The definitive version is available at `http://http://diglib.eg.org/`.

F. P. Vidal, M. Garnier, N. Freud, J. M. Létang, and N. W. John: Simulation of X-ray Attenuation on the GPU. In *Proceeding of TPCG 2009 - Theory and Practice of Computer Graphics*, pp. 25-32, 17-19 June 2009 Cardiff, UK. ISBN 978-3-905673-71-5. **Winner of Ken Brodlie Prize for Best Paper**

DOI: 10.2312/LocalChapterEvents/TPCG/TPCG09/025-032

ACM CCS: I.3.5 Computer Graphics: Physically based modeling; I.3.7 Computer Graphics: Raytracing; J.2 Computer Applications: Physics.
Keywords: Three-Dimensional Graphics and Realism, Raytracing, Physical Sciences and Engineering, Physics.


```
@inproceedings{Vidal2009TPCG,
   Author = {F. P. Vidal and M. Garnier and N. Freud and
      J. M. Létang and N. W. John},
   Title = {Simulation of X-ray Attenuation on the GPU},
   Booktitle = {Proceeding of TCPG'09 - Theory and Practice of Computer
      Graphics},
   Month = jun,
   Year = 2009,
   Annotation = {Cardiff, UK, 17~---~19 Juin 2009},
   Pages = {25-32},
   Publisher = {Eurographics},
   doi = {10.2312/LocalChapterEvents/TPCG/TPCG09/025-032},
   Abstract = {In this paper, we propose to take advantage of computer
      graphics hardware to achieve an accelerated simulation of
      X-ray transmission imaging, and we compare results with a fast and
      robust software-only implementation. The running times of the GPU
      and CPU implementations are compared in different test cases. The
      results show that the GPU implementation with full floating point
      precision is faster by a factor of about 60 to 65 than the CPU
      implementation, without any significant loss of accuracy. The
      increase in performance achieved with GPU calculations opens up
      new perspectives. Notably, it paves the way for physically-realistic
      simulation of X-ray imaging in interactive time.},
}
```

Simulation of X-ray Attenuation on the GPU

F. P. Vidal [1,*], M. Garnier [2], N. Freud [3], J. M. Létang [3], and N. W. John [4]
[1] Bangor University, Dean Street, UK,
[2] INSA-Rennes, France
[3] INSA-Lyon, France
[*] Now at INRIA - Saclay–Île-de-France, France

Abstract

In this paper, we propose to take advantage of computer graphics hardware to achieve an accelerated simulation of X-ray transmission imaging, and we compare results with a fast and robust software-only implementation. The running times of the GPU and CPU implementations are compared in different test cases. The results show that the GPU implementation with full floating point precision is faster by a factor of about 60 to 65 than the CPU implementation, without any significant loss of accuracy. The increase in performance achieved with GPU calculations opens up new perspectives. Notably, it paves the way for physically-realistic simulation of X-ray imaging in interactive time.

Categories and Subject Descriptors (according to ACM CCS): I.3.5 Computer Graphics: Physically based modeling; I.3.7 Computer Graphics: Raytracing; J.2 Computer Applications: Physics.

Keywords: Three-Dimensional Graphics and Realism, Raytracing, Physical Sciences and Engineering, Physics.

1 Introduction

The simulation of X-ray imaging techniques such as radiography or tomography is extensively studied in the physics community and different physically-based simulation codes are available. Deterministic methods based on ray-tracing are commonly used to compute direct images (i.e. images formed by the X-ray beam transmitted without interaction through the scanned object) of computer-aided design (CAD) models. Ray-tracing provides a fast alternative to Monte Carlo methods [4]. Such programs are very useful to optimize experiment parameters, to conceive imaging systems, or to take into account non-destructive testing during the design of a mechanical structure [1, 10]. However, even with fast ray tracing algorithms, the simulation of complex X-ray imaging systems still requires very long computation times and is not suitable for an interactive use as would be required in a medical training tool.

Physics-based simulations are traditionally performed on CPUs. However, there is a growing interest for general-purpose computation on GPUs (GPGPU) and this has been an active area of research some time [13].

In this paper, we present an efficient simulation of X-ray attenuation through complex objects, that makes use of the capability improvement of today's graphics cards. We also compare the performance of this GPU approach with an efficient software-only implementation. To our knowledge this is the first GPU-based X-Ray attenuation simulation. Such a simulation tool can be deployed in medical virtual interactive applications for training fluoroscopy guidance of needles, catheters and guidewires [18], and can also be useful to speed-up current physics-based simulation where computational accuracy is critical.

The following Section gives an overview of the context and objectives of this work. The implementation of our simulation scheme is described in Section 3. The results and performance comparisons with a software-only implementation are given in Section 4. The last section discusses the work carried out and provides directions for further work.

2 Context and objectives

To date, there are two different kinds of X-ray simulation algorithms:

- probabilistic methods, based on Monte Carlo trials;
- determinist or analytic methods, based on ray-tracing (these include the resolution of the Boltzmann transport equation).

Monte Carlo simulations can produce very accurate X-ray images, but they are computationally expensive, which prevents their use in any interactive applications. For example, to simulate an image consisting of 10^6 pixels, with a noise level of 1%, at least 10^{10} photons have to be cast (depending on the attenuation in the object). This would take days of computation time if using only a single PC. This time can be reduced using a cluster of PCs, a supercomputer, or Grid computing. Pasciak *et al.* show the possibilities of performing Monte Carlo simulations applied to radiation transport using a field-programmable gate array (FPGA) [14]. However, so far no realistic object geometry is implemented. One of the problems encountered is the fact that FPGAs cannot be programmed using standard programming languages, and low level design has to be used at the gate level.

