Realistyczny rendering krajobrazów leśnych generowanych proceduralnie

(Realistic rendering of procedural generated forest landscapes)

Bartosz Rudzki

Praca inżynierska

Promotor: dr Andrzej Łukaszewski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

8 stycznia 2020

Streszczenie

Strzeszczenie po polsku...

Abstract

English abstract...

Spis treści

Rozdział 1.

Wprowadzenie

Pracy skupia się na zbadaniu i zastosowaniu dwóch pojęć z dziedziny grafiki komputerowej: realistyczny rendering i generowanie proceduralne.

Pierwsze zagadnienie dotyczy przedstawienia wcześniej przygotowanego modelu w sposób zrozumiały dla człowieka. Modelem może być plik opisujący kształt dzbanka. Przykładowa reprezentacja to zbiór trójkątów rozmieszczonych w przestrzeni z uwzględnieniem dodatkowych informacji takich np. kolor. Rendering takiego modelu polegałby na zamianie wszystkich tych danych na obraz, który mógłby być pokazany człowiekowi. O realistycznym renderingu możemy mówić wtedy, kiedy stworzony przez nas obraz przypomina prawdziwy dzbanek, który chcieliśmy zamodelować.

Generowanie proceduralne skupia się na tworzeniu modeli przy pomocy algorytmów. Człowiek może wpływać na finalny produkt przy pomocy zmiany parametrów jednak nie uczestniczy w samym procesie tworzenia. Jest to bardzo wydajna metoda pozwalająca na tworzenie nieskończenie wielu różnych modeli o zadanych właściwościach jak np. rośliny.

Celem programu napisanego w ramach tej pracy jest generowanie prostego krajobrazu leśnego, a następnie względnie realistyczny rendering stworzonego środowiska. Sam program jest wysoce sparametryzowany, dzięki czemu użytkownik może realnie wpływać na finalny rezultat.

Rozdział 2.

Zastosowane rozwiązania

W tym rozdziale przedstawię najważniejsze rozwiązania, które zastosowałem w swoim programie. Postaram przybliżyć się najważniejsze zagadnienia głownie po to, aby potencjalny użytkownik programu świadomie mógł wpływać na jego działanie.

2.1. Generowanie proceduralne

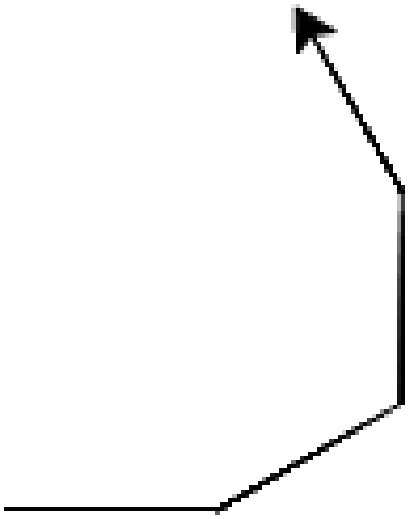
2.1.1. Rośliny

Kształt roślin posiada wiele regularności. Dzięki temu można opisać je przy pomocy tak zwanych L-systemów. Jest to sposób reprezentacji modelu pod postacią zestawu reguł tworzących gramatykę. Zaczynając od jednego symbolu jesteśmy wstanie wygenerować zbiór symboli - słowo. Osiągamy to poprzez zdefiniowanie dodatniej liczby różnych produkcji. Są to reguły zamieniania wybranego symbolu na słowo. Przykładowy L-System może wyglądać następująco:

$$\begin{aligned} axiom &: \alpha \\ prod1 &: \alpha \rightarrow \alpha\beta\alpha \\ prod2 &: \beta \rightarrow \beta\beta \end{aligned}$$

Produkcje aplikowane są jednocześnie w obrębie jednej iteracji. Podając liczbę iteracji możemy generować coraz dłuższe ciągi. Dla N = 1 otrzymujemy $\alpha\beta\alpha$, a dla N = 2 mamy $\alpha\beta\alpha\beta\beta\alpha\beta\alpha$.

