Innovating in a Software Graphics Pipeline

Paul Lalonde

Intel Corporation

User innovation matters

Traditional API
-Mostly fixed function
-Some programmable points

Extended Pipeline
-More fixed function
-More programmable parts
-But still under vendor control

Extensible Pipeline
-Usable as-is
-Data structure & process conventions
-User can open it up and extend it

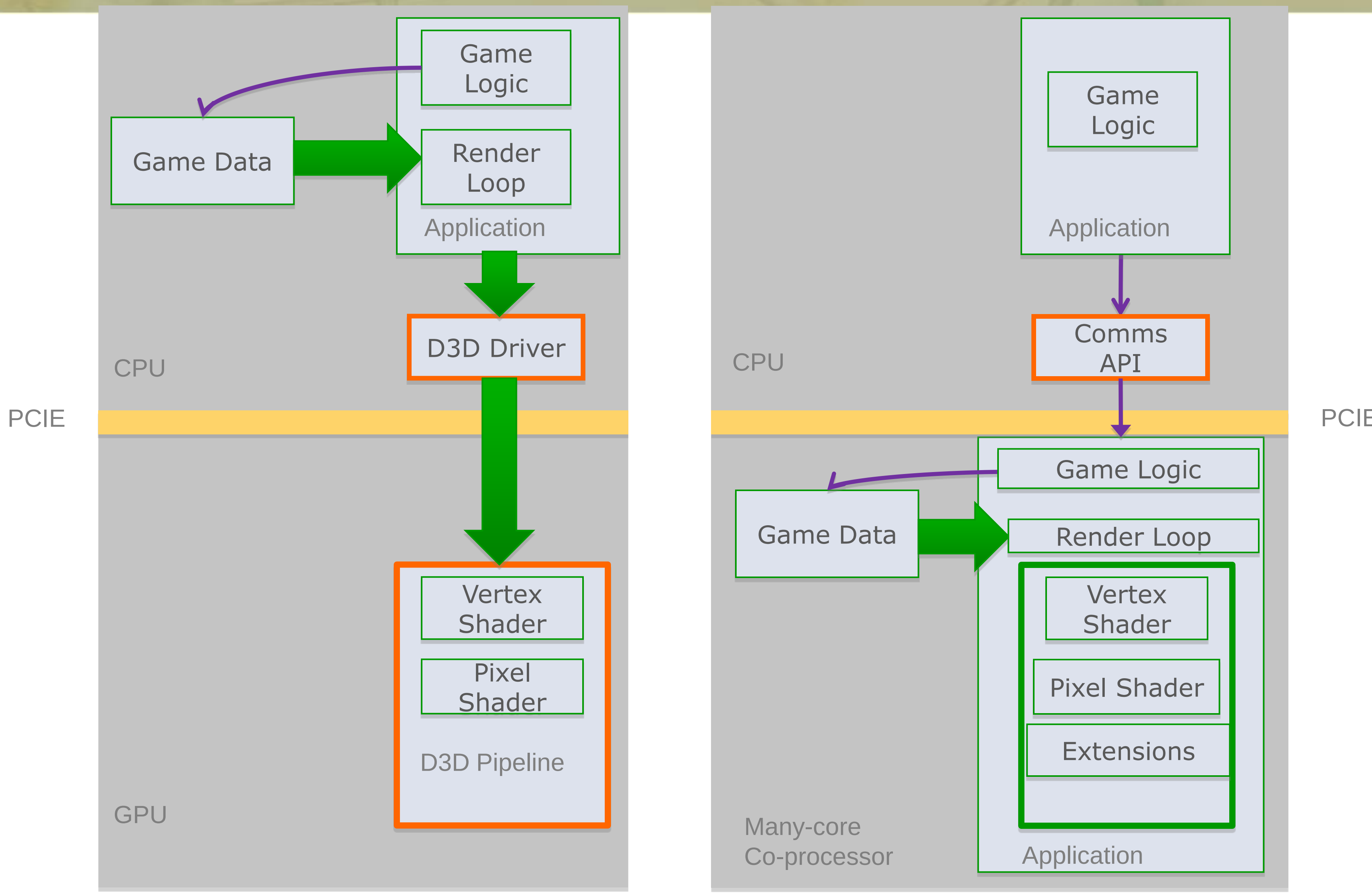
Bare metal
-Totally flexible
-Only feasible to a few elite devs

Graphics APIs

- State-based, programmable at particular stages
- Restricted data model
- Innovation bottlenecks
  - Years of delay from proposal of feature to acceptance in the standard
  - Extensions either not supported (DX), or supported in ad hoc, vendor-incompatible ways (OpenGL)
- To be fair, also the best existing parallel programming model because of many of these restrictions

