Developing Mobile 3D Applications with OpenGL ES and M3G

Kari Pulli	Nokia Research Center
Jani Vaarala	Nokia
Ville Miettinen	Hybrid Graphics
Tomi Aarnio	Nokia Research Center
Mark Callow	HI Corporation

Today's program

- Start at ?:??
- Intro & OpenGL ES overview
  25 min, Kari Pulli
- Using OpenGL ES
  40 min, Jani Vaarala
- OpenGL ES performance
  25 min, Ville Miettinen
- Break ?:?? – ?:??
- M3G API overview
  45 min, Tomi Aarnio
- Using M3G
  40 min, Mark Callow
- Closing & Q&A
  5 min, Kari Pulli

Challenges for mobile gfx

- Small displays
  - getting much better
- Computation
  - speed
  - power / batteries
  - thermal barrier
- Memory

Fairly recently mobile phones used to be extremely resource limited, especially when it comes to 3D graphics. But Moore's law is a wonderful thing.

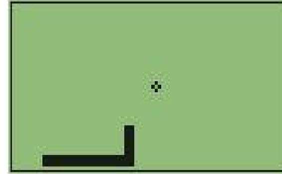
The displays used to be only 1-bit black-and-white displays, that update slowly, with resolutions like 48 x 84 pixels. However, the display technology has developed by leaps and bounds, first driven by the digital cameras, now by mobile phones. Only 12-bit colors are beginning to be old-fashioned, 16 or 18 bit color depths are becoming the norm, 24 bit can't be too far ahead. The main resolution for Nokia's S60 used to be 176 x 208 (upper right), now it's getting to 240 x 320 and 352 x 416, Nokia Communicator (middle) is 640 x 200, Nokia 770 is 800 x 400 (bottom).

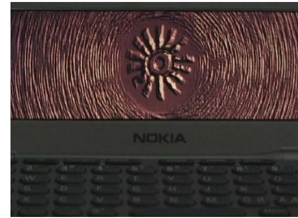
CPUs used to be tiny 10+ MHz ARM 7's, now 100-200 MHz ARM 9's are norm, pretty soon it'll be 400-600 MHz ARM 11's. It is still very rare to find hardware floating point units even in higher end PDAs, but eventually that will also be available. But the biggest problem is power. All those megahertzes and increased pixel resolutions eat power, and the battery technology does not increase as fast as other components. So the amount of power in batteries compact enough to be pocketable is a limiting factor. But even if we suddenly had some superbatteries, we couldn't use all that power. More and more functionality on smaller physical size means that designing hardware so it doesn't generate hotspots that fry the electronics becomes increasingly challenging.

And memory is always a problem. Current graphics cards have 128, 256, and even more megabytes of memory, just for graphics, frame buffers, textures caches, and the like. Mobile devices have to deal with MBs that you can count with your fingers and toes, and that must be enough for the ROM / "hard drive"

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

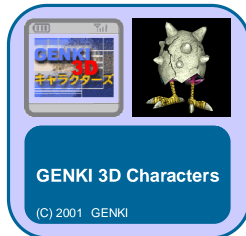


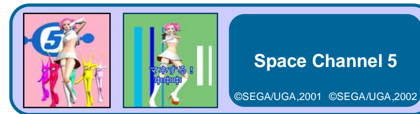


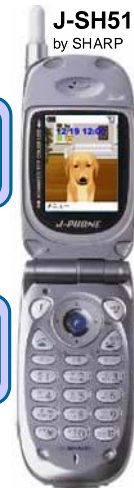
Around 2001, at least in Europe and Americas, the state of the art for mobile graphics was games such as Snake. Considering that in 2001 alone Nokia shipped over 100 million phones, most with Snake, with very few other games available, Snake is at least one of the most played electronic games ever.

In 2001 an old Amiga demo was ported to Nokia communicator, causing a sensation at the Assembly event in Finland.

State-of-the-art in 2001: Japan







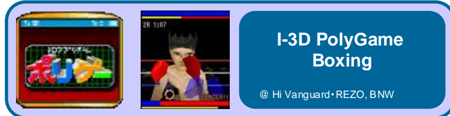
- High-level API with skinning, flat shading / texturing, orthographic view

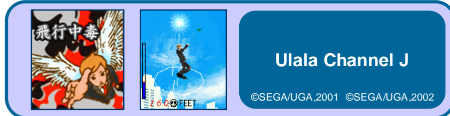
State-of-the-art in 2002: GSM world

- 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

State-of-the-art in 2002: Japan

- Gouraud shading, semi-transparency, environment maps





3d menu

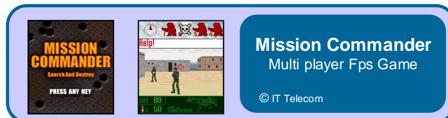
State-of-the-art in 2003: GSM world

- 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

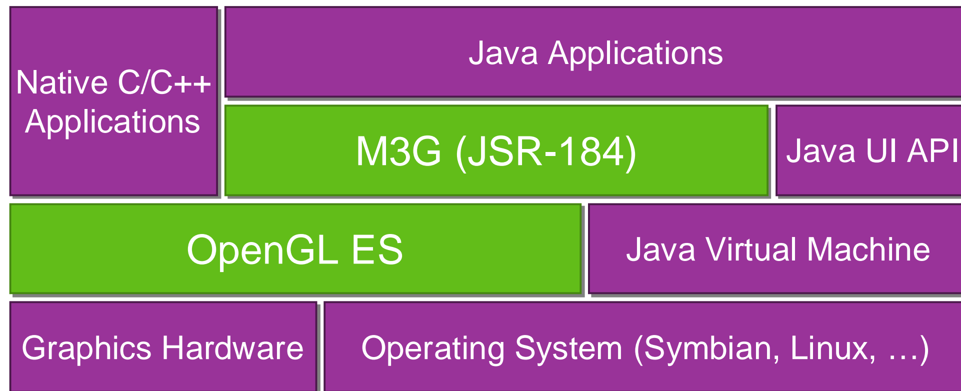


Mobile 3D in 2004

- 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 3D APIs



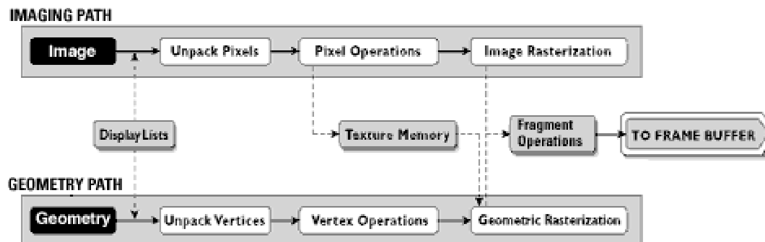
The green parts show the content of today's course. We will cover two mobile 3D APIs, used by applications, either the so-called native C/C++ applications, or Java midlets (the mobile versions of applets). The APIs use system resources such as memory, display, and graphics hardware if available. OpenGL 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).

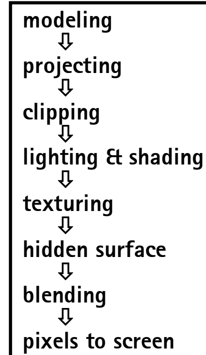
Overview: OpenGL ES

- 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





- 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

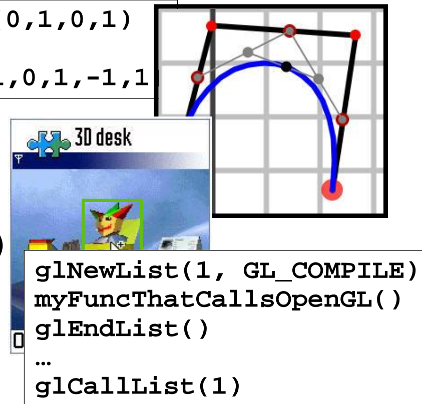
Outline

- 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?

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_TRIANGLES, 0, 4 );
```

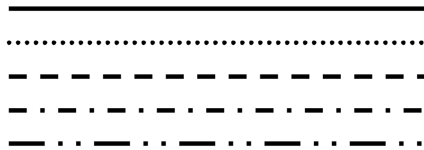
OpenGL|ES

Functionality: in / out? (3/7)

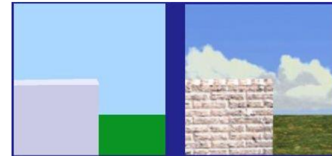
- Simplify rendering modes
  - double buffering, RGBA, no front buffer access
- Emulating back-end missing functionality is expensive or impossible
  - full fragment processing is **IN**
    alpha / depth / scissor / stencil tests,
    multisampling,
    dithering, blending, logic ops)

Functionality: in / out? (4/7)

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

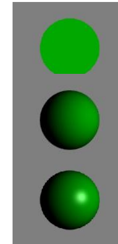


Functionality: in / out? (5/7)

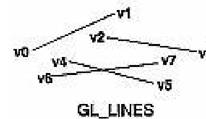


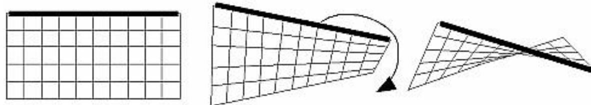
- 2D texture maps **IN**
  - 1D, 3D, cube maps **OUT**
  - borders, proxies, priorities, LOD clamps **OUT**
  - multitexturing, texture compression **IN** (optional)
  - texture filtering (incl. mipmaps) **IN**
  - new: paletted textures **IN**

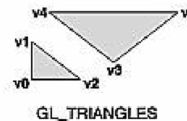
Functionality: in / out? (6/7)



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







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

The great "Floats vs. fixed-point" debate

- 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

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
    - lately even across APIs (OpenGL ES, OpenVG)

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
  - PM event => all Context lost
  - Disp & 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

On-Device Implementations

- NokiaGL (SW)
- N93 (HW)
- Imagination MBX
- NVidia GoForce 3D
- ATI Imageon
- Toshiba T4G
- …

The models shown

Nokia 6630

Dell Axim 50v

Gizmondo

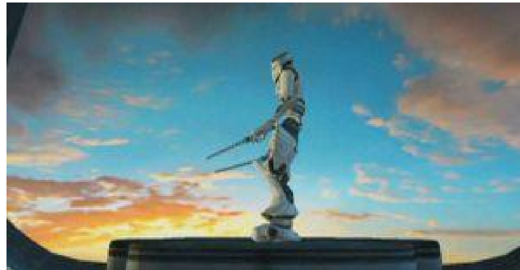
LG 3600

Sharp V602SH

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?

Using OpenGL ES

Jani Vaarala

Nokia

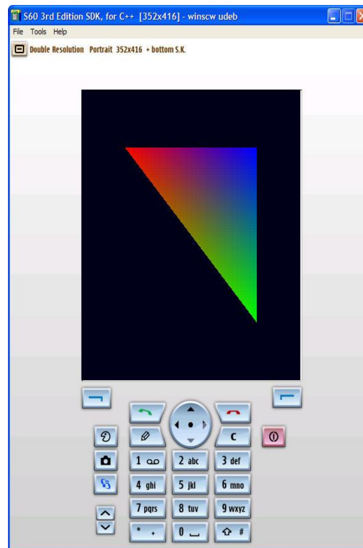
Using OpenGL ES

- Simple OpenGL ES example
- Fixed point programming
- Converting existing code

-We will use Symbian S60 as an example, as there are already openly programmable devices out there that come with preinstalled OpenGL ES support

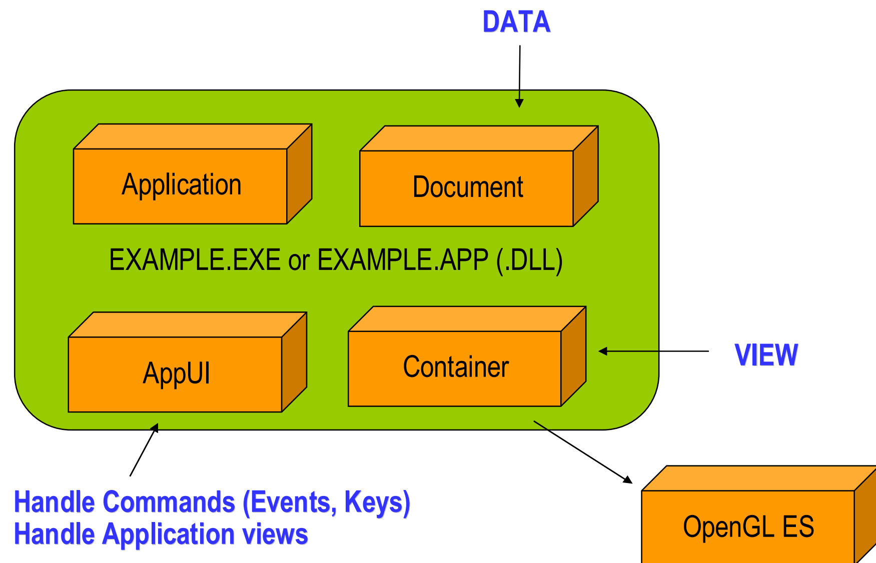
-Example code works with S60 3rd Edition SDK and devices (like N93)

"Hello OpenGL ES"



-This is what we are aiming for: single smooth shaded triangle on the emulator (and on the device).

Symbian App Classes



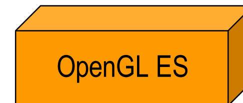
-Symbian UI framework follows Model-View-Controller model.

"Hello OpenGL ES"

```
/* ===============================================================
 *   "Hello OpenGL ES" OpenGL ES code.
 *
 *   Eurographics 2006 course on mobile graphics.
 *
 *   Copyright: Jani Vaarala
 * ===============================================================
*/

#include <e32base.h>
#include "SigTriangleGL.h"

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

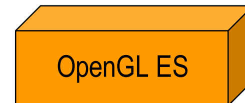


-First we define 3 vertices of a triangle.

-We use static const for two reasons: it's a good habit to mark it as const for compiler and under Symbian global data is not allowed.

“Hello OpenGL ES”

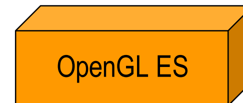
```
static const GLubyte colors[3 * 4] =
{
        255,   0,     0,     255,
        0,     255,   0,     255,
        0,     0,     255,   255
};
```



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

"Hello OpenGL ES"

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



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

"Hello OpenGL ES"

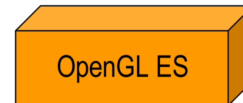
```
TInt CSigTriangleGL::DrawCallback( TAny* aInstance )
{
   CSigTriangleGL* instance = (CSigTriangleGL*) aInstance;

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

   eglSwapBuffers    (instance->iEglDisplay,instance->iEglSurface);

   /* To keep the background light on */
   if (!(instance->iFrame%100))        User::ResetInactivityTime();

   instance->iFrame++;
   return 0;
}
```



- This is the render callback. We just clear the color buffer, translate camera a bit and draw a triangle.
- Code keeps a running frame counter. Every once in a while call is made to User::ResetInactivityTime( ) to reset the inactivity counters (to avoid dimming of display backlight).