Finalnym produktem L-systemu jest słowo, które możemy potraktować jako polecenia dla rysującego żółwia. Żółw czytając słowo od lewej do prawej będzie interpretować każdy symbol jako komendę. Przykłady poleceń to: idź prosto rysując linię albo skręć w lewo o wcześniej ustalony kąt. W przypadku gdy żółw nie zna symbolu nie robi nic i czyta dalej. Rysunek 2.1 przedstawia przykładową interpretację słowa $\alpha\beta\alpha\beta\beta\alpha\beta\alpha$ gdzie α to idź naprzód, a β skręć w lewo o 30°.



Rysunek 2.1: Interpretacja dla słowa $\alpha\beta\alpha\beta\beta\alpha\beta\alpha$, gdzie α to idź naprzód, a β skręć w lewo o 30°.

Aby żółw był wstanie rysować rośliny potrzebujemy symbol pozwalający się rozgałęziać. Można osiągnąć to poprzez dodanie stosu pamiętającego aktualny stan żółwia oraz symbole na nim operujące:

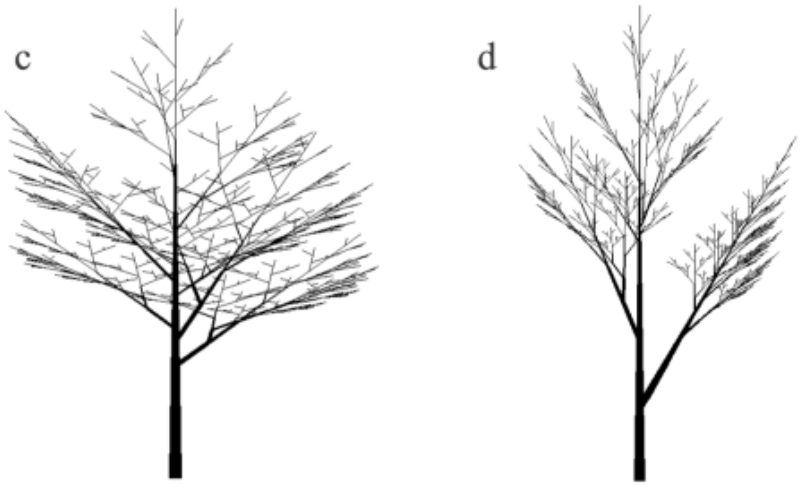
[- odłóż aktualny stan na stos

] - ściągnij i zaaplikuj stan z góry stosu

Stanem żółwia jest wszystko co zmienia się przy pomocy symboli np. pozycja, orientacja lub długość rysowanej linii. Rysunek 2.2 pokazuje przykładowe L-systemy z książki *The Algorithmic Beauty of Plants*[1], które używają symboli tworzących rozgałęzienia.

Rysunek 2.2: Przykład użycia symboli operujących na stosie[1]

Wspomniana wcześniej książka Prusinkiewicza i Lindenmayer'a[1] dokładnie opisuje L-systemy jako zagadnienie naukowe oraz podaje wiele przykładów produkcji tworzących realistycznie wyglądające rośliny. Część z nich wymaga bardziej zaawansowanych technik produkcji symbolów np. L-systemy parametryczne. Do tej pory zakładaliśmy, że wszystkie parametry (np. kąt skrętu, długość kroku) były ustalone na początku i nie zmieniały się. Parametryzując symbole i produkcje możemy wpływać lepiej na finalny kształt. Ma to zastosowanie między innymi w generowaniu drzew. Rysunek 2.3 przedstawia produkcje drzew wymyślonych przez Hondę[2], a pokazanych u Prusinkiewicza[1].




```
n = 10
#define r1 0.9     /* contraction ratio for the trunk */
#define r2 0.6     /* contraction ratio for branches */
#define a0 45      /* branching angle from the trunk */
#define a2 45      /* branching angle for lateral axes */
#define d 137.5    /* divergence angle */
#define wr 0.707   /* width decrease rate */

ω  : A(1,10)
p1: A(l,w) : * → !(w)F(l)[&(a0)B(l*r2,w*wr)]/(d)A(l*r1,w*wr)
p2: B(l,w) : * → !(w)F(l)[-(a2)$C(l*r2,w*wr)]C(l*r1,w*wr)
p3: C(l,w) : * → !(w)F(l)[+(a2)$B(l*r2,w*wr)]B(l*r1,w*wr)
```

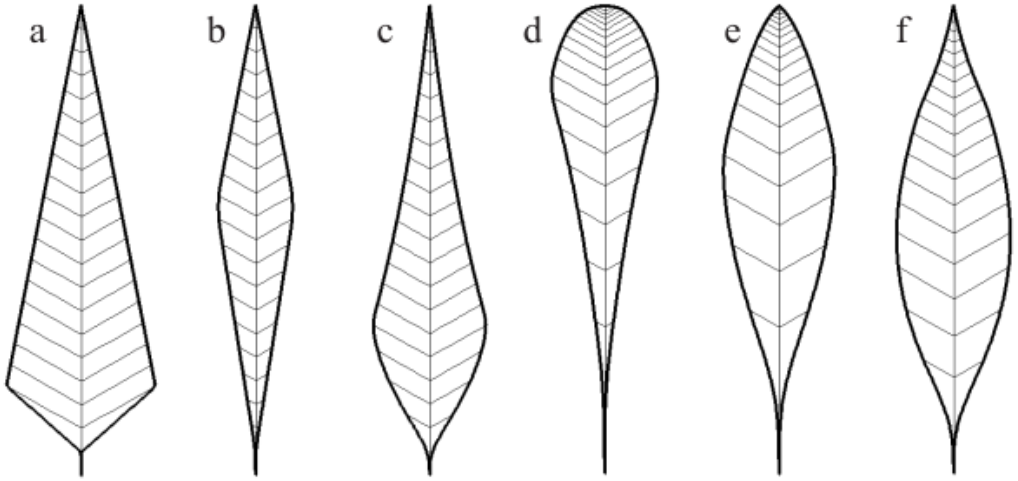
Rysunek 2.3: L-system parametryczny generujący drzewa [1][2]

L-systemy mogę również generować wielokąty. Jest to szczególnie przydatne w tworzeniu liści. Rysunek 2.4 prezentuje kilka rodzajów liści wygenerowanych z jednego L-systemu. Możliwe jest to dzięki wprowadzeniu nowych symboli:

{ - zacznij tworzyć nowy wielokąt

. - zapisz aktualną pozycję jako wierzchołek

} - zakończ aktualny wielokąt



n=20, δ=60°

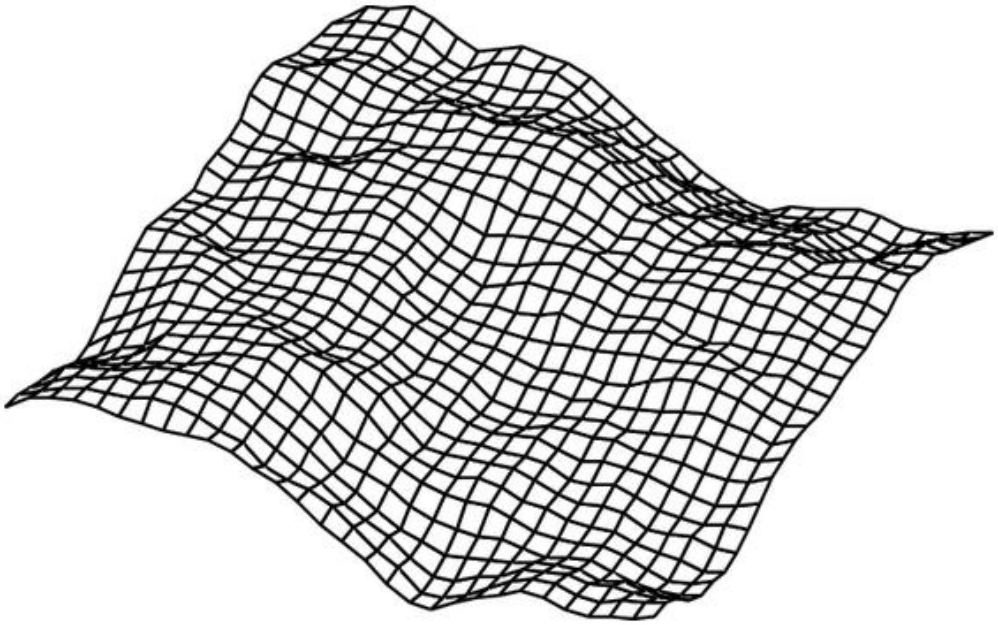
```
#define LA 5     /* initial length - main segment */
#define RA 1     /* growth rate - main segment */
#define LB 1     /* initial length - lateral segment */
#define RB 1     /* growth rate - lateral segment */
#define PD 1     /* growth potential decrement */

ω  :  {.A(0)}
p1 :  A(t)     :  *    → G(LA,RA)[-B(t).][A(t+1)][+B(t).]
p2 :  B(t)     :  t>0  → G(LB,RB)B(t-PD)
p3 :  G(s,r)   :  *    → G(s*r,r)
```

