The Graphical Pipeline

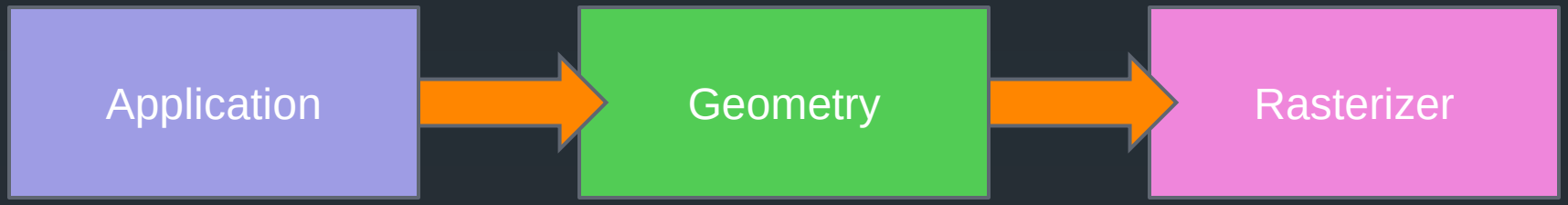
The Graphical Pipeline and the Graphics Processing Unit

Computer Graphics Basics

The Graphical Pipeline

The Graphical Pipeline

The theoretical graphical pipeline consists of three stages:

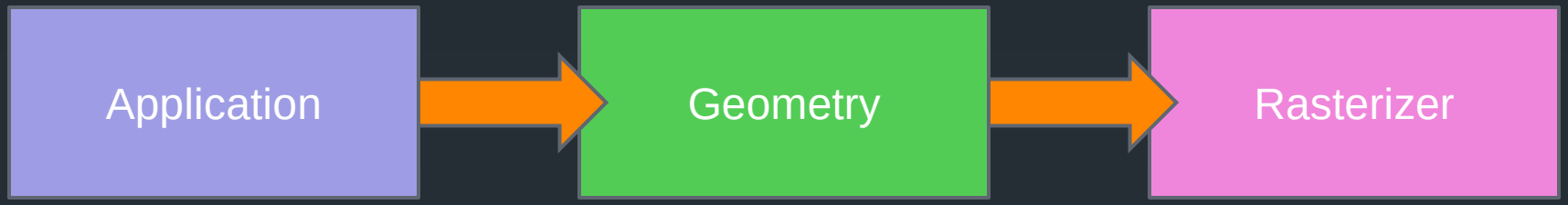


The Application Stage:

- This stage is the code stage, code is written in a high level language (C++, C#) using a graphics API and executes on the CPU.
- It controls and configures future stages in the pipeline e.g. blending, clipping, testing, etc.
- It is responsible for creating, initializing and updating all scene elements e.g. lights, camera, geometry, textures, etc.
- This stage sends geometric data (vertex lists) down the pipeline using draw calls.

The Graphical Pipeline

The theoretical graphical pipeline consists of three stages:

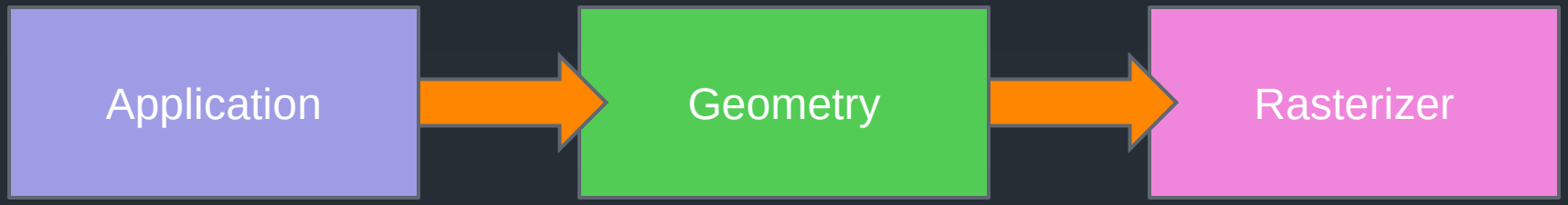


The Geometry Stage:

- This stage modifies the geometric data (vertices) sent down the pipeline by applying geometric transformations, vertex lighting, etc.
- The vertex and geometry shaders play a large role in the geometry stage. They are written in a High Level Shading Language (HLSL, GLSL, Cg) and are executed by the GPU's stream processors.
- This stage is also responsible for view projection, clipping and screen mapping.

