Games and Simulation

2021-2022

Fernando Birra
Rui Nóbrega

The Graphics Processing Unit (GPU)

Fernando Birra
Rui Nóbrega

Latency

- All processors face latency
- Accessing data takes some amount of time
- The further away the information is from the processor, the longer the wait
- Local registers are faster than main memory
- Different architectures deal with latency in different ways

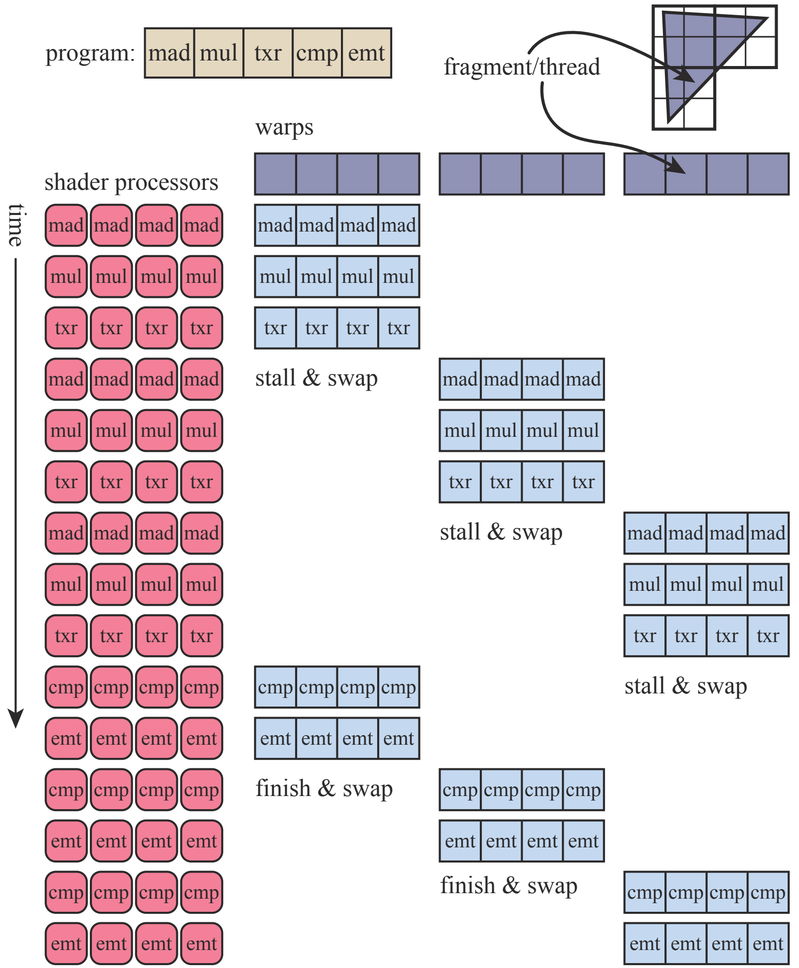
Latency

- CPUs have low core count and chip area can be occupied with large caches, circuits for speculative execution (branch prediction), instruction reordering, register renaming and cache pre-fetching.

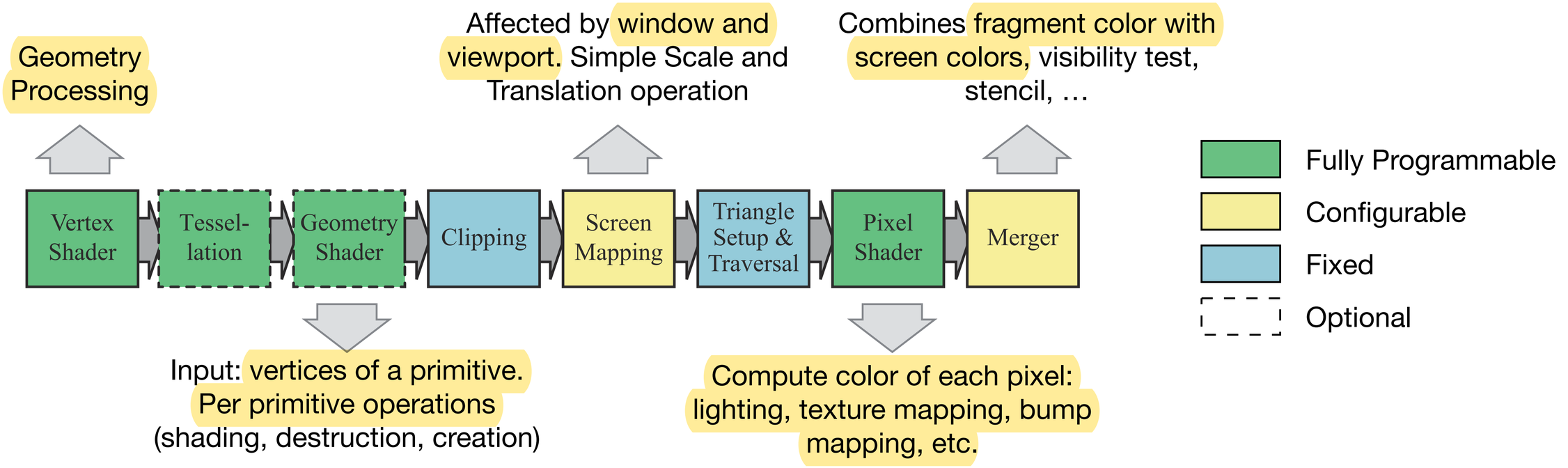
- GPUs have a very large number of processor cores (in the thousands) working together as a very wide stream processor - ordered sets of similar data are processed in turn. Vertices or pixels are handled in a massively parallel fashion, with near to no interdependency, in a setting that aims for throughput. The higher core count and less sophisticated control logic leads to higher latency for each shader core.

Data-Parallel Architectures

- Threads are grouped in warps (wavefronts)
- Each thread has input memory and registers
- Warps execute the same instruction in lock-step mode (SIMD)
- When a thread/warp stalls, it is swapped out. No data needs to be copied!
- Threads that have started are named “in flight” (resident)
- The number of resident threads (occupancy) depends on the number of registers used by the shader program
- High occupancy promotes low latency
- Thread divergence (branching) reduces efficiency



GPU Pipeline Overview



The Programmable Shader Stage

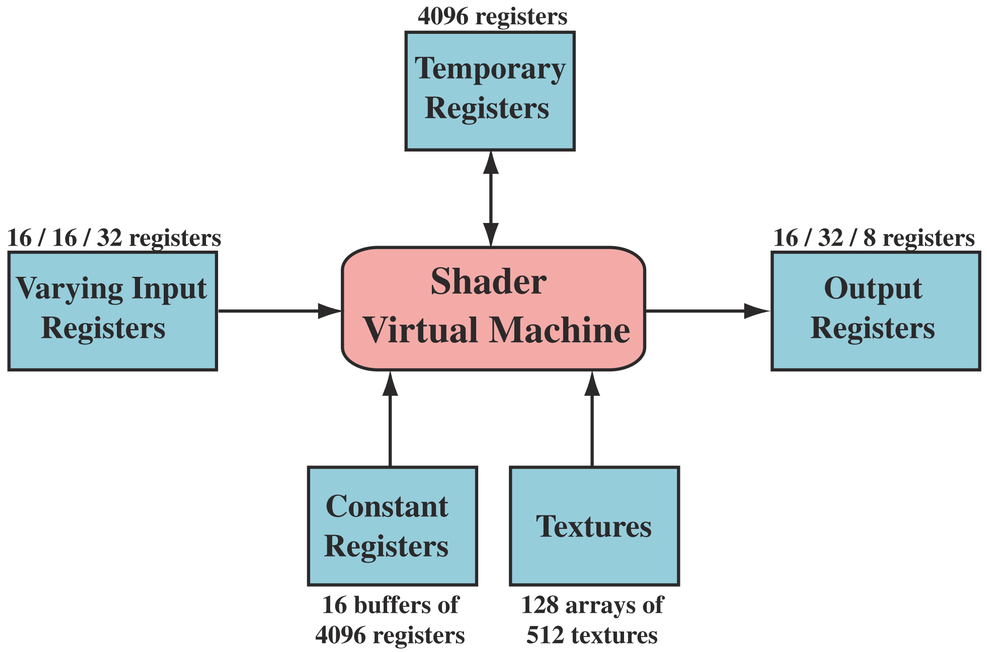
- Modern shader programs use a unified shader design
- Vertex, pixel, geometry and tessellation shaders share a common programming model
- The same instruction set is used - Unified shader architecture
- A pool of identical shader cores is allocated to different parts of the pipeline as seen fit
  - A mesh with finer triangles will need more vertex shaders than a mesh with larger ones.
  - A unified shader architecture can do load balancing while a fixed pool distribution cannot
- Shaders are programmed with C-like shading languages (HLSL, GLSL)
- HLSL can be compiled to intermediate language (IL or DXIL) and stored. Upon execution the graphics driver translates this IL to the specific instruction set of the hardware.

