

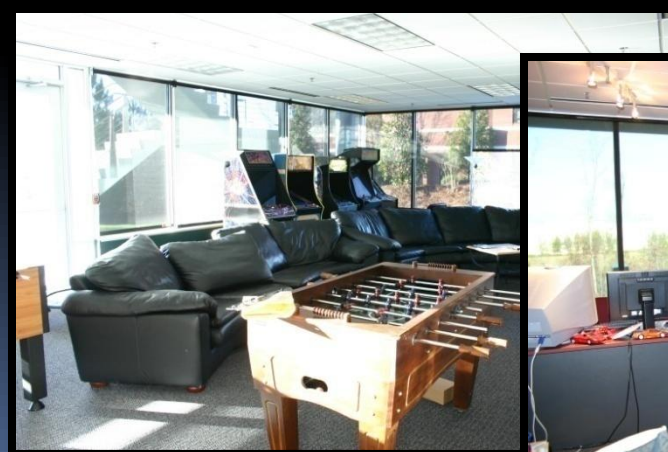
Tim Sweeney
Epic Games Inc

tim@epicgames.com

COMPUTING & GRAPHICS IN GAMES

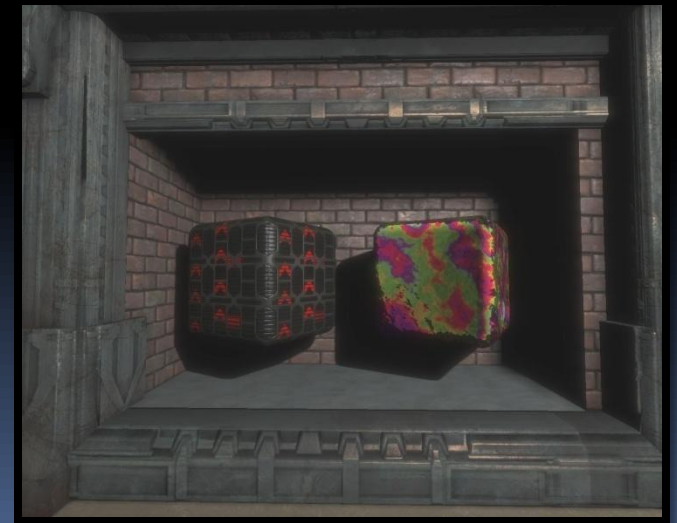
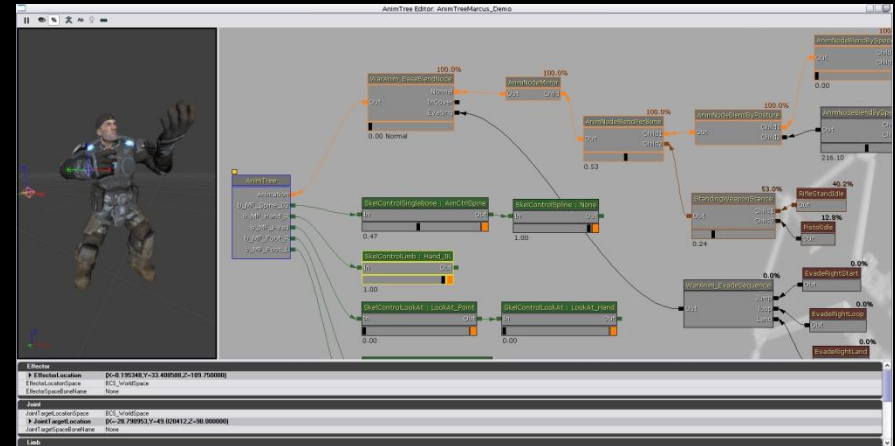
Background: Epic Games

- Located in Raleigh, North Carolina
- Shipped ~40 games from 1991-2007
 - ZTZ, Jill of the Jungle, Jazz Jackrabbit, Epic Pinball
 - Unreal, Unreal Tournament
 - Gears of War
- Game Engine provider, 1996-2007



Background: Unreal Engine

- Unreal Engine 1
 - For PC
 - 1996-1999
 - Software renderer
- Unreal Engine 2
 - PC, PlayStation 2, Xbox
 - 2000-2005
 - DirectX7 ("Hardware T&L")
- Unreal Engine 3
 - Xbox 360, PlayStation 3, PC
 - DirectX9, 100% shader-based
 - 2006-2011