Traditional Rendering Model

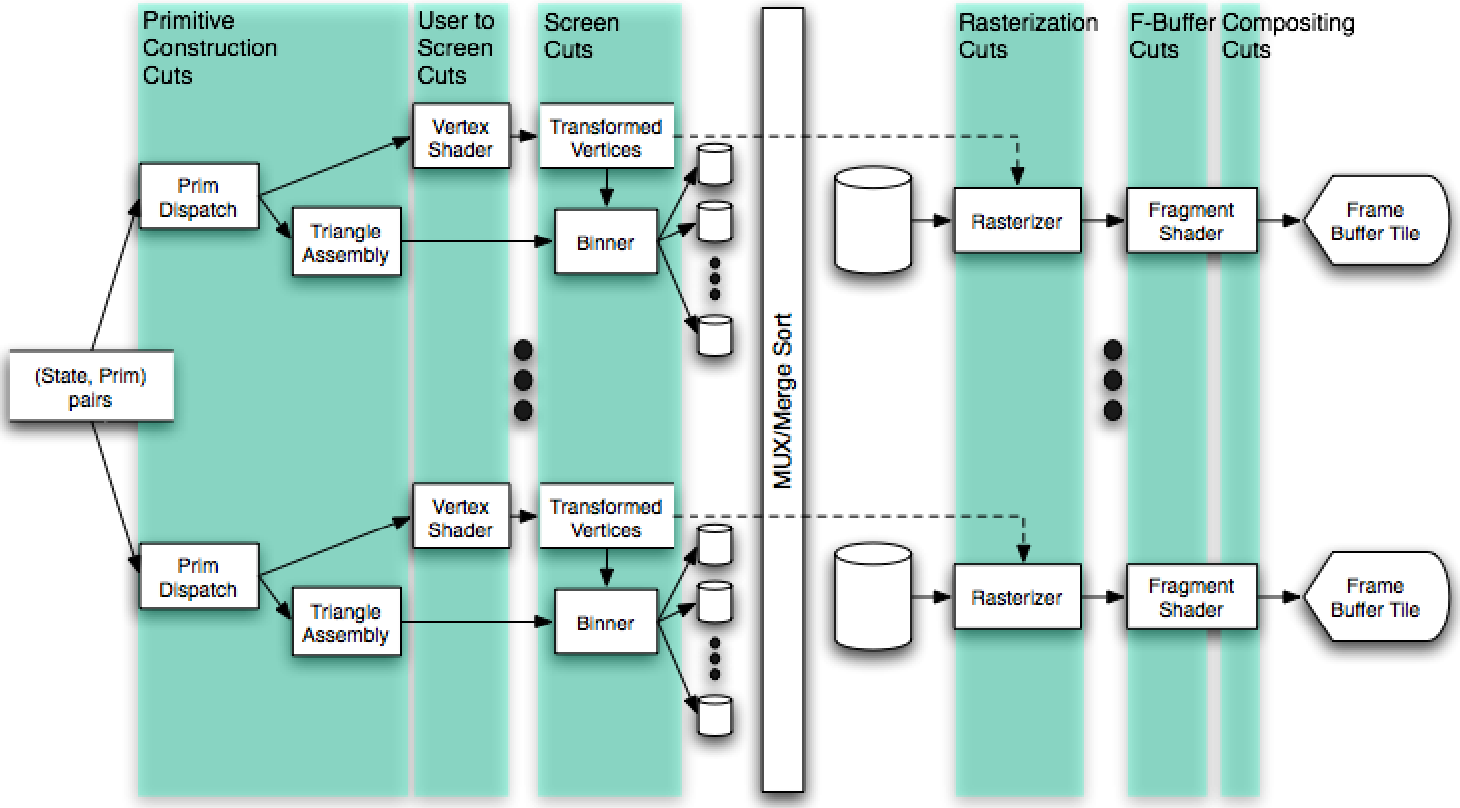
Software Rendering Model



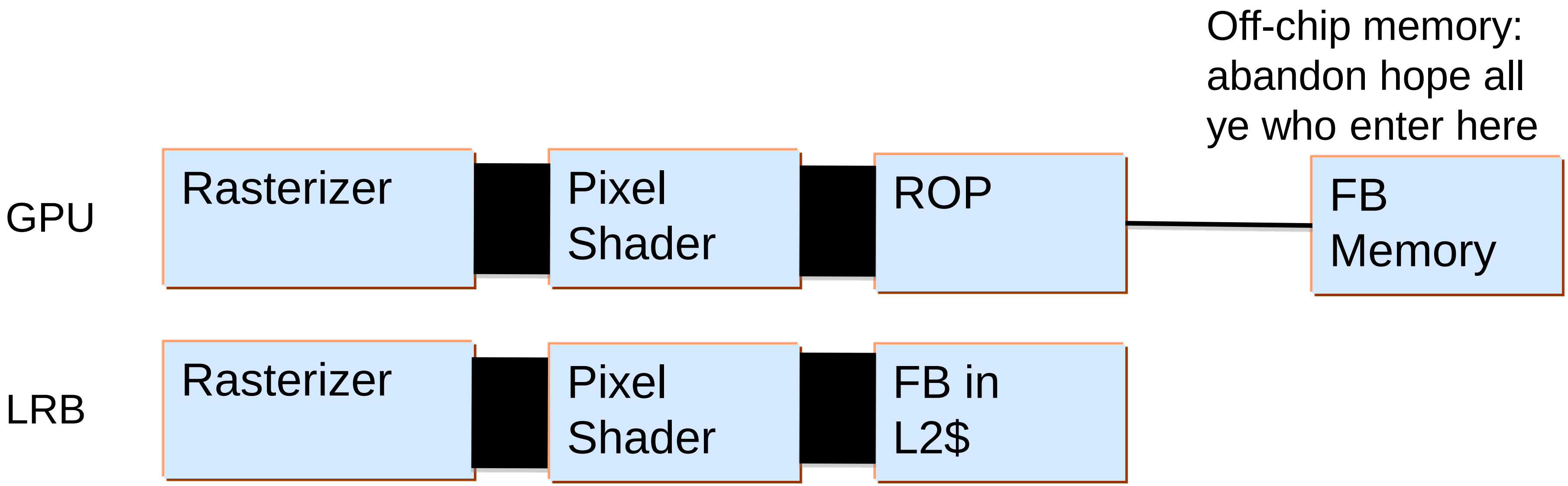
Many-Core Graphics

- C/C++/Parallel Languages and Libraries
  - Throughput constraints
    - VPU utilization is important, “shader compilers” are important, even in user space (See Tim Foley’s talk)
- WDDM complements our memory model
  - Heap, resources can be declared on host
  - WDDM ensures declared resources are resident when LRB Native code is called
- Pushbuffer model hides PCIe latency
- Frame buffer presented without tunneling back to host

