
SIGGRAPH2007

The Mobile 3D Ecosystem

SIGGRAPH2007

Kari Pulli	**Nokia Research Center**
Jani Vaarala	**Nokia**
Ville Miettinen	**NVIDIA**
Robert Simpson	**AMD**
Tomi Aarnio	**Nokia Research Center**
Mark Callow	**HI Corporation**

Today's program: Morning

- Start at 8:30
- Intro & OpenGL ES overview
  45 min, Kari Pulli
- Using OpenGL ES 1.x
  50 min, Jani Vaarala
- OpenGL ES on PyS60
  10 min, Kari Pulli
- Break 10:15 – 10:30
- OpenGL ES performance considerations
  45 min, Ville Miettinen
- OpenGL ES 2.0
  60 min, Robert Simpson
- N95 raffle
- Break 12:15

Today's program: Afternoon

- Start at 13:45
- M3G Intro
  10 min, Kari Pulli
- M3G API overview
  65 min, Tomi Aarnio
- M3G in the Real World 1
  30 min, Mark Callow
- Break 15:30 – 15:45
- M3G in the Real World 2
  60 min, Mark Callow
- M3G 2.0
  30 min, Tomi Aarnio
- Closing & Q&A
  15 min, Kari Pulli
- N95 raffle
- Finish at 17:30

Course Evaluations

http://www.siggraph.org/courses_evaluation

4 Random Individuals will win an ATI Radeon^{tm} HD2900XT AMD Smarter Choice

N95 raffle

2 Random Individuals will win a Nokia N95
- OpenGL ES 1.1 and M3G 1.1 HW
- 5 Megapixel camera
- GPS, mapping
- MP3 player
- W-LAN
- …

Put a card with name, affiliation, and feedback to a box, one phone is raffled in the morning session, one in the afternoon.

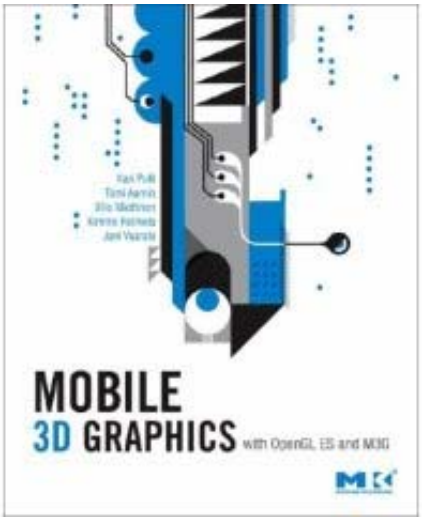
NOKIA
Connecting People

Special Issue of IEEE CGA: Mobile Graphics

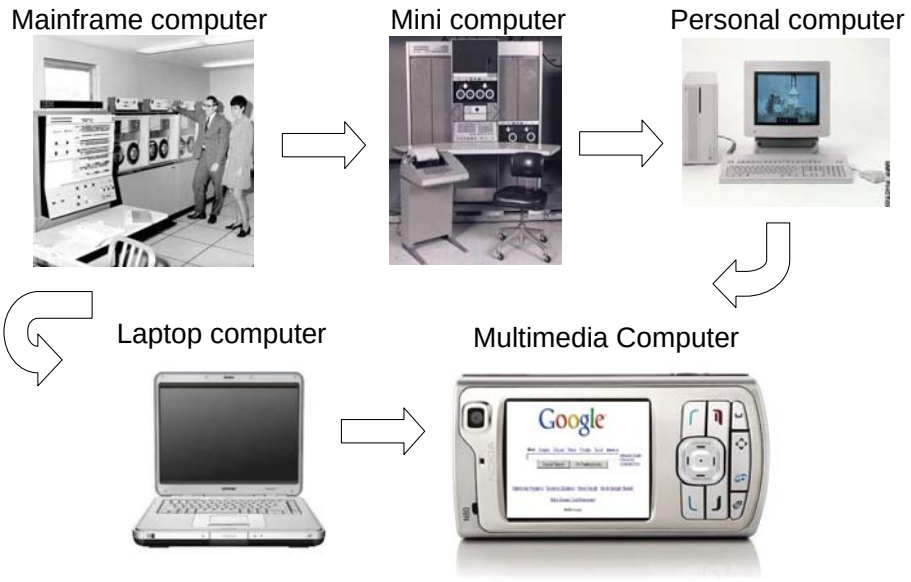
- The July-August 2008 issue, the abstracts are due Oct 31, 2007. We are looking for research or system papers in any areas of mobile graphics, including:
  - Mobile gaming
  - Graphics hardware
  - User interface toolkits and design tools
  - Augmented reality
  - Interaction design
  - Input techniques and technologies
  - Multimodal interfaces (such as speech, haptics, and sensors)
  - Programming languages and file formats (such as vector graphics)
  - Software architectures and service approaches
  - Visualization on small displays
  - Adapting multimedia to mobile clients
  - Techniques for rendering and browsing the Web
  - Design principles for information presentation
  - Research into particular domains (health care, developing world)

Mobile 3D Graphics: with OpenGL ES and M3G

- Book on mobile 3D APIs coming out soon
- Pulli, Vaarala, Miettinen, Aarnio, Roimela
- Can already pre-order from Amazon
- Morgan Kaufmann booth should have more information



Evolution of the Computer



The computers of the 50s and 60s were large enough to fill a room. The minicomputers of the 70s and 80s were still massive beasts.

The PC still takes over half of ones desk, laptops shrank them so they could be carried around in a bag. Finally, the high-end cell phones pack the same computation that you found a couple of years ago in a laptop into a form factor that fits your palm and pocket.

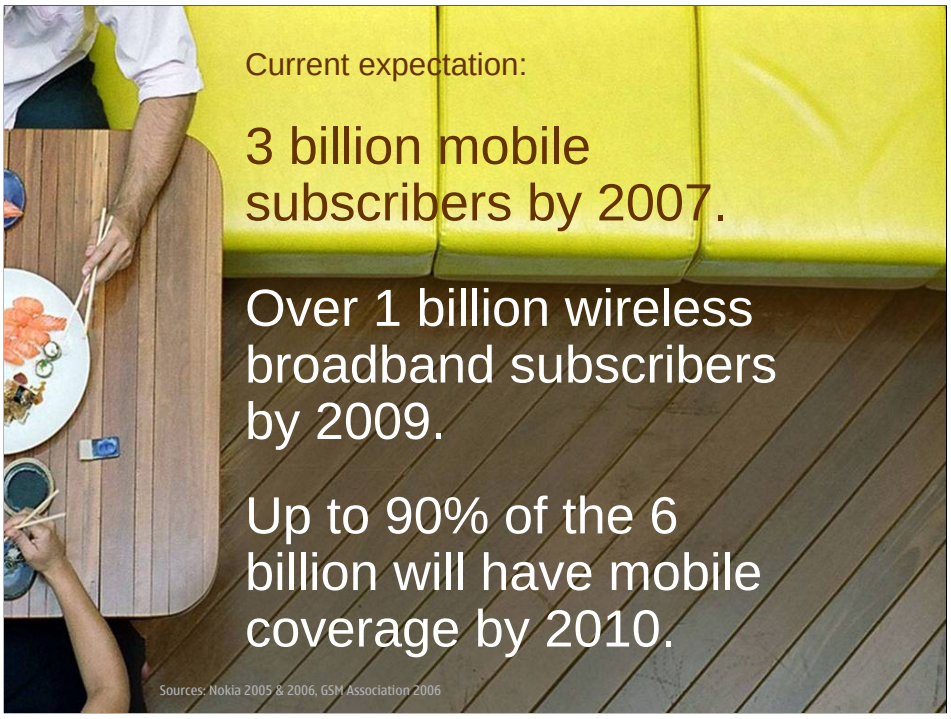
Pervasive Mobile Computing

- Mobile phones are the largest and fastest growing market - ever
  - The largest ever market opportunity for the graphics industry
- Handsets are becoming personal computing platform
  - Not “just” phones: A real computer in your hand
- Sophisticated media processing is a key
  - Just like it has been on the PC
  - Games are one of the first handheld media applications

These mobile handheld computers form the fastest growing computing platform, for graphics and many other technologies as well.

They are ubiquitous, most people have at least one.

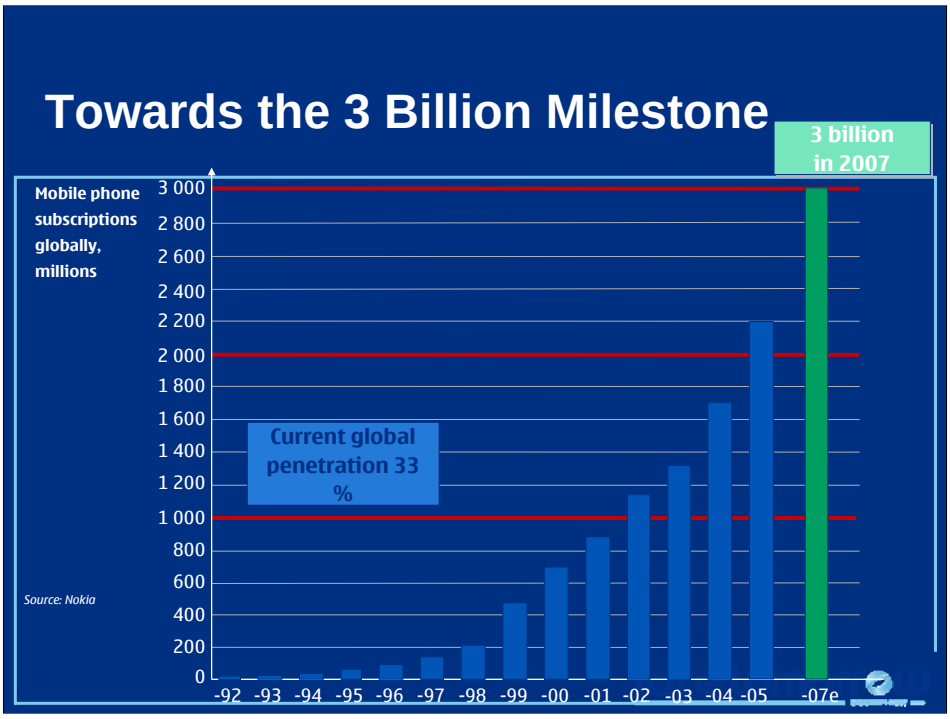
They are not “just” phones, but general personal computers, able to process many types of media such as audio, video, graphics, and imaging.



Here are some numbers to show how pervasive they are.

3 billion mobile subscribers globally this year, 1 billion wireless broadband subscribers in two more years,

and by 2010 there will be more people with mobile phones than there are people with a tooth brush today (~4 billion).



Here's the recent growth rate. Globally we're still in the exponential growth phase, though in the industrial world the growth is hitting the top end of the typical S-curve.

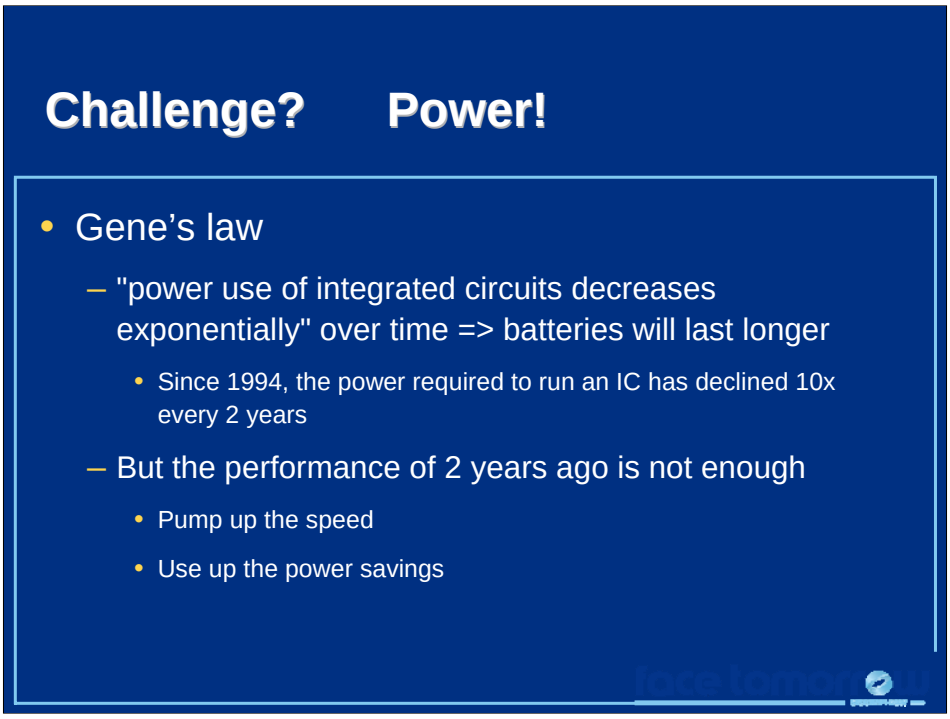
Challenge? Power!

- Power is the ultimate bottleneck
  - Usually not plugged to wall, just batteries
- Batteries don't follow Moore's law
  - Only 5-10% per year

Let's go through some of the challenges in mobile computing, contrasting to desktop computing.

The first, and most important one is power. Mobile devices usually operate on batteries, whereas PCs are usually directly connected to power grid. And people expect longer standup times from their phones than from their laptops.

The battery efficiency grows at a much slower rate than the general IC technology, which has followed Moore's law for the last four decades..

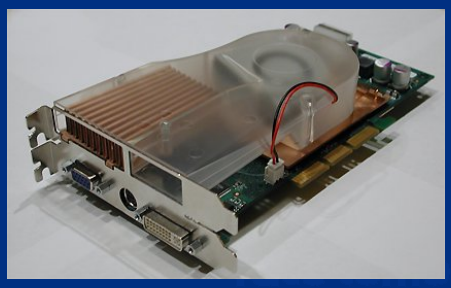


Luckily there is a corollary to the Moore's law, known as Gene's law, which says that as the chips get smaller, they use less power.

So with the same batteries you can get more output. But the expectations rise all the time, people expect more and faster of everything in newer models.

Challenge? Thermal mgt!

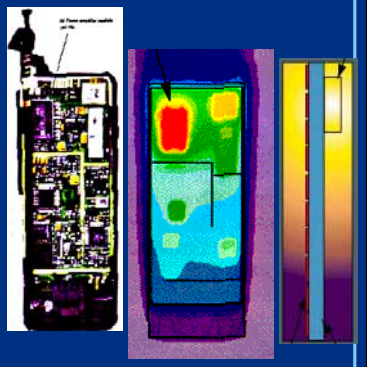
- But ridiculously good batteries still won't be the miracle cure
  - The devices are small
  - Generated power must get out
  - No room for fans



Another problem is due to the small physical size. Even if unlimited power was available, computation generates heat, which needs to be released. And as opposed to PC's there's no room for air or liquid cooling.

Challenge? Thermal mgt!

- Thermal management must be considered early in the design
  - Hot spot would fry electronics
    - Or at least inconvenience the user…
  - Conduct the heat through the walls, and finally release to the ambient



Instead the thermal management has to be taken into account early on in the design.

The pictures show first two pictures of a bad thermal design, where the radio creates a hot spot, that would prevent shrinking the device any further. Such a hot spot could fry up the electronics, or make it uncomfortable to handle or wear.

On the right we see another design (seen from the side) where the heat is much better distributed.

Let’s take a look at some of the key enablers for mobile graphics. Probably the most important, and most rapid development has happened in small LCD displays.

The resolution of mobile displays started small, the display of the small red phone in the image was around 48 x 84 pixels, black and white (or black and green).

But displays have improved in bounds and leaps, first driven by the demand from the digital camera displays. Now resolutions such as 320 x 240 with 16 bits of color or more are common.

Only the very cheapest phones nowadays have monochrome displays.

Some displays such as the one on the N800 internet tablet are quite a bit sharper than what is needed for typical broadcast TV.

Future? Displays!

- Physical size remains limited
  - TV-out connection
  - Near-eye displays?
  - Projectors?
  - Roll-up flexible displays?

allaboutsymbian.com

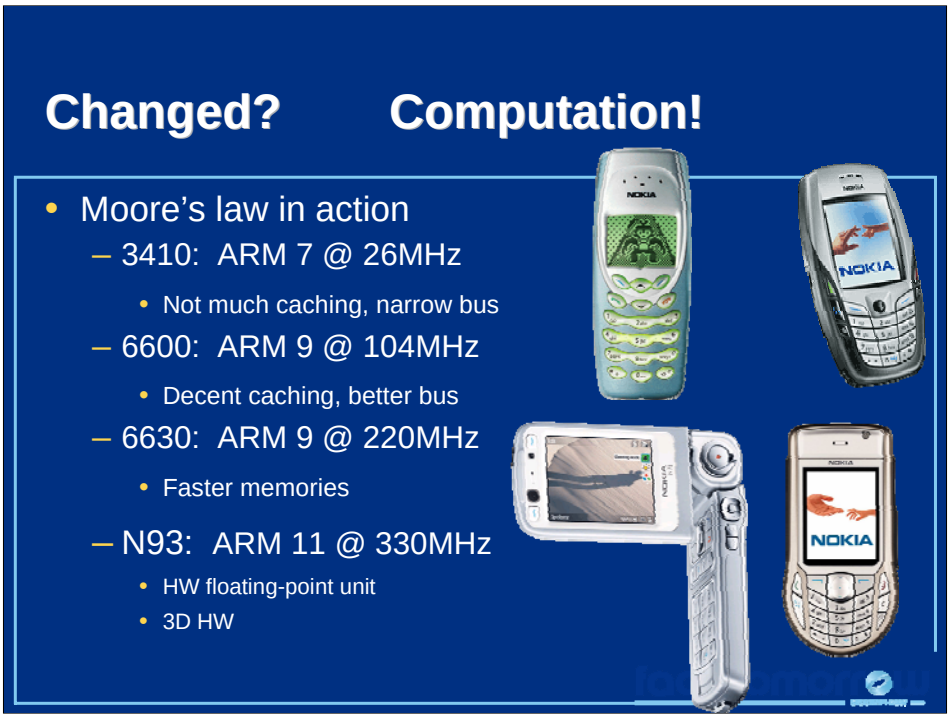
The physical size of the displays remains a challenge with devices that should be pocketable (assuming you don't want to take the route of growing the pocket sizes).

Some high-end models allow a TV-out connection, for example with N93 and N95 you can view your video clips and still images via a TV or projector at 640 x 480 resolution, even though the handset only has a 320 x 240 display.

Near-eye displays provide a potential for as high resolution displays as the human eye can handle, though in order to catch on, they need to get as light and sleek as in Mission Impossible (it should be easier without the built-in explosives).

With miniature lasers you can fit a projector into a small enough a form factor for cell phones, and perhaps even low enough power consumption.

Flexible displays that can be rolled up and expanded when needed are starting to come out of research laboratories.



Another key enabler is the increased amount of computation power that's available. Here we see Moore's law in action.

(Dates based on Aug 2007)

3410 launched about 5 years ago, coming with an ARM 7 CPU running at 26 MHz, and the rest of the architecture was pretty limited as well.

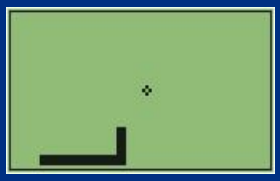
6600 came about 4 years ago, with a bigger processor, over 100 MHz clock, with better caches and bus.

6630 shipped almost 3 years ago, that doubled the clock speed, came with faster memories.

N93 shipped last year, again with much faster processing, and for the first time with a hardware floating point unit and 3D graphics hardware.

State-of-the-art in 2001: GSM world

- The world's most played electronic game?
  - According to The Guardian (May 2001)
- Communicator demo 2001
  - Remake of a 1994 Amiga demo
  - <10 year from PC to mobile

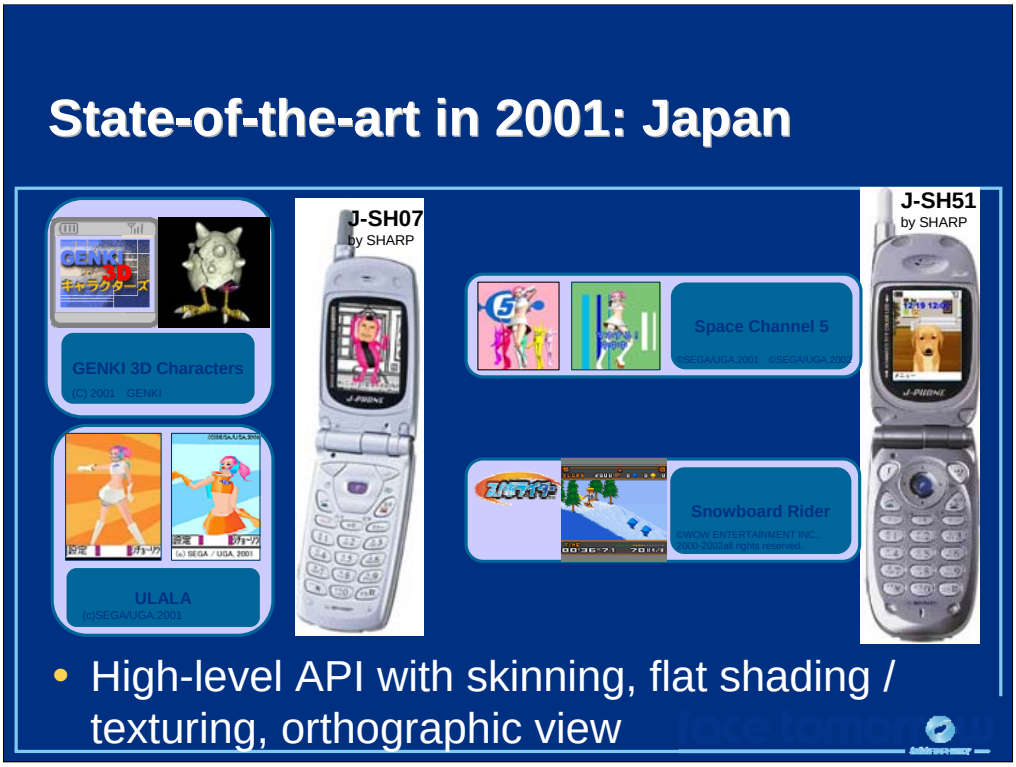


Let's take a look at the brief history of mobile graphics.

Around 2001, at least in Europe and Americas, the state of the art for mobile graphics was games such as Snake. Considering that in 2001 alone Nokia shipped over 100 million phones, most with Snake, with very few other games available, Snake is at least one of the most played electronic games ever. Then there were not many other mobile games to choose from, so a lot of people also played the game, whereas today there are so many choices that no single game gathers as much mindshare.

In 2001 an old Amiga demo was ported to Nokia communicator, causing a sensation at the Assembly demo competition organized yearly in Finland, and showing that you can, in fact, do better looking graphics than the Snake on handhelds.

State-of-the-art in 2001: Japan



- High-level API with skinning, flat shading / texturing, orthographic view

At the same time in Japan there were more devices with color displays, inviting the first commercial graphics applications.

The trend seems to be that the cool gadgets typically come first in the far east, then Europe, and finally in the Americas…

The first applications were simple games, mascots or screen savers, or “mascots” like the dog or the Ulala go-go girl.

The APIs were high-level APIs aimed for such applications.

The graphics engines were pretty modest in terms of features, featuring a only orthographic, or parallel projection cameras, flat shading and very simple if any lighting, and simple texture mapping.

State-of-the-art in 2002: GSM world

- 3410 shipped in May 2002
  - A SW engine: a subset of OpenGL including full perspective (even textures)
  - 3D screensavers (artist created content)
  - FlyText screensaver (end-user content)
  - a 3D game

In 2002 the first GSM phone with a 3D graphics engine shipped. It didn't have a nice color screen, but a 1-bit 96x65 display.

State-of-the-art in 2002: Japan

- Gouraud shading, semi-transparency, environment maps

I-3D PolyGame Boxing

@ Hi Vanguard : REZO, BNW

Ulala Channel J

©SEGA/UGA_2001 ©SEGA/UGA_2002

KDDI Au 3D Launcher

©SAN-X+GREEN CAMEL

3d menu

State-of-the-art in 2003: GSM world

- N-Gage ships
- Lots of proprietary 3D engines on various Series 60 phones

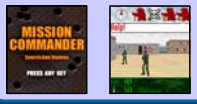
Fathammer's Geopod on XForge

State-of-the-art in 2003: Japan

- Perspective view, low-level API

Ridge Racer

@ Namco



Mission Commander

Multi player Fps Game

© IT Telecom

Mobile 3D in 2004

- 6630 shipped late 2004
  - First device to have both OpenGL ES 1.0 (for C++) and M3G (a.k.a JSR-184, for Java) APIs
- Sharp V602SH in May 2004
  - OpenGL ES 1.0 capable HW but API not exposed
  - Java / MascotCapsule API

2005 and beyond: HW

Mobile graphics evolution snapshot

Spider-Man 2: The Hero Returns
Sony Pictures

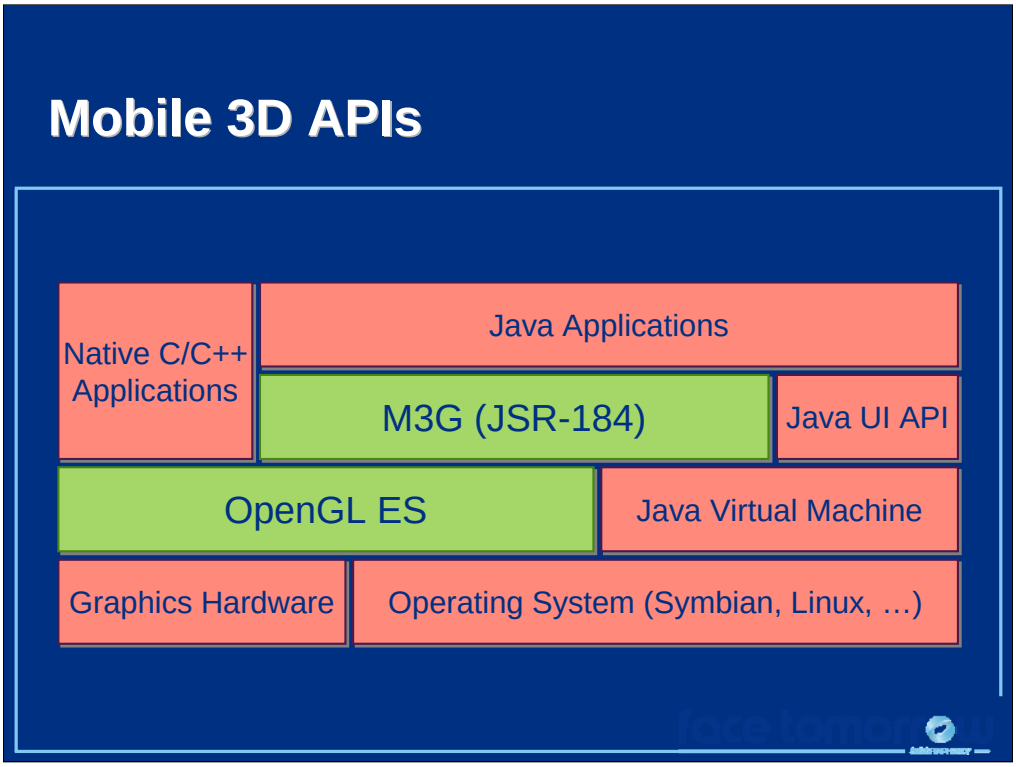
2D

Spider-Man 2 3D: NY Subway
Sony Pictures

Software 3D

Spider-Man 2
Activision

Accelerated 3D



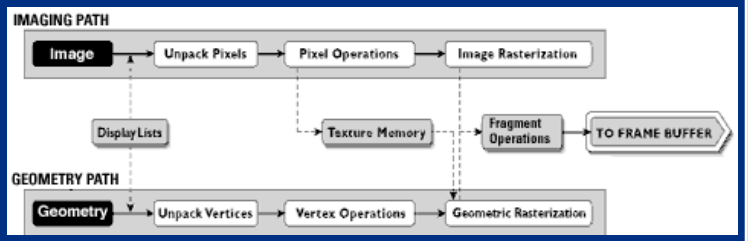
The green parts show the content of today's course. We will cover two mobile 3D APIs, used by applications, either the so-called native C/C++ applications, or Java midlets (the mobile versions of applets). The APIs use system resources such as memory, display, and graphics hardware if available. OpenGL ES is a low-level API, that can be used as a building block for higher level APIs such as M3G, or Mobile 3D Graphics API for J2ME, also known as JSR-184 (JSR = Java Standardization Request).

Overview: OpenGL ES

- Background: OpenGL & OpenGL ES
- OpenGL ES 1.0
- OpenGL ES 1.1
- EGL: the glue between OS and OpenGL ES
- How can I get it and learn more?

What is OpenGL?

- The most widely adopted graphics standard
  - most OS's, thousands of applications
- Map the graphics process into a pipeline
  - matches HW well



modeling
⇩
projecting
⇩
clipping
⇩
lighting & shading
⇩
texturing
⇩
hidden surface
⇩
blending
⇩
pixels to screen

- A foundation for higher level APIs
  - Open Inventor; VRML / X3D; Java3D; game engines

What is OpenGL ES?

- OpenGL is just too big for Embedded Systems with limited resources
  - memory footprint, floating point HW
- Create a new, compact API
  - mostly a subset of OpenGL
  - that can still do almost all OpenGL can

OpenGL ES 1.0 design targets

- Preserve OpenGL structure
- Eliminate un-needed functionality
  - redundant / expensive / unused
- Keep it compact and efficient
  - <= 50KB footprint possible, without HW FPU
- Enable innovation
  - allow extensions, harmonize them
- Align with other mobile 3D APIs (M3G / JSR-184)

Adoption

- Symbian OS, S60
- Brew
- PS3 / Cell architecture

Sony's arguments: Why ES over OpenGL

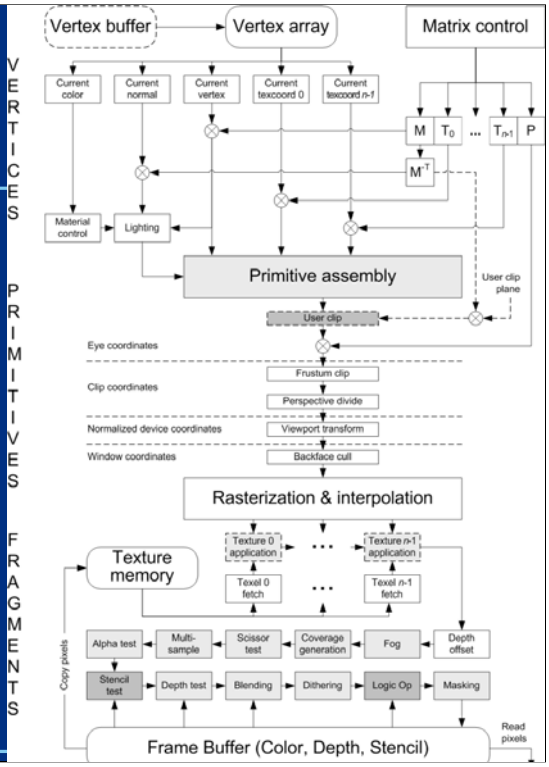
- OpenGL drivers contain many features not needed by game developers
- ES designed primarily for interactive 3D app devs
- Smaller memory footprint

Outline

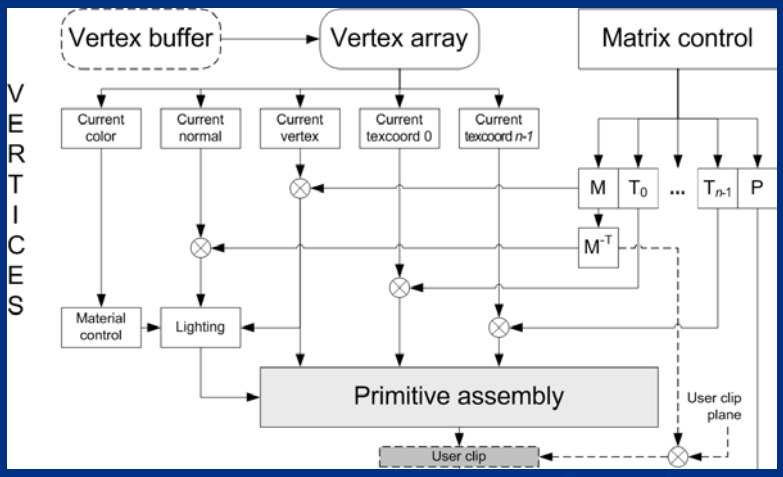
- Background: OpenGL & OpenGL ES
- OpenGL ES 1.0
- OpenGL ES 1.1
- EGL: the glue between OS and OpenGL ES
- How can I get it and learn more?

OpenGL ES Pipe

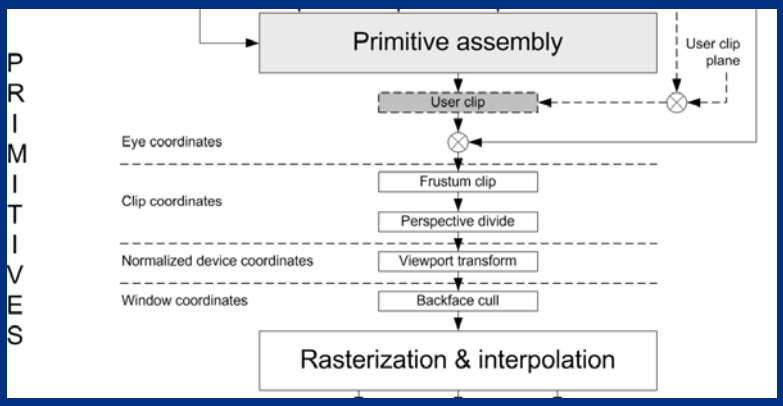
- Here's the OpenGL ES pipeline stages
  - vertices
  - primitives
  - fragments



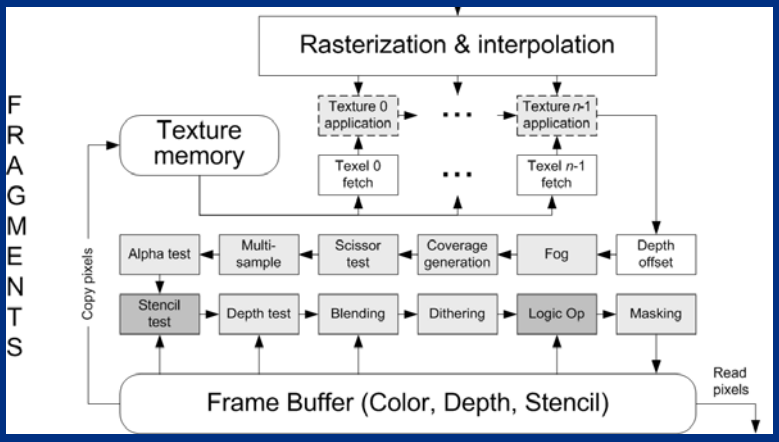
Vertex pipeline



Primitive processing



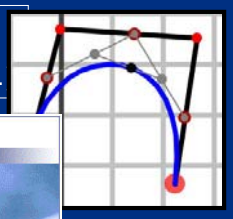
Fragment pipeline

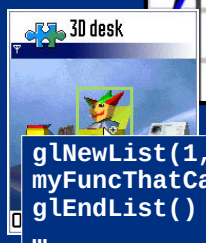


Functionality: in / out? (1/7)

- Convenience functionality is OUT
  - GLU
    (utility library)
  - evaluators
    (for splines)
  - feedback mode
    (tell what would draw without drawing)
  - selection mode
    (for picking, easily emulated)
  - display lists
    (collecting and preprocessing commands)

```
gluOrtho2D(0,1,0,1)
vs.
glOrtho(0,1,0,1,-1,1
```






```
glNewList(1, GL_COMPILE)
myFuncThatCallsOpenGL()
glEndList()
…
glCallList(1)
```

Functionality: in / out? (2/7)

- Remove old complex functionality
  - glBegin – glEnd (**OUT**); vertex arrays (**IN**)
  - new: coordinates can be given as bytes

```
glBegin(GL_POLYGON);
glColor3f (1, 0, 0);
glVertex3f(-.5, -.5, .5);
glVertex3f( .5, .5, .5);
glColor3f (0, 1, 0);
glVertex3f( .5,-.5, .5);
glVertex3f(-.5, .5, .5);
glEnd();
```

```
static const GLbyte verts[4 * 3] =
{    -1,  1,  1,      1,  1,  1,
      1, -1,  1,     -1, -1,  1 };
static const GLubyte colors[4 * 3] =
{   255,  0,  0,    255,  0,  0,
      0,255,  0,      0,255,  0 };
glVertexPointer( 3,GL_BYTE,0, verts );
glColorPointerf( 3,GL_UNSIGNED_BYTE,
                 0, colors );
glDrawArrays( GL_TRIANGLE_STRIP,
              0, 4 );
```

Functionality: in / out? (3/7)

- Simplify rendering modes
  - double buffering, RGBA, no front buffer access
- Emulating back-end missing functionality is expensive or impossible
  - full fragment processing is **IN**

    alpha / depth / scissor / stencil tests,
    multisampling,
    dithering, blending, logic ops)

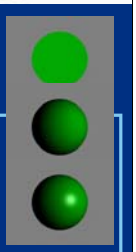
Functionality: in / out? (4/7)

- Raster processing
  - ReadPixels **IN**, DrawPixels and Bitmap **OUT**
- Rasterization
  - **OUT:** PolygonMode, PolygonSmooth, Stipple

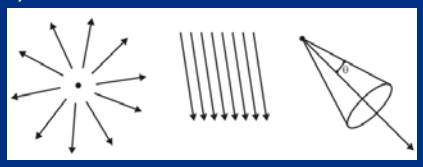
Functionality: in / out? (5/7)

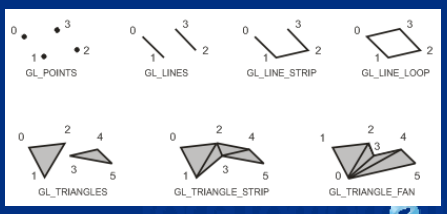
- 2D texture maps **IN**
  - 1D, 3D, cube maps **OUT**
  - borders, proxies, priorities, LOD clamps **OUT**
  - multitexturing, texture compression **IN** (optional)
  - texture filtering (incl. mipmaps) **IN**
  - new: paletted textures **IN**

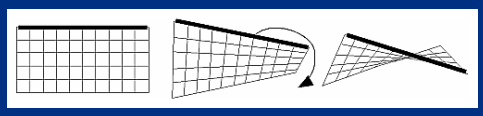
Functionality: in / out? (6/7)



- Almost full OpenGL light model **IN**
  - back materials, local viewer, separate specular **OUT**
- Primitives
  - **IN:** points, lines, triangles
  - **OUT:** quads & polygons







Functionality: in / out? (7/7)

- Vertex processing
  - **IN:** transformations
  - **OUT:** user clip planes, texcoord generation
- Support only static queries
  - **OUT:** dynamic queries, attribute stacks
    - application can usually keep track of its own state

Floats vs. fixed-point

- Accommodate both
  - integers / fixed-point numbers for efficiency
  - floats for ease-of-use and being future-proof
- Details
  - 16.16 fixed-point: add a decimal point inside an int

```
glRotatef( 0.5f,    0.f   , 1.f,    0.f    );
                    vs.
glRotatex( 1 << 15,    0   , 1 << 16,    0    );
```

  - get rid of doubles

Outline

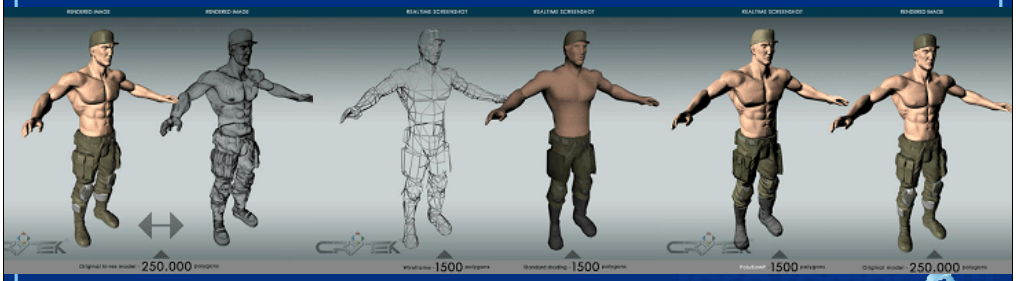
- Background: OpenGL & OpenGL ES
- OpenGL ES 1.0
- OpenGL ES 1.1
- EGL: the glue between OS and OpenGL ES
- How can I get it and learn more?

OpenGL ES 1.1: core

- Buffer Objects
  - allow caching vertex data
- Better Textures
  - >= 2 tex units, combine (+,-,interp), dot3 bumps, auto mipmap gen.
- User Clip Planes
  - portal culling (>= 1)
- Point Sprites
  - particles as points not quads, attenuate size with distance
- State Queries
  - enables state save / restore, good for middleware

Bump maps

- Double win
  - increase realism
  - reduce internal bandwidth -> increase performance



OpenGL ES 1.1: optional

- Draw Texture
  - fast drawing of pixel rectangles
  - using texturing units
  - (data can be cached),
  - constant Z, scaling
- Matrix Palette
  - vertex skinning
  - (>= 3 matrices / vertex, palette >= 9)

Outline

- Background: OpenGL & OpenGL ES
- OpenGL ES 1.0
- OpenGL ES 1.1
- EGL: the glue between OS and OpenGL ES
- How can I get it and learn more?

EGL glues OpenGL ES to OS

- EGL is the interface between OpenGL ES and the native platform window system
  - similar to GLX on X-windows, WGL on Windows
  - facilitates portability across OS's (Symbian, Linux, …)
- Division of labor
  - EGL gets the resources (windows, etc.) and displays the images created by OpenGL ES
  - OpenGL ES uses resources for 3D graphics

EGL surfaces

- Various drawing surfaces, rendering targets
  - *windows* – on-screen rendering ("graphics" memory)
  - *pbuffers* – off-screen rendering (user memory)
  - *pixmaps* – off-screen rendering (OS native images)

EGL context

- A rendering context is an abstract OpenGL ES state machine
  - stores the state of the graphics engine
  - can be (re)bound to any matching surface
  - different contexts can share data
    - texture objects
    - vertex buffer objects
    - even across APIs (OpenGL ES, OpenVG, later others too)

Main EGL 1.0 functions

- Getting started
  - eglInitialize() / eglTerminate(), eglGetDisplay(), eglGetConfigs() / eglChooseConfig(), eglCreate**X**Surface() (**X** = Window | Pbuffer | Pixmap), eglCreateContext()
- eglMakeCurrent(display, drawsurf, readsurf, context)
  - binds context to current thread, surfaces, display

Main EGL 1.0 functions

- eglSwapBuffer(display, surface)
  - posts the color buffer to a window
- eglWaitGL( ), eglWaitNative(engine)
  - provides synchronization between OpenGL ES and native (2D) graphics libraries
- eglCopyBuffer(display, surface, target)
  - copy color buffer to a native color pixmap

EGL 1.1 enhancements

- Swap interval control
  - specify # of video frames between buffer swaps
  - default 1; 0 = unlocked swaps, >1 save power
- Power management events
  - PowerMgmnt event => all Context lost
  - Display & Surf remain, Surf contents unspecified
- Render-to-texture [optional]
  - flexible use of texture memory

Outline

- Background: OpenGL & OpenGL ES
- OpenGL ES 1.0 functionality
- OpenGL ES beyond 1.0
- EGL: the glue between OS and OpenGL ES
- How can I get it and learn more?

SW Implementations

- Gerbera from Hybrid
  - Free for non-commercial use
  - http://www.hybrid.fi
- Vincent
  - Open-source OpenGL ES library
  - http://sourceforge.net/projects/ogl-es
- Reference implementation
  - Wraps on top of OpenGL
  - http://www.khronos.org/opengles/documentation/gles-1.0c.tgz

HW implementations

- There are many designs
- The following slides gives some idea
  - rough rules of thumb
    - 1-5 M Tri / sec
    - 1 pixel / clock
    - clock speeds 50MHz – 200+MHz
    - power consumption should be < 100 mW

Bitboys

- Graphics processors
  - **G12:** OpenVG 1.0
  - **G34:** OpenGL ES 1.1
  - **G40:**
  - Flipquad antialiasing
  - Max clock 200MHz
- Partners / Customers
  - NEC Electronics
  - Hybrid Graphics (drivers)

ATI bought BB


ATI
ATI IMAGEON 3D
Imageon 2300
OpenGL ES 1.0
Vertex and raster HW
32-bit internal pipe
16-bit color and Z buffers
AMD bought ATI
Imageon 3D adds
OpenGL ES 1.1 extension pack
Vertex shader
HyperZ
Audio codecs, 3D audio
Partners, customers
Qualcomm
LG SV360, KV3600
Zodiac
ATI
IMAGEON 3D

AMD Handheld Graphics

AMD Smarter Choice

1st generation (Imageon 2300)

- OpenGL ES 1.0 (1st conformant implementation)
- Vertex and raster HW
- 32-bit internal pipe, 16-bit color and Z buffers
- Integrated QVGA buffer
- Imaging / Video codecs
- 1 Mtri/s, 100 Mpix/s

2nd generation (Imageon 2380)

- OpenGL ES 1.1
- Vertex shader, HyperZ
- Audio codecs, 3D audio
- 3.5 Mtri/s, 125 Mpix/s

3rd generation (to be announced)

- OpenGL ES 2.0
- Full HW OpenVG 1.1
- Unified Shaders
- OpenGL ES 2.0 and OpenVG cores are also available as IP

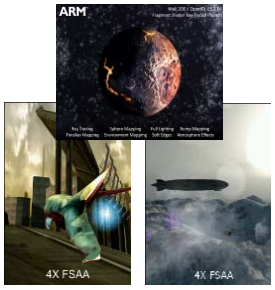
Falanx

- **Mali 110**
  - OpenGL ES 1.1 + extensions
  - 4x / 16x full screen anti-aliasing
  - Video codecs (e.g., MPEG-4)
  - 170-40[illegible]
  - 2.8M [illegible]
- **Mali 20[illegible]**
  - Open[illegible] Mob.
  - 5M Tri / s, 100M Pix / s, 11 instr. / cycle
- **Partners / Customer**
  - Zoran

ARM bought Falanx

ARM® Mali™ Architecture

- Compared to traditional immediate mode renderer
  - 80% lower per pixel bandwidth usage, even with 4X FSAA enabled
  - Efficient memory access patterns and data locality: enables performance even in high latency systems
- Compared to traditional tile-based renderer
  - Significantly lower per-vertex bandwidth
  - Impact of scene complexity increases is substantially reduced
- Other architectural advantages
  - Per frame autonomous rendering
  - No renderer state change performance penalty
  - On-chip z / stencil / color buffers
    - minimizes working memory footprint
- Acceleration beyond 3D graphics (OpenVG etc.)



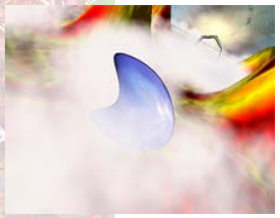
	Mali200	MaliGP2	Mali55
Anti-Aliasing	4X / 16X	4X / 16X	4X / 16X
OpenGL®ES 1.x	YES	YES	YES
OpenGL®ES 2.x	YES	YES	NO
OpenVG 1.x	YES	NA	YES
Max CLK	275MHz	275MHz	200MHz
Fill rate Mpix / s	275	NA	100
Triangles / s	9M	9M	1M

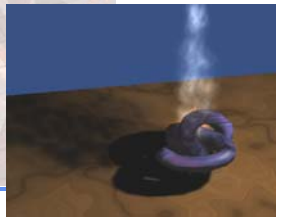
DMP Inc.

Visualize the future

- **PICA200 graphics core** PICA
  - 3D Features
    - OpenGLES 1.1
    - DMP's proprietary "Maestro" extensions
      - **Very high quality graphics with easier programming interface**

MAESTRO
      - **Per-fragment lighting,**
      - **Shadow-mapping,**
      - **Procedural texture,**
      - **Polygon subdivision (Geo shader), and**
      - **Gaseous object rendering.**
  - Hardware Features
    - » **Performance: 20Mtri/s, 400Mpixel/s@100MHz**
    - » **Core size: 500Kgate – 4Mgate**
    - » **Power consumption: 0.5-2mW/MHz**
    - » **Max. clock freq. 200MHz (90nm and 130nm)**





Imagination Technologies
POWERVR MBX & SGX 2D/3D Acceleration

Imagination

- **5th Generation Tile Based Deferred Rendering**
  - Market Proven Advanced Tiling Algorithms
  - Order-independent Hidden Surface Removal
  - Lowest silicon area, bandwidth and power
  - Excellent system latency tolerance
- **POWERVR SGX: OpenGL ES 2.0 in Silicon Now**
  - Scalable from 1 to 8 pipelines and beyond
  - Programmable multi-threaded multimedia GPU
  - Optimal load balancing scheduling hardware
  - Vertex, Pixel, Geometry shaders + image processing
- **Partners/Customers**
  - TI, Intel, Renesas, Samsung, NXP, NEC, Freescale, Sunplus, Centrality & others unannounced

www.powervrinsider.com

Market-leading Ecosystem with more than 1650 members

POWERVR MBX: The de-facto standard for mobile graphics acceleration, with >50 PowerVR 3D-enabled phones shipping worldwide

	PowerVR MBX Family	PowerVR SGX Family
OpenGL	ES1.1	2.0, ES1.1 and ES2.0
Direct3D	Mobile	Mobile, 9L and 10.1
OpenVG	1.0	1.0.1 and 1.1
Triangles/Sec	1.7M … 3.7M	1M … 15.5M
Pixels/Sec	135M … 300M	50M … 500M

Performance quoted at 100MHz for MBX, MBX Lite and for SGX510 to SGX545.
Peak SoC achievable performance not quoted, e.g. <50% Shader load for Tri/Sec.
Performance scales with clock speeds up to 200MHz and beyond.
Planned future cores will offer higher performance levels.

MITSUBISHI ELECTRIC

Changes for the Better

Mitsubishi

- Z3D family
  - Z3D and Z3D2 out in 2002, 2003
    - Pre-OpenGL ES 1.0
    - Embedded SRAM architecture
  - Z3D3 in 2004
    - OpenGL ES 1.0, raster and vertex HW
    - Cache architecture
    - @ 100 MHz: 1.5M vtx / s, 50-60 mW, ~250 kGates
  - Z3D4 in 2005
    - OpenGL ES 1.1
- Partners / Customers
  - Several Japanese manufacturers

Z3D

First mobile 3D HW?

Graphics Engine

GiPump™ Series

GiPump™ NX1005
; Mobile 3D graphics acc. with camera control functions
- OpenGL ES 1.1 / GIGA / JSR184
- 5M poly/s, 80M pix/s @ 80MHz, JPEG codec (3M pixel), ~QVGA display
- Cellular phone, smart phone, etc.

GiPump™ NX1007
; High end 3D graphics acc. for mobile
- OpenGL ES 1.1 + Ext. / GIGA / JSR184
- 12.5M poly/s, 200M pix/s @ 100MHz, ~SVGA display, PIP supports
- PND, PMP, game device, mobile device, etc.

GiPump™ NX1008
; Mobile 3D graphics acc. with stereoscopic display
- OpenGL ES 1.1 / GIGA / JSR184
- 5M poly/s, 80M pix/s @ 80MHz, ~QVGA display, stereoscopic display
- Cellular phone, smart phone, etc.

GiPump™ NX1009
; Economical mobile 3D graphics accelerator
- OpenGL ES 1.1 + Ext. / GIGA / JSR184
- 12.5M poly/s, 200M pix/s @ 100MHz, ~SVGA display, boost mode
- Cellular phone, Smart phone, etc.

GiPump™ NX2001
; 3D Graphics enhanced multimedia processor
- OpenGL ES 2.0 / 1.1 Ext. / JSR184 / D3DM
- 10M poly/s, 200M pix/s @ 200MHz, ~SVGA display
- PND, PMP, game device, mobile device, etc.

Service Solutions

Nexus Mobile Platform™
Gaming Device Platform
(OS: WinCE, Linux, RTOS, etc.)
To: Game Device Maker

NX1008TK™
3D Reference B/D
GiPump™ Integration Platform
To: Device Developer

GiPump™ SDK
NXsdk with Emulator
NXsdk Shader+
NXm3g Engine
NX3D Engine & Tools

GiPump™ Partners : Samsung, SKT, Other Device Manufactures

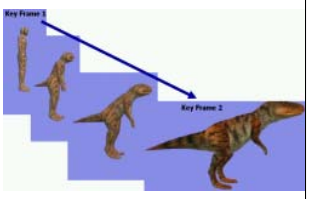
* GiPump™ : Pronounced, "G", "I", "Pump". It means "Graphics / Image Pump".
* GIGA (Giga Instruction Giga Acceleration) : SK Telecom's mobile 3D graphics platform

NVidia

- GoForce 5500 handheld GPU
  - 3D geometry and rasterization HW
  - OpenGL ES 1.1, D3D Mobile, OpenVG 1.0
  - 1.3M tri / s, 100M pix / s (@ 100 MHz)
  - Programmable pixel micro shaders
  - 40 bit signed non-int (overbright) color pipeline
  - Dedicated 2D engine (bitblt, lines, alpha blend)
  - Supersampled anti-aliasing, up to 6 textures
  - <50mW avg. dynamic power cons. for graphics
  - 10MPxl camera support, XGA LCD, MPEG-4 video, audio
- Partners / Customers
  - Motorola, Sony Ericsson, Samsung, LG, Kyocera, O2, HTC, Marvell, Freescale, …

Sony PSP

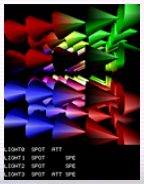
- Game processing unit
  - Surface engine
    - tessellation of Beziers and splines
    - skinning (<= 8 matrices), morphing (<= 8 vtx
    - HW T&L
    - 21 MTri / s (@ 100 MHz)
  - Rendering engine
    - basic OpenGL-style fixed pipeline
    - 400M pix / s (@ 100 MHz)
  - 2MB eDRAM
- Media processing engine
  - 2MB eDRAM
  - H.264 (AVC) video up to 720x480 @ 30fps



TAKUMI

- **GSHARK-TAKUMI Family**
  - **GP**
    - OpenGL ES 1.0
    - 0.5M tri/s @100MHz, 170Kgate
  - **GT**
    - OpenGL ES 1.1
    - 1.4M tri/s @100MHz, < 30mW
  - **G2**
    - OpenGL ES 1.1
    - 5M tri/s @100MHz
- Partners / Customers
  - NEC Electronics

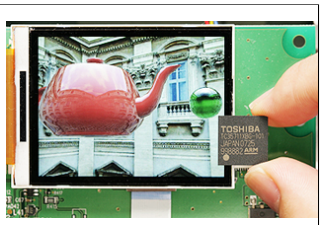
- Concepts & Architecture
  - Small Gate Counts
  - Low Power Consumption
  - Vertex Processor (T&L)
  - Dedicated 2D Sprite Engine
  - Target Application
    - Mobile Phone and Digital AV Equipments such as DTV, STB, DSC, PMP, etc.

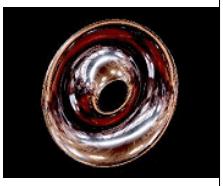


TAKUMI CORPORATION

Toshiba

- TC35711XBG
  - Programmable shader
  - Plan to support OpenGL ES2.0
  - Large embedded memory for
    - Color and Z buffer
    - Caches for vertex arrays, textures
    - Display lists (command buffer)
  - 50M vtx / sec, 400M pix / sec (@ 100 MHz)
  - WVGA LCD controller
  - 13mm x 13mm x 1.2mm 449Ball BGA





Vivante GPU for Handheld

- OpenGL ES 1.1 & 2.0 and D3D 9.0
- Unified vertex & pixel shader
- Anti-Aliasing
- AXI/AHB interface
- GC500
  - 3 mm^2 die area in 65nm (1.8mm x 1.2mm)
  - 10 MPolygons/s and 100 MPixel/s at 200 MHz
  - 50mW GPU core power
- Scalable solution to 50 MPolygons/s and 1 GPixels/s (GC1000, GC4000)
- **Silicon proven solution**
- Designed into multiple 65nm SoCs

VIVANTE

SDKs

- Nokia S60 SDK (Symbian OS)
  - http://www.forum.nokia.com
- Imagination SDK
  - http://www.pvrdev.com/Pub/MBX
- NVIDIA handheld SDK
  - http://www.nvidia.com/object/hhsdk_home.html
- Brew SDK & documentation
  - http://brew.qualcomm.com

OpenGL ES 1.1 Demos

Questions?