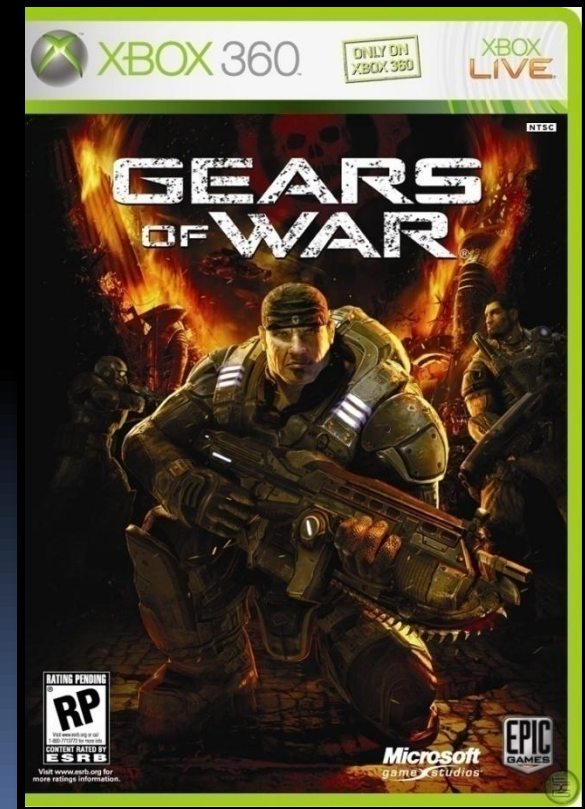


Background: Unreal Engine 3

- General-purpose feature set for 3D game development
 - Rendering
 - Physics
 - Sound
 - Animation
 - Scripting
 - Networking
 - Artificial Intelligence
 - Tools
- Xbox 360, PlayStation 3, PC
- Over 150 projects in development
 - Typical game budget is \$10-20M
- ~60 man-years of development effort

Perspective: Gears of War

- Resources
 - 10 programmers
 - 25 artists
 - 24 month development cycle
 - \$12M budget
- Software Dependencies
 - 1 middleware game engine
 - ~20 middleware libraries
 - OS graphics APIs, sound, input, ...



Perspective: Gears of War

Gears of War
Gameplay Code
~250,000 lines C++, script code

Unreal Engine 3
Middleware Game Engine
~750,000 lines C++ code

DirectX
Graphics

Ageia PhysX
Physics

Ogg
Vorbis
Music
Codec

Speex
Speech
Codec

wx
Widgets
Window
Library

ZLib
Data
Compression

...

Outline

- Languages
- Memory Management
- Concurrency
- Vectorization
- Correctness

Languages

C++ (Stroustrup, 1983)

- ❑ Object-oriented descended of C (1972)
- ❑ Paradigm: “Programming by side-effect”
- ❑ Unsafe pointers, casts, etc

What are the hard problems?

- ❑ Memory management
- ❑ Correctness
- ❑ Concurrency



```
int sum(int* elements,int count)
{
    int accumulator=0;
    for(int i=0; i<count; i++)
        accumulator+=elements[i];
    return accumulator;
}
```

Languages

Java (1991), C# (2001)

C-style languages extended with

- Garbage collection
 - Introspection
 - Type safety guarantees
 - Various helpful constructs (iterators, etc)
-
- What are the hard problems?
 - Correctness
 - Concurrency



MEMORY MANAGEMENT

The Problem of Memory Management

Memory Management Problems in C++

- Dangling pointers
 - read from a pointer after it's freed
 - write to a pointer after it's freed
- Memory leaks
 - allocate memory and never free it

The Problem of Memory Management

Memory Management Problems in C++

- Dangling pointers
 - read from a pointer after it's freed
 - write to a pointer after it's freed
- Memory leaks
 - allocate memory and never free it

Garbage collection has solved this problem completely in Java and C#

CONCURRENCY

CONCURRENCY

Game Programming Today: The Problem of Concurrency

Why Concurrency?

- Xbox 360
 - 3 CPU cores, 6 hardware threads
 - 24-wide GPU
- PlayStation 3
 - 1 CPU core, 2 hardware threads
 - 7 SPU cores
 - 48-wide GPU
- PC
 - 1-4 CPU cores today, "Many Cores" soon

Intel Says: Future performance gains will come mainly from more cores, rather than higher clock rates

The C++/Java/C# Model: “Shared State Concurrency”

- We have some threads
- There is a bunch of shared state
- Any thread can modify any state at any time
- All synchronization is explicit, manual
- No compile-time verification of correctness properties:
 - Deadlock-free
 - Race-free
 - Invariants

Game Programming Today: Concurrency in Unreal Engine 3

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

Fine for 4 cores
What about 10's of cores?

