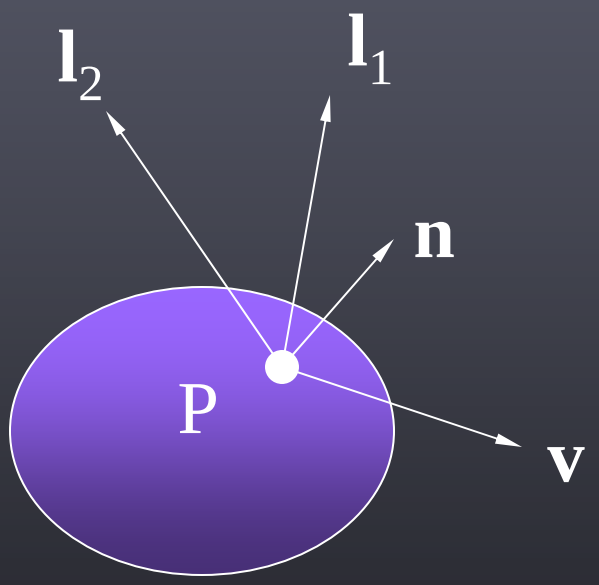
Shaders

Gustav Taxén
gustavt@csc.kth.se

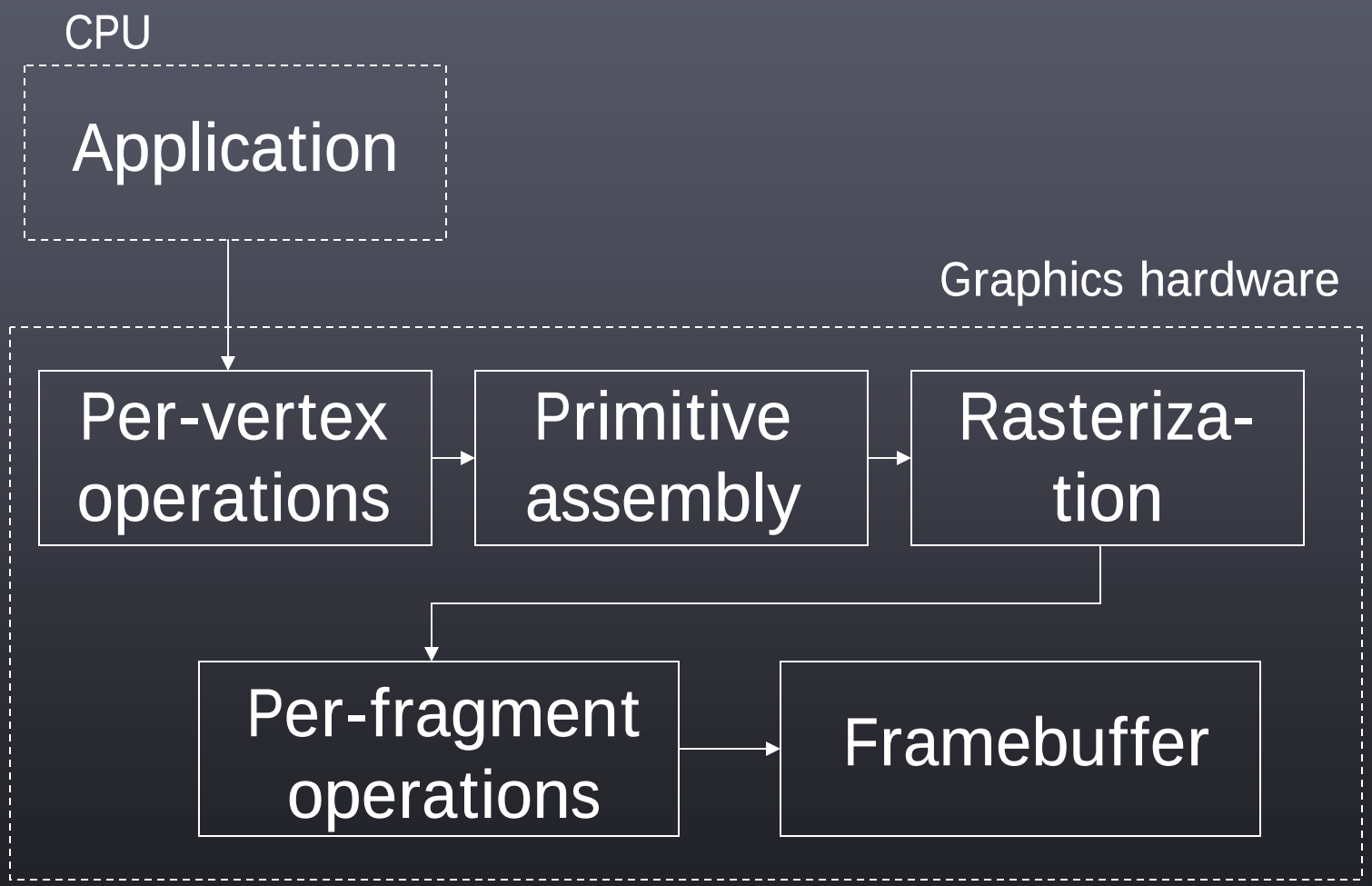
2D1640 Grafik och Interaktionsprogrammering VT 2007

