NYU Center for Data Science

Week 13.1: Graphics processing units

DS-GA 1004: Big Data

This week

- **Graphics processing units (GPUs)**
- GPGPUs and CUDA
- Software frameworks

Example: gradient descent (serial)

$\mathbf{min}_w \sum_n f(x_n ; w)$

- Initialize w
- **for** i ← 1 **to** ITERATIONS:
  - Initialize $G \leftarrow 0$
  - **for** n = 1 .. N:
    - $G \mathrel{+}= \nabla_w f(x_n ; w)$
  - $w \leftarrow w - G$

Total time:

ITERATIONS * N * [per-point gradient cost]

Example: gradient descent (Spark version)

```
val points = spark.textFile(...).map(parsePoint).cache()
var w = Vector.random(D)

for (i ← 1 to ITERATIONS)
        val grad = spark.accumulator(new Vector(D))

        for (p ← points)
                val grad_p = ∇w f(p ; w)
                grad += grad_p

        w -= grad.value
```

Total time:

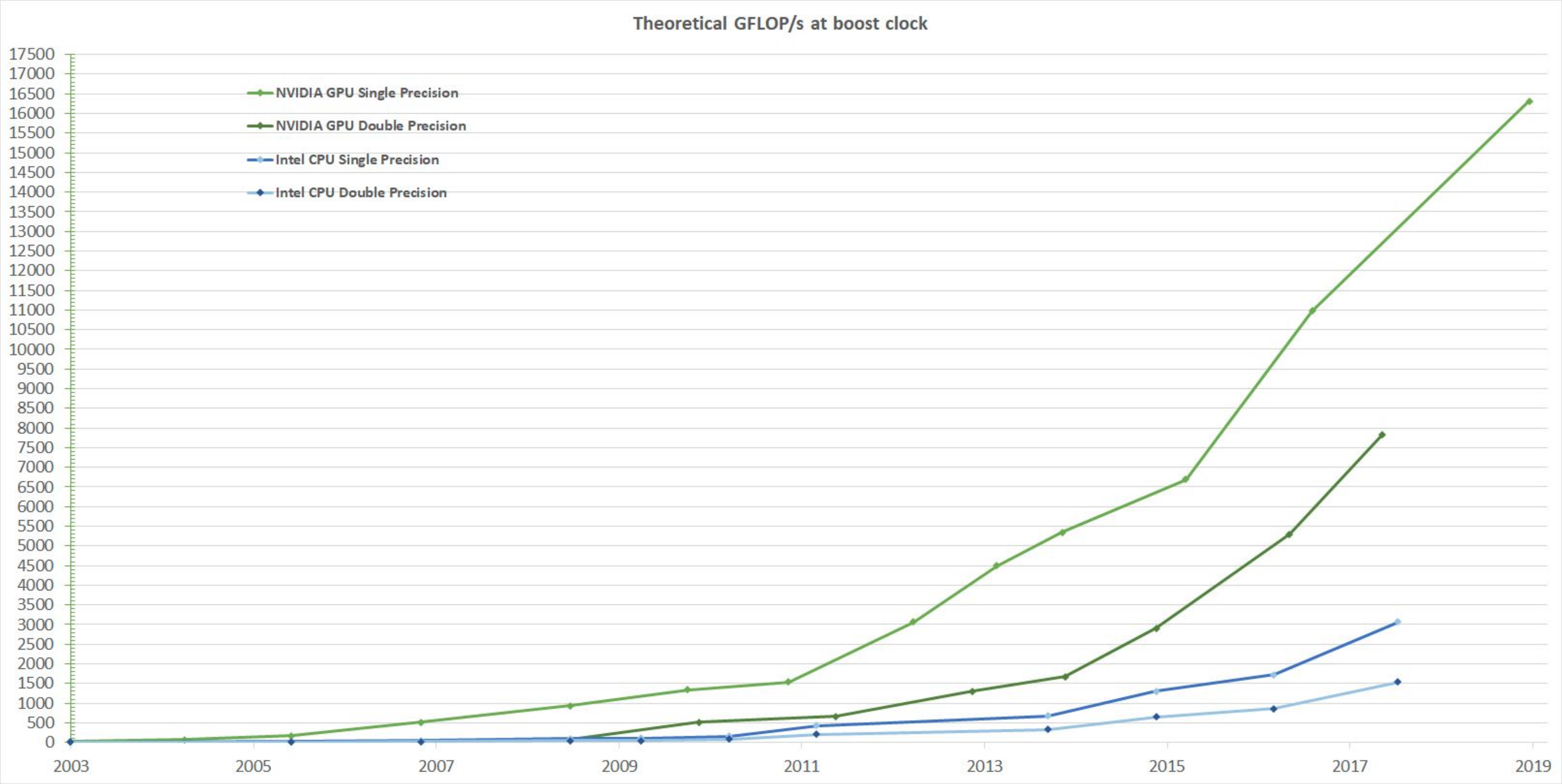
ITERATIONS * (~N/k) * [per-point gradient cost]