Rysunek 2.4: L-system parametryczny generujący liście [1]

Wcześniej opisane zastosowania L-systemów pokazują dlaczego jest to idealny kandydat do tworzenia realistycznych modeli roślin. Różnorodność definiowanych kształtów, rozbudowywanie funkcjonalności poprzez dodawanie nowych symboli oraz łatwe parametryzowanie to jedne z wielu zalet tej techniki. Wszystko to sprawia, że L-systemy są często wykorzystywane w wielu różnych projektach - również w tym.

2.1.2. Teren

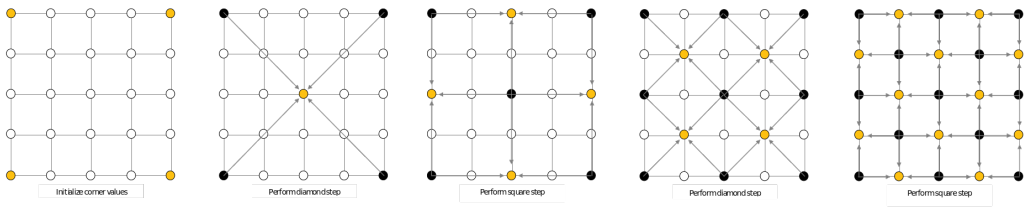
Problem generowania terenu można sprowadzić do problemu generowania mapy wysokości. Jest to macierz wypełniona liczbami reprezentującymi wysokość wybranych punktów. Owe punkty pomiaru rozłożone są równomiernie na osi X i Z. Mając taką macierz w łatwy sposób jesteśmy wstanie wygenerować teren - tworzymy kwadraty dla każdej czwórki sąsiadujących punktów - patrz rysunek 2.5.



Rysunek 2.5: Przykładowy teren wygenerowany przy użyciu mapy wysokości[3]

Istnieje wiele algorytmów generujących mapy wysokości. Mój wybór padł na tak zwany *Diamond-square algorithm*. Dla ustalonego N generuje on mapę wysokości o rozmiarze 2^N+1 x 2^N+1. Na początku ustala się wartości w rogach macierzy. Następnie wykonuje się na przemiennie fazę diamond i square, aż do ustalenia wszystkich wysokości.

Faza diamond polega na wygenerowaniu wysokości w środku każdego kwadratu, który nie ma ustalonej tej wartości. Faza square robi to samo tylko, ze dla punktów tworzących kształt rombu. Rysunek 2.6 obrazuje przebieg algorytmu.



Rysunek 2.6: Przebieg algorytmu diamond-square dla N = 2.

Obliczanie nowej wartości w środku polega na wzięciu średniej z wysokości wierzchołków tworzących kształt (kwadrat, romb). Dodatkowo dodajemy do obliczanej wartości pewien czynnik losowy, z przedziału (-X, X) dla ustalonego X. Dla

każdej kolejnej pary faz ów czynnik jest zmniejszany poprzez przemnożenie go przez pewne $R < 1$.

Diamond-square algorithm jest zarówno prosty jak i efektywny. Ustalając kilka parametrów jesteśmy wstanie tworzyć bardzo różnorodny teren.

