Praca inżynierska p.t.:

„Wizualizacja wzrostu roślin w grach komputerowych”

Autor: Michał Chojnacki, Politechnika Łódzka, kierunek: informatyka (WFTiMS)

Spis treści

Rozdział I - wstęp

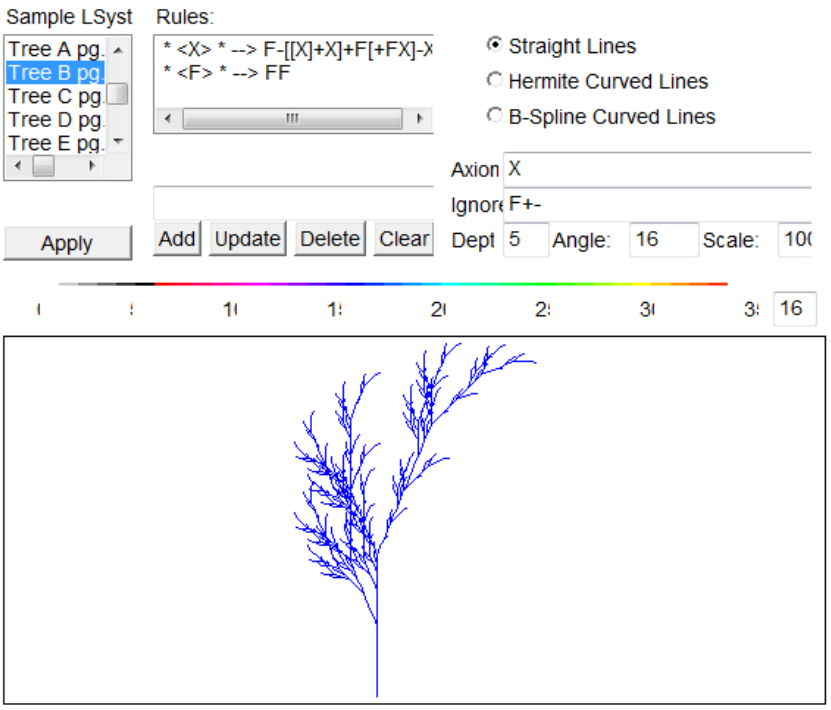
Przez ostatnią dekadę można zaobserwować wielki postęp w dziedzinie generowania i wyświetlania w czasie rzeczywistym efektów trójwymiarowych. Rosnące wymagania użytkowników dotyczące estetyki gier komputerowych zmuszają projektantów do tworzenia coraz to bardziej realistycznych środowisk. Wraz ze wzrostem ilości pamięci operacyjnej oraz przepustowości kart graficznych w komputerach przeciętnych graczy, coraz częściej są to środowiska otwarte – rozmiaru wioski, miasta lub niekiedy nawet wielkiej metropolii. Ważnym składnikiem dekoracyjnym, mającym wpływ na ich realizm jest roślinność – pod postacią drzew, zarośli i trawy. Ich ręczne modelowanie jest bardzo pracochłonnym i powtarzalnym zajęciem.

Celem pracy jest opracowanie systemu umożliwiającego tworzenie roślinności w różnych stadiach rozwoju oraz obserwowanie jej wzrostu. Docelowo ma on za zadanie generowanie rośliny, aż do osiągnięcia żądanego stadium i zapisanie efektu jako modelu 3D. Połączenie elementów losowości ze ściśle definiowaną gramatyką, określającą specyfikę rośliny umożliwi generowanie przewidywalnych, acz zróżnicowanych roślin, odciążając projektantów gry z powtarzalnych zadań, a pozwalając im skupić się na bardziej wyrafinowanych częściach środowiska gry.

Zakres pracy obejmuje analizę sposobu reprezentacji figur samopodobnych (fraktali), za pomocą których mogą być przedstawione drzewa i byliny. Nacisk zostanie położny na system Lindenmayera – język formalny, który jest obecnie używany do opisu roślin. Efektem działania systemu jest graf trójwymiarowy, którego węzły są miejscami podziału konarów na gałęzie. Zadaniem części wizualizacyjnej jest przekształcenie grafu na trójwymiarowy, oteksturowany model, który można oglądać ze wszystkich stron. Zostanie to zrealizowane w języku C#, na platformie XNA 4.0. Proces konstruowania modelu jest wieloetapowy – wymaga wygładzenia krawędzi grafu rośliny za pomocą krzywych Beziera, obudowania ich cylindryczną siatką, wygenerowania współrzędnych mapowania tekstur i ostatecznie nadaniu losowości powierzchni. Modele mogą być eksportowane w formacie COLLADA. Wszystkie etapy zostaną omówione w kolejnych rozdziałach.

Bieżący stan zagadnienia

Generowanie roślin za pomocą L-systemów jest dość popularnym zadaniem. W Internecie pełno jest apletów Javy, w których można generować rośliny w dowolnym stadium rozwoju, a nawet określać własne reguły produkcji. Jednakże żadna z tych aplikacji nie pozwala na płynną kontrolę wzrostu drzewa. Można co najwyżej określić ilość iteracji L-systemu, która jest wartością dyskretną.



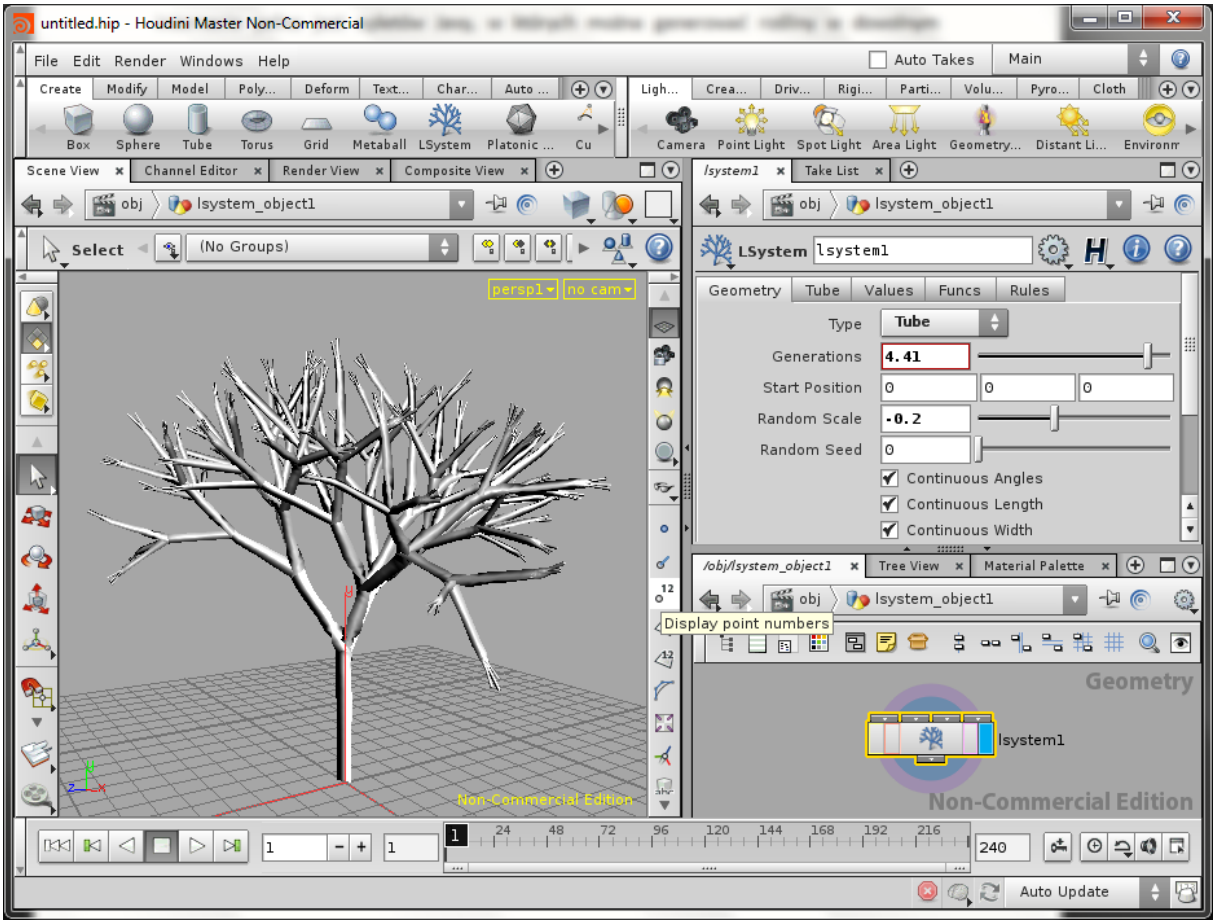
Aplet z http://www.biologie.uni-hamburg.de/b-online/lsystjava/LSys.html

Program Houdini to rozbudowany kombajn do tworzenia animacji, który implementuje również L-systemy. To, co go odróżnia od apletów dostępnych w Internecie to możliwość płynnej animacji wzrostu drzewa. Stopień rozwoju drzewa regulowany jest przez suwak, określający jego wiek. Program oferuje 3 rodzaje interpolacji:

1. Płynne kąty – powoduje to wyrastanie nowych gałęzi współliniowych do macierzystej gałęzi, a wraz z ich starzeniem rozchodzenie się ich na boki, aż do osiągnięcia zadanego kąta.
2. Płynna długość – gałęzie wydłużają się stopniowo, zamiast być rysowane generacja po generacji, jak w popularnych w Internecie apletach.

3. Płynna grubość – gałęzie zwężają się ku końcowi tak, aby ich grubość w tym miejscu była równa grubości gałęzi potomnych

Pod względem realizmu najważniejsze jest płynne wydłużanie się gałęzi, tak więc na to będzie położony największy nacisk w pracy.



Wszystkie napotkane implementacje L-systemów operowały na regułach podawanych w formie tekstowej. Program Houdini oferuje aż 33 symbole tekstowe, często w formie znaków ~ ! @ ^ itd., które mogą być bardzo trudne do zapamiętania. Jednym z celów tej pracy będzie stworzenie programu, w którym użytkownik nie będzie musiał znać dziesiątek symboli, a będzie miał je podane w formie graficznych bloków.

Rozdział II – systemy Lindenmayera

Kod genetyczny jako czynnik sterujący wzrostem.

Rozwój organizmów żywych prowadzi do powstania wysoce skomplikowanych form, składających się z milionów elementów. Tworzą one złożoną, acz zorganizowaną strukturę przestrzenną, której kształt zależny jest od indywidualnego toku ewolucji organizmu. Ewolucja na przestrzeni jednostki jest złożeniem zmian, które następowały w czasie trwania wszystkich poprzednich pokoleń, oraz zmian własnych, będącym efektem wzrostu i rozwoju. Przebieg tych drugich uwarunkowany jest kodem genetycznym, będącym zbiorem instrukcji, które mają bezpośredni wpływ na proces rozwoju organizmu.

Z punktu widzenia informatyki można zauważyć, że genom (materiał genetyczny zawarty w podstawowym zespole chromosomów) jest bardzo wygodnym tworem, ponieważ gwarantuje równoległe wykonanie szeregu identycznych operacji na zbiorze wszystkich komórek organizmu (architektura SIMD). W dziedzinie symulowania rozwoju roślin, które w dużym uproszeniu są figurami samopodobnymi (fraktalami), istnieje naturalna potrzeba implementacji takiego kodu genetycznego. Dzięki temu za pomocą jednego ciągu operacji możliwe było by automatyczne wygenerowanie całej rośliny w dowolnym stadium rozwoju. Pozwoliłoby to odciążyć programistę z obowiązku programowej kontroli rozwoju każdego fragmentu rośliny, ponieważ cały „scenariusz” byłby zakodowany w genomie i wykonywałby się wszędzie, gdzie to niezbędne, z każdą iteracją symulacji.

L-systemy.

Odpowiedzią na tę potrzebę są L-systemy. Wprowadzone w 1968 roku przez węgierskiego biologa Aristida Lindenmayera są gramatyką formalną, służącą do modelowania wzrostu i rozwoju roślin. Początkowo L-systemy znalazły zastosowanie przy opisie wzrostu prostych organizmów, takich jak drożdże, glony czy kolonie bakterii. W późniejszym czasie zostały rozszerzony tak, że nadawały się do modelowania bardziej złożonych rośliny, jak drzewa czy byliny.