Alternatively, the ray-tracing principle has been adapted to X-ray simulation [7, 4]. Here, all intersections between a ray and an object have to be considered and radiation attenuation is computed by considering the thickness penetrated by the ray going through the object characterized by its density and attenuation coefficient. The 3D scene is typically made up of objects described by triangle meshes. The main reason to use triangle meshes is to make the render process fast as many algorithms in real-time 3D graphics have been developed for such geometry representation, including polygon clipping and filling, etc. and also the classic Z-buffer algorithm to remove hidden faces. A modified version of the Z-buffer, known as the L-buffer (for length buffer), can be used to store the length of a ray crossing a given 3D object [4]. The simulation of radiographic images from CT data sets has been also reported [7, 9]. More recently, volume rendering by ray-casting has been adapted to the realistic simulation of X-rays in a virtual reality environment [12]. Ray-casting can also be used to implement a hybrid determinist/probabilistic approach to compute the dose deposited in cancerous and healthy tissues during radiotherapy treatment [6]. In this case, each voxel corresponds to a cube characterized by its attenuation and energy-absorption coefficients. Using this approach, the attenuation of the incident X-ray beam is computed for each voxel traversed. Laney *et al.* proposed a GPU simulation of based on volume rendering of unstructured data [8]. Using a 3D texture, ray-tracing through voxel data is also possible on GPU to simulate fluoroscopic images [17]. In this method, voxels are processed as parallelepiped boxes. Yan *et al.* adapted GPU volume rendering by ray-casting to generate digitally reconstructed radiographs (DRRs) for image guided radiation therapy (IGRT) [20]. The original ray casting algorithm creates a high quality image by casting a ray for each pixel into the volume and compositing the light reflected back to the viewer from a set of samples along the ray [11]. An alternative adaptation of GPU volume rendering to reconstruct DRRs is splatting [16]. In splatting, voxels are "thrown" at the image in a forward projection, forming a footprint, and the result is accumulated in the image plane [19]. The previous approaches to simulate X-ray images using GPU implementation all make use of volume rendering.

The hypothesis of this work is that using GPUs can provide the real-time simulation of X-ray imaging techniques from surface models and that the simulated results still have all the required numerical accuracy. As the core building block of this type of simulation is the ray tracing algorithm, the work is focused on its implementation using GPUs and comparison with a CPU

implementation of the same method, described by Freud *et al.* [4]. The scope of the validation of our GPU implementation is limited to the assessment of the potential of GPUs to accelerate X-ray imaging simulation and to provide accurate results. In this paper, we consider test cases with a point source of monochromatic X-rays, and homogeneous objects with triangle meshes. Only the directly transmitted photons are simulated, using the X-ray exponential attenuation law. Physically more realistic situations can be simulated in a straightforward manner by introducing additional loops, to take into account polychromatic X-rays or focal spots causing geometric unsharpness [2]. The simple case studied in this work also constitutes the core calculation for more complex simulations involving emission of secondary radiation, such as scattered or fluorescence photons [5], or emission of γ photons by radiotracers in nuclear medicine applications.

3 Simulation algorithm

3.1 Attenuation law

The attenuation law, also called the Beer-Lambert law, relates the absorption of light to the properties of the material through which the light is travelling. The integrated form for a monochromatic incident X-ray beam (i.e. all the incident photons have the same energy) is:

$$N_{out}(E) = N_{in}(E) \times e^{\left(-\int \mu(E,\rho(x),Z(x))\mathrm{d}x\right)} \tag{1}$$

with $N_{in}(E)$ the number of incident photons at energy E, $N_{out}(E)$ the number of transmitted photons and μ the linear attenuation coefficient (in cm^{-1}). μ can be seen as a probability of interaction by unit length. It depends on: i) E - the energy of incident photons, ii) ρ - the material density of the object, and iii) Z - the atomic number of the object material.

3.2 Overview

Specific algorithms can be implemented as shader programs that will be executed directly on the GPU to replace the parts of the fixed graphics rendering pipeline [15]. A shader program is twofold i) a vertex shader (or vertex program) that substitutes major parts of the vertex operations of the fixed function of the geometry processing unit, and ii) a fragment shader (or fragment program) that substitutes major parts of the fragment operations of the traditional fixed function of the rasterization unit. Such programs are written in a shading language such as the OpenGL shading language (GLSL) by the OpenGL Architecture Review Board. More recently Nvidia released CUDA technology to use the standard C language to implement programs that run directly on the graphics processor without the need of a graphics Application Programming Interface (API).

The algorithm presented below has been implemented using GLSL. Figure 1 shows the simulation pipeline. The principle of computing direct images is to emit rays from the X-ray source to every pixel of the detector. For each ray, the total path length through each object is determined using geometrical computations. Finally, the attenuation of X-rays for a given pixel is computed using the recorded path lengths and X-ray attenuation coefficients. Eq. 1 can be written as follows:

$$N_{out} = N_{in} \times \exp\left(-\sum_{i=0}^{i<objs} \mu(i)L_p(i)\right) \tag{2}$$

with $objs$ the total number of objects and $L_p(i)$ the path length of the ray in the i^{th} object. It can be decomposed to illustrate the different rendering passes:

1. compute and store the path length of every object, i.e. $L_p(i)$ in Eq. 2,
2. make use of the first pass to compute $\sum_i \mu(i)L_p(i)$ in Eq. 2,
3. make use of the second pass to compute the number of transmitted photons using the attenuation law.

