Grafika komputerowa

ROZWIĄZANIA SPRZĘTOWE I PROGRAMISTYCZNE

Przyspieszanie sprzętowe

Przyspieszanie sprzętowe grafiki (*hardware accelerated graphics*)

- procesor główny (CPU) przesyła wywołanie funkcji graficznej do akceleratora;
- akcelerator – np. procesor GPU na karcie graficznej – wykonuje operacje bez udziału CPU;
- implementacja sprzętowa procedur graficznych pozwala zwiększyć szybkość tworzenia grafiki i odciąża CPU.

Podwójne buforowanie

Podwójne buforowanie (*double buffering*)

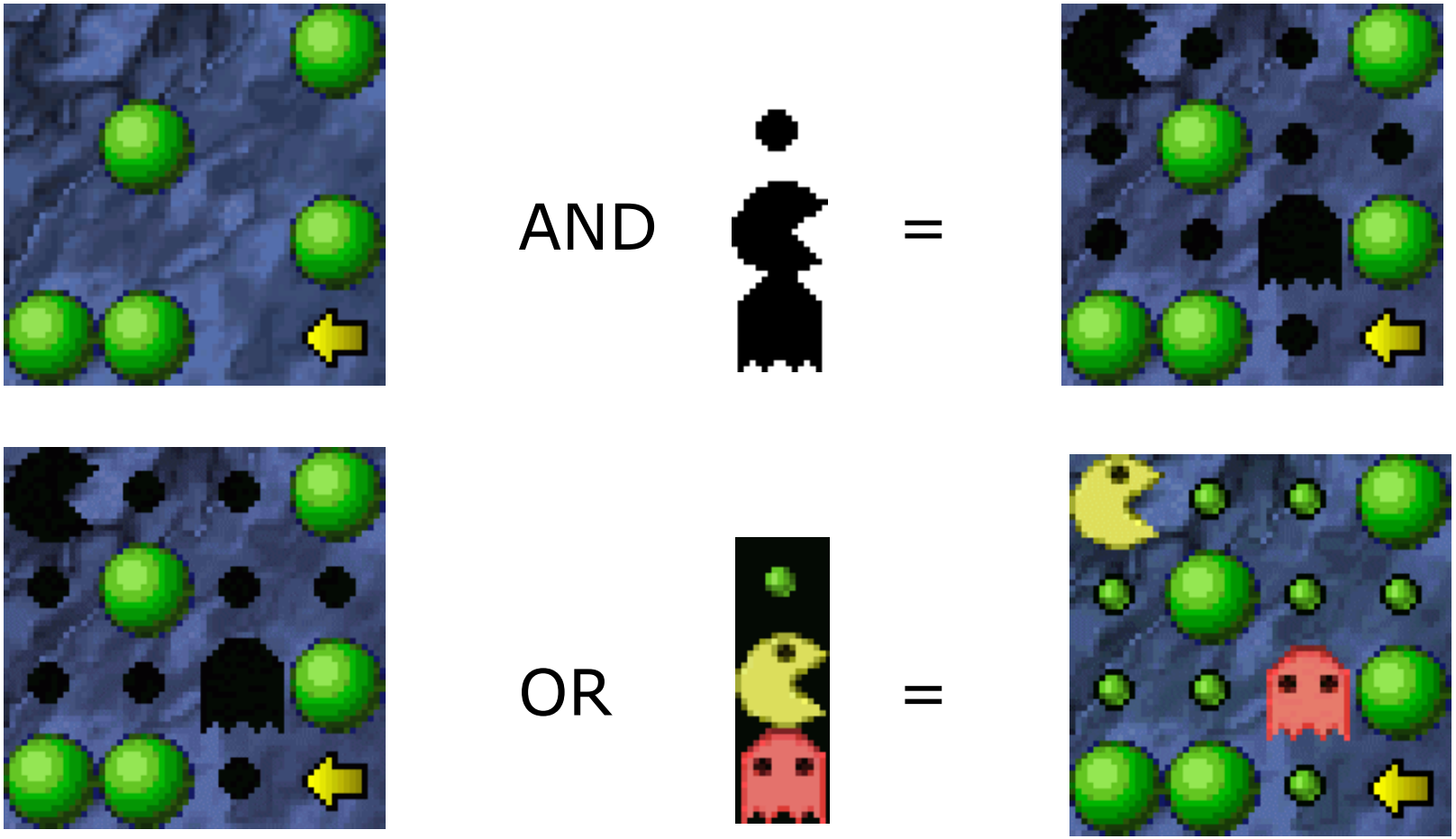
- Tworzenie grafiki „bezpośrednio na ekranie” powoduje zniekształcenia (np. migotanie) z powodu stałego odświeżania ekranu.
- Podwójne buforowanie:
  - dodatkowy obszar (bufor) pamięci,
  - tworzenie grafiki w tym buforze,
  - przeniesienie zawartości całego bufora na ekran w jednym kroku

