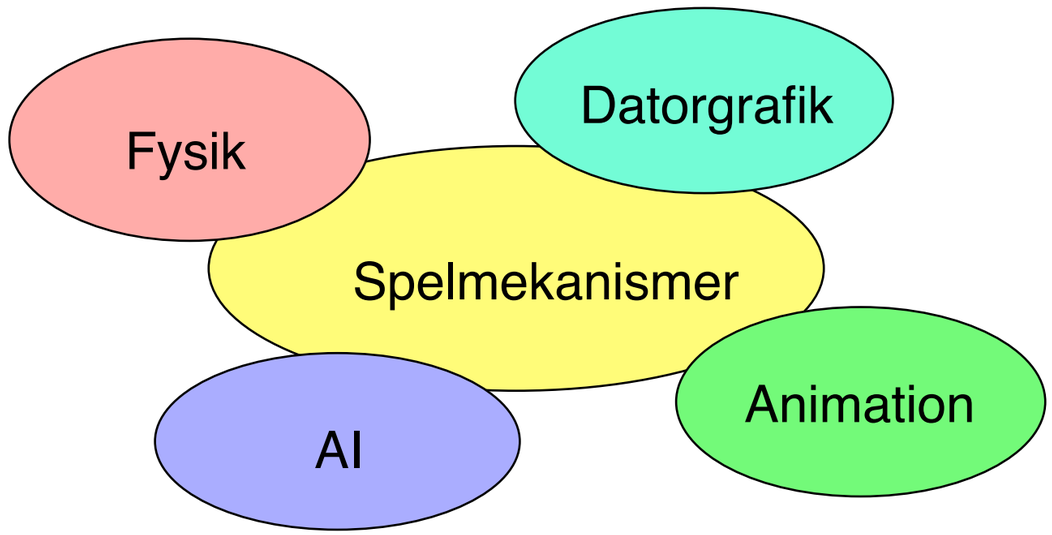
TSBK 03

Teknik för avancerade datorspel

Ingemar Ragnemalm, ISY



Föreläsning 5

GPU computing

- GPU computing/GPGPU, vad och varför
- Partikelsystem i shaders
- Intro till CUDA
- Lite om Compute Shaders

GPU Computing/GPGPU
Generella beräkningar på GPUs

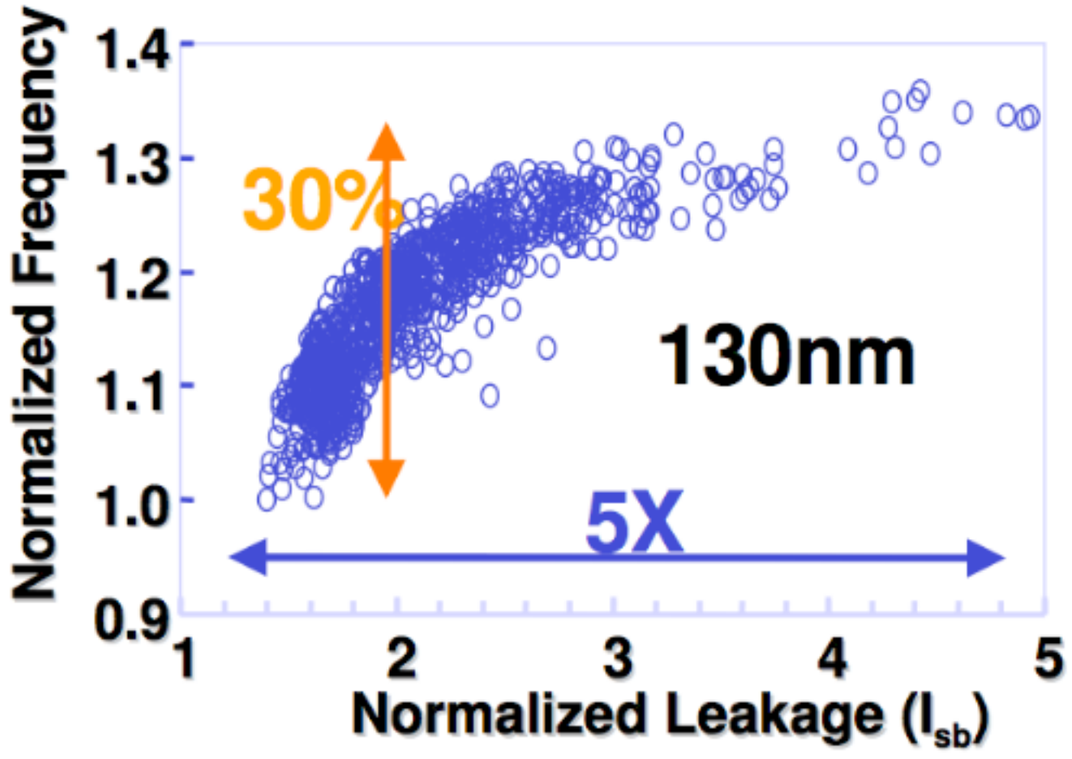
Intressant trend som kom i början på 2000-talet: Försök använda GPU'ns beräkningskraft för andra problem än grafik!

Argument för: GPU'erna uppvisar stor prestandaökning medan CPU'er håller på att avstanna.

Nyckelkomponenter: Shaders, flyttalsbuffrar, parallellism.

