

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

22 stycznia 2020

Streszczenie

Praca przedstawia program, który realistycznie renderuje wcześniej wygenerowane krajobrazy leśne. Opisuje zastosowane rozwiązania z dziedziny generowania proceduralnego i realistycznej grafiki. Należą do nich między innymi: L–systemy, algorytm Diamond-Square oraz path tracing. Pokazuje jak uruchomić program i wpływać na jego działanie przy pomocy plików konfiguracyjnych. Wyjaśnia podstawową architekturę aplikacji i użyte narzędzia. Prezentuje przykładowe efekty i porównuje je z komercyjnymi implementacjami.

The paper presents a program that realistically renders previously generated forest landscapes. It describes used solutions from procedural generation and realistic graphics fields. Some of them are: L–systems, the Diamond-Square algorithm and path tracing. It shows how to run the program and affects its effects with configuration files. Explains the basic application architecture and used tools. Presents example effects and compares it to commercial implementations.

Spis treści

1. Wprowadzenie	7
2. Generowanie proceduralne krajobrazów	9
2.1. Rośliny	9
2.2. Teren	12
2.3. Tekstury	14
2.4. Rozmieszczanie roślin	15
2.5. Ograniczenie liczby modeli	15
3. Realistyczny rendering	17
3.1. Path tracer	17
3.2. Niebo	18
3.3. Słońce	19
3.4. Wybór miejsca renderingu	19
4. Poradnik użytkowania	21
4.1. Uruchomienie programu	21
4.2. Tryb zwykły i debugowania	22
4.3. Pliki konfiguracyjne	22
4.3.1. Ustawienia path tracingu	23
4.3.2. Ustawienia kamery	23
4.3.3. Ustawienia debugowania	24
4.3.4. Ustawienia nieba	24
4.3.5. Ustawienia terenu	24

4.3.6. Ustawienia tekstur	25
4.3.7. Bazowe ustawienia generatorów	25
4.3.8. Definiowanie własnych L-systemów	26
4.3.9. Ustawienia predefiniowanych L-systemów	27
4.3.10. Ustawienia rozmieszczenia roślin	27
5. Szczegóły programistyczne	29
5.1. Wykaz użytych narzędzi	29
5.2. Struktura projektu	29
5.2.1. Moduł <i>engine</i>	30
5.2.2. Moduł <i>rendering</i>	31
5.2.3. Moduł <i>generating</i>	31
5.2.4. Moduł <i>utils</i>	32
6. Zastosowanie	33
6.1. Przykłady użycia	33
6.2. Porównanie z komercyjnymi rozwiązaniami	36
Bibliografia	39

Rozdział 1.

Wprowadzenie

Oba zagadnienia z tytułu pracy: realistyczny rendering oraz generowanie proceduralne przynależą do poddziedziny informatyki zwanej grafiką komputerową. Nauka ta, w dużym uproszczeniu, skupia się w jaki sposób przy pomocy komputera tworzyć obraz zrozumiały dla człowieka.

To co widzą ludzie na ekranie komputera to finalny efekt procesu zwanego renderingiem. Zaczyna się on od zdefiniowania modelu. Opisuje on wszystko to co chcielibyśmy pokazać. Przykładowa reprezentacja to zbiór trójkątów rozmieszczonych w przestrzeni z uwzględnieniem dodatkowych informacji takich np. kolor. Następnie komputer zamienia tak zapisane dane na obraz. To w jaki sposób odbywa się zamiana wpływa na jakość tego co zostanie pokazane. Komputer może generować nawet kilkaset różnych klatek obrazu na sekundę jeśli nie jest wymagana wysoka realistyczność obrazu. Jeśli chcemy, żeby efekt końcowy dawał złudzenie fotorealizmu musimy zastosować bardziej skomplikowane algorytmy wymagające większej mocy obliczeniowej. Ta praca skupia się na drugim przypadku.

Sam model, często przygotowywany przy człowieku, może być stworzony w całości przez komputer. Na tym polega generowanie proceduralne. Model to efekt algorytmu. Człowiek może wpływać na jego działanie poprzez zmianianie parametrów, jednak nie musi być zaangażowany w sam proces tworzenia. Jest to bardzo wydajna metoda pozwalająca na generowanie nieskończonie wielu różnych modeli o zadanych właściwościach jak np. rośliny.

Celem pracy było napisanie programu, który generuje proceduralnie prosty krajobraz leśny, a następnie realistycznie renderuje stworzone środowisko. Sam program jest wysoce sparametryzowany pozwalając użytkownikowi wpływać na większość generowanych rzeczy oraz sam proces renderingu.

Rozdział 2 i 3 opisują rozwiązania zaimplementowane w programie. Pierwszy z nich skupia się na generowaniu proceduralnym poszczególnych części krajobrazu. Kolejny przedstawia jak można w sposób realistyczny wyświetlić wygenerowany wcześniej świat.

Rozdział 4 to poradnik użytkowania. Prezentuje jak pobrać i zainstalować program. Przedstawia przydatne dla użytkownika informacje na temat programu oraz jak wpływać na jego konfiguracje.

Rozdział 5 zawiera informacje dla programistów. Opisuje i argumentuje użycie wszystkich technologii. Przedstawia strukturę projektu oraz nakreśla zaimplementowaną architekturę.

Rozdział 6 przedstawia kilka przykładowych obrazów stworzonych przez program. Wskazuje dobre praktyki używania oraz nakreśla potencjalne problemy i ograniczenia. Dodatkowo porównuje wygenerowane obrazy z profesjonalnymi, stworzonymi przy pomocy komercyjnych programów.

Rozdział 2.

Generowanie proceduralne krajobrazów

2.1. Rośliny

Kształt roślin posiada wiele regularności. Dzięki temu można opisać je przy pomocy tak zwanych L–systemów. Jest to sposób reprezentacji modelu jako reguł tworzących gramatykę. Zaczynając od jednego symbolu jesteśmy w stanie 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:

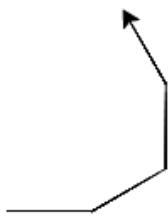
$$\begin{aligned} axiom &: \alpha \\ prod1 &: \alpha \rightarrow \alpha\beta\alpha \\ prod2 &: \beta \rightarrow \beta\beta \end{aligned}$$

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 polecień 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° .

