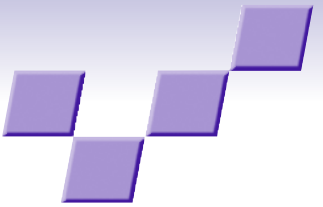

The State of the Art in Mobile Graphics Research

Tolga Capin ■ *Bilkent University*

Kari Pulli ■ *Nokia Research Center Palo Alto*

Tomas Akenine-Möller ■ *Lund University*

High-quality computer graphics let mobile-device users access more compelling content. Still, the devices' limitations and requirements differ substantially from those of a PC. This survey of mobile graphics research describes current solutions in terms of specialized hardware (including 3D displays), rendering and transmission, visualization, and user interfaces.

Mobile phones are virtually omnipresent. In 2008, 3.3 billion people—half the world population—use mobile phones, according to the International Telecommunications Union. By 2010, Nokia expects that there will be as many mobile phone users as toothbrush users (4 billion). Over the past 10 years, the phone has expanded from being just a phone to being a full multimedia unit, on which you can play games (see Figure 1), shoot photos, listen to music, watch television or video, send messages, and do videoconferencing.

One factor leading to the widespread adoption of mobile phones has been the dramatic improvement in display technologies. Displays used to be monochromatic and small (48 × 84 pixels). Today, we have 24-bit (16.8 million colors) displays with VGA resolution (640 × 480 pixels). Consequently, mobile phones have the potential to deliver graphics to the masses.

The mobile context differs vastly from the PC context. A mobile phone

- is always with you,
- is always connected to the network and can find its location and provide access to location-based services and navigation, and
- supports applications that require a graphics-intensive user interface.

In addition, most mobile phones include a camera, which allows many possibilities for better user interaction with the device, as well as augmented reality (AR) applications that combine digital images (rendered graphics models) with real-world images (such as those on a camera viewfinder).

Standard mobile graphics APIs have laid the foundation for much mobile graphics research and applications. For 3D graphics, there's OpenGL ES, which is a low-level API based on the popular OpenGL, and M3G (JSR 184), which is designed on top of OpenGL ES and supports scene graphs, animation, and file formats for mobile Java. Kari Pulli and his colleagues cover various uses of OpenGL ES and M3G.[1] For 2D vector graphics, there's OpenVG, a low-level API similar to OpenGL, and Scalable Vector Graphics API for mobile Java (JSR 226). A description of these and other related APIs appears elsewhere.[2]

By *mobile*, we mostly mean handheld devices. So, although aviation or car displays are certainly mobile, they fall outside this article's scope. Here, we aim to survey the state of mobile graphics research. We don't address issues related to particular applications and development tools. We also don't discuss mobile gaming in depth; Mark Callow and his colleagues provide a good overview of mobile-game development and distribution.[3] Jörg Baus and his colleagues survey 2D and 3D maps for navigation, which is also mostly beyond this article's scope.[4] Furthermore, we concentrate on interactive graphics because noninteractive graphics can be simply rendered on other devices and rendered as simple bitmaps. For this reason, we also address user interfaces and handheld interaction techniques.

 Published by the IEEE Computer Society

Handhelds' limitations

Compared to the desktop, handheld devices are limited by

- power supply,
- computational power,
- physical display size, and
- input modalities.

Mobile devices' fundamental problem is that they're battery operated. Whereas many other aspects of computing follow Moore's law, battery technology develops much more slowly. The display is one of the largest consumers of power, and graphics applications keep the display, often with a backlight, constantly on. Innovation is required at the hardware level for lower power consumption, while diligence is required at the software level for power-aware mobile applications. Finally, the devices are small; even if more power were available, that power would turn into heat, which can damage circuits unless the design process considered the thermal aspects early on.

Mobile device CPUs also have limited computing power. A related limitation is internal bandwidth for memory accesses, which increases more slowly than raw computing power and consumes much power. Another limitation is cost: mass-market consumer devices should be cheap, which limits the silicon budget. For example, only the most recent high-end phones support floating-point units. Having dedicated graphics hardware helps the devices get by with lower-clock-rate CPUs.

Although the pixel pitch ratio is increasing at a stable rate, the requirements to keep the devices handheld and pocketable means that the devices' physical size has an upper bound. Whereas the largest displays might have a diameter of up to 5 inches, many devices have much smaller displays.

Furthermore, mobile devices currently support key-based interfaces through joypad and direction keys and a numerical keyboard. On larger devices, additional keys provide a better user experience for complex tasks because keys can be dedicated to specific tasks. Smart phones can't easily use such keys owing to limited physical space. Interaction with touch-sensitive screens has emerged as an alternative, but most solutions require two-handed interaction, which causes additional attentional overhead in users.

Finally, there's an order of magnitude difference between high- and low-end devices in graphics processing and computational capacity. A particular technique might run efficiently in one device but be inefficient on another. This requires solutions that can scale down to low-end mobile phones and up to larger devices, even PCs.

Figure 1. High-quality graphics games have reached mobile devices.

Industry and academia researchers have developed several solutions to these problems. The following sections describe the key approaches.

Graphics hardware

A given task, such as 3D rendering, can always be done more efficiently on special-purpose hardware than on a general-purpose CPU. It's possible to write a rendering engine fully in software executing on a CPU, providing maximum flexibility. In fact, most mobile 3D engines are still software implementations. However, dedicated graphics hardware can provide both faster execution and lower power consumption. Dedicated graphics processing units (GPUs) are already available on high-end smart phones. Some GPUs are available on a separate chip, but often the GPU and CPU are on the same chip, which decreases manufacturing costs.

