Moving Mobile Graphics

Cramming Software onto Mobile GPUs

Lecturer
Andrew Garrard
Samsung R&D Institute UK

SIGGRAPH
August 2015

Keywords: GPU, Mobile Device, OpenGL ES

Abstract

Mobile systems are increasing in graphical capability despite the constraints of embedded platforms. Making the most of this performance requires careful consideration, and different techniques than may be familiar to desktop developers. This course highlights the differences of embedded graphics systems and ways to leverage them.

Course Summary

Desktop graphics cards have achieved very high performance levels, but at the cost of size, power consumption and heat. These issues mean that embedded GPUs must take a very different approach to achieve the performance that is expected of a computing device by the modern consumer. The design decisions made by embedded GPU designers influence the best practices for the mobile graphics developer. For example, tile-based renderers have high local frame-buffer bandwidth, at the cost of suffering performance degradations as geometry counts increase. The modern mobile device contains a very high-quality screen attached to powerful but efficient rendering hardware; the onus is on the developer to make the most of the display while keeping the efficiency of the system within the constraints imposed by battery life and heat dissipation.

By considering the bandwidth requirements of common rendering operations, the flow of data through the GPU, computational limits of embedded devices and the optimisations typically performed by mobile rendering architectures, this course discusses the pitfalls of following a conventional desktop rendering work-flow, and offers suggestions for optimising rendering efficiency across multiple mobile platforms. The application developer should offer the best user experience, targeting the GPU's considerable resources where they will enhance the final result in real-world situations. By examining the restrictions on performance imposed by current mobile rendering solutions and comparing with the desktop graphics industry, it is possible to discuss developments in mobile graphics and the impact of potential hardware and API changes with a view to future-proofing applications.

About the lecturer

Andrew Garrard
a.garrard@samsung.com
Samsung R&D Institute UK
http://www.samsung.com
Communications House
South Street
Staines-upon-Thames
Surrey
United Kingdom

Andrew Garrard is a Senior Software Engineer at Samsung R&D Institute in the United Kingdom (SRUK). He works in the Graphics Innovation Lab, with a focus on embedded graphics, compute and machine vision. His professional history includes four years working in hardware ray tracing, three years working on workstation graphics drivers and five years writing assembly code for embedded graphics hardware. He has been programming in computer graphics since getting a Sinclair Spectrum for Christmas in 1982, although he did not make it to SIGGRAPH until 1999.

He is one of the chapter leaders of the Khronos UK Chapter, involving presentations and introductions to Khronos standards — most notably OpenVX, for which he is the SRUK representative and an active participant in that fledgling computer vision standards group.

He presented a talk entitled "Moving to mobile graphics and GPGPU: forget everything you know" at SIGGRAPH 2013 and a course (on which these notes are based) entitled "Moving Pictures: Making the Most of the Mobile" at SIGGRAPH 2014.

Contents

1. *Why are mobile GPUs different?*

1.1 Desktop graphics made mobile

Processing power in computers has grown exponentially over time, but these advances have been far outstripped by the capabilities of computer graphics hardware. What took days of rendering time for a feature film in the 1990s was interactive a decade later. Affordable modern GPUs offer teraflops of processing power and hundreds of gigabytes per second of bandwidth. Desktop GUI displays have grown from the common 14" VGA-and-below to 20"-and-above screens with HD resolutions. Consumers are accustomed to affordable photo-realistic graphics.

To achieve this performance, desktop graphics cards use supercomputer techniques. Large, compute-heavy chips mean a lot of heat dissipation. Keeping the GPU fed requires fast, dedicated memory. A modern graphics card is dense with memory devices, capped with a large heat sink and fan, and powered directly. The fastest devices barely fit in the average tower case and place high demands on airflow and power supply design.

The mobile revolution has raised opposite demands. Consumers want, above all, a device that will fit in a pocket (or bag, for a tablet). The form-factor of desktop — or even laptop — computers is not an option. Things have to be done differently.

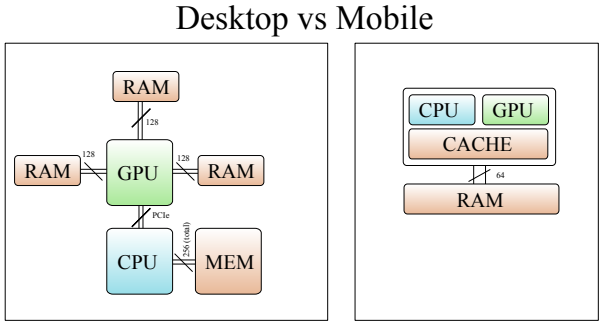

Figure 1.1: Desktop GPUs have dedicated memory with fast access. Mobile GPUs generally have a single, small, memory bus shared between the CPU and GPU.

1.2 Cutting it down

The first mobile devices had no graphical capability at all, and early monochrome LCDs required no graphics hardware to speak of. Yet the demands of a visual age mean that consumers have not been happy with such devices. Increasing capabilities and the demands of interactivity made colour displays and, later, hardware acceleration necessary. It is tempting to think that the only aspect to making a mobile graphical system is to cut down a desktop device until it fits.

Reality is not so simple. Cutting a 24" HD desktop monitor down to fit a 4" phone leads to a display with only 320×180 pixels. It is certainly possible for a simple GPU to handle a display of this resolution. With low resolution, and typically low bit depths, bandwidth demands are small. The small frame buffer might fit onto a single, affordable SoC.

Unfortunately for mobile hardware designers, things are not so simple. While acceptable for a microcomputer of the early 1980s, a display quality this low is not sufficient for modern uses. Fun though Snake is to play, the mobile internet and the advance of the app store mean that content consumers have higher demands. A display that has thirty-six times less area and thirty-six times fewer pixels than a desktop system cannot hope to be as useful.

1.3 Upping resolution

As content-consumption devices, mobile screens have grown to match or exceed the resolution of desktop systems. The requirement to match the written page has pushed tablet resolutions, even where the larger screen size of a tablet has made the pixel pitch less of a requirement. The latest high-end cell phones typically support 1920×1080 screens or more, with multiple tablets supporting 2560×1600 resolution.

High resolutions — especially in the bit depths necessary for high quality image browsing and video playback — require a larger frame buffer than can practically fit on a single IC. Increasing resolutions demand larger textures, more geometry and more compute power. The small form factor of mobile devices makes it tempting to think of them as having lower demands than their desktop predecessors. In reality, even the power savings of subsequent silicon processes have not been sufficient to bring mobile performance up to the required levels. To achieve effective interactive rates with good graphical quality, mobile GPUs must take different approaches from their desktop cousins.

1.4 Limiting size and power

The demands on a modern GPU necessarily require a large power draw, no matter how clever the architecture of the GPU. Having this power available in a mobile form factor necessitates a large battery. Particularly when the device must fit in the hand or pocket, maximising the battery area means that the circuitry must be as small as possible. Where

the area of a desktop graphics card supports distributing the heat produced by the core and by external RAM chips, no such option is available in mobile.

While it is possible to attach a rudimentary heat sink to mobile devices, active air flow is not currently considered practical, so the density of circuitry provides its own challenges. The shape of mobile devices is at least amenable to offering a large surface area for its volume, but a screen is hardly an ideal conductor of heat. It is common for users to cover mobile devices with protective cases, which impact on cooling, and at best the device must function when insulated by resting on furniture. Since the front of the device is the only portion that is guaranteed to be exposed, it must be responsible for most of the heat dissipation, yet since users interact with these devices via touch screens, the device may not grow so warm as to be uncomfortable to hold.

Mobile devices must therefore incorporate a thermal throttling mechanism. As the device heats up, the clock speed of the GPU and CPU are reduced to moderate the current. Where a desktop system can simply increase the fan speed, mobile devices cannot sustain high power demands for an extended period of time. Even when the desired performance can be matched in testing, it is therefore necessary for the mobile developer to monitor the system's efficiency and ensure that performance will not drop over time.

1.5 Bandwidth vs processing

Space and cost requirements typically necessitate that off-chip RAM be shared with the CPU — almost all mobile devices have a unified memory architecture. A single bus is more efficient in both space and power. However, space and power requirements also demand that such a bus should be only as large as necessary for the required performance. Any system must be designed for an assumed work-load; even if the GPU work load is under control, the CPU's demands on the memory bus need to be considered.

Modern integrated circuits are so efficient that driving even small off-chip circuitry can require much more power than work done within the chip. On-chip caches and efficient ordering of memory accesses are necessary to keep bandwidth demands as low as possible. Even so, the onus is on the application author to be sympathetic to this design, and to code in such a way as to maximise efficiency.

2. *Tiling*

2.1 Reducing frame buffer bandwidth

Many — although not all — mobile GPUs make use of a technique known as *tile-based rendering*.

The conventional *immediate-mode* rasterising process involves processing each rendering primitive in turn, updating any pixels that are covered by the view of the primitive. The vertices of each triangle are conceptually transformed into screen coordinates, then all the pixels which fall within this triangle are drawn onto the frame buffer. This approach means that geometry can be processed simply, in the order in which it is submitted.

However, because there need not be any spatial relationship between consecutive triangles, the immediate-mode rendering approach can require multiple locations within the frame buffer to be accessed repeatedly and in an arbitrary order. In a mobile GPU, accessing an off-chip frame buffer in this way puts demands on the limited bus bandwidth and consumes power.

We can compare this approach with a conventional ray caster (the first, eye ray, pass of a ray tracer). In a ray-casting system, each pixel is considered in turn, and the ray passing through that pixel is tested against all of the geometry in the scene to see what primitive is nearest to the eye — typically this is done with the assistance of some form of spatial acceleration structure. While ray casting requires no dependency between the rays passing through the scene, this independence means that no advantage can be taken of coherency between adjacent rays: calculations are not shared between multiple rays intersecting the same geometry. Ray casting in this way increases the computational cost of rendering and requires a large amount of memory traffic to traverse the scene representation for every pixel.

A ray caster can be accelerated by treating rays as bundles and amortising work across all the rays in the bundle. A tiled rasteriser is a very similar solution: the screen is divided into rectangular tiles which are independently rasterised. This approach requires the geometry contained within the scene to be processed repeatedly, but — unlike a simple ray caster — once per tile rather than once per pixel. The approach is equivalent to iterating over the screen area one tile at a time, setting the viewport to the tile area, and rendering the scene — except that the tiled rendering is automated.