The Programmable Shader Stage

- Basic data types: 32-bit single-precision floating point scalars and vectors
- Modern GPUs also support 32-bit integers and 64-bit floats
- Floating point vectors typically contain positions (xyzw) normals, colors (rgba), matrix rows, texture coordinates (uvwq)
- The language model also supports aggregate data types (structures, arrays and matrices)
- A draw call to the API, to draw a group of similar primitives sets the machinery in motion

The Programmable Shader Stage

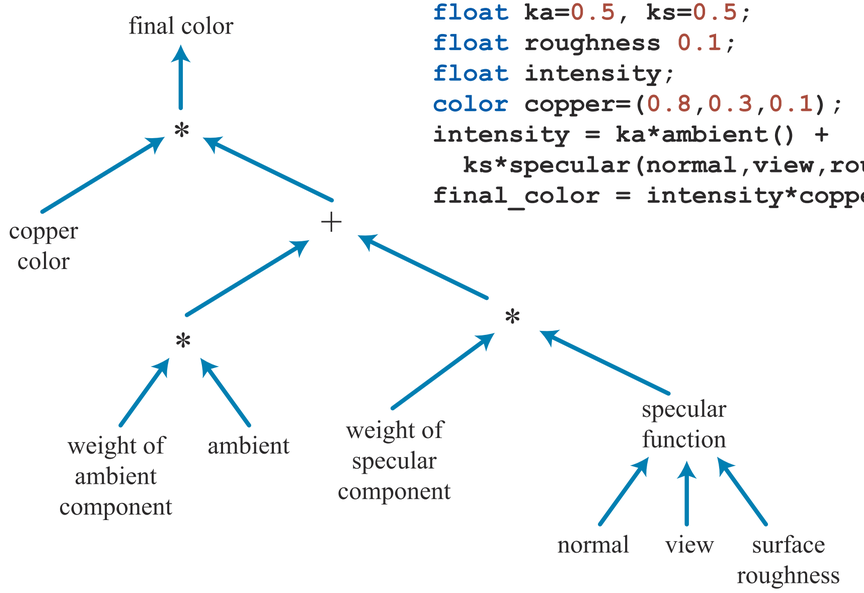
- Each programmable shader stage has two types of input:
  - Uniform inputs (remain constant throughout the draw call)
  - Varying inputs (coming from vertex data or from rasterization)
- Input and output register are fewer than constant and temporary registers.
- Efficient operations: *, /, intrinsic functions atan(), sqrt(), log(), vector normalization, reflection, dot and cross products, matrix transpose and determinants.
- 2 types of flow control: static (based on the values of uniform variables) and dynamic (based on varying inputs)
- Static flow control has no thread divergence, while dynamic flow control may have serious costs in performance.



Unified shader model 4.0 architecture and register layout

The Evolution of Programmable Shading and APIs

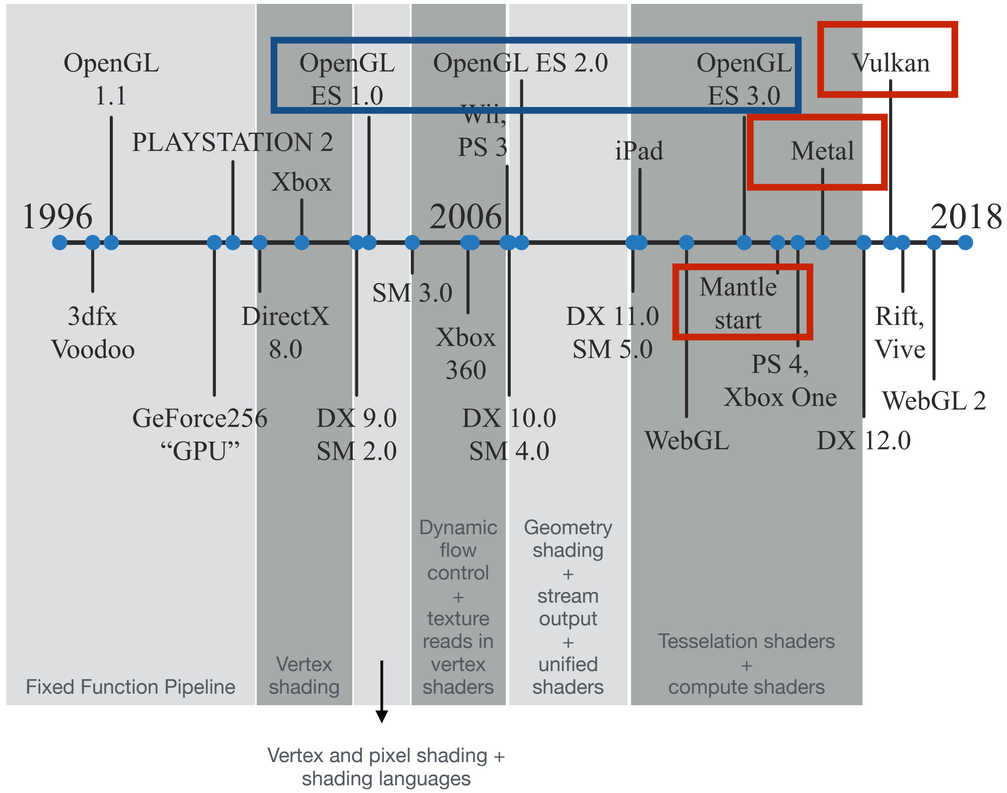
- Cook's shade trees [1984] were the first programable framework for shading
- RenderMan's shading language adopted the idea in the late 1980s. It is still used today for movie production
- OpenShadingLanguage is also inspired in this idea and is used in several products (Blender/Cycles, Sony Imageworks/Arnold, appleseed, V-Ray, …)

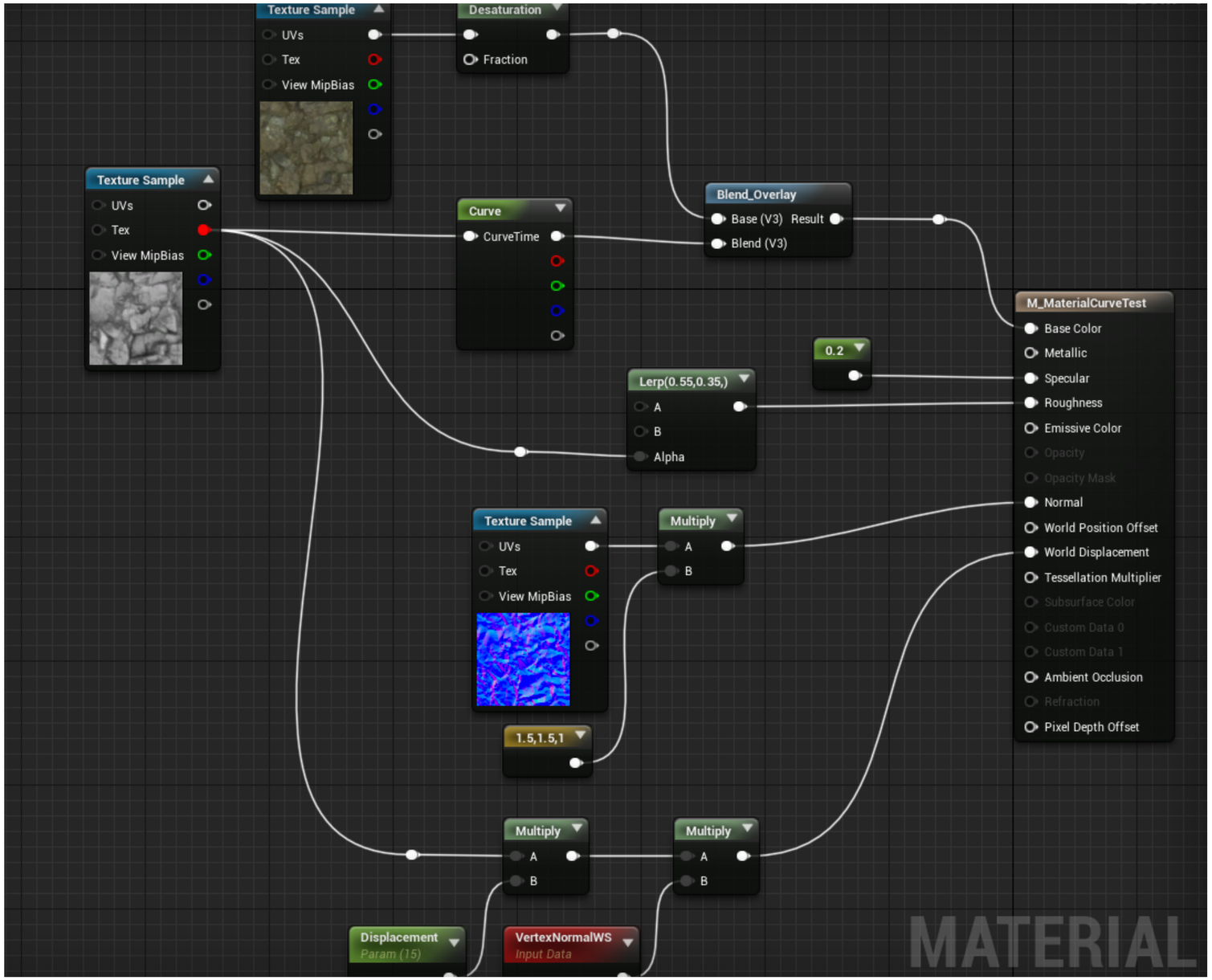


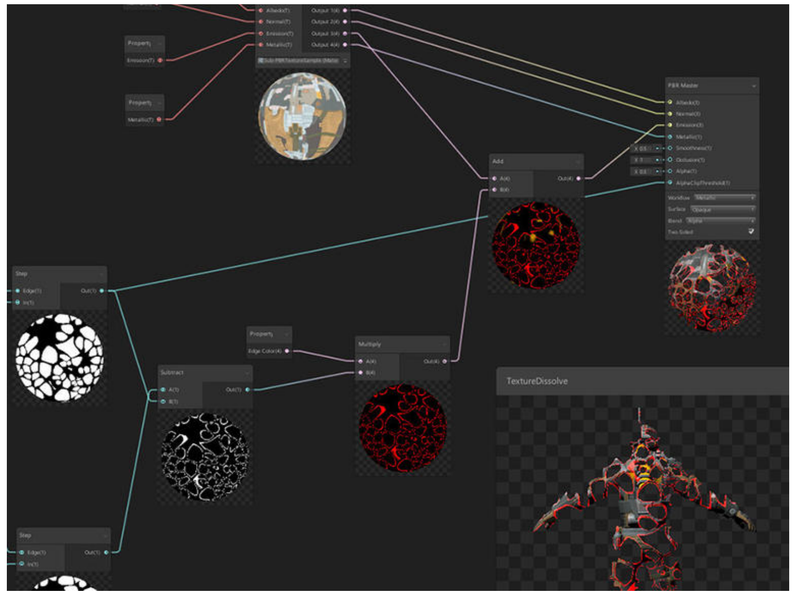

```
float ka=0.5, ks=0.5;
float roughness 0.1;
float intensity;
color copper=(0.8,0.3,0.1);
intensity = ka*ambient() +
  ks*specular(normal,view,roughness);
final_color = intensity*copper;
```