- + Cost of communicating to **accumulator**
- + Cost of distributing data (**points**) and parameters (**w**)

Alternative strategies

1. Gradient descent → Stochastic gradient descent (SGD)
    a. Iterate on small batches instead of the whole dataset
    b. In expectation, does the right thing (estimating gradient)
    c. **Probably what you should be doing anyway!**
    d. **But… requires a lot of communication!**
        i. **Every iteration / minibatch uses different data**
        ii. **weight vector is always changing**

2. **Use a different kind of computer**



https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Why GPUs?

- Current best practice is to use GPUs to train deep networks
- **WHY? What about GPUs makes this work?**
- Can other processes be similarly accelerated by using GPUs?
- Speedups from parallelism usually come from constraints…
    - *What can't we accelerate with GPUs?*

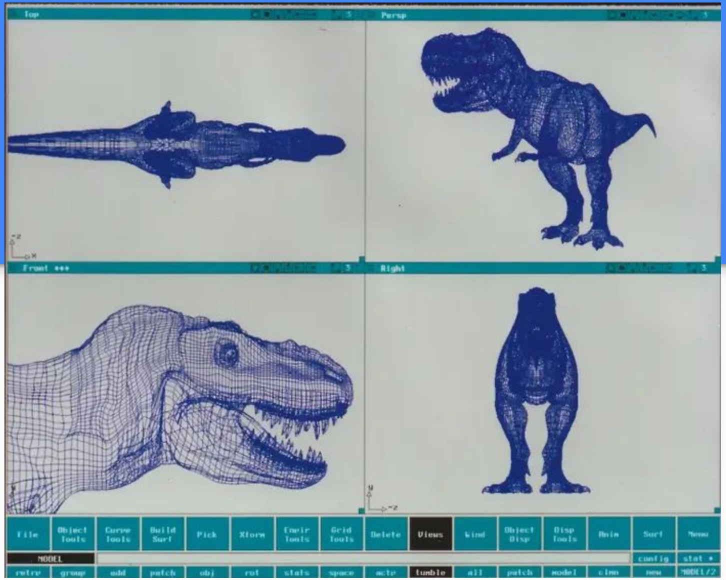
GPUs and the rendering pipeline

3D rendering

- Inputs:
  - 3D meshes
  - Textures, surface properties
  - Light sources
  - Camera position

- Outputs:
  - 2D array of pixels (rendered scene)

