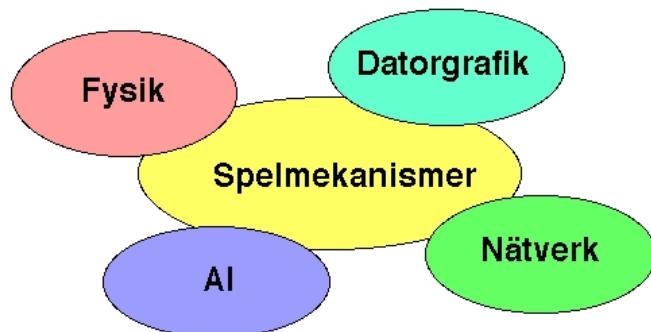


# **TSBK 03**

## **Teknik för avancerade datorspel**

**Ingemar Ragnemalm, ISY**



Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

## **Föreläsning 5**

- **Intro till GPGPU**
- **Kollisionsdetektering**
- **Text**
- **Orthographic mode**
- **Anti-aliasing med FBO**

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# **GPGPU**

## **General-purpose**

**Intressant trend: Försök använda GPU'ns beräkningskraft för andra problem än grafik!**

**gpgpu.org**

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

**Detta argument har dock kommit lite på skam de senaste åren.**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# **GPGPU**

**Exempel på tillämpningar:**

- **Bildbehandling**
- **Bildanalys**
- **Ekvationssystem**
- **Wavelettransform**
- **Fouriertransform**
- **Cosinustransform**
- **Level sets**
- **Videokodning**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# GPGPU

## Problem:

- Algoritmer måste vara av parallel natur. Man kan inte utnyttja resultat från grannpixlar som genereras i samma iteration.
- Databussen är (var) en flaskhals! Processning i GPU'n kan ta mindre tid än det tar att läsa ut resultatet.

Lönar det sig att processa på GPU'n?

Motargument: Multicore-CPU'er håller som bäst på att bli vanliga. Kan GPUn sätta något mot det?

Ingemar  
Ragnemalm  
ingis@isy.liu.se

## Typisk GPGPU: se filtreringsdelen

### Samma grundkoncept:

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

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# GPGPU-konceptet

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

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

## En GPU är en SIMD-processor!

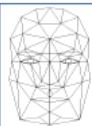
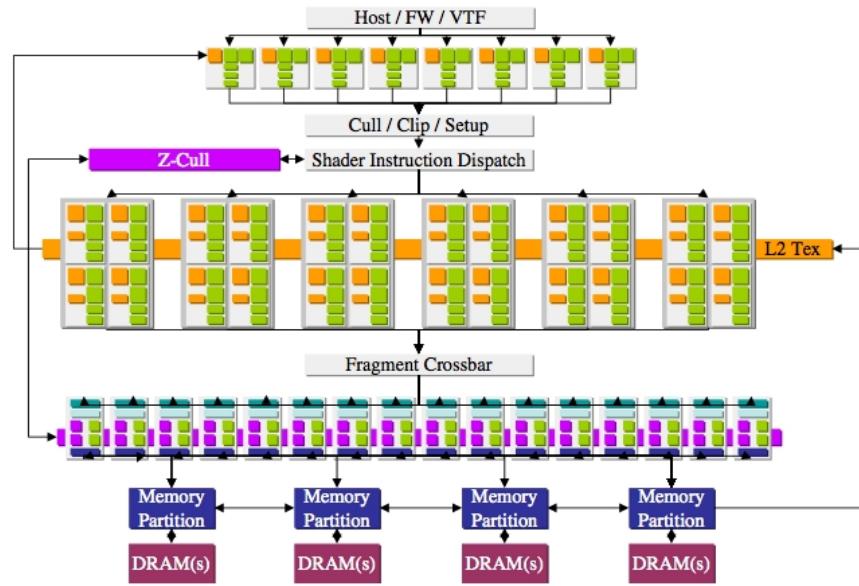
Kopplingen till fixa pipelinen blir allt svagare.  
Allt mindre av hårdvaran är hårdkodad för en  
enda specifik uppgift!

Generalisering som också givit  
prestandavinster! (Varför?)

