Analiza możliwości i porównanie nowych języków do programowania grafiki

Na przykładzie *C for Graphics*
i *OpenGL 2.0 Shading Language*

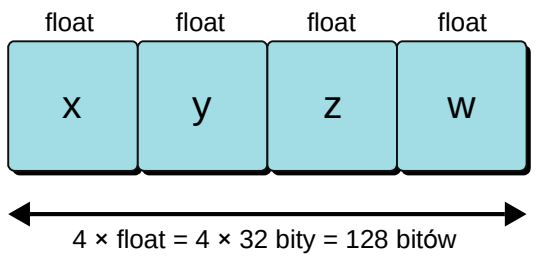
autor: Marcin Rociek
promotor: dr inż. Witold Alda

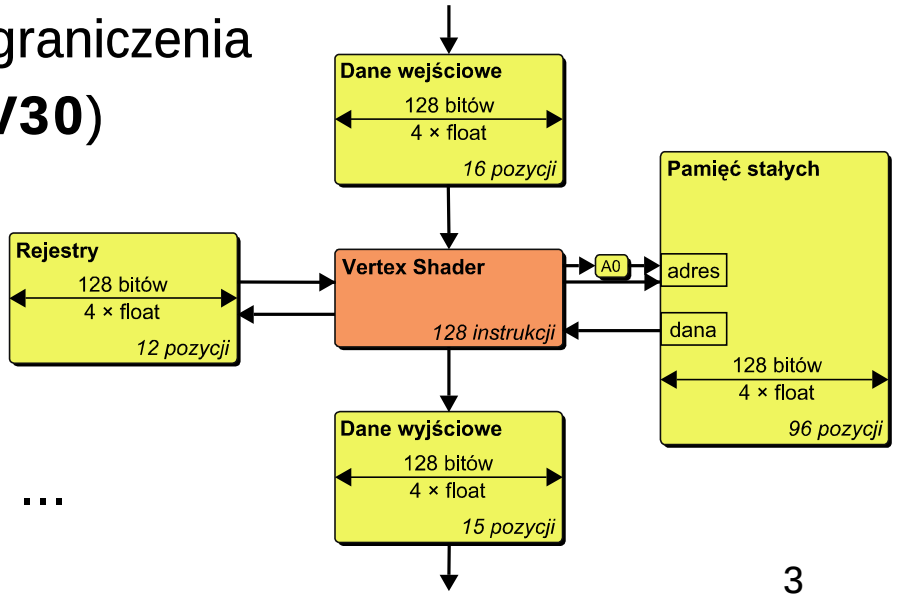
Wprowadzenie

- 2 tory rozwoju grafiki:
  - Grafika interaktywna (gry)
    - Czas renderingu: < 33ms
    - Wydajne karty graficzne
    - Niewygodne programowanie
  - Grafika o wysokiej jakości (filmy)
    - Czas renderingu liczony nawet w godzinach
    - Duże komputery
    - Zaawansowane metody opisu wyglądu (shadery)
- Postęp technologiczny powoduje zmniejszenie dystansu

Współczesne akceleratory graficzne

- Rozwój szybszy niż rozwój CPU
- Ewolucja: bierny bufor ramki → akcelerowana „czarna skrzynka” → programowalny GPU
- Typowo graficzne dane (wektory) i instrukcje (**`dp3`**, **`lit`**, **`mad`**, ...)
- Rok 2001: *NVIDIA GeForce3* (**NV20**)
  - Vertex Shader
    - 128 instrukcji, brak pętli
  - Fragment Shader
    - 12 instrukcji, 4 tekstury, ograniczenia
- Rok 2002: *NVIDIA GeForceFX* (**NV30**)
  - Vertex Shader
    - 256 instrukcji, pętle, podprocedury, ...
  - Fragment Shader
    - 1024 instrukcji, 16 tekstur, ...





