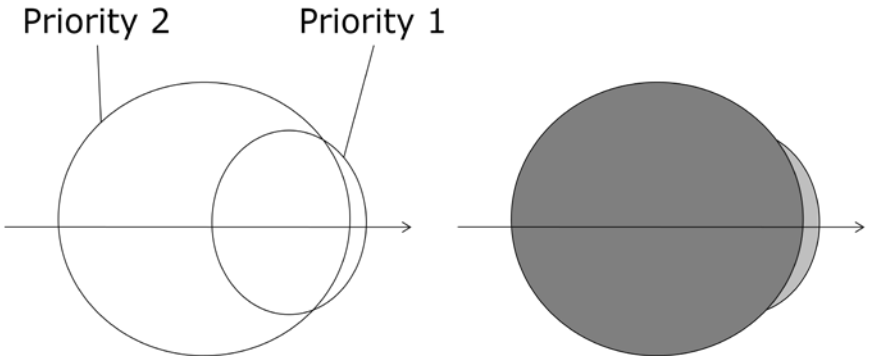

Figure 3: Each object has a priority that describes which shape lies on top: Only the shape with the highest priority is drawn.

The application of the previously described steps yields a list of line segments $r(x, y, i)$ that describe the path of the ray through the scene.

E. ABSORPTION MODELING

As the path of the ray through the scene is now resolved, the correct absorption value for each pixel can be determined according to Lambert-Beer's Law [8]

$$N(x,y) = \sum_{b=0}^{B-1} N_b \cdot e^{-\sum_i \mu(r(x,y,i),E(b)) * \delta(r(x,y,i))},$$

where $N(x,y)$ is the number of detected photons in the respective detector cell, N_b the number of initial photons emitted from the source at energy $E(b)$, $\mu(r(x,y,i),E(b))$ is the energy-dependent absorption value of segment $r(x,y,i)$, $\delta(r(x,y,i))$ its length, and B the number of energy bins. Note that in order to resolve $\mu(r(x,y,i),E(b))$ only the ID of the material that describes segment $r(x,y,i)$ is required. The actual value is determined from a look-up table as provided for example in reference [33]. If only one energy bin is chosen, the formulation simplifies to the monochromatic case without summation (B=1).

III. RESULTS

In the following, we evaluate the performance of our projection generation pipeline at three levels. On the lowest level, we investigate whether the time-variant surface spline tessellation creates acceptable results. We do this by comparison to state-of-the-art 1-D and 2-D tessellation algorithms.

On the next level, we evaluate whether the projection generation is computationally efficient in terms of runtime for different tessellation grid sizes.

With these intermediate results showing acceptable error margins, we can then evaluate the method for the task of 3-D reconstruction.

All experiments were conducted using a Nvidia Quadro FX 5800 graphics card.

A. SPLINE TESSELLATION

Three sample splines were tessellated 1000 times at one million locations each and the average computation time was measured. As the 1-D spline shape, we chose a 3-D curve with 12 control points. For 2-D, the same shape was extended to a curved surface but with 12x8 control points. In the 3-D tessellation task, the right atrium of XCAT was chosen with 31x12x35 control points. This is a common number of control points for the XCAT phantom. We also implemented the same algorithms on the CPU using the Java programming language in double precision as a baseline system for accuracy comparison. All GPU experiments were performed in single precision.

Table 1: Overview of the results using hardware acceleration: Tessellation was performed with one million sample points each. The dimensionality refers to the internal dimension of the spline: 1-D is a curved line, 2-D a surface, and 3-D a time-variant surface.

