Tim Sweeney
CEO, Founder
Epic Games
tim@epicgames.com

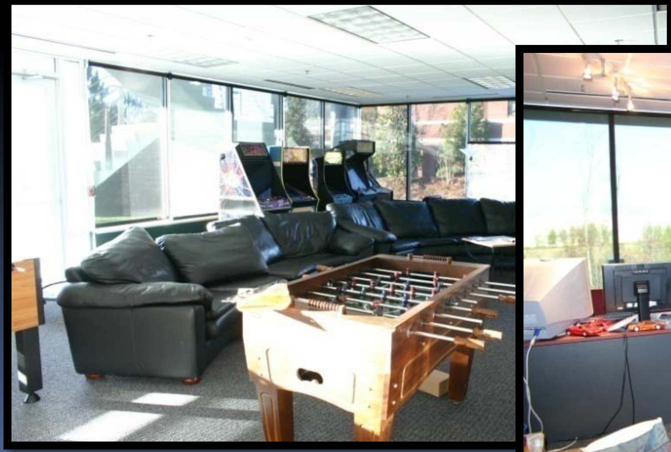
THE END OF THE GPU ROADMAP

Background:

Epic Games

Background: Epic Games

- Independent game developer
- Located in Raleigh, North Carolina, USA
- Founded in 1991
- Over 30 games released
  - Gears of War
  - Unreal series
- Unreal Engine 3 is used by 100's of games



History:
Unreal Engine

Unreal Engine 1
1996-1999

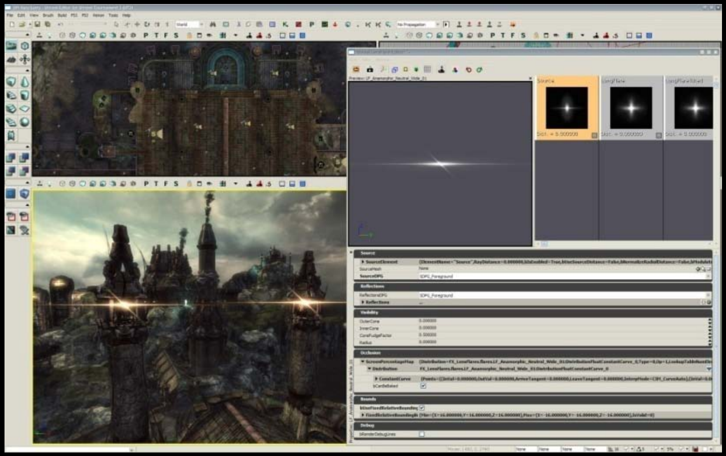
- First modern game engine
  - Object-oriented
  - Real-time, visual toolset
  - Scripting language
- Last major software renderer
  - Software texture mapping
  - Colored lighting, shadowing
  - Volumetric lighting & fog
  - Pixel-accurate culling
- 25 games shipped

Unreal Engine 2
2000-2005

- PlayStation 2, Xbox, PC
- DirectX 7 graphics
- Single-threaded
- 40 games shipped

Unreal Engine 3
2006-2012

- PlayStation 3, Xbox 360, PC
- DirectX 9 graphics
  - Pixel shaders
  - Advanced lighting & shadowing
- Multithreading (6 threads)
- Advanced physics
- More visual tools
  - Game Scripting
  - Materials
  - Animation
  - Cinematics…
- 150 games in development



Unreal Engine 3 Games

Mass Effect (BioWare)

Army of Two (Electronic Arts)

Undertow (Chair Entertainment)

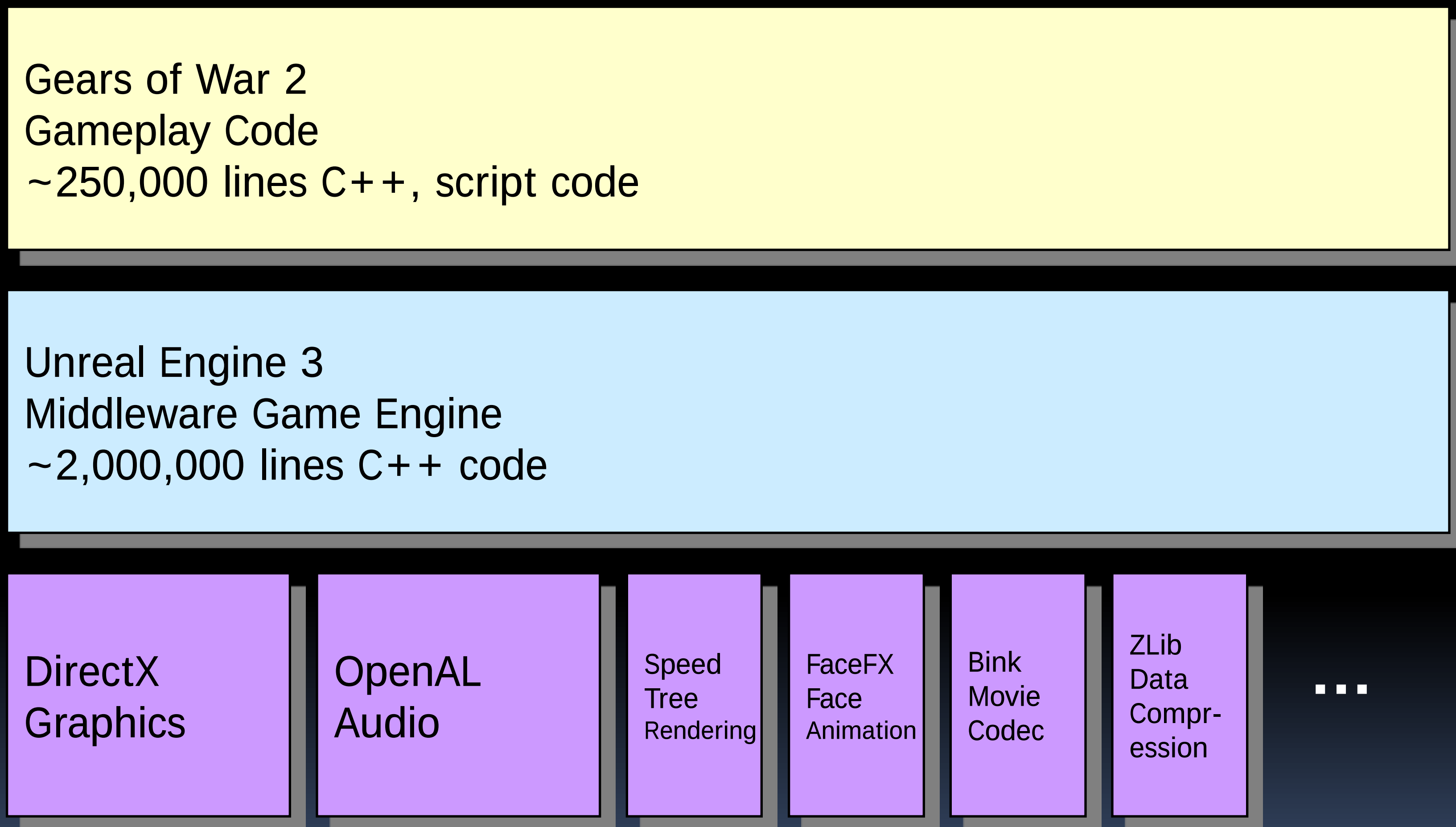
BioShock (2K Games)

