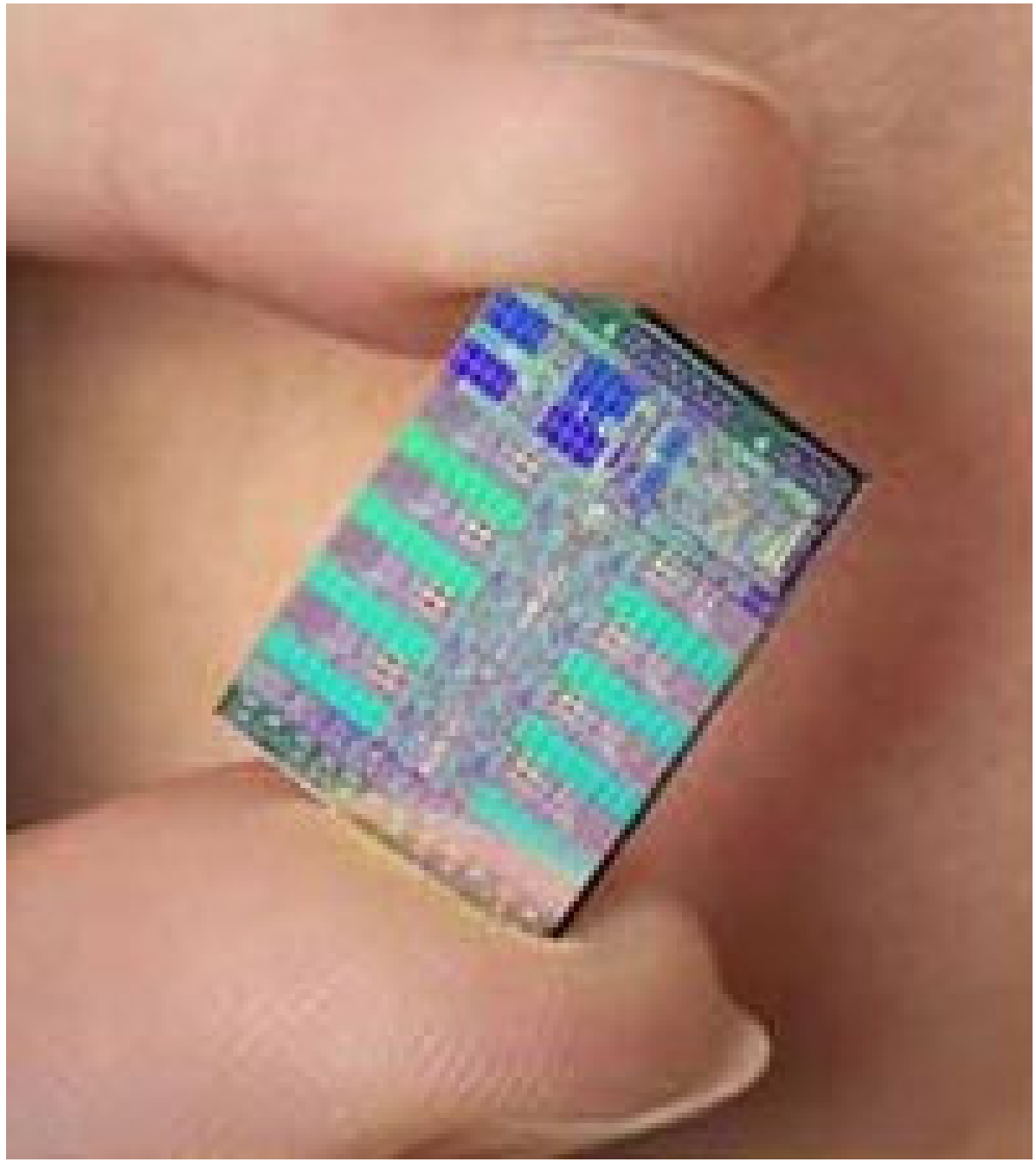
Developing Software for the PS3/Cell Platform

