# Total Recall: A Debugging Framework for GPUs

Ahmad Sharif and Hsien-Hsin Sean Lee, Georgia Institute of Technology

# Outline

•Motivation

•Related Work

•Goals

•Key Concepts

•Basic implementation

•Acceleration

•Challenges/Future Work

•Conclusion

# Motivation for a GPU Debugger

- GPUs are massively parallel machines w/ billion transistor budgets
  - Hard for CPU programmers to debug shader code
- Lack of native debugging support (breakpoints, watchpoints, etc.)
- Debugging is a time sink

*"GPU programmers have just a small handful of languages to choose from, and few if any full-featured debuggers and profilers." (Owens et al., A Survey of General-Purpose Computation on Graphics Hardware, COMPUTER GRAPHICS forum, 2007)*

# Related Work

- PIX by MS (for D3D) has a pixel history feature
    - Does not allow debugging across render targets, though
- GLSL Devil by Strengert et al allows debugging of OpenGL shaders
- gDebugger by GraphicsRemedy
    - No single stepping as of May 2007
- REF_RAST & Visual Studio by MS
    - Too slow for big/complex shaders
- Shadesmith by Purcell et al
- Relational Debugging Engine by Duca et al

# Total Recall Goals

•Application transparent debugger

•Given a frame consisting of series of: [SetX]* [DrawX]* Present, and breakpoint conditions, obtain *entire history* of the pixel that hits the breakpoint.

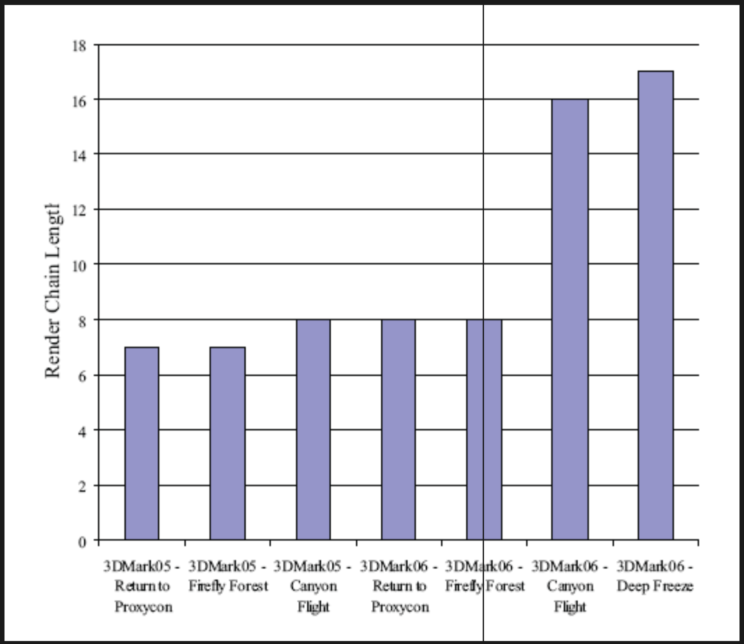
•Deterministically replay all conditions that led to breakpoint condition.

•Done on the CPU

•Stepping/Watchpoints/etc. become easy to do this way

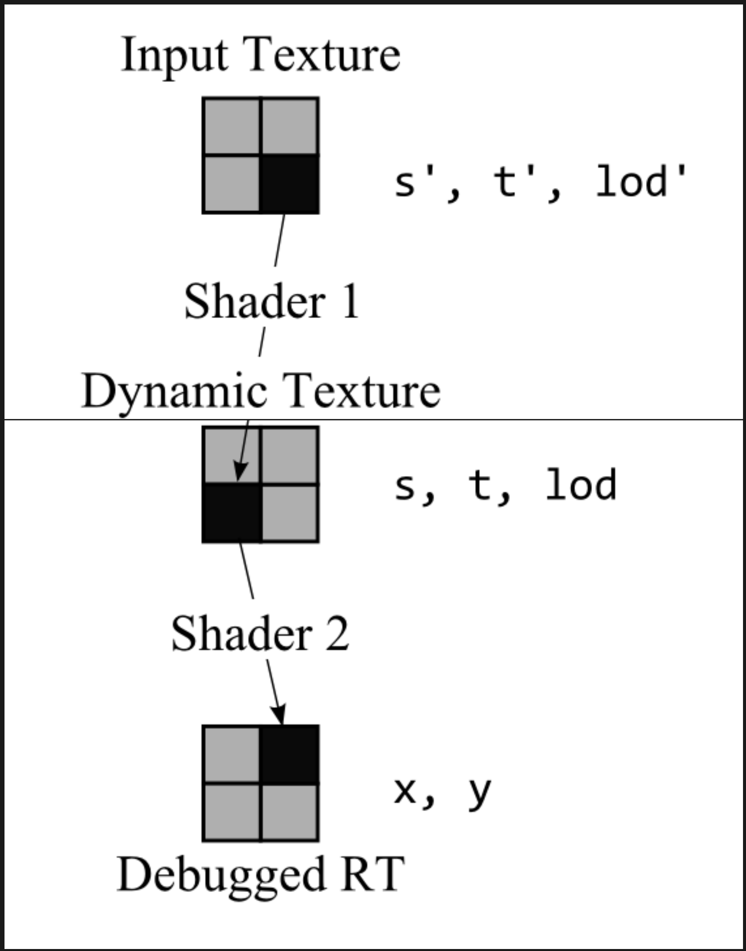
# Total Recall Goals II

- Debug multipass in a unified fashion
  - Ex: Env/Shadow Maps, Deferred shading, etc.
- Current debuggers only debug single render pass
  - Need a way to debug multiple render passes



# Multipass Debugging of pixel shaders

```
// Linearized execution stream

float4 val1;

float4 val2;

// Look up static texture

val1 = lookup(input_tex, s’, t’, lod’);

// Run it through the shader

dyn_tex[s,t] = shader1(val1);

// Look up dynamic texture now

val2 = lookup(dyn_tex, s, t, lod);

// Run it through second shader

output[x,y] = shader2(val2)

// This is the output that hit the
   breakpoint
```



# Key Features of the Debugger

- Breakpoints
- Support 2 kinds
  - Pixel coordinate breakpoints
  - Conditional breakpoints
- Once a breakpoint is hit, need to figure out all input data for deterministic replay
- Obtain only necessary data without too much overhead
- Need to go deeper than just a couple of draw calls
  - Need entire frame in memory!
- Need emulation module

# Breakpoint Conditions

- 2 kinds of breakpoints
  - Break at certain condition
  - Break at certain pixel location
- Conditional breakpoints:
  - Bind debug render target; write on condition; occlusion query to check if hit
- Pixel breakpoints
  - Clear 4 sub-rectangles of z-buffer to lowest value
  - Occlusion Query to check if hit

# Pixel Shader Inputs

- Bind debug RT & pass-through pixel shader
  - RT has to be big, otherwise require multiple passes
  - Scatter support?
- s, t values obtained from inputs; dx, dy to compute mip-levels for filtering

# Main Loop

- Intercept and record all program state
- Breakpoint hit?
- Obtain shader inputs
  - Include texture coordinates
- Program breakpoint at coordinates, replay scene stored in memory

