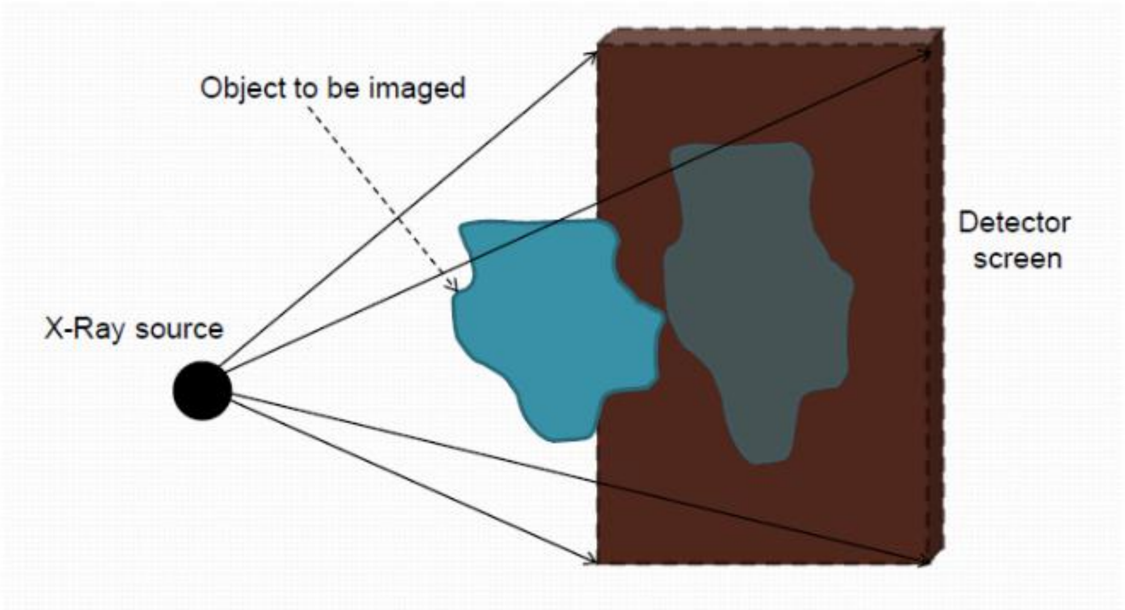
New algorithms to simulate X-ray radiography

Koushik Viswanathan

Fundamental algorithm



$$N(E) = N_0(E)\Delta\Omega\prod_i \exp[-\mu_i(E)x_i]$$

$$= N_0(E)\Delta\Omega\exp\left[\sum_i -\mu_i(E)x_i\right].$$

Fundamental algorithm

- Ray tracing – more formally termed Ray casting, as we follow only primary rays, from the source to the detector [1][2]
- As is well known, spawn one ray per detector pixel and follow its path through space
- Use of a triangular mesh – Employ ray-triangle intersections tests along path of the ray [3]
- Sort intersection points by distance from source and compute path lengths in-between them

