Computers & Graphics 38 (2014) 140–149

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on CAD/Graphics 2013

Effective traversal algorithms and hardware architecture for pyramidal inverse displacement mapping

Hyuck-Joo Kwon [a], Jae-Ho Nah [b], Dinesh Manocha [b], Woo-Chan Park [a,*]

[a] Department of Computer Engineering, Sejong University, 98 Gunja-Dong, Gwangjin-Gu, Seoul 143-747, Republic of Korea
[b] University of North Carolina at Chapel Hill, Chapel Hill, NC, USA

ARTICLE INFO

Article history:
Received 5 August 2013
Received in revised form
12 October 2013
Accepted 19 October 2013
Available online 6 November 2013

Keywords:
Graphics processors
Hardware architecture
Displacement mapping
Image pyramid

ABSTRACT

We present an effective traversal algorithm and a hardware architecture to accelerate inverse displacement mapping. This includes a set of techniques that are used to reduce the number of iterative steps that are performed during inverse displacement mapping. For this purpose, we present two algorithms to reduce the number of descending steps and two algorithms to improve the ascending process. All these techniques are combined; we observe up to 66% reduction in the number of iterative steps as compared to other pyramidal displacement-mapping algorithms. We also propose a novel displacement-mapping hardware architecture based on the novel techniques. The experimental results obtained from the FPGA and ASIC evaluation demonstrate that our novel architecture offers many benefits in terms of chip area, power consumption, and off-chip memory accesses for mobile GPUs.

1. Introduction

Displacement mapping is a widely used computer graphics technique that shows the effect of the actual movement of geometric points according to a given height field. Modern GPUs offer hardware support for displacement mapping [1–3], and it is widely used to convey depth and details. However, displacement mapping is regarded as more expensive than other mapping techniques, as it involves dealing with a lot of additional geometry and therefore has a high computational load.

Inverse displacement-mapping [4] algorithms, such as parallax mapping [5], parallax occlusion mapping [6,7], relief mapping [8,9], and pyramidal displacement mapping [10–12], have been proposed to improve the performance of displacement mapping. These approaches can determine the intersection point between a ray and a height field by projecting the ray on the height field without changing the geometry. Moreover, inverse displacement mapping has been widely used in CPU and GPU implementations [13].

State-of-the-art methods for inverse displacement mapping can be categorized into two classes [12,13]: approximation (unsafe) and accurate (safe) algorithms. The approximation algorithms, such as parallax occlusion mapping and relief mapping, are fast, but are relatively low-accuracy: there is no guarantee that they will find the correct intersection. A more accurate algorithm that has been proposed in [10–12,14,15] is the per-pixel ray-tracing technique with an image pyramid. The pyramidal displacement map is an image pyramid of the mipmap consisting of many levels of sub-images. By gradually descending from the root level to the leaf level of this map, the accurate intersection point between a ray and a height field can be computed. In [10], visiting a node one level down or visiting a neighbor node using node crossing occurs during the traversal of an image pyramid. Meanwhile, Tevs et al. [12] and Drobot [15] proposed ascending techniques for effective empty-space skipping, which reduced the number of iteration steps for traversal at grazing angles compared with [10]. Dick et al. [16] applied a pyramidal mipmap to GPU ray casting for terrain rendering.

According to [10,12], pyramidal displacement mapping provides several advantages over other accurate algorithms, such as relaxed cone stepping [17] and safety zone techniques [18–20]. First, pyramidal displacement mapping has lower memory requirements and needs only a simple mipmap construction; the other methods require a long off-line preprocessing time and have large memory requirements. Thus, pyramidal displacement mapping is more suitable for large-scale or dynamic height fields. Second, pyramidal displacement mapping provides better image quality than relaxed cone stepping because relaxed cone stepping can miss thin geometry [12].

Main results: In this paper, we present a set of improvements to maximize the traversal performance of pyramidal displacement mapping. Our approach consists of four sub-algorithms: start-level decision, multi-level down, selective level-up, and coherent level-up. The first and second sub-algorithms effectively reduce the required number of iteration steps at front angles. The start-level decision

* Corresponding author. Tel.: +82 2 3408 3752; fax: +82 2 3409 3755.
E-mail addresses: pwchan@sejong.ac.kr, pwchan@sejong.edu (W.-C. Park).

0097-8493/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cag.2013.10.024

algorithm directly calculates the start traversal level of an image pyramid using only the ray information, and the multi-level down algorithm descends multiple levels at one time before the first node crossing. Both algorithms can be used simultaneously. The third and fourth sub-algorithms improve the ascending process of [12,15] by removing unnecessary level switching. The selective level-up algorithm ascends a level only if the predicted intersection point is not located in the neighbor node in the node crossing. The coherent level-up algorithm ascends a level when the consecutive node crossing at the same level occurs. The former is suitable for dedicated hardware architecture due to low iteration steps and the latter is suitable for current programmable GPUs due to its simplicity. According to our experimental results on a NVIDIA GTX460 GPU, the combination of the presented sub-algorithms increases frame rates by up to 85% and 56% compared to [10] and [12], respectively. More importantly, our approach is more robust for both front and grazing angles than previous algorithms [10,12]. This feature helps to maintain frame rates in interactive graphics applications (e.g., games).

We also present a hardware architecture consisting of a height-map traverse pipeline and a texture-mapping unit. The traverse pipeline was specially designed to accelerate the presented traversal algorithms. To evaluate the feasibility of our architecture, we integrated the presented hardware unit into an existing ray-tracing hardware architecture [21,22]. This approach can also be combined with current programmable GPUs, which are primarily designed for rasterization. According to our ASIC evaluation, our hardware architecture can achieve real-time performance with fewer hardware resources, memory accesses, and lower power consumption. Thus, our proposed hardware unit has high potential utility in desktop/mobile GPUs.

The rest of the paper is organized in the following manner. We give an overview of pyramidal displacement mapping in Section 2. In Section 3, we present a set of improved traversal algorithms. In Section 4, we present the hardware architecture and its implementation. In Section 5, we provide the experimental results.

2. Pyramidal displacement mapping

A pyramidal displacement map is a quad-tree image pyramid created through a pre-computing process. An image pyramid is a hierarchical collection of sub-level images from $2^0 \times 2^0$ to $2^n \times 2^n$, where n is the maximum level. The original displacement map data might be specified as mipmap level 0, so that each leaf texel (or node) indicates a displacement value for the actual surface. In the upper-level image, each inner texel (i, j) is obtained by storing the maximum value among the four texels $(2i, 2j)$, $(2i+1, 2j)$, $(2i, 2j+1)$, and $(2i+1, 2j+1)$ in the lower-level image. Thus, the root texel denotes the globally maximum height value, whereas an inner texel indicates the locally maximum height value [10,13].

