JavaOne℠

Sun's 2002 Worldwide Java Developer Conference™

The J2SE™ 1.4 Release, OpenGL®, and New I/O

High-Performance 3D Graphics for the Desktop Client

Sven Goethel
President
Jausoft

Kenneth Russell
Member of Tech Staff
Sun Microsystems, Inc.

Session 3167

Presentation Goal

Show how to build high-performance 3D graphics applications using the Java™ programming language

Learning Objectives

As a result of this presentation, you will be able to:

- Understand how New I/O benefits high-throughput applications
- See how OpenGL®, for Java™ Technology takes advantage of New I/O
- Build effective 3D graphics applications using the Java™ 2 Platform, Standard Edition, 1.4 release and OpenGL, for Java Technology
- See several cool technology demonstrations

Speakers' Qualifications

Sven Goethel is the primary developer of "OpenGL, for Java Technology", a free software (LGPL) programming language binding for the Java platform, for the OpenGL 3D graphics API

Kenneth Russell is a member of the Java HotSpot™ VM group and a contributor to the New I/O and OpenGL, for Java Technology projects

Presentation Thesis

You can write portable, high-performance 3D applications and games **today** using the Java 2 Platform, Standard Edition, version 1.4 and OpenGL, for Java Technology

Presentation Agenda

- New I/O vs. Java Native Interface (JNI) in the J2SE 1.3 platform and earlier
- OpenGL overview
- OpenGL, for Java Technology
- Demos
- Performance Hints

Problem Statement

Pre-1.4 JNI technology provides limited interaction with data managed by the Java virtual machine (JVM™) implementation

J2SE 1.3 Platform Example #1

Sending float[] down to native code

```
float[] myArray = new float[10];
// ... fill in with data ...
sendDataToC(myArray);
// ... later ...
releaseCData(myArray);
```

J2SE 1.3 Platform Example #1

Native code:

```
float* ptr;

JNIEXPORT void JNICALL Java_MyClass_sendDataToC
(JNIEnv* env, jobject unused, jfloatArray arr)
{
  ptr=(*env)->GetFloatArrayElements(env,arr,NULL);
  C_function_requiring_float_ptr(ptr);
}
JNIEXPORT void JNICALL Java_MyClass_releaseCData
(JNIEnv* env, jobject unused, jfloatArray arr)
{
  (*env)->ReleaseFloatArrayElements(env, arr,
                                    ptr, JNI_ABORT);
}
```

J2SE 1.3 Platform Example #1

Native code:

```
float* ptr;

JNIEXPORT void JNICALL Java_MyClass_sendDataToC
(JNIEnv* env, jobject unused, jfloatArray arr)
{
  ptr=(*env)->GetFloatArrayElements(env,arr,NULL);
  C_function_requiring_float_ptr(ptr);
}
JNIEXPORT void JNICALL Java_MyClass_releaseCData
(JNIEnv* env, jobject unused, jfloatArray arr)
{
  (*env)->ReleaseFloatArrayElements(env, arr,
                                    ptr, JNI_ABORT);
}
```

J2SE 1.3 Platform Example #1

Native code:

```
float* ptr;

JNIEXPORT void JNICALL Java_MyClass_sendDataToC
(JNIEnv* env, jobject unused, jfloatArray arr)
{
  ptr=(*env)->GetFloatArrayElements(env,arr,NULL);
  C_function_requiring_float_ptr(ptr);
}
JNIEXPORT void JNICALL Java_MyClass_releaseCData
(JNIEnv* env, jobject unused, jfloatArray arr)
{
  (*env)->ReleaseFloatArrayElements(env, arr,
                                    ptr, JNI_ABORT);
}
```

J2SE 1.3 Platform Example #1 Discussion

- Upon call to GetFloatArrayElements, JVM must return float* which does not move in memory (unaffected by garbage collection, or GC)
- Can be implemented in one of two ways
  - Copy data out of garbage-collected heap into malloc'ed space
  - "Pin" object

J2SE 1.3 Platform Example #1 Discussion

Problems:

- Copying can impose unacceptable overhead for certain kinds of applications
- Depending on GC algorithm, pinning is difficult or impossible to implement

J2SE 1.3 Platform Example #2

Sending float[] down to native code (again)

```
float[] myArray = new float[10];
// ... fill in with data ...
sendDataToC(myArray);
```

J2SE 1.3 Platform Example #2

Native code:

```
JNIEXPORT void JNICALLJava_MyClass_sendDataToC
(JNIEnv* env, jobject unused, jfloatArray arr)
{
  float* ptr =
    (float*) (*env)->GetPrimitiveArrayCritical
      (env, arr, NULL);

  C_function_requiring_float_ptr(ptr);

  // C routine must be "done" with pointer by now
  (*env)->ReleasePrimitiveArrayCritical
      (env, arr, ptr, JNI_ABORT);
}
```

J2SE 1.3 Platform Example #2 Discussion

- Specification of "Get/Release Critical" routines imposes severe restrictions on what can occur between them
  - No returning between Get/Release
  - No calling other JNI functions
  - No blocking calls like **`select()`** or **`read()`**
  - Must not access pointer outside Get/Release

J2SE 1.3 Platform Example #2 Discussion

- Restrictions increase probability that “pinning” will occur
  - Java HotSpot™ VM implements by disabling GC between them
- However, restrictions usually result in having to copy data anyway, defeating the purpose

J2SE 1.3 Platform Example #2 Discussion

- Even if pinning is implemented, can not talk to outside memory directly
  - Video card RAM
  - Sound card buffers
- No way to make “fake array” wrapping arbitrary memory region

The Java™ 2 Platform, Standard Edition (J2SE™) 1.4 Release and New I/O

- **`java.nio`** provides solutions for the two fundamental problems
  - Passing JVM accessible data to C functions
  - Making data not managed by the JVM accessible to Java programming language code ("Java code")
- Does so with
  - High performance
  - Same safety as arrays

NIO Buffers

Classes which define APIs for accessing primitive data

- **`get()`**, **`put()`** methods

Direct buffers provide access to outside memory

New JNI routines allow Java/C programming language interaction

- Programs for the Java platform can operate on arbitrary data

NIO Example #1

Sending floating-point data to native code:

```
final int SIZEOF_FLOAT = 4;
FloatBuffer fbuf =
  ByteBuffer.allocateDirect(10 * SIZEOF_FLOAT).
    asFloatBuffer();
for (int i = 0; i < 10; i++) {
  fbuf.put(i, computeDatum(i));
}
sendDataToC(fbuf);
```

NIO Example #1

Native code:

```
JNIEXPORT void JNICALL Java_MyClass_sendDataToC
(JNIEnv* env, jobject unused, jobject buf)
{
  float* ptr = (float*)
   (*env)->GetDirectBufferAddress(env, buf);
  C_function_requiring_float_ptr(ptr);
}
```

NIO Example #1 Discussion

- Java code responsible for holding reference to direct buffer
  - Avoiding unexpected GC
- Otherwise, no restrictions on use of pointer in native code

NIO Example #2

Simple example: inverting video

```
ByteBuffer buf = getVideoCardMemory();
// Assuming R, G, B components
int size = 3 * width * height;
for (int i = 0; i < size; i++) {
  buf.put(i, (byte) (255 - (buf.get(i) & 0xFF)));
}
```

NIO Example #2

Native code:

```
JNIEXPORT jobject JNICALL
Java_MyClass_getVideoCardMemory
(JNIEnv* env, jobject unused)
{
  void* ptr   = Get_Video_Card_Memory();
  int width   = Get_Screen_Width();
  int height = Get_Screen_Height();
  int bytesPerPixel = Get_Screen_Depth();
  return (*env)->
    NewDirectByteBuffer(env, ptr,
                    width * height * bytesPerPixel);
}
```