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

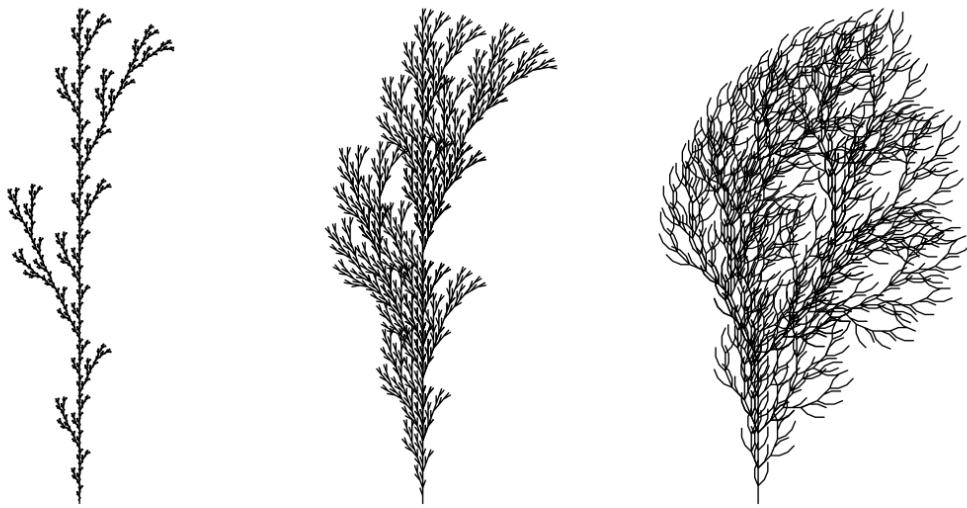
[– odłóż aktualny stan na stos



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° .

] – 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.



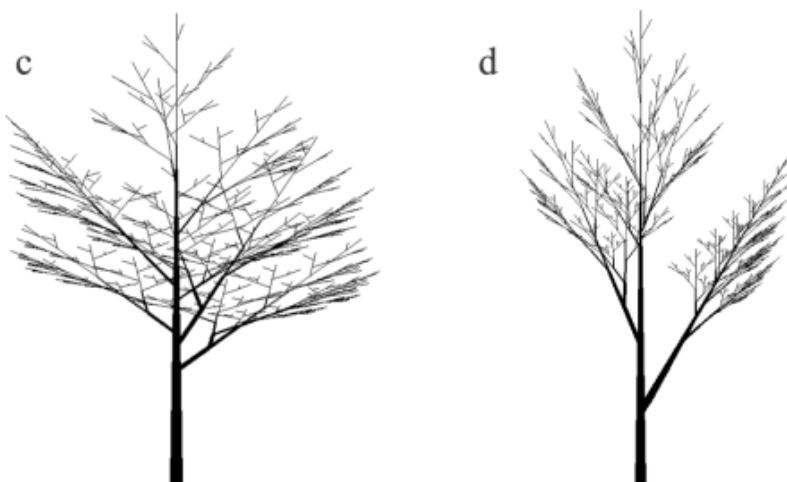
a
 $n=5, \delta=25.7^\circ$
F
 $F \rightarrow F [+F] F [-F] F$

b
 $n=5, \delta=20^\circ$
F
 $F \rightarrow F [+F] F [-F] [F]$

c
 $n=4, \delta=22.5^\circ$
F
 $F \rightarrow FF-[-F+F+F]+[+F-F-F]$

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)

```

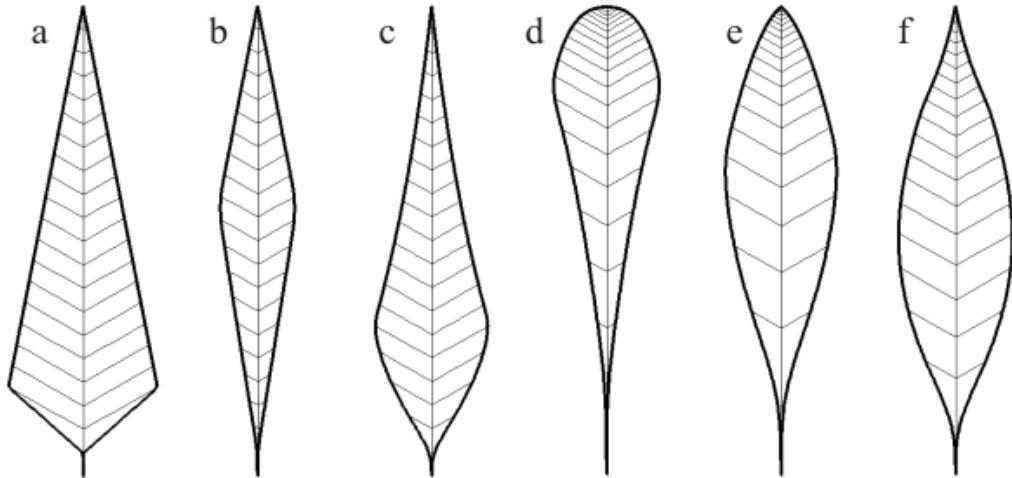
Rysunek 2.3: L-system parametryczny generujący drzewa [1][2]

L-systemy mogą 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



$n=20, \delta=60^\circ$

