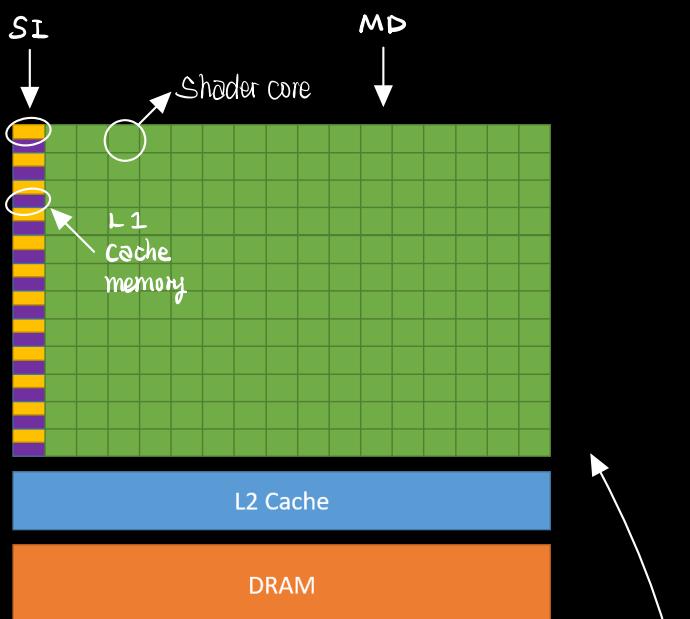
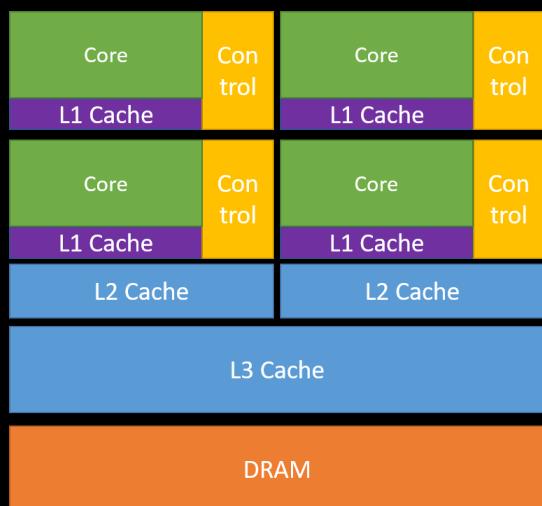


# GPU Architecture

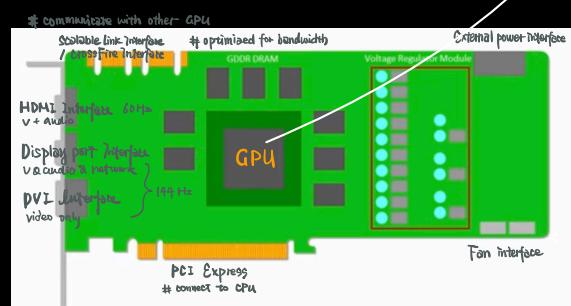
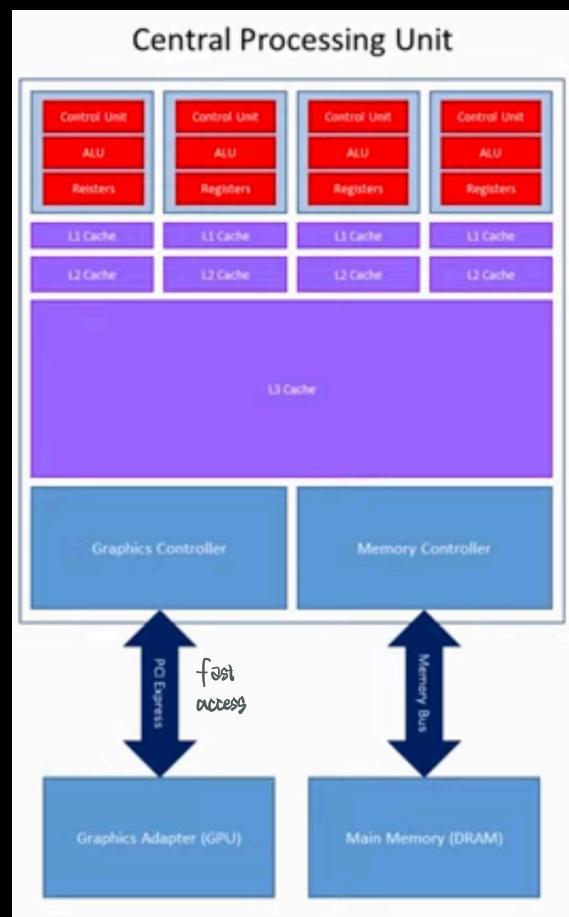