NIO Summary

`GetDirectBufferAddress`

Outbound data transfer

`NewDirectByteBuffer`

Inbound data transfer

Individual element access via **`get/put`**

OpenGL

- 3D graphics library developed by Silicon Graphics in early 1990's
- Runs on every major operating system
- Hardware range from supercomputers to PCs
- Low-level, immediate-mode API
  - Can build higher-level, retained-mode APIs on top of it
  - Java 3D™ API does this (largely in native code)
  - SGI's Open Inventor and OpenGL Performer APIs

OpenGL

- Abstraction is a state machine
- Set up properties for geometric primitives
    - Color, texture, shininess, opacity
- Send geometric primitives (usually triangles) to graphics card

OpenGL

Trivial example:

```
glBegin(GL_TRIANGLES);
glVertex3f(1.0f, 0.0f, 0.0f);
glVertex3f(0.0f, 1.0f, 0.0f);
glVertex3f(1.0f, 1.0f, 0.0f);
glEnd();
```

OpenGL

- Most flexible way of drawing geometry with OpenGL is via vertex arrays
  - Set up region of memory containing 3D points
  - Transfer connectivity information to card
  - Allows application to modify geometry without having to make on the order of one function call per triangle

OpenGL

Vertex array example:

```
// Set up data buffer
// Two adjacent triangles forming a square
GLfloat* coords   = { 1.0f, 0.0f, 0.0f,
                      1.0f, 1.0f, 0.0f,
                      0.0f, 1.0f, 0.0f,
                      0.0f, 0.0f, 0.0f };
GLint*   elements = { 0, 1, 2, 1, 2, 3 };
glEnableClientState(GL_VERTEX_ARRAY);
// Size of vertices (2, 3, 4), type of vertices,
// stride between vertices (unused), ptr to data
glVertexPointer(3, GL_FLOAT, 0, coords);
// Geometry type, num primitives, indices' data type
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT,
               elements);
```

OpenGL

- OpenGL semantics are strict and vertex arrays are not as efficient as desired
  - Application can not continue until glDrawElements call is complete
- NVidia® and ATI® have devised extensions to allow parallel processing of vertex arrays
  - Allocate memory on AGP bus or on card itself
- More on this later

JavaOne

OpenGL®, for Java™ Technology

- Jausoft's programming language binding, for the Java platform, for OpenGL
  - Licensed under the "Lesser GNU Public License" (LGPL)
  - Provides Java technology-based APIs for accessing all OpenGL routines
  - Default implementation for many operating systems

OpenGL, for Java Technology

- Highly portable
  - Works on development kits from Sun™, IBM, Apple
  - Java™ Platform releases 1.1.x through 1.4
  - Works on Netscape™ and Internet Explorer VMs
  - Binaries available for GNU/Linux, Mac OS, Solaris™, Windows
  - Should work on any Java 2 Platform + OpenGL + Unix® + X11 environment
    - QNX + X11 + OpenGL + J2ME platform
    - OSGI/Automotive Systems

OpenGL, for Java Technology

How it works

- C2J program parses C header files (gl.h, glu.h)
  - Using current Mesa3D OpenGL compatible headers
  - C2J is LGPL and part of OpenGL, for Java technology
- Generates JNI based code and Java platform interfaces
- A few routines are coded by hand, but most are autogenerated
- Binding for OpenGL 1.3 plus extensions (983 functions) all based upon the same well tested primitives in the C2J compiler

OpenGL, for Java Technology and New I/O

- OpenGL, for Java Technology 2.8 includes built-in **`java.nio`** support
- Vertex arrays, textures, other large objects can be stored in **`java.nio`** direct buffers
- Allows fast, robust, portable 3D applications to be written with no native code in the application

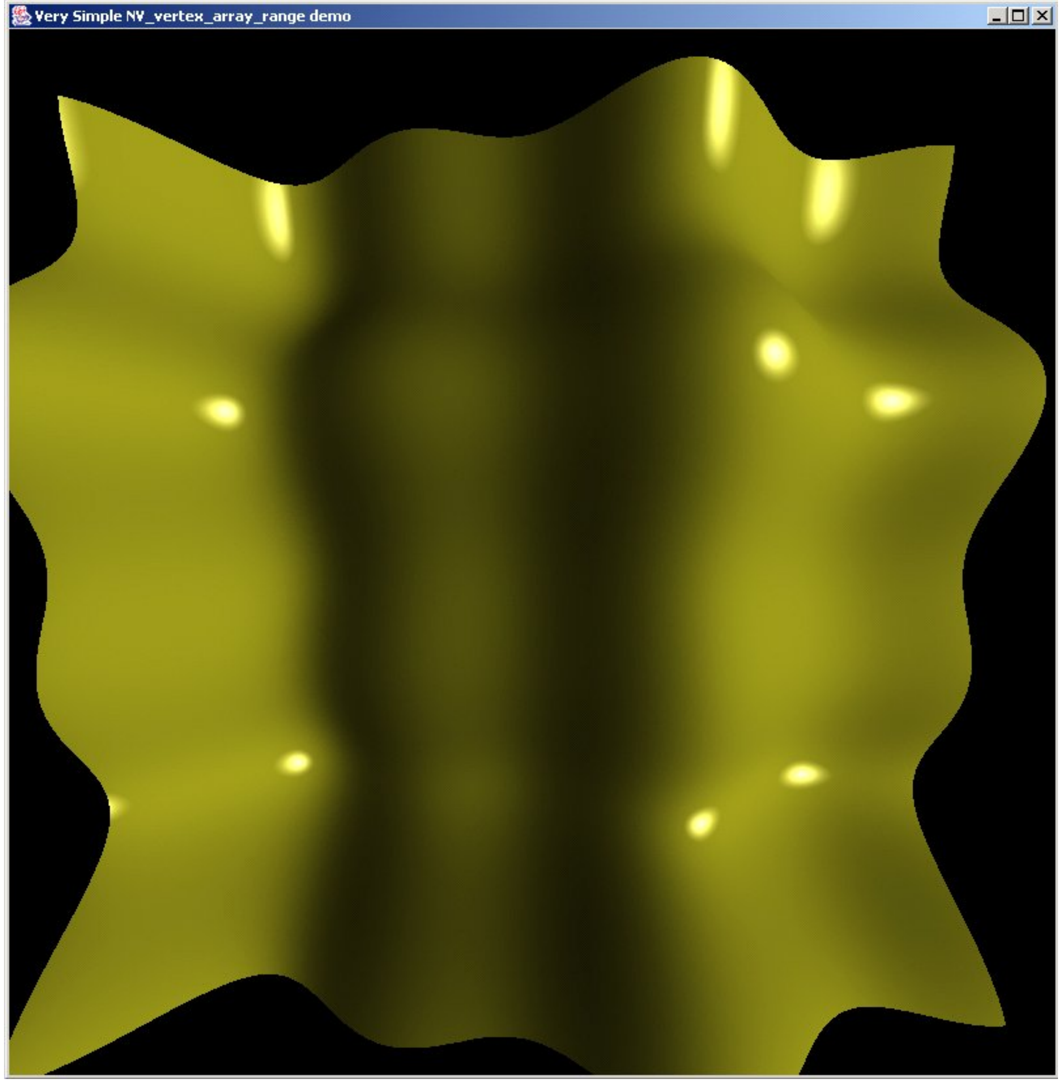
Unique Features

- Easy-to-use, multithreaded user API
  - Animations or still frames
  - Textured objects
  - Screen snapshots
- Provides access to all vendor extensions with no additional native code
  - Can test for and use optimized routines; i.e., NVidia vertex array range extension
- Compatible with full-screen support in the J2SE 1.4 release
  - Sun's Java™ Development Kit (JDK™): **`java -Dsun.java2d.noddraw=true`**

JavaOne

Demo

NVidia vertex_array_range Demo



JavaOne

NVidia vertex_array_range Demo

- C++ version illustrates 2x speedups with this extension
- Runs at 30 Hz on PIII, 700 MHz, GeForce 256
- Amounts to different version of malloc()
  - Minimal change for C/C++ programs

NVidia vertex_array_range Demo

- Ported to the Java platform using JDK 1.4 software; OpenGL, for Java Technology 2.8; Java HotSpot™ Client VM
- Frame rate of port is 27 Hz
  - 90% of optimized C++ speed
- Java programming language code now able to take advantage of leading-edge 3D hardware
- On faster PCs, Java technology version is not as competitive (65–75% of C speed)
  - More optimizations to be done in compilers (e.g., Java HotSpot VM)

Arkanae Demo

JavaOne

Arkanae Demo

Free software 3D fantasy/adventure game

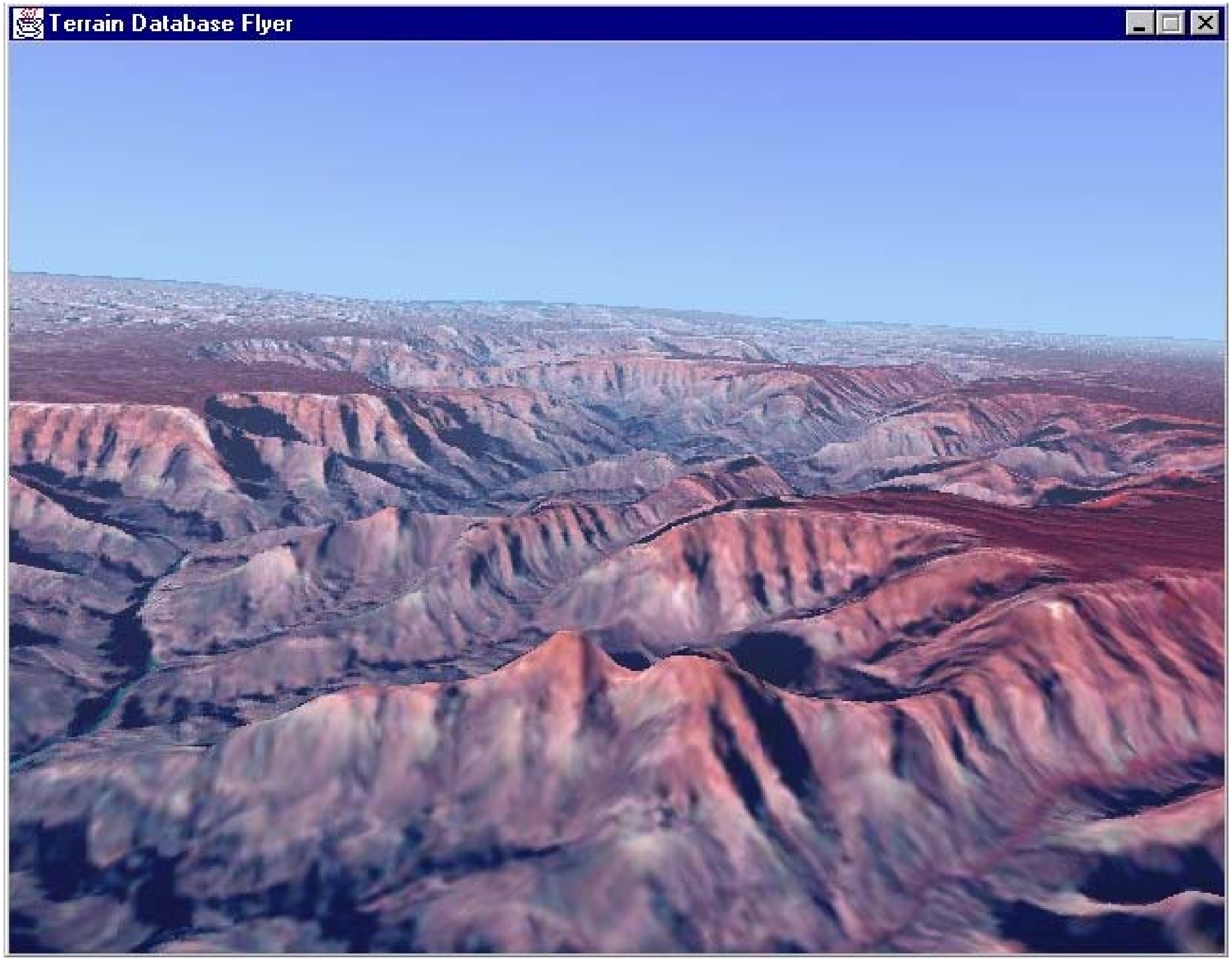
Core team: Bertrand and Jean-Baptiste Lamy

