Augmented Reality on Mobile Phones

Daniel Wagner Lukas Gruber Dieter Schmalstieg

Graz University of Technology

Figure 1. Examples of Augmented Reality on phones. Left: a virtual character on a square marker. Right: a virtual animal on a tracked playing card. In both cases the live camera feed is drawn as a video background and the virtual object is rendered on top of it.

1 Introduction

Our motivation comes from research on a novel kind of user interface called Augmented Reality (AR). The real world as viewed with a camera is augmented with virtual objects, which are spatially registered in the scene. Probably the most widely recognized example is the "Eye of Judgment", a computer/board game for Sony's Playstation 3[1] that puts virtual game characters on play cards, similar to the image shown in Figure 1, right.

Recently, mobile phones with cameras have become attractive as inexpensive AR devices. Mobile phones have a market penetration of 100% in many industrial countries now, and their graphics performance is constantly increasing, primarily driven by the desire for compelling mobile games. Unfortunately, the phone market is highly fragmented in terms of platforms and capabilities, so that delivering a graphics application to a wide user base is a very tedious task.

What is interesting about our work to develop AR experiences on mobile phones is that until recently, AR was considered too demanding to be executed on mobile phones due to the requirement to simultaneously perform video acquisition, image processing and graphics synthesis in real-time. We have recently shown that compelling AR applications are possible on phones [schmalstieg-ismar07]. The solutions we have developed are not limited to AR, but can be applied to any high performance graphics/video application.

The purpose of this article is to provide some insight in how to approach performance sensitive issues for mobile phone development, and how to obtain sufficient platform independence to allow dissemination for a reasonable large user base. First, we describe the special needs and techniques of AR applications. The article continues with issues and solutions when developing for Symbian and Windows Mobile, OpenGL ES and Direct3D, software and hardware rendering and scene-graph as well as video processing issues.

2 Developing Augmented Reality Applications

Augmented Reality combines real and virtual by accurately combining both in a singe view. This works by establishing a single coordinate system that is shared between the real and virtual world. The strategy is to first estimate the position and orientation (pose) of the real camera in the real world. The

[1] http://www.us.playstation.com/EyeofJudgment/

virtual camera is then set at the very same pose as the real one and hence looks at the same part of the (virtual) scene. Therefore, when drawing virtual content, it shows up at the same place in the rendered image as a real object would do in the real world from the point of view of the real camera.

2.1 Mainloop of an AR Application

Augmented Reality applications share many features with typical games. Both aim at running at maximum speed with minimum delay to user input and both render a mixture of 2D and 3D graphics. In this chapter we outline the typical mainloop of an AR application as shown in Figure 2, including tracking, rendering and network communication.

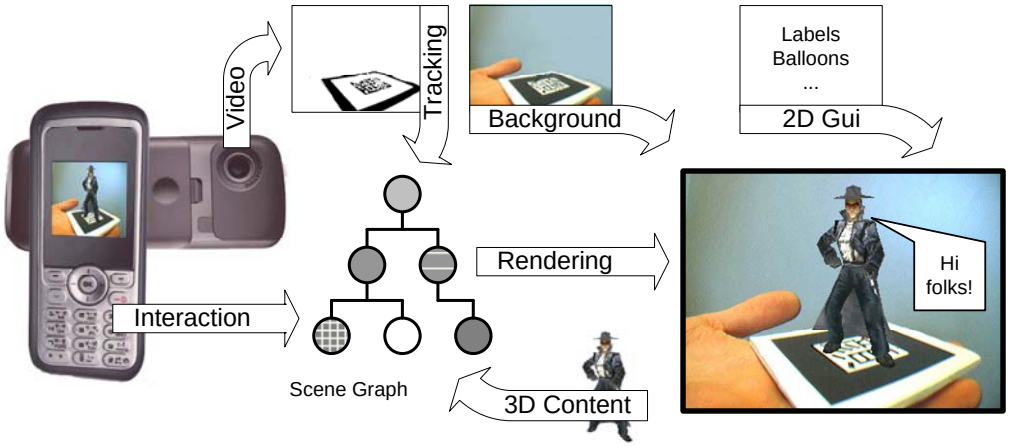

Figure 2. Dataflow in Augmented Reality on Mobile Phones

Pre-render actions. Similar to games, a new frame starts by reacting to user input as well as other data, such as coming in over the network and sending out new requests. Due to the highly graphical focus, the pre-render actions most often modify the scene-graph, before it is rendered later on in the "3D rendering" step.

Network sending. Network communication is slow compared to other data flow inside the application. Hence it is advised to split networking up into a sending and receiving part: Data is sent out early in the frame and interleaved with computationally expensive actions until the reply is expected.

Camera image reading. Retrieving and processing video images from the camera is an integral part of every AR application, except for those using an optical see-through path, such as with some head-mounted displays (HMD). The received camera image must be converted into a format that suits the needs of the renderer as well as the pose estimator. In many cases this also includes up-scaling the color image to match the screen's resolution. Chapter 8 goes into details on performance techniques for image processing on mobile phones.

Pose estimation describes the process of calculating the devices position and orientation in 3D space so that virtual content can be put correctly next to real one. There are many different methods of estimating a pose of which most use special hardware such as magnetic, infrared or inertial sensors. Since these sensors are not available in mobile phones today, computer vision (analyzing the camera image) is the most obvious way to go. Chapter 2.3 provides an overview on pose estimation using markers.

Video background. Drawing the camera image as a background on the screen simulates a see-through capability as if the phone was a frame rather than a solid object. This method is also called "Magic Lens".

3D Rendering. Now that the video background, representing the real work is on the screen the application renders virtual content on top of it. To create a convincing augmentation, the virtual camera has to be setup to use the same parameters as the real one, as described in chapter 2.2. A

general problem with this method is that virtual content always shows up in front of the real one. Chapter 2.4 describes techniques to overcome this restriction and to effectively allow real objects (which are only available as flat pixels in the video background) correctly occluding virtual content.

2D Rendering. Similar to games, most AR applications also show 2D information as well as user interface items on the screen in the form of a head up display (HUD).

Frame Flipping. The creation of visual output finishes with displaying the current back buffer on the screen.

Network receiving. This step completes the network communication started in the "Network sending" step. The six steps since then included major processing tasks and typically take around 20-50 milliseconds to complete. Hence, much time has passed since network requests have been sent out. Replies from the server or peers are likely to have arrived by now so that no time is lost waiting for them.

2.2 Off-Axis Projection Camera

One of the most fundamental differences in rendering between typical 3D applications such as games or graphical editors and AR applications is the way the virtual camera is setup. In order to correctly place virtual next to real objects the virtual camera has to mimic the real camera's attributes, called the extrinsic and intrinsic parameters.

Extrinsic parameters describe a camera's location and orientation. If a virtual camera is located at the same position as a real camera and also points into the same direction, then it observes the same part of the scene as the real camera. Yet, this is not fully true, since we have not specified attributes such as "zoom" yet. Naturally, when zooming out a camera sees a larger part of a scene than when it is zoomed in. This is where the intrinsic camera parameters come into the game: They describe the camera's focal length (the "zoom factor"), the principle point (center of projection), the camera's resolution in pixels and finally the distortion of the camera's lens. Estimating the intrinsic parameters of a camera is a highly complicated issue and beyond of the scope of this article. Fortunately there exist tools such as the MATLAB camera calibration toolbox[2] that do the job very well and require only minimal insight. All that is required is taking a few pictures of a calibration pattern and the software gives us all the data we need.

To remain independent of any physical properties focal length and principle point are typically specified in pixels. Most people assume that the center of projection is in the middle of a camera's image. Yet, this is not true for most real cameras. In practice, the principle point is easily off by a few percent. Camera resolution, principle point and focal length are "linear" parameters and can therefore be modeled using a projection matrix. Lens distortion on the other hand is a non-linear effect and typically treated separately. Most AR applications only correct for radial distortion during the pose estimation (see next chapter), but ignore it during rendering.