Bit blit (BitBLT)

- *Bit blit* polega na skopiowaniu danych z bufora w pamięci do bufora obrazu.
- Jest to jedna z pierwszych operacji, które zaimplementowano sprzętowo w układach graficznych.
- Jest wykonywana bardzo szybko.
- Zapobiega zniekształceniom obrazu.
- Umożliwia stosowanie operacji logicznych AND, OR – maskowanie wybranych fragmentów obrazu.

Bit blit (BitBLT)

Ilustracja *blittingu* z zastosowaniem masek:



Przyspieszanie grafiki 2D

Przyspieszanie sprzętowe grafiki w kartach graficznych PC:

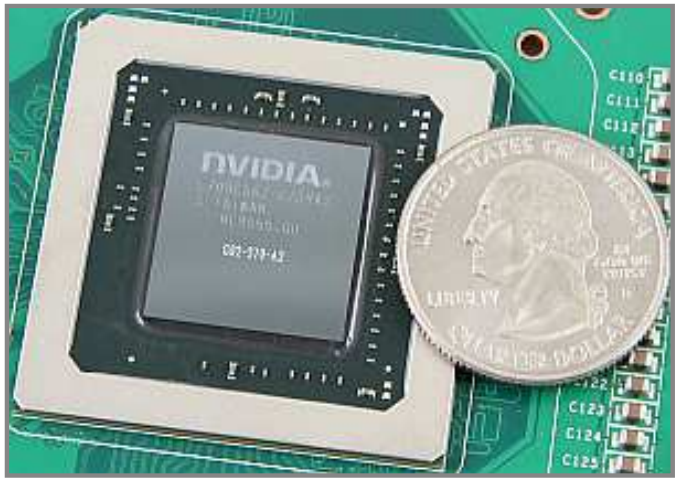
- wprowadzone w latach 90.
- sprzętowa implementacja rysowania prymitywów 2D, np. prostokątów
- przyspieszało rysowanie np. okienek w graficznym systemie operacyjnym (*windows accelerators*)
- współczesne GPU nadal mają zaimplementowane przyspieszenie sprzętowe procedur grafiki 2D.

Karty graficzne

- GPU – procesor graficzny (*Graphics Processing Unit*)
- Video BIOS (firmware)
- Pamięć RAM (GDDR)
  - bufor obrazu
  - bufor wierzchołków (*vertex buffer*)
  - bufor tekstur (*texture buffer*)
  - bufor głębokości (*z-buffer*)
  - bufor maski (*stencil buffer*)
  - pamięć programów (shaderów)

Procesor GPU

- Procesor GPU jest zoptymalizowany do równoległego wykonywania operacji, np. na wielu wierzchołkach lub wielu pikselach równocześnie.
- Architektura SIMD – „jedna instrukcja, wiele danych”.
- Większość operacji związanych z rasteryzacją jest wykonywana w pełni sprzętowo na GPU.



Rozwiązania sprzętowe

Rozwiązania sprzętowe dla grafiki 3D

- Dawne karty graficzne: brak akceleracji 3D, grafika tworzona przez CPU
- Akceleratory 3D – dodatkowe karty graficzne (*3dfx Voodoo* – 1996 r.)
- Współczesne karty graficzne – pełna akceleracja grafiki 3D. Główni producenci:
  - *NVidia* (karty *GeForce*),
  - *AMD* (karty *Radeon*)
  - *Intel*
- Układy zintegrowane – notebooki, smartfony

Oprogramowanie grafiki 3D

Warstwy logiczne oprogramowania:

- aplikacja (gra, silnik gry)
- API – interfejs programistyczny – procedury niezależne od sprzętu (*DirectX, OpenGL*)
- sterowniki karty graficznej – tłumaczenie poleceń API na instrukcje dla konkretnego modelu GPU
- oprogramowanie karty graficznej (*firmware, Video BIOS*) – sprzętowa realizacja poleceń dotyczących tworzenia grafiki, bezpośrednie sterowanie pracą GPU.

