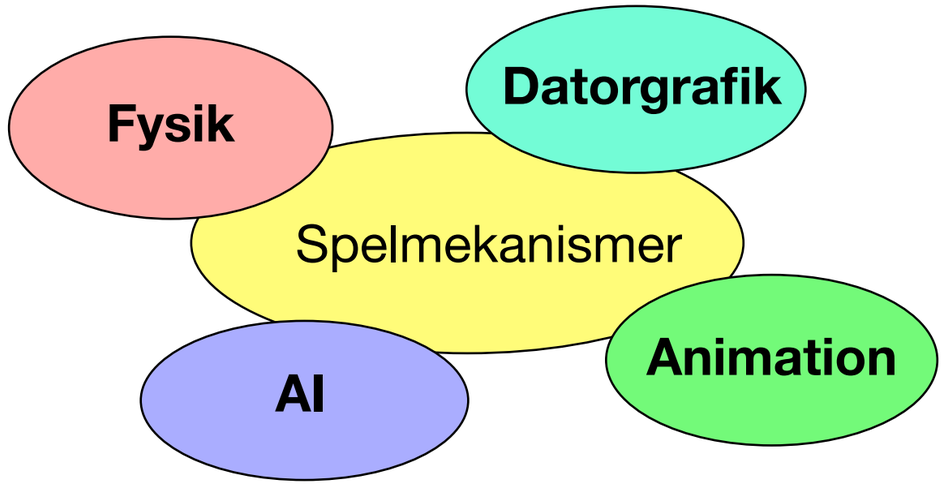
TSBK 03
Teknik för avancerade datorspel

Ingemar Ragnemalm, ISY



Föreläsning 6

GPU computing

- GPU computing, vad och varför
- Partikelsystem i shaders
- Compute Shaders
- Intro till CUDA

Föreläsningsfrågor

1. När är GPUer mycket snabbare än en CPU?

2. Vad har texturerna för roll för partikelsystemssimuleringen?

3. Vad för slags problem bör köras på GPU?

4. Hur kan partikelsystem interagera med omgivningen?

5. Varför blev transponering snabbare med shared memory?

GPU Computing
Generella beräkningar på GPUs som inte nödvändigtvis är bilder

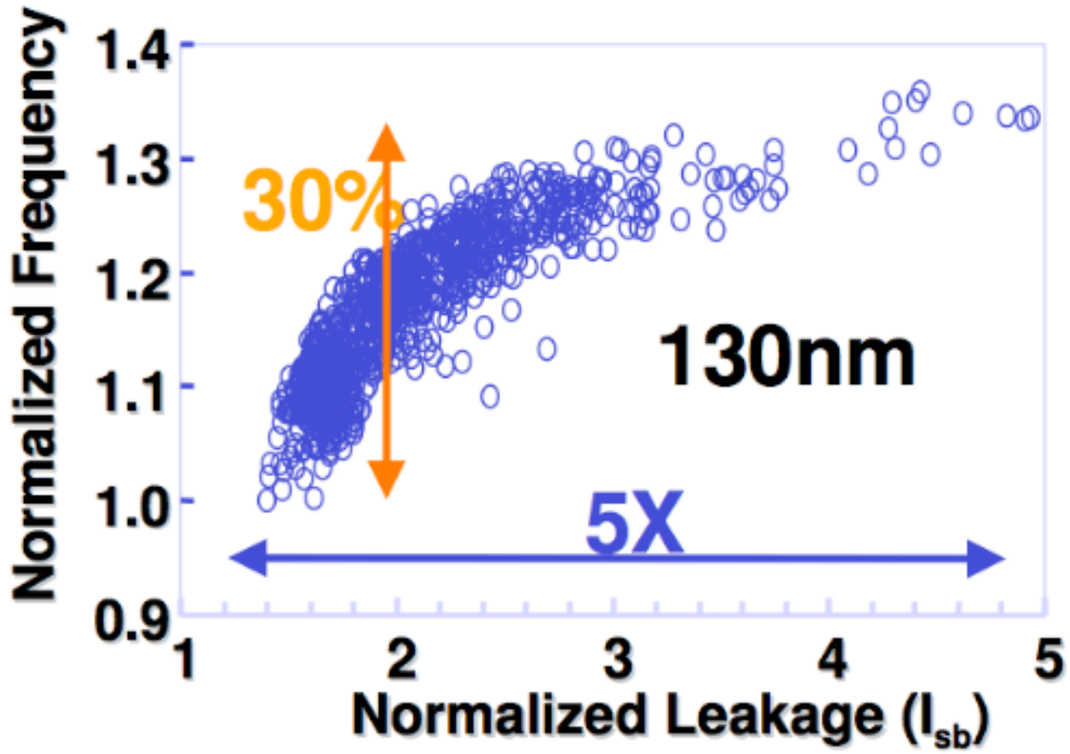
Använd GPU’ns beräkningskraft för andra problem än grafik!

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!

Topp-GPUs 163 TFLOPS!