To find the accurate intersection point between a ray and a height field, the image pyramid is traversed from root level n and leaf level 0. An example of this process is shown in Fig. 1. First, at the texture coordinate P of the ray, the height value d_1 of the image pyramid's root level is read. If the current position P of the ray is advanced to P_1 where the ray and d_1 meet, the image pyramid is descended by one level. Then, the height value d_2 of the current mipmap level is read at P_1. If d_2 is greater than d_1, P_1 is advanced to P_2 where the ray and d_2 meet, and the image pyramid is descended by one level; otherwise, the position P_1 does not move, but the image pyramid is descended by one level. This process is repeated until it reaches the leaf level.

While the image pyramid is searched, the ray cannot be advanced over the boundary of the current node, because we have no information out of the current node. To address this problem, we must check whether the advanced position of the ray lies inside the boundary of the current node. If it crosses, the ray is moved to the boundary of the crossed node. This process is called "node crossing" and the neighbor node is visited as a result.

In [12,15], traversal algorithms were proposed to reduce the number of node crossings in [10] through ascending the mipmap level. Tevs et al.'s method [12] ascends the mipmap level by one in cases where the ray resides at a node boundary divisible by two. This reduces an unnecessary level ascension when traversing to sibling nodes. In contrast, Drobot's method [15] simply performs the mipmap level ascension if node crossing occurs. This algorithm costs very little to implement, but it may cause an unnecessary level ascension. According to our experiments, [12] is faster than [15] in general cases. Therefore, we will only consider [12] when we discuss level-up algorithms.

3. Proposed traversal algorithm

Fig. 2 shows the processing flow of the proposed traversal algorithm, which consists of three main steps. In the first step, the algorithm proposed in Section 3.1 determines the start level of the traversal for the generated ray. This process mainly consists of end-position calculations and start-level decisions. In the second step, the traversal to find the intersection point between the ray and the height field is accomplished from the start level. The process flow varies by the occurrence of node crossings. If node crossing occurs, the level-up method proposed in Section 3.3 is applied; otherwise, the level-down method in Section 3.2 is applied. Lastly, when the traversal finishes, the target texture coordinate is calculated using the intersection point between the view vector and the height map.

3.1. Start-level calculation

The previous traversal algorithms for image pyramids such as [10,12] initiate the traversal from the root level. The proposed

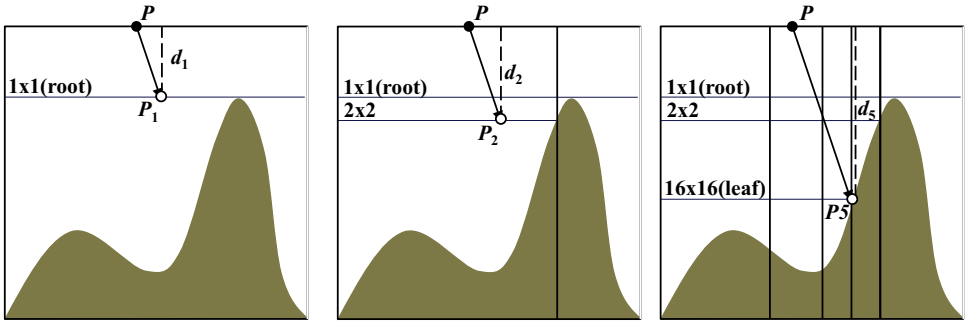

Fig. 1. An example of the traversal process of the pyramidal displacement map.

algorithm in this section determines the start level by calculating the lowest level at which node crossing does not occur for the given ray. This means that mipmap levels higher than the start level do not need to be traversed. This algorithm does not require reading the height map because only the start and end points of the ray are used for the calculation.

Fig. 3 illustrates an example case of a ray from a starting point of P to an ending point of A. P is the input texture coordinate. To calculate A, we use this equation in ray tracing: $r(t)=P+t\cdot D$, where P is the ray's origin, D is the ray's direction vector, and t is an interval parameter. We call this process the end-position calculation. Because the interval of t is equal to the range of the height map value [0,1], the end position A with the maximum t value is $r(1)=P+D$. In Fig. 3, the coordinates of P and A at level 0 are (4,3) and (6,2), respectively. This means that no node crossing occurs in mipmaps from level 2 or higher; therefore, the traversal can start at level 2. In contrast, the previous algorithms always start traversal at the highest level.

Let us assume that two points are $P=(P_x,P_y)$ and $A=(A_x,A_y)$ and the image size of mipmap level 0 is 8×8. After that, P_x, P_y, A_x, and A_y become 3 bits each. In this case, the location of P_x on the image of mipmap level k corresponds to the value of the most significant $3-k$ bits of P_x. For example, in Fig. 3, the location of P_x is $100_{(2)}$ at level 0 and $1_{(2)}$ at level 2. The same applies to A_x. Thus, if the most significant r bits of P_x and A_x are equal, it means that P_x and A_x are at the same node until the r level. This characteristic applies to P_y and A_y , which are the y coordinates of the two points. This can be used to calculate the start level of traversal.

The start-level calculation procedure is described as follows. Assuming that two points are $P=(P_x,P_y)$ and $A=(A_x,A_y)$, a bit-wise XOR (exclusive-or) operation is performed for P_x and A_x and for P_y and A_y. After that, the results of the first and second operations are compared, and the greater number is taken. With this result value, we determine how many bits of leading zeros this result has. Finally, the start level is calculated by subtracting the number of leading zeros from the total number of levels.

In Fig. 3, the results of P_x bit-wise XOR A_x and P_y bit-wise XOR A_y are 010_2 and 001_2, respectively. The greater value of these two is 010_2, and this value has 1 leading zero. When the number of leading zeros (1) is subtracted from the total number of levels (3), the start level becomes 2.

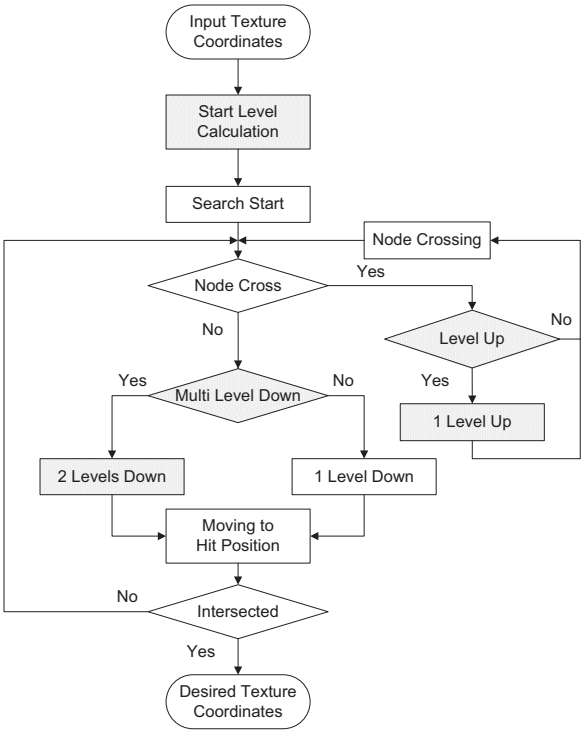

Fig. 2. Processing flow of the proposed traversal algorithm. Shaded parts indicate additional parts for our algorithms compared to [10].