Program dla Vertex Shadera

- Przykładowy kompletny program (13 instrukcji):

```
# przekształcenie współrzędnych
dp4 o[HPOS].x, c[0], v[OPOS];
dp4 o[HPOS].y, c[1], v[OPOS];
dp4 o[HPOS].z, c[2], v[OPOS];
dp4 o[HPOS].w, c[3], v[OPOS];
# obliczenie wektora normalnego N'
dp3 r0.x, c[4], v[NRML];
dp3 r0.y, c[5], v[NRML];
dp3 r0.z, c[6], v[NRML];
dp3 r1.x, c[32], r0;     # r1.x = L o N'
dp3 r1.y, c[33], r0;     # r1.y = H o N'
mov r1.w, c[38].x;
lit r2, r1;              # obliczenie współczynników oświetlenia
mad r3, c[35].x, r2.y, c[35].y;      # diffuse + ambient
mad o[COL0].xyz, c[36], r2.z, r3;    # + specular
```

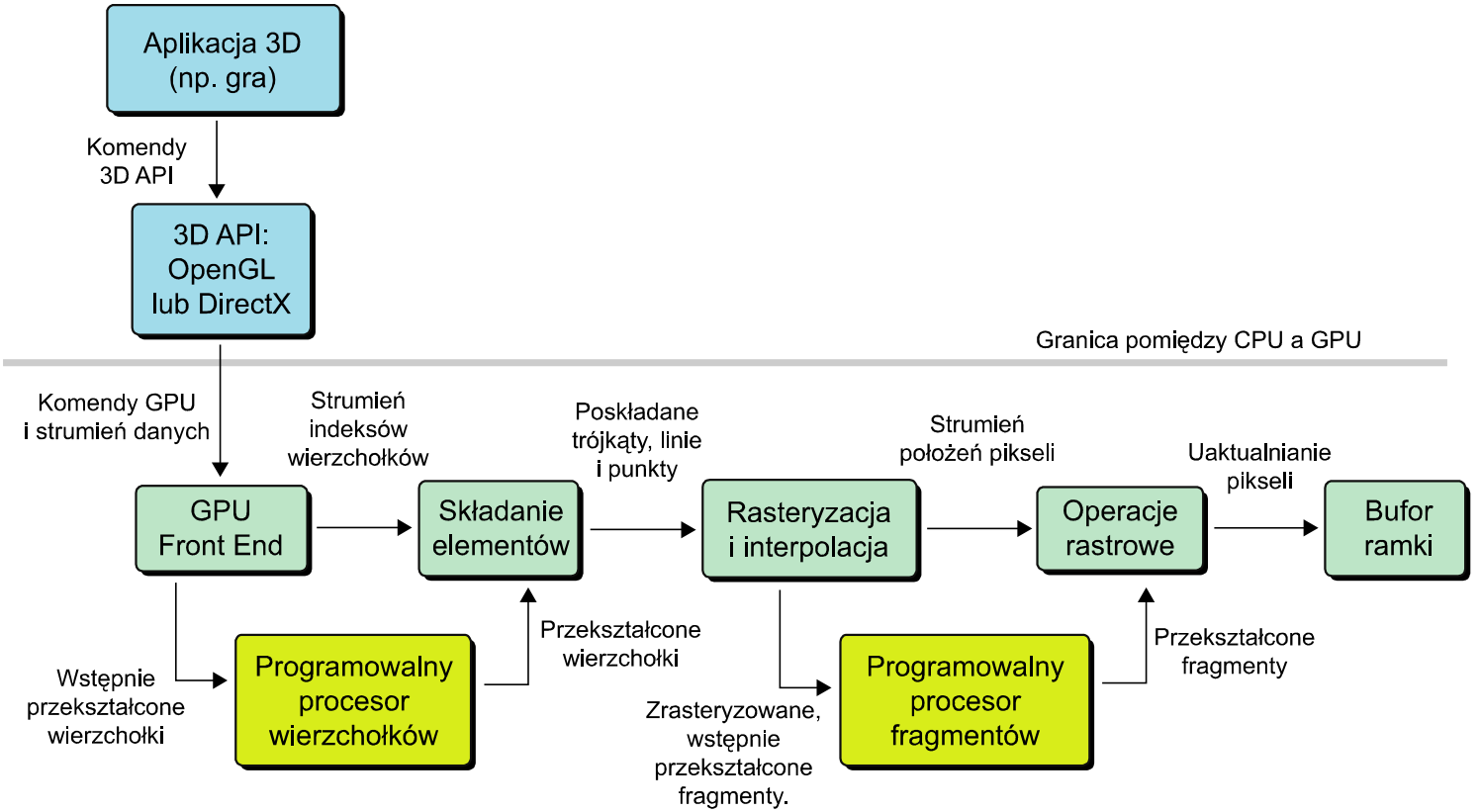
- Programowanie na niskim poziomie jest trudne i czasochłonne
- Kod jest nieczytelny