Programowanie grafiki - interfejsy

Obecnie programiści korzystają z gotowych interfejsów programistycznych (API):

- *DirectX* – Microsoft (system Windows)
- *OpenGL* – otwarty standard (Linux, iOS, Windows, OpenGL ES - systemy mobilne)

Interfejsy *DirectX* i *OpenGL* pozwalają programistom na uruchamianie procedur zaimplementowanych sprzętowo w układach GPU do tworzenia grafiki 3D. Umożliwiają pisanie oprogramowania w sposób niezależny od docelowej platformy sprzętowej.

Zgodność sprzętowa

Co oznacza, że „karta graficzna jest zgodna z DirectX w wersji #”?

- w wersji DirectX # dodano obsługę określonej funkcji,
- BIOS karty graficznej posiada zaimplementowane procedury pozwalające uzyskać taki efekt,
- sterownik karty graficznej potrafi uruchomić wymaganą procedurę, po otrzymaniu określonego polecenia od programu

Podstawowe funkcje API 3D

Najważniejsze operacje, które zostały zaimplementowane sprzętowo na GPU:

- bufor pamięci wierzchołków
- przetwarzanie siatek wielokątowych,
- pamięć tekstur, próbkowanie tekstury
- filtrowanie tekstur, mipmapping
- bufor głębokości (*z buffer*)
- bufor maski (*stencil buffer*)
- modele cieniowania płaskiego i Gourouda
- obcinanie trójkątów (*clipping*)

TLC - Transform, Lighting and Clipping

Transform, Lighting & Clipping (TLC)
– zbiór operacji dotyczących:

- przekształcania modelu 3D (*transform*)
  - przekształcenia siatki,
  - rzutowanie
- oświetlenia sceny
  - nakładanie, próbkowanie, filtracja tekstur
  - cieniowanie (model ADS)
- obcinania widoku:
  - przycinanie trójkątów
  - usuwanie niewidocznych powierzchni

TLC - Transform, Lighting and Clipping

- Przed wersją *DirectX 7.0*, operacje TLC musiały być wykonywane przez CPU.
- Wersja 7.0 (1999 r.) wprowadziła sprzętowe przyspieszenie operacji TLC.
- Przyniosło to prawdziwą rewolucję w grafice komputerowej 3D, ponieważ od tej pory najważniejsze operacje tworzenia grafiki mogły być wykonywane na GPU, znacznie odciążając CPU i przyspieszając renderowanie obrazu.

Ustalony potok renderingu

Fixed function pipeline (ustalony potok r.)

- Potokiem nazywamy kolejno wykonywane operacje, tutaj dotyczące renderingu
- W ustalonym potoku renderingu, kolejność operacji jest z góry zdefiniowana.
- Programista musi podać tylko parametry operacji, np. przekształcenia siatki albo parametry źródeł światła.
- Nie ma możliwości wpływania na to, w jaki sposób te procedury są wykonywane.
- Dostępne w DirectX do wersji 9 włącznie.

Ustalony potok renderingu

- Zaletą jest prostota programowania. Jednak podejście to znacznie ogranicza możliwości tworzenia grafiki.
- Transformacje siatek wielokątowych – można łatwo przekształcić cały model, ale nie można modyfikować poszczególnych werteksów.
- Cieniowanie – metoda jest zaszyta w API (np. Gourouda), nie można zmodyfikować oświetlenia zgodnie z własnymi wymaganiami.