Shading

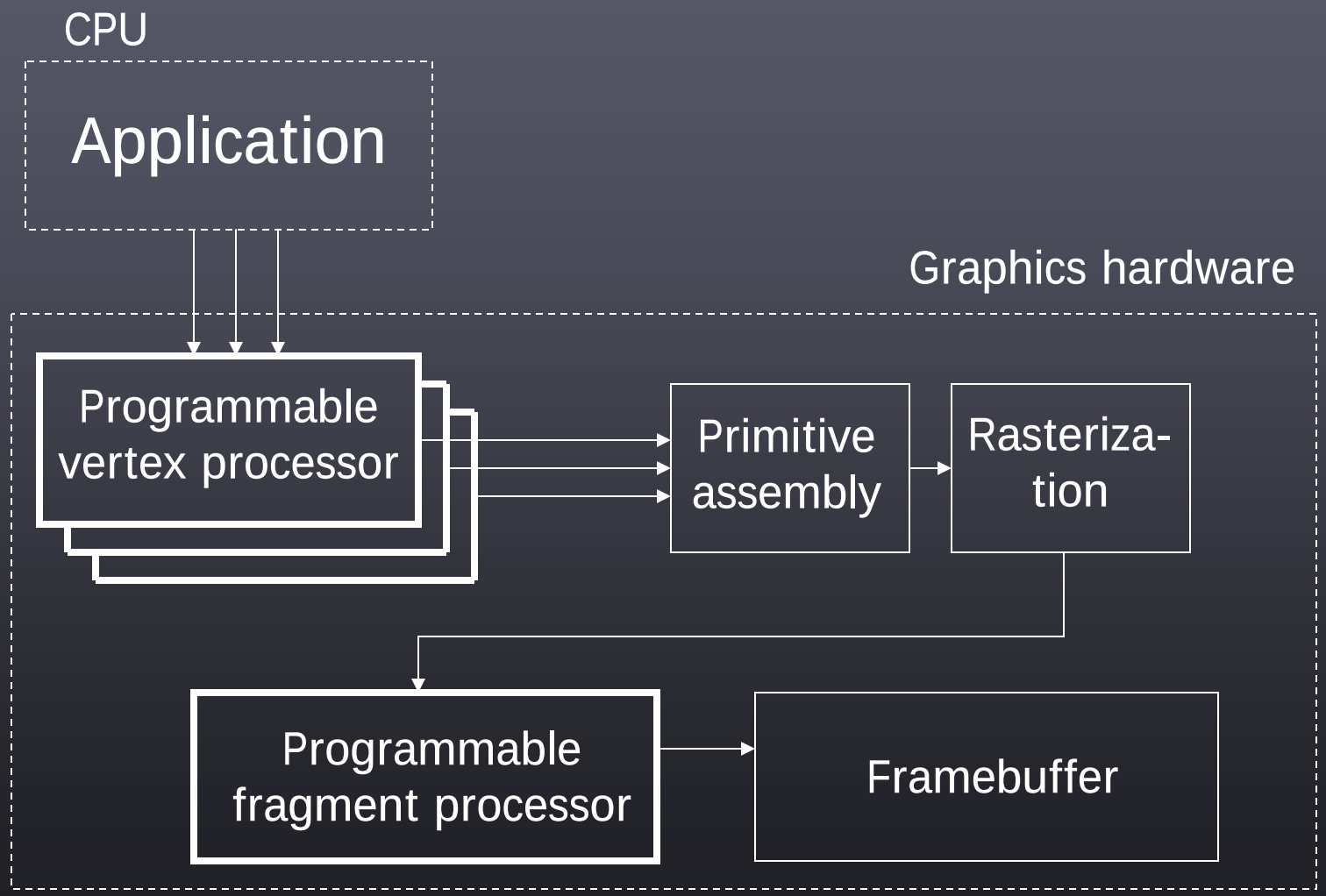


Givet en punkt P på en yta, en normal $\mathbf{n}$, riktningsvektorer $\mathbf{l}_i$ mot ljuskällor och en kamerariktning $\mathbf{v}$, ta reda på vilken färg P ska ha.