Mike Acton
macton@insomniacgames.com

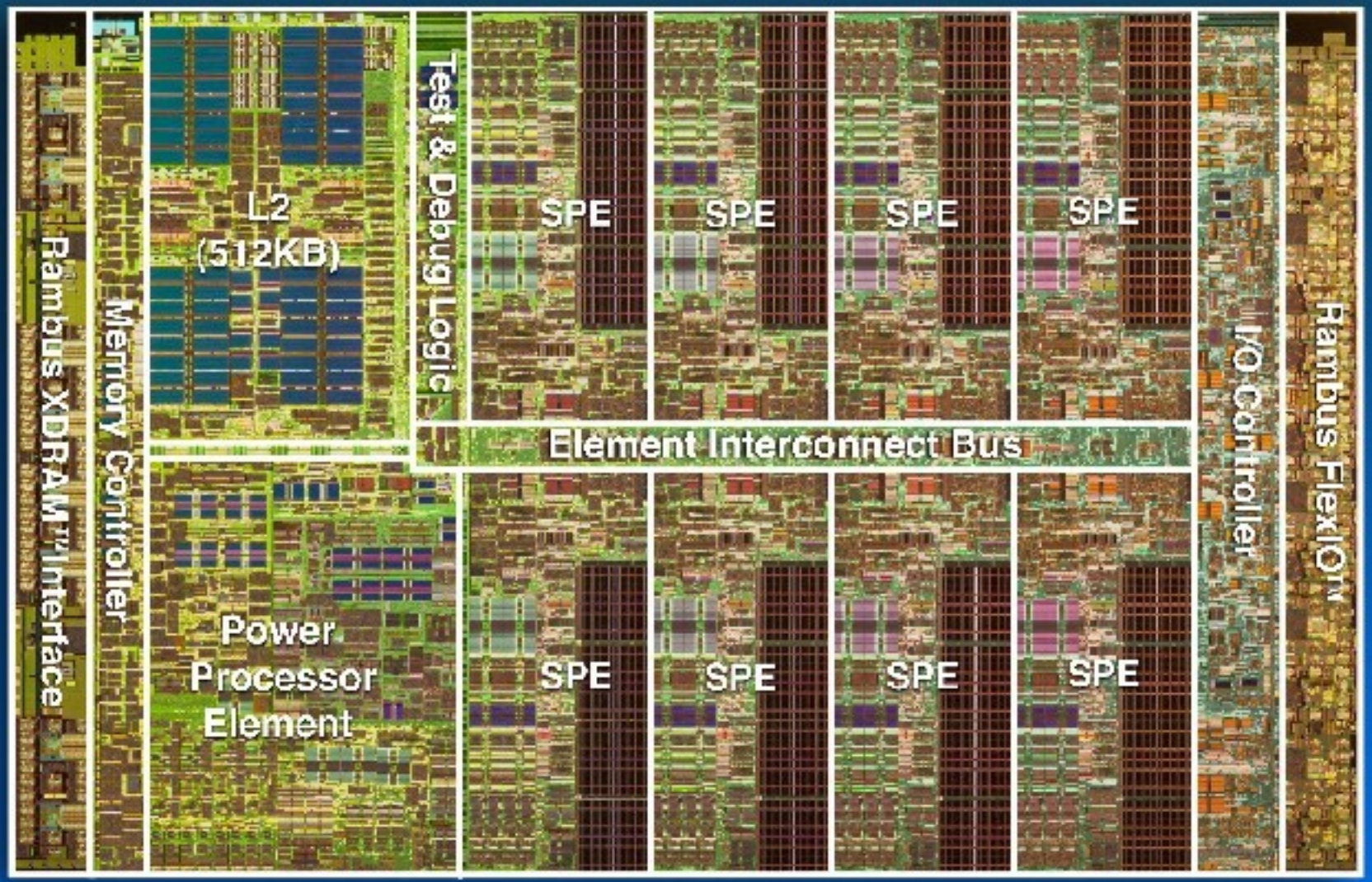

PLAYSTATION 3
RATCHET & CLANK FUTURE
TOOLS OF DESTRUCTION
RATING PENDING
RP
CONTENT RATED BY ESRB
PLAYSTATION 3
RATCHET & CLANK
QUEST FOR BOOTY
7+
www.pegi.info
INSOMNIAC GAMES
SONY
Only On PlayStation.
PLAYSTATION Network
RESISTANCE 2
MATURE 17+
M
CONTENT RATED BY ESRB
INSOMNIAC GAMES
SONY
DAILY GAME