How Hard is Concurrency?

- If it costs X (time, money, pain) to develop an efficient single-threaded algorithm, then...
 - Multithreaded version costs $2X$
 - PlayStation 3 Cell version costs $5X$
 - “GPGPU” version is generally unthinkable ($20X?$)
- $>2X$ will never be economical for a software company
- This is an argument against:
 - Cell – $5X$
 - GPGPU – $10X$
 - FPGA – $25X?$

Concurrency: Usage Cases

- Input/Output loop
- Graphics
- Numerical Algorithms
- Game World Simulation

Concurrency: Graphics

- Pixel processing
- Vertex processing
- Rasterization

Concurrency: Numerical Algorithms

Numerical Algorithms

- Collision Detection
- Physics Solver
- Path Finding
- Scene Traversal (rendering)
- Sound synthesis

Threading isn't too hard for algorithms
with simple data dependencies

Concurrency:

Game World Simulation

Gears of War game world simulation



Concurrency: Game World Simulation

Gears of War game simulation

- We have 1000+ interacting objects
- Instanced from 1000's of classes
- Each updated 30X per second
- Each object is controlled by complex object-oriented logic
 - Written by many different programmers
- Each object update potentially modifies itself and other objects

Game Programming Today: The Problem of Concurrency

Game World Simulation

Solutions?

- Manual locks and synchronization?
- Message-passing concurrency?
- Pure functional programming?
- Transactions?

These questions are applicable to concurrency in almost any Object-Oriented codebase.

Concurrency in Game World Simulation: Software Transactional Memory

See “Composable memory transactions”;
Harris, Marlow, Peyton-Jones, Herlihy

The idea:

- Update all objects concurrently in arbitrary order, with each update wrapped in an atomic {...} block
- With 10,000's of updates, and 5-10 objects touched per update, collisions will be low
- ~2-4X STM performance overhead is acceptable: if it enables our state-intensive code to scale to many threads, it's still a win

Claim: Transactions are the only plausible solution to large-scale concurrent mutable state

VECTORIZATION

VECTORIZATION

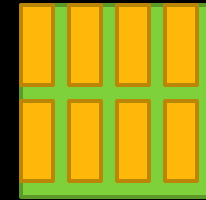
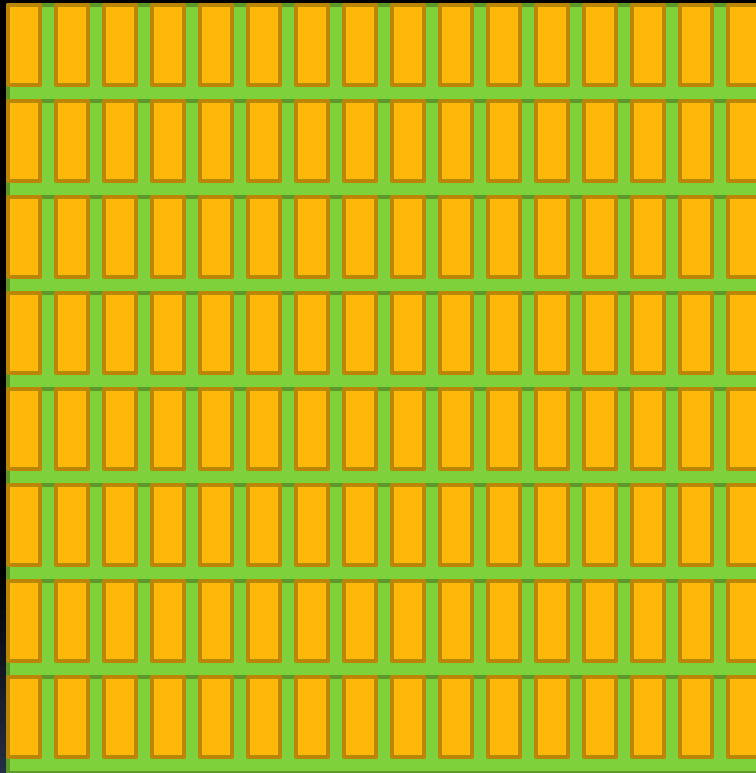
Why Vectorization?