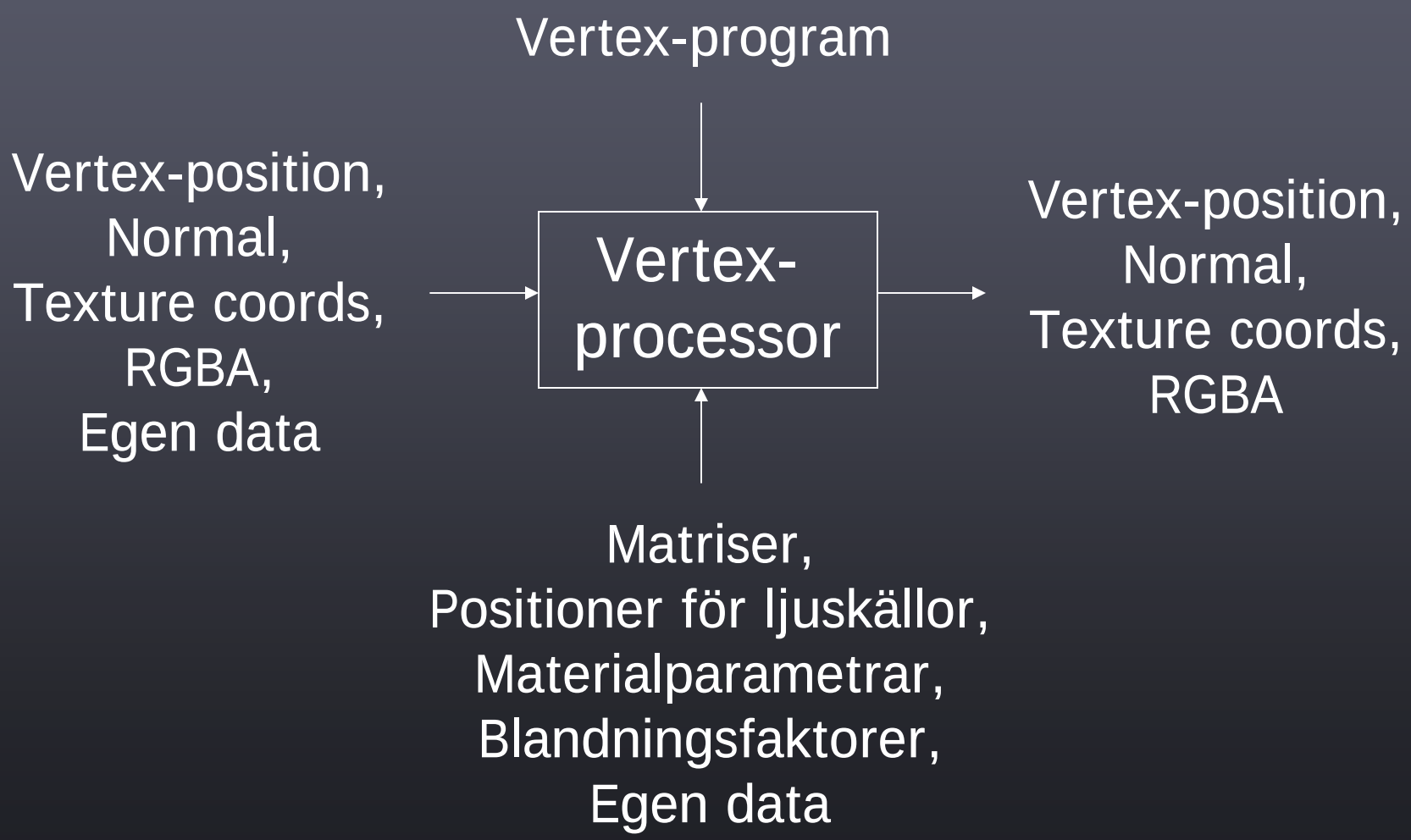
Fixed-function-pipelines



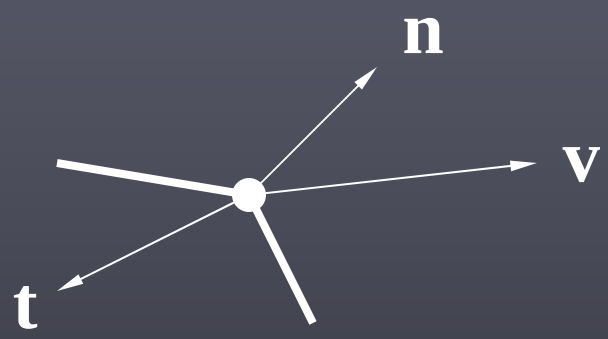
Programerbara pipelines



Vertexprocessorer

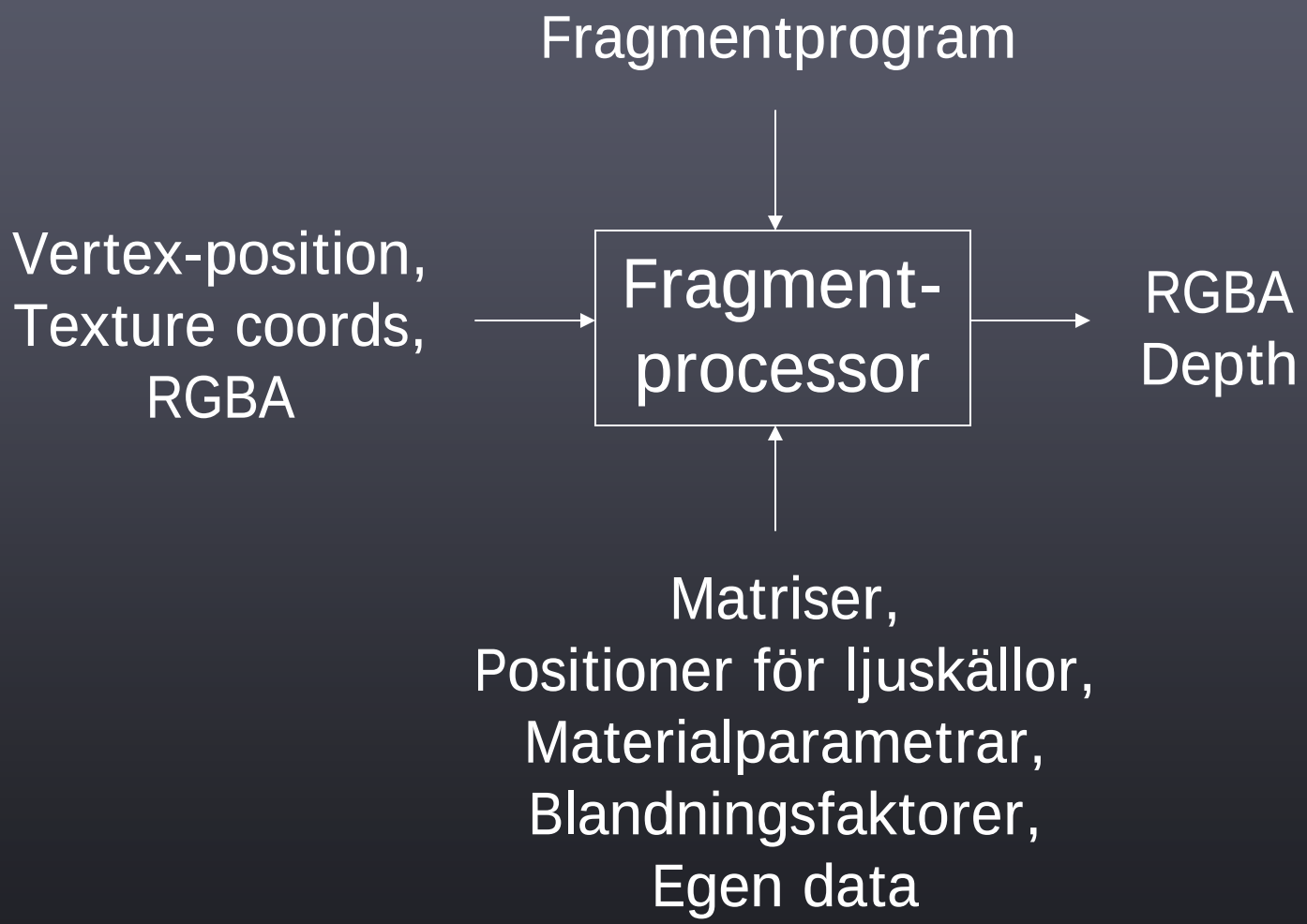


Vertexprocessorer




```
T = transmission_vector(N,V);
(s,t) = to_background_tex_coord(T);
color = texture_lookup(s,t);
```

Fragmentprocessorer



Fragmentprocessorer

```
Ctex =
  texture_lookup("wall",s,t);
Ntex =
  tex_lookup("normal_map",s,t);

color = Ctex * dot(Ntex, L);
```

Syntax-exempel

D3D Vertex Shader Assembler

```
dp3 r7.w, VECTOR_VERTEXTOLIGHT, VECTOR_VERTEXTOLIGHT
rsq VECTOR_VERTEXTOLIGHT.w, r7.w
dst r7, r7.wwww, VECTOR_VERTEXTOLIGHT.wwww
mul r6, r5, ATTENUATION.w
```

OpenGL Vertex Program Assembler

```
MOV   R1, R2.yzwx;
MUL   R1.xyz, R2, R3;
ADD   R1, R2, -R3;
RCP   R1, R2.w;
```

nVidia Cg

```
struct app2vert : application2vertex {
    float3 Position; //in object space
    float3 Normal;
    float2 TexCoord;
};
struct vert2frag : vertex2fragment {
    float4 HPosition; // in projection space
    float4 TexCoord0;
};

vert2frag main(app2vert IN,
  uniform float4x4 ModelViewProj,
  uniform float4x4 TexTransform) {
    vert2frag OUT;
    float4 HPosition;
    HPosition.xyz = IN.Position;
    HPosition.w = 1.0;
    OUT.HPosition = mul (HPosition , ModelViewProj);
    OUT.TexCoord0 = mul (HPosition, TexTransform);
    return OUT;
}
```

Andra alternativ

- Microsoft DirectX9 / DirectX10:
  High-Level Shading Language (HLSL)
- OpenGL 2.0:
  OpenGL Shading Language

