Design and Implementation of a Rendering Algorithm in a SIMD Reconfigurable Architecture (MorphoSys)

Javier Davila, Alfonso de Torres, Jose Manuel Sanchez, Marcos Sanchez-Elez, Nader Bagherzadeh*, F. Rivera
Dpt. Arquitectura de Computadores y Automatica, Universidad Complutense de Madrid (SPAIN).
*Dpt. Electrical Engineering and Computer Science, University of California Irvine (USA).

Abstract

In this paper we analyze a 3D image rendering algorithm and the different mapping schemes to implement it in a SIMD reconfigurable architecture. 3D image render is highly computational and has an important restriction in execution time due to the requirement to get interactive results. We demonstrate that the execution of this algorithm in MorphoSys can take advantage of the available parallel resources, as well as of the possibility of one cycle configuration change. In this paper we show that it is possible to implement the rendering algorithm in our coarse grain reconfigurable architecture, obtaining values over 100 fps.

1. Introduction

Reconfigurable architectures have had an important growth in the last years as academic as commercial level. The reconfigurable systems consist of a programmable hardware controlled by different control and configuration points, which decide the hardware functionality required in each moment to execute different applications [1].

The coarse grain reconfigurable architectures consist of a matrix with some functional units, which can work to word level instead of bit level, like FPGAs (fine grain) [16][3]. This granularity reduces the architecture area, consumed power, delays and configuration times compared with FPGAs.

For more than one decade ago, different coarse grain reconfigurable architectures have appeared. Examples of those architectures are: Morphosys [4] targe architecture of this work; RAPID [5] which uses linear arrays and it can be programmed through a high level language, like C. PACT XXP [6] including one AMBA bus as communication channel. REMARC [7] which consists of one MIPS-II RISC processor and an 8x8 array, made with 16 bits nano-processors, each one of them with a little memory. All the nano-processors are connected to a global control unit. CHAMALEON architecture [8] is made up of a reconfigurable unit, a programmable processor and in/out device. ADRES [2] is a coarse grain reconfigurable architecture developed at IMEC. All these architectures have common characteristics as one cycle context changing, array of reconfigurable cells with a rich interconnectivity, on-chip data memory … All these features have been optimized to execute multimedia and signal processing applications.

In the last years, many multimedia interactive applications have been implemented in different platforms like cell phones or PDAs. Among the applications that are having bigger growth, we can find 3D image processing applications. They are present in games, museums or virtual shop. They process a large amount of data, and they have a strong computational work. Moreover, these applications have an important inherent parallelism that makes them perfect candidates for being executed in reconfigurable architectures. Particularly, renderization algorithm [9], it is one of the most used in many applications to represent in the screen (2D) the 3D target scene. This algorithm is usually implemented in PC graphic cards [10] obtaining excellent results in execution time, but these cards are not frequently used to execute other kind of applications. However, the reconfigurable architectures have demonstrate that can executed many of the multimedia and DSP applications obtaining a competitive performance [16], as well as some graphic applications [7]. Therefore, the coarse grain reconfigurable architectures are the perfect candidates to executed one of the most used algorithm in graphics, renderization, as this paper demonstrate.

The paper is organized as follow. Section 2 describes Morphosys architecture where we implement the rendering algorithm, explained in Section 3. Section 4 exposes the different strategies, which have been studied, to implement the algorithm in Morphosys. Section 5 optimizes these strategies solving the z-buffer problem. Section 6 show experimental results obtained and Section 7 concludes the paper.