The Cell Processor

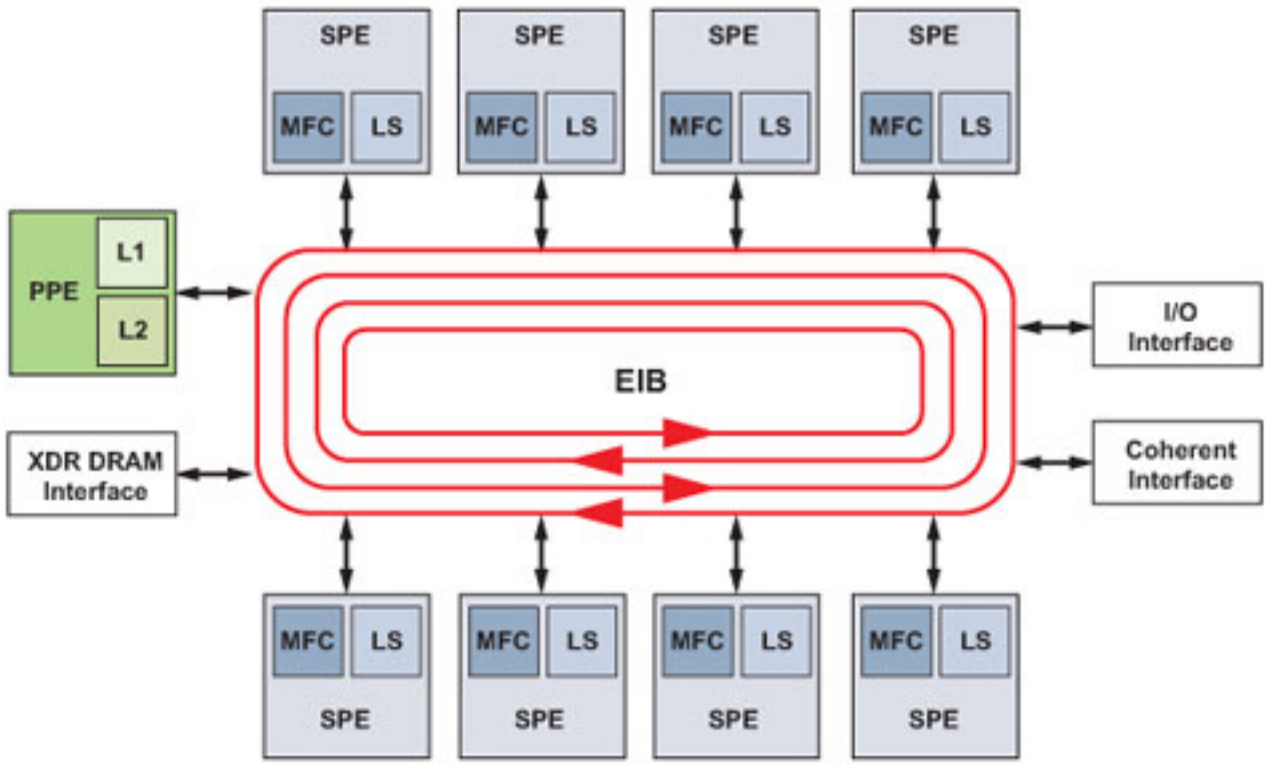
A quick introduction to the hardware

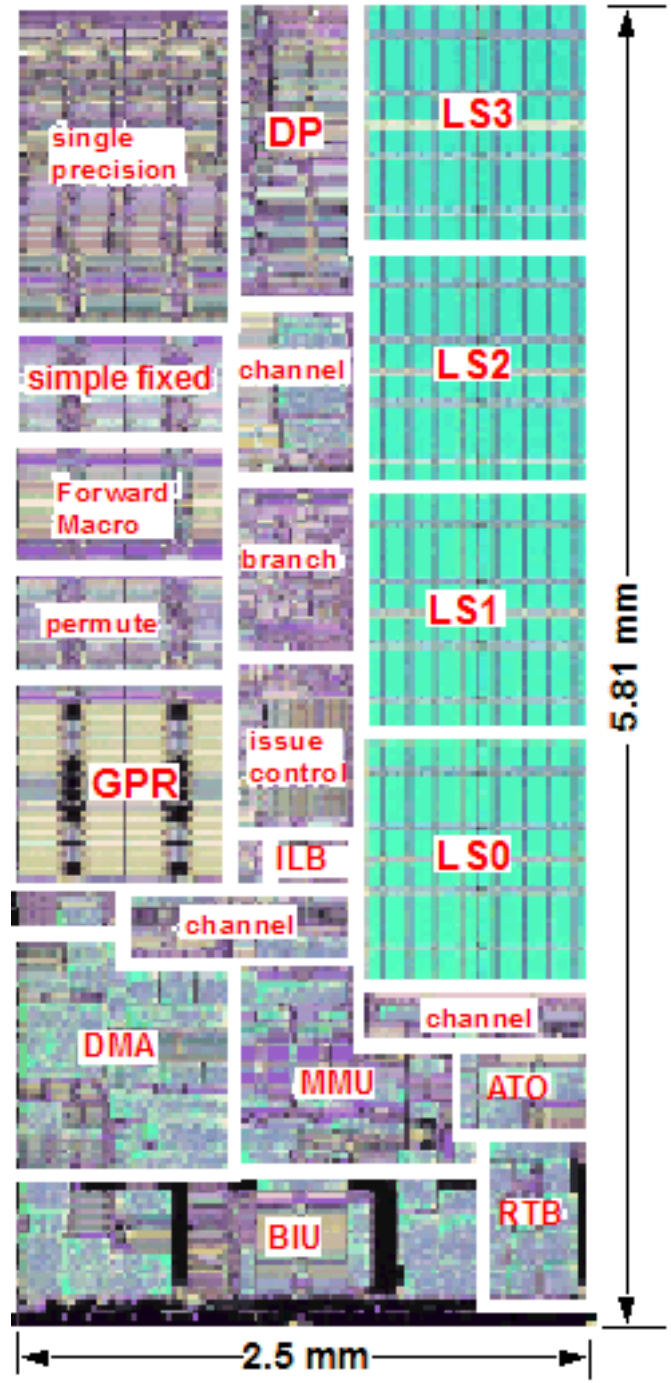


Cell Broadband Engine Processor



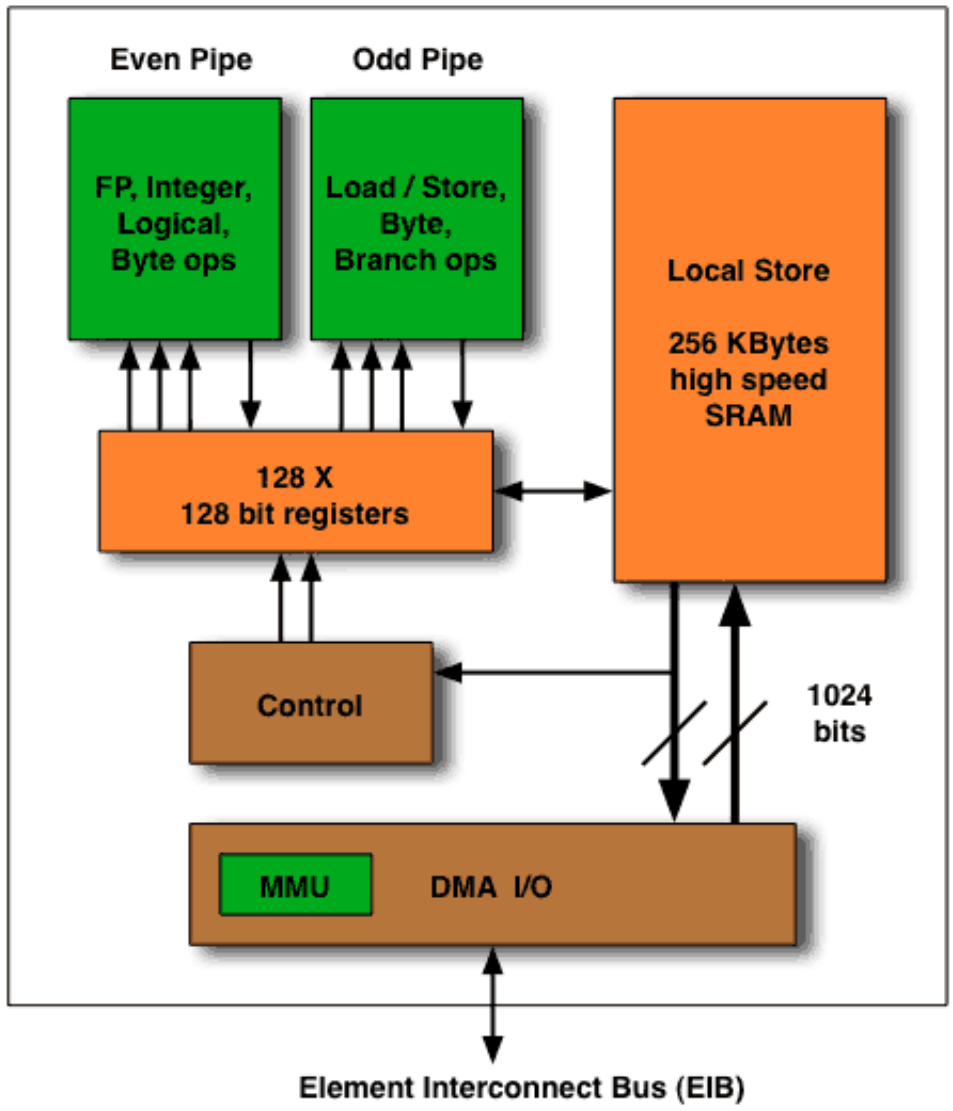
IBM


SPE
MFC
LS
SPE
MFC
LS
SPE
MFC
LS
SPE
MFC
LS
PPE
L1
L2
EIB
I/O
Interface
XDR DRAM
Interface
Coherent
Interface
MFC
LS
SPE
MFC
LS
SPE
MFC
LS
SPE
MFC
LS
SPE


single precision
DP
LS3
simple fixed
channel
LS2
Forward Macro
branch
LS1
permute
issue control
GPR
LS0
ILB
channel
channel
DMA
MMU
ATO
BIU
RTB
5.81 mm
2.5 mm

Cell SPE Architecture

Each SPE is an independent vector CPU
capable of 32 GFLOPs or 32 GOPs (32 bit @ 4GHz.)



The Cell is...

- ... not a magic bullet.
- ... not a radical change in high-performance design.
- ... fun to program for!

Programming for Games

Quick background of game development

Performance

Mostly "soft" real-time
(60Hz or 30Hz)

Languages and Compilers

Other Processers and I/O

GPU, Blu-ray, HDD, Peripherals, Network

Many Assets

Art, Animation, Audio

"Game" vs. "Engine" code

Divisions of development

Our Approach to *the Cell* for games

What does it change?

WARNING!

“Manual Solution” “Very optimized codes but at cost of programmability."

Marc Gonzales, IBM

Good solutions for the Cell will be good solutions on other platforms.

High-performance code is
easy to
port to the Cell.

Poorly performing code from
any
platform is hard to port.

Data and code optimization are *merely important* on conventional architectures

... On the Cell, they're critical.

Common Complaints

#1 "But, it's too hard!"

Multi-processing is not new

- Trouble with the SPUs usually is just trouble with multi-core.
- You can't wish multi-core programming away.
- It's part of the job.