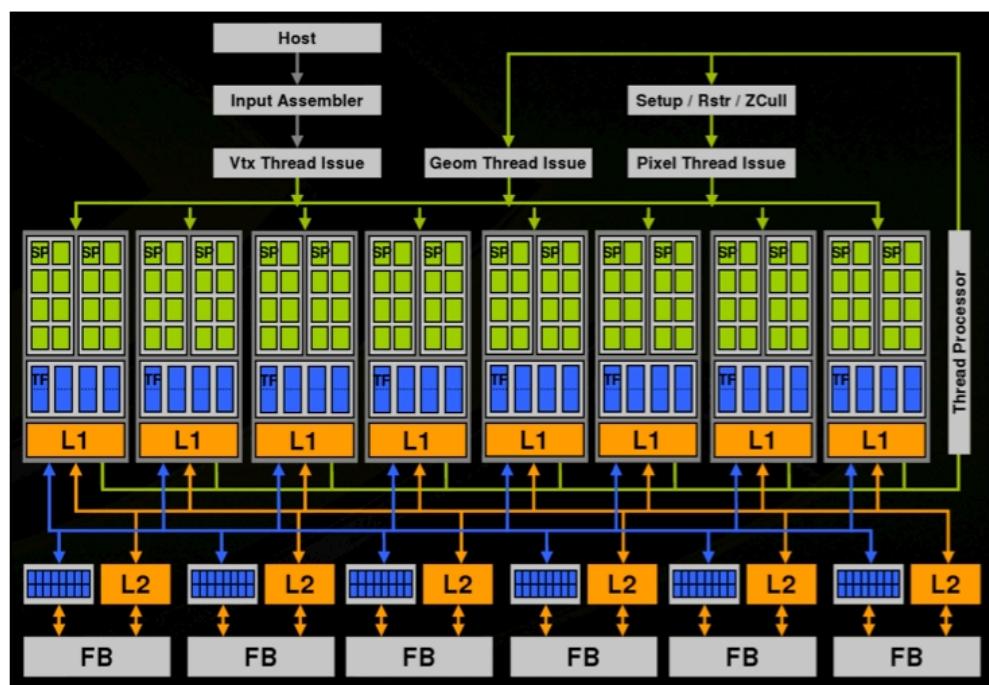
Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)



## 7800 - older GPU



**G80  
(8x00)**



# Enkelt exempel: process-array

Array av data läggs i textur

```
uniform sampler2D texUnit;  
void main(void)  
{  
    vec4 texVal = texture2D(texUnit, gl_TexCoord[0].xy);  
    gl_FragColor = sqrt(texVal);  
}
```

Hämtas sedan tillbaka till GPU igen.

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

## Specifika problem och lösningar

- Interpolation
- Sortering
- Reduktion
- Villkorssatser

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# Interpolation

Ett beräkningstrick för optimering

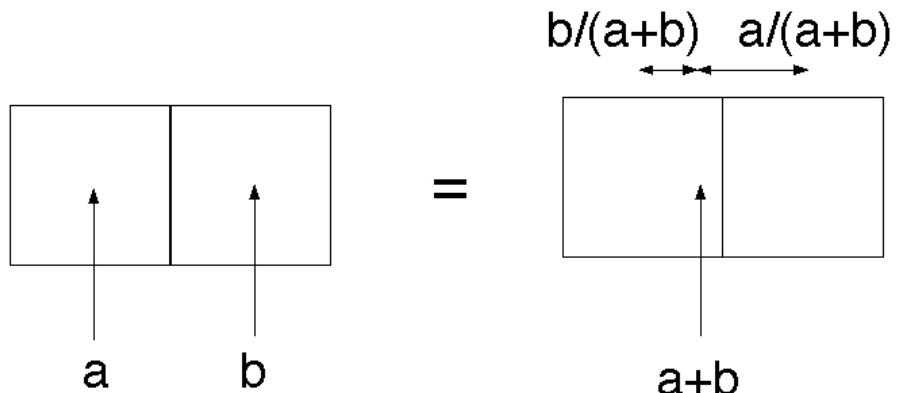
Texturrendering har linjär interpolation i hårdvara!

Slå upp textur på icke-heltals-koordinat och du får automatiskt linjärinterpolation mellan grannarna!

Kan utnyttjas för motsvarande beräkningar!

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# Interpolation



Ger färre texturaccesser vid filtrering!

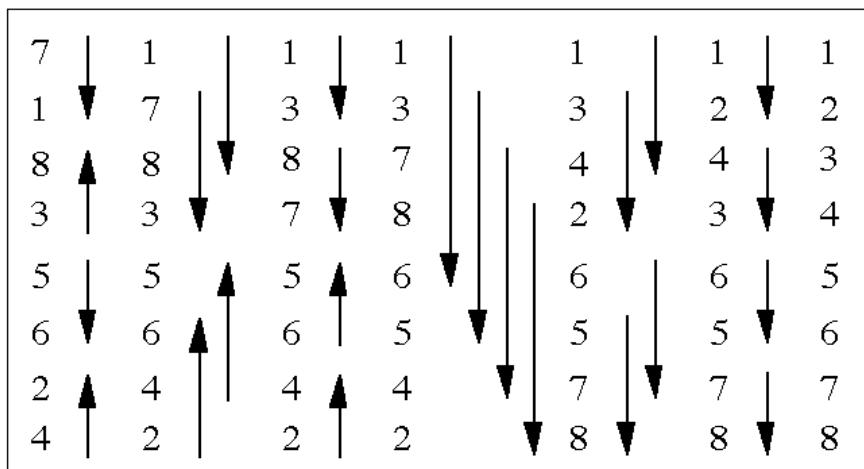
Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# Sortering

Parallel exekvering -> QuickSort fungerar inte!

