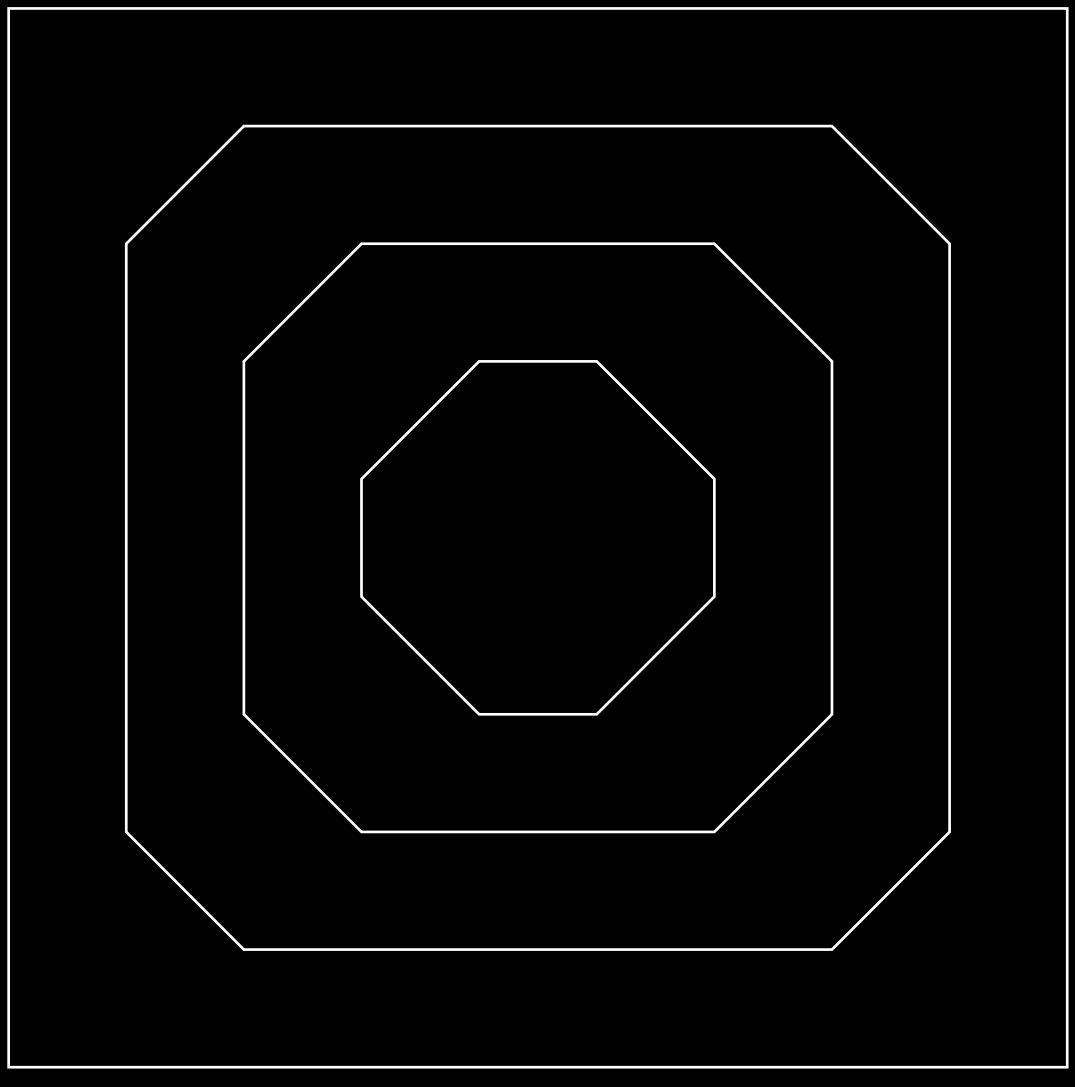
Monografie Politechniki Łódzkiej

Wizualizacja zjawisk topnienia i sublimacji

Dominik Szajerman



Dominik Szajerman

Wizualizacja zjawisk topnienia i sublimacji

Łódź 2016

Dominik Szajerman

Wizualizacja zjawisk topnienia i sublimacji

Monografie Politechniki Łódzkiej
Łódź, 2016

Recenzenci:
prof. dr hab. Mykhaylo Yatsymirskyy
dr hab. inż. Piotr Napieralski

Redaktor Naukowy Wydziału Fizyki Technicznej,
Informatyki i Matematyki Stosowanej:
dr hab. inż. Aneta Poniszewska-Marańda

Projekt okładki:
dr inż. Krzysztof Guzek

Instytut Informatyki Politechnika Łódzka
90-924 Łódź, ul. Wólczańska 215
e-mail: dominik.szajerman@p.lodz.pl

WYDAWNICTWO POLITECHNIKI ŁÓDZKIEJ
90-924 Łódź, ul. Wólczańska 223
tel. 42-631-20-87, 42-631-29-52
fax 42-631-25-38
e-mail: zamowienia@info.p.lodz.pl
www.wydawnictwa.p.lodz.pl

ISBN 978-83-7283-888-9

Reprodukcja z materiałów dostarczonych przez Autorów

Nakład 50 egz. Ark. wyd. 12,0. Ark. druk. 14,0. Papier offset. 80 g, 70 x 100
Wykonano w Drukarni „Quick-Druk" s.c. 90-562 Łódź, ul. Łąkowa 11
Nr 2243

Książkę tę dedykuję mojej żonie, Annie.

Spis treści

1 Wprowadzenie **9**

1.1 Przemiany fazowe 9

1.2 Aktualny stan wiedzy w zakresie symulacji i wizualizacji przemian fazowych 12

1.3 Zawartość niniejszej monografii 23

1.4 Wykaz używanych oznaczeń i skrótów 25

2 Reprezentacje geometrii obiektów trójwymiarowych **27**

2.1 Klasyfikacja reprezentacji obiektów trójwymiarowych 27

2.2 Siatka trójkątów 39

2.3 Poprawność budowy siatki trójkątów 43

2.4 Jakość siatki trójkątów 45

2.5 Atrybuty siatki trójkątów 48

2.6 Rendering siatek trójkątów 50

2.6.1 Klasyczny potok renderingu 51

2.6.2 Programowalny potok renderingu 56

2.7 Podsumowanie 61

3 Modyfikacja siatki obiektu w procesie sublimacji **67**

3.1 Deformacja siatki powierzchni 67

3.2 Upraszczanie siatki 75

3.3 Podział obiektu 78

3.4 Czynniki fizyczne 79
3.5 Reprezentacja deformowalnej siatki trójkątów 81
3.6 Algorytmy przetwarzania siatki trójkątów 84
3.7 Podsumowanie 90

4 Programowanie jednostek graficznych 91
4.1 Technologie GPGPU 91
4.2 GPGPU realizowane przez programowalny potok renderingu . 96
4.2.1 Model cieniowania 4.0 97
4.2.2 Tekstury w przetwarzaniu danych na GPU 105
4.2.3 Mapowanie problemu przetwarzania na GPU 109
4.2.4 Optymalizacja kodu programów cieniowania 113
4.2.5 Operacje strumieniowe 116
4.3 Przykład mapowania problemu na GPGPU 119
4.4 Podsumowanie 122

5 Optymalizacja algorytmów dla GPU 125
5.1 Propozycja odpowiednika reprezentacji indeksowej dla GPU . 125
5.2 Operacje strumieniowe 130
5.3 Deformacja siatki 134
5.4 Usuwanie krawędzi siatki 136
5.5 Usuwanie sklejonych trójkątów 138
5.6 Rendering obiektu 139
5.7 Podsumowanie 140

6 Wizualizacja bryły lodu 145
6.1 Światło rozproszone 146
6.2 Odbicia zwierciadlane i rozbłyski 148
6.3 Transmisja światła w lodzie 150
6.4 Nierówności powierzchni 152
6.5 Podsumowanie 154

7 Wizualizacja topnienia 159
7.1 Zjawisko topnienia a zjawisko sublimacji 159
7.2 Wizualizacja fazy ciekłej 160
7.2.1 Przezroczystość wody 162
7.2.2 Odbicia otoczenia na powierzchni wody 163
7.2.3 Wypukłość wody 164
7.3 Ruch fazy ciekłej 166
7.3.1 Rozlewanie się cieczy 167
7.3.2 Przyrost ilości cieczy 169
7.4 Podsumowanie 172

8 Weryfikacja wyników 175
8.1 Scenariusz weryfikacji 175
8.2 Konfiguracja stanowiska 177
8.3 Weryfikacja wyników przetwarzania geometrii 178
8.4 Porównywanie konturów dla dalszych fotografii sekwencji . . 184
8.5 Podsumowanie 185

9 Podsumowanie 187

Bibliografia 191

Spis rysunków 203

Spis tabel 211

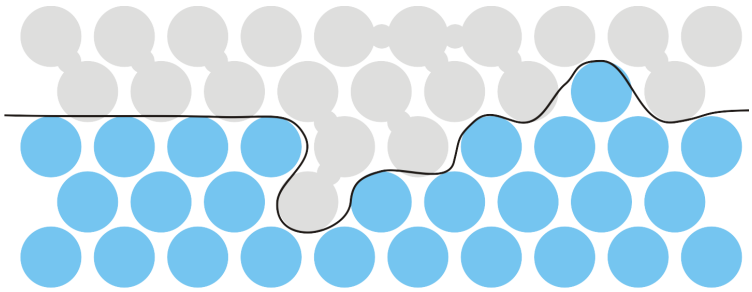
1

Wprowadzenie

Niniejsza monografia dotyczy wizualizacji zjawisk topnienia i sublimacji, które są przejściem fazowym z ciała stałego odpowiednio do cieczy i gazu. Modelem granicy między dwoma fazami jest powierzchnia międzyfazowa, dlatego topnienie i sublimacja mogą być rozpatrywane jako przesuwanie powierzchni międzyfazowej z towarzyszącą mu wymianą ciepła. Wizualizacja omawianych zajawisk wymaga omówienia różnych jej aspektów – od sposobu reprezentacji danych graficznych, przez algorytmy przetwarzania tych danych i ich optymalizację, problemy renderingu czasu rzeczywistego, po metody weryfikacji jej wyników. Wymienione kwestie zostały zebrane w niniejszej książce.

1.1 Przemiany fazowe

Przemiana fazowa (przejście fazowe, zmiana stanu skupienia) jest taką zmianą układu fizycznego lub chemicznego, przy której zachodzi skokowa zmiana parametrów układu. Takim parametrem może być na przykład ciepło właściwe lub gęstość. Najczęściej przemiany fazowe rozpatrywane są w funkcji



Rysunek 1.1. Powierzchnia międzyfazowa (*na podstawie opisu w [5]*).

temperatury. Niniejsza monografia zajmuje się wizualizacją zjawisk topnienia i sublimacji, a więc przejścia z fazy stałej do fazy ciekłej w przypadku topnienia lub do fazy gazowej w przypadku sublimacji. Wizualizacja tych zjawisk wiąże się z ukazaniem stopniowej deformacji obiektu w fazie stałej. Przejście do fazy gazowej nie wymaga wizualizacji bezbarwnego gazu, natomiast topnienie związane jest z wizualizacją powstałej cieczy.

W fizycznym modelu topnienia energia cząstek cieczy i ciała stałego różni się o wartość stałą dla danej substancji. Jest to ciepło topnienia, czyli energia, jaką trzeba dostarczyć zadanej ilości substancji, aby zmienić jej fazę. Temperatura topnienia jest temperaturą, w której przy zadanym ciśnieniu zachodzi proces przemiany fazowej. Modelem granicy między dwoma fazami substancji podczas zachodzenia przemiany jest powierzchnia międzyfazowa (rys. 1.1). W temperaturze topnienia proces topnienia i krystalizacji zachodzi równocześnie, przy czym jednakowo szybko tylko wtedy, gdy temperatura powierzchni międzyfazowej odpowiada stanowi równowagi. Powyżej tej temperatury przewagę zyskuje szybkość procesu topnienia, poniżej – krzepnięcia.

Szybkość topnienia i krystalizacji można uzależnić od kilku wielkości różnych dla obu stanów skupienia [5]:

— energii aktywacji topnienia i krystalizacji,
— liczby atomów cieczy i atomów ciała stałego przy powierzchni międzyfazowej,

— liczby atomów o odpowiedniej energii poruszających się w kierunku normalnym do powierzchni międzyfazowej w obu fazach w stosunku do całkowitej liczby atomów,
— prawdopodobieństwa oddania nadmiaru energii do cieczy przez cząstkę wbudowaną do sieci krystalicznej i prawdopodobieństwa pobrania brakującej energii przez cząstkę opuszczającą sieć krystaliczną,
— częstotliwości drgań cząstek w obu fazach.

Przyjmując fizyczne uproszczenie ciała stałego jako izotropowej bryły o jednorodnej makroskopowo powierzchni topnienia, proces topnienia może być rozpatrywany jak każdy proces wymiany ciepła, jeśli tylko uwzględni się przesuwanie powierzchni międzyfazowej spowodowane topnieniem [5].

Podobnie może być rozpatrywana sublimacja, która zachodzi zawsze przy temperaturze mniejszej niż temperatura punktu potrójnego[1] dla danej substancji [89].

Topnienie i sublimacja to procesy zachodzące płynnie od powierzchni obiektu ku jego wnętrzu. Dzieje się tak dlatego, że każda molekuła substancji emituje bądź absorbuje energię cieplną zgodnie z drugą zasadą dynamiki. Energia jest dostarczana do substancji przez otoczenie. Energia danej molekuły płynnie wzrasta dopóki nie osiągnie wartości ciepła topnienia. Wtedy zachodzi zmiana jej fazy.

Molekuły można uporządkować w warstwy zależnie od ich odległości od powierzchni obiektu. Zewnętrzne warstwy obiektu absorbują więcej energii (ciepła) niż wewnętrzne, do których dociera tylko niezaabsorbowana jej część. Wewnętrzne warstwy osiągają więc energię ciepła topnienia później niż zewnętrzne. Z tego powodu przy tej samej masie cieńsze obiekty lub obiekty o nieregularnej powierzchni topnieją szybciej niż grubsze lub o gładkiej powierzchni mimo, że energia potrzebna do zakończenia procesu jest taka sama. Podsumowując, szybkość topnienia zależy od szybkości przepływu ciepła, a więc od cech geometrycznych obiektu.

[1] Punkt potrójny – stan, w którym dana substancja może istnieć w trzech fazach jednocześnie w równowadze termodynamicznej; określa go temperatura i ciśnienie punktu potrójnego charakterystyczne dla danej substancji.

W podejściu objętościowym opis zjawiska topnienia i sublimacji bazuje na obliczaniu energii zaabsorbowanej i przekazywanej przez każdy fragment substancji (np. molekułę). Kiedy molekuła osiągnie energię ciepła topnienia przestaje absorbować energię i zostaje usunięta ze zbioru molekuł w fazie stałej. Może to być rozpatrywane jako ruch powierzchni obiektu.

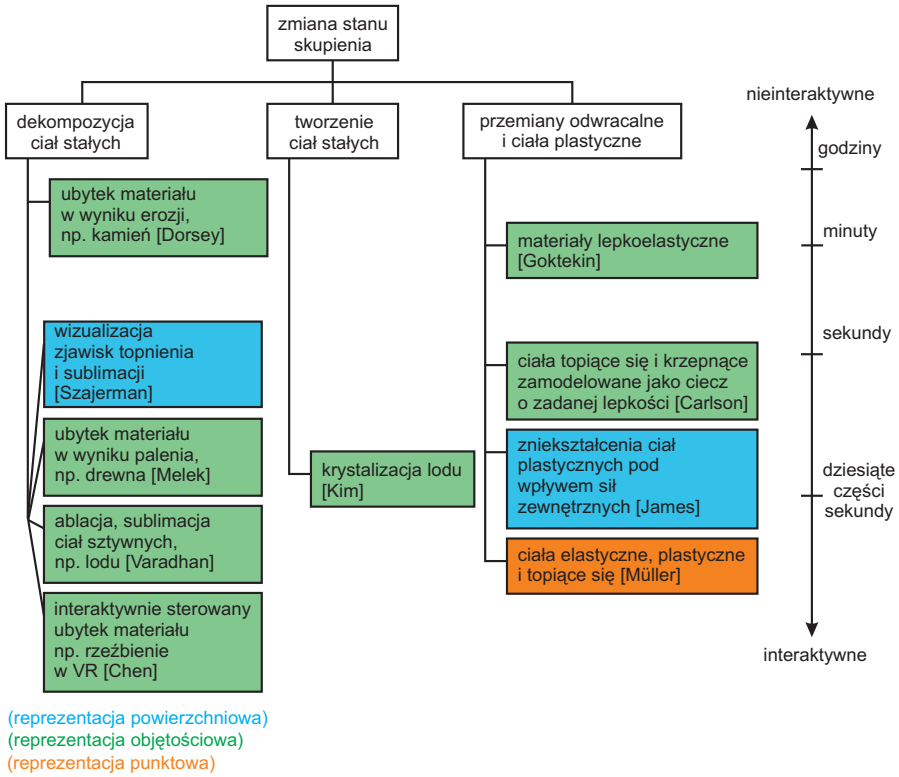
Niniejsza monografia bazuje na wykorzystaniu deformacji powierzchni obiektu, a więc podejścia powierzchniowego, do uzyskania efektów podobnych, jak uzyskiwane w podejściu objętościowym.

1.2 Aktualny stan wiedzy w zakresie symulacji i wizualizacji przemian fazowych

Symulacja procesów fizycznych jest jednym z głównych kierunków badań w informatyce. Z kolei wizualizacja symulacji jest jednym z zastosowań grafiki komputerowej. Z tego względu zjawiska przemian fazowych znalazły swoje miejsce w badaniach grafiki komputerowej jako wizualizacja na potrzeby symulacji. Spotyka się również wizualizację bardziej uogólnionych zjawisk zmian atrybutów wizualnych obiektów, tworzoną na potrzeby wymyślonych efektów specjalnych.

Podział technik wizualizacji tych zjawisk może być dokonany ze względu na kierunek zmian kształtu wizualizowanego obiektu (rys. 1.2). Pierwszą grupę zjawisk tworzą te polegające na dekompozycji ciał stałych. Dekompozycja ta może polegać na topnieniu ciał, które podczas tego procesu stają się plastyczne, jak na przykład wosk [7, 8]. Może to być także topnienie i sublimacja ciał sztywnych, takich jak lód [96, 23, 22]. Inny rodzaj efektów wizualnych to ubytek i zmiana wyglądu drewna podczas palenia [56] lub kamienia w wyniku erozji [13, 66], jak również efekty interaktywne, na przykład na potrzeby sztuki – rzeźbienie w rzeczywistości wirtualnej [11]. Drugą grupę zjawisk stanowią zjawiska odwrotne, których wynikiem będzie stworzenie ciała stałego. Może to być twardnienie substancji takich jak lawa, cement, wosk czy lód [7, 8], czy też krzepnięcie i krystalizacja [44, 45, 46].

Trzecią możliwością jest symulacja lub wizualizacja ciał plastycznych zapoczątkowana w [92], ciał znajdujących się na pograniczu ciał stałych i cieczy [35, 59, 60] lub ciał lepkoelastycznych [28].



Rysunek 1.2. Klasyfikacja wizualizacji z punktu widzenia kierunku zmian stanów skupienia. Wybrane techniki uporządkowane są według czasu obliczenia pojedynczego kroku przetwarzania geometrii i renderingu jednej ramki.

Zastosowanie powyższych metod wizualizacji może być bardzo różne:

— edukacyjne, na przykład interaktywny trening operacji medycznych [96],
— sztuka w wirtualnej rzeczywistości [11],
— aplikacje interaktywne [35, 59],

— naukowe, jeśli wizualizacja jest połączona z symulacją, na przykład wizualizacja krystalizacji [44], wpływu pogody na materiały [13], ablacji[2] [96],
— rozrywka – gry komputerowe (np. „1C Company: Cryostasis", „Melt The Ice Cubes" dla iPod, „YoyoGames: Melting Ice") oraz filmy animowane i filmowe efekty specjalne (np. „Terminator 2", „House of Wax", „Ice Age").

W zależności od zastosowania danej techniki wizualizacji wymagany jest rendering w czasie umożliwiającym interakcję (np. aplikacje treningowe) lub rendering, w którym czas obliczeń nie ma kluczowego znaczenia (np. efekty specjalne dla filmu). Różny może być również wymagany stopień wierności odwzorowania zjawiska. Choć nie jest to regułą, gdy pożądana jest możliwie wierna symulacja zjawisk fizycznych używa się objętościowych modeli obiektów. Techniki te są powolne i zazwyczaj nie nadają się do interakcji [13, 56, 7, 23]. Techniki interaktywne posługują się innymi niż objętościowa reprezentacjami obiektów, na przykład siatką w metodzie elementów brzegowych (ang. BEM – Boundary Element Method) [35] lub powierzchnią bazującą na punktach [59].

W przypadku objętościowej reprezentacji obiektu stosowane są techniki renderingu wolumetrycznego:

— konstrukcja izopowierzchni[3] metodą Marching Cubes [51] lub Marching Tetrahedrons [61] na podstawie wartości wokseli,
— konwersja zbioru brzegowych wokseli na elastyczną siatkę powierzchni z ograniczeniami (ang. Constrained Elastic Surface Nets) opisana przez S. Gibson w [27] i użyta w [96] do objętościowego renderingu ablacji,

[2] ablacja geol. (deszczowa) – zmywanie przez deszcze powierzchniowej warstwy zwietrzeliny albo gleby; geol. (lodowcowa) – topnienie lodowców wskutek działania energii słonecznej albo ciepła z głębi Ziemi *(źródło: Słownik wyrazów obcych i zwrotów obcojęzycznych Władysława Kopalińskiego)*.

[3] izopowierzchnia – trójwymiarowy analog izokonturu; powierzchnia reprezentująca punkty o tej samej wartości w przestrzeni.

— techniki oparte na cząsteczkach, na przykład technika Particle Splatting[4] użyta przez M. Carlsona [7] do wizualizacji materiałów lepkich i ich topnienia oraz zastosowana przez B. Adamsa [1] do wizualizacji topnienia i cieczy,
— objętościowe rzucanie promieni (ang. Volume Ray Casting) użyte przez autora [87] do renderingu ablacji i sublimacji.

Reprezentacja objętościowa obiektu pozwala na dogodną implementację podejścia fizycznego do wizualizacji zjawisk, na przykład rozchodzenia się ciepła w materiale, czy badania obiektów składających się z materiałów o różnych właściwościach.

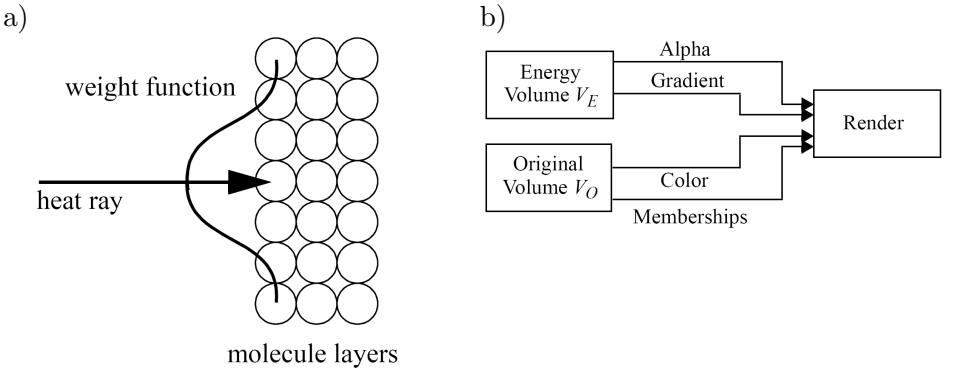
Część z wymienionych technik objętościowych zapewnia czasy wykonania zbliżone do interaktywnych w związku z czym techniki te mogą być przydatne do wizualizacji topnienia lub sublimacji ciał sztywnych.

Metoda zaproponowana przez H. Varadhana i K. Muellera w [96] opiera się na fizycznym modelu sublimacji. Zakłada, że topiony materiał paruje do niewidzialnego gazu, a więc nie zajmuje się wizualizacją zachowania cieczy. Dodatkowo, zaimplementowany w niej model rozchodzenia ciepła w obiekcie pozwala na symulację przydatną nawet w zastosowaniach medycznych. W każdym kroku symulacji:
— skierowane źródło ciepła działa na materiał z zadanego punktu przestrzeni (rys. 1.3a),
— energia cieplna jest propagowana przez materiał dopóki nie rozproszy się całkowicie – cała rozproszona energia jest absorbowana przez materiał,
— gdy zaabsorbowana w materiale energia jest większa niż ciepło przemiany fazowej, wtedy materiał zmienia stan skupienia.

Rendering odbywa się na bazie dwóch osobnych zbiorów danych objętościowych. Pierwszy, to oryginalny obraz wolumetryczny topionego obiektu

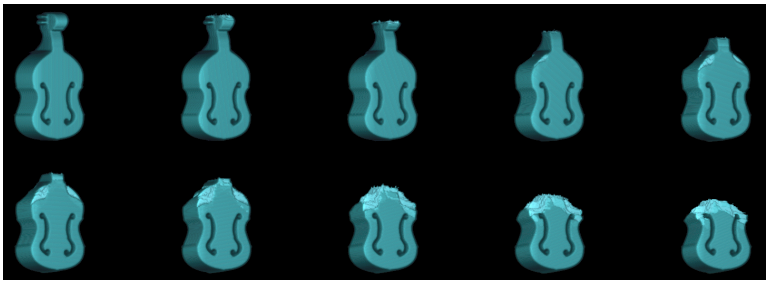
[4]Particle Splatting – ogólna nazwa zbioru algorytmów wizualizacji bazujących na renderingu elementów powierzchni obiektu z małych elementów, na przykład z wykorzystaniem ray tracingu [7] lub renderingu i blendingu sfer [1].

(V_O), który dostarcza do potoku renderingu dane o kolorze i przynależności danej molekuły, dzięki czemu można topić obiekty składające się z różnych materiałów. Drugi zbiór danych (V_E) przechowuje informację o energiach molekuł, o gęstości materiału i przezroczystości. Obliczenia związane z procesem topnienia odbywają się w V_E natomiast rendering korzysta z obu zbiorów (rys. 1.3b).



Rysunek 1.3. Idea wolumetrycznego renderingu ablacji: a) rozkład energii cieplnej pochodzącej z promienia na powierzchni materiału, b) źródła danych dla renderingu objętościowego (*źródło: [96]*).

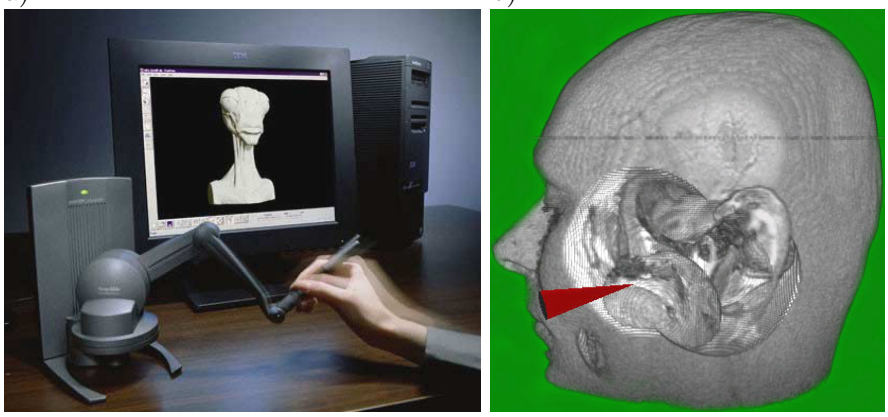
Efekty renderingu zjawiska sublimacji przedstawione są na rysunku 1.4.



Rysunek 1.4. Topnienie wiolonczeli z góry (*źródło: [96]*).

Kolejna technika objętościowa przydatna w aplikacjach interaktywnych została opracowana przez H. Chena [11]. Utworzono interfejs sprzętowy pozwalający na odczuwanie przez użytkownika powierzchni obrabianego materiału i wirtualne narzędzia do jego obróbki. Topnienie, palenie, tłoczenie i inne czynności, które pozwalają na rzeźbienie w rzeczywistości wirtualnej działają jako operatory na zbiorze wokseli składających się na obrabiany obiekt. Operatory te działają odpowiednio szybko, gdyż aplikacja z założenia jest interaktywna. Rysunek 1.5 pokazuje interfejs sprzętowy oraz topienie modelu przy użyciu opracowanego systemu.

a) b)



Rysunek 1.5. Rzeźbienie w wirtualnej objętości: a) interfejs sprzętowy do wirtualnego rzeźbienia, b) topienie modelu głowy (*źródło: [11]*).

Metoda opracowana przez M. Carlsona [7] pozwala na animację materiałów, które topią się, płyną i zestalają się (krzepną) – na przykład świece, lawa, czy cement (rys. 1.6). Opiera się na fizycznej animacji cieczy. Ciała stałe są traktowane jako ciecz o bardzo dużej lepkości. Do symulacji procesu topnienia lub krzepnięcia została tu zmodyfikowana metoda Marker-and-Cell (MAC), która pierwotnie służyła do symulacji zachowania cieczy. MAC opiera się na równaniach Naviera-Stokesa dla płynów. Ilość płynu dopływającego do punktu jest równa ilości płynu odpływającego (1.1):

$$\nabla \cdot \vec{u} = 0, \tag{1.1}$$

gdzie: $\vec{u}$ – prędkość płynu.

Natomiast chwilowa zmiana prędkości płynu w danym punkcie dana jest równaniem (1.2):

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u} \cdot \nabla)\vec{u} + \nabla \cdot (\nu \nabla \vec{u}) - \frac{1}{\rho}\nabla p + f, \tag{1.2}$$

gdzie: p – ciśnienie, ρ – gęstość (tutaj $= 1$), ν – lepkość.

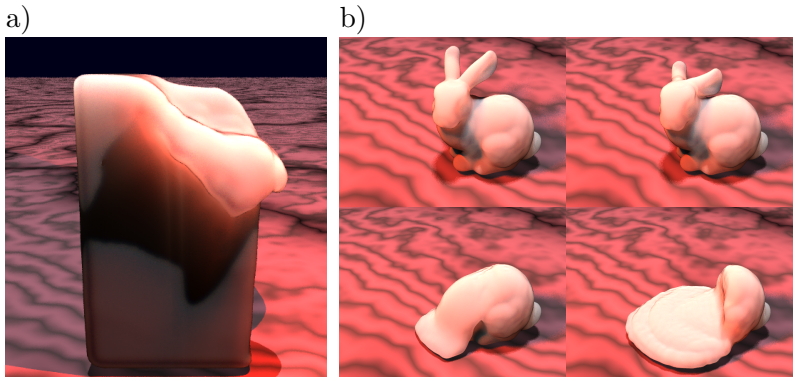
Na chwilową zmianę prędkości płynu w danym punkcie składa się:

— konwekcja – określa kierunek w jakim otaczający płyn przesuwa dany punkt,

— bezwładność dyfuzji – wpływa na szybkość tłumienia zmiany prędkości wokół punktu,

— ciśnienie – pokazuje jak mała cząstka płynu jest pchana w kierunku od wysokiego, do niskiego ciśnienia,

— siły zewnętrzne – na przykład grawitacja, albo nacisk na płyn.

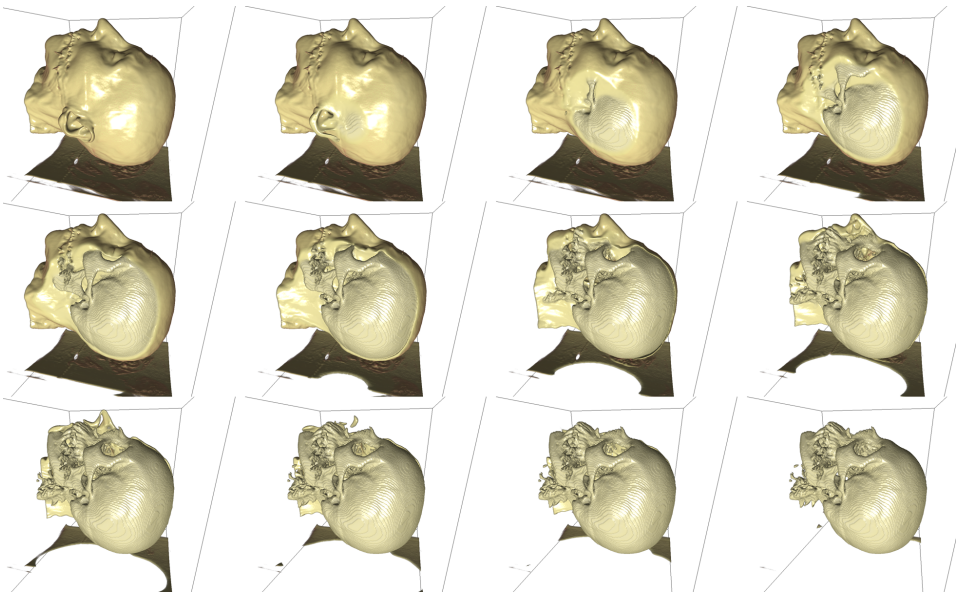


Rysunek 1.6. Ciała topiące się i krzepnące zamodelowane jako ciecz o zadanej lepkości: a) topnienie bloku wosku, b) topnienie woskowego modelu Stanford Bunny (*źródło: [7]*).

Metoda MAC opiera się na dwóch zestawach danych: podziale przestrzeni na prostopadłościenne komórki („Cell"), w których ustalana jest prędkość płynu oraz dużej liczbie cząsteczek płynu służących do oznaczania („Mark"), które komórki są wypełnione płynem blisko jego powierzchni

(interfejs powietrze-płyn). Przestrzeń podzielona na prostopadłościenne komórki jest typowym modelem objętościowym. Do jego renderingu użyta jest metoda Particle Splatting. Każda komórka zostaje podzielona na 64 woksele, których wypełnienie zależy od rozkładu cząsteczek zawartych w ich środku. Aby zlikwidować nierówności powierzchni wynikające z reprezentacji wokselowej używany jest filtr niskoprzepustowy, który wytraca małe szczegóły na powierzchni obiektu (rys. 1.6).

Objętościowy rendering ablacji [87] używający objętościowy Ray Casting zawdzięcza odpowiednio dużą szybkość obliczeń algorytmom implementowanym na GPU[5] zamiast CPU [49] (rys. 1.7).



Rysunek 1.7. Ablacja ciała z modelu głowy (*źródło: opracowanie własne [87]*).

Wymienione metody mają wspomniane wcześniej wady. Są czasochłonne i potrzebują znacznej pamięci na przechowywanie danych modelu. Zazwyczaj bazują na reprezentacji objętościowej, która wymaga specjalnego

[5] ang. Graphics Processing Unit – procesor graficzny, jednostka obliczeniowa znajdująca się w kartach graficznych.

traktowania podczas wizualizacji, na przykład rozwiązania problemu integracji obiektów reprezentowanych objętościowo ze sceną opartą o model powierzchniowy, rzucania cieni, czy przesłaniania. Choć do rozwiązania niektórych problemów opracowano algorytmy przeznaczone do zaprogramowania na GPU [15, 49, 75, 31], a więc działające w czasie wystarczającym do interakcji, to nie włączano ich do prac dotyczących wizualizacji przemian fazowych i deformacji materiałów.

Warto zauważyć, że przyspieszenie obliczeń uzyskiwane na GPU zachęciło badaczy do używania GPU także do innych obliczeń niż rendering, jakkolwiek nie są to obliczenia dotyczące przemian fazowych. Zostały one zestawione poniżej:

— chemia i biochemia: analiza oraz modelowanie cząsteczek i ich interakcji [83, 73, 91, 79],
— astronomia: analiza danych [74, 25],
— elektronika: symulacja działania układów [4, 30],
— medycyna: obróbka i analiza obrazów [58, 32, 80, 57],
— fizyka: obliczenia na potrzeby architektury gier komputerowych i pętli symulacji fizycznych [104],
— fizyka: silniki fizyczne [38],
— algorytmy ewolucyjne, programowanie genetyczne [50, 52, 100, 95, 72],
— bazy danych [18, 33],
— algorytmy korekcji błędów [17]
— data mining: klastrowanie [101, 54],
— synteza dźwięku przestrzennego (WFS) [93],
— ekstrakcja cech ruchu z sekwencji wideo w czasie rzeczywistym [82],
— symulacja pracy sieci komputerowych [103],
— symulacja ruchu pojazdów w sieci drogowej [67, 81],
— rozwiązywanie układów równań liniowych, reprezentowanych dużymi, gęstymi macierzami [24],
— szybkie algorytmy sortowania [10],
— obliczanie dyskretnej transformaty Fouriera [29, 64].

Przegląd literatury ukazał pewien trend w badaniach nad wizualizacją zjawisk topnienia i sublimacji. Zjawiska te zazwyczaj są modelowane jako przepływ ciepła we wnętrzu obiektu. Daje to poprawne pod względem fizycznym wyniki, ale wymaga do obliczeń sporo zasobów komputera, to znaczy czasu procesora i pamięci. Pociąga to za sobą nieprzydatność lub małą przydatność danej metody w aplikacjach renderingu czasu rzeczywistego, prezentacjach interaktywnych, grach komputerowych, w których dokładność z punktu widzenia fizyki nie jest tak ważna, jak krótki czas obliczeń. Stąd zainteresowanie możliwościami, jakie wynikają z właściwości najczęściej używanej w grafice trójwymiarowej powierzchniowej reprezentacji wielokątowej, której główną zaletą jest wsparcie sprzętowe. Przedstawione w niniejszej monografii rozwiązanie bazuje na empirycznym modelu zjawisk topnienia i sublimacji opartym na spostrzeżeniu, że szybkość zachodzenia tych zjawisk zależy od cech geometrycznych obiektu, a dokładniej od nieregularności ich powierzchni.

Poniżej wyspecyfikowane zostały główne założenia dla prezentowanego rozwiązania problemu wizualizacji przemian fazowych:

1. Wizualizacja ciał sztywnych.
2. Zjawiska sublimacji i topnienia oparte na modelu powierzchni międzyfazowej.
3. W czasie trwania zjawiska bryły mogą się dzielić na części, ale nie powstają w nich otwory.
4. Rendering czasu rzeczywistego.
5. Algorytmy uruchamiane na CPU i GPU.

Ad 1. Poszukiwane jest rozwiązanie problemu wizualizacji materiałów, dla których granica między fazami jest ostra, na przykład lód-woda. A jednocześnie granica ta nie przesuwa się z powodów innych, niż zmiana stanu skiupienia. Nie zakłada się możliwości wizualizacji przemian odwracalnych

mimo, iż w temperaturze krzepnięcia topnienie i krzepnięcie zachodzi równocześnie, a końcowy stan ciała zależy od tego, który proces ma przewagę [5]. Dla przypadku topnienia została zaproponowana wizualizacja cieczy pod topionym obiektem.

Ad 2. Zaproponowane struktury danych i algorytmy oparte zostaną na wielokątowej reprezentacji powierzchni międzyfazowej.

Ad 3. Reprezentacja objętościowa ma tę przewagę nad dowolną powierzchniową, że tworzenie otworów w topionym lub sublimującym obiekcie jest jej oczywistą cechą. Dla reprezentacji powierzchniowej zagadnienie to jest bardziej skomplikowane i może wymagać algorytmów o dużej złożoności obliczeniowej potrzebnej na wykrycie samoprzecięć siatki. W wyniku ich wykrycia, w miejscach, w których występują, powinny pojawić się otwory w obiekcie. Ze względu na założenie renderingu czasu rzeczywistego (zał. **4.**) praca obejmuje jedynie podział obiektu na mniejsze części na skutek trwania zjawiska sublimacji lub topnienia. Wtedy podejście powierzchniowe ma przewagę nad objętościowym, pozwalając na przykład na łatwe wykrycie faktu podziału obiektu dzięki zastosowaniu charakterystyki Eulera [36].

Ad 4. Wybór powierzchniowego modelu przemian fazowych podyktowany jest szansą na uzyskanie szybkości obliczeń akceptowalnej w aplikacjach renderingu czasu rzeczywistego[6]. Obecnie oznacza to poszukiwanie algorytmów przyspieszających obliczenia niezbędne do wygenerowania kolejnej ramki wizualizacji obiektu.

Ad 5. Proces wizualizacji przemian fazowych posiada dwa główne etapy. Zadaniem etapu pierwszego jest utworzenie aktualnej siatki geometrii bryły, natomiast zadaniem etapu drugiego jest jej rendering. Od kilku lat karty graficzne umożliwiają zaprogramowanie własnego potoku renderingu, co umożliwia rendering w czasie rzeczywistym obrazów o cechach fotorealistycznych. Dlatego etap renderingu zaplanowany został na GPU.

[6] Rendering czasu rzeczywistego – technika pozwalająca otrzymać kolejną ramkę wizualizacji w czasie zapewniającym odbiorcy komfort odbioru obrazu (brak „cięcia" animacji).

Ze względu na obserwowany trend do przenoszenia także innych obliczeń na kartę graficzną, algorytmy tworzenia geometrii dla przemian fazowych opracowane zostały w wersjach na CPU i na GPU. Umożliwia to dokonanie porównań obu sposobów obliczeń.

Niniejsza monografia zawiera zbiór algorytmów pozwalających na wizualizację zjawisk topnienia i sublimacji obiektów reprezentowanych powierzchniowo. Przebieg tych zjawisk może być kontrolowany poprzez dobór wartości współczynników. Szczegółowe założenia poszczególnych algorytmów zawarte są w rozdziałach je opisujących.

1.3 Zawartość niniejszej monografii

Monografia została podzielona na dziewięć rozdziałów.

Dwa rozdziały poświęcono omówieniu zagadnień związanych z reprezentowaniem geometrii obiektów trójwymiarowych i przetwarzaniu geometrii dla wizualizacji sublimacji obiektów.

Rozdział drugi opisuje stosowane w grafice komputerowej sposoby reprezentacji brył, ich zastosowania oraz cechy charakterystyczne. Szczególną uwagę poświęcono siatkom wielokątowym, porównaniu ich zalet i wad. Przedstawione zostały zagadnienia związane z poprawnością, jakością i atrybutami siatki trójkątów w reprezentacji powierzchniowej. Następnie omówiono klasyczny i programowalny potok renderingu siatek trójkątów.

Rozdział trzeci opisuje metodę deformacji siatki wielokątowej modelującej powierzchnię międzyfazową w wizualizacji topnienia i sublimacji. Metoda opiera się na empirycznym modelu zjawisk, w którym przebieg zjawiska deformacji siatki uwarunkowany jest nieregularnością powierzchni. Ponadto zaprezentowane zostały:

— sposób upraszczania siatki, tam gdzie jest to konieczne,
— zagadnienie podziału obiektu na części,
— uwzględnianie czynników fizycznych wpływających na topnienie lub sublimację,

— szczegółowa reprezentacja geometrii sublimujących obiektów,
— algorytmy powstałe na bazie opracowanych metod.

Kolejne dwa rozdziały prezentują zagadnienia programowania zadań ogólnego przeznaczenia na jednostkach graficznych.

Rozdział czwarty przedstawia możliwości współczesnych procesorów graficznych w zakresie ich programowania. Omówiony jest programowalny potok renderingu: programy cieniowania wierzchołków, geometrii i fragmentów (pikseli), ich możliwe zastosowania, warunki wykonania, ograniczenia oraz uruchamianie. Druga część rozdziału prezentuje koncepcje oraz możliwości zastosowania GPU do obliczeń ogólnego przeznaczenia. Przedstawia typowe operacje przeprowadzane za pomocą GPU, zasady przechowywania zbiorów danych w teksturach, przetwarzanie strumieniowe i rendering do pamięci. Rozdział podaje również przykład mapowania algorytmu na GPU.

Rozdział piąty omawia przystosowanie algorytmów zaproponowanych w rozdziale trzecim do uruchamiania ich na GPU: sposób przechowywania reprezentacji indeksowej w teksturach, wprowadzone przez autora operacje strumieniowe i mapowanie algorytmów przy ich pomocy.

Rozdział szósty przedstawia model materiału wieloskładnikowego lodu, jako przykładu materiału ulegającego topnieniu. Omówione zostało użycie programowalnego potoku renderingu do implementacji wszystkich potrzebnych składników materiału.

Rozdział siódmy opisuje, w jaki sposób do wizualizacji sublimacji dodać wizualizację fazy ciekłej, aby tą drogą uzyskać wizualizację topnienia. Zaproponowano algorytmy wizualizacji i animacji fazy ciekłej na GPU.

W rozdziale ósmym przedstawiono metodę weryfikacji algorytmów wizualizacji poprzez pomiar podobieństwa konturów.

Rozdział dziewiąty jest podsumowaniem przedstawionych rozwiązań.

1.4 Wykaz używanych oznaczeń i skrótów

Spis najważniejszych oznaczeń używanych w tekście, we wzorach oraz algorytmach:
k_e – współczynnik szybkości przesunięcia wierzchołka wzdłuż sumy wektorów krawędzi
k_g – globalny współczynnik szybkości przesuwania powierzchni międzyfazowej
$k_l()$ – lokalny współczynnik (funkcja) szybkości przesuwania powierzchni międzyfazowej
k_n – współczynnik szybkości przesunięcia wierzchołka wzdłuż normalnej
k_r – długość krawędzi siatki poniżej której następuje jej usunięcie
M – wektor przesunięcia wierzchołka w algorytmie deformacji siatki
M_E – wektor przesunięcia wierzchołka obliczony z krawędzi wychodzących z wierzchołka
M_N – wektor przesunięcia wierzchołka obliczony z normalnej wierzchołka
N – wektor normalny w wierzchołku siatki wielokątowej
N_f – wektor normalny trójkąta w siatce wielokątowej

Spis skrótów:
BSP (ang. Binary Space Partitioning) – binarny podział przestrzeni
CAD (ang. Computer-Aided Design) – projektowanie wspomagane komputerowo
Cg (ang. C for graphics) – język programowania shaderów (opracowany przez firmę nVidia)
CNC (ang. Computer Numerical Controlled machine tools) – obrabiarki sterowane numerycznie
CPU (ang. Central Processing Unit) – procesor główny komputera
CSG (ang. Constructive Solid Geometry) – konstruktywna geometria brył
CUDA (ang. Compute Unified Device Architecture) – zunifikowana architektura urządzeń liczących

GLSL (ang. OpenGL Shading Language) – język programowania shaderów dla biblioteki graficznej OpenGL
GPU (ang. Graphics Processing Unit) – procesor graficzny, jednostka obliczeniowa znajdująca się w kartach graficznych
GPGPU (ang. General-Purpose computation on Graphics Processing Units) – programowanie zadań ogólnego przeznaczenia z wykorzystaniem GPU
GS (ang. Geometry Shader) – program cieniowania geometrii
HLSL (ang. High Level Shader Language) – język programowania shaderów wysokiego poziomu (dla biblioteki graficznej DirectX)
IA (ang. Input Assembler) – asembler danych wejściowych
MES – metoda elementów skończonych
MRT (ang. Multi Render Target) – wielokrotny cel renderingu
NURBS (ang. Non-Uniform Rational B-Spline) – niejednorodna ułamkowa krzywa B-sklejana
OM (ang. Output Merger) – etap łączenia wyjścia
PC (ang. Personal Computer) – komputer osobisty (typu desktop lub mobilny)
PS (ang. Pixel Shader) – program cieniowania fragmentów (pikseli)
RGB (ang. Red, Green, Blue) – czerwony, zielony, niebieski
ROP (ang. Raster Operations) – operacje rastrowe
RS (ang. Rasterize Stage) – etap rasteryzacji
RT (ang. Render Target) – cel renderingu
SIMD (ang. Single Instruction, Multiple Data) – jedna instrukcja, wiele danych
SM (ang. Shader Model) – model cieniowania
SO (ang. Stream Output) – wyjście strumieniowe
VS (ang. Vertex Shader) – program cieniowania wierzchołków

2

Reprezentacje geometrii obiektów trójwymiarowych

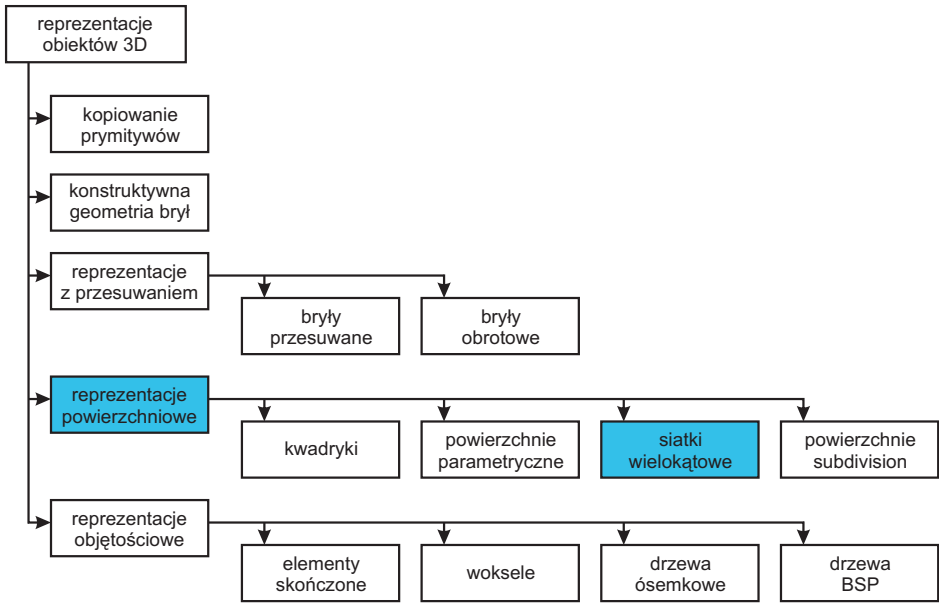
Reprezentacje geometrii obiektów trójwymiarowych ewoluowały wraz z coraz to nowymi możliwościami przechowywania danych oraz ich wykorzystania, między innymi do prezentacji graficznej. Mając na uwadze model powierzchni międzyfazowej niniejszy rozdział po przedstawieniu klasyfikacji reprezentacji obiektów skupia się na opisie siatek trójkątów, ich budowie, strukturach danych i zagadnieniach ich renderingu jako potencjalnie najlepszych w tym zastosowaniu.

2.1 Klasyfikacja reprezentacji obiektów trójwymiarowych

Do reprezentacji obiektów trójwymiarowych na potrzeby modelowania i wizualizacji używa się wielu bardzo różniących się między sobą reprezentacji. Wybór reprezentacji zdeterminowany jest:

— konkretnym zastosowaniem danego modelu (np. projektowanie inżynierskie, medycyna, modelowanie),

— technikami renderingu (np. śledzenie promieni, rendering wielokątowy),
— ewentualnym jego przetwarzaniem (np. deformacje obiektu, przebieg procesów wewnątrz obiektu),
— pożądanym czasem przetwarzania i dostępnymi zasobami komputera (np. rendering czasu rzeczywistego, rendering offline na potrzeby filmu).



Rysunek 2.1. Klasyfikacja reprezentacji obiektów trójwymiarowych.

Rysunek 2.1 przedstawia główne reprezentacje obiektów trójwymiarowych używane w modelowaniu i wizualizacji. Opisy różnych reprezentacji można znaleźć między innymi w [21, 97, 36].

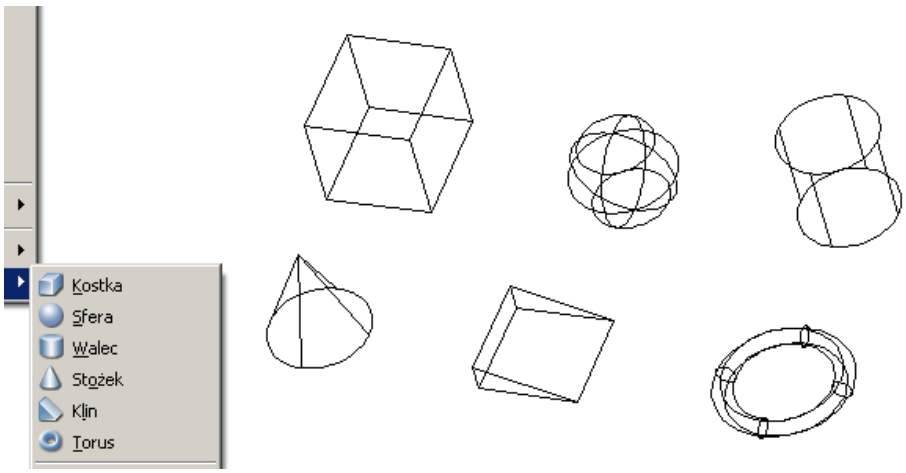
Koncepcja **kopiowania prymitywów** (ang. primitive instancing) jest używana w każdym programie do modelowania dwu- i trójwymiarowego w tym w CAD[1]. System definiuje zbiór sparametryzowanych prymitywów

[1]ang. Computer-Aided Design – projektowanie wspomagane komputerowo.

(np. prostopadłościan, walec, sfera, stożek) lub bardziej złożonych (np. śruba, koło zębate, nakrętka). Każdy prymityw ma w systemie algorytm rysowania, do którego użytkownik dostarcza wartości parametrów by uzyskać konkretne atrybuty obiektu lub jego transformacje: translacje, rotacje czy skalowanie. Na przykład dla sześcianu, który w systemie wbudowany jest jako sześcian jednostkowy `Cube`, stworzenie jego instancji, która znajdzie się w zadanym punkcie przestrzeni (x, y, z), z orientacją zadaną trzema kątami (θ, ϕ, ψ) i o skali zadanej współczynnikami (s_x, s_y, s_z) wymaga następującego przekształcenia:

$$\texttt{InstantiatedCube} = \texttt{Translate}(x, y, z) * \texttt{Rotate}(\theta, \phi, \psi) * \texttt{Scale}(s_x, s_y, s_z) * \texttt{Cube}$$

Zaletą tego rozwiązania jest powtarzalność i precyzyjne określenie kształtu bryły za pomocą oczywistych dla użytkownika parametrów. Wynika stąd przydatność tej reprezentacji w projektowaniu inżynierskim i modelowaniu (rys. 2.2). Natomiast wadą jest konieczność opracowywania nowego algorytmu rysowania, jeśli zachodzi konieczność wprowadzenia nowego obiektu do systemu.