2.1.3. Tekstury

Mając wygenerowany model - kształt, musimy nadać mu również kolor. Jednym z założeniem programu było w pełni proceduralne podejście. Dotyczy to również tekstur. Każda tekstura w programie opisywana jest przez materiał. Materiał oblicza kolor na podstawie funkcji *calcDiffuse(pos3D)* $\rightarrow$*color*, gdzie $pos3D$ to pozycja w świecie. W celu uproszczenia i ujednolicenia systemu zdecydowałem się zaimplementować jeden wzór, który został zastosowany do każdego materiału. Wygląda on następująco:

$$calcDiffuse(pos3D) = mix(color1, color2, factor)$$
$$gdzie$$
$$factor = clamp(noise(pos3D * posF) * valF, 0, 1)$$

Finalny kolor to mieszanka $color1$ i $color2$ w stosunku $1 - factor$ i $factor$. Sam $factor$ to odpowiednio obliczona wartość szumu Perlina. Dzięki $posF$ możemy skalować szum, a dzięki $valF$ możemy wymuszać faworyzowanie wartości bliższych 0 albo 1.

Jedynym materiałem w programie odbiegającym od tej metody jest materiał terenu. Łączy on ze sobą dwa inne materiały - grunt i skały. Do pewnej ustalonej wysokości kolor obliczany jest tylko dla gruntu. Analogicznie powyżej pewnego poziomu stosuję tylko kolor skał. Po środku dochodzi do mieszania kolorów w stosunku zależnym od odległości do ustalonych granic:

$$mix(calcRockColor(pos), calcGroundColor(pos), groundFactor)$$
$$gdzie$$
$$groundFactor = (groundEndY - posY)/(groundEndY - rockStartY)$$

Zaprezentowane rozwiązanie okazało się wystarczające, aby osiągnąć zadowalające rezultaty. Jest łatwe do implementacji, ujednolicone dla użytkownika oraz na tyle rozbudowane, aby wyrażać niebanalną kolorystkę.

2.1.4. Rozmieszczanie roślin

Po wygenerowaniu wystarczającej liczby modeli roślin trzeba je umieścić w świecie. Można to robić w mniej lub bardziej wyszukany sposób. W tym przypadku zde-

cydowałem się na jedno z najprostszych rozwiązań - losowe rozmieszczanie wewnątrz komórek kraty. Krata o wymiarach N x M posiada $N * M$ kwadratowych komórek o ustalonym rozmiarze. Wewnątrz takiej komórki z pewnym prawdopodobieństwem możemy umieścić losowy model. Pozycja modelu wewnątrz kraty również jest losowa i może zostać ograniczona przez parametr mówiący jak bardzo od środka może zostać wylosowana pozycja. Przykładowo dla wartości 0 obiekt zawsze będzie umieszczany w środku kraty, a dla 1 może pojawić się w całej kracie.

2.1.5. Wybór miejsca renderingu

Aby teren wydawał się realistyczny musi zajmować dużą część świata. Dobrze byłoby wpływać na to czy chcemy renderować naszą scenę ze szczytu góry czy z terenu nizinnego. W jakim kierunku powinna być skierowana kamera oraz co ważniejsze uwzględniać ukształtowanie otaczającego ją terenu. W przypadku gdy kamera zostanie umieszczona przed górą chcielibyśmy, aby patrzyła do góry, a kiedy jest na szczycie w dół.

Odpowiednie umieszczenie możemy wybrać poprzez parametr sugerujący na jakiej wysokości chcemy się znajdować. Dobrze zdefiniować również minimalną odległość od krawędzi końca świata, aby uniknąć sytuacji patrzenia w pustą przestrzeń. Żeby uwzględnić ukształtowanie terenu przy kierunku patrzenia możemy dodać parametr mówiący jak daleko od nas znajduję się miejsce które nas interesuje. Następnie obliczamy wysokość tego miejsca i korygujemy nasz kierunek patrzenia względem otrzymanej wartości.

2.1.6. Ograniczenie liczby modeli

Przy generowaniu tak dużego świata musimy liczyć się z ograniczeniami sprzętowymi. Niezoptymalizowane modele otrzymywane z L-systemów mają nawet po kilkanaście tysięcy trójkątów! W takim przypadku dobrym sposobem tworzenia sceny jest umieszczenia wszystkich generowanych modeli względnie blisko kamery. Duże modele będą zasłaniać znaczną część pola widzenia ukrywając niewypełnioną przestrzeń. Rozwiązanie nie jest idealne ale ma szanse wygenerować scenę o zadowalającym rezultacie.