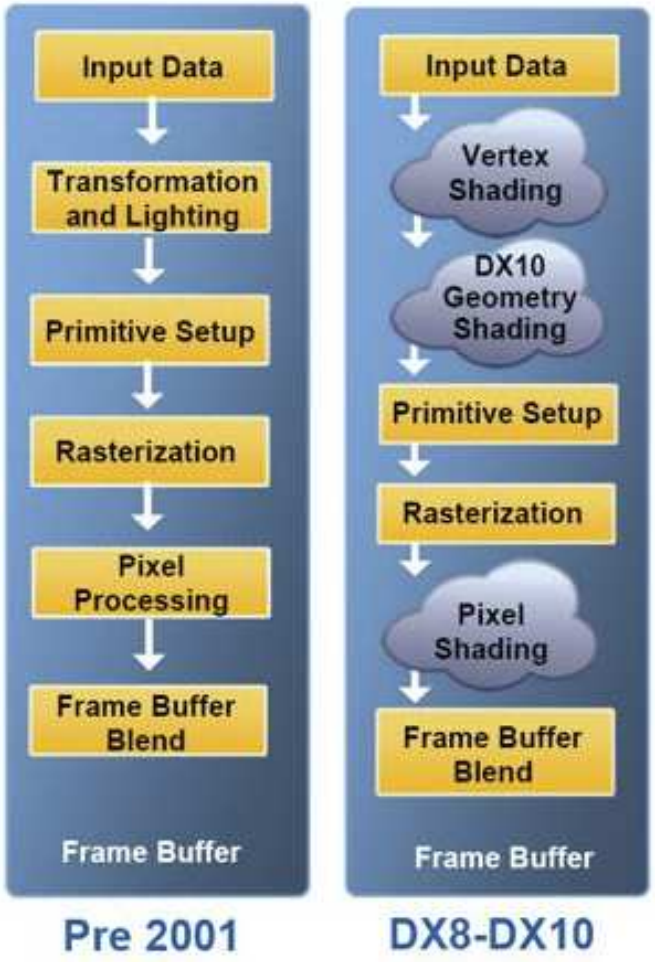
Programowalny potok renderingu

Programmable pipeline (programowalny potok)

- Programista musi sam zaimplementować istotne etapy renderingu.
- Ma on jednak pełny wpływ na ich przebieg, za pomocą programów nazywanych *shaderami*
- Trudniejsze programowanie, znacznie większe możliwości.
- Wprowadzone częściowo w DirectX 8 (2000 r.), znacznie rozwinięte w DirectX 10, gdzie jest jedyną możliwością programowania grafiki.

Programowalny potok renderingu

- Programowalny potok „zwraca” kontrolę nad przebiegiem rasteryzacji do programisty, w miejscach oznaczonych „chmurką”.
- Zadaniem programisty jest wypełnienie tych miejsc za pomocą *shaderów*.
- Shader jest to program uruchamiany na GPU.



Vertex shader

- *Vertex shader* (shader wierzchołkowy)
  – program, który jest wywoływany jeden raz dla każdego wierzchołka siatki trójkątowej danego modelu obiektu.
- Podstawowym zadaniem jest konwersja współrzędnych wierzchołka w ukł. wsp. modelu na współrzędne widoku 2D (po rzutowaniu).
- Istnieje pełna dowolność – możemy zaimplementować tylko operacje afiniczne, ale możemy też każdy werteks przekształcić w inny sposób.

Vertex shader

Podstawowy sposób wykorzystania VS:

- programista podaje dane wejściowe:
  - wszystkie dane wierzchołka,
  - macierze przekształceń
- wewnątrz VS, programista implementuje (za pomocą podanych macierzy):
  - przekształcenia afiniczne modelu,
  - ew. przekształcenia poj. werteksów
  - konwersję do ukł. świata,
  - rzutowanie,
  - obliczenia światła wierzchołkowego.

Vertex shader

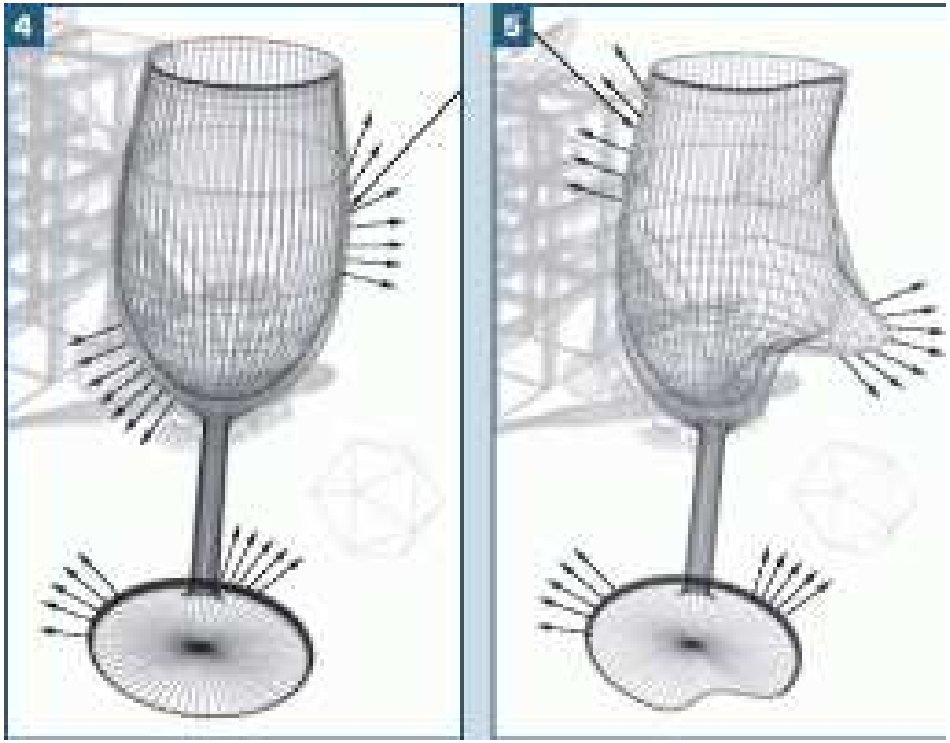
Wartości zwracane z VS do potoku renderingu:

- VS MUSI (!) zwrócić współrzędne werteksu po rzutowaniu
- programista może zwrócić też inne wartości – zostaną one automatycznie interpolowane przez rasteryzer, np.:
  - współrzędne tekstury,
  - współrzędne wektora normalnego,
  - barwa wierzchołka

Vertex shader

Przykładowe efekty, jakie można uzyskać za pomocą przekształcania werteksów w VS:

- zniekształcenia obiektów (np. uszkodzenia),
- ruchy powierzchni wody (fale),
- mimika twarzy, itp.



Fragment (pixel) shader

- *Fragment shader* (OpenGL)*, Pixel shader* (DirectX) – shader fragmentowy, program uruchamiany jeden raz dla każdego fragmentu (piksela) docelowego obrazu.
- Jego zadaniem jest ustalenie barwy każdego z fragmentów/pikseli obrazu.
- Barwa ta może być:
  - obliczona w FS (światło fragmentowe, np. metoda Phonga),
  - pobrana z interpolowanych wyników VS (światło wierzchołkowe, np. metoda Gourouda)

Fragment (pixel) shader

Zastosowanie jednostek *fragment shader* pozwala uzyskać efekty związane z oświetleniem, takie jak np.:

- chropowatość powierzchni (*bump mapping*),
- odbicia lustrzane,
- cienie,
- systemy cząsteczkowe,
- tonowanie barwy (HDR)

Shadery a cieniowanie

- Należy pamiętać, że funkcje dotyczące cieniowania nie są sztywno powiązane z konkretnym typem shadera.
- Używamy danego shadera w zależności od tego, w których miejscach obliczamy oświetleniu obiektu.
- Jeżeli obliczamy oświetlenie w wierzchołkach, użyjemy do tego VS, a FS pobierze barwę zinterpolowaną przez rasteryzer.
- Jeżeli obliczamy we fragmentach, FS obliczy oświetlenie na podstawie wektorów normalnych, zinterpolowanych przez rasteryz.

Geometry shader

- *Geometry shader* (shader geometrii) wprowadzono w DirectX 10 (2007 r), jednocześnie usunięto ustalony potok.
- VS pozwala modyfikować każdy werteks, ale trudno modyfikować grupy werteksów, np. cały trójkąt.
- Do tego celu wprowadzono GS.
- W przeciwieństwie do VS i FS, ten typ shadera jest opcjonalny (można go pominąć).
- GS ma zwrócić na wyjściu dowolną liczbę werteksów.