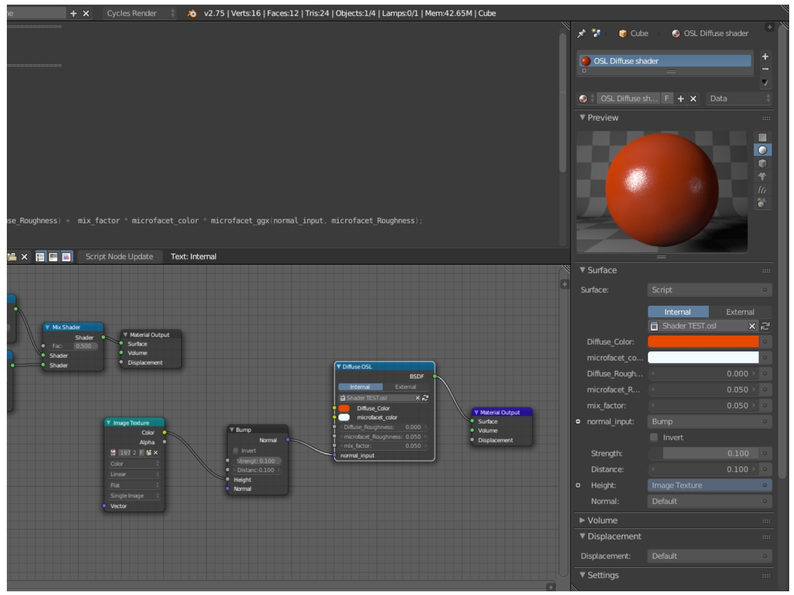
The Evolution of Programmable Shading and APIs

- Consumer level accelerated 3D graphics started with 3dfx VoodooGraphics Card [1996] (daughter card)
- First real programmable GPUs with DirectX 8.0 Vertex shading abilities written in assembly language. Still no branching and limited data types
- DirectX 9.0 [2002] programmable vertex and pixel shaders. Addition of texture reads, writing floating point values, static flow control and a shading language HLSL. OpenGL came up with GLSL
- Shader Model 3.0 added dynamic flow control and texture lookups in vertex shaders. More resources available to shaders (registers, shader length)
- DirectX 10.0 introduced SM 4.0 with geometry shaders, stream output and a unified shader model
- DirectX 11.0 + SM 5.0 introduced the tessellation shader and the compute shader (General Purpose GPU Computing)
- Mantle API [2013] intended to strip the graphics driver overhead and put the control in the developer, leading to a very low level API and better CPU multiprocessor support (Metal and Vulkan followed)
- DirectX 12 is an API redesign and also incorporates this low level API idea. No new functionality is exposed from previous 11.3 release
- Mobile device support is achieved through OpenGL ES lineage of releases + Metal (On Apple devices)



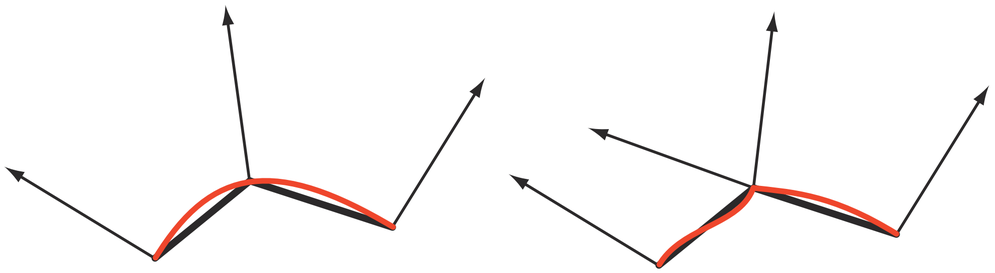

Texture Sample
UVs
Tex
View MipBias
Desaturation
Fraction
Texture Sample
UVs
Tex
View MipBias
Curve
CurveTime
Blend_Overlay
Base (V3) Result
Blend (V3)
M_MaterialCurveTest
Base Color
Metallic
Specular
Roughness
Emissive Color
Opacity
Opacity Mask
Normal
World Position Offset
World Displacement
Tessellation Multiplier
Subsurface Color
Custom Data 0
Custom Data 1
Ambient Occlusion
Refraction
Pixel Depth Offset
0.2
Lerp(0.55,0.35,)
A
B
Alpha
Texture Sample
UVs
Tex
View MipBias
Multiply
A
B
1.5,1.5,1
Multiply
A
B
Multiply
A
B
Displacement
Param (15)
VertexNormalWS
Input Data
MATERIAL

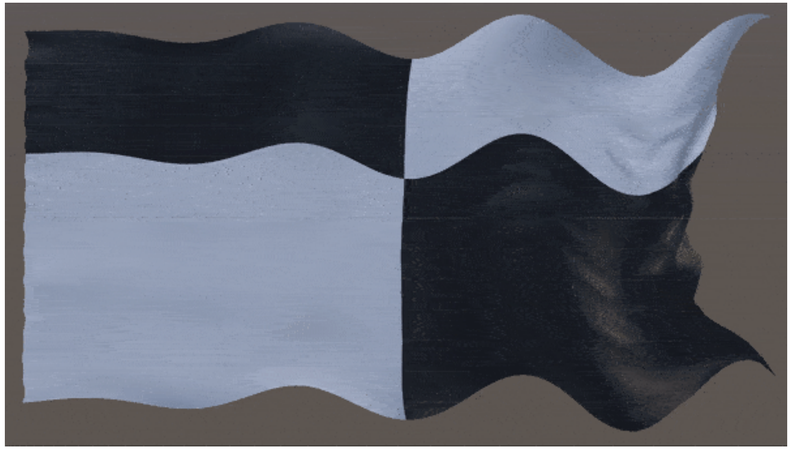


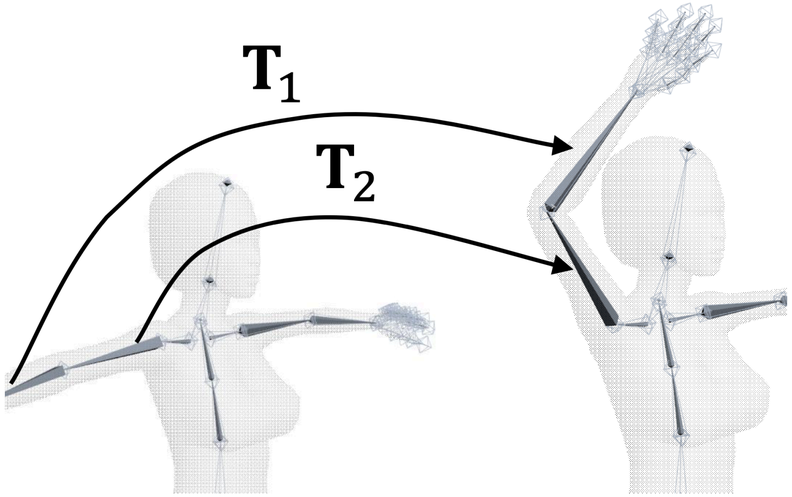


The Vertex Shader

- It is the first programmable stage but some data manipulations takes place before (feed the shaders with data from buffers and constant registers
- Meshes are represented by a set of vertices, each with a specific position on the model surface. Vertex data may also include color, texture coordinates and surface normals (among others)
- The vertex shader processes the vertices of the mesh but without knowledge of the actual triangles in it
- From the set of inputs of each vertex, the shader **has to** output a final vertex position, along with data to be Interpolated across the primitive
- Typical behavior is to go from model space to clip space (Model + View + Projection transformation)
- It cannot create nor destroy vertices and vertices are processed in an independent way
- The driver can decide how many GPU processors to allocate to process a stream of vertices