Although modern graphics engines, such as OpenGL ES 2.0, provide programmable components—so-called shaders—a lot of functionality still isn't programmable but consists of blocks of fixed functionality that can be parameterized and turned on or off. Fixing the functionality allows more efficient implementations and latency hiding. Triangle setup, texture fetch and filtering, and blending operations can be more efficient when implemented in dedicated logic.

The key to good graphics performance and low power consumption is to reduce the internal traffic between the processing elements and the memory. So, mobile graphics solutions focus on how to compress and even completely avoid that traffic. Reducing the traffic is even more important because computation power increases more quickly than memory bandwidth. For example, John Owens reports that the yearly processing capability growth is about 71 percent, while dynamic RAM bandwidth grows only by 25 percent.[5] This difference suggests that one should take great care when designing a GPU architecture.

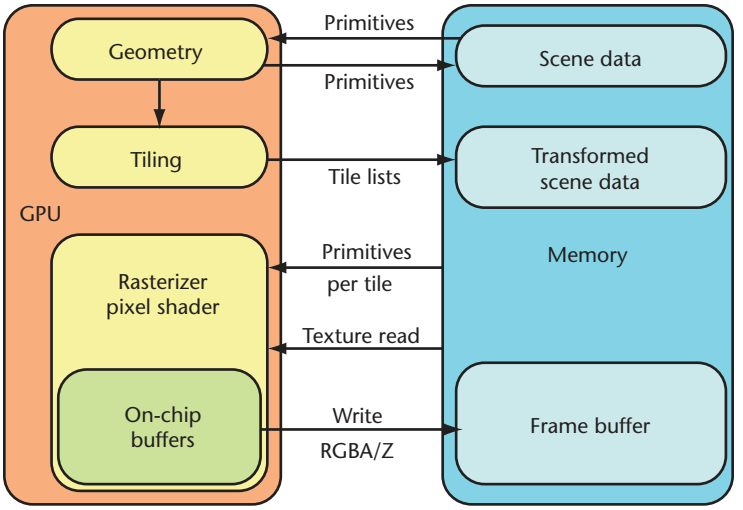

Figure 2. A tiling architecture. The primitives are being transformed and stored in external memory. There they are sorted into tile lists, where each list contains the triangles overlapping that tile. This makes it possible to store the frame buffer for a tile (for example, 16 × 16 pixels) in on-chip memory, which makes accesses to the tile's frame buffer extremely inexpensive.

Compression

Compression not only saves storage space, but it also reduces the amount of data sent over a network or a memory bus. For GPUs, compression and decompression (codec) have two major targets: textures and buffers.

Textures are read-only images glued onto geometrical primitives such as triangles. A texture codec algorithm's core requirements include fast random access to the texture data, fast decompression, and inexpensive hardware implementation. The random-access requirement usually implies that a block of pixels is compressed to a fixed size. For example, a group of 4 × 4 pixels can be compressed from 3 × 8 = 24 bits per pixel down to 4 bits per pixel, requiring only 64 bits to represent the whole group. As a consequence of this fixed-rate compression, most texture compression algorithms are lossy (for example, JPEG) and usually don't reproduce the original image exactly. Because textures are read-only data and usually compressed offline, the time spent compressing the image isn't as important as the decompression time, which must be fast. Such algorithms are sometimes called asymmetric.

As a result of these requirements, developers have adopted Ericsson Texture Compression (ETC) as a new codec for OpenGL ES.[6] ETC stores one base color for each 4 × 4 block of texels and modifies the luminance using only a 2-bit lookup index per pixel. This technique keeps the hardware decompressor small. Currently, no desktop graphics APIs use this algorithm.

Buffers are more symmetric than textures in terms of compression and decompression because both processes must occur in hardware in real time. For example, the color buffer can be compressed, so when a triangle is rendered to a block of pixels (say, 4 × 4) in the color buffer, the hardware attempts to compress this block. If this succeeds, the data is marked as compressed and sent back to the main memory in compressed form over the bus and stored in that form. Most buffer compression algorithms are exact to avoid error accumulation. However, if the algorithm is lossy, the color data can be lossily compressed and later recompressed, and so on, until the accumulated error exceeds the threshold for what's visible. This is called *tandem compression*, meaning that if compression fails, you must have a fallback that guarantees an exact color buffer—namely, sending the data uncompressed.[7]

Depth and stencil buffers might also be compressed. The depth buffer deserves special mention because its contents are proportional to 1/*z*, and when viewed in perspective, the depth values over a triangle remain linear. Depth-buffer compression algorithms heavily exploit this property, which accounts for higher compression rates. A survey of existing algorithms appears elsewhere.[8]

Interestingly, all buffer codec algorithms are transparent to the user. All action takes place in the GPU and is never exposed to the user or programmer, so there's no need for standardization. There's no major difference for buffer codec on mobile devices versus desktops, but mobile graphics has caused renewed interest in such techniques.

Tiling architectures

Tiling architectures aim to reduce the memory traffic related to frame-buffer accesses using a completely different approach. Tiling the frame buffer so that a small tile (such as a rectangular block of pixels) is stored on the graphics chip provides many optimization and culling possibilities. Commercially, Imagination Technologies and ARM offer mobile 3D accelerators using tiling architectures. Their core insight is that a large chunk of the memory accesses are to buffers such as color, depth, and stencil.