A Rendering Organization



Framebuffer RMW



- Pixel shader can read from current render target
- Compute new value based on existing one
- Blending can all happen with simple math in pixel shader
- More generally, pixel shader can maintain and update a small data structure at each pixel

Read-modify-write

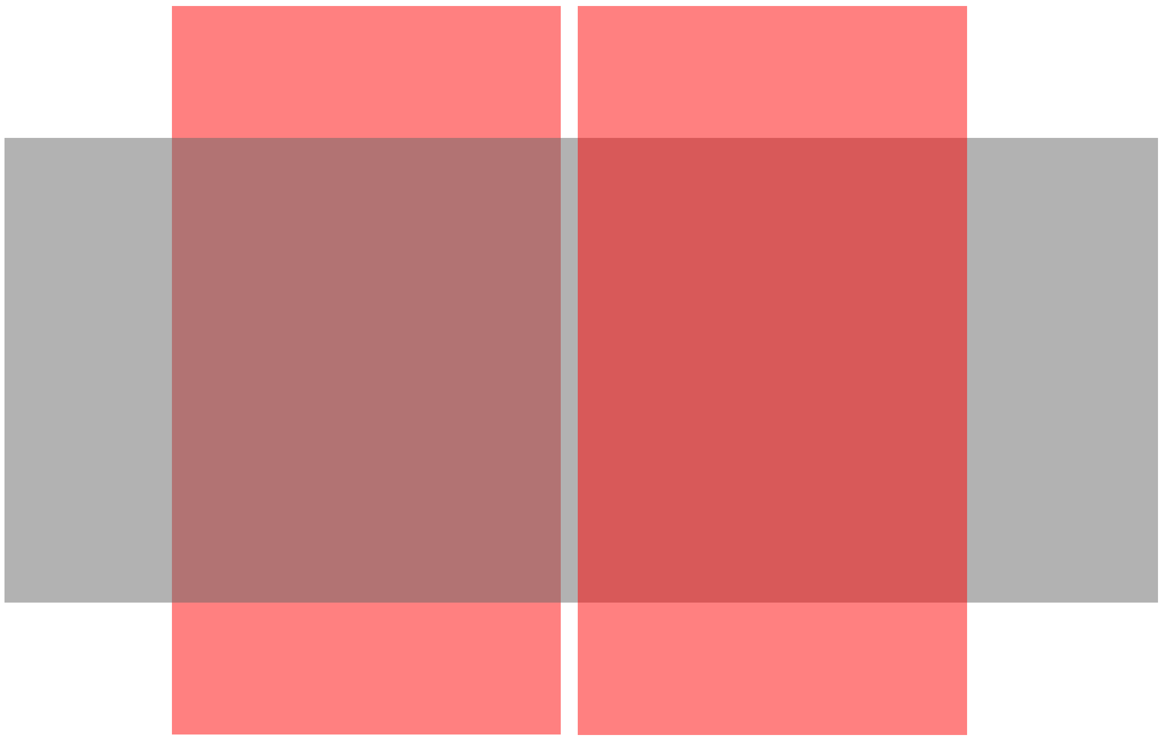
```
void myPixelShader(
  ...
  in out float fbDepth : SV_Depth,
  in out float4 fbValue : SV_Target)
{
  // depth test in user code
  if (fragPos.z > fbDepth)
    return;
  fbDepth = fragPos.z;

  // "blend" using matrix multiply
  fbValue = mul(m, fbValue);
}
```