Rendering one tile at a time means that the frame buffer for a tile can be rendered entirely on-chip in cache or dedicated RAM. This ensures that each pixel needs to be

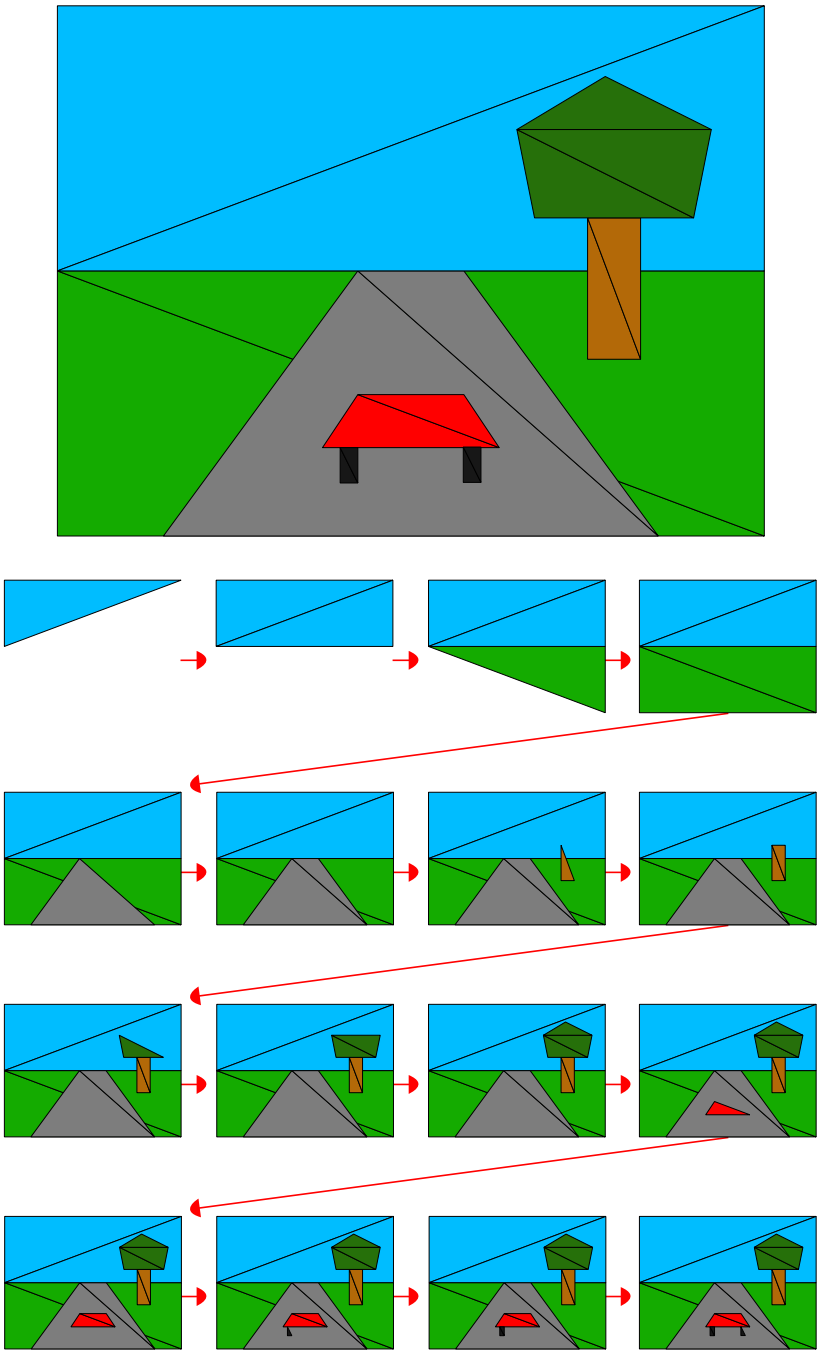

Figure 2.1: Immediate rendering order: Each triangle is processed in turn, building up the image one triangle at a time.

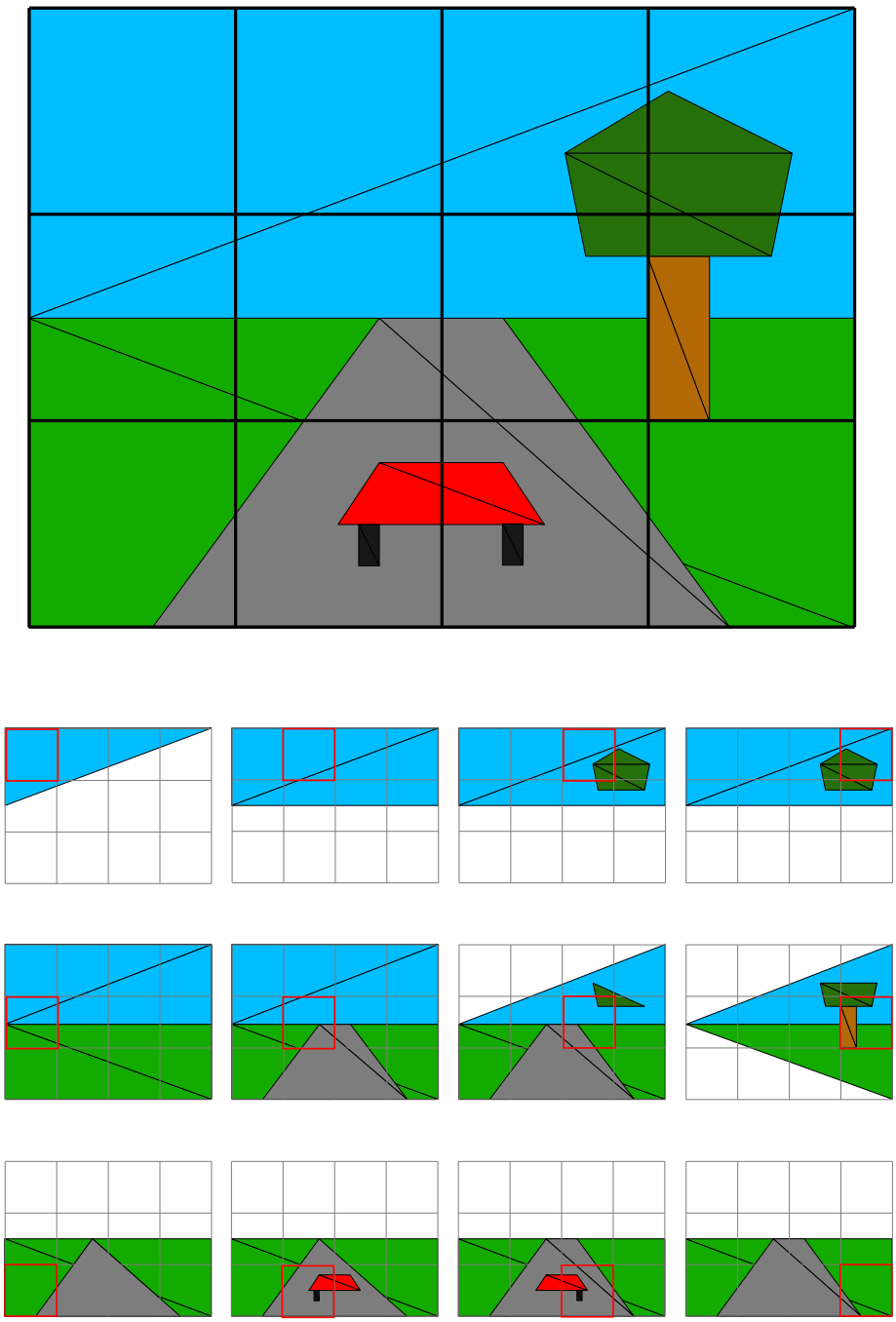

Figure 2.2: Tiled rendering: The image is broken into tiles which are rendered independently. Only the primitives which intersect a tile are examined when the tile is rendered.

written to external memory only once. If the overhead of accessing the geometry is lower than the overhead of accessing the frame buffer, total bandwidth requirements are reduced.

2.2 There's more than one way to do it

Each vendor has its own take on tiling. Typically the graphics processing begins with a vertex shading pass. As in a conventional graphics pipeline, the vertex shading pass determines the on-screen location of graphics primitives. In a tiled renderer, this pass makes it possible to determine which tiles are associated with each primitive — or, equivalently, which geometry is associated with any tile. Once all the geometry has been processed (and, therefore, once all the primitives that may be associated with any tile has been determined) each tile is processed in turn. Because the geometry has been classified by the first pass, primitives that are not included in any given tile are not considered when the tile is processed.

The exact mechanism used to implement the tiling pass varies by device, according to the designers' decisions about how best to balance the overhead of classifying the geometry and the time taken to render each tile. The form of the processed primitives varies, as does the size of a tile and whether multiple tiles can be processed in parallel. No matter the implementation, there is always an overhead to increasing the amount of geometry compared with using an immediate-mode renderer. The difficulty for an application author is to achieve the desired image quality without incurring an undue geometric overhead, particularly if the application must extract the best performance from multiple GPUs.

2.3 Discarding the unnecessary

It is common to begin the rendering process for a frame by clearing the frame buffer, Z and stencil buffers. One advantage of tile-based rendering is that, if the depth and stencil buffers are not needed for the next frame, there is no need to write this data out to external memory: it can be discarded as soon as the tile has been rendered, and need not be reloaded before the tile is rendered again in the next frame. This can save a useful amount of memory bandwidth, but requires the API to be told that these buffers will not be needed again. For example, in OpenGL ES, the EXT_discard_framebuffer extension can be used to inform the API that on-chip data need not be transferred to main memory for later use, and retrieved in the next frame.

Tiled renderers have a particular advantage in anti-aliasing. Super-sampling in a tile-based renderer can be implemented by rendering the tile at a higher nominal resolution and performing a down-sampling step in the process of writing a larger on-chip tile to a smaller in-memory representation of a frame buffer. This does not make super-sampling free: the shading — and especially texture access — cost of rendering at a higher internal resolution can still be significant, meaning that multi-sampling can still be worthwhile. In either case, tiled renderers avoid the need for additional data to be stored off-chip,

typically giving anti-aliased rendering a lower overhead than a simple immediate-mode renderer could achieve.

2.4 Overdraw and deferred shading

Because the frame buffer — including the Z buffer — can reside on-chip in a tile-based renderer, the overhead of overdraw can be lower than for an immediate-mode renderer. When shading is expensive, this means that making use of deferred shading techniques can be valuable. The conventional implementation of this approach involves first rendering the geometry in the scene, writing only to a depth buffer, then performing a second pass with fragment shaders enabled. In the second pass, any complex fragment shaders colouring pixels that would have been overdrawn will be excluded by being behind the primed Z buffer. The cost of writing repeatedly to the Z buffer on a tile-based renderer is lower than would be the case if the buffer needed to be written to memory.

Countering this advantage is the cost of processing more geometry. Some hardware is capable of performing deferred shading as part of its internal process, avoiding unnecessary overdraw. For these systems, performing an explicit deferred shading pass is actually counter-productive, and geometry should ideally be submitted in an order to minimise the number of state changes, limiting the number of API calls. On other architectures, the choice of deferred rendering is determined in part by the shading cost and whether it is possible to minimise overdraw in some other way — for example by attempting to submit geometry in front-to-back order. The evaluation order is, of course, complicated by transparent surfaces.

3. *Keeping things moving*

3.1 Deferred rendering

The rendering process with a tile-based renderer involves an initial classification pass, which must have processed all the geometry submitted to a scene before rasterising can begin. This inherently turns rendering into a two-stage process, whereas an immediate-mode rasteriser can begin fragment shading as soon as a primitive's vertices have been processed. This latency is the cost of increasing throughput.

The extent of this overhead varies. On some devices, vertex shading (binning) and fragment shading (tile rendering) follow immediately. In others implementations, the two passes are explicitly split between frames so that the different demands of vertex and fragment shaders can be balanced. That is, as geometry is submitted, vertex shading is performed, classifying the geometry by the tiles it covers; while this is progressing, the tiles of the previous frame are being rasterised. The result is an increase in throughput at the cost of an increase in latency, similar to the "alternate frame rendering" modes of NVIDIA's SLI and AMD's CrossFireX. Developers should be aware of this difference between frame rate and latency when considering latency-sensitive applications.

3.2 Reading and writing too soon

The cost of any system which is dependent on exchanging throughput for latency is that any attempt to minimise latency will have an effect on throughput. Any approach which requires rasterising part of a frame's data in order to render the rest of the frame is likely to introduce a bubble into the pipeline. Where possible, removing dependencies from the pipeline can avoid a potential performance loss on those platforms which split the rendering across multiple frames.

Some operations require the CPU to wait for rasterising to complete before rendering can continue. In OpenGL ES, glReadPixels is the obvious example of this, but the same applies to occlusion queries. Where possible, approximating an occlusion query by using the result of a previous frame's rendering can be an approach which reduces overheads.

