Interactive Illustrative Rendering on Mobile Devices

Scientists, engineers, and artists regularly use illustrations in design, training, and education to display conceptual information, describe problems, and solve those problems. Offering such practitioners readily accessible visual tools for automated illustration has been an important challenge in computer graphics. Researchers have developed many advanced rendering techniques on desktop platforms to facilitate illustration generation, but adapting these techniques to mobile platforms has not been easy. Compared to their desktop counterparts, mobile devices have many hardware limitations, including low screen resolution, limited input interfaces and battery life, low system-bus bandwidth, slow CPU clock speed, limited storage capacity, and a lack of advanced graphics hardware.

To address mobile devices’ hardware issues, some researchers and developers use remote visualization. This approach employs a client-server model in which the server-side provides most of the rendering tasks, while the mobile client simply receives the streamed images. Remote visualization is most popular for displaying complex 3D surface models and volumetric data on mobile platforms. To achieve a satisfactory performance on remote clients, the approach relies on a network connection, which might not always be available, or networking bandwidth, which might be inadequate.

In recent years, the computational power of mobile devices such as PDAs and cell phones has significantly increased. Continuous improvements in processors, graphic chips, displays, power management, and wireless technology have greatly enhanced mobile devices’ programmability and usability. Moreover, some vendors have released standardized software platforms for developing 3D graphical applications on mobile devices—such as OpenGL ES and Direct3D Mobile—which function as subsets of their ancestral, desktop APIs. With these improvements in both hardware and software platforms, it’s become more feasible for users to perform visualization tasks, including interactive illustrations, using local resources on mobile devices. However, such tasks still pose difficulties.

Here we discuss how advanced illustrative rendering techniques, such as interactive cutaway views, ghosted views, silhouettes, and selective rendering, have been adapted to mobile devices. We also present MobileVis, our interactive, illustrative 3D graphics and text rendering system that lets users explore 3D models’ interior structures, display parts annotations, and visualize instructions, such as assembly and disassembly procedures for mechanical models.

Illustrations in traditional visualization

Technical manuals, scientific illustrations, textbooks, and encyclopedias have adopted a variety of illustrative techniques (see Figure 1) to convey object shapes and forms, reveal complicated models’ interior structure, and describe procedural steps. Inspired by these techniques, computer graphics researchers have developed many rendering algorithms to automate some of these illustrative conventions on desktop platforms. Viola and colleagues summarize some current techniques for visualizing volumetric models’ internal structures, including cutaway and ghosted views, importance-driven rendering, and exploded and deformation views.

All of these techniques, however, require recent graphics cards, processors, and programming interfaces; these are readily available on desktop platforms, but unavailable on mobile devices such as PDAs and cell phones. This computational-capabilities gap will continue to be a problem: Mobile device battery capacities generally improve at a much slower rate than CPU and graphics chips, which forces hardware (especially pro-
cessing units) to be clocked at much lower rates. Still, to give mobile users the same functionality they have enjoyed with desktop 3D graphics applications, we must adapt these desktop-based illustrative rendering algorithms to mobile devices.

**Local visualization on mobile devices**

Although considerable literature covers remote visualization for mobile devices, the amount of published work on local 3D visualization on such devices is limited. This is because few mobile devices target graphical applications. Until a few years ago, most handheld devices could barely render a Gouraud-shaded cube, let alone perform demanding and complicated visualization tasks. Today, however, mobile devices have developed to the point where direct 3D rendering at interactive rates is feasible. With the release of OpenGL ES, the first standardized 3D programming interface for mobile devices, it’s becoming more common for mobile game developers to use OpenGL ES and Mobile 3D Graphics (M3G) to quickly develop 3D Java games. Still, R&D on mobile 3D visualization applications falls far behind.

Pocket Cortona (http://www.parallelgraphics.com/products/cortona), which has been used in the construction industry, is one of the few available viewers for visualizing 3D models on PDAs. The software requires VR Modeling Language models, however, and has limited surface shading modes for rendering. IBM researchers developed a 3D CAD model viewer for PDA-based product data management. The system consists of a file format converter, a scene graph library, and a triangle rasterizer. However, the rendering modes for viewing 3D models are also limited to surface and wireframe.
The MobileVis system

We developed MobileVis in C/C++ using OpenGL ES 1.0, a subset of the OpenGL API for embedded systems. OpenGL ES offers a lightweight interface for developing 3D graphics applications on low-end devices for several reasons. OpenGL ES developers removed many functions from the original OpenGL API, including the `glBegin()` and `glEnd()` semantics; OpenGL ES favors vertex arrays, which generate less overhead for large data sets. The developers also added functions, the most significant being fixed-point data types. These data types support the computational capabilities of embedded processors, which often lack floating-point processing units. Many of OpenGL's computationally intensive features, such as multitexturing and dynamic OpenGL state retrieval, are either removed or become optional in OpenGL ES, depending on the vendor's implementation.

Our MobileVis system consists of three modules:

- **a data transcoder** that converts 3ds Max models into a customized binary format to quickly load models on mobile devices,
- **a renderer** equipped with a toolbox of illustrative rendering modes, and
- **a labeling system** that dynamically displays labels and text information associated with 3D model parts.

Data structures and transcoder

Because most mobile devices don’t have floating-point processing units, we created a data transcoder that converts 3D models from floating point to a binary format using only signed and unsigned short integers. Each vertex’s data consist of a position (x, y, z), a normal (nx, ny, nz), and a color/opacity (r, g, b, a). We represent each position and normal component with a two-byte signed short integer and each color/opacity component with one byte. Thus, we need a total of 16 bytes per vertex. To minimize floating-point operations, MobileVis uses fixed-point math to perform all computations, including projection transformations and model view matrices.

OpenGL ES supports vertex arrays only for sending vertices to the graphics pipeline. Because we use many vertices multiple times at the rendering phase, we use indexed vertex arrays. We thus maintain a vertex index list that follows the order of the original model’s triangle list. We represent each element in the new vertex index list as an unsigned short integer.

To retain the hierarchical position information of an original 3D model’s different parts, we process the model in a predefined order and store the converted vertex array data sequentially. To identify the vertex indices of an object in the vertex index list, we keep an array of each object’s triangles. With this array, we can easily locate the selected object’s vertex indices by summing the total triangles of all objects previously stored in the vertex index list.

Interactive illustrative rendering

We’ve developed a toolkit for interactive exploration of 3D polygonal models on mobile devices. The toolkit includes the following illustrative rendering styles: silhouettes, selective rendering, cutaway views, ghosted views, magic lens views, offset examination views, and animations.

**Silhouettes.** Silhouettes are important for conveying a 3D object’s shape and spatial relationships. With polygonal models, silhouettes are defined as mesh edges that share a front- and a back-facing polygon. Isenberg and colleagues categorize existing silhouette algorithms into three groups: image-space, object-space, and hybrid. They recommend image-space hybrid algorithms when developers need interactive frame rates or devices have limited memory or slow processors. However, image-space algorithms usually require reading and operating on a z-buffer, which is inaccessible in the OpenGL ES profile. Therefore, we implemented a modified hybrid algorithm based on the Raskar and Cohen two-pass silhouette algorithm.

OpenGL ES doesn’t offer line drawing via polygon mode `glPolygonMode()`, so we rendered polygons in wireframe mode in the silhouette algorithm’s second pass. As Figure 2 shows, however, this adaptation generates line artifacts—the polygonal mesh’s wireframe edges—on relatively large, flat surfaces. To remove the artifacts, we used a silhouette-detection technique modified from an existing edge buffer method. Rather than using two bits per edge (as the original algorithm proposed), we stored one bit per edge. Our two-pass silhouette algorithm works as follows:
1. Initialize all the bits in the bit buffer to 0.
2. Identify and mark silhouette edges by going through all of the mesh edges and flipping bits corresponding to all back-facing triangle edges. As a result, the bits corresponding to silhouette edges will have a value of 1 because they’re flipped only once; backside edges are flipped twice (and become 0), and front-side edges are unmodified (and remain 0).
3. Perform two-pass rendering. In the first pass, draw front-facing polygons with a slight offset of the polygon nearest the viewer; set the depth buffer mask to TRUE and the frame buffer mask to FALSE. In the second pass, change depth and frame buffer masks and draw the silhouette edges that the previous step found, setting a pass condition in the depth test to “less than or equal to.”

Selective rendering. Silhouettes let users easily perceive object shapes. By combining silhouettes with other rendering styles, we can easily create a focus view of important model features, while still conveying overall shapes and context information. Figure 3 shows an example in which we used silhouette rendering mode for the 3D models’ overall shapes, and surface and transparency rendering modes for interior feature objects. We rendered both images in our MobileVis system on a PDA with 240 × 320 display resolution. To do this, we first specified a rendering style, then clicked on the objects to be rendered. Because OpenGL ES doesn’t intuitively support selection mode, we rendered object IDs into the frame buffer with depth test enabled, then read the color buffer back.

Cutaway views. Desktop platforms use various cutaway illustration approaches, such as constructive-solid-geometry-based cutouts, stencil buffer-based cutouts, and texture-based cutouts. However, for mobile devices, such techniques are problematic. Some methods (such as CSG operations) are too computationally expensive. Others require a stencil buffer or hardware-accelerated fragment operations, and most mobile devices—including the Dell Axim x51v PDA we used in our tests—don’t support either (although OpenGL ES 1.0 does support the stencil test). We therefore implemented a selective planar cutout technique for cutaway illustration in MobileVis. We offer users an interface to dynamically enable and disable $n (1 \leq n \leq 5)$ clipping planes and individually control their positions. In the cutaway mode, MobileVis applies clipping planes on user-selected objects, which it maintains in a cutaway list and renders $n$ times. Each time, it activates only the selected clipping plane. It renders unpicked objects once before rendering selected objects.

Figure 4 shows two cutaway-view examples. In both, we combined the cutaway views with selective rendering to reveal the polygonal models’ interior structures. We implemented the cutout-surface capping—forming the surface shading by clipping the plane and the object the plane clips—by having MobileVis perform an extra rendering pass on the polygonal model’s back faces, with lighting disabled and depth testing enabled.

Ghosted view. The ghosted view rendering technique is similar to that in cutaway views. Rather than do a hard-edge cutaway, however, we gradually reduce opacity in a selected occluding region to allow a transparent view of interior objects while still preserving occluding object features, such as edges. Ghosted views help users better understand interior structures and their spatial relations to contextual objects. In MobileVis, we implement ghosted viewing by modulating the vertices’ opacity values near a user-selected point in an object. We modify the opacity of the target object’s vertex as

$$
\alpha = \begin{cases} 
1, & d \geq r \\
\frac{d}{r}, & \alpha_o < d < r \\
\alpha_o, & d \leq \alpha_o r 
\end{cases}
$$

(1)

where $d$ is the distance between the vertex and the user-selected triangle’s central point; $\alpha_o$ is a predefined minimal opacity value between 0 and 1 (such as 0.15), and $r = 0.25 * \min(\dim_x, \dim_y, \dim_z)$, where $\dim_x, \dim_y, \dim_z$ are the sizes of $x, y, z$ dimensions. Figure 5 shows an example of ghosted views.

Magic lens. It’s possible to view a magic lens as a magnifying-glass extension and use it to illustrate and emphasize the details in a part of the model. In so doing, you can show the objects displayed under the lens in different rendering styles or with varying degrees of con-
textual details. In MobileVis, we magnify the model portion under the lens and also assign the magic lens see-through capability. We do this because many mechanical 3D models have a shell or covering object that is relatively large and less interesting—that is, it contains fewer features—than the interior objects it occludes. Therefore, for the magic lens view, we define a function that decides whether the portion of an object under the magic lens should be drawn. If the area an object covers under the magic lens exceeds a threshold value, it’s likely to be a shell or covering object that occludes potentially interesting model details. In such cases, we remove the object portion that’s under the magic lens and magnify the occluded objects.

By experimenting on testing models, we obtained an optimal threshold value and thus defined the visibility function as Equation 2:

\[
\text{visibility}_i = \begin{cases} 
1, & \frac{c_i}{S} \leq 0.3 \\
0, & \frac{c_i}{S} > 0.3 
\end{cases}
\]

where \(c_i\) is the coverage (in pixels) of the object \(i\), and \(S\) is the lens’s area size. Figure 6a shows an example of a magic lens view. In this image, the car’s driver-side body cover occults the view of the car’s interior objects. Through a magic lens, MobileVis uses Equation 2 to define the cover portion under the lens as an occluding object and thus removes it. It then exposes and magnifies the car’s interior structures. The lens magnification factor in the magnified image is 1.7.

Offset examination. Illustrators often spatially displace different model parts to uncover prominent features inside a model. The offset parts can either be displaced toward the viewer (and retain their relative spatial relationships) or spread out on the plane perpendicular to the viewing vector, and thus form an exploded view (see Figure 1). In the first case, the displacement typically lets users effectively illustrate and emphasize a focal area within a 3D model. In MobileVis, we implement an offset examination mode in which the
system gradually and interactively zooms user-selected objects toward the viewer. In addition to spatial displacement, we also magnify the target objects. Furthermore, users can transform offset objects to closely examine their parts without affecting other model objects (see Figure 6b).

**Peeling and animation.** Traditional artists visualize procedure information, such as a CAD model’s assembly and disassembly sequences, using either

- exploded views with sequence numbers and text explanations of procedural steps, or
- arrow procedural views with arrows from one step’s image to the next (as in Figure 1).

In computer graphics, artists often use animation when they can achieve an interactive frame rate of rendering. In our MobileVis system, we implemented an animation mode to visualize procedure information. Users can specify an assembly/disassembly sequence by entering a peeling order that lets them interactively select and remove individual model objects, clicking and dragging them with a stylus or a mouse. The system also records the sequence and can play it back on request.

**Annotations**

We integrated a dynamic labeling system into MobileVis so users can display the parts labels interactively (see Figure 7 on the next page). We define a label as a tuple consisting of the label text, an anchor position on the corresponding model, and the screen space position for the label. Because MobileVis is an interactive system, labels must dynamically reposition themselves as the view and objects change.

The labeling system can parse label data both from manually created XML files or from 3ds models directly. When using manually defined data, the user must define the object’s anchor points that the labels refer to, as well as the label text. If the labels are parsed directly from the 3ds model, MobileVis calculates the anchor position by taking a centroid of the respective object part and placing the anchor at the centroid.

Users can view labels in three viewing modes:

- **Internal:** MobileVis places labels relatively close to the anchor point. It avoids label-to-label occlusions by detecting label clusters and repositioning them using an optimization procedure.
- **Ring-based:** MobileVis calculates a circular extent around the target object using 2D projections of the object’s bounding box points and dynamically...
updates the circle as the view changes. It places labels uniformly on the ring by determining the closest position to the projected anchor point.

**Flush left/right:** MobileVis uses a similar technique to the ring-based mode to position labels, but uniformly distributes the labels on the bounding box's left- and right-most extents.

Because the MobileVis labeling system works on displays with varying resolutions, we define label size as a function of the target screen resolution. This procedure is reasonably simple because the labels are texture-mapped and scale easily.

### Results

We tested our system on a Dell Axim x51v PDA with 240×320 screen resolution, an Intel 624-MHz XScale CPU, 64 Mbytes of RAM, 16 Mbytes of video memory, and Microsoft Windows Mobile 5.0. Once the operating system booted, the PDA had about 30 Mbytes of memory available to run applications. Figure 8 shows our system running on the PDA.

Table 1 shows the number of vertices and triangles in our 3D test models and how MobileVis performed in different rendering modes. We used several testing models, including an auxiliary power unit from a Boeing 777 data set, which also supplied the mechanical part (“B-part”). Figure 4 shows cutaway views from both of these models.

The first column of Table 1 shows several primitive rendering modes, including point and surface modes, and various illustrative rendering modes. To obtain the performance statistics for the selective rendering mode, we defined a 3D model's major occluding object in silhouette mode and its other parts in surface mode. Figure 3a shows an example of the setup. Similarly, for the ghosted viewing mode, we applied the opacity modulation at the vertices only to a relatively large occluding object, even though MobileVis lets users apply modulations to multiple objects simultaneously.

In the cutaway view mode, we used two clipping planes and drew the models in surface mode. Because the number of rendering passes in cutaway view is the same as the number of enabled clipping planes, MobileVis performance in cutaway mode varies depending on the total number of activated clipping planes. In the offset examination view mode, we selected one or two model parts and offset them for close examination.

Figure 9a plots MobileVis performance for 3D models in varying rendering modes. As model size increases, the frame rate drops. On a PDA, one significant factor that directly affects rendering performance—in addition to low CPU clock rate, limited memory, and slow bus speed—is the size of the instruction and data caches. If the code and data being executed fits into the cache, the system can operate at the CPU’s full clock speed. Otherwise, it will constantly access memory and can’t execute.
at full speed. Our test PDA had an Intel XScale CPU with a 32-Kbyte instruction cache and a 32-Kbyte data cache. As we mentioned earlier, the vertex data structure we implemented requires 16 bytes per vertex. Therefore, a geometry buffer with more than 2,048 vertices will overflow the data cache. Also, because the OpenGL ES API requires that we use vertex and color arrays for all rendering tasks, we keep a list of vertex indices for all model objects. MobileVis constantly calls on this index list during rendering, which also affects performance.