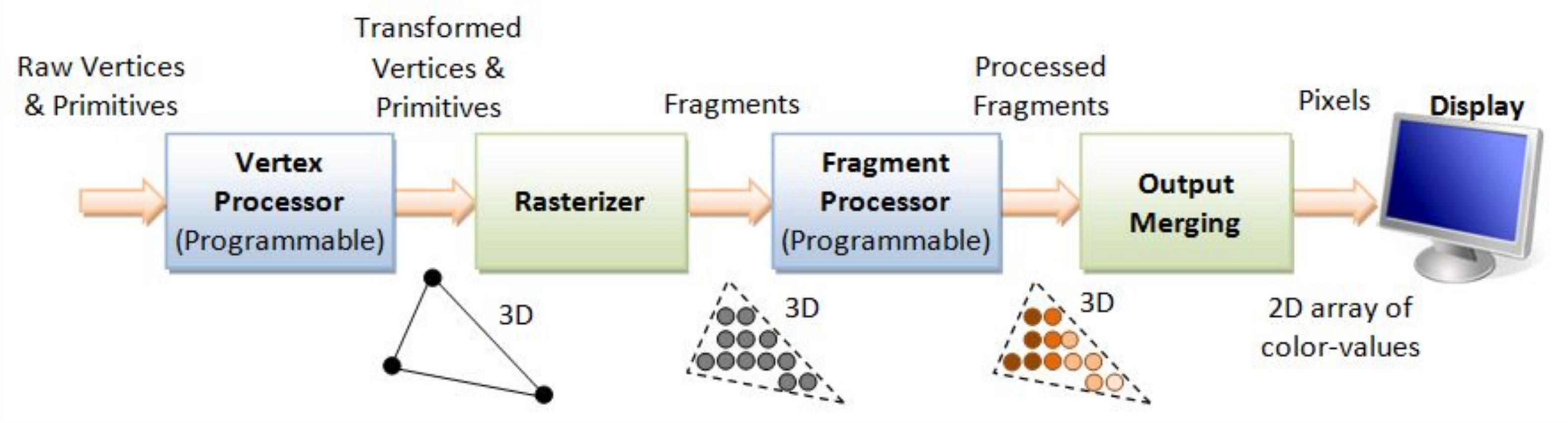
- Video games have real-time constraints



Computational challenges:

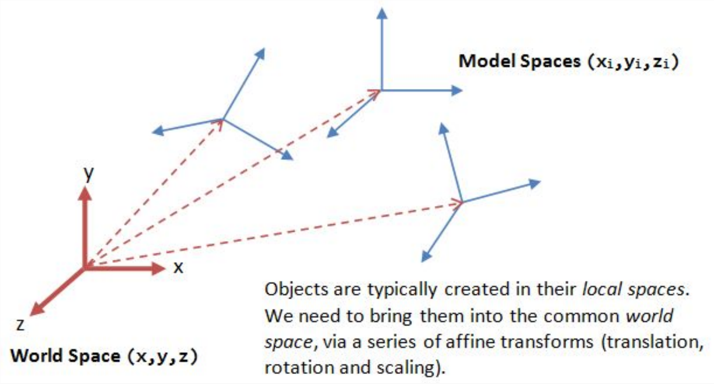
- Scene complexity (# surfaces)
- Output resolution (# pixels)

https://www.businessinsider.com/how-cgi-works-in-jurassic-park-2014-7



Graphics rendering pipeline

https://personal.ntu.edu.sg/ehchua/programming/opengl/CG_BasicsTheory.html



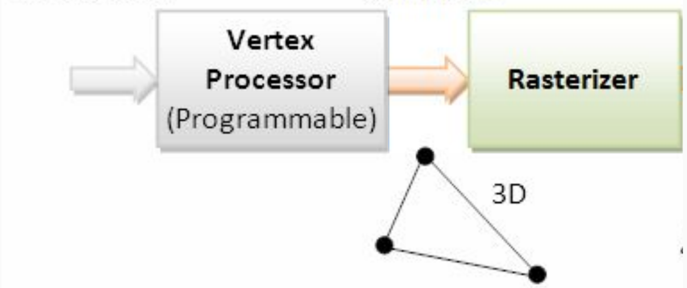
1. Vertex processing

- Coordinate transformations for each model
- Camera transformation
- Camera lens / field of view / etc