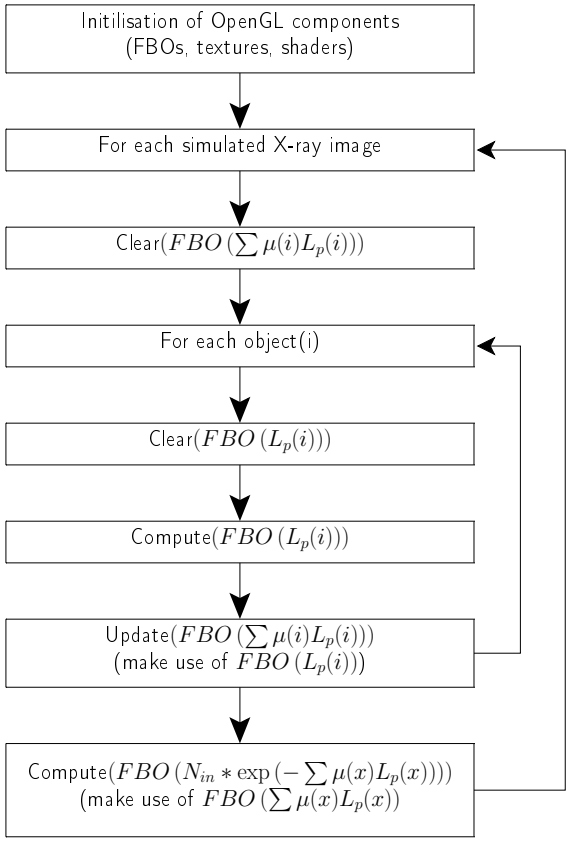


Figure 1: Pipeline to compute the X-ray attenuation.

Multi-pass rendering algorithms are usually implemented using a 2D texture attached to a framebuffer object (FBO), a relatively new extension of the OpenGL API. This makes it possible to render the 3D scene into a framebuffer that is not displayed but saved into a 2D texture. For example, the effect is that the L-buffers computed during the first rendering pass will never be written to the screen framebuffer. Using a texture attachment, the result of that rendering pass is stored into a 2D texture. During the second rendering pass, a rectangle of the size of the detector is displayed making use of this texture to compute $\sum_i \mu(i)L_p(i)$ in Eq. 2. Similarly, during the final rendering pass, a rectangle of the size of the detector is displayed making use of this texture to compute the X-ray attenuation. Storing intermediate rendering passes in textures attached to FBOs is a compulsory stage.

Floating point precision is necessary in the L-buffer, and this can be obtained via off-screen rendering (floating point texture attached to a FBO).

Finally, displaying the results of the simulation is not always necessary. For example, one application is the optimization of experimental parameters in the imaging chain. For this purpose, it is necessary to simulate large series of images with no need to visualize every image. The X-ray attenuation image is therefore stored as a floating point texture attached to a FBO.

3.3 Computation of path length

To evaluate Eq. 4, a shader program is used to compute the L-buffer for every object ($L_p(i)$). The result is stored in $FBO\,(L_p(i))$. The X-ray source and detector parameters are taken into account using the OpenGL projection and modelview matrices: the projection matrix is set to match the X-ray detector's geometrical properties and the modelview matrix is set so that the camera position matches the X-ray source position (see Figure 2).

The naive approach to compute the path length (L_p) of the ray in objects consists of determining and sorting the intersection points. This can be handled using the well-known depth-peeling technique [3], that is used to render semi-transparent polygonal geometries without sorting polygons. However this is a multi-pass technique, which is a computational overhead. To efficiently perform path length computations, we use the algorithm presented by Freud *et al* for GPU programming. This method is more effective in our application as it only requires a single pass and no intersection ordering is needed. By convention in OpenGL, triangles of a mesh are described so that their respective normal vectors are outward. Consider the geometry setup described in Figure 3. This is a 2D representation of a scene made up of a disk in which a rectangular hole has been made. Let μ_d be the attenuation coefficient of the disk. In this case, the path length is given by:

$$L_p = (d_2 - d_1) + (d_4 - d_3) \tag{3}$$

where d_1 to d_4 are the distances from the X-ray source to the successive intersection points of the ray with the triangle mesh. We can observe in Figure 3 that the ray penetrates into the disk when the dot product between $\mathbf{viewVec}$ and $\mathbf{N_i}$, the normal of the triangle at the intersection point, is positive. Conversely, the ray leaves an object if the dot product between $\mathbf{viewVec}$ and $\mathbf{N_i}$ is negative. The path length of the ray in a given object can be written as follows:

$$L_p = \sum_i -sgn(\mathbf{viewVec} \cdot \mathbf{N_i})d_i \tag{4}$$

where i refers to the i^{th} intersection found in an arbitrary order, d_i is the distance from the X-ray source to the intersection point of the ray with the triangle, $sgn(\mathbf{viewVec} \cdot \mathbf{N_i})$ stands for the sign of the dot product between $\mathbf{viewVec}$ and $\mathbf{N_i}$. This dot product and d_i must be computed for each intersection point. These operations can be efficiently achieved on the GPU using a fragment program. During the rendering stage, hidden surface removal algorithms such as Z-buffer and back-face culling are disabled so that every triangle of the polygon mesh is taken into account. In the vertex program, we first compute the viewing vector ($\mathbf{viewVec}$). The position of the vertex being handled by the geometry processing unit is stored and will be used later in the fragment program to compute the distance of the intersection to the X-ray source. The normal vector of the