- HLSL är i princip ekvivalent med Cg
- GLSL skiljer sig inte alltför mycket

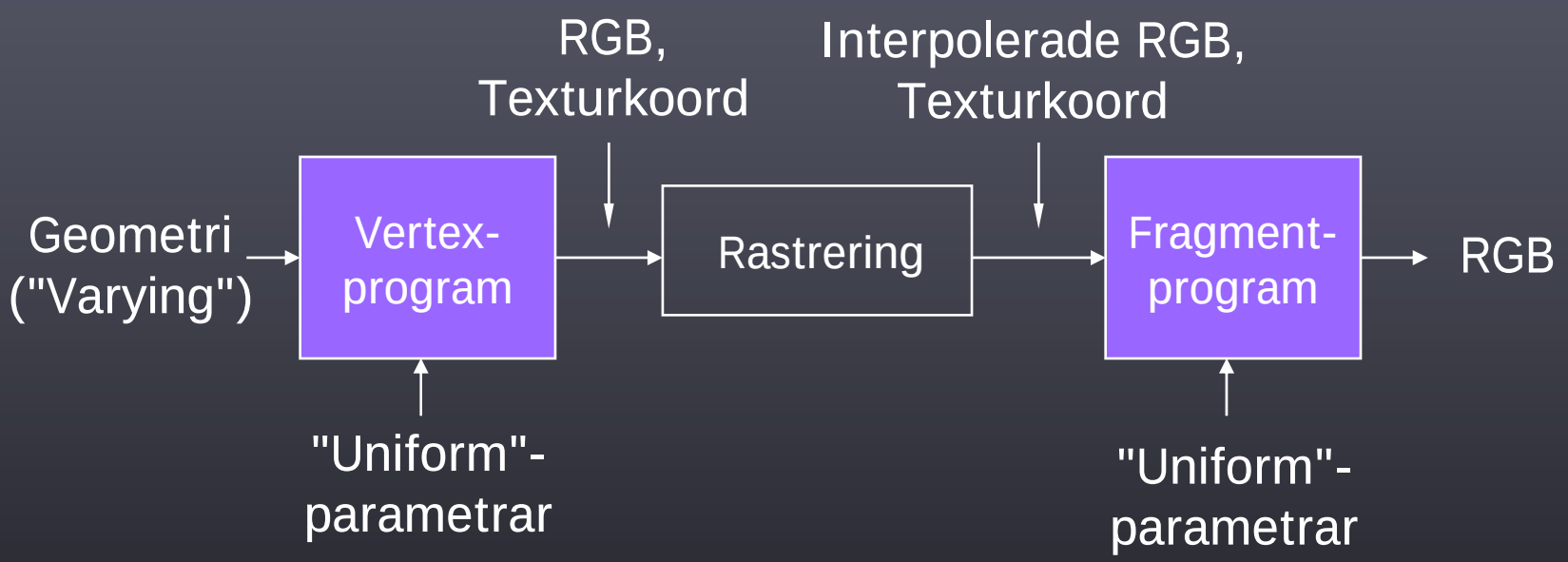
Vi är mitt i en revolution...

- Vertexprogram och fragmentprogram kan skrivas i ett enda språk och kompileras till OpenGL och Direct3D
- Högnivåshaders med loopar m.m. kan användas direkt i realtidsapplikationer
- Designers och formgivare kan exportera sina Maya/3DSMax-material direkt till hårdvaran

Realtidsgrafik idag

http://www.nvidia.com/

Cg och OpenGL



Cg och OpenGL

- Vertexprogram ersätter OpenGLs transformation- och ljussättningssteg
- Fragmentprogram ersätter (delvis) OpenGLs "operationer på fragment"
- Så vi måste sköta transformationer, färgblandning och texturkombinationer själva!