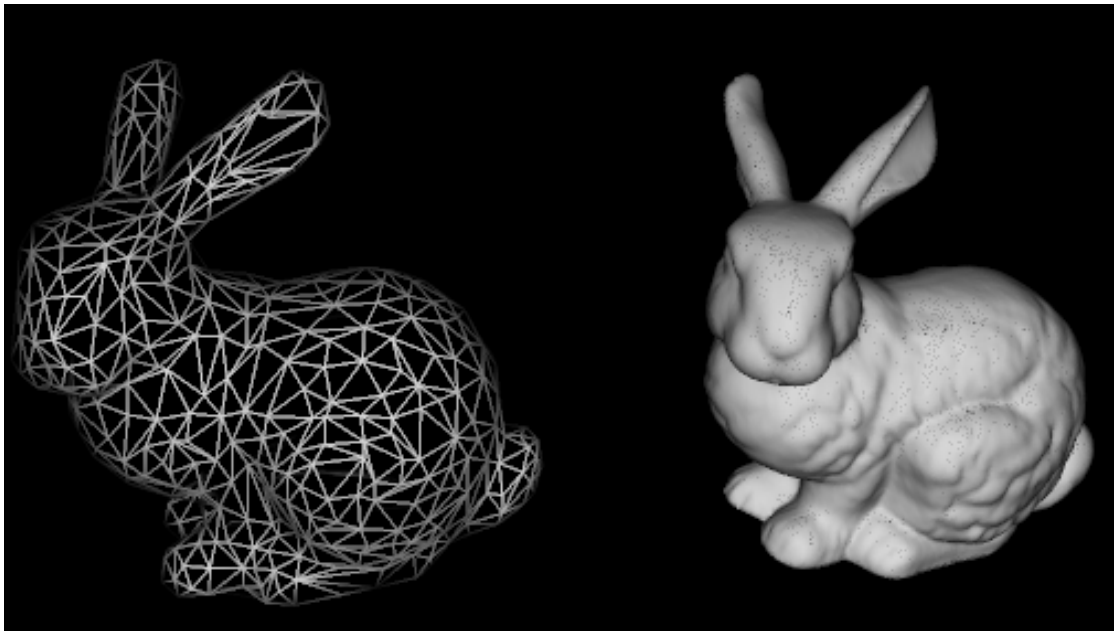
Geometry shader

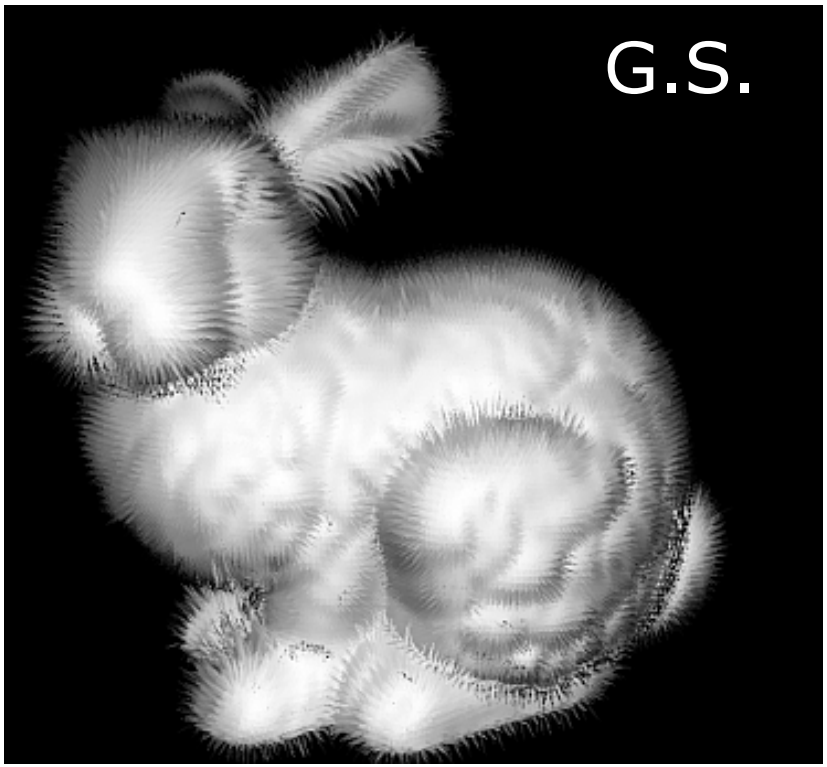
Możliwości GS:

- operacje na grupach werteksów (typowo: na całych trójkątach siatki)
- dodawanie nowych werteksów (zagęszczanie siatki)
- usuwanie werteksów (np. dodanie otworów)
- zmiana rozdzielczości siatki (*remeshing*)
- wykorzystanie np. do tworzenia cieni (*shadow volume*) i mapowania tekstur w technice *cube mapping*.