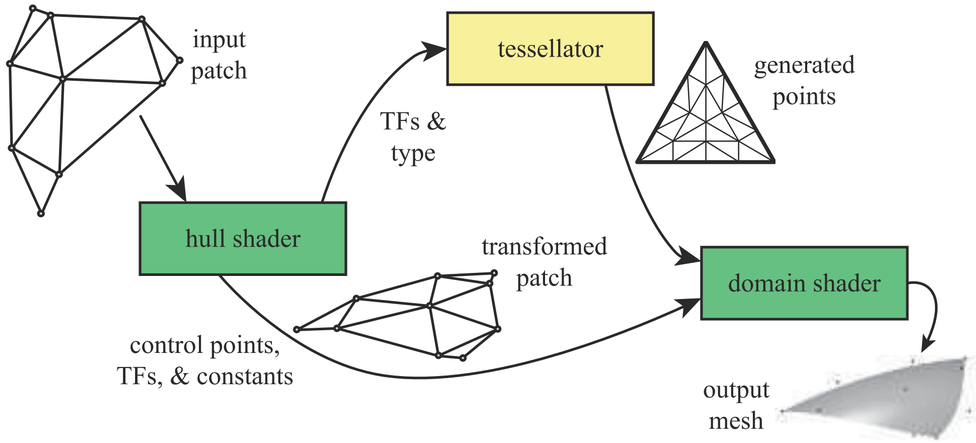




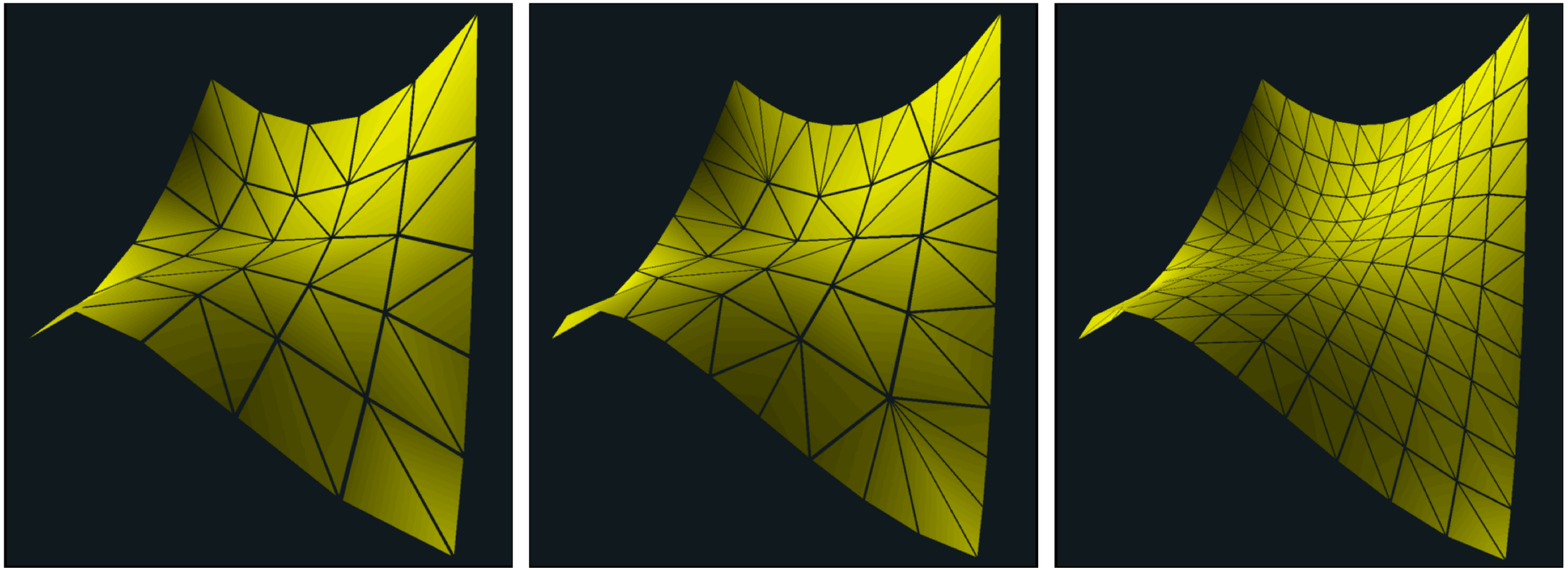

$\mathbf{T}_1$
$\mathbf{T}_2$

The Tessellation Stage

- Suitable to render curved surfaces
- Processes a patch (control mesh) and generates a more refined mesh
- Less stress on the bus between CPU and GPU
- Level of detail can vary depending on actual needs
- Comprises 3 stages:
  - Hull shader/Tessellation control shader
  - Tessellator/Primitive generator (fixed function)
  - Domain shader/Tessellation evaluation shader

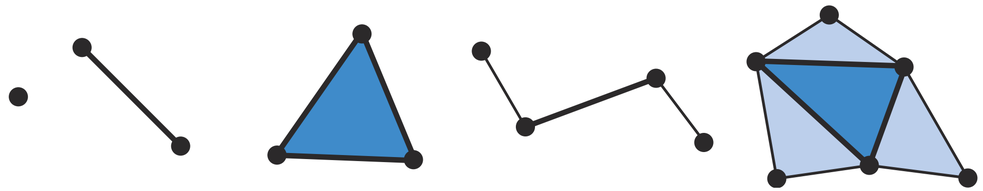


The Tessellation Stage



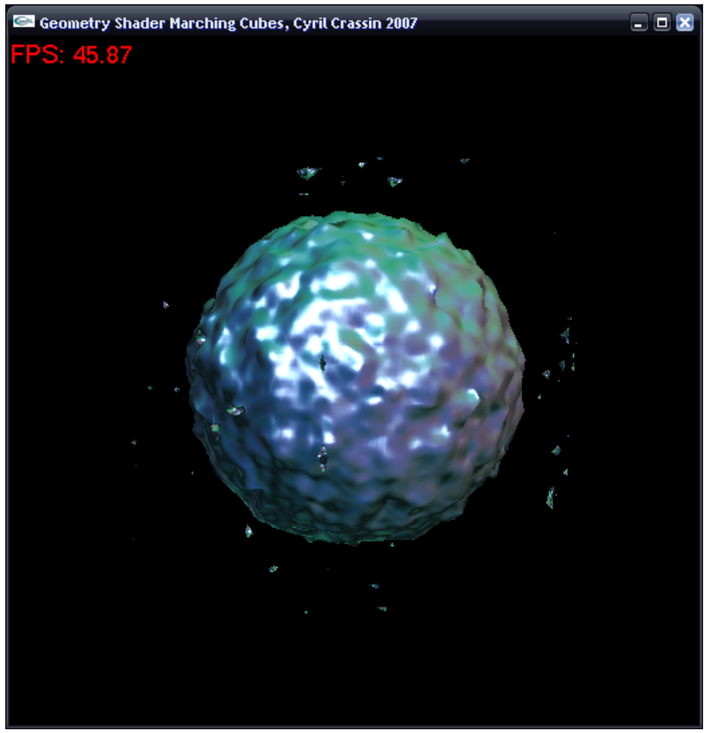
The Geometry Shader

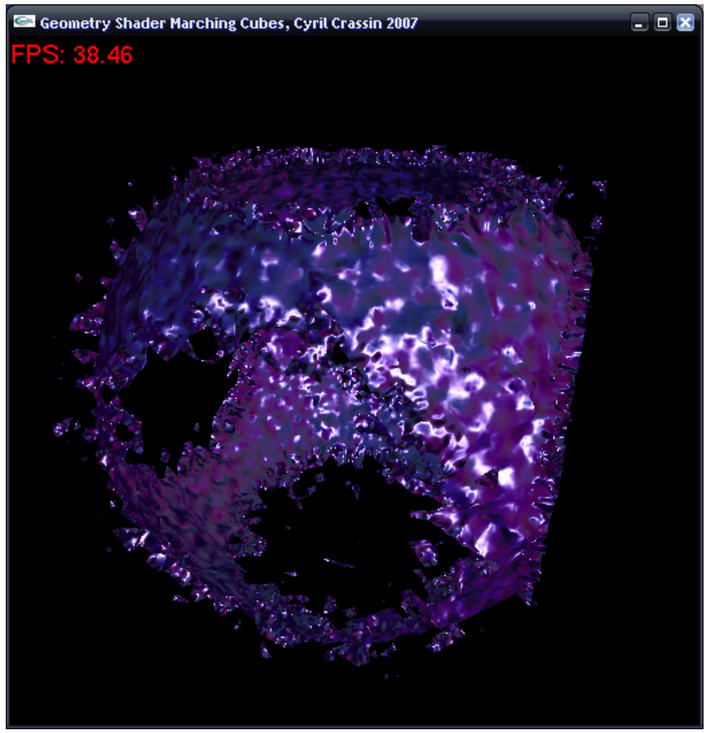
- The geometry shader can turn primitives into other primitives
  - Triangle -> lines
  - Lines -> quads
  - Point -> quad
- Outputs 0+ vertices (treated as points, lines os strips of triangles)
- The least used from all the shaders (some mobile devices implement it in software as it is the more generic one and deviates from traditional shader model)

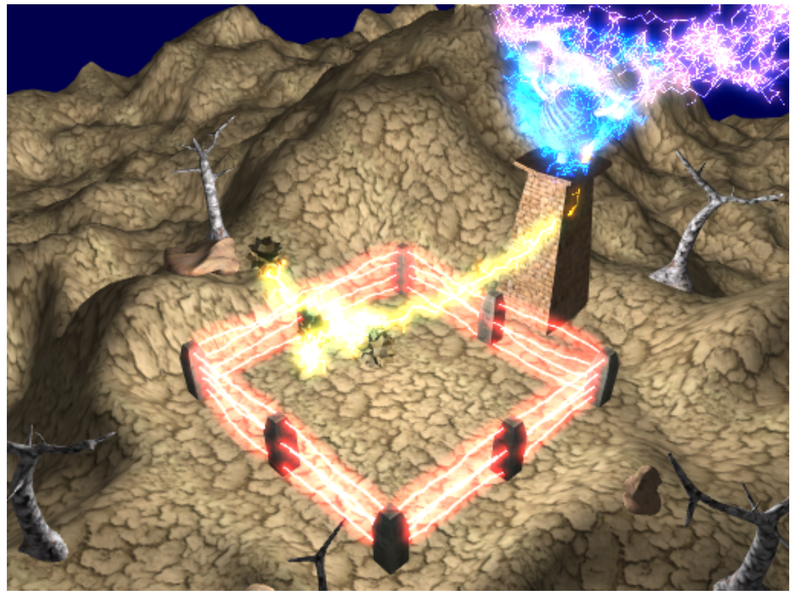


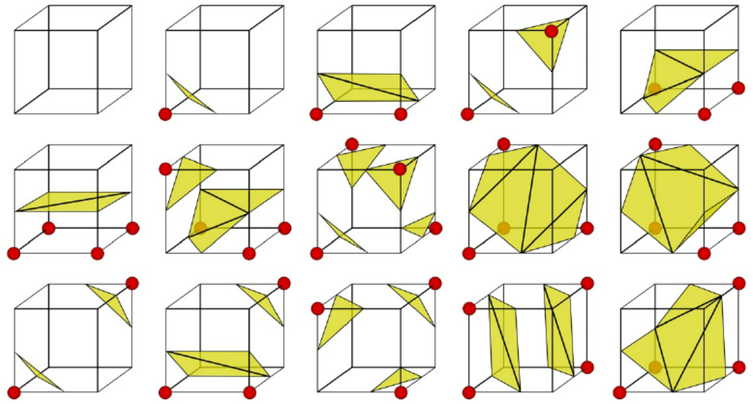
Inputs:

SM 4.0
point, line [+2 points], triangle [+3 points]
SM 5.0
Up to 32 control points