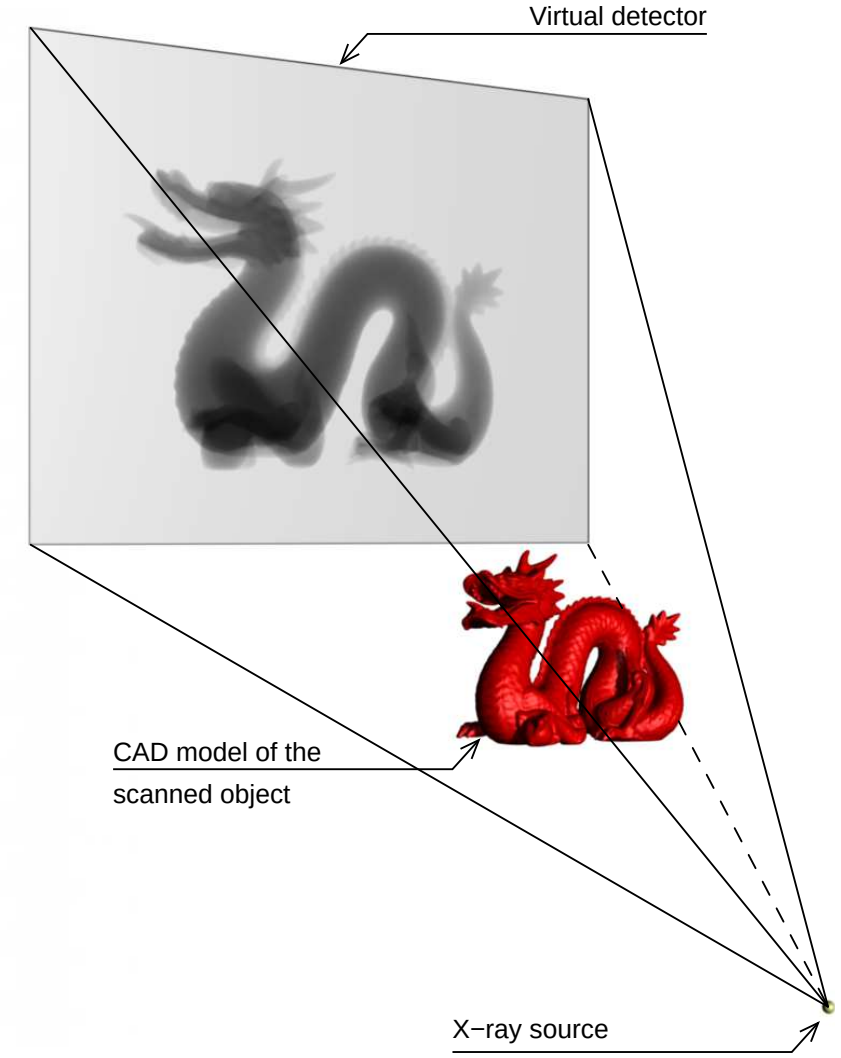


Figure 2: Radiographic simulation.

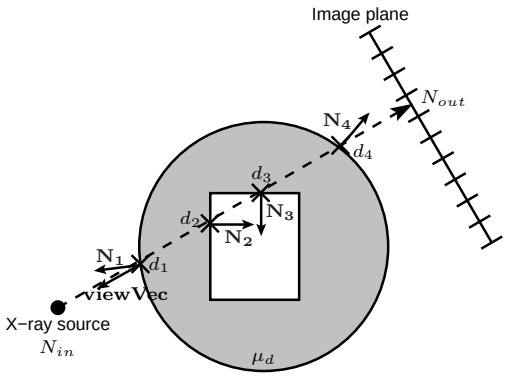
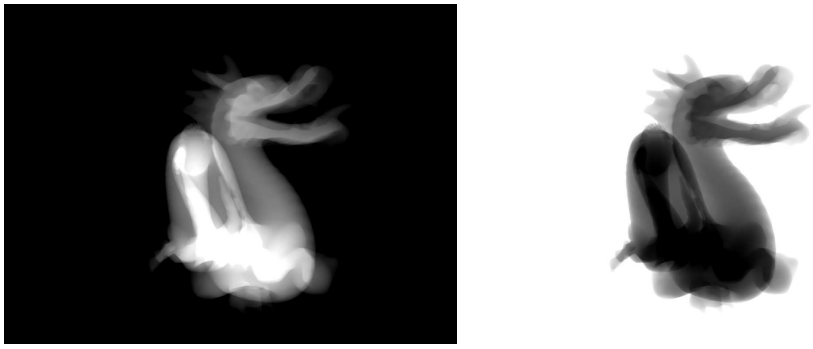


Figure 3: Principle of the computation of path length.

vertex is stored and will be automatically interpolated to be used later in the fragment shader. In the fragment shader, we evaluate the sign of the dot product between **viewVec** and the normal vector ($\mathbf{N_i}$). Note that the calculation of L_p using Eq. 4 raises robustness issues, notably when rays encounter triangle edges or vertices, or when the normal vector $\mathbf{N_i}$ is perpendicular to the viewing direction **viewVec**. These issues are addressed in [4] in the case of a CPU implementation, and we propose a method to address these on the GPU (see Section 3.5).

To evaluate Eq. 4, fragment values computed from overlapping intersections at a given pixel of the detector (i.e. intersection points found along the corresponding ray) must be added to each other into the framebuffer. In practice the current value that is computed by the fragment program needs to be combined with the value that is already in the framebuffer. This operation is known as *blending*. It is not possible to perform the blending operation within the fragment program alone because a fragment program does not give any access to the current value of the fragment in the framebuffer. Without blending, the new fragment will overwrite the value in the framebuffer. The only way to avoid this is to enable the OpenGL built-in blending function. Using the blending function *glBlendFunc(GL_ONE, GL_ONE)*, it is possible to update the value that is already in the framebuffer by adding the new value computed by the fragment program. Figure 4(a) shows the L-buffers corresponding to Figure 4(b).



(a) L-buffer. (b) Radiographic image.

Figure 4: Examples of 1024×768 images computed from a polygon mesh consisting of 202,520 triangles.