2. Morphosys

MorphoSys [4] is a coarse grain reconfigurable architecture developing in the University of California, Irvine. Current demand on speed of multimedia

3-9810801-0-6/DATE06 © 2006 EDAA

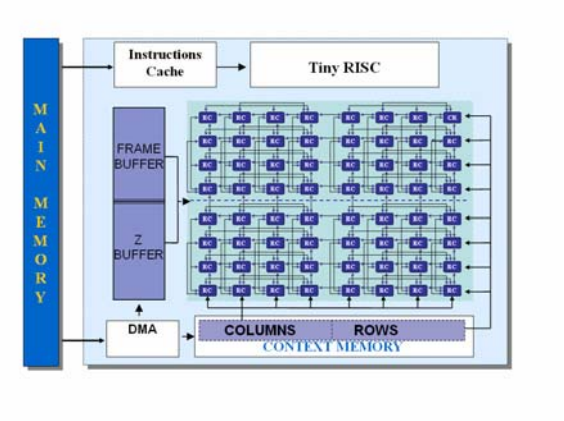

Figure 1. MorphoSys Architecture

applications execution makes MorphoSys reconfigurable architecture an interesting alternative. Its architectural scheme is shown in Figure 1.

The RC Array is the most important part of the reconfigurable module. It is composed by 64 reconfigurable cells, placed in 8 rows by 8 columns array. The RCs (Reconfigurable Cells) have a set of interconnections for communicating. The RC Array is divided into four 4x4 quadrants. Each cell is connected with four nearest neighbors in each quadrant and also with all the cells of its row and its column. There are interquadrant connections too.

Each reconfigurable cell seems a data-path of a microprocessor. It has an ALU-multiplier, a shift unit of 32 bits, multiplexers that select ALU input, a register bank and a context register.

The Context Memory (CM) stores the different configurations (contexts) that the architecture needs for executing the application. Then the cell in the same row or column share at one time the same context, it is a SIMD system but letting execution of different configurations in each row or columns at one time. This method lets to reduce the size of the context memory because it has to store one configuration for all the cells in the same column or row. It also reduces the time of loading contexts and the complexity of the interconnection network. Configuration load penalty is minimum because the contexts are stored in the CM, which is an internal memory. The context is loaded into the RC internal context registers in one cycle, which allows dynamic configuration. Furthermore, it is possible loading contexts into context memory at the same time that one configuration is executed in the RC Array.

The Frame Buffer (FB) is the internal memory of the reconfigurable module. It stores the application input data and results. The FB has two memory banks. This division lets loading output data to one bank at the same time the other bank loads data to reconfigurable cells without increasing execution time.

The DMA Controller controls traffic between main memory and Frame Buffer or Context Memory but it does not allow simultaneous transfers of data and contexts. TinyRISC is a 32 bits RISC processor, that controls the system. It has an instructions set as a MIPS, adding control instructions of Mosphosys. TinyRISC loads contexts in the CM, loads data from main memory to the FB, stores Morphosys results into main memory, and controls the DMA and RC Array. It decides which configuration is executed at each time.

We have added another internal memory to the original MorphoSys model, the Z-buffer, it was designed for executing graphic applications. We store depth information of screen pixels into it. Data is structured as an array with the same number of rows and columns as the width and height of the screen. Each pixel is stored into Z-buffer only if it has lower depth than the stored pixels. So, we only store pixels of the visible objects, not of the hidden parts.

3. Rendering algorithm

Rendering is the process of generating an image from a model. The model is a description of three dimensional objects, it would contain geometry, viewpoint, texture and lighting information. Then, rendering algorithm target is to project a 3D scene in a graphic device (2D).

It is one of the major sub-topics of 3D computer graphics. In the 'graphics pipeline' it is the last major step, giving the final appearance to the models and animation. It has uses in: computer games, simulators, movies special effects, and design visualisation. Each one employs a different balance of features and techniques. As a product, a wide variety of renders are available.