Måste använda parallel sorteringsalgoritm.

Bitonic Merge Sort är lämplig



Ingemar  
Ragnemalm  
ingis@isy.liu.se

# GPGPU i framtiden?

Spelfysik? Har slagit ut fysikkorten!

Tar multicore-CPU över i längden?

Hybrider, vanlig CPU plus SIMD?

GPGPU ser ut att spela en viss roll åtminstone  
som coprocessor.

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# **Mer GPGPU i kursen:**

**Introduktion till CUDA (Johan Hedborg)**  
**Introduktion till OpenCL (Jens Ogniewic)**

**(Korta delar av föreläsningar.)**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

## **GPU'ernas historia**

**“...med vilken jag dagligen måste föra  
en massa tråkiga samtal om hur  
vädret var förr och sånt där...”**

**(H. Alfredsson, “Gamle man”)**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Stenåldern

**70-tal: Forskning på specialiserad grafikhårdvara**

**80-tal: En del 3D-spel, linjegrafik eller extremt förenklat. "Snabba grafikkort" betydde att de inte hade för mycket wait states. En del hårdvarustöd för 2D-sprites.**

**90-tal: Mjukvarurendering superhett fram till 1995.  
Wolfenstein3D, Doom i 320x240 pixlar. 3D-motorerna hade svåra begränsningar.**

**1991: 2D-acceleratorer för grafikproffs.**

**3D-acceleratorer för proffs under början av 90-talet.**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# Shaderprogram

**Renderman, Pixar, 1988**

**Målet var offlinerendering (Star Wars 1, Sagan om ringen, Toy Story...)**

**Renderman shading language, C-liknande språk**

**6 (!) olika typer av shaders:**

- Light source shader
- Volume shader
- Transformation shader
- Surface shader
- Displacement shader
- Imager shader

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# **SGI - proffssystemen**

**Helt dominerande under 90-talet,  
multiprocessordatorer. 64-bitars CPU (MIPS) redan  
1992!**

**Skapade OpenGL ur det interna IrisGL.**

**Har ägt Alias|Wavefront och Cray.**

**Allt bättre GPUer i vanliga PCI/AGP-kort åt upp  
SGIs marknad. Mycket Linux - som SGI själva  
stöttat!**

**2006: SGI i kris.  
MIPS/IRIX nedlagt.**



Ingemar  
Ragnemalm  
ingis@isy.liu.se

## **Revolutionen!**

**1996 kom 3dfx Voodoo 1, 3D-acceleration för  
konsumenter!**

- 50 MHz**
- 4-6 Mb VRAM**
- ca 3000:-**
- Saknar VGA-controller och 2D-grafik! Kopplades  
in mellan skärm och vanligt grafikkort!**
- En texturenhet, ingen multitexturering!**

**Historiens mest betydelsefulla grafikkort!**

**Första spelet med stöd för 3dfx: Tomb Raider.  
Senare även bl.a. Quake 1.**

Ingemar  
Ragnemalm  
ingis@isy.liu.se

# **1996-2000**

**Snabbare, högre “fill-rate”, mer finesser.**

**3dfx Voodoo 2: två texturenheter!**

**1999: Transformationer och ljussättning på GPU i första GeForce-kortet. (Möjliggör högre polygonantal.)**

**3dfx tillbakagång, p.g.a. felsatsningar och konflikt med konsoltillverkare mm. NVidia dominrar allt mer.**

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# **2001: Programmerbara shaders**

**Den andra stora revolutionen!**

**GeForce 3 första kortet**

**ATi följe efter med Radeon 8500**

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# **2001-2005**

**Utvecklingen av programmerbara shader går snabbt framåt.**

## **2001: Första generationen (GF3, 8500, GF4)**

Vertex shader: 128 instr. Ej flow control.  
Fragment shader: 8-14 instr.

## **2002: Andra generationen (9000-serien, GF FX)**

Vertex shader: 256 instr. Flow control.  
Fragment shader: 96-512 instr.

## **2004: Tredje generationen (NV 6k/7k-serier, ATI X)**

Vertex shader: 512 instr. Dyn flow control, prediction.  
Fragment shader: >512 instr. Dyn flow control, prediction.