Rysunek 2.2. Prymitywy trójwymiarowe (prostopadłościan, sfera, walec, stożek, graniastosłup i torus) dostępne w aplikacji AutoCAD.

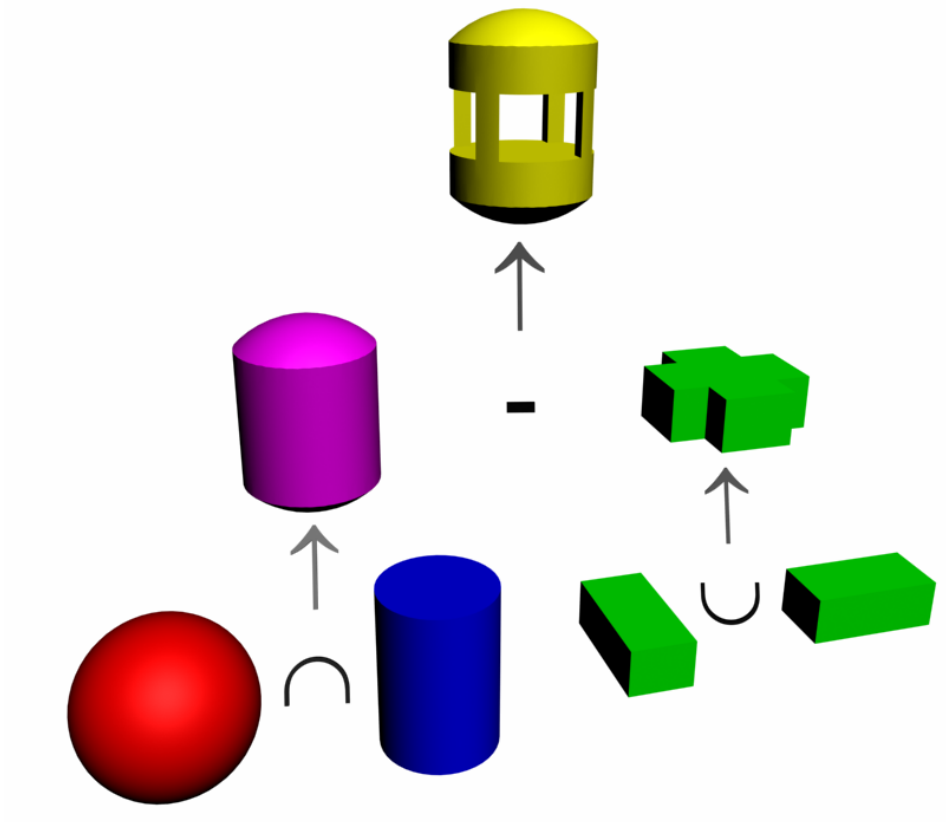
Więcej możliwości modelowania niż kopiowanie prymitywów dostarcza **konstruktywna geometria brył** (ang. CSG – Constructive Solid Geometry). Zdefiniowane w systemie prymitywy mogą być poddawane transformacjom i łączone regularyzowanymi operatorami boolowskimi: suma ($\cup^*$), część wspólna ($\cap^*$) i różnica ($-^*$) (rys. 2.3). Operatory te gwarantują, że wynikiem działania na dwóch bryłach będzie nowa bryła. Jest to ważne w aplikacjach inżynierskich i modelowaniu, gdzie istotna jest dokładność matematyczna. Za pomocą CSG można uzyskać złożone kształty bazujące na stosunkowo prostych obiektach podstawowych. Dzięki temu ręczne modelowanie jest bardzo proste: polega na doborze prymitywu, jego transformacji i sposobu połączenia z innymi. Również bardzo łatwo jest rozstrzygnąć przynależność punktu w przestrzeni do danej bryły CSG: wystarczy wyniki testów dla zawierania w prymitywach składowych połączyć tymi samymi operatorami boolowskimi, co bryły. Można tego mechanizmu użyć do szybkiej detekcji kolizji w systemach interaktywnych.

W **reprezentacjach z przesuwaniem** (ang. sweep representations) jest udostępniona grupa operacji, które przesuwają („omiatają") dwuwymiarowe kształty w celu uzyskania obiektów trójwymiarowych. Dwuwymiarowy kształt może być (rys. 2.4):

— wyciągnięty wzdłuż odcinka (ang. extrude),
— wyciągnięty wzdłuż profilu (ang. extrude rail),
— obrócony wokół osi (ang. lathe),
— obrócony po ścieżce (ang. rail revolve).

Innym rodzajem reprezentacji z przesuwaniem jest rozpinanie powierzchni między krzywymi granicznymi (ang. loft).

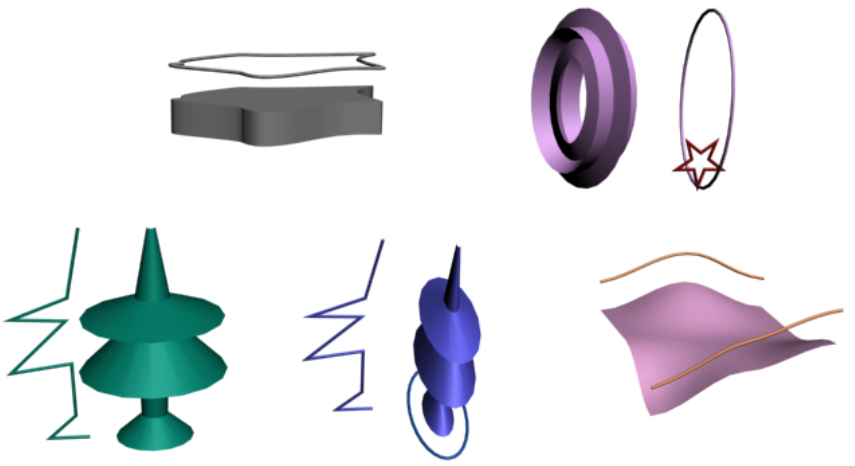
Reprezentacja z przesuwaniem jest przydatna przy projektowaniu kształtów elementów przeznaczonych do wykonania przez obrabiarki. Na przykład dla wycinarek, gdy wykrojnik będący reprezentacją przekroju elementu służy do wycięcia kształtu (extrude) lub tokarek, których zasada działania opiera się na ruchu głowicy tnącej po zaplanowanej ścieżce (lathe). Wadą



Rysunek 2.3. Konstrukcja brył CSG: obiekt końcowy powstał w programie 3ds Max jako wynik odejmowania części wspólnej sfery i walca oraz sumy dwóch prostopadłościanów.

tej reprezentacji jest możliwość uzyskania nieprawidłowych kształtów tworzonej powierzchni, na przykład samoprzecinających się. W praktyce aplikacje modelujące pozwalają stworzyć obiekt sweep, a następnie konwertują go na inną reprezentację, jeśli ma być dalej przetwarzany.

W **reprezentacji powierzchniowej** (brzegowej, ang. B-rep) bryła jest reprezentowana przez zbiór połączonych elementów powierzchni. Powierzchnia jest granicą między bryłą a resztą przestrzeni. Istnieje kilka rodzajów reprezentacji powierzchniowej: kwadryki, powierzchnie parametryczne, siatki wielokątowe, powierzchnie subdivision.

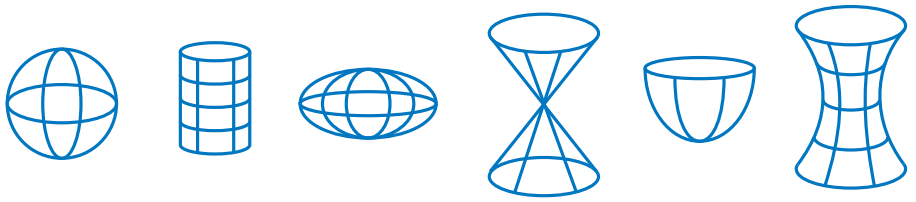


Rysunek 2.4. Reprezentacje z przesuwaniem.

Kwadryki są powierzchniami drugiego stopnia (2.1):

$$f(x,y,z) = a{\cdot}x^2{+}b{\cdot}y^2{+}c{\cdot}z^2{+}2d{\cdot}xy{+}2e{\cdot}yz{+}2f{\cdot}xz{+}2g{\cdot}x{+}2h{\cdot}y{+}2j{\cdot}z{+}k = 0. \tag{2.1}$$

W zależności od wartości współczynników $(a, b, c..., h, j, k)$ mogą reprezentować sferę, walec, elipsoidę, stożek, paraboloidę, hiperboloidę (rys. 2.5). Cechuje je łatwość wyznaczania normalnej do powierzchni, testowania przynależności punktu do powierzchni oraz obliczania przecięcia powierzchni z promieniem. Ta ostatnia cecha spowodowała, że kwadryki są chętnie używane w systemach śledzenia promieni. Wadą kwadryki jest trudność łączenia powierzchni z segmentów i brak lokalnej kontroli nad kształtem powierzchni.

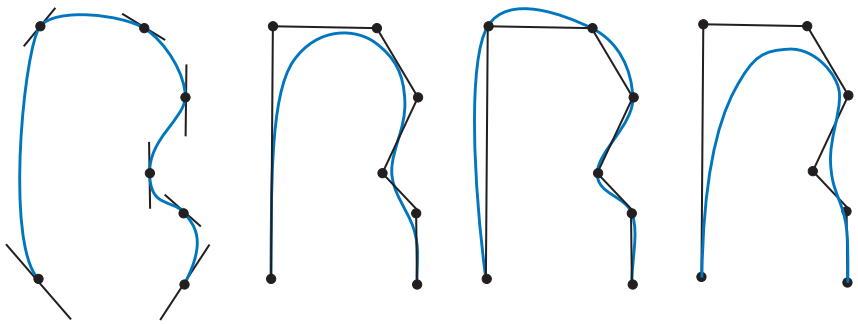


Rysunek 2.5. Kwadryki: sfera, walec, elipsoida, stożek, paraboloida, hiperboloida

Powierzchnie parametryczne posiadają zalety kwadryk, ale nie przejawiają ich wad. Ogólne podejście polega na tym, aby używać funkcji wyższego stopnia niż 2. Najczęściej używa się parametrycznych powierzchni bikubicznych, które są uogólnieniem parametrycznych krzywych trzeciego stopnia (rys. 2.6). Powierzchnia bryły jest w tym przypadku reprezentowana przez siatkę płatów, inaczej nazywanych łatami (ang. patch), które przestawia się parametrycznie. Równanie 2.2 służy do parametryzowania płatów (rys. 2.7):

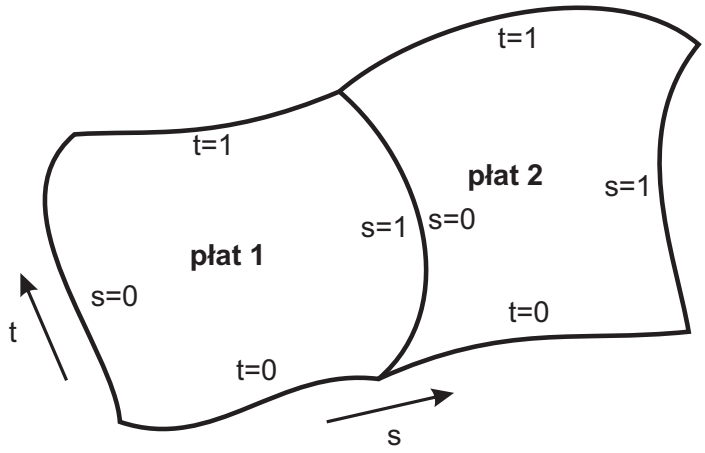
$$Q(s,t) = (x(s,t), y(s,t), z(s,t)), \tag{2.2}$$

gdzie $x(), y(), z()$ są ciągłymi funkcjami zmiennych $s \in < 0, 1 >, t \in < 0, 1 >$.



Rysunek 2.6. Krzywa Beziera, krzywa B-sklejana, krzywa typu Cardinal oraz NURBS.

Każdy płat rozpięty jest na punktach kontrolnych i może być sterowany za pomocą prowadnic określonych przez wektory styczne. Programy graficzne często udostępniają powierzchnie Beziera oraz powierzchnie NURBS (ang. Non-Uniform Rational B-Spline – niejednorodna ułamkowa B-sklejana). Zaletą tych pierwszych jest łatwość modelowania przy pomocy wektorów stycznych oraz niezmienniczość względem trzech podstawowych transformacji: przesunięcia, obrotu i skalowania. Zaletą NURBS natomiast jest niezmienniczość nie tylko względem przesunięcia, obrotu i skalowania, ale także względem perspektywy. Przekształcenie perspektywiczne wystarczy wtedy stosować tylko do punktów kontrolnych powierzchni

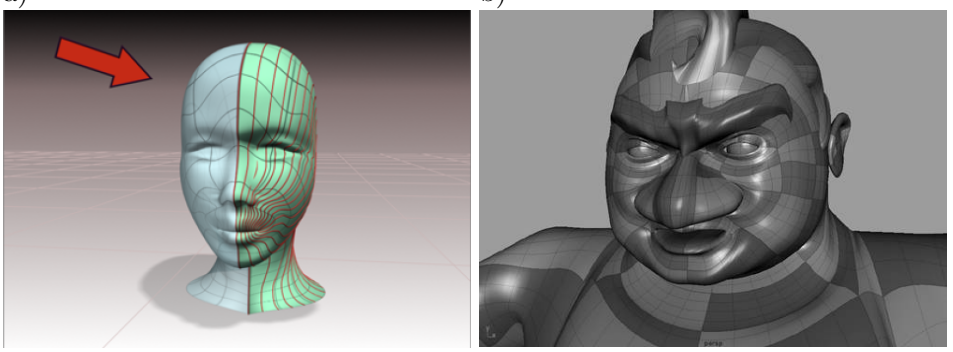


Rysunek 2.7. Powierzchnia składająca się z dwóch płatów.

NURBS. Niestety, powierzchnie te mogą być problematyczne przy modelowaniu, ponieważ ze względu na kwadratowy kształt płata wymagają sporej wyobraźni i planowania przekształceń przed rozpoczęciem modelowania (rys. 2.8a). Błędy wynikające ze złego przyporządkowania fragmentów powierzchni NURBS do końcowego kształtu są zazwyczaj nieusuwalne i kończą się powtórnym modelowaniem. Dla większej elastyczności w budowaniu kształtu można tworzyć go z wielu płatów NURBS (rys. 2.8b). Wtedy mogą pojawić się problemy z ciągłością powierzchni na brzegach płatów, ponieważ położenie punktów kontrolnych NURBS ma większy zasięg niż w przypadku płatów Beziera, gdzie każdy segment krzywej jest kontrolowany lokalnie przez swoje własne punkty kontrolne. Ponadto modelowanie w czasie rzeczywistym wymaga sporej liczby obliczeń i nie wszystkie aplikacje wykonują to sprawnie.

Główną zaletą powierzchni parametrycznych jest możliwość reprezentacji dowolnych gładkich kształtów, łatwe wyznaczanie normalnych oraz krzywizny dla każdego punktu powierzchni. Wszelkie prace projektowe, dla których nie wystarcza kopiowanie prymitywów, CSG czy obiekty sweep są możliwe za pomocą reprezentacji powierzchniami parametrycznymi. Również ta reprezentacja zapewnia ścisły opis analityczny powierzchni.

a) b)



Rysunek 2.8. Model głowy utworzony z płatów NURBS: a) jednego; b) wielu [53].

Obiekty o kształtach zaprojektowanych za pomocą powierzchni parametrycznych, dzięki numerycznemu opisowi mogą być wytworzone bez żadnych dodatkowych konwersji przez obrabiarki numeryczne CNC[2] [97]. Wybrane kwadryki, powierzchnie Beziera i powierzchnie NURBS są dostępne w bibliotekach graficznych, na przykład w OpenGL.

Dzięki coraz większym możliwościom obliczeniowym komputerów i rozwojowi algorytmów konwersji na siatkę wielokątową powierzchnie parametryczne stają się coraz popularniejszą reprezentacją używaną w grafice trójwymiarowej. W liczbie aplikacji ustępują jedynie właśnie **siatkom wielokątowym**, które są najpopularniejszą reprezentacją brzegową.

Jako siatki wielokątowej najczęściej używa się **siatki trójkątów** lub czworokątów. Cechy, możliwości, strukturę i podstawowe operacje na siatce wielokątowej opisują następne podrozdziały.

Swoją popularność siatki trójkątów zawdzięczają faktowi, że grafika trójwymiarowa od samego początku preferuje tę reprezentację. Od początku

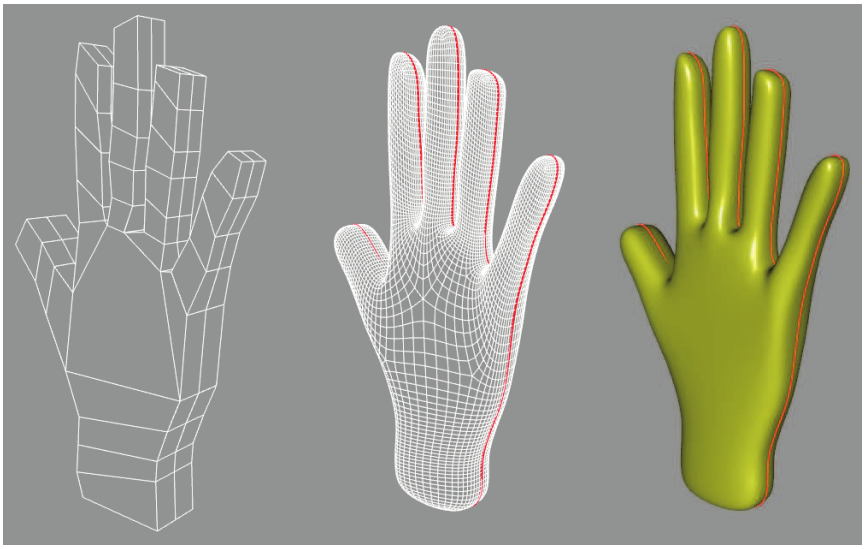
[2]CNC – ang. Computer Numerical Controlled (machine tools) – obrabiarki sterowane numerycznie bezpośrednio przez załadowany do nich program stworzony przez aplikację do projektowania

wspomagał je sprzęt dedykowany tworzeniu grafiki trójwymiarowej (np. akcelerator Voodoo firmy 3Dfx Interactive). Algorytmy renderingu i przetwarzania danych graficznych zazwyczaj pracują na obiektach o takiej reprezentacji. Dostępne są dla niej algorytmy przeprowadzania operacji boolowskich, upraszczania, wygładzania, detekcji kolizji. Również silniki fizyczne (np. Havok) modelujące dynamikę bryły sztywnej pozwalają na użycie w obliczeniach siatek wielokątowych.

Ponadto formaty plików grafiki 3D w ten sposób przechowują struktury reprezentujące obiekty: .x, .fbx dla gier komputerowych, .3ds, .ase, .obj, .vrml, .ply, Collada dla aplikacji grafiki komputerowej, czy .dxf dla aplikacji wspomagających projektowanie.

Podstawową wadą tej reprezentacji jest jej charakter aproksymacyjny – przy pomocy wielokątów można tylko przybliżać kształt wielu obiektów. Prace projektowe na siatce trójkątów są rzadko spotykane, ponieważ wymagają testów spójności siatek reprezentujących bryły i innych zabiegów związanych z niedokładnością reprezentacji. Również ręczne modelowanie nie jest intuicyjne, gdyż bezpośrednia manipulacja wiąże się tylko z lokalnymi operacjami na wybranych wierzchołkach, krawędziach i trójkątach. Niektóre aplikacje graficzne (np. 3ds Max) łagodzą ten problem pozwalając edytować siatki za pomocą tak zwanych modyfikatorów, które umożliwiają parametryzowanie nielokalnych operacji na siatkach grupując na przykład wierzchołki do wspólnych przekształceń.

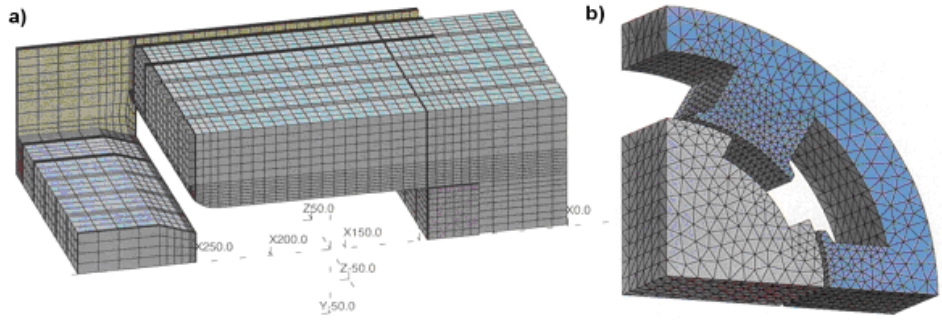
Powierzchnie subdivision są sposobem budowania gładkich powierzchni z siatek wielokątowych. Wejściową siatką jest tak zwana siatka kontrolna. Natomiast kształt powierzchni wyjściowej jest wynikiem rekursywnego procesu podziału składowych wielokątów na mniejsze, które lepiej aproksymują modelowaną powierzchnię (rys. 2.9). Podział składowych wielokątów odbywa się poprzez stworzenie nowych wierzchołków, a przez to i wielokątów. Pozycje nowych wierzchołków są obliczane na podstawie sąsiednich, już istniejących. Położenia istniejących wierzchołków są zmieniane lub nie, zależnie od tego, czy do podziału został zastosowany algorytm z rodziny



Rysunek 2.9. Modelowanie subdivision dłoni: siatka kontrolna, siatka graniczna, rendering [62].

interpolujących (np. Butterfly [14]), czy aproksymujących (np. Catmull and Clark [9], $\sqrt{3}$ [48]). Modelowanie powierzchni subdivision może odbywać się na wszystkich poziomach podziału. Po edycji siatki na danym poziomie można dokonać podziału następnego poziomu i w ten sposób zajmować się coraz mniejszymi szczegółami kształtu. Algorytmy podziału pracują na trójkątach albo czworokątach. Możliwość określenia poziomów złożoności pozwala na zastosowanie ich w aplikacjach grafiki, które używają mechanizmu LOD (ang. level of detail – poziom szczegółowości) lub transmitują dane obiektów przez sieć komputerową. Można wyświetlać wtedy transmitowaną scenę trójwymiarową, której dane jeszcze nie są kompletne.

W **reprezentacji objętościowej** bryła jest dekomponowana na zbiór sąsiadujących ze sobą, nie przecinających się brył. Dzięki temu można zamodelować nie tylko powierzchnię, ale i wnętrze bryły. W obliczeniach inżynierskich często używaną reprezentacją objętościową jest dekompozycja badanego urządzenia (czasami także jego otoczenia) na elementy skończone, które są ze sobą połączone węzłami, krawędziami lub ścianami. W celu



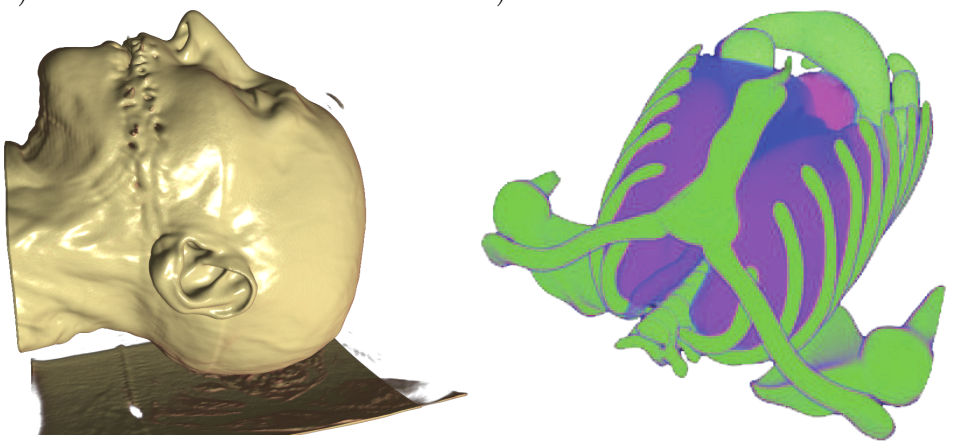
Rysunek 2.10. Siatki MES wygenerowane automatyczne przez pakiet programowy OPERA: a) model magnesu – obszarami regularna siatka zbudowana z elementów sześciennych, b) model maszyny elektrycznej – nieregularna siatka zbudowana z elementów czworościennych [106].

przeprowadzenia analizy naprężeń, temperatury lub innych parametrów metodą elementów skończonych (MES) należy wybrać odpowiedni typ elementu skończonego, na przykład sześcian, czworościan (rys. 2.10). W zależności od stosowanej techniki obliczeń wartości analizowanego parametru obliczane są dla elementów skończonych, ich węzłów, krawędzi lub ścian.

W zastosowaniach medycznych popularna jest reprezentacja wokselowa (ang. voxel – volumetric pixel) (rys. 2.11). Woksele są prostopadłościanami, na które równomiernie podzielona jest przestrzeń. Zbiór wokseli może być traktowany jako reprezentacja funkcji $R^3 \rightarrow R$ (pola skalarnego). Każdemu wokselowi przypisana jest wtedy wartość, która może reprezentować gęstość czy przynależność do określonej tkanki. Atrybuty wokseli mogą być podczas renderingu, dzięki funkcji przeniesienia (ang. transfer function), reprezentowane przez kolory dla łatwiejszej analizy obrazu (rys. 2.11b).

Reprezentacja wokselowa wymaga dużej pamięci do przechowywania danych. W aplikacjach wymagających oszczędności pamięci używane są więc podziały rekursywne, które tworząc hierarchie umożliwiają lepszą organizację użycia pamięci.

a) b)



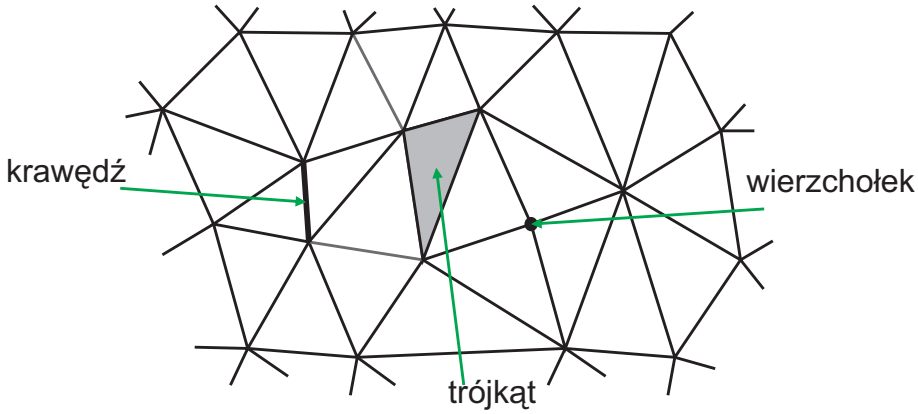
Rysunek 2.11. Wizualizacja zbioru medycznego złożonego z wokseli [87]: a) strukturę wokselową można zaobserwować na podbródku, b) efekt działania transfer function na atrybutach wokseli.

Hierarchicznym wariantem reprezentacji wokselowej są drzewa ósemkowe, które rekursywnie dzielą przestrzeń na pół w każdym z trzech wymiarów.

Z kolei hierarchicznym wariantem reprezentacji wielokątowej są drzewa BSP (binarnego podziału przestrzeni), które rekursywnie dzielą przestrzeń na pary podprzestrzeni za pomocą płaszczyzn, w których leżą wielokąty siatki. Zbudowana w ten sposób powierzchnia jest taka sama, jak siatka źródłowa.

2.2 Siatka trójkątów

Siatka wielokątów jest zbiorem wielokątów połączonych krawędziami i wierzchołkami, który aproksymuje reprezentowaną powierzchnię (rys. 2.12). Krawędź siatki jest bokiem wielokąta, a wierzchołek siatki jest końcem krawędzi, albo wierzchołkiem wielokąta. Ze względu na elementarność oraz wypukłość, w przestrzeni trójwymiarowej najczęściej stosuje się trójkąty. Siatki trójkątów są standardowymi danymi dla renderingu wspomaganego przez karty graficzne.



Rysunek 2.12. Siatka trójkątów składająca się z wierzchołków, krawędzi i trójkątów.

Istnieje kilka możliwości zbudowania struktury przechowującej dane siatki trójkątów zależnie od jej przeznaczenia, oczekiwań odnośnie obróbki i renderingu:

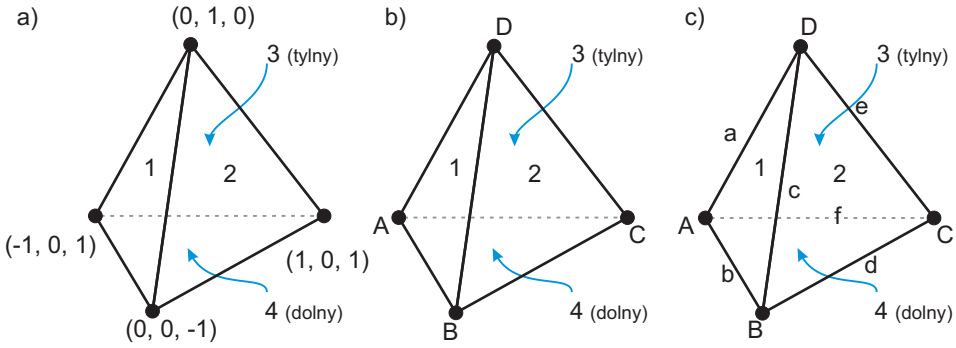
— reprezentacja bezpośrednia – siatka jest zapamiętana jako tablica trójkątów, przy czym każdy trójkąt jest określony przez trzy wierzchołki (V_1, V_2, V_3),

— reprezentacja indeksowa – przechowywane są dwie tablice: tablica wierzchołków siatki oraz tablica indeksów do wierzchołków w kolejno rysowanych trójkątach,

— reprezentacja krawędziowa – przechowywane są trzy tablice: tablica wierzchołków, tablica krawędzi zawierająca indeksy par wierzchołków określających krawędzie oraz tablica indeksów do krawędzi kolejno rysowanych trójkątów,

— reprezentacja Baumgarta [3, 111] (ang. Winged Edge) – reprezentacja zawierająca opis geometrii i topologii, w której każda z krawędzi reprezentowana jest przez zbiór wskaźników: wskaźniki na dwa wierzchołki geometryczne, wskaźniki na cztery dodatkowe krawędzie wychodzące z tych wierzchołków oraz wskaźniki na dwa trójkąty współdzielące tę krawędź.

Dane wierzchołka mogą dotyczyć tylko jego współrzędnych przestrzennych (x, y, z), lecz często są uzupełniane o kolor, normalną i współrzędne tekstury.



Rysunek 2.13. Tetrahedron w reprezentacji: a) bezpośredniej, b) indeksowej, c) krawędziowej i Baumgarta. Przyjęte zostały następujące oznaczenia elementów w reprezentacjach: wierzchołków przy pomocy wielkich liter A, B, C, D, trójkątów przy pomocy liczb 1, 2, 3, 4 oraz krawędzi przy pomocy małych liter a, b, c, d, e, f.

Rysunek 2.13 przedstawia cztery reprezentacje tetrahedronu. W strukturach danych tych reprezentacji zapisany jest on następująco:

— reprezentacja bezpośrednia:

tablica współrzędnych wierzchołków trójkątów			
1	$(0,1,0)$	$(0,0,-1)$	$(-1,0,1)$
2	$(0,1,0)$	$(1,0,1)$	$(0,0,-1)$
3	$(0,1,0)$	$(-1,0,1)$	$(1,0,1)$
4	$(0,0,-1)$	$(1,0,1)$	$(-1,0,1)$

— reprezentacja indeksowa:

tablica wierzchołków	
A	$-1,0,1$
B	$0,0,-1$
C	$1,0,1$
D	$0,1,0$

tablica trójkątów	
1	D,B,A
2	D,C,B
3	D,A,C
4	B,C,A

— reprezentacja krawędziowa:

tablica wierzchołków	
A	$-1, 0, 1$
B	$0, 0, -1$
C	$1, 0, 1$
D	$0, 1, 0$

tablica krawędzi	
a	A, D
b	A, B
c	B, D
d	B, C
e	C, D
f	A, C

tablica trójkątów	
1	a, c, b
2	c, e, d
3	a, f, e
4	b, d, f

— reprezentacja Baumgarta – tablice wierzchołków i trójkątów nie są jednoznaczne, mogą być różne zależnie od wybranych krawędzi:

tablica powiązań			
krawędź	wierzchołki	trójkąty	krawędzie powiązane
a	A, D	$3, 1$	e, f, b, c
b	A, B	$1, 4$	c, a, f, d
c	B, D	$1, 2$	a, b, d, e
d	B, C	$2, 4$	e, c, b, f
e	C, D	$2, 3$	c, d, f, a
f	A, C	$4, 3$	d, b, a, e

tablica wierzchołków		powiązana krawędź
A	$-1, 0, 1$	a
B	$0, 0, -1$	b
C	$1, 0, 1$	f
D	$0, 1, 0$	c

tablica trójkątów	powiązana krawędź
1	a
2	c
3	a
4	b

Struktury danych siatek mają różną złożoność, co pozwala dostosować reprezentację do aktualnych potrzeb. Kryterium może być rodzaj operacji, które algorytmy wykonują na tych siatkach. Tabela 2.1 zestawia możliwości danej reprezentacji w zakresie wyszukiwania elementów siatki. Z reguły, im bardziej skomplikowana struktura danych, tym łatwiej znaleźć elementy

siatki sąsiadujące z rozpatrywanym. Jednakże tylko reprezentacja indeksowa pozwala przejść bezpośrednio z trójkąta do jego wierzchołków, bez redundancji danych charakterystycznej dla reprezentacji bezpośredniej.

Tabela 2.1. Zestawienie cech reprezentacji siatek trójkątów. Łatwość operacji rozumiana jest jako brak konieczności przeszukiwania całej tablicy elementów, w celu znalezienia poszukiwanego.

cecha	bezpośrednia	indeksowa	krawędziowa	Baumgarta
każdy wierzchołek zapamiętany tylko raz	-	+	+	+
łatwo zmienić współrzędne wierzchołka	-	+	+	+
łatwo znaleźć trójkąty o wspólnej krawędzi	-	-	+	+
łatwo znaleźć krawędź wspólną dla trójkątów	-	-	-	+
łatwo znaleźć krawędzie łączące się w wierzchołku	-	-	-	+
łatwo znaleźć współrzędne wierzchołków trójkąta znając jego indeks lub położenie w strumieniu danych	+	+	-	-

2.3 Poprawność budowy siatki trójkątów

Wizualizacja i przetwarzanie siatki trójkątów reprezentującej model bryły stawia przed tą siatką określone wymagania. Siatka powinna reprezentować powierzchnię obiektu, a więc powinna być zbiorem jednospójnym – nie może mieć otworów, gdyż wtedy nie zamknie poprawnie bryły. Jeśli siatka reprezentuje obiekt podzielony na części, to wymaganie jednospójności odnosi się do każdej z tych części z osobna. Drugim wymaganiem jest, aby każdy trójkąt pełnił funkcję granicy między obiektem, a otoczeniem. Wymagania te można zapisać również w poniższy sposób:

— każdy trójkąt ma trzy wierzchołki – tablica trójkątów zawiera indeksy, które nie mogą być puste,

— każda krawędź ma dwa wierzchołki i jest częścią wspólną dwóch trójkątów,
— z każdego wierzchołka wychodzą co najmniej trzy krawędzie i trzy trójkąty, wychodzących krawędzi jest tyle samo, co trójkątów.

Taka organizacja struktury danych i taki sposób utrzymania ich spójności gwarantuje, że algorytmy uruchamiane na siatce zawsze odnajdą elementy siatki sąsiednie do rozpatrywanego i będą mogły wykonać na nich obliczenia. Ponadto każdy algorytm modyfikujący siatkę powinien wykonać operację tak, aby powyższe wymagania pozostały spełnione również w zmodyfikowanej siatce.

Spełnienie powyższych wymagań można sprawdzić znając liczbę wierzchołków (vertex V), liczbę krawędzi (edge E), liczbę ścian (face F), liczbę otworów (inaczej pętli) w ścianach (hole H), liczbę części, na które podzielony jest obiekt (shell S) oraz liczbę tuneli[3] (genus G) w reprezentowanych przez siatkę obiektach. Pomocna przy tym jest charakterystyka Eulera [36, 21, 108] (2.3):

$$\chi = V - E + F - H - 2(S - G) \tag{2.3}$$

Charakterystyka Eulera dla spełniającej powyższe wymagania siatki wynosi $\chi = 0$. Mając daną siatkę można ją modyfikować dodając lub usuwając wierzchołki, krawędzie lub trójkąty. Po każdej modyfikacji poprawność siatki może zostać zweryfikowana poprzez ponowne obliczenie jej charakterystyki Eulera. Jeżeli do modyfikacji siatki są używane tylko operatory Eulera, to jest gwarancja, że jej charakterystyka Eulera nie zmieni się. Operatory tworzenia i operatory usuwania elementów siatki zostały zestawione w tabeli 2.2. Natomiast tabela 2.3 pokazuje charakterystyki Eulera dla wybranych siatek wielokątowych.

[3]Tunel przechodzi przez obiekt na wylot.

Tabela 2.2. Operatory Eulera: tworzenie i usuwanie elementów siatki.

nazwa operatora i czynność	zmiana liczby V	E	F	H	S	G
MEV tworzy krawędź i wierzchołek	+1	+1				
MFE tworzy ścianę i krawędź		+1	+1			
MSFV tworzy shell, ścianę i wierzchołek	+1		+1		+1	
MSG tworzy część i tunel					+1	+1
MEKL tworzy krawędź i usuwa otwór		+1		-1		
KEV usuwa krawędź i wierzchołek	-1	-1				
KFE usuwa ścianę i krawędź		-1	-1			
KSFV usuwa shell, ścianę i wierzchołek	-1		-1		-1	
KSG usuwa shell i tunel					-1	-1
KEML usuwa krawędź i tworzy otwór		-1		+1		

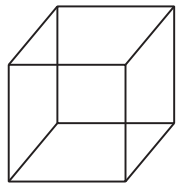
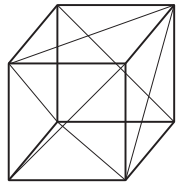
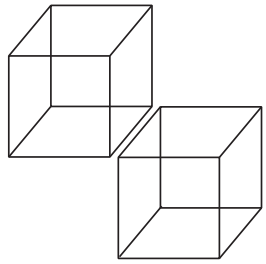
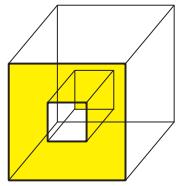
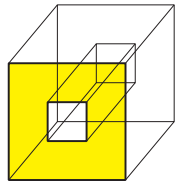
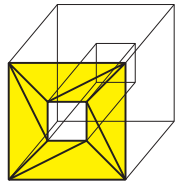
2.4 Jakość siatki trójkątów

Jakość siatki może mieć wpływ na przebieg algorytmów, które na niej pracują. Zarówno tych, które ją przetwarzają, jak i potoku renderingu. Ogólnie, im rozmiary poszczególnych trójkątów oraz krawędzi są bardziej do siebie zbliżone, tym jakość siatki lepsza. Do liczbowego określenia jakości siatki można użyć trzech miar. Oblicza się je na podstawie cech charakteryzujących poszczególne trójkąty i zależności między nimi, a następnie uśrednia dla całej siatki.

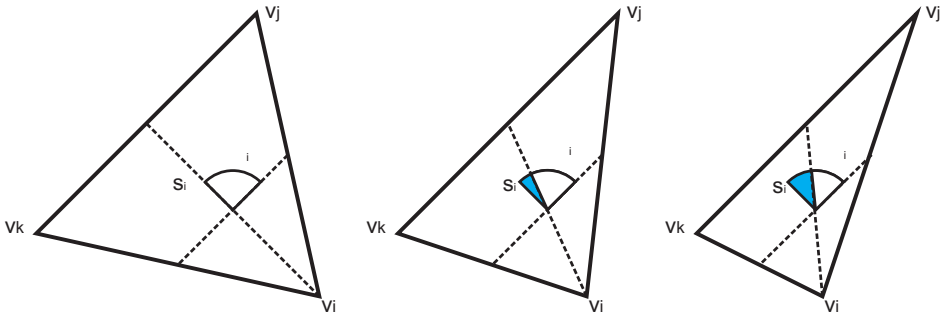
Pierwszą z tych miar jest **skręcenie trójkąta (s)**. Jest to minimalna wartość kąta (α_i) między środkową trójkąta, a odcinkiem łączącym środki przeciwległych boków, odjęta od 90° (2.4):

$$s_i = 90^\circ - \alpha_i. \tag{2.4}$$

Tabela 2.3. Charakterystyki Eulera dla wybranych siatek.

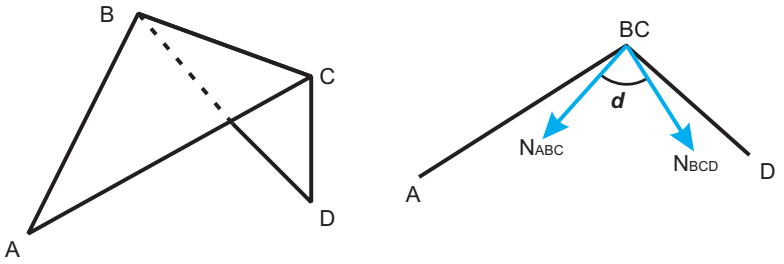
siatka i obraz	V	E	F	H	S	G	χ
sześcian zbudowany z czworokątów 	8	12	6	0	1	0	0
sześcian zbudowany z trójkątów 	8	18	12	0	1	0	0
podwójny sześcian zbudowany z czworokątów 	16	24	12	0	2	0	0
sześcian z otworem (pętlą) w ścianie zbudowany z wielokątów z otworami 	16	24	11	1	1	0	0
sześcian z tunelem zbudowany z wielokątów z otworami 	16	24	10	2	1	1	0
sześcian z tunelem, przednia ściana sześcianu zbudowana jest z trójkątów 	16	32	17	1	1	1	0

Mieści się w zakresie $< -90°, 90° >$, a wartość pożądana, czyli najmniejsze skręcenie to $0°$ (rys. 2.14).



Rysunek 2.14. Skręcenie trójkąta.

Drugą z miar przydatnych w badaniu jakości siatki jest **miara kąta dwuściennego dla krawędzi (d)**, czyli kąt między normalnymi trójkątów tej krawędzi (rys. 2.15). Jej zakres to $< 0°, 180° >$, a wartość pożądana jest wartością minimalną, czyli równą $0°$.



Rysunek 2.15. Miara d kąta dwuściennego dla krawędzi BC. N_{ABC} i N_{BCD} są normalnymi trójkątów krawędzi BC (odpowiednio ABC i BCD).

Ostatnią miarą jakości siatki jest **jakość trójkąta (q)**. Jest to odchylenie od trójkąta równobocznego, obliczane jako odchylenie wartości q (równanie 2.5) od jedności:

$$q = 4\sqrt{3}\frac{A}{a^2 + b^2 + c^2}, \tag{2.5}$$

gdzie: A – pole trójkąta; a, b, c – długości boków trójkąta.

Wartość $q = 1$ występuje dla trójkątów równobocznych. Natomiast wartości minimalne, bliskie zeru, są najbardziej niepożądane ponieważ charakteryzują cienkie i wydłużone trójkąty.

Mając miary jakości poszczególnych elementów siatki można obliczyć je dla całej badanej siatki trójkątów. Średnie skręcenie (2.6):

$$Q_s = \frac{1}{n_f} \sum_{i=1}^{n_f} s_i, \tag{2.6}$$

gdzie n_f – liczba trójkątów w siatce.

Średnia wartość kąta dwuściennego krawędzi siatki (2.7):

$$Q_d = \frac{1}{n_e} \sum_{i=1}^{n_e} d_i, \tag{2.7}$$

gdzie n_e – liczba krawędzi w siatce.

Średnia jakość trójkąta w siatce (2.8):

$$Q_q = \frac{1}{n_f} \sum_{i=1}^{n_f} q_i, \tag{2.8}$$

gdzie n_f – liczba trójkątów w siatce.

2.5 Atrybuty siatki trójkątów

Atrybuty siatki trójkątów są z reguły związane z jej wierzchołkami. W zastosowaniu do renderingu wierzchołki mają zwykle następujące atrybuty:

- współrzędne przestrzenne $V = (x, y, z)$ – jest to jedyny obowiązkowy atrybut wierzchołka,
- kolor RGB zadany w trakcie tworzenia modelu lub obliczony w czasie renderingu,
- wektor normalny (N),
- współrzędne teksturowania (s, t).

Aplikacje przetwarzające siatki mogą wierzchołkom przypisywać również inne cechy, na przykład właściwości fizyczne takie, jak ciśnienie, czy temperatura.

Także trójkąty siatki mogą mieć przypisane atrybuty. Jednym z najczęściej występujących atrybutów jest normalna (N_f), czyli wektor prostopadły do powierzchni trójkąta. Można go obliczyć z iloczynu wektorowego krawędzi trójkąta (2.9):

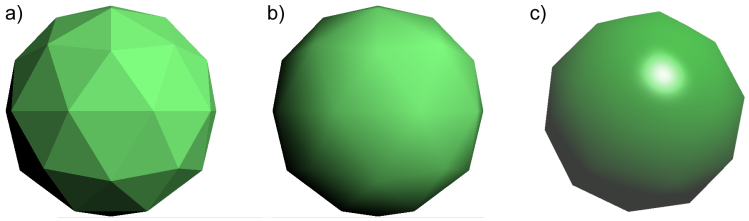
$$N_f = ((V_2 - V_1) \times (V_3 - V_1))^\circ, \tag{2.9}$$

gdzie:

V_1, V_2, V_3 – położenia wierzchołków trójkąta,

symbol $^\circ$ oznacza funkcję normalizującą wektor: $W^\circ = \frac{W}{||W||}$.

Po odpowiednim ustaleniu konwencji zapisu orientacji trójkątów (tzn. czy zewnętrzna strona trójkąta ma wierzchołki ułożone zgodnie z ruchem wskazówek zegara, czy odwrotnie), można wektor (N) traktować jako zawsze skierowany na zewnątrz obiektu.

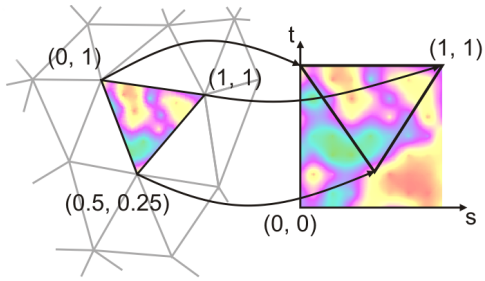


Rysunek 2.16. Cieniowanie sfery reprezentowanej przez siatkę trójkątów: a) płaskie, b) Gourauda, c) Phonga.

Użycie do renderingu wektorów normalnych obliczonych z równania (2.9) daje efekt cieniowania płaskiego z widocznym podziałem siatki (rys. 2.16a). Dla uzyskania gładkiego pocieniowania powierzchni potrzebne są wektory normalne w wierzchołkach (rys. 2.16bc). Mogą być one obliczone z równania powierzchni lub uśrednione na podstawie reprezentacji siatkowej. W tym drugim przypadku wektor normalny w wierzchołku (N) jest znormalizowaną sumą wektorów normalnych trójkątów, które zawierają ten wierzchołek (2.10):

$$N = (\sum_{i=1}^{n_f} N_f^i)^\circ, \tag{2.10}$$

gdzie: n_f – liczba trójkątów wychodzących z wierzchołka, N_f^i – normalna i-tego trójkąta.



Rysunek 2.17. Mapowanie tekstury na siatkę trójkątów.

Tekstura powstaje z obrazu i jest prostokątną tablicą danych, a współrzędne tekstury (s, t) są liczbami zmiennoprzecinkowymi z zakresu $< 0, 1 >$. Współrzędne te determinują, który teksel[4] na mapie tekstury odpowiada danemu wierzchołkowi (rys. 2.17). Są one interpolowane między wierzchołkami tak samo, jak kolory wierzchołków w cieniowaniu trójkąta [78].

Atrybuty wierzchołków: kolor i współrzędne teksturowania mogą być wielokrotne. Standardy pozwalają na dołączenie dwóch kanałów koloru i ośmiu kanałów współrzędnych teksturowania do jednego wierzchołka. Zazwyczaj takie przypisanie odbywa się w aplikacji do modelowania obiektu.

2.6 Rendering siatek trójkątów

Siatki wielokątowe, jako jedna z najpopularniejszych reprezentacji, są obsługiwane zarówno przez renderery używane w przemyśle filmowym[5], jak i przez renderery używające technik globalnego oświetlenia[6], lecz ich głównym obszarem zastosowań jest rendering czasu rzeczywistego, sprzętowo zaimplementowany na GPU. Współczesne karty graficzne wspierają zarówno klasyczny, jak i programowalny potok renderingu siatek wielokątowych.

[4] Texel (ang. texture element) – podstawowy element budujący teksturę.

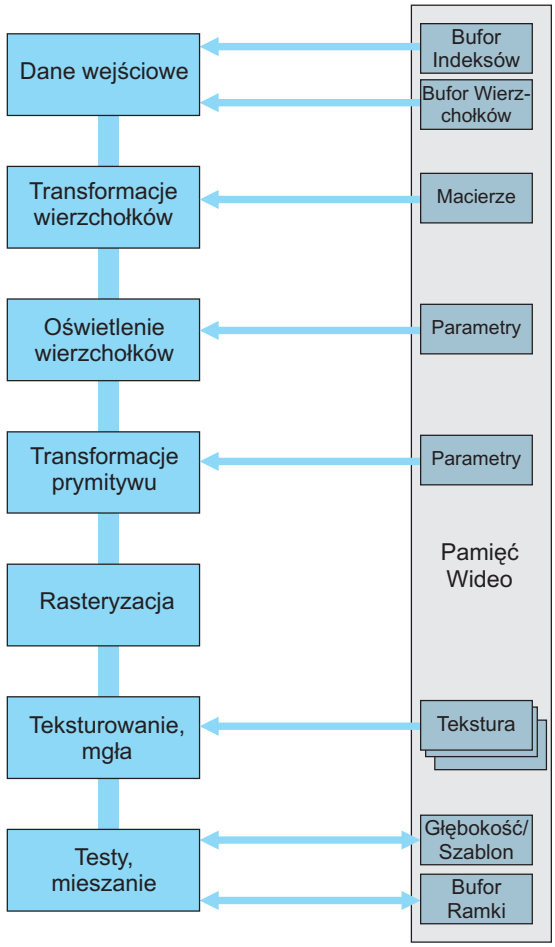
[5] algorytm REYES używany przez RenderMan obsługuje wiele reprezentacji brył, w tym siatki wielokątowe.

[6] np. V-Ray i Mental Ray programu 3ds Max.

2.6.1 Klasyczny potok renderingu

Klasyczny potok renderingu jest potokiem przetwarzania danych graficznych, którego kolejne etapy odbywają się według stałych, zdefiniowanych algorytmów (ang. fixed-function rendering pipeline). Między innymi dlatego nazywany jest potokiem z Z-buforem i cieniowaniem Gourauda. Z-bufor jest nazwą algorytmu określania powierzchni widocznych, natomiast cieniowanie Gourauda nazwą algorytmu obliczania kolorów fragmentów, na które dzielony jest wielokąt siatki. Oprócz tych dwóch najbardziej charakterystycznych algorytmów, w potoku renderingu wykonuje się wiele innych, lecz jedyne, co programista może zmieniać to parametry, i warianty wykonania tych algorytmów. Etapy klasycznego potoku renderingu wielokątów przedstawia rysunek 2.18.

1. Przekazane karcie graficznej dane wejściowe potoku, stanowią atrybuty kolejnego prymitywu siatki: położenie w lokalnym układzie współrzędnych, kolor, wektor normalny, współrzędne teksturowania. Istotna jest również informacja o typie prymitywu (punkt, linia, trójkąt, czworokąt), czyli sposobie łączenia wierzchołków.
2. Transformacje wierzchołków – przekształcenie współrzędnych wierzchołka z lokalnego układu współrzędnych obiektu do układu współrzędnych świata, a następnie do układu punktu widzenia (kamery). Przekształcane są również współrzędne wektora normalnego. W OpenGL przekształcenie następuje bezpośrednio do współrzędnych punktu widzenia.
3. Oświetlenie – obliczenie koloru wierzchołka z modelu oświetlenia Phonga-Blinna.
4. Transformacje prymitywu – działają na wierzchołkach prymitywu:
   — Najważniejszą operacją jest obcinanie prymitywu przez płaszczyzny bryły widzenia. Jest ono realizowane we współrzędnych jednorodnych obcinania, więc należy do nich najpierw przekształcić współrzędne 3D wierzchołków prymitywu. Obcinanie punktów polega na przepuszczeniu wierzchołka do następnego etapu lub



Rysunek 2.18. Etapy klasycznego potoku renderingu.

odrzuceniu go. Obcinanie linii lub wielokątów może tworzyć dodatkowe wierzchołki.

