GPU Computing with fragment shaders

”Classic GPGPU”

Use graphics shaders for general-purpose computing.

Adapt your data and computing to fit the graphics pipeline.

Hot until CUDA arrived, now overshadowed by CUDA and OpenCL.

Why is classic GPGPU interesting?

- Highly suited to all problems dealing with images, computer vision, image coding etc

- Parallelization ”comes natural”, you can’t avoid it and good speedups are likely. Fewer pitfalls.

- Highly optimized (for graphics performance).

- Compatibility is vastly superior!

- Very much easier to install!

So what is not so good?

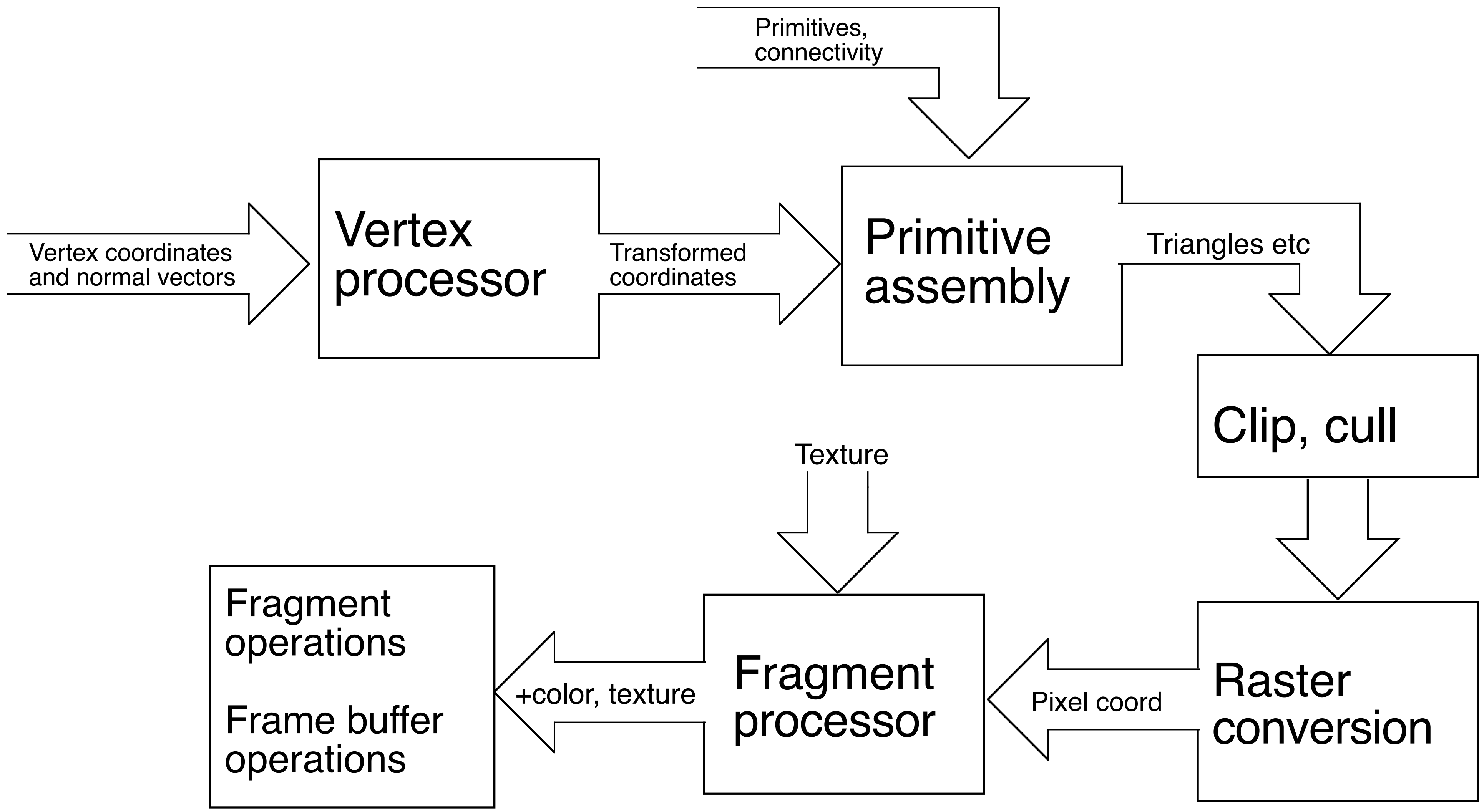
- Must map data to image data

- Computing controlled by pixels in output image

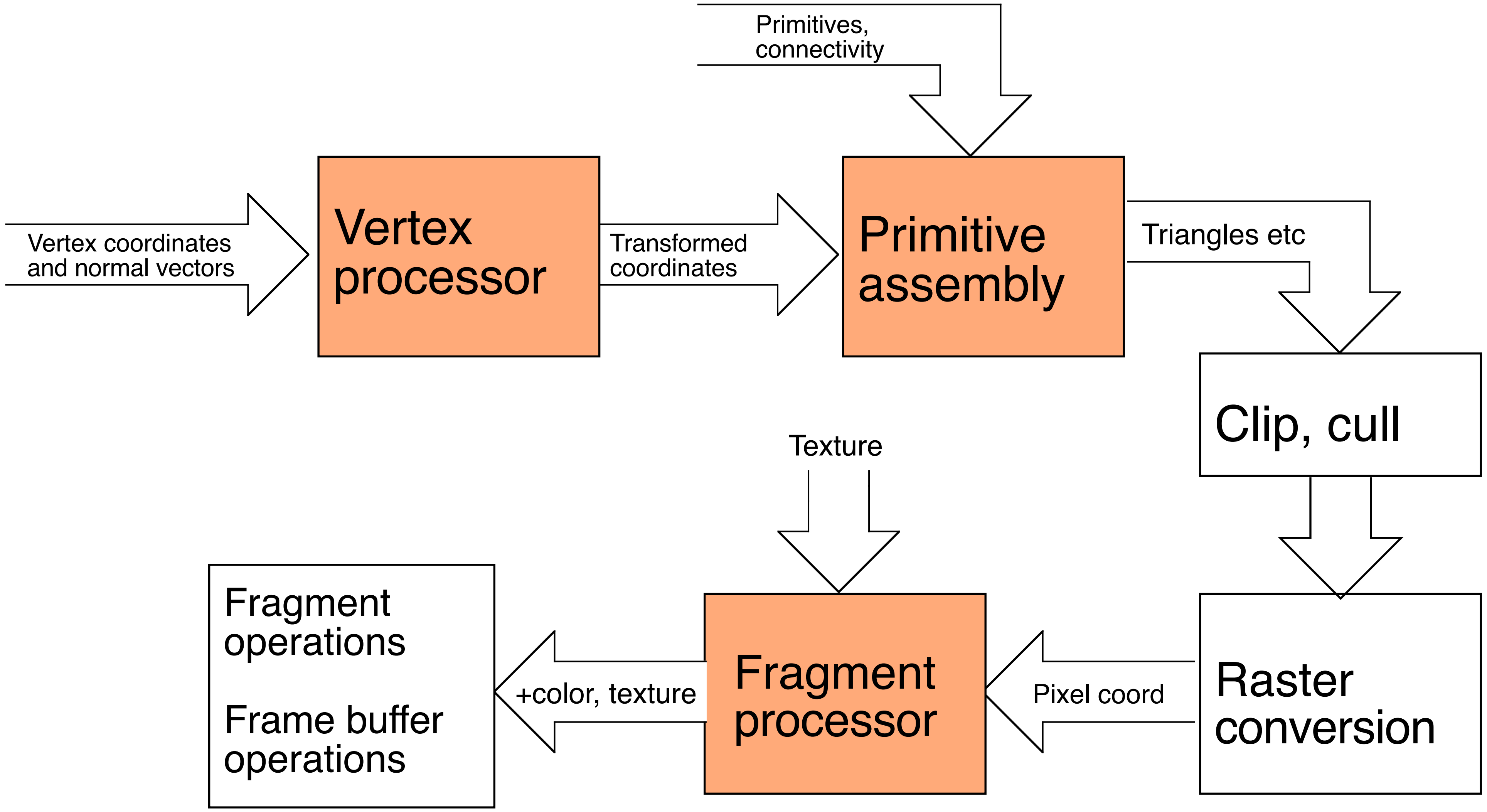
- No shared memory access

However: OpenGL 4 adds much flexibility, moves closer to CUDA and (especially) OpenCL. Writable textures, atomics, synchronization...