SIGGRAPH2007

Using OpenGL ES

Jani Vaarala

Nokia

Using OpenGL ES

- Simple OpenGL ES example
- EGL configuration selection
- Texture matrix example
- Fixed point programming
- Converting existing code

“Hello OpenGL ES”

-This is what we are aiming for: single smooth shaded triangle

Hello OpenGL ES, EGL initialization

```
/* ================================================================
 *  "Hello OpenGL ES" OpenGL ES code.
 *
 *  Siggraph 2007 course on mobile graphics.
 *
 *  Copyright: Jani Vaarala
 * ================================================================
*/

#include <GLES/gl.h>
#include <GLES/egl.h>

EGLDisplay      display;
EGLContext      context;
EGLSurface      surface;
EGLConfig       config;
```

-Here are the headers and basic EGL variables that we will be using

Hello OpenGL ES, EGL initialization

```
EGLint attrib_list[] =
{
    EGL_BUFFER_SIZE,   16,
    EGL_DEPTH_SIZE,    15,
    EGL_SURFACE_TYPE,   EGL_WINDOW_BIT,
    EGL_NONE
};

void init_egl(void)
{
    EGLint numOfConfigs;

    display = eglGetDisplay( EGL_DEFAULT_DISPLAY );
    eglInitialize( display, NULL, NULL );
    eglChooseConfig( display, attrib_list, &config, 1 , &numOfConfigs );
    surface = eglCreateWindowSurface( display, config, WINDOW( ), NULL );
    context = eglCreateContext( display, config, EGL_NO_CONTEXT, NULL );
    eglMakeCurrent( display, surface, surface, context );
}
```

-attrib_list defines attributes for the configuration that we want to use: at least 15 bits in depth buffer and 16 bits in color buffer

-WINDOW( ) is a macro that is used here to indicate the place where the windowing system specific window type goes into

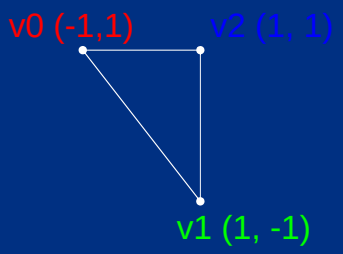
-Basic EGL initialization can be done like this, but in real-world applications also the error checking should be included and config selection may be a little bit more involved

Hello OpenGL ES, OpenGL ES part

```
#include <GLES/gl.h>

static const GLbyte vertices[3 * 3] =
{
    -1,   1,   0,
     1,  -1,   0,
     1,   1,   0
};

static const GLubyte colors[3 * 4] =
{
        255,   0,     0,     255,
        0,     255,   0,     255,
        0,     0,     255,   255
};
```



-Each vertex has different color (full R, full G, full B).

Hello OpenGL ES, OpenGL ES part

```
void init( )
{
    glClearColor          ( 0.f, 0.f, 0.1f, 1.f );
    glMatrixMode          ( GL_PROJECTION );
    glFrustumf            ( -1.f, 1.f, -1.f, 1.f, 3.f, 1000.f );
    glMatrixMode          ( GL_MODELVIEW );
    glShadeModel          ( GL_SMOOTH );
    glDisable             ( GL_DEPTH_TEST );
    glVertexPointer       ( 3, GL_BYTE, 0, vertices );
    glColorPointer        ( 4, GL_UNSIGNED_BYTE, 0, colors );
    glEnableClientState   ( GL_VERTEX_ARRAY );
    glEnableClientState   ( GL_COLOR_ARRAY );
    glViewport            ( 0, 0, GET_WIDTH(), GET_HEIGHT() );

    INIT_RENDER_CALLBACK(drawcallback);
}
```

-OpenGL ES setup code, sets up a vertex array and a color array

-INIT_RENDER_CALLBACK is a platform specific way to set up a render callback to function drawcallback (called by timer for example)

Hello OpenGL ES, OpenGL ES part

```
void drawcallback(void)
{
   glClear          ( GL_COLOR_BUFFER_BIT );
   glLoadIdentity   ( );
   glTranslatef     ( 0.f, 0.f, -5.f );
   glDrawArrays     ( GL_TRIANGLES, 0, 3 );

   eglSwapBuffers( display, surface );
}
```

- This is the render callback. We just clear the color buffer, translate camera a bit and draw a triangle.
- Finally we call eglSwapBuffers( ) to copy the surface to the display

EGL config sorting

ATTRIBUTE	DEFAULT VALUE	SELECTION RULE	SORT PRIORITY	SORT ORDER
EGL_BUFFER_SIZE [16]	0	AtLeast	3	Smaller
EGL_DEPTH_SIZE [15]	0	AtLeast	6	Smaller
...				

- Selection rule: minimum requirement
- Sort priority: which attrib is sorted first
- Sort order: how attrib is sorted
- One way of sorting
- Not optimal for all applications

-EGL config sorting is quite complex

-Still it does not support all cases that the application might want

Example of sorted list of configs

EGL_CONFIG_ID	EGL_BUFFER_SIZE (Sort priority = 3)	EGL_DEPTH_SIZE (Sort priority = 6)
5	16	15
2	16	32
40	24	15
11	32	15
3	32	32
30	32	32

Sorted first, smaller comes first

Sorted next, smaller comes first

Sorted last (if otherwise no unique order exists), smaller comes first

Example EGL config selection

```
EGLConfig select_config(int type, int color_bits, int depth_bits, int stencil_bits)
{
    EGLBoolean      err;
    EGLint          amount, attrib_list[5*2];   /* fits 5 attribs */
    EGLConfig       best_config, configs[64]; /* max 64 configs considered */
    EGLint *ptr;

    ptr = &attrib_list[0];

    /* Make sure that the config supports target surface type */
    *ptr++ = EGL_SURFACE_TYPE;
    *ptr++ = type;

    /* For color, we require minimum of <color_bits> bits */
    *ptr++ = EGL_BUFFER_SIZE;
    *ptr++ = color_bits;

    /* For depth, we require minimum of <depth_bits> bits */
    if(depth_bits)
    {
        *ptr++ = EGL_DEPTH_SIZE;
        *ptr++ = depth_bits;
    }
```

-Here is an example showing how the configuration selection might work in real world

-First we fill the attrib list with our required attributes

Real-world EGL config selection

```
if(stencil_bits)
{
        ptr[0] = EGL_STENCIL_SIZE;
        ptr[1] = stencil_bits;
        ptr[2] = EGL_NONE;
}
else
{
        ptr[0] = EGL_NONE;
}

err = eglChooseConfig( display, &attrib_list[0], &configs[0], 64, &amount );

if(amount == 0)
{
        /* If we didn't have get any configs, try without stencil */
        ptr[0] = EGL_NONE;
        err = eglChooseConfig( display, &attrib_list[0], &configs[0], 64, &amount );
}
```

-If there is a stencil requirement, we try to get config that has stencil

-If there is no config that matches stencil requirement, we try without (e.g., app turns off stencil shadows because of no stencil support)

Real-world EGL config selection

```
    if(amount > 0)
    {
            /* We have either configs w/ or w/o stencil, not both. Find one with best AA */
            int i,best_samples;
            best_samples    = 0;
            best_config     = configs[0];

            for(i=0 ; i<amount ; i++)
            {
                    int samp;
                    eglGetConfigAttrib( display, configs[i], EGL_SAMPLES, &samp );
                    if(samp > best_samples)
                    {
                            best_config  = configs[i];
                            best_samples = samp;
                    }
            }
    }
    else    best_config = (EGLConfig)0;             /* no suitable configs found */

    return best_config;
}
```

-If we did get some configs (with or without stencil), we find the one with most samples in multisampling mode

-For performance it might be wise to select config with 1 samples or 2 samples (in our case we just want the best one)

Texture matrix example

```
void appinit_glass(void)
{
    GLint texture_handle;

    /* View parameters */
    glMatrixMode          ( GL_PROJECTION );
    glFrustumf            ( -1.f, 1.f, -1.f, 1.f, 3.f, 1000.f );
    glMatrixMode          ( GL_MODELVIEW );

    /* Reset state */
    glEnable              ( GL_DEPTH_TEST );
    glClearColor          ( 0.f, 0.f, 0.1f, 1.f );

    /* Enable vertex arrays */
    glEnableClientState   ( GL_VERTEX_ARRAY );
    glEnableClientState   ( GL_TEXTURE_COORD_ARRAY );
```

-Here is another OpenGL ES example

-This one uses texture matrix for doing glTexEnv( ) type of things

-Normalized Vertex coordinates are used as texture coordinates

-Texture coordinates are transformed using the texture matrix to fake a glass-like look (rotate + scale x,y,z into s,t)

Texture matrix example

```
    /* Setup texture */
    glEnable            ( GL_TEXTURE_2D );

    glGenTextures       ( 1, texture_handle );
    glBindTexture       ( GL_TEXTURE_2D,  texture_handle );
    glTexImage2D        ( GL_TEXTURE_2D,  0, GL_RGB, 256, 256, 0,
                          GL_RGB, GL_UNSIGNED_BYTE, texture_data );
    glTexEnvi           ( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
                          GL_MODULATE );
    glTexParameteri     ( GL_TEXTURE_2D,  GL_TEXTURE_MIN_FILTER,
                          GL_LINEAR );
    glTexParameteri     ( GL_TEXTURE_2D,  GL_TEXTURE_MAG_FILTER,
                          GL_LINEAR );
    glTexParameteri     ( GL_TEXTURE_2D,  GL_TEXTURE_WRAP_S,
                          GL_CLAMP_TO_EDGE );
    glTexParameteri     ( GL_TEXTURE_2D,  GL_TEXTURE_WRAP_T,
                          GL_CLAMP_TO_EDGE );
}
```

-First we set up texture object

-Note that we use MODULATE as a TexEnv (will be modulated with default color)

Texture matrix example

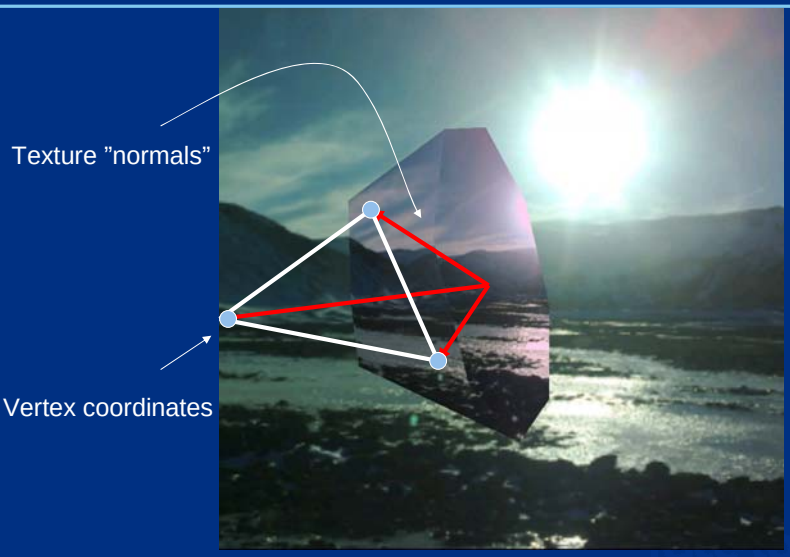
```
int render(float time)
{
    glClear          ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    /* draw background with two textured triangles */
    glMatrixMode             ( GL_TEXTURE );
    glLoadIdentity           ( );
    glMatrixMode             ( GL_PROJECTION);
    glLoadIdentity           ( );
    glMatrixMode             ( GL_MODELVIEW);
    glLoadIdentity           ( );
    glColor4ub               ( 255, 255, 255, 255 );
    glScalef                 ( 2.f, -2.f, 0.f );
    glTranslatef             ( -0.5f, -0.5f, 0.f );
    glVertexPointer          ( 2, GL_BYTE, 0, back_coords );
    glTexCoordPointer        ( 2, GL_BYTE, 0, back_coords );
    glDrawArrays             ( GL_TRIANGLE_STRIP, 0, 4 );
```

-First we render the texture as a background

-Default color is WHITE -> with MODULATE the texels are copied 1:1 from texture

Texture matrix example, coordinates

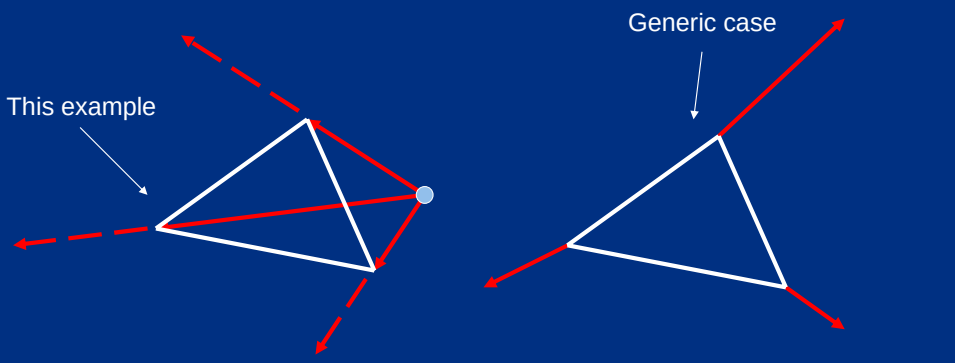


Texture matrix example, coordinates

We just take the (x,y) of the texture coordinate output

Texture matrix example, coordinates

Texture matrix example, coordinates



In this example we use the same data for vertex and texture ”normals” as the object is cut away from roughly tesselated sphere (all coordinates unit length)

This is NOT possible for general objects. You should use separate normalized normals for other objects

Texture matrix example

```
glMatrixMode        ( GL_PROJECTION );
glLoadIdentity      ( );
glFrustumf          ( -1.f, 1.f, -1.f, 1.f, 3.f, 1000.f );

glMatrixMode        ( GL_MODELVIEW );
glLoadIdentity      ( );
glTranslatef        ( 0, 0, -5.f );
glRotatef           ( time*25, 1.f, 1.f, 0.f );    /* (1) */
glRotatef           ( time*15, 1.f, 0.f, 1.f );

glMatrixMode        ( GL_TEXTURE );
glLoadIdentity      ( );
glTranslatef        ( 0.5f, 0.5f, 0.f );          /* [-0.5,0.5] -> [0,1]     */
glScalef            ( 0.5f, -0.5f, 0.f );          /* [-1,1]   -> [-0.5,0.5] */
glRotatef           ( time*25, 1.f, 1.f, 0.f );  /* identical rotations! */
glRotatef           ( time*15, 1.f, 0.f, 1.f );  /* see (1) */
```

-Next is the object

-We setup the modelview matrix so that it rotates when time goes by

-And we use exact same rotations in the texture matrix to match the modelview

-Also, we translate and scale the resulting x,y so that they are suitable as texture coordinates (r component not used)

Texture matrix example

```
    /* use different color for the (glass) object vs. the background */
    glColor4ub              ( 255, 210, 240, 255 );
    glVertexPointer         ( 3,GL_FIXED, 0, vertices );
    glTexCoordPointer       ( 3,GL_FIXED, 0, vertices );
    glDrawArrays            ( GL_TRIANGLES, 0, 16*3 );
}
```

- Finally we set the default color to kind of blueish red color (color of the glass) and draw the object

Texture matrix example

Fixed point programming

- Why should you use it?
  - Most mobile handsets don’t have a FPU
- Where does it make sense to use it?
  - Where it makes the most difference
  - For per-vertex processing: morphing, skinning, etc.
  - Per vertex data shouldn’t be floating point
- OpenGL ES API supports 32-bit FP numbers

Fixed point programming

- There are many variants of fixed point:
  - Signed / Unsigned
  - 2’s complement vs. Separate sign
- OpenGL ES uses 2’s complement
- Numbers in the range of [-32768, 32768 [
- 16 bits for decimal bits (precision of 1/65536)
- All the examples here use 16.16 fixed point

•Fixed point scale is 2^16 (65536, 0x10000).

Float to fixed and vice versa

- Convert from floating point to fixed point

```
#define float_to_fixed(a)  (int)((a)*(1<<16))   or
#define float_to_fixed(a)  (int)((a)*(65536))
```

- Convert from fixed point to floating point

```
#define fixed_to_float(a)  (((float)a)/(1<<16)) or
#define fixed_to_float(a)  (((float)a)/(65536))
```

SATURATION

Fixed point programming

- Examples:

```
0x0001 0000 =  65536       =     “1.0f”
0x0002 0000 =  2*65536     =     “2.0f”
0x0010 0000 =  16*65536    =     “16.0f”
0x0000 0001 =  1/65536     =     “0.0000152587…”
0xffff ffff = -1/65536(-0x0000 0001)
```

Fixed point operations

- Addition

```
#define add_fixed_fixed(a,b) ((a)+(b))
```

- Multiply fixed point number with integer

```
#define mul_fixed_int(a,b) ((a)*(b))
```

- MUL two FP numbers together

```
#define mul_fixed_fixed(a,b)                    \
          (int)((((int64)a)*((int64)b)) >> 16)
```

- Note: int64 depends on the compiler -> you should replace that with the platform/compiler 64-bit type (examples: int64, __int64, long long)

Fixed point operations and scale

Addition:

a & b = original float values

S = fixed point scale (e.g., 65536)

result = (a * **S**) + (b * **S**) = (a + b) * **S**

- **Scaling term keeps the same**
- **Range of the result is 33 bits**
- **Possible overflow by 1 bit**

Fixed point operations and scale

Multiplication:

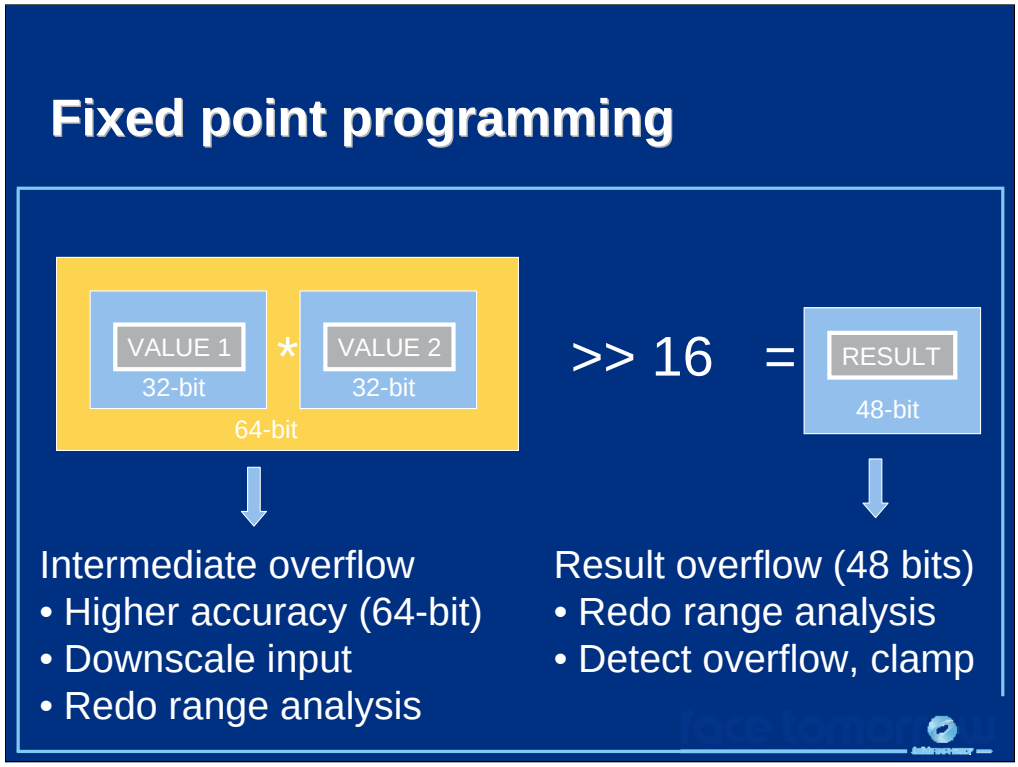
a & b = original float values

S = fixed point scale (e.g., 65536)

result	=	(a * **S**) * (b * **S**)	= ((a * b) * **S^2**)
final	=	((a * b) * **S**^2) **/ S**	= (a * b) * **S**

- **Scaling term is squared (S^2) and takes 32 bits**
- **Also the integer part of the multiplication takes 32 bits**

  => need 64 bits for full s16.16 * s16.16 multiply



-Multiplying two 32-bit numbers with standard C “int” multiply gives you lower 32 bits from that multiplication.

-Intermediate value may need 64 bits (high 32-bits cannot be ignored in this case).

-This can occur for example if you multiply two fixed point numbers together (also two fixed point scales multiplied together at the same time).

-Solution 1: use 64-bit math for the intermediate, use 64-bit shifter to get the result down.

-Solution 2: downscale on the input (just for this operation), for example divide input operands by 2^4, take that into account in result.

-Solution 3: redo the range analysis.

-Also the result may overflow (even if internal precision of 64-bit would be used for intermediate calculation).

-Solution 1: redo the ranges.

-Solution 2: clamp the results (it’s better to clamp than just overflow. Clamping limits the resulting error, with ignored overflow the errors easily become very large).

Fixed point programming

- Division of integer by integer to a fixed point result

```
#define div_int_int(a,b)                    \
        (int)((((int64)a)*(1<<16))/(b))
```

(a*S)/ b = (a/b)*S

- Division of fixed point by integer to a fixed point result

```
#define div_fixed_int(a,b) ((a)/(b))
```

- Division of fixed point by fixed point

```
#define div_fixed_fixed(a,b)                \
        (int)((((int64)a)*(1<<16))/(b))
```

(a*S*S)/(b*S) = (a/b)*S

Notes about overflows:

-MUL two FP numbers together can overflow in the intermediate calculation (a*b), an example: 2.0 * 2.0 (intermediate is: 2*2*1^16*1^16, requires 35 bits intermediate incl. sign bit).

-If the operation can be done with 32x32 -> 64-bit multiply, followed by 16-bit shift, overflow only occurs if the result after the shift does not fit into 32-bit (in that case either the range has to be changed or the destination should be carried over in 64-bit number).

-Division of integer by integer can overflow if a is not in the range [-32768,32767] (because multiplication of a by (1<<16) does not fit in to 32 bits).

-Division of fixed by integer cannot overflow, but results may become zero.

-Division of fixed by fixed may overflow if a is not in range]-1.0, 1.0[, intermediate overflow.

Fixed point programming

- Power of two MUL & DIV can be done with shifts
  - a * 65536 = a << 16, a / 256 = a >> 8
- Fixed point calculations overflow easily
- Careful analysis of the range requirements is required

=>

Always add validation code to your fixed point code

Fixed point programming

```
#if defined(DEBUG)
int add_fix_fix_chk(int a, int b)
{
  int64 bigresult = ((int64)a) + ((int64)b);
  int smallresult = a + b;
  assert(smallresult == bigresult);
  return smallresult;
}
#endif

#if defined(DEBUG)
#  define add_fix_fix(a,b) add_fix_fix_chk(a,b)
#else
#  define add_fix_fix(a,b) ((a)+(b))
#endif
```

-Do all of the fixed point operations with macros and not by direct calculus.

-Create DEBUG variants for every operation you do in fixed point (even simplest ADD, MUL, …). When you are compiling debug builds, all operations should assert that no overflows occur. If overflow assert is triggered, something needs to be done (ignore if not big enough visual impact, change ranges, etc.).

Fixed point math functions

- Complex math functions
  - Pre-calculate for the range of interest
- An example: Sin & Cos
  - Sin table between [0, 90°], fixed point angle (S = 2048)
  - Generate other angles and Cos from the table
  - Store in a short table (16-bit) as s0.16 (S = 32768)
  - Range for shorts is [-32768,32767] ([-1.0, 1.0[for s0.16 FP)
  - Equals to [-1.0, +1.0[for s0.16 FP (+1.0 not included !)
  - Negative values stored in the table (can represent -1.0)

Sin table precalculation

```
void calculate_table(short *out)
{
  int i;

  for(i=0;i<2048;i++)
  {
     float angle = (0.5f*PI*i)/2048.0;
     out[i] = -(int)(sin(angle)*32768);
  }
}
```

Sin function

```
inline int fp_sin(int angle)
{
  int ph    = angle & (2048 + 4096);  /* phase */
  int ang   = angle & 2047;           /* sub-angle */

  /* return negated values (was stored negated) */
  if(ph == 0)         return –((int)table[ang]);
  else if(ph == 2048) return –((int)table[2048-ang]);
  else if(ph == 4096) return  (int)table[ang];
  else                return  (int)table[2048-ang];
}
```

How to use fp_sin()

```
void do_something(int ang)
{
  int i;

  for( i=0; i<1000; i++)
  {
      int tmp;
      tmp = (vin[i*3] * fp_sin(ang)) >> 15;
      vout[i*3] = tmp;
  }
}
```

- note: fp_sin returns integers
  => it can also return 32768 (1.0)

- it does not fit inside s0.16 fixed point number !

Performance

- fp_sin() is rather complex
- Simple optimization: calculate 360 degrees
- Downside: takes more memory
- And: to handle 1.0 we have to use S = 16384

Sin table precalculation (360 deg)

```
void calculate_table(short *out)
{
  int i;

  for(i=0;i<2048*4;i++)
  {
     float angle = (2.f*PI*i)/2048.0;
     out[i] = (int)(sin(angle)*16384;
  }
}
```

Sin function (360 deg)

```
inline int fp_sin(int angle)
{
  return ((int)table[angle & 8191]);
}
```

Example: Simple morphing (LERP)

- Simple fixed point morphing loop (16-bit data, 16-bit coeff)

```
#define DOLERP_16(a,b,t) (short)(((((b)-(a))*(t))>>16)+(a))

void lerpgeometry(short *out, const short *inA, const short *inB,
   int count, int scale)
{
   int i;

   for(i=0; i<count; i++)
   {
        out[i*3+0] = DOLERP_16(inB[i*3+0], inA[i*3+0], scale);
        out[i*3+1] = DOLERP_16(inB[i*3+1], inA[i*3+1], scale);
        out[i*3+2] = DOLERP_16(inB[i*3+2], inA[i*3+2], scale);
   }
}
```

-Morphing is done for 16-bit vertex data (16-bit vertices, 16-bit normals).

-This is done to make the fixed point math to fit inside of 32-bit integers.

-Standard 32-bit mul and addition is enough here.

Converting existing code

- OS/device conversions
  - Programming model, C/C++, compiler, CPU
- Windowing API conversion
  - EGL API is mostly cross platform
  - EGL Native types are platform specific
- OpenGL -> OpenGL ES conversion

Example: Symbian porting

Programming model

- C++ with some changes (e.g., exceptions)
- Event based programming (MVC), no main / main loop
- Three level multitasking: Process, Thread, Active Objects

- ARM CPU
  - Unaligned memory accesses will cause exception (unlike x86)

- OpenC (http://www.forum.nokia.com/openc)

Example: EGL porting

- Native types are OS specific
  - EGLNativeWindowType (RWindow *)
  - EGLNativePixmapType (CFbsBitmap *)
  - Pbuffers are portable
- Config selection
  - Select the color depth to be same as in the display
- Windowing system issues
  - What if render window is clipped by a system dialog?
  - Only full screen windows may be supported

- Even though Pbuffers are “portable” in the sense that they are OS independent in the EGL API, there may be implementations that do not support Pbuffers at all.

OpenGL porting

- glBegin/glEnd wrappers
  - _glBegin stores the primitive type
  - _glColor changes the current per-vertex data
  - _glVertex stores the current data behind arrays and increments
  - _glEnd calls glDrawArrays with primitive type and length

```
_glBegin(GL_TRIANGLES);
  _glColor4f(1.0,0.0,0.0,1.0);
  _glVertex3f(1.0,0.0,0.0);
  _glVertex3f(0.0,1.0,0.0);
  _glColor4f(0.0,1.0,0.0,1.0);
  _glVertex3f(0.0,0.0,1.0);
_glEnd();
```

-In the code above color is only specified twice, but in the vertex arrays it needs to be specified for each vertex.

-_glVertex3f call copies the current color, normal, texcoord to the vertex arrays even if those are not changed in the emulated code.

OpenGL porting

- Display list wrapper
  - Add the display list functions as wrappers
  - Add all relevant GL functions as wrappers
  - When drawing a list, go through the collected list

OpenGL porting

```
void _glEnable( par1, par2 )
{
  if( GLOBAL()->iSubmittingDisplayList )
  {
    *(GLOBAL()->dlist)++ = DLIST_CMD_GLENABLE;
    *(GLOBAL()->dlist)++ = (GLuint)par1;
    *(GLOBAL()->dlist)++ = (GLuint)par2;
  }
  else
  {
    glEnable(par1,par2);
  }
}
```

-This is a example of a wrapped glEnable( ) call. Internally it checks if the display list is being built. If it is, we just collect the data from this function call to the list for later execution.

-Note: Display Lists allow for all sorts of optimizations in _theory_ (like precalculating things for occlusion culling, analyzing vertex ranges, …), but it is hard to do in practice. For example, here we should perhaps analyze also if the enable actually has any effect, or if it creates a “state block” that could be tracked and the rendering optimized inside the display list code.

-Doing optimal display lists on these devices with small amount of memory is tricky. If you really need performance for the emulated application, convert the application to use vertex arrays instead.

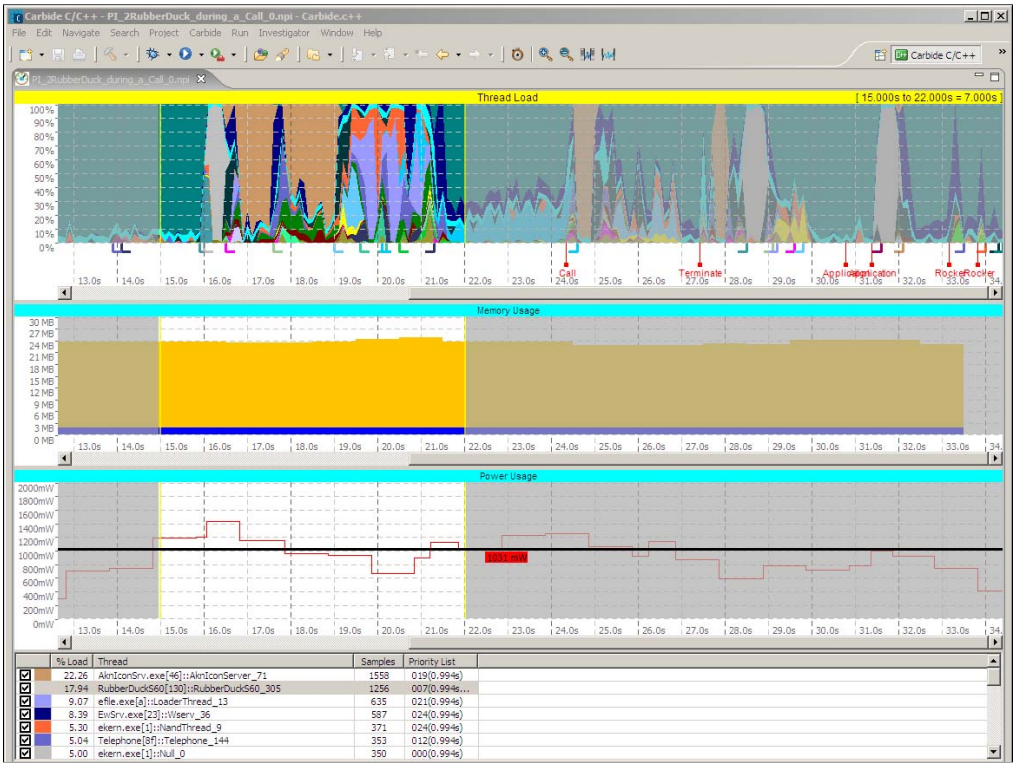
OpenGL porting

- Vertex arrays
  - OpenGL ES supports only vertex arrays
  - SW implementations get penalty from float data
  - Use as small types as possible (byte, short)
  - For HW it shouldn't make a difference, mem BW
  - With OpenGL ES 1.1 always use VBOs

-Memory usage is crucial. If your geometry fits into 8-bit without degradation in quality, do it. It uses less memory and can save some CPU cycles from transforms on the side (for example, ARM multiplication of 32x8 can be 2 cycles, whereas 32x32 can be 5 cycles).

OpenGL porting

- No quads
  - Convert a quad into 2 triangles
- No real two-sided materials in lighting
  - If you really need it, submit front and back triangles
- OpenGL ES and querying state
  - OpenGL ES 1.0 only supports static getters
  - OpenGL ES 1.1 supports dynamic getters
  - For OpenGL ES 1.0, create own state tracking if needed


Carbide C/C++ - P1_2RubberDuck_during_a_Call_0.npi - Carbide.c++
File Edit Navigate Search Project Carbide Run Investigator Window Help
Carbide C/C++
P1_2RubberDuck_during_a_Call_0.npi
Thread Load
[15.000s to 22.000s = 7.000s]
Call
Terminate
Application
Application
Rocker
Rocker
Memory Usage
Power Usage
1031 mW
% Load
Thread
Samples
Priority List
22.26 AknIconSrv.exe[46]::AknIconServer_71 1558 019(0.994s)
17.94 RubberDuckS60[130]::RubberDuckS60_305 1256 007(0.994s...
9.07 efile.exe[a]::LoaderThread_13 635 021(0.994s)
8.39 EvSrv.exe[23]::Wserv_36 587 024(0.994s)
5.30 ekern.exe[1]::NandThread_9 371 024(0.994s)
5.04 Telephone[3f]::Telephone_144 353 012(0.994s)
5.00 ekern.exe[1]::Null_0 350 000(0.994s)

Demo

- Sequel to game One (Nokia)

Questions?


SIGGRAPH2007

OpenGL ES on PyS60

Kari Pulli

Nokia Research Center

Python: Great for rapid prototyping

- Python
  - designed to be as small, practical, and open as possible
  - easy and fun OO programming
- sourceforge.net/projects/pyS60
  - Python 2.2.2 on Symbian S60
  - wrappers for phone SDK libraries
  - can extend in Symbian C++

Python bindings to OpenGL ES

- Almost direct bindings
- OpenGL ES functions that take in pointers typically take in a Python list
- Next we'll show a full S60 GUI program with OpenGL ES

Import libraries

```
import appuifw    # S60 ui framework
import sys

from glcanvas  import *
from gles      import *
from key_codes import *
```

Application class, data

```
class Hello:
    vertices = array( GL_BYTE, 3,
                      [-1, 1, 0,
                        1,-1, 0,
                        1, 1, 0] )
    colors = array( GL_UNSIGNED_BYTE, 4,
                    [255, 0,   0,  255,
                     0, 255,   0,  255,
                     0,   0, 255,  255] )
```

Initialize the application

```
def __init__(self):                    # class constructor
    self.exiting = False                # while !exit, run
    self.frame, self.angle = 0, 0       # set variables
    self.old_body = appuifw.app.body
    try:                                # create surface
        c = GLCanvas( redraw_callback = self.redraw,
                      resize_callback = self.resize )
        appuifw.app.body = c
        self.canvas = c
    except Exception, e:
        appuifw.note( u"Exception: %s" % (e) )
        self.start_exit()
        return
    appuifw.app.menu = [(u"Exit", self.start_exit)]
    c.bind( EKeyLeftArrow,  lambda:self.left() )
    c.bind( EKeyRightArrow, lambda:self.right() )
    self.initgl()
```

Keyboard and resize callbacks

```
def left(self):
    self.angle -= 10

def right(self):
    self.angle += 10

def resize(self):
    if self.canvas:
        glViewport( 0, 0,
                    self.canvas.size[0],
                    self.canvas.size[1] )
```

Main loop

```
def start_exit(self):
    self.exiting = True

def run(self):
    app = appuifw.app
    app.exit_key_handler = self.start_exit
    while not self.exiting:
        self.canvas.drawNow()
        e32.ao_sleep( 0.01 )
    app.body             = self.old_body
    self.canvas          = None
    app.exit_key_handler = None
```

Initialize OpenGL ES

```
def initgl(self):
    glMatrixMode( GL_PROJECTION )
    glFrustumf  ( -1.0, 1.0, -1.0, 1.0,
                  3.0, 1000.0 )
    glMatrixMode( GL_MODELVIEW )
    glDisable   ( GL_DEPTH_TEST )
    glShadeModel( GL_SMOOTH )
    glClearColor( 0.0, 0.0, 0.1, 1.0 )
    glVertexPointerb( self.vertices )
    glColorPointerub( self.colors )
    glEnableClientState( GL_VERTEX_ARRAY )
    glEnableClientState( GL_COLOR_ARRAY )
```

Draw cycle

```
def redraw(self, frame=None):
    if self.canvas:
        glClear( GL_COLOR_BUFFER_BIT )
        glLoadIdentity()
        glTranslatef( 0.0, 0.0, -5.0 )
        glRotatef   ( self.angle,
                      0.0, 0.0, 1.0 )
        glRotatef   ( self.frame,
                      0.0, 1.0, 0.0 )
        glDrawArrays( GL_TRIANGLES, 0,3 )
        self.frame += 1
```

Using the class

```
appuifw.app.screen = 'full'
try:
    app = Hello()
except Exception, e:
    appuifw.note( u"Cannot start: %s" %
                  (e) )
else:
    app.run()
del app
```


SIGGRAPH2007

High-performance OpenGL ES 1.x

Ville Miettinen

NVIDIA

Targeting the ”mobile platform”

- CPU speed and available memory varies
  - Current range ~30Mhz - 600MHz, ARM7 to ARM11, no FPUs
- Different resolutions
  - QCIF (176x144) to VGA (640x480), antialiasing on higher-end devices
  - Color depths 4-8 bits per channel (12-32 bpp)
- Portability issues
  - Different CPUs, OSes, Java VMs, C compilers, ...
  - OpenKODE from the Khronos Group will help to some extent

Graphics capabilities

- General-purpose multimedia hardware
  - Pure software renderers (all done using CPU & integer ALU)
  - Software + DSP / WMMX / FPU / VFPU
  - Multimedia accelerators
- Dedicated 3D hardware
  - Software T&L + HW tri setup / rasterization
  - Full hardware acceleration
- Performance: 50K – 2M tris, 1M – 100M pixels / sec
- Next gen: 30M+ tris, 1000M pixels / sec

Standards help somewhat

- Act as hardware abstraction layers
  - Provide programming interface (API)
  - Same feature set for different devices
  - Unified rendering model
- Performance cannot be guaranteed

Scalability

- Successful application has to run on hundreds of different phone models
  - No single platform popular enough
- Same game play but can scale video and audio
- Design for lowest-end, add eye candy for high-end
  - Scalability has to be built into the design

3D content is easy to scale

- Separate low and high poly count 3D models
- Different texture resolutions & compressed formats
- Rendering quality can be scaled
  - Texture filtering, perspective correction, blend functions, multi-texturing, antialiasing

Special effects

- Identify special effects
  - Bullet holes, skid marks, clouds, ...
  - Cannot have impact on game play
    - Fog both game play and visual element
    - Multiplayer games have to be fair
- Users can alter performance by controlling effects

Tuning down the details

- Particle systems
  - Number of particles, complexity, visuals
  - Shared rendering budget for all particle systems
- Background elements
  - Collapse into sky cubes, impostors
- Detail objects
  - Models to have “important” and “detail” parts

Profiling

- Performance differences often system integration issues - not HW issues
- Measuring is the only effective way to find out how changes in code affect performance
- Profile on actual target device if possible
- Public benchmark apps provide some idea of graphics performance
- gDEBugger ES for gfx driver profiling

Identifying bottlenecks

- Three groups: application code, vertex pipeline, pixel pipeline
  - Further partitioned into pipeline stages
  - Overall pipeline runs as fast as its slowest stage
- Locate bottlenecks by going through each stage and reducing its workload
  - If performance changes, you have a bottleneck
- Apps typically have multiple bottlenecks

Pixel pipeline bottlenecks

- Find out by changing rendering resolution
  - If performance increases, you have a bottleneck
  - Either texturing or frame buffer accesses
- Remedies
  - Smaller screen resolution, render fewer objects, use simpler data formats, smaller texture maps, less complex fragment and texture processing

Vertex pipeline bottlenecks

- Vertex processing or submission bottlenecks
  - Find out by rendering every other triangle but using same vertex arrays
- Remedies
  - Submission: smaller data formats, cache-friendly organization, fewer triangles
  - Vertex processing: simpler T&L (fewer light sources, avoid dynamic lighting and fog, avoid floating-point data formats)

Application code bottlenecks

- Two ways to find out
  - Turn off all application logic
  - Turn off all rendering calls
- Floating-point code #1 culprit
- Use profiler
  - HW profilers on real devices costly and hard to get
  - Carbide IDE from Nokia (S60 and UIQ Symbian)
  - Lauterbach boards
  - Desktop profiling (indicative only)

Changing and querying the state

- Rendering pipes are one-way streets
- Apps should know their own state
  - Avoid dynamic getters if possible!
- Perform state changes in a group at the beginning of a frame
- Avoid API synchronization
  - Do not mix 2D and 3D libraries!

”Shaders”

- Combine state changes into blocks (”shaders”)
  - Minimize number of shaders per frame
  - Typical application needs only 3-10 ”pixel shaders”
    - Different 3-10 shaders in every application
    - Enforce this in artists’ tool chain
- Sort objects by shaders every frame
  - Split objects based on shaders

Complexity of shaders

- Software rendering: everything costs!
  - Important to keep shaders as simple as possible
    - Even if introduces additional state changes
    - Example: turn off fog & depth buffering when rendering overlays
- Hardware rendering: Usually more important to keep number of changes small

Model data

- Keep vertex and triangle data short and simple!
  - Better cache coherence, less memory used
- Make as few rendering calls as possible
  - Combine strips with degenerate triangles
- Weld vertices using off-line tool
- Order triangle data coherently
- Use hardware-friendly data layouts
  - Buffer objects allow storing data on server

Transformation pipeline

- Minimize matrix changes
  - Changing a matrix may involve many hidden costs
  - Combine simple objects with same transformation
  - Flatten and cache transformation hierarchies
- ES 1.1: Skinning using matrix palettes
  - CPU doesn't have to touch vertex data
- ES 1.1: Point sprites for particle effects

Rendering pipeline

- Rendering order is important
  - Front-to-back improves depth buffering efficiency
  - Also need to minimize number of state changes!
- Use culling to speed up rendering pipeline
  - Conservative: frustum culling & occlusion culling
    - Portals and Potentially Visible Sets good for mobile
  - Aggressive culling
    - Bring back clipping plane in, drop detail & small objects

Lighting

- Fixed-function lighting pipelines are so 1990s
  - Drivers implemented badly even in desktop space
  - In practice only single directional light fast
  - OpenGL's attenuation model difficult to use
  - Spot cutoff and specular model cause aliasing
  - No secondary specular color
  - Flat shading sucks
  - Artifacts unless geometry heavily tessellated

Lighting (if you have to use it)

- Single directional light usually accelerated
- Pre-normalize vertex normals
- Avoid homogeneous vertex positions
- Turn off specular illumination
- Avoid distance attenuation
- Turn off distant non-contributing lights

Lighting: the fast way

- While we're waiting for OpenGL ES 2.0 drivers
  - Pre-computed vertex illumination good if slow T&L
  - Illumination using texturing
    - Light mapping
    - ES 1.1: dot3 bump mapping + texture combine
    - Less tessellation required
  - Combining with dynamic lighting: color material tracking

Environment mapping

Environment map taken in a Chinese restaurant (the stairs had a side rail with spherical handles).

Starlancer by Microsoft. All illumination done using texture-based lighting effects.

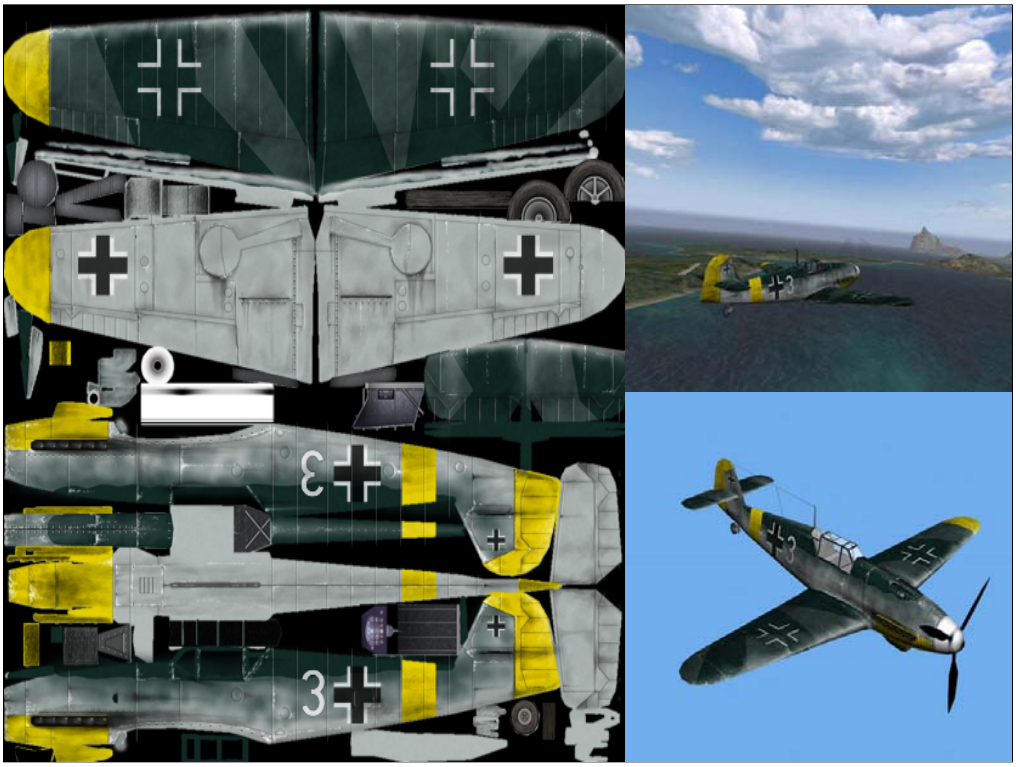
Baked lighting (baked both into vertices and textures) combined with various reflection-mapping effects. Images by Hybrid Graphics, rendered using a real-time software raserizer.

Textures

- Mipmaps always a Good Thing™
  - Improved cache coherence and visual quality
  - ES 1.1 supports auto mipmap generation
- Avoid modifying texture data
- Keep textures ”right size”, use compressed textures
- Different strategies for texture filtering & perspective correction
  - SW implementations affected

Textures (cont'd)

- Multitexturing
  - Always faster than doing multiple rendering passes
  - ES 1.1: support at least two texturing units
  - ES 1.1: TexEnvCombine neat toy
- Use small & compressed texture formats
- Texture atlases: combining multiple textures
  - Reduces texture state changes



Textures and shots from Kesmai's Air Warrior 4 (never published)

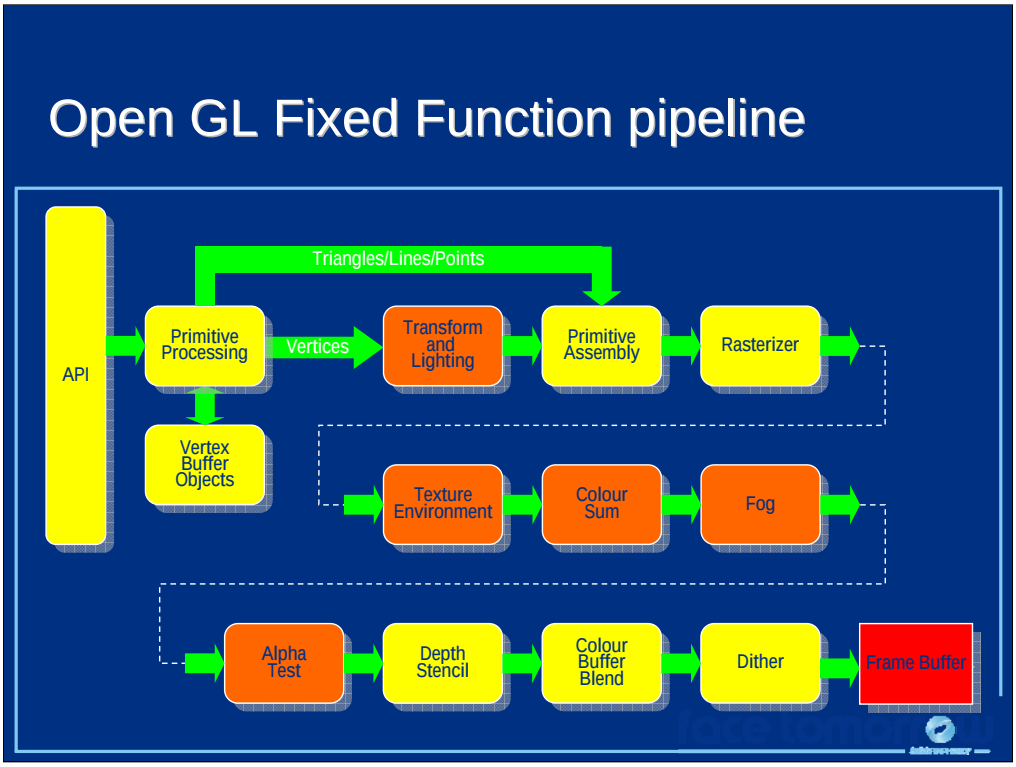

SIGGRAPH2007

ES 2.0 Overview

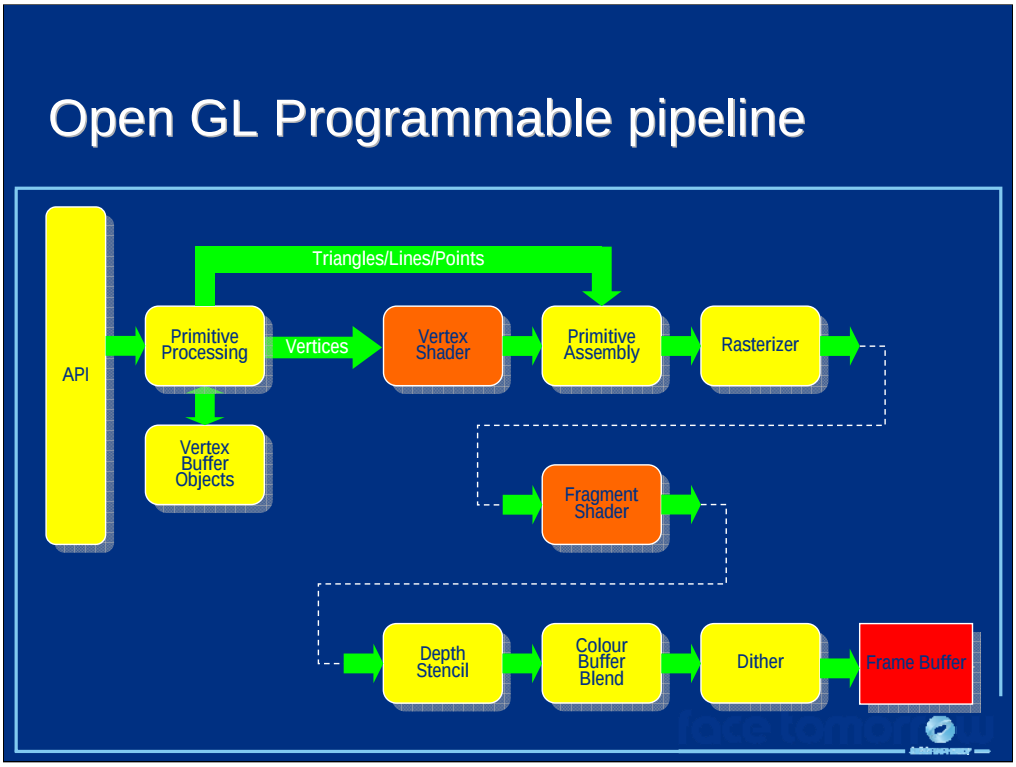
Robert J. Simpson
AMD

Contents

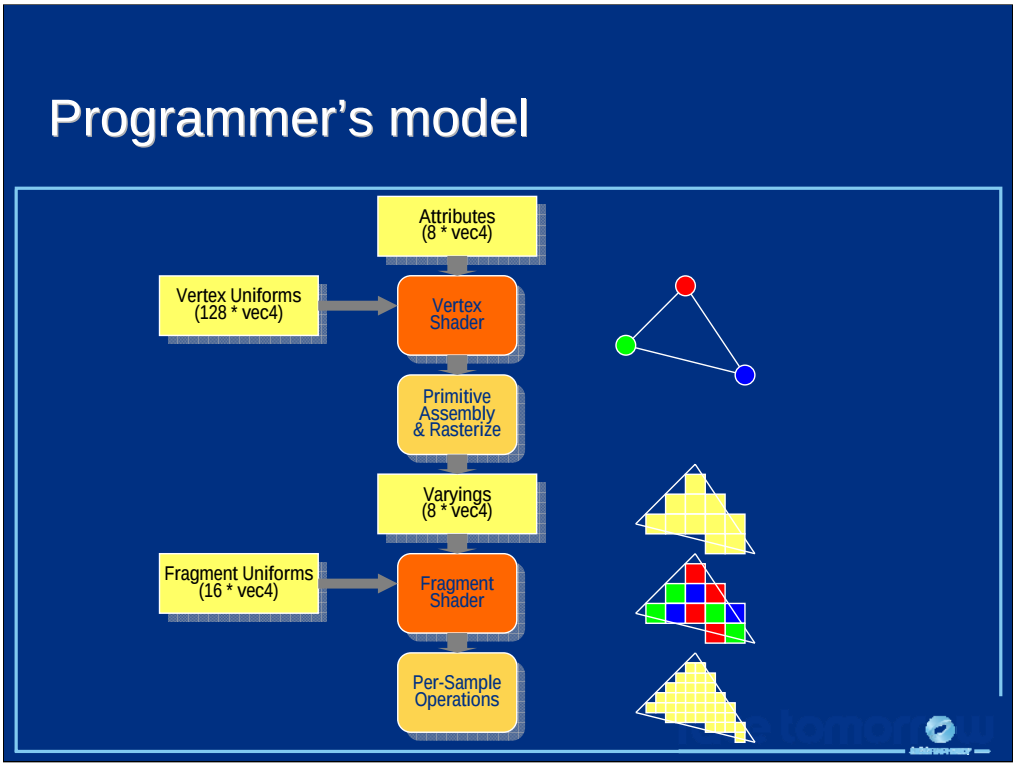
- The OpenGL ES 2.0 Pipeline
- API Overview
- GLSL ES 1.00 Overview
- Writing an ES 2.0 Application
- Examples
- Future Directions



This shows the existing ES 1.1 pipeline. The boxes coloured orange represent the functions that will be replaced by shaders in ES 2.0. The transform and lighting block is replaced by the vertex shader. The texture environment, colour sum, fog and alpha test are all replaced by the fragment shader.



This shows how the vertex and fragment shaders fit into the ES 2.0 pipeline.



The input to the pipeline is a set of attributes. Each vertex can have up to 8 attributes and each attribute can be up to a 4-vector in size. Attributes are not shared between vertices. Note that for each draw call (e.g. a triangle list or a triangle strip), all the vertices must have the same number and type of attributes.

