Data parallelism and graphics

Interactivity

- Interactive rendering applications
  - Video game, architectural modeling, 3D visualization (as opposed to offline rendering, e.g., movies)
- Requirement: Must be interactive
- How fast is enough?
- Monitors refresh at 60 Hertz (60 Hz)
  - Renders 60 images in each second
- Display 6 frames per second (6 FPS)
  - Each image will be repeated 10 times!
  - Testing shows this is the bare minimum

http://www.realtimerendering.com/udacity/?load=demo/unit1-fps.js

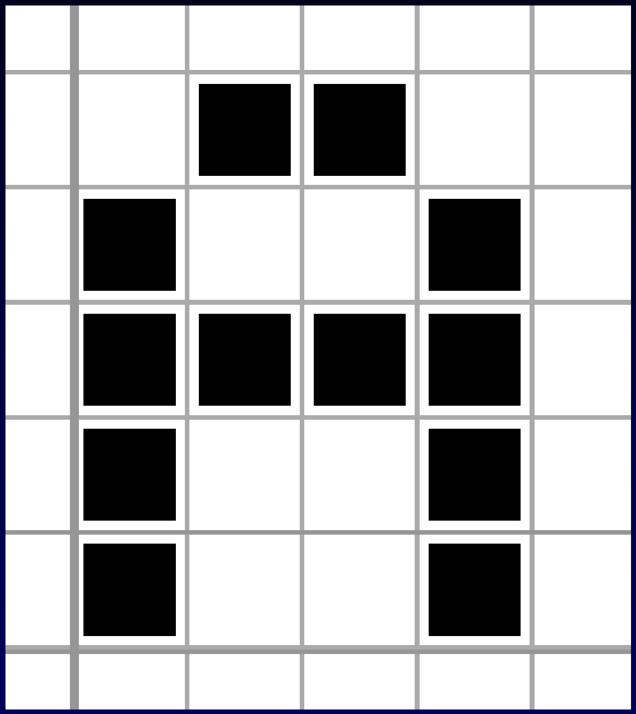
Real-time graphics

- Demands are often much higher
- 30-60 FPS are common thresholds
  - Tied to the 60Hz *refresh rate*
    Each image being displayed once or twice
  - 60 FPS is upper limit
- If game takes 100ms to generate the frame
  - 10 FPS result
  - Display will still refresh at 60Hz but it will overwrite with same image

Screen resolution

- Pixel: A point in your screen
- Holds an RGB color value

- Higher screen resolution
  leads to larger number of pixels
  leads to better image
  leads to lower higher processing cost
  leads to lower FPS!



Pixel throughout

- A screen resolution of 1024 by 768 (modest!)
- A frame rate of 60 FPS
- How many pixels are we drawing per second?
  - A: 47,185,920
  - In each second the GPU has to calculate the color of a pixel over 47 million times!
  - A lot of computation
  - And that’s a lower bound
    - Objects can overlap and be drawn on top of others

How can GPUs be so fast?!

A: Parallel processing

Data Parallelism for Graphics

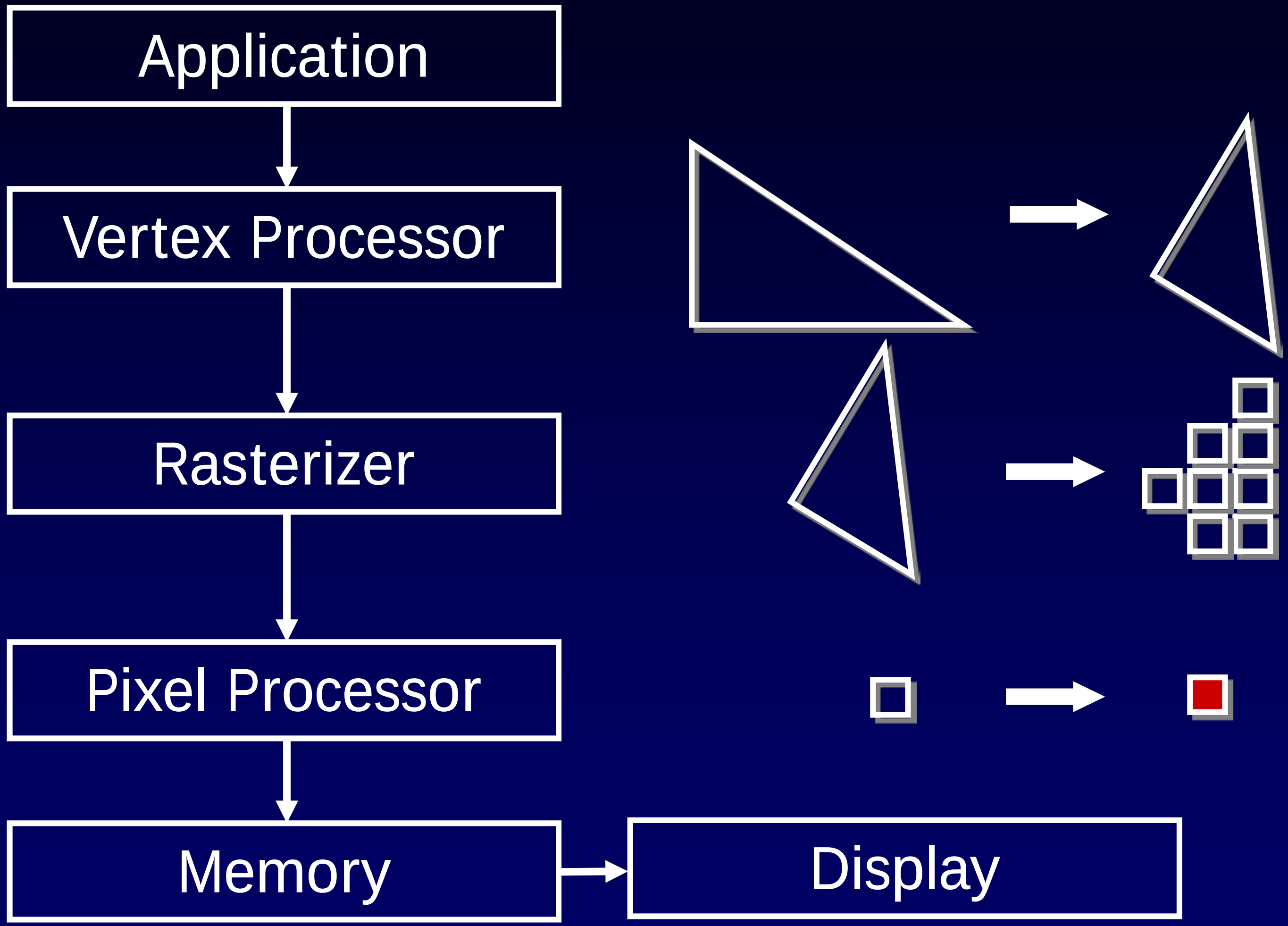
- GPUs are designed for graphics
  - Most popular consumer need for highly parallel computation
- GPUs process *independent* vertices & pixels
  - Temporary registers are zeroed
  - No shared or static data (not in graphics APIs)
  - No read-modify-write buffers
- Data-parallel processing
  - GPUs architecture is ALU-heavy
    - Multiple vertex & pixel pipelines

Today's GPUs

- Efficient architecture
- 1000+ parallel processors!
  - Working on vertices
  - Working on pixels
- SIMD processing
  - Must execute same code
  - Ok for graphics applications

