Terrain Rendering Engine (TRE)

May 13th, 2005

Introduction

This paper provides an overview of the Cell-based Terrain Rendering Engine (TRE). It discusses ray-casting as a rendering technique, the challenges of ray-casting on the Cell microprocessor, the software organization of the TRE, and the resulting performance. This paper is meant to be an overview of these topics, not a deep dive.

Ray-casting Overview

The TRE uses a highly optimized height field ray-caster to render images. Ray-casting, like ray-tracing, follows mathematical rays of light from the viewer's virtual eye, through the view plane, and into the virtual world. The first location where the ray is found to intersect the terrain is considered to be the visible surface. Unlike ray-tracing which generates additional secondary rays from the first point of intersection to evaluate reflected and refracted light, ray-casting evaluates lighting with a more conventional approach. At this first point of intersection a normal is computed to the surface and a surface shader is evaluated to determine the illumination of the sample. Ray-casting has the performance advantage, unlike polygon rendering, of only evaluating the visible samples. The challenge of producing a high performance ray-caster is finding the first intersection as efficiently as possible. While ray-casting is inherently a data-parallel computational problem, the entire scene must be considered during each ray intersection test.

The TRE is optimized to render height fields which are 2D arrays of height values that represent the surface to be viewed. A technique call "vertical ray coherence [1]" was used to optimize the ray-casting of such surfaces. This optimization uses "vertical cuts", which are half-planes perpendicular to the plane of the height map, all radiating out from the vertical line containing the eye point. Each such vertical cut slices the view plane in a not-necessarily-vertical line [figure 1]. The big advantage is that all the information necessary to compute all of the samples in that vertical cut is contained in a close neighborhood of the intersection between the vertical cut and the height map. Furthermore, each sample in the vertical cut, when computed from the bottom to top, only needs to consider the height data found in the buffer starting at the last known intersection. These two optimizations:

a) Greatly reduce the amount of height data that needs to be considered when processing each vertical cut of samples.
b) Continue to reduce the data that needs to be considered as the vertical cut is processed.

Cell Processor Overview

The Cell Processor consists of one dual-threaded Power Processor Element (PPE) with eight SIMD Synergistic Processor Elements (SPEs), along with an on-chip memory controller (MIC), and an on-chip controller for a configurable I/O interface (BIC) interconnect with a coherent on-chip Element Interconnect Bus (EIB) [5][9][figure 3][figure 4].

The key attributes of this concept are:

(1) A high frequency design, allowing the processor to operate at a low voltage and low power while maintaining high performance [6].

(2) Power Architecture® compatibility, to provide a conventional entry point for programmers, for virtualization and multi-OS support, and to be able to leverage IBM's experience in designing and verifying symmetric multiprocessors (SMP).

(3) SIMD architecture, supported by both the vector (VMX) extensions on the PPE and the SPE, as one of the means to improve game/media and scientific performance at improved power efficiency.

(4) A power and area efficient PPE that implements the Power Architecture ISA and supports the high design frequency.

(5) SPE with local memory, asynchronous coherent DMA, and a large unified register file, in order to improve memory bandwidth and to provide a new level of combined power efficiency and performance [7][8].

(6) A high-bandwidth on-chip coherent fabric and high bandwidth memory, to deliver performance on memory bandwidth intensive applications and to allow for high-bandwidth on-chip interactions between the processor elements.

(7) High bandwidth flexible I/O configurable to support a number of system organizations, including a single BE configuration with dual I/O interfaces, and a glue-less coherent dual processor configuration.

(8) A full custom modular implementation to maximize performance per watt and performance per square mm of silicon and to facilitate the design of derivative products.

These features enable the Cell Processor to excel at many rendering tasks including ray-casting. The key to performance on the Cell processor is leveraging the SPEs efficiently by providing multiple independent data parallel tasks that optimize well in a SIMD processing environment.

