Computer Graphics:
4-Modern Graphics

Prof. Dr. Charles A. Wüthrich,
Fakultät Medien, Medieninformatik
Bauhaus-Universität Weimar
caw AT medien.uni-weimar.de

Introduction

- Themes of the this lesson will be:
  - modern graphics hardware architectures
  - modern graphics hardware programming, I.e. shading languages
- It will be far from complete, but hopefully it will give you an idea

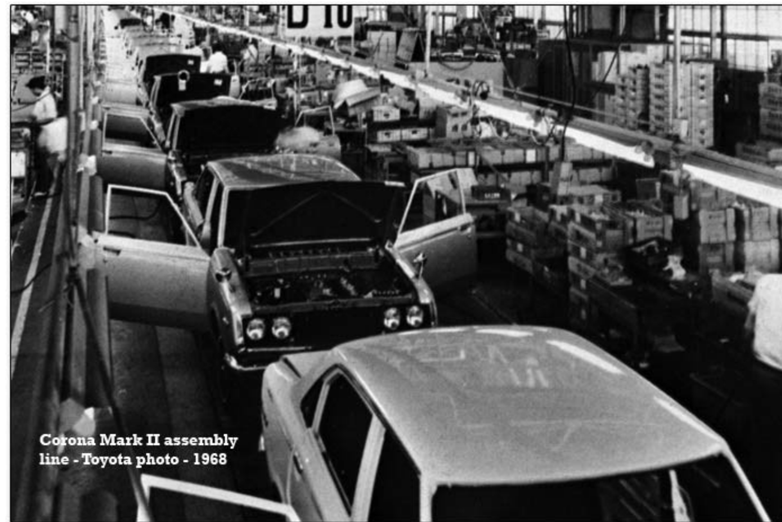
Graphics hardware

- The Graphics Application pipeline

• Supplies geometric data: – points, – polygons, – Curves • Converts into triangles	• Apply transformations • Shading • Clipping	• Fill pixel by pixel surviving triangles • Uses interpolation on vertex data

From the gfx pipeline to hardware

- This pipeline can be seen as a production line (assembly line):
  - Polygons are fed in and processed in stages



Copyright © 1968
Toyota Motor Corporation

From the gfx pipeline to hardware

- While one polygon gets e.g. transformed for viewing another one gets rasterized
- Like in pipeline processors
- Once processed data is handed elsewhere, something else is done with the data
- This is what pipeline processors do

Copyright © 1968
Toyota Motor Corporation

From the gfx pipeline to hardware

- Now the trick is to make processing so that balance load is even, and this is most difficult:
  - Triangles might be of different size, so that their processing might take different time and unbalance the pipeline
  - Or they might be too many (small triangles), so that they are too many to process in the required time and not use optimally the bandwidth
- The result is that the pipeline has to be optimized as much as possible:
  - too many polygons implies slow geometry processing
  - too big triangles mean also a problem because the single ones render slowly (rasterizer -> scan convertion slow)

From the gfx pipeline to hardware

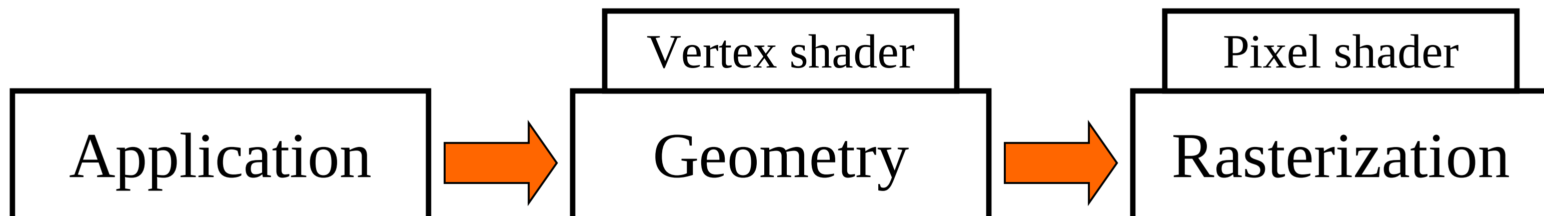
- To solve this, one can use parallelism: instead of one only unit one uses many in parallel to perform longer tasks like geometry processing or the scanline algorithm
- Between the stages FIFO queues are used
  - to facilitate the filling of the various stages and
  - to prevent backwards stalling in the pipeline (Stau)