Performance of GPU vs CPU

- NVIDIA GeForce 8800
 - 2 Ghz
 - 8 GPU cores
 - 16-wide vectors per core
 - Texture sampling with 1-cycle throughput
 - 512 GFLOPS
- Intel Core 2 Quad
 - 3 GHz
 - 4 Cores
 - 4-wide vectors (but a broken programming model)
 - 50 GFLOPS

GPU: 10X raw performance
100X texture sampling throughput

GPU Programming Today: Quantifying shader performance



Intel
Core 2 Duo

Nvidia
GeForce 8800

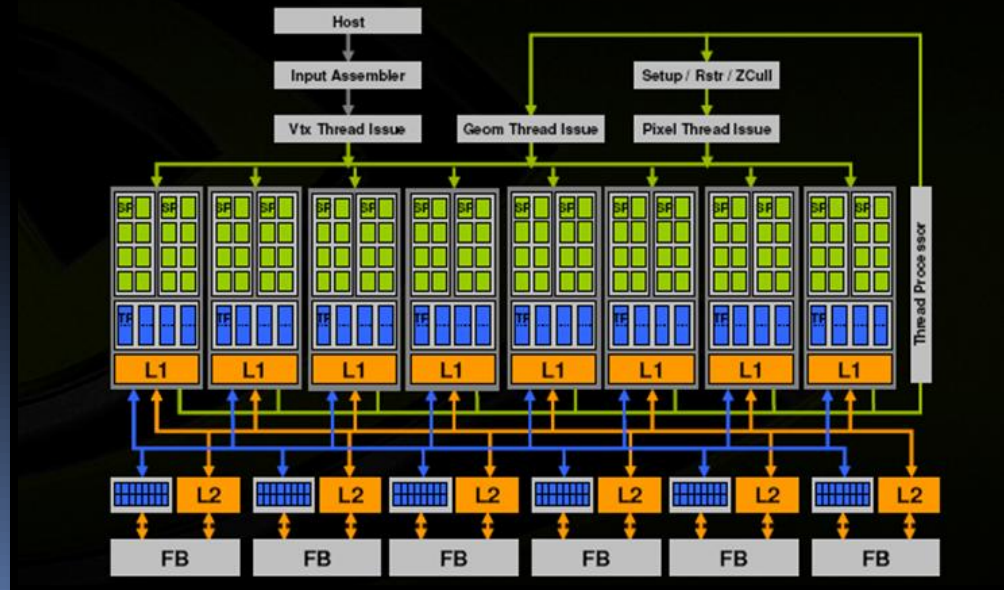
GPU: 10X raw performance, 100X texture throughput
...Why?

Convergence: Inevitability



Intel?

NVIDIA GeForce 8800



Future CPU/GPU Model

Hardware Model

- Three performance dimensions
 - Clock rate
 - Cores
 - Vector width
- Executes two sorts of code
 - Scalar code (just like x86, PowerPC)
 - Vector code (SSE, AltiVec – but better)
- Some fixed-function makes sense
 - Texture sampling
 - Rasterization

Future Programming Models: Vectorization

Or: Supporting “Wide Vectors” efficiently

Note: We’ll see two distinct, confusing uses of the word “vector” from here on

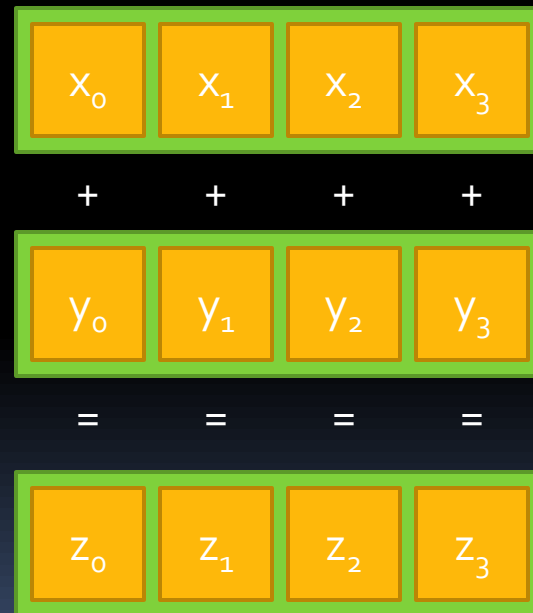
Future Programming Models: Vectorization

- “Old SIMD” (Single Instruction, Multiple Data)
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 SIMD” (Single Instruction, Multiple Data)
Intel SSE, Motorola AltiVec

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



Future Programming Models: Vectorization

- Data parallel SIMD (ATI, NVidia)
 - N-wide vectors
 - N-wide arithmetic
 - N is large
 - N=16 for NVidia GeForce 8800
 - N=48 for Xbox 360 ATI GPU
 - Data parallel loads/stores
 - Load N-wide vector register from scalars from N *independent* memory addresses
 - Analogous to register-indexed constant access in DirectX

Data Parallel SIMD > Traditional SIMD

- Old SIMD is generally only useful when dealing with “vector-like” data types:
 - 4-component XYZW vectors from graphics
 - 4x4 matrices
- Old SIMD is not “universal”
- Data Parallel SIMD is “universal”:
 - Any program fragment with a statically-known call graph free of sequential dependencies can be compiled into an equivalent N-wide data parallel program

Data Parallel SIMD “Universality”

```
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.

Data Parallel “Universality”

```
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!

Data Parallel SIMD “Universality”

```
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}  
  
    cmplx[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 Then & Now

- SSE/Altivec vectorization didn't pan out because the instruction set lacked:
 - Vectorized load/store instructions:
 - Masked vector load
 - Masked vector store
 - Sufficiently nice masking, swizzling, condition-extraction
- With the addition of those instructions, vectorization is “universal”

Impediments to Vectorization

- Statically unknown control flow
 - Loop-dependent function pointers
 - C++/Java virtual functions
- Sequential Dependencies
 - Calls to functions with side-effects
 - Potential pointer aliasing
 - Random-access memory *writes*

You can design a language that makes data parallelism much more natural: HLSL, Haskell, ML, NESL, ...

But you can't design "non-data parallelism" out of a general computing language. The constructs above are vital to many programming domains.

CORRECTNESS & SECURITY

Mainstream CPU Programming Today: The Problem of Correctness

Example (C#):

Given a vertex array and an index array, we read and transform the indexed vertices into a new array.

```
Vertex[] Transform (Vertex[] Vertices, int[] Indices, Matrix m)
{
    Vertex[] Result = new Vertex[Indices.length];
    for(int i=0; i<Indices.length; i++)
        Result[i] = Transform(m, Vertices[Indices[i]]);
    return Result;
};
```

What can possibly go wrong?

Mainstream CPU Programming Today: The Problem of Correctness

