Tim Sweeney
Epic Games
tim@epicgames.com

WILD SPECULATION ON CONSUMER WORKLOADS: 2012-2020

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
- Leading supplier of Game Engines

History:
Unreal Engine

Unreal Engine 1
1996-1999

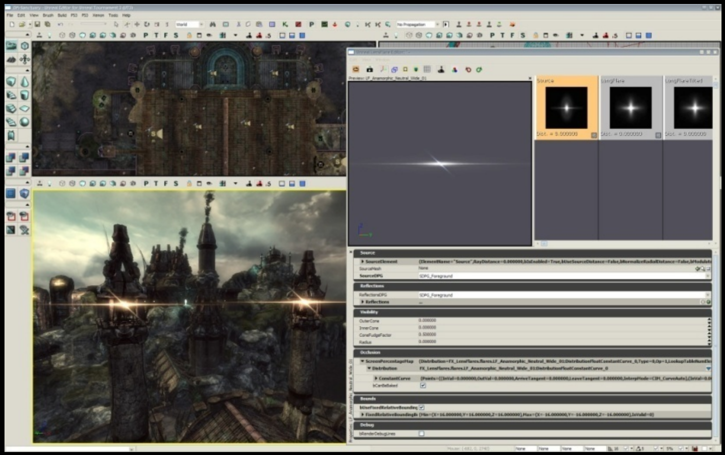
- First modern game engine
  - Object-oriented
  - Real-time, visual toolset
  - Scripting language
- Last major software renderer
  - Software texture mapping
  - Colored lighting, shadowing
  - Volumetric lighting & fog
  - Pixel-accurate culling
- 25 games shipped

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)

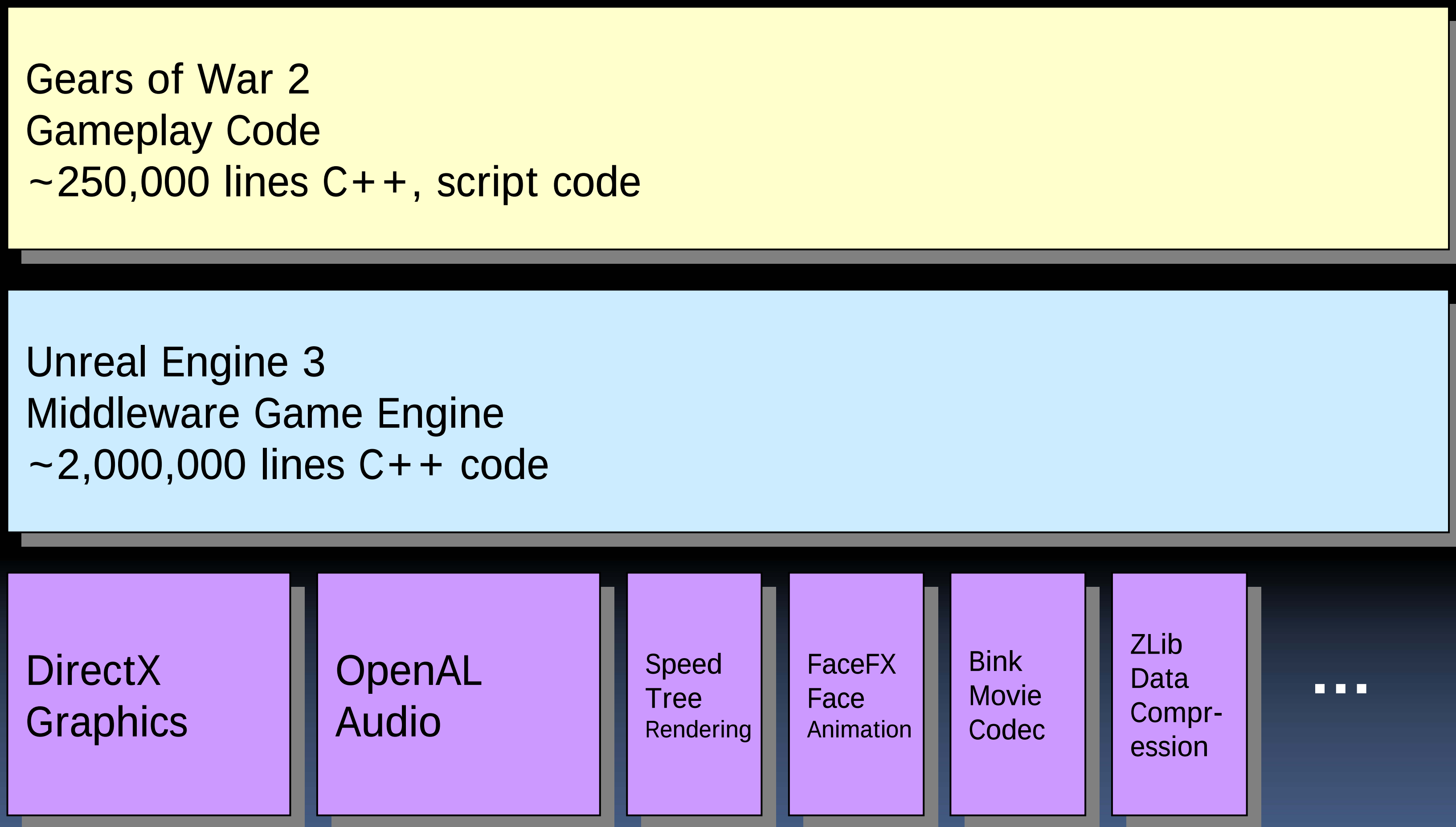
BioShock (2K Games)

