CS4621/5621 Fall 2015

Particle Systems and Compute Shaders

Professor: Kavita Bala
Instructor: Nicolas Savva

with slides from Balazs Kovacs, Eston Schweickart, Daniel Schroeder, Jiang Huang and Pramook Khungurn over the years.

Announcements

- PPA2 demos (Monday Nov 9)
- PPA3 Particle system (out Monday)
- Final project proposals (due Nov 11)

Today

- Particle Systems
- PPA3 Overview
- Compute Shaders

Particle System Resources

Physically Based Modeling Notes:

http://www.pixar.com/companyinfo/research/pbm2001/index.html

Differential Equation Basics

Particle Dynamics

Rigid Body Dynamics

Particle System

Reeves, SIGGRAPH 83

Physically-Based Animation

Model with physical attributes

Ordinary Differential Equation (ODE)

- Have function f() for derivative of x
  - Mass, moment of inertia, elasticity, etc.
- Derive differential equations by applying Newtonian physics
- Specify initial conditions: position, velocity
- Specify external forces (maybe keyframe)
- Solve for motion

Ordinary Differential Equation (ODE)

Have function f for derivative of x

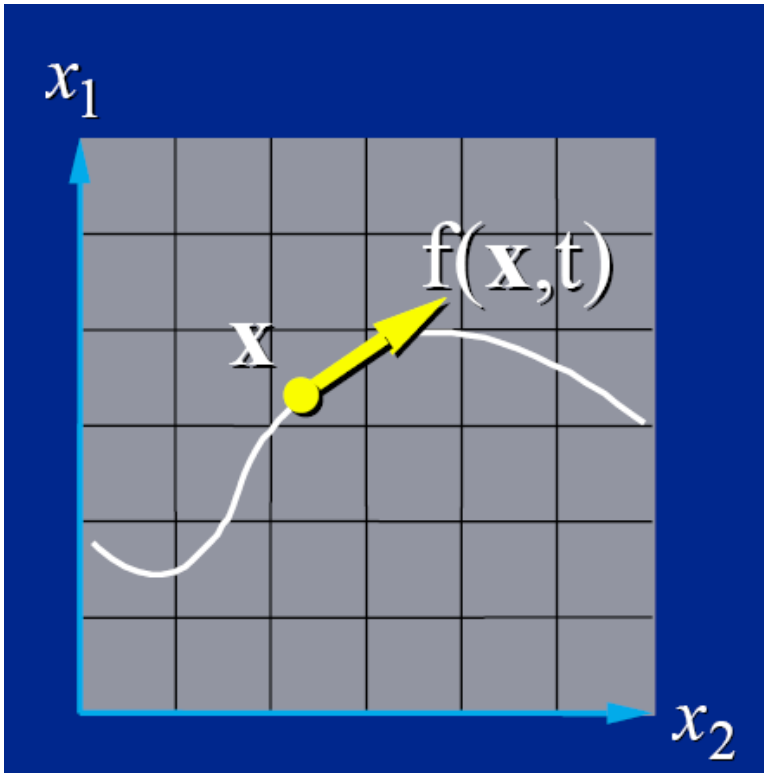
$$\dot{x} = f(x(t))$$

x is state

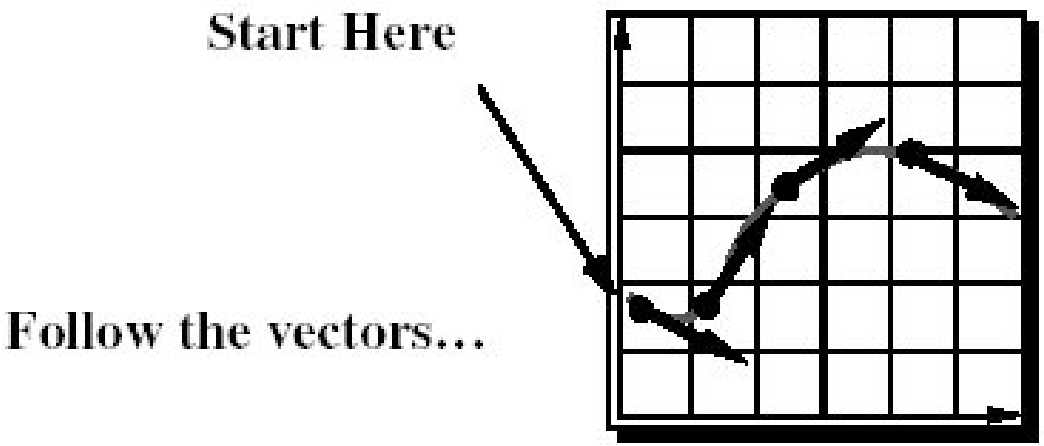
x is a moving point

f is known

f is its velocity



Euler Method



Move a little step along the derivative

to the next position where h is the step size

$$x(t_0 + h) = x(t_0) + h\dot{x}(t_0)$$

Beyond Euler

- Euler is a first order method
- We can improve the accuracy of each step

if we extend to second derivatives

- Consider Taylor series expansion:

$$x(t_0 + h) = x(t_0) + h\dot{x}(t_0) + \frac{h^2}{2!}\ddot{x}(t_0) + \frac{h^3}{3!}\dddot{x}(t_0) + \ldots$$

Euler: only first 2 terms

– Error dominated by h^2

Use simpler methods if they get the job done

On to physics...

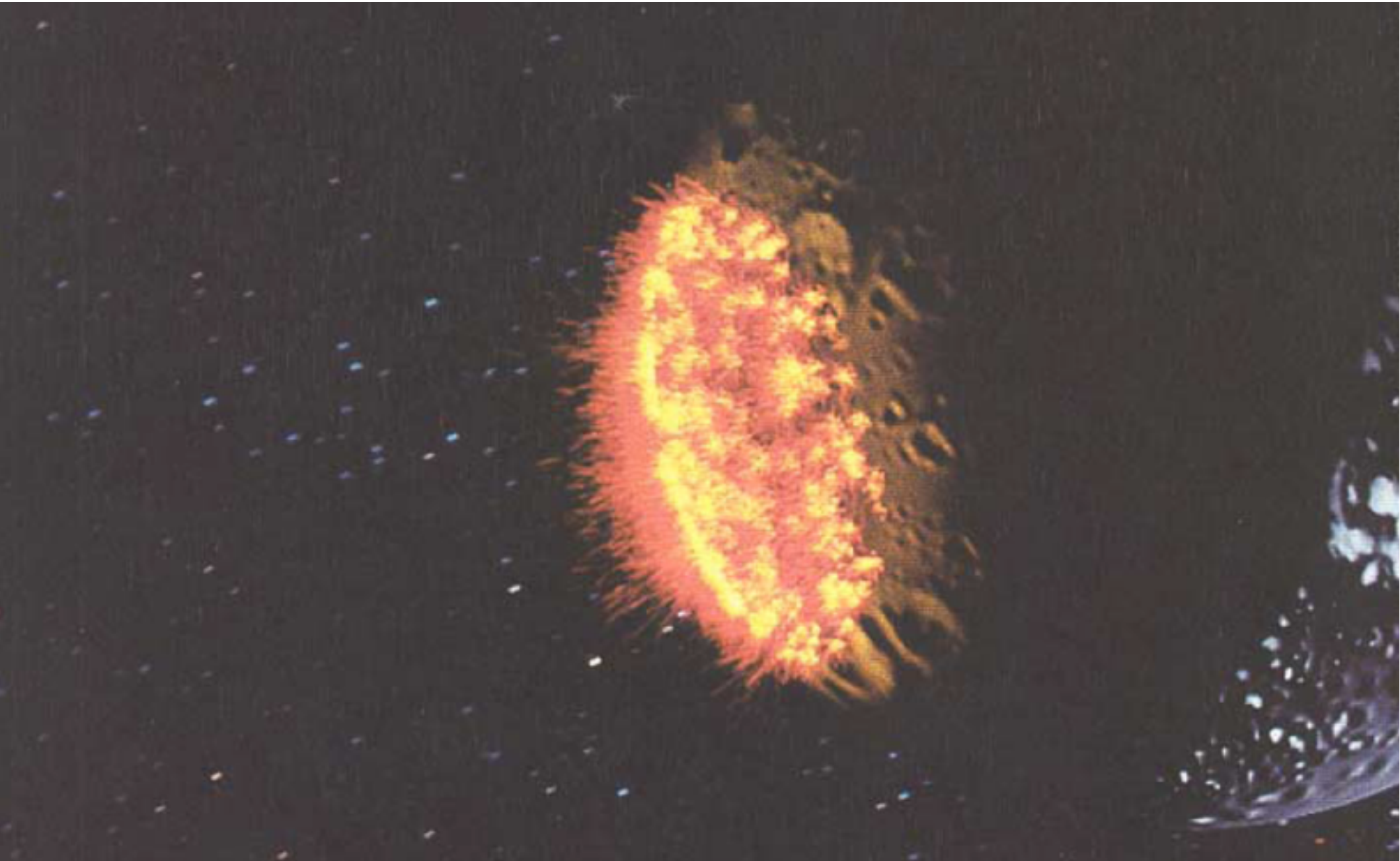
Particle Systems

Small objects, approximated as point masses

- Rotational motion is ignored
- They can be used in great numbers, without bogging down the system
- Can be used to simulate smoke, fire, clouds, and even cloth
- Reeves ’83: Star Trek II: Wrath of Khan

GENESIS EFFECT PLANET

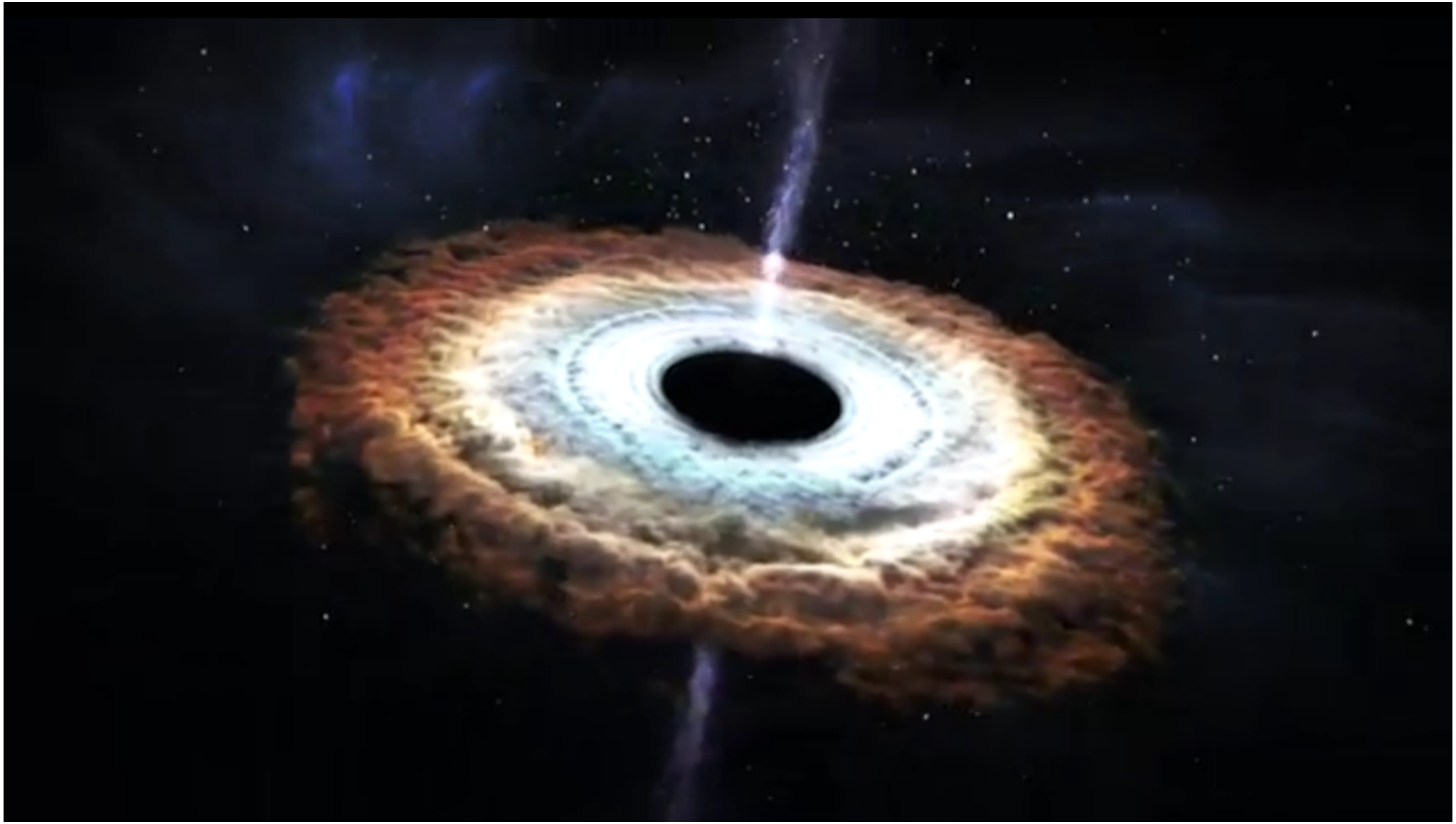
GENESIS EFFECT PLANET



Modern Particle System

Nvidia state of the art realtime particle system

Modern Particle System



Modern Particle System Demo

smoke simulation using CUDA

How do they work?