## CPU

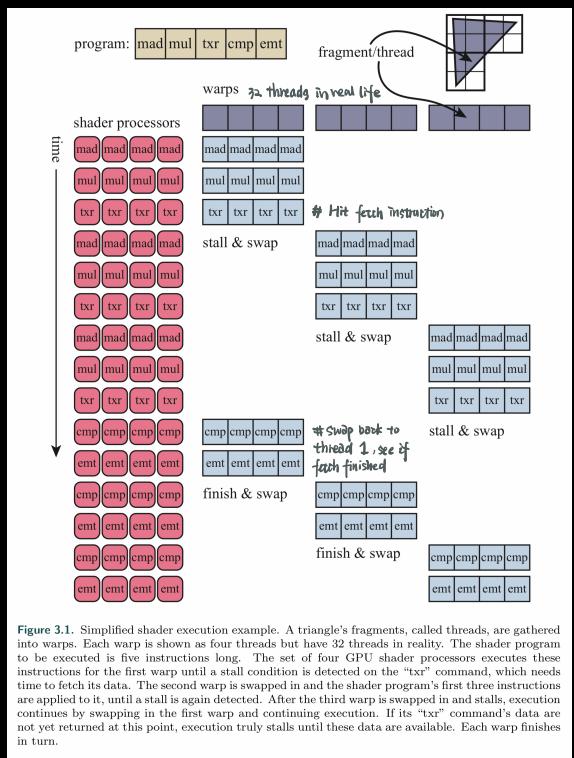
100s of thread in parallel  
high single-thread performance

## GPU

1000s of thread in parallel  
lower single-thread performance  
to achieve better throughput

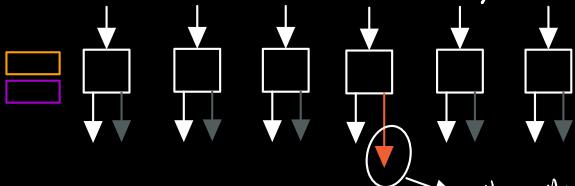


# SIMD - Make GPU as busy as possible Latency Hiding Mechanics < warping - out >



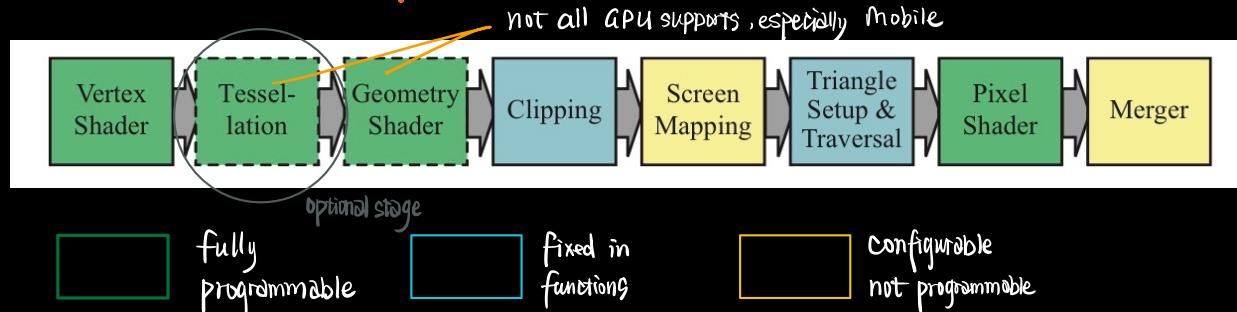
## Latency Factors

- × fewer threads
- × shader program structure
  - amount of registers used in each thread  
more registers needed for each thread
  - less thread can be generated → parallel
- × occupancy - amount of warps that can resident < working >
- × frequency of memory fetching → affects more latency hidden
- × dynamic branching → branch divergence