The Graphical Pipeline

The theoretical graphical pipeline consists of three stages:



The Rasterizer Stage:

- This stage is responsible for producing the displayed image. It does this by converting the 3D geometric data into a 2D color image.
- The pixel shader calculates color values for pixels. It is written in a High Level Shading Language (HLSL, GLSL, Cg) and is executed by the GPU's stream processors. It is responsible for all texturing and per pixel lighting operations.
- This stage also performs fragment (similar to a pixel) testing and pixel blending.

The Modern GPU pipeline and the DirectX10 pipeline

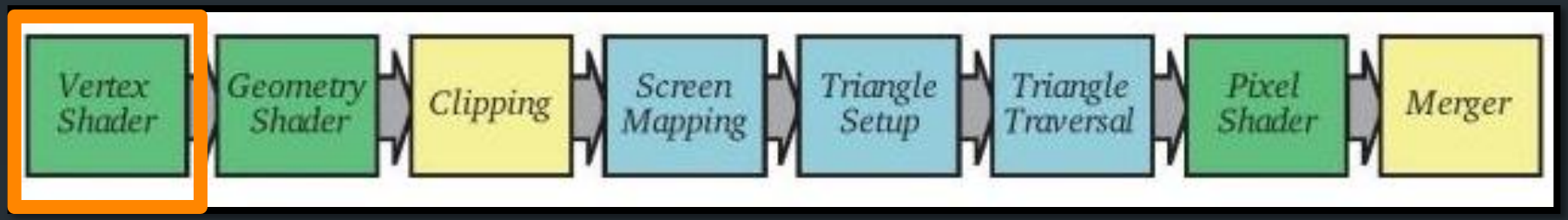
The Modern Graphical Pipeline

The Modern Graphical Pipeline

- Above is the rendering pipeline found on most modern GPUs.

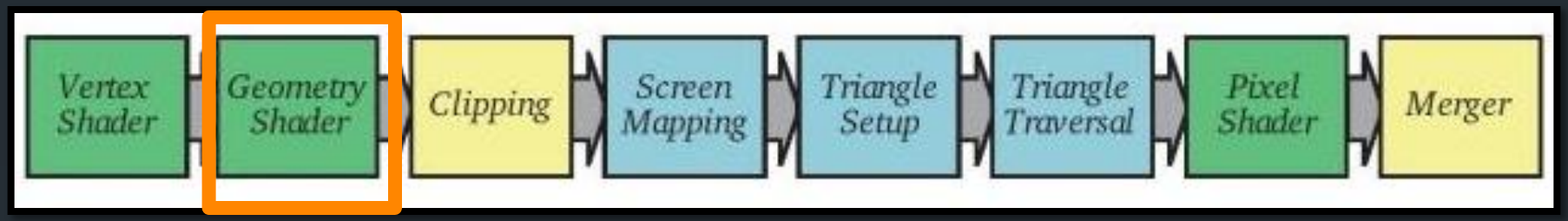
- Green stages are fully programmable using shader languages such as HLSL/Cg/GLSL
- Yellow stages are configurable through the graphical API but not programmable
- Blue stages are completely fixed in their function

MGP: Vertex Shader



- Processes every vertex sent down the pipeline independently. Incoming streams of vertices can be processed in parallel on the GPU's multiple shader processors.
- Cannot create or destroy vertices. It can only modify or add vertex data.
- The vertex shader stage is commonly used to apply transformations and deformations to geometry.

MGP: Geometry Shader



- Only present in SM 4.0+
- Takes a single primitive (point, line, triangle) and its defining vertices as input (may also take in adjacency info).
- It can generate additional vertices per primitive and so modify the primitive or generate new primitives from it.

MGP: Clipping & Screen Mapping

- After the vertices have been processed by the VS/GS stages, they are tested against the canonical view volume (clipping) and vertices that pass are sent down the pipeline for rasterization.
- The screen mapping stage is the first stage in the rasterization pipeline, it maps the horizontal/vertical dimensions of the canonical view volume to the dimensions of the currently active viewport.

