Dynamic sampling rate: harnessing frame coherence in graphics applications for energy-efficient GPUs

Martí Anglada[1] · **Enrique de Lucas**[2] · **Joan-Manuel Parcerisa**[1] · **Juan L. Aragón**[3] · **Antonio González**[1]

Abstract

In real-time rendering, a 3D scene is modelled with meshes of triangles that the GPU projects to the screen. They are discretized by sampling each triangle at regular space intervals to generate fragments which are then added texture and lighting effects by a shader program. Realistic scenes require detailed geometric models, complex shaders, high-resolution displays and high screen refreshing rates, which all come at a great compute time and energy cost. This cost is often dominated by the fragment shader, which runs for each sampled fragment. Conventional GPUs sample the triangles once per pixel; however, there are many screen regions containing low variation that produce identical fragments and could be sampled at lower than pixel-rate with no loss in quality. Additionally, as temporal frame coherence makes consecutive frames very similar, such variations are usually maintained from frame to frame. This work proposes Dynamic Sampling Rate (DSR), a novel hardware mechanism to reduce redundancy and improve the energy efficiency in graphics applications. DSR analyzes the spatial frequencies of the scene once it has been rendered. Then, it leverages the temporal coherence in consecutive frames to decide, for each region of the screen, the lowest sampling rate to employ in the next frame that maintains image quality. We evaluate the performance of a state-of-the-art mobile GPU architecture extended with DSR for a wide variety of applications. Experimental results show that DSR is able to remove most of the redundancy inherent in the color computations at fragment granularity, which brings average speedups of 1.68x and energy savings of 40%.

✉ Martí Anglada
manglada@ac.upc.edu

Extended author information available on the last page of the article

1 Introduction

The growing computing capabilities of today's mobile devices allow for real-time rendering of complex 3D scenes. Supporting users' demands for ever richer applications clashes with the battery-operated nature of those devices, which makes energy efficiency paramount. In particular, previous studies have described the GPU as the most energy-demanding component of mobile SoCs for graphics workloads [1–3].

In real-time graphics, the geometry of objects is modelled with a set of vertices, which the GPU processes, assembles into flat polygons (normally triangles), and projects to the screen plane. Triangles are then discretized into arrays of pixels by sampling their surfaces at regular intervals to produce a fragment (i.e., a set of attributes such as color, depth, normal, etc.) for each sampled location, often just once per pixel. Fragments are then conveniently textured, shaded and blended to obtain a final color value per pixel.

On the one hand, if parts of the scene contain high spatial frequencies, sampling triangles at a low rate may be insufficient to capture fine details and would cause aliasing effects, such as jagged edges or flickering. Supersampling is an approach that relieves these artifacts by sampling at higher rates and combining the results into a single color. However, it involves large energy and performance costs since it increases the number of generated fragments [4], and it is well documented that fragment processing is the most energy consuming stage of the graphics pipeline due to the amount of computations and memory accesses required [5–7]. For that reason, supersampling is rarely applied in real-time rendering applications, especially in mobile GPUs, and scenes are usually rendered at one color sample per pixel.

On the other hand, sampling all triangles at the same rate is not efficient because not all parts of the screen require the same sampling rate [8]. Figure 1 illustrates this phenomenon by comparing two different regions of the screen in a given frame: while 1c contains significant level of detail, 1b is homogeneous with a single color and, therefore, does not require per-pixel sampling. We have quantified, for a variety of mobile graphics applications, the number of 16x16-pixel regions of the screen that do not contain enough level of detail for them to require one sample per pixel. Figure 2 shows that on average almost half of the screen can be processed at a lower sampling rate without affecting image quality. Properly identifying and removing the large amount of resources devoted to these unnecessary computations can lead to a substantial reduction in energy consumption.

Based on the above observations, this paper presents Dynamic Sampling Rate (DSR): a hardware mechanism that dynamically finds and applies, for each part of the scene, the optimal sampling rate, i.e., the lowest sampling rate that does not cause visible artifacts in the rendered image. DSR is designed for Tile-Based Rendering (TBR) architectures [9], a common pipeline organization in mobile devices that divides the screen into rectangular sections -tiles- and renders them in succession, allowing the storage of temporary values in on-chip buffers to avoid their corresponding accesses to main memory. After the rendering of a tile to the
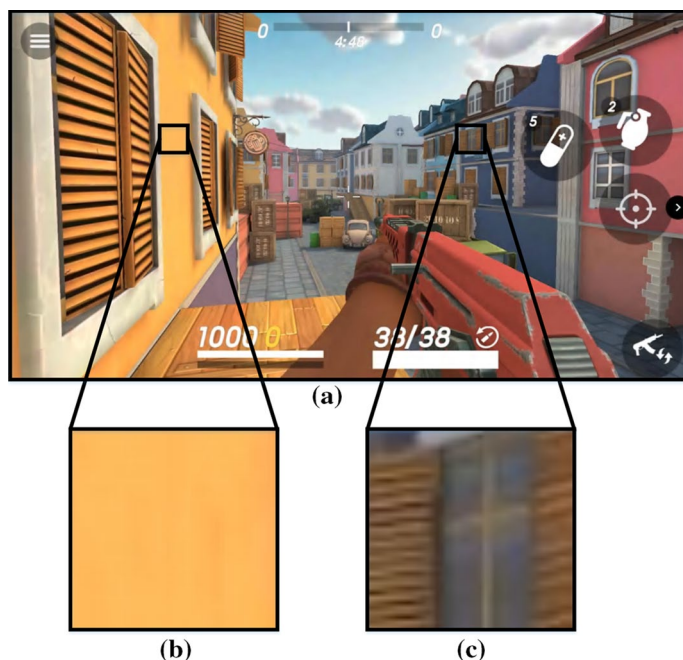
Fig. 1 Difference in level of detail across a frame. **a** Frame of the game *Guns of Boom*. **b** Region with low level of detail. **c** Region with significant level of detail
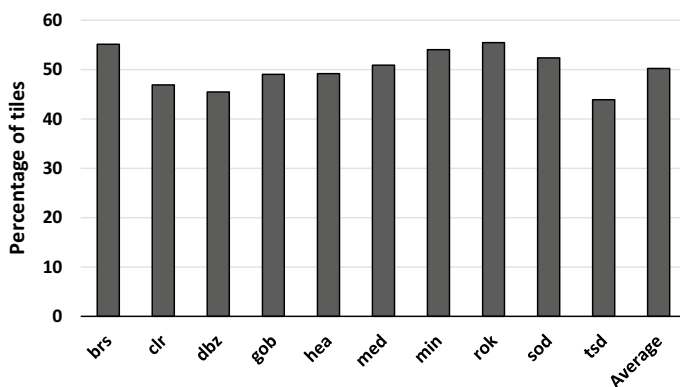

Fig. 2 Number of 16×16 tiles that can be sampled at a rate lower than one sample per pixel without generating per-tile visible artifacts. Section 4 describes the methodology employed for this categorization

on-chip color buffer finishes, DSR computes the Discrete Cosine Transform of the resultant tile image, analyzes the spatial frequencies present in it and decides, based on a simple heuristic, the best sampling rate for the tile: whether it could have been sampled at a lower rate without sacrificing image quality, whether it contains enough detail that the sampling rate needs to be increased or whether

the sampling rate is already optimal. The estimated-best rates for all the tiles are then stored in a small on-chip Lookup Table and are queried during the following frame, before the rendering of each tile. DSR estimates the best sampling rate for a tile in one frame and applies it in the following frame because it takes advantage of the well-known frame coherence property of graphics animations [10]: visual smoothness is achieved by producing a (quick) succession of frames where a large subset of tiles are very similar between two consecutive frames.