From the gfx pipeline to hardware

- During history, graphics cards evolved from the end of the pipeline upwards
  - Early 1990s: Visibility and rasterization was in hardware
  - 1999: transformations and lighting moved to the graphics cards
    - Ambient, diffuse and Phong shading, alpha channel blending and fog moved to HW
  - Interfaces however were different for each vendor, which was not good for developers

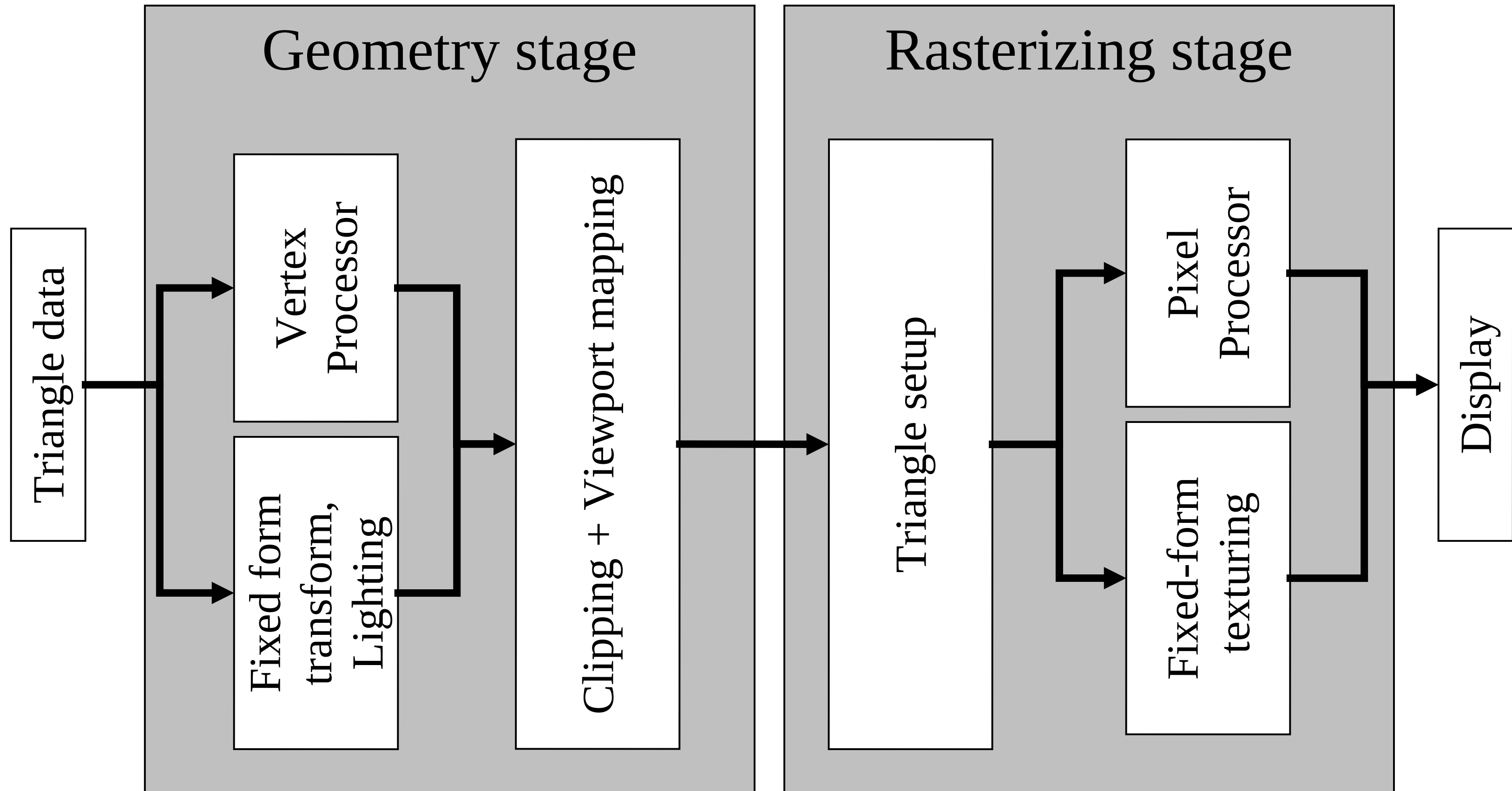
Programmable processors

- To solve this, instead of fixed function processors, programmable geometry and fragment processors were introduced in parallel to the fixed function stages
- These generic processors were baptized
  - Vertex shaders
  - Fragment shaders (pixel shader in DirectX)



