Programmable Graphics Hardware

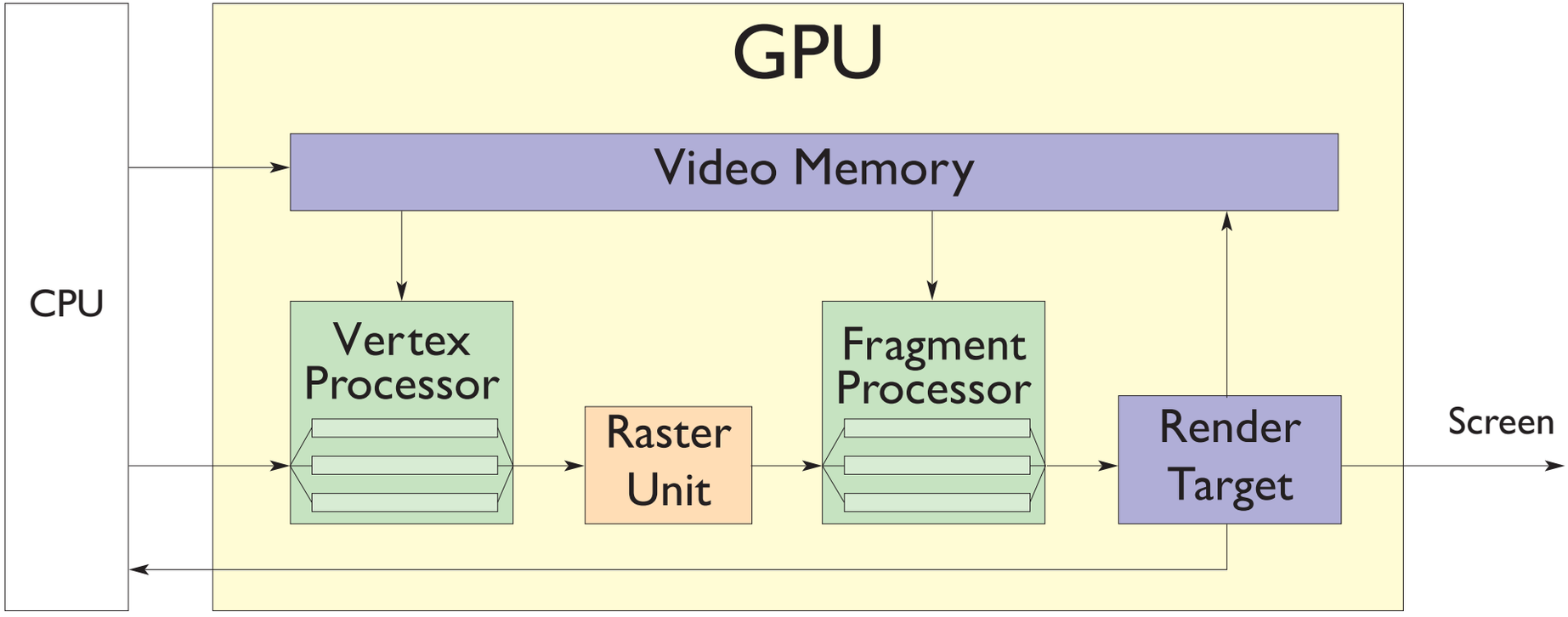
A brief Introduction into Programmable Graphics Hardware

- Hardware Graphics Pipeline
- Shading Languages
- Tools
- GPGPU
- Resources

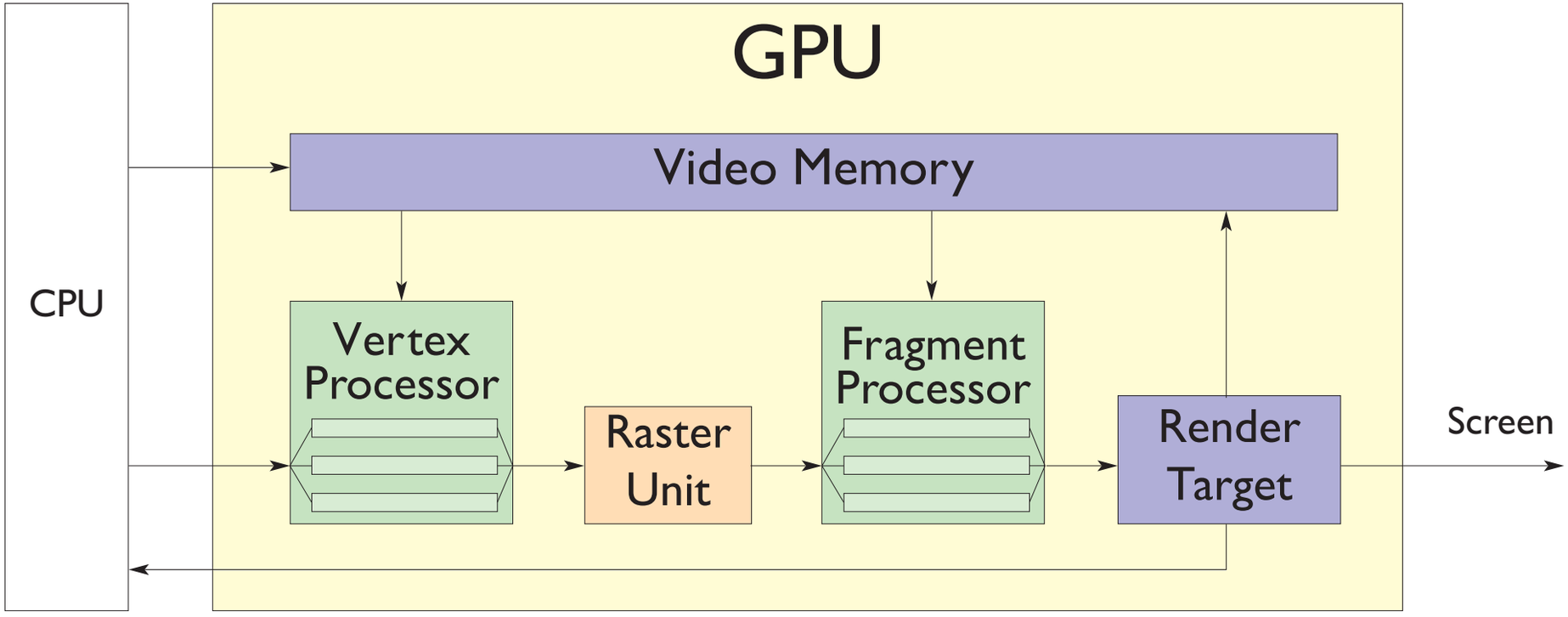
Hardware Graphics Pipeline

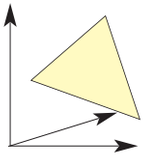
From vertices to pixels
... and what you can do in between.