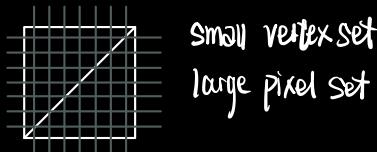


other threads  
Stall & wait for  
this, they cannot  
work separately  
because of SI

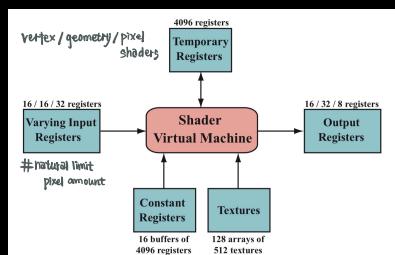
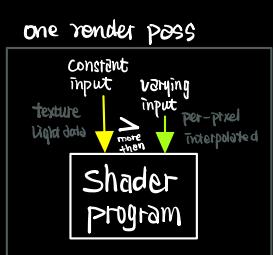
## GPU & Render Pipeline



Common Shader Core - All shaders use same Instruction set architecture



GPU can balance the resources



constant

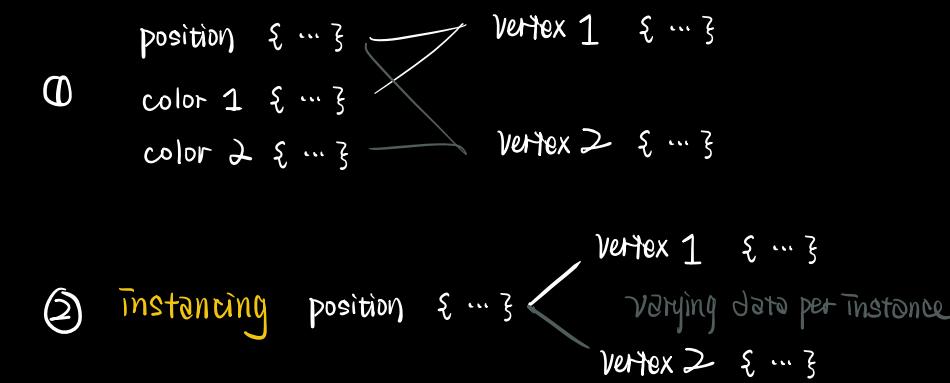
writing

static flow control

dynamic flow control

powerful low performance

## Data assembler

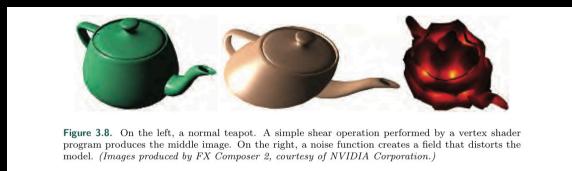


model space  
Vertex shader  
clip space

Possible uses

- Vertex Blending → Animation
- Silhouette Rendering

- Object generation, by creating a mesh only once and having it be deformed by the vertex shader.
- Animating character's bodies and faces using skinning and morphing techniques.
- Procedural deformations, such as the movement of flags, cloth, or water
- Particle creation, by sending degenerate (no area) meshes down the pipeline and having these be given an area as needed.
- Lens distortion, heat haze, water ripples, page curls, and other effects, by using the entire framebuffer's contents as a texture on a screen-aligned mesh undergoing procedural deformation