Mer konsumentvänligt: Ca 44 TFLOPS

CPU: 576 GFLOPS

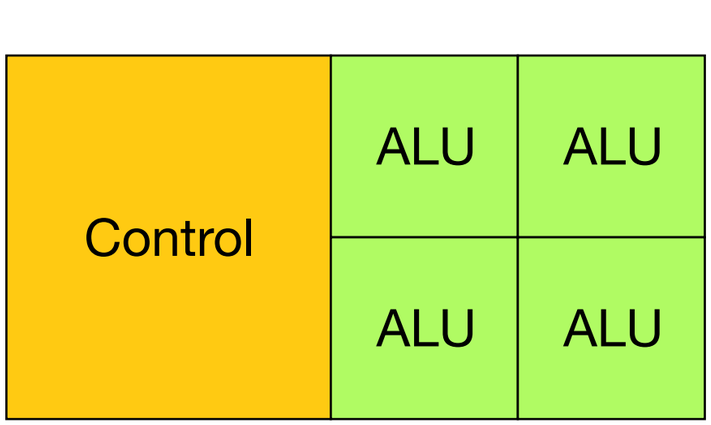
Bild borttagen av copyrightskäl

Superdatorn på ditt skrivbord

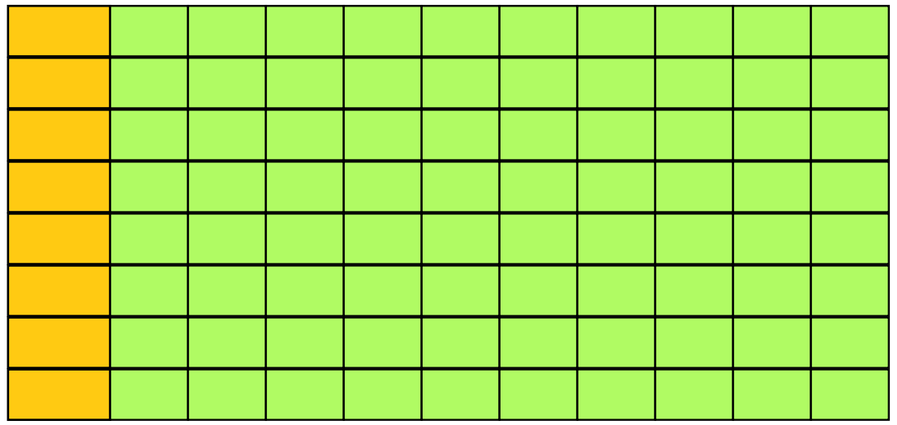
Hur är detta möjligt?

Area use:

CPU



GPU



SIMD architecture, många små processorer, alla gör samma sak! SIMD = Single Instruction, Multiple Data

Flera möjliga APIer

Klassisk GPU Computing: Fragment shaders

Compute shaders (OpenGL mfl)

CUDA (NVidia)

OpenCL (öppen)

Direct Compute (Microsoft)

Metal (Apple)

Typisk ”klassisk” GPU Computing: som filtreringsdelen i HDR

Samma grundkoncept:

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

Enklaste GPU Computing: 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.