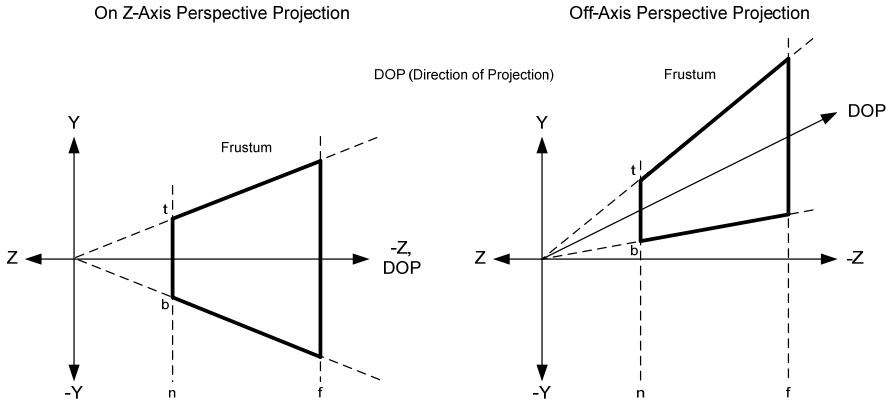

Figure 3. On-axis and off-axis camera models

[2] http://www.vision.caltech.edu/bouguetj/calib_doc/

To finally make the virtual camera see exactly the same part as the real one, we need to create a custom projection matrix (called off-axis projection matrix) that honors the real camera's intrinsic parameters. Figure 3 shows the differences between a regular "on-axis" and an off-axis camera. In the on-axis case, the viewing direction is perpendicular to the projection plane. Instead, in the off-axis case, the viewing direction is perturbed and the center of projection is not in the middle of the of the camera plane.

To calculate the projection matrix, it is more suitable to imagine the center of projection as fixed at the 0/0/0 coordinate and to shear the camera's viewing frustum. This way the only thing that changes when going from on- to off-axis are the coordinates of where frustum's side planes intersect with the near plane, marked with 'n' in Figure 3. Similarly, the intersections are marked with t (for top) and b (for bottom). The side planes intersect at the l (left) and r (right), but are not visible in Figure 3, since it shows the camera from the side.

OpenGL ES comes with a glFrustum() function that allows us to directly pass these n,f,t,b,l,r values to specify an off-axis projection matrix. The following code snippet shows how to calculate a 4x4 projection matrix directly from the intrinsic parameters. As always in this article we use row-major order, which means that the resulting matrix has to be transposed in order to be passed to OpenGL ES (find more on matrices in chapter 5).

```
// Calculating an off-axis projection matrix from intrinsic parameters
//
float matrix[16];
#define MAT(Y,X)  matrix[Y*4+X]

float dx = principalPointX-cameraWidth/2, dy=principalPointY-cameraHeight/2;
float x, y, a, b, c, d;

x =  2.0f * focalLengthX / cameraWidth;
y = -2.0f * focalLengthY / cameraHeight;
a = 2.0f * dx / cameraWidth;
b = -2.0f * (dy+1.0f) / cameraHeight;
c = (farPlane+nearPlane)/(farPlane-nearPlane);
d = - nearPlane *(1.0f+c);

MAT(0,0) = x;     MAT(0,1) = 0.0f;  MAT(0,2) = 0.0f;  MAT(0,3) = 0.0f;
MAT(1,0) = 0.0f;  MAT(1,1) = y;     MAT(1,2) = 0.0f;  MAT(1,3) = 0.0f;
MAT(2,0) = a;     MAT(2,1) = b;     MAT(2,2) = c;     MAT(2,3) = 1.0f;
MAT(3,0) = 0.0f;  MAT(3,1) = 0.0f;  MAT(3,2) = d;     MAT(3,3) = 0.0f;

#undef MAT
```

2.3 Pose Estimation

In the previous chapter we have seen how to setup a virtual camera to reflect a real camera's intrinsic parameters. What is still missing is how to calculate the real camera's location and orientation. This process is called pose estimation. In many AR applications pose estimation is done by the use of an attached camera and computer vision algorithms. To make things easier, artificial objects (markers), optimized for easy detection are deployed in the real scene. Figure 1 shows two different marker types.

To estimate a camera's pose a set of well known points must be detected in the camera image. For a planar case, 4 points are sufficient to uniquely calculate the camera's 3D position and orientation. Hence, many markers have a square shape, which allows using the 4 corners for pose estimation.

The problem of estimating a camera's pose using a square marker can be split into:

- Finding squares in the camera image;
- Checking all found squares for being valid markers; and
- Estimating the camera's pose relative to one or more valid markers.

The remainder of this chapter gives an overview on the aforementioned three steps. A detailed description goes beyond the scope of this article, but can be found in [Wag07].

Finding squares in the camera image typically starts by performing a thresholding operation that converts a color or grayscale image into pure black and white (see Figure 4). Next the software searches for closed contours. Using a pure b/w image a counter is found by scanning every line from left to right searching for changes from black to white or white to black. The contour is followed until it either ends at the image border or in the starting position (right image in Figure 4). Only in the latter case the contour is kept for further processing. Next, a square fitting algorithm rejects all contours which are not quadrilaterals. A quadrilateral is defined by having exactly four corners. So any contour with less or more than four corners is discarded. Finally the quadrilaterals are checked for convexity and minimum size.

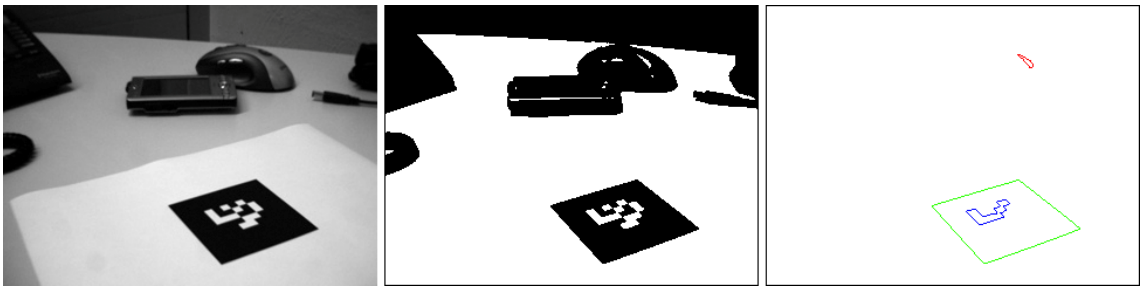

Figure 4. Thresholding and contour detection.

Checking found squares for being valid markers rejects those squares which do not contain a valid marker pattern. To check the pattern, it must be unprojected (unwarped) from the image into its original square shape. It is not sufficient to linearly interpolate the marker square since the pattern has been undergone a perspective projection. Hence, we need to find the homography, a 3x3 projection matrix that maps the unprojected 2D marker corners into the image. The homography is defined only up to scale and therefore has 8 degrees of freedom. Since any 2D point has two degrees of freedom (corresponding to its x- and y-coordinates) four points are sufficient to solve this linear system, which is typically done using singular value decomposition (SVD) or Cholesky factorization. For details please see [Wag07].

The homography matrix basically defines a local coordinate system that can be used to sample points (see Figure 5). Although the homography allows us to sample at an arbitrary resolution, it is not possible to reconstruct marker patterns of too small size. Generally a minimum size of roughly 20x20 pixels in the camera image is sufficient for robust marker detection.

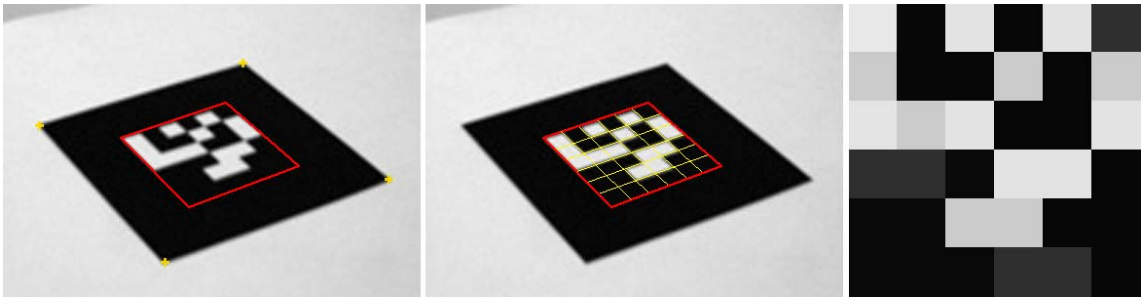

Figure 5. Marker pattern unprojection

Estimating the camera's pose can be done directly by using the homography calculated in the previous step. Yet, this provides only a coarse estimation and requires refinement to prevent jitter (virtual content would not remain stable on the marker). To gain better quality, the corner coordinates are now compensated for lens distortion. Furthermore, while the homography optimizes the algebraic error, we do now optimize the pose for minimal reprojection error. Since projection is a non-linear operation, a non-linear optimizer such as Gauss-Newton is required. Although the pose calculated from the homography is not highly accurate, it serves as a good starting point for the Gauss-Newton iteration. Hence, typically only two or three iterations are required.