"Hello OpenGL ES"

```
void CSigTriangleContainer::ConstructL(const TRect& /* aRect */)
{
   iGLInitialized = EFalse;

   CreateWindowL();
   SetExtentToWholeScreen();
   ActivateL();

   CSigTriangleGL* gl = new (ELeave) CSigTriangleGL( );
   gl->Construct(Window());

   iGLInitialized = ETrue;
}

CSigTriangleContainer::~CSigTriangleContainer()
{
}
```

-ConstructL( ) will be called by the app framework to initialize the View. iGLInitialized is used to block GL calls before actual initialization is done (window operations may cause calls to SizeChanged function).

-We set the extent to fill the whole screen and call the constructor for the GL part of the application. We give in to that constructor a Symbian window class (RWindow) that we get from the Window( ) function.

-After the constructor returns, GL is in initialized state.

“Hello OpenGL ES”

```
void CSigTriangleContainer::SizeChanged()
{
    if(iGLInitialized)
    {
        glViewport(0,0,Size().iWidth,Size().iHeight);
    }
}

void HandleResourceChange( TInt aType )
{
    if(aType == KEikDynamicLayoutSwitch)
    {
        // Screen resolution changed, make window fullscreen in a new resolution
        SetExtentToWholeScreen();
    }
}

TInt CSigTriangleContainer::CountComponentControls() const
{
    return 0;
}

CCoeControl* CSigTriangleContainer::ComponentControl(TInt /* aIndex */) const
{
    return NULL;
}
```

-SizeChanged( ) will get called when the application window changes size. If GL is not initialized we don’t change the viewport here (if context is not valid, calling GL functions may crash the application)

-HandleResourceChange needs to be implemented to support Layout switching in the scalable UI architecture. Resolution of the device may change on the fly for example when the display is rotated.

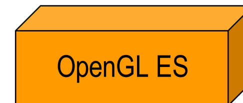
“Hello OpenGL ES”

```
/**************************************************************
 * Initialize OpenGL ES context and initial OpenGL ES state *
 **************************************************************/
void CSigTriangleGL::Construct(RWindow aWin)
{
   iWin = aWin;

   iEglDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
   if(iEglDisplay == NULL )    User::Exit(-1);

   if(eglInitialize(iEglDisplay,NULL,NULL) == EGL_FALSE)
                                User::Exit(-1);

   EGLConfig  config,colorDepth;
   EGLint     numOfConfigs = 0;
```



-This is our GL initialization code, called from the View.

-eglGetDisplay(EGL_DEFAULT_DISPLAY) – get the default display to render to

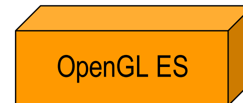
-eglInitialize( ) - initialize EGL on that display

"Hello OpenGL ES"

```
switch( iWin.DisplayMode() )
{
     case (EColor4K):   { colorDepth = 12; break; }
     case (EColor64K):  { colorDepth = 16; break; }
     case (EColor16M):  { colorDepth = 24; break; }
     default:
                          colorDepth = 32;
}

EGLint attrib_list[] = {    EGL_BUFFER_SIZE, colorDepth,
                            EGL_DEPTH_SIZE,  15,
                            EGL_NONE                    };

if(eglChooseConfig(iEglDisplay,attrib_list,&config,1,
     &numOfConfigs ) == EGL_FALSE) User::Exit(-1);
```



-iWin.DisplayMode( ) - find out the display mode of the window (match config with that)

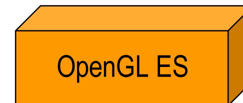
-eglChooseConfig( ) - choose the best matching config (see EGL spec for selection criteria)

“Hello OpenGL ES”

```
iEglSurface = eglCreateWindowSurface(iEglDisplay, config, &iWin, NULL );
if( iEglSurface == NULL )                          User::Exit(-1);

iEglContext = eglCreateContext(iEglDisplay,config, EGL_NO_CONTEXT, NULL );
if( iEglContext == NULL )                          User::Exit(-1);

if( eglMakeCurrent( iEglDisplay, iEglSurface, iEglSurface,
              iEglContext ) == EGL_FALSE )        User::Exit(-1);
```



-eglCreateWindowSurface( ) - create a window surface for rendering

-eglCreateContext( ) - create a rendering context (multiple contexts may be used, but not at the same time)

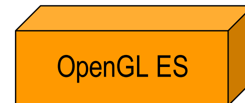
-eglMakeCurrent( ) - make surface current and context current to the display and the thread

"Hello OpenGL ES"

```
/* Create a periodic timer for display refresh */
iPeriodic = CPeriodic::NewL( CActive::EPriorityIdle );

iPeriodic->Start( 100, 100, TCallBack(
                         SigTriangleGL::DrawCallback, this ) );

initGLES();
```



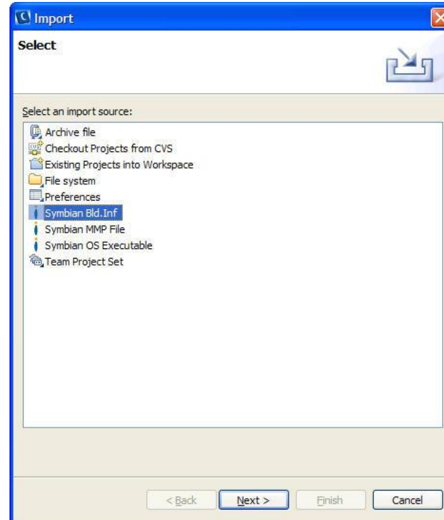
-Cperiodic::NewL( ) - create a Symbian Active Object (form of co-operative multi-tasking) for providing a timer callback

-initGLES( ) - call the GL initialization part shown before

Carbide C++ Express

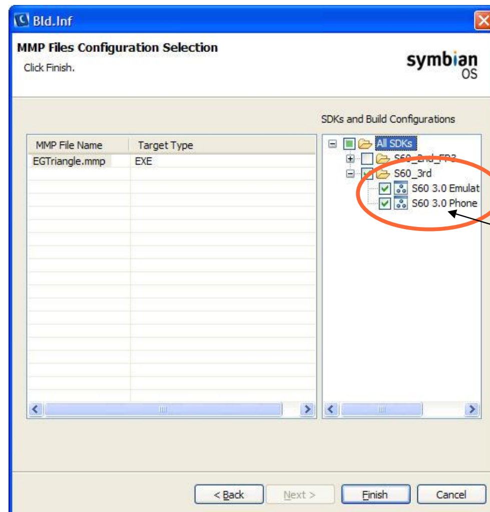
- Free IDE for S60 development from
  - http://www.forum.nokia.com
- Supports 2nd edition and 3rd edition SDKs
- Here we focus on 3rd edition
  - Future devices will be 3rd edition (e.g., N93)

Importing project



Importing project

Importing project

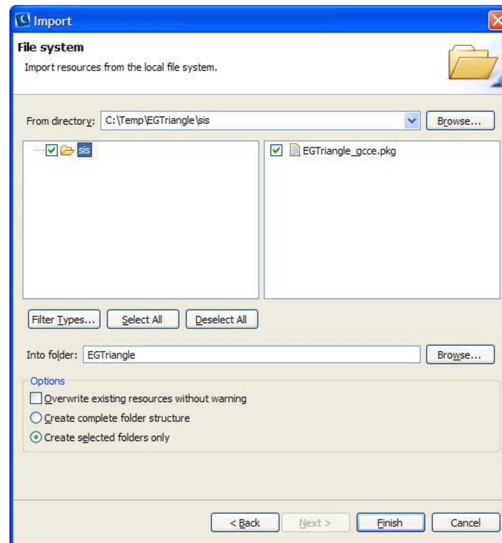


Select emulator configuration and phone configuration (GCCE) under S60_3rd.

Importing .PKG file (for .SIS)

- Select from menu: File -> Import
- Select “File System”
- Navigate to folder “sis” and import .PKG file
  - “EGTriangle_gcce.pkg”
- Build will automatically generate install file

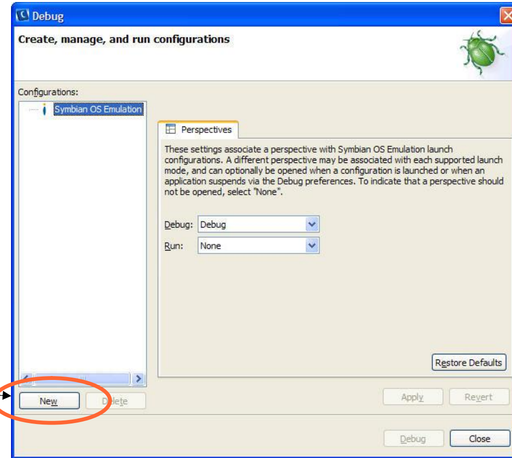
Importing .PKG file



Compiling & Debugging

- Select from menu: Project -> Build ALL
- Select from menu: Run -> Debug

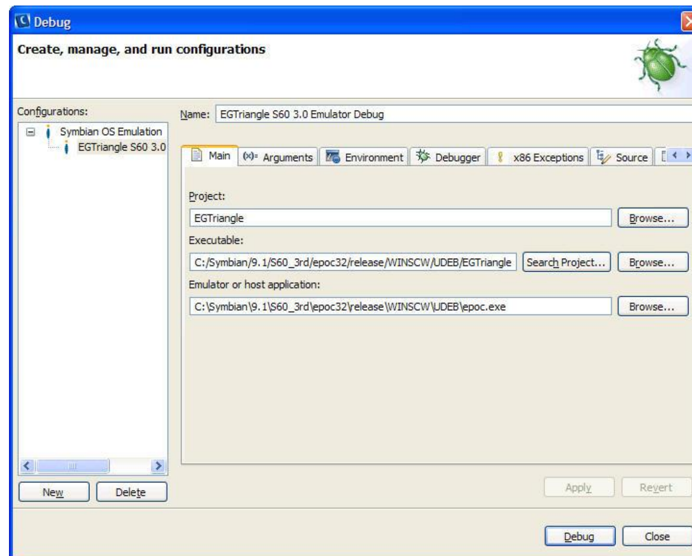
Creating debug config



Click "New" to create new debug config.

- Select NEW to create new debug configuration

Creating debug config



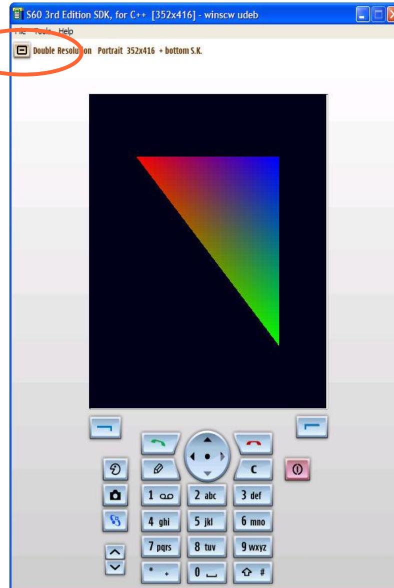
- Right values should be filled automatically by IDE

Selecting application

- When emulator starts, navigate to “Installat.” folder
- Select application to launch (EGTriangle)

Application

Click this button to cycle through resolutions and check that your application works in all resolutions.



Getting it to HW

- Go to menu: Window -> Open Perspective -> Other
- Select “Symbian (default)”
- Go to menu: Window -> Show view -> Build Configurations

Selecting build configuration

Click this button to open a list of possible build configurations. Select “S60 3.0 Phone (GCCE) Release”

Installation file

- Build the project (CTRL-B)
- Installation file is generated during build
- Select it from C/C++ Projects view
  - EGTriangle_GCCE.sis
- From context menu select “copy”
- Paste it to desktop and send using bluetooth

Fixed point programming

- Why to 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 fixed point

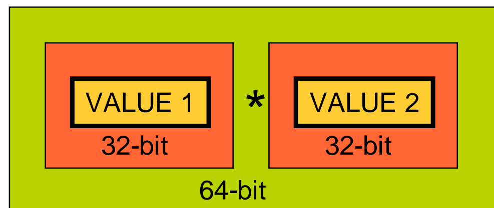
•Fixed point scale is 2^16 (65536, 0x10000).

Fixed point programming

- Examples:

```
0x0001 0000 =  "1.0f"
0x0002 0000 =  "2.0f"
0x0010 0000 =  "16.0f"
0x0000 0001 =  1/0x10000(0x10000 = 2^16)
0xffff ffff = -1/0x10000(-0x0000 0001)
```

Fixed point programming



>> 16 = RESULT

Intermediate overflow
- Higher accuracy (64-bit)
- Downscale input
- Redo range analysis

Result overflow
- Redo range analysis
- Detect overflow, clamp

-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

- Convert from floating point to fixed point

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

- Convert from fixed point to floating point

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

- Addition

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

- Multiply fixed point number with integer

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

Notes about overflows:

-conversion from float is not possible if input number is not in the right range [-32768, 32768[.

-conversion from fixed reduces accuracy (float has 25 bits for mantissa and sign, whereas fixed point uses 32 bits) E.g., (32767.0 + 1/65536 = 32767.0000152). If accuracy is crucial, convert to double to preserve the result.

-add can overflow by one bit (e.g. by adding 32767.0 + 32767.0), result overflows. If you use add for averaging, you may also divide both input numbers by two and then just add them together. This doesn't overflow in the intermediate calculations, but it loses some accuracy (lowest bit from both inputs).

-multiplying fixed point number with integer can overflow if result does not fit into 32-bit, examples: 32767.0 * 2 or 2.0 * 16384.