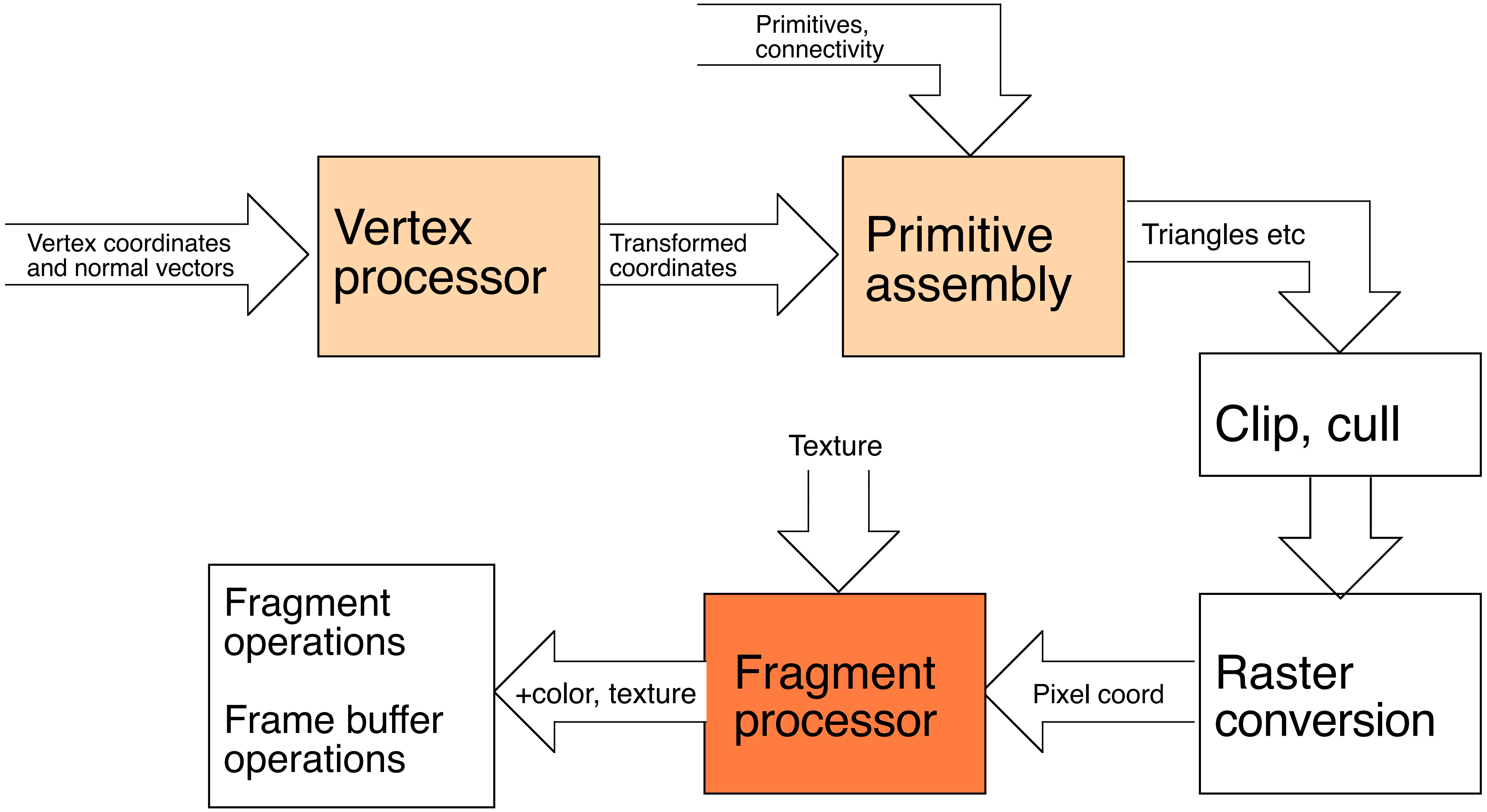
The OpenGL pipeline



Out of these, three are programmable!

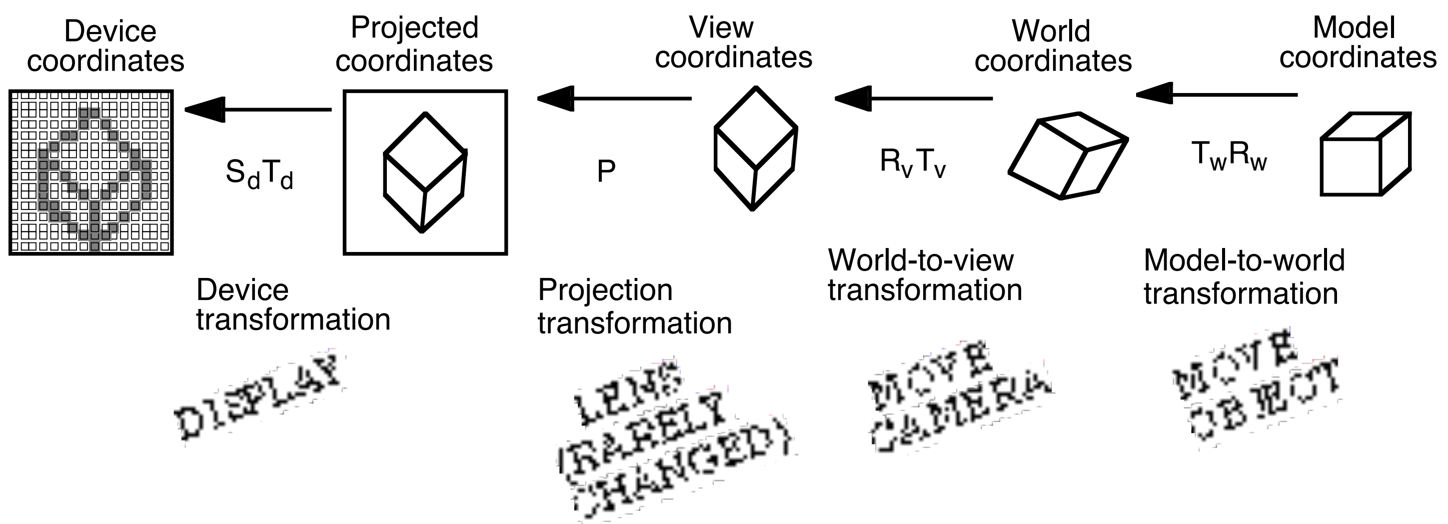


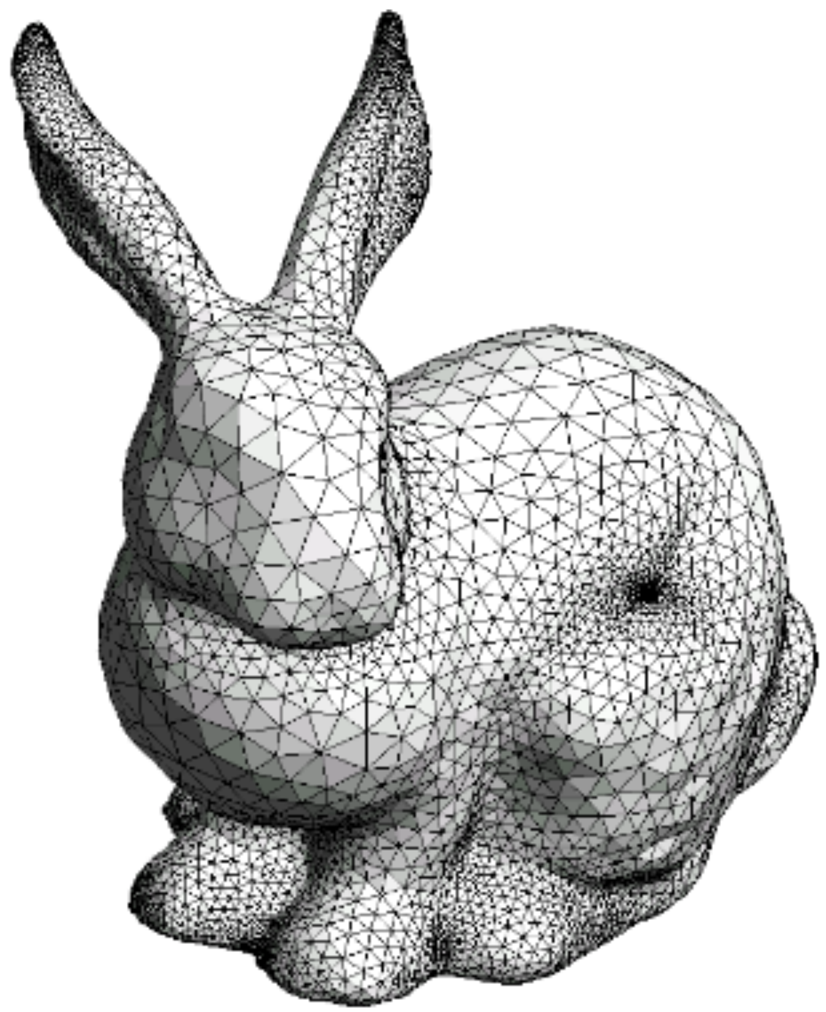
But only one creates easily accessible output data!



Typical OpenGL situation

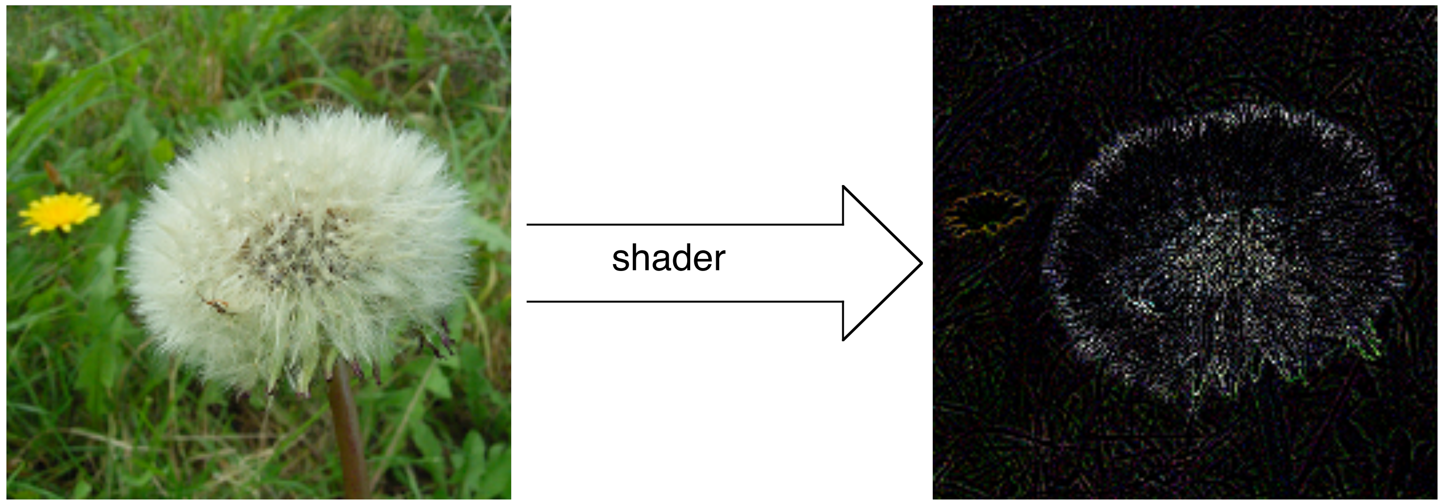
- Complex geometry
- Many transformations
- Perspective projection
- Lighting and material calculations for the surfaces
- Many texture accesses for interpolation and supersampling





Typical GPU Computing with fragment shaders (also used in filtering in graphics):

- Render to a single rectangle covering the entire image buffer.
- Use FBOs for effective feedback
- Floating-point buffers
- Ping-ponging, many pass with different shaders



Render image 1:1 Output

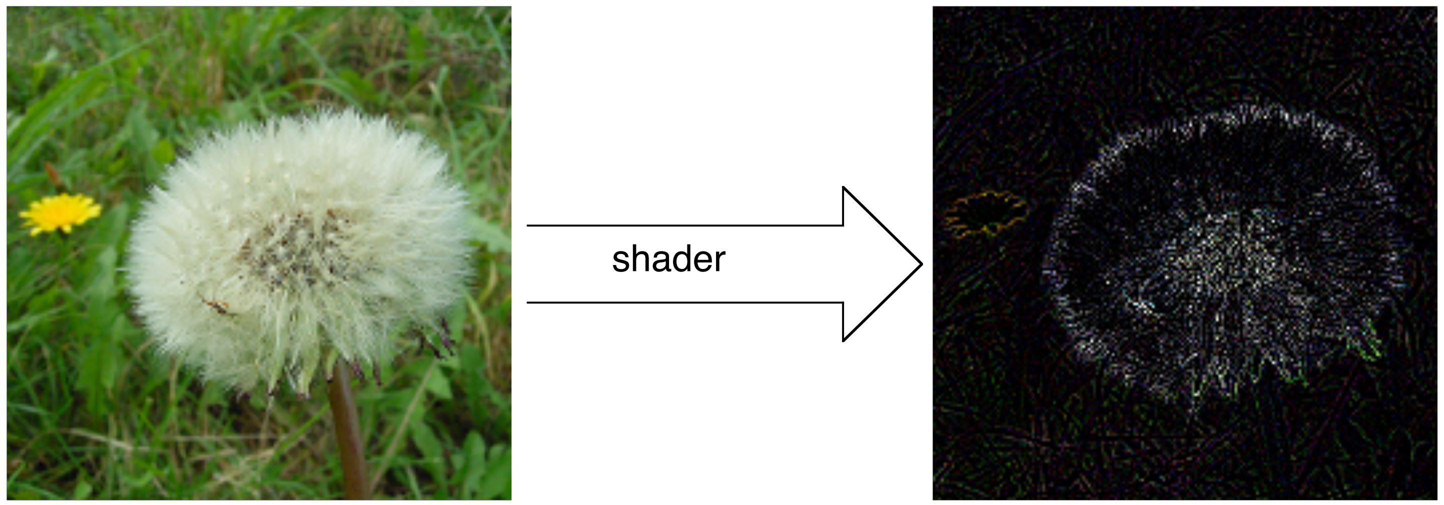
Computing model

- Array of input data = texture
- Array of output data = resulting frame buffer
- Computation kernel = shader
- Computation = rendering
- Feedback = switch between FBO's or copy frame buffer to texture

Computation = rendering

Typical situation:

- Texture and frame buffer same size
- Render the polygon over the entire frame buffer



Texture Frame buffer

Kernel = shader

Shaders are read and compiled to one or more program objects. A GPGPU application can use several shaders in conjunction!

Activate desired shader as needed using glUseProgram();

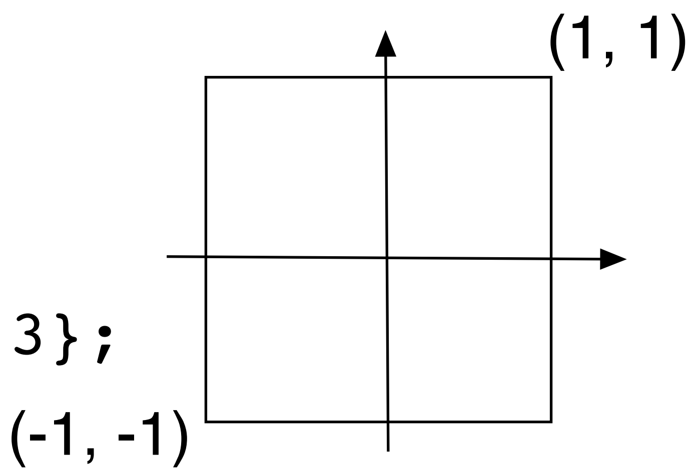
The fragment shader performs the computation:

```
uniform sampler2D texUnit;
in vec2 texCoord;
out vec4 fragColor;

void main(void)
{
   vec4 texVal  = texture(texUnit, texCoord);
   fragColor = sqrt(texVal);
}
```