The rendering algorithm used in our work transforms a 3D scene made up of triangles in a 2D representation. The algorithm processes all the triangles in the scene, and it transforms their 3D points coordinates (vertex) in the corresponding projection coordinates, and these last coordinates are the screen pixels in the graphic device (Figure 2). The implemented rendering algorithm is the scanline rendering [9], it works on a point-by-point basis rather than polygon-by-polygon basis. Some point in a line is calculated, followed by successive points in the line. When the line is finished, rendering proceeds to the next

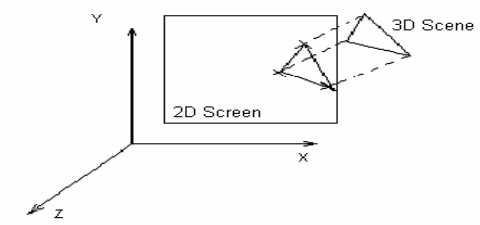

Figure 2. Parallel projection of one triangle

line. This kind of algorithm can be easily integrated with the Phong reflection model [11] and Z-buffer algorithm [12].

The rendering algorithm repeats the following steps for each triangle that compose the scene:

- It orders the triangles points (x_0,y_0,z_0), (x_1,y_1,z_1) and (x_2,y_2,z_2), using the 'y' coordinate from smaller to bigger, in such a way that the A vertex correspond with the smallest 'y' value and the C vertex with the greatest one. It is done to make triangle filling easier.

-These vertexes are projected into the screen.

-Triangles pixels are drawn in two stages for each 'y' line of pixels. The algorithm calculates first the edge pixels of the line, these are the pixels of the lines AB, AC and BC. And later, the pixels in between these two edges are drawn.

- After drawn one line, the 'y' coordinate is increased by one, and the new edges are calculated. Later the pixels in between these two edges are also computed. The algorithm continues till drawn the whole triangle (Figure 3).

- The z value of each pixel, which is required for the depth test is calculated through linear extrapolation from the z_0, z_1 and z_2 vertex values.

In order to facilitate the triangle edge calculation the render can divide the triangle into two sub-triangles (Figure 3). Firstly, the pixels between AB and AC are drawn until the 'y' coordinate is greater than the 'y' value for vertex B. Then, the algorithm draws the pixels between lines BC and AC. It is always possible to divide the triangle into two pieces because we have ordered the triangle vertex before they are drawn.

In order to obtain an image with enough realism we apply to each pixel a Gouraud shading algorithm [13]. We calculate the light intensity in each triangle vertex applying $I_d = I_i K_d \cos\alpha$, which is the intensity of diffused light given by Lambert's Law. I_i stands for the intensity of the light source, α is the angle between the surface normal and a line from the surface point to the light source (it varies between 0 and 90 degrees), and K_d is a constant between 0 and 1, which is an approximation to the diffuse reflectivity which depends on the nature of the material and the wavelength of the incident light. This equation can be also written as the dot product of two unit vector:

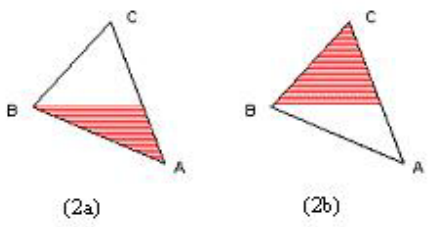

Figure 3. (a): filling pixels between AB and AC. (b): filling pixels between BC and AC.

$$I_d = I_i K_d (\boldsymbol{L} \cdot \boldsymbol{N})$$

Where $\boldsymbol{N}$ is the surface normal vector and $\boldsymbol{L}$ is the direction of vector from the light source to the point on the surface. We calculate this value for the three vertex of the triangle, and then, we realize a linear interpolation form this point to compute the light intensity in each pixel of that triangle. This method produces results with high quality in a reduced computation time.

Finally, the algorithm checks the parts of the image visible, deleting the hidden ones. The 3D scene is composed by several objects, described as a set of triangles, some objects may occlude the others in the final 2D image. We use the Z-Buffer algorithm to compute the visible part of the image. In this case, the algorithm has to draw in the screen the lower depth pixels, those closer to the screen. The Z-Buffer is represented in a matrix structure, like screen pixels matrix. This matrix stores the 'z' value and the colour of the current pixel in the position 'x' and 'y', its coordinates. When a new triangle is rendered the new pixel colours and 'z' value is only stored if this new 'z' coordinate is smaller than the current 'z' in the 'x' and 'y' position of the Z-Buffer.