Fixed point programming

- MUL two FP numbers together

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

  - If another multiplier is in] -1.0, 1.0 [, no overflow
- Division of integer by integer to a fixed point result

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

- 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) (((a)*(1<<16))/(b))
```

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
- Fixed point calculations overflow easily
- Careful analysis of the range requirements is required
- Always try to use as low bit ranges as possible
  - 32x8 MUL is faster than 32x32 MUL (some ARM)
  - Using unnecessary “extra bits” slows execution
- Always add debugging code to your fixed point math

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 programming

- Complex math functions
  - Pre-calculate for the range of interest
- An example: Sin & Cos
  - Sin table between [0, 90°]
  - Fixed point angle
  - Generate other angles and Cos from the table
  - Store as fixed point ((short) (sin(angle) * 32767))
  - Performance vs. space tradeoff: calculate for all angles

Fixed point programming

- Sin
  - 90 ° = 2048 (our angle scale)
  - Sin table needs to include 0° and 90°

```
INLINE fp_sin(int angle)
{
   int phase                    = angle & (2048 + 4096);
   int subang                   = angle & 2047;

   if( phase == 0 )             return sin_table  (subang);
   else if( phase == 2048 )     return sin_table  (2048 - subang);
   else if( phase == 4096 )     return -sin_table (subang);
   else                         return -sin_table (2048 - subang);
}
```

- This function can be easily converted to be just single table lookup by precalculating SIN from 0 to 360+90 (both SIN and COS can then be referenced from the same table) if the angles are guaranteed to be between [0,360].

Example: Morphing

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

```
#define DOMORPH_16(a,b,t) (TInt16)(((((b)-(a))*(t))>>16)+(a))

void MorphGeometry(TInt16 *aOut, const TInt16 *aInA, const TInt16
   *aInB, TInt aCount, TInt aScale)
{
   int i;

   for(i=0; i<aCount; i++)
   {
        aOut[i*3+0] = DOMORPH_16(aInB[i*3+0], aInA[i*3+0], aScale);
        aOut[i*3+1] = DOMORPH_16(aInB[i*3+1], aInA[i*3+1], aScale);
        aOut[i*3+2] = DOMORPH_16(aInB[i*3+2], aInA[i*3+2], aScale);
   }
}
```

-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

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

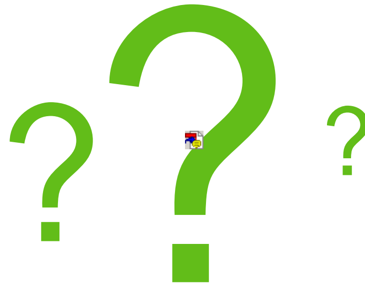
- Vertex arrays
  - OpenGL ES supports only vertex arrays
  - SW implementations get penalty from float data
  - Use as small types as possible (byte, short)
  - For HW it shouldn't make a difference, mem BW
  - With OpenGL ES 1.1 use VBOs

-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 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

Questions?



Building scalable 3D applications

Ville Miettinen

Hybrid Graphics

What is this ”mobile platform”?

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

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 HW
- Performance: 50K – 2M tris, 1M – 100M pixels

Dealing with diversity

- Problem: running the same game on 100+ different devices
  - Same gameplay but can scale video and audio
- Scalability must be built into game design
- Profile-based approach

3D content is easy to scale

- Separate low and high poly 3D models
- Different texture resolutions & compressed formats
- Scaling down special effects not critical to game play (particle systems, shadows)
  - Important to realize what is a ”special effect”
- Rendering quality controls
  - Texture filtering, perspective correction, blend functions, multi-texturing, antialiasing

Building scalable 3D apps

- OpenGL ES created to standardize the API and behavior
  - ES does not attempt to standardize performance
  - Two out of three ain't bad
- Differences between SW/HW configurations
  - Trade-off between flexibility and performance
  - Synchronization issues

Building scalable 3D apps

- Scale upwards, not downwards
  - Bad experiences of retro-fitting HW titles to SW
  - Test during development on lowest-end platform
- Both programmers and artists need education
  - Artists can deal with almost anything as long as they know the rules…
  - And when they don’t, just force them (automatic checking in art pipeline)

Reducing state changes

- Don't mix 2D and 3D calls !!!!
  - Situation may become better in the future, though...
- Unnecessary state changes root of all evil
  - Avoid changes affecting the vertex pipeline
  - Avoid changes to the pixel pipeline
  - Avoid changing textures

”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: Important to keep shaders as simple as possible
  - Do 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

Of models and stripping

- Use buffer objects of ES 1.1
  - Only models changed manually every frame need vertex pointers
  - Many LOD schemes can be done just by changing index buffers
- Keep data formats short and simple
  - Better cache coherence, less memory used

Triangle data

- Minimize number of rendering calls
  - Trade-off between no. of render calls & culling efficiency
  - Combine strips using degenerate triangles
  - Understanding vertex caching
    - Automatically optimize vertex access order
    - Triangle lists better than their reputation
- Optimize data in your art pipeline (exporters)
  - Welding vertices with same attributes (with tolerance)
    - Vertices/triangle ratio in good data 0.7-1.0
  - Give artists as much automatic feedback as possible

Transformations and matrices

- 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
  - Characters, natural motion: grass, trees, waves
- ES 1.1: Point sprites

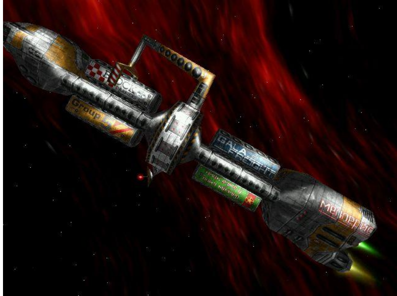
Lighting and materials

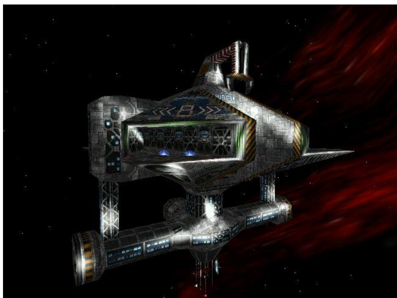
- Fixed-function lighting pipelines are so 1990s
  - Drivers implemented badly even in desktop space
  - In practice only single directional light fast
  - OpenGL's attenuation model difficult to use
  - Spot cutoff and specular model cause aliasing
  - No secondary specular color

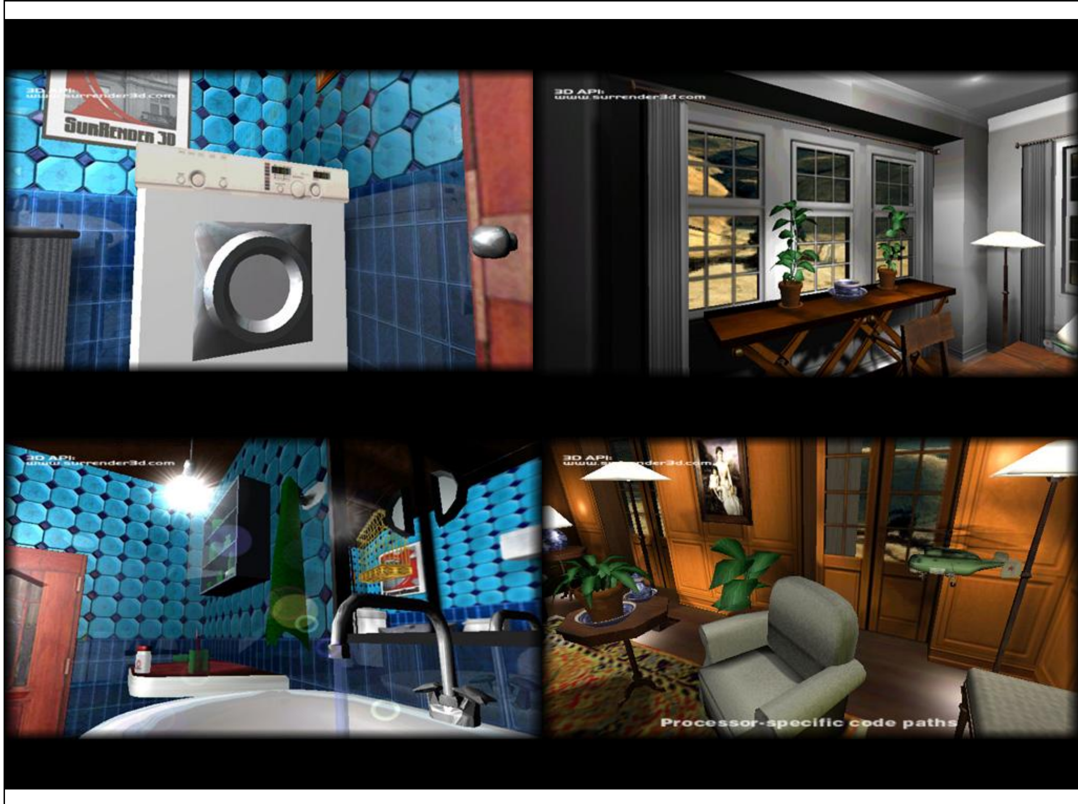
Lighting: the fast way

- While we're waiting for OpenGL ES 2.0...
  - 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
- Color material tracking for changing materials
- Flat shading is for flat models!

Illumination using multitexturing






3D API:
www.surrender3d.com
SurRender 3D
3D API:
www.surrender3d.com
3D API:
www.surrender3d.com
Processor-specific code paths

Textures

- Mipmaps always a Good Thing™
  - Improved cache coherence and visual quality
  - ES 1.1 supports auto mipmap generation
- Different strategies for texture filtering
- SW: Perspective correction not always needed
- Avoid modifying texture data
- Keep textures ”right size”, use compressed textures

Textures

- Multitexturing
  - Needed for texture-based lighting
  - Always faster than doing multiple rendering passes
  - ES 1.1: support at least two texturing units
  - ES 1.1: TexEnvCombine neat toy
- Combine multiple textures into single larger one
  - Reduce texture state changes (for fonts, animations, light maps)

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

Object ordering

- Sort objects into optimal rendering order
  - Minimize shader changes
  - Keep objects in front-to-back order
    - Improves Z-buffering efficiency
  - Satisfying both goals: bucketize objects by shader, sort buckets by Z

Thank you!

- Any questions?

M3G Overview

Tomi Aarnio

Nokia Research Center

I’ll give you an overview of the Mobile 3D Graphics API, with some performance tips.

Mark will then show you actual code examples.

Objectives

- Get an idea of the API structure and feature set
- Learn practical tricks not found in the spec

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.

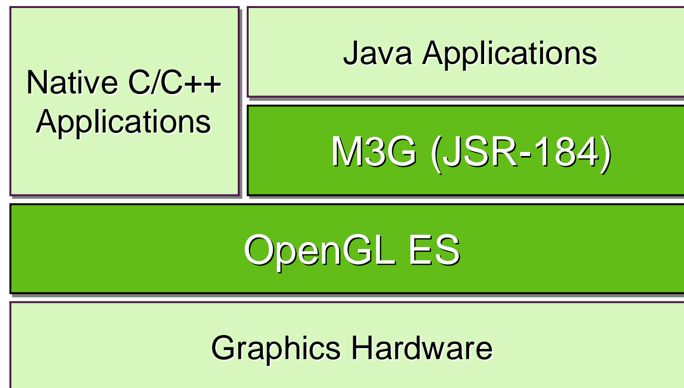
Prerequisites

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

What you should know to get the most out of this session?

Well, I’m sure you have adequate background since you’re still sitting here after the first presentations.

Mobile 3D Graphics APIs



This diagram you just saw a minute ago, but I'm replicating it here to emphasize that M3G really builds on the feature set of OpenGL ES.

Why Should You Use Java?

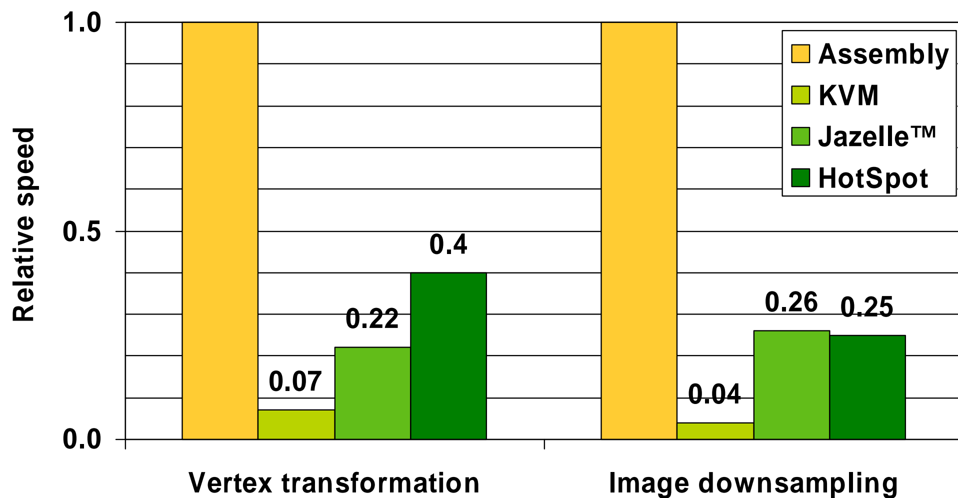
- It has the largest and fastest-growing installed base
  - 1.2B Java phones had been sold by June 2006 (source: Ovum)
  - Nokia alone had sold 350M Java phones by the end of 2005
  - Less than 50M of those also supported native S60 applications
- It increases productivity compared to C/C++
  - Memory protection, type safety ➔ fewer bugs
  - Fewer bugs, object orientation ➔ better productivity

So why should you use Java in the first place? Two reasons.

First, for most devices out there, it's the only way to get your code in. Phones with an open OS are few and far between. Let the figures here speak for themselves.

Second, it's easier and faster to write code in Java compared to C/C++, not to mention assembly.

Java Will Remain Slower



Benchmarked on an ARM926EJ-S processor with hand-optimized Java and assembly code

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 tall orange bars represent native code.

First we have the KVM, which is used in 90% of phones today. You can see that it's quite slow.

Then we have Jazelle, which is a hardware accelerator from ARM. Big improvement, but still not close.

Finally we have a HotSpot VM from Sun. Well, 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. Not good.

Why?

- Array bounds checking
- Dynamic type checking
- No stack allocation (heap only)
- Garbage collection
- Slow Java-native interface
- No access to special CPU features
- Stack-based (non-RISC) bytecode
- Unpredictable JIT compilers

No Java compiler or accelerator can fully resolve these issues

So why is it that not even hardware acceleration can make Java run as fast as native code? Some reasons are listed on this slide.

First we have things related to run-time error checking – array bounds checking, dynamic type checking, managed memory allocation.

Then we have the slow Java-native interface. Function calls are slow, and data traffic is slower still.

One important thing is that you get no access to SIMD instructions and other special CPU features. When you're working in native code, you can get a big performance boost by writing some of your critical routines in assembly and using the ARM equivalents of Intel's MMX and SSE.

Then finally, there's the problem that Java bytecode has a stack-based execution model, whereas the ARM and probably most other embedded CPUs are RISC processors. It's hard for the VM to compile stack-based code into fast register-based code, and that's probably one of the reasons why the HotSpot VM performs so badly. But there are other reasons, too.

So the bottom line is that Java will remain slower and consume more memory than native code, and we just have to live with that fact. The performance gap will become smaller, but it will not go away.

M3G Overview

Design principles

Getting started

Basic features

Performance tips

Deforming meshes

Keyframe animation

Summary & demos

Here we have the agenda.

I'll start by explaining some fundamental design issues, then proceed through the API in bottom-up order.

M3G Design Principles

#1 | No Java code along critical paths

- Move all graphics processing to native code
  - Not only rasterization and transformations
  - Also morphing, skinning, and keyframe animation
  - Keep all data on the native side to avoid Java-native traffic

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

Well, 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.

M3G Design Principles

#2

Cater for both software and hardware

- Do not add features that are too heavy for software engines
  - Such as per-pixel mipmapping or floating-point vertices
- Do not add features that break the OpenGL 1.x pipeline
  - Such as hardcoded transparency shaders

Secondly, we wanted the API to work well on today's software-based handsets as well as the hardware-accelerated ones in the future.

We had a rule that features that cannot be done efficiently in software will not be included. Per-pixel mipmapping and floating-point vertex arrays fell into that category.

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.

M3G Design Principles

#3 Maximize developer productivity

- Address content creation and tool chain issues