Have forces

- Want to find positions
- Earlier we looked at first order equation
  - Now, second order equation

How do they work?

Have forces

- Want to find positions
- Integrate the particle equations of motion
- Have a pair of ODEs

$\ddot{x}$ = a = F/m

$\dot{x}$ = v

System states

Every particle has a state s

- – s = (position, velocity, mass, age, color, ...)
- – p and v vary with time
- – Each p and v is a 3-vector

- The entire system state is S
  - – S = (p1, v1, p2, v2, p3, v3, ...)
  - – Can think of S as just a vector in 6n dimensions
- P, V, A, and F are 3n-vectors

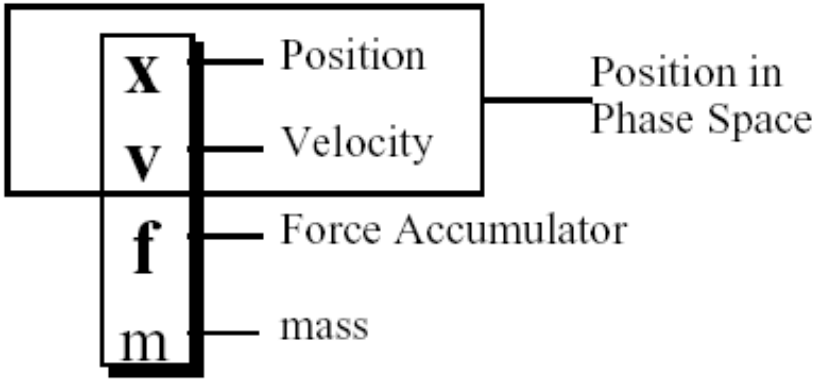
Simulation Loop

Particle loop

- – Initialize/Emit particles
- – Run integrator (evaluate derivatives)
- – Update particle states
- – Render
- – Repeat!

- Worry about memory
- – Don’t allocate/deallocate; recycle same memory block!

Particle Implementation

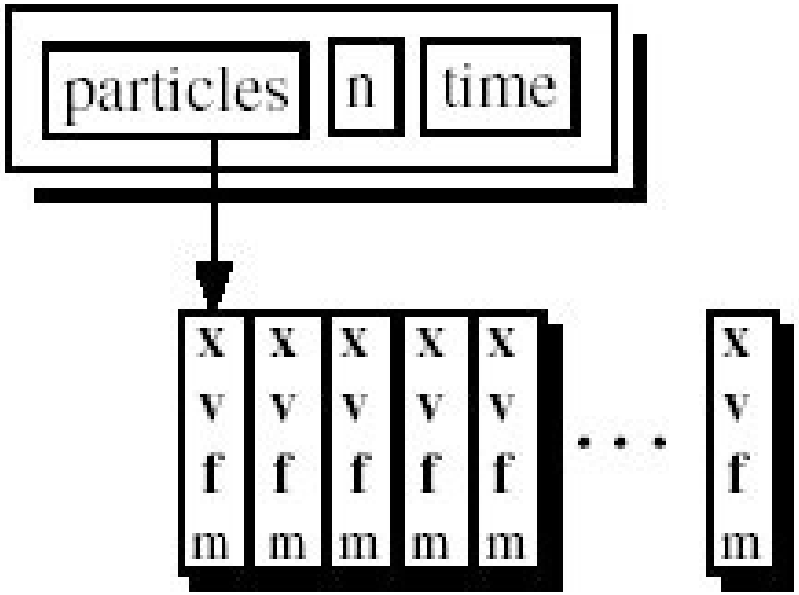


Each particle

represented by a

minimum of 10 values

Particle Structure



Additional values are possible

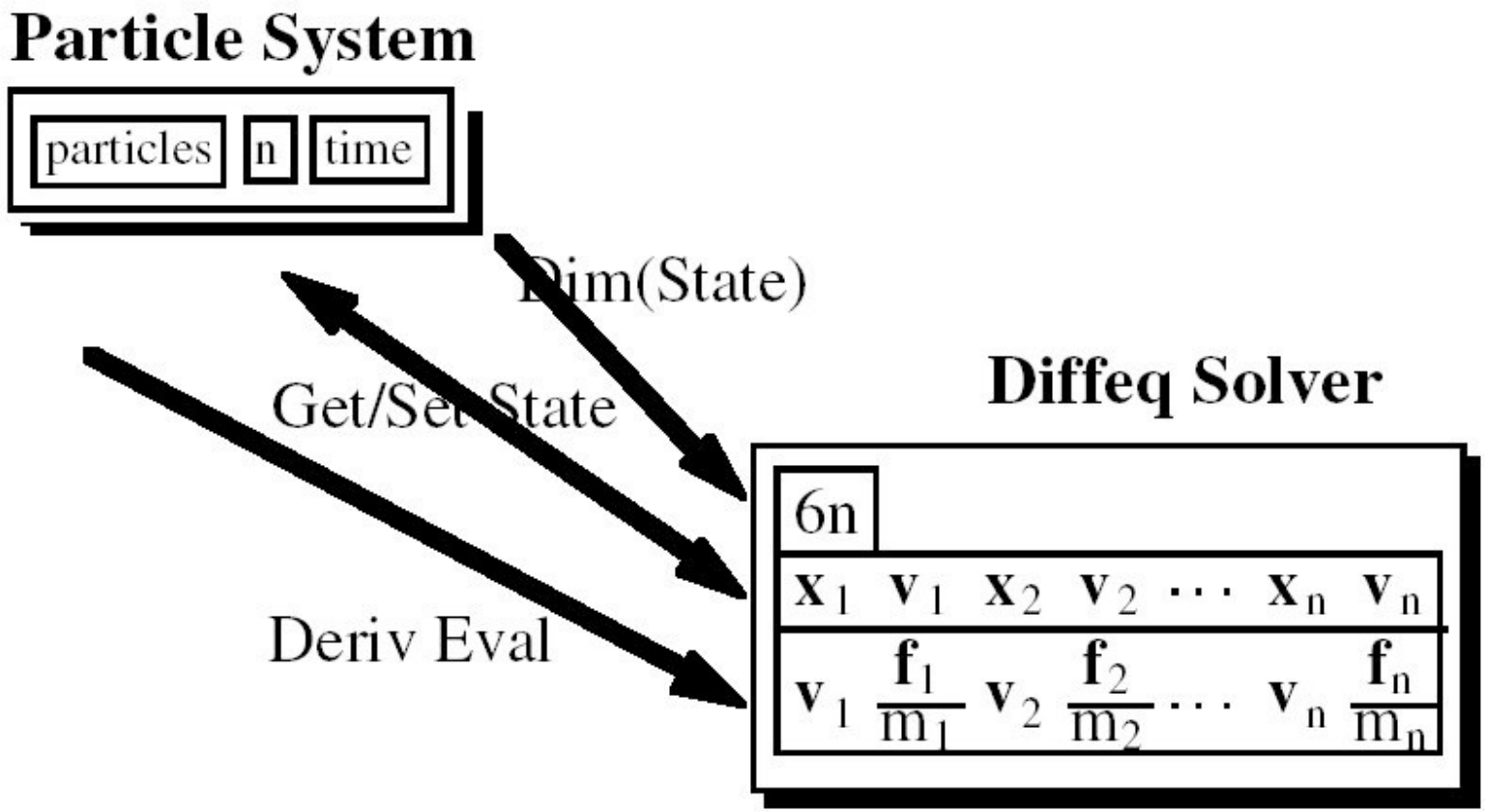
- – Electric charge
- – Color
- – Particle age

Integration

How do we implement an integrator?

- – Write a black-box that works on any f function
- – Takes an initial value f at time t, a function f′(value, time) and timestep h. Returns f(t+h)
- – The integrator can be completely separate from the particle representations
- – If your system has complex forces, repeated f′ evaluations become the bottleneck

Interface to Integrator



Particle system must allow the solver to read and write state and call the derivative function

In code

```
/* length of state derivative, and force vectors */
int ParticleDims(ParticleSystem p){
  return(6 * p->n);
};

/* gather state from the particles into dst */
int ParticleGetState(ParticleSystem p, float *dst){
  int i;
  for(i=0; i < p->n; i++){
    *(dst++) = p->p[i]->x[0];
    *(dst++) = p->p[i]->x[1];
    *(dst++) = p->p[i]->x[2];
    *(dst++) = p->p[i]->v[0];
    *(dst++) = p->p[i]->v[1];
    *(dst++) = p->p[i]->v[2];
  }
}
```

In code

```
/* scatter state from src into the particles */
int ParticleSetState(ParticleSystem p, float *src){
  int i;
  for(i=0; i < p->n; i++){
  p->p[i]->x[0] = *(src++);
  p->p[i]->x[1] = *(src++);
  p->p[i]->x[2] = *(src++);
  p->p[i]->v[0] = *(src++);
  p->p[i]->v[1] = *(src++);
  p->p[i]->v[2] = *(src++);
  }
}

/* calculate derivative, place in dst */
int ParticleDerivative(ParticleSystem p, float *dst){
 int i;
 Clear_Forces(p);   /* zero the force accumulators */
 Compute_Forces(p); /* magic force function */
  for(i=0; i < p->n; i++){
    *(dst++) = p->p[i]->v[0];    /* xdot = v */
    *(dst++) = p->p[i]->v[1];
    *(dst++) = p->p[i]->v[2];
    *(dst++) = p->p[i]->f[0]/m; /* vdot = f/m */
    *(dst++) = p->p[i]->f[1]/m;
    *(dst++) = p->p[i]->f[2]/m;
  }
}
```

In code

```
void EulerStep(ParticleSystem p, float DeltaT){
  ParticleDeriv(p,temp1);   /* get deriv */
  ScaleVector(temp1,DeltaT)       /* scale it */
  ParticleGetState(p,temp2);      /* get state */
  AddVectors(temp1,temp2,temp2);  /* add -> temp2 */
  ParticleSetState(p,temp2);      /* update state */
  p->t += DeltaT;                 /* update time */
}
```

Integration

Euler Method

– $S(t+h) = S(t) + h*S'(S(t),t)$

– What's S' ?

– $S' = (P', V') = (V, A) = (V, F/m)$

– Simple to implement

– Requires only one evaluation of S'

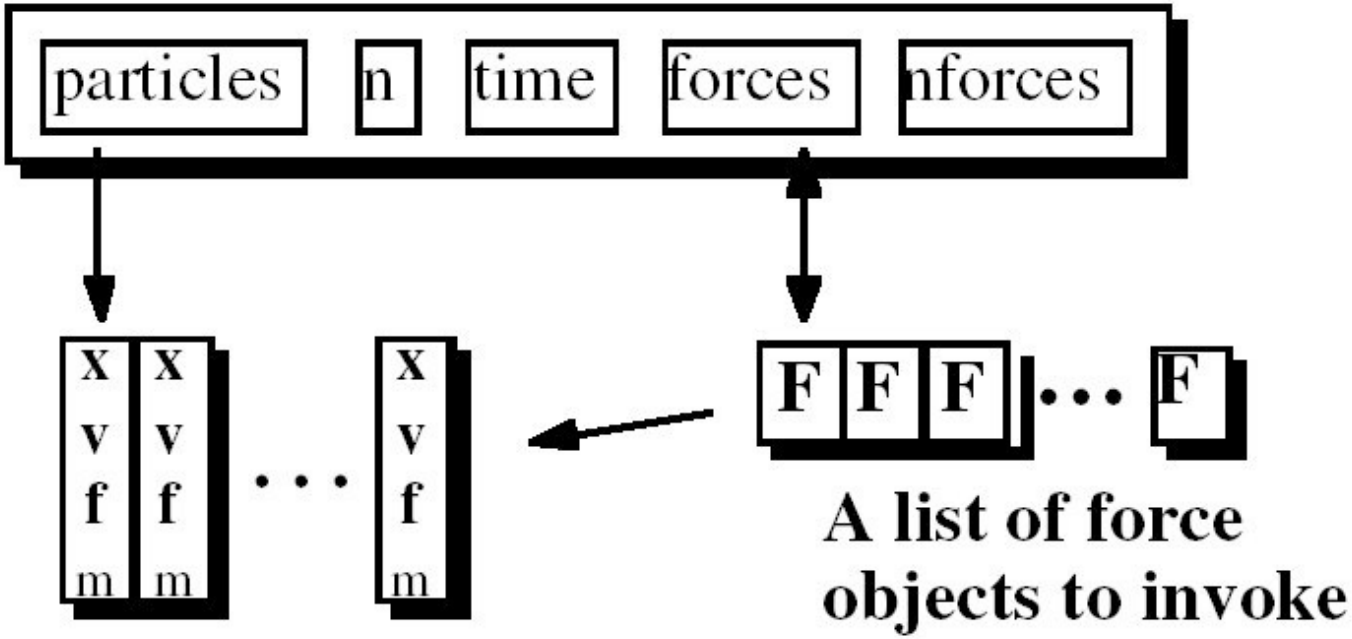
– Simple enough to be coded directly into the simulation loop

The Derivative Function

How to implement the S’ function

- Want V and A
- Know V is just the particle’s current velocity
- a = F/m. Evaluate forces here

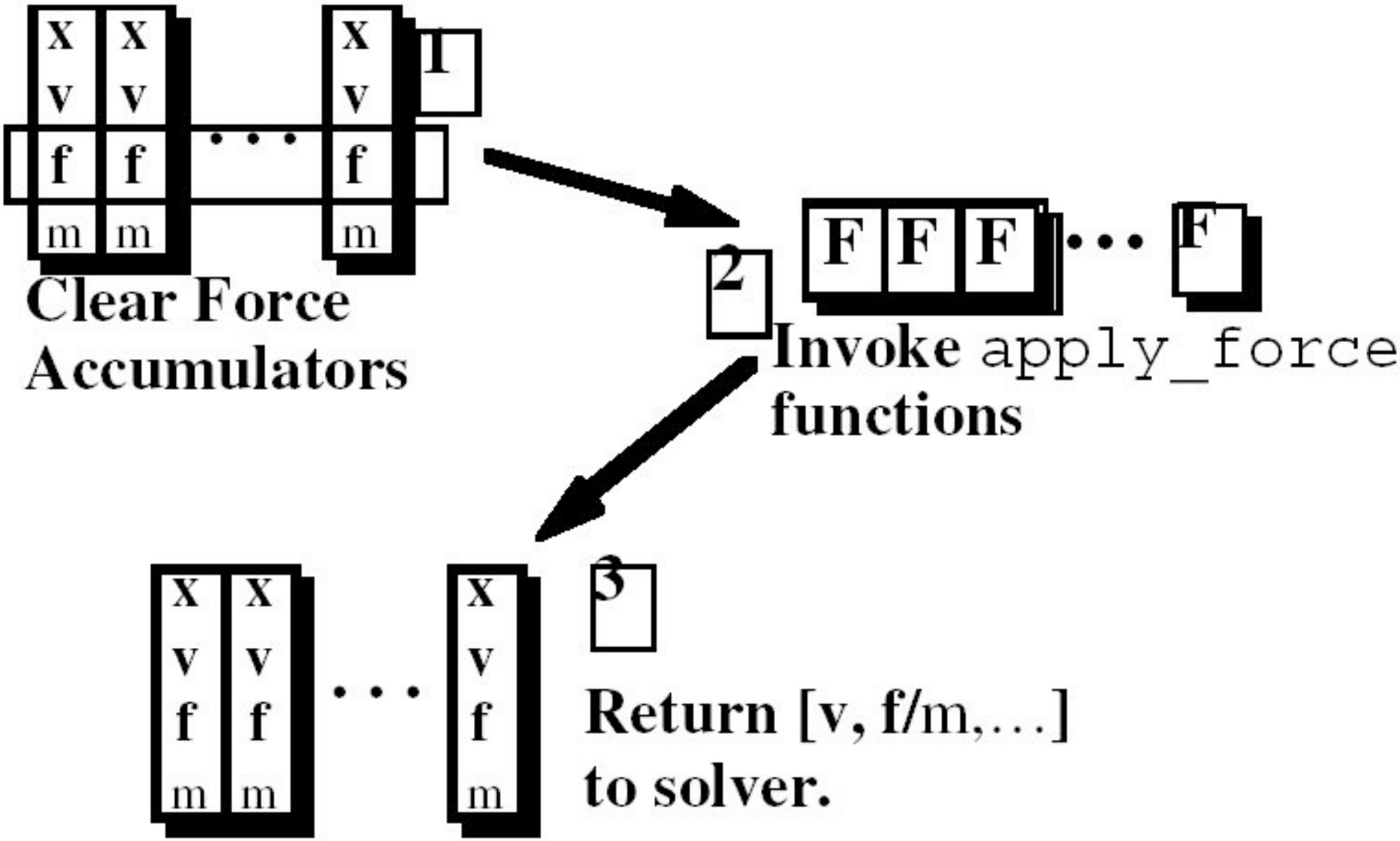
Forces



Forces are stored at the system level

They are invoked during the derivative evaluation loop

Derivative Evaluation



Forces

Typically, have multiple independent forces

- – For each force, add its contribution to each particle
- – Need a force accumulator variable per particle
- – Or accumulate force in the acceleration variable, and divide by m after all forces are accumulated
- – Need to evaluate F at every time step
- – The force on one particle may depend on the positions of all the others

Forces

Example forces

– Earth gravity, air resistance

– Force fields

– Wind

– Attractors/Repulsors

– Vortices

– Springs, mutual gravitation

Forces

Earth Gravity

- f = -9.81*(particle mass in Kg)

Drag

- f = -k*v

Uniform Wind

- f=k

Simulation Loop Recap

A recap of the loop (for each timestep):

- – Initialize/Emit particles
- – Run integrator (evaluate derivatives)
- – Update particle states
- – Render
- – Repeat!

Particle Systems

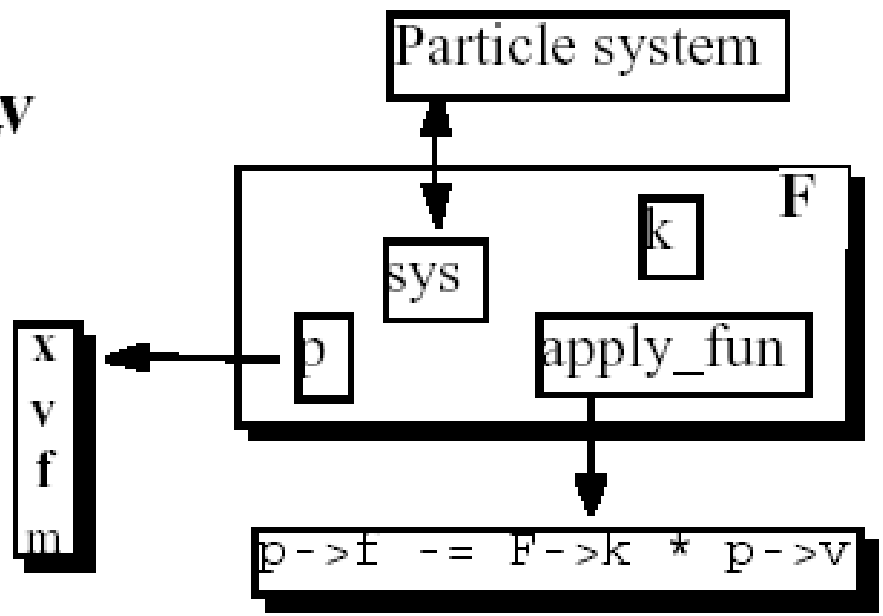
New particles are born, old die

- At each time step
- – Update attributes of all particles
- – Delete old
- – Create new (recycle space)
- – Display current state

Applying a force

Force Law:

$$\mathbf{f}_{drag} = -k_{drag}\mathbf{v}$$



A Force Object: Viscous Drag

Emitters

Usually described as a surface from which particles appear

- Object with position, orientation
- Regulates particle “birth” and “death”
- Usually 1 per particle system

Many user definable parameters:

size, mass, age, emitter size, initial velocity and direction, emission rate, collision detection (internal and external), friction coefficients, global forces, particle split times, delays, and velocities, color evolution, etc.

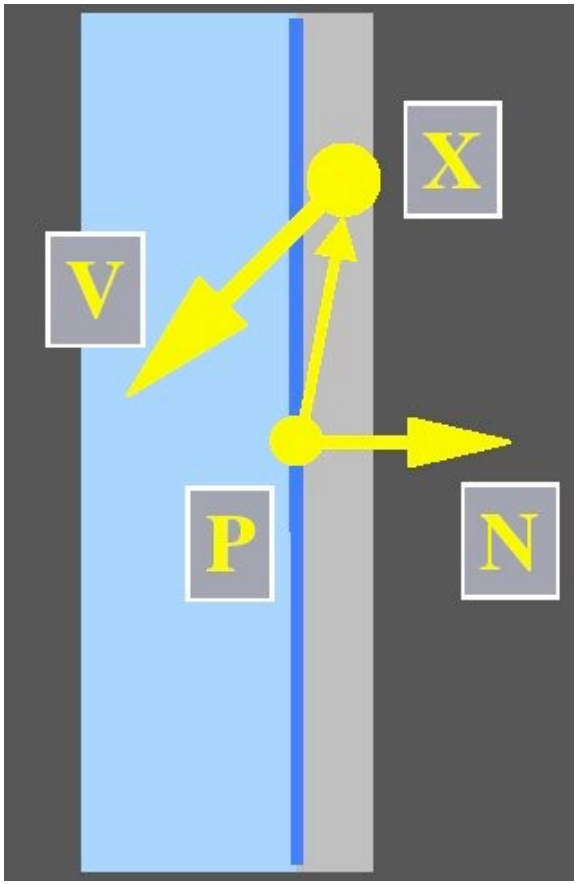
Particle-Object Collision Detection

- With very simple objects, this is easy
- Plane: Test the sign of $(x-p) \cdot n$
- Box: Check six planes!
- Arbitrary closed object: Cast ray from p to some outside point. If it intersects object an odd number of times, p is inside.
- Relies on having a CLOSED object!
- Should accelerate intersection with a grid or octree

Collision Detection

- Can be very complex in general
- Start with a simple case:
  particle-plane collision

$$(x - p).N$$



- Response

$$v_n = (v.N)N$$

$$v_t = v - v_n$$

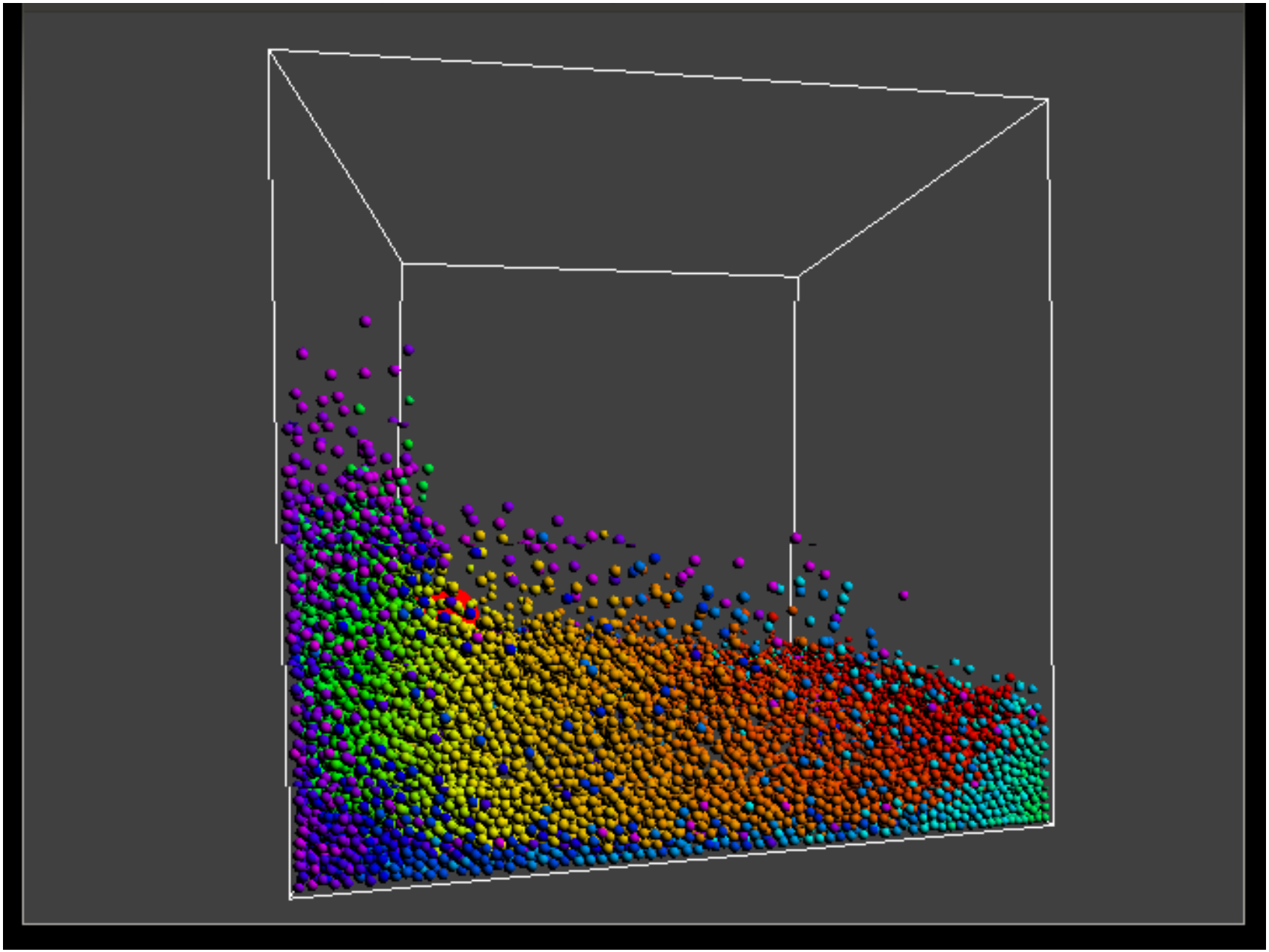
Response

- Elastic collision
  - Reflect normal component
  - Leave tangential component unchanged

- Inelastic collision
  - r, coefficient of restitution

$$v' = v_t - r v_n$$

Particle Collision Demo



Physically-based Animation

Must obey laws of physics

A lot harder to simulate

Not just interpolation

Must solve for equilibrium solutions

- Navier-Stokes equations

AI for large simulations

- Want to represent large groups
- Too manually intensive for keyframe
- Autonomous agents
  - eg., armies, flocks of birds

Autonomous Agents

Each entity has state: velocity, goal, etc.

A limited perception of neighbors

High-level goals:

Collision avoidance

- Don’t hit each other
- Don’t hit scene objects
- Similar direction of travel: being part of group
- Global flock centering too restrictive
- Flock splitting around hurdles important

Rules to model

Physics

- Gravity

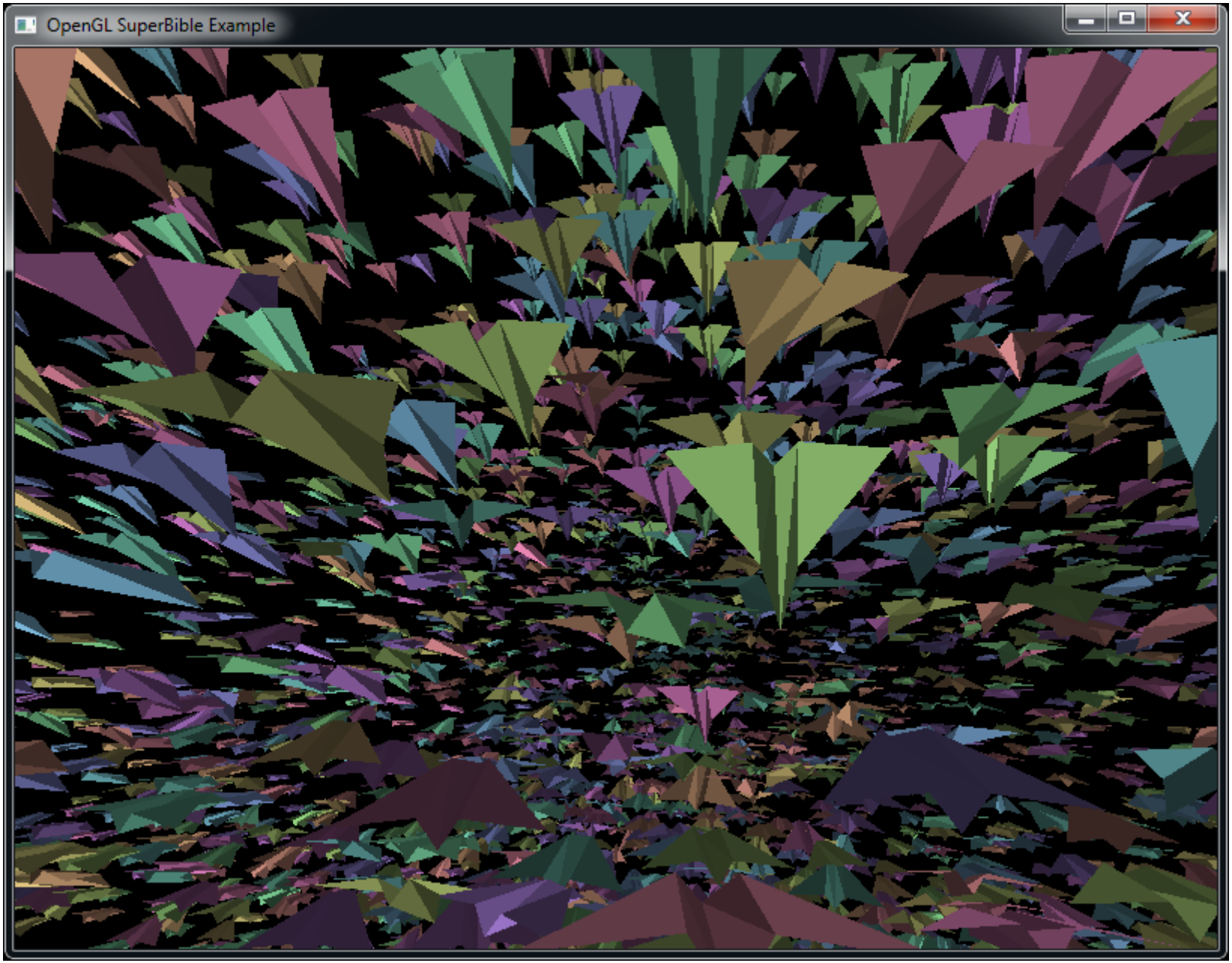
Perception

- What local information is available
- Neighbors, no designated leader to follow

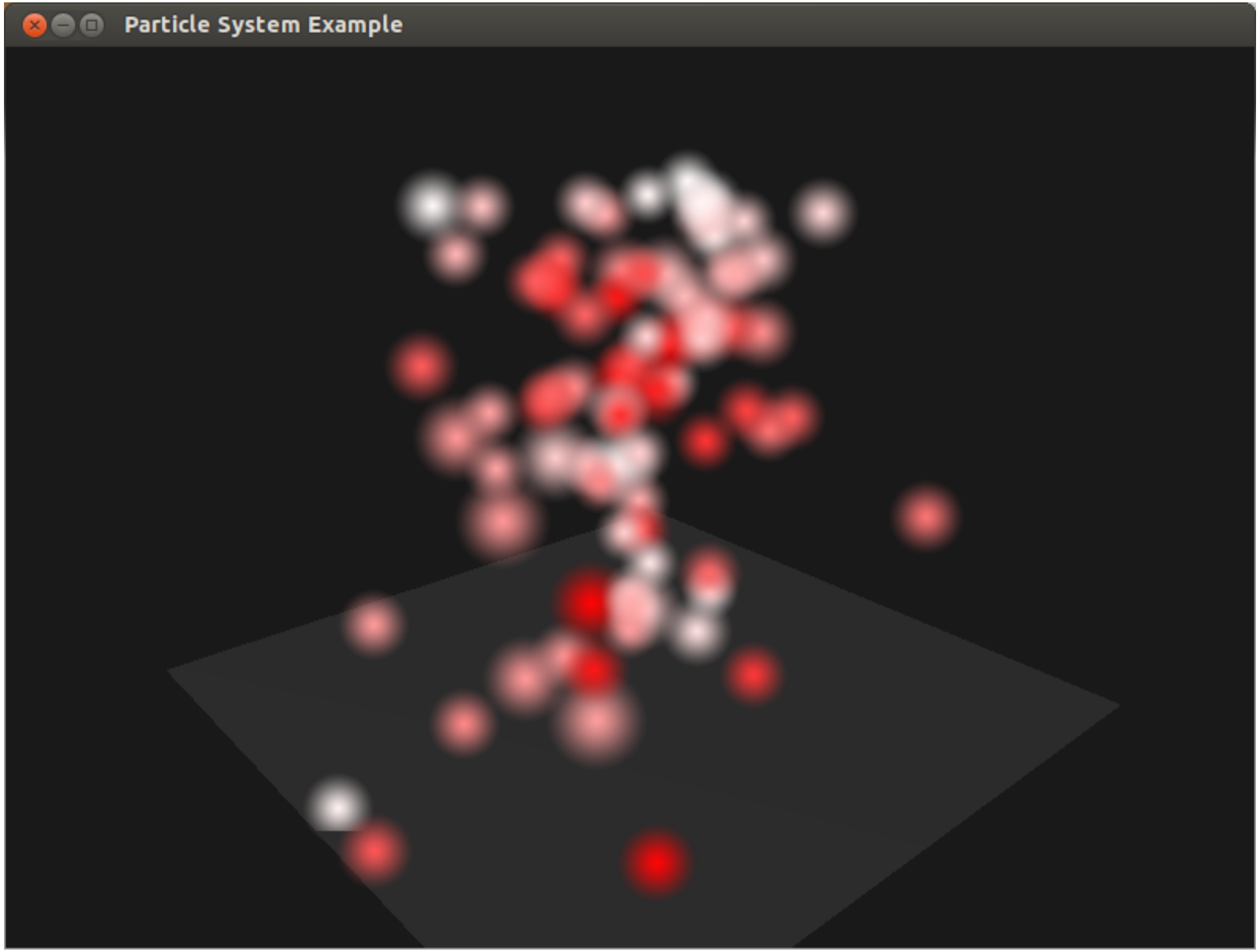
Reasoning

- Rules by which members decide where to go
- Collision avoidance, velocity matching

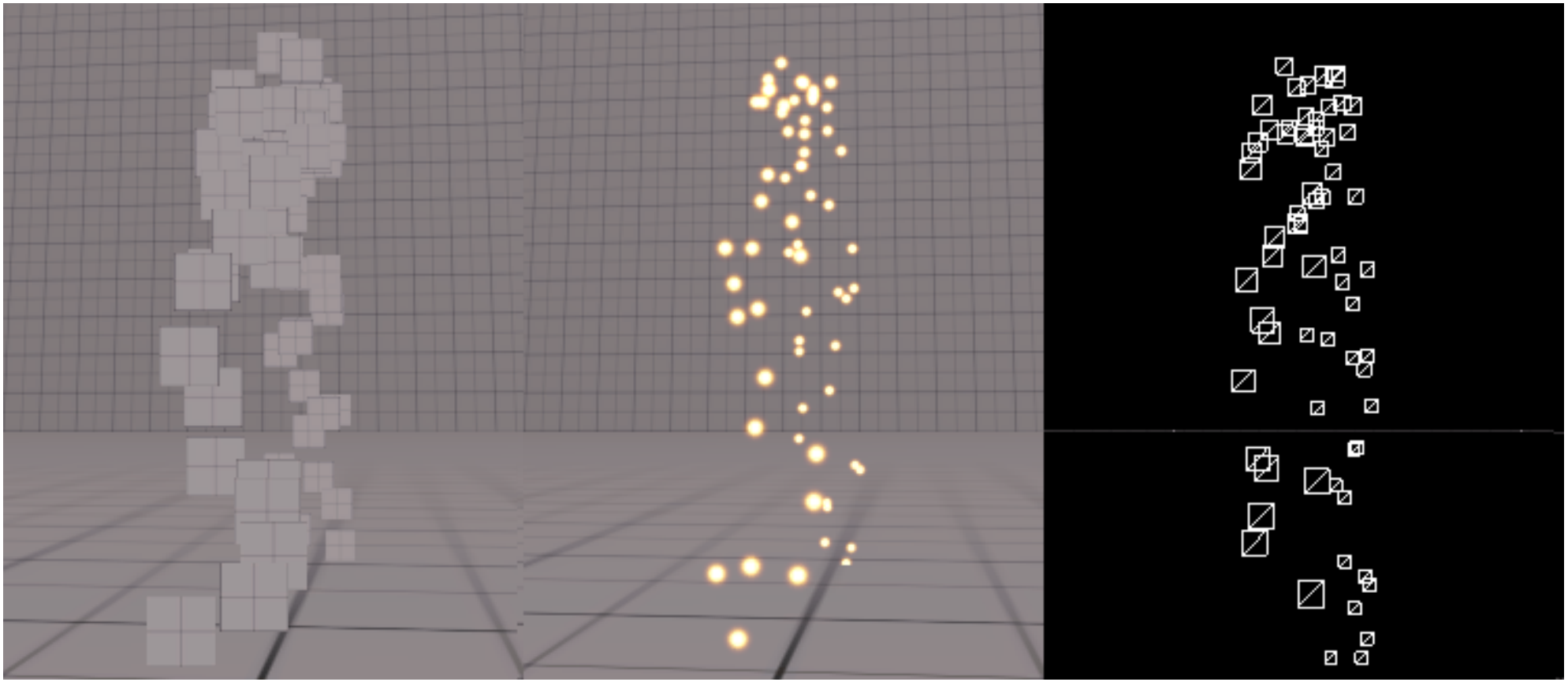
Airplane flocking demo



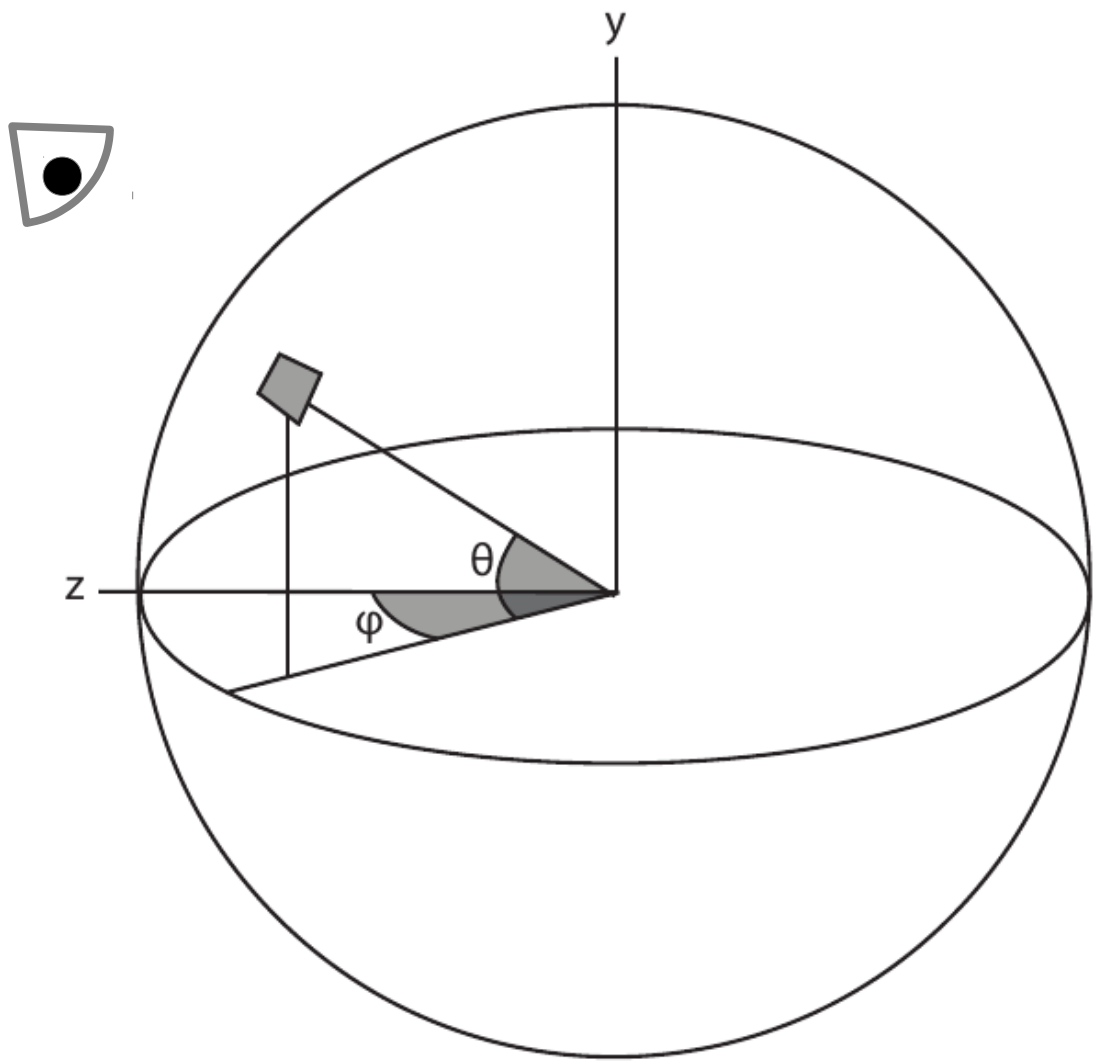
PPA3 Particle System



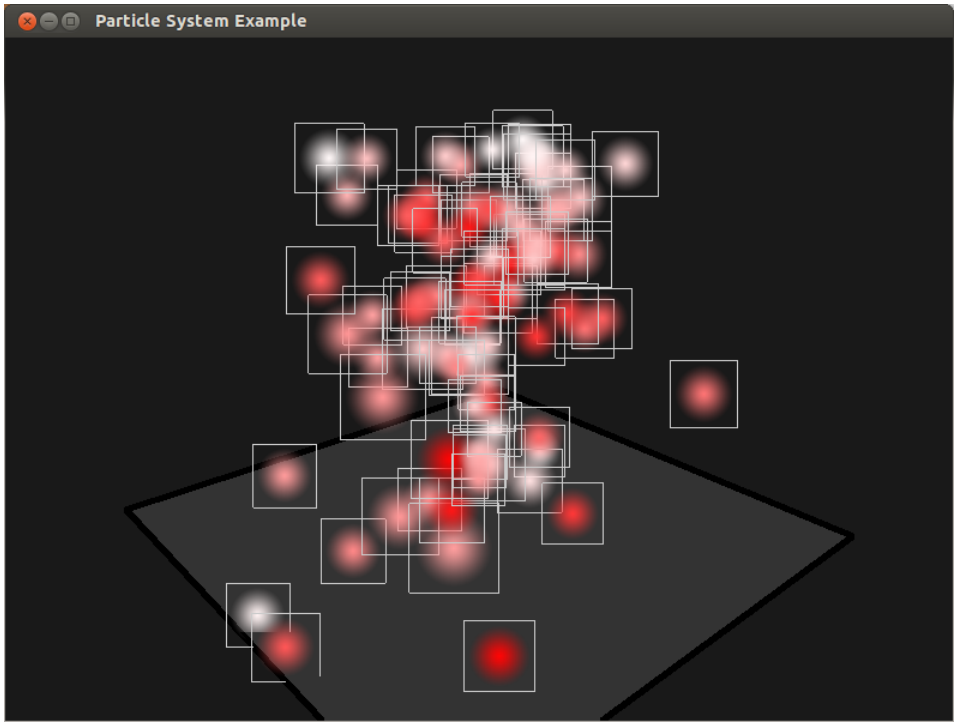
Particle System - Billboards

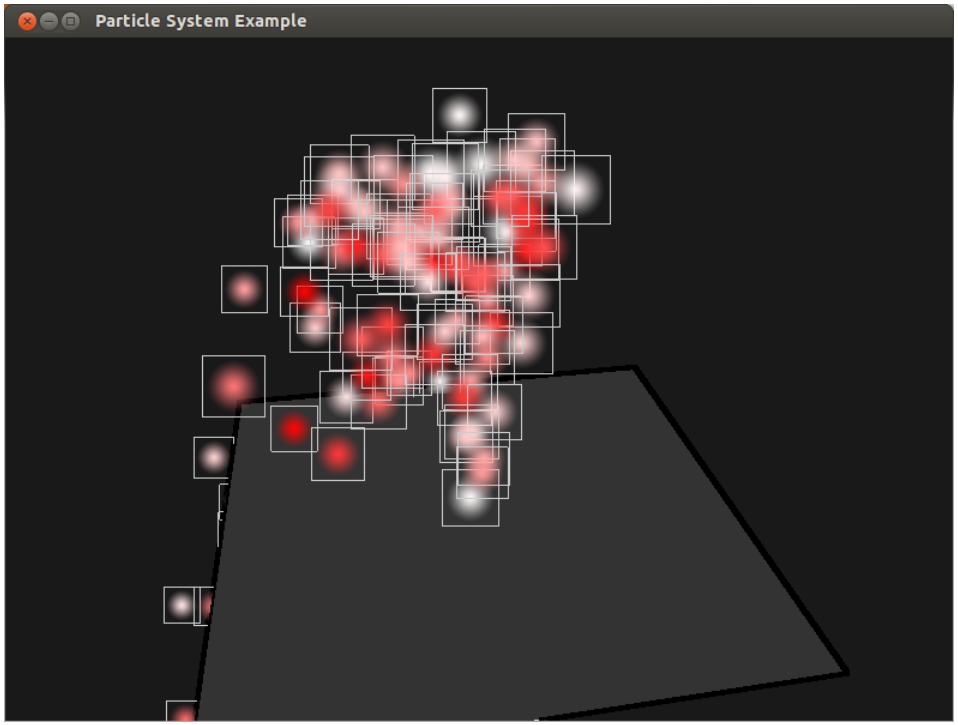


Particle System - Billboards



Particle System - Billboards



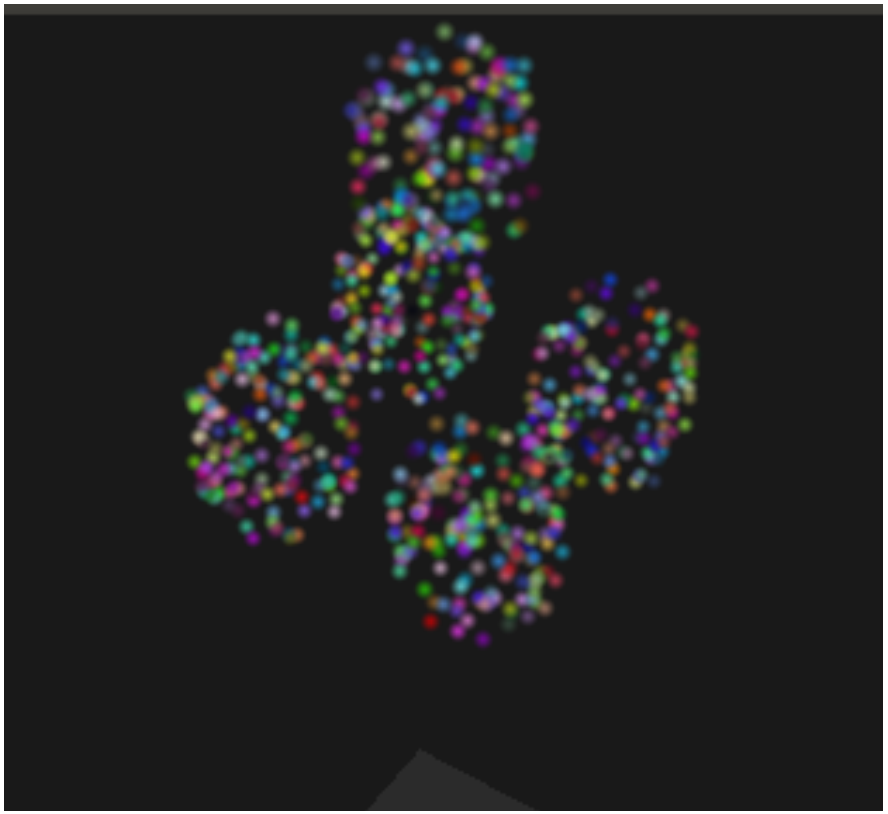


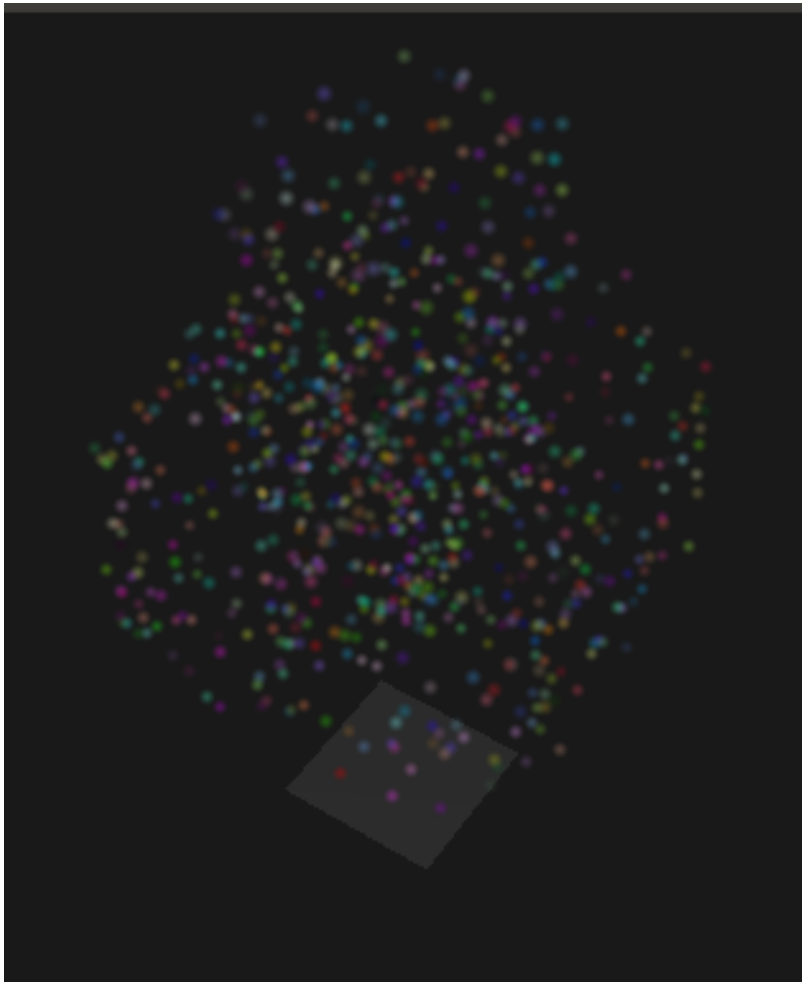
PPA3 Particle System

Assignment tasks

- Integrator timestep
- Accumulate forces (gravity, wind)
- Orient Billboards
- visualize velocity vectors

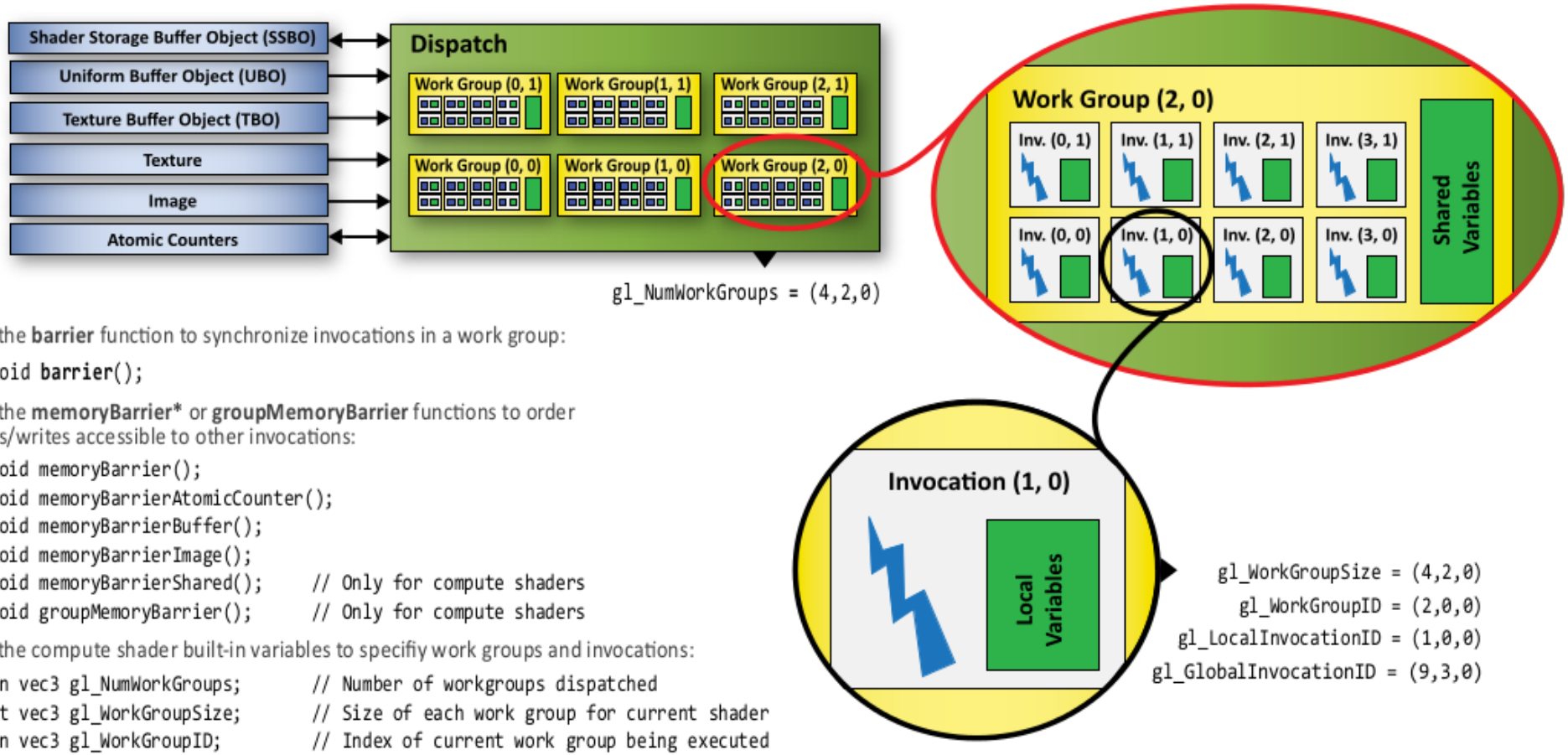
PPA3 Particle System Demo





Compute Shaders

OpenGL Compute Programming Model and Compute Memory Hierarchy



Use the **barrier** function to synchronize invocations in a work group:

```
void barrier();
```

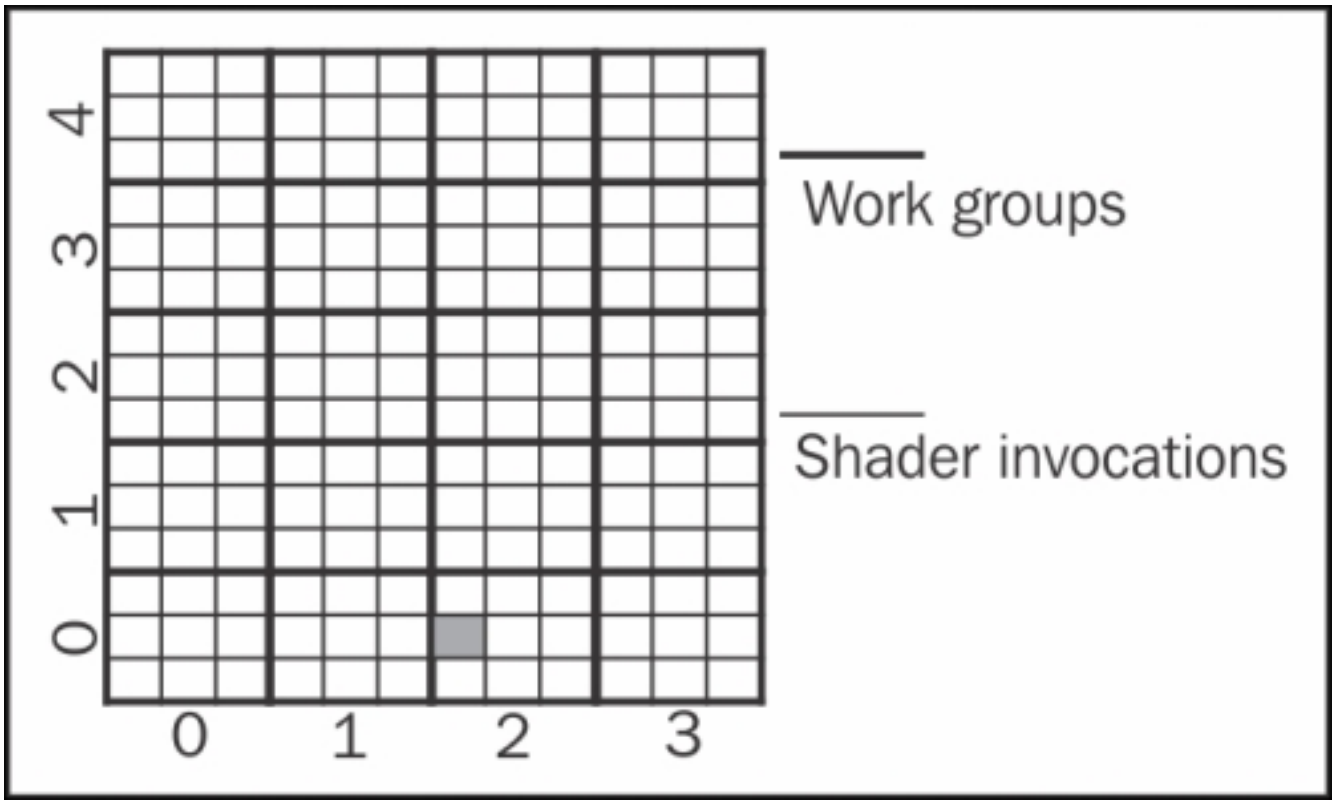
Use the **memoryBarrier*** or **groupMemoryBarrier** functions to order reads/writes accessible to other invocations:

```
void memoryBarrier();
void memoryBarrierAtomicCounter();
void memoryBarrierBuffer();
void memoryBarrierImage();
void memoryBarrierShared();     // Only for compute shaders
void groupMemoryBarrier();      // Only for compute shaders
```

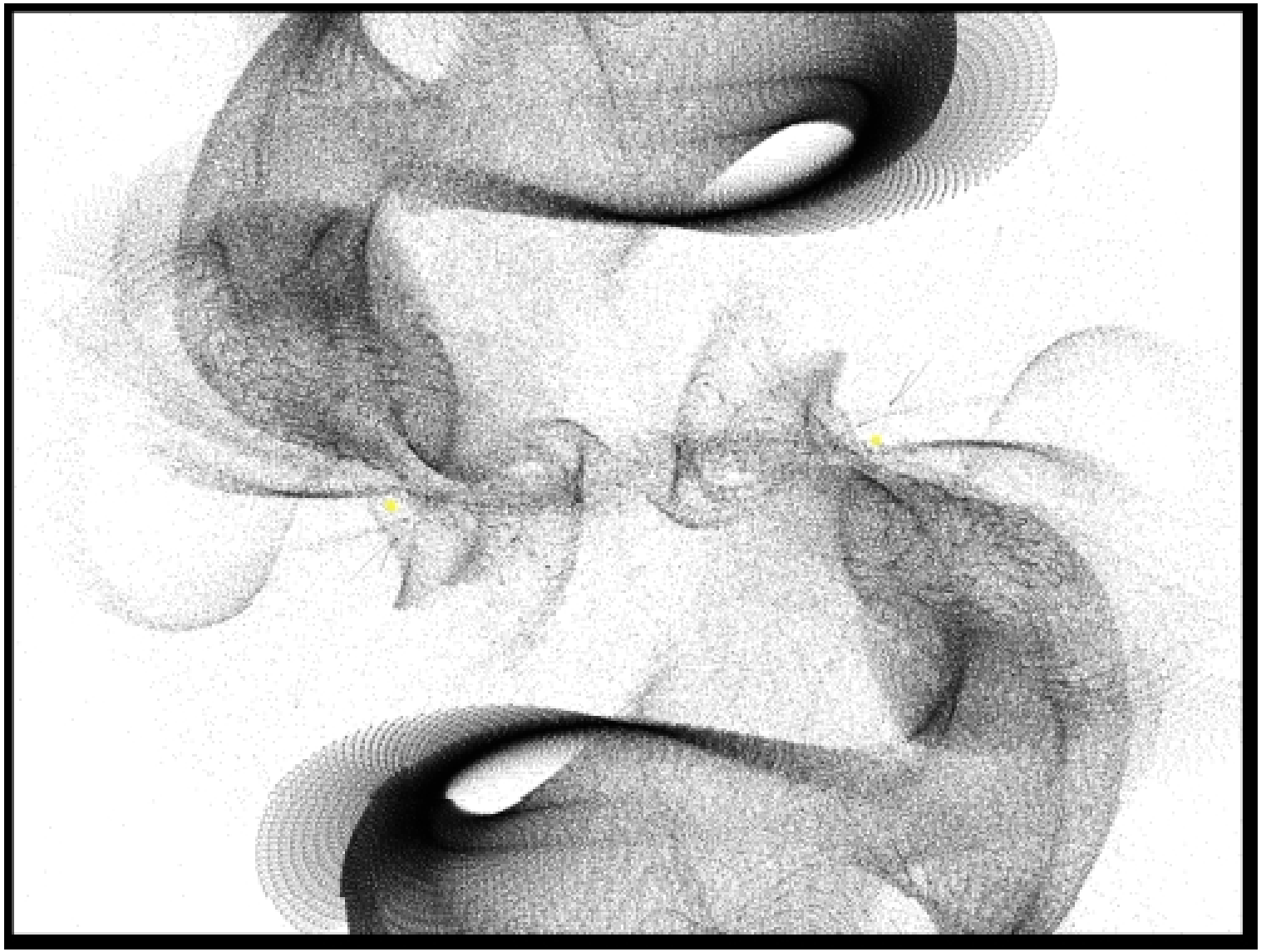
Use the compute shader built-in variables to specifiy work groups and invocations:

```
   in vec3 gl_NumWorkGroups;       // Number of workgroups dispatched
const vec3 gl_WorkGroupSize;       // Size of each work group for current shader
   in vec3 gl_WorkGroupID;         // Index of current work group being executed
   in vec3 gl_LocalInvocationID;   // index of current invocation in a work group
   in vec3 gl_GlobalInvocationID;  // Unique ID across all work groups and threads. (gl_GlobalInvocationID = gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID)
```

Compute Shaders



Compute Shaders



Compute Shaders

To use a compute shader in your program:

- Create a compute shader with glCreateShader() using the type GL_COMPUTE_SHADER

- Set the shader source with glShaderSource() and compile it with glCompileShader()

- Attach it to a program object with glAttachShader() and link it with glLinkProgram()

- Make the program current with glUseProgram()

Compute Shaders

Launch compute workloads with glDispatchCompute() or glDispatchComputeIndirect()

In your compute shader:

- Specify the local workgroup size using the local_size_x, local_size_y and local_size_z input layout qualifiers.

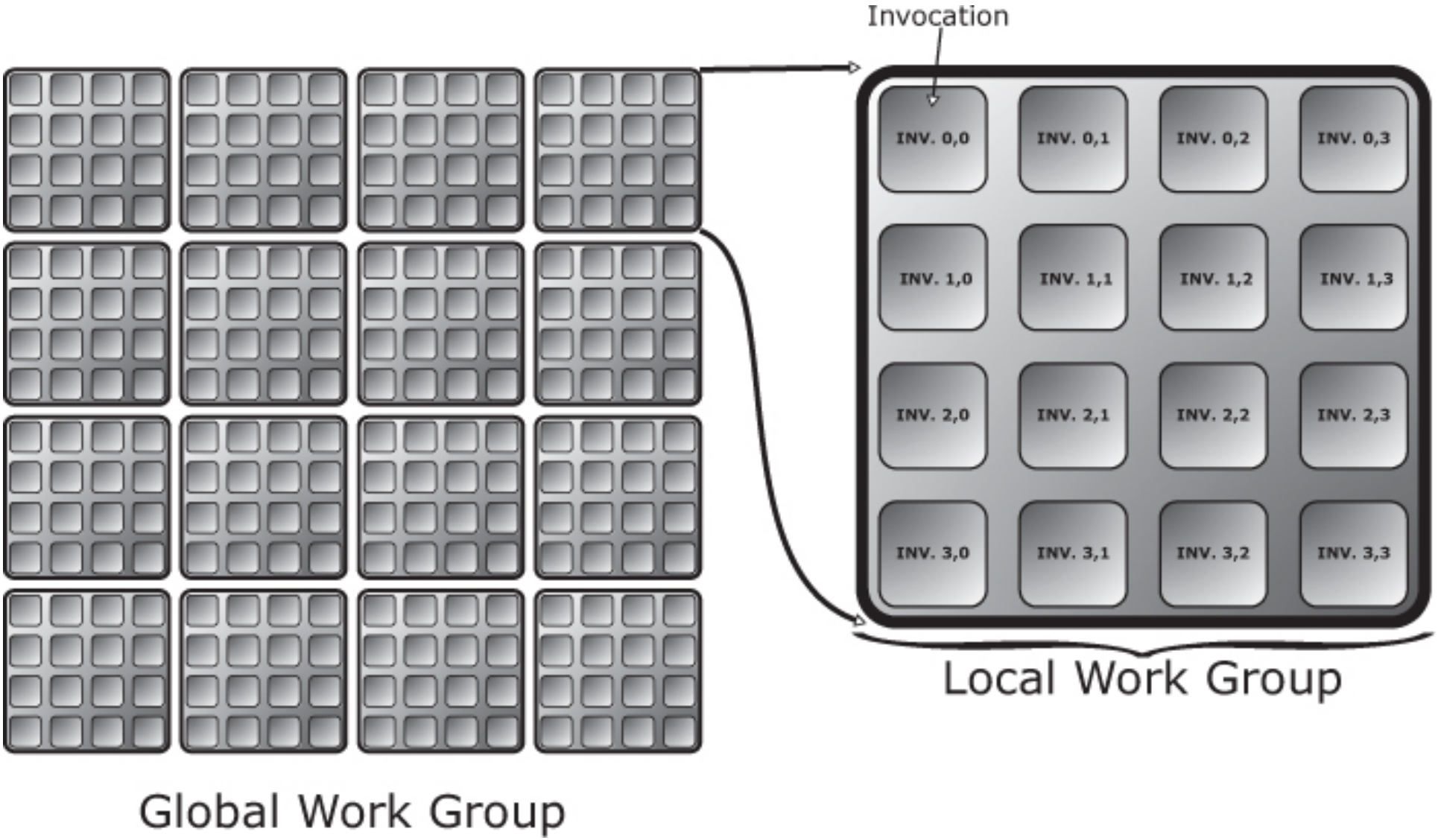
- Read and write memory using buffer or image variables, or by updating the values of atomic counters.

Compute Shaders

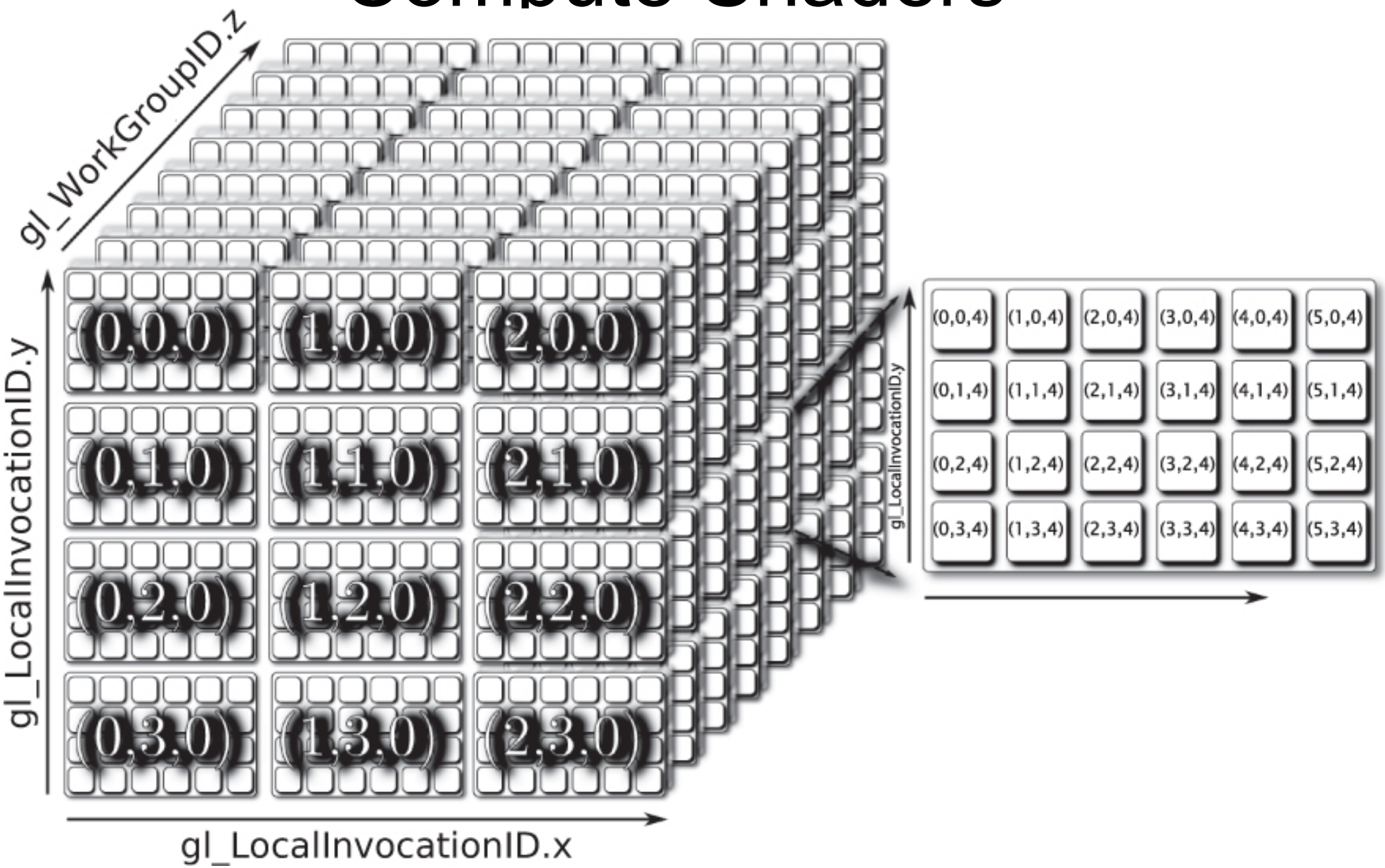
Variable	Type	Meaning
`gl_WorkGroupSize`	`uvec3`	The number of invocations per work group in each dimension. Same as what is defined in the layout specifier.
`gl_NumWorkGroups`	`uvec3`	The total number of work groups in each dimension.
`gl_WorkGroupID`	`uvec3`	The index of the current work group for this shader invocation.
`gl_LocalInvocationID`	`uvec3`	The index of the current invocation within the current work group.
`gl_GlobalInvocationID`	`uvec3`	The index of the current invocation within the global compute space.

gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID

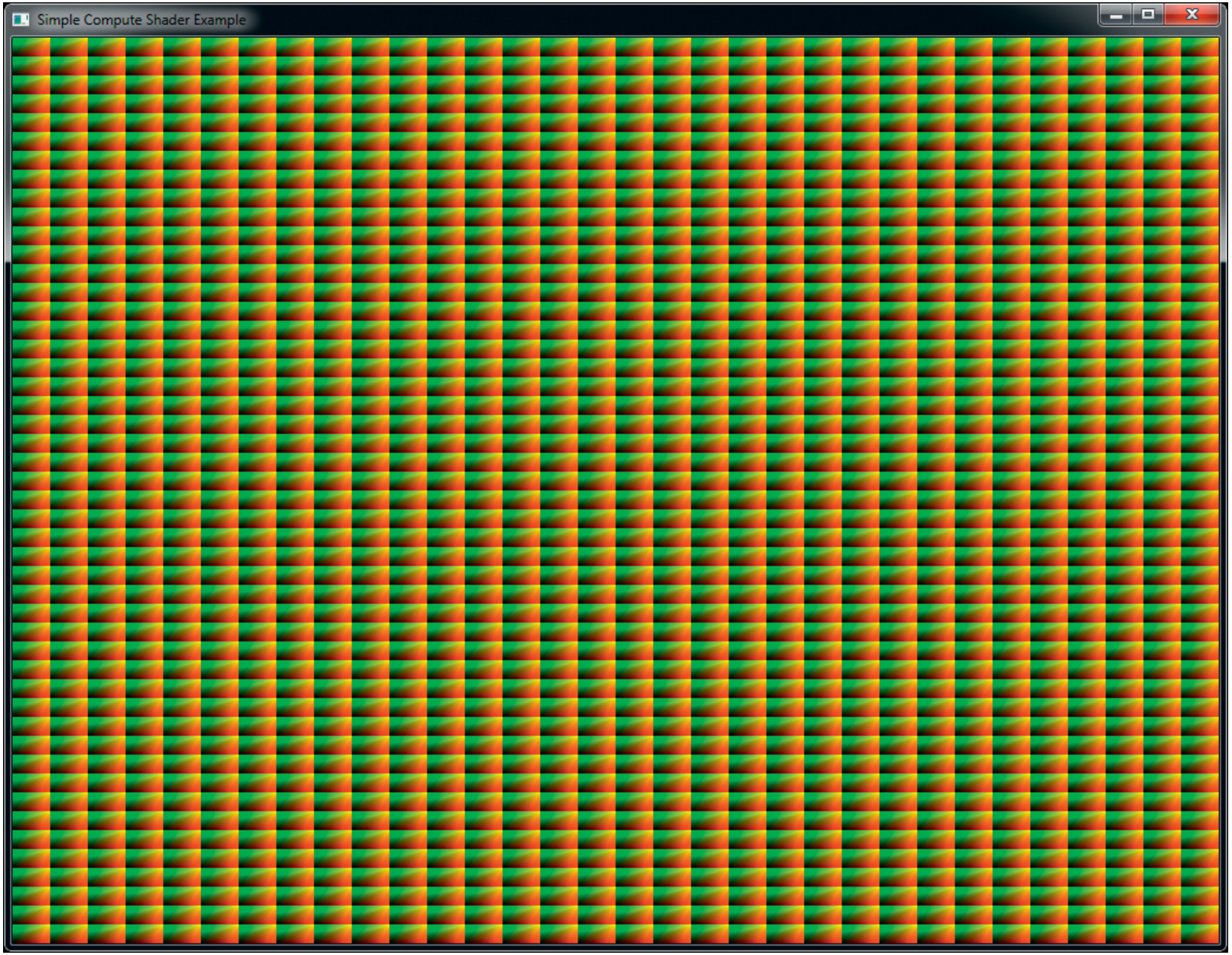
Compute Shaders



Compute Shaders



Compute Shaders



Compute Shaders

Limitations

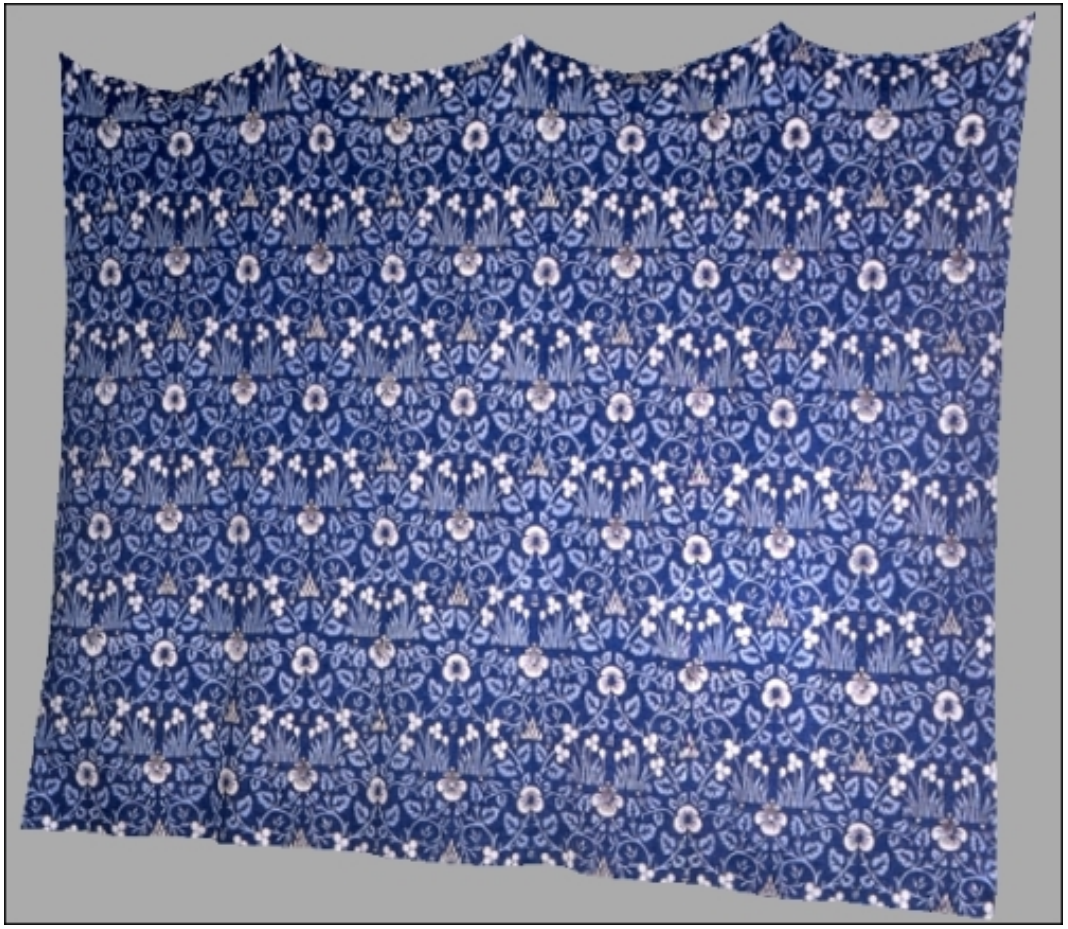
GL_MAX_COMPUTE_WORK_GROUP_COUNT

GL_MAX_COMPUTE_WORK_GROUP_SIZE

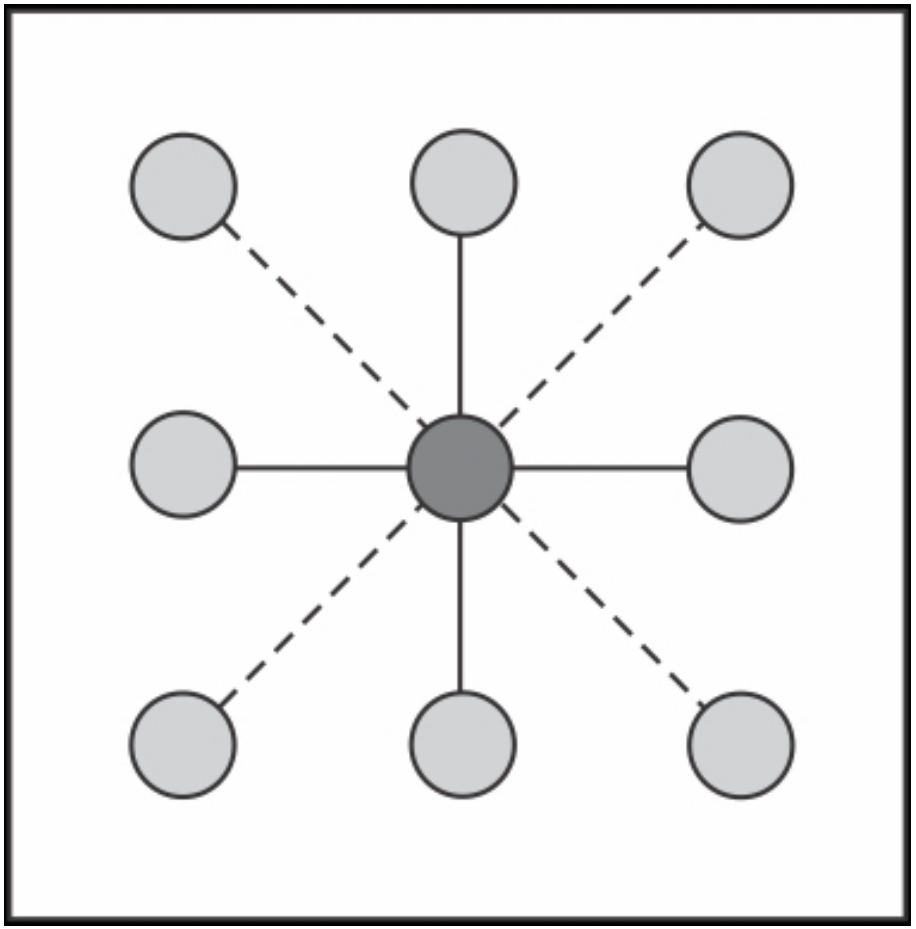
GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS

GL_MAX_COMPUTE_SHARED_MEMORY_SIZE

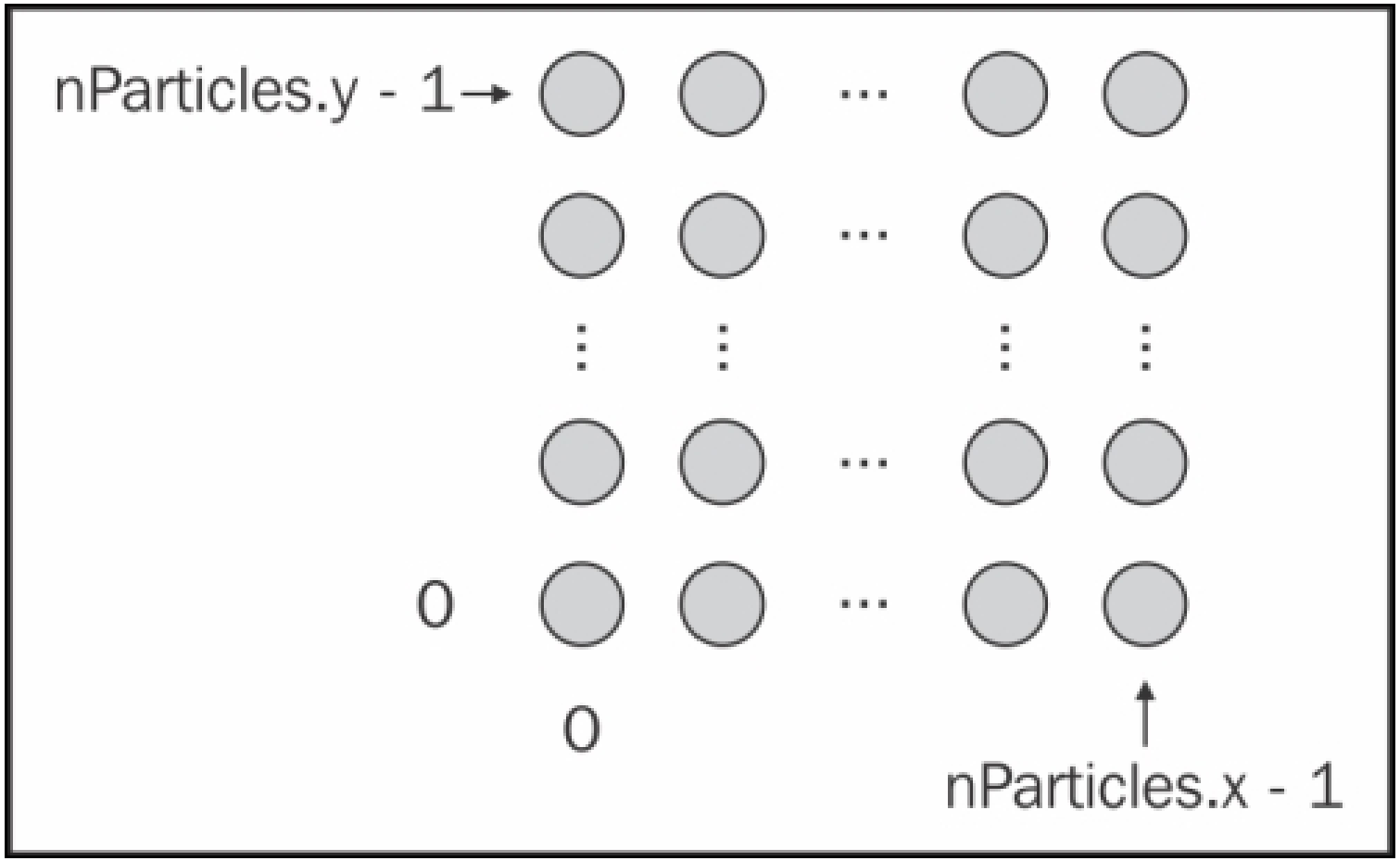
Compute Shaders



Compute Shaders



Compute Shaders



Compute Shaders