Mostly linear or affine transformations
(Rotation / translation / scaling)

Outputs:
All vertices mapped to camera coordinates (x, y, z)

https://personal.ntu.edu.sg/ehchua/programming/opengl/CG_BasicsTheory.html



2. Rasterizer

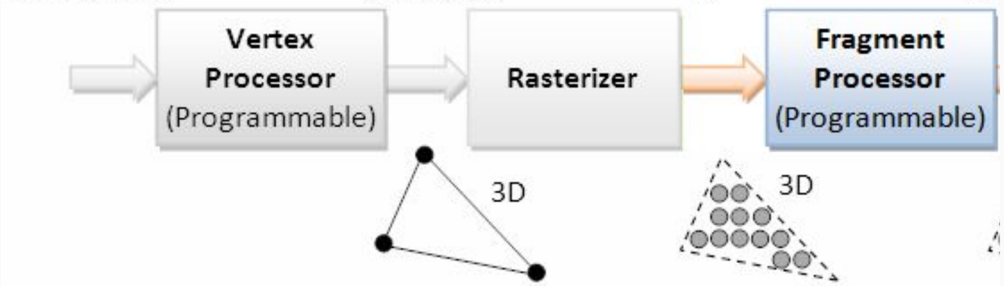
- Scans the scene for data to render at each pixel coordinate (x, y)
- Object vertices don't necessarily line up to pixel coordinates ⇒ meshes are interpolated

Outputs:
"Pixels" (or fragments) containing data to render at each pixel

Notes:

Vertices → Fragments is not generally 1-to-1
This step is not programmable

https://personal.ntu.edu.sg/ehchua/programming/opengl/CG_BasicsTheory.html



3. Fragment processing (aka “pixel shading”)

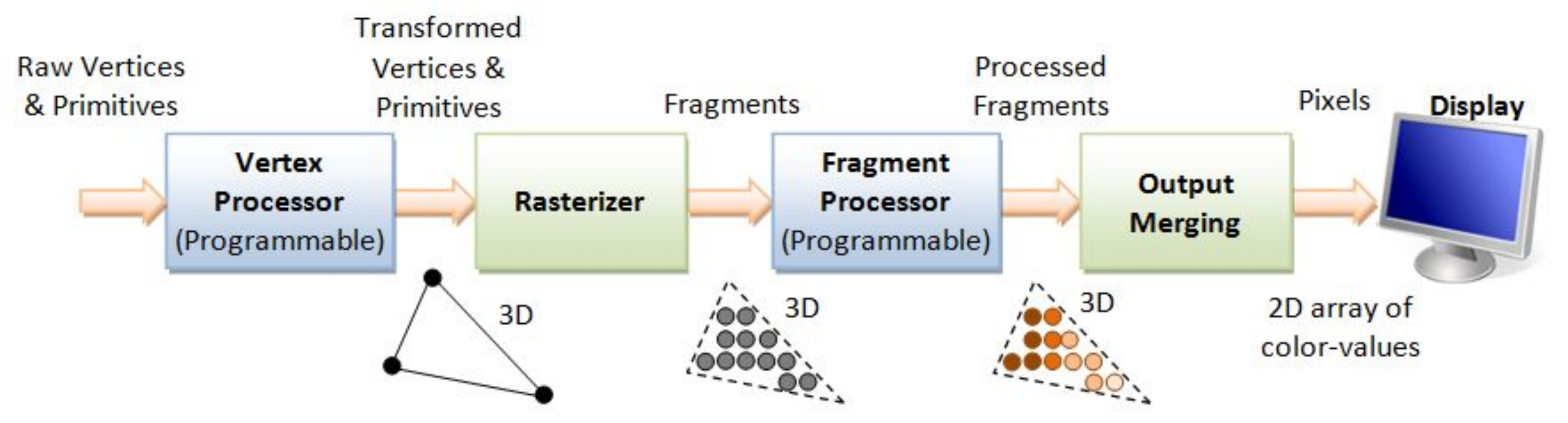
- Texture mapping, lighting
- Other visual effects (e.g. blur, masking, etc)