- Improving at faster rate than CPUs

GPU processing at a glance

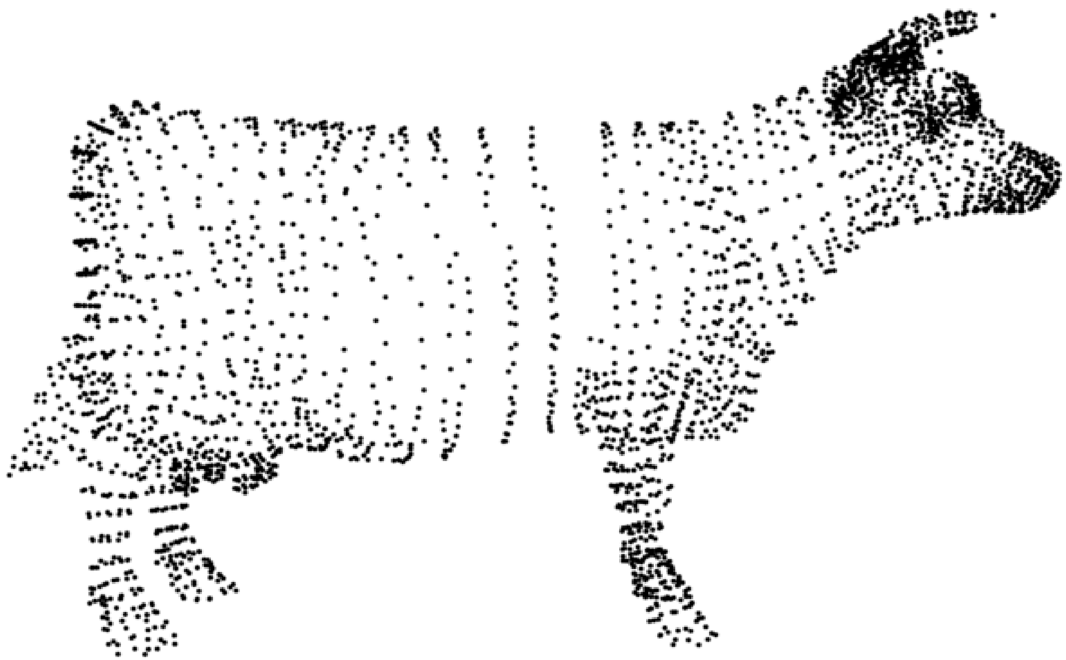


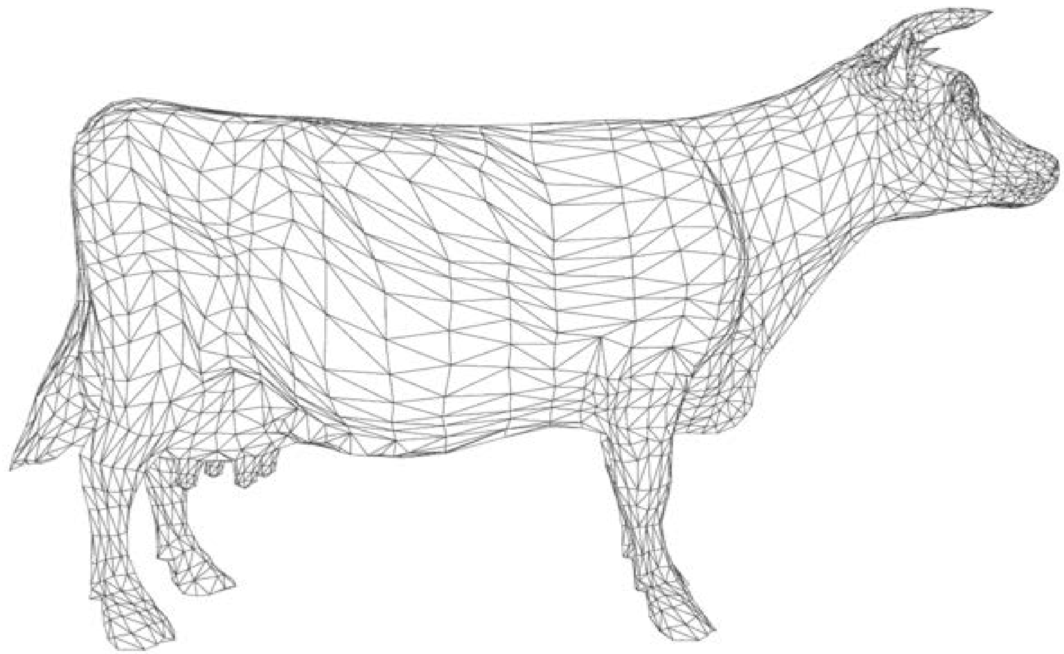
Vertex Processor

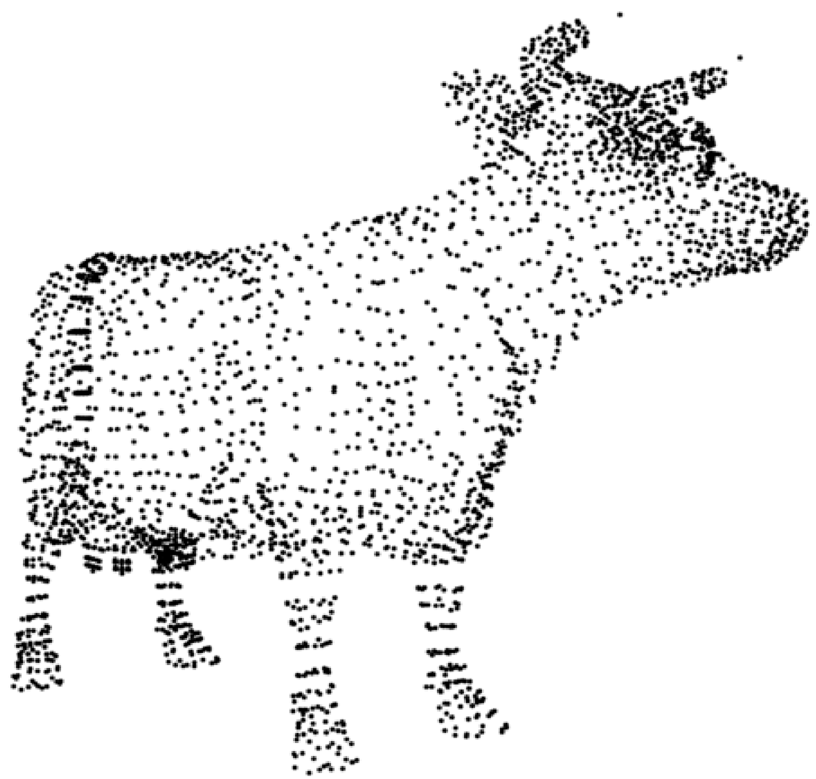
From object coordinates to screen coordinates

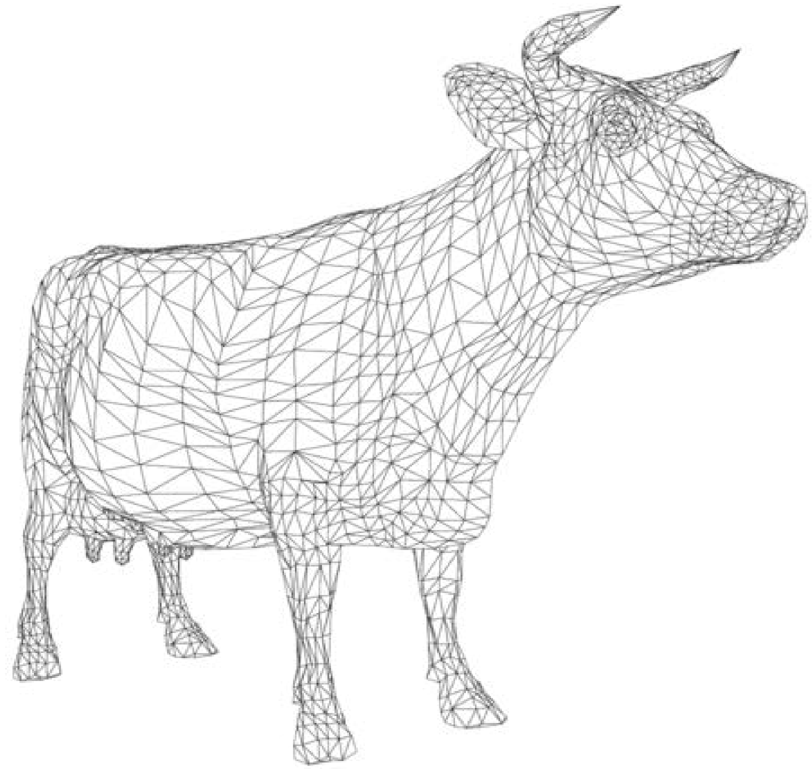
Geometry

- Composed of two parts
  - Vertex data in a vertex array/buffer
  - Primitives in an index/element array/buffer
- Vertices must include a position
  - Color and other optional parameters
- Primitives reference one or more vertices
  - Triangles, lines, points
- Geometry transformations act on vertices
  - And as a consequence change primitives too