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# **2006: Ny shadermodell**

**Den tredje stora revolutionen**

**NVidia G80, GeForce 8800 första kortet**

**Nytt shaderspråk för GPGPU**

**Totalrevision av hårdvaran**

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

# **2006-2008: Ageia PhysX, fysikkort (PPU)**

**En satsning som gick i stöpet. Ageias PhysX-kort  
gav inga imponerande prestanda. 2008 köps Ageia  
av NVIDIA.**

**2008: Fysik i hårdvara ser ut att vara en ren GPU-  
fråga!**

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)

## **Framtiden**

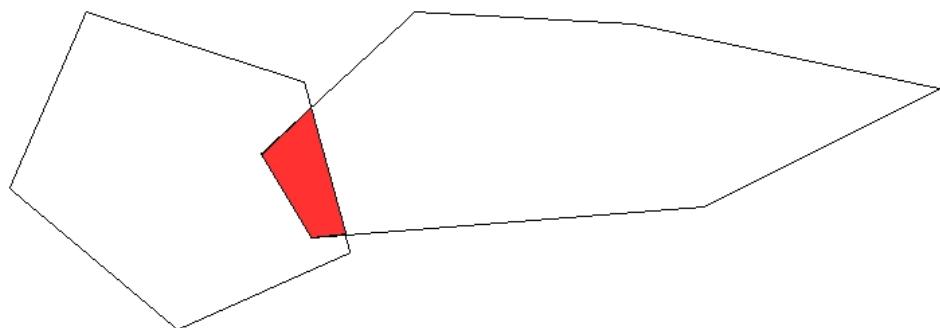
**GPUer har gått från dyra proffssystem till billiga  
konsumentprodukter och mynnat ut i det mest  
lyckade parallelprocessorsystemmet för  
massmarknadsprodukter!**

**GPUer kommer att vara viktiga länge än. Stort  
prestandsförsprång. Tveksamt om CPUer ens kan  
komma ifatt.**

Ingemar  
Ragnemalm  
[ingis@isy.liu.se](mailto:ingis@isy.liu.se)



# Kollisionsdetektering



# Grundkursen

- Space subdivision
- Omskrivande objekt
- Detaljtest: containment + edge tests
- Förenklade fall: sfärer i AABB-värld



# Fas 1: Space subdivision

- Octtree/quad tree
  - BSP-träd
  - Linjärsortering



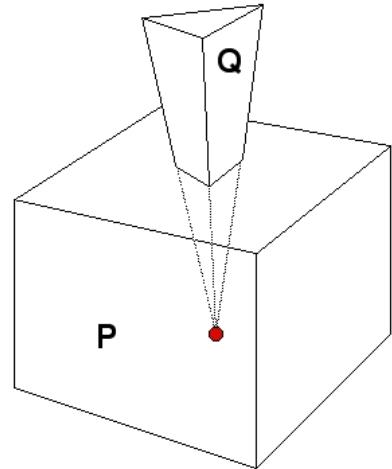
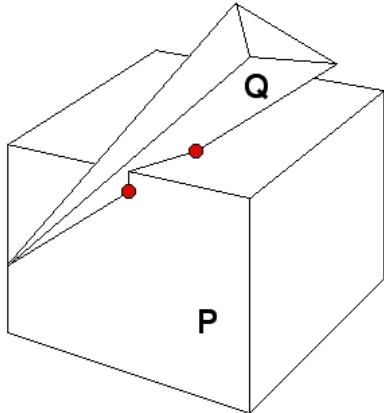
# Fas 2: Omskrivande objekt