Outputs:

Color values for each fragment

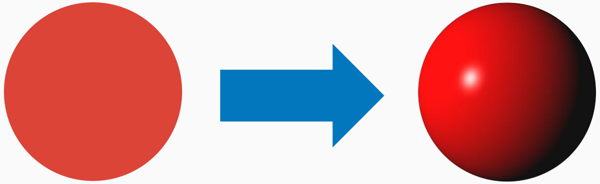
May include occluded objects (discarded later)

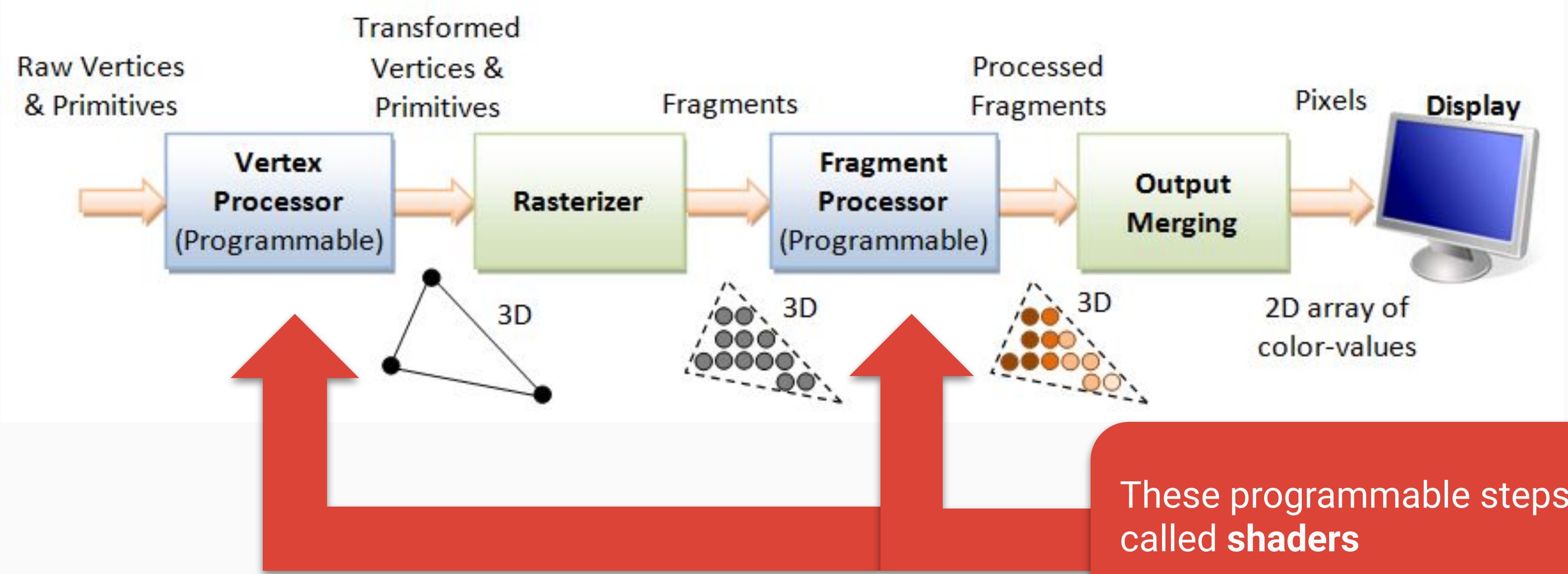
https://personal.ntu.edu.sg/ehchua/programming/opengl/CG_BasicsTheory.html