3.2. Multi-level down algorithm

The multi-level down approach is an improved version of the previous traversal algorithm that descends sequentially by one level [10]. In [10], a lower node is visited through the one-level down process and a neighbor node is visited through node crossing. As described in Section 4.1, the frequency of the level down process is generally higher than that of node crossing at front and oblique angles. This implies that using a multi-level down algorithm is likely to reduce the number of iteration steps.

The proposed algorithm simply changes the level-down range, allowing the algorithm to descend multiple n levels of an image pyramid at once. The flow of the proposed algorithm is as follows. First, it determines whether node crossing occurs at the current level. The multiple-level descent is used only if node crossing does not occur. In this case, we determine the optimal n value. If there are no node crossings by this time (the first check) and the current level is more than one (the second check), we descend two levels; otherwise, one level. The reason for the first check is that the proposed algorithm is especially useful until a ray visits the first leaf node. The second check prevents underflow. The descent levels are capped at a maximum of two based on the experimental results in Section 5.2.

Fig. 4 shows an example of the traversal process for the proposed multi-level down algorithm. In the previous one-level down algorithm, traversal performs sequentially from level 4 to level 0. In the proposed algorithm, traversal starts from A, which is level 4, proceeds to B, which is level 2, and lastly to C, which is level 0. For the example in Fig. 4, this is a difference of five iteration steps (prior algorithm) compared to three (proposed algorithm).

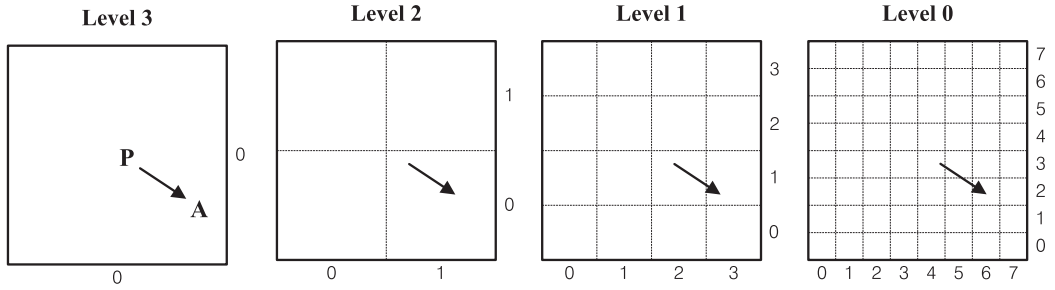

Fig. 3. An example of a start-level decision. In this example, the optimal start level is 2.

3.3. Selective and coherent level-up algorithms

We improve the previous level-up algorithms with two methods. The first method is the selective level-up algorithm, in which we also consider the intersection point. This makes it possible to ascend the level more effectively than [12], because the ascent is made only if empty-space maximizing can be achieved.

Fig. 5 (left) illustrates an example of the selective level-up method. If the current location is P, the intersection point P_1 is calculated first, using the ray and the height value 2 in the current node. Then, the node index difference between the coordinates of P and P_1 is calculated. If this difference is one, we visit the neighbor node at the same level instead of ascending one level, since the ray certainly pierces the neighbor node; otherwise, leveling up is determined by the same rules as [12]. In Fig. 5-(left), the selective level-up algorithm directly visits the neighbor node because the node index difference between P and P_1 is 1. We also apply a simple code-level optimization; Tevs's implementation [12] multiplies 0.5 by the node index and checks the digits below the decimal point of the calculated value. In contrast, we only take the and (&) operator to the least significant bit of the calculated value.

We next present the coherent level-up algorithm (Fig. 5 (right)). In this algorithm, in order to prevent unnecessary linear searches, this algorithm only ascends levels if consecutive node crossing occurs at the same level. Because this approach induces traversal at the same level whenever possible, it has advantages in terms of SIMD efficiency and memory access. Additionally, this method has very low overhead because it operates by referring a boolean value.

The two presented level-up algorithms have several pros and cons. The selective level-up algorithm can reduce the number of iteration steps, but it slightly increases the total calculation cost. Thus, it is suitable for our loop-based hardware pipeline architecture in Section 4. In contrast, the coherent level-up algorithm does not have a significant advantage in terms of the number of iteration steps. However, this method is more GPU-friendly because it is quite simple and increases coherence. The increased coherence may also increase cache hit rates and SIMD efficiency.

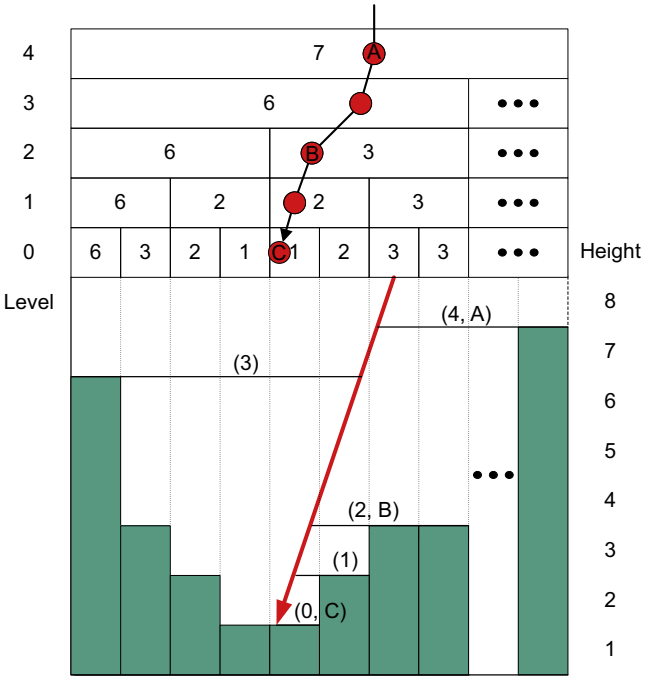

Fig. 4. An example of a two-level down traversal process.

4. Hardware architecture and implementation

This section describes our hardware architecture and its implementation, as well as the software implementation on a GPU.