Vertex transformation

Object or Model Coordinates → World Coordinates → Eye or Camera Coordinates → Clip Space → Device Coordinates → Window or Screen Coordinates

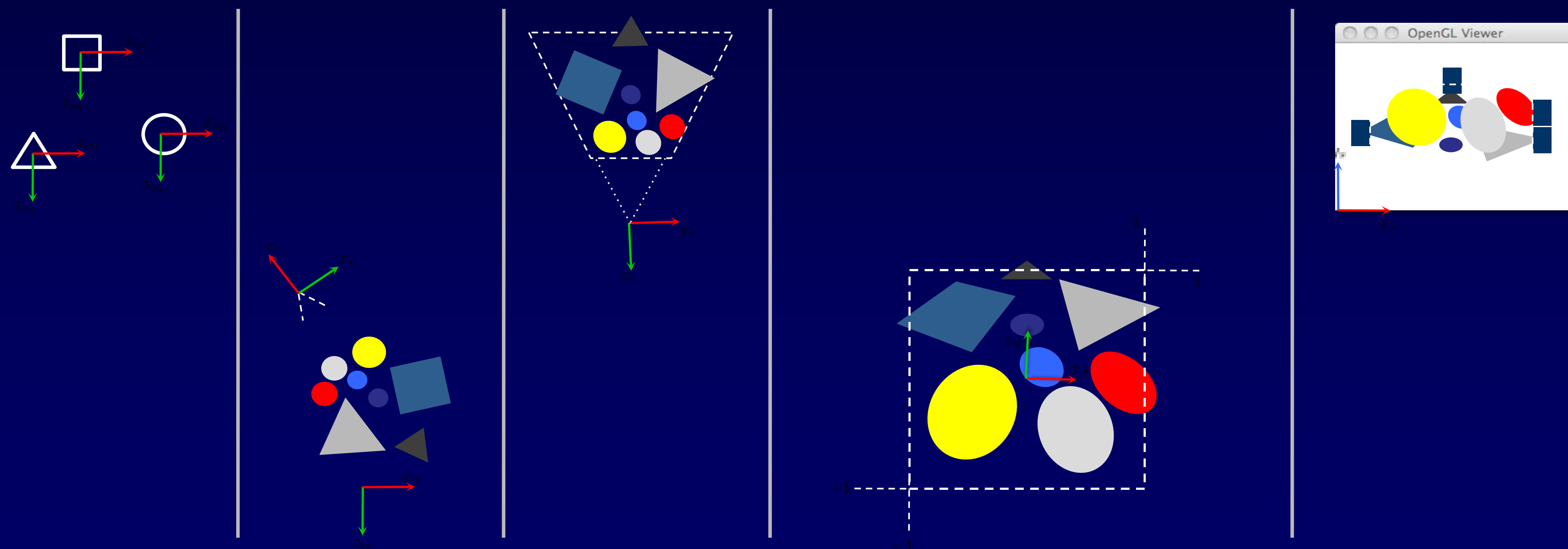
Modeling transformation

Viewing transformation

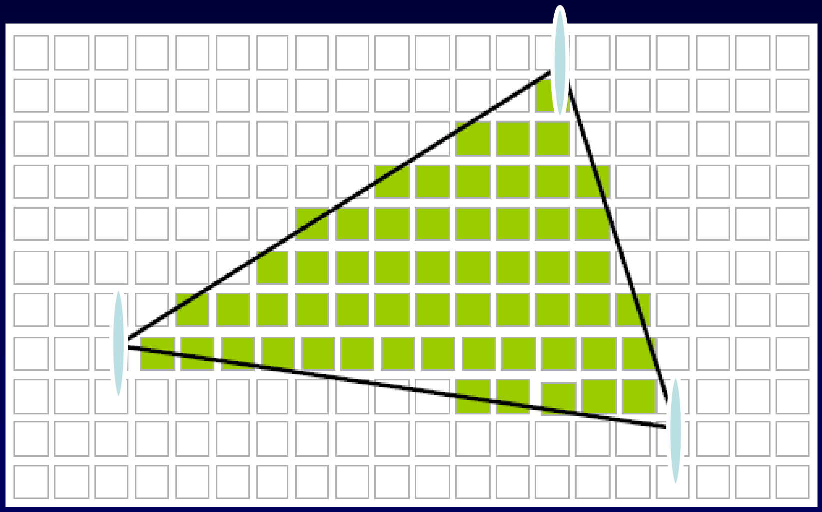
Projection transformation

Perspective division

Viewport transformation



Rasterizer

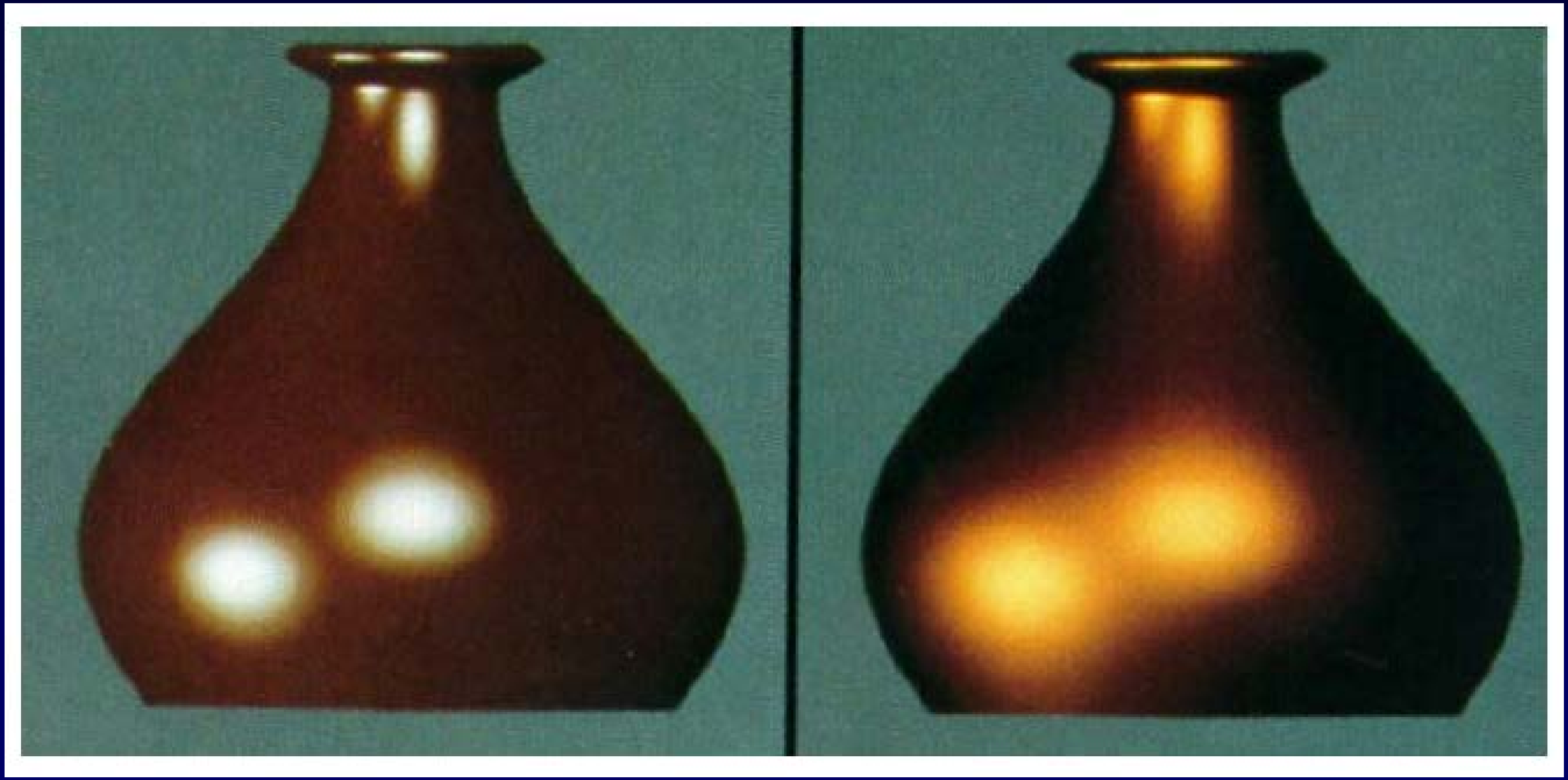


From vertices and primitives to pixels

Pixel processor

