LSIM: Ultra Lightweight Similarity Measurement for Mobile Graphics Applications

Yu-Chuan Chang[1], Wei-Ming Chen[1,2], Pi-Cheng Hsiu[2], Yen-Yu Lin[2], Tei-Wei Kuo[1,2]
[1] Department of Computer Science and Information Engineering, National Taiwan University, Taiwan
[2] Research Center for Information Technology Innovation, Academia Sinica, Taiwan
{r05922057, d04922006}@csie.ntu.edu.tw, {pchsiu, yylin}@citi.sinica.edu.tw, ktw@csie.ntu.edu.tw

ABSTRACT

Perceptual similarity measurement allows mobile applications to eliminate unnecessary computations without compromising visual experience. Existing pixel-wise measures incur significant overhead with increasing display resolutions and frame rates. This paper presents an ultra lightweight similarity measure called LSIM, which assesses the similarity between frames based on the transformation matrices of graphics objects. To evaluate its efficacy, we integrate LSIM into the Open Graphics Library and conduct experiments on an Android smartphone with various mobile 3D games. The results show that LSIM is highly correlated with the most widely used pixel-wise measure SSIM, yet three to five orders of magnitude faster. We also apply LSIM to a CPU-GPU governor to suppress the rendering of similar frames, thereby further reducing computation energy consumption by up to 27.3% while maintaining satisfactory visual quality.

CCS CONCEPTS

• **Computer systems organization → Embedded software**; • **Computing methodologies →** *Graphics processors*; *Perception*;

KEYWORDS

Similarity measurement, perceptual quality, energy savings, graphics applications, mobile devices

1 INTRODUCTION

Graphics-intensive applications, especially mobile games, are the most popular class of applications on smartphones. With the rapid growth of hardware computing power and display resolution, the frames in these applications look more realistic and appealing. However, increased frame rates and resolutions lead to higher energy consumption. Because users do not always perceive every frame update [9], reducing the number of similar frames is a sensible way to maximize resource usage efficiency. *Content similarity measurement* has many usage scenarios [10, 13]. For instance, a cloud game service can examine the quality of video streaming sent to clients to control network resources [13]. It can also be used to optimize algorithms and parameter settings of image processing systems, such as video encoders [14]. Recently, it has been used to reduce the computation workloads of deep neural networks by reusing the intermediate results of the previous frame as the result of the current frame when two frames are similar [7].

An intuitive way to determine whether two frames are identical is to compare their RGB values pixel by pixel. However, because graphics-intensive applications run at high frame rates, the direct comparison of image pixels is highly burdensome due to the high display resolution of mobile devices. To reduce computational cost while maintaining acceptable comparison accuracy, Kim et al. [9] proposed a grid-based comparison scheme, where the center pixel of each grid is regarded as the RGB values of the pixels in the grid. Subsequently, Chen et al. [4] found that in graphics applications, each object in a frame is rendered via several function calls provided by the graphics library, and two frames are identical if their function call parameters are exactly the same.

Users may not perceive every frame update, and thus tolerate small differences between frames. Various content similarity measures have been proposed. The peak signal-to-noise ratio (PSNR) [12] is a simple measure that computes the mean squared error but does not comply with the human visual system [15]. Under the hypothesis that human visual perception is highly adapted for extracting structural information from a scene, the *Structural Similarity Index Measurement* (SSIM) [16] assesses image similarity based on the degradation of structural information. SSIM is widely used to assess image similarity but involves computationally intensive pixel-level image processing and is inapplicable for real-time graphics-intensive applications. Inspired by an observation that a majority of visual information is conveyed by patterns of contrasts from brightness changes, Hwang et al. [3] exploited the differences in the luminance values between two frames to approximate SSIM. Inevitably, the computational overhead of these pixel-wise measures increases with the increasingly-high resolution and frame rate.

In this paper, we present an ultra lightweight similarity measure, called LSIM, for mobile graphics applications. LSIM assesses the similarity between frames based on the function call parameters that render virtual objects. The notion is inspired by the following observations. First, for graphics applications, a scene frame is composed of several objects, and the number of objects is much smaller than the number of pixels. Object-wise similarity measurement could be very lightweight. Second, objects are rendered one by one via graphics function calls, and the content changes between frames are mainly caused by the transformation of individual objects, including *translation*, *rotation*, and *scaling*. The transformation determines an object's *position*, *orientation*, and *size* as perceived by the human eye on the screen.

Realizing this lightweight similarity measurement raises several design challenges. First, obtaining the magnitude of an object's transformation between frames is a major challenge. Because an object is translated, rotated, and scaled according to its *transformation matrix*, whose elements are the parameters of function calls invoked in the graphics library, we decompose the transformation matrix to obtain the object's *position*, *orientation*, and *size* on the device screen. Another challenge is to assess the content similarity in line with the perception of the human eye. We extend a pixel-wise measure [8], which quantifies the impact of object transformation on human perception, to assess the similarity between two frames based on objects. Finally, to evaluate the efficacy of LSIM, we implemented it in Android and conducted experiments on a Google Nexus 5x smartphone using popular graphics gaming apps. The experimental results show that LSIM is highly correlated with SSIM. However, the time and energy required for LSIM to compare a pair of frames are respectively less than 0.3ms and 5mJ, while SSIM requires more than 1600ms and 6000mJ. In addition, we

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317856

applied LSIM to a CPU-GPU governing framework, namely UCCG [4], and replaced the frame comparison procedure to reduce similar frames instead of identical frames. This modified governor, denoted by UCCG_L, can further reduce computation energy consumption by up to 27.3% while maintaining satisfactory visual quality (SSIM ≥ 0.98 [2]).