```
#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 */

 $\omega$  : { .A(0) }
 $p_1$  : A(t)   : *    → G(LA,RA)[-B(t).][A(t+1)][+B(t).]
 $p_2$  : B(t)   : t>0 → G(LB,RB)B(t-PD)
 $p_3$  : G(s,r) : *    → G(s*r,r)
```

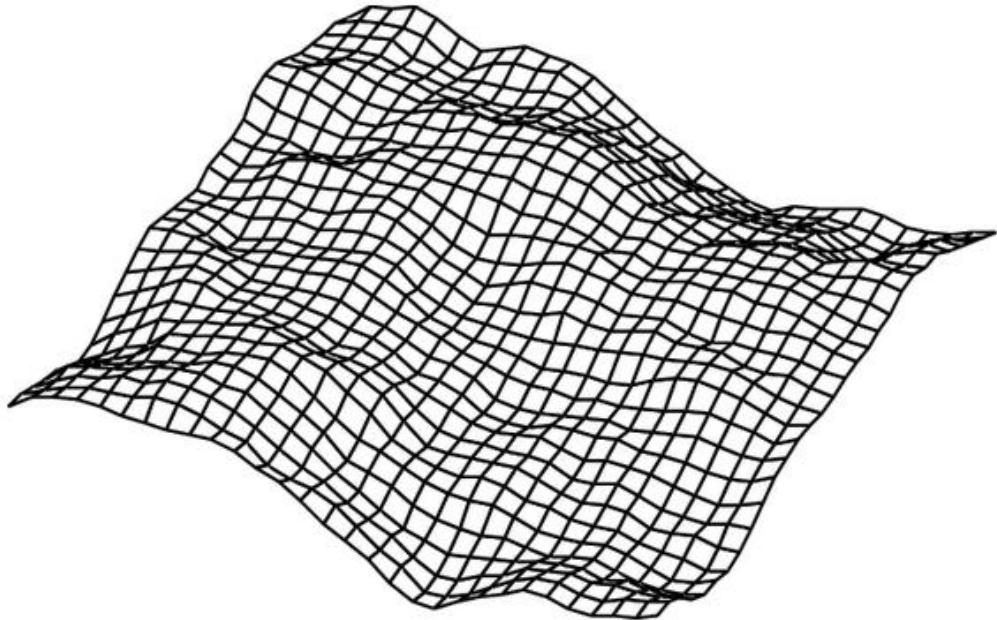
Rysunek 2.4: L–system parametryczny generujący liście [1].

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.

2.2. 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

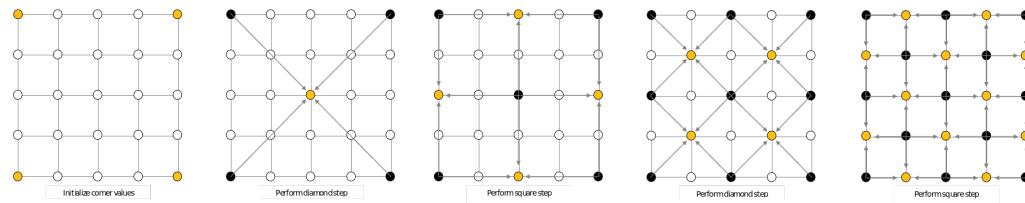
taką macierz w łatwy sposób jesteśmy w stanie wygenerować teren – tworzymy kwadraty dla każdej czwórki sąsiadujących punktów (rysunek 2.5).



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

Istnieje wiele algorytmów generujących mapy wysokości. W programie użyto tak zwanego *Diamond-square algorithm*. Dla ustalonego N generuje on mapę wysokości o rozmiarze $2^N+1 \times 2^N+1$. 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 wartości. Faza square robi to samo tylko, że 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 ustalone $R < 1$.

Diamond-square algorithm jest zarówno prosty jak i efektywny. Ustalając kilka parametrów jesteśmy w stanie tworzyć bardzo różnorodny teren. W łatwy sposób możemy wpływać na dokładność modelu oraz kształt. Wszystko to sprawia, że ten algorytm jest dobrym kandydatem do generowania terenu.

2.3. Tekstury

Aby wygenerowany model był kompletny poza kształtem musimy również nadać mu kolor. Jednym z założeń programu było w pełni proceduralne podejście. Dotyczy to również tekstur. Każda tekstura w programie opisywana jest przez materiał. Materiał oblicza kolor na podstawie funkcji $\text{calcDiffuse}(\text{pos3D}) \rightarrow \text{color}$, gdzie pos3D to pozycja w świecie. W celu uproszczenia i ujednolicenia systemu zaimplementowano jeden wzór, który został zastosowany do każdego materiału. Wygląda on następująco:

$$\begin{aligned}\text{calcDiffuse}(\text{pos3D}) &= \text{mix}(\text{color1}, \text{color2}, \text{factor}) \\ \text{gdzie} \\ \text{factor} &= \text{clamp}(\text{noise}(\text{pos3D} * \text{posF}) * \text{valF}, 0, 1)\end{aligned}$$

Finalny kolor to mieszanka color1 i color2 w stosunku $1 - \text{factor}$ i factor . Sam factor to odpowiednio obliczona wartość szumu Perlina [4]. Dzięki posF możemy skalować szum, a dzięki valF możemy wymuszać faworyzowanie wartości bliższych 0 albo 1.

Jednym 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 aplikowany jest tylko kolor skał. Pośrodku dochodzi do mieszania kolorów w stosunku zależnym od odległości do ustalonych granic:

$$\begin{aligned}\text{mix}(\text{calcRockColor}(\text{pos}), \text{calcGroundColor}(\text{pos}), \text{groundFactor}) \\ \text{gdzie} \\ \text{groundFactor} &= (\text{groundEndY} - \text{posY}) / (\text{groundEndY} - \text{rockStartY})\end{aligned}$$

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

2.4. Rozmieszczenie roślin

Po wygenerowaniu wystarczającej liczby modeli roślin trzeba je umieścić w świecie. Można to robić w mniej lub bardziej wyszukany sposób. W tym przypadku zastosowano jedno z najprostszych rozwiązań – losowe rozmieszczenie wewnątrz komórek kraty. Krata o wymiarach $N \times M$ 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 pozostać wylosowana. Przykładowo dla wartości 0 obiekt zawsze będzie umieszczany w środku kraty, a dla 1 może pojawić się w całej kracie.

2.5. 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, która renderuje obraz. Duże modele będą zasłaniać znaczną część pola widzenia ukrywając niewypełnioną przestrzeń. Rozwiążanie nie jest idealne ale ma szanse wygenerować scenę o zadowalającym rezultacie. To podejście ma swoje odzworowanie w implementacji rozmieszczenia roślin, które jest robione zależnie od położenia kamery.

Rozdział 3.

Realistyczny rendering

Do tej pory przyjrzaliśmy się technikom pozwalającym generować różne modele składające się w jeden wielki świat. W tej sekcji skupimy się jak uchwycić część tego co udało się stworzyć. Sam proces zwany renderingiem w tym przypadku przypominać będzie nic innego jak zrobienie zdjęcia. Wcześniej pojawiło się pojęcie kamery. Jest to niewidoczny obiekt umieszczony w świecie, zwrócony w określonym kierunku. To co należy zrobić to uchwycić obraz z jego perspektywy.

3.1. 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* [5]. Samo równanie korzystając z zasady zachowania energii wylicza radiancję światła wychodzącą z ustalonego punktu na płaszczyźnie dla zadanego kierunku. Wygląda ono następująco:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\Omega} f_r(p, \omega_o, \omega_i) * L_i(p, \omega_i) * \cos\theta_i * d\omega_i$$