Coloring the geometry

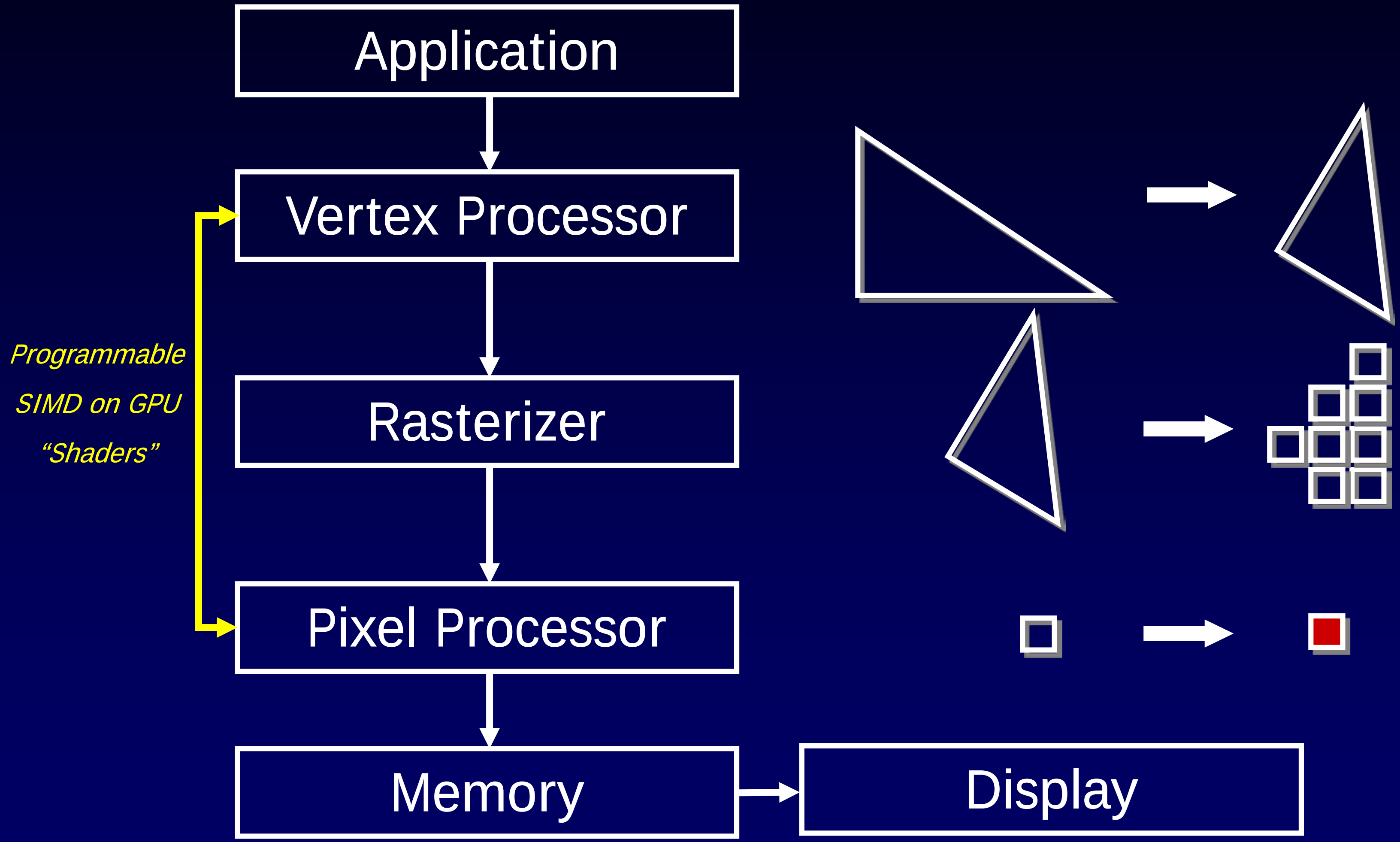
Use a popular illumination model



Pixel processor

Applying texture images

Programmability



Unified rendering pipeline

- No longer has dedicated vertex and pixel processors
- All programmable computation shares the same set of processors
- A complex scheduler determines their allocation when rendering a frame

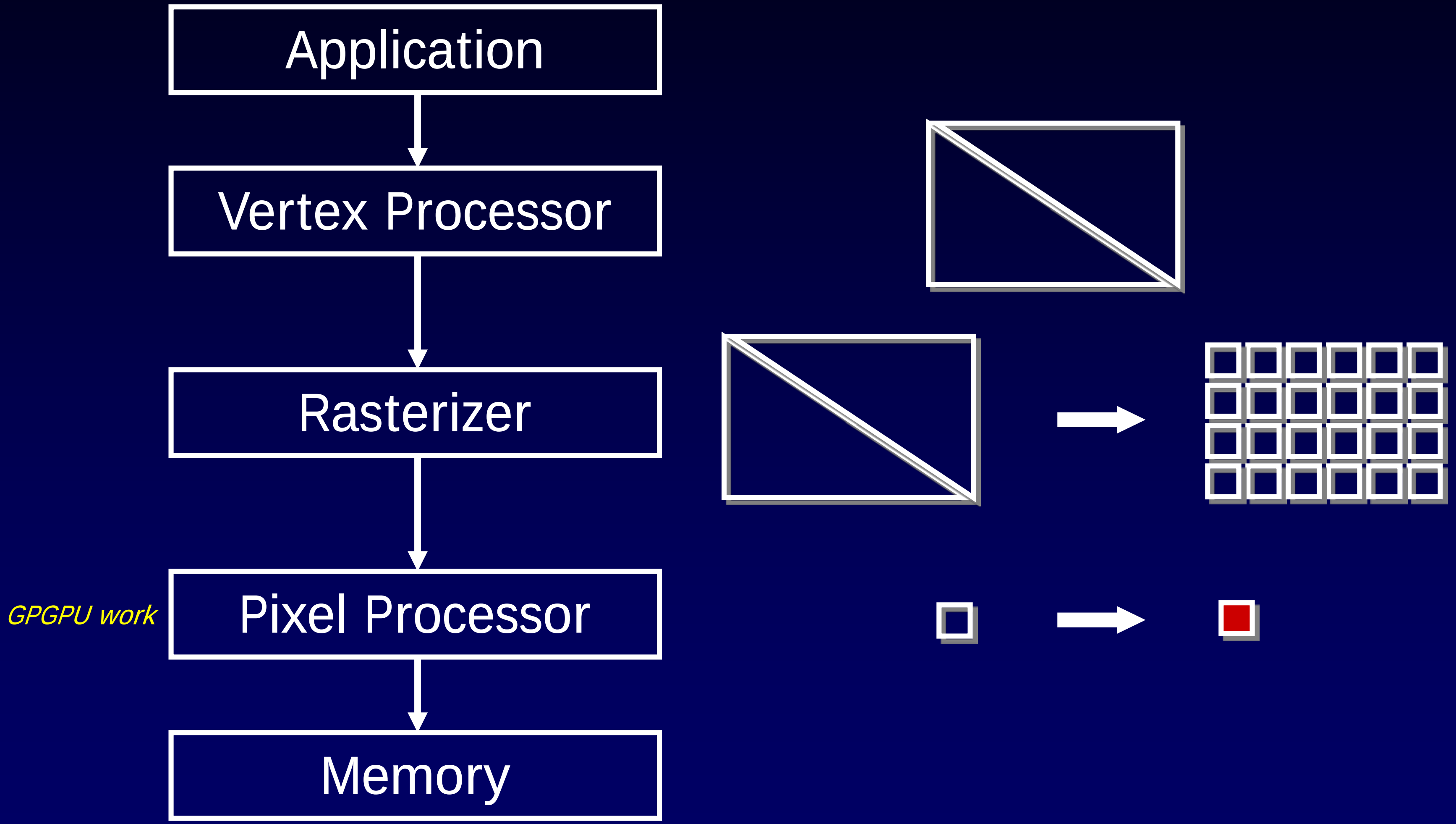
- GPU “shaders” processes vertices and pixels