Ideally, we'd like the memory for the entire frame buffer on-chip, which would make such memory accesses extremely inexpensive. However, this isn't practical for the whole frame buffer, but storing a small tile of, say, 16 × 16 pixels of the frame buffer on-chip is feasible. When all rendering has been finished to a particular tile, its contents can be written to the external frame buffer in an efficient block transfer. Figure 2 illustrates this concept.

However, tiling has the overhead that all the triangles must be buffered and sorted into correct tiles after they're transformed to screen space. A tiling

unit creates, for each tile, a list of triangles overlapping with that tile. Each tile can then be processed in turn or in parallel with others. This architecture's main advantage is that frame-buffer accesses become inexpensive. This must be balanced with the cost of buffering and sorting the triangles.[9] It's still unknown whether a traditional architecture or tiling is best. The optimal approach also depends on the content being rendered. For example, if the overdraw factor is high, the tiled approach can be a winner, but if there are many long but thin triangles, the traditional nontiled approach might work better.

Culling

Even better than compressing data is to avoid processing it. To cull means "to select from a group," and this often amounts to avoiding processing data that doesn't contribute to the final image. One particular technique stores (in a cache) the maximum, Z_{max}, of the depth values in a block of pixels, and when rendering to this block, the GPU estimates conservatively whether the triangle is behind Z_{max}. If so, all per-pixel processing can be avoided in that block because the triangle will be hidden.[10]

A similar technique stores the minimum, Z_{min}, of the depth values and determines whether a triangle is definitely in front of all rendered geometry in a block. If so, depth buffer reads can be avoided in the block.[11] You can also use Z_{min} to handle other depth tests. These two techniques are often called *Z*-culling.

Another technique uses occlusion queries. The programmer renders, for example, a bounding box around a complex object, and the occlusion query counts how many fragments on the box are visible. If no fragments are visible, then the bounding box is hidden and rendering the complex object can be avoided. Another approach, called delay streams, can also be used for occlusion culling.[12] The idea is to process the triangles as usual, write to the depth buffer, and delay other per-pixel processing. Instead, the triangles are put in a first-in, first-out queue (that is, the delay stream). When the delay stream is full, the "occluding power"—that is, the Z_{max} values—builds up substantially. As the triangles leave the delay stream, they are tested against the Z_{max} values, and many fragments can be skipped because they're now occluded by other surfaces.

With the advancement of programmable shaders, more work is being put into pure computation. At some point, it's likely that the GPU will become compute-bound—that is, limited in performance because of too much computation. One solution is to spend more time on shader compiler optimization, but that only takes you so far. Another solution is to avoid executing the pixel shader when you can determine that the computation results won't contribute to the final image anyway. For example, consider a block of pixels that are all in shadow (completely black). If a high-level mechanism could determine conservatively that these pixels are all in shadow, then per-pixel shadow computations could be avoided.

This is another type of culling, and the basic idea is implemented in the programmable culling unit (PCU).[13] The PCU executes the pixel shader once over an entire block of pixels. For conservative output, the computations are carried out using interval arithmetic, so the input is the intervals of the block's in-parameters. The total number of instructions decreased from 48 to 71 percent, which indicates that a performance increase of about 2 times is possible. In addition, the memory bandwidth usage decreased by 14 to 28 percent. Interestingly, the PCU can also operate in a lossy mode. The programmer can activate this by instructing the pixels to be killed if the contribution is less than, say, 1 percent of the maximum intensity. In such a case, the threshold of when per-pixel processing should commence provides a knob that the user can set to trade off image quality for performance.

At some point, it's likely that the GPU will become compute-bound—that is, limited in performance because of too much computation.

Adaptive voltage scaling

The techniques we just discussed are high-level solutions. Other methods reduce power usage at the hardware level. Several researchers have proposed low-power GPUs with conventional power management strategies. Bren Mochocki and his colleagues analyze how such factors as resolution, frame rate, level of detail, lighting, and texture maps affect power consumption of mobile 3D graphics pipeline stages.[14] On the basis of this analysis, they use dynamic voltage and frequency scaling (DVFS) schemes for different pipeline stages. Using a prediction strategy for workloads for the different stages, DVFS could decrease power consumption by 40 percent.

3D displays and rendering

Many solutions for mobile 3D displays don't require additional peripherals, such as glasses or head gear. Such displays are often called autostereoscopic

displays. Rendering to such displays can be more expensive than rendering to a regular display. So, specialized algorithms and hardware can help reduce the workload.

To give the sensation of 3D to a stationary observer, a device must exploit a key source of 3D perception: the binocular parallax. All autostereoscopic displays exploit the binocular parallax through direction-dependent displaying. This means that the device must provide different views for each eye.

Existing solutions employ either a volumetric, multiview, or holographic display. The display most applicable to mobile devices is the multiview display, which uses lens arrays or parallax barriers to direct or select simultaneously displayed images depending on the viewpoint. All these solutions provide a single or multiple observer location from where a stereo pair of images is visible, while other positions yield unfocused or incorrect views.

In parallel with advances in graphics hardware and displays, we're witnessing a dramatic increase in the complexity of graphics models on mobile devices.

Stereo rendering generally costs twice as much in computation and bandwidth. However, for a larger angle of usage (that is, larger than the angle between the two eyes of an observer), some displays use even more views, which requires more processing. Specialized hardware can potentially render to autostereoscopic displays more efficiently because the images for the left and right eyes are similar. In contrast, with a brute-force implementation, the scene is rendered first to the left eye and then to the right eye.