gdzie:

p – punkt przecięcia promienia z płaszczyzną

ω_o – kierunek wychodzącego światła

ω_i – ujemny kierunek przychodzącego światła

Ω – półsfera położona na płaszczyźnie zawierającej wszystkie wartości ω_i

$L_o(p, \omega_o)$ – radiancja spektralna wychodząca z punktu p w kierunku ω_o

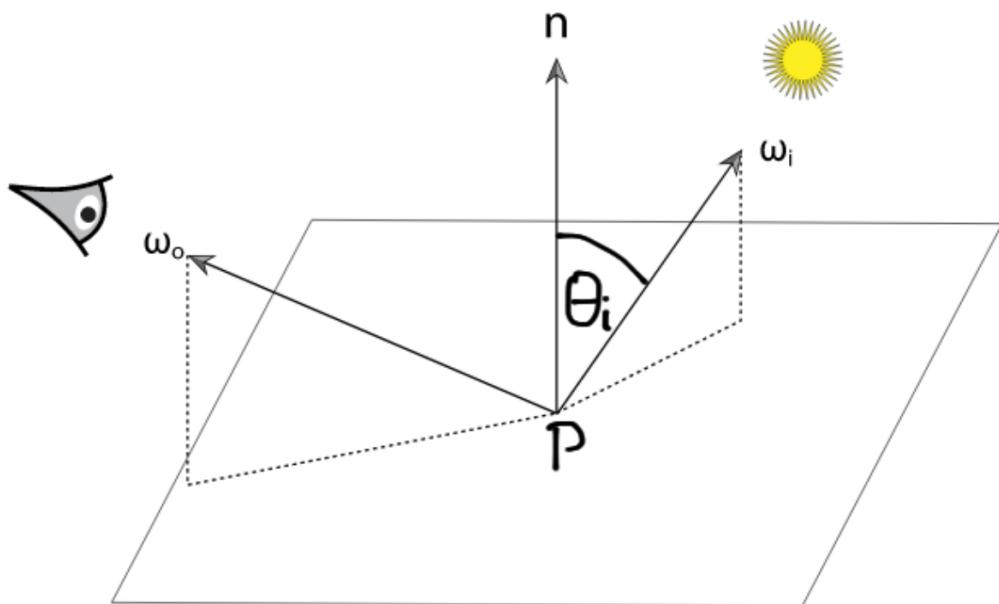
$L_e(p, \omega_o)$ – radiancja spektralna emitowana przez płaszczyznę w kierunku ω_o z punktu p

$f_r(p, \omega_o, \omega_i)$ – dwukierunkowa funkcja rozkładu odbicia (BRDF)

$L_i(p, \omega_i)$ – radiancja spektralna przychodząca z kierunku ω_i do punktu p

$\cos\theta_i$ – współczynnik osłabiający zależny od kierunku przychodzącego promienia ω_i i wektora normalnego płaszczyzny

Rysunek 3.1 przedstawia sytuację obliczania radiancji dla ustalonego promienia przychodzącego.



Rysunek 3.1: Sytuacja obliczania radiancji dla ustalonego promienia przychodzącego [5].

3.2. Niebo

Path tracing często stosuje się dla zamkniętych scen z kilkoma źródłami światła. W przypadku renderowania krajobrazów mamy do czynienia z czymś zupełnie innym – jedno źródło światła (słońce) oraz pół otwarta scena z wielkim niebem pochłaniającym większość promieni. Fizycznie poprawny model nieba został opisany

w pracy A. J. Preetham'a, Peter Shirley'a i Brian Smits'a pod tytułem *A Practical Analytic Model for Daylight* [6], która powstała na Uniwersytecie w Utah. Model opisuje niebo w danym miejscu na ziemi o ustalonej porze.

Został również udostępniony gotowy kod implementujący zaprezentowane rozwiązanie. Dołączony do programu, wykorzystywany jest jak zewnętrzna biblioteka bez wnikania w szczególne implementacyjne. Niebo w programie to półsfera rozciągająca się nad całym wygenerowanym terenem. Implikuje to, że im większy jest teren, tym większe jest niebo.

3.3. 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 realistyczne cienie obiektów dołożono dodatkowo słońce jako źródło światła o dużej mocy. Umieszczane jest zgodnie z wyliczeniami z modelu nieba. Znajduje się na półsferze nieba stąd odległość słońca od ziemi jest zależna od wielkości terenu (patrz poprzednia sekcja).

3.4. 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 uwzględnić 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 znajduje się miejsce, które nas interesuje. Następnie obliczamy wysokość tego miejsca i korygujemy nasz kierunek patrzenia względem otrzymanej wartości.

Rozdział 4.

Poradnik użytkowania

W poprzednich dwóch rozdziałach opisano rozwiązania zastosowane w programie. Ten skupi się jak go uruchomić oraz przedstawi w jaki sposób wpływa na rendering i generowane elementy.

4.1. Uruchomienie programu

Program był pisany i testowany na systemie operacyjnym *Ubuntu 18.04*. Użyty język programowania to *C++14*. Repozytorium można sklonować z adresu: <https://github.com/bratek20/RPForest>.

Program korzysta z następujących narzędzi i bibliotek:

- *cmake*
- *GLM*
- *GLEW*
- *GLFW3*
- *Embree3*
- *OpenEXR*

W celu instalacji programu należy uruchomić skrypt *install.sh* w głównym katalogu. Skrypt instaluje wszystkie wymagane biblioteki oraz przydatne narzędzia. Na końcu uruchamia skrypt budujący program – *build.sh*. Tworzy on folder *build*, który posłuży jako miejsce dla plików programu *cmake* i go uruchamia. Na końcu w folderze *bin* pojawi się plik binarny programu – *rpproject.exe*. Należy uruchomić go z linii poleceń w folderze *bin*. Program posiada dwa opcjonalne argumenty wywołania.

Pierwszy to nazwa folderu z plikami konfiguracyjnymi. Katalog musi znajdować się w folderze *configs*. Znajduje się tam domyślna konfiguracja *default*, która jest wczytywana jeśli użytkownik nie poda żadnego argumentu.

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.

4.2. Tryb zwykły i debugowania

Program ma dwa tryby: zwykły i debugowania.

Tryb zwykły po uruchomieniu od razu generuje finalny obraz i kończy działanie.

Tryb debugowania otwiera okno wyświetlające wygenerowany świat. Widzimy go z pozycji kamery. Możemy poruszać się w świecie w celu wyboru lepszego miejsca renderingu i wygenerować finalny obraz klikając odpowiedni przycisk na klawiaturze. Wyświetlany obraz w trybie debugowania różni się od generowanej finalnego obrazu. Wynika to z faktu zastosowania zupełnie innej technologii, w celu płynnego wyświetlania świata w czasie rzeczywistym.

