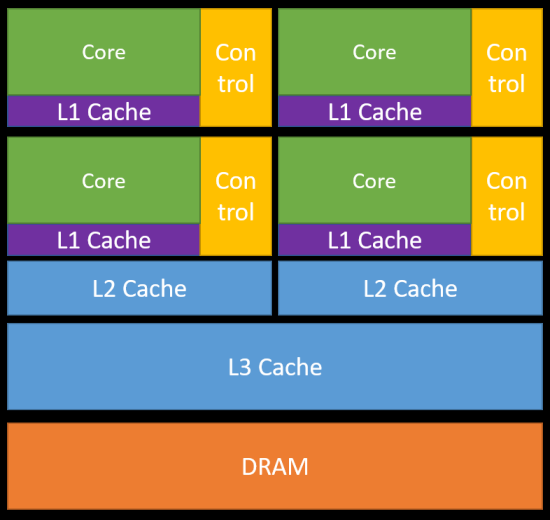
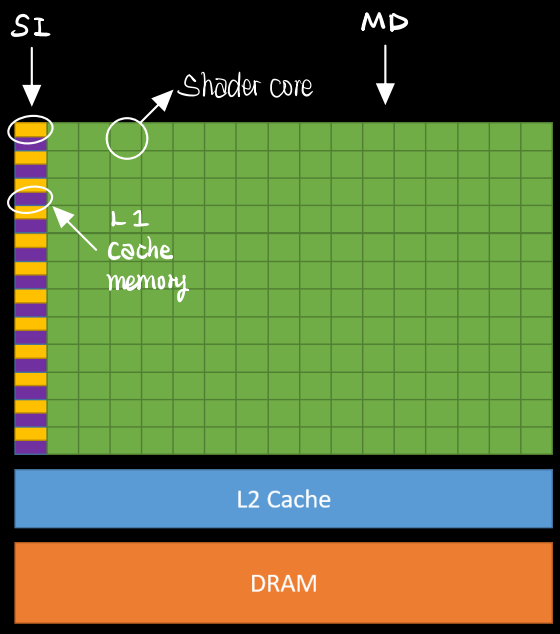
GPU Architecture





CPU

10s of thread in parallel

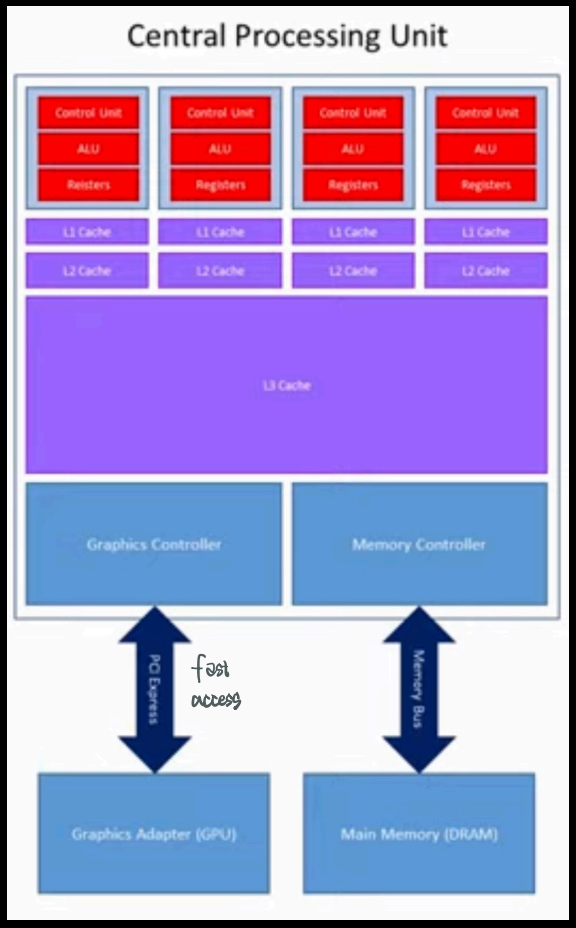
high single-thread performance

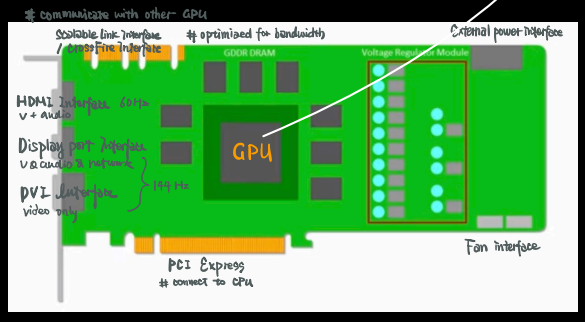
GPU

10000s of thread in parallel

lower single-thread performance

to achieve better throughput





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

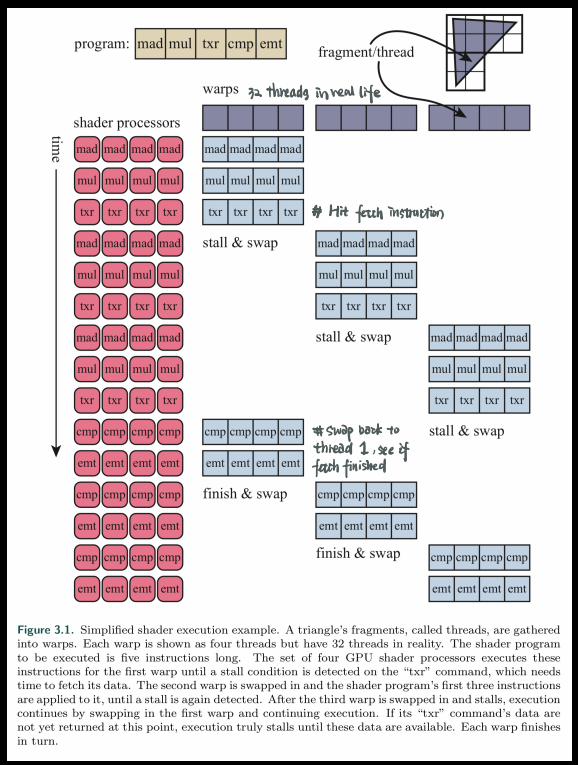
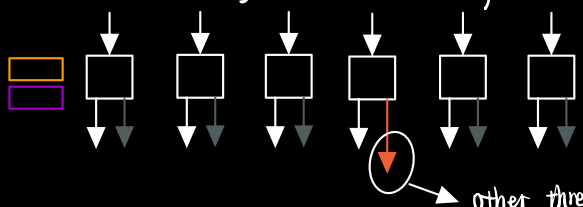


Figure 3.1. Simplified shader execution example. A triangle's fragments, called threads, are gathered into warps. Each warp is shown as four threads but have 32 threads in reality. The shader program to be executed is five instructions long. The set of four GPU shader processors executes these instructions for the first warp until a stall condition is detected on the "txr" command, which needs time to fetch its data. The second warp is swapped in and the shader program's first three instructions are applied to it, until a stall is again detected. After the third warp is swapped in and stalls, execution continues by swapping in the first warp and continuing execution. If its "txr" command's data are not yet returned at this point, execution truly stalls until these data are available. Each warp finishes in turn.

Latency Factors

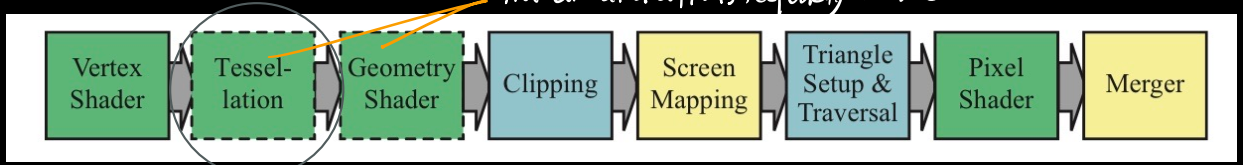
- × fewer threads
- × shader program structure
  - amount of registers used in each thread
    more registers needed for each thread
    less thread can be generated ↘ parallel
- × occupancy – amount of warps that can resident <workin>
- × frequency of memory fetching → affects more latency hidden
-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



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

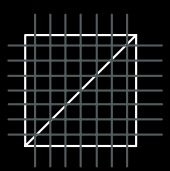
GPU & Render Pipeline

not all GPU supports, especially Mobile



fully programmable fixed in functions configurable not programmable

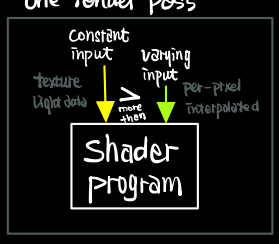
Common Shader Core – All shaders use same instruction set architecture

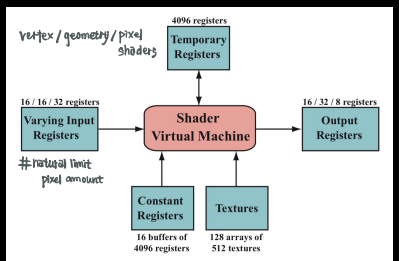


small vertex set
large pixel set

GPU can balance the resources

one render pass





constant → Static flow control

varying → dynamic flow control

powerful low performan

data assembler

① position { ... } → vertex 1 { ... }
color 1 { ... } → vertex 2 { ... }
color 2 { ... }

② Instancing position { ... } → vertex 1 { ... }, varying data per instance, vertex 2 { ... }

All within simple draw call

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

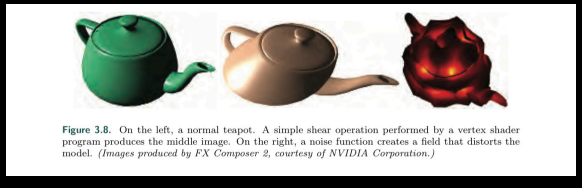


Figure 3.8. On the left, a normal teapot. A simple shear operation performed by a vertex shader program produces the middle image. On the right, a noise function creates a field that distorts the model. *(Images produced by FX Composer 2, courtesy of NVIDIA Corporation.)*

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 vertecies
can also do per-vertex calculation as a vs.

patch primitive : control points defining surfaces Bezier Curves

hull shader

① How many triangles should be generated
② Performs processings on each control point
③ Adding & removing patch primitives

Control points & tessellation control data & tessellation types

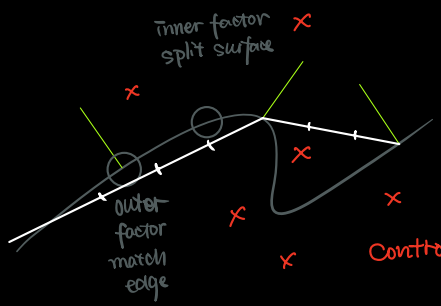
tessellation factors

inner or outer

how much tessellation occurs inside triangle & how much exterior edge is split.

tesselator

A fixed function stage
① Do the vertex adding



domain shader

① compute the vertex data for each tessellated vertex. < 插值 >

single object & associated vertecies

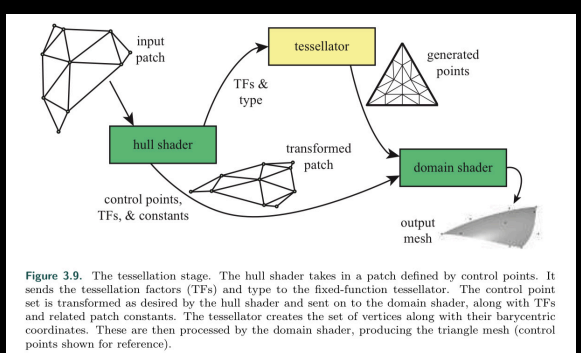


Figure 3.9. The tessellation stage. The hull shader takes in a patch defined by control points. It sends the tessellation factors (TFs) and type to the fixed-function tessellator. The control point set is transformed as desired by the hull shader and sent on to the domain shader, along with TFs and related patch constants. The tessellator creates the set of vertices along with their barycentric coordinates. These are then processed by the domain shader, producing the triangle mesh (control points shown for reference).

Geometry Shader turns primitives into other primitives

vertices representing extended primitives

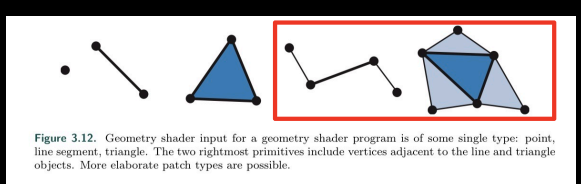


Figure 3.12. Geometry shader input for a geometry shader program is of some single type: point, line segment, triangle. The two rightmost primitives include vertices adjacent to the line and triangle objects. More elaborate patch types are possible.

result
① Modify incoming data
② Making limited num of copies
expand 1 face to a cube map
Create cascaded shadow maps

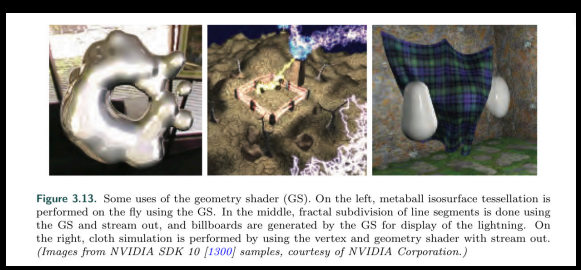
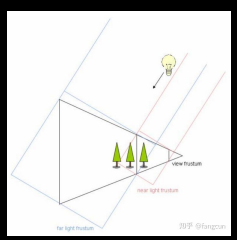


Figure 3.13. Some uses of the geometry shader (GS). On the left, metaball isosurface tessellation is performed on the fly using the GS. In the middle, fractal subdivision of line segments is done using the GS and stream out, and billboards are generated by the GS for display of the lightning. On the 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.)*

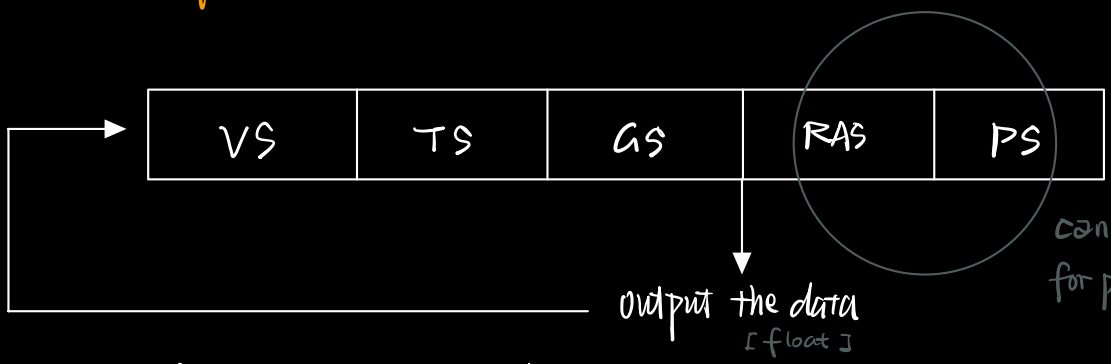


Drawbacks

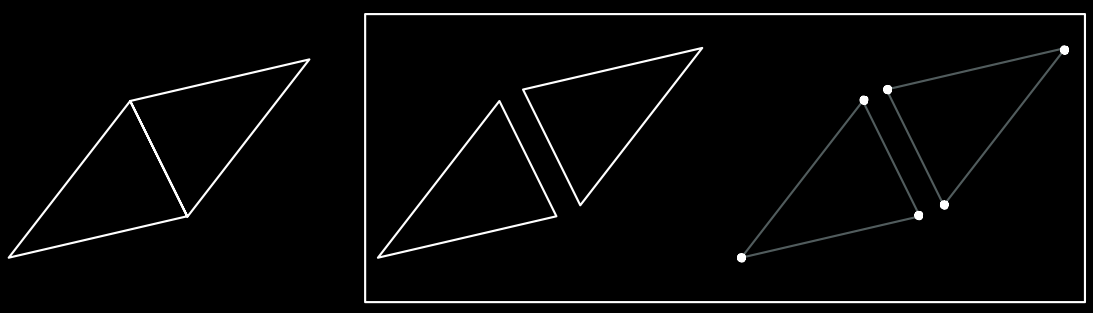
The geometry shader is guaranteed to output results from primitives in the same order that they are input. This affects performance, because if several shader cores run in parallel, results must be saved and ordered. This and other factors work against the geometry shader being used to replicate or create 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 vertecies



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



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

Stream output stage

Cliping

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 :

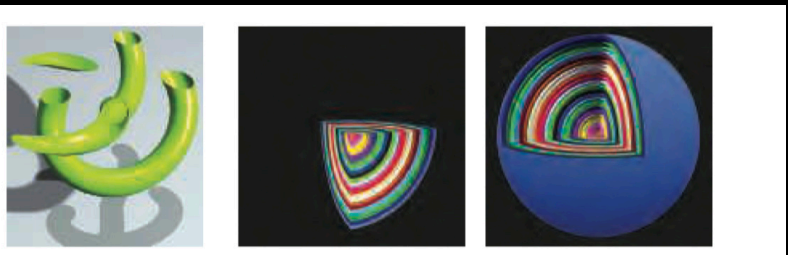


Figure 3.14. User-defined clipping planes. On the left, a single horizontal clipping plane slices the object. In the middle, the nested spheres are clipped by three planes. On the right, the spheres' surfaces are clipped only if they are outside all three clip planes. *(From the three.js examples webgl_clipping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

multiple MRT
buffer render target

4~8개

can be various sizes

GENERALLY
same x,y position

pixel

some GPU requires some depth bit.

pixel image

object identifier

world space 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 exeptions where pixel shader can get info about neighbors INDIRECTLY → gradient value :

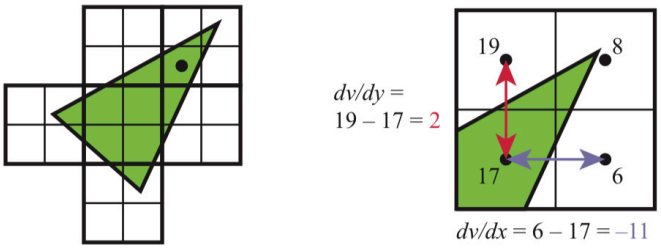
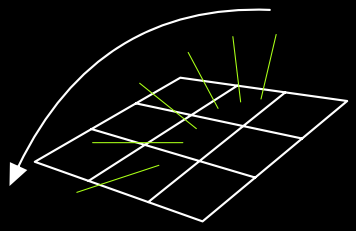
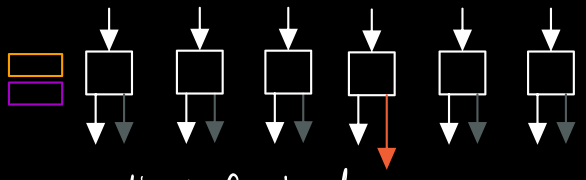


Figure 3.15. On the left, a triangle is rasterized into quads, sets of 2 × 2 pixels. The gradient computations for the pixel marked with a black dot is then shown on the right. The value for v is shown for each of the four pixel locations in the quad. Note how three of the pixels are not covered by the triangle, yet they are still processed by the GPU so that the gradients can be found. The gradients in the x and y screen directions are computed for the lower left pixel by using its two quad neighbors.



get the gradian 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 ⇒ 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

z-test, discard if depth failed

△ 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 accessibl

△ thread group → 1 ~ 1024 thread in DX 11

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 1× faster than on Pixel Shader.

Other Uses

- particle system
- mesh processing — facial animation
- culling
- image filtering
- improving depth precision — shadows

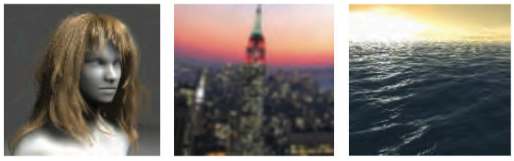


Figure 3.16. Compute shader examples. On the left, a compute shader is used to simulate hair affected by wind, with the hair itself rendered using the tessellation stage. In the middle, a compute shader performs a rapid blur operation. On the right, ocean waves are simulated. *(Images from NVIDIA SDK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