However, it makes sense to render a single triangle to both views before proceeding with the next triangle.[15] Aravind Kalaiah and Tolga Capin use this rendering order to reduce the number of vertex shader computations.[16] Splitting the vertex shader into view-independent (computed once) and view-dependent parts can greatly reduce vertex shader computations. In the following per-pixel processing stage, a simple sorting procedure in a generalized texture space greatly improves the texture cache hit ratio, keeping the texture bandwidth close to that of monoscopic rendering.[15]

In addition, Jon Hasselgren and Tomas Akenine-Möller introduce approximate rendering in the multiview pipeline, so that fragment colors in all neighboring views can be approximated from a central view when possible.[15] When approximative rendering is acceptable, you can avoid many per-pixel shader instruction executions. For stereo rendering, about 95 percent of the computations and bandwidth is avoided for the left view (the right view must be rendered as usual).

Rendering and transmission

In parallel with advances in graphics hardware and displays, we're witnessing a dramatic increase in the complexity of graphics models on mobile devices. Here, we highlight recent advances in rendering and transmitting such models on mobile devices.

To overcome the complexity of representing the mesh connectivity, numerous solutions convert input mesh models to internal, more efficient representations. Florent Duguet and George Drettakis's solution uses point-based graphics.[17] They create point samples from an input mesh as a preprocess or procedurally on the fly and create a hierarchical representation of the object samples' bounding volumes. During rendering, the processing of the hierarchy stops at a specified depth, achieving flexible rendering that's scalable to the mobile device's speed requirements and screen size. This approach is also memory efficient because it doesn't need to keep the whole model in main memory.

An alternative approach eliminates rendering nonimportant parts of the graphical content. Vidya Setlur and her colleagues' method considers the human perception system's limitations for retargeting 2D vector animations for small displays.[18] They aim to preserve key objects' recognizability in a vector graphics animation by exaggerating the important objects' features and eliminating insignificant parts during rendering. Instead of uniformly scaling down the input to small displays, this perceptually based solution uses nonuniform scaling of objects, based on the objects' importance in the scene.

Jingshu Huang and her colleagues try a different approach to rendering complex models on small screens.[19] Their MobilVis system adapts well-known illustrative rendering techniques, such as interactive cutaway views, ghosted views, silhouettes, and selective rendering, to mobile devices to more clearly convey an object's shapes, forms, and interior structures.

Although these solutions provide more efficient results than basic graphics rendering, they're still limited by the devices' processing power. Because mobile devices are always connected to the network, remote rendering becomes a viable alternative. Typically, this technique uses a client–server approach. The rendering occurs on a high-performance server or a PC; the mobile client receives intermediate results

over a network connection and renders the final results. Chun-Fa Chang and Shyh-Haur Ger present an image-based remote-rendering solution, where the client receives depth images from the server and applies a 3D warping method, achieving near-real-time rates.[20] Daoud Hekmatzada and his colleagues present a nonphotorealistic rendering solution, based on drawing silhouettes and contour lines as primitives.[21]

A related problem is the transmission of complex models to mobile devices. Downloading such models through the air requires much bandwidth. In Xiaonan Luo and Guifeng Zheng's solution for transmitting meshes, the mobile device communicates with a wired IP server via an IP network and a wireless channel.[22] This solution is based on a flexible progressive mesh coding technique that adapts to different bit-rate and error-resilience requirements, while minimizing computational complexity usually associated with a transcoder. Azzedine Boukerche and Richard W.N. Pazzi present a streaming protocol for 3D virtual-environment exploration on mobile devices; they address network transmission problems such as rate and congestion control.[23] Siddhartha Chattopadhyay and his colleagues describe power-aware compression and transmission of motion capture data for mobile devices.[24]

Several issues must be solved for remote rendering, such as connectivity problems, latency for transmitting user input, and rendered images. Hybrid solutions that balance processing between on-device and remote rendering present interesting research possibilities.

Visualization and user interfaces

The key challenges in mobile visualization and user interfaces relate to small displays and the limited amount of interaction hardware compared to the desktop (for example, there's no mouse or a full-size keyboard). Interaction is an important component of most graphics applications.

Visualization

Presenting large amounts of graphical data and complex user interface components more effectively on small displays is a key research topic. When the data complexity exceeds what mobile displays can show, users must manually browse through the large data. This can easily happen when rendering and visualizing 2D data (such as maps or documents) or 3D data (such as medical data or city models). Scalable and zoomable user interfaces also require such visualization techniques. Luca Chittaro surveys problems and solutions for visualizing five types of data for mobile applications such as text, pictures, maps, physical objects, and abstract data.[25]

Patrick Baudisch and Ruth Rosenholtz propose the classification of the two following approaches to visualization on mobile devices.[26]

Overview + Detail. These approaches are based on displaying two different views of the data simultaneously—one for context and one for detail. While the user navigates around the large data in the context view, the detailed view displays the area in focus.

Focus + Context. These approaches use a single view into data, with nonuniform scaling of data elements. The most prominent solution is the fish-eye view, which magnifies the data in the user's attention and renders distant objects in progressively smaller sizes. Fish-eye views are mostly used in maps and menus.[27]

One example of this approach is speed-dependent adaptive zooming. Tolga Capin and Antonio Haro capture the device's physical movement from camera input, which they analyze to determine scroll direction and magnitude.[28] The zoom level increases or decreases depending on the scroll's magnitude. For example, when a user moves a phone, the view zooms out and the display shows an overall view. When the user stops moving the phone, the zooming level gradually increases and the display shows a detailed view.