Pełny opis możliwych akcji:

- zmiana kierunku patrzenia kamery – ruch myszką
- poruszanie kamerą – W (przód), S (tył), A (lewo), D (prawo) lub odpowiednie strzałki
- rendering aktualnego widoku – przycisk P
- wypuszczenie testowych promieni – przycisk L. Po naciśnięciu przycisku w kierunku patrzenia kamery, zostanie wystrzelone kilka promieni. Liczba promieni oraz liczba odbić każdego z nich jest zależna od ustawień path tracingu.

4.3. 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.

Wszystkie pliki powinny znajdować się w katalogu umieszczonym w folderze *configs*, którego nazwa została podana jako pierwszy argument programu. Ich rozszerzenia oraz umieszczenie w podfolderach ma znaczenie.

Poniżej zostaną opisane parametry poszczególnych plików konfiguracyjnych, ich lokalizacja oraz zastosowanie. Przyjęta konwencja to:

- *NazwaParametru : typ = wartość domyślna jeśli istnieje* – opis

Jeśli nie jest sprecyzowana nazwa pliku, a jedynie rozszerzenie to znaczy, że może być wiele plików konfiguruujących daną funkcjonalność. W takim przypadku użytkownik może tworzyć nowe pliki lub usuwać istniejące.

4.3.1. Ustawienia path tracingu

Plik opisujący path tracing umieszczony jest w głównym folderze konfiguracyjnym i ma nazwę *pathTracer.conf*.

Można zdefiniować następujące parametry:

- *PhotoName : string* – nazwa pliku do którego zostanie zapisany finalny render. Będzie on w formacie exr i umieszczony zostanie w folderze *photos*.
- *Resolution : vec2* – rozmiar renderowanej fotografii
- *Samples : int = 1* – liczba promieni wystrzelona per jeden pixel
- *MaxRayBounces : int = 5* – maksymalna dopuszczalna liczba odbić jednego promienia

4.3.2. Ustawienia kamery

Plik opisujący ustawienia kamery z głównego folderu: *camera.conf*.

Parametry:

- *LookHeight : float = 1* – wysokość liczona od powierzchni ziemi
- *LookDirection : vec3 = 0 0 1* – kierunek patrzenia
- *LookDistance : float = 2* – odległości od kamery, względem której zostanie dostosowane pochylenie kamery (patrzenie pod góre/w dół)
- *EdgeMinOffset : float = 10* – minimalna odległość od krawędzi świata
- *ExpectedPositionY : float = 0* – oczekiwana pozycja Y, na której powinna zostać umieszczona kamera. Dzięki temu można wymuszać np. robienie zdjęć ze szczytów gór poprzez ustawienie bardzo dużej wartości (kamera zawsze będzie *LookHeight* nad ziemią).

4.3.3. Ustawienia debugowania

Plik do zmiany ustawień debugowania znajdujący się w głównym folderze: *debug.conf*.

Parametry:

- *Enabled* : *bool* = *false* – czy jest włączony tryb debugowania
- *Resolution* : *vec2* = *800 600* – rozmiar okna debugowania
- *CameraVelocity* : *float* = *10* – prędkość poruszania się kamery

4.3.4. Ustawienia nieba

Plik do konfiguracji nieba z głównego folderu: *sky.conf*. Większość parametrów służy do ustawień modelu nieba opracowanego w pracy z Uniwersytetu w Utah [6].

Parametry słońca:

- *SunColor* : *vec3* – kolor słońca
- *SunPower* : *float* – moc słońca

Parametry modelu nieba:

- *SkyLuminanceFactor* : *float* – mnożnik do zmniejszania jasności pochodzącej od nieba
- *Latitude* : *float* – szerokość geograficzna
- *Longitude* : *float* – długość geograficzna
- *StandardMeridian* : *int* – numer strefy czasowej
- *JulianDay* : *int* – dzień
- *TimeOfDay* : *float* – czas
- *Turbidity* : *float* – zagęszczenie mgły

4.3.5. Ustawienia terenu

Ustawienia terenu wpływające na algorytm *diamond-square* oraz teksturowanie są opisane w pliku: *terrain.conf* umieszczonym w głównym folderze.

- *N* : *int* – wpływa na liczbę elementów mapy wysokości. Finalny rozmiar to $2^N + 1 \times 2^N + 1$.

- *Size* : *float* – wielkość generowane terenu
- *InitHeight* : *float* – wartość do wyliczania początkowych wartości na rogach mapy. Zostaną wylosowane z przedziału ($-InitHeight : InitHeight$).
- *Spread* : *float* – początkowa rozpiętość losowej wysokości dodawanej do średnianych wartości. Losowa wysokość losowana jest z przedziału ($-Spread : Spread$), gdzie *Spread* będzie maleć z każdą iteracją algorytmu.
- *SpreadReductionRate* : *float* – jak zmienia się rozpiętość względem kolejnych iteracji algorytmu. Parametr powinien mieć wartość mniejszą niż 1.
- *RockStartY* : *float* = 10 – od jakiej wysokości powinien być aplikowany materiał skał
- *GroundEndY* : *float* = 20 – do jakiej wysokości powinien być aplikowany materiał gruntu. Parametr powinien mieć wartość większą niż *RockStartY*.