Game Development: 2009

Gears of War 2: Project Overview

- Project Resources
  - 15 programmers
  - 45 artists
  - 2-year schedule
  - $12M development budget
- Software Dependencies
  - 1 middleware game engine
  - ~20 middleware libraries
  - Platform libraries

Gears of War 2: Software Dependencies



Hardware:

History

Computing History

1985 Intel 80386: Scalar, in-order CPU

1989 Intel 80486: Caches!

1993 Pentium: Superscalar execution

1995 Pentium Pro: Out-of-order execution

1999 Pentium 3: Vector floating-point

2003 AMD Opteron: Multi-core

2006 PlayStation 3, Xbox 360: “Many-core”

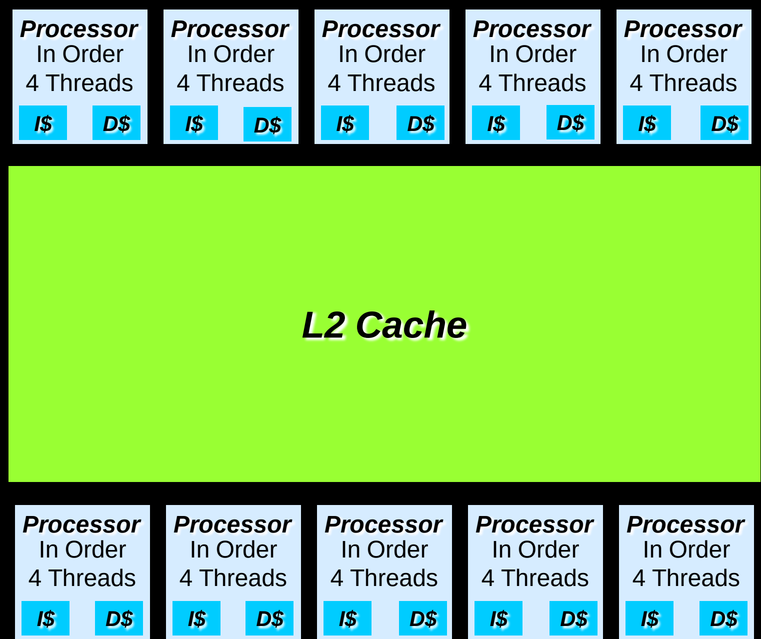
…and we’re back to in-order execution

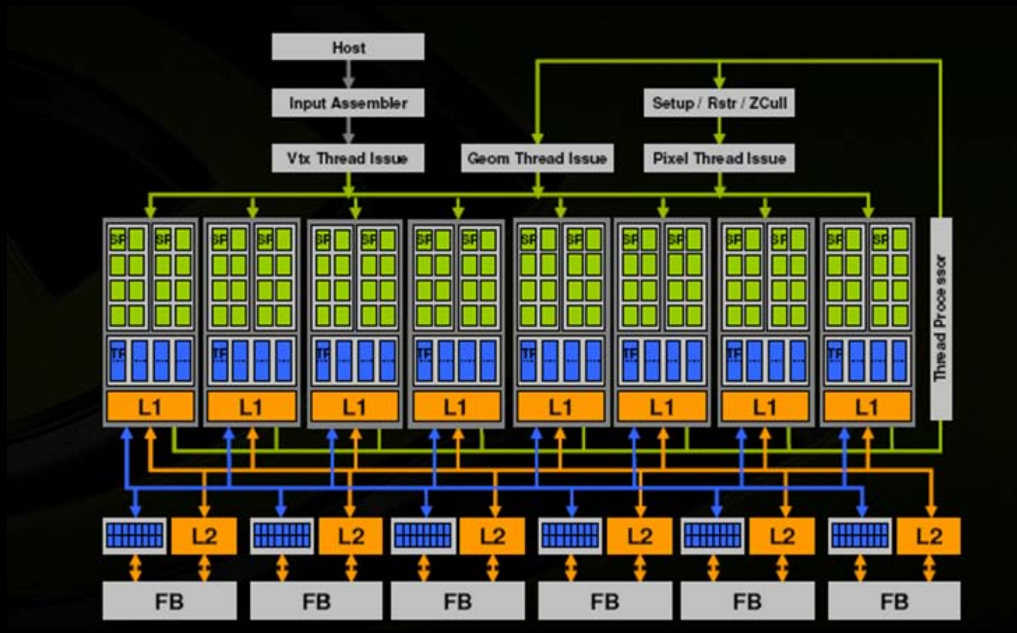
Graphics History

1984	3D workstation (SGI)
1997	GPU (3dfx)
2002	DirectX9, Pixel shaders (ATI)
2006	GPU with full programming language (NVIDIA GeForce 8)
2009?	x86 CPU/GPU Hybrid (Intel Larrabee)

Hardware:
2012-2020

Hardware: 2012-2020





Intel Larrabee

- x86 CPU-GPU Hybrid
- C/C++ Compiler
- DirectX/OpenGL
- Many-core, vector architecture
- Teraflop-class performance

NVIDIA GeForce 8

- General Purpose GPU
- CUDA “C” Compiler
- DirectX/OpenGL
- Many-core, vector architecture
- Teraflop-class performance

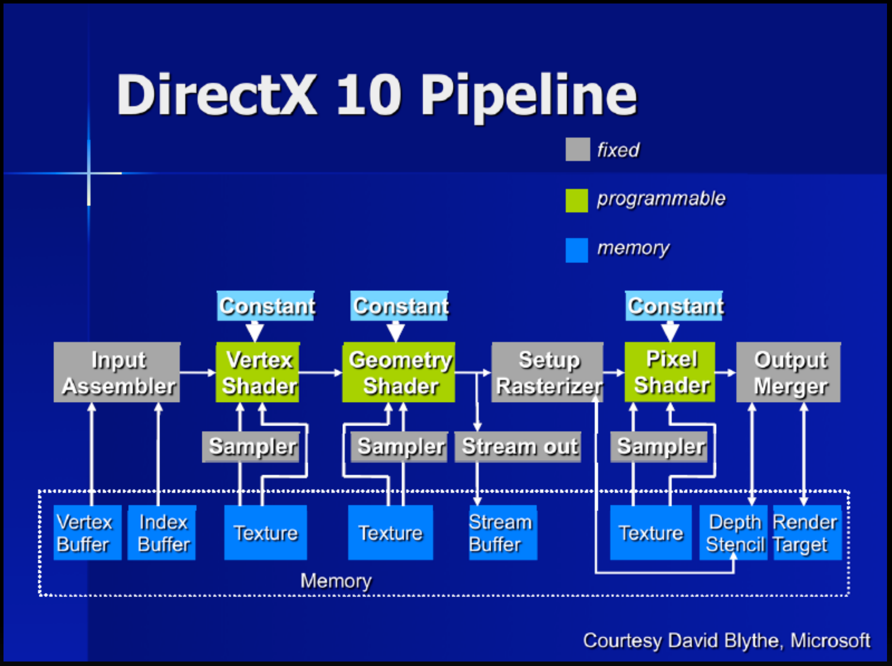
Hardware: 2012-2020

CONCLUSION

CPU, GPU architectures are getting closer

THE GPU TODAY

The GPU Today



- Large frame buffer
- Complicated pipeline
- It's fixed-function
- But we can specify
  shader programs
  that execute in certain pipeline stages

Shader Program Limitations

- No random-access memory writes
  - Can write to current pixel in frame buffer
  - Can't create data structures
- Can't traverse data structures
  - Can hack it using texture accesses
- Hard to share data between main program and shaders programs
- Weird programming language