The remainder of this paper is organized as follows. Section 2 provides some background information and the research motivation. We then present the design and rationale behind LSIM in Section 3. The experimental results are reported in Section 4. Section 5 contains some concluding remarks.

2 BACKGROUND AND MOTIVATION

2.1 Graphics Rendering

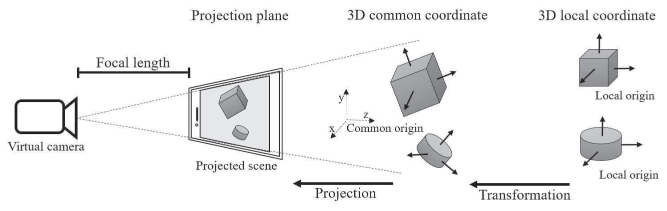


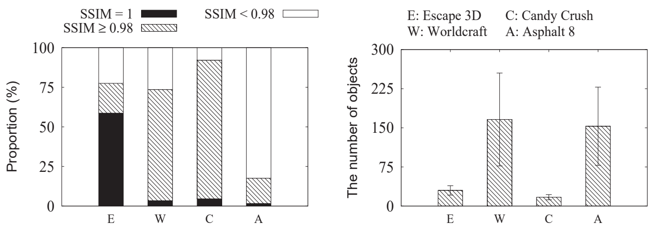
Figure 1: Object construction, transformation, and projection

To render a 3D frame on the device screen, a mobile system requires a sequence of steps in computer graphics called the *graphics pipeline*, where a GPU is normally employed for acceleration via function calls to graphics libraries, such as OpenGL ES. A graphic frame comprises 3D objects that have a variety of shapes. First, each object is constructed around its own *3D local coordinate system* which does not consider position, size, or orientation in relation to other objects in the frame. Then, all the objects are brought together to form a 3D scene in the *3D common coordinate system*. This process is called *transformation*. Next, the 3D scene is projected onto the 2D common coordinate system for display on the flat screen. This process is called *projection*. Figure 1 shows how two objects are constructed, transformed and projected to the screen. The virtual camera, which is imagined to take and display a view of a 3D scene, is set at the origin of the 3D common coordinate system; consequently, the 3D scene projected on the 2D screen is affected by the *focal length* between the camera and the projection plane. When the plane comes closer to the camera, more scene content is visible on the screen, yet the objects look smaller on the screen, and vice versa. Finally, the projected scene is written into the *frame buffer* and displayed on the screen by the display driver.

Object transformation contains a sequence of operations, including *translation*, *rotation*, and *scaling*, which determine an object's attributes, respectively its *position*, *orientation*, and *size* in a scene frame. In the 3D common coordinate system, translation changes an object's position by moving the origin of its local coordinate system, rotation adjusts its orientation by rotating the axes of its local coordinate system, and scaling modulates the scale in each of its local coordinate axes to stretch or shrink the object size. In graphics rendering, each operation on an object is a linear map represented by a matrix, and all the operations are combined into a single *transformation matrix* by matrix multiplication. Note that every operation is relative to the origin of the common coordinate system, so different orders of operations may lead to different transformation results. Object projection is also represented by a matrix and incorporated into the transformation matrix by matrix multiplication. Accordingly, each object has a corresponding matrix to transform and project it from its 3D local coordinate system onto the 2D common coordinate system. Two frames are identical if they have the same objects, and for each object, their matrices are the same. Furthermore, the difference between two frames may not be perceivable if they have *similar* object matrices.

2.2 Characteristic Profiling

To investigate the similarity between adjacent frames rendered in graphics-intensive applications, we conducted an experiment on a Google Nexus 5x smartphone with four 3D games running at up to 60 FPS: Can You Escape 3D, Worldcraft, Candy Crush, and Asphalt 8. The smartphone features an LCD display with a Full HD resolution of 1920×1080 pixels. To ensure reproducible results, we used a recorder called RepetiTouch to record and replay all the touch events and recorded all rendered frames through the `screenrecord` service. The service output a video clip encoded in MPEG-4, which was later converted into a sequence of JPEG images. To assess the similarity between two adjacent frames, we used SSIM [16], a measure specially designed to comply with human visual perception and widely used to assess image similarity. The resultant SSIM score is a decimal value between -1 and 1, where 1 indicates two identical images and a score above 0.98 indicates high visual similarity [2, 5].



