A Mobile 3D Display Processor with A Bandwidth-Saving Subdivider

Seok-Hoon Kim, Sung-Eui Yoon, Sang-Hye Chung, Young-Jun Kim, Hong-Yun Kim, Kyusik Chung, and Lee-Sup Kim

Abstract— **A mobile 3D display processor with a subdivider is presented for higher visual quality on handhelds. By combining a subdivision technique with a 3D display, the processor can support viewers see realistic smooth surfaces in the air. However, both the subdivision and the 3D display processes require a high number of memory operations to mobile memory architecture. Therefore, we make efforts to save the bandwidth between the processor and off-chip memory. In the subdivider, we propose a re-computing based depth-first scheme that has much smaller working set than prior works. The proposed scheme achieves about 100:1 bandwidth reduction over the prior subdivision methods. Also the designed 3D display engine reduces the bandwidth to 27% by reordering the operation sequence of the 3D display process. This bandwidth saving translates into reductions of off-chip access energy and time. Consequently the overall bandwidth of both the subdivision and the 3D display processes is affordable to a commercial mobile bus. In addition to saving bandwidth, our work provides enough visual quality and performance. Overall the 3D display engine achieves 325fps for 480×320 display resolution.**

***Index Terms*—computer graphics, microprocessors, multimedia systems, three-dimensional displays**

I. Introduction

During the past few years, a cellular phone has rapidly evolved into a powerful entertainment tool, which can be used as not only a phone, but also a video player, a TV, a game console, a camera, a navigation, etc. As many contents in different platforms require high-quality visual appearances, visual quality becomes a key differentiating factor among mobile devices. In order to make various and realistic effects, modern handhelds use two key techniques: 3D graphics and 3D display.

In modern 3D graphics, subdivision surfaces have received significant attentions, because they can support smooth surfaces, leading to high-quality rendering. Owing to the smoothness, the subdivision surface is now adopted as a representation in the latest 3D graphics API such as Direct3D 11 [1]. Together with 3D graphics feature, a 3D display is used in more and more handhelds, because it can give users immersive feeling, which has not experienced before. In handhelds, an auto-stereoscopic type display is used as a 3D display, because it does not require additional peripherals such as glasses or head gear [3].

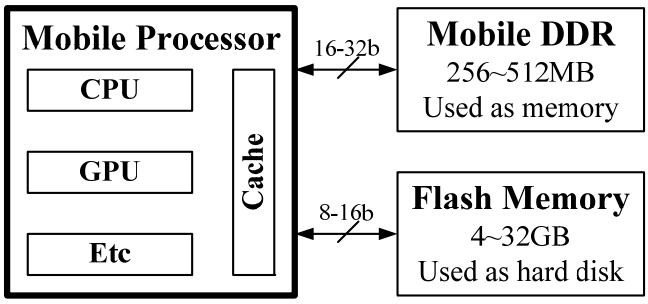


Fig. 1 The memory architecture of handhelds is different from the memory architecture of PCs. All the processors share the same bus and a unified memory which is outside the chip.

Although the subdivision surfaces and the 3D display provide realistic experiences to users, even high-end mobile phones (e.g., iPhone) cannot support them in real-time. This is mainly because the memory architecture of handhelds is different from that of desktop PCs [2] as shown in Fig. 1, whereas both the subdivision and the 3D display processes have many memory operations. In handhelds, many heterogeneous processors (e.g., a GPU and a CPU) share a unified memory through the same bus because of small physical size and limited power, where the memory is outside the chip. This memory architecture causes frequent off-chip memory accesses and bus traffic jams, significantly under-utilizing the computing power of mobile processors.

This paper proposes a mobile 3D display processor with a bandwidth-saving subdivider that provides high visual quality on handhelds. In order to run both the subdivision and the 3D display processes given the mobile memory architecture, we enormously reduce the bandwidth requirement for both the subdivision and the 3D display processes.

In the subdivider, we propose a re-computing based depth-first scheme. Most prior approaches perform subdivision

Manuscript received October 9, 2001.

S.-H. Kim and K. Chung were with Department of Electrical Engineering, Korea Advanced Institute of Science & Technology, Daejeon, 305-701 Republic of Korea. They are now with Samsung Electronics, Giheung, Gyunggi, Republic of Korea (e-mail: hool84@mvlsi.kaist.ac.kr).

S.-H. Chung, Y.-J. Kim, H.-Y. Kim, and L.-S. Kim are with Department of Electrical Engineering, Korea Advanced Institute of Science & Technology, Daejeon, 305-701 Republic of Korea.

S.-E. Yoon is with Division of Web Science and Technology and Department of Computer Science, Korea Advanced Institute of Science & Technology, Daejeon, 305-701 Republic of Korea.

for all the input faces in parallel using breadth-first scheme [4][5]. Although they are optimized for the parallel architecture of modern GPUs employed in desktop PCs, they need more than several Gbytes for the bus bandwidth whereas the fastest mobile bus supports up to 625Mbytes/s [6]. The proposed depth-first scheme executes subdivision for just a single target face with its one-ring neighborhood, instead of for all the input faces. Since the working set of our depth-first scheme is small with a high data coherency, our method requires a low data bandwidth between the subdivider and off-chip memory, by utilizing a small amount of on-chip memory (e.g., 20KB SRAMs). In order to further reduce the bandwidth requirement, we propose a compact edge-less data structure that is optimized for our subdivision algorithm. The implemented subdivider has less than 1% bandwidth requirement of prior subdivision methods [4][5]. In addition, we propose adaptive subdivision scheme which can control refinement level according to depth, the distance from a view-position. Together with bandwidth saving effects, our subdivider provides enough subdivision quality compared with the latest prior work [4] that targets on desktop PCs. To the best of our knowledge, our work is the first attempt for a mobile subdivider.

Also the proposed 3D display engine reduces the bandwidth requirement by changing operation sequence. The 3D display process for an auto-stereoscopic display needs to generate many intermediate images from a pair of stereo images by interpolation. Our processor produces them in an interleaving order instead of serial order, which makes the intermediates not be stored in off-chip memory. As a result, the bandwidth requirement of the 3D display process is reduced by 73% compared with a typical approach.

The processor is implemented within 4.5mm×4.5mm die using 0.13μm CMOS technology. It integrates 965K gates, runs at 50MHz, and consumes 140mW at 1.2V.

The remainder of this paper is arranged as follows. Section II briefly describes the background of subdivision surfaces and 3D display process, and Section III overviews the processor. Section IV and VI explain the proposed ideas in the order of a subdivider, a 3D graphics engine, and a 3D display engine. Then, Section VII explains how this work saves power consumption, and Section VIII shows the improvements of this work compared with previous work. Section IX summarizes and concludes this paper.

II. Background, Prior Work, and Problem

A. Catmull-Clark Subdivision Surfaces

Many tessellation schemes have been proposed, and they are classified as subdivision surfaces (e.g., Catmull-Clark subdivision surfaces [7] and Loop subdivision surfaces [8]) and parametric surfaces (e.g., NURBS and Bézier patches [9]). Parametric surfaces are computed by higher-order equations and control points, which do not tessellate geometry vertices itself but the parameters of surfaces such as u and v. Higher-order equations use the tessellated parameters as variables and produce the vertices of smooth surfaces. On the other hand, subdivision surfaces directly tessellate geometry vertices without any parameters. It refines an input mesh (LOD_k) by given equations, producing an output mesh (LOD_{k+1}) that is finer than the input mesh. Then, the output mesh (LOD_{k+1}) is again used as an input mesh to the next tessellation level (LOD_{k+2}). By performing this process iteratively, we can construct a series of finer meshes that converge to the limit surface of the subdivision surfaces.

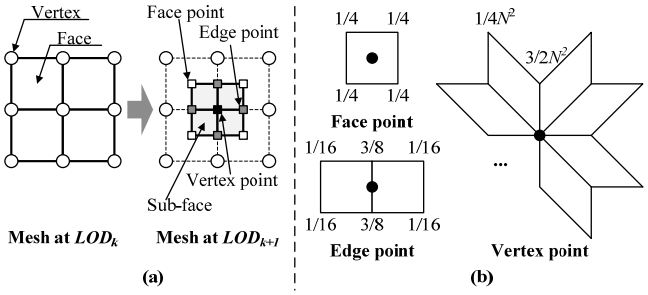


Fig. 2 (a) Catmull-Clark subdivision refines a face with *N* edges into *N* sub-faces by creating three kinds of points; a face point, an edge point, and a vertex point. (b) Each point is computed by the weighted sum of neighbor vertices. In the vertex point, *N* means a valence number that is the number of edges connected to the vertex under the subdivision.