  - HLSL rather than C/C++

Result: "The Shader ALU Plateau"

Antialiasing Limitations

- MSAA & Oversampling
  - Every 1 bit of output precision costs up to 2X memory & performance!
  - Ideally want 10-20 bits
- Discrete sampling (in general)
  - Texture filtering only implies antialiasing when shader equation is linear
    - Most shader equations are nonlinear

Aliasing is the #1 visual artifact in Gears of War

Texture Sampling Limitations

- Inherent artifacts of bilinear/trilinear
- Poor approximation of Integrate(color,area) in the presence of:
  - Small triangles
  - Texture seams
  - Alpha translucency
  - Masking
- Fixed-function = poor scalability
  - Megatexture, etc

Frame Buffer Model Limitation

- Frame buffer: 1 (or n) layers of 4-vectors, where n = small constant
- Ineffective for
  - General translucency
  - Complex shadowing models
- Memory bandwidth requirement =

  FPS * Pixel Count * Layers Depth * pow(2,n)

  where n = quality of MSAA

Summary of Limitations

- “The Shader ALU Plateau”
- Antialiasing limitations
- Texture Sampling limitations
- Frame Buffer limitations

The Meta-Problem:

- The fixed-function pipeline is too fixed to solve its problems
- Result:
  - All games look similar
  - Derive little benefit from Moore's Law
    - Crysis on high-end NVIDIA SLI solution only looks at most marginally better than top Xbox 360 games

This is a market BEGGING to be disrupted :-)

So…

Return to 100% "Software" Rendering

- Bypass the OpenGL/DirectX API
- Implement a 100% software renderer
  - Bypass all fixed-function pipeline hardware
  - Generate image directly
  - Build & traverse complex data structures
  - Unlimited possibilities

Could implement this…

- On Intel CPU using C/C++
- On NVIDIA GPU using CUDA (no DirectX)

Software Rendering in Unreal 1 (1998)

Ran 100% on CPU

No GPU required!