Elementy składowe L-systemu

- *A* – alfabet, czyli zestaw symboli, który jest wykorzystywany przez reguły produkcji
- *ω* – stan początkowy (aksjomat), który stanowi punkty wyjścia dla algorytmu
- *P* – zbiór reguł produkcji, wyrażany w postaci sposobu zamiany istniejących symboli na nowe, w obrębie alfabetu. Reguła produkcji składa się z poprzednika i następnika, np. A → AB

Główne cechy L-systemów.

- Rekursywność. L-systemy służą do opisu roślin, które w dużym uproszczeniu są figurami samopodobnymi (fraktalami), tak więc rekursja jest tutaj w pełni uzasadniona – jest ona naturalną metodą opisu figur fraktalnych. Poprzez zwiększanie głębokości rekursji można w prosty sposób kontrolować stopień rozwoju organizmu oraz jego złożoność.
- Dyskretność. Cecha ta wywodzi się z dyskretności samego języka formalnego, tworzonego przez gramatykę z alfabetu, który musi być zawsze skończonym zbiorem. Istnieją rozwinięcia L-systemów, które działają w sposób ciągły, lecz podstawowa ich forma pozostaje dyskretna. Ma to w konkretne odzwierciedlenie w naturze, ponieważ roślinę można traktować jako skończony zbiór części składowych, takich jak konary, gałęzie, łodygi itd.
- Bezkontekstowość. L-systemy działają na przestrzeni podstawowej jednostki budulcowej danego organizmu (analogia do komórki w biologii), nie uwzględniając sąsiadów, ani parametrów środowiska, w którym się znajduje (np. ograniczenie przestrzeni życiowej). Reguły produkcji są stosowane lokalnie, do konkretnych elementów. Jest to w zupełności wystarczające dla symulacji tworów, z myślą o których L-systemy zostały stworzone, lecz może stanowić problem przy generowaniu większych lub bardziej skomplikowanych roślin.

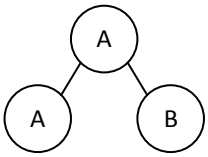
Zasada działania L-systemów.

U podstaw L-systemów leży pojęcie przepisywania. Polega ono na zastępowaniu termów formuły za pomocą innych termów, zgodnie z uprzednio zdefiniowanymi regułami. Rozwinięcie tego do *równoległego przepisywania* zakłada stosowanie reguł jednocześnie, w każdej iteracji. Spójrzmy na jeden z pierwszych L-systemów, stworzonych przez Lindenmayera, służący do opisu rozwoju glonów. Zgodnie z wcześniejszym opisem składników, mamy:

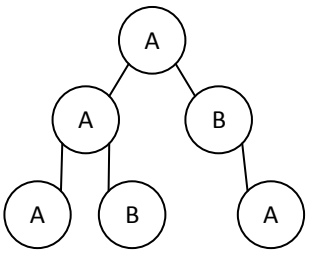
- Alfabet: A B
- Stan początkowy: A
- Reguły produkcji:
  - A → AB
  - B → A

Iteracja 0:

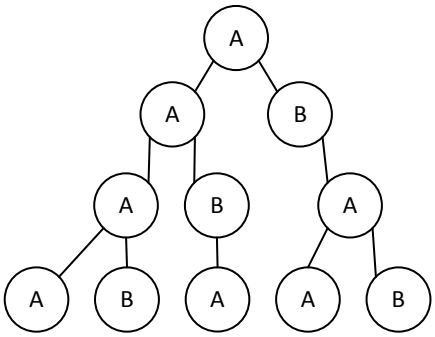
Iteracja 1:



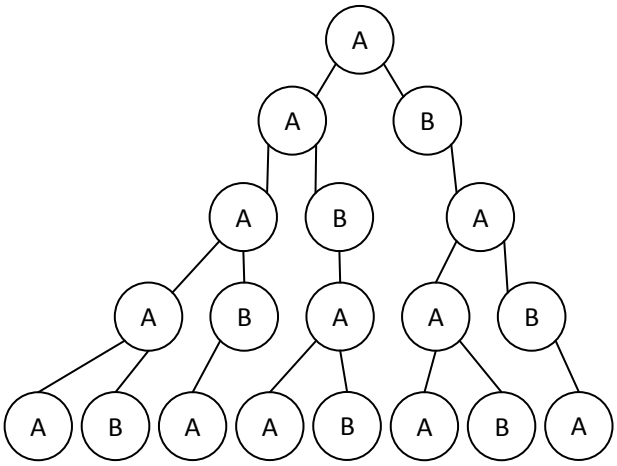
Iteracja 2:



Iteracja 3:



Iteracja 4:



Ciekawostką jest, że powyższy L-system generuje taki organizm, którego liczby elementów powstałych w każdej iteracji są kolejnymi składnikami ciągu Fibonacciego.

Grafika żółwia

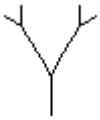
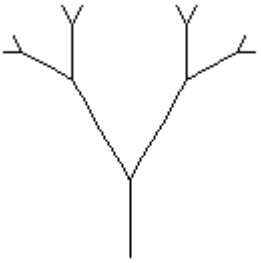
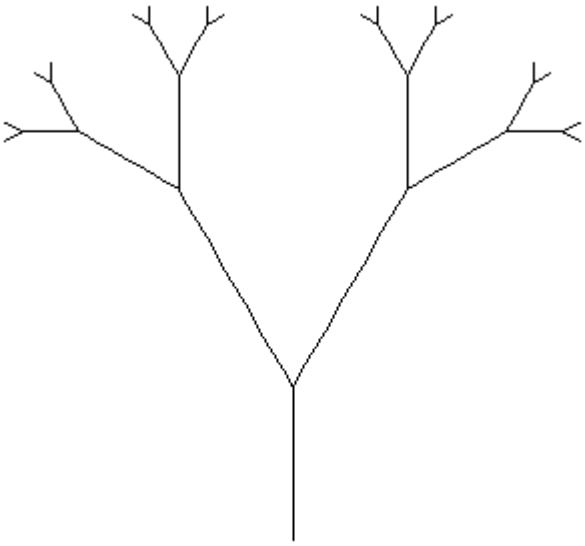
Powyższy L-system pozwala jedynie stworzyć graf, posiadający informacje o rodzaju i połączeniu węzłów, który pozbawiony jest jednak opisu ich przestrzennego ułożenia. Aby go uwzględnić, zmodyfikujemy poprzednią definicję L-systemów, wprowadzając kolejny element – stałe. Stałe to dodatkowe symbole niosące informację, które nie są poprzednikami żadnej z reguł produkcji. Automatycznie pojawia się też pojęcie zmiennych, które są symbolami podlegającymi podmianie. Na wstępie zakładamy, że kursor („żółw” – analogia do języka Logo) rysujący grafikę znajduje się u dołu obszaru rysowania i skierowany jest pionowo w górę. Dla zapewnienia absolutnie podstawowego rysowania w dwóch wymiarach potrzebujemy następujących symboli:

- +/- – skieruj żółwia w lewo/prawo o ustalony kąt.

- F – wykonaj krok z rysowaniem w obecnie ustalonym kierunku.
- [– zapisz na stosie obecny stan żółwia (pozycję, obrót i inne parametry, np. kolor).
-] – zdejmij ze stosu ostatni stan żółwia i przypisz mu go.

Posłużmy się następującym przykładem, wykorzystując nowe symbole:

- Stałe: + -
- Zmienne: F X
- Stan początkowy: X
- Reguły produkcji:
  - F → FF
  - X → F[+FX][-FX]

Iteracja	Rysunek	Słowo
1.		F[+FX][-FX]
2.		FF[+FFF[+FX][-FX]][-FFF[+FX][-FX]]
3.		FFFF[+FFFFFF[+FFF[+FX][-FX]][-FFF[+FX][-FX]]][-FFFFFF[+FFF[+FX][-FX]][-FFF[+FX][-FX]]]
4.		FFFFFFFF[+FFFFFFFFFFFF[+FFFFFF[+FFF[+FX][-FX]][-FFF[+FX][-FX]]][-FFFFFF[+FFF[+FX][-FX]][-FFF[+FX][-FX]]]][-FFFFFFFFFFFF[+FFFFFF[+FFF[+FX][-FX]][-FFF[+FX][-FX]]][-FFFFFF[+FFF[+FX][-FX]][-FFF[+FX][-FX]]]]

Na podstawie przedstawionych grafik można stwierdzić, że drzewo rozwija się w sposób oczekiwany i logiczny. Przykładowy L-system został stworzony w ten sposób, aby przy każdej iteracji drzewo zwiększało długości swoich konarów dwukrotnie. Gwarantuje to reguła produkcji F → FF – każde przesunięcie żółwia do przodu odpowiada dwóm przesunięciom w kolejnej iteracji. Druga reguła (X → F[+FX][-FX]) definiuje rozgałęzienia. Przedłuża ona istniejącą gałąź, a następnie tworzy rozgałęzienie, z którego powstają dwie nowe gałęzie. Na ich końcach reguła pozostawia wtedy symbol X, co oznacza, że rozgałęzienia w następnej iteracja dotyczyć będą tylko liści (rzadko kiedy drzewo rozgałęzia się wewnątrz).

Można zauważyć szybki wzrost skomplikowania słowa wraz z kolejnymi iteracjami. L-system generuje skomplikowany opis drzewa, lecz dzieje się to automatycznie i jedynym zadaniem programisty jest przetworzenie kolejno wszystkich symboli na odpowiednie operacje rysowania.

L-systemy niedeterministyczne

Do tej pory zestaw reguł i symboli był w pełni deterministyczny. Z jednej strony generowało to ładne, idealnie symetryczne drzewa, lecz z drugiej – były one pozbawione realizmu. W świecie natury losować jest rzeczą naturalną i pożądaną. Wprowadzenie jej do L-systemów jest prostym zadaniem.

Losowość na poziomie symboli można osiągnąć dzięki sparametryzowaniu niektórych symboli. Na przykład wcześniej wykorzystywane symbole obrotu o stały kąt (+ i -) można zastąpić funkcją $\propto(x)$, gdzie x będzie sumą pewnego kąta i czynnika losowego. Podobnie można postąpić z symbolem F, zamieniając go na $F(x)$, co zaowocuje powstawaniem gałęzi o losowych długościach.

Ciekawym rozwiązaniem jest wprowadzenie losowości na poziomie reguł. Należy zdefiniować szereg alternatywnych reguł o wspólnym poprzedniku i określić stopień prawdopodobieństwa ich zajścia. Należy przy tym zadbać, aby wśród danych poprzedników dokładnie jedna reguła została wylosowana. Przykładowo można zdefiniować je w następujący sposób:

- F → FF {0.9}
- X → F[+FX][-FX] {0.5}
- X → F-[[X]+X]+F[+FX] {0.3}
- X → F[+F][-F]X {0.2}

W ten sposób można tworzyć o wiele bardziej zróżnicowane drzewa, składające się z fragmentów o odmiennej budowie. Najbardziej realistyczne efekty uzyska się łącząc obydwie metody, z zachowaniem balansu pomiędzy losowością a regularnością drzewa.

Bibliografia:
P. Prusinkiewicz, A. Lindenmayer: *The algorithmic beauty of plants*
P. Prusinkiewicz, J. Hanan, R. Mech: *An L-system-based plant modeling language*
A. Spicher, O. Michel: *Declarative modeling of a neurulation-like process*

Rozdział III – Obiektowe systemy Lindenmayera

Klasyczne podejście.

W poprzednim rozdziale przedstawione zostało klasyczne podejście do L-systemów. Operuje ono na znakach (tekście), więc jest ono dość dobre do zademonstrowania zasady ich działania oraz zaprezentowania wyników.

Należy jednak zauważyć, że wykorzystanie znaków do reprezentacji alfabetu L-systemu (zwanych dalej symbolami) narzuca na użytkownika spore ograniczenia oraz obowiązki. Aby ograniczyć długość zdania, tekstowe symbole powinny być jak najkrótsze, ale przez to mogą być trudne do zapamiętania.