  - Export art assets into a compressed file (.m3g)
  - Load and manipulate the content at run time
  - Need scene graph and animation support for that
- Minimize the amount of “boilerplate code”

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.

M3G Design Principles

#4	Minimize engine complexity
#5	Minimize fragmentation
#6	Plan for future expansion

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 some 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 ugly legacy.

Why a New Standard?

- OpenGL ES is too low-level
  - Lots of Java code, function calls needed for simple things
  - No support for animation and scene management
  - Fails on Design Principles 1 (performance) and 3 (productivity)
  - …but may become practical with faster Java virtual machines
- Java 3D is too bloated
  - A hundred times larger (!) than M3G
  - Still lacks a file format, skinning, etc.
  - Fails on Design Principles 1, 3, and 4 (code size)

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 our 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.

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

M3G Overview

Design principles

Getting started

Basic features

Performance tips

Deforming meshes

Keyframe animation

Summary & demos

Okay, let's first take a look at the M3G programming model, then continue with the features.

The Programming Model

- Not an "extensible scene graph"
  - Rather a black box – much like OpenGL
  - No interfaces, events, or render callbacks
  - No threads; all methods return only when done
- Scene update is decoupled from rendering
  - `render` ➔ Draws an object or scene, no side-effects
  - `animate` ➔ Updates an object or scene to the given time
  - `align` ➔ Aligns scene graph nodes to others

What we have here is a 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.

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.

Main Classes

Graphics3D — 3D graphics context
Performs all rendering

Loader — Loads individual objects
and entire scene graphs
(.m3g and .png files)

World — Scene graph root node

Rendering State

- Graphics3D contains global state
  - Frame buffer, depth buffer
  - Viewport, depth range
  - Rendering quality hints
- 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

Graphics3D: How To Use

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

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

M3G Overview

Design principles

Getting started

Basic features

Performance tips

Deforming meshes

Keyframe animation

Summary & demos

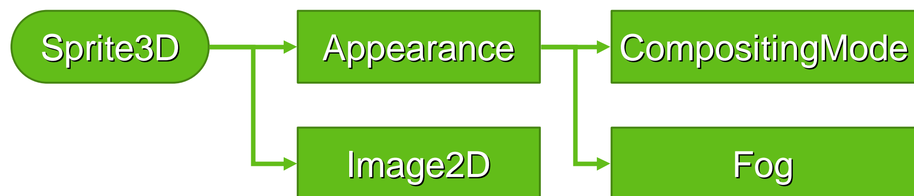
Renderable Objects

Sprite3D — 2D image placed in 3D space; Always facing the camera

Mesh — Made of triangles; Base class for meshes

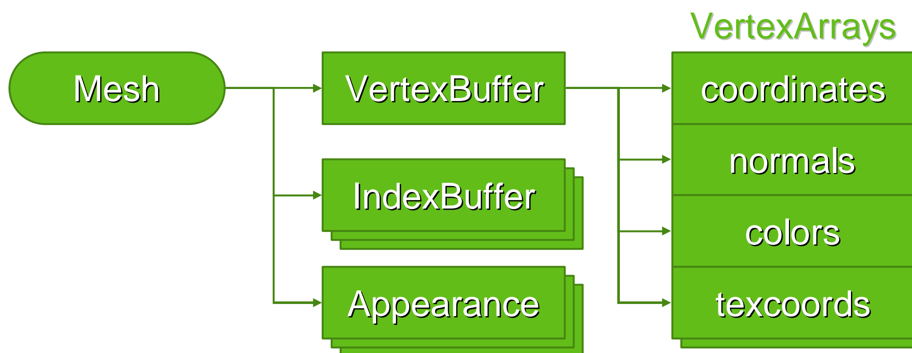
Sprite3D

- 2D image with a position in 3D space
- Scaled mode for billboards, trees, etc.
- Unscaled mode for text labels, icons, etc.
- Not useful for particle effects – too much overhead



Mesh

- A common VertexBuffer, referencing VertexArrays
- IndexBuffers (submeshes) and Appearances match 1:1



VertexBuffer Types

	Byte	Short	Fixed	Float	2D	3D	4D
Vertices	✓	✓	✗	✗	✗	✓	✗
Texcoords	✓	✓	✗	✗	✓	✓	✗
Normals	✓	✓	✗	✗		✓	
Colors	✓		✗	✗		✓	✓

Relative to OpenGL ES 1.1

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.

IndexBuffer Types

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

Relative to OpenGL ES 1.1 + point sprite extension

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

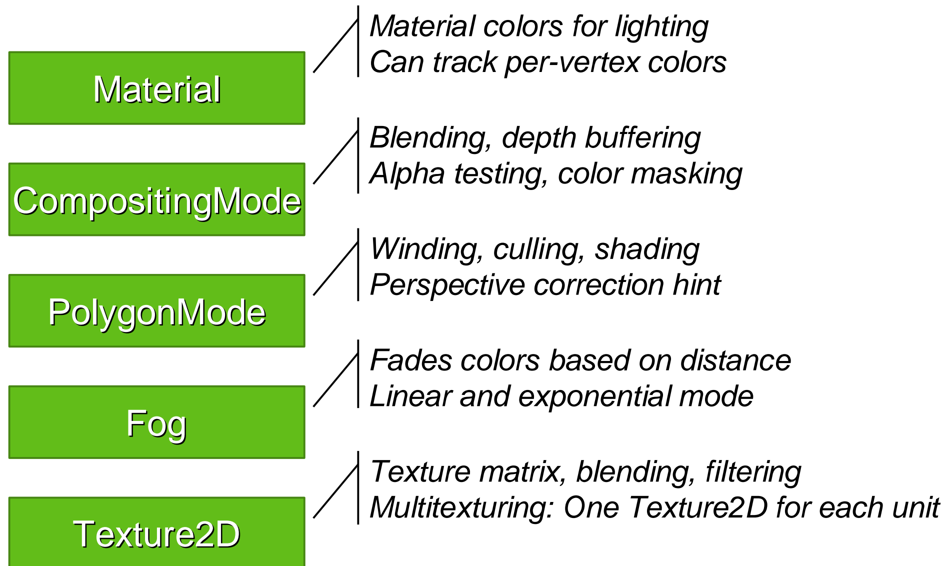
On hindsight, triangle lists should've been included, since they are easier to use and are not necessarily any slower than strips.

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

Buffer Objects

- Vertices and indices are stored on server side
  - Very similar to OpenGL Buffer Objects
  - Allows caching and preprocessing (e.g., bounding volumes)
- Tradeoff – Dynamic updates have some overhead
  - At the minimum, just copying in the Java array contents
  - In the worst case, may trigger vertex preprocessing

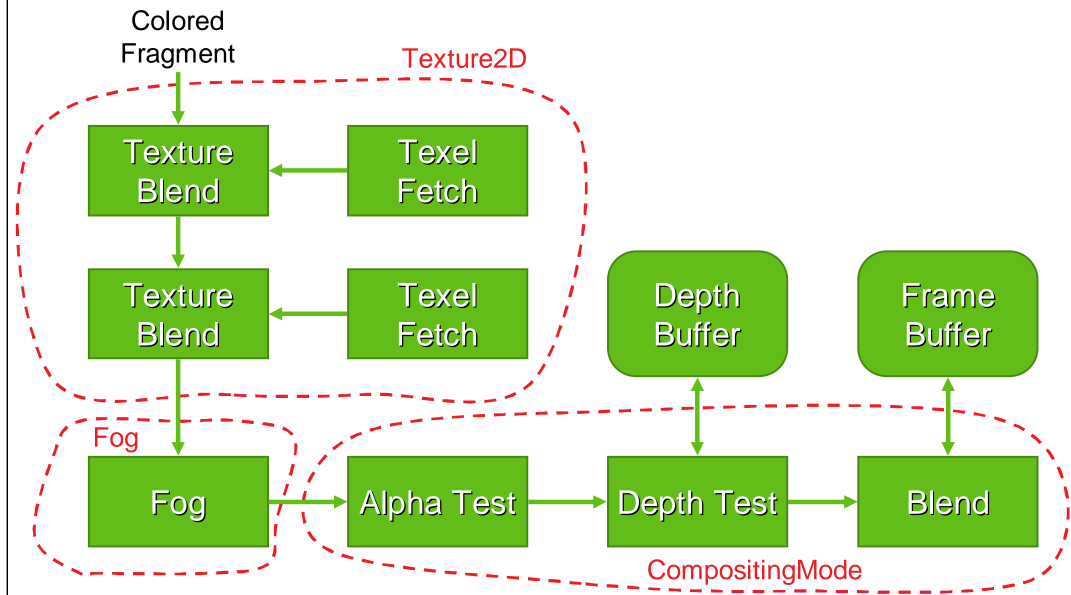
Appearance Components



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

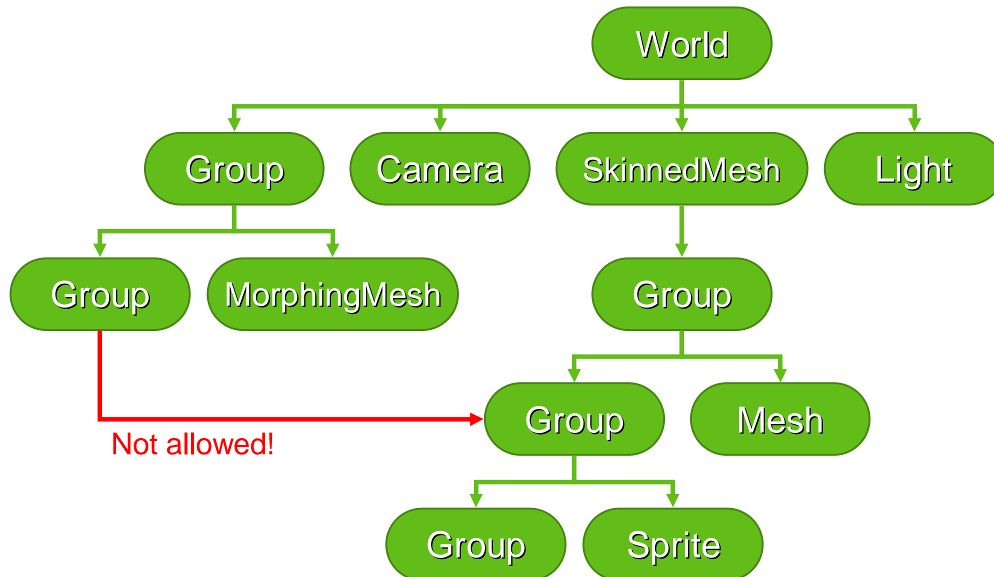
This saves memory space, reduces garbage collection, and allows implementations to quickly sort objects based on their rendering state.

The Fragment Pipeline



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.

The Scene Graph

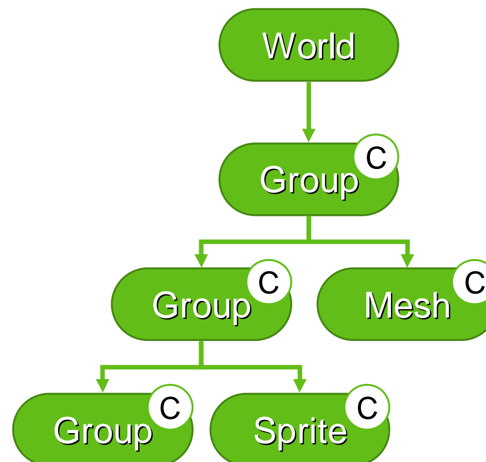


Scene graph nodes can't have more than one parent, so the scene graph is actually just a tree.

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.

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
- Composite: **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

The File Format

Characteristics

- Individual objects, entire scene graphs, anything in between
- Object types match 1:1 with those in the API
- Optional ZLIB compression of selected sections
- Can be decoded in one pass – no forward references
- Can reference external files or URIs (e.g. textures)
- Strong error checking

M3G Overview

Design principles

Getting started

Basic features

Performance tips

Deforming meshes

Keyframe animation

Summary & demos

Retained Mode

- Use the retained mode
  - Do not render objects separately – place them in a World
  - Minimizes the amount of Java code and method calls
  - Allows the implementation to do view frustum culling, etc.
- Keep Node properties simple
  - Favor the T R S components over M
  - Avoid non-uniform scales in S
  - Avoid using the alpha factor

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.

Rendering Order

- Use layers to impose a rendering order
  - Appearance contains a layer index (an integer)
  - Defines a global ordering for submeshes & sprites
  - Can simplify shader state for backgrounds, overlays
  - Also enables multipass rendering in retained mode
- Optimize the rendering order
  - Shader state sorting done by the implementation
  - Use layers to force back-to-front ordering

Textures

- Use multitexturing to save in T&L and triangle setup
- Use mipmapping to save in memory bandwidth
- Combine small textures into texture atlases
- Use the perspective correction hint (where needed)
  - Usually much faster than increasing triangle count
  - Nokia: 2% fixed overhead, 20% in the worst case

Meshes

- Minimize the number of objects
  - Per-mesh overhead is high, per-submesh also fairly high
  - Lots of small meshes and sprites to render ➔ bad
  - Ideally, everything would be in one big triangle strip
  - But then view frustum culling doesn't work ➔ bad
- Strike a balance
  - Merge simple meshes that are close to each other
  - Criteria for “simple” and “close” will vary by device

Shading State

- Software vs. hardware implementations
  - SW: Minimize per-pixel operations
  - HW: Minimize shading state changes
  - HW: Do not mix 2D and 3D rendering
- In general, OpenGL ES performance tips apply

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

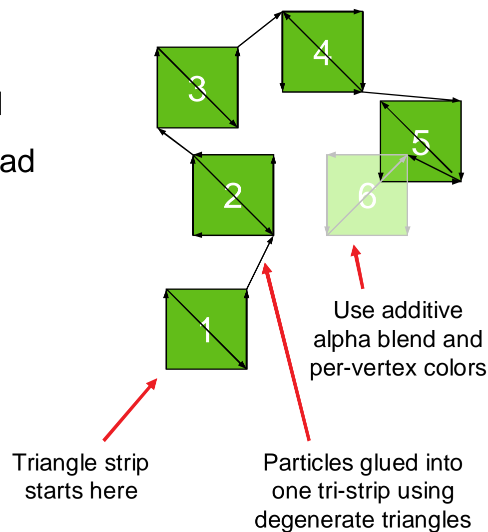
Particle Effects

Several problems

- Point sprites are not supported
- Sprite3D has too much overhead

Put all particles in one Mesh

- One particle == two triangles
- All glued into one triangle strip
- Update vertices to animate
  - XYZ, RGBA, maybe UV



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.

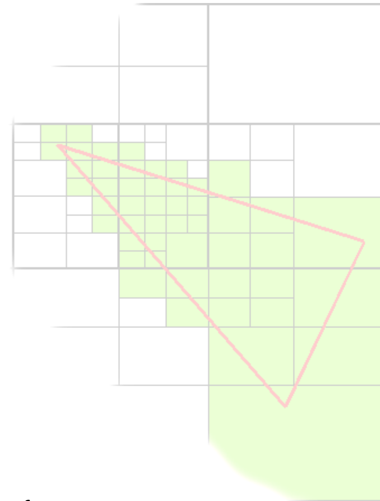
It is more efficient to represent particles as textured quads, all glued into one big triangle strip that can be drawn in a single call. To make the particles face the viewer, set up automatic node alignment for the Mesh that encloses the particle system.

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

Terrain Rendering

Easy terrain rendering

- Split the terrain into tiles (Meshes)
- Put the meshes into a scene graph
- The engine will do view frustum culling



Terrain rendering with LOD

- Preprocess the terrain into a quadtree
- Quadtree leaf node == Mesh object
- Quadtree inner node == Group object
- Enable nodes yourself, based on the view frustum

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.

M3G Overview

Design principles

Getting started

Basic features

Performance tips

Deforming meshes

Keyframe animation

Summary & demos

Deforming Meshes

MorphingMesh — *Vertex morphing mesh*

SkinnedMesh — *Skeletally animated mesh*

MorphingMesh

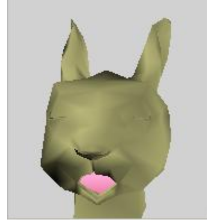
- Traditional vertex morphing animation
  - Can morph any vertex attribute(s)
  - A base mesh **B** and any number of morph targets $\mathbf{T}_i$
  - Result = weighted sum of morph deltas

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

- 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

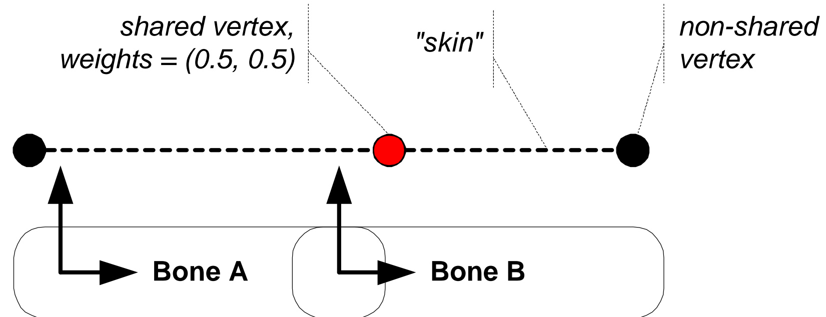
$$v' = \sum_i w_i \mathbf{M}_i \mathbf{B}_i v$$

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

In the equation,

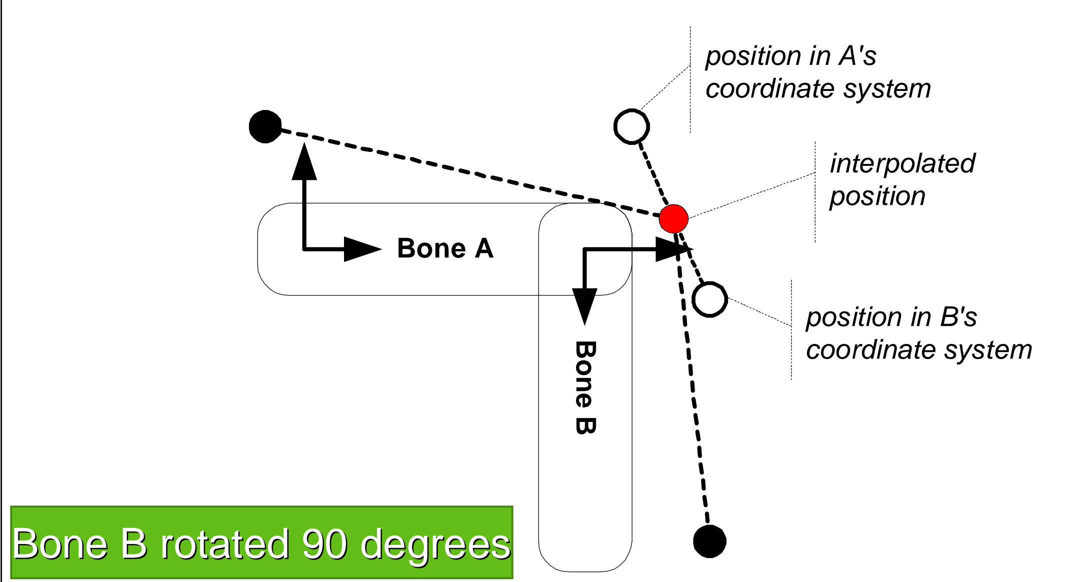
- **v** is the vertex position in the SkinnedMesh node’s coordinates
- $\mathbf{B}_i$ is the fixed at-rest transformation from SkinnedMesh to bone $\mathbf{N}_\mathbf{i}$
- $\mathbf{M}_i$ is the dynamic transformation from bone $\mathbf{N}_\mathbf{i}$ to SkinnedMesh
- $\mathbf{w}_\mathbf{i}$ is the weight of bone $\mathbf{N}_\mathbf{i}$ (the weights are normalized)
- $\mathbf{0} \leq \boldsymbol{i} \leq \mathbf{N}$, 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

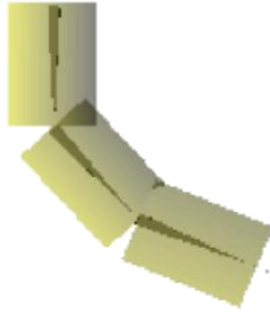
SkinnedMesh



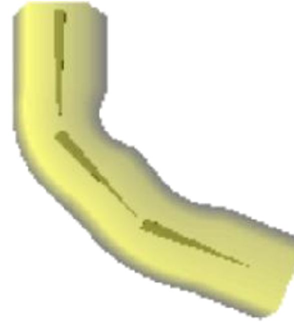
The empty dots show where the vertex would end up if it were associated with just one of the bones, respectively.

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

SkinnedMesh



No skinning



Smooth skinning
two bones per vertex

M3G Overview

Design principles

Getting started

Basic features

Performance tips

Deforming meshes

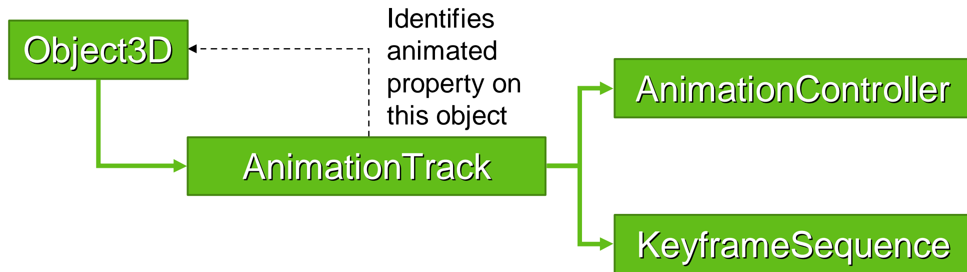
Keyframe animation

Summary & demos

Animation Classes

- **KeyframeSequence** — *Storage for keyframes. Defines interpolation mode*
- **AnimationController** — *Controls the playback of one or more sequences*
- **AnimationTrack** — *A link between sequence, controller and target*
- **Object3D** — *Base class for all objects that can be animated*

Animation Classes



KeyframeSequence

KeyframeSequence

Keyframe is a time and the value of a property at that time

Can store any number of keyframes

Several keyframe interpolation modes

Can be open or closed (looping)