Features

- Real-time colored lighting
- Volumetric Fog
- Tiled Rendering
- Occlusion Detection

Software Rendering in 1998 vs 2012

60 MHz Pentium could execute:

16 operations per pixel
at 320x200, 30 Hz

In 2012, a 4 Teraflop processor would execute:

16000 operations per pixel
at 1920x1080, 60 Hz

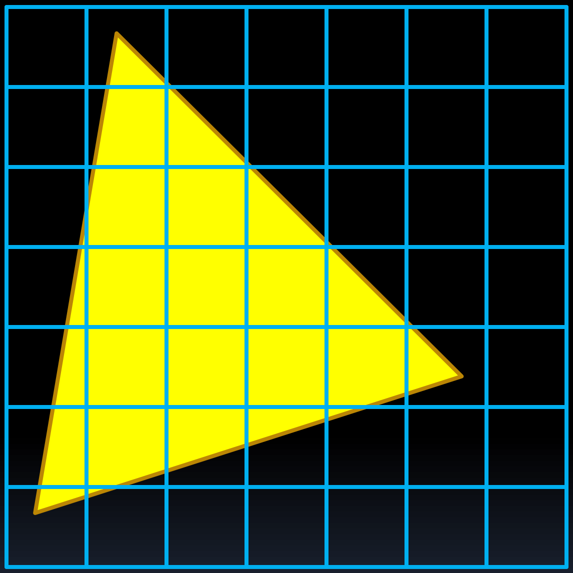
Assumption: Using 50% of computing power for graphics, 50% for gameplay

Future Graphics: Raytracing

- For each pixel
  - Cast a ray off into scene
  - Determine which objects were hit
  - Continue for reflections, refraction, etc
- Consider
  - Less efficient than pure rendering
  - Can use for reflections in traditional render

Future Graphics:
The REYES Rendering Model

- “Dice” all objects in scene down into sub-pixel-sized triangles



- Rendering with
  - Flat Shading (!)
  - Analytic antialiasing
  - Per-pixel occlusion (A-Buffer/BSP)
- Benefits
  - Displacement maps for free
  - Analytic Antialiasing
  - Advanced filtering (Gaussian)
  - Eliminates texture sampling

Future Graphics:
The REYES Rendering Model

Today's Pipeline

- Build 4M poly "high-res" character
- Generate normal maps from geometry in high-res
- Rendering 20K poly "low-res" character in-game

Potential 2012 Pipeline

- Build 4M poly "high-res" character
- Render it in-game!
- Advanced LOD scheme assures proper sub-pixel sized triangles

Future Graphics:
Volumetric Rendering

- Direct Voxel Rendering
  - Raycasting
  - Efficient for trees, foliage
- Tesselated Volume Rendering
  - Marching Cubes
  - Marching Tetrahedrons
- Point Clouds
- Signal-Space Volume Rendering
  - Fourier Projection Slice Theorem
  - Great for clouds, translucent volumetric data

Future Graphics: Software Tiled Rendering

- Split the frame buffer up into bins
  - Example: 1 bin = 8x8 pixels
- Process one bin at a time
  - Transform, rasterize all objects in the bin
- Consider
  - Cache efficiency
  - Deep frame buffers, antialiasing

Hybrid Graphics Algorithms

- Analytic Antialiasing
  - Analytic solution, better than 1024x MSAA
- Sort-independent translucency
  - Sorted linked-list per pixel of fragments requiring per-pixel memory allocation, pointer-following, conditional branching (A-Buffer).
- Advanced shadowing techniques
  - Physically accurate per-pixel penumbra volumes
  - Extension of well-known stencil buffering algorithm
  - Requires storing, traversing, and updating a very simple BSP tree per-pixel with memory allocation and pointed following.
- Scenes with very large numbers of objects
  - Fixed-function GPU + API has 10X-100X state change disadvantage

Graphics: 2012-2020
Potential Industry Goals

Achieve movie-quality:

- Antialiasing
- Direct Lighting
- Shadowing
- Particle Effects
- Reflections

Significantly improve:

- Character animation
- Object counts
- Indirect lighting

SOFTWARE IMPLICATIONS

Software Implications

Software must scale to…

- 10’s – 100’s of threads
- Vector instruction sets

Software Implications

Programming Models

- Shared State Concurrency
- Message Passing
- Pure Functional Programming
- Software Transactional Memory

Multithreading in Unreal Engine 3: "Task Parallelism"

- Gameplay thread
  - AI, scripting
  - Thousands of interacting objects
- Rendering thread
  - Scene traversal, occlusion
  - Direct3D command submission
- Pool of helper threads for other work
  - Physics Solver
  - Animation Updates

Good for 4 threads.
No good for 100 threads!

"Shared State Concurrency"
The standard C++/Java threading model

- Many threads are running
- There is 512MB of data
- Any thread can modify any data at any time
- All synchronization is explicit, manual
  - See: LOCK, MUTEX, SEMAPHORE
- No compile-time verification of correctness properties:
  - Deadlock-free
  - Race-free
  - Invariants

Multithreaded Gameplay Simulation: Manual Synchronization

Idea:

- Update objects in multiple threads
- Each object contains a lock
- “Just lock an object before using it”

Problems:

- “Deadlocks”
- “Data Races”
- Debugging is difficult/expensive

Multithreaded Gameplay Simulation: “Message Passing”

Idea:

- Update objects in multiple threads
- Each object can only modify itself
- Communicate with other objects by sending messages

Problems:

- Requires writing 1000’s of message protocols
- Still need synchronization

Pure Functional Programming

"Pure Functional" programming style:

- Define algorithms that don't write to shared memory or perform I/O operations

  (their only effect is to return a result)

Examples:

- Collision Detection
- Physics Solver
- Pixel Shading

Pure Functional Programming

"Inside a function with no side effects,
sub-computations can be run in any order,
or concurrently,
without affecting the function's result"

With this property:

- A programmer can explicitly multithread the code, safely.
- Future compilers will be able to automatically multithread the code, safely.