- Sfärer
- AABB
- OBB
- k-DOP



## Fas 3: Detaljtest

### Containment+edge tests



## Mer kollisionsdetektering

- Beräkna snittvolym
- Separerande plan (SAT)
- Gilbert-Johnson-Keerthi

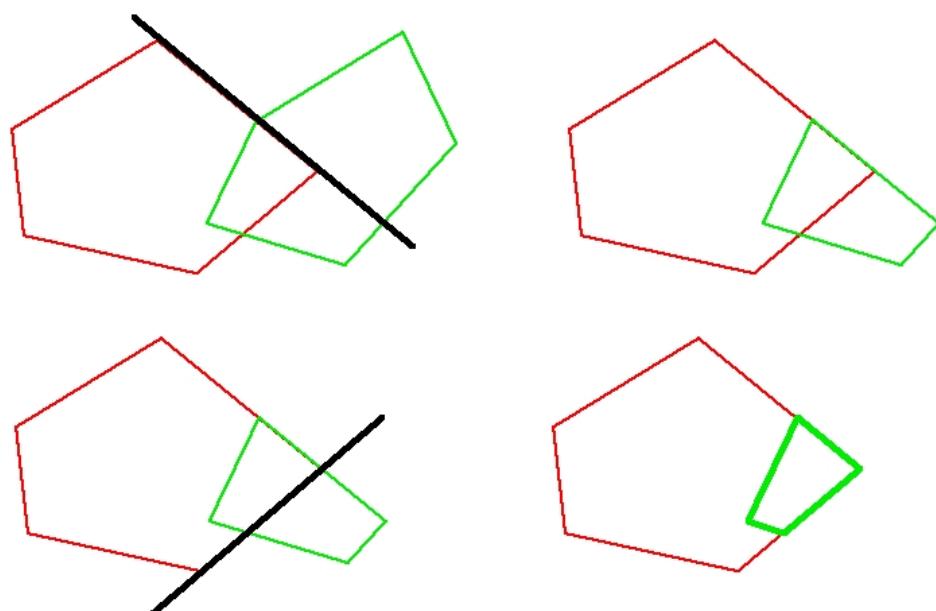


# Snittvolym

Lättare att beräkna än containment/edge!

Containment/edge testar alla vertexar mot alla ytor i andra objektet, full  $O(NM)$

Snittvolym testar alla vertexar i ena mot en yta i taget, minskar nomalt snabbt.  $O(N \log M)$





# Snittvolym

Ger resultat som är användbart för kollisionsrespons!

Hyfsad hastighet för enkla modeller.

Viktigt att börja med rätt plan.

Beräkning av volymen bortkastad när vi inte har överlapp.

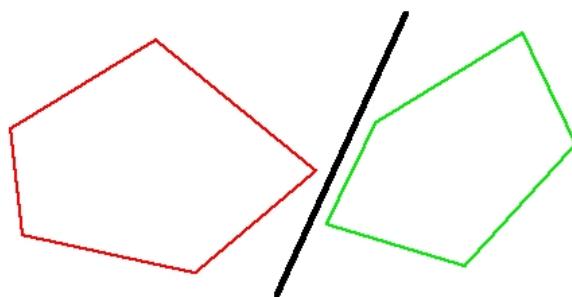
Snabbare tester är möjliga om vi inte behöver resultatet.



# SAT = Separating Axis Theorem

Säger att om två konvexa volymer inte överlappar så finns alltid en axel längs vilken de två volymerna är helt separerade.

I 3D betyder det att det finns ett separerande plan.





## SAT = Separating Axis Theorem

Mycket snabb om vi sparar föregående separerande plan mellan två objekt som är nära varandra.

För sådana fall är komplexiteten  $O(N+M)$  i det typiska fallet.

När planet inte längre stämmer söker man efter ett nytt.

Används bäst med kollisionshantering där objekten inte tillåts överlappa.



## Gilbert-Johnson-Keerthi

Mycket snabb algoritm för kollisionstestning.

Baseras på matematisk morfologi.

Söker iterativt fram de två punkter på objekten som är närmast varandra, samt detekterar överlapp.



## Matematisk morfologi

Läran om former.

Grundoperationer: Krympning och expansion.  
Viss likhet med faltningsoperationer.

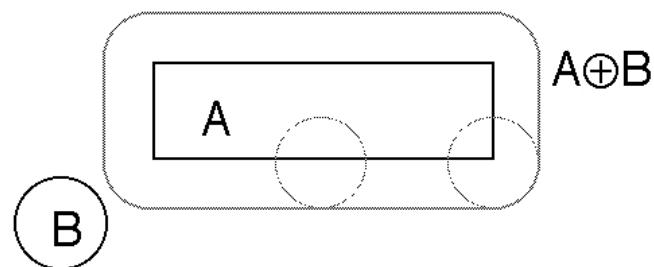
Expansion (dilation) = minkowski addition,  
expanderar en form med en annan

Krympning (erosion) = minkowski subtraction,  
krymper en form med en annan



## Expansion = Minkowskiaddition

Objektet A expanderas med strukturelementet B.

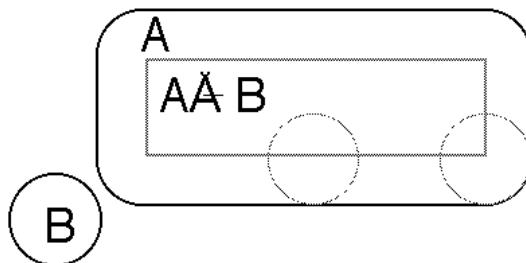


Om B överlappar A är punkten i resultatmängden.



## Krympning = Minkowskisubtraktion

Objektet A krymps med strukturelementet B.



Om hela B ryms i A är punkten i resultatmängden.



## Princip för GJK

GJK utnyttjar Minkowskiaddition mellan de två objekten som skall testas. (Det ena vänt.)

Detta förenklar problemet av sökning av två objekt till sökning av ett.