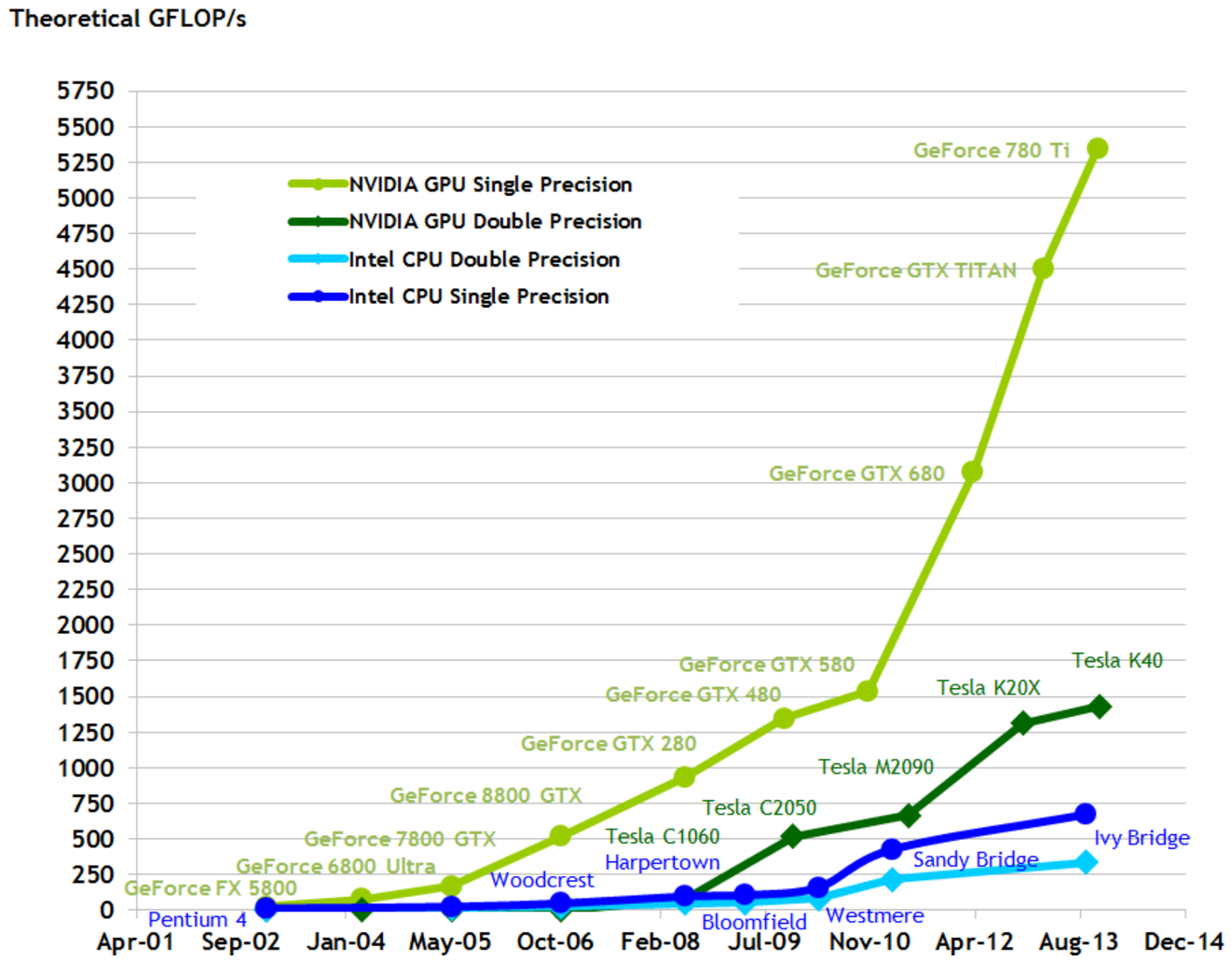
Power wall

Vi kan inte längre höja frekvensen. Effekten stiger brant.



Men vi kan öka ANTALET!

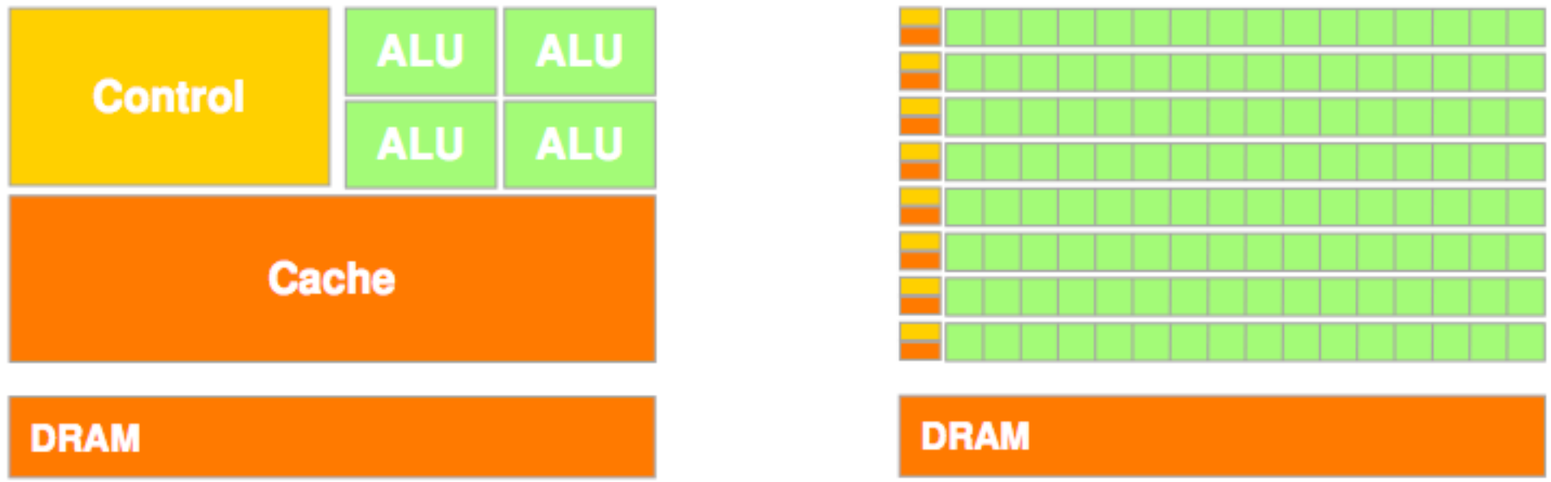
The GFLOPS race




S.Leib
AS YOU CAN SEE, THE CENTRAL ROLE HERE IS PLAYED BY THE INTEL XEON CPU...
INTEL
NVIDIA GPU
SUPERCOMPUTER
INTEL CPU

Hur är detta möjligt?