See: "**Implementing Lazy Functional Languages on Stock Hardware**";
Simon Peyton Jones; Journal of Functional Programming 2005

Multithreaded Gameplay Simulation: Software Transactional Memory

Idea:

- Update objects in multiple threads
- Each thread runs inside a transaction block
  and has an atomic view of its "local" changes to memory
- C++ runtime detects conflicts between transactions
  - Non-conflicting transactions are applied to "global" memory
  - Conflicting transactions are "rolled back" and re-run

Implemented 100% in software; no custom hardware required.

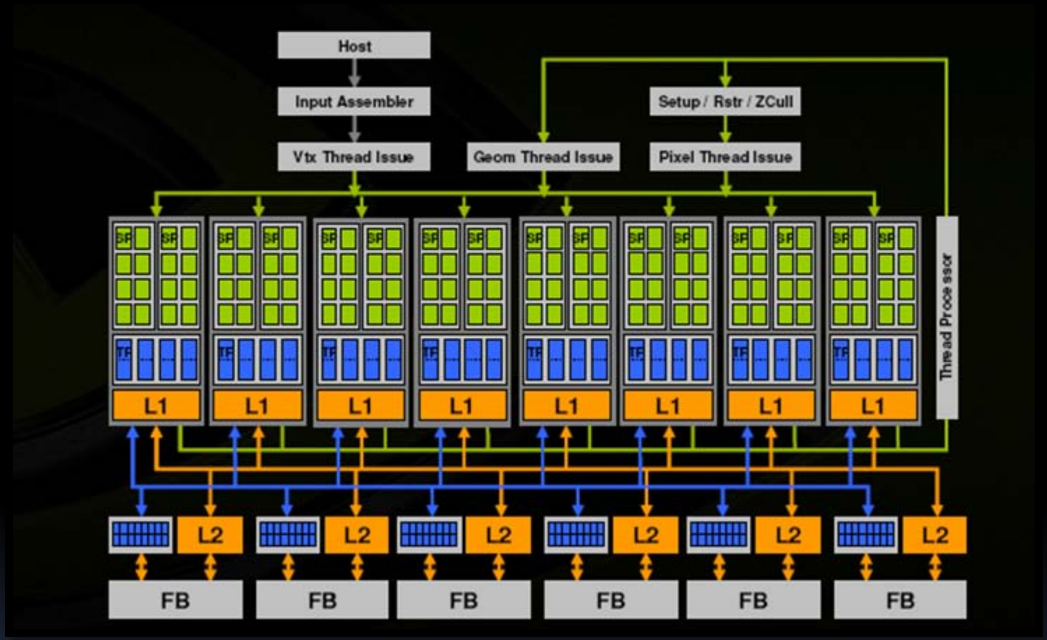
Problems:

- "Object update" code must be free of side-effects
- Requires C++ runtime support
- Cost around 30% performance

See: "Composable Memory Transactions"; Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. ACM Conference on Principles and Practice of Parallel Programming 2005

Vectorization

Supporting “Vector Instruction Sets” efficiently



NVIDIA GeForce 8:

- 8 to 15 cores
- 16-wide vectors

Vectorization

C++, Java compilers generate “scalar” code

GPU Shader compilers generate “vector” code

- Arbitrary vector size (4, 16, 64, …)
- N-wide vectors yield N-wide speedup

Vectorization: “The Old Way”

- “Old Vectors” (SIMD):
  Intel SSE, Motorola Altivec
  - 4-wide vectors
  - 4-wide arithmetic operations
  - Vector loads
    Load vector register from vector stored in memory
  - Vector swizzle & mask

Future Programming Models: Vectorization

- "Old Vectors"
  Intel SSE, Motorola Altivec

```
vec4 x,y,z;
...
z = x+y;
```

Vectorization: “New Vectors”

(ATI, NVIDIA GeForce 8, Intel Larrabee)

- 16-wide vectors
- 16-wide arithmetic
- Vector loads/stores
  - Load 16-wide vector register from scalars from 16 *independent* memory addresses, where the addresses are stored in a vector!
  - Analogy: Register-indexed constant access in DirectX
- Conditional vector masks

“New SIMD” is better than “Old SIMD”

- “Old Vectors” were only useful when dealing with vector-like data types:
  - “XYZW” vectors from graphics
  - 4x4 matrices

- “New Vectors” are far more powerful:

  Any loop whose body has a statically-known call graph free of sequential dependencies can be “vectorized”, or compiled into an equivalent 16-wide vector program. And it runs up to 16X faster!

“New Vectors” are universal

```
int n;
cmplx coords[];
int color[] = new int[n]

for(int i=0; i<n; i++) {
    int j=0;
    cmplx c=cmplx(0,0)
    while(mag(c) < 2) {
          c=c*c + coords[i];
          j++;
    }
    color[i] = j;
}
```

(Mandelbrot set generator)

This code…

- is free of sequential dependencies
- has a statically known call graph

Therefore, we can mechanically transform it into an equivalent data parallel code fragment.