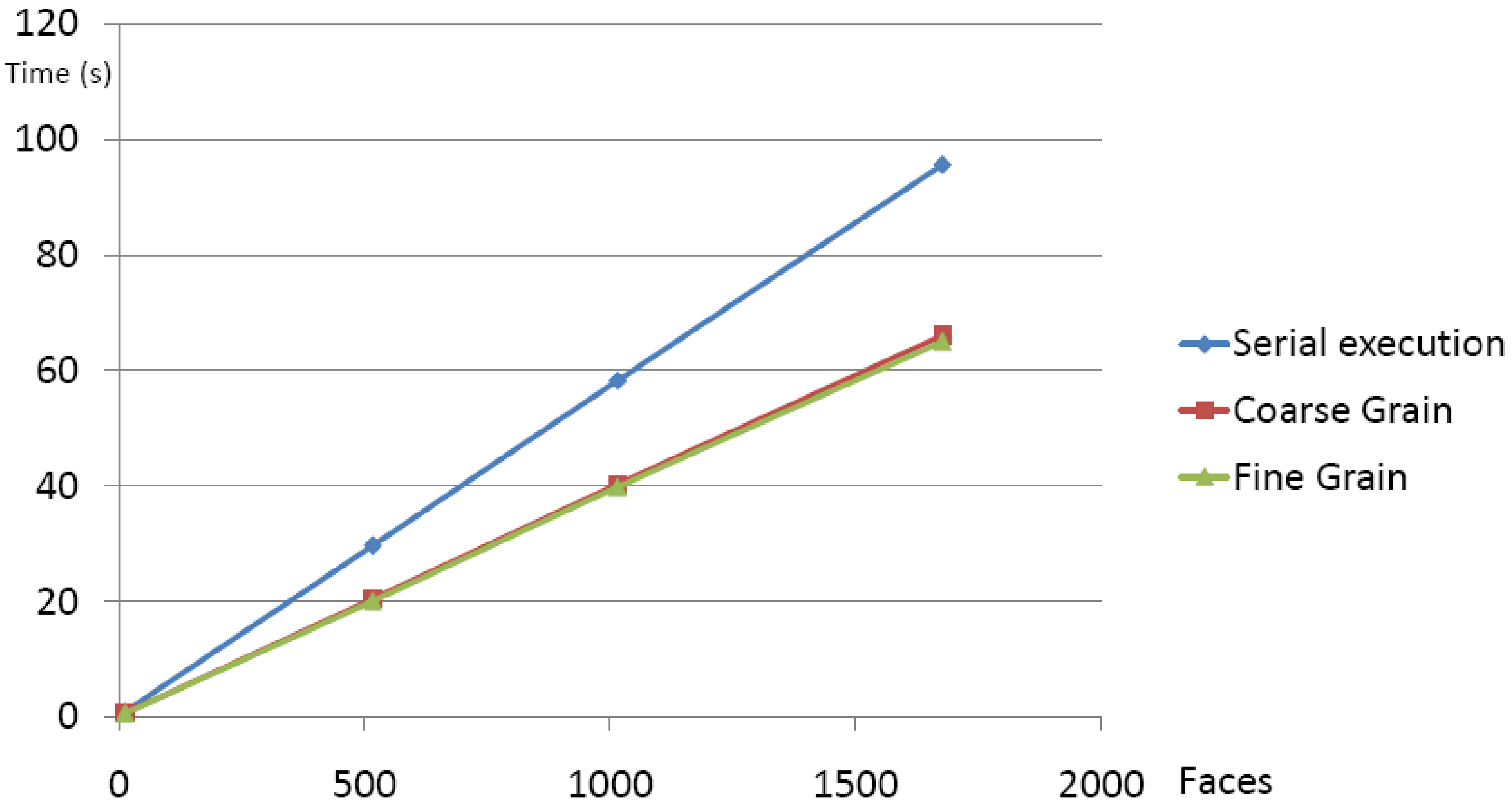
Fundamental algorithm

- A brief recap of the algorithm:
- For each pixel of the detector
  - Start a ray, going backwards towards the source, set rayValue(i,j) = 0
  - Perform bounding box intersection query, if successful,
    - Perform ray-triangle intersection query, with the sorted triangle list obtained from the CAD model
    - Sort all collision points by distance from source
    - Since there are even no. of collisions (closed mesh) compute distances between successive points, add all of them and set rayValue(i,j) = distance * attenuation
  - Set pixel value = rayValue(i,j)

Optimizations to the technique

- As is known, ray-tracing is inherently parallelizable, so we employ multi-threading to reduce total computation time
- Tests performed with CPU and GPU multi-threading
  - Upto 10x speedup from the initial serial processing
  - GPU acceleration improved on CPU multi-threading, implemented with CUDA

Optimizations to the technique: CPU Multi-threading



Optimizations to the technique: GPU Multi-threading

- GPU: nVidia GeForce 8600GT
  - Clock speed of 1.19 GHz
  - 4 multi-processors, 32 cores
- Same test case:
  - 512x512 pixel detector
  - Object: 571 vertices and 517 triangles
  - One CUDA thread per ray traced
- Scan time: Average of 3.39 seconds! (nearly 10x reduction)

Fundamental limitations

- The Ray Tracing technique is a per-pixel operation and scales linearly with simulated detector size
- With an increase in polygon count, number of intersection tests increases drastically
- Use of octrees/ KD-Trees can offset this limitation
  - Spatial data structures work very well in graphics rendering
  - CAD models being extremely dense, spatial subdivision techniques will not work out as well for triangular meshes
  - They would work better in case of voxelized analysis and simulations [7]