Załóżmy, że tworzymy L-system, który posłużyło rysowania drzew składających się z linii w 3 wymiarach. Wedle klasycznego podejścia programista potrzebuje zaimplementować, a użytkownik zapamiętać następujących symbole:

+ obrót w lewo o stały kąt wokół osi y

− obrót w prawo o stały kąt osi y

& obrót w lewo o stały kąt wokół osi x

^ obrót w prawo o stały kąt wokół osi x

\ obrót w lewo o stały kąt wokół osi z

/ obrót w prawo o stały kąt wokół osi z

F narysowanie segmentu o stałej długości

Dodatkowo symbole w takiej formie nie są w żaden sposób sparametryzowane – korzystają z domyślnych wartości kąta i kroku. W praktyce kąt definiuje się jako małą wartość, np. 15 stopni, a jego uzyskanie np. 60 stopni sprowadza się do powtórzenia symbolu czterokrotnie. Prowadzi to do znacznego wydłużenia reguł produkcji i zdań wynikowych, gdzie powtórzone symbole przytłaczają swoją ilością. Rozwiązaniem tego jest parametryzacje, którą można wprowadzić poprzez dodanie do symboli argumentów, np. +(60) zamiast ++++. Nie zwalania to jednak z pamiętania znaczeń symboli.

Podejście obiektowe

Program opisywany w tej pracy wykorzystuje podejście obiektowe, które jest rozszerzeniem klasycznych systemów Lindenmayera. Jego celem jest redukcja ilości symboli, ich uogólnienie i parametryzacja.

Powyższy przykład redukuje się do następujących symboli:

- *Angle*(axis, a) – dodaje do aktualnego kąta kursowa wartość a wokół odpowiedniej osi axis
- *Forward*(f) – powoduje dodanie nowego segmentu drzewa oraz przesunięcie kursora na jego koniec.

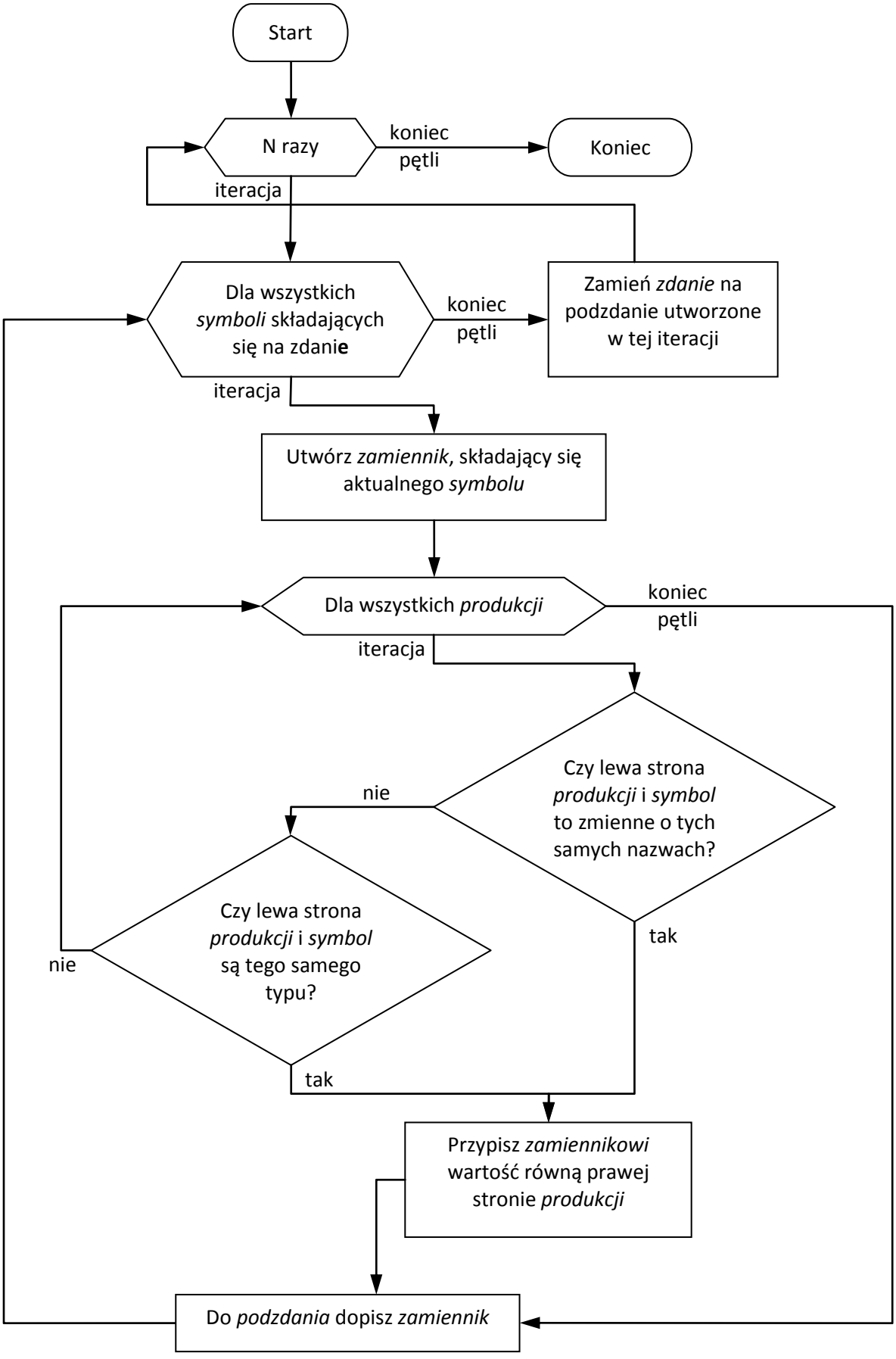
Dodatkowo obiektowy L-System wprowadza pozostałe niezbędne symbole:

- *Axiom* – występuje jedynie w roli poprzednika jednej z reguł produkcji. Oznacza on, że jej prawa część jest aksjomatem
- *Push* – powoduje zapisanie bieżącego stanu na stosie. Stan to aktualny kąt oraz skala
- *Pop* – powoduje zdjęcie ostatniego stanu ze stosu i nadpisanie nim bieżącego stanu
- *Scale*(n) – informacja dla parsera (omówiony dalej), że długość kolejnych gałęzi należy pomnożyć przez iloczyn obecnego współczynnika skali i *n*
- *Var*(s) – występuje w dwóch funkcjach. Jako poprzednik określa sekwencję, która będzie służyła do podmiany. Jako następnik oznacza miejsce, w którym należy dokonać podmiany. S jest nazwą tej zmiennej, dzięki czemu można używać wiele różnych zmiennych o intuicyjnych nazwach, np. „pień” albo „gałąź”

Tak więc w programie zaimplementowane są klasy LSymAngle, LSymForward, LSymAxiom, LSymPush, LSymPop, LSymScale i LSymVar, które biorą udział w budowie L-systemu.

Algorytm obliczania L-systemu

Podstawowa idea algorytmu iteracji L-systemu pozostaje niezmieniona – opiera się on na zamienianiu kolejnych elementów zdania na podstawie zdefiniowanych reguł. Pojawia się jedynie nowy blok odpowiedzialny za porównywanie zmiennych.


Start
N razy
koniec pętli
Koniec
iteracja
Dla wszystkich *symboli* składających się na zdanie
koniec pętli
Zamień *zdanie* na podzdanie utworzone w tej iteracji
iteracja
Utwórz *zamiennik*, składający się aktualnego *symbolu*
Dla wszystkich *produkcji*
koniec pętli
iteracja
Czy lewa strona *produkcji* i *symbol* to zmienne o tych samych nazwach?
nie
tak
Czy lewa strona *produkcji* i *symbol* są tego samego typu?
nie
tak
Przypisz *zamiennikowi* wartość równą prawej stronie *produkcji*
Do *podzdania* dopisz *zamiennik*

Schemat blokowy wykorzystuje pętle *foreach* dla poprawy czytelności. Ma to dodatkowe uzasadnienie w tym, że liczniki pętli nie są nigdzie wykorzystywane.

Algorytm przetwarza każdy symbol składający się na aktualne zdanie L-systemu. Przy pierwszej iteracji jest to zawsze aksjomat (stan początkowy). Tenże symbol poddawany jest próbie podmiany. Jeśli zakończy się ona powodzeniem (będą reguły produkcji, których lewa strona będzie tym symbolem), to zostanie on zamieniony na prawą stronę danej produkcji. Jeśli zamiana zakończy się niepowodzeniem, symbol pozostanie bez zmian. Niezależnie od powodzenia, podmieniony lub nie symbol zostanie dopisany do podzdania, czyli wyniku wszystkich podmian w danej iteracji.

Symbole są porównywane po ich *rolach*. Klasa, po której dziedziczą wszystkie symbole definiuje stosowne pole i wymaga jego wypełnienia już w konstruktorze. W przypadku zmiennych ich role są identyczne, więc należy sprawdzić dodatkowy parametr – nazwę zmiennej, stąd drugi blok warunkowy na schemacie.

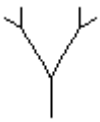
Po zakończeniu przetwarzania wszystkich symboli w pętli, aktualne zdanie L-systemu jest zamieniane na uprzednio wygenerowane podzdanie i cały proces zaczyna się od nowa. Po wykonaniu wszystkich iteracji, L-system zawiera zdanie składające się z symboli, które w dalszej kolejności posłużą do zbudowania drzewa (struktury danych).

Konwersja zdania L-systemu do grafu

Płynna animacja rozwoju graficznej reprezentacji L-systemu jest zadaniem trudnym. Najprostszym rozwiązaniem jest przycięcie gotowego zdania do pewnej długości, która będzie się zwiększać wraz z postępem animacji. W ten sposób będzie generowana coraz większa część ostatecznej formy drzewa. Zaletą tego podejścia jest brak potrzeby stosowania przejściowych struktur danych – rysujemy „na bieżąco”, posługując się jedynie zasadami grafiki żółwia. Wadą jest nierealistyczny wzrost, co zostanie zademonstrowane poniżej.

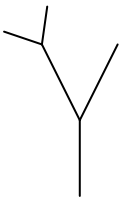
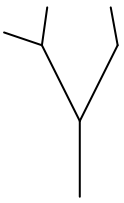
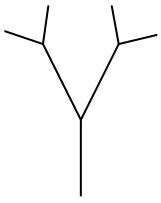
Wykorzystamy uproszczoną wersję L-systemu, omówionego w poprzednim rozdziale

- Stałe: + -
- Zmienne: X
- Stan początkowy: X
- Reguły produkcji:
  - X → F[+FX][-FX]

Iteracja	Rysunek	Słowo
1.		F[+FX][-FX]
2.		F[+F[+FX][-FX]][-F[+FX][-FX]]

Druga iteracja jest odpowiednio złożona, aby przedstawić problem tego podejścia, a jednocześnie nie sprawi, że pogubimy się w symbolach.

Teraz zbadajmy efekt obcinania słowa. Kolejne iteracje zostały dobrane tak, aby ilość kroków rysujących żółwia („F”) była zawsze o 1 większa w każdym następnym stanie.

Długość słowa	Ilość kroków żółwia	Słowo	Rysunek
1	1	F	
4	2	F[+F	
7	3	F[+F[+F	
12	4	F[+F[+FX][-F	
18	5	F[+F[+FX][-FX]][-F	
21	6	F[+F[+FX][-FX]][-F[+F	
29	7	F[+F[+FX][-FX]][-F[+FX][-FX]]	

Z obserwacji płyną następujące wnioski:

- Ścieżka wzrostu jest tożsama ze ścieżką odwiedzania węzłów przez algorytm przeszukiwania w głąb
- Ten typ wzrostu nie odpowiada wzrostowi roślin, obserwowanym w naturze
- Dobranie odpowiedniej długości słowa dla dowolnego L-sytemu jest zadaniem trudnym. Należy odnajdywać kolejne kroki rysujące („F”) i przycinać zdanie tuż po nich
- Po zakończeniu rysowania stos stanów żółwia może nie być pusty

Remedium okazuje się użycie formy przejściowej – grafu (a konkretnie drzewa, lecz aby uniknąć dwuznaczności tego słowa w kontekście tejże pracy, w dalszej części będzie mowa o grafie). Zdanie L-systemu zostaje zinterpretowane do postaci grafu, którego węzłami są informacje dotyczące gałęzi mających początek w danym miejscu.