“New Vectors” Translation

```
for(int i=0; i<n; i++) {
    …
}
```

```
for(int i=0; i<n; i+=N) {
    i_vector={i,i+1,..i+N-1}
    i_mask={i<n,i+1<N,i+2<N,..i+N-1<N}
    …
}
```

Standard data-parallel loop setup

Note: Any code outside this loop
(which invokes the loop)
is necessarily scalar!

"New Vectors" Translation

```
int n;
cmplx coords[];
int color[] = new int[n]

for(int i=0; i<n; i++) {
     int j=0;
     cmplx c=cmplx(0,0)
     while(mag(c) < 2) {
          c=c*c +
coords[i];
          j++;
     }
     color[i] = j;
}
```

Note: Any code outside this loop (which invokes the loop) is necessarily scalar!

```
int n;
cmplx coords[];
int color[] = new int[n]

for(int i=0; i<n; i+=N) {
     int[N] i_vector={i,i+1,..i+N-1}
     bool[N] i_mask={i<n,i+1<N,i+2<N,..i+N-1<N}

     complx[N] c_vector={cmplx(0,0),..}

     while(1) {
          bool[N] while_vector={
                    i_mask[0] && mag(c_vector[0])<2,
                    ..
          }
          if(all_false(while_vector))
                    break;
          c_vector=c_vector*c_vector + coords[i..i+N-1 : i_mask]
     }
     colors[i..i+N-1 : i_mask] = c_vector;
}
```

Loop Index Vector

Loop Mask Vector

Vectorized Loop Variable

Vectorized Conditional:
Propagates loop mask
to local condition

Mask-predicated
vector write

Mask-predicated
vector read

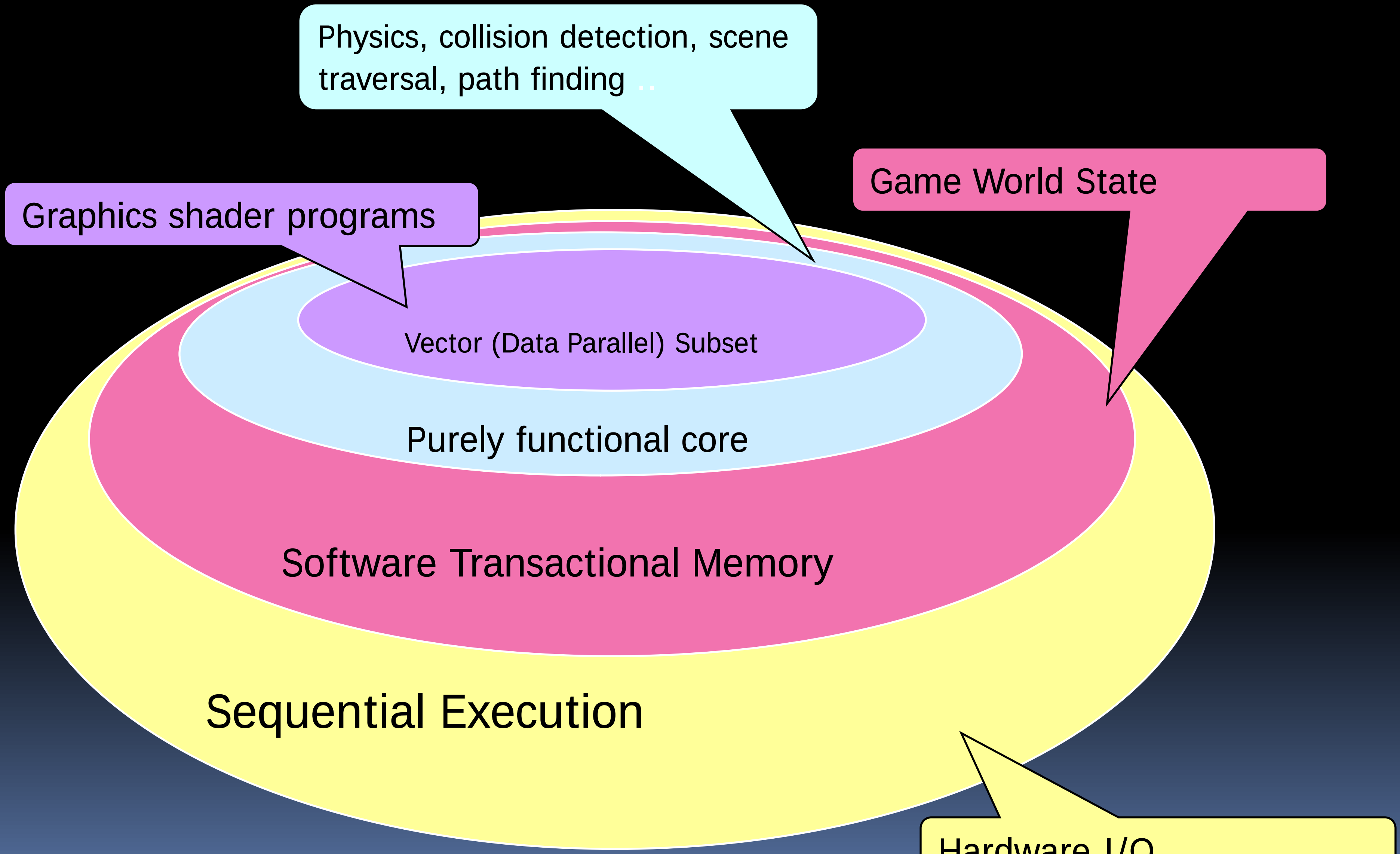
Vectorization Tricks

- Vectorization of loops
  - Subexpressions independent of loop variable are scalar and can be lifted out of loop
  - Subexpressions dependent on loop variable are vectorized
  - Each loop iteration computes an “active mask” enabling operation on some subset of the N components
- Vectorization of function calls
  - For every scalar function, generate an N-wide vector version of the function taking an N-wide “active mask”
- Vectorization of conditionals
  - Evaluate N-wide conditional and combine it with the current active mask
  - Execute “true” branch if any masked conditions true
  - Execute “false” branch if any masked conditions false
  - Will often execute both branches

Vectorization Paradigms

- Hand-coded vector operations
  - Current approach to SSE/Altivec
- Loop vectorization
  - See: Vectorizing compilers
- Run a big function with a big bundle of data
  - CUDA/OpenCL
- Nested Data Parallelism
  - See NESTL
  - Very general set of “vectorization” transforms for many categories of nested computations

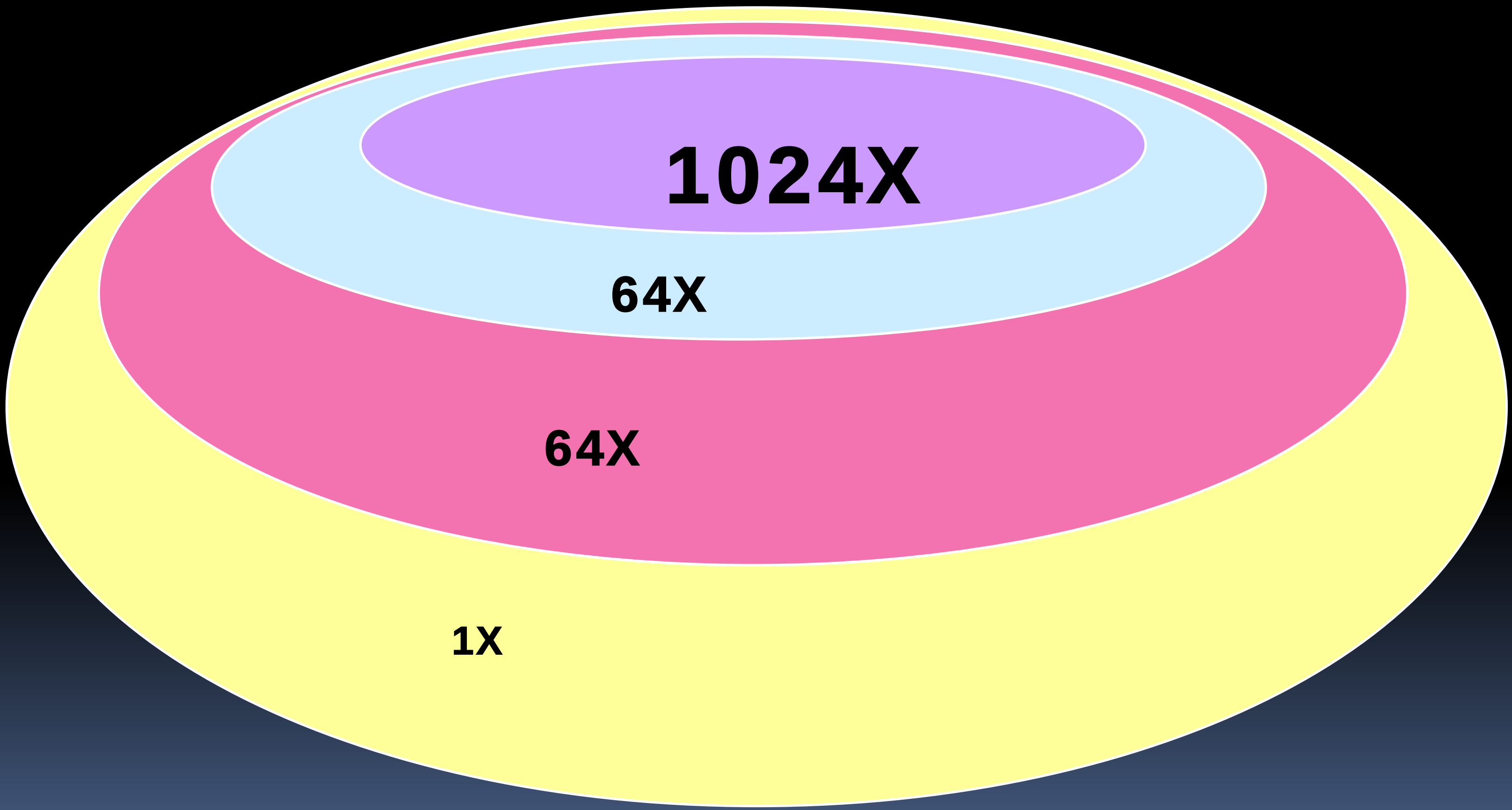
Layers: Multithreading & Vectors



Potential Performance Gains*: 2012-2020

Up to…

- 64X for multithreading
- 1024X for multithreading + vectors!