GPU Shading languages

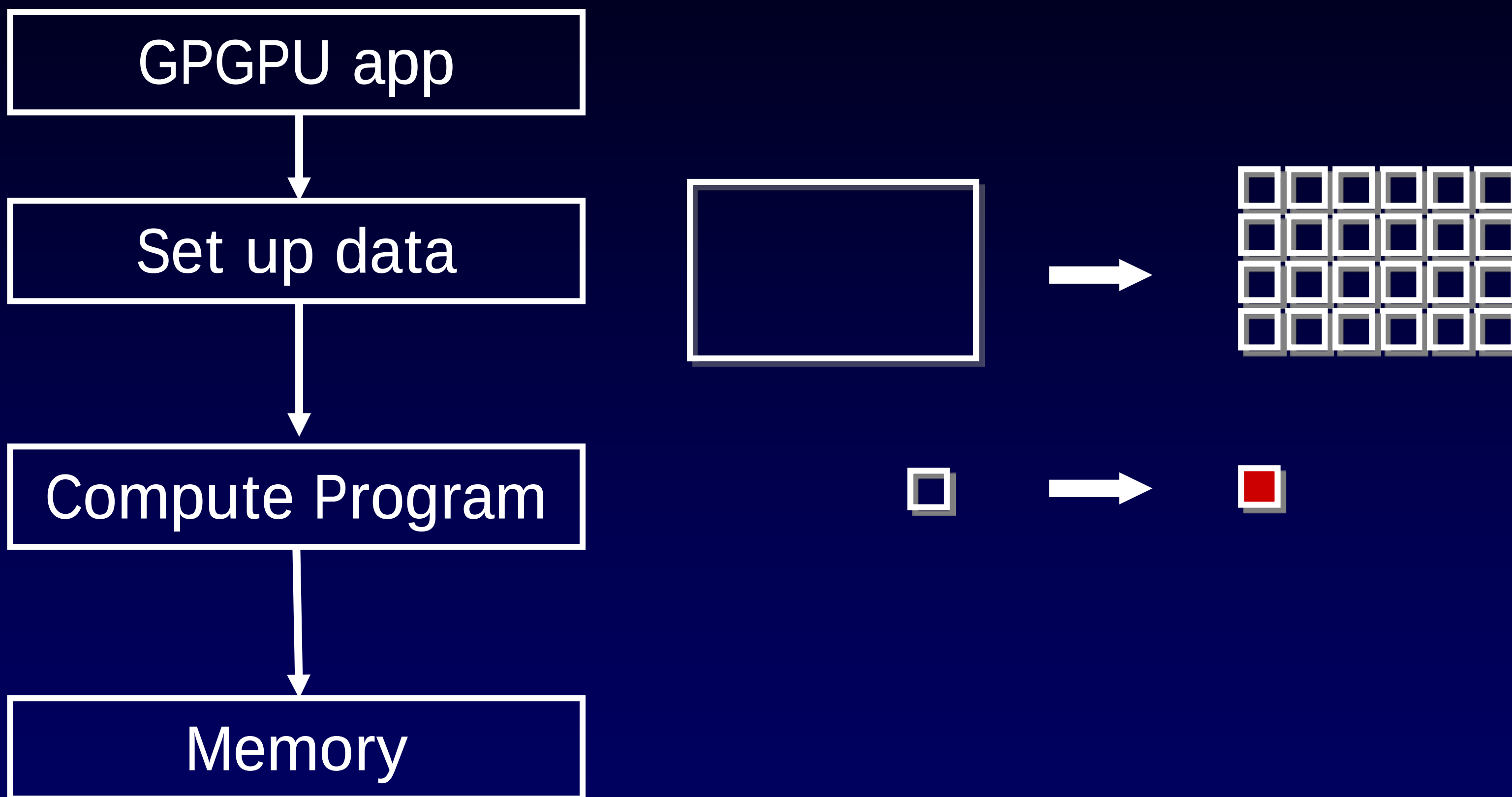
- GLSL (OpenGL standard library)
- HLSL (Microsoft DirectX)
- CG (Nvidia proprietary)

- All C-like languages
- 2D-4D vector and matrix data types
  - Enough to represent geometry and color

GPGPU usage scenario

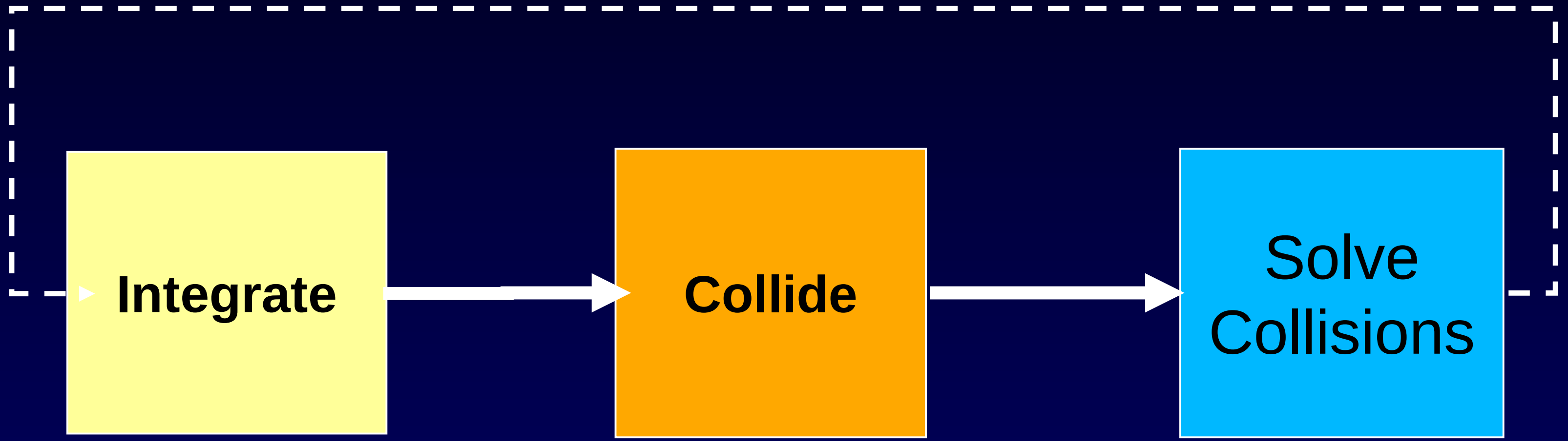


GPGPU Programming Model



Physical simulations on the GPU

Physics is a data parallel task



100%
data parallel

80%
data parallel
(coarse + fine)

95%
data parallel
(dependency)

Mapping concepts to GPU

- Properties:
  - Algorithm must be data-parallel
  - Computations should ideally be local

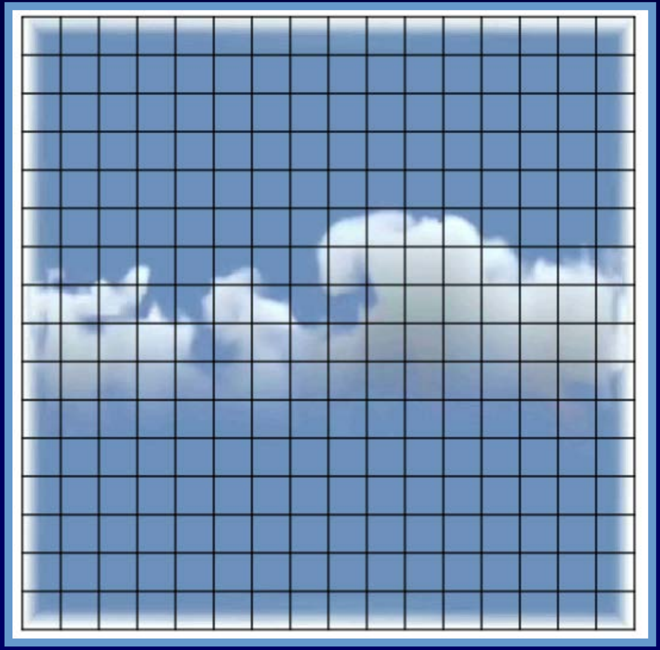
- Now, how do we map it?