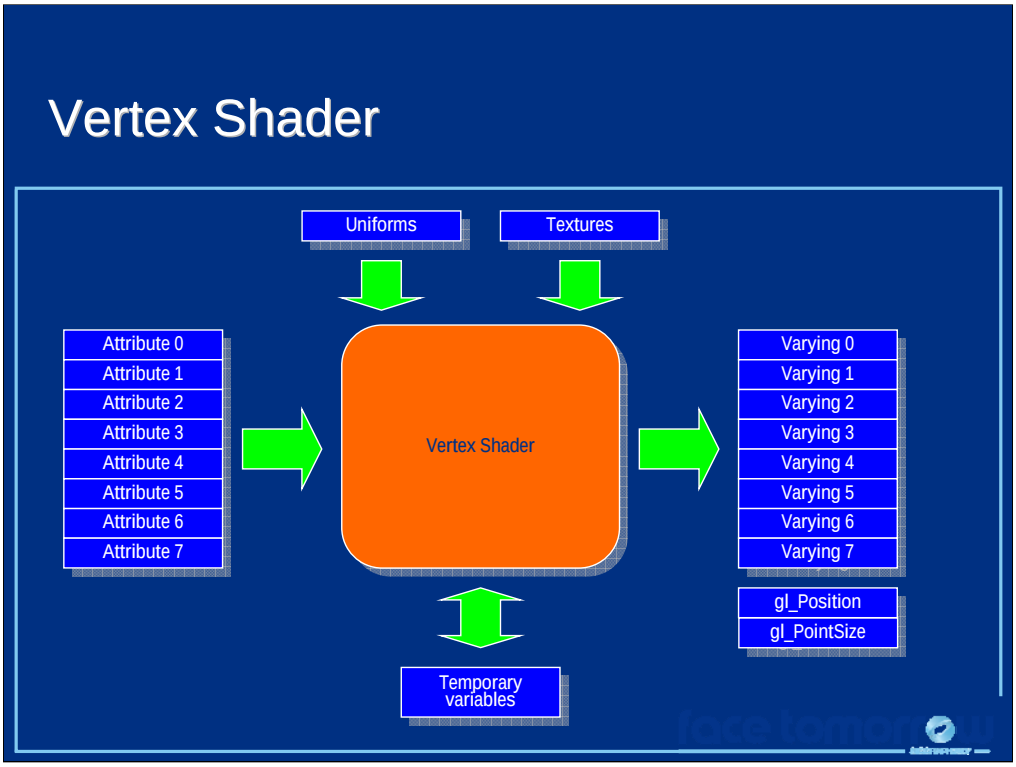
The vertex shader is run once for each vertex. The inputs to the vertex shader are the vertex attributes and the vertex uniforms. Attributes are used to specify values that vary relatively frequently such as position that usually varies per vertex. Uniforms are used to specify values that vary less frequently such as transforms and light positions.

For each vertex processed, as set of value is output from the vertex shader to the rasterizer. The number and types of values output does not need to be the same as the number and types input.

Up until this point, the values are per-vertex. The rasterizer inputs each triangle with its associated vertex values and generates a set of fragments, one for each pixel of the triangle. The vertex input values are interpolated across the triangle (taking into account the perspective) and the per-fragment values that are output are known as 'varyings'

The fragment shader operates in a similar manner to the vertex shader. The shader runs once for each fragment input. It reads the incoming varyings and the fragment uniforms and outputs a colour value.

This is the end of the programmable operations. After the fragment shader, the rest of the pipeline is fixed function. The per-sample operations are depth-stencil, colour blend and dither. Depending on the implementation, these may be performed at per-pixel resolution or may (in the case of multi-sampling) be performed at a higher resolution.

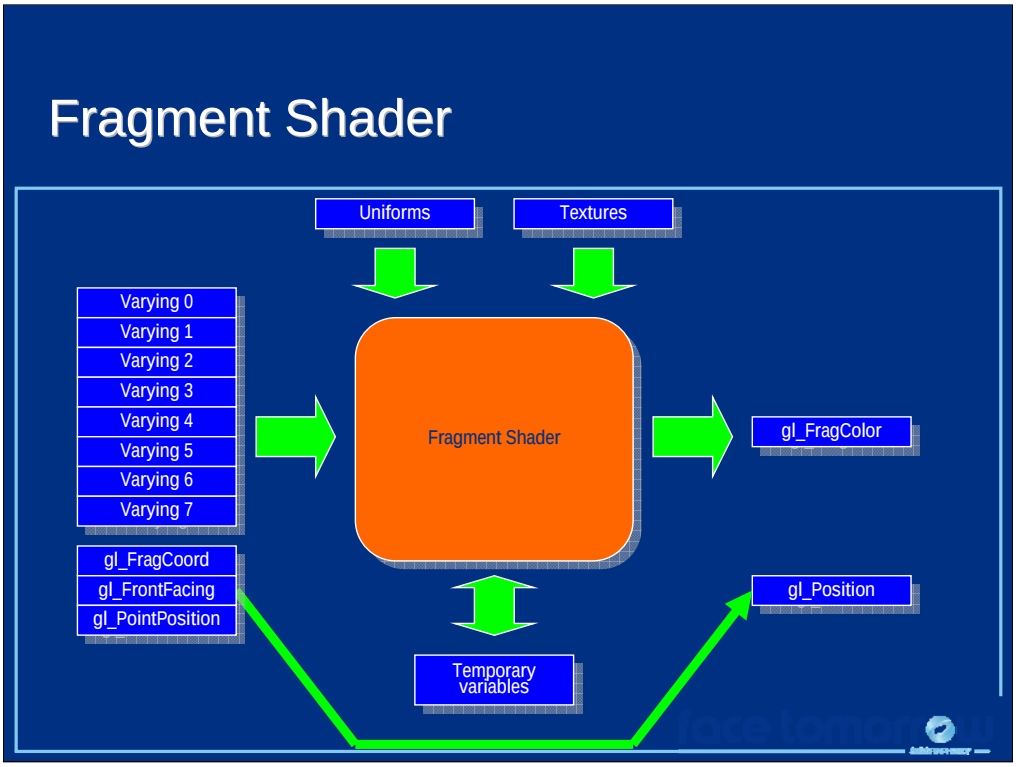


This shows the programmers view of the vertex shader. The (up to) 8 attributes are input on the left. Uniforms can be read by the shader. In some implementations, the vertex shader may also access textures. While the shader is running, it has access to a number of temporary variables. Note that these variables cannot be shared between vertices.

The output is a set of user-specified varyings. The total size must be less that 8 4-vectors and there are a set of rules that describe how to calculate the space required given a set of varyings. See the ESSL specification for details.

In addition there are some fixed function variables that the shader may write to. The most important is the transformed position gl_Position. This value is used by the rasterizer and (except in a few very special cases) should always be written to. The gl_PointSize should be written to by the vertex shader for point primitives.

After primitive assembly the gl_FrontFacing flag (not shown) is set automatically according to whether the triangle is front facing or back facing.



The fragment shader inputs the user-defined varyings, the fixed function variables and the fragment uniforms. The shader can read textures. It also has a set of temporary variables available. Like the vertex shader, these temporary variables cannot be shared.

The output from the fragment shader is a colour value. The position of the fragment is output automatically.

The Vertex Shader

- The vertex shader can do:
  - Transformation of position using model-view and projection matrices
  - Transformation of normals, including renormalization
  - Texture coordinate generation and transformation
  - Per-vertex lighting
  - Calculation of values for lighting per pixel

In general, any operation that takes in a vertex and outputs a modified form of that vertex, one at a time, can be performed by the vertex shader.

The Vertex Shader

- The vertex shader cannot do:
  - Anything that requires information from more than one vertex
  - Anything that depends on connectivity.
  - Any triangle operations (e.g. clipping, culling)
  - Access colour buffer

The vertex shader does not have access to neighbouring vertices or any per-pixel or per-fragment values.

The Fragment Shader

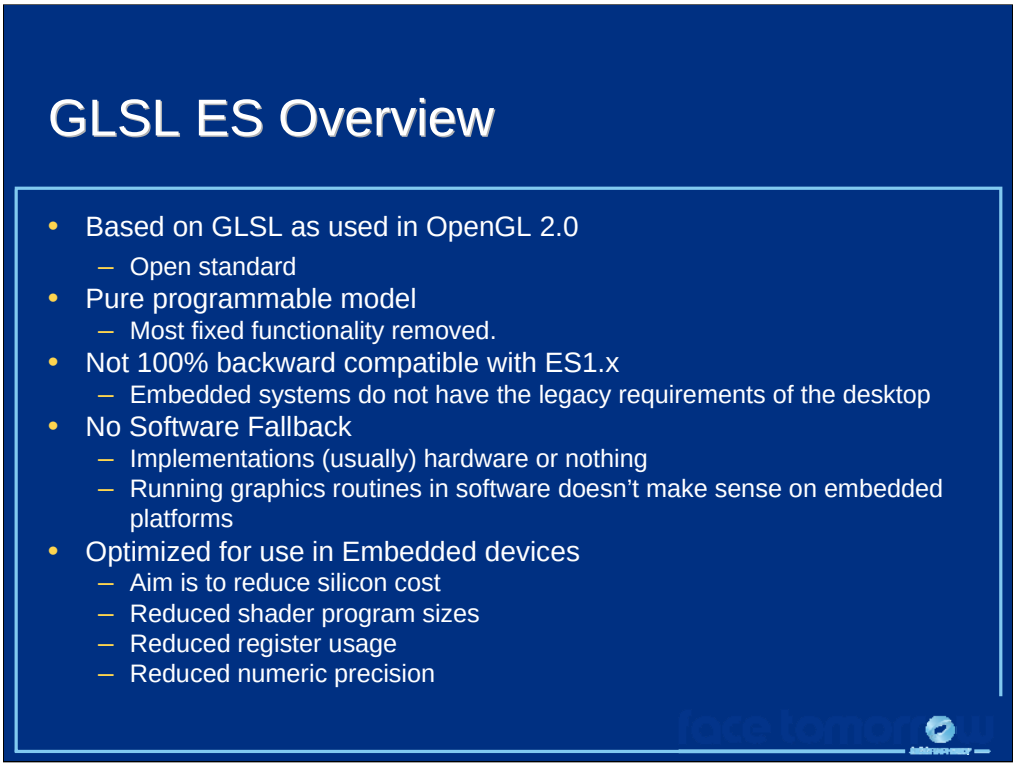
- The fragment shader can do:
  - Texture blending
  - Fog
  - Alpha testing
  - Dependent textures
  - Pixel discard
  - Bump and environment mapping

In general any operation that takes in a fragment and generates a colour can be performed by the fragment shader.

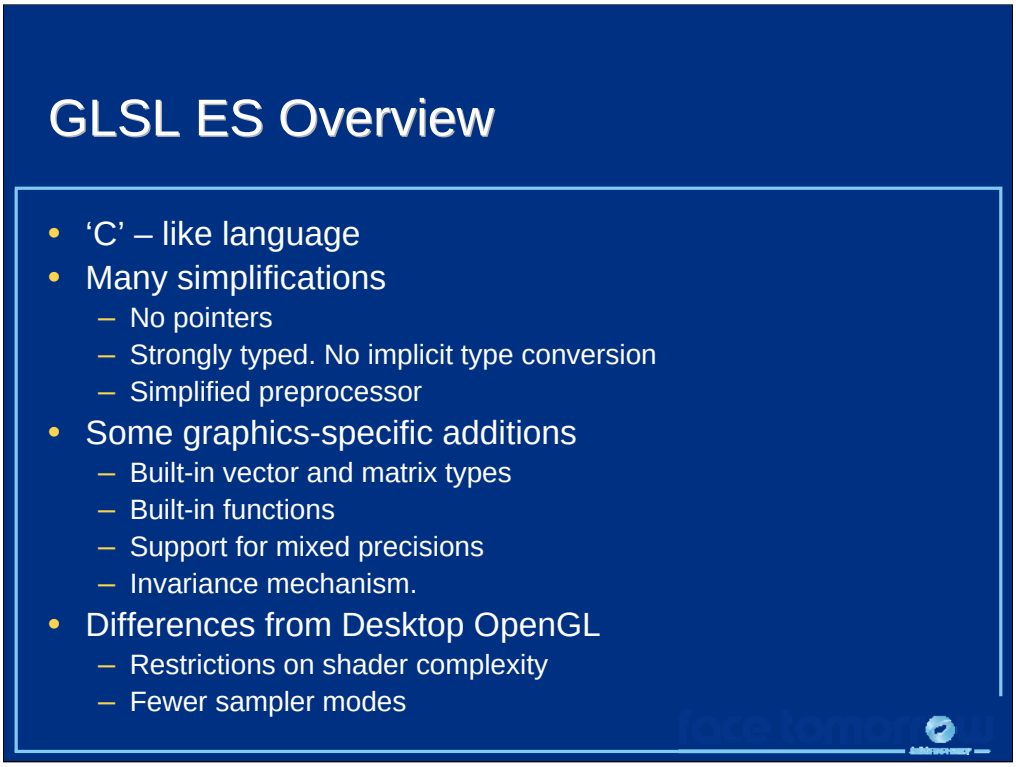
The Fragment Shader

- The fragment shader cannot do:
  - Blending with colour buffer
  - ROP operations
  - Depth or stencil tests
  - Write depth

Certain operations are not possible due to the difficulties they cause with implementations. For example, programmable blending operations require a read before or during shader execution and this can incur a high silicon and/or performance cost. Depth and stencil operations may be done at higher that pixel resolution if multi-sampling is used and so requires a 3rd type of shader. Programming these functions is generally less useful although it is possible it may be introduced in a later version of the specification.



GLSL ES (sometimes abbreviated to ESSL) is designed to be simple to use and simple to implement. The main difference between the desktop and embedded versions of the language is the removal of most of the fixed function from ESSL. The interaction between the fixed and programmable functions can be quite complex. The smaller size of embedded applications means that it will be easier to port ES 1.1 applications to a pure programmable API than desktop apps.

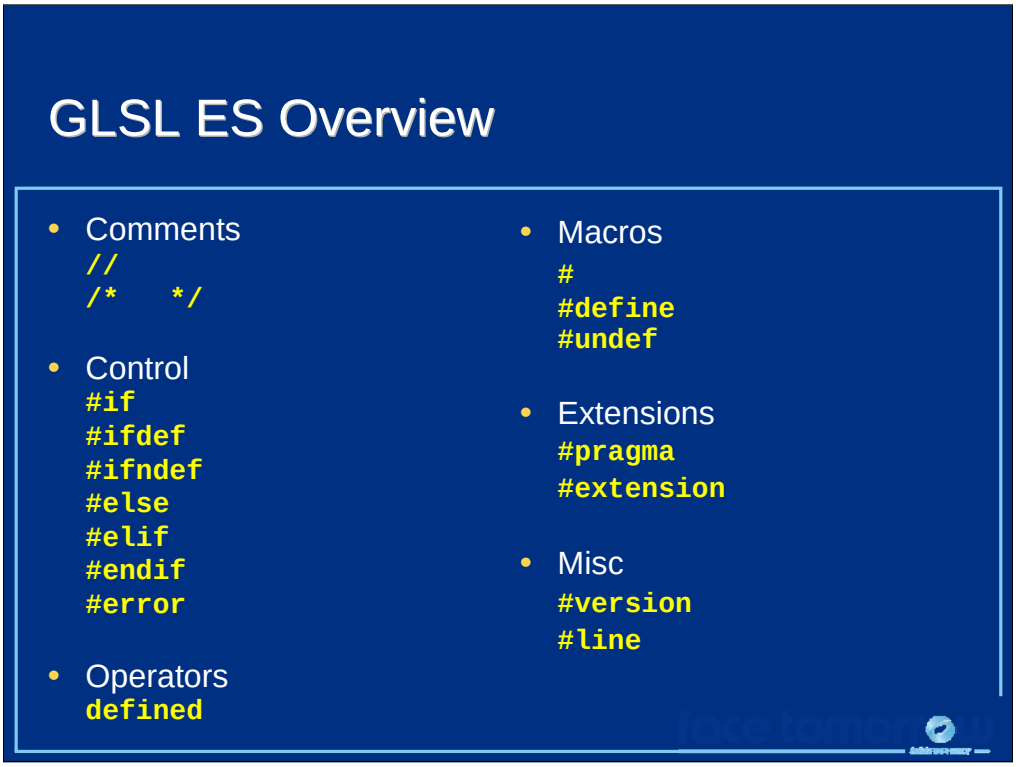


'C'-like languages are generally easier for developers to learn, given the widespread used of C/C++/Java. However a number of features have been removed as they complicate the compiler, are difficult to implement efficiently in hardware and can be a common source of bugs. So the language is strongly typed with no implicit type conversion. There are no pointers and no goto statements.

A number of graphics-specific additions have been made. There are new intrinsic types for vectors and matrices, a comprehensive set of 'helper' functions for common graphics operations. There is support for mixed precisions which allows more effective use to be made of the hardware. Not all operations need to be float32 and full float32 is very expensive for mobile devices.

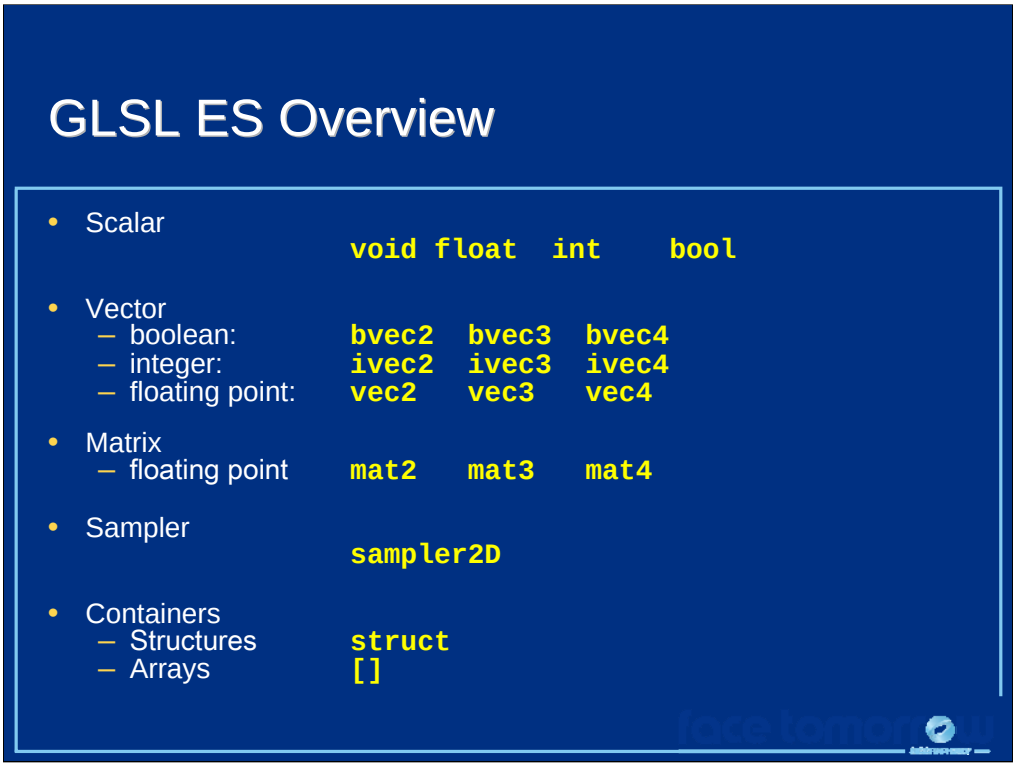
The invariance mechanism has been added to ESSL (and subsequently to desktop GLSL) in recognition of the increasing sophistication of compiler technology (see later for details).

Desktop GL has the philosophy that applications will always run, even if that means emulating the GL pipeline in software. In the embedded market, this does not make sense. Having an application that takes seconds or minutes to render a frame is considered pointless. The philosophy for GL ES is that applications should run fast enough or not at all. The ES working group is in the process of specifying a set of parameters that developers will be able to use to determine if a given shader will run across a range of implementations. Given the variety of implementations already in the market, this is a difficult task and is unlikely to be 100% accurate. However it is considered very important that there is portability across implementations.

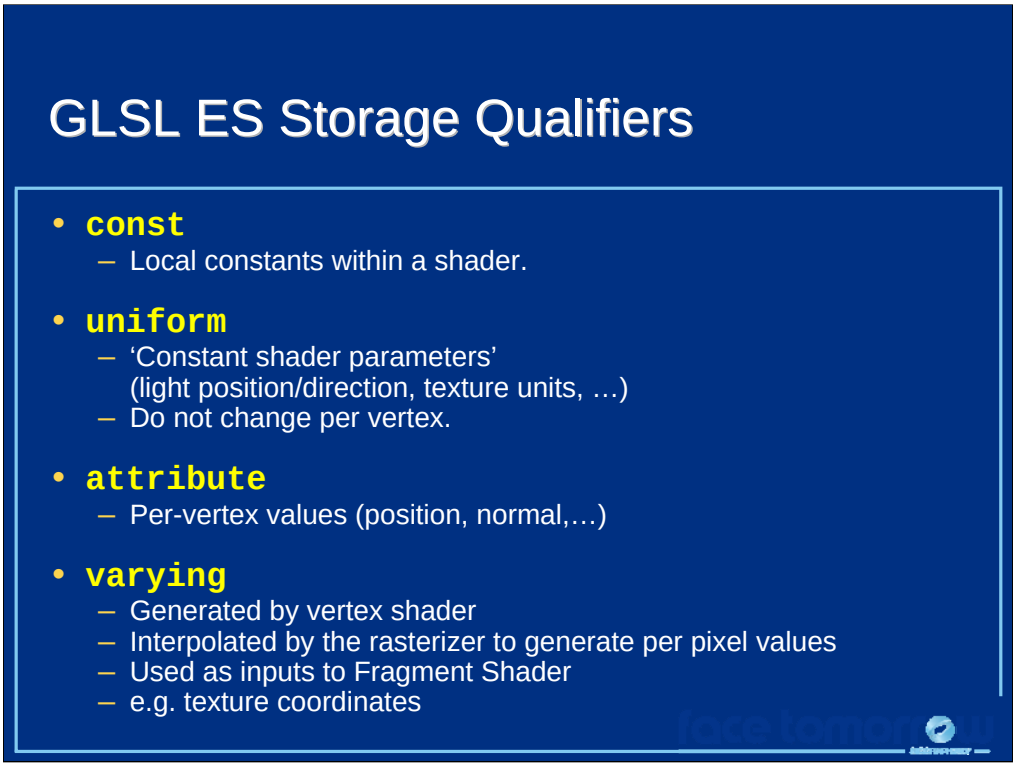


The preprocessor is a simplified version of the c++ preprocessor. Note that as with c++, the preprocessor is not a simple ‘search and replace’ function. Rather, it is integrated with the tokenization of the source code. See the ESSL and c++ specifications for details.

#version and #extension have been added.



These intrinsic types are available and can be used in a similar way to int and float in c++. Note there are several restrictions concerning arrays and structures.



Constants are an integral part of shaders. They cannot be changed and are not visible outside of the shader.

Uniforms are used for values that only need to be changed occasionally. They can only be changed between draw calls and even then changes can be expensive. Typically these are used for values that change per frame such as transforms, light position etc. They can also be used to change the texture being used by a shader.

Attributes are used for values that change frequently, typically for values that change per vertex.

Varyings are used as the interface between the vertex and fragment shaders.

Function parameter Qualifiers

- Used to pass values in or out or both e.g.

```
bool f(in vec2 in_v, out float ret_v)
{
      ...
}
```

- Qualifiers:

`in`	Input parameter. Variable can be modified
`const in`	Input parameter. Variable cannot be modified.
`out`	Output parameter.
`inout`	Input and output parameter.

- Functions can still return a value
  - But need to use a parameter if returning an array

In addition to return values, values may be passed out of functions using out or inout.

Function Parameter Qualifiers

- Call by value ‘copy in, copy out’ semantics.
  - Not quite the same as c++ references:

```
bool f(inout float a, b)
{
      a++;
      b++;
}

void g()
{
      float x = 0.0;
      f(x,x);            // x = 1.0 not 2.0
}
```

When a function is called, all the (in and inout) actual parameters are evaluated and the values copied to the formal parameters. When the function returns, the values are copied back from the formal to the actual parameters. This is slightly different from c++ references. In the above example, a and b are assigned the value 0.0 on entry to the function f. Both have the value 1.0 when the function returns and this value is copied twice into the actual parameter x. Using c++ references would result in x being incremented twice and having a final value of 2.0

GLSL ES Overview

- Order of copy back is undefined

```
bool f(inout float a, b)
{
      a = 1.0;
      b = 2.0;
}

void g()
{
      float x ;
      f(x,x);   // x = 1.0 or 2.0
                // (undefined)
}
```

Precision Qualifiers

- `lowp float`
  - Effectively sign + 1.8 fixed point.
  - Range is -2.0 < x < 2.0
  - Resolution 1/256
  - Use for simple colour blending
- `mediump float`
  - Typically implemented by sign + 5.10 floating point
  - -16384 < x < 16384
  - Resolution 1 part in 1024
  - Use for HDR blending.

Embedded devices need to make maximum use of the available hardware. The single precision in desktop GL has been replaced by 3 precisions for the desktop. Not all implementations will directly support all precisions, they may be mapped to a higher precision. However developers are encouraged to specify lower precisions when they are sufficient.

Precision Qualifiers

- **`highp float`**
  - Typically implemented by 24 bit float (16 bit mantissa)
  - range $\pm 2^{62}$
  - Resolution 1 part in 2^{16}
  - Use of texture coordinate calculation
    - e.g. environment mapping
- single precision (float32)
  - Not explicit in GLSL ES but usually available in the vertex shader (refer to device documentation)

Precision Qualifiers

- Precision depends on the operands:

```
lowp float x;
mediump float y;
highp float z = x * y;
```

(evaluated at medium precision)

- Literals do not have any defined precision

```
lowp float x;
highp float z = x * 2.0 + 1.2;
```

(evaluated at low precision)

Operations involving operands with different precisions are performed at the higher precision. Literals and expressions containing only literals do not have implicit precision. In these cases, the precision of the consuming expression is used to specify the precision of the literals (see specification for details).

Constructors

- Replaces type casting
- No implicit conversion: must use constructors
- All named types have constructors available
  - Includes built-in types, structures
  - Excludes arrays
- Integer to Float:

```
int n = 1;
float x,y;
x = float(n);
y = float(2);
```

Constructors

- Concatenation:

```
float x = 1.0,y = 2.0;
vec2 v = vec2(x,y);
```

- Structure initialization

```
struct S {int a; float b;};
S s = S(2, 3.5);
```

Swizzle operators

- Use to select a set of components from a vector
- Can be used in L-values

```
vec2 u,v;
v.x = 2.0;          // Assignment to single
                    // component
float a = v.x;      // Component selection
v.xy = u.yx;        // swap components
v = v.xx;           // replicate components
v.xx = u;           // Error
```

- Component sets: Use one of

`xyzw` **OR** `rgba` **OR** `stpq`

Note that the maximum size of a vector is a vec4 so e.g. v.xyxyxy is illegal.

Indexing operator

- Indexing operator

```
vec4 u,v;
float x = u[0]; // equivalent to u.x
```

- Must use indexing operator for matrices

```
mat4 m
vec4 v = m[0];
m.x; // error
```

Works the same way as c++. However the compiler will check that any constant indices are within the array bounds.

GLSL ES Overview

- Operators

```
++  --  +  -  ! () []
*  /  +  -
<  <=  >  >=
==  !=
&&  ^^  ||
?:
=  *=  /=  +=  -=
```

- Flow control

```
x == y ? a : b
if else
for while do
return break continue
```

`discard` (fragment shader only)

The discard operation terminates the fragment shader. This can be used to implement e.g. alpha testing.

Built-in Variables

- Aim of ES is to reduce the amount of fixed functionality
  - Ideal would be a totally pure programmable model
  - But still need some
- Vertex shader

```
vec4    gl_Position;    // Write-only
float   gl_PointSize;   // Write-only
```

- Fragment shader

```
vec4    gl_FragCoord;   // Read-only
bool    gl_FrontFacing; // Read-only
vec2    gl_PointCoord;  // Read-only
float   gl_FragColor;   // Write only
```

Although the philosophy of ES 2.0 is to remove fixed functionality, some still remains. The rasterizer is a highly specialized piece of hardware and it is unlikely to be replaced by a programmable unit in the near future. Consequently, there must be a way for the vertex shader to specify the position of the transformed vertex to the rasterizer. This is done using gl_Position. Likewise gl_PointSize for point primitives.

No triangle operations are currently programmable. However the result of back face culling can be useful for 2-sided lighting in the fragment shader so the gl_FrontFacing flag is readable by the fragment shader.

The output of the fragment shader, gl_FragColor is the final pre-defined variable.

Built-in Functions

- General
  pow, exp2, log2, sqrt, inversesqrt
  abs, sign, floor, ceil, fract, mod,
  min, max, clamp
- Trig functions
  radians, degrees, sin, cos, tan,
  asin, acos, atan
- Geometric
  length, distance, cross, dot, normalize,
  faceForward, reflect, refract

GLSL ES Overview

- Interpolations

```
mix(x,y,alpha)
   x*( 1.0-alpha) + y*alpha)
step(edge,x)
   x <= edge ? 0.0 : 1.0
smoothstep(edge0,edge1,x)
   t = (x-edge0)/(edge1-edge0);
   t = clamp( t, 0.0, 1.0);
   return t*t*(3.0-2.0*t);
```

- Texture

```
texture1D, texture2D, texture3D, textureCube
texture1DProj, texture2DProj, textureCubeProj
```

GLSL ES Overview

- Vector comparison (vecn, ivecn)
  ```
  bvecn lessThan(vecn, vecn)
  bvecn lessThanEqual(vecn, vecn)
  bvecn greaterThan(vecn, vecn)
  bvecn greaterThanEqual(vecn, vecn)
  ```
- Vector comparison (vecn, ivecn, bvecn)
  ```
  bvecn equal(vecn, vecn)
  bvecn notEqual(vecn, vecn)
  ```
- Vector (bvecn)
  ```
  bvecn any(bvecn)
  bvecn all(bvecn)
  bvecn not(bvecn)
  ```
- Matrix
  ```
  matrixCompMult (matn, matn)
  ```

Invariance: The Problem

- Mathematical operations are not precisely defined
- Same code may produce (slightly) different results
- Two cases to consider:
  - Invariance within a shader
  - Invariance between shaders

Invariance

- Consider a simple transform in the vertex shader:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

x' = ax + by + cz + dw

But how is this calculated in practice?

- There may be several possible code sequences

Invariance

e.g.

```
MUL R1, a, x
MUL R2, b, y
MUL R3, c, z
MUL R4, d, w
ADD R1, R1, R2
ADD R3, R3, R4
ADD R1, R1, R3
```

or

```
MUL R1, a, x
MADD R1, b, y
MADD R1, c, z
MADD R1, d, w
```

These instruction sequences perform equivalent mathematical operations but due to the limited precision of the hardware, the result will be different. Usually the differences are small but they can be significant.

Invariance

- Three reasons the result may differ:
  - Use of different instructions
  - Instructions executed in a different order
  - Different precisions used for intermediate results (only minimum precisions are defined)
- But it gets worse...

Invariance

- Modern compilers may rearrange your code
  - Values may lose precision when written to a register
  - Sometimes it is cheaper to recalculate a value rather than store it in a register.
    But will it be calculated the same way?

e.g.

```
uniform sampler2D tex1, tex2;
...
const vec2 pos = a + b * c;
vec4 col1 = texture2D(tex1, pos);
...
vec4 col2 = texture2D(tex2, pos); // is this
                                  // the same
                                  // value?
gl_FragColor = col1 – col2;
```

This compiler technique is known as ‘rematerialization’ and is sometimes used to save registers. Registers are in short supply, especially in the fragment shader and spilling a value to external memory is usually not an option due to the significant performance cost.

Invariance

- Solution is in two parts:
  - invariant keyword specifies that specific variables are invariant e.g.

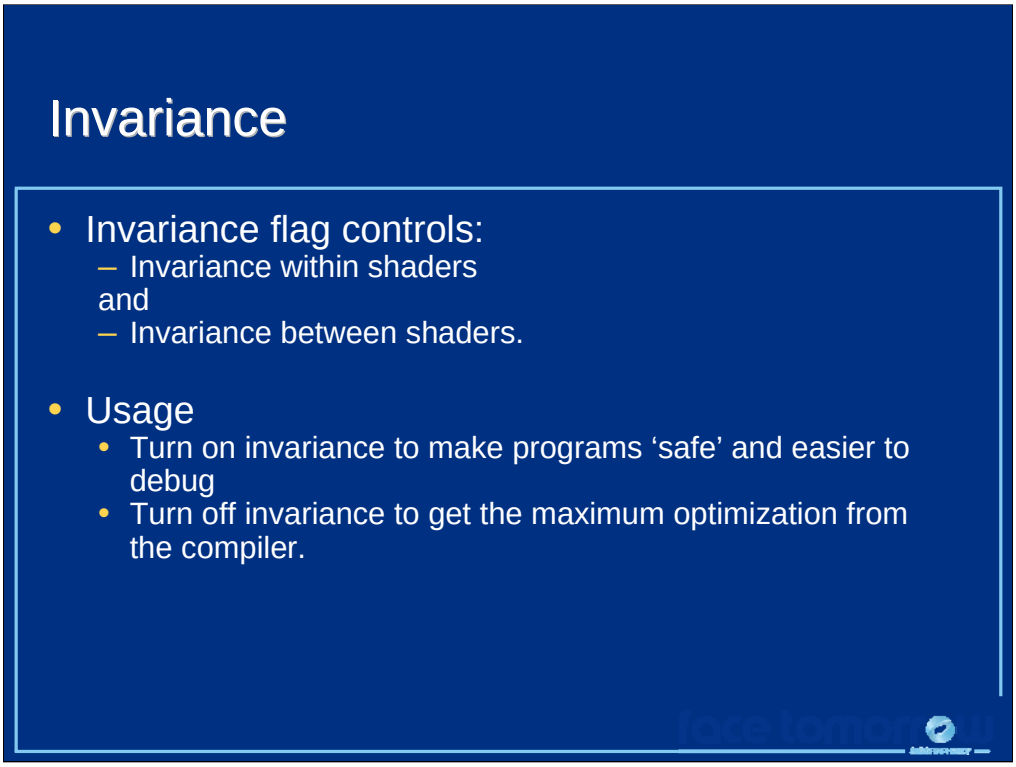
```
invariant varying vec3 LightPosition;
```

    Currently can only be used on certain variables

  - Global switch to make all variable invariant

```
#pragma STDGL invariant(all)
```

An alternative to this approach would be to fully specify the precision and order of operations. This would reduce the scope for compiler optimizations in the many cases where invariance is not important.



Unless invariance is known to be an issue, the developer is recommended to start with invariance turned off. If problems occur which might be due to (lack of) invariance, the #pragma STDGL invariant(all) can be used to quickly check if this is the case. Only then should the developer consider using the invariant qualifier.

Writing an application - Overview

- Initialize EGL
- Setup shader, pipeline state
- Create vertex buffers, textures
- Bind buffers
- Draw

EGL is the interface to the window system for the OS.

Writing an App – Initialization

- Set up EGL
- Compile and Link shaders
- Create and bind Textures
- Bind (or get) attributes
- Set up uniforms
- Create Vertex Buffers
- Map buffer data

Looking at the intial steps in more detail. After initializing EGL, all the input parameters to the shaders are initialized. This corresponds to the types of data mentioned before i.e.:

Source code

Textures

Attributes

uniforms

Writing an App – EGL Initialization

```
EGLDisplay egl_display =
eglGetDisplay(EGL_DEFAULT_DISPLAY);

int ok = eglInitialize(egl_display,
                       &majorVersion,
                       &minorVersion)
```

eglGetDisplay gets a connection to the display device being used. EGL_NO_DISPLAY is returned if there is no matching display.

eglInitialize initializes the display and returns the version of EGL.

EGL Initialization

Set up attributes for EGL context

```
EGLint attr[MAX_EGL_ATTRIBUTES];

attr[nAttrib++] = EGL_RED_SIZE;
attr[nAttrib++] = 5;
...
attr[nAttrib++] = EGL_DEPTH_SIZE;
attr[nAttrib++] = 16;
attr[nAttrib++] = EGL_STENCIL_SIZE;
attr[nAttrib++] = 0;
...
```

EGL Initialization (cont)

```
eglChooseConfig(egl_display,
                attrib_list,
                &egl_config,
                1,
                &num_configs);

eglCreateWindowSurface(egl_display,
                       egl_config,
                     NativeWindowType (hWnd),
                       NULL);
```

eglChooseConfig returns a list of all EGL frame buffer configurations that match specified attributes. In the above example, the value ‘1’ is used to force the API to return a maximum of one configuration.

eglCreateWindowSurface creates a new EGL window surface.
EGL_NO_SURFACE is returned if the call fails.

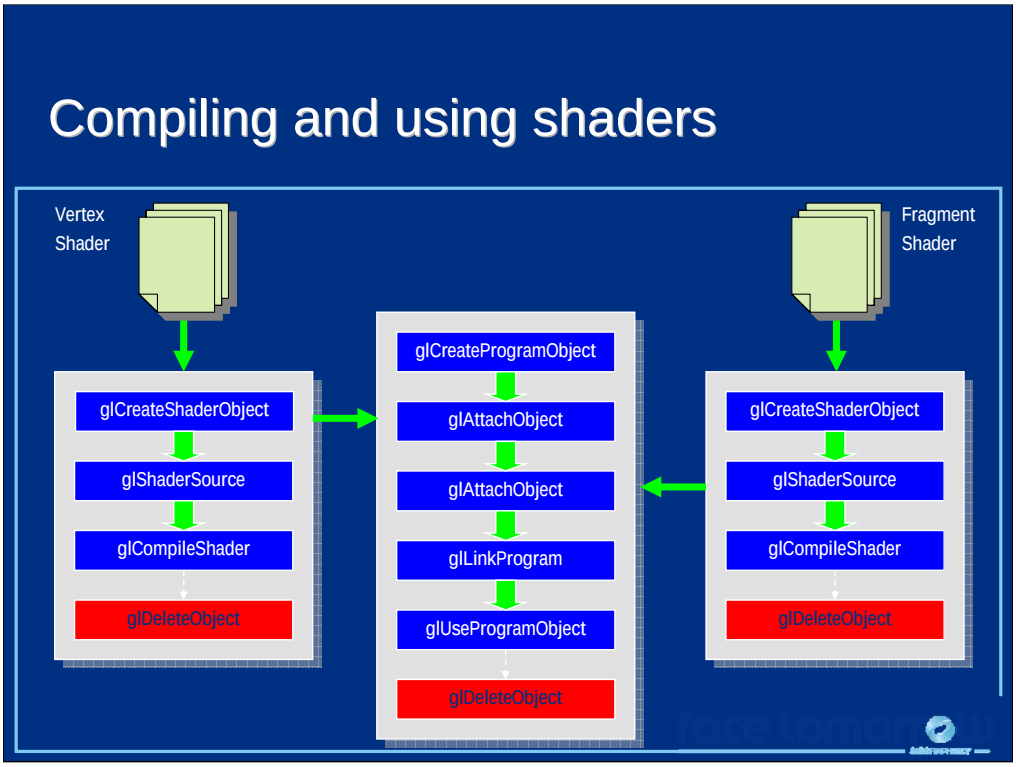
EGL Initialization: Creating a context

```
context = eglCreateContext(egl_display,
                           egl_config,
                           EGL_NO_CONTEXT,
                           NULL);

eglMakeCurrent(egl_display,
               egl_surface,
               egl_surface,
               egl_context);
```

eglCreateContext creates a new EGL rendering context. EGL_NO_CONTEXT is returned if the call fails. The EGL_NO_CONTEXT is used to specify the surface is not being shared (to simplify this example).

eglMakeCurrent attaches an EGL rendering context to the EGL surface



This shows the flowchart for compiling and linking shaders. GL is based on objects and each object must be explicitly created and deleted.

glShaderSource specifies the source code to be compiled. In principle, the two shaders are compiled separately and then linked together in a similar way to source files for a c++ program. However it should be appreciated that most of the compilation may actually occur at link time since this is when all the information about the program is available to the compiler.

Compiling and Linking Shaders

- Create objects

```
program_handle = glCreateProgram();
// Create one shader of object of each type.
GLuint vertex_shader_handle
     = glCreateShader (GL_VERTEX_SHADER);
GLuint fragment_shader_handle
     = glCreateShader (GL_FRAGMENT_SHADER);
```

And here is example code showing how the functions are used.

Compiling Shaders

- Compile vertex shader (and fragment shader)

```
char* vert_source = ...
const char* vert_gls[1] = {vert_source};
glShaderSource(vertex_shader_handle,
               1,
               vert_gls,
               NULL );
glCompileShader(vertex_shader_handle);
GLint vertCompilationResult = 0;
glGetShaderiv(vertex_shader_handle,
              GL_COMPILE_STATUS,
              &vertCompilationResult);
```

Linking Shaders

- Attach shaders to program object and link

```
glAttachShader(program_handle,
               vertex_shader_handle);
glAttachShader(program_handle,
               fragment_shader_handle);
glLinkProgram (program_handle);
```

- Note that many compilers will only report errors at link time.

Setting up Attributes

- Can bind attributes before linking e.g.

```
glBindAttribLocation (prog_handle, 0, “pos");
```

- Or get attribute location after linking:

```
GLint p;
p = glGetAttribLocation (prog_handle, “pos");
```

- Can do a combination.

Attributes can be specified by the user or left to the driver. If the application can benefit from having the attributes in a particular order then the glBindAttributeLocation should be used. Otherwise it is usual to use glGetAttribLocation.

Setting up Textures

- Texture samplers are ***Uniforms*** in GLSL ES
- First Generate ID and specify type (cube map)

```
uint32 Id;
glGenTextures(1, &Id);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, Id);
```

glGenTextures generates the texture _names_ not the actual textures.

The glActiveTexture call is used to specify which texture unit subsequent calls will affect. GL_TEXTURE0 refers to texture unit 0 (units 0 to 7 are available).

glBindTexture specifies the type of texture (in this case a cube map) and associates a name with the texture.

Setting up Textures (cont)

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X,
             0,
             GL_RGBA,
             width,
             height,
             0,
             GL_RGBA,
             GL_UNSIGNED_BYTE,
             image [0].pixels);

glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, ...
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, ...
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, ...
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, ...
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, ...
```

Having specified the texture stage and type of texture, glTexImage2D is the call used to specify the actual texture data.

Since we are dealing with a cube map, there are 6 separate calls, one for each of the faces of the cube map.

Setting up Uniforms

- Must do this after glUseProgram:

```
glUseProgram(prog_handle);
```

- Use glGetUniformLocation e.g.

```
GLint loc_sky_box =
  glGetUniformLocation (prog_handle,"skyBox“);
```

- Can then set value e.g.

```
GLint texture_unit = 0;
glUniform1i (loc_sky_box,texture_unit);
```

The current (active) program is first specified with glUseProgram. The program is then queried for the locations of the uniforms so that they can be initialized.

Setting up Attribute Buffers

- Create buffer names

```
GLuint bufs[1];
glGenBuffers (1, bufs);
```

- Create and initialize buffer

```
glBindBuffer (GL_ARRAY_BUFFER,
  bufs[0]);
glBufferData (GL_ARRAY_BUFFER,
  size_bytes, p_data, GL_STATIC_DRAW);
```

The setting up of attributes is similar to the way textures are set up. Names must be generated for all the attributes. The glBindBuffer specifies the name of the buffer to use and then the actual buffer data is specified using glBufferData.

Setting up Attribute Buffers (cont)

- Specify format:

```
glBindBuffer(GL_ARRAY_BUFFER, bufs[0]);
glVertexAttribPointer(0,       // index
                      4,       // size
                      GL_FLOAT,// type
                      GL_FALSE,// norm
                      0,
                      NULL );
```

The glBufferData call doesn't say anything about the format of the data, it is just a byte array at this point. The glVertexAttribPointer is used to specify the format. In this case we are specifying vector-4 floats for attribute index 0.

Note the extra glBindBuffer is not necessary unless GL_ARRAY_BUFFER has been bound to another buffer.

Drawing the frame

- Clear frame buffer
- Set render state
- Enable array
- DrawArray

This shows the minimum that must be done at draw time. The frame buffer is usually cleared at the start of each frame. Any fixed function render state must be set up and then the scene can be drawn.

Drawing

- Enable array and Draw

```
glEnableVertexAttribArray( 0 );
glBindBuffer (GL_ARRAY_BUFFER,0);

glDrawArrays (GL_TRIANGLE_STRIP,0,
              n_vertices);
```

glEnableVertexAttribArray must be called for each attribute used in a draw call. In this example we are just enabling attribute 0.

The glBindBuffer with a parameter 0 is used to unbind the buffer.

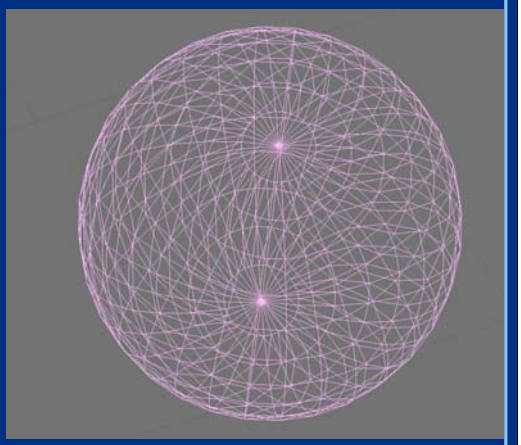
The glDrawArrays call does the actual drawing.

Example: Water demo

This is a simple demo showing how to implement reflections in water. Although similar techniques can be implemented with ES 1.1, there are some subtleties that require shaders.

Skybox

- Geometry is a sphere
- Use position to access a cube map



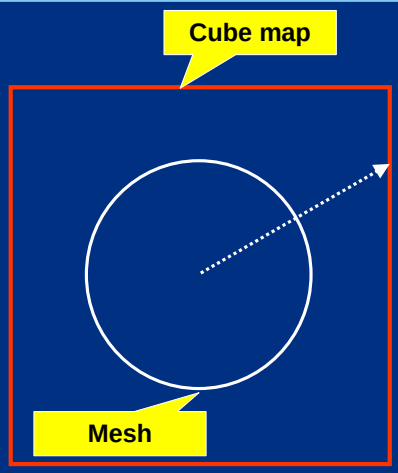
The technique used here is to position the viewer inside a sphere. A cube map is mapped onto the sphere. The cube map is used in two ways: firstly as the skybox and secondly to provide something to reflect in the water.

Cube Map

The cube map is stored a 6 separate images.

Skybox

- Can use position to access cube map
- Don’t need to normalize.
- No need for separate normals



The skybox is a simple sphere and the cube map can be accessed very simply using the position of the vertices in the sphere. No need for separate normals.

Sky box: Vertex shader

```
uniform mat4 view_proj_matrix;
uniform vec4 view_position;
attribute vec4 rm_Vertex;
varying vec3 vTexCoord;
void main(void)
{
  vec4 newPos = vec4(1.0);
  newPos.xyz  = rm_Vertex.xyz + view_position.xyz;
  gl_Position = view_proj_matrix * vec4(newPos.xyz, 1.0);
  vTexCoord = rm_Vertex.xyz;
}
```

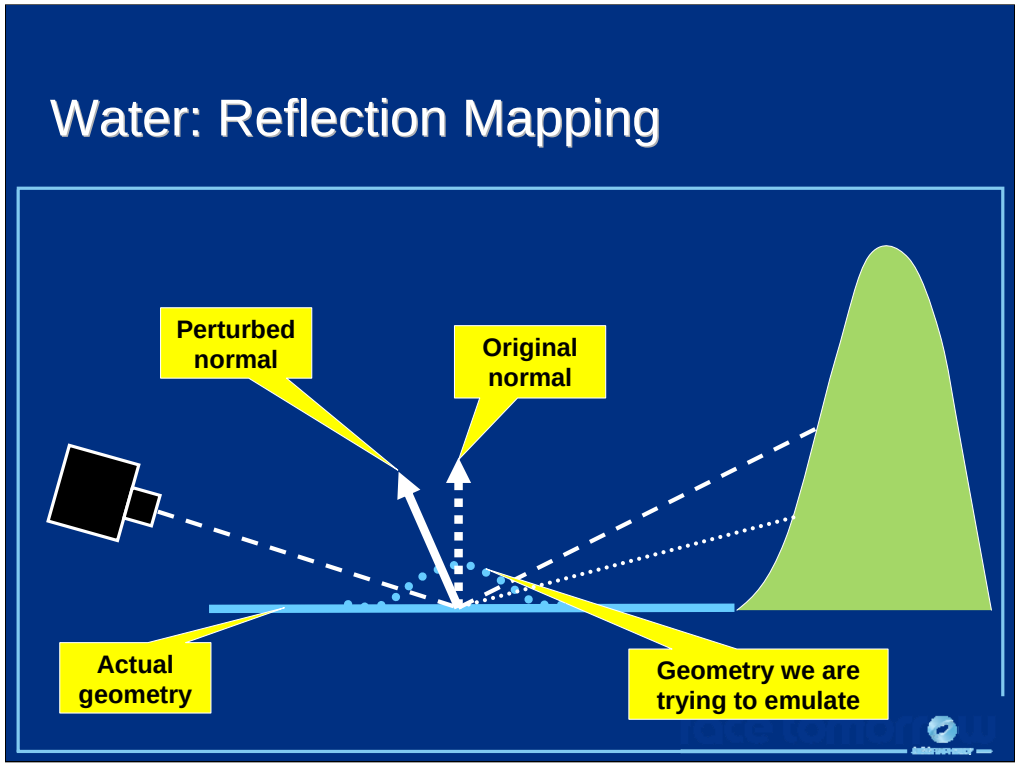
The vertex shader performs only a simple transform. The view position is separated from the rest of the matrix in this example although this is not necessary.

Note that the texture coordinates are simply the original vertex positions.

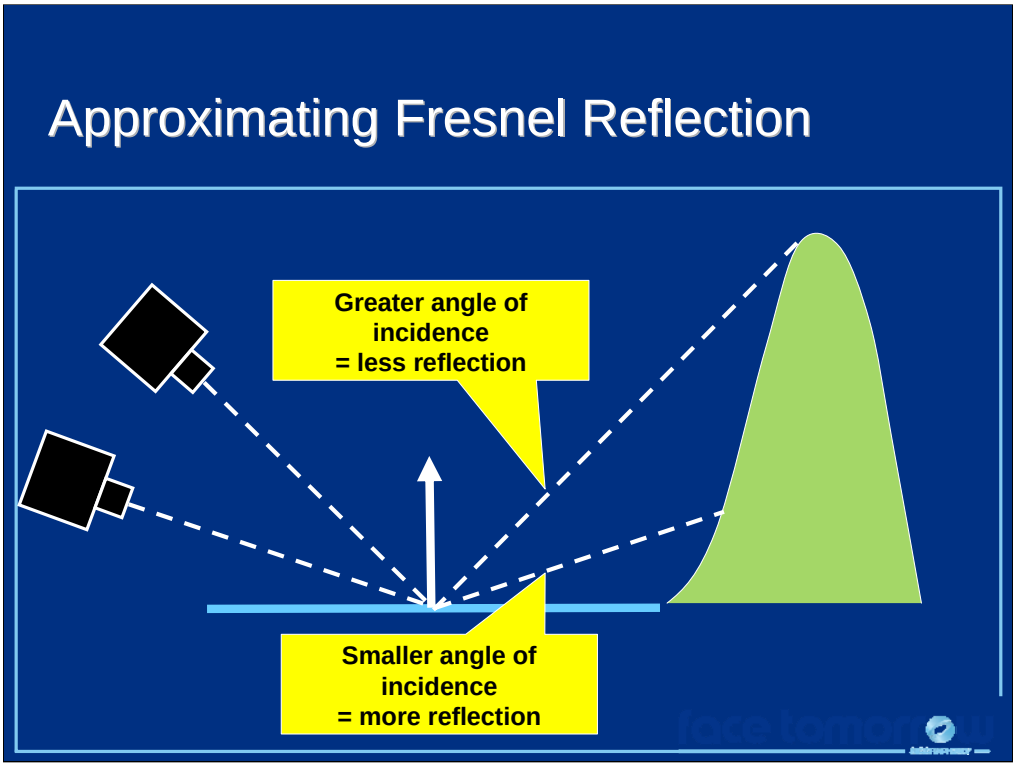
Sky box: Fragment Shader

```
precision highp float;
uniform samplerCube skyBox;
varying vec3 vTexCoord;
void main(void)
{
  gl_FragColor =
          textureCube(skyBox,vTexCoord);
}
```

All the fragment shader does is a texture lookup.



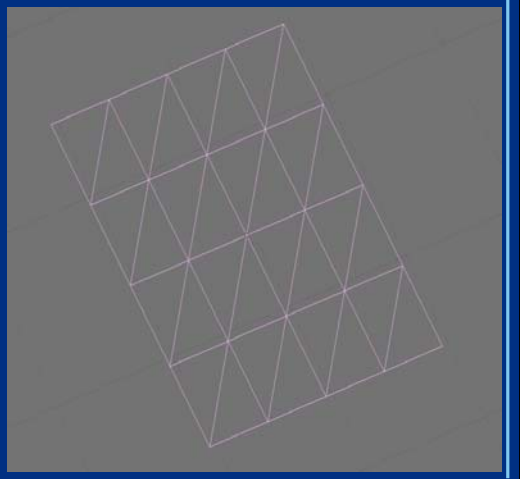
The viewer (camera) is on the left. The water is drawn as a flat surface (i.e. there is no geometry representing the ripples). To emulate the ripples, the normal is perturbed and this is used to modify the reflection vector.



This is where we leave the world of fixed function. In order to correctly render reflections, the amount of light reflected depends on the angle of incidence. The lower the angle, the more light is reflected.

Water

- Geometry is a simple grid
- Uses the same cubemap as the skybox



The water is drawn as a flat surface

Water Ripples

- Use noise texture for bump map.
- Exact texture not important
  - Try experimenting

For the ripple effect a noise texture is used. In this example no attempt is made to simulate the way ripples move. Rather the noise texture is moved across the water with a constant velocity.

Water: Vertex Shader

```
uniform vec4 view_position;
uniform vec4 scale;
uniform mat4 view_proj_matrix;
attribute vec4 rm_Vertex;
attribute vec3 rm_Normal;
varying vec2 vTexCoord;
varying vec3 vNormal;
varying vec3 view_vec;
```

Declaration of all the uniforms and varyings.

Water: Vertex Shader (cont)

```
void main(void)
{
  vec4 Position = rm_Vertex.xyzw;
  Position.xz *= 1000.0;
  vTexCoord   = Position.xz * scale.xz;
  view_vec    = Position.xyz -
                view_position.xyz;
  vNormal     = rm_Normal;
  gl_Position = view_proj_matrix * Position;
}
```

This is just a simple transform. The water rectangle is scaled up but this could equally be done outside the shader.

Water: Fragment Shader

```
uniform sampler2D noise;
uniform samplerCube skyBox;
uniform float time_0_X;
uniform vec4  waterColor;
uniform float fadeExp;
uniform float fadeBias;
uniform vec4  scale;
uniform float waveSpeed;
varying vec2 vTexCoord;
varying vec3 vNormal;
varying vec3 vViewVec;
```

Declaration of all the uniforms and varyings.

Water Fragment Shader (cont)

```
void main(void)
{
   vec2 tcoord = vTexCoord;
   tcoord.x += waveSpeed  * time_0_X;
   vec4 noisy = texture2D(noise, tcoord.xy);
   // Signed noise
   vec3 bump = 2.0 * noisy.xyz - 1.0;
   bump.xy *= 0.15;

   // Make sure the normal always points upwards
   bump.z = 0.8 * abs(bump.z) + 0.2;
```

The x component of the texture is changed at constant speed to emulate the moving of the ripples.

The noise texture is signed so must be mapped onto a positive range.

The various factors used are best found by experimenting. Care should be taken to make sure the normal is always pointing upwards. Only small perturbations are required for a good effect.

Water Fragment Shader (cont)

```
// Offset the surface normal with the bump
bump = normalize(vNormal + bump);

// Find the reflection vector
vec3 reflVec = reflect(vViewVec, bump);
vec4 refl = textureCube(skyBox, reflVec.yzx);
```

The bump vector is used to perturb the normal.

This modified normal is then used to find the reflection vector which in turn is used to access the same texture cubemap that was used for the skybox.

Water Fragment Shader (cont)

```
  float lrp =
   1.0 - dot(-normalize(vViewVec), bump);