Aby opisać daną gałąź, potrzebujemy kilku podstawowych informacji, takich jak:

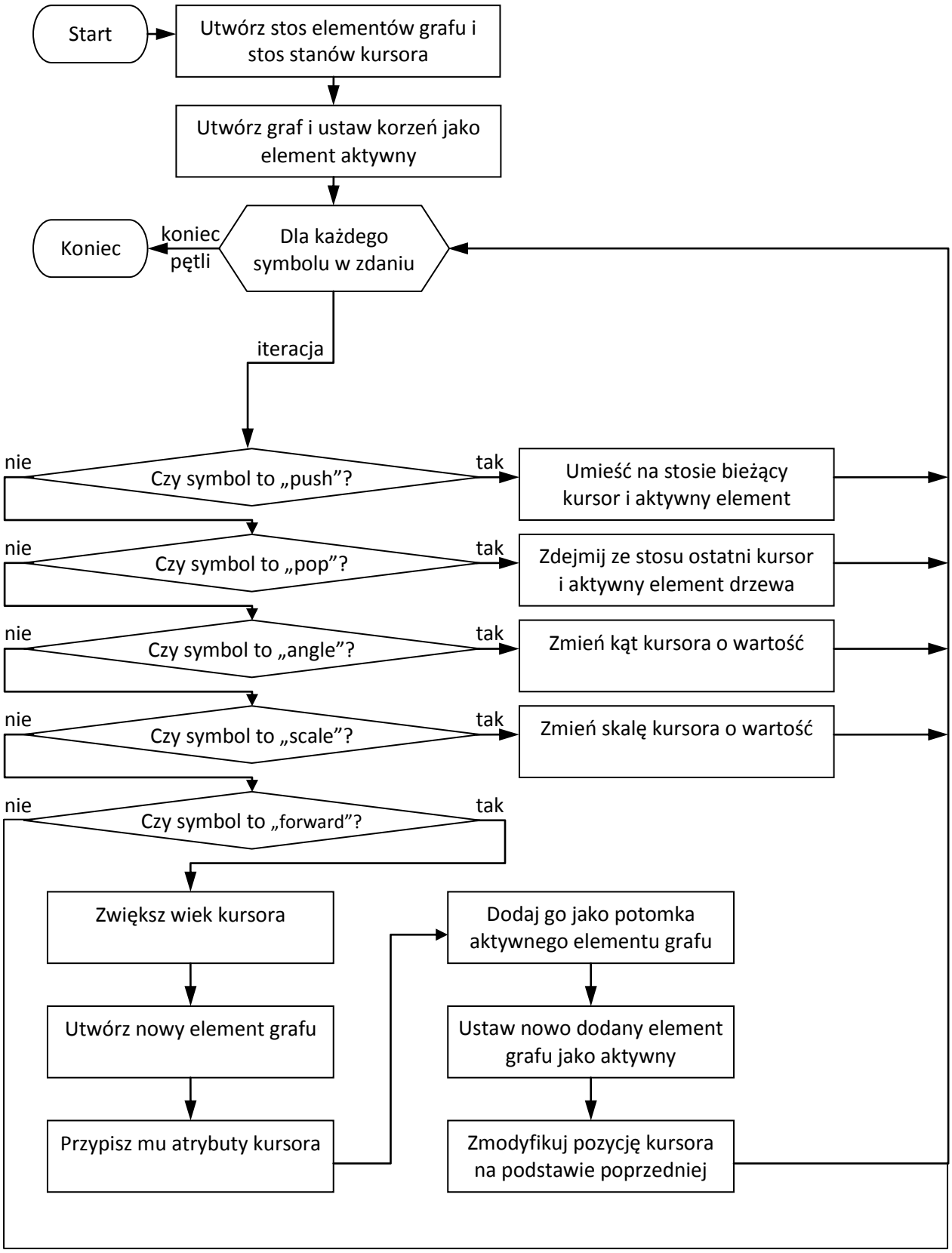
- Początek (współrzędne 3D)
- Rotacja (kwaternion)
- Długość (skalar)
- Skala (skalar)
- Generacja – odgrywa kluczową rolę w animacji. Określa wiek danej gałęzi. W trakcie animacji wzrostu drzewa wyświetla się te gałęzie, które nie są starsze niż zadany wiek.

Procedura zamiany zdania L-systemu w graf imituje działanie żółwia. Utrzymywane są dwa stosy: dla kursora oraz dla drzewa. Tworzony jest pusty graf drzewa. Inicjalizowany jest wirtualny kursor, który przechowuje atrybuty takie, jak definicja gałęzi. Początkowo wskazuje on „w górę", ponieważ jest to kierunek wzrostu drzewa. Stan kursora jest modyfikowany poprzez symbole L-systemu – np. „angle" zmienia wartość jego kąta.
Najważniejsza logika kryje się w symbolu „forward". Powoduje on:

- Dopisanie kolejnego elementu do listy potomków elementu aktywnego i uczynienie tego elementu elementem aktywnym. Zapewnia to ciągłość łańcucha poprzednik-następnik. (obsługa grafu)
- Zmodyfikowanie pozycji kursora na podstawie jego atrybutów tak, aby znajdował się on w tym miejscu, gdzie w następnej iteracji będzie umieszczony kolejny element. Kursor przesuwany jest o wektor o długości ostatnio stworzonego elementu grafu przemnożonej przez skalę, obrócony o aktualną wartość kąta. (obsługa kursora)

Jest to odpowiednik rysowania w podejściu bezpośrednim. Jednakże zamiast rysować elementy drzewa, oblicza się ich atrybuty (pozycja, rotacja itd.) i zapisuje do grafu, który obrazuje połączenia między nimi. Taka postać ułatwi potem animację, czyli rysowanie drzewa „do pewnego stopnia".

Algorytm tworzenia grafu atrybutów gałęzi



Animacja grafu drzewa

Wzrost drzewa można uogólnić do corocznego pojawiania się nowych pączków na końcach istniejących gałęzi. Pączki te rozwijają się, przeobrażając się w gałęzie, przez co same stają się początkiem kolejnych pączków. Wprowadzone zostaje pojęcie generacji i wieku. Generacja to liczba całkowita, oznaczająca rok od początku istnienia drzewa, w którym nastąpiło pączkowanie, które utworzyło daną gałąź. Natomiast wiek to liczba rzeczywista, określająca z ilu generacji gałęzi składa się w danym momencie drzewo. Łatwo zauważyć, że część całkowita wieku drzewa określa generację jego najmłodszych gałęzi.

Animacja drzewa będzie polegać na płynnym zwiększaniu wieku drzewa, na podstawie którego będzie określana liczba generacji do narysowania. Najmłodsza generacja gałęzi będzie przycinana do długości $(\text{wiek} - \lfloor\text{wiek}\rfloor) \cdot 100\%$ długości docelowej.

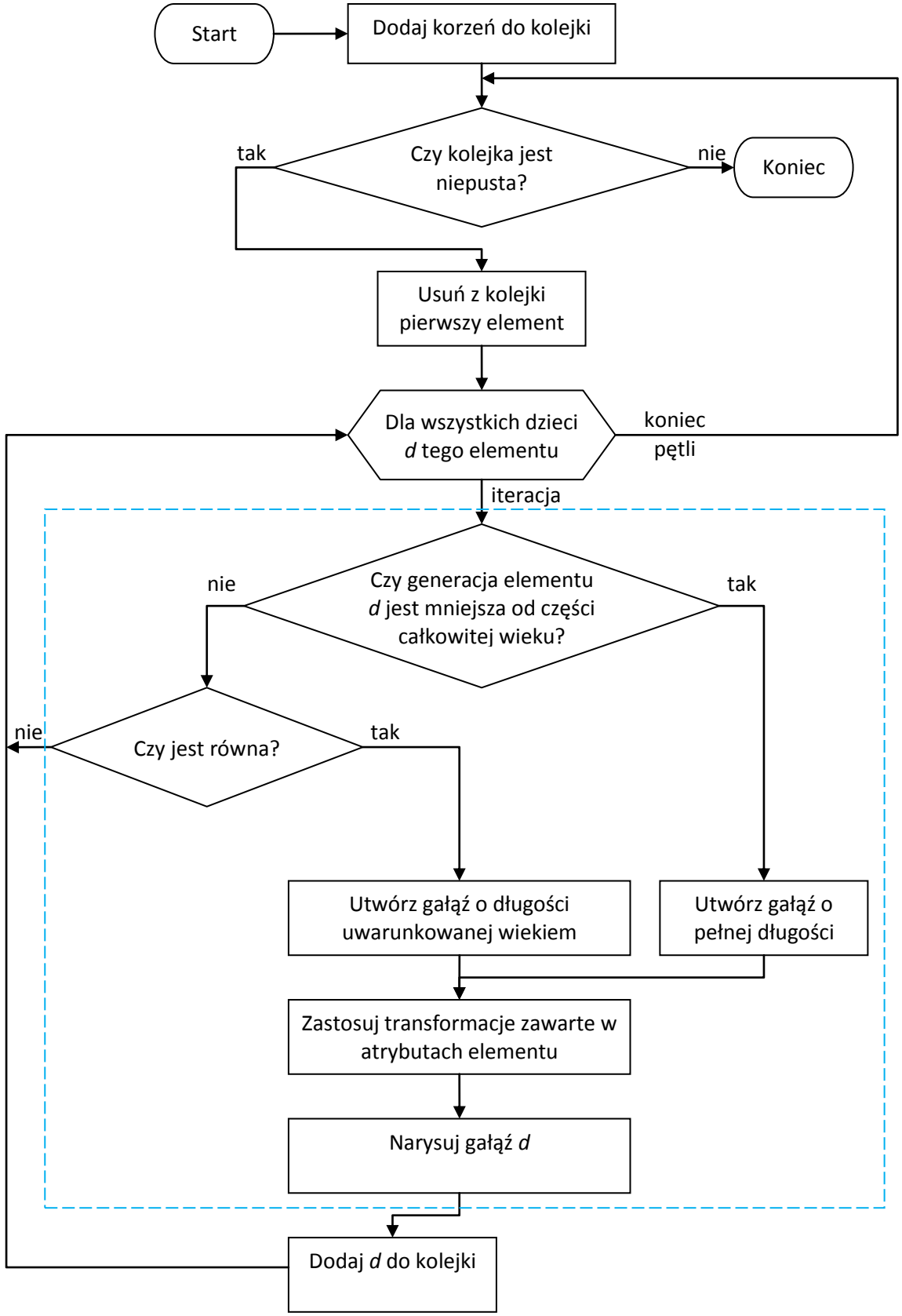
Dzięki przechowywaniu łańcucha powiązań rodzic-dzieci, animacja drzewa sprowadza się do odfiltrowania grafu na podstawie atrybutu wieku jego węzłów. Wystarczy, że dowolny algorytm przeszukujący (DFS lub BFS) odwiedzi wszystkie elementy grafu i narysuje jedynie te, które spełnią zadane kryterium wieku.

Poniżej przedstawiono schemat blokowy algorytmu BFS, w którym dokonuje się rysowanie elementów drzewa. Dla czytelności fragment odpowiedzialny za samo rysowanie zaznaczono niebieską ramką. Następuje tam podzielenie gałęzi drzewa na 3 kategorie, zależnie od ich wieku:

- Wystarczająco stare, aby mogły być narysowane w całości
- Odpowiednio stare, aby mogły być narysowane, ale zbyt młode, aby w całości. Gałęzie najmłodszej generacji, które należy przyciąć do $(\text{wiek} - \lfloor\text{wiek}\rfloor) \cdot 100\%$ długości docelowej.
- Gałęzie zbyt młode, aby mogły być narysowane

Do gałęzi, które mają zostać narysowane zastosowywane są absolutne transformacje (pozycja i rotacja), zawarte w węzłach.

Płynność animacji zapewniona jest przez płynne wydłużanie gałęzi w najmłodszej generacji.


Start
Dodaj korzeń do kolejki
Czy kolejka jest niepusta?
tak
nie
Koniec
Usuń z kolejki pierwszy element
Dla wszystkich dzieci d tego elementu
koniec pętli
iteracja
Czy generacja elementu d jest mniejsza od części całkowitej wieku?
nie
tak
Czy jest równa?
nie
tak
Utwórz gałąź o długości uwarunkowanej wiekiem
Utwórz gałąź o pełnej długości
Zastosuj transformacje zawarte w atrybutach elementu
Narysuj gałąź d
Dodaj d do kolejki

Rozdział IV – OpenTK i renderowanie w czasie rzeczywistym

Do tej pory mowa była zawsze o „rysowaniu”, lecz nigdy nie sprecyzowano jak ma się ono odbywać. Celem tego rozdziału jest omówienie procesu przetwarzania informacji zawartych w węzłach grafu wygenerowanego drzewa na trójwymiarowe obiekty.

