EG UK Computer Graphics & Visual Computing (2016)
Cagatay Turkay and Tao R. Wan (Editors)

Interactive GPU-based Image Deformation for Mobile Devices

Jan Ole Vollmer, Matthias Trapp, and Jürgen Döllner

Hasso Plattner Institute, University of Potsdam, Germany

Abstract

Interactive image deformation is an important feature of modern image processing pipelines. It is often used to create caricatures and animation for input images, especially photos. State-of-the-art image deformation techniques are based on transforming vertices of a mesh, which is textured by the input image, using affine transformations such as translation, and scaling. However, the resulting visual quality of the output image depends on the geometric resolution of the mesh. Performing these transformations on the CPU often further inhibits performance and quality. This is especially problematic on mobile devices where the limited computational power reduces the maximum achievable quality. To overcome these issue, we propose the concept of an intermediate deformation buffer that stores deformation information at a resolution independent of the mesh resolution. This allows the combination of a high-resolution buffer with a low-resolution mesh for interactive preview, as well as a high-resolution mesh to export the final image. Further, we present a fully GPU-based implementation of this concept, taking advantage of modern OpenGL ES features, such as compute shaders.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1. Introduction

Image deformation or image warping, i.e., the manipulation of an image to correct distortions or creatively apply new ones, is a fundamental functionality in image processing. Image deformation often relies on an *deformation mesh* to enable non-destructive operations. Further, such a mesh offers maximum control, is simple to use, easy to exchange and can be implemented using graphics processing units (GPUs) [YCB05]. Such mesh-based deformation displaces the mesh vertices and subsequently textures the deformed mesh with the input image. In contrast to fully automatic deformation approaches, this paper focuses on mesh-based deformation on mobile devices that enables a user to interactively deform an image using translation and scale transformations.

Problem Statement. The implementation of interactive image deformation on mobile devices presents a classic time vs. quality trade-off. On the one hand, a high resolution of the underlying deformation mesh is required to maximize the resulting visual quality of the deformed image, which results in a high number of vertices to be processed, thus increasing the geometric complexity. This is especially important in times of ever-increasing camera resolutions of mobile devices which demand a corresponding increase of mesh resolutions to maintain an acceptable visual quality. On the other hand, mobile devices only provide comparatively low computational power that limits the maximum number of vertices processable at interactive frame rates. In addition, power consumption is an issue here, as excessive computations typically quickly drain a device's battery. As the interactivity is a hard constraint, the only remaining adjustment option is to reduce the mesh resolution and thus the visual quality accordingly.

Traditional approaches represent the deformation mesh using an array of vertices. For every deformation step, the CPU performs deformation operations on the vertex array which is subsequently transferred to the GPU to render the mesh, thereby applying the deformation to the input image. This approach imposes additional performance issues, as (1) the CPU-based deformation is slow compared to a GPU-based approach and (2) update latencies occur due to the required data transfer from CPU memory to GPU memory, depending on the geometric complexity of the mesh. Both further reduce the maximum number of vertices (and thus, the resulting visual quality) that can be processed while maintaining an interactive frame rate.

Contributions. To overcome the problem of limited visual quality, we propose the following novel approach: We introduce an intermediate *deformation buffer* which encodes vertex displacements at a high resolution that is independent of the deformation mesh's resolution. Instead of applying deformation operations directly to the mesh, our approach applies these to the deformation buffer. Subsequently, a deformation mesh with equal or lower resolution is rendered using the displacement information stored in the deformation buffer. This decoupling of buffer and mesh resolution allows two modes of operation: (1) an interactive mode using a low-resolution deformation mesh and (2) a high-quality mode us-

DOI: 10.2312/cgvc.20161303

ing a high-resolution mesh that can be used to export the final image with maximum visual quality. In addition, the buffer-based encoding of deformations supports persistence and exchange of *deformation templates* via standard image file formats.

To overcome the performance issues of CPU-based implementations, our technique is designed to fully utilize GPUs to apply deformation operations. The approach supports standard application functionalities, such as undo/redo, deformation operation parametrization, as well as import/export of deformations. We prototypically implemented our approach based on OpenGL ES [Sim15] for demonstration and evaluation purposes. To summarize, this paper presents the following contributions: (1) a novel image deformations concept based on an intermediate deformation buffer that decouples buffer and mesh resolutions; and (2) a fully GPU-based realization of this concept that performs faster than traditional CPU-based implementations.

2. Related Work

This section covers related and previous work in GPU-based image deformation. Beside global deformations described by Barr [Bar84], free-form deformation introduced by Sederberg and Parry [SP86] is one of the first deformation approaches. This popular technique has been extended and adapted in numerous ways. The concept is to deform a separate deformation object and compute the new coordinates of the affected object (e.g., a deformation mesh) based on the local coordinates of the deformed object. They used a lattice structure composed of tricubic Bezier patches as deformation object. Further, Akleman [Akl97] used a series of lines as control structure to construct his deformation template. The approach was inspired by a transformation technique introduced by Beier and Neely [BN92] denoted as feature-based image metamorphosis. It utilizes one or more simple line shapes to effectively warp or deform pixel locations based on a technique called field morphing. Although the final transformation is in many ways different from a deformation, the underlying feature-based transformation algorithm shares similarities with deformation algorithms. Later, Akleman et al. [APL00] presented a new deformation tool for making extreme deformations based on simplistic complexes. It is based on implicit free-form deformations (IFFD) proposed by Crespin [Cre99]. Instead of simple lines, Akleman used deformation primitives like triangles, which enable a more flexible deformation control. Based on IFFD, a set of these deformation primitives can be applied to the underlying geometry, each affecting only a certain part of the target object. All the above deformation techniques rely on some kind of control structure that is aligned to the target object. A deformation of one or more of these control objects causes a deformation of the target object. Our technique does not feature control structures, but instead enables the user to manipulate the image directly using touch gestures.