Game Development: 2008

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

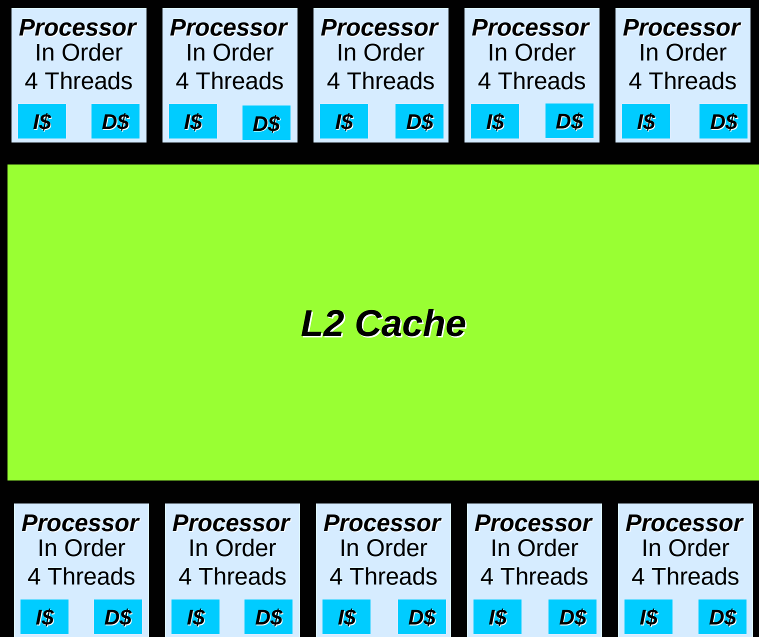
Graphics History

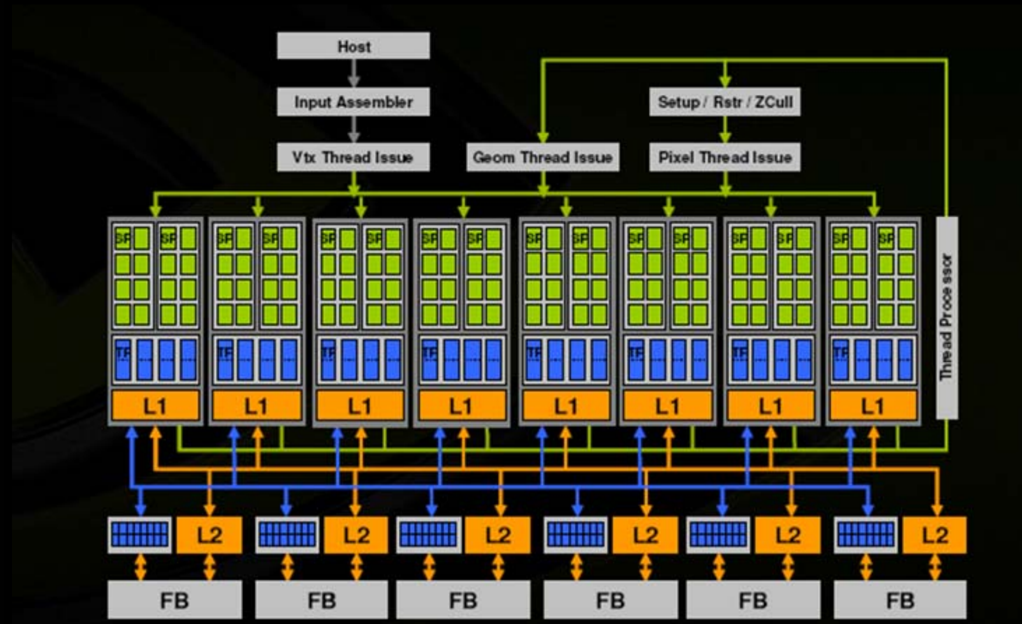
1984	3D workstation (SGI)
1997	GPU (3dfx)
2002	DirectX9, Pixel shaders (ATI)
2006	GPU with full programming language (NVIDIA GeForce 8)
2009?	x86 CPU/GPU Hybrid (Intel Larrabee)