More insidiously, the deferred fragment shading pass means that texture access may not complete as quickly as expected. Implementations must be aware of the dependencies

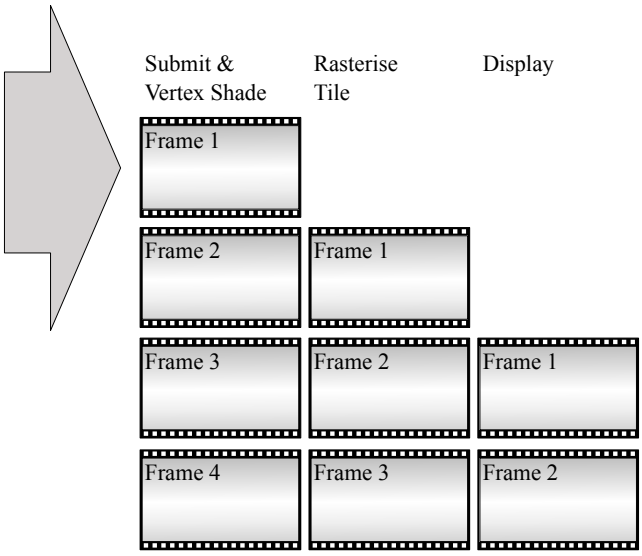

Figure 3.1: Deferred rendering: While the application is submitting one frame for vertex shading, the previous frame is being fragment-shaded. The frame before that is on the display!

in rendering order imposed by environment mapping, for example. However, when texture updates originate from the GPU, latency can be more costly to hide. Typically multiple copies of textures need to exist to support this case, potentially incurring a copy between textures during each frame. Manually retaining multiple copies of a texture and alternating which one is accessed can help here.

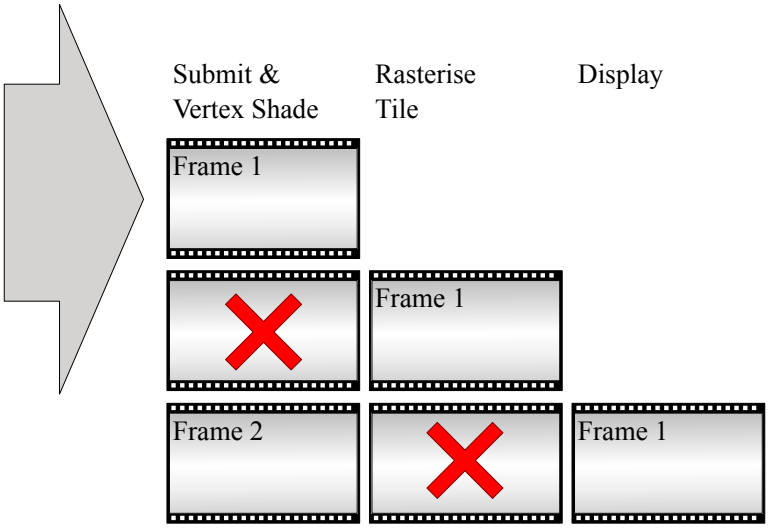

Figure 3.2: Blocking rendering: If the rendering of frame 2 depends on frame 1 being fully rasterised, the system must wait for the rasterisation pass to complete, losing parallelism.

4. *Geometry limits*

4.1 The perils of polygon proliferation

Any regular spatial partitioning scheme provides the chance for a triangle to be partially contained within more than one tile. When more than one tile holds the primitive, the triangle's representation needs to be processed for each tile in which it is contained. If the triangle data is stored for each tile that contains it, the representation must be written repeatedly as well as read; if the triangle representations are referred to in a master list, there is an overhead both to reading the triangles which are contained in the tile and to skipping over those triangles that are not contained in the tile. 2D applications can try to manage this by aligning geometry so that it does not unnecessarily cross tile boundaries. (The tile size varies by architecture, but several use powers of two.) Hierarchical tiling schemes can be very efficient, but suffer when a lot of geometry crosses a high-level tile boundary; this should be rare, but uneven performance can result when a large amount of overdraw hits such a boundary — level of detail control can help to limit this.

While increasing the geometry count carries an inherent cost in any graphics system, this cost is magnified in a tiled renderer because of the need to consider polygons repeatedly. The more tiles are used to render the scene, the higher this relative overhead will be; therefore tiled renderers see a higher geometry load on larger screen sizes — although, hopefully, a lower frame buffer transfer load than an immediate-mode render would achieve. In some architectures, enabling higher super- or multi-sampling rates can also reduce the tile size: the size of the on-chip frame buffer is limited, and — while only the reduced representation gets written to the off-chip frame buffer — the number of tile passes required to cover the image is limited by the full pixel size before the oversampling is resolved to final screen resolution, producing a higher cost than might have been expected simply by allowing for the additional fill and shader rates.

This disadvantage is exacerbated because most schemes for binning triangles according to tile are conservative — for example, testing only the screen-space bounding box of the triangles — thereby including a detectable fraction of triangles that are not, in fact, contained within the tile. This can become especially evident with long, thin triangles: not only are these likely to cover multiple tiles, they may be incorrectly associated with many more tiles than needed. Applications may be able to benefit by arranging tessellation to be axis-aligned, if this can be done without greatly increasing the polygon count: long, thin triangles are less costly if wide or tall rather than both.

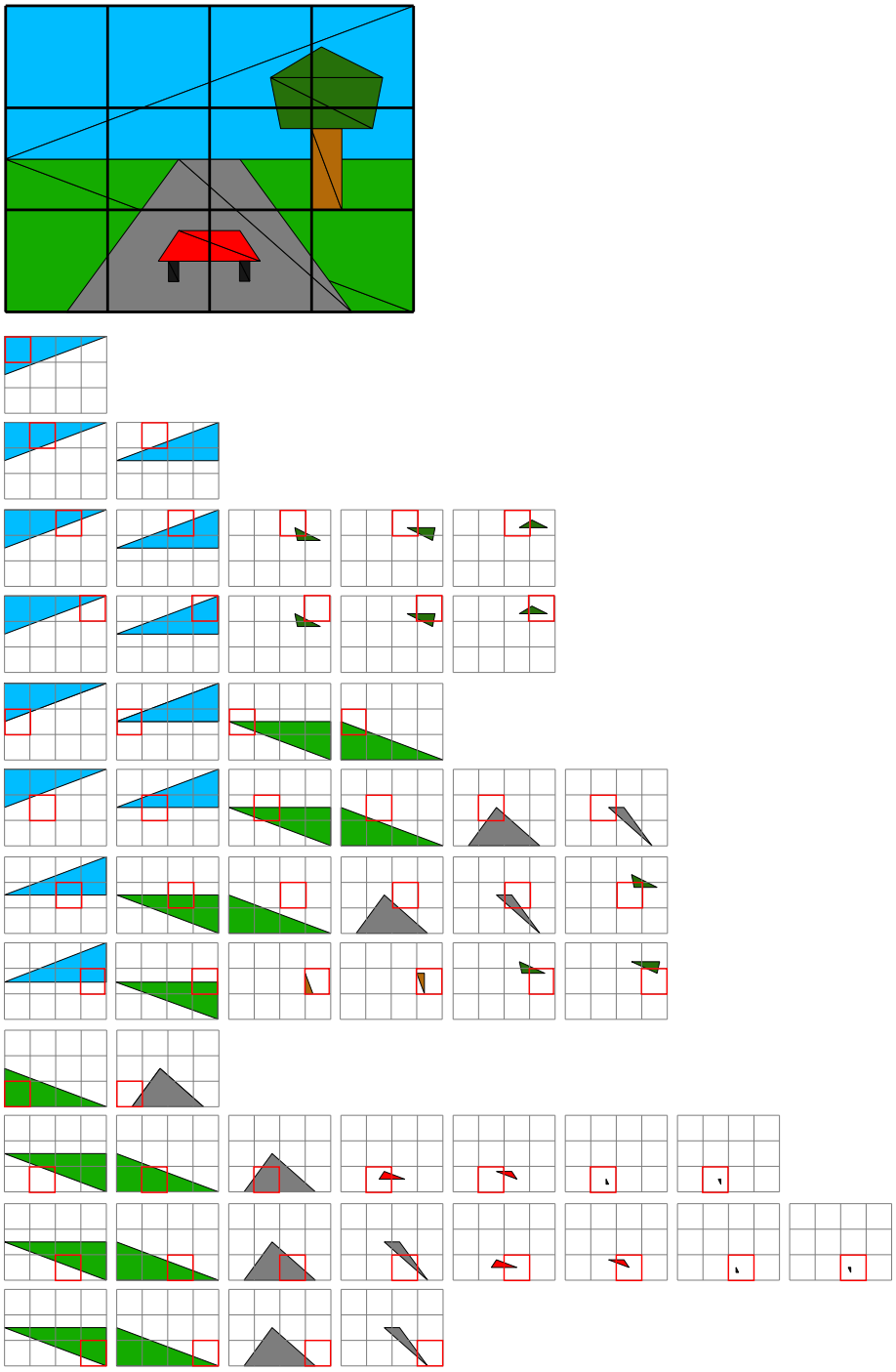

Figure 4.1: Polygon lists for each tile — note how often polygons are repeated

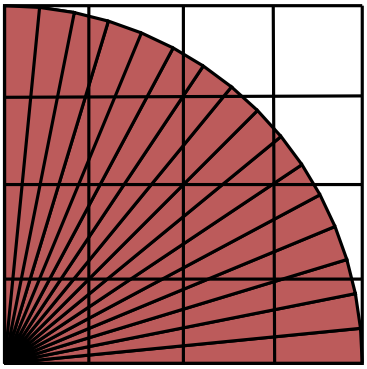

Figure 4.2: A triangle fan in a tiled renderer

4	4	3	0
5	5	4	3
8	8	5	4
16	8	5	4

Figure 4.3: Triangles intersecting each tile (total: 86)

8	6	3	0
11	9	6	3
14	12	9	6
16	14	11	8

Figure 4.4: Triangle bounding boxes intersecting each tile (total: 136)

Because tiled rasterisers must keep a record of the geometry to be rendered — sometimes replicated, depending on the representation — there is a necessary upper limit on the number of primitives that can be processed before the implementation runs out of space. To support more complex scenes, the implementation must render the geometry currently available to it when its buffer are full, then resume with any additional geometry that the application wishes to submit. Doing this typically involves the temporary storage of much more state than is generally necessary at the end of a frame — for example, storing the Z buffer or multi-sample buffers, which may not need to persist between frames, but are needed to complete the rendering of a single frame. As a result, running out of space for geometry storage — while it should not happen during a well-optimised application — can show a heavy performance hit.

Within this limit, geometry-heavy effects such as particle systems can actually not be as inefficient as one might expect on a tiled renderer. The reason for this is that very small geometry often fits mostly within one tile. On some, but not all, hardware it can be beneficial to try to retain some spatial coherency between consecutive polygons even in a particle system, to ensure that the irrelevant geometry can be skipped efficiently. For example, some implementations store a compressed mask associating the primitive list with each tile. The compression allows large chunks of the geometry list to be skipped efficiently by tiles that are not affected by them — but if consecutive triangles are distributed throughout the scene, this compression is less effective and tiles will often touch more of the primitive buffer. Effects that can be implemented in a particle system can sometimes be approximated effectively with a fragment shader effect, and — however unsatisfactory to purists — that may be a good solution for mobile devices. Tilers that store independent primitive lists for each tile are less affected by ordering, although they can have a larger overhead when primitives appear in multiple tiles.

Since tiled renderers are a common solution in mobile environments, we can deduce that these overheads are small enough that tiling is still beneficial for many current scenes. The balance between geometry access requirements and frame buffer storage depends on the scene and the details of the hardware — but the programmer can try to avoid the worst cases. Sometimes there can be no substitute for increasing the amount of geometry being rendered — there is no practical substitute to that. Yet these efficiency issues imposed by tiled renderers mean that application developers may benefit from using other solutions for maintaining image quality.

4.2 Tessellation/geometry shaders

Tessellation and geometry shaders can help to support large polygon counts on a desktop system by magnifying the effect of a simple graphics API invocation. Enabling this functionality on a tiled renderer has its own set of challenges. It may seem that an efficient way of supporting large amounts of geometry is exactly what a tiled renderer needs. However, the flexible nature of the programmability in geometry and fragment shaders means that the only practical way to determine which tiles are affected by transformed geometry is to generate and transform each triangle. If the geometry is recorded in its tessellated form during the binning phase of a tile-based renderer, there is no bandwidth

advantage over each triangle being specified explicitly — though the work of generating the triangles is still moved from the CPU to the GPU.

There are approaches to tessellation that can have lower bandwidth requirements, but making the best of any GPU-accelerated tessellation scheme on a tiled renderer involves finding a way to avoid undue memory accesses.

4.3 Silhouettes and shader transparency

The amount of geometry used to represent an object typically has the most visible effect on the edges of the object. It is well-documented technique to use surface shading, for example with ray casting through a height field, to describe an object's bounds. While programmable holes in shaders (e.g. using fragment discard) can result in a slower path through some shading hardware — and of course the fragment shader is evaluated for pixels that are not part of the final surface — this must be balanced against the overhead of additional geometry. Partial shader transparency can reduce the need for super-sampling and an increased shading rate in this approach, but implementing efficient order-independent transparency can be challenging, especially without relying on proprietary extensions.

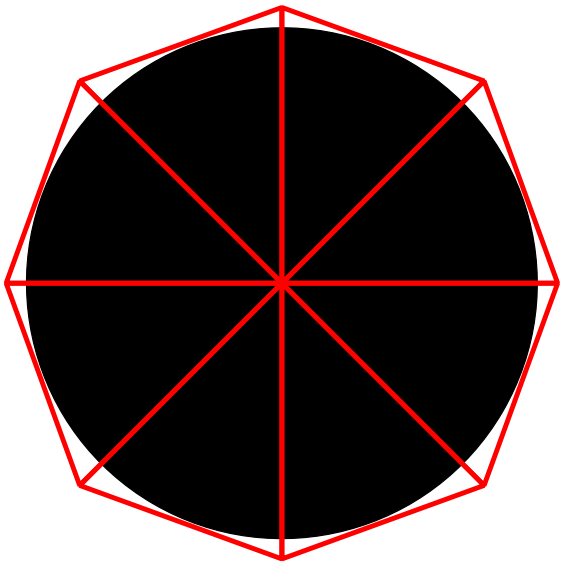

Figure 4.5: Using a fragment shader with discard to achieve a silhouette

4.4 Putting vertices on edges

An alternative to relying on fragment shaders to produce smooth boundaries is to arrange the geometry such that more vertices are visible near the visible boundary of an object. While this is commonly considered to be the work of a tessellation shader, in some cases a similar effect can be achieved by careful relocation of existing vertices — as a simple example, rendering a sphere as its circular silhouette (with the fragment shader making the spherical shape) can be cheaper than using a polyhedron to get the same quality. While this technique can add complexity to a vertex shader, the resulting reduced geometry count and the reduced size of the remaining polygons can help performance.

4.5 Index buffers: caching vs locality

Using index buffers to re-use vertices is a well-known optimisation, allowing both a reduction in vertex bandwidth and allowing the vertex shader calculation results to be reused. However, this cache may be quite limited on a mobile device. If the indexed look-up (which itself requires a small amount of bandwidth) results in frequent cache misses, the memory transfer overhead of using an indexed buffer may actually be significantly higher than using a simple vertex buffer directly.

Step size	*Desktop*	*Mobile #1*	*Mobile #2*
1	$200ms$	$2.6s$	$2.4s$
4	$350ms$	$2.5s$	$7.5s$
16	$550ms$	$4s$	$9.5s$
64	$550ms$	$5s$	$8.1s$
256	$400ms$	$3s$	$9s$

Figure 4.6: Time to render 2^{24} triangles 10,000 times, with indexed vertices and different steps between consecutively-rendered triangles
The desktop system is a conventional immediate-mode rasteriser
Mobile system #1 stores triangles in separate lists per tile
Mobile system #2 has one triangle list, indexed separately per tile

4.6 Culling before submission

Barring an efficient way to exclude geometry from being rendered in the API, the more the CPU can avoid requesting geometry to be rendered the better. As ever, there is a balance to be struck here: much of the work to get performance from a modern rendering API is achieved by submitting large amounts of geometry with the minimum number of CPU calls, so the exclusion of unnecessary geometry must be done without breaking the useful scene description into chunks so small that they incur a large overhead to render.

4.7 Making the CPU do the work

While it is beneficial for latency reasons to keep the CPU as removed as possible from the rendering process, it is worth remembering that the multi-core SIMD-aware CPUs on an embedded device are quite powerful, and often less constrained than the GPU by being in a mobile environment. With a suitable level of buffering to ensure that buffer accesses do not cause the GPU to stall, performance can be gained by sharing the work between the GPU and the CPU cores, especially if the required processing is a poor fit to the GPU hardware, for example by being condition-heavy.

5. *Small texture caches*

5.1 Proceed with Format (Y/N)?

Among the many resources that is restricted on a mobile device is the texture cache. Since bandwidth is already a problem, regular texture cache missing can be a large performance hazard. The first and simplest solution to this is careful selection of texture formats and sizes. If your texture does not need an alpha channel, some hardware can represent it in less space. If your texture would contain enough information in 16-bit mode — or, when available, using an indexed palette — this can save even more space.

The resolution of the texture should be selected carefully. Accessing an unnecessarily large texture is an obvious way to thrash a texture cache, and to require additional bandwidth when that texture is accessed. Accessing multiple textures unnecessarily can have a similar effect — particularly, if the textures are being used to store different components corresponding to the same coordinates, attempting to pack the data can be a worthwhile performance saving.

5.2 Compression schemes

Even more so than desktop systems, compressed texture formats can appreciably help reduce bandwidth requirements further, once the smallest possible textures are in use. Unfortunately, writing portable applications that use compressed textures is made slightly awkward by the proliferation of standards. The good news is that in the latest version of OpenGL ES, support is provided for the ASTC format (family), which provide good quality compression across platforms. At the time of writing, support is still making its way into products, so the biggest risk here is with legacy device support.

Texture compression schemes typically come in families. It is not sufficient merely to decide to use "texture compression" — attention should be paid to the best mode of compression for the content. Not packing an alpha channel can markedly reduce the size of some compressed formats, for example. Since texture compression schemes are typically block-based, better quality can sometimes be achieved be aligning detail to block boundaries.

While it is likely to be less friendly to hardware, in a shader that is light on computation

but heavy on texture access, it can be worth trying to implement a proprietary scheme that suits the characteristics of your data. Sometimes built-in but uncompressed formats can be more effective with some types of content — although there is a risk that some APIs expand out some formats to a canonical, larger, representation.

5.3 MIP maps — friend or foe?

MIP maps provide a convenient way to support reasonable texture filtering quality with good performance on any graphics device. There is typically an overhead to performing trilinear filtering — it is uncommon for mobile silicon to dedicate enough hardware to provide this for free — but MIP maps still provide a good solution for general purpose rendering.

However, it can be useful to pay close attention to what is being rendered. Many mobile applications, even those that use a 3D rendering API to draw the display, produce essentially affine transformations: each pixel in the destination projects to a pixel of the same size in the source. It is common for pixel processing to take place in a pattern designed to achieve good locality of reference during texturing. Multiple texture accesses are sometimes not a big performance problem, but cache misses are. If multiple pixels access the same MIP level, there is a good chance of these accesses falling in the same cache line. Requiring access to multiple MIP levels effectively doubles the cache requirements (although one MIP level would be expected to change more slowly than the other). As such, it can sometimes be more efficient to perform higher-order texture sampling at one MIP level in preference to accessing multiple MIP levels and blending between them, even though the filter support may be larger and more filter taps required.

As ever, there is a trade-off here: there is a limit to the number of texture accesses that can be in flight at any given point, and additional filter taps put demands on shader resources. Particularly, using a large number of temporary registers can result in a loss of parallelism, which in turn can result in an inability to hide the large number of texture accesses. The resulting performance will depend heavily on the texture demands of the shader program, the details of the hardware architecture, and the texture itself.

5.4 More compute, less bandwidth

It is a common approach in fragment shader writing to use small textures to store the results of calculations. These textures can then be used as a look-up table, reducing the amount of heavyweight calculations that the shader must perform.

While this approach remains valid in a mobile environment — especially on hardware for which a simple texture access is appreciably more efficient than the bilinearly-filtered version, and where register pressure from complex calculations can easily impinge upon the available parallelism — it is important to remember the cache limitations of a mobile GPU when applying this approach. A small look-up table can be valuable; one large enough to evict other texture access information from a cache may not be. The amount of latency

that can be hidden in a mobile GPU is small enough that it is much easier to invoke cache thrashing behaviour — and the resulting loss of performance can be enough to hide a lot of calculation, especially if those calculations are simple and can be performed in parallel.

Additional computation can also be useful in reducing the texture requirements. Beyond the obvious (in the sense of "a small matter of programming") sense of actually describing the texture effects algorithmically, shaders can also be used to warp texture coordinates. While on some architectures this removes the opportunity for an optimisation that can reduce the overall shader latency, warping texture coordinates can provide a convenient way to provide texture detail where it is needed and limit the bandwidth requirements where that detail would be redundant.

This optimisation is especially useful since most texture formats have a fixed data rate (since this simplifies random access); this means that, unlike a general image representation format such as JPEG, it is not possible to encode detailed areas of the image with more bits and plain areas with fewer bits. Deforming texture coordinates can offset this limitation, making better use of texture resources and improving texture cache hit rates. The same approach can be effected by careful association of texture coordinates with vertices, but additional fragment shader work can be used to avoid discontinuities and achieve the same result in fewer primitives.

5.5 Shadow maps vs stencil shadows

Several mechanisms for implementing shadows are well known. Of these, shadow maps are known for being relatively easy to generate and efficient to access. Stencil shadow volumes have fallen out of favour, since in most renderers they produce a large amount of geometry that is has a lot of pixel coverage, leading to inefficient access patterns on a typical GPU.

In a tile based renderer, the increased geometry count of stencil shadows remains a problem — and is arguably exacerbated. However, the actual stencil rendering operations can be extremely efficient in a tiled renderer, since the stencil buffer can be retained entirely on the GPU. The stencil rendering operations are light weight, and have simple shading requirements, at least if the accumulation is performed in a single pass, although efficient support for multiple light sources in this solution can be complicated. It is more costly to resort to multiple passes over the geometry, masked by the stencil buffer, to process the illumination component from each light source — while this allows for multiple light sources, it magnifies the geometry overhead inherent in a tiled renderer.

However, mobile architectures also suffer during shadow mapping. Shadow maps are typically quite large structures (even allowing for compression techniques), and locality of reference is not guaranteed — they are a poor fit to the limited texture cache resources on average embedded GPUs. Which of these solutions is best will be decided by the number of lights in the scene, the efficiency with which each approach can be implemented, and details of the architecture. As ever, experimentation is best.

5.6 Environment maps vs cheating

Rendered environment maps — as opposed to pre-baked reflections — suffer similar problems to shadow mapping. Rendering to cube map surfaces necessarily requires an increase in the amount of geometry, especially where the API does not make automatic replication an option. After doing all this hard work, often some of the cube map faces will prove to be invisible in the final scene — and much of the "reflected" geometry that would be visible in a surface may be unseen due to blockage by other geometry. While effective and simple, this solution is indubitably wasteful. If the reflective surface is highly curved, there may be very poor locality of reference in the resulting texture accesses, and the appropriate shading rate can be hard to determine.

If the solution to this were simple, environment maps would not be a popular approach. However, reflections are a good case for the computer graphics tradition of "cheating". If a reflective surface is large and flat, the most efficient approach — even though it means replicating geometry — may be to perform portal culling and render the mirrored geometry directly. If the reflection is rendered collectively, there is a good change of the most efficient mapping to the tiler's binned architecture.

The approach of reflecting the geometry becomes much less appropriate as the number of reflective surfaces increases: particularly, curved surfaces — or approximations to them — would require in inordinately large number of reflections to represent. However, here we have an advantage: the human eye is very poor at identifying objects that have been distorted by being reflected in a curved surface. This typically means that, though an environment cube map may have to be generated, the textures required can be low resolution, and often low-quality proxy geometry can be substituted. If all else fails, it can be worth considering ray tracing for small reflective components in the scene: while the shaders may be very slow, if the number of pixels on which they run can be kept small enough the cost can be manageable.

6. *Hadn't we dealt with that?*

6.1 Precision

Desktop GPUs used to have a number of restrictions within which the application developer needed to work. As hardware has expanded, these restrictions have been lifted, enabling ever-increasing quality in rendering and ease of development.

Mobile GPUs are not less developed than their desktop cousins, but some of the advances that have been made in desktop have carried a heavy overhead in silicon area and power consumption. Once these features were present on the desktop, removing them became impractical, but mobile architectures have often chosen to limit their capabilities in order to focus on maximum performance on common work loads.

A particular case in point is that of the precision at which calculations are performed. It is common for a desktop GPU to need to be capable of accurate single- and double-precision floating-point operations — these GPUs are used for scientific modelling, for financial transactions, for cinematic rendering, and it is important that silicon area be devoted to getting the right results. While GPU computation is starting to grow in the mobile space, the priority is still very much on efficient rendering performance, and basic computer graphics can work very well even with limited ALUs.

Legacy GPUs may only have low-precision operations available, but even more modern solutions can get a significant performance win from working in less than full floating-point precision, and it is not unusual for some full-precision or 32-bit integer calculations to be costly even when working with a "32-bit" GPU. The good news is that ensuring that shaders are optimised for lower-precision work can bring appreciable performance increases; the bad news is that being unaware of the hardware's power optimisations can result in errors in the output which the programmer may not have expected. This behaviour is very dependent upon implementation, and it is not sufficient to test on a single mobile architecture and assume that the application is portable. This is true even if the test platform is known to operate at reduced precision, since sometimes rounding errors can cancel out.

6.2 Texture limits

Desktop GPUs are used to large texture areas being available. This is not universally true in the mobile space. The memory requirements are not necessarily an issue — cellphones with 3GB of RAM are now reasonably common and there is a move towards 64-bit address spaces, mirroring the workstation graphics situation from the Windows XP era — but the requirements of performing calculations that can provide accurate texture coordinate access can be an issue. For this reason, many mobile GPUs retain a relatively small maximum texture size. Of course, the bandwidth requirements of accessing such large textures in a mobile environment are also likely to be a problem. Nevertheless, it can be worth coding applications in such as way as to stay well within portable texture size limits. The number of textures that can be accessed within a shader can also be restrained on some mobile GPUs — much more so than on the desktop.

6.3 Shader resources

Desktop GPUs are designed to handle relatively complex shaders or compute kernels efficiently. In mobile devices, the work typically done by a shader is much simpler, and efficiency in simple cases is of a higher priority. The available die area for shader registers and for cache is relatively smaller on a mobile GPU, and these resources are scaled appropriately for common, simple shaders — not for the potential worst case.

In a typical GPU design, both for mobile and desktop, the number of threads or warps that can be running is determined by how the register file is shared out between them. In turn, the number of parallel threads determines how much latency can be hidden when a memory access is required. Typically it is possible for shader complexity to reach a limit at which the latency for even in-cache texture accesses cannot be hidden by the available number of threads, at which point performance starts to drop off precipitously. This point can be reached with a simpler shader in a mobile GPU than on the desktop, leading to unexpectedly poor performance compared to the theoretical relative ALU speeds of the architectures.

7. *Things you can't do*

7.1 Treat frame buffer processing as quick

It has become common on desktop graphics cards to perform post-processing effects on the frame buffer to increase visual quality. While writing to the frame buffer — at least components of it — can be efficient enough in a mobile GPU, the reduced memory bandwidth means that multiple passes to implement depth of field or lens flare can be extremely costly. It can be more effective to perform a separate rendering pass at a lower resolution and use this as a texture source when applying effects. Interpolation of the smaller image can often be performed with some hardware acceleration, reducing the shader resources required. If only the low-resolution information is required, further gains can be made by attempting to use only nearest-neighbour point sampling.

Those effects which do not rely on accessing multiple pixels — for example, colour grading — can be more efficiently performed by adding the processing to the end of fragment shading. Even if this requires the additional code to be processed repeatedly (if no Z pre-pass is performed and the hardware is unable to remove overdraw for us), the cost of the additional processing is often lower than that of reading a frame buffer-sized texture.

The size of the frame buffer is a major factor in how well attempts to process it will perform. This includes basic rendering, of course. It can sometimes be sufficient to render at a lower resolution than the final display resolution and perform an up-scaling — this has a cost in terms of sharpness, but in some rendering situations, this may not be visible. Most obviously, the format of the frame buffer can have a large effect on memory requirements: while full-colour rendering produces the best image quality, rendering to a 16-bit target can give large gains, especially on simple scenes.

7.2 Chain dependencies

The deferred rendering architecture of many mobile GPUs has a necessary effect on dependent accesses — trying to read from the frame buffer during the rendering process can be extremely costly on a mobile GPU. In theory, access within the current tile ought to be cheap, but it may not be possible to configure memory access to support this — requiring

that the frame buffer and all intermediate rendering state be written out to memory. Most commonly, the whole frame will need to be written out to external memory before a pixel access can be performed, and this includes all intermediate state used during rendering which might otherwise have been able to stay on-chip. While frame buffer access also affects parallelism on an immediate-mode renderer, the overhead can have a catastrophic effect on a tiled renderer's performance, removing many of the performance advantages of using a tiled architecture in the first place. Needless to say, if there is any possibility of using a substitute for frame buffer data, one should consider it.

It is not only frame buffer accesses which can introduce this kind of cost in a tiled renderer: in some cases, switching between render targets can be a trigger to processing. If the same render target is used multiple times within a frame, intermediate state can again need to be written out to external RAM, to be recovered when rendering to that target continues. It is therefore often useful on a mobile device to collect all accesses to a frame buffer together — for example, rendering shadow and environment maps before processing the main scene, rather than producing them at the time their associated geometry is reached.

7.3 Trust optimisations to work everywhere

The underlying architecture of common desktop devices has converged; with exceptions, optimisations that are applicable to one architecture will work well on another. As may be seen by the number of caveats in these notes, the same is not true in the mobile space: there are more solutions being attempted to resolve the power and space constraints imposed on these GPUs, different innovations are being tried, and there are more key players involved in producing competitive units.

Unfortunately for the application developer, the route to the best performance can often be to detect the hardware in use and attempt to accommodate it. While much of the optimisation work is implicit and performed by the drivers, with any non-trivial application it can be easy to trip up the driver's optimisations, especially if code is tested only on a subset of GPU architectures. While GPU vendors are in a constant battle to ensure that their hardware works well on all application code, an awareness of the different rendering solutions can make life easier.

One particular situation in which different rendering paths can be useful is that of tile-based deferred renderers that remove the necessity for Z pre-pass. For these architectures, any processing from front-to-back order, or explicit initial passes to prime the Z buffer, tend to hurt performance — it is more important to keep the number of state changes that occur during rendering under control. For other architectures, at least when running complex shaders, and explicit Z pass or some sorting prior to submission can provide an important performance win.

Even the layout of ALUs can vary widely. In some architectures, a warp can consist of many threads, and losing parallelism between shader invocations can result in a large loss of performance: for these it can be important to try to eliminate as much branch divergence from code as possible; this is especially the case in computation APIs. On other architectures, the cost of divergent branching can be negligible. On some architectures

the native work unit may be a 32-bit ALU operation; others work in larger SIMD blocks, which offers good performance on a number of graphics-related workloads but can suffer performance problems when the shader compiler is unable to expose enough instruction-level parallelism to keep the ALU busy. Some architectures have dedicated processing that can be performed on entry to a shader which can reduce the computation overhead of the shader itself. It is generally the case that mobile GPUs are less parallel than their desktop cousins, and can lose slightly less performance to branch divergence.

Differences in cache size can also heavily affect algorithmic performance, considering the heavy cost of external memory access, exacerbated by differing levels of support for texture formats. If an implementation converts a particular format to a 32-bit texture during upload in order to support it, both bandwidth and caching performance can be much lower than expected: just because a driver claims to support a particular format does not guarantee that this support needs to be efficient.

7.4 Think of the display as "small"

Finally, there is always a danger when writing for a mobile device to assume that, because the display area is small, the amount of frame buffer processing to be performed will also be small. While there are short cuts that can be taken based on the display size — something we will discuss later — the actual pixel count on a mobile device is typically no lower than for a desktop system, especially since resolution has been used as a differentiating factor for mobile devices but has — relatively — stagnated in the PC market. However restricted the GPU performance and its memory bandwidth may be, it is dangerous to consider the rendering demands to be similarly cut down.

8. *Form factors*

8.1 So much for common resolutions

In the past, it was common to need to support a range of pixel counts and, to an extent, aspect ratios, because the user could configure a CRT to display many different resolutions. Countering this was the knowledge that, with a CRT, the application author could impose full-screen resolution choices without impact on the image quality.

The proliferation of desktop LCDs removed the flexibility for the developer: the fixed resolution of panels meant that picking an arbitrary resolution could have a negative effect due to interpolation. Fortunately for developers, the number of resolutions in which desktop panels were manufactured was relatively limited, especially if the most obscure resolutions were excluded. The move to wide-screen panels introduced more aspect ratios with which to be concerned (typically 5:4, 4:3, 16:10 and 16:9). Manufacturing demands of laptops increased the range of common resolutions somewhat, yet the number of options was still relatively manageable.

The requirements of price/performance decisions and different product shapes means that mobile devices extend the number of resolutions that need support yet further; higher resolutions and the largest viable screen have improved the web browsing and video playback abilities of devices, and it has been the job of the application author to ensure that other software is sufficiently flexible to accommodate all possible options. The developer must also accommodate different physical pixel sizes in a way that is rarely a priority in the desktop space. A simple desktop application can work at a single size in a scrollable window. Mobile applications are, almost exclusively, designed to take over the entire screen, and therefore must accommodate whatever shape or size that screen might be. The wide range of display resolutions means that relying on the resolution of the screen and anti-aliasing support to make the display look good may not work well: some form of hinting and attempts to align content to the pixel grid can be valuable.

Common Android device resolutions (source: Wikipedia)			
240 × 240	640 × 480	1024 × 600	1440 × 900
320 × 240	800 × 480	1024 × 768	1920 × 1080
400 × 240	800 × 600	1280 × 720	1920 × 1200
480 × 320	854 × 480	1280 × 768	1920 × 1280
640 × 360	960 × 540	1280 × 800	2560 × 1600

8.2 Aspects of ratios

Since the move to fixed-resolution LCD panels, panels have generally been made in relatively few shapes. Mobile devices are necessarily somewhat more variable, allowing designers flexibility in device shape. The onus is on the application author to make good use of the screen area. Simply stretching the display can lead to highly-distorted textures, yet the range of available screens can make explicit support for each shape impractical. This means that applications need good dynamic layout support (and testing).

8.3 Button bars

Despite this flexibility, a finite number of panels are on the market. This ought seem to simplify the range of resolutions that need considering. However, some platforms devote areas of screen to interaction mechanisms. Even within an operating system, hardware may be designed with dedicated input areas, or some of the screen area may be appropriated for soft buttons. This effect can increase the number of resolutions that need to be supported inconveniently, especially if different platform revisions change the button bar size.

8.4 Rotation

For an ideal user experience, applications must also support the user's choice of screen orientation. While less important for some games (which can enforce that the user rotate the device to landscape mode, for example), it is still convenient for the user to be able to choose which landscape orientation to support. For general applications, the author may provide a completely different experience when the device is in a portrait orientation, or choose to share large portions of the rendering framework between configurations.

8.5 From phones to tablets

A user's experience depends not only on the screen resolution and shape, but also on its physical size. This is especially true since most mobile devices are touch-screen driven. Touch-screens place user interface requirements on software — particularly the lack of pixel-perfect selection ability and the need for larger target areas. The ideal way to present an application on a small cell phone may be very different from how it appears on a 12" tablet. Authors need to be aware of (currently) unusual form-factors as well: mobile operating systems are increasingly used on televisions and large-format touch screens, in automotive scenarios, and in very small devices such as watches. The pixel count of such devices is not necessarily closely related to the display size.

The size of pixels has a direct effect on the necessary image quality. In a modern high-density cell phone, individual pixels are small enough that they are hard to see at

any viewing distance; for static content consumption, this is a desirable property. In a large tablet with a relatively modest resolution — as in "feature phones" — individual pixels are far more a concern. This means that the author may need to be more directly concerned with anti-aliasing quality, hinting (aligning scene elements to the screen pixel grid) and even sub-pixel rendering.

The best use of screen real-estate also varies by device format. While 1920×1080 is a common resolution for both computer screens and cell phones, the larger angular view offered by a typical monitor compared with a cell phone held at a comfortable viewing distance makes the experience very different. This can be seen by the layout of most web pages: it has become conventional for web text to have extensible borders on either side, ensuring that the text itself does not expand beyond a comfortable line length for viewing. Text in newspapers has long been split into multiple columns for the same reason, and text layout packages typically support a large border when filling a page with a single column.

On a typical cell phone, using the same font size in pixels that would be appropriate on the desktop results in something very hard to read, and a waste of screen real-estate. The pixel size of mobile text is therefore typically larger than on the desktop, with the extra resolution allowing an improved text quality. The additional real-estate of a desktop system is better used to support multi-tasking or control elements. Tablet designs fall between these extremes: it can be uncomfortable to view text across the entire width of a large tablet in landscape mode, yet — especially in portrait configuration — there is little area for fitting additional content on the screen. These user-interface concerns apply even to mobile gaming with unusual interaction mechanisms.

Figure 8.1: Increase in screen sizes over time
2010: HTC Desire — 3.7" 800×480
2011: HTC Sensation — 4.3" 960×540
2012: Samsung Galaxy S III — 4.8" 1280×720
2013: Google Nexus 5 — 4.95" 1920×1080

9. *Getting it on-screen*

9.1 Composition overhead

In desktop systems, particularly when gaming, it is common for an application to be able to take over the entire display. Mobile systems can sometimes stop this from being possible, to a greater or lesser extent — a status bar and, often, soft buttons can cut into the screen area. This can result in a display of a slightly unusual size (it would be optimistic to assume that all such intrusions are conveniently tile-aligned, for example). On many cases, the composition step used to generate the final display can be performed by overlay hardware, similar to the solutions available for hardware cursors on desktop systems. The flexibility of this overlay can sometimes be limited, however, and the selection of the wrong sizes, formats or orientations for each composition layer can be enough to force and additional software compositing pass.

9.2 Getting data into your API

An optimist may look at a shared memory architecture and believe that this approach will avoid all need for texture and buffer uploads. Sometimes, the situation can be this simple — although we do not always want it to be. It is common for graphics hardware to make use of optimised rearrangements of texture data in an attempt to maximise locality of reference during rendering operations — enforcing the application's image layout may result in a performance hit in rendering. In addition, to save silicon area, some hardware may not natively support every combination of texture format — for example, supporting all possible arrangements of A, R, G and B can be wasteful; the texture upload process can be used to transform the texture layout as needed.

In other cases, especially if the host application is frequently updating the texture or vertex data, it can be beneficial to have a direct mapping of the memory, where APIs make this possible. As discussed above, the danger here is that the application must aware of the deferred nature of tiled renderers: updating a texture every frame in the expectation that it is no longer "in use" may cause costly blocks of the pipeline or copies of texture data to ensure that the correct data is still available when the renderer needs it. Estimating the length of the pipeline and double- or triple-buffering can help greatly.

9.3 Communications

The benefits of efficient texture handling are most evident when the content is not sourced in a host application. Streaming images directly from the camera into a rendering API can be much more efficient and require much lower latency than requiring host copies to perform these operations. The same is even more true — and more frustrating for the user — when it comes to transferring texture and geometry data between other APIs running on the same hardware as the graphics API. There is no good reason, other than API restrictions, why a GPU compute API should need to involve the host application when providing processed images or vertices for rendering by a graphic API.

Efficient means of communication between APIs is an area under development. Check for your latest extensions to ensure that the most efficient approach is being used!

10. *K.I.S.S.*

10.1 Frame buffer compression

Mobile graphics hardware and drivers perform a lot of complex optimisations in order to try to maximise performance. Unfortunately, the more complex the application's rendering pattern, the harder it is to perform some of these optimisations. One example of this is that there are architectures with optimisations to avoid updating the frame buffer if the content is unchanged from its previous value; in other hardware the frame buffer itself is compressed to reduce bandwidth requirements. While a complex background full of slightly animated objects (dancing plants, for example) may appear not to involve much movement, it can have a high cost in terms of memory bandwidth compared with a simpler interface. A cloud floating in the sky may have a much bigger overhead than one might expect from the number of draw calls required to render it.

Not all frame buffer updates are caused by deliberate animation effects. Minor changes in geometry caused by a combination of user input and rounding error can cause minor frame buffer changes. These changes may be as small as a fractional change in sample position, or a sub-pixel change in geometry which introduces an anti-aliased shift in pixel values. Particularly on high-resolution displays, these changes may not be visible to the user, and certainly have a small enough effect that they offer very little improvement in the user experience. Performing deliberate quantisation of parameters during rendering to avoid these changes — potentially with some form of hysteresis to accommodate value that fall near a rounding error boundary — can enable large bandwidth savings.

The frame buffer may also itself be compressed, typically using a lossless, variable-rate scheme. Because the compression depends on predicting the content of the frame buffer, simpler images are likely to compress better and have lower bandwidth. In a 2D application, aligning the regions of detail so that they touch a limited number of tiles can allow other tiles to compress better.

10.2 Tiles don't excuse massive overdraw

Tiled renderers have very fast access to the local frame buffer, compared with the external frame buffer bandwidth. It can be tempting to treat rendering performance

within the frame buffer as free, or only limited by the geometry overhead, because the frame buffer access is so much faster.

On GPUs which perform in-order shading, it can be dangerous to ignore both the overhead of running shaders and the bandwidth requirements of texturing. A Z pre-pass or an attempt to perform front-to-back sorting can save a large amount of shading work, even if all the rendering remains within the frame buffer.

Even once these optimisations are in place, rasterisation is not entirely free. Large fragments within a tile can still take many cycles to process. Sometimes, ordering and a Z pre-pass are not enough to hide calculation costs — most notably when alpha blending is used. While some hardware has accelerated blending available during pixel write, a fully-shaded fragment still contains multiple cycles of processing. If it is possible to determine the position of multiple scene components algorithmically, it can be worth considering performing a combination step in a single fragment shader.

This is especially true for static scene components: composition can be optimised by blending multiple layers within one fragment shader rather than performing a number of independent alpha-blended passes. As ever, there is a trade-off against the number of texture stages available, the potential loss of parallelism by use of shader registers for coordinates, and the risk of thrashing the texture cache lines, so experimentation on the target architectures is worthwhile.

10.3 Pre-process your textures

As we have stressed repeatedly, texture access is costly for mobile devices — in bandwidth and computation overhead. Many applications that use a 3D API for rendering are still present a 2D interface, however — or behave that way much of the time. The requirements of portability mean that it is common for high resolution textures to be needed, but in many devices these can be excessive.

Where possible, a large saving in bandwidth requirements can be gained by processing the textures on start-up to scale them to the correct display resolution — though the author needs to be aware that there can be more than one "correct resolution" when the application supports portrait and landscape rendering. This can have the advantage of allowing a higher quality of scaling than would be practical during rendering, and may allow textures to be generated by pre-composing elements in desired positions, allowing the components to be aligned to a pixel grid rather than relying on uniform scaling.

Not only can this approach reduce overdraw and the number of draw calls, it can allow the footprint of the application to be reduced by allowing the texture memory for the components to be reclaimed. If the final texture can be rendered using point sampling, the fragment shading cost can be reduced — while there is often hardware support for bilinear filtering, point sampling often requires less latency, reducing the amount of parallelism required for maximum performance and allowing more advanced algorithmic effects to be applied during shading.

11. *If in doubt, cheat*

11.1 Micro pixels and macro effects

The extremely high resolution of many modern mobile displays — typically exceeding 500 pixels per inch — is enough that individual pixels can be all but invisible, especially if the contrast is low. Anti-aliasing can give the impression of a higher resolution, and a high resolution can compensate for anti-aliasing: depending on the content, anti-aliasing can be unnecessary, and if alpha blending or multi-sampling can be at least partially disabled, there can be a significant performance win. With a sufficiently high-resolution texture, the same can apply to filtering — having an off-by-one error in a point-sampled pixel result can be visually imperceptible.

However, it is necessary to bear in mind that many small pixels build up to a large effect. Moiré patterns can cause accumulative effects on visibility — but this is true even when the samples are super-sampled, in many approaches. Sufficient and correct filtering is always needed when the content contains repeating patterns, but this does not mandate its use in producing a visually-accurate boundary to an area of solid colour — for example when rendering large text.

As ever, there are times when relying on high display resolution will be inadequate. While many modern mobile devices do have high resolution screens, this is not a universally safe assumption. Older devices can have appreciably lower screen resolutions; embedded software often runs on larger screens such as tablets and, especially, televisions, which can have very large pixels. The user's viewing distance is also a factor — even a large television may make use of limited anti-aliasing if the viewer is sufficiently distant (billboard photographs often have quite low resolution, by comparison), and even the highest resolution current displays can have visible pixels if the user is viewing very closely.

11.2 Frame rate is the enemy of play time

For a smooth user experience, it is obviously beneficial to support a sufficiently high frame rate. However, "sufficiently high" is a concept that carries a trade-off: few optimisations can make as much of a difference to the power consumption of an application as

simply not drawing as many frames. It can be tempting to assume that a target of 60 frames per second is always desirable, but the visual impact of dropping to a lower frame rate, depending on the application, can be quite unintrusive — cinema has survived for nearly a century with an effective 24 frames per second rate.