Geometry Shader Marching Cubes, Cyril Crassin 2007
FPS: 45.87

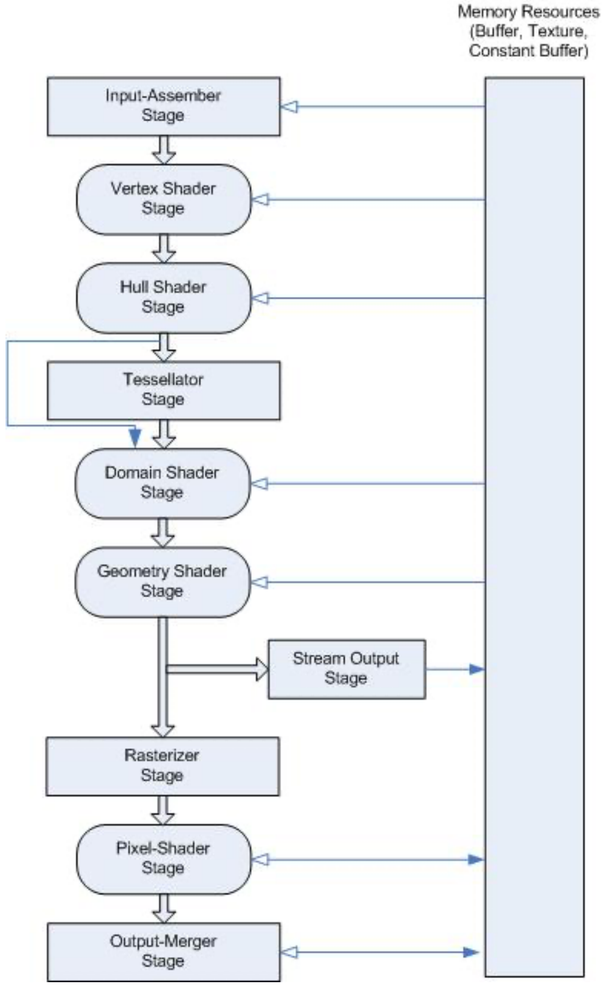

Geometry Shader Marching Cubes, Cyril Crassin 2007
FPS: 38.46





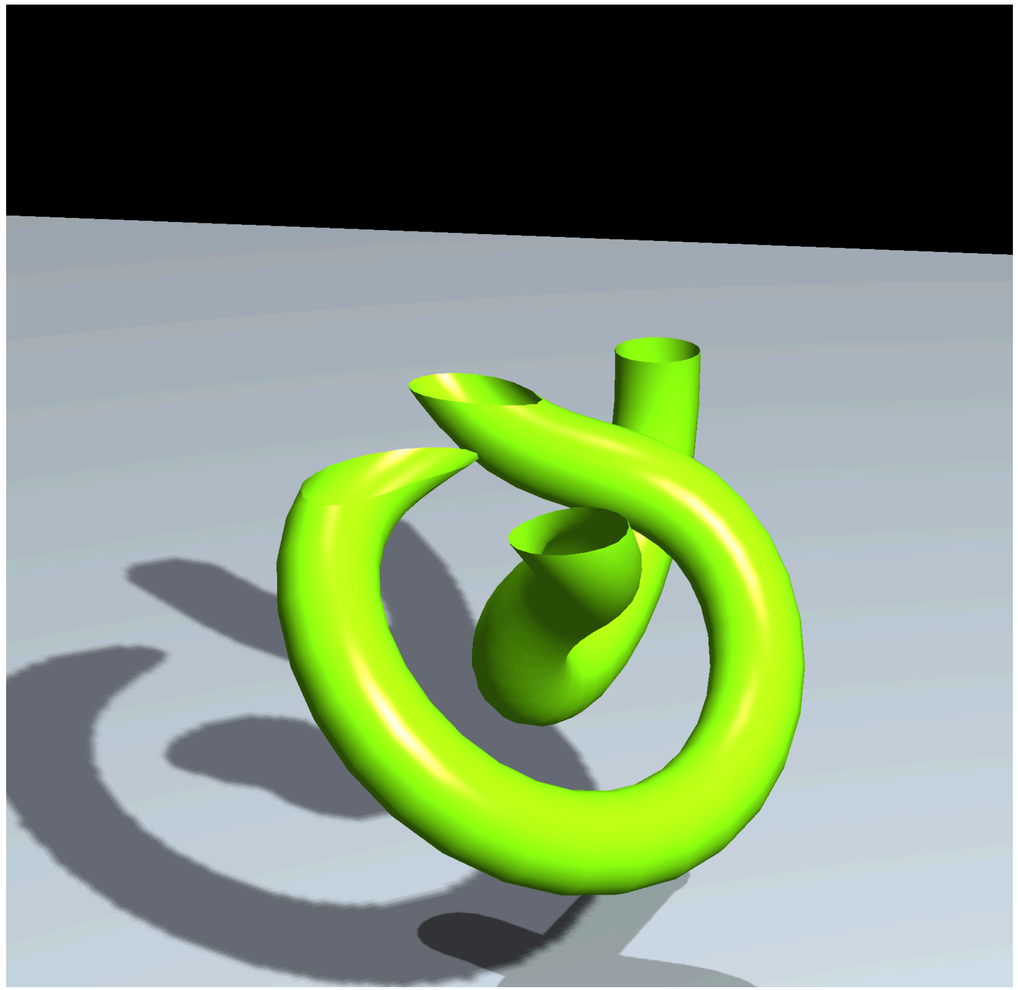
Stream Output

- SM 4.0 introduced the idea of using the first stages of the pipeline as a non-graphical stream processor [rasterization can even be turned off]
- Vertex shader [-> tessellation shader -> geometry shader] -> output stream
- The output stream (ordered list of vertices) could be sent back through the pipeline (iterative process)
- Stream output works on primitives (not on vertices) and outputs floating point values. A triangle will generate 3 vertices. If input data shares vertices, the share is lost
  - Normally it is used with point primitives
- In OpenGL this is called *transform feedback* since we transform vertices and return them to the start of the pipeline
- Primitives are sent to the output in the order they were received

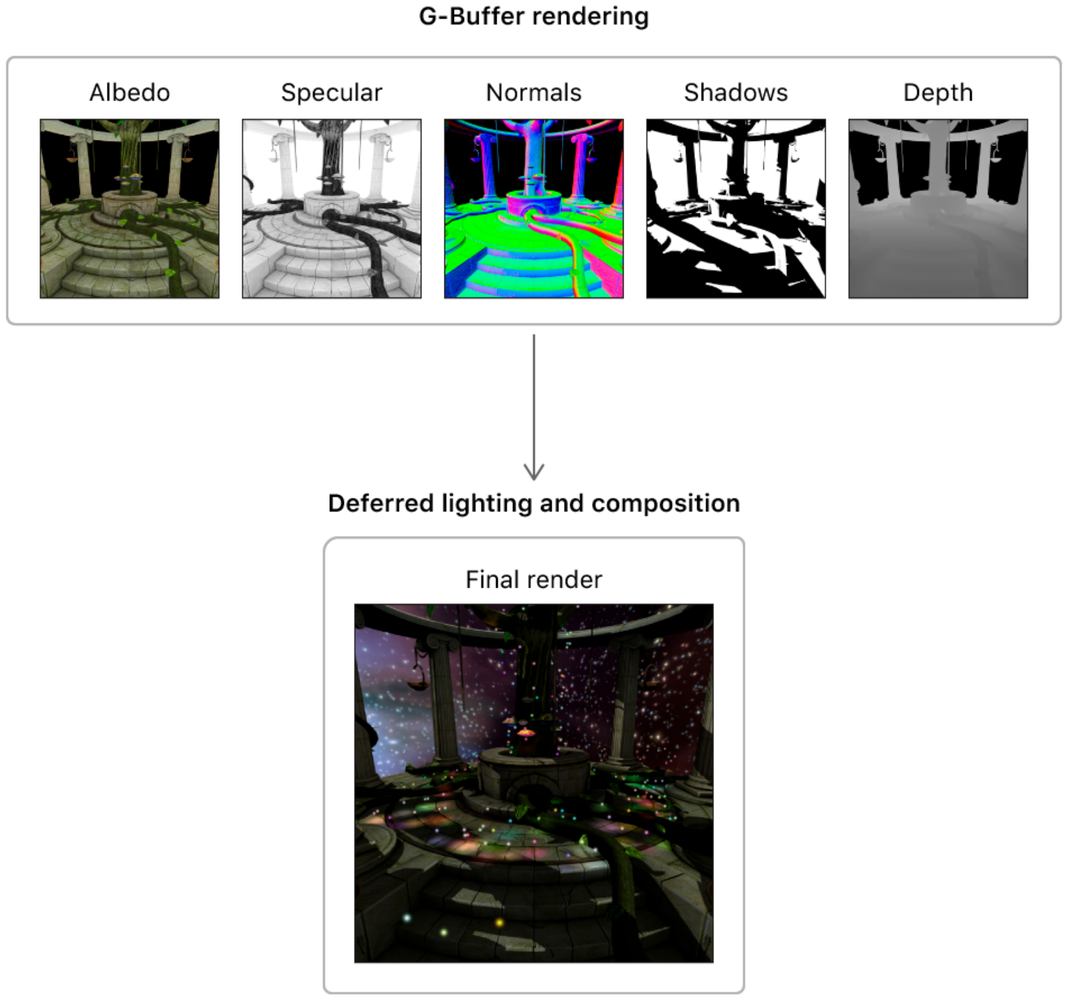


The Pixel Shader

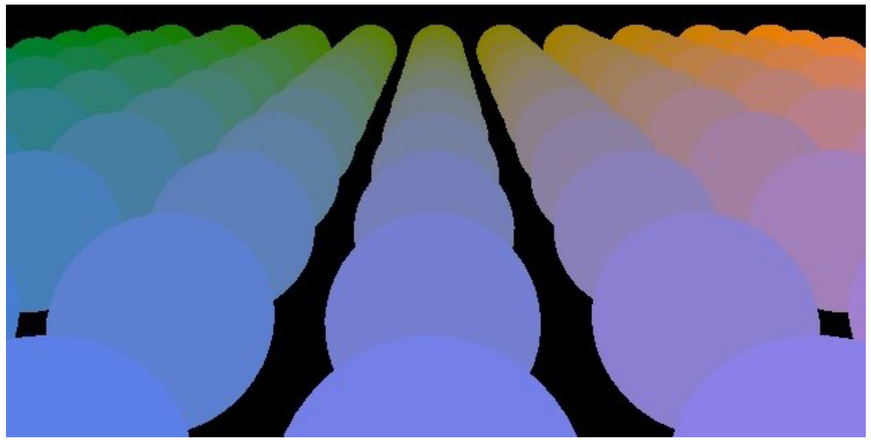
- The pixel shader is also known as fragment shader (a pixel that is only partially covered by the primitive)
- Its primary goal is to compute a final color (and opacity) for the pixel/ fragment (or discard it)
- The pixel shader receives an interpolated z-value (it can be modified too)
- interpolated quantities can be perspective corrected (interpolated in World Space) or not (interpolated in screen space)
- The pixel shader inputs are the vertex shader outputs after interpolation
- Additional inputs are present (screen space position of the fragment, which side of the face it belongs to)
- It is possible to output to more than one render target
  - World space position, normal, object ids, etc…
  - Allow for alternative rendering pipelines such as deferred shading
- Cannot read adjacent pixel data (not directly at least)