4.1. Hardware architecture

Fig. 6 represents the proposed hardware architecture, which includes five main units (view calculation, traverse, texture/

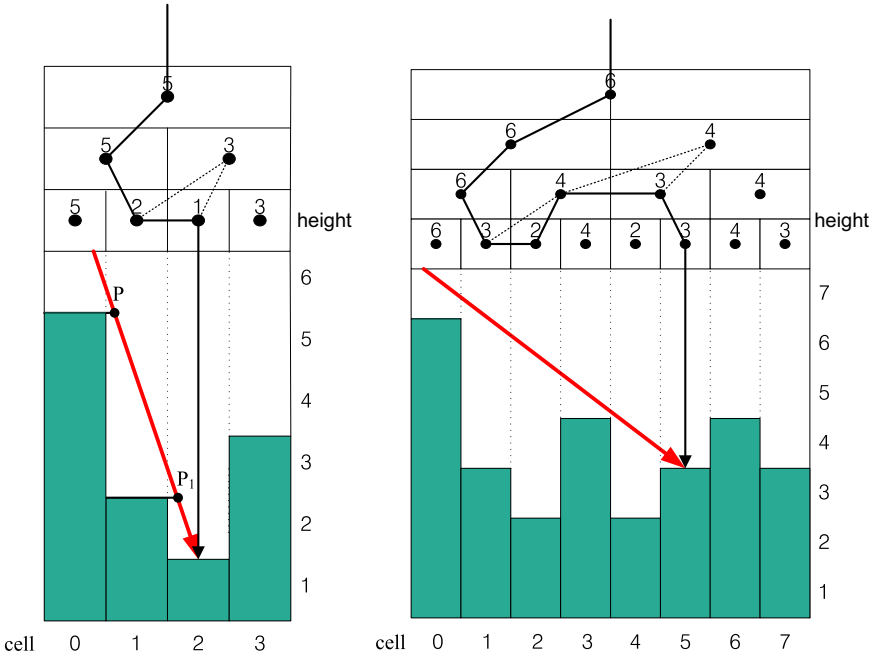

Fig. 5. Examples of the selective (left) and coherent (right) level-up algorithms. Dotted arrows indicate [12]. In the left example, the selective level-up algorithm removes unnecessary level ascending. In the right example, the coherent level-up algorithm reduces the occurrence of level-switching and achieves more coherent traversal.

normal address calculation, texture/normal read, and color calculation), the controller, two FIFOs, and two caches. A TBN (tangent–bitangent–normal) matrix, a view vector, and texture coordinates are input data. Each unit is pipelined and the controller manages loop operations.

The overall processing flow is as follows. The switch unit transmits the input data to the view-calculation unit when the controller allows. The view-calculation unit calculates the view vector on the tangent space by using the TBN matrix and the view vector on the world coordinate system for each ray; then its result is stored into FIFO #0. The traverse unit finds the intersection point between the view vector and the height map, then stores the result into FIFO #1. The texture/normal address-calculation unit calculates the addresses for both the texture and normal maps. The texture/normal read unit reads the texture and normal data using the calculated texture and normal addresses, respectively. Finally, the color-calculation unit calculates the color value using the texture and normal data.

The traverse unit is fully pipelined for high performance, and the other units are designed as semi-pipelined structures to share the hardware resources efficiently. Because this organization may lead to a load-balancing problem between the traverse unit and the other main units, we insert two input/output FIFO buffers to resolve the problem.

The main role of the controller is to prevent overflow of FIFO #0. If FIFO #0 is full, the controller commands the switch unit to not receive the data. As a result, the view-calculation unit is stalled to prevent the new data from being inputted into FIFO #0. If the color-calculation unit outputs a color value for a pixel, then the controller allows the switch unit to transmit new data, because there is an empty entry in two FIFOs.

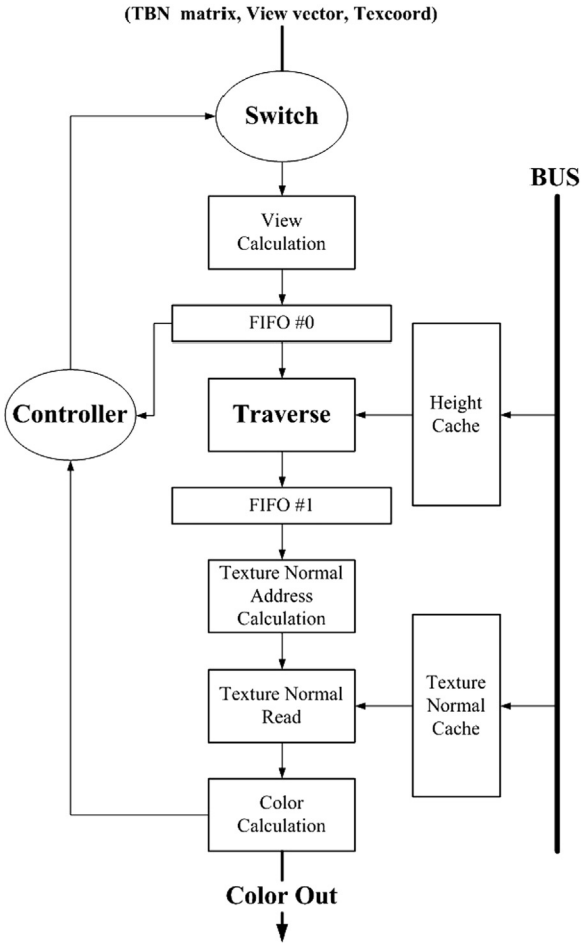

Fig. 6. Our proposed hardware architecture.

Because the traverse unit is fully pipelined, it accesses height data quite frequently. Thus, the traverse unit has an exclusive height cache. On the other hand, the unified texture/normal cache deals with texture and normal data access. Because the texture read stage has a semi-pipeline structure, one unified cache is more efficient than two separate caches.

The traverse unit processes iteratively, so its pipeline is in the form of a cascade; that is, the uppermost pipeline stage is processed after finishing the lowest pipeline stage. The height cache reduces the miss penalty dramatically by non-blocking processing; if the current cache requests reveal a miss, miss handling is performed and the subsequent pipeline stages are progressed simultaneously without stall. The process of the request that induces a cache miss is delayed to the next iteration.

4.2. Traverse pipeline

As shown in Fig. 7, the traverse unit consists of 15 pipeline stages. The processing flow behavior of Fig. 7 is fundamentally identical to that of Fig. 2. On the right side of Fig. 7, two-cycle latency is required for floating-point addition (Fadd) and floating-point division (Fdiv); in contrast, one-cycle latency is required for floating-point multiplication (Fmul) and floating-point comparison (Fcomp). We provide the numbers of floating-point arithmetic units, and integer arithmetic and cache access are performed during empty sections in the pipeline stages.

The processing flow of Fig. 7 is as follows. The pipeline has two modes. We first perform "Start-Level Calculation" with the first mode. Here, the end point A for input P is calculated at the hit-point calculation unit. In this mode, only the "Hit-Position Calculation" and "Start-Level Decision" parts on the left side are used. After that, the second mode is enabled for the iterative search. This mode uses all functions in Fig. 7 except for "Start-Level Decision". We first calculate the height address (HA) in which the height value of the current level is stored, and then read the data from the height cache. HA calculation requires two values from pipeline P15: the hit point of the current level and the current level value. After that, the hit point of the current level and the node-crossing position are calculated simultaneously. We then perform three tests (node crossing, multi-level down, and level-up) simultaneously in the NC/MLD/LU test unit.