There is no requirement that the frame rate for an application be fixed. When the user is not interacting with the device, avoiding redraw — or reducing the redraw rate to a minimum that is suitable for ambient animation or informational effects — can save a lot of power, with the frame rate ramping up automatically once responsiveness is needed.

Not the whole frame needs to be rendered at full speed. There have been good demonstrations of how insensitive the eye can be to delayed indirect illumination effects — so at least some heavyweight calculations can be performed at lower frame rates. The same can apply to reflections — though the effectiveness in that case depends on how visible the reflections are. It can easily be acceptable to perform lower rates of rendering — or composition — to intermediate buffers and textures in some situations.

11.3 Interpolate (or extrapolate)

Just because an application has been designed not to run at the device's maximum frame rate does not mean that an interactive experience cannot be emulated. It is quite common to know in general how large areas of the image are moving, allowing the screen area to be redrawn shifted by an offset. There is a caveat here with tiled renderers — reading the output of the previous frame's rendering can be a performance hazard — but solutions like this can provide much smoother motion than simple rendering at the lower frame rate. If latency is not an issue, it may even be possible to perform intelligent interpolation when the frames bracketing the interpolation are fully known.

Generating additional frames by extrapolating from previous motion can be useful as a means of reducing perceived latency — which can be especially helpful in a tiled renderer that has an inherent frame delay. While one solution to hiding latency is to extrapolate the user's inputs — assuming that a turn or scroll will continue at its current rate, for example — this is only of limited help if the rendering itself takes a long time and the rendering rate is limited. Combining this approach with motion interpolation and image-based rendering can provide good interactivity and perceptual responsiveness at a reduced cost.

11.4 Cloud rendering

Performing high quality rendering on a mobile device is inherently a high-power operation, no matter the effectiveness of the techniques discussed in this course. However, many mobile devices have convenient network access — at least most of the time. This means that the small, power-constrained GPU local to the device may be able to offload some work to a remote, high-power render farm solution. Latency is an inherent problem in cloud-based systems, although not always insurmountable for many styles of application.

However, it may be that not all the rendering is sensitive to the same latency concerns — for example, direct visibility testing and direct illumination, which are relatively cheap to compute, may need to be rendered on the local device, but indirect illumination effects might be best rendered in a cloud.

Of course, the network communication itself also uses power — and the networking power drain (and cost of processing the data being transmitted and received) needs to be balanced against that saved by not performing the rendering locally. This solution can, however, be effective in expanding the absolutely rendering performance beyond that available in a mobile form-factor, particularly in high-quality, low frame-rate applications such as product visualisation.

Unfortunately, it is common for mobile devices to lose reception sometimes — or for the network quality to be insufficient. Being clever about the amount of data that needs to be transferred can help appreciably, as well as endearing the developer to the user who may have to pay for an bandwidth allocation, but the user should be provided with a fall-back path to ensure that the application remains usable in the absence of a connection..

Conveniently, when the remote rendering is used primarily to provide secondary visual effects, it can be relatively easy to incorporate a lower-quality rendering solution into the pipeline — but if the main bulk of the rendering is transferred into the cloud, the back-up solution may require a complete replication of the rendering path. This is not necessarily catastrophic — mobile devices can often be fast enough to support reasonable rendering quality, for all the concerns in this course; the issue is that the power demands of doing so are detrimental to the user experience — but if the on-device rendering only has to last until a network connection is restored, this can be an acceptable cost.

11.5 A greasy blob with a hand in the way

When providing graphics for user interaction, it is easy to forget that there is something on the other side of the touch screen. Performing high quality, low latency rendering for the interaction point seems like a good idea in a desktop environment. On a mobile device, the focus may be where the user is touching the screen — which inherently means that there is a finger between the display and the user's eyes. This approach can be used to hide quite some otherwise quite complex rendering.

The quality of the display itself can also be limited. When developing on a desktop system, it can be easy to assume that every pixel and every colour gradation is visible. The small display on a mobile device is typically small, not clean, and often viewed under unfavourable lighting conditions — it is easy to spend too much computational effort trying to produce effects that the user cannot see because of the practicalities of the device.

Even under ideal conditions, it would be optimistic to believe that the user can make out every pixel on a mobile display. Pixel count is not the only differentiating factor in a mobile purchase, and as resolutions increase to suit the minority of users (and marketing departments), pixels will increasingly be smaller than can be made out by their owners.

12. *Aesthetics are not everything*

12.1 Look as good as you can, but no more

Pretty computer graphics have a "wow factor" — they can make an application, especially a game, seem appealing, and show off a device well on a shop floor. Sufficient graphics to tell the user about the state of the application can be intuitive, and even immersive. There is no harm in taking pride in producing the best visual quality practicable.

But the quest for perfection is costly. Every minor step in image quality places increasing demands on the device and on the developer. While users value pretty æsthetics, it can be easy to forget, while staring closely at a few troublesome pixels, that the most important thing is functionality.

12.2 The cost of artistry

To the application developer, it may seem unprofessional to provide a "good enough" solution. While heroically trying to provide the best quality applications for users, it can be easy to lose sight of the fact that the user may not appreciate the result.

One reason that mobile app stores are so popular is that the typical cost of software is quite low — particularly in relation to the PC and console market, for which the effort spent on development has often had to justify the cost of the shelf space required to distribute the software. Doubling the developer time on a mobile app may improve the user experience, but it will also increase the development cost and potentially necessitate increasing the price of the app. Customers may be far more interested in a bargain than in perfection.

The highest quality rendering may demand cutting-edge hardware, even after all tricks and optimisations have been exhausted. This is a great way to demonstrate the merits of upgrading to the latest cell phone, but the majority of the market (who, ironically, may be prohibited from an economical upgrade by network contract deals) may simply be excluded from being able to use the app effectively.

Even when the hardware is capable, high rendering quality places high demands on power consumption and phone heating. Slightly enhanced graphics may look nice, but software which makes the device uncomfortably hot to touch and which empties the battery

in a few minutes is frustrating for the user. Games or applications with simpler graphics may lack the visual appeal, but if the experience of using the software is the same, static scene components, lower resolution or bit depth and large static areas — potentially of solid colour — can give the user most of the user experience but for much longer.

12.3 The app store vs the Blu-Ray

Developers familiar with PC and console environments may be used to vast amounts of geometry and texture data, made available over multi-gigabyte physical storage media. There is a temptation to try to provide mobile users with the most complete experience possible and provide a similar amount of content. With the relatively fast connections on recent devices, this is certainly possible, and a number of applications perform in-app downloads of a large amount of content.

However, just because something *can* be done does not meant that it *should* be done. When developing with a local connection on a desktop device, it can be easy to forget that users often have to pay for their data allowance. A download of a few hundred megabytes may be fast (or at least, fast enough — users are rarely pleased at having to wait for a few minutes before the application works), but it can still be costly on some contracts. In addition, storage space on mobile devices can be quite limited — and memory capacity even more so. However proud the production team may be of the ten-minute, full HD, effects-laden introductory video, many customers would rather have the space back for music and personal photos.

13. *Using your advantages*

13.1 Pixel density is good for detail

While much of this course has been spend apologising for the limitations of mobile devices and trying to circumvent them, there are advantages as well. For example, the high pixel density on mobile devices make them well-suited to presenting very fine detail. A desktop display with a similar resolution is often large enough that the image looks pixellated, which can impact the perceived clarity. A 500ppi display looks very sharp because it *is* very sharp. Photographic detail is preserved nearly to the level of the eye, text can be rendered with a very high perceived quality — especially since the size of text in relation to the total screen area is typically so much larger than for a desktop display.

13.2 PenTile and colour frequency

A number of mobile displays use unconventional pixel layouts as a means to balance efficiency and brightness. Desktop systems have, for years, supported sub-pixel rendering to achieve higher spatial luminance resolution — particularly in font rendering. Though similar solutions may be beneficial — patents permitting — for legacy, lower-resolution devices, the computational costs for these solutions may not be justified. Alternative layouts used in mobile devices often do not represent the full colour resolution at each pixel; the perceptual image quality reduction is considered to be low compared with the power savings.

Developers can take advantage of the reduced spatial resolution in some channels to reduce the amount of graphical work required. Shaders and texture formats can be optimised to generate only the representable amount of pixel resolution. The same rendering performance advantage can be used in a more conventional display layout, at the cost of reducing the display quality slightly.

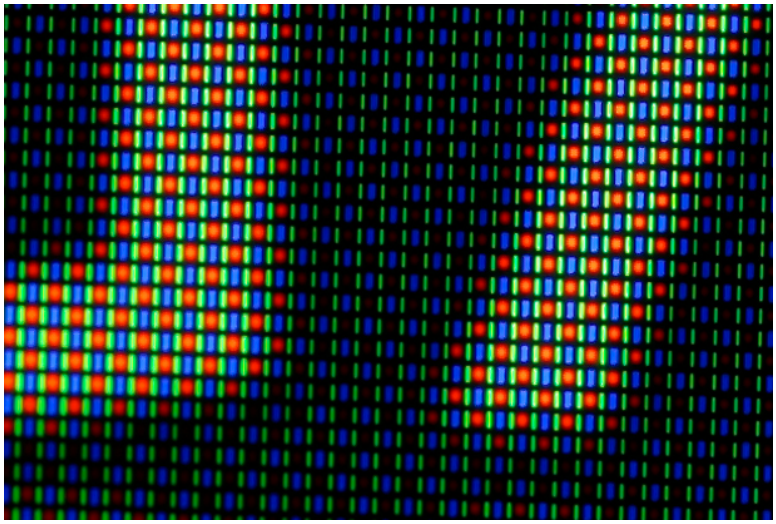

Figure 13.1: PenTile RGBG sub-pixel layout on an OLED display

13.3 Suit the application to the interface

It is tempting to take popular desktop or console software and attempt to port it directly to a mobile device. The most obvious problem with this tends to be with interaction methods — most mobile devices lack a usable keyboard or mouse, and gaming interaction mechanisms — whether direct controllers or based on object tracking — do not translate well. In contrast, touch interfaces are ubiquitous on mobile devices, and most products offer a range of other sensors such as cameras, microphone, GPS, accelerometers and gyroscopes. The pixel-perfect interface of a desktop machine is good for content creation, but inconvenient to emulate in a touch screen.

The design of the graphics of an application must be tailored to the interaction mechanism, but also to the viewing experience. Mobile solutions — video eye-wear aside — cannot provide the immersive experience of a cinema display or a large screen television; this means that a first-person shooter game will always seem different on a tablet to the TV experience, but mobile solutions offer natural interaction for augmented reality and multi-touch solutions. Casual gaming has become ubiquitous, becoming the most popular use of cell phones. There is less call for immersive detail, more for simple, easily-visible content that can still be used when viewing conditions are poor and the player cannot dedicate undivided attention to the experience. Mobile applications are also being used by an increasing number of less technically-savvy users, who benefit from means of interaction that are simplistic and, in many cases, easy to see. This necessary change in style has a corresponding effect on the rendering solution used.

Not that it is impossible to play a first-person shooter on a tablet — or a simple puzzle game on a PC. But these are the exception, not the rule.

14. *APIs are changing*

14.1 Historical APIs getting more flexible

Function innovations tend to happen in desktop hardware, not least because there is competitive pressure exclusively on the graphics performance separate form the rest of the system, and because it is easy to update driver software in desktop computers. Mobile hardware is an inherent component of the system, and replacing the mobile GPU without completely replacing the device is infeasible. Drivers on mobile systems can also take longer to validate and update.