Hardware Graphics Pipeline

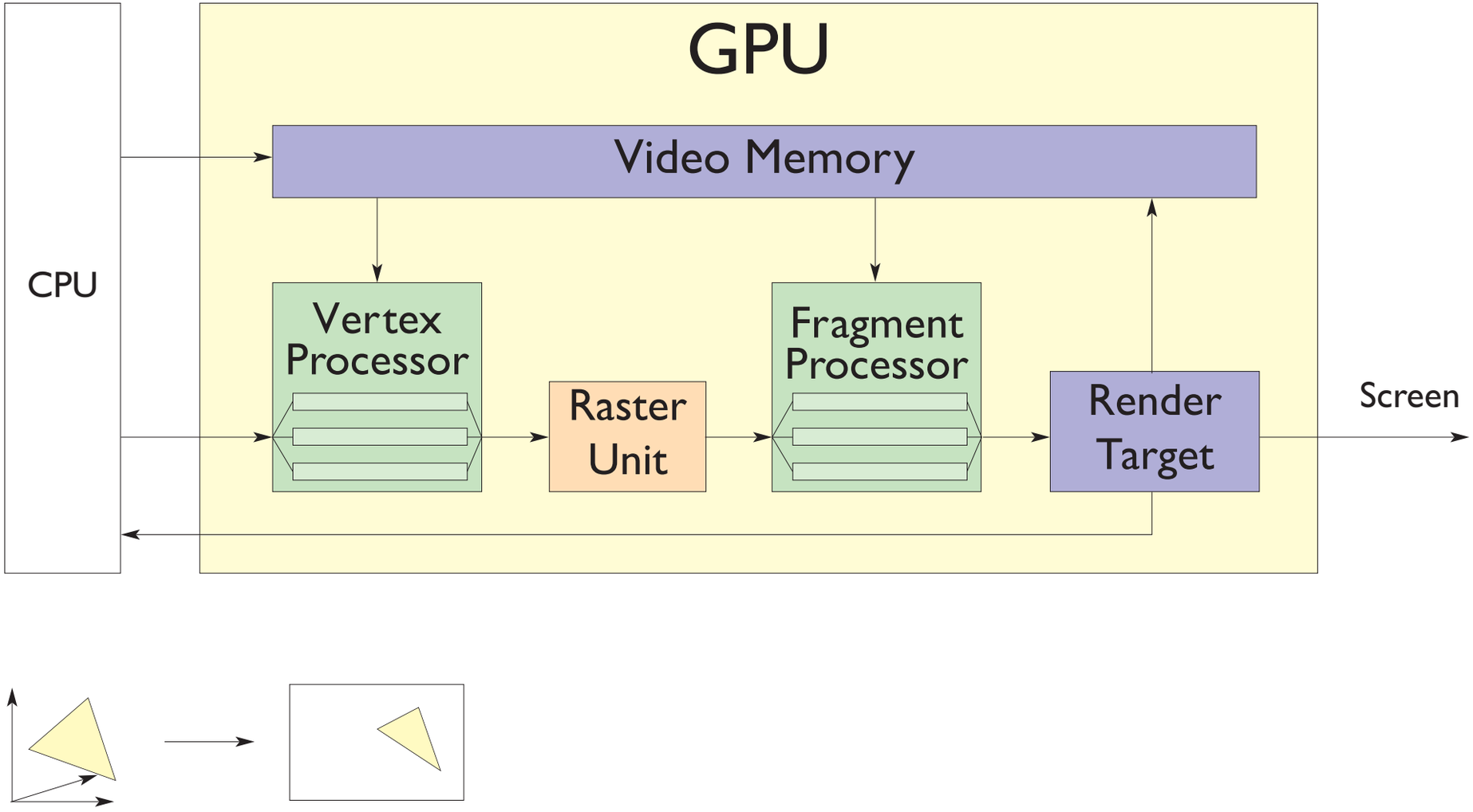


Hardware Graphics Pipeline

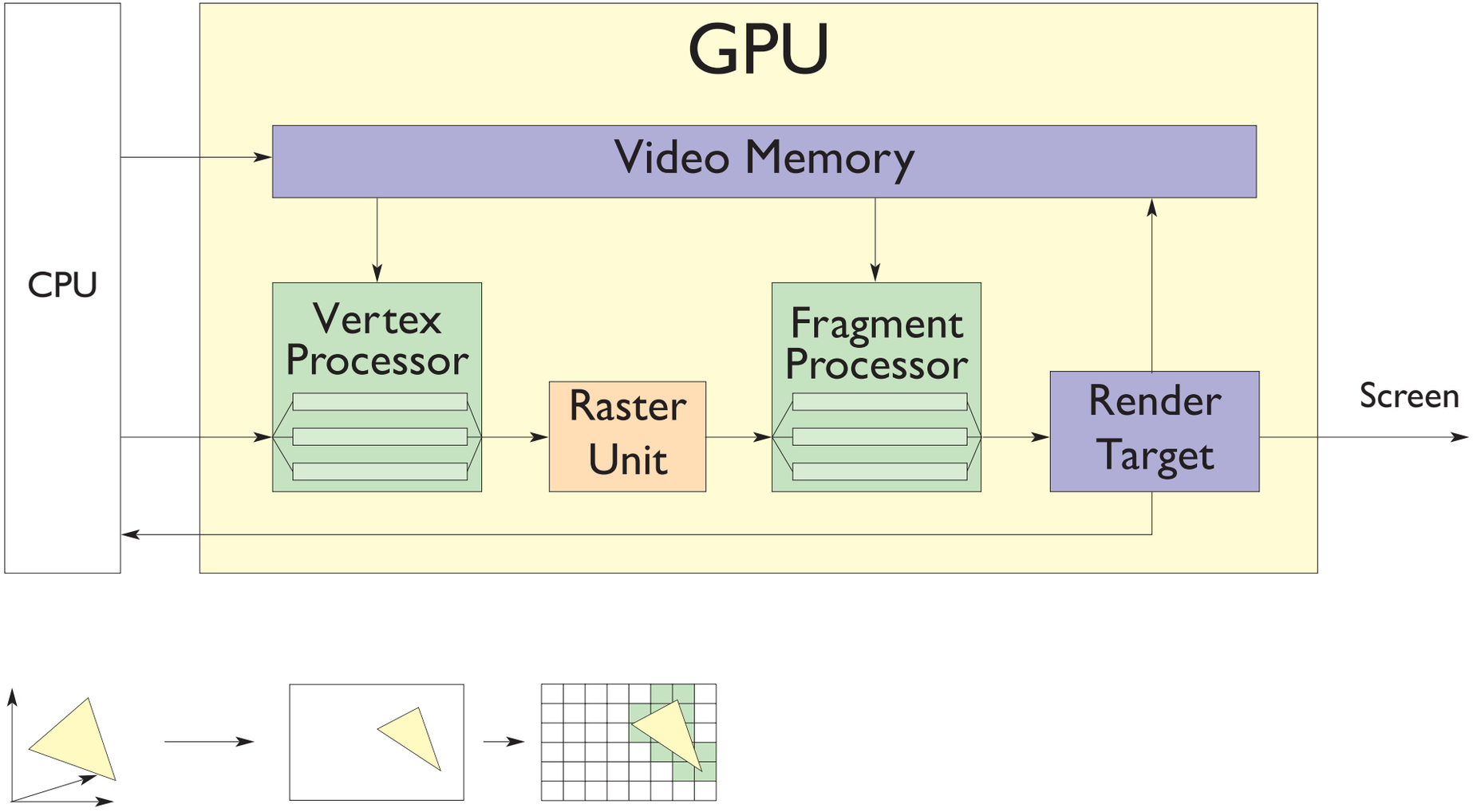




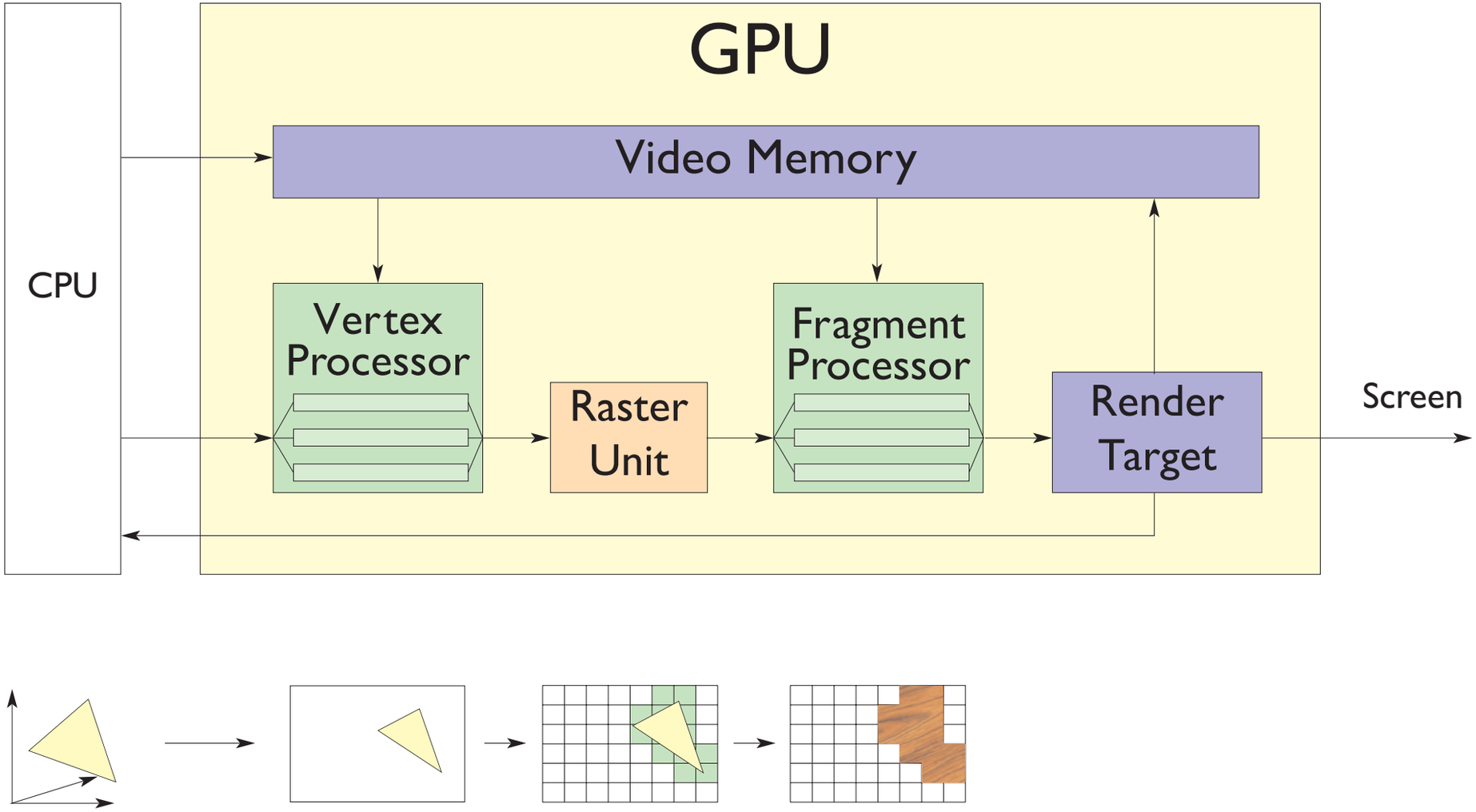
Hardware Graphics Pipeline



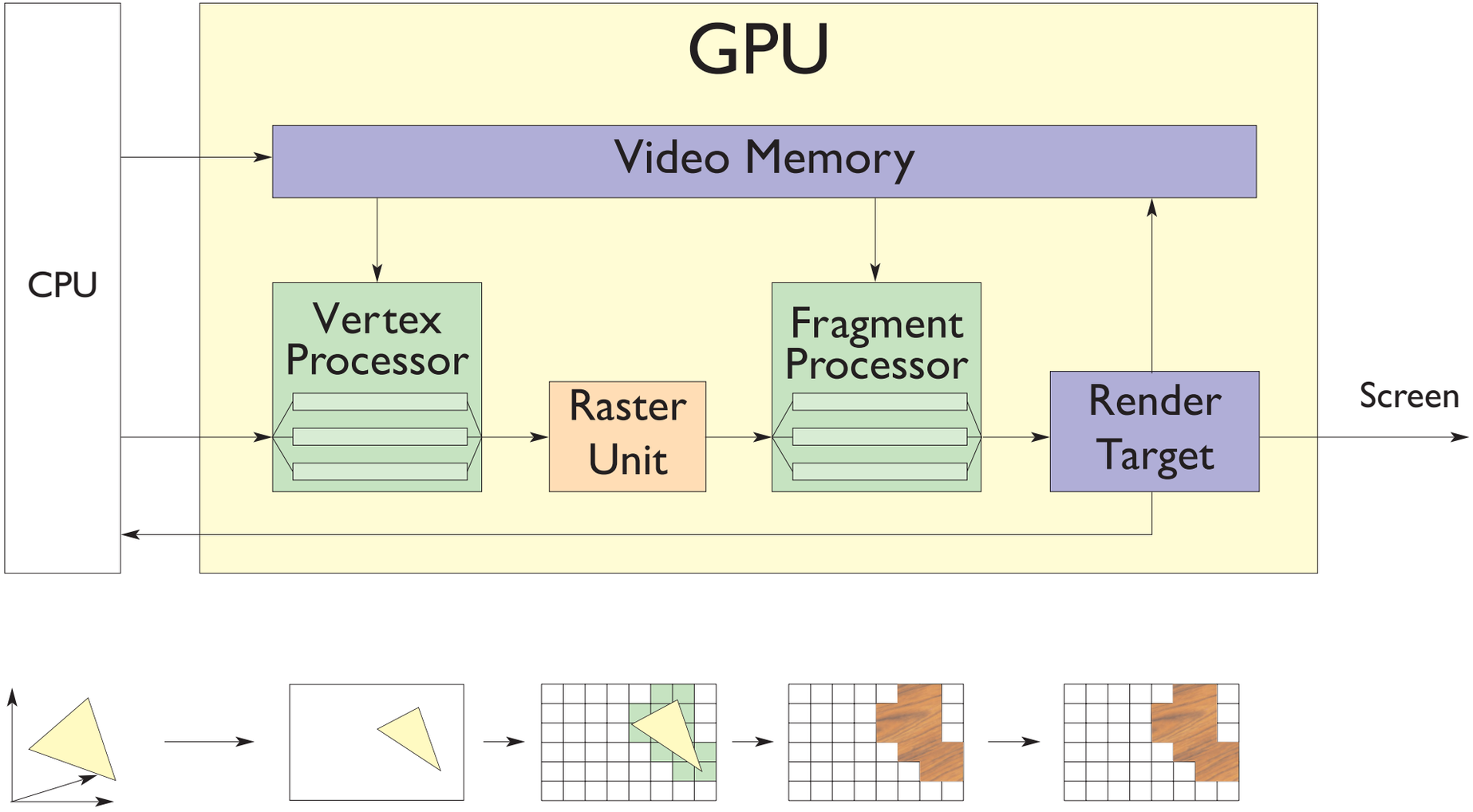
Hardware Graphics Pipeline



Hardware Graphics Pipeline



Hardware Graphics Pipeline



General Properties

- Strictly sequential processing (pipeline)
- Active vertex / fragment shader program is executed for each vertex / fragment
- Floating-point pipeline
  - No native support for integer / boolean data types
- GPU primitive
  - Vertex Shader: Vertex
  - Fragment Shader: Fragment
- Shader programs often called kernels

Fine-Grain Parallelism

- Each graphics primitive is considered as independent
- The program flow for each GPU primitive is considered as (almost) identical
- Multiple GPU primitives are processed in parallel
- Scheduling cannot be controlled by the user
- No simultaneous read / write of a buffer
- Multi-GPU solutions extend to high-level parallelism with explicit control for the user

Uniform Parameters

- Infrequently changing
- Available in every stage of the pipeline
- Built-in or user-defined
- Examples:
  - Model-View-Projection matrix
  - Textures
  - Parameters of shading model

Varying Parameter

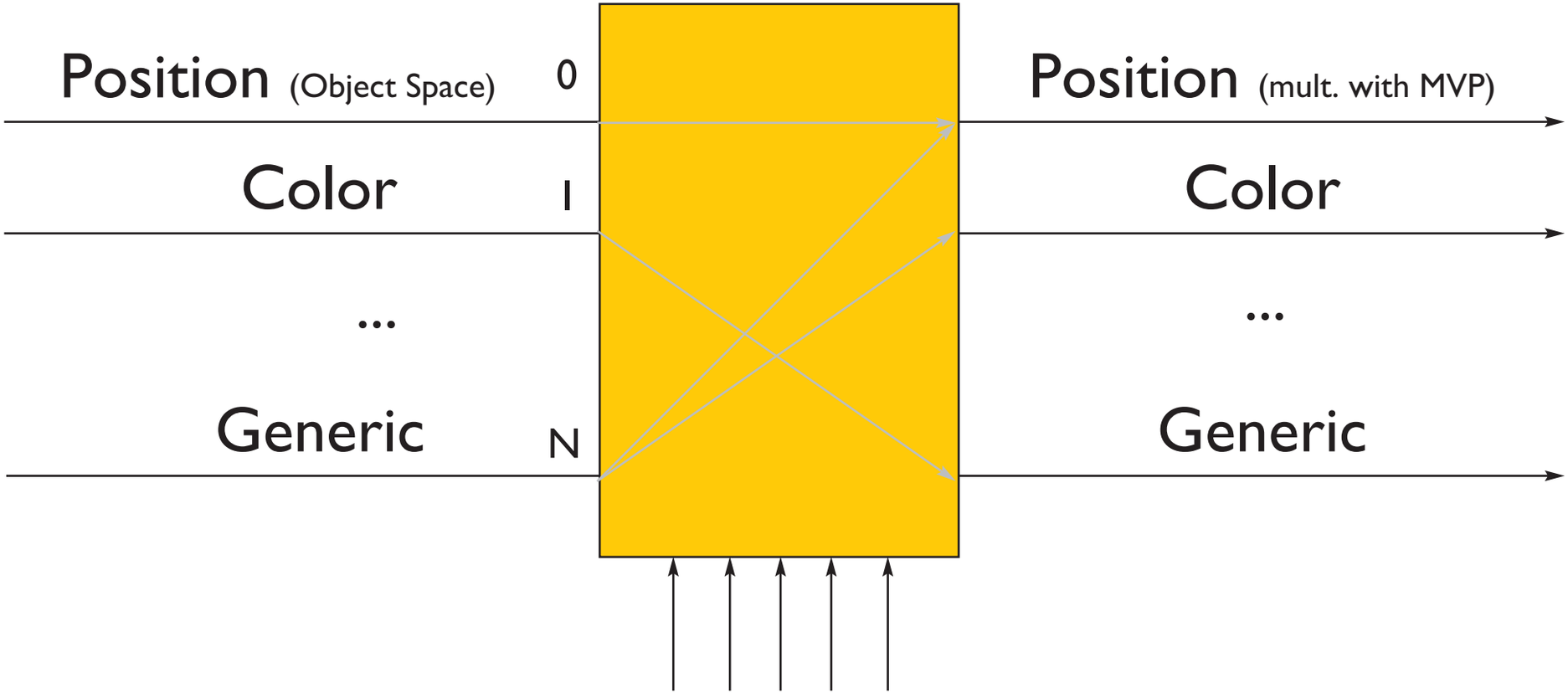
- Change for every GPU primitive
- Availability depends on the stage in the graphics pipeline
- Interpolation in the rasterization unit
- Built-in or user-specified
- Examples:
  - Vertex position
  - Vertex color
  - Fragment color

Vertex Processor

- Up to 8 parallel units
- MIMD architecture*
  - Units execute same program but operate independent
- Texture fetch possible*
  - High latency
  - Only 2D textures
  - Displacement Mapping
- Instruction slots: 512

* Nvidia NV4X architecture

Vertex Processor



Uniform Variables

MVP, Texture Matrix, Textures, User-defined, ...

Functionality

- Vertex transformation
- Normal, texture coordinate transformation
- Lighting
- Point size computation

- Vertex position generation
- Texture coordinate generation
- Detail generation
- Skinning

Functionality

- Feedback loop
  1. Render-to-Texture
  2. Copy to PBO
  3. Use PBO as VBO

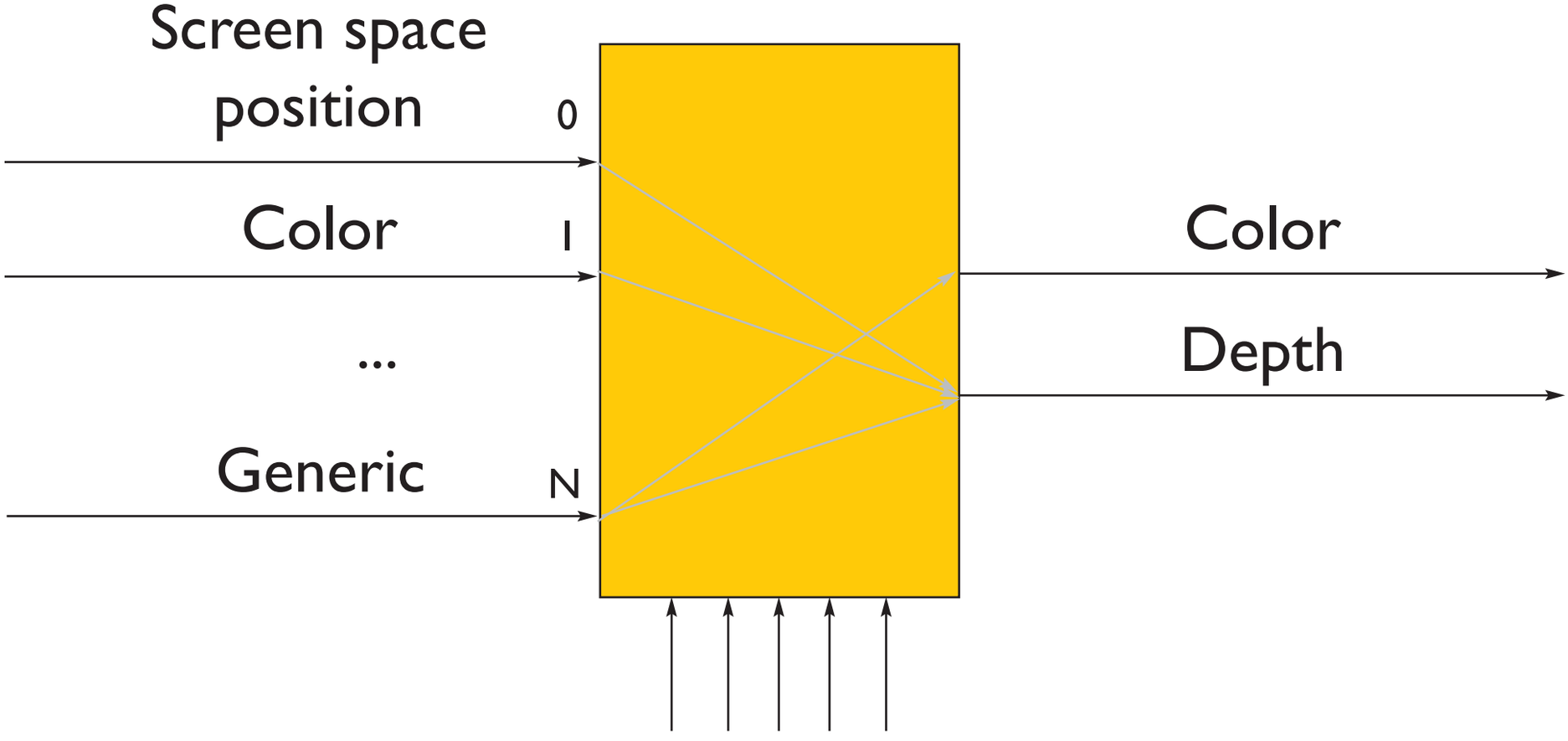
Rasterization Unit

- Perspective division
- Primitive assembly
- Clipping
- Interpolation of all varying parameters
- Rasterization
- . Early-Z culling (hierarchical and / or per fragment)

Fragment Processor

- Up to 48 parallel units
- Quad of pixels are processed together
- Very deep pipelines to hide texture fetch latency
- Effectively SIMD architecture
  - All occurring program paths have to be computed for for all fragments in a batch (4 x pipeline depth)
  - Incurs very high penalty for incoherent branching
- Instruction slots: 1024 - 4096

Fragment Processor



Functionality

- Very high floating-point computing power
- Fragment position fixed
- Computing / altering the depth value possible
- Texture lookups
  - Samplers = textures
  - Dependent texture lookups
  - 1D, 2D and 3D textures
  - Explicit mipmap level addressing

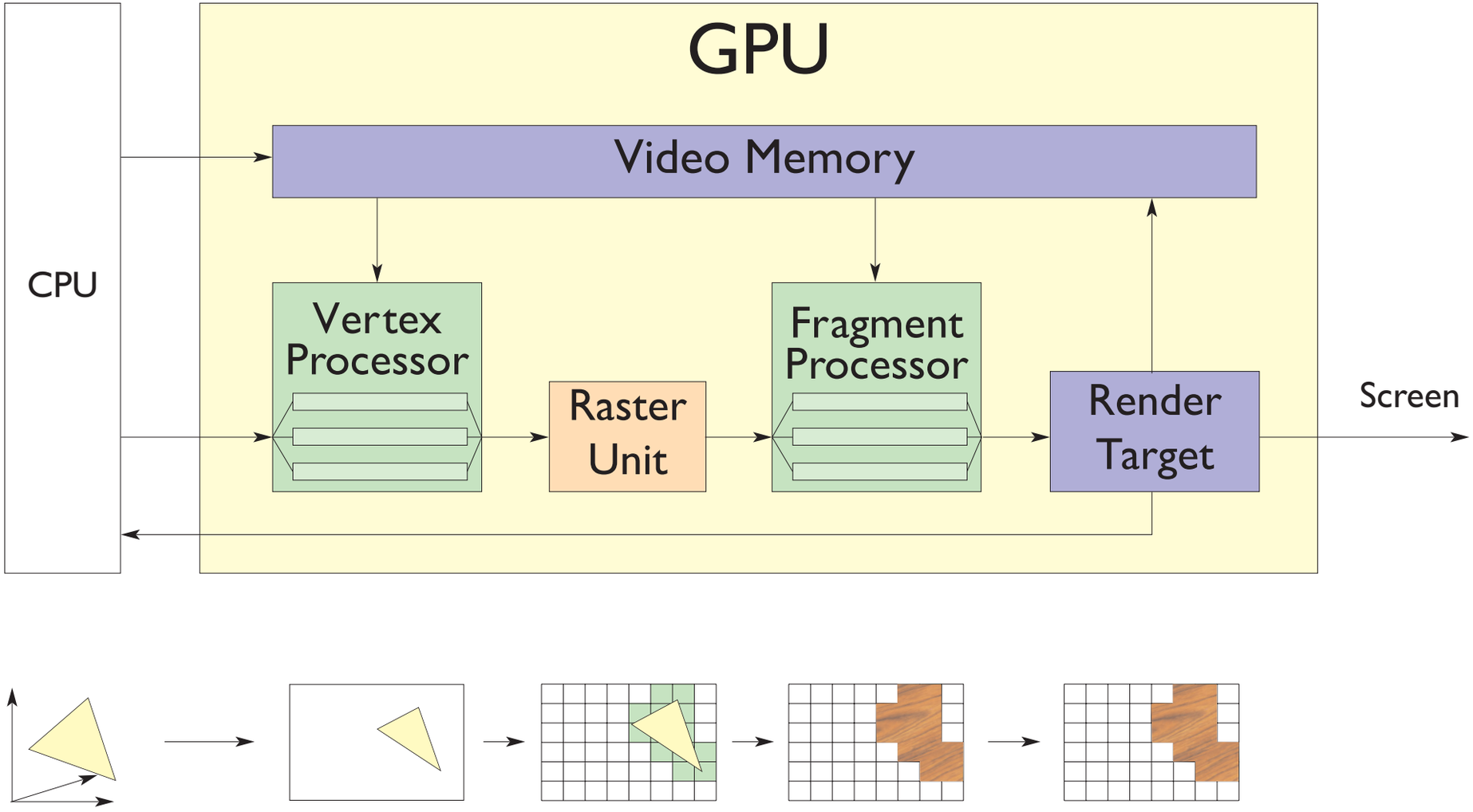
Functionality (cont'd)

- Result can be up to
  - 16 full precision floating point values
  - 32 half precision floating point values
  - 64 8-bit unsigned char values
- Feedback loop: Render-to-Texture
  1. Render to offscreen target with texture as attachment
  2. Use texture as data source in the next render pass

Framebuffer

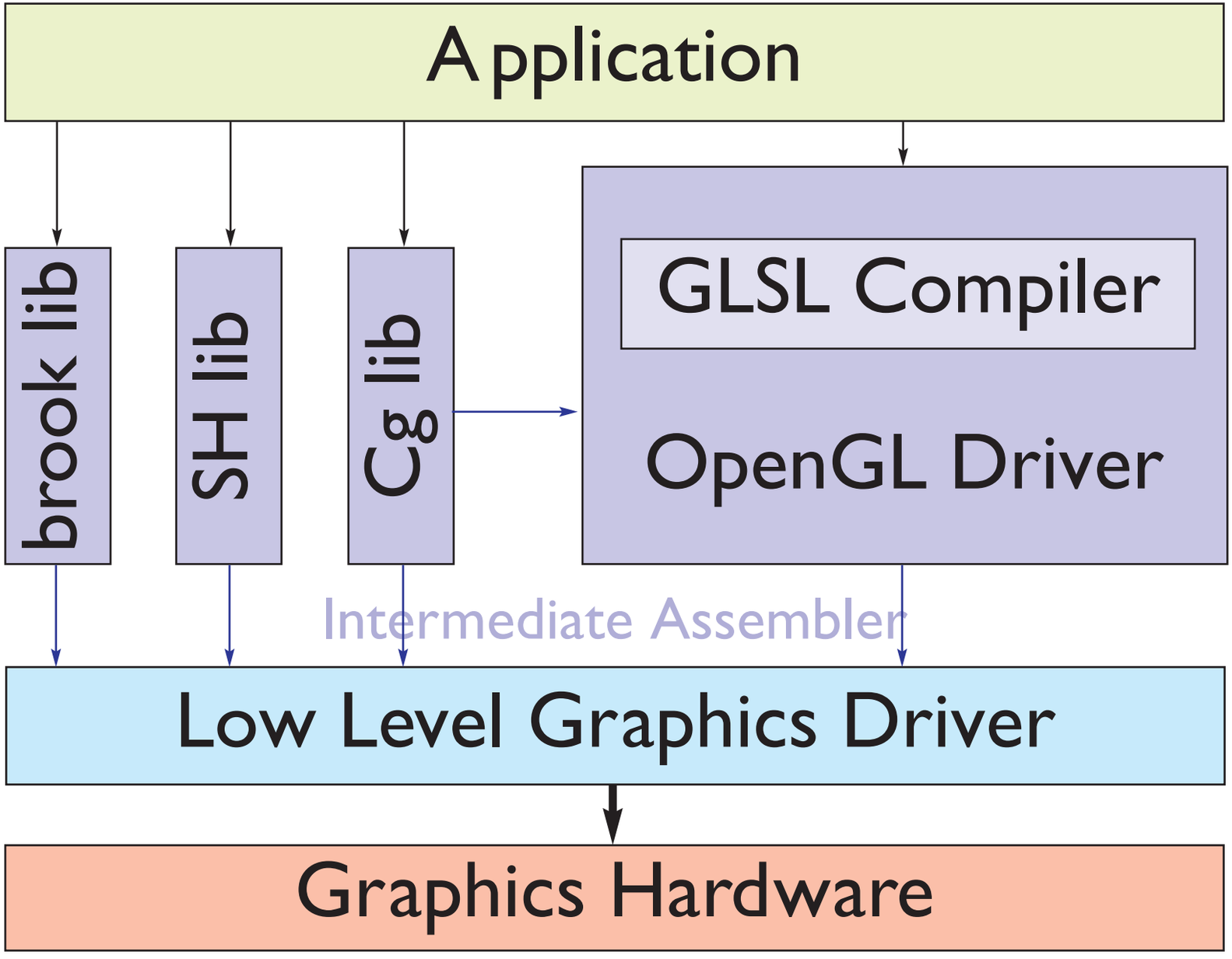
- Can be onscreen or offscreen buffer
- Can have up to four Color buffers (MRTs)
- Can have depth / stencil / accumulation buffer
- Onscreen render target: Provided by Windowing System
  - Visual determines properties
- Offscreen render target: Framebuffer object (FBO)
  - Attachments determine properties
- Various parameters can be set by the client API
  - `glBlendEquation(), ...`

Hardware Graphics Pipeline



High-Level Shading Languages

From smooth shaded, textured fragments
... to physical accurate BRDF on graphics hardware.


Application
brook lib
SH lib
Cg lib
GLSL Compiler
OpenGL Driver
Intermediate Assembler
Low Level Graphics Driver
Graphics Hardware

General Considerations

- Based on C/C++
- Additions to support graphics
- Several limitations compared to C/C++

Languages

- GLSL: OpenGL Shading Language
  - Part of OpenGL 2.0
  - Developed by 3DLabs, maintained by ARB forum

Languages (cont'd)

- Cg: C for Graphics
  - Developed and maintained by Nvidia
  - Graphics API independent
  - Proprietary
    - Can be used with ATI cards when compiling to ASM
- HLSL: High-Level Shading Languages
  - Developed and maintained by Microsoft
  - Shading Language for DirectX

Extensions for Graphics

- OpenGL state is available via built-in uniform variables
  - Availability depends on stage of the pipeline
  - `gl_ModelViewMatrix`
  - `gl_LightPos[ n]`
  - `gl_ClipPlane[ n]`
- Parameters of GPU primitive are available via built-in varying variables
  - Availability depends on stage of the pipeline
  - `gl_Position`
  - `gl_FrontColor`

Extensions for Graphics

- Data types for textures
  - `sampler1D`, `sampler2D`, `samplerRECT`, `sampler3D`
  - `texture1D()`, `texture2D()`, ...
  - `texture2Dshadow()`, `texture2Dshadow()`
    - Special lookup for shadow mapping
    - Perspective correct depth value comparison is performed automatically,
  - `sampler1D( name, tex_coord.st, mipmap_level)`
    - Mipmap level can be specified explicitly

Extensions for Graphics

- Additional type qualifiers
  - attribute (generic vertex attribute, varying type)
  - uniform
  - varying
- Built-in data types for vectors and (square-)matrices
  - `vec2`, `vec3`, `vec4`, `ivec2`, `ivec3`, `bvec2`, `bvec3`, ...
  - `mat2, mat3, mat4`

Extensions for Graphics

- Build-in functions for frequently needed functionality
  - `operator*( mat, mat)`, `operator*( mat, vec)`, `operator+( mat, mat)`
  - `dot()`, `cross()`, `normalize()`, `length()`
- Token `discard` to abort any further processing of a fragment
- Special functions for boolean operations on vectors
  - `any(), all()`

Restrictions (compared to C/C++)

- No double, unsigned int, ... data types
- No strings and character data types
- No dynamic memory allocation and no pointers
  - Array indexing has to be compile time constant
- No enums and unions
- No classes
  - No templates
- No bit-wise operations

Restrictions (compared to C/C++)

- No stack for registers
  - No context switches
  - No “real” function calls
- No switch statement, goto statement and labels
- Branching and looping available but instructions significantly more expensive than on a CPU processor

Example: Vertex Shader

```
uniform float diffuse;

// vertex shader entry point
void main() {

  // compute eye space position of vertex
  vec3 es_pos = gl_ModelViewMatrix * gl_Vertex;

  // apply normal transformation
  vec3 n = gl_NormalMatrix * gl_Normal;

  // compute light vector
  vec3 light_vec =
          gl_LightSource[0].position - es_pos;
```

Example: Vertex Shader

```
    // vectors have to be normalized
    light_vec = normalize( light_vec);
    n = normalize( n);

    // compute diffuse color contribution
    gl_FrontColor = max(dot(light_vec, n), 0.0)
                                        * diffuse;

    // apply texture transformation
    gl_TexCoord[ 0]  = gl_TextureMatrix[ 0]  *
                                    gl_MultiTexCoord0;
}
```

Example: Fragment Shader

```
// define texture
uniform sampler2D texture;

// fragment shader program entry point
void main() {

   // lookup texel
   float4 ctexel = texture2D( texture,
                              gl_TexCoord[ 0] .st);

   // modulate fragment color and texel
   gl_FragColor = ctexel * 0.5 + gl_Color * 0.5;
}
```

GLSL

+ Built-in uniforms
+ No separate libraries required
+ Extensions
+ Good portability (compared to Cg)
— No compiler flags (in the standard)
— No includes (in the standard)
— No control over intermediate assembler

Cg

+ Structs with member functions
+ Interface structs (Compile time polymorphism)
+ Command line flags
+ Faster evolution
— Setting up default unfiorms tedious and error prone
— Proprietary (but can be used on ATI / 3DLabs cards)

Miscellaneous

- Compiler still (very) buggy
  - Look at the assembler to verify correctness
- Test shader with offline compiler
  - `cgc` compiler from Nvidia (`-oglsl` for GLSL)
  - GLSLvalidate from 3DLabs
- Debug properly
  - Setup debugging environment before start coding

Other Languages for GPU Programming

- Motivation: Simplify GPU programming
- Abstract programming models (e.g. stream processor)
- Integration of shader programming into client side program code
- Hardware independent
  - Graphics API / Shading Language implementation have slight differences between vendors
  - Compilation for CPU (helpful for debugging) or other parallel architecture with similar capabilities possible

Other Languages for GPU Programming

- Brook
  - Developed at the graphics lab in Stanford
  - GPU as stream processor
  - Mainly for GPGPU applications
- SH
  - Developed at the graphics lab in Waterloo
  - Graphical and general purpose computations
  - Shader can be created at runtime using C/C++ language features (metaprogramming)

Tools

The things which come after printf debugging ...

Debugging / Profiling

- No direct debugging possible
  - No breakpoints
  - No stepping through a program
- GPU Printf Debugger: Readback framebuffer content and analyze
  - Writing content to an image often helpful
  - Floating point render targets if precision matters, e.g. FBO with float texture as color attachment

Debugging / Profiling (cont'd)

- GDebugger
  - OpenGL debugger
  - OpenGL state can be queried and compared / saved
  - Allows on-the-fly modification and re-compilation shader programs
  - Textures can be queried via an integrated image viewer
  - GUI integration for performance counter from 3DLabs and Nvidia, ATI will follow soon

Debugging / Profiling (cont'd)

- GPUBench
  - Developed at the graphics lab in Stanford
  - Benchmark for GPUs with focus on features which are important for GPGPU
  - Provides brief summary over GPU capabilities
- Profiling libraries
  - Available from 3DLabs and Nvidia
  - Can be integrated into application
  - For many purposes integration into GDebugger sufficient

Shader Creation Tools

- Avoid overhead to write a full OpenGL / DirectX application just to write a shader
- Provide easy-to-use interface to change shader parameter, set lighting, load textures, ...
- Nvidia: FXComposer
- 3DLabs / ATI: RenderMonkey
- Typhoon Labs: Shader Designer (Win / Linux)

What's coming next?

- Geometry shader
  - Located after vertex processor in the existing pipeline
  - Primitive information available
  - Can generate geometry on the fly
- Unified shader processor units
  - In hardware, no distinction between vertex, geometry and fragment shader
  - Enables arbitrary feedback loops
- Support for integer in hardware

Questions?

Resources

- *OpenGL Shading Language (2nd Edition)*, Randi J. Rost, The first chapters gives an introduction into the GPU graphics pipeline. The next chapters provide information about the OpenGL Shading Language, in particular on how the client side API works, how to write a first shader and then different shaders are explained and its implementation is discussed.
- *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics,* R. Fernando, M, J. Kilgard, Similar to the OpenGL Shading Language book by Randi Rost (see above) but it uses Cg as shading language.
- *www.gpgpu.org*, Homepage on General Purpose Programming on GPUs, Has many material and links related to GPUs in general ans GPGPU. The FAQ (http://www.gpgpu.org/wiki/FAQ) provides a profound resource for writing shaders.
- *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M. Pharr, R.Fernando, Six chapters discuss general programming models and techniques for GPUs as well as a concrete architecture. The remaining book is a collection of technical reports on advanced graphics and general purpose algorithms which use the GPU.
- *Programming Graphics Hardware*, R. Fernando, M. Harris, M. Wloka and C. Zeller, Tutorial Notes, Eurographics 2004, August 2004, Evolution of GPUs, its working principles and recent trends.
- *Reality Engine graphics*, K. Akeley, SIGGRAPH 1993, July 1993, Good explaination of the graphics pipeline (as implemented on SGI workstation) and the functionality of the different stages. The architecture discussed differs slightly from current GPUs but most discussions are still valid.

Resources (cont'd)

- *http://www.cs.unc.edu/Events/Conferences/GP2/*, GPGPU conference, the proceedings (can be found under program) provide an overview over applications possible on the GPU.
- *http://developer.3dlabs.com/openGL2/index.htm*, 3DLabs developer support homepages,
  - *http://developer.3dlabs.com/downloads/index.htm*, Code examples and helpful tools, e.g. offline parser for GLSL
  - *http://developer.3dlabs.com/documents/index.htm#Presentations*, Randi Rost's presentation *GLSL Overview* gives a concise overview about GLSL, its API and some examples (parts of this slides are based on his presentation)
  - *http://developer.3dlabs.com/documents/glsl manpage_index.htm*, GLSL API man pages online
- *http://developer.nvidia.com/page/home.html*, Nvidia's developer support website
  - *http://developer.nvidia.com/object/gpu_programming_guide.html*, Many useful tips on how to write fast GPU applications
  - *http://developer.nvidia.com/page/event_calendar.html*, Presentation slides from various conferences including Siggraph and Eurographics. Helpful as an introduction for topics covered, often with good examples.
  - *http://download.developer.nvidia.com/developer/SDK/Individual_Samples/samples.html*, Many code examples, each covers a specific feature, similar to the OpenGL redbook examples, programs use either Cg or GLSL, Source code for Windows available, partially also for Linux
  - *http://developer.nvidia.com/page/tools.html*, Various tools for developer, including Cg libraries, FXComposer, Nvidia photoshop plugin, tool to generate normal maps, profiling tools, ...
  - *ftp://download.nvidia.com/developer/cg/Cg_1.4/Docs/CG_UserManual_1-4.pdf*, The "How to get started" for Cg, containing explanation of the client and server side functionality and program examples.

Resources (cont'd)

- *http://ww.ati.com/developer/*, ATI's developer support website with code examples, demos, presentations, technical reports
- *http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=47,* Tutorial how to write a simple vertex shader and the corresponding client side code
- *www.libsh.org*, SH metaprogramming language
- *http://graphics.stanford.edu/projects/brookgpu/*, BrookGPU language
- *http://graphics.stanford.edu/projects/gpubench/*, GPU benchmark, check also the results section