Geometry shader

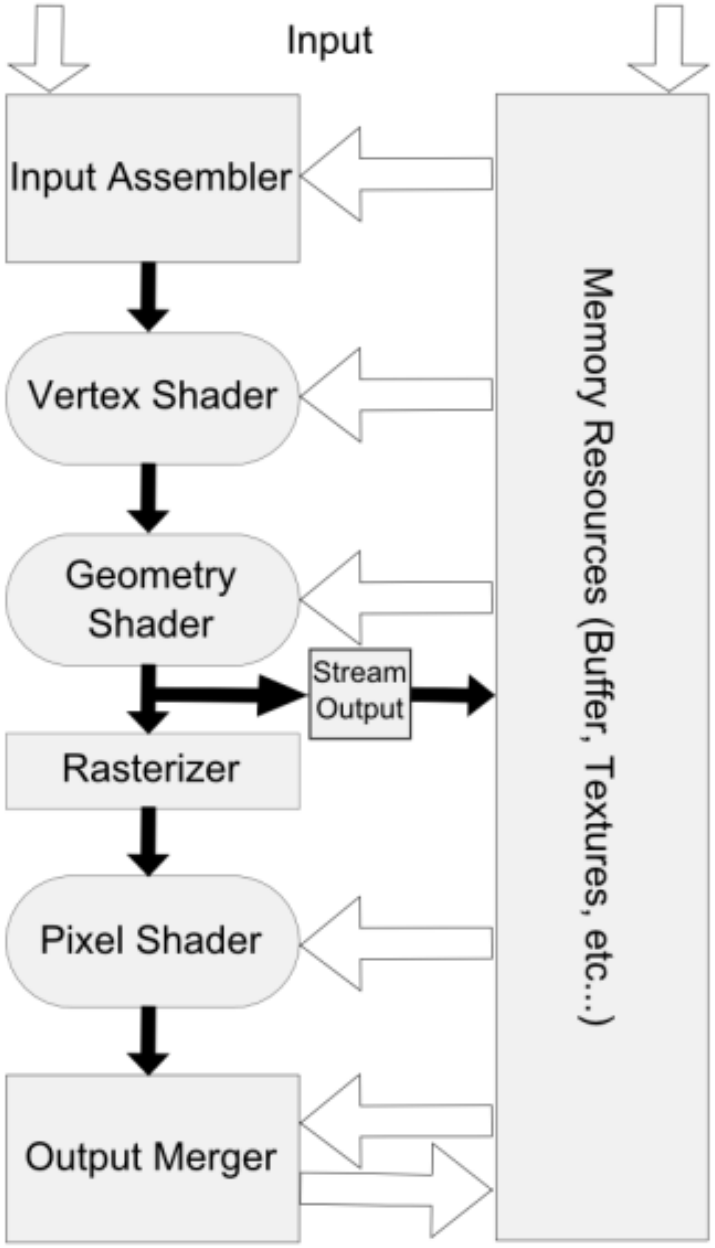
Przykład wykorzystania – efekt futra





Potok renderingu w Direct3D

- *Input Assembler* - dostarcza dane
- *Vertex shader* - transformacje wierzchołków siatki (werteksów)
- *Geometry shader* - przetwarzanie prymitywów (grup werteksów)
- *Stream output* - zachowanie stanów pośrednich w pamięci
- *Rasterizer* – konwersja werteksów na piksele, przycinanie, interpolacja
- *Pixel shader* - operacje na pikselach obrazu 2D (modyfikacje barwy)
- *Output merger* - łączy wyniki działania różnych procedur w obraz końcowy



Unified Shader Architecture

- W pierwszych wersjach API trzeba było programować oddzielnie każdy z typów shadera, nieraz przy użyciu różnych języków.
- *Unified Shader Architecture* – wprowadzona w *DirectX* 10 zintegrowana jednostka „3 w 1”: VS + FS + GS
- Pozwala programować wszystkie shadery w jednakowy sposób.
- OpenGL stosuje termin *Shader Model*.

Programowanie shaderów

- Do programowania shaderów stosuje się specjalne języki, będące rozszerzeniem języka C.
- DirectX – język HLSL (*High Level Shading Language*).
- OpenGL – język GLSL (*OpenGL Shading Language*).
- Różna składnia, podobna filozofia.
- Można też stosować asembler.

DirectX

- DirectX jest systememem do tworzenia oprogramowania pod system Windows.
- Główny komponent (dawniej: *Direct3D*) dotyczy tworzenia grafiki 3D.
- Zawiera też procedury obsługi okien, myszy i klawiatury, sterowników gier, itp.
- Moduł *DirectSound* do obsługi dźwięku.
- Inne komponenty nie są już praktycznie używane (obsługa grafiki 2D, wideo, MIDI).
- Języki: C++, C#.

OpenGL

- *OpenGL* jest wieloplatformową specyfikacją procedur graficznych dla różnych systemów operacyjnych.
- *S*pecyfikacja zawiera zbiór definicji funkcji i opis ich działania.
- Producenci sprzętu muszą zadbać o implementację tych funkcji w sterownikach sprzętu.
- Producenci mogą też dodawać własne funkcje w formie rozszerzeń (*extensions*).

OpenGL

- W przeciwieństwie do DirectX, OpenGL jest bardziej niskopoziomowe, zawiera jedynie procedury dotyczące bezpośrednio grafiki.
- Stosuje się pomocnicze biblioteki, m.in:
  - GLUT, GLFW – obsługa okien programu i urządzeń wejściowych (mysz, klaw.)
  - GLmath – operacje matematyczne, np. na macierzach przekształceń
  - GLEW – obsługa rozszerzeń OpenGL
  - OpenAL – podsystem do dźwięku

Biblioteki - nakładki

Biblioteki stanowiące nakładki na graficzne API, ułatwiające pisanie wieloplatformowych programów z użyciem grafiki 3D:

- SDL – *Simple DirectMedia Layer*
- OGRE – *Object-Oriented Graphics Rendering Engine*
- Horde3D
- Irrlicht
- Allegro

Obliczenia równoległe na GPU (GPGPU)

Moc GPU można wykorzystać nie tylko do grafiki, ale również do przyspieszania obliczeń równoległych nie związanych z grafiką:

- *CUDA* (NVidia)
- *OpenCL* (otwarta specyfikacja, wieloplatformowa)
- *DirectCompute* (Microsoft, DirectX)
- *C++Amp* (Microsoft, tylko Windows)

LuxRender – przykład programu do renderingu oświetlenia globalnego, wykorzystującego OpenCL do obliczeń na GPU.