In out parameter bound to framebuffer semantic

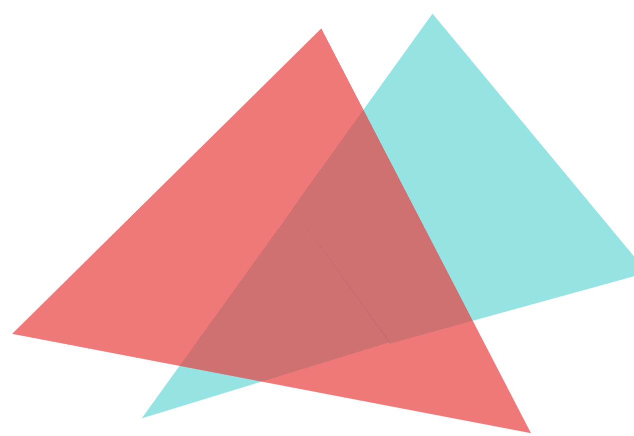
Shader can read previous values from color/depth buffers at current pixel

Translucency



You must sort by depth

So you have to sort per fragment



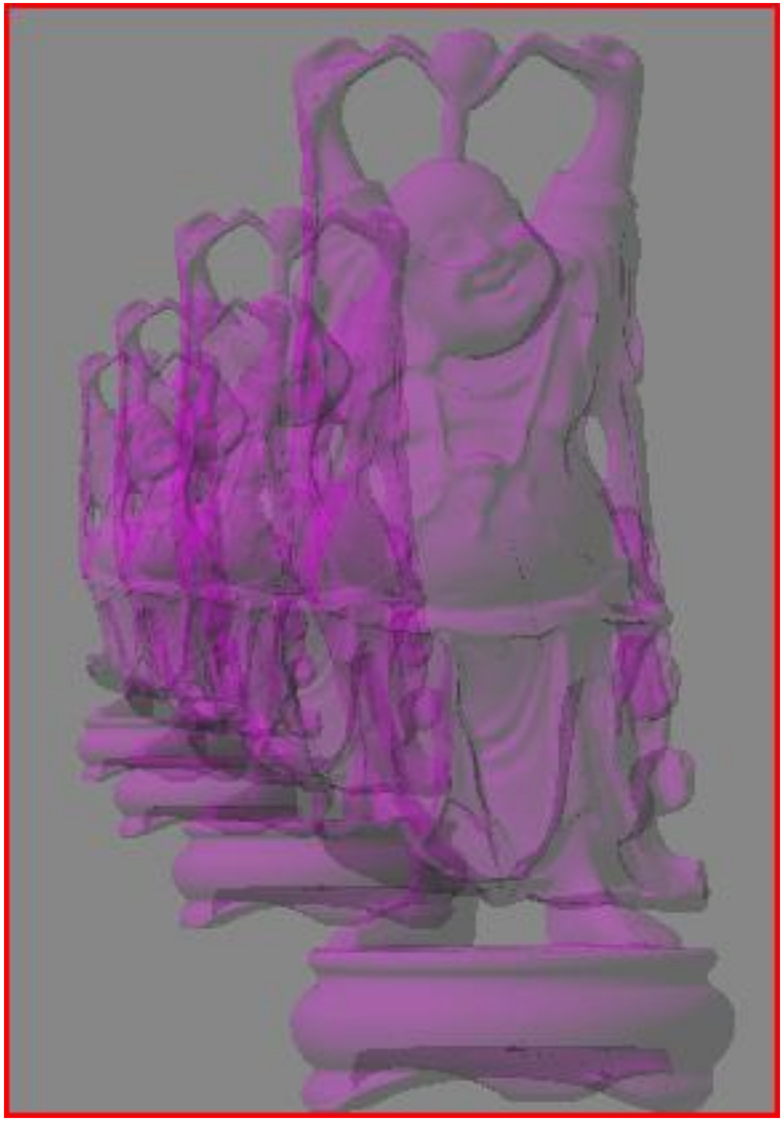
K-Buffers (Bavoil et al. 2007)

- Store k frame buffers containing layer data
- Layered depth images, layered shadows
- (Limited) order independent transparency
  - Sort k layers at once, but can’t sort any more than that

- Fixed allocation—*k* layers
- Performance vs. flexibility trade-off

- Falls out easily from frame buffer RMW
- Allocate n render targets
- Do sorting/ordering in shader

K-Buffer Transparency



Model courtesy of the Stanford Computer Graphics Laboratory

List Render Targets (LRT)

- Allow arbitrary number of layers to be stored per pixel
  - Can store *all* fragments in a pixel, not just closest
  - Do in "user space": let shader writer decide which ones to store, when to remove old ones, etc
- Render objects as usual
  - List of fragments is accumulated at each pixel
- Render quad running resolve shader
  - Sort if needed, composite layers, compute output values
  - Run out of L2$, avoid main memory bandwidth

List Textures

- Naïve implementation is linked list per pixel
- Smarter is linked list per SIMD width cluster of fragments, with masks
- Lists can be unsorted, sorted on insertion, post-sorted
- Principal optimization need is cache residence of working set of fragments & memory management within the List Texture
  - Thread safety of insertions & sorts
  - Smart cache management within a tile's working set
- Exposed as a List<> templated type in the shader

List Render Targets

```
struct FBSample {
  float4 color : SV_Target;
  float depth : SV_Depth;
};

void myPixelShader(
  ...
  in out List<FBSample> fb)
{
  fb.Insert( FBSample(col, depth) );
  float4 c = 0;
  for( FBSample s : fb )
    c = c * s.color.a + s.color;
}
```

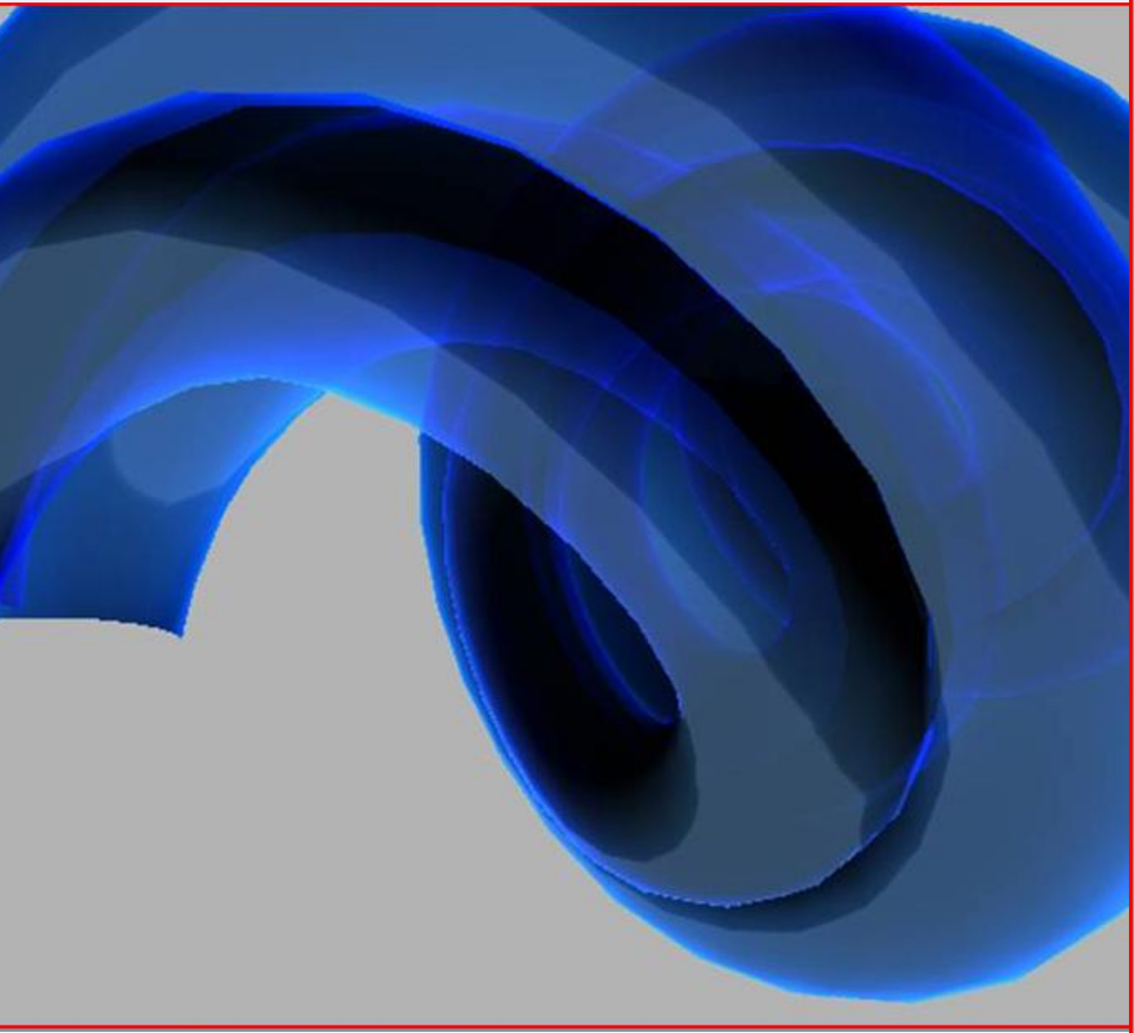
Declare per-layer data type

Declare shader input for read, write or both

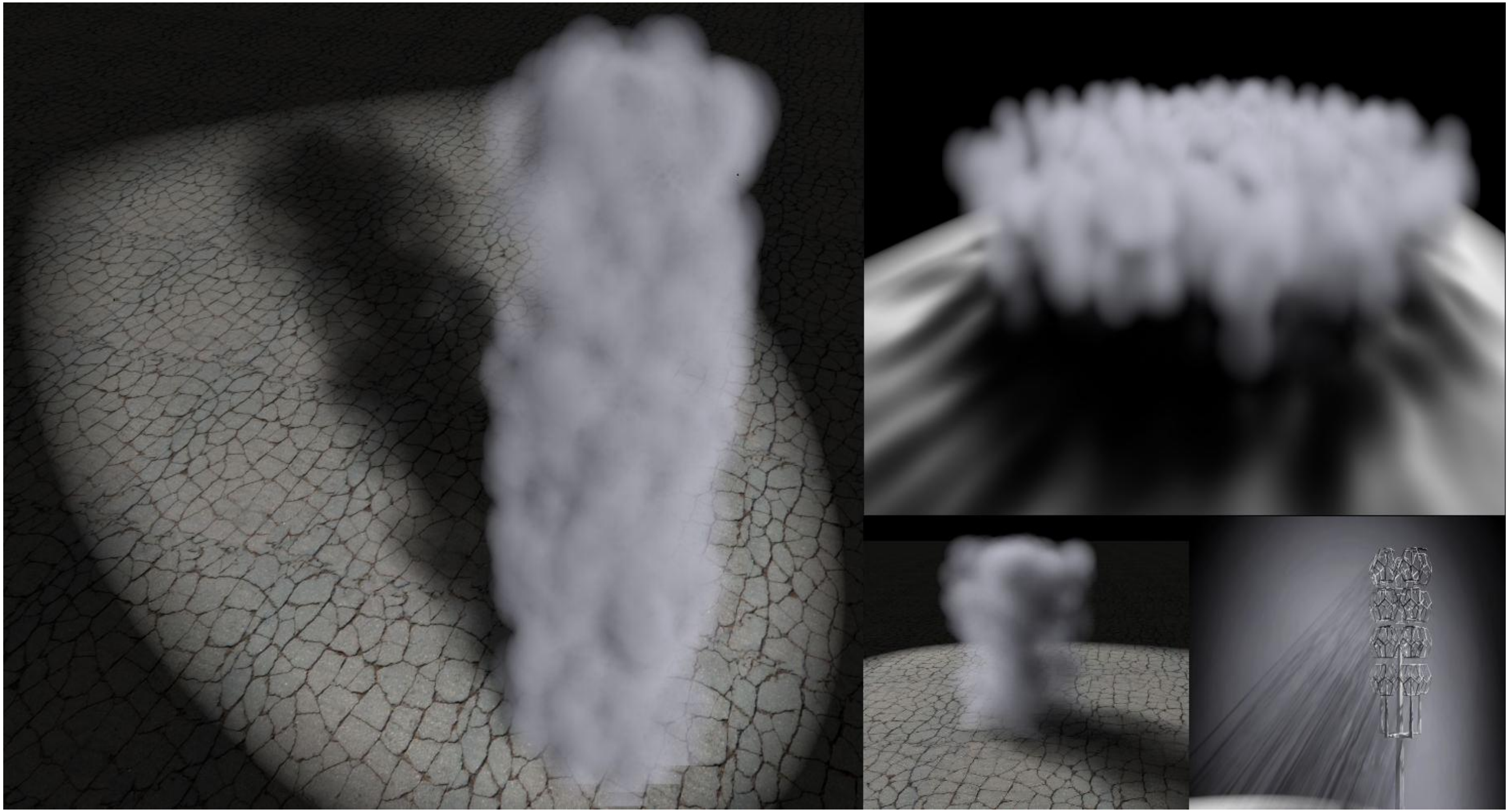
Insert new layers, or …

… iterate over layers at current pixel/sample to resolve

LRT Examples



Buddha model courtesy of the Stanford Computer Graphics Laboratory

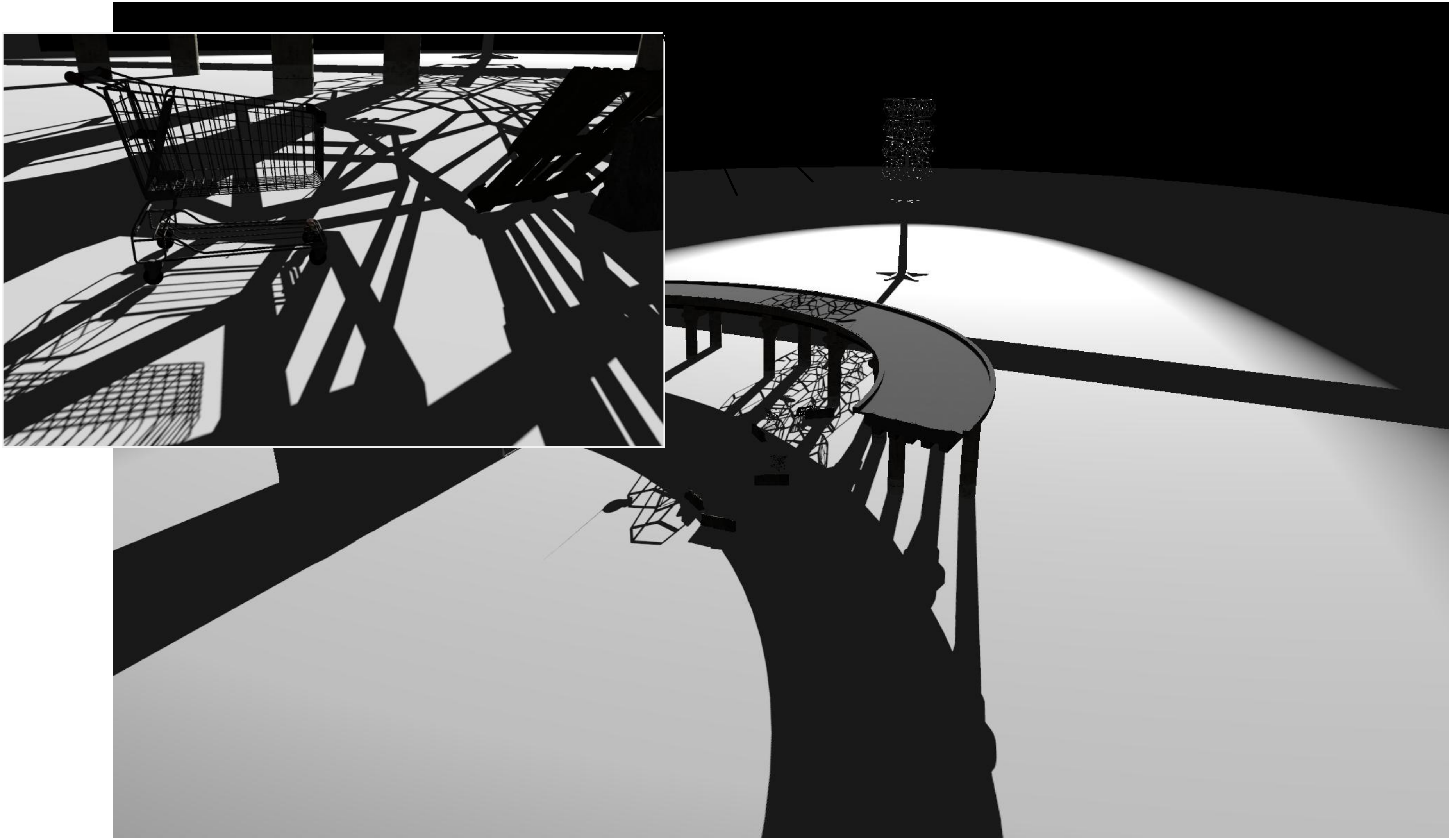


Programmable Resolve

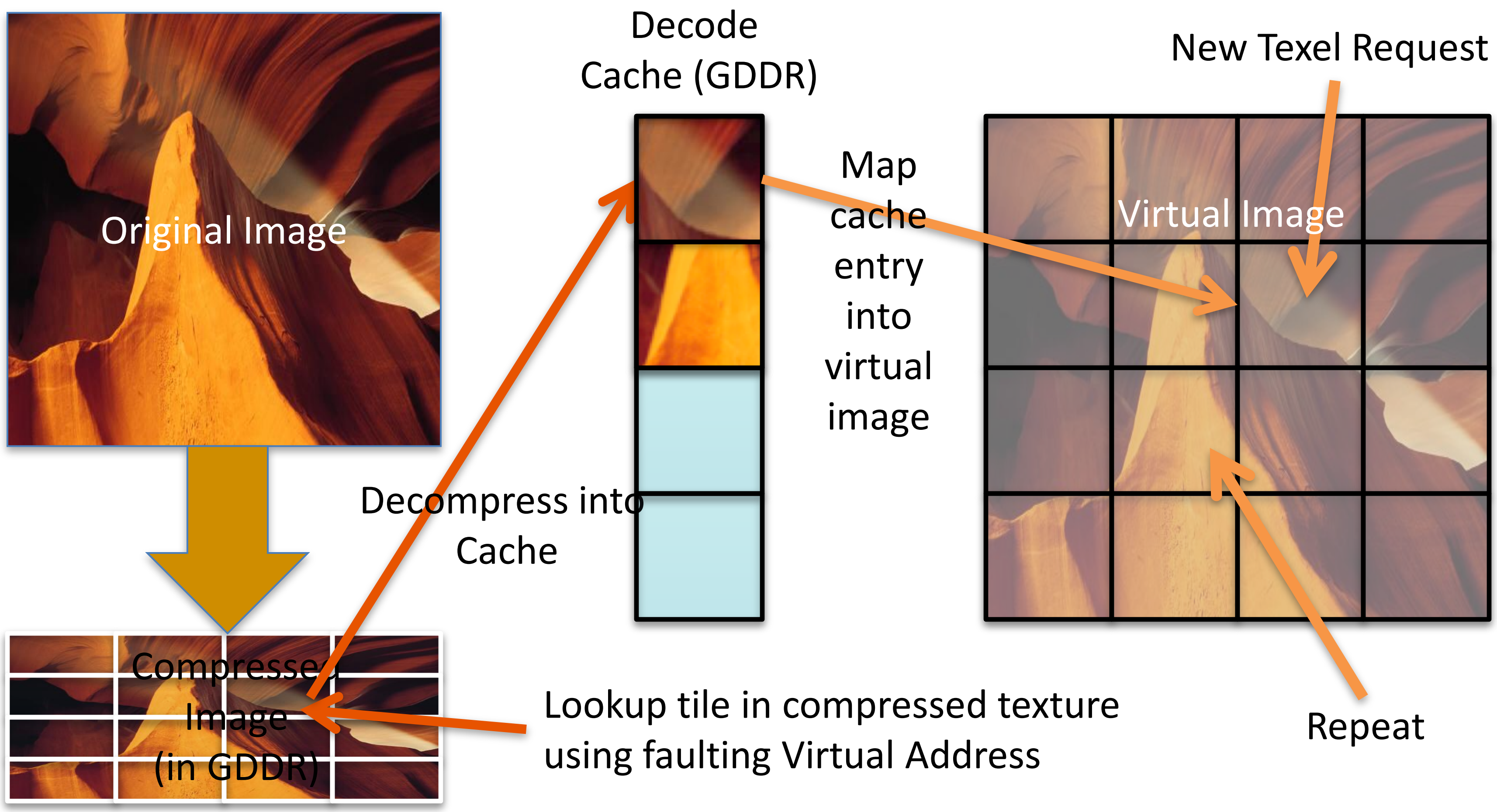
- After rasterizer backend finishes a tile, framebuffer is still in L2$
- Useful opportunity to do image processing before write to memory
- Typical GPU approach:
- Render to texture (commit to main memory)
- Draw full-screen quad, read from texture (read back from memory)
- Bandwidth intensive
- Limited ability to leverage caches for reuse

Programmable Resolve

- Call out to user-space code with pointer to framebuffer tile
  - Do all processing out of L2$
  - No off-chip bandwidth cost
  - Uses: image processing, building data structures, reductions, …
- Can do processing either with data-parallel code or with just a few threads
  - High performance available from either
  - Choose one where computation is more easily expressed



Demand-Paged Textures



Conclusion

- Each of these techniques is more expensive than straight rasterize/shade
- But each can save multiple geometry submission passes and GPU->CPU->GPU traffic, leading to higher system efficiency
- None of these extensions requires particular GPU hardware features from the vendor.
  - All can be done in user space
  - The value of the software pipeline is that it is extensible, not just extended.

Larrabee allows the ultimate in extensible pipelines

Thank you

Backup

Shader Support for DPT

- Generated code looks like:

```
while(!done) {
   err = sample(texture, u, v)
   if(err == PNP) {
     UserCB(cpuPage, txsPage, tileNo, userContext)
   } else {
     done = true
   }
}
```

Managing the DPT Cache

- Many update strategies are possible
  - Just in time, Mip-Level surrogates, just-too-late
- Use any available physical pages to hold computed/decoded textures
- Allow the paging daemon to discard these instead of swapping them.
- The OS knows pretty well what’s in use and what’s stale, so why duplicate that in the DPT implementation?
- Multiple caches can be managed independently for different uses: worlds vs characters, for example

Execution Model

- The renderer buffers up an entire frame's worth of draw commands
- This makes it possible for it to do **all** front-end graphics processing before starting back-end
- *Thus, changing state and/or switching render targets is cheap*
- *Mostly pointer updates in the same memory space*
- Also allows a nice alternative for submission
- Typical GPU:
  – For each render target, draw each visible object
- Software Rendering:
  – For each object, draw to all relevant render targets