GPU Computing-konceptet i fragment shaders

- 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

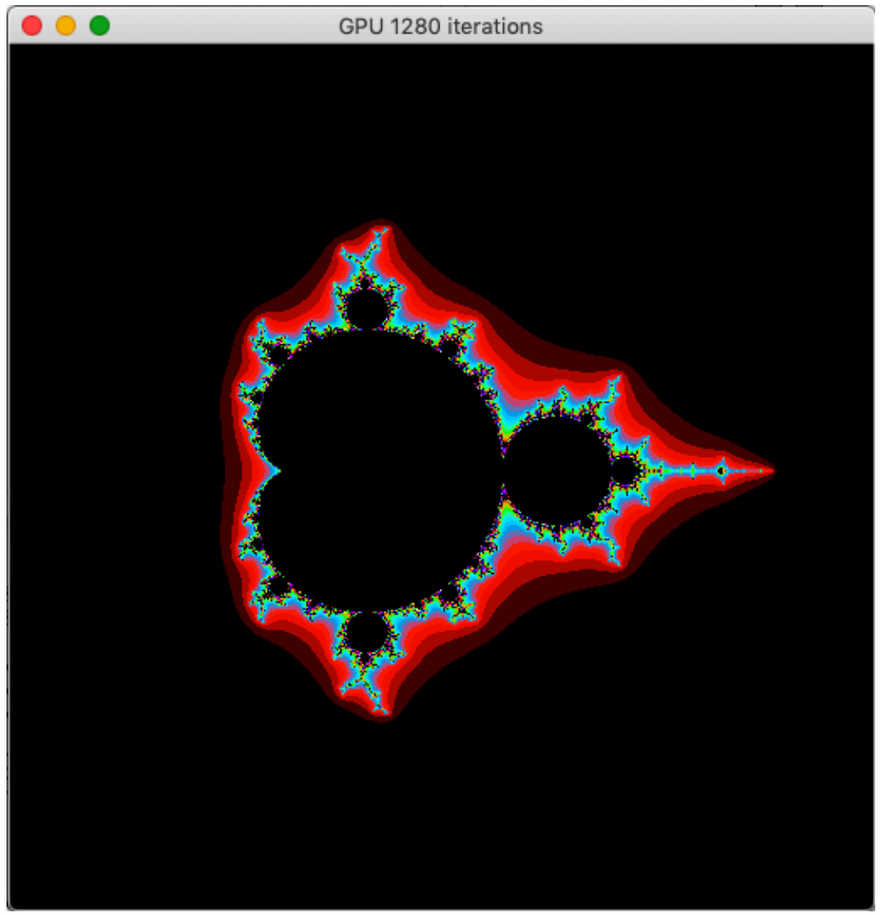
Array av data läggs i textur

```
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.

Enkelt prestandaexempel: Mandelbrot



Trivial parallelliserbart problem!
(Nästan.)

Shader-baserad GPU Computing 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

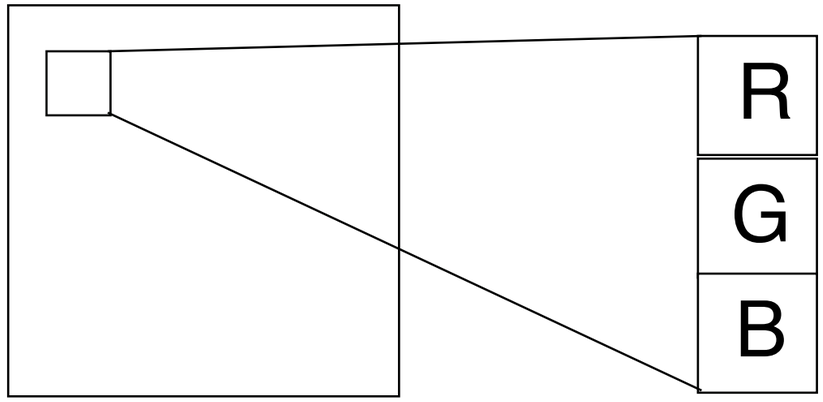
Uppdatering:

Position p = p + v * dt

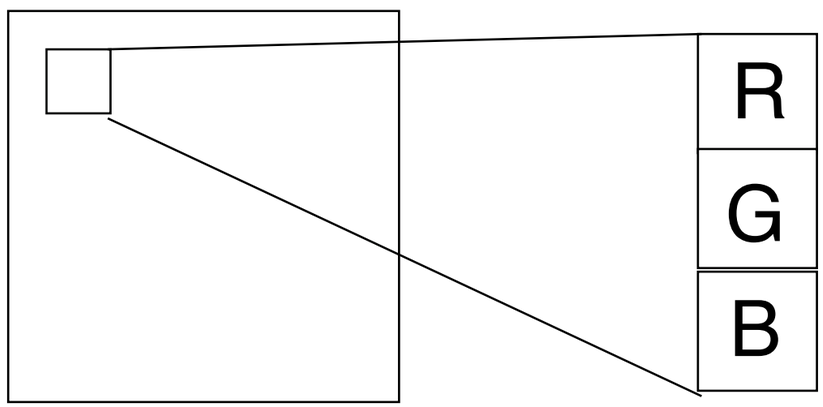
Hastighet v = v + a * dt

Lagra data i texturer!

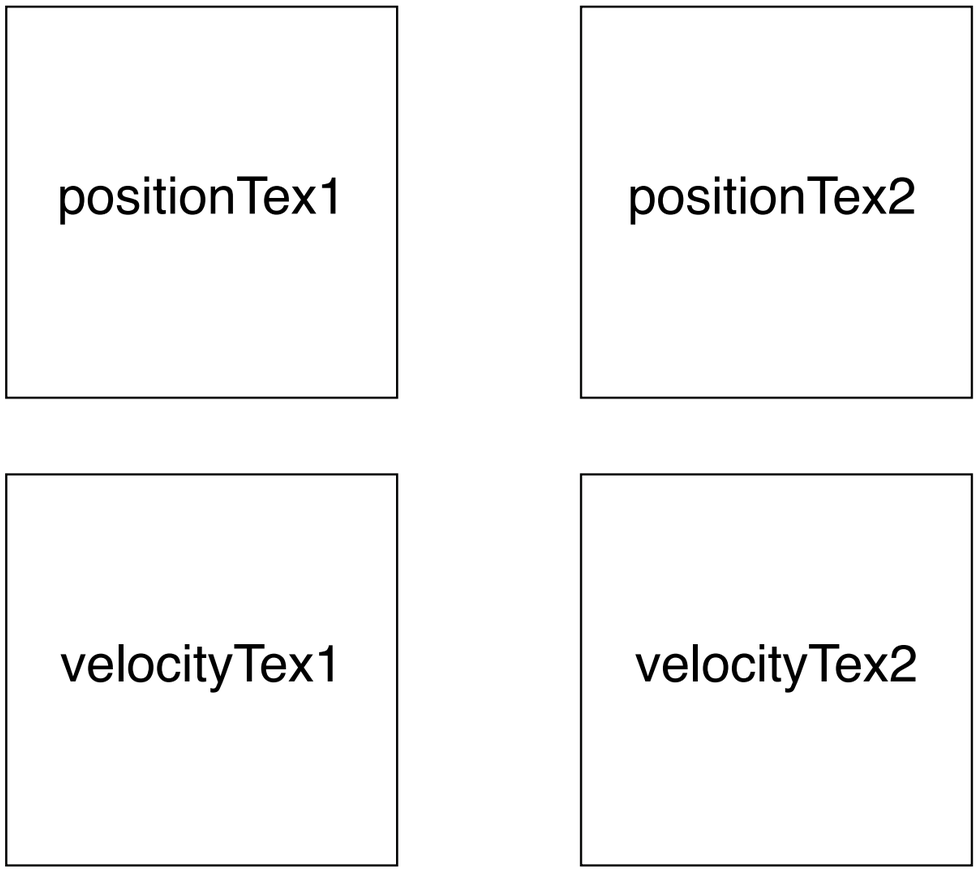
positionTex



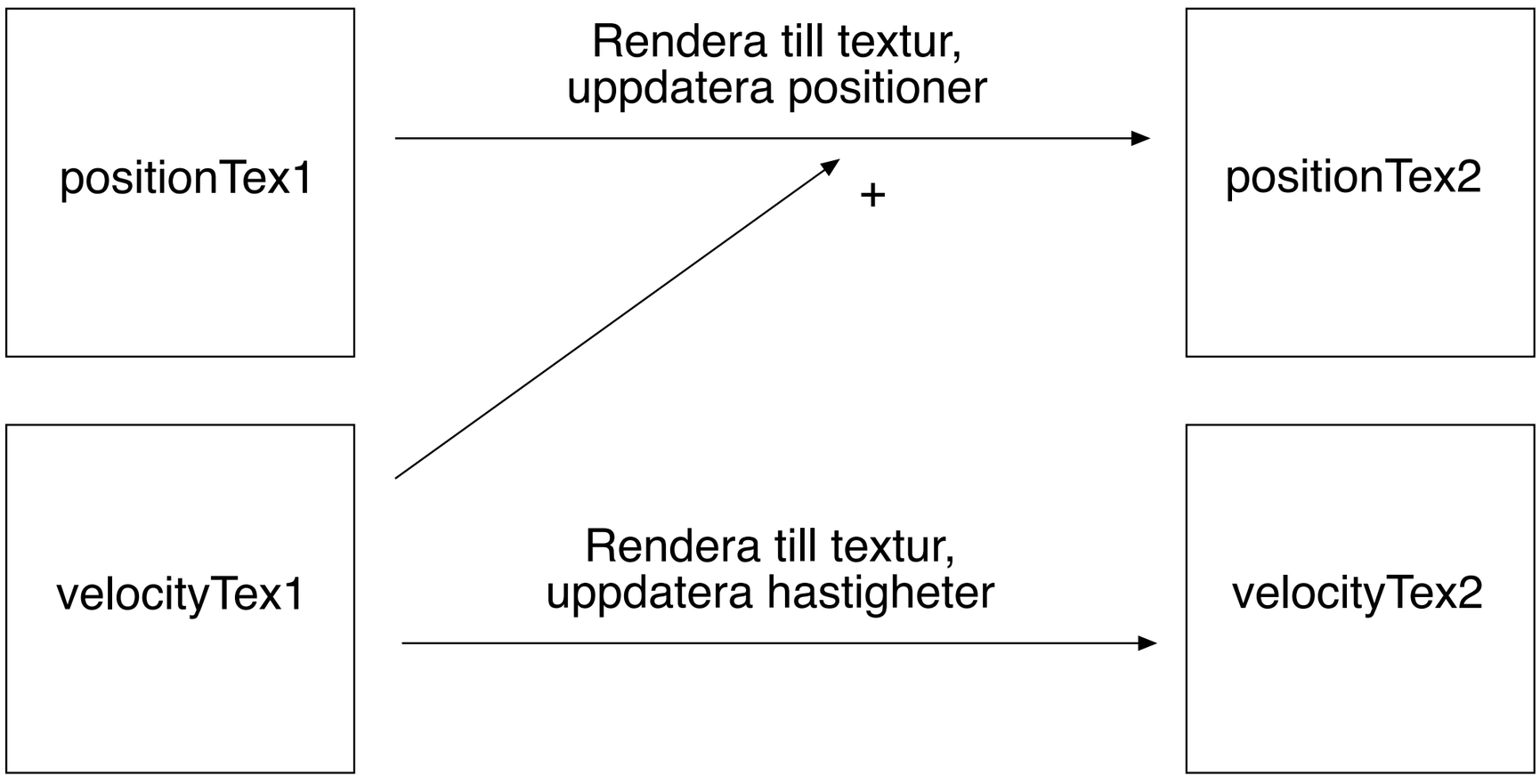
velocityTex



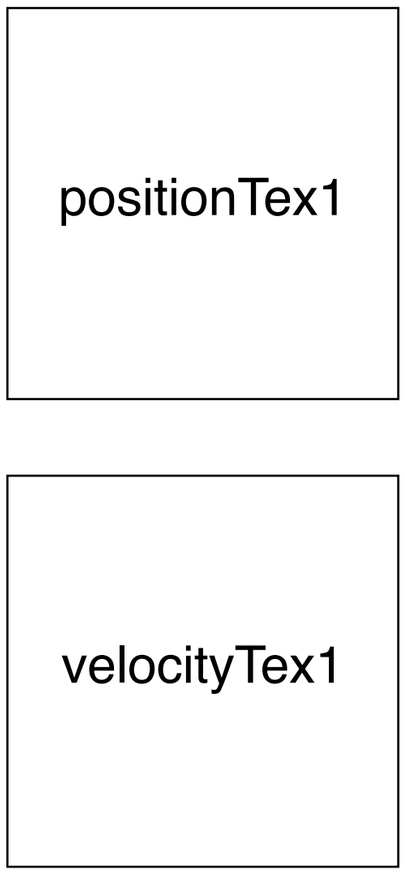
Dubbla texturer för ping-ponging



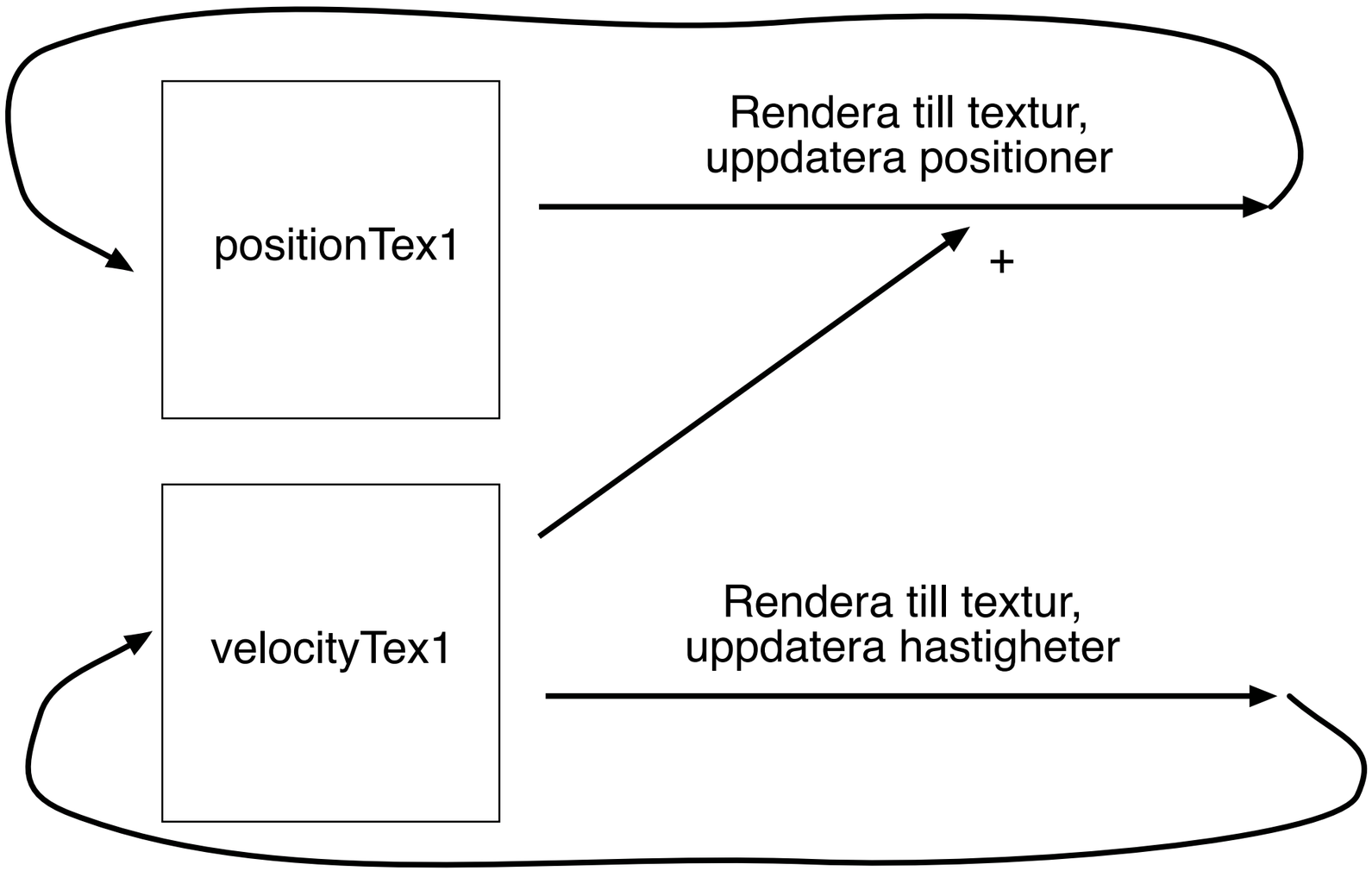
Rendera till FBO för att uppdatera



Kan vi jobba 1-1?



Uppdatera 1-1?



Borde inte fungera men beroende på dubbelbuffring kan det fungera

Rendering av partikelsystem

Görs typiskt med billboarding

Billboard = riktas alltid mot kameran

Texturerad quad med textur med transparens

Enklast: Tag bort rotation

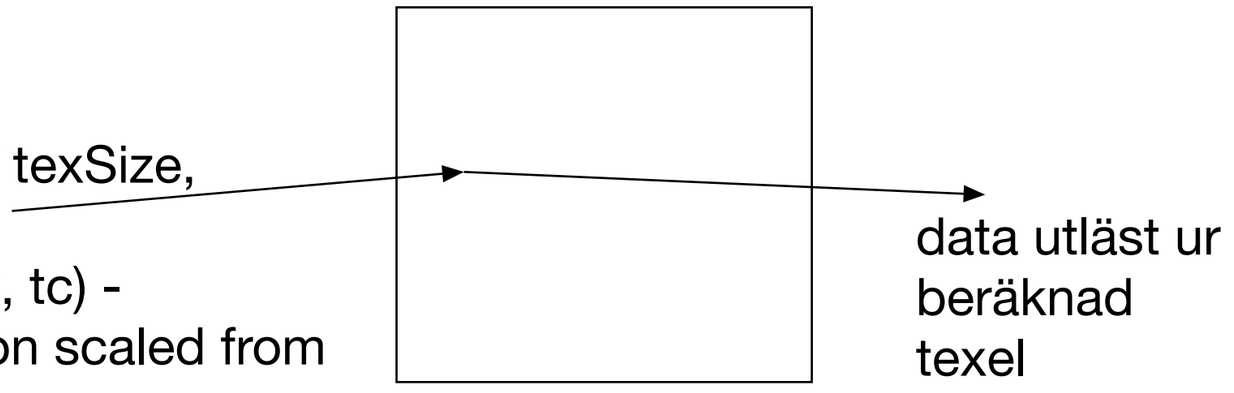
Rendering med instancing:

Intancing = Rendera många gånger med ett anrop.

Beräkna position i textur från instancing-index

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

```
int i = gl_InstanceID;
int x = i / texSize;
int z = i % texSize;
vec2 tc = vec2((float(x)) / texSize,
(float(z)) / texSize);
vec4 p = (texture(posTex, tc) -
vec4(0.5)) * 3.5; // Position scaled from
texture value
```

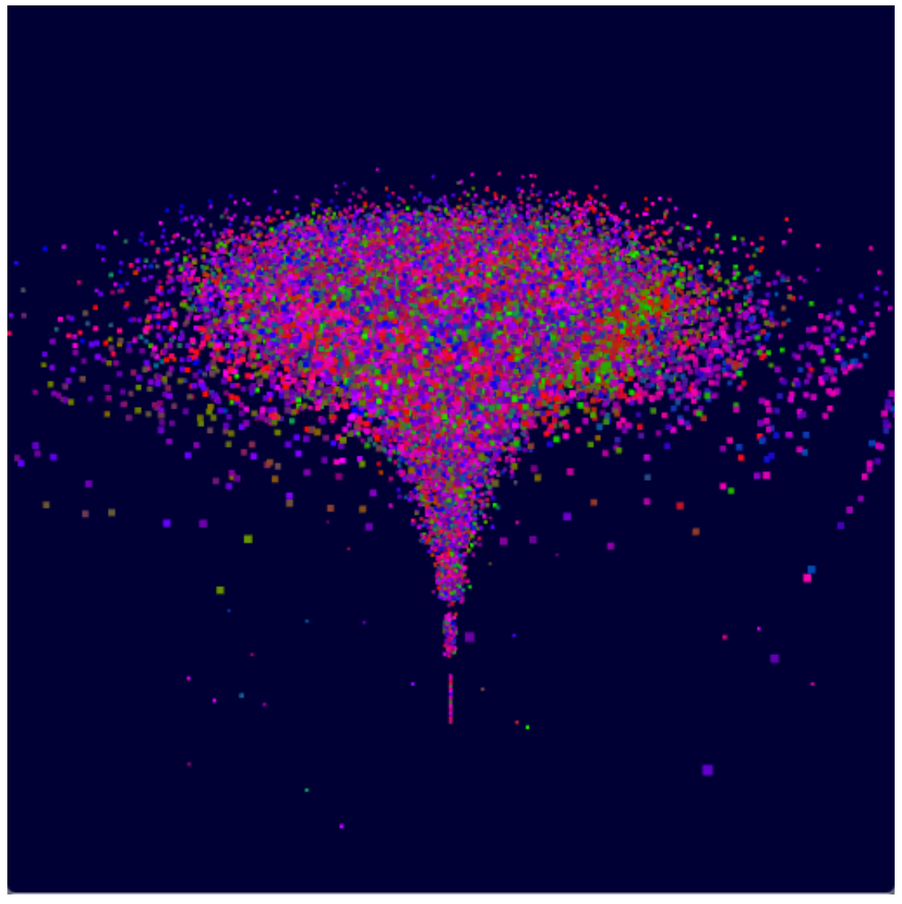


Exempel 1: 8-bitars partikelsystem

Alla texturer/FBOer är RGBA8888

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8,
width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
NULL);
```