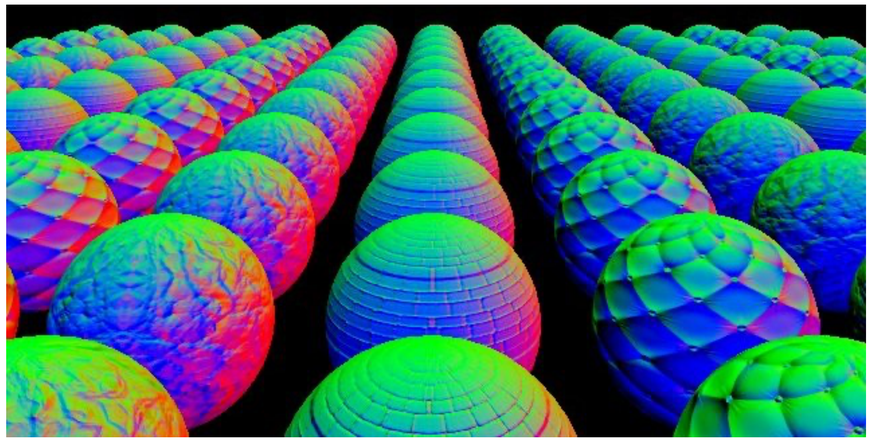


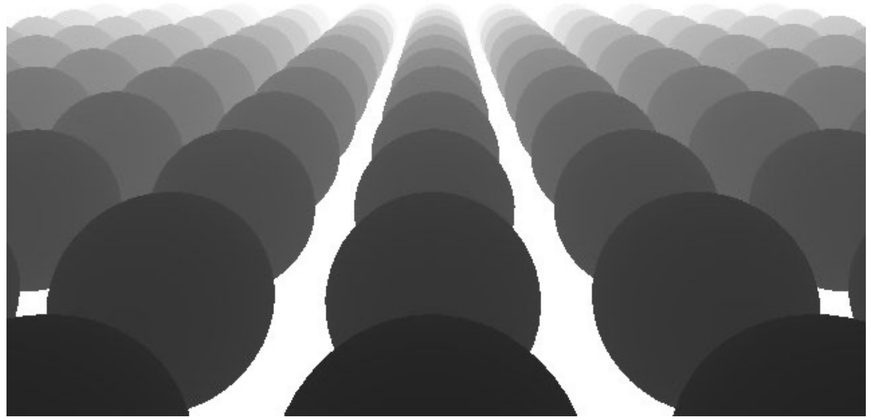
Deferred Shading

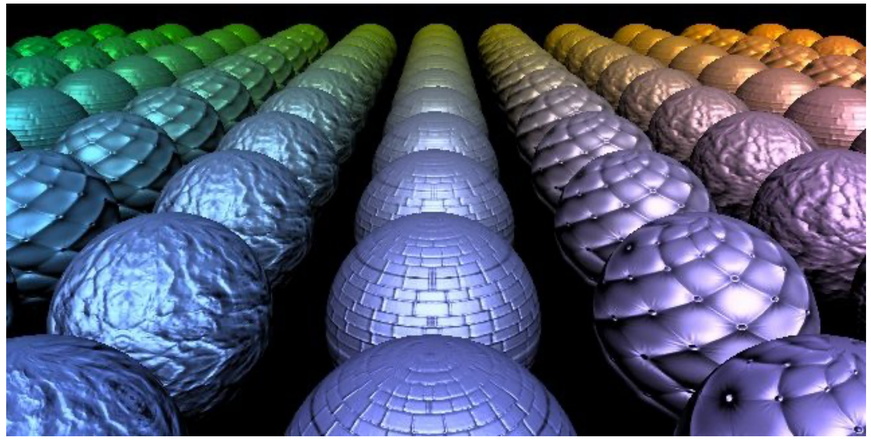


Deferred Shading



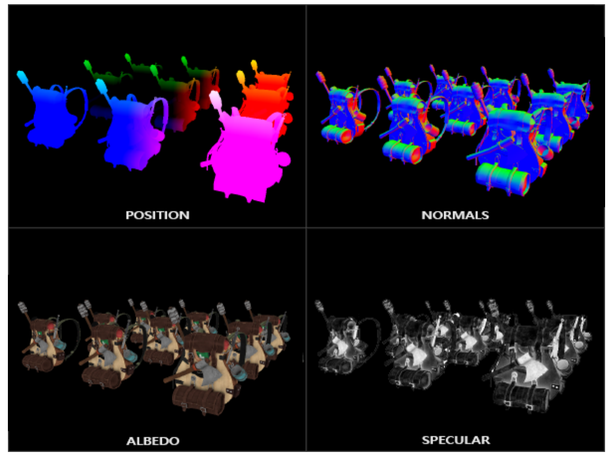


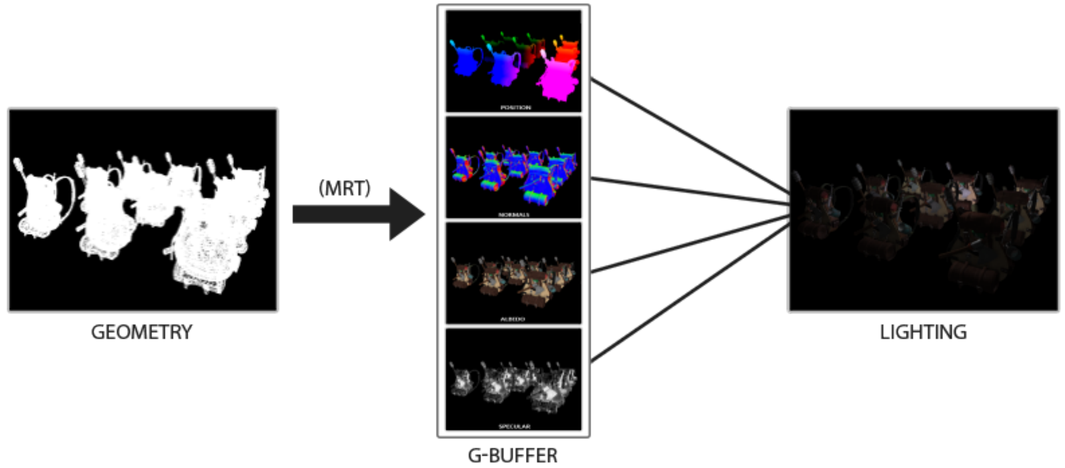




Deferred Shading

- The base idea is to defer or postpone most of the heavy rendering tasks to a later stage
  - 1st pass: Geometry pass - render all types of geometric information (positions, normals, colors, specular values, etc.) to separate textures (render targets). This is called a G-Buffer (Geometry Buffer)
  - 2nd pass: Lighting pass - render a quad filling the screen. For each fragment (visibility has already been determined), compute the lighting
- Pros: no wasted time computing expensive lighting on fragments that will not be visible
- Cons: No alpha blending
- It is possible to combine forward with deferred rendering
- This technique paves the way for rendering hundreds or thousands of light sources in a scene! How?



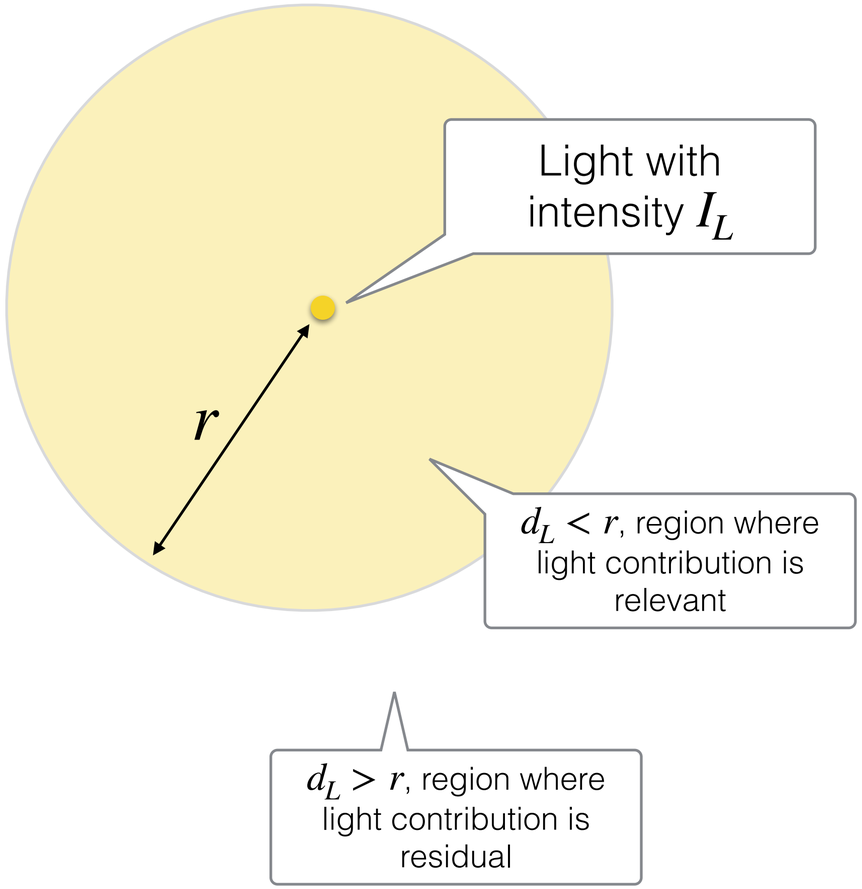


Deferred Shading and lots of lights

- Every light source has a limited radius. From light source attenuation term we can compute this radius by choosing a suitable lower threshold for I and solve for distance

$$I = \frac{I_L}{c_0 + c_1 d_L + c_2^2 d_L}$$

- Having the lighting stage shader iterate through each light source and check for its distance is doable but not very efficient
- It is better to draw spheres located at each light source, with the corresponding radius) in the lighting stage (one call per light source) and accumulate each light's contribution
- Two spheres touching each other should not occlude each other (depth write disabled)
- Only half of the pixels of a sphere should be considered and additive blending should be turned on
- We go from $\mathcal{O}(N_O \times N_L) \Rightarrow \mathcal{O}(N_O + N_L)$