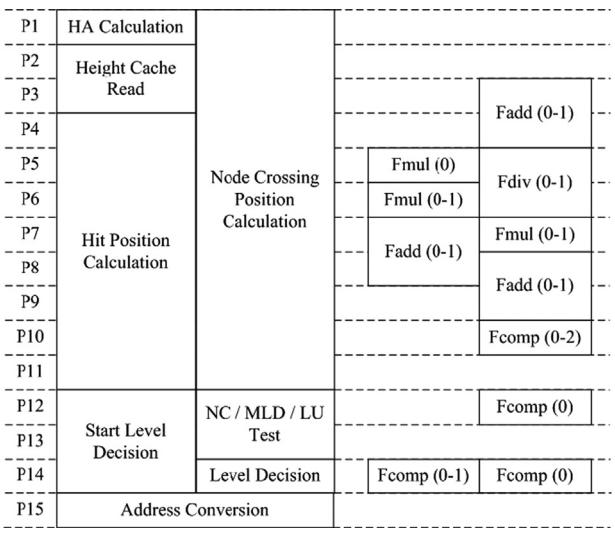

Fig. 7. The pipeline stages of the traverse unit. Left: the location of the functions. Right: the layout of the floating-point arithmetic units to handle these functions.

If node crossing occurs, the node crossing position is transferred to the level-decision unit; otherwise, the hit point of the current level is transferred to the address-conversion unit. In the level-decision unit, the next level is determined according to the results of the three tests, and then transferred to the address-conversion unit. The address-conversion unit converts the inputted floating-point texture coordinate into an integer texture address and simultaneously checks whether the decided next level reaches the termination condition. If true, the current texture coordinate is outputted as the final result of the hit point; otherwise, the converted texture address and the decided level are forwarded into pipeline P1.

4.3. Hardware implementation

For hardware implementation, we used the Dynalith Systems iNEXT-V6 board, which contains four Xilinx Virtex-6 LX550 FPGA chips, 2 GB of DDR2 memory, and 8 MB of SRAM. A TFT LCD board with 800×480 screen resolution is attached to the iNEXT-V6 board.

We integrated our displacement hardware unit into the existing ray-tracing hardware architecture, RayCore [22], which is the upgraded version of WRTX [21]. Our architecture is implemented on each FPGA chip and operates at a speed of 84 MHz. In each chip, the 64-bit bus at 84 MHz is used to access external memory. Table 1 shows the list of hardware resources for each unit. We use a 24-bit floating-point format (1 sign bit, 7 exponent bits, and 16 fraction bits). Our design occupies approximately 67% of the FPGA's logic cells and 25% of the FPGA's memory resources.

4.4. GPU implementation

To evaluate the GPU performance of the presented algorithm, we used DirectX 11 and Shader Model 5.0. The benchmark environment is the AMD DetailTessellation11 sample in the DirectX SDK. We added the quad-tree displacement-mapping shader code in [15] and the maximum mipmap generation into the benchmark source code. We disabled the level-up code in [15] making this implementation fundamentally identical with [10]. To implement Tevs et al.'s level-up method [12] in our experimental environment, we referred to their GLSL code. The implementation of our algorithms was described in Section 3.

5. Experimental results

In Sections 5.1–5.3, we describe platform-independent results. After that, we describe and analyze the results of the proposed algorithm on a GPU and that of the proposed hardware architecture on an FPGA and an ASIC. Finally, we describe the suitability of our hardware architecture for mobile GPUs. We used a screen resolution of 1024×768 for Sections 5.1–5.4, and 800×480 for Section 5.5. We used seven scenes in the DetailTessellation11 sample (Fig. 8). The Stones and Rocks scenes, respectively, have the highest and the lowest coherence between adjacent height maps among the scenes. The resolution of all height maps is 256×256. We also used three angles in Fig. 9 to measure performance variation by different angles.

Table 1
Hardware complexity: the number of floating-point arithmetic elements per FPGA chip.

	2input adder	3input adder	Comparator	Multiplier	Divider	Square root
View calc.	–	1	–	3	1	–
Traverse	6	–	7	5	2	–
Addr. calc.	1	–	–	1	–	–
Color calc.	1	1	–	3	1	1
Total	8	2	7	12	4	1

5.1. Probability of node crossing

Table 2 shows the probability of node crossing during the traversal of the image pyramid for [10,12]. The experiment found that the average probability of node crossing at front angles was only 3–22% for both the algorithms. This result means a high probability of leveling down during traversal at the front angles, so the start-level decision and multi-level down algorithms make good use of this tendency. Furthermore, the level-up method [12] reduces the frequency of node crossing, so it can increase performance at grazing angles.

5.2. The number of iteration steps of the multi-level down algorithm

Table 3 shows the number of iteration steps with different level-down values (n). Neither the start-level decision nor selective/coherent level-up algorithms were applied. Note that the n value of one is identical to [10]. The n value of two reduces the number of iteration steps at all angles more than the n value of one, so we select the n value of two in further experiments.

5.3. Comparison of the number of iteration steps

In Table 4, we compare the number of iteration steps between previous algorithms [10,12] and our algorithms. In front angles, the start-level decision and multi-level down algorithms considerably reduce the number of iteration steps. In grazing angles, the selective level-up algorithm more effectively reduces the number of steps than [12], and the number of steps of the coherent level-up algorithm is as seen in the results of [12]. When we use the start-level decision, multi-level down, and the selective level-up algorithms together, the number of iteration steps is reduced by 16–65% and 5–66% compared to [10] and [12], respectively. In our loop-based H/W architecture, the number of iteration steps directly affects the overall performance.

5.4. Performance evaluation on a GPU

To evaluate the GPU rendering performance of the presented algorithms, we used the following hardware: a 3.3 GHz Intel Core i5-2500 quad-core CPU, 8 GB of DDR SDRAM, and an nVIDIA Geforce GTX460 GPU.

Table 5 describes the results on the GPU. As seen in the results from the number of iteration steps in Section 5.3, the start-level calculation and multi-level down algorithms achieve performance improvements at front angles. However, at grazing angles, the coherent level-up algorithm is more beneficial than the selective level-up algorithm and Tevs et al.'s algorithm [12]. The reason that the coherent level-up algorithm is more GPU-friendly than other algorithms is its simplicity and coherence.