"Cache and DMA data design too complex"

Enforced simplicity

Not that different from calling memcpy

SPU DMA vs. PPU memcpy

SPU DMA

DMA from main ram to local store

```
wrch    $ch16, ls_addr
wrch    $ch18, main_addr
wrch    $ch19, size
wrch    $ch20, dma_tag
il      $2,    MFC_GET_CMD
wrch    $ch21, $2
```

PPU Memcpy

PPU memcpy from far ram to near ram

```
mr $3, near_addr
mr $4, far_addr
mr $5, size
bl     memcpy
```

DMA from local store to main ram

```
wrch    $ch16, ls_addr
wrch    $ch18, main_addr
wrch    $ch19, size
wrch    $ch20, dma_tag
il      $2,    MFC_PUT_CMD
wrch    $ch21, $2
```

PPU memcpy from near ram to far ram

```
mr $4, near_addr
mr $3, far_addr
mr $5, size
bl     memcpy
```

Conclusion: If you can call memcpy, you can DMA data.

But you get extra control

SPU Synchronization

Example Sync

```
DMA from main ram to local store
```

```
Do other productive work while DMA
is happening...
```

```
(Sync) Wait for DMA to complete

il      $2,    1
shl     $2,    $2,  dma_tag
wrch    $ch22, $2
il      $3,    MFC_TAG_UPDATE_ALL
wrch    $ch23, $3
rdch    $2,    $ch24
```

Fence: Transfer after previous with the same tag

```
PUTF  Transfer previous before this PUT
PUTLF Transfer previous before this PUT LIST
GETF  Transfer previous before this GET
GETLF Transfer previous before this GET LIST
```

Barrier: Transfer after previous and before next with the same tag

```
PUTB  Fixed order with respect to this PUT
PUTLB Fixed order with respect to this PUT LIST
GETB  Fixed order with respect to this GET
GETLB Fixed order with respect to this GET LIST
```

Lock Line Reservation

```
GETLLAR Gets locked line. (PPU: lwarx, ldarx)
PUTLLC  Puts locked line. (PPU: stwcx, stdcx)
```

"My code can't be made parallel."

Yes. It can.

"C/C++ is no good for parallel programming"

The solution is to understand the issues -- not to hide them.

"But I'm just doing this one little thing... it doesn't make any difference."

Is it really just you?
Is it really just this one thing?

It's all about the SPUs

Designing code for concurrency

“What's the easiest way to split programs into SPU modules?”

Let someone else do it.

But if that someone is you...

Rules and guidelines

Rule #1

DATA IS MORE IMPORTANT THAN CODE

- Good code follows good data.
- Fast code follows good data.
- Small code follows good data.

Guess what follows *bad* data.

Rule #2

WHERE THERE IS ONE,
THERE IS MORE THAN ONE

The "domain-model design" Lie.

Rule #3

SOFTWARE IS NOT A PLATFORM

Unless you are a
CS Professor.

The real difficulty is in the unlearning.

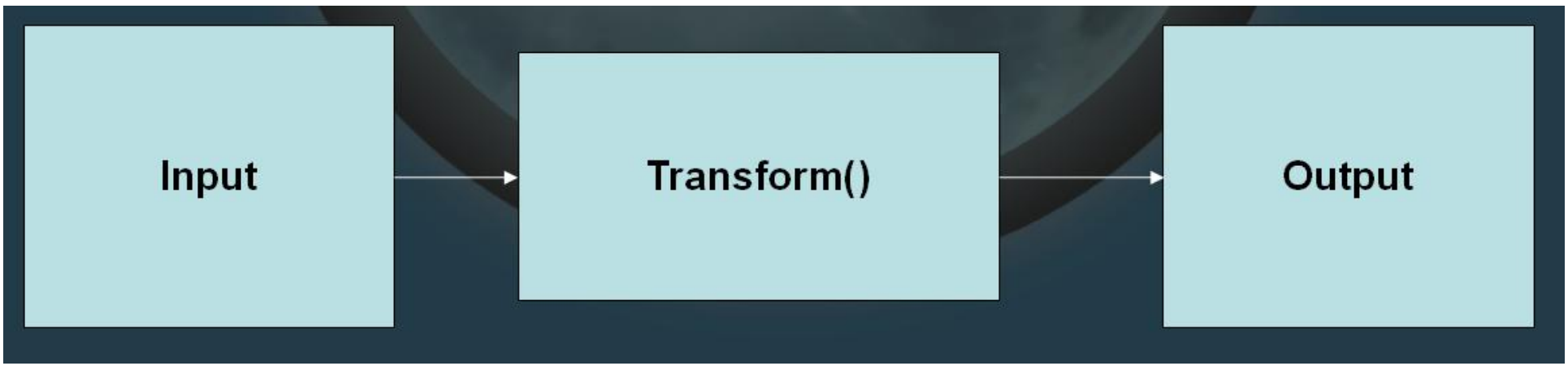
The ultimate goal:

Get everything on the SPUs.

Complex systems can go on the SPUs

- Not just streaming systems
- Used for any kind of task
- But you do need to consider some things...