  // Interpolate between the water color and
  // reflection
  float blend = fadeBias + pow(lrp, fadeExp);
  blend = clamp(blend ,0.0, 1.0);
  gl_FragColor = mix(waterColor, refl, blend);
}
```

Finally the reflected colour is blended with the water colour. When the bump vector moves the normal towards the viewer, this increases the angle of incidence. The shader therefore decreases the ‘blend’ value which increases the contribution of the water colour.

Since there is a fair amount of ‘cheating’ going on in this example, it is important to constrain certain values to prevent artefacts. Hence the blend factor is clamped to the range 0..1

Programming Tips

- Check for errors regularly
  - Use e.g.

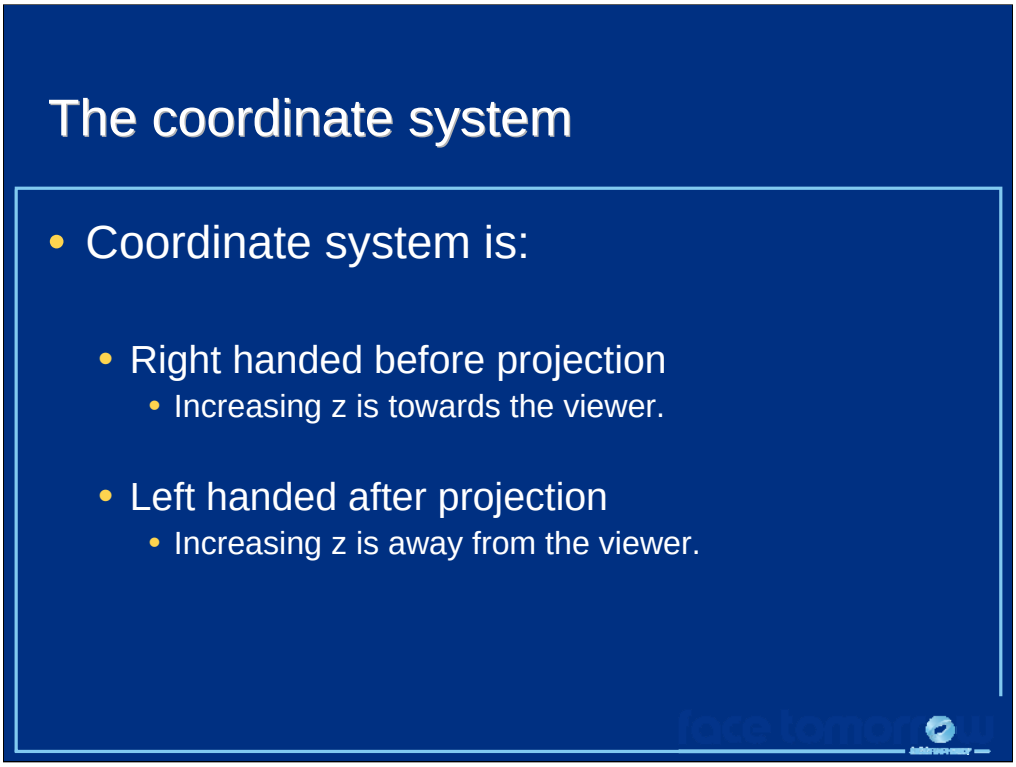
```
assert(!glError ());
```

  - But remember glError () gets the last error:

```
... // error here
Glint error = glError ();
...
assert(!glError ()); // No error
```

The glError() reads the last error code. If there has only been one error, subsequent calls to glError() will return 0. It is therefore easy to miss errors if glError() is used incorrectly.

Debugging GL applications can be a challenge when writing your first application and it is easy to introduce errors. Be sure to check every value returned and always check for errors.



The coordinate system often confuses newcomers.

Matrix Convention

- Matrices are column-major
  - column index varies more slowly
- Vectors are columns
- But this is purely convention
- Only the position in memory is important
  - Translation specified in elements 12,13,14

The projection matrix

- You need to provide a projection matrix e.g.

$$\begin{bmatrix} \frac{2.0*near}{right-left} & 0.0 & \frac{right+left}{right-left} & 0.0 \\ 0.0 & \frac{2.0*near}{top-bottom} & \frac{top+bottom}{top-bottom} & 0.0 \\ 0.0 & 0.0 & \frac{-(far+near)}{far-near} & \frac{-2.0*far*near}{far-near} \\ 0.0 & 0.0 & -1.0 & 0.0 \end{bmatrix}$$

- near and far are both *positive*

Performance Tips

- Keep fragment shaders simple
  - Fragment shader hardware is expensive.
  - Early implementations will not have good performance with complex shaders.
- Try to avoid using textures for function lookups.
  - Calculation is quite cheap, accessing textures is expensive.
  - This is more important with embedded devices.

Performance Tips (cont)

- Minimize register usage
  - Embedded devices do not support the same number of registers compared with desktop devices. Spilling registers to memory is expensive.
- Minimize the number of shader changes
  - Shaders contain a lot of state
  - May require the pipeline to be flushed
  - Use uniforms to change behaviour in preference to loading a new shader.

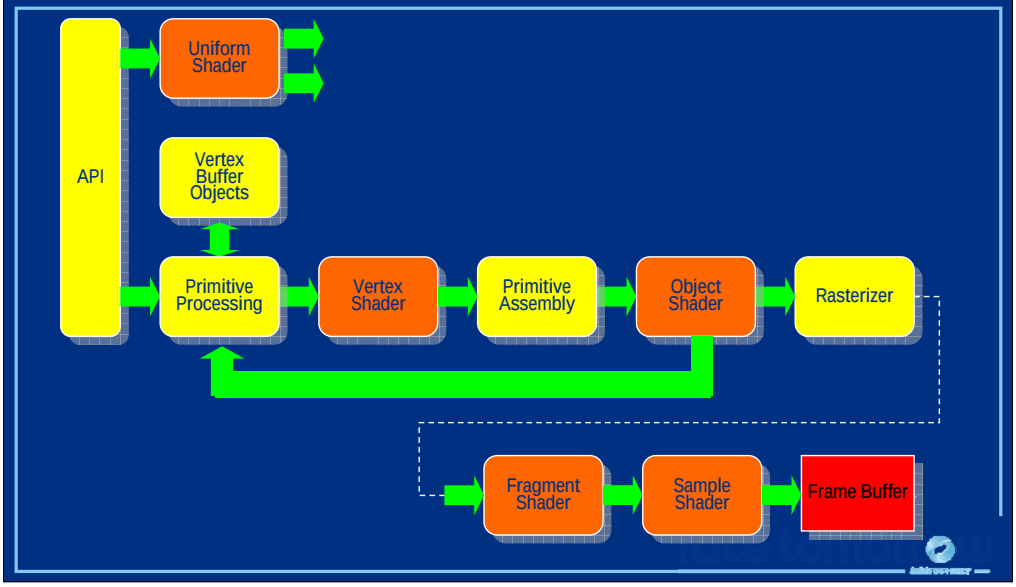
Future Directions

- Sample Shaders
  - Enables alpha testing at per-sample resolution
  - Enables more of the fixed function pipeline to be removed.
  - Allows more programmability when using multi-sampling.
  - e.g. Read and write depth and stencil

Future Directions

- Object (Geometry) Shaders
  - Programmable tessellation
  - Higher order surfaces
  - Procedural geometry
  - Possibility of accelerating many more algorithms e.g. shadows, occlusion culling.

Future ES Pipeline?




SIGGRAPH2007

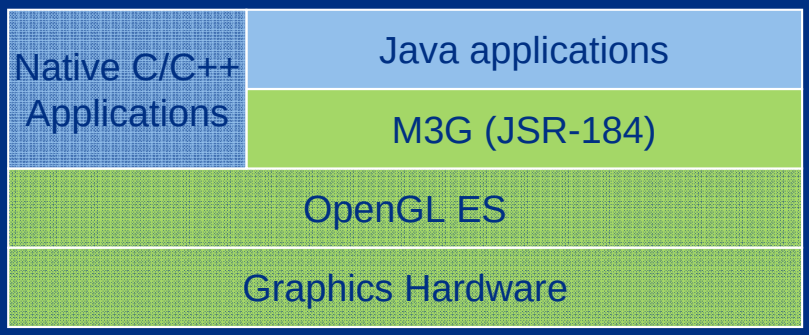
M3G Intro

SIGGRAPH2007

Kari Pulli

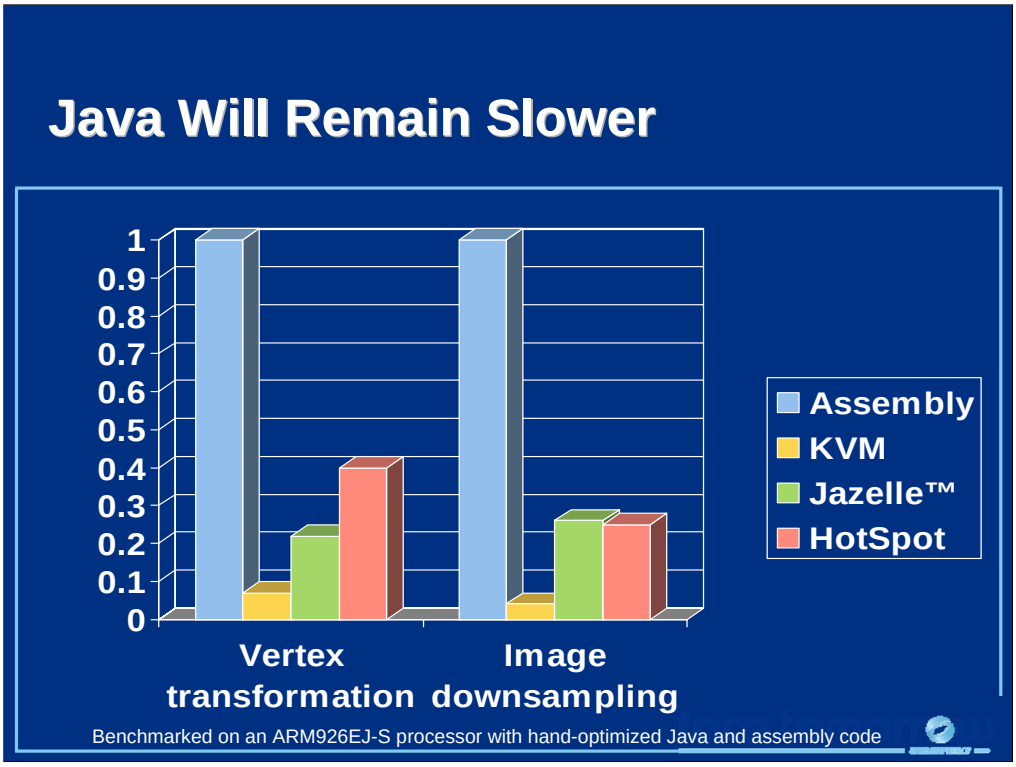
Nokia Research Center

Mobile 3D Graphics APIs



Why Should You Use Java?

- Largest and fastest growing installed base
  - 1200M phones running Java by June 2006
  - The majority of phones sold today support Java
- Better productivity compared to C/C++
  - Much fewer opportunities to introduce bugs
  - Comprehensive, standardized class libraries



But of course there are problems too. Java has a reputation of being slow, and that's certainly true for mobile phones.

To give you an idea, this graph here compares three different Java virtual machines against assembly code.

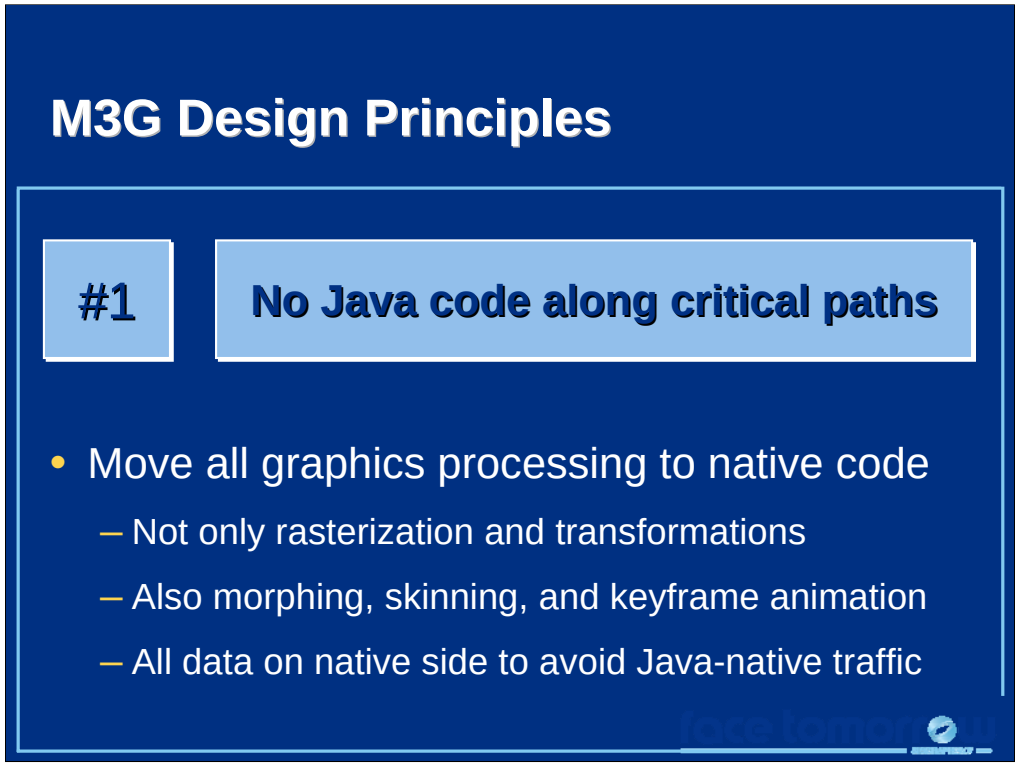
The tallest bars represent ARM assembly code, with relative performance of 1.0. No special CPU features, such as SIMD instructions, were used. If they were, the difference to Java would be much larger.

First, we have the KVM, which was used in most mobile phones until recently. Native code is 10-20x faster.

Then we have Jazelle, which is a hardware accelerator from ARM. Big improvement, but native code is still 3-4x faster.

Finally we have a HotSpot VM from Sun. It matches Jazelle in these benchmarks, but in real life, it's a disaster. The compiler and the compiled code together take up so much RAM that you can only keep the most frequently and most recently used pieces of code in cache. So, when you encounter a new monster in an action game, the compiler kicks in and the game freezes for half a second. Developers have to use some ugly tricks to work around these problems.

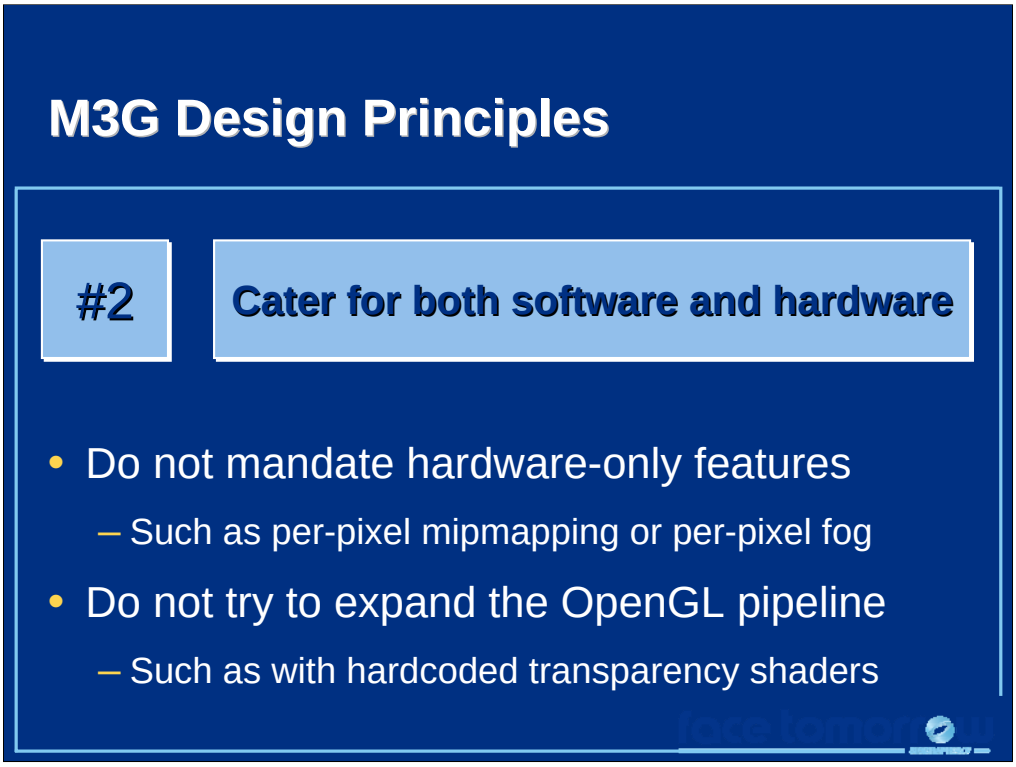
Ahead-of-time (AOT) compilers are not included in this benchmark. AOT



So with that background in mind, let's see what our main design principles were.

The most important thing of course is to free the apps from doing rasterization and transformations in Java. That's simply too slow.

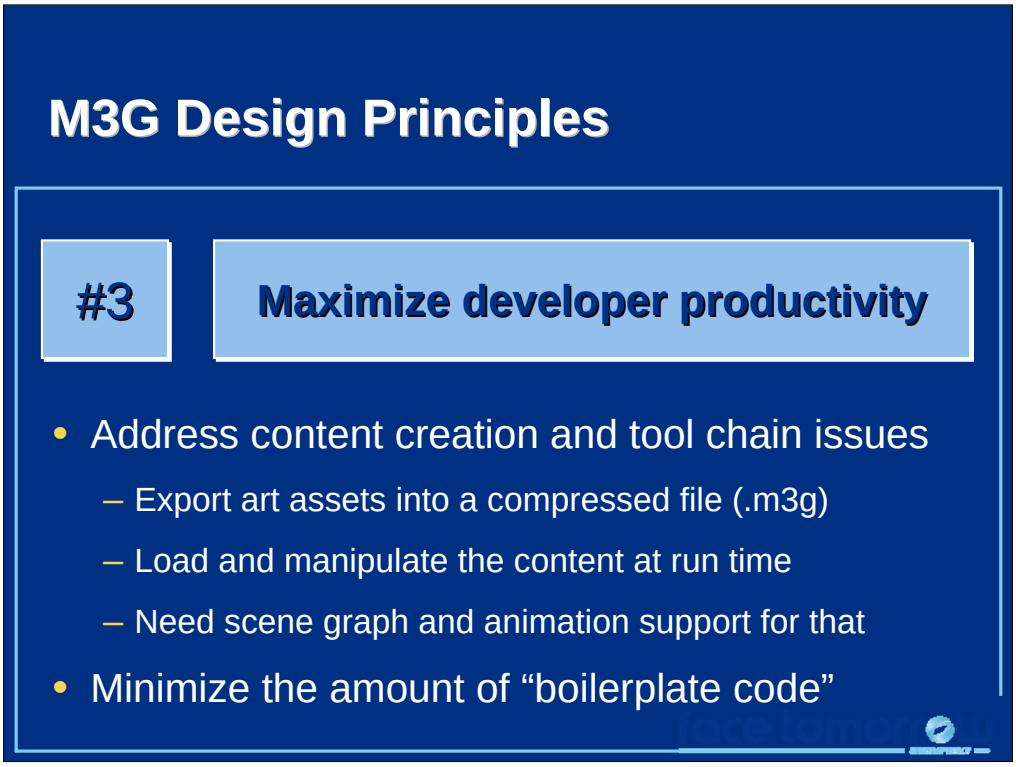
But when we have those in native code, then other things become the bottlenecks. So, we decided to go for a retained mode, scene graph API and keep all scene data on the native side. We also decided to include all functionality that can be generalized well enough. As a result, we have things like morphing, skinning and keyframe interpolation in the API.



Secondly, we wanted the API to work well on software-based handsets, which form the vast majority even today, as well as the hardware-accelerated ones.

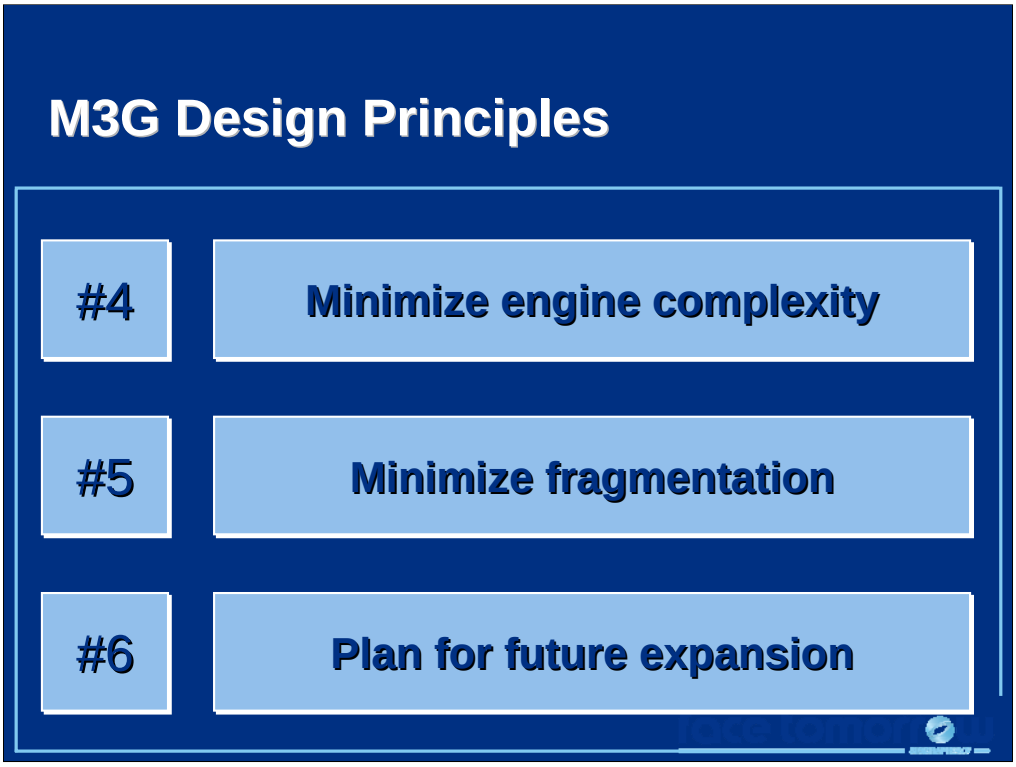
We had a rule that features that cannot be done efficiently in software will not be required or even included. Almost anything that is computed “per pixel” falls into that category. Thus, it suffices to select a mipmap level on a per-triangle basis, rather than per-pixel. The same applies for fog. Bilinear and trilinear texture filtering are also optional.

On the other hand, we had a rule that no feature would be included that cannot be easily implemented on fixed-function hardware, even if it would be a useful feature and easy to do in software. Various hardcoded effects for e.g. transparency and reflection were proposed, but rejected on that basis.



Third, we didn’t want to leave content creation and tool chain issues hanging in the air. We wanted to have a well-defined way of getting stuff out from 3dsmax and other tools, and manipulating that content at run time. That’s of course another reason to have scene management and animation features in the API. We also defined a file format that matches the features one-to-one.

Furthermore, we wanted the API to be at a high enough level that not much “boilerplate” code needs to be written to get something done.



Here are some more design issues that we had to keep in mind.

Number four, minimize engine complexity. This meant that a commercial implementation should be doable in 150k, including the rasterizer.

Number five, minimize fragmentation. This means that we wanted to have a tight spec, so that you don't have to query the availability of each and every feature. There are no optional parts or extensions in the API, although texture filtering and some other rendering quality hints were left optional. For instance, perspective correction.

And finally, we wanted to have a compact API that can be deployed right away, but so that adding more features in the future won't cause too much ugly legacy. Several features that seemed likely to be soon deprecated were dropped on that basis (e.g. logic ops and points). Some predictions went wrong: for example, lines have not yet been replaced by triangles…

Why a New Standard?

- OpenGL ES is too low-level
  - Lots of Java code and function calls needed
  - No support for animation and scene management
- Java 3D is too bloated
  - A hundred times larger (!) than M3G
  - Still lacks a file format, skinning, etc.

Okay, so why did we have to define yet another API, why not just pick an existing one?

OpenGL ES would be the obvious choice, but it didn't fit the Java space very well, because you'd need a lot of that slow Java code to get anything on the screen. Also, you'd have to do animation yourself, and keep all your scene data on the Java side. Basically you'd spend more time writing your code, and yet the code would run slower in the end. That might change in the future, when Java VMs become faster, but don't hold your breath.

The other choice that we had was Java 3D. At first it seemed to match our requirements, and we gave it a serious try. But then it turned out that the structure of Java 3D was simply too bloated, and we just couldn't simplify it enough to fit out target devices. Besides, even though the Java 3D is something like a hundred times larger than M3G, it still lacks crucial things like a file format and skinning. It's also too damn difficult to use.

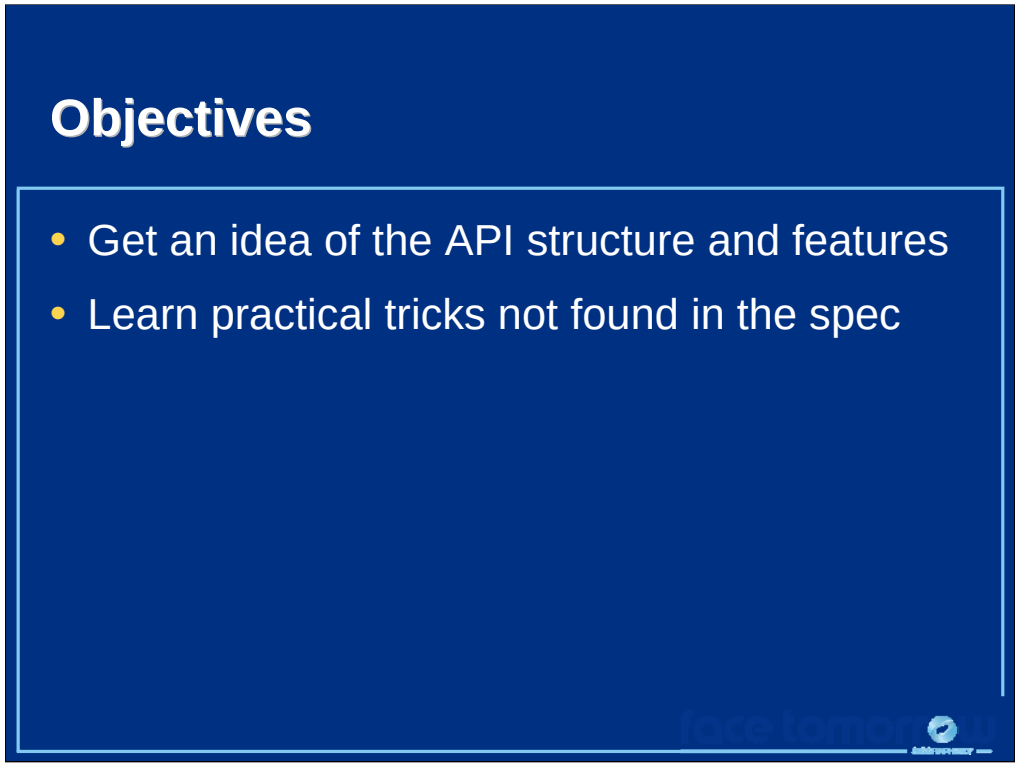
So we decided to re-invent the wheel. Let's see how it works.


SIGGRAPH2007

M3G API Overview

Tomi Aarnio

Nokia Research Center



After this session you should have a good idea of what features you can find in the API, and have some tricks up your sleeve on how to use those features effectively on real devices.

Prerequisites

- Fundamentals of 3D graphics
- Some knowledge of OpenGL ES
- Some knowledge of scene graphs

M3G API Overview

Getting started

Rendering

Scene graph

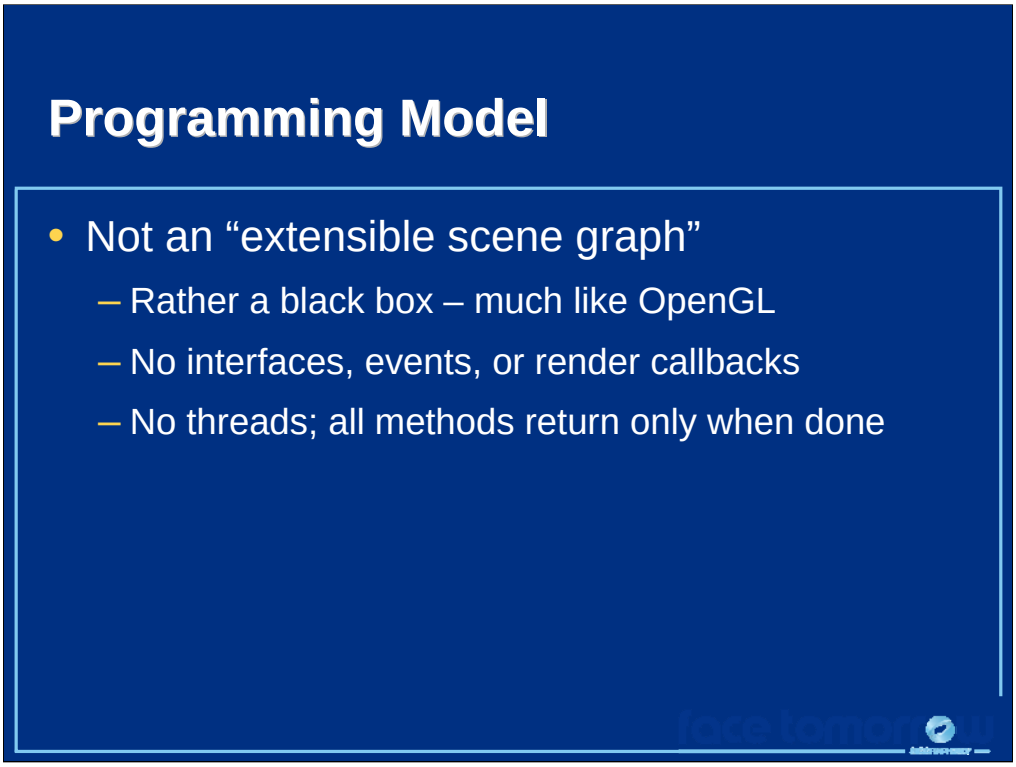
Performance tips

Deformable meshes

Keyframe animation

Demos

Let’s first take a look at the M3G programming model, then continue with the features in a bottom-up order.



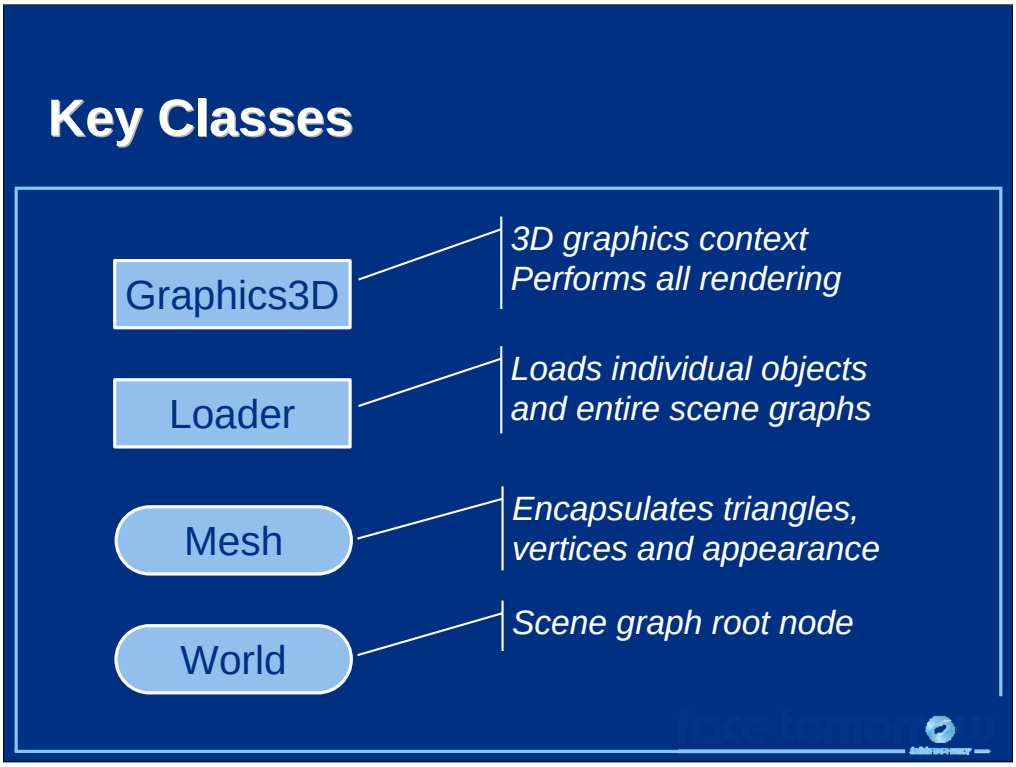
M3G is a fairly simple, monolithic API. It’s not the usual “extensible scene graph”, but rather a black box. There are no rendering callbacks, no events, no interfaces, and no background threads. This means that you can’t add your own custom objects into the scene graph and expect the engine to call your draw() routine in the middle of the rendering traversal. In this respect, M3G is very much like OpenGL: you don’t expect callbacks from glDrawElements, either.

Programming Model

- Scene update is decoupled from rendering
  - **`render`** ➔ Draw the scene, no side-effects
  - **`animate`** ➔ Update the scene to the given time
  - **`align`** ➔ Re-orient target cameras, billboards

Also in the good black-box tradition, when you tell the API to render something, it does just that, with no side-effects. It doesn't change the scene graph. When you need to change something, you call animate() or align() or you use the individual set methods.

Key Classes


Graphics3D
3D graphics context
Performs all rendering
Loader
Loads individual objects
and entire scene graphs
Mesh
Encapsulates triangles,
vertices and appearance
World
Scene graph root node

Graphics3D: How to Use

- Bind a target to it, render, release the target