DSR addresses the shortcomings of prior work on the area of sampling at coarser granularities to reduce the number of shader executions. The majority of approaches rely on heavy efforts from the programmer to specify which components of the scene contain less details such as certain lights [11] or vertex attributes [8]. Conversely, DSR decides the sampling rates in a completely transparent manner to the programmer. Unlike approaches in which a static sampling rate is set for parts of the scene (such as particular regions of the screen [12] or fragments in the boundary between triangles [13]), DSR dynamically and continuously adapts the sampling rate of the entire scene at tile granularity to closely track image changes. Additionally, to the best of our knowledge, DSR is the first work to take advantage of frame coherence for this purpose. Although frame coherence has been proposed to skip some shader executions in the so-called Checkerboard rendering [14], such scheme is lossy and may affect image quality. DSR, on the other hand, only reduces the sampling rate in tiles that do not contain high spatial frequencies and, therefore, DSR does not produce visual artifacts. Previous approaches that dynamically change the sampling rate must compute their estimates in the middle of the pipeline execution [15], whereas DSR is architected to not introduce any time overhead by overlapping the frequency analysis of one tile with the rendering of the next one.

To summarize, the main contributions of this paper are:

- A new hardware technique, completely transparent to the programmer, that estimates the lowest possible sampling rate to which each tile may be rendered without producing visual artifacts and applies it during the following frame by taking advantage of frame coherence.
- A dynamic mechanism based on real-time analysis of the spatial frequencies that continuously adapts the per-tile sampling rate to track the image changes that occur over time.
- A comprehensive architectural description of the frequency analysis unit and how it is integrated within the graphics pipeline in a way that it causes no timing overheads.
- An implementation and evaluation that shows that Dynamic Sampling Rate reduces the shading activity by 66% and memory accesses by 28%, yielding speedups of 1.68x and an overall energy reduction of 40% on average.

The rest of the paper is organized as follows. Section 2 reviews the common pipeline organization of mobile GPUs and the Discrete Cosine Transform as a mean to mapping an image into the frequency domain. Section 3 describes the design of DSR. Section 4 presents the approach used to set the DSR's parameters. Section 5 illustrates the implementation of DSR and the changes required to the baseline GPU.

Section 6 presents the experimental framework and Sect. 7 quantifies the benefits of applying DSR. Section 8 outlines some related work and Sect. 9 sums up the main conclusions of this work.

2 Background

2.1 Tile-based rendering

In a Tile-Based Rendering GPU, the image is rendered in two decoupled steps, as shown in Fig. 3. First, the Geometry Pipeline processes the vertices of the three-dimensional models, transforming and assembling them into triangles. Triangles are then binned into the tiles that they overlap. Once the whole geometry has been processed and stored, the Raster Pipeline executes the second step, one tile at a time. By working on tiles, all temporary color values can be stored in the on-chip Color Buffer, which is only flushed to main memory once all triangles of the tile have been processed. The execution in the Raster Pipeline starts by fetching the primitives from memory and dispatching them to the rasterizer. Then, the rasterizer samples the surface of triangles at regular space intervals, generating *Fragments*: points inside the triangle with interpolated information at each point. Adjacent fragments are then arranged into groups of four called *Quads* and are sent to the Fragment Shaders which compute their colors by executing user-defined programs in lockstep mode for the four fragments. The color of each pixel in the tile is held in the Color Buffer, and it is obtained from one fragment, by blending multiple adjacent fragments, or by replicating the same fragment into multiple adjacent pixels, depending on whether the sampling rate is equal, higher or lower than one per pixel, respectively.
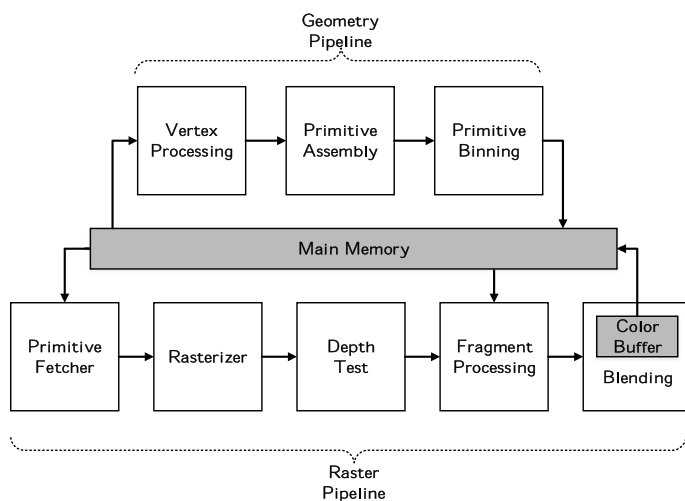

**Fig. 3** Tile-based rendering GPU

2.2 Frequency analysis

The typical sampling rate of a scene is one sample per pixel. According to the Nyquist Sampling Theorem [16], such a sampling rate allows capturing changes in the image every two pixels or more, i.e., with a frequency smaller or equal than two pixels. As shown in Fig. 1, not all regions of the screen contain high frequencies, or changes in the image in a short space. Therefore, a (much) lower sampling rate may be enough to represent the original signals. We propose to analyze the frequencies of each rendered tile to decide the sampling rate to be applied to it in the following frame.

A well known mechanism to obtain the frequency components of an image is the Discrete Cosine Transform (DCT) [17]. As a Fourier-related transform, the DCT maps a function (an image) from the spatial domain to a set of coefficients of basis functions localized in the frequency spectrum. Those basis functions correspond to sinusoids of a certain frequency and are visually represented in Fig. 4. It can be seen that as either the x or y axis increase, the basis function is a sinusoid with higher variation rate, i.e., with higher frequency. Applying a 2D DCT to a block of NxN pixel colors results in a NxN matrix of values, the coefficients of the linear combination of basis functions which represent the original image in the frequency domain. The coefficient present in each element of the matrix indicates how much of that particular frequency is found in the original image.

The 2D DCT has several characteristics that make it an ideal choice for the type of real-time frequency analysis we require to find the optimal sampling rate for a tile:

- It assumes an even symmetry of the function: by construction, the image is mirrored in all its borders, which avoids artificial high frequency components that other transforms introduce by only considering a NxN pixel subset of the image.
- It has very high energy compaction, which means that the great majority of frequency information is summarized in the upper-left region of the result matrix. This allows us to make sampling rate decisions only considering a subset of the NxN coefficients.

Fig. 4 DCT basis functions for N = 8 pixels
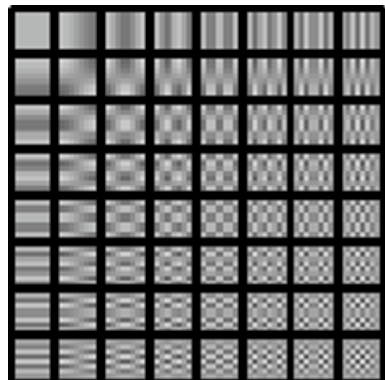
- It has a low complexity cost in comparison with other transforms as only cosines are computed.

Additionally, the 2D DCT is a separable function, which allows for the linear computation of all the elements in one dimension followed by the linear computation of all the elements in the second dimension. These characteristics allow us to implement a fast and energy efficient hardware unit to analyze the frequency components of a tile, explained in more detail in Sect. 5.

3 Dynamic sampling rate

This section describes how the 2D DCT is used to estimate the optimal sampling rate for a tile, i.e., the lowest sampling rate that does not introduce visible artifacts in the overall frame, and how to dynamically adapt it to image changes over time.