1-D	Time [ms]	Accuracy [mm]
CPU (Java)	278	-
Open CL	13	4.24 E-7
Open CL Texture	7	0.0018

2-D	Time [ms]	Accuracy [mm]
CPU (Java)	1097	-
Open CL	27	9.63 E-7
Open CL Texture	1	0.0031

3-D	Time [ms]	Accuracy [mm]
CPU (Java)	4146	-
Open CL	81	4.35 E-5
Open CL Texture	10	0.028

Table 1 reports the results. As expected the CPU implementation using the Java virtual machine and double precision was significantly slower than the GPU implementation. The main point of the CPU implementation was to get a baseline result in terms of computational accuracy. In the 1-D case the accuracy of the OpenCL implementation is within floating point precision. Hence, there is no loss of accuracy; this is also the case for the 2-D spline surface implementation. Interestingly, a slight loss of accuracy is found in the 3-D OpenCL implementation. A much higher loss in terms of accuracy is found as soon as texture units are used in the computation. In the 3-D case, the accuracy already degrades to 0.028 mm. Although this is a dramatic increase, the accuracy is still sufficient for our application, which targets imaging with a detector pixel size of at least 0.15 mm. Hence, we do not expect any visible artifact from the accelerated tessellation. In terms of computation times, the texture units show very good results. The time for one million evaluations in 3-D is 10 ms.

B. PROJECTION GENERATION

Next, we evaluated the speed of the projection generation using a more realistic scenario. We generated monochromatic projection data of 23 time-variant surface splines of the XCAT phantom [14]. Each spline surface was sampled with a grid of $U \times V$ points with $U, V \in [50, 70, 100]$ at time $t = 0$ and drawn into a

640×480 projection image (cf. Figure 4). In total we looked at five different setups of the simulation routine:

- A traditional 2-D raster image triangle drawing routine using depth value encoding in integer values in [0.5 mm]. This drawing routine represents a projection data generation routine that employs a course quantization. As the triangles were rather small, we used a local buffering strategy to fill the append buffer. The algorithm is denoted as "depth as integer" in the following.
- The same triangle drawing routine as in "depth as integer", but with a floating-point 32 bit depth encoding (denoted as "no anti alias" in the following).
- The same triangle drawing routine as in "depth as integer", with floating-point 32 bit depth encoding and 4-fold anti aliasing, i.e. the image is actually generated in a resolution that is four times higher in each dimension. After drawing, the image is then resampled to its intended size (denoted as "4x anti alias" in the following). One effect of the anti aliasing is that the area of the triangles is increased by a factor of 16 that inhibits a local buffering strategy. Thus a global buffering strategy had to be chosen.
- The proposed method using correct depth and triangle values (denoted as "proposed" in the following, cf. Section II.C). For the proposed method, a local buffering strategy could be employed.
- A CPU-based ray-casting technique that we refer to as the gold standard (Denoted as "CPU" in the following).

Except for the CPU version, all algorithms were implemented on graphics hardware. Runtime was measured including all steps to generate the projection (tessellation, drawing, and evaluation of the absorption model). Memory transfer times were not included.

Table 2 displays an overview of the computation times. For a grid of 50×50 sample points the local append buffer strategy failed, because with increasing area of the triangles, the number of intersections per thread becomes too large and exceeds the size of the local memory. Hence, the experiments for "depth

as integer", "no anti alias", and "proposed" could not be conducted. Instead, we reported the corresponding numbers using a global buffering strategy in brackets. Note that the global buffering strategy increases runtime significantly due to the large number of atomic operations.

With larger grid sizes, i.e. smaller triangles, a clear trend towards the local buffering strategy can be seen. With a growing number of triangles, there is only a slight increase in ray-object-intersections, which causes the number of drawn points in the append buffer to increase. Hence, the best computational performance is found at a grid size of 70×70 pixels here. However, the increase in runtime to increase to a 100×100 grid is only minor.

The ray-casting-based CPU method has a dramatic increase in runtime with a growing number of triangles, as each ray is intersected with each triangle.