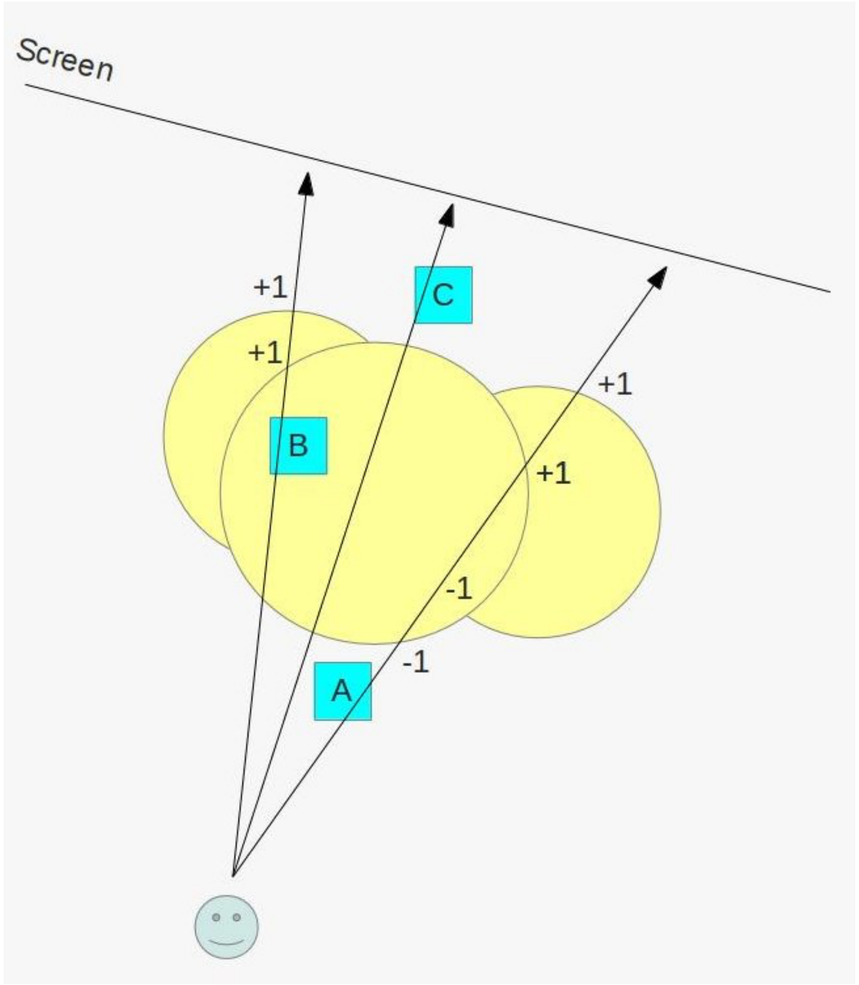


Stencil buffer

- The stencil buffer is a buffer with the same resolution as the framebuffer and the depth buffer, and typically with 1 byte per pixel.
- Stencil and depth tests control if the pixel is written to the frame buffer
- Application defines stencil test(s) and stencil operation(s)
- Possible stencil test functions:
  - Always pass
  - Always fail
  - Less/greater than
  - Less/greater or equal than
  - Equal
  - Not equal
- Possible stencil operation:
  - Keep the stencil unchanged
  - Replace with 0
  - Increment/decrement
  - Invert the bits
- Stencil operations are configured independently for these situations:
  - Stencil test failure
  - Depth test failure
  - Depth test success
- Stencil tests and operations can be independently set for each face side (front/back)

Deferred Shading and lots of lights (details)

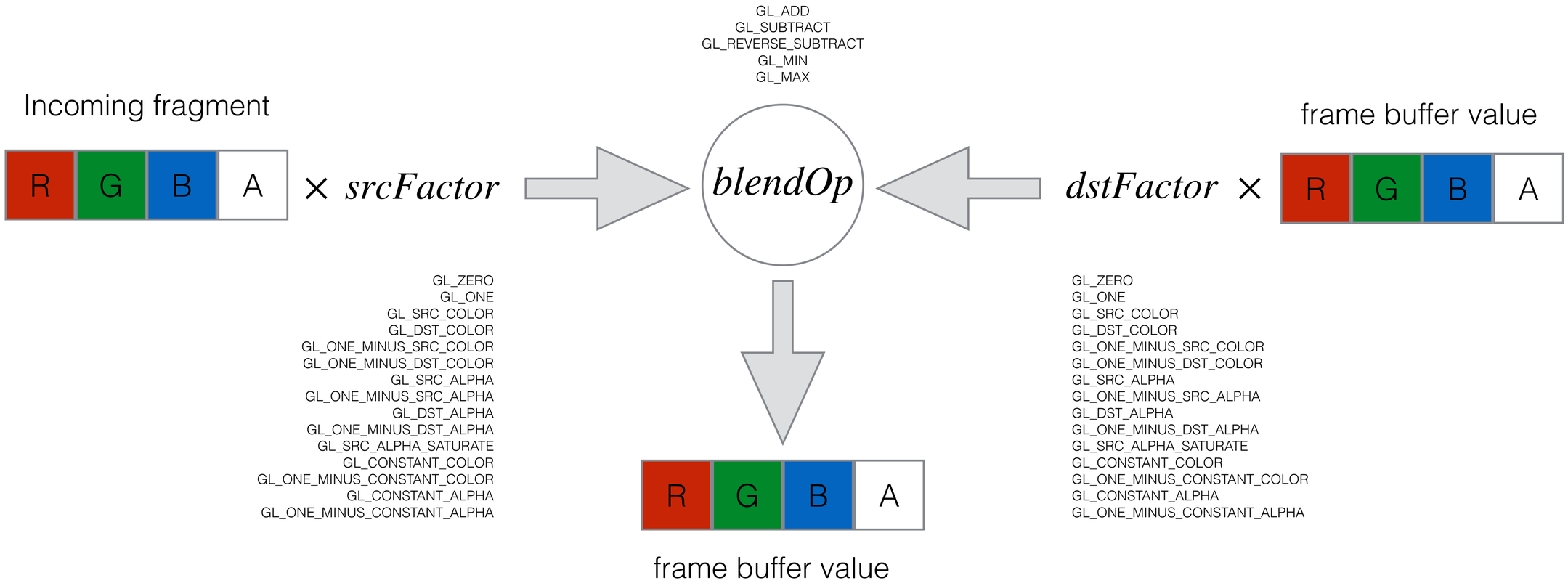
- Render objects as usual into the G-Buffer, including the depth buffer data
- Disable writing into the depth buffer
- Disable back face culling
- Set the stencil test to always succeed (we just need the stencil operation here)
- Set stencil operation for back facing polys to increment when depth test fails; otherwise no change
- Set stencil operation for front facing polys to decrement when depth test fails; otherwise no change
- Render sphere for light with null pixel shader
- Finally render the sphere for the light again and set the stencil test to pass if different from 0.
- This will also work for several light sources



The Merging Stage

- Combines depths and colors of the individual fragments
- Stencil buffer and z-buffer operations occur in this stage
- If a fragment is visible it gets blended with the existing color or replaces it
- Some GPUs allow some merging operations to be performed before the pixel shader gets executed (avoid spending time computing colors for non visible fragments)

Alpha Blending



Over operator (back to front order): $A_{src}RGB_{src} + (1 - A_{src})RGB_{dst}$

Under operator (front to back order): $C_{dst} = A_{dst}(A_{src}C_{src}) + C_{dst}$