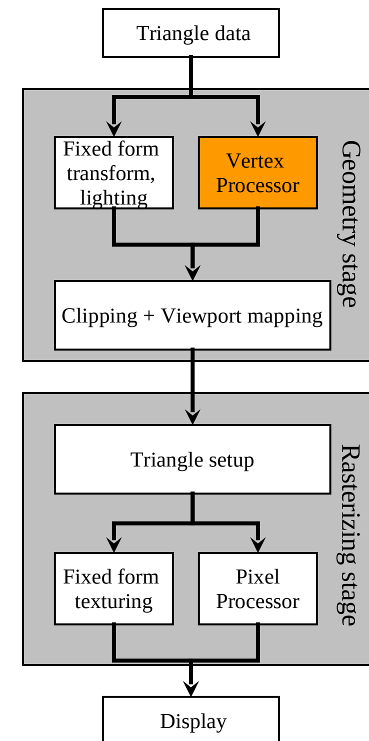
Programmable processors

- The pipeline looked then like this:



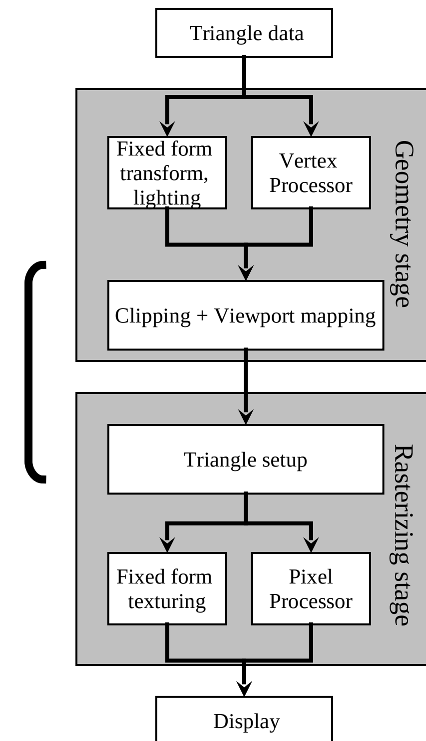
Vertex processor

- Vertex processor processes one vertex of a triangle at a time:
  - Recieves coordinate values
  - Receives a set of constants for this vertex (surface properties)
- Has a number of operators to manipulate these coordinate values:
  - Dot product, subtract, normalize...)
- Outputs a new vertex
  (which might have a new format):
  - New XYZ
  - Normals, colors, texture coordinates....
- Can therefore deform geometry in world or view space



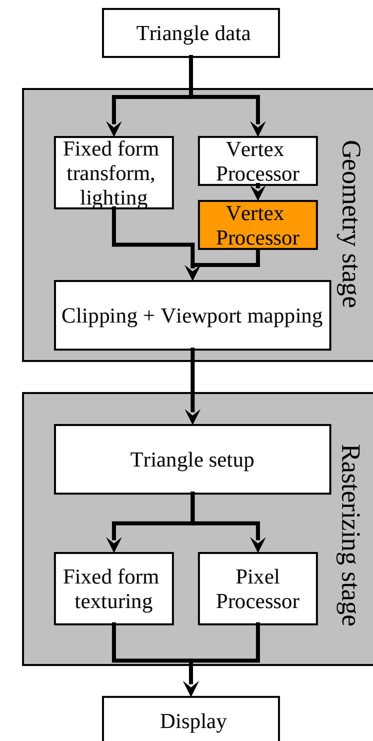
Vertex processor

- Vertex processor and fixed transform processor output data for the vertices of the triangle
- Then culling and clipping is done and the triangle is passed to the raster processor
- In a first stage the triangle is setup for interpolating across its surface