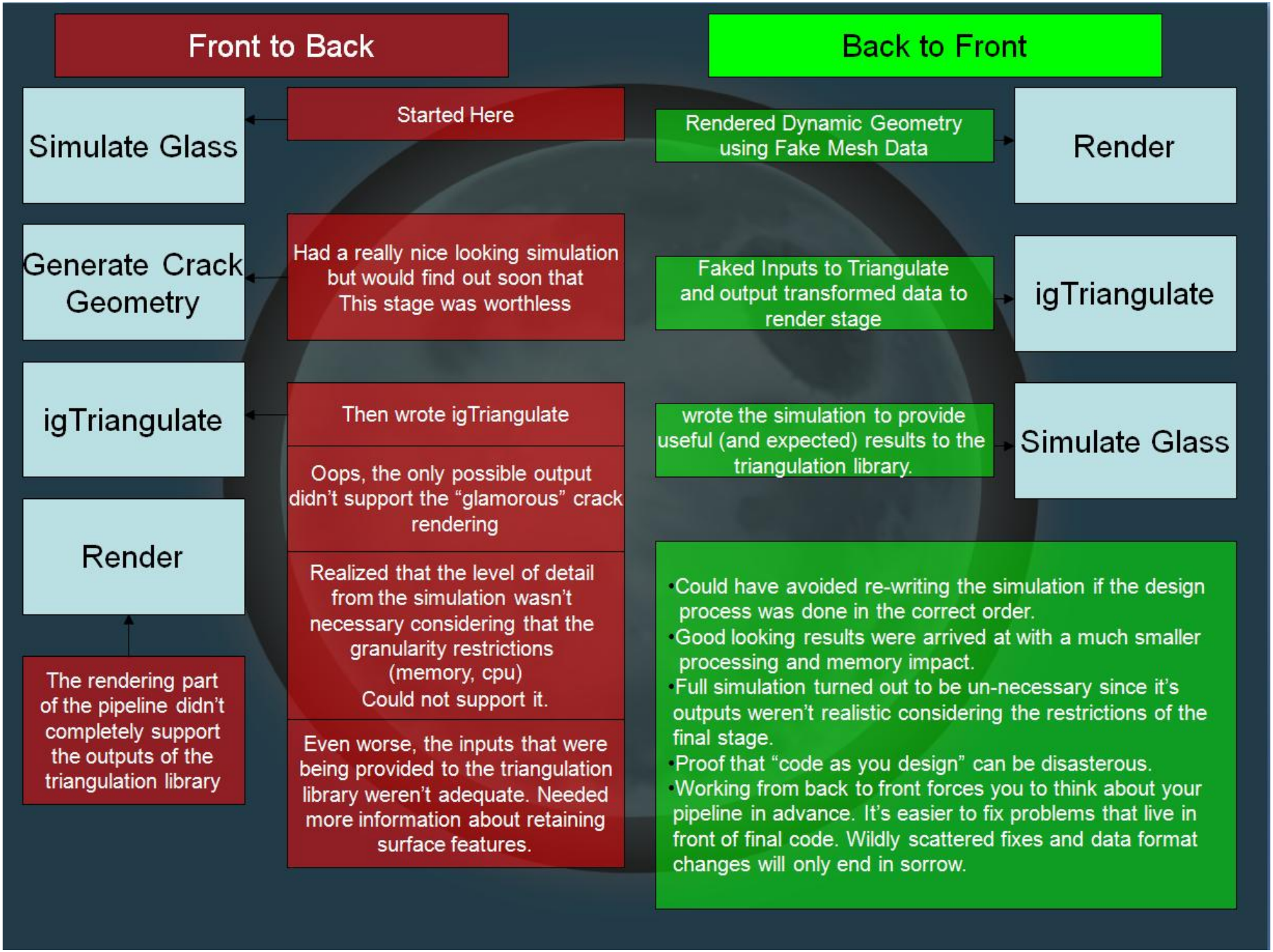
- Data comes first.
- Goal is minimum energy for transformation.
- What is energy usage?
  - CPU time.
  - Memory read/write time.
  - Stall time.



Design the transformation pipeline back to front.

- Start with your destination data and work backward.

Changes are inevitable -- Pay less for them.


Front to Back
Back to Front
Simulate Glass
Started Here
Rendered Dynamic Geometry using Fake Mesh Data
Render
Generate Crack Geometry
Had a really nice looking simulation but would find out soon that This stage was worthless
Faked Inputs to Triangulate and output transformed data to render stage
igTriangulate
igTriangulate
Then wrote igTriangulate
wrote the simulation to provide useful (and expected) results to the triangulation library.
Simulate Glass
Oops, the only possible output didn't support the "glamorous" crack rendering
Render
Realized that the level of detail from the simulation wasn't necessary considering that the granularity restrictions (memory, cpu) Could not support it.
The rendering part of the pipeline didn't completely support the outputs of the triangulation library
Even worse, the inputs that were being provided to the triangulation library weren't adequate. Needed more information about retaining surface features.
•Could have avoided re-writing the simulation if the design process was done in the correct order.
•Good looking results were arrived at with a much smaller processing and memory impact.
•Full simulation turned out to be un-necessary since it's outputs weren't realistic considering the restrictions of the final stage.
•Proof that "code as you design" can be disasterous.
•Working from back to front forces you to think about your pipeline in advance. It's easier to fix problems that live in front of final code. Wildly scattered fixes and data format changes will only end in sorrow.

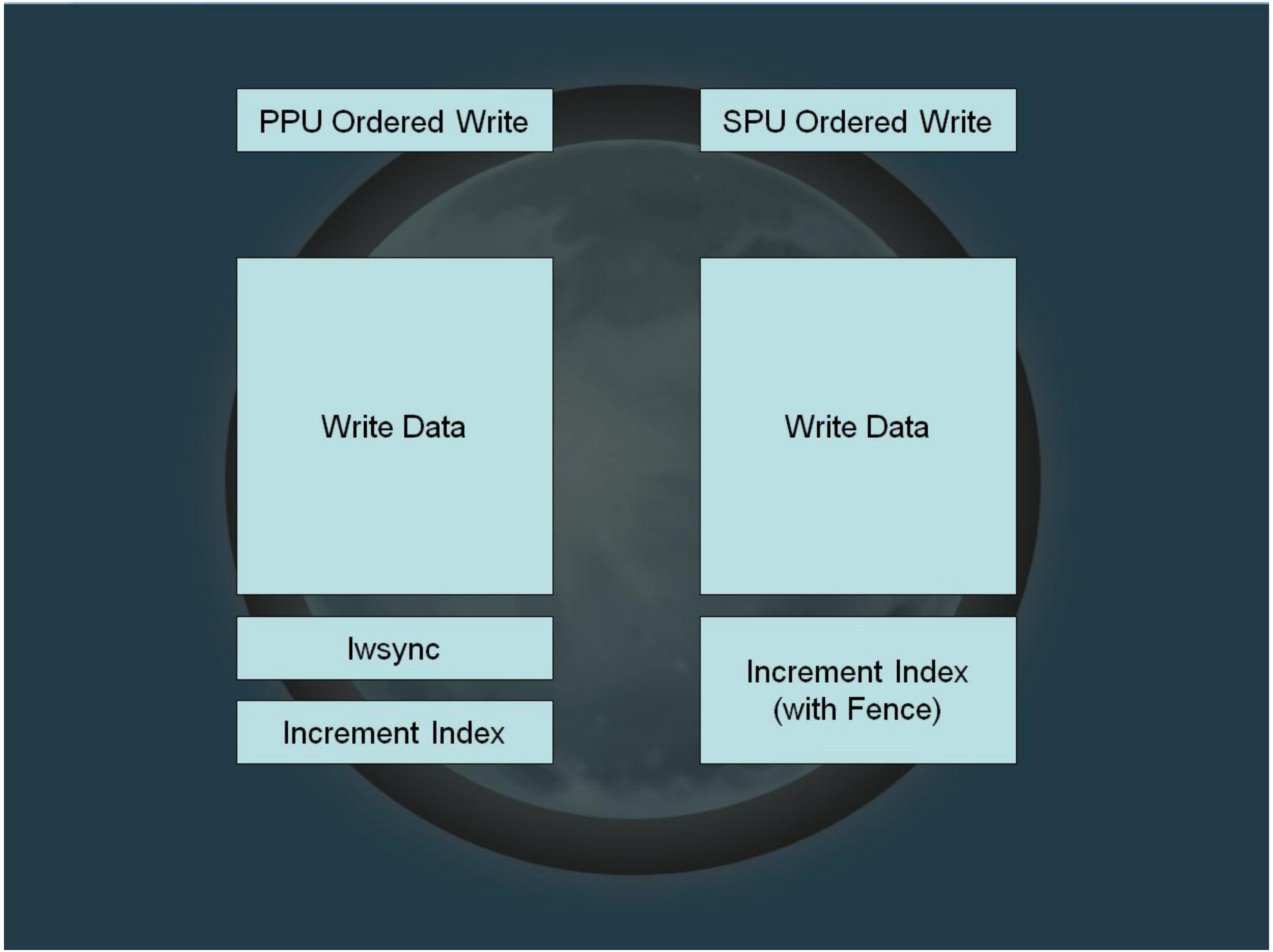
SPUs use the canonical data.

Best format for the most expensive case.

Minimize synchronization

Start with the smallest synchronization method possible.

Often is lock-free single reader, single writer queue.


PPU Ordered Write
SPU Ordered Write
Write Data
Write Data
lwsync
Increment Index
Increment Index
(with Fence)

Load balancing

Data centric design will give coarser divisions.

Start with simplest task queues

For constant time transforms:

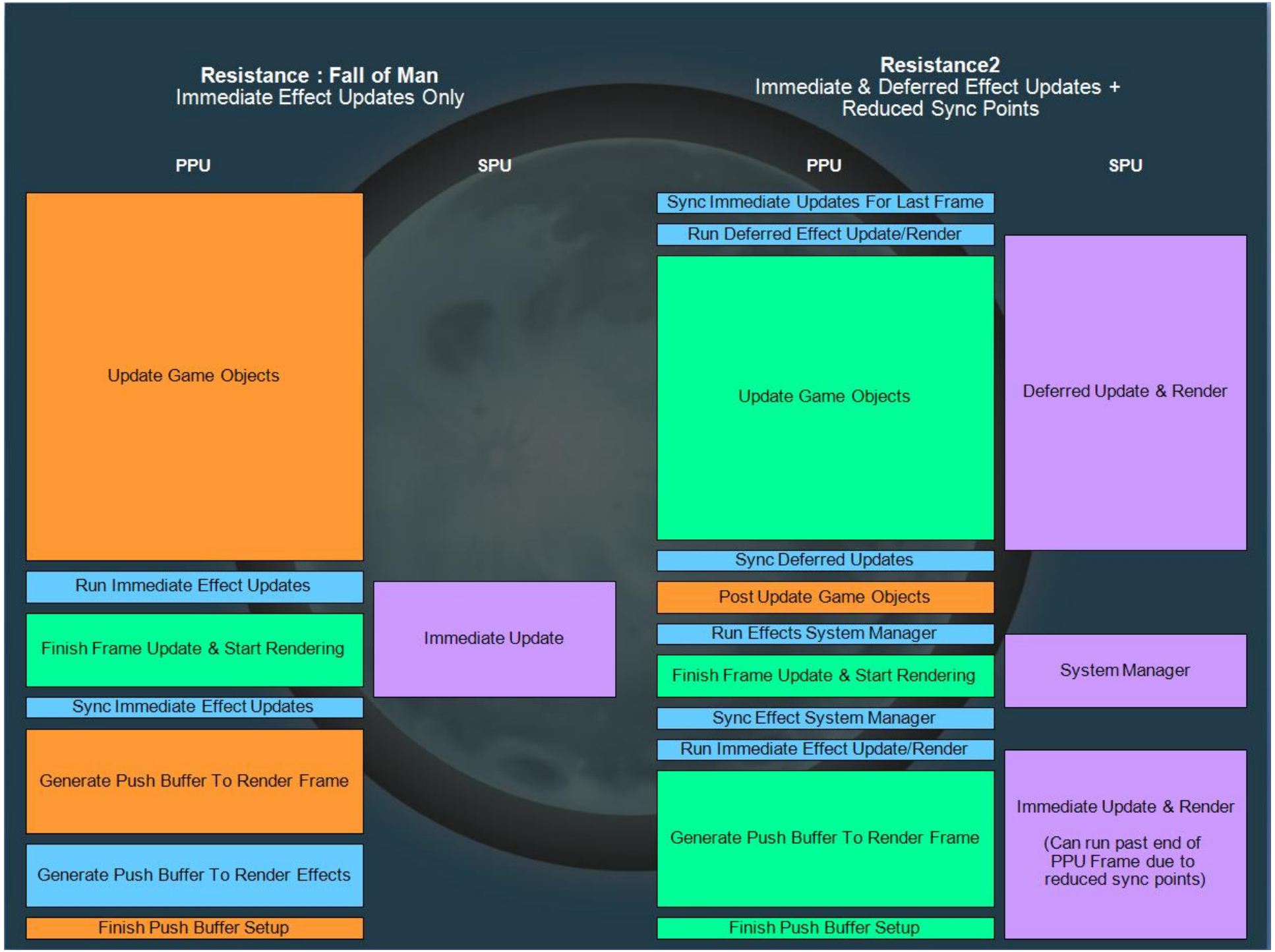
- Divide into multiple queues

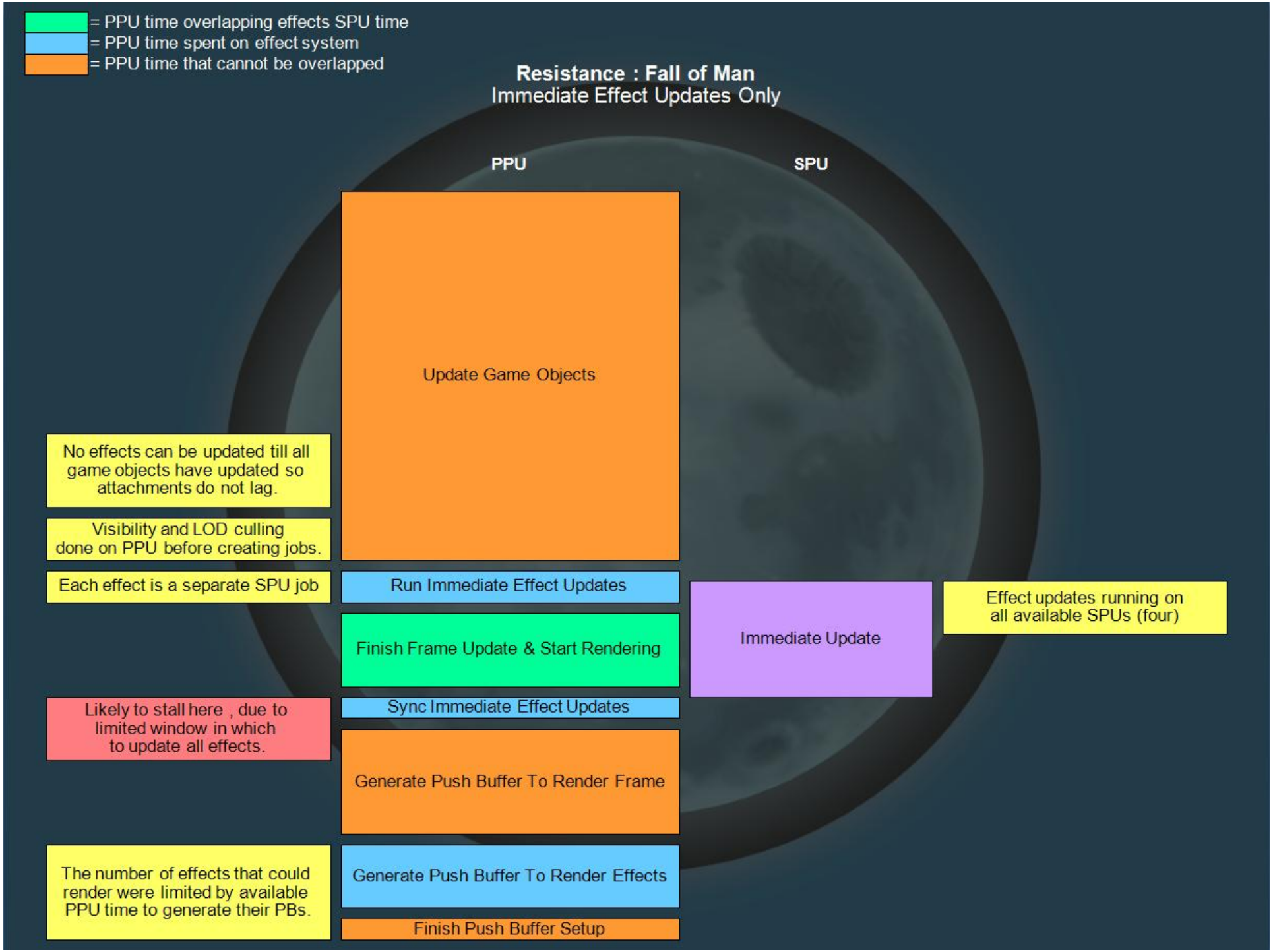
For other transforms:

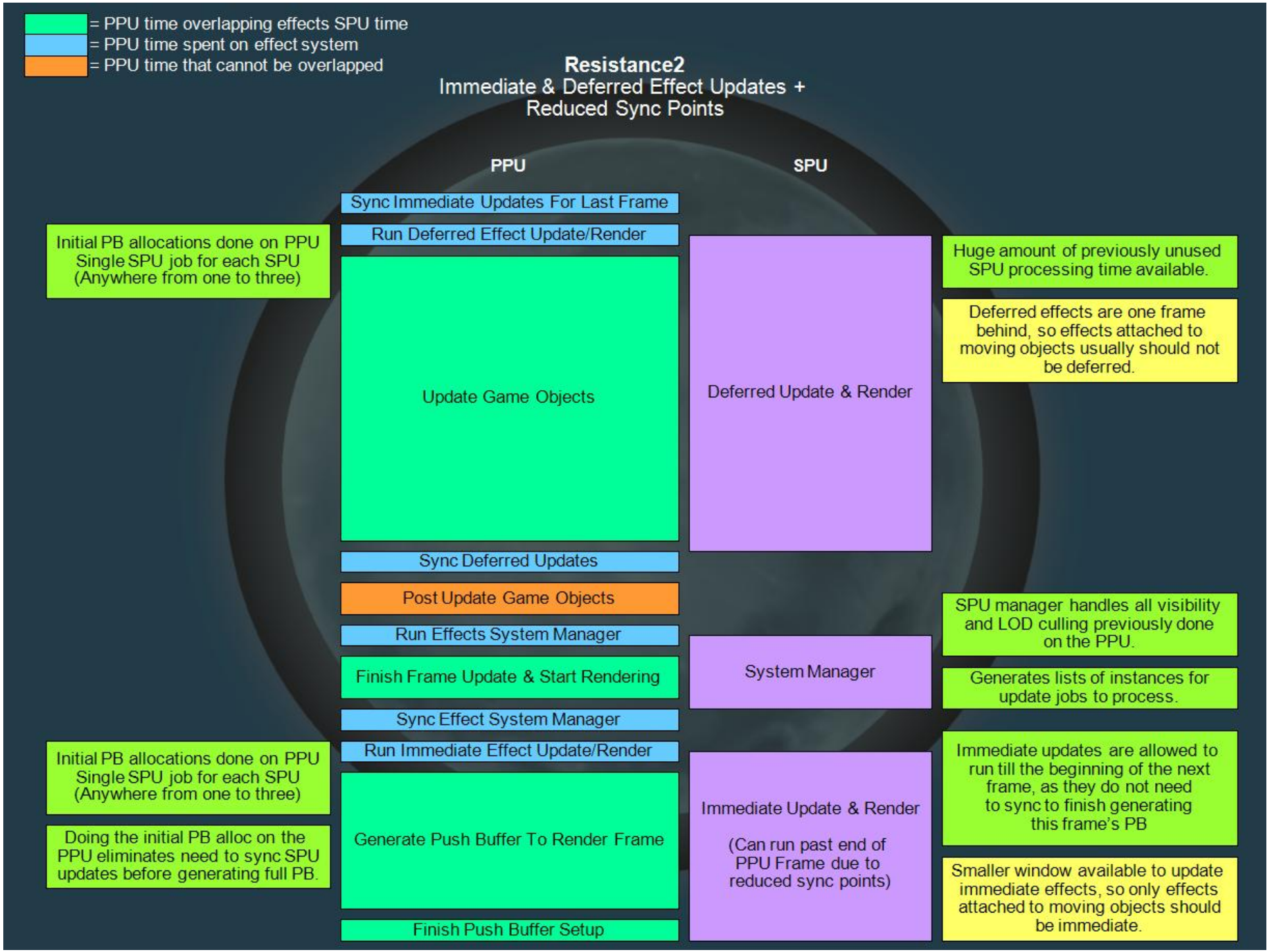
- Use heuristic to decide times and a single entry queue to distribute to multiple queues.