When the rendering process of the tile finishes, the Color Buffer contains the final color for all the pixels of the tile. We propose to add a small hardware unit that takes these colors as inputs to compute the 2D DCT and analyzes the resulting matrix of coefficients to determine if the current sampling rate for the tile is optimal. All the DCT coefficients are first aggregated into a single value that summarizes the amount of high-frequency information of the tile. Although a wide variety of metrics exist, we empirically determined that the maximum absolute value among the coefficients corresponding to high-frequency diagonals suffices, and we will refer to it as *MaxC* (we term here diagonal k as the set of all elements of the matrix whose row index plus column index is equal to k: for instance, diagonal 3 consists of elements (0, 3), (1, 2), (2, 1) and (3, 0)). The rationale under this choice is that, intuitively, we are more interested in knowing if the largest high-frequency component is big enough to justify a high sampling rate rather than considering the effect of multiple high-frequency components combined. The low-frequency components of the matrix are not taken into account in the computation of *MaxC*.

Figure 5 illustrates the determination of *MaxC* in a 5×5 coefficient matrix in which we consider diagonals 0 through 3 as low-frequency diagonals. As shown, *MaxC* is 3, since it is the highest absolute value among all the high-frequency diagonals. Although larger values appear in the low-frequency diagonals, they are ignored.

Then, a simple test is conducted to decide the new sampling rate for the tile: *MaxC* is compared against two different thresholds.

- The first threshold, which we label *Reduce Threshold* (T_R), represents the maximum frequency a tile can contain for it to be sampled at a rate lower than the current one. If *MaxC* is lower than the Reduce Threshold, the sampling rate for the tile is reduced.
- The second threshold, which we label *Increase Threshold* (T_I) represents the maximum frequency a tile can contain for it to be sampled at the current rate. If *MaxC* is greater than the Increase Threshold, the sampling rate for the tile is increased.
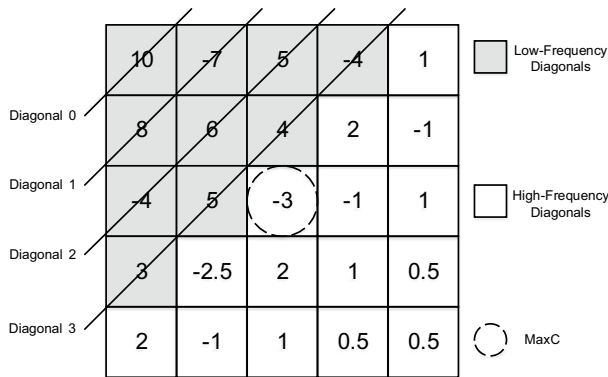
Fig. 5 *MaxC* determination example

In the case that *MaxC* is neither lower than the Reduce Threshold nor greater than the Increase Threshold, the sampling rate for the tile does not change. The new sampling rate for each tile is stored and used to process it in the next frame. The scene is, therefore, not sampled uniformly neither in space nor time: each tile is rasterized with an independent sampling rate and it may be modified across frames to adapt to image changes.

Figure 6 shows the Finite State Machine (FSM) that manages the dynamic sampling rate determination. We consider five different sampling rates: sampling at the center of every pixel (baseline sampling rate) and sampling at the center of every square block of 4, 16, 64 or 256 pixels (as shown in Fig. 7). We will refer to these sampling rates as $1\times, 1/4\times, 1/16\times, 1/64\times$ and $1/256\times$, respectively. These sampling rates are motivated by the baseline GPU architecture employed in this work, which utilizes tiles of 16×16 pixels. Each state in the FSM corresponds to halving the previous sample rate in both *X* and *Y* dimensions, and the lowest state only generates one sample per tile. The transitions among states are controlled by the heuristic decision described above, based on a *<T, D>* tuple that contains: the Thresholds (*T*) to which *MaxC* is compared to, and the number of low-frequency matrix Diagonals that are ignored (*D*) for its computation. We label as $<T_R, D_R>$ the tuples for the Reduce transitions and as $<T_I, D_I>$ the tuples for the Increase transitions.
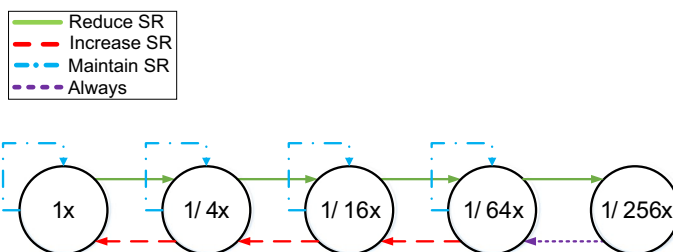

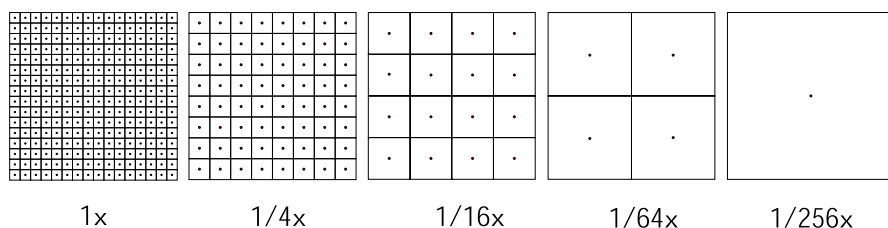**Fig. 6** Dynamic sampling rate Finite-State Machine

Fig. 7 The five sampling rates considered in our experiments, from 1× (left) to 1/256× (right)

As images generated with lower sampling rates have fewer high-frequency components and different sampling rate requirements, each transition in the FSM has individual values for $<T_R, D_R>$ and for $<T_I, D_I>$. Apparently, the FSM has 4 Increase and 4 Decrease transitions. However, at 1/256× rate, fragments are sampled just once and the resulting tile contains a single plain color. As there is no spatial frequency in it, the heuristic cannot make decisions based on the coefficient matrix. Our FSM conservatively forces the 1/256× state to always transition back to 1/64×. Consequently, we must set parameter values for 3 Increase and 4 Reduce transitions in the FSM. Adequate values for these parameters have been empirically determined through extensive experiments with the objective to reduce GPU activity (samples) while keeping the original image quality. Section 4 describes the methodology followed to find such optimal $<T_R, D_R>$, $<T_I, D_I>$ values for each sampling rate.

Although in this work we consider sampling once per fragment to be the highest sampling rate, DSR can easily be integrated in GPUs that allow higher sampling rates, such as the ones implementing Supersampling Antialiasing (SSAA). The maximum sampling rate that those GPUs provide will be used as DSR's baseline sampling rate. DSR will then selectively apply SSAA by dynamically determining which regions of the screen require it. DSR can also be combined with the widely spread Multisampling Antialiasing (MSAA) approach [18]. With MSAA, depth is sampled at several points per pixel (usually 4), while the fragment shader is only computed at the pixel center. All the samples that pass the depth test receive the output of the shader. As DSR only changes the frequency in which shaders are executed, DSR can also be applied on systems that use MSAA.

4 Parameter selection

This section describes the empirical methodology to find the best values for DSR parameters such that frames are rendered at the lowest possible average sample rate (ASR) without producing any visible error.

As depicted in Algorithm 1, we perform an exhaustive parameter exploration. For each parameter combination under test we render all frames, adjusting the sample rate of each tile according to the output of the heuristic. During the search, any combination that produces even a single erroneous frame is directly

discarded. Otherwise, the achieved ASR across all frames is computed for that combination. Eventually, we choose the parameter combination that produces the lowest ASR.

Frame errors are computed by comparing the image quality of the produced frames with respect to the frame rendered at baseline sampling rate using the Mean Structural Similarity Index (MSSIM [19]), a widely adopted, perceptually-based quality metric that estimates the visual impact of changes in image luminance and contrast caused by compression distortions. The MSSIM has been shown to outperform other similarity metrics that just measure differences in pixel color, such as PSNR and MSE, in terms of quality [20, 21] as it correlates better with the perception of the human visual system. A frame error occurs whenever the obtained MSSIM is lower than 95, as it is the point at which defects can be discerned by human beings [22].

Algorithm 1 Basic parameter search

    Input: Set of frames
    Output: Values for the $< T_I, D_I >$ and
$< T_R, D_R >$ **tuples that produce the lowest ASR**

1: **for each** parameter combination **do**
2:     **for each** frame **do**
3:         **for each** tile **do**
4:             $DCT = \text{compute\_dct}(tile, frame)$
5:             **if** $MaxC(D_R, DCT) < T_R$ **then**
6:                 $next = SR[tile, frame] - 1$ ▷ Reduce
7:             **else if** $MaxC(D_I, DCT) \geq T_I$ **then**
8:                 $next = SR[tile, frame] + 1$ ▷ Increase
9:             **else**
10:                 $next = SR[tile, frame]$ ▷ Stay
11:             **end if**
12:             $SR[tile, frame + 1] = next$
13:         **end for**
14:         **if** $\text{contains\_errors}(frame)$ **then**
15:             discard parameter combination
16:         **end if**
17:     **end for**
18:     $\text{compute\_ASR}(parameter\_combination, SR)$
19: **end for**

Each parameter combination contains a set of 14 different parameters (four $<T_R, D_R>$ pairs for the Reduce transitions and three $<T_I, D_I>$ pairs for the Increase transitions). Even considering just a few values for each parameter (say *n*), the sheer amount of combinations to consider (n^{14}) makes an exhaustive exploration

unfeasible. We adopt instead a divide and conquer approach in which we first only focus on finding the best parameters for the Increase transitions. Next, those values are used and kept constant in Algorithm 1 to find the best parameters for the Reduce transitions. By splitting the parameter search into two steps, we substantially limit the number of combinations to explore and we can execute an exhaustive search.

Note however that during the first step we cannot apply Algorithm 1: without values set for the $<T_R, D_R>$ pairs, the procedure lacks a mechanism to dynamically reduce the sample rates and the FSM never reaches the lowest states. Consequently, we must provide an alternative sampling rate reduction mechanism for this first step. Such mechanism must produce tiles at low enough sampling rates that Increase transitions are required to prevent errors due to undersampling. Otherwise (if Increase decisions were never required) we would not test the capabilities of the parameter combinations to produce a low ASR while not producing frame errors.

To build an effective reduction mechanism we first conduct a simple preliminary experiment that finds near-optimal sample rates for each of the tiles in all frames. Those values will act as references and will stay constant during the exploration of the Increase parameters. The reduction mechanism consists in always choosing the lowest sampling rate between the reference value and the outcome of the heuristic (which either increases the sampling rate or keeps it the same).

This preliminary experiment first generates the images of all tiles in all frames at all five sampling rates. It then sequentially analyzes tile by tile the five alternatives and selects the lowest one that does not produce visible errors compared with the same tile at baseline sampling. We term these sample rates *Local Minimum*, because image discrepancies are not analyzed at full frame level but just at tile level. As such, they may not be the optimal sample rates (optimal values may be lower when discrepancies are analyzed at frame level) but they are low enough to be used as a reference in our reduction mechanism.

Algorithm 2 shows the procedure to find the best parameters for the Increase transitions (the first step). Akin to Algorithm 1, for each tile it computes the DCT and decides whether or not to increase the current sampling rate according to the $<T_I, D_I>$ parameters under test (Lines 5–9). However, unlike Algorithm 1, it next considers overriding that decision by choosing instead the stored Local Minimum sample rate for the next frame (Line 11) in case that it is lower. As the algorithm can select a sampling rate lower than the Local Minimum, the found parameters gravitate towards the optimal sampling rates.

Algorithm 2 LocMin parameter search

Input:LocalMinimum, $< T_R, D_R >$
Output:Values for the $< T_I, D_I >$ tuples that produce the lowest ASR

1:　**for each** parameter combination **do**
2:　　　**for each** frame **do**
3:　　　　**for each** tile **do**
4:　　　　　$DCT = \text{compute\_dct}(tile, frame)$
5:　　　　　**if** $MaxC(D_I, DCT) \geq T_I$ **then**
6:　　　　　　$next = SR[tile, frame] + 1$　　　　　　　　　　$\triangleright$ Increase
7:　　　　　**else**
8:　　　　　　$next = SR[tile, frame]$　　　　　　　　　　　$\triangleright$ Stay
9:　　　　　**end if**
10:　　　　$locmin = LocalMinimum[tile, frame + 1]$
11:　　　　$SR[tile, frame + 1] = min(next, locmin)$
12:　　　**end for**
13:　　　**if** contains_errors($frame$) **then**
14:　　　　discard parameter combination
15:　　　**end if**
16:　　**end for**
17:　　compute_ASR($parameter\_combination, SR$)
18:　**end for**

---

5 Implementation

This section describes the combinational logic and memory structures required to implement Dynamic Sampling Rate, and how the frequency analysis and sample rate determination are integrated within the Raster Pipeline.

5.1 Pipeline integration

The Dynamic Sample Rate technique uses a FSM (see Fig. 6) to dynamically determine the sampling rate of each tile based on its current state and its *MaxC*. It requires a new hardware structure called *Sampling Rate Table* (SRT), with one entry per tile, that holds the state of each tile in a frame. Since the FSM has 5 different states, a state can be represented with 3 bits. Consequently, for a frame resolution of 1080×1920 pixels (as modelled in our experiments) there are 8100 tiles and the storage overhead of the SRT is 2.96 KB.

Other than the SRT, Dynamic Sampling Rate requires very minor modifications to the pipeline, as shown in Fig. 8. Tiles are scheduled and primitives are fetched in the same way as in the baseline because the sampling rate only affects the discretization process. The Rasterizer still produces Quads (square groups of four adjacent fragments), so they can be depth tested, shaded and blended as in the baseline. The main difference is that the screen area covered by each fragment is bigger than a
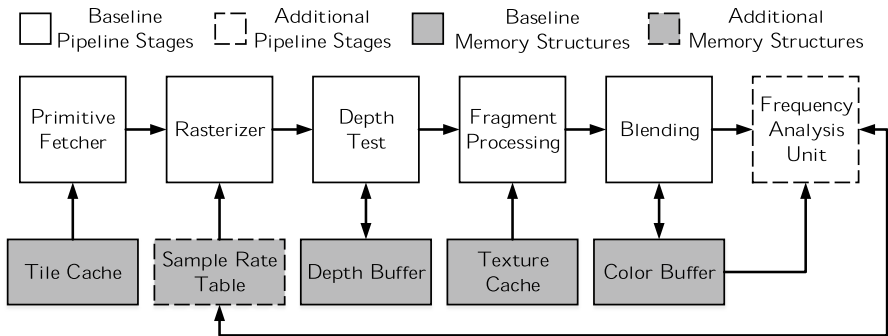
Fig. 8 Raster pipeline with DSR

pixel when the sampling rate is lower than 1×. We refer to those fragments as *Superfragments* and to a group of four Superfragments as *Superquads*. Producing a superfragment at a sampling rate of 1/N × N only requires sampling at the center of a grid of N × N pixels.

Whenever a tile starts its processing, its state (hence the associated sampling rate) is fetched from the Sampling Rate Table. The Rasterizer generates Superfragments according to the stored state. The Depth and Color Buffers already have capacity to hold temporary values for the 256 pixels (16 × 16 pixel tiles) of the baseline resolution. Fragments within a Superfragment share depth and color, so, only one read/write operation in the Depth Buffer is executed when depth testing a Superfragment and only one read/write operation in the Color Buffer is executed when blending a Superfragment. This results in some entries of the Color Buffer not being initialized after a tile finishes its processing.

When all primitives in a tile have been processed, the final color values of the Superfragments present in the Color Buffer are upsampled by replicating their value to all pixels that belong to the Superfragment. Afterwards, the contents of the Color Buffer are transferred to main memory and the DCT computation of the tile starts in the Frequency Analysis Unit.

5.2 Frequency analysis unit

The 2D DCT is a separable function [23]. This property allows to transform a N × N *Input* image into the frequency domain by successively applying 1D transforms, first along the rows and then along the columns (or vice-versa). By considering separability, the well-known 2D-DCT formula can be rearranged as shown in Eq. 1:

$$DCT(p,q) = \alpha(p)\alpha(q) \sum_{m=0}^{N-1} \cos\frac{(2m+1)\pi p}{2N} \sum_{n=0}^{N-1} Input_{mn} \cos\frac{(2n+1)\pi q}{2N} \qquad (1)$$

where $0 \leq p, q \leq N - 1$ and the scale factors α are defined as:

$$\alpha_p = \alpha_q \begin{cases} \frac{1}{\sqrt{N}} & \text{if p = 0 or q = 0} \\ \sqrt{\frac{2}{N}} & \text{otherwise.} \end{cases} \quad (2)$$

The 2D-DCT formula is usually expressed in matrix notation as [24]:

$$DCT = K \, Input \, K^T = (K(K \, Input)^T)^T \quad (3)$$

where K is the so-called *Kernel Matrix*, that contains precomputed values for both the scale factors and the cosine functions in the form of:

$$K_{pq} = \begin{cases} \frac{1}{\sqrt{N}} & \text{if } p = 0 \\ \sqrt{\frac{2}{N}} \cos \frac{(2q+1)\pi p}{2N} & \text{otherwise.} \end{cases} \quad (4)$$

Our frequency analysis scheme uses the Synopsys's implementation of the 2D DCT transform from their DesignWare library [25]. This module is based on the afore-mentioned Kernel Matrix precomputation and row-column decomposition. The computation of the 2D DCT shown in Eq. 3 is divided in two steps, decoupled by an auxiliary buffer (*Aux*) that holds temporary results. The first step computes the 1D-DCT of the rows ($Aux = (K \, Input)^T$) and the second step completes the 2D computation ($DCT = (K \, Aux)^T$). In the Synopsys implementation, a single buffer is used for storing both the temporary and final results. This buffer, which we name *DCT Buffer*, is written by columns and read by rows to emulate the two transpositions.

Figure 9 shows a block diagram of the Frequency Analysis Unit and its data-flow: the input data is read from the Color Buffer ① and is multiplied by the Kernel Matrix ② using a series of compute units. Each unit computes the 1D-DCT of a row and stores the result in the DCT Buffer ③. Since the tiles in our modeled GPU are composed of 16x16 pixels and the frequency analysis unit contains 4 compute units, each unit sequentially processes 4 rows. Once the 16 rows have been processed, the second pass is performed: the temporary contents of the DCT Buffer ④ are multi-plied to the Kernel Matrix and stored back in the DCT Buffer ⑤, four columns at a time, until the final 2D DCT is computed.

The original Synopsys design operates sequentially in each row and column, as it does not have hardware to compute multiple 1D-DCTs in parallel. This implies sig-nificant time overheads to compute the entire 2D-DCT of 16×16 elements. We have slightly modified the design by replicating the compute units. Experimentally, we have determined that with 4 compute units the frequency analysis and sampling rate determination do not cause stalls in the pipeline and the energy and area overheads are minimal (results in Sect. 7).

Once the DCT computation ends, the hardware ⑥ estimates the best sampling rate for that tile using the scheme presented in Sect. 3: it first uses the matrix of coef-ficients (ignoring the D first diagonals) to compute *MaxC*; then, following the FSM in Fig. 6, it decides the new tile state (hence a corresponding sampling rate), based on the current state ⑦ and the comparison between *MaxC* and the *T* threshold.
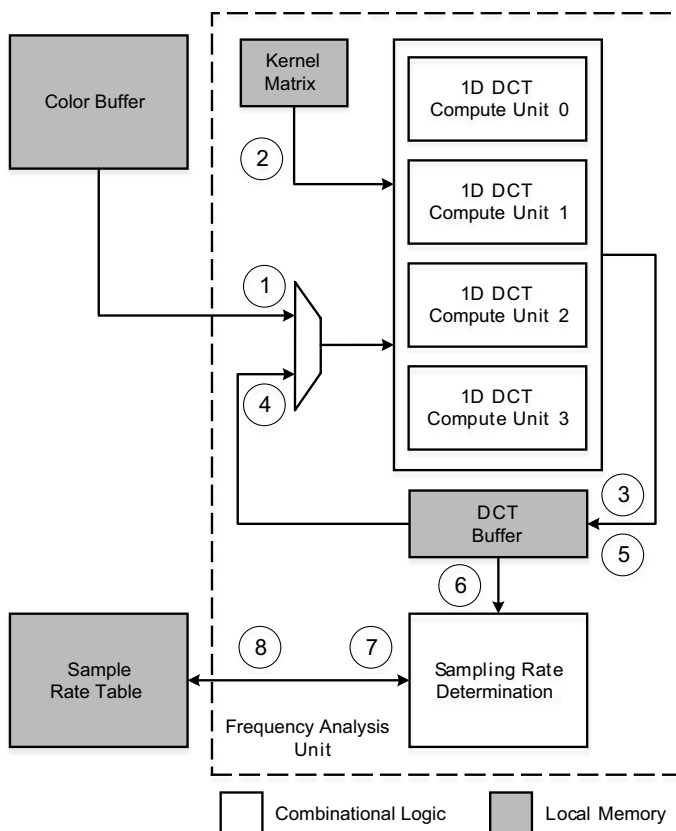
Fig. 9 Frequency Analysis Unit overview

Finally, the new tile state is stored in the SRT ⑧ to indicate the sampling rate to be used in the following frame.

6 Experimental framework

The set of benchmarks employed in our experiments include ten unmodified commercial Android graphics applications that represent the current landscape of real-time rendering in mobile devices (see Table 1). The benchmark set consists of contemporary applications with tens of millions of downloads in Google Play [26] and includes a variety of workloads: from 2D applications with simple models (e.g., *tsd*) to 3D applications with detailed scenes (e.g., *gob*). The applications are also diverse in how the camera is placed and moved through the scene. The set includes benchmarks with the static-camera scenes (*clr*, *dbz*, *hea*,

Table 1 Benchmark suite

Benchmark	Alias	Genre
Brawl Stars	brs	Beat'em Up
Clash Royale	clr	Real-Time Strategy
Dragon Ball Z: Dokkan Battle	dbz	Board Game, Puzzle
Guns of Boom	gob	First-Person Shooter
Hearthstone	hea	Collectible Card Game
Merge Dragons!	med	Puzzle
Minecraft	min	Sandbox
Rise of Kingdoms: Lost Crusade	rok	Real-Time Strategy
Sonic Dash	sod	Endless Runner
Toy Story Drop!	tsd	Puzzle

tsd) and simple scrolls (*brs*, *med*, *sod*, *rok*) that characterize mobile applications and also scenes with free-from and swift camera movements (*gob*, *min*).

The experimental framework used in this work is composed of three different stages:

1. The benchmarks are first run on a smartphone equipped with an Adreno 530 GPU and a 5.15-inch, 1080p display. The smartphone is connected to GAPID [27], an open source debugging tool that captures the OpenGL API calls of an application to the graphics card driver. After all the loading screens have been cleared, the game is played for several seconds with human-generated inputs, which allows GAPID to obtain a file containing all the executed OpenGL commands of 100 frames of archetypal execution.
2. The logged commands are then fed to the software-based back-end included in Gallium3D [28], which implements all the stages of the Graphics Pipeline and runs it on a CPU. We instrument the execution of the software renderer to obtain a complete instruction and memory trace of the application.
3. The trace drives the execution of the cycle-accurate simulator of the TEAPOT toolset [29], from which we obtain timing and energy results. The parameters used in our experiments are presented in Table 2 and model a TBR architecture resembling the ARM Mali-450 GPU [30]. The simulator has been extended to include all the combinational logic and local memory structures required by DSR. Additionally, the Frequency Analysis Unit described in Sect. 5.2 has been implemented in VHDL and synthesized to obtain its delay and power using the Synopsys Design Compiler, the modules of the DesignWare library and the 28/32 nm technology library from Synopsys [31].

Table 2 GPU simulation parameters

Baseline GPU parameters	
Tech Specs	400 MHz, 1 V, 32 nm
Screen Resolution	1080×1920
Tile Size	16×16 pixels
Main memory	
Latency	50–100 cycles
Bandwidth	4 B/cycle (dual channel LPDDR3)
Size	1 GB
Queues	
Vertex (2x)	16 entries, 136 bytes/entry
Triangle, Tile	16 entries, 388 bytes/entry
Fragment	64 entries, 233 bytes/entry
Caches	
Vertex Cache	64 bytes/line, 2-way associative, 4 KB, 1 bank, 1 cycle
Texture Caches (4×)	64 bytes/line, 2-way associative, 8 KB, 1 bank, 1 cycle
Tile Cache	64 bytes/line, 8-way associative, 128 KB, 8 banks, 1 cycle
L2 Cache	64 bytes/line, 8-way associative, 256 KB, 8 banks, 2 cycles
Color Buffer	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle
Depth Buffer	64 bytes/line, 1-way associative, 1 KB, 1 bank, 1 cycle
Non-programmable stages	
Primitive assembly	1 triangle/cycle
Rasterizer	16 attributes/cycle
Early Z test	32 in-flight quad-fragments
Programmable stages	
Vertex Processor	1 vertex processor
Fragment Processor	4 fragment processors
Additional hardware	
Sample Rate Table	8100 entries, 4 bits/entry

7 Results

In this section, we present the energy and performance gains of our proposal compared to those of the baseline GPU. Figure 10 shows the energy consumption of the whole system (GPU plus memory) with our DSR proposal normalized to the Baseline described in Section 2.1. We can see that having independent and dynamic sampling rates for each tile achieves an average 40% reduction of energy, with savings up to 67% (for *dbz*). Figure 10 also shows the minor costs of activating DSR: the static and dynamic energy consumption of the Sampling Rate Table, and the
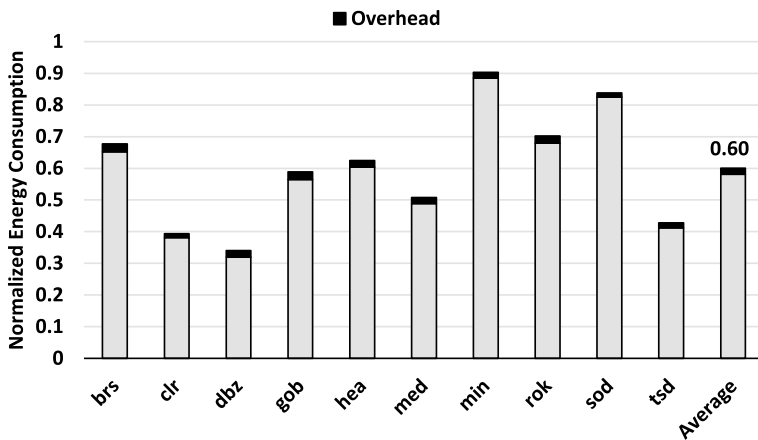
Fig. 10 Energy consumption of DSR compared to the baseline GPU

logic and temporary memory required to compute the 2D DCT of the tiles (Fig. 9). All together, they represent less than 2% of the total energy consumption and less than 1% of the area of the baseline GPU.

Figure 11 shows the reduction in execution cycles of DSR normalized to the Baseline design and broken down into Geometry and Raster cycles. On average, our proposal leads to 1.9× speedup in the Raster Pipeline, with maximums of more than 4× (*dbz*). This translates into a 36% global execution time reduction, since the Geometry Pipeline contains no modifications with respect to the baseline. Note that we do not incur in any execution time penalty, as the frequency analysis of the tiles and their sampling rate determination is completely overlapped with the Raster Pipeline activity.
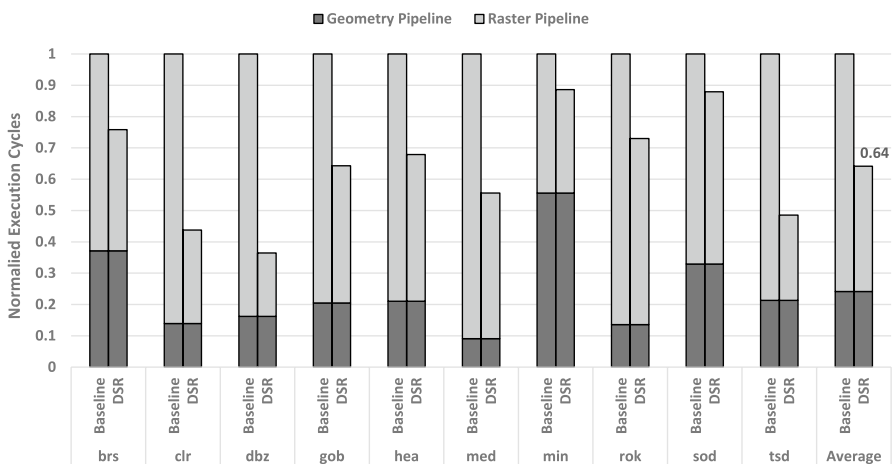
Fig. 11 Execution time of DSR compared to the baseline GPU

The benefits in energy consumption and execution time of DSR are caused by sampling most tiles at lower rates, as shown in Fig. 12. Benchmarks with static scenes, (*clr*, *dbz*, *hea* or *tsd*) are able to achieve a considerable drop in Average Sample Rate, as tiles will generally maintain its frequency across frames. Therefore, tiles that do not require to be sampled at 1× rate will have the opportunity to be sampled at a lower rate, and that lower rate will be maintained for a long period of time. DSR is also able to obtain an important ASR reduction in applications for which the scene is moving, such as *med* or *rok*. While constantly changing geometry results in a majority of the tiles needing to be sampled in the baseline rate, there is still a significant portion of the scene that can be rendered at a much lower rate, yielding ASR reductions of more than 35%. Those portions of the scene can even be found in applications that experience swift, constant changes in the camera, such as *sod*.

On average, less than half of the tiles are sampled at the baseline rate, while almost 40% of the tiles are processed using the two lowest sampling rates (1/64× and 1/256×). The Average Sample Rate across all benchmarks and frames is thus reduced to 0.36 samples per fragment. This greatly reduces the activity of the Fragment Shaders, as shown in Fig. 13. DSR reduces the average number of processed fragments by 66% and the number of texture accesses to main memory by 28% when compared to the Baseline. The gap between both numbers is caused by an increase in sparsity: as samples are taken at larger intervals, the likelihood of reusing a texture cache line is smaller than in the Baseline. However, the great reduction in processed fragments still allows for significant savings in overall texture traffic.

