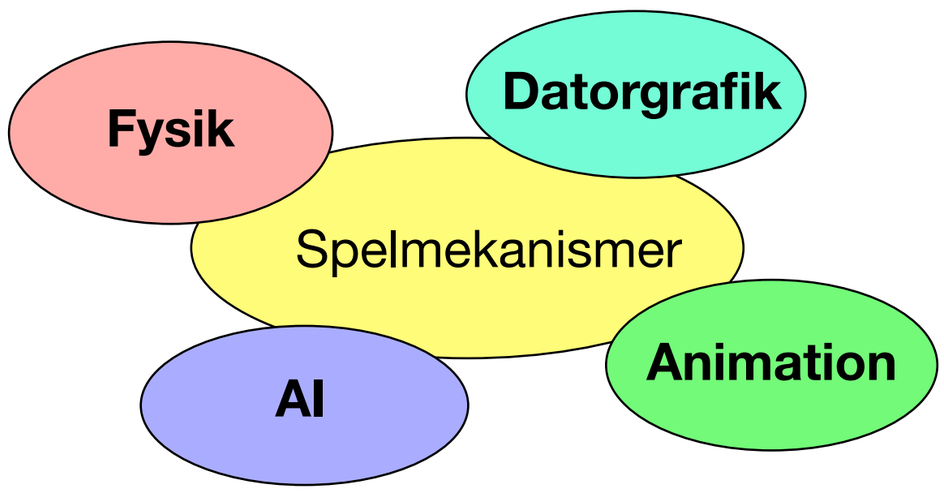


TSBK 03
Teknik för avancerade datorspel

Ingemar Ragnemalm, ISY



Föreläsning 4

Avancerade shaders

- Multipass-shaders
  - Filter, faltning
  - Flyttalsbuffrar
- High dynamic range
- Bump mapping med utvidgningar

Labbar och duggor

Labbar redovisas på plats

Svar på labbfrågor skrivs ner och besvaras under redovisningen

Duggor före varje labb, ca 15 minuter

En omdugga med alla på första reservlabben

Exempeldugga:

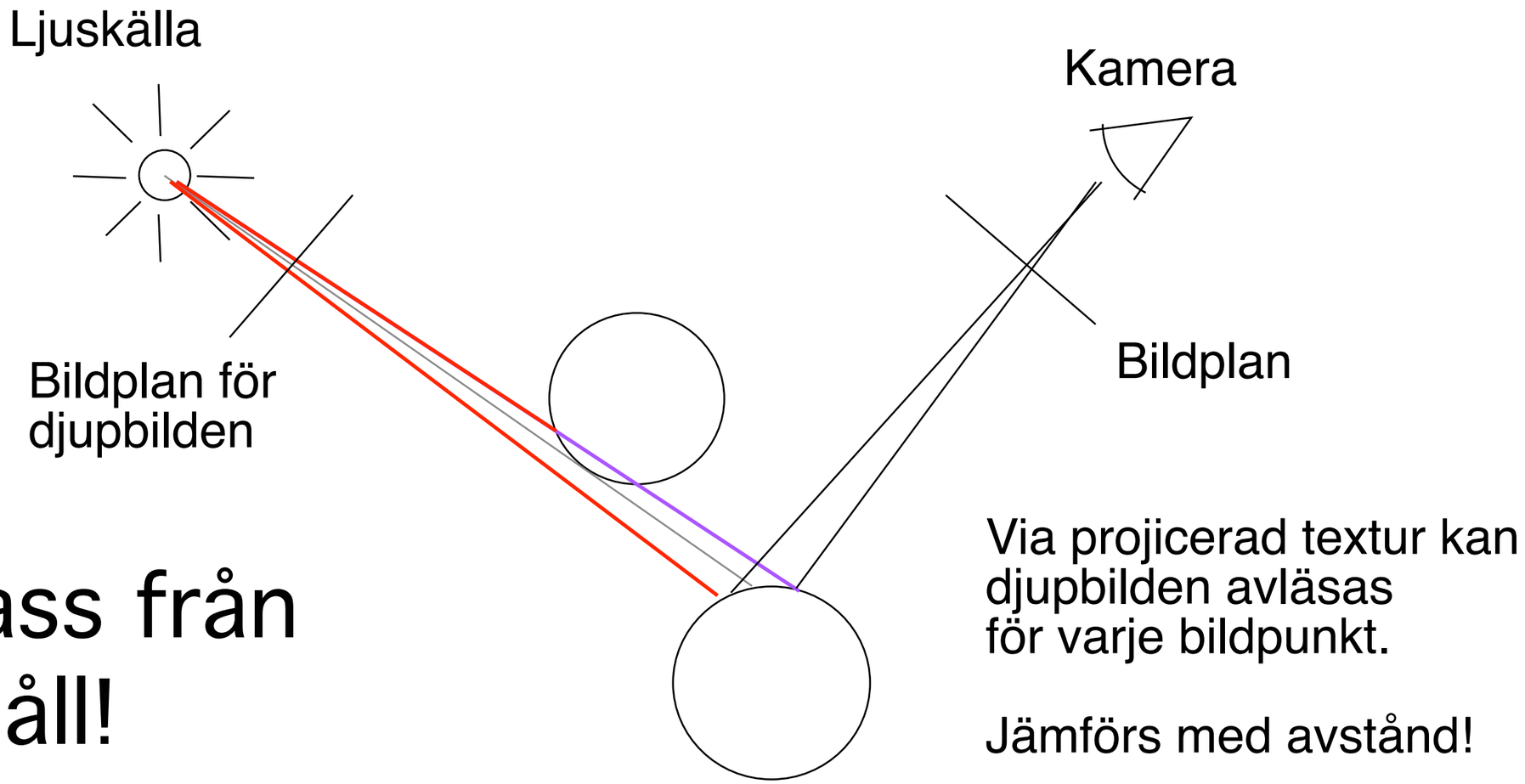
Multipass-shaders

Enkla shaders kan göras i ett pass.

De flesta avancerade behöver flera pass.

Rendera till textur, sedan från denna textur till ännu en...

Shadow maps - princip



Två pass från olika håll!

Via projicerad textur kan djupbilden avläsas för varje bildpunkt.

Jämförs med avstånd!

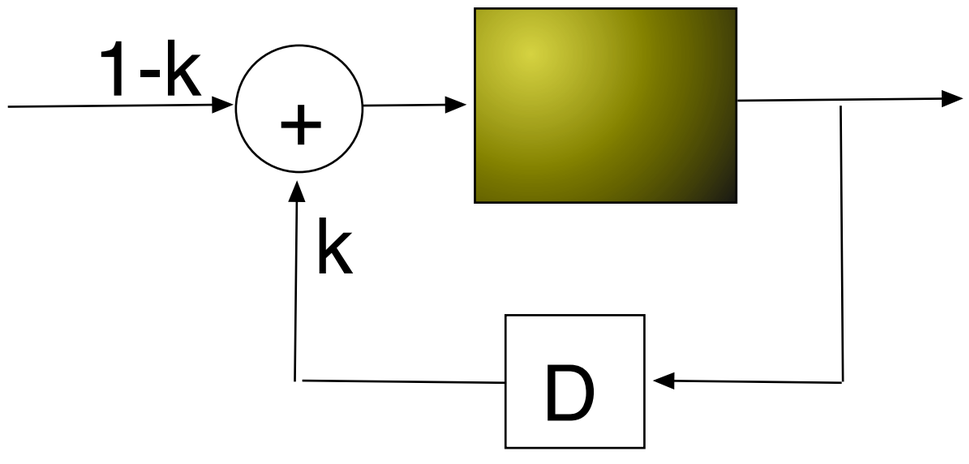
Rekursiv temporalfiltrering

Ger en ”eftersläpning”

Bilden sammanvägs med en tidigare

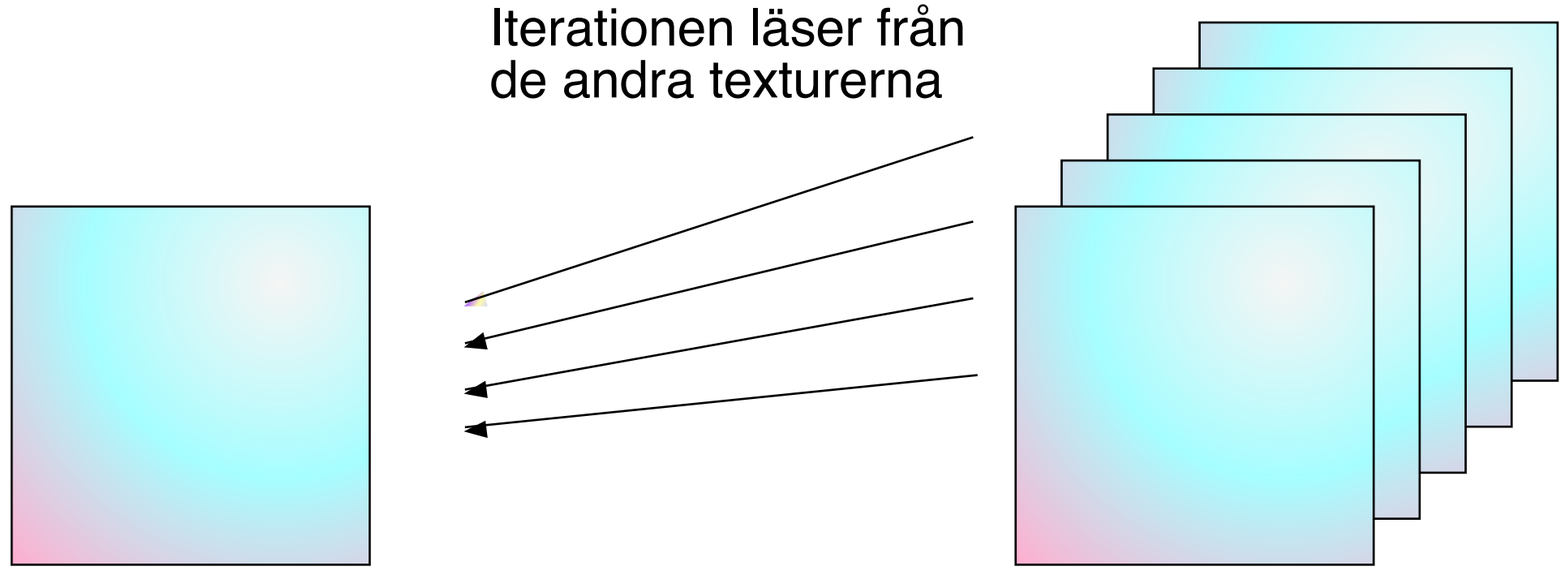
Kräver bara att en enda äldre bild sparas!