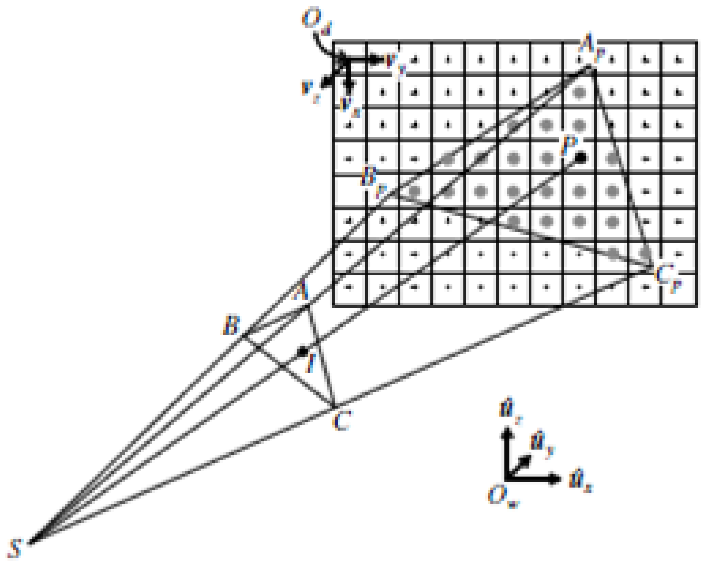
An alternative approach

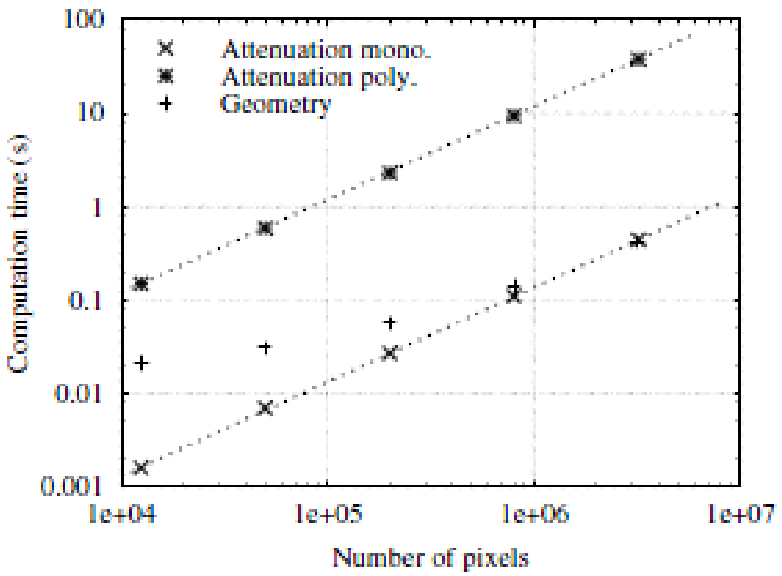
- An alternative technique exists, heavily inspired by traditional Z-buffer based Rasterization methods used widely in computer graphics [4]
- Advantage – each face is tested exactly once. The algorithm is expected to scale near-linearly with number of faces
- As number of faces goes approx. beyond the order of 10,000, scalability becomes important

Projection technique

- What Freud et.al. do in [4] is project the face onto the detector, use the face-plane equation and determine the intersection point of the ray spawned from pixels inside the face's projection
- Determination of pixels within the face's projection are done using traditional polygon-filling techniques, used in rasterization [5]

Reported results





Test case:

- CAD model with 13328 faces
- Image size = 200 x 1000 pixels
- Simulation time (Geometric) = ~0.1 – 1s

Application of the algorithm

- Widely used algorithm, used in various similar imaging simulation applications
  - Casting applications – Use of a triangular mesh for simulating radiography [6]
  - Same group – Use of voxels and a ray-box intersection scheme [7], inspired by the same algorithm [4]
- Simulation times with triangular meshes are not frequently reported

Original projection algorithm

Break up of the actual algorithm as described in [4]:

1. Project all object vertices onto the detector plane
2. Scan each facet's projection to identify pixels whose center is located inside the facet's projection
3. For each of the previous pixels, calculate the position of the intersection point on the facet
4. For each ray (or pixel), determine the attenuation path length L in the object and store it in an 'L-buffer'

Original projection algorithm

- In the algorithm described by [4], there are two major steps – the pixel information determination (step 2) and the intersection point computation (step 3)
- Traditional polygon-filling algorithms work by determining the intersection of scan lines (of the raster) with the edges of the polygon
- Once interior points are determined, algorithm [4] then reverse calculates the intersection point of the ray with the facet

Enhancements?

- We could, in theory, combine steps 2 and 3 into one, in order to reduce computation time
- The idea is to replace step 2 with an alternative technique
  - Based on the barycentric coordinate system for a triangle
  - Uniqueness of barycentric coordinates of a point irrespective of projection

Barycentric coordinates

- 3-Tuple describing any interior point in a triangle in terms of distances from the 3 vertices
  - Linearly dependant coordinates, only 2 independent coordinates actually needed
  - Represented by (u,v,1-u-v)
- Key property: Barycentric coordinates of an interior point of a triangle remain the same irrespective of which plane it is projected on

Proposed algorithm

The proposed modified projection algorithm is as follows

1. Project each of the triangular facets onto the detector, after the required transformation
2. Compute the Minimum Bounding Rectangle (MBR) for the projected triangle
3. For each pixel inside the MBR, compute the barycentric coordinates (u,v), reject if $u,v<0$ or $u+v>1$
4. Using the barycentric coordinates, interpolate the depth d of the face-pixel from the source position, and store it in a buffer L, biased by the relative direction between the face normal and the line joining source to current pixel
5. Once all triangles are checked, buffer L yields final result

Proposed algorithm

Potential problem with the proposed algorithm is that a lot more computations are needed to determine the barycentric coordinates

- Exploit the large coherence between successive pixels inside the MBR!
- Pre-computing edges of each of the faces, computation complexity per face is O(n) where n is size of the MBR in pixels
- As n is usually very low compared to the detector size for a well-tessellated model, computation time per face is relatively low

Proposed algorithm

- Largest gains are seen in models with a huge number of triangles as the results shown later indicate
- For highly detailed models, area occupied and hence size of MBR are very small, so relative computation difference per face is very low – sometimes the net computation time is lesser!
- Models that previously couldn't be simulated with standard ray-tracing, can now be easily handled

Reported simulation results

- Very few papers actually report scan times as well as test model complexity quantitatively
- Freud et.al. [4] report scan times of ~0.5 sec for a model of 13328 triangles
- Bellon et.al. [8] reported ~35 sec for a 2048x2048 pixel detector, for a model containing over 100,000 triangles
- Reiter el.al. [9] have made a comparison between two implementations – one on a multi-core CPU and another on a GPU. They've reported simulation time of ~1.1 seconds for a 200,000 triangle model with a 2048x2048 pixel detector, using a GPU and approx. 9.7 seconds using a multi-core CPU

So how does it measure up?

The proposed algorithm was tested with a model consisting of over 800,000 triangles, and with a 2048x2048 pixel detector, took approx 20 seconds on an Intel Pentium 4, 3.0GHz Processor

- This is more than 4 times as many triangles as the test model in [9] and more than 8 times as many triangles as the model used in [8]
- Computation times are highly dependant on the number of pixels being affected, so a direct comparison is not easily possible

Some test results

A few popular CAD models were run through the algorithm. The results for a geometric simulation, with a 512x512 pixel detector are summarized in the table below

Model	Faces	Time1 (s)	Time2 (s)	Time3 (s)	Time4 (s)	Time5 (s)	Average (s)
Bunny	53,582	0.396	0.392	0.393	0.391	0.392	0.3928
Horse	96,967	0.375	0.374	0.376	0.372	0.373	0.374
Dragon	871,414	1.27	1.287	1.287	1.272	1.27	1.2772

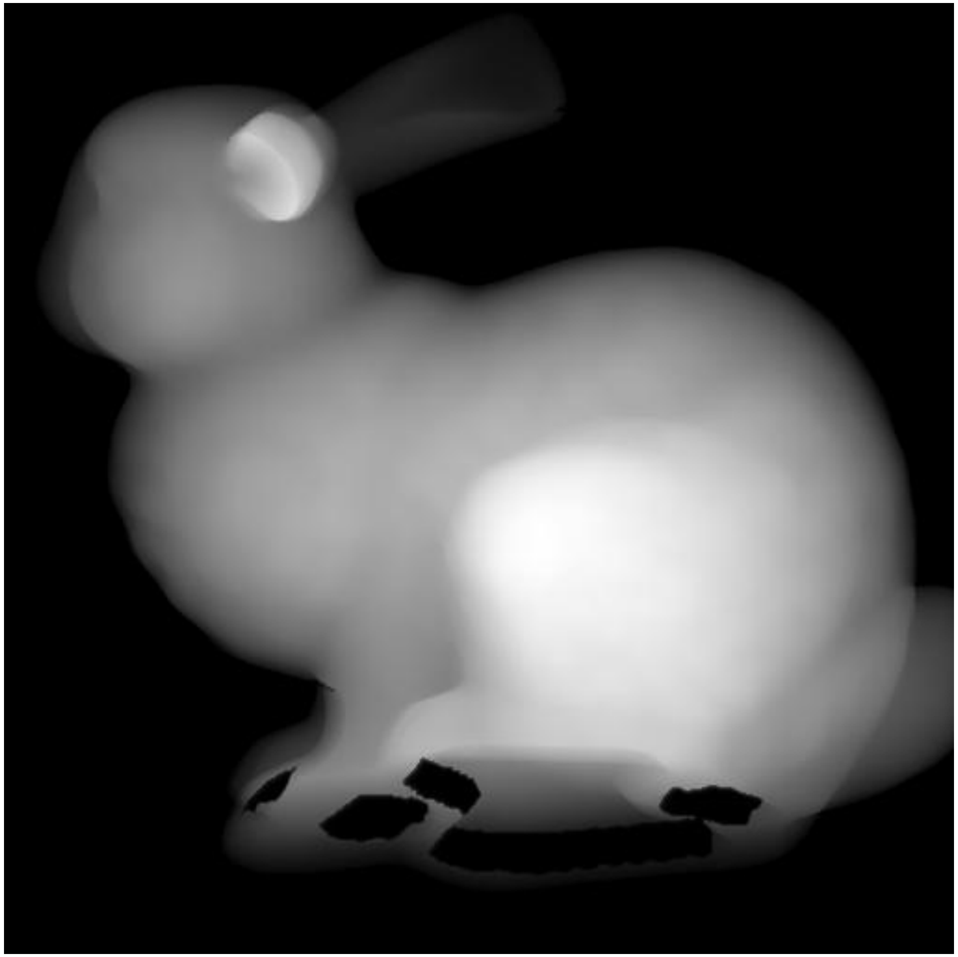
Some test results

- The models used for testing were taken from the Stanford 3D Scanning repository, maintained by the Stanford Computer Graphics Laboratory
- Simulation times reported on the previous slide were obtained on a modest Pentium IV, 3.0 GHz PC with 512 MB of RAM, running Windows XP Professional

Sample scans

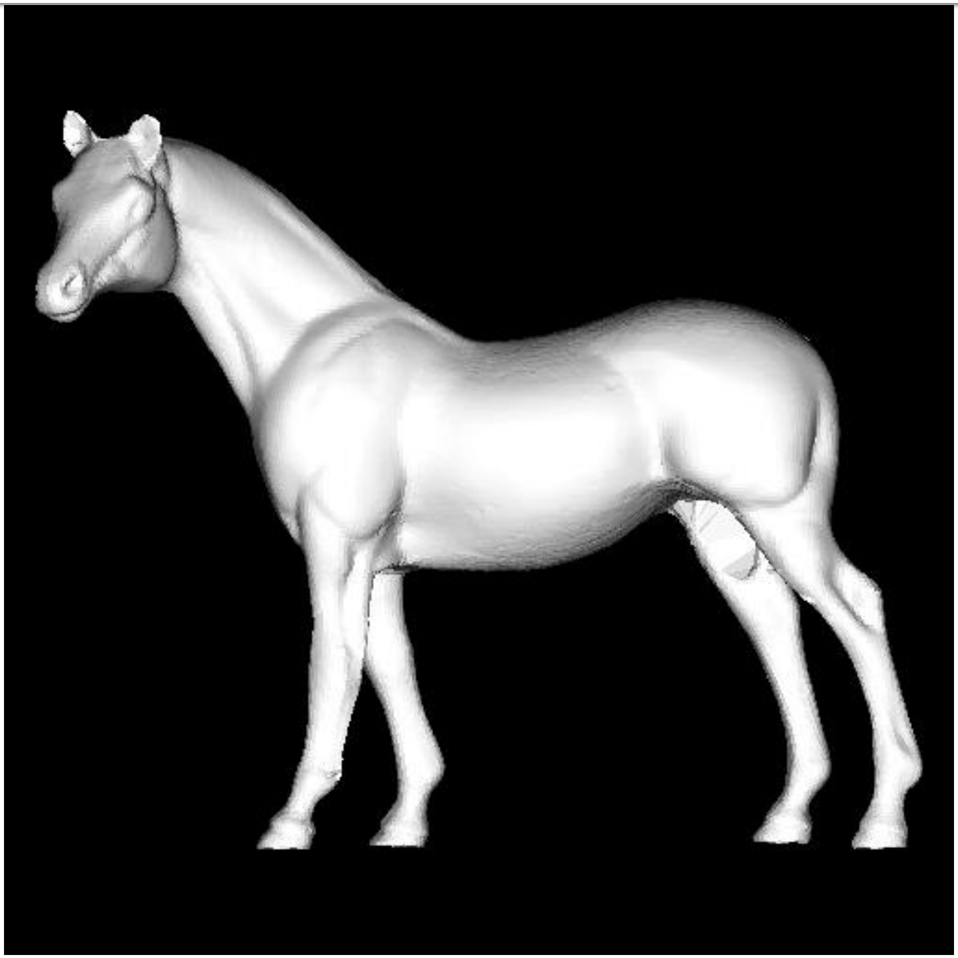
The Stanford Bunny
35,947 vertices
53, 582 faces

Sample scans



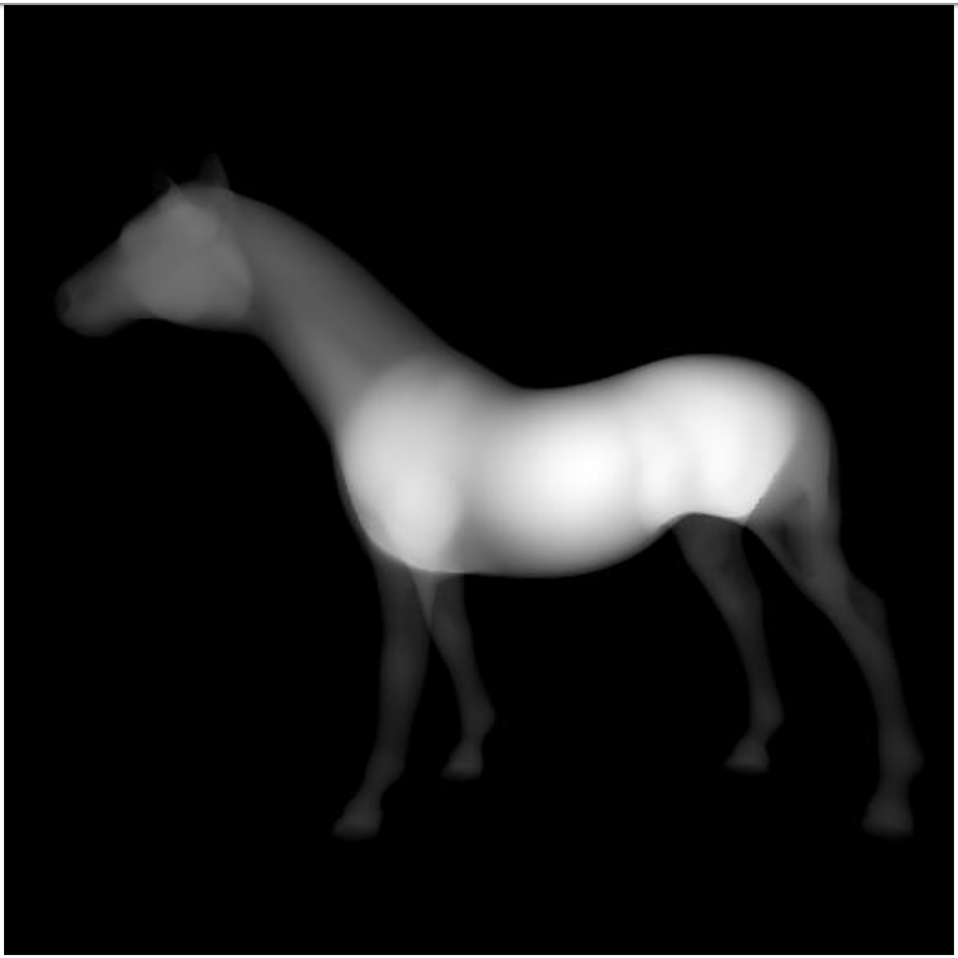
Simulated radiographic projection
512x512 pixel detector
NOTE: The holes at the bottom are present in the original model, below the feet. Since this is a cone-beam projection, they are projected onto the detector

Sample scans



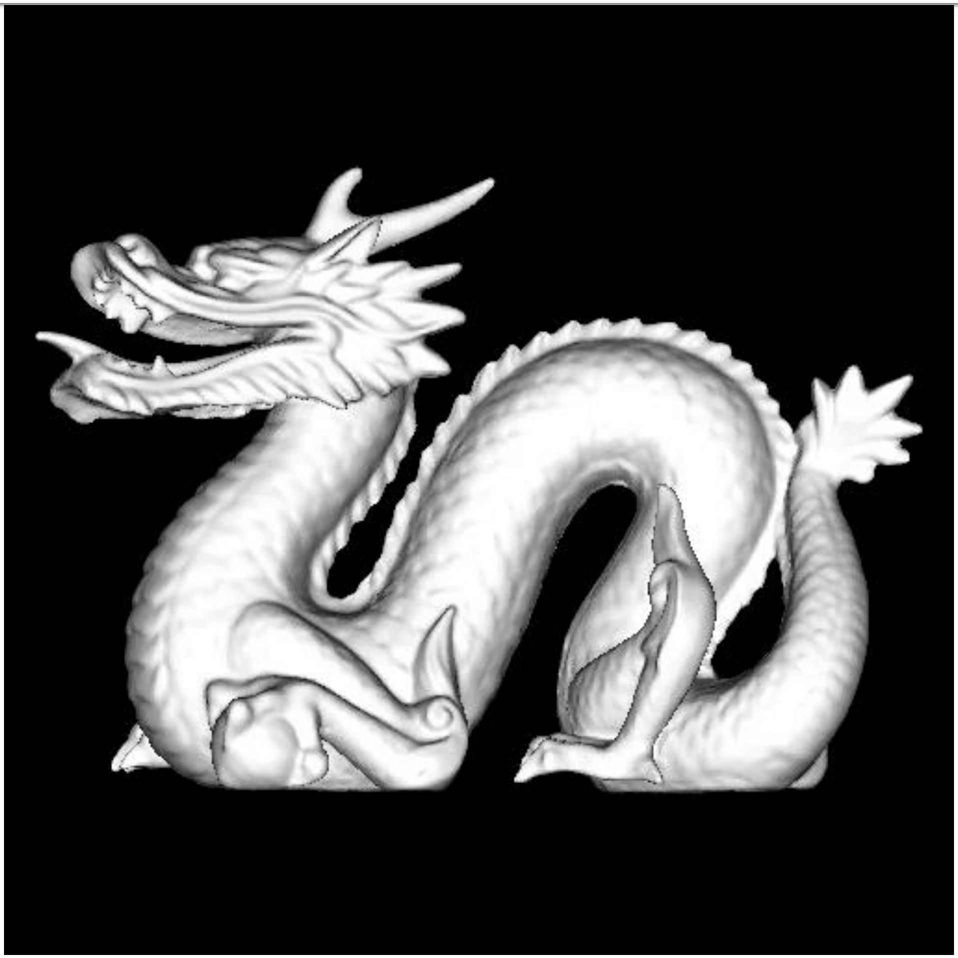
Horse model
Courtesy Cyberware, Inc.
48,485 vertices
96,967 faces

Sample scans



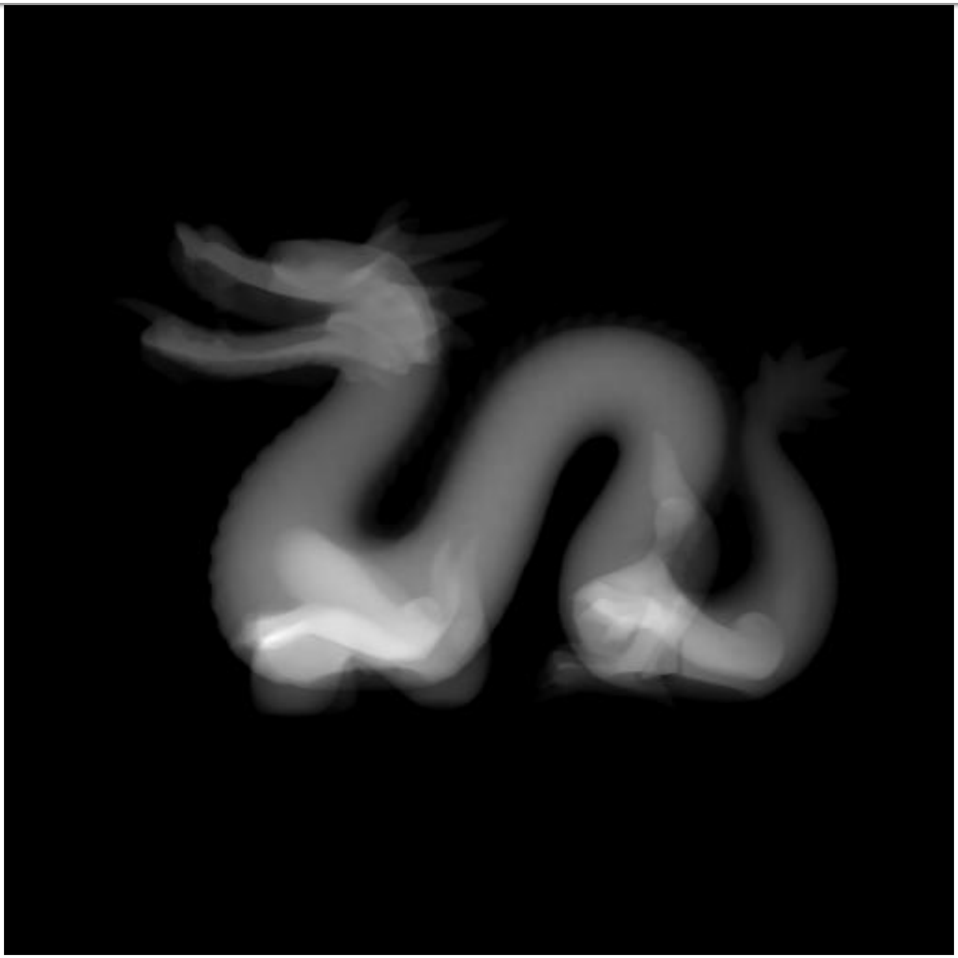
Simulated radiographic projection
Using a 512x512 pixel detector

Sample scans



Chinese Dragon
Source: Stanford Computer Graphics Laboratory
566,098 vertices
871,414 faces

Sample scans



Simulated radiographic projection
Again using a 512x512 pixel detector

References

1. Nicolas Freud, Philippe Duvauchelle, Daniel Babot, “Simulation of X-Ray NDT Imaging Techniques”, 15th WCNDT, Roma, 2000
2. Philippe Duvauchelle, Nicolas Freud, Valerie Kaftandjian, Daniel Babot, “A computer code to simulate X-ray imaging techniques”, Nuclear Instruments and Methods in Physics Research B 170 (2000) 245-58
3. T.Möller, B.Trumbore, "Fast, Minimum Storage Ray-Triangle Intersection", Journal of Graphics Tools, vol. 2, 21-28, 1997
4. N. Freud, P. Duvauchelle, J.M. Le´tang, D. Babot, “Fast and robust ray casting algorithms for virtual X-ray imaging”, Nuclear Instruments and Methods in Physics Research B 248 (2006) 175–180
5. J.D. Foley, A. van Dam, S.K. Feiner, J.F. Hughes, Computer Graphics: Principles and Practice in C, second ed., Addison-Wesley, Boston, 1997, p. 1175
6. Ning Li, Sung-Hee Kim, Ji-Hyun Suh, Sang-Hyun Cho, Jung-Gil Choi, Myoung-Hee Kim, “Virtual X-ray imaging techniques in an immersive casting simulation environment”, Nuclear Instruments and Methods in Physics Research B 262 (2007) 143–152
7. Ning Li, Hua-Xia Zhao, Sang-Hyun Cho, Jung-Gil Choi, Myoung-Hee Kim, “A fast algorithm for voxel-based deterministic simulation of X-ray imaging “- Computer Physics Communications 178 (2008) 518–523
8. Carsten Bellon, Gerd-Rüdiger Jaenisch, “aRTist – Analytical RT Inspection Simulation Tool”, International Symposium on Digital industrial Radiology and Computed Tomography, June 25-27, 2007
9. M.Reiter, M.M.Malik, C.Heinzl, D.Salaberger, E.Gröller, H.Lettenbauer, J.Kastner, “Improvement of X-Ray image acquisition using a GPU based 3DCT simulation tool “, International Conference on Quality Control by Artificial Vision, May 2009.

Realtime scan simulation

DEMO TIME!