Shadery w języku wysokiego poziomu

- Rendering offline (produkcja filmów)
  - *RenderMan Shading Language* (Pixar)
    - Wzorcowy punkt odniesienia dla pozostałych rozwiązań
- Rendering w czasie rzeczywistym
  - *Stanford Shading Language* (Stanford University)
    - Od renderingu wieloprzebiegowego po wykorzystywanie sprzętowych shaderów
  - *SGI Interactive Shading Language* (SGI)
    - Nakładka, nie wykorzystuje sprzętowych shaderów
  - *PixelFlow Shading Language* (University of North Carolina + HP)
    - Dedykowany sprzęt
  - *C for Graphics* (NVIDIA + Microsoft)
    - Tylko dla sprzętowych shaderów
  - *OpenGL Shading Language* (3Dlabs)
    - **Będzie** tylko dla sprzętowych shaderów

C for Graphics - Cg

- NVIDIA + Microsoft
  - Składniowo *Cg* jest tym samym co *DirectX 9 HLSL*
- Wersja beta: czerwiec 2002, pierwsze oficjalne wydanie: grudzień 2002.
- Model programowania GPU:



Cg - c.d.

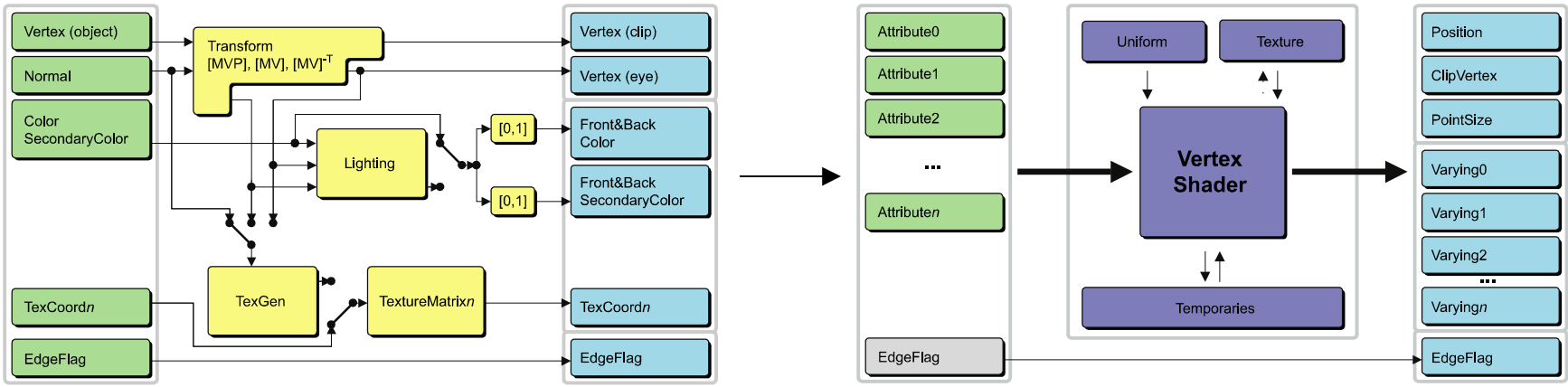
- Przeznaczony dla różnych rozwiązań niskopoziomowych

- Każdy z tzw. *profili* definiuje pewien podzbiór możliwości języka, np.:
  - Maksymalną ilość wygenerowanych instrukcji
  - Dostępne operatory, funkcje, pętle
  - Pewne zależności pomiędzy zmiennymi i funkcjami
    - Np. *GeForce3* może wykonać *dependent texture read* tylko w pewnych ściśle określonych warunkach
      - *„error C9999: Dependent texture operations don't meet restrictions of texture shaders"*