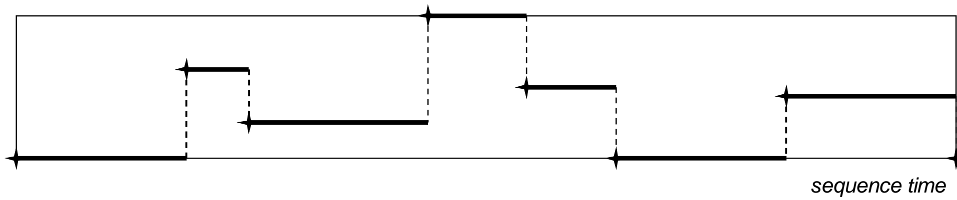


Diagram courtesy of Sean Ellis, Superscape

KeyframeSequence

KeyframeSequence

Keyframe is a time and the value of a property at that time

Can store any number of keyframes

Several keyframe interpolation modes

Can be open or closed (looping)

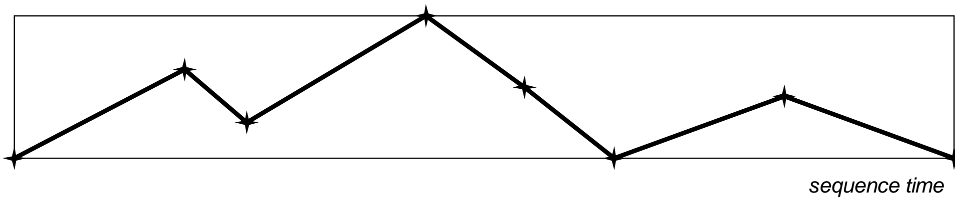


Diagram courtesy of Sean Ellis, Superscape

KeyframeSequence

KeyframeSequence

Keyframe is a time and the value of a property at that time

Can store any number of keyframes

Several keyframe interpolation modes

Can be open or closed (looping)

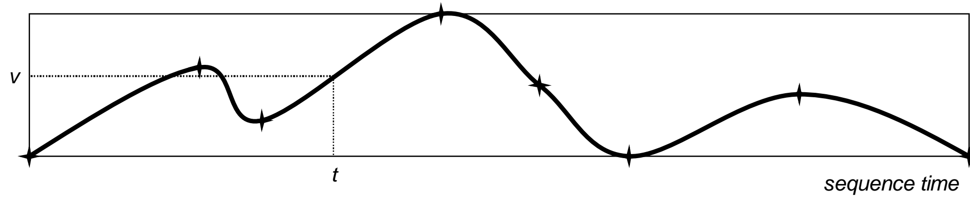


Diagram courtesy of Sean Ellis, Superscape

AnimationController

AnimationController

Can control several animation sequences together

Defines a linear mapping from world time to sequence time

Multiple controllers can target the same property

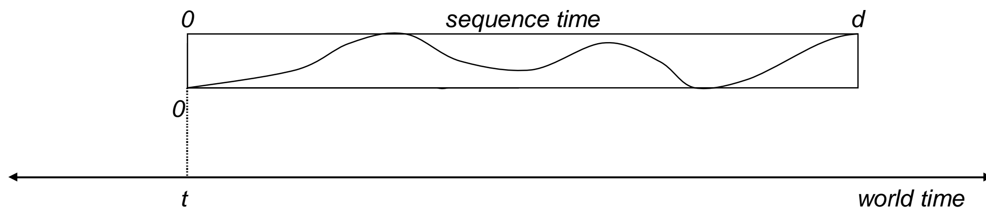


Diagram courtesy of Sean Ellis, Superscape

Animation

1. Call animate(worldTime)

4. Apply value to animated property

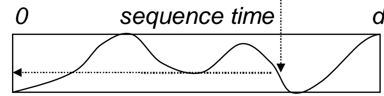
2. Calculate sequence time from world time

Object3D

AnimationTrack

AnimationController

KeyframeSequence



3. Look up value at this sequence time

Diagram courtesy of Sean Ellis, Superscape

Animation

Tip: Interpolate quaternions as ordinary 4-vectors

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

M3G Overview

Design principles

Getting started

Basic features

Performance tips

Deforming meshes

Keyframe animation

Summary & demos

Predictions

- Resolutions will grow rapidly from 128x128 to VGA
  - Drives graphics hardware into all high-resolution devices
  - Software rasterizers can't compete above 128x128
- Bottlenecks will shift to Physics and AI
  - Bottlenecks today: Rasterization and any Java code
  - Graphics hardware will take care of geometry and rasterization
  - Java hardware will increase performance to within 50% of C/C++
- Java will reinforce its position as the dominant platform

Summary

- M3G enables real-time 3D on mobile Java
  - By minimizing the amount of Java code along critical paths
  - Designed for both software and hardware implementations
- Flexible design leaves the developer in control
  - Subset of OpenGL ES features at the foundation
  - Animation & scene graph features layered on top

Installed base growing by the millions each month

Demos

Playman Winter Games – Mr. Goodliving

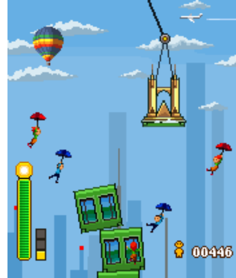
2D

3D

Playman World Soccer – Mr. Goodliving

- An interesting 2D/3D hybrid
- Cartoon-like 2D characters set in a 3D scene
- 2D overlays for particle effects and status info

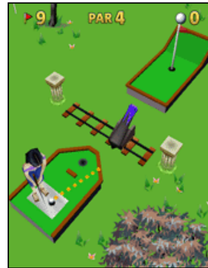
Tower Bloxx – Sumea

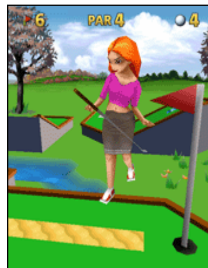


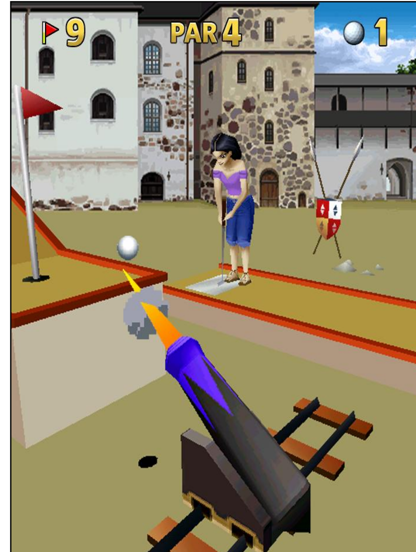
- Puzzle/arcade mixture
- Tower building mode is in 3D, with 2D overlays and backgrounds
- City building mode is in pure 2D

Mini Golf Castles – Sumea

- 3D with 2D background and overlays
- Skinning used for characters
- Realistic ball physics







Q&A

Thanks: Sean Ellis, Kimmo Roimela,
Nokia M3G team, JSR-184 Expert Group,
Mr. Goodliving (RealNetworks),
Sumea (Digital Chocolate)

Using M3G

Mark Callow

Chief Architect

Agenda

- Game Development Process
- Asset Creation
- Program Development
- MIDlet Structure
- A MIDlet Example
- Challenges in Mobile Game Development
- Publishing Your Content

M3G Game Demo

EXTREME AIR SNOWBOARDING™

SUMEA

Copyright 2005, Digital Chocolate Inc.

Game Development Process

- Traditional Java Game

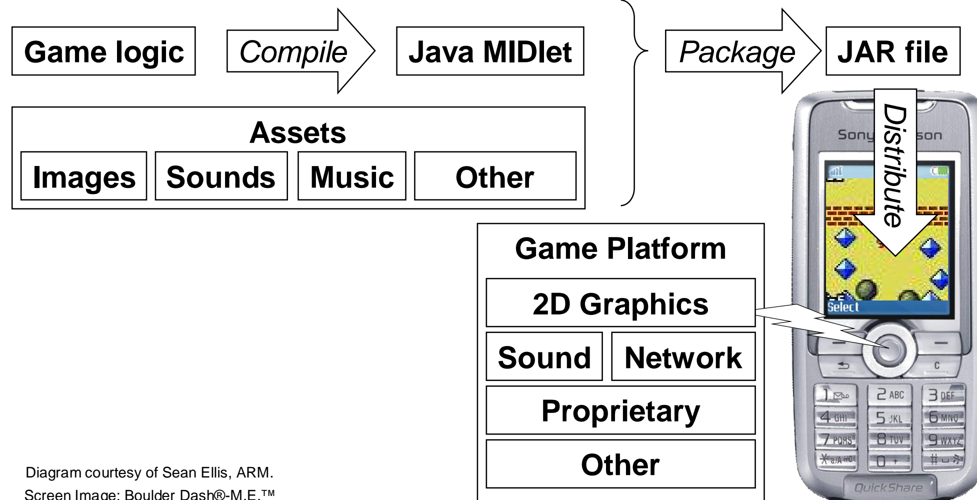


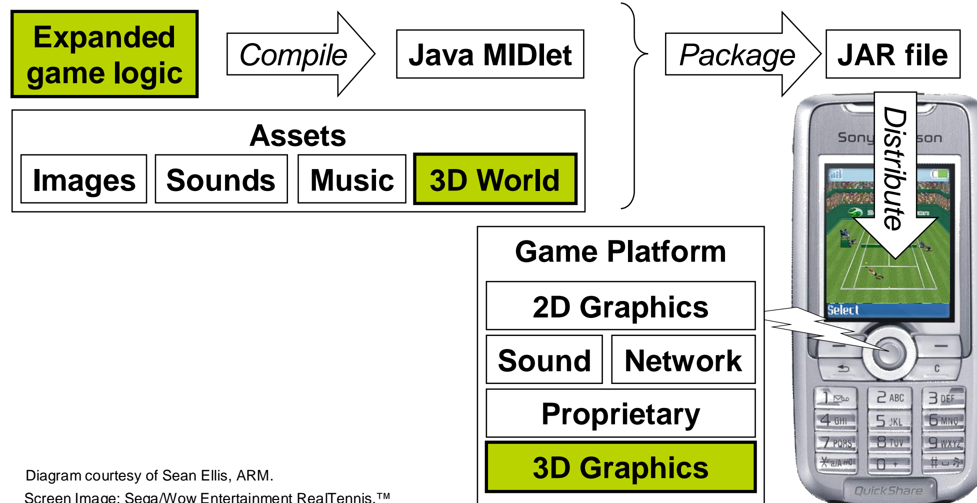
Diagram courtesy of Sean Ellis, ARM.
Screen Image: Boulder Dash®-M.E.™

Let’s have a quick look at the various steps involved in creating a traditional Java game. We have a game platform such as MIDP 2 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 assets such as images, sounds and music. Finally we distribute the game package to the customers.

I’ll be discussing each of these steps during the presentation.

M3G Development Process

- How M3G Fits



Now what does M3G bring to the party? First and foremost of course is 3D graphics. This means your assets will include 3d models or a 3d scene. You also have the opportunity 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.

Asset Creation

- Textures & Backgrounds

For any real m3g application, some art assets have to be created before the program can do anything useful. So let's 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.

Asset Creation

- Audio Tools

Expanded game logic

Compile

Assets

Images | Sounds | Music

Audio Production Tool; e. g.

- Sony Sound Forge®

Commonly Used Formats:

- Wave, AU, MP3, SMAF

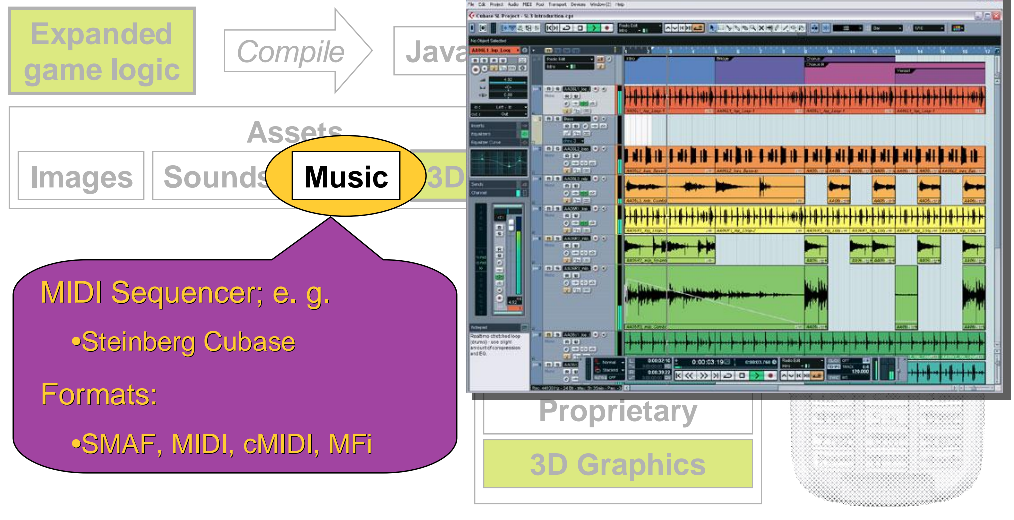
3D Graphics

The J2ME (MIDP 2.0) specification does not seem to have a list of formats for which support is required. 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 have contain both recorded audio and synthesizer sequences.

Asset Creation

- Music Tools



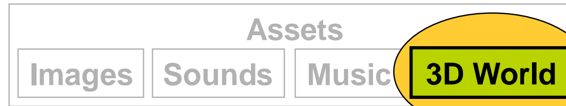
MFi (Melody Format for i-Mode) is supported on all i-Mode phones worldwide. As with SMAF, MFi can hold both MIDI-like data (cMIDI) and custom samples.

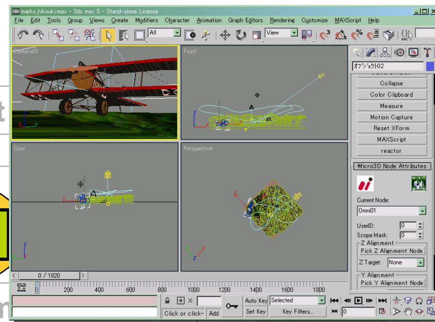
cMIDI is compact MIDI which reduces the range of allowed MIDI data thereby reducing the file size.

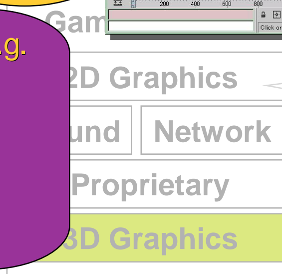
For all of your audio , you will mostly be dealing with hardware designed for ring tones. It is important that you understand the capabilities of the chip in your target phone.

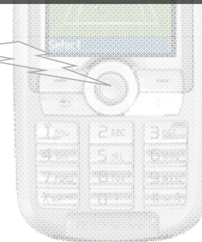
Asset Creation

- 3D Models



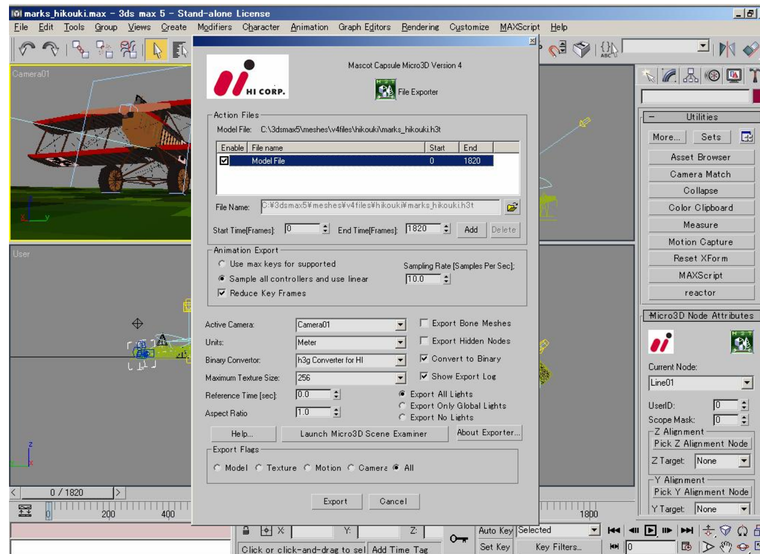




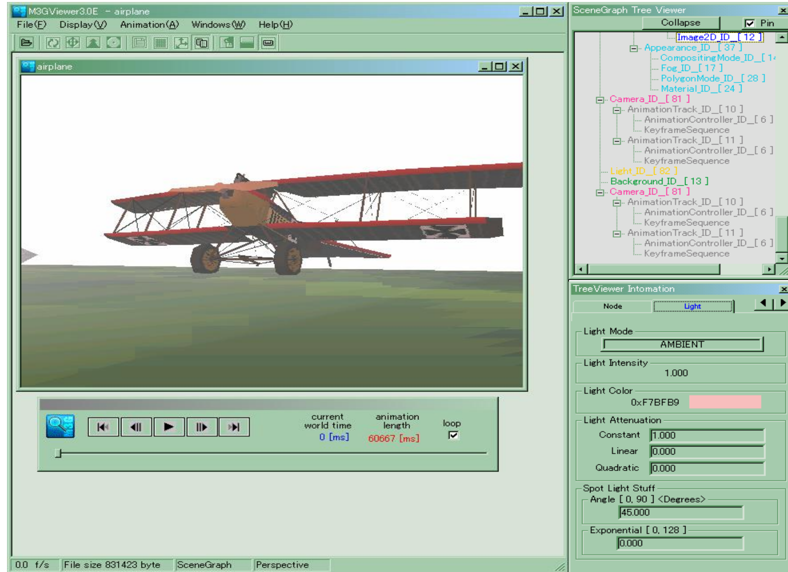


3d Modeler with M3G plug-in; e.g.
- Lightwave
- Maya
- 3d studio max
- Softimage|XSI

Export 3d Model to M3G



M3G File Viewer



Demo: On a Real Phone

Tips for Designers 1

- *TIP: Don't use GIF files*
  - *The specification does not require their support*
- *TIP: Create the best possible quality audio & music*
  - It's much easier to reduce the quality later than increase it
- *TIP: Polygon reduction tools & polygon counters are your friends*
  - Use the minimum number of polygons that conveys your vision satisfactorily

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

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.

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*

Use of linear interpolation for quaternions was already mentioned in Tomi's presentation.

Tips for Designers 3

- *TIP: Use background images*
  - Can be scaled, tiled and scrolled very flexibly
  - Generally much faster than sky boxes or similar
- *TIP: Use sprites as impostors & labels*
  - Generally faster than textured quads
  - Unscaled mode is (much) faster than scaled
- *LIMITATION: Sprites are not useful for particle systems*

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

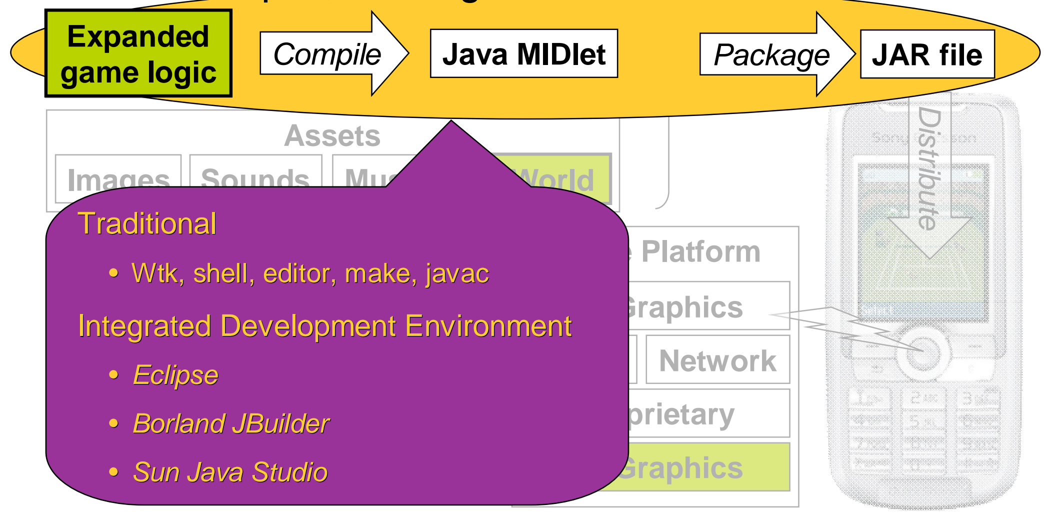
In some implementations Loader.load("/img.png") will load the image file via a MIDP image because native code is unable to read from a java stream. This requires more memory during loading.

Agenda

- Game Development Process
- Asset Creation
- Program Development
- MIDlet Structure
- A MIDlet Example
- Challenges in Mobile Game Development
- Publishing Your Content

Program Development

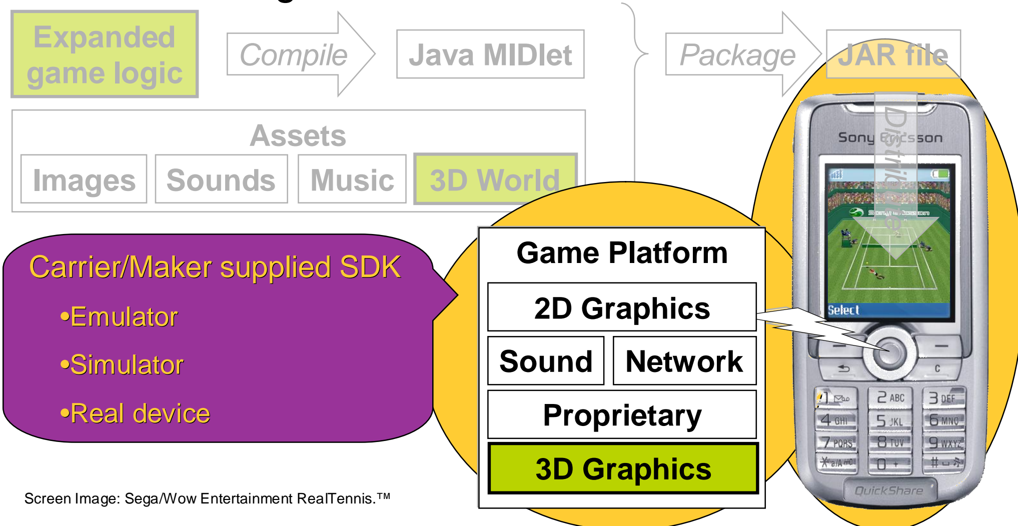
- Edit, Compile, Package



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.

You can also use Sun’s Wireless Tool Kit, or similar, which saves you from having to write a make file and lets you build your MIDlet with the push of a button.

Alternatively you can use a full IDE such as Borland’s JBuilder, Sun’s Java Studio or Eclipse.



For testing and debugging you need to use an SDK supplied by either the carrier 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.

Sun's J2ME Wireless Toolkit (WTK 2.2) provides a generic emulator for MIDP/CLDC. Two problems must be noted with this 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 not load M3G files that encode KeyframeSequence values as short. They must be float. Several carriers make their SDK's by customizing WTK even though the JVM and the MIDP & 3D renderer implementations used in WTK are often not those used in the real phones.

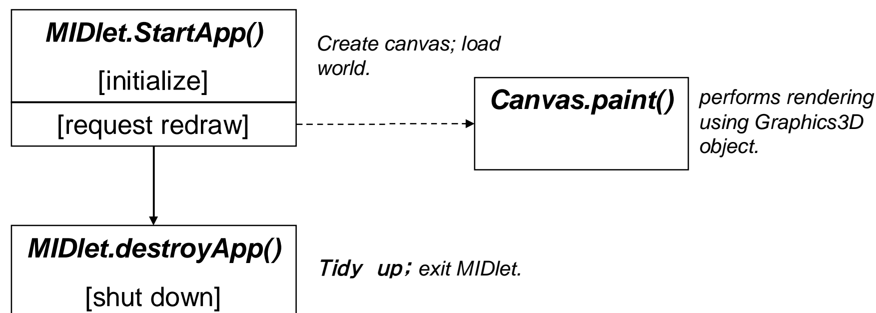
This is a common problem with "emulators". They can be quite different from the real devices and there is typically no relationship between performance in an "emulator" and performance on the real device.

Agenda

- Game Development Process
- Asset Creation
- Program Development
- MIDlet Structure
- A MIDlet Example
- Challenges in Mobile Game Development
- Publishing Your Content

The Simplest MIDlet

- Derived from MIDlet,
- Overrides three methods

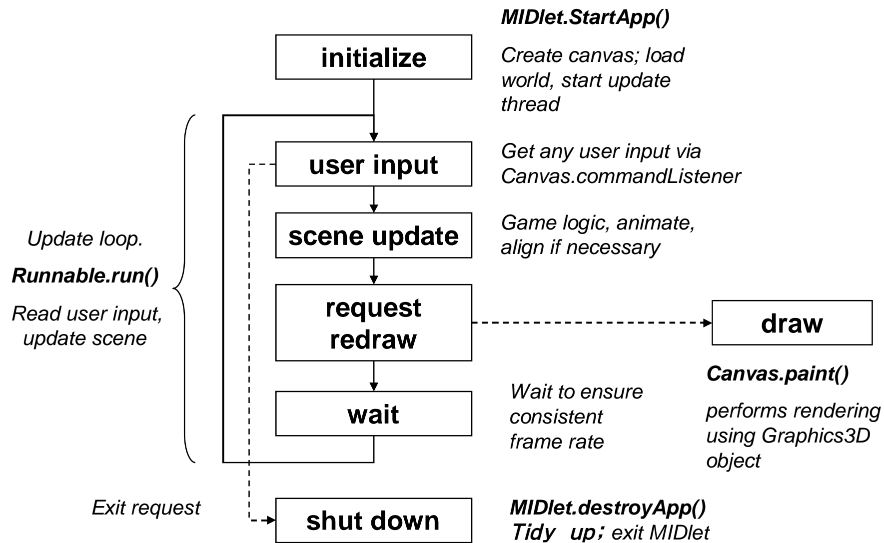


- And that's it.

We've looked at creating assets and tools to use 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.

A More Interesting MIDlet



Flow-chart courtesy of Sean Ellis, Superscape

So, here's the diagram updated to show 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.

Initialize

- Load assets: world, other 3D objects, sounds, etc.
- Find any objects that are frequently used
- Perform game logic initialization
- Initialize display
- Initialize timers to drive main update loop

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. One tip here that works well is to divorce the logic from the representation. Instead of rotating the 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. 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.

Draw Tips

- *TIP: Don’t do 2D drawing while Graphics3D is bound*

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.

MIDlet Review

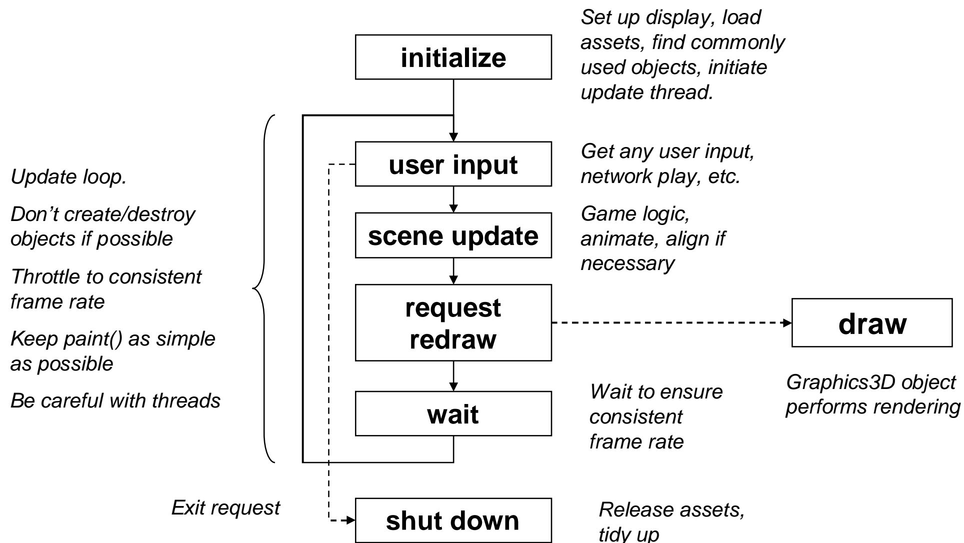


Diagram courtesy of Sean Ellis, Superscape

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

- Game Development Process
- Asset Creation
- Program Development
- MIDlet Structure
- A MIDlet Example
- Challenges in Mobile Game Development
- Publishing Your Content

Demo: UsingM3G MIDlet

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

UsingM3G MIDlet

- Displays Mesh, MorphingMesh and SkinnedMesh
- Loads data from .m3g files
- View can be changed with arrow keys
- Animation can be stopped and started
- Animation of individual meshes can be stopped and started.
- Displays frames per second.

UsingM3G Framework