The implementations of most of the discussed work do not used GPUs or focus on desktop consumer graphics hardware and do not account for the limitation of mobile devices. Our work is mainly influenced by the concept of camera textures [SBGS06]. The authors use an image-based representation of deformations applied in 3D camera space. We extend this idea by focusing on manipulating this representation interactively and applying it to a 2D deformation mesh. In [CS07], a sophisticated overview for mesh deformation utilizing programmable shaders in 3D is presented. The authors distinguish between procedurally defined deformations and deformation textures. With respect to this, our approach can be considered hybrid: it uses a combination of procedural and image-based deformation operations modifying a deformation texture that is used for deformation rendering. Further, in [MF12a,MF12b], a real-time deformation approach using the transform feedback mechanism of GPUs is presented and applied to physically-based deformation models. We consider the concepts of this work as a possible implementation approach and adapt it to GPU-based processing for mobile devices.

As alternative to deformations created using a combination of scaling and translation operations, GPU-based implementations of cage-based and spline-based methods [SF98] are relevant to our work. In [LYNH14], a GPU-accelerated image deformation technique based on splines is presented. The authors propose a parallel implementation using CUDA. Instead of using proprietary APIs, we focus on using the standardized and open OpenGL ES API. Meng et al. present an approach for multi-cage image deformation on the GPU [MZDP11] that enables the deformation of specific regions-of-interest while keeping local details. We incorporate a similar feature by enabling a user to define masks which can control the amount of applied deformations.

3. Conceptual Overview

This section describes a general conceptual overview and major components of the presented image deformation concept. This concept can be implemented in various ways based on OpenGL (ES) and the OpenGL (ES) Shading Language as well as modern Direct3D APIs, not limited to embedded systems. A variant based on OpenGL ES is discussed in Section 4.

3.1. Overview of Rendering Pipeline

Figure 1 (next page) shows an simplified overview of our deformation concept. Independent of the actual implementation, it uses the following data components:

Input Image: An image or photograph represented by a *2D texture object* that is uploaded to video memory only once during the complete deformation process.

Deformation Buffer: Similar to a 2D gridded mesh, this buffer stores *deformation vectors* that encode the displacement w.r.t. a normalized image space. These vectors can be interpolated linearly during fetching.

Deformation Mesh: To deform the input image based on the deformation buffer, a 2D triangulated grid mesh with texture coordinates is required to drive the deformation, i.e., color interpolation. Its vertices are displaced using the deformation vectors stored in the deformation buffer.

Deformed Image: As the result of the deformation application stage, this image is presented to the user for visual feedback of a particular deformation operation.

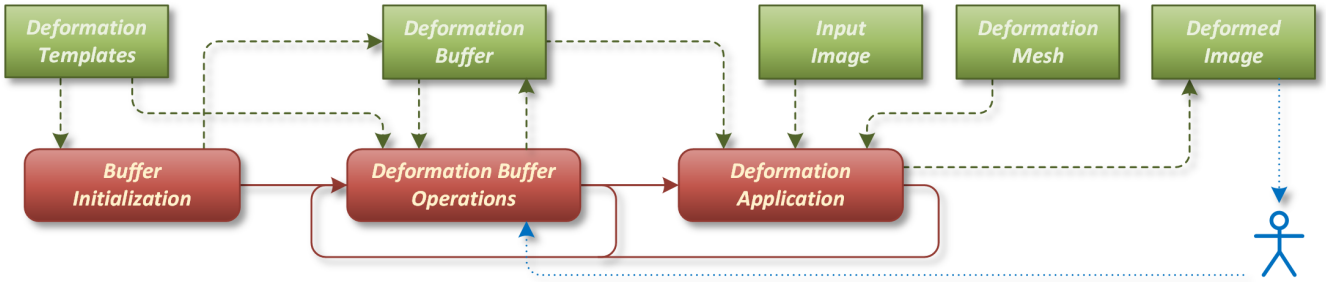


Figure 1: Conceptual overview of interactive GPU-based rendering technique for image deformation. It shows the data flow (green lines) between the data components (green) manipulated by operations (red).

(a) Original image. (b) Translate only. (c) Scale only.

Figure 2: Exemplary application of translation (b) and scaling deformation operators (c) to an input image (a).

(a) Initial. (b) Translation. (c) Rotation.

Figure 3: Exemplary scaling effect using symmetric deformation settings with different reference coordinate systems: (a) standard reference system, (b) translated, and (c) translated and rotated reference system.

