Introduction to Game Programming

Steven Osman
sosman@cs.cmu.edu
Introduction to
Game Programming

Introductory stuff
Look at a game console: PS2
Some Techniques (Cheats?)
What is a Game?

Half-Life 2, Valve
Designing a Game

Computer Science
Art
Music
Business
Marketing
Designing a Game

Music
Art
Computer Science
Business
Marketing
History
Geography
Psychology
Sociology
Physics
Literature
Education
Writing
Civics/Politics
...Just to name a few
Designing a Game

Find out more from an industry veteran @

Professor Jesse Schell’s class:
Game Design

(Entertainment Technology Center)
The Game Engine

Graphics & Animation
Physics
Controller Interaction
AI Primitives
Sound
Networking
Scripting system
The Game Logic

Game rules
Non-Player Characters (NPC) AI
Interface, etc.

Often (but not necessarily) implemented in scripting language
Magic Formula

1. Read Player Input
2. Update World State
3. Apply Game Rules
4. Draw Frame
5. Repeat
Game Programming is *hard*

- Players want complex graphics (why?)
- Game must run fast (30fps+)
- AI isn’t exactly trivial
- We want networking but no latency
- Physics is already hard. Now do it in real-time

- ... And do it all in time for Christmas
To most, this is the PS2
To technophiles, this is a PS2
To us, this is the PS2

Source: http://playstation2-linux.com/projects/p2lsd
Emotion Engine Core “EE Core”

Source: http://playstation2-linux.com/projects/p2lsd
Emotion Engine Core “EE Core”

Runs at about 300 megahertz
MIPS I & II, subset of MIPS III & IV
Math coprocessor
SIMD Instructions

16k instruction cache
  8k data cache
16k scratch pad
SISD Instructions
Single Instruction, Single Data

Modified from: http://arstechnica.com/articles/paedia/cpu/ps2vspc.ars/5
MIMD
Multiple Instruction, Multiple Data

Source: http://arstechnica.com/articles/paedia/cpu/ps2vspc.ars/5
SIMD
Single Instruction, Multiple Data

Source: http://arstechnica.com/articles/paedia/cpu/ps2vspc.ars/5
Which is Better?

Sure, 4 independent instruction streams (MIMD) would be nice, but it would require more memory.

But media applications do not require instruction-level parallelism, so SIMD is fine.
Sneak Peek

The safe money says next generation from Sony will be highly parallel (=MIMD)
There’s a good chance that this may include parallel SIMD instructions

Now go ask your Architecture professor what that’s even called! (I like MIM²D)
PS2 SIMD Support

PS2 has lots of SIMD support:
Parallel instructions on core CPU
• 2x64-bits, 4x32-bits, 8x16-bits or 16x8-bits ➔ Homework 4 example
Vector Unit 0 through micro & macro mode
Vector Unit 1
• Both VU’s do 4x32-bit floating point
SISD Example:
Vector/Matrix Multiplication

\[
\begin{bmatrix}
  a & b & c & d \\
  e & f & g & h \\
  i & j & k & l \\
  m & n & o & p \\
\end{bmatrix}
\begin{bmatrix}
  s \\
  t \\
  u \\
  v \\
\end{bmatrix}
= \begin{bmatrix}
  x \\
  y \\
  z \\
  w \\
\end{bmatrix}
\]

\[
x = a*s + b*t + c*u + d*v
\]
\[
y = e*s + f*t + g*u + h*v
\]
\[
z = i*s + j*t + k*u + l*v
\]
\[
w = m*s + n*t + o*u + p*v
\]

16 multiplications, 12 additions.
Additions can be eliminated with MADD.
SIMD Example:
Vector/Matrix Multiplication

\[
\begin{bmatrix}
  a & b & c & d \\
  e & f & g & h \\
  i & j & k & l \\
  m & n & o & p \\
\end{bmatrix}
\begin{bmatrix}
  s \\
  t \\
  u \\
  v \\
\end{bmatrix}
= \begin{bmatrix}
  x \\
  y \\
  z \\
  w \\
\end{bmatrix}
\]

First, load columns into registers:

- VF03 = \{c, g, k, o\}
- VF01 = \{a, e, i, m\}
- VF04 = \{d, h, l, p\}
- VF02 = \{b, f, j, n\}
- VF05 = \{s, t, u, v\}
**SIMD Example:**

Vector/Matrix Multiplication

VF01 = \{a, e, i, m\}  
VF02 = \{b, f, j, n\}  
VF03 = \{c, g, k, o\}  
VF04 = \{d, h, l, p\}  
VF05 = \{s, t, u, v\}

MUL ACC, VF01, VF05[x]  // acc = \{a*s, e*s, i*s, m*s\}
MADD ACC, VF02, VF05[y]  // acc += \{b*t, f*t, j*t, n*t\}
MADD ACC, VF03, VF05[z]  // acc += \{c*u, g*u, k*u, o*u\}
MADD VF06, VF04, VF05[w]  // VF06 = acc + \{d*v, h*v, l*v, p*v\}

Only 4 instructions! (compared to 16 or 28 instructions)
Matrix/Matrix multiplication is 4 dot products

Compare:

16 ($=4 \times 4$) instructions

to

64 ($=4 \times 16$) assuming MADD

or

112 ($=4 \times 28$) instructions, without MADD!
Vector Units (VU0 & VU1)

Source: http://playstation2-linux.com/projects/p2lsd
Vector Units (VU0 & VU1)

VU0 – 4k data, 4k code
• Can be used in “Micro” or “Macro” mode

VU1 – 16k data, 16k code
• Micro mode only
• Connected directly to the GS
• Can do clipping & a few more instructions
Vector Unit = Vertex Shader?

Absolutely not.

The vector units do much, much more than a vertex shader!

At the most trivial level, a vertex shader (not sure about the absolute latest) cannot create geometry.
What are they for?

One approach
VU0: Animation, Physics, AI, Skinning, etc…
VU1: Transformation, clipping & lighting

Another approach
VU0: Transformation, lighting
VU1: Transformation, lighting
* I don’t think anyone ever uses it this way
Graphics Synthesizer (GS)

Source: http://playstation2-linux.com/projects/p2lsd
Graphics Synthesizer

The “graphics chip” of the PS2
Not a very smart chip!
… but a very fast one.

Supports:
• Alpha blending
• Z Testing
• Bi- and tri-linear filtering
Graphics Synthesizer (GS)

Per-second statistics:
- 2.4 gigapixel fill rate
- 150 million points
- 50 million sprites
- 75 million untextured triangles
- 37.5 million textured triangles
I/O Processor (IOP)

Source: http://playstation2-linux.com/projects/p2lsd
I/O Processor (IOP)

Built from a PlayStation!
Gives backward compatibility
IOP used to access the Sound Processing Unit (SPU2), controllers, CD & Hard Drive, USB and FireWire port
IOP has 2MB memory
SPU has 2MB memory
Image Processing Unit (IPU)

Source: http://playstation2-linux.com/projects/p2lsd
Image Processing Unit (IPU)

MPEG 2 decoding support
At a high level, hand over encoded data, retrieve results when they’re ready
The Job of a PS2 Programmer

Keep the system busy!
Have all processors running
• Double buffer everything
• Reduce waiting on others
→ Stream textures for next model while processing current model
• Reduce data dependency stalls
• Pair instructions where possible
The Job of a PS2 Programmer

Avoid stalling on memory access
  • Use the scratch pad
Avoid cache misses as much as possible
  • Use the scratch pad
  • Code & Data locality
  • Avoid C++ overdose
  • Prefetch data into cache
Frame Rate Drop

Source: http://www.research.scea.com/

heavily EE bound, optimize EE first!
Let’s draw a triangle

Ultimate goal is to prepare a “GIF Packet”:

GIF tag
→ Description of data to follow
Register data (already transformed & lit)
→ XYZ Coordinates
→ RGB Colors
→ UV or ST Texture coordinates
Sample GIF Packet (Parsed)
Let’s draw a triangle

Step 1 (EE): Do animation to update object, camera & light matrices

Step 2 (EE): Cull objects that cannot be seen

Step 3 (EE): Send camera, lights and untransformed objects to VU, texture to GS

So far, just like OpenGL, right?
Let’s draw a triangle

Step 4 (VU1): Transform vertices, do “trivial clipping”
Step 5 (VU1): Non-trivial clipping – chop up triangles. More triangles or triangle fan.
Step 6 (VU1): Compute lighting
Step 7 (VU1): Assemble GIF packet
Step 8 (VU1): Kick data to GS
Case Study 1: Shadows
Stencil Buffer

Stencil Buffer is sort of like the Z-Buffer: Additional bit plane(s) that can determine whether a pixel is drawn or not.
OpenGL Stencil Buffer Support

```c
glutInitDisplayString("stencil>=1 rgb depth double");
glutCreateWindow("stencil buffer example");
...

// clear to zero

glClearStencil(0);

// clear to zero

glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
...

// enable stencil test

glEnable(GL_STENCIL_TEST);

// disable stencil test

glDisable(GL_STENCIL_TEST);

Tests: never, always, =, !=, <, >, <=, >= some value
```

Source: http://developer.nvidia.com
OpenGL Stencil Buffer Support

To use the stencil buffer:

```c
glStencilFunc(GL_EQUAL, // comparison function
  0x1, // reference value
  0xff); // comparison mask
```

To update the stencil buffer:

```c
glStencilOp(GL_KEEP, // stencil fail
             GL_DECR, // stencil pass, depth fail
             GL_INCR); // stencil pass, depth pass
```

`glStencilMask(0xff);` // Which bits to update

Source: http://developer.nvidia.com
Case Study 2: Normal Mapping

What if we could read in normals from a texture?

Source: http://playstation2-linux.com/download/p2lsd/ps2_normalmapping.pdf
Normal Mapping

(a) Wireframe  
(b) Gouraud shading

(c) DOT3 diffuse  
(d) DOT3 diffuse + specular

Source: http://playstation2-linux.com/download/p2lsd/ps2_normalmapping.pdf
Normal Mapping

These normals don’t need to be simple interpolations of the vertices – we can add the appearance of detail.

With a “pixel shader,” it’s fairly easy – at each pixel, read in the normal from the map.

Can it be done without one?
Normal Mapping

High-level Overview:

Instead of a texture being color values, let it be normal values.
Instead of vertex colors being colors of edges, let them be light direction from that edge.
Normal Mapping

Now when we render the scene we get:
\[ v.r \times t.r, v.g \times t.g, v.b \times t.b \]
\( (v=\text{vertex color, } t=\text{texture color}) \)
But since \( v=l \), and \( t=n \ldots \)

We just need to add the \( r, g, b \) for \( n \dot l \)!
Just multiply the resulting colors by the light intensity, \( l \)
Bump Mapping

Take a height-field
Compute its gradient
This gives you “deltas” to add to your current normals
Bump Mapping

Top images from: http://www.3dxperience.com/html/resources.html
The future?
Case Study 3: Simple Motion Detection

Image A={r_{a1}, g_{a1}, b_{a1}, r_{a2}, g_{a2}, b_{a2}, ...}
Image B={r_{b1}, g_{b1}, b_{b1}, r_{b2}, g_{b2}, b_{b2}, ...}

PixelChangeBitmask={C_1, C_2, ...}
Where C_i=changed(r_{ai}, r_{bi}) ||
      changed(g_{ai}, g_{bi}) ||
      changed(b_{ai}, b_{bi})
Simple Motion Detection

Trivial_Changed(a, b) {
    bigger=max(a,b);
    smaller=min(a,b);

    return a-b > delta;
    // or
    return a-b > delta && a * fraction >= b;
}