```
void paint(Graphics g) {
   myGraphics3D.bindTarget(g);
   myGraphics3D.render(world);
   myGraphics3D.releaseTarget();
}
```

So how do you use it? It's as easy as 1-2-3: bind a target, render, release the target. As shown here.

The "Graphics" object represents the frame buffer in Java MIDP.

In the debug build you should also catch any exceptions after each bind, render, and release call.

Rendering State

- Graphics3D contains global state
  - Frame buffer, depth buffer
  - Viewport, depth range
- Most rendering state is in the scene graph
  - Vertex buffers, textures, matrices, materials, …
  - Packaged into Java objects, referenced by meshes
  - Minimizes Java-native data traffic, enables caching

M3G API Overview

Getting started

Rendering

Scene graph

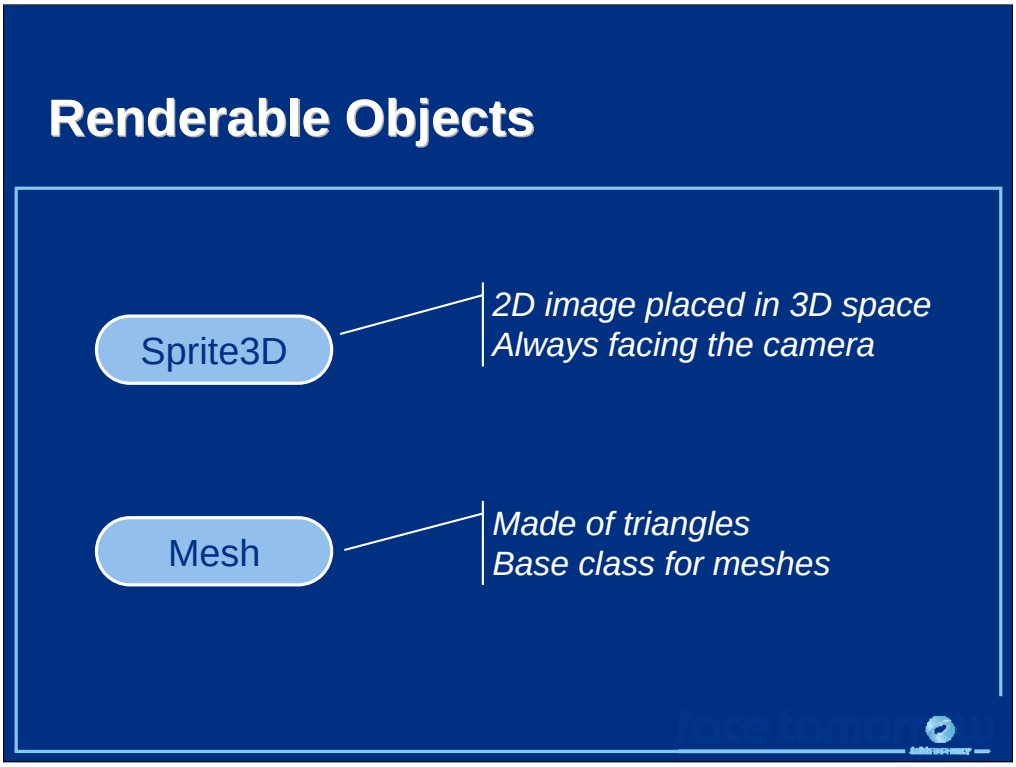
Performance tips

Deformable meshes

Keyframe animation

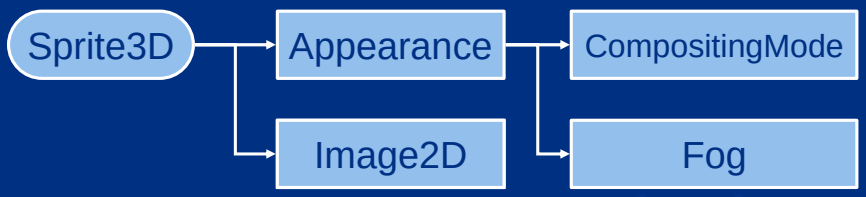
Demos

Renderable Objects



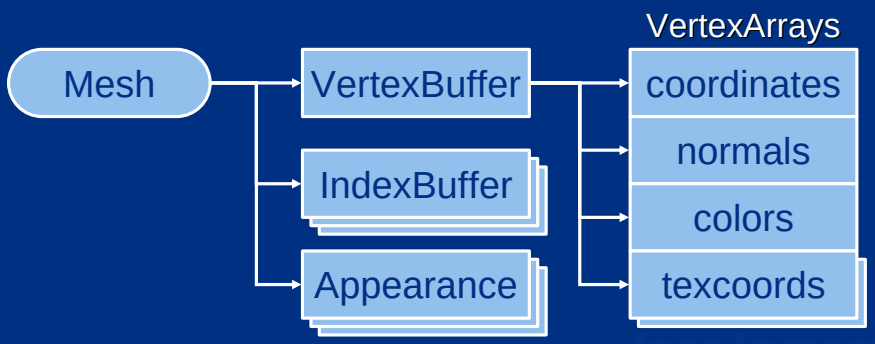
Sprite3D

- 2D image with a position in 3D space
  - Scaled mode for billboards, trees, etc.
  - Unscaled mode for text labels, icons, etc.
  - Too much overhead for particle effects



Mesh

- One VertexBuffer, containing VertexArrays
- 1..*N* submeshes (IndexBuffer + Appearance)



IndexBuffer Types

	Byte	Short	Implicit	Strip	Fan	List
Triangles	✓	✓	✓	✓	✗	✗
Lines	✗	✗	✗	✗	✗	✗
Points	✗	✗	✗			✗
Point sprites	✗	✗	✗			✗

Relative to OpenGL ES 1.1

The set of rendering primitives was reduced to a minimum: triangle strips with 16-bit indices (equivalent to glDrawElements) or implicit indices (glDrawArrays).

On hindsight, triangle lists should've been included, since they are easier to use and are not necessarily any slower than strips.

Point sprites are missing for a good reason: The M3G spec had been publicly available for almost a year before point sprites were added into OpenGL ES.

VertexBuffer Types

	Byte	Short	Fixed	Float	2D	3D	4D
Vertices	✓	✓	✗	✗	✗	✓	✗
TexCoords	✓	✓	✗	✗	✓	✓	✗
Normals	✓	✓	✗	✗		✓	
Colors	✓		✗	✗		✓*	✓
PointSizes			✗	✗			

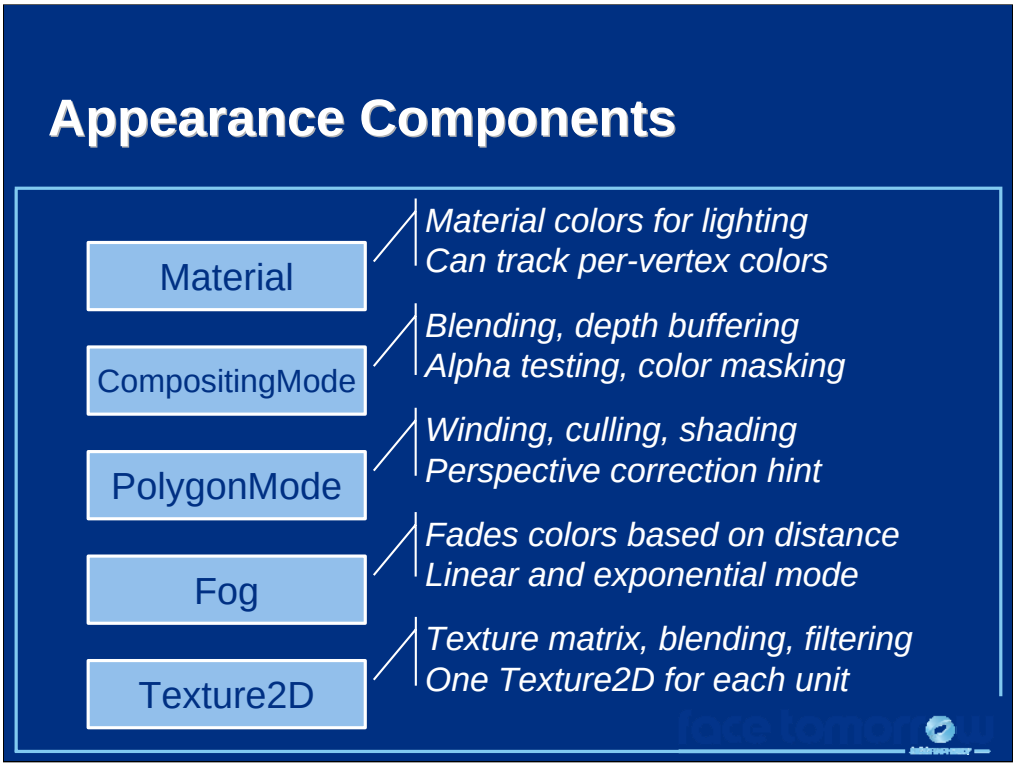
* OpenGL ES only supports RGBA colors

Floating point vertex arrays were excluded for performance and code size reasons. To compensate, there are floating point scale and bias terms for vertex and texcoord arrays. They cause no overhead, since they can be implemented with the modelview or texture matrix.

Homogeneous 4D coordinates were dropped to get rid of nasty special cases in the scene graph, and to speed up skinning, morphing, lighting and vertex transformations in general.

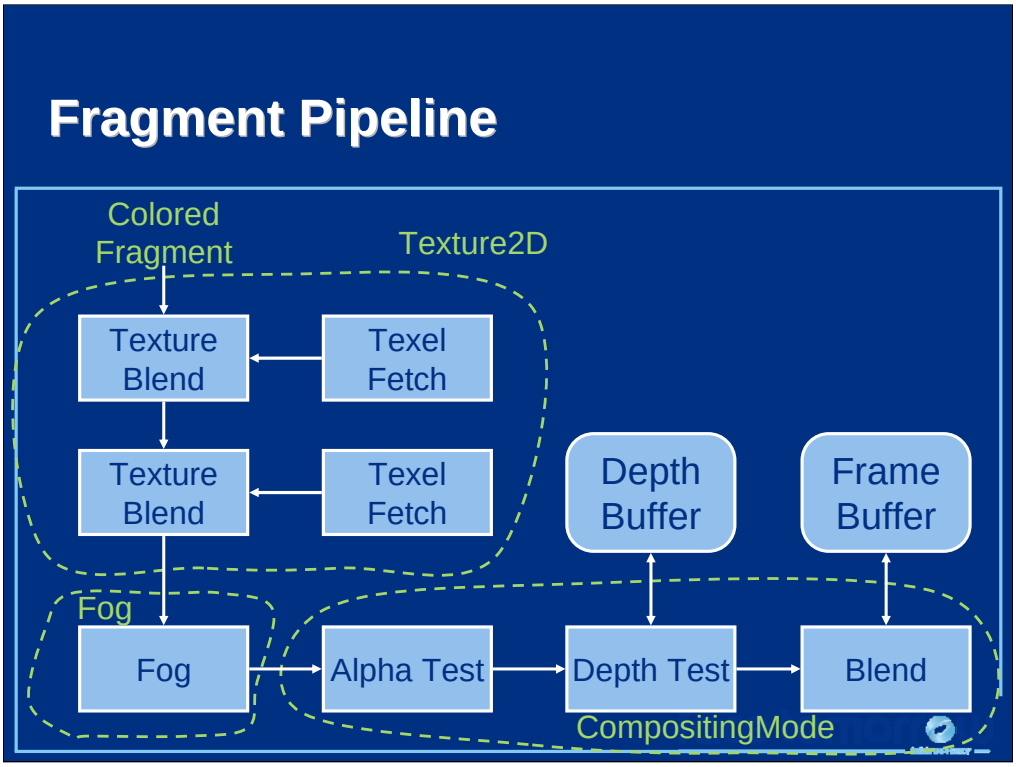
Vertex and Index Buffer Objects

- Vertices and indices are stored on server side
  - Similar to OpenGL Buffer Objects
  - Reduces data traffic from Java to native
  - Allows caching, bounding box computation, etc.



Functionally related blocks of rendering state are grouped together. Appearances as well as individual Appearance components can be shared by arbitrary number of meshes.

This saves memory space, reduces garbage collection, and allows implementations to quickly sort objects based on their rendering state.



Here is a high-level view of the M3G/OpenGL fragment pipeline, and how some of the Appearance components map onto that. The other components would map to the transformation & lighting pipeline in a similar way.

M3G API Overview

Getting started

Rendering

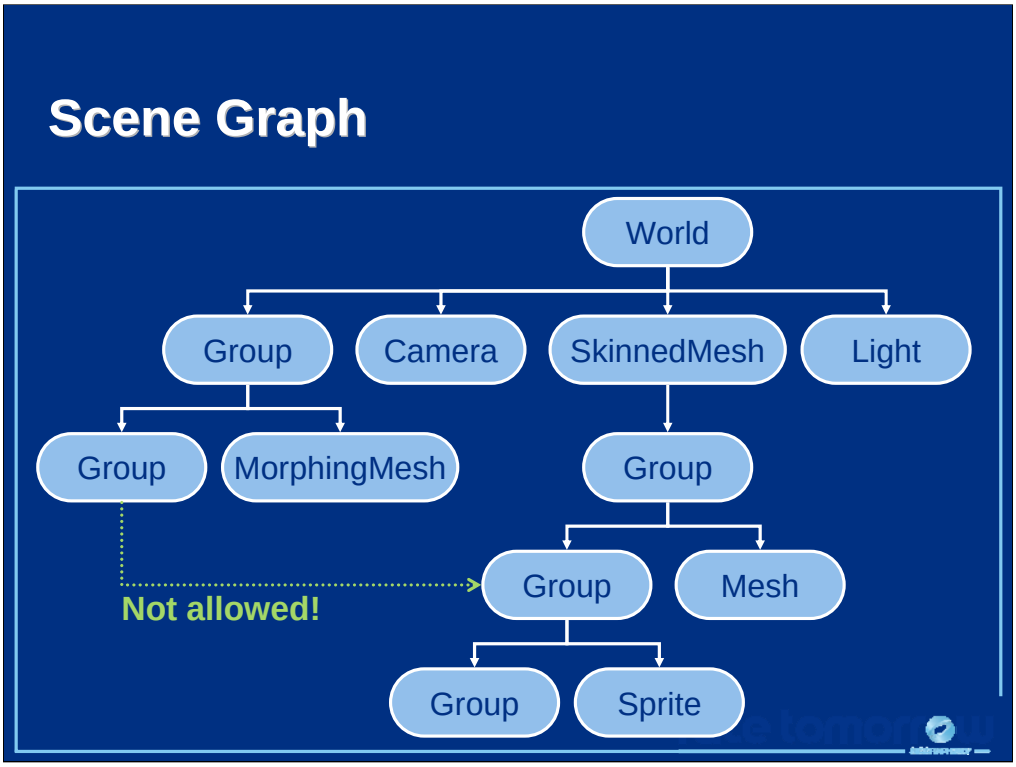
Scene graph

Performance tips

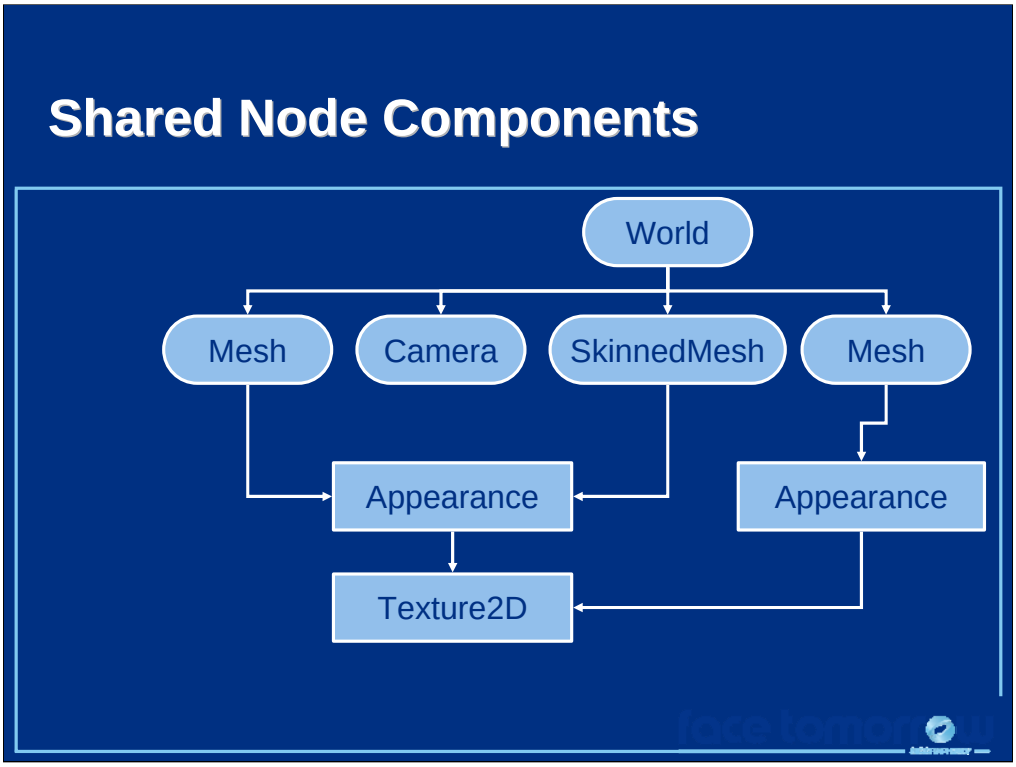
Deformable meshes

Keyframe animation

Demos



Scene graph nodes can’t have more than one parent, so the scene graph is actually just a tree.

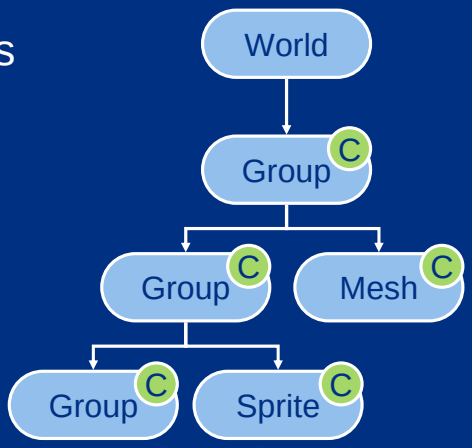


Even though nodes can't be instanced, their component objects can. Textures, vertices, and all other substantial data is in the components, and only referenced by the nodes.

In this example, the Mesh and MorphingMesh share a common Appearance, which in turn shares a Texture2D with another Appearance.

Node Transformations

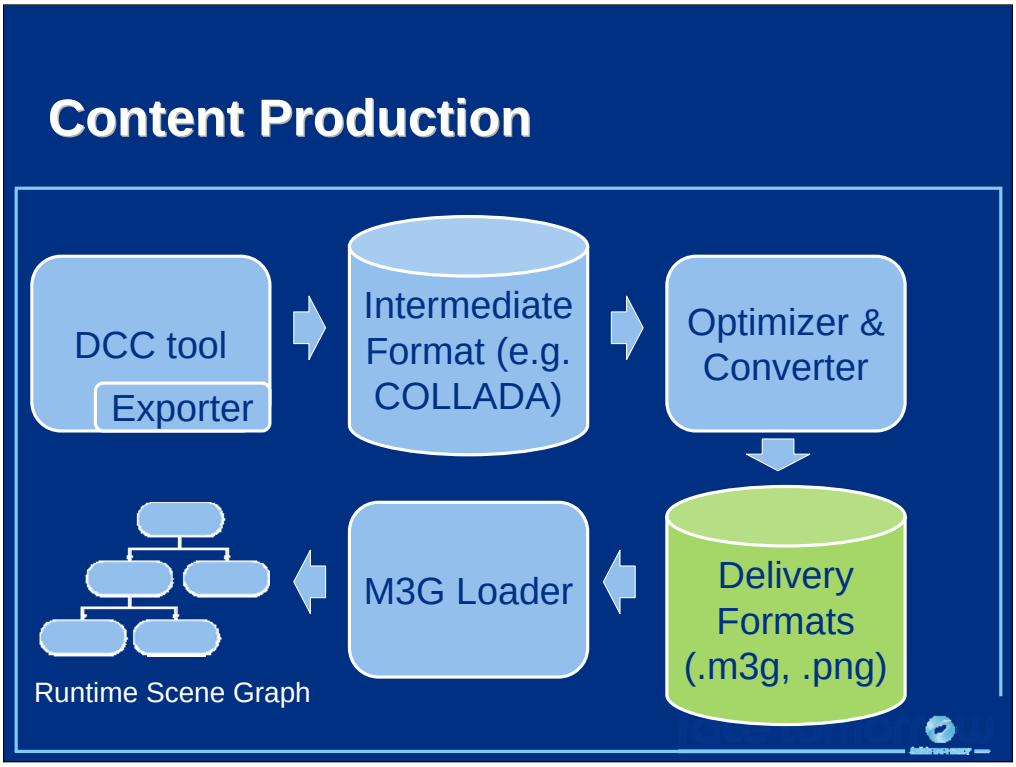
- From this node to the parent node
- Composed of four parts
  - Translation T
  - Orientation R
  - Non-uniform scale S
  - Generic 3x4 matrix M
- **C = T R S M**



Other Node Features

- Automatic alignment
  - Aligns the node's Z and/or Y axes towards a target
  - Recomputes the orientation component (R)
- Inherited properties
  - Alpha factor (for fading in/out)
  - Rendering enable (on/off)
  - Picking enable (on/off)
- Scope mask

Content Production



M3G File Format

- Small size, easy to decode
- Matches 1:1 with API features
- Stores individual objects, entire scenes
- ZLIB compression of selected sections
- Can reference external files – e.g. textures
- Highly portable – no extensions

M3G API Overview

Getting started

Rendering

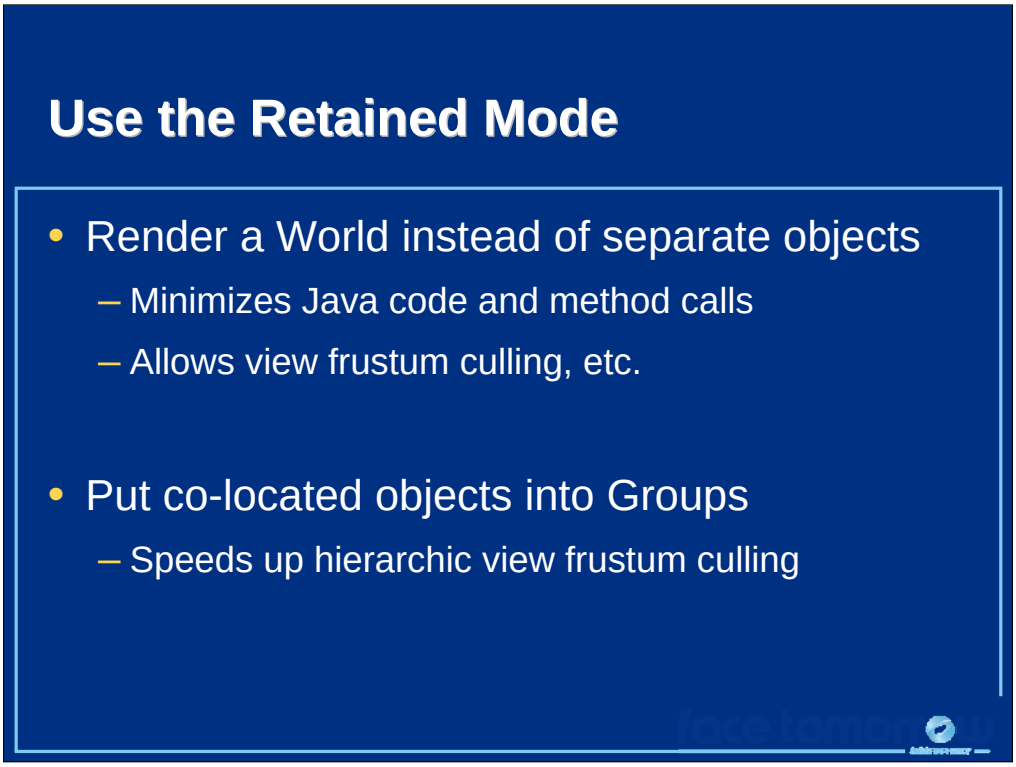
Scene graph

Performance tips

Deformable meshes

Keyframe animation

Demos



M3G engines generally perform shader state sorting and view frustum culling in retained mode.

However, any culling done by the engine is very conservative. The engine does not know which polygon mesh is a wall that's going to stay where it is, for instance. If you have a scene that could be efficiently represented as a BSP tree, you can't expect the engine to figure that out. You need to construct the tree yourself, and keep it in the application side.

Simplify Node Properties

- Transformations
  - Favor the **T R S** components over **M**
  - Avoid non-uniform scales in **S**
  - Use auto-alignment sparingly
- Keep the alpha factor at 1.0

Optimize Rendering Order

- `Appearance.setLayer(int layer)`
  - Defines a global ordering for submeshes
  - Within each layer, opaque objects come first
- Use layers for…
  - Making sure that overlays are drawn first
  - Making sure that distant objects are drawn last
  - Multipass effects (e.g. for lighting)

Objects should be generally rendered in a front-to-back order to minimize overdraw. Overlays should therefore be drawn first, and distant objects such as sky boxes last. The layer index, defined in Appearance, allows you to enforce the right ordering when you're using the retained mode.

Multipass rendering means that the same submesh is drawn several times with different Appearance settings. We might apply a base texture with diffuse lighting in the first pass, followed by an ambient occlusion map in the second pass, and a dynamic specular highlight in the third pass.

Optimize Texturing

- Multitexturing is faster than multipass
  - Transformation and setup costs cut by half
- Use mipmaps to save memory bandwidth
  - Tradeoff: 33% extra memory consumption
- Combine small textures into a texture atlas

Use Perspective Correction

- Much faster than increasing triangle count
  - Nokia: 2% fixed overhead, 20% in the worst case
- Pitfall: Quality varies by implementation
  - Refer to quality scores at www.jbenchmark.com

Reduce Object Count

- Per-Mesh processing overhead is high
- Per-submesh overhead is also fairly high

- Merge
  - Meshes that are close to each other
  - submeshes that have a common Appearance

Avoid Dynamic Geometry

- `VertexArray.set(…)` can be slow
  - Java array contents must be copied in
  - May also trigger bounding box updates, etc.
  - Replace with morphing or skinning where possible
- IndexBuffers have no `set(…)` method at all
  - `new IndexBuffer(…)` per frame is not a good idea
  - Switch between predefined IndexBuffers instead

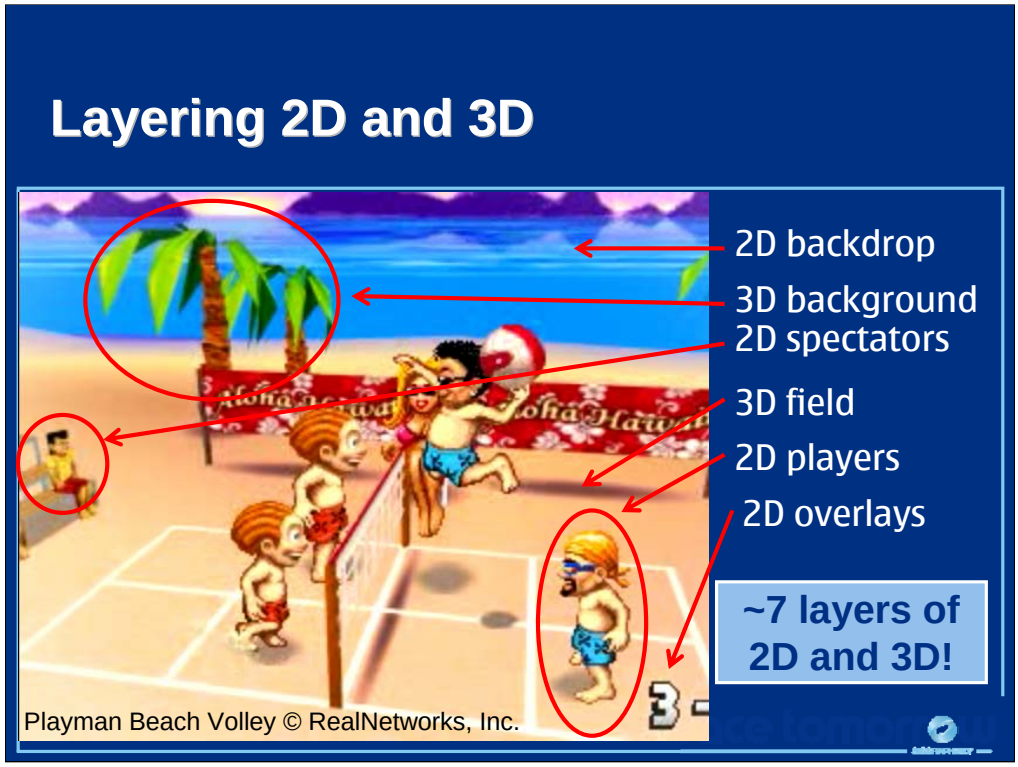
Beware of Exporters

- Exported content is not always optimal
  - Lighting enabled, but overwritten by texture
  - Lighting disabled, normal vectors still included
  - Alpha blending enabled, but alpha always 1.0
  - 16-bit vertices when 8 bits would be enough
  - Perspective correction always enabled
  - …
- Always review the exported scene tree!

Hardware vs. Software

- Shading state
  - SW: Minimize per-pixel operations
  - HW: Minimize shading state changes
- Mixing 2D and 3D rendering
  - SW: No performance penalty
  - HW: Substantial penalty (up to 3x)

Most OpenGL ES performance tips given by Ville in the earlier presentation apply also for M3G applications.



The current practice is to mix 2D and 3D quite liberally. Playman Beach Volley by Mr. Goodliving, Ltd., is a good example: there are something like 7 layers of 2D and 3D stacked on top of each other. This gives a pretty neat, cartoonish look, and does not require a high-end handset.

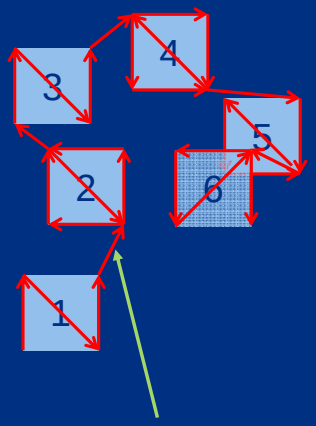
Unfortunately, high-end hardware-accelerated devices generally do not perform well with this kind of content, because the 2D routines are running on the CPU, and synchronizing the results of 2D and 3D rendering may require extra frame buffer copying. You'll be better off doing pure 3D on HW accelerated devices.

Use Picking with Caution

- `myWorld.pick(…)` can be very slow
- Restrict the pick ray to
  - meshes in a specific Group
  - meshes with a specific scope mask
- Use simplified geometry for picking
  - `setPickingEnable(true)`
  - `setRenderingEnable(false)`

Particle Effects

- Point sprites – not available
- Sprite3D – much too slow



- Put all particles in one Mesh
  - One particle == two triangles
  - Animate by `VertexArray.set(…)`

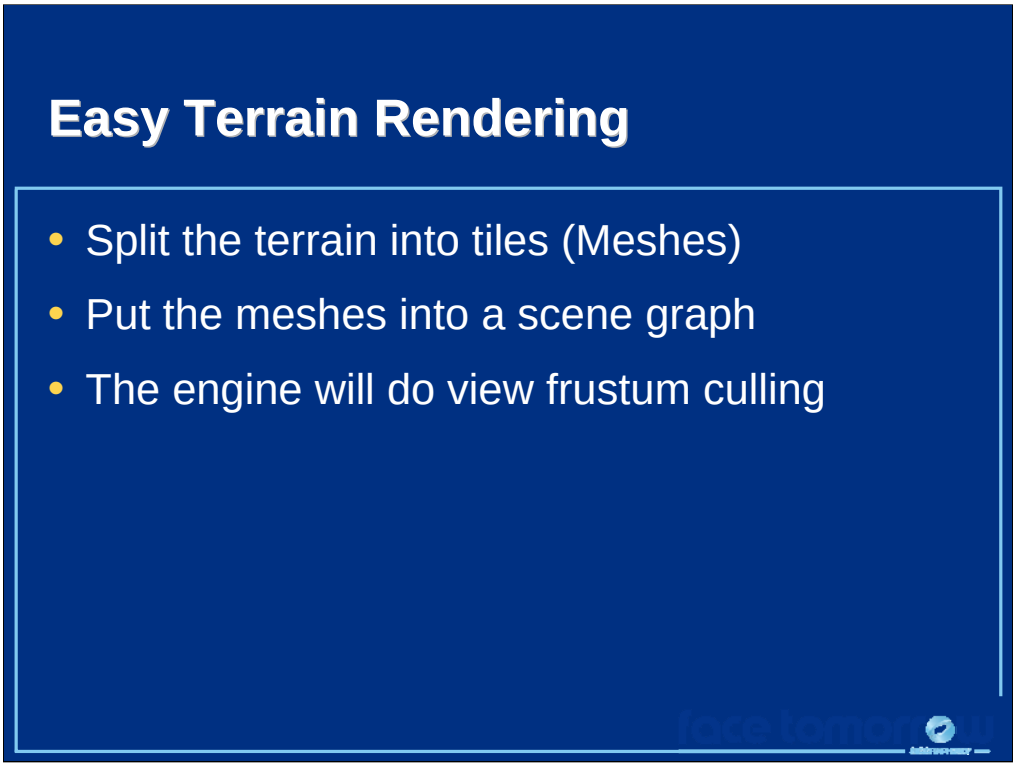
Particles glued into a tri-strip

So how should you implement a particle system, given that points and point sprites are not supported?

The first idea that comes to mind is to use Sprite3D. However, that would make every particle an independent object, each with its own modelview matrix, texture, and other rendering state. This implies a separate OpenGL draw call and lots of overhead for each particle.

It is more efficient to represent particles as textured quads, all glued into one big triangle strip that can be drawn in a single call. To make the particles face the viewer, set up automatic node alignment for the Mesh that encloses the particle system.

At run time, just update the particles' x, y, z coordinates and colors in their respective VertexArrays.

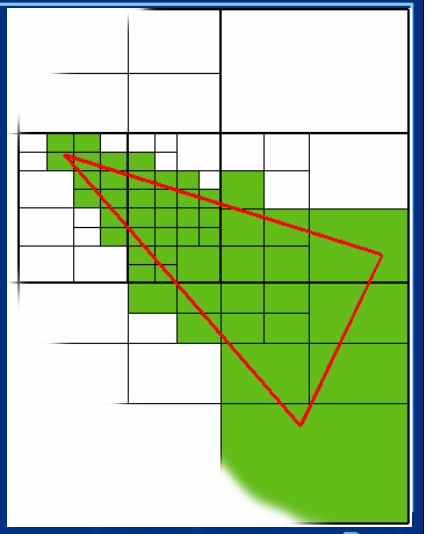


When splitting the terrain, keep in mind that the per-mesh overhead can be surprisingly high – especially on hardware accelerated platforms where the actual rasterization is fast. The optimal tile size varies by device, but any less than 100 polygons per mesh will most likely be counterproductive.

Since the modelview matrix of each tile will be unique, small rounding errors in the vertex pipeline may cause cracks between tiles. A simple solution is to make the tiles overlap each other a bit.

Terrain Rendering with LOD

- Preprocess into a quadtree
  - leaf node == Mesh
  - inner node == Group
- Use `setRenderingEnable` based on the view frustum



M3G API Overview

Getting started

Rendering

Scene graph

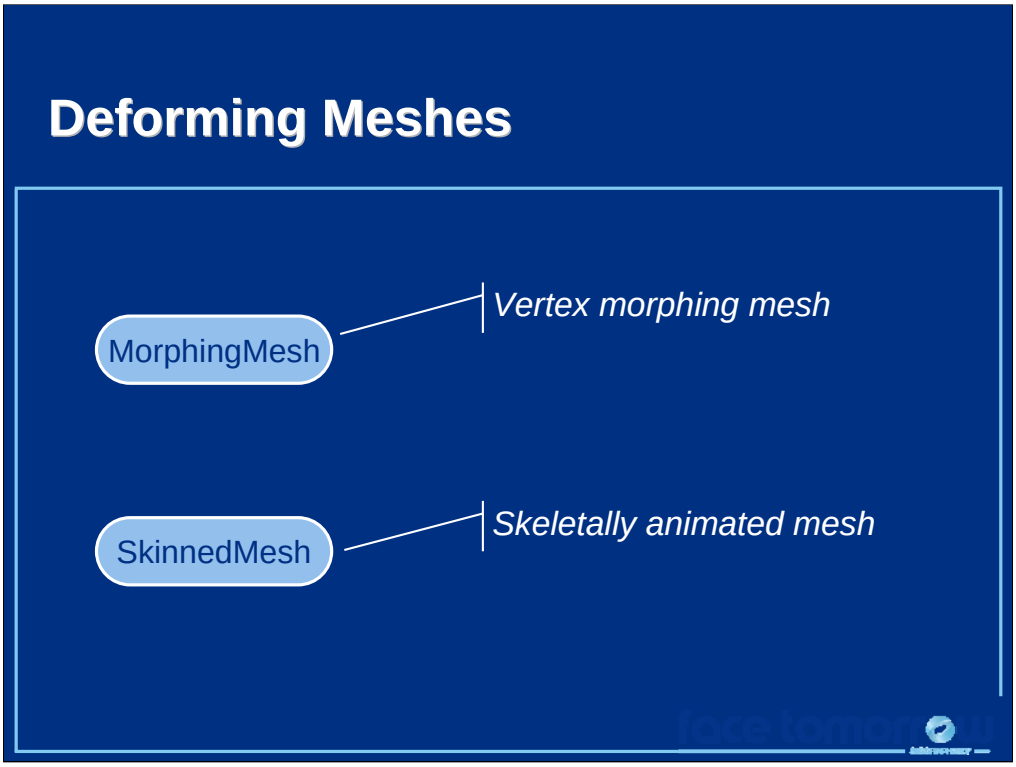
Performance tips

Deformable meshes

Keyframe animation

Demos

Deforming Meshes



MorphingMesh

- Traditional vertex morphing animation
  - Can morph any vertex attribute(s)
  - A base mesh **B** and any number of morph targets $\mathbf{T}_i$
  - Result = weighted sum of morph deltas

$$\mathbf{R} = \mathbf{B} + \sum_i w_i (\mathbf{T}_i - \mathbf{B})$$

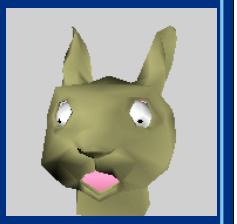
- Change the weights w_i to animate

MorphingMesh

Base

Target 1
eyes closed

Target 2
mouth closed



Animate eyes
and mouth
independently

SkinnedMesh

- Articulated characters without cracks at joints
- Stretch a mesh over a hierarchic “skeleton”
  - The skeleton consists of scene graph nodes
  - Each node (“bone”) defines a transformation
  - Each vertex is linked to one or more bones

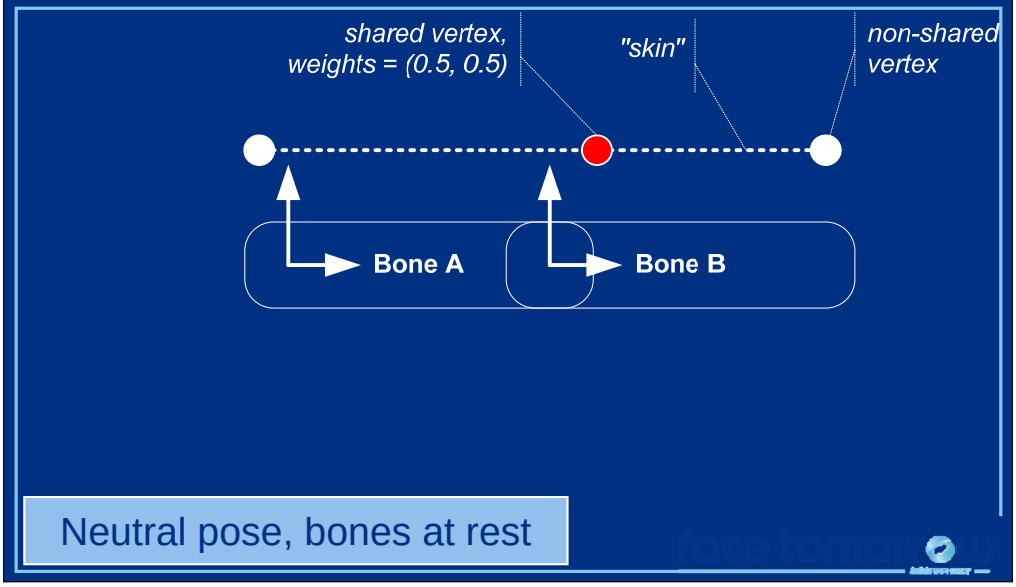
$$v' = \sum_i w_i \mathbf{M}_i \mathbf{B}_i v$$

  - $\mathbf{M}_i$ are the node transforms – v, w, $\mathbf{B}$ are constant

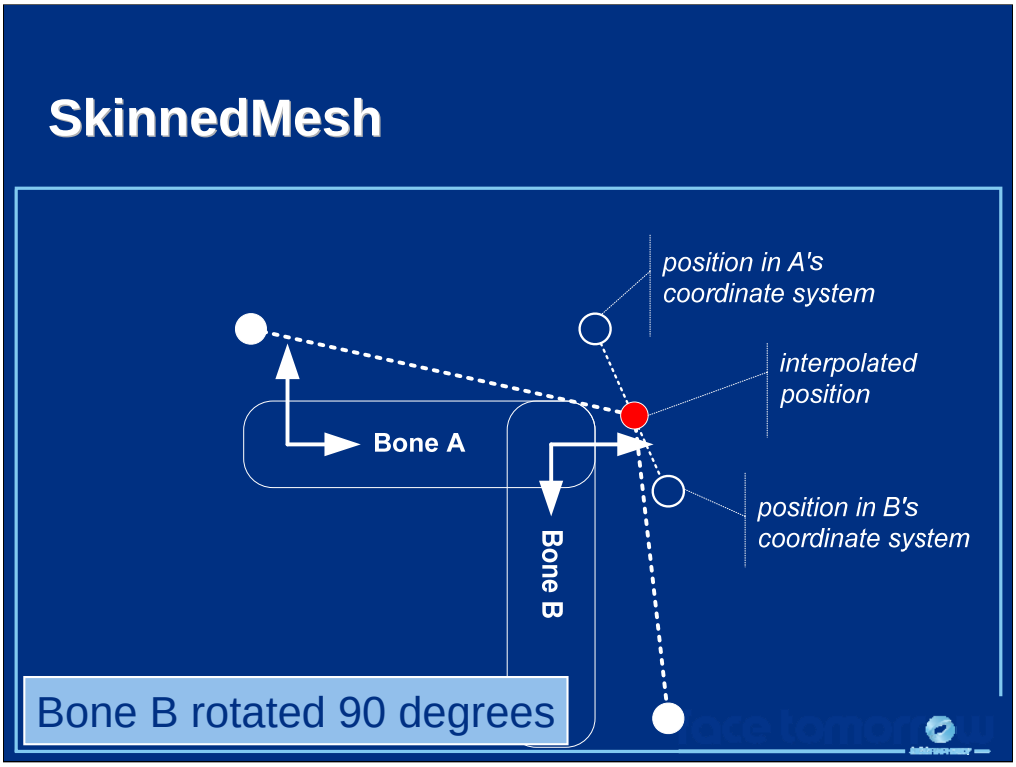
In the equation,

- **v** is the vertex position in the SkinnedMesh node’s coordinates
- $\mathbf{B}_i$ is the fixed at-rest transformation from SkinnedMesh to bone $\mathbf{N}_i$
- $\mathbf{M}_i$ is the dynamic transformation from bone $\mathbf{N}_i$ to SkinnedMesh
- $\mathbf{w}_i$ is the weight of bone $\mathbf{N}_i$ (the weights are normalized)
- $\mathbf{0} \le \boldsymbol{i} \le \mathbf{N}$, where **N** is the number of bones associated with **v**
- **v’** is the final position in the SkinnedMesh coordinate system

SkinnedMesh



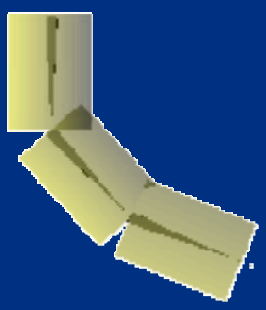
Neutral pose, bones at rest



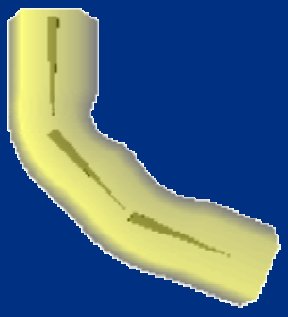
The empty dots show where the vertex would end up if it were associated with just one of the bones, respectively.

As the vertex is weighted equally by bones A and B, the final interpolated vertex lies in between the empty dots.

SkinnedMesh



Mesh

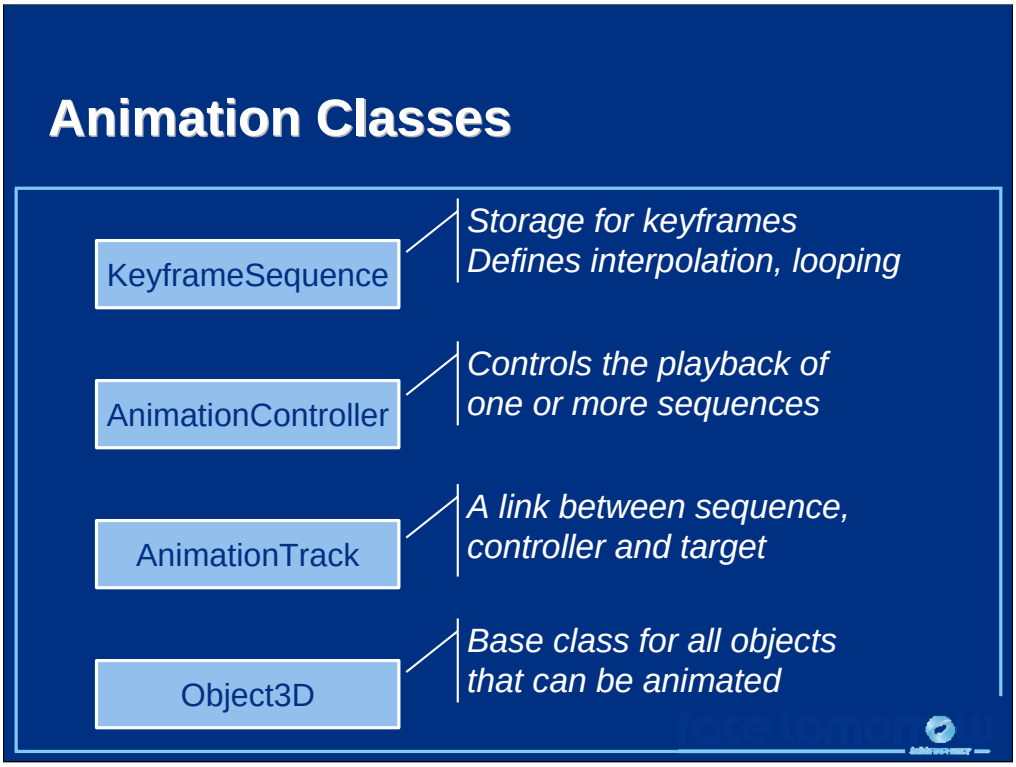


SkinnedMesh

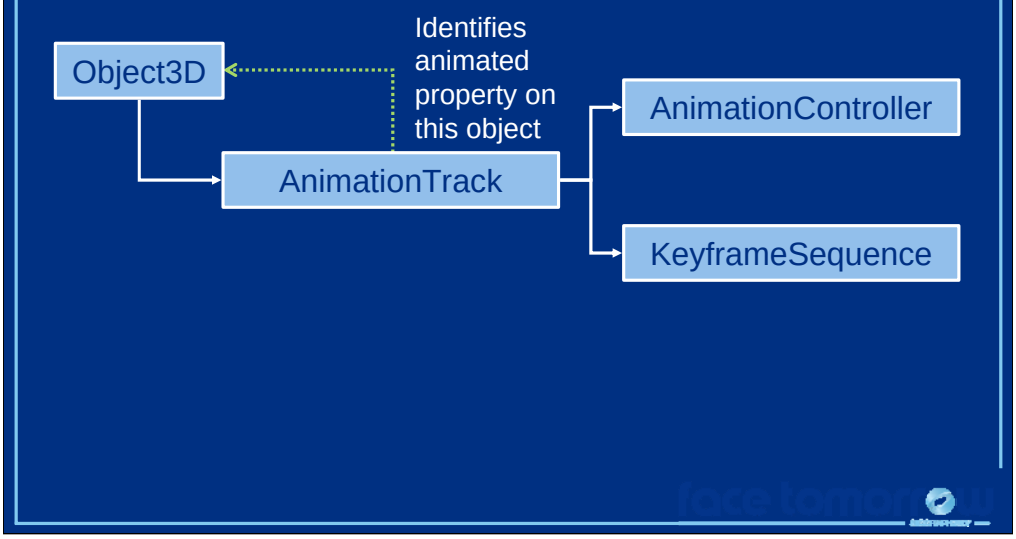
M3G API Overview

- Getting started
- Rendering
- Scene graph
- Performance tips
- Deformable meshes
- **Keyframe animation**
- Demos

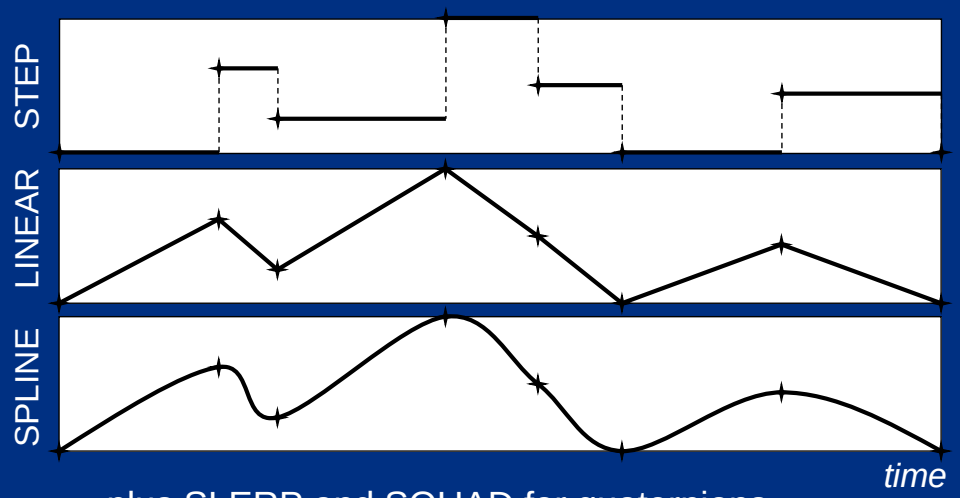
Animation Classes



Animation Classes



KeyframeSequence: Interpolation Modes



…plus SLERP and SQUAD for quaternions

AnimationController: Timing and Speed

- Maps world time into sequence time
- Can control any number of sequences

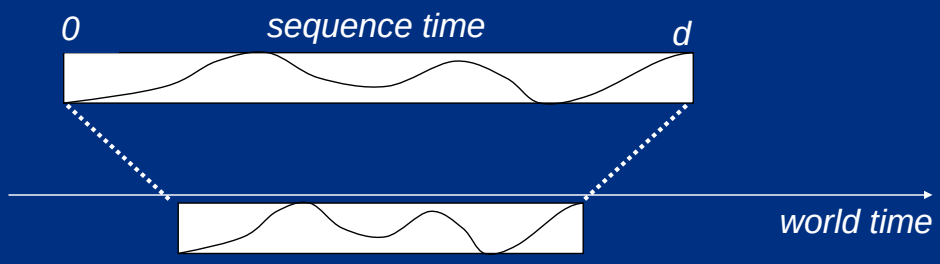
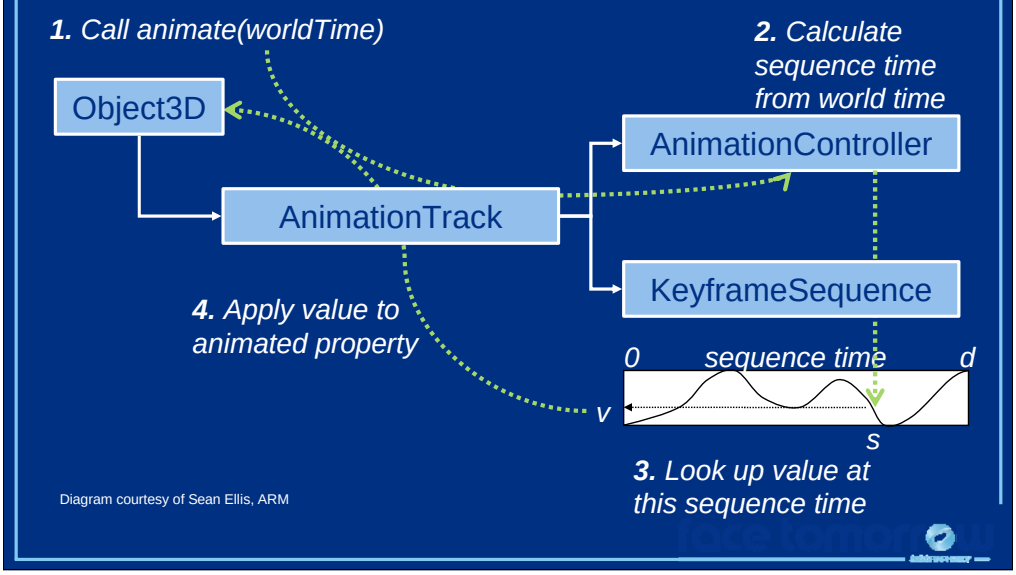


Diagram courtesy of Sean Ellis, ARM

Animation


1. Call animate(worldTime)
2. Calculate sequence time from world time
Object3D
AnimationController
AnimationTrack
KeyframeSequence
4. Apply value to animated property
0
sequence time
d
v
s
3. Look up value at this sequence time
Diagram courtesy of Sean Ellis, ARM

Animation

Tip: Interpolate quaternions as ordinary 4-vectors

- Supported in HI Corp's M3G Exporter
- SLERP and SQUAD are slower, but need less keyframes
- Quaternions are automatically normalized before use

M3G API Overview

- Getting started
- Rendering
- Scene graph
- Performance tips
- Deformable meshes
- Keyframe animation
- **Demos**

Summary

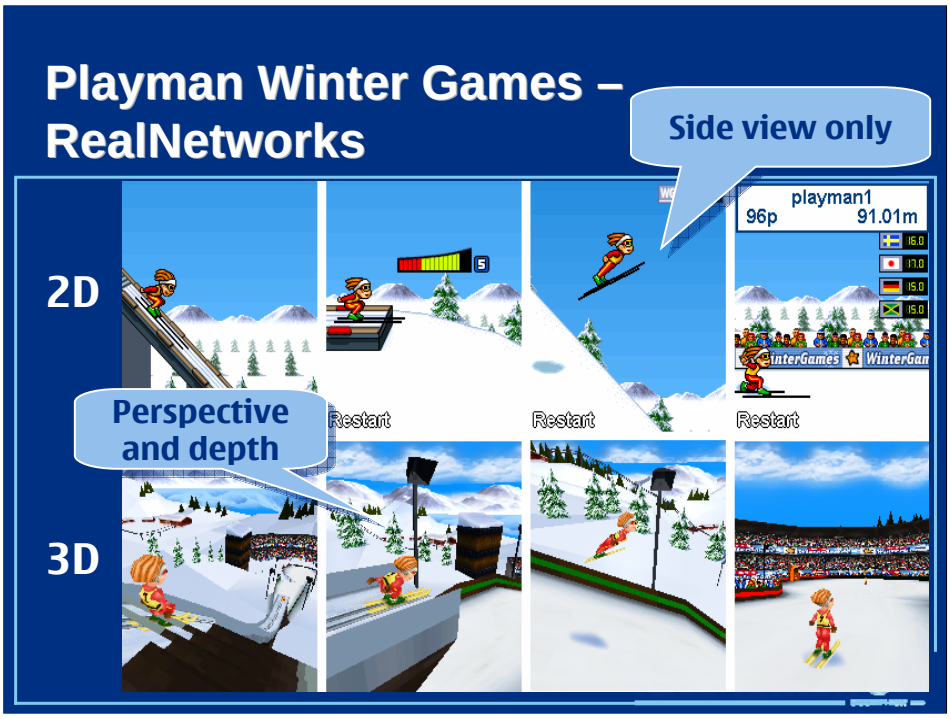
- M3G enables real-time 3D on mobile Java
  - Minimizes Java code along critical paths
  - Designed for both software and hardware
- OpenGL ES features at the foundation
- Animation & scene graph layered on top

30’000 devices sold during this presentation

Demos

SIGGRAPH2007

Playman Winter Games – RealNetworks



Playman World Soccer – RealNetworks

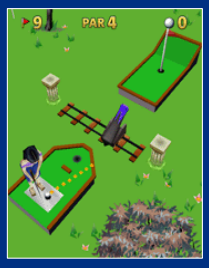
- 2D/3D hybrid
- Cartoon-like 2D figures in a 3D scene
- 2D particle effects etc.

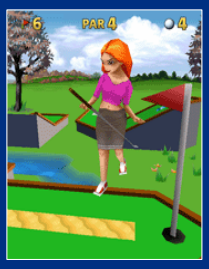
Tower Bloxx – Digital Chocolate

- Puzzle/arcade mixture
- Tower building mode is in 3D, with 2D overlays and backgrounds
- City building mode is in pure 2D

Mini Golf Castles – Digital Chocolate

- 3D with 2D background and overlays
- Skinned characters
- Realistic ball physics





Rollercoaster Rush – Digital Chocolate

- 2D backgrounds
- 3D main scene
- 2D overlays

Q&A

Thanks:

Sean Ellis (ARM)

Kimmo Roimela (Nokia)

Markus Pasula (RealNetworks)

Sami Arola (Digital Chocolate)


SIGGRAPH2007

M3G in the Real World

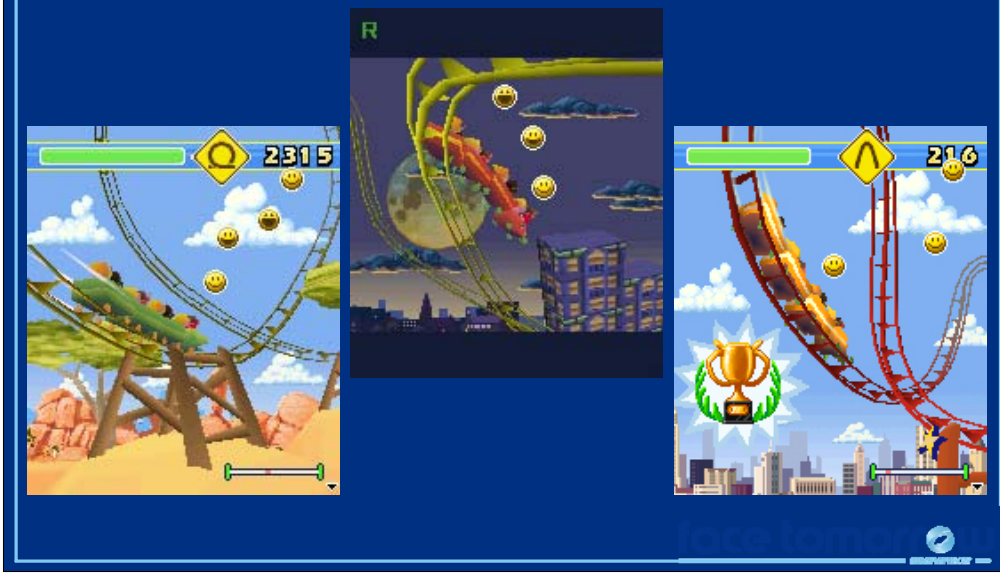
Mark Callow

Chief Architect

An M3G Game

DIGITAL CHOCOLATE

Rollercoaster Rush 3D™



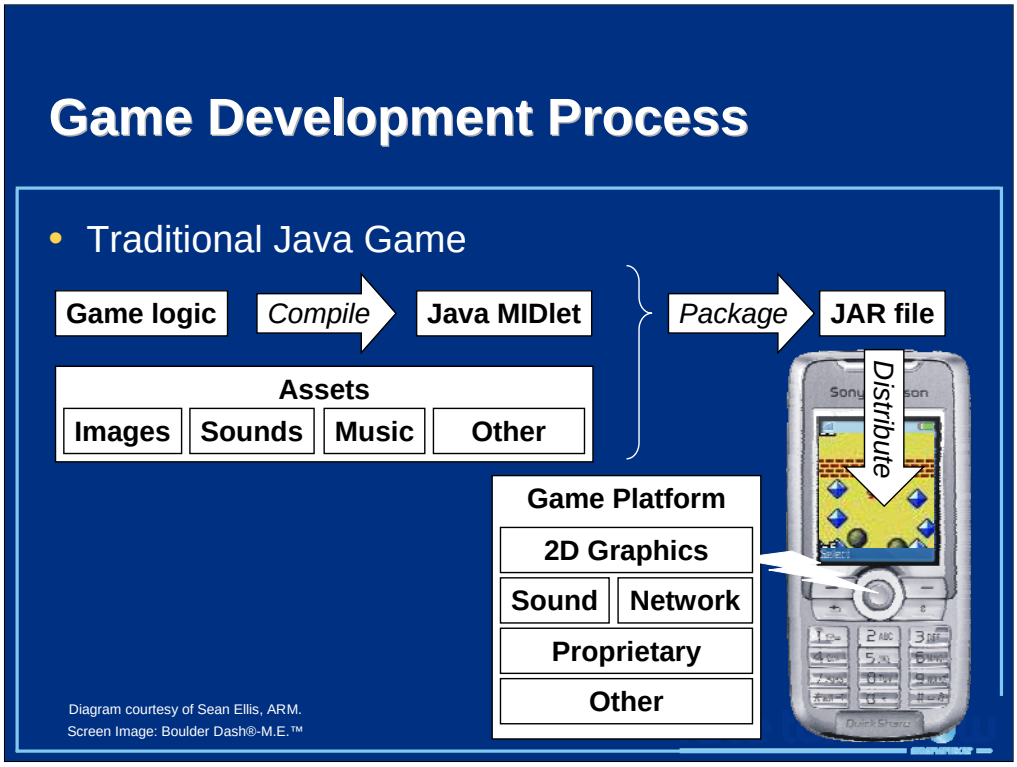
Agenda

- J2ME game development
- Tools
- COFFEE BREAK
- The structure of a MIDlet
- A walkthrough a sample game
- Why mobile game development is hard
- Publishing your content

A MIDlet is a J2ME applet.

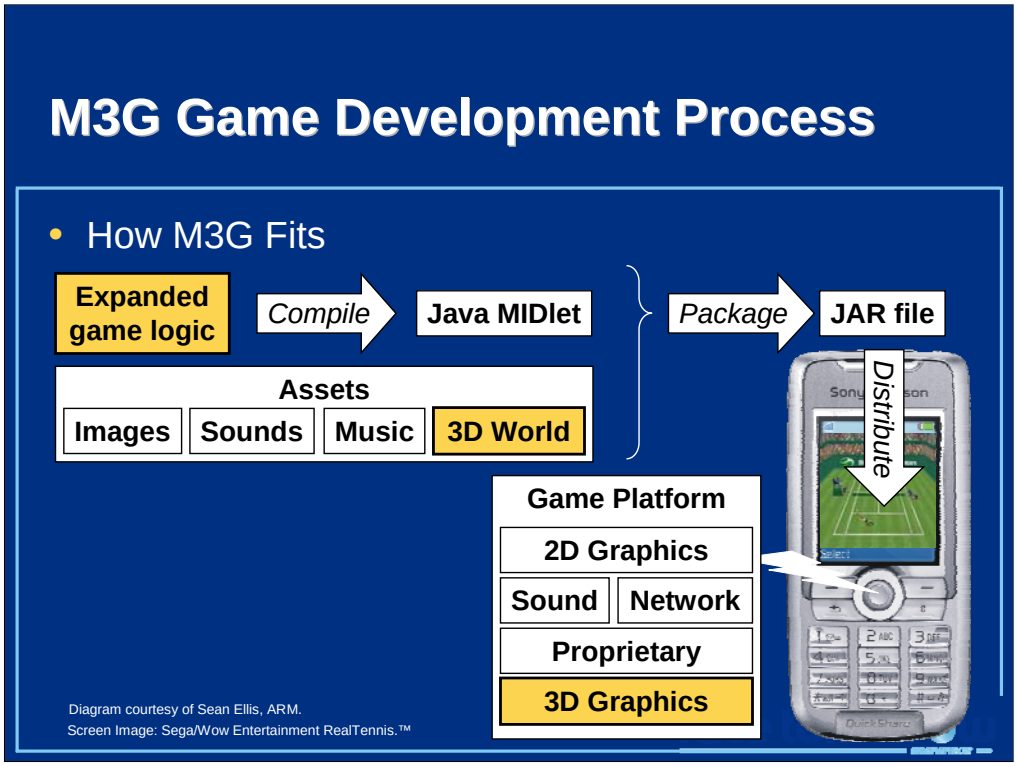
Agenda

- J2ME game development
- Tools
- COFFEE BREAK
- The structure of a MIDlet
- A walkthrough a sample game
- Why mobile game development is hard
- Publishing your content



We begin with a quick look at the various steps involved in creating a traditional Java game. We have a game platform such as MIDP 2 embedded in the mobile device. We need to write our game code targeted for this platform and compile it to a MIDlet. We package this into a JAR file together with the game's assets such as images, sounds and music. Finally we distribute the game package to the customers.

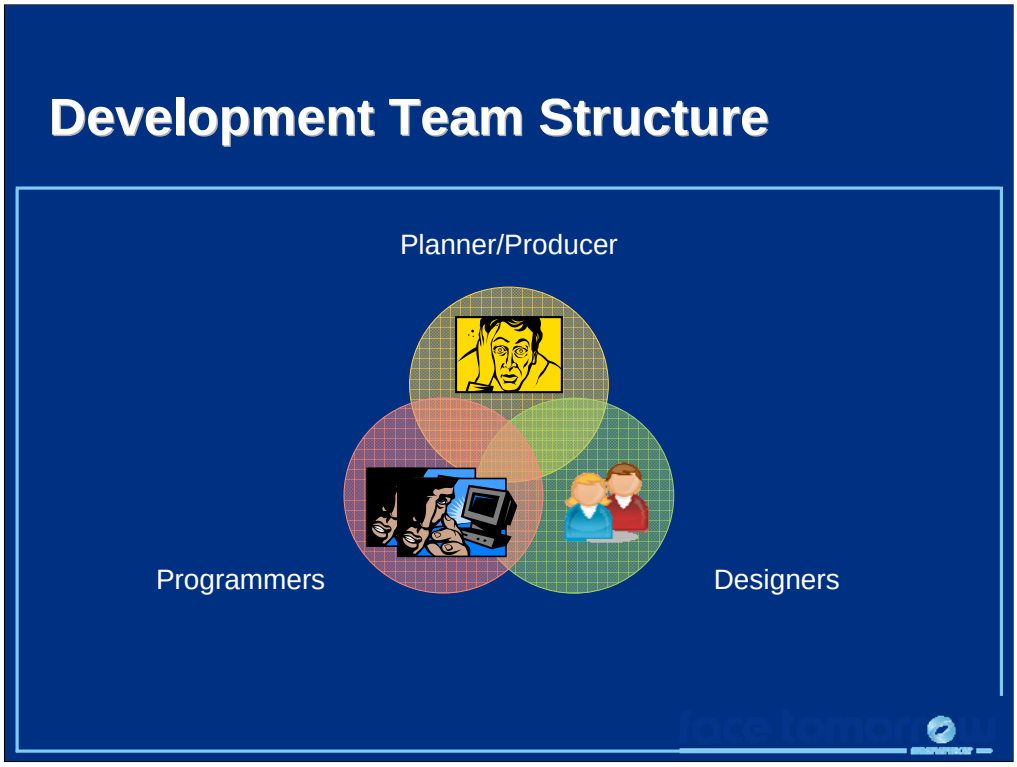
I'll be discussing each of these steps.



Now what does M3G bring to the party? First and foremost of course, 3D graphics is added to the game platform. This means your assets will include 3d models or a 3d scene.

You will also need to expand your game logic.

Effective use of 3D influences all aspects of a game's design and must be considered from the beginning of the design process.



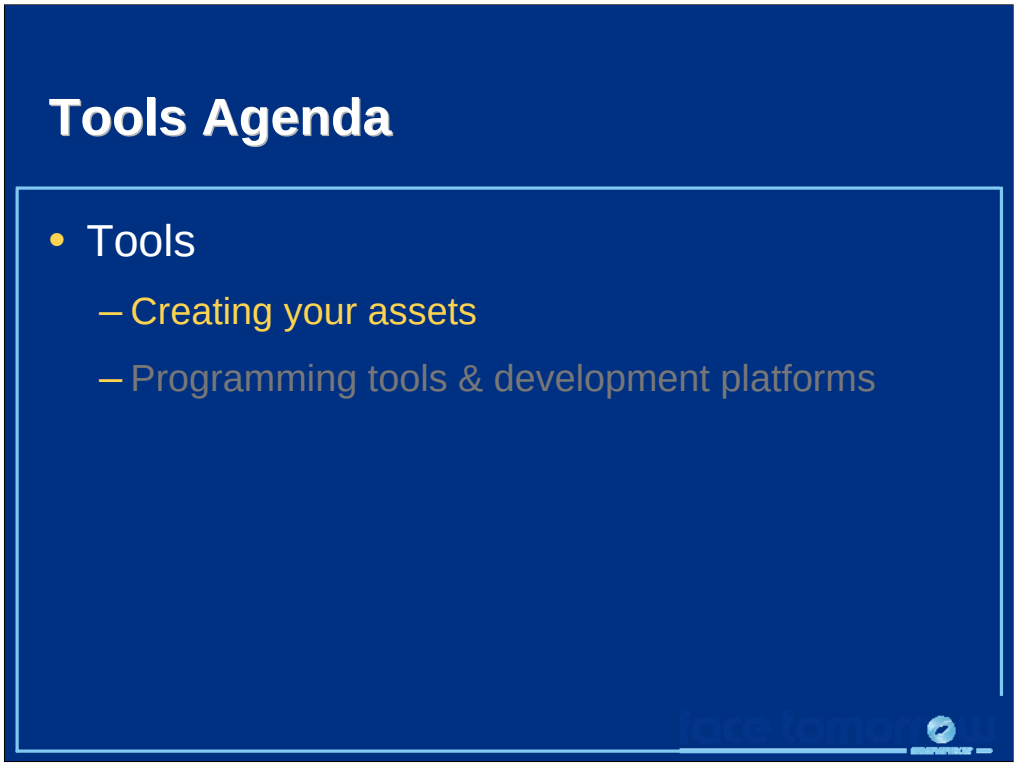
We have just seen that we need to create both artistic assets and program code. This means there is a need for both designers and programmers in the development team. Both teams will be guided by a Planner or Producer who plans the game and makes the overall decisions in consultation with the client.

It is important to enable the designers and programmers to work as independently as possible. This requires careful planning and keeping the programming side as general as possible. Artists must be able to make and artistic decisions and show them for approval without having to get a programmer to make code changes.

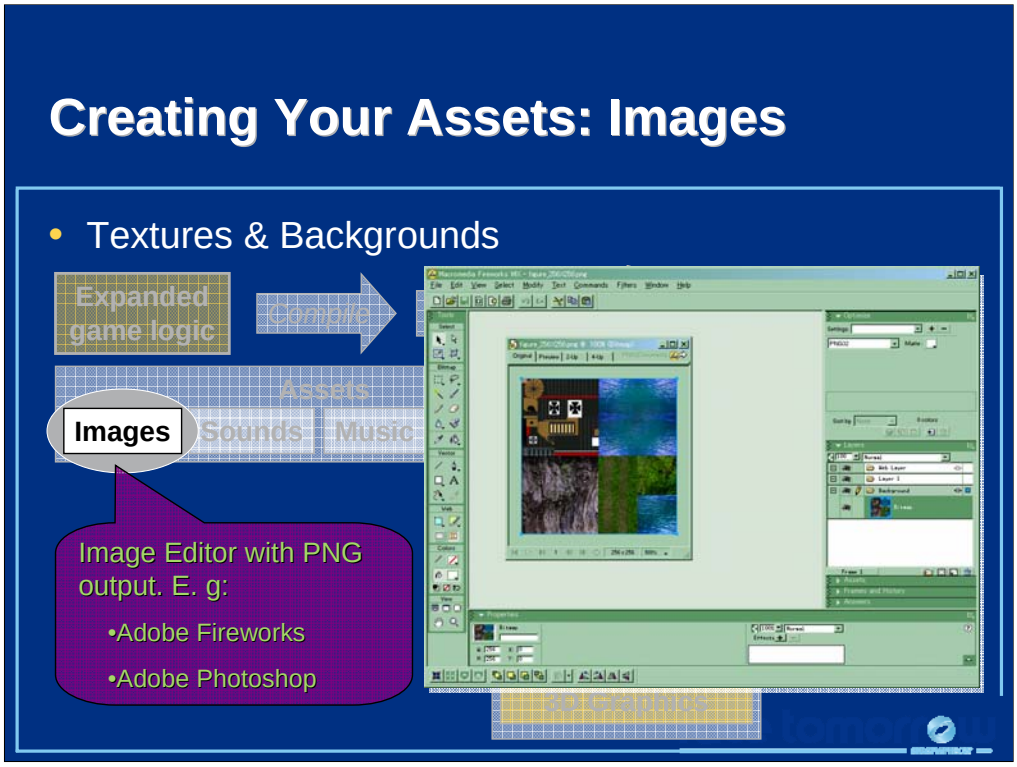
For example, whenever you need an opening door on a game level, the program should simply run a key-framed door opening animation which will be provided by the designer along with the style of door. Doing the animation programmatically would mean that the programmer would need to be involved when the designer decides to change from a sliding door to an exploding door.

Agenda

- J2ME game development
- Tools
- COFFEE BREAK
- The structure of a MIDlet
- A walkthrough a sample game
- Why mobile game development is hard
- Publishing your content



For any real m3g application, some art assets have to be created before the program can do anything useful. So we'll look first at creating the assets and then at the programming.



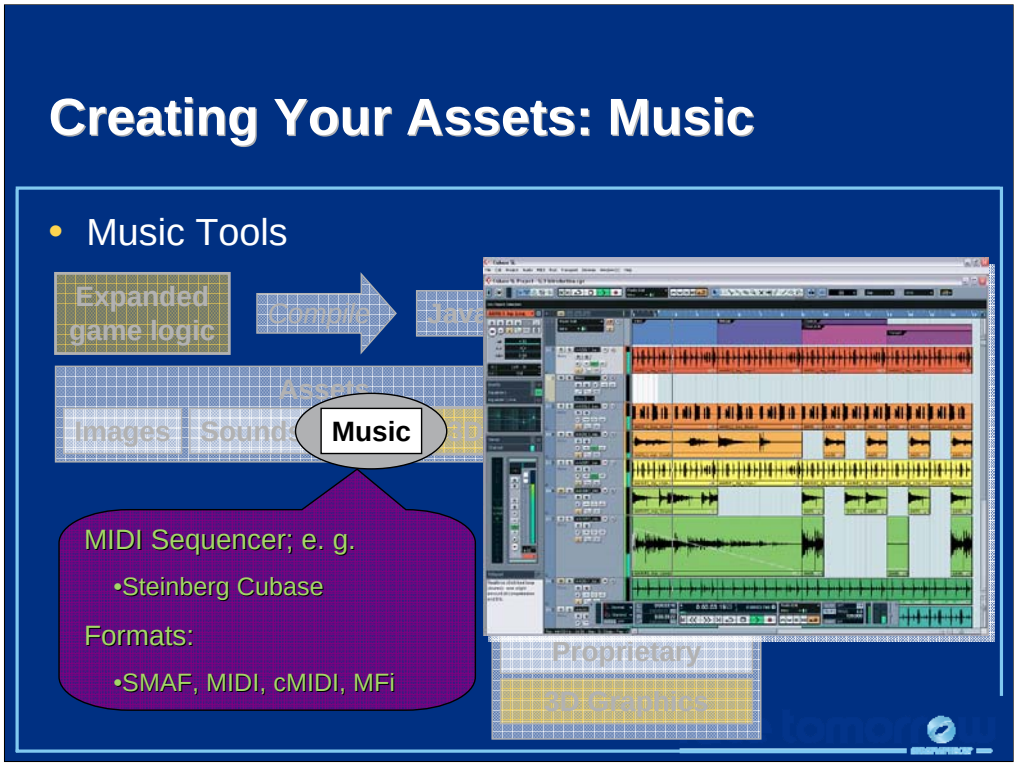
Textures and background images can be provided as PNG format files or the image data can be included directly in an M3G file. We recommend creating these assets in PNG format. PNG compresses better than plain zlib.

Some M3G plug-ins for 3d modeling tools automatically convert texture maps to PNG format. If so, you can use any texture map format supported by your modeling tool.

Do not use GIF files. Some M3G implementations appear to support GIF files as an accidental side-effect of the underlying MIDP implementation. Do not be fooled. The spec. does not require GIF support and many implementations do not support the format.

The J2ME (MIDP 2.0) specification does require support of any particular sound format. The formats listed here are commonly used.

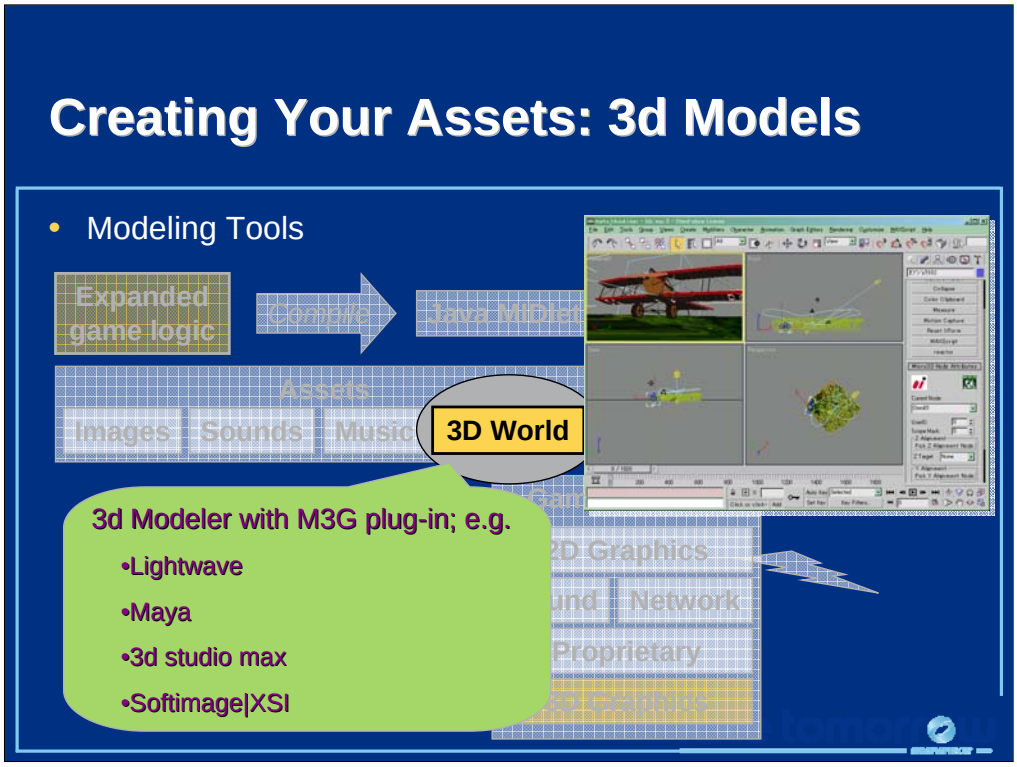
SMAF (Synthetic music Mobile Application Format) is a Yamaha invented format directly supported by chips used in many handheld portable devices. The file extension is .mmf. SMAF files can contain both recorded audio and synthesizer sequences.



cMIDI is compact MIDI which reduces the range of allowed MIDI data thereby reducing the file size.

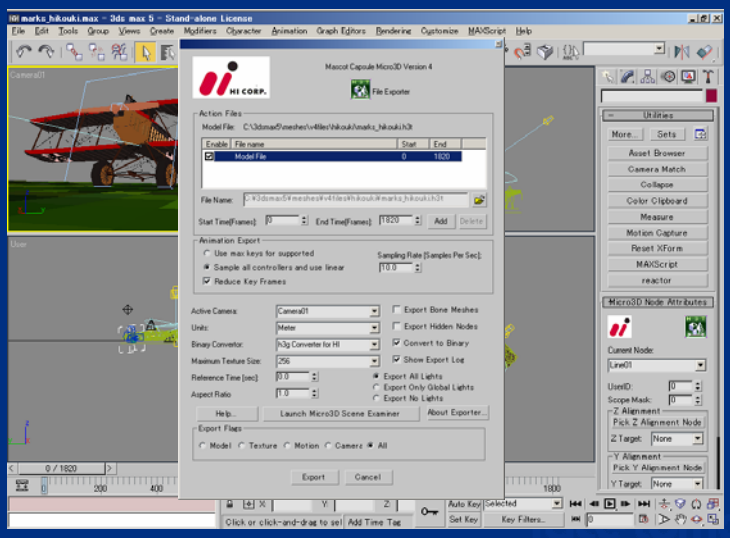
MFi (Melody Format for i-Mode) is supported on all i-Mode phones worldwide. As with SMAF, MFi can hold both MIDI-like data (cMIDI) and custom samples.

For all of your audio , you will mostly be dealing with hardware designed for ring tones. It is important that you understand the capabilities of the chip in your target phone.



A list of available M3G plug-ins is given at the end of this presentation.

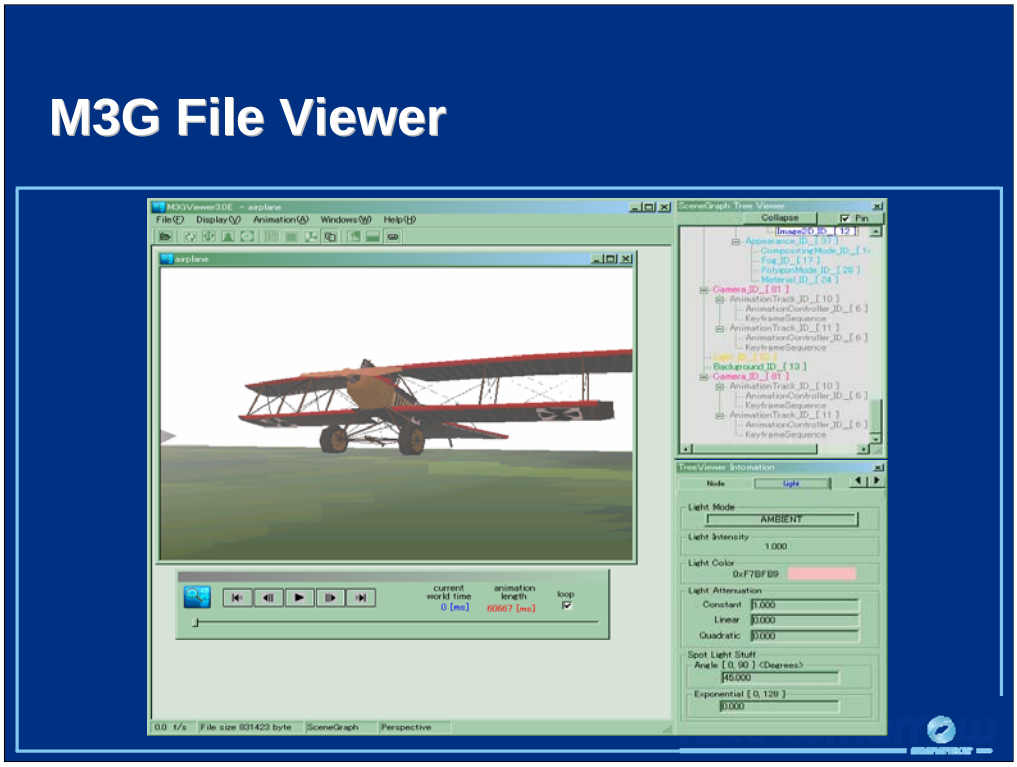
Export 3d Model to M3G



The plug-ins usually provide a way to set various M3G related parameters and save them with the DCC tool file. For example in the bottom right corner of this picture of HI's 3d Studio Max plug-in, you can see a utility for setting M3G node attributes.

When you are exporting the file a dialog appears that lets you specify various parameters for the export, such as what lighting to export, whether to sample controllers or use the controller keys specified in max and which camera should be the active camera.

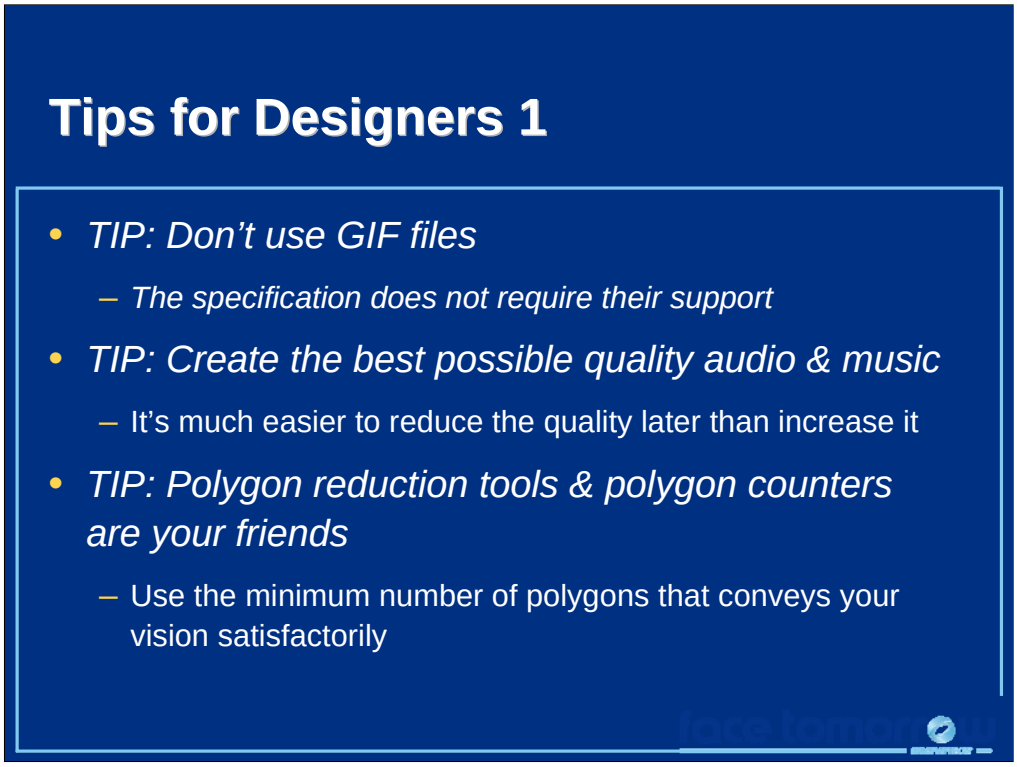
M3G File Viewer



This is HI's M3G file viewer. It uses the same M3G engine as is found in many mobile phones.

HI's plug-ins actually export an intermediate text format called H3T rather than M3G. A converter is provided for converting H3T to M3G. Have an intermediate file in editable text format can be very useful. The M3GViewer can display both H3T and M3G files.

Demo: On a Real Phone



Since we are looking at the tools for creating 3D model assets, this is a good time for some tips for designers.

Don't use GIF.

As mentioned earlier, when designing sound it is important to be aware of the capabilities of the target phone. Since these vary widely, it is best to create the original audio assets at the best possible quality.

Polygon reduction.

Tips for Designers 2

- *TIP: Use light maps for lighting effects*
  - Usually faster than per-vertex lighting
  - Use luminance textures, not RGB
  - Multitexturing is your friend
- *TIP: Try LINEAR interpolation for Quaternions*
  - *Faster than SLERP*
  - *But less smooth*

Tomi already mentioned use of linear interpolation for quaternions in his presentation.

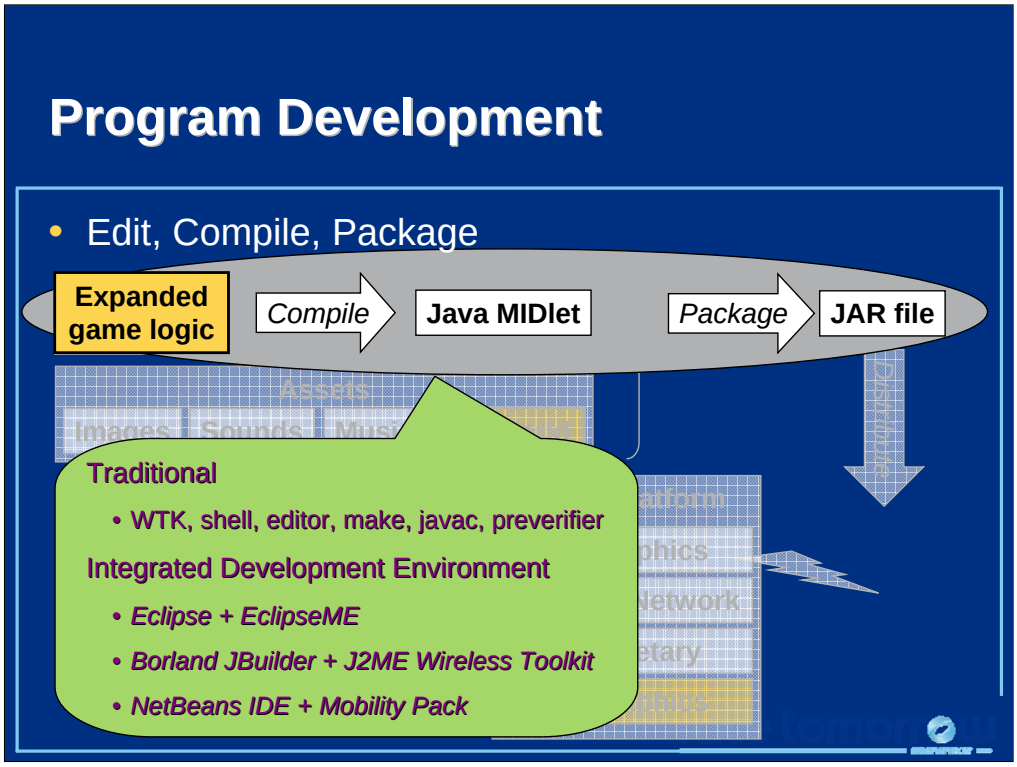
Tips for Designers 3

- *TIP: Favor textured quads over Background & Sprite3D*
  - Background and Sprite3D will be deprecated in M3G 2.0
  - Were intended to speed up software renderers
  - but implementation is complex, so not much speed up and no speed up at all with hardware renderers
  - Nevertheless Sprite3Ds are convenient to use for 2D overlays and Backgrounds are convenient when background scrolling is required.
- *LIMITATION: Sprites not useful for particle systems*

Sprites may not be faster than textured quads when a GPU is used for rendering.

Tools Agenda

- Tools
  - Creating your assets
  - Programming tools & development platforms



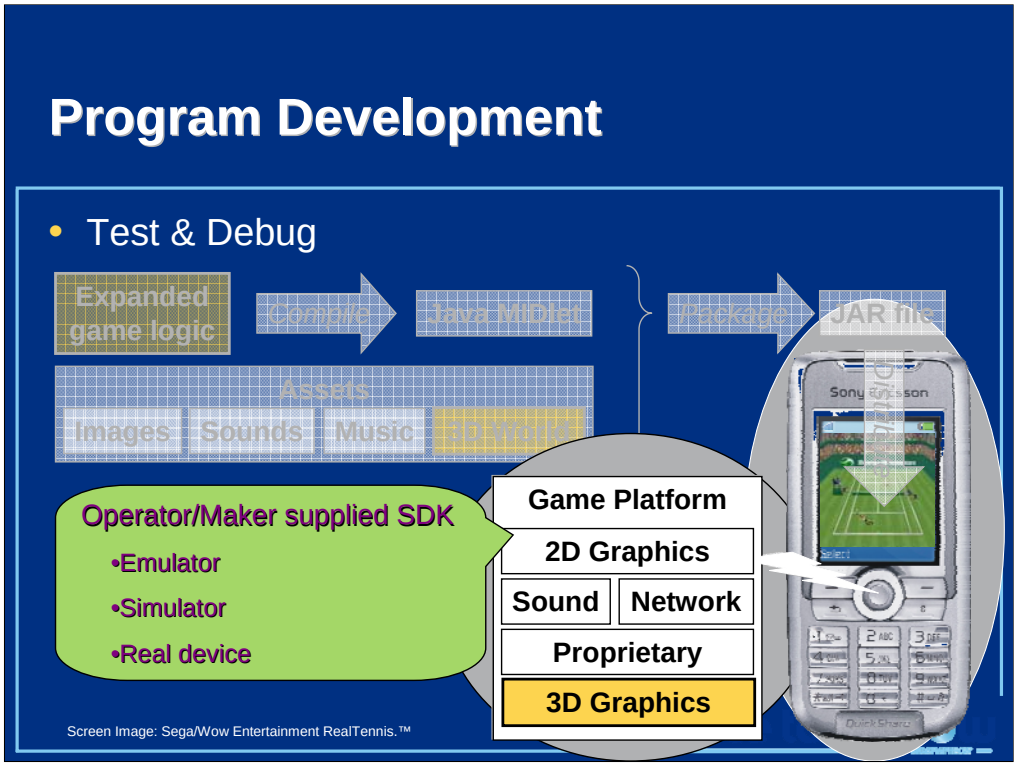
For the edit, compile build cycle you can use a traditional pipeline with a command line shell, programmer's editor, make and the standard java compiler from JDK 1.4.x or 1.5.x.

You can run the resulting MIDlet class in the emulator that comes with the Sun Java™ Wireless Toolkit for CLDC. (WTK). We see more of WTK in a moment.

Alternatively you can use a full IDE such as Borland's JBuilder, NetBeans (5.5 at time of writing) + NetBeans Mobility Pack or Eclipse with eclipseme.

NetBeans and Eclipse are free, Open-Source Integrated Development Environments for software developers. The IDE runs on many platforms including Windows, Linux, Solaris, and the MacOS.

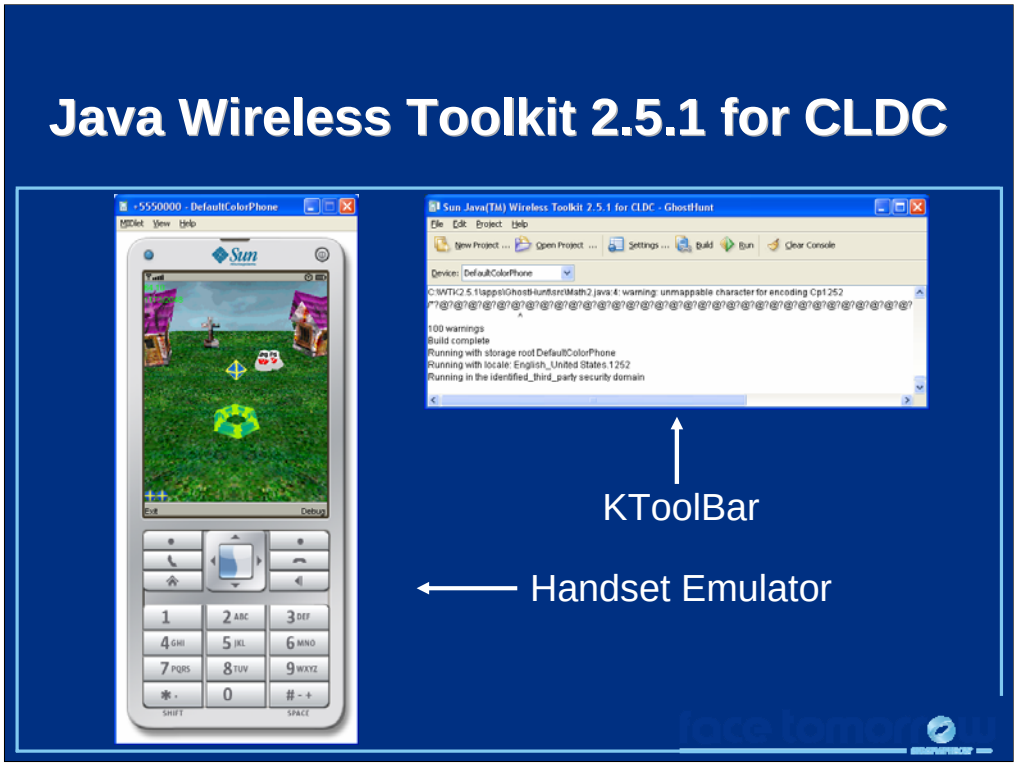
The NetBeans Mobility Pack for CLDC/MIDP adds everything to the IDE you need to create, test and debug applications for the Mobile Information Device Profile (MIDP) 2.0, and the Connected, Limited Device Configuration (CLDC) 1.1. You can easily integrate third-party emulators, including WTK, for a robust testing environment. In the same way, Eclipseme adds J2ME support to Eclipse.



For testing and debugging you need to use an SDK supplied by either the operator or the handset maker. These SDKs contain an “emulator”, usually a PC application that provides the functional environment of the real device. In at least one case, Sony Ericsson, the SDK includes a way to link to a real handset allowing applications to be tested and debugged on the real device. This is the ideal arrangement.

The Sun Java™ Wireless Toolkit for CLDC, mentioned earlier, includes a generic emulator for MIDP/CLDC. Several operators (e.g. Vodafone Global, Sprint, Softbank) and handset makers (e.g. Sony Ericsson) make their SDK’s by customizing WTK even though the JVM, MIDP & 3D renderer implementations in WTK are often not those used in the real phones.

This is a common problem with “emulators”. They can be quite different from the real devices and there is typically no relationship between performance in an “emulator” and performance on the real device.



This shows the Sun Java™ Wireless Toolkit 2.5.1 for CLDC mentioned in previous slides. It is a simple toolkit for building and running MIDP applications. You need to use an external editor for editing the source and there is no support for debugging except for skid marks (System.out.print). You can use KToolBar to build and run your MIDlet with the push of a button, saving you from having to write a make file and type long commands at the command line.

Demonstrate **compiling & running**!!!

Two problems must be noted with the WTK version 2.2 emulator. It will load GIF files as textures. This is permitted but not required by the M3G spec. As I noted earlier, you should avoid GIF files. Second it will fail to load M3G files with KeyframeSequence values encoded as shorts. They must be encoded as floats to keep WTK happy. As of this writing, I do not know if these problems have been fixed in more recent versions.

The download URLs for Wireless Toolkit 2.5.1 and many other WTK-based SDK's are given on the SDK slides at the end of the presentation. **WTK 2.5.1, released in June is the first version available for Linux as well as Windows.**

NetBeans + Mobility Pack + SE SDK

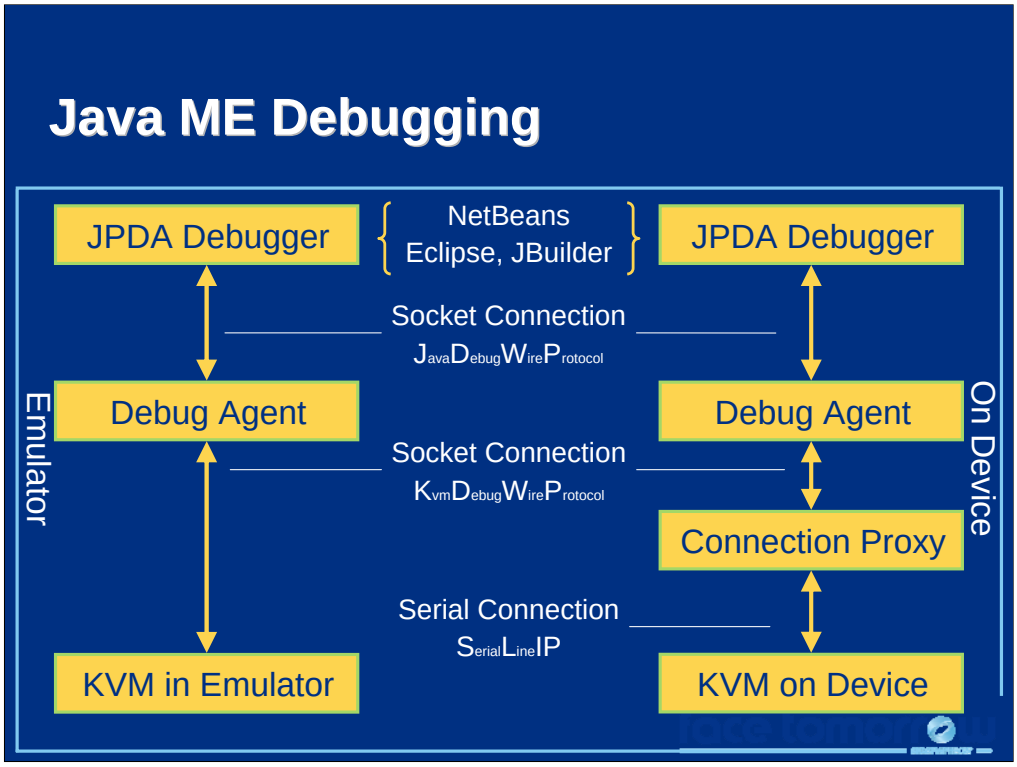
This shows NetBeans 5.5 with the Mobility Pack and Sony Ericsson's WTK-based SDK. NetBeans can be use for both Java and C/C++ applications. Within the IDE you can build, run and debug your Java code.

The emulator in Sony Ericsson's SDK, in common with many others, can be used to run and debug MIDlets that you are developing with NetBeans. You can also use NetBeans or Eclipse to debug MIDlets running on a real SE phone.

Demonstrate Platform Manager, changing code, building & debugging.

You will get lots of notices from Windows Firewall as the debuggers and proxies connect up.

Instructions for connecting phones for SE's "On Device Debug", are somewhat lacking. You must remove the Sony Ericsson PC Suite from your PC unless you can find some way to turn it off. When PC Suite is running it hides all the ports and therefore the device from the debug connection proxy.



To make KVM run with JPDA-compatible debugger IDEs without a huge memory overhead, a Debug Agent (also known as debug proxy) is interposed between the KVM and the JPDA-compatible debugger. The Debug Agent performs some of the debug commands on behalf of the JVM and allows many of the memory-consuming components of a JPDA-compliant debugging environment to be processed on the development workstation instead of the KVM. This reduces the memory overhead that the debugging interfaces have on the KVM and target devices. Communication between the Debug Agent and the KVM uses the KDWP, which is a strict subset of the JDWP.

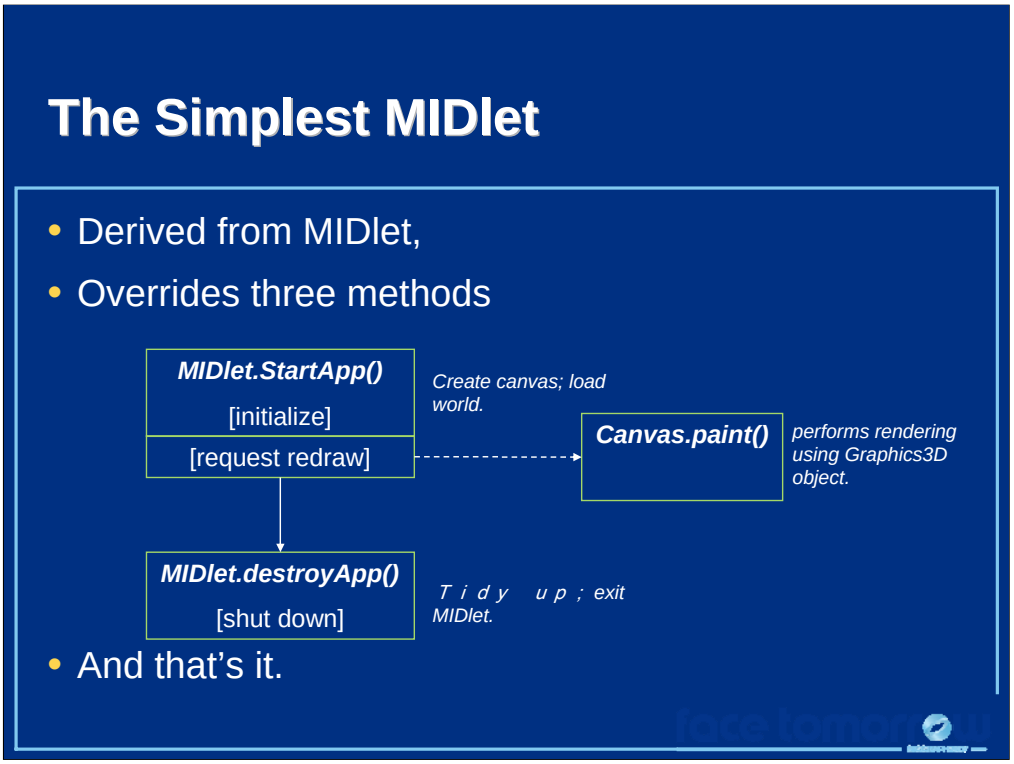
Here we first see the processes involved in debugging on an emulator and then on a real device.

Agenda

- J2ME game development
- Tools
- COFFEE BREAK
- The structure of a M
- A walkthrough a sar m
- Why mobile game nt is hard
- Publishing your content

Agenda

- J2ME game development
- Tools
- COFFEE BREAK
- The structure of a MIDlet
- A walk through a sample game
- Why mobile game development is hard
- Publishing your content



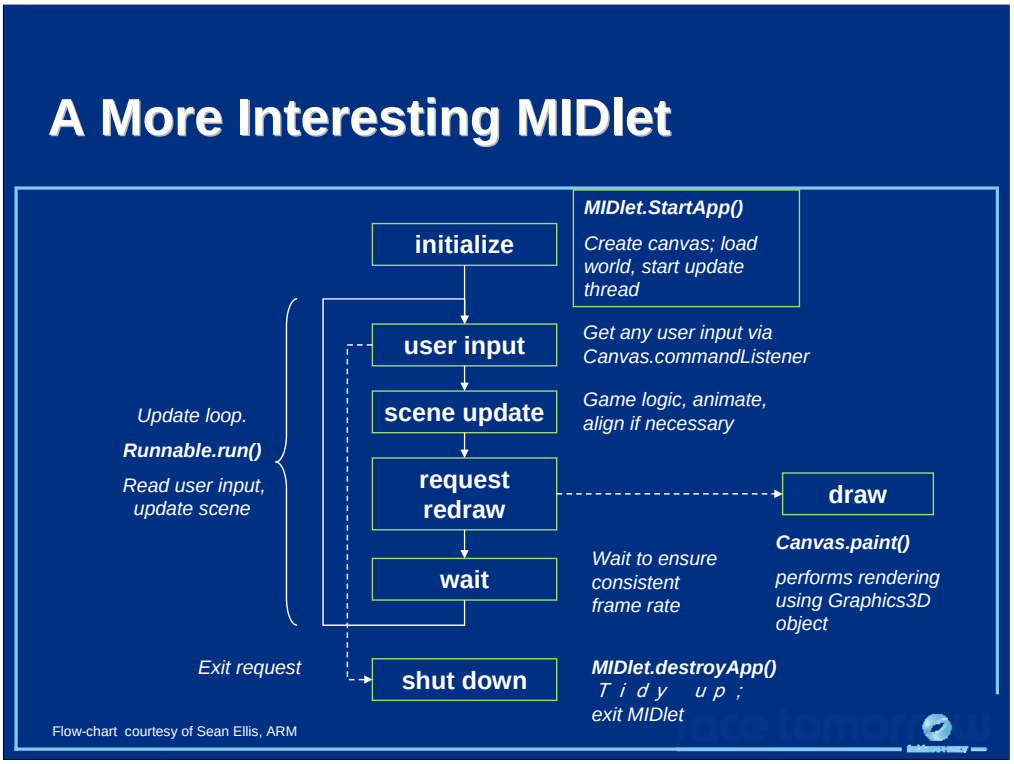
We've looked at creating assets and at tools to use for writing and debugging the programs. What does an actual program look like?

Here we'll look at the structure of a MIDlet, beginning with the simplest possible example? It's a class derived from MIDlet that overrides just 3 methods.

startApp just creates a canvas for display and loads the world to display; it requests a redraw which results in the overridden paint method being called which renders a view to the screen.

destroyApp does some tidying up. And that's it.

Of course, that's not very interesting. We don't get any updates, and the display is static, but it shows the absolute basics. By modifying the world and repainting, you can easily create animated 3D scenes. Let's have a look at the structure of a MIDlet with an update loop.



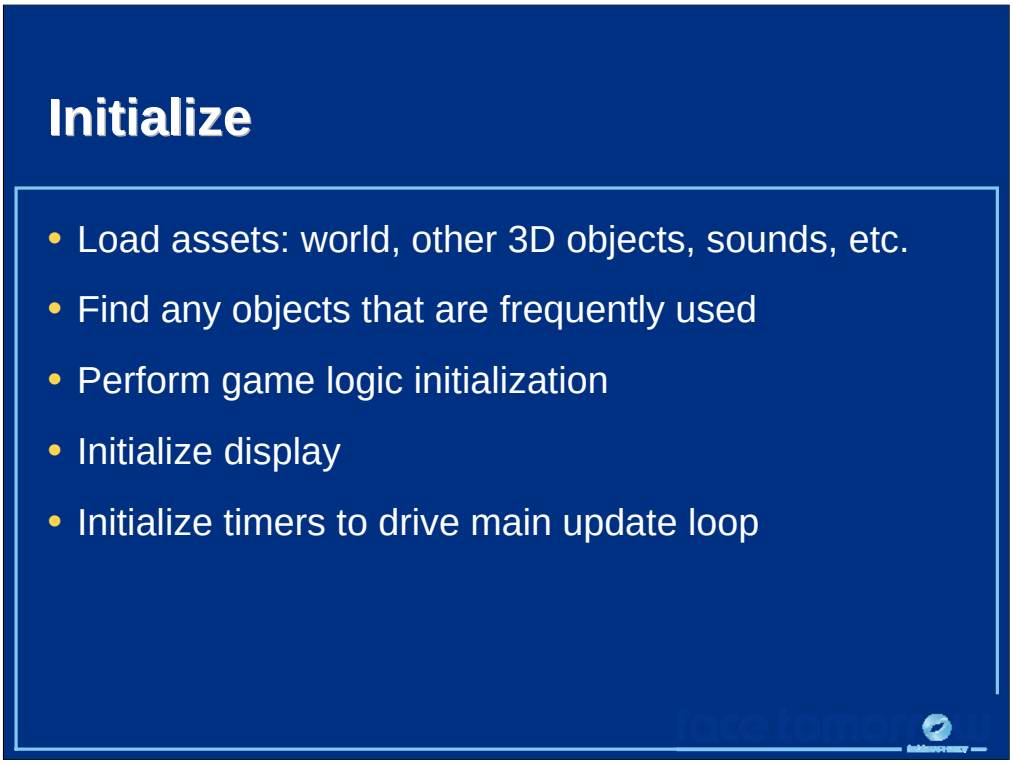
Here's the diagram updated to shown the main update loop. The MIDlet implements the Runnable interface, which means providing one more method, *run()* which contains the update loop.

The update loop reads user input, updates the scene, requests a redraw and then waits until the next frame is scheduled. Waiting ensures a consistent frame rate.

MIDlet Phases

- Initialize
- Update
- Draw
- Shutdown

Let’s look at each of these phases in more detail.



Initialization gets us into a state where we can start the game.

First, we load all the assets we need, both for the 3D scene and any other UI elements, music, sounds, etc.

We should then look up any frequently used objects in the World, to save time in the main game loop. For example, we can find the player's object, any non-player characters, etc. Of course, we need to initialize anything that the actual game logic requires (monster strengths, high-score tables, network connections to other players, or whatever).

Then we initialize the display, and the timers we use to drive the main update loop, and kick off our first update.

Update

- Usually a thread driven by timer events
- Get user input
- Get current time
- Run game logic based on user input
- Game logic updates world objects if necessary
- Animate
- Request redraw

The update is usually attached to timer and other events. Obviously, we need to respond to the user, so getting any input from them is the first thing to do, and get the current time. We get the current time once to avoid problems if the various steps here take significant time.

The next thing to do is to run the game logic based on the user input. While this will be different for each game, the net effect of this is that it updates the state of objects in the world as necessary. Opened a door? Rotate the door object. Picked up a health bonus? Make it invisible, update your health, change size of health bar. Call animate to ensure that any animations actually run, then request a redraw.

Update Tips

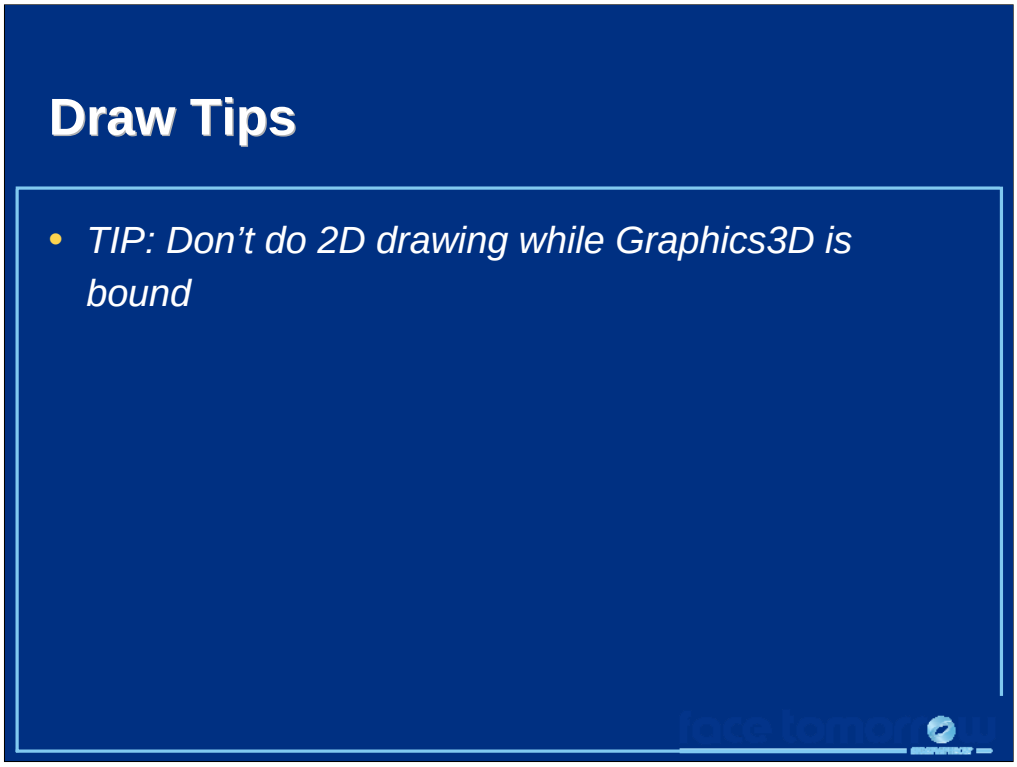
- *TIP: Don't create or release objects if possible*
- *TIP: Call* system.gc() *regularly to avoid long pauses*
- *TIP: cache any value that does not change every frame; compute only what is absolutely necessary*

If at all possible, don't create or release objects in the main loop. If you do have to do this, call system.gc() regularly to ensure that you don't get large garbage collections that ruin the flow of the game. Cache any values that are not changing every frame in order to avoid unnecessary recomputation.

Draw

- Usually on overridden paint method
- Bind Graphics3D to screen
- Render 3D world or objects
- Release Graphics3D
  - …whatever happens!
- Perform any other drawing (UI, score, etc)
- Request next timed update

After each update, we request a redraw. This usually results in a call to an overridden paint method on a canvas. This is fairly simple – we just need to bind the Graphics3D to the screen, render the world, and release it. Remember that there is only one Graphics3D so we need to release it whatever happens! (The best way to do this is in a finally clause.) Then we can do any 2D UI drawing (score, health, etc) and request another update in an appropriate amount of time.

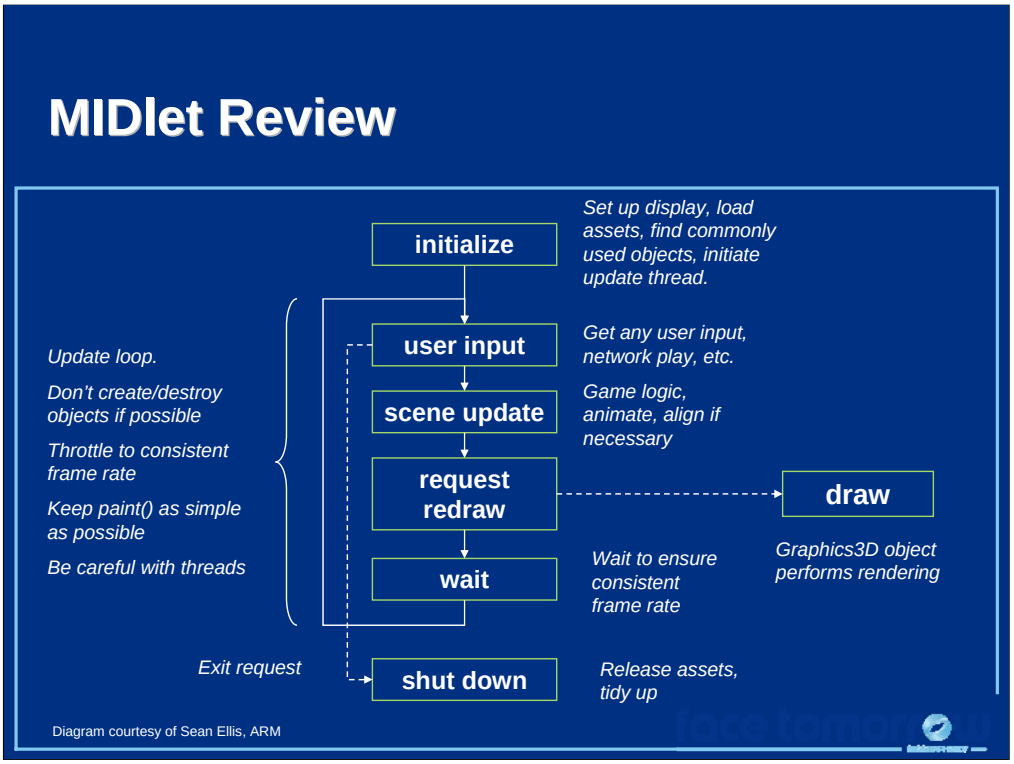


One restriction is that you can’t do 2D drawing while the Graphics3D is bound to the screen, so you have to do it either before or after (or both).

Shutdown

- Tidy up all unused objects
- Ensure once again that Graphics3D is released
- Exit cleanly
- Graphics3D should also be released during pauseApp

On shutdown, we just need to tidy up. It’s usually friendly to ensure that the Graphics3D really has been released before exiting. This should also happen if a call is made to pauseApp, since the new application that is taking over the screen may also need to use 3D.



So, here's a diagram recapping what we have learned.

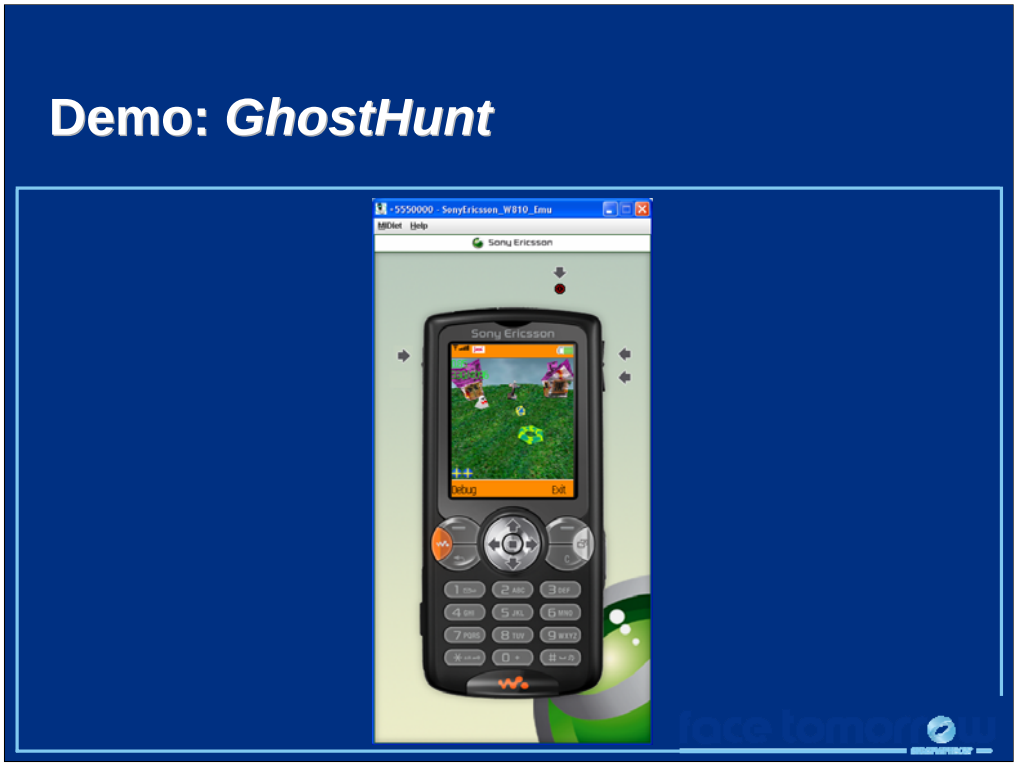
Note that if nothing is happening, we don't need to continually redraw the screen – this will reduce processor load and extend battery life. Similarly, simple scenes on powerful hardware may run very fast; by throttling the framerate to something reasonable, we extend battery life and are more friendly to background processes.

Let's look at a real example

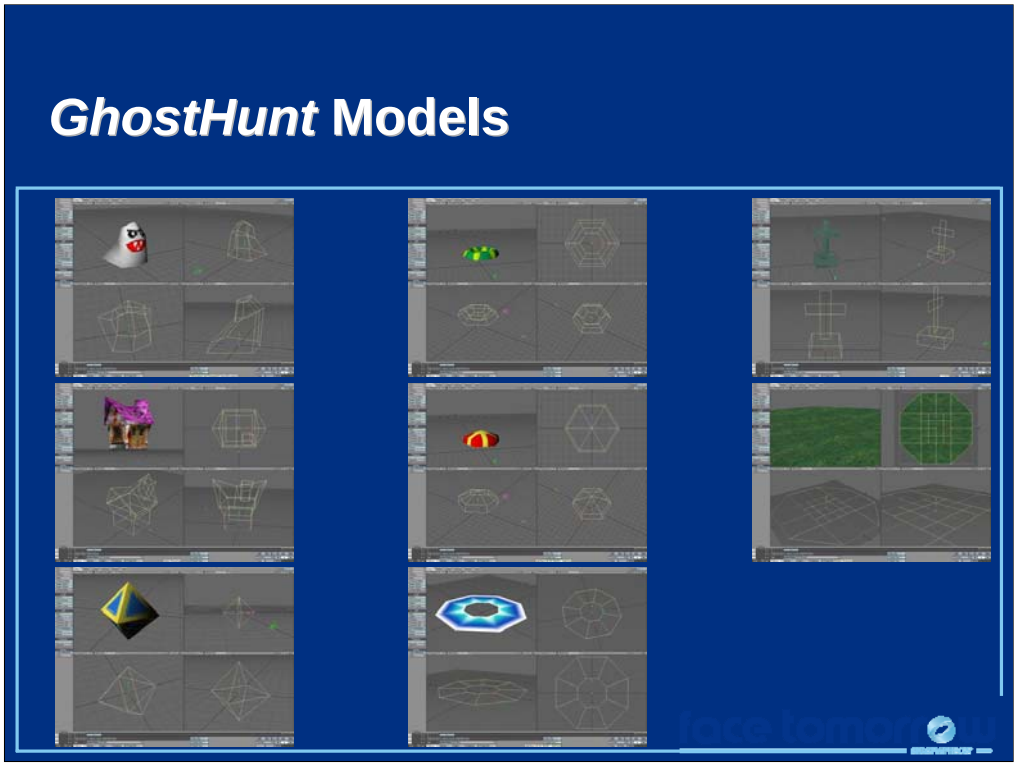
Agenda

- J2ME game development
- Tools
- COFFEE BREAK
- The structure of a MIDlet
- A walkthrough a sample game
- Why mobile game development is hard
- Publishing your content

Demo: *GhostHunt*



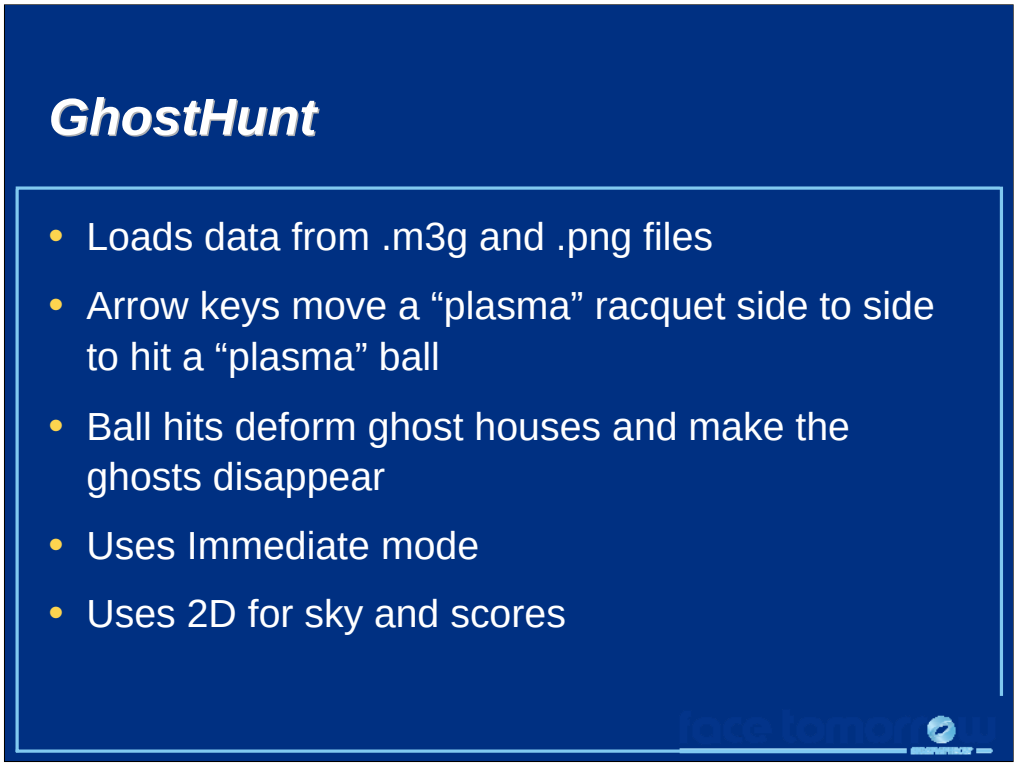
Let’s have a look at the MIDlet in action before diving into the code.



Whack the ghosts & their houses with this plasma ball using this plasma racquet. Occasionally you will receive a power up and the racket changes to this. There are obstacles in the form of crosses. The game takes place on this simple stage.

All models were created in Lightwave and exported to separate M3G files.

Here are all the assets both 2D & 3D used by GhostHunt. Each asset is in a separate file.



GhostHunt uses Immediate mode so as to easily have full control over all the objects.

The course notes for this course on the official DVD have a walkthrough of a Retained mode example, *UsingM3G* instead of *GhostHunt*. An attachment to the course notes contains source code for both *UsingM3G* and *GhostHunt*. The course-notes attachment also contains HI’s *M3G Tutorial* which is also retained-mode based. Please study *UsingM3G* or *M3G Tutorial* at your leisure to learn about using Retained mode.

GhostHunt Framework

- MainApp.java – MIDlet specialization; handles initialization & data loading; contains run thread
- SubApp.java – canvas specialization
- Math2.java – math library

GhostHunt: initialization

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;
import javax.microedition.m3g.*;

class MainApp extends MIDlet implements CommandListener {
  MainApp() {
    exit_command   = new Command("Exit" , Command.EXIT   , 0);
    select_command = new Command("Debug", Command.SCREEN, 0);

     /* Create canvas */
    subapp = new SubApp ();
    subapp.addCommand (exit_command);
    subapp.addCommand (select_command);
    subapp.setCommandListener (this);

    SystemInit ();
    prog_number = PROG_SPLASH;
    WorkInit ();
    GameInit ();
    DataLoad ();
 }
```

Here's the MainApp constructor. The interesting parts will be highlighted in green as we step through:

<Click to animate> First, soft key commands for exit and debug are set up;

<Click> At these 4 lines the canvas is created, the soft key commands are added and MainApp is set up as the listener for those commands.

<Click> This function loads the data. The preceding initialization functions simply initialize a bunch of instance variables. There is nothing worth studying so we'll move on and look at data loading.

GhostHunt: loading data

```
DataLoad() {
  try {
    image [TITLE_SP] = Image.createImage ("/title.png");
    …
  } catch (Exception e) {
    System.out.println ("------------ SP Load");
    ApplicationEnd ();
  }

  try {
    load_data [RACKET_DATA] = Loader.load("/racket.m3g");
  } catch (Exception e) {
    …
  }
  mesh [RACKET_DATA] = (Mesh)load_data [RACKET_DATA][0];
  vbuf [RACKET_DATA] = mesh [RACKET_DATA].getVertexBuffer();
  ibuf [RACKET_DATA] = mesh [RACKET_DATA].getIndexBuffer(0);
  app  [RACKET_DATA] = mesh [RACKET_DATA].getAppearance(0);
  …
}
```

<Click> All of the MIDP images are loaded with Image.createImage.

<Click> Notice that exceptions generated due to a loading error are caught and the MIDlet is terminated.

<Click> M3G Loader.load is used to load all .m3g files. All returned Object3Ds are stored in the *load_data* array.

<Click> Again notice that exceptions are caught.

<Click> After a successful load, the mesh is stored in the global array of meshes and references to the mesh's various components, are stored in other appropriate arrays.

GhostHunt: MIDlet functions

```
public void startApp () {
  thread = new Thread () {
    public void run () {
      GameStart ();
    }
  };
  // Call the new thread's run method.
  thread.start ();
}

public void pauseApp ()
{
    thread = null;
}

public void destroyApp (boolean unconditional)
{
    ApplicationEnd();
}
```

Here are the overrides of startApp(), pauseApp() & destroyApp.

startApp activates the Game Start thread and saves a reference to the thread in an instance variable.

pauseApp & destroyApp are very simple. pauseApp kills the Game Start thread. When the MIDlet manager wants to resume the MIDlet, it calls startApp.

DestroyApp requests the MIDlet to exit. If unconditional is true, the MIDlet must exit.

GhostHunt: GameStart thread

```
void GameStart () {
  Thread thisThread = Thread.currentThread();
  Display.getDisplay (this).setCurrent (subapp);
  while (thread == thisThread) {
    prev_time = now_time;
    do {
     now_time = System.currentTimeMillis ();
    } while ((now_time - prev_time) < SYSTEM_SPEED);
    loop_rate = (now_time - prev_time) / SYSTEM_SPEED;
    if (loop_rate > 5.0f) { /* More than loop rate limit */
      loop_rate = 5.0f;
    }
       /* do game stuff here … */
    try {
      Thread.sleep (1);
    } catch (InterruptedException e) {
      ApplicationEnd ();
    }
  }
}
```

This is the GameStart thread, the MIDlet's heart.

<Click>We have an infinite loop which exits when the instance variable thread no longer contains a reference to the current thread that is saved before the loop is entered. This is the recommended way to kill off a thread in MIDP as it avoid issues with thread interruption.

<Click> At the end of the loop, it sleeps to give other threads a chance to run.

<Click>The loop throttles itself to the predefined SYSTEM_SPEED to provide somewhat predictable performance on different hardware and implementations. and maintains the scene update rate (loop rate) even when drawing is slow.

<Click> At the heart of the loop here, the game functions, which are shown on the next slide, are called.

GhostHunt: “do game stuff here ...”

```
void GameStart () {
  …
  switch (prog_number) {
    case PROG_SPLASH: /* Splash */
     SplashProg ();
     break;
   case PROG_TITLE: /* Title     */
     TitleProg ();
     break;
   case PROG_GAME: /* Game       */
     GameProg ();
     break;
  }
  …
}
```

This is the code that sits at the comment **`do game stuff here …`** on the preceding slide. It simply calls a function appropriate to the current stage as indicated by prog_number. SplashProg would draw the splash screen, if we had one. Instead it simply advances prog_number to PROG_TITLE.

TitleProg watches for the Fire key, while blinking the letters “PUSH START” on and off. After the fire key has been pressed, it does some game initialization and advances prog_number to PROG_GAME.

<**Click**> GameProg is the real meat. It uses key input to move the plasma racquet, animates all the other objects, computing collisions on the way and then draws the scene.

GhostHunt: TitleProg

```
void TitleProg ()
{
  key_dat = subapp.sys_key; /* Get keypresses */

  if ((key_dat & KEY_FIRE) != 0) /* it is fire key */
  {
    racket_tx = 0.0f;
    racket_tz = 0.0f; /* for initializing camera */
    WorkInit ();
    GameInit ();
   …
    prog_number = PROG_GAME;
  }

/*------ Updating------*/
  start_loop++;

/*------ Drawing ------*/
  subapp.repaint ();
}
```

We'll look at TitleProg as it is relatively simple, making it easy to see the relationship with our Canvas override SubApp.

<Click> First of all any key presses are retrieved from our canvas, subapp. These are checked for the Fire key. Then the "scene" is updated.

<Click> In this case, scene update consists of incrementing the loop counter for blinking the letters "Push Start".

Finally the "scene" is drawn by calling supapp's repaint method **<Click>**.

The structure of GameProg is identical. Of course the key handling is different and "scene" update is much more complex. We'll come back to GameProg later.

First we'll look at SubApp.

SubApp: *GhostHunt's* Canvas

```
public class SubApp extends Canvas {
  int cnt;
  static int keydata [] = { UP, LEFT, RIGHT, DOWN, FIRE };
  int length = keydata.length;

  static int sys_key = 0;

  synchronized public void paint (Graphics graphics) { }

  …

  protected void keyPressed (int key) { }

  protected void keyRepeated (int key) { }

  protected void keyReleased (int key) { }
}
```

The Canvas deals with painting the scene and capturing key presses. The key methods of SubApp are shown on this slide. All the other methods are in the service of the paint() method.

<Click> You will recall that paint() is called by the system after the MIDlet has called the repaint() method and repaint is called by MainApp.TitleProg and MainApp.GameProg.

GhostHunt: key handling

```
static int keydata [] = { UP, LEFT, RIGHT, DOWN, FIRE };

protected void keyPressed (int key) {
  for (cnt = 0; cnt < length; cnt++) { /* Search key data. */
    if (getGameAction (key) == keydata [cnt]) {
      sys_key |= (1 << cnt);
    }
  }
}

protected void keyReleased (int key) {
  for (cnt = 0; cnt < length; cnt++) { /* Search key data. */
    if (getGameAction (key) == keydata [cnt]) {
      sys_key &= (~(1 << cnt));
    }
  }
}
```

Before looking at paint(), we'll take a quick look at how key presses are handled.

The key press is converted to the corresponding MIDP game action **<Click>** and if that action exists in our keydata array, the corresponding bit **<Click>** is set in sys_key. Please study the details for yourselves.

SubApp *paint* Method

```
synchronized public void paint (Graphics graphics) {
/*------ select drawing process ------*/
   switch (MainApp.prog_number)
   {
   case MainApp.PROG_SPLASH:
        SplashDraw (graphics);  /* Splash */
        break;

   case MainApp.PROG_TITLE:
        TitleDraw (graphics);   /* Title  */
        break;

   case MainApp.PROG_GAME:
        GameDraw (graphics);    /* Game */
        break;
   }

   Math2.Rand ();
}
```

The paint method is pretty simple. The drawing method appropriate to the stage of the game is called and the random number generator is forced to update its seed.

SplashDraw doesn't do anything and TitleDraw uses standard MIDP drawing. Since we're focused on M3G, we'll look at GameDraw.

GameDraw

```
void GameDraw (Graphics graphics)
{
  …
  graphics.drawImage (MainApp.image[MainApp.BG_SP], 0, 0,
   Graphics.TOP | Graphics.LEFT); /* 2D background sprite */

  MainApp.g3d.bindTarget (graphics);
  MainApp.g3d.clear (MainApp.background);

  /*------ camera setup ------*/
  …
  /*------ draw 3D objects ------*/
  …

  MainApp.g3d.releaseTarget ();

  /*------ draw score, items etc. in 2D ------*/
  …
}
```

GameDraw first draws **<Click>** a 2D background sprite – which is the sky.

Then it binds **<Click>** the Graphics3D instance to the MIDP graphics target and begins 3D drawing. The first step **<Click>** is to clear the buffers by calling Graphics3D.clear(). MainApp.background is set to only clear the depth buffer. **We avoid spending time clearing the color buffer because (a) the sky has already been drawn and (b) our Ground object will fill the rest of the buffer.**

GameDraw then draws the rest of the 3D objects, **<Click>** releases the Graphics3D and draws the score, acquired items and performance data using 2D drawing.

This combination of 2D & 3D works very well when a software renderer underlies M3G but, as was noted earlier, may be less than optimal when a GPU underlies M3G.

GameDraw: camera set-up

```
MainApp.ctrans.setIdentity();
MainApp.ctrans.postTranslate( MainApp.camera_tx,
                              MainApp.camera_ty,
                              MainApp.camera_tz );
MainApp.ctrans.postRotate( MainApp.camera_ry,
                           0.0f, 1.0f, 0.0f );
MainApp.ctrans.postRotate( MainApp.camera_rx,
                           1.0f, 0.0f, 0.0f );
MainApp.ctrans.postRotate( MainApp.camera_rz,
                           0.0f, 0.0f, 1.0f );

MainApp.g3d.setCamera( MainApp.camera, MainApp.ctrans );
```

To set up the camera, GameDraw **<Click>** sets the camera transform for the current position and orientation of the camera. GhostHunt calculates the orientation by computing a rotation around each axis.

Then it sets its camera **<Click>** and this transform into the Graphics3D instance.

Remember that in Immediate mode, the camera's node transform is ignored.

GameDraw: draw 3d objects

```
for (count = 0; count != MainApp.GHOST_MAX; count++)
{
  if (MainApp.ghost_draw_flag [count] != 0) {
    data  = count * 2;
    x     = MainApp.ghost_xz [data + 0];
    z     = MainApp.ghost_xz [data + 1];
    r     = MainApp.ghost_r  [count   ];
    trans = MainApp.trans[MainApp.GHOST_M + count];

    trans.setIdentity   ();
    trans.postTranslate (x, 0.0f, z);
    trans.postRotate    (r, 0.0f, 1.0f, 0.0f);
    trans.postScale     (MainApp.ghost_scale [count],
                         MainApp.ghost_scale [count],
                         MainApp.ghost_scale [count]);

    MainApp.g3d.render (MainApp.vbuf [MainApp.GHOST_DATA],
                        MainApp.ibuf [MainApp.GHOST_DATA],
                        MainApp.app  [MainApp.GHOST_DATA],
                        trans);
  }
}
```

Here we see how the ghosts are drawn. The drawing of all other objects is very similar.

All MainApp variables are set by GameProg which we saw being called regularly by the Game Start thread. We'll study this in a minute.

<Click> Loop through each possible ghost checking a flag to see if it should be drawn.

Retrieve **<Click>** position and rotation information from the arrays of ghost data in MainApp, set up **<Click>** our transform, and **<Click>** draw the submesh that we saw loaded earlier.

If you look at the source code, you will notice that the last 3D object drawn is the Ground object. This is an optimization to ensure that pixels at the locations of foreground objects are drawn only once (not counting times when more than 1 foreground object occupies the same pixel locations).

GameProg

```
void GameProg() {
  key_dat_old = key_dat; /*---- Get key data ----*/
  key_dat     = subapp.sys_key;

  CameraWorldSet ();
  if (Math2.DistanceCalc2D (0.0f, 0.0f, ball_tx,
   ball_tz) > 1.5f) {
    CameraSet (15.0f * (1.0f / loop_rate));
  }

  if (freeze_time == 0) /* The Game is not frozen */ {
    /*------- do game calculations ------*/
    …
  }
  EffectProg ();
  subapp.repaint ();
  …
}
```

At last, we can look at the actual game processing. I'm going to focus on the 3D parts. There are many other parts to GameProg that I will not show today. Please study the source code.

<Click> CameraWorldSet calculates the camera position and orientation so it has a good view of the racquet and the ball.

<Click> CameraSet is called when the ball is far away in order to smooth the camera rotation so the viewer doesn't become seasick.

If the game is not in a frozen state then **<Click>** game calculations are performed.

Finally **<Click>** EffectProg is called to handle effects like making ghosts and ghost houses disappear, and then

<Click> the canvas is told to repaint itself.

GameProg: do game calculations

```
  RacketProg (key_dat, key_dat_old); /*-- Plasma Racket --*/

  if (racket_break_flag != 1) /*- Racket not destroyed -*/
    BallProg ();

  GhostProg ();

  if (racket_break_flag != 1) /*- Racket not destroyed -*/ {
    BallHit          ();      /*--- Collision Decision ---*/
    RacketBreakCheck ();
  }

  house = HouseCheck (); /*------ Final Check ------*/
  if (house == 0)  /* All ghost houses are destroyed. */ {
    /*------ make all remaining ghosts disappear ------*/
    …
    freeze_time = (int)(MOJI_CLEAR_WAIT * (1.0f/loop_rate));
    moji_number = MOJI_CLEAR;
  }
}
```

Here is the heart of the game calculations. We only have time to look at a couple of items.

We'll begin with **<Click>** BallProg and then look at **<Click>** BallHit.

BallProg: compute new ball position

```
void BallProg () {
  …
  ball_speed_rate = ball_speed * loop_rate;

  dis = Math2.DistanceCalc2D(ball_tx, ball_tz, 0.0f, 0.0f);
  pd = Math2.DistanceCalc2D(ball_tx2, ball_tz2, 0.0f, 0.0f);
  if ((dis > 2.0f) && pd > dis)) /* Homing is necessary */ {
    angle = Math2.AngleCalc (ball_tx, ball_tz, 0.0f, 0.0f);
    if (Math2.DiffAngleCalc (angle, ball_vec) > 0.0f) {
      ball_vec -= (0.6f * loop_rate);
    } else {
      ball_vec += (0.6f * loop_rate);
    }
  }
  Math2.RotatePointCalc (ball_speed_rate, ball_vec);
  ball_tx2 = ball_tx; /* Save the previous coordinates */
  ball_tz2 = ball_tz;
  ball_tx += Math2.calc_x;
  ball_tz += Math2.calc_y;
}
```

BallProg computes **<Click>** a new ball position based on the ball speed and direction vector.

<Click> The new position is saved in MainApp instance variables.

<Click> Before computing a new position, BallProg checks to see if the ball is moving too far away from the player.

<Click> If so, the ball's direction vector is adjusted here.

BallHit

```
void BallHit () {
  …
  /*------ racket collision detection ------*/
  …
  /*------ ghost house collision detection ------*/
  …
  /*------ ghost collision detection ------*/
  …
  /*------ obstacle (cross) collision detection ------*/
  …
  /*------ warp hole collision detection ------*/
  …
  /*------ check for outside the field ------*/
  …
}
```

BallHit checks for collisions between the ball and all the game objects, the racquet, ghosts, ghost houses etc. They are all quite similar.

We'll look at just one: **<Click>** racquet collision.

BallHit: racket collision detection

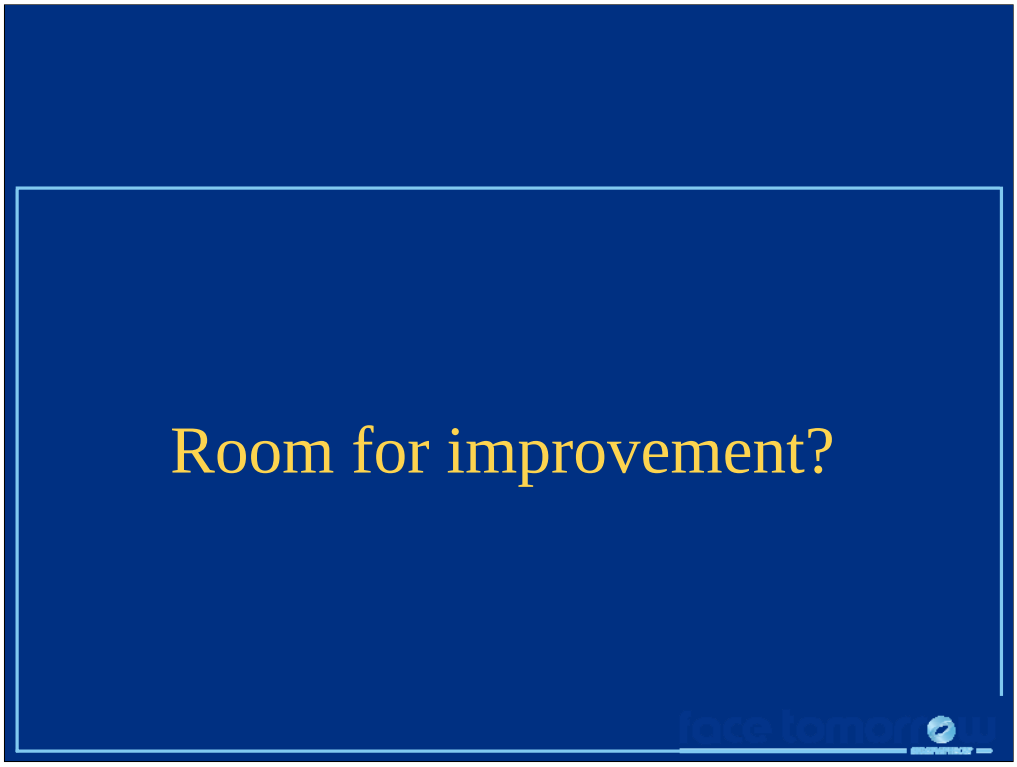
```
void BallHit () {
  …   /* final static int Math2.ANGLE = 360 */
  dist = Math2.DistanceCalc2D (racket_tx, racket_tz
                               ball_tx, ball_tz);
  if (dist <= BALL_RACKET_DISTANCE) {
    angle = Math2.AngleCalc (ball_tx, ball_tz, racket_tx,
                             racket_tz);
    diff  = Math2.DiffAngleCalc(angle, ball_vec
                                     + (Math2.ANGLE/2.0f));
    if (Math2.Absf (diff) > (Math2.ANGLE / 4.0f)) {
      /* Feasible angle for collision */
       ball_vec = angle + (diff * -1.0f);

       Math2.RotatePointCalc (ball_speed_rate, ball_vec);
       ball_tx  = ball_tx2 + Math2.calc_x;
       ball_tz  = ball_tz2 + Math2.calc_y;
     }
  }
  …
}
```

<Click> First the distance between the ball and the racquet is calculated.

<Click> If close enough for a possible collision, the angle between ball & racquet is compared with the ball vector.

<Click> If the angle is sufficient, a collision is deemed to have happened and the new ball vector is computed.


Room for improvement?

Improvement 1: simpler drawing

```
for (count = 0; count != MainApp.GHOST_MAX; count++)
{
  if (MainApp.ghost_draw_flag [count] != 0) {
    …
    MainApp.g3d.render (MainApp.vbuf [MainApp.GHOST_DATA],
                        MainApp.ibuf [MainApp.GHOST_DATA],
                        MainApp.app  [MainApp.GHOST_DATA],
                        trans);
    MainApp.g3d.render (MainApp.mesh[MainApp.GHOST_DATA],
                        trans)
  }
}
```

If you recall, ProgDraw() uses the sub-mesh variant of Graphics3D.render(). This requires that many arguments must be passed across the KVM Native Interface (KNI) and also that references to all of the mesh components be stored in the Java heap.

In this case, the same result **<Click>** can be achieved by using the node variant of Graphics3D.render().

Could GhostHunt use the world variant of Graphics3D.render()? Yes it could. It would not result in much performance gain in this case as a graph of this game's scene would have only a single level.

Improvement 2: no busy waiting

```
while (thread == thisThread) {
  prev_time = now_time;
  do {
    now_time = System.currentTimeMillis ();
  } while ((now_time - prev_time) < SYSTEM_SPEED);
  loop_rate = (now_time - prev_time) / SYSTEM_SPEED;
  if (loop_rate > 5.0f) { /* More than loop rate limit */
    loop_rate = 5.0f;
  }
/* do game stuff here … */
  try {
    Thread.sleep (1);
  } catch (InterruptedException e) {
    ApplicationEnd ();
  }
}
```

You will recall that the run loop in GameStart did busy waits. We can modify it so that it sleeps which is usually more system friendly.

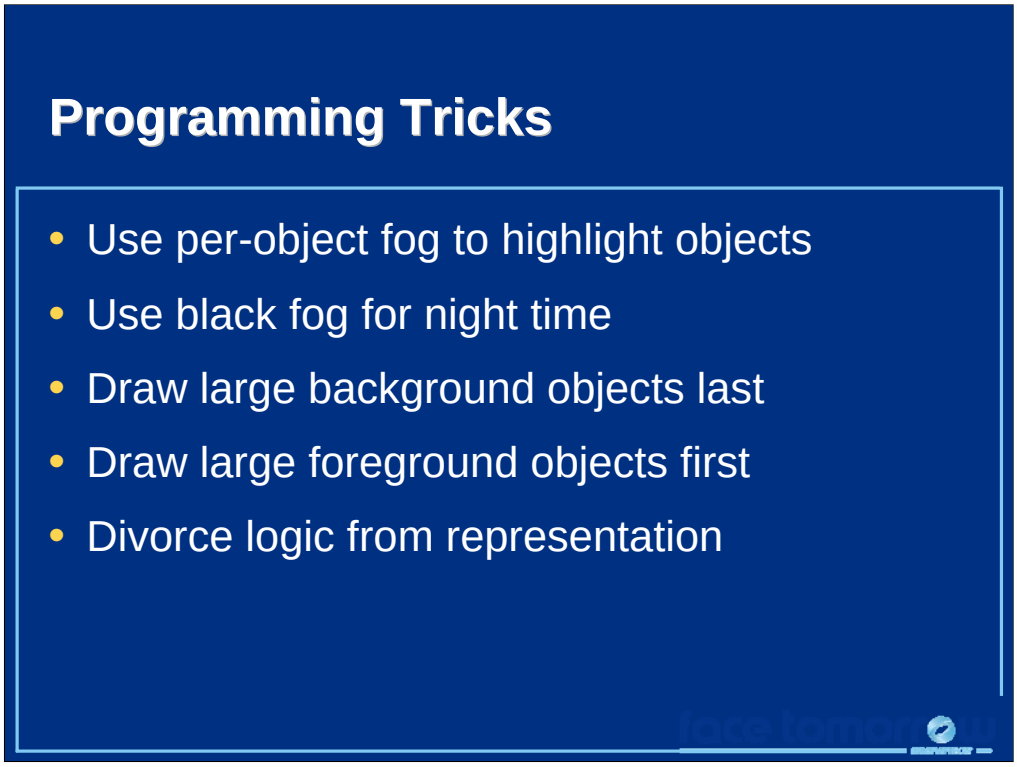
Improvement 2: no busy waiting

```
while (thread == thisThread) {
  prev_time = now_time;
  do {
    now_time = System.currentTimeMillis ();
  } while ((now_time - prev_time) < SYSTEM_SPEED);
  now_time = System.currentTimeMillis();
  long sleep_time = SYSTEM_SPEED + prev_time - now_time;
  if (sleep_time < 0)
    sleep_time = 1; /* yield anyway so other things can run */
  try {
    Thread.sleep(sleep_time);
  } catch (InterruptedException e) {
    ApplicationEnd ();
  }
  if (thread != thisThread) return;
  now_time = System.currentTimeMillis ();
  loop_rate = (now_time - prev_time) / SYSTEM_SPEED;
  if (loop_rate > 5.0f) { /* More than loop rate limit */
    loop_rate = 5.0f;
  }
/* do game stuff here … */
}
```

In current CLDC & CDC implementations, only one application at a time can run so what this buys you is battery friendliness. However CLDC & CDC will soon feature multi-tasking virtual machines (MVM) so will become even more important to not busy wait.

Note 1: it is necessary to check the variable “thread” to see if the main thread requested an exit while this thread was sleeping because variables used by /* do game stuff here … */ may have been freed. We could use Thread.interrupt() to terminate the thread but that is not supported in CLDC 1.0 and old habits die hard.

Note 2: the “thread” check should be right after the sleep for best performance but then I would have had to modify the animation on these slides. That is too painful given the unbelievable number of bugs and misfeatures in this part of PowerPoint.



You always knew there was a reason for per-object Fog!!

Note: When rendering in Retained mode (World, Group) most implementations will attempt to sort the submeshes into the most effective order for the underlying renderer.

One tip that works well is to divorce the logic from the representation. Instead of rotating a door object to open it, just start the “Open Door” animation. This creates fewer dependencies between the assets and the logic, and allows the asset designers to use rotating, sliding, dilating or exploding doors as they see fit.

Agenda

- J2ME game development
- Tools
- COFFEE BREAK
- The structure of a MIDlet
- A walkthrough a sample game
- Why mobile game development is hard
- Publishing your content

The number one problem developers face is device fragmentation.

The second problem is … **<Click>** and the third problem is … **<Click>**

Fragmentation occurs because operators, handset makers and even infrastructure vendors are struggling to differentiate their wares to fend off looming commoditization and because of poor standards and implementations. The result is that content makers have to make sometimes hundreds of SKUs (Stock Kontrol Units) of a single game.

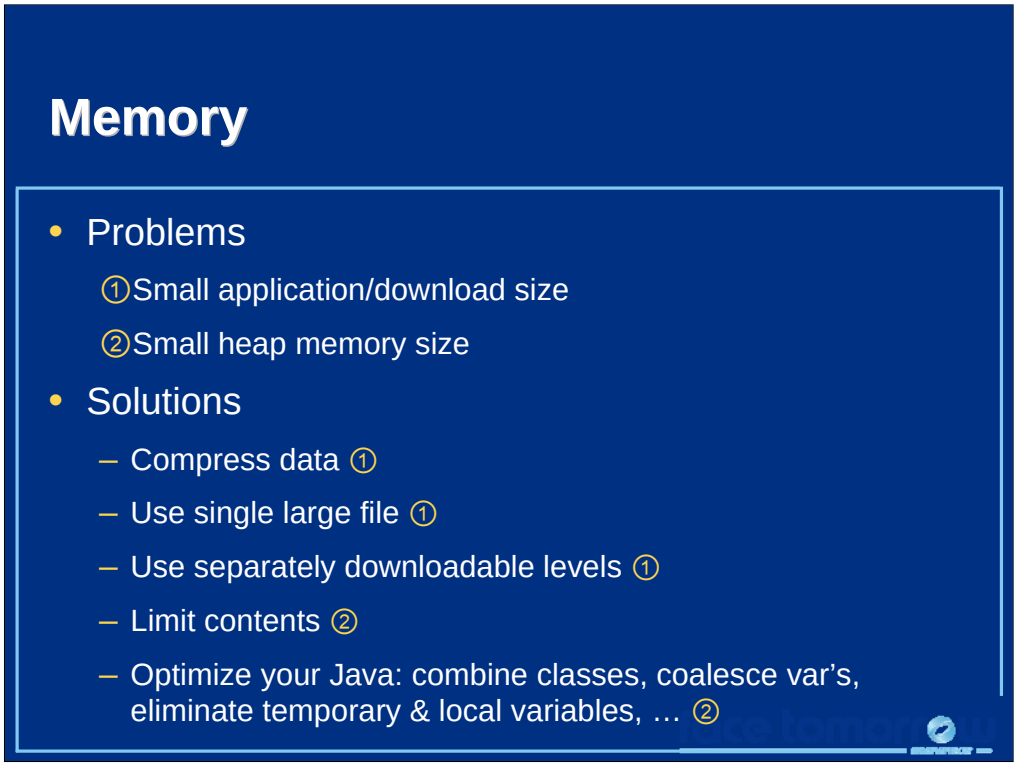
This is an interesting paper on how to alleviate some of the problems when using NetBeans. **J2ME MIDP Device Fragmentation Tutorial with Marv The Miner -** http://www.netbeans.org/kb/articles/tutorial-j2mefragmentation-40.html

Download size limits are increasing thanks to 3G but 256k is still a common size limit.

Poor Input Devices: Input devices are typically limited to the 12 key-pad plus a navigation array and a few extra buttons. Yes, game console style pads are coming but they are still the rare exception.

Why Mobile Game Development is Hard

- No floating point hardware
- No integer divide hardware
- Many tasks other than application itself
  - Incoming calls or mail
  - Other applications
- Short development period
- Tight $100k – 250k budget



Let's look at some of the issue in detail and see some solutions.

…

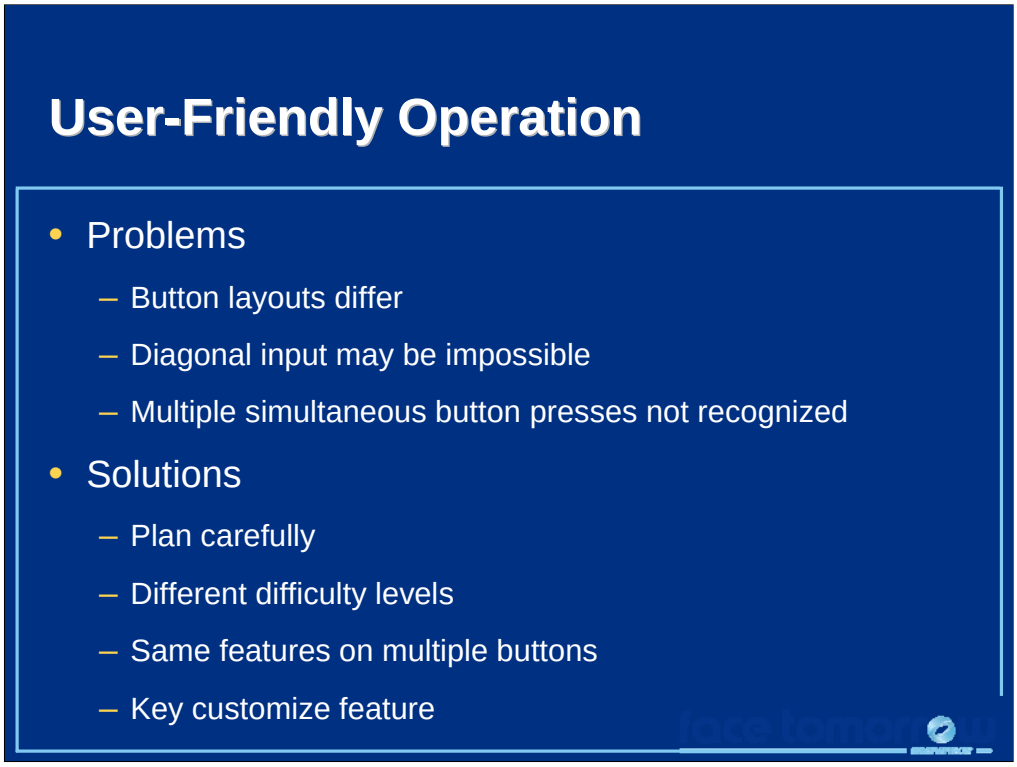
There is a > 70 byte overhead per file in the JAR file, an overhead which is dependent on the path length. Also zip compression does not work across files.

At least one tool is available for automatic optimization of heap and file sizes: Innaworks mBooster: http://www.innaworks.com/mBooster.html. People from this company gave an interesting talk at GDC 2007 on "Pushing the size and performance of JavaME games on today's handsets". Unfortunately I've not been able to find the slides on-line.

Note that in some implementations Loader.load("/img.png") will load the image file via a MIDP image because native code is unable to read from a java stream. This requires more memory during loading. In such cases, use a .m3g file containing an Image2D instead.

Performance

- Problems
  - ① Slow system bus & memory
  - ② No integer divide hardware
- Solutions
  - Use smaller textures ①
  - Use mipmapping ①
  - Use byte or short coordinates and key values ①
  - Use shifts ②
  - Let the compiler do it ②



What is most important in the game is the operation, which functions as a communication line between the player and the game. Even within the same group of handsets, the sense of operation differs by how the buttons are placed, which as a result changes the difficulty of the game itself. These issues must be considered very carefully from the planning stage.

When porting onto other types of handsets, game operation is one of the items that generates problems in the development. For example, diagonal input may have worked on the original handset whereas it may be unavailable on the handset to which the game is being ported. Also there are some cases where handsets fail to recognize more than one button being pressed at the same time.

I cannot provide you with an overall solution; however, I would like to introduce you some examples of how HI coped with these issues in our past content.

1) Types of handset can be discerned to diversify the difficulty of the contents.

2) Let the player play at a lower difficulty level when diagonal input is ineffective by keeping a diagonal input flag in the program. When the diagonal input becomes effective, then the game can switch to its normal level of the difficulty.

3) Allocate the same features, such as “jump” and “attack” to multiple buttons or embed a key customize feature.

With these countermeasures, the problems can be alleviated to an extent. Depending on the types of the game, there may be more efficient ways to solve the problem. This is where planners and programmers can leverage their ideas.

Many Other Tasks

- Problem
  - Incoming calls or mail
  - Other applications
- Solution
  - Create library for each handset terminal

Agenda

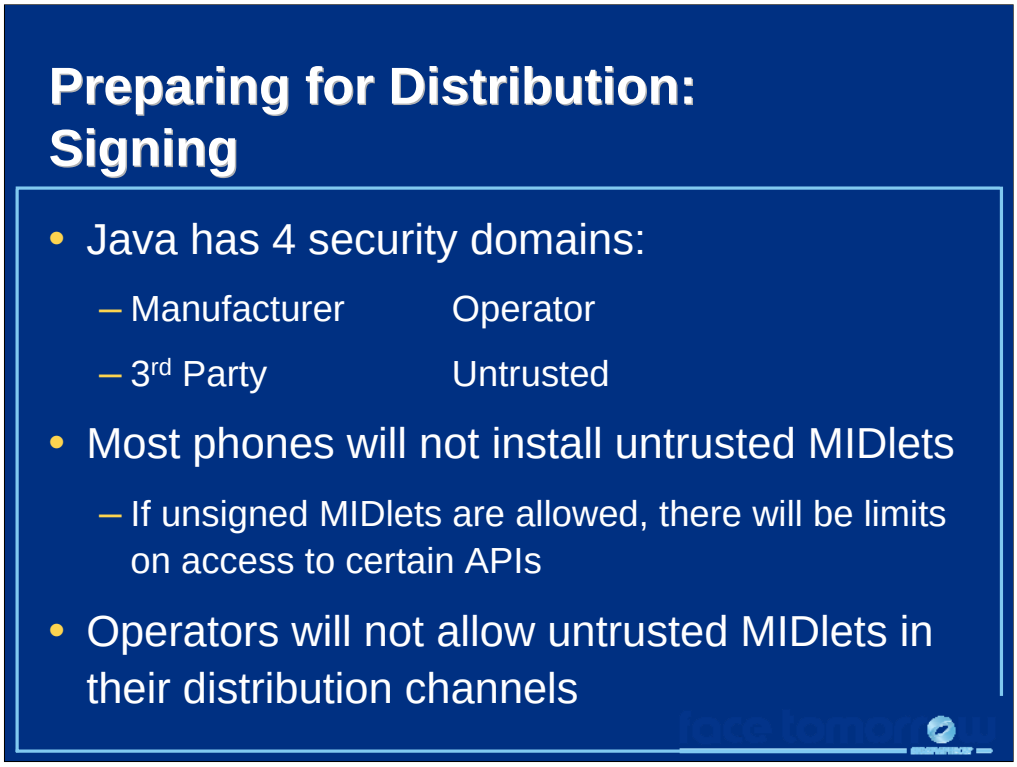
- J2ME game development
- Tools
- COFFEE BREAK
- The structure of a MIDlet
- A walkthrough a sample game
- Why mobile game development is hard
- Publishing your content

Publishing Your Content Agenda

- Publishing your content
  - Preparing contents for distribution
  - Getting published and distributed

Preparing for Distribution: Testing

- Testing on actual handsets essential
  - May need contract with operator to obtain tools needed to download test MIDlets to target handset.
  - May need contractor within operator's region to test over-the-air aspects as handset may not work in your area
- Testing services are available
  - e.g. www.tirawireless.com



J2ME phones can have multiple trusted 3rd party domains.

Examples of API access limitations:

- no access to the phone book
- certain API requests will have to be authorized by the user *each* time they are called

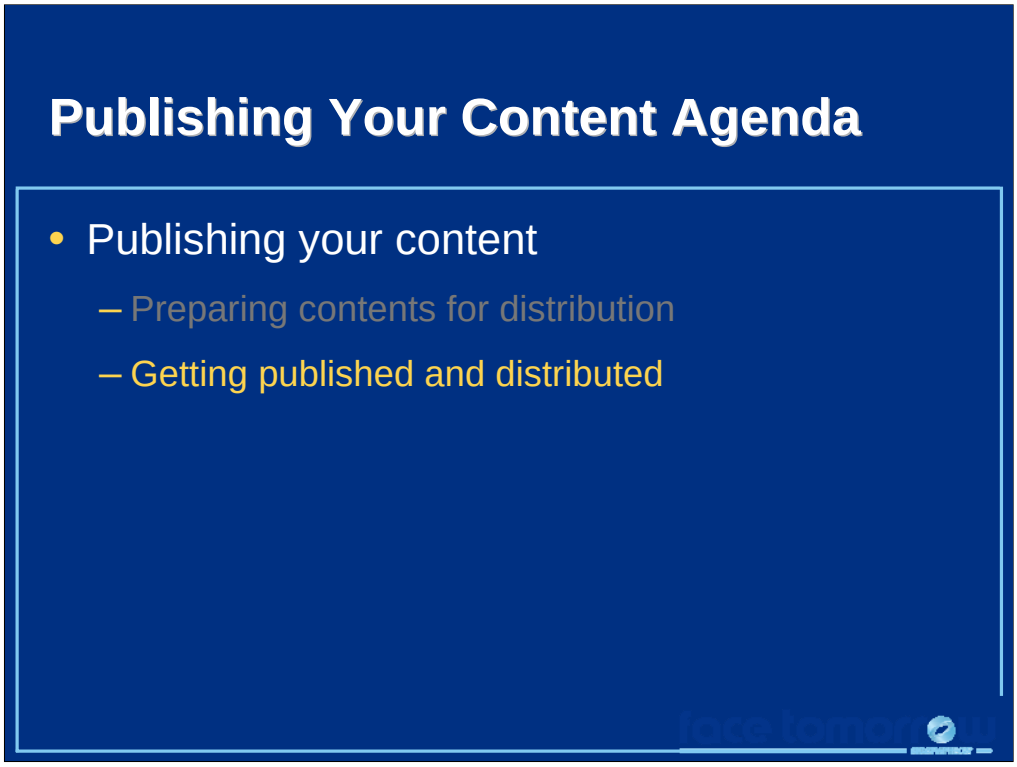
Preparing for Distribution: Signing

- Your MIDlet must be certified and signed using a 3rd party domain root certificate
- Method varies by operator and country
  - Many makers and operators participate in the Java Verified Program to certify and sign MIDlets for them
- To get certification, MIDlet must meet all criteria defined by JVP and must pass testing

URL is www.javaverified.com

After successful testing the JVP signs the MIDlet and returns your signed MIDlet to you.

Native applications using OpenGL need to go through similar programs such as True BREW.



The following section is useful for all mobile game developers not just J2ME/M3G developers.

Publishing Your Content: Distribution Channels

- Game deck
  - e.g. “More Games button”
- Off deck, in portal
  - e.g. Cingular’s *Beyond MEdia Net*
- Off deck, off portal
  - Independent of operator
  - Premium SMS or web distribution

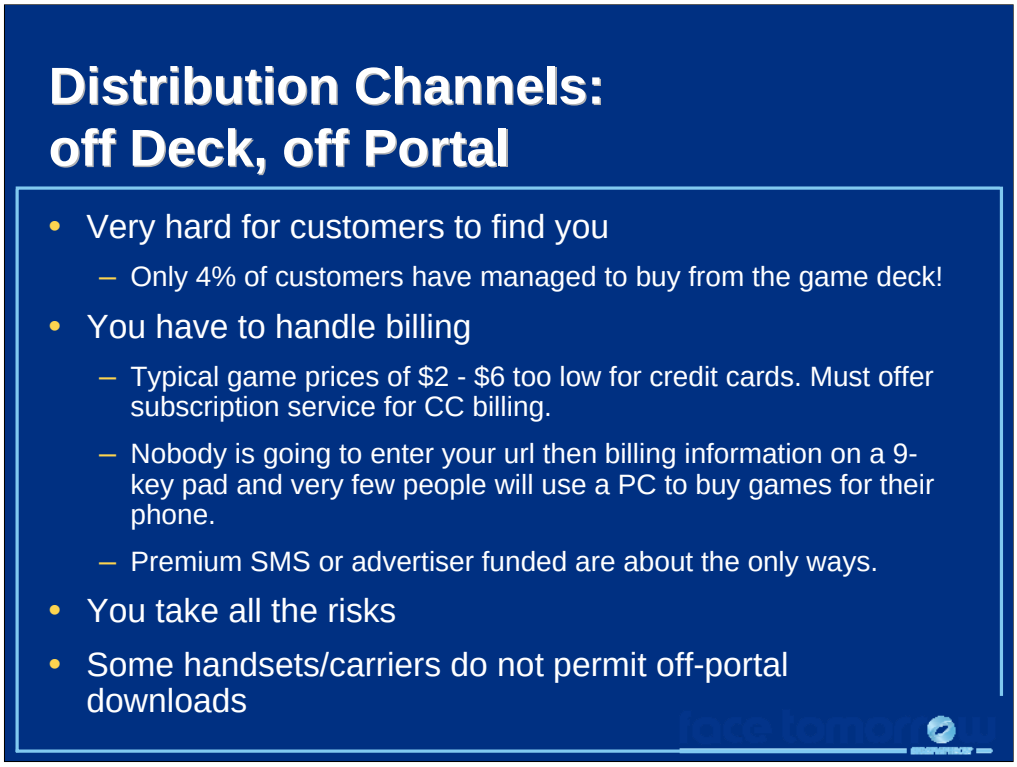
Distribution Channels: Game Deck

- Customers find you easily
  - but many carriers only allow a few words of text to describe and differentiate the on-deck games
- Operator does billing
  - No credit worries
- Operator may help with marketing
  - or they may not
- Shelf space limited

Due to the limited shelf space and the small content acquisition teams at the operators, virtually the only way to get “on deck” is to work with a publisher who is already “on deck”.

Distribution Channels: off Deck, in Portal

- Hard to find you. Need viral marketing
  - Customers must enter search terms in operator's search box
  - or find URL in some other way
- Operator does billing, may help with marketing
- May be able to get here without a publisher



This is a hard row to hoe.

Don't even think about non-over-the-air distribution for mobile. It's not the way mobile works especially in markets like Japan where far more people have "keitai denwa" than have PCs. Also MIDlet downloads from PC's are disabled in many handsets.

Furthermore some operators disable MIDlet downloads from anywhere but their own portals.

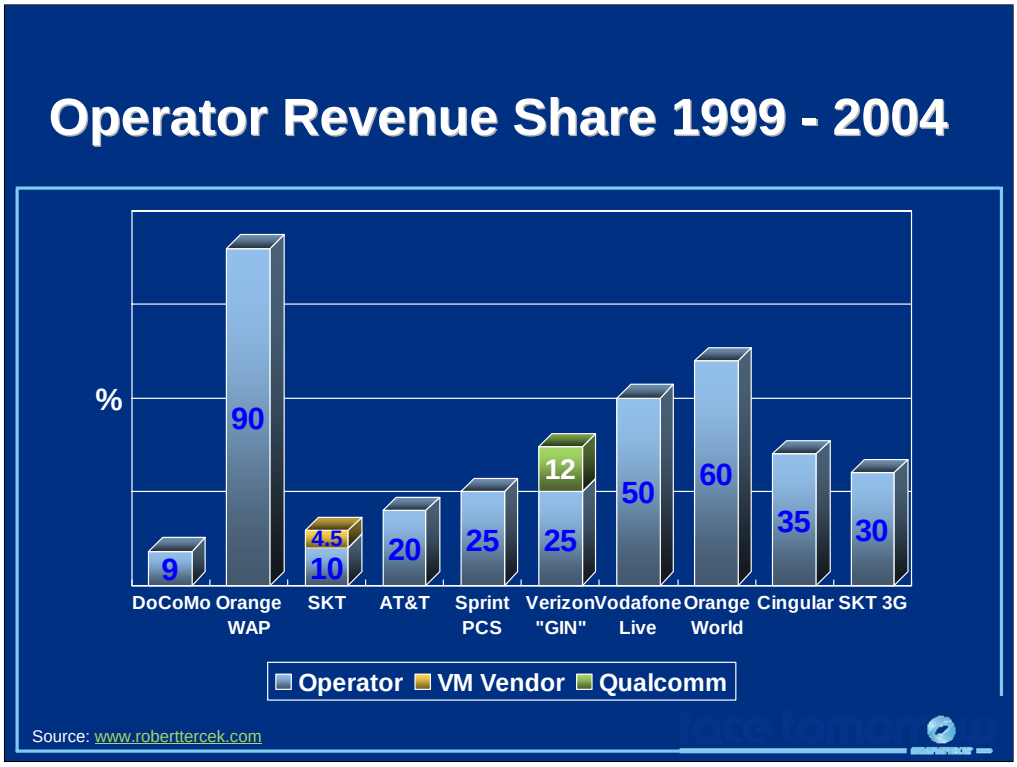
The best way to get wide distribution is to team up with an operator-approved publisher.

Let's look at some of the billing mechanisms in place …

The subscription model is not popular in Europe, in part due to some sleazy marketing practices by some unscrupulous portals that would slam consumers with unwanted subscriptions.

Games are often adapted to the billing mechanisms in order to generate billable events, for example premium SMS games that require periodic refills of scarce supplies.

Micropayments are small payments charged to the customer's phone bill. Because it increases ARPU, operators, of course, prefer the subscription model.



What is the operators' revenue share for providing deck space, billing services and, perhaps, marketing? Here are some numbers.

Orange France, needless to say, doomed many European publishers to bankruptcy

Sprint later raised their share to 30 - 35%

Verizon GIN is Verizon's Get it Now service based on BREW.

Vodafone live provides free airtime on browsing for games.

"Some carriers take 60% but they genuinely act as the retailer and create excellent results through their investment in marketing. Others take 35-40% and leave the publishers to do most of the marketing." attributed to the GM of a European game publisher.

Going On-Deck

- Find a publisher and build a good relationship with them
- **Japan:** Square Enix, Bandai Networks, Sega WOW, Namco, Infocom, etc.
- **America:** Bandai America, Digital Chocolate, EA Mobile, MForma, Sorrent
- Europe: Digital Chocolate, Superscape, Macrospace, Upstart Games

If you choose to go for the on-deck route, find a publisher already on the operator's deck and build a good relationship with them.

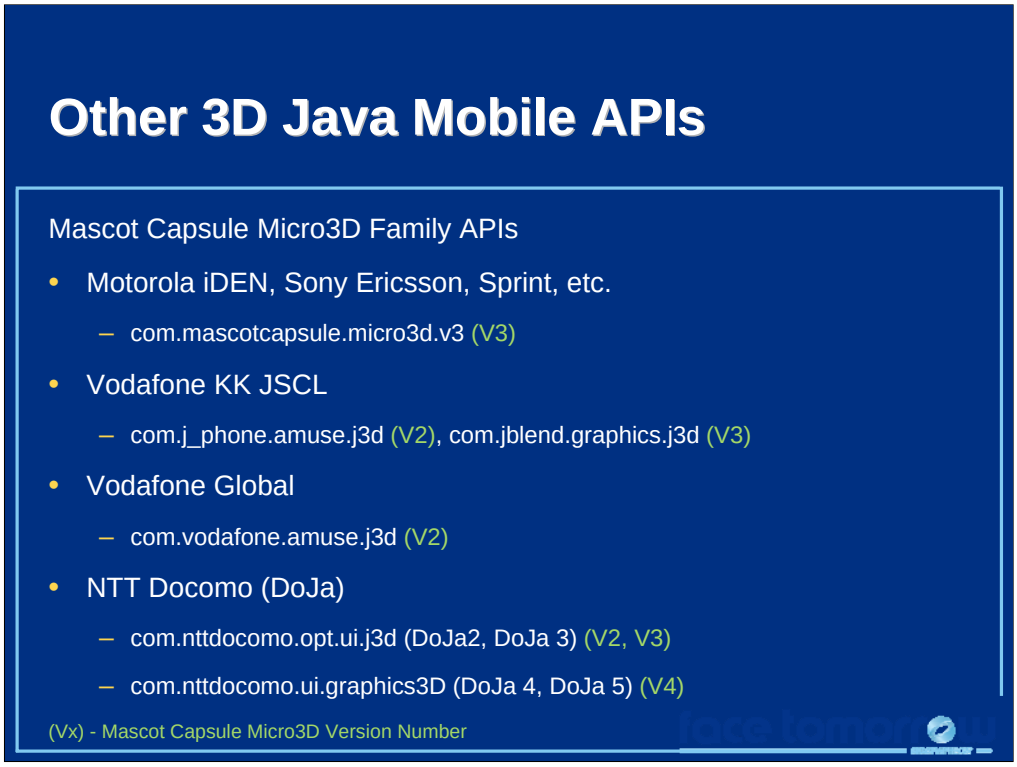
As with books, a good publisher should help with marketing as well as distribution.

Going Off-Deck

- There are off-deck distribution services:
  - thumbplay, www.thumbplay.com
  - playphone, www.playphone.com
  - gamejump, www.gamejump.com free advertiser supported games
- These services may be a good way for an individual developer to get started

Show one of the web sites!!

This concludes the main agenda.



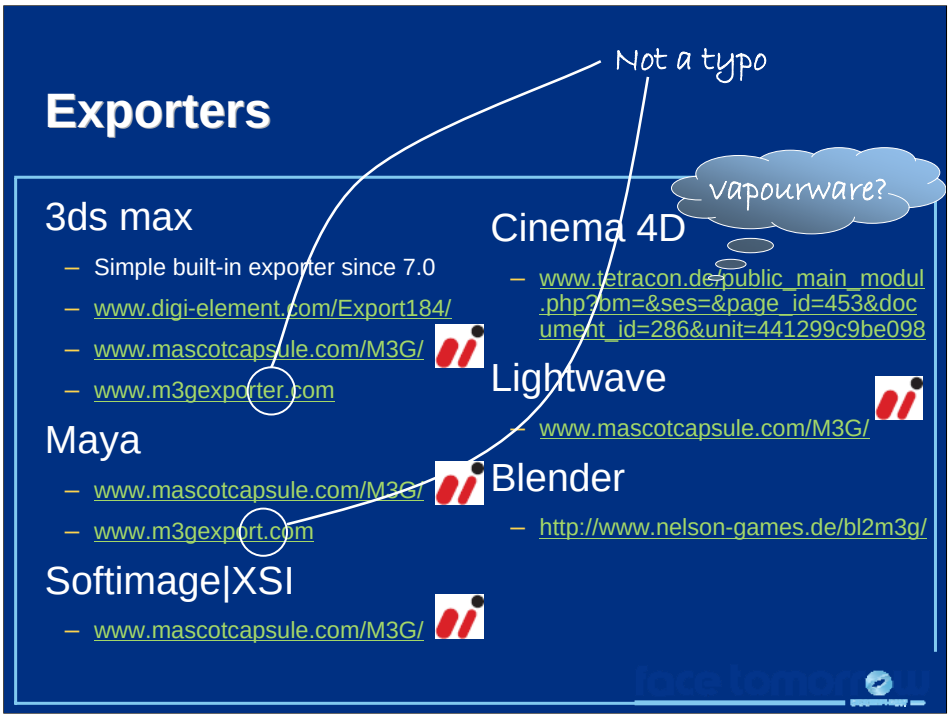
For sake of completeness, I'll mention some other 3D Java APIs you will find on various mobile devices. These are all based on HI's MascotCapsule Micro3D Engine. MascotCapsule Micro3D Version 3 pre-dates M3G by 1 year. Version 4 supports M3G. The APIs above are found on many handsets.

Mascot Capsule V3 Game Demo

Just because it's a really cool game…

Summary

- Use standard tools to create assets
- Many J2ME SDKs and IDEs are available
- Basic M3G MIDlet is relatively easy
- Programming 3D Games for mobile is hard
- Getting your content marketed, distributed and sold is a huge challenge



m3gexport.com under Maya is NOT a typo.

Cinema4D plug-in appears to be vapourware.

SDKs

- Motorola iDEN J2ME SDK
  - idenphones.motorola.com/iden/developer/developer_tools.jsp
- Nokia Series 40, Series 60 & J2ME
  - www.forum.nokia.com/java
- Softbank MEXA & JSCL SDKs
  - developers.softbankmobile.co.jp/dp/tool_dl/java/tech.php
  - developers.softbankmobile.co.jp/dp/tool_dl/java/emu.php

The first Softbank URL is for the downloading the documentation. The second is for downloading the emulators. Softbank developer support is all in Japanese.

You need to complete a simple registration before you can download the SDK but the web page is in Japanese. There are 2 SDKs. MEXA (**M**obile **E**ntertainment e**X**tension **A**PI) and JSCL (**J**-Phone **S**pecific **C**lass **L**ibraries) Both are based on Sun's Wireless Toolkit (WTK). You'll need the MEXA SDK for M3G. Look for "MEXA SDK" and click on the link which says "ツール". You can click through the Privacy Policy and License by clicking the button labeled "同意する" (agree).

The SDK also contains Mascot Capsule V3 support – com.jblend.graphics.j3d.

Eclipse plug-ins are available for both the MEXA & JSCL SDKs.

Version 2.5.1 of Sun's Java Wireless Toolkit (WTK), released in June 07, is the first version available for Linux as well as Windows.

The Sony Ericsson, Sprint and Vodafone SDKs are all based on WTK.

Vodafone global requires you become a partner of Via Vodafone in order to obtain the SDK. You have to submit a questionnaire about your content and business plan before they will even talk to you. Very unfriendly! However since the SDK is just WTK with Vodafone skins for the emulator windows and support for some additional Vodafone specific APIs, you can go a long way without it.

Softbank's MEXA SDK is an easier-to-obtain alternative to Vodafone's. Because Softbank used to be Vodafone KK, they have handsets with Vodafone VSCL APIs, and their SDK still contains support for them.

IDE's for Java Mobile

- Eclipse Open Source IDE
  - www.eclipse.org & eclipseme.org
- JBuilder 2005 Developer
  - www.borland.com/jbuilder/developer/index.html
- NetBeans
  - www.netbeans.info/downloads/index.php
  - www.netbeans.org/products/
- Comparison of IDE's for J2ME
  - www.microjava.com/articles/J2ME_IDE_Comparison.pdf

The open source Eclipse IDE is largely written in Java and has many java development tools. The additional EclipseME component is needed for Java ME development.

Like Eclipse, NetBeans is free. It seems to have replaced Sun Java Studio.

All of these IDE's rely on Sun's Wireless Toolkit or other UEI-compliant wireless toolkit for platform emulation.

The "Comparison of IDE's" paper, written in 2002, is a little out of date now.

Other Tools

- Macromedia Fireworks
  - www.adobe.com/products/fireworks/
- Adobe Photoshop
  - www.adobe.com/products/photoshop/main.html
- Sony SoundForge
  - www.sonymediasoftware.com/products/showproduct.asp?PID=961
- Steinberg Cubase
  - www.steinberg.de/33_1.html
- Yamaha SMAF Tools
  - smaf-yamaha.com/

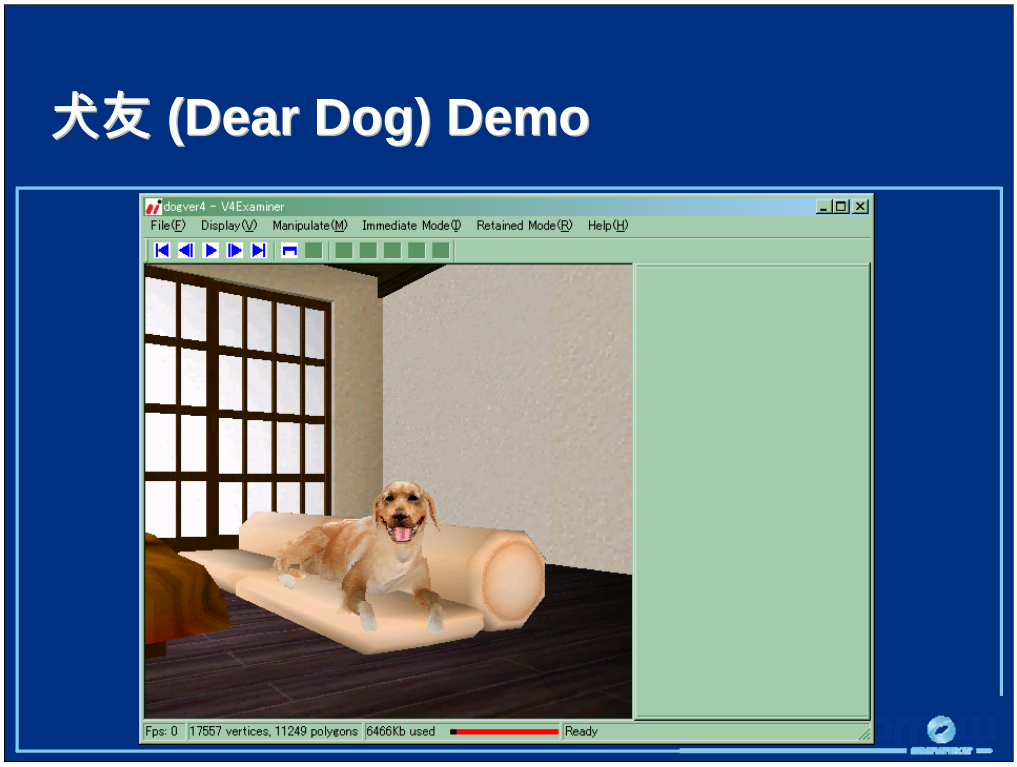
Other Tools

- Java optimizer - Innaworks mBooster
  - www.innaworks.com/mBooster.html
- Porting Platforms
  - www.tirawireless.com
  - www.javaground.com

Services

- MIDlet verification & signing
  - www.javaverified.com
- Porting & testing
  - www.tirawireless.com
- Off deck distribution
  - www.thumbplay.com
  - www.playphone.com
  - www.gamejump.com

犬友 (Dear Dog) Demo



While I take your questions, I'll leave a final demo running. We created this to show the richness that is technically possible with M3G. When we first made this animation in late 2003, it was too big to load into a real phone. There are several phones today that can load it but the frame rate is not good – most likely due to the skinning being done in software.

Demonstrate dog animation


SIGGRAPH2007

M3G 2.0
Sneak Preview

Tomi Aarnio

Nokia Research Center

What is M3G 2.0?

- Mobile 3D Graphics API, version 2.0
  - Java Specification Request 297
  - Successor to M3G 1.1 (JSR 184)
- Work in progress
  - Early Draft is out for review (www.jcp.org)
  - Developer feedback is much appreciated!

Who's Behind It?

Hardware vendors • AMD, ARM • NVIDIA, PowerVR	Device makers • Nokia, Sony Ericsson • Motorola, Samsung
Platform providers • Sun, Ericsson • HI, Aplix, Acrodea	Developers • Digital Chocolate • RealNetworks • Superscape

Graphics hardware vendors, device vendors and platform providers are well represented in the Expert Group.

We could use more contribution from developers, though.

M3G 2.0 Preview

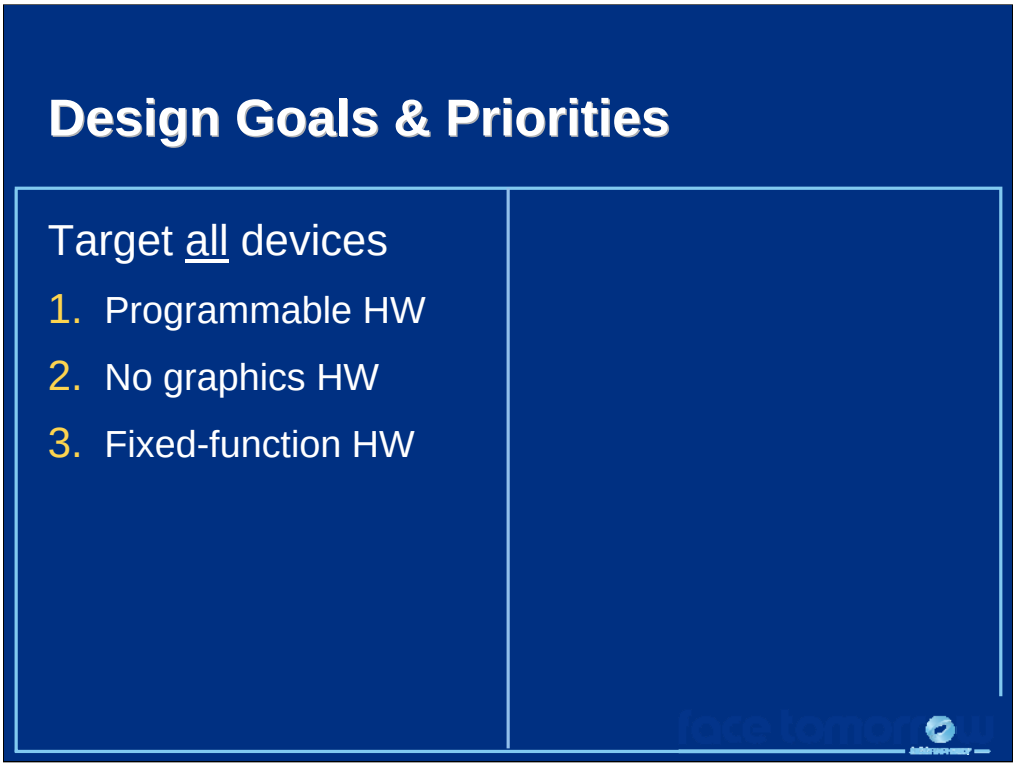
Design

Fixed functionality

Programmable shaders

New high-level features

Summary, Q&A

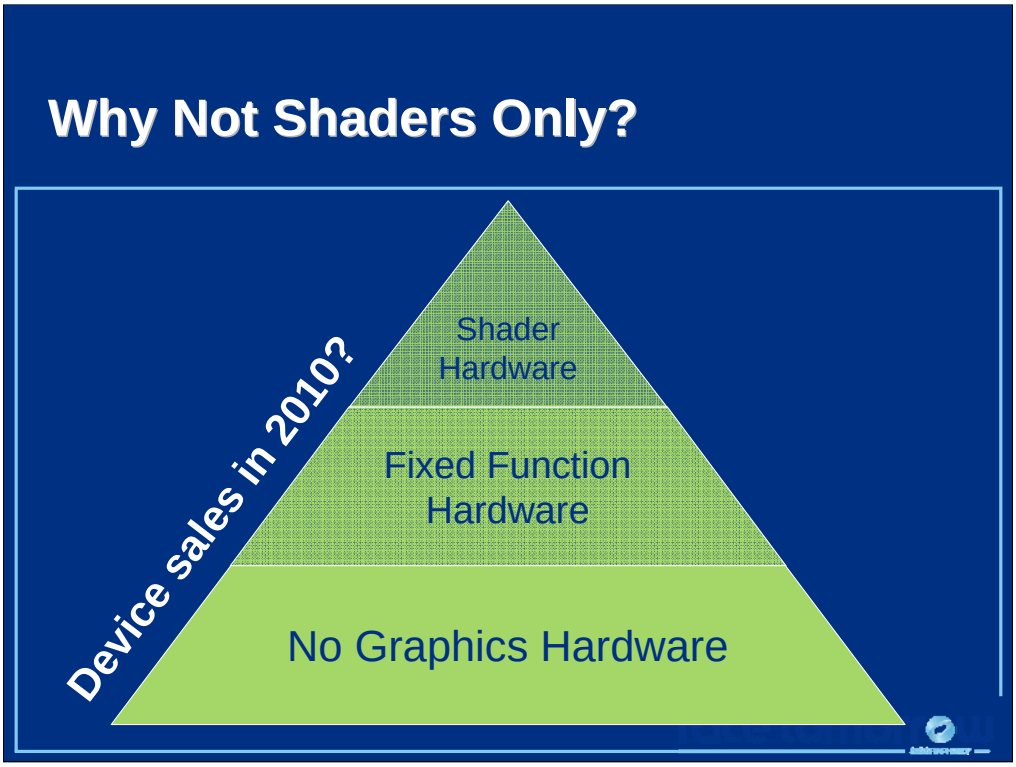


One of our main design goals is to support the whole range of devices that will be coming out in 2008 and later.