When we use the start-level decision, multi-level down, and coherent level-up algorithms together, we achieve performance improvements over [10,12] by up to 85.4% (Four Shapes at the grazing angle) and 56.4% (Stones at the front angle), respectively. The average performance improvements over [10] and [12] are 21.8% and 15.8%, respectively. Fig. 10 shows the relative performance of [12] and the mixed use of our approaches compared to [10]. Tevs et al.'s work [12] shows much better performance at the grazing angle than [10], but it decreases performance at the front angle by approximately 30% due to its overheads and unnecessary level ascending. In contrast, our approach increases performance in the majority of cases. This result means that our approach,

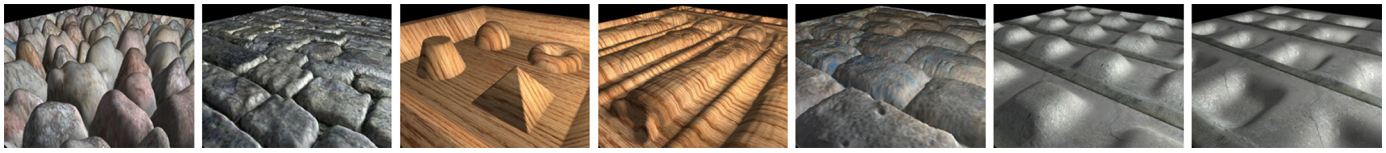

Fig. 8. Benchmark scenes: Rocks, Stones, Four Shapes, Saints, Wall, Bump, and Dent.

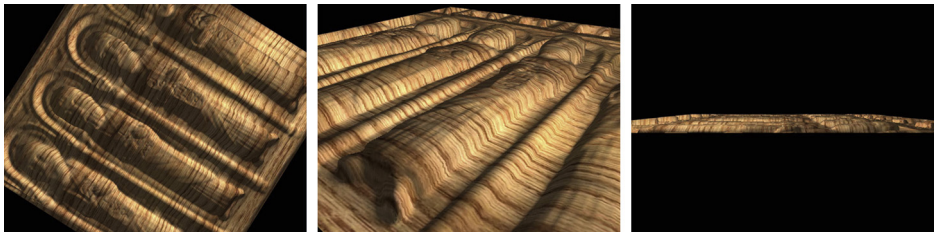

Fig. 9. Rendered images of three angles: a front angle (left), an oblique angle (middle), and a grazing angle (right).

Table 2
Probability of node crossing (%).

	Front		Oblique		Grazing	
	[10]	[12]	[10]	[12]	[10]	[12]
Rocks	22.4	19.2	54.5	36.9	76.4	47.9
Stones	3.8	3.7	17.9	15.0	51.5	36.1
Four shapes	8.5	8.1	39.7	28.3	58.1	37.9
Saints	5.9	5.5	27.5	21.6	62.1	39.8
Wall	5.1	4.9	21.0	16.6	52.1	35.9
Bump	7.6	7.3	38.2	27.5	72.5	47.3
Dent	5.1	4.9	23.5	18.3	38.5	26.9

Table 3
Comparison of the number of iteration steps for different level-down values (n) in the Rocks scene.

n	Front	Oblique	Grazing
1	11.5	19.7	37.9
2	8.9	18.2	37.8
3	8.4	18.2	38
4	9.6	19.1	42.7

unlike previous algorithms, is robust for angle changes; this feature is important in interactive 3D graphics.

We also performed an additional experiment in the Four Shapes scene by uniformly distributing a set of view samples at the hemisphere. In this experiment, we increased camera angles 10 degrees at a time, from 0 to 180 degrees. As in the previous results, the result in Fig. 11 confirms again that our approach is more robust for various camera angles than previous algorithms.

According to Table 5, the mixed use of the presented algorithms does not always achieve peak performance. The reason is that the start-level calculation and multi-level down algorithms and the coherent level-up algorithm have different characteristics from one another. The calculation cost of the former algorithms would be useless at grazing angles, and the coherent level-up algorithm would negatively affect the performance at front angles. However, these negative effects only cause a small amount of performance degradation. Therefore, the mixed use of the presented algorithms is advantageous in terms of average performance.

5.5. Performance evaluation of our H/W architecture

This section describes the performance evaluation results of our dedicated hardware architecture. We used the BART Kitchen scene with 110 K triangles [23] (Fig. 12) to show how our displacement-mapping units are integrated as part of the RayCore system. We combined four displacement maps (Rocks, Stones, Four Shapes, and Saints) into a single 512×512 texture and mapped it to the floor of the kitchen. We used two different settings of the maximum ray recursion depth: 0 (ray casting) and 10 (Whitted ray tracing with reflection and refraction). Note that the last setting requires the tracing of approximately three times more rays for reflection and refraction. We also measured the performance of the four scenes with each displacement map. In this experiment, we only used an oblique angle.

Table 6 summarizes the results on the FPGA board. This table includes cache hit rates, memory traffic, shading unit utilization, FPS, and performance (Mpixels/s). The displacement-mapping performance on FPGA is 4.8–9.8 Mpixels/s at 84 MHz. Because the Kitchen scene is more complex than the other four scenes, the Kitchen scene shows lower throughput than the others. The memory traffic in the worst case (Kitchen with the ray depth 10) is quite low: 115 MB/s for 4.8 M pixels.

For the ASIC evaluation, we used TSMC's 28-nm high-performance, low-power process and the Synopsis design compiler. The presented displacement-mapping unit was synthesized up to 650 MHz with a voltage of 1.0 V, so we set the target frequency to 500 MHz with some margin. The die size of four displacement-mapping units is approximately 0.8 mm^2. The internal power consumption of the displacement-mapping unit, derived from the Synopsis design compiler, is 0.3 W including that of caches and the AXI bus interface.

Table 7 summarizes our GPU, FPGA, ASIC experimental results in the Rocks scene. As with [24], the ASIC performance is linearly scaled up proportional to the clock frequency because the memory traffic is not a bottleneck; in all experiments, the required memory bandwidth is less than 0.7 GB/s at 500 MHz, which is much lower than the peak memory transfer rates of the current desktop GDDR5 memory (up to 288.4 GB/s) and the current mobile LPDDR3 memory (14.9 GB/s). Consequently, the ASIC version of our architecture can provide sufficient performance for real-time rendering (50 FPS at HD 720p) with only four small units.

Table 4
Comparison of the number of iteration steps per pixel. Bold values indicate the lowest step (best case). Abbreviations: F – front angles, O – oblique angles, and G – grazing angles.