```
Vertex[] Transform (Vertex[] Vertices, int[] Indices, Matrix m)
{
    Vertex[] Result = new Vertex[Indices.length];
    for(int i=0; i<Indices.length; i++)
        Result[i] = Transform(m, Vertices[Indices[i]]);
    return Result;
};
```

May be NULL

May be NULL

May contain indices outside of the range of the Vertex array

May be NULL

Allocation can fail

Will the compiler realize this can't fail?

Could dereference a null pointer

Array access might be out of bounds

Mainstream CPU Programming Today: The Problem of Correctness

- Common failure cases that mainstream compilers can't statically detect:
 - NULL pointer access
 - Out-of-bounds array access
 - Memory leaks, dangling pointers (C++)
 - Ran out of memory / finite resources
 - Infinite loops
 - "Program fails to meet its specification"
- Today's work-arounds
 - Runtime crashes or corrupts resources (C++)
 - Runtime checking & exception (Java, C#)

Mainstream CPU Programming Today: The Problem of Correctness

- Some problems are fundamental to computing:
 - Finite resources
 - Potential infinite loops (viz. Turing completeness)
 - Program's specification is wrong
- But...

Mainstream CPU Programming Today: The Problem of Correctness

- Some problems are fundamental to computing:
 - Finite resources
 - Potential infinite loops (viz. Turing completeness)
 - Program's specification is wrong
- But...

The other problems are just
artifacts of crappy languages

Future Programming Models: Security & Correctness

Goal:

If the compiler doesn't beep, then your program really is correct, and safe.

(notwithstanding the halting problem, finite resources, and other *inherent* limits)

What we would like to write...

An index buffer containing natural numbers less than n

An array of exactly known size

Implicit Parameters

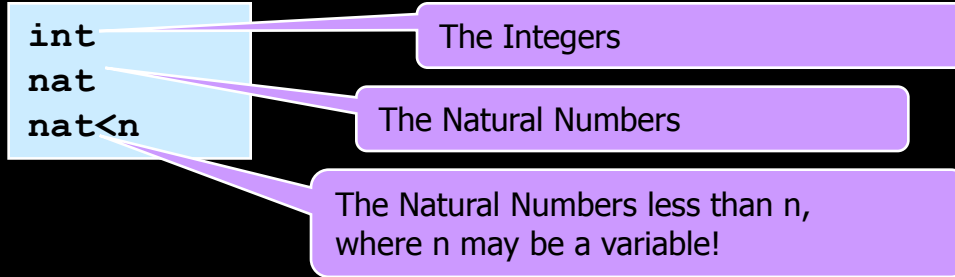
```
Transform{n:nat} (Vertices:[n]Vertex, Indices:[ ]nat<n,m:Matrix)
  : [n]Vertex =
  foreach(i in Indices)
    Transform(m,Vertices[i])
```

Iteration instead of explicit indexing

The only possible runtime failure is
out-of-memory

How might this work?

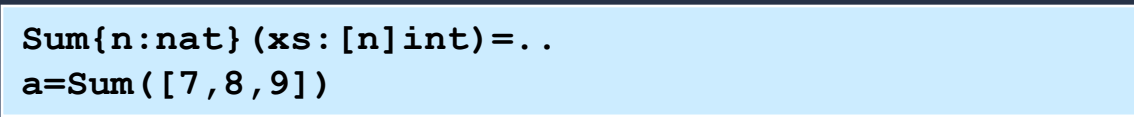
- Dependent types



- Dependent functions

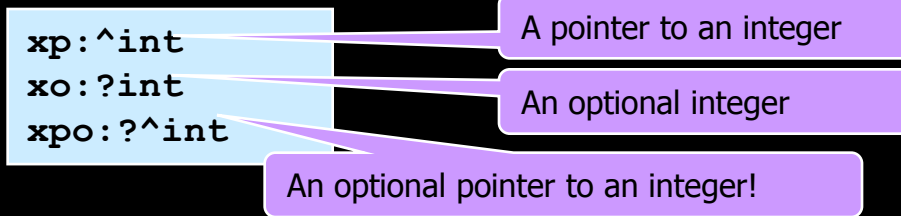


- Implicit Parameters (universal quantification)



How might this work?

- Separating the "pointer to t" concept from the "optional value of t" concept



- Comprehensions (a la Haskell), for safely traversing and generating collections

```
Successors(xs:[]int):[]int=  
  foreach(x in xs)  
    x+1
```

How might this work?

- A guarded casting mechanism for cases where need a safe "escape":

Here, we cast `i` to type of natural numbers bounded by the length of `as`, and bind the result to `n`

We can only access `i` within this context

If the cast fails, we execute the else-branch

```
GetElement(as:[]string, i:int):string=  
  if(n:nat<as.length=i)  
    as[n]  
  else  
    "Index Out of Bounds"
```

All potential failure must be explicitly handled, but we lose no expressiveness.

See Icon, Ontic for similar ideas

Aside: “Iteration” vs “Comprehension”

Sorting in Haskell: “Comprehension”

```
sort [] = []
sort (x:xs) = sort [y | y<-xs, y<x] ++
               [x] ++
               sort [y | y<-xs, y>=x]
```

- Simple
- Declarative
- No Side Effects
- Implicitly thread-parallel
- $O(n \cdot \log(n)/N)$ parallel performance

- Complicated
- Hopelessly sequential
- $O(n \cdot \log(n))$ performance

Sorting in C: “Iteration”

```
int partition(int y[], int f, int l);
void quicksort(int x[], int first, int last) {
    int pivIndex = 0;
    if(first < last) {
        pivIndex = partition(x,first, last);
        quicksort(x,first, (pivIndex-1));
        quicksort(x, (pivIndex+1),last);
    }
}

int partition(int y[], int f, int l) {
    int up,down,temp;
    int cc;
    int piv = y[f];
    up = f;
    down = l;
    do {
        while (y[up] <= piv && up < l) {
            up++;
        }
        while (y[down] > piv ) {
            down--;
        }
        if (up < down ) {
            temp = y[up];
            y[up] = y[down];
            y[down] = temp;
        }
    } while (down > up);
    temp = piv;
    y[f] = y[down];
    y[down] = piv;
    return down;
}
```

Aside:

“Iteration” vs “Comprehension”

Sorting in Haskell: “Comprehension”

```
sort []      = []
sort (x:xs) = sort [y | y<-xs, y<x] ++
                 [x      ] ++
                 sort [y | y<-xs, y>=x]
```

Sorting an empty array yields an empty array

Sorting an array beginning with the element x , followed by the elements xs yields ...

Generate a new array containing all elements y , with each element y sequentially in xs , Where $y \geq x$

Declarative programs are:

- Predictable
- Free of side effects
- Implicitly parallel

Future Programming Models: Uninitialized Data

- Can we make this work?