— Dzielenie przez W jednorodnych współrzędnych wierzchołków daje współrzędne rzutu perspektywicznego.

— Przekształcenie widoku – konwersja do trójwymiarowych współrzędnych ekranu.

5. Rasteryzacja – podział prymitywu na elementarne fragmenty[7]. Każdy fragment otrzymuje kolor interpolowany dwuliniowo na podstawie kolorów wierzchołkowych (metoda cieniowania Gourauda) i głębię, czyli odległość od kamery.
6. Teksturowanie – łączenie koloru fragmentu z kolorem teksela.
7. Mgła – łączenie koloru fragmentu z kolorem mgły.
8. Testy nożyc, przezroczystości, szablonu i głębi (algorytm Z-bufora).
9. Mieszanie (ang. blending) – łączenie koloru fragmentu z kolorem piksela, który jest zapisany w buforze koloru.
10. Zapis do bufora koloru.

Stała funkcjonalność powoduje, że użytkownik nie ma wpływu na algorytmy i kolejność wykonywania powyższych czynności. Ma jedynie wpływ na dane wprowadzane do potoku renderingu:

— typ rysowanych prymitywów
— współczynniki materiału i parametry źródeł światła używane w modelu oświetlenia Phonga-Blinna,
— macierze transformacji,
— parametry mgły,
— warunki testów i operatory mieszania,
— mapy i parametry tekstur.

Oświetlenie wierzchołka jest obliczane metodą Phonga-Blinna. Kolor (I) wierzchołka jest sumą czterech składowych (2.11, rys. 2.19):

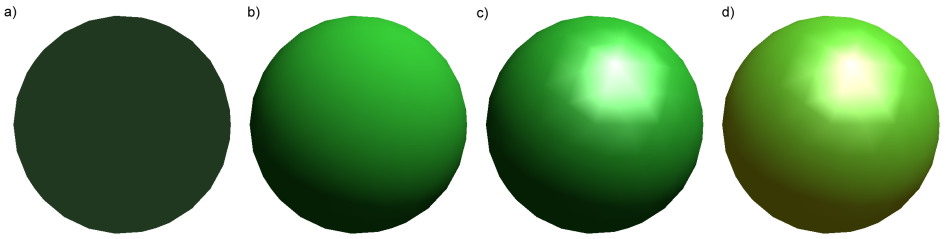
$$I = I_a + I_d + I_s + I_e, \tag{2.11}$$

gdzie:

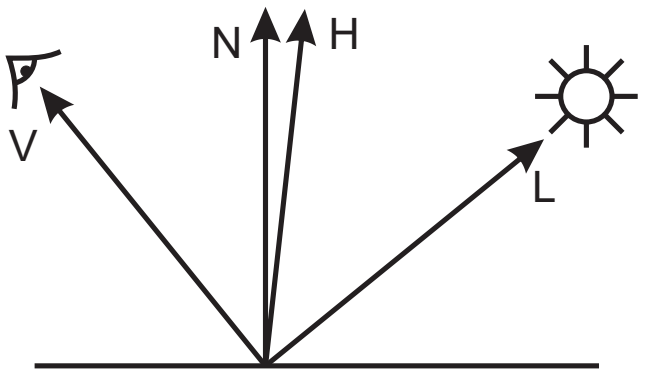
I_a – składowa otoczenia (ang. ambient), niezależna od położenia wierzchołka w obiekcie, imituje światło rozproszone przez elementy środowiska, to znaczy inne obiekty,

[7] fragment stanie się pikselem, który można wyświetlić na ekranie dopiero wtedy, gdy przejdzie wszystkie operacje i testy.

I_d – składowa rozproszenia (ang. diffuse), odpowiedzialna za wygląd powierzchni matowych, zależna od kąta między normalną (N), a wektorem kierunkowym źródła światła (L),
I_s – składowa rozbłysku (ang. specular), odpowiedzialna za wygląd powierzchni błyszczących, zależna od kierunku obserwacji (V),
I_e – składowa emisji (ang. emission), dodatkowy składnik barwy imitujący efekt samoświecenia dla powierzchni lamp.



Rysunek 2.19. Model oświetlenia Phonga-Blinna: a) składowa ambient; b) dołączona składowa diffuse; c) dołączona składowa specular; d) dołączona składowa emission.



Rysunek 2.20. Wektory w modelu oświetlenia Phonga-Blinna.

W potoku renderingu OpenGL równanie oświetlenia Phonga-Blinna ma następującą postać (równanie 2.12, rys. 2.20) [78]:

$$I = I_e + I_{env} \cdot k_a + \sum_{i=0}^{l-1} [f_{att_i} \cdot f_{spot_i} (I_{zra_i} k_a + I_{zrd_i} k_d (N \cdot L) + I_{zrs_i} k_s (H \cdot N)^n)] \tag{2.12}$$

gdzie:

I – finalny kolor wierzchołka,
I_e – intensywność emisji,
I_{env} – intensywność światła rozproszonego w środowisku,
l – liczba źródeł światła,
$f_{att_i} = \frac{1}{k_c + k_l \cdot d + k_q \cdot d^2}$ – współczynnik tłumienia i-tego źródła światła,
k_c, k_l, k_q – współczynniki tłumienia światła,
d – odległość wierzchołka od źródła światła,
f_{spot_i} – efekt wprowadzony przez reflektor (ang. spotlight effect),
$I_{zra_i}, I_{zrd_i}, I_{zrs_i}$ – składowe intensywności i-tego światła oświetlającego bezpośrednio wierzchołek, odpowiednio ambient, diffuse i specular,
k_a, k_d, k_s – współczynniki odbicia światła definiujące materiał obiektu, odpowiednio ambient, diffuse i specular,
n – wykładnik rozbłysku, określający wielkość rozbłysku światła,
N – znormalizowany wektor normalny w wierzchołku,
L – znormalizowany wektor kierunkowy światła,
V – znormalizowany wektor kierunkowy obserwacji,
$H = (L + V)^\circ$ – znormalizowany wektor połowiczny.

Wszystkie wektory mają długość 1.

Poszczególne składowe mogą się różnić w potokach renderingu oferowanych przez różne biblioteki graficzne, np. w DirectX jest jeden, globalny składnik ambient [12]. Natomiast w OpenGL są dwa składniki ambient: globalny $I_{env} k_a$ i powiązany z konkretnym światłem $I_{zra_i} k_a$.

Podczas rasteryzacji uzyskane w powyższy sposób kolory wierzchołków są interpolowane dwuliniowo (cieniowanie Gourauda [21]) na fragmenty prymitywu, co uniemożliwia uzyskanie wewnątrz prymitywu koloru jaśniejszego, niż w wierzchołkach (rys. 2.16b, 2.24a). Konsekwencją takiego podejścia jest utrata rozbłysków całkowicie mieszczących się wewnątrz prymitywu.

Mapa tekstury jest próbkowana metodą najbliższego sąsiada lub z interpolacją liniową i uzyskany kolor teksela jest łączony z kolorem fragmentu za pomocą wybranego przez programistę operatora: replace, modulate, decal lub blend. Aby uzyskać efekt pocieniowania obiektu, należy połączyć teksturę z achromatycznym obiektem[8] operatorem modulate mnożącym kolor teksela przez kolor fragmentu (rys. 2.21a). Fotograficzne mapy tekstury mają zapisany także rozkład oświetlenia. Dlatego do łączenia takiej tekstury z obiektem używa się operatora replace zamieniającego kolory fragmentów obiektu na kolory tekstury (rys. 2.21b[9]). Operator decal jest użyteczny w przypadku obiektów parawanowych (tzn. z obszarami przezroczystymi).

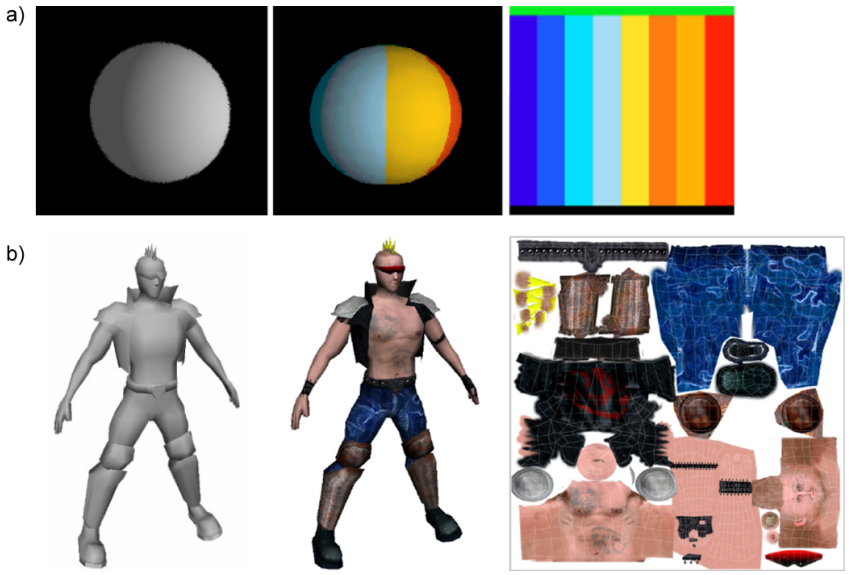
2.6.2 Programowalny potok renderingu

Współczesne karty graficzne oprócz klasycznego potoku renderingu oferują także możliwość zaprogramowania własnych funkcji przetwarzania dla poszczególnych etapów (rys. 2.22).

Etap **cieniowania wierzchołków** (ang. Vertex Shading) jest realizowany przez program cieniowania (shader) wierzchołków. Jego zadaniem jest wykonanie czynności realizowanych przez początkowe etapy klasycznego potoku renderingu, a więc przekształcenie współrzędnych wierzchołka do jednorodnych współrzędnych obcinania i obliczenie dla wierzchołka wartości, które są przeznaczone do interpolacji na fragmentach. Niedostępną w klasycznym potoku renderingu funkcjonalnością programów wierzchołków jest

[8] achromatyczny obiekt – współczynnik odbicia ma równe składowe (RGB), co odpowiada szarości.

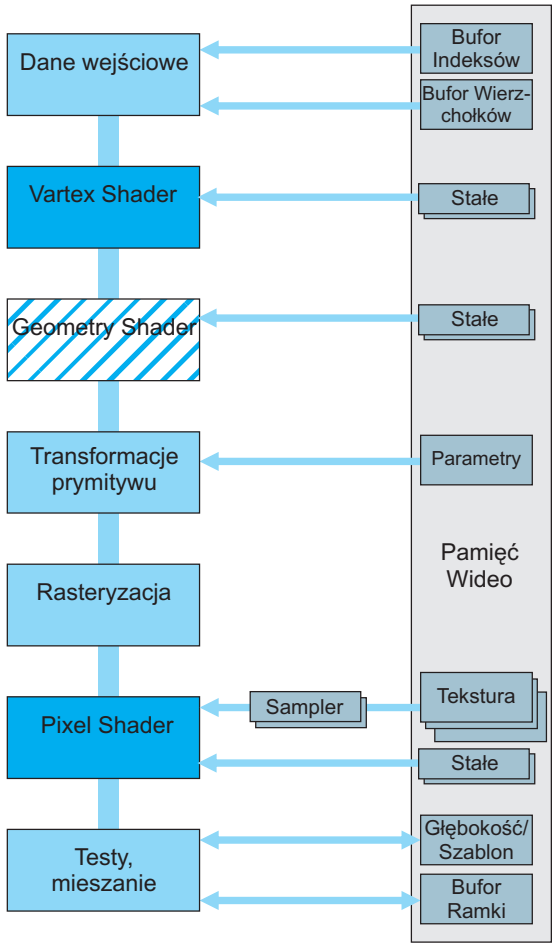
[9] Źródło: J. Grobelny, Praca dyplomowa magisterska, Instytut Informatyki PŁ.



Rysunek 2.21. Łączenie koloru fragmentu z kolorem teksela: a) rozkład oświetlenia został obliczony dla fragmentów, więc operator modulate pomnożył kolor fragmentu przez kolor teksela; b) rozkład oświetlenia zapisany jest w fotograficznej teksturze, więc operator replace zastąpił kolor fragmentu kolorem teksela.

możliwość interpretacji większej liczby parametrów związanych z wierzchołkiem. Poza położeniem, normalną, kolorem i współrzędnymi teksturowania może to być: drugi kanał koloru, wektory binormalny i styczny w wierzchołku i inne.

Etap cieniowania wierzchołków może wykonywać dowolne obliczenia związane z danym wierzchołkiem, jednak nie może sam stworzyć, ani usunąć wierzchołka z potoku. Taką możliwość dają wprowadzone w najnowszych wersjach GPU **programy cieniowania geometrii** (ang. Geometry Shader). Są one wykonywane na danych wierzchołków przed rasteryzacją. Służą do zmiany geometrii rysowanych obiektów. Jest to możliwe dzięki udostępnieniu funkcji usuwania i tworzenia wierzchołków w potoku. Dzięki temu narysowanie jednego wierzchołka przez aplikację może powodować narysowanie całego trójkąta. Można również zaplanować kryteria usuwania wierzchołków



Rysunek 2.22. Etapy programowalnego potoku renderingu.

i usuwać niepotrzebne wierzchołki nie w aplikacji, ale dopiero na etapie renderingu.

Po rasteryzacji następuje **cieniowanie fragmentów** (ang. Pixel Shading), które może realizować zadania z etapów klasycznego potoku renderingu związane z łączeniem koloru fragmentu z kolorami tekstur i mgły. Programy cieniowania fragmentów otrzymują zinterpolowane dane wierzchołków oraz tekstury, które mogą próbkować. Dzięki temu mogą łączyć kolor otrzymany z wcześniejszych etapów, kolor obliczony przez siebie oraz kolor

tekstury. Miejsce próbkowania tekstury może być również obliczone w programie cieniowania fragmentów. Zadaniem programu cieniowania fragmentów jest wyznaczenie finalnego koloru fragmentu dla dalszego przetwarzania (testów i mieszania) lub usunięcie go.

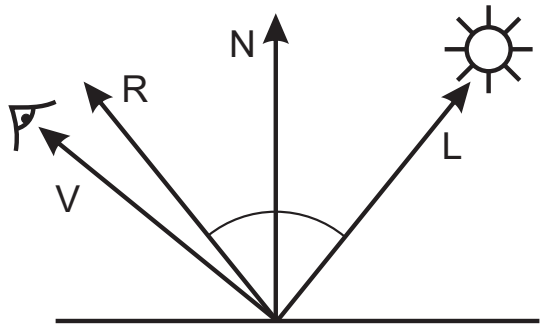
Programy cieniowania wierzchołków i programy cieniowania geometrii w najnowszych wersjach modelu cieniowania mogą próbkować tekstury, choć nie jest to ich typowym zastosowaniem, gdyż nie generują bezpośrednio koloru. Ta ich cecha może być jednak użyta w innych celach. Na przykład przy generowaniu terenu z mapy wysokości, położenia wierzchołków terenu zależą od koloru (jasności) teksela z tej mapy.

Z pomocą programowalnego potoku renderingu można zaimplementować inny od klasycznego model oświetlenia. Jednym z najczęściej używanych jest model Phonga z cieniowaniem Phonga. Kolor wierzchołka w tym modelu jest zbudowany z takich samych składowych, jak w modelu Phonga-Blinna i ma postać (równanie 2.13, rys. 2.23):

$$I = I_{zra} \cdot k_a + I_{zrd} k_d (N \cdot L) + I_{zrs} k_s (R \cdot V)^n \tag{2.13}$$

gdzie: R – odbity względem wektora normalnego (N) wektor kierunkowy światła (L).

Pozostałe oznaczenia są analogiczne, jak w (2.12).



Rysunek 2.23. Wektory w modelu oświetlenia Phonga.

Program cieniowania wierzchołków oblicza wektory z równania (2.13) i przekazuje je do etapu rasteryzacji.

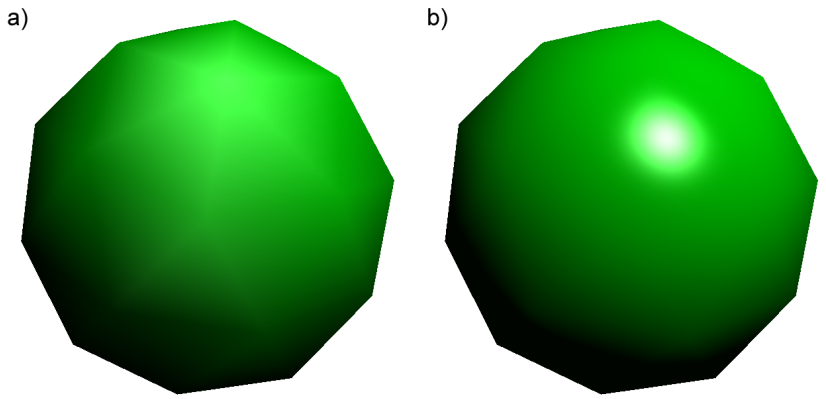
```
vertex_shader() {
 przekształć współrzędne wierzchołka z lokalnego układu współrzędnych
  do układu kamery
 przekształć współrzędne wierzchołka z układu kamery
  do jednorodnych współrzędnych obcinania
 przekształć wektor normalny N w wierzchołku do układu kamery
 przekształć wektor oświetlenia L w wierzchołku do układu kamery
 przekształć wektor punktu widzenia V w wierzchołku do układu kamery
 przekaż do interpolacji obliczone wektory
}
```

Każdy z wektorów jest w etapie rasteryzacji poddany interpolacji liniowej w celu obliczenia wartości wektora dla poszczególnych fragmentów. Wartość ta jest przekazywana do programu cieniowania fragmentów, w którym dla każdego fragmentu obliczany jest jego kolor.

```
fragment_shader() {
 znormalizuj otrzymany wektor normalny N fragmentu
 znormalizuj otrzymany wektor oświetlenia L fragmentu
 znormalizuj otrzymany wektor punktu widzenia V fragmentu
 oblicz R: wektor odbity światła L względem wektora normalnego N
 oblicz kolor fragmentu na podstawie wzoru Phonga (2.13):
  kolor_wynikowy = Ika
                 + Ikd * saturate( dot( N,L) )
                 + Iks * pow( saturate( dot(R,V) ), n )
}
```

W programie fragmentów otrzymane wektory należy ponownie znormalizować, gdyż wektory jednostkowe poddane interpolacji zmieniają kierunek, ale nie zachowują długości. Efekt działania modelu Phonga obliczanego w pikselach na obiekcie pokazuje rysunek 2.24b).

Równie łatwo i z lepszymi wynikami, niż w potoku klasycznym można dzięki programom cieniowania zaimplementować modele z materiałami zwierciadlanymi i przezroczystymi (rys. 2.25, 2.26).

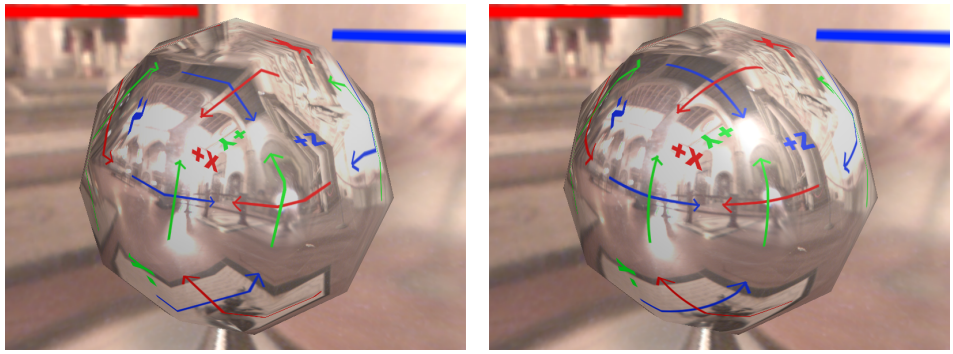


Rysunek 2.24. Porównanie renderingu klasycznego z renderingiem zaprogramowanym na GPU: a) Phong-Blinn z cieniowaniem Gourauda; b) Phong z cieniowaniem Phonga zaimplementowanym w programie cieniowania. Na rzadkiej siatce cieniowanie Gourauda zgubiło rozbłysk.

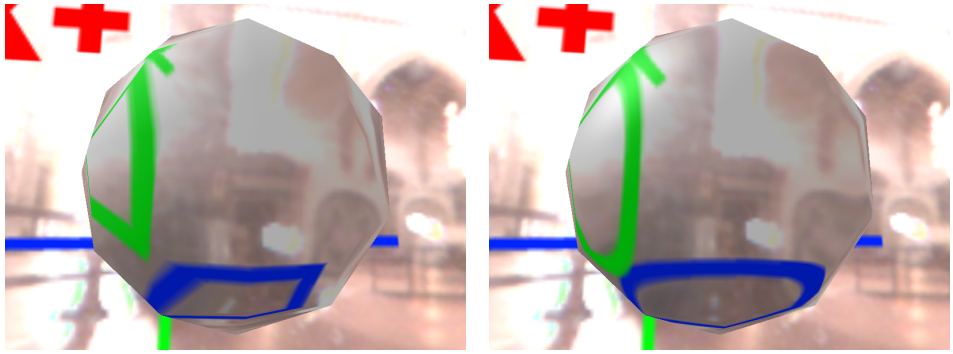
2.7 Podsumowanie

Omówione w niniejszym rozdziale reprezentacje nierzadko bardzo się między sobą różnią, gdyż różne były cele ich powstania i możliwości ich użycia. Wyboru reprezentacji należy dokonać na podstawie kryteriów przedyskutowanych poniżej (tab. 2.4):

1. **Dowolność kształtu reprezentowanych obiektów**
   Można to zrealizować przez udostępnienie pewnych klas obiektów, jak w przypadku reprezentacji sweep i kwadryk. Możliwa jest również całkowita dowolność kształtu w reprezentacjach objętościowych i używających powierzchni obiektów (powierzchnie parametryczne, siatki wielokątowe, powierzchnie subdivision). Wyjątkiem jest tu CSG, gdzie dowolność kształtu jest ograniczona przez dostępność obiektów prymitywnych i zbiór operatorów na nich działających.
2. **Łatwość modelowania**
   Obowiązuje ogólna zasada, że im prostsza reprezentacja tym łatwiejsze jest ręczne modelowanie. Łatwo jest modelować przy pomocy kopiowania prymitywów, reprezentacji sweep, czy CSG. Bardzo intuicyjne



Rysunek 2.25. Mapowanie środowiska na obiekcie odbijającym zwierciadlanie: klasyczne (po lewej), przy pomocy programu cieniowania (po prawej).



Rysunek 2.26. Mapowanie środowiska na obiekcie przezroczystym: klasyczne (po lewej), przy pomocy programu cieniowania (po prawej).

w modelowaniu są powierzchnie parametryczne (łatwiejsze powierzchnie Beziera niż NURBS) i subdivision. Siatki wielokątowe zazwyczaj modyfikuje się za pośrednictwem dodatkowych operatorów ponieważ modelowanie bezpośrednie jest żmudne. Stosunkowo trudno jest modelować używając reprezentacji objętościowych.

3. **Możliwość akwizycji**
   Automatycznie pozyskiwane dane ze skanerów 3D i tomografii komputerowej mogą być reprezentowane przez zbiory elementów. Skanowanie

trójwymiarowe daje chmurę punktów konwertowalną na siatkę wielokątową. Tomografia komputerowa i rezonans magnetyczny są głównym źródłem danych w reprezentacji wokselowej.

4. **Dokładność odwzorowania kształtów**

   Reprezentacje z przesuwaniem, CSG, kopiowanie prymitywów, kwadryki, powierzchnie parametryczne są reprezentacjami, które mogą z matematyczną dokładnością reprezentować pewne klasy kształtów. Pozostałe reprezentacje są mniej lub bardziej dokładnymi aproksymacjami. Siatki wielokątowe mogą być dokładne dla obiektów, których powierzchnię można złożyć z prymitywów będących wielokątami siatki. Reprezentacja BSP odwzorowuje dokładnie siatkę powierzchni, z której powstała, ale siatka ta ze swej natury może niedokładnie aproksymować powierzchnię.

5. **Zamykanie brył**

   Obok dokładności jest ważną cechą w projektowaniu. Zapewniają je następujące reprezentacje:

   — kopiowanie prymitywów, jeśli algorytmy prymitywów to przewidują,
   — CSG – dzięki regularyzowanym operatorom (również, jeśli same prymitywy wejściowe mają tę cechę),
   — reprezentacje objętościowe ze swej natury.

   Reprezentacja sweep przedstawia bryły pod dodatkowymi warunkami. Kwadryki są obiektami zamkniętymi lub otwartymi zależnie od typu. Obiekty w pozostałych reprezentacjach muszą przechodzić szereg testów lub spełniać dodatkowe założenia, żeby reprezentować poprawne bryły. Dotyczy to reprezentacji powierzchniowych. Siatki wielokątowe można testować przy pomocy charakterystyki Eulera omówionej w niniejszym rozdziale.

6. **Oszczędność zasobów**

   Najbardziej oszczędne pod względem zasobów pamięci są reprezentacje przechowujące obiekty jako zbiór parametrów, czyli kopiowanie

prymitywów, CSG[10], reprezentacja z przesuwaniem, kwadryki. Więcej zasobów będą potrzebować powierzchnie parametryczne, jeszcze więcej powierzchnie subdivision i siatki wielokątowe. Najwięcej pamięci zużywają zazwyczaj reprezentacje objętościowe, z których drzewa, z racji hierarchicznej organizacji są najoszczędniejsze, natomiast reprezentacja elementami skończonymi i reprezentacja wokselowa są najbardziej wymagające.

7. Dostępność narzędzi i formatów plików

Najwięcej narzędzi obróbki i formatów plików obsługuje siatki wielokątowe (trójkątne). Ostatnio, dzięki zwiększającej się dostępności sprzętu z dużą pamięcią, również reprezentacje objętościowe zyskują coraz to nowe narzędzia i formaty plików. Na przykład oprogramowanie pozwalające na oglądanie obrazów CT, „GEHC Microview", obsługuje używany w medycynie format DICOM. Natomiast silniki gier komputerowych korzystają z szerokiego wachlarza reprezentacji, aby zapewnić jak najlepszy wynik wizualny w czasie rzeczywistym. Są to zarówno siatki wielokątowe, jak i drzewa BSP, czy ósemkowe. Narzędzia takie, jak BRL-CAD[11] dla CSG, czy CADRE[12] dla elementów skończonych są dedykowane projektom inżynierskim.

[10] Gdy operacja jest możliwa na obiektach reprezentowanych przez dwie siatki wielokątowe to oprogramowanie musi zapamiętać obie siatki i operator, co zajmie więcej pamięci, niż siatka wynikowa. Zaleca się wtedy konwersję bryły CSG na siatkę wielokątową.

[11] Ballistic Research Laboratory CAD – program do modelowania brył pozwalający interaktywnie edytować geometrię i dokonywać jej analizy oraz renderować ją (ray-tracing). *Źródło:* `http://brlcad.org/`.

[12] CADRE Analytic – program pozwalający między innymi: modelować w 3D za pomocą elementów skończonych, rysować mapy sił, analizować stabilność i wyboczenia, rysować wykresy obciążeń. *Źródło:* `http://www.cadreanalytic.com/`.

8. Techniki renderingu

Obecnie jedynie rendering siatek trójkątnych jest bezpośrednio wspomagany sprzętowo. Pozostałe reprezentacje muszą zostać poddane konwersji na siatkę trójkątów przed renderingiem sprzętowym lub korzystać z programowalnych potoków renderingu. Algorytmy konwersji do siatki trójkątów mają różną efektywność dla różnych reprezentacji wejściowych. Ponadto może okazać się, że potok oferowany przez sprzęt graficzny nie spełnia wymogów jakości obrazu. Alternatywą dla sprzętowego wielokątowego potoku renderingu jest w wielu programach do modelowania i animacji komputerowej (np. 3ds Max) rendering scanline, natomiast fotorealistyczne obrazy sceny powstają z użyciem algorytmów globalnego oświetlenia. Popularną alternatywą dla renderingu wielokątowego jest rendering technikami śledzenia promieni dobrze pracujący z powierzchniami, dla których szybko można obliczyć punkt przecięcia z promieniem (np. kwadrykami). Dla reprezentacji objętościowych zamiast konwersji na siatkę wielokątową można zastosować objętościowe rzucanie promieni. Z kolei w produkcji filmowej, wymagającej najwyższej jakości modeli i obrazów, od wielu lat używany jest algorytm REYES operujący głównie na powierzchniach krzywoliniowych (stosowany w programie RenderMan) [70].

Algorytmy przedstawione w dalszej części niniejszej monografii modyfikują siatkę trójkątów. Ich szczegóły zaprezentowane w następnym rozdziale uzasadniają wybór reprezentacji indeksowej do przetwarzania geometrii obiektów topiących się i sublimujących.

Tabela 2.4. Wybrane cechy reprezentacji. Symbol + oznacza spełnienie kryterium, **o** spełnienie pod pewnymi warunkami, − niespełnienie lub duże trudności w spełnieniu. Liczba + w kolumnie „oszczędność" to stopień oszczędności.

reprezentacja	dowolność kształtu	łatwość modelowania	możliwość akwizycji	dokładność odwzorowania	zamykanie brył	oszczędność zasobów	dostępność *narzędzi* i formatów
kopiowanie prymitywów	−	+	−	+	+	+++	*CAD, RenderSoft, VRML Editor*; .vrml, .dxf
CSG	o	+	−	+	+	+++	*POV-Ray, RenderMan, BRL-CAD*; .pov, .rib
sweep	o	+	−	+	o	+++	*CAD*
kwadryki	o	o	−	+	+/−	+++	*POV-Ray, RenderMan*; .pov, .rib
powierzchnie parametryczne	+	+	+	+	−	++	*3ds Max, Maya, BRL-CAD; Rhinoceros 3D*; .max, .3dm
siatki wielokątowe	+	o	+	+/−	−	+	*3ds Max, Maya, RenderMan*; .3ds, .ase, .obj, .vrml, .ply, Collada, .dxf, .rib
powierzchnie subdivision	+	+	−	−	−	++	*RenderMan*; .rib
MES	+	nd.	−	−	+	+	*CADRE*, .fem
wokselowa	+	o	+	−	+	+	*VolPack, GEHC Microview*; .vff, DICOM, .raw, .dds, .pvm
drzewa ósemkowe	+	o	−	−	+	++	*silnik gier Ogre*; .oct
drzewa BSP	+	−	−	−	−	+	*silniki gier wywodzone z Quake*; .bsp

3

Modyfikacja siatki obiektu w procesie sublimacji

Zarówno sublimacja, jak i topnienie są zjawiskami, które zachodzą w miarę upływu czasu. Ich wizualizacja opiera się na obliczaniu kolejnych stanów wyglądu obiektu i wyświetlaniu ich. Techniki i algorytmy przedstawione w niniejszym rozdziale są realizacją pojedynczego *cyklu obliczeń*, przetwarzającego bieżący stan obiektu do następnego stanu, w miarę postępu wizualizowanego zjawiska.

3.1 Deformacja siatki powierzchni

Przedstawiona technika deformacji siatki w trakcie sublimacji oparta jest na przesuwaniu powierzchni międzyfazowej w każdym cyklu obliczeń. Rolę powierzchni międzyfazowej pełni powierzchnia obiektu reprezentowana przez siatkę trójkątów. Przesuwanie powierzchni międzyfazowej może więc być realizowane jako przesuwanie trójkątów wchodzących w skład siatki, a dokładniej wierzchołków, na których rozpięte są te trójkąty. Powierzchnia ta jednocześnie reprezentuje aktualną geometrię obiektu oraz umożliwia jego

wizualizację. Proponowana technika opiera się na cyklicznym przesuwaniu wierzchołków siatki przy uwzględnieniu:

— jego otoczenia przez inne wierzchołki – takie, które leżą na przeciwnych końcach krawędzi, które z niego wychodzą,
— wrażliwości wierzchołka na czynniki zewnętrzne, na przykład nacisk i temperaturę.

Dlatego podczas obliczeń uwzględniane są zarówno położenia wierzchołka, jak i położenia oraz orientacje powiązanych z nim trójkątów i krawędzi oraz sąsiadujących wierzchołków. Przesuwanie wierzchołków odbywa się poprzez obliczenie w każdym cyklu obliczeń indywidualnych **wektorów przesunięcia** (M) dla każdego z nich.

Siatka reprezentująca sublimujący obiekt kurczy się, dlatego w kolejnych cyklach następuje zmiana topologii siatki w kierunku jej upraszczania poprzez usuwanie wierzchołków, krawędzi i trójkątów wytypowanych przez algorytmy.

Poniżej opisana została technika wyznaczania wektorów przesunięcia poszczególnych wierzchołków siatki w kolejnych cyklach obliczeń. Deformacja siatki powinna być uzależniona od jej cech geometrycznych, dlatego wymaga się aby były spełnione poniższe **założenia** dla wektora przesunięcia wierzchołka:

Zał. 1. Powinien być skierowany do wewnątrz obiektu tak, aby przesunięcie wzdłuż niego „kurczyło" obiekt.

Zał. 2. Jego długość powinna być tym większa, im większą wypukłość tworzy wierzchołek, a więc powinna zależeć od nieregularności powierzchni. Im powierzchnia jest bardziej nieregularna, tym większa jest szybkość przepływu energii z otoczenia do fazy stałej i szybciej przebiega zjawisko (rozdział 1.1). W związku z tym wierzchołki bardziej wysunięte (większa nieregularność) powinny być przesuwane szybciej od pozostałych.

Zał. 3. Powinien być obliczany lokalnie, to znaczy na podstawie parametrów sąsiadujących z nim trójkątów i ich wierzchołków.

Zał. 4. Jego długość powinna dać się regulować (skalować) przy pomocy globalnego i lokalnych współczynników reprezentujących czynniki zewnętrzne wpływające na przebieg sublimacji.

Zastosowane zostały dwa podejścia do obliczania wektora przesunięcia w każdym cyklu dla każdego wierzchołka. Pierwsze z nich uwzględnia fakt, że przesuwanie powierzchni międzyfazowej można przedstawić jako „cofanie" w kierunku wnętrza obiektu trójkątów wchodzących w skład siatki reprezentującej powierzchnię międzyfazową. Cofanie to może być realizowane przez przesunięcie trzech wierzchołków trójkąta wzdłuż wektora o zwrocie przeciwnym do wektora normalnego tego trójkąta. Suma wektorów normalnych trójkątów we wspólnym i-tym wierzchołku daje normalną wierzchołkową N^i, zatem wektor przesunięcia N_N^i i-tego wierzchołka wyrażony jest wzorem (3.1):

$$M_N^i = -N^i. \tag{3.1}$$

Podstawiając do (3.1) równanie (2.10) otrzymuje się wzór na wektor przesunięcia (M_N^i) obliczony na podstawie wektorów normalnych trójkątów zbudowanych na i-tym wierzchołku (3.2):

$$M_N^i = -(\sum_{j=1}^{n_f^i} N_{f_j}^i)^\circ, \tag{3.2}$$

gdzie:

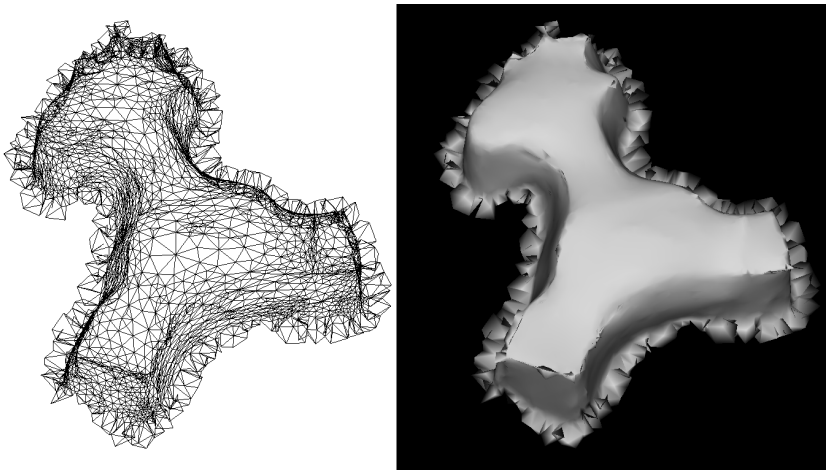
n_f^i – liczba trójkątów o wspólnym, i-tym wierzchołku,

$N_{f_j}^i$ – normalna j-tego trójkąta o wspólnym, i-tym wierzchołku.

Zaletą tak obliczonego wektora przesunięcia jest możliwość skorzystania z danych, które są potrzebne także w innych etapach wizualizacji, na przykład w czasie obliczenia koloru z równań modelu oświetlenia.

Siatka trójkątów aproksymuje powierzchnię obiektu. Konsekwencją tego jest fakt, że wektor normalny w wierzchołku (N) obliczony na podstawie normalnych trójkątów (N_f), a nie samej powierzchni, może teoretycznie nie być skierowany na zewnątrz reprezentowanego obiektu. Może to się stać jeśli kąt dwuścienny między dwiema normalnymi trójkątów o wspólnej krawędzi

wychodzącej z tego wierzchołka jest szczególnie niekorzystny (duży). Informuje o tym miara d jakości siatki zdefiniowana w rozdziale 2.4. W związku z tą niedokładnością w obliczaniu wektora normalnego (N), również wektor M_N^i może nie być skierowany do wnętrza obiektu. Jednakże w praktyce taka sytuacja występuje rzadko. Dlatego przyjęto, że obliczony wektor M_N^i spełnia założenie **1.** Ponadto wektor ten spełnia założenie **3.**

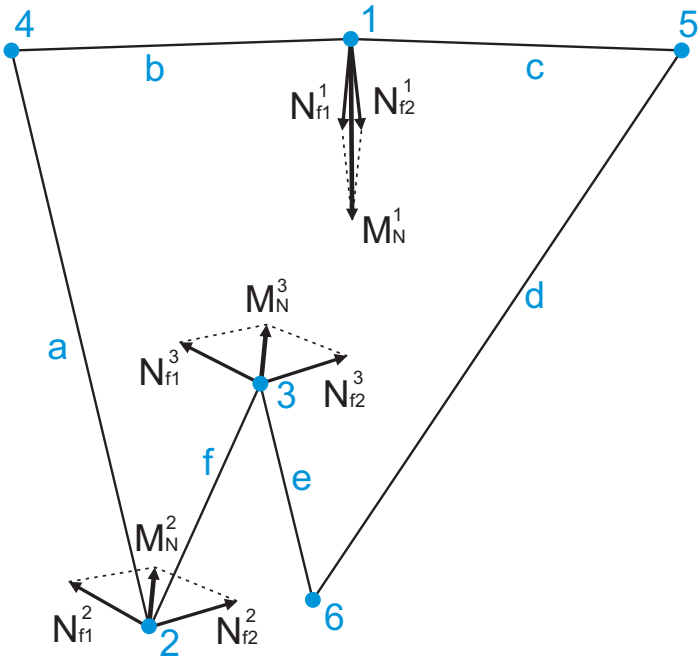


Rysunek 3.1. Błędy w siatce deformowanej przy pomocy M_N^i: siatka oraz rendering obiektu po 1500 cyklach. Widać trójkąty, które „wyszły" z wnętrza obiektu.

Jedną z wad takiej techniki obliczeń jest powstawanie błędów w siatce (rys. 3.1). Drugą jest nie spełnianie założenia **2.** Rysunek 3.2 przedstawia dwuwymiarowe analogie ilustrujące brak możliwości oceny nieregularności powierzchni na podstawie M_N^i:

— wektory $M_N^2 = \frac{N_{f_1}^2 + N_{f_2}^2}{2}$ i $M_N^3 = \frac{N_{f_1}^3 + N_{f_2}^3}{2}$ mają równe długości mimo, iż kąt między krawędziami (a, f) wychodzącymi z wierzchołka (2) jest wypukły, a kąt między krawędziami (f, e) wychodzącymi z wierzchołka (3) wklęsły,

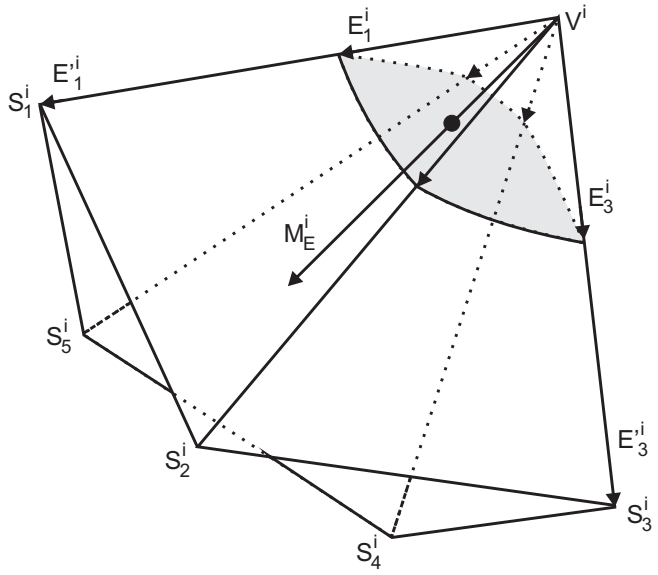
— wektor M_N^1 jest dłuższy od wektorów M_N^2 i M_N^3 mimo, iż wierzchołek 1 leży na prawie płaskiej powierzchni.



Rysunek 3.2. Wszystkie wektory M_N^i są skierowane do wnętrza obiektu 2D. Wektor M_N^1 jest najdłuższy, pozostałe krótsze. Długość nie zależy wprost od wysunięcia wierzchołka. Dwuwymiarowa analogia obiektu 3D – krawędzie w 2D reprezentują trójkąty w 3D.

Nie można więc tylko na podstawie długości M_N^i ocenić stopnia wypukłości bryły przy wierzchołku, ponieważ krótki wektor M_N^i może oznaczać zarówno dużą wypukłość, jak i dużą wklęsłość. Podobnie jest w przypadku trójwymiarowym kiedy z samej długości M_N^i nie można nawet ocenić, czy wierzchołek dla którego wektor jest obliczony tworzy wypukłość, czy wklęsłość.

Drugie podejście [84] opiera się na obliczaniu przesunięcia wierzchołków w taki sposób, aby skracały się wychodzące z nich krawędzie. Wektor przesunięcia M_E^i i-tego wierzchołka jest obliczany na podstawie krawędzi $E_j'^i$ wychodzących z tego wierzchołka (gdzie j – indeks krawędzi wychodzącej z i-tego wierzchołka) (rys. 3.3). Aby obliczyć wektor M_E^i należy wykonać operacje przedstawione poniżej.



Rysunek 3.3. Wektory E_j^i leżące wzdłuż krawędzi wychodzących z i-tego wierzchołka i ich suma M_E^i.

Obliczenie wektora przesunięcia M_E^i (rys. 3.3)

1. Obliczyć wektory $(E_j^{'i})$ leżące wzdłuż krawędzi wychodzących z i-tego wierzchołka (3.3):

$$E_j^{'i} = S_j^i - V^i, \tag{3.3}$$

gdzie:

V^i – położenie przetwarzanego wierzchołka,

S_j^i położenie wierzchołka na drugim końcu j-tej krawędzi wychodzącej z wierzchołka i.

2. Znormalizować wektory $(E_j^{'i})$ (3.4):

$$E_j^i = (E_j^{'i})^\circ. \tag{3.4}$$

3. Zsumować wektory (E_j^i) (3.5):

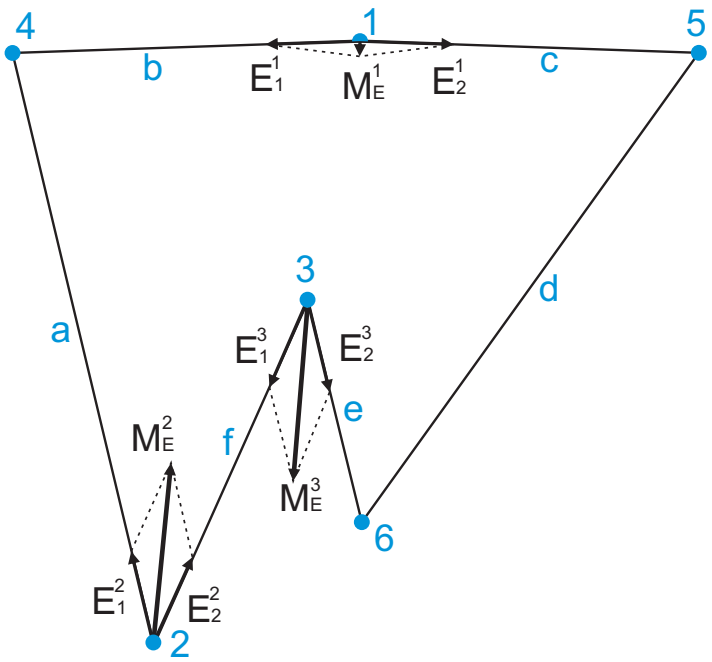
$$M_E^i = \frac{1}{n_e^i} \sum_{j=1}^{n_e^i} E_j^i = \sum_j (S_j^i - V^i)^\circ, \tag{3.5}$$

gdzie n_e^i – liczba krawędzi wychodzących z i-tego wierzchołka.

Tak obliczony wektor spełnia założenie **3.**, gdyż jest obliczany lokalnie na podstawie sąsiednich wierzchołków. Nie spełnia natomiast założenia **1.** ponieważ wektory E_j^i nie wskazują wnętrza obiektu i ich suma również nie ma tej cechy. Wektor M_E^i może wskazywać zarówno zewnętrze, jak i wnętrze obiektu. Spełnia założenie **2.**, co ilustruje rysunek 3.4 przedstawiający dwuwymiarową analogię obiektu 3D:

— wektor M_E^2 jest dłuższy od M_E^1, ponieważ jego wierzchołek tworzy większą wypukłość,

— wektor M_E^3 ma taką samą długość, jak M_E^2; ale z samych długości nie można wywnioskować, czy wierzchołek tworzy wypukłość, czy wklęsłość.



Rysunek 3.4. Wektor M_E^1 jest krótki, ponieważ wierzchołek (1) tworzy nieznaczną wypukłość. Wektory M_E^2 i M_E^3 są dłuższe i takiej samej długości mimo, iż wierzchołek (2) tworzy wypukłość, a (3) wklęsłość. Dwuwymiarowa analogia obiektu 3D – krawędzie w 2D reprezentują trójkąty w 3D.

Podobnie jest w przypadku trójwymiarowym: im większą wypukłość tworzy wierzchołek, tym dłuższy jest obliczony dla niego wektor M_E^i. Użycie wektora M_E^i nie powoduje błędów w deformowanej siatce, natomiast może doprowadzić do likwidacji wklęsłości w obiekcie – bryła zamiast „kurczyć się" może lokalnie wyrównywać wklęsłości, co jest efektem braku wymuszenia zwrotu M_E^i (nie spełniania założenia **1.**).

Mimo, iż oba przedstawione podejścia mają swoje wady, to odpowiednie użycie poprzez połączenie obu wektorów (M_N^i, M_E^i) utworzy wektor przesunięcia (M^i), który spełnia wszystkie trzy przyjęte założenia (**1.**, **2.** i **3.**) (3.6):

$$M^i = \begin{cases} k_E M_E^i + k_N M_N^i, & \text{gdy } M_N^i \cdot M_E^i > 0, \\ k_N M_N^i & \text{w przeciwnym przypadku,} \end{cases} \tag{3.6}$$

gdzie:

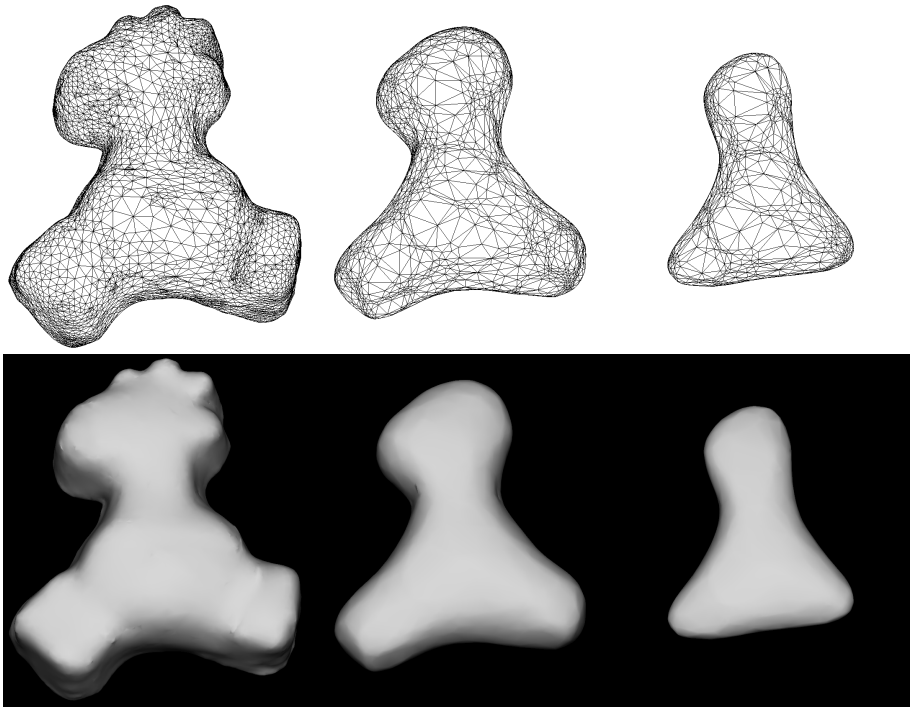
k_E – współczynnik przesunięcia wierzchołka wzdłuż M_E^i

k_N – współczynnik przesunięcia wierzchołka wzdłuż M_N^i.

Tak zdefiniowany wektor przesunięcia spełnia założenie **1.** w takim samym stopniu jak M_N^i ponieważ badana jest zgodność kierunków M_N^i i M_E^i. Jeżeli kąt między nimi jest mniejszy od kąta prostego (iloczyn skalarny dodatni), to M_E^i jest traktowany jako skierowany do wewnątrz obiektu i uwzględniany w wektorze M^i. W przeciwnym wypadku tylko wektor M_N^i bierze udział w przesunięciu i-tego wierzchołka. Użycie M_N^i powoduje wspomniane wcześniej błędy deformacji siatki, dlatego jego wkład w wektor M^i jest niewielki – eksperymenty wykazały, że współczynnik przesunięcia wzdłuż normalnej (k_N) powinien być znacznie mniejszy od współczynnika przesunięcia wzdłuż sumy wektorów krawędzi (k_E)[1]. Dzięki temu dla wierzchołków, dla których zachodzi zgodność kierunków M_N^i i M_E^i spełnione jest założenie **2.** Ponadto M^i jako ważona suma M_N^i i M_E^i spełnia założenie **3.** Tak obliczone wektory przesunięcia dodawane są do odpowiadających im wierzchołków w każdym cyklu obliczeń.

[1] W badanych bryłach $k_E \approx 100 k_N$.

Wynik działania algorytmu opartego na wyżej przedstawionych obliczeniach prezentuje rysunek 3.5.



Rysunek 3.5. Deformowana siatka: wejściowa, po 800 cyklach, po 1600 cyklach.

3.2 Upraszczanie siatki

Przesuwanie wierzchołków w kierunku wnętrza obiektu powoduje, że znajdują się one coraz bliżej siebie, łączące je krawędzie są coraz krótsze, a trójkąty coraz mniejsze. Ma to dwie konsekwencje:

— trójkąty, które są bardzo małe w stosunku do pozostałych wprowadzają zbyteczny narzut obliczeniowy w procesie renderingu,

— na podstawie bardzo małych trójkątów obliczane są wektory potrzebne do deformacji siatki (M_N^i, M_E^i), co powoduje coraz większe błędy wynikające z ograniczonej precyzji reprezentacji liczb.

O ile pierwsza konsekwencja wpływa tylko na czas wykonania poszczególnych cykli obliczeń o tyle druga powoduje nieprawidłowe wyniki już na początku wizualizacji procesu. Z każdym następnym cyklem błędy obliczeń powiększają się i prowadzą do błędów deformacji siatki.

Aby zapobiec temu zjawisku zastosowana została technika upraszczania siatki poprzez usuwanie krótkich krawędzi. Usuwanie zaczyna się od przeszukania siatki i znalezienia krawędzi, których długość jest mniejsza od zadanej minimalnej długości krawędzi – k_r (3.7):

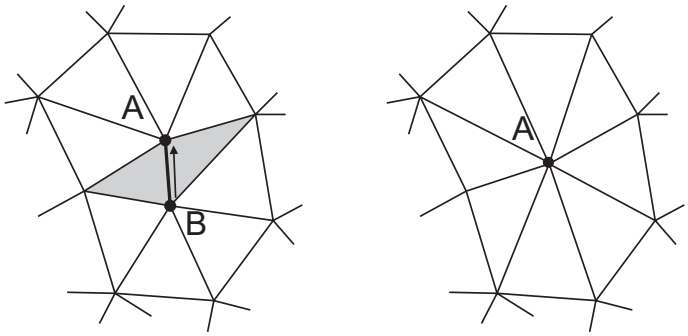
$$E_j^{'i} < k_r \tag{3.7}$$

gdzie j – indeks j-tej krawędzi i-tego wierzchołka.