Obegränsat antal pass!



“Ping-pong”-ing

Iterationen läser från
de andra texturerna



Med framebuffer objects
kan utbilden vara en textur!

Indata: Ett antal texturer.
Begränsas av antalet texturenheter.

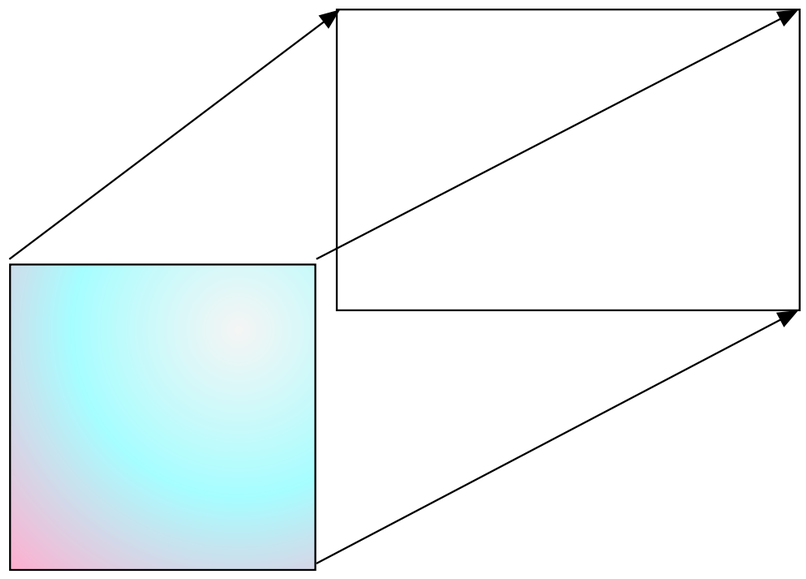
Återskrivning ersätts med att texturerna byter roll i processen

Geometri

Oftast bara en enkel rektangel tvärs över viewporten!

```
GLfloat quadVertices[] =
  {-1.0f, -1.0f, 0.0f,
  -1.0f, 1.0f, 0.0f,
  1.0f, 1.0f, 0.0f,
  1.0f, -1.0f, 0.0f};
GLfloat quadTex[] =
  {0.0f, 0.0f,
  0.0f, 1.0f,
  1.0f, 1.0f,
  1.0f, 0.0f};

GLuint quadIndices[] = {0, 1, 2, 0, 2, 3};
```



ping-pong med FBOs

Från den

Välj källa:

glBindTexture(GL_TEXTURE_2D, tx1);

Välj destination:

glBindFramebuffer(GL_FRAMEBUFFER, fb);
glFramebufferTexture2D(GL_FRAMEBUFFER,
GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, tx2, 0);

Till den

Filtrering, faltning

Vanligt problem: applicera filter i form av faltningskärnor

Alla typer av linjära filter:

- Medelvärdesbildning (smoothing)
- Gradient, embossing

Faltning (Convolution)

$$(f \otimes g)(t) = \sum f(\tau) \cdot g(t-\tau)$$

Observera att ena signalen “vänds”.
(Annars blir det korrelation, en annan operation.)

Detta kvittar för vanliga LP-filter.