Ray-casting on Cell

When implementing ray-casting of height fields we use vertical ray coherence to break down the rendering task into data parallel work blocks, or vertical cuts of screen space samples, using the PPE, and dispatching the blocks to ray kernels running on the SPEs. It is key that the SPEs work as independently as possible so as to not over task the PPE, given the 8 to 1 ratio, and we don't want the SPE's streaming data flowing through the PPE's cache hierarchy. These work blocks are therefore mathematical descriptions of the vertical cuts based on the current view parameters. Where the plane intersects the view screen are the locations of the accumulation buffer to be modified [figure 1]. Where the plane intersects the height map are the locations of the input height data needed to process the vertical cut of samples [figure 2]. The PPE communicates the range of work blocks each SPE is responsible for via base address, count, and stride information once per frame. The SPE then uses this information to compute the address of each work block which is then read into local store via a DMA read and processed. This decouples the PPE from the vertical cut by vertical cut processing allowing it to prep the next frame in parallel. The SPEs place the output samples in an accumulation buffer using a gather DMA read and a scatter DMA write. Each SPE is responsible for four regions of the screen and the vertical cuts are processed in a round robin fashion, one vertical cut per region, and left to right within a each region, so no synchronization is need on the output as no two SPEs will ever attempt to modify the same locations in the accumulation buffer even with two vertical cuts in flight (double buffering) per SPE. The SPEs are therefore free to run at their own pace processing each vertical cut without any data synchronization on the input or output. None of the SPE's input or output data is touched by the PPE, thereby protecting the PPE's cache hierarchy from transient streaming data.

Ray Kernel

Each SPE executes a ray kernel that consists of four functional code blocks.

1) A command parser that looks for work in the mailbox and dispatches the proper code fragment.
2) Load/store code that fetches the work blocks and computes the DMA lists to read the height/color data and write the accumulation data.
3) Ray intersection code that searches for the first intersection of each ray and height data.
4) Surface shader code that evaluates the surface at each intersection and computes the output color of each sample.

The command parser runs on a per frame basis as work is communicated from the PPE. Work is processed on a per vertical cut basis where a work block is fetched, DMA lists for the input data are computed, intersections are located, samples are evaluated with the surface shader, and then written back to the accumulation buffer. Within each SPE the ray kernel processes the data using double buffered inputs and outputs so as to cover up the memory latency and data load time. Height and accumulation data is brought in for vertical cut n+1 while samples are computed for vertical cut n. Texture color information is interleaved into the height map for each location, 16 bits of color (5/6/5) and 16 bits of height, so both are in vector registers when the intersection is located [figure 6]. The SPE's DMA engine is optimized to fetch multiples of quad words (128 bits) and the surface shader requires a four by four neighborhood to evaluate the normal so the interleaved height/color data structure works well. Samples are computed in a SIMD fashion by searching for four intersections at a time and when all four are located they are then evaluated in parallel by the surface shader.

The surface shader provides both terrain illumination and atmospheric effects. The current shader contains the following functionality:

1) Surface normal computation via four by four height neighborhood
2) Texture filtering via a two by two color neighborhood.
3) Bump mapping via a normal perturbation function [3]
4) Surface illumination via diffuse reflection and ambient lighting model.
5) Visible sun plus halo effects
6) Non-linear atmospheric haze
7) Non-linear ground fog
8) Resolution independent Clouds computed via multiple octaves of Perlin noise evaluated on the fly [4].

Dynamic multi-sampling is implemented by adjusting the distance between rays along the vertical axis (vertical cut) based on the under or over sampling of the prior four rays [2]. The sampling rate along the horizontal axis is a function of the size of the image, and is computed to ensure that no pixel is skipped over, even after accounting for round off error.

The resulting IBM XLC compiled C code occupies 30KB of local store and uses 212KB of buffer space plus 5KB of stack.