A 6DOF pose is usually stored as a 3x4 matrix, where the left 3x3 sub-matrix represents the orientation and the right-most column vector presents the position. The representation is optimal for many purposes, since 3D points can be easily transformed by multiplying their homogenous form with the pose matrix. Yet, optimizing all 12 values of a 3x4 matrix with six degrees of freedom only is a waste of processing power. Hence, the pose is converted into a more compact 6-vector form before the refinement step and then back into its 3x4 matrix form. This 6-vector (see Figure 6) is composed of 3 values for position plus 3 rotation values of a quaternion (although a quaternion has four values, its length defined to be one and the fourth value can always be reconstructed). After refinement, the 6-vector is converted back to a 3x4 matrix that can be extended to a 4x4 matrix for rendering by adding a fourth line of (0 0 0 1).

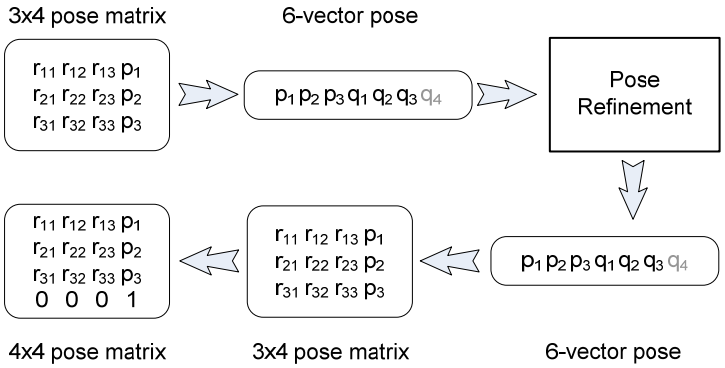

Figure 6. Pose representations during the pose refinement step

2.4 Handling Occlusion between real and virtual Objects

Seamless blending between the real and the virtual world is a fundamental target in Augmented Reality systems. This concerns a set of different topics like applying real world lighting onto virtual objects or computing realistic physical interactions. Furthermore correct occlusion handling between real and virtual objects is an important issue. The visual perception can be disturbed significantly by virtual objects, which occlude real world objects if the virtual object is actually placed behind the real world object (see Figures 7 and 8a). Ignoring occlusion handling can lead to wrong depth perception and important parts of the real world information could be lost. The problem is caused by the missing scene-depth information of the real world in the rendering pipeline.

Figure 7. Incorrect occlusion caused by missing scene-depth information.

Several approaches have been developed so far. There are two related methods how to handle occlusion between real and virtual objects[Bree96]. The first method (depth-based) tries to gather an entire depth map of the real world. Dynamic scene reconstruction based on multi-vision can create such a depth map, which is then employed in the render pass. The depth values are written directly into the depth buffer to perform a correct depth test with the virtual objects. Although this method reconstructs the depth map of the entire environment it has some serious drawbacks, especially for

mobile devices: Scene reconstruction by stereoscopic vision is based on a two-camera system, which is still uncommon on mobile devices. Moreover the scene reconstruction is a computational costly and not very stable task.

The second method (model-based) is more applicable to mobile phones. The idea is to create a so called “phantom” object, which represents a virtual counterpart of a real world object. The virtual phantom object is superimposed to its genuine. During rendering only the geometry of the “phantom” is rendered and no color output is created. Hence, the z-buffer is filled with the geometric properties of the supposed real world object and a correct occlusion can be computed. In Figure7a the virtual object V is theoretically occluded by the real world object A. Figure7b shows a “phantom” for object A. The result is a correct occlusion of the virtual object V.

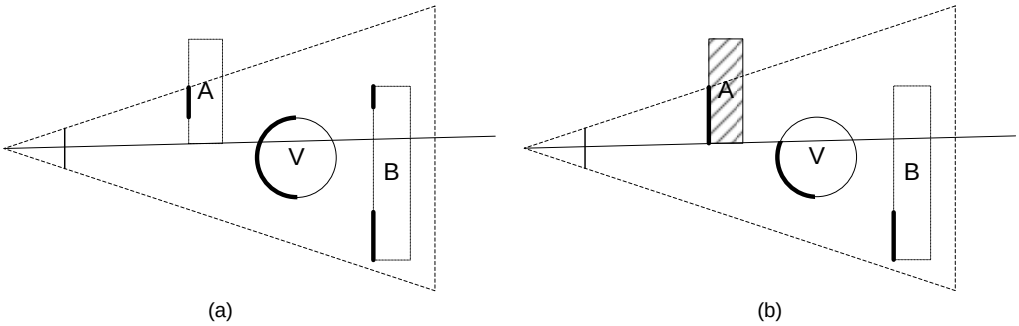

Figure 8. Visibility without (left) and with (right) a phantom object. Bold lines show visible edges.

The entire workflow for occlusion handling of static real world objects is presented in the following steps. In Figure 8a an example of a “phantom” object is given. Figure 8b shows the final results.

Create a “phantom” object of the real object model. Only the geometric properties are of interest. Textures and colors do not have to be defined.

Add the “phantom” object to the virtual scene. This additionally implies the possibility to add physical behavior to the augmented reality application. A collision detection system, for example, could be added to provide physical interaction between real world objects and virtual objects.

Superimpose the “phantom” object to the real world object. This task is called the “registration” step. As long as the real world object does not move the “phantom” can be placed relative to the computed pose (see Section 2.3). In case of a moving object the object itself has to be tracked indivudually.

The render pass can be subdivided into six separate tasks:

1. Clear the z-buffer
2. Clear the color buffer
3. Disable RGB color writing
4. Render “phantom” objects
5. Enable RGB color writing
6. Render virtual objects

(a) (b)

Figure 9. The left image shows a superimposed "phantom" object. The right image shows the final result of the occlusion handling.

3 Platform considerations

The mobile phone market of today is very fragmented in terms of operating systems. Standardized operating systems such as Symbian and Windows Mobile are targeted at the higher end of the market, while the lower end is dominated by proprietary, closed operating systems which can only be extended with Java applications. Java is problematic in that it does not allow performance-critical operations close to the hardware, while the remaining platforms such as iPhone, Blackberry or Android have proprietary or immature development models. In this article, we are only reporting on our experiences with Windows Mobile and Symbian.

OpenGL ES has been widely accepted as the cross-platform graphics API of choice for mobile phones. It is supported in Windows Mobile, Symbian, iPhone and other platforms. First mobile phones with hardware support for the latest OpenGL ES 2.0 standard are expected in 2008/2009. However, Windows Mobile also supports Direct3D Mobile, a direct competitor of OpenGL ES. Direct3D Mobile[3] was introduced with Windows CE 5.0 (the basis for all current Windows Mobile versions), has a similar feature set to Direct3D version 6 and is only available on Windows Mobile phones. In contrast to OpenGL ES, its API is object oriented, but targets exactly the same feature set as OpenGL ES 1.1. Although Direct3D is inferior to OpenGL ES in most ways, some phones only come with Direct3D drivers for hardware accelerated rendering. Hence it is suggested to support both APIs for a broad coverage of target devices.

Finally, M3G is a scene-graph API for Java that is implemented on top of OpenGL ES. Java code typically runs 4-10x slower on mobile phones than native code [Pul05]. Hence, most M3G renderers are implemented in native code, preinstalled by the OEM, whereas only the application itself is written in Java. Since M3G is a high-level API, only little Java code must be executed. Unfortunately, AR requires executing computationally expensive code (e.g. image processing), which is too slow to run in Java on phones. This problem is acerbated as Java-only phones have usually lower end hardware, and therefore a Java-based approach was considered not useful for our purposes.

Mobile phone hardware is mostly driven by space and battery power considerations and hence very different from today's desktop computers [Möl08]. CPUs based on ARM-designs are predominant. Except for some high-end devices, these CPUs do not have floating point units. Floating point math must therefore be emulated which is around 40 times slower than integer math. If present, the GPU is typically part of the mobile phone CPU today, which also means that GPU and CPU share the same memory.

Basically any modern mobile phone today has some kind of hardware graphics acceleration. Yet, most phones today can only accelerate only 2D operations for GUI drawing. Even more the respective hardware units are not directly accessible. Hence, when we write about devices with and without a GPU, we actually mean with and without hardware 3D acceleration.

[3] http://msdn.microsoft.com/en-us/library/aa452478.aspx

There are three major obstacles that a platform independent render engine has to overcome in order to run on a majority of mobile phones as well as on the PC (mostly for development and testing). These are

- Operating System: There is a large and growing number of operating systems currently in use on mobile phones. For simplicity, this article only considers Windows Mobile and Symbian as well as Windows XP.
- Graphics Library: As mentioned earlier, OpenGL ES is available on every platform, but often only as a software implementation. While this represents a highly portable fallback solution, Direct3D Mobile must also be supported for maximum device coverage.
- Hardware vs. software rendering: Not considering the inherent differences between hardware and software renderer implementations can easily make a performance drop of 50%.