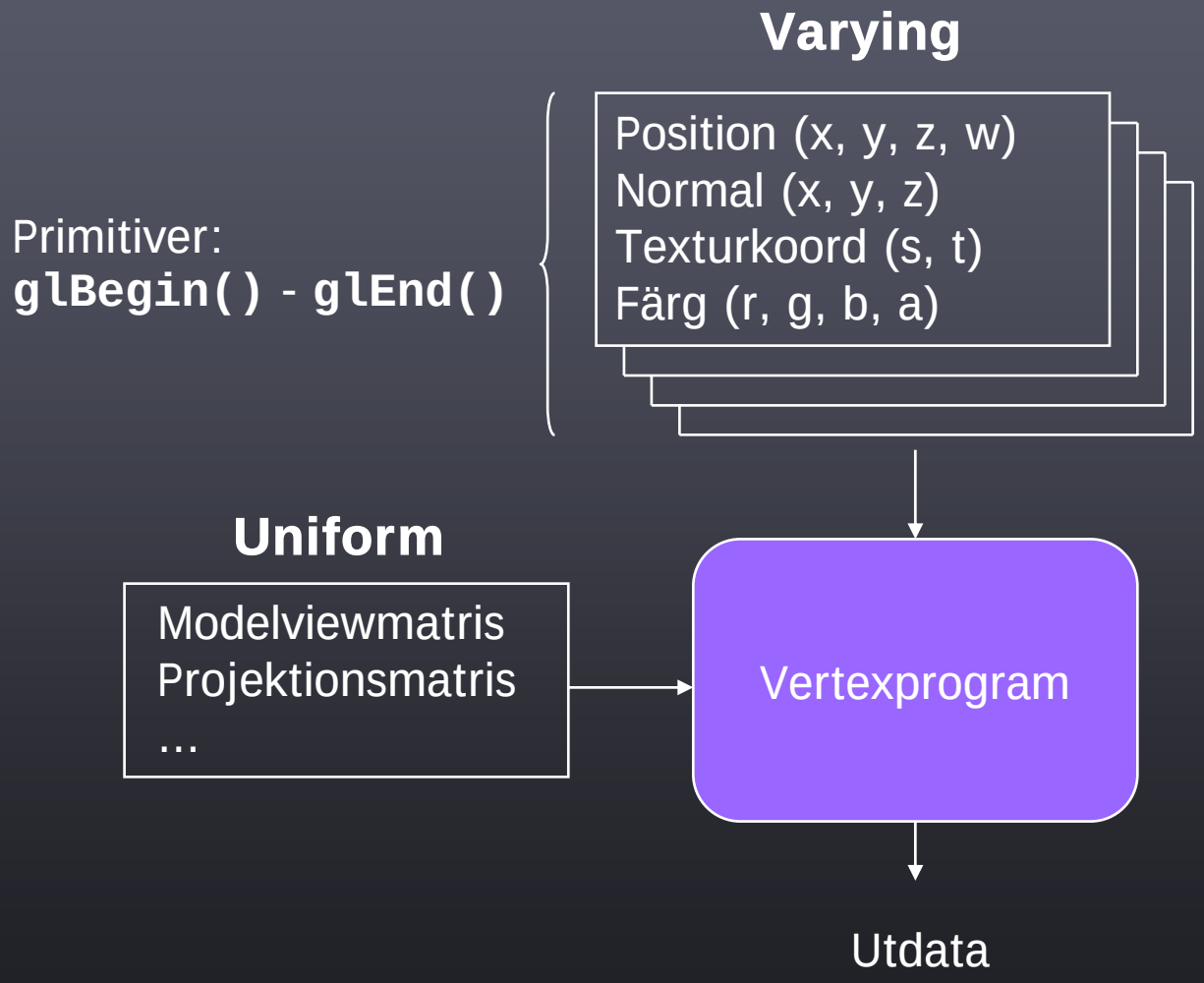
Cg profiles

- Olika grafikkort har olika funktionalitet
- Man kan kompilera Cg-program för olika typer av grafikkort, s.k. profiler ("**profiles**")
- Fråga grafikkortet vilken profil som stöds

Vertexprogram

- Indata per hörn ("**varying**"):
  - Position (måste alltid finnas)
  - Normal
  - Texturkoordinater
  - Färg
  - Användardefinierade parametrar (skalär, 2-, 3-, 4-vektor)
  - (Och lite annat också)
- Fasta parametrar: samma värde under hela exekveringen ("**uniform**")

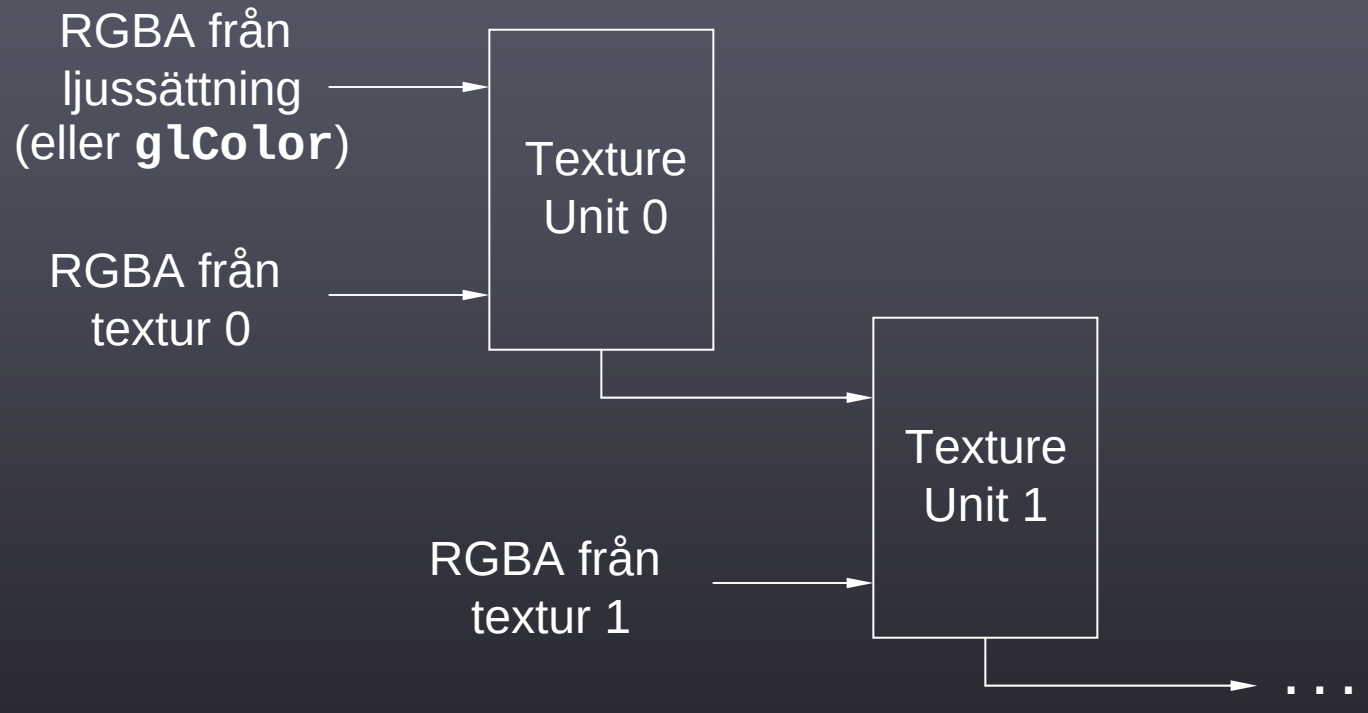
Vertexprogram



Fragmentprogram

- Indata till fragmentprogrammet **måste** vara samma som utdata från vertexprogrammet (+ ev. uniform-parametrar)
- Fragmentprogram **måste** ha en färg (RGBA) som utdata

Multitexturing



Texturkoordinater som indata

- Multitexturing har ingen effekt i en programmerbar pipeline
- Texturkoordinater är 4D-vektorer!
- Så man utnyttjar texturkoordinaterna för att skicka varying-data!
- Till vertex-program och mellan vertex- och fragment-program

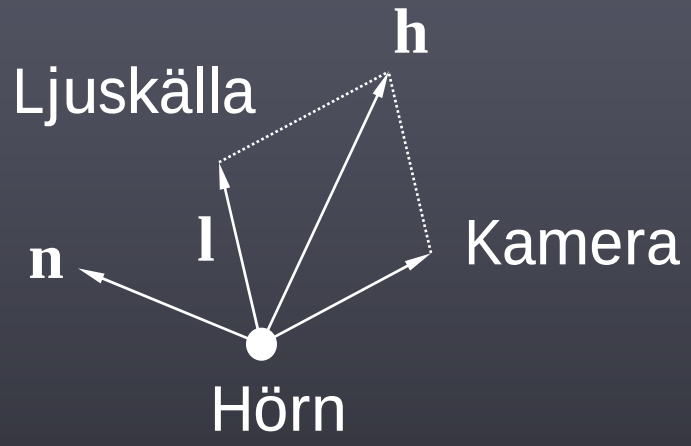
Att köra Cg-program (OpenGL)

- Skapa en kontext där programmen lagras
- Välj profile
- Kompilera (eller hämta från binärfil)
- Aktivera en profil och ett program
- Knyt parametrar
- Rita geometrin

Exempel

Vi skriver ett par enkla cg-program "live".

Phong + Phong shading



$$\mathbf{c} = \mathbf{l}_{\text{col}} \cdot (k_d \cdot (\mathbf{n} \bullet \mathbf{l}) + k_s \cdot (\mathbf{n} \bullet \mathbf{h})^{shininess})$$

Phong + Phong shading: Vertexprogram

- Varying:
  - Position
  - Normal
  - Materialparametrar (**`Kd`**, **`Ks`**)
- Uniform:
  - Modelview-matrisen och projection-matrisen
  - Modelview-matrisen
  - Inv. transp. av modelview-matrisen

```
void main(float4 Pobject      : POSITION,
          float3 Nobject      : NORMAL,
          float3 diffuse      : TEXCOORD0,
          float3 specular     : TEXCOORD1,
          uniform float4x4 ModelViewProj,
          uniform float4x4 ModelView,
          uniform float4x4 ModelViewIT,

          out float4 HPosition    : POSITION,
          out float3 Peye         : TEXCOORD0,
          out float3 Neye         : TEXCOORD1,
          out float3 Kd           : COLOR0,
          out float3 Ks           : COLOR1)
{
    HPosition = mul(ModelViewProj, Pobject);
    Peye = mul(ModelView, Pobject).xyz;
    Neye = mul(ModelViewIT, float4(Nobject, 0)).xyz;
    Kd = diffuse;
    Ks = specular;
}
```

Indata

Utdata

Indata

```
float4 Pobject       : POSITION
```

Datatyp
(Här: 32-bit 4D-vektor)

Parameternamn

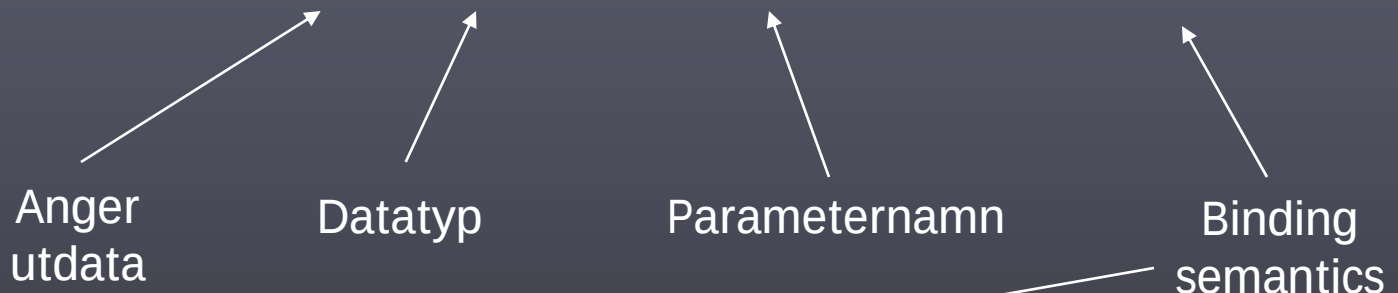
Hur Cg ska
tolka parametern
("binding semantics")

POSITION	Position
NORMAL	Normal
TEXCOORD0	Texturkoord för enhet 0
TEXCOORD1	Texturkoord för enhet 1
...	
COLOR eller **COLOR0**	RGB(A)-färg

m.fl.