```
class MyClass
{
    const int a=c+1;
    const int b=7;
    const int c=b+1;
}
MyClass myvalue = new C; // What is myvalue.a?
```

This is a frequent bug. Data structures are often rearranged, changing the initialization order.

- Lessons from Haskell:
 - Lazy evaluation enables correct out-of-order evaluation
 - Accessing circularly entailed values causes thunk reentry (divergence), rather than just returning the wrong value
- Lesson from Id90: Lenient evaluation is sufficient to guarantee this

Future Programming Models: Avoiding Runtime failure

Solved problems (Java, C#):

- Random memory overwrites (no pointer arithmetic)
- Memory leaks (garbage collection)

Solveable:

- Accessing arrays out-of-bounds
- Dereferencing null pointers
- Integer overflow
- Accessing uninitialized variables

50% of the bugs in Unreal can be traced to these problems!

Avoiding Runtime Failure: Conclusion

Reasonable type-system extensions can statically eliminate all:

- Out-of-bounds array access
- Null pointer dereference
- Accessing of uninitialized variables
- Integer overflow

We should achieve this with a simple set of building blocks (option types, dependent types, references, ...) rather than all-encompassing abstractions like Java/C# "objects".

See Haskell for excellent implementation of:

- Comprehensions (for)
- Option types (Maybe)
- Non-NULL references (IORef, STRef)
- Out-of-order initialization

THE NEXT PROGRAMMING MODEL

Future Programming Models: The Essential Tradeoffs

- Developer Productivity
 - Programmers are expensive
 - Platforms with high development cost fail
- Performance
 - Scale to “lots of cores”
 - Scale to “wide vectors”
- Security, Correctness
 - The outside world is a hostile place
 - Users hate buggy software

Future Programming Models: Scaling to “Lots of Cores”

Or: Multithreading without the locking, deadlocks,
and race conditions.

Concurrency Solutions For Programming Languages

- Side-Effects Free Programming
a.k.a. Declarative Programming
 - Safe, implicit data parallelism
 - Safe, implicit thread parallelism
 - Practical for many numerical, computational algorithms
- Software Transactional Memory
 - Only viable solution for complex object-oriented systems, e.g. gameplay simulation

Language Issues: Concurrency

- Make concurrency the default
 - Natural thread parallelism
 - Natural data parallelism
 - Awkward sequentiality where necessary
- Implicit concurrency where possible
 - Explicit concurrency where necessary

There isn't "One True Concurrency Solution"

Obstacles to Concurrency & Vectorization

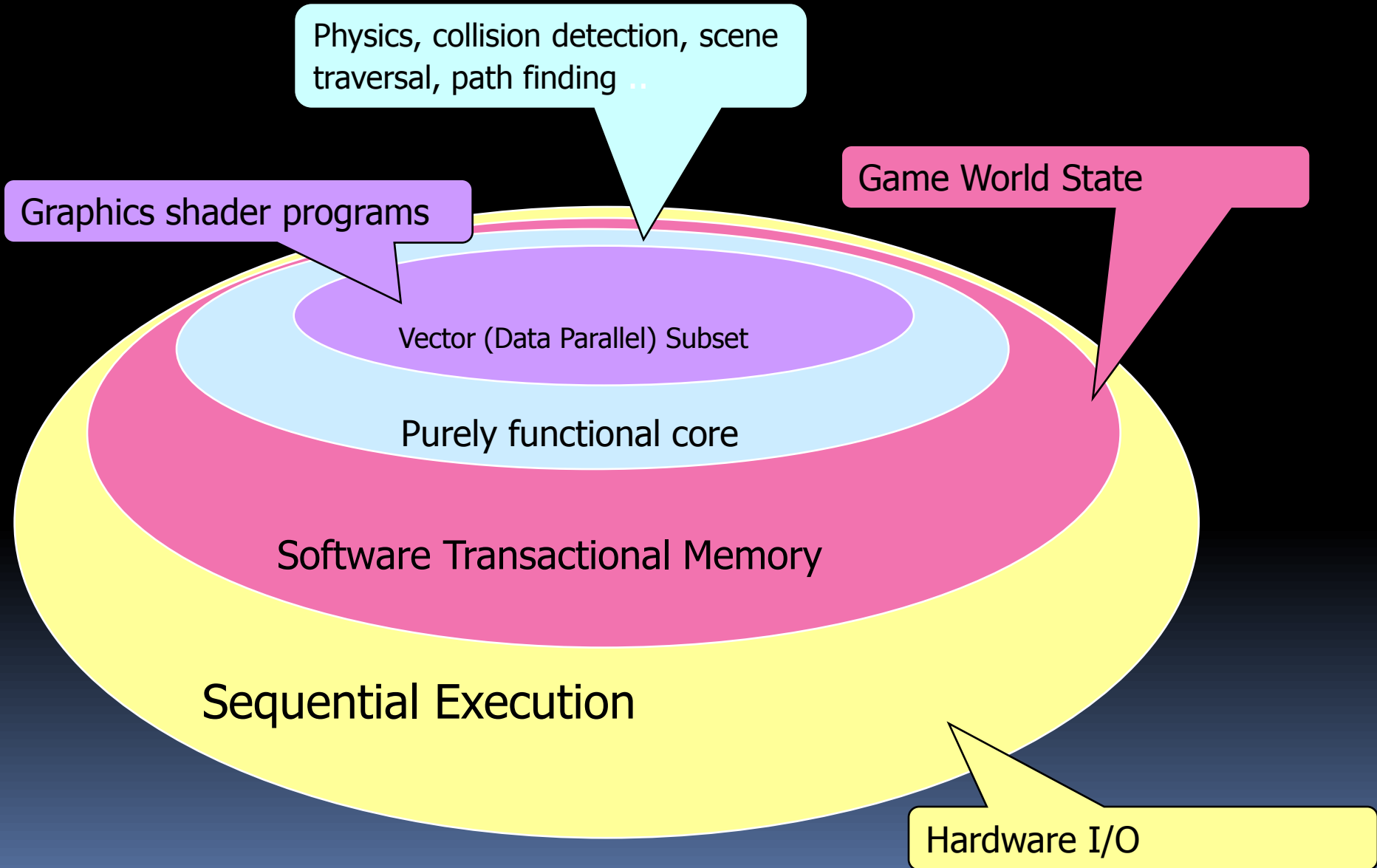
- Programming by Side Effect