(a) SSIM scores of adjacent frames (b) Number of objects per frame

Figure 2: Characteristic profiling of graphics applications

Figure 2(a) shows the proportions of SSIM scores between every pair of adjacent frames in the investigated games. We observed that a large number of frames are similar to their preceding frames (i.e., SSIM≥ 0.98) and sometimes even identical, but these frame updates may not be perceived by the human eye. To save computation energy, redrawing highly-similar frames is unnecessary because the previously-rendered frame in the frame buffer is similar to them and could be displayed instead to be perceived by the user again. However, using a conventional similarity measure like SSIM to detect unnecessary frame rendering involves computationally intensive pixel-wise image processing. Due to the frequent frame rendering of online graphics applications, the similarity measurement should be applicable at runtime with a justifiable computational overhead. We observed that the display content updates are mainly caused by object changes, and the number of objects (typically, a few dozens to hundreds in a frame) is far smaller than the number of pixels (1920×1080 or higher on modern smartphones), as shown in Figure 2(b). These observations inspire us to develop an ultra lightweight similarity measure based on object transformation.

3 LIGHTWEIGHT SIMILARITY MEASUREMENT

3.1 Design Overview

In mobile graphics applications, content updates are mainly caused by changes to objects, and the magnitude of change between two frames can be derived from the *transformation matrices* that specify the objects. Figure 3 shows a flowchart of the proposed similarity measure called LSIM. Let A and B denote two given frames with n objects[1] to be rendered by the GPU. Each object is transformed and projected according to its transformation matrix, which *implicitly* specifies the object's attributes including position, orientation, and size information. First, we individually decompose each object's matrix to obtain its three attributes in each of the two frames. Then, based on the attributes, we compute the magnitude of change to the object, where the object's change in translation, rotation, and scaling is respectively quantified by the differences between the object's positions, orientations, and sizes

[1]Two frames are deemed dissimilar if they have different numbers of objects because objects which appear or disappear suddenly are usually perceivable.

in the two frames. Finally, we calculate an overall similarity score of the two frames based on the magnitude of object change between them. The proposed measure raises several design challenges. One major challenge is to decompose an object's transformation matrix into its three attributes. Another challenge is to compute the magnitude of object change and quantify the similarity between two frames. We address these challenges in Sections 3.2 and 3.3 respectively.

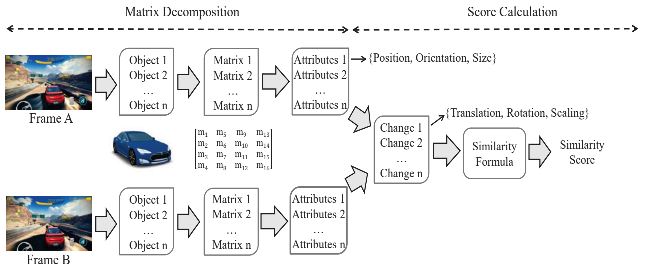


Figure 3: Flowchart of LSIM

3.2 Transformation Matrix Decomposition

Computer graphics usually use *homogeneous coordinates* to perform transformation and projection operations, and these operations on a single object are combined into a single transformation matrix. Homogeneous coordinates represent N-dimensional coordinates with $N+1$ numbers; that is, a 3D point (x, y, z) is represented by (x, y, z, w), where $w = 1$ permanently. For instance, OpenGL uses the equation below to transform each point of an object from its 3D local coordinate system into the 2D common coordinate system:

$$\begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \\ \hat{w} \end{bmatrix}, \tag{1}$$

where the 4×4 matrix is the transformation matrix, $(x, y, z, 1)$ is the object's point in its own 3D local coordinate system, and $(\hat{x}, \hat{y}, \hat{z}, \hat{w})$ is the transformed object's point in the 2D common coordinate system. Note that $\hat{w}$ needs to be converted back to 1 by dividing all dimensions by $\hat{w}$, so that $(\frac{\hat{x}}{\hat{w}}, \frac{\hat{y}}{\hat{w}})$ indicates the point's position on the screen while $\frac{\hat{z}}{\hat{w}}$ indicates the point's depth in the screen. Now, we present how to decompose an object's transformation matrix to obtain its position, orientation, and size information.

3.2.1 Object Position. An object's position is represented by its origin on the screen. In the transformation matrix, m_{13}, m_{14}, and m_{15} are related to translation. Specifically, an object is translated along the x-, y-, and z-axis of the 3D common coordinate system and then projected onto the 2D common coordinate system according to the three elements, where m_{13} and m_{14} change the object's position and m_{15} changes the object's depth. Figure 4 shows a simple example, where the object's local coordinate origin is at the 3D common coordinate origin before translation. If m_{13} is assigned 0.5, the object's origin will be moved along the positive x-axis of the 2D common coordinate system by 0.5 after translation and projection. Then, the object is *linearly* mapped to the *screen coordinate system* to be displayed on the screen. The screen coordinates range from (0,0) to the screen width and height (in pixels).

An object comprises a set of 3D points and is represented by its origin $(0, 0, 0, 1)$ in the local coordinate system. To obtain an object's position on the screen, we first assign the object's local coordinate origin to Equation (1) to obtain the transformed object's position $(m_{13}, m_{14}, m_{15}, m_{16})$ in the 2D common coordinate system. Next, to convert $\hat{w}$ back to 1, we divide all dimensions by $\hat{w}$ and obtain $(\frac{m_{13}}{m_{16}}, \frac{m_{14}}{m_{16}}, \frac{m_{15}}{m_{16}}, 1)$, where

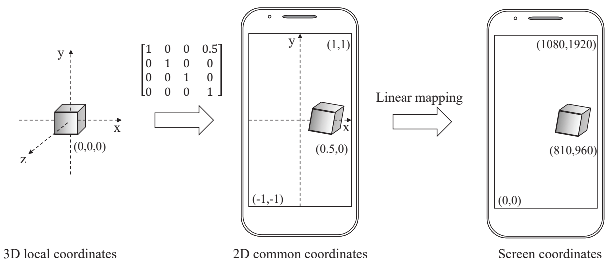


Figure 4: An illustration of object translation

$(\frac{m_{13}}{m_{16}}, \frac{m_{14}}{m_{16}})$ indicates the object's position on the screen and $\frac{m_{15}}{m_{16}}$ indicates the object's depth. Finally, we transform $(\frac{m_{13}}{m_{16}}, \frac{m_{14}}{m_{16}})$ into the screen coordinates $(\overline{x}, \overline{y})$ by simple linear mapping. The screen coordinates $(\overline{x}, \overline{y})$ are the object's position on the screen.

3.2.2 Object Orientation. Extracting an object's rotation from the transformation matrix is a bit tricky, because the rotation, scaling, and projection operations are mixed up in the upper-left 3×3 section of the matrix in Equation (1). Fortunately, we observe that graphics rendering always applies the rotation, scaling, and projection operations in a certain multiplication order, as follows:

$$P \times R \times L = \begin{bmatrix} p_x & 0 & 0 & 0 \\ 0 & p_y & 0 & 0 \\ 0 & 0 & p_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} r_1 & r_4 & r_7 & 0 \\ r_2 & r_5 & r_8 & 0 \\ r_3 & r_6 & r_9 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} l_x & 0 & 0 & 0 \\ 0 & l_y & 0 & 0 \\ 0 & 0 & l_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{2}$$

where L is a scaling matrix to scale a local object, R is a rotation matrix to orientate an object, and P is the scaling matrix used by the projection operation to scale all objects. The rotation matrix R is an *orthogonal matrix* comprising three orthogonal unit vectors. These three vectors respectively specify the new directions to which the x-, y-, and z-axis of the local coordinate system point after rotation. Specifically, the three local coordinate axes are respectively changed to (r_1, r_2, r_3), (r_4, r_5, r_6), and (r_7, r_8, r_9) after rotation. The scaling matrix L is a *diagonal matrix*, where l_x, l_y, and l_z respectively specify the object's scaling ratios along the x-,y-, and z-axis in the common coordinate system. Similarly, the projection operation can also scale an object with the scaling matrix P, but it scales all objects in the scene together instead of a single object.

Based on the above observation, we extract the rotation matrix R according to some properties of orthogonal and diagonal matrices. The multiplication result of $P \times R \times L$ is written as

$$P \times R \times L = \begin{bmatrix} p_x l_x r_1 & p_x l_y r_4 & p_x l_z r_7 & 0 \\ p_y l_x r_2 & p_y l_y r_5 & p_y l_z r_8 & 0 \\ p_z l_x r_3 & p_z l_y r_6 & p_z l_z r_9 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3}$$

where the upper-left 3×3 section is the same as the counterpart of the transformation matrix in Equation (1). The rotation matrix is an orthogonal matrix whose inverse is equal to its transpose, while either scaling matrix is a diagonal matrix whose inverse is the reciprocals of elements of the main diagonal. Therefore, the multiplication result of $(L^{-1} \times R^{-1} \times P^{-1})^T$ can be written as

$$(L^{-1} \times R^{-1} \times P^{-1})^T = \begin{bmatrix} \frac{r_1}{p_x l_x} & \frac{r_4}{p_x l_y} & \frac{r_7}{p_x l_z} & 0 \\ \frac{r_2}{p_y l_x} & \frac{r_5}{p_y l_y} & \frac{r_8}{p_y l_z} & 0 \\ \frac{r_3}{p_z l_x} & \frac{r_6}{p_z l_y} & \frac{r_9}{p_z l_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{4}$$

Note that because $(L^{-1} \times R^{-1} \times P^{-1})^T = [(P \times R \times L)^{-1}]^T$, we can obtain the upper-left 3×3 elements in Equation (4) by simply inverting and then transposing the corresponding elements of the transformation matrix in Equation (1). Interestingly, because scaling ratios are often positive, if we multiply each element of the matrix in Equation (3) with the corresponding element of the matrix in Equation (4), and then take the square root with the sign of each resultant element assigned as that of the corresponding, we can obtain the rotation matrix R of the object[2].

3.2.3 Object Size. To obtain an object's size, we derive the object's scaling ratios along the x-,y-, and z-axis when it is displayed on the screen. During graphics rendering, the scaling matrices L and P scale the object along the x-, y-, z-axis when it is transformed from its local coordinate system to the 3D common coordinate system and then projected to the 2D common coordinate system, respectively. Finally, the $\hat{w}$ value, which is related to the object's depth, scales the object in the 2D common coordinate system on the screen. The elements with respect to the scaling matrices L and P, as shown in Equation (5), can be obtained by multiplying each element of the matrix $P \times R \times L$ in Equation (3) by the corresponding element of the inverse of the rotation matrix R derived previously in Section 3.2.2.

$$P \times R \times L \circ R^{-1} = \begin{bmatrix} p_x l_x & p_x l_y & p_x l_z & 0 \\ p_y l_x & p_y l_y & p_y l_z & 0 \\ p_z l_x & p_z l_y & p_z l_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad (5)$$

where l_x, l_y, l_z are respectively the scaling ratios of the object along the x-,y-, and z-axis in the 3D common coordinate system, while p_x, p_y, p_z are the scaling ratios applied when all objects in the same scene are projected altogether onto the 2D common coordinate system.