	Rocks			Stones			Four shapes			Saints			Wall			Bump			Dent		
	F	O	G	F	O	G	F	O	G	F	O	G	F	O	G	F	O	G	F	O	G
[10]	11.6	19.7	37.9	9.3	10.9	18.4	9.8	14.9	21.3	9.5	12.4	23.6	9.4	11.3	18.7	8.6	12.9	28.9	8.4	10.4	12.7
[12]	12.6	19.6	30.2	9.5	11.5	19.2	10.2	15.6	20.0	9.8	13.2	20.7	9.7	11.8	19.2	8.9	13.4	25.3	8.6	19.8	12.7
(a) Start-level calculation	9.5	19.3	37.9	4.4	8.4	18.0	7.4	14.3	21.3	6.6	11.4	23.3	5.2	9.4	18.3	8.5	12.9	28.9	8.3	10.4	12.7
(b) Multi-level down	8.9	18.2	37.8	5.6	7.8	16.4	6.9	13.8	21.6	6.1	10.1	23.2	4.1	8.0	23.0	6.2	12.9	36.6	5.5	8.0	11.0
(c1) Selective level-up	11.6	18.4	29.2	9.3	11.0	18.2	9.8	14.4	19.7	9.5	12.3	20.0	9.4	11.3	18.5	8.6	12.3	24.7	8.4	10.3	12.4
(c2) Coherent level-up	12.1	19.8	31.6	9.3	11.3	19.3	10.0	15.4	20.6	9.6	12.9	21.4	9.5	11.7	19.3	8.7	13.2	26.1	8.4	10.7	13.0
Mixed use of (a+b+c1)	**8.0**	**16.5**	**28.3**	**3.2**	**6.6**	**15.8**	**5.6**	**12.6**	**17.9**	**4.7**	**9.5**	**18.8**	**3.9**	**6.6**	**15.1**	**6.1**	**11.4**	**24.1**	**5.5**	**7.8**	**10.3**

Table 5
Comparison of rendering performance on a GPU (frames per second). Bold values indicate the highest performance.

	Rocks			Stones			Four shapes			Saints			Wall			Bump			Dent		
	F	O	G	F	O	G	F	O	G	F	O	G	F	O	G	F	O	G	F	O	G
[10]	458	185	273	583	428	609	563	293	448	567	333	429	579	386	593	610	224	308	633	337	651
[12]	356	204	463	462	353	607	452	293	817	454	312	669	462	336	610	478	285	539	504	334	930
(a) Start-level calculation	470	182	270	718	452	603	602	287	446	622	328	425	686	394	588	559	221	308	579	326	647
(b) Multi-level down	**482**	177	258	717	**466**	568	**640**	276	417	**658**	325	380	**717**	**409**	563	**660**	207	275	**724**	325	603
(c1) Selective level-up	369	197	436	475	354	586	462	293	777	463	315	632	471	333	583	497	286	505	515	352	891
(c2) Coherent level-up	411	**220**	**485**	555	398	**634**	529	**325**	**865**	540	**346**	**699**	548	373	**639**	574	**313**	**565**	598	**392**	**988**
Mixed use of (a+b+c2)	431	205	443	**723**	435	611	606	313	831	641	**346**	659	708	400	617	580	298	535	632	385	959

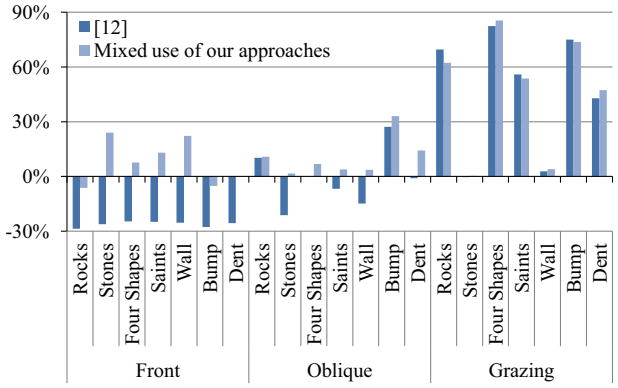

Fig. 10. Performance improvements of [12] and our approach over [10]. The result is obtained from Table 5.

5.6. Suitability for mobile GPUs

Mobile devices have different characteristics than do laptop or desktop GPUs. Thus, the most important design criteria for mobile graphics hardware are performance per watt (power efficiency) and performance per square millimeter (area efficiency) [25]. In addition, reducing off-chip memory accesses is important for power efficiency and high performance [25].

According to the results on the ASIC evaluation, our hardware architecture satisfies these criteria: small chip area (0.8 mm^2), low power consumption (0.3 W), and low off-chip memory accesses (0.03 GB/s for simple scenes and 0.3–0.7 GB/s for complex scenes). Therefore, the proposed hardware unit has high potential utility as an additional fixed unit on existing mobile GPUs.

6. Conclusion and Future Work

In this paper, we proposed a set of improvements for pyramidal displacement mapping. As a result, the number of iteration steps decreased by up to 66% compared to the previous algorithms. We also proposed a hardware architecture using the algorithm and integrated it into the existing ray-tracing hardware [22]. With the FPGA and ASIC evaluations, we showed promising results of real-time inverse displacement mapping.

The main advantage of our approach is its wide applicability. Our algorithm can be easily added into the existing pyramidal displacement mapping. Thus, it is useful for existing real-time 3D graphics applications using GPUs. Our architecture is especially valuable for mobile 3D graphics. Even though high-quality 3D graphics on mobile devices are increasingly important, current mobile GPUs provide neither sufficient shader performance for real-time inverse displacement mapping nor DirectX11 tessellation [3]. Thus, many mobile 3D graphics applications limit the triangle count and sacrifice the details. Our architecture can resolve this problem with real-time inverse displacement mapping.

In future studies, we would like to extend our approach to ray casting for terrain rendering [16] and to combine our approach with other concepts in [12,15,26,27] such as level-of-detail, height blending, and silhouette processing. Finally, an ASIC chip of the entire ray-tracing system with the displacement-mapping unit and its full paper will be announced in the near future.

Acknowledgments

This work was supported by Siliconarts and in part by the National Research Foundation of Korea Grant funded by the

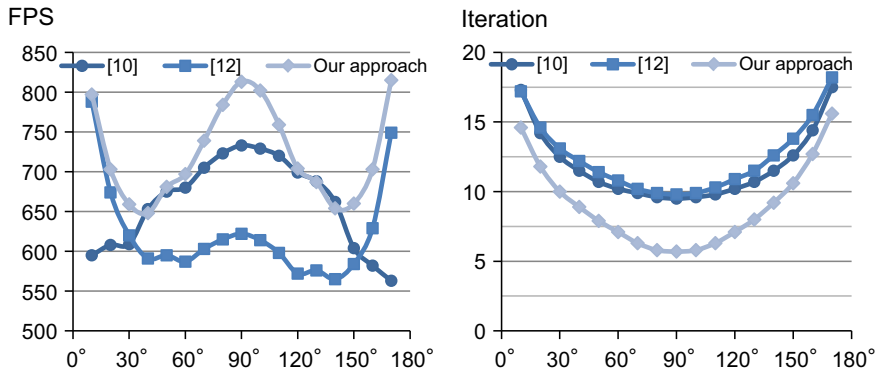

Fig. 11. An experimental result in the Four Shapes scene with uniformly distributed view samples at the hemisphere: FPS (left) and the number of iteration steps (right).