Among those surfaces, the Catmull-Clark Subdivision Surface (CCSS) has been the standard modeling tool for various applications because of the following two reasons: 1) the CCSS can be applied to two-manifold meshes that have arbitrary topologies without any continuity limitation and 2) geometry data (e.g., vertices) of the CCSS are freely modified without introducing artifacts.

In the CCSS, a face with N edges is subdivided into N faces at every subdivision level. At each subdivision level, we create three kinds of points: face, edge, and vertex points as shown in Fig. 2. A face point for a face is computed by averaging the vertices of the face. Then, an edge point for an edge is computed by averaging two end-vertices of the edge and two face points of the faces sharing the edge. We also create a vertex point for each vertex by computing a weighted sum of the vertex, all the face, and edge points around the vertex. We then construct sub-faces by connecting these points. By performing this subdivision process iteratively, we can construct a series of finer meshes that converge to the limit surface of the CCSS.

Typically, an output mesh computed at a subdivision level is used as an input mesh to the next subdivision level. During the subdivision process, we must utilize an efficient data structure to find and update the connectivity information of CCSSs. These operations are performed by various indexing operations [10][11]. However, a naive data structure for CCSSs requires a large amount of storage and a wide bus bandwidth.

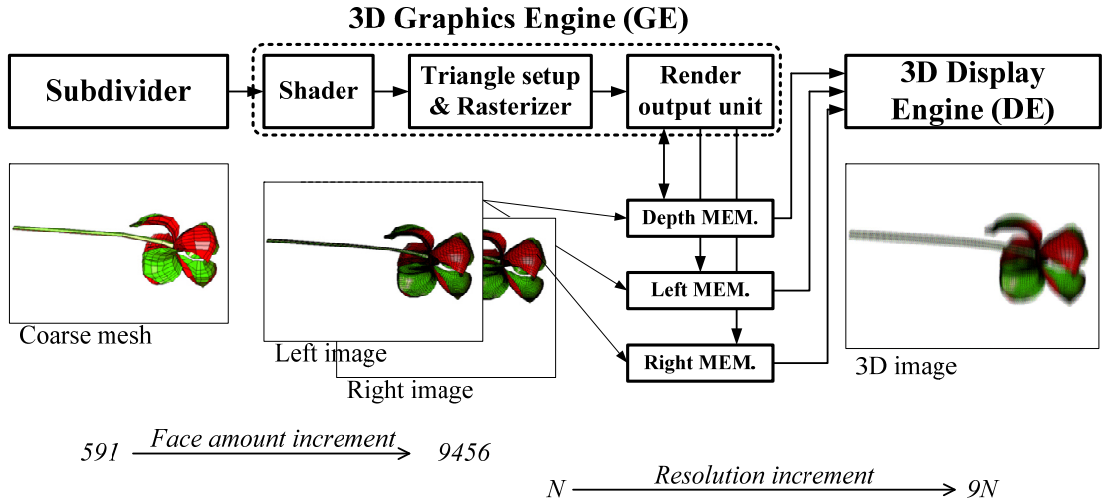


Fig. 3 This figure describes the overall architecture of our processor. A subdivider makes smooth surfaces from coarse input mesh, then a 3D graphics engine renders a pair of stereo images and a depth-map, and finally a 3D display engine synthesizes a 3D image from the output data of the 3D graphics pipeline.

B. Subdivision Related Works

Most prior methods of CCSSs are designed for GPU-friendly parallel subdivisions [4][5][11]. Bunnell [5] and Shiue et al. [11] perform the subdivision of CCSSs directly on a GPU. They store meshes into textures and employ multi-pass rendering for subdivision operations. They need to perform at least one subdivision level on a CPU to treat the extraordinary vertices of initial meshes. This multi-pass rendering and the subdivision on the CPU can require a lot of off-chip memory accesses in the mobile memory architecture. Recently, Patney et al. [4] proposes a robust adaptive subdivision for CCSSs on a highly parallel GPU. This method achieves a high performance on the GPU, by employing the breadth-first subdivision scheme that performs the subdivision process for all the faces of the mesh in parallel. However, it is designed mainly for desktop PCs and is not suitable for handhelds. Since the breadth-first scheme in the parallel method requires a large working set, it can cause a significant amount of off-chip memory accesses in the mobile memory architecture.

Our previous work [12] accomplished tessellation on mobile platforms using parametric surfaces instead of subdivision surfaces. It quickly tessellates parameters, (u, v), using fast difference, and accelerates the computations for higher-order equation using dual-core shaders. Although the previous work enables to run tessellation on mobile devices using small bandwidth requirement, it directly inherits the parametric surfaces' limitations such as piecewise continuity and inconvenient topology control. Also, it executes adaptive tessellation by computing LODs per object, which makes different triangles have the same LOD. Its dual-core shaders provide high computing power, but consume more than 200mW, which is still high in the mobile platform.

C. 3D Display Process

An auto-stereoscopic display refracts the light from LCD pixels using lens array or parallax barrier attached on an LCD and delivers different images to each eye of a viewer, giving depth-perception to the viewer. For a viewer to observe different images depending on his eye position (view-position), the auto-stereoscopic display prepares multiple images taken from different view-positions and mixes them according to the lens array pattern or the barrier pattern. If an auto-stereoscopic display mixes nine images, it is called as a 9-view auto-stereoscopic display.

The 3D display process for the auto-stereoscopic display consists of a View Interpolation Process (VIP) and a Multiplexing Process (MP). The VIP produces intermediate images for in-between view-positions from a pair of stereo images and a depth-map, which relieves the burden of rendering all the images. Then, the MP allocates the sub-pixels of the stereo images and the intermediates into the sub-pixels of the auto-stereoscopic display according to the lens array pattern, synthesizing a 3D image.

D. Mobile Memory Architecture

The computing power of modern handhelds is enough to perform computation-intensive operations including the evaluations of CCSSs. However, handhelds have a quite different memory architecture compared to that of desktop PCs as shown in Fig. 1 [2]. In handhelds, a memory located outside the chip is shared between different processors, and the widths of bus and memory are limited within 32 or 16 bits. Thus the bus becomes the main performance bottleneck in the chip, which significantly underutilizes the high computing power of recent mobile GPUs and other processors. Furthermore, frequent memory access increases off-chip memory access time.

While an on-chip memory access takes only a single cycle, an off-chip memory access takes tens of cycles. In the aspect of power, each access operation drives high capacitance for the bus, consuming additional energy. Since handhelds are constrained by battery capacity, the additional energy consumption poses a severe burden to the overall system of handhelds.

III. Overall Architecture

Fig. 3 shows the operation flow of our processor that consists of a subdivider, a 3D Graphics Engine (GE), and a 3D Display Engine (DE). First, the subdivider refines input meshes into finer meshes, producing a smooth surface. Then, the GE renders stereo images from the fine meshes by applying two different view transformations and inherently produces a depth-map. During the rendering process, a shader in the GE imposes various effects to the fine meshes, and the other units in the GE generate pixels within the primitives of the fine meshes. The DE receives the rendered stereo images and the depth-map and synthesizes a 3D image by accomplishing the VIP and the MP. The combined architecture provides synergetic coupling effects such that viewers can see smooth and realistic objects floating in the air through a 3D display.

IV. Bandwidth-Saving Subdivider

In this section we provide an overview of our subdivision approach, followed by our compact data structure and our adaptive subdivision method. Throughout the paper, we use LOD_k to denote a refined mesh computed after performing k subdivision levels to the base mesh; LOD_0 represents the base mesh.

A. Re-Computing based Depth-First Subdivision Scheme

To perform the CCSS in the mobile memory architecture, we propose a re-computing based depth-first scheme. The depth-first scheme iteratively subdivides a face of LOD_0 until its refined mesh satisfies termination criteria, and then begins to subdivide another face of LOD_0 shown in Fig. 4. This depth-first scheme has not been well adopted for evaluating CCSSs on the GPU, mainly because of the following two reasons: 1) it requires stack operations that do not suit well to the streaming architecture like GPUs and 2) it is not straightforward to efficiently maintain data structures that support crack-free adaptive subdivision and provide the one-ring neighborhood information of a face for CCSSs.