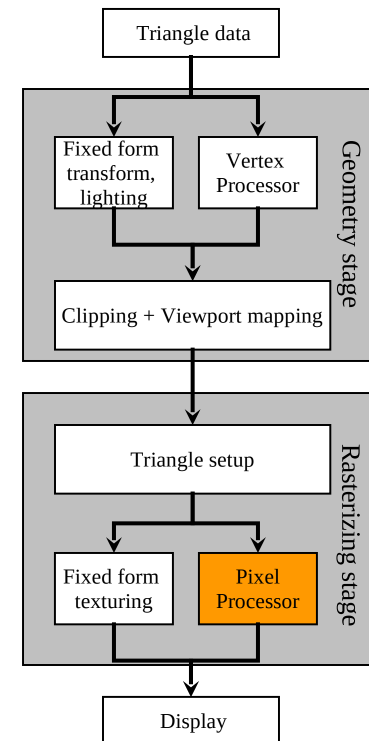
Geometry processor

- Since 2006, an additional geometry shader has been added to the pipeline right after the vertex shader
- Use is optional
- Its inputs are single objects (triangles, lines, points)
- Such primitives can be extended
- The geometry shader processes the primitive and outputs other primitives (points, polylines or triangle strips)
- This allows to modify



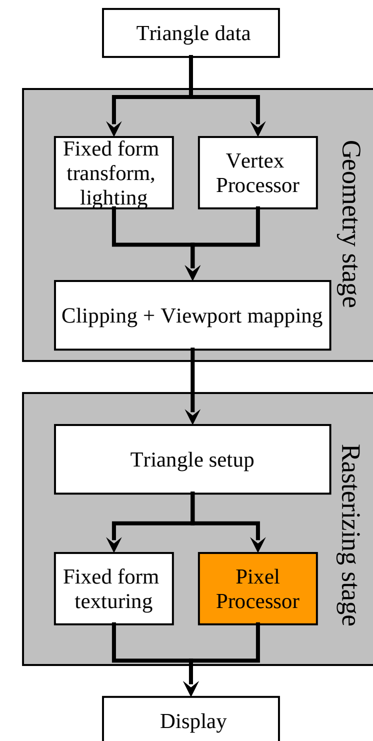
Fragment (pixel) processor

- Whenever a pixel of a triangle is drawn, the fragment processor is capable of accessing
  - The interpolated values from vertex data
  - Stored constant data (originally textures): such texture access can also be used to access indirectly other textures (*dependent texture read*)
- Per pixel the fragment processor manipulates these data
- It optionally then writes these data to the Z-buffer, computing first
  - RGB values
  - Optionally Z-values
- Pixel processor operations can only be done by the graphics hardware (too slow on the CPU)



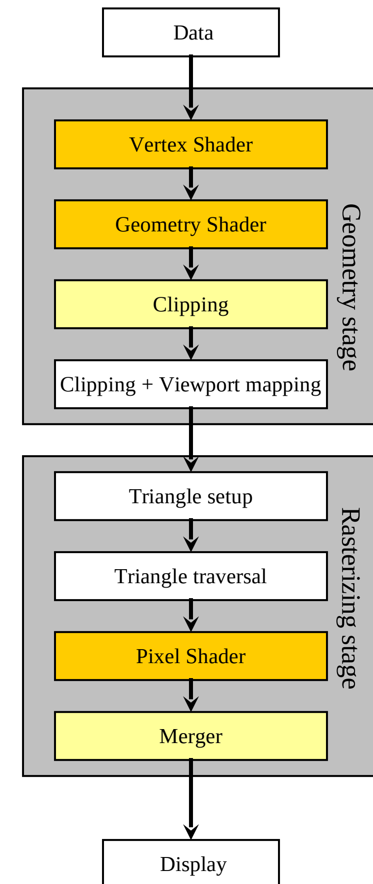
Fragment (pixel) processor

- Originally, the fragment processor was able only to perform integer (fixed point) operations
- Nowadays, they can do full floating point minimizing discretization effects
- Moreover, it can render to multiple buffers (*multiple targets*)
- These targets can be reused as textures again fur further computations
- Originally, conditional access was used for doing multiple pass renderings: remember the illumination equation?
- Conditional texture access and floating point precision allows parallel execution of more complex functions