Möjlig 5x5 faltningskärna för LP-filter

1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

Sobeloperator (gradient)

-1	0	1
-2	0	2
-1	0	1

Enklare gradientfilter

-1	1

-1
1

eller

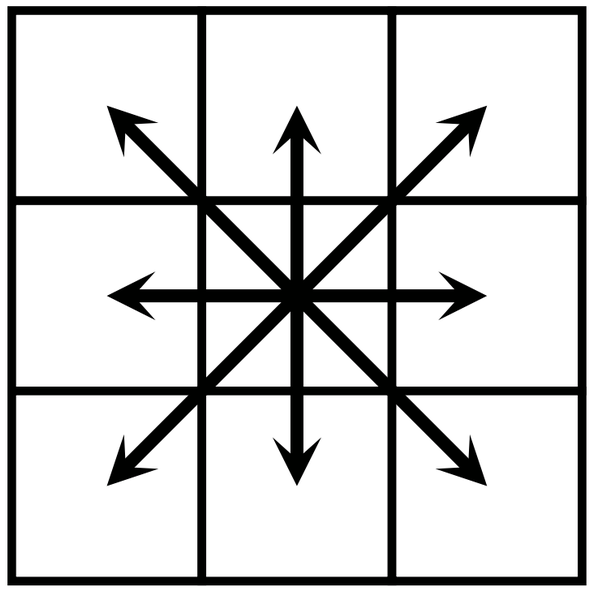
-1	0	1

-1
0
1

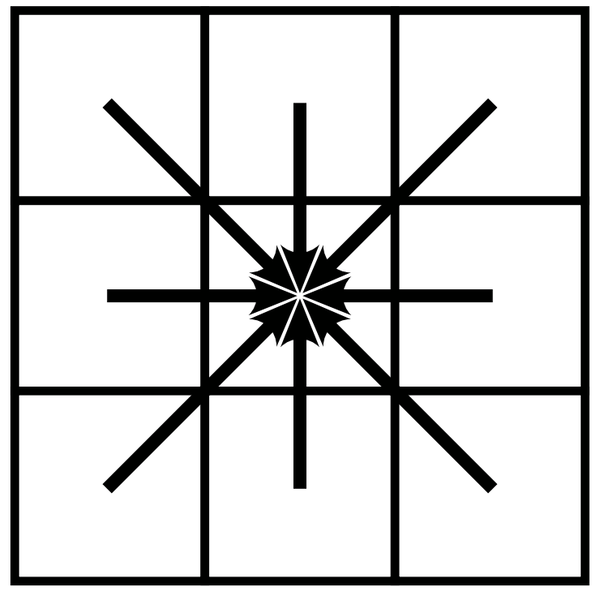
Laplace

0	-1	0
-1	4	-1
0	-1	0

Implementation av faltning



Scatter



Gather

Scatter eller gather?

Scatter: Dålig för GPUer! Kan inte alls utföras i shader. Kan utföras i t.ex. CUDA, men mindre effektivt än gather.

Gather: Enkel och effektiv på GPU! Stämmer bra med grundmodellen med fast utdata-fragment!

Separerbara filter

1	4	6	4	1
4	16	24	16	4
6	24	36	24	6
4	16	24	16	4
1	4	6	4	1

$=$

$$\begin{bmatrix}1 & 2 & 1\end{bmatrix} \otimes \begin{bmatrix}1 & 2 & 1\end{bmatrix} \otimes \begin{bmatrix}1\\2\\1\end{bmatrix} \otimes \begin{bmatrix}1\\2\\1\end{bmatrix}$$

Kan ge betydande acceleration för stora filter, om det kan göras.

Separerbara filter

Vad är flaskhalsen?

Beräkningar?
25 MUL blev 12

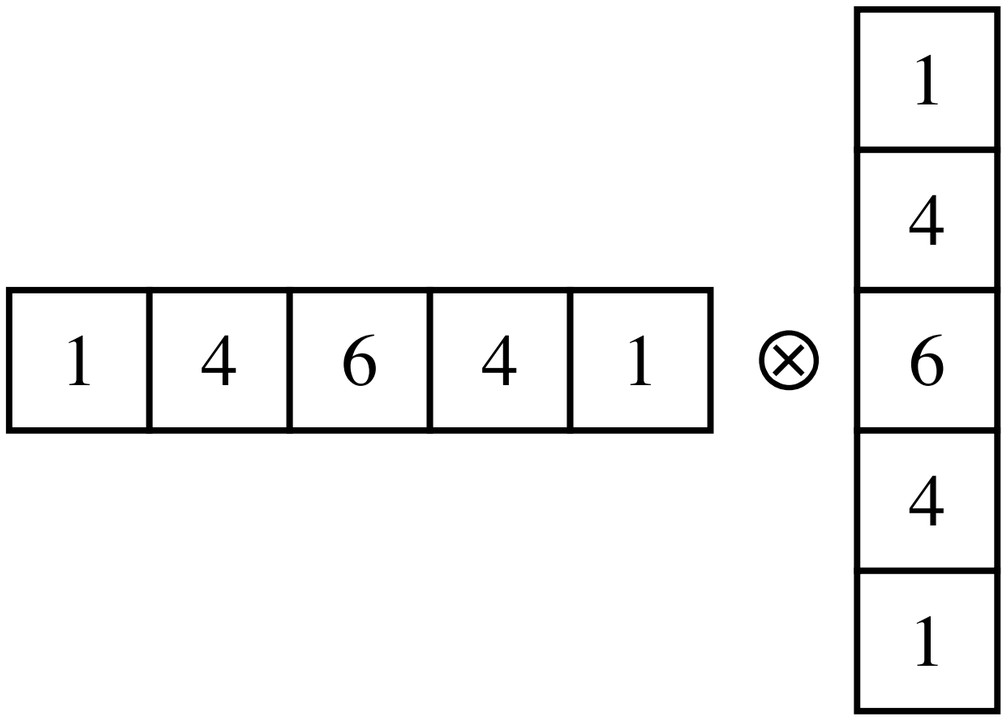
Texturaccesser? 25 blev 12.