```
int sum(int* elements,int count)
{
    int accumulator=0;
    for(int i=0; i<count; i++)
        accumulator+=elements[i];
    return accumulator;
}
```

- Pervasive Runtime Exceptions

```
for(i=0; i<1000; i++)
    x[i] = a[i] + b[i][0]
```

Concurrency & Data Parallelism: Summary



Future Programming Models: Summary

- Future programming languages will be very different
 - Not evolutionary like C++, Java, C#
 - Major assumptions must be revisited
 - To reduce use of mutable state
 - To expose thread- and data-parallelism
 - Programming idioms must change
 - Runtime Checks -> Compile-Time Checks
 - Side Effects -> Declarative Programming
 - Explicit Synchronization -> Transactions

Future Programming Models: The Ecosystem

- C/C++ will remain important
 - Many teams will be porting existing codebases
 - They will do concurrency “the hard way”
 - Expect language extensions for concurrency
 - OpenMP for improved thread concurrency
 - New extensions for improved (HLSL-like) data parallelism
- Multiple next-generation concurrent programming languages will be available in 2009
 - Concurrent Haskell / Data Parallel Haskell
 - Microsoft’s concurrent C#, C++ R&D work
 - The DARPA HPC languages
 - New contenders

If no “One True Language” emerges, developers will mix & match languages to get the job done

CONCLUSION

CONCLUSION

QUESTIONS / DISCUSSION

Keeping the Hardware Busy

"Computer architects place far too much emphasis on keeping the most expensive component of the system supplied with work at all times, at terrible expense to all other components of the system, not least its programmers."

- Edsger W. Dijkstra

On Architecture

"Systems reflect the organizations that build them"

- Conway's Law, "Soul of a New Machine"

Computing's Central Challenge

“Computing's central challenge is how not to make a mess of it”

- Edsger Dijkstra, 1972