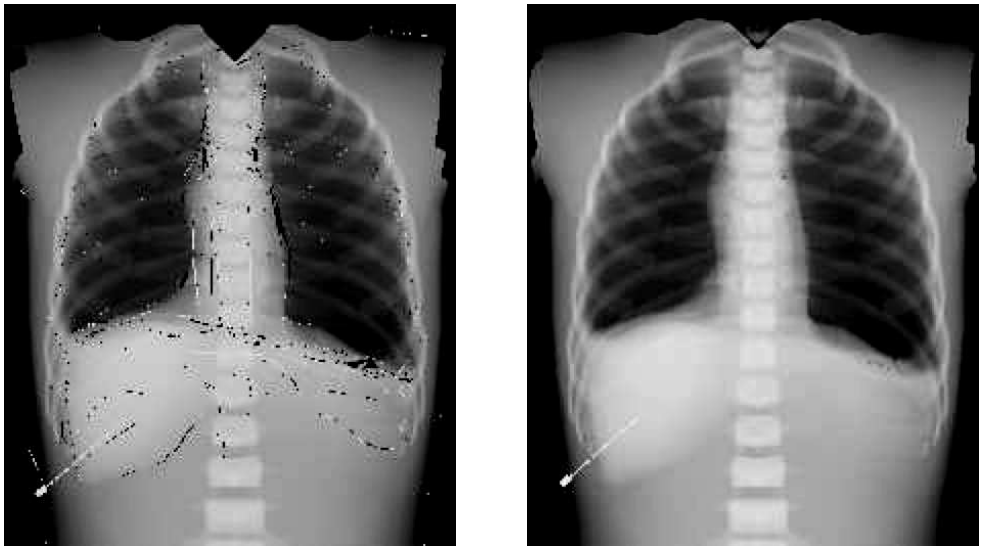
3.4 Computation of the X-ray attenuation

An intermediate stage is required to compute $\sum_i \mu(i)L_p(i)$ in Eq. 2. This second pass is stored into another FBO, called $FBO\left(\sum_i \mu(i)L_p(i)\right)$. A textured rectangle of the size of the X-ray detector is drawn using the texture that is attached to $FBO\left(L_p(i)\right)$. To compute $\sum_i \mu(i)L_p(i)$, *glBlendFunc(GL_CONSTANT_ALPHA, GL_ONE)* is used with *glBlendColor(1.0, 1.0, 1.0,* $\mu(i)$*)*.

Similarly, in the final stage, a textured rectangle of the size of the X-ray detector is rendered to compute the total attenuation (N_{out} in Eq. 2). This can be achieved by a fragment program that makes use of the texture attached to $FBO\left(\sum_i \mu(i)L_p(i)\right)$. Figure 4(b) shows the computed image from the L-buffer of Figure 4(a).

3.5 Correcting Artefacts

When intersections occur between a ray and an object, there should be the same number of incoming and outgoing intersections. However, some intersections may be duplicated when the ray hits triangle edges or vertices. Also, uncertainty occurs when the normal vector $\mathbf{N_i}$ is perpendicular to the viewing direction. In these cases, black or white pixel artefacts in the final image will appear depending on the orientation of the normal vector. Figure 5(a) shows such a X-ray image from a complex scene without artefact correction. It makes use of a human model made up of the ribs,



(a) Without filtering. (b) With adaptative filtering.

Figure 5: Effect of the artefact correction filtering.

spine, sternum, diaphragm, lungs, cartilage, liver and skin.

However, it is possible to detect for each pixel if such artefacts will occur and correct them using image processing. Indeed, Eq. 5 should always be null for every pixel:

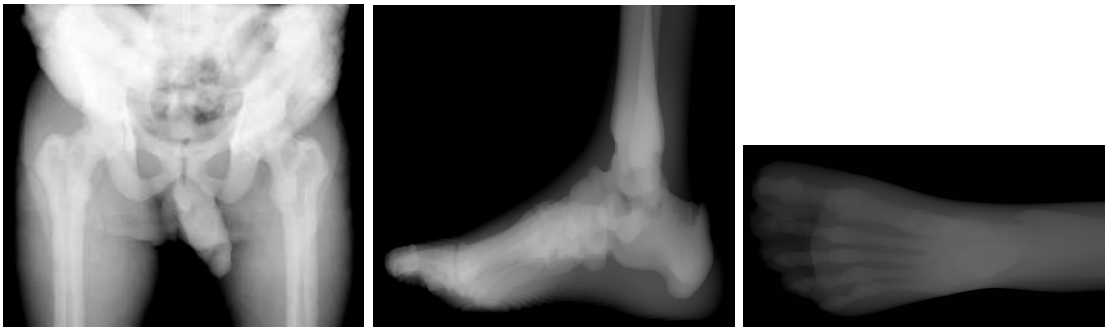
$$\sum_{i=1}^{n} sgn(\mathbf{viewVec}.\mathbf{N_i}) \tag{5}$$

with n the number of intersections between the ray and the processed triangle mesh. The fragment shader used to compute the L-buffer can be extended so that the sign of the dot product is stored into the green channel of the L-buffer texture. The sum operation in Eq. 5 is performed by taking advantage of the blending function used during the L-buffer computations. Before using any value of the L-buffer, we check the validity of the green component. If the green component is not null, then the L-buffer value is invalid. To avoid the artefact, it is replaced by the average value of the valid pixels within its direct neighbourhood. Figure 5(b) shows the X-ray image corresponding to Figure 5(a) when artefact correction is enabled.

4 Results and discussion

Radiographs usually represent the negative images of the attenuation, e.g. highly attenuating materials such as bones are in white and gas in black. Figure 6 presents such medical images. CT datasets have been segmented to extract polygon meshes. The hip model is composed of the bowels, fat, muscle and bones. Note that the hands are visible on the top of the image. The foot model is made of muscle and bones only. In [18], we show how to integrate our GPU implementation within an interactive training simulator for percutaneous transhepatic cholangiography procedures. It makes use of dynamic data that simulates the patient respiration.