All three topics listed above require careful consideration when aiming for a portable render engine. Additionally to that Augmented Reality applications have require fast video retrieval and processing from the built-in camera, as discussed in the next section.

4 Application Initialization

So far, most of the concepts and code snippets presented where actually completely OS independent (ignoring the fact the Direct3D is only available on Windows platforms). The major difference between the platforms lies in the way the application window is created. Additionally support for user input, file access, sound etc. have to be considered, but are not discussed here.

Even though an application might run in full screen so that no "window" is visible, most modern operating systems still bind an application to a window e.g. for managing system events. To create a full screen application the window is then either enlarged to cover to whole screen (including the omnipresent title and button bars) or the application switches the display into a "real" graphics mode. Some hardware accelerated implementations even require running in full screen for optimal performance.

4.1 Window creation on Windows XP and Mobile

Creating a window for rendering on Windows XP and Mobile is very similar and requires a call to a single API function:

```
// Code fragment to create a rendering window on Windows Mobile.
//
window = CreateWindow(”MyWindowClass”,”MyApplication”,WS_VISIBLE,
                          CW_USEDEFAULT,CW_USEDEFAULT,
                          GetSystemMetrics(SM_CXSCREEN),
                          GetSystemMetrics(SM_CYSCREEN),
                          NULL,NULL,hInstance,NULL);
```

The code snipped above creates a window that spans the whole screen. The returned handle can be directly used as the window parameter for the eglCreateWindowSurface function. On Windows Mobile it is also advised to disable the title and the button bar to make sure that they don't show up in front of the render target:

```
// Code to disable the title and button bar
//
SHFullScreen(GameBase::GetWindow(), SHFS_HIDESIPBUTTON | SHFS_HIDETASKBAR);
```

4.2 Window creation on Symbian

While Windows developers can choose to create simple windows or follow the MFC or ATL application frameworks, Symbian developers have no choice. Even if not put into use, a Symbian application always has to implement the model-view-controller framework. This involves implementing the CAknApplication interface, which creates the application's document (derived from CEikDocument). The document object then creates the user interface (derived from CAknAppUi), which in turn creates the application's view (derived from CCoeControl).

A call to the view's Window() method finally provides the window pointer that can be passed to the eglCreateWindowSurface() function.

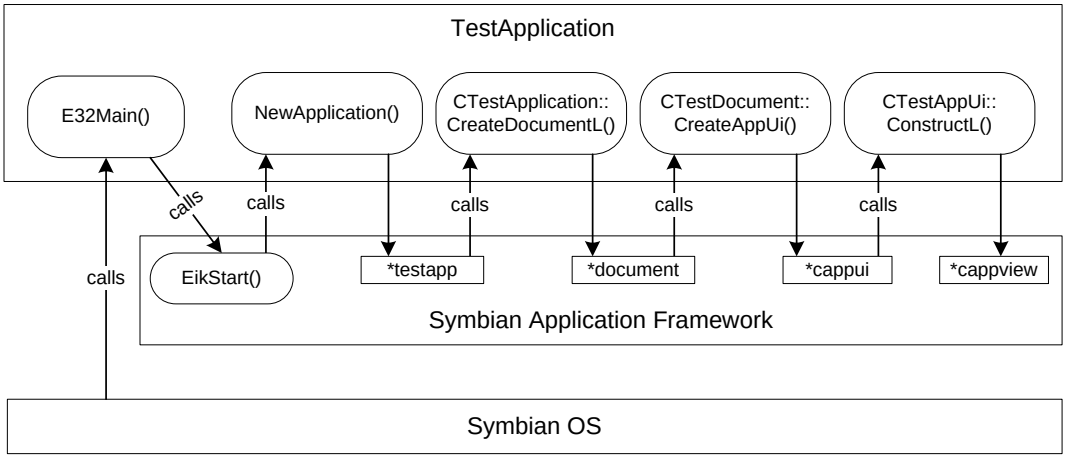

Figure 10. Sequence of initializing up a Symbian application

4.3 Initializing OpenGL ES

After the window has been created, the EGL code for setup up the OpenGL ES context is identical for both Windows and Symbian platforms:

```
// Code fragment to create the OpenGL ES rendering context.
//
const EGLint configAttribs[] =
{
       EGL_RED_SIZE,       5,
       EGL_GREEN_SIZE,     6,
       EGL_BLUE_SIZE,      5,
       EGL_ALPHA_SIZE,     EGL_DONT_CARE,
       EGL_DEPTH_SIZE,     16,
       EGL_STENCIL_SIZE,   EGL_DONT_CARE,
       EGL_SURFACE_TYPE,   EGL_WINDOW_BIT,
       EGL_SAMPLE_BUFFERS, 0,
       EGL_NONE
};

EGLint majorVersion, minorVersion, numConfigs;
EGLDisplay eglDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
eglInitialize(eglDisplay, &majorVersion, &minorVersion);
eglGetConfigs(eglDisplay, NULL, 0, &numConfigs);
eglChooseConfig(eglDisplay, configAttribs, &eglConfig, 1, &numConfigs);
EGLContext eglContext = eglCreateContext(eglDisplay, eglConfig, NULL, NULL);
EGLSurface eglWindowSurface = eglCreateWindowSurface(eglDisplay, eglConfig, window,
NULL);
```

In the code snippet above, which works on Windows as well as Symbian, the window parameter represents either a handle or a pointer to the previously created window. configAttribs is an array of type & value pairs. It defines settings such as color depth, stencil size, etc.

4.4 Initializing Direct3D Mobile

Similar to OpenGL ES, Direct3D Mobile requires filling at structure with configuration data. Other than OpenGL ES, which uses an array with key/value pairs, Direct3D introduces a specific struct of type D3DMPRESENT_PARAMETERS.

```
D3DMPRESENT_PARAMETERS d3dmpp;
memset( &d3dmpp, 0, sizeof(d3dmpp) );
d3dmpp.Windowed = TRUE;
d3dmpp.SwapEffect = D3DMSWAPEFFECT_DISCARD;
d3dmpp.BackBufferFormat = D3DMFMT_UNKNOWN;
```

```
d3dmpp.EnableAutoDepthStencil = TRUE;
d3dmpp.AutoDepthStencilFormat = D3DMFMT_D16;
pD3DM = Direct3DMobileCreate(D3DM_SDK_VERSION);
pD3DM->CreateDevice(D3DMADAPTER_DEFAULT, D3DMDEVTYPE_DEFAULT, hWnd, 0, &d3dmpp, &
pd3dmDevice);
```

The code above first fills the parameters struct with the required settings. Then Direct3D Mobile is initialized and finally a device object is created that is automatically bound to the previously created window via the hWnd parameter.

5 Graphics API Abstraction

There are two major design choices for how to abstract a graphics API, which both have their clear advantages and weaknesses. Hence, both approaches are common in practice.

A thick-layer approach provides a high-level unified rendering API to the application programmer that most often includes many advanced methods such as rendering of complete levels, sky-boxes or animated characters. Naturally such an approach strongly depends on the actual application, e.g. the type of game to develop. An advantage of such a high-level approach is that the implementation has full control over the wrapped graphics API and can therefore gain optimal performance. Furthermore such an approach can effectively hide even large differences of APIs. For example, a high-level renderer could provide exactly the same API, while wrapping a polygon rasterizer (such as Direct3D or OpenGL) or a raytracer or a volume renderer. Even though the latter two approaches might not use the concept of polygonal data at all, the high level of abstraction could easily annihilate this fundamental difference.

The alternative to the approach above is a thin-layer wrapper that tries to increase the level of abstraction over the wrapped API as little as possible. The obvious advantage is that only little code has to be written for a thin layer. At the same time, this approach only works if the various APIs to be wrapped are similar enough to fit under the common thin layer. In practice, most 3D engines put another layer on top of the thin layer, which then adds high-level features.

Although not visible at first sight, Direct3D Mobile and OpenGL ES 1.x are actually very similar. Although the former implements an object oriented API, while the latter uses a state machine concept, both APIs target exactly the same kind of feature set, which naturally led to similar results.