4.3.6. Ustawienia tekstur

Wszystkie pliki konfiguracyjne materiałów znajdują się w folderze *materials* i mają rozszerzenie *.mat*. W plikach ustala się wartości parametrów ze wzoru 2.3..

- *Color1* : *vec3* – pierwszy kolor do mieszania
- *Color2* : *vec3* – drugi kolor do mieszania
- *NoisePositionFactor* : *float* = 1 – czynnik zmieniający pozycję obliczania szumu
- *NoiseValueFactor* : *float* = 1 – czynnik zmieniający wartość szumu

4.3.7. Bazowe ustawienia generatorów

Program pozwala definiować własne L–systemy oraz udostępnia kilka predefiniowanych. Oba podejścia konfigurowane są przez odpowiednie pliki, które wspólnie dzielą część parametrów:

- *N* : *int* – liczba iteracji aplikowania produkcji L–systemu
- *Height* : *float* – finalna wysokość wygenerowanego modelu
- *ConeBasePoints* : *int* = 3 – ile punktów w podstawie mają cylindry tworzące model
- *Material* : *string* – nazwa materiału (Leaf, Bark, Plant)

- *InitRadius : float* – początkowy promień cylindrów tworzących model. Finalnie model jest przeskalowany tak by miał wysokość *Height*, więc ten parametr służy do zmieniającego stosunku grubości do wysokości.

4.3.8. Definiowanie własnych L–systemów

Wszystkie pliki definiujące własne L–systemy mają rozszerzenie *.lsys*. Można umieścić je w jednym z trzech folderów: *trees*, *plants* albo *leafs*. Umieszczenie wpływa na to jak program będzie traktować dany L–system.

- *TurnAngle : float = 0* – kąt skrętu żółwia w lewo/prawo
- *PitchAngle : float = 0* – kąt skrętu żółwia w dół/górę
- *RollAngle : float = 0* – kąt skrętu wokół własnej osi
- *Axiom : char* – początkowy symbol
- *Prod : char > string* – pojedyncza definicja produkcji. Plik może zawierać kilka takich wpisów (każdy w osobnej linii) i wszystkie zostaną uwzględnione. Poprawny zapis to symbol, znak *>* (dla lepszej czytelności) i produkowane słowo np. *Prod A > ABC*.

Dostępne symbole sterujące żółwiem to:

- *F* – idź w przód tworząc cylinder
- *f* – idź w przód
- *[* – rozpoczęj tworzenie gałęzi
- *]* – zakończ tworzenie gałęzi
- *+* – obróć się w prawo o *TurnAngle*
- *—* – obróć się w lewo o *TurnAngle*
- *^* – obróć się w górę o *PitchAngle*
- *&* – obróć się w dół o *PitchAngle*
- *\-* – obróć się wokół osi zgodnie z ruchem wskazówek zegara o *RollAngle*
- */* – obróć się wokół osi przeciwnie do ruchu wskazówek zegara o *RollAngle*
- *|* – obróć się w lewo o 180°
- *{* – zacznij tworzyć wielokąt
- *.* – zapisz aktualną pozycję jako wierzchołek wielokąta
- *}* – skończ tworzyć wielokąt

4.3.9. Ustawienia predefiniowanych L–systemów

W programie zostały zaimplementowane dwa L–systemy: jeden do generowania drzew, drugi do liści.

Pliki konfiguracyjne do generowania drzew mają rozszerzenie *.honda* i muszą znajdować się w folderze *trees*. Opisują one wartość L–systemu wymyślonego przez Honde [2] i zaprezentowanego przez Prusinkiewicza [1] – patrz Rysunek 2.3.

Parametry z rysunku z dokładnością do wielkości liter:

- $R1 : float$ – współczynnik zmniejszania długości pnia
- $R2 : float$ – współczynnik zmniejszania długości gałęzi
- $A0 : float$ – kąt pod którym rosną gałęzie
- $A2 : float$ – kąt pod którym wyrastają pomniejsze gałęzie
- $D : float$ – kąt o ile rotuje wokół własnej osi żółw po stworzeniu gałęzi
- $Wr : float$ – współczynnik zmniejszania promienia pnia i gałęzi

Przykładowy L–system tworzący liście został przedstawiony na rysunku 2.4. Pliki go opisujące mają rozszerzenie *.family* i umieszczone są w folderze *leafs*. Zawierają:

- $Delta : float$ – kąt skrętu żółwia w lewo/prawo
- $LA : float$ – początkowa długość głównego segmentu
- $RA : float$ – współczynnik wzrostu głównego segmentu
- $LB : float$ – początkowa długość bocznych segmentów
- $RB : float$ – współczynnik wzrostu bocznych segmentów
- $PD : float$ – ubytek rosnienia bocznych fragmentów

4.3.10. Ustawienia rozmieszczania roślin

Grupy elementów można umieszczać w świecie przy pomocy plików o rozszerzeniu *.spawner* umieszczonego w folderze *sprawners*.

- $CameraOffset : vec2$ – przesunięcie środka kraty względem pozycji kamery
- $XGridN : int = 1$ – liczba komórek kraty wzdłuż osi X
- $ZGridN : int = 1$ – liczba komórek kraty wzdłuż osi Z

- *CellSize : float = 0* – rozmiar pojedynczej komórki
- *GeneratorType : string* – typ użytego generatorów (Tree, Plant)
- *SpawnProbability : float = 1* – szansa na stworzenie pojedynczego elementu