We observe that transformation would not change an object's shape. Normally, the object's shape created in its 3D local coordinate system will not change in the 3D common coordinate system. Thus, L can be regarded as a *uniform matrix*, where the scaling ratios along the three axes are the same (i.e., $l_x = l_y = l_z = l$), and the compound scaling ratios of L and P along the x-, y-, z-axis after projection are respectively $p_x l$, $p_y l$, and $p_z l$. In homogeneous coordinates, the $\hat{w}$ value is used to make a closer object look larger on the screen, so its value depends on the object's depth. Recall that a transformed point needs to be normalized by dividing all dimensions by $\hat{w}$ such that $\hat{w}$ is converted back to 1; in other words, the object is scaled $\frac{1}{\hat{w}}$ times on the screen, compared with its size in the 2D common coordinate system. As derived previously in Section 3.2.1, $\hat{w} = m_{16}$ of the transformation matrix in Equation (1). In conclusion, an object is scaled respectively by $\{\frac{p_x l}{m_{16}}, \frac{p_y l}{m_{16}}, \frac{p_z l}{m_{16}}\}$ along the x-, y-, and z-axis on the screen, compared with its original size in the local coordinate system.

3.3 Similarity Score Calculation

Now, we present how to calculate the similarity score between two frames A and B based on their objects' attributes, including position $(\overline{x}, \overline{y})$, orientation R, and size $\{\frac{p_x l}{m_{16}}, \frac{p_y l}{m_{16}}, \frac{p_z l}{m_{16}}\}$, obtained in Section 3.2. Prior research has proposed various perceptual quality measures that consider psychological characteristics of the human visual system to score the resemblance between two images. To quantify object transformation, it was found in [8] that the effect of primitive object transformation, including translation, rotation, and scaling, are decisive factors and are mutually independent. Accordingly, the effect of pixel transformation on visual perception was modeled as follows [8]:

$$T(\Delta v^i, \Delta r^i, \{\Delta s_x^i, \Delta s_y^i, \Delta s_z^i\}) + H(i), \qquad (6)$$

where function $T()$ quantifies the effect of the i_{th} pixel's transformation, function $H()$ quantifies the relationship between the pixel's transformation and the other pixels', Δv^i is the visual angle caused by translation, Δr^i is the rotation angle caused by rotation, and $\{\Delta s_x^i, \Delta s_y^i, \Delta s_z^i\}$ are respectively the scaling ratios of size along the x-, y-, and z-axis between the two frames. Note that this formula quantifies the object transformation based on pixels, whereas the basic unit of our measure is an object instead of a pixel. Therefore, we revise the formula[3] while setting the coefficients at their default values in [8] to quantify the resemblance between frames A and B as follows:

$$LSIM(A, B) = \max_{1 \le i \le n} [T(\Delta v^i, \Delta r^i, \{\Delta s_x^i, \Delta s_y^i, \Delta s_z^i\}) + H(i)], \qquad (7)$$

where n is the number of objects in either frame, and i indicates the i_{th} object instead of a pixel. The similarity score of an object is a decimal value between 0 and ∞, where 0 indicates that the two frames are identical. LSIM selects the maximum score among all objects as the similarity score between the two frames because the object with the largest transformation is often most salient to the human eye.

To obtain the parameters required for function $T()$, we use each object's respective positions, orientations, and sizes in frames A and B. Given an object's positions $(\overline{x}, \overline{y})$ in both frames and the eye-to-screen distance[4], we simply calculate the visual angle Δv caused by the object's translation based on a trigonometric function that relates an angle of a triangle to the lengths of its adjacent and opposite sides. The rotation angle Δr of an object is estimated based on the change of the object's rotation matrix R between the two frames. Recall that the rotation matrix comprises an object's direction vectors along the three local coordinate axes in a frame. We calculate the angle between each pair of direction vectors in the two frames according to the law of cosines, and select the maximum among the three angles as Δr because the object in the preceding frame needs to rotate around an arbitrary axis by at least this angle to match its orientation in the current frame. To derive an object's size scaling ratios, we directly divide the object's size $\{\frac{p_x l}{m_{16}}, \frac{p_y l}{m_{16}}, \frac{p_z l}{m_{16}}\}$ in the current frame by its size in the preceding frame. The visual angles, rotation angles, and scaling ratios of all n objects are also used by function $H()$ to quantify the relationship between an object's transformation and the others'. Finally, according to Equation (7), LSIM gives a similarity score between two frames.

4 PERFORMANCE EVALUATION

4.1 Experiment Setup

We integrated LSIM into a graphics library, called OpenGL ES 2.0, supported by Android 6.0 on the Google Nexus 5x smartphone. In OpenGL, whenever a new frame is to be rendered, a function named `eglSwapBuffersWithDamageKHR` will be called to swap the image buffer used for the frame being displayed on the screen and that used for the frame in preparation. Accordingly, we implement LSIM in this function to capture the frame to be rendered. With each object's transformation matrix acquired via `glUniformMatrix4fv` and the frame's number of objects obtained via `glDrawElements`, LSIM calculates the similarity score between the frame to be rendered and the frame being displayed according to the measurement proposed in Section 3.