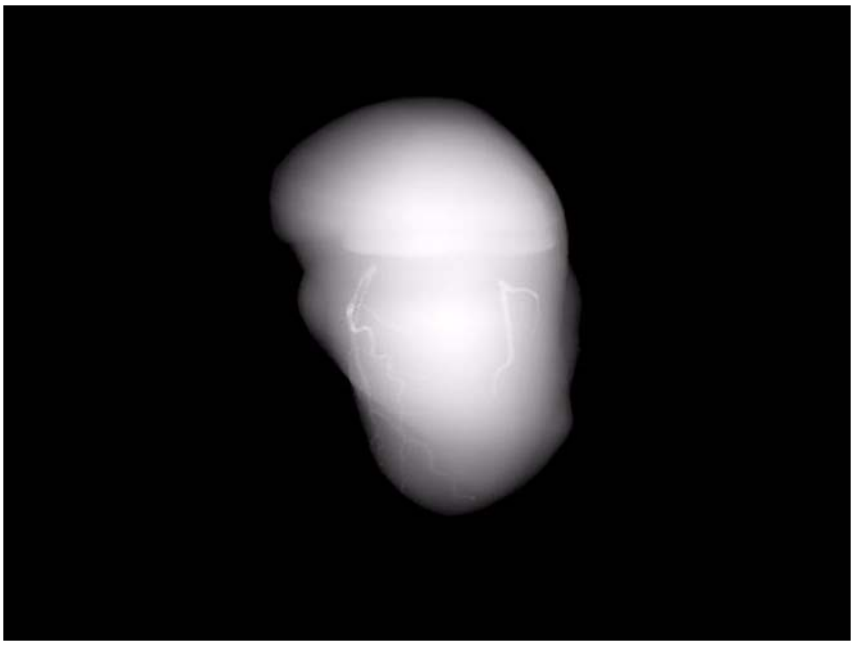

Figure 4: Sample scene of an animated heart-like shape with 23 spline surfaces ($\boldsymbol{t} = \mathbf{0}$). We increased the contrast of the coronary arteries for demonstration purposes.

Table 2: Computation times using the heart scene at time $\boldsymbol{t = 0}$ at a resolution of $\boldsymbol{640 \times 480}$. The different columns report different rasterization sizes for the splines. The total computation time of the proposed method for $\boldsymbol{100 \times 100 \times 23 \times 2 = 460000}$ triangles is 254 ms (using the local buffering scheme). In cases where the local buffering strategy failed, we report the computation time for the global strategy in brackets.

X-Cat Heart (23 Splines)	Buffering Strategy	50 x 50 Time [ms]	70 x 70 Time [ms]	100 x 100 Time [ms]
Depth as integer	local	- (477)	168	194
No anti alias	local	- (477)	152	193
4x anti alias	global	6552	6700	6903
Proposed	local	- (405)	165	254
CPU	-	47807	82074	142975

C. RECONSTRUCTION ACCURACY

Lastly, we evaluated the effect of different projection methods on the reconstruction quality. Although we aim at modelling dynamic scenes, we chose to evaluate this using a static scene, as we wanted to be independent of the accuracy of a motion compensated reconstruction algorithm. We generated 360 monochromatic projections (640×480 pixels, detector element size 0.75×0.75 mm^2, source detector distance 120 cm, source to center of rotation distance 80 cm) over 360 degrees of 23 time-variant spline surfaces of the XCAT phantom. The diameter of the heart was approximately 15 cm. The phantom contains only two relevant materials: Blood and heart tissue. Both were simulated as water-like material at a density of 1.06 [g/cm^3] for blood and 1.05 [g/cm^3] for the heart tissue. The contrast in Figure 4 was simulated as dense water with 4.0 [g/cm^3]. As image quality is improved with a higher grid size, we chose to use the 100×100 tessellation grid. We chose 70 kV for the monochromatic energy for projection generation.

Reconstruction was performed using an FDK-type reconstruction [8] in a 256^3 voxel grid. The voxel size was chosen to be 1 mm in each dimension. No noise was added to the projections, in order to investigate only the geometric error that was introduced by the simulation. We chose a Shepp-Logan kernel in the

filtering step.

Furthermore, we directly computed cosine-weighted projection images (cf. Section II.B) in the graphics hardware implementations. The approaches “depth as integer”, “no anti alias”, and “proposed” showed similar performance as in the second experiment. The approach “4x anti alias” had an execution time that was about 24 times higher than the proposed approach. The runtime of the CPU version was 14.2 hours which was the reason to develop the proposed method in the first place.

Figure 5 displays qualitative results of the different algorithms. While all five versions seem to yield satisfactory projection images (top row), the quality of the reconstructed images varies dramatically. Only correct modelling of the drawing process provides an accurate reconstructed image as seen in the proposed and the CPU methods. While anti aliasing helps to reduce image artefacts, some error still remains as shown when using the narrow visualization window presented in the last row.