3.2. Deformation Buffer Operations

The system supports several procedural deformation operations that modify the deformation buffer in different ways. The general parametrization for these operations includes a deformation *radius* $r \in [0,1] \subset \mathbb{R}$, a deformation *strength factor* $s \in \mathbb{R}_0^+$, and the deformation *origin* $o \in \mathbb{NDC} := [-1,1]^2 \subset \mathbb{R}^2$ corresponding to the location of the touch gesture. A Hermite interpolation function $H(a_0, a_1, x) \rightarrow [0,1] \subset \mathbb{R}$ with the edges a_0 and a_1 ($a_0 < a_1$) is used to smooth transitions between deformed and undeformed areas. Figure 2 shows the supported deformation operations *translation* and *scale*. An additional *repair* operation is supported as well. The operations are performed by modifying each deformation vector $\vec{v}$ separately as follows: The impact $p \in [0,1] \subset \mathbb{R}$ for all operations is computed as $p = s \cdot (1 - H(0, r, \|\vec{v} - o\|))$ and subsequently used in each of the following deformation operations:

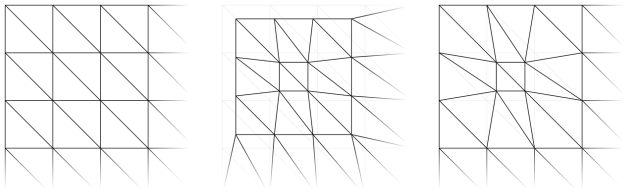
Translation: Move the vertex along the normalized input direction $\vec{d} \in \mathbb{NDC}$, $\|\vec{d}\| = 1$:
$\vec{v}' := \vec{v} + p \cdot \vec{d}$ (Figure 2b).
Scale: Move the vertex further away from the origin:
$\vec{v}' := \vec{v} + p \cdot (\vec{v} - o)$ (Figure 2c).
Repair: Restore the vertex to its original, undeformed position $\vec{v}_0$:
$\vec{v}' := (1 - p) \cdot \vec{v} + p \cdot \vec{v}_0$

Further, possible high-level deformation operations include *interpolation* between two deformation textures as well as *undo* and *redo*. Interpolation is performed by vector-wise linear interpolation, while undo or redo replaces the deformation buffer contents using transfer operations supported by the rendering hardware.

Symmetric Deformation Operations. To support more advanced deformation editing, e.g., for face deformations, *symmetric* operations are supported by mirroring the deformation origin with respect to the particular axes of a *reference coordinate system* represented by an orthonormal basis. Thus, a user can choose which axes should be used for symmetric deformations, i.e., horizontal and/or vertical symmetry, respectively. Figure 3 shows a comparison of scaling deformation operation using a single axis symmetry, each with a different reference coordinate system. The reference coordinate system can be adjusted using touch gestures. Initially, it is set to the screen center with zero rotation angle.

Constraining Deformation Operations. The impact of the deformation operations described above can be controlled using *global* and *local constraints*. The *axis lock* constraint supports the user in controlling the global application of deformation operations by limiting them to a single axis of the reference coordinate system. The remaining axis is locked, i.e., the values stored in the deformation buffers are not modified with respect to this axis.

In addition to these global constraint methods, we further support local deformation constraints by encoding a respective *deformation weight* within the deformation buffer. These constraints can comprise the following two options: a *user selection mask* defines areas that should remain fixed or areas that should be deformed instead. The user brushes regions in the deformation buffer by adding or removing mask regions. Subsequently, these are used the control the drop-off of the deformation operations. Further, *border fixation* avoids the introduction of imaging artifacts during interactive de-



(a) Original. (b) No fixation. (c) With fixation.

Figure 4: Impact of deformation weights for border fixation when applying a scaling deformation with a negative factor (c). The border vertices of the an unformed deformation mesh (a) are scaled inwards (b) without taking the deformation weights into account.

formation by fixing the border vertices to avoid the application of deformation operators. Figure 4 shows an applied scaling deformation *without* (4b) and *with* (4c) border fixation.

3.3. Template-Based Deformation

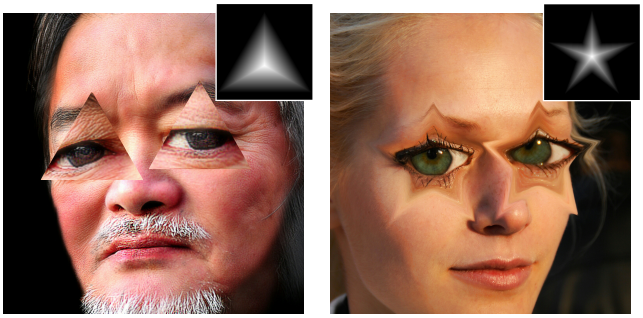
To this extent, the application of deformation operations results in smooth vertex displacement due to the described parametrization, as well as the limitations of touch interfaces (also known as *fat finger problem*). Although tablet devices with pen support can reduce this problem, deformations exhibiting sharp corners, acute angle, or deformation shapes supporting fine detail cannot be achieved using the parametrization provided so far. Therefore, the presented approach can be extended to support user-defined shapes that can be easily created and managed using standard raster drawing software such as GIMP or Photoshop and incorporated into the system.

Deformation templates are represented using 2D textures containing deformation buffer contents (*explicit templates*) or grayscale values (*implicit templates*). For implicit templates, the use of binary masks yield hard edges in the deformation results. In case this is not desired, one can apply implicit templates based on distance fields [Gre07] (Figure 5), which support both, hard edges and smooth drop-offs. The required distance-fields can be computed in a pre-processing step on the CPU or GPU, e.g., using jump flooding [RT06] or the parallel bending algorithm [CTMT10].

An implicit template is applied by changing the operation impact p introduced in Section 3.2 based on a value $q \in [0,1] \subset \mathbb{R}$ sampled from the template image and a user-controllable smoothness factor $u \in (0,1] \subset \mathbb{R}$. The sampling coordinates $c_q \in [0,1]^2 \subset \mathbb{R}^2$ are computed as $c_q := \frac{\vec{v}-o}{r} \cdot \sqrt{2} \cdot 0.5 + 0.5$. Multiplication with $\sqrt{2}$ is required to extend the possible sampling space to include the template's corners. The operation impact is then computed as $p := s \cdot H(0,u,q)$.

4. Real-Time Implementation

This section covers details of the GPU-based implementation. We prototypically implemented the concept described in the previous section based on an Android application using Java (JDK 7) and OpenGL ES 3.1 [Lee15] in combination with the OpenGL ES Shading Language 3.1 [Sim15].



(a) Triangular template ($u = 0.0$). (b) Star template ($u = 0.75$).

Figure 5: Examples of deformation templates (shown in insets).

4.1. Overview of Implementation Variants

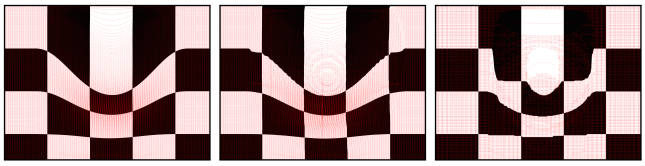
The deformation data must be stored in a way that supports fast write access from shaders in order to implement GPU-based deformation operations. OpenGL ES provides two means of storing data in GPU memory: *textures* (specialized data stores for images, writable via rendering to a framebuffer object or image store operations) and *buffers* (general purpose data stores, writable via transform feedback or shader storage buffer objects). Based on these two storage options, the general concept can be implemented in three different variations on modern graphics hardware, each of them comprising major or minor drawbacks:

Render-to-Texture: The deformation data is stored in a texture and modified by attaching it to a framebuffer object before rendering to it. Since feedback loops are explicitly forbidden by the OpenGL ES specification [Lee15, p. 223-225], two textures are actually required which are used alternately for reading and writing (ping-pong rendering).

Transform Feedback: The deformation data is stored in a buffer and modified using transform feedback [MF12b]. Similar to the render-to-texture approach, two buffers are required as undefined behavior may occur when a transform feedback buffer is used simultaneously for other operations [Lee15, p. 285f], i.e., rasterization in this case.

Compute Shader: The deformation data is stored in the same manner as for the transform feedback approach; however, a single buffer suffices, as compute shaders in combination with shader storage buffer objects allow the direct manipulation of the buffer content. However, this implementation requires OpenGL ES 3.1 which is available only on a limited number of devices.

Our experiments show that the buffer storage option is superior for two reasons: (1) not all mobile devices support 32-bit floating point textures; however, less than 32 bits of precision results in artifacts during application, as shown in Figure 6 (next page); and (2) the data must be transferable to main memory to support undo/redo and import/export functionality, which we were unable to accomplish for 32-bit floating point textures on any device due to texture read-back limitation of the OpenGL ES API. Given the availability of OpenGL ES 3.1, we prefer the compute shader implementation (described below) over transform feedback due to the reduced memory consumption.



(a) 32 bit precision. (b) 16 bit precision. (c) 8 bit precision.

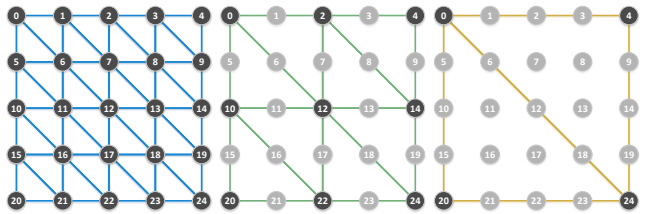
Figure 6: Artifacts occurring for lower precisions (b), (c) compared to full 32 bit precision (a). Red dots indicate the vertex positions.

4.2. Deformation Buffer Implementation

This section covers details of the deformation buffer representation and its initialization, as well as import/export and undo/redo functionalities.

Deformation Buffer Representation. The deformation buffer is a linearized representation of the deformation grid, consisting of two-component vectors with 32 bit precision which directly store the deformed vertex positions. In addition to the buffer, a texture is used to represent the user selection mask since it allows sampling at arbitrary positions using bilinear interpolation. Prior to performing deformation operations, the deformation buffer is required to be initialized once. The initialization is performed procedurally by computing the initial, undeformed position for each vertex in the grid. This process is implemented using buffer mapping [Lee15, p. 53-57] and subsequently generating the vertex data into the mapped buffer on the CPU. Alternatively, an existing deformation template can be imported as described below. In addition, the user selection mask texture is initialized to 1.

Data Persistence and Storage. Several operations can require the transfer of deformation data between video and main memory. Using buffer mapping, this task can be accomplished by mapping the deformation buffer into main memory and then copying data from or to the mapped buffer [Lee15, p. 53-57]. Subsequently, the copied data can be used for undo/redo and import/export functionality. The first is implemented by keeping a stack of copies in main memory, creating a new entry each time an input gestures is completed. If large buffer resolutions or stack sizes are used, it may be required to swap older entries out to disk to reduce memory consumption.



(a) Stride = 1. (b) Stride = 2. (c) Stride = 4.

Figure 7: Different mesh resolutions can be achieved using index buffers with different row and column strides based on the same vertex buffer.

```
#version 310 es
precision mediump float;
layout(local_size_x = 64, local_size_y = 1) in;

uniform vec2  u_Position;
uniform vec2  u_Direction;
uniform float u_AspectRatio;
uniform float u_Radius;
uniform float u_Strength;

// Vertex positions
layout(std430, binding = 0) buffer Positions {
  vec2 positions[];
};

void translate(inout vec2 vertex) {
  // test for circle of influence
  float dist = distance(vertex / vec2(1.0, u_AspectRatio),
     u_Position / vec2(1.0, u_AspectRatio));
  if (dist < u_Radius) {
    // compute offset
    float drop = smoothstep(0.0, u_Radius, dist);
    vec2 offset = u_Direction * u_Strength * drop;
    // apply offset
    vertex += offset;
  }
}

void main(void) {
  // check invocation range
  int index = int(gl_GlobalInvocationID.x);
  if (index >= positions.length())
    return;

  // translate vertex
  translate(positions[index]);
}
```

Listing 1: Exemplary GLSL ES code of a compute shader program for the translation deformation operation. Features such as symmetry, constraints, deformation templates, and optimizations are omitted for brevity.

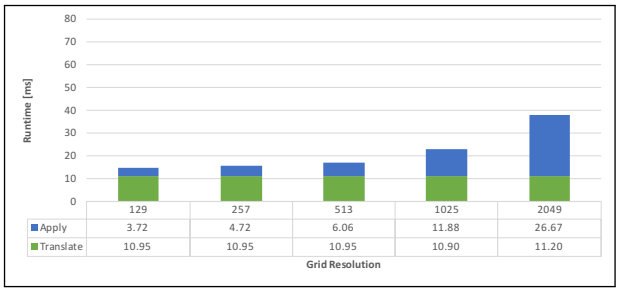
For data exchange, the copied data can be encoded in a suitable exchange format that supports 32 bit floating point values, such as TIFF or OpenEXR and shared with other users subsequently.

4.3. Compute Shader Execution

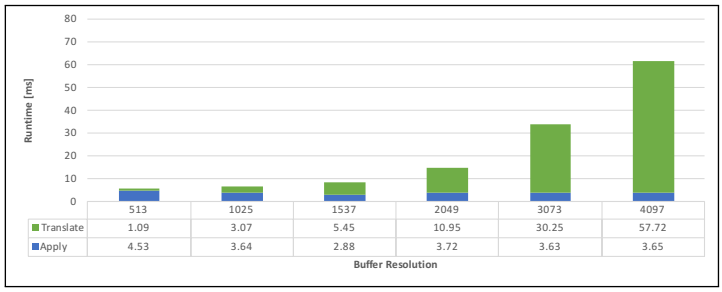
The specific deformation operations are implemented using compute shader programs. We use separate programs for each operation to reduce code complexity. As at most, one user input event is processed per frame, this separation does not result in additional program switches during rendering that might degrade performance. To clarify the implementation of deformation operations, Listing 1 shows a compute shader program for the translation deformation operation. The operation parameters are passed to the shader program using uniform variables, i.e., the program must be executed once for every input event. The deformation buffer can be directly modified when bound as shader storage buffer object.

4.4. Application of Image Deformation

After a deformation operation is applied to the deformation buffer, the actual image deformation is performed subsequently by rendering a uniform grid mesh (deformation mesh) using the deformation buffer as vertex buffer and texturing the mesh with the input image. To support grid resolutions m other that the deformation buffer resolution n, indexed rendering (Figure 7) is used with index buffers



(a) Varying grid resolution.



(b) Varying buffer resolution.

Figure 8: Runtime in milliseconds of the *translate* and *apply* operations of our GPU-based approaches on the *HTC Nexus 9* device with respect to varying grid sizes and a constant buffer size of 2049 (a), as well as varying buffer sizes and a constant grid size of 129 (b).

having different row and column strides. This implies, that the number of buffer subdivisions $n-1$ must be an integer multiple of the number of grid subdivisions $m-1$. To maximize the number of grid resolutions that can be combined with a specific buffer resolution, we use values calculated as 2^k+1 with $k \in \mathbb{N}^+$. Rendering complexity and memory consumption of the index buffer is reduced by using a single triangle strip to represent the entire grid. The texture coordinates required to sample the input image are computed as:

$$x = \frac{v_I \bmod n}{n-1} \quad \text{and} \quad y = \frac{\left\lfloor \frac{v_I}{n} \right\rfloor}{n-1}$$

based on the vertex ID v_I supplied by OpenGL to the vertex shader and corresponding to the vertex index specified in the index buffer [Lee15, p. 274], as well as the deformation buffer size $n \in \mathbb{N}^+$, thus no additional memory is required.

