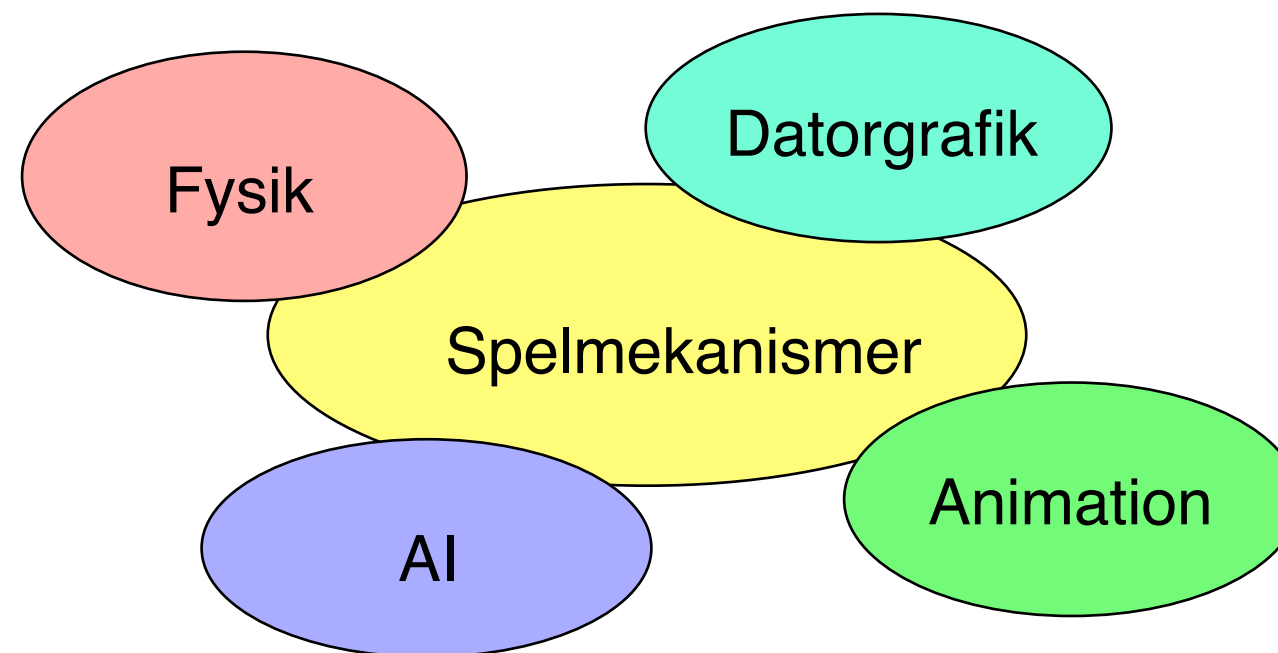




TSBK 03

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!

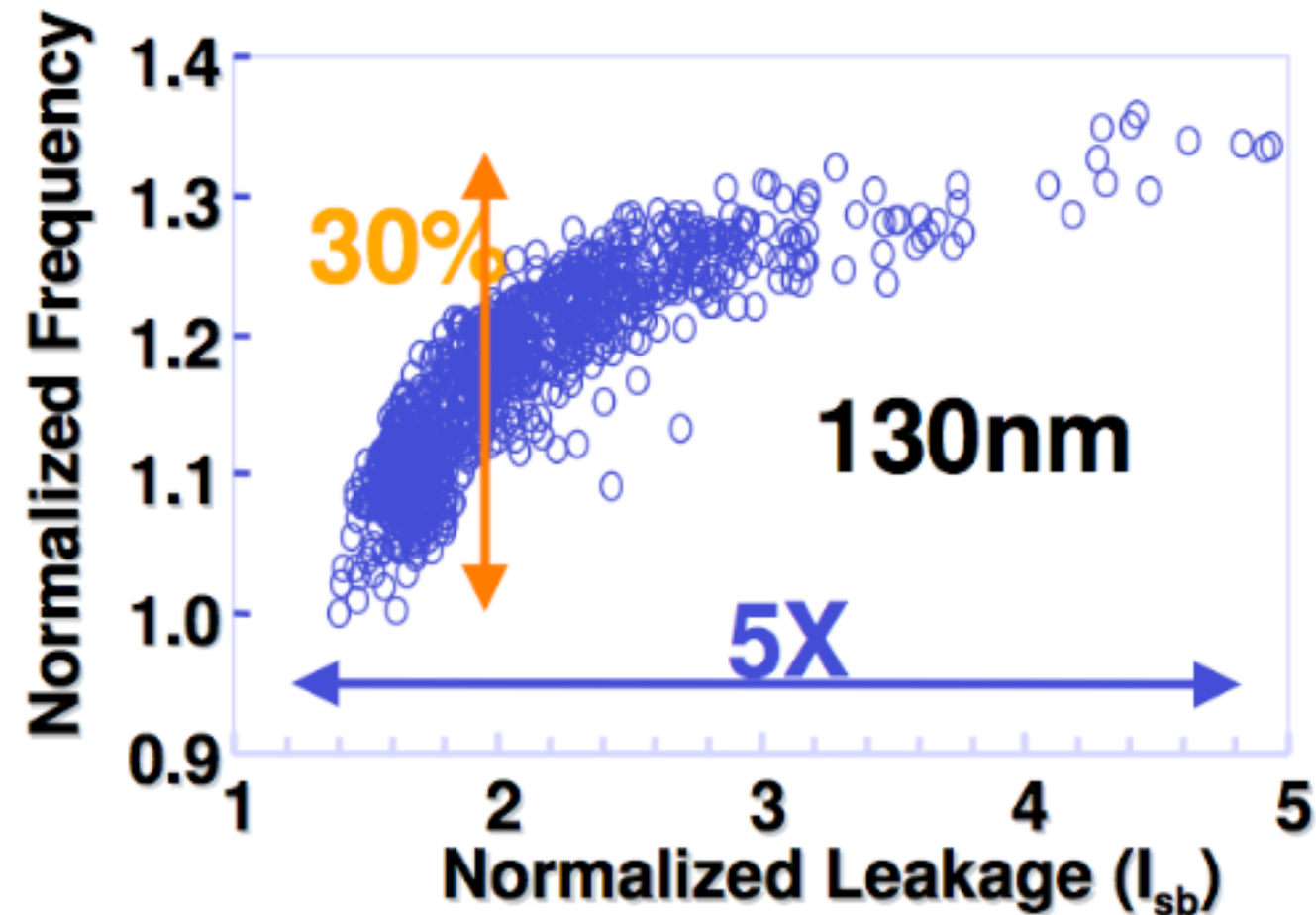
Argument för: GPU'erna uppvisar stor prestandaökning medan CPU'er håller på att avstanna.

Nyckelkomponenter: Shaders, flyttalsbuffrar, parallellism.



Power wall

Vi kan inte längre höja frekvensen. Effekten stiger brant.



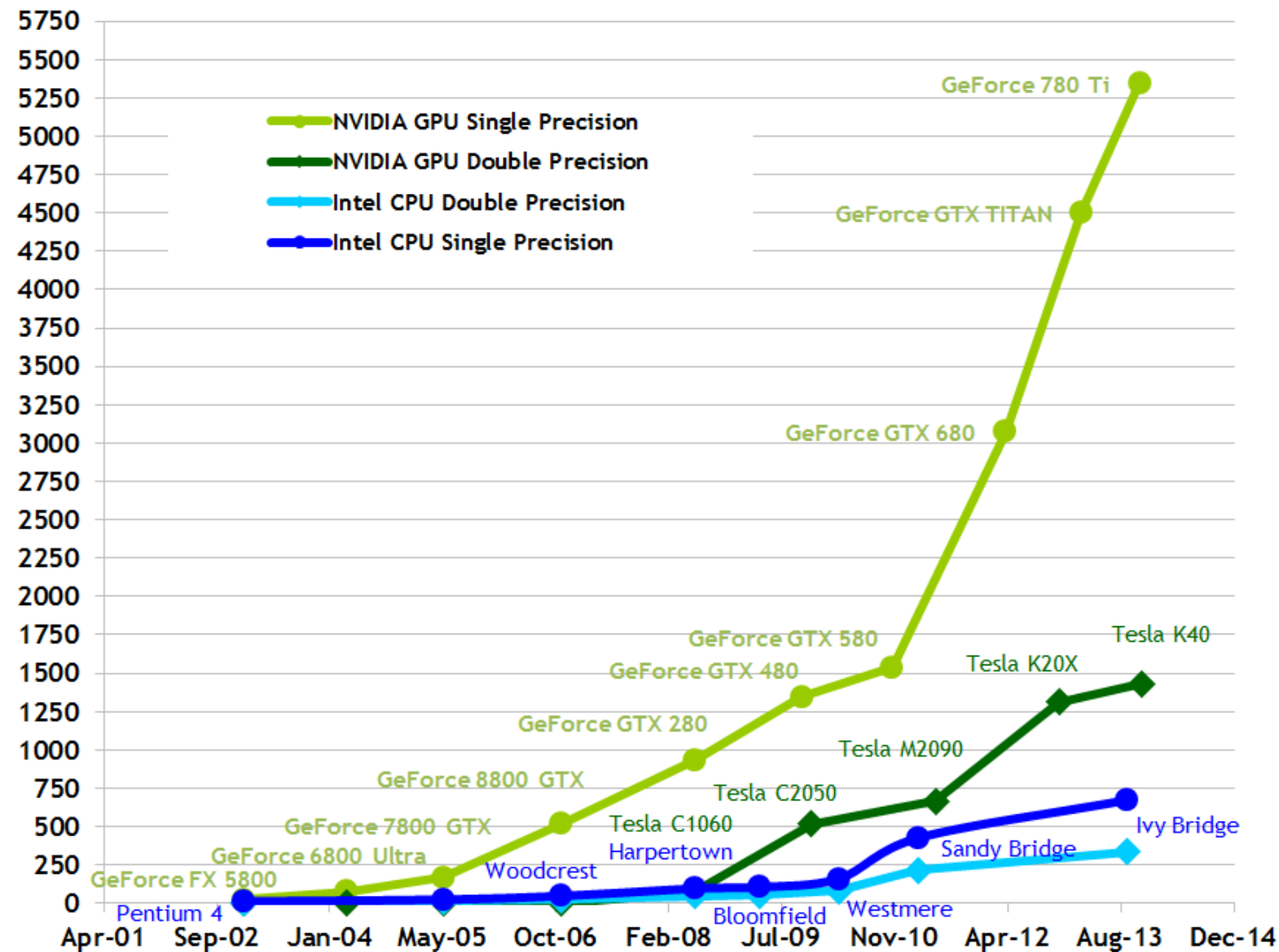
Men vi kan öka ANTALET!



Information Coding / Computer Graphics, ISY, LiTH

The GFLOPS race

Theoretical GFLOP/s



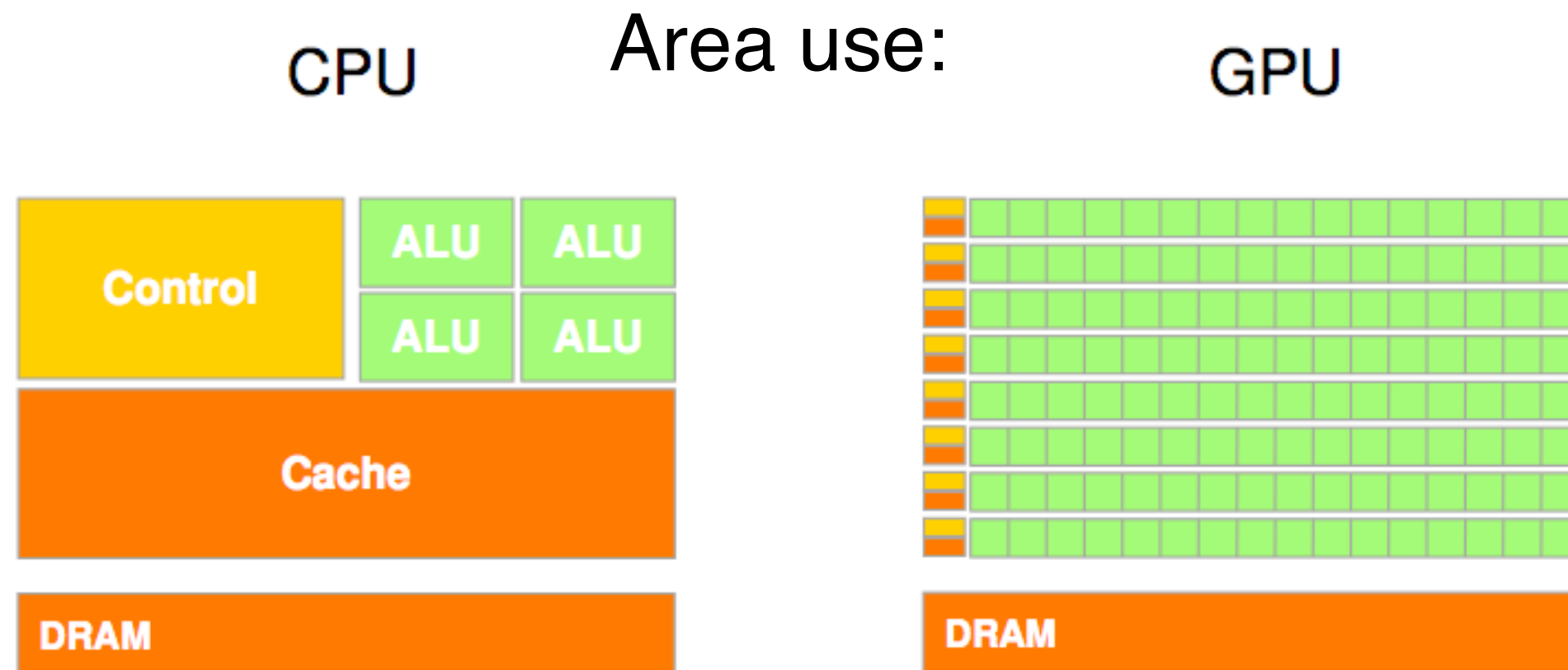


Information Coding / Computer Graphics, ISY, LiTH





Hur är detta möjligt?



Men i synnerhet: SIMD architecture, många små
processorer



Typisk ”klassisk” GPGPU: som filtreringsdelen i labb 1, HDR

Samma grundkoncept:

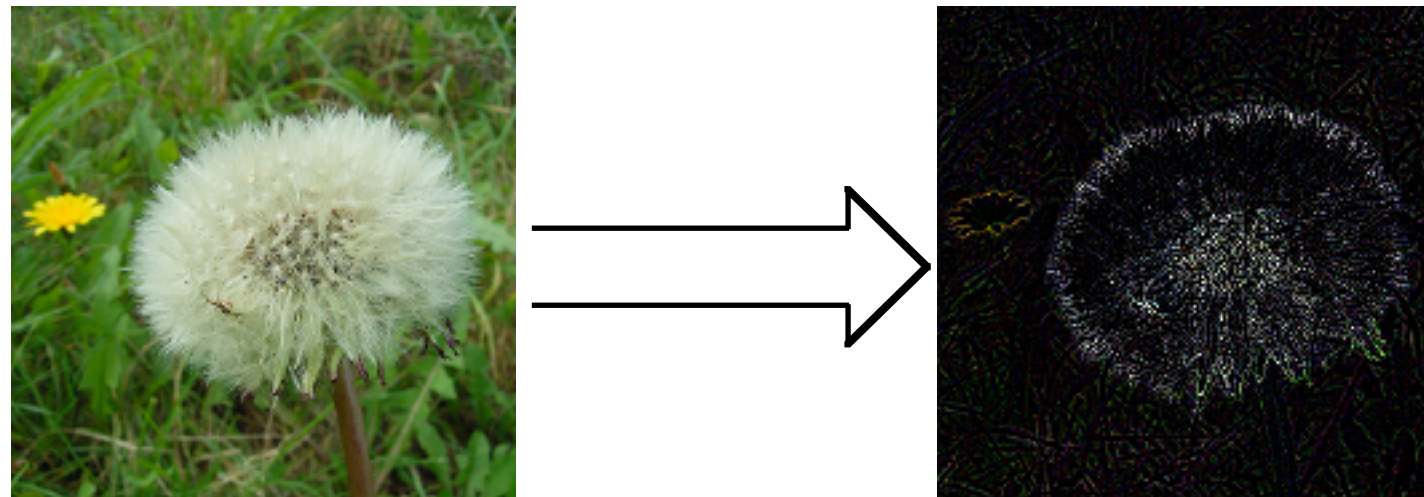
- Rendera till rektangel över hela bilden
 - Gärna FBOs
 - Flyttalsbuffrar
- Ping-ponging, flera pass med olika shaders



Enklaste GPGPU: Processing 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 renderingsituation:

- 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

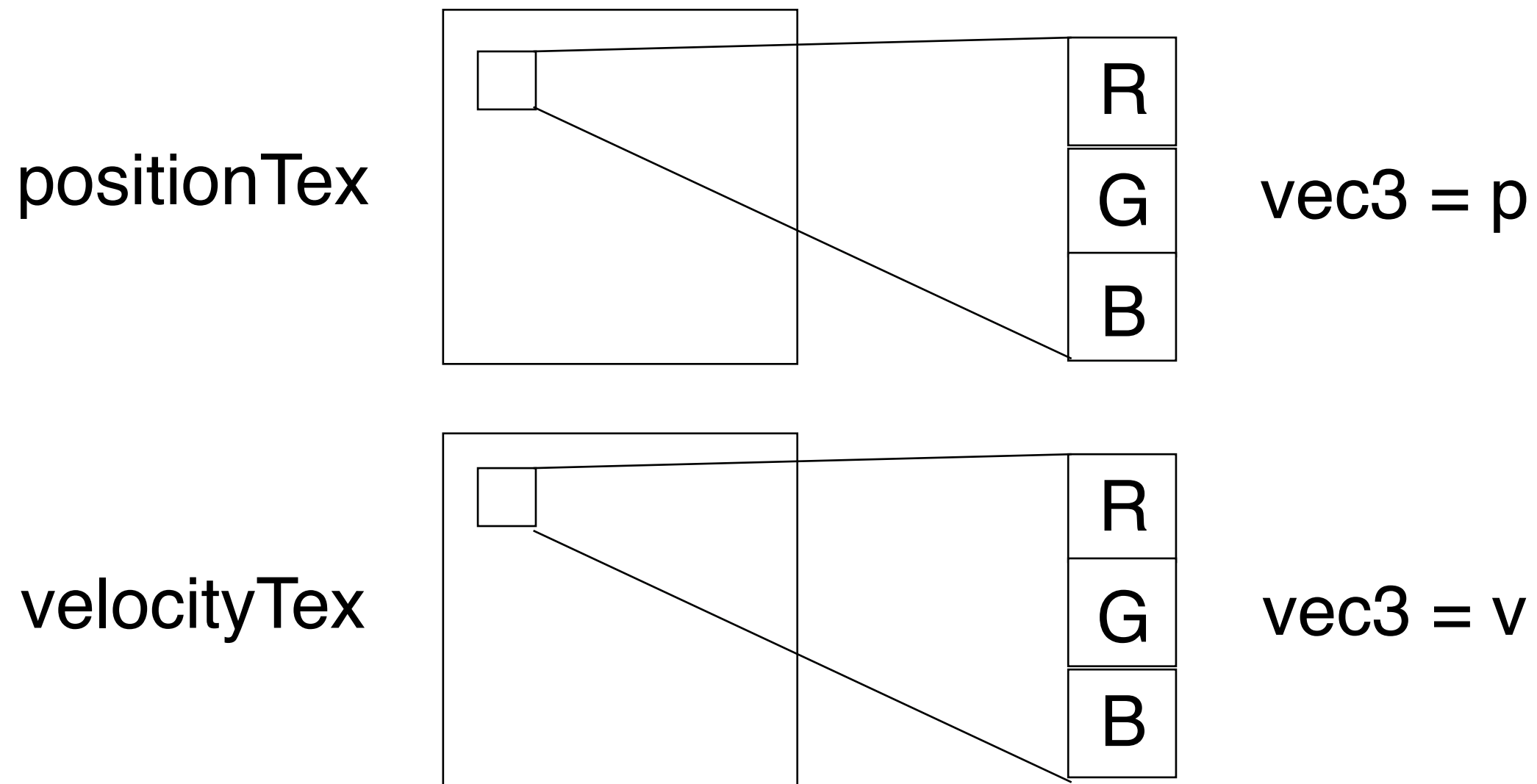
Uppdatering:

$$\text{Position } p = p + v * dt$$

$$\text{Hastighet } v = v + a * dt$$



Lagra data i texturer!





Dubbla texturer för ping-ponging

positionTex1

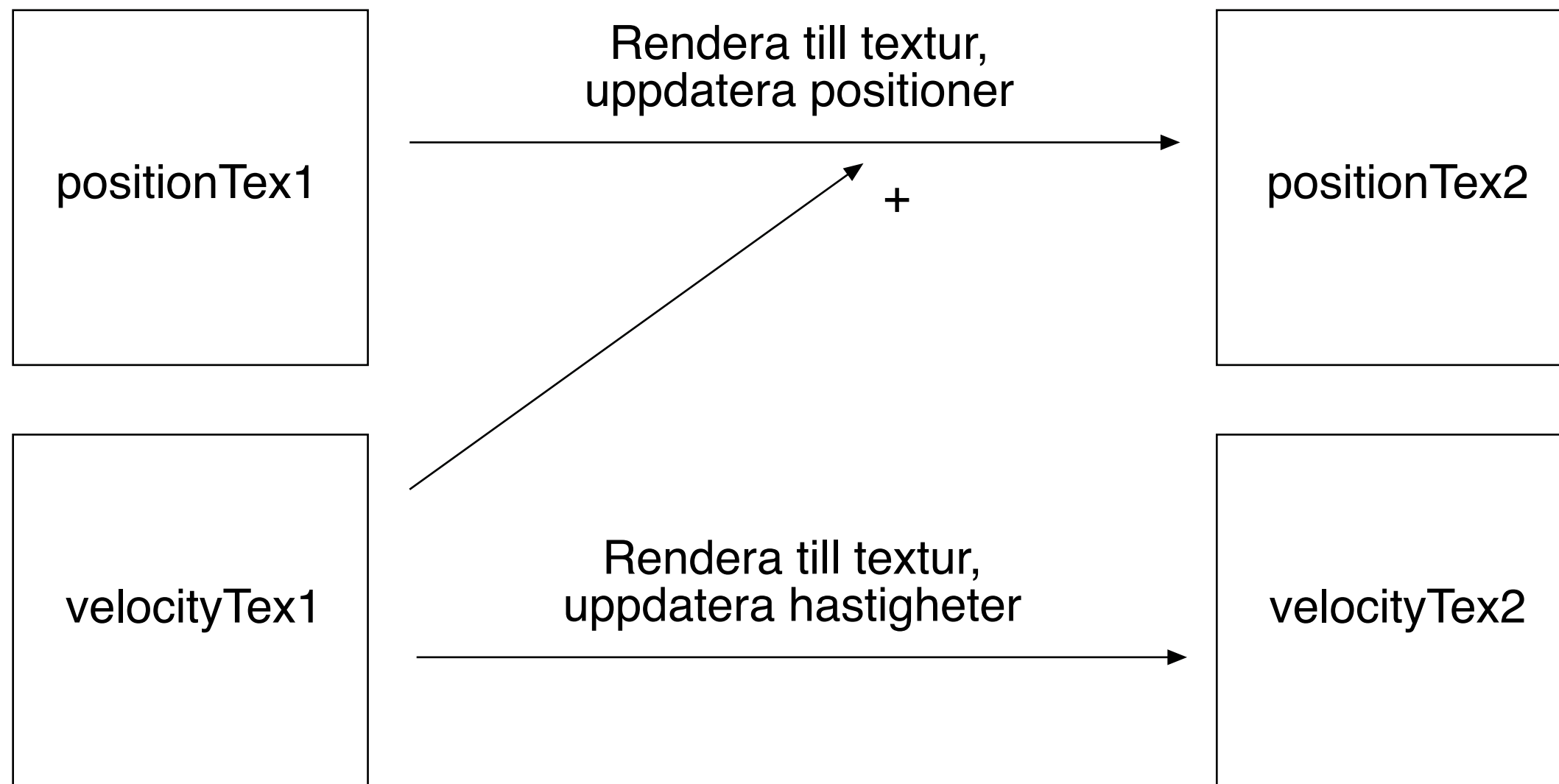
positionTex2

velocityTex1

velocityTex2



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!

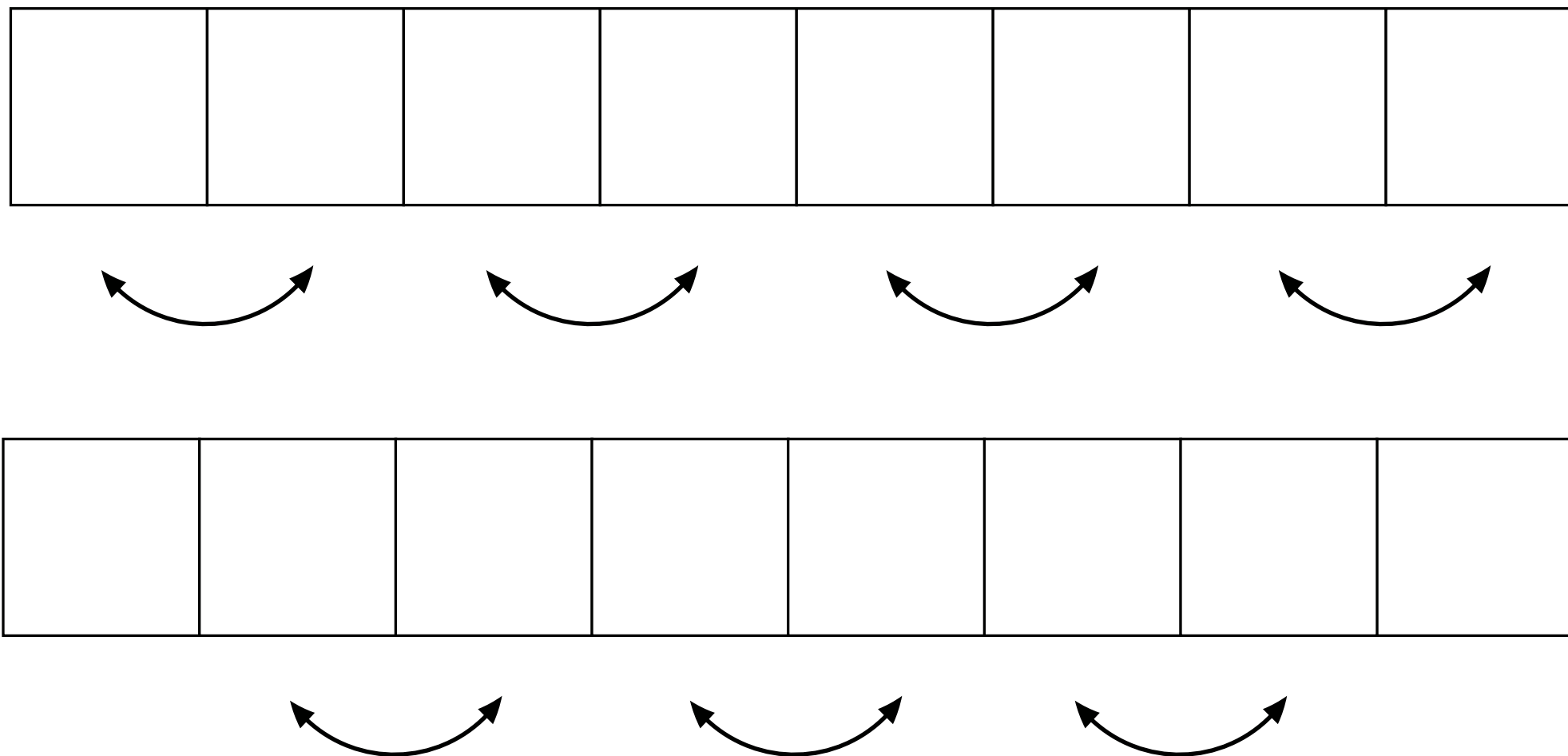
Partiklar med semitransparens!

Sortera på avstånd från kameran

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



Parallel "odd-even" bubblesort: Jämför parvis





Trick för att slippa sortera

1) discard()

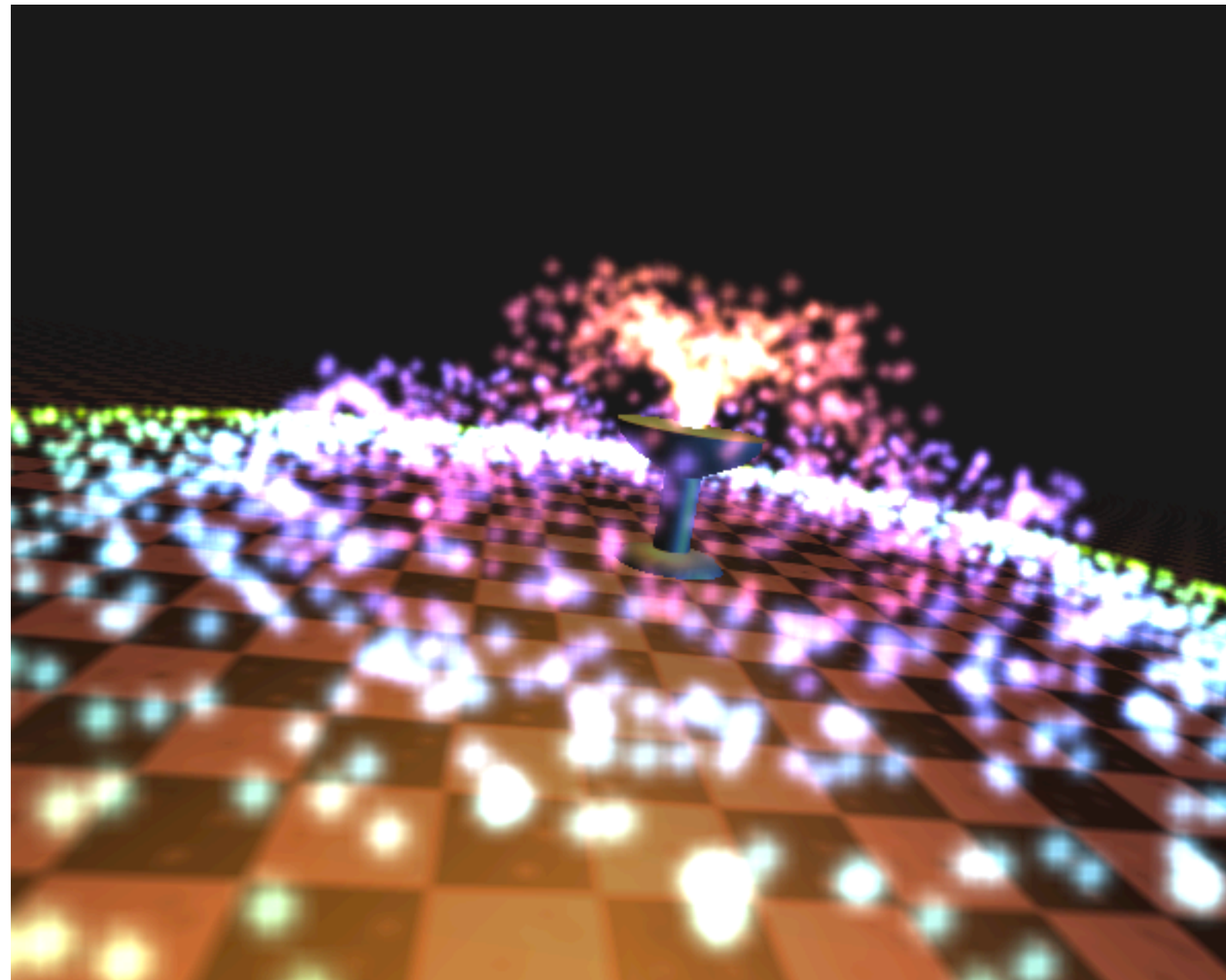
Bäst med skarpa kanter i textur

2) additiv blending

Fungerar för partklar utan detalj



Gammalt exempel med additiv blendning

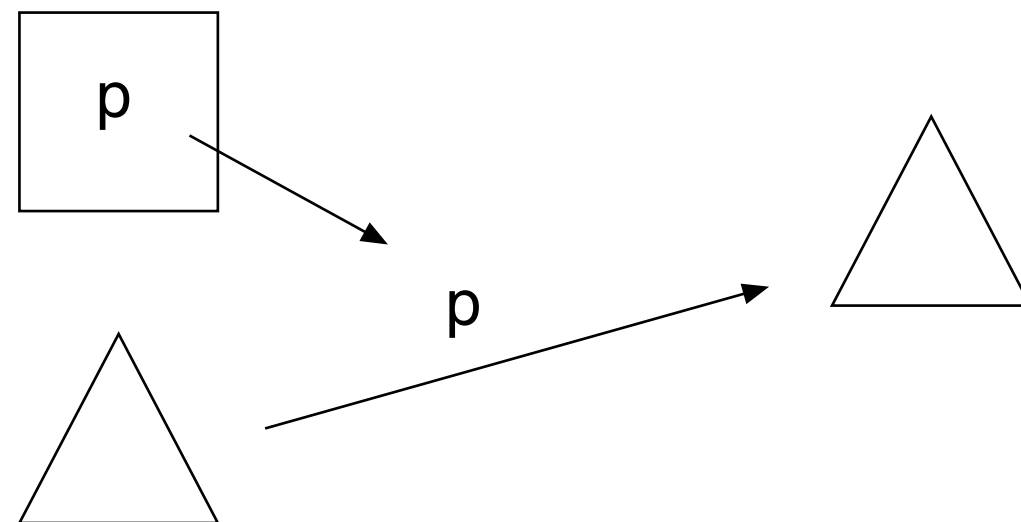




Rendering med instancing:

Beräkna position i textur från instancing-index

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





Kollisionsdetektering för partikelsystem

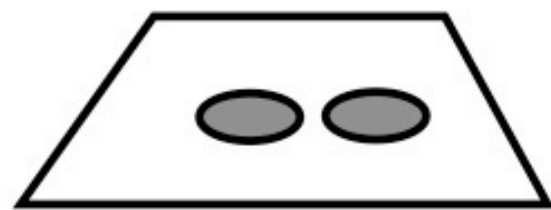
Inom systemet: Svårt problem - många till många

- Dela upp i celler! Octrees möjligt, uniform uppdelning ofta effektivare.
- Test mot omgivning: Kan förenklas t.ex. med Z-buffer!



Exempel: Regn/snö mm

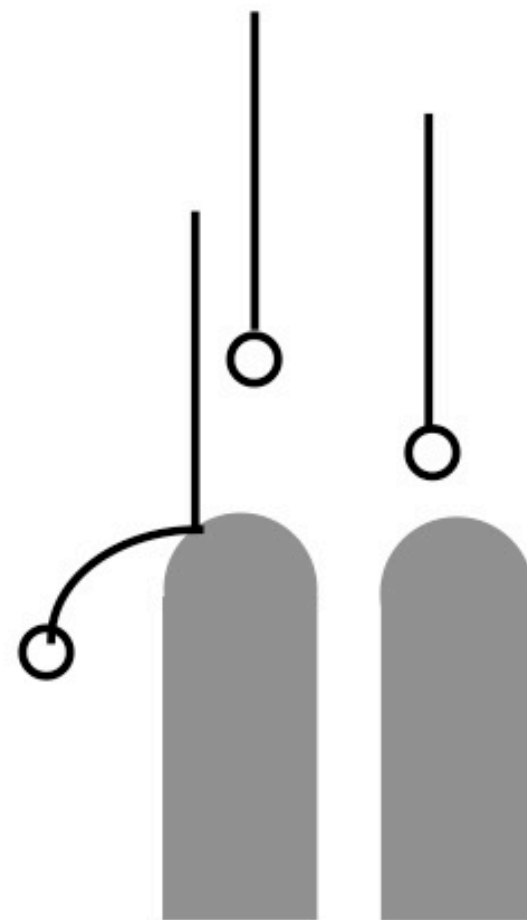
A Kamera för att rendera Z-buffer uppifrån



Z-buffer



Scen

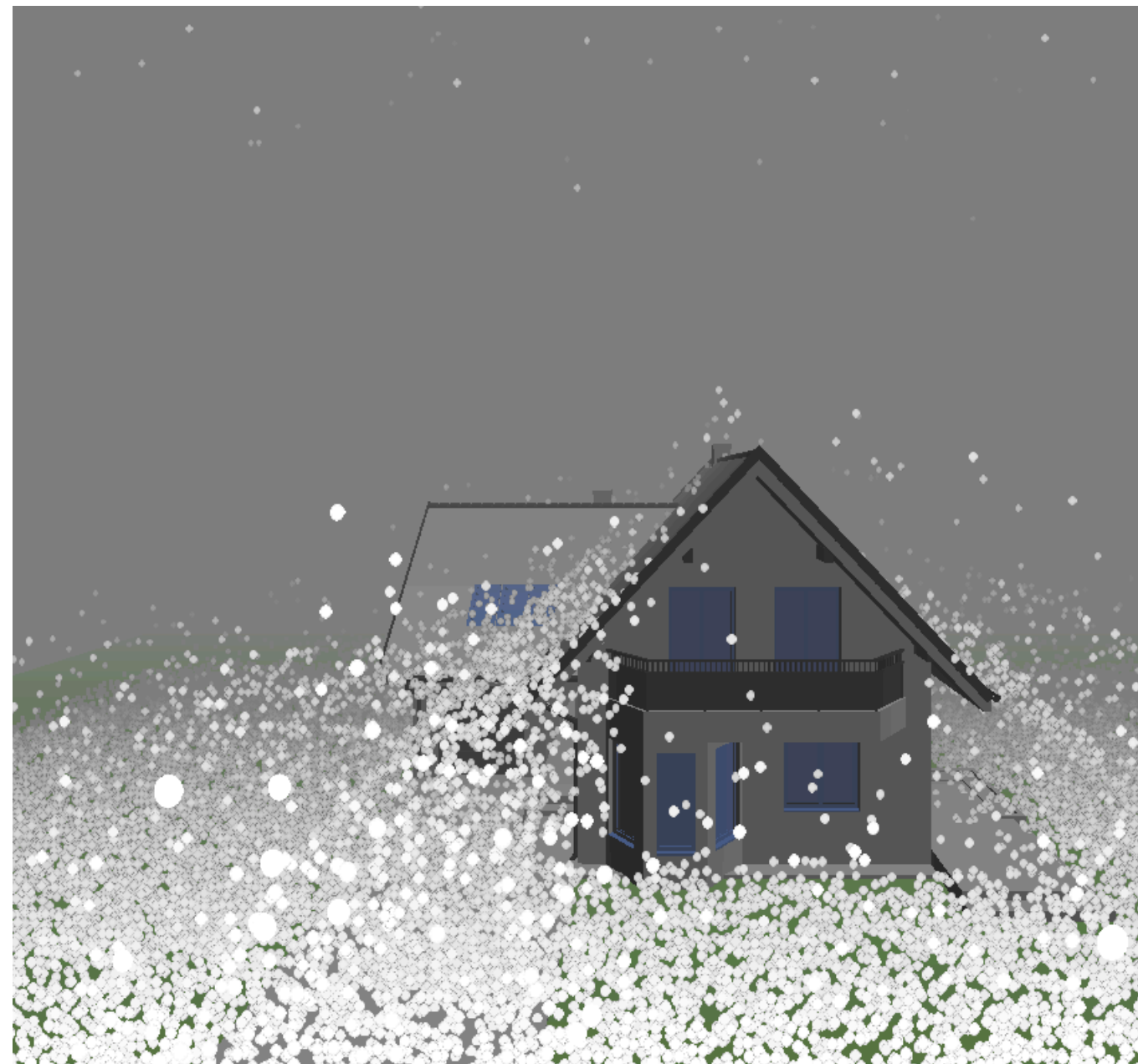


Partiklar kan nu studsas av objekt enbart genom att inspektera Z-bufferen!



Projekt: Hailstorm

Partikelsystem
i shaders med
instancing



Kollision med
omgivning
med Z-buffern



Sammanfattning:

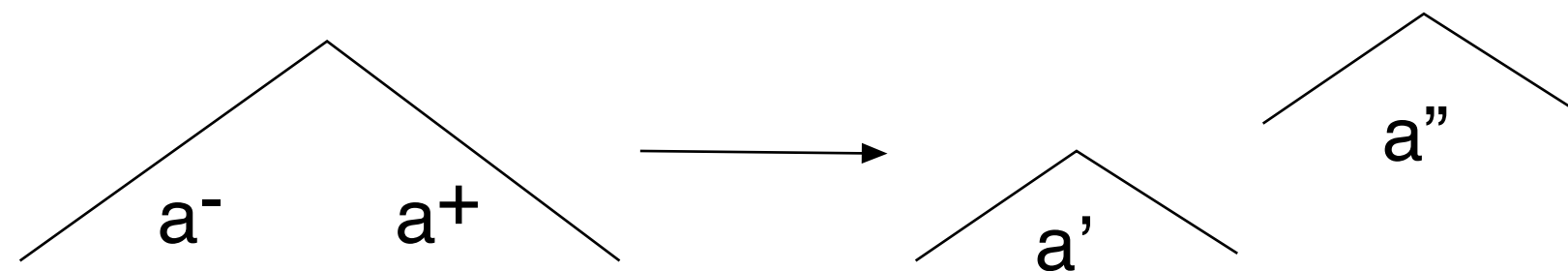
- Lagra partikeldata i texturer, en per texel
 - Bind texturerna till FBOer
- Uppdatera genom rendering med ping-ponging
 - Kollisionsdetektering
 - Sortering (om det behövs)
 - Rendera partiklar med instancing



Sortering

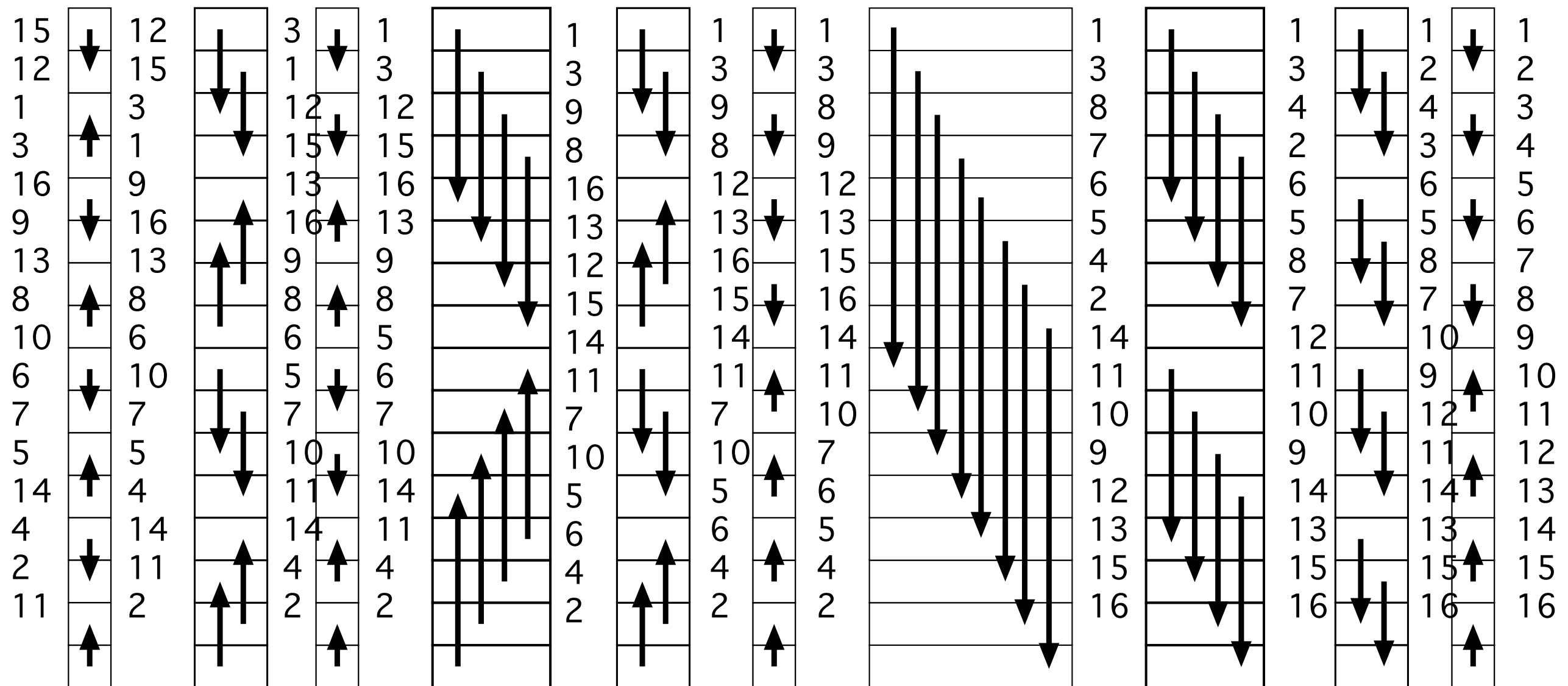
Algoritm passande för GPU: Bitonic Merge Sort

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





Information Coding / Computer Graphics, ISY, LiTH





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



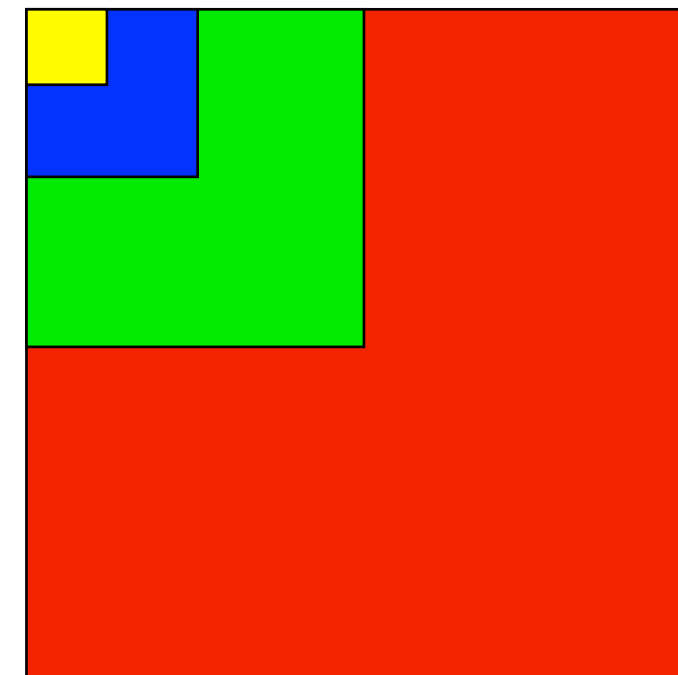
2D passar shaders väl,
typiskt 4-to-1 per pass

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?