The difference in the generated projection images is emphasized in Figure 6. The figure shows line profiles created with the different algorithms. While the artifacts in “depth as integer” are clearly visible, only slight differences between “no anti alias” and the CPU version can be found. The “4x anti alias” and “proposed” approaches show almost no difference to the CPU version.

Similar observations can also be made when looking at quantitative measures in the reconstructions reported in Table 3. We measured the noise that was caused by the reconstruction method as the standard deviation in the left ventricle (the top right circle in the presented slice). Only “4x anti alias”, “proposed”, and “CPU” yield standard deviations below 1 HU. The proposed method yields almost as little noise as the CPU method while “4x anti alias” has almost twice the noise. In terms of runtime, the proposed method is in the same range as the traditional triangle-based methods “depth as integer” and “no anti alias”.

Table 3: Measurements of image noise caused by the reconstruction method with a 100x100 sampling.

Heart Scene (23 Splines)	Mean [HU]	Standard Deviation [HU]	Total Runtime	Speed-up factor (vs. CPU)
Depth as integer	59.4	8.76	77.9 s	656.2 x
No anti alias	59.5	2.50	77.9 s	656.2 x
4x anti alias	60.5	0.735	2428.1 s	21.1 x
Proposed	60.5	0.468	98.5 s	519.0 x
CPU	60.9	0.404	14.2 h	1.0 x

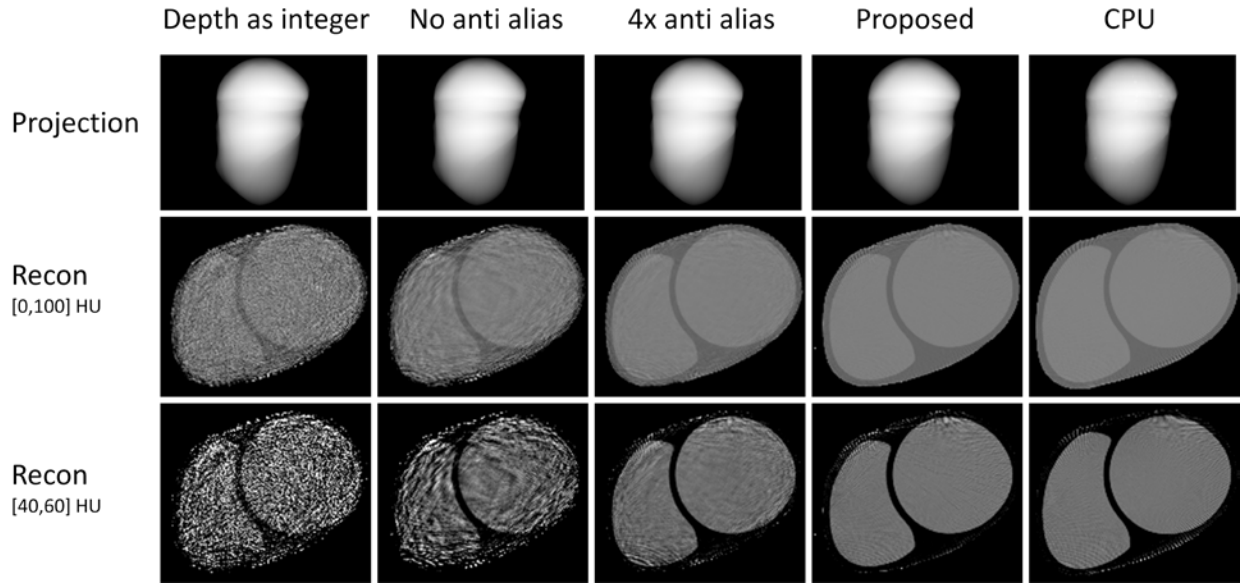

Figure 5: Results of the different simulation methods. Anti aliasing helps to increase reconstruction quality. Accurate processing, however, yields a quality that is similar to the ray-casting approach on the CPU.