Image Compression Kernel

In addition to the SPEs running the ray-kernels one SPE is reserved for image compression. The compression kernel operates on the finished accumulation buffer which is organized in a column major 2D array of single precision floating point (red, green, blue, count) data, one float per channel. The count channel contains the number of samples accumulated in the pixel. The image compression kernel is responsible for the following tasks:

1) Normalization of the accumulation buffer
2) Compression of the normalized buffer.
3) Clearing of the accumulation buffer

The compression kernel reads sixteen by sixteen pixel tiles from the accumulation buffer into local store using DMA lists. These tiles are then normalized by dividing the color channels by the sample count for each pixel. The tile is then compressed using a multi-stage process involving color conversion, discrete cosine transformation (DCT), quantization, and run length encoding. The resulting compressed data is then written to a holding buffer in local store which when full is DMA written back to system memory for network delivery. As each tile is processed a tile of zeros is returned to the accumulation buffer providing the clear operation.

TRE System Components

The TRE was implemented using a client server model. The client, implemented on an Apple G5 system, is connected to the server via a gigabit Ethernet. The client specifies the map, path, and rendering parameters to the server which in turn streams compressed images back to the client for display.

TRE Server

The TRE server, implemented on the STIDC Bring-up boards, runs in both a uni-processor (UP) and symmetric multi-processor (SMP) mode [figure 5]. The server scales across as many SPEs as there are available in the system. One SPE is reserved for image compression and the remaining available SPEs run ray kernels. Three threads execute on the available Muti-Threading (MT) slots of the PPE. The three threads responsibilities break down as follows:

1) Frame preparation and SPE work communication.
2) Network tasks involving image delivery.
3) Network tasks involving client communication.

Thread one requires the most processor resource followed by threads two and three, so threads two and three share the same MT processor affinity in the uni-processor model.

The server implements a three frame deep pipeline to exploit all the parallelism in the Cell processor. In stage one, the image preparation phase, PPE thread one decomposes the view screen into work blocks based on the vertical cuts dictated by the view parameters. Stage two, the sample generation phase, is where the SPEs decompose the vertical cuts into samples and store them in the accumulation buffer. In stage three, the image compression/network phase, the compression SPE encodes the finished accumulation buffer and PPE thread two delivers it to the network layer. All three stages are simultaneously active on different execution cores of the chip maximizing the image throughput.

TRE Client

The TRE client directs the rendering of the TRE server. It provides for the following facilities:

1) Map loading and delivery to the server
2) Smooth path generation both user directed and random.
3) Joy stick directed paths
4) Rendering parameter modification
    a. Sun angle
    b. Lighting parameters
    c. Fog and haze control
    d. Output image size
5) Server connection and selection
6) Map cropping
7) Streaming image decompression and display

TRE Performance

The server has many rendering parameters that effect performance including:

1) Output image size
2) Map size
3) Visibility to full fog/haze
4) Multi-sampling rate

For benchmarking purposes the following values will be selected [figure 7]:

1) 1280x720 (720p) output image size
2) 7455x8005 Map size
3) 2048 map steps to full haze (10m map -> 20Km visibility)
4) 1.33 x (2 – 8 Dynamic) or ~2-32 samples per pixel

With these settings the following rendered and compressed relative image rates were captured:

Processor	Performance
2.0 GHz G5 VMX	1 (No Image Encode)
2.4 GHz UP Cell	36
3.2 GHz UP Cell	50
2.4 GHz 2-way SMP Cell	75

The dependency-reducing 128 vector registers and throughput boost of dual issue enable the SPE to execute the ray kernel at an impressive 0.70 to 0.80 cycles per instruction (CPI). The two blocks of code that account for the majority of ray kernel cycles are the intersection test and shader. The intersection test is broken into two phases, the first searches for the initial ray's intersection and the second more carefully finds each intersection for rays in the vertical cut thereafter. The compute and data fetch phases of the ray kernel execute in parallel by exploiting the asynchronous execution of the SPE's DMA engine (SMF) and SIMD core (SPU). Multiple input and output buffers are used to decouple the two phases.