To further assess the performance of our method, we first compare the computation time with a CPU implementation. Then, we compare computed images with a reference image simulated using the CPU implementation. The images have been computed on GPUs using full floating point precision (128 bits per pixel) or half floating point precision (64 bits per pixel). Three GPUs from NVIDIA have been selected: i) GeForce 8800 GTX, a high-end gaming graphics processor,



(a) Hip. (b) Left foot. (c) Left foot.

Figure 6: Simulated radiographs.

ii) GeForce 8600M GT, a graphics processor for laptops, and iii) Quadro FX 3500, a high-end professional graphics processor for workstations. The test results of the CPU implementation are based on an Intel Core 2 Duo E6600 (2.4 Ghz) and 2 GB of RAM with 64-bit Linux operating system.

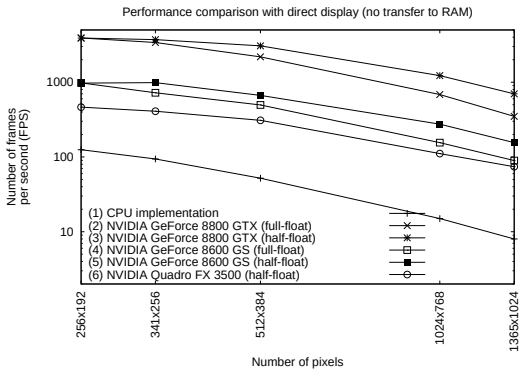
4.1 Computation time

The computational performance is given in number of generated frames per second (FPS). Figure 4(b) shows an example of the computed images. Using test objects with 11,102, 47,794, 202,520 and 871,414 triangles, the running times of the GPU and CPU implementations to generate a predefined animation of 1000 frames were recorded. We also simulated images of increasing resolutions. The average cover of the detector area by the test object is 21.5%.

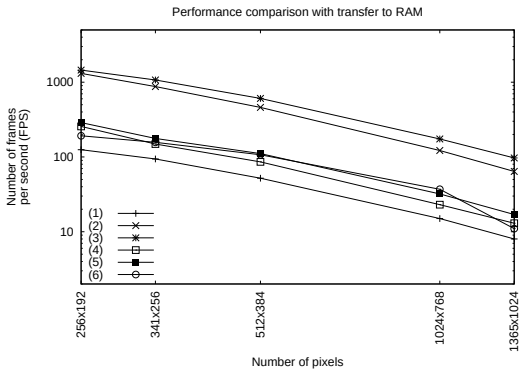
When assessing the performance, two different scenarios can be identified as regards the data transfer between GPU and CPU, which can be a bottleneck. In many cases, there is no need to transfer any data from the GPU to the CPU (see Figures 7(a) and 7(c)). For example, to simulate a radiograph taking into account the finite size of the X-ray tube focus (causing geometric unsharpness), many projections have to be carried out with a collection of source points representing the focal spot. These image contributions only have to be integrated to obtain the final image. The integration can be done in the same FBO using the blending function. If every simulated image has to be transferred to the RAM (see Figures 7(b) and 7(d)), the time required to transfer the data may become the limiting factor. A test case confirmed this assumption when small numbers of polygons are considered. For objects with a high number of triangles, the data transfer is a limited expense in the overall computation time.

It can be observed in Figure 7(a) that when the number of pixels becomes very high, the number of FPS tends to decrease linearly with a slope equal to -1 in the logarithmic graph. It corresponds to the fact that the fragment calculations become the prevailing component in the computation time, and the number of FPS is then inversely proportional to the number of pixels. The same type of behaviour is observed with respect to the number of triangles of the mesh (Figure 7(c)). When the number of triangles increases, the number of FPS also tends to decrease linearly with a slope of -1, meaning that the vertex calculations prevail in the computation time. In the case of objects with 871,414 triangles, the GPU implementation using full floating point precision is up to 61 times faster than the CPU implementation. With the least powerful GPU, the performance obtained using the highest resolution triangle mesh still enables interactive frame rates.

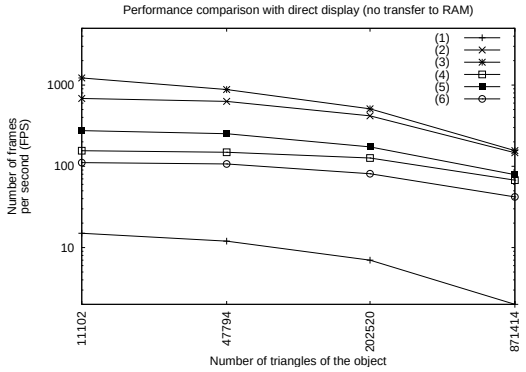
When every frame is transferred from the GPU to the RAM, the number of FPS tends to decrease linearly with a slope equal to -1 in the logarithmic graph and the number of FPS is then inversely proportional to the number of pixels (see Figure 7(b)). The number of FPS tends to be



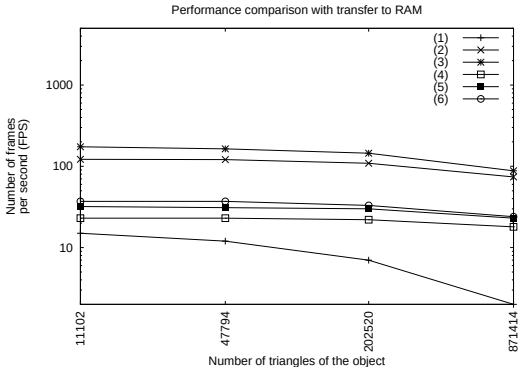
(a) Number of radiographs computed in one second, from a polygon mesh consisting of 11,102 triangles, with respect to the image resolution, with direct display (no transfer to RAM).



(b) Idem Fig. 7(a) but with transfer of each frame from GPU to RAM.



(c) Number of radiographs (1024×768 pixels) computed in one second, with respect to the polygon mesh resolution, with direct display (no transfer to RAM).