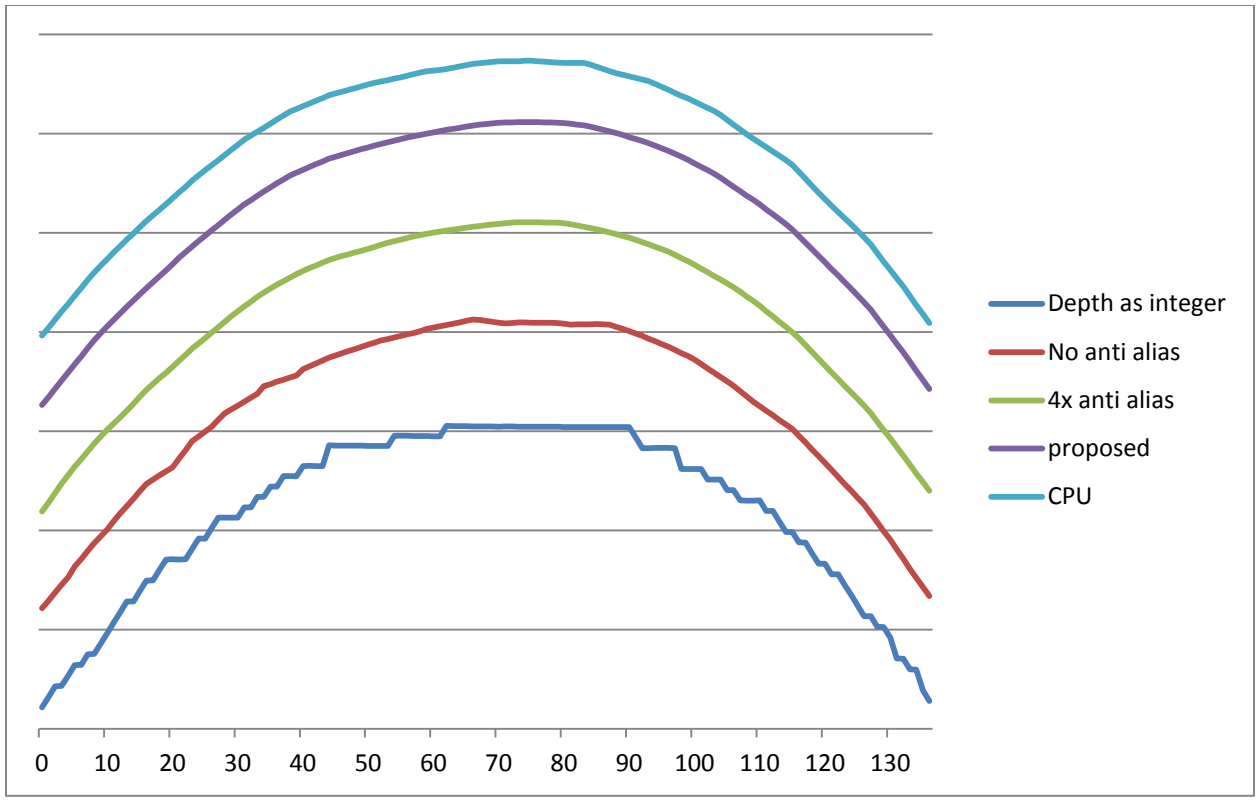

Figure 6: Comparison of line profiles through the projection images. While the effects in the projection image can still be observed clearly for the "depth as integer" and "no anti alias", the differences between "proposed" "4x anti alias" and "CPU" are almost invisible in the projection image profile. Note that for visualization purposes, the contours are displayed with gaps in between, as their overlap inhibits comparison.

IV. DISCUSSION

Tessellation of the spline surfaces was extremely successful. All errors were within acceptable margins even using texture memory. Interestingly, there was almost no loss in runtime when changing from 1-D to 3-D. We ascribe this effect to the fact that each spline evaluation was called individually and the time to copy the evaluated points into the computer's main memory was included in the runtime measurement. The evaluation costs almost no time compared to the memory transfers. Compared to the execution time of the remaining processing the runtime of the tessellation is negligible. Thus, tessellation time was not explicitly measured in the other experiments.