Conclusion

In this paper we have outlined how a compute intensive graphics technique, ray-casting, can be decomposed into data parallel work blocks and executed efficiently on multiple SIMD cores within one or multiple Cell processors. We have also shown that by pipelining the work block preparation, execution, and the delivery of results, additional parallelism within the Cell processor can be exploited. One order of magnitude performance improvements over similarly clocked single threaded processors can be achieved when computational problems are decomposed and executed in this manner on Cell. Graphics techniques once considered offline tasks can now be executed at interactive speeds using a commodity processor.

References

[1] Cheol-Hi Lee, "A Terrain Rendering Method Using Vertical Ray Coherence", The Journal of Visualization and Computer Animation

[2] Cohen-Or, D., Rich, E. Lerner, U., and Shenkar, V., "A real-time photo-realistic visual flythrough," IEEE® Trans, Vis. Comp. Graph., Vol 2, No 3, September 1996.

[3] Blinn, "Simulation of Wrinkled Surfaces", Computer Graphics, (Proc. Siggraph), Vol. 12, No. 3, August 1978, pp. 286-292.

[4] Perlin, Ken, "Making noise," http://www.noisemachine.com/talk1/

[5] H. Peter Hofstee, "Power Efficient Processor Architecture and the Cell Processor", to appear, Proceedings 11th Conference on High Performance Computing Architectures, Feb. 2005.

[6] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V.Zyuban, P.N. Strenski, and P.G. Emma "Optimizing pipelines for power and performance," in *Conf. Proc. 35th Annual IEEE/ACM International Symposium on Microarchitecture 2002*. pp. 333-344.

[7] B. Flachs, S. Asano, S. H. Dhong, H. P. Hofstee. G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H-J. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano. "The Microarchitecture of the Streaming Processor for a CELL Processor," *IEEE International Solid-State Circuits Symposium*, Feb. 2005.

[8] T. Asano, T. Nakazato, S. Dhong, A. Kawasumi, J. Silberman, O. Takahashi, M. White, H. Yohsihara, "A 4.8GHz Fully Pipelined Embedded SRAM in the Streaming Processor of a CELL processor", *IEEE International Solid-State Circuits Symposium*, Feb. 2005.

[9] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, K. Yazawa, "The Design and Implementation of a First-Generation CELL Processor," *IEEE International Solid-State Circuits Symposium,* Feb. 2005.