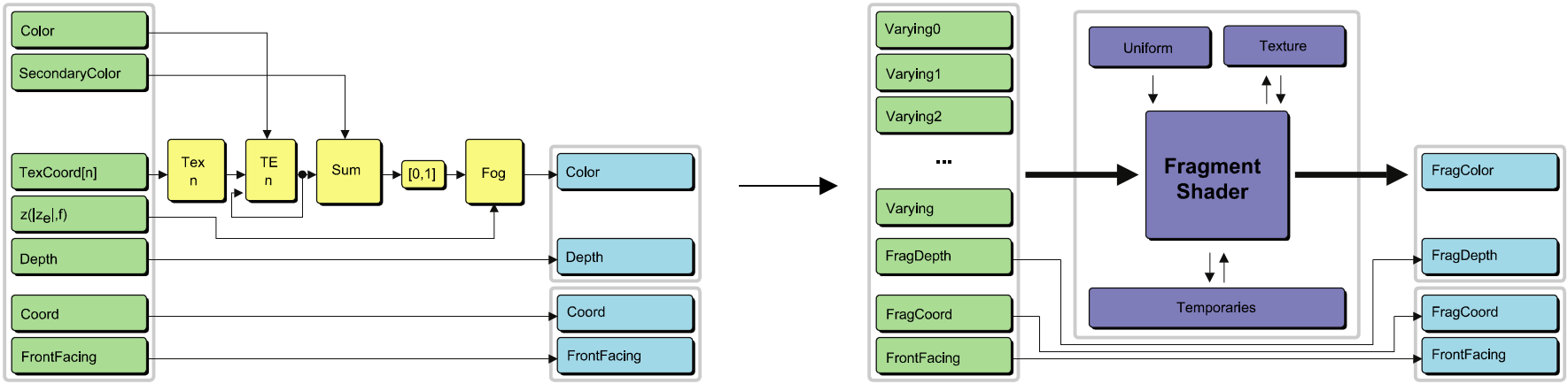
- Biblioteka Runtime umożliwia korzystanie z Cg w aplikacji
  - Kompilowanie programów w locie
  - Możliwość kompilowania dla różnych profili

OpenGL Shading Language - GLslang

- *OpenGL 1.4* (obecnie)
  - Nie nadąża za rozwojem sprzętu
  - Niewygodne w użyciu rozszerzenia
- *OpenGL 2.0*
  - Promowany i opracowywany głównie przez *3Dlabs*
  - Testowa implementacja tylko na kartach *Wildcat VP*
  - Projektowany pod następne generacje sprzętu
  - Wymiana pewnych stałych części potoku *OpenGL* na programowalne



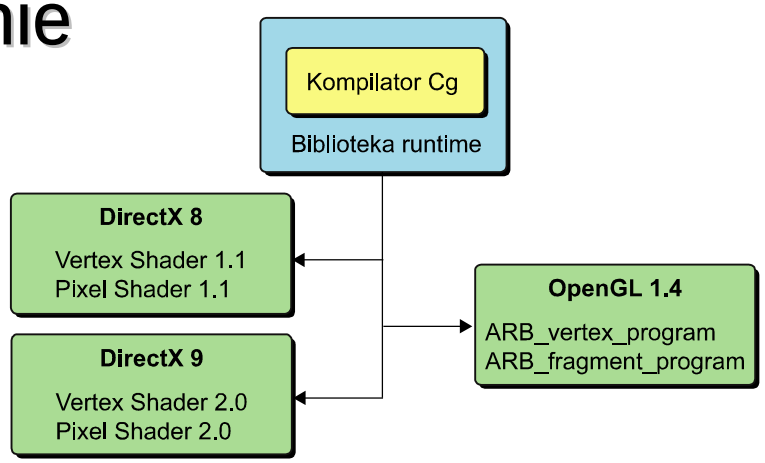
GLslang - c.d.