For our use case, we assume that the projections that are generated will be used to create a specific reconstruction. Thus, the number of sampling points will be dependent on the reconstruction grid and the size of the tessellated organ. For the present paper, we chose a fixed number of sampling points per spline

in order to enable a better comparison with respect to runtime. As a rule-of-thumb, we picked the number of sampling points such that the spacing between points was approximately equal to the resolution of the reconstruction grid. Thus, we suggest that the number of sampling points be increased for large organs and decreased for smaller organs. In order to prevent long triangles, we furthermore suggest picking the longest edge of the bounding box of the organ to determine the sampling points. This heuristic has been used successfully [34][35] [36][37] [38].

There are two main limitations of the append buffer strategy. The first one is graphics memory. At present we used projection images of 640x480 pixels. This corresponds to 1.17 MB in floating point accuracy. With a consumer graphics card with 512 MB of memory this allows us to have about 400 intersection points on average per pixel. With respect to the phantom this implies that there should be no more than 200 convex shapes on average on the intersection path of the ray. If the objects are non-convex this number is even lower. With respect to commonly used phantoms, like XCAT, this number should be sufficient as most rays intersect only 10 to 15 shapes.

The second limitation is the atomic access to the append buffer. With the global access scheme runtime will increase with the number of drawn points as the append buffer pointer is accessed in a global atomic way. This inhibits parallelization. However, as shown in the experiments section, the problem could be reduced by using a local buffering strategy which better preserves parallelizability.

At present we focussed our pipeline on flat panel detector geometries. In 3rd generation CT scanners, however, cylindrical detectors are used. At present the pipeline does not handle this kind of geometry. However, this is not a fundamental problem of the software architecture, as it is designed in a modular fashion. In order to implement cylindrical detectors, one would have to adapt the projection in Step 2 of Figure 1 to a cylindrical geometry. Furthermore, one would have to incorporate the cylindrical shape of the detector into Step 3 of Figure 1. There the drawing routine would need to incorporate the curved shape of the detector into the interpolation process.

The simulation of a focal spot of finite size could also be realized with the software by computation of multiple images for discrete focal spot positions. The final image with finite focal spot size is then

obtained as a weighted sum of the different focal spot positions.

V. SUMMARY

In this paper, we propose a fast simulation method for x-ray absorption images. The method adapts several techniques that have already been successfully applied in computer graphics. We use hardware tessellation to generate a mesh of triangles. In the state-of-the-art in computer graphics this is performed for static 2-D surfaces. We have extended this to moving 2-D surfaces. The triangles are then drawn into an append buffer that is used to keep track of all intersections of a pixel with the scene. As we require more information in the subsequent absorption model, this process was augmented to enable use of x-ray material property descriptors. Once all intersections within the scene are available, an absorption model is applied directly on the graphics card to generate an x-ray image. Using this method we are able to generate x-ray projection image simulations of realistically moving objects like the beating heart within 254 ms with an accuracy that is suitable for CT reconstruction.

The software and source code that was created for this article can be found at http://conrad.stanford.edu.

ACKNOWLEDGMENTS

We thank Christian Eisenacher for his ideas and Paul Segars for his supportive comments during the preparation of this manuscript and Christoph Forman for his support in setting up the web site for this article.

This work was supported in part by NIH under grants R01EB003524 and R01HL087917, and the Lucas Foundation. The authors gratefully acknowledge funding of the Erlangen Graduate School in Advanced Optical Technologies (SAOT) by the German Research Foundation (DFG) in the framework of the German excellence initiative.

REFERENCES

[1] Fahrig R, Dixon R, Payne R, Morin RL, Ganguly A, Strobel N, "Dose and image quality for a cone-beam C-arm CT system," *Med Phys*, vol. 33, pp. 4541-50, 2006.

[2] Hausleiter J, Meyer T, Hadamitzky M, Huber E, Zankl M, Martinoff S, Kastrati A, Schömig AT, "Radiation Dose Estimates From Cardiac Multislice Computed Tomography in Daily Practice," *Circulation*, vol. 113, pp. 1305-1310, 2006.

[3] Borsdorf A, Raupach R, Flohr T, Hornegger J, "Wavelet based Noise Reduction in CT-Images Using Correlation Analysis," *IEEE Transactions on Medical Imaging*, vol. 27, no. 12, pp. 1685-1703, 2008.