Render a single polygon

- Texture and frame buffer same size
- Render polygon over entire frame buffer

```
GLfloat quadVertices[] = { -1.0f, -1.0f, 0.0f,
                        -1.0f, 1.0f, 0.0f,
                         1.0f, 1.0f, 0.0f,
                        1.0f, -1.0f, 0.0f};
 GLuint quadIndices[] = {0, 1, 2, 0, 2, 3};
```



Program structure:

- Set up OpenGL
- Upload data to texture
- Load shaders from file and compile
- Draw quad on screen (of off screen) using OpenGL
- Data is computed by the fragment shader, per pixel
- Output can be downloaded as image data

Examples...

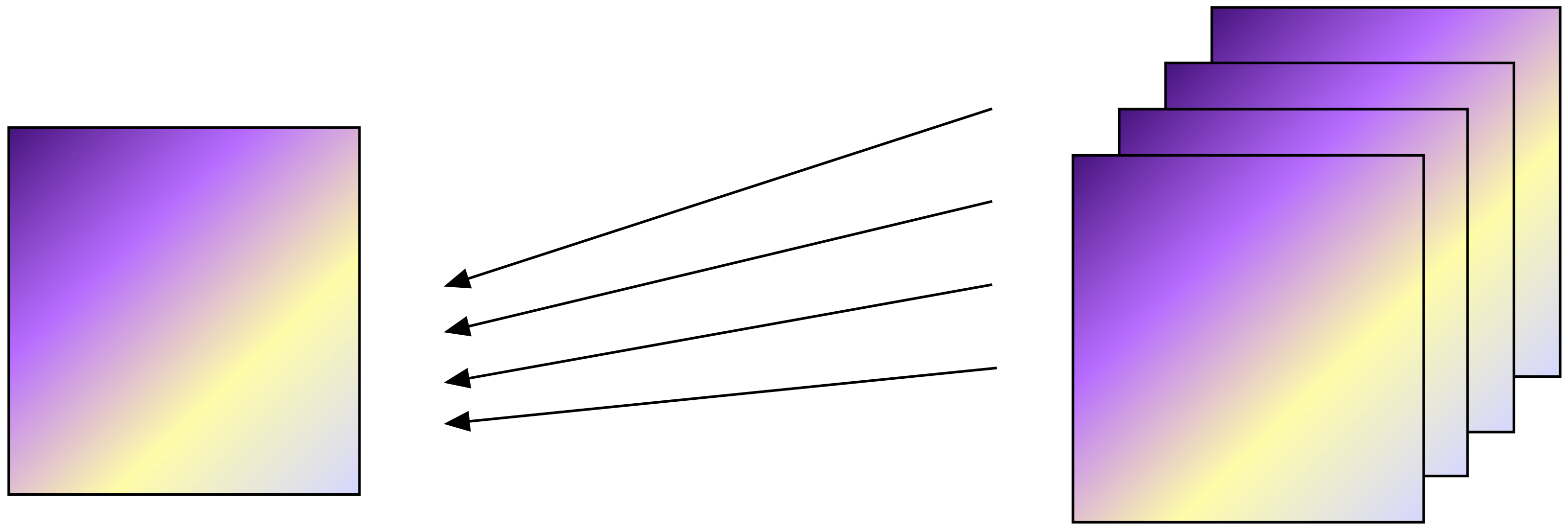
Feedback

We must be able to pass output from one operation as input of the next!

Solution: Render to texture, ”framebuffer objects”, create a texture used as input for a later stage

"Ping-pong"-ing

The kernel reads from one or more texture, writes into the frame buffer



Using "framebuffer objects" the output image can be a texture

Input data is a number of textures. Limited by the number of texturing units available.