2.2. Realistyczny rendering

Do tej pory opisywałem techniki pozwalające generowanie różnych modeli składających się w jeden wielki świat. W tej sekcji skupię się jak uchwycić część tego co udało się stworzyć. Sam proces zwany renderingiem w tym przypadku przypominać będzie nic innego jak zrobienie zdjęcia. Wcześniej wspomniałem o koncepcie kamery,

która znajduje się w świecie oraz jest zwrócona w określonym kierunku. To co należy zrobić teraz to uchwycić obraz z jej perspektywy.

2.2.1. Path tracer

Głównym zadaniem path tracer'a jest obliczanie koloru docierającego do kamery z ustalonego kierunku. Kolor w tym przypadku to nic innego jak promień światła, który po wielu odbiciach dotarł do kamery. Postaramy odtworzyć się to zjawisko, z tym wyjątkiem, że promień zostanie wypuszczony od kamery i będzie podążać do światła. Cały proces jest wysoce losowy, jako że promień mógł dotrzeć do nas wieloma różnymi ścieżkami. W celu otrzymania jak najlepszych wyników w jednym kierunku będziemy puszczać wiele promieni, a na końcu je uśrednimy.

Aby otrzymać realistyczny obraz path tracer przy obliczaniu koloru będzie korzystać z tak zwanego *równania renderingu*. Samo równanie korzystając z zasady zachowania energii wylicza radiancję światła wychodzącą z ustalonego punktu na płaszczyźnie dla zadanego kierunku. W uproszczonej formie wygląda ono następująco:

$$L_o = L_e + \int_{\Omega} L_i * f_r * cos\theta * d\omega_i \tag{2.1}$$

2.2.2. Niebo

Path tracing często stosuje się dla zamkniętych scen z kilkoma źródłami światła. W przypadku rendorowania krajobrazów mamy do czynienia z czymś zupełnie innym. Mamy jedno źródło światła (słońce) oraz pół otwartą scenę z wielkim niebem pochłaniającym większość promieni. Fizycznie poprawny model nieba został opisany w pracy *A Practical Analytic Model for Daylight*[4], która powstała na Uniwersytecie w Utah. Opisuje on niebo w danym miejscu na ziemi o ustalonej porze.

Został również udostępniony gotowy kod implementujący zaprezentowane rozwiązanie. Dołączyłem go do swojego programu i korzystam z niego jak z zewnętrznej biblioteki nie wnikając w szczegóły implementacyjne. Niebo w moim programie to półsfera rozciągająca się nad całym wygenerowanym terenem. Implikuje to, że im większy jest teren, tym większe jest niebo.

2.2.3. Słońce

O ile model nieba z poprzedniej sekcji dobrze oddaje kolorystykę nieba to jest ona dosyć jednolita. Dodatkowo jako, że niebo otacza scenę praktycznie z każdej strony, to nie może być traktowane jako główne źródło światła. Żeby otrzymać ładne cienie obiektów dołożyłem dodatkowo własne słońce jako źródło światła o

dużej mocy. Model nieba oblicza również pozycję słońca i korzystam z tego, aby odpowiednio umieścić moje. Umieszczam je na półsferze nieba stąd odległość słońca od ziemi jest zależna od wielkości terenu (patrz poprzednia sekcja).

Rozdział 3.

Poradnik użytkowania

Poprzedni rozdział opisywał rozwiązania, które zastosowałem w programie. Ten skupi się jak uruchomić program oraz na przedstawieniu w jaki sposób można wpływać na rendering i generowane elementy.

3.1. Uruchomienie programu

W celu kompilacji programu należy uruchomić skrypt *build.sh* w głównym katalogu. Utworzy on folder *build*, który posłuży jako miejsce dla plików programu *cmake*. Dodatkowo w folderze bin powinien pojawić się plik binarny do uruchamiania programu - *rpforest.exe*. Program posiada dwa opcjonalne argumenty wywołania.