- *RandomPositionPercentage : float = 0.5* – na jak dużym obszarze wokół środka komórki mogą być tworzone elementy
- *MinElementScale : float = 0.5* – minimalna wartość, która może zostać wybrana do skalowania pojedynczego elementu
- *MaxElementScale : float = 2* – maksymalna wartość, która może zostać wybrana do skalowania pojedynczego elementu

Rozdział 5.

Szczegóły programistyczne

5.1. Wykaz użytych narzędzi

Program napisany jest w języku **C++**. Można w nim pisać obiektowo, a zatem dbać o efektywność programu. Dostępne jest wiele bibliotek graficznych, które są dobrze udokumentowane. Z uwagi na popularność języka w przypadku jakichkolwiek trudności, łatwo można znaleźć materiały z propozycją rozwiązania problemu.

Większość obliczeń matematycznych korzysta z typów i operacji zawartych w bibliotece **GLM**. Jest to mała i prosta w użyciu biblioteka zawierająca wszystkie przydatne funkcje matematyczne. Udostępnia również typy takie jak *vec3* lub *mat4*, które są często używane w grafice komputerowej.

Tryb debugowania korzysta z OpenGL w celu wyświetlania świata. Użyte biblioteki to **GLEW** oraz **GLFW3**. Są one ogólnie dostępne, dobrze udokumentowane i łatwe w użyciu.

Path tracer korzysta z biblioteki **Embree3**. Pozwala ona w bardzo szybki sposób obliczać przecięcia promieni z trójkątami, co jest kluczowym elementem path tracingu.

Do tworzenia obrazów w formacie *.exr* wykorzystana została biblioteka **OpenEXR**. Format zapisu pozwala na manipulowanie wieloma parametrami wygenerowanego obrazu w celu poprawienia wyglądu. Pliki w tym formacie można wyświetlić np. przy pomocy programu *exrdisplay*.

5.2. Struktura projektu

Program składa się z czterech modułów:

- *engine* – baza całego programu, przypominająca silnik 3D. Implementuje podstawowe funkcjonalności wykorzystywane i rozszerzane przez inne moduły.

- *rendering* – moduł implementujący wszystkie zagadnienia związane z path tracingiem
- *generating* – moduł implementujący wszystkie zagadnienia związane z generowaniem proceduralnym
- *utils* – moduł implementujący niezależne lub pomocnicze funkcjonalności

5.2.1. Moduł *engine*

Zarówno rendering przy pomocy OpenGL jak i techniką path tracingu wymaga reprezentacji obiektu, który ma renderować. Podstawowym sposobem opisów złożonych kształtów jest przedstawienie ich przy pomocy zbioru trójkątów. Każdy trójkąt (klasa **Triangle**) składa się z trzech wierzchołków (klasa **Vertex**), które poza pozycją w świecie zawierają takie informacje jak wektory normalne (reprezentujące orientację w przestrzeni). Pojedynczy zbiór trójkątów trzymany jest w obrębie obiektu klasy **Mesh**, która dodatkowo definiuje kolorystykę reprezentowanego kształtu przy pomocy materiału (klasa **Material**). Finalny model (klasa **Model**) to zbiór mesh'ów. Pozwala to wyrazić dowolnie skomplikowany (pod względem kształtu i kolorystki) obiekt.

Jedną z najważniejszych klas modułu jest **Actor**. Reprezentuje on dowolny obiekt w świecie. Aktor może mieć kształt opisany przy pomocy klasy **Model**. Posiada on dodatkowo zbiór parametrów pozwalający wpływać na finalną reprezentację np. pozycja, skala, rotacja. Dzięki temu możemy umieścić ten sam model w różnych miejscach w świecie i zmienić jego rozmiar.

Cały świat reprezentowany przy pomocy klasy **Scene**. Scena to rodzaj aktora, który zbiera wszystkich innych aktorów, odpowiednio ich inicjalizuje i rozmieszcza. Jednym z aktorów umieszczonych na scenie jest między innymi kamera (klasa **Camera**). Scena wykorzystuje funkcjonalność aktora: relacja rodziniec-dziecko. Każdy aktor może mieć niezerową liczbę dzieci. Wszystkie parametry rodzica wpływają na dziecko np. zmieniając pozycję rodzica, przemieścimy wszystkie jego dzieci oraz dzieci tych dzieci (wnuki) itd. Dodatkowo akcje jak aktualizowanie (funkcja **update**) albo renderowanie w trybie debugowania (funkcja **render**) działają pierw na rodzica, a następnie na jego wszystkie dzieci.

Moduł implementuje dodatkowo wiele klas, bez których program nie mógłby funkcjonować:

- **Window** – wyświetla okno trybu debugowania
- **Input** – zbiera akcje użytkownika i pozwala na nie reagować
- **Assets** – wczytuje zasoby potrzebne programowi (np. pliki konfiguracyjne) i udostępnia je innym częścioom programu

5.2.2. Moduł *rendering*

Najważniejsza klasa modułu to **PathTracer**. Implementuje ona obliczanie koloru pochodzącego z danego kierunku. Korzysta z dwóch pomocniczych klas:

- **AccStruct** – interfejs struktury obliczającej przeciecie danego promienia z najbliższym trójkątem. Przykładowa implementacja to **EmbreeWrapper**, która wykorzystuje bibliotekę Embree do obliczania przecięć.
- **SkyLightSampler** – klasa obliczająca kolor nieba oraz próbująca promienie pochodzące od słońca

Dodatkowo w głównym folderze modułu znajduje się podfolder *ASky*, zawierający implementacje modelu nieba udostępnioną przez pracowników z Uniwersytetu z Utah [6].

5.2.3. Moduł *generating*

Implementacje wszystkich zagadnień opisanych w pracy znajdują się w tym folderze. Duża część kodu zawiera implementację różnych symboli z L-systemów. Bazą jest klasa **Symbol**, która udostępnia dwie metody wirtualne: **vector <SymbolPtr> produce()** oraz **void process(ProcessContext&)**. Pierwsza metoda służy do produkcji słowa dla aktualnego stanu symbolu. Pozwala to implementować zarówno parametryczne jak i losowe symbole. Druga metoda służy do wpływania na aktualny stan przetwarzania słowa z wygenerowanego L-systemu. Operuje ona na obiekcie struktury **ProcessContext**. Struktura składa się między innymi: aktualny stan żółwia (klasa **Turtle**), stos odłożonych stanów żółwia (przy pomocy operatora rozgałęziania) oraz wygenerowany do tej pory model.

Samo generowanie odbywa się w klasach pochodnych klasy **Generator**. Generator wymaga symbolu startowego oraz konfiguracji bazowej **GeneratorConfig**, aby wygenerować finalny model.

Przykładowe klasy pochodne generatora to **LSysGenerator** albo **Honda**. Pierwszy implementuje obsługę plików konfiguracyjnych *.lsys*, a drugi prezentuje implementację predefiniowanego L-systemu.

Pozostałe klasy, na które warto zwrócić uwagę to:

- **DiamondSquareTerrain** – klasa implementująca *Diamond-square algorithm*. Udostępnia również metodę do obliczania wysokości terenu dla zadanego punktu oraz optymalnego umiejscowienia kamery.
- **SpawnerActor** – specjalny rodzaj aktora, który rozmieszcza wygenerowane modele w zależności od konfiguracji z pliku *.spawner*

5.2.4. Moduł *utils*

Główna klasa modułu to **Utils**. Jest to klasa statyczna zawierająca wszystkie pomocnicze funkcje, wykorzystywane przez inne moduły. Dodatkowo posiada wiele stałych jak np. **PI** albo **INF**.

Inne klasy modułu to:

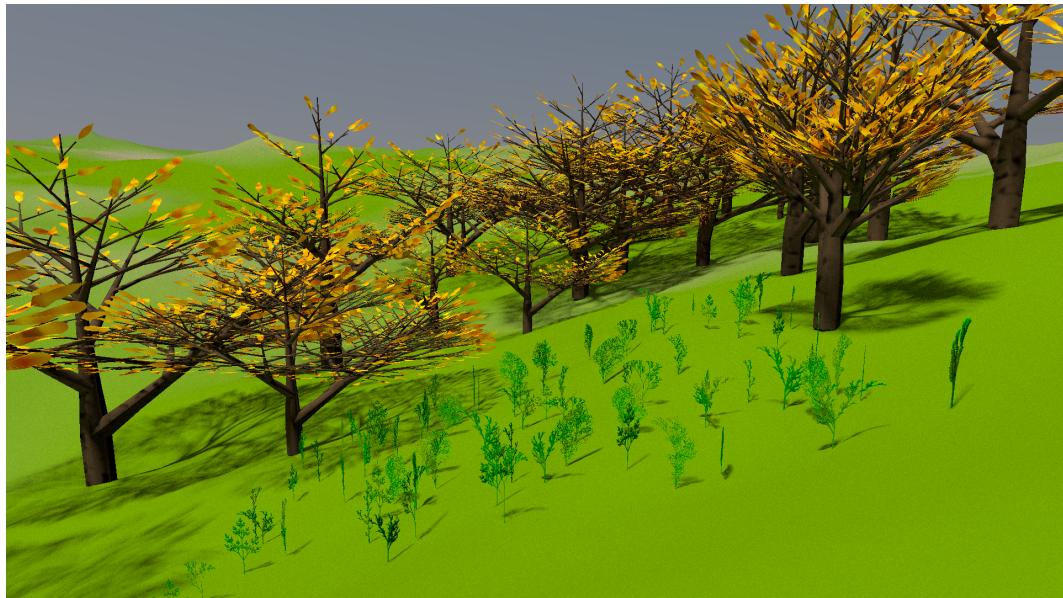
- **Random** – implementuje wiele losowych metod jak np. losowanie wektora na półsferze, obliczanie szumu perlina albo wybieranie elementu z tablicy
- **Timer** – prosta klasa do mierzenia czasu

Rozdział 6.

Zastosowanie

6.1. Przykłady użycia

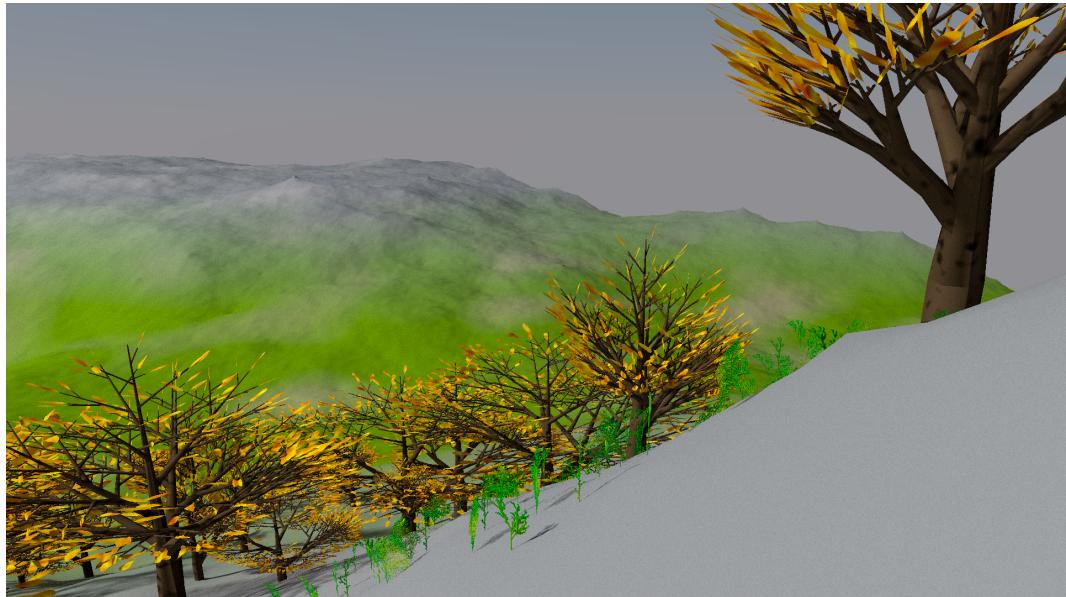
Program posiada wiele parametrów wpływających na generowany obraz. Konfiguracja *default* powstała, by potencjalny użytkownik nie musiał pisać wszystkiego od początku. Przykładowy wygenerowany obraz został przedstawiony na rysunku 6.1. Ustawienia path tracer'a to 256 próbek dla rozdzielczości 1920 x 1080 (patrz ustawienie path tracer'a 4.3.1.).



Rysunek 6.1: Przykładowy render dla konfiguracji *default*.

Zmieniając oczekiwana wysokość kamery (patrz konfiguracja kamery 4.3.2.) w łatwy sposób można otrzymać zdjęcia ze szczytu góry – rysunek 6.2. Możliwe, że początkowe ustawienie kamery nie będzie idealne. Często warto wymusić tryb de-

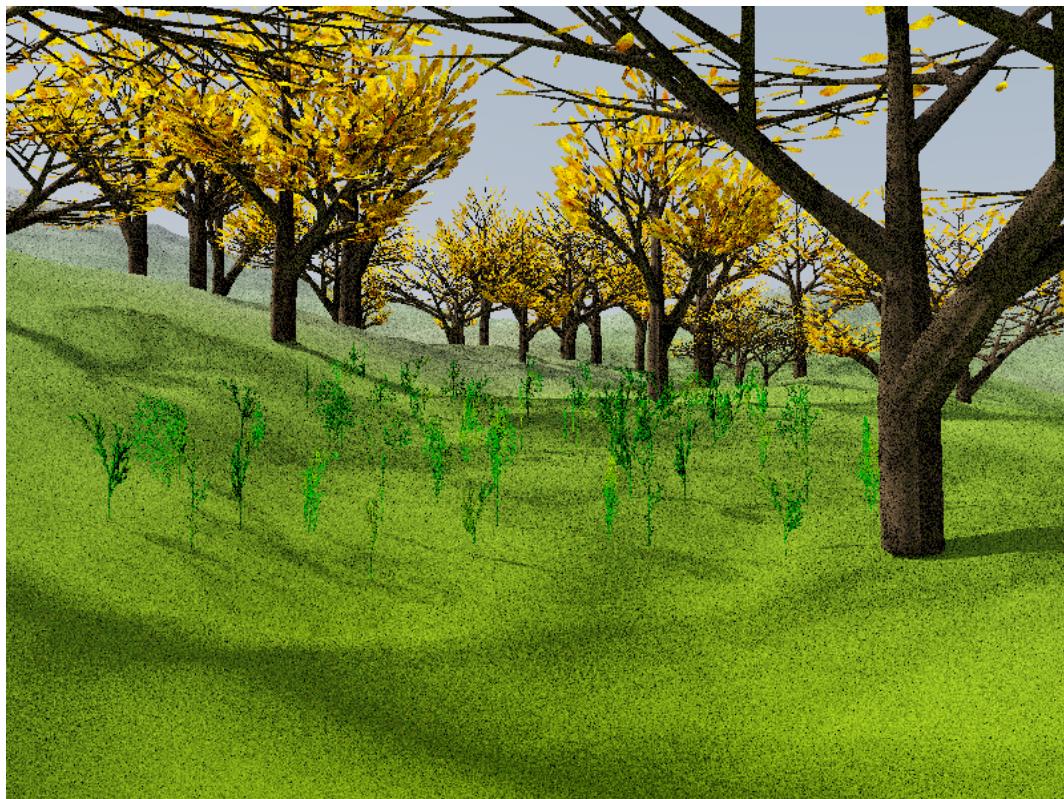
bugowania, aby ręcznie je poprawić (patrz konfiguracja trybu debugowania 4.3.3.).



Rysunek 6.2: Przykładowy render dla konfiguracji *default* z wymuszeniem kamery na szczytce góry.

Dobrą praktyką jest kopiowanie konfiguracji *default* w celu utworzenia nowej. W kopii możemy zmieniać interesujące nas parametry. W taki sposób powstała konfiguracja *debug* oraz *dead*. Pierwsza z nich ma mniejszą rozdzielcość oraz liczbę próbek w celu obniżenia czasu tworzenia obrazka. Zdjęcie ma widoczny szum ale generuje się kilka sekund – rysunek 6.3. Jest to przydatne do testowania różnych ustawień programu.

Konfiguracja *dead* to przykład jak w łatwy sposób otrzymać widocznie różny efekt od konfiguracji *default*. Zmieniona kolorystyka, brak liści oraz wymuszenie ustawiania drzew w ścieżkę sprawiają, że finalny obraz 6.4 znaczaco różni się od poprzednich.



Rysunek 6.3: Przykładowy render dla konfiguracji *debug*.



Rysunek 6.4: Przykładowy render dla konfiguracji *dead*.

Jednym z limitów programu jest liczba generowanych obiektów na scenie. Niska optymalizacja modeli tworzonych przy pomocy L–systemów wