Programowanie Grafiki Komputerowej

OpenGL Shading Language cz. 1 - wprowadzenie do shaderów

Grzegorz Pawiński

Katedra Informatyki, Politechnika Świętokrzyska w Kielcach



KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI



Politechnika Świętokrzyska

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY



Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

" Program Rozwojowy Potencjału Dydaktycznego Politechniki Świętokrzyskiej w Kielcach: *kształcenie w nowoczesnych obszarach techniki* "

Program Operacyjny Kapitał Ludzki, Priorytet IV Szkolnictwo wyższe i nauka, Działanie 4.1. Wzmocnienie i rozwój potencjału dydaktycznego uczelni oraz zwiększenie liczby absolwentów kierunków o kluczowym znaczeniu dla gospodarki opartej na wiedzy, Poddziałanie 4.1.1. Wzmocnienie potencjału dydaktycznego uczelni.

Umowa UDA–POKL.04.01.01-00395/09-00

Spis treści

1. Potok renderowania
   - Ustalony potok renderowania
   - Programowy potok renderowania
2. Shadery
   - Vertex shader
   - Pixel shader
   - Shadery w potoku renderowania
3. Programowanie shaderów
   - Języki programowania shaderów
   - Tworzenie shaderów
   - Błędy kompilacji i linkowania
4. Bibliografia

Uwagi

- Animacje są uruchamine, gdy plik PDF jest wyświetlany na pełnym ekranie.
- Animacja jest uruchamiana dopiero po naciśnięciu spacji (program Adobe Reader).
- Klatki animacji wyświetlane są co 2 sekundy.
- Symulacje są uruchamiane po kliknięciu w ikonę z odpowiednim systemem operacyjnym. Przeglądarka wyświetli okno z ostrzeżeniem, należy go zaakceptować. W systemie Linux należy dodatkowo kliknąć dwukrotnie w nazwę programu.

Wymagania do uruchomienia symulacji

- Do uruchamiania symulacji z pliku PDF, wymagany jest program Adobe Reader w wersji 5.0 lub nowszej.
- W systemie Windows, należy umieścić bibliotekę *glut32.dll* w katalogu systemowym (zwykle *C:\Windows*), a w jego podkatalogu *System32* bibliotekę *glew32.dll*. Biblioteki można znaleźć TUTAJ.
- W systemie Linux, biblioteki należy umieścić w katalogu */usr/lib*. Biblioteki można znaleźć TUTAJ.
- Symulacje programów zostały napisane dla sprzętu komputerowego obsługującego OpenGL w wersji 2.0 lub nowszej.

Potok renderowania

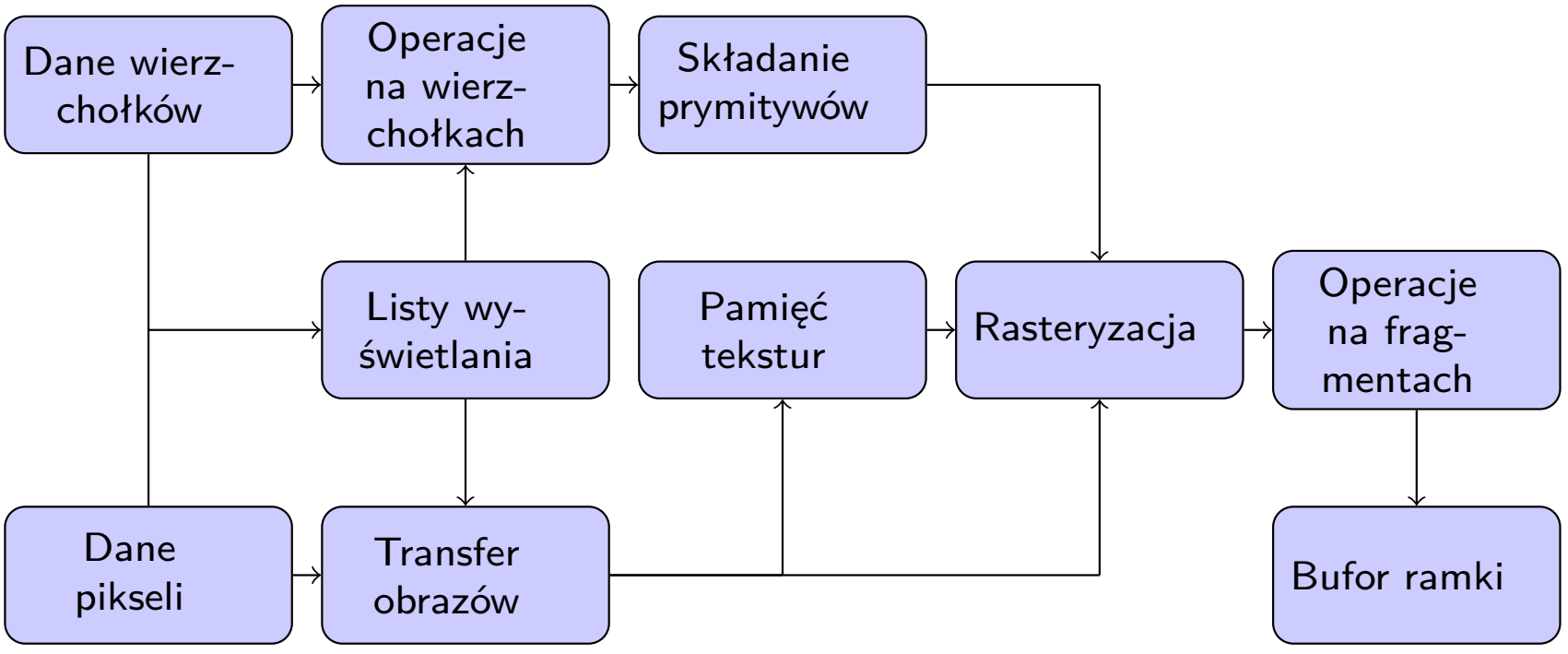
Potok renderowania OpenGL (ang. *OpenGL pipeline*) jest to zestaw operacji wykonywanych na danych wejściowych zanim zostaną wyrenderowane na ekranie. Etapem końcowym potoku jest wygenerowanie ***bufora ramki***. Dane wejściowe mogą być dwojakiego rodzaju:

- ***dane wierzchołków*** (ang. *vertex data*) - dotyczą obiektów i prymitywów w przestrzeni 3D.
- ***dane pikseli*** (ang. *pixel data*) - dotyczą obiektów tekstur i bitmap.

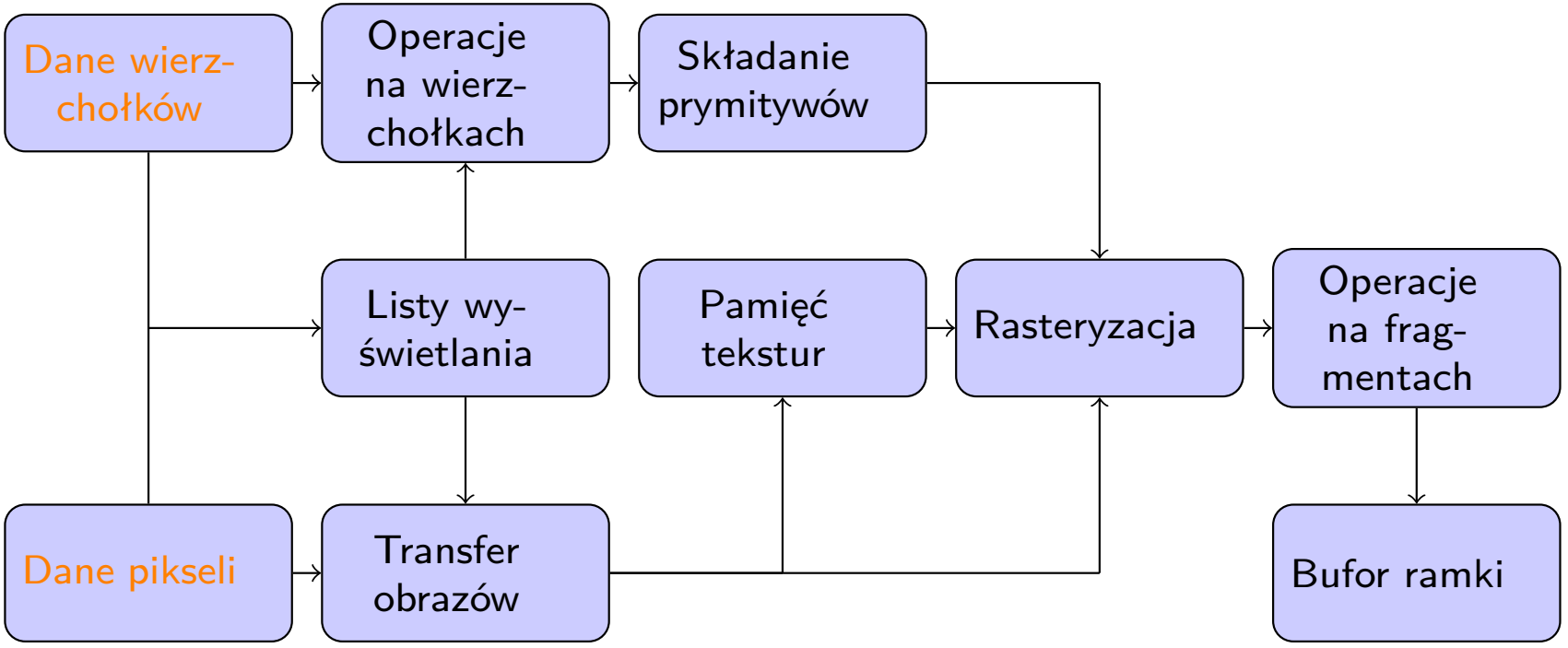
Ustalony potok renderowania

W ***ustalonym potoku renderowania*** (ang. *fixed pipeline*) programista nie ma wpływu na operacje wykonywane w potoku, może tylko przekazywać do nich parametry. Transofmracje modeli 3D są opisywane macierzami, a oświetlenie jest obliczane na podstawie danych zapisanych w modelu. Trudno jest przekształcać pojedyncze wierzchołki i uzyskać efekty specjalne. Ustalony potok renderowania pozwala na łatwe programowanie, ale ogranicza możliwości programisty.

Schemat ustalonego potoku renderowania



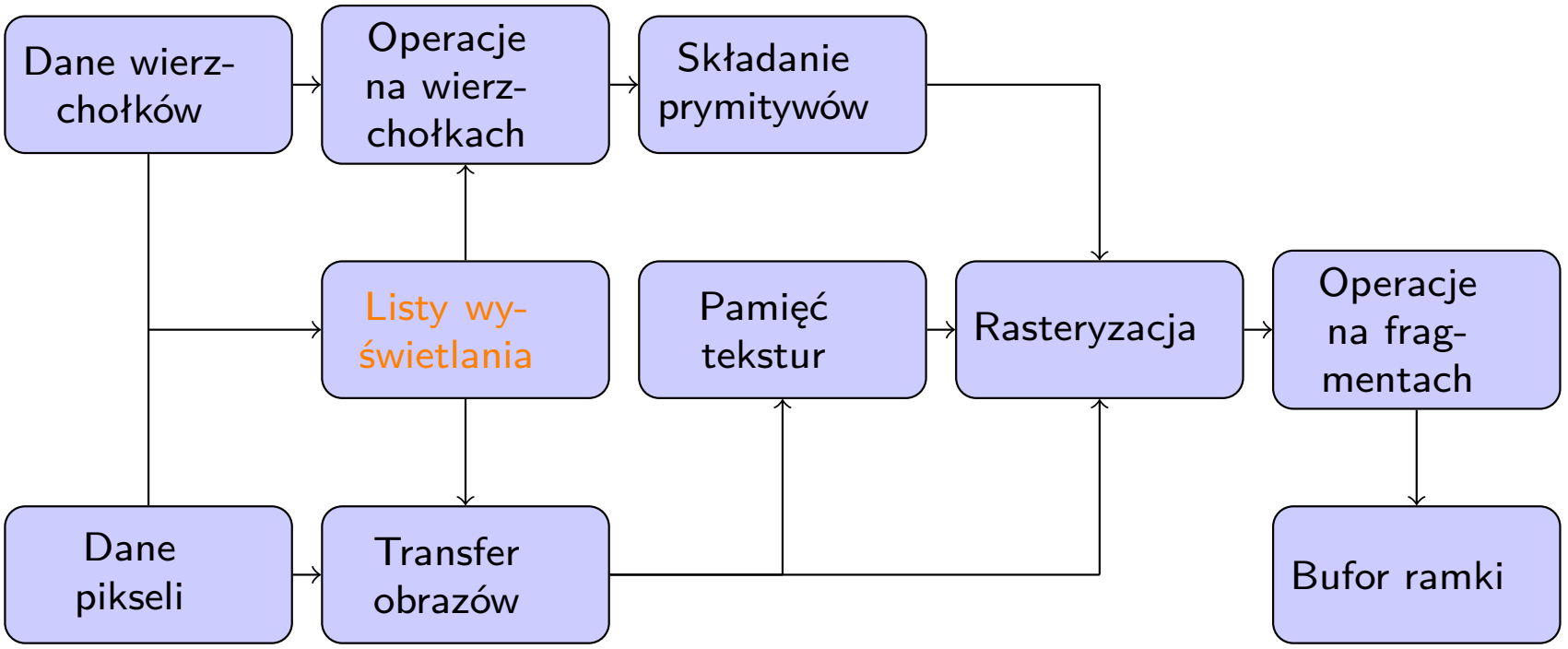
Schemat ustalonego potoku renderowania



Opis bloku

Na tym etapie ***dane wierzchołków*** i ***dane pikseli*** wczytywane są z dysku. Wierzchołki są zbiorem atrybutów, tj. pozycja w przestrzeni, składowe koloru, współrzędne tekstur itp.

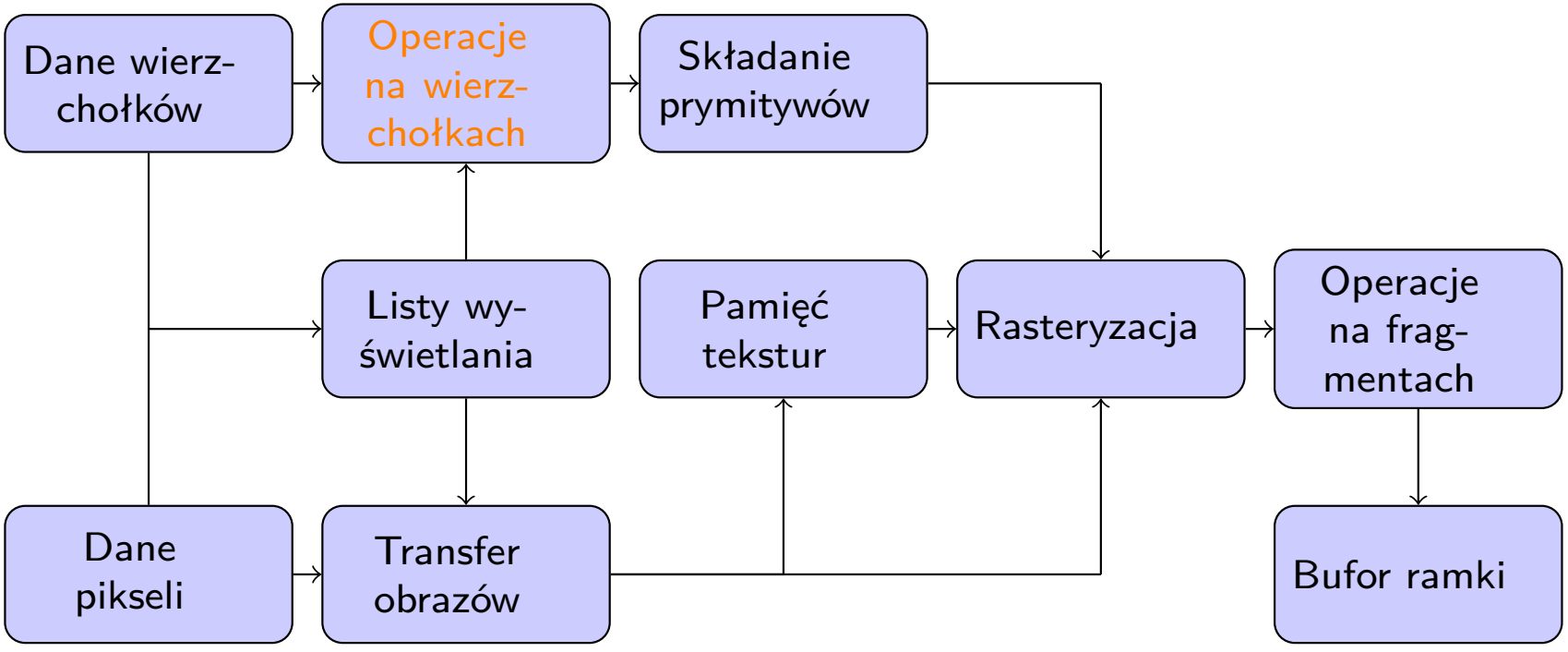
Schemat ustalonego potoku renderowania



Opis bloku

Na podstawie danych wejściowych tworzone są ***listy wyświetlania*** (ang. *display lists*), które przechowują skompilowane instrukcje OpenGL. Instrukcje są wykonane później, co znacznie przyspiesza renderowanie grafiki.

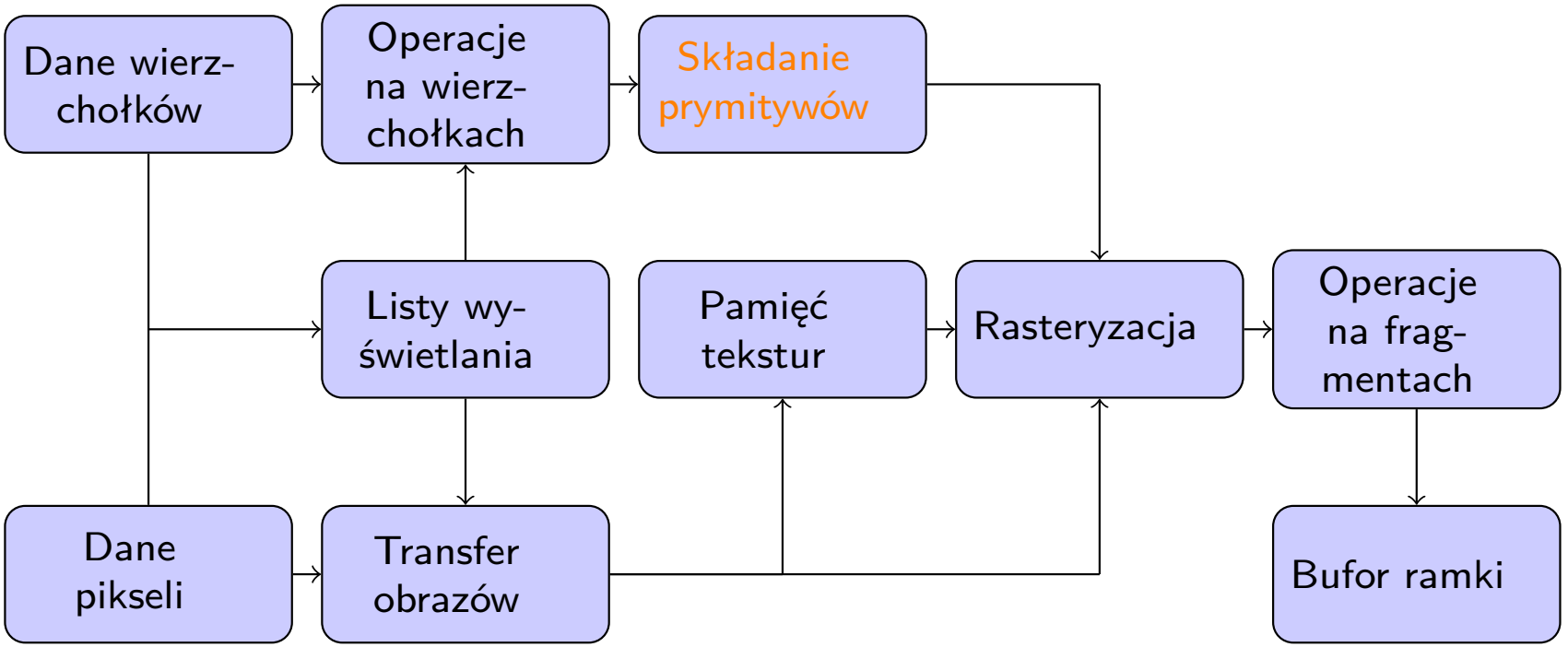
Schemat ustalonego potoku renderowania



Opis bloku

Operacje na wierzchołkach obejmują transformacje położenia wierzchołków i normalnych, obliczenie oświetlenia i wyznaczenie kolorów dla każdego wierzchołka.

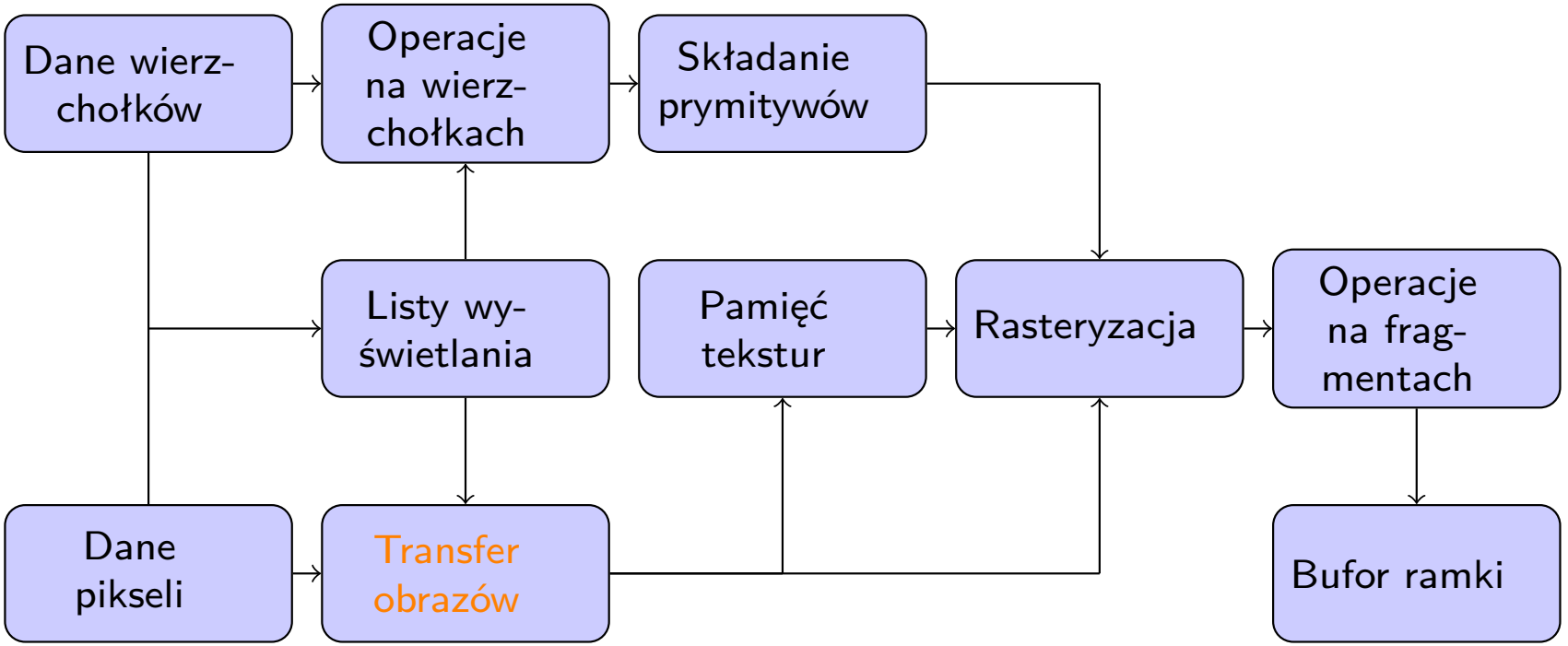
Schemat ustalonego potoku renderowania



Opis bloku

W tym bloku wierzchołki są ze sobą łączone tworząc prymitywy (punkty, linie i wielokąty). Prymitywy są transformowane przez macierz rzutowania i obcinane do ***bryły widzenia***. Następuje rzutowanie perspektywiczne i usuwanie powierzchni niewidocznych.

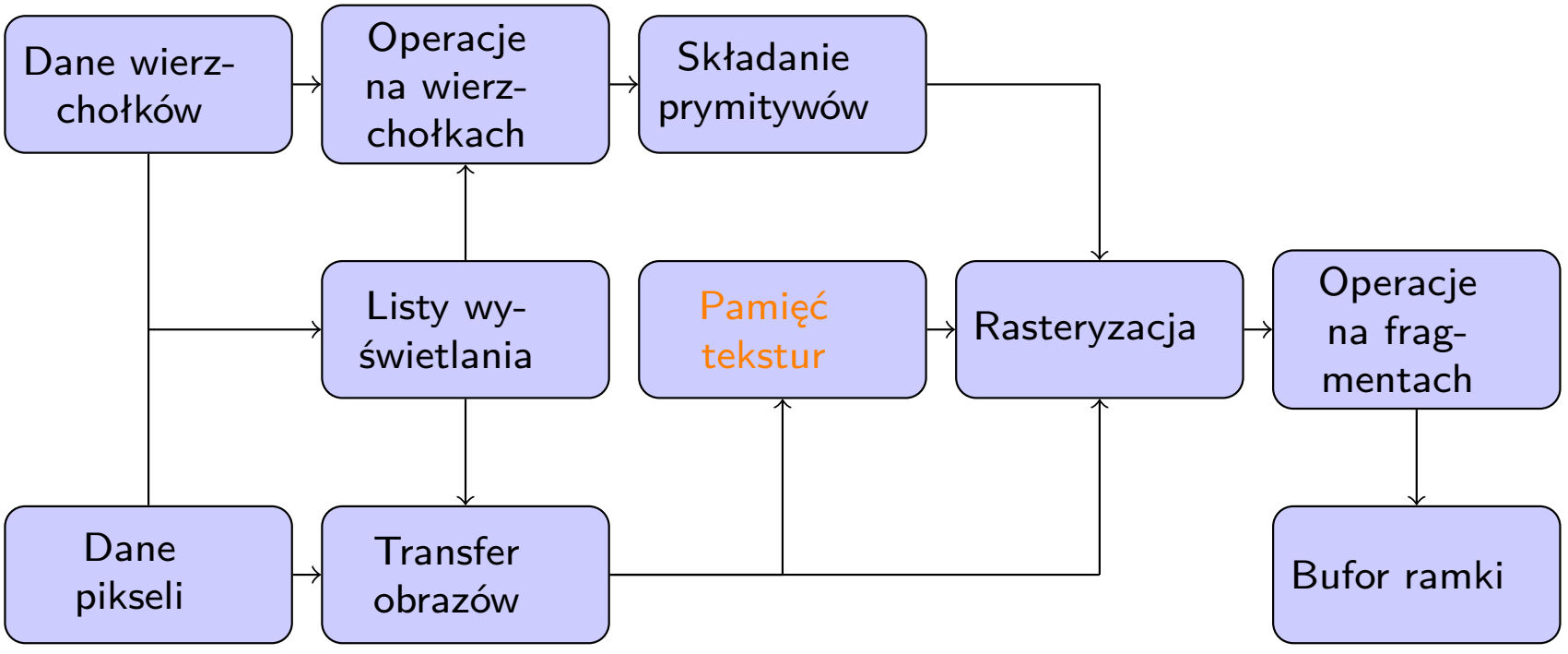
Schemat ustalonego potoku renderowania



Opis bloku

Piksele są odczytywane z pamięci użytkownika, a następnie przekształcane. W rezultacie otrzymywane są współrzędne tekstur. Przetworzone dane zapisuje są w ***pamięci tekstur*** lub przesyła bezpośrednio do ***rasteryzacji***.

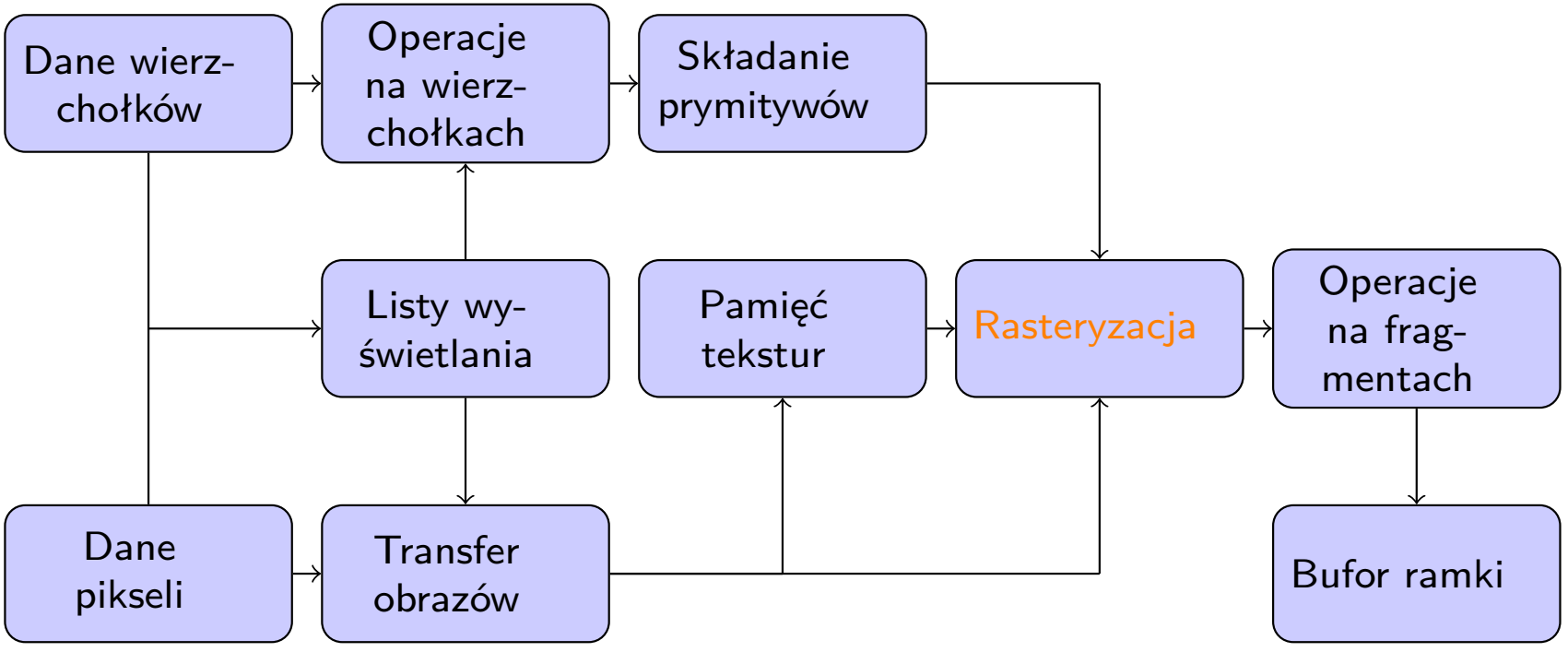
Schemat ustalonego potoku renderowania



Opis bloku

W tym bloku zapamiętywane są tekstury, które będą nakładane na kształty geometryczne.

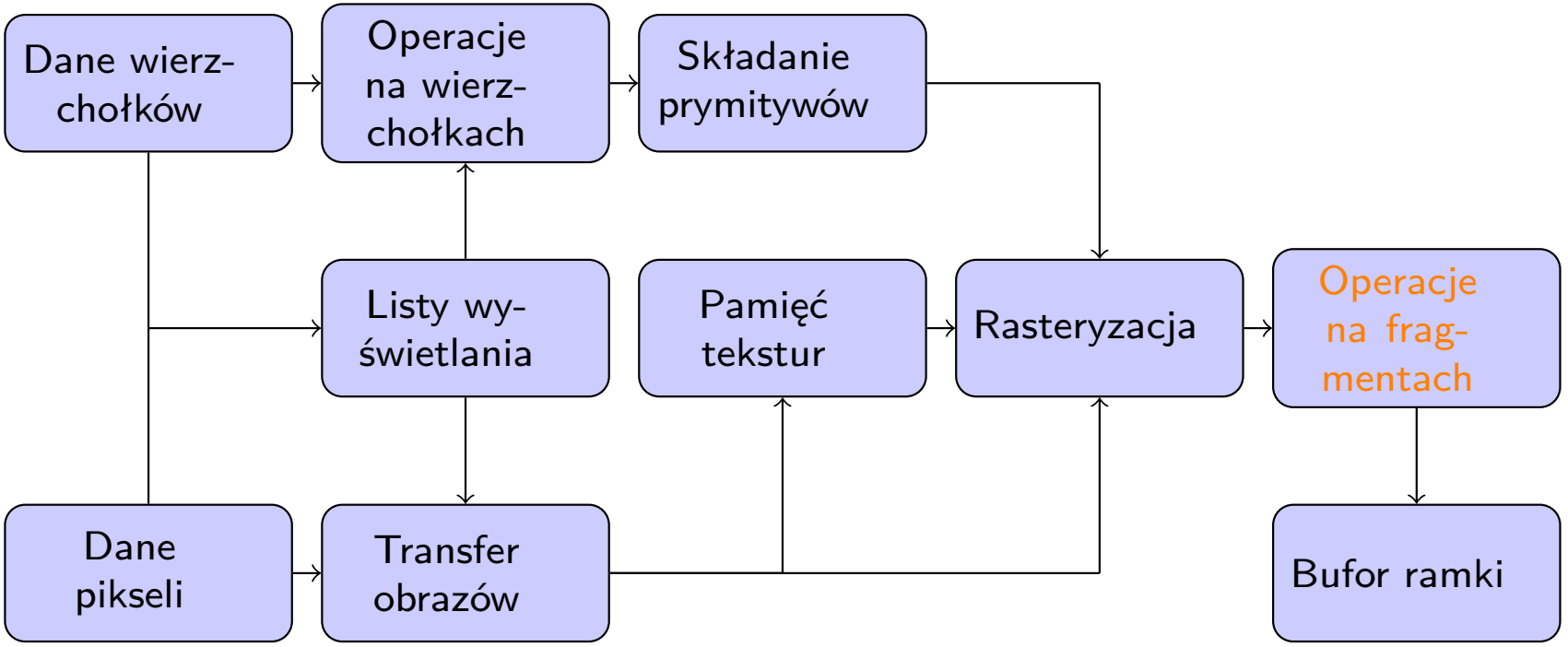
Schemat ustalonego potoku renderowania



Opis bloku

Resteryzacja (ang. *rasterization*) określa fragmenty i pozycję pikseli w prymitywach. ***Fragment*** odpowiada jednemu pikselowi w buforze ramki i jest wykorzystywany do jego aktualizacji. Ponadto, przechowują informacje o kolorze, głębokości, normalnych, wsółrzędnych tekstur, itp.

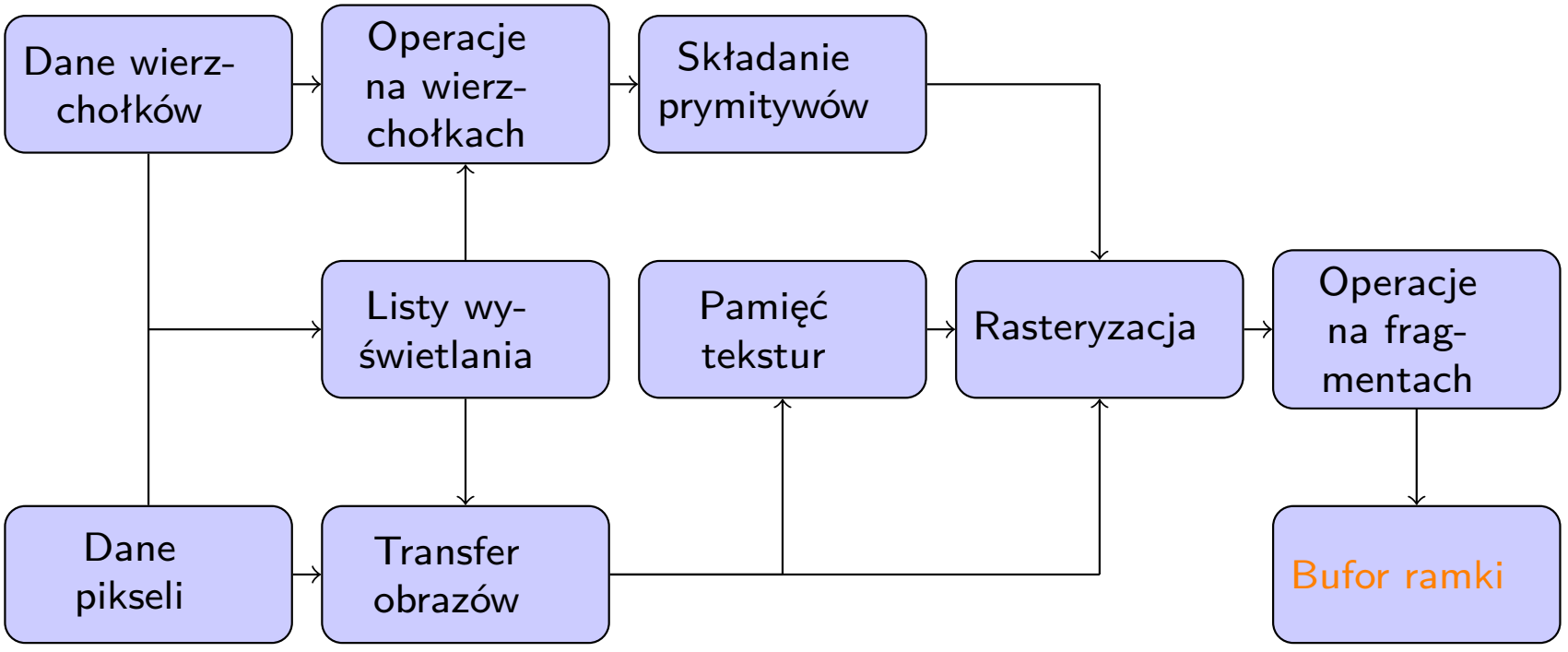
Schemat ustalonego potoku renderowania



Opis bloku

Przed wykonaniem ***operacji na fragmentach*** (ang. *fragment operation*) generowane są teksle, a następnie wykonywane są obliczenia związane z mgłą, mieszaniem kolorów, operacje bitowe, testy (obcinania, testy alfa, głębokości, itp.).

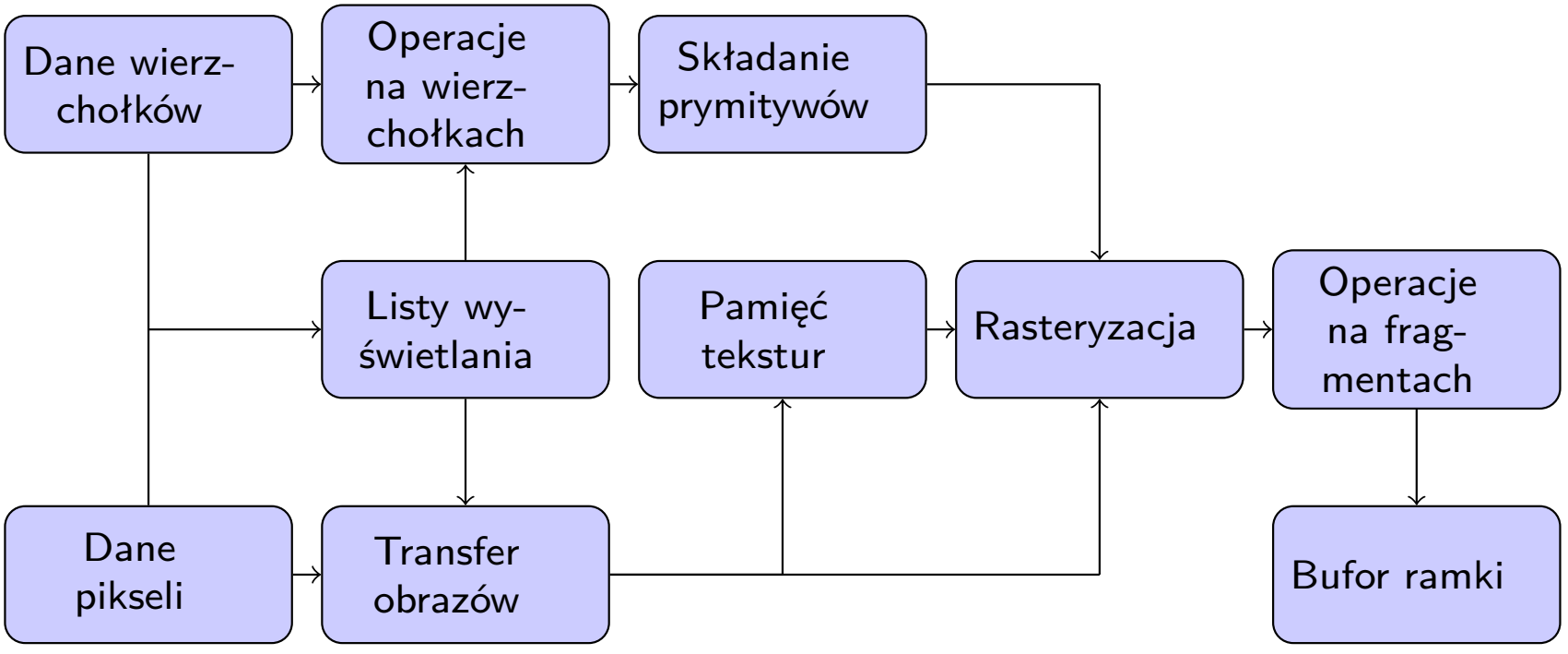
Schemat ustalonego potoku renderowania



Opis bloku

Jeśli operacje zakończą się powodzeniem, informacje przechowywane we fragmentach są wykorzystywane do uaktualnienia wartości pikseli w ***buforze ramki***.

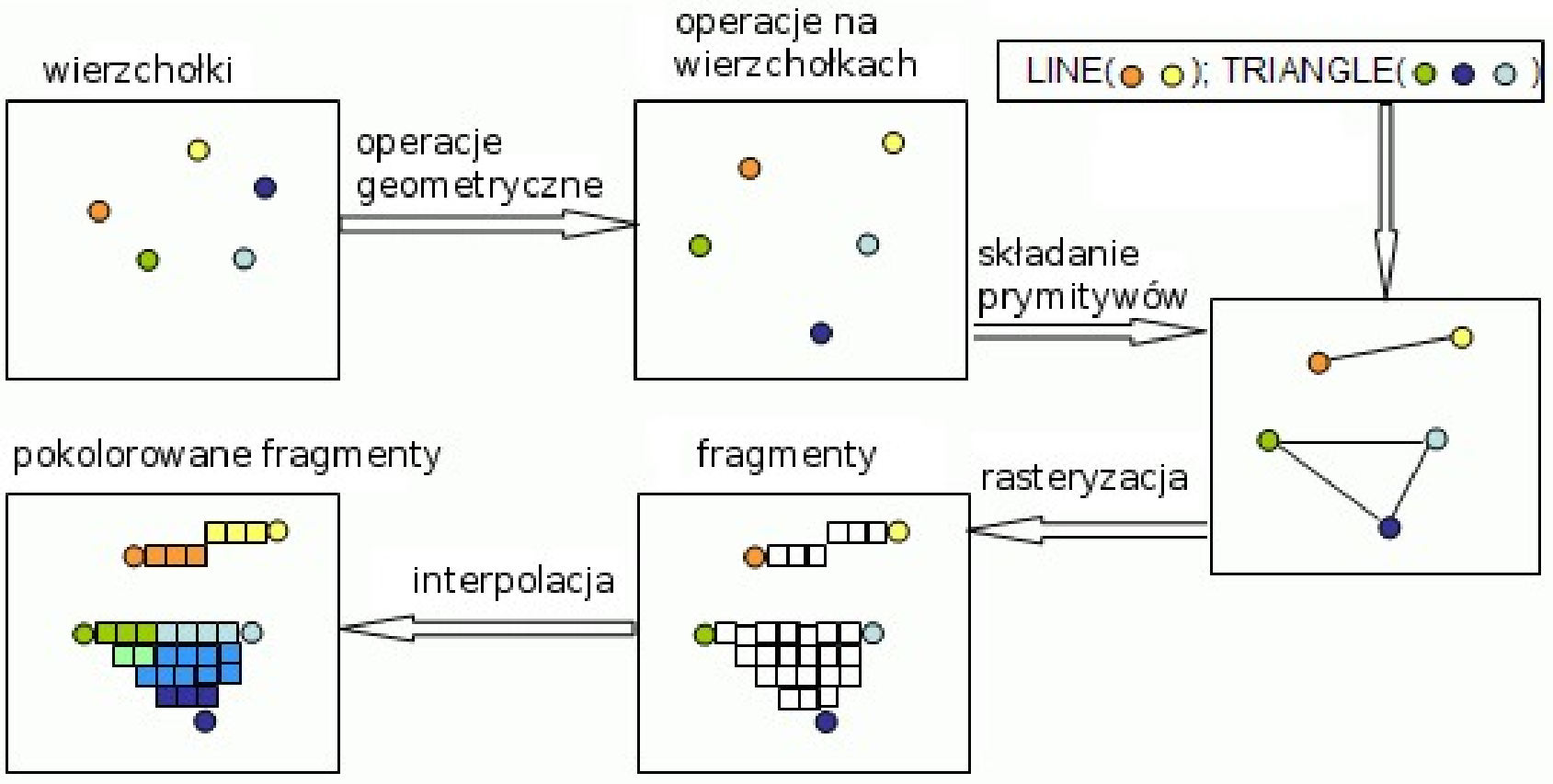
Schemat ustalonego potoku renderowania



Opis bloku

Jeśli operacje zakończą się powodzeniem, informacje przechowywane we fragmentach są wykorzystywane do uaktualnienia wartości pikseli w ***buforze ramki***.

Wizualizacja potoku renderowania



Programowy potok renderowania

Ustalony potok renderowania nie pozwalał na wykorzystanie w pełni mocy obliczeniowych kart graficznych. ***Programowy potok renderowania*** (ang. *programmable pipeline*) dostarcza programiście większe możliwości i daje bardziej elastyczne podejście do tworzenia efektów. Za jego pomocą można wykonywać operacje na wierzchołku - transformacje, ustawianie kolorów, współrzędnych tekstur oraz pokrywać materiałem obiekty. Jest jednak trudniejszy w programowaniu, ponieważ programista sam musi zadbać o uruchomienie poszczególnych etapów renderingu.

Wstęp

Shader jest programem, który pozwala programiście kontrolować potok renderingu w procesorze graficznym (***GPU*** - ang. *graphical processing unit*). Za jego pomocą można wpływać na przebieg obliczeń i renderingu grafiki 3D. Shadery są wykonywane równolegle z programem procesora i zwykle wykorzystują wiele niezależnych rdzeni GPU.

Wstęp

Shader jest programem, który pozwala programiście kontrolować potok renderingu w procesorze graficznym (***GPU*** - ang. *graphical processing unit*). Za jego pomocą można wpływać na przebieg obliczeń i renderingu grafiki 3D. Shadery są wykonywane równolegle z programem procesora i zwykle wykorzystują wiele niezależnych rdzeni GPU.
OpenGL i DirectX obsługują 3 rodzaje shaderów:

- ***Shader wierzchołków*** (ang. *vertex shader*) - wykonuje operacje osobno dla każdego wierzchołka.
- ***Shader geometrii*** (ang. *geometry shader*) - pozwala na usuwanie lub tworzenie nowych wierzchołków.
- ***Shader pikseli*** (ang. *pixel/fragment shader*) - wykonuje operacje na kolorach osobno dla każdego piksela (fragmentu) sceny.

Vertex shader

Vertex shader jest realizowany sprzętowo, przy użyciu jednostki ***vertex processor***. Nie posiada informacji o innych wierzchołkach. Za jego pomocą można modyfikować pozycję wierzchołka, jego barwę, współrzędne tekstury i wektor normalny.

Vertex shader

Vertex shader jest realizowany sprzętowo, przy użyciu jednostki ***vertex processor***. Nie posiada informacji o innych wierzchołkach. Za jego pomocą można modyfikować pozycję wierzchołka, jego barwę, współrzędne tekstury i wektor normalny.
Jest wykorzystywany do uzyskiwania efektów tj.:

- zniekształcenie obiektów.
- ruch powierzchni wody.
- mimika twarzy, itp.

Pixel shader

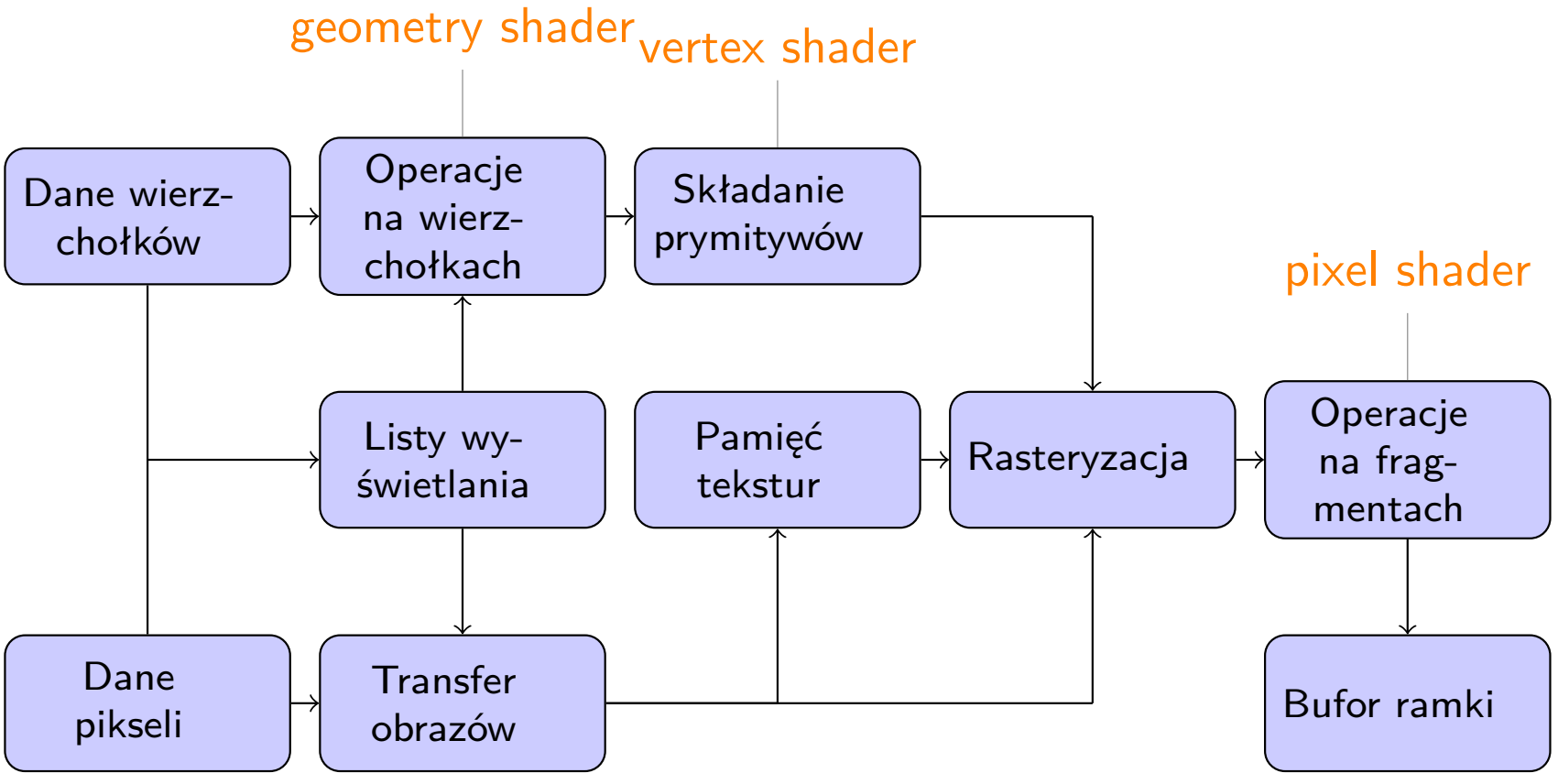
Pixel shader pozwala modyfikować barwę piksela obrazu podczas rasteryzacji. Jest realizowany sprzętowo, za pomocą jednostki ***pixel processor***. Pozwala na obliczenia kolorów i współrzędnych tekstury dla każdego piksela, nałożenie tekstury, obliczenie mgły, itp.

Pixel shader

Pixel shader pozwala modyfikować barwę piksela obrazu podczas rasteryzacji. Jest realizowany sprzętowo, za pomocą jednostki ***pixel processor***. Pozwala na obliczenia kolorów i współrzędnych tekstury dla każdego piksela, nałożenie tekstury, obliczenie mgły, itp.
Pozwala na uzyskanie efektów związanych z oświetleniem tj.:

- chropowatość powierzchni.
- fale na wodzie.
- odbicia w wodzie.
- cienie.
- eksplozje, itp.

Rozmieszczenie shaderów w potoku renderowania



Wstęp

Dla programowania shaderów, utworzone zostały specjalistyczne języki programowania:

- ***Cg*** - został opracowany przez firmę nVidia we współpracy z Microsoftem, jest kompatybilny z OpenGL i DirectX.
- ***HLSL*** (ang. *High Level Shader Language*) - został opracowany przez Microsoft, jest używany w Direct3D.
- ***GLSL*** (ang. *OpenGL Shading Language*) - był dostępny jako rozszerzenie ARB w OpenGL 1.4, dopiero w wersji 2.0 został z nim zintegrowany.

Ich składnia jest zbliżona do C/C++. Języki zawierają gotowe klasy, struktury, zmienne, instrukcje sterujące oraz funkcje pomocnicze.

Konfiguracja shaderów w OpenGL

Shadery nie są emulowane programowo, do działania wymagają odpowiedniego sprzętu komputerowego. Biblioteka *GLEW* upraszcza użycie rozszerzeń *GL_ARB_fragment_shader* i *GL_ARB_vertex_shader* oraz funkcji OpenGL 2.0 do korzystania z shaderów. Różnice pomiędzy dwoma podejściami są niewielkie, zwykle nieznacznie różnią się nazwy funkcji. Jednak, OpenGL 2.0 jest obowiązującym standardem, dlatego tylko ten sposób będzie dalej rozważany.

Sprawdzenie kompatybilności z OpenGL 2.0

Sprawdzenie kompatybilności z OpenGL 2.0

```
#include <stdio.h>
#include <GL/glew.h>
#include <GL/glut.h>

int main(int argc, char *argv[])
{

   glutInit (&argc,argv);
   glutInitDisplayMode (GLUT_RGB);
   glutInitWindowSize (400,400);
   glutCreateWindow ("Test Kompatybilnosci");

   glewInit();

      if (glewIsSupported("GL_VERSION_2_0"))
      printf("Kompatybilny z OpneGL 2.0\n");
   else {
      printf("OpenGL 2.0 nie jest wspierany\n");
      return 1;
   }

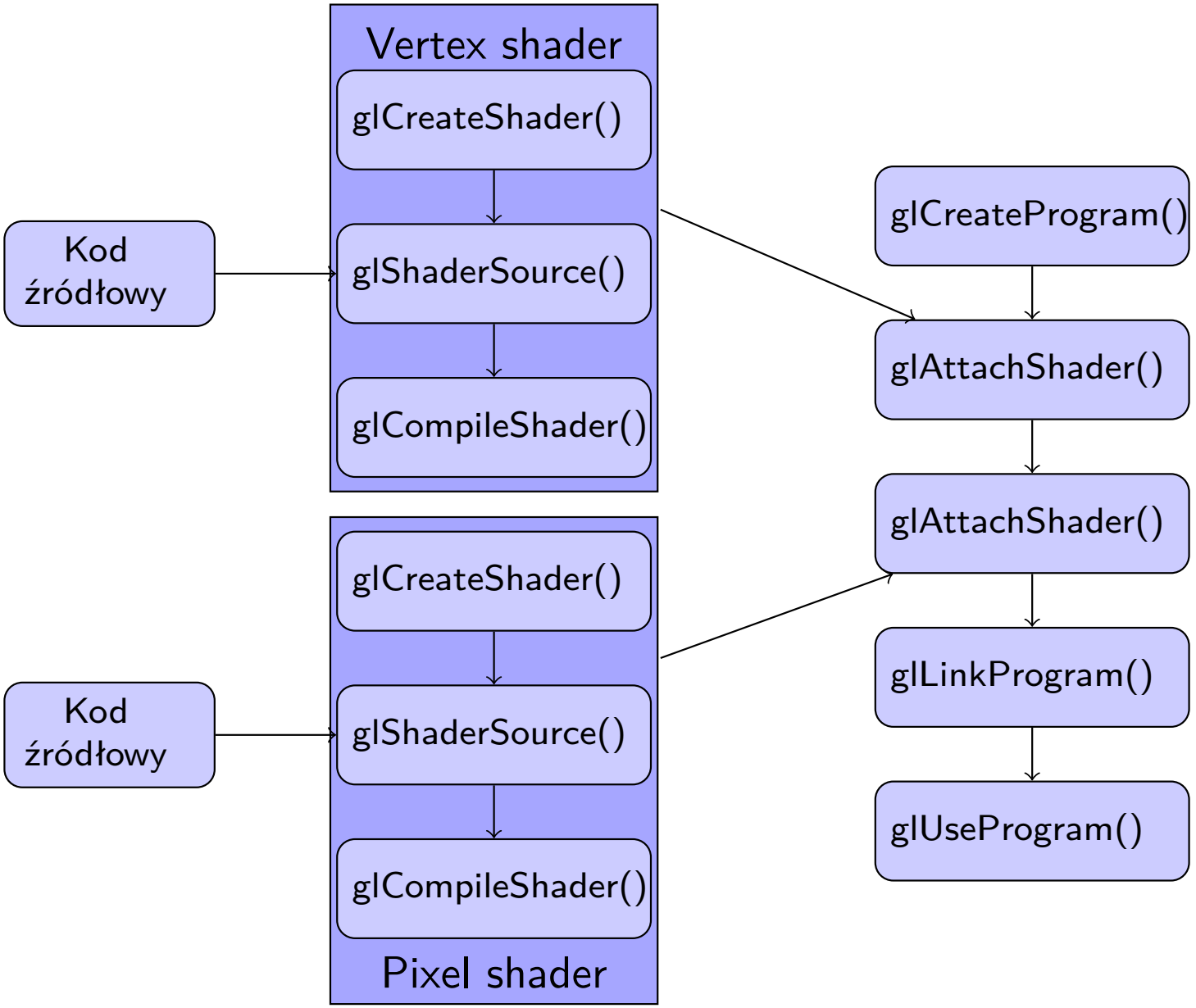
   return 0;
}
```

Tworzenie shaderów

Kod źródłowy shaderów (podobnie jak moduły w języku C) musi być skompilowany niezależnie od programu głównego, a następnie zlinkowany z kodem wynikowym pozostałych shaderów i programu głównego.

Do programu można dodawać wiele shaderów, ale może być tylko jedna funkcja *main()* dla wszystkich *vertex shaderów* i jedna dla wszystkich *pixel shaderów*.

Etapy tworzenia shaderów



Utworzenie obiektu shadera

Pierwszym etapem jest utworzenie obiektu shadera, który będzie przechowywał wszystkie informacje. Funkcja zwraca referencję do shadera.

Utworzenie obiektu shadera

GLuint **glCreateShader**(GLenum *shaderType*);

Parametry funkcji

- *shaderType* - typ shadera, przyjmuje wartości: *GL_VERTEX_SHADER* lub *GL_FRAGMENT_SHADER*

Wczytanie kodu źródłowego

Do shadera należy przekazać kod źródłowy, zwykle odczytuje się go z pliku. Kod źródłowy przekazywany jest w postaci tablicy ciągu znaków.

Wczytanie kodu źródłowego

VOID **glShaderSource**(GLUINT *shader*, INT *numOfStrings*,
CONST CHAR ***strings*, INT **lenOfStrings*);

Parametry funkcji

- *shader* – referencja do shadera.
- *numOfStrings* – liczba ciągów znaków w tablicy.
- *strings* – tablica ciągów znaków.
- *lenOfStrings* – tablica z długościami każdego ciągu znaków lub NULL jeżeli ciągi zakończone są kodem zera.

Kompilacja shadera

Ostatnim etapem jest kompilacja shadera.

Kompilacja shadera

VOID **glCompileShader**(GLUINT *shader*)

Parametry funkcji

- *shader* – referencja do shadera.

Utworzenie programu shadera

Na początku, należy utworzyć obiekt programu shadera, który będzie przechowywał wszystkie informacje. Można utworzyć dowolną liczbę programów i przełączać się między nimi nawet w trakcie renderowania jednej klatki animacji. Funkcja nie przyjmuje żadnych parametrów i zwraca referencję do programu.

Utworzenie programu shadera

GLuint **glCreateProgram**(void);

Dołączenie shadera do programu

Następnym krokiem jest dołączenie do programu shadera, który został już wcześniej utworzony.

Dołączenie shadera do programu

VOID **glAttachShader**(GLUINT *program*, GLUINT *shader*)

Parametry funkcji

- *program* – referencja do programu.
- *shader* – referencja do shadera, który ma zostać dołączony.

Dołączenie shadera do programu

Następnym krokiem jest dołączenie do programu shadera, który został już wcześniej utworzony.

Dołączenie shadera do programu

VOID **glAttachShader**(GLUINT *program*, GLUINT *shader*)

Parametry funkcji

- *program* – referencja do programu.
- *shader* – referencja do shadera, który ma zostać dołączony.

Uwagi

Ten sam shader można dołączać do różnych programów.

Linkowanie programu

Ostatnim etapem jest linkowanie programu. Po tej operacji kod źródłowy shadera może być modyfikowany, a shader ponownie kompilowany i linkowany bez wpływu na program.

Linkowanie programu

VOID **glLinkProgram**(GLUINT *program*)

Parametry funkcji

- *program* – referencja do programu.

Używanie programów

Można mieć wiele programów podlinkowanych i gotowych do użycia, ale dopiero po wywołaniu funkcji *glUseProgram*, program jest rzeczywiście ładowany i używany. Jeśli program jest w użyciu i jest ponownie linkowany, to automatycznie będzie znowu używany.

Użycie programu

VOID **glUseProgram**(GLUINT *program*)

Parametry funkcji

- *program* – referencja do programu, który ma być użyty lub zero, aby powrócić do standardowych procedur OpenGL.

Odłączenie shadera

Istnieje również funkcja do odłączania shaderów od programu. Tylko shadery, które są odłączone mogą zostać usunięte. Gdy shader jest podłączony, zostanie tylko zaznaczony do usunięcia, a usunięty dopiero po odłączeniu go ze wszystkich programów.

Odłączenie shadera

VOID **glDetachShader**(GLUINT *program*, GLUINT *shader*)

Parametry funkcji

- *program* – referencja programu do odłączenia.
- *shader* – referencja do shadera.

Usunięcie shadera i programu

Ostatecznie shader, jak również i program można usunąć.

Usunięcie shadera i programu

VOID **glDeleteShader**(GLUINT *shader*)
VOID **glDeleteProgram**(GLUINT *program*)

Parametry funkcji

- *shader* – referencja shadera do usunięcia.
- *program* – referencja programu do usunięcia.

Błędy kompilacji

Szukanie błędów w kodzie źródłowym shadera jest utrudnione, ponieważ nie można podejrzeć żadnych wartości. Istnieją funkcję, za pomocą których można sprawdzić status kompilacji.

Sprawdzenie statusu kompilacji

VOID **glGetShaderiv**(GLUINT *object*, GLENUM *type*, INT **param*)

Parametry funkcji

- *object* – referencja do programu lub shadera.
- *type* – przyjmuje tutaj wartość *GL_COMPILE_STATUS*.
- *param* – zwraca status, *GL_TRUE* - OK lub *GL_FALSE* - błąd.

Błędy linkowania

Podobnie sprawdza się błędy podczas linkowania.

Sprawdzenie statusu linkowania

VOID **glGetProgramiv**(GLUINT *object*, GLENUM *type*, INT **param*)

Parametry funkcji

- *object* – referencja do programu lub shadera.
- *type* – przyjmuje tutaj wartość *GL_LINK_STATUS*.
- *param* – zwraca status, *GL_TRUE* - OK lub *GL_FALSE* - błąd.

Błędy linkowania

Podobnie sprawdza się błędy podczas linkowania.

Sprawdzenie statusu linkowania

VOID **glGetProgramiv**(GLUINT *object*, GLENUM *type*, INT **param*)

Parametry funkcji

- *object* – referencja do programu lub shadera.
- *type* – przyjmuje tutaj wartość *GL_LINK_STATUS*.
- *param* – zwraca status, *GL_TRUE* - OK lub *GL_FALSE* - błąd.

Uwagi

W obu funkcjach parametr *type* może przyjmować więcej wartości, w zależności od rodzaju sprawdzanych informacji.

Informacje o błędach

Można uzyskać więcej informacji np. o błędach składni, a także o niekompatybilności ze sprzętem lub oprogramowaniem.

Pobranie informacji o błędach

VOID **glGetShaderInfoLog**(GLUINT *obj*, INT *max*, INT **len*, CHAR **log*)
VOID **glGetProgramInfoLog**(GLUINT *obj*, INT *max*, INT **len*, CHAR **log*)

Parametry funkcji

- *obj* – referencja do programu lub shadera.
- *max* – maksymalna długość informacji o błędzie.
- *len* – rzeczywista długość informacji.
- *log* – zwracana wiadomość o błędach.

Informacje o błędach

Można uzyskać więcej informacji np. o błędach składni, a także o niekompatybilności ze sprzętem lub oprogramowaniem.

Pobranie informacji o błędach

VOID **glGetShaderInfoLog**(GLUINT *obj*, INT *max*, INT **len*, CHAR **log*)
VOID **glGetProgramInfoLog**(GLUINT *obj*, INT *max*, INT **len*, CHAR **log*)

Parametry funkcji

- *obj* – referencja do programu lub shadera.
- *max* – maksymalna długość informacji o błędzie.
- *len* – rzeczywista długość informacji.
- *log* – zwracana wiadomość o błędach.

Uwagi

Informacje o błędach mogą być różne, na różnym sprzęcie komputerowym i dla różnych sterowników.

Przykład

Przykładowy program zawiera etapy konfiguracji shadera i programu - ich tworzenie, linkowanie oraz używanie. W oknie tekstowym wyświetlane są informacje o pomyślnym skonfigurowaniu shaderów lub o błędach podczas kompilacji i linkowania.

Przykładowy program





Bibliografia


R. Fernando, Mark J. Kilgard:
Język Cg. Programowanie grafiki w czasie rzeczywistym..
Wydawnictwo Helion, Gliwice, 2003.


OpenGL Shading Language.
http://www.opengl.org/documentation/glsl


Lighthouse 3D - GLSL Tutorial
http://www.lighthouse3d.com/opengl/glsl



KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI



Politechnika Świętokrzyska

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ SPOŁECZNY



Projekt współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego

" Program Rozwojowy Potencjału Dydaktycznego Politechniki Świętokrzyskiej w Kielcach: *kształcenie w nowoczesnych obszarach techniki* "

Program Operacyjny Kapitał Ludzki, Priorytet IV Szkolnictwo wyższe i nauka, Działanie 4.1. Wzmocnienie i rozwój potencjału dydaktycznego uczelni oraz zwiększenie liczby absolwentów kierunków o kluczowym znaczeniu dla gospodarki opartej na wiedzy, Poddziałanie 4.1.1. Wzmocnienie potencjału dydaktycznego uczelni.

Umowa UDA–POKL.04.01.01-00395/09-00