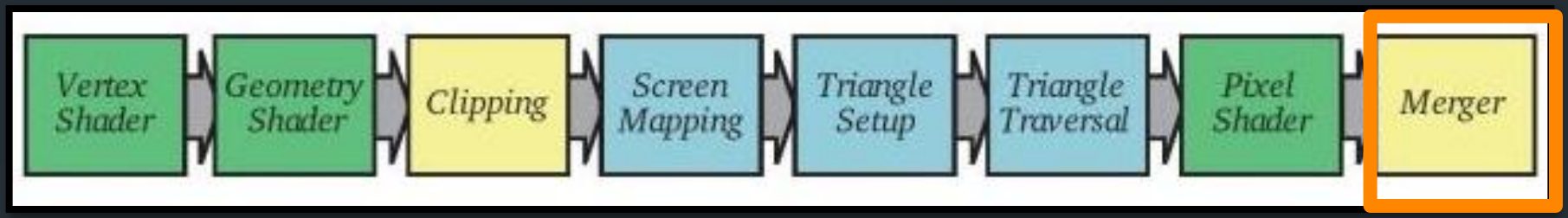
MGP: Triangle Setup & Traversal

- This stage take vertices as input and constructs triangles from them. Triangles are constructed from vertex streams using a primitive topology (triangle list, triangle strip).
- Triangle traversal moves over a triangle's surface determining which pixels in the frame buffer will be affected by it. For each affected pixel, a fragment is created containing the affected pixel's position, a depth value and any additional data (from the triangles vertex data) the pixel shader requires. Vertex data interpolation occurs here.

MGP: Pixel Shader

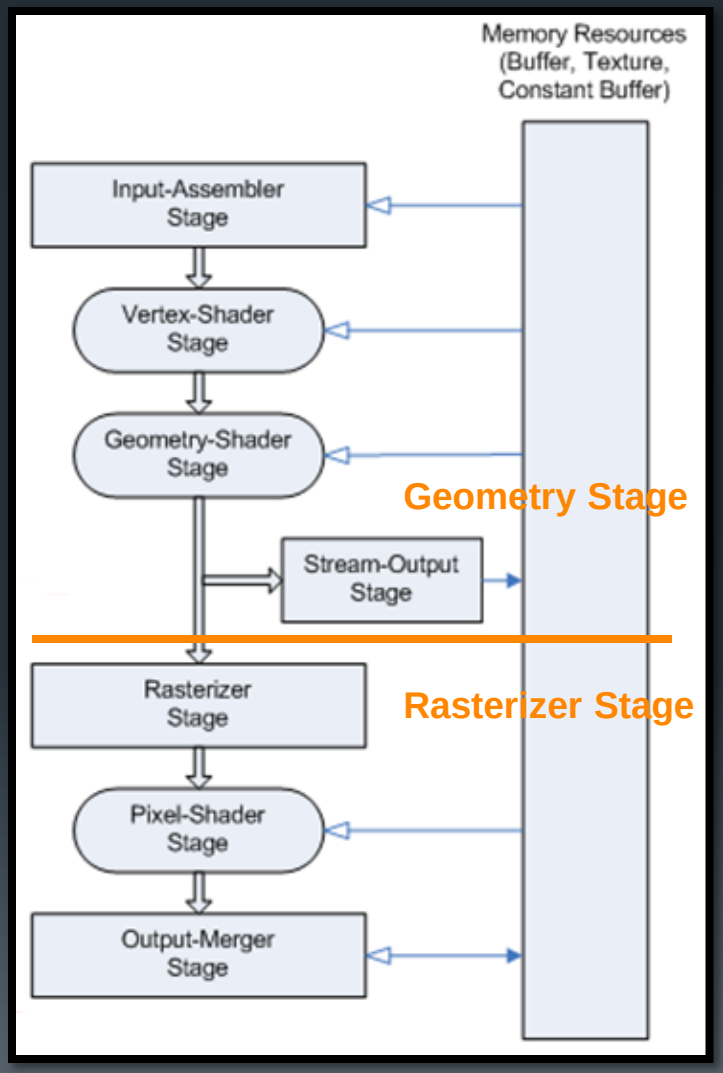
- Receives fragments as input. Fragment data contains all necessary information needed for lighting and texturing calculations.
- The pixel shader calculates the final fragment color, this color is used in the final merging stage.
- The depth value generated in earlier stages (triangle traversal) can also be modified by the pixel shader.