There are three basic design choices for implementing a thin layer approach:

- Virtual methods: This is the text book approach for object-oriented programming. A set of interfaces, specified using pure virtual methods is defined. Each implementation must implement all methods or the compiler will complain. While this is the most beautiful approach from a pure object-oriented point of view, it severely suffers from low performance if the virtual functions are called at a high frequency. The problem lies in the way virtual methods are implemented. Typically, the compiler creates a function lookup table for each class. When a method is called, the CPU first looks up the method and then calls it. Since the method's address is not known in advance this flushes the processor pipeline and hence creates a stall. The negative effect depends on the frequency at which these functions are called, as well as the length of the processor pipeline. Although graphics functions are not called that often and mobile phone processors have short pipelines, we do not recommend using this approach.
- Non-virtual, non-inline methods: Using this approach, graphics API specific functions are wrapped inside an implementation file. Classes are implemented without usage of virtual members, so there is no penalty for virtual function calls anymore. On the other hand it is not possible to formally specify an interface to be implemented. While virtual methods are resolved at runtime (when the method is called), these methods are resolved at link time. If methods are wrongly declared or not declared at all the compiler complains. If methods are declared, but implemented, the linker complains. This approach trades object-oriented design principles for faster code: Only when a function is called, the compiler can tell if the function's signature matches the specification.

- Inline functions: The usage of inline functions or methods clearly marks the fastest approach. Since the compiler is able to see the actual implementation of the wrapper when a function is called, it can often optimize the wrapper completely away. The downside of this approach is that the whole 3D engine now depends on the graphics API, although hidden from the programmer. Functions are bound at compile time and can not be switched at run-time by loading a different DLL.

Using the 2nd approach (non-virtual, non-inline methods), the easiest method is to informally define a common API that is implemented by both the OpenGL ES and the Direct3D Mobile wrapper. The wrapper can be outsourced to a separate module for static or dynamic linking (lib or dll). A choice that remains is whether to implement an object-oriented or a C-style API. Since OpenGL ES and Direct3D Mobile each use a different method, the choice seems arbitrary. Yet, considering the problems with writing to static data on certain platforms, such as Symbian, an object-oriented approach provides higher portability.

Most graphics concepts such as lights and materials are mostly the same on both OpenGL ES and Direct3D Mobile. It is therefore advised to create simple, platform independent structures for storing their configurations. The following code snippets show example implementations that can be easily interpreted by both wrappers:

```
/// Specifies properties of a light
struct Light {
	enum TYPE {
		TYPE_DIRECTIONAL,
		TYPE_POINT
	};

	TYPE	type;
	Vec3X	position;
	Vec4X	ambient, diffuse, specular;
};

/// Specifies properties of a material
struct Material
{
	Color4X	ambient, diffuse, specular, emission;
	FixedX	shininess;
};
```

Vec3X and Vec4X (as well as Color4X) are 3D and 4D vectors based on 32-bit fixed point values. FixedX represents a single 32-bit fixed point. While most data types (3D and 4D vectors) are compatible between OpenGL ES and Direct3D, special care has to be taken when it comes to handling of matrices. Both renderers use 4x4 matrices to specify transformations. Yet, Direct3D (Mobile) expects matrices in row-major order, as most text books use it. This way, the internal array stores the matrix entries row by row. OpenGL (ES) on the other hand, expects matrices in a column-major order. The render engine developer has to decide which methods to use (most developers prefer row-major) and take care to transpose matrices for the graphics API that does not follow the selected order.

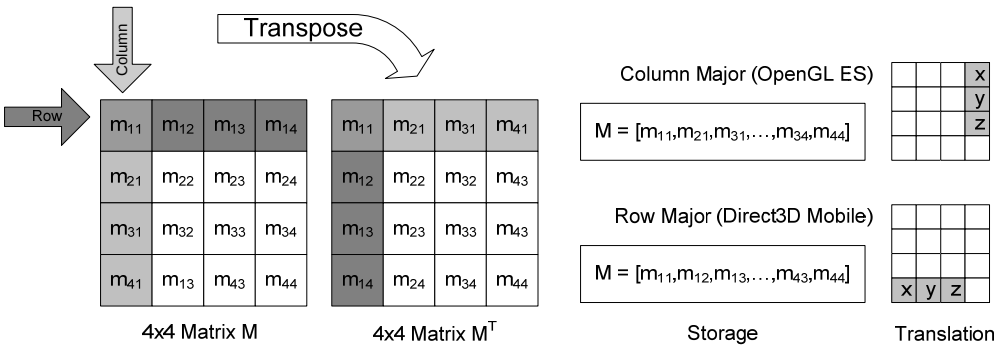

Figure 11. Column major vs. row major matrices

5.1 Texture Management

Other than for lights, materials, etc. which can be described compactly and efficiently using a common structure, textures require more work. The major difference is that OpenGL ES and Direct3D Mobile do not use the same texture coordinate system: OpenGL ES specifies the origin in the lower left corner and Direct3D Mobile in the upper left corner. Hence texture coordinates have to be adapted in one direction or the other. Mikey Wetzel gives an excellent overview on the coordinate systems differences between OpenGL and Direct3D in ShaderX6 [Wen08]. Instead of adapting texture coordinates one can also flip the texture image itself. This can easily be done during the loading of textures, but might create problems with textures of other sources such as when using PBuffers.

Textures are always stored in video memory, requiring the developer to keep a handle, either an object id or pointer that identifies the texture. This suggests the idea of putting this handle into a higher level object, a texture class that provides additional features, such as loading of textures of book keeping texture memory usage.

5.2 Geometry Management Considerations

Compared to textures, management of geometry data is much more involved and requires API specific solutions. This decision is driven by the much more data intensive nature of geometry as well as the fact that geometry is specified quite differently in OpenGL ES and Direct3D Mobile. There are three main variants, how vertex (and index) data can be specified across the two APIs:

- Direct3D vertex buffers
  Buffers are created once; updating then requires locking the buffer, writing new data and finally unlocking. Since all types of vertex data (coordinate, normals, etc.) go into a single buffer, the buffer layout, called “flexible vertex format” has to be specified and data has to be packed accordingly. For drawing the vertex (and index buffer) is bound and followed by a call to DrawPrimitive() or DrawIndexedPrimitive().
- OpenGL ES vertex arrays
  Different to both other variants all geometry data stays in system memory (application allocated) when using vertex arrays. Hence, there is no preparation step. All types of vertex data are specified as separate pointers right before rendering the object. Since strides can be specified too, data can be interleaved similar as for Direct3D, which often results in improved cache coherence. Finally, glDrawElements() (for indexed data, also requiring a pointer an index array) or glDrawArrays() (for non-indexed) rendering is called.
- OpenGL ES vertex buffers objects
  Similar to Direct3D, vertex buffers are created once and then bound before usage, such as updating or rendering. The very same render functions (glDrawElements and glDrawArrays) as for vertex arrays are called. Yet, the renderer internally uses the bound buffers rather than array pointers.

While Direct3D Mobile always requires using vertex buffers, OpenGL ES comes in two variants: OpenGL ES 1.0 does not know about vertex buffers, which was only added with version 1.1. Instead, data has to be always specified using vertex arrays, which is ineffective for hardware implementations that prefer storing vertex data in graphics memory. On the other hand, using vertex arrays requires less API calls (no buffers need to be created or updated), which can be superior for geometry that is very small or changes on a frame by frame basis. In practice though, one can expect that hardware OpenGL ES renderers support vertex arrays by implementing them on top of vertex buffers internally, which annihilates the advantage of using vertex arrays in this case too.

For optimal performance, a renderer must therefore implement support for all three variants. It is obvious that support for Direct3D Mobile vs. OpenGL ES is selected at compile time. Yet, also support for vertex buffers is compile time dependent, since OpenGL ES 1.0 implementations miss the required API functions and would therefore prevent the application from binding the OpenGL ES DLL at startup. As an alternative, the application could bind the buffer functions manually at startup and thereby dynamically detect vertex buffer support.

In the following we describe how to implement a geometry buffer class that provides a unique API for all three variants. The API of such a geometry buffer can be very simple: As a minimum it must allow specifying vertices, normals, texture coordinates and colors as well as indices. Furthermore hints for telling the renderer about the nature of the data (changes often, rarely, never) are useful so that the renderer can decide how to store the data. After all parameters have been set, the object can be rendered using a single call (not taking surface attributes into account). This means that the geometry buffer manages all data storage internally.