```
import java.io.IOException;
import javax.microedition.lcdui.*;
import javax.microedition.midlet.*;

public class Cans extends MIDlet implements CommandListener {
 Command cmdExit = new Command("Exit", Command.SCREEN, 1);
 Command cmdPlayPause = new Command("Ctrl",Command.SCREEN,1);
 private TargetCanvas tcanvas = null;
 Thread renderingT = null;
 private String Filename = "/coffee.m3g";

 public void startApp() {
     if (tcanvas == null)
       init();

     renderingT = new Thread(tcanvas);
     renderingT.start();
     tcanvas.startPlay();
 }
```

We've called our MIDlet class Cans. The interesting parts are highlighted in green: the override of startApp() and initialization of a couple of commands. startApp() initializes everything then kicks off the rendering thread.

Thread.start() calls the thread's run() method. We'll look at that later.

UsingM3G Framework

```
public void pauseApp() {
  if (tcanvas.isPlaying)
    tcanvas.pausePlay();
  renderingT.yield();
  renderingT = null;
}

public void destroyApp(boolean u) {
  pauseApp()
  tcanvas = null;
}
```

Here are the overrides of pauseApp() & destroyApp(). They are very similar functions.

UsingM3G Framework

```
synchronized public void commandAction(Command c,
                                       Displayable d)
{
  if (c==cmdExit) {
    notifyDestroyed();
    return;
  } else if (c==cmdPlayPause) {
    if (tcanvas.isPlaying)
      tcanvas.pausePlay();
    else
      tcanvas.startPlay();
  }
}
```

This shows how to handle the exit and play-pause commands. You can study this yourselves later.

UsingM3G Initialization

```
// From class Cans
public void init() {
  Display disp = Display.getDisplay(this);
  tcanvas = new TargetCanvas(Filename);
  if (tcanvas.hasException)
    notifyDestroyed();
  tcanvas.setCommandListener(this);
  tcanvas.addCommand(cmdExit);
  tcanvas.addCommand(cmdPlayPause);
  disp.setCurrent(tcanvas);
}
```

This shows the initialization. It's pretty much the same for any MIDlet. Please study it yourselves later.

UsingM3G Initialization

```
class TargetCanvas extends Canvas implements Runnable
  … // instance variable declarations elided
  public TargetCanvas(String m3gFile)
  {
    try
    {
      fileName = m3gFile;
      g3d = Graphics3D.getInstance();
      Load();
      w = getWidth();
      h = getHeight();
      cameraManip = new CameraManip(gWorld);
    }
    catch(IOException e)
    {
      System.out.println("loading fails:"+fileName);
      hasException = true;
    }
  }
```

Now it gets more interesting. We begin to use the M3G API. The canvas constructor loads the 3d data and creates a CameraManip object. This handles rotation of the scene-graph camera.

Note that TargetCanvas extends Canvas not GameCanvas because GameCanvas swallows key strokes from the number keys and we use the number keys to as controls. It implements the Runnable interface so we can run the update loop from a Thread.

Loading the 3D data

```
// class TargetCanvas
void Load() throws IOException {
  loadObjs = Loader.load(fileName);
  if (loadObjs==null)
    throw new RuntimeException("M3G file error");

  /* find the world node */
  for (int i=0; i<loadObjs.length; ++i) {
    if (loadObjs[i] instanceof World) {
     gWorld = (World)loadObjs[i];
     hasWorld = true;
     break;
    }
  }

  if (!hasWorld)
    throw new RuntimeException(
             "World node not found; incorrect m3g file?");
```

This method loads the m3g file using the M3G Loader and verifies that it contains a World node.

Loading the 3D Data (Cont.)

```
    meshController =
      (AnimationController)gWorld.find(meshControllerId);
    morphingMeshController =
       (AnimationController)gWorld.find(morphingMeshControll
erId);
    skinnedMeshController =
       (AnimationController)gWorld.find(skinnedMeshControlle
rId);

    /* Clean up after the loading process. */
    System.gc();
 }
```

After loading the file, the Load method looks for some scene-graph objects needed for the user controls to function.

TargetCanvas *run* method

```
public void run()
{
  for(;;) {
    long start, elapsed;
    start = System.currentTimeMillis();
    handleInput();
    repaint(); // Request paint()
    elapsed = System.currentTimeMillis() - start;
    // if (want to measure true frame rate)
    // Unfriendly to system!!
    //renderTime += (int)elapsed;
    // else {
    renderTime += (elapsed < 50) ? 50 : (int)elapsed;
    try {
      if (elapsed < 50) Thread.sleep(50-elapsed);
    } catch (InterruptedException e) { }
    //}
  }
}
```

This is the Thread's run method, the MIDlet's heart.

Basically we have an infinite loop. First it checks the input at which point the scene may be modified. Then it initiates rendering by requesting a repaint. After this the thread will sleep, provided rendering the frame did not take too long.

An alternative option is to just increment the render time and return to the top of the loop. This is very unfriendly to the system but is necessary in order to measure the true frame rate.

TargetCanvas *paint* method

```
synchronized protected void paint(Graphics g)
{
  if (loadObjs == null) return;
  g.setClip(0, 0, w, h);
  try
  {
    g3d.bindTarget(g);
    g3d.setViewport(0, 0, w, h);
    render();
  } finally { g3d.releaseTarget(); }

  g.setColor(0xffffffff);
  g.drawString("fps: " + fps, 2, 2, g.TOP|g.LEFT);
}
```

Here's our override of the Canvas paint method. We bind the graphics to the rendering target, set up the viewport and then render the 3D scene. Note that we make sure to always call releaseTarget(). After that we use the 2D api to draw the frame rate.

TargetCanvas *render* method

```
void render()
{
    if (isPlaying) {
      frameCount++;
      fps = (int)((1000*frameCount) / renderTime) ;
      /* update the scene */
      gWorld.animate((int)renderTime);
    }
    g3d.render(gWorld);
}
```

Here's the render method. The world is animated to update everything to the current renderTime, then it is rendered. So you can see that basic use of M3G is very simple. Most of what you've seen is standard for any MIDlet.

The MIDlet's input handling is standard MIDP code. It sets or clears state variables according to detected key presses. Since this is an M3G course, we won't spend time on basic MIDP stuff. Please study the source code yourselves. We'll study something uniquely 3D, manipulating the camera.

Camera Manipulation

```
/**
 * A camera manipulator. This class applies rotations to
 * a World’s activeCamera that make it rotate around the
 * prime axes passing through the World's origin.
 */
public class CameraManip
{
  public CameraManip(World world) { }

  public void buildCameraXform() { }

  public void
  baseRotate(float dAngleX, float dAngleY, float dAngleZ){ }

  public void
  rotate(float dAngleX, float dAngleY, float dAngleZ) { }

  public void setCameraXform() { }
}
```

The CameraManip class applies rotations to a World’s activeCamera that make it rotate around the prime axes passing through the World's origin.

The application maintains variables holding deltaX and deltaY rotations. Each time an arrow button is clicked a small value is added to or subtracted from these values. The input handler then calls cameraManip.rotate(deltaX, deltaY, 0).

Initializing CameraManip

```
public CameraManip(World world) {
     Transform world2Cam = new Transform();
     float[] matrix = new float[16];
     /* … class variable initialization elided */

     curCamera = world.getActiveCamera();
     if (curCamera != null) {
       curCamera.getTransformTo( world, world2Cam );
       world2Cam.get( matrix );
       distToTarget = (float)Math.sqrt( matrix[3]*matrix[3]
                                  + matrix[7]*matrix[7]
                                  + matrix[11]*matrix[11] );

       curCamera.getTransform( curOriginalXform );
       rotate( 0, 0, 0 );
       world2Cam = null;
     }
}
```

The constructor

- Initializes all the class variables
- Computes the distance between the camera and the world origin, using getTransformTo to obtain the transform from world to camera
- Saves the transform of the current camera.
- Calls rotate(0, 0, 0) to complete the initialization.
- Lastly it sets the world2Cam = null to make sure the transform will be garbage collected.

Rotating the Camera

```
public void rotate(float dAngleX, float dAngleY,
                   float dAngleZ) {
    if (curCamera == null) return;

    baseRotate( dAngleX, dAngleY, dAngleZ );
    Transform rotTrans = new Transform();

    rotTrans.postRotate( angleY, 0, 1, 0 );
    rotTrans.postRotate( angleX, 1, 0, 0 );

    float pos[] = { 0, 0, distToTarget, 1 };
    rotTrans.transform( pos );
    dx = pos[0];
    dy = pos[1];
    dz = pos[2] - distToTarget;

    buildCameraXform();
    setCameraXform();
    rotTrans = null;
}
```

•baseRotate sets class variables angleX, angleY and angleZ to the values of its parameters modulo 360.
•Creates a temporary transform. new Transforms are set to Identity
•Computes the delta from the original camera position to the desired position by rotating the point (0, 0, distToTarget) by the desired angle.
•Saves the delta in class variables, dx, dy, dz.
•Builds the new camera transform
•Sets the new camera transform
•Sets the temporary transform to null so it will be garbage collected.

Building the Camera Transform

```
public void buildCameraXform() {
    cameraXform.setIdentity();
    rotateXform.setIdentity();
    transXform.setIdentity();

    transXform.postTranslate( dx, dy, dz );

    // rotate about the x-axis then the y-axis
    rotateXform.postRotate( angleY, 0, 1, 0 );
    rotateXform.postRotate( angleX, 1, 0, 0 );

    cameraXform.postMultiply( transXform );
    cameraXform.postMultiply( rotateXform );
}

public void setCameraXform() {
    cameraXform.postMultiply( curOriginalXform );
    curCamera.setTransform( cameraXform );
}
```

We keep separate position and rotation transforms for the camera.
•Multiply the position transform (transXform) by the computed delta from the previous position
•Multiply the rotation transform (rotateXform) by the X and Y angles.
•Multiply the two transforms into the saved camera transform, transform first.
•*setCameraXform()* multiples the transform computed by buildCameraXform by the original camera transform and sets this matrix to the camera.

The effect is to move the camera to the position required by a rotation around the surface of the sphere with radius *distToTarget* and then orient the camera so it’s z-axis points toward the original origin.

Agenda

- Game Development Process
- Asset Creation
- Program Development
- MIDlet Structure
- A MIDlet Example
- Challenges in Mobile Game Development
- Publishing Your Content

Now we'll move on to look at the special challenges of developing games for mobile phone handsets.

Why Mobile Game Development is Difficult

- Application size severely limited
  - Download size limits
  - Small Heap memory
- Small screen
- Poor input devices
- Poor quality sound
- Slow system bus and memory system

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 Difficult

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

Memory

- Problems
  ①Small application/download size
  ②Small heap memory size
- Solutions
  - Compress data ①
  - Use single large file ①
  - Use separately downloadable levels ①
  - Limit contents ②
  - Get makers to increase memory ②

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

User-Friendly Operation

- Problems
  - Button layouts differ
  - Diagonal input may be impossible
  - Multiple simultaneous button presses not recognized
- Solutions
  - Plan carefully
  - Different difficulty levels
  - Same features on multiple buttons
  - Key customize feature

What is most important in the game is the operation, which functions as a communication line between the game and the player. Even within the same group of the mobile terminals, 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 terminals, game operation is one of the items that generates problems in the development. For example, diagonal input may have worked on the original mobile terminal whereas it may be unavailable on the mobile terminal to which the game is being ported. Also there are some cases where terminals fails to recognize more than one button being pressed at the same time.

We cannot provide you with overall solution; however, I would like to introduce you some examples on how we coped with these issues in our past contents.

1) Types of mobile terminals can be discerned to diversify the difficulty of the contents.

2) Let the player play in 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 a certain extent. Depending on the types of the game, I surmise there may be more efficient way to solve the problem. So 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

- Game Development Process
- Asset Creation
- Program Development
- MIDlet Structure
- A MIDlet Example
- Challenges in Mobile Game Development
- Publishing Your Content

Publishing Your Content

- Can try setting up own site but
  - it will be difficult for customers to find you
  - impossible to get paid
  - may be impossible to install MIDlets from own site
- Must use a carrier approved publisher
- Publishers often run own download sites but always with link from carrier's game menu.
- As with books, publishers help with distribution and marketing

This section describes the situation for the mobile phone market.

Don't even think about non-over-the-air distribution for mobile. It's not the way mobile works. Some carriers have MIDlet downloads from PC's disabled in their handsets.

Some carriers disable MIDlet downloads from anywhere but their own web sites. The villains may mostly be Japanese carriers. Perhaps the anti-monopoly authorities are more effective in other parts of the world. Vodafone KK does both of these things and, reportedly SIM-locks their handsets.

The bottom line is you must use a carrier approved publisher.

Publishing Your Content

- Typical end-user cost is $2 - $5.
- Sometimes a subscription model is used.
- Carrier provides billing services
  - Carriers in Japan take around 6%
  - Carriers in Europe have been known to demand as much as 40%! They drive away content providers.
- In some cases, only carrier approved games can be downloaded to phones
  - Enforced by handsets that only download applets OTA
  - Developers must have their handsets modified by the carrier

Common subscription model is a flat monthly fee for access to the publisher's entire game library.

Game add-ons are often used. For example, connection to game site to record high scores, chat with fellow players etc. Some sites even sell game upgrades (either for points won in the game or for cash) that will help you do better. A motorcycle racing game for example provides upgrades that make the bikes go faster.

Publishers

- 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

Other 3D Java Mobile APIs

Mascot Capsule Micro3D Family APIs

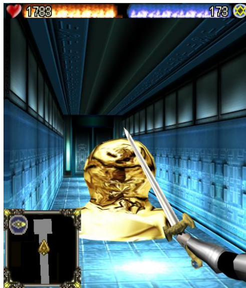
- Motorola iDEN, Sony Ericsson, Sprint, etc.)
  - com.mascotcapsule.micro3d.v3 (V3)
- Vodafone KK JSCL
  - com.j_phone.amuse.j3d (V2), com.jblend.graphics.j3d (V3)
- Vodafone Global
  - com.vodafone.amuse.j3d (V2)
- NTT Docomo (DoJa)
  - com.nttdocomo.opt.ui.j3d (DoJa2, DoJa 3) (V2, V3)
  - com.nttdocomo.ui.graphics3D (DoJa 4) (V4)

Mascot Capsule Micro3D Version Number

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 Mascot Capsule Micro3D Engine Mascot Capsule 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

DEEP LABYRINTH®
DELUXE EDITION



Copyright 2005, by Interactive Brains, Co., Ltd.

Just because it’s a really cool game…

Summary

- Use standard tools to create assets
- Basic M3G MIDlet is relatively easy
- Programming 3D Games for mobile is hard
- Need good relations with carriers and publishers to get your content distributed

Exporters

3ds max
- Simple built-in exporter since 7.0
- www.digi-element.com/Export184/
- www.mascotcapsule.com/M3G/
- www.m3gexporter.com

Maya
- www.mascotcapsule.com/M3G/
- www.m3gexport.com

Softimage|XSI
- www.mascotcapsule.com/M3G/

Cinema 4D
- www.c4d2m3g.com
  - Site appears to be defunct

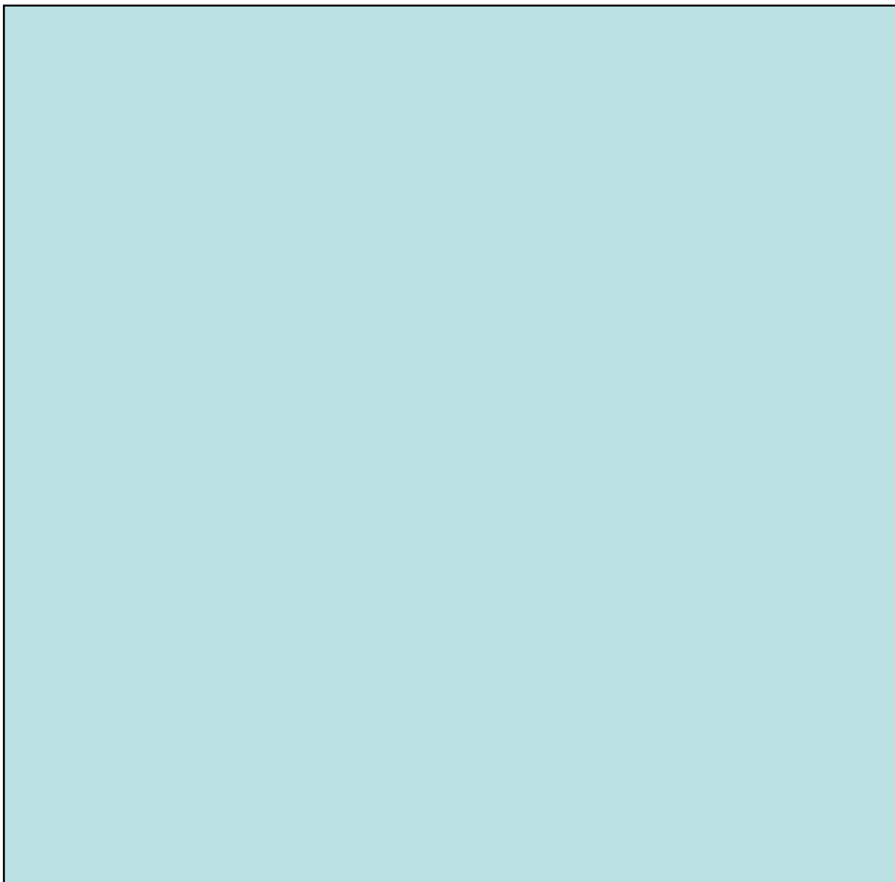
Lightwave
- www.mascotcapsule.com/M3G/

Blender
- http://www.nelson-games.de/bl2m3g/

m3gexport.com under Maya is NOT a typo.

SDKs

- Motorola iDEN J2ME SDK
  - idenphones.motorola.com/iden/developer/developer_tools.jsp
- Nokia Series 40, Series 60 & J2ME
  - www.forum.nokia.com/java
- Sony Ericsson
  - developer.sonyericsson.com/java
- Sprint Wireless Toolkit for Java
  - developer.sprintpcs.com
- Sun Wireless Toolkit
  - java.sun.com/products/j2mewtoolkit/download-2_2.html



SDKs

- VFX SDK (Vodafone Global)
  - via.vodafone.com/vodafone/via/Home.do
- VFX & WTKforJSCL (Vodafone KK)
  - developers.vodafone.jp/dp/tool_dl/java/emu.php

pVodafone global requires you become a partner of Via Vodafone. You have to submit a questionnaire before they will even talk to you. Very unfriendly.

Vodafone KK is a little more friendly. You just need to complete a simple registration before you can download the SDK. But the web page is in Japanese. There are 2 SDKs. VFX is Vodafone Global's SDK. WTKforJSCL has JSCL (**J**-Phone **S**pecific **C**lass **L**ibraries) instead of M3G. Both are based on Sun's Wireless Toolkit (WTK).

IDE’s for Java Mobile

- Eclipse Open Source IDE
  - www.eclipse.org
- JBuilder 2005 Developer
  - www.borland.com/jbuilder/developer/index.html
- Sun Java Studio Mobility
  - www.sun.com/software/products/jsmobility
- Comparison of IDE’s for J2ME
  - www.microjava.com/articles/J2ME_IDE_Comparison.pdf

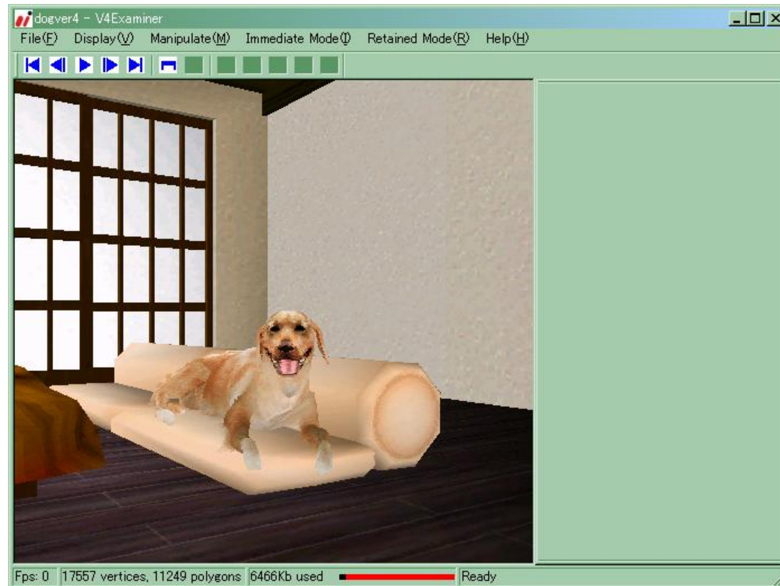
Although Eclipse is largely written in Java and has many java development tools, it is not clear at the time of writing that Eclipse has a specific set of tools for supporting J2ME.

Sun Java Studio Mobility is available at no cost by “simply register[ing] for a Sun online account”.

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/

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. Unfortunately this particular animation is too big to load into a real phone … today.

Thanks: HI Mascot Capsule Version 4 Development Team, Koichi Hatakeyama, Sean Ellis, JSR-184 Expert Group

Demonstrate dog animation

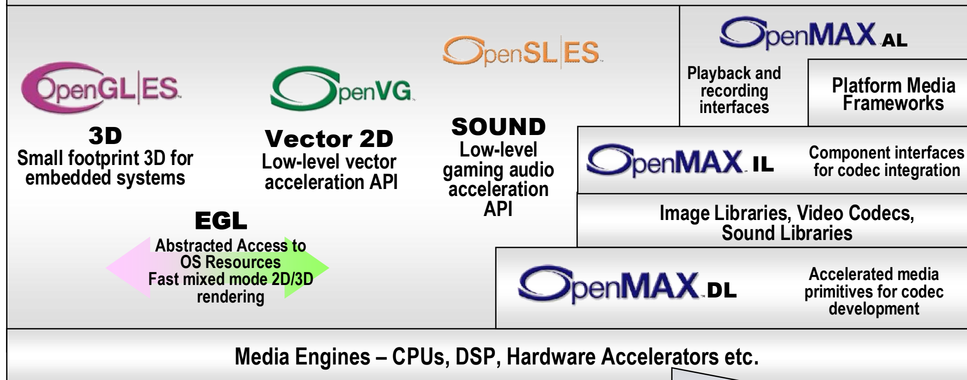
Closing & Summary

- We have covered
  - OpenGL ES
  - M3G

KHRONOS GROUP API palette

The Khronos API family provides a complete ROYALTY-FREE, cross-platform media acceleration platform

Applications or middleware libraries (JSR 184 engines, Flash players, media players etc.)

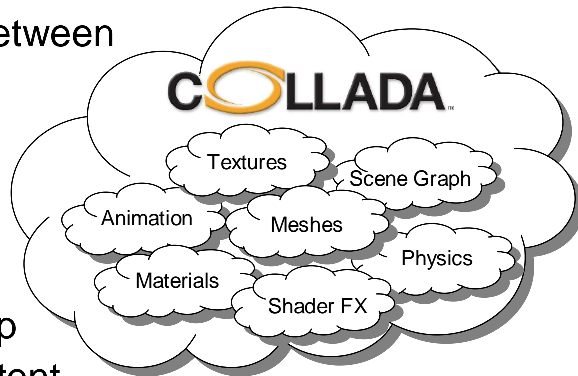


Media Engines – CPUs, DSP, Hardware Accelerators etc.

Khronos defines low-level, FOUNDATION-level APIs.
“Close to the hardware” abstraction provides portability AND flexibility

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



Shaders? Yes!

- OpenGL ES 2.0
  - subset of OpenGL 2.0, with very similar shading language
  - spec draft at SIGGRAPH 05, conformance tests summer 06, devices 08 (?)
- M3G 2.0
  - adds shaders and more to M3G 1.1
  - first Expert Group meeting June 06

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
  - work just starting

EGL evolution

- 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

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
- Standards for 2D vector graphics