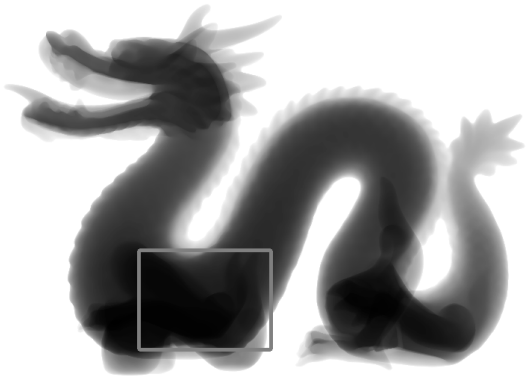
(d) Idem Fig. 7(c) but with transfer of each frame from GPU to RAM.

Figure 7: Number of radiographs of the whole object, with 21.5% detector coverage, computed in one second.

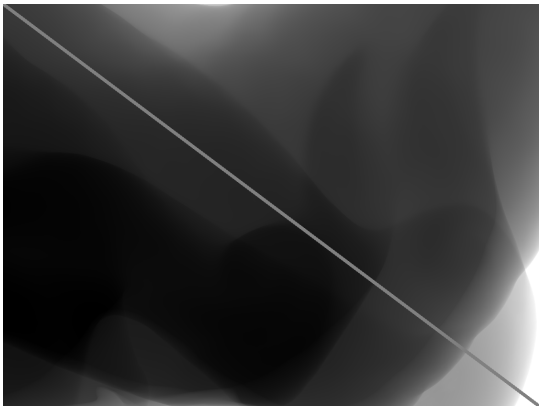
constant when the number of triangles increases, unless the number of triangles is very high (see Figure 7(d)). It corresponds to the fact that i) the time required to transfer the data, which is constant at a given pixel resolution, becomes the limiting factor when small numbers of polygons are considered, and ii) for objects with a high number of triangles, the data transfer becomes negligible. In the case of an image with 1024×768 pixels, transferring every frame to the RAM, the performance is up to 9 times slower for objects with 11,102 triangles and 2 times slower for objects with 871,414 triangles.

4.2 Accuracy

To validate the accuracy of our GPU implementation, we simulate an image with the same physical parameters on every platform and we compare intensities pixel by pixel with a reference image computed with the CPU implementation (see Figure 8). The gray square in Figure 8(a) shows the



(a) The gray rectangle indicates the reference image used for accuracy tests.



(b) Reference image used for accuracy comparison (1024×768 pixels). See Figure 9 for profiles corresponding to the gray line.

Figure 8: Test image used in accuracy comparison.

region of the scanned object that has been chosen for the accuracy comparison. Comparing the results with the full image would underestimate the average error because of the high proportion of rays which do not intersect the object. In the chosen region, 99.42% of the rays are attenuated by the object. Figure 9 shows clos-up diagonal profiles of the images computed with the GPUs and with the CPU. It illustrates that simulations performed on GPUs are relatively close to the reference simulation. Profiles extracted from the images computed with full floating point precision accurately match the profile from the reference image. This contrasts with computations performed using half floating point precision.

To quantify inacuracy, disparity measurements using the pixelwise relative error were computed for each test image computed on GPUs with respect to the reference image (see Table 1). The error metrics is computed pixelwise as follows:

$$\delta(i,j) = \frac{|A(i,j) - B(i,j)|}{B(i,j)} \tag{6}$$

with **A** the image computed on GPUs and **B** the reference image. These results confirm our hypothesis that a fast and accurate GPU implementation of X-ray simulation can be implemented with full floating precision. Using half floating point precision, the accuracy of computations is somewhat reduced but the relative error stays below 1.2%.

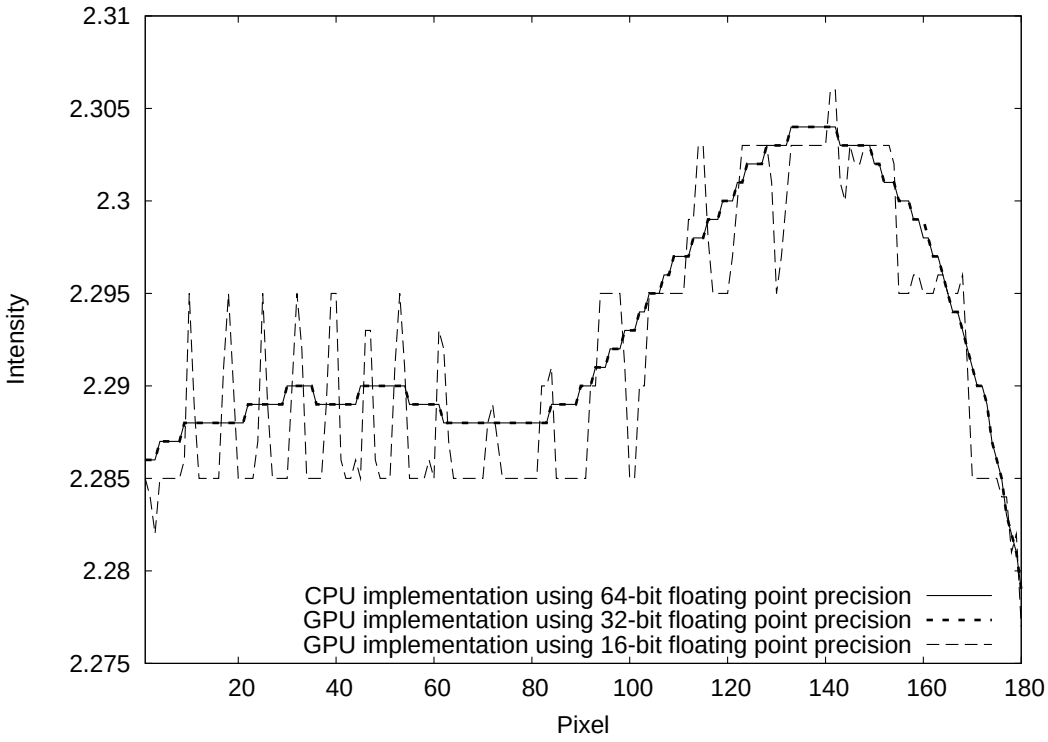


Figure 9: Close-up of profiles diagonal profiles corresponding to Figure 8(b).