Graphics rendering pipeline

https://personal.ntu.edu.sg/ehchua/programming/opengl/CG_BasicsTheory.html





These programmable steps are called **shaders**

Graphics rendering pipeline

https://personal.ntu.edu.sg/ehchua/programming/opengl/CG_BasicsTheory.html

Parallelism in graphics

- Linear transformations can be applied **independently** to each **vertex**
- Texturing and lighting are also **independent** across **fragments**
- Specialized hardware can parallelize to meet real-time constraints
- To be cost-effective, each vertex or pixel processor needs to be **simple**

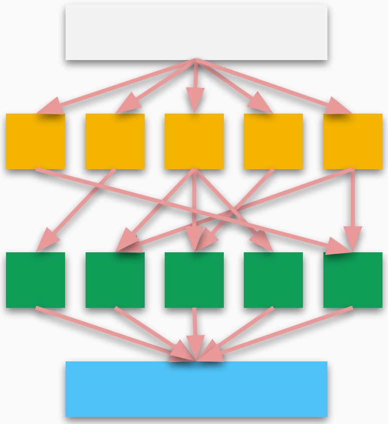
Shaders

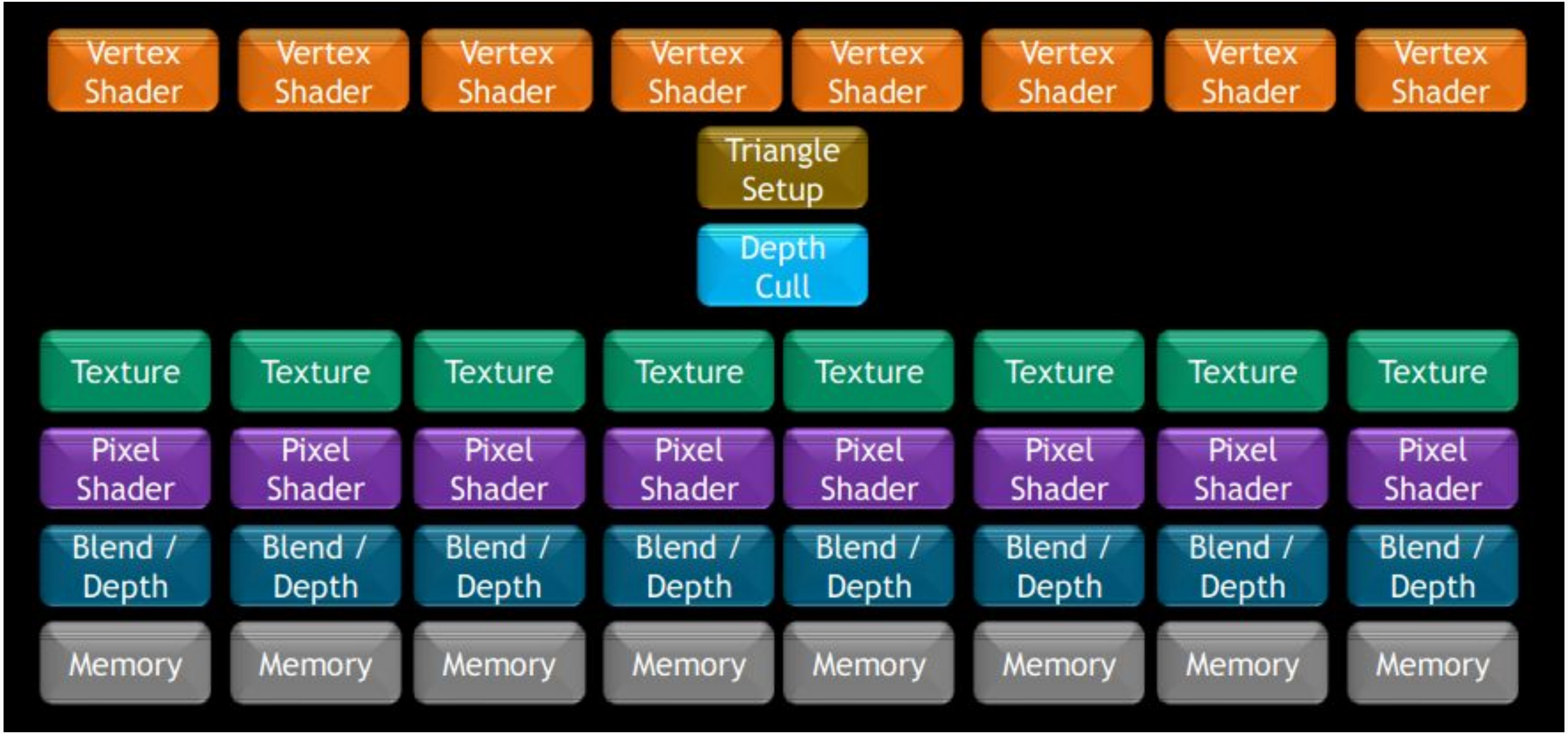
- "**Shaders**" are short programs that are applied independently to each vertex or fragment.