Rendered scenes in real-time applications tend to smoothly vary across consecutive frames. Therefore, the sampling rate requirements of tiles may evolve over time. As DSR analyzes the frequencies of the scene after rendering each tile, it manages to dynamically capture such changes and quickly adjust the sampling rates of all individual tiles accordingly. Figure 14 illustrates this process by depicting the fluctuations in the sampling rate of 4 tiles of the application *gob*, starting at the beginning of the captured execution and running it for 50 frames.
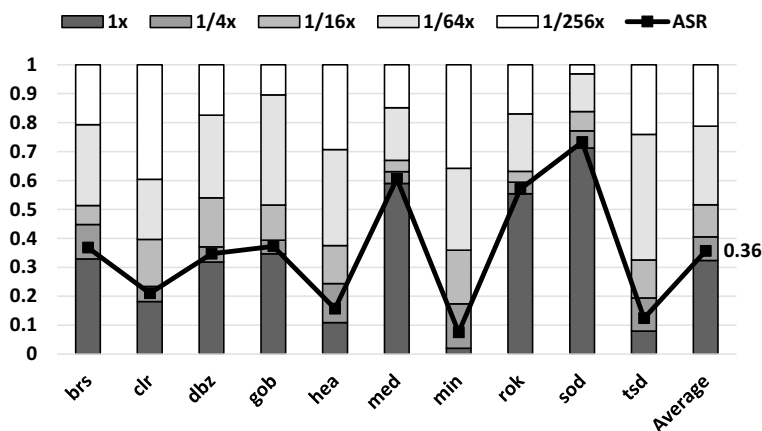

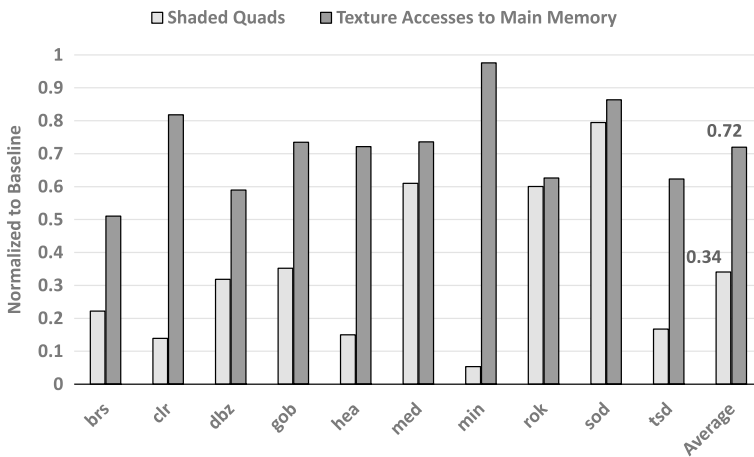**Fig. 12** Breakdown of sampling rates

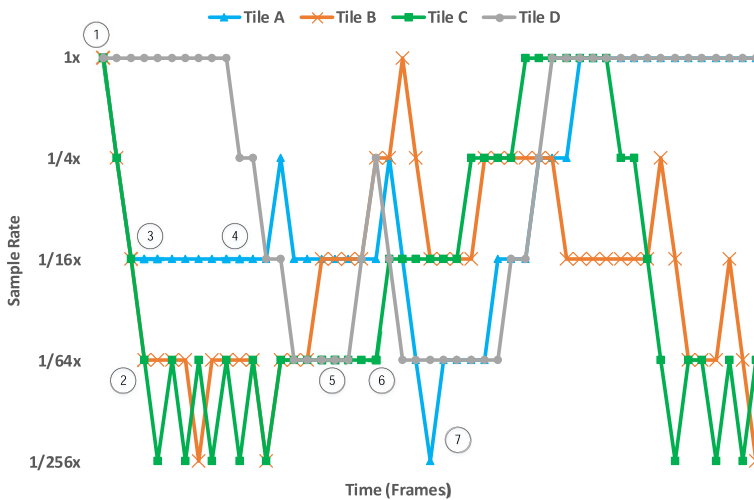**Fig. 13** Shader activity of DSR compared to the baseline GPU

Fig. 14 Evolution of the sampling rate of 4 different tiles over several frames

We can observe that DSR starts sampling all the tiles at the maximum rate, 1× (①). Tiles A,B and C can be sampled at a much lower sampling rate, and spend a small transitory period of time continuously reducing their sampling rate (②) until their optimal rate for their current spatial frequency is found (③). Tiles remain in their estimated-optimal sampling rates (e.g., ④, ⑤) until the spatial frequencies in them change (e.g., ⑥). Note how every time that a tile is sampled at 1/256x rate (e.g., ⑦), the sampling rate is immediately increased in the following frame, as described in Fig. 6. It can be observed that the scene is not sampled uniformly neither in space (in a particular frame the sampling rate of the 4 tiles

is normally different) nor in time (the sampling rate of a particular tile changes across the frames).

In our experiments, DSR has not produced a single error in all the generated frames, i.e., has not rendered any frame with a MSSIM lower than 95 when compared with the frame rendered at baseline sampling rate. Despite our benchmarks containing swift camera movements and object displacements across the screen, the similarity between consecutive frames allows the reuse of the estimated-best sampling rates without producing any visual artifacts. Albeit a sparse phenomenon, more abrupt alterations may occur in a particular frame, such as in a change of scene. The correctness of DSR cannot be guaranteed in these rare scenarios, as it is based on frame coherency. We have performed an experiment to quantify the effect that DSR has on image quality whenever there is a scene change. To do so, three additional 100-frame traces for the benchmarks listed in Table 3 have been generated. Each trace contains two different scene changes, emulated by entering and exiting the pause or settings menu of the application. With the renderization of these applications' frames with DSR active, we have observed that only the frame rendered immediately after each of the six scene changes is erroneous. Subsequent frames are indistinguishable from frames rendered at baseline sampling rate. It is well documented that the human eye requires some time to construe visual information, as scenes cannot be properly recreated in less than 67ms [32]. This time is greater than what a single frame lasts in 30 frames per second, the frame rate which is considered to be the minimum acceptable [33, 34], so we can conclude that these potential errors affect only a single frame and will not be perceived by the user.

8 Related work

There is a lot of interest in the graphics community in reducing shading costs so that more complex and realistic scenes can be rendered. Several techniques dynamically detect regions of the screen that can be sampled at lower rates by adding additional pipeline stages before or after the shading process. Deferred Adaptive Compute Shading [35] divides the framebuffer into levels, subsets of pixels progressively farther apart. Fragments are only shaded if the neighbor pixels from the previous level are not similar. Otherwise, the color of the fragment is computed by averaging the results of its neighbors. The work of Sathe and Akenine-Möller [13] reuses shading computations for triangles that share an edge by adding a comparison queue before the processing stage. In Adaptive Image-Space Sampling [15], the resolution is reduced in areas that contain less perceivable

Table 3 Additional benchmarks for the image quality experiment

Benchmark	Genre
Alto's Odyssey	Endless Runner
PlayerUnknown's Battlegrounds	Battle Royale
Homescapes	Puzzle

detail, which are evaluated with an additional pass after the geometry processing. Conversely, DSR is architected to not introduce any time overhead by completely overlapping the sampling rate estimation for a tile with the rendering of the next one.

To avoid the runtime overhead of determining components with less detail, several works allow the programmer to statically determine the sampling rate. In coarse Pixel Shading [8], the sample rate of each primitive can be controlled based on their vertex attributes. He et al. [11] design new language abstractions that grant each shader program the ability to determine which components can be processed at which rate. In NVIDIA's Variable Rate Shading [12], the programmer decides which sampling rate to apply in each section of the screen. DSR, on the other hand dynamically estimates the best sampling rates in each tile by using a hardware-only mechanism, in a completely transparent manner to the programmer.

Frame coherence has been previously leveraged to reduce the number of samples to process. In Checkerboard rendering [14], each frame shades an alternate half of the pixels in the screen. The color of the non-shaded half is obtained by applying reconstruction filters to the results obtained in the preceding frame. A large number of shading computations are avoided at the cost of some visual artifacts, since the lossy nature of the reconstruction and the fixed undersampling cannot perfectly reproduce neither motion nor visibility changes. In contrast, DSR estimates sampling rates at the finer granularity of tiles, can render tiles at the small rate of only one fragment per tile and does not affect image quality because it only reduces the sampling rate whenever a tile does not contain high spatial frequencies.