Runs fast; quite polished

Application is built on top of OpenGL, for Java Technology and itself contains no native code

Get it at `http://arkanae.tuxfamily.org/`

Grand Canyon Demo



Grand Canyon Demo

Introduced at the 2001 JavaOne[SM] conference

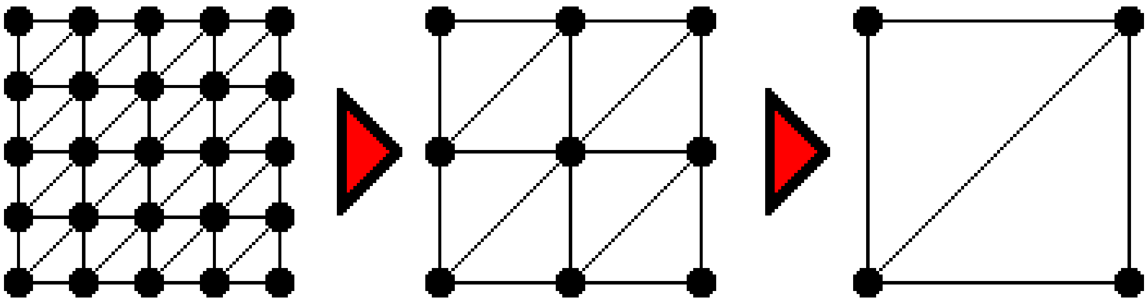
300 MB data set visualized in real time with Sun JDK 1.4; OpenGL, for Java Technology 2.8; and Java HotSpot Client VM

Grand Canyon Demo

- Multiresolution algorithm
  - More detail for terrain closer to camera
  - Data set divided into square tiles
    - 513x513 vertices; 15x13 tiles
  - Highest resolution memory-mapped in using **`java.nio`**
    - 100 MB of geometric data
- Every vertex, every frame is processed by Java programming language code (“Java code”)

Grand Canyon Demo

To render tile at lower resolution, recursively drop every other sample



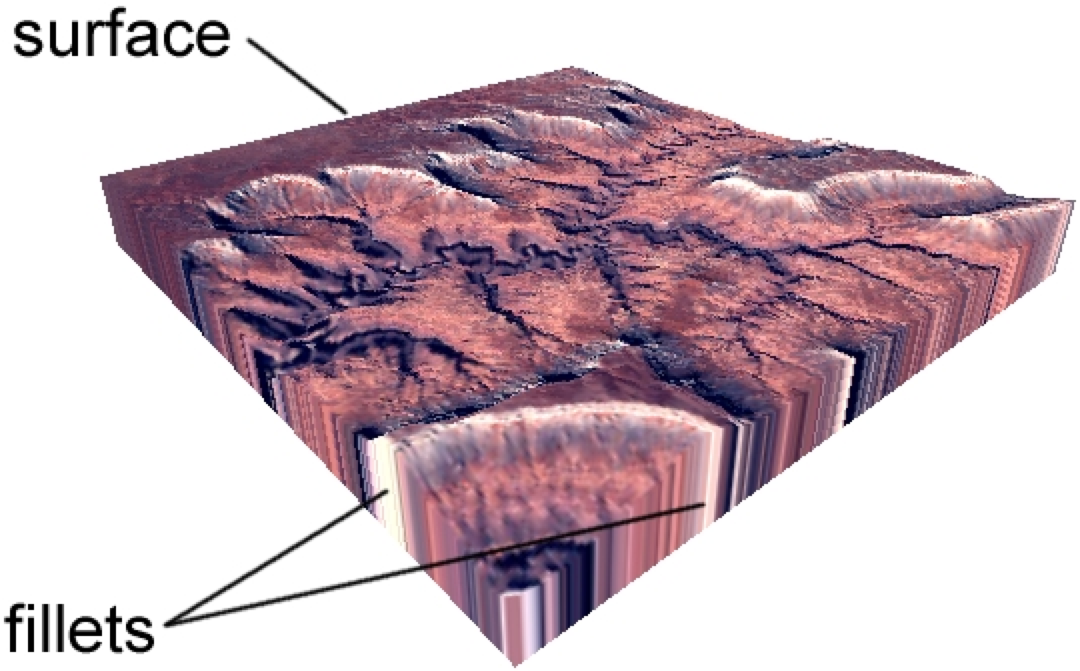
Done every frame for every visible tile by Java code

- Output buffer is a `java.nio` direct FloatBuffer

View culling, collision detection

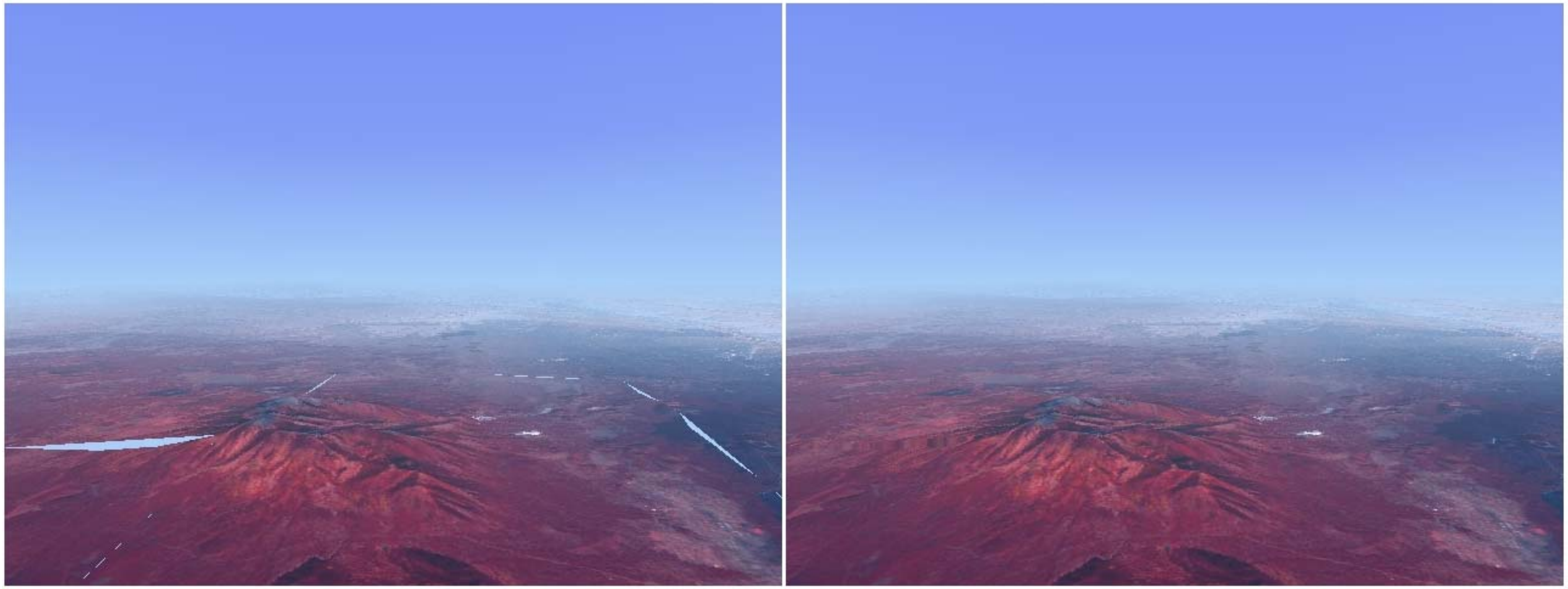
Grand Canyon Demo

Cracks between tiles at differing resolutions are patched with “fillets”