Benjamin Bederson and his colleagues developed DateLens, a fish-eye interface for a calendar on mobile devices.[29] The user first sees an overview of a large time period using a graphical representation of each day's activities. Choosing a particular day expands the area representing that day and reveals the appointment list in context.

Recently, Amy Karlson and her colleagues proposed AppLens and LaunchTile design solutions that adapt the UI to multiple devices with different resolutions and aspect ratios.[30] AppLens uses a tabular fish-eye approach for integrated access and notification for nine applications. LaunchTile uses pure zooming within a landscape of applications to accomplish the same goals. A further development of LaunchTile is the zoomable fish-eye visualization of Zumobi, for Web browsing on mobile devices (see Figure 3).

Courtesy of Zumobi (www.zumobi.com).

Figure 3. Zumobi's user interface. The interface platform supports a zoomable Web-browsing experience on mobile devices.

Figure 4. The Halo approach displays arcs at the detailed view's borders. The ring's radius is proportional to the distance.

Image courtesy of Patrick Baudisch and Ruth Rosenholtz.

Another problem is visualizing the location of off-screen objects because small mobile displays can't display all data at once. Solutions to this problem augment the detailed view with visual references to off-screen objects. For example, Baudisch and Rosenholtz use the "street lamp" metaphor, with an associated halo that includes a red arc at the detailed view's borders.[26] Figure 4 illustrates the Halo approach.

3D user interfaces

Three-dimensional user interfaces are a key application of visualization on mobile devices, especially those with autostereoscopic displays. Creating 3D interfaces that approach the richness of 3D reality has long been a research target of several other research groups, particularly for desktop environments. Ben Shneiderman and Catherine Plaisant analyzed the features of effective 3D interfaces, primarily for desktop and near-to-eye display domains, and proposed numerous guidelines.[31] These include making better use of occlusion, shadows, and perspective; minimizing the number of steps in navigation; and improving text readability with better rendering and contrast with the background.

Graphics hardware support for OpenGL ES 2.0 in a mobile phone opens up new possibilities for user interfaces owing to the programmable nature of that API. Because 3D UI rendering solutions developed for desktop computers don't scale down well to mobile devices, a different set of widgets must be developed. In Figure 5, photos, videos, and applications drop down at the far end and move toward the front. The user can "catch" a photo, video, or application and make it active. This includes showing the video or photo in higher resolution or activating the application. Programmable vertex and pixel shaders render depth-of-field effects and motion blur. These shaders also animate "wobbly" windows using vertex skinning.

Directly manipulating content

Mobile devices are currently limited in the mode of interaction they provide to users. However, direct-manipulation interfaces provide a more intuitive interaction than current key-modal and menu-based systems.[31] Users can manipulate individual objects, each with a direct display representation. They apply actions directly to objects by selecting them and then choosing a command. Graphical representation is key for direct manipulation: users manipulate, through selection events and moving a pointing device, a graphical or iconic representation of the underlying data. Dragging an object by the pointer is an example of this interaction mode.

Recently, stylus- and thumb-based interaction with touch-sensitive screens has emerged as a solution for mobile direct manipulation.[32] Stylus-based interaction, although accurate for selecting objects in a small screen, requires both hands and has caused additional attentional overhead.[33] To overcome this problem, researchers have developed one-handed thumb-based interaction. Apple's iPhone is the most prominent example; with a multitouch capacitive touch screen, it lets users interact with applications and type using their thumbs. Karlson and her colleagues have further developed several high-level gestures for more intuitive interaction with their zoomable user interface solution.[30]

Researchers have also incorporated physical sensors such as accelerometers in mobile devices for richer user interaction.[34] However, such sensors produce error buildup over time. One way to overcome this is by merging relative continuous data from physical sensors with absolute but potentially intermittent data. This approach has provided good results and could lead to reliable tracking solutions.

Figure 5. A sequence of images from the SocialRiver user interface. Using OpenGL ES 2.0, SocialRiver implements motion blur, depth of field, and vertex skinning. Video input can also be composited.

Courtesy of The Astonishing Tribe AB (www.tat.se).

Alternatively, researchers have proposed several solutions where incoming camera video estimates phone motion and interacts with the user's physical environment.[28] With camera-based interaction, users move the pointer or change the view by moving the phone instead of interacting with the screen or keypad. Correctly interpreting the objects' observed motion or the camera's global motion from video requires accurate tracking. Among the recent solutions, Jari Hannuksela and his colleagues propose region-based matching that uses a sparse set of features for motion analysis and a Kalman filter-based tracker for estimation.[35] Capin and Haro's solution tracks individual corner-like features observed in the entire incoming camera frames.[28] This lets the tracker recognize sudden camera movements of arbitrary size, as long as at least some features from the previous frame are still visible. The tradeoff is that the tracker can't detect rotations.

Augmented reality

AR, which augments video with graphics, can be contrasted with virtual reality, which renders everything with computer graphics, and telepresence, which conveys reality somewhere else by transmitting video and audio. Whereas many mobile-graphics applications resemble desktop-graphics applications (only with more constraints and less performance), AR provides a user experience on a mobile system that's different from, and better than, the desktop user experience. Here, we discuss some early AR systems.

