The Next Mainstream Programming Language:

A Game Developer's Perspective

Tim Sweeney
Epic Games

Outline

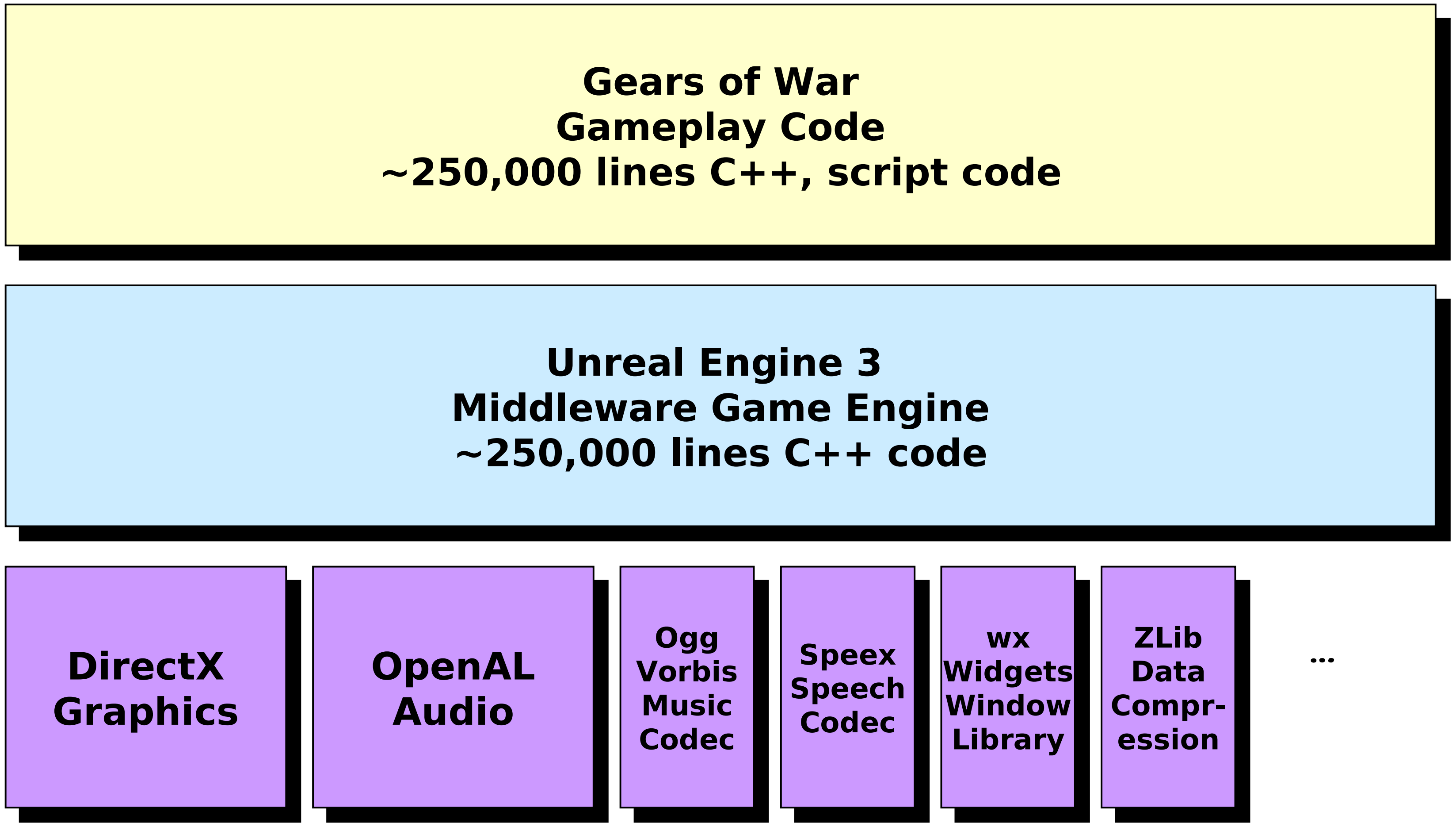
- Game Development
  - Typical Process
- What's in a game?
  - Game Simulation
  - Numeric Computation
  - Shading
- Where are today's languages failing?
  - Concurrency
  - Reliability

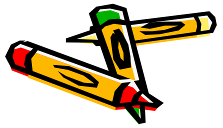
Game Development

Game Development: Gears of War

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

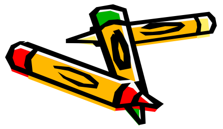
Software Dependencies





Game Development: Platforms

- The typical Unreal Engine 3 game will ship on:
  - Xbox 360
  - PlayStation 3
  - Windows
- Some will also ship on:
  - Linux
  - MacOS



What's in a game?

The obvious:

- Rendering
- Pixel shading
- Physics simulation, collision detection
- Game world simulation
- Artificial intelligence, path finding

But it's not just fun and games:

- Data persistence with versioning, streaming
- Distributed Computing (multiplayer game simulation)
- Visual content authoring tools
- Scripting and compiler technology
- User interfaces

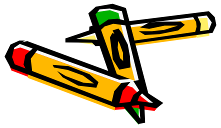
Three Kinds of Code

- Gameplay Simulation
- Numeric Computation
- Shading

Gameplay Simulation

Gameplay Simulation

- Models the state of the game world as interacting objects evolve over time
- High-level, object-oriented code
- Written in C++ or scripting language
- Imperative programming style
- Usually garbage-collected



Gameplay Simulation – The Numbers

- 30-60 updates (frames) per second
- ~1000 distinct gameplay classes
  - Contain imperative state
  - Contain member functions
  - Highly dynamic
- ~10,000 active gameplay objects
- Each time a gameplay object is updated, it typically touches 5-10 other objects

Numeric Computation

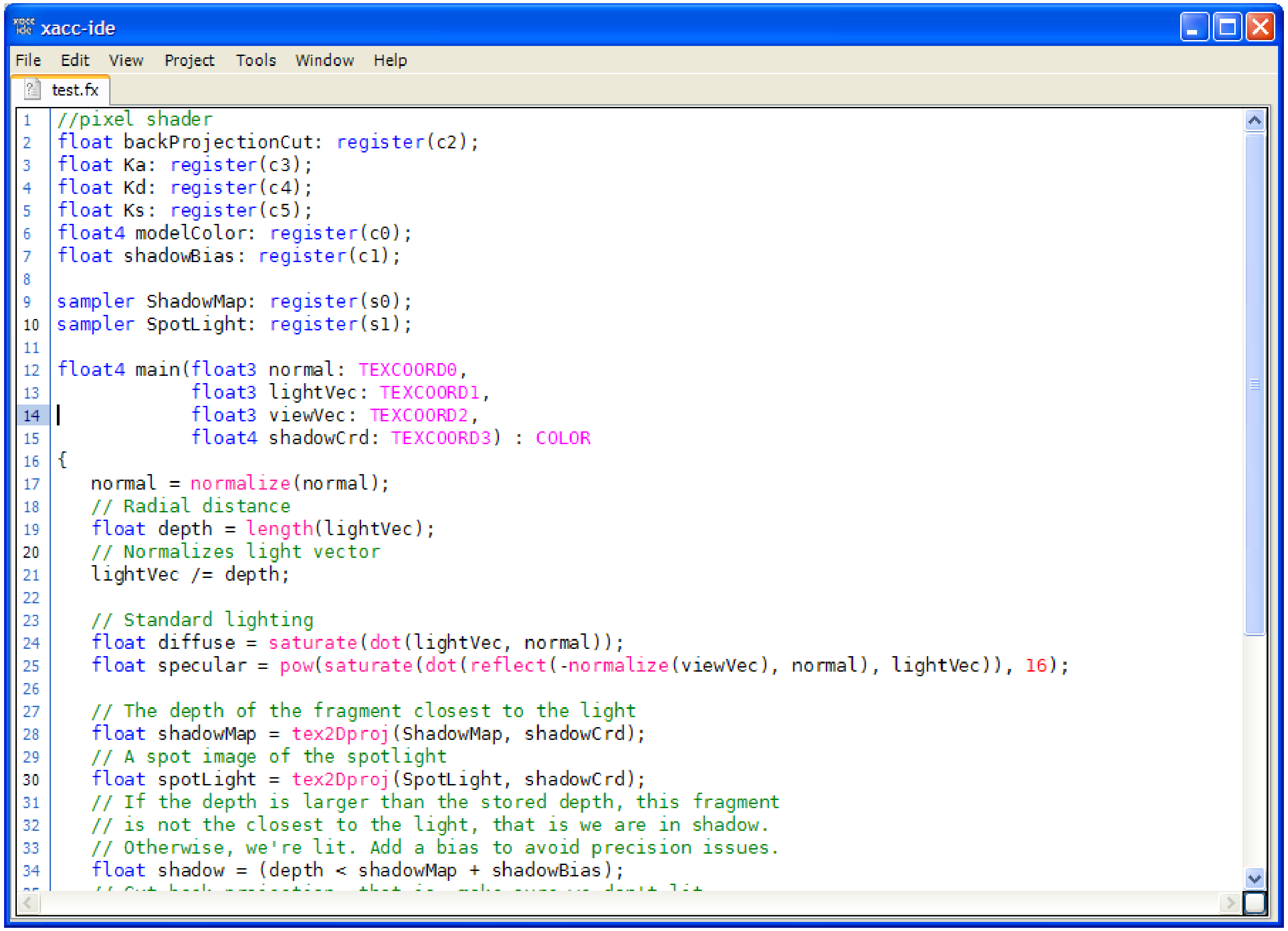
- Algorithms:
  - Scene graph traversal
  - Physics simulation
  - Collision Detection
  - Path Finding
  - Sound Propagation
- Low-level, high-performance code
- Written in C++ with SIMD intrinsics
- Essentially functional
  - Transforms a small input data set to a small output data set, making use of large constant data structures.

Shading

Shading

- Generates pixel and vertex attributes
- Written in HLSL/CG shading language
- Runs on the GPU
- Inherently data-parallel
  - Control flow is statically known
  - "Embarassingly Parallel"
  - Current GPU's are 16-wide to 48-wide!

Shading in HLSL



```
//pixel shader
float backProjectionCut: register(c2);
float Ka: register(c3);
float Kd: register(c4);
float Ks: register(c5);
float4 modelColor: register(c0);
float shadowBias: register(c1);

sampler ShadowMap: register(s0);
sampler SpotLight: register(s1);

float4 main(float3 normal: TEXCOORD0,
            float3 lightVec: TEXCOORD1,
            float3 viewVec: TEXCOORD2,
            float4 shadowCrd: TEXCOORD3) : COLOR
{
   normal = normalize(normal);
   // Radial distance
   float depth = length(lightVec);
   // Normalizes light vector
   lightVec /= depth;

   // Standard lighting
   float diffuse = saturate(dot(lightVec, normal));
   float specular = pow(saturate(dot(reflect(-normalize(viewVec), normal), lightVec)), 16);

   // The depth of the fragment closest to the light
   float shadowMap = tex2Dproj(ShadowMap, shadowCrd);
   // A spot image of the spotlight
   float spotLight = tex2Dproj(SpotLight, shadowCrd);
   // If the depth is larger than the stored depth, this fragment
   // is not the closest to the light, that is we are in shadow.
   // Otherwise, we're lit. Add a bias to avoid precision issues.
   float shadow = (depth < shadowMap + shadowBias);
   // Cut back projection, that is, make sure we don't lit
```

Shading – The Numbers

- Game runs at 30 FPS @ 1280x720p
- ~5,000 visible objects
- ~10M pixels rendered per frame
  - Per-pixel lighting and shadowing requires multiple rendering passes per object and per-light
- Typical pixel shader is ~100 instructions long
- Shader FPU's are 4-wide SIMD
- ~500 GFLOPS compute power

Three Kinds of Code

	Game Simulation	Numeric Computation	Shading
Languages	C++, Scripting	C++	CG, HLSL
CPU Budget	10%	90%	n/a
Lines of Code	250,000	250,000	10,000
FPU Usage	0.5 GFLOPS	5 GFLOPS	500 GFLOPS

What are the hard problems?

- Performance
  - When updating 10,000 objects at 60 FPS, everything is performance-sensitive
- Modularity
  - Very important with ~10-20 middleware libraries per game
- Reliability
  - Error-prone language / type system leads to wasted effort finding trivial bugs
  - Significantly impacts productivity
- Concurrency
  - Hardware supports 6-8 threads
  - C++ is ill-equipped for concurrency

Performance

Performance

- When updating 10,000 objects at 60 FPS, everything is performance-sensitive
- But:
  - Productivity is just as important
  - Will gladly sacrifice 10% of our performance for 10% higher productivity
  - We never use assembly language
- There is not a simple set of “hotspots” to optimize!

That’s all!

Modularity

Unreal's game framework

Gameplay module

Base class of gameplay objects

Members

```
package UnrealEngine;

class Actor
{
   int Health;
   void TakeDamage(int Amount)
   {
        Health = Health – Amount;
        if (Health<0)
               Die();
   }
}

class Player extends Actor
{
   string PlayerName;
   socket NetworkConnection;
}
```

Game class hierarchy

Generic Game Framework

```
Actor
    Player
    Enemy
    InventoryItem
        Weapon
```

Game-Specific Framework Extension

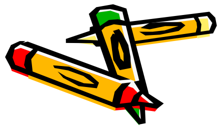
```
Actor
Player
    Enemy
        Dragon
        Troll
    InventoryItem
        Weapon
            Sword
            Crossbow
```

Software Frameworks

- The Problem:
  Users of a framework
  need to extend the functionality
  of the framework's base classes!

- The workarounds:
  - Modify the source

    …and modify it again with each new version
  - Add references to payload classes, and dynamically cast them at runtime to the appropriate types.



Software Frameworks

- The Problem:
  Users of a framework
  want to extend the functionality
  of the framework's base classes!

- The workarounds:
  - Modify the source
    …and modify it again with each new version
  - Add references to payload classes, and dynamically cast them at runtime to the appropriate types.
  - These are all error-prone:
    Can the compiler help us here?

What we would like to write…

Base Framework

```
package Engine;

class Actor
{
    int Health;
    …
}
class Player extends Actor
{
    …
}
class Inventory extends Actor
{
    …
}
```

Extended Framework

```
Package GearsOfWar extends Engine;

class Actor extends Engine.Actor
{
    // Here we can add new members
    // to the base class.
    …
}
class Player extends Engine.Player
{
    // Thus virtually inherits from
    // GearsOfWar.Actor
    …
}
class Gun extends GearsOfWar.Inventory
{
    …
}
```

The basic goal:

To extend an entire software framework's class hierarchy in parallel, in an open-world system.

Reliability

Or:

If the compiler doesn't beep, my program should work

Dynamic Failure in Mainstream Languages

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?

Dynamic Failure in Mainstream Languages

May be NULL

May be NULL

May contain indices outside of the range of the Vertex array

May be NULL

```
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;
};
```

Will the compiler realize this can't fail?

Could dereference a null pointer

Array access might be out of bounds

Our code is littered with runtime failure cases,

Yet the compiler remains silent!

Dynamic Failure in Mainstream Languages

Solved problems:

- Random memory overwrites
- Memory leaks

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!

What we would like to write…

An index buffer containing natural numbers less than n

An array of exactly known size

Universally quantify over all natural numbers

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

Haskell-style array comprehension

The only possible failure mode:

divergence, if the call to Transform diverges.

How might this work?

- Dependent types

```
int
nat
nat<n
```

The Integers

The Natural Numbers

The Natural Numbers less than n, where n may be a variable!

- Dependent functions

```
Sum(n:nat,xs:[n]int)=..
a=Sum(3,[7,8,9])
```

Explicit type/value dependency between function parameters

- Universal quantification

```
Sum{n:nat}(xs:[n]int)=..
a=Sum([7,8,9])
```

How might this work?

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

```
xp:^int
xo:?int
xpo:?^int
```

A pointer to an integer

An optional integer

An optional pointer to an integer!

- 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":

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

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

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

Analysis of the Unreal code

- Usage of integer variables in Unreal:
  - 90% of integer variables in Unreal exist to index into arrays
    - 80% could be dependently-typed explicitly, guaranteeing safe array access without casting.
    - 10% would require casts upon array access.
  - The other 10% are used for:
    - Computing summary statistics
    - Encoding bit flags
    - Various forms of low-level hackery
- "For" loops in Unreal:
  - 40% are functional comprehensions
  - 50% are functional folds

Accessing uninitialized variables

- 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

Dynamic Failure: Conclusion

Reasonable type-system extensions could statically eliminate all:

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

See Haskell for excellent implementation of:

- Comprehensions
- Option types via Maybe
- Non-NULL references via IORef, STRef
- Out-of-order initialization

Integer overflow

The Natural Numbers

```
data Nat = Zero | Succ Nat
```

Factoid: C# exposes more than 10 integer-like data types, none of which are those defined by (Pythagoras, 500BC).

In the future, can we get integers right?

Can we get integers right?

Neat Trick:

- In a machine word (size 2^n), encode an integer $\pm 2^{n-1}$ or a pointer to a variable-precision integer
- Thus "small" integers carry no storage cost
- Additional access cost is ~5 CPU instructions

But:

- A natural number bounded so as to index into an active array is guaranteed to fit within the machine word size (the array is the proof of this!) and thus requires no special encoding.
- Since ~80% of integers can dependently-typed to access into an array, the amortized cost is ~1 CPU instruction per integer operation.

This could be a viable tradeoff

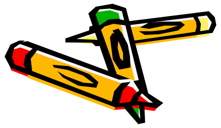
Concurrency

The C++/Java/C# Model: "Shared State Concurrency"

- The Idea:
  - 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

The C++/Java/C# Model: "Shared State Concurrency"

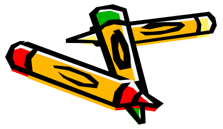
- This is hard!
- How we cope in Unreal Engine 3:
  - 1 main thread responsible for doing all work we can't hope to safely multithread
  - 1 heavyweight rendering thread
  - A pool of 4-6 helper threads
    - Dynamically allocate them to simple tasks.
  - "Program Very Carefully!"
- Huge productivity burden
- Scales poorly to thread counts



There must be a better way!

Three Kinds of Code: Revisited

- Gameplay Simulation
  - Gratuitous use of mutable state
  - 10,000's of objects must be updated
  - Typical object update touches 5-10 other objects
- Numeric Computation
  - Computations are purely functional
  - But they use state locally during computations
- Shading
  - Already implicitly data parallel



Concurrency in Shading

- Look at the solution of CG/HLSL:
  - New programming language aimed at "Embarassingly Parallel" shader programming
  - Its constructs map naturally to a data-parallel implementation
  - Static control flow (conditionals supported via masking)

Concurrency in Shading

Conclusion: The problem of *data-parallel* concurrency is effectively solved(!)

"Proof": Xbox 360 games are running with 48-wide data shader programs utilizing half a Teraflop of compute power...

Concurrency in Numeric Computation

- These are essentially pure functional algorithms, but they operate locally on mutable state
- Haskell ST, STRef solution enables encapsulating local heaps and mutability within referentially-transparent code
- These are the building blocks for implicitly parallel programs
- Estimate ~80% of CPU effort in Unreal can be parallelized this way

In the future, we will write these algorithms using referentially-transparent constructs.

Numeric Computation Example: Collision Detection

A typical 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);
```

```
Vec3   = data Vec3 float float float
Hit    = data Hit float Vec3
collide :: (vec3,vec3)->Maybe Hit
```

Numeric Computation Example: Collision Detection

- Since `collisionCheck` is effects-free, it may be executed in parallel with any other effects-free computations.
- Basic idea:
  - The programmer supplies effect annotations to the compiler.
  - The compiler verifies the annotations.

```
collide(start:Vec3,end:Vec3):?Hit

print(s:string)[#imperative]:void
```

A pure function (the default)

Effectful functions require explicit annotations

  - Many viable implementations (Haskell's Monadic effects, effect typing, etc)

In a concurrent world, imperative is the wrong default!

Concurrency in Gameplay Simulation

This is the hardest problem…

- 10,00's of objects
- Each one contains mutable state
- Each one updated 30 times per second
- Each update touches 5-10 other objects

Manual synchronization (shared state concurrency)
is
hopelessly intractible here.

Solutions?

- Rewrite as referentially-transparent functions?
- Message-passing concurrency?
- Continue using the sequential, single-threaded approach?

Concurrency in Gameplay Simulation: Software Transactional Memory

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

The idea:

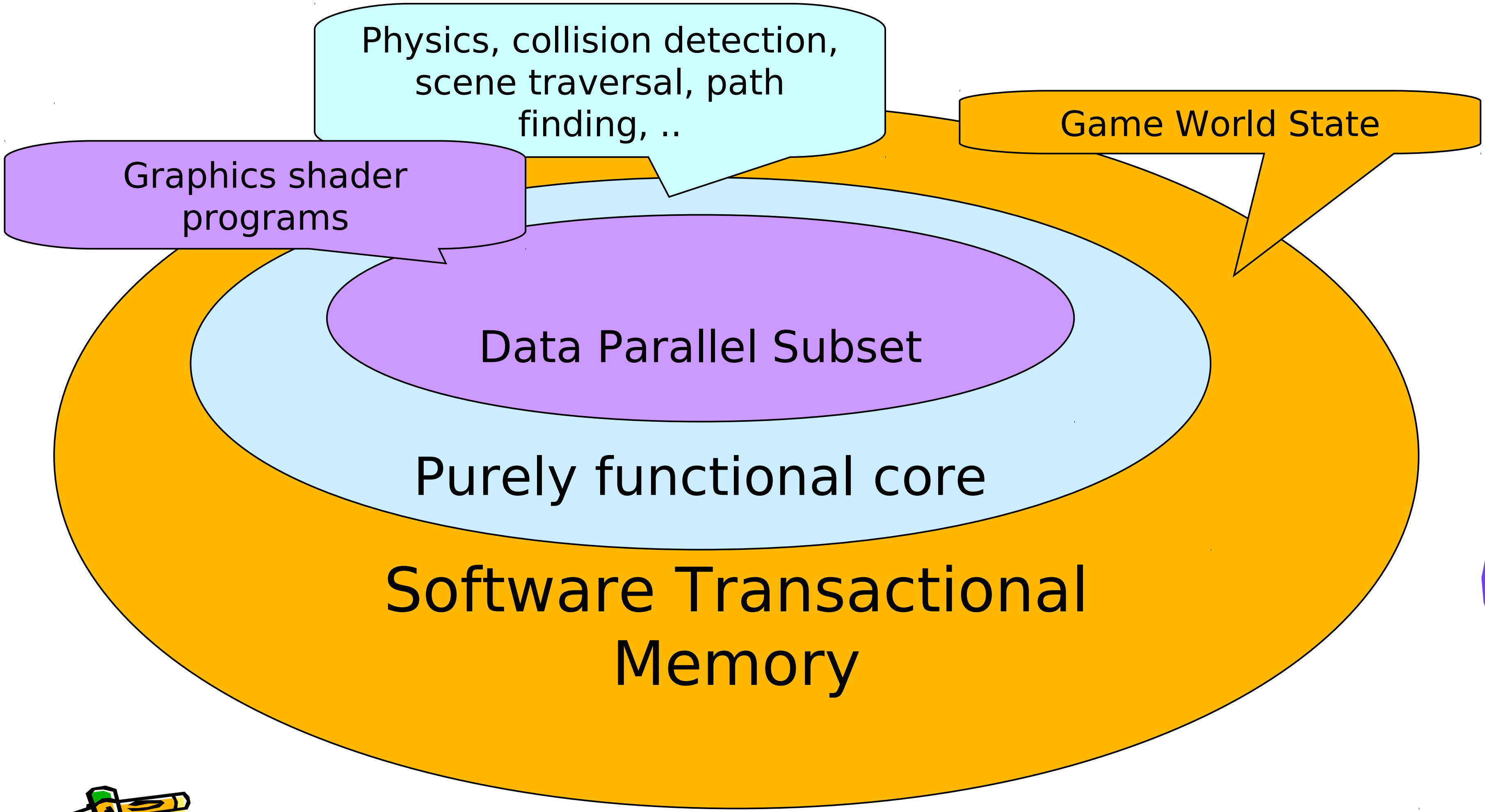
- Update all objects concurrently in arbitrary order, with each update wrapped in an atomic {...} block
- With 10,000's of updates, and 5-10 objects touched per update, collisions will be low
- ~2-4X STM performance overhead is acceptable: if it enables our state-intensive code to scale to many threads,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

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

Three Kinds of Code: Revisited

	Game Simulation	Numeric Computation	Shading
Languages	C++, Scripting	C++	CG, HLSL
CPU Budget	10%	90%	n/a
Lines of Code	250,000	250,000	10,000
FPU Usage	0.5 GFLOPS	5 GFLOPS	500 GFLOPS
Parallelism	**Software Transactional Memory**	**Implicit Thread Parallelism**	**Implicit Data Parallelism**

Parallelism and purity


Physics, collision detection, scene traversal, path finding, ..
Game World State
Graphics shader programs
Data Parallel Subset
Purely functional core
Software Transactional Memory

Musings

On the Next Maintream Programming Language

Musings

There is a wonderful correspondence between:

- Features that aid reliability
- Features that enable concurrency.

Example:

- Outlawing runtime exceptions through dependent types
  - Out of bounds array access
  - Null pointer dereference
  - Integer overflow

  Exceptions impose sequencing constraints on concurrent execution.

Dependent types and concurrency must evolve simultaneously

Language Implications

Evaluation Strategy

- Lenient evaluation is the right default.
- Support lazy evaluation through explicit suspend/evaluate constructs.
- Eager evaluation is an optimization the compiler may perform when it is safe to do so.

Language Implications

Effects Model

- Purely Functional is the right default
- Imperative constructs are vital features that must be exposed through explicit effects-typing constructs
- Exceptions are an effect

Why not go one step further and define partiality as an effect, thus creating a foundational language subset suitable for proofs?

Performance – Language Implications

Memory model

- Garbage collection should be the only option

Exception Model

- The Java/C# "exceptions everywhere" model should be wholly abandoned
  - All dereference and array accesses must be statically verifyable, rather than causing sequenced exceptions
- No language construct except "throw" should generate an exception

Syntax

Requirement:

- Must not scare away mainstream programmers.
- Lots of options.

```
int f{nat n}(int[] as,natrange<n> i)
{
    return as[i];
}
```

C Family: Least scary, but it's a messy legacy

```
f :: forall n::nat. ([int],nat<n) -> int
f (xs,i) = xs !! i
```

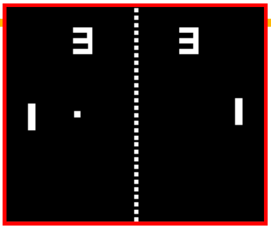
Haskell family: Quite scary :-)

```
f{n:nat}(as:[]int,i:nat<n)=as[i]
```

Pascal/ML family: Seems promising

Conclusion

A Brief History of Game Technology



- 1972 Pong (hardware)

- 1980 Zork (high level interpreted language)

- 1993 DOOM (C)

- 1998 Unreal (C++, Java-style scripting)

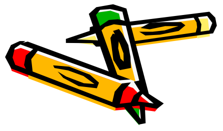
- 2005-6 Xbox 360, PlayStation 3 with 6-8 hardware threads

- 2009 Next console generation. Unification of the CPU, GPU. Massive multi-core, data parallelism, etc.

The Coming Crisis in Computing

- By 2009, game developers will face…
- CPU's with:
  - 20+ cores
  - 80+ hardware threads
  - >1 TFLOP of computing power
- GPU's with general computing capabilities.
- Game developers will be at the forefront.
- If we are to program these devices productively, you are our only hope!

Questions?



Backup Slides

The Genius of Haskell

- Algebraic Datatypes
  - Unions done right
    Compare to: C unions, Java union-like class hierarchies
  - Maybe t
    C/Java option types are coupled to pointer/reference types
- IO, ST
  - With STRef, you can write a pure function that uses heaps and mutable state locally, verifyably guaranteeing that those effects remain local.

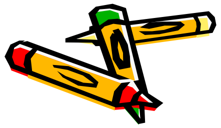
The Genius of Haskell

Sorting in Haskell

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

Sorting in C

```
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;
}
```



Why Haskell is Not My Favorite Programming Language

- The syntax is … scary
- Lazy evaluation is a costly default
  - But eager evaluation is too limiting
  - Lenient evaluation would be an interesting default
- Lists are the syntactically preferred sequence type
  - In the absence of lazy evaluation, arrays seem preferable

Why Haskell is Not My Favorite Programming Language

- Type inference doesn't scale
  - To large hierarchies of open-world modules
  - To type system extensions
  - To system-wide error propagation

```
f(x,y) = x+y
a=f(3,"4")
```

…

```
ERROR - Cannot infer instance
*** Instance   : Num [Char]
*** Expression : f (3,"4")
```

???

```
f(int x,int y) = x+y
a=f(3,"4")
```

…

```
Parameter mismatch paremter 2 of call to f:
   Expected: int
   Got:      "4"
```