Każda tak znaleziona krawędź zostaje usunięta w kilku krokach (rys. 3.6):

Usuwanie zbyt krótkich krawędzi (rys. 3.6)

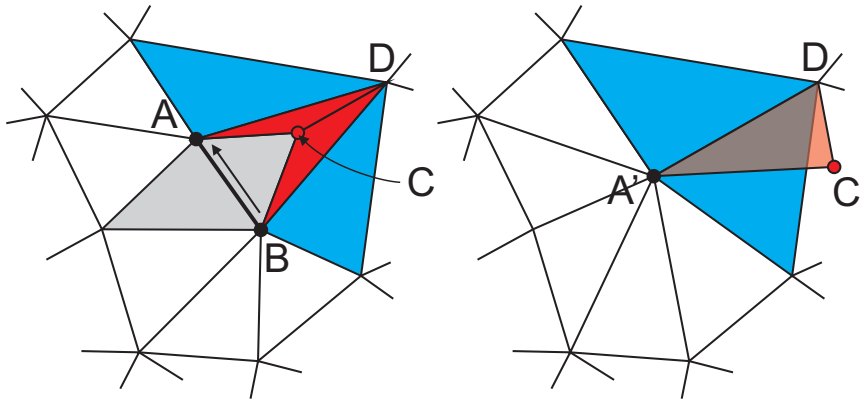
1. usunięcie dwóch trójkątów, dla których krawędź jest wspólna,
2. stworzenie nowego wierzchołka w połowie usuwanej krawędzi – atrybuty nowego wierzchołka są interpolowane liniowo z dwóch wierzchołków, które są końcami usuwanej krawędzi,
3. usunięcie dwóch wierzchołków będących końcami krawędzi,
4. zmiana indeksów wierzchołków w trójkątach zawierających usunięte wierzchołki na indeks nowego wierzchołka.



Rysunek 3.6. Usuwanie krawędzi: usunięty zostaje wierzchołek B na rzecz A – drugiego wierzchołka usuwanej krawędzi; usunięte są również dwa zaznaczone trójkąty.

Opisana technika stosowana jest po każdym cyklu obliczeń deformacji siatki opisanym w poprzednim podrozdziale. Tak upraszczana siatka nie generuje błędów wynikających z ograniczonej precyzji liczb i zmniejsza liczbę obliczeń potrzebnych do deformacji w miarę postępu wizualizowanego procesu. Dzięki temu im obiekt jest mniejszy, tym jego przetwarzanie zajmuje mniej czasu.

Zmiana topologii siatki powyższą techniką może spowodować w niektórych sytuacjach *sklejenie trójkątów*. Rysunek 3.7 pokazuje usunięcie krawędzi (AB), które może być ich wynikiem. Odbywa się to przez połączenie pary wierzchołków A i B w jeden A'. Sklejenie trójkątów (ACD) i (BDC) polega na tym, że trójkąty mają w jego wyniku te same trzy wierzchołki (lecz uporządkowane w odwrotnej kolejności: $A'CD$ i $A'DC$) i te same trzy krawędzie. Dodatkowo jeden z wierzchołków (A') ma przyporządkowane tylko dwa trójkąty, a jedna z krawędzi $(A'D)$ aż cztery trójkąty. Dlatego też po usunięciu krawędzi spełniających kryterium długości (3.7) siatka powinna być przeszukana w celu znalezienia błędu sklejenia trójkątów. Zignorowanie tych błędów powoduje błędy w następnych cyklach upraszczania siatki, które powodują nieprawidłowe działanie wszystkich algorytmów wchodzących w skład przetwarzania.

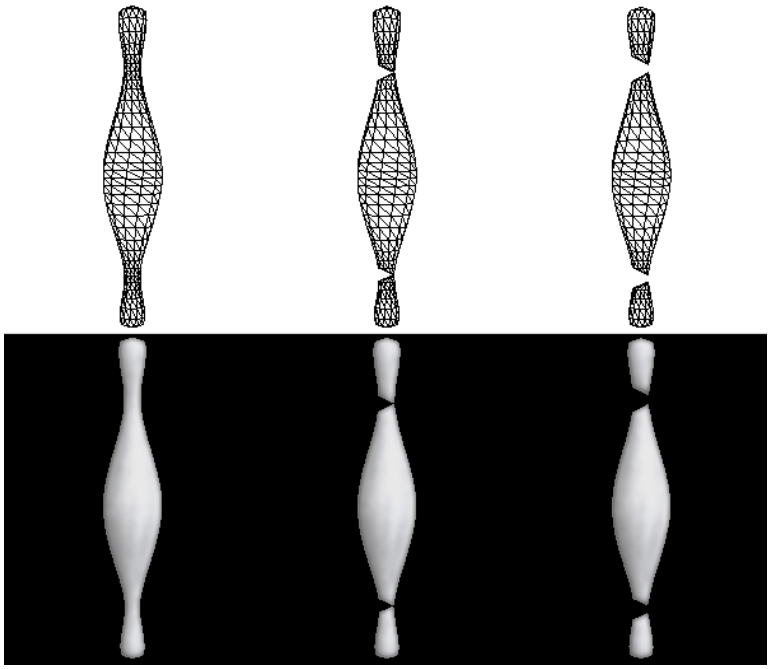


Rysunek 3.7. Sklejenie trójkątów ACD i BDC podczas usuwania krawędzi AB.

Szukanie i usuwanie *sklejonych trójkątów* (rys. 3.7)
1. dla każdego wierzchołka ustawić licznik zbudowanych na nim trójkątów na 0, **2.** dla każdego trójkąta zwiększyć liczniki jego wierzchołków o 1, **3.** wyszukać wierzchołki z licznikami równymi 2, powiązane z takim wierzchołkiem dwa trójkąty usunąć, usunąć również ten wierzchołek.

3.3 Podział obiektu

Geometria obiektu podczas zjawiska sublimacji deformuje się, a obiekt się „kurczy" tracąc szczegóły wyglądu powierzchni. Ponieważ jego kształt początkowy może być dowolny, a szybkość przesuwania powierzchni międzyfazowej jest zależna od nieregularności geometrii, może się zdarzyć, że obiekt mający przewężenia rozdzieli się na osobne części (rys. 3.8).



Rysunek 3.8. Podział obiektu na części: wynik trzech kolejnych cykli obliczeń.

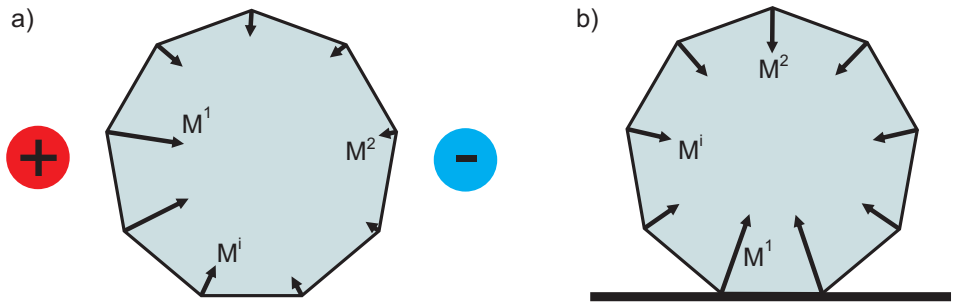
Charakter deformacji z podrozdziału (3.1) powoduje, że przewężenia obiektu pogłębiają się z każdym cyklem. Dzięki technice upraszczania siatki zaproponowanej w poprzednim podrozdziale coraz krótsze krawędzie łączące wierzchołki w przewężeniu zostają usunięte wraz z towarzyszącymi im trójkątami i wierzchołkami. Doprowadza to do sytuacji, kiedy przewężenie stanowią dwa sklejone trójkąty (w sensie błędu opisanego w poprzednim podrozdziale). W takim przypadku zostają one usunięte zgodnie z przedstawioną procedurą. Jest to też moment rozdzielenia pierwotnej siatki obiektu na dwie osobne siatki. Od tej chwili przetwarzaniu podlegają dwie siatki, które modelują dwa rozdzielne sublimujące lub topiące się obiekty. Siatki te spełniają warunek $\chi = 0$ dla wzoru (2.3) po odpowiednim zwiększeniu wartości S.

3.4 Czynniki fizyczne

Wstęp do niniejszej monografii zawiera listę czynników fizycznych, od których zależy przebieg i szybkość procesów przemian fazowych, między innymi ciepło topnienia, ilość substancji i temperaturę. Celem przedstawionych metod jest wizualizacja a nie symulacja, dlatego czynniki te mogą być zastąpione przez zespół parametrów dobranych doświadczalnie tak, by wizualizacja była dla widza jak najbardziej wiarygodna. Główną rolę odgrywa tu szybkość poruszania się powierzchni międzyfazowej. Jest ona odbierana jako szybkość przebiegu zjawiska. Sterowanie tą szybkością poprzez dobór współczynników przebiegu deformacji pozwoli na naśladowanie wpływu zewnętrznych czynników fizycznych bez wnikania w ich rzeczywiste pochodzenie i charakter.

Zależność szybkości sublimacji od temperatury otoczenia reprezentuje globalny współczynnik szybkości topnienia $k_g \in < 0,1 >$. Dla uzyskania większej elastyczności przebiegu wizualizacji oprócz globalnego współczynnika wprowadzony został lokalny współczynnik szybkości topnienia k_l. Pozwala on wizualizować sytuacje, gdy jedna część obiektu znajduje się pod

wpływem innych czynników niż pozostała (rys. 3.9). Na przykład, gdy otoczenie obiektu charakteryzuje się nierównomiernym rozkładem temperatur (strumień ciepłego powietrza, nasłonecznienie itp.). Ponadto pozwala zaimplementować zależność temperatury topnienia od dodatkowego czynnika fizycznego jakim jest ciśnienie, włączając w to nacisk innych obiektów lub obiektu na podłoże pod wpływem własnej masy w polu grawitacyjnym. Współczynnik k_l zależy od położenia w przestrzeni, więc jest on w zasadzie funkcją $k_l : R^3 \rightarrow < 0, 1 >$ (polem skalarnym). Dla celów wizualizacji może on być dany analitycznie lub dyskretnie, na przykład jako zbiór danych zapisanych w tablicy trójwymiarowej wraz z funkcją odczytującą dane z tej tablicy.



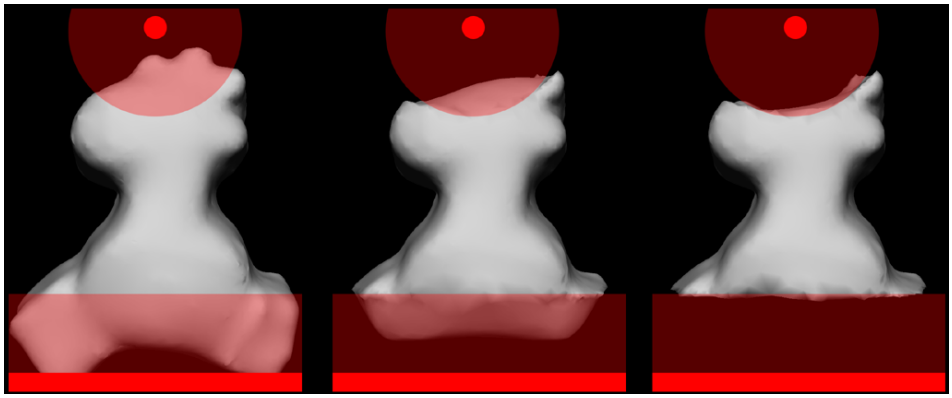
Rysunek 3.9. Wpływ zewnętrznych czynników lokalnych na długość wektora przesunięcia: a) wpływ rozkładu temperatury – M^1 jest dłuższy od M^2, ponieważ leży w cieplejszym obszarze; b) wpływ ciśnienia – M^1 jest dłuższy od M^2, ponieważ na dolną powierzchnię bryły działa siła nacisku.

Tak zaproponowane współczynniki szybkości przesunięcia powierzchni międzyfazowej zostały użyte przy obliczaniu nowych położeń $V^{i(nowe)}$ wierzchołków o pierwotnym położeniu V^i w kolejnych cyklach deformacji siatki (3.8):

$$V^{i(nowe)} = V^i + k_g k_l(V^i) M^i \tag{3.8}$$

Przeskalowany wektor M^i użyty do przesunięcia wierzchołka w nowe położenie spełnia założenie **4.**

Rysunek 3.10 przedstawia obiekt topiony z uwzględnieniem lokalnego współczynnika szybkości topnienia.



Rysunek 3.10. Obiekt topiony z uwzględnieniem lokalnego współczynnika szybkości topnienia, który modeluje punktowe źródło ciepła oraz płaski nacisk podłoża. Zaznaczone przezroczyste obszary pokazują zasięg działania każdego źródła.

3.5 Reprezentacja deformowalnej siatki trójkątów

Reprezentacja siatki trójkątów użytej do implementacji opisanych wcześniej technik powinna umożliwiać szybki dostęp do poszczególnych jej składników, który jednocześnie nie jest okupiony złożonością zależności między nimi. Reprezentacja winged-edges, która umożliwia przejście w jednym kroku w obie strony między wierzchołkiem a sąsiadującą krawędzią, między krawędzią a sąsiadującym trójkątem, jest nieefektywna, gdy zmienia się topologia siatki, gdyż wymaga zmian wpisów we wszystkich trzech tablicach. Reprezentacja krawędziowa również wymaga utrzymywania spójności trzech tablic. Reprezentacja indeksowa jest dosyć wymagająca pod względem wyszukiwania sąsiednich elementów strukturalnych, lecz bardzo łatwo można usunąć elementy wytypowane przez algorytmy. W przedstawionej metodzie

używane są dwa rodzaje operacji usuwania: usuwanie krawędzi oraz usuwanie sklejonych trójkątów. Na usunięcie krawędzi opisane w podrozdziale 3.2 składa się:

- usunięcie wpisu reprezentującego krawędź z tablicy krawędzi (jeśli taka tablica istnienie w danej reprezentacji),
- usunięcie wpisu reprezentującego wierzchołek będący jednym z końców krawędzi oraz zmianę atrybutów drugiego wierzchołka – końca krawędzi,
- usunięcie z tablicy trójkątów dwóch wpisów reprezentujących dwa trójkąty zbudowane na tej krawędzi,
- przeindeksowanie (zmiana) n wpisów reprezentujących trójkąty lub krawędzie (jeśli istnieją w danej reprezentacji) odwołujących się do usuniętego wierzchołka.

Na usunięcie sklejonych trójkątów opisane w podrozdziale 3.2 składa się:

- usunięcie dwóch wpisów reprezentujących dwa trójkąty w tablicy trójkątów,
- usunięcie jednego wpisu reprezentującego wierzchołek w tablicy wierzchołków.

Tabela 3.1 zestawia liczbę potrzebnych usunięć lub zmian wpisów w poszczególnych tablicach, aby reprezentacja siatki była spójna. Ponieważ usunięcie krawędzi techniką opisaną w podrozdziale 3.2 realizowane jest przez „sklejenie" wierzchołków: jeden jest usuwany, a drugi przesuwany, to wszystkie trójkąty lub krawędzie (zależnie od reprezentacji), które indeksują usunięty wierzchołek powinny być zmodyfikowane tak, aby indeksowały ten wierzchołek, który pozostał. Liczba tych trójkątów lub krawędzi jest równa n. Przy reprezentacji bezpośredniej należy dodatkowo wziąć pod uwagę konieczność wyszukania redundantnych wierzchołków (przejście całej tablicy). Symbol „–" oznacza brak tablicy elementów tego rodzaju w rozpatrywanej reprezentacji.

Bazując na tym zestawieniu oraz na zasadzie, że im mniej tablic trzeba obsłużyć, tym kod implementacji będzie prostszy podjęto decyzję o zastosowaniu reprezentacji indeksowej do przechowywania siatki deformowanej przez opracowane algorytmy.

Tabela 3.1. Zestawienie liczby operacji na tablicach według reprezentacji.

operacja	reprezentacja	wierzchołki		krawędzie		trójkąty	
		usuń	zmień	usuń	zmień	usuń	zmień
usunięcie krawędzi	bezpośrednia	–		–		2	n
	indeksowa	**1**	**1**	–		**2**	**n**
	krawędziowa	1	1	1	n	2	0
	winged-edges	1	1	1	n	2	0
usunięcie sklejonych trójkątów	bezpośrednia	–		–		2	0
	indeksowa	**1**	**0**	–		**2**	**0**
	krawędziowa	1	0	2	0	2	0
	winged-edges	1	0	2	0	2	0

Reprezentacja ta zawiera dane zapisane w dwóch tablicach: wierzchołków i trójkątów. Tablice są stałej długości. Usunięcie elementu zapisanego w tablicy odbywa się poprzez ustawienie w tym elemencie odpowiedniej flagi. Fizycznie element ten będzie istniał, lecz nie będzie przetwarzany przez algorytmy. Dzięki temu nie trzeba przeindeksowywać tablic i zmieniać ich długości. Sprzyja to optymalizacji algorytmów pod kątem szybkości przetwarzania.

Aby spełnić wymagania algorytmów reprezentacja przechowuje dodatkowe informacje powiązane z wierzchołkami i trójkątami:

a) tablica wierzchołków ($V[]$):
   — flaga istnienia wierzchołka (e:`bool`): gdy $e = 1$, to wierzchołek istnieje, gdy $e = 0$, to nie istnieje; usuwanie wierzchołka odbywa się przez wyzerowanie tej flagi,
   — wektor pozycji wierzchołka ($P = (x, y, z)$:`3×float`) w lokalnym układzie współrzędnych

- wektor normalny (N:`3×float`),
- wektor przesunięcia (M:`3×float`),
- liczba trójkątów zbudowanych na wierzchołku (t:`int`),
- pozycja przeindeksowania (i:`int`), potrzebna przy łączeniu wierzchołków podczas usuwania krawędzi w przedstawionym w następnym podrozdziale Algorytmie 2,

b) tablica trójkątów ($T[]$):

- flaga istnienia trójkąta (e:`bool`): gdy $e = 1$, to trójkąt istnieje, gdy $e = 0$, to nie istnieje; usuwanie trójkąta odbywa się przez wyzerowanie tej flagi,
- uporządkowane indeksy trzech wierzchołków ((a, b, c):`3×int`) w tablicy wierzchołków,
- wektor normalny trójkąta (N_f:`3×float`).

Taka reprezentacja umożliwia przejście w jednym kroku od trójkąta do trzech wierzchołków, które go reprezentują. Staje się natomiast niewygodna przy innych operacjach i pozornie może wymagać wielu przeszukań obu tablic. Okazało się jednak, że implementacja zaprezentowanych w niniejszym rozdziale algorytmów nie jest kosztowna.

3.6 Algorytmy przetwarzania siatki trójkątów

Poniżej zaprezentowano algorytm deformacji siatki (Algorytm 1). Obliczane są po kolei normalne trójkątów N_f (linie $3 - 5$) i nieznormalizowane normalne wierzchołków N (linie $6 - 13$). Następnie normalne wierzchołków są normalizowane oraz zerowane są wartości M oraz t (linie $14 - 19$), a później obliczane są wektory wchodzące w skład sumy M (linie $20 - 30$). Na końcu wierzchołki przesuwane są zgodnie z obliczonymi wektorami przesunięcia (linie $32 - 35$).

Algorytm 1 Deformacja siatki

inputs
  tablica wierzchołków V[], tablica trójkątów T[]
for all istniejącego trójkąta **do**
  {oblicz normalną trójkąta}
  $N_f = ((V[b].P - V[a].P) \times (V[c].P - V[a].P))^\circ$
for all istniejącego wierzchołka **do**
  {wyzeruj normalną}
  $N = [0, 0, 0]$
for all istniejącego trójkąta **do**
  {do normalnej każdego wierzchołka trójkąta dodaj normalną trójkąta}
  $V[a].N+ = N_f$
  $V[b].N+ = N_f$
  $V[c].N+ = N_f$
for all istniejącego wierzchołka **do**
  {znormalizuj normalną}
  $N = N^\circ$
  {wyzeruj wektor przesunięcia oraz liczbę trójkątów}
  $M = [0, 0, 0]$
  $t = 0$
for all istniejącego trójkąta **do**
  {do wektora M każdego wierzchołka dodaj wektor leżący wzdłuż krawędzi i zwiększ licznik krawędzi}
  $V[a].M+ = (V[b].P - V[a].P)^\circ$
  $V[a].t+ = 1$
  $V[b].M+ = (V[c].P - V[b].P)^\circ$
  $V[b].t+ = 1$
  $V[c].M+ = (V[a].P - V[c].P)^\circ$
  $V[c].t+ = 1$
for all istniejącego wierzchołka **do**
  {podziel sumę M przez liczbę krawędzi}
  $M/ = t$
  {przesuń w nowe położenie}
  if $N \cdot M^\circ \leqslant 0$ **then**
    $P+ = (k_E * M - k_N * N) * k_g * k_l(P)$
  else
    $P- = k_N * N * k_g * k_l(P)$

Do usuwania krawędzi siatki służy Algorytm 2, który opiera się na znalezieniu wierzchołków będących końcami krawędzi krótszych od k_r. Każda taka para tworząca krawędź musi zostać zastąpiona jednym wierzchołkiem. Realizowane jest to w ten sposób, że wierzchołek znalezionej krawędzi **o wyższym indeksie** w tablicy wierzchołków powinien zostać usunięty ($e = 0$), a ten **o niższym indeksie** powinien być przesunięty i zastąpić usunięty wierzchołek we wszystkich trójkątach, które go używały. Kryterium indeksu zostało wprowadzone, ponieważ każda krawędź, która spełni warunek usuwania (3.7) spełni go dwa razy, gdyż będzie wykryta w każdym ze swoich dwóch trójkątów. Pozwala to uniknąć niejednoznaczności przy usuwaniu wierzchołka. Aby oznaczyć wierzchołki przeznaczone do usunięcia oraz ich wierzchołki „z pary" wprowadzono do tablicy wierzchołków pozycję przeindeksowania (i). Wierzchołek, który zostaje w danym cyklu usunięty posiada i równe indeksowi wierzchołka, który go zastąpi. Dzięki temu można łatwo identyfikować zarówno trójkąty, których wierzchołki należy przeindeksować, jak i wierzchołki, które należy usunąć.

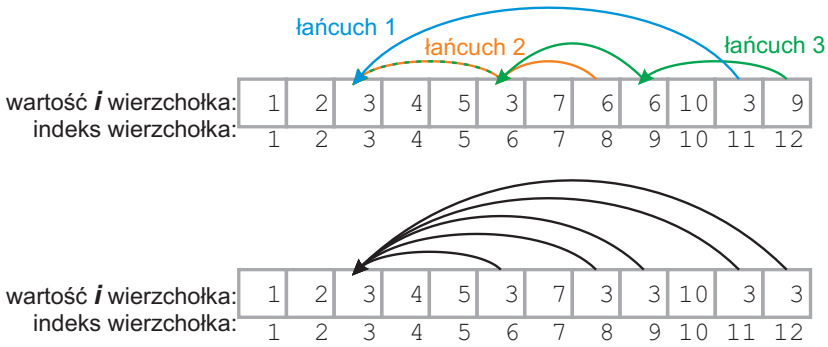
W liniach $7-15$ Algorytmu 2 zastosowano optymalizację liczby instrukcji warunkowych. W przypadku, gdy krawędź wychodząca z pierwszego indeksowanego przez trójkąt wierzchołka (a) jest krótsza od k_r, to nie jest porównywany jego indeks (a) z indeksem drugiego wierzchołka (b lub c). Dzieje się tak dlatego, że indeksy wierzchołków w trójkątach są posortowane w ten sposób, że pierwszy przechowywany indeks jest zawsze najmniejszy. Sortowanie to nie powinno zmienić orientacji trójkąta zapisanej zgodnie z konwencją opierającą się na kolejności rysowania wierzchołków. Posortowanie indeksów wierzchołków w trójkątach powinno być zapewnione przed pierwszym cyklem wykonania algorytmów, a zawarte jest w Algorytmie 3.

W liniach $16-20$ Algorytmu 2 jeżeli przykładowy wierzchołek ma zostać przeindeksowany na inny wierzchołek, który ma być przeindeksowany na jeszcze inny, to jego wartość i należy przekierować na ostatni wierzchołek w takim *łańcuchu przeindeksowań*. Należy przeindeksować wszystkie wierzchołki na takie, które nie są już dalej przeindeksowywane. Rysunek 3.11

Algorytm 2 Usuwanie krawędzi

inputs
  wierzchołki V[], trójkąty T[] z posortowanymi indeksami wierzchołków
for all istniejącego wierzchołka **do**
  i =indeks {przypisz i indeks tego wierzchołka w tablicy wierzchołków}
for all istniejącego trójkąta **do**
  {wykryj krótką krawędź i przypisz odpowiedniemu wierzchołkowi z pary indeks drugiego wierzchołka krawędzi}
  if odległość $(V[a], V[b]) < k_r$ **then**
    V[b].i = a
  else if odległość $(V[b], V[c]) < k_r$ **then**
    if $b < c$ **then**
      $V[c].i = b$
    else
      $V[b].i = c$
  else if odległość $(V[c], V[a]) < k_r$ **then**
    $V[c] = a$
for all istniejącego wierzchołka **do**
  zmienna tymczasowa: j = i {usuń *łańcuchy przeindeksowań*}
  while $j \neq V[j].i$ **do**
    $j = V[j].i$
  $i = j$
for all istniejącego wierzchołka **do**
  $V[i].P = (P + V[i].P)/2$ {przesuń wierzchołki z par, które nie będą usunięte}
  if i $\neq$ indeks **then**
    $e = 0$ {usuń odpowiednie wierzchołki}
for all istniejącego trójkąta **do**
  {przeindeksuj te wierzchołki w trójkącie, które mają zostać usunięte, na wierzchołki z ich pary}
  if $a \neq V[a].i$ **then**
    $a = V[a].i$
  if $b \neq V[b].i$ **then**
    $b = V[b].i$
  if $c \neq V[c].i$ **then**
    $c = V[c].i$
for all istniejącego trójkąta **do**
  {jeżeli dwa indeksy w trójkącie pokrywają się, to zostaje on usunięty}
  if $a = b$ **or** $b = c$ **or** $c = a$ **then**
    $e = 0$

pokazuje tablicę z trzema przykładowymi *łańcuchami przeindeksowań* i tę samą tablicę po ich usunięciu. Pętla w tym kroku skończy się na wierzchołku, którego wartość i jest równa jego indeksowi.



Rysunek 3.11. Likwidacja *łańcuchów przeindeksowań*.

W linii 22 każdy wierzchołek, który ma być usunięty uśrednia swoją pozycję z wierzchołkiem z pary, na rzecz którego będzie usunięty. Wierzchołek, który nie będzie usuwany uśrednia swoją pozycję sam ze sobą, więc nic się nie zmienia.

Algorytm 3 – usuwania sklejonych trójkątów, zlicza trójkąty zbudowane na poszczególnych wierzchołkach i po wykryciu sklejonych trójkątów usuwa je i związany z nimi wierzchołek. Ponadto w liniach $19 - 33$ sortowane są indeksy wierzchołków w trójkątach.

Algorytmy przystosowane są do korzystania z reprezentacji indeksowej: przechodzą albo całą tablicę wierzchołków, albo trójkątów, a przy przechodzeniu tablicy trójkątów odwołują się do indeksowanych przez nie wierzchołków. Reprezentacja indeksowa umożliwia wykonanie tych właśnie operacji bezpośrednio. Dodatkowo przyjęty model usuwania elementu jest również bardzo prosty przy jej zastosowaniu.

Dzięki rozłożeniu cyklu przetwarzania na szereg niezależnych przetwarzań tablic wierzchołków lub trójkątów, możliwe było uzyskanie $O(n)$ – liniowej czasowej złożoności obliczeniowej algorytmów.

Algorytm 3 Usuwanie sklejonych trójkątów

inputs
    tablica wierzchołków V[], tablica trójkątów T[]
for all istniejącego wierzchołka **do**
    {wyzeruj liczbę trójkątów}
    $t = 0$
for all istniejącego trójkąta **do**
    {zwiększ licznik trójkątów każdego indeksowanego wierzchołka}
    $V[a].t+ = 1$
    $V[b].t+ = 1$
    $V[c].t+ = 1$
for all istniejącego trójkąta **do**
    {jeżeli którykolwiek z wierzchołków trójkąta ma przypisane tylko dwa trójkąty, to znaczy, że trójkąt jest sklejony z innym i trzeba go usunąć}
    if $V[a].t = 2$ **or** $V[b].t = 2$ **or** $V[c].t = 2$ **then**
        $e = 0$
for all istniejącego wierzchołka **do**
    {jeżeli wierzchołek ma przypisane tylko dwa trójkąty, to znaczy, że są one sklejone i trzeba go usunąć}
    if t $= 2$ **then**
        $e = 0$
for all istniejącego trójkąta **do**
    {posortuj wierzchołki w trójkącie}
    zmienna tymczasowa $min = 0$
    zmienna tymczasowa $old\_b = b$
    if $a < b$ **then**
        if $a > c$ **then**
            $min = 2$
    else if $b < c$ **then**
        $min = 1$
    else
        $min = 2$
    if $min = 1$ **then**
        $b = c, c = a, a = old\_b$
    else if $min = 2$ **then**
        $b = a, a = c, c = old\_b$

3.7 Podsumowanie

Niniejszy rozdział prezentuje zbiór technik i powstałych na ich bazie algorytmów, których celem jest takie modyfikowanie siatki obiektu, aby jego zmieniająca się geometria naśladowała zachowanie się geometrii rzeczywistego obiektu podczas sublimacji. Omówiony został algorytm deformacji siatki, jej upraszczania i korekty. Omówiony algorytm deformacji uwzględnia poprzez zbiór współczynników globalne i lokalne czynniki fizyczne wpływające na przebieg wizualizowanego zjawiska. Algorytmy uwzględniają zmianę topologii siatki wraz z możliwością jej rozłączania tak, by nadal w poprawny sposób reprezentowała rozłączne części bryły.

Zaprezentowane zostały algorytmy o liniowej czasowej złożoności obliczeniowej, które przetwarzają w całości tablicę wierzchołków lub trójkątów. Dzięki temu algorytmy są w większości gotowe do implementacji na procesorze graficznym opisanej w rozdziale 5.

4

Programowanie jednostek graficznych

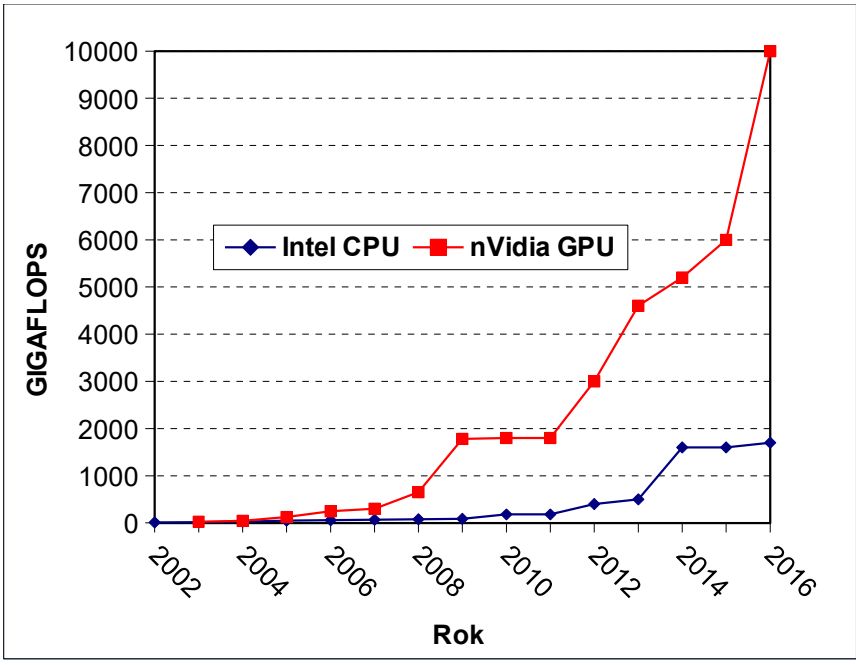
Na przestrzeni ostatnich kilkunastu lat miał miejsce ogromny wzrost mocy obliczeniowej komputerów PC. Jednak nie było to efektem zwiększania szybkości procesorów głównych, a raczej efektem umieszczenia coraz mocniejszych procesorów na kartach graficznych. Niniejszy rozdział pokazuje, jak można wykorzystać ten trend do przypieszenia działania przedstawianych algorytmów.

4.1 Technologie GPGPU

Procesory graficzne już od chwili wprowadzenia na rynek w roku 1999[1] wykazywały większe moce obliczeniowe i szybszy ich przyrost roczny niż CPU (rys. 4.1).

Spowodowało to wzrost zainteresowania tymi jednostkami nie tylko zgodnie z ich przeznaczeniem, to znaczy do przyspieszenia renderingu. Zaczęto poszukiwać możliwości ich zastosowania w innych zadaniach obliczeniowych. Powstała idea GPGPU (ang. General-Purpose Computation on Graphics

[1] 31. sierpnia 1999 pojawiła się w sprzedaży karta graficzna nVidia GeForce 256.



Rysunek 4.1. Zmiana liczby operacji zmiennoprzecinkowych na sekundę na przestrzeni ostatnich lat – porównanie CPU i GPU (*źródło: na podstawie* `http://intel.com`, `http://www.gpureview.com`, `http://www.karlrupp.net`).

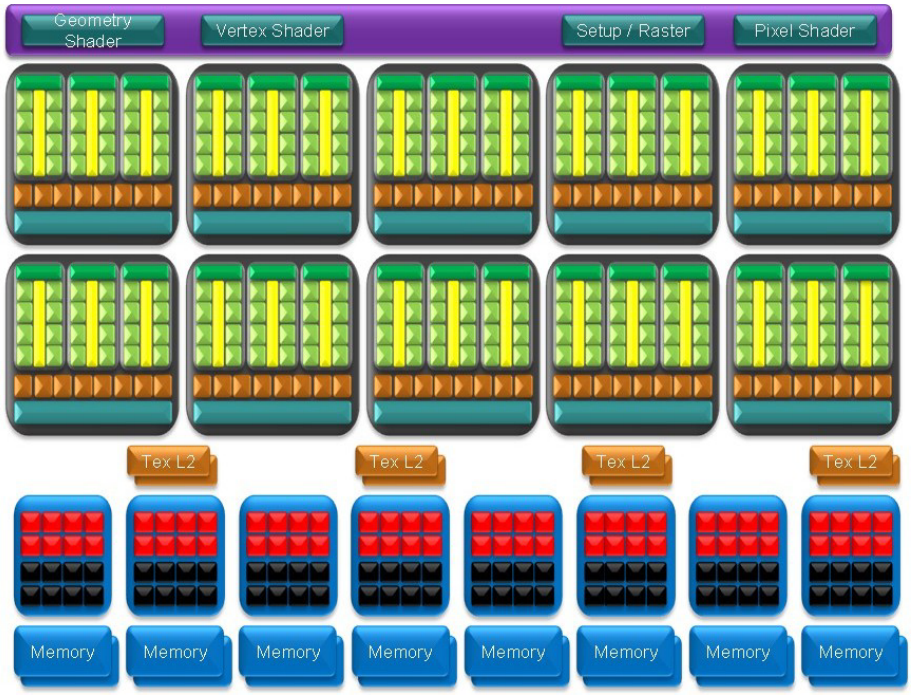
Processing Unit) – programowania zadań ogólnego przeznaczenia z wykorzystaniem GPU. Kilka ostatnich lat było czasem bardzo szybkiego rozwoju GPGPU od prostego wykorzystania programowalnego potoku renderingu do specjalizowanych interfejsów pozwalających implementować na GPU dowolne algorytmy w językach wysokiego poziomu [110]:

1. programowanie z wykorzystaniem API[2] biblioteki graficznej, przetwarzanie równoległe implementowane w programach cieniowania fragmentów w asemblerze lub w językach wysokiego poziomu: Cg, GLSL, HLSL,
2. języki wysokiego poziomu zaproponowane przez naukowców: Sh (University of Waterloo) i BrookGPU (Stanford University),

[2] API (ang. Application Programming Interface) – interfejs programowania aplikacji.

3. interfejsy zaproponowane przez firmy komercyjne: nVidia CUDA, AMD Stream, Khronos Group OpenCL, które pozwalają wykorzystać nawet do 2560 rdzeni procesora GPU.

Dwa najpopularniejsze obecnie podejścia w badaniach naukowych, to wykorzystanie CUDA (np. [73, 91, 79, 29]) i programów przetwarzania fragmentów w programowalnym potoku renderingu (np. [24, 103, 67, 82]). GPU



Rysunek 4.2. Architektura procesora graficznego GeForce GTX 280 (*źródło: [109]*).

cechuje wysoka wydajność i architektura pozwalająca na równoległe przetwarzanie danych (rys. 4.2). Duże przyspieszenia wykonania w stosunku do implementacji na CPU uzyskuje się mapując na GPU algorytmy charakteryzujące się wysokim udziałem operacji, które mogą być wykonane równolegle w rozwiązaniu danego problemu. Ilustruje to prawo Amdahla [2]:

przyspieszenie działania programu (A), przez mapowanie jego części na przetwarzanie równoległe wyrażone jest wzorem 4.1:

$$A = \frac{1}{(1-P) + \frac{P}{N}}, \tag{4.1}$$

gdzie:

P – udział operacji, które można wykonać równolegle,

N – liczba równoległych potoków, na które podzielony zostanie problem.

Jeśli w rozwiązaniu pewnego problemu $P = 0.6$, to znaczy 60% operacji można wykonać równolegle i jest do dyspozycji $N = 5$ procesorów, to $A \approx 1,92$, czyli można osiągnąć prawie dwukrotne przyspieszenie wykonania programu. Gdy $N \to \infty$, to $A \to 2,5$. Jest to największe teoretyczne przyspieszenie, jakie można osiągnąć dzięki przetwarzaniu równoległemu w przypadku tego problemu.

Jedną z klas zadań, które charakteryzują się wysokimi wartościami P są problemy rozwiązywane przy pomocy przetwarzania strumieniowego (ang. stream processing). Dane dla takich zadań są strumieniem, czyli zbiorem danych o takiej samej strukturze. Przetwarzanie polega na uruchamianiu serii operacji (kerneli, ang. kernel functions) dla każdego elementu strumienia, a wyjściem jest kolejny strumień. Strumieniowanie jednorodne (ang. uniform streaming) polega na uruchamianiu jednego kernela dla wszystkich elementów strumienia – jest to typowe użycie. Obliczenia wykonywane przez każdy kernel na danym elemencie strumienia są niezależne od innych elementów. Kernele nie komunikują się między sobą – cechuje je **bezkontekstowość operacji**. Dzięki takim restrykcjom przetwarzanie strumieniowe jest odmianą przetwarzania równoległego, które jest dużo prostsze od innych podejść równoległych. Oprogramowanie i sprzęt mogą być dużo prostsze, jeśli nie muszą uwzględniać zarządzania wątkami, ich komunikacją i synchronizacją, jak to jest w klasycznym przetwarzaniu równoległym, na przykład na wielordzeniowych CPU. W przetwarzaniu strumieniowym rozwiązanie problemu podzielone jest na etapy przetwarzania, które reprezentowane są

przez ciąg kerneli połączonych strumieniami. Strumień wyjściowy danego kernela jest strumieniem wejściowym następnego.

Pokrewnym do przetwarzania strumieniowego podejściem do obliczeń równoległych jest SIMD (ang. single instruction, multiple data). Jest to technika przetwarzania pozwalająca na uzyskanie równoległości na poziomie danych. Polega na przetwarzaniu większej liczby danych przy pomocy jednej instrukcji. Typowym przykładem są tu procesory wektorowe, dla których pojedynczy rejestr nie jest liczbą, lecz wektorem liczb. Na przykład procesor główny konsoli gier Nintendo Gamecube może operować na *paired singles*, czyli rejestrach 2×32-bitowych, przechowujących dwie niezależne dane przetwarzane jedną instrukcją.

Procesory graficzne łączą opisane wyżej cechy, to znaczy strumieniowo przetwarzają dane wektorowe. Dzięki temu pierwszy cel ich istnienia – implementacja programowalnego potoku renderingu, jest bardzo efektywnie przez nie realizowany. Etapy przetwarzania wierzchołków i geometrii operują na wektorowych danych reprezentujących między innymi współrzędne przestrzenne wierzchołków, wektory normalne i współrzędne teksturowania. W programach nazywanych programami cieniowania (ang. shaders) dane te reprezentują rejestry będące wektorami czterowymiarowymi (x, y, z, w). Pozwala to na obliczenia z użyciem współrzędnych jednorodnych, a także na inne obliczenia, które wymagają wektorów trój- i dwuwymiarowych a także skalarów[3]. Z kolei etap przetwarzania fragmentów operuje na wektorach otrzymanych podczas rasteryzacji z interpolacji oraz na kolorach, które reprezentowane są najczęściej w modelu RGB z kanałem przezroczystości Alpha (A). Zatem są to wektory czterowymiarowe: (R, G, B, A). Strumieniami dla programów cieniowania są bufory wierzchołków (VBO, ang. Vertex Buffer Object) zawierające wektorowe dane powiązane z wierzchołkami (np. siatki i ich atrybuty) oraz tekstury będące zbiorami tekseli (tzn.

[3]nie wszystkie składowe wektora muszą być używane w obliczeniach, lecz na przykład obliczenia na skalarach uważa się za mało efektywne, jeśli można ich uniknąć przez inną budowę programu.

pikseli tekstury) o kolorach wyrażonych wektorami jedno-, dwu-, trój- lub czterowymiarowymi.

Algorytmy mapowane na GPU są zbiorem kerneli przetwarzających strumienie wektorowych danych. Dzięki temu współczynnik P we wzorze 4.1 jest stosunkowo wysoki, gdyż nierównoległe w algorytmach są tylko czynności związane z przygotowaniem przez CPU strumieni danych i kerneli dla GPU oraz organizacja sekwencji ich wykonania. Ponadto dobre warunki wykonania tych algorytmów na GPU zapewnia duży współczynnik N. Liczba rdzeni w karcie graficznej nVidia GeForce GTX-295 wynosi 2×240. Pozwala to osiągnąć duże przyspieszenia działania tych algorytmów zgodnie z prawem Amdahla, w stosunku do podejścia szeregowego na CPU. Dodatkowo mapowanie problemu na przetwarzanie strumieniowe jest łatwiejsze niż mapowanie na uogólnione równoległe przetwarzanie na CPU.

4.2 GPGPU realizowane przez programowalny potok renderingu

Możliwości przetwarzania danych niezwiązanych z renderingiem przez programowalny potok renderingu zwiększają się wraz z kolejnymi wersjami GPU obsługującymi coraz nowsze modele cieniowania (ang. Shader Model). Model cieniowania jest definicją programów cieniowania, dostępnych w nich funkcji, ich parametrów, liczby rejestrów i innych cech. Porównanie najważniejszych z nich zawiera tabela 4.1 umieszczona w podsumowaniu niniejszego rozdziału.

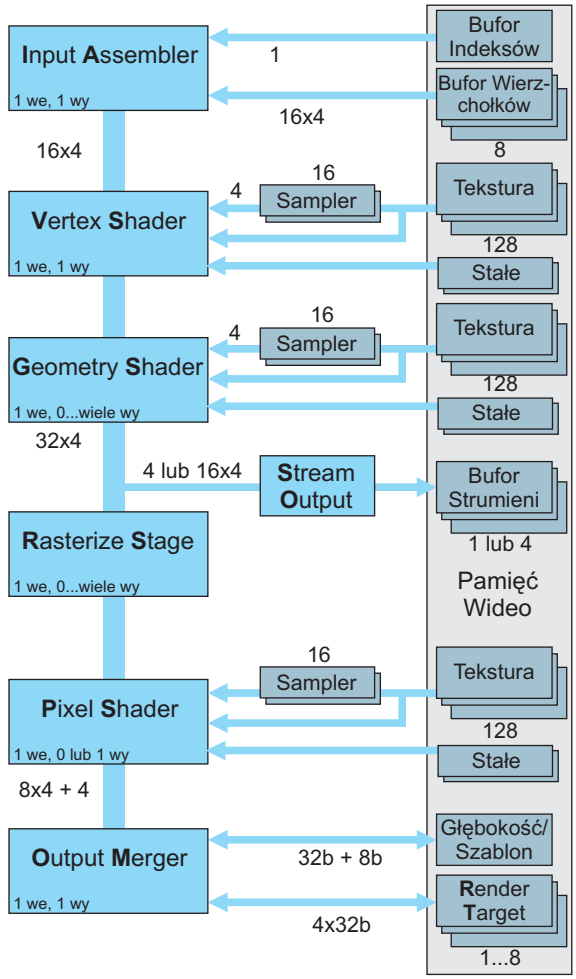
Możliwości definiowane w kolejnych modelach cieniowania bardzo szybko rosną. Widać to zwłaszcza po możliwościach programów cieniowania fragmentów, które są najbardziej przydatne w przetwarzaniu danych ogólnego przeznaczenia.

4.2.1 Model cieniowania 4.0

Specyfikacja potoku renderingu oferowanego przez model cieniowania (SM 4.0) znacznie rozszerza programowalny potok renderingu oferowany przez poprzednie wersje modeli cieniowania. Zostały do niego dodane nowe etapy, a istniejące zostały uogólnione (rys. 4.3). Jest to kolejny krok w kierunku unifikacji programów cieniowania (ang. unified shader model). Unifikacja ta polega na udostępnieniu możliwie takiego samego zbioru instrukcji, a więc daniu programiście tych samych możliwości przy programowaniu każdego typu programów cieniowania. Niezależnie od tego, czy jest to program cieniowania wierzchołków, geometrii, czy fragmentów, może on odczytywać tekstury, bufory danych i wykonywać te same operacje arytmetyczne. Oczywiście istnieją pewne, opisane niżej w tym podrozdziale, ograniczenia w unifikacji wynikające z konkretnego przeznaczenia każdego typu programów cieniowania.

Potok renderingu wypełnia danymi cel renderingu (**RT** – ang. Render Target). Jest to bufor, do którego GPU zapisuje kolory i głębokości pikseli będące efektem działania potoku renderingu. Domyślnym celem renderingu jest tylny bufor, czyli część pamięci wideo zawierająca ramkę obrazu. W mechanizmie podwójnego buforowania po wypełnieniu tej ramki wywołuje się funkcję zamiany buforów tylnego i przedniego i wtedy zawartość tej ramki zostaje wyświetlona na ekranie. Nowoczesne GPU pozwalają zdefiniować RT w innym miejscu pamięci karty graficznej i podłączyć go do tekstury. Dzięki temu otwiera się możliwość renderingu do tekstur, które później używane są jako tekstury do renderingu na ekran. Typowym zastosowaniem jest tu rendering otoczenia do tekstury sześciennej w celu zamodelowania odbić środowiska (rozdział 6). Rozszerzeniem pojęcia RT jest **MRT** (ang. Multi Render Target), czyli mechanizm, który pozwala renderować więcej niż jeden kolor pikseli w pojedynczym przejściu potoku renderingu. Uzyskuje się to poprzez podłączenie kilku tekstur do RT jako wyjścia renderingu. W przetwarzaniu GPGPU można traktować RT jako interfejs pamięci tylko do zapisu.

Rozszerzony programowalny potok renderingu jest przedstawiony na rysunku 4.3. Składa się on z asemblera danych wejściowych (**IA** – ang. Input Assembler), programu cieniowania wierzchołków (**VS** – ang. Vertex Shader), programu cieniowania geometrii (**GS** – ang. Geometry Shader), wyjścia strumieniowego (**SO** – ang. Stream Output), etapu rasteryzacji (**RS** – ang. Rasterize Stage), programu cieniowania fragmentów (**PS** – ang. Pixel Shader) i etapu łączenia wyjścia (**OM** – ang. Output Merger).



Rysunek 4.3. Rozszerzony potok renderingu modelu cieniowania 4.0 (*źródło: na podstawie [6]*).

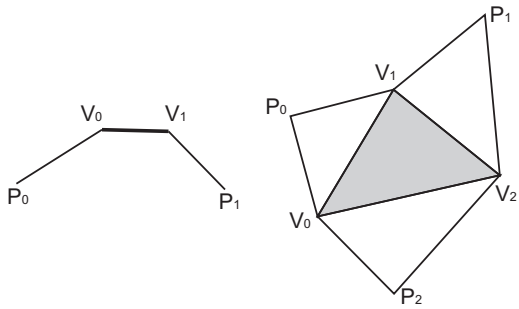
Asembler danych wejściowych (**IA**) odczytuje dane wierzchołków ze strumieni wejściowych podłączonych do buforów wierzchołków. Odczytane wartości są konwertowane na reprezentację zmiennoprzecinkową. W tej reprezentacji 32-bitowych liczb typu float przeprowadzane są wszystkie opisane niżej obliczenia i przekazywanie danych. Każda porcja danych związana z wierzchołkiem może zawierać do 16 wektorów 4-wymiarowych. Dane kolejnych wierzchołków odczytywane są kolejno z bufora wierzchołków chyba, że dany jest bufor indeksów. Wtedy dane wierzchołków odczytywane są zgodnie z kolejnością indeksów. Indeksowanie pozwala na optymalizację cieniowania wierzchołków. Wynik działania programu cieniowania wierzchołków zależy od indeksu. Wyniki można zapisywać do pamięci podręcznej, co umożliwia wykorzystanie zapamiętanego wyniku, gdy zachodzi potrzeba przetwarzania wierzchołka o indeksie, który już był przetwarzany.

Model cieniowania 4.0 wprowadza także mechanizm tworzenia instancji obiektów. Zbiór wierzchołków należących do powielanego obiektu jest „tagowany" identyfikatorami wierzchołka, prymitywu i instancji, które mogą być użyte przez dalsze etapy cieniowania. Dzięki temu instancje można później różnicować, poprzez nadanie im różnych transformacji i cech wizualnych. Instancje odciążają połączenie CPU-GPU, gdyż dzięki nim można przesyłać do karty graficznej dużo mniej informacji o obiektach (dużo krótsze bufory wierzchołków), jeśli obiekty się powtarzają.

Program cieniowania wierzchołków (**VS**) najczęściej używany jest do przekształceń, czyli transformowania wierzchołka z przestrzeni obiektu do jednorodnych współrzędnych obcinania. Program cieniowania wierzchołka otrzymuje dane jednego wierzchołka (do 16 wektorów z IA) i zwraca na wyjściu przetworzone dane tego wierzchołka. Nie ma możliwości odwołania się do danych innych wierzchołków, nawet sąsiednich. Programy cieniowania wierzchołków mają wspólny zestaw cech (unifikacja) z innymi programowalnymi etapami cieniowania, który obejmuje wspólną listę instrukcji, dostęp do maksymalnie 128 tekstur i do 16 buforów wartości stałych.

Program cieniowania geometrii (**GS**) otrzymuje dane wierzchołków pojedynczego prymitywu (punktu, odcinka, trójkąta) i generuje dane wierzchołków jednego lub większej liczby prymitywów. Może również nie wygenerować prymitywu (usunąć wierzchołki z dalszego przetwarzania). Rodzaj prymitywów wyjściowych nie musi być taki sam, jak wejściowych, ale dla danego programu cieniowania geometrii musi być stały. Ponieważ program cieniowania geometrii może drastycznie zwiększyć liczbę wierzchołków w potoku, to została ona ograniczona maksymalną liczbą 1024 32-bitowych wartości dla wszystkich danych wszystkich stworzonych przez jedno wywołanie GS wierzchołków. Oprócz usuwania i dodawania wierzchołków program cieniowania geometrii może zmieniać lub dodawać atrybuty przetwarzanym wierzchołkom. Atrybuty te mogą mieć związek z całymi prymitywami, gdyż program cieniowania geometrii ma dostęp do danych wszystkich wierzchołków prymitywu. Dzięki temu można na przykład obliczyć normalną przetwarzanego trójkąta, co niemożliwe jest w shaderze wierzchołków.

Ponadto program cieniowania geometrii odcinka i trójkąta ma dostęp do wierzchołków prymitywów połączonych z aktualnie przetwarzanym. Można więc mieć dostęp do 3 wierzchołków trójkąta i 3 wierzchołków sąsiadujących bądź 2 wierzchołków odcinka i 2 wierzchołków sąsiadujących (rys. 4.4).



Rysunek 4.4. Wierzchołki przetwarzanego w GS prymitywu (V) i wierzchołki prymitywów przylegających (P) (*źródło: na podstawie [6]*).

Programy cieniowania geometrii mogą również preparować strumienie danych o pożądanych cechach (na przykład przez konwersję lub pakowanie) dla następnego etapu potoku (SO), jeśli jest on włączony.

Wyjście strumieniowe (**SO**) kopiuje podzbiór danych wierzchołków wychodzących kolejno z programu cieniowania geometrii (GS) do bufora wyjściowego. Mogą być maksymalnie 4 bufory wyjściowe. Wyjście strumieniowe może wygenerować 1 strumień wyjściowy 16-elementowy lub 4 strumienie wyjściowe jednoelementowe. Element jest zawsze pojedynczą liczbą 32-bitową typu float – nie ma możliwości konwersji. Jednakże konwersja danych i pakowanie mogą być zaimplementowane w programie cieniowania geometrii, więc nie jest to wada.

Rasteryzacja (**RS**) jest nieprogramowalnym etapem, którego zadaniem jest:

— obcinanie,
— culling[4],
— dzielenie perspektywiczne,
— transformacja widoku,
— składanie prymitywów,
— test nożyc,
— generowanie wartości głębokości dla fragmentu,
— Z-culling[5],
— generowanie fragmentów.

Wejściem RS są wierzchołki i atrybuty pojedynczego prymitywu, a wyjściem strumień fragmentów kierowanych do programu cieniowania fragmentów. Program cieniowania fragmentów wybiera sposób w jaki atrybuty wierzchołków są interpolowane, aby uzyskać atrybuty fragmentu:

— brak interpolacji,

[4]Culling – mechanizm usuwania z potoku renderingu trójkątów odwróconych do punktu widzenia niewidoczną stroną.