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

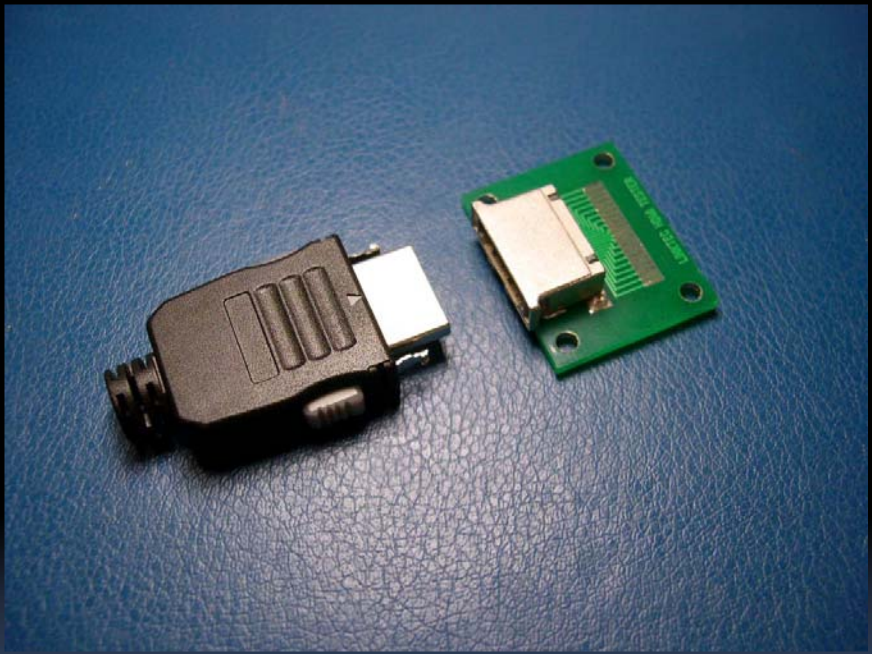
- General Purpose GPU
- CUDA "C" Compiler
- DirectX/OpenGL
- Many-core, vector architecture
- Teraflop-class performance

Hardware: 2012-2020

CONCLUSION

CPU, GPU architectures are converging

The Graphics Hardware of the Future



All else is just *computing!*

Potential Hardware 2012-2020: 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?

Market Analysis:
Teraflop Consumer Applications

Teraflop Consumer Applications

Teraflop Consumer Applications

1. Games

Teraflop Consumer Applications

1. Games

THE END

Game Development:

2012-2020

Game Development: 2012-2020

- Programming
  - How do we write code for 100 cores?
- Graphics
  - What's possible beyond DirectX / OpenGL?
- Implications for Performance Analysis

PROGRAMMING: 2012-2020

Programming: 2012-2020
The Essentials

- Developer Productivity
  - A programmer's time is valuable
  - Productivity is very important!
  - We must make multi-core programming easy!
- Performance
  - Supporting "many cores" with multithreading
  - Scaling to "vectors instruction sets"
  - Understanding Performance Implications

Multithreading: 2012-2020

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 cores.
No good for 40 cores!

"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

Multithreaded Gameplay Simulation

- 1000+ of game objects
- Each object is:
  - Modifyable
  - Updated once per frame
    - Each update touches 5-10 other objects
    - Updates are object-oriented,
      so control-flow isn't statically known
- Code written by 10's of programmers
  - They aren't computer scientists!

How to Multithread?

Multithreaded Gameplay Simulation

Problems:

- Games must scale to "many cores" (20-100)
- Must avoid all single-threaded bottlenecks

Solutions:

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

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

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

Multithreaded Gameplay Simulation: Conclusion

Manual synchronization:

- Very difficult, error-prone

Message passing

- Difficult, error-prone

Transactional Memory

- Easy! Almost as easy as single-threading.
- Analysis indicates reasonable cost,
  low conflict rate (1-5%)

Claim: Transactional memory is the only productive, scalable solution for multithreaded gameplay simulation.

Transactional Memory: Implementation

Implementation options:

- Indirect all memory reads/writes to change-list (e.g. using a hash table)
- Generate minimal change-list for undo
- Store transaction detail per memory-cell

Threading Model

- Main thread invokes transactions
- Launch 1 thread per running transaction
  - Ideally want to run 1 transaction per hardware thread

Transactional Memory: Expectations

- Game runs at 30-60 frames per second
- 1000 - 100,000 transactions per frame
- Low conflict rate
  Guess: 5%
- Transactions significantly impact performance
  Guess: 30% to 100% overhead
  - Extra instructions for all mutable memory reads/writes
  - Working set increases, cache efficiency decreases
- But it's worth it!
  Willing to trade performance for productivity!