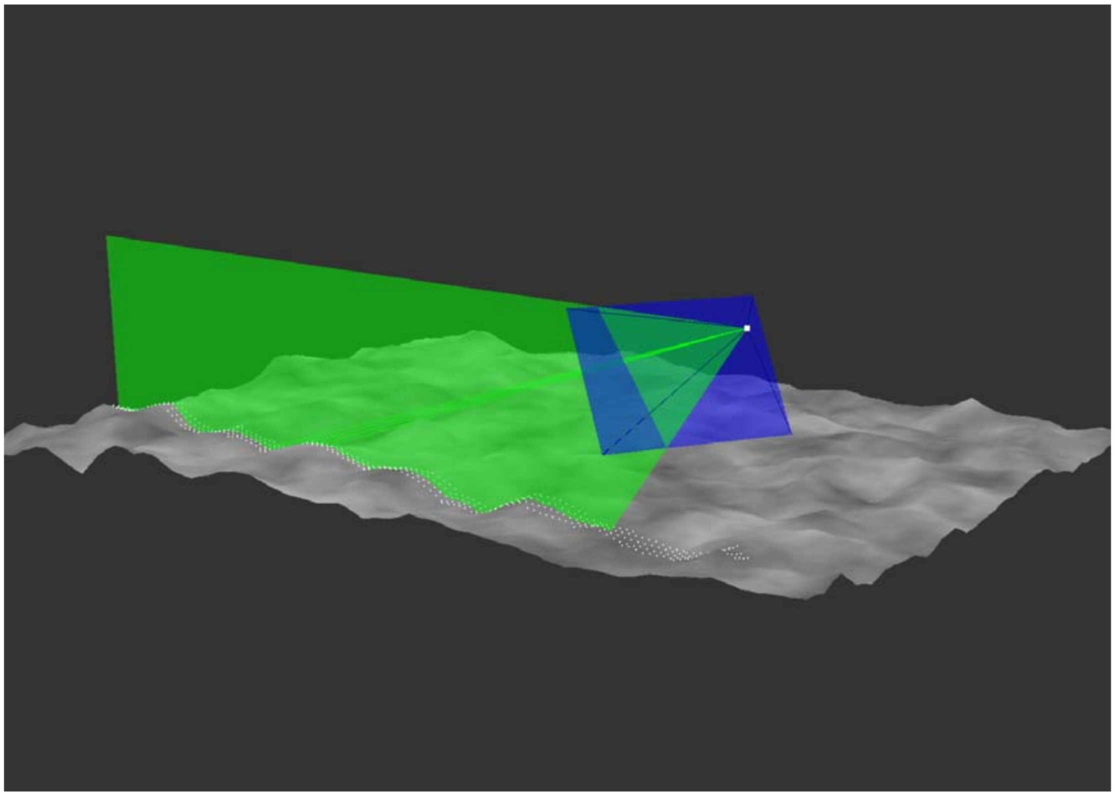


Figure 1: Height Field Ray-Casting.

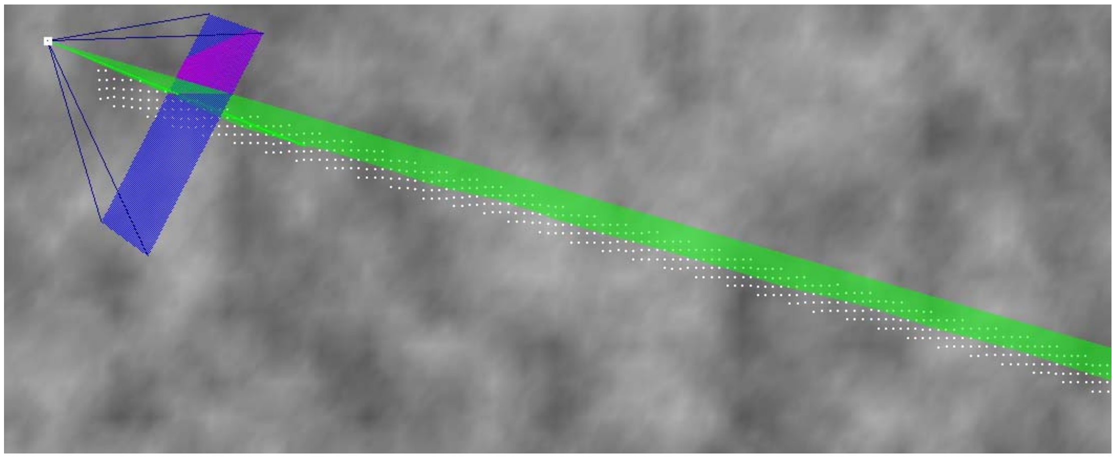


Figure 2: Height Field Ray-Casting overhead.

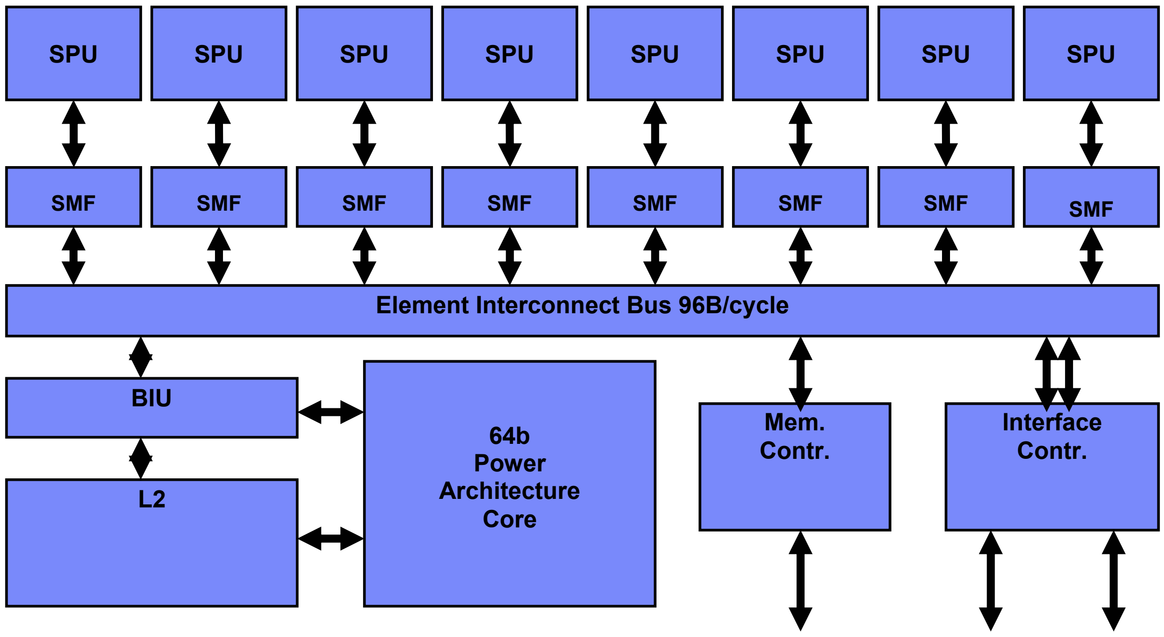


Figure 3: Cell Processor block diagram.

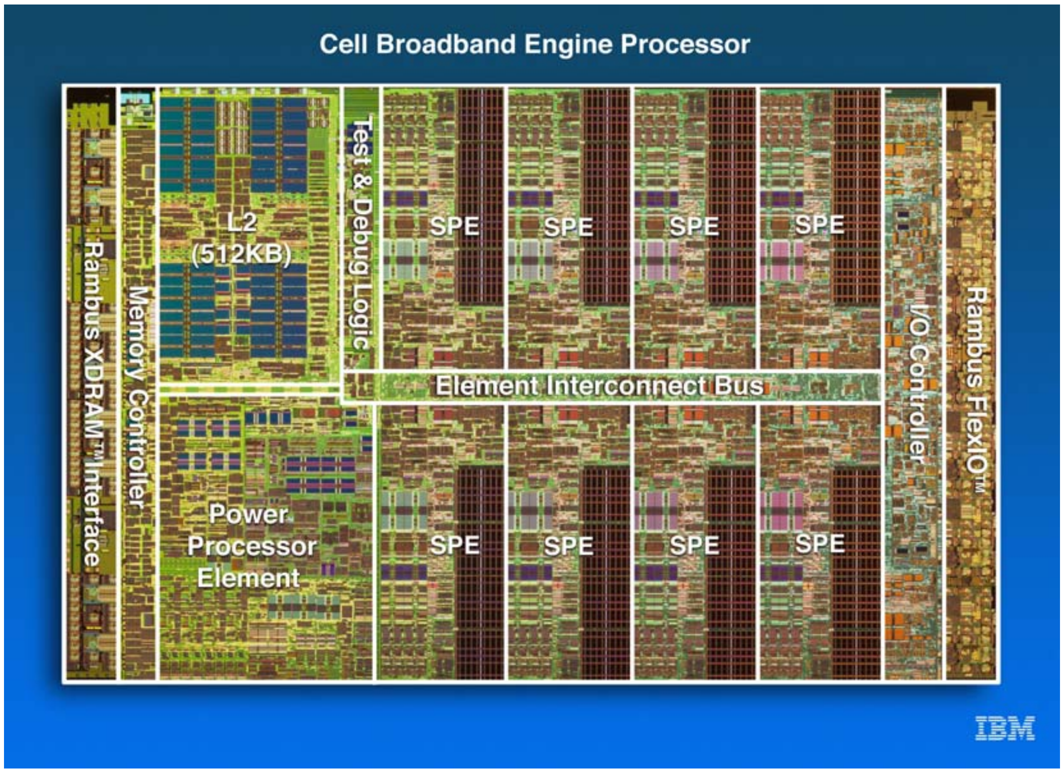
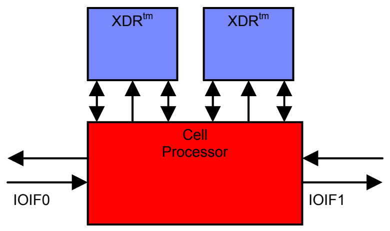
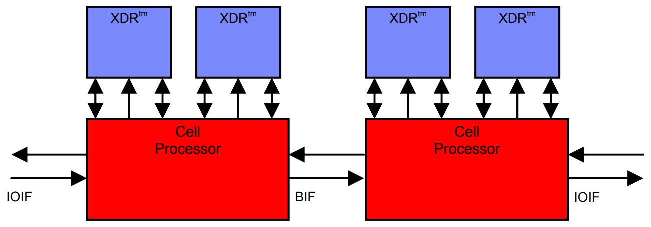


Figure 4: Cell processor die photo.



Uni-processor (UP)



Symmetric multi-processor (SMP)

Figure 5: Cell processor configurations.

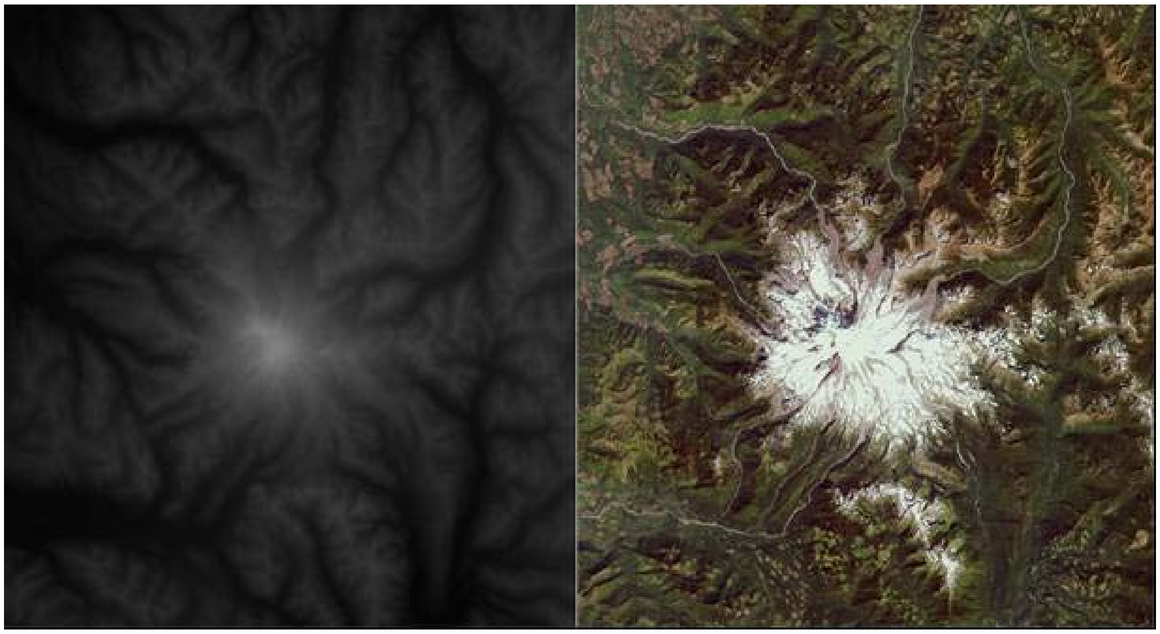


Figure 6: TRE Input Data.

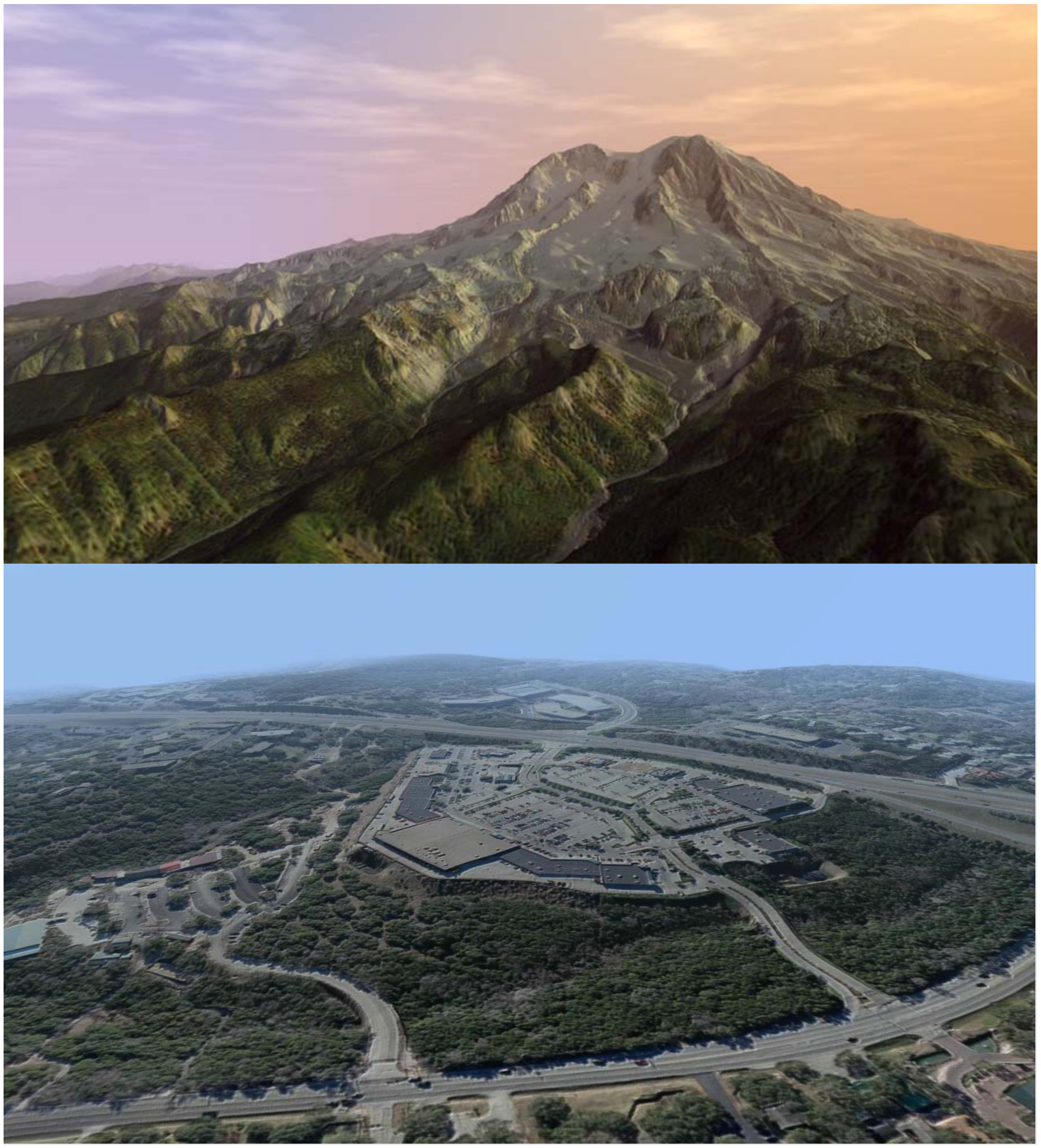


Figure 7: TRE Output

Authored by IBM

Barry Minor, bminor@us.ibm.com
Gordon Fossum, fossum@us.ibm.com
Van To, vto@us.ibm.com