[5]Z-Culling – mechanizm usuwania z potoku renderingu fragmentów które nie przechodzą testu głębokości. Od klasycznego testowania głębokości różni się tym, że usunięcie odbywa się przed a nie po uruchomieniu programu cieniowania dla danego fragmentu.

— interpolacja z korekcją perspektywiczną,
— interpolacja bez korekcji perspektywicznej.

Program cieniowania fragmentów (**PS**) otrzymuje atrybuty pojedynczego piksela i zwraca pojedynczy fragment charakteryzujący się kolorem i opcjonalnie wartością głębi. Zależnie od liczby celów renderingu (RT) może być do ośmiu wyjściowych kolorów na jeden fragment. Program cieniowania fragmentów może też usunąć fragment z dalszego przetwarzania. Może również zmienić przekazywaną z etapu rasteryzacji (RS) wartość głębokości, która przekazywana jest do następnego etapu przetwarzania.

Etap łączenia wyjścia (**OM**, dawniej **ROP** – ang. Raster Operations) przeprowadza na otrzymanym fragmencie test szablonu (ang. stencil test) i test głębokości (ang. depth test) oraz przeprowadza mieszanie (ang. blending) koloru fragmentu z kolorem piksela zapisanym w celu renderingu (RT). Używa pojedynczego bufora głębokości/szablonu i do ośmiu celów renderingu. Funkcja mieszania jest wspólna dla wszystkich celów, ale samo mieszanie można dla każdego celu niezależnie włączyć lub wyłączyć.

Jak widać, w modelu cieniowania 4.0, wszystkie etapy standardowego potoku renderingu, które można było wyrazić jako ciąg instrukcji programu cieniowania, zostały udostępnione jako programowalne: transformacje i zmiana liczby wierzchołków, oświetlenie, teksturowanie i mgła. Pozostałe bloki – IA, RS i OM – zostały uelastycznione dla lepszej interakcji z blokami programowalnymi. Dalsze próby przenoszenia części przetwarzania do bloków programowalnych mogłyby poskutkować dużymi kosztami zmniejszenia efektywności i szybkości obliczeń. Przykładowo OM jest jedynym etapem, w którym używane są operacje read-modify-write w pamięci. Jest to potrzebne dla końcowego etapu mieszania koloru fragmentu i koloru piksela w RT. Przeniesienie funkcjonalności testowania głębokości, szablonu i mieszania do programu cieniowania fragmentów spowodowałoby problemy z zarządzaniem potokiem renderingu i z maksymalnie efektywnym wykorzystaniem pamięci [6].

Model cieniowania 4.0 wprowadza unifikację programów cieniowania. Wszystkie programy cieniowania wykonywane są na tym samym rdzeniu, który definiuje rejestry jako 4-wymiarowe wektory liczb 32-bitowych oraz zapewnia arytmetyczne operacje na nich tak, jak w starszych modelach cieniowania.

Ponadto każdy programowalny etap przetwarzania modelu 4.0:

— wykonuje arytmetyczne operacje na skalarach, wektorach 2-, 3- oraz 4-wymiarowych oraz macierzach $(3 \times 3, 4 \times 4)$, operacje iloczynów: wektorowego i skalarnego, mnożenie i transponowanie macierzy, funkcje trygonometryczne, wartość bezwzględną, zaokrąglanie, obcinanie, konwersje miar kątów, obliczanie wyznacznika macierzy, funkcje eksponencjalne, dzielenie z resztą, interpolację liniową, funkcję obliczającą oświetlenie, logarytmy, minimum i maksimum, generator szumu, potęgowanie, pierwiastkowanie, signum i inne,
— oblicza funkcje geometryczne: odległość między punktami, długość wektora, normalizację, wektor odbicia i załamania,
— wykonuje arytmetyczne i logiczne operacje na liczbach całkowitych oraz konwersje,
— ma dostęp do zbioru (4096 wektorów 4×32-bitowych) rejestrów tymczasowych ogólnego przeznaczenia (odczyt i zapis), które mogą być indeksowane,
— obsługuje oddzielne instrukcje do filtrowanego i niefiltrowanego odczytu tekstur,
— ma do dyspozycji osobne punkty podłączania tekstur (128) oraz ich samplerów (16),
— może odczytywać wartości stałe z 16 banków po 4096 wektorów 4×32-bitowych.

Wszystkie funkcje matematyczne działające na skalarach (jak wartość bezwzględna) działają również na wektorach. Wtedy każda składowa wektora

jest poddawana działaniu funkcji, lecz dzieje się to jednocześnie dla wszystkich składowych. Właściwość tę należy wykorzystywać do wektoryzowania tworzonych programów w celu przyspieszenia ich działania.

Podzielenie pamięci stałych na 16 banków umożliwia rozdzielenie operacji ich zmiany przez CPU i operacji podłączania ich do potoku renderingu. Dzięki temu można podzielić stałe na grupy o różnej częstotliwości aktualizacji (raz na ramkę, raz na obiekt, raz na instancję obiektu) i zmniejszyć dzięki temu wymagania komunikacyjne na linii CPU-GPU. Tak zunifikowany model jest bardzo bliski temu, co możliwe jest do osiągnięcia na CPU i powoduje duży skok możliwości obliczeniowych w porównaniu z poprzednimi modelami cieniowania.

Mimo unifikacji poszczególne programy cieniowania nie mają identycznych możliwości. Specjalizacja polega na różnej konfiguracji rejestrów wejściowych i wyjściowych oraz wprowadzeniu dodatkowych instrukcji. Program cieniowania wierzchołków z wejściem i wyjściem skonfigurowanym jako 16 wektorów 4D float32 definiuje wspólną funkcjonalność.

Program cieniowania geometrii (GS) może przyjąć 6 razy tyle danych co VS, gdyż w najbardziej rozbudowanej wersji wywołania przyjmuje następujące dane: 3 wierzchołki trójkąta wraz z 3 wierzchołkami sąsiednimi. Ponieważ GS może zwrócić 0, 1 lub więcej wierzchołków lub prymitywów, więc nie może używać modelu rejestrów wyjściowych, jak pozostałe typy programów cieniowania. Zamiast tego dostępna jest dodatkowa instrukcja `emit`, która sygnalizuje wyjście danych kolejnego wierzchołka do następnego etapu przetwarzania. Dostępna jest również instrukcja `cut`, która sygnalizuje koniec strumienia wierzchołków należących do przetwarzanego właśnie prymitywu. Każdy wierzchołek wychodzący z programu cieniowania geometrii (GS) może mieć do 32 wektorów 4D float32. Jest to dwa razy więcej niż zwraca program cieniowania wierzchołków. Dodatkowe dane są używane do przekazywania danych o obcinaniu i cullingu do etapu rasteryzacji (RS) oraz do przekazywania informacji o prymitywie do programu cieniowania fragmentów (PS).

Program cieniowania fragmentów obsługuje maksimum 32 wektory 4D float32 na wejściu wtedy, gdy włączone jest przetwarzanie geometrii w GS. W przeciwnym przypadku dostępne jest 16 wektorów. Z każdym wektorem związana jest w czasie kompilacji informacja o metodzie interpolacji jego składowych. Program cieniowania fragmentów (PS) zwraca do 8 (zależnie od liczby celów renderingu) wektorów 4D zawierających informację o kolorze oraz informację o głębokości. Oprócz tego pozwala usunąć fragment z dalszego przetwarzania. Nie zostanie on wtedy zapisany do celu renderingu (RT). Mimo, iż dzięki unifikacji wszystkie typy programów cieniowania mogą odczytywać tekstury, to tylko program cieniowania fragmentów (PS) ma dostęp do mipmap[6].

4.2.2 Tekstury w przetwarzaniu danych na GPU

Tekstury w grafice komputerowej są tablicami przechowującymi dane kolejnych tekseli, czyli ich kolory. Wyróżnia się tekstury:

— jednowymiarowe (1D),
— dwuwymiarowe (2D),
— trójwymiarowe (3D),
— sześcienne (CUBE) – każda taka tekstura składa się z sześciu tekstur dwuwymiarowych,
— prostokątne (RECT) – tekstury dwuwymiarowe o dowolnych rozmiarach, jednak bez możliwości próbkowania ich.

Rozmiary tekstur standardowych są potęgami liczby 2, dopuszczalne rozmiary tekstur dwuwymiarowych to $2^m \times 2^n$ dla naturalnych m, n. Maksymalny rozmiar tekstury 2D w każdym z wymiarów wynosi 16384 teksele.

Standardy grafiki komputerowej przewidują różne formaty liczbowego zapisu koloru tekseli. Kolor może być zapisany w teksturze za pomocą 1, 2,

[6]Mipmapa – w filtrowaniu tekstur na użytek grafiki 3D, to zbiór przygotowanych przed właściwym renderingiem tekstur, będących wynikiem skalowania tekstury wejściowej. Każda następna tekstura w mipmapie ma szerokość i wysokość dwa razy mniejszą od poprzedniej. Celem jest przyspieszenie renderingu i poprawa jego jakości.

3 lub 4 składowych koloru. O teksturze mówi się wtedy, że jest 1-, 2-, 3- lub 4-kanałowa. Współczesny sprzęt wspiera model koloru RGBA. Przewiduje on przechowywanie koloru teksela za pomocą czterech kanałów wyznaczających kolejno wartości czerwonej, zielonej, niebieskiej składowej koloru oraz przezroczystości. Ich reprezentacja jest standardowo 8-bitową liczbą całkowitą bez znaku (`OpenGL: GL_RGBA, GL_UNSIGNED_BYTE`). W związku z tym dopuszczalne wartości mieszczą się w zbiorze $\{0, 1, ...255\}$. Taka reprezentacja wystarcza do większości zastosowań grafiki 3D, ale może być niewystarczająca, gdy trzeba przetwarzać dane o ogólniejszym charakterze. Dlatego też rozszerzenia standardu OpenGL przewidują możliwość przechowywania wartości składowych koloru przy pomocy innych reprezentacji, na przykład zmiennoprzecinkowych, gdyż oferują one większy zakres dla przechowywanych wartości. Wartości zmiennoprzecinkowe mogą być 16- lub 32-bitowe (`OpenGL: GL_RGBA16F_ARB lub GL_RGBA32F_ARB, GL_FLOAT`).

Tak zdefiniowane **tekstury mogą służyć jako struktury przechowujące dane** do obliczeń ogólnego przeznaczenia:

— są tablicami określonego wymiaru i rozmiaru,

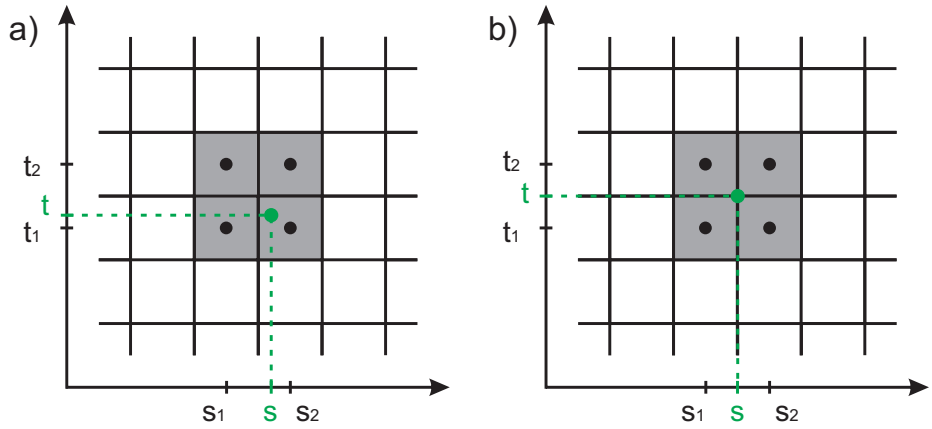
— ich elementy (teksele) mają wewnętrzną strukturę – są skalarami lub wektorami 2-, 3- lub 4-wymiarowymi liczb całkowitych lub zmiennoprzecinkowych, które niekoniecznie muszą być składowymi koloru, lecz mogą zawierać dowolne dane.

Tekstury dla programów cieniowania są danymi tylko do odczytu. Odczytywane są dzięki funkcjom próbkowania tekstury z zadanych miejsc będących współrzędnymi tekseli w teksturze. Próbkowanie odpowiada tutaj odczytowi wartości z tablicy spod zadanego współrzędnymi miejsca. Funkcje próbkowania tekstur dwuwymiarowych wymagają dwuwymiarowych współrzędnych, aby odczytać spod nich wartość koloru teksela. W przypadku standardowych tekstur współrzędne odczytu teksela są zawsze z zakresu $< 0, 1 >$ niezależnie od rozmiarów tekstury w pikselach. Czyli współrzędne $(0.5, 0.5)$ zawsze oznaczają środkowy piksel tekstury dwuwymiarowej.

Planując strukturę tekstury wejściowej dla przetwarzania GPGPU należy wziąć pod uwagę wektoryzowanie obliczeń. Jeśli w rozwiązaniu problemu na CPU bierze udział tablica skalarów, to z powodu oszczędności pamięci należałoby dobrać do jej przechowywania format tekstury zapewniający przechowywanie jednego kanału koloru (np. GL_RED). Jednakże wynik próbkowania takiej tekstury w programie cieniowania będzie skalarem, mniej efektywnie przetwarzanym niż wektor. Dlatego, jeśli rozwiązywany problem na to pozwala należy użyć tekstury czterokanałowej (GL_RGBA) która w kolejnych składowych koloru będzie przechowywała kolejne wartości oryginalnej tablicy. Dzięki temu tekstura będzie miała cztery razy mniej elementów od oryginalnej tablicy. Natomiast pojedyncze próbkowanie tekstury spowoduje odczyt czterech wartości tablicy na raz, które mogą być przetwarzane równolegle w programie cieniowania jako składowe wektora. Pomiary dokonane przez autora wykazały, że takie użycie tekstury daje spodziewane prawie czterokrotne przyspieszenie działania programu w stosunku do wersji z teksturą jednokanałową.

Zaletą próbkowania tekstur przez GPU, w porównaniu z odczytem zawartości tablic przez CPU, jest możliwość odczytu wartości z tekstury w jednym z dwóch trybów próbkowania. Może to być próbkowanie najbliższego sąsiada lub interpolacja liniowa sąsiadów. W przypadku tekstur jednowymiarowych interpolowana jest wartość dwóch sąsiadów, dla tekstur dwuwymiarowych wykonywana jest interpolacja dwuliniowa czterech sąsiadów, a dla tekstur trójwymiarowych interpolacja trójliniowa ośmiu sąsiadów. Różnice między trybami próbkowania widoczne są w przypadku, gdy używane są współrzędne (s, t) odczytu tekstury, które nie adresują środka teksela (rys. 4.5a). Jest to możliwe ponieważ współrzędne odczytu tekstury są reprezentowane przez liczby zmiennoprzecinkowe. Dzięki interpolacji liniowej możliwe są dodatkowe optymalizacje kodu kernela. Jeśli zadanie wiąże się z obliczeniem średniej wartości kolorów sąsiednich tekseli, to próbkowanie tekstury „między" tymi tekselami powoduje uzyskanie wartości tej średniej w jednym odczycie. Jeśli do obliczenia jest średnia wartość kolorów czterech

tekseli ułożonych w teksturze w bloku 2×2 (rys. 4.5b) to wystarczy jedna operacja próbkowania z interpolacją dwuliniową, zamiast czterech. Dodatkowym argumentem za używaniem tego mechanizmu jest fakt, że pomiary przeprowadzone przez autora wykazały, że zmiana trybu odczytu tekstury z najbliższego sąsiada na interpolację nie wydłuża czasu wykonania.



Rysunek 4.5. Próbkowanie tekstury w punkcie (s, t): a) w przypadku próbkowania najbliższego sąsiada zostanie zwrócona wartość teksela o współrzędnych (s_2, t_1) w mapie tekstury, w przypadku próbkowania z interpolacją zwrócona zostanie zinterpolowana dwuliniowo wartość czterech sąsiednich (oznaczonych szarym kolorem) tekseli; b) jedna operacja próbkowania dokładnie w środku między czterema tekselami odczytuje średnią wartość ich kolorów.

Rejestry GPU, podobnie jak kolory tekseli w teksturze czterokanałowej, są 4-wymiarowymi wektorami. Używane liczby są z reguły zmiennoprzecinkowe niezależnie od formatu stosowanych tekstur wejściowych. W przypadku, gdy tekstura przechowuje liczby całkowite, na wejściu zostają one przeskalowane na liczby zmiennoprzecinkowe poprzez dzielenie przez wartość maksymalną. Czyli zakres liczb całkowitych $< 0, 255 >$ zostaje przekonwertowany na zmiennoprzecinkowy zakres $< 0, 1 >$. Konwersja taka jest

automatyczna podczas odczytu tekstury. W przypadku tekstur z elementami zmiennoprzecinkowymi, na wejściu pobierana jest bezpośrednia zmiennoprzecinkowa wartość.

Wyjściowy kolor obliczony przez program cieniowania fragmentów jest traktowany analogicznie. Jeśli cel renderingu (RT) przechowuje dane o kolorach jako wartości zmiennoprzecinkowe, to bezpośrednio zapisywana jest wartość zmiennoprzecinkowa. Jeśli wyjściowe wartości są całkowite, to wartości składowych kolorów, które nie mieszczą się przed konwersją w zakresie $< 0, 1 >$ zostają obcięte, tzn. wartości mniejsze od 0 stają się zerem, większe od 1 stają się jedynką. Następnie zakres $< 0, 1 >$ zostaje przekonwertowany na zakres liczb całkowitych $< 0, 255 >$.

4.2.3 Mapowanie problemu przetwarzania na GPU

Omówione wcześniej w podrozdziale 4.2.1 cechy modelu cieniowania 4.0, jak:

— unifikacja programów cieniowania,
— możliwość obsługi dużej ilości danych w strukturach tablicowych, czyli teksturach o różnych wymiarach i rozmiarach oraz bankach parametrów,
— możliwość użycia wielokrotnego celu renderingu (MRT) jako wielokrotnego wyjścia dla programu cieniowania fragmentów (PS) lub użycia wyjściowego strumienia wierzchołków jako wyjścia strumieniowego (SO),
— spora wielkość elementów strumieni (do 32 wektorów czterowymiarowych typu `float`),
— możliwość zunifikowanego użycia wyjściowych tekstur jednego etapu przetwarzania, jako wejściowych tekstur następnego etapu przetwarzania,

powodują, że model ten bardzo dobrze nadaje się do celów przetwarzania ogólnego przeznaczenia – GPGPU.

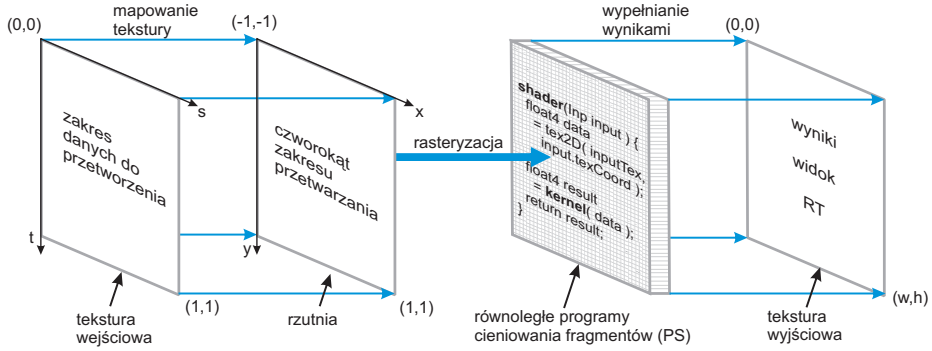
Motywacją badań nad GPGPU była możliwość znacznego przyspieszenia działania programów dzięki przetwarzaniu równoległemu (strumieniowemu). Zadania, które tego wymagają są wykonywane wolno przez CPU dlatego, że CPU oferują sekwencyjne przetwarzanie danych, a jest ich często znaczna ilość. Dane te są często tablicami liczb lub struktur liczb zapisanych w standardowych reprezentacjach zmiennoprzecinkowych. Pozwala to zaadaptować programy cieniowania na GPU do obliczeń ogólnego przeznaczenia, gdyż GPU obsługujące model cieniowania 4.0 posługują się standardową (IEEE-754) reprezentacją liczb zmiennoprzecinkowych pojedynczej precyzji, co pozwala na taką samą ich interpretację po stronie CPU (który przygotowuje dane oraz odbiera wyniki), jak i GPU (który wykonuje właściwe obliczenia).

Podstawowy, jednoetapowy schemat przetwarzania GPGPU składa się z następujących czynności:

Jednoetapowe przetwarzanie GPGPU (rys. 4.6)

1. przygotowanie i załadowanie do pamięci karty graficznej danych zapisanych w teksturze (teksturach),
2. kompilacja programu cieniowania fragmentów (PS) – kernela operacji,
3. ustawienie widoku (viewport) na rozmiar tekstury wyjściowej wyrażony w pikselach (w, h),
4. ustawienie rysowania w jednorodnej przestrzeni obcinania,
5. podłączenie do celu renderingu (RT) tekstury wyjściowej o odpowiednim wymiarze i rozmiarze,
6. narysowanie **czworokąta** o wielkości widoku spowoduje uruchomienie programów cieniowania dla wszystkich rysowanych pikseli (rys. 4.6),
7. odczytanie rezultatu z tekstury w celu renderingu (RT).

Mapowanie problemu przetwarzania na GPU można opisać odnosząc się do wyżej opisanych czynności, jako odpowiedników czynności wykonywanych na CPU w celu rozwiązania tego problemu:



Rysunek 4.6. Jednoetapowe przetwarzanie GPGPU.

Ad 1. Czynność ta odpowiada przygotowaniu danych w tablicach dla CPU.

Ad 2. Przetwarzanie tablicy na CPU wiąże się zazwyczaj z organizacją pętli, która iteruje po elementach tablicy. W przypadku programów cieniowania, czynności które wykonuje kernel są zmapowanymi na instrukcje GPU czynnościami, które na CPU są czynnościami wewnątrz pętli. Należy zwrócić uwagę, że dane wywołanie kernela może wykonać dowolną liczbę odczytów tekstury wejściowej. Ponadto tekstur wejściowych może być więcej niż jedna. Natomiast jedno wywołanie kernela wykonuje tylko jeden zapis do tekstury wyjściowej.

Ad 3., **Ad 4.**, **Ad 5.** Czynności te pozwalają na właściwe adresowanie tekseli tekstury wyjściowej tak, aby każde wywołanie kernela PS obliczało pożądaną wartość i zapisywało ją w odpowiednie miejsce tej tekstury.

Ad 6. Narysowanie czworokąta wielkości widoku powoduje jego rasteryzację, czyli zamianę na strumień fragmentów i uruchomienie dla każdego z nich programu cieniowania fragmentów (PS) zawierającego kernel operacji. W ten sposób odbywają się obliczenia GPGPU.

Ad 7. Ponieważ celem GPGPU jest wykonanie obliczeń a nie rendering, to zapisane w RT dane nie są wyświetlane, tylko wypełniają teksturę wyjściową. Można jej użyć jako wejściowej w następnych etapach przetwarzania na GPU lub ściągnąć do pamięci głównej, dostępnej dla CPU.

W celu rozwiązania postawionego problemu na GPU można połączyć dowolną liczbę przetwarzań zaprojektowanych według powyższego schematu w wieloetapowe przetwarzanie, gdzie tekstura wyjściowa poprzedzającego etapu jest wejściową następnego.

Jednym z najważniejszych ograniczeń, które towarzyszy programom cieniowania od samego ich powstania, jest to, że dany zasób nie może być jednocześnie wejściem i wyjściem danego etapu przetwarzania. Dzieje się tak dlatego, że operacje read-modify-write są operacjami, które dużo trudniej synchronizować niż sam odczyt, albo sam zapis. Wprowadzenie takich możliwości mogłoby spowodować bardzo duży spadek efektywności wykonywania programów na GPU. Dlatego możliwość jakiegokolwiek łączenia wyników tego samego etapu przetwarzania powstałych w dwóch różnych przebiegach PS jest dostępna tylko w module łączenia wyjścia (OM). Tylko ten ostatni etap potoku przetwarzania podczas zapisu wartości fragmentu w RT może zmieszać (blend) wartość fragmentu z wartością zastaną w RT. Praktyczne użycie tej cechy w GPGPU pokazuje rozdział 5.

Jeżeli etapów przetwarzania jest więcej, a tekstury będące wynikami pośrednimi kolejnych etapów przetwarzania są takie same (rozmiar, format), to w celu oszczędności pamięci tekstur można zastosować technikę **ping-pong**. Polega ona na naprzemiennym traktowaniu dwóch tekstur A i B jako wejściowych lub wyjściowych w kolejnych etapach przetwarzania. To znaczy, że jeśli w i-tym kroku przetwarzania próbkowana jest tekstura A, a do RT jest podłączona tekstura B, to w $(i+1)$-szym kroku tekstura B jest próbkowana a A jest wyjściem, z kolei w $(i+2)$-im A jest znowu wejściem a B wyjściem, i tak dalej. Takie podejście jest często spotykane w rozwiązaniach GPGPU, ponieważ ułatwia zarządzanie teksturami i oszczędza pamięć.

W przypadku, gdy jedna, czterokanałowa tekstura wyjściowa nie jest w stanie pomieścić wszystkich wyników przetwarzania, można zastosować mechanizm wielokrotnego celu renderingu (MRT). Wtedy program cieniowania fragmentów może wygenerować więcej niż jeden wektor 4D zawierający wyniki obliczeń. Tekstury podłączone do MRT mogą być później

próbkowane przez następne etapy przetwarzania z gwarancją, że wartości odczytane w tych samych współrzędnych powstały w tym samym wywołaniu kernela.

4.2.4 Optymalizacja kodu programów cieniowania

Aby stworzyć efektywny, szybki program przetwarzania danych na GPU, należy stosować poniższe zasady [105]:

1. Wektoryzować przetwarzanie: należy korzystać z równoległości przetwarzania na poziomie danych. Dodawanie dwóch wektorów 4D jest szybsze niż czterech par skalarów. Przydatne w wektoryzowaniu jest mieszanie składowych wektorów. Polega ono na tymczasowej zmianie kolejności składowych wektora użytego w wyrażeniu. Przykładowo obliczenie iloczynu wektorowego (r) dwóch wektorów 3D, a i b, można dzięki mieszaniu zapisać:
   $r.xyz = a.yzx * b.zxy - a.zxy * b.yzx$
   spowoduje to jednoczesne (wektorowe) obliczenie wartości trzech wyrażeń:
   $r.x = a.y * b.z - a.z * b.y$
   $r.y = a.z * b.x - a.x * b.z$
   $r.z = a.x * b.y - a.y * b.x$
   Składowe wektorów (x, y, z, w) zapisuje się po kropce. Dla każdego wektora czterowymiarowego $c = c.xyzw$. Składowe podczas mieszania można powtarzać, na przykład $a = b.zyyz$.
2. Używać funkcji wbudowanych, które są zawsze szybsze niż samodzielnie zbudowany fragment kodu wykonujący tę samą operację.
3. Wiedzieć, jak szybkie są poszczególne funkcje. Na przykład użycie funkcji `abs()` (wartość bezwzględna) i `saturate` (obcinanie do zakresu $< 0, 1 >$) nie generuje dodatkowego kodu i czasu wykonania. Kod:
   $a = \text{abs}(b) + \text{saturate}(c)$
   wykona się tak samo szybko, jak:
   $a = b + c$

4. Przygotować skomplikowane obliczeniowo funkcje jako tekstury: odczyt z tekstury gotowej, wstępnie obliczonej wartości jest zazwyczaj szybszy niż ciąg obliczeń w programie cieniowania.
5. Używać typu danych o minimalnej potrzebnej precyzji oraz tekstur o minimalnej liczbie kanałów.
6. Unikać w miarę możliwości kodu wykonywanego warunkowo. Co prawda model cieniowania 4.0 zapewnia dynamiczną kontrolę przepływu programu, lecz nie należy nadużywać tych możliwości. Szczególnie duże straty szybkości mają miejsce, jeśli wykonanie bloku kodu jest uwarunkowane wartością zmiennej obliczoną w bieżącym programie cieniowania.

Ostatni punkt dotyczy optymalizacji kontroli przepływu programu. Nie jest to aspekt programowania GPU, który można łatwo przenosić z implementacji CPU mimo, iż na liście instrukcji GPU widnieją instrukcje warunkowe i pętle. Model cieniowania 4.0 obsługuje rozgałęzianie przepływu programów cieniowania, ale korzystanie z tego wymaga, aby cały czas mieć na uwadze strumieniowy charakter przetwarzania. Istnieje kilka technik wprowadzania instrukcji warunkowych do programów dla GPU. Wśród nich mechanizmy predykacji i rozgałęziania dynamicznego są technikami, które zapewnia sprzęt [65].

Predykacja na GPU odbywa się przez ewaluację obu ścieżek rozgałęzienia, a następnie odrzucenie wyniku tej, dla której warunek rozgałęzienia nie jest prawdziwy. Wadą tego rozwiązania jest bardzo prawdopodobne wydłużenie czasu działania programu poprzez konieczność wykonywania niepotrzebnych instrukcji. Jeżeli warunkowe jest wykonanie jednej instrukcji należy rozważyć użycie instrukcji, których wynik działania jest obliczany warunkowo, na przykład `step(), faceforward()`.

W przypadku napotkania konstrukcji `if-then-else` w kodzie programu cieniowania przetwarzany jest pewien zbiór elementów. Jeżeli warunek dla wszystkich elementów jest ewaluowany do tej samej wartości, to zbiór

przetwarzany jest wskazaną przez nią ścieżką. Jeżeli jednak część elementów wymaga przetwarzania przy pomocy jednego odgałęzienia, a część przy pomocy drugiego, to program wykonywany jest dwa razy. Raz według jednego, a drugi raz według drugiego wariantu przebiegu. Efekt przetwarzania elementów, które w danym przebiegu nie spełniają warunku odgałęzienia jest tracony. Może to powodować duże straty szybkości przetwarzania. Jednak w rozgałęzieniach dynamicznych oferowanych przez model cieniowania 4.0, GPU może podzielić jednostki przetwarzające na podzbiory (grupa SIMD). Dzięki temu wzrasta prawdopodobieństwo, że w danym podzbiorze, mniejszym niż pierwotnie, parametry wszystkich elementów będą powodowały ewaluację warunku do tej samej wartości, a nawet jeśli się tak nie zdarzy, to nadmiarowe przetwarzanie dotyczy stosunkowo niewielkiej liczby elementów. Karty graficzne nVidia GeForce serii 8 oferują podział na 16-elementowe podzbiory.

Podczas projektowania aplikacji GPGPU nie można polegać tylko na mechanizmach sprzętowych, gdyż przy niesprzyjających danych wejściowych szybkość przetwarzania może spaść poniżej założonej granicy. Aby nie dopuścić do tego można zastosować wybrane techniki optymalizacji rozgałęziania na etapie projektowania. Zazwyczaj polega to na przeniesieniu warunku do któregoś poprzedniego etapu przetwarzania, gdzie rozgałęzienie może być efektywniej zrealizowane.

Przykładem statycznego rozstrzygania rozgałęziania (ang. Static Branch Resolution) jest eliminacja rozgałęzień wewnątrz pętli. Jeżeli realizowany jest algorytm działający na dwuwymiarowej siatce, której elementy brzegowe są przetwarzane inaczej, niż elementy wewnętrzne posiadające pełne sąsiedztwo, należy zastosować dwa kernele obsługujące te dwa przypadki. Wtedy osobnym przetwarzaniem objęte są dwie klasy elementów, a przetwarzające je pętle nie mają w środku rozgałęzienia różnicującego przetwarzanie elementu brzegowego i wewnętrznego. Technika ta nazywana jest również dzieleniem strumienia na podstrumienie.

Drugim przykładem jest zapamiętywanie wyników obliczeń. Jeżeli jakieś rozgałęzienie daje ten sam wynik przez kilka iteracji, to należy zapamiętać jego wynik i ewaluować tylko odgałęzienia, których wyniki obliczeń się zmieniają.

Model cieniowania 4.0 zapewnia mechanizm Z-culling, który zapobiega cieniowaniu pikseli, które nie będą widoczne. Polega on na porównywaniu głębokości wychodzącego z etapu rasteryzacji (RS) fragmentu z wartością zapisaną w buforze głębokości. Fragmenty, które nie przeszły tego testu są odrzucone bez uruchamiania programu ich cieniowania, co powoduje oszczędność czasu w porównaniu z sytuacją, kiedy byłyby odrzucone dopiero na etapie łączenia wyjścia (OM) przez test głębokości.

4.2.5 Operacje strumieniowe

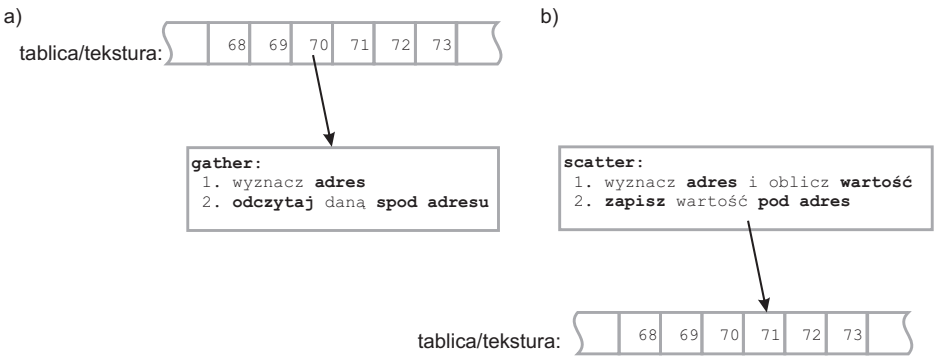
Model programowania strumieniowego jest użyteczną abstrakcją dla programowania GPU. Istnieje kilka fundamentalnych operacji na strumieniach, które są implementowane przez aplikacje GPGPU jako etapy przetwarzania: mapowanie, redukcja, gather i scatter, skanowanie, sortowanie, wyszukiwanie oraz filtrowanie strumienia.

Mapowanie to najprostsza operacja opisana w podrozdziale 4.2.3. Polega na obliczeniu wartości zadanej funkcji na każdym elemencie strumienia wejściowego i zapisanie jej w strumieniu wyjściowym. Prostym przykładem jest tu dodanie stałej wartości do kanałów koloru w buforze koloru, aby rozjaśnić obraz.

Redukcja zachodzi, gdy z większego strumienia wejściowego obliczany jest mniejszy strumień (nawet jednoelementowy). Przykładowo redukcja może być użyta do zsumowania elementów strumienia, albo znalezienia w nim największej wartości. W każdym etapie przetwarzania rozmiar wyjścia jest zredukowany o połowę względem rozmiaru wejścia. Wykonanie redukcji strumienia n-elementowego w $O(\frac{n}{p} \log n)$ krokach używając GPU

(dla p elementów przetwarzanych w jednym kroku) jest szybsze od sekwencyjnego wykonania tej redukcji na CPU w $O(n)$ krokach. Każdy kernel odczytuje ze strumienia wejściowego dwie wartości, łączy je operatorem redukcji (np. dodawanie lub maksimum) i zapisuje jeden element strumienia wyjściowego. Proces jest powtarzany, dopóki strumień wyjściowy kolejnego etapu przetwarzania nie będzie miał długości jeden. W tym przypadku zawiera on obliczoną wartość redukcji. Redukcja ma swój wariant dwuwymiarowy, wtedy zmniejszanie rozmiaru strumienia zachodzi w obu wymiarach, a kernel wykonuje obliczenia na czterech elementach wejściowych.

Gather i scatter są dwoma podstawowymi operacjami na pamięci. Jeśli operacje odczytu i zapisu odwołują się do pamięci pośrednio, to są nazywane odpowiednio gather i scatter (rys. 4.7). Dla GPU operacja gather jest zasadniczo operacją próbkowania tekstury w wybranym miejscu. Jest to dostęp pośredni do pamięci poprzez odwołanie się do współrzędnych teksturowania w odczytywanej teksturze.



Rysunek 4.7. Operacje na pamięci: a) gather – pośredni odczyt, b) scatter – pośredni zapis.

Operacja scatter, czyli zapisu pod wybrany adres jest trudniejszą operacją na GPU. Fragmenty mają przypisany adres docelowy w celu renderingu (RT) i programy cieniowania fragmentów nie mogą go zmienić. Natomiast mogą to zrobić programy cieniowania wierzchołków lub geometrii. Programy

te od modelu cieniowania 4.0 mogą próbkować tekstury. Mogą również obliczać wyjściowe położenia rysowanych wierzchołków, co przekłada się na położenia rysowanych pikseli w RT. Znając na etapie przetwarzania wierzchołków wielkość tekstury wyjściowej i widoku można rysować wierzchołki jako punkty (`GL_POINTS`) w odpowiednich miejscach, wypełniając wybrane piksele tekstury wyjściowej. Jest to analogiczne do zapisu pod wybrane indeksy tablicy wyjściowej.

Skanowanie (ang. scan, prefix sum, prefix reduction, partial sum – suma częściowa) jest dla danego elementu obliczaniem sumy wszystkich elementów poprzedzających go. Dla GPU opracowane zostały algorytmy [77] działające w czasie $O(n)$, a więc takim samym, jak optymalne rozwiązanie sekwencyjne.

Sortowanie ma opracowanych kilka efektywnych algorytmów realizowanych na CPU. Przebieg wielu z nich jest zależny od dostarczonych danych i wymaga operacji scatter. Powoduje to, że realizacja zadania sortowania jest trudna do zaprogramowania na GPU. Z pomocą mogą przyjść tu sieci sortujące. Ich główną ideą jest podzielenie przetwarzania na kroki, których liczba zależy od wielkości strumienia wejściowego, ale w trakcie działania algorytmu jest stała. Dodatkowo węzły sieci mają stałą sieć połączeń, dzięki czemu operacje scatter mogą zostać zamienione na gather. Pozwala to na sortowanie w czasie $O(n \log^2 n)$.

Search pozwala na wyszukanie konkretnego elementu w strumieniu. Istniejące rozwiązania przeszukiwania binarnego pozwalają na równoległość wielu poszukiwań, każde z nich wykonując sekwencyjnie w jednym programie cieniowania.

Filtrowanie strumienia pozwala algorytmom na wybór podzbioru strumienia do przetworzenia. Na przykład, przy rozstrzyganiu kolizji do dalszego przetwarzania bierze się obiekty, które pozytywnie przeszły test kolizji brył otaczających. Używając kombinacji skanowania i przeszukiwania można filtrować strumienie w $O(\log n)$ przebiegach.

Mając dane różne efektywne rozwiązania powyższych elementarnych zadań można z nich składać bardziej skomplikowane aplikacje GPGPU rozwiązujące równolegle problemy, których rozwiązanie sekwencyjne trwałoby dużo dłużej.

4.3 Przykład mapowania problemu na GPGPU

Zadanie:

Dana jest jednowymiarowa tablica 65536 wektorów dwuwymiarowych liczb typu float. Należy obliczyć sumę i różnicę elementów każdego wektora.

```
float tabInput[65536][2]={ 1, 3, 4, 2, 9, 5 ... };
```

Implementacja zostanie wykonania w OpenGL i języku Cg.

Wejście:

Ponieważ rozmiar tablicy wejściowej jest większy od 8192, należy użyć tekstury dwuwymiarowej i zmapować jednowymiarowy indeks tablicy na dwuwymiarowe znormalizowane współrzędne teksturowania. Należy wyznaczyć taką szerokość i wysokość tekstury wejściowej, aby pomieściła ona 65536 elementów. Pamiętając o zaleceniu wektoryzowania przetwarzania należy uwzględnić fakt, że można zmieścić dwa wektory dwuwymiarowe w jednym czterowymiarowym. Nie uniemożliwi to przetwarzania, ponieważ aby wygenerować wynik potrzebny jest odczyt tylko danego wektora pierwotnie dwuwymiarowego. Docelowa tekstura wejściowa będzie miała 2 razy więcej kanałów, ale 2 razy mniej elementów. W tym przypadku rozmiary tekstury będą wynosiły $256 \times 128 \times 4$ kanały. Musi ona przechowywać 32-bitowe liczby typu float. Tekstura powinna być odczytywana przez sampler, który odczytuje najbliższego sąsiada.

Wyjście:

Każdemu wektorowi dwuwymiarowemu na wejściu odpowiada dwuwymiarowy wektor na wyjściu. Tekstura wyjściowa będzie miała zatem takie same parametry, co wejściowa i tak samo będzie organizowała dane dwóch kolejnych wektorów 2D w jednym wektorze 4D.

Warunki wykonania: Należy 32768 razy uruchomić kernel w programie cieniowania fragmentów na celu renderingu (RT) z podłączoną teksturą wyjściową. Podłączenie tekstury o zadanych rozmiarach do RT pozwala zacząć rysowanie na prostokątnym obszarze o tych rozmiarach. Aby zmapować piksele wejściowe na wyjściowe jeden do jednego, należy parametry widoku ustawić na te same rozmiary:

```
glViewport(0, 0, 256, 128);
```

Następnie po podłączeniu tekstury wejściowej oraz programu cieniowania fragmentów (PS) należy narysować czworokąt na całym widoku:

```
glBegin(GL_QUADS);          // rysuj prostokąt
    glTexCoord2f(0.0, 0.0); // współrzędne tekstury (s, t)
                            // dla wierzchołka czorokąta
    glVertex2f(-1.0, -1.0); // współrzędne przestrzenne (x, y)
                            // wierzchołka czworokąta
    glTexCoord2f(1.0, 0.0);
    glVertex2f(1.0, -1.0);
    glTexCoord2f(1.0, 1.0);
    glVertex2f(1.0, 1.0);
    glTexCoord2f(0.0, 1.0);
    glVertex2f(-1.0, 1.0);
glEnd();
```

W ten sposób podane współrzędne wierzchołków i teksturowania wymagają od programu cieniowania wierzchołków (VS) przepuszczenia ich bez jakiejkolwiek obróbki:

```
struct VShaderInput {
    float4 position:POSITION;
    float4 tex0:TEXCOORD0;
};

struct VShaderOutput {
    float4 position:POSITION;
    float4 tex0:TEXCOORD0;
};
```

```
VShaderOutput vertex_shader(VShaderInput input)
{
    PShaderInput result;
    result.tex0 = input.tex0;          // przekaż do rasteryzacji
                                       // otrzymaną współrzędną
                                       // teksturowania
    result.position = input.position; // przekaż do rasteryzacji
                                       // otrzymaną współrzędną
                                       // przestrzenną
    return result;
}
```

Na powyższym listingu znajduje się definicja pojedynczego wierzchołka – elementu strumienia wejściowego (VShaderInput) i wyjściowego (VShaderOutput) programu cieniowania wierzchołków. Poszczególne atrybuty wierzchołka są wektorami 4D. Znaczenie danego atrybutu określa semantyka. Semantyka `POSITION` informuje, że w tym polu elementu strumienia ma się znaleźć pozycja wierzchołka otrzymana z asemblera danych wejściowych (IA). Z kolei `TEXCOORD0` łączy wskazany element strumienia z pierwszym kanałem współrzędnych teksturowania. Analogicznie jest w przypadku definicji składowych strumienia wyjściowego. Zadaniem kernela jest przypisanie wartości strumienia wejściowego do odpowiadających im elementów strumienia wyjściowego.

Kernel:

Funkcja, która wykonuje zasadnicze obliczenia zaimplementowana zostanie w programie cieniowania fragmentów. Należy pamiętać o jak największej zwartości kodu:

```
sampler2D inputTexture;

float4x4 m = { 1,  1, 0,  0,
               1, -1, 0,  0,
               0,  0, 1,  1,
               0,  0, 1, -1 };
```

```
float4 pixel_shader(VShaderOutput input)
{
    return mul( tex2D( inputTexture, input.tex0 ), m );
}
```

Strumień wyjściowy programu cieniowania wierzchołków jest jednocześnie wejściowym dla programu cieniowania fragmentów. W tablicy stałych umieszczona została macierz 4×4. Wynik próbkowania tekstury wejściowej w miejscu wskazanym przez współrzędne teksturowania `input.tex0` jest wektorem 4D. Składają się na niego dwa wektory 2D pierwotnej tablicy wejściowej. Pomnożenie wektora 4D przez macierz powoduje obliczenie pożądanych sum i różnic i wpisanie ich w odpowiednie miejsca wyjściowego wektora 4D. Mnożenie wektora przez macierz podczas kompilacji programu cieniowania fragmentów (PS) dzieli się na cztery iloczyny skalarne, które są operacjami niepodzielnymi. Dzięki wektoryzowaniu przetwarzania zapis jest zwarty, a wykonanie najszybsze z możliwych.

4.4 Podsumowanie

Zaletą zaprezentowanego podejścia jest przede wszystkim jego prostota. Kernele implementowane jako programy cieniowania fragmentów są łatwe do tworzenia, przejrzyste w analizie i szybkie w wykonaniu. Ponadto mają, tak jak i pozostałe typy programów cieniowania, jasno zdefiniowaną konfigurację danych wejściowych i wyjściowych. Dzięki strumieniowemu modelowi przetwarzania nie trzeba się zajmować zarządzaniem wątkami i pamięcią korzystając z zalet przetwarzania równoległego. W związku z tym model cieniowania 4.0 odciąża programistę. Jednocześnie daje mu dużo swobody w korzystaniu z potencjału GPU poprzez udostępnienie w sumie dużej ilości zasobów: tekstur i rejestrów dla przetwarzania. Ponadto istnieje szereg gotowych rozwiązań typowych zadań przetwarzania równoległego, które można wykorzystać w aplikacjach GPGPU.

Następny rozdział opisuje mapowanie przedstawionych w rozdziale 3 algorytmów na GPU z wykorzystaniem modelu cieniowania 4.0.

Tabela 4.1. Porównanie możliwości poszczególnych modeli cieniowania (*na podstawie [71, 6, 107]*).

	SM 1.0	SM 2.0	SM 3.0	SM 4.0
programy cieniowania wierzchołków (**VS** – vertex shader)				
liczba slotów dla instrukcji	128	256	$\geqslant 512$	4096
maksymalna liczba wykonanych instrukcji	65536	65536	65536	65536
liczba rejestrów stałych	$\geqslant 96$	$\geqslant 256$	$\geqslant 256$	16×4096
liczba rejestrów tymczasowych	12	12	32	4096
liczba rejestrów wejściowych	16	16	16	16
liczba tekstur	-	-	4	128
kontrola przepływu operacji	-	S	S/D	D
operacje na liczbach całkowitych	-	-	-	+
operacje bitowe	-	-	-	+
programy cieniowania fragmentów (**PS** – pixel shader)				
liczba slotów dla instrukcji (odczyty tekstury + operacje arytmetyczne)	4+8	32+64	$\geqslant 512$	$\geqslant 65536$
maksymalna liczba wykonanych instrukcji (odczyty tekstury + operacje arytmetyczne)	4+8	32+64	65536	bez limitu
liczba rejestrów stałych	8	32	224	16×4096
liczba rejestrów tymczasowych	2	12	32	4096
liczba interpolowanych rejestrów wejściowych (współrzędne teksturowania + kanały koloru)	4+2	8+2	10	32 (16 bez GS)
liczba rejestrów wyjściowych	liczba celów renderingu + kanał głębokości			
liczba tekstur	8	16	16	128
kontrola przepływu operacji	-	-	S/D	D
operacje na liczbach całkowitych	-	-	-	+
operacje bitowe	-	-	-	+
mieszanie składowych wektorów	-	-	+	+
programy cieniowania geometrii (**GS** – geometry shader)				
dostępność w danym modelu	-	-	-	+
liczba rejestrów wejściowych	-	-	-	6×16
liczba rejestrów wyjściowych	-	-	-	$32 \times$ liczba wierzchołków
maksymalna liczba tworzonych wierzchołków	-	-	-	deklarowana statycznie

5

Optymalizacja algorytmów dla GPU

Niniejszy rozdział przedstawia sposób implementacji na GPU algorytmów modyfikacji siatki zaprezentowanych w rozdziale 3. Podobnie, jak w implementacji na CPU, algorytmy zostały podzielone na etapy realizowane w odpowiedniej kolejności na tablicach wierzchołków i trójkątów. Przetwarzają one dane w tych tablicach zmieniając ich zawartość, lecz nie zmieniając ich wielkości.

5.1 Propozycja odpowiednika reprezentacji indeksowej dla GPU

Dane dla programów cieniowania przechowywane są w teksturach. Na potrzeby prezentowanego przetwarzania zostały utworzone dwie tekstury:

— tekstura wierzchołków jest odpowiednikiem tablicy wierzchołków,

— tekstura trójkątów jest odpowiednikiem tablicy trójkątów.

Tekstury te służą do przechowywania danych o wierzchołkach i trójkątach w taki sposób, aby były możliwe obliczenia wykonywane na nich przez programy cieniowania. Wszystkie używane w przetwarzaniu tekstury przechowują liczby w 32-bitowym formacie zmiennoprzecinkowym (`float32`).

Tekstura wierzchołków jest czterokanałowa (RGBA). Składowe RGB każdego elementu przechowują współrzędne przestrzenne (x, y, z) reprezentowanego wierzchołka. Składowa A przechowuje flagę istnienia wierzchołka (e). Ponieważ konieczna jest możliwość przechowywania więcej niż 8192 elementów (maksymalny rozmiar pionowy lub poziomy tekstury w modelu cieniowania 4.0), zastosowano dwuwymiarową standardową prostokątną teksturę o wymiarach odpowiednich do przechowywania pożądanej liczby elementów – wierzchołków. Jednowymiarowy indeks z tablicy wierzchołków będzie odpowiadał dwuwymiarowym współrzędnym teksela. W dalszej części opisu algorytmu będą używane dwa pojęcia opisujące położenie danej w teksturze:

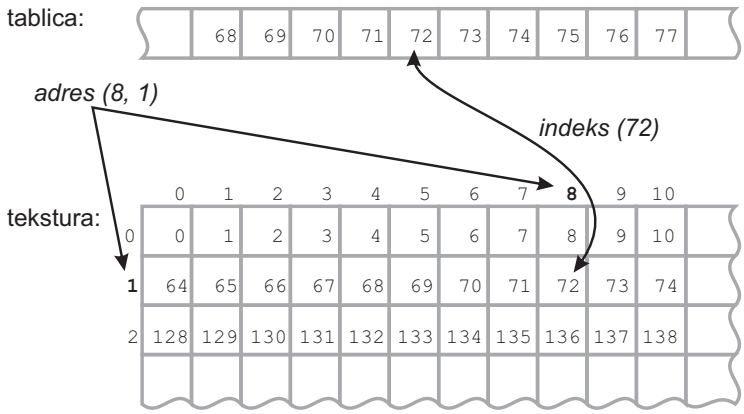
— *adres* – dwuwymiarowe współrzędne (s, t) teksela w teksturze,
— *indeks* – jednowymiarowy indeks (i) teksela w teksturze.

Dla tekstury o zadanych wymiarach istnieje jednoznaczne przejście z *adresu* do *indeksu* i odwrotnie (wzór 5.1, rys. 5.1):

$$i = s + w \cdot t, \tag{5.1}$$

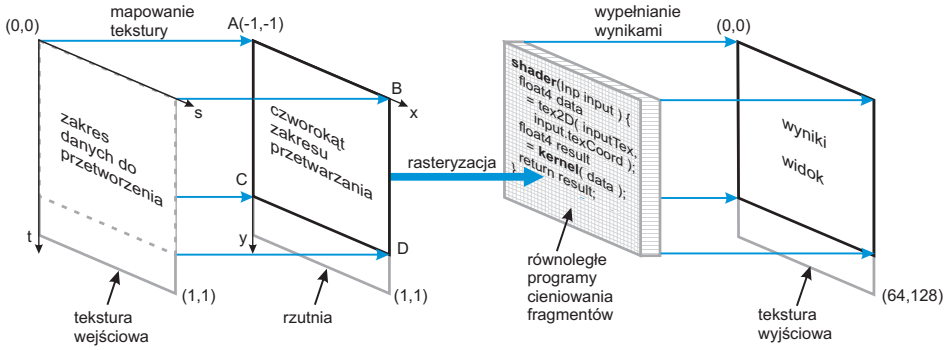
gdzie w – szerokość tekstury przechowującej tablicę.

Reprezentacja tablicy jednowymiarowej w dwuwymiarowej teksturze przechowuje nadmiarowe elementy tekstury, których nie ma w tablicy. Jednak przy standardowych teksturach o rozmiarach $2^m \times 2^n$ nie da się tego uniknąć, gdyż liczba elementów w takiej teksturze jest również potęgą liczby 2. Istotne jest, aby unikać przetwarzania tych nadmiarowych elementów. W schemacie przetwarzania jednoetapowego opisanym w podrozdziale 4.2.3 można to zrobić przy pomocy takiego narysowania wierzchołków **czworokąta zakresu przetwarzania**, aby wypełniać tylko potrzebną (a więc



Rysunek 5.1. Indeksy tablicy odpowiadające adresom tekstury.

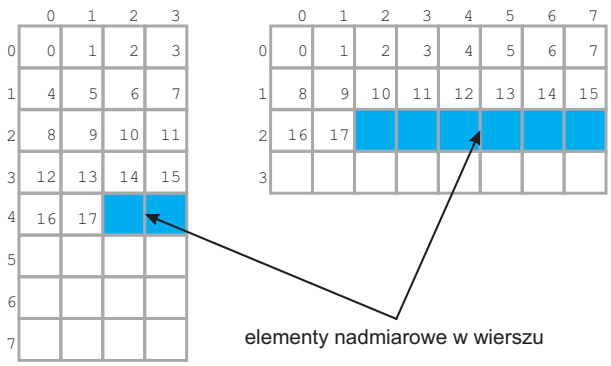
zawierającą dane) część tekstury wyjściowej podczas przetwarzania. Ideę pokazuje rysunek 5.2.



Rysunek 5.2. Przetwarzanie danych z tekstury za pomocą rysowania czworokąta zakresu przetwarzania.

W ten sposób można wyeliminować z przetwarzania nadmiarowe wiersze tekstury. Zazwyczaj liczba elementów tablicy nie jest wielokrotnością szerokości tekstury. Eliminacja elementów nadmiarowych w ostatnim wierszu tekstury zawierającym dane odbywa się więc w kodzie programu cieniowania, który za pomocą instrukcji warunkowej eliminuje te elementy, które

oznaczono flagą, jako nieistniejące. Konieczność optymalizacji liczby wykonywanych instrukcji warunkowych narzuca dodatkowy warunek: szerokość tekstury powinna być mniejsza lub równa jej wysokości. Dzięki temu liczba elementów nadmiarowych w ostatnim przetwarzanym wierszu jest mniejsza, niż byłaby, gdyby szerokość tekstury była większa od jej wysokości (rys. 5.3).



Rysunek 5.3. Elementy nadmiarowe w wierszu. Jest ich mniej, gdy tekstura ma mniejszą szerokość niż wysokość.

Bazując na powyższych zasadach dotyczących rozmiarów tekstury, przykładowa tablica 8002 wierzchołków będzie przechowywana, jako tekstura o rozmiarze 64×128 tekseli. W przypadku przetwarzania tej tekstury za pomocą czworokąta zakresu, każdy z 8002 wierzchołków zapamiętanych w teksturze wejściowej zostaje odczytany przez odpowiadające mu wywołanie programu cieniowania fragmentów i zapisany w wyjściowej teksturze podłączonej do RT. Należy przy tym pamiętać, że o ile współrzędne teksturowania tekstury wejściowej mieszczą się w zakresie $< (0,0),(1,1) >$, to rysowanie odbywa się na rzutni (rys. 5.2). W przedstawionym przykładzie wyniki przetwarzania trafiają w odpowiednie miejsce dzięki:

— ustawieniu jako tekstury wyjściowej rozmiaru 64×128,
— ustawieniu rozmiaru widoku na rozmiar tekstury wyjściowej (64×128),
— narysowaniu czworokąta zakresu przetwarzania: jego wierzchołki umieszczone są w punktach $A = (-1,-1)$, $B = (1,-1)$, $C = (-1, 0.96875)$,

$D = (1, 0.96875)$. Dolna krawędź czworokąta (CD) pokrywa się z ostatnim wierszem tekstury zawierającym dane wierzchołków. W rozpatrywanym przykładzie jest to wiersz 126 ponieważ zawiera dane wierzchołka o indeksie 8001. W znormalizowanych współrzędnych rzutowania $y = (126/128 - 0.5) * 2$. Więc taką właśnie współrzędną y mają punkty C i D,

— przypisane wierzchołkom czworokąta zakresu współrzędne teksturowania: $T_A = (0, 0)$, $T_B = (1, 0)$, $T_C = (0, 126/128)$, $T_D = (0, 126/128)$ Dzięki temu program cieniowania będzie próbkował teksturę wejściową w odpowiednich miejscach zadanych współrzędnymi teksturowania z zakresu $< (0, 0), (1, 1) >$.

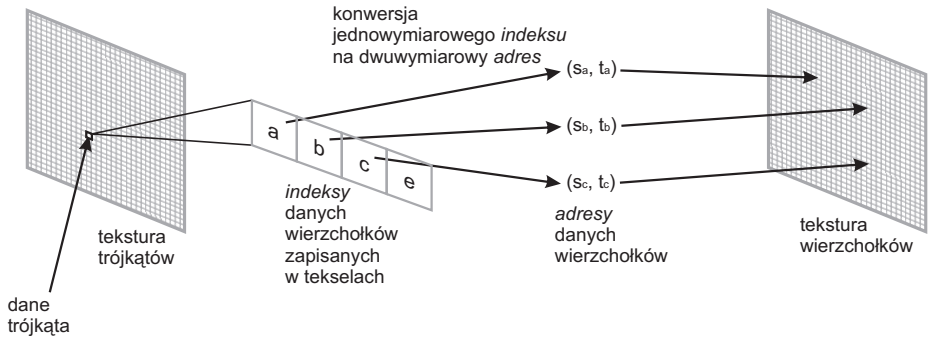
Dzięki opisanemu mechanizmowi każde wywołanie programu cieniowania fragmentów (PS) pobierze dane z odpowiedniego miejsca tekstury wejściowej i po przetworzeniu zapisze je w odpowiadającemu mu miejscu tekstury wyjściowej.

Sposób budowania i przetwarzania **tekstury trójkątów** jest analogiczny. Elementy tekstury trójkątów również są wektorami czterowymiarowymi. Składowe RGB przechowują *indeksy* kolejnych trzech wierzchołków trójkąta w teksturze wierzchołków, a składowa A przechowuje flagę istnienia trójkąta (e). Aby obliczyć współrzędne wierzchołków danego trójkąta należy, analogicznie do tablicowej reprezentacji indeksowej:

— odczytać *indeksy* (a, b, c) jego wierzchołków z tekstury trójkątów,—
odwzorować te indeksy na trzy *adresy* w tablicy wierzchołków,

— odczytać teksturę wierzchołków spod każdego z trzech *adresów*
(rys. 5.4).

Indeksy wierzchołków zapisane w teksturze dla każdego trójkąta są uporządkowane w kolejności rysowania zgodnie z konwencją zapisu orientacji trójkątów. Ponadto wierzchołek o najniższym *indeksie* jest zapisany jako pierwszy (w kanale R). Ma to znaczenie dla optymalizacji Algorytmu 2 z rozdziału 3.6.



Rysunek 5.4. Odczyt współrzędnych wierzchołków trójkąta.

Podczas projektowania przetwarzania tekstur należy pamiętać o podstawowym ograniczeniu programów cieniowania: tekstura wejściowa nie może być jednocześnie wyjściową. Dlatego każde przetworzenie czy to danych wierzchołków, czy trójkątów łączy się z wypełnieniem danymi nowej tekstury. Dlatego też tekstura wierzchołków i trójkątów są zdublowane a przetwarzanie odbywa się według schematu ping-pong (str. 112).

5.2 Operacje strumieniowe

Aby przetwarzać tekstury opisane w poprzednim podrozdziale zdefiniowano cztery operacje przetwarzania różniące się rodzajem tekstury wejściowej i wyjściowej oraz sposobem ich przetwarzania. Z operacji tych zbudowane są potoki przetwarzania realizujące algorytmy z rozdziału 3.6.

Operacja VQUAD (czworokąt zakresu w celu przetwarzania danych wierzchołków) – jest rodzajem operacji **mapowania** opisanej w rozdziale 4.2.5. Przetwarza dane wierzchołków w ten sposób, że mapuje teksturę wierzchołków na drugą teksturę wierzchołków, modyfikując dane wierzchołków przy pomocy programu cieniowania fragmentów. Cechy tej operacji:

— wejście: tekstura lub tekstury o wymiarach tekstury wierzchołków,
— wyjście: tekstura o wymiarach tekstury wierzchołków, rozmiar widoku ustawiony na wielkość tekstury wierzchołków wyrażonej w pikselach,

rozmiar czworokąta zakresu przetwarzania ustawiony tak, aby przetworzyć dane wszystkich wierszy tekstury zawierających dane wierzchołków (tak jak opisano w poprzednim podrozdziale),
— PS: dla każdego wierzchołka odczytuje jego współrzędne przestrzenne i wykonuje na nich przetwarzanie (kernel),
— zapisuje: wartość obliczoną na podstawie danych z tekstury wejściowej do tego samego miejsca tekstury wyjściowej.

Operacja TQUAD (czworokąt zakresu w celu przetwarzania danych trójkątów) – jest również operacją **mapowania**. Analogicznie, jak VQUAD przetwarza dane trójkątów:
— wejście: tekstura lub tekstury o wymiarach tekstury trójkątów i wierzchołków,
— wyjście: tekstura o wymiarach tekstury trójkątów, rozmiar widoku ustawiony na wielkość tekstury trójkątów wyrażonej w pikselach, rozmiar czworokąta zakresu przetwarzania ustawiony tak, aby przetworzyć dane wszystkich wierszy tekstury zawierających dane trójkątów,
— PS: dla każdego trójkąta odczytuje indeksy wierzchołków składowych i wykonuje na nich przetwarzanie (kernel),
— zapisuje: wartość obliczoną na podstawie danych z tekstury wejściowej trójkątów do tego samego miejsca tekstury wyjściowej.

Operacja TPTS (rysowanie punktów reprezentujących trójkąty) – jest rodzajem operacji **scatter** opisanej w rozdziale 4.2.5. Przetwarza dane w ten sposób, że aplikacja CPU rysuje punkty (`GL_POINTS`) o współrzędnych będących *adresami* kolejnych trójkątów w teksturze trójkątów. Można to zrobić przy pomocy poniższego kodu OpenGL:

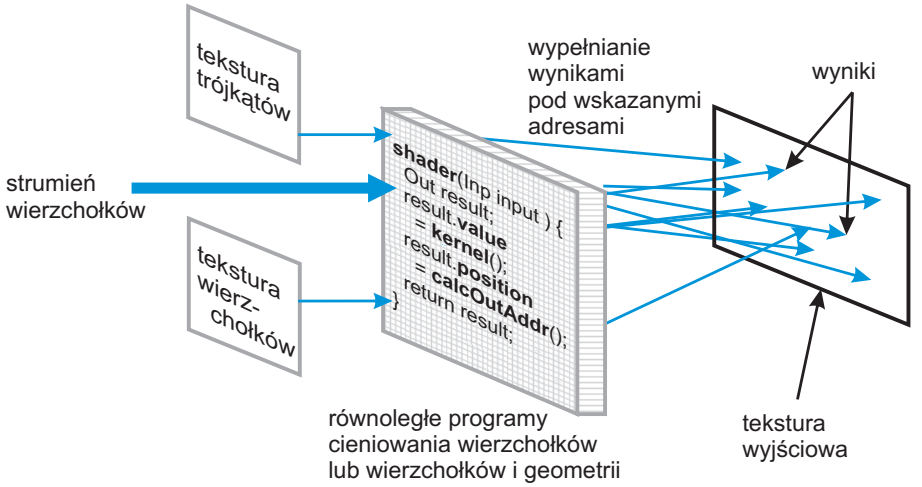
```
glBegin(GL_POINTS);
// tnum - liczba trójkątów w teksturze
for ( int i=0; i<tnum; ++i )
  // ttexw - szerokość tekstury w pikselach
  // ttexh - wysokość tekstury w pikselach
    glVertex2f( (i%ttexw + 0.5f) / ttexw, (i/ttexw + 0.5f) / ttexh );
glEnd();
```

Punkty te są przetwarzane przez program cieniowania wierzchołków, VS (opcjonalnie przez program cieniowania geometrii, GS). Program ten w pożądany w kernelu sposób oblicza wartości i *adresy* w teksturze wierzchołków, pod które te wartości należy zapisać. Przetwarzanie to nie zmienia więc całej tekstury wyjściowej a tylko teksele, które zostały wskazane przez VS/GS. Realizowane jest w poniższy sposób:

- wejście: tekstura lub tekstury o wielkości tekstury trójkątów i wierzchołków, aplikacja CPU „rysuje" punkty, czyli uruchamia VS tyle razy, ile jest trójkątów w teksturze trójkątów,
- wyjście: tekstura o wielkości tekstury wierzchołków, rozmiar widoku ustawiony na wielkość tekstury wierzchołków,
- VS: dla każdego trójkąta odczytuje *indeksy* wierzchołków składowych, ma więc dostęp do ich *adresów* w teksturze wierzchołków oraz współrzędnych przestrzennych wierzchołków z tej tekstury; ma za zadanie wygenerować *adres* zapisu do tekstury wyjściowej oraz wartość, którą należy tam zapisać (scatter),
- opcjonalny GS: używany, gdy jeden wygenerowany przez aplikację punkt powinien zostać przetworzony na trzy wpisy do tekstury wyjściowej (VS nie może dodawać nowych punktów, a GS może),
- zapisuje: wartość od VS/GS pod *adresem* przez nie wskazanym.

Ideę przetwarzania pokazuje rysunek 5.5.

Operacja VPTS (rysowanie punktów reprezentujących wierzchołki) jest również operacją **scatter**. Przetwarza dane w ten sposób, że aplikacja CPU rysuje punkty o współrzędnych będących *adresami* kolejnych wierzchołków w teksturze wierzchołków (analogicznie, jak trójkąty w TPTS). Punkty te są przetwarzane przez program cieniowania wierzchołków (VS). Program ten w pożądany w kernelu sposób oblicza wartości i *adresy* w teksturze wierzchołków, pod które te wartości należy zapisać. Przetwarzanie to nie zmienia więc całej tekstury wyjściowej a tylko teksele, które zostały wskazane przez VS. Realizowane jest w poniższy, analogiczny sposób, jak w TPTS:



Rysunek 5.5. Operacja scatter realizowana na programie cieniowania wierzchołków (VS) lub geometrii (GS).

— wejście: tekstura lub tekstury o wielkości tekstury wierzchołków, aplikacja CPU „rysuje" punkty, czyli uruchamia VS tyle razy, ile jest wierzchołków w teksturze wierzchołków,
— wyjście: tekstura o wielkości tekstury wierzchołków, rozmiar widoku ustawiony na wielkość tekstury wierzchołków,
— VS: dla każdego wierzchołka odczytuje dane z nim związane w teksturze wejściowej; ma za zadanie wygenerować *adres* zapisu do tekstury wyjściowej oraz wartość, którą należy tam zapisać (scatter),
— zapisuje: wartość obliczoną przez VS pod *adresem* przez niego wskazanym.

Kolejne podrozdziały opisują użycie zaproponowanej w poprzednim podrozdziale reprezentacji i zdefiniowanych w tym podrozdziale operacji w kolejnych algorytmach przetwarzania siatki obiektu.

5.3 Deformacja siatki

W skład algorytmu deformacji siatki (Algorytm 1) wchodzi obliczenie normalnych trójkątów i normalnych wierzchołków, wektorów wzdłuż krawędzi oraz wektorów przesunięcia, a następnie przesunięcie wierzchołków siatki w nowe miejsca.

Wektory normalne trójkątów (N_f) są trójwymiarowe i jest ich tyle samo, co trójkątów w siatce. W związku z tym potrzebna jest na ich przechowywanie **tekstura normalnych trójkątów**, jako analog tablicy normalnych. Jest ona dwuwymiarowa i ma takie same wymiary jak tekstura trójkątów. Obliczenie normalnych trójkątów jest więc operacją TQUAD:

— wejście: tekstura trójkątów, tekstura wierzchołków,
— wyjście: tekstura normalnych trójkątów,
— dla każdego trójkąta odczytuje *indeksy* wierzchołków składowych oraz współrzędne ich wierzchołków spod tych *indeksów* w teksturze wierzchołków,
— oblicza normalną trójkąta zgodnie ze wzorem (2.9),
— zapisuje: obliczoną normalną trójkąta.

Mając normalne trójkątów można obliczyć normalne wierzchołków. Wiąże się to z zapisem wartości pod *adresy* **tekstury normalnych wierzchołków** wskazane przez *indeksy* wierzchołków trójkątów siatki, więc jest operacją TPTS z włączonym blendingiem[1]:

— wejście: tekstura trójkątów, tekstura normalnych trójkątów (N_f), tekstura wierzchołków,
— wyjście: tekstura normalnych wierzchołków (N) wypełniona wstępnie zerami,
— VS: odczytuje *indeksy* wierzchołków danego trójkąta, zwraca te *indeksy* oraz normalną (N_f) odczytaną z tekstury normalnych trójkątów,

[1] blending jest sumą ważoną wartości fragmentu i wartości piksela przechowywanego w celu renderingu (RT).

— GS: odczytuje przekazane z VS wartości, konwertuje *indeksy* na *adresy* w teksturze wyjściowej i generuje trzy punkty różniące się *adresem* zapisu w teksturze normalnych wierzchołków i mające tę samą wartość będącą normalną (N_f) przetwarzanego trójkąta,
— zapisuje: normalne trójkątów (N_f) pod adresami w teksturze normalnych wierzchołków (N).

Opisany potok dokonuje zapisu pod danym *adresem* wyjściowej tekstury normalnych wierzchołków wartość tyle razy, ile trójkątów odwołuje się do wierzchołka odpowiadającego temu *adresowi*. Nie można przy pomocy operacji programu cieniowania odczytać poprzedniego wpisu i dodać do niego aktualnie wygenerowanego – jest to operacja read-modify-write (str. 112). Jedynym etapem programowalnego potoku renderingu, który zapewnia tę operację jest etap łączenia wyjścia (OM), który oprócz testów wykonuje także blending. Ustawienie schematu mieszania na dodawanie koloru źródła do celu (`glBlendFunc(GL_ONE, GL_ONE)`) spowoduje, że wygenerowana wartość normalnej trójkąta (N_f) zostanie **dodana** do zapisanych tam poprzednio normalnych innych trójkątów. Wskutek tego, dzięki blendingowi, uzyskać można operację sumowania normalnych trójkątów w normalnej wierzchołka (N).

Po powyższym etapie przetwarzania należy znormalizować obliczone sumy dodatkową operacją VQUAD: jej wejściem jest wygenerowana właśnie tekstura nieznormalizowanych normalnych wierzchołków a wyjściem tekstura normalnych wierzchołków.

Następnie należy obliczyć wektory przesunięcia M wierzchołków. W tym celu należy zsumować wektory E_j krawędzi zbudowanych na tych wierzchołkach. Odpowiednikiem linii $20 - 27$ Algorytmu 1 jest operacja TPTS z włączonym blendingiem:
— wejście: tekstura trójkątów, tekstura wierzchołków,
— wyjście: tekstura wektorów M_E wypełniona wstępnie zerami,
— VS: odczytuje *indeksy* wierzchołków spod otrzymanych współrzędnych oraz przekazuje te *indeksy* do GS,

— GS: oblicza wektory (E_j) wierzchołków przetwarzanego trójkąta i generuje trzy punkty różniące się pozycją wyjściową i wektorem czterowymiarowym $(E_{j_x}, E_{j_y}, E_{j_z}, 1)$,
— zapisuje: obliczone zsumowane wektory E_j oraz ich liczbę w ostatniej składowej wektora w.

Następnie operacja VQUAD dzieli sumę wektorów przez ich liczbę, aby uzyskać wektory M_E w teksturze wyjściowej.

Mając obliczone wszystkie niezbędne składniki wektora przesunięcia można przesunąć wierzchołki siatki w nowe miejsce. Operacja VQUAD z włączonym blendingiem:
— wejście: tekstura normalnych wierzchołków, tekstura wektorów M_E,
— wyjście: tekstura wierzchołków,
— PS: oblicza wektory M na podstawie wzoru (3.6), i danych stałych k_N, k_E. Jeśli używany jest współczynnik k_g, to jest tu także stałą, jeśli $k_l()$, to jest tu fragmentem kodu programu cieniowania lub instrukcją próbkowania tekstury (podrozdział 3.4),
— zapisuje: wektory przesunięcia M które są dodawane dzięki blendingowi do bieżących położeń zawartych w teksturze wyjściowej wierzchołków.

5.4 Usuwanie krawędzi siatki

Usuwanie krawędzi siatki (Algorytm 2) jest najbardziej złożoną fazą przetwarzania. Po jej wykonaniu krawędzie mniejsze od zadanego progu k_r powinny zostać usunięte. Pierwszym etapem przetwarzania jest stworzenie przy pomocy operacji VQUAD **tekstury przeindeksowań wierzchołków** (o wielkości tekstury wierzchołków) – odpowiednika pola pozycji przeindeksowania (i) w algorytmie na CPU. Po tym etapie każdy jej element zawiera współrzędne (*adres*) samego siebie. Jest to odpowiednik sytuacji, w której indeksy tablicy są równe wartościom zapisanym pod tymi indeksami.

Następnie realizowana jest operacja TPTS, która realizuje działania zawarte w liniach $5-15$ Algorytmu 2:

— wejście: tekstura wierzchołków, tekstura trójkątów,
— wyjście: tekstura przeindeksowań,
— VS: realizuje działania z linii $5-15$ algorytmu – sprawdza warunki długości krawędzi oraz generuje wartość *indeksu* wierzchołka usuwanego i *indeksów* wierzchołka będącego drugim końcem usuwanej krawędzi (operacja przeindeksowania wierzchołków),
— zapisuje: *adres* tego wierzchołka z pary, który pozostaje, pod adresem wierzchołka z pary, który zostanie usunięty (przeindeksowany na ten, który pozostaje).

Tak zaprogramowany etap tworzenia tekstury przeindeksowań zmienia w niej tylko te teksele, które odpowiadają wierzchołkom przeznaczonym do usunięcia. Następnie tekstura przeindeksowań zostaje przetworzona operacją VQUAD, aby usunąć *łańcuchy przeindeksowań* (linie $16-20$ Algorytmu 2).

Linie $21-23$ Algorytmu 2 zakładają szeregowe przetwarzanie. Współrzędne **kolejnych** wierzchołków, na które przeindeksowywane są usuwane wierzchołki, są z nimi uśredniane. Wynik przetworzenia pewnego wierzchołka może więc wpływać na kolejne przetwarzane w pętli wierzchołki. Ten fragment algorytmu wymaga więc przeformułowania, aby nadawał się dla przetwarzania równoległego. Jest to realizowane przez obliczenie średniej ze współrzędnych wszystkich wierzchołków, które są przeindeksowywane na dany wierzchołek i ustawienie tej średniej jako jego pozycji (do tej średniej dolicza się również pozycja jego samego). Operacja VPTS z blendingiem:

— wejście: tekstura wierzchołków, tekstura przeindeksowań,
— wyjście: tekstura wierzchołków,
— VS: generuje jako wartość wyjściową współrzędne przestrzenne przetwarzanego wierzchołka $(x, y, z, w = 1)$ a jako adres zapisu *adres* wierzchołka na który jest przeindeksowywany,

— zapisuje: zsumowane współrzędne wierzchołków, w pozycji wierzchołka, na który zostaną przeindeksowane.

Kolejna operacja VQUAD dzieli zsumowane współrzędne każdego wierzchołka przez w otrzymując ich średnią oraz jednocześnie usuwa (ustawia współrzędną $w = 0$) niepotrzebne już wierzchołki siatki (linie $24 - 25$ Algorytmu 2).

Następnie należy zrealizować działania przedstawione w liniach $26 - 33$ algorytmu. Jest to przetwarzanie danych trójkątów, więc ma zastosowanie operacja TQUAD:

— wejście: tekstura trójkątów, tekstura przeindeksowań
— wyjście: tekstura trójkątów,
— PS: jeżeli *indeks* któregoś wierzchołka trójkąta jest zaznaczony do usunięcia w teksturze przeindeksowań, to zmień mu *indeks* na *indeks* wierzchołka, na który zostaje przeindeksowany,
— zapisuje: trójkąt ze zmienionymi wartościami *indeksów* wierzchołków.

Na końcu należy wykonać działania z linii $34 - 37$ Algorytmu 2: przy pomocy operacji VQUAD usunąć z tekstury trójkątów ($w = 0$) te trójkąty siatki, które mają zdublowany wierzchołek.

5.5 Usuwanie sklejonych trójkątów

Usuwanie sklejonych trójkątów z siatki (Algorytm 3) polega na odnalezieniu wierzchołków w siatce, które mają tylko dwa powiązane trójkąty i usunięcie ich oraz tych trójkątów. *Tekstura liczby trójkątów* jest odpowiednikiem pola liczby trójkątów (t) w wierzchołku w Algorytmie 3. Najpierw tekstura ta jest zerowana, a następnie operacja TPTS z blendingiem dodaje do elementów tej tekstury odpowiadających wierzchołkom, liczbę trójkątów powiązanych (linie $6 - 10$ algorytmu 3):

— wejście: tekstura trójkątów,
— wyjście: tekstura liczby trójkątów (t) wstępnie wyzerowana,

— VS: odczytuje *indeksy* wierzchołków danego trójkąta, zwraca te *indeksy* do GS,
— GS: odczytuje przekazane z VS *indeksy*, konwertuje je na *adresy* i generuje trzy punkty o wartości 1 różniące się *adresem*,
— zapisuje: sumę jedynek wygenerowanych przez GS pod danym adresem w teksturze wyjściowej.

Następnie operacją TQUAD wystarczy przejść teksturę trójkątów i usunąć każdy trójkąt, który indeksuje wierzchołek mający wartość 2 w teksturze liczby trójkątów (linie $11-14$ algorytmu). Analogicznie operacją VQUAD należy przejść teksturę wierzchołków i usunąć każdy, który w teksturze liczby trójkątów ma wartość 2 (linie $15-18$).

Ostatnią operacją nie związaną z korekcją, tylko z poprawnością danych, jest operacja TQUAD, która sortuje indeksy wierzchołków w trójkątach, gdyż podczas przeindeksowywania w Algorytmie 2 ten porządek mógł być zaburzony.

5.6 Rendering obiektu

Całe przetwarzanie siatki geometrii odbywa się na GPU, dlatego przed renderingiem nie należy ściągać z pamięci wideo tekstur zawierających aktualną jego geometrię i renderować przy pomocy CPU. Również do renderingu warto wykorzystać programy cieniowania, które na podstawie danych z tekstury wierzchołków i trójkątów wygenerują geometrię obiektu. Przychodzi tu z pomocą operacja TPTS renderująca wyjątkowo trójkąty, a nie punkty:
— wejście: tekstura wierzchołków, tekstura trójkątów,
— wyjście: bufor koloru do wyświetlenia na ekranie,
— VS: odczytuje *indeksy* wierzchołków danego trójkąta, zwraca te *indeksy* do GS,
— GS: odczytuje przekazane z programu cieniowania wierzchołków *indeksy*, przelicza ja na *adresy* w teksturze wierzchołków, a następnie generuje trzy wierzchołki trójkąta na podstawie ich położeń przestrzennych

oraz macierzy przekształceń współrzędnych obiektu do współrzędnych rzutowania,
— etap rasteryzacji tworzy strumień fragmentów z prymitywów wygenerowanych przez GS,
— PS: wykonuje operacje związane z wygenerowaniem barwy fragmentu,
— zapisuje: obliczony przez PS kolor piksela.

Przekształcenie współrzędnych wierzchołka do współrzędnych rzutowania oraz obliczanie końcowego koloru piksela w programie cieniowania fragmentów są typowymi dla potoku renderingu czynnościami, dzięki którym na ekranie można obejrzeć wyrenderowany obiekt.

5.7 Podsumowanie

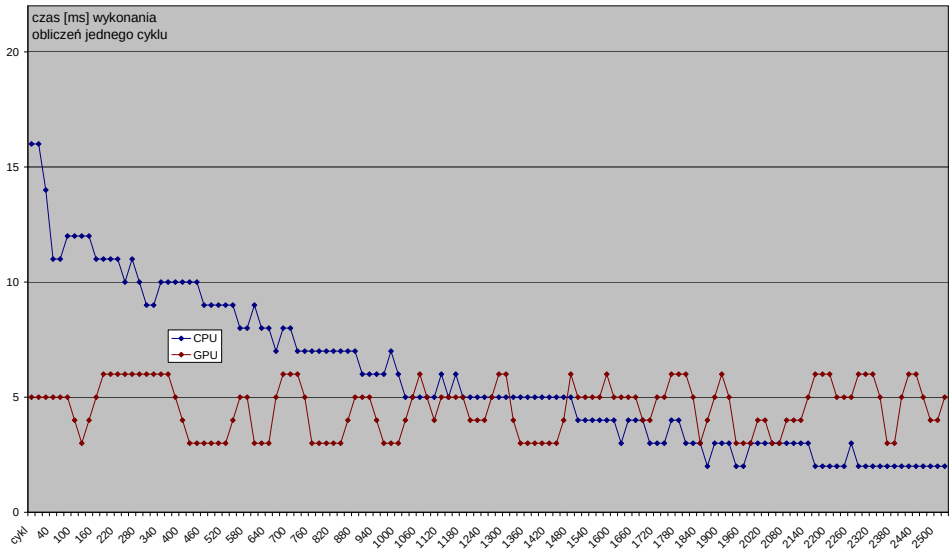
Podsumowaniem niniejszego rozdziału jest porównanie czasów trwania cyklu przetwarzania trzech obiektów na CPU oraz na GPU na komputerze zawierającym procesory:
— CPU: Intel Core 2 Duo P8400 (2,26 GHz, 1066 MHz FSB),
— GPU: nVidia GeForce 9600M GT (32 procesory strumieniowe, rdzeń 500 MHz, zegar jednostki cieniowania 1250 MHz).

Użyte zostały trzy obiekty testowe o różniącej się liczbie trójkątów w siatce. Ich parametry i osiągnięte średnie wyniki przetwarzania przedstawia tabela 5.1.

Tabela 5.1. Wyniki testów szybkości dla obiektów testowych. Czasy podane są w milisekundach.

obiekt	początkowa liczba trójkątów w siatce	średni czas trwania cyklu obliczeń na CPU	średni czas trwania cyklu obliczeń na GPU
A	192	< 1	4
B	16000	5.8	4.6
C	32000	20	6

Wykresy 5.6 (obiekt B) i 5.7 (obiekt C) przedstawiają wykresy osiągniętych czasów przetwarzania w wybranych cyklach. Można zaobserwować widoczny wzrost prędkości przetwarzania przy użyciu GPU w stosunku do szybkości przetwarzania na CPU. Najwyraźniejsza, kilkukrotna różnica czasów jest widoczna dla pierwszych cykli przetwarzania dla obiektu C posiadającego siatkę o największej liczbie trójkątów. Głównym wnioskiem jest zatem **opłacalność przenoszenia na GPU przetwarzania dużych obiektów**.



Rysunek 5.6. Czas wykonania obliczeń pojedynczego cyklu w milisekundach na CPU i na GPU. Początkowo obiekt B składał się z 16000 trójkątów.

Czas trwania pojedynczego cyklu przetwarzania na CPU wraz z kolejnymi cyklami zmniejsza się. Wykres 5.7 pokazuje zależność czasu przetwarzania na CPU od liczby pozostałych w siatce trójkątów. Widać, że czas w implementacji na CPU zależy od liczby pozostałych trójkątów w siatce obiektu.

Z kolei czas przetwarzania na GPU wynosi około 4-5 ms i w niewielkim stopniu zależy od początkowej liczby trójkątów w siatce. Ma to dwie przyczyny:

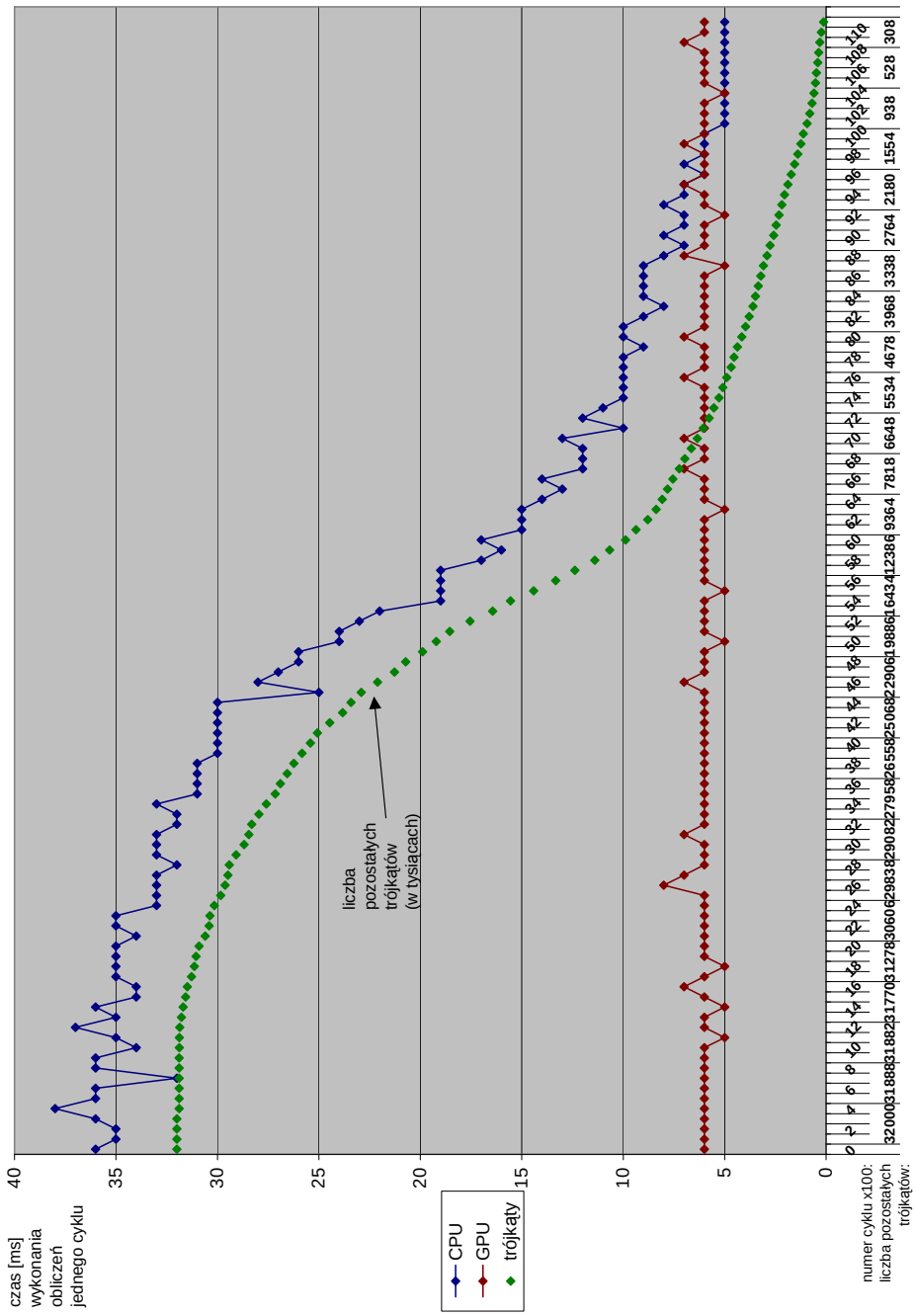
— czas przetwarzania na GPU nie może być mniejszy niż około 4 milisekundy ze względu na narzut spowodowany komunikacją z CPU, uruchamianiem kolejnych etapów przetwarzania, podłączaniem celów renderingu oraz tekstur, czyli wszystkich tych czynności, które nie wiążą się z przetwarzaniem równoległym, ale są niezbędne, aby je uruchomić,

— mechanizmy kontroli przepływu programu użyte w implementacji algorytmów na GPU oferowane przez model cieniowania 4.0 nie odrzucają z dalszego przetwarzania trójkątów (i wierzchołków) siatki zaznaczonych jako usunięte. Jeśli byłaby potrzeba optymalizacji szybkości algorytmu na GPU należałoby użyć do odrzucania innych mechanizmów niż instrukcje kontroli przepływu programu (np. Z-cullingu opisanego na stronie 116).

Implementacja na GPU pozwoliła na osiągnięcie szybkości przetwarzania potrzebnej w **aplikacjach renderingu czasu rzeczywistego** nawet dla dużych obiektów[2] (w sensie liczby trójkątów). Pozwala na animację z szybkością nawet powyżej 150fps[3].

[2] obiekty modelowane z myślą o renderingu czasu rzeczywistego mają obecnie siatki nie większe niż kilkadziesiąt tysięcy trójkątów.

[3] fps – ramek na sekundę.



Rysunek 5.7. Czas wykonania obliczeń pojedynczego cyklu w milisekundach na CPU i na GPU. Początkowo obiekt C składał się z 32000 trójkątów.

6

Wizualizacja bryły lodu

Opisany w tym rozdziale materiał wieloskładnikowy lodu powstał jako część badań autora nad wizualizacją lodu [86, 85]. Bazuje on na obserwacji wyglądu lodu, który może być bardzo różny w zależności od tego, czy lód znajduje się w równowadze temperatur z otoczeniem, czy przeciwnie topi się lub oziębia. Rendering w czasie rzeczywistym złożonej interakcji między światłem i lodem nie był do tej pory przedmiotem intensywnych badań. W większości są to prace nad technikami opartymi na fizyce nie przeznaczonymi dla renderingu czasu rzeczywistego, na przykład dotyczące formowania się sopli [41] lub kryształów lodu [44, 45]. Prace związane z topnieniem i deformacją obiektów omówione we wstępie do niniejszej monografii koncentrują się na wizualizacji materiałów innych niż lód. Nieliczne prace dotyczące renderingu czasu rzeczywistego lodu [76] pojawiły się w podobnym czasie, co prace autora i podobnie bazują na obserwacji wyglądu lodu i konstrukcji wirtualnego materiału na jej podstawie.

Topniejący lód ma gładką powierzchnię silnie odbijającą zwierciadlanie i o stosunkowo dużej przezroczystości, dzięki czemu widać jego wnętrze. W pozostałych przypadkach, gdy woda krzepnie, albo lód znajduje się

w równowadze termodynamicznej, na jego powierzchni tworzy się duża liczba mikroskopijnych kryształów. Każdy z nich odbija światło zwierciadlanie, ale mała powierzchnia każdego z nich powoduje, że jest ono postrzegane raczej jako światło rozproszone. Lód jest półprzezroczystym materiałem i jego interakcja ze światłem zmienia się w zależności od wewnętrznej struktury, która zależy od warunków zewnętrznych takich jak temperatura i ciśnienie. Główne cechy wyglądu lodu to [76]:

1. mlecznobiały wygląd pochodzący od pęcherzyków powietrza we wnętrzu bryły,
2. odbicia zwierciadlane na zewnętrznej powierzchni i rozbłyski we wnętrzu (na szczelinach),
3. transmisja światła zza przezroczystego obiektu powoduje załamanie na granicach ośrodków[1],
4. nieregularny kształt i drobne nierówności powierzchni, przede wszystkim, gdy lód znajduje się w równowadze termodynamicznej lub krzepnie.

Są to tylko najważniejsze cechy wyglądu lodu, ale połączenie ich w jeden materiał wieloskładnikowy powoduje osiągnięcie wiarygodnej wizualizacji.

6.1 Światło rozproszone

Mlecznobiały kolor lodu jest obserwowany, gdy lód znajduje się w równowadze termodynamicznej. Jest to kolor zarówno jego wnętrza (pęcherzyki powietrza rozpraszają podpowierzchniowo), jak i zewnętrza (kryształki o bardzo małych powierzchniach rozpraszają na powierzchni). Kolor ten można zamodelować jako składniki ambient i diffuse (rys. 6.1a) materiału w modelu Phonga (6.1):

$$I = c_a + c_d(N \cdot L), \tag{6.1}$$

[1] względny współczynnik załamania $\eta = 1,31$ dla lodu w temperaturze $-7^{\circ}C$ i dla długości fali $\lambda = 589nm$.

gdzie:

I – kolor wyjściowy

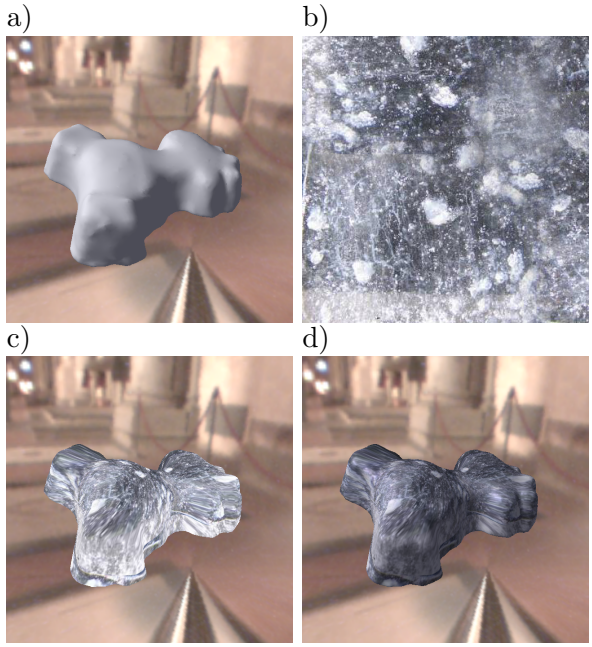
c_a – składnik ambient (w modelu Phonga iloczyn $I_{zra} \cdot k_a$),

c_d – składnik diffuse (w modelu Phonga iloczyn $I_{zrd} \cdot k_d$),

N – normalna,

L – wektor kierunkowy światła.

Wygląd wnętrza lodu jest imitowany przez teksturę fotograficzną przezroczystej bryły lodu (rys. 6.1b). Jest to bardzo niedokładne rozwiązanie, ale ma tę zaletę, że nie wymaga modelowania wnętrza bryły. W [76] wnętrze bryły jest zamodelowane geometrycznie i wymaga renderingu wieloprzebiegowego oraz dodatkowego narzutu czasowego, gdy geometria wizualizowanego obiektu zmienia się. Rysunek 6.1c) pokazuje wygląd obiektu po nałożeniu na niego tekstury przy pomocy operatora replace, natomiast rysunek 6.1d) pokazuje modulację tej tekstury przy pomocy rozkładu oświetlenia.



Rysunek 6.1. Światło rozproszone: a) składniki ambient i diffuse, b) fotografia wyglądu lodu, c) tekstura fotografii lodu zmapowana na obiekt d) modulacja tekstury wyglądu lodu przy pomocy oświetlenia ambient i diffuse.

Widać, że ostatni efekt z modulacją jest mniej wiarygodny od efektu z replace (dużo ciemnych obszarów i brak światła przechodzącego przez lód), ale tak przygotowany materiał jest dobry jako jeden ze składników ostatecznej wersji materiału (6.2):

$$I = (c_a + c_d(N \cdot L)) \cdot t_{diff}, \tag{6.2}$$

gdzie: t_{diff} – kolor odczytany z fotograficznej tekstury „diffuse" zawierającej obraz lodu.

6.2 Odbicia zwierciadlane i rozbłyski

Jeśli lód topnieje, to jego powierzchnia staje się gładka i zaczyna odbijać światło w sposób zwierciadlany. Jeżeli dodatkowo jest przezroczysty, to można zaobserwować również rozbłyski na szczelinach w jego wnętrzu. Nie są one uwzględniane ze względu na brak modelu wnętrza. Rozbłyski światła i odbicia elementów otoczenia na powierzchni bryły wykonane zostały za pomocą odpowiednio składnika specular modelu oświetlenia Phonga oraz mapowania środowiska.

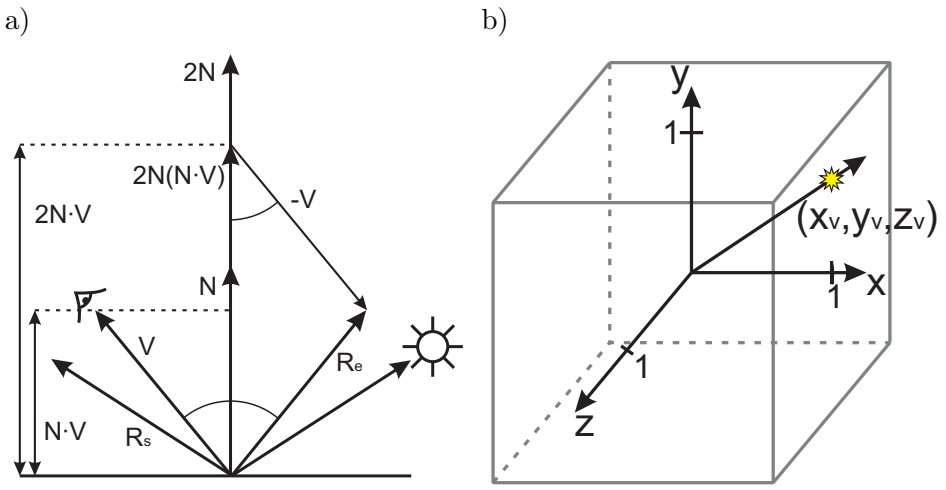
Mapowanie odbić środowiska polega na ich odczycie z tekstury sześciennej obrazującej widok otoczenia obserwowany z obiektu (tutaj z bryły lodu). Odczyt ten następuje z miejsca wskazanego przez promień (V) wychodzący z punktu obserwacji i promień (R_e) odbity od obiektu zgodnie z prawem odbicia mówiącym, że promień odbity leży w płaszczyźnie padania, a kąt odbicia jest równy kątowi padania (obliczany względem wektora normalnego). Rysunek 6.2a) pokazuje sposób obliczania wektora odbicia (6.3):

$$R_e = 2N(N \cdot V) - V \tag{6.3}$$

gdzie R_e – odbity względem wektora normalnego (N) wektor kierunkowy obserwacji (V).

Analogicznie oblicza się promień R_s, który jest odbitym promieniem (L) emitowanym przez źródło światła. Promień ten służy do generowania

plamki odbicia źródła światła na powierzchni obiektu jeśli wektor (R_s) jest dostatecznie blisko kierunku obserwacji (V).



Rysunek 6.2. Odczyt mapy środowiska: a) wektor odbicia otoczenia (R_e), b) zależnie od współrzędnych wektora R_e lub T wybrana jest jedna z sześciu ścian-tekstur i próbkowany jest jej teksel.

Tak obliczone wartości odbicia specular i odbicia otoczenia zostały dołączone do materiału wieloskładnikowego (6.4):

$$I = (c_a + c_d(N \cdot L)) \cdot t_{diff} + c_s(R_s \cdot V)^n + c_r t_{env}(R_e), \qquad (6.4)$$

gdzie:

c_s – składnik specular (w modelu Phonga iloczyn $I_{zrs} \cdot k_s$),

c_r – współczynnik udziału odbicia otoczenia w kolorze materiału,

n – wykładnik rozbłysku, określający wielkość rozbłysku światła,

R_s – wektor odbicia specular,

R_e – wektor odbicia środowiska ze wzoru 6.3,

$t_{env}(R_e)$ – kolor odczytany z tekstury środowiska z miejsca wskazanego przez wektor R_e (rys. 6.2b).

6.3 Transmisja światła w lodzie

Materiały przezroczyste przepuszczają dużą część padającego na nie światła. Zjawisko przechodzenia światła przez granicę dwóch ośrodków nazywane jest załamaniem lub refrakcją światła. Dla dielektryków takich, jak powietrze, szkło, woda, czy lód współczynnik refrakcji jest stosunkiem prędkości światła w próżni do prędkości światła w danym materiale – v (6.5):

$$\eta = \frac{c}{v}. \tag{6.5}$$

Współczynnik ten zależy od temperatury i długości fali światła i determinuje, zgodnie z prawem Snella, jak zależy kąt załamania (Θ_t) od kąta padania (Θ_i) światła na granicę dwóch ośrodków (6.6):

$$\eta_i \sin \Theta_i = \eta_t \sin \Theta_t, \tag{6.6}$$

gdzie:

η_i – współczynnik refrakcji dla ośrodka, z którego światło wychodzi,

η_t – współczynnik refrakcji dla ośrodka, do którego światło wchodzi.

Materiał załamujący światło uzyskać można analogicznie do materiału zwierciadlanego – próbkując mapę środowiska. Wektor załamania (T), służący do określenia miejsca jej odczytu oblicza się następująco [55] (wzór 6.7, rys. 6.3):

$$T = \eta V - N\left(\eta(N \cdot V) + \sqrt{1 - \eta^2(1 - (N \cdot V)^2)}\right), \tag{6.7}$$

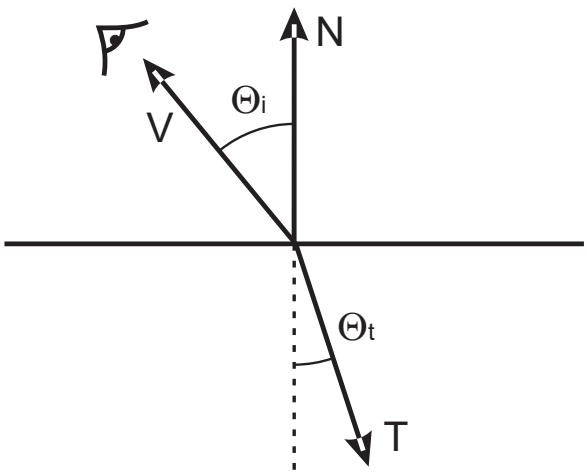
gdzie:

$\eta = \frac{\eta_i}{\eta_t}$ – względny współczynnik załamania na granicy danych ośrodków,

N – normalna,

V – wektor kierunkowy obserwacji.

W opisanej metodzie uwzględnione zostało jednokrotne załamanie światła, tylko na powierzchni obiektu zwróconej przodem do punktu obserwacji, ponieważ wielokrotne załamanie wymaga renderingu wieloprzejściowego [102] lub wstępnych obliczeń, gdy zmienia się geometria obiektu [40, 26].



Rysunek 6.3. Odczyt mapy środowiska: wektor załamania światła (T).

Współczynnik załamania światła zależy od długości fali, więc w obiekcie zachodzi zjawisko rozszczepienia światła (dyspersja chromatyczna). Jeżeli względny współczynnik załamania $\eta > 1$, to dla fal krótszych kąt załamania (Θ_t) jest mniejszy, a współczynnik załamania (η) jest większy. Na przykład lód w temperaturze $-7°C$ ma współczynnik załamania $\eta_B = 1,3169$ dla światła niebieskiego ($\lambda_B = 431nm$), $\eta_G = 1,3119$ dla zielonego ($\lambda_G = 527nm$), $\eta_R = 1,3079$ dla światła czerwonego ($\lambda_R = 656nm$). Aby w wizualizacji uwzględnić dyspersję chromatyczną należy trzykrotnie próbkować teksturę środowiska (t_{env}) wektorami $T_R(n_R), T_G(n_G), T_B(n_B)$ i użyć pojedynczej składowej – odpowiednio R, G i B z kolejnych odczytów (6.8):

$$t_{env} = (t_{env.R}, t_{env.G}, t_{env.B}), \tag{6.8}$$

gdzie:

t_{env} – kolor odczytany z tekstury środowiska z miejsca wskazanego przez trzy wektory T_R, T_G, T_B uwzględniające dyspersję chromatyczną,

$t_{env.R}$ – składowa czerwona tekstury środowiska odczytana z miejsca wskazanego przez wektor T_R obliczony na podstawie współczynnika załamania światła czerwonego η_R; analogicznie jest dla zielonego i niebieskiego.

Materiał przezroczysty transmituje tylko część światła. Pozostałą odbija lub absorbuje. Żeby obliczyć udziały światła odbitego i transmitowanego w dielektryku należy posłużyć się równaniem Fresnela [99] (6.9):

$$F(\Theta_i) = \frac{(g-c)^2}{2(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right), \tag{6.9}$$

gdzie: $c = \cos(\Theta_i), g = \sqrt{(\frac{\eta_t}{\eta_i})^2 + c^2 - 1}$.

Obliczanie równania Fresnela jest czasochłonne, dlatego zazwyczaj obliczana jest aproksymacja Schlicka dla dielektryków (6.10):

$$F(\Theta_i) = F(0^\circ) + (1 - F(0^\circ))(1 - \cos\Theta_i)^5, \tag{6.10}$$

gdzie $F(0^\circ) = (\frac{n_t - n_i}{n_t + n_i})^2$ jest obliczone z równania (6.9).

Równanie (6.10) dobrze aproksymuje współczynnik Fresnela dla dielektryków w zakresie współczynnika załamania $\eta \in (1,4; 2,2)$.

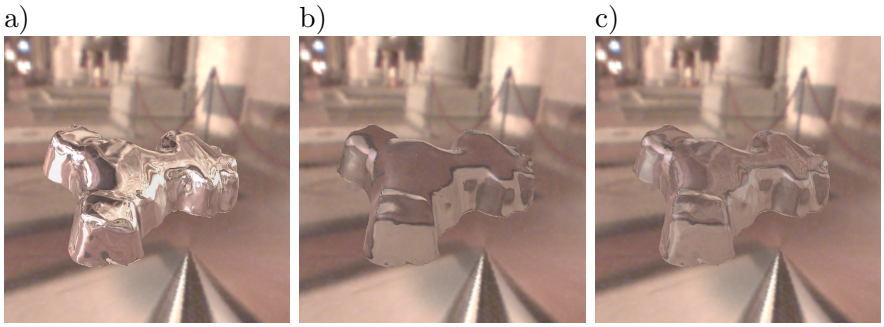
Równanie oświetlenia dla materiału wieloskładnikowego uwzględniającego wszystkie omówione cechy lodu i zjawiska ma następującą postać (6.11):

$$I = (c_a + c_d(N \cdot L)) \cdot t_{diff} + c_s(R_s \cdot L)^n + c_r F t_{env}(R_e) + c_t(1-F)t_{env}(T), \tag{6.11}$$

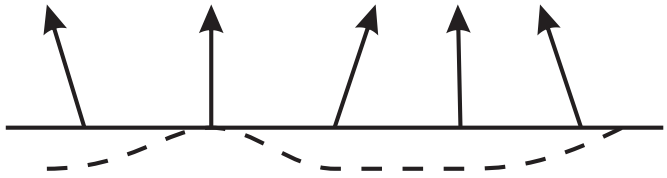
gdzie c_t – współczynnik udziału transmisji światła w kolorze materiału, a pozostałe oznaczenia, jak w równaniach (6.4), (6.8) i (6.10). Rysunek 6.4 przedstawia łączenie materiału zwierciadlanego i przezroczystego za pomocą współczynnika Fresnela.

6.4 Nierówności powierzchni

Jeśli modelowana bryła lodu ma mieć nierówną, chropowatą powierzchnię, to nie należy modelować nierówności geometrycznie, gdyż zwiększyłoby to w sposób znaczący liczbę trójkątów w siatce obiektu. W zamian można zrealizować za pomocą lokalnego zaburzenia kierunku wektora normalnego (N) używanego do obliczeń składnikach materiału (rys. 6.5).



Rysunek 6.4. Współczynnik Fresnela: a) materiał zwierciadlany, b) materiał przezroczysty załamujący światło, c) połączenie odbicia i załamania.



Rysunek 6.5. Modyfikacja normalnych powierzchni powoduje wrażenie jej nierówności (linia przerywana).

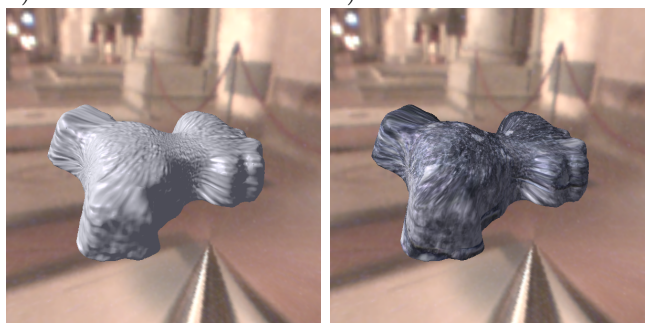
Techniką umożliwiającą wprowadzanie zaburzenia kierunku normalnych jest mapowanie normalnych [43], które wprowadza do przetwarzania dodatkową teksturę. Jest ona trzykanałowa, a w kolejnych kanałach przechowuje współrzędne (x, y, z) wektorów normalnych jako wartości (r, g, b) tekseli. Wartości te używają zakresu kolorów $< 0, 1 >$, dlatego zostają przeskalowane przed użyciem do zakresu $< -1, 1 >$, który jest potrzebny dla reprezentacji współrzędnych znormalizowanych wektorów normalnych (6.12):

$$N.xyz = 2(t_{normalmap}.rgb - \frac{1}{2}), \tag{6.12}$$

W każdym punkcie powierzchni obiektu taka normalna zastępuje oryginalną normalną powierzchni obiektu dając pożądane zaburzenie. Takie zmapowanie nierówności powoduje zmianę kierunku odbijania, albo załamywania światła w równaniu (6.11) ponieważ kierunki te liczone są względem

normalnej. Rysunek 6.6 pokazuje nałożenie tego efektu na materiał modelu oświetlenia Phonga oraz na ten sam materiał z nałożoną, modulowaną teksturą.

a) b)

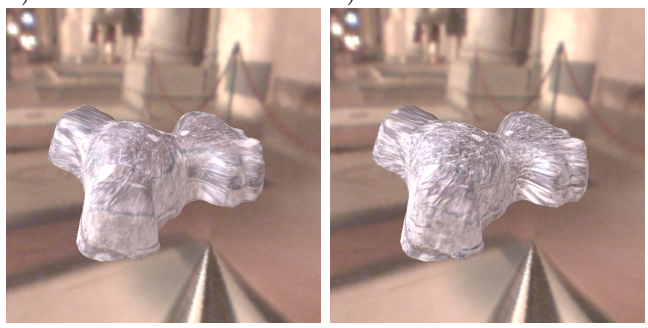


Rysunek 6.6. Obiekt oświetlony przy pomocy modelu oświetlenia Phonga: a) z mapowaniem nierówności; b) z mapowaniem nierówności oraz modulacją tekstury lodu (t_{diff}).

6.5 Podsumowanie

Rysunek 6.7 przedstawia dwa warianty końcowego wyglądu materiału wieloskładnikowego na obiekcie – bryle lodu.

a) b)



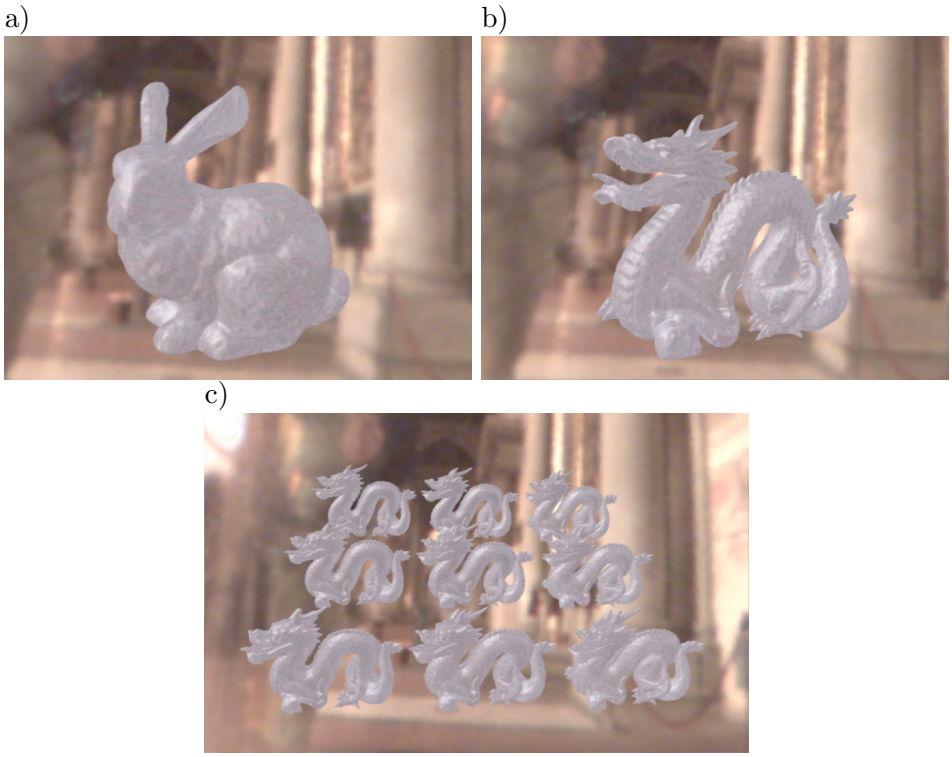
Rysunek 6.7. Materiał wieloskładnikowy: a) lód o gładkiej powierzchni; b) lód o nierównej powierzchni.

Rysunek 6.8. Materiał wieloskładnikowy na sublimującym obiekcie.

Rysunek 6.8 przedstawia zachowanie się materiału wieloskładnikowego w różnych cyklach sublimacji obiektu, natomiast rysunek 6.9 przedstawia porównanie wyglądu sfotografowanej i wyrenderowanej bryłki lodu.

Rysunek 6.9. Sfotografowana (po lewej) oraz wyrenderowana (po prawej) bryłka lodu.

Materiał wieloskładnikowy wymaga obliczeń poszczególnych składników, powtarzanych dla każdego piksela powstającego obrazu obiektu, wykonywanych na GPU. Należy zmierzyć, czy obliczenia związane z tak skomplikowanym materiałem nie wpłyną na szybkość renderingu. W tym celu materiał wieloskładnikowy został nałożony na obiekty o złożoności dużo większej niż potrzebna w zastosowaniach wymagających renderingu czasu

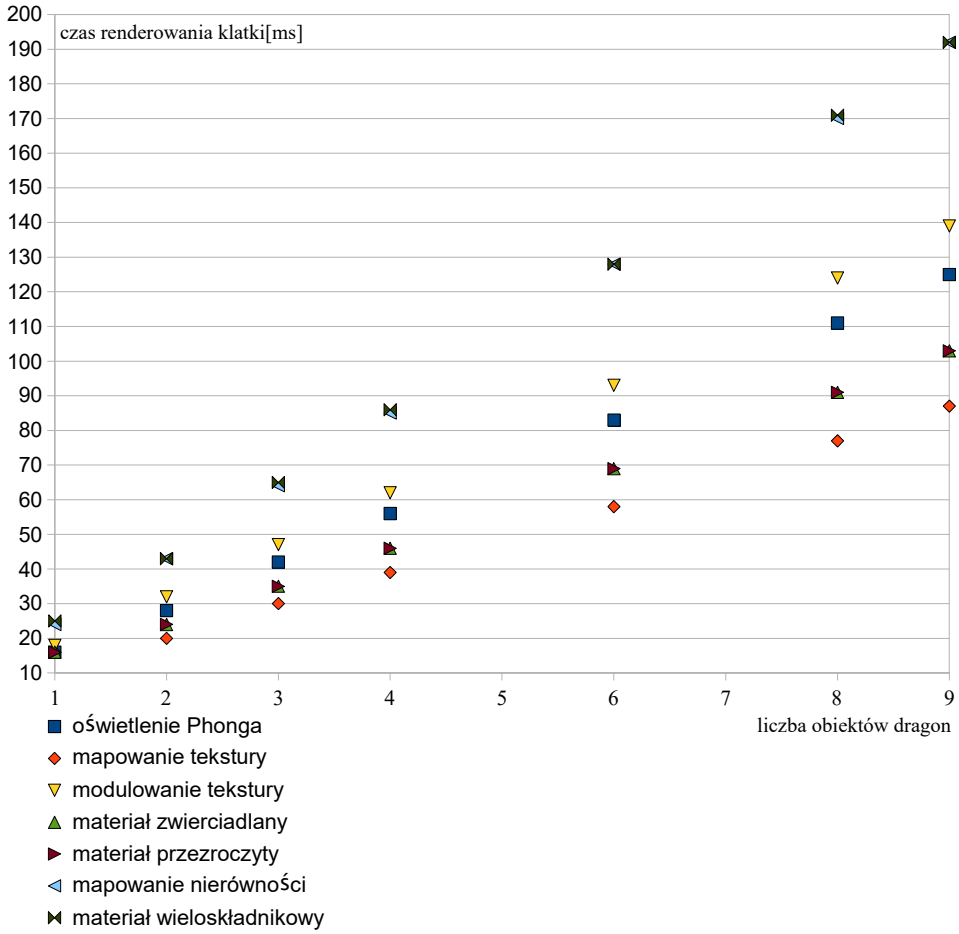


Rysunek 6.10. Obiekty testowe: a) bunny (około 69k trójkątów), b) dragon (około 871k trójkątów), c) 9 obiektów dragon (około 7842k trójkątów)

rzeczywistego. O ile do wizualizacji topnienia w niniejszej monografii wystarczały obiekty złożone z około 10000 trójkątów, o tyle testy materiału wieloskładnikowego odbyły się dla obiektów zawierających 69000 trójkątów i więcej. Testowane były obiekty bunny, dragon oraz grupy obiektów dragon (rys. 6.10).

Tabela 6.1 oraz wykres 6.11 pokazują osiągnięte czasy renderingów obiektów i grup obiektów o bardzo dużej, jak na rendering czasu rzeczywistego liczbie trójkątów. Przedstawione czasy zostały osiągnięte na karcie graficznej nVidia GeForce 6800GT. Jak widać, nawet obiekty o 100-krotnie większej liczbie elementów, niż potrzebna (milion trójkątów) do wizualizacji sublimacji, mogą być wyrenderowane z szybkościami większymi niż 40 ramek na sekundę. Dlatego też zaproponowany materiał wieloskładnikowy oprócz

możliwie wiarygodnego odwzorowania wyglądu obiektu umożliwia również stosowanie go w aplikacjach renderingu czasu rzeczywistego.



Rysunek 6.11. Wykres zależności czasu trwania renderingu ramki od liczby obiektów dragon (zawierających 871414 trójkątów każdy).

Tabela 6.1. Zestawienie czasu (w ms) renderingu pojedynczej ramki zawierającej obiekty o dużej liczbie trójkątów z materiału wieloskładnikowego.

model	bunny	dragon						
liczba obiektów	1	1	2	3	4	6	8	9
liczba trójkątów	69451	871414	1742828	2614242	3485656	5228484	6971312	7842726
oświetlenie Phonga	16	16	28	42	56	83	111	125
mapowanie tekstury	16	16	20	30	39	58	77	87
modulacja tekstury materiał	16	18	32	47	62	93	124	139
zwierciadlany materiał	16	16	24	35	46	69	91	103
przezroczysty mapowanie	16	16	24	35	46	69	91	103
nierówności materiał	16	24	43	64	85	128	170	192
wieloskładnikowy	16	25	43	65	86	128	171	192

7

Wizualizacja topnienia

Zjawisko topnienia różni się od prezentowanego wcześniej zjawiska sublimacji z punktu widzenia wizualizacji. O ile w zjawisku sublimacji faza stała zmienia się w gazową, której nie trzeba wizualizować, bo jest niewidoczna, o tyle przy topnieniu należy uwzględnić wizualizację powstałej w jego wyniku cieczy. Ciecz może mieć różne właściwości wizualne oraz podobnie, jak powierzchnia międzyfazowa podczas topnienia jest w ruchu, co należy uwzględnić w tym przypadku.

7.1 Zjawisko topnienia a zjawisko sublimacji

Wizualizacja zjawiska topnienia składa się z:

- wizualizacji sublimującej bryły lodu opisanej w rozdziale poprzednim,
- wizualizacji wody powstającej ze stopionego lodu, która jest autorskim rozwiązaniem.

Masa fazy ciekłej zwiększa się kosztem masy fazy stałej, co powinno być uwzględnione w wizualizacji. Wraz z kurczeniem się topniejącego obiektu ciecz powinna zajmować coraz większy obszar.

Przyjęte zostały następujące **założenia**, zgodnie z którymi przeprowadzana jest wizualizacja cieczy:

Zał. 1. Wizualizowana ciecz rozlewa się na płaskim obiekcie (podłożu), na którym umieszczony jest topniejący obiekt. Nie jest uwzględnione spływanie cieczy po obiekcie.

Zał. 2. Kształt plamy cieczy zależy od kształtu topniejącego obiektu oraz od cech fizycznych podłoża.

Zał. 3. Ciecz rozlewa się tak długo, jak długo trwa topnienie.

Jeżeli do przedstawionej wcześniej wizualizacji procesu sublimacji zostanie dołączona wizualizacja cieczy, to da to wizualny efekt topnienia obiektu.

7.2 Wizualizacja fazy ciekłej

Fotografia 7.1 przedstawia wygląd rozlanej wody na płaskiej powierzchni.

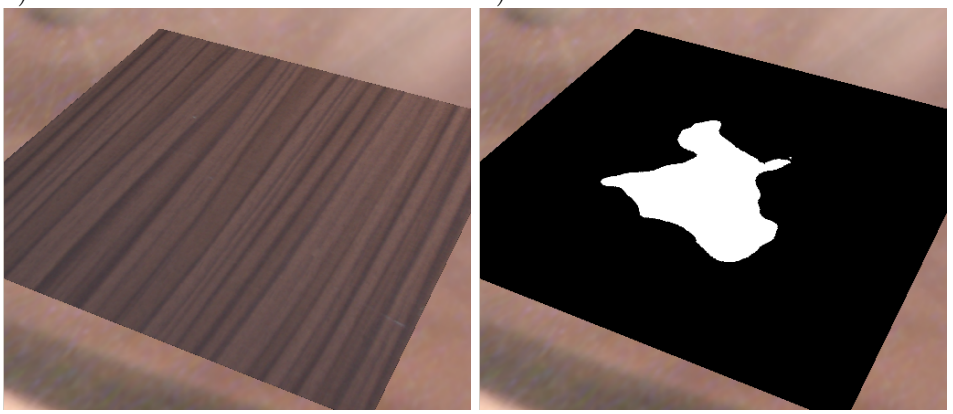
Rysunek 7.1. Fotografia rozlanej wody.

Ze względu na przyjęte założenie **1.** można było zrezygnować z geometrycznego modelowania plamy cieczy na rzecz innych technik wizualizacji.

Topniejący lód zamienia się w wodę, której wygląd, gdy jest rozlana na płaskiej powierzchni, posiada poniższe cechy:

— kolory powierzchni przykrytej wodą wydają się ciemniejsze (wilgoć) lub bardziej skontrastowane ze sobą,
— powierzchnia wody odbija środowisko,
— plama jest wypukłością – tworzy warstwę o pewnej grubości.

a) b)



Rysunek 7.2. Wizualizacja cieczy: a) oryginalny widok podłoża; b) tekstura maski – biały kolor identyfikuje położenie plamy cieczy.

Przyjęto, że cechy wyglądu podłoża modelowane są przy pomocy modelu oświetlenia Phonga (2.13) i modulacji tekstury koloru (rys. 7.2a). Położenie cieczy określa jednokanałowa tekstura maski, której kolor koduje informację o obecności lub nieobecności cieczy w danym miejscu powierzchni (rys. 7.2b). W zadaniu wizualizacji maska może być narysowana lub wygenerowana dowolnym algorytmem. Następny podrozdział (7.3) opisuje, jak wygenerować maskę dla kolejnych cykli obliczeń, aby powstała animacja rozlewającej się po podłożu cieczy.

Wszystkie obliczenia związane z wyglądem cieczy realizowane są na GPU w programie cieniowania fragmentów uruchamianym dla podłoża.

Założenie, że ciecz rozlewa się na płaskim podłożu pozwala na uproszczenie obliczeń. Współrzędne dwuwymiarowe (x, y) punktu podłoża odpowiadają dwuwymiarowym współrzędnym teksturowania (s, t) na jego powierzchni. Układ współrzędnych przestrzeni tekstury odpowiada lokalnemu układowi współrzędnych podłoża.

7.2.1 Przezroczystość wody

Barwa podłoża obliczana jest z modelu Phonga i modulowana teksturą koloru (rys. 7.2a). Podniesienie składowych RGB do potęgi o wykładniku większym od 1 powoduje przyciemnienie koloru przy jednoczesnym zachowaniu wyniku w pożądanym zakresie. W ten sposób jest zrealizowana wizualizacja pierwszej cechy wody, czyli przezroczystości w tych miejscach podłoża (s, t), na których ciecz jest obecna (7.1):

$$I = ((c_a + c_d(N \cdot L)) \cdot t_{diff} + c_s(R_s \cdot V)^n)^{1+a \cdot t_{mask}}\,, \tag{7.1}$$

gdzie:

I – finalny kolor fragmentu w punkcie (s, t),

c_a – składnik ambient (w modelu Phonga iloczyn $I_{zra} \cdot k_a$),

c_d – składnik diffuse (w modelu Phonga iloczyn $I_{zrd} \cdot k_d$),

c_s – składnik specular (w modelu Phonga iloczyn $I_{zrs} \cdot k_s$),

N – normalna,

L – wektor kierunkowy światła.

R_s – odbity promień (L),

$t_{mask} \in \{0, 1\}$ – wartość odczytana z maski obecności cieczy w punkcie podłoża odpowiadającym rysowanego fragmentowi,

t_{diff} – kolor odczytany z tekstury podłoża,

a – współczynnik zmiany koloru (przyjęto $\frac{6}{10}$).

Zmieniany jest tylko kolor fragmentów, które leżą w obszarze „zalanym wodą", ponieważ dla pozostałych fragmentów współczynnik potęgi jest równy jeden. Efekt wizualny przedstawiony jest na rysunku 7.3a).

7.2.2 Odbicia otoczenia na powierzchni wody

Podłoże może być matowe lub błyszczące. Woda ma powierzchnię odbijającą otoczenie. Efekt ten uzyskuje się w grafice komputerowej przez mapowanie środowiska. Analogicznie, jak w rozdziale 6 użyto sześciennej mapy środowiska. W przedstawionej metodzie jest to nie zmieniająca się mapa środowiska. Gdyby zmieniło się położenie wizualizowanego obiektu (lodu) lub obiektów z jego otoczenia, mapa środowiska również powinna ulec zmianie. Należałoby wyrenderować do niej nowy widok otoczenia obiektu.

Kolor odbicia otoczenia zostaje odczytany z tekstury środowiska t_{env} z miejsca wskazanego przez wektor odbicia zwierciadlanego R_e (6.3) i dołączony do równania 7.1 uwzględniającego przezroczystość wody 7.2:

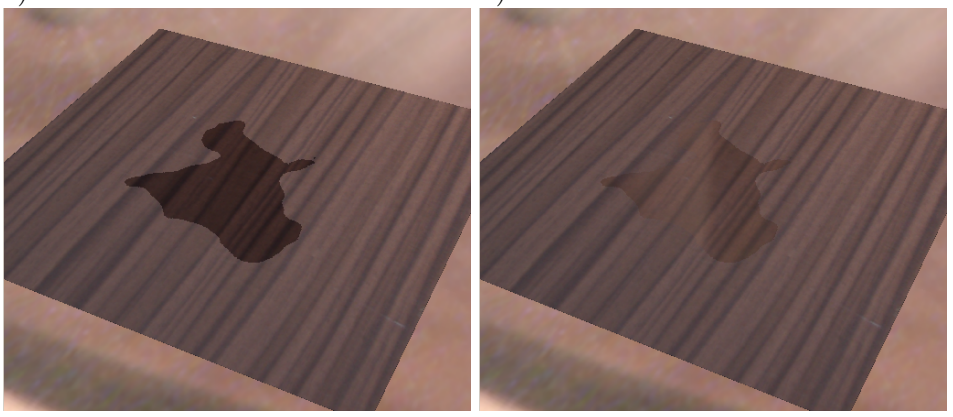
$$I = ((c_a + c_d(N \cdot L)) \cdot t_{diff} + c_s(R_s \cdot V)^n)^{1+a \cdot t_{mask}} + c_r t_{env}(R_e) \cdot t_{mask}. \tag{7.2}$$

gdzie:

c_r – współczynnik odbić środowiska (przyjęto $\frac{2}{10}$), pozostałe wielkości, jak w (7.1).

Efekt wizualny przedstawia rysunek 7.3b).

a) b)



Rysunek 7.3. Zmiana koloru oryginalnej tekstury: a) wizualizacja „przezroczystej" plamy; b) dołączony składnik odbić środowiska.

7.2.3 Wypukłość wody

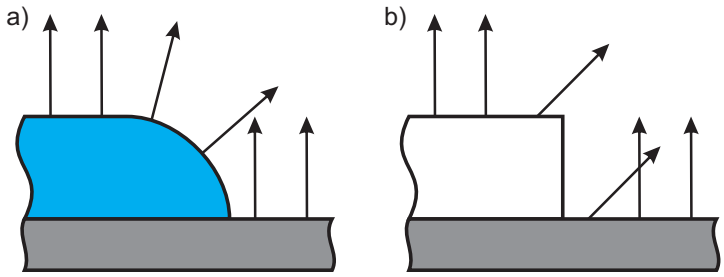
Wypukłość warstwy wody nie musi być modelowana geometrycznie. Zamiast tego zastosowano lokalną modyfikację wektorów normalnych. Modyfikacja ta powinna wystąpić na brzegach plamy. Wektory normalne należy skierować tak, aby plama postrzegana była jako wypukła (rys. 7.5a). Zmodyfikowana normalna (N^*) w danym punkcie powierzchni podłoża (7.3):

$$N^* = [c_s(t^{left}_{mask} - t^{rigth}_{mask}), c_t(t^{up}_{mask} - t^{down}_{mask}), 1]^\circ, \tag{7.3}$$

gdzie:

(c_s, c_t) – współczynniki regulujące pozorną wysokość powierzchni cieczy nad powierzchnię podłoża (przyjęto $\frac{7}{10}$),
t^{left}_{mask} – wartość teksela maski położonego w mapie na lewo $(s-1, t)$ od obliczanego punktu podłoża (s, t),
t^{right}_{mask} – wartość teksela maski położonego w mapie na prawo $(s+1, t)$,
t^{up}_{mask} – wartość teksela maski położonego w mapie poniżej $(s, t-1)$,
t^{down}_{mask} – wartość teksela maski położonego w mapie powyżej $(s, t+1)$.

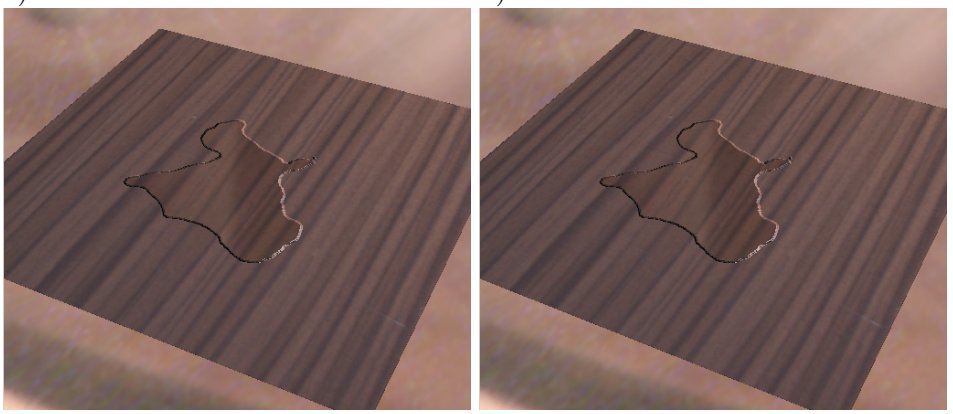
Należy tu zwrócić uwagę na uproszczenie wynikające z założenia **1.**, które powoduje, że wektor $N = [0, 0, 1]$ jest wektorem normalnym prostopadłym do płaskiego podłoża.



Rysunek 7.4. Wektory normalnych: a) obliczone według cech geometrycznych obiektu plamy; b) obliczone z maski i równania (7.3).

Rysunek 7.4 przedstawia modyfikację wektora normalnego. Niezmodyfikowane wektory normalne pozostają prostopadłe do podłoża. Wektory

a) b)



Rysunek 7.5. Wrażenie grubości warstwy cieczy: a) efekt modyfikacji wektora normalnego na brzegach plamy, aby stworzyć wrażenie ich wypukłości, b) odczyt tekstury podłoża z przesunięciem, aby stworzyć wrażenie załamania światła w warstewce wody.

na brzegu powinny być pochylone. Wartości t_{mask} dla fragmentów poza brzegami plamy są albo wszystkie zerami, albo wszystkie jedynkami. Zatem ich różnice wynoszą zero i wektor normalny nie zmienia się. Natomiast dla fragmentów na brzegu plamy wartości maski różnią się, co powoduje pochylenie wektora normalnego w takim kierunku, w jakim byłby pochylony, gdyby plama wody modelowana była geometrycznie.

Warstewka wody mając pewną grubość załamuje światło (rys. 7.5b). Dla widza istotne jest to, że geometryczna linia (oś) obserwacji (bez załamania) zostanie zastąpiona przez optyczną linię obserwacji (z załamaniem w wodzie). Natomiast stopień tego załamania może być obliczany w sposób uproszczony. Dobrym przybliżeniem jest przeskalowanie rzutu wektora obserwacji (V) na podłoże w celu uzyskania wektora załamania – T_{xy} (7.4):

$$T_{xy} = -d \cdot V_{xy} \cdot t_{mask} \tag{7.4}$$

gdzie:

T_{xy} – wektor załamania światła w wodzie zrzutowany na podłoże,

V_{xy} – rzut wektora obserwacji fragmentu na podłoże,

d – współczynnik skalujący (przyjęto $\frac{1}{10}$),

t_{mask} – maska obecności cieczy na podłożu. W obszarze podłoża nie pokrytym wodą teksel $t_{mask} = 0$, więc $T_{xy} = [0, 0]$.

Wektor T_{xy} został użyty do obliczenia przesunięcia linii obserwacji na teksturze podłoża (7.5):

$$t^*_{diff}(s,t) = t_{diff}(s - d \cdot V_x \cdot t_{mask}, t - d \cdot V_y \cdot t_{mask}). \tag{7.5}$$

Przesunięcie to zależy od kierunku obserwacji i daje wrażenie efektu załamania światła w warstewce wody (rys. 7.5b).

Równanie oświetlenia dla danego fragmentu podłoża jest analogiczne do równania (7.2). Różni się od niego próbkowaniem tekstury podłoża (t^*_{diff}) i wektorem normalnym (N^*) używanym do obliczania składnika specular oraz wektora $R^*_e = 2N^*(N^* \cdot V) - V$ (7.6):

$$I = \Big((c_a + c_d(N^* \cdot L)) \cdot t^*_{diff} + c_s(R_s \cdot V)^n\Big)^{1+a \cdot t_{mask}} + c_r t_{env}(R^*_e) \cdot t_{mask}. \tag{7.6}$$

7.3 Ruch fazy ciekłej

Ruch fazy ciekłej realizowany jest jako zmiana wielkości plamy cieczy wraz z upływem czasu. Po każdym cyklu przetwarzania geometrii w celu deformacji obiektu topionego występuje więc faza przetwarzania kształtu fazy ciekłej. W rzeczywistym zjawisku topnienie (tworzenie się fazy ciekłej) i rozlewanie się wody są jednoczesne. Przedstawiona metoda dzieli te zjawiska na dwa oddzielne kroki wykonywane po cyklu obliczeń związanych z sublimacją:

1. rozlewanie się cieczy powstałej do tej pory,

2. dodanie nowej porcji cieczy.

Aby utworzyć mechanizm rozlewania się cieczy opisany poniżej, bieżący kształt plamy cieczy zapamiętywany jest w teksturze maski, ale wartości zapamiętane w poszczególnych tekselach maski nie są wartościami boolowskimi, ale zmiennoprzecinkowymi ($\geqslant 0$). Przyjęty został pewien próg (p).

Jeżeli wartość w tekselu jest nie mniejsza niż p, to ciecz jest w tym miejscu obecna. Zatem równania (7.3), (7.2) i (7.4) powinny uwzględniać operację progowania. Użyto w tym celu funkcji `step()` o poniższej definicji (7.7):

$$\text{step}(a,x) = \begin{cases} 0, \text{gdy } x < a, \\ 1, \text{gdy } x >= a \end{cases} \tag{7.7}$$

Wtedy, po użyciu funkcji `step()` z zadanym progiem p:

— modyfikacja normalnej (7.8):

$$\begin{aligned} N^* = \; & [c_s(\ \text{step}(p, t^{left}_{mask}) - \ \text{step}(p, t^{right}_{mask})), \\ & c_t(\ \text{step}(p, t^{up}_{mask}) - \ \text{step}(p, t^{down}_{mask})), \\ & 1]^\circ, \end{aligned} \tag{7.8}$$

— wektor załamania (7.9):

$$T_{xy} = -d \cdot V_{xy} \cdot \ \text{step}(p, t_{mask}), \tag{7.9}$$

— próbkowanie podłoża zgodnie z optyczną linią obserwacji – załamaną w wodzie (7.10):

$$t^*_{diff}(s,t) = t_{diff}(s - d \cdot V_x \cdot \ \text{step}(p, t_{mask}), t - d \cdot V_y \cdot \ \text{step}(p, t_{mask})), \tag{7.10}$$

— kolor końcowy (7.11):

$$\begin{aligned} I = \; & \Big((c_a + c_d(N^* \cdot L)) \cdot t^*_{diff} + c_s(R_s \cdot V)^n\Big)^{1+a \cdot \ \text{step}(p, t_{mask})} \\ & + c_r t_{env}(R^*_e) \cdot \ \text{step}(p, t_{mask}). \end{aligned} \tag{7.11}$$

Mając tak zdefiniowaną maskę i jej użycie w wizualizacji można rozlanie dotychczasowej i dodawanie nowej porcji cieczy zrealizować jako operacje na niej.

7.3.1 Rozlewanie się cieczy

Rozlewanie się cieczy jest równoznaczne ze zwiększaniem wartości odpowiednich tekseli maski powyżej progu p. Aby zrealizować powiększanie się plamy cieczy powinna być zwiększana wartość zapamiętana w tekselach sąsiadujących z tekselami należącymi już do plamy. Jednocześnie powinna być

zmniejszana średnia wartość tekseli zajętych przez ciecz, aby nie powiększała się ona aż do zajęcia całej tekstury. Takie warunki spełnia filtr rozmywający. Filtr użyty w przedstawionej metodzie ma rozmiaru 5×5 i dany jest maską (7.12):

$$f = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix} /16 \tag{7.12}$$

Taka maska filtru została wybrana ze względu na jej wielkość (5×5) i wynikającą z niej możliwość optymalizacji szybkości operacji filtrowania w programie cieniowania. Przychodzi tu z pomocą mechanizm dwuliniowej interpolacji tekstury opisany w podrozdziale 4.2.2. Jedynki w masce filtru są ułożone sąsiadująco w blokach (2×2), wystarczy więc próbkować miejsca w środku tych bloków, aby uzyskać wartości średnie. Na przykład aby spróbkować średnią lewego-górnego bloku tekseli odpowiadającą elementom maski filtru $f[1,1], f[1,2], f[2,1], f[2,2]$, należy odczytać wartość odpowiadającą pozycji $f[1\frac{1}{2}, 1\frac{1}{2}]$ pamiętając, że tekstury adresowane są współrzędnymi zmiennoprzecinkowymi. Tekstura adresowana jest współrzędnymi z zakresu $< 0, 1 >$, więc dla położenia (s,t), dla którego obliczana jest wartość wynikowa filtru, współrzędne pozycji próbkowanego lewego-górnego bloku wynoszą (7.13):

$$C = \left(s - \frac{1\frac{1}{2}}{w}, t - \frac{1\frac{1}{2}}{h} \right), \tag{7.13}$$

gdzie w, h – odpowiednio szerokość oraz wysokość tekstury wyrażone w tekselach.

Po odczytaniu czterech średnich obliczona jest średnia z tych średnich. W ten sposób tekstura nie musi być próbkowana 16 razy – wystarczą 4 próbki, co również cztereokrotnie skraca czas wykonania fragmentu programu cieniowania odpowiedzialnego za próbkowanie tekstury. Oczywiście nie jest to uniwersalna optymalizacja i przypadki jej użycia są ściśle ograniczone do niewielkiej liczby analogicznie konstruowanych masek.

Fragment programu cieniowania, który realizuje ten filtr wygląda więc następująco:

```
// inputMaskSize - rozmiar tekstury maski w tekselach
// (wysokość=szerokość)
// inputMask - wejściowa tekstura maski
// texCoords - współrzędne teksturowania dla przetwarzanego teksela

fragment_shader()
// przesunięcie o półtora piksela w układzie współrzędnych tekstury:
 float s = 1.5f / inputMaskSize;

// próbkuj zawartość tekstury wejściowej w miejscu
// (texCoords.x - s, texCoords.y - s):
 mask  = tex2D( inputMask, texCoords + float2( -s, -s ) );
// analogicznie pozostałe bloki:
 mask += tex2D( inputMask, texCoords + float2( -s,  s ) );
 mask += tex2D( inputMask, texCoords + float2(  s, -s ) );
 mask += tex2D( inputMask, texCoords + float2(  s,  s ) );
 mask /= 4.0f;
 return mask;
```

Należy zwrócić uwagę, że na brzegach tekstury próbkowane wartości uzupełniane są zerami. Powoduje to obniżanie średniej wartości pikseli w każdym cyklu przetwarzania, więc rozlewanie kończy się, gdy wartości w teksturze maski nie osiągają progu p.

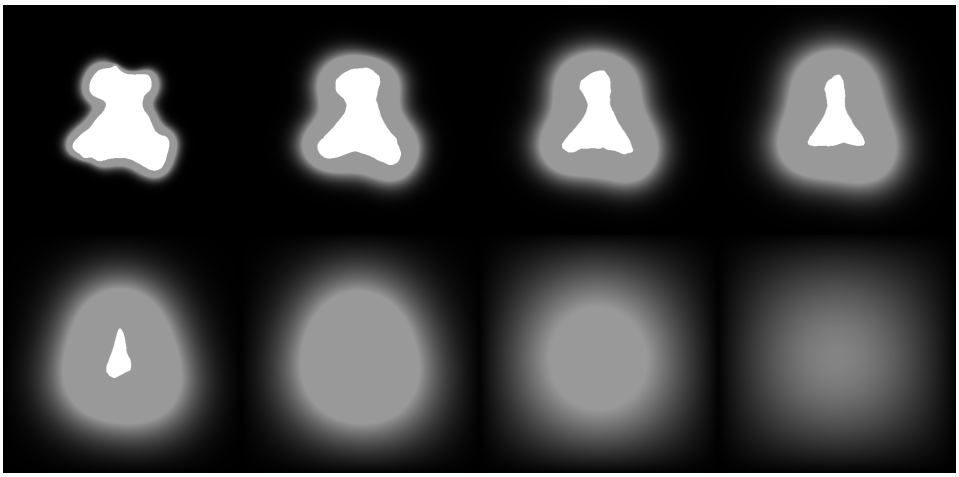
7.3.2 Przyrost ilości cieczy

Przyrost ilości cieczy oznacza zwiększanie wartości tekseli umiejscowionych bezpośrednio pod topniejącym obiektem. Można to rozumieć jako dodawanie nowych porcji cieczy w kolejnych iteracjach. Powoduje to zwiększenie średniej wartości tekseli w teksturze maski. Aby uzyskać wrażenie topnienia obiektu, kształt obszaru tekseli, do których są dodawane wartości jest kształtem konturu topniejącego obiektu. Nie jest brany pod uwagę cały kontur,

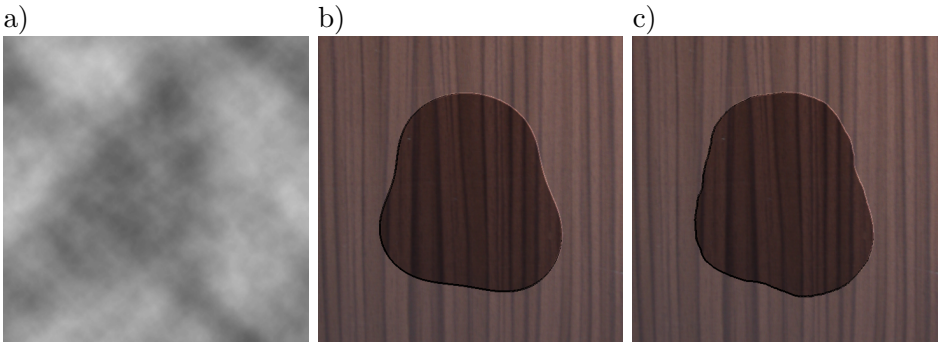
ale jego część najbliższa podłożu. Abu uzyskać obraz tej części wykonywany jest dodatkowy przebieg renderingu:

— ustawiana jest macierz rzutowania prostokątnego o wymiarach widoku takich jak wymiary podłoża oraz z bardzo małą wartością dalszej płaszczyzny obcinania – około 10% grubości topionego obiektu liczonej wzdłuż kierunku prostopadłego do podłoża, aby renderować tylko bliską podłożu część konturu obiektu topniejącego,
— punkt widzenia jest ustawiany pod topionym obiektem,
— wykonywany jest rendering „bliskiego" konturu obiektu topniejącego z wyłączonym oświetleniem,
— rendering następuje do celu renderingu (RT) z podłączoną teksturą maski, która nie jest czyszczona przed renderingiem, dzięki temu zmieniają się (zwiększają) tylko wartości tekseli stanowiące „bliski" kontur obiektu.

Po renderingu zawartość tekstury jest złożeniem rozmytego stanu poprzedniego i aktualnego kształtu dołu obiektu topionego. Rysunek 7.6 pokazuje wynik kilku cykli działania opisanego mechanizmu w teksturze maski.



Rysunek 7.6. Tekstura maski po 100, 400, 700, 1000, 2000, 3000, 6000, 10000 cyklach. Na szaro wyrenderowany jest rozmyty stan poprzedni a na biało aktualnie wyrenderowany kształt dołu obiektu topionego.



Rysunek 7.7. Wpływ cech podłoża na rozlewanie się cieczy: a) tekstura cech powierzchni podłoża, b) plama cieczy progowana stałą wartością, c) plama cieczy progowana teksturą cech powierzchni.

Obiekt topniejący jest renderowany do tekstury maski tak długo, aż stopnieje. Wtedy przestaje być renderowany i w drugim kroku nie są już dodawane nowe wartości do tekstury. Od tej chwili jest ona już tylko rozmywana aż do zaniknięcia. Stanowi to wypełnienie założenia **3.**

Zawartość maski zależy od kształtu konturu topniejącego obiektu, więc spełnione jest również częściowo założenie **2.** Aby spełnić je całkowicie, czyli aby kształt plamy cieczy zależał dodatkowo od cech fizycznych podłoża, wprowadzona została do wizualizacji jeszcze jedna dana. Dzięki temu próg p nie jest stałą, lecz funkcją położenia punktu na podłożu i jest odczytywany z odpowiedniego miejsca tekstury cech powierzchni podłoża (rys. 7.7a). Jeśli w różnych punktach podłoża wartość progu p jest różna, to uzyskuje się efekt większej „łatwości" rozchodzenia się cieczy w niektórych miejscach, natomiast w innych mniejszej. Dzięki temu zabiegowi kształt plamy cieczy mimo, iż jest przede wszystkim zależny od konturu topniejącego obiektu, to jest bardziej nieregularny.

Teksturę cech powierzchni podłoża (t_p) reprezentującą funkcję p można traktować jako teksturę nierówności (miejsca położone niżej zalane są szybciej, niż miejsca położone wyżej) albo skłonności do łączenia się z lub odpychania cieczy (w przypadku wody hydrofilowość i hydrofobowość). Wprowadzenie tej tekstury zwiększa realizm wizualizacji i spełnia **2.** (rys. 7.7bc) oraz powoduje następujące zmiany w:

— równaniu modyfikacji normalnej (7.14):

$$\begin{aligned} N^* = \quad & [c_s(\ \mathrm{step}(t_p^{left}, t_{mask}^{left}) - \ \mathrm{step}(t_p^{right}, t_{mask}^{right})), \\ & c_t(\ \mathrm{step}(t_p^{up}, t_{mask}^{up}) - \ \mathrm{step}(t_p^{down}, t_{mask}^{down})), \\ & 1]^\circ, \end{aligned} \tag{7.14}$$

gdzie $t_p^{left}, t_p^{right}, t_p^{up}, t_p^{down}$ – tekstura cech powierzchni podłoża próbkowana w miejscach analogicznych do miejsc próbkowania tekstury maski,

— wektorze załamania (7.15):

$$T_{xy} = -d \cdot V_{xy} \cdot \ \mathrm{step}(t_p, t_{mask}), \tag{7.15}$$

— próbkowaniu podłoża załamanym wektorem obserwacji (7.16):

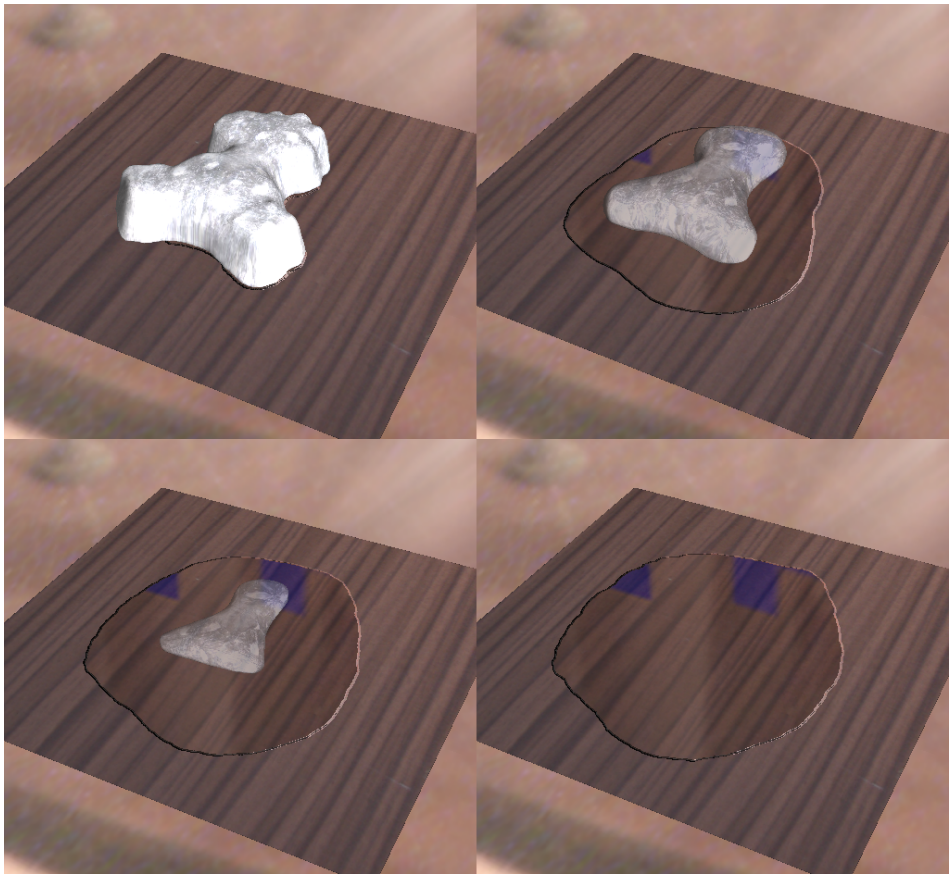
$$t^*_{diff}(s,t) = t_{diff}(s - d \cdot V_x \cdot \ \mathrm{step}(t_p, t_{mask}), t - d \cdot V_y \cdot \ \mathrm{step}(t_p, t_{mask})), \tag{7.16}$$

— równaniu oświetlenia (7.17):

$$\begin{aligned} I = \quad & \Big((c_a + c_d(N^* \cdot L)) \cdot t^*_{diff} + c_s(R_s \cdot V)^n\Big)^{1+a \cdot \ \mathrm{step}(t_p, t_{mask})} \\ & + c_r t_{env}(R_e^*) \cdot \ \mathrm{step}(t_p, t_{mask}). \end{aligned} \tag{7.17}$$

7.4 Podsumowanie

Niniejszy rozdział przedstawił sposób wizualizacji cieczy przy pomocy materiału nakładanego na podłoże oraz jej animacji przy pomocy badania konturów dolnej części obiektu. Pozwoliło to na dodanie do wizualizacji sublimacji elementów, które tworzą z niej wizualizację topnienia, której przykład ukazany jest na rysunku 7.8.



Rysunek 7.8. Wizualizacja topnienia na początku oraz po 800, 1600 i 2547 cyklach.

8

Weryfikacja wyników

Topnienie i sublimacja są procesami zmieniającymi kształt powierzchni międzyfazowej, a przez to kształt i obrys obiektu. Przedstawiane mechanizmy wizualizacji powinny więc pozwalać na oddanie rzeczywistych zmian kształtów obiektów podczas tych zjawisk zachodzących w naturze. Niniejszy rozdział przedstawia opracowaną metodę weryfikacji wyników bazującej na porównywaniu kształtów.

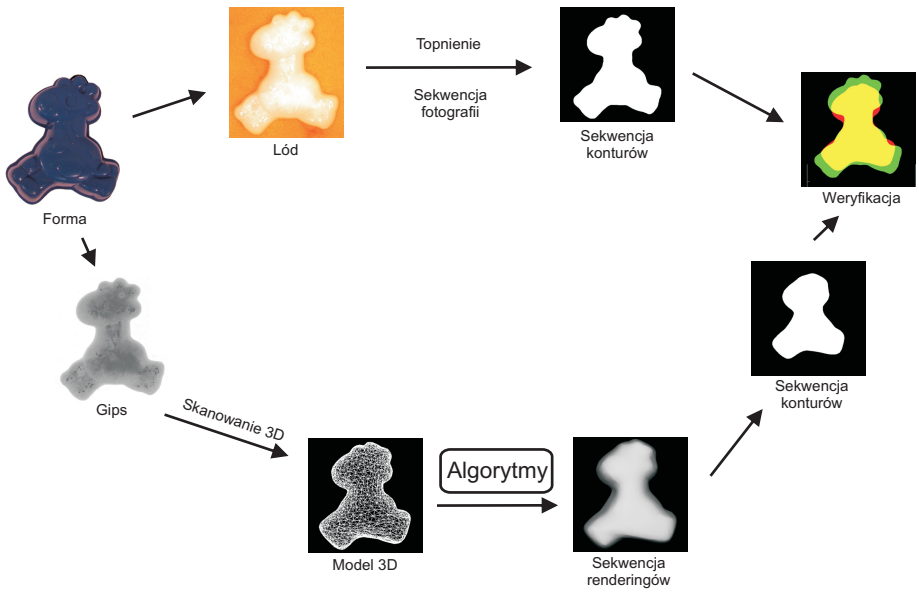
8.1 Scenariusz weryfikacji

Porównanie przebiegów zjawiska naturalnego i wizualizowanego poprzez konfrontację kształtów pociągnęło za sobą konieczność opracowania zbioru warunków przebiegu zjawiska rzeczywistego i zbioru analogicznych warunków przebiegu wizualizacji tego zjawiska za pomocą grafiki 3D [88]:

— obiekt wirtualny jest wizualizowany przy pomocy algorytmów autora, a obiekt rzeczywisty jest fotografowany,

— obiekt wirtualny i rzeczywisty mają ten sam kształt wejściowy,

— cykle przetwarzania obiektu wirtualnego odpowiadają kolejnym fotografiom w sekwencji,
— punkt obserwacji w renderowanej scenie jest zgodny z położeniem i orientacją aparatu fotograficznego,
— współczynnik szybkości topnienia (k_g) jest dobrany doświadczalnie do temperatury otoczenia obiektu rzeczywistego,
— badane cechy geometrii obiektu renderowanego odpowiadają kształtowi obiektu rzeczywistego.



Rysunek 8.1. Ścieżki przetwarzania danych do weryfikacji wyników działania metod.

Aby spełnić powyższe założenia przygotowane zostały dwie ścieżki przetwarzania danych, jedna dla obiektu renderowanego, a druga dla rzeczywistego (rys. 8.1):

A. Pozyskiwanie i przetwarzanie danych obiektu rzeczywistego:
  1. przygotowanie bryły lodu w formie,
  2. wykonanie sekwencji fotografii bryły lodu w trakcie topnienia,

3. uzyskanie sekwencji konturów z fotografii przy pomocy oprogramowania do obróbki grafiki rastrowej (Adobe Photoshop).

B. Pozyskiwanie i przetwarzanie danych obiektu wirtualnego:

1. przygotowanie bryły gipsu w tej samej formie,
2. skanowanie 3D bryły gipsowej, aby uzyskać siatkę trójkątów,
3. wykonanie sekwencji renderingów konturów siatki w kolejnych cyklach topnienia.

Weryfikacja wyników przetwarzania geometrii odbywa się poprzez porównanie sekwencji konturów powstałych z obu ścieżek przetwarzania.

8.2 Konfiguracja stanowiska

Modele gipsowe rekonstruowane były techniką fotomodelowania. Sekwencja fotografii konturów topiącego się obiektu została wykonana w „Laboratorium Rzeczywistości Wirtualnej i Kluczowania Obrazu" Instytutu Informatyki Politechniki Łódzkiej (rys. 8.2).

Rysunek 8.2. Konfiguracja stanowiska fotograficznego w Laboratorium.

— obiekt został umieszczony na wsiąkliwym podłożu, aby nie trzeba było filtrować z obrazu rozlewającej się wody,
— aparat fotograficzny[1] został umieszczony nad obiektem i skierowany pionowo w dół – jest to odpowiednik renderowania na płaszczyźnie poziomej; w scenie rzeczywistej najłatwiej jest uzyskać kontur obiektu w tej płaszczyźnie,
— aparat został odsunięty tak wysoko, jak to było możliwe,
— zoom został ustawiony na maksymalny, co dało minimalny dostępny kąt widzenia – jest to najlepsze przybliżenie rzutowania prostopadłego oferowane przez użyty aparat fotograficzny,
— fotografie były wykonywane co 40 sekund,
— temperatura mieściła się w zakresie $25 - 35°C$,
— początkowa temperatura bryły lodu wynosiła około $-10°C$,
— bryły lodu ważyły około 250 g.

Sekwencja topnienia pojedynczej bryły lodu trwała w powyższych warunkach ponad 2 godziny dając około 200 fotografii w sekwencji.

8.3 Weryfikacja wyników przetwarzania geometrii

Weryfikacja wyników przetwarzania geometrii została przeprowadzona na podstawie badania konturów obiektu rzeczywistego w sekwencji fotografii i obiektu wirtualnego w sekwencji renderingów. **Kontur obiektu** odpowiada w badaniu rzutowi obiektu na płaszczyznę poziomą. Miarą (P) stopnia podobieństwa konturów z renderingu i fotografii jest (wzór 8.1, rys. 8.3):

$$P = \frac{S_A + S_B}{S_A + S_I + S_B}, \tag{8.1}$$

gdzie:

S_A – powierzchnia rzeczywistego konturu z wyłączeniem części wspólnej,

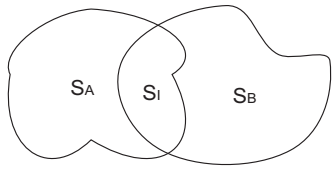
[1] Nikon Coolpix 995, maksymalny zoom optyczny 4×, ustawienia: przesłona $F5.1$, czas 1/15", ISO 100, rozmiar 2048 × 1536 pikseli.

S_B – powierzchnia wyrenderowanego konturu z wyłączeniem części wspólnej,

S_I – powierzchnia części wspólnej.

Wszystkie powierzchnie są obliczane z obrazów jako liczby pikseli; $P \in < 0, 1 >$ i $P = 0$ dla identycznych konturów, a $P = 1$ dla rozłącznych. Miara ta jest więc dogodna w interpretacji. Podobne miary stosowane są w metodach klasyfikacji oraz segmentacji obrazu (np. [94]).

Przyjęto, że kontury są wystarczająco podobne, aby uznać jakość wizualizacji za dostatecznie dobrą, jeśli w badanej sekwencji fotografii i renderingów wartości P nie będą przekraczały $\frac{2}{10}$. Rysunek 8.4 pokazuje dwa przykładowe kontury ustawione względem siebie tak, aby uzyskać wartość $P = 0,20115$. Widać, że mimo takiej wartości P zarysy tych konturów są podobne.