We choose the depth-first subdivision scheme, mainly because it has a much smaller working set during the subdivision process than that of the breadth-first subdivision scheme. In our depth-first subdivision scheme, we first bring a target face and its one-ring neighborhood of LOD_0 stored in off-chip memory into on-chip memory. Then, we subdivide the target face until its refined mesh satisfies the subdivision criteria. During this subdivision process, subdividing a face

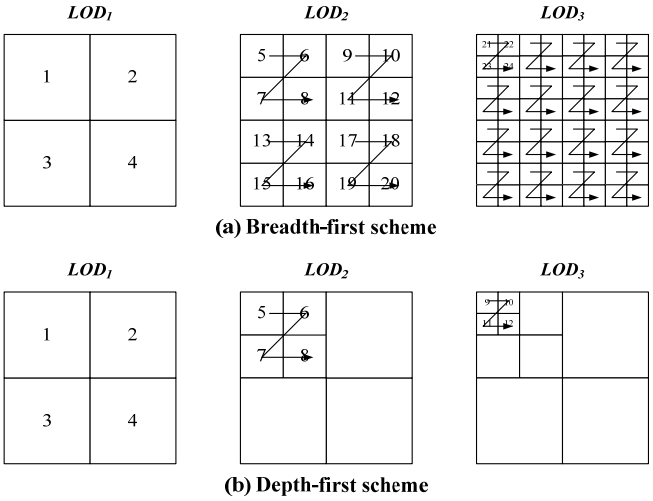


Fig. 4 Breadth-first scheme subdivides all the faces of LOD_k and begins to subdivide the faces of LOD_{k+1}. Depth-first scheme subdivides a face of LOD_0 up to LOD_k and subdivides a next face of LOD_0. The number shown in each figure indicates the subdivision ordering.

requires its one-ring neighborhood information. Therefore, as we subdivide the target face, we also have to refine the faces of the one-ring neighborhood of the target face. We store the refined mesh of the target face and its neighboring faces in on-chip memory. When the refined mesh of the target face satisfies the subdivision criteria, the subdivider sends them directly to a vertex shader, instead of storing the refined mesh in off-chip memory. Then, we begin to subdivide a next face of LOD_0 in the same manner mentioned in above. Therefore, we can store the working set of subdividing a target face into a small on-chip memory and avoid expensive off-chip memory accesses by utilizing data stored in the on-chip memory.

Note that we also subdivide the neighboring faces of the target face. There can be two different approaches for handling these refined meshes of those neighboring face: re-computing and re-loading. The re-loading method stores these refined meshes in off-chip memory for later use by loading them from off-chip memory. On the other hand, the re-computing method discards all the refined meshes, and allows the subdivider to re-compute them when we subdivide those neighboring faces of target faces of the base mesh. We adopt the re-computing method, since it performs better than the re-loading method, although it causes many redundant computations (e.g., 50% more computations than the reloading method). Note that the re-computing based depth-first method does not perform any extra off-chip memory access during the subdivision process, once we fetch a target face and its neighborhood from off-chip memory.

To support our design decision, we estimate the performance of these two different methods by simulating the architecture of handhelds. We refer to a Mobile DDR datasheet available from Samsung [13] to estimate off-chip memory access time. However, we ignore the wire delay between on-chip memory

Table 1 Access time of re-loading scheme and re-computing scheme

Access time (μs)	*LOD_1*	*LOD_2*	*LOD_3*
Re-loading	13.6	36.2	88.6
Re-Computing	4.4	7.2	12.8

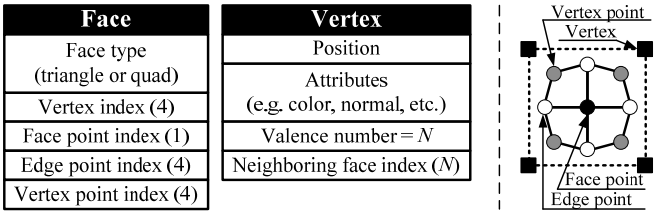


Fig. 5 The left figure shows our edge-less data structure, which consists of face and vertex structures. The number shown in each parenthesis indicates the count of corresponding elements. The right figure shows different points computed from vertices of a face.

and off-chip memory, because the wire delay is not explicitly exposed and can vary depending on a specific system. For the simulation, we perform the subdivision for a regular mesh that has a valence number of 4 to construct LOD_3. In the re-loading scheme, all the refined meshes of the neighboring faces are transferred between on-chip memory and off-chip memory during every subdivision process.

As can be seen in Table 1, the re-computing scheme has a shorter access time than the re-loading scheme. The re-computing scheme continually improves as we compute more refined meshes. Since we ignore the wire delay of the re-loading scheme, the re-computing scheme can show higher improvement than the re-loading scheme.

A downside of our method is that it requires additional on-chip memory to contain the refined meshes of a target face and its neighboring faces. When we support up to LOD_3 and use our edge-less data structure, which will be explained in a later section, the required size of on-chip memory is 20 KB. This size is affordable on recent mobile phones; for instance, the size of PowerVR GPU in iPhone is 64KB [14].

B. Edge-less Data Structure

We propose a compact edge-less structure, to further reduce the bandwidth requirement between on-chip memory and off-chip memory, under the re-computing based depth-first subdivision scheme. The edge-less data structure maintains two separate structures: vertex and face structures. It does not record any edge information and thus has a lower bandwidth requirement, compared to other commonly used data structures such as half-edge [10][11] and render-dynamic [4] data structures. More specifically, the edge-less data structure has 73% and 66% lower bandwidth than the half-edge and render dynamic data structures respectively. Fig. 5 shows the elements of the edge-less data structure.

Suppose that we plan to subdivide a target face. To do that,

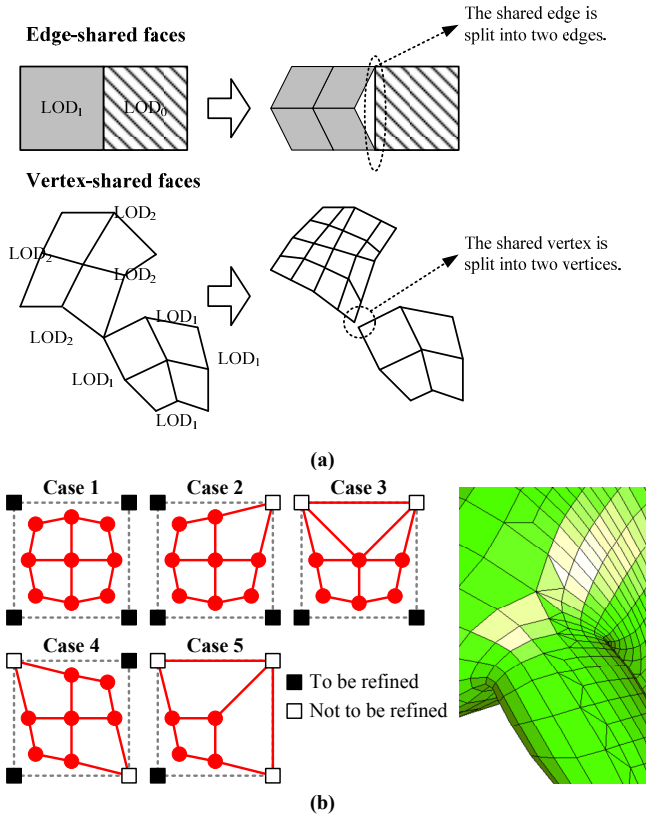


Fig. 6 (a) If LODs are computed per face or per edge, there remains crack possibility. (b)The left and right figures show five mesh connectivity patterns used to provide crack-free adaptive subdivisions and the result of such subdivision on a mesh respectively.

we load data of the target face and its one-ring neighboring faces. Hence, we first load the face element of the target face and then load the vertices of the face by referring to the vertex indices stored in the face element. For each vertex of the face, we also load its neighboring faces by referring to the neighboring face indices stored in its vertex element. Then, we also load the vertices of these neighboring faces. In Section IV-E, we will describe how to use the edge-less data structure on the subdivision process with an example.