Transactional Memory: Performance Measurement

- Each transaction is single-threaded
  Thus performance is easy to understand
- Transaction conflicts are the essential consideration
  - Can sequentialize all code + add 2X overhead
  - Want to be able to…
    - Measure conflict rate
    - Attribute conflicts to particular transactions & memory reads/writes
    - Ideally, analyze all *potential* conflicts in a set of transactions to be run
      Not just those that actually occurred nondeterministically

Pure Functional Programming

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 Example: Collision Detection

A collision detection algorithm takes a line segment and determines when and where a point moving along that line will collide with a (constant) geometric dataset.

```
struct vec3
{
    float x,y,z;
};
struct hit
{
    bool  DidCollide;
    float Time;
    vec3  Location;
};
hit collide(vec3 start,vec3 end);
```

(it doesn’t modify any shared memory)

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

Pure Functional Programming: Threading Model

Threading Model

- Start a pool of helper threads
  - 1 per hardware thread
- Top-level program executed in one thread
- All execution may be multithreaded
  - “Small” subexpressions evaluated normally
  - “Large” subexpressions evaluated via thunk
    - Thunk data structure represents a suspended computation
  - Can evaluate thunks in
    - Same thread
    - Separate thread

Pure Functional Programming: Cache Locality Considerations

- Mustn't hand off sub-expression computations to random threads
- Use algorithm like "work stealing"
  - Each thread contains a work queue
  - If no helper threads available, add thunk to creating thread's queue
  - When one thread stalls, help another thread finish its work by pulling from the tail of its queue
  - Arrange work-stealing order in hierarchy to improve locality

Pure Functional Programming: Workload Expectations (Part 1)

- 80-90% of future game execution time may be spent in pure functional code
  - Rendering, Physics, Collision, Animation, …
- Threads will handle *tiny* chunks of work
  - 100 - 10,000's of instructions
    Thus thread coordination overhead must be small!
  - Expect 500M - 1G thread invocations per second
- Thunk dependencies are the key consideration!
  - Can accidentally sequentialize execution!

Pure Functional Programming: Workload Expectations (Part 2)

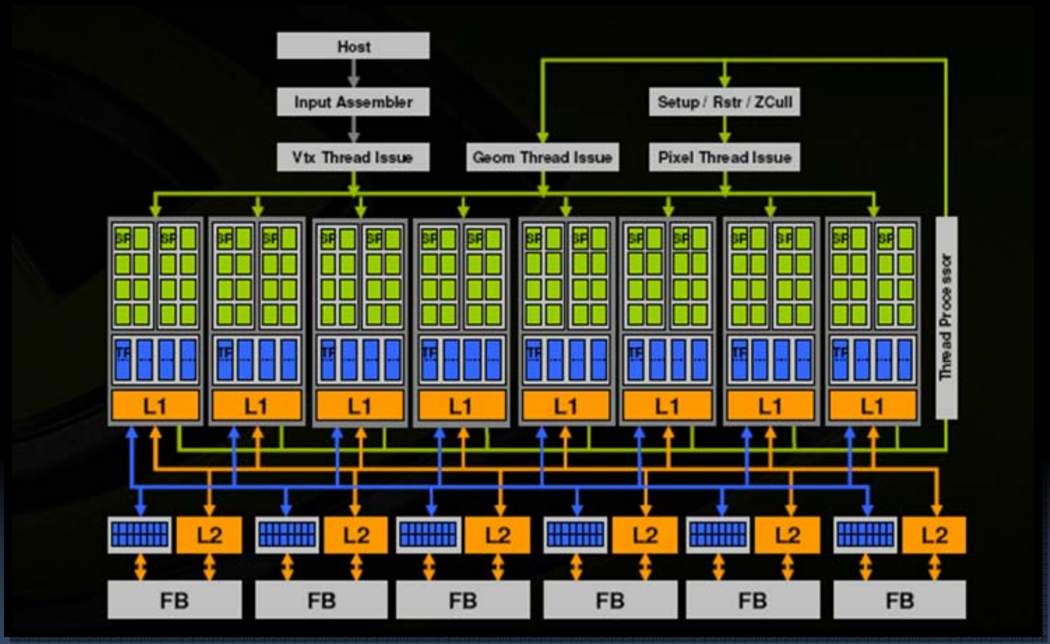
- Games will be able to scale to…
  - 10's of threads with C++
  - 100's of threads with future programming languages
    Where we write functional & transactional code,
    and compiler/runtime do the threading for us.
- Thread coordination will significantly impact performance
  Guess: 30% to 100% overhead
- But it's worth it!
  Willing to trade performance for productivity!

Implicit Multithreading: Performance Measurement

- Threading performance is non-deterministic
  Even when programming model is deterministic
  - Lots of threads executing dynamically-scheduled work
  - Cache locality depends on dynamic threading decisions
- Performance attribution is tricky
  - Must associate performance counters with code currently running on thread
    …which may change every 100 instructions!

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

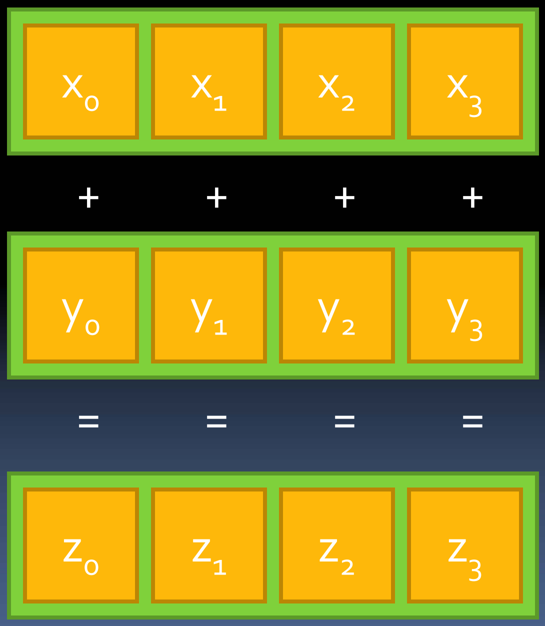
Vectorization: “The Old Way”