Table 1: Disparity measurements.

Precision	GPU	Maximum error	Average error
full float	GeForce 8800 GTX	$2.55e^{-3}$	$2.19e^{-6}$
half float	GeForce 8800 GTX	$1.20e^{-2}$	$1.36e^{-3}$
full float	GeForce 8600M GT	$2.55e^{-3}$	$2.22e^{-6}$
half float	GeForce 8600M GT	$1.20e^{-2}$	$1.36e^{-3}$
half float	Quadro FX 3500	$1.12e^{-2}$	$1.36e^{-3}$

5 Conclusion

The simulation of X-ray transmission imaging using common CPU-based approaches is highly time consuming. The use of the GPU allows the simulation to be accelerated considerably. Our implementation has proved to be both fast and accurate.

Acknowledgements

This work has been partially funded by the UK Department of Health under the Health Technology Devices programme and commissioned by the National Institute for Health Research (NIHR). This is independent research and the views expressed are those of the authors and not necessarily those of the NHS, the NIHR or the Department of Health.

References

[1] A. Bonin, B. Chalmond, and B. Lavayssière. Monte-Carlo simulation of industrial radiography images and experimental designs. *NDT & E International*, 35(8):503–510, 2002.

[2] P. Duvauchelle, N. Freud, V. Kaftandjian, and D. Babot. A computer code to simulate x-ray imaging techniques. *Nuclear Instruments and Methods in Physics Research B*, 170(1-2):245–258, 2000.

[3] C. Everitt. Interactive order-independent transparency. White paper, NVIDIA OpenGL Applications Engineering, 2001. Available at `http://developer.nvidia.com/object/Interactive_Order_Transparency.html` (accessed 27$^{\text{th}}$ March 2008).

[4] N. Freud, P. Duvauchelle, J. M. Létang, and D. Babot. Fast and robust ray casting algorithms for virtual X-ray imaging. *Nuclear Instruments and Methods in Physics Research B*, 248(1):175–180, 2006.

[5] N. Freud, J.-M. Létang, and D. Babot. A hybrid approach to simulate X-ray imaging techniques, combining Monte Carlo and deterministic algorithms. *IEEE Transactions on Nuclear Science*, 52(5):1329–1334, 2005.

[6] N. Freud, J. M. Létang, C. Mary, C. Boudou, C. Ferrero, H. Elleaume, A. Bravin, F. Estève, and D. Babot. Fast dose calculation for stereotactic synchrotron radiotherapy. In *Proceedings of the 29th IEEE EMBS*, pages 3914–3917, 2007.

[7] F. Inanc, J. N. Gray, T. Jensen, and J. Xu. Human body radiography simulations: development of a virtual radiography environment. In *Physics of Medical Imaging*, volume 3336, pages 830–837, 1998.

[8] D. Laney, S. P. Callahan, N. Max, C. T. Silva, S. Langer, and R. Frank. Hardware-accelerated simulated radiography. In *IEEE Visualization 2005 (VIS' 05)*, pages 343–350, 2005.

[9] D. Lazos, Z. Kolitsi, and N. Pallikarakis. A software data generator for radiographic imaging investigations. *IEEE Transactions on Information Technology in Biomedicine*, 4(1):76–79, 2000.

[10] J.-M. Létang, N. Freud, and G. Peix. Signal-to-noise ratio criterion for the optimization of dual-energy acquisition using virtual X-ray imaging: application to glass wool. *Journal of Electronic Imaging*, 13(3):436–449, 2004.

[11] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[12] N. Li, S.-H. Kim, J.-H. Suh, S.-H. Cho, J.-G. Choi, and M.-H. Kim. Virtual x-ray imaging techniques in an immersive casting simulation environment. *Nuclear Instruments and Methods in Physics Research B*, 262:143Ű–152, 2007.

[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[14] A. S. Pasciak and J. R. Ford. A new high speed solution for the evaluation of monte carlo radiation transport computations. *IEEE Transactions on Nuclear Science*, 53(2):491–499, 2006.

[15] R. J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, 2$^{\text{nd}}$ edition, 2006.

[16] J. Spoerk, H. Bergmann, F. Wanschitz, S. Dong, and W. Birkfellner. Fast DRR splat rendering using common consumer graphics hardware. *Medical Physics*, 34(11):4302–4308, 2007.

[17] F. P. Vidal, N. W. John, and R. M. Guillemot. Interactive physically-based x-ray simulation: CPU or GPU? In *Medicine Meets Virtual Reality 15*, pages 479–481, 2007.

[18] P. Villard, F. P. Vidal, C. Hunt, F. Bello, N. W. John, S. Johnson, and D. A. Gould. Simulation of percutaneous transhepatic cholangiography training simulator with real-time breathing motion. In *Proceeding of the 23rd International Congress of CARS - Computer Assisted Radiology and Surgery*, 2009.

[19] L. Westover. Interactive volume rendering. In *Proceedings of the 1989 Chapel Hill workshop on Volume visualization*, pages 9–16, 1989.

[20] H. Yan, L. Ren, D. J. Godfrey, and F. F. Yin. Accelerating reconstruction of reference digital tomosynthesis using graphics hardware. *Medical Physics*, 34(10):3768–3776, 2007.