5. Results and Discussion

This section discusses the presented real-time image deformation techniques by means of a performance evaluation and comparison, by stating conceptual and technical limitations, and future work.

5.1. Performance Evaluation

We evaluated the performance of two prototypical implementations for a common mobile image deformation application: (1) the presented GPU-based approaches using compute shaders and (2) a traditional CPU-based approach, all considering different deformation buffer and deformation grid resolutions.

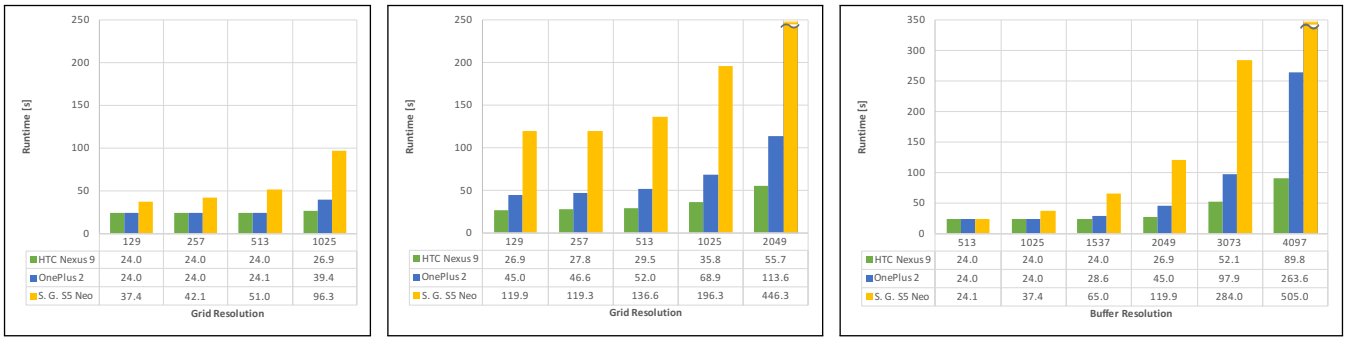
Test Setup. The performance measurements are conducted on the three devices with different CPU/GPU combinations listed in Table 1 and running Android 5.1 with support for OpenGL ES 3.1, according to their specifications. The *OnePlus Two* and *Samsung Galaxy S5 Neo* devices failed to produce valid time measurements using OpenGL query objects [Lee15], despite advertising the support of the *EXT_disjoint_timer_query* extension, thus detailed per-operation timings are only provided for the *HTC Nexus 9* device. For each setting, a sequence of $N = 1410$ touch events is generated at a rate of 60 Hz and sent to the application while measuring the total runtime t required to process the entire sequence using

Table 1: Devices used for the performance evaluation.

Device	CPU	GPU
HTC Nexus 9	NVIDIA Tegra K1 (Denver)	
OnePlus Two	Qualcomm Snapdragon 810	Qualcomm Adreno 330
Samsung Galaxy S5 Neo	Samsung Exynos 7580	ARM Mali-T720

the `System.nanoTime()` function. The theoretical lower bound for t is $\frac{N}{60\,\text{Hz}} = 23.5\,\text{s}$, though results shown in Figure 9 (next page) suggest an actual minimum runtime of approx. 24 s, as this number is measured repeatedly across all devices, hence we consider this the optimum. We evaluate the total runtime measurements based on the application's *event processing rate* $R := \frac{N}{t}$, whereby we consider an event processing rate of $R \geq 30\,\text{Hz}$, i.e., a total runtime of $t = \frac{N}{30\,\text{Hz}} = 47\,\text{s}$ interactive. In addition, the *EXT_disjoint_timer_query* extension is used to measure the runtime of individual GPU operations, if applicable (see above) with an interactive time-to-frame of $\leq 33.3\,\text{ms}$. Note that the individual operation runtime is not bound by the input event rate and that operations not directly related to deformation, such as displaying on screen, are unaccounted for.

Performance Results. Figure 8a shows the results of the performance measurements for individual deformation operations of the proposed technique with respect to size of the deformation grid and a constant deformation buffer size of 2049×2049 on the *HTC Nexus 9* device. As expected, the runtime of the *apply* operation increases with the grid size, while the runtime of the *translate* operation does not depend on the grid size. Both techniques remain interactive up to a grid resolution of 1025×1025. The runtime with respect to the deformation buffer size and a constant deformation grid size of 129×129 in Figure 8b confirm these findings with an interactive time-to-frame up to a buffer resolution of 2049×2049. Table 2 (next page) shows the memory consumption of the deformation buffer and grid for a subset of these sizes, calculated as described in Section 5.2.



(a) Varying grid resolution (buffer: 1025). (b) Varying grid resolution (buffer: 2049). (c) Varying buffer resolution (grid: 129).

Figure 9: Total time in milliseconds required to process the entire input event sequence with increasing grid resolutions for constant buffer resolutions of 1025 (a) and 2049 (b), as well as increasing buffer resolutions for a constant grid resolution of 129 (c).

Table 2: Memory consumption in megabytes of the deformation and index buffer of various sizes.

Buffer	Size						
	257	513	1025	1537	2049	3073	4097
Deformation	0.5	2.0	8.0	18.0	32.0	72.0	128.1
Index	0.5	2.0	8.0	18.0	32.0	72.0	128.0

Performance Comparison. Figure 10 shows the results of the performance comparison for a CPU-based implementation (written in Java) in terms of the total time required to process the sequence of generated events on different devices. Using the CPU-based approach, the time complexity is $O(m^2)$ for the grid resolution m as expected on all devices with the *HTC Nexus 9* being capable of processing all incoming events at optimum rate up to a grid resolution of 93×93, while the other two devices fail to do so from a grid resolution of 65×65 and 93×93, respectively. At grid resolutions of 193×193 and 257×257, none of the devices provides interactive editing with event processing rates as low as 4.3 Hz.

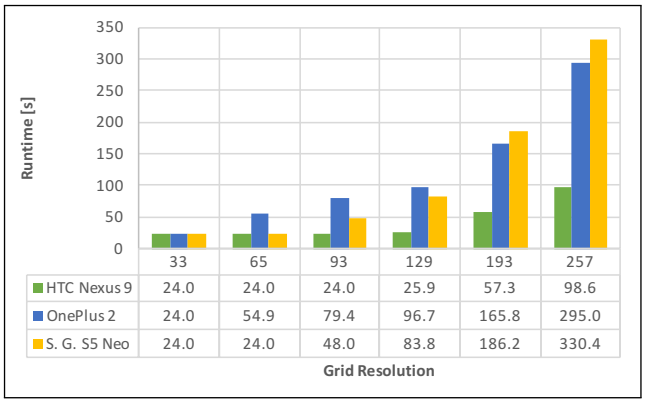


Figure 10: Total time in seconds required to process the entire input event sequence using a traditional CPU-based approach.

Our GPU-based approach achieves the optimum rate on the *HTC Nexus 9* and *OnePlus Two* devices for grid resolutions up to 513×513 using a constant deformation buffer resolution of 1025×1025 and at an interactive processing rate for a grid resolution matching this buffer resolution, while the *Samsung Galaxy S5 Neo* does not achieve the optimum rate for any of the tested resolutions and remains interactive up to a grid resolution of 257×257, as shown in Figure 9a. For a constant buffer resolution of 2049×2049, none of the devices was capable of processing the input events at optimum rate, while the *HTC Nexus 9* and *OnePlus Two* devices achieved an interactive rate up to a grid resolution of 1025×1025 and 513×513, respectively, as shown in Figure 9b. Using a constant grid resolution of 129×129 and varying buffer resolutions, our approach can process deformation buffers up to 1537×1537 at optimum rate on the *HTC Nexus 9* and *OnePlus Two* devices and up to 2049×2049 interactively, while the *Samsung Galaxy S5 Neo* achieves nearly optimum rate at a buffer resolution of 513×513 and interactive rates up to 1025×1025 (Figure 9c).

Performance Discussion. The presented GPU-based deformation technique provides superior deformation precision at interactive processing rates by means of higher resolution grids, resulting in a better output quality compared to the traditional CPU-based approach. The performance figures confirm our assumption that the decoupling of mesh and buffer resolutions results in improved visual quality, as higher buffer resolutions are achievable. Since the runtime performances for deformation and application operations are independent of each other, the runtime of any combination of deformation buffer and grid sizes can be obtained by adding their individual runtimes. This can be used to achieve an interactive editing mode using a high-resolution deformation buffer in combination with a low-resolution grid for preview, followed by an export step using a high-resolution grid in combination with the same deformation buffer, thus maximizing the visual quality of the final image. However, these runtimes depend on the specific device's GPU and the differences suggest a varying suitability for compute shader intensive operations amongst currently available GPUs, either due to their hardware architecture or possibly driver implementation status.

5.2. Limitations and Future Work

Both of our presented approaches and the traditional CPU-based technique are limited by the amount of memory M required:

$$M = \underbrace{b \cdot n^2 \cdot 2 \cdot 4\,\mathrm{B}}_{\text{Vertices}} + \underbrace{\Big((m-1)^2 \cdot 2 + (m-1) \cdot 2\Big) \cdot 4\,\mathrm{B}}_{\text{Indices}}$$

Here, b denotes the number of deformation buffers required, n denotes the buffer resolution, and m denotes the grid resolution. The above equation assumes two components for the vertex position with four bytes each, as well as four bytes per index. Each of the three approaches imposes different constraints on these variables CPU ($m = n$, $b = 1$), transform feedback ($m \leq n$, $b = 2$), and compute shader ($m \leq n$, $b = 1$). Thus, the transform feedback technique exhibits a higher memory consumption compared to the CPU-based technique, while the compute shader approach has an equal memory consumption if using grid resolutions matching the buffer resolution. Besides counterbalancing the limitations stated above (e.g., by packing float values into multiple buffers with 8 bit precision), the presented approach has potential for further future research. We plan to implement more deformation operations (e.g., swirl or liquify) and the support for more advanced puppet image warping [Jac13], as well as automatic tessellation and subdivision of deformed mesh primitives for increased control in fine details. Furthermore, we strive towards an automatic mapping of rigged face-meshes (generated by approaches similar to [UPG*14]) onto deformation buffers, to easily control facial deformation and enable deformation presets accordingly.