- “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

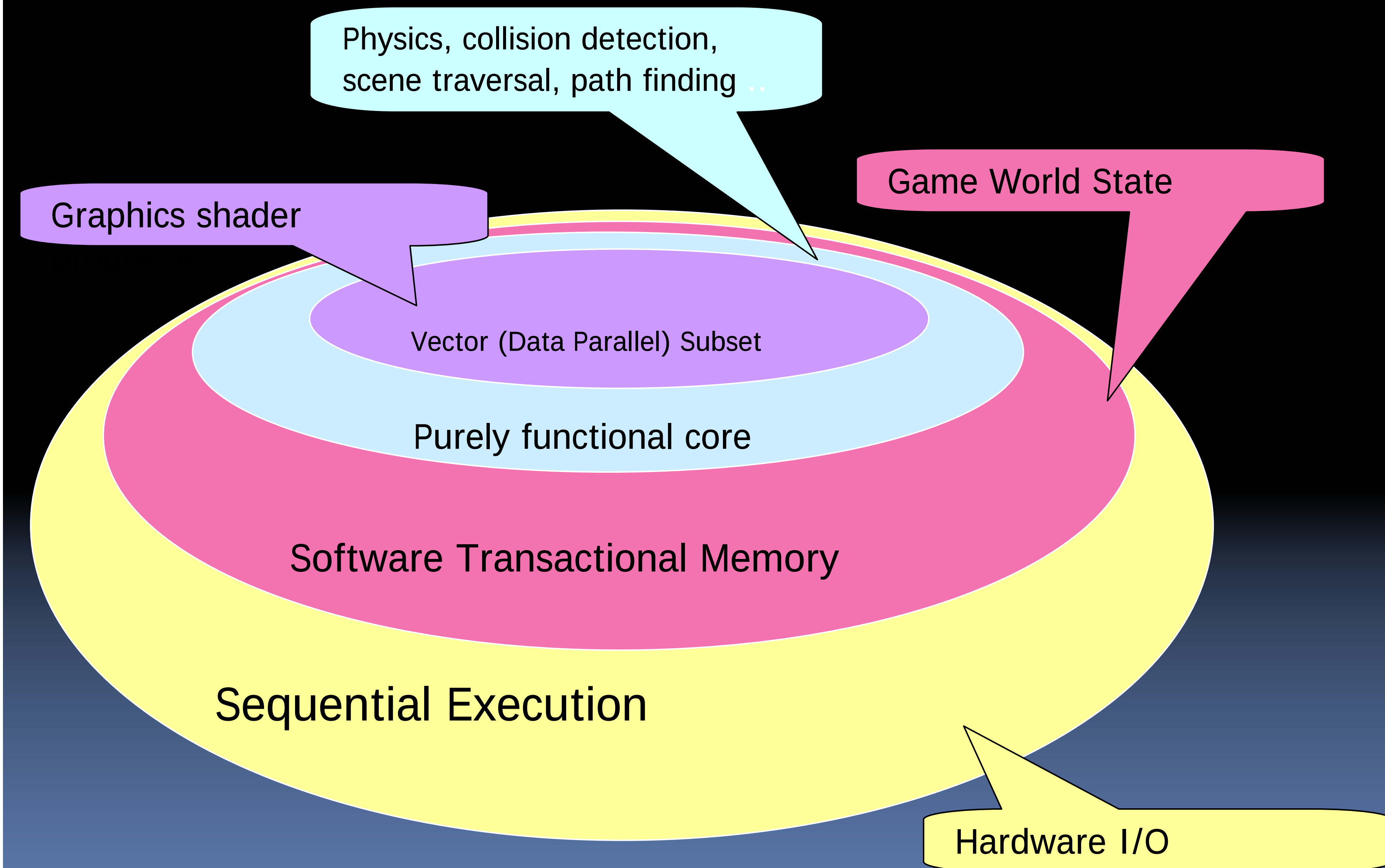
Mask-predicated
vector write

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

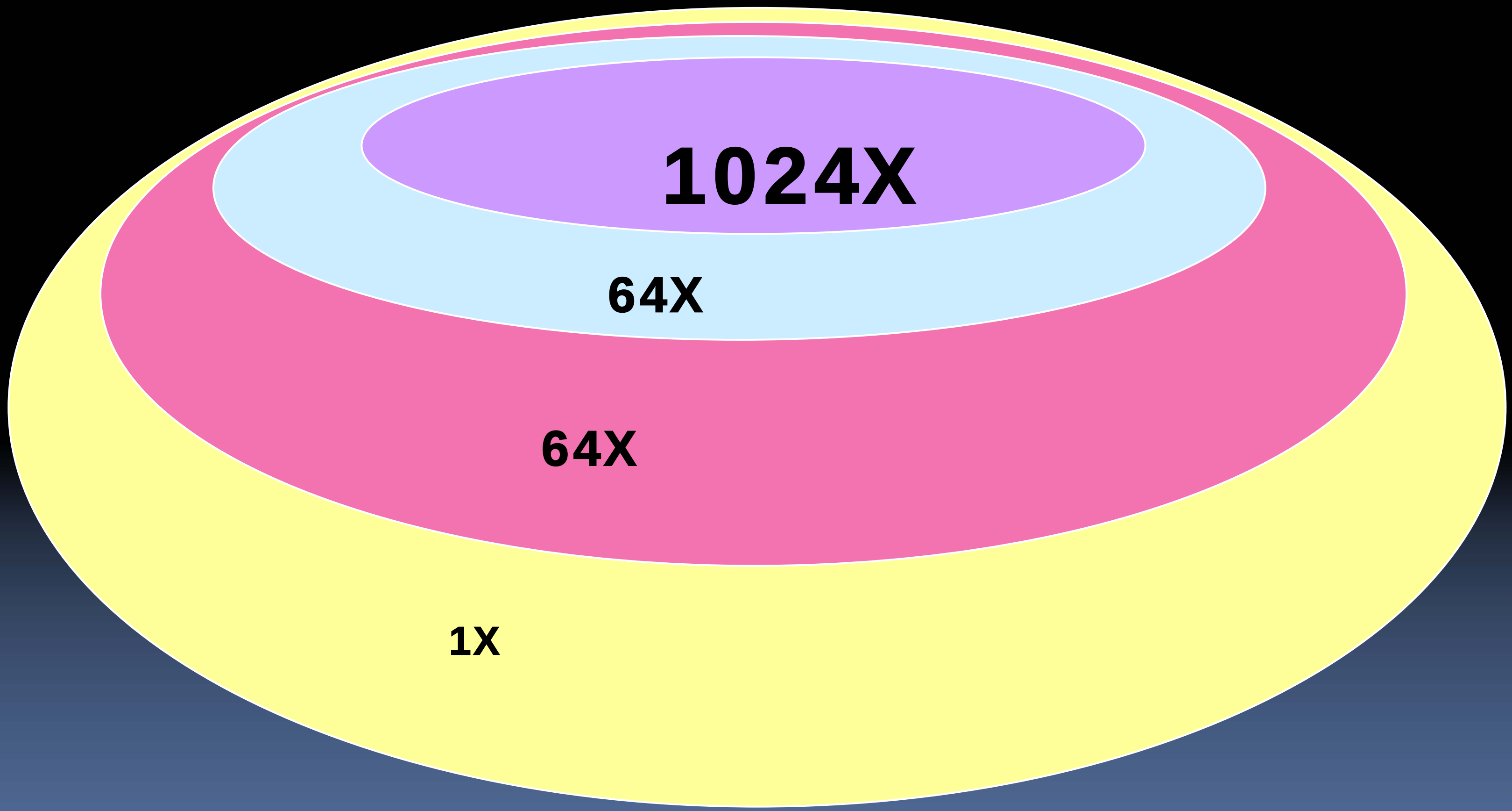
Layers: Multithreading & Vectors



Potential Performance Gains*: 2012-2020

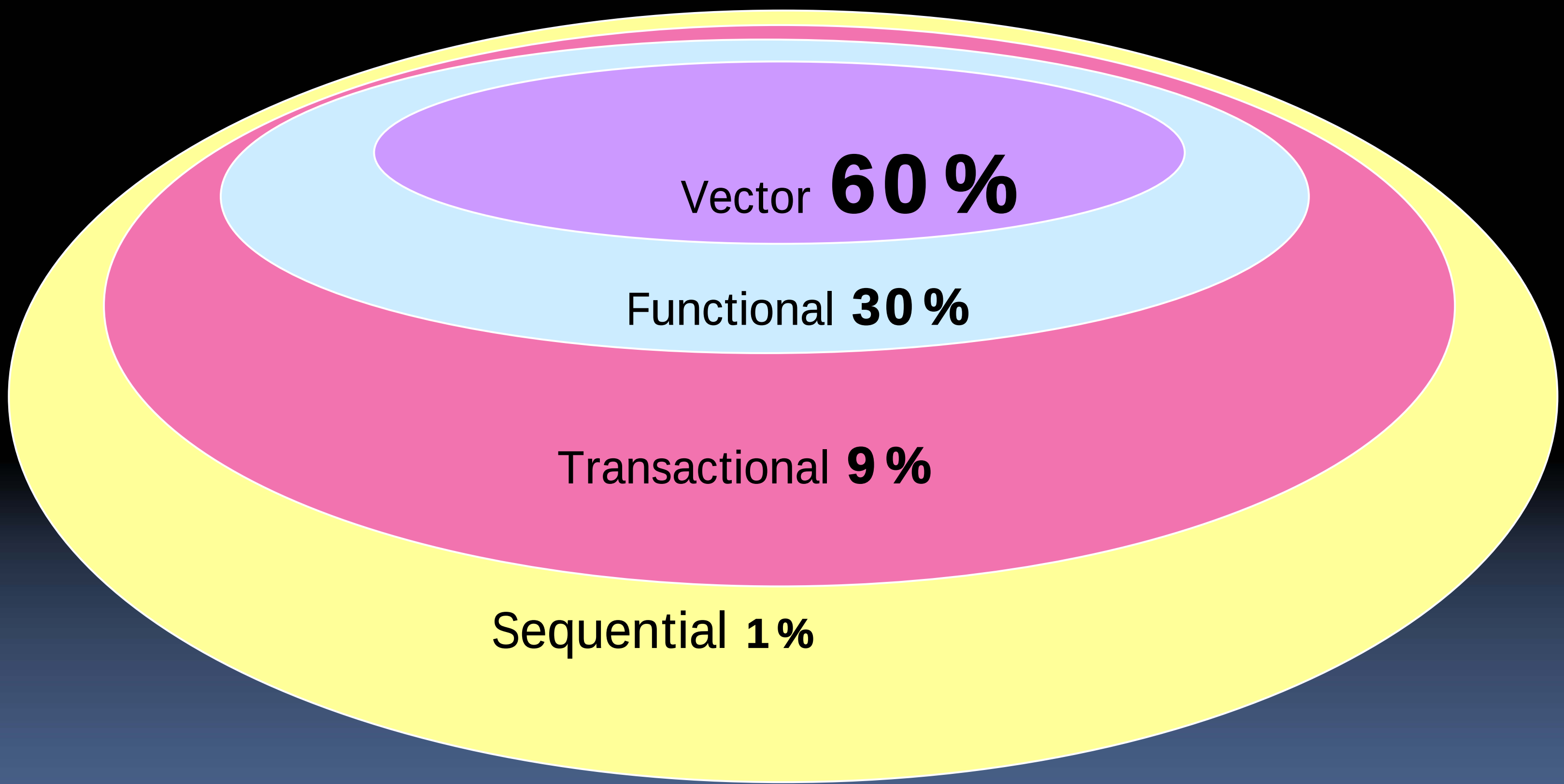
Up to…

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



* My estimate of feasibility based on Moore's Law

Potential Thread Execution Profile for 2012-2020 games*



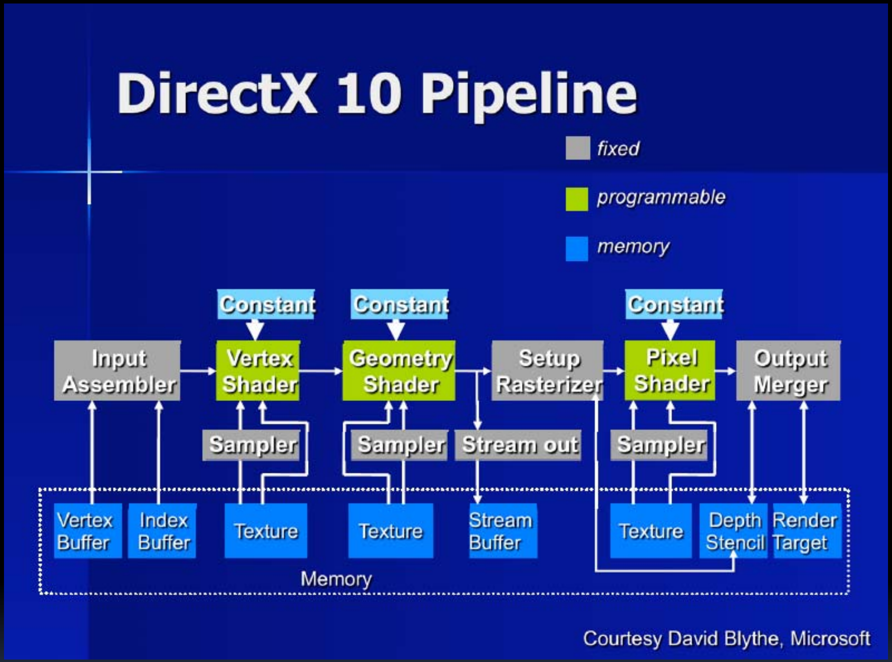
* Wild guesses based on analysis and extrapolation of Unreal Engine 3 systems and performance characteristics

Vectorizable Workload Considerations

- Subtle compiler feature now impacts performance up to 16X
- What do we really care to measure?
  FLOPS or IPS?
- Cache complications
  What happens given 16X more FLOPS unsupported by 16X more bandwidth or cache?

GRAPHICS: 2012-2020

GPU Programming in 2008: The Model



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

GPU Programming Today:
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++

GPU Programming Today: Problems

- All games look similar