We have chosen a simple rendering algorithm, but it obtains, as experimental results demonstrate, a high quality images in an interactive execution time. Moreover, we will be able to add improvements incrementally, adding new rendering characteristic to the algorithm pipeline.

4. Implementation Strategies

There are several possible implementation strategies for the rendering algorithm into MorphoSys. The strategy finally chosen must minimize the execution time. It can be achieved by minimizing the period of time in which the reconfigurable cells are idle and reducing the time wasted in data transfers not overlapped with computation.

The rendering algorithm implemented was described in the section above. From that description and taking into account the MorphoSys architecture we must find the different macro-tasks (kernels) that compose it. We can infer the existence of 4 different kernels (Figure 4). The 3D scene, described by triangles, is stored in the FB. The "Vertex Ordering" kernel orders the tree triangles vertex. The "Edge Pixel" kernel finds the edge pixels of the corresponding triangle edges. The "z-buffer" kernel sends the pixels found to the z-buffer. This kernel has to send the pixels obtained by the "edge pixels" kernel and also by the "line pixels" kernel. The internal pixels are calculated by the "line pixel" kernel, which found one by one the pixels in the x-line between the two edges. When the algorithm ends one x-line, it has to calculate the new edges of the upper x-line. The control point 2 (Figure 4) checks if the last pixel founded is the last on the x-line. We also need a control point to check if the triangle is completely render (control point 1, Figure 4). Moreover, the final pixel

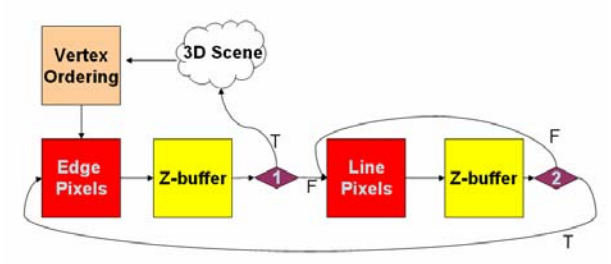

Figure 4. Flow Diagram of the Rendering Algorithm

colour, and all the environment effects can be added to the diagram flow after the edge and line pixels kernels.

The kernels' internal loops and if-then-else structures are easily solved by pseudo-branch contexts, which were previously used for mapping, for example, the ray-tracing algorithm [14]. The problem appears when we try to mapping the control points 1 and 2 at Figure 4, in a SIMD reconfigurable architecture, as MorphoSys. For example, the control point 2 checks if all the pixels between the two edges of the line have been rendered. Then it is very likely that several reconfigurable cells have ended the line rendering process, while the others require render more pixels. In that case, the RCs that have ended that process are idle while the others are executing the loop, due to the SIMD execution model. In the other control point, control point 1, it checks if the triangle is completely render. The triangles sizes and positions are completely random, so in the most of the cases we would have a great number of RCs idle. The implementation finally chosen must minimize this effect.

There several possible strategies:

- The first one implies the execution of one triangle in each RC. This means the parallel execution of 64 triangles of different sizes, with an important problem of coherence. Beside that, there is also a problem of memory bandwidth due to the triangles concurrent execution. The 64 cells are sending to the z-buffer 64 possible pixels at the same time that could address the same pixel in memory. This occurs when two cells obtain the same pixel but with different 'z' values (different depth).
- Other possible strategy involves the execution of one kernel per row (or column) of the RC Array, since MorphoSys can be configured per rows. It can be done if the different rows configurations are known at compilation time. This solution could be performed executing the first kernel in the first row, the second in the second one, and go on. An improvement of this solution would imply the usage of one column to perform the "z-buffer" kernel. We could map 2x8 rendering pipelines. However, the kernels have very different execution time, then the most part of the computation time is being spent in the execution of the "line pixels" kernel while the rest of the RC Array (6 rows of 8 RCs) is idle. Although we could think of use the idle rows to perform "line pixels", this is not possible because the position and color of the next pixel in the line is calculated through the current pixel.
- The vertex ordering allows us to execute each half of the triangle independently. Therefore, we can map half triangle in each RC. There are two differences between this solution and the first one which make it more suitable for MorphoSys. First, it only renders 32 triangles which make the possibility of differences among them lower. But, we are not increasing the execution time because ideally, as it processes half triangle, it takes half the time. Second in the first solution we should send pixels one by one to the z-buffer to avoid incoherencies. In this case we can send two pixels at times, because they belong to the same triangle, then they cannot address the same pixel in the screen, so the same address in the z-buffer.

However, there is still a problem with z-buffer kernel, because it still takes for the third strategy 32 cycles. During this time the 96% of the RCs are idle.

5. The z-buffer bottleneck

From the above discussion is clearly that the transfer of pixels' data to the z-buffer is the application bottleneck. In this section we discuss the different approaches to reduce its effect.

One possible solution would imply the usage of one row of RCs to discriminate the different pixels. Each row of the architecture sends its corresponding pixels to this row, and it performs the z-buffer algorithm deleting those pixels that have the same position in the screen and greater depth. However, it is impracticable because the time required to check 56 pixels is huge compared with pixel kernels calculation. For example the "line pixels" kernel only takes 18 cycles. Then, there would be 7 rows the majority of the time idle.

An improvement to this solution involves a kind of row sequential strategy. We propose to execute the whole algorithm in each row, keeping one row to execute the z-buffer kernel, but with several cycles delay for different rows. Therefore, the row dedicated to perform the z-buffer only has to compare 8 pixels at time. Even so, it takes at least 16 cycles. This solution is completely inefficient, above all because it takes more time than send one pixel a time to the z-buffer by the row that produce them, which takes 8 cycles.

It is clear from the above discussion that a specific row for help the z-buffer is not required. Moreover, if we apply row sequential to the third strategy, the algorithm only takes 4 cycles per row to send the pixels, because the two pixels from the same triangle cannot belong to the same pixel, so they can be sent at a time.

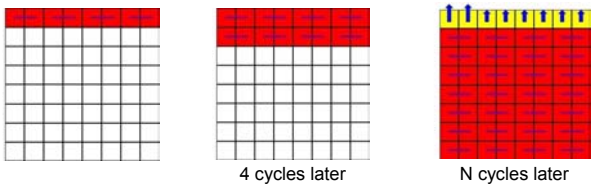

Figure 5. Different snapshots of the RC-Array for the execution

Therefore, the final strategy chosen is: execute the algorithm in rows sequentially, where each row also performs "z-buffer" kernel (Figure 5). The rows in white are idle, the rendering algorithm is executed over the first row, after the fourth cycle the rendering algorithm begins to be executed in the second row. 4 cycles later it begins to be executed in the third row and go on. Then after 'N' cycles, depends on the final render chosen, the first row send its pixels in four cycles (we call this the "z-buffer" kernel), and so, 'N+4' cycles later the second row send its pixels.

The previous solution has a problem depending on the "edge pixels" and "line pixels" execution times. It is because after these two kernels the "z-buffer" kernel is executed, then, it can occur that when one row is sending the pixels after the edge kernel other row, at the same time, is sending pixels after the line kernel. In order to avoid this situation we must add idle cycles to each row after the first execution of the z-buffer kernel (Figure 6). For our implementation we know that the idle cycles we should be five, but the algorithm dedicate 3 cycles to check the control point 1, so finally the rows are only two cycles idle. For example, in the Figure 6.a in the next cycle the "line pixels" kernel begins on row one. In the Figure 6.b in the next cycle the first row will begin to send the pixel, meanwhile in this cycle the last row is ending these transfers.