6. Conclusions

In this paper, we presented a novel concept for interactive image deformation based on a deformation buffer storing deformation data independent of the grid used for rendering. The presented performance evaluation confirmed our expectation, that the decoupling of buffer and mesh resolutions increases the maximum achievable resolution by using a low-resolution mesh during interactive editing and a high-resolution mesh during the final export for maximum quality. Additionally, our fully GPU-based implementation overcomes performance limitations of traditional CPU-based techniques, further increasing the maximum resolution.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. This work was funded by the Federal Ministry of Education and Research (BMBF), Germany, within the InnoProfile Transfer research group “4DnD-Vis” (http://www.4dndvis.de).

References

[Akl97] AKLEMAN E.: Making caricatures with morphing. In *ACM SIGGRAPH 97 Visual Proceedings: The Art and Interdisciplinary Programs of SIGGRAPH '97* (1997), SIGGRAPH '97, ACM, p. 145. 2

[APL00] AKLEMAN E., PALMER J., LOGAN R.: Making extreme caricatures with a new interactive 2d deformation technique with simplicial complexes. In *In Proceedings of Visual'2000* (2000), pp. 100–105. 2

[Bar84] BARR A. H.: Global and local deformations of solid primitives. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques* (1984), SIGGRAPH '84, ACM, pp. 21–30. 2

[BN92] BEIER T., NEELY S.: Feature-based image metamorphosis. *SIGGRAPH Comput. Graph. 26*, 2 (July 1992), 35–42. 2

[Cre99] CRESPIN B.: Implicit free-form deformations. In *Proceedings of implicit surfaces* (1999), vol. 99, pp. 17–23. 2

[CS07] CORREA C. D., SILVER D.: Programmable shaders for deformation rendering. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware* (2007), GH '07, Eurographics Association, pp. 89–96. 2

[CTMT10] CAO T.-T., TANG K., MOHAMED A., TAN T.-S.: Parallel banding algorithm to compute exact distance transform with the gpu. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2010), I3D '10, ACM, pp. 83–90. 4

[Gre07] GREEN C.: Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses* (2007), SIGGRAPH '07, ACM, pp. 9–18. 4

[Jac13] JACOBSON A.: *Algorithms and Interfaces for Real-Time Deformation of 2D and 3D Shapes*. PhD thesis, ETH Zürich, 2013. 8

[Lee15] LEECH J.: *OpenGL ES Version 3.1*. The Khronos Group Inc., Apr. 2015. 4, 5, 6

[LYNH14] LUO W., YANG X., NAN X., HU B.: Gpu accelerated 3d image deformation using thin-plate splines. In *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)* (Aug 2014), pp. 1142–1149. 2

[MF12a] MOVANIA M. M., FENG L.: A novel gpu-based deformation pipeline. *ISRN Computer Graphics 2012* (2012), 8. 2

[MF12b] MOVANIA M. M., FENG L.: Realtime physically-based deformation on the gpu using transform feedback. In *OpenGL Insights*, Cozzi P., Riccio C., (Eds.). CRC Press, July 2012, ch. 17, pp. 231–246. http://www.openglinsights.com/. 2, 4

[MZDP11] MENG W., ZHANG X., DONG W., PAUL J.-C.: Multicage image deformation on gpu. In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry* (2011), VRCAI '11, ACM, pp. 155–162. 2

[RT06] RONG G., TAN T.-S.: Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games* (2006), I3D '06, ACM, pp. 109–116. 4

[SBGS06] SPINDLER M., BUBKE M., GERMER T., STROTHOTTE T.: Camera textures. In *GRAPHITE '06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and South East Asia* (2006), ACM Press, pp. 295–302. 2

[SF98] SINGH K., FIUME E.: Wires: A geometric deformation technique. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (1998), SIGGRAPH '98, ACM, pp. 405–414. 2

[Sim15] SIMPSON R. J.: *The OpenGL ES Shading Language Language Version: 3.10 Document Revision: 4*. The Khronos Group Inc., Apr. 2015. 2, 4

[SP86] SEDERBERG T. W., PARRY S. R.: Free-form deformation of solid geometric models. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques* (1986), SIGGRAPH '86, ACM, pp. 151–160. 2

[UPG*14] UNZUETA L., PIMENTA W., GOENETXEA J., LUÍS PAULO S., DORNAIKA F.: Effcient generic face model fitting to images and videos. *Image and Vision Computing 32*, 5 (2014), 321–334. 8

[YCB05] YANG Y., CHEN J., BEHESHTI M.: Nonlinear perspective projections and magic lenses: 3d view deformation. *Computer Graphics and Applications, IEEE 25*, 1 (Jan 2005), 76–84. 1