$$A_{dst} = 0 + (1 - A_{src})A_{dst}$$

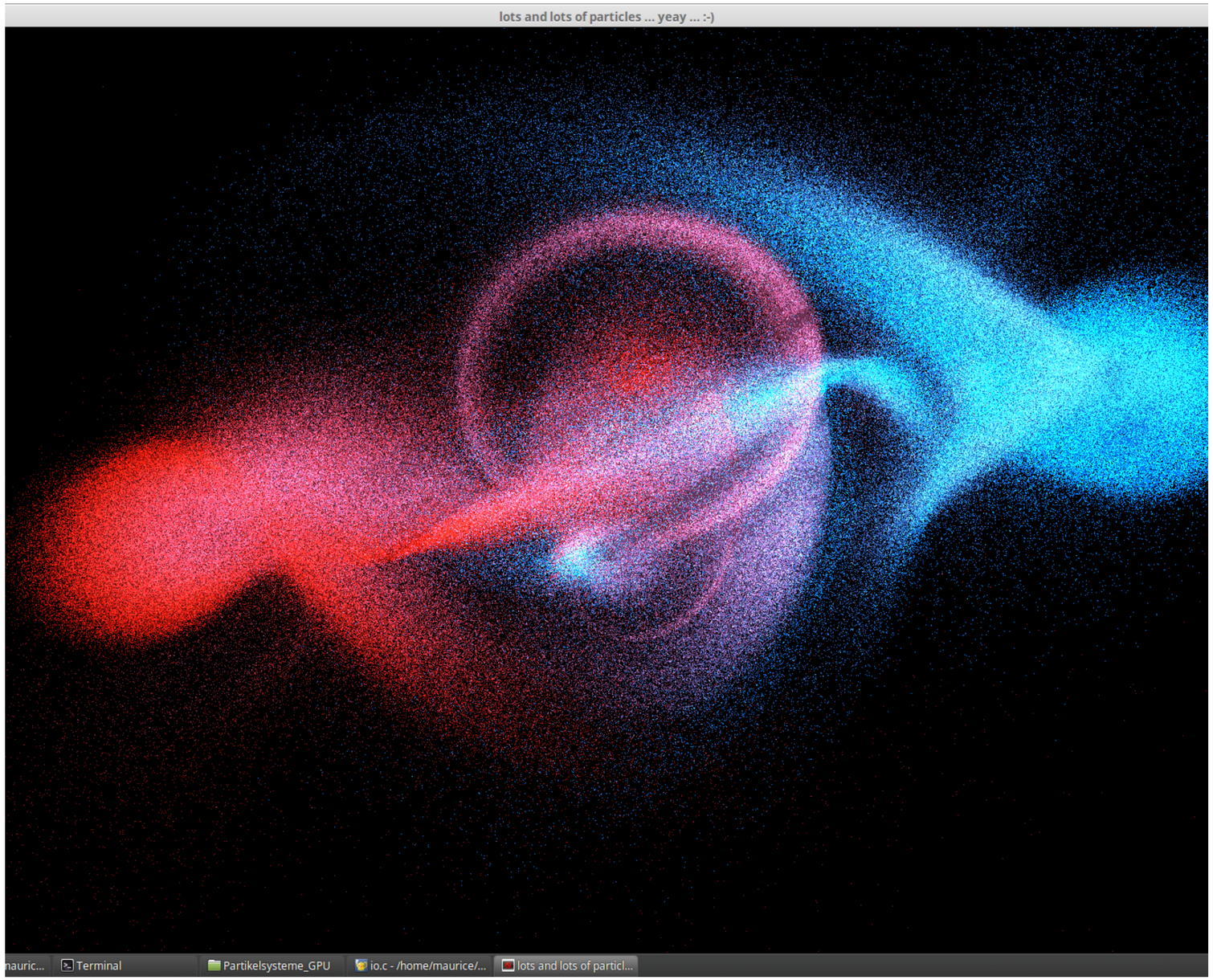
$$C_{dst} = A_{dst}C_{bg} + C_{dst}$$

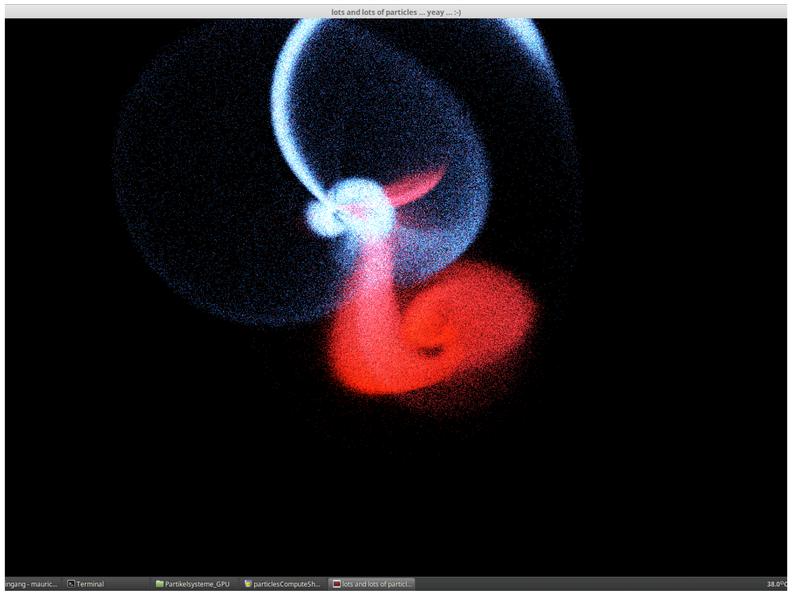
The Compute Shader

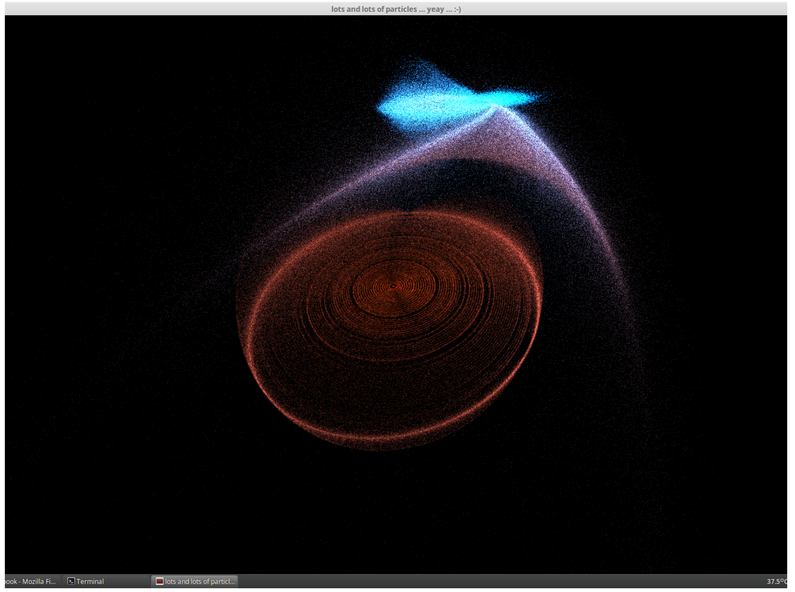
- The GPU can also be used to perform general computations
- Application areas where GPUs are commonly used:
  - Financial markets (stock order predictions)
  - Neural networks and deep learning
  - Physics simulation, particle systems
  - Virtual currency
- CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language) are the most widely used platforms (languages, compilers and libraries) to explore the power of the GPUs for general purpose computing.

The Compute Shader

- The Compute Shader was introduced in DirectX 11
- It is a shader that is not locked to a specific part of the graphics pipeline, is invoked by the graphics API and it shares the resources (pool of shader processors) used in the pipeline.
- It receives input data, can access buffers (such as textures) for input and output. Threads and warps (thread bundles running on a same processing unit) are visible in compute shaders
- Each invocation of a compute shader receives a thread index.
- Thread groups offer x- y- and z- coordinates in a grid and share resources among threads, such as fast shared memory (typically 32KB in DirectX).


lots and lots of particles ... yeay ... :-)
Terminal
Partikelsysteme_GPU
io.c - /home/maurice/...
lots and lots of particl...


lots and lots of particles ... yeay ... :-)


lots and lots of particles ... yeay ... :-)

Compute Shader (Example)

```
#version 430 core

// Process particles in blocks of 128
layout (local_size_x = 128, local_size_y = 1, local_size_z = 1) in;

layout (std430, binding = 0) buffer PositionBuffer {
     vec3 positions[];
};
layout (std430, binding = 1) buffer VelocityBuffer {
     vec4 velocities[];
};
layout (binding = 2) buffer AttractorBuffer {
     vec4 attractors[];
};
layout (std430, binding = 3) buffer LifeBuffer {
     float lifes[];
};
// Delta time
uniform float dt;

highp float rand(vec2 co)
{
    highp float a = 12.9898;
    highp float b = 78.233;
    highp float c = 43758.5453;
    highp float dt= dot(co.xy ,vec2(a,b));
    highp float sn= mod(dt,3.14);
    return fract(sin(sn) * c);
}

float vecLen (vec3 v)
{
     return sqrt(v.x*v.x + v.y*v.y + v.z*v.z);
}

vec3 normalize (vec3 v)
{
     return v / vecLen(v);
}

vec3 calcForceFor (vec3 forcePoint, vec3 pos)
{
     // Force:
     float gauss = 10000.0;
     float e = 2.71828183;
     float k_weak = 1.0;
     vec3 dir = forcePoint - pos.xyz;
     float g = pow (e, -pow(vecLen(dir), 2) / gauss);
     vec3 f = normalize(dir) * k_weak * (1+ mod(rand(dir.xy), 10) - mod(rand(dir.yz), 10)) / 10.0 * g;
     return f;
}
```

```
void main(void)
{
     uint index = gl_GlobalInvocationID.x;

     int i;
     float newDT = dt * 100.0;


     vec3 forcePoint = vec3(0);

     for (i = 0; i < 32; i++) {
          forcePoint += attractors[i].xyz;
     }

     // Read the current position and velocity from the buffers
     vec4 vel = velocities[index];
     vec3 pos = positions[index];
     float newW = lifes[index];

     float k_v = 1.5;

     vec3 f = calcForceFor(forcePoint, pos) + rand(pos.xz)/100.0;

     // Velocity:
     vec3 v = normalize(vel.xyz + (f * newDT)) * k_v;

     // Eine leichte Anziehung richtung Schwerpunkt...
     v += (forcePoint-pos) * 0.00005;

     // Pos:
     vec3 s = pos + v * newDT;

     newW -= 0.0001f * newDT;

     // If the particle expires, reset it
     if (newW <= 0) {
          s  = -s + rand(s.xy)*20.0 -rand(s.yz)*20.0;
          //v.xyz *= 0.01f;
          newW = 0.99f;
     }

     lifes[index] = newW;
     // Store the new position and velocity back into the buffers
     positions[index] = s;
     velocities[index] = vec4(v,vel.w);

}
```

Example taken from: https://github.com/MauriceGit/Partikel_accelleration_on_GPU

Further readings and resources

- Cap. 3 Real Time Rendering - T Akenine-Möller et. Al (adopted book)
- Unity Grass Shader Tutorial - https://roystan.net/articles/grass-shader.html
- Deferred Shading Tutorial - http://ogldev.atspace.co.uk/www/tutorial35/tutorial35.html
- Particle System Compute Shader - https://github.com/MauriceGit/Partikel_accelleration_on_GPU