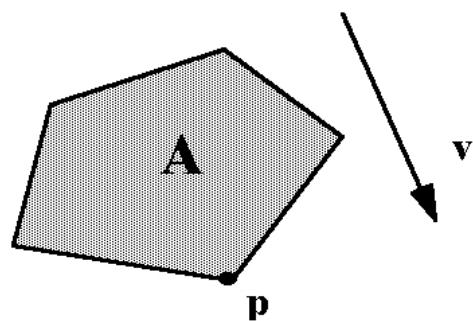
Genom att definiera sökningen med *support mapping* kan sökningen göras linjärt över ytan.



## Support mapping

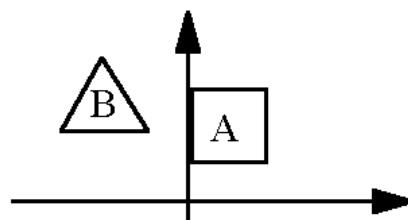
En operation som beräknar extrempunkt längs given axel.

Om objektet läggs på ett plan med axeln som normalvektor så är support mappingen av objektet den punkt som den då står på.



## GJK, exempel

Objekten A och B skall testas enligt nedan.

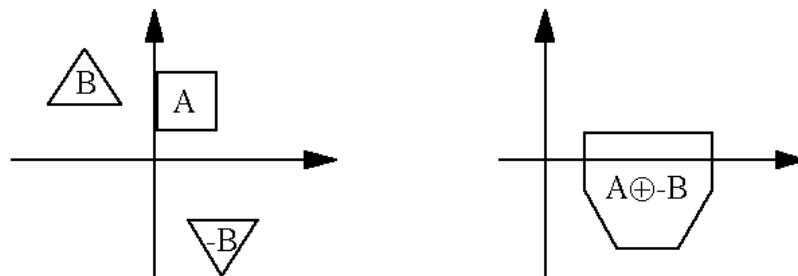




## GJK, exempel

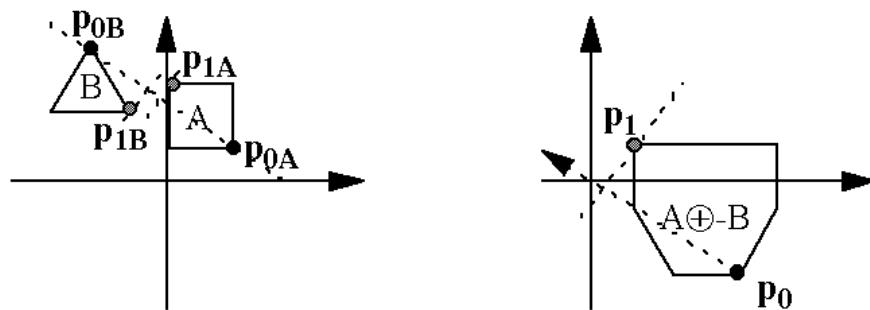
B vänds till -B och Minkowskiadderas med A

Positionen relativt origo är relevant!



## GJK, exempel

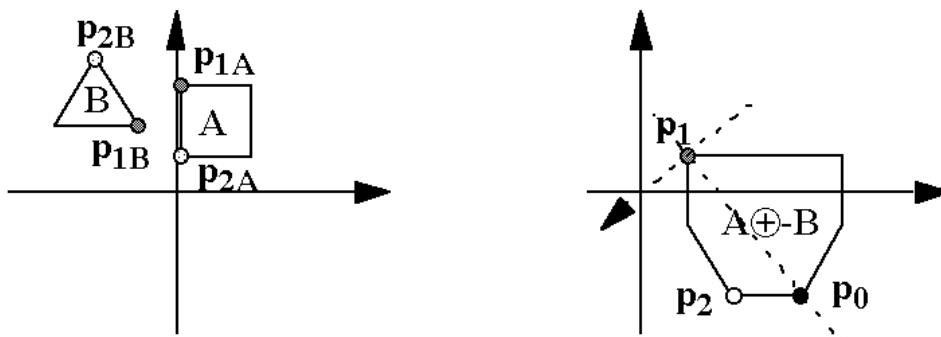
Starta i någon punkt  $p_0$ . Sök support mappingen mot origo. Detta är nästa punkt,  $p_1$ .





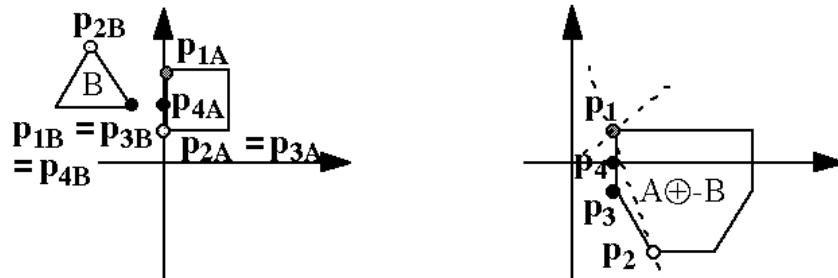
## GJK, exempel

Andra steget: Supportmappning för normalen till  $(p_0-p_1)$ .