  - Due to fixed-function pipline stages
- "The shader ALU plateau"
  - Given 10X more pixel shader performance, games may only look 2X better
- Poor anti-aliasing model (MSAA)

Future Graphics:
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

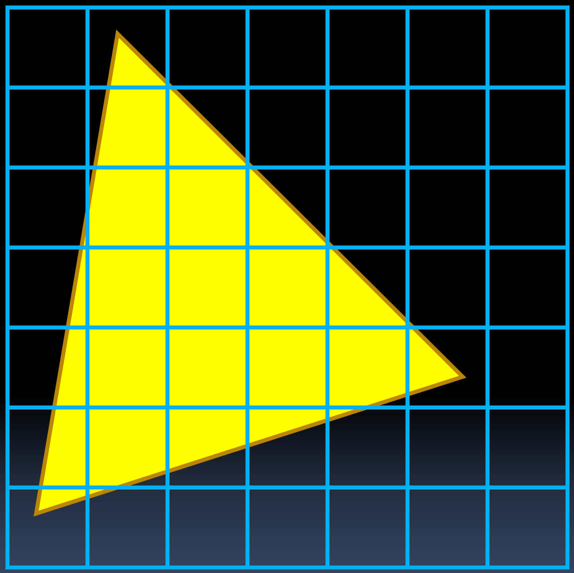
Assumption: Using 50% of computing power for graphics, 50% for gameplay

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
- 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

Graphics: 2012-2020
Workload Implications

- Must abandon legacy benchmarks (DirectX/OpenGL)
  - But what to measure next?
- Software graphics will improve load-balancing
  - Can freely move FLOPS between graphics and other game tasks

    Expect to see games explore the design space quite broadly!
- Graphics remains almost "perfectly parallel"
  - Likely to remain primary consumer of FLOPS

FINAL THOUGHTS

Current hardware & programming models are too tricky!

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 architectures with difficult programming models
- For tools, techniques, and languages that improve productivity

Language Considerations

C++ remains essential

- 100% of today's games are C++
- Major language transitions take 10 years

New mainstream language eventually

- C++/Java/C# threading model is too low-level
- Major opportunities in…
  - Parallel workload performance
  - Productivity
  - Safety/security

Future Mainstream Programming Language Expectatuions

- Garbage collection
  - Productivity gain outweighs cost
- First-class effects declarations
  - Vectorizable subset
  - Pure functional subset
  - Transactional subset
  - Sequential full language
- More powerful type system
  - Safe arrays, memory access, etc
  - More compile-time verification (a la hardware)
- Low-level, single-thread performance 30% to 2X worse than C
- Performance analysis more difficult

CONCLUSION

Teraflop Consumer Computing: Conclusions

- Crazy new hardware is coming…
  - Lots of cores
  - Vector instruction sets
- Software will change considerably
  - Transactional & Functional programming
  - Scaling to 100's of threads & wide vectors
  - Return to software graphics
- Developers will be willing to sacrifice performance in order to gain productivity more than before
  - High-level programming beats low-level programming
  - Better multithreading abstractions

END