Jako podsystemu graficznego użyto OpenGL z uwagi na jego popularność i dobrą dokumentację. Jest kilka bibliotek, które umożliwiają korzystanie z OpenGL z poziomu języka C#, ale na uwagę zasługuje pakiet OpenTK. Łączy on 3 podsystemy – OpenGL, OpenCL i OpenAL.

Zalety OpenTK

- Ścisła typizacja parametrów funkcji. Oryginalna implementacja OpenGLa jest stworzona w jednej wielkiej przestrzeni nazw, a większość funkcji przyjmuje w parametrach wartości typu integer. Konsekwencją tego jest współistnienie stałych o złudnie podobnych nazwach, które mogę zmylić programistę i zmuszają do ciągłego zerkania w dokumentację.

  Przykładem mogą być stałe GL_TEXTURE, GL_TEXTURE_2D, GL_TEXTURE0 – na początku ciężko zapamiętać którą z nich należy przekazać do funkcji glEnable, aby włączyć wyświetlanie tekstur. OpenTK rozwiązuje ten problem tworząc szereg enumeracji, przeznaczonych dla konkretnych funkcji. Np. funkcja Enable przyjmuje argumenty inne niż BindTexture.
- Wykorzystanie przestrzeni nazw do podziału funkcji zależnie od ich zadania. W pakiecie *jogl* albo *glut* wszystkie funkcje jak i stałe są w jednej przestrzeni nazw. Tworzy to listę kilkuset elementów o nazwach zaczynających się od „gl", niepogrupowanych w żaden sposób.

  OpenTK dzieli funkcje i enumeracje zależnie od przeznaczenia, umieszczając je w odpowiednich przestrzeniach nazw. Dzięki temu nie dochodzi do zanieczyszczenia domyślnej przestrzeni setkami elementów w żaden sposób nie związanych ze sobą.

- Przeciążanie funkcji. Zamiast definiować funkcje `glVertex3f`, `glVertex3d`, `glVertex3i`, `glVertex3fv`, `glVertex3dv`, `glVertex3iv`, definiuje się jedną – `Vertex3`. Rodzaj wywołania jest uzależniony od typu przykazywanych do niej parametrów.
- Gotowy zestaw typów matematycznych, używanych w grafice 3D – wektory, macierze, kwaterniony, oraz funkcje operujące na nich oraz między nimi, a także operatory. Funkcje są tak przeciążone, aby mogły np. przyjmować 3-elementowy wektor (Vector3) zamiast tablicy 3 liczb typu float.

Wadą, jak każdego programu stworzonego w kodzie zarządzanym jest powolność działania. Jednak przy odpowiednim zaprogramowaniu aplikacji, ma ona prawo działać bardzo wydajnie. Kluczową sprawą jest ograniczenie wywołań zarządzanych metod. OpenGL jest architekturą klient-serwer, gdzie klient to w tym przypadku zarządzany OpenTK, a serwer znajduje się na karcie graficznej i stanowi bardzo szybki i zoptymalizowany kod. Tak więc należy dążyć do sytuacji, gdzie większość operacji będzie odbywała się po stronie serwera – karty graficznej, a nie samego programu. OpenGL umożliwia takie działanie – np. poprzez użycie buforów wierzchołków zamiast trybu bezpośredniego.

Tryby wyświetlania geometrii w OpenGL

Istnieje kilka sposobów przesyłania prymitywów do karty graficznej. Pojęcie prymitywu jest stworzone wyłącznie dla wygody programisty, ponieważ OpenGL i tak w efekcie przekształca je na trójkąty.

1. Tryb natychmiastowy – wykorzystuje funkcję Vertex3 w celu przesłania do karty wierzchołka w dowolnym miejscu programu. Tryb nazywa się natychmiastowym, ponieważ nie występuje w nim buforowanie ani optymalizacja danych, a efekty są widoczne od razu na ekranie. Ta metoda jest akceptowalna w przypadku tworzenia niewielkich obiektów (np. układy współrzędne, manipulatory), ale potężnie ogranicza wydajność aplikacji przy renderowanie bardziej złożonych obiektów, gdyż przesłanie pojedynczego wierzchołka jest kosztowne czasowo.

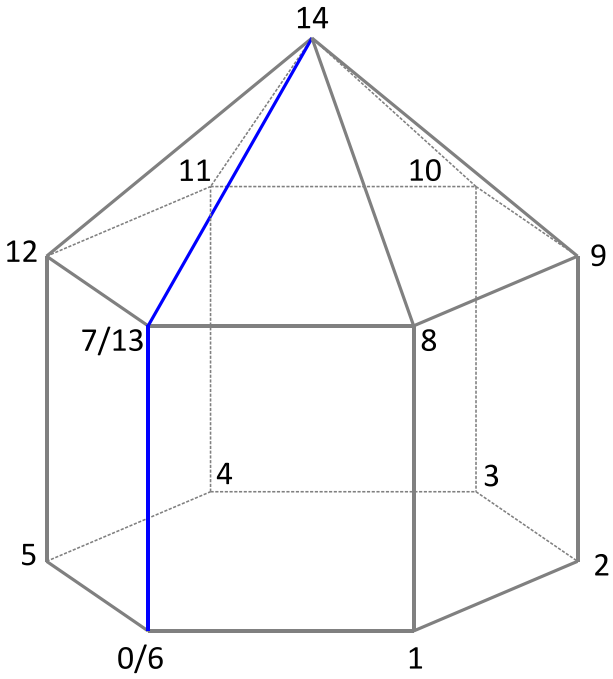
2. Listy wyświetlania – również wykorzystuje tryb bezpośredni, ale na czas tworzenia listy efekty komend trybu natychmiastowego nie trafiają na ekran, lecz są składowane w pamięci. Pozwala to automatycznie zoptymalizować listę poleceń. Np. z uwagi na to, że OpenGL jest maszyną stanu, możliwym okazuje się usunięcie niektórych poleceń, które nie zmieniają obecnego stanu. Lista wyświetlania jest kompilowana, co zmniejsza jej rozmiar i przesyłana za jednym zamachem na kartę graficzną. Eliminując potrzebę wielokrotnych wywołań funkcji Vertex3 znacznie poprawia się wydajność aplikacji. Minusem list wyświetlania jest brak możliwości ich modyfikacji po tym, jak już zostały skompilowane, tak więc nadają się one do obiektów statycznych, lub rzadko zmiennych.
3. Bufory wierzchołków – metoda łączy zalety obydwu poprzednich. Bufory wykorzystują bardzo szybką pamięć karty graficznej do składowania danych. Co więcej, dane te mogą być modyfikowane już po znalezieniu się na karcie graficznej. Jako, że wszystkie niezbędne informacje potrzebne do narysowania obiektu znajdują się już w pamięci karty, to narysowanie ich jest błyskawiczne i nie obciąża procesora.

Do zadania, jakim jest wyświetlanie dynamicznie rozwijającego się drzewa bufory wierzchołków są rozwiązaniem optymalnym. Na ich rzecz przemawia bardzo mały narzut czasu procesora, potrzebnego do przesłania wszystkich danych na kartę graficzną,

Cylinder jako podstawowy element drzewa

Patrząc na drzewo, wyróżnić można dwa podstawowe elementy – gałęzie i liście. Jako, że te drugie nie są przedmiotem tej pracy, skupimy się na gałęziach, ponieważ to one tworzą strukturę drzewa i są bezpośrednio odpowiedzialne za jego kształt. Gałęzie całkiem trafnie można przybliżyć za pomocą cylindrów, w których sparametryzowana jest wysokość oraz promień podstawy.

Jako, że biblioteka OpenTK nie ma wbudowanych funkcji typu *utility*, tworzących podstawowe bryły geometryczne, więc cylinder należy zaimplementować samodzielnie. Pozwala to na dużą dowolność przy wyborze topologii oraz pewnych cech charakterystycznych cylindra. Kształt bryły użytej w programie przedstawiony jest na poniższym rysunku.



Podstawowe cechy tej bryły:

- Składa się z bocznej powierzchni cylindra i stożka. Stożek sprawia, że zgięcia i rozgałęzienia wyglądają płynniej, niż w przypadku płaskiego zakończenia
- Ilość ścian aproksymujących krzywiznę może być dowolna.
- Bryła nie posiada dolnej podstawy. To miejsce jest zwykle wystarczająco zasłaniane przez stożek poprzedniego elementu

Pozycje

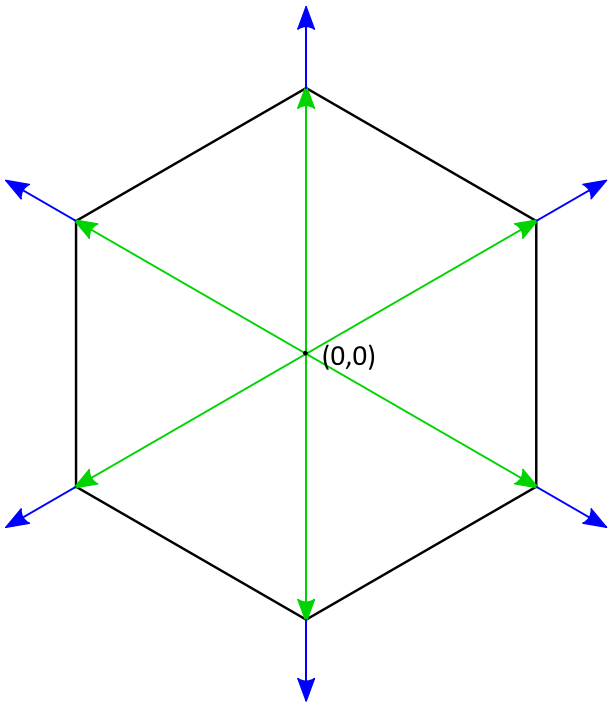
Budowę cylindra należy rozpocząć od współrzędnych punktów. Współrzędne kolejnych wierzchołków dolnej podstawy wyrażone są wzorem $\mathrm{v_i} = (\mathrm{rsin}(\alpha \mathrm{i}), 0, \mathrm{rcos}(\mathrm{ai}))$, gdzie *i* jest numerem wierzchołka, a $\alpha = \frac{2\pi}{\text{ilość ścian}}$ czyli krok w radianach. Wierzchołki należące do niebieskiej krawędzi są zdublowane, aby umożliwić późniejsze kafelkowanie tekstury.

Współrzędne górnej podstawy to współrzędne dolnej, ze zmienioną współrzędną *y*, która jest równa wysokości cylindra. Wykorzystanie tego faktu pozwala nie obliczać ponownie wartości funkcji trygonometrycznych.

Współrzędna wierzchołka tworzącego stożek wynosi $(0, h + r, 0)$. Wysokość *h+r* została dobrana empirycznie.

Normalne

W przypadku cylindra obliczenie wartości normalnych jest banalnym zadaniem, ponieważ są one równe znormalizowanym pozycjom wierzchołków, tzn. $n_v = \left(\frac{v_x}{|v|}, 0, \frac{v_z}{|v|}\right)$. Łatwo to zauważyć na poniższym rysunku.

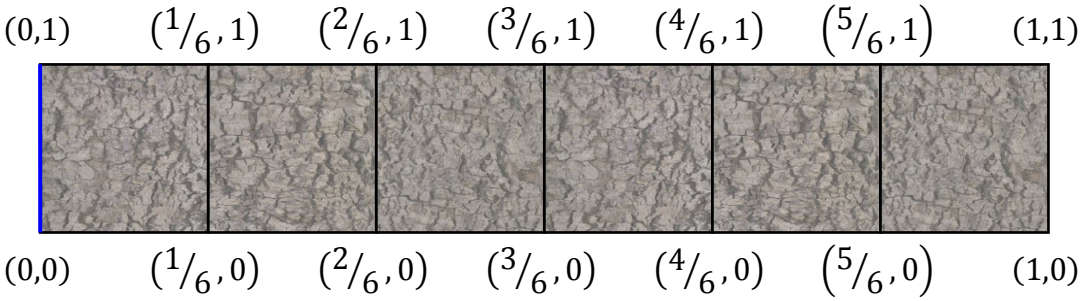


Wektory zielone to pozycje wierzchołków (w założeniu, że środek cylindra to (0,0)), które po normalizacji stają się normalnymi, reprezentowanymi przez kolor niebieski.