[4] Wang J, Li T, Lu H, Liang Z, "Penalized weighted least-squares approach to sinogram noise reduction and image reconstruction for low-dose X-ray computed tomography," *IEEE Trans Med Imaging*, vol. 25, pp. 1272-83, 2006.

[5] Zhu L, Wang J, Xing L, "Noise suppression in scatter correction for cone-beam CT," *Medical Physics*, vol. 36, pp. 741-52, 2009.

[6] Maier A, Wigström L, Hofmann HG, Hornegger J, Zhu L, Strobel N, FahrigR, "Three-dimensional Anisotropic Adaptive Filtering of Projection Data for Noise Reduction in Cone Beam CT," *Med. Phys.*, vol. 38, no. 11, pp. 5896-5909, 2011.

[7] La Riviere PJ, "Penalized-likelihood sinogram smoothing for low-dose CT," *Medical Physics*, vol. 32, pp. 1676-83, 2005.

[8] Buzug T, *Computed Tomography - From Photon Statistics to Modern Cone-Beam CT*. Berlin, Germany: Springer, 2008.

[9] Allison J, Amako K, Apostolakis J, Araujo H, Dubois PA, Asai M, Barrand G, Capra R, Chauvie S, Chytracek R, Cirrone GAP, Cooperman G, Cosmo G, Cuttone G, Daquino GG, Donszelmann M, et al., "Geant4 developments and applications," *IEEE Transactions on Nuclear Science*, vol. 53, no. 1, pp. 270-278, 2006.

[10] Shepp LA, Logan BF, "The Fourier reconstruction of a head section," *IEEE Transactions on nuclear science*, vol. 21, pp. 21-43, 1974.

[11] Segars WP, Mahesh M, Beck TJ, Frey EC, Tsui BMW, "Realistic CT simulation using the 4D XCAT phantom," *Med. Phys.*, vol. 35, pp. 3800-3808, 2008.

[12] Fung GSK, Segards WP, Gullberg GT, Tsui BMW, "Development of a model of the coronary arterial tree for the 4D XCAT phantom," *Physics in Medicine and Biology*, vol. 56, pp. 5651-5663, 2011.

[13] Segars WP, Lalush DS, Tsui BWM, "Modeling Respiratory Mechanics in the MCAT and Spline-Based MCAT Phantoms," *IEEE Trans Nucl Sci*, vol. 48, no. 1, pp. 89-97, 2001.

[14] Segars WP, Lalush DS, Tsui BMW, "A Realistic Spline-Based Dynamic Heart Phantom," *IEEE Trans Nucl Sci*, vol. 46, no. 3, pp. 503-506, 1999.

[15] Fung GSK, Taguchi K, Tsui BMW, Stierstorfer K, Flohr TG, Segars WP, "XCAT/DRASIM: a realistic CT/human-model simulation package," in *Proc. SPIE 7961*, Lake Buena Vista, FL, USA , 2011, p. 79613D.

[16] Keil A, *Dynamic Variational Level Sets for Cardiac 4D Reconstruction*. Munich, Germany: Disstertation, TU Munich, 2010.

[17] Prümmer M, *Cardiac C-Arm Computed Tomography: Motion Estimation and Dynamic Reconstruction*. Erlangen: Dissertation, Universität Erlangen-Nürnberg, 2009.

[18] Rohkohl C, Lauritsch G, Keil A, Hornegger J, "CAVAREV—an open platform for evaluating 3D and 4D cardiac vasculature reconstruction," *Physics in Medicine and Biology*, vol. 55, pp. 2905-2915, 2010.

[19] Scherl H, Keck B, Kowarschik M, Hornegger J, "Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA)," in *In IEEE Nuclear Science Symposium, Medical Imaging Conference Record*, Honolulu, HI, United States, 2008, pp. 6:4464-4466.

[20] Rohkohl C, Keck B, Hofmann H, Hornegger J, "RabbitCT - an open platform for benchmarking 3D cone-beam reconstruction algorithms," *Med. Phys.*, vol. 36, no. 9, pp. 3940-3944, 2009.