Filtering, convolution

Common problem, highly suited for shaders.

All kinds of linear filters:

- **Low-pass filtering (smoothing)**
- **Gradient, embossing**

Must be done by gather operations, not scatter!

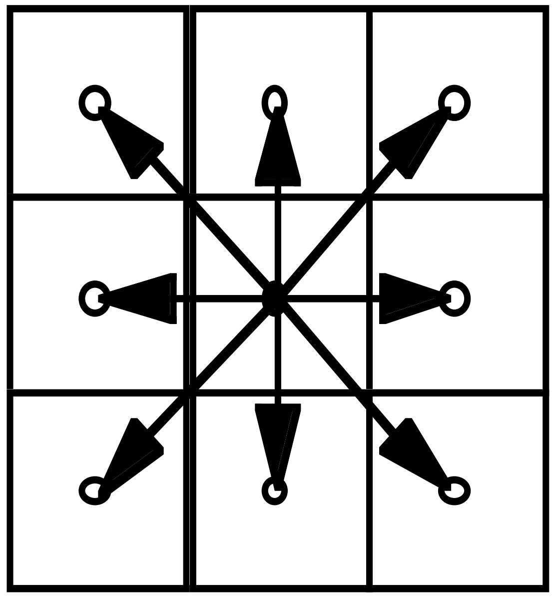
Example: high pass filter

```
#version 150

out vec4 outColor;

in vec2 texCoord;
uniform sampler2D tex;

void main(void)
{
  float h, v;
  const float offset = 1.0/512.0;

  vec4 c  = texture(tex, texCoord);
  vec4 r  = texture(tex, texCoord + vec2( offset,  0.0));
  vec4 l  = texture(tex, texCoord + vec2(-offset,  0.0));
  vec4 u  = texture(tex, texCoord + vec2(    0.0, offset));
  vec4 d  = texture(tex, texCoord + vec2(    0.0,-offset));
  outColor = (-4.0*c + r + l + u + d);
}
```

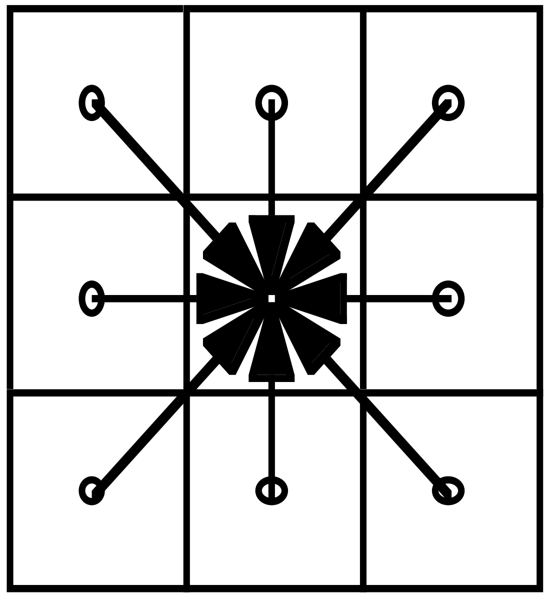
	1	
1	-4	1
	1	

More graphics heritage: Index data by steps of 1/size, not 1!

Scatter vs gather



Scatter



Gather

Shaders give output for *one* pixel -> gather only!

How about CUDA/OpenCL?

Scatter vs gather: You usually prefer gather. Less synchronization! (Remember, synchronization comes for a cost!)

Separable filters: Optimization just as valid for all techniques! (But particularly common in shaders, for images.)

Reduction, sorting

Same methods as I have mentioned before.

Bitonic sort suitable.

Reduction by tree structure.

In the past: Fixed output per thread. This is getting less fixed.

- **Write to texture possible.**
- **Synchronization supported.**

Conclusions:

- **Shader-based GPGPU is not dead, it is just not hyped**

Superior compatibility and ease of installation makes it highly interesting for the forseeable future. Especially suitable for all image-related problems.

- **How to do GPGPU with shaders**

FBOs, Ping-ponging, algorithms, special considerations.

But stay tuned for Compute Shaders to change things...