Jako normalną czubka stożka przyjęto wektor $(0,1,0)$.

Współrzędne teksturowania

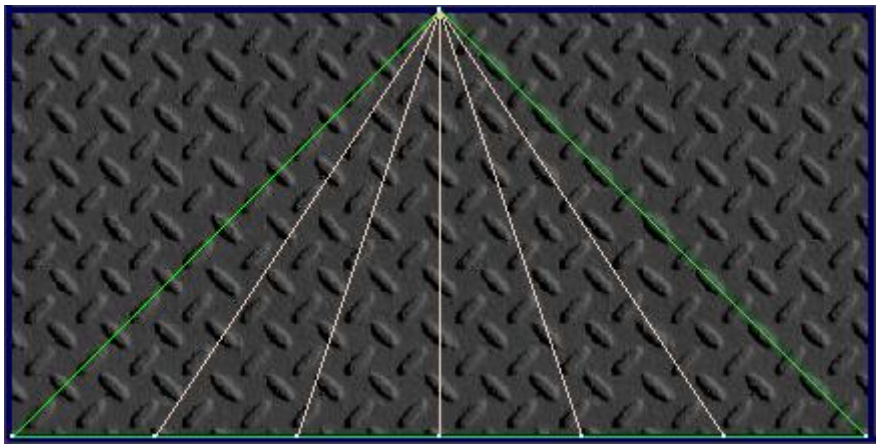
Obliczenie koordynatów UV jest nieco trudniejszym zadaniem. Najlepsze podejście to przeskakiwanie raz po górnej, a raz po dolnej podstawie, np. 0,7,1,8,2,9… Jeśli chcemy, aby tekstura była w pełni rozciągnięta o wysokości cylindra, oraz owijała go przez całą swoją długość, do każdego wierzchołka musimy przypisać współrzędne w następujący sposób:



Współrzędne UV mogą mieć dowolne wartości, ale w przedziale 0-1 odpowiadają dokładnie rozmiarowi tekstury. Ma być ona jednokrotnie powtórzona na obwodzie cylindra, więc składowa U będzie wynosić kolejno $^0/_6, ^1/_6, ^2/_6, ^3/_6, ^4/_6, ^5/_6, ^6/_6$. Natomiast rozciągnięcie tekstury w pionie zapewni ustawienie składowej V wierzchołków dolnego wiersza na 0, a górnego na 1.

Teraz widać dlaczego należało zdublować wierzchołki znajdujące się na niebieskiej linii. Bez tego współrzędna U była by interpolowana z $^5/_6$ do 0, co skutkowało by zniekształceniem tekstury na ostatniej ścianie cylindra. Wprowadzenie dodatkowych 2 wierzchołków o współrzędnej U równej 1 pozwala na prawidłowe owinięcie tekstury dookoła cylindra.

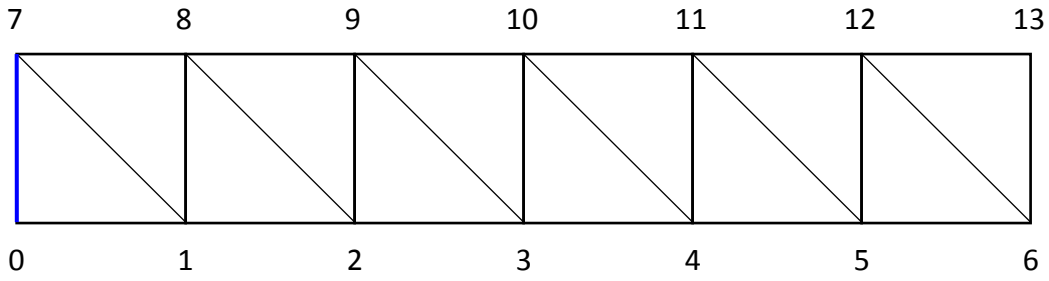
Pozostaje jeszcze kwestia stożka. Zauważmy, że wszystkie oprócz jednego wierzchołka są już zmapowane (należą do cylindra). Współrzędne samego czubka wyrażają się wzorem $(^1/_2, 0)$.



Wygląd takiego mapowania można podejrzeć w modyfikatorze Unwrap w programie 3DS Max. Jest to pewien kompromis, ponieważ nie można określić takich współrzędnych, aby tekstura na stożku była mapowana bez deformacji. Aby to osiągnąć, trzeba by zwielokrotnić wierzchołek czubka stożka tak, aby każdy z nich miał własne współrzędne UV. Jednakże idealne mapowanie na stożku nie jest tak ważne, ponieważ i tak w większości przypadków stożek będzie zakrywany przez początek następnej gałęzi.

Indeksy

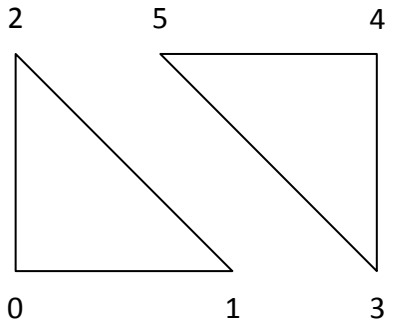
Ponieważ indeksowanie wierzchołków jest najbardziej efektywnym sposobem wyświetlania obiektów składających się z wielu ścian, przypadku cylindra trzeba będzie obliczyć indeksy. Pomoże w tym poniższy rysunek.



Zaindeksowanie wierzchołków w taki sposób pozwoli wygenerować indeksy za pomocą 6 równań dla każdej ściany. Prześledźmy ich układ dla pierwszych czterech trójkątów: 017 187 128 298. Widać, że wierzchołki pojawiają się trzykrotnie, więc zysk z postaci indeksowej będzie znaczny. Dodatkowo pozwala to wywnioskować poniższe równania.

Najprościej będzie się poruszać co jedną ścianę, czyli dwa trójkąty. Daje to 6 równań:

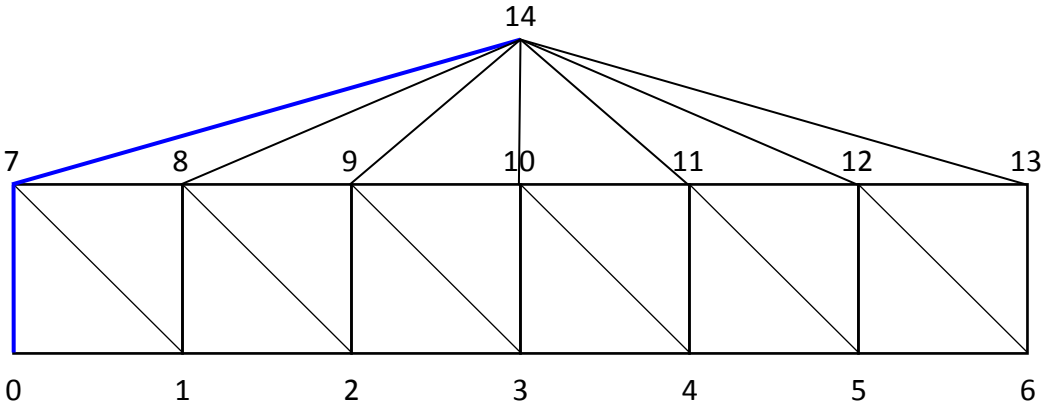
$$\text{indeks}_0 = \text{i}$$
$$\text{indeks}_1 = \text{i} + 1$$
$$\text{indeks}_2 = \text{i} + \text{n} + 1$$
$$\text{indeks}_3 = \text{i} + 1$$
$$\text{indeks}_4 = \text{i} + \text{n} + 2$$
$$\text{indeks}_5 = \text{i} + \text{n} + 2$$



Kolejność rysowania	Numer wierzchołka
0	0
1	1
2	7
3	1
4	8
5	7

(gdzie *n* = ilość ścian, *i* = numer ściany, liczony od 0)

Pozostaje jeszcze kwestia stożka. Na szczęście nie jest to trudne zadanie. Trzeba wygenerować tyle trójkątów, ile jest ścian cylindrów.



Korzystając z ilustracji, łatwo można obliczyć indeksy dwóch pierwszych trójkątów stożka: 8 14 7, 9 14 8. Widać prostą zależność, którą można wyrazić wzorem:

$\text{indeks}_0 = n + 2$

$\text{indeks}_1 = w$

$\text{indeks}_2 = n + 1$

(gdzie *n* to ilość ścian, *w* to numer wierzchołka odpowiadającego czubkowi)

Jeśli powtórzymy ją dla każdej ściany, otrzymamy indeksy wierzchołków tworzących powierzchnię stożka.

Podsumowanie

Dla 6-bocznej bryły mamy:

- 2 × 6 + 2 + 1 = 15 wierzchołków (6 2-wierzchołkowych krawędzi + jedna krawędź zdublowana + 1 wierzchołek jako czubek stożka)
- Tyle samo normalnych
- Tyle samo współrzędnych UV
- 6 × 6 + 6 × 3 = 54 indeksów (po 6 indeksów na każdą ścianę cylindra + po 3 indeksy na każdą ścianę stożka)

Przechowując indeksy w `uint`, a resztę danych we `float`, taka bryła zajmie

(15 × 3 × 4) + (15 × 3 × 4) + (15 × 2 × 4) + (54 × 4) = 696 bajtów.

Renderowanie cylindrów

W przypadku drzewa składającego się z kilkuset cylindrów, każdorazowe przesyłanie ich danych osobno byłoby bardzo nieefektywne. Karta graficzna preferuje duże porcje danych, a najlepiej umieszczone w jej własnej pamięci. Dlatego w tym przypadku najwydajniejszym rozwiązaniem będzie pobranie listy wszystkich wierzchołków w drzewie oraz wszystkich indeksów i przesłanie ich za jednym razem na kartę, celem wyrenderowania.

Lista wierzchołków i indeksów jest tworzona poprzez doklejanie do niej tablic wierzchołków i indeksów, pochodzących z poszczególnych cylindrów. W przypadku indeksów należy odpowiednio je zwiększyć, aby uniknąć nakładania ich na siebie. W praktyce każdy cylinder ma taką samą listę indeksów, co po sklejeniu w jedną wielką listę powodowało by indeksowanie jedynie wierzchołków pochodzących od pierwszego cylindra. Tak więc przy każdorazowym doklejeniu indeksów należy ich wartości zwiększyć o obecną ilość wierzchołków na liście.

W efekcie otrzymujemy dwie listy: jedną z wierzchołkami, drugą ze zmodyfikowanymi indeksami. Teraz wystarczy przesłać je na kartę graficzną. W OpenTK ten mechanizm działa analogicznie jak w OpenGL:

1. Na początku tworzone są dwa bufory: indeksów (elementów) i wierzchołków

```
vboID = new uint[2];
GL.GenBuffers(2, vboID);
```

2. Następuje wysłanie indeksów

```
GL.BindBuffer(BufferTarget.ElementArrayBuffer, vboID[1]);
GL.BufferData(BufferTarget.ElementArrayBuffer, (IntPtr)(indices.Count *
sizeof(uint)), indices.ToArray(), BufferUsageHint.StaticDraw);
```

3. Następuje wysłanie wierzchołków

```
GL.BindBuffer(BufferTarget.ArrayBuffer, vboID[0]);
GL.BufferData(BufferTarget.ArrayBuffer, (IntPtr)(vertices.Count * vertexSize),
vertices.ToArray(), BufferUsageHint.StaticDraw);
```

4. Ustawiane są wskaźniki tablic oraz włączane tryby tablic wierzchołków

```
GL.InterleavedArrays(InterleavedArrayFormat.T2fN3fV3f, 0, IntPtr.Zero)
```

5. Następuje narysowanie trójkątów na podstawie wysyłanych danych

```
GL.DrawElements(BeginMode.Triangles, indices.Count,
DrawElementsType.UnsignedInt, 0);
```

Warto zwrócić uwagę na funkcję `InterleavedArrays`. Wykonuje ona szereg czynności związanych z wyświetlaniem tablic wierzchołków:

- Włącza tablice wierzchołków, normalnych i współrzędnych teksturowania
- Ustawia wskaźniki do tychże na podstawie pierwszego parametru

Zapis `T2fN3fV3f` oznacza sposób składowania danych wierzchołka w pamięci – 2×`float`(UV) + 3×`float`(normalne) + 3×`float`(pozycje). Odpowiada to strukturze

```
struct Vertex {
	public Vector2 TexCoord;
	public Vector3 Normal;
	public Vector3 Position;
}.
```

Zastępuje to 3 polecenia: `TexCoordPointer`, `NormalPointer` i `VertexPointer`, wraz z offsetami w bajtach, które należało by im przekazać.