We conducted a series of experiments on the Google Nexus 5x smartphone with some popular 3D gaming apps. The smartphone features an Adreno 418 GPU that can operate at 6 frequency levels and an LCD

[2]If an object's transformation matrix has different upper-left 3×3 sections in two frames but its rotation matrix cannot be decomposed in this way, we consider the frames dissimilar.

[3]Any pixel-wise similarity measure that quantifies object transformation based on translation, rotation, and scaling could be used instead.

[4]A typical smartphone viewing distance is 12 inches [1, 6].

display with a Full HD resolution of 1920×1080 pixels. To reduce the potential influence of human intervention, we used a recorder called RepetiTouch to record and replay all the touch events for each investigated app. This ensured that the same interaction patterns were reproducible in the experiments. We used the power monitor produced by Monsoon Solutions to measure the smartphone's transient power and energy consumption.

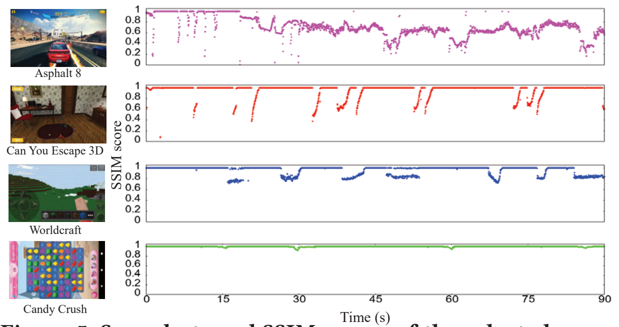


Figure 5: Snapshots and SSIM scores of the selected games

The efficacy of the proposed scheme is mainly determined by two factors: the *frequency* and the *magnitude* of scene change in the 3D game. To exhaustively evaluate our scheme, we chose four representative games, namely Asphalt 8, Can You Escape 3D, Worldcraft, and Candy Crush, such that the frequency and magnitude of scene changes in these games are different. Figure 5 shows some snapshots of the games and the SSIM score between each pair of adjacent frames. Asphalt 8, which is a typical racing game, comprises diverse scenes; thus, the SSIM scores vary frequently and significantly. For Can You Escape 3D, the magnitude of scene change is also significant, but the consecutive scenes are longer and more similar than those in Asphalt 8. In contrast, for Worldcraft, the SSIM scores vary frequently, but the changes are small. Finally, the SSIM scores vary slightly and the changes are small in Candy Crush.

We conducted two sets of experiments to validate the accuracy of LSIM and evaluate its efficacy. First, we validate whether the similarity score given by LSIM is highly correlated to the score given by SSIM [16], whereas the computational overhead of LSIM makes it appropriate for online applications. Then, to evaluate efficacy, we applied LSIM to a CPU-GPU governing framework [4] as a use case study, where the rendering of similar frames is suppressed to save computation energy while maintaining satisfactory perceptual quality.

4.2 Accuracy and Overhead

First, we compare the respective overhead of LSIM and SSIM in terms of the time and energy consumption required to score a pair of frames on the smartphone. To this end, we intercepted the necessary OpenGL function calls to record the transformation matrices of all objects while using Android's built-in API to record all rendered frames. We then used LSIM and SSIM to score each pair of adjacent frames, respectively. To correlate the respective scores given by LSIM and SSIM, we used the *Pearson correlation coefficient* (PCC) [11], a widely used metric to measure the linear correlation between two variables. The resultant PCC is a decimal value between -1 and 1, where an absolute value between 0.7 and 1 indicates a strong correlation[5].

[5]LSIM gives a score of ∞ if the numbers of objects in two frames are different. However, PCC cannot deal with the infinity, so those frames scored infinity by LSIM were excluded.

4.2.1 Overhead Measurement. Table 1 shows the longest time taken by LSIM and SSIM to score two frames on the smartphone. LSIM requires much less time than SSIM, because the time of LSIM depends on the number of objects, while the time of SSIM depends on the number of pixels. Importantly, LSIM requires less than 0.3ms to compare a pair of frames, while the time constraint of rendering a frame is 16.6ms for a 3D game running at 60 FPS. This indicates that LSIM is applicable for online scenarios, but SSIM is not. In addition, the average energy consumed to compare a pair of frames is less than 5mJ for LSIM but greater than 6000mJ for SSIM.

		Escape 3D	Worldcraft	Candy Crush	Asphalt 8
Time (ms)	SSIM	1681.51	1681.89	1604.53	1679.72
	LSIM	0.03	0.23	0.01	0.17
Energy (mJ)	SSIM	6014.80	6090.30	6073.89	6100.30
	LSIM	1.50	4.30	0.94	3.00

Table 1: Computational and energy overhead

4.2.2 Correlation Between LSIM and SSIM. Figure 6 shows the correlation (assessed by PCC) between the scores given by LSIM and SSIM. For all four games, LSIM is highly correlated with SSIM. The result indicates that the content change between adjacent frames is mainly caused by the change of objects, and LSIM is capable of quantifying the magnitude of change. Interestingly, we observe that for some pairs of dissimilar frames, the scores given by LSIM and SSIM are not perfectly linearly correlated. The reason is that LSIM is more sensitive to object transformation than SSIM. This also explains why the linear correlation between LSIM and SSIM in Asphalt 8 is slightly lower than that in other games. Nevertheless, the respective values of PCC between LSIM and SSIM for the four games are 0.910, 0.902, 0.916 and 0.829, all of which are still in the strongly correlated range between 0.7 and 1 [11].