Once all the information is loaded, then we compute the face, edge, and vertex points of the target face and its one-ring neighborhood. To compute an edge point of an edge, two end-vertices of the edge and the two face points of the faces that share the edge are required. However, the edge-less structure does not explicitly record any edge data. Instead, we store those data (e.g., vertices, face points, etc.) in registers and reconstruct the necessary edge information based on those stored data. To compute the vertex point of a vertex, all the face and edge points around the vertex and the valence number of the vertex are required. The valence number is stored within the vertex structure. Also, we store all those points in registers and compute the vertex point without accessing data stored in off-chip memory. A detailed subdivision algorithm and these registers will be introduced in Section IV-E.

C. Adaptive Subdivision

Most previous approaches [4][15] perform adaptive subdivision depending on view-dependent refinement criteria and generate crack-free adaptive meshes. These techniques perform such operations using all the available information about the refined mesh at every subdivision level, causing a high bandwidth requirement between the GPU and off-chip memory. Instead, we propose a simple adaptive subdivision method that does not rely on any information other than the information of a target face under the subdivision. Therefore, it can be independently applied to any faces.

If LODs are evaluated per face, edge-shared faces might have different LODs as shown in Fig. 6-(a). In this case, the shared edges have different curvatures, making cracks. To prevent these cracks, LODs would be evaluated per edge. However, it also makes cracks when faces share a vertex as shown in Fig. 6-(a). In order to prevent all kinds of cracks, our subdivider evaluates LODs per vertex. Since cracks are occurred when connected vertices have different LODs, we draw all the cases that the vertices of a face have different LODs. A face of CCSSs is a quad consisting of four vertices so that only one, two, or three vertices can have different LODs in a face; it is impossible that zero or four vertices have different LODs in a rectangular. For example, if two vertices have different LODs, there come out two patterns. Depending on whether or not each vertex passes the refinement criteria, we choose one among five different mesh connectivity patterns (Fig. 6-(b)) of the face.

A user can arbitrarily configure the refinement criteria on the specific registers of the chip (REG_{LOD1}, REG_{LOD2}, and REG_{LOD3}) through the chip's host interface. Then, the judge unit in the subdivider compares the depth of each vertex with the configured values in REG_{LOD1}, REG_{LOD2}, and REG_{LOD3}. For example, if the depth is larger than the value in REG_{LOD2} and smaller than the value in REG_{LOD3}, the vertex should be subdivided up to LOD_2.

Note that our mesh connectivity patterns consist of both the quads and triangles. Since the GE of our processor treats only triangles, the subdivider splits every quad into two triangles at the end of the subdivision process. Our adaptive refinement can be independently applied to any faces.

D. Subdivider Architecture

Fig. 7 shows the subdivider architecture where two on-chip memories are integrated. In subdivision operations, the loaded vertices at LOD_k do not be used any more at higher LODs (LOD_{k+1}, LOD_{k+2}, …). Due to very low utilization of the loaded vertices, we design the on-chip memories as buffers not caches. The on-chip memories store the face and vertex data represented in the edge-less data structure. They store all the refined meshes of the target face (*TF*) and one-ring neighborhood faces (*NF*s) up to LOD_3 with a maximum valence number of 8. Also the subdivider has different data

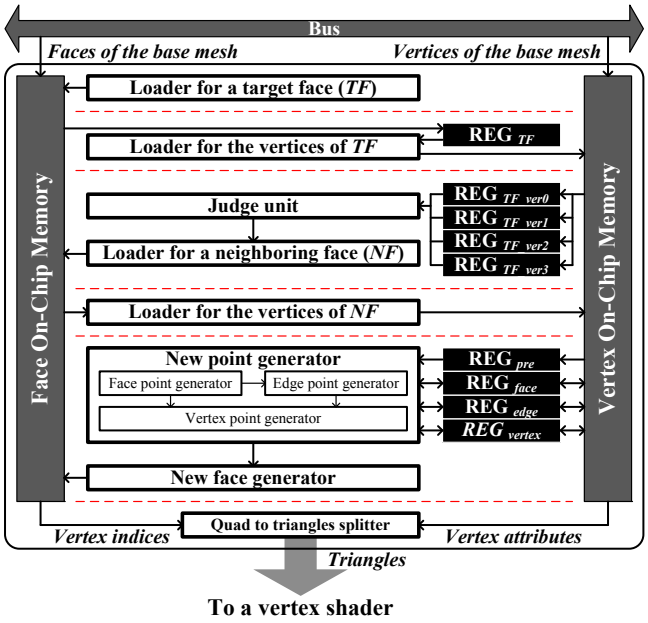


Fig. 7 This figure shows a block diagram of the architecture of our mobile subdivider.

loaders as well as different point and face generators. During the subdivision process, the subdivider uses various registers to prevent redundant computations and loading operations.

The subdivider first loads a *TF* from the face memory and then loads its vertices from the vertex memory. Then, the judge unit decides whether to subdivide each vertex more or not according to programmed depth criteria. If a vertex to be subdivided, the subdivider loads the *NF*s of the vertex and then loads four vertices of each *NF*. By averaging four vertices of each *NF*, we make a P_f of the *NF*. We can make P_e from the second *NF*, because two P_f have been created. When computing all the face and edge points around the vertex, we make a P_v. Then, the subdivider processes the second vertex of the *TF* and continues to perform the same process for the third and fourth vertices of *TF*. During the subdivision process, the subdivider creates sub-faces and stores them in on-chip memory. After subdividing the *TF* at LOD_k, the subdivider begins to subdivide one of the sub-faces. This process is recursively performed until all the sub-faces of the *TF* satisfy smoothness criteria. Then, the subdivider sends only the refined meshes of the *TF* to the shader and discards those of the *NF*s according to the proposed re-computing scheme.

When creating new points, *TF*, *NF*s, and their vertices need to be loaded, but they have dependency each other. As a result, the subdivider must wait all the geometry data to be fetched from off-chip memory. In order to reduce this waiting latency, we insert special registers (REG_{TF_ver0}, REG_{TF_ver1}, REG_{TF_ver2}, REG_{TF_ver3}, and REG_{pre}) to avoid redundant off-chip memory accesses for the same vertices and faces. For example, A target face's vertices are loaded when deciding whether to subdivide the target face more or not. These vertices are redundantly used when computing a face point, an edge point, and a vertex point. They are also used when connecting computed points to create

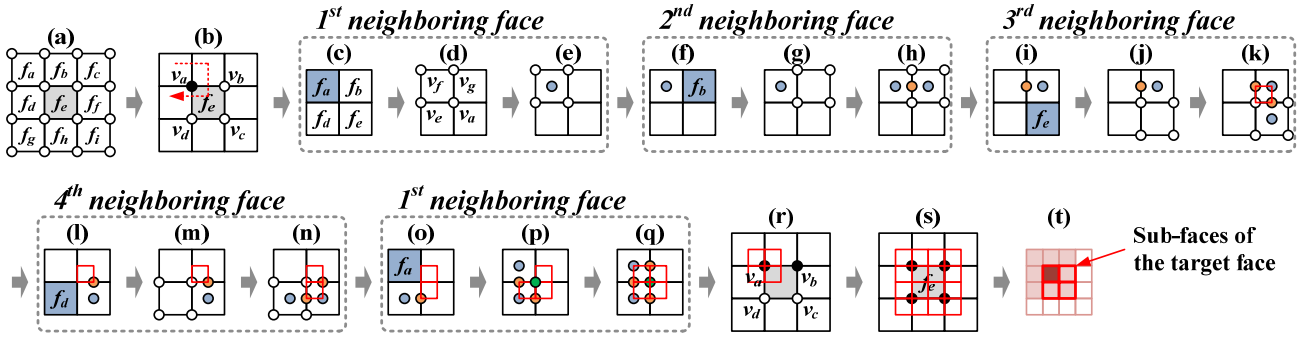


Fig. 8 This figure shows a series of operations of our algorithm to subdivide a target face, f_e shown in (a).

new faces. To avoid the redundant accesses, we store the vertices of a target face in REG_{TF_ver0}, REG_{TF_ver1}, REG_{TF_ver2}, and REG_{TF_ver3}. Other special registers are used in similar purpose. When subdividing a face of regular meshes (valence number is 4) from LOD_k to LOD_{k+1}, the proposed scheme without the special registers loads 64 vertices and 16 faces. By adding the special registers, we load only 32 vertices and 9 faces.