Bildgenomgångar? 1 blev 4.

Troligast texturaccesser! Minnesaccesser är dyra, beräkningar billiga!

Testa!

Separerbara filter



Två stora 1D-filter

Beräkningar?
25 MUL blev 10

Texturaccesser? 25 blev 10.

Bildgenomgångar? 1 blev 2.

Ser bättre ut!

Alternativ metod för LP-filter:

Nedsampling!

Sampla ner med 2x2 per steg med linjär filtrering.

Snabbt - utnyttjar GPUns inbyggda filter!

Mindre exakt, mindre frihet än faltning.

Flyttalsbuffrar

Framebuffer normalt RGBA 8888, 32 bitar.

“Varför skulle man någonsin behöva mer?”

Men shaderprogrammen jobbar uttryckligen med flyttal.

En textur/FBO kan allokeras med flyttal!

Flyttalsbuffrar

Flera olika precisioner:

FP64: s53.11
FP32: s23.8
FP24: s16.7
FP16: s10.5

Lägre precision ger ofta bättre prestanda.

Allokering av flyttalsbuffer

glTexImage2D(GL_TEXTURE_2D,0,
GL_RGBA32F,texSize,texSize,
0,GL_RGBA, GL_FLOAT, data);

data = NULL skapar en tom textur, eller med odefinerade data

Dynamic Range i flyttalsbuffer

Flyttalsbuffrar har enormt mycket större dynamik än 8 bitars heltal!

Detta ger möjlighet att rendera med högre dynamik. Dock kan man inte visa det på skärmen...

High Dynamic Range

Postprocessing i shaders används för att ta tillvara den rikare informationen.
Typiska HDR-effekter:

- Blooming (glow)
- Tone mapping (jfr histogram-utjämning mfl histogramoperationer)

Blooming

I vanlig grafik blir områden ljusare än 1 “clampade” till 1.0, “urfrätta”.

Vi kan använda den “urfrätta” delen på bättre sätt än att kasta bort den!

Vi vill sprida ljuset till omgivningen för att ge ett “sken”. Lågpassfilter!

Algoritm

- Rendera scenen normalt, till flyttalstextur.
- Trunkera denna (till ny textur) så vi enbart behåller värden över 1.0.
- Lågpassfiltrera detta.
- Lägg ihop resultatet med originalscenen (med lämplig viktning).

Shaders för blooming

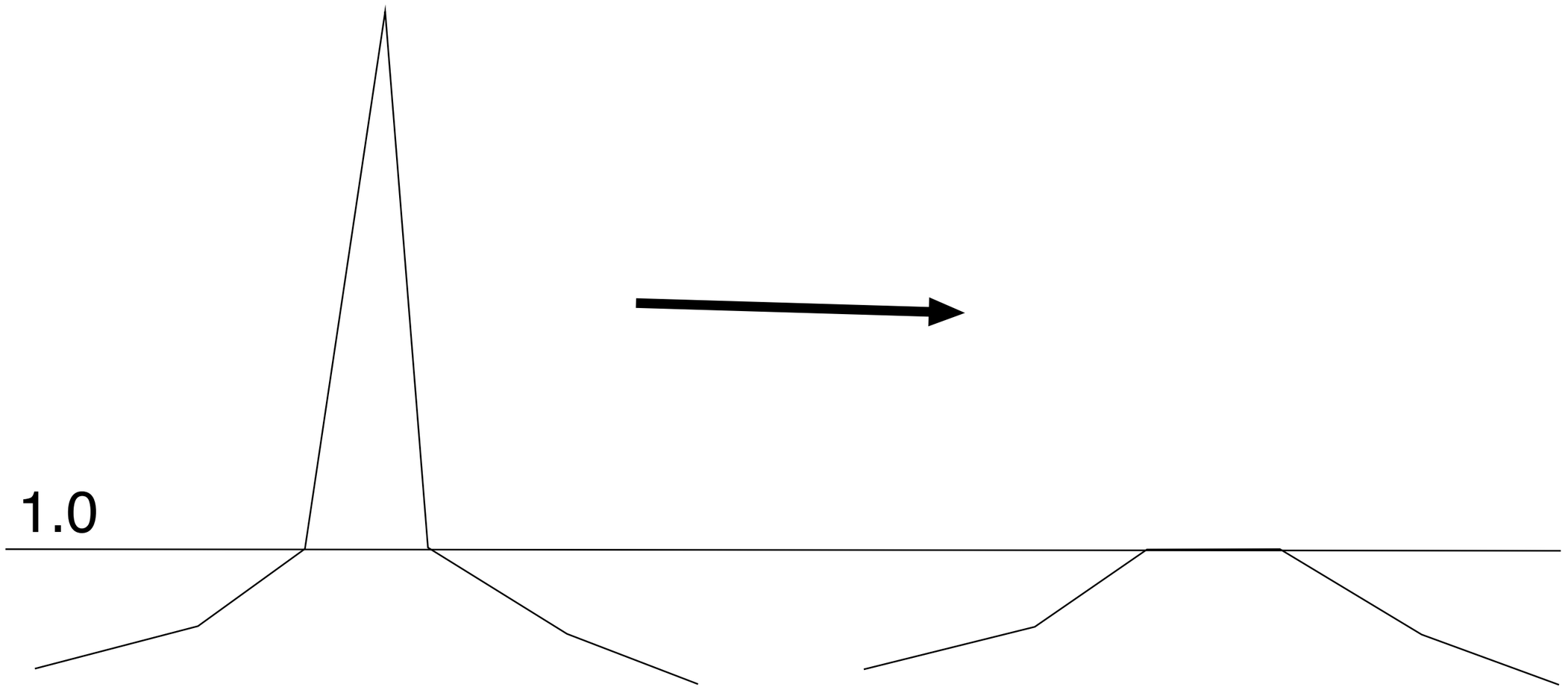
Normal rendering: Görs hur man önskar.

Trunkering kan göras med enkel shader.

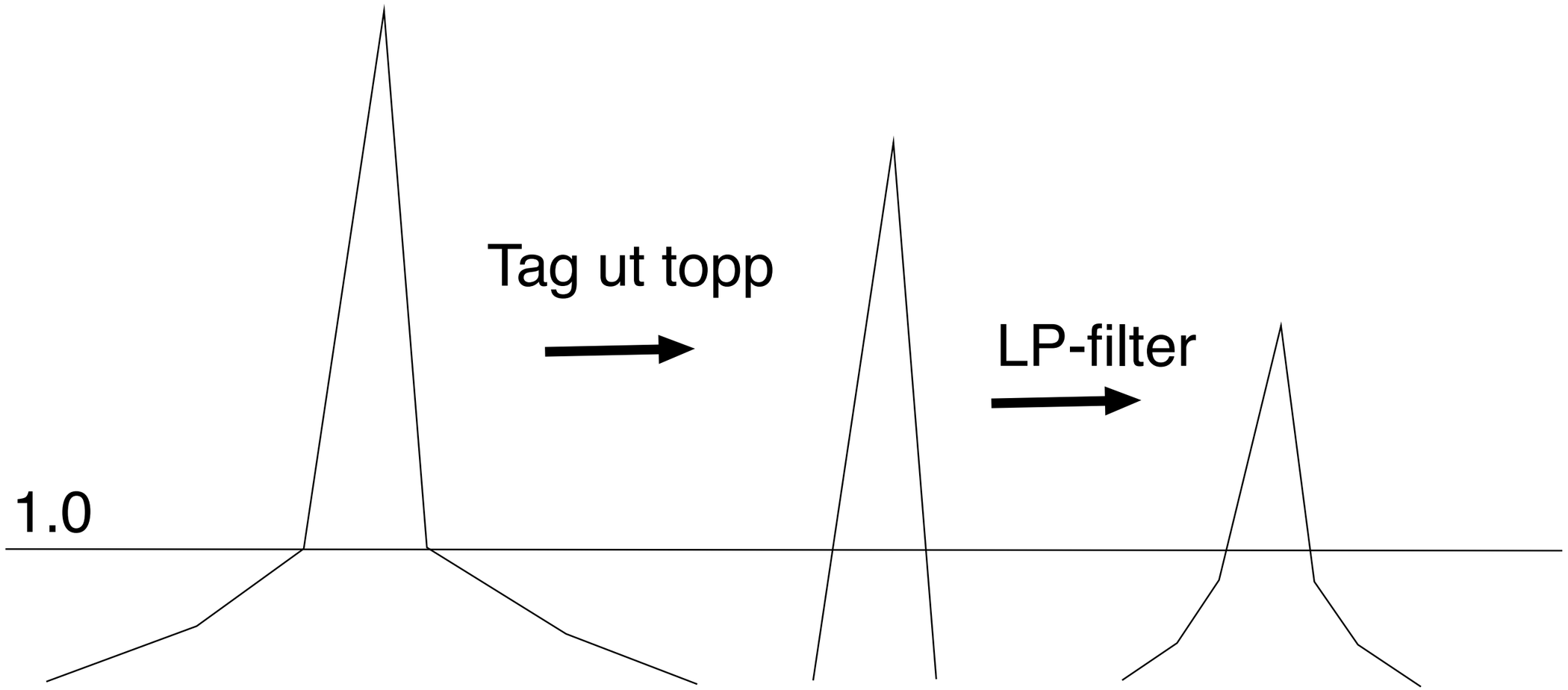
Lågpassfilter görs med flera pass LP-filter i shader.

Avslutande sammanviktning görs med en enkel shader.

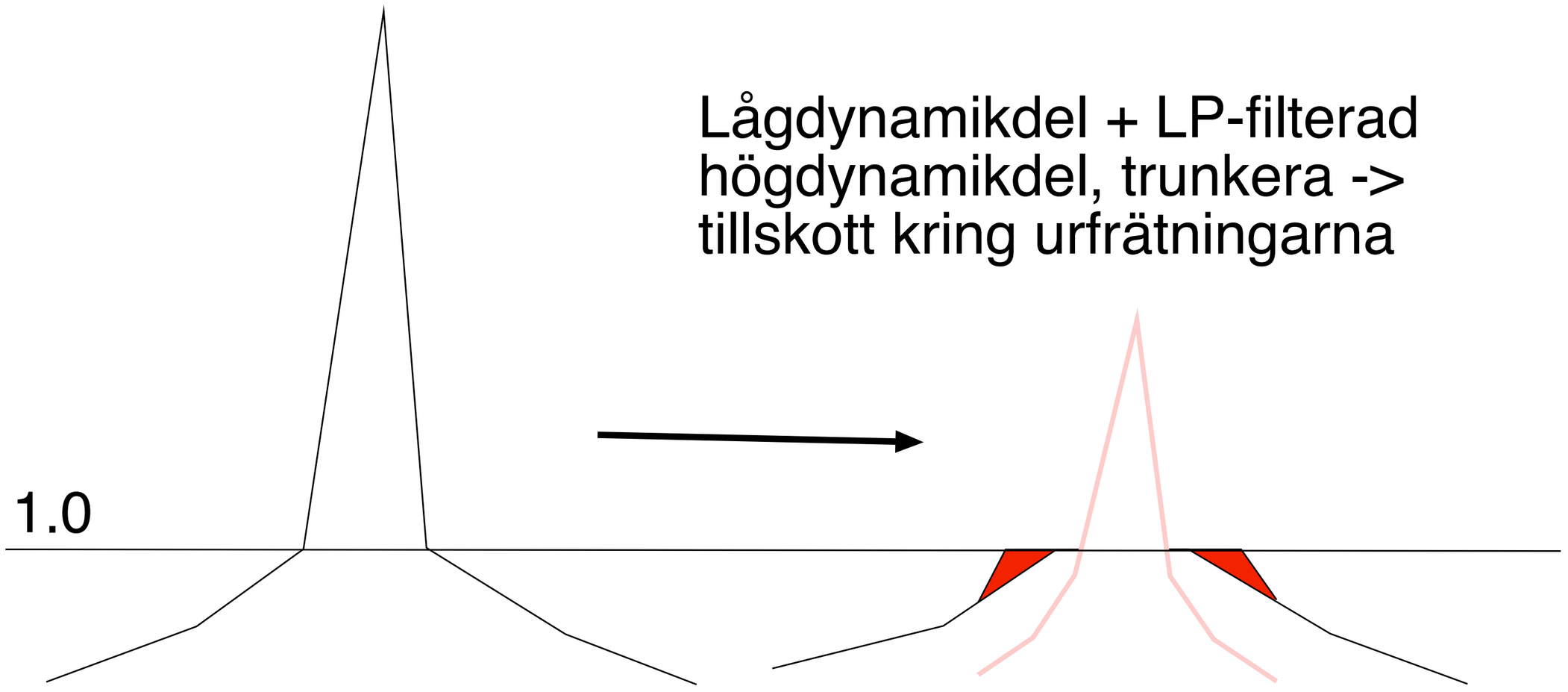
Vanlig ”urfrätning” av ljusa områden



Filtera överskott



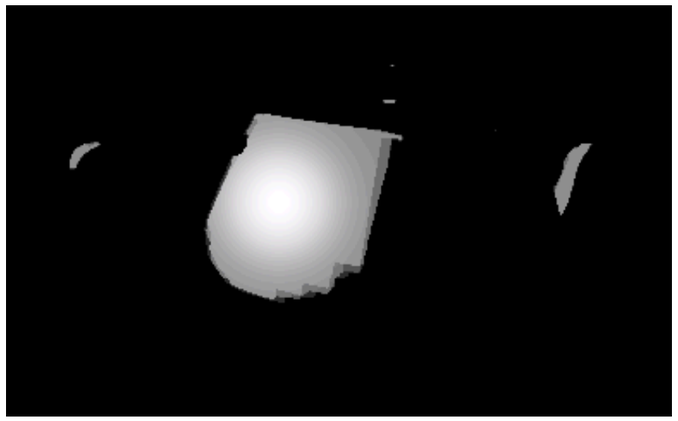
Lägg till filtrerat



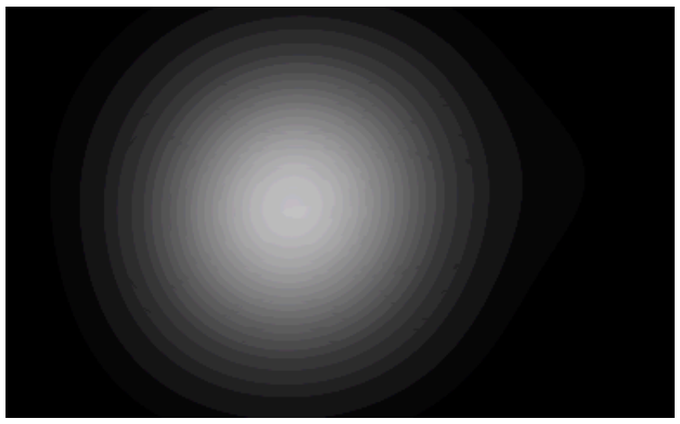
Exemplet från boken



Normal rendering



Overflow



Overflow filterat (hårt!)



Resultat

Observera i liveexemplet

Texturen på tekannan blir inte suddig!
Bara ljusa områden suddas ut!

Exempel med kod

Kod för
- initiering
- shaders
- ping-pong

Initiering

Flera flyttalstexturer

```
// initialize texture 1
glGenTextures(1, &tex1); // texture id
glBindTexture(GL_TEXTURE_2D, tex1);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, width, height, 0, GL_RGBA,
                        GL_FLOAT, 0L); // NULL = Empty texture
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Initiering

...som får en matchande FBO

```
void initFBO(GLuint *fb0, GLuint *rb0, GLuint tex0)
{
// create objects
    glGenFramebuffers(1, fb0); // frame buffer id
    glGenRenderbuffers(1, rb0); // render buffer id
    glBindFramebuffer(GL_FRAMEBUFFER, *fb0);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                           GL_TEXTURE_2D, tex0, 0);

// Renderbuffer (KAN SKIPPAS I MÅNGA FALL)
    glBindRenderbuffer(GL_RENDERBUFFER, *rb0);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, width, height);
// attach renderbuffer to framebuffer depth buffer
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                           GL_RENDERBUFFER, *rb0);
    printf("Framebuffer %d\n", *fb0);
}
```

En del kod bör paketeras:

Shaderladdning packar vi alltid ner! Alltid detsamma:

- Ladda filer från disk.
- Kompilera varje delshader.
- Länka ihop.
- Rapportera InfoLog.

GL_utilities innehåller en hyfsad laddare.

Även FBO-laddning är (mestadels) återanvändbar.
Har man en som funkar så kan man oftast utgå från den.