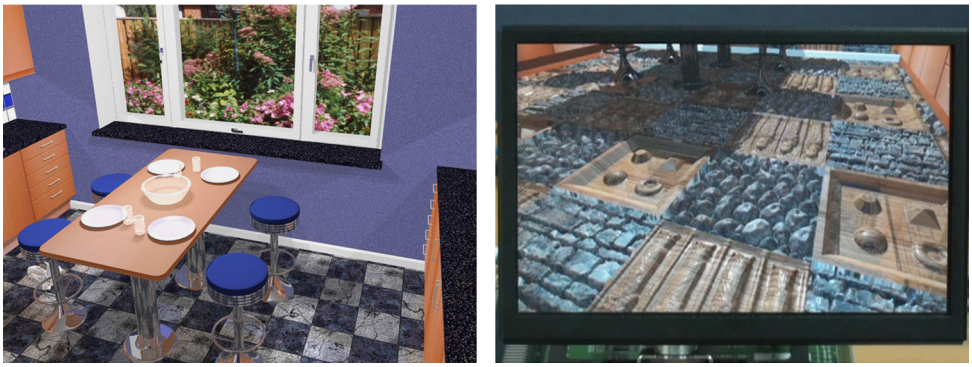

Fig. 12. Two images of the Kitchen scene rendered by a S/W ray tracer [23] (left) and our H/W ray tracer (right). In the right image, we used a floor-emphasized viewpoint for displacement mapping.

Table 6
Experimental results on the FPGA.

	Kitchen		Shapes			
	depth 0	depth 10	Rocks	Stones	Four	Saints
Cache hit rate (%)	96	91	99	99	99	99
Mem traffic (MB/s)	47.3	115.8	4.7	3.9	4.4	4.4
Shading unit utilization (%)	76	74	94	94	92	92
Frames per second	15	6	22	22	22	21
Mpixels/s	5.4	4.8	7.5	9.8	9.0	7.5

Table 7
Comparison of different platforms.

	GPU (GTX460)	FPGA (Virtex 6)	ASIC
Mpixels/s	149.9	7.5	44.6
Clock (MHz)	750 (Core) 1500 (Shader)	84	500
Number of cores	336	4	4
Process (nm)	40	–	28
Area (mm^2)	332	–	0.8
Power consumption (W)	160 (TDP)	–	0.3

Korean Government (Ministry of Education, Science and Technology) [NRF-2012R1A6A3A03040332], Army Research Office, National Science Foundation, and Intel. The CAD tools were supported by IDEC. We thank Art Tevs for sharing his GLSL code.

Appendix A. Supplementary material

Supplementary data associated with this article can be found in the online version of http://dx.doi.org/10.1016/j.cag.2013.10.024.

References

[1] Hirche J, Ehlert A, Guthe S, Doggett M. Hardware accelerated per-pixel displacement mapping. In: Proceedings of graphics interface; 2004. p. 153–8.
[2] Blythe D. The Direct3D 10 system. ACM Trans Graph (TOG) 2006;25 (3):724–34.
[3] Gee K. Direct3D 11 tessellation. In: Microsoft Gamefest 2008; 2008.
[4] Patterson JW, Hoggar SG, Logie JR. Inverse displacement mapping. Comput Graph Forum 1991;10(2):129–39.
[5] Kaneko T, Takahei T, Inami M, Kawakami N, Yanagida Y, Maeda T, et al. Detailed shape representation with parallax mapping. In: Proceedings of the international conference on artificial reality and telexistence; 2001. p. 205–8.
[6] Brawley Z, Tatarchuk N. Parallax occlusion mapping: self-shadowing, perspective-correct bump mapping using reverse height map tracing. In: Shader X^3, Cengage learning; 2005. p. 135–54 [chapter 2.5].
[7] Tatarchuk N. Dynamic parallax occlusion mapping with approximate soft shadows. In: Proceedings of i3D, ACM; 2006. p. 63–9.
[8] Policarpo F, Oliveira MM, Comba JLD. Real-time relief mapping on arbitrary polygonal surfaces. In: Proceedings of i3D, ACM; 2005. p. 155–62.
[9] Policarpo F, Oliveira MM. Relief mapping of non-height-field surface details. In: Proceedings of i3D, ACM; 2006. p. 55–62.
[10] Oh K, Ki H, Lee C-H. Pyramidal displacement mapping: a GPU based artifacts-free ray tracing through an image pyramid. In: Proceedings of virtual reality software and technology, ACM; 2006. p. 75–82.
[11] Schroders MFA, Gulik RV. Quadtree relief mapping. In: Proceedings of graphics hardware; 2006. p. 61–6.
[12] Tevs A, Ihrke I, Seidel H-P. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. In: Proceedings of i3D, ACM; 2008. p. 183–90.
[13] Szirmay-Kalos L, Umenhoffer T. Displacement mapping on the GPU—state of the art. Comput Graph Forum 2008:1567–92.
[14] Cohen D, Shaked A. Photo-realistic imaging of digital terrain. Comput Graph Forum 1993;12(3):363–73.
[15] Drobot M. Quadtree displacement mapping with height blending. In: GPU Pro 2, AK Peters; 2010. p. 117–48 [chapter 3].
[16] Dick C, Krüger J, Westermann R. GPU ray-casting for scalable terrain rendering. In: Proceedings of the eurographics symposium on rendering; 2009. p. 43–50.
[17] Policarpo F, Oliveira MM. Relaxed cone stepping for relief mapping. In: GPU Gems 3. Addison Wesley; 2007. p. 409–28 [chapter 18].
[18] Donnelly W. Per-pixel displacement mapping with distance functions. In: GPU Gems 2. Addison Wesley; 2005. p. 123–36 [chapter 8].
[19] Baboud L, Décoret X. Rendering geometry with relief textures. In: Proceedings of graphics interface; 2006. p. 195–201.

[20] Jeschke S, Mantler S, Wimmer M. Interactive smooth and curved shell mapping. In: Proceedings of the eurographics symposium on rendering; 2007. p. 351–60.
[21] Park W, Nah J, Park J, Lee K, Kim D, Kim S, et al. An FPGA implementation of Whitted-style ray tracing accelerator. In: Proceedings of the IEEE symposium on interactive ray tracing; 2008. p. 187.
[22] Siliconarts, Raycore series 1000. Technical Report. ⟨http://www.siliconarts.co.kr/gpu-ip⟩; 2012.
[23] Lext J, Assarsson U, Möller T. A benchmark for animated ray tracing. IEEE Comput Graph Appl 2001;21:22–31.
[24] Woop S, Brunvand E, Slusallek P. Estimating performance of a ray-tracing ASIC design. In: IEEE symposium on interactive ray tracing 2006; 2006. p. 7–14.
[25] NVIDIA. Whitepaper: NVIDIA Tegra 4 family GPU architecture. Technical Report; 2013.
[26] Wang L, Wang X, Tong X, Lin S, Hu S, Guo B, et al. View-dependent displacement mapping. ACM Trans Graph 2003;22(3):334–9.
[27] Wang X, Tong X, Lin S, Hu S, Guo B, Shum H-Y. Generalized displacement maps. In: Proceedings of the eurographics symposium on rendering; 2004. p. 227–33.