Area use:



Men i synnerhet: SIMD architecture, många små processorer

Typisk ”klassisk” GPGPU: som filtreringsdelen i HDR

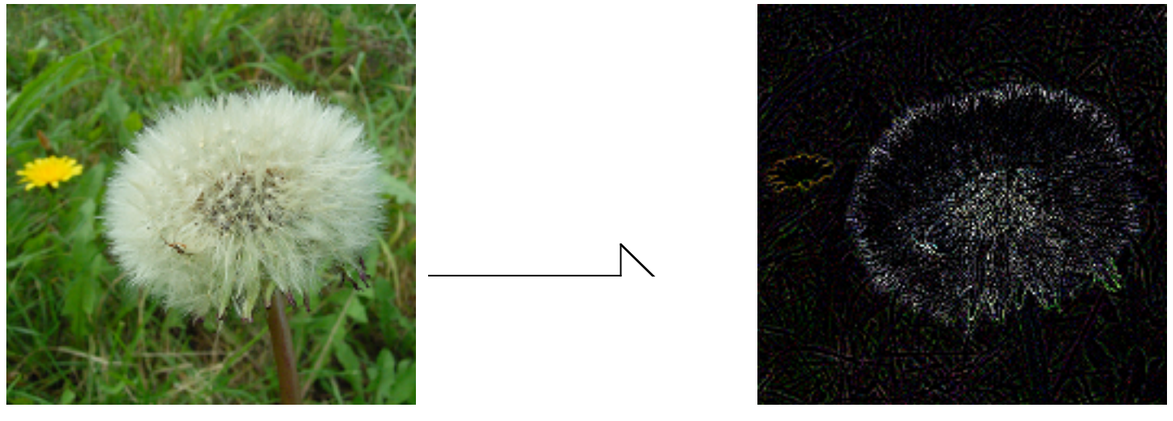
Samma grundkoncept:

- Rendera till rektangel över hela bilden
- Gärna FBOs
- Flyttalsbuffrar
- Ping-ponging, flera pass med olika shaders

Enklaste GPGPU: Processning av existerande bilder

Skillnad: Vi jobbar med en bild, en rektagulär yta, från början.

Linjära filter precis som i shaders, möjligheter till multipass och kombination av bilder.



GPGPU-konceptet

- Array av indata = textur
- Array av utdata = resulterande frame buffer
- Beräkningskärna = shader
- Beräkning = rendering
- Återkoppling = växling av FBO eller kopiering av textur

OBS skillnaden i renderingssituation:

- Extremt enkel geometri. Inget behov av vertexshaders.
- Mycket arbete på pixelnivå. Stora krav på fragmentshaders, höga precisionskrav.

Enkelt exempel: process-array

Array av data läggs i textur

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

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

Hämtas sedan tillbaka till CPU igen.

Shader-baserad GPGPU idag?

Om ditt problem passar i grafikpipelinen!

Utmärkt för problem som lätt mappas på texturdata.

Problem: Data bör mappas på RGBA. Renderingsdelarna gör ditt program mer komplicerat.

Att processa partikelsystem med shaders

Stora partikelsystem bör hanteras på GPU! Detta kan göras med shaders, men även med CUDA/CL/CS.

CPU: Ohanterbart med mer än ganska små system.

Partikelsystem lämpligt för parallella beräkningar.

Shaders nära grafiken, men lite mindre flexibelt än t.ex. CUDA.

Minimalt partikelsystem

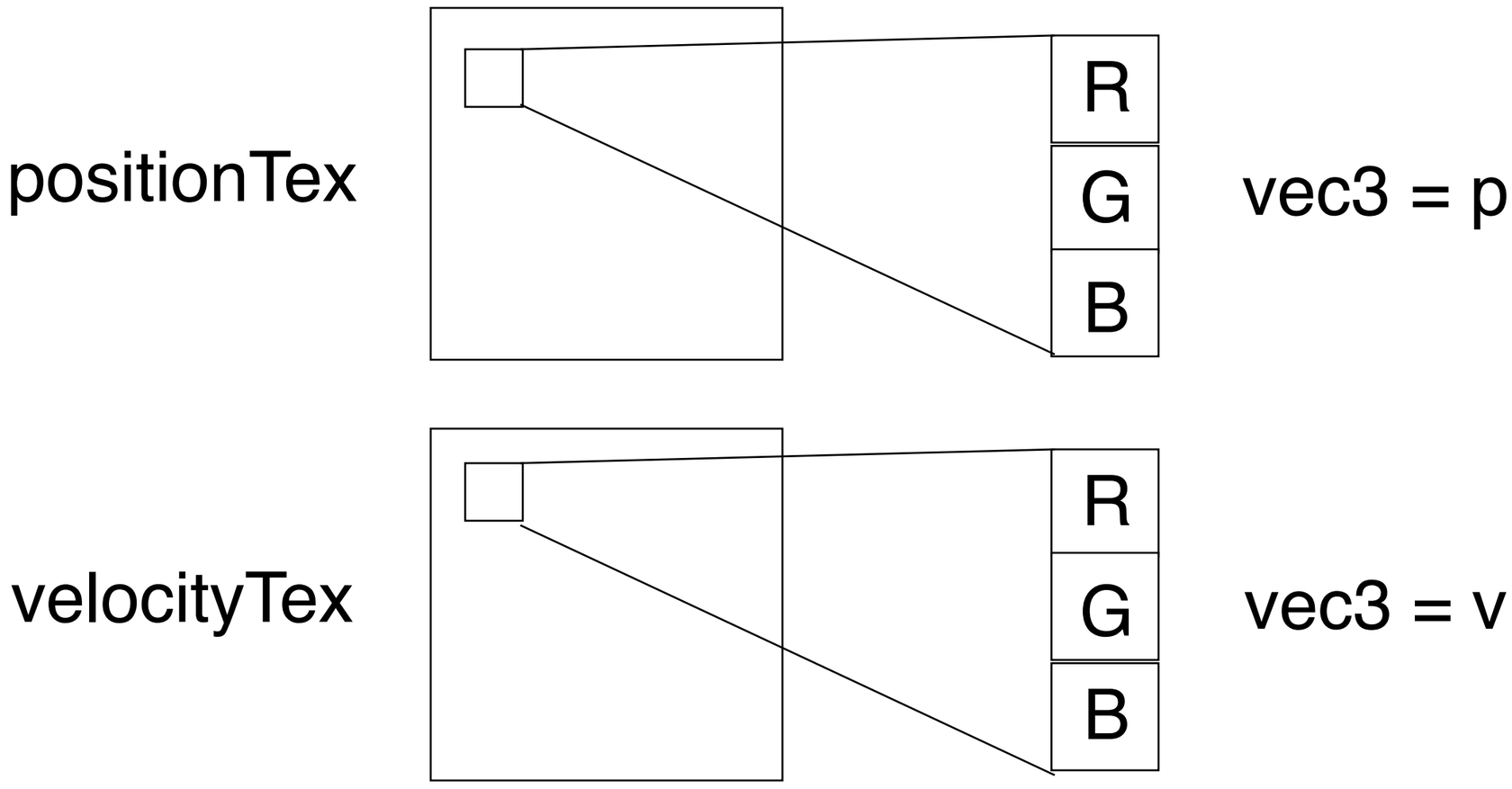
Position p
Hastighet v

Uppdatering:

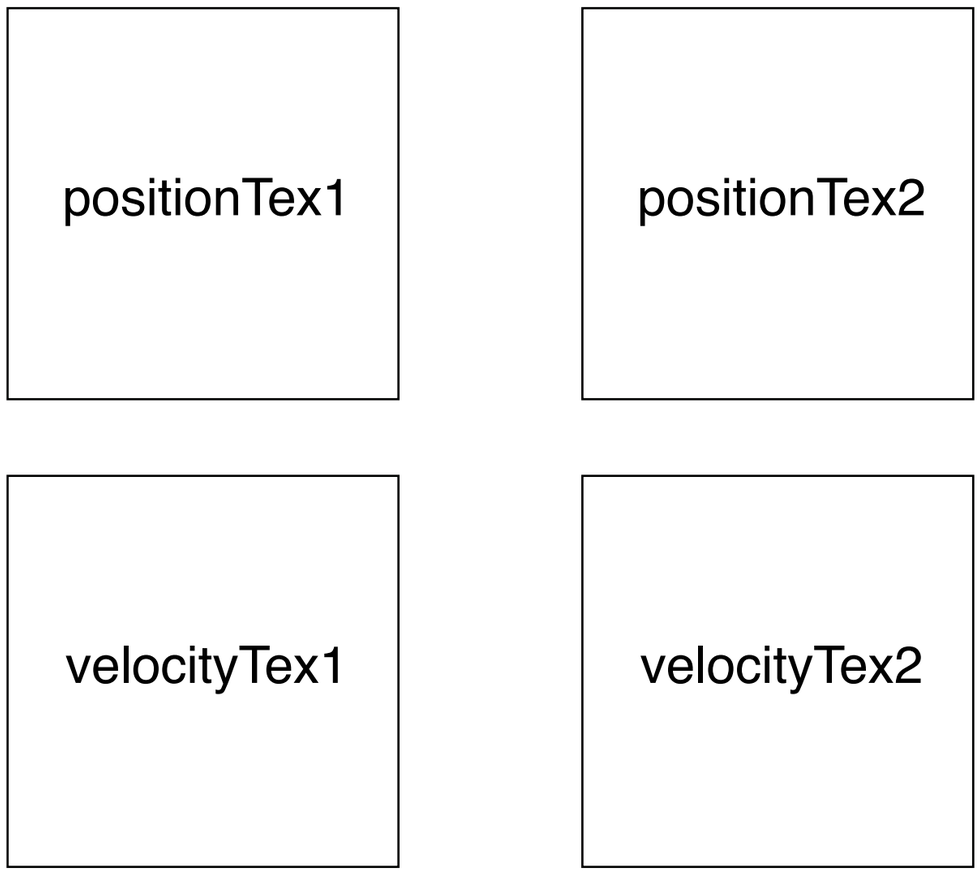
Position p = p + v * dt

Hastighet v = v + a * dt

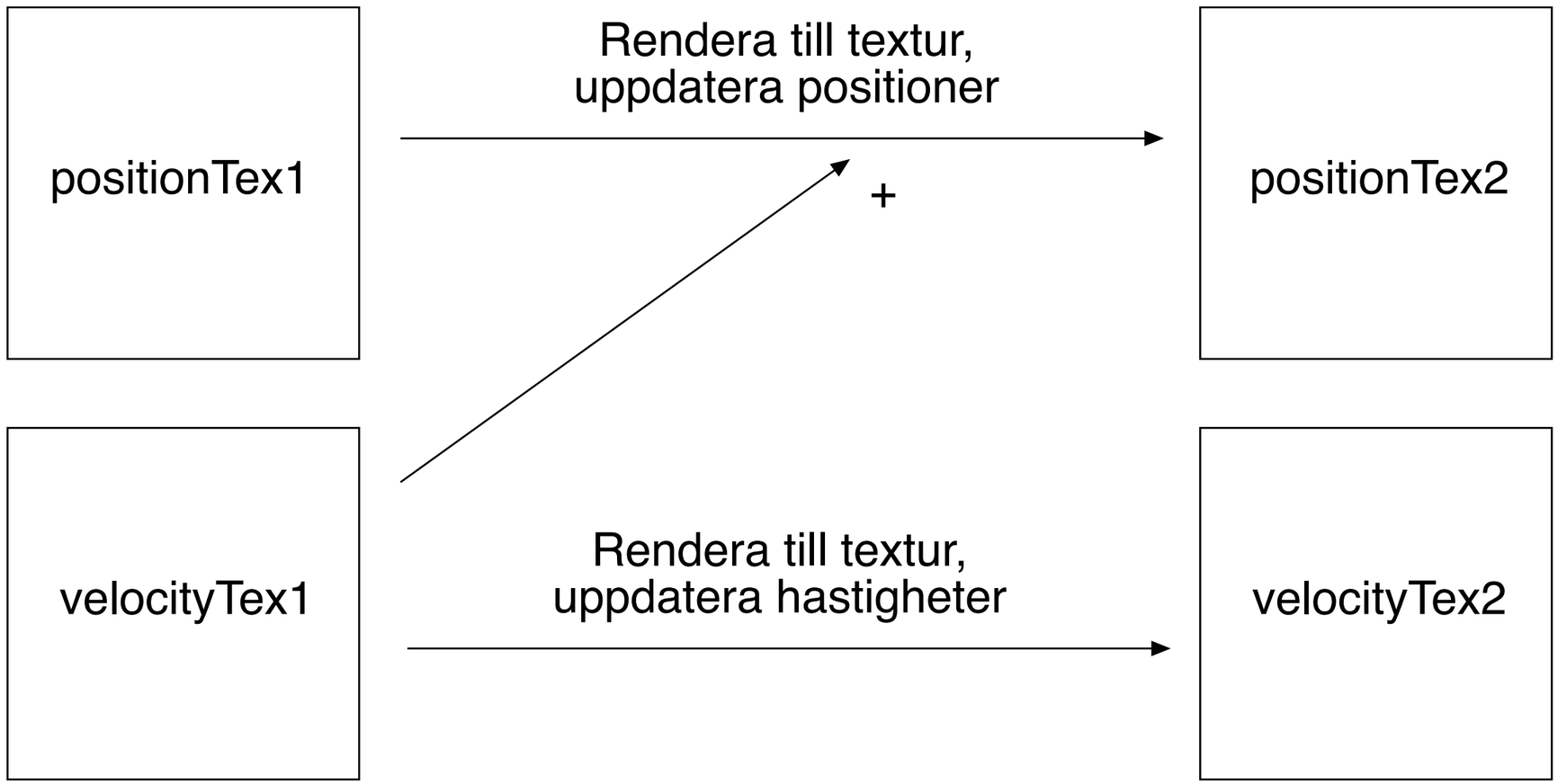
Lagra data i texturer!