One early example of mobile AR is the Touring Machine.[36] The main system consisted of a backpack loaded with a computer and batteries. The user wore a head-mounted display and camera and held a tablet and stylus for input. The system worked as a campus tour guide, displaying the building names and related information over the buildings on its optical-see-through head-mounted display. Two surveys cover the basic components and problems of AR[37] and developments in the late 1990s.[38]

Jun Rekimoto and Katashi Nagao's Navicam was an early handheld AR system.[39] It consisted of a handheld display that showed real-time camera imagery. The images were passed to a workstation for analysis. If the system recognized color-coded ID tags, it would superimpose situation-sensitive information over the camera image and display it on the device. This kind of video-see-through system has many advantages over optical-see-through systems. Optical systems are open-loop control systems that require a good world model and accurate tracking of the user's eye position. A video system provides a much easier closed-loop control system because it analyzes the image, localizes the annotated object only with respect to the camera, and overlays the annotations with the target.

Courtesy of Anders Henrysson.

Figure 6. In AR Tennis, the camera tracks markers on the table and the other player's camera. The players attempt to bounce the ball back and forth in a virtual tennis court.

Whereas NaviCam was tethered to a workstation, Daniel Wagner and Dieter Schmalstieg created the first autonomous handheld AR system.[40] They ported ARToolkit (www.hitl.washington.edu/artoolkit), a popular library for many AR demos that tracks camera position with respect to square markers, to a PDA. The system offloaded the tracking to a server for faster frame rates, and the graphics rendering used a proprietary subset of OpenGL. Soon after, other researchers implemented similar systems on mobile phones, such as Mathias Möhring and his colleagues, who implemented their own tracker,[41] and Anders Henrysson and his colleagues, who adapted Wagner's ARToolkit port to Symbian.[42] Both these systems used OpenGL ES for graphics rendering.[1]

AR is also useful in gaming, and several games feature an AR phone. In Mosquito Hunt by Siemens, virtual mosquitoes are drawn over live video from a camera. By moving the phone and tracking the motion flow in the camera, users try to zap the mosquitoes. In Marble Revolution2 (www.bit-side.com/311.html), the motion flow guides a marble through a maze. Kick Real (www.kickreal.de) shows a soccer ball on the ground that users can kick. AR Tennis tracks markers on a table to anchor a tennis field and tracks additional markers on players' phones for a collaborative or competitive tennis game (see Figure 6).[42]

Most mobile AR systems use markers to track the camera's relative position with respect to objects or use optical flow to track the phone motion. More recently, some systems have done away with markers. The PhoneGuide is a museum guide based on camera phones (see the Projects in VR article in

Figure 7. The PhoneGuide. The user points a camera phone to an object in a museum (left). A Bluetooth beacon gives an approximate location (top right). The system recognizes the image and provides additional information (bottom right).

this issue for more on this).[43] As Figure 7 illustrates, the PhoneGuide determines the user's approximate location using Bluetooth beacons, so the vision system only needs to distinguish between a smaller set of objects. The system splits the input image into bins, each bin produces a global feature vector consisting of various histograms and ratios (colors, intensities, edges, and so on), and a neural network uses the inputs for recognition. Herbert Bay and his colleagues also created a museum guide.[44] Their system runs on a tablet PC and uses local scale-invariant Speeded Up Robust Features (SURF) to recognize objects. Such local feature matchers work better even if the objects have different backgrounds or are partially occluded. SURF has also been ported to camera phones.[45]

The primary remaining challenges in mobile AR are object recognition and real-time tracking for unprepared markerless environments. Overcoming these challenges allows annotating views with labels or arrows pointing to objects of interest. A secondary problem is seamlessly blending the graphics objects with the real scene with correct occlusions and shading. This requires modeling the environment and the current illumination levels in real time on the device.

Clearly, we need specialized graphics hardware for power-efficient graphics, but much research remains to be done. We believe that the best way around the battery capacity problem is to continue work on all fronts, which includes more efficient high-level graphics hardware algorithms, intelligent low-level power management, and clever software techniques for rendering and transmission. This also includes handling large, complex models and data sets. For both software and hardware techniques and algorithms, it would be convenient to have a knob that the user can turn to trade off image quality and operation time. Approximate rendering for graphics hardware is a field that hasn't been investigated thoroughly, and we expect that many new innovations will emerge.

Autostereoscopic displays can provide a major breakthrough on mobile devices before it does so on desktops. Interestingly, several such displays can already switch between displaying a standard 2D image and conveying a 3D autostereoscopic experience. Graphics APIs could easily add support for these displays. For 3D TV and video, open issues in standardization organizations still exist. The main practical obstacle for autostereoscopic displays is creating content that fully benefits from such displays.

User interfaces is an area where much innovation will happen at every level. The low-level APIs, such as OpenVG and OpenGL ES are there, but using 3D so that it truly enhances the user experience is still an active research issue. Multimodal interfaces that integrate voice, gesture, stylus or finger input, and keyboard input with interactive graphics and sound rendering, and take human perceptual and cognitive capabilities into account, will create interaction that's easier and more fun. Games are traditionally good at creating interfaces that are naturally easy to use; hopefully, these UI aspects will become more widespread in mobile UIs.

Because most mobile devices have a camera, exploring how we can integrate AR functionality into such cameras is worth exploring. However, the killer AR application has yet to be discovered. The future of mobile graphics is exciting, and our community will continue to invent new algorithms, techniques, and applications that exploit the context of mobility.

Acknowledgments

The Swedish Foundation for Strategic Research supported Tomas Akenine-Möller through a grant on mobile graphics, and additional support came from a Knowledge Foundation visualization grant. The European Commission FP7 3DPHONE project (grant FP7-213349) and FP6 3DTV project (grant FP6-511568) supported Tolga Capin.

