GPU-programmering i WebGL

Jørgen Aarmo Lund

Introduksjon

Hvorfor GPU-programmering?

- Stadig mer aktuell
- Hva skjer ”under panseret” når vi tegner GUI-widgets?
- Krever parallell programmering på et annet nivå

Mål

Lage en animasjon av snurrende DIPS-logo

Historie

- 90-tallet: ”3D-akseleratorer”
- 2000: Første programmerbare GPUer
- I dag: GPGPU - general-purpose computing on GPUs

Hva skiller GPUer fra vanlige prosessorer?

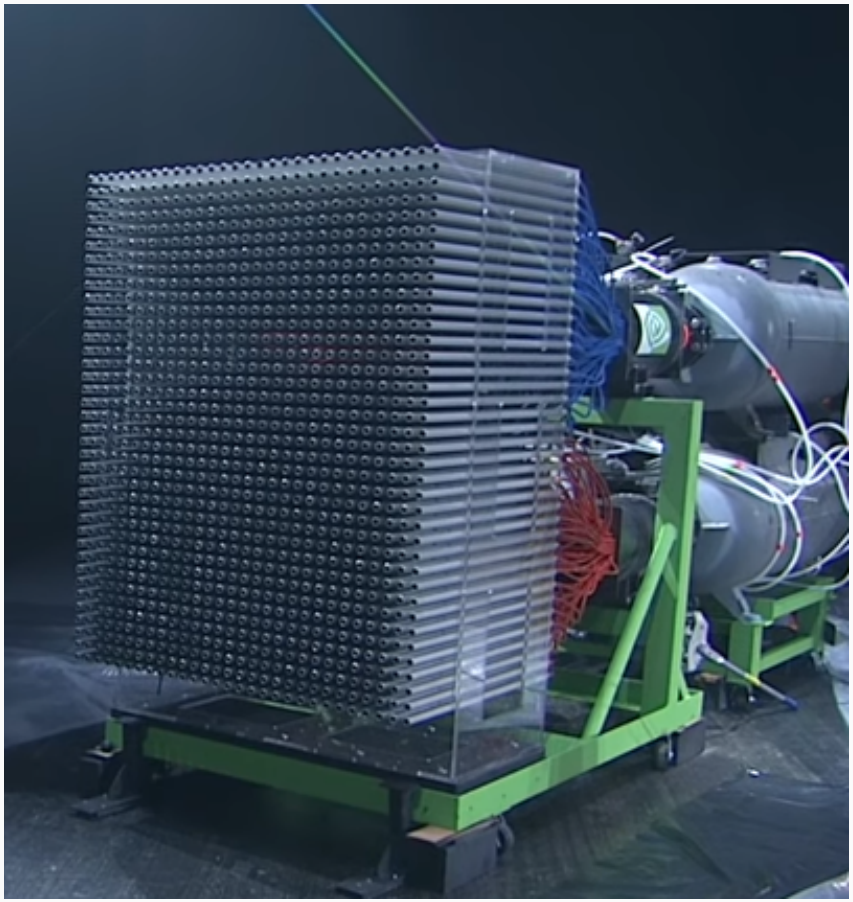
GPUer er massivt parallelle,
designet for gjennomstrømming (throughput)

Mythbusters demonstrerer

https://www.youtube.com/watch?v=-P28LKWTzrI

Hva skiller GPUer fra vanlige prosessorer?

- Massivt parallell arkitektur
  - Mange små kjerner: et *RTX 2080 Super*-kort kommer med over 3000 ”shader processors”
- Throughput
  - Store arbeidsmengder
  - Forsøker å minimere kommunikasjon mellom CPU og GPU



WebGL

- Standard for 2D- og 3D-grafikk i nettleseren
- Bygger på *OpenGL*-standarden
  - GPU-kode (”shadere”) programmeres i *GLSL*, et C-lignende språk
- Støttes av de fleste nettlesere