[21] Siegl C, Hofmann HG, Keck B, Prümmer M, Hornegger J, "OpenCL, a Viable Solution for High-performance Medical Image Reconstruction?," in *Proc. SPIE 7961*, Lake Buena Vista, FL, USA, 2011, p. 79612Q.

[22] Sigg C, Hadwiger M, "Fast Third-Order Texture Filtering," in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Matt Pharr, Ed.: Addison-Wesley, 2005, ch. 20, pp. 313-329.

[23] Myers K, Bavoil L, "Stencil routed A-Buffer," in *ACM SIGGRAPH 2007 sketches*, San Diego, California, 2007, p. no pagination.

[24] Bavoil L, Callahan SP, Lefohn A, Comba JLD, Silva CT, "Multi-Fragment Effects on the GPU using the k-Buffer," in *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, 2007, pp. 97-104.

[25] Yang JC, Hensley J, Grün H, Thibieroz N, "Real-Time Concurrent Linked List Construction on the GPU," *Computer Graphics Forum*, vol. 29, no. 4, pp. 1297-1304, 2010.

[26] Spoerk J, Bergmann H, Birkfellner W, Wanschitz F, Dong S, "Fast DRR splat rendering using common consumer graphics hardware," *Medical Physics*, vol. 34, pp. 4302-4308, 2007.

[27] Villard PF, Vidal FP, Hunt C, Bello F, John NW, Johnson S, Gould DA, "A prototype percutaneous transhepatic cholangiography training simulator with real-time breathing motion," *International journal of computer assisted radiology and surgery*, vol. 4, no. 6, pp. 571-578, 2009.

[28] Piegl LA, Tiller W, *The NURBS Book (Monographs in Visual Communication)*.: Springer; 2nd edition, 1996.

[29] Ruijters D, Romeny BMH, Suetens P, "Accuracy of GPU-based B-Spline Evaluation," in *Proc. Tenth IASTED International Conference on Computer Graphics and Imaging (CGIM),* Innsbruck, Austria, 2008, pp. 117-122.

[30] Ruijters D, Romeny BM, Suetens P, "Efficient GPU-Based Texture Interpolation using Uniform B-Splines," *Journal of Graphics, GPU, and Game Tools*, vol. 13, no. 4, pp. 61-69, 2008.

[31] Hartley R, Zisserman A, *Multiple View Geometry in computer vision*, 2nd ed. Cambridge, United Kingdom: Cambridge Univ. Press, 2003.

[32] Foley JD, van Dam A, Feiner SK, Hughes JF, *Computer Graphics: Principles and Practice in C*, 2nd ed.: Addison-Wesley, 1995.

[33] Hubbell JH, Seltzer SM, "Tables of X-Ray Mass Attenuation Coefficients and Mass Energy-Absorption Coefficients from 1 keV to 20 MeV for Elements Z = 1 to 92 and 48 Additional Substances of Dosimetric Interest," *NISTIR*, vol. 5632, 1989.

[34] Choi JH, Maier A, Keil A, Fahrig R, "Acquisition of 3D Knee Geometry under Weight-Bearing Conditions using a C-arm CT Scanner," in *Proc. RSNA*, Chicago, IL, USA, 2011.

[35] Bögel M, Maier A, Hofmann HG, Hornegger J, Fahrig R, "Diaphragm Tracking in Cardiac C-Arm Projection Data," in *Proc. BVM*, Berlin, Germany, 2012, pp. 33-38.

[36] Bögel M, Maier A, Hofmann HG, Hornegger J, Fahrig R, "Diaphragm Tracking for Respiratory Motion Compensated Cardiac C-Arm CT," in *Proc. CT Meeting*, Salt Lake City, UT, USA, 2012, pp. 13-16.

[37] Bögel M, *Diaphragm Tracking for Respiratory Motion Compensated C-arm Cardiac Reconstruction*. Erlangen, Germany: University of Erlangen-Nuremberg, 2011.

[38] Mueller K, Zheng Y, Lauritsch G, Rohkohl C, Schwemmer C, Maier AK, Fahrig R, Hornegger J, "Evaluation of interpolation methods for motion compensated tomographic reconstruction for cardiac angiographic

535