  Examples:
  - Apply rotations to all vertices in a mesh
  - Set color value for a fragment by indexing a texture map

- Sounds a bit like a **mapper**, right?

- Shader code tends to have simple control flow





Specialized hardware (ca. 2003) https://www.nvidia.com/content/nvision2008/tech_presentations/Technology_Keynotes/NVISION08-Tech_Keynote-GPU.pdf

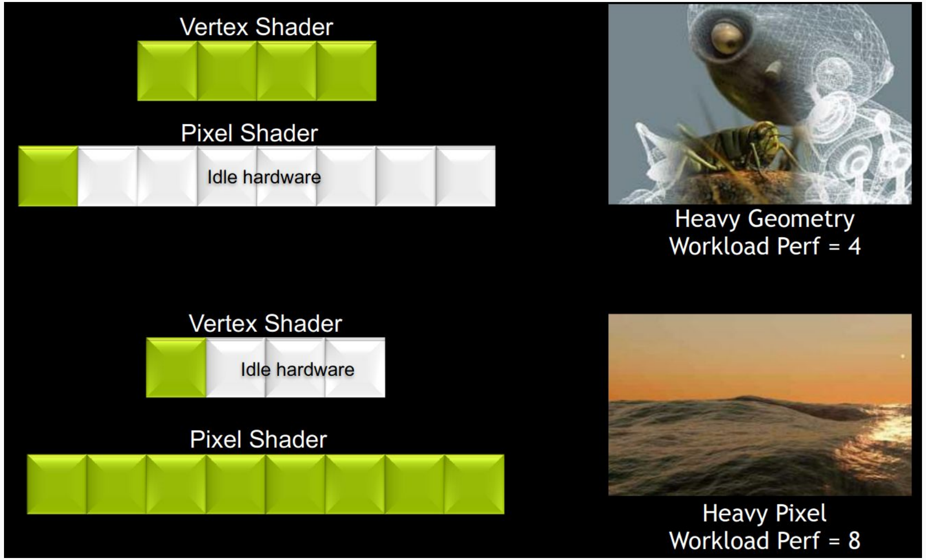
Specialized shader units

- Older GPUs had separate processors for **vertex shading** and **pixel shading**

- This works well when the load is balanced, but real scenes rarely are!
    - **# Vertices** ≠ **# Fragments**

- Unbalanced load means idle processors!

- Remember **key skew**?



https://www.nvidia.com/content/nvision2008/tech_presentations/Technology_Keynotes/NVISION08-Tech_Keynote-GPU.pdf

Summary

Part 1: Graphics processing

- GPUs were designed to optimize for low-latency, highly parallel operations
- Shader programs are simple, computed in parallel, and combined
- Specialized shader units can suffer from imbalance similar to key-skew...
- Can we do something more general?

... come back for part 2!