9 Conclusions

This paper proposes Dynamic Sampling Rate (DSR), a novel microarchitectural technique to reduce shader executions by determining the lowest sampling rate for each tile in a frame that does not reduce the overall quality of the rendered images. DSR analyzes the frequency components of a tile once it has been processed and decides the rate in which the tile's triangles will be sampled in the following frame. The sampling rate prediction leverages the frame-to-frame coherence inherent in animated graphics applications, which results in a high likelihood that the frequency components of a tile are maintained across consecutive frames.

We have shown that for a set of unmodified commercial Android applications DSR reduces the fragment-level redundancy by 66% on average with minimal hardware overhead, leading to an average speedup of 1.68× and energy savings of 40%.

References

1. Patil S, Kim Y, Korgaonkar K, Awwal, I, Rosing, TS (2015) Characterization of user's behavior variations for design of replayable mobile workloads. International Conference on Mobile Computing, Applications, and Services, pp 51–70
2. AnandTech: Qualcomm Snapdragon S4 (Krait) Performance preview. Accessed = 2022-02-11 (2012). http://www.anandtech.com/show/5559/qualcomm-snapdragon-s4-krait-performance-preview-msm8960-adreno-225-benchmarks/4
3. Anglada M, de Lucas E, Parcerisa J, Aragón JL, Marcuello P, González A (2019) Rendering elimination: early discard of redundant tiles in the graphics pipeline. 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp 623–634. https://doi.org/10.1109/HPCA.2019.00014
4. Maule M, Comba JL, Torchelsen R, Bastos R (2012) Transparency and anti-aliasing techniques for real-time rendering. 2012 25th SIBGRAPI Conference on Graphics, Patterns and Images Tutorials, pp 50–59. IEEE
5. Shebanow M (2013) An evolution of mobile graphics. Keynote talk at High Performance Graphics
6. Pool J (2012) Energy-precision tradeoffs in the graphics pipeline. Ph.D. thesis, The University of North Carolina at Chapel Hill
7. de Lucas E (2018) Reducing redundancy of real time computer graphics in mobile systems. Ph.D. thesis, UPC, Computer Architecture Department
8. Vaidyanathan K, Salvi M, Toth R, Foley T, Akenine-Möller T, Nilsson J, Munkberg J, Hasselgren J, Sugihara M, Clarberg P, Janczak T, Lefohn A (2014) Coarse pixel shading. In: Proceedings of High Performance Graphics, pp 9–18. Eurographics Association
9. Akenine-Moller T, Strom J (2008) Graphics processing units for handhelds. Proc IEEE 96(5):779–789
10. Hubschman H, Zucker SW (1982) Frame-to-frame coherence and the hidden surface computation: constraints for a convex world. ACM Trans Graphics 1(2):129–162
11. He Y, Gu Y, Fatahalian K (2014) Extending the graphics pipeline with adaptive, multi-rate shading. ACM Transactions on Graphics (TOG) 33(4):142
12. NVIDIA: NVIDIA GPU Turing Architecture. Accessed = 2022-02-11 (2018). https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf
13. Sathe R, Akenine-Möller T (2015) Pixel merge unit. Eurographics (Short Papers), pp 53–56
14. Mcferron T, Lake A Checkerboard rendering for real-time upscaling on intel® integrated graphics
15. Stengel M, Grogorick S, Eisemann M, Magnor M (2016) Adaptive image-space sampling for gaze-contingent real-time rendering. IComputer Graphics Forum, vol 35, pp 129–139. Wiley Online Library
16. Nyquist H (1928) Certain topics in telegraph transmission theory. Trans Am Inst Electr Eng 47(2):617–644
17. Ahmed N, Natarajan T, Rao KR (1974) Discrete cosine transform. IEEE Trans Comput 100(1):90–93
18. Akeley K (1993) Reality engine graphics. In: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques, pp 109–116. ACM

19. Wang Z, Bovik AC, Sheikh HR, Simoncelli EP et al (2004) Image quality assessment: from error visibility to structural similarity. IEEE Trans Image Process 13(4):600–612
20. Gao X, Lu W, Tao D, Li X (2009) Image quality assessment based on multiscale geometric analysis. IEEE Trans Image Process 18(7):1409–1423
21. Ma Q, Zhang L, Wang B (2010) New strategy for image and video quality assessment. J Electron Imaging 19(1):011019
22. Flynn JR, Ward S, Abich J, Poole D (2013) Image quality assessment using the ssim and the just noticeable difference paradigm. In: International Conference on Engineering Psychology and Cognitive Ergonomics, pp 23–30. Springer
23. Rao KR, Yip P (2014) Discrete cosine transform: algorithms, advantages, applications. Academic Press
24. Sihvo T, Niittylahti J (2005) Row-column decomposition based 2d transform optimization on subword parallel processors. International Symposium on Signals, Circuits and Systems, 2005. ISSCS 2005, vol 1, pp 99–102. IEEE
25. Synopsys: DesignWare 2D DCT. Accessed = 2022-02-11 (2021). https://www.synopsys.com/dw/ipdir.php?c=DW_dct_2d
26. Google: Google Play. Accessed = 2022-02-11 (2008). https://play.google.com
27. Google: GAPID (Graphics API Debugger). Accessed = 2022-02-11 (2019). https://developers.google.com/vr/develop/unity/gapid
28. 3D, M.: Gallium3D. Accessed = 2022-02-11 (2009). https://www.freedesktop.org/wiki/Software/gallium
29. Arnau J-M, Parcerisa J-M, Xekalakis P (2013) Teapot: a toolset for evaluating performance, power and image quality on mobile graphics systems. In: Proceedings of the 27th International ACM Conference on Supercomputing, pp 37–46. ACM
30. ARM: ARM Mali-450 GPU. Accessed = 2022-02-11 (2012). https://developer.arm.com/products/graphics-and-multimedia/mali-gpus/mali-450-gpu
31. Synopsys: Synopsys. Accessed = 2022-02-11 (1986). https://synopsys.com
32. Goldstein E.B, Brockmole J (2016) Sensation and Perception. Cengage Learning
33. Janzen BF, Teather RJ (2014) Is 60 fps better than 30? The impact of frame rate and latency on moving target selection. Proceedings of the Extended Abstracts of the 32nd Annual ACM Conference on Human Factors in Computing Systems, pp 1477–1482. ACM
34. Debattista K, Bugeja, Spina S, Bashford-Rogers T, Hulusic V (2018) Frame rate versus resolution: a subjective evaluation of spatiotemporal perceived quality under varying computational budgets. Computer Graphics Forum, vol 37, pp 363–374. Wiley Online Library
35. Mallett I, Yuksel C (2018) Deferred adaptive compute shading. Proceedings of the Conference on High-Performance Graphics, pp 1-4. ACM

Authors and Affiliations

Martí Anglada[1] · **Enrique de Lucas[2]** · **Joan-Manuel Parcerisa[1]** · **Juan L. Aragón[3]** · **Antonio González[1]**

Enrique de Lucas
enrique.delucas@imgtec.com

Joan-Manuel Parcerisa
jmanel@ac.upc.edu

Juan L. Aragón
jlaragon@um.es

Antonio González
antonio@ac.upc.edu

1   Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Jordi Girona 1-3, Barcelona 08034, Spain

2   Imagination Technologies, Imagination House, King's Langley WD4 8LZ, UK

3   Dept. Ingeniería y Tecnología de Computadores, Universidad de Murcia, Campus de Espinardo, Murcia 30100, Spain