5.3 Geometry Rendering with Direct3D Mobile

The following code snippet shows in a simplified form (production code would include many checks) how geometry is specified and rendered in Direct3D.

```
// Direct3D Mobile – Creating and filling the vertex buffer
//
unsigned int fvfSize = getFVFSize(geometryFlags);
unsigned int dstStep = fvfSize/4;
unsigned int fvfD3Dflags = getFVFD3DFlags(geometryFlags);
pd3dmDevice ->CreateVertexBuffer(numVertices*fvfSize, 0, fvfD3Dflags, pool, &pVB));
pVB->Lock(0, 0, (void**)&pVertices, 0);
const float* src = verticesPtr;
float* dst = (float*)pVertices;

for(size_t i=0; i<numVertices; i++)
{
       dst[0] = src[0];
       dst[1] = src[1];
       dst[2] = src[2];
       dst += dstStep;
       src += 3;
}

if(geometryFlags & FLAGS_NORMALS)
{
  …
}

pVB->Unlock();

// Direct3D Mobile – Rendering
//
pd3dmDevice->SetStreamSource(0, pVB, fvfSize);
D3DMPRIMITIVETYPE                          d3dPrimType                                =
(D3DMPRIMITIVETYPE)RendererD3D::translatePrimTypeToNative(primType);
size_t numPrims = Render::getPrimCountFromType(primType, numVertices);
PD3DMDevice->DrawPrimitive(d3dPrimType, 0, numPrims);
```

The first two lines calculate the Direct3D flexible vertex format and the vertex size depending on the set of geometry attributes, such as normals, texture coordinates, etc. that will be rendered. Next a vertex buffer of correct size is created. The appropriate memory pool (video or system memory) has been determined beforehand. Finally the vertex data (only 3D coordinates here) is copied into the buffer, which then unlocked.

For rendering the object, the vertex buffer is specified as stream source. DrawPrimitive() requires passing the primitive type, which has to be translated from a platform independent value to a Direct3D parameter.

5.4 Geometry Rendering using OpenGL ES Vertex Arrays

The use of vertex arrays, requires that the respective data is stored in system memory. This is different from the Direct3D Mobile and OpenGL ES vertex buffer objects, which store data in graphics memory. This means that valid vertex arrays must be available each time the object is rendered, which requires deciding, who is responsible for maintaining the vertex arrays: The API wrapper can either

buffer the data internally (doubling the memory requirements in case the application programmer decides to also keep a copy) or it can simply demand that the application programmer keeps the vertex arrays as long as the object is rendered. While the first approach can waste memory, the latter one is dangerous if the programmer does not follow the rules.

Since no OpenGL ES buffers are required in this approach, the following code snippet only shows the rendering part.

```
// OpenGL ES Vertex Arrays – Rendering
//
GLenum mathType = RendererGL::getInternalMathTypeAsGLEnum();
GLenum glPrimType = RendererGL::translatePrimTypeToNative(primType);

glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, mathType, 0, getVertices());

if(const void* normals = getNormals())
{
       glNormalPointer(mathType, 0, normals);
       glEnableClientState(GL_NORMAL_ARRAY);
}
else
       glDisableClientState(GL_NORMAL_ARRAY);

If(const void* texCoords = getTextureCoordinate())
{
       …
}

glDrawArrays(glPrimType, 0, (GLsizei)numVertices);
```

getInternalMathTypeAsGLEnum() returns either GL_FLOAT or GL_FIXED, depending on whether the engine stores data as floating point or fixed point internally. Same as for the Direct3D case, a function that translates the platform independent primitive type to an OpenGL ES parameter is required. Then vertex data (which is expected to always be available) is specified. In case other geometry attributes (normals, etc.) are present, the respective client state is activated. Finally the 3D object is rendered using a single call.

5.5 Geometry Rendering using OpenGL ES Vertex Buffer Objects

OpenGL ES vertex buffers are very similar as in Direct3D Mobile, except that each geometry attribute is specified separately rather than using a single buffer. Each buffer is then bound and the respective data set is uploaded into graphics memory.

```
// OpenGL ES Vertex Buffers – Creating and filling the vertex buffer
//
GLenum bufferType = getBufferTypeFromHints();
glGenBuffers(getNumBuffers(), buffers);

glBindBuffer(GL_ARRAY_BUFFER, buffers[VERTEX_BUFFER]);
glBufferData(GL_ARRAY_BUFFER,    (GLsizeiptr)(numVertices*sizeof(FixedX)*3),
getVertices(), bufferType);

if(getNormals())
{
       glBindBuffer(GL_ARRAY_BUFFER, buffers[NORMAL_BUFFER]);
       glBufferData(GL_ARRAY_BUFFER,     (GLsizeiptr)(numVertices*sizeof(FixedX)*3),
getNormals(),bufferType);
}

if(getTextureCoordinates())
{
       …
}
```

```
// OpenGL ES Vertex Buffers – Rendering
//
GLenum mathType = RendererGL::getInternalMathTypeAsGLEnum();
GLenum glPrimType = RendererGL::translatePrimTypeToNative(primType);

glEnableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, buffers[VERTEX_BUFFER]);

if(Components&Render::PRIM_NORMALS)
{
	glEnableClientState(GL_NORMAL_ARRAY);
	glBindBuffer(GL_ARRAY_BUFFER, buffers[NORMAL_BUFFER]);
}
else
	glDisableClientState(GL_NORMAL_ARRAY);

if(Components&Render::PRIM_TEXCOORDS)
{
	…
}

glDrawArrays(glPrimType, 0, (GLsizei)numVertices);
```

As can be seen, rendering works very similar to OpenGL ES with vertex arrays, whereas the buffering concept is similar to Direct3D (except for the interleaving).

The API of the envisioned geometry buffer is much simpler than working directly with Direct3D Mobile or OpenGL ES. Specifying data requires only a single call per data type and rendering is performed with a single function call too. Of course this is not because OpenGL ES or Direct3D Mobile would be unreasonable complex. The price for a simplified API is paid with a reduced flexibility, which on the other hand might be perfectly reasonable for a render engine with a clearly defined purpose.

5.6 Coordinate Systems and Matrix Stacks

The final difference between the Direct3D Mobile and OpenGL ES APIs discussed here concerns coordinate systems and how to specify transformations.

OpenGL (ES) uses a right-handed coordinate system whereas Direct3D (Mobile) uses a left-handed coordinate system. Yet, this is not entirely true, since the actual renderer does not care about handedness. In the end the renderer works with 4x4 matrices and does not know about the specific meanings of the matrices in use. On the other hand, OpenGL's "high level" matrix functions such as glTranform(), glRotate(), etc. create right-handed matrices. Since there is no counterpart of such operations in Direct3D Mobile, it is advised to not use these functions anyways and instead write custom routines that stick to one handedness.

The same as for object transformation also applies to projection matrices. Creating custom projections ensures compatibility. How to calculate off-axis projection matrices, as required for AR applications has been described in chapter 2 of this article.

Another difference between OpenGL ES and Direct3D Mobile is that Direct3D Mobile uses three rather than just two matrices to define the object and camera transformation. Additional to the world and projection matrix, Direct3D Mobile knows a view matrix.

Direct3D Mobile does not include any kind of attribute or matrix stacks. Although matrix stacks are available in OpenGL ES, there are several good reasons for not using them and instead creating custom stacks for OpenGL ES too (assuming that matrix stacks are required by the renderer…):

- Portability: A render engine that runs on top of OpenGL ES and Direct3D Mobile has to implement custom matrix stacks for Direct3D Mobile anyways. Hence, the same mechanism can be used for OpenGL ES too.
- Performance: As pointed out by Atkin [Atk06], most software implementations implement all matrix operations in floating point in order to achieve full precision and numeric range as

required for full compliance. Yet, many applications do not benefit from the enhanced precision and range, but suffer from the decreased performance due to floating point usage. With custom matrix operations the developer can decide whether to use floating point or fixed point.

- Stack size: The sizes of the matrix stacks are typically very small and easily create problems with large scene-graphs. E.g. the typical limit for the projection matrix stack is 2, which is not enough when more than two cameras are present in a scene-graph.

6 Hardware vs. Software Rendering

While both hardware and software implementations of OpenGL ES and Direct3D mobile provide the same feature set, their performance characteristics are different as the OpenGL ES software implementations tend to be faster. They are typically implemented using run-time code generation to overcome the complexity problem of the pixel pipeline: OpenGL ES allows specifying around a dozen parameters that define the fragments output color. General purpose code that can cope with all combinations would either be enormously slow or immensely large. Instead advanced implementations create optimized code for each parameter combination as needed at run-time.

Besides pure software and hardware implementations, mixed approaches are common. Similar to the introduction of hardware 3D support on desktop PCs in the 1990s, some mobile designs today only implement the pixel stage in hardware, while running the vertex stage in the driver and therefore on the CPU.

Hardware rendering obviously has the advantage of providing much more raw processing power than software rendering, which usually removes the need to carefully reduce the vertex count of 3D meshes. Furthermore texturing is typically as fast as simple Gouraud shading, making texturing a free option to select. Unfortunately, these GPUs, which are primarily targeting games, are often not very fast in uploading textures, which poses a severe problem for efficient rendering of the video background in an AR application.