Data Streams & Kernels

- Streams
  - Collection of records requiring similar computation
    - Vertex positions, Voxels, etc.
  - Provide data parallelism
- Kernels
  - Functions applied to each element in stream
    - transforms, PDE, …
  - Few dependencies between stream elements
    - Encourage high Arithmetic Intensity

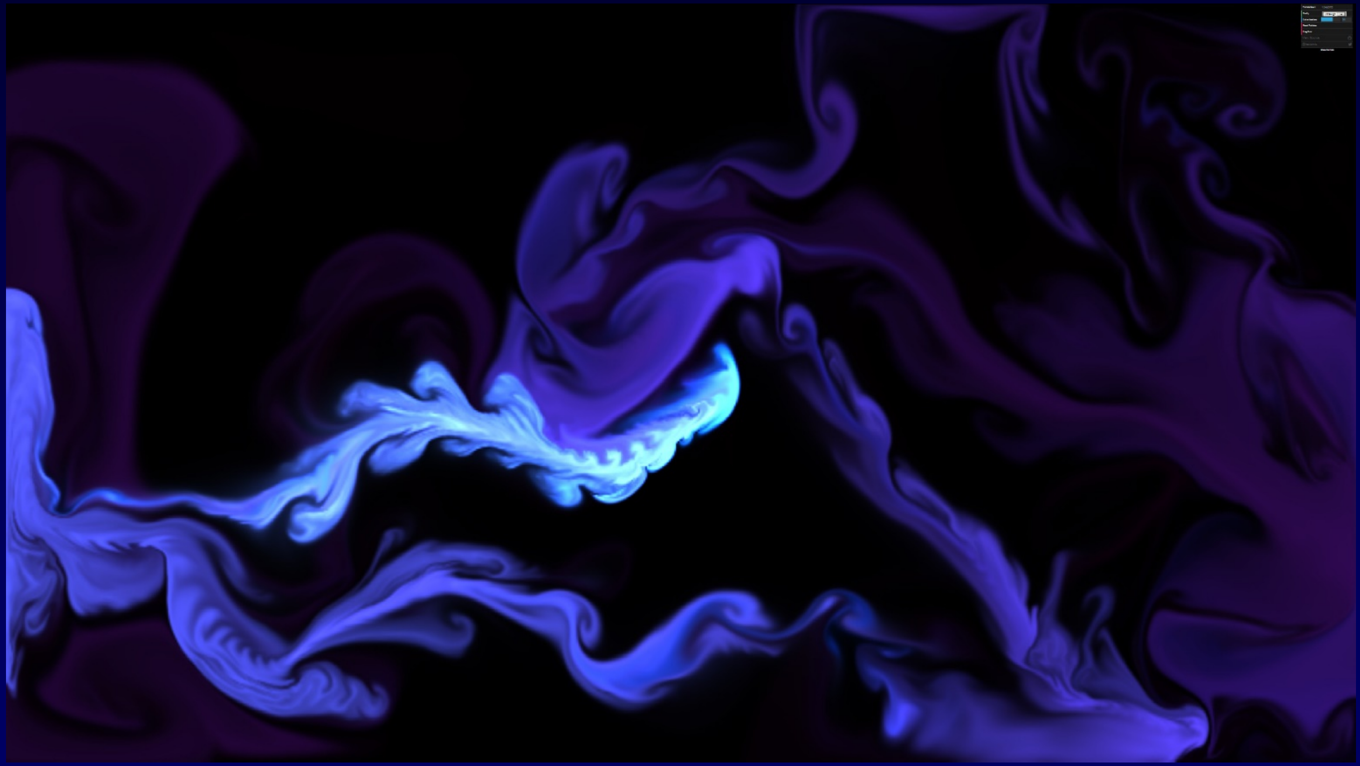
Example: Fluid simulation

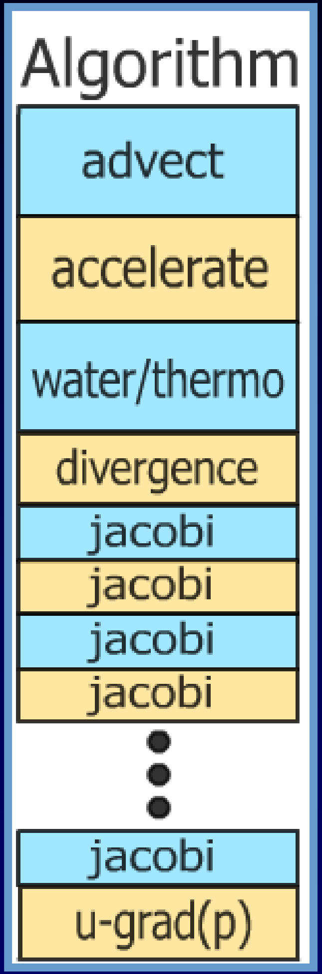
- Uses a simulation grid
- Common GPGPU rendering pipeline computation style
  - Textures represent computational grids = streams
- Many computations map to grids
  - Matrix algebra
  - Image & Volume processing
  - Global Illumination
  - Physically-based simulation
- Non-grid streams can be mapped to grids



Fluid simulation

- Navier stokes equations for fluid simulation





Kernels

advect

CPU

```cpp
for (int j = 1; j < height - 1; ++j)
{
  for (int i = 1; i < width - 1; ++i)
  {
    // get velocity at this cell
    Vec2f v = grid(x, y);

    // trace backwards along velocity field
    float x = (i - (v.x * timestep / dx));
    float y = (j - (v.y * timestep / dy));

    grid(x,y) = grid.bilerp(x, y);
  }
}
```

C++

GPU

```
void advect(float2        uv   : WPOS,
        out float4        xNew : COLOR,

    uniform float         dt,  // timestep
    uniform float         dx,  // grid scale
    uniform samplerRECT u,  // velocity
    uniform samplerRECT x)  // state
{
  // trace backwards along velocity field
  float2 pos = ub - dt * f2texRECT(u, uv) / dx;

  xNew = f4texRECTbilerp(x, pos);
}
```

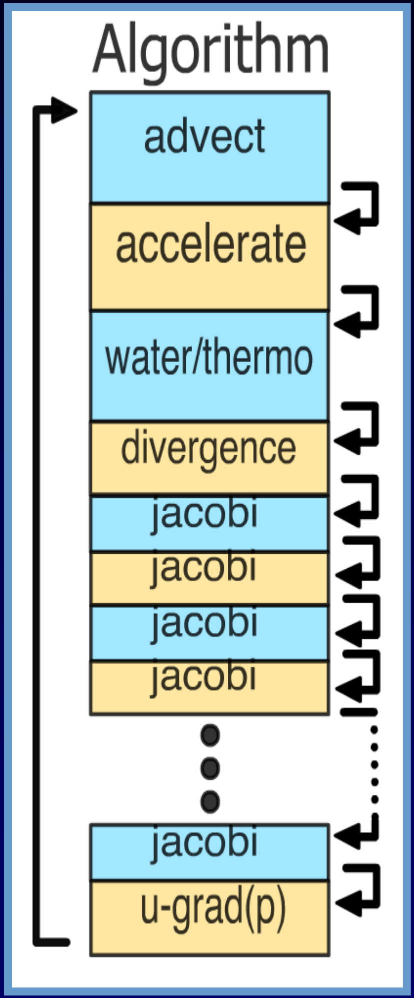
Cg

Kernel / loop body / algorithm step = Fragment Program

Feedback

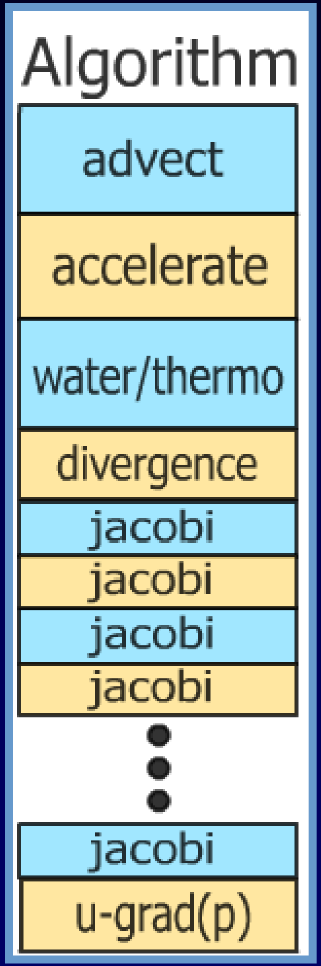
- Each algorithm step depends on the results of previous steps

- Each time step depends on the results of the previous time step



GPU Simulation

- Algorithm steps are pixel shaders
  - Computational *kernels*
- Current state is stored in textures
- Feedback via render to texture



Invoking Computation

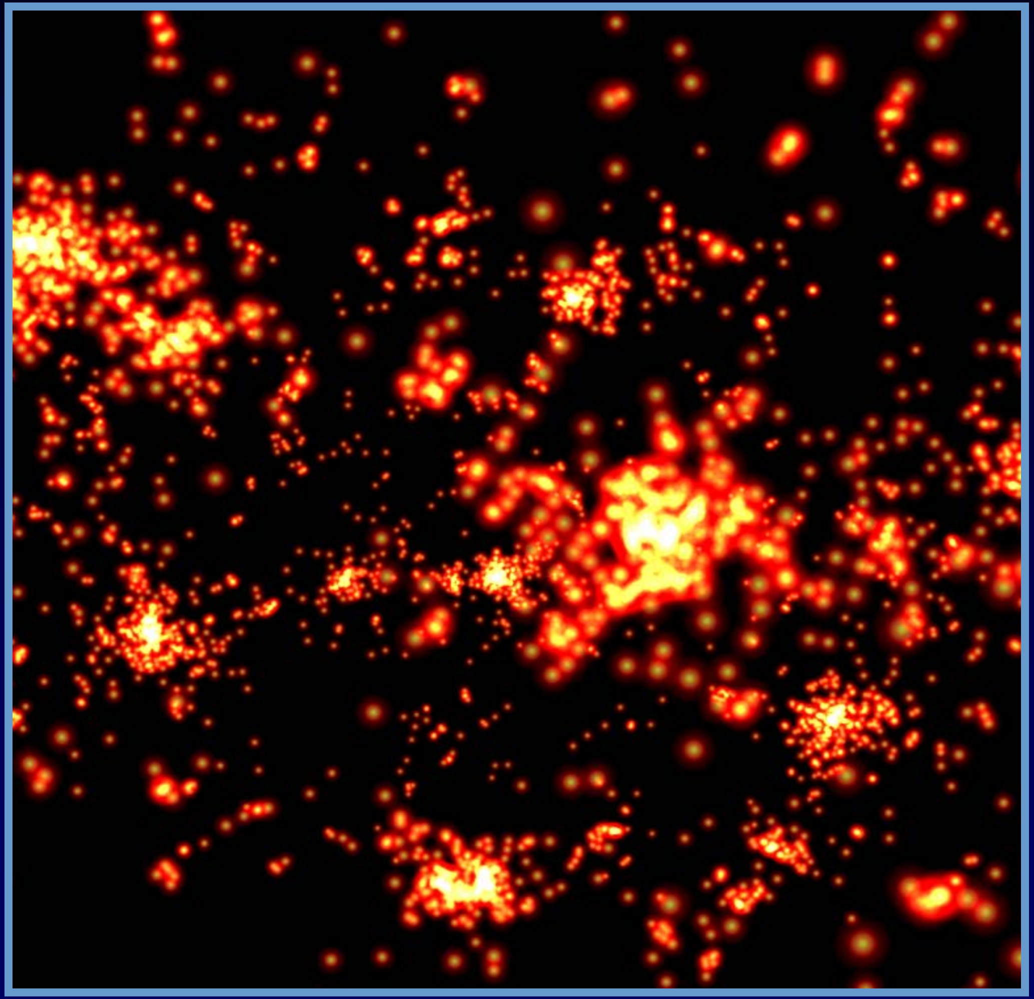
- Full screen quadrilateral invokes computation
- So, rasterization = kernel invocation

Examples:

https://www.youtube.com/watch?v=7EOjAdmURY4

https://haxiomic.github.io/GPU-Fluid-Experiments/html5/?q=Medium

Example: N-Body Simulation

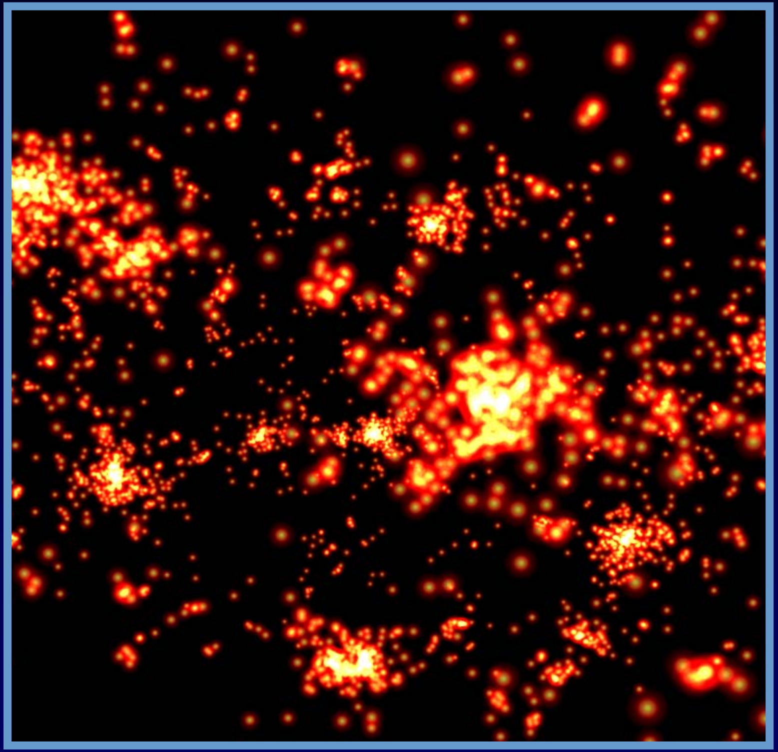


- https://www.youtube.com/watch?v=CPuVfiWLlHI

GPU computation

- Brute force ☹
- N = 8192 bodies
- N^2 gravity computations

- 64M force comps. / frame
- ~25 flops per force

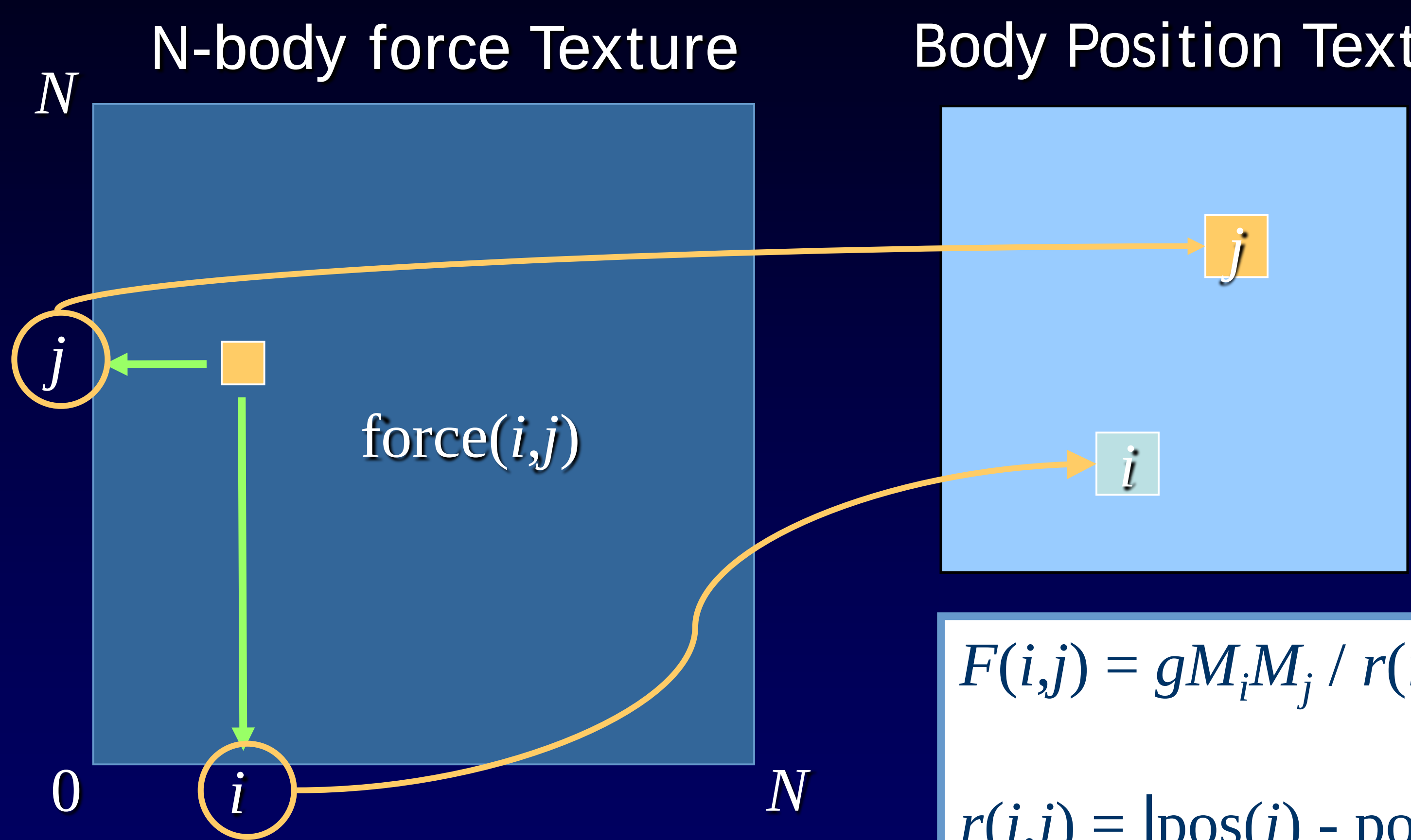


Nyland, Harris, Prins, GP^2 2004 poster

Computing Gravitational Forces

- Each body attracts all other bodies
  - *N* bodies, so N^2 forces
- Draw into an *N*x*N* buffer
  - Pixel (*i*,*j*) computes force between *i* and *j*
  - Very simple fragment program
    - More than 8192 bodies makes it trickier
      - Limited by max buffer size…

Computing Gravitational Forces



$$F(i,j) = gM_iM_j / r(i,j)^2,$$

$$r(i,j) = |\mathrm{pos}(i) - \mathrm{pos}(j)|$$

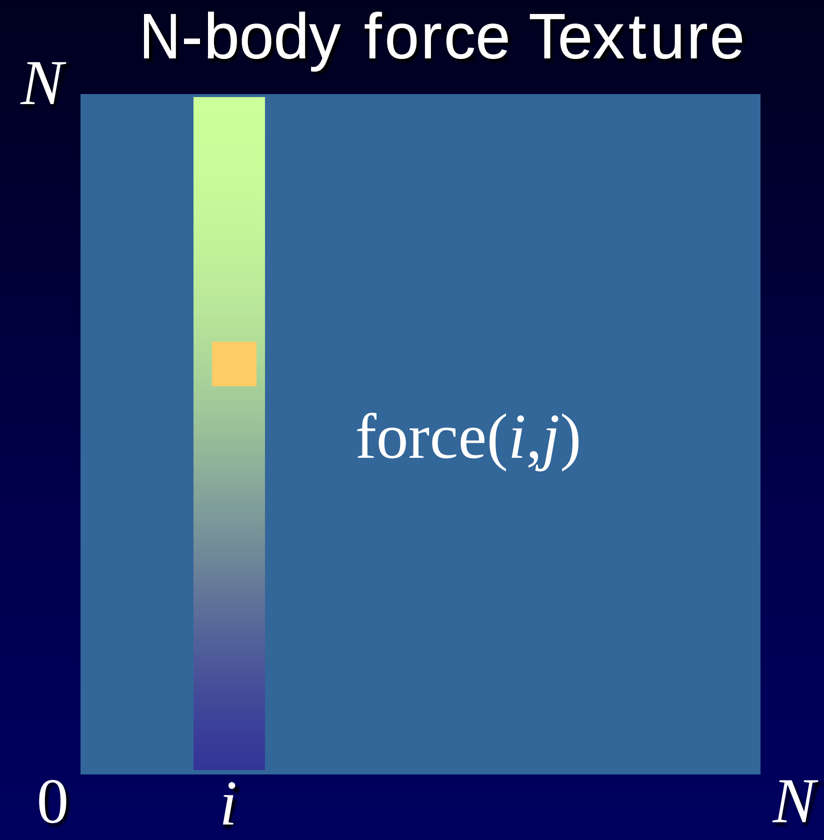
Force is proportional to the inverse square of the distance between bodies

Computing Gravitational Forces

```
float4 force(float2 ij       : WPOS,
     uniform sampler2D pos) : COLOR0
{
   // Pos texture is 2D, not 1D, so we need to
   // convert body index into 2D coords for pos tex
   float4 iCoords = getBodyCoords(ij);
   float4 iPosMass = texture2D(pos, iCoords.xy);
   float4 jPosMass = texture2D(pos, iCoords.zw);
   float3 dir = iPos.xyz - jPos.xyz;
   float r2 = dot(dir, dir);
    dir = normalize(dir);
   return dir * g * iPosMass.w * jPosMass.w / r2;
}
```

Computing Total Force

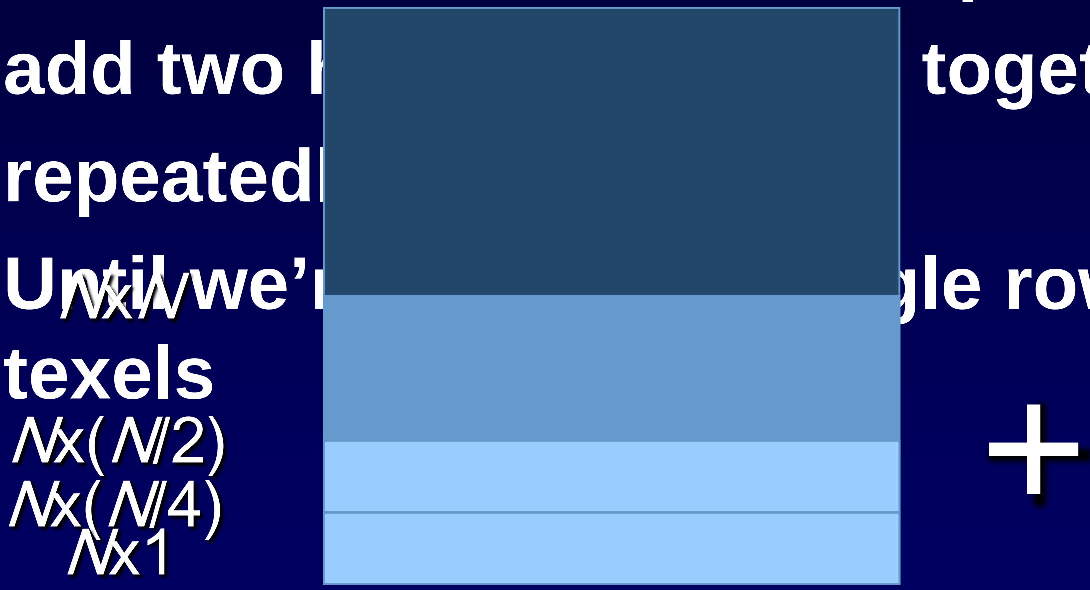
- Have: array of (i,j) forces
- Need: total force on each particle i
  - Sum of each column of the force array
- Can do all N columns in parallel



Parallel Reduction

Parallel Reductions

- 1D parallel reduction:
  - sum N columns or rows in parallel
  - add two h together
  - repeatedl
  - Until we'r gle row of texels



Requires $\log_2 N$ steps

Update Positions and Velocities

- Now we have a 1-D array of total forces
  - One per body
- Update Velocity
  - $u(i,t+dt) = u(i,t) + F_{total}(i) * dt$
  - Simple pixel shader reads previous velocity and force textures, creates new velocity texture
- Update Position
  - $x(i, t+dt) = x(i,t) + u(i,t) * dt$
  - Simple pixel shader reads previous position and velocity textures, creates new position texture

PhysX

- Nvidia middle-ware SDK
- Works on any CUDA-ready GPU
- Supports rigid body dynamics, soft body dynamics, vehicle dynamics, particles, volumetric fluid simulation, and cloth simulation

- https://www.youtube.com/watch?v=6vipmar3wS4

- https://www.youtube.com/watch?v=pEX13W-IuLA

Optimize by force: Dual and Triple GPU Physics

Second (or third) GPU can be used for graphics or physics simulation, or share the primary card