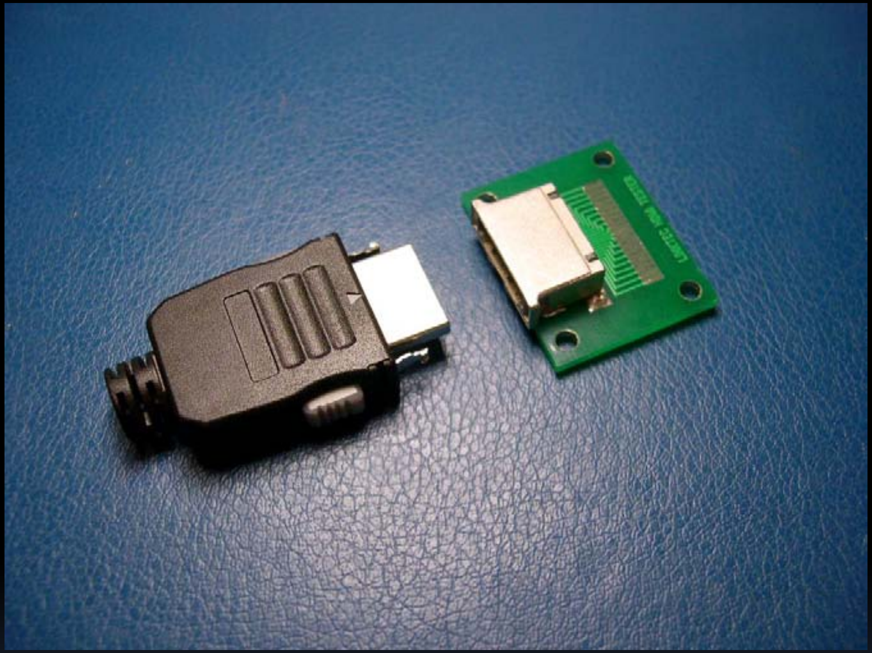
* My estimate of feasibility based on Moore's Law

Multithreading & Vectorization: Who Choses?

- Hardware companies impose a limited model on developers
  - Sony Cell, NVIDIA CUDA, Apple OpenCL
- Hardware provides general feature; languages & runtimes make it nice; users choose!
  - Tradeoffs
    - Performance
    - Productivity
    - Familiarity

HARDWARE IMPLICATIONS

The Graphics Hardware of the Future



All else is just ***computing!***

Future Hardware:
A unified architecture for computing and graphics

Hardware Model

- Three performance dimensions
  - Clock rate
  - Cores
  - Vector width
- Executes two kinds of code:
  - Scalar code (like x86, PowerPC)
  - Vector code (like GPU shaders or SSE/Altivec)
- Some fixed-function hardware
  - Texture sampling
  - Rasterization?

Vector Instruction Issues

- A future computing device needs…
  - Full vector ISA
    - Masking & scatter/gather memory access
    - 64-bit integer ops & memory addressing
  - Full scalar ISA
    - Dynamic control-flow is essential
  - Efficient support for scalar<->vector transitions
    - Initiating a vector computation
    - Reducing the results
    - Repacking vectors
    - Must support billions of transitions per second

Memory System Issues

Effective bandwidth demands will be *huge*

Typically read 1 byte of memory per FLOP

4 TFLOP of computing power

demands

4 TBPS of effective memory bandwidth!

Yes, really!

Memory System Issues

<u>Threads (GPU)</u>

- Hide memory latency
- Lose data locality

<u>Caches (CPU)</u>

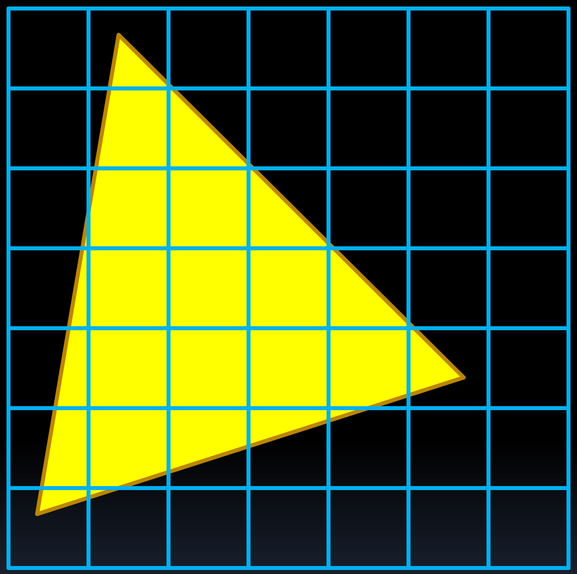
- Expose memory latency
- Exploit data locality to minimize main memory bandwidth

Memory System Issues

- Cache coherency is vital
  - It should be the default

Revisiting REYES

- "Dice" all objects in scene down into sub-pixel-sized triangles
  - Tile-based setup
- Rendering with
  - Flat Shading
    - No texture sampling
  - Analytic antialiasing
  - Per-pixel occlusion (A-Buffer/BSP)



Requires no artificial software threading or pipelining.

LESSONS LEARNED

Lessons learned:
Productivity is vital!

Hardware will become 20X faster, but:

- Game budgets will increase less than 2X.

Therefore…

- Developers must be willing to sacrifice performance in order to gain productivity.
- High-level programming beats low-level programming.
- Easier hardware beats faster hardware!
- We need great tools: compilers, engines, middleware libraries…

Lessons learned:
Today's hardware is too hard!

- If it costs X (time, money, pain) to develop an efficient single-threaded algorithm, then…
  - Multithreaded version costs **2X**
  - PlayStation 3 Cell version costs **5X**
  - Current "GPGPU" version is costs: **10X** or more
- Over 2X is uneconomical for most software companies!
- This is an argument against:
  - Hardware that requires difficult programming techniques
  - Non-unified memory architectures
  - Limited "GPGPU" programming models

Lessons learned:
Plan Ahead

Previous Generation:

- Lead-time for engine development was 3 years
- Unreal Engine 3:
  - 2003: development started
  - 2006: first game shipped

Next Generation:

- Lead-time for engine development is 5 years
- Start in 2009, ship in 2014!

So, let's get started!

CONCLUSION

END