Shaders:

- Enkel vertexshader
- Trunkera
- Filtrera
- Sammanslagning

Enkel vertexshader

indata är en rektangel över hela bilden!

```
in vec3 in_Position;

out vec3 norm;
out vec2 texCoord;

void main(void)
{
   texCoord = in_Position / 2.0 + vec3(0.5);
   gl_Position = vec4(in_Position, 1.0);
};
```

Eller motsv texturkoordinater

Trunkering i shader

```
uniform sampler2D texUnit;

in vec2 texCoord;
out vec4 fragColor;

void main(void)
{
  vec4 col = texture(texUnit, texCoord);
  fragColor.r = max(col.r - 1.0, 0.0);
  fragColor.g = max(col.g - 1.0, 0.0);
  fragColor.b = max(col.b - 1.0, 0.0);
}
```

Filter

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

```
void main(void)
{
  float offset = 1.0 / texSize;
  vec4 c  = texture(texUnit, texCoord);
  vec4 l  = texture(texUnit, texCoord +
                    vec2(offset, 0.0));
  vec4 r  = texture(texUnit, texCoord +
                    vec2(-offset, 0.0));
   fragColor = (c + c + l + r) * 0.25;
}
```

1	2	1

Sammanslagning

```
uniform sampler2D texUnit;
uniform sampler2D texUnit2;
in vec2 texCoord;
out vec4 fragColor;

void main(void)
{
   vec4 a  = texture(texUnit, texCoord);
   vec4 b  = texture(texUnit2, texCoord);
   fragColor = (a*0.3 + b*1.0);
}
```

Körning av shader

Speciell funktion i GL_utilties, UseFBO(), gör det lätt att köra många. Tar shader, texturer och ut-FBO som parametrar.

- Ställ in texturer och FBO
- Vertex shader utan projektion eller modelview
- Rita rektangel
- Ställ tillbaka

Blooming med HDR, slutsatser

- Flera enkla shaders, appliceras i flera pass
- Filtrera som en galning (och optimera sedan)
- Flyttalsbuffrar för bästa resultat (halvdant möjligt utan, men med fula trick)

Bonus liveexempel med kod

Baserade på labbmaterialet utan att förstöra labben!

Rendera live och gör sedan något mer:

- Postprocessing från FBO
- Kombination av två FBOer