Pierwszy to ścieżka do pliku konfiguracyjnego path tracingu. Kilka przykładowych plików znajduje się w folderze bin. Wszystkie mają rozszerzenie *.rtc* i ich struktura zostanie opisana w kolejnej sekcji. W przypadku nie podania ścieżki zostanie załadowany plik *default.rtc*.

Drugi argument to ziarno losowości. W przypadku nie ustawienia żadnego, zostanie wybrane losowe - zależne od czasu. W logach programu można znaleźć ziarno dla danego uruchomienia. Możemy wybrać je podobnie jeśli chcemy wygenerować ten sam świat.

3.2. Pliki konfiguracyjne

Każdy plik konfiguracyjny jest parsowany w ten sam sposób. Pojedynczy wpis składa się z nazwy parametru i argumentów zależnych od typu parametru. Kolejność nie ma znaczenia. Każdy parametr musi być zdefiniowany w osobnej linii. Część parametrów ma domyślne wartości

3.2.1. Ustawienia path tracingu

.rtc

- *string PhotoName* - nazwa
- *vec2 Resolution* - rozmiar
- *int Samples = 1* -
- *int MaxRayBounces* = 5 -
- *bool DebugMode* = 0 -

3.2.2. Ustawienia kamery

camera.conf

- *float LookHeight = 1* -
- *vec3 LookDirection = (0, 0, 1)* -
- *float LookDistance = 2* -
- *float EdgeMinOffset = 10* -
- *float ExpectedPositionY = 0* -
- *float DebugVelocity = 10* -

3.2.3. Ustawienia nieba

sky.conf

- *vec3 SunColor* -
- *float SunPower* -
- *float SkyLuminanceFactor* -
- *float Latitude* -
- *float Longitude* -
- *int StandardMeridian* -
- *int JulianDay* -
- *float TimeOfDay* -
- *float Turbidity* -

3.2.4. Ustawienia terenu

terrain.conf

- *int N* -
- *float Size* -
- *float InitHeight* -
- *float Spread* -
- *float SpreadReductionRate* -
- *float RockStartY = 10* -
- *float GroundEndY = 20* -

3.2.5. Ustawienia Tekstur

.mat

- *vec3 Color1* -
- *vec3 Color2* -
- *float NoisePositionFactor = 1* -
- *float NoiseValueFactor = 1* -

3.2.6. Bazowe ustawienia generatorów

- *int N* -
- *float Height* -
- *int ConeBasePoints = 3* -
- *string Material* -
- *float InitRadius* -

3.2.7. Definiowanie własnych L-systemów

.lsys

- *float TurnAngle = 0* -
- *float PitchAngle = 0* -

- *float RollAngle = 0* -
- *string Axiom* -
- *char > string Prod* -

3.2.8. Ustawienia predefiniowanych L-systemów

.honda

- *float R1* -
- *float R2* -
- *float A0* -
- *float A2* -
- *float D* -
- *float Wr* -

.family

- *float Delta* -
- *float LA* -
- *float RA* -
- *float LB* -
- *float RB* -
- *float PD* -

3.2.9. Ustawienia rozmieszczania roślin

.spawner

- *vec2 CameraOffset* -
- *int XGridN = 1* -
- *int ZGridN = 1* -
- *float CellSize = 0* -
- *string GeneratorType* -

- *float SpawnProbability = 1* -
- *float RandomPositionPercentage = 0.5* -
- *float MinElementScale = 0.5* -
- *float MaxElementScale = 2* -

Rozdział 4.

Szczegóły programistyczne

Rozdział 5.

Przykłady użycia

Bibliografia

[1] P. Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, Berlin, Heidelberg, 1990.

[2] H. Honda. Description of the form of trees by the parameters of the tree-like body: Effects of the branching angle and the branch length on the shape of the tree-like body. 1971.

[3] Miguel Frade, Francisco Vega, and Carlos Cotta. Breeding terrains with genetic terrain programming: The evolution of terrain generators. *Int. J. Computer Games Technology*, 2009, 12 2009.

[4] Brian Smits A. J. Preetham, Peter Shirley. A practical analytic model for daylight.