Silniki fizyczne

Silnik fizyczny (*physics engine*) służy do realistycznego odwzorowania praw fizyki w animacjach komputerowych, np. w grach.

- Modele fizyczne w grach są z konieczności uproszczone (praca w czasie rzeczywistym).
- Silnik fizyczny „zwalnia” programistę od samodzielnego implementowania praw fizyki w komputerowym świecie.

Popularny silnik fizyczny: *PhysX* firmy NVidia. Wykorzystuje GPU do obliczeń.

PhysX

Najważniejsze zjawiska realizowane przez silnik:

- dynamika ciał sztywnych (newtonowska),
- dynamika ciał plastycznych (odkształcenia)
- detekcja kolizji i reakcja na nie
- ruch postaci (*character controller*), *ragdoll*
- ruch pojazdów (*vehicle dynamics*)
- systemy cząsteczkowe
- symulacja cieczy (*fluid simulation*)
- symulacja materiału (*cloth simulation*),
  w tym rozdarcia i układanie się
  na powierzchniach obiektów

PhysX APEX

APEX – system zbudowany na bazie PhysX, umożliwia tworzenie łańcuchów operacji związanych z modelami fizycznymi.

Wysokopoziomowy interfejs do PhysX.

- *APEX Clothing* – symulacja materiału
- *APEX Destruction* – zniszczenia obiektów
- *APEX Particles* – systemy cząsteczkowe
- *APEX Turbulence* – symulacja płynów, dymu i podobnych efektów
- *APEX ForceField* – symulacja ruchu pod wpływem wymuszającej siły

Inne silniki fizyczne

- *NVidia FleX* – nowy silnik systemów cząsteczkowych, traktuje wszystkie obiekty jako zbiory cząsteczek.
- *Havok* – konkurencyjny system Intela.
- *Bullet* (open source) - dynamika brył sztywnych i elastycznych, detekcja kolizji.
- *Advanced Simulation Library* (ASL)
  - open source, implementuje wiele złożonych procesów fizycznych, głównie do zaawansowanych symulacji.

Silnik gry (Game engine)

Game engine („silnik gry”) – warstwa oprogramowania ułatwiająca tworzenie gier poprzez wykorzystanie gotowych procedur:

- rendering obrazu 3D, shadery
- modele animacji (np. szkieletowej),
- modele fizyczne,
- skrypty do sterowania obiektami,
- modele sztucznej inteligencji,
- obsługę dźwięku i muzyki,
- interfejs użytkownika

Unity

Unity – obecnie najbardziej popularny silnik gier. Uproszczona wersja jest dostępna za darmo.

- Edytor scen, animacji i skryptów.
- Model renderingu.
- Kinematyka odwrotna w animacji.
- Modele fizyczne – wykorzystują PhysX.
- Animacje 2D – *sprites*, model fizyczny 2D.
- Animacje 3D, systemy cząsteczkowe, symulacja materiału.
- Zaawansowany model ruchu pojazdów.
- Zaawansowana detekcja kolizji.

Unity

Zaawansowane efekty renderingu obsługiwane programowo przez *Unity*:

- mapowanie nierówności (*bump mapping*)
- mapowanie odbić (*reflection m.*)
- mapowanie paralaksy (*paralax m.*)
- okluzja otoczenia (*ambient occlusion*)
- dynamiczne cienie (metoda mapy cienia)
- renderowanie do tekstury (np. mapowanie środowiska)
- pełnoekranowe efekty post-processingu

Inne silniki gier

- *Unreal Engine* (komercyjny)
- *Blender Game Engine*, *Irrlicht* (darmowe)
- Silniki pisane dla konkretnych gier, obecnie wypierane przez *Unity*, np.:
  - *Chrome Engine* polskiej firmy *Techland*,
  - RAGE (Rockstar, m.in. *Grand Theft Auto*)
  - CryEngine (*Crytek*, m.in. *Far Cry*)
  - Luminous Studio (*Square Enix*, m.in. *Final Fantasy*)