E. Runtime Algorithm

In this section we present a simplified explanation of our subdivision algorithm using an example simple mesh. Suppose that we subdivide a target face, f_e, in the simple mesh shown in Fig. 4-(a). Our method performs the following steps with the example mesh:

1) We initialize all the registers with zero. We bring a target face (e.g., f_e in Fig. 8-(a)), its neighboring faces, and their vertex information from the base mesh stored at off-chip memory to on-chip memory. We do not perform any additional access to off-chip memory, while subdividing the target face.
2) We load the target face, f_e, from on-chip memory and store it in REG_{TF} for later use. Then, we load four vertices, v_a, v_b, v_c, and v_d in Fig. 8-(b), of f_e from on-chip memory and store them into REG_{TF_ver}. We process these loaded vertices starting from the vertex v_a in the order that vertices are loaded. For each vertex, we first check whether we have to subdivide it or not, by using a judge unit. The judge unit decides whether or not to subdivide each vertex, according to programmed depth criteria. Suppose that the judge unit decides to subdivide v_a. Then, we load the neighboring faces of v_a from on-chip memory in the clockwise direction (e.g., the order of f_a, f_b, f_e, and f_d as shown in Fig. 8-(b)). We load them in the clockwise direction in order to easily reconstruct edges that are shared by two neighboring faces. As we load each of them, we set it as a current face and process it as described below.
3) We access the first neighboring face, f_a in Fig. 8-(c), of the vertex v_a. Whenever we access a neighboring face, we compute its face point. To compute a face point of the face, we load four vertices of the face from on-chip memory in the clockwise direction (e.g., v_a, v_e, v_f, and v_g as shown in Fig. 8-(d)). Whenever we load a vertex of the current face, we add it to REG_{face}. After adding the fourth vertex of the face to REG_{face}, we compute the face point (e.g., the blue circle in Fig. 8-(e)) of the face by dividing the value of REG_{face} by four. Whenever the face point is computed, it is added to REG_{edge} and REG_{vertex} for computing edge and vertex points. Finally we initialize REG_{face}. Note that the current face and its next neighboring face share an edge. We store the two end-vertices of the edge in registers during the traversal of those neighboring faces; hence, we avoid the need to store the edge information in the edge-less data structure. Note that at least one of these two end-vertices is one of vertices of the target face, which is already stored in REG_{TF_ver}. We store another vertex in REG_{pre}. By accessing these registers, we can reconstruct the edge information.
4) We access the second neighboring face, f_b in Fig. 8-(f), of the vertex v_a. We load four vertices of the face. Since the current face and its previous face share an edge, we can fetch two end-vertices of the edge from registers (e.g., REG_{TF_ver} and REG_{pre}) and load other two vertices from on-chip memory. We then perform operations with registers to create the face point of the face in a similar manner, as we did in the step (3). From the second neighboring face, we can compute the edge point of an edge that is shared between the current face and its previous face. In this case, the edge consists of v_a and vg and is denoted as e. To compute the edge point, we add two end-vertices of the edge e into REG_{edge}. Since REG_{edge} stores the sum of two face points and two end-vertices of the edge e, we can compute the edge point (e.g., the orange circle in Fig. 8-(h)) of the edge e. Whenever we compute an edge point, we add it to REG_{vertex} and initialize REG_{edge}.
5) Whenever loading a neighboring face, we check whether the face is the target face. Since the third neighboring face, f_e in Fig. 8-(i), is the target face stored in REG_{TF}, we can get its vertices (e.g., white circles in Fig. 8-(j)) by accessing REG_{TF_ver}. We compute the face and edge points of the current face in a similar manner as we did at the step (4). From the third neighboring face, we can construct a subface, the red rectangle in Fig. 8-(k), by connecting two edge points, a face point and a vertex point. At this time,

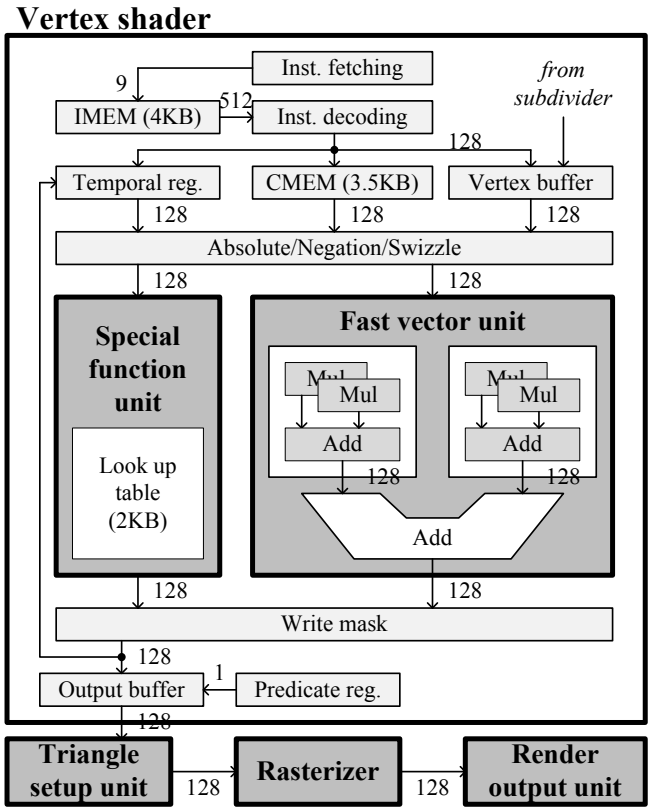


Fig. 9 This figure shows the architecture of our 3D graphics rendering engine. A shader receives the vertices of refined meshes from the subdivider and provides various rendering effects. The rest units fill pixels in the rendered primitives and send them to the 3D display engine. To increase performance, the shader uses a fast vector unit, SIMD architecture, and multi-threading technique.

the geometry of the vertex point is not known. However, we create its index and space in on-chip memory, and then fill its content later.

6) As we did at the step (5), we load the fourth neighboring face, f_d in Fig. 8-(l), and perform similar operations with the registers to create its face point, edge point, and subface as shown in Fig. 8-(n).
7) We access the next neighboring face, which is, in fact, the first neighboring face. We load its face point from the on-chip memory and perform similar operations with registers to create the last edge point and a sub-face in a similar manner, as we did at the step (6). Since all the points around the vertex v_a are accumulated in REG_{vertex}, we construct a vertex point, the green circle in Fig. 8-(p), and produce a final sub-face as shown in Fig. 8-(q).
8) We process the next vertex, v_b in Fig. 8-(r), of the target face. If the judge unit decides to subdivide v_b, we then perform the same process from the step (3) to step (7) with v_b. Otherwise, we skip the subdivision process for vb. Then, we continue to perform the same procedure for the third and fourth vertices, v_c and v_d. To connect vertices of the target face f_e and the points created from the subdivision process, we apply one of the five mesh connectivity patterns shown in Fig. 6.
9) After subdividing the target face, different sub-faces are created as shown in Fig. 8-(s). We store them in the on-chip memory and then our depth-first subdivision scheme begins to subdivide one of the sub-faces (e.g., sub-faces with thick red lines in Fig. 8-(t)) of the target face. This process is recursively performed until all those sub-faces satisfy the programmed subdivision criteria.

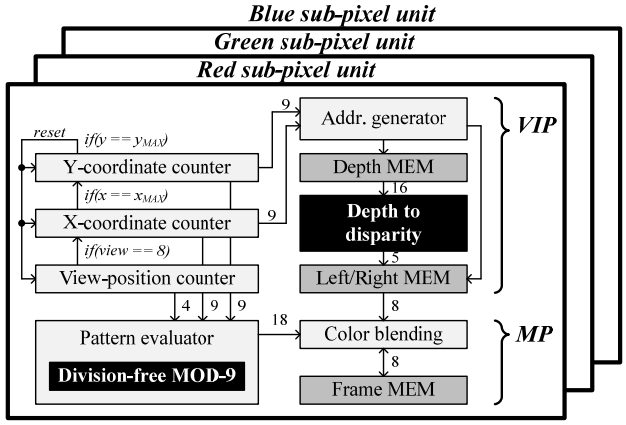


Fig. 10 The 3D display engine consists of three identical units to exploit data parallelism during 3D display process. It reduces bandwidth requirement by reordering the operation sequence of 3D display process. In both the VIP and the MP, division-free datapath reduces the latency of critical paths.