As Table 1 shows, the silhouette rendering mode is relatively slow compared to other modes. This is because it requires view-dependent computations to find potential silhouette edges in Step 2 of the silhouette algorithm. MobileVis must fully traverse the mesh at this step and then traverse it twice in Step 3. In addition to this multiple-pass rendering, the PDA’s limited memory and cache sizes further deteriorate performance.

In addition to the PDA tests, we tested our system on a desktop platform using OpenGL 2.0. The desktop had a 1.6-GHz Pentium M CPU, 512 Mbytes of RAM, and an Nvidia GeForce 6200 graphics card. Figure 9b plots our system’s desktop rendering performance; the desktop frame rates are generally 2 to 9 times higher than those on the PDA.

Future work

For future work, we plan to improve some of our illustration techniques’ performance on mobile devices. We’ll also run a user study with Air Force Research Laboratory to evaluate different illustration techniques’ effectiveness for rendering multiresolution 3D models for training and maintenance. Based on the evaluation results, we’ll extend our system to allow semi-automatic rendering style selection and graphical-representation generation at different levels of detail.

Acknowledgments

We thank Boeing Company for generously providing us with a Boeing 777 data set, and Matthias Deller of the German Research Center for Artificial Intelligence for supplying the V8 engine data set. We also thank Professor Yung-Hsiang Lu from the Electrical and Computer Engineering Department at Purdue University for his help on this project. Our work was supported by US Air Force grant AFRL-WS 06-1469, FA8650-05-2-6648; National Science Foundation Information Technology Grant AFRL-WS 06-1469, FA8650-05-2-6648; and the German Research Center for Artificial Intelligence with a Boeing 777 data set, and Matthias Deller of the German Research Center for Artificial Intelligence for supplying the V8 engine data set. We also thank Professor Yung-Hsiang Lu from the Electrical and Computer Engineering Department at Purdue University for his help on this project. Our work was supported by US Air Force grant AFRL-WS 06-1469, FA8650-05-2-6648; National Science Foundation Information Technology Grant AFRL-WS 06-1469, FA8650-05-2-6648; and the German Research Center for Artificial Intelligence with a Boeing 777 data set.

References


Jingshu Huang is a PhD student in the School of Electrical and Computer Engineering at Purdue University. Her research interests include scientific visualization, 3D graphics, animations, rendering, and digital entertainment. Huang has an MS in educational technology from Purdue University. Contact her at jhuang2@purdue.edu.
Brian Bue is an associate member of the Information Systems and Computer Science staff at the NASA Jet Propulsion Laboratory. His research interests include information visualization and visual analytics, with emphasis on mobile applications. He has an MS in computer science from Purdue University. Contact him at brian.d.bue@jpl.nasa.gov.

Avin Pattath is a PhD student in the PURPL Lab at Purdue University. His research interests include mobile visualization, visual analytics, computer graphics, and computer vision. He is a student member of ACM Siggraph. He can be reached at orapattath@purdue.edu.

David S. Ebert is a professor in the School of Electrical and Computer Engineering at Purdue University and director of the Purdue University Regional Visualization and Analytics Center. His research interests are in scientific, medical, and information visualization, computer graphics, and animation, and in procedural techniques, including volume rendering, illustrative visualization, visual analytics, mobile rendering, procedural texturing, modeling, animation, and modeling natural phenomena. Ebert has served on the ACM Siggraph Executive Committee and as editor in chief for IEEE Transactions on Visualization and Computer Graphics. He is editor and coauthor of Texturing and Modeling: A Procedural Approach, 3rd ed. (Morgan Kaufmann, 2003) He has a PhD from the Computer and Information Science Department at the Ohio State University. Contact him at ebertd@purdue.edu.

Krystal M. Thomas is an industrial engineer in the US Air Force Research Laboratory, Wright-Patterson Air Force Base, Ohio. Her research interests include passive data collection applications and immersive training environments. She has a BS in industrial and systems engineering from Wright State University. Contact her at krystal.thomas@wpafb.af.mil.