Pure hardware implementations are usually well balanced. Smaller, mixed designs often implement only the rasterization stage in hardware. At the lower end of performance are pure software renderers that typically suffer from bottlenecks in the pixel pipeline, whereas the mixed designs are more often vertex limited (see [Sch07]). The Intel 2700G and the nVidia Goforce 4500 GPUs are typical examples for mixed designs that execute the vertex stage on the CPU and rasterization on the GPU.

	Pure Software	Mixed S/W-H/W	Pure Hardware
Vertex Stage	Software	Software	Hardware
Pixel Stage	Software	Hardware	Hardware
Typical Limits	Pixels	Vertices	-
Framebuffer Access	Yes	No	No
Fast Texturing	No	Yes	Yes

Table 1. Comparing software, hardware and mixed implementations.

The main bottleneck of pure software renderers is usually the pixel pipeline, especially when making intensive use of texturing. As a unique advantage, these implementations allow direct frame buffer access, which enables copying the video background directly into frame buffer, thereby by-passing slow texture mapping routines. Furthermore, since the frame buffer of these designs is always in system memory, this copy operation is extremely fast. Under specific circumstances such as when rendering simple 3D graphics only, a pure software implementation can clearly outperform a built-in graphics chip.

A high level Augmented Reality toolkit is therefore required to implement multiple render paths to make optimal use of the strengths of each kind of renderer. It has to provide different routines for 2D graphics operations, such as drawing video background or 2D GUI elements. When running on hardware 3D, direct frame buffer access is usually not available and one has to rely on texture

mapping to draw bitmaps onto the screen, while in the software rendering case, these bitmaps can be directly copied into the frame buffer.

The following two subsections explain how to optimally setup for hardware (on-screen) and software (off-screen) rendering. Since we assume that Direct3D Mobile will only be used for hardware accelerated rendering, we restrict the discussion on how to setup OpenGL ES.

6.1 On-screen rendering

On-screen rendering is kind of the default and simpler case, since OpenGL ES contains comfortable EGL[4] methods for setting up the render target. All that is required is a call to the

```
EGLSurface eglCreateWindowSurface(EGLDisplay dpy,
      EGLConfig config, NativeWindowType win,
const EGLint *attrib list);
```

function. The only parameter that is different from the other eglCreate functions is NativeWindowType, which refers to a previously created window. On Windows (XP and Mobile) this type represents a standard window handle, while on Symbian it is a pointer to a RWindow object. In both cases the application developer has to create a native window beforehand, which OpenGL ES then binds its render target to.

A hardware accelerated OpenGL ES driver always specific to the device it is installed on and hence knows how to correctly bind its render target to a window, so this is guaranteed to work well. Unfortunately, this is not the case for software rendering: Due to bad display drivers on many current mobile phones, blitting the framebuffer into video memory can easily fail. OpenGL ES software implementations are typically not prepared to handle such cases, which is another reason to use off-screen rendering, when no hardware acceleration is available.

6.2 Off-screen rendering

The EGL library provides two methods of off-screen rendering. The first method uses a PBuffer, which is allocated in non-visible graphics memory. Its main purpose is to support accelerated off-screen rendering, e.g. for binding one render target as a texture to be use for rendering into a second render target. Similar to window surfaces, PBuffers do not provide direct access. Because of that we don't go into any more detail on PBuffers.

A PixMap represents a frame buffer that is always located in system memory, which means that it allows direct access to the stored pixel data. At the same time, this also means that hardware accelerated OpenGL implementations do not support PixMap surfaces since they can only render into video memory.

Creating a PixMap render target works similar to creating a window surface. Instead of creating a window beforehand, the application developer simply creates a native bitmap using OS functions, which is then passed to

```
EGLSurface eglCreatePixmapSurface(EGLDisplay dpy,
EGLConfig config, NativePixmapType pixmap, const EGLint *attrib list);
```

either as a handle (Windows XP or Mobile) or pointer (Symbian). Since the bitmap was created manually, one can use OS specific functions to gain access to the internal pixel data, e.g. for blitting a full-screen video background or drawing 2D GUI elements.

The main difference between PixMap and window surfaces lies in the fact that OpenGL ES does not feel responsible for presenting the PixMap render target on the screen. While one can simply call eglSwapBuffers() for the window surface, drawing of PixMaps has to be done manually. On the other hand, this allows more advanced methods of bit blitting, such as using specialized libraries (e.g. PocketHAL[5]) that support various methods of frame buffer blitting (DirectDraw, GDI, Gapi, raw

[4] http://www.khronos.org/egl/

[5] http://www.droneship.com

video access or even device specific methods) to gain optimal performance as well as compliance. For this reason alone, PixMaps present the preferred method when using software rendering.

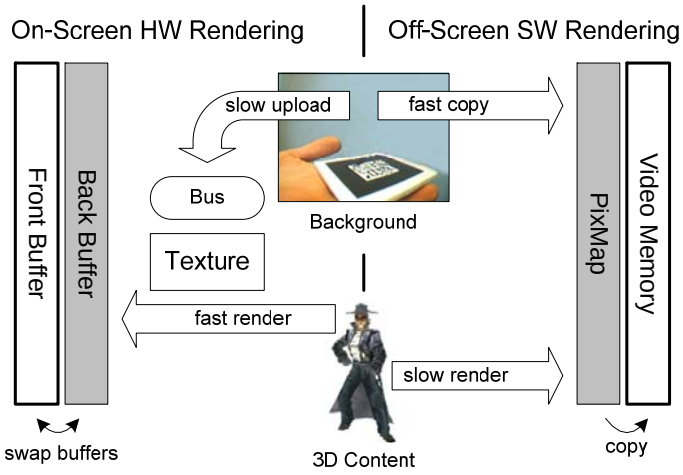

Figure 12. Software vs. hardware rendering in an AR application

7 Scene-graph Rendering

Although not so common in games, scene-graphs are omnipresent in graphics research. Their clear strength is flexibility. A scene-graph describes a hierarchical relationship between objects, which naturally models the real world: Most objects in close proximity are not somehow related to each other. Hence, objects can be grouped and parent-child relationships can be defined in a scene-graph. As an example, a hand can be modeled as the child object of the arm, which again is the child object of the character's body. When the body moves, both hand and arm move along. When the arm is lifted, the hand naturally shall be lifted too. The relationships are executed by traversing the scene-graph, and applying transformations of an object to all its children too.

Scene-graphs are typically implemented in object oriented languages, such as C++. Two concepts are highly important for scene-graphs: Reflection and fields. Reflection describes the ability of an object to report about its own state and structure. In most scene-graphs every node reports its own type as well as all its attributes, typically called fields. This mechanism allows high level concepts, such as modifying a node's attributes without knowing the node's type itself. This is important to make scene-graphs extendable: When a developer implements a new node type, other nodes can interact with the new node, even though the new node was not known at compile time of the other nodes.

Reflection is important to implement the concept of fields. Rather than just simple data containers, fields are active objects that detect when their state is changed and can react accordingly. E.g. a field might forward its state to another field. These "field connections" allow data flow other than just along parent-child relation ships: Data can flow across or even into or out of the graph. Augmented Reality applications typically use this concept to forward tracking information (the camera pose) directly into the related transform nodes. After the corresponding field connections have been set up, all parties communicate autonomously.

In C++, scene-graph reflection is typically implemented using class members: static member variables contain class specific information. Unfortunately writing to static data is not allowed on most Symbian devices. Even though newer OS versions allow this, it is still recommended to not use it. Instead a single instance, a scene-graph database, implemented as a singleton is recommended. At startup, each node registers its type and all reflection data at the database. A simple trick solves the problem of defining unique node type ids: A pointer to a static class member (e.g. the node type name) is unique – even across multiple modules (DLLs). Hence, this also allows introducing of new node types at runtime. While the type name of the node also provides a unique identifier, string comparisons are slow and therefore not recommended.