Forberedelser

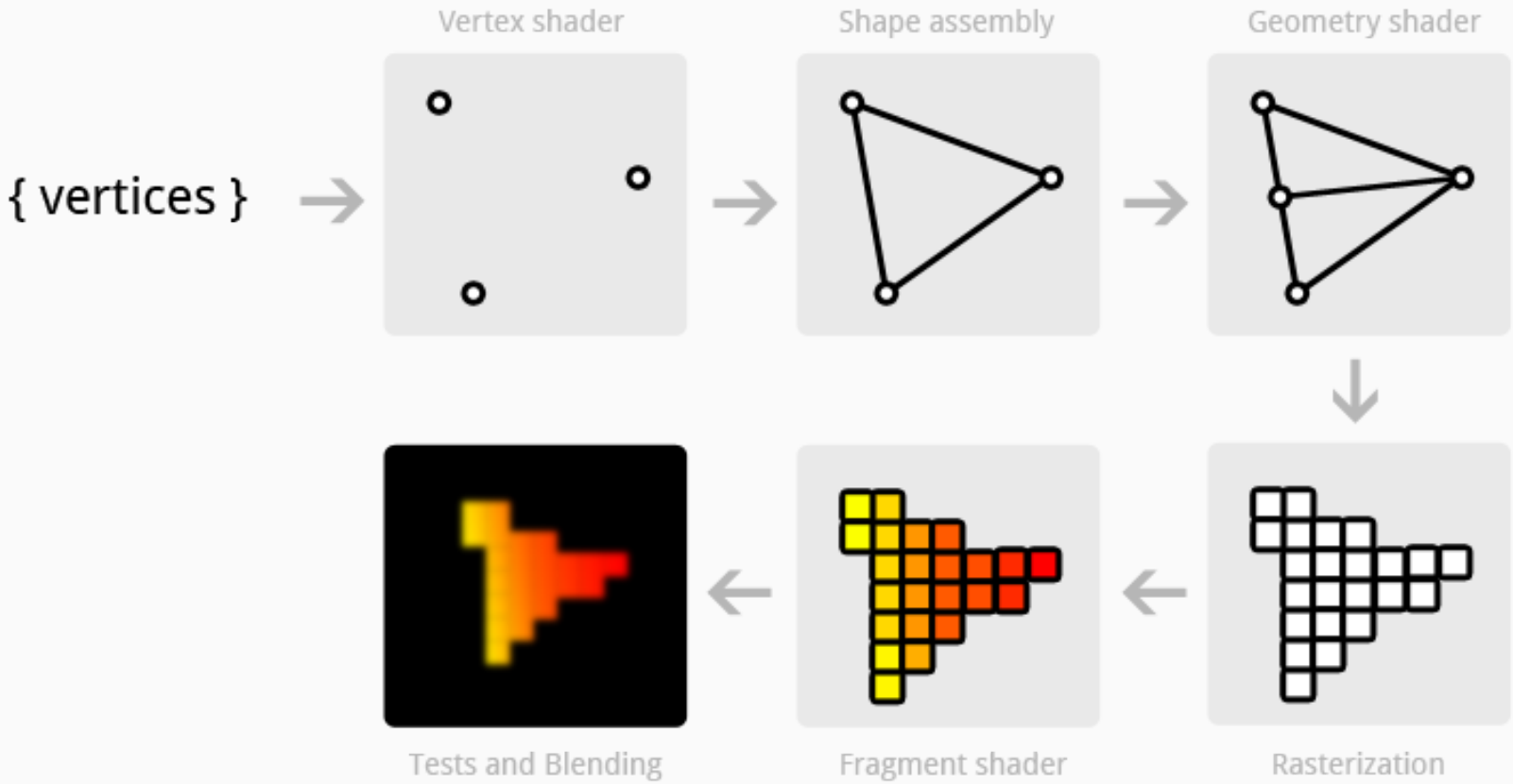
- Kjekt å ha:
  - Firefox eller Chrome
  - Oppdaterte skjermkortdrivere
  - Test nettleseren med *https://get.webgl.org*
- Klon ned repoet fra *https://github.com/jaalu/WebGLWorkshop*
- Mye ”boilerplate”-kode - ferdig kode forberedt

Klar

```
$ git checkout start
```

Hvordan tegne en trekant

Pipeline



(av Alexander Overvoorde, fra *https://open.gl/drawing*)

Hva er vi nødt til å gjøre?