Evolving to generic processors

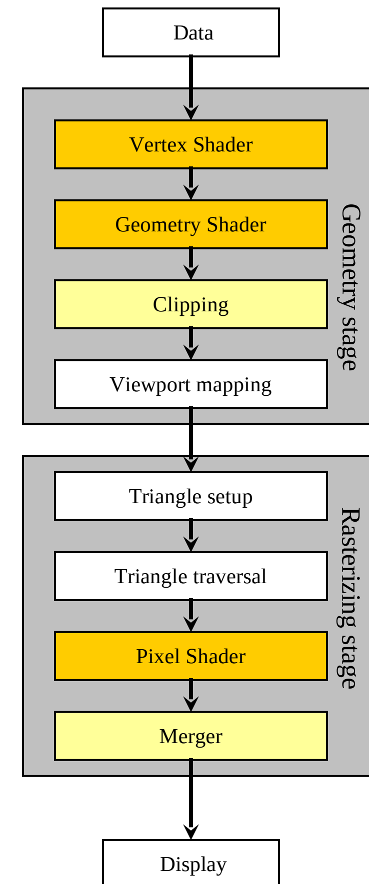
- The instruction sets of vertex and pixel shaders (and the new geometry shader) have become more and more complex
  - Branching (IF)
  - Dynamic IF
  - FOR loops
  - Multiple function calls also possible
- Programs written for such units are called *shaders*
- Since the addition of complex functionality in shaders, slowly the fixed parts of the processing can be replaced in the more flexible vertex and pixel units
- Moreover, the Geometry and Fragment stage programming instructions have converged in time to become very similar
- With removal of the specialized units, the pipeline becomes like this:



Evolving to generic processors

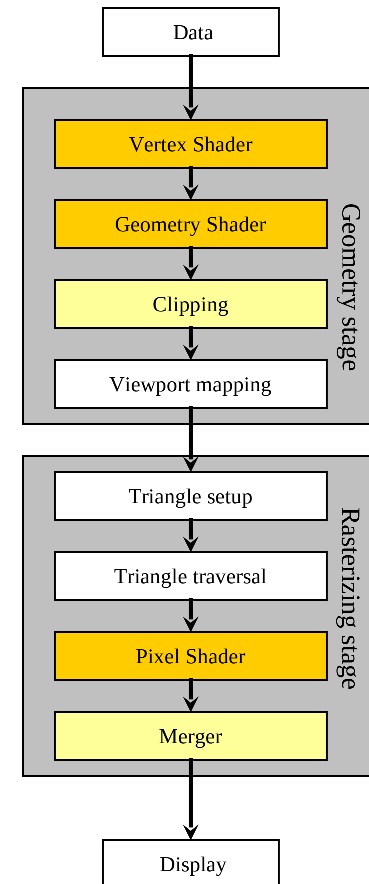
- In the last stage, called the merger, the depth and color of the individual fragments are combined with the frame buffer.
- Performs:
  - stencil buffer
  - Z-buffer
  - Color blending (transparency).

- Note that in the picture we have noted
  - Orange: fully programmable stages
  - Yellow: configurable stages
  - White: fixed stages