Then work your way up.

- Is there a pre-existing sync point that will work? (e.g. vsync)
- Can you split your data into need-to-sync and don't-care?


Resistance : Fall of Man
Immediate Effect Updates Only
PPU
SPU
Update Game Objects
Run Immediate Effect Updates
Finish Frame Update & Start Rendering
Sync Immediate Effect Updates
Generate Push Buffer To Render Frame
Generate Push Buffer To Render Effects
Finish Push Buffer Setup
Immediate Update
Resistance2
Immediate & Deferred Effect Updates +
Reduced Sync Points
PPU
SPU
Sync Immediate Updates For Last Frame
Run Deferred Effect Update/Render
Update Game Objects
Deferred Update & Render
Sync Deferred Updates
Post Update Game Objects
Run Effects System Manager
Finish Frame Update & Start Rendering
System Manager
Sync Effect System Manager
Run Immediate Effect Update/Render
Generate Push Buffer To Render Frame
Immediate Update & Render
(Can run past end of
PPU Frame due to
reduced sync points)
Finish Push Buffer Setup


= PPU time overlapping effects SPU time
= PPU time spent on effect system
= PPU time that cannot be overlapped
Resistance : Fall of Man
Immediate Effect Updates Only
PPU
SPU
Update Game Objects
No effects can be updated till all game objects have updated so attachments do not lag.
Visibility and LOD culling done on PPU before creating jobs.
Each effect is a separate SPU job
Run Immediate Effect Updates
Finish Frame Update & Start Rendering
Immediate Update
Effect updates running on all available SPUs (four)
Likely to stall here , due to limited window in which to update all effects.
Sync Immediate Effect Updates
Generate Push Buffer To Render Frame
The number of effects that could render were limited by available PPU time to generate their PBs.
Generate Push Buffer To Render Effects
Finish Push Buffer Setup


= PPU time overlapping effects SPU time
= PPU time spent on effect system
= PPU time that cannot be overlapped
Resistance2
Immediate & Deferred Effect Updates +
Reduced Sync Points
PPU
SPU
Sync Immediate Updates For Last Frame
Run Deferred Effect Update/Render
Initial PB allocations done on PPU
Single SPU job for each SPU
(Anywhere from one to three)
Update Game Objects
Deferred Update & Render
Huge amount of previously unused SPU processing time available.
Deferred effects are one frame behind, so effects attached to moving objects usually should not be deferred.
Sync Deferred Updates
Post Update Game Objects
Run Effects System Manager
Finish Frame Update & Start Rendering
System Manager
SPU manager handles all visibility and LOD culling previously done on the PPU.
Generates lists of instances for update jobs to process.
Sync Effect System Manager
Run Immediate Effect Update/Render
Initial PB allocations done on PPU
Single SPU job for each SPU
(Anywhere from one to three)
Doing the initial PB alloc on the PPU eliminates need to sync SPU updates before generating full PB.
Generate Push Buffer To Render Frame
Immediate Update & Render
(Can run past end of PPU Frame due to reduced sync points)
Immediate updates are allowed to run till the beginning of the next frame, as they do not need to sync to finish generating this frame's PB
Smaller window available to update immediate effects, so only effects attached to moving objects should be immediate.
Finish Push Buffer Setup

Write “optimizable” code.

Simple, self-contained loops
Over as many iterations as possible
No branches

Transitioning from "legacy" systems...

An example from RCF

FastPathFollowers C++ class

- And it's derived classes
- Running on the PPU
- Typical Update() method
- Derived from a root class of all “updatable” types

Where did this go wrong?

What rules where broken?

- Used domain-model design
- Code “design” over data design
- No advatage of scale
- No synchronization design
- No cache consideration

Result

- Typical performance issues
- Cache misses
- Unnecessary transformations
- Didn't scale well
- Problems after a few hundred updating

Step 1: Group the data together

“Where there's one, there's more than one.”

- Before the update() loop was called,
- Intercepted all FastPathFollowers and derived classes
- Removed them from the update list.
- Then kept in a separate array.

Step 1: Group the data together

- Created new function, UpdateFastPathFollowers()
- Used the new list of same type of data
- Generic Update() no longer used
- (Ignored derived class behaviors here.)

Step 2: Organize Inputs and Outputs

- Define what's read, what's write.
- Inputs: Position, Time, State, Results of queries, Paths
- Outputs: Position, State, Queries, Animation
- Read inputs. Transform to Outputs.
- Nothing more complex than that.

Step 3: Reduce Synchronization Points

- Collected all outputs together
- Collected any external function calls together into a command buffer
- Separate Query and Query-Result
- Effectively a Queue between systems
- Reduced from many sync points per “object” to one sync point for the system

Before Pattern

Loop Objects

- Read Input 0
- Update 0
- Write Output
- Read Input 1
- Update 1
- Call External Function
- Block (Sync)

After Pattern (Simplified)

Loop Objects

- Read Input 0, 1
- Update 0, 1
- Write Output, Function to Queue

Block (Sync)

Empty (Execute) Queue

Next: Added derived-class functionality

- Similarly simplified derived-class Update() functions into functions with clear inputs and outputs.
- Added functions to deferred queue as any other function.
- Advantage: Can limit derived functionality based on count, LOD, etc.

Step 4: Move to PPU thread

- Now system update has no external dependencies
- Now system update has no conflicting data areas (with other systems)
- Now system update does not call non-re-entrant functions
- Simply put in another thread

Step 4: Move to PPU thread

- Add literal sync between system update and queue execution
- Sync can be removed because only single reader and single writer to data
- Queue can be emptied while being filled without collision

See also: R&D page at insomniacgames.com on multi-threaded optimization

Step 5: Move to SPU

- Now completely independent thread
- Can be run anytime
- Move to new SPU system
- Using SPU Shaders

SPU Shaders

- On SPU, Code is data
- Double buffer / stream same as data
- Very easy to do (No need for special libraries)
  - Compile the code
  - Dump the object as binary
  - Load binary as data
  - Jump to binary location (e.g. Normal function pointer)
  - Pass everything as parameters, the ABI won't change.

The 256K Barrier

The solution is simple:

- Upload more code when you need it.
- Upload more data when you need it.
- Data is managed by traditional means
- i.e. Double, triple fixed-buffers, etc.
- Code is just data.

The End

- Programming for the SPUs is not really differentThese issues are not going to go away.
- Teams need practice and experience.
- Modern systems still benefit from heavy optimization.
- Design around asynchronous processing.
- Don't be afraid to learn and change.