- Implementere steg i pipeline
  - Vertex shader: bestemmer plassering av punkter
  - Fragment shader: bestemmer fargelegging av piksler
- Forberede og overføre data til GPUen
- Be WebGL om å sende figuren vår gjennom pipelinen

Kode

```
$ git checkout part1
```

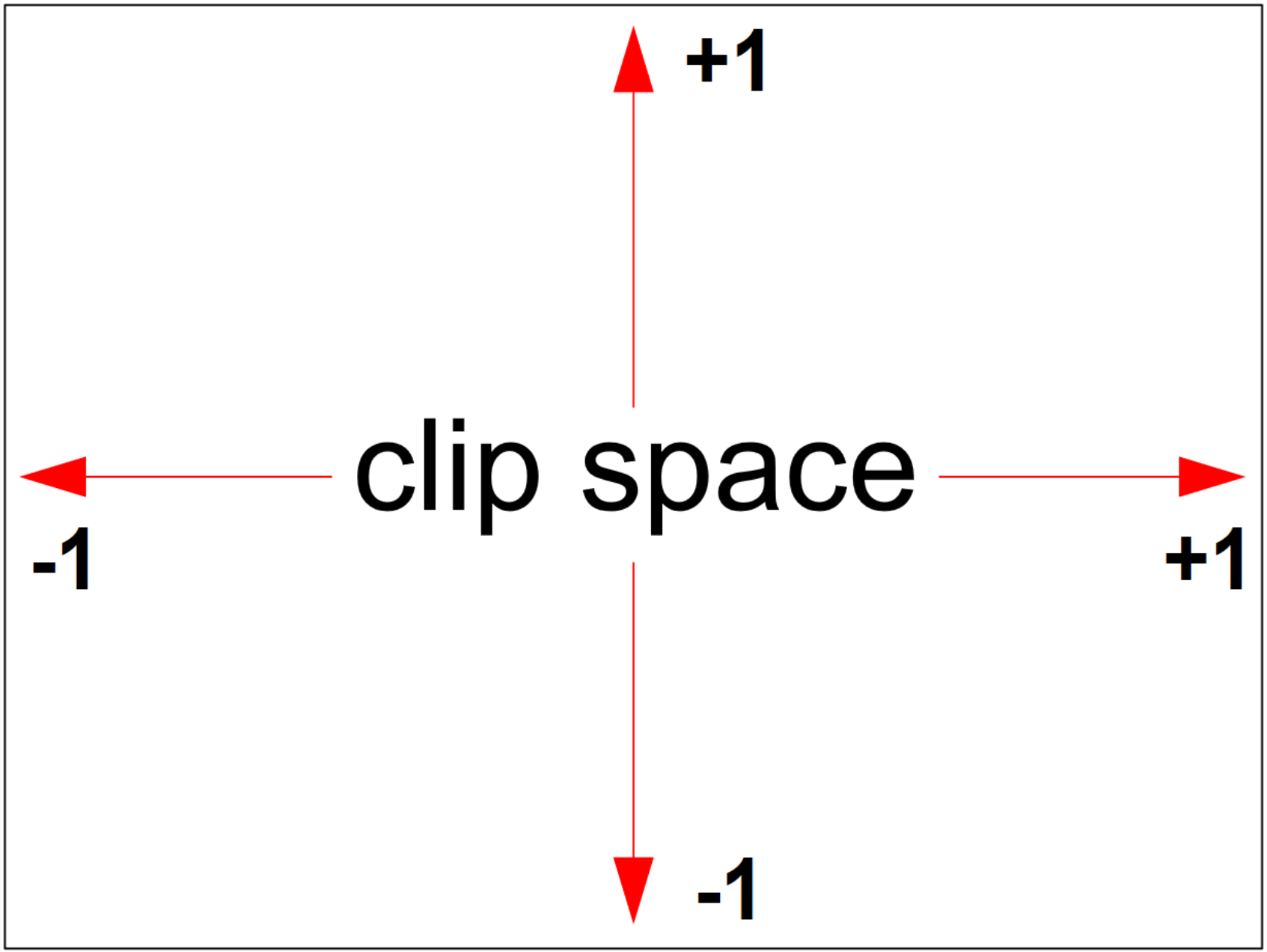
Vertex shader

```
attribute vec4 inputPosition;

void main() {
    gl_Position = inputPosition;
}
```