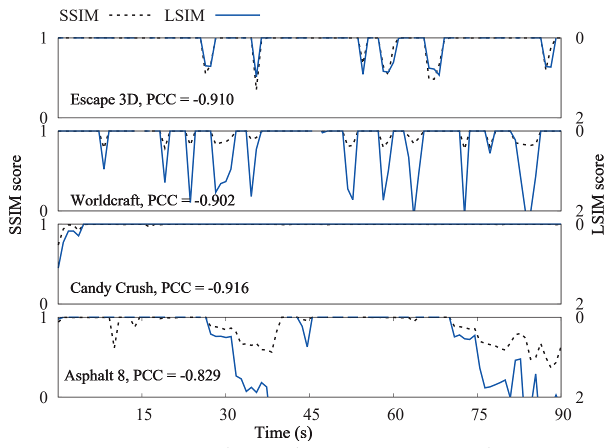


Figure 6: Correlation between LSIM and SSIM

4.3 A Use Case: Energy and Quality Tradeoff

As a use case study, we applied LSIM to a governing framework, called UCCG [4], which dynamically scales up and down the CPU/GPU frequencies to reduce energy consumption while maintaining visual quality. Specifically, UCCG scales down the frequencies when *identical* frames are detected and, consequently, the frame in the image buffer is displayed several times to achieve the screen's *fresh rate*. We replaced UCCG's detection module to suppress the rendering of *similar* frames whose LSIM scores are above 0.0009 (which approximately corresponds to an SSIM score above 0.98, i.e., highly similar, based on our profiling) and denote the modified governor as UCCG$_L$. The governors trade off visual quality for energy savings. Thus, the energy saved by UCCG and UCCG$_L$, compared with that used by Android's governor (denoted as NATIVE) to run a 3D game, was adopted as a performance metric. Moreover, to quantify the perceived quality, we used SSIM to assess the resemblance between each frame rendered by NATIVE and its counterpart under UCCG or UCCG$_L$.

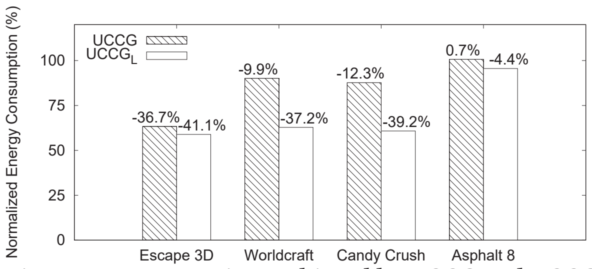


Figure 7: Energy savings achieved by UCCG and UCCG$_L$

4.3.1 Energy Consumption. Figure 7 shows the energy savings achieved by UCCG and UCCG$_L$, where the computation energy consumption is measured by subtracting the energy consumed for the smartphone in the idle state from the energy consumed to run each 3D game. For Escape 3D, both UCCG and UCCG$_L$ can provide considerable energy savings because the game has many identical frames. However, for Worldcraft and Candy Crush, whose respective scenes vary frequently and slightly but the changes are all small, UCCG$_L$ can provide further energy savings over UCCG by avoiding the rendering of a large number of similar frames. Interestingly, for Asphalt 8, whose scenes vary frequently and significantly, UCCG could even increase the energy consumption if the energy savings is offset by its extra computational overhead. By contrast, UCCG$_L$ can still leverage the small number of similar frames to save energy. Overall, compared to UCCG, UCCG$_L$ can further save computation energy by 4.4 to 27.3%, depending on the number of similar frames in the 3D game.

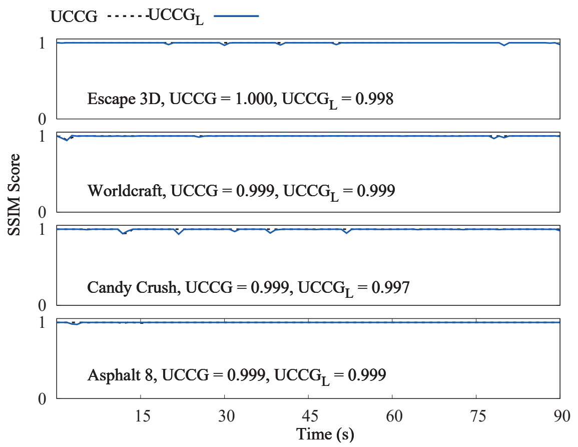


Figure 8: SSIM scores between frames under UCCG and UCCG$_L$

4.3.2 Visual Quality. Figure 8 shows the variations in the visual quality (assessed by the SSIM score, with 1 representing no quality loss) under UCCG and UCCG$_L$. As expected, UCCG causes almost no quality loss because those identical frames, although not rendered, could still be displayed on the screen and perceived by the user. By contrast, UCCG$_L$ achieves relatively worse visual quality for a game comprised of consecutive similar frames, as can be seen by comparing Escape 3D and Candy Crush with Asphalt 8 and Worldcraft. However, the loss is deemed too subtle to be discerned by the human eye or, at least, does not incur adverse interference to the user, because the average SSIM score approaches 1 for every game and the lowest score between any two adjacent frames exceeds 0.98 [2]. To sum up, a CPU/GPU governor like UCCG can use LSIM to ease the energy consumption of similar frames while maintaining satisfactory visual quality for mobile 3D games. Many other applications that require real-time content similarity measurement could also benefit substantially from LSIM.