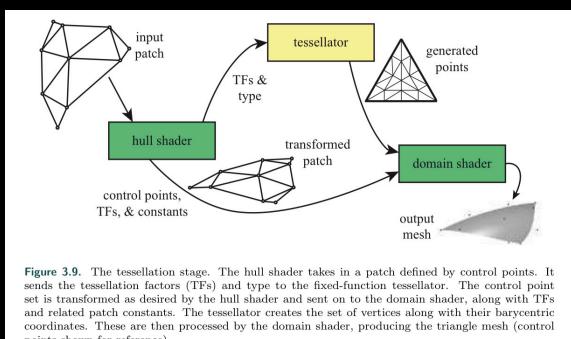
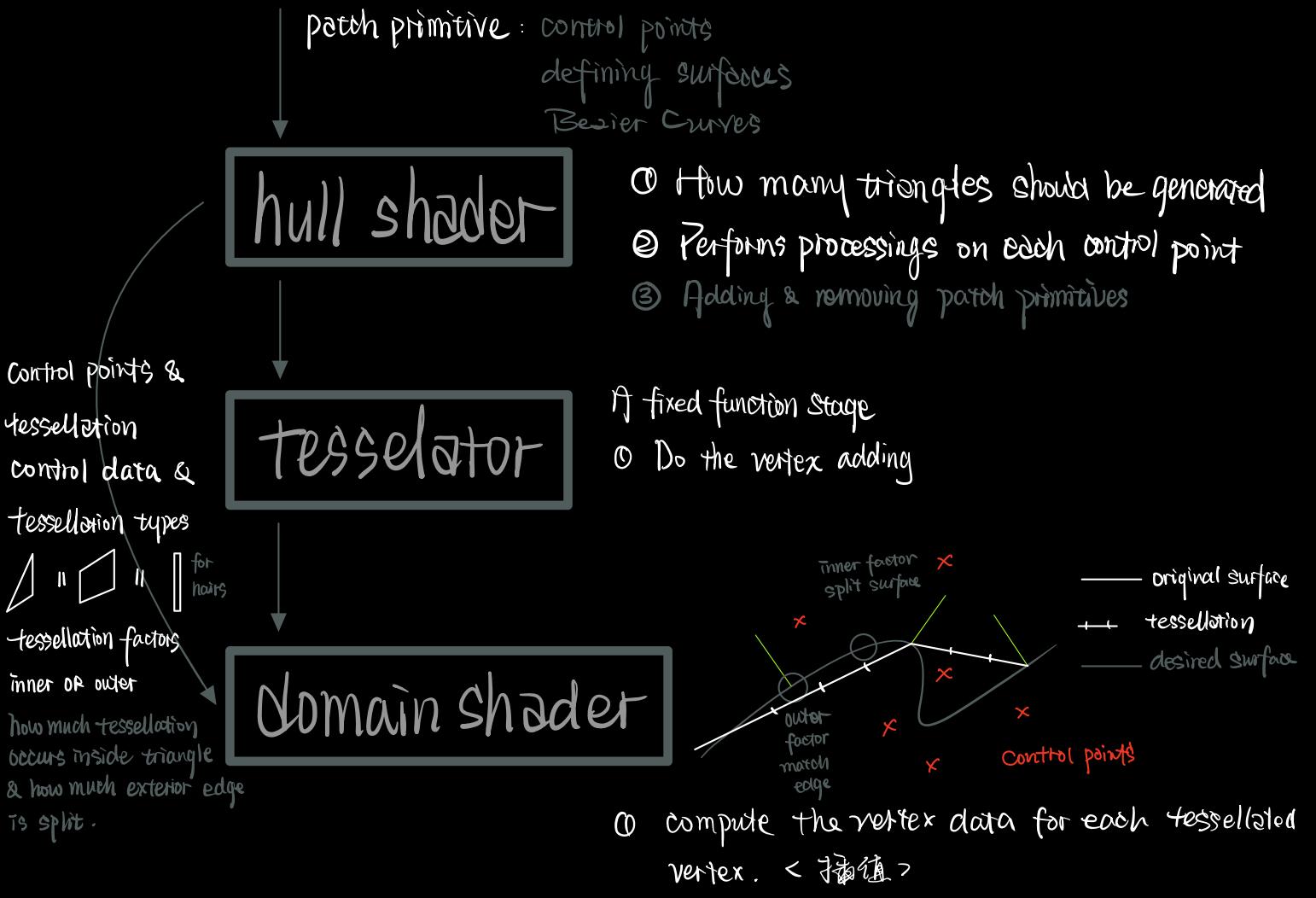
## Tessellation Stage Subdivision → allows rendering curved surfaces

hull shader sets the type & number of subdivisions LOD

tesselator actually do the subdivision

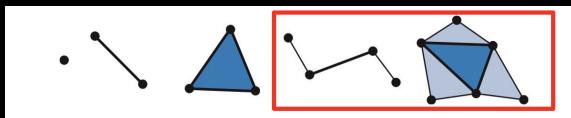
domain shader called for each newly generated vertex

sets the type of primitive to generate/ways to space vertices  
can also do per-vertex calculation as a vs.



# Geometry Shader turns primitives into other primitives

vertices representing extended primitives



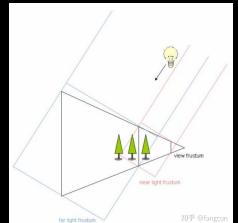
## result

- ① Modify incoming data
- ② Making limited num of copies

expand 1 face to a cubemap  
create cascaded shadow map



Figure 3.13. Some uses of the geometry shader (GS). On the left, metaball isosurface tessellation is performed on the fly using the GS. In the middle, fractal subdivision of line segments is done using the GS and stream out, and billboards are generated by the GS for display of the lightning. On the right, cloth simulation is performed by using the vertex and geometry shader with stream out. (Images from NVIDIA SDK 10 [1300] samples, courtesy of NVIDIA Corporation.)

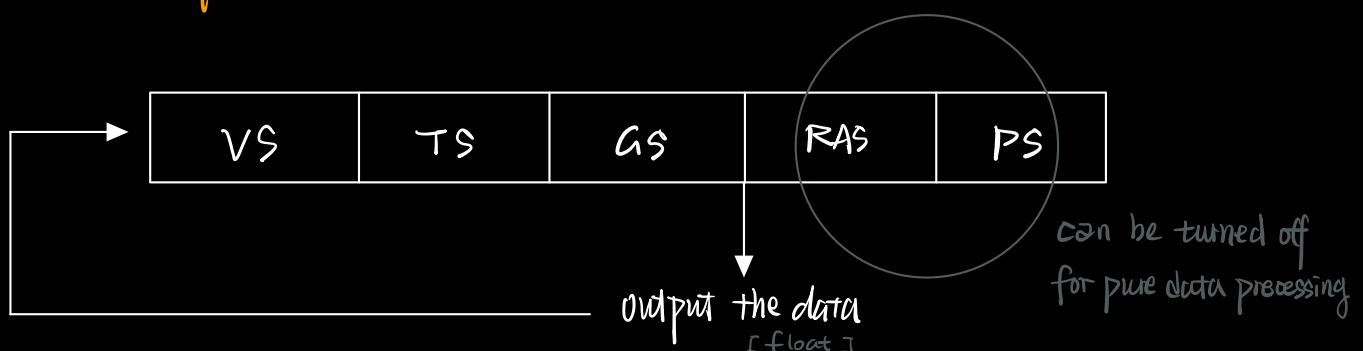


## Drawbacks

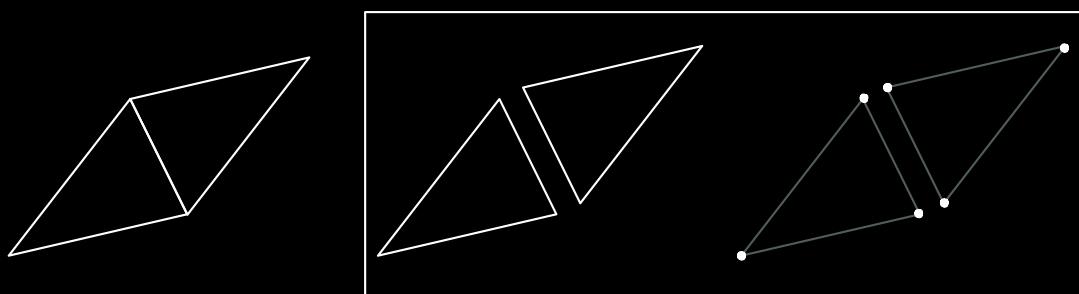
The geometry shader is guaranteed to output results from primitives in the same order that they are input. This affects performance, because if several shader cores run in parallel, results must be saved and ordered. This and other factors work against the geometry shader being used to replicate or create a large amount of geometry in a single call [175, 530].

After a draw call is issued, there are only three places in the pipeline where work can be created on the GPU: rasterization, the tessellation stage, and the geometry shader. Of these, the geometry shader's behavior is the least predictable when considering resources and memory needed, since it is fully programmable. In practice the geometry shader usually sees little use, as it does not map well to the GPU's strengths. On some mobile devices it is implemented in software, so its use is actively discouraged there [69].

Stream Output works on primitives, not vertices



- △ useful in water & other particle simulation
- △ input order is kept



△ vertex sharing won't be kept. Therefore we usually send vertices directly instead of meshes

Stream output stage

## Clipping

## Screen Mapping

Rasterization Interpolation type & timing are configurable

VS outputs  
 interpolation  
 results  
 +  
 Screen position  
 ...

## Pixel Shader

computes and outputs a fragment's color

can also possibly produce an opacity value and optionally modify its z-depth  
discard an incoming fragment :

multiple MRT  
 buffer render target  
 4~8 T  
 can be various sizes

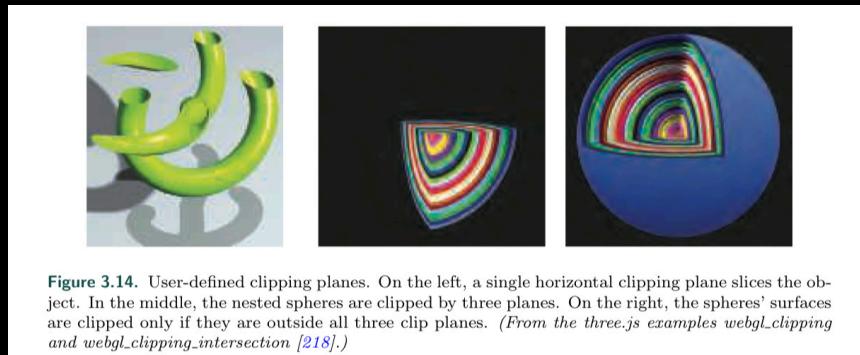


Figure 3.14. User-defined clipping planes. On the left, a single horizontal clipping plane slices the object. In the middle, the nested spheres are clipped by three planes. On the right, the spheres' surfaces are clipped only if they are outside all three clip planes. (From the three.js examples `webgl_clipping` and `webgl_clipping_intersection` [218].)

↓      ↓      ↓      GENERALLY  
 Some x,y position  
 Some GPU requires same  
 depth bit.

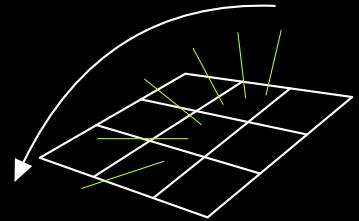
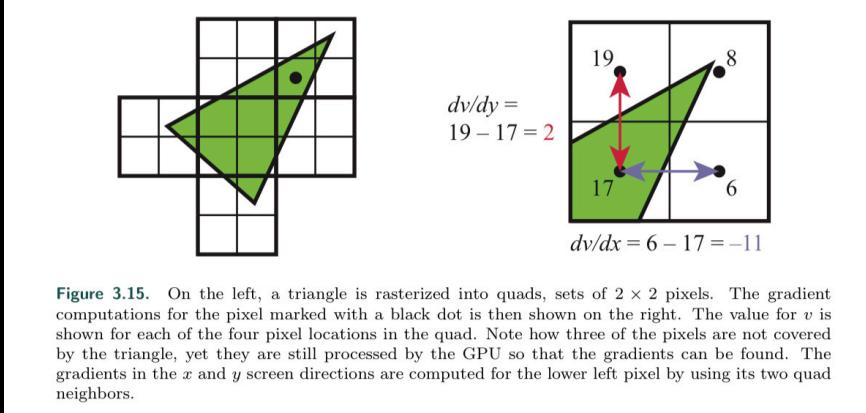
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
pixel	object	world space
image identifier	distance	

deferred shading visibility & shading are done in later passes

- ① PASS record material & position
- ② PASS effectively apply illumination

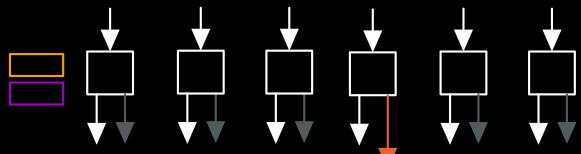
△ Pixel Shading cannot access adjacent neighbor pixels, but can achieve this by using render results as input to later passes  
<image-space effects>

△ There are also exceptions where pixel shader can get info about neighbors INDIRECTLY → gradient value:



get the gradient change of the interpolated values on  $x/y$  axeses.