Rysunek 8.3. Oznaczenia użyte w mierze stopnia podobieństwa konturów: S_A kontur obiektu rzeczywistego, S_B kontur obiektu wirtualnego, S_I część wspólna konturów.

Rysunek 8.4. Oznaczenia użyte w mierze stopnia podobieństwa konturów: kod kolorystyczny: ciemny szary – S_A, czarny – S_B, jasny szary – S_I.

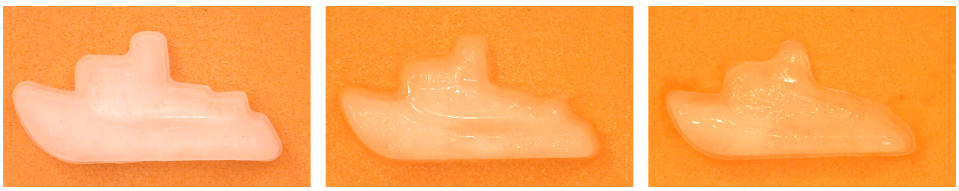
Użyte zostały cztery formy, z których uzyskane zostały zbiory danych o umownych nazwach (rys. 8.5):

1. żyrafa – 10823 trójkąty,
2. samolot – 10683 trójkąty,

3. statek – 10874 trójkąty,
4. okręt – 10799 trójkątów.

Rysunek 8.5. Formy dla lodu i gipsu: żyrafa, samolot, statek, okręt.

Lód podczas topnienia zmienia swój wygląd. Im jest cieńszy tym jest bardziej przezroczysty, a cieńszy staje się w wyniku topnienia. Utrudnia to poprawne odnajdywanie konturu obrazu na tych fotografiach z sekwencji, które wykonane są później, w fazie zaawansowanego stopienia. Najtrudniejsza pod tym względem okazała się bryła lodu powstała z formy okrętu, która już od pierwszych fotografii miała tendencję do zacierania konturów (rys. 8.6).



Rysunek 8.6. Przezroczystość przy zarysie topniejącego obiektu: 1., 50. i 100. fotografia sekwencji.

W praktyce umożliwiło to użycie stu pierwszych fotografii sekwencji wykonywanych dla każdego obiektu. Odpowiada to ponad godzinie rzeczywistego czasu trwania zjawiska topnienia i średnio siedmiuset cyklom obliczeń algorytmów wizualizacji topnienia.

Wykresy umieszczone poniżej przedstawiają wartości podobieństwa konturów dla poszczególnych topionych obiektów: żyrafy (8.7), samolotu (8.8), statku (8.9) oraz okrętu (8.10).

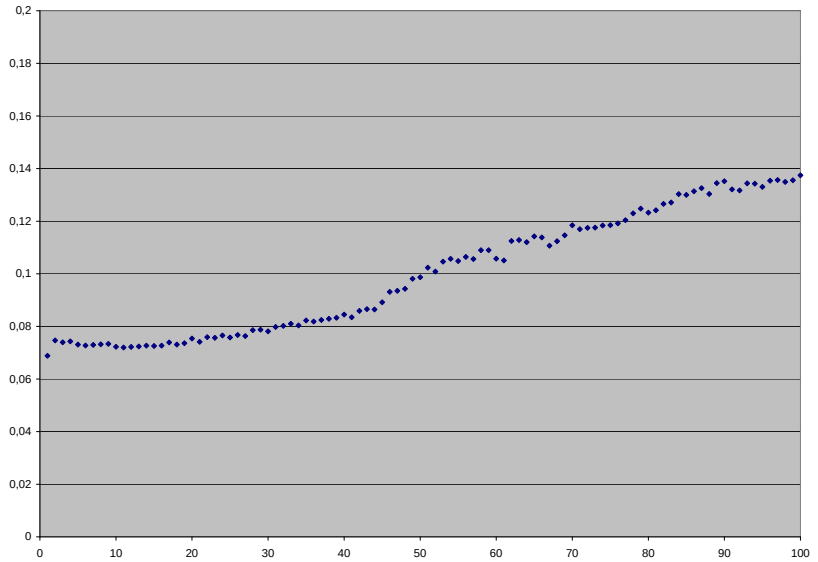
Cechą wspólną tych wykresów jest to, że nawet pierwsza wartość jest różna od zera, co oznacza, że kontur wejściowy z fotografii oraz kontur wejściowy wyrenderowany różnią się od siebie już przed pierwszym cyklem przetwarzania siatki. Ma to kilka przyczyn:

— niedokładność wynikająca z konieczności użycia formy: bryła lodu zawsze nieco różni się od bryły gipsu mimo, że są z tej samej formy; między innymi dlatego, że woda zastygając w formie zwiększa swoją objętość, a gips nie,

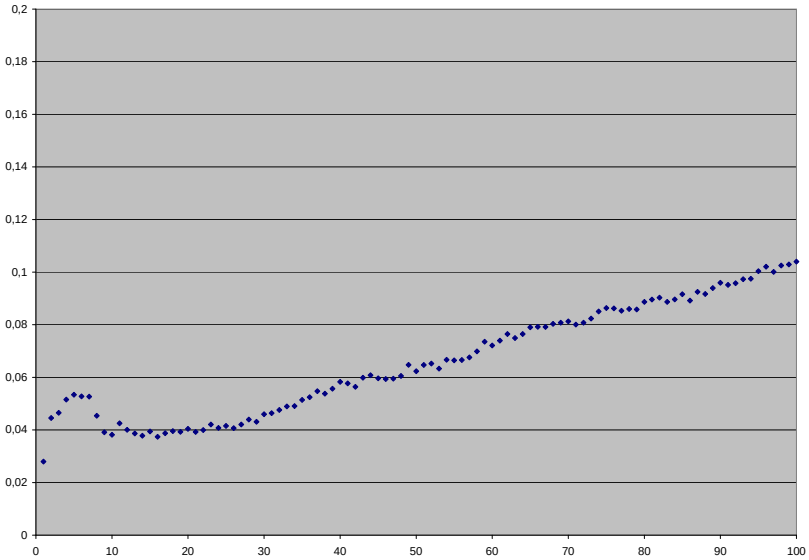
— niedokładność skanowania 3D bryły gipsu i aproksymacja jej powierzchni przez siatkę trójkątów,

— błąd wynikający z geometrii układu optycznego aparatu fotograficznego,

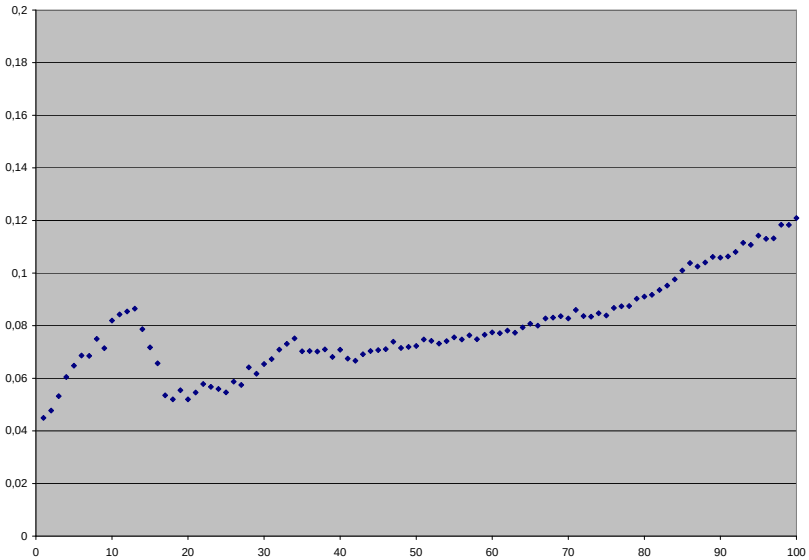
— niedokładność odwzorowania wymiarów, kątów i relacji w scenie renderowanej w stosunku do środowiska rzeczywistego.



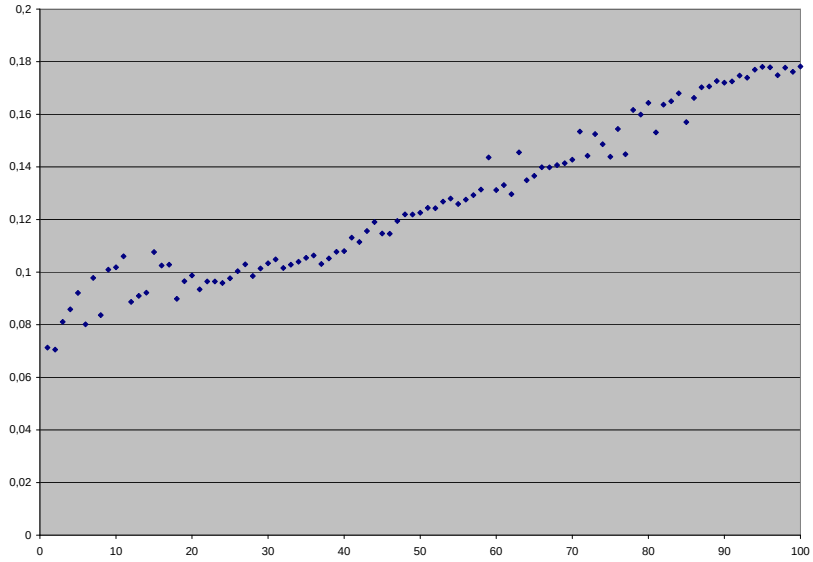
Rysunek 8.7. Wykres zmiany podobieństwa konturów renderingu i fotografii dla żyrafy.



Rysunek 8.8. Wykres zmiany podobieństwa konturów renderingu i fotografii dla samolotu.



Rysunek 8.9. Wykres zmiany podobieństwa konturów renderingu i fotografii dla statku.



Rysunek 8.10. Wykres zmiany podobieństwa konturów renderingu i fotografii dla okrętu.

W świetle powyższych niedogodności należy wartości podobieństwa P_1 wejściowych konturów uznać za dobre (rys. 8.12a na końcu niniejszego rozdziału):

1. dla żyrafy $P_1 = 0,068818$,
2. dla samolotu $P_1 = 0,027969$,
3. dla statku $P_1 = 0,044926$,
4. dla okrętu $P_1 = 0,07131$.

W związku z charakterem wizualizowanego zjawiska, a po części jego nieprzewidywalnością w sensie mnogości czynników, które mogą wpłynąć na jego szybkość i przebieg (przedstawionych w rozdziale 1.1) należało się spodziewać, że wartość P będzie miała tendencję do pogarszania się (rośnięcia) z każdym cyklem obliczeń i z każdą fotografią w sekwencji, choć na wykresach można zaobserwować miejscowe zaburzenia w tej tendencji. Dodatkowym czynnikiem powiększającym wartość P jest kurczenie się zbiorów pikseli wchodzących w skład konturów. Pikseli jest coraz mniej, więc

każde zaburzenie staje się coraz bardziej widoczne, ponieważ każdy piksel ma większy udział w powierzchni konturu.

Na uwagę zasługują także końcowe wartości P_{100} (rys. 8.12d):

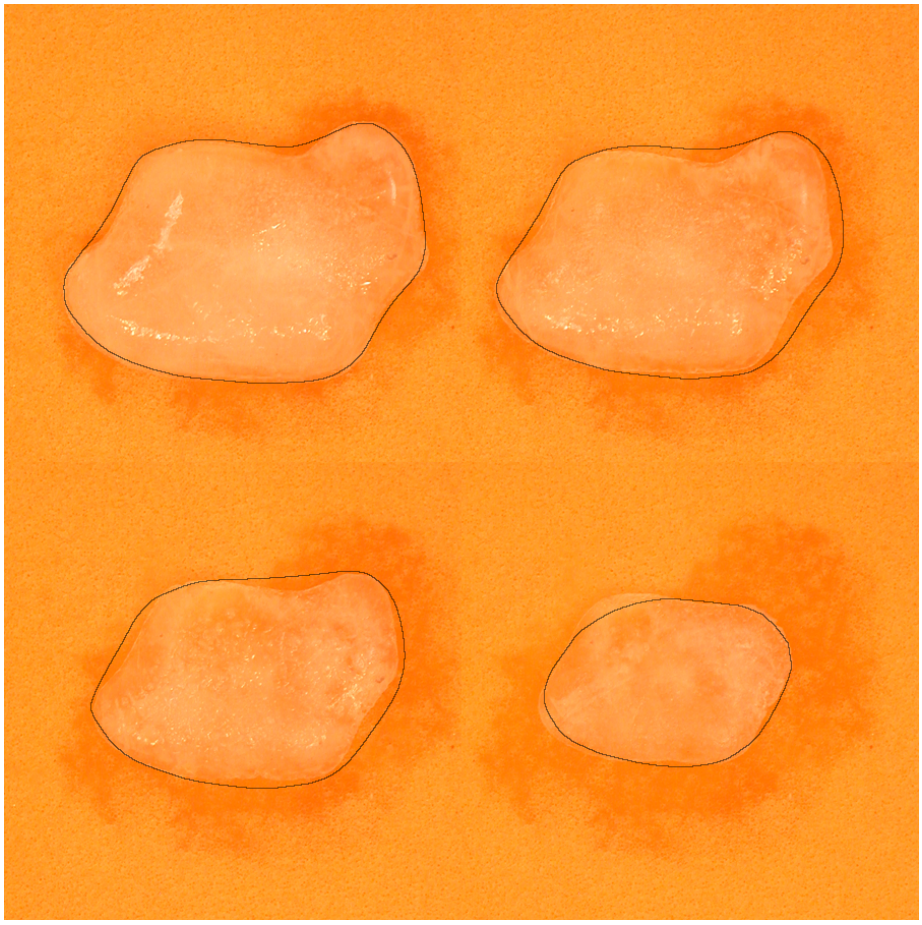
1. dla żyrafy $P_{100} = 0,137467$,
2. dla samolotu $P_{100} = 0,10399$,
3. dla statku $P_{100} = 0,120915$,
4. dla okrętu $P_{100} = 0,178187$.

Relacje między tymi wartościami są takie same, jak dla P_1, co sugeruje, że początkowe dopasowanie konturów rzutuje na dalsze porównywanie i wyniki pomiarów.

Wartość P_{100} dla okrętu zbliżyła się najbardziej do granicy $\frac{2}{10}$. Może to być konsekwencją kilku czynników: okręt był najmniejszym testowanym obiektem, w związku z tym bryła lodu w jego kształcie najszybciej ze wszystkich stawała się przezroczysta powodując niedokładności wyznaczania konturu z fotografii. Ponadto jako najmniejszy był najbardziej podatny na zmieniające się czynniki zewnętrzne podczas wykonywania sekwencji fotografii.

8.4 Porównywanie konturów dla dalszych fotografii sekwencji

Z uwagi na brak możliwości automatycznego porównywania konturów dla dalszych niż setna fotografii sekwencji spróbowano ręcznie dopasować nie przetworzone na kontur fotografie lodu do wyrenderowanej siatki wirtualnego obiektu. Wybrano obiekt samolot, gdyż topił się najdłużej i dał najdłuższą sekwencję fotografii. Rysunek 8.11 pokazuje ręcznie nałożone siatki dla 150., 200., 250. i 300. fotografii w sekwencji. Widać, że mimo ponad dwugodzinnego czasu trwania rzeczywistego zjawiska i wykonania przez aplikację ponad 1800 cykli przetwarzania nadal można na ostatniej parze fotografia-siatka dostrzec pewne podobieństwo konturu.



Rysunek 8.11. Siatka obiektu wirtualnego nałożona na fotografię bryły lodu: 150., 200., 250. i 300. fotografia w sekwencji.

8.5 Podsumowanie

Wyniki określone wartością podobieństwa konturów nie przekraczają $P_{100} = 0,\ 20$. Biorąc pod uwagę podobieństwo wejściowych konturów wynoszące do $P_1 = 0{,}07$ oznacza to różnicę kształtu obliczonego i sfotografowanego na poziomie około 10%. Ponadto kontur obiektu rzeczywistego i wirtualnego są na tyle podobne w badanej fazie przebiegu zjawiska (rys. 8.11 i 8.12), że wyniki przetwarzania geometrii badanych obiektów można uznać za **wiarygodne**.

Rysunek 8.12. Podobieństwo konturów – czarne piksele należą tylko do konturu renderingu, ciemne szare do konturu z fotografii, a jasne szare są częścią wspólną: a) podobieństwo konturów wejściowych (P_1), b) podobieństwo dla 33. fotografii (P_{33}), c) podobieństwo dla 67. fotografii (P_{67}), d) podobieństwo dla setnej fotografii (P_{100}).

9

Podsumowanie

Niniejsza monografia przedstawiła wiele idei i rozwiązań. Pierwszą z nich jest idea wizualizacji zmiany stanu skupienia oparta na pojęciu powierzchni międzyfazowej. Umożliwiło to stosowanie od samego początku powierzchniowej reprezentacji obiektu – siatki trójkątów. Znane z literatury metody bazują w większości na reprezentacjach objętościowych. Ponadto zaprezentowane zostały struktury danych dla siatki trójkątów i algorytmy przetwarzania tej siatki. Aby uwzględnić globalne i lokalne zmiany szybkości przesuwania powierzchni międzyfazowej wprowadzono współczynniki imitujące wpływ temperatury otoczenia i ciśnienia. Algorytmy usuwania z siatki elementów niepotrzebnych z punktu widzenia dalszego przetwarzania, a mogących prowadzić do błędów w następnych cyklach obliczeń (rozdział 3) wprowadzone zostały w celu uzyskania stabilności numerycznej. Następnie przedstawiono potok przetwarzania danych oferowany przez programowalne jednostki graficzne oraz przedyskutowano podstawowe operacje strumieniowe, które można na nich wykonywać i dzięki temu uzyskiwać przyspieszenia działania programów ogólnego przeznaczenia (rozdział 4). W dalszej kolejności opisano rozwiązanie mapowania

na GPU algorytmów deformacji siatki, w tym sposób przechowywania indeksowej reprezentacji siatki trójkątów w teksturach przetwarzanych przez GPU. Reprezentacja ta jest odpowiednikiem reprezentacji przedstawionej w rozdziale 3.5. Przedstawiono również zestaw operatorów strumieniowych działających na tej reprezentacji. Dzięki temu mapowanie wszystkich potrzebnych algorytmów z rozdziału 3.6 polega na wywoływaniu sekwencji tych operatorów w odpowiedniej kolejności z zadanymi zbiorami danych wejściowych i zdefiniowanym wyjściem. Implementacja jest aplikacją GPGPU, ponieważ używa potoku przetwarzania GPU dla przetwarzania danych niezwiązanych bezpośrednio z renderingiem (rozdział 5). Z kolei dla metod związanych bezpośrednio z renderingiem zaprezentowano materiał wieloskładnikowy, który wizualizuje wygląd brył lodu (rozdział 6) oraz sposób obliczania kolejnych faz ruchu plamy wody powstałej w wyniku topnienia obiektu i rozlewającej się po powierzchni podłoża. Metody te są przeznaczone do implementacji na GPU. Korzystają z programowalnego potoku renderingu (rozdział 7). Na końcu opisano metodę weryfikacji poprawności przetwarzania geometrii przy pomocy porównywania konturów rzeczywistej bryły lodu z jej modelem (rozdział 8).

Podsumowując należy podkreślić, że tendencja do mapowania algorytmów czasochłonnych obliczeń na programowalne jednostki graficzne jest uzasadniona znacznym wzrostem wydajności tych jednostek i spadkiem czasu wykonywania na nich obliczeń ogólnego przeznaczenia. Nie inaczej jest w przypadku przedstawionych algorytmów, kiedy po ich zmapowaniu na GPU uzyskać można zauważalnie krótsze czasy przetwarzania danych (do sześciu razy krótszy czas przetwarzania geometrii).

Przyszłe badania rozszerzające przedstawione metody mogłyby:

1. uwzględniać podłoża o różnych kształtach,
2. uwzględnić możliwość tworzenia się otworów w obiekcie podczas trwania topnienia lub sublimacji,
3. uszlachetnić wizualizację cech wyglądu lodu i wody,

4. optymalizować szybkość wykonania programów na GPU poprzez zastosowanie Z-cullingu lub przeniesienie implementacji do środowiska programistycznego CUDA.

Bibliografia

[1] B. Adams, T. Lenaerts, P. Dutré, *Particle Splatting: Interactive Rendering of Particle-Based Simulation Data*, Report CW453, Katholieke Universiteit Leuven, 2006

[2] G. M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, AFIPS spring joint computer conference, 1967

[3] B. G. Baumgart, *Winged-Edge Polyhedron Representation for Computer Vision*, National Computer Conference, Stanford University, 1975

[4] A. M. Bayoumi, Y. Y. Hanafy, *Massive Parallelization of SPICE Device Model Evaluation on GPU-Based SIMD Architectures*, IFMT, ACM, 2008

[5] R. Błaszczyk, *Szybkość krystalizacji i topnienia lodu w strumieniu wody i rozcieńczonych roztworów wodnych*, Praca doktorska przedstawiona na Radzie Wydziału Chemii Spożywczej Politechniki Łódzkiej, 1976

[6] D. Blythe, *The Direct3D 10 System*, Microsoft Corporation, 2006

[7] M. Carlson, P. J. Mucha, R. B. Van Horn III, G. Turk, *Melting and Flowing*, Proceedings of the 2002 ACM SIGGRAPH

[8] M. Carlson, *Rigid, melting, and flowing fluid*, PhD Thesis, Georgia Institute of Technology, 2004

[9] E. Catmull, J. Clark, *Recursively generated B-spline surfaces on arbitrary topological meshes*, Computer-Aided Design 10(6), 1978, 350-355

[10] D. Cederman, P. Tsigas, *GPU-Quicksort: A Practical Quicksort Algorithm for Graphics Processors*, ACM Journal of Experimental Algorithmics, Vol. 14, Article No. 1.4, 2009

[11] H. Chen, H. Sun, *Real-time Haptic Sculpting in Virtual Volume Space*, Proceedings of the ACM symposium on Virtual reality software and technology, ACM 2002

[12] K. Dempski, *Real-Time Rendering Tricks and Techniques in DirectX*, Premier Press, 2002

[13] J. Dorsey, A. Edelman, H. Jensen, J. Legakis, H. Pedersen, *Modeling and rendering of weathered stone*, Proc. SIGGRAPH, 1999, 225-234

[14] N. Dyn, D. Levin, J. A. Gregory, *A butterfly subdivision scheme for surface interpolation with tension control*, ACM Transactions on Graphics (TOG) archive Volume 9, Issue 2, 1990, 160-169

[15] K. Engel, M. Hadwiger, J. M. Kniss, C. Rezk-Salama, D. Weiskopf, *Real-Time Volume Graphics*, A K Peters, Ltd, 2006

[16] W. Engel (editor), *Shader X5. Advanced rendering techniques*, Charles River Media, 2006

[17] G. Falcao, V. Silva, L. Sousa, *How GPUs Can Outperform ASICs for Fast LDPC Decoding*, ICS'09, ACM, 2009

[18] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, P. V. Sander, *GPUQP: Query Co-Processing Using Graphics Processors*, SIGMOD'07, ACM, 2007

[19] P. Fearing, *Computer Modelling Of Fallen Snow, SIGGRAPH*, 37-46, 2000

[20] R. Fernando et al., *GPU Gems*, Addison-Wesley, 2003

[21] J. D. Foley, A. Van Dam, S. K. Feiner, J. F. Huges, *Computer Graphics: Principles and Practice*, Addison-Wesley, 1995

[22] M. Fujisawa, K. T. Miura, *Animation of Ice Melting Phenomenon Based on Thermodynamics with Thermal Radiation*, GRAPHITE 2007, Perth, Western Australia, 2007

[23] I. Fujishiro, E. Aoki, *Volume graphics modeling of ice thawing*, Proc. Volume Graphics Workshop, 2001, 69-81

[24] N. Galoppo, N. K. Govindaraju, M. Henson, D. Manocha, *LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware*, Proceedings of the 2005 ACM/IEEE SC'05 Conference, 2005

[25] J. Gao, E. Ford, J. Peters, *Parallel Integration of Planetary Systems on GPUs*, ACM-SE'08, ACM, 2008

[26] O. Génevaux, F. LARUE, J. DISCHLER, *Interactive Refraction on Complex Static Geometry using Spherical Harmonics*, In Proceedings of the 2006 symposium on Interactive 3D graphics and games, ACM Press, New York, 2006, 145-152

[27] S. F. F. Gibson, *Constrained Elastic Surface Nets: generating smooth surfaces from binary segmented data*, Proc. MICCAI, 1998, 888-898

[28] T. G. Goktekin, A. W. Bargteil, J. F. O'Brien, *A Method for Animating Viscoelastic Fluids*, SIGGRAPH 2004 Papers

[29] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, J. Manferdelli, *High Performance Discrete Fourier Transforms on Graphics Processors*, SC2008, IEEE, 2008

[30] K. Gulati, J. F. Croix, S. P. Khatri, R. Shastry, *Fast Circuit Simulation on Graphics Processing Units*, IEEE, 2009

[31] M. Hadwiger, P. Ljung, C. Rezk-Salama, T. Ropinski, *GPU-Based Volume Ray-Casting with Advanced Illumination*, Tutorial at the IEEE Visualization Conference, 2008

[32] T. D. R. Hartley, U. Catalyurek, A. Ruiz, F. Igual, R. Mayo, M. Ujaldon, *Biomedical Image Analysis on a Cooperative Cluster of GPUs and Multicores*, ICS'08, ACM, 2008

[33] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, P. V. Sander, *Relational Joins on Graphics Processors*, SIGMOD'08, ACM, 2008

[34] D. S. Immel, M. F. Cohen; D. P. Greenberg, *A radiosity method for non-diffuse environments*, Siggraph 1986

[35] D. L. James, D. K. Pai, *ArtDefo: Accurate Real Time Deformable Objects*, SIGGRAPH 99, 1999

[36] M. Jankowski, *Elementy grafiki komputerowej*, WNT, 2006

[37] W.-K. Jeong, K. Kähler, J. Haber, Hans-Peter Seidel, *Automatic Generation of Subdivision Surface Head Models from Point Cloud Data*, A K Peters, 2002

[38] M. Joselli, E. Clua, A. Montenegro, A. Conci, P. Pagliosa, *A New Physics Engine with Automatic Process Distribution between CPU-GPU*, Sandbox Symposium, ACM, 2008

[39] J. T. Kajiya, *The rendering equation*, Siggraph 1986

[40] A. Khan, *Dual-sided Refraction Simulation*, Shadertech Contest, 2004

[41] D. Kharitonsky, J. Gonczarowski, *Physically based model for icicle growth*, The Visual Computer: International Journal of Computer Graphics, Springer-Verlag, 1993, 88-100

[42] P. Kiciak, *Podstawy modelowania krzywych i powierzchni*, Wydawnictwa Naukowo-Techniczne, Warszawa, 2005

[43] R. Fernando, M. J. Kilgard, *The Cg Tutorial: The Definitive Guide To Programmable Real-time Graphics*, Addison Wesley, 2003

[44] T. Kim, M. C. Lin, *Visual Simulation of Ice Crystal Growth*, Eurographics/SIGGRAPH Symposium on Computer Animation, 2003

[45] T. Kim, M. Henson, M. C. Lin, *A Hybrid Algorithm for Modeling Ice Formation*, Eurographics/ACM SIGGRAPH Symposium on Computer Animation, 2004

[46] T. Kim, D. Adalsteinsson, M. C. Lin, *Modeling Ice Dynamics As A Thin-Film Stefan Problem*, Eurographics/ACM SIGGRAPH Symposium on Computer Animation, 167-176, 2006

[47] S. King, R. Crawfis, W. Reid, *Fast volume rendering and animation of amorphous phenomena*, Proc. Volume Graphics Workshop, 1999, 229-242

[48] L. Kobbelt, $\sqrt{3}$-*Subdivision*, International Conference on Computer Graphics and Interactive Techniques, ACM, 2000, 103-112

[49] J. Krüger, R. Westermann, *Acceleration techniques for GPU-based volume rendering*, 2003

[50] T. E. Lewis, G. D. Magoulas, *Strategies to Minimise the Total Run Time of Cyclic Graph Based Genetic Programming with GPUs*, GECCO'09, ACM, 2009

[51] W. E. Lorensen, H. E. Cline, *Marching cubes: A high resolution 3D surface construction algorithm*, Proc. International Conference on Computer Graphics and Interactive Techniques, 1987, 163-169

[52] O. Maitre, L. A. Baumes, N. Lachiche, A. Corma, P. Collet, *Coarse Grain Parallelization of Evolutionary Algorithms on GPGPU Cards with EASEA*, GECCO'09, ACM, 2009

[53] C. Maraffi, *Maya Character Creation: Modeling and Animation Controls*, New Riders Publishing, 2003

[54] W. Ma, G. Agrawal, *A Translation System for Enabling Data Mining Applications on GPUs*, ICS'09, ACM, 2009

[55] T. McReynolds, D. Blythe, *Advanced Graphics Programming Using OpenGL*, Morgan Kaufmann, 2005

[56] Z. Melek, J. Keyser, *Multi-Representation Interaction For Physically Based Modeling*, Proceedings of the 2005 ACM symposium on Solid and physical modeling

[57] P. Mistry, S. Braganza, D. Kaeli, M. Leeser, *Accelerating Phase Unwrapping and Affine Transformations for Optical Quadrature Microscopy using CUDA*, Second Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU 2009), ACM, 2009

[58] M. Moazeni, A. Bui, M. Sarrafzadeh, *Accelerating Total Variation Regularization for Matrix-Valued Images on GPUs*, CF'09, 2009

[59] M. Müller, R. Keiser, A. Nealen, M. Pauly, M. Gross, M. Alexa, *Point Based Animation of Elastic, Plastic and Melting Objects*, Eurographics/ACM SIGGRAPH Symposium on Computer Animation, 2004

[60] M. Müuller, B. Heidelberger, M. Teschner, M. Gross, *Meshless Deformations Based on Shape Matching*, Association for Computing Machinery, Inc., 2005

[61] H. Müller, M. Wehle, *Visualization of Implicit Surfaces Using Adaptive Tetrahedrizations*, Dagstuhl '97, Scientific Visualization, 1997, 243-250

[62] A. Nasri, W. Bou Karam, F. Samavati, *Sketch-Based Subdivision Models*, EUROGRAPHICS Symposium on Sketch-Based Interfaces and Modeling, 2009

[63] H. Nguyen (editor), *GPU Games 3*, Addison-Wesley, 2007

[64] A. Nukada, Y. Ogata, T. Endo, S. Matsuoka, *Bandwidth Intensive 3-D FFT kernel for GPUs using CUDA*, SC2008, IEEE, 2008

[65] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell, *A Survey of General-Purpose Computation on Graphics Hardware*, Computer Graphics Forum, 26(1), 2007, 80–113

[66] N. Ozawa, I. Fujishiro, *A morphological approach to volume synthesis of weathered stone*, Proc. Volume Graphics Workshop, 1999, 367-378

[67] K. S. Perumalla, B. G. Aaby, S. B. Yoginath, S. K. Seal, *GPU-based Real-Time Execution of Vehicular Mobility Models in Large-Scale Road Network Scenarios*, ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation, 2009

[68] M. Pharr, G. Humpreys, *Physically Based Rendering: From Theory to Implementation*, Morgan-Kaufmann, 2004

[69] M. Pharr, R. Fernando, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation* , 2005

[70] S. Raghavachary, *Rendering for Beginners: Image synthesis using RenderMan*, Elsevier, FocalPress, 2005

[71] Ashu Rege, *Shader Model 3.0*, NVIDIA Developer Technology Group, 2004

[72] D. Robilliard, V. Marion, C. Fonlupt, *High Performance Genetic Programming on GPU*, BADS'09, ACM, 2009

[73] C. I. Rodrigues, D. J. Hardy, J. E. Stone, K. Schulten, W. W. Hwu, *GPU Acceleration of Cutoff Pair Potentials for Molecular Modeling Applications*, CF'08, ACM, 2008

[74] D. W. Roeh, V. V. Kindratenko, R. J. Brunner, *Accelerating Cosmological Data Analysis with Graphics Processors*, GPGPU-2 '09, ACM, 2009

[75] T. Ropinski, J. Kasten, K. Hinrichs, *Efficient Shadows for GPU-based Volume Raycasting*, Proceedings of the 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG 2008), 17-24, 2008

[76] S. Seipel, A. Nivfors, *Efficient rendering of multiple refractions and reflections in natural objects*, The Annual SIGRAD Conference, Special Theme: Computer Games, 2006

[77] S. Sengupta, A. E. Lefohn, J. D. Owens, *A Work-Efficient Step-Efficient Prefix-Sum Algorithm*, In Proceedings of the Workshop on Edge Computing Using New Commodity Architectures, 2006, 26–27

[78] D. Shreiner, M. Woo, J. Neider, T. Davis, *The OpenGL Programming Guide*, Addison-Wesley Professional, 6th edition, 2007

[79] J. E. Stone, J. Saam, D. J. Hardy, K. L. Vandivort, W. W. Hwu, K. Schulten, *High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs*, GPGPU'09, ACM, 2009

[80] S. S. Stone, J. P. Haldar, S. C. Tsao, W. W. Hwu, Z.-P. Liang, B. P. Sutton, *Accelerating Advanced MRI Reconstructions on GPUs*, CF'08, ACM, 2008

[81] D. Strippgen, K. Nagel, *Using common graphics hardware for multi-agent traffic simulation with CUDA*, SIMUTools 2009, ICST, 2009

[82] R. Strzodka, C. Garbe, *Real-Time Motion Estimation and Visualization on Graphics Cards*, IEEE Visualization, 2004

[83] B. Sukhwani, M. C. Herbordt, *GPU Acceleration of a Production Molecular Docking Code*, GPGPU '09, ACM, 2009

[84] D. Szajerman, *Mesh representation of simply connected 3d objects in visualisation of melting phenomena*, Machine Graphics & Vision, Vol. 15, No. 3/4, 2006, 621-630

[85] D. Szajerman, A. Wojciechowski, *GPU Computations in Visualization of the Ice Surface*, Proc. of the XII International Conference – System Modelling and Control, Zakopane, Poland, 2007

[86] D. Szajerman, M. Pietruszka, *Real-time ice visualisation on the GPU*, Journal of Applied Computer Science, Vol. 16, No. 2, 2008, 89-106

[87] D. Szajerman, M. Szymczyk, *GPU-based optimisation of volumetric ablation rendering*, Proc. of the XIII International Conference – System Modelling and Control, Zakopane, Poland, 2009

[88] M. Pietruszka, D. Szajerman, *The Method for Verifying Correctness of the Shape's Changes Calculation in the Melting Block of Ice*, International Conference on Computer Vision and Graphics 2010, Computer Vision and Graphics, LNCS 6475, Springer, 2010

[89] S. Szczeniowski, *Fizyka doświadczalna. Wyd. I. Cz. II: Ciepło i fizyka cząsteczkowa*, Państwowe Wydawnictwo Naukowe, Warszawa, 1953

[90] A. Szczęsna, *Wielorozdzielcze przetwarzanie nieregularnych siatek powierzchni z wykorzystaniem falek drugiej generacji*, Rozprawa doktorska, Politechnika Śląska, 2007

[91] G. Tan, Z. Guo, M. Chen, D. Meng, *Single-particle 3D Reconstruction from Cryo-Electron Microscopy Images on GPU*, ICS'09, ACM, 2009

[92] D. Terzopoulos, J. Platt, A. Barr, K. Fleischer, *Elastically deformable models*, Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH 87, 1987, 205–214

[93] D. Theodoropoulos, C. B. Ciobanu, Georgi Kuzmanov, *Wave Field Synthesis for 3D Audio: Architectural Prospectives*, CF'09, ACM, 2009

[94] Arkadiusz Tomczyk, Piotr S. Szczepaniak, *Segmentation of Heart Image Sequences Based on Human Way of Recognition*, Springer Lecture Notes in Computer Science, 2009, 225-235

[95] S. Tsutsui, N. Fujimoto, *Solving Quadratic Assignment Problems by Genetic Algorithms with GPU Computation: A Case Study*, GECCO'09, ACM, 2009

[96] H. Varadhan, K. Mueller, *Volumetric Ablation Rendering*, The Eurographics Association, 2003, 53-60

[97] A. Watt, F. Policarpo, *3D Games. Real-time Rendering and Software Technology*, ACM Press, 2001, 81

[98] X. Wei, W. Li, K. Mueller, A. Kaufman, *Simulating Fire with Texture Splats*, IEEE Visualization, 2002, 227-237

[99] M. Wloka, *Technical Report. Fresnel Reflection*, nVidia, 2002

[100] M. L. Wong, *Parallel Multi-Objective Evolutionary Algorithms on Graphics Processing Units*, GECCO'09, ACM, 2009

[101] R. Wu, B. Zhang, M. Hsu*Clustering Billions of Data Points Using GPUs*, UCHPC-MAW'09, ACM, 2009

[102] C. WYMAN, *An Approximate Image-Space Approach for Interactive Refraction*, In Proceedings of ACM SIGGRAPH 2005, ACM Press, New York, 2005, 1050-1053

[103] Z. Xu, R. Bagrodia, *GPU-accelerated Evaluation Platform for High Fidelity Network Modeling*, 21st International Workshop on Principles of Advanced and Distributed Simulation, 2007

[104] M. P. M. Zamith, E. W. G. Clua, A. Conci, A. Montenegro, R. C. P. Leal-Toledo, P. A. Pagliosa, L. Valente, B. Feij, *A Game Loop Architecture for the GPU Used as a Math Coprocessor in Real-Time Applications*, ACM Computers in Entertainment, Vol. 6, No. 3, Article 42, 2008

[105] nVidia, *Cg Toolkit. User's manual. A developer's Guide to Programmable Graphics*, 2005

[106] *Vector*, Electromagnetics Newsletter, vol. 12, no. 2, 1996

Publikacje internetowe:

[107] M. Bieńkowski, *DirectX10 bez tajemnic*, witryna internetowa (stan na 18. października 2009) `http://pclab.pl/art26921-5.html`

[108] E. W. Weisstein, *Euler Characteristic*, witryna internetowa (stan na 16. października 2009) `http://mathworld.wolfram.com/EulerCharacteristic.html`

[109] nVidia, *Technical Brief NVIDIA GeForce GTX 200 GPU Architectural Overview*, `http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf`

[110] *GPGPU Programming*, witryna internetowa (stan na 17. października 2009) `http://gpgpu.org`

[111] *The Winged-Edge Data Structure*, witryna internetowa (stan na 21. października 2009) `http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/model/winged-e.html`

Spis rysunków

1.1 Powierzchnia międzyfazowa (*na podstawie opisu w [5]*). . . . 10
1.2 Klasyfikacja wizualizacji z punktu widzenia kierunku zmian stanów skupienia. Wybrane techniki uporządkowane są według czasu obliczenia pojedynczego kroku przetwarzania geometrii i renderingu jednej ramki. 13
1.3 Idea wolumetrycznego renderingu ablacji: a) rozkład energii cieplnej pochodzącej z promienia na powierzchni materiału, b) źródła danych dla renderingu objętościowego (*źródło: [96]*). 16
1.4 Topnienie wiolonczeli z góry (*źródło: [96]*). 16
1.5 Rzeźbienie w wirtualnej objętości: a) interfejs sprzętowy do wirtualnego rzeźbienia, b) topienie modelu głowy (*źródło: [11]*). . 17
1.6 Ciała topiące się i krzepnące zamodelowane jako ciecz o zadanej lepkości: a) topnienie bloku wosku, b) topnienie woskowego modelu Stanford Bunny (*źródło: [7]*). 18
1.7 Ablacja ciała z modelu głowy (*źródło: opracowanie własne [87]*). 19

2.1 Klasyfikacja reprezentacji obiektów trójwymiarowych. 28
2.2 Prymitywy trójwymiarowe (prostopadłościan, sfera, walec, stożek, graniastosłup i torus) dostępne w aplikacji AutoCAD. 29
2.3 Konstrukcja brył CSG: obiekt końcowy powstał w programie 3ds Max jako wynik odejmowania części wspólnej sfery i walca oraz sumy dwóch prostopadłościanów. 31

2.4 Reprezentacje z przesuwaniem. 32
2.5 Kwadryki: sfera, walec, elipsoida, stożek, paraboloida, hiperboloida 32
2.6 Krzywa Beziera, krzywa B-sklejana, krzywa typu Cardinal oraz NURBS. 33
2.7 Powierzchnia składająca się z dwóch płatów. 34
2.8 Model głowy utworzony z płatów NURBS: a) jednego; b) wielu [53]. 35
2.9 Modelowanie subdivision dłoni: siatka kontrolna, siatka graniczna, rendering [62]. 37
2.10 Siatki MES wygenerowane automatyczne przez pakiet programowy OPERA: a) model magnesu – obszarami regularna siatka zbudowana z elementów sześciennych, b) model maszyny elektrycznej – nieregularna siatka zbudowana z elementów czworościennych [106]. 38
2.11 Wizualizacja zbioru medycznego złożonego z wokseli [87]: a) strukturę wokselową można zaobserwować na podbródku, b) efekt działania transfer function na atrybutach wokseli. . . 39
2.12 Siatka trójkątów składająca się z wierzchołków, krawędzi i trójkątów. 40
2.13 Tetrahedron w reprezentacji: a) bezpośredniej, b) indeksowej, c) krawędziowej i Baumgarta. Przyjęte zostały następujące oznaczenia elementów w reprezentacjach: wierzchołków przy pomocy wielkich liter A, B, C, D, trójkątów przy pomocy liczb 1, 2, 3, 4 oraz krawędzi przy pomocy małych liter a, b, c, d, e, f. 41
2.14 Skręcenie trójkąta. 47
2.15 Miara d kąta dwuściennego dla krawędzi BC. N_{ABC} i N_{BCD} są normalnymi trójkątów krawędzi BC (odpowiednio ABC i BCD). 47

2.16 Cieniowanie sfery reprezentowanej przez siatkę trójkątów: a) płaskie, b) Gourauda, c) Phonga. 49

2.17 Mapowanie tekstury na siatkę trójkątów. 50

2.18 Etapy klasycznego potoku renderingu. 52

2.19 Model oświetlenia Phonga-Blinna: a) składowa ambient; b) dołączona składowa diffuse; c) dołączona składowa specular; d) dołączona składowa emission. 54

2.20 Wektory w modelu oświetlenia Phonga-Blinna. 54

2.21 Łączenie koloru fragmentu z kolorem teksela: a) rozkład oświetlenia został obliczony dla fragmentów, więc operator modulate pomnożył kolor fragmentu przez kolor teksela; b) rozkład oświetlenia zapisany jest w fotograficznej teksturze, więc operator replace zastąpił kolor fragmentu kolorem teksela. 57

2.22 Etapy programowalnego potoku renderingu. 58

2.23 Wektory w modelu oświetlenia Phonga. 59

2.24 Porównanie renderingu klasycznego z renderingiem zaprogramowanym na GPU: a) Phong-Blinn z cieniowaniem Gourauda; b) Phong z cieniowaniem Phonga zaimplementowanym w programie cieniowania. Na rzadkiej siatce cieniowanie Gourauda zgubiło rozbłysk. 61

2.25 Mapowanie środowiska na obiekcie odbijającym zwierciadlanie: klasyczne (po lewej), przy pomocy programu cieniowania (po prawej). 62

2.26 Mapowanie środowiska na obiekcie przezroczystym: klasyczne (po lewej), przy pomocy programu cieniowania (po prawej). 62

3.1 Błędy w siatce deformowanej przy pomocy M_N^i: siatka oraz rendering obiektu po 1500 cyklach. Widać trójkąty, które „wyszły” z wnętrza obiektu. 70

3.2 Wszystkie wektory M_N^i są skierowane do wnętrza obiektu 2D. Wektor M_N^1 jest najdłuższy, pozostałe krótsze. Długość nie zależy wprost od wysunięcia wierzchołka. Dwuwymiarowa analogia obiektu 3D – krawędzie w 2D reprezentują trójkąty w 3D. 71

3.3 Wektory E_j^i leżące wzdłuż krawędzi wychodzących z i-tego wierzchołka i ich suma M_E^i. 72

3.4 Wektor M_E^1 jest krótki, ponieważ wierzchołek (1) tworzy nieznaczną wypukłość. Wektory M_E^2 i M_E^3 są dłuższe i takiej samej długości mimo, iż wierzchołek (2) tworzy wypukłość, a (3) wklęsłość. Dwuwymiarowa analogia obiektu 3D – krawędzie w 2D reprezentują trójkąty w 3D. 73

3.5 Deformowana siatka: wejściowa, po 800 cyklach, po 1600 cyklach. 75

3.6 Usuwanie krawędzi: usunięty zostaje wierzchołek B na rzecz A – drugiego wierzchołka usuwanej krawędzi; usunięte są również dwa zaznaczone trójkąty. 76

3.7 Sklejenie trójkątów ACD i BDC podczas usuwania krawędzi AB. 77

3.8 Podział obiektu na części: wynik trzech kolejnych cykli obliczeń. 78

3.9 Wpływ zewnętrznych czynników lokalnych na długość wektora przesunięcia: a) wpływ rozkładu temperatury – M^1 jest dłuższy od M^2, ponieważ leży w cieplejszym obszarze; b) wpływ ciśnienia – M^1 jest dłuższy od M^2, ponieważ na dolną powierzchnię bryły działa siła nacisku. 80

3.10 Obiekt topiony z uwzględnieniem lokalnego współczynnika szybkości topnienia, który modeluje punktowe źródło ciepła oraz płaski nacisk podłoża. Zaznaczone przezroczyste obszary pokazują zasięg działania każdego źródła. 81

3.11 Likwidacja *łańcuchów przeindeksowań*. 88

4.1 Zmiana liczby operacji zmiennoprzecinkowych na sekundę na przestrzeni ostatnich lat – porównanie CPU i GPU (*źródło: na podstawie* `http://intel.com`, `http://www.gpureview.com`, `http://www.karlrupp.net`). 92
4.2 Architektura procesora graficznego GeForce GTX 280 (*źródło: [109]*). 93
4.3 Rozszerzony potok renderingu modelu cieniowania 4.0 (*źródło: na podstawie [6]*). 98
4.4 Wierzchołki przetwarzanego w GS prymitywu (V) i wierzchołki prymitywów przylegających (P) (*źródło: na podstawie [6]*). 100
4.5 Próbkowanie tekstury w punkcie (s,t): a) w przypadku próbkowania najbliższego sąsiada zostanie zwrócona wartość teksela o współrzędnych (s_2,t_1) w mapie tekstury, w przypadku próbkowania z interpolacją zwrócona zostanie zinterpolowana dwuliniowo wartość czterech sąsiednich (oznaczonych szarym kolorem) tekseli; b) jedna operacja próbkowania dokładnie w środku między czterema tekselami odczytuje średnią wartość ich kolorów. 108
4.6 Jednoetapowe przetwarzanie GPGPU. 111
4.7 Operacje na pamięci: a) gather – pośredni odczyt, b) scatter – pośredni zapis. 117

5.1 Indeksy tablicy odpowiadające adresom tekstury. 127
5.2 Przetwarzanie danych z tekstury za pomocą rysowania czworokąta zakresu przetwarzania. 127
5.3 Elementy nadmiarowe w wierszu. Jest ich mniej, gdy tekstura ma mniejszą szerokość niż wysokość. 128
5.4 Odczyt współrzędnych wierzchołków trójkąta. 130
5.5 Operacja scatter realizowana na programie cieniowania wierzchołków (VS) lub geometrii (GS). 133

5.6 Czas wykonania obliczeń pojedynczego cyklu w milisekundach na CPU i na GPU. Początkowo obiekt B składał się z 16000 trójkątów. 141
5.7 Czas wykonania obliczeń pojedynczego cyklu w milisekundach na CPU i na GPU. Początkowo obiekt C składał się z 32000 trójkątów. 143

6.1 Światło rozproszone: a) składniki ambient i diffuse, b) fotografia wyglądu lodu, c) tekstura fotografii lodu zmapowana na obiekt d) modulacja tekstury wyglądu lodu przy pomocy oświetlenia ambient i diffuse. 147
6.2 Odczyt mapy środowiska: a) wektor odbicia otoczenia (R_e), b) zależnie od współrzędnych wektora R_e lub T wybrana jest jedna z sześciu ścian-tekstur i próbkowany jest jej teksel. . . . 149
6.3 Odczyt mapy środowiska: wektor załamania światła (T). . . . 151
6.4 Współczynnik Fresnela: a) materiał zwierciadlany, b) materiał przezroczysty załamujący światło, c) połączenie odbicia i załamania. 153
6.5 Modyfikacja normalnych powierzchni powoduje wrażenie jej nierówności (linia przerywana). 153
6.6 Obiekt oświetlony przy pomocy modelu oświetlenia Phonga: a) z mapowaniem nierówności; b) z mapowaniem nierówności oraz modulacją tekstury lodu (t_{diff}). 154
6.7 Materiał wieloskładnikowy: a) lód o gładkiej powierzchni; b) lód o nierównej powierzchni. 154
6.8 Materiał wieloskładnikowy na sublimującym obiekcie. 155
6.9 Sfotografowana (po lewej) oraz wyrenderowana (po prawej) bryłka lodu. 155
6.10 Obiekty testowe: a) bunny (około 69k trójkątów), b) dragon (około 871k trójkątów), c) 9 obiektów dragon (około 7842k trójkątów) 156

6.11 Wykres zależności czasu trwania renderingu ramki od liczby obiektów dragon (zawierających 871414 trójkątów każdy). . . 157

7.1 Fotografia rozlanej wody. 160

7.2 Wizualizacja cieczy: a) oryginalny widok podłoża; b) tekstura maski – biały kolor identyfikuje położenie plamy cieczy. . . . 161

7.3 Zmiana koloru oryginalnej tekstury: a) wizualizacja „przezroczystej" plamy; b) dołączony składnik odbić środowiska. . . . 163

7.4 Wektory normalnych: a) obliczone według cech geometrycznych obiektu plamy; b) obliczone z maski i równania (7.3). . . 164

7.5 Wrażenie grubości warstwy cieczy: a) efekt modyfikacji wektora normalnego na brzegach plamy, aby stworzyć wrażenie ich wypukłości, b) odczyt tekstury podłoża z przesunięciem, aby stworzyć wrażenie załamania światła w warstewce wody. 165

7.6 Tekstura maski po 100, 400, 700, 1000, 2000, 3000, 6000, 10000 cyklach. Na szaro wyrenderowany jest rozmyty stan poprzedni a na biało aktualnie wyrenderowany kształt dołu obiektu topionego. 170

7.7 Wpływ cech podłoża na rozlewanie się cieczy: a) tekstura cech powierzchni podłoża, b) plama cieczy progowana stałą wartością, c) plama cieczy progowana teksturą cech powierzchni. 171

7.8 Wizualizacja topnienia na początku oraz po 800, 1600 i 2547 cyklach. 173

8.1 Ścieżki przetwarzania danych do weryfikacji wyników działania metod. 176

8.2 Konfiguracja stanowiska fotograficznego w Laboratorium. . . 177

8.3 Oznaczenia użyte w mierze stopnia podobieństwa konturów: S_A kontur obiektu rzeczywistego, S_B kontur obiektu wirtualnego, S_I część wspólna konturów. 179

8.4 Oznaczenia użyte w mierze stopnia podobieństwa konturów: kod kolorystyczny: ciemny szary – S_A, czarny – S_B, jasny szary – S_I. 179
8.5 Formy dla lodu i gipsu: żyrafa, samolot, statek, okręt. 180
8.6 Przezroczystość przy zarysie topniejącego obiektu: 1., 50. i 100. fotografia sekwencji. 180
8.7 Wykres zmiany podobieństwa konturów renderingu i fotografii dla żyrafy. 181
8.8 Wykres zmiany podobieństwa konturów renderingu i fotografii dla samolotu. 182
8.9 Wykres zmiany podobieństwa konturów renderingu i fotografii dla statku. 182
8.10 Wykres zmiany podobieństwa konturów renderingu i fotografii dla okrętu. 183
8.11 Siatka obiektu wirtualnego nałożona na fotografię bryły lodu: 150., 200., 250. i 300. fotografia w sekwencji. 185
8.12 Podobieństwo konturów – czarne piksele należą tylko do konturu renderingu, ciemne szare do konturu z fotografii, a jasne szare są częścią wspólną: a) podobieństwo konturów wejściowych (P_1), b) podobieństwo dla 33. fotografii (P_{33}), c) podobieństwo dla 67. fotografii (P_{67}), d) podobieństwo dla setnej fotografii (P_{100}). 186

Spis tabel

2.1 Zestawienie cech reprezentacji siatek trójkątów. Łatwość operacji rozumiana jest jako brak konieczności przeszukiwania całej tablicy elementów, w celu znalezienia poszukiwanego. . 43
2.2 Operatory Eulera: tworzenie i usuwanie elementów siatki. . . 45
2.3 Charakterystyki Eulera dla wybranych siatek. 46
2.4 Wybrane cechy reprezentacji. Symbol + oznacza spełnienie kryterium, **o** spełnienie pod pewnymi warunkami, – niespełnienie lub duże trudności w spełnieniu. Liczba + w kolumnie „oszczędność" to stopień oszczędności. 66

3.1 Zestawienie liczby operacji na tablicach według reprezentacji. 83

4.1 Porównanie możliwości poszczególnych modeli cieniowania (*na podstawie [71, 6, 107]*). 123

5.1 Wyniki testów szybkości dla obiektów testowych. Czasy podane są w milisekundach. 140

6.1 Zestawienie czasu (w ms) renderingu pojedynczej ramki zawierającej obiekty o dużej liczbie trójkątów z materiału wieloskładnikowego. 158

ISBN 978-83-7283-888-9