6. We remind the reader that in case of any RC of the same row ended one of the two loops before the others it remains idle until all the cells on the same row finalize the corresponding loop.

7. Experimental Results

MorphoSys is designed to be running at 450MHz. It has 512x16 internal RC-RAM, 4x16Kx16 FB, 64Kx16 Z-buffer, 8x1Kx32 Context Memory, and 16 internal registers in each RC. The chip size is less than 30mm^2 using 0.13um CMOS technology. Thus, MorphoSys is more power efficient than general-purpose processors.

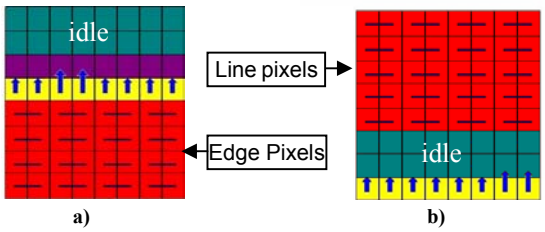

Figure 6. Idle cycles added to the pipeline

Strategy	Cycles	
	without optimization	with optimization
Sequential Processor	16995436	
One triangle per RC	5008328	1006711
One triangle every 2 RCs	2676626	872093

Table 1. Different strategies: comparative results

The algorithm was translated into MorphoSys Assembly and then into machine code. The Simulation is run on MorphoSys processor simulator "Mulate" [4].

Table 1 shows the experimental results obtained with the different strategies over the example of Figure 7 (Stanford's bunny). These results demonstrate that the strategy with the lowest execution time is the third with the z-buffer optimization

We have also rendered several images with different number of triangles, as is shown in figures from 7 to 11. The largest one corresponds to Thorax and hip with more than one million of triangles, we obtain 79 fps as Table 2 shows. In the case of figures of lower number of triangles our implementation reaches more than 600 fps which demonstrate that interactivity is possible in MorphoSys. This also indicates that we can improve the rendering algorithm in order to obtain a better quality images because we have enough frame per second.

If we compare our results with commercial render architectures, we have obtained similar results in triangle per second. For example, "Chromium" [15], which is a cluster of 32x2 Pentium III render images a rate of 71 million of triangles per second; the Nvidia GeForce FX 5200 [10], that is a standard graphic card, render images as a rate of 81 million of triangles per second. Moreover, MorphoSys architecture can also be used to implement a wide range of multimedia and DSP algorithms obtained competitive results.

Image	**Triangles**	**fps**	**T/s**
Bunny (Fig. 7)	69451	516	35836716
Horse (Fig. 8)	96966	657	63706662
Buda (Fig. 9)	1087716	133	14466.228
Thorax and hip(Fig. 10)	1136745	73	82982385
Hand (Fig. 11)	654666	128	83797248

Table 2. Frame per second (fps) and triangles per second (T/s) for different images.

8. Conclusions

In this paper we have demonstrated that is possible to get 3D image representations and interacting with them.

Algorithm has been implemented against a coarse-grained reconfigurable hardware device with a

SIMD execution model. In this implementation we have found all cells are doing useful job most part of the time. The render can be easly improved adding more realistic effects and keeping interactive results. At architectural level we could improve the use of z-buffer in the way of using a smaller size memory than screen pixels.

References

1. Catherine Compton (Northwestern University) and Scott Hauck (University of Washington), "Reconfigurable Computing: A Survey of Systems and Software". Pages: 1-2.

2. Katholic Universiteit and Vrije Universiteit of Belgium, "ADRES: An Architecture with tightly coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix". Págs: 1-3.

3. R. Hartenstein (1997). "The Microprocessor is no more General Purpose: why Future Reconfigurable Platforms will win; invited paper", *Proc. International Conference on Innovative Systems in Silicon, ISIS'97*, Austin, Texas, USA, October 8-10.

4. H. Singh, M.-H Lee, G. Lu et al. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Trans. on Computers, 49(5):465-481, May 2000.*

5. C. Ebeling, D. Cronquist, et al. "RaPiD – reconfigurable pipelined datapath". In *Proc. International Workshop on Field Programmable Logic and Applications*, 1996. Pages: 23-25.

6. *PACT XPP Technologies*, 2003. http://www.pactcorp.com.

7. T. Miyamori and K. Olukotun: REMARC: Reconfigurable Multimedia Array Coprocessor; Proc. ACM/SIGDA FPGA '98, Monterey, Feb. 1998. Page: 261.

8. Marcos Sánchez-Élez, Seminario Internacional sobre Sistemas Dinámicamente Reconfigurables (Universidad de Antioquia, Colombia), "Arquitecturas Reconfigurables de Grano Grueso".

9. Alan Watt, "3D Computer Graphics". 3rd Edition. Addison-Wesley.

10. www.nvidia.com

11. R. L. Cook , K. E. Torrance, A Reflectance Model for Computer Graphics, ACM Transactions on Graphics (TOG), v.1 n.1, p.7-24, Jan. 1982

12. Watkins, G. "A Real-Time Visible Surface Algorithm", Computer Science Department, University of Utah, UTECH-CSC-70-101, June 1970

13. X.J. Guo, B. Land. "Phong shading and Gouraud shading". http://www.nbb.cornell.edu/neurobio/land/OldStudentProjects/cs490-95to96/guo/report.html

14. M. Sanchez-Elez, H. Du, N. Tabrizi, et al. "Algorithm Optimizations and Mapping Scheme for Interactive Ray Tracing on a Reconfigurable Architecture" Computer & Graphics 27 (2003), Elsevier.

15. G. Humphreys, M. Eldridge, M. Everett et. al. "WireGL: A Scalable Graphics System for Clusters" Proceedings of the SIGRAPH 2001.

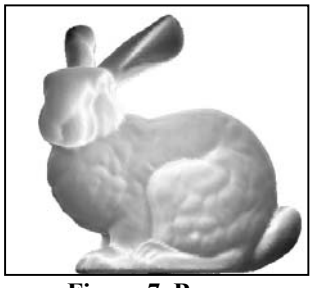

Figure 7. Bunny

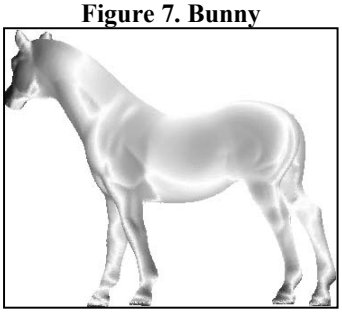

Figure 8. Horse

Figure 9. Buda

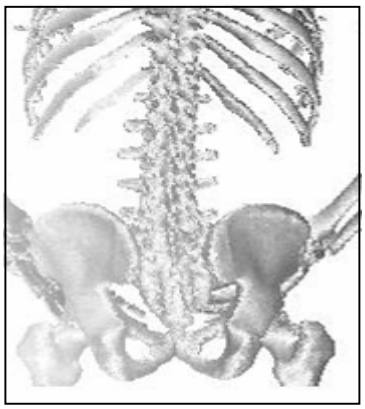

Figure 10. Thorax and hip

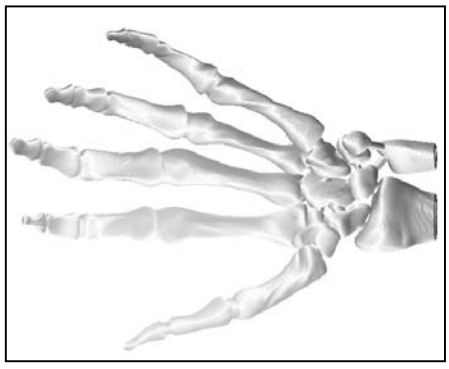

Figure 11. Hand