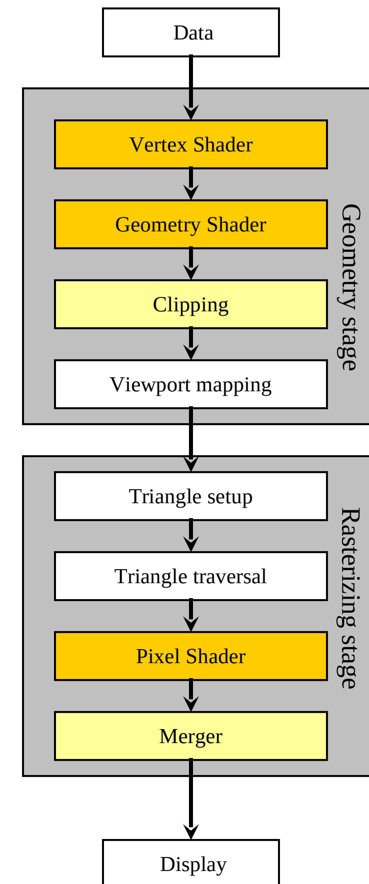
Evolving to generic processors

- In parallel, C-like *shading languages* have been developed
  - Main representatives: HLSL, CG, GLSL, CUDA, OpenCL
  - These languages are translated by the compiler into a unified Intermediate Language (IL)
  - IL is some sort of machine independent assembly language
- The IL is then translated into machine code by the drivers of the graphics card



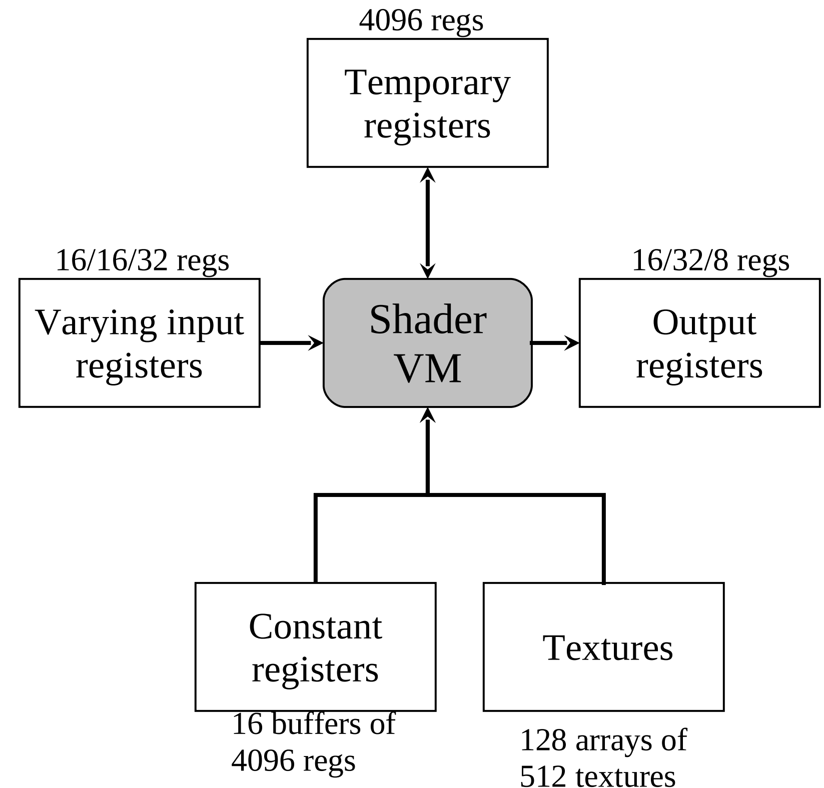
Virtual machine

- The virtual machine is a processor with various registers and data sources and can be programmed with a set of instructions
- The processor has 4 way SIMD capabilities
  - Each register contains four independent variables, usually floating points
  - Recently they can contain also integers
  - Typically they are positions (homogeneous), normals, rows of a matrix, colors or texture coords.
  - The card would also support aggregate data structures such as array, matrices and structs
- To facilitate working with vectors one can do:
  - swizzling (reordering, replicating of vector elements)
  - Masking: using only some of the vector components



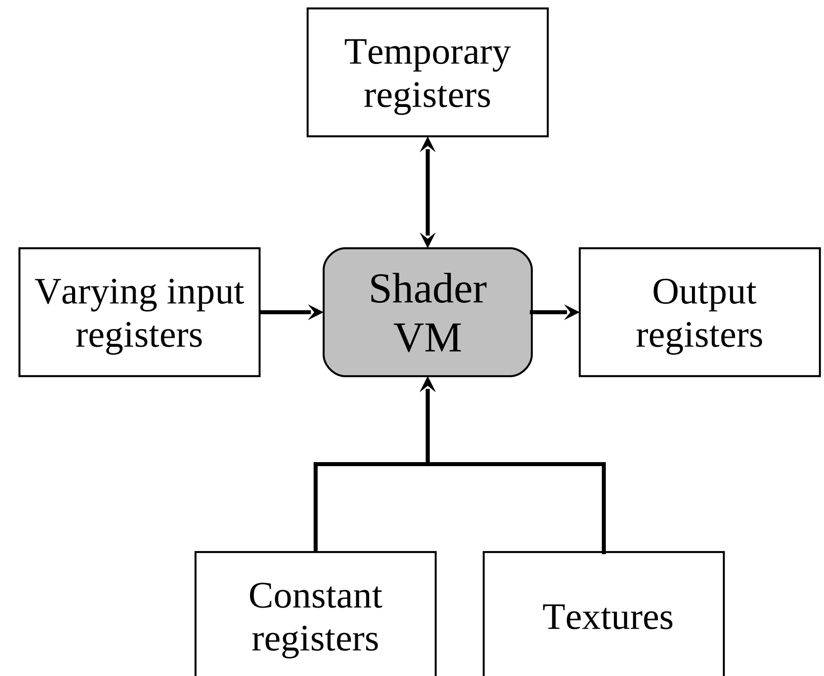
Virtual machine

- What happens when the graphics subsystem is executed?
  - A *draw call* invokes the graphics API to draw a group of primitives, causing the pipeline to execute
- Each programmable shader accepts two types of input:
  - *Uniform inputs*: values stay the same through the whole draw call - they reside in read only registers or read-only buffers
    - A texture is a uniform input: originally colors, nowadays however it is a large array of data
  - *Varying inputs*, different for each vertex or pixel processed by the shader: they are much less in number
- Aside from that there are also general purpose temporary registers, used as scratch
- All registers can be addressed as arrays



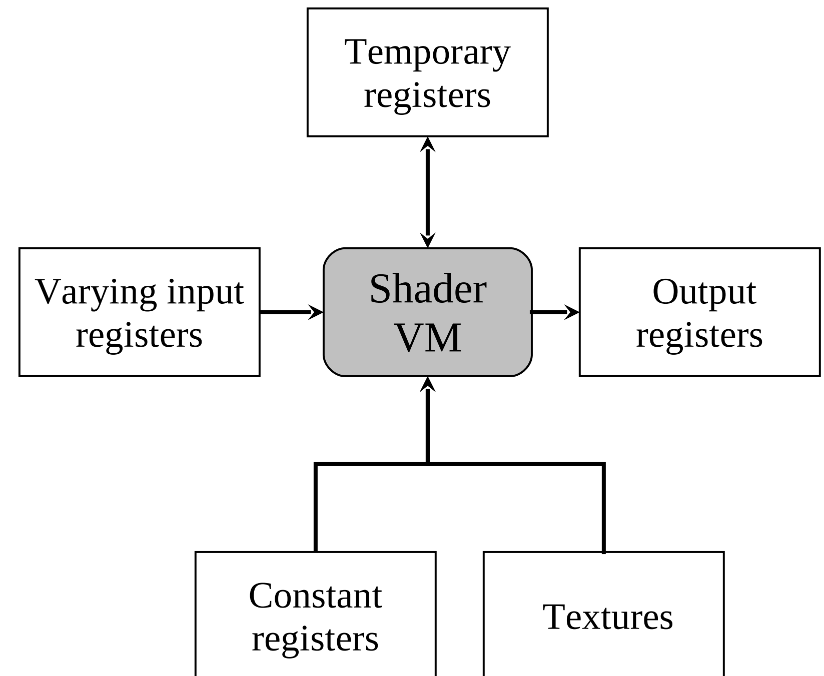
Virtual machine

- Which operations can I do faster on graphics HW?
  - Scalar and vector mult, additions
  - Any combination of the above (e.g. dot product)
- Texturing operations are also fast, but since the results are big textures, it might take time to retrieve the results
- Slower are other operations, such as computing the arctangent



Virtual machine

- There are two ways of performing program flow control:
  - Static flow control: based on the values of uniform input
    - I.e. flow stays same during draw call
  - Dynamic flow control: bases on the values of varying inputs
    - More costy, because it might change the flow and do things a shader is not necessarily optimized for
- The shader program can be compiled, and then loaded into the GPU