## GJK, exempel

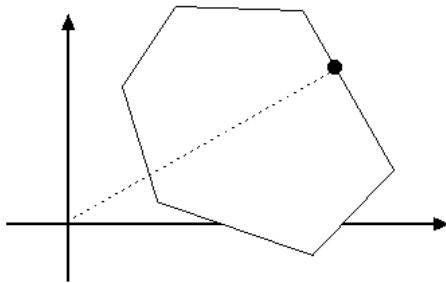
Fortsättning: Tag två nästa punkterna hittills, sök supportmappning för normal mellan dessa tills minimum hittas (ev på yta).





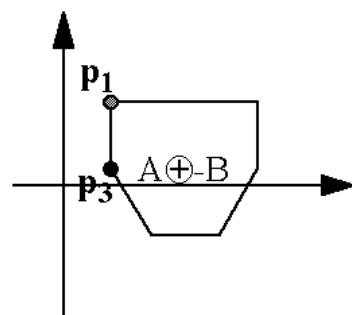
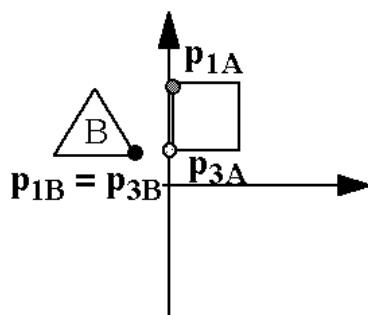
## Beräkning av supportmappning

Kan göras med lokal sökning - men man kan fastna i lokalt minimum första sökningen! Testa om baksida.



## GJK, exempel

Starta nästa gång i punkt från senaste sökningen.  
Ger söktid på så gott som konstant tid!





## GJK, kommentarer

Populär algoritm, mycket snabb, lämplig för komplexa scener med många kollisioner och nära kollisioner.

Kan formuleras så att Minkowskiadditionen aldrig utförs explicit, bara lokalt mellan enstaka punkter.

Minkowskiadditionen med -B kallas ibland "Minkowskisubtraktion". Ej att förväxla med krympning.



## Introduktion till textgenerering

Text är väl inget svårt?

Jo, det är betydligt mer komplext än det verkar, många alternativ, fler än man tror.

- GLUT mm
- Bitmapsfonter
- Förgenererade texturer
  - Texturfonter
  - Outlinefonter
  - 3D-fonter



## Text i OpenGL med GLUT, WGL, AGL...

Färdiga lösningar finns. Lätt tillgängliga.

WGL, AGL mfl: Icke korsplattformslösningar.

GLUT: Funkar... men ser ut som !”#€%&



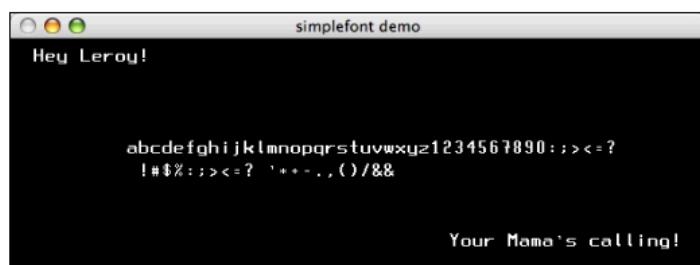
## Bitmapfonter

Enkel lösning baserad på glBitmap.

Varje tecken en display list.

Kan byggas direkt från konstanter i kod.

Kan ej roteras eller skalas.



Min variant: "SimpleFont".  
Smidig för labb-bruk. Bättre  
än GLUTs bitmapfonter.



## Förgenererade texter i texturer

**Man kan alltid lägga in en text i en textur.**

**Kan ligga i bild på disk.**

**Texten kan ritas on-line i bildbuffer av operativsystemets ritfunktioner, sedan laddas upp till textur.**

**Lättdesignat. Ineffektivt för dynamiska texter.**



## Texturfonter

**En hel font i en bild laddas till en textur.**

**Varje tecken laddas till en display list.**

**Snabbt, snyggt, kan roteras och skalas (till en gräns).**

```
! "#$%& ' ( ) *+ , - . /  
0123456789 : ; <=>?  
@ABCDEFGHIJKLMNO  
PQRSTUVWXYZ [ \ ] ^ _  
`abcdefghijklmno  
pqrstuvwxyz { | } ~
```