When the pixel shader requests a gradient value, the difference between adjacent fragments is returned



A unified core has this capability to access neighboring data—kept in different threads on the same warp—and so can compute gradients for use in the pixel shader

△ gradient fetch does not support dynamic flow control : broken parallelism

## ShaderStorageBufferObject

unordered access view  
a data storage accessible to all shader programs < after DX11.3>

△ implements atomic through hardware

→ avoids race condition

→ May cause stall = atomic  $\Rightarrow$  waiting

## RasterizerOrderView

→ Similar to UAV, but order enhanced

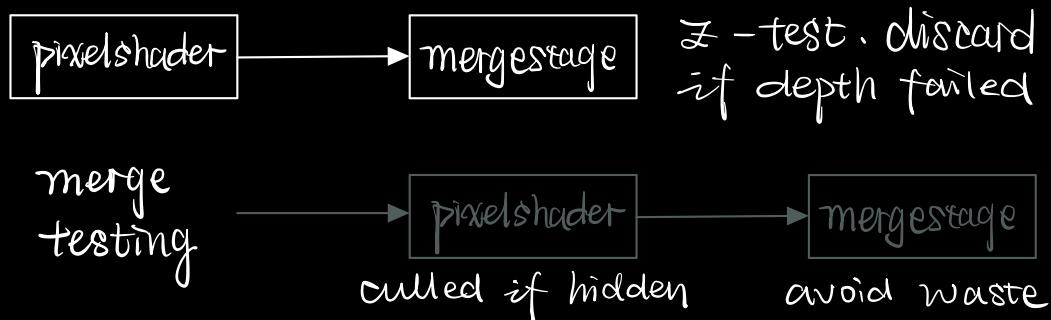
→ Can replace merge stage functionality

△ VS can merge to another pixel, but if that pixel is not rendered yet, stall happens

# Merging Stage

merging stage is where the depths and colors of the individual fragments (from pixel shader) are combined with the framebuffer  
△ Stencil buffer & z buffer & color blending

## Early Z-test



- △ early-z is turned-off if z-value discard or modification is detected in pixel shader making pipeline less efficient

## Highly Configurable color blending

The most common are combinations of multiplication, addition, and subtraction involving the color and alpha values, but other operations are possible, such as minimum and maximum, as well as bitwise logic operations.

# Compute Shader

independent from render stages

- △ more exposed to warp & thread information — each invocation gets a thread index accessible
- △ thread group → 1 ~ 1024 thread in  $bx\,by$

These thread groups are specified by x-, y-, and z-coordinates, mostly for simplicity of use in shader code. Each thread group has a small amount of memory that is shared among threads. In DirectX 11, this amounts to 32 kB. Compute shaders are executed by thread group, so that all threads in the group are guaranteed to run concurrently

- △ Compute shaders can access data generated on GPU, avoids the delay sending data between GPU & CPU
  - post-processing is benefited by such feature < modifying rendered image >
- △ computing things like average illumination can be done by communicating through accessing the shared memory. This is 1x faster than on Pixel Shader.

## Other Uses

particle system

mesh processing — facial animation

culling

image filtering

improving depth precision — shadows

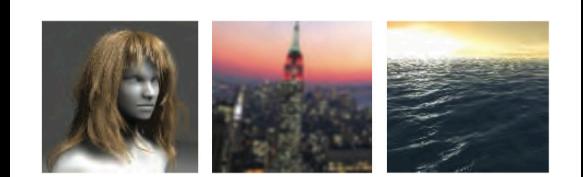


Figure 3.16. Compute shader examples. On the left, a compute shader is used to simulate hair affected by wind, with the hair itself rendered using the tessellation stage. In the middle, a compute shader performs a rapid blur operation. On the right, ocean waves are simulated. (Images from NVIDIA SDK 11 [1301] samples, courtesy of NVIDIA Corporation.)