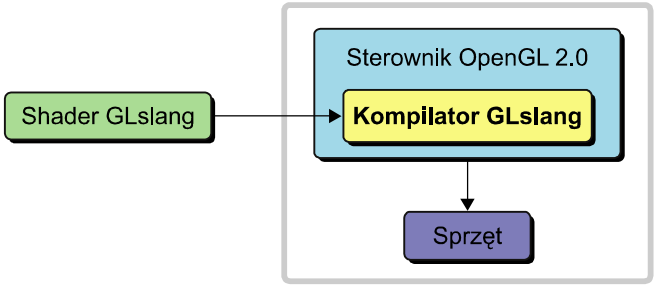
- *GLslang*
  - Ma ograniczyć powstawanie kolejnych rozszerzeń
  - Składnia oparta na C + dodatki specyficzne dla GPU
    - Typy wektorowe, macierzowe

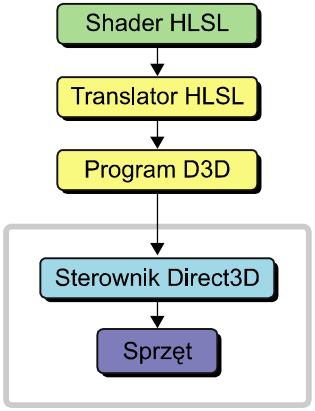
Porównanie

- *Cg:* język „ponad podziałami” (w teorii)
- *GLslang:* wbudowany (jak *HLSL* w *DirectX*)



- *Cg:* tworzy kod pośredni (jak *HLSL*)
- *GLslang:* kompilowany do postaci binarnej

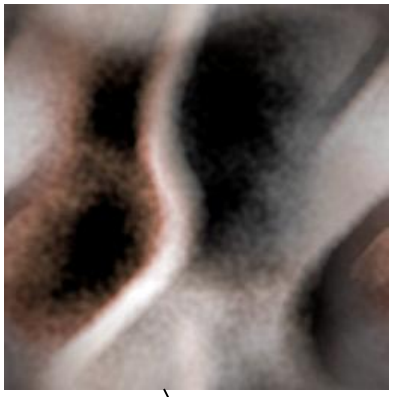




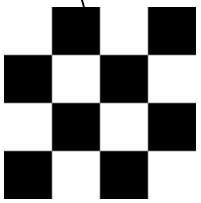
- Różnice składniowe
  - Cg: `float4x4 M;` `float4 x, y;` `y = mul(M, x);`
  - GLslang: `mat4 M;` `vec4 x, y;` `y = M*x;`
- Różnice w sposobie przekazywania parametrów
- Dostępność

Przykładowy program dla GPU

- Fragment shader



```
float3 texel0 = tex2D(texture1, IN.texCoord0).rgb;
float3 texel1 = tex2D(texture2, IN.texCoord1).rgb;
float3 color = lerp(texel0, texel1, tex2D(mask, IN.texCoord2).r);
return float4(IN.lighting.y*color + IN.lighting.z, 1.0);
```



Przykładowy program dla GPU

- Po skompilowaniu...

ps_1_1

```
def c0, 0.000000, 0.000000, 1.000000, 0.000000
def c1, 1.000000, 0.000000, 0.000000, 0.000000
def c2, 0.000000, 1.000000, 0.000000, 0.000000
tex t0
tex t1
tex t2
dp3 t2.rgb, c1, t2
dp3 t3.rgb, c2, v0
dp3 r1.rgb, c0, v0
mov r0.rgb, t2
lrp t0.rgb, r0, t0, t1
mad r0.rgb, t3, t0, r1
+ mov r0.a, c0.b
```

ps_2_0

```
    dcl_2d  s0
    dcl_2d  s1
    dcl_2d  s2
    def     c0, 0.000000, 0.000000, 0.000000, 1.000000
    dcl     v0.yz
    dcl     t0.xy
    dcl     t1.xy
    dcl     t2.xy
texld   r0, t1, s1
texld   r1, t0, s0
add     r0.xyz, r0, -r1
texld   r2, t2, s2
mad     r0.xyz, r2.x, r0, r1
mad     r0.xyz, v0.y, r0, v0.z
mov     r0.w, c0.w
mov     oC0, r0
```

arbfp1

```
PARAM c0 = {0, 0, 0, 1};
TEMP R0;
TEMP R1;
TEMP R2;
TEX R0.xyz, fragment.texcoord[1], texture[1],
    2D;
TEX R1.xyz, fragment.texcoord[0], texture[0],
    2D;
ADD R0.xyz, R0, -R1;
TEX R2.x, fragment.texcoord[2], texture[2], 2D;
MAD R0.xyz, R2.x, R0, R1;
MAD result.color.xyz, fragment.color.primary.y,
    R0, fragment.color.primary.z;
MOV result.color.w, c0.w;
```