**Bökgilt att lägga in nya fonter.  
Lättast med monospacefonter.**

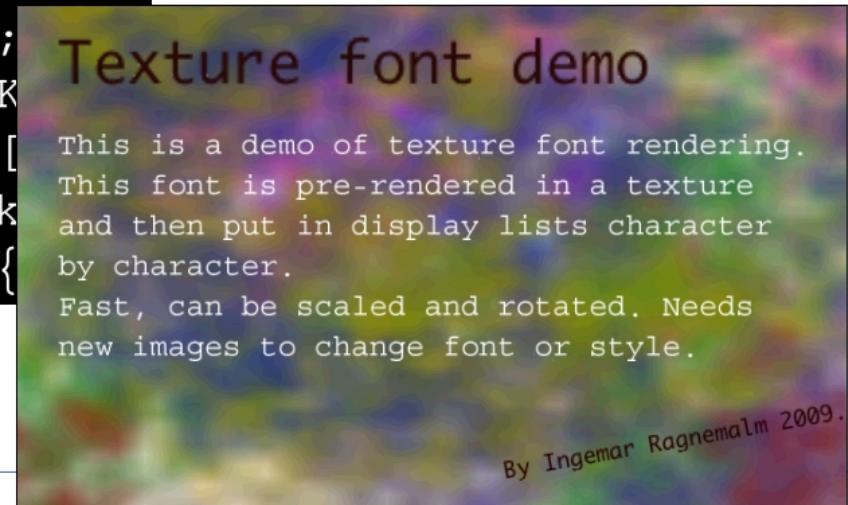
**Bästa lösningen för spel?**



## Texturfonter

Spelprogrammerarnas favorit?

```
! "#$%& ' () *+ , - . /  
0123456789 : ;  
@ABCDEFGHIJK  
PQRSTUVWXYZ [  
`abcdefghijklk  
pqrstuvwxyz {
```



## Outlinefonter

Generera text från TrueType- eller Postscriptfonter.

Korsplattformslösningar finns! (Bl.a. FTGL.)

Bygg polygonobjekt från fontbeskrivning. Kräver speciella trick, "tesselators".

Mer aliasing än texturfonter (lösas med FSAA)





## 3D-fonter

Variant av outlinefonter. Trivial utvidgning, bortsett från behovet att generera normalvektorer.



## Textgenerering

Mina varianter:

*SimpleFont*. Enkel bitmapsfontrenderare.

*TextureFont*. Texturfontrenderare. Bra till mycket. Använder pngLite.

*VectorFont*. Baserad på gammal Mac-demokod. Ej helt portabel än, men nästan. (Bara lite filhantering kvar.)  
Gjord för att slippa det strul jag upplevt med FTGL.

Alla kommer att finnas tillgängliga i god tid till projekten.



## Orthographic mode

2D i OpenGL

OpenGL är inte bara 3D! Med Orthographic mode ställer man in för 2D-bruk.

Texter renderas ofta i 2D.

Spritemotorer kan göras i OpenGL.



## Orthographic mode

```
void Reshape(h, v: Longint); cdecl;
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, h, v, 0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glViewPort(0, 0, h, v);

    gHeight = v;
    gWidth = h;

    glClear(GL_COLOR_BUFFER_BIT); // Clear Screen
}
```

Orthographic mode, pixel coordinates, Y down!



## Anti-aliasing

**Anti-aliasing per primitive:** Gammalt i OpenGL. Dåligt.

**Full-screen anti-aliasing (FSAA):** Finns i OpenGL.  
Supersampling.

**Måste ställas in vid start:** GLUT\_MULTISAMPLE

Aktiveras med glEnable(GL\_MULTISAMPLE);

Funkar inte på alla GPUer.



## Anti-aliasing utan GL\_MULTISAMPLE?

Vad gör man när GL\_MULTISAMPLE inte funkar?

FSAA med FBO!

Rendera till 2x2 större buffer i FBO.

Aktivera interpolation.

Rendera resultatet över hela framebuffern.

Ger perfekt resultat (om den stora buffern är tillåten.)



## Anti-aliasing i FBO



## Slut på ”grafikdelen”

Mer grafiknära som kommer:

- Animation
- Kollisionsrespons
- Beteenden

Vad får man göra projekt på?

- Med i denna kurs - OK
- Med i grundkursen - OK om inte banalt
  - Fråga mig!



## Inlämningsuppgifter

10. Jämför containment/edgetestning, snittvolym och GJK i termer av snabbhet (beräkningskomplexitet).
11. En operation är kritisk för GJK's effektivitet. Vilken? Hur bör den utföras?
12. När en algorim skall implementeras i GPGPU med shaders måste den skrivas om, ofta från grunden, trots att språken kan vara ganska lika. Nämnn två skäl till detta.

Deadline nästa fredag!