Funkcja `DrawElements` rysuje trójkąty, ponieważ poza quadami jest to jedyna topologia, która zezwala tworzenie osobnych powierzchni. W przypadku cylindrów najbardziej optymalne i intuicyjne było by użycie quad stripa dla boków, a triangle fana dla stożka. Jednakże te topologie sprawdzą się wyłącznie przy rysowaniu jednego cylindra. W naszym przypadku dane wszystkich cylindrów przechowywane są w postaci liniowej, co wymusza oddzielanie ich co pewien czas, aby cylindry nie zlały się w jedną bryłę. Ponieważ bufory wierzchołków nie przyjmują komend Begin i End, jedynym rozwiązaniem jest rysowanie niepołączonych prymitywów.

Rozdział V – eksport do formatu COLLADA

Aby wyniki pracy programu były w jakiś sposób użyteczne, należy wprowadzić możliwość eksportu uzyskanej geometrii do pliku tak, aby potem można było ją wykorzystać np. w scenie 3D w dowolnym popularnym programie do grafiki trójwymiarowej.

Format COLLADA

COLLADA to format pośredniczący, oparty na XMLu, stworzony z myślą o wymianie assetów (czyli modeli, tekstur, shaderów) pomiędzy różnymi narzędziami typu DCC (digital content creation). Jest on wspierany przez popularne programy jak np. Autodesk 3DS Max, Autodesk Maya, Google Sketchup, jak również silniki 3D: CryEngine2, Unreal i Unity. Ponieważ plik .dae jest dokumentem XML zgodnym z pewną XML Schemą, jego implementacja jest bardzo prosta i automatyczna. W przypadku języka C# procedura wygląda następująco:

1. Należy pobrać schemę ze strony http://www.khronos.org/collada/
2. Za pomocą polecenia XSD, dostępnego w pakiecie Visual Studio na podstawie schemy wygenerować odpowiednie klasy:
   ```
   xsd collada_schema_1_4.xsd /c
   ```
3. Powstaje plik collada_schema_1_4.cs, który należy dołączyć do projektu

Odczyt pliku .dae odbywa się za pomocą deserializacji pliku XML:

```
public static COLLADA Load(Stream stream) {
    StreamReader str = new StreamReader(stream);
    XmlSerializer xSerializer = new XmlSerializer(typeof(COLLADA));
    return (COLLADA)xSerializer.Deserialize(str);
}
```

Jak widać jest to bardzo proste i oszczędza mnóstwa ręcznego parsowania. Zapis, jak można się domyślić, odbywa się poprzez serializację obiektu typu COLLADA z odpowiednio wypełnionymi polami.

Taki mechanizm nie jest pozbawiony niedogodności. Klasy automatycznie wygenerowane przez narzędzie xsd narzucają trochę nieintuicyjne typy danych – np. jednoelementowe tablice, lub typ „object” tam, gdzie spodziewalibyśmy się bardziej

specjalizowanej klasy. Tak więc programowe tworzenie pliku .dae po raz pierwszy może być trudnym zadaniem. Na szczęście format COLLADA jest bardzo logicznie zbudowany, co pomaga odnaleźć się w gąszczu znaczników.

Uproszczona struktura formatu COLLADA

Format składa się z bibliotek, które są powiązane odniesieniami.

- Root (COLLADA)
    - asset
    - library_images
    - library_materials
    - library_effects
    - library_geometries
    - library_visual_scenes
    - scene

asset – zawiera informacje dotyczące zawartości pliku. Poza opcjonalnymi danymi takimi jak autor, data utworzenia lub narzędzie, które do tego posłużyło, znajduje się tam informacja na temat osi, która ma być interpretowana jako „góra”, ponieważ różne programy korzystają z różnych układów współrzędnych.

library_images – definiuje tekstury użyte w scenie – ich nazwę, identyfikator oraz ścieżkę, z której należy je zaimportować

library_materials – definiuje materiały. W najprostszej formie materiał wykorzystuje pewien efekt, zdefiniowany w odpowiedniej bibliotece

library_effects – definiuje efekty. Mogą być one rozumiano jako style powierzchni, lub shadery. Każdy efekt posiada profile, zawieracące technikę. Techniką może być np. blinn, phong lub lambert, czyli model cieniowania. Model jest wzbogacony o odpowiednie tekstury – powierzchni, odbić, wgłębień itd, lub informacje o kolorach, jeśli nie posiada tekstur.

library_geometries – definiuje obiekty 3D użyte w scenie. Obiekt może składać się z kilku siatek – meshy. Na siatkę typowego obiektu składają się pozycje wierzchołków, normalne, współrzędne teksturowania oraz indeksy mówiące jak połączyć wierzchołki w prymitywy.

library_visual_scenes – określa sceny dostępne w dokumencie. Zazwyczaj jest to jedna scena. Scena składa się z węzłów – node – które w naturalny sposób określają hierarchię obiektów. Każdy węzeł zawiera informacje na temat transformacji obiektu względem rodzica, użytej geometrii oraz materiału do niej przypisanego.

scene – wybiera aktywną scenę z library_visual_scenes, która ma być widoczna po załadowaniu pliku.

Zapisywanie modelu drzewa do pliku

Jak już wcześniej wspomniano, zapisanie pliku sprowadza się do serializacji obiektu typu COLLADA z odpowiednio wypełnionymi polami. Jednakże w przypadku tak rozbudowanego formatu, liczba tych pól oraz powiązań między nimi jest dość duża, dlatego warto prześledzić ten proces krok po kroku.

1. W bibliotece *library_geometries* zdefiniowanie nowego obiektu i jego siatki. Siatka ma sloty zwane źródłami (*sources*), przeznaczone dla różnych atrybutów wierzchołków.
    a. Stworzenie tablicy pozycji wierzchołów w formacie XYZXYZ…, normalnych w formacie XYZXYZ… oraz współrzędnych tekstur w formacie UVUV… i przypisanie ich do slotów. Ilość, przeplot i znaczenie kolejnych liczb definiują akcesory, opisujące te tablice.
    b. Zdefiniowanie sekcji *vertices*, która zawiera referencję do tablicy pozycji wierzchołków. Jest to wymagane przez parsery.
    c. Stworzenie sekcji poligonów
        i. Przypisanie nazwy materiału dla tej grupy poligonów.
        ii. Określenie uprzednio zdefiniowanych tablic jako źródeł danych do indeksowania przez indeksy oraz zdefiniowanie semantyk, informujących importujący program jakie jest znaczenie danych w tablicach – czy są to pozycje, normalne czy wsp. UV.

iii. Stworzenie listy wszystkich poligonów w siatce. Poligon może mieć dowolną ilość boków, ale w tym przypadku są to trójkąty. Trójkąt jest zdefiniowany przez 9 liczb – indeks **p**ozycji, **n**ormalnej i wsp. **U**V dla każdego jego wierzchołka. Są one przeplecione: PNUPNUPNU.

2. Stworzenie materiału określającego wygląd powierzchni
   a. W bibliotece *library_images* umieszczenie informacji o lokalizacji pliku tekstury.
   b. W bibliotece *library*_effects utworzenie nowego efektu
      i. Przypisanie mu standardowego profilu i techniki.
      ii. W technice wybranie shadera phonga.
      iii. W shaderze określenie tekstury dla składowej *diffuse* oraz koloru dla składowej *specular.*
   c. W bibliotece library*_materials* stworzenie nowego materiału, wykorzystującego stworzony uprzednio efekt.
3. Skonstruowanie sceny w bibliotece *library_visual*_scenes
   a. Stworzenie węzła określającego jedyny obiekt w scenie
      i. Nadanie mu zerowej transformacji w postaci zlinearyzowanej macierzy jednostkowej 4x4: 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1.
      ii. Zinstancjonowanie zdefiniowanej uprzednio geometrii.
      iii. Przypisanie materiałów do nazw zdefiniowanych w grupach poligonów.
4. Umieszczenie odwołania do tej sceny w sekcji *scene.*
5. Wypełnienie sekcji *asset* – kluczowe jest pole *up_axis,* natomiast reszta jest opcjonalna.

Pełen eksport wymaga jeszcze zapisania pliku tekstury pod ścieżką zdefiniowaną w *library_images.*

Warto zwrócić uwagę na to, że przechowywanie wszystkich cylindrów w postaci liniowej listy wierzchołków i indeksów jest bezpośrednio związane z eksportem. Do eksportera przekazuje się aktualny bufor wierzchołków i indeksów bezpośrednio z renderera, co czyni eksport bieżącego stanu drzewa bardzo prostym.

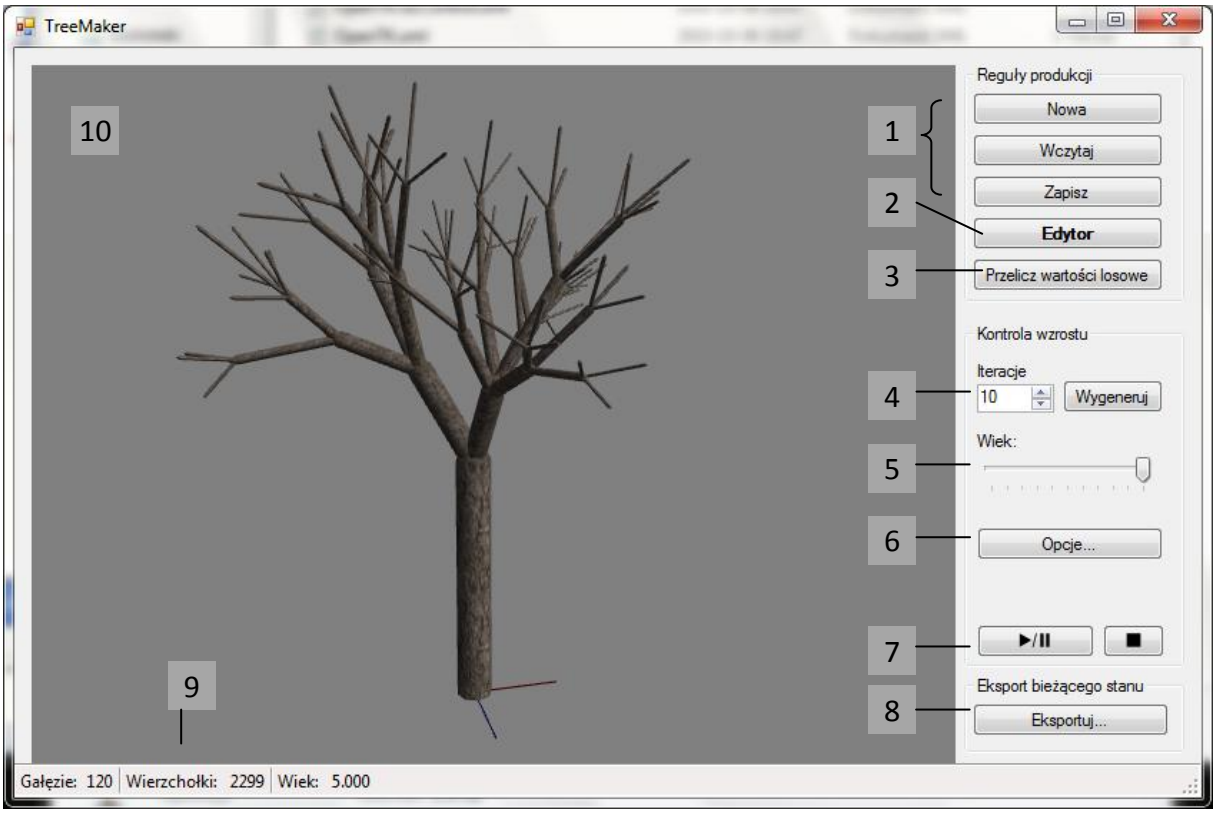
Rozdział VI – opis programu dyplomowego

Celem niniejszego rozdziału jest pokazanie dotychczasowo omówionej teorii w praktyce, w postaci programu generującego drzewa, zwanego później *TreeMaker*.