•Devices with programmable shaders are the first priority, because we really want that hardware to be accessible from Java.

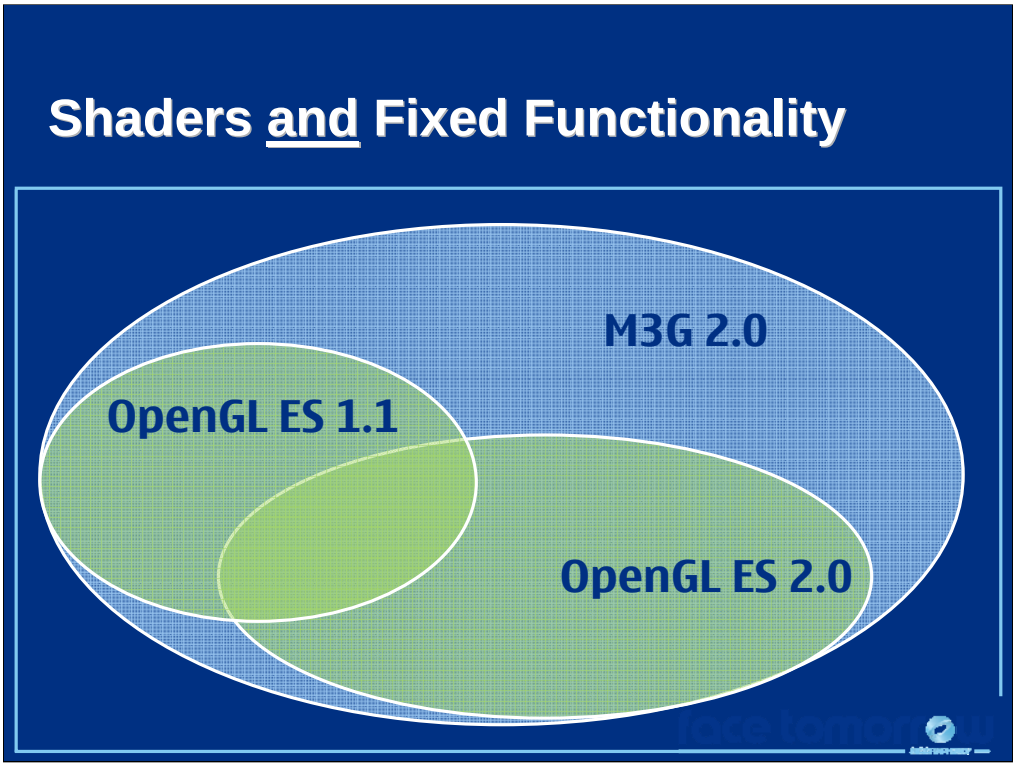
•Devices without any 3D acceleration are the second priority, because they are shipping in huge volumes.

•Finally, fixed-function hardware will ship in large volumes for years to come, and we want to leverage that hardware better than M3G 1.1 does.

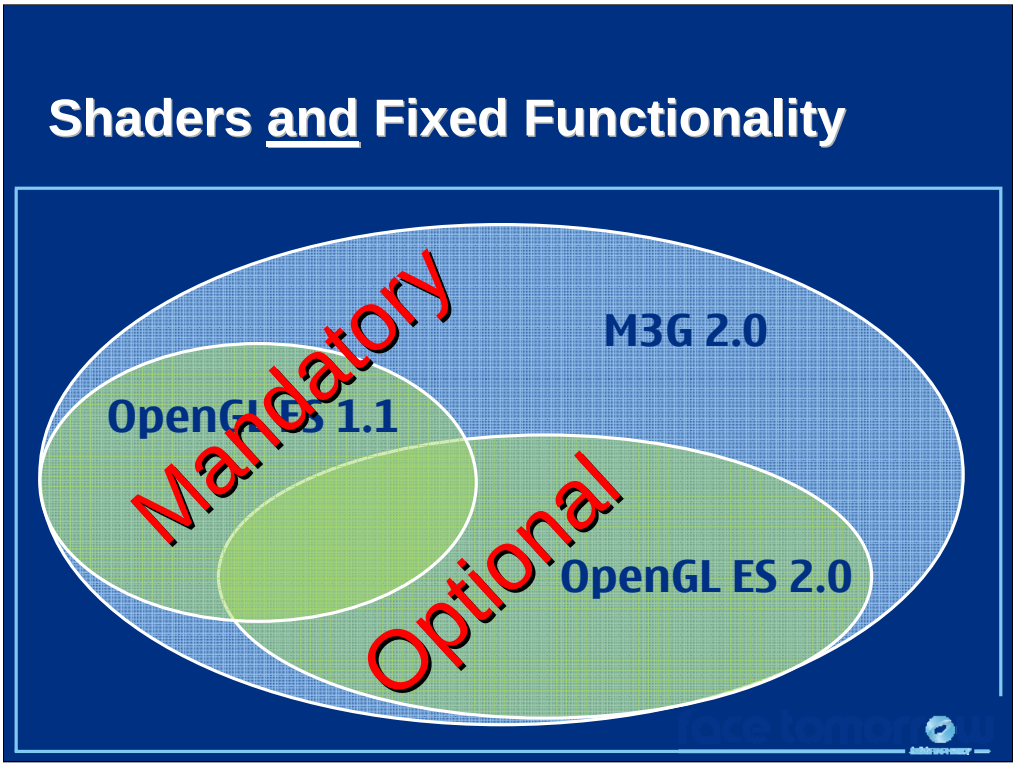


Programmable graphics hardware will be a “must have” for gaming devices, but only “nice to have” for the rest. In fact, this applies to graphics hardware in general.

We can’t afford to ignore the majority of devices that will only have fixed-function hardware or “software acceleration”.

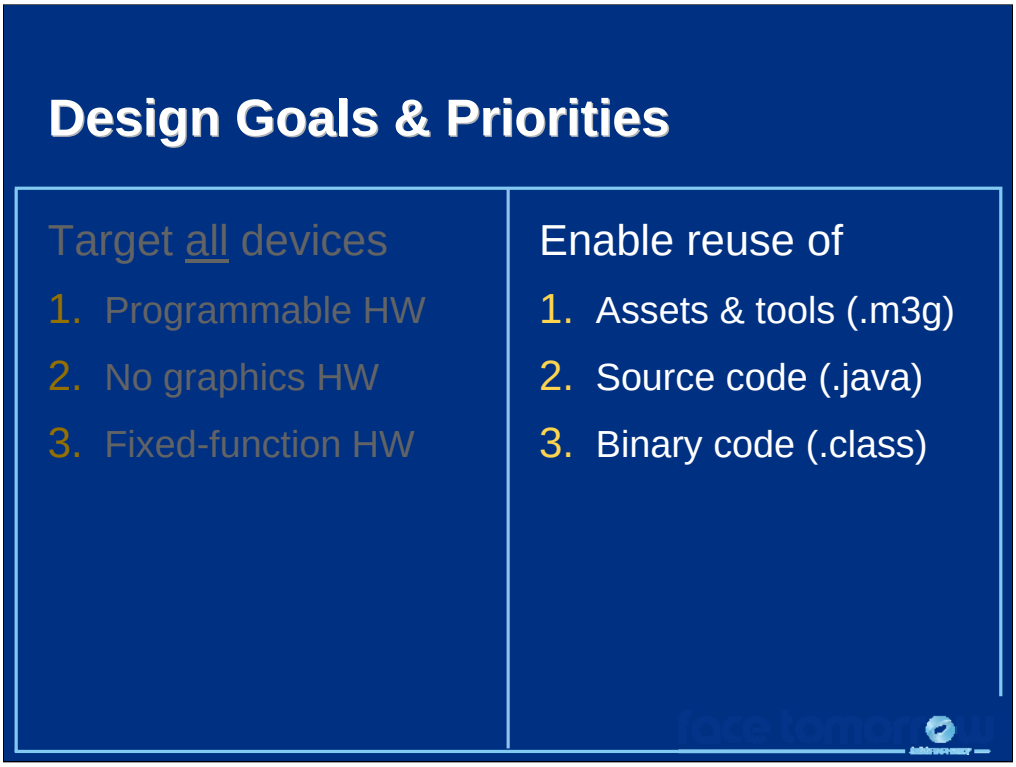


Some of our target devices will run on OpenGL ES 1.1, others on 2.0, so we have to accommodate both.



Of course, it's impossible to run shaders on fixed-function hardware, and impractical on the CPU, so we have to leave them optional. The fixed-function part is **not** optional, though.

High-end devices will therefore implement the whole of M3G 2.0, including both shaders and fixed functionality. Lower-end devices will omit shaders and a few other things.

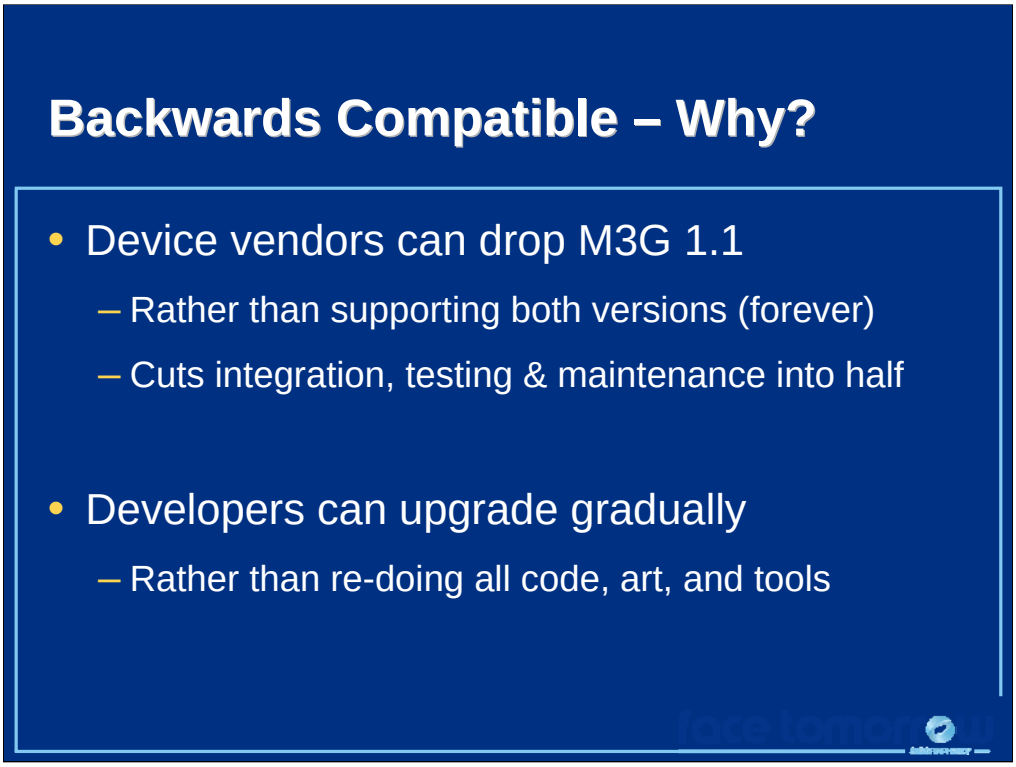


Besides targeting all devices, we also want to keep the standard backwards compatible.

•Being able to reuse art assets and tools is the most important thing in this respect.

•The ability to reuse existing source code is nice to have, but not an absolute requirement.

•Binary compatibility is less important, because applications are generally rebuilt for every device that comes out.



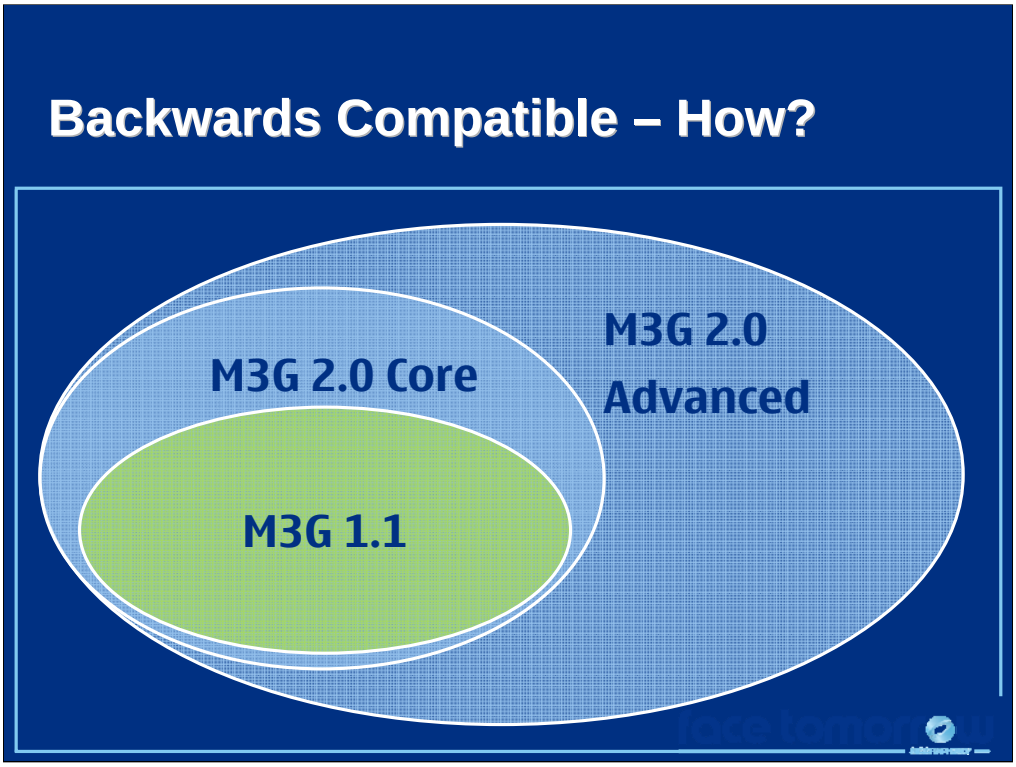
Why is it so important to keep the API backwards compatible?

First of all, it allows device vendors to drop M3G 1.1 immediately. Otherwise, the old version would have to be dragged along for an eternity. The old engine would take up extra ROM and RAM, but more importantly, it would be a maintenance burden. Rather than integrating and testing one 3D engine with every new piece of software or hardware, you'd have to do it for two separate engines!

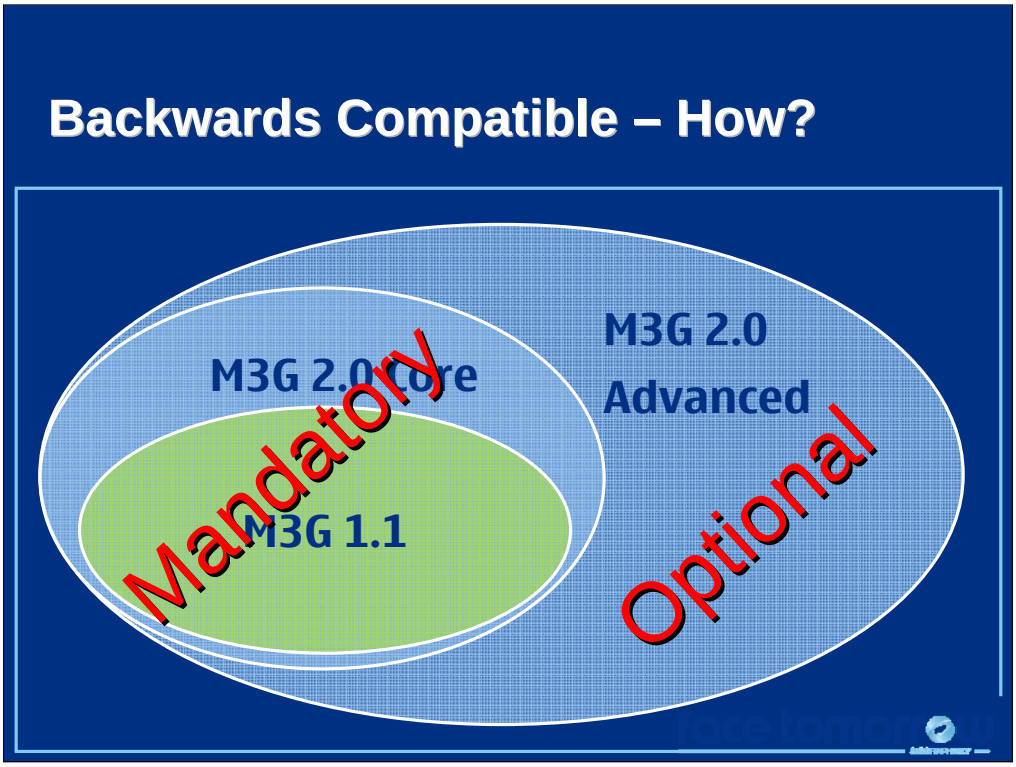
Secondly, it allows developers to upgrade to 2.0 at their own pace. For instance, they can dress up an existing M3G 1.1 title for 2.0 handsets by adding a few shader effects, rather than doing a completely new version. Even if they do decide to write a new version from scratch, they can still benefit from existing assets and tools.

Finally,

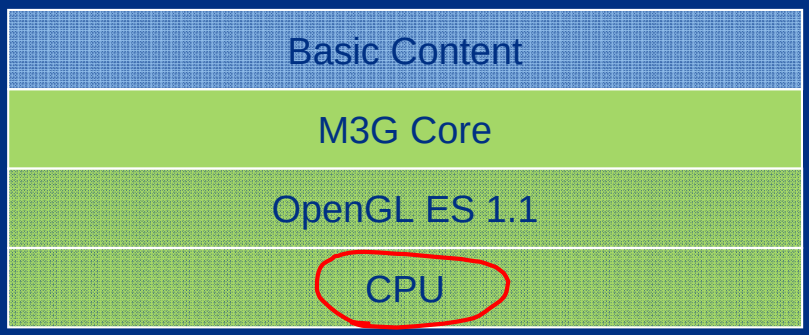
Backwards Compatible – How?



Backwards Compatible – How?

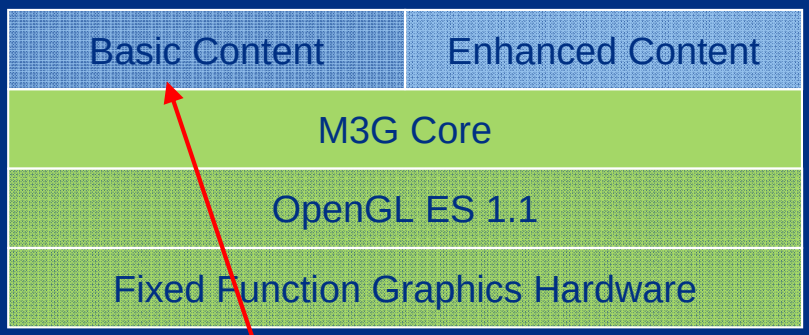


Serving the Low End…



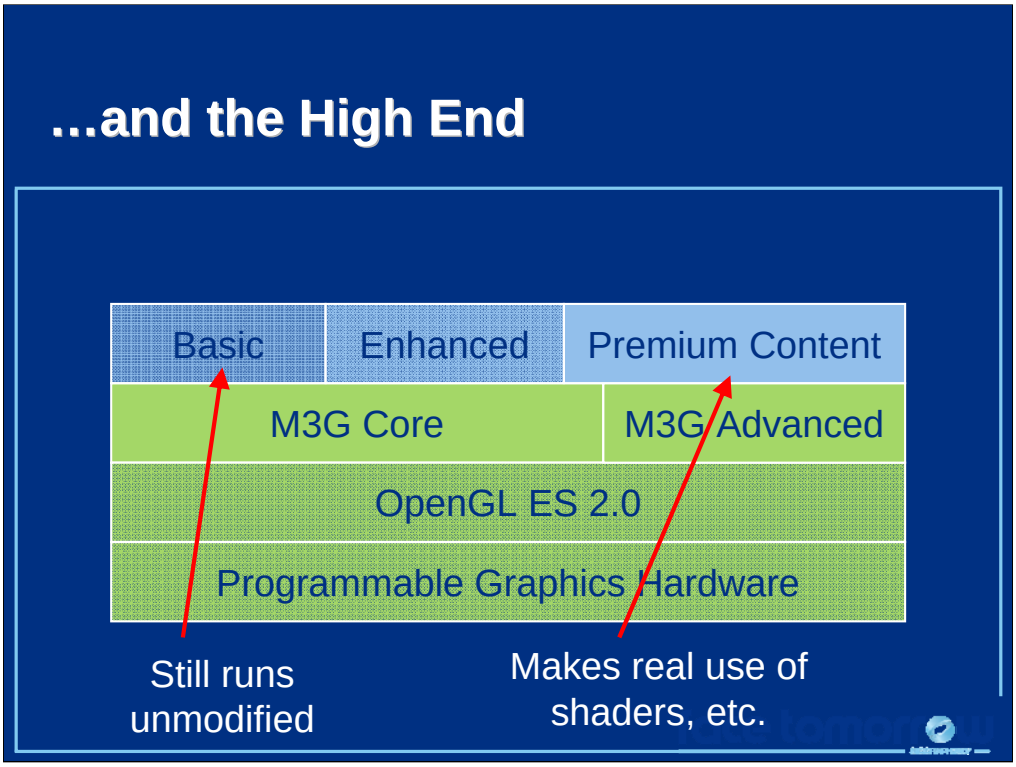
No graphics hardware!

…the Mid Category…



Runs unmodified on
mid-range devices

…and the High End



The Downsides

- Must support fixed functionality on ES 2.0
  - Extra implementation burden
- The API is not as compact as it used to be
  - A pure shader API could have ~20% fewer classes
- Need to drag along obsolete features
  - Flat shading, Sprite3D, Background image
  - Can be deprecated, but not totally removed

This “all-in-one” approach has some drawbacks, too.

•Implementing the fixed-function pipe with shaders is non-trivial. For instance, you will likely have to generate new shaders at run time.

•The API gets bigger and more complicated than if it were fixed-function-only or shader-only.

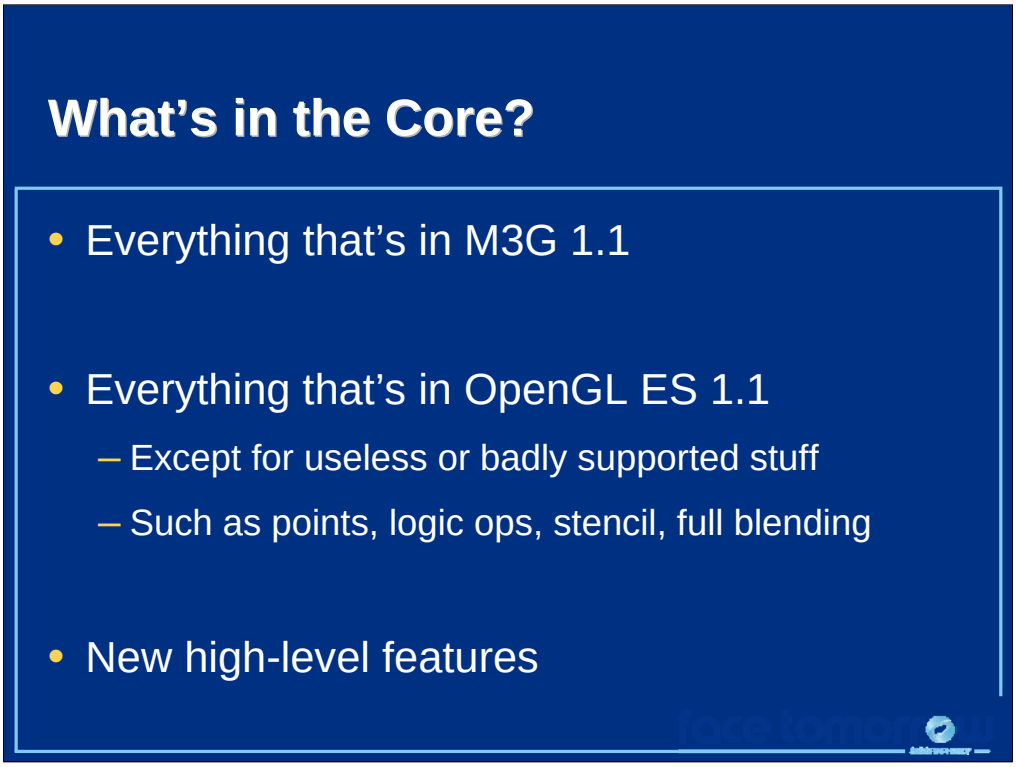
•Backwards compatibility requires that we drag along obsolete features.

Core vs. Advanced

- High-level features are common to both
  - Scene graph
  - Animation
- The differences are in rendering
  - Core ➔ OpenGL ES 1.1
  - Advanced ➔ OpenGL ES 2.0

Packages

- `javax.microedition.m3g`
  - Contains the entire Core Block
  - Also some Advanced features, e.g. cube maps
- `javax.microedition.m3g.shader`
  - Only present in Advanced implementations



With the introduction of point sprites, ordinary points have become fairly useless. They no longer exist in OpenGL ES 2.0, so they were not included in M3G. The same applies for logic ops, except that they were never really useful in the first place.

Stencil buffering would be very useful, but is not well supported by existing fixed-function hardware, so it was deferred to the Advanced Block.

There are also known issues with some source/destination blending modes on certain pieces of existing hardware. Also, the blending modes in OpenGL ES 1.1 have annoying limitations, such as the lack of separate blending functions for color and alpha. As a result, the M3G Core Block only contains a small set of predefined modes that are guaranteed to work everywhere, while the Advanced Block includes the full set of ES 2.0 blending modes (without the ugly restrictions).

What's in the Advanced Block?

- Everything that's in OpenGL ES 2.0
  - Vertex and fragment shaders
  - Cube maps, advanced blending
  - Stencil buffering

Stencil buffering and the full set of frame buffer blending modes are only supported in the Advanced Block.

M3G 2.0 Preview

Design

Fixed functionality

Programmable shaders

New high-level features

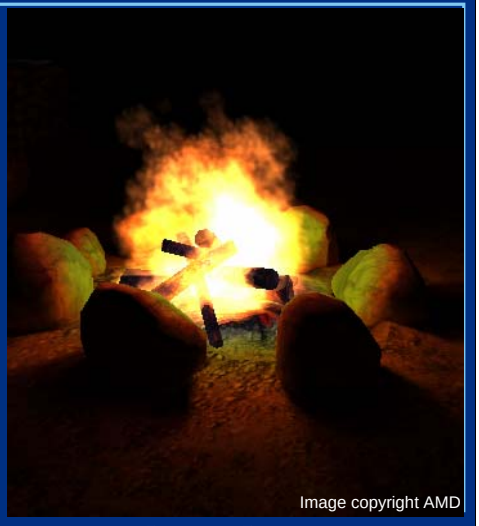
Summary, Q&A

M3G 2.0 Core vs. 1.1

- Better and faster rendering
- More convenient to use
- Fewer optional features

Point Sprites

- Ideal for particle effects
- Much faster than quads
- Consume less memory
- Easier to set up

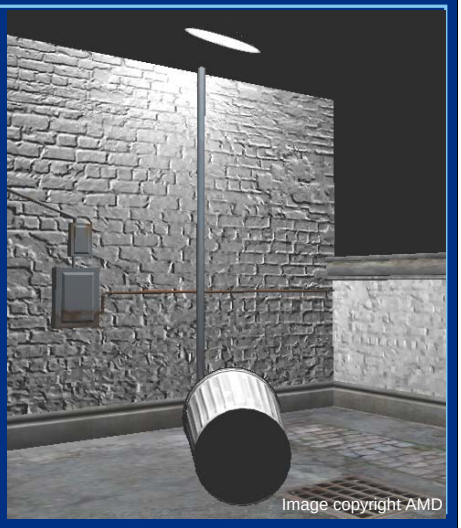


Better Quality Texturing

- Upgraded the baseline
  - At least two texture units
  - At least 1024x1024 maximum size
- Mandated optional features
  - Perspective correction
  - Mipmapping
  - Bilinear filtering

Bump Mapping

- Fake geometric detail
- Feasible even w/o HW

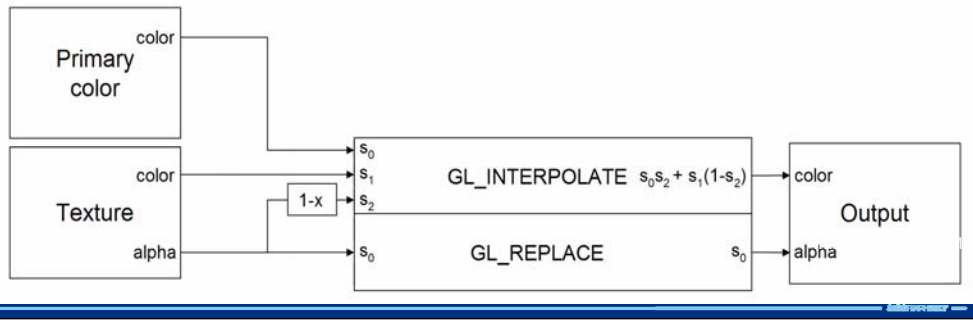


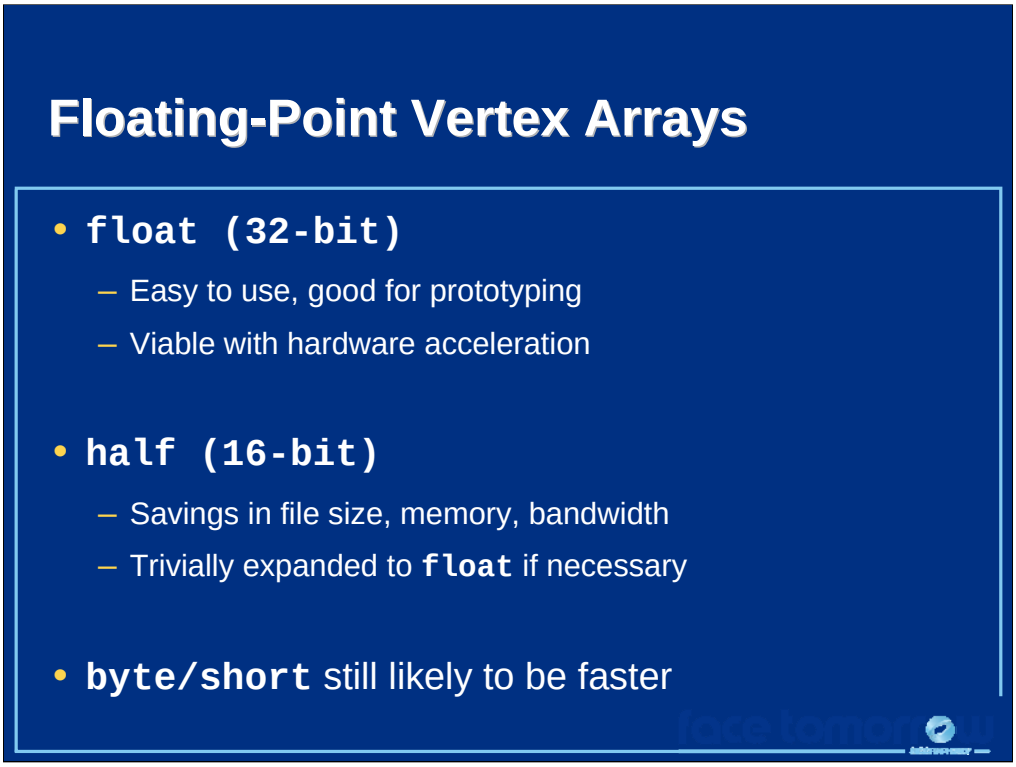
Bump Mapping + Light Mapping

- Bump map modulated by projective light map

Texture Combiners

- Precursor to fragment shaders
  - Can do a lot more than bump and light mapping
  - Not very easy to use, though





Half-float (FP16) vertex arrays are also useful at the API level even if there's no hardware support:

•They reduce file size compared to 32-bit float arrays.

•They can be easily supported in software implementations.

•When supported, they are faster and need less memory.

•They can be trivially expanded to float if necessary.

Triangle Lists

- Much easier to set up than strips
  - Good for procedural mesh generation
  - Avoid the expensive stripification
- No performance penalty
  - Can be even faster with good vertex ordering
  - Assuming a vertex cache

Primitives – M3G 1.x

	Byte	Short	Implicit	Strip	Fan	List
Triangles	✓	✓	✓	✓	✘	✘
Lines	✘	✘	✘	✘	✘	✘
Points	✘	✘	✘			✘
Point sprites	✘	✘	✘			✘

Relative to OpenGL ES 1.1

Primitives – M3G 2.0

	Byte	Short	Implicit	Strip	Fan	List
Triangles	✓	✓	✓	✓	✗	✓
Lines	✓	✓	✓	✓	✗	✓
Points	✗	✗	✗			✗
Point sprites	✓	✓	✓			✓

Relative to OpenGL ES 1.1

Points, triangle fans, and line loops were not added, as their use cases are very limited.

Also, points are not supported by OpenGL ES 2.0.

VertexBuffer Types – M3G 1.x

	Byte	Short	Fixed	Float	2D	3D	4D
Vertices	✓	✓	✗	✗	✗	✓	✗
TexCoords	✓	✓	✗	✗	✓	✓	✗
Normals	✓	✓	✗	✗		✓	
Colors	✓		✗	✗		✓*	✓
PointSizes			✗	✗			

* OpenGL ES 1.1 only supports RGBA colors

VertexBuffer Types – M3G 2.0

	Byte	Short	Fixed	Float Half	2D	3D	4D
Vertices	✓	✓	✓	✓	✓	✓	✓
TexCoords	✓	✓	✓	✓	✓	✓	✓
Normals	✓	✓	✓	✓		✓	
Colors	✓		✓	✓		✓*	✓
PointSizes			✓	✓			

* OpenGL ES 1.1 only supports RGBA colors

Deprecated Features

- Background image
  - Use a sky box instead
- Sprite3D
  - Use textured quads or point sprites instead
- Flat shading
  - Can’t have this on OpenGL ES 2.0!

Deprecated Features Cont'd

- Two-sided lighting
  - Requires duplicated geometry on OpenGL ES 2.0
- Local camera lighting (a.k.a. local viewer)
  - Only a hint that was poorly supported
- Less accurate picking
  - Skinning and morphing not taken into account

M3G 2.0 Preview

Design

Fixed functionality

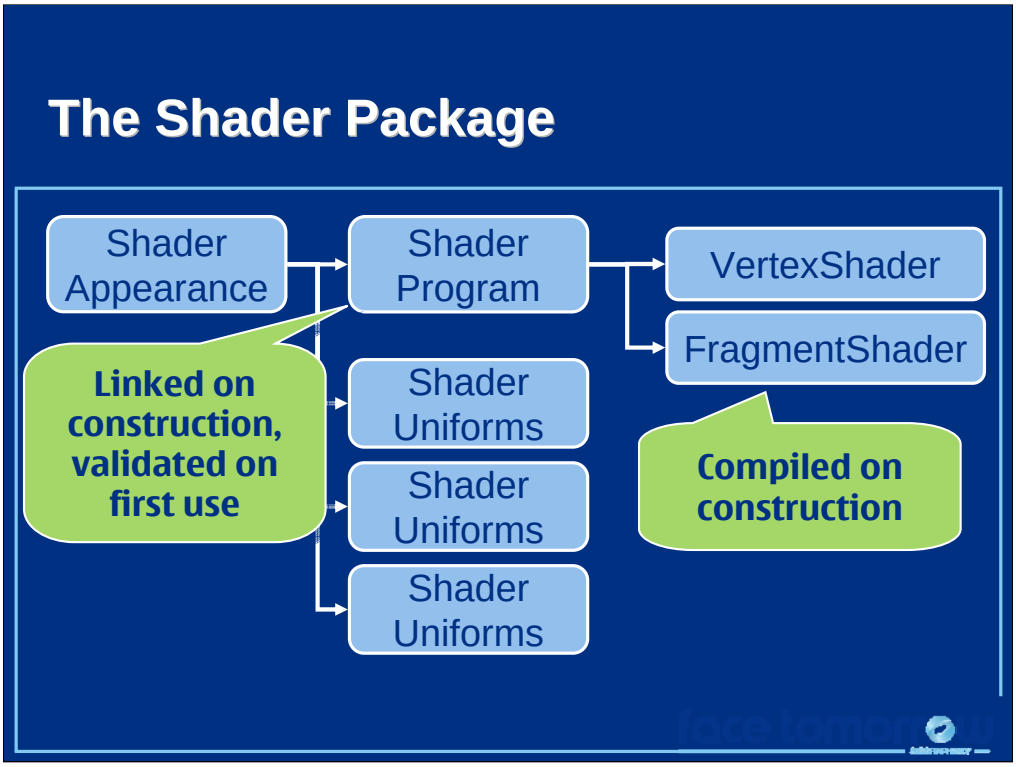
Programmable shaders

New high-level features

Summary, Q&A

Shading Language

- GLSL ES v1.00
  - Source code only
  - Binary shaders would break the Java sandbox
- Added a few preprocessor `#pragma`'s
  - To enable skinning, morphing, etc.
  - Apply for vertex shaders only



ShaderAppearance includes a ShaderProgram and a set of ShaderUniforms.

VertexShader and FragmentShader are compiled on construction. ShaderPrograms are linked on construction.

Each ShaderUniform object can contain an arbitrary number of shader variables. The shader variables can be user-defined or bound to a scene graph property (such as a node transformation, a camera projection matrix, or a light intensity).

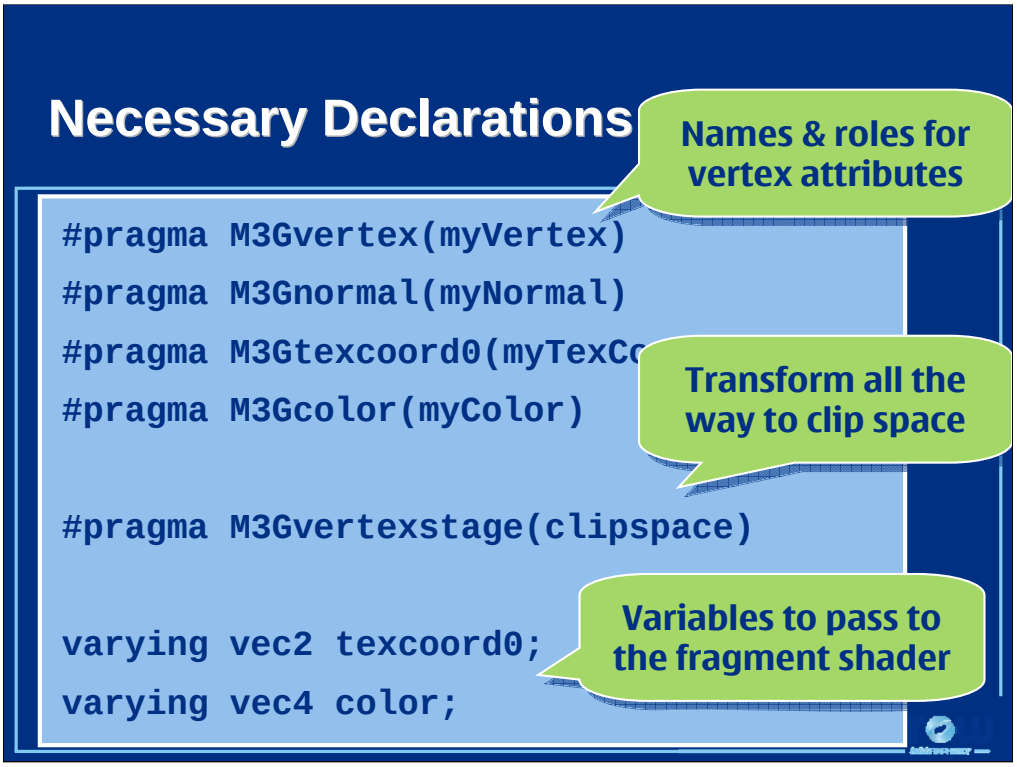
Why Multiple ShaderUniforms?

- So that uniforms can be grouped
  - Global constants – e.g. look-up tables
  - Per-mesh constants – e.g. rustiness
  - Per-frame constants – e.g. time of day
  - Dynamic variables – e.g. position, orientation
- Potential benefits of grouping
  - Java object reuse – less memory, less garbage
  - Can be faster to bind a group of variables to GL

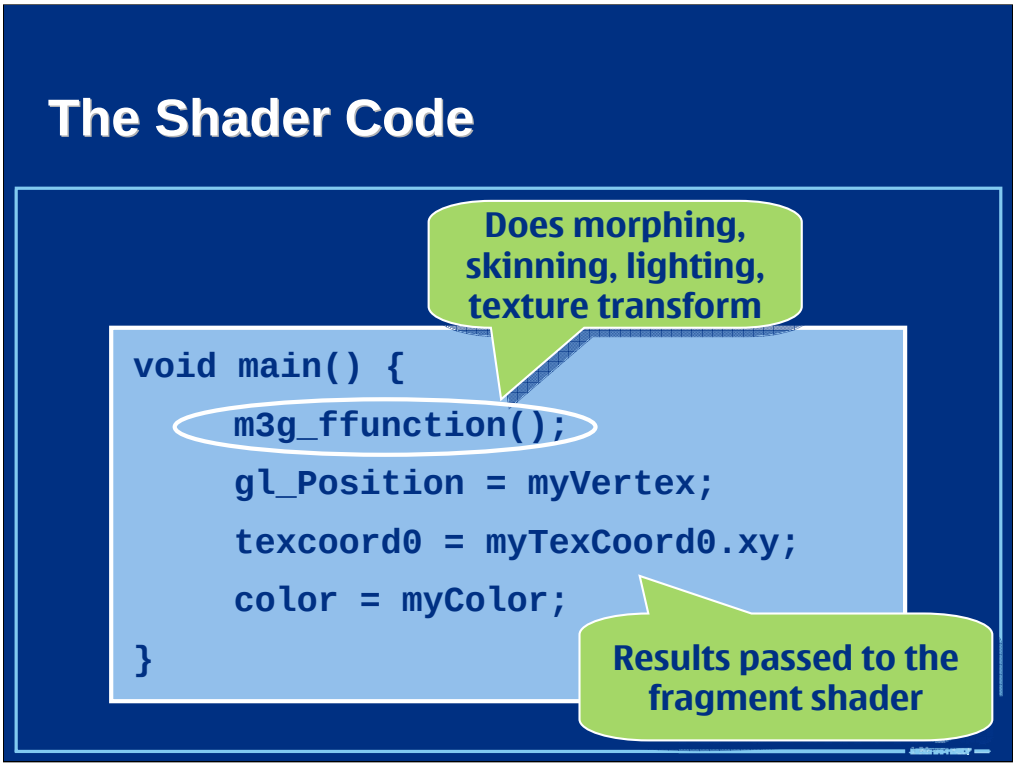
A Fixed-Function Vertex Shader

- A small example shader
- Replicates the fixed-function pipeline using the predefined `#pragma`'s

Necessary Declarations



The Shader Code



M3G 2.0 Preview

Design

Fixed functionality

Programmable shaders

New high-level features

Summary, Q&A

Scene Graph

- Added automatic render-to-texture
- Otherwise mostly unchanged
- Some convenience methods
  - Can use quaternions instead of axis/angle
  - Can enable/disable animations hierarchically

File Format

- Updated to match the new API
  - File structure remains the same
  - Same parser can handle both old & new
- Better compression for
  - Textures (ETC, JPEG)
  - SkinnedMesh, IndexBuffer

Multichannel Keyframe Sequences

- *N* channels per KeyframeSequence object
  - Same number of keyframes in all channels
  - Shared interpolation mode
  - Shared time stamps
- Huge memory savings with skinning
  - M3G 1.1: two Java objects per bone
  - M3G 2.0: two Java objects per mesh

Things Under Consideration

- Bounding volumes (provided by user)
- Texture compression (run-time encoding)
- Combined morphing and skinning

- Less likely to be included
  - Texture generation
  - Collision detection
  - Particle systems
  - Mesh modifiers

M3G 2.0 Preview

Design

Fixed functionality

Programmable shaders

New high-level features

Summary, Q&A

Summary

- M3G 2.0 will replace 1.1, starting next year
  - Existing code & assets will continue to work
  - Developers can upgrade at their own pace
- Several key improvements
  - Expanded fixed-function feature set
  - Programmable shaders to the mass market
  - Better performance across all device categories

Q&A

Thanks:

M3G 2.0 Expert Group

Dan Ginsburg (AMD)

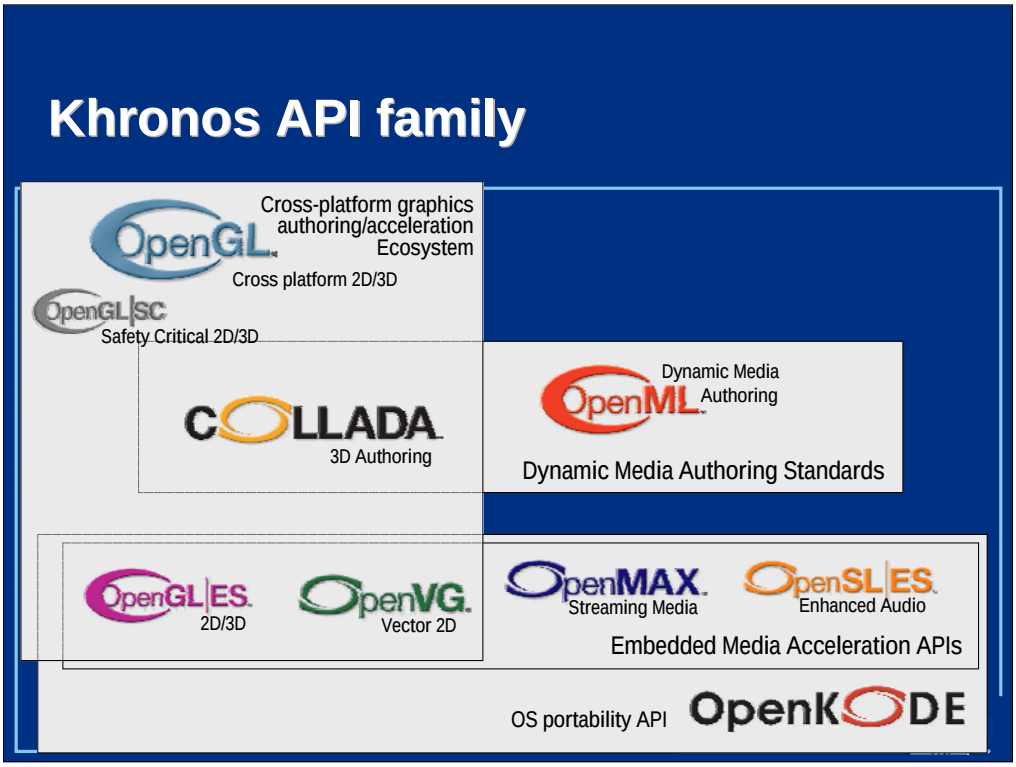
Kimmo Roimela (Nokia)


SIGGRAPH2007

Closing & Summary

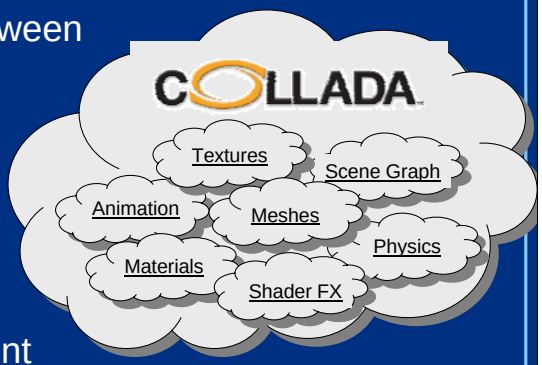
- We have covered
  - OpenGL ES
  - M3G

Khronos API family



COLLADA

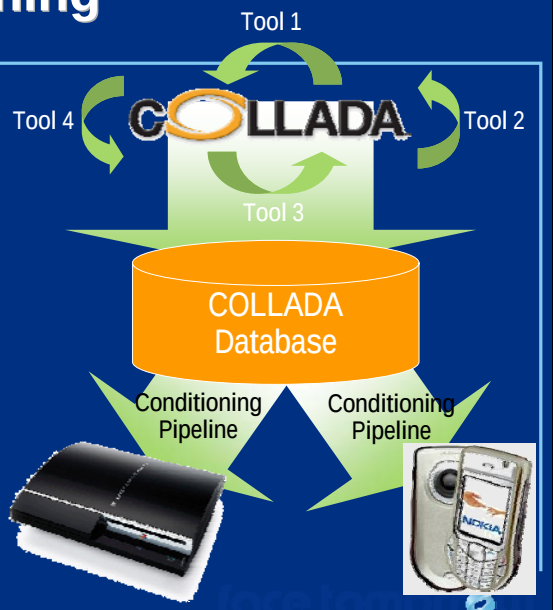
- An open interchange format
  - to exchange data between content tools
  - allows mixing and matching tools for the same project
  - allows using desktop tools for mobile content



Collada conditioning

- Conditioning pipelines take authored assets and:
- 1. Strips out authoring-only information
- 2. Re-sizes to suit the target platform
- 3. Compresses and formats binary data for the target platform

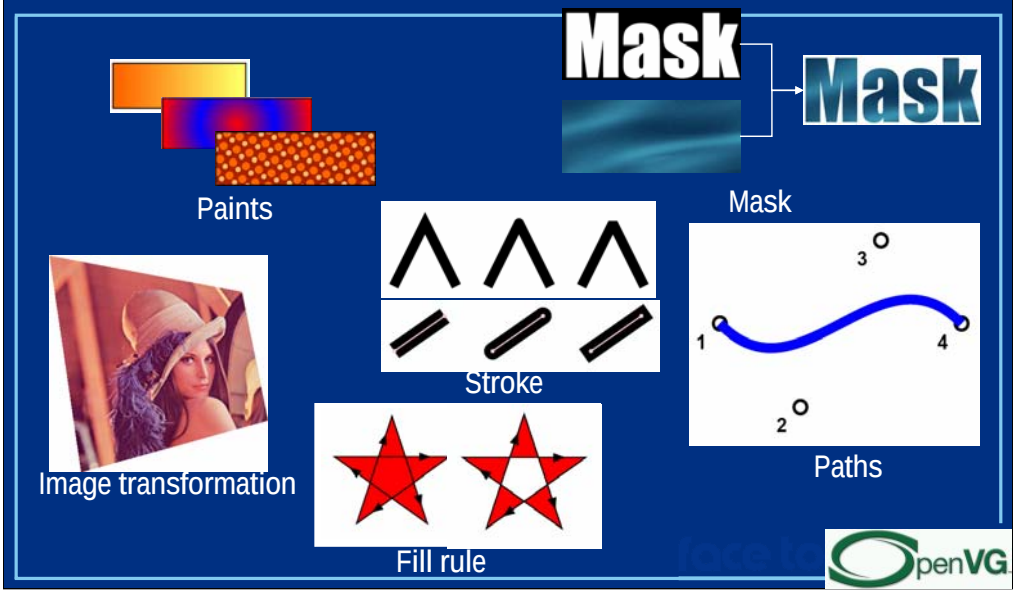
- Different target platforms can use the same asset database with the appropriate conditioning pipeline



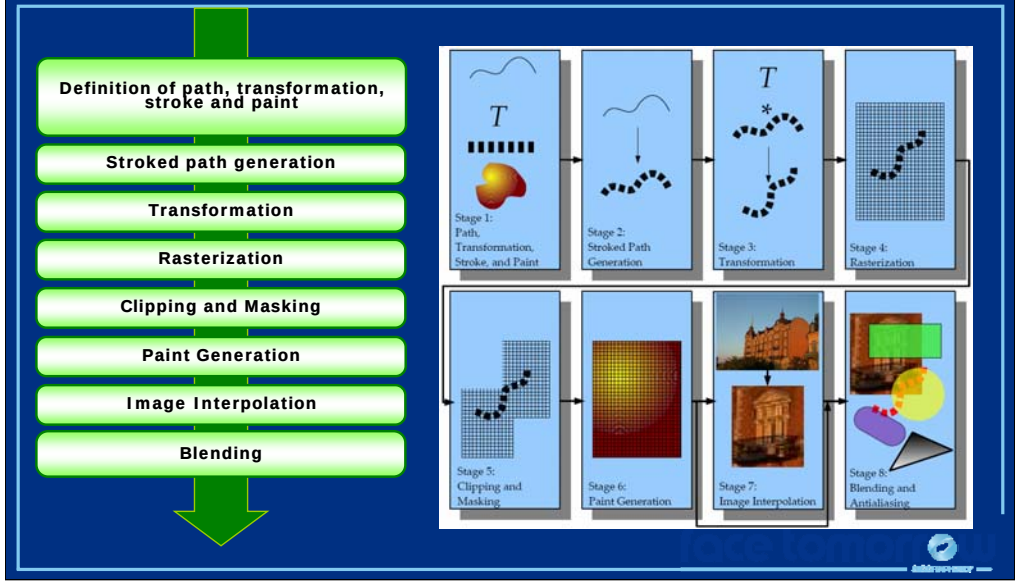
2D Vector Graphics

- OpenVG
  - low-level API, HW acceleration
  - spec draft at SIGGRAPH 05, conformance tests summer 06
- JSR 226: 2D vector graphics for Java
  - SVG-Tiny compatible features
  - completed Mar 05
- JSR 287: 2D vector graphics for Java 2.0
  - rich media (audio, video) support, streaming
  - may still complete in 07

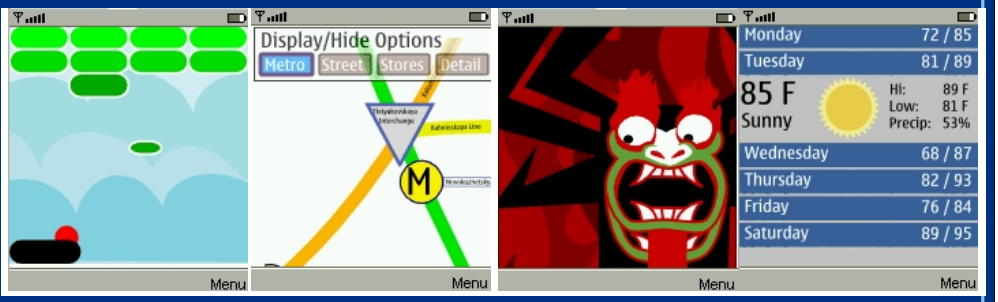
OpenVG features



OpenVG pipeline



JSR-226 examples



Game, with skins

Scalable maps, variable detail

Cartoon

Weather info

Combining various APIs

- It's not trivial to efficiently combine use of various multimedia APIs in a single application
- EGL is evolving towards simultaneous support of several APIs
  - OpenGL ES and OpenVG now
  - all Khronos APIs later

OpenGL ES and OpenVG

OpenGL ES
Accurately represents
PERSPECTIVE and
LIGHTING

OpenVG
Accurately represents
SHAPE and
COLOR

OpenVG ideal for advanced composting user interfaces
OpenGL ES for powerful 3D UI effects

Summary

- Fixed functionality mobile 3D is reality NOW
  - these APIs and devices are out there
  - go get them, start developing!
- Better content with Collada
- Solid roadmap to programmable 3D
- New standards for 2D vector graphics


SIGGRAPH2007