Mobile GPUs started out as extremely primitive devices, and their desktop equivalents have had a large head start when it comes to capabilities. The APIs of mobile devices have followed suit, being heavily cut down compared with desktop equivalents — although some of the missing functionality has more to do with the removal of deprecated functionality than with GPU abilities. While desktop GPUs still support more features than the latest mobile GPUs, the gap has narrowed appreciably in recent generations. As more functionality transfers from the desktop graphics interfaces, the capabilities converge — even if, as this course has discussed, the best way to use that functionality may differ. As more features become available, it will be possible to support ever more advanced graphics in a mobile form-factor — and, as ever, developers will find ways to push graphical techniques beyond the expectations of the API designers.

14.2 Next generation APIs

While OpenGL ES has served the graphics industry well for years, there are some cases where the API obstructs performance. Historically, OpenGL's support for multi-threaded operation has been rudimentary, with the result that almost all rendering operations take place on the main rendering thread. High-level APIs are so effective at abstracting the hardware that it can be hard for a driver to establish what the application wishes to achieve. The optimisations in the hardware and driver can be broken when the system must allow for application behaviour that has the possibility of producing incorrect rendering. A GLES driver also has responsibility for its own memory allocation, which allows for internal texture representations to be hidden, but also makes it hard for the user to alias

memory use and to support texture streaming.

There has been a recent move to more explicit graphics APIs: Mantle, Metal, DirectX 12 and Vulkan. These allow high-overhead parts of rendering such as command buffer generation and shader compilation to be processed when the user can afford this overhead, and on a thread other than that responsible for geometry submission. They can also allow much of this costly work to be reused. By being so explicit, less driver intelligence is needed, greatly reducing the driver CPU cost in some cases. The cost of these APIs is that the application must explicitly request the operations required during the rendering process, which exposes the different paths required by different rendering hardware. The experience may have benefits even to GLES programmers, since visibility of the operations performed by the driver can give insight about unexplained overheads. While an application that is completely limited by the performance of shading may see little benefit from a next-generation API (although some allow improved management of multiple rendering passes and therefore bandwidth reduction), it is easy for a GLES application to get tied up in driver behaviour and not saturate the GPU.

The additional flexibility required by next generation code means that applications written for these new APIs may be more verbose than GLES. Prototyping may be faster in an older-style API; the new APIs likely require less effort to turn into a production-ready optimised solution, since there is less need to reverse-engineer driver behaviour. There are tasks that remain better-suited to GLES, but developers should not reject explicit APIs in the belief that they are too complicated. Layers and utilities exist to simplify the use of the new APIs, and game engines are leading adoption, so there may be ways to limit the cost of support to applications. However, currently, GLES dominates the adoption of mobile graphics APIs, and it should not be dismissed yet.

15. *The future*

15.1 Hardware getting faster

While the power constraints on mobile GPUs are not going to disappear in the foreseeable future, there is no reason to believe that the performance of mobile devices is going to stop increasing — or that efficiency gains will not continue. This means graphics quality will continue to increase. The ways to get the best performance from a mobile GPU may well change over time as more techniques become available, but there will likely always be a need to work around bandwidth limits.

For many years, desktop operating systems and productivity software expanded to fill the increasing capabilities of CPUs. Relatively recently, we have reached the stage where CPUs have become fast enough to manage the most typical things that people do with them — this has in part been responsible for a reduction in profitability in the PC market. The PC graphics market has long been driven by gaming, a market that has also reduced somewhat with the rise of casual gaming. However, mobile GPU capabilities still lag behind the requirements of application developers — mobile GPU performance continues to grow, and application software continues to expand to use all the performance available.

15.2 Caches getting bigger

One change that is likely as a consequence of the reduction in process size is the increase in on-chip caches. To an extent, this change will negate some of the advice in this course — it will become less necessary, to accept a small reduction in image quality in return for better cache performance.

However, as scene complexity increases, the demands on caches are only likely to increase. The onus is on GPU developers to find the best balance to meet user demands — but it is up to the application developer to make the best use of the resources available at any time.

15.3 Shared address spaces

Current mobile demands make the only practical solution for memory access from the GPU and CPU one which involves a shared interface. The address space flexibility of desktop systems is also increasing, and APIs are likely to make better use of shared address spaces in the future. This can only make it easier to reduce the overhead of communication between the CPU and GPU.

15.4 Memory busses still limited

As computation performance increases, bandwidth has always proven to be an increasing problem. Ever more clever solutions to this problem have been attempted over the years, but there is no way to add more raw bandwidth to a system without using more power and adding more cost to the device. Whatever hardware solutions are found, reducing the memory requirements and improving locality of reference will always be valuable optimisations.

15.5 Are device resolutions levelling off?

For several years now, mobile device resolutions have been increasing, acting as a differentiation factor. We have reached a point where the pixels are hard to see, and the benefits of increasing resolution are becoming questionable, especially when the considered resolution does not fit a standard media content format. Ever increasing display resolutions have a cost in display brightness, colour space, reliability, and the difficulty of producing defect-free panels. Added to the computation cost of generating content for the higher screen resolution, it seems that display resolution is unlikely to continue increasing incrementally for the foreseeable future.

There is a tempting misconception that pixels *should* be visible. For rendering a general-purpose scene, the pixels are only a representation of infinite-resolution geometry. Ideally, pixels should be small enough to avoid being visible as square regions, but only effective anti-aliasing can represent a fully continuous source. The transition point between visible and invisible depends on the user and the viewing conditions, but there is no doubt that the pixels on the latest displays are discretely visible to ever-fewer users. What remains to be seen is when the market decides that pixels are small enough, and that higher performance and better efficiency are more important.

15.6 Efficiency increases

Whatever the future brings, we can assume that GPU designers will continue to find new solutions for managing the power requirements of their designs, while allowing performance

to increase. The trick for the application developer is to ensure that applications can make use of these solutions — just as the GPU vendors need to ensure that their hardware optimisations are as applicable as possible to real world software. GPU design and driver software makes increasingly intelligent attempts to reduce the overheads of rendering. While the application author must avoid incurring unnecessary overheads, often the best thing to do is to behave as simply as possible. The best performance comes from letting the driver apply its optimisations: this is most likely to happen if the GPU vendors anticipated what the user was going to do — behaving predictably and in line with common practice — and if the rendering scheme is simple enough for the driver to optimise.

15.7 Is the desktop route the way to go?

Desktop graphics cards have mostly avoided tile-based rendering solutions. It is too common to need to support a very large amount of geometry, particularly in the workstation market. As mobile scenes grow in complexity, one solution is to move to immediate renderers even for the mobile market. This is certainly a solution to the needs for tessellation and geometry shading in mobile APIs. Frame buffer access optimisations in immediate renderers can partially compensate for the bandwidth advantage of a tiler, but on current work loads, tiled renderers seem to have a current advantage. Whether this remains true in the future remains to be seen — and depends heavily on what is to be rendered.

15.8 Projections

Future innovations may not only happen in the design of the GPU. Much of what has been suggested in this course is applicable to small displays, but there have already been some attempts to incorporate pico-projectors into cell phones, giving the ability to fit a very large display in a small device. Other solutions with flexible displays have been researched, and there are emerging solutions for using remote, larger, displays with mobile devices. Combined with the remote rendering solutions mentioned above, we may come to live in a world where the data is mobile, but the display and the GPU is not.

16. *Wrapping up*

16.1 Bandwidth will always be constrained

In summary, mobile graphics is constrained, more than anything else, by bandwidth limitations. More bandwidth means more power — worse battery life, more device heating — and more cost. The best way to extract performance from a mobile GPU, therefore, is to minimise the bandwidth requirements of the application.

16.2 Make maximum uses of local memory

External memory bandwidth — the RAM interface shared with the CPU — is the most precious resource. Making best use of the on-chip frame buffer and caches — limited though they may be — can have a big effect on performance. Mobile caches are smaller than desktop GPUs, yet the penalty of missing the cache is greater.

16.3 Don't go crazy with the geometry

Many mobile GPUs use tiled rendering. These architectures make efficient use of frame-buffer bandwidth, but suffer increased overheads as geometry count increases compared with the immediate-mode rendering common in desktop devices. The extent of this overhead depends on what exactly is being rendered, but, as a general guideline, it helps to find a way of producing an effect without resorting to increasing the triangle count.

16.4 Textures — size and locality

Textures can be the biggest consumer of caches and external memory access. Keeping the textures small — in resolution and in format — can be the secret to staying within the device's limits. Texture bandwidth can sometimes be traded for additional shader computation, in terms of filtering, colour format and preserving locality of reference. The

large bandwidth available to desktop GPUs makes them much less sensitive to abuse with large and incoherent textures; moving to a mobile requires more care to preserve performance.

16.5 Desktop assumptions don't apply

Mobile GPUs diverge more than desktop systems. A good optimisation on one system may not apply, or may be actively detrimental, on another. Mobile hardware has more restrictions than desktop, since any non-critical functionality costs space and power. Not every desktop API feature is yet available on mobile. Modern desktop GPUs are designed to be flexible and powerful.

Mobile GPUs are designed for absolute performance efficiency. Relying on more advanced or secure features can be a problem, even if they are implemented on some hardware. Even the preferred way to write a shader may vary between mobile architectures. Simpler solutions are more likely to be supported by the optimisations of hardware vendors. Mobile hardware and APIs are evolving rapidly, and — so long as future-proofing is considered — knowing the most efficient approach may depend on monitoring the latest specifications.

16.6 Beauty fades — making it last

Implementing the highest quality graphics that can fit into current hardware can be good for advertising, but might not be so good for the user. Getting close to the ideal solution through approximation can be well worth a large performance gain. It can be short-sighted to stop optimising once the desired frame rate is reached: further optimisation leads to important power savings and avoid thermal shut-down that saps performance. If a slight reduction in image quality in the interests of power saving seems unpalatable, consider how dated any application looks when the next year's cutting edge software ships; however, a game that offers hours of game play will still have the same selling point several years later.

16.7 Working *with* the mobile screen

The very high resolution of mobile screens provides advantages to display quality and allows short-cuts to be made. Mobile screens are rarely viewed under perfect conditions. Even when they are, the resolution is high enough that the user's eyesight may not be able to make out every detail. Producing the per-pixel quality expected on a desktop system can be unnecessary on a mobile device — physical limitations can stop the viewer from perceiving everything that is rendered, so approximations can be made. Some display hardware already makes similar approximations with resolution in the interests of increased power consumption and perceived quality — approximations that can can be mirrored in

software for further benefits. The different means of interaction with mobile devices means that simpler graphics can be more suitable and preferable.

16.8 Offering the best experience

As graphics programmers and artists, it is natural to want to produce the highest possible image quality. Enjoyable though this can be, that is not actually what the customer is likely to want. Mobile gaming is not cinematography: spending less effort producing an affordable product and producing imagery that is "good enough" can be far more welcome, especially to the general populace less trained in spotting graphical perfection than this audience.

With the latest high-end mobile GPUs and efficient programming techniques, it is possible to achieve very high quality rendering effects. However, doing so may result in a phone which becomes uncomfortably warm to touch and which empties its battery in minutes. While users appreciate high quality graphics, they also appreciate more time using their devices. High-quality imagery does not have to mean photo-realism and complex shaders.

Finishing a SIGGRAPH course by telling the attendees not to produce the most immersive graphical experience that the hardware can achieve seems the wrong message. Instead, think of it this way: high quality graphics costs. It costs engineering time, it costs artist time, it costs power and it costs silicon area. Our job is to give the user a bargain. And we should all have a pay rise.