At the end of this subdivision process, we get the subfaces of the target face as well as the sub-faces of neighboring faces of the target face. We only send the sub-faces of the target face to a vertex shader. With the re-computing subdivision scheme, we discard all those sub-faces created from neighboring faces of the target face, instead of maintaining them in off-chip memory. Our algorithm applied to a target face of valence number 4 can be easily applied to target faces with different valence numbers.

V. 3D Graphics Engine

The proposed 3D display processor needs a pair of stereo images and a depth-map to perform the 3D display process. The GE renders a pair of stereo images by applying two different view transformations and inherently produces a depth-map during rendering process. The GE consists of a shader, a triangle setup unit, a rasterizer, and a render output unit as shown in Fig. 9.

The shader has a programmable architecture for a high degree of rendering flexibility and executes rendering instructions for the vertices of the finer meshes that are the output of the subdivider. It supports up-to-date mobile 3D graphics API, OpenGL|ES 2.0 [16]. The shader utilizes 4-way Single Instruction Multiple Data (SIMD) architecture to exploit data parallelism between the elements of vertex attributes. Together, the shader applies the multi-threading technique to hide the latency of other operations using the independency

between vertices [17][18]. There are two arithmetic units in the shader: a Fast Vector Unit (FVU) and a Special Function Unit (SFU). The FVU especially accelerates vector operations such as a 4-input dot product (DP-4) [19], because the most frequent operations in shader programs are 4×4 matrix multiplications for geometry transformation that are composed of DP-4. Since the FVU not only accelerates the DP-4, but also produces the results of four multiplications and two additions, it executes the fundamental arithmetic instructions of shader programs. In typical vector units, both the carry-propagated additions and normalizations at the end of each multiplication have critical latency. The FVU in this work omits or replaces them with carry-save additions, which reduces the latency of DP-4 by 30% compared with prior work [17]. Trigonometric functions such as exponent, reciprocal, logarithm, and square root are hard to be executed through basic multiplications and additions. The SFU accelerates them using a pre-computed look-up-table. Both the FVU and SFU are compliant with IEEE 754 single-precision floating-point format.

The triangle setup unit and the rasterizer produce the pixels of each triangle based on a tile. The render output unit performs the rest of rendering process such as a depth test or an alpha test.

VI. 3D Display Engine

Fig. 10 shows the DE architecture where the DE receives the output of the GE, stereo images and a depth-map, and performs the 3D display process.

Conventionally the VIP is executed per image; the VIP completely produces *i* th view-image and then begins to interpolate *i+1* th view-image. After the VIP produces all the intermediates, the MP starts mixing operations by loading all the images including the intermediates and a pair of stereo images. During conventional 3D display process, the DE must store and load all the intermediates in off-chip memory, resulting in bandwidth waste.

In this work, we adopt the reordered VIP idea that is presented in our prior work [18]. Since the typical VIP produces intermediates in the order of view-positions, it loads input pixel by changing the x- and y-coordinates of an input pixel while fixing the view-position. Instead the reordered VIP changes the view-position, while maintaining the x- and y-coordinates of an input pixel. It makes the DE load only single pixel during producing the intermediate pixels for all the view-positions. Consequently the reordered VIP completely removes the memory operations for the intermediates, and the bandwidth requirement for 3D display process is reduced by 88.9% for the target 3D display.

When designing the 3D display processor, there exist two critical paths. One is depth to disparity conversion, and the other is a modulus operation to get a remainder of division by 9 (MOD-9). In the VIP, a disparity value is required per pixel for interpolation. Since the GE produces a depth instead of the disparity in this processor, we need to convert a depth to a disparity, which includes a division operation. The MP performs the MOD-9 operation to allocate an intermediate pixel to proper position on the display. Since the modulus operation is implemented by a divider in SYNOPSYS DesignWare [20], it becomes a critical path. In 0.13µm CMOS technology, other datapath takes less than 6.5ns, but the conversion process takes 14.5ns and the modulus operation takes 11.1ns. In our previous implementation [18], we design division-free datapath for both the critical paths. Using the division-free datapath, the conversion operation takes only 2.8ns, and the modulus operation takes only 3.3ns. As a result, the DE shows even 325fps frame rate at mobile display resolution (e.g., 480x320 of iPhone and Android phones) for the target 3D display, a 9-view auto-stereoscopic display. The DE always produces a pixel of a 3D image per cycle, so the overall performance is in inverse proportion to display resolution. A user can configure the resolution of a 3D image and the number of view-positions whose maximum supporting ranges are SXGA resolution (1280×1024) and 9 view-positions.

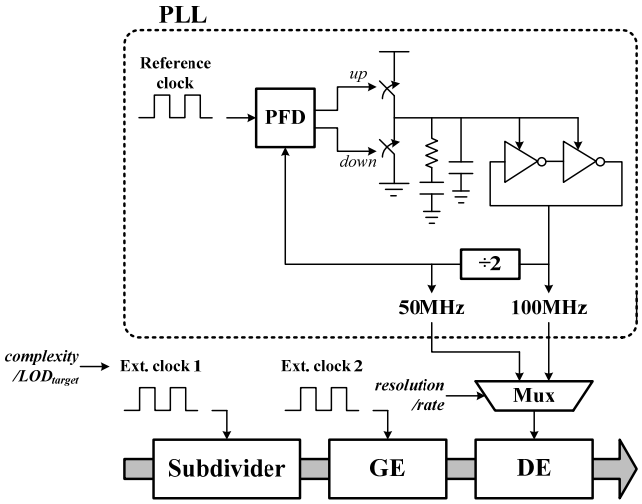
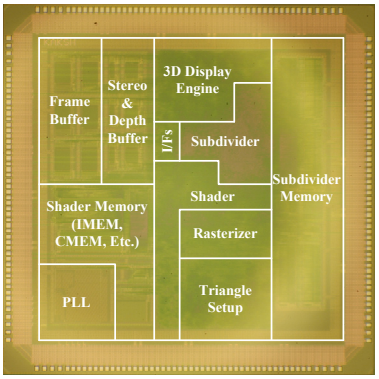


Fig. 11 This work scales down the operating frequencies of the subdivider and the DE depending on model complexity, LOD_k, resolution, and synthesis rate. Since power consumption is linearly proportion to operating frequency, the power consumption is linearly decreased by scaling down the frequency.

VII. Power Saving Scheme By Scaling Down Frequency

Since handhelds are constrained by battery capacity, power saving technique is indispensible when designing a processor. Our work saves power consumption by scaling down the operating frequencies of each functional block. Since power consumption is computed by CV^2f, where C is capacitance, V is voltage, and f is frequency, scaling down frequency makes power consumption to be linearly decreased.

In this processor we scale down the frequencies of the subdivider and the DE, because they do not need to run at the maximum rate at all times. Whereas the subdivider has



Technology	0.13µm 1P6M CMOS
Voltage	1.2V(core), 3.3V(I/O)
Die Size	965K gates in 4.5mm x 4.5mm
Frequency	50MHz
Power	140mW
Subdivision	Catmull-Clark subdivision Subdivide 770K faces/s 1/100 B/W saving
3D Graphics	SIMD & multi-threading Tiling rasterization OpenGL|ES 2.0
3D Display	9-view lenticular display 326fps at 480x320

Fig. 12 The left figure shows a die photo, and the right figure summarizes chip features.

maximum 770K faces per second subdivision rate at 50MHz, the amounts of subdivided faces may be less than 770K faces depending on a model complexity and LOD_{target}. We define the target subdivision rate by the following metrics. First we compute the number of faces to be subdivided ($N_{subdivided}$) in (1), and then computes the operating frequency by a proportional expression between $N_{subdivided}$ and the number of faces at 50MHz (770,000 faces) in (2).

$$N_{subdivided} = N_{initial} \times weight \quad (1)$$

where $N_{subdivided}$ is the number of faces to be subdivided, $N_{initial}$ is the number of initial faces, *weight* is a variable depending on LOD_{target} (*weight* is 1 at LOD_1, 5 at LOD_2, and 21 at LOD_3).

$$f_{scaling} = 50\text{MHz} \times (N_{subdivided} / 770{,}000) \quad (2)$$

where $f_{scaling}$ is the target frequency to be scaled.

Also the DE may not be running at the maximum rate. 100MHz, at all times. Depending on a target resolution and a target synthesis rate, we scale down the frequency of the DE as 50MHz.

When the operating frequency of some functional block is less than or equal to 50MHz, the functional block receives a clock signal from an external device. On the other cases, a clock signal is transferred from a PLL that is integrated in the chip and makes either 50MHz or 100MHz. Since the maximum operating frequency of the subdivider is 50MHz, the clock frequency of the subdivider is externally controlled. On the other hand, the DE can be running at more than 50MHz, it can select one of 50MHz or 100MHz from the PLL.

VIII. Results

In this section we show that the designed processor significantly saves bandwidth requirement while conserving visual quality and achieving enough performance compared with the prior work that are optimized for desktop PCs.

A. Chip Implementation

The processor integrates 965K gates in 4.5mm×4.5mm die using 0.13µm CMOS technology. It consumes 140mW at 50MHz and 1.2V. The subdivider integrates about 77K gates with 20 KB SRAMs. Because its small chip size, the subdivider can be used as a dedicated block in handhelds similarly to the rasterizer. Fig. 12 shows chip features and a die photo.

B. Memory Bandwidth Result

Subdivision operations are mainly composed of memory loading operations, and those memory operations are dependent each other. Thus, we compare our work with previous works based on memory related performances such as bandwidth, off-chip memory access time, and off-chip memory access energy.

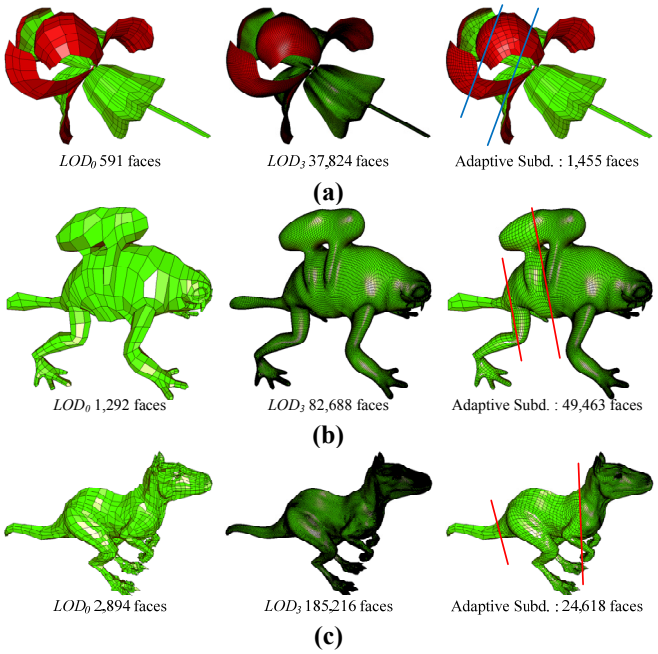


Fig. 13 This figure shows benchmark models. Each column from the leftmost shows the base mesh (i.e. LOD_0), LOD_3, and adaptively subdivided mesh, where a depth, the distance from a view-point, is used as a refinement criterion; two lines in the rightmost column indicate the threshold where the resolution changes.

We measure the bandwidth requirement based on the following two conditions: 1) Use of the burst mode and 2) ignoring the wire delay between on-chip and off-chip memories. The burst mode automatically fetches the next data element without sending address signals, resulting in a higher transfer rate than the normal mode. Also, we ignore the wire delay, since the delay is hard to be measured and is changeable depending on mobile phones. Based on these two assumptions, we measure the bandwidth requirement by computing a weighted sum of the number of off-chip memory access, where the weights are corresponding to the sizes of data. Equation (3) shows how to compute bandwidth requirement. We compute the numbers of off-chip memory accesses for face, edge and

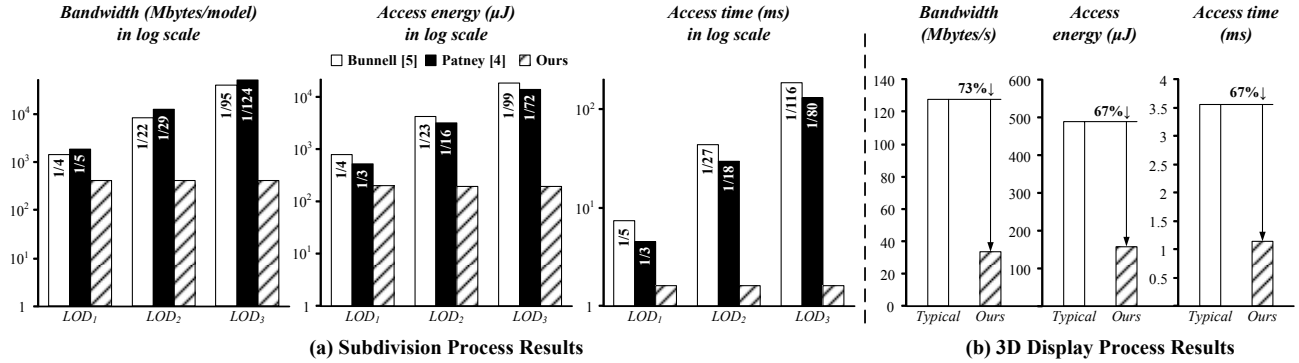


Fig. 14 (a) Our subdivider has about 100:1 reduction in memory performance at LOD_3 over the prior work [4][5]. The ratios in bars are the results of our method over those of the prior work [4][5]. (b) In 3D display process, our work reduces bandwidth by 73% and off-chip memory access energy and time by 67% compared with typical scheme.

vertex by using a C-model simulator. The data size of face, edge and vertex are computed by analyzing the entries of each data structure.

We also analyze off-chip memory access time and energy consumption using a datasheet for the commercial off-chip memory, mobile DDR memory [13], based on the measured bandwidth requirement. Equation (4) and Equation (5) show how to compute memory access time and energy.

$$bandwidth = (N_{face\_access} \times DataSize_{face}) + (N_{edge\_access} \times DataSize_{edge}) + (N_{vertex\_access} \times DataSize_{vertex}) \quad (3)$$

$N_{face\ access}$, $N_{edge\ access}$,and $N_{vertex\ access}$ are the numbers of off-chip memory accesses for face, edge, and vertex at each data structure, respectively.

$DataSize_{face}$, $DataSize_{edge}$,and $DataSize_{vertex}$ are the data size of face, edge, and vertex at each data structure, respectively.

$$Access\ time = t\_RP + t\_RCD + t\_AC + (t\_CK \times 2) + (t\_CK / 2 \times BurstLength) \quad (4)$$

- *t_RP*: The minimum delay time to precharge memory cells. This is required for activating memory cells accessed by the current RAS (Row Address Strobe) signal.
- *t_RCD*: The minimum delay time between RAS signal to CAS (Colum Address Strobe) signal (RAS latency)
- *t_AC*: The minimum delay time to access output data from clock edge
- *t_CK*: The minimum clock cycle time from rising edge to rising edge
- *BurstLength*: In burst addressing mode, the number of data to be transferred per transaction

$$Access\ energy = VDD \times [\ (I\_DD2N \times t\_RP) + (I\_DD3N \times t\_RCD) + \{I\_DD4 \times (t\_AC + t\_CK \times 2 + t\_CK / 2 \times BurstLength)\}\ ] \quad (5)$$

- *I_DD2N*: Precharge standby current in non-power down mode
- *I_DD3N*: Active standby current in non-power down mode
- *I_DD4*: Operating current in burst mode
- *VDD*: Operating voltage
- *BurstLength*: In burst addressing mode, the number of data to be transferred per transaction

When Mobile DDR is activate, it takes 2 cycles to receive the first burst data ($t\_CK \times 2$), and two burst data are transferred per cycle ($t\_CK/2 \times BurstLength$). Access energy is estimated by multiplying current and time to operating voltage (*VDD*). The current is divided into *I_DD2N*, *I_DD3N*, and *I_DD4* according to operation mode. Since *I_DD2N* is precharging current, it is multiplied to *t_RP*. Similarly, since *I_DD3N* is standby current, it is multiplied to *t_RCD*. Finally, operating current, *I_DD4*, is multiplied to a total operating time including the delay time ($t\_AC + (t\_CK \times 2) + (t\_CK / 2 \times BurstLength)$).

We compare our subdivider with prior methods that evaluate CCSSs, which are the method of Patney et al. [4] using the render-dynamic data structures and the method of Bunnell [5] using the half-edge data structure. For the various tests and comparison, we use three benchmark models: *Iris*, *Monster Frog*, and *Killeroo* models shown in Fig. 13. The base meshes of these models have only quads with a maximum valence of 8.

Fig. 14-(a) shows the bandwidth requirement, estimated off-chip access time, and estimated off-chip energy consumption for the most complex model, *Killeroo*, with different subdivision approaches. Other benchmarks such as *Iris* and *Monster Frog* show similar results. Our subdivider has constant memory accesses regardless of LOD_k, because the depth-first based re-computing scheme loads just the target face and its neighborhood of LOD_0. Especially for LOD_3, our work shows about 100:1 reduction over the prior work [4][5] for all the memory performances and for all the benchmarks.

We do the same procedure for the DE as shown in Fig. 14-(b). When measuring the bandwidth requirement of the DE, we assume that the DE runs at 60fps rate for 480×320 display resolution. The 3D display process has a fixed bandwidth

requirement regardless of benchmarks. Our work reduces bandwidth requirement by 73%, and similarly reduces both the off-chip memory access time and energy by 67% compared with typical 3D display process.

We compare the overall chip bandwidth with a commercial portable bus, ExpressCard 2.0 PCI [6] that has 625Mbytes/s bandwidth. When measuring the bandwidth, the chip subdivides *Killeroo* up to LOD_3 and runs the DE at 60fps from the subdivided model. In this environment, the subdivider and the DE consume about 267Mbytes/s that is less than half of the ExpressCard bandwidth. We can use the rest of the ExpressCard bandwidth for performing other 3D graphics techniques.

C. Performance and Quality Comparison

The subdivider refines 770K faces per second at 50MHz. The subdivider's performance cannot be shown as an absolute frame rate, because it is changeable depending on model complexity. Since there has been no mobile subdivider, we indirectly compare the subdivision rate with the latest subdivision work [4] that targets on desktop PCs. While Patney et al. [4] takes 31.34ms to subdivide *Killeroo* up to LOD_5, our work takes 79.01ms to subdivide the same model up to LOD_3. Considering the different LODs, the Patney et al.'s method is 40 times faster than our work. However, the Patney et al.'s method requires about 100 times higher bandwidth, access time, and energy consumption than our method and thus is considered inappropriate for handhelds that have the mobile memory architecture. Additionally our subdivider has 520K transistors, whereas Nvidia GTX280 [21] used in the Patney et al.'s work has 1.4 billion transistors, which shows how our subdivider is tiny; GTX280 is about 2700 times larger.

It has been known that LOD_5 can produce a smooth surface [22], but our work supports up to LOD_3 because we found it strikes a good balance between the subdivision quality given the typical handheld screen resolution (480×320) and the size of on-chip memory (e.g., 20 KB, 44 KB, and 139 KB for supporting up to LOD_3, LOD_4, and LOD_5, respectively). We quantitatively verify our subdivider is feasible on handhelds. We estimate subdivision quality by the average number of faces per pixel in the screen space. For this test, we use *Killeroo*, and assume that only half of faces are visible due to back-face culling and the projected scene occupies 70% of screen space. Our work targets 480×320 resolution for handhelds, and the Patney et al.'s method assumes 1600×1200 for desktop PCs. Given this configuration, our work provides 0.9faces/pixel and the Patney et al.'s method has 1.1faces/pixel. In conclusion, our subdivider provides similar quality to the Patney et al.'s method designed for PCs.

In 3D display processing, the DE already achieves impressive performance, 325fps at 480×320 resolution of a 9-view lenticular 3D display, using the division-free datapath. Since 3D display processing is a kind of image processing, the 3D display engine always shows a fixed performance regardless of model complexity.

IX. Conclusion

As visual quality becomes the key feature of handhelds, there are increasing demands of modern 3D graphics and 3D display techniques on handhelds. Unfortunately these techniques are memory-intensive applications, which cannot be supported given mobile memory architecture.

Our mobile 3D display processor with a subdivider can render smooth surfaces on a mobile 3D display. The subdivider attempts to use on-chip memory instead of off-chip memory by minimizing the working set. Together the 3D display engine removes the bandwidth to access intermediates by reordering the operation sequence of the 3D display process. Consequently the subdivider saves memory bandwidth less than 1% of the prior methods [4][5], and the 3D display engine reduces the bandwidth requirement by 73%. This bandwidth-saving result translates into a significant improvement of off-chip access energy and time. The subdivider has competitive subdivision rate and quality considering chip area and bandwidth performance. Also the 3D display engine shows impressive performance such as 325fps on mobile resolution (480×320). We wish our research to contribute the state-of-art on the next-generation mobile devices.

Acknowledgment

The chip was fabricated through the MPW of IC Design Education Center (IDEC). This work was supported by the IT R&D program of MKE/[KI002134, Wafer Level 3D IC Design and Integration] and by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2011-0000320). We would like to thank *Killeroo* model of Headus Inc (http://www.headus.com.au).

References

[1] Gee K., "Direct3D 11 tessellation," Presentation, Gamefest (2008)
[2] Akenine-Moller T., Strom J., "Graphics processing units for handhelds," Proc. of the IEEE (2008), pp.779–789
[3] P. Benzie, et al., "A survey of 3DTV displays: techniques and technologies," IEEE Transactions on Circuits and Systems for Video Technology (2007), 17
[4] Patney A., Ebeida M.S. and Owens J. D., "Parallel view-dependent tessellation of Catmull-Clark subdivision surfaces," ACM High Performance Graphics (2009), pp. 99–108
[5] Bunnell M., "Adaptive tessellation of subdivision surfaces with displacement mapping," GPU Gems 2 (2005), pp.109–122
[6] Wikipedia, List of device bit rates (2010) [Online] Available: http://en.wikipedia.org/wiki/List_of_device_bit_rates
[7] Catmull E., Clark J., "Recursively generated B-spline surfaces on arbitrary topological meshes," Computer Aided Design (1978)
[8] Zorin D., "Subdivision for modeling and animation," SIGGRAPH Course Notes (2000)
[9] Farin G., "Curves and surfaces for computer aided geometric design a practical guide (5th ed.)," Academic Press Professional Inc. (2002)

[10] Kettner L., "Using generic programming for designing a data structure for polyhedral surfaces," Elsevier Comput. Geom. Theory Appl. (1999) 13, 1, pp.65–90
[11] Shiue L. J., Jones I., Peters J., "A realtime GPU subdivision kernel," ACM Trans. Graph. (2005), 24, 3 pp.1010–1015
[12] Kyusik Chung, Chang-Hyo Yu, Donghyun Kim, and Lee-Sup Kim, "Shader-based tessellation to save memory bandwidth in a mobile multimedia processor," Elsevier Computer & Graphics (2009)
[13] Samsung, Mobile DDR k4x56323pg, datasheet (2007) [Online] Available: http://www.samsung.com/global/business/semiconductor
[14] Imagination, PowerVR SGX, datasheet (2010) [Online] Available: http://www.imgtec.com/powervr/sgx_series5.asp.
[15] Lorenz H., Doellner J., "Dynamic mesh refinement on GPU using geometry shaders," WSCG (2008), pp.97–104.
[16] OpenGL|ES 2.0 [Online] Available: http://www.khronos.org/opengles/
[17] Seok-Hoon Kim, et al., "A 36fps SXGA 3-D display processor embedding a programmable 3-D graphics rendering engine," IEEE Journal of Solid State Circuits (2008), 43, 5, pp.1247-1259
[18] Seok-Hoon Kim, Hong-Yun Kim, Kyusik Chung, Donghyun Kim, Lee-Sup Kim, "A 116fps/74mW heterogeneous 3D-media processor for 3D display applications," IEEE Journal of Solid State Circuits (2010)45, 3, pp.652-667
[19] Donghyun Kim, Lee-Sup Kim, "A floating-point unit for 4D vector inner product with reduced latency," IEEE Trans. Computers (2009), 58, 7, pp.890-901
[20] Synopsys, Designware [Online] Available: http://www.synopsys.com
[21] Nvidia, GTX280, datasheet (2010) [Online] Available: http://www.nvidia.com/docs/IO/55506/GPU_Datasheet.pdf.
[22] Loop C., Schaefer S., "Approximating Catmull-Clark subdivision surfaces with bicubic patches," ACM Trans. Graph. (2008), 27, 1, pp.1–11