Endelig plassering settes i *gl_Position*
”inputPosition” blir hentet fra et *buffer*

Koordinatsystem



Fragment shader

```
void main() {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Endelig farge settes i *gl_FragColor*
(Farger i RGBA - rødt, grønt, blått, ”alpha”)

Sette opp pipelinen

- Kildekode knyttes til shadere, kompileres
- Deretter kobles de kompilerte shaderne sammen til et *program*
- Programmet gir hele stien fra punkter til ferdig grafikk på skjermen

Laste inn data

- Vi lager et *buffer* for koordinatene
- Bruker *bindBuffer*, *bufferData* for å overføre data fra minnet til GPU-minnet
- Vi må uttrykkelig beskrive hvordan GPUen skal tolke punkter fra bufferet (*vertexAttribPointer*)

Tegning

- Vi har definert shaderkode, fortalt GPU hvor data skal
- Kjører *`usePrograM`* for å fortelle hvilket shaderprogram som skal brukes
- Endelig kan vi kjøre *`drawArrays`* for å signalisere at vi skal tegne former

Oppgradering

- Litt større modell i filen *logo.js*, i form av listen *logoCoordinates*

```
$ git checkout logo
```

Transformasjoner

Transformasjoner

- Hvordan roterer vi logoen?

Trigonometri

$$x' = x\cos\theta - y\sin\theta$$

$$y' = x\sin\theta + y\cos\theta$$

Transformasjonsmatriser

- Tungvint å gjøre dette for hånd
- Har mange punkter som må transformeres på samme måte - der GPU briljerer!
- Lager en *transformasjonsmatrise* for rotasjonen:

$$R\mathbf{v} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

- Bruker biblioteket *glMatrix* (`http://glmatrix.net/`)

Sende data til GPU

- Legger til matrisen som del av vertex-shaderen:

```
attribute vec4 inputPosition;

uniform mat4 transformationMatrix;

void main() {
    gl_Position = transformationMatrix * inputPosition;
}
```

- ”Uniform” - lik for alle punktene

Ferdig kode

```
$ git checkout part2
```

Animasjon

Animasjon

- Hvordan tegner vi mer enn ett bilde?
  - `requestAnimationFrame`
  - Tar et callback for å tegne et nytt bilde
  - Må kalles kontinuerlig for å fortsette animasjon
- I callback, for hvert bilde:
  - Regn ut ny transformasjon
  - Visk ut bilde
  - Tegn nytt bilde

Ferdig kode

```
$ git checkout part3
```

Konklusjon

Konklusjon

- Har implementert enkel animasjon
- Massivt parallelt
  - Vi definerer operasjon for hvert punkt og hver piksel
- Må aktivt ta stilling til hva som skal sendes til GPU
  - Bufre, variabler

GPGPU

Rammeverk for GPGPU:

- CUDA: Nvidias GPGPU-rammeverk, bindinger til Python og .NET
  - Hybridizer, Alea: .NET-IL til CUDA
- OpenCL: Åpen GPGPU-standard, bindinger til Python og .NET
  - OpenTK, Silk.NET
- *GPU.js* (`https://gpu.rocks/`) - Transpilering av JavaScript til shaderkode i nettleseren

Spørsmål/diskusjon?