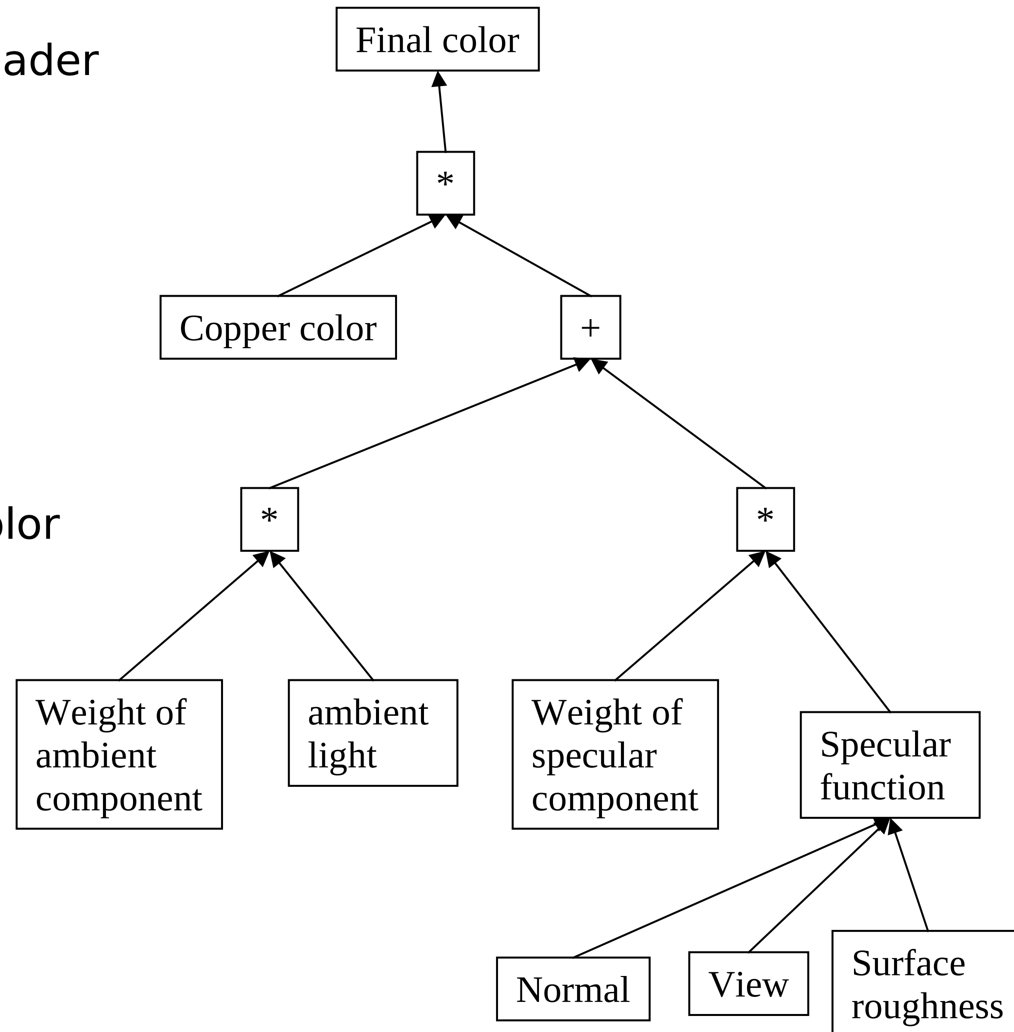


Shade trees

- Here a simple example of a shader program:

```
float ka=0.5, ks=0.5;
float roughness=0.1;
float intensity;
color copper=(0.8,0.3,0.1);
intensity=ka*ambient()+
   *specular(normal,view,roughness);
final_color=intensity*copper
```

- This corresponds to a tree of “effects” to obtain a certain color



Shading languages

- HLSL, CG, GLSL are evolutions from shade trees
- Syntax similar to C
- Newer:
  - proprietary: CUDA
  - Open: OpenCL

- Flow simiilar to C, BUT data can be large arrays

End

+++ Ende - The end - Finis - Fin - Fine +++ Ende - The end - Finis - Fin - Fine +++