Główne cechy programu:

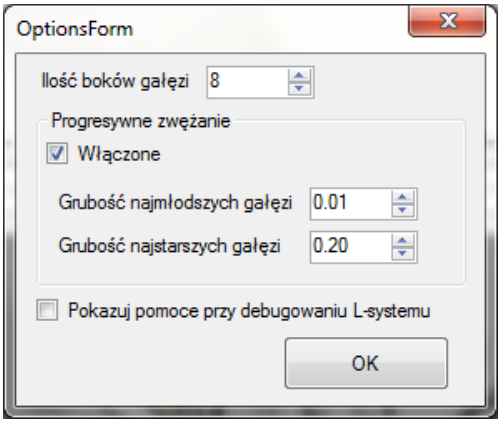
- Napisany w C#
- Wyświetlanie oparte na OpenGL, dzięki bibliotece OpenTK
- Oblicza drzewo i animację w czasie rzeczywistym
- Daje możliwość zmiany wariantu drzewa poprzez ponowne obliczenie wszystkich wartości losowych
- Umożliwia edycję drzewa za pomocą graficznego edytora produkcji
- Reguły produkcji mogą być zapisywane i odczytywane
- Pozwala wybrać stopień rozwoju drzewa (ilość iteracji l-systemu) oraz jego wiek
- Aktualny stan drzewa można wyeksportować do formatu COLLADA, wraz z teksturą
- Niskie obciążenie procesora i karty graficznej podczas pracy z programem

Okno główne programu:



Opis elementów interfejsu.

1. Przyciski do zarządzania regułami produkcji. Można wyczyścić aktualne (nowa), wczytać oraz zapisać. Reguły produkcji są zapisywane w formie dokumentu XML, który zawiera informacje na temat poszczególnych reguł produkcji, symboli wchodzących w ich skład oraz ich atrybutów. Co więcej zapisywany jest aksjomat, jako symbol specjalnego typu.
2. Otwiera edytor reguł produkcji, który zostanie omówiony później
3. Dla przypomnienia, atrybuty symbolu mogą być zdefiniowane jako konkretna liczba, albo przedział, z którego będzie wylosowana konkretna wartość. Ten przycisk powoduje ponowne wylosowanie ich we wszystkich symbolach.
4. Określa ilość iteracji L-systemu, czyli głębokość, na którą obliczane jest drzewo. Kliknięcie przycisku Wygeneruj spowoduje ponowne obliczenie systemu wraz z nowymi wartościami losowymi.
5. Suwak wieku służy do ręcznego określania stadium rozwoju drzewa. Pełni też on rolę wskaźnika postępu animacji
6. Wyświetla okienko opcji:



Można tu zdefiniować:

a. Ilość boków gałęzi. Drzewa oglądane z bardzo bliska powinny mieć min. kilkanaście boków, ale te robiące za tło obędą się trzema.
b. Funkcja „progresywne zwężanie” modyfikuje grubość gałęzi tak, aby każda następna generacja była węższa. Dzięki niej nie trzeba w regułach produkcji ręcznie dbać o grubość gałęzi. Wtedy promień danej gałęzi wyraża się wzorem

$$r = \max\left(\frac{\text{obecny wiek drzewa} - \text{generacja gałęzi}}{\text{wiek całego drzewa}} \cdot r_{max} \cdot \text{skala}, r_{min}\right)$$

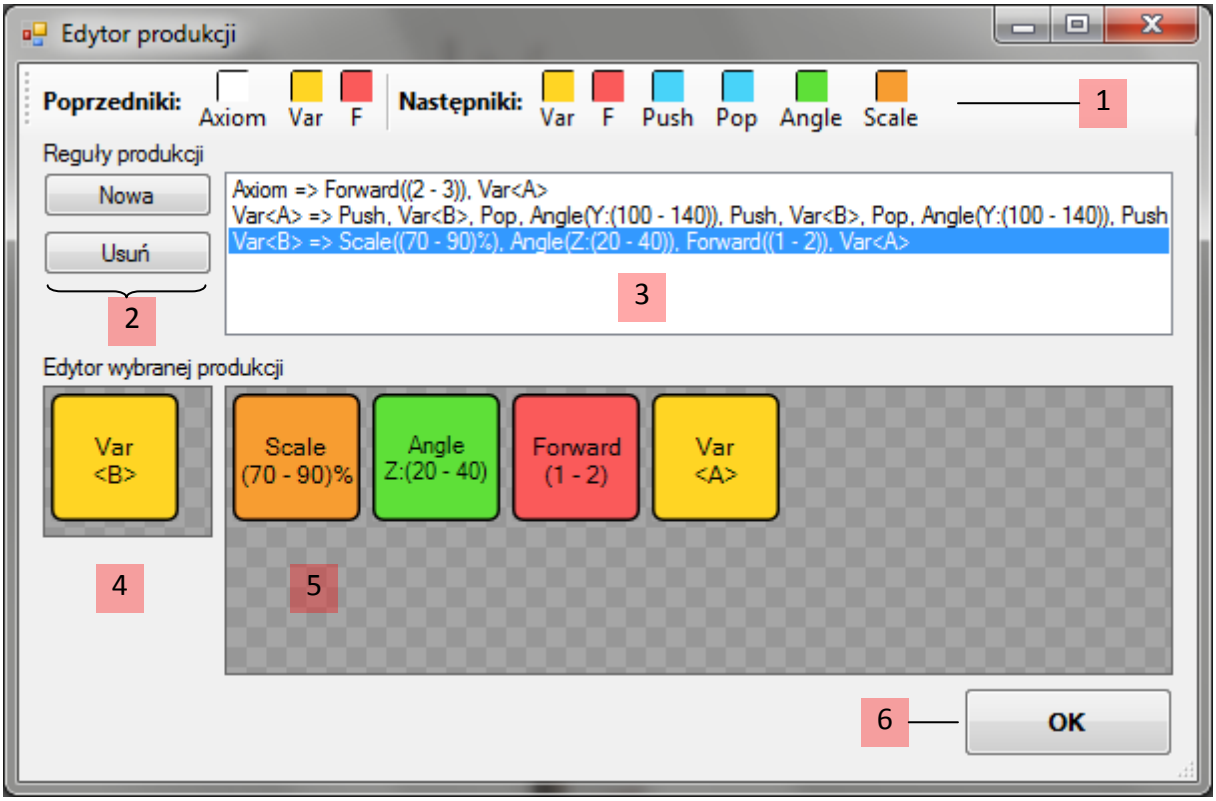
Oznacza to, że grubość gałęzi jest liniowo zależna od aktualnego wieku drzewa i generacji gałęzi, a zakres zmian jest od r_{min} do r_{max}.

c. Grubości gałęzi: r_{min} i r_{max}.

d. Wyświetlanie informacji pomocniczych – po wybraniu tej opcji na każdej gałęzi wyświetli się numer jej generacji, a także pojawi się nowy przycisk wyświetlający drzewo w formie grafu dla łatwiejszej analizy łańcucha poprzednik-następnik.

7. Przyciski służące do kontroli animacji.
8. Przycisk wyświetlający okno dialogowe, umożliwiające wybór miejsca zapisu wyeksportowanego aktualnego stanu drzewa.
9. Pasek stanu pokazujący ilość cylindrów, wierzchołków i aktualny wiek drzewa.
10. Kontrolka wyświetlająca efekty pracy OpenGLa. Służy również do nawigacji po scenie za pomocą myszy.

Okno edytora produkcji



1. Pasek narzędzi umożliwiający wstawianie nowych symboli w roli poprzedników lub następników reguł produkcji.
2. Przyciski służące do dodawania i usuwanie reguł produkcji.
3. Lista zdefiniowanych reguł produkcji. Wybranie pozycji powoduje wyświelenie jej w edytorze.
4. Lewa strona produkcji. Może znajdować się tam 1 z 3 symboli.
5. Prawa strona produkcji. Można tam umieścić dowolną ilość symboli.
6. Przycisk zamykający edytor i zatwierdzający zmiany.

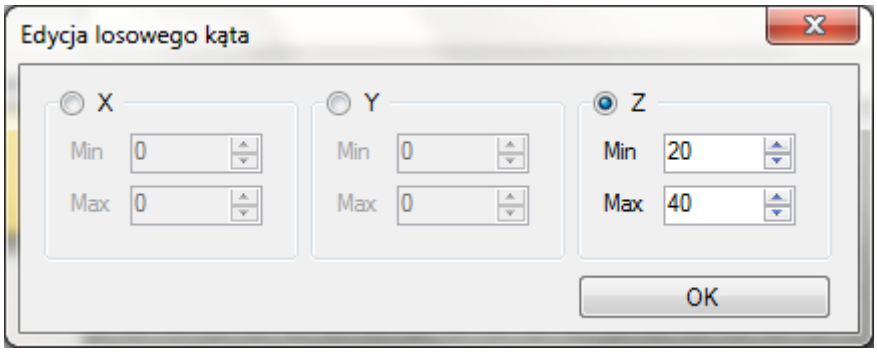
Rolą edytora jest intuicyjne układanie symboli w reguły produkcji tak, aby użytkownik nie musiał zapamiętywać szeregu znaków, będących alfabetem „klasycznego” L-systemu. Dodatkowo edytor umożliwia wygodne przesuwanie symboli techniką drag-and-drop, wycinanie, kopiowanie oraz wklejanie.

Podwójne kliknięcie na symbol powoduje edycję jego atrybutów:

- Osi oraz kąta w przypadku „angle”
- Długości w przypadku „forward”
- Nazwy zmiennej w przypadku „var”
- Współczynnika skali w przypadki „scale”

Dodatkowo wszystkie wartości liczbowe są podawane w formacie od-do, dzięki czemu można kontrolować stopień ich losowości.

Przykładowe okno edycji kąta zwalnia użytkownika od zapamiętywania 6 symboli:



Zakończenie

Gry komputerowe

Zaprezentowane algorytmy są na tyle wydajne, że z powodzeniem zostać użyte w grach komputerowych. Przykładowy scenariusz zakłada sadzenie drzew na mapie i animację ich wzrostu wraz z upływem czasu (na wzór SimCity).

Innym podejściem jest eksport wygenerowanych drzew w celu użycia ich jako modeli w grze. Pozwala to dowolnie dostosować efekt końcowy – dopasować konkretne gałęzie, podmienić tekstury, wygiąć lub przechylić całe drzewo itd. Format COLLADA jest obecnie obsługiwany przez większość popularnych programów do grafiki 3D, więc nie ma problemu z jego importem.

Możliwe pola rozwoju

Praca nie wyczerpuje wszystkich zagadnień związanych z symulacją wzrostu drzew. Należy skupić się na kwestiach związanych z realizmem.

- Zdarzenia podziału gałęzi następują zawsze w tych samych momentach wzrostu drzewa. Jest to związane z algorytmem, który pozwala na jedynie jedną „rosnącą" generację. Mimo że na wiosnę wszystkie drzewa wypuszczają nowe pędy, to nigdy nie robią tego jednego dnia, tak więc tworzenie nowych gałęzi w nieco bardziej losowych momentach poprawiłoby realizm animacji.
- Gałęzie są prostymi cylindrami. Ta forma jest odpowiednia dla małych roślin, lecz większe drzewa mają gałęzie zwykle nieco powyginane przez grawitację, wiatr lub choroby. Prostym rozwiązaniem byłby podział cylindrów tak, aby miały więcej płaszczyzn poziomych, do których współrzędnych można dodać niewielką wartość szumu. Sprawi to, że gałęzie będą mniej sztuczne, lecz znacząco zwiększy ilość poligonów, więc może utrudnić animację drzewa w czasie rzeczywistym.
- Format COLLADA pozwala na eksport animacji, więc była by to atrakcyjna funkcja programu. Niestety okazało się to zadaniem skomplikowanym i wymagającym długiej analizy, więc bardzo wymagającym czasowo, przez co nie zostało zrealizowane w zaplanowanych ramach czasu.