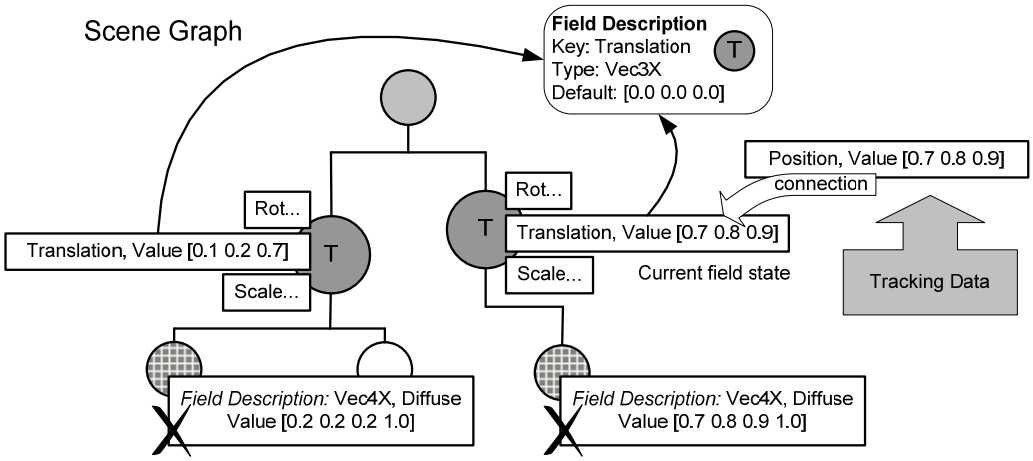

Figure 13. Field description sharing in a scene-graph

Not only scene-graph nodes, but also their fields require reflection. Since a single node typically contains several fields, their overall number quickly sums up. Hence, it is not appropriate that each field stores a complete copy of its reflection information (such as name, type, default value, etc.) since this would be an enormous waste of memory. E.g. a single integer field requires only 4 bytes of real data, but easily 10 times as much for reflection data. Instead it is preferred to share type information among all fields of the same owner type. For example, all translation fields of transformation nodes can share a pointer to the very same field descriptor object that stores the data about all translation fields belonging to a transform node. Especially for large graphs, which use the same node types again and again, this can save a lot of memory.

8 Video and Image Processing

What makes graphics for Augmented Reality applications different to typical game related graphics is the strong need for a live drawing camera feed as video background, which is then usually augmented with 3D renderings. Even more, the video feed is most often analyzed with computer vision techniques. This dual purpose puts different requirements on the video images; while computer vision algorithms are typically performed on grayscale images, rendering requires images in RGB format. On many phones, the video feed is delivered in neither of these formats, hence requiring conversion.

In AR applications images from the built-in phone camera have to be displayed on the screen and analyzed for pose tracking (determining the devices position in space) at a frame by frame basis. Since image data is large considering the low memory bandwidth and computational power of a mobile phone, only highly optimized approaches lead to satisfying results. Most cameras built into phones today already include simple processing units that allow outsourcing some of the processing tasks.

For example, most cameras in modern mobile phones perform lens undistortion internally. Every lens produces distortions to some extend. Typically, the lower the quality of the lens and the wider the field of view, the stronger radial distortions become. For accurate pose estimation results, these distortions have the compensated, which can be computationally expensive. Yet, since many camera phones already perform this step to some degree, the CPU is freed from doing it.

Furthermore, application programmers often have the choice between various video resolutions and sometimes even different pixel formats. This requires the programmer to choose which combination of parameters yields optimal results. To make things even more complicated the choice is also influenced by the rendering system to find the video mode that best fits the screen's pixel format and resolution. Hence it is ideal to perform a few benchmarks during the first application start-up.

	Bits per pixel	**Suitability for tracking**	**Suitability for rendering**	**Conversion to grey scale**	**Conversion to RGB**
YUV	12 or 16	very good	not possible	very fast	slow
RGB565	16	bad	good	slow	-

RGB32	32	average	good	medium	-

Table 2. Suitability and easiness of format conversion of common camera pixel formats.

The most common pixel formats delivered by today's mobile phone cameras are: YUV (either 12 or 16 bits), RGB565 and RGB32. The last one is most well known on PCs. It stores the red, green and blue channels interleaved in a double word, optionally using the fourth byte for storing an alpha channel. Since RGB32 requires 4 bytes per pixel it is rather uncommon on mobile phones, which are scarce in memory size and bandwidth. The similar RGB24 format is uncommon on mobile phones, since an RGB24 pixel can not be directly read into a single register, because ARM CPUs do not allow unaligned memory accesses. Instead 3 byte accesses plus merging would be required which would tremendously slow down processing speed.

RGB565 is the dominant pixel format on mobile phones. It stores all three color channels in a single word (2 bytes, storing green at higher bit rate) thereby halving the memory usage compared to RGB32. YUV is insofar different from the aforementioned pixel formats in that it is not based on the RGB color model, but uses a brightness (Y) and two chrominance (UV) channels instead. Since the human eye is more sensitive to brightness than colors, this allows saving bandwidth by storing the Y channel at higher data rates than U and V without loosing much quality. YUV appears in two main variants: YUV12, which stores pixels in three separate planes (Y at full resolution plus U and V at half resolution) and YUV16, which stores pixels interleaved in a YUYVYUYV… format.

Table 2 gives an overview of the strengths and weaknesses of the three aforementioned pixel formats. YUV can store images with a little as 12 bits pixel. Since U and V are available at lower resolutions they must be upsampled for displaying a color image. The format is highly suitable for computer vision (pose tracking) since the grayscale channel is directly accessible at full resolution. On the other hand, the format is unsuitable for rendering, since today's renderers use RGB formats mostly. RGB565 and RGB32 are both directly suitable for rendering. It depends on the actual renderer implementation, which of both formats is optimally supported, while the other one is typically converted internally. RGB32 is suitable for computer vision tasks since conversion to grayscale is fast by summing up (either with or without weighting) the three channels. RGB565 requires first extracting the three color channels from the 16-bit word before they can be summed up. In practice this is often speeded up using a lookup table.

9 Fixed Point vs. Floating Point

Any computationally intensive mobile phone application has to carefully select when to use floating point. Although some high-end mobile phones today include hardware floating point units (FPUs), most deployed devices still have to emulate these operations in software. Instead developers often use fixed point, which is based on native integer math extended a pretended fractional part. In general the following observation for execution time holds for math operations on mobile phones:

$$\text{32-bit fixed point} < \text{H/W floating point} < \text{64-bit fixed point} \ll \text{S/W floating point}$$

Although the performance difference is small, fixed-point math using 32-bit integers is generally the fastest method, even if a hardware floating point unit is available. This is because modern ARM CPUs can do most integer operations in a single clock circle. Even more due to the long tradition in using fixed point math, these CPUs can do bit-shifting for free on most operations. As an effect, fixed point is typically as fast as simple integer math. Yet, some modern ARM CPUs have vector floating point (VFP) units that can run multiple operations in parallel. If code can be parallelized, this can yield a considerable speedup. In practice manually created assembler code is required for good results.

In many cases 32-bit fixed point is not accurate enough or does not provide enough numeric range. Since software emulated floating point is much slower (typically 30-40 times slower than 32-bit fixed point), a good compromise is using 64-bit fixed point. Yet, since today's mobile phones use 32-bit CPUs, fixed point based on 64-bit integers is generally 3-4 times slower.

Yet, even if hardware floating point units are available, specific operations such as square root and sine/cosine are sometimes still emulated and hence execute extremely slow. If code is intensive on these operations it can therefore be faster to use 64-bit fixed point even though an FPU is available.

10 Conclusion

This article covered a broad range of techniques that are important for creating games and other graphics intensive applications that shall run on a large number of currently deployed phones. Special focus has been put on the specific requirements of Augmented Reality systems.

References

[Atk06] Phil Atkin, High-performance 3D graphics for handheld devices, http://www.khronos.org/developers/library/kmaf_cambridge_2006/High-performance-3D-for-handhelds_NVIDIA.pdf

[Bree96] D. E. Breen, R. T. Whitaker, E. Rose, and M. Tuceryan. Interactive Occlusion and Automatic Object Placement for Augmented Reality. Computer Graphics Forum, 1996, 15(3), pp. 11-22

[Möl08] Thomas Akenine-Möller, Jacob Ström: 'Graphics Processing Units for Handhelds', Proceedings of the IEEE, 2008, 96 (5), pp. 779-789

[Pul05] Kari Pulli, Tomi Aarnio, Kimmo Roimela, Jani Vaarala: 'Designing Graphics Programming Interfaces for Mobile Devices', Computer Graphics and Applications, 2005, 25 (6), pp. 66-75

[Sch07] Dieter Schmalstieg, Daniel Wagner, Experiences with Handheld Augmented Reality, Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR 2007), pp. 3-15

[Wag07] Daniel Wagner, PhD Thesis, http://studierstube.org/thesis/Wagner_PhDthesis_final.pdf

[Wen08] Mikey Wentzel, "Shaders Gone Mobile: Porting from Direct3D 9.0 to OpenGL ES 2.0", ShaderX6, Charles River Media, pp. 415-434, 2008