5 CONCLUDING REMARKS

We have presented LSIM, an ultra lightweight similarity measure for mobile graphics applications. Unlike existing pixel-wise measures, LSIM exploits the transformation metrics of objects. To validate its accuracy and overhead, we integrated LSIM into the OpenGL ES graphics library used in the Google Nexus 5X smartphone. The scores given by LSIM and SSIM are highly correlated, yet LSIM is three to five orders of magnitude faster than SSIM, thereby allowing for real-time frame comparison during graphics rendering. As a use case study, we applied LSIM to a CPU-GPU governor called UCCG [4] to suppress the rendering of similar frames. The smartphone was found to further achieve energy savings of 4.4 to 27.3% while maintaining the visual quality of the played 3D mobile game. To enable LSIM to quantifies not only object transformation but also changes in object colors, future work will seek to extend LSIM to consider the textures bound to each object. We hope more use cases will benefit from LSIM.

ACKNOWLEDGEMENT

This work was supported in part by the Ministry of Science and Technology under grant 107-2628-E-001-001-MY3.

REFERENCES

[1] Y. Bababekova, M. Rosenfield, J. E. Hue, and R. R. Huang. Font Size and Viewing Distance of Handheld Smart Phones. *Optometry & Vision Science*, 88(7):795–797, 2011.
[2] A. Bartolini, M. Ruggiero, and L. Benini. Visual Quality Analysis for Dynamic Backlight Scaling in LCD Systems. In *Proc. of IEEE/ACM DATE*, pages 1428–1433, 2009.
[3] C. Hwang, S. Pushp, C. Koh, J. Yoon, Y. Liu, S. Choi, and J. Song. RAVEN: Perception-aware Optimization of Power Consumption for Mobile Games. In *Proc. of ACM MobiCom*, pages 422–434, 2017.
[4] W.-M. Chen, S.-W. Cheng, and P.-C. Hsiu. A User-Centric CPU-GPU Governing Framework for 3D Mobile Games. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2018.
[5] X. Chen, J. Zheng, Y. Chen, M. Zhao, and C. J. Xue. Quality-retaining OLED Dynamic Voltage Scaling for Video Streaming Applications on Mobile Devices. In *Proc. of IEEE/ACM DAC*, pages 1000–1005, 2012.
[6] P.-C. H. H.-Y. Lin, C.-C. Hung and T.-W. Kuo. Duet: An OLED and GPU Co-management Scheme for Dynamic Resolution Adaptation. In *Proc. of IEEE/ACM DAC*, pages 1–6, 2018.
[7] L. N. Huynh, Y. Lee, and R. K. Balan. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. In *Proc. of ACM MobiSys*, pages 82–95, 2017.
[8] P. Kellnhofer, T. Ritschel, K. Myszkowski, and H.-P. Seidel. Transformation-aware Perceptual Image Metric. *Journal of Electronic Imaging*, 25(5):053014, 2016.
[9] D. Kim, N. Jung, and H. Cha. Content-centric Display Energy Management for Mobile Devices. In *Proc. of IEEE/ACM DAC*, pages 41:1–41:6, 2014.
[10] C.-H. Lin, C.-K. Kang, and P.-C. Hsiu. Catch Your Attention: Quality-retaining Power Saving on Mobile OLED Displays. In *Proc. of IEEE/ACM DAC*, pages 42:1–42:6, 2014.
[11] B. Ratner. The Correlation Coefficient: Its Values Range Between +1/-1, or Do They? *Journal of Targeting, Measurement and Analysis for Marketing*, 17(2):139–142, 2009.
[12] H. R. Sheikh, M. F. Sabir, and A. C. Bovik. A Statistical Evaluation of Recent Full Reference Image Quality Assessment Algorithms. *IEEE Trans. on Image Processing*, 15(11):3440–3451, 2006.
[13] S. Shi, C.-H. Hsu, K. Nahrstedt, and R. Campbell. Using Graphics Rendering Contexts to Enhance the Real-time Video Coding for Mobile Cloud Gaming. In *Proc. of ACM MM*, pages 103–112, 2011.
[14] S. Wang, A. Rehman, Z. Wang, S. Ma, and W. Gao. Perceptual Video Coding Based on SSIM-Inspired Divisive Normalization. *IEEE Trans. on Image Processing*, 22(4):1418–1429, 2013.
[15] Z. Wang, A. C. Bovik, and L. Lu. Why Is Image Quality Assessment So Difficult? In *IEEE ICASSP*, volume 4, pages 3313–3316, 2002.
[16] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image Quality Assessment: From Error Visibility to Structural Similarity. *IEEE Trans. on Image Processing*, 13(4):600–612, 2004.