Dubbla texturer för ping-ponging



Rendera till FBO för att uppdatera



Två problem kvar:

Sortering

Rendering

Rendering av partikelsystem

Görs typiskt med billboarding

Billboard = riktas alltid mot kameran

Texturerad quad med textur med transparens

Enklast: Tag bort rotation

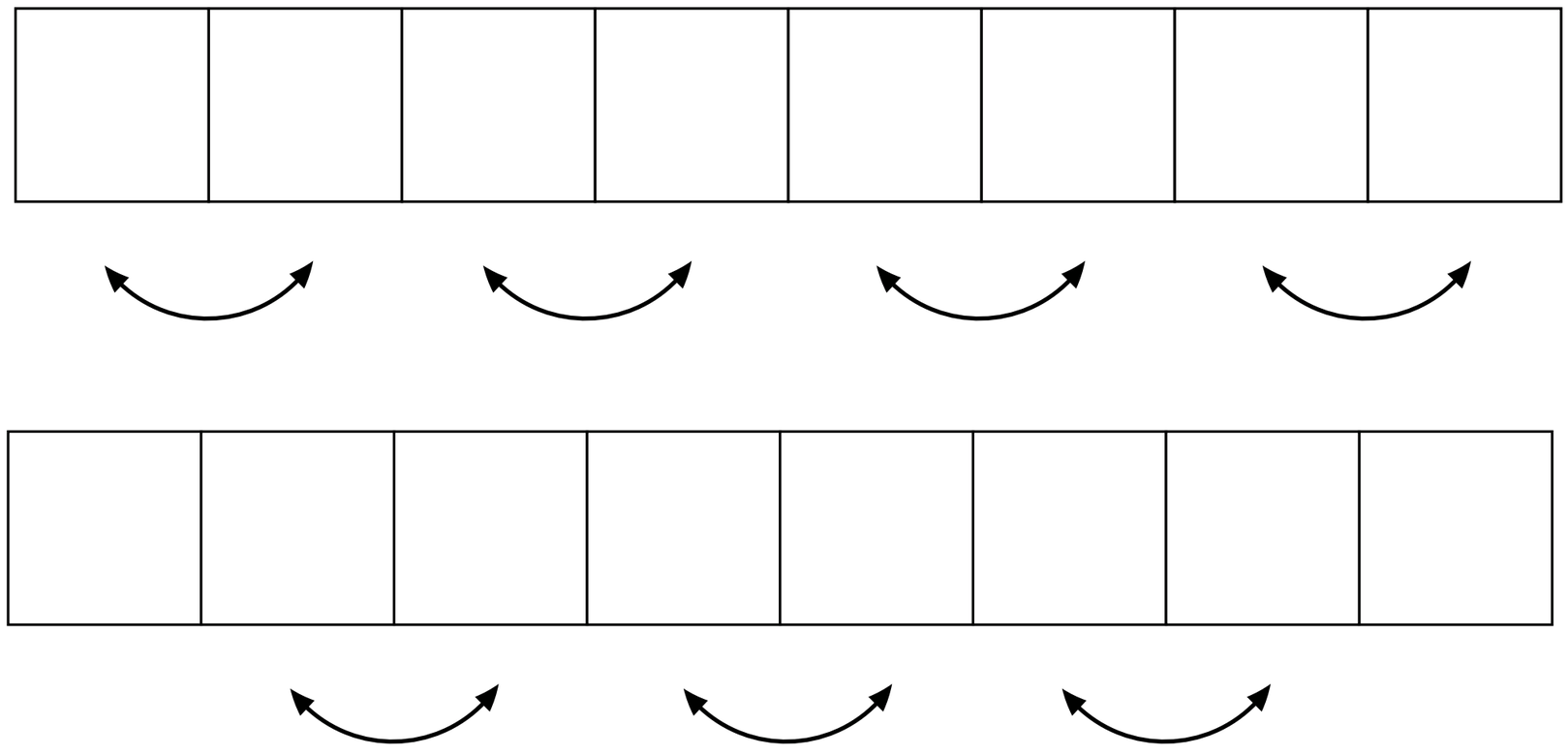
Sortering behövs i många fall!

Partiklar med semitransparens!

Sortera på avstånd från kameran

- QuickSort olämplig
- Bitonic Merge Sort bättre
- Parallell bubblesort i några steg per frame kan vara ett bättre val!

Parallell bubblesort: Jämför parvis



Trick för att slippa sortera

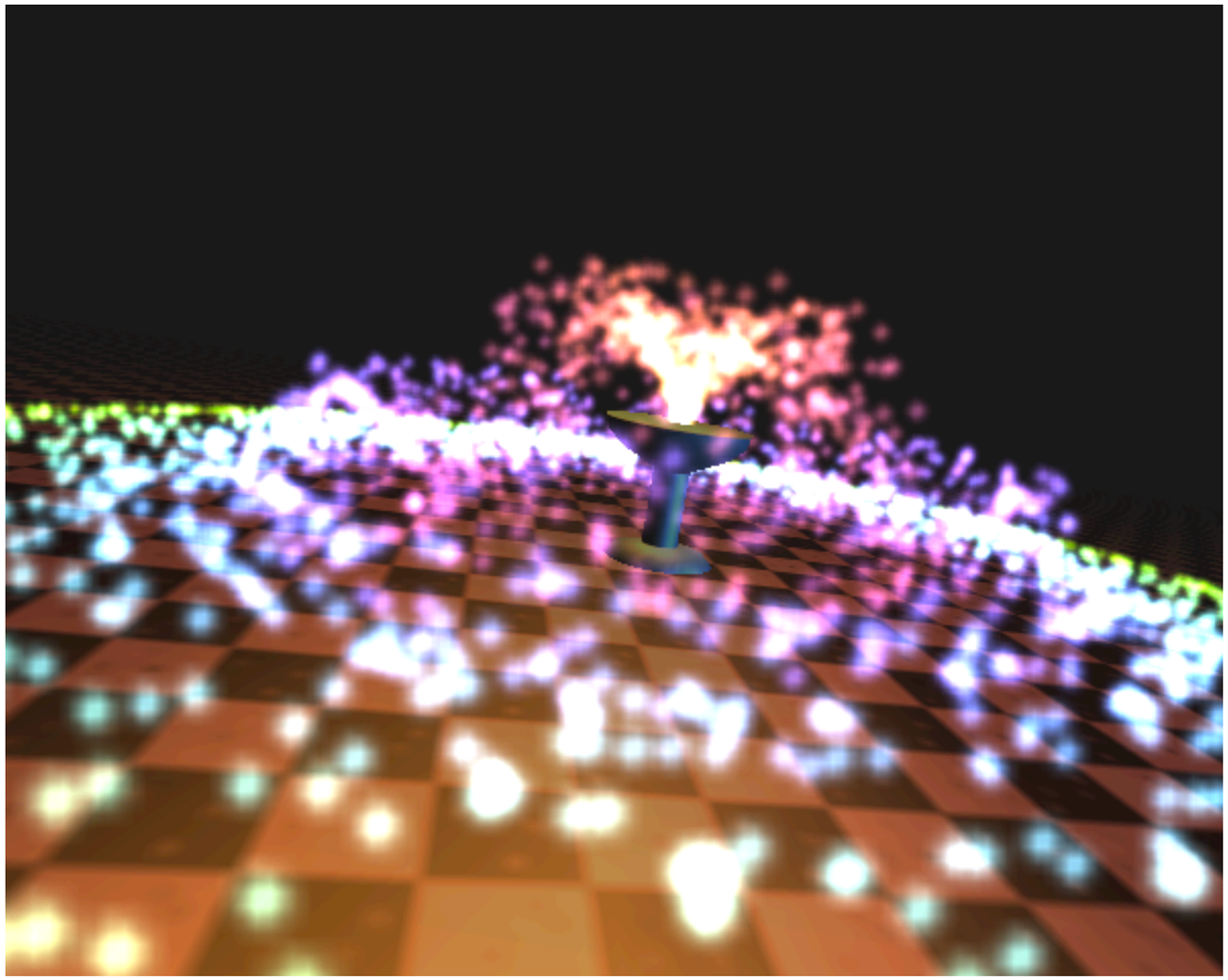
1) discard()

Bäst med skarpa kanter i textur

2) additiv blending

Fungerar för partklar utan detalj

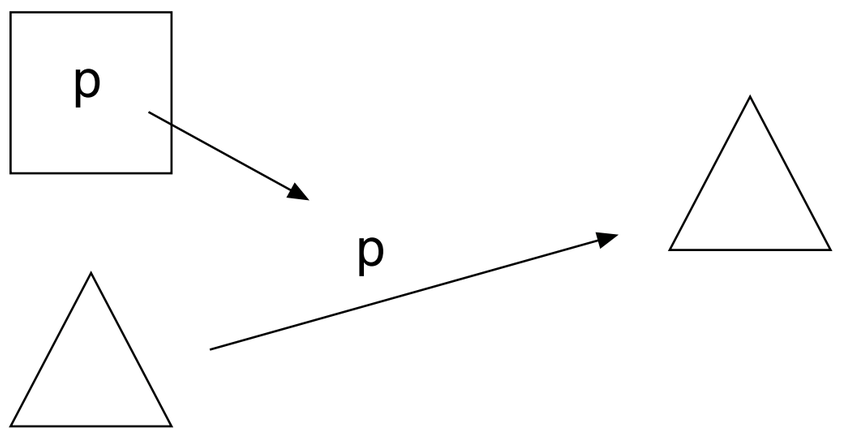
Gammalt exempel med additiv blendning



Rendering med instancing:

Beräkna position i textur från instancing-index

Sätt position (vertexshader) på billboard från position läst ur textur



Kollisionsdetektering för partikelsystem

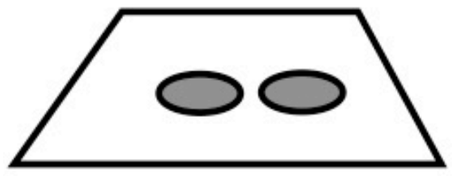
Inom systemet: Svårt problem - många till många

- Dela upp i celler! Octrees möjligt, uniform uppdelning ofta effektivare.

- Test mot omgivning: Kan förenklas t.ex. med Z-buffer!

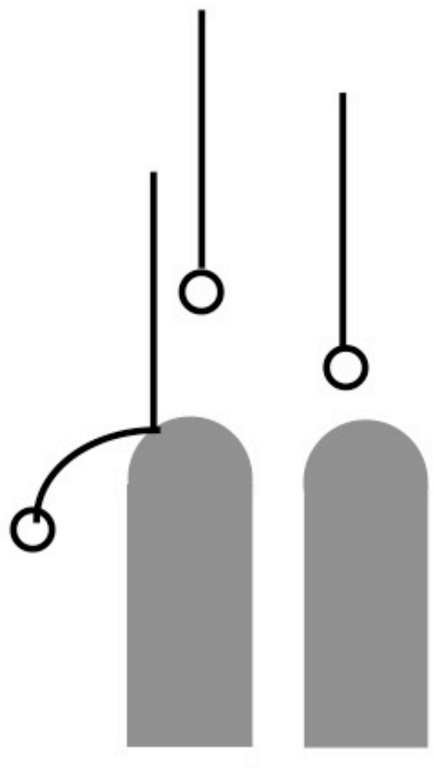
Exempel: Regn/snö mm

Kamera för att rendera Z-buffer uppifrån



Z-buffer

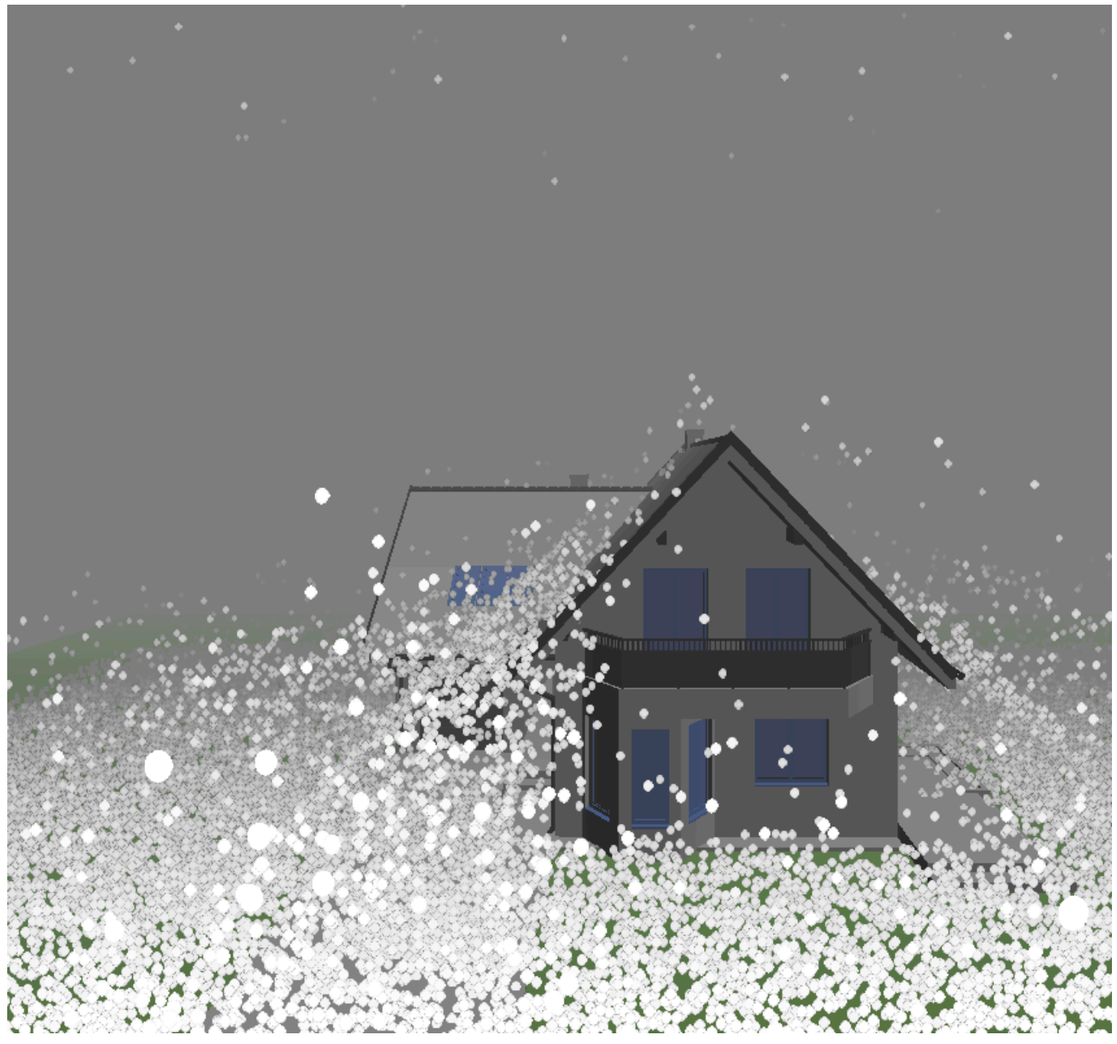
Scen



Partiklar kan nu studsa av objekt enbart genom att inspektera Z-buffern!

Projekt: Hailstorm

Partikelsystem i shaders med instancing



Kollision med omgivning med Z-buffern

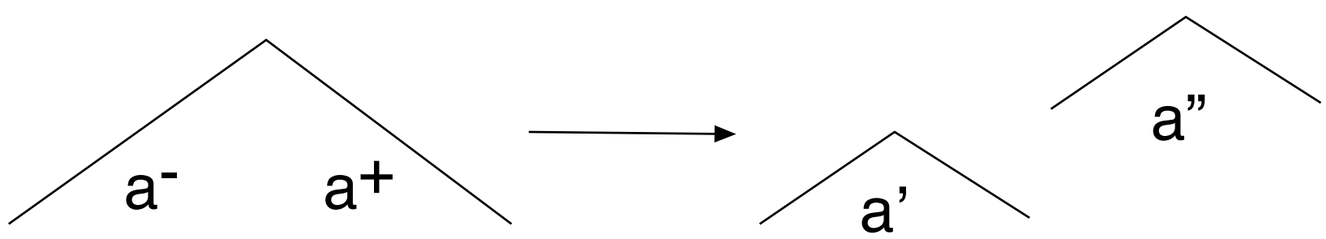
Sammanfattning:

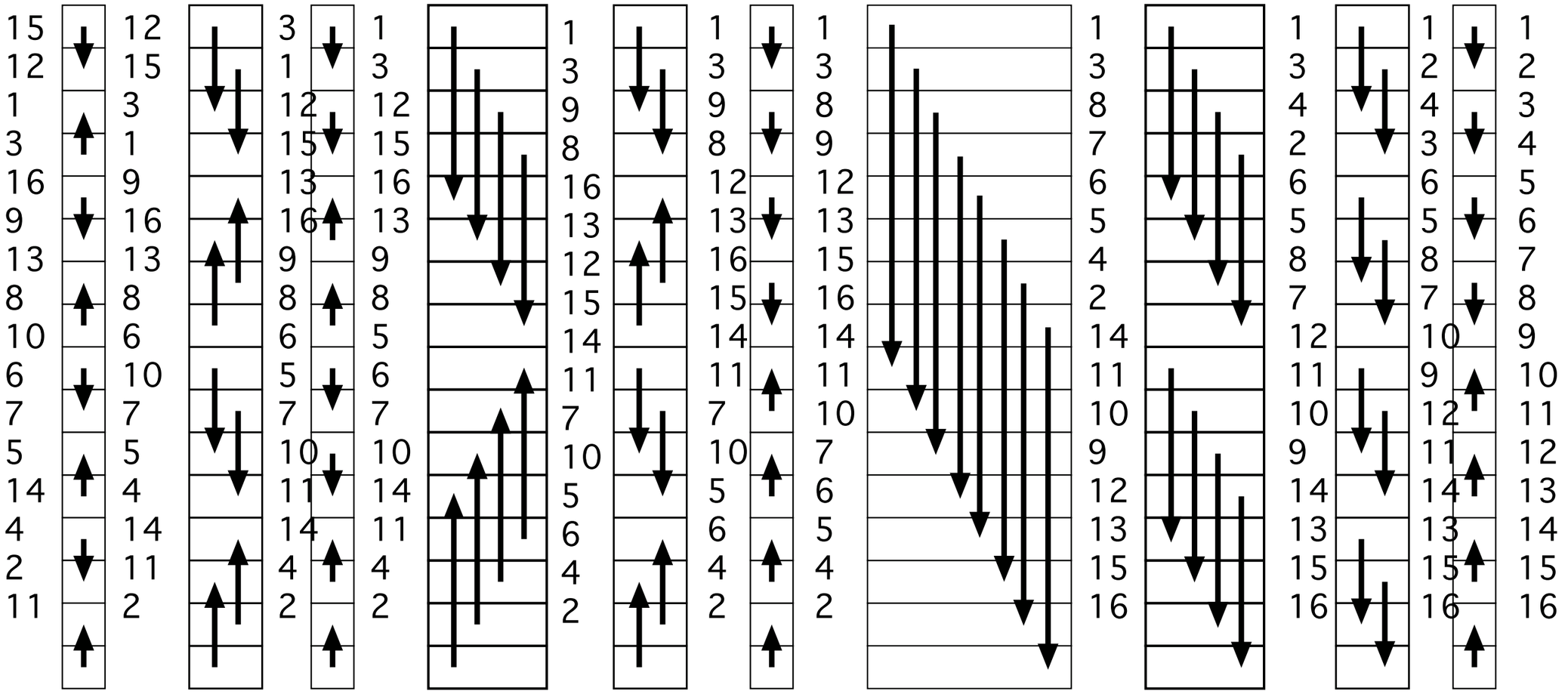
- Lagra partikeldata i texturer, en per texel
- Bind texturerna till FBOer
- Uppdatera genom rendering med ping-ponging
- Kollisionsdetektering
- Sortering (om det behövs)
- Rendera partiklar med instancing

Sortering

Algoritm passande för GPU: Bitonic Merge Sort

Baseras på att en bitonic mängd kan delas i två som inte överlappar:




15 12 3 1 1 1 1 1
12 15 1 3 3 3 3 3
1 3 12 12 9 9 8 8
3 1 15 15 8 8 9 9
16 9 13 16 16 12 12 12
9 16 16 13 13 13 13 13
13 13 9 9 12 16 15 15
8 8 8 8 15 15 16 16
10 6 6 5 14 14 14 14
6 10 5 6 11 11 11 11
7 7 7 7 7 7 10 10
5 5 10 10 10 10 7 9
14 4 11 14 5 5 6 12
4 14 14 11 6 6 5 13
2 11 4 4 4 4 4 15
11 2 2 2 2 2 2 16
1 1 1 1 1 1 1 1
3 3 4 4 2 2 3 3
1 2 4 3 3 3 4 4
2 3 3 4 4 4 5 5
3 4 2 6 6 6 6 6
4 5 6 5 8 8 7 7
5 6 8 7 7 7 8 8
6 7 7 8 10 9 9 9
7 8 9 10 9 12 11 10
8 9 12 11 11 11 10 11
9 10 11 12 12 14 13 12
10 11 13 13 14 13 15 13
11 12 14 14 15 15 16 14
12 13 15 15 16 16 14 15
13 14 16 16 13 10 12 16

Reduktion

Att extrahera lite information från en hel bild.

Exempel: Max, min, medelvärde

Kan paralleliseras i shaders.

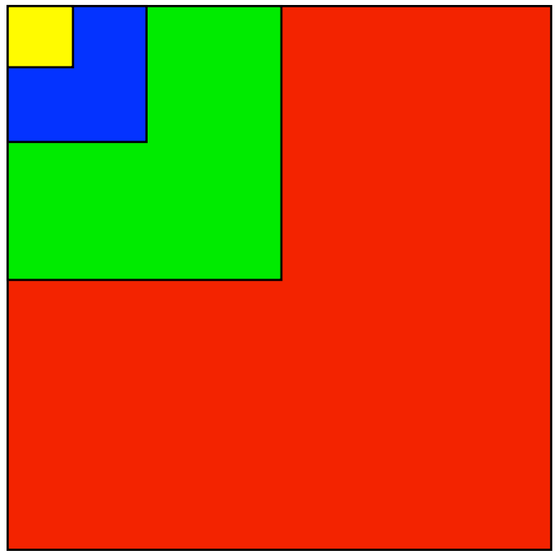
Ta ut resultatet med glReadPixels som en eller ett fåtal pixlar

2D passar shaders väl, typiskt 4-to-1 per pass

Pyramidhierarki

47	2	3	57	5	12	7	8
10	20	6	13	14	15	16	17
19	11	21	22	23	68	25	26
38	29	64	31	32	33	35	34
37	28	39	49	53	42	41	52
46	1	48	40	61	51	44	43
55	71	4	58	69	62	50	60
30	65	66	67	24	59	70	56

47	57	15	17
38	64	68	35
46	49	61	52
71	67	69	70



if-satser i shaders

Villkorlig kod kan vara en prestandasänkare

All kod kan exekveras!

Orsak: Shaders körs i *warps*, parallellt, på SIMD-vis.

Gör närliggande trådar olika beslut?