Det ger en upplösning på 256 steg för både position och hastighet!



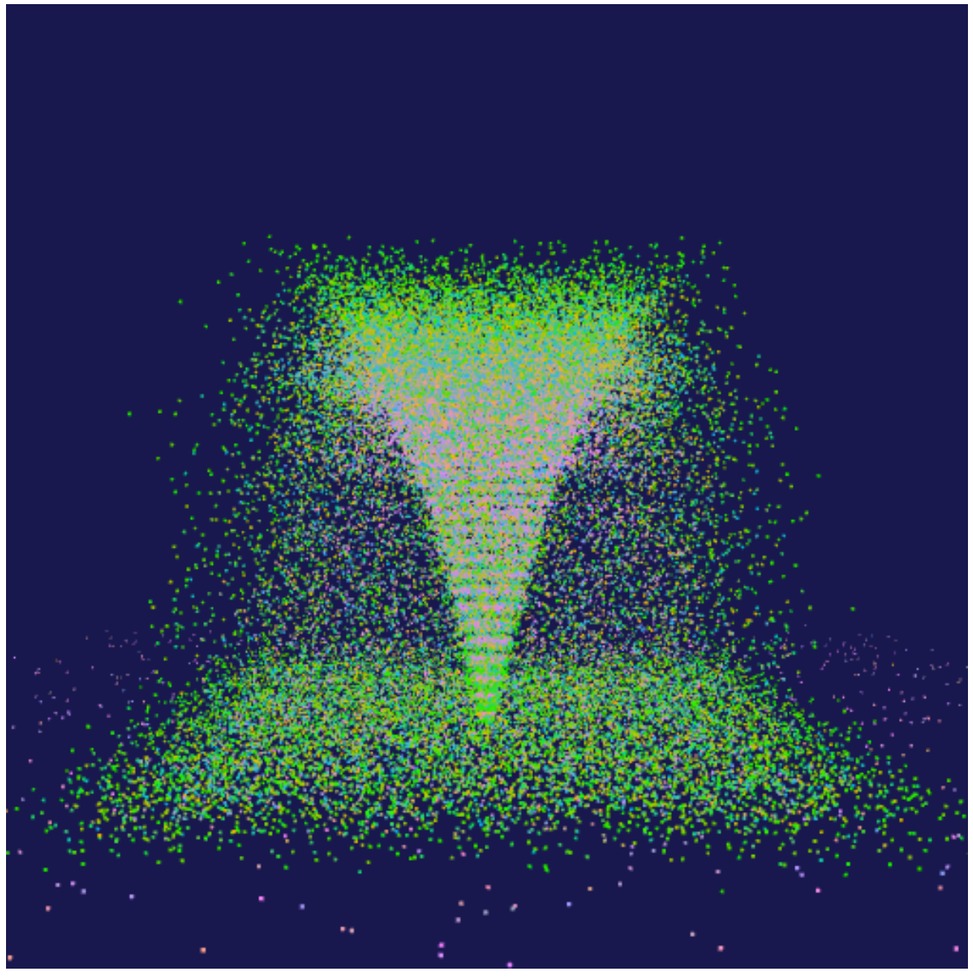
Inte kass. Lite begränsad i storlek och rörelsefrihet.

Bättre: Flyttalstexturer!

Ny tillämpning för flyttalstexturer!

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F,
width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE,
NULL);
```

8 bitar byts mot 32 (24 bitar förutom exponenten!)



Mycket bättre. Stor frihet på alla sätt.

Minimala exempel!

Finns på kurssidan.

Avsedda som bas att bygga på.

Klarar inte sortering eller beroenden av andra partiklar eller omgivningen (mer än plat mark)

Tre problem kvar:

Snyggare partiklar

Sortering

Beroenden av andra partiklar och omgivningen

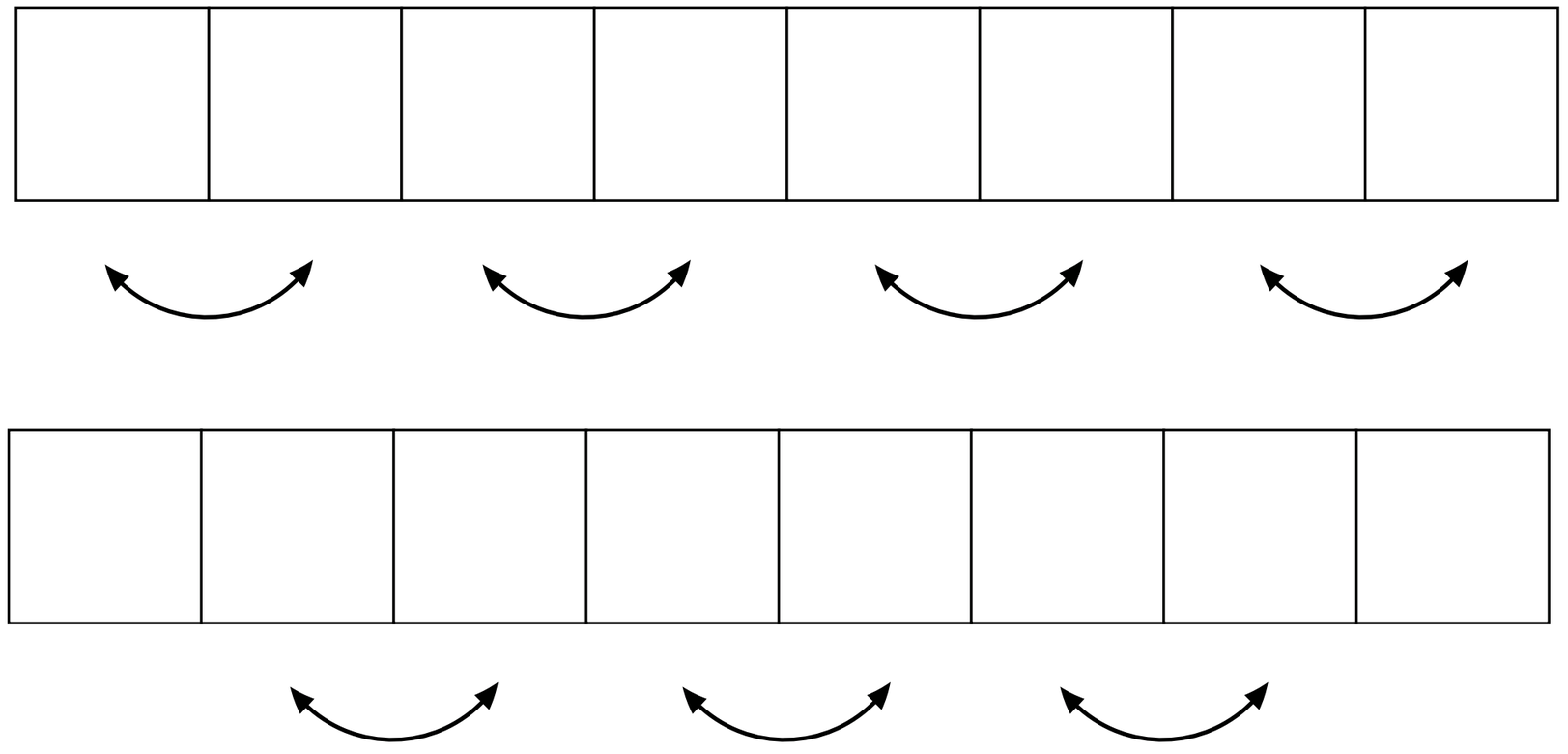
Sortering behövs i många fall!

Partiklar med semitransparens (nära kameran)!

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 minska problemen

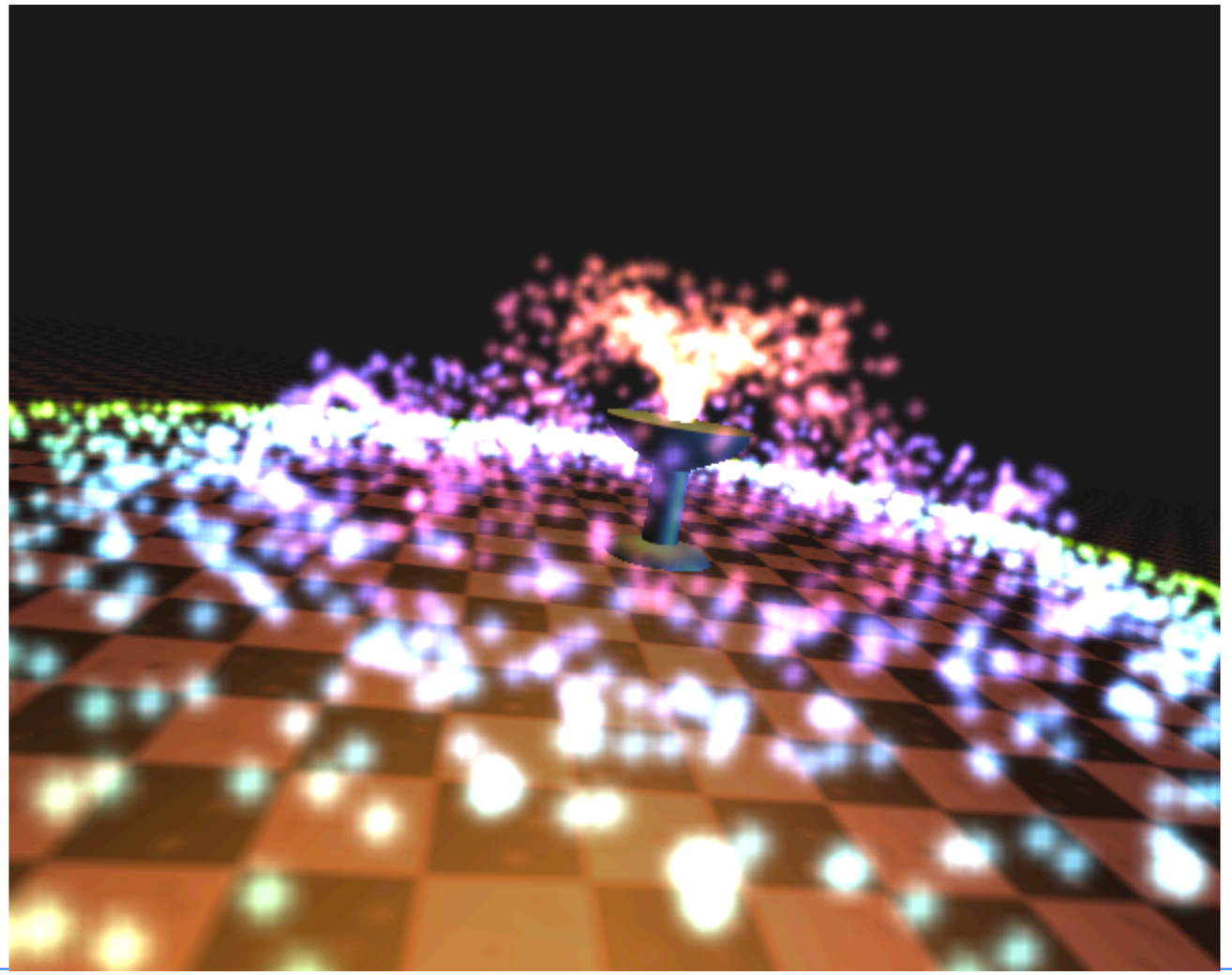
1) discard()

Bäst med skarpa kanter i textur

2) additiv blending

Fungerar för partiklar utan detalj

Gammalt exempel med additiv blendning

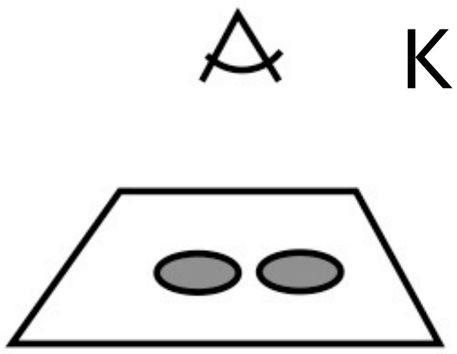


Kollisionsdetektering för partikelsystem

Inom systemet: Svårt problem - många till många

- Dela upp i celler! Octrees möjligt, uniform uppdelning ofta effektivare.
- Uppdatera cellerna när det behövs, när objekt rör sig utanför sin cell.
- Lagring knepigt, löses bäst med *spatial hashing*. för att lagra referens till alla partiklar i en stor array
- Test mot omgivning: Kan förenklas t.ex. med Z-buffer!

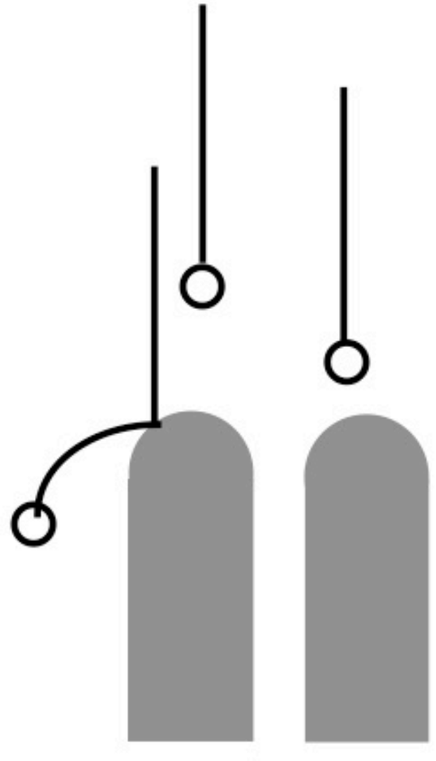
Exempel: Regn/snö mm



Kamera för att rendera Z-buffer uppifrån

Z-buffer

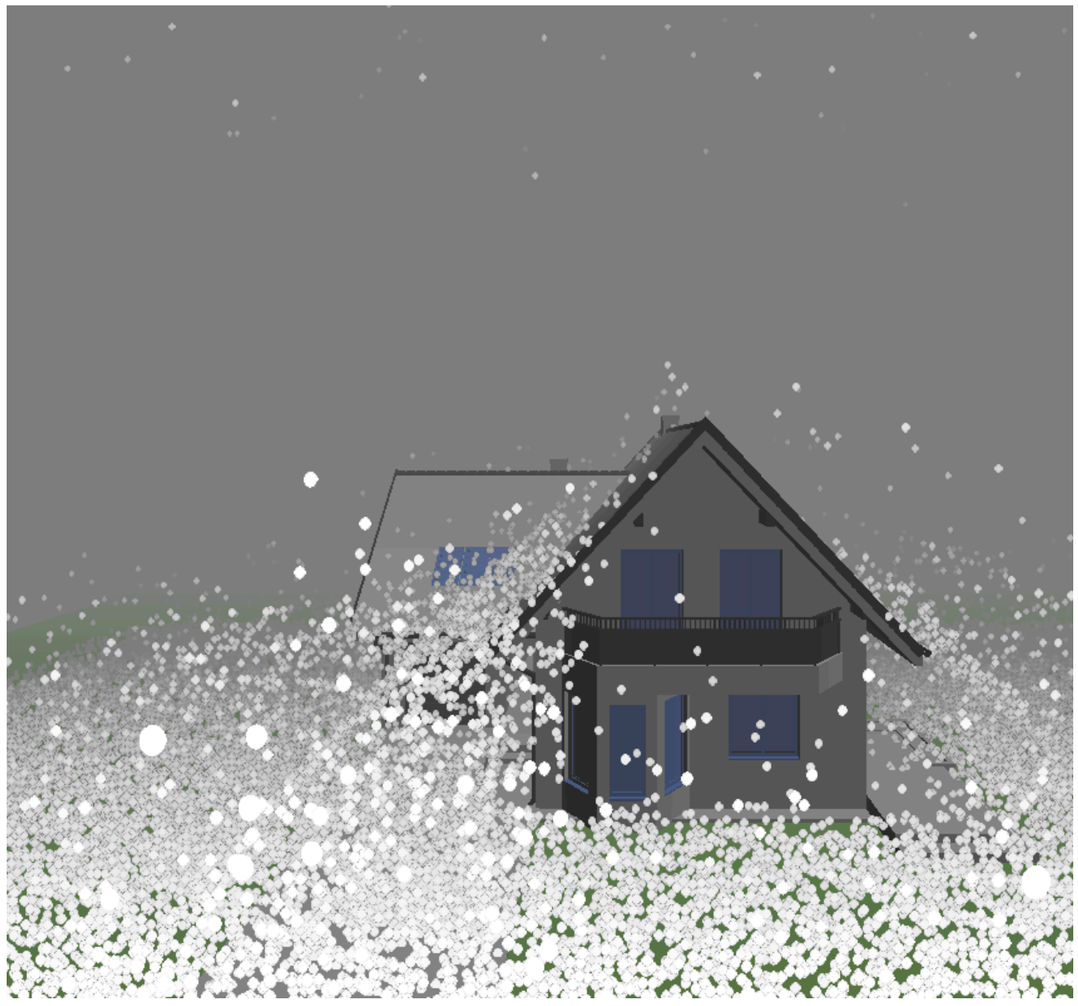
Scen



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