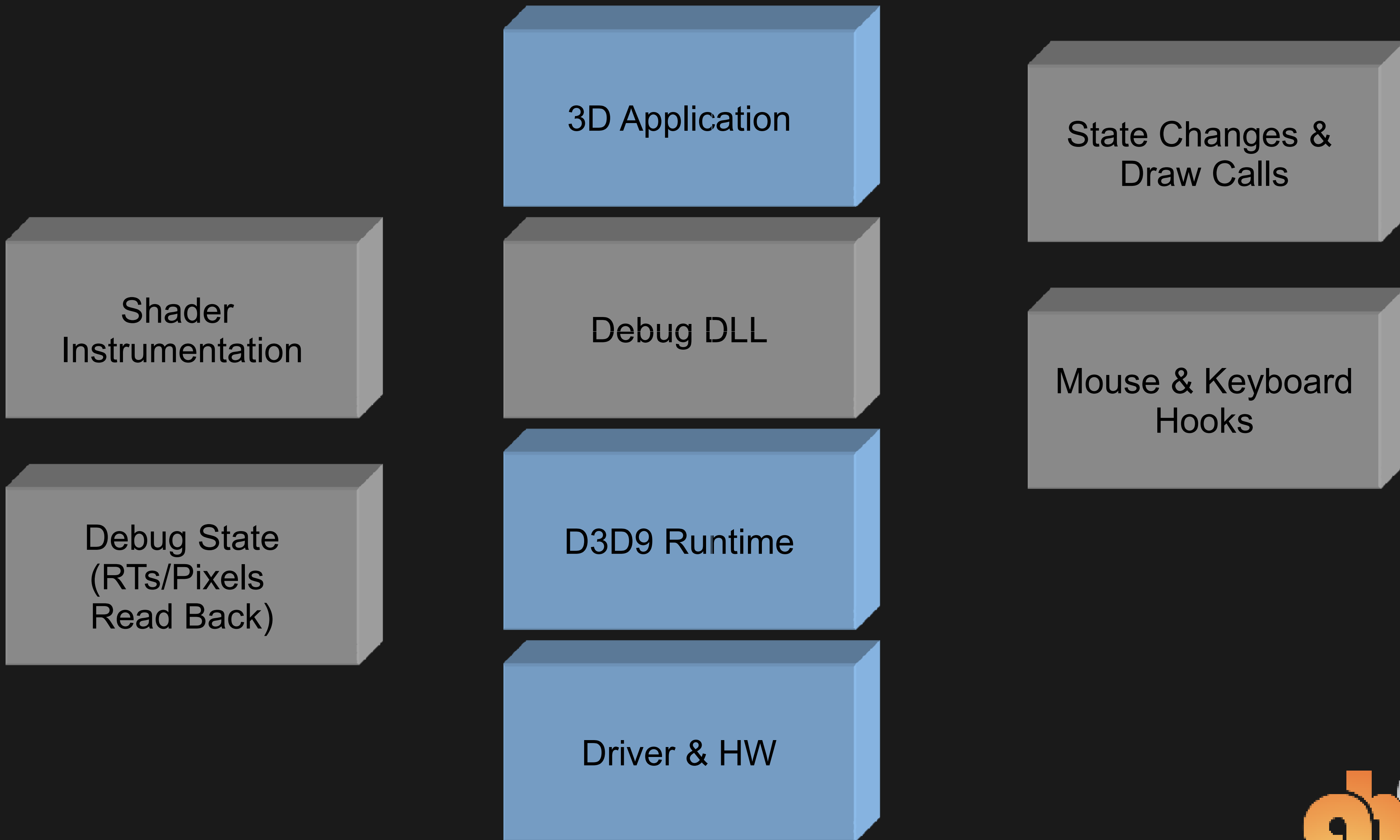
# SW Architecture of Implementation

- Used Direct3D 9
- DLL that encapsulates D3D exported interfaces
  - Saves per frame state changes
  - Pixel breakpoints implemented
  - Performs several passes to obtain complete history
  - Uses occlusion queries and temporary render targets
- Shader emulation can be done via a vendor-provided library

# Intercepting DLL

- DLL exports CreateDevice()
- Wraps IDirect3DDevice9, IDirect3DVertexBuffer, IDirect3DIndexBuffer, etc.
- From the IDirect3DDevice9 interface, rest are hooked
- Every SetX() and DrawX() calls are recorded in replay buffers
    - Memory requirements vary: several MBs per frame to hundreds of MBs per frame
- Mouse hooked to indicate pixel of interest (Win32 Hooks)

# Diagram of Implementation



# Challenges

- Proprietary floating point formats
  - Functional emulation library can solve it
- Texture super-sampling/multi-sampling
- Alpha Blending (multiple primitives causing write at the same pixel)

# Acceleration

- Low resolution debug render targets
- Main loop is fill-intensive
- Sub-divide screen into parts, and replay only relevant parts
- Track dependencies using bitvector
  - Propagate on shader texture read
  - Expose to debugger so it can be made use of
- Once dependencies are replayed, emulate like usual

# Future Work

- Extension to GS/VS
- Extension to GPGPU
  - Entire history of single particle in PS
- History of race conditions (two writes to single memory location)

# Conclusions

- A framework for debugging is presented with a sample implementation
  - Allows debugging of breakpoints via selective emulation
  - Makes GPU debugging look like CPU debugging
  - Hardware support for acceleration is proposed
- Limitations
  - Relies on runtime/driver/hardware to behave correctly
  - Deviations from actual results possible in emulation unless vendor provides emulation library

# Questions

- Please email ahmad@gatech.edu