MGP: Merger Stage



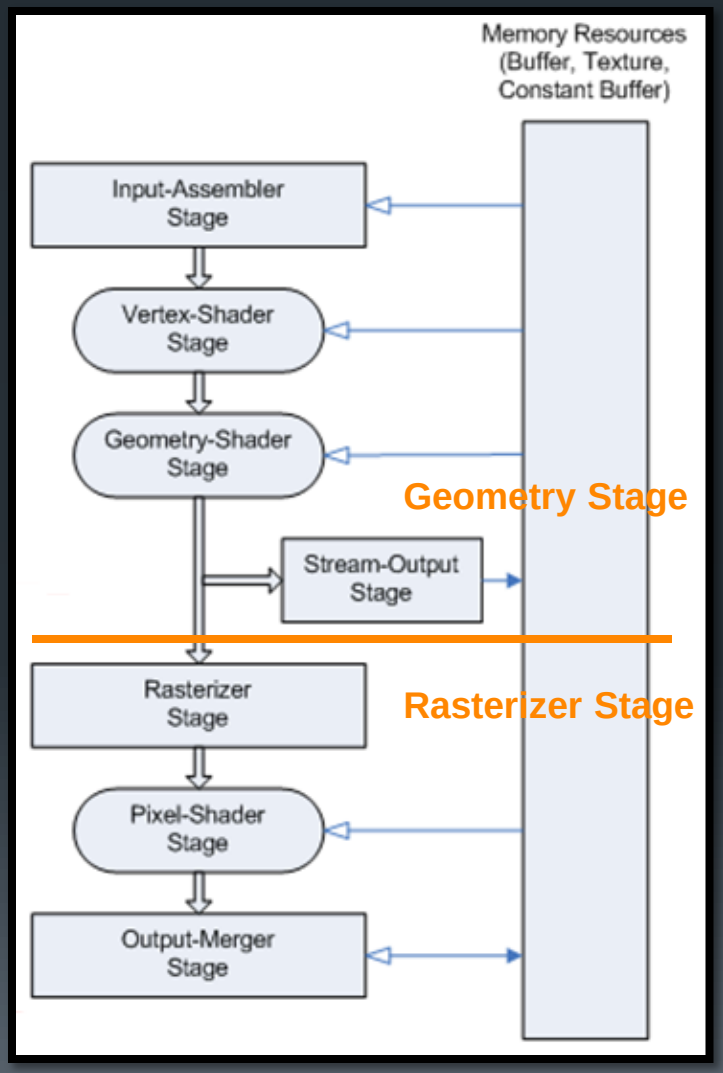
- This is the final stage, pixel testing and blending occurs here.
- Fragments are tested against certain parameters (e.g. depth testing) and if they pass they are sent to the blending stage.
- Fragments refer to specific pixels, and if they pass the testing stage. Their color values are blended with the existing pixel color (in the frame buffer). This blending is user configurable.
- While this stage is not programmable, it is highly configurable in regards to how testing and blending occurs.

The DirectX10 Pipeline



- Input-Assembler (IA): Reads input data from user created buffers, assembles and optimizes vertex buffers to be used by other stages.
- Vertex Shader (VS): Does all vertex operations and calculations on each vertex sent to it by the IA stage
- Geometry Shader (GS): The GS is used to generate new geometry

The DirectX10 Pipeline



- Rasterizer (RS): converts vector information into a raster image (primitives → fragments).
- Pixel Shader (PS): per-pixel operations like lighting and post processing.
- Output Merger (OM): Generates the final rendered pixels from a combination of testing and blending.

Programmable Shaders & GPU Architecture

The Graphics Processing Unit

Programmable Shaders: History

- Pre-2001 there was no hardware support for programmable shaders, software attempts (pixar's renderman) made use of multi-pass rendering to simulate shaders. ID software's Quake III scripting language was one of the first commercial successes of such a software shading system.
- In 2001, the geforce3 was released supporting Shader Model (SM) 1.1. This allowed for basic programmability using an assembly-like language and was very limited (only allowed 12 instructions per program and had no flow control)
- In 2002 SM2.0 (geforce4, ATI 9 series) was released allowing for fully programmable shaders. Microsoft and NVIDIA released HLSL/Cg. SM2.0 added flow control and so allowed for complex shader programs. These languages were based on the c syntax and featured elements from pixar's renderman shading language.