Utdata

```
out float4 HPosition    : POSITION
```



POSITION eller **HPOS**	Position (homogena koordinater)
TEXCOORD0	Texturkoord för enhet 0
TEXCOORD1	Texturkoord för enhet 1
…	
COLOR0	RGB(A)-färg
m.fl.	

Programkod

```
{
  HPosition = mul(ModelViewProj,
                  Pobject);

  Peye = mul(ModelView, Pobject).xyz;
  Neye = mul(ModelViewIT,
             float4(Nobject, 0)).xyz;

  Kd = diffuse;
  Ks = specular;
}
```

Modelview · Projection

Objektkoord för hörnet

Hörnets ögonkoord

$(\text{Modelview}^{-1})^{T}$

Nobject = hörnets normal

Skicka vidare **Kd** och **Ks** till fragment-programmet

cgGLSetStateMatrixParameter() används i OpenGL-koden för att koppla matriserna till Cg-parametrar

Phong + Phong shading: Fragmentprogram

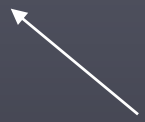
- Varying:
  - Vertexposition i ögonkoordinater
  - Normal i ögonkoordinater
  - **Kd** och **Ks**
- Uniform:
  - Ljuskällans position i ögonkoordinater
  - Ljuskällans färg (RGB)
  - Shininess

```
half diffuse(half4 l) { return l.y; }
half specular(half4 l) { return l.z; }

half4 main(float3 Peye         : TEXCOORD0,
           half3 Neye          : TEXCOORD1,
           half2 uv            : TEXCOORD2,
           half3 Kd            : COLOR0,
           half3 Ks            : COLOR1,
           uniform float3 Plight,
           uniform half3 lightColor,
           uniform half3 shininess) : COLOR
{
    half3 N = normalize(Neye);
    half3 L = normalize(Plight - Peye);
    half3 V = normalize(-Peye);
    half3 H = normalize(L + V);
    half NdotL = dot(N, L), NdotH = dot(N, H);
    half4 lighting = lit(NdotL, NdotH, shininess);
    half3 C = lightColor *
        (diffuse(lighting) * Kd + specular(lighting) * Ks);
    return half4(C, 1);
}
```

Indata och utdata

```
half4 main(...)
```



16-bitars 4D-vektor
(tolkas automatiskt som RGBA för fragmentprogram)

Programkod

```
half3 N = normalize(Neye);
half3 L = normalize(Plight - Peye);
half3 V = normalize(-Peye);
half3 H = normalize(L + V);
```

Normera normalen

Beräkna vektorn H
(och normera den)

Programkod (forts)

```
half diffuse(half4 l) { return l.y; }
half specular(half4 l) { return l.z; }
```

Hjälpfunktioner (se nedan)

```
half NdotL = dot(N, L), NdotH = dot(N, H);
```

← (N • L) och (N • H)

```
half4 lighting = lit(NdotL,
                     NdotH,
                     shininess);
```

←

x-värdet: 1
y-värdet: (N • L) om > 0, annars 0
z-värdet: $(N \bullet H)^{shininess}$ om > 0, annars 0
w-värdet: 0

```
half3 C = lightColor *
  (diffuse(lighting) * Kd +
   specular(lighting) * Ks);

return half4(C, 1);
```

← RGBA (A är normalt 1)

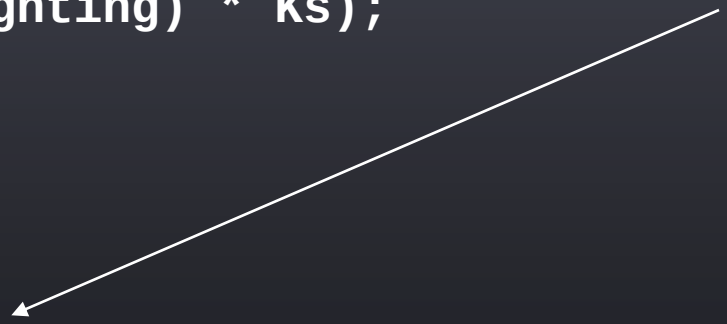
Resultat

Texturer

- Skapa texturobjekt i OpenGL (som vanligt)
- Uniform-parametrar i fragmentprogrammet
- Koppla aktivt OpenGL-texturobjekt till Cg-parameter med kommandot **`cgGLSetTextureParameter()`**
- Aktivera texturering med **`cgGLEnableTextureParameter()`**

Texturer (forts)

```
half4 main(float3 Peye          : TEXCOORD0,
           ...
           half2 uv             : TEXCOORD2,
           ...
           uniform sampler2D diffuseMap,
           ...
           uniform half3 shininess) : COLOR) {
  ...
  half3 C = lightColor *
  (diffuse(lighting) * Kd * (half3)tex2D(diffuseMap, uv).xyz +
   specular(lighting) * Ks);
  ...
}
```



`tex2D(diffuseMap, uv)` ger RGBA för texturen på position **`uv`**.
Detta tolkas om som en 16-bits 3D-vektor (RGB): x, y och z tas med.

Demo

Mer info

http://developer.nvidia.com/