Grand Canyon Demo

- Cracks relatively minor feature of landscape
  - Fillets allow independent processing of tiles and faster inner loops



Grand Canyon Demo

- Lower-resolution textures are generated offline
- Appropriate resolution memory-mapped in using **`java.nio`**
- Textures paged in by background thread
  - Advantageous in multi-CPU systems
  - Very easy to implement using Java technology
    - **`synchronized`** keyword
    - Thread and Collections APIs
- Memory-mapped texture data passed directly down into OpenGL, for Java Technology

Grand Canyon Demo

- Run-time statistics:
  - Roughly 90,000 triangles per frame at 45 fps
  - 4.0 million tris/second; up to 5.0 in some areas

Pup Demo

Pup Demo

- Developed by the Synthetic Characters Group at The Media Lab, MIT
- Showcases research in behavior systems for intelligent, interactive 3D animated characters

Pup Demo

- Sophisticated behavior system research and learning algorithms
  - Inspired by ethology (study of animal behavior)
- Run-time motion blending and animation
  - Walk left/straight/right
  - Sit happy/sad
- All done in the Java programming language

Pup Demo

- Graphics system
  - Skinning on complex model
    - 53 joints, > 2000 vertices
  - Custom vertex shaders
    - Cartoon shading
    - Real-time shadows
  - Originally implemented in C++ using Microsoft's Direct3D
    - Hooked into the Java platform with large quantities of native code

Pup Demo

- Graphics system ported to JDK 1.4 software and OpenGL, for Java Technology 2.8
  - Minimal scene graph written to wrap OpenGL, for Java Technology
  - Skinning implemented in Java programming language
  - Cartoon shading and shadows implemented using OpenGL techniques
  - Eliminates nearly all native code in application
    - Remaining: game controller…

Pup Demo

Results

- Java programming language port of graphics system is 86% of the speed of optimized C++
- Can be debugged with no performance penalty
  - Full-speed debugging in J2SE 1.4 release
- Up to 11% faster than C++ debug build

Performance Hints

When using direct buffers in conjunction with JNI, *always* set the byte order

```
ByteBuffer.order(ByteOrder.nativeOrder())
```

This is a correctness issue

Very easy to forget

Write utility class for allocating direct buffers and make this call before returning them

JavaOne

Performance Hints

- Use absolute **`put(index, data)`** and **`get(index)`** methods in inner loops instead of **`put(data)`** and **`get()`**
  - Typically have a loop index available anyway
  - Non-absolute versions maintain internal indices
    - Duplicated work
  - Absolute versions generate code very similar to array indexing (i.e., fast)

Performance Hints

- In inner loops, access only locals instead of data members
  - Sometimes tricky to see with presence of inner classes

Performance Hints

```
class MyClass {
  FloatBuffer myBuf;
  // ...
  void doComputation() {
    for (int i = 0; i < size; i++) {
      // Avoid
      myBuf.put(i, computeNextDatum());
    }
  }
}
```


JavaOne

Performance Hints

```
class MyClass {
  FloatBuffer myBuf;
  // ...
  void doComputation() {
    // Better
    FloatBuffer buf = myBuf;
    for (int i = 0; i < size; i++) {
      buf.put(i, computeNextDatum());
    }
  }
}
```

Performance Hints

- Avoid mixing use of direct and non-direct buffers in applications
  - Compilers for the Java HotSpot VM currently will not be able to inline accessors well
    - Other DKs for Java technology may do better
  - Problem we are taking very seriously and will address in future release

Summary

- J2SE™ 1.4 release; OpenGL, for Java™ Technology; and New I/O reach previously unattainable performance levels for the Java programming language
- Can write high-performance 3D applications in the Java programming language today
  - Portability, safety, and ease-of-development of Java technology
- Already fast; future releases will only be faster

Conclusion

Start writing 3D applications and games in the Java programming language!

Q&A

JavaOne℠

Sun's 2002 Worldwide Java Developer Conference™

BEYOND
BOUNDARIES