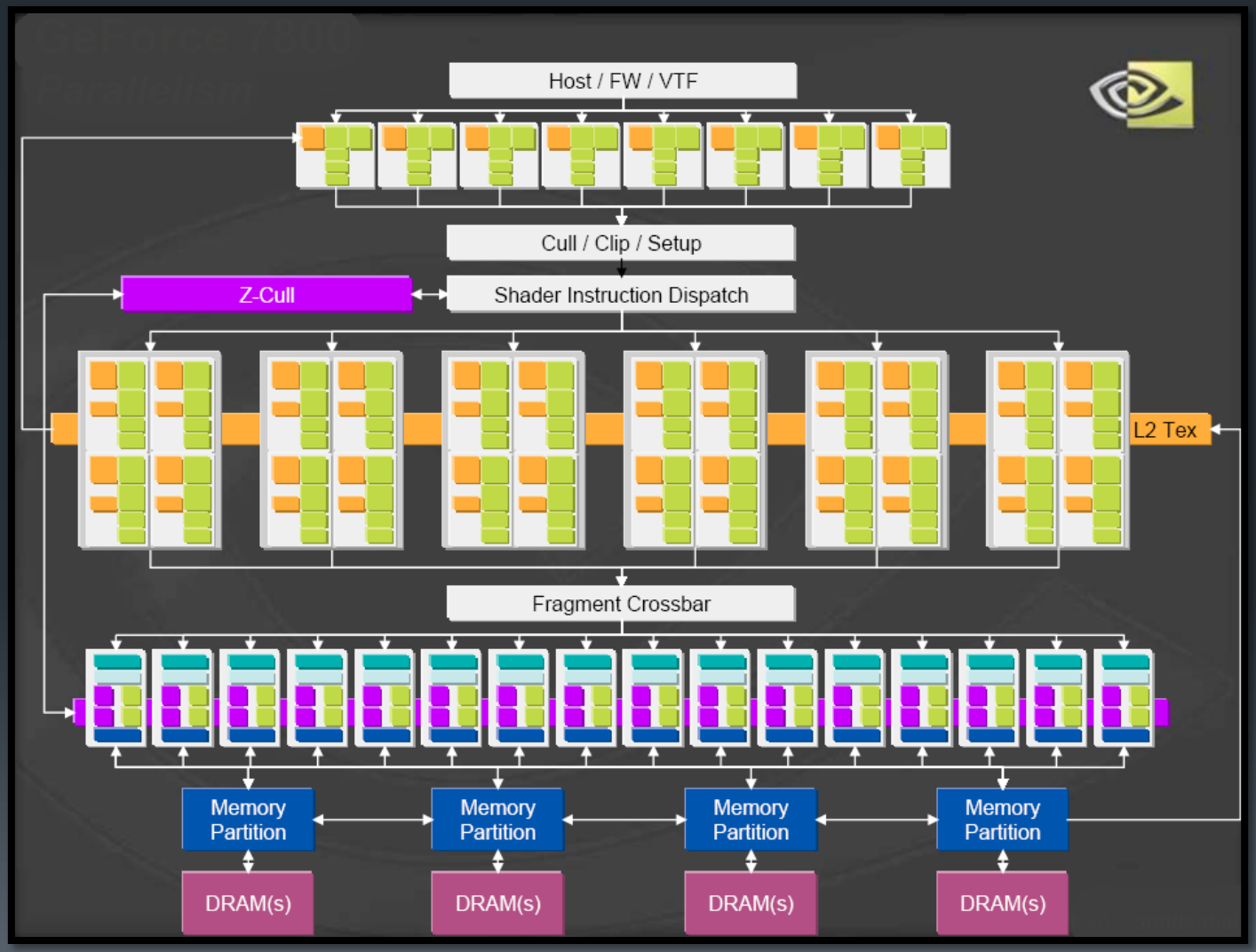
Programmable Shaders: History

- In 2004, SM3.0 was released (geforceFX, ATI X series), SM3.0 was an incremental improvement, turning optional features into requirements and adding limited support for texture reads in vertex shaders. Current gen consoles all have SM3.0 capable hardware (Xbox 360/PS3).
- In 2007, DirectX10 was released with SM4.0, this release offered a uniform programming model for all shaders (unified shader model), resource limits were increased. Integer data types and bitwise operators were included. HLSL moved closer to becoming a fully fledged programming language. Most 3D applications had moved to a heavy reliance on programmable shaders.
- In 2009, DirectX11 is released featuring SM5.0. SM5.0 adds another shader stage: the compute shader and offers dynamic linking in HLSL. Most important DX11 is a superset of DX10 and can be run with a reduced feature set on DX10 (and DX9) hardware by querying the device capabilities.

GPU Architecture

- The GPU is simply a hardware implementation of the graphical pipeline.
- As the pipeline has evolved so has the GPU. Early GPUs were nothing more than a collection of raster operators (ROPS) and geometry engines with very limited capabilities
- With the introduction of programmable shaders, GPUs evolved to contain specialized vertex and pixel shader processors

GPU Architecture - NVIDIA 7800

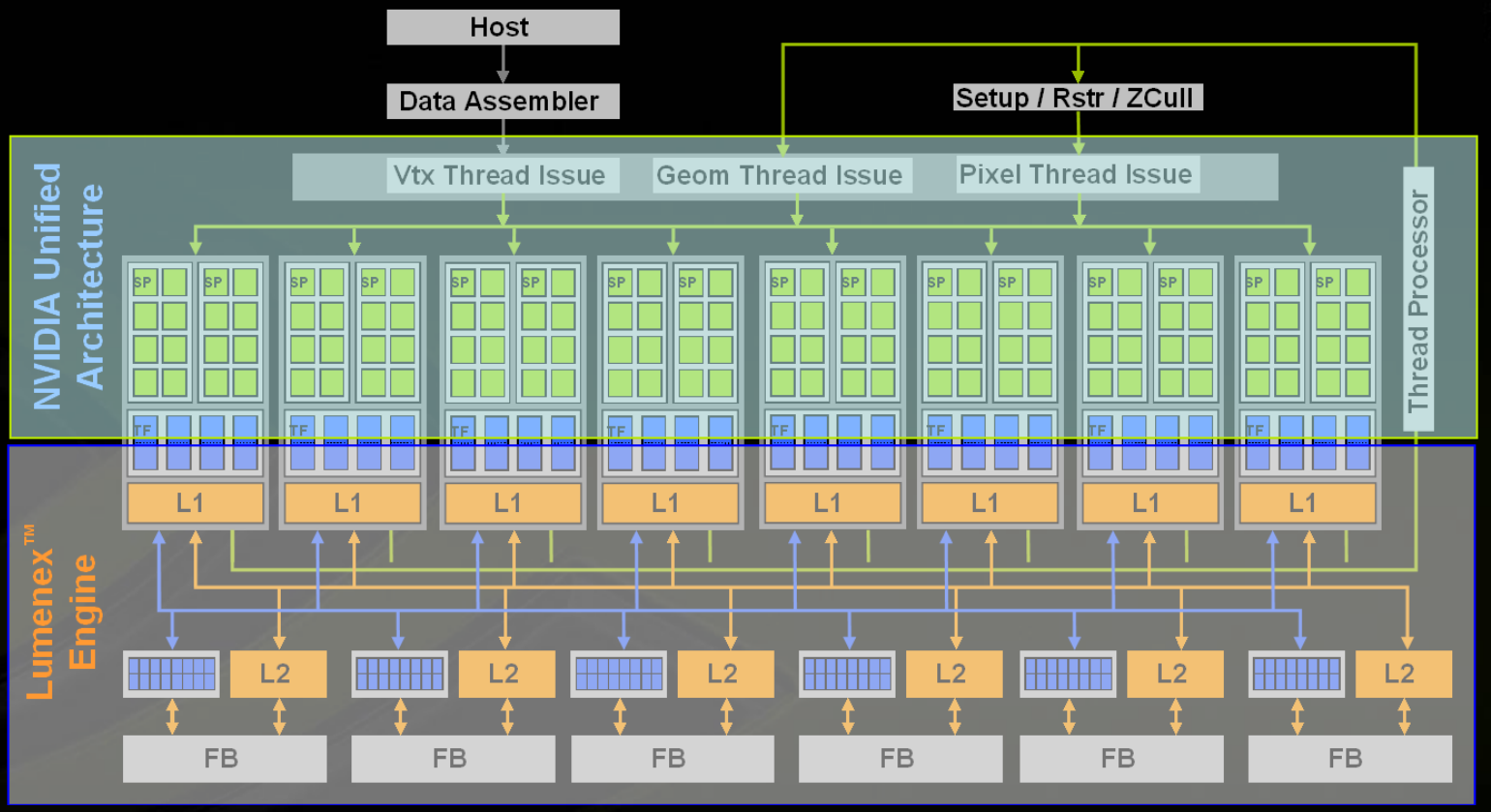


GPU Architecture

- From shader model 4.0, GPUs have moved to a unified shader model. This means that the various shader programs (vertex/pixel) are all programmed in a similar fashion
- GPUs could replace their specialized shader processors with generalized stream processors.