References

1. K. Pulli et al., *Mobile 3D Graphics with OpenGL ES and M3G*, Morgan Kaufmann, 2007.
2. K. Pulli, "New APIs for Mobile Graphics," *Proc. SPIE*

Electronic Imaging: Multimedia on Mobile Devices II, SPIE, 2006, pp. 1–13.
3. M. Callow, P. Beardow, and D. Brittain, "Big Games, Small Screens," *ACM Queue*, Nov./Dec. 2007, pp. 2–12.
4. J. Baus, K. Cheverst, and C. Kray, "Map-Based Mobile Services," *Map-Based Mobile Services Theories, Methods and Implementations*, Springer, 2005, pp. 193–209.
5. J.D. Owens, "Streaming Architectures and Technology Trends," *GPU Gems 2*, Addison-Wesley, 2005, pp. 457–470.
6. J. Ström and T. Akenine-Möller, "iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones," *Proc. ACM Siggraph/Eurographics Conf. Graphics Hardware*, ACM Press, 2005, pp. 63–70.
7. J. Rasmusson, J. Hasselgren, and T. Akenine-Möller, "Exact and Error-Bounded Approximate Color Buffer Compression and Decompression," *Proc. ACM Siggraph/Eurographics Symp. Graphics Hardware*, Eurographics Assoc., 2007, pp. 41–48.
8. J. Hasselgren and T. Akenine-Möller, "Efficient Depth Buffer Compression," *Graphics Hardware 2006: Eurographics Symp. Proc.*, A K Peters, 2006, pp. 103–110.
9. I. Antochi et al., "Scene Management Models and Overlap Tests for Tile-Based Rendering," *Proc. EUROMICRO Symp. Digital System Design*, IEEE CS Press, 2004, pp. 424–431.
10. S. Morein, "ATI Radeon HyperZ Technology," *Proc. Workshop Graphics Hardware* (Hot3D), ACM Press, 2000; www.graphicshardware.org/previous/www_2000/presentations/ATIHot3D.pdf.
11. T. Akenine-Möller and J. Ström, "Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones," *ACM Trans. Graphics* (Proc. Siggraph), vol. 22, no. 3, 2003, pp. 801–808.
12. T. Aila, V. Miettinen, and P. Nordlund, "Delay Streams for Graphics Hardware," *ACM Trans. Graphics* (Proc. Siggraph), vol. 22, no. 3, 2003, pp. 792–800.
13. J. Hasselgren and T. Akenine-Möller, "PCU: The Programmable Culling Unit," *ACM Trans. Graphics* (Proc. Siggraph), vol. 26, no. 3, 2007, article 92.
14. B.C. Mochocki et al., "Signature-Based Workload Estimation for Mobile 3D Graphics," *Proc. 43rd Ann. Conf. Design Automation* (DAC 06), ACM Press, 2006, pp. 592–597.
15. J. Hasselgren and T. Akenine-Möller, "An Efficient Multi-View Rasterization Architecture," *Proc. Eurographics Symp. Rendering*, Eurographics Assoc., 2006, pp. 61–72.
16. A. Kalaiah and T. Capin, "Unified Rendering Pipeline for Autostereoscopic Displays," *Proc. 3DTV Conf.*, IEEE Press, 2007, pp. 1–4.
17. F. Duguet and G. Drettakis, "Flexible Point-Based Rendering on Mobile Devices," *IEEE Computer Graphics and Applications*, vol. 24, no. 4, 2004, pp. 57–63.
18. V. Setlur et al., "Retargeting Vector Animation for Small Displays," *Proc. 4th Int'l Conf. Mobile and Ubiquitous Multimedia* (MUM 05), ACM Press, 2005, pp. 69–77.
19. J. Huang et al., "Interactive Illustrative Rendering on Mobile Devices," *IEEE Computer Graphics and Applications*, vol. 27, no. 3, 2007, pp. 48–56.
20. C.-F. Chang, and S.-H. Ger, "Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering," *Proc. 3rd IEEE Pacific Rim Conf. Multimedia* (PCM 02), LNCS 2532, Springer, 2002, pp. 1105–1111.
21. D. Hekmatzada, J. Meseth, and R. Klein, "Non-Photorealistic Rendering of Complex 3D Models on Mobile Devices," *Proc. 8th Ann. Conf. Int'l Assoc. Mathematical Geology*, vol. 2, Alfred-Wegener-Stiftung, 2002, pp. 93–98.
22. X. Luo and G. Zheng, "Progressive Meshes Transmission over a Wired-to-Wireless Network," *Wireless Networks*, vol. 14, no. 1, 2006, pp. 47–53.
23. A. Boukerche and R.W.N. Pazzi, "Performance Evaluation of a Streaming-Based Protocol for 3D Virtual Environment Exploration on Mobile Devices," *Proc. Int'l Symp. Modeling Analysis and Simulation of Wireless and Mobile Systems* (MSWiM 06), ACM Press, 2006, pp. 20–27.
24. S. Chattopadhyay, S.M. Bhandarkar, and K. Li, "Human Motion Capture Data Compression by Model-Based Indexing: A Power Aware Approach," *IEEE Trans. Visualization and Computer Graphics*, vol. 13, no. 1, 2007, pp. 5–14.
25. L. Chittaro, "Visualizing Information on Mobile Devices," *Computer*, vol. 39, no. 3, 2007, pp. 40–45.
26. P. Baudisch and R. Rosenholtz, "Halo: A Technique for Visualizing Off-Screen Objects," *Proc. SIGCHI Conf. Human Factors in Computing Systems* (CHI 03), ACM Press, 2003, pp. 481–488.
27. K. Hornbaek and M. Hertzum, "Untangling the Usability of Fisheye Menus," *ACM Trans. Computer–Human Interaction*, vol. 14, no. 2, 2007, article 6.
28. T. Capin and A. Haro, "Mobile Camera Based User Interaction," *Handbook of Research on User Interface Design and Evaluation for Mobile Technology*, Information Science Reference, 2008, pp. 541–555.
29. B. Bederson et al., "Datelens: A Fisheye Calendar Interface for PDAs," *ACM Trans. Computer–Human Interaction*, vol. 11, no. 1, 2004, pp. 90–119.
30. A.K. Karlson, B.B. Bederson, and J. Sangiovanni, "AppLens and launchTile: Two Designs for One-Handed Thumb Use on Small Devices," *Proc. SIGCHI Conf. Human Factors in Computing Systems* (CHI 05), ACM Press, 2005, pp. 201–210.
31. B. Shneiderman, and C. Plaisant, *Designing the User Interface*, 4th ed., Addison-Wesley, 2004.
32. S.J.V. Nichols, "New Interfaces at the Touch of a

Fingertip," *Computer*, vol. 40, no. 8, 2007, pp. 12–15.
33. J. Pascoe, N. Ryan, and D. Morse, "Using While Moving: HCI Issues in Fieldwork Environments," *ACM Trans. Computer-Human Interaction*, vol. 7, no. 3, 2000, pp. 417–437.
34. K. Hinckley et al., "Sensing Techniques for Mobile Interaction," *Proc. 13th Ann. ACM Symp. User Interface Software and Technology* (UIST 00), ACM Press, 2000, pp. 91–100.
35. J. Hannuksela, P. Sangi, and J. Heikkilä, "Vision-Based Motion Estimation for Interaction with Mobile Devices," *Computer Vision and Image Understanding*, vol. 108, nos. 1–2, 2007, pp. 188–195.
36. S. Feiner et al., "A Touring Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment," *Proc. 1st Int'l Symp. Wearable Computers*, IEEE CS Press, 1997, pp. 74–81.
37. R. Azuma, "A Survey of Augmented Reality," *Presence: Teleoperators and Virtual Environments*, vol. 6, no. 4, 1997, pp. 355–385.
38. R. Azuma et al., "Recent Advances in Augmented Reality," *IEEE Computer Graphics and Applications*, vol. 21, no. 6, 2001, pp. 34–47.
39. J. Rekimoto and K. Nagao, "The World through the Computer: Computer Augmented Interaction with Real World Environments," *Proc. 8th Ann. ACM Symp. User Interface and Software Technology* (UIST), ACM Press, 1995, pp. 29–36.
40. D. Wagner and D. Schmalstieg, "First Steps towards Handheld Augmented Reality," *Proc. 7th IEEE Int'l Symp. Wearable Computers* (ISWC 03), IEEE CS Press, 2003, pp. 127–136.
41. M. Möhring, C. Lessig, and O. Bimber, "Video See-Through AR on Consumer Cell-Phones," *Proc. 3rd IEEE and ACM Int'l Sym. Mixed and Augmented Reality* (ISMAR 04), IEEE Press, 2004, pp. 252–253.
42. A. Henrysson, M. Billinghurst, and M. Ollila, "Face to Face Collaborative AR on Mobile Phones," Proc. *4th IEEE and ACM Int'l Symp. Mixed and Augmented Reality* (ISMAR 05), IEEE Press, 2005, pp. 80–89.
43. E. Bruns et al., "Enabling Mobile Phones to Support Large Scale Museum Guidance," *IEEE MultiMedia*, vol. 14, no. 2, 2007, pp. 16–25.
44. H. Bay, B. Fasel, and L. Van Gool, "Interactive Museum Guide: Fast and Robust Recognition of Museum Objects," *Proc. 1st Int'l Workshop Mobile Vision*, Springer Verlag, 2006.
45. W.-C. Chen et al., "Efficient Extraction of Robust Image Features on Mobile Devices," *Proc. Int'l Symp. Mixed and Augmented Reality* (ISMAR 07), IEEE Press, 2007, pp. 281–282.

Tolga Capin *is an assistant professor in Bilkent University's Department of Computer Engineering. He has contributed to various mobile graphics standards. His research interests include mobile graphics platforms, human–computer interaction, and computer animation. Capin received his PhD in computer science from the Ecole Polytechnique Federale de Lausanne. Contact him at tcapin@cs.bilkent.edu.tr.*

Kari Pulli *is a research fellow at Nokia Research Center. He has been an active contributor to several mobile graphics standards and recently wrote a book about mobile 3D graphics. Pulli received a PhD in computer science from the University of Washington and an MBA from the University of Oulu. Contact him at kari.pulli@nokia.com.*

Tomas Akenine-Möller *is a professor in Lund University's Department of Computer Science. His research interests are graphics hardware for mobile devices and desktops, new computing architectures, collision detection, and high-quality rapid rendering. Akenine-Möller received his MSc in computer science and engineering from Lund University and his PhD in graphics at the Chalmers University of Technology. He received the best paper award at Graphics Hardware 2005 with Jacob Ström for the ETC texture compression scheme, which is now part of the OpenGL ES API. Contact him at tam@cs.lth.se.*