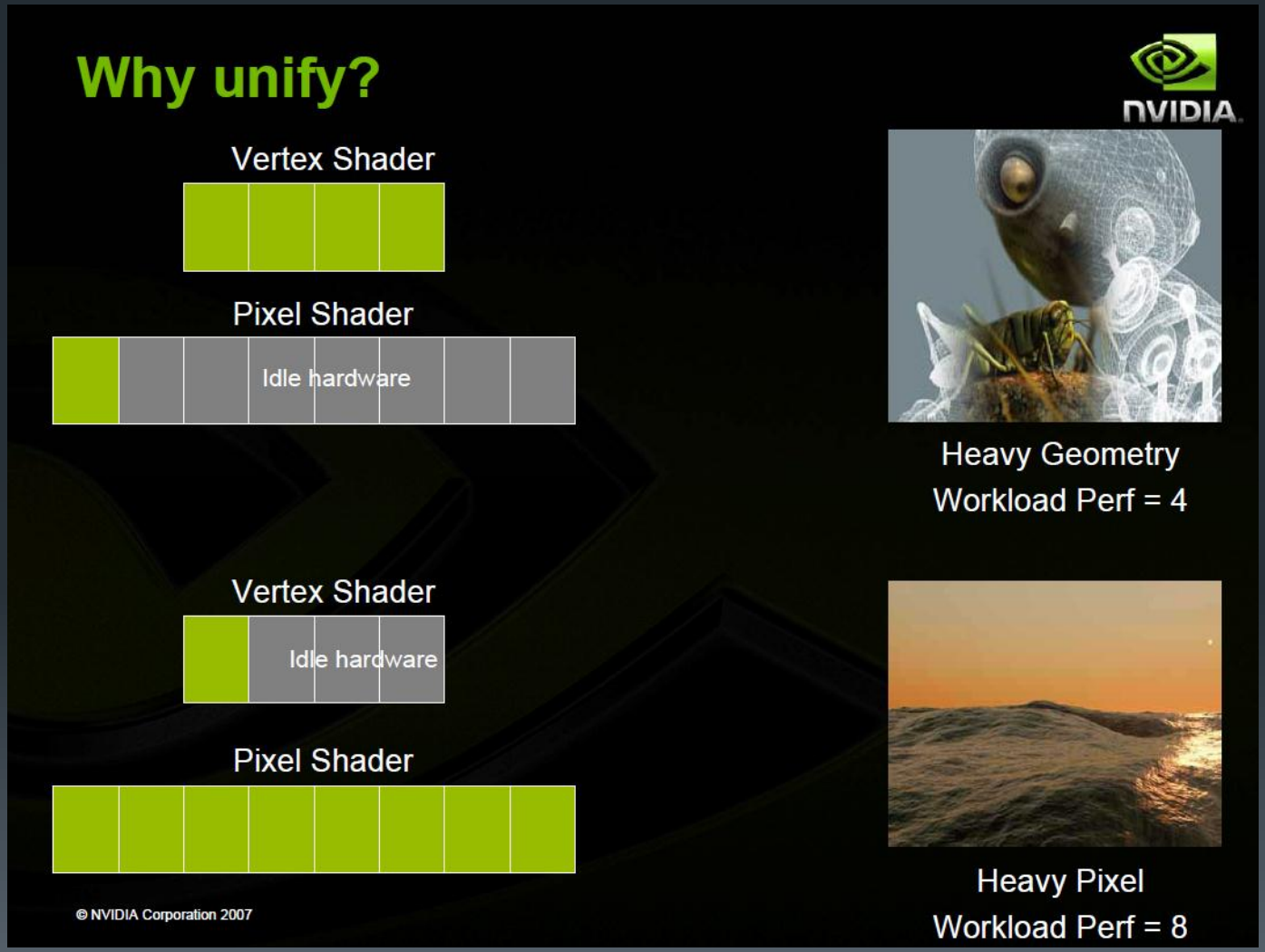
GPU Architecture – NVIDIA G80

GeForce 8800 Architecture



- Lumenex™ Engine Key Technologies:
  - 128-bit HDR
  - 16x AA
  - HDR+AA

GPU Architecture: Unified Shaders



Slides from GPU Architecture: Implications & Trends - David Luebke, NVIDIA Research – SIGGRAPH 2008

GPU Architecture: Unified Shaders

Why unify?

Unified Shader

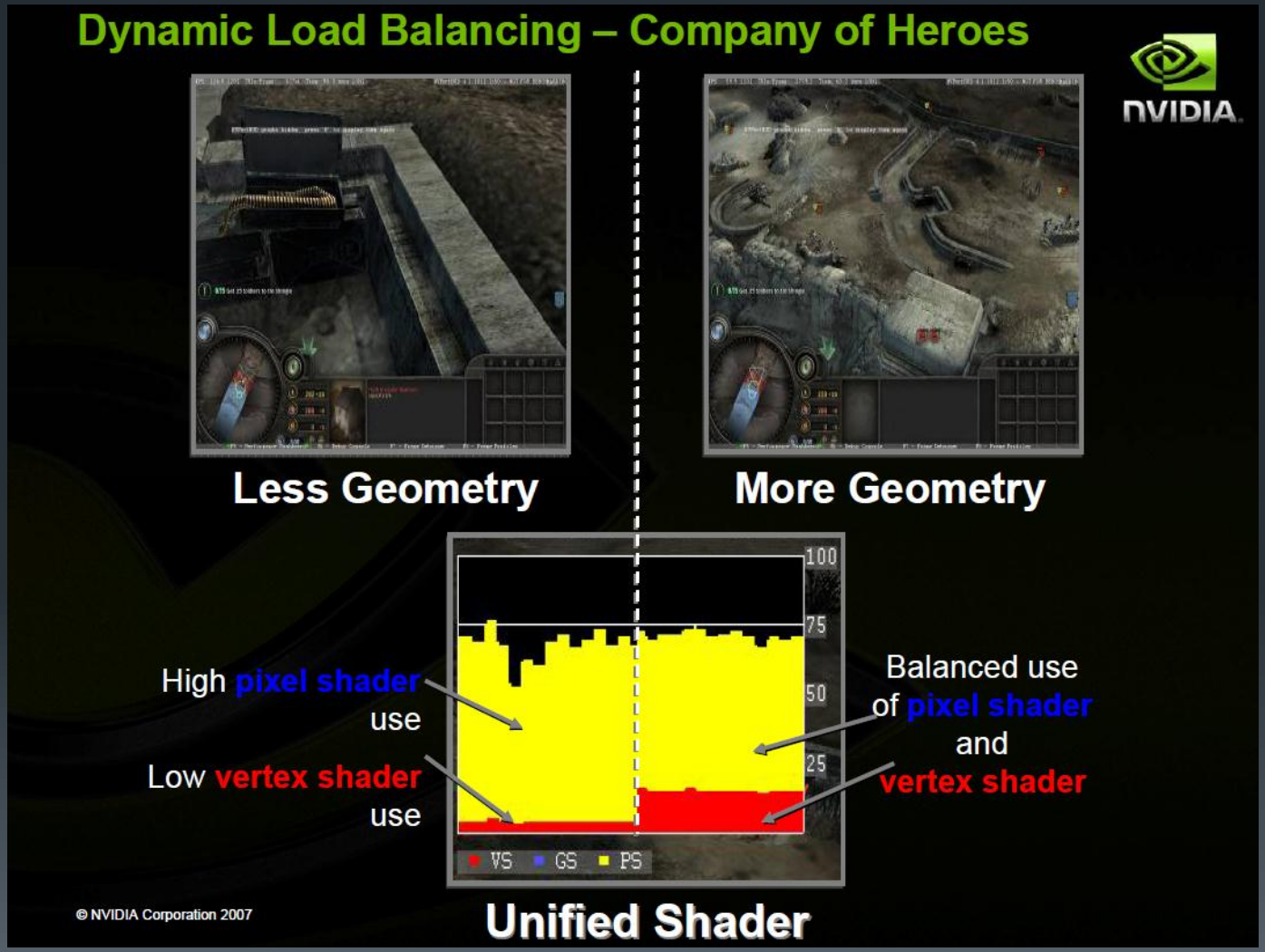
Heavy Geometry
Workload Perf = 11

Unified Shader

Heavy Pixel
Workload Perf = 11

Slides from GPU Architecture: Implications & Trends - David Luebke, NVIDIA Research – SIGGRAPH 2008

GPU Architecture: Unified Shaders



Slides from GPU Architecture: Implications & Trends - David Luebke, NVIDIA Research – SIGGRAPH 2008

The Modern GPU – The GTX 580

16 Stream multiprocessors (SMs), each containing 32 stream processors.

512 stream processors total.

<u>Processing power:</u>

GTX580: 1581 GFLOPs

i7 2600k: 120 GFLOPs

Conclusion

- The graphical pipeline describes the process that geometry goes through before being finally rasterized to the display.
- The graphical API is a software implementation of the modern graphical pipeline which controls and configures the hardware implementation (the GPU).
- GPU architecture has changed significantly over the last decade. The GPU has moved from consisting of specialized hardware for specific tasks like vertex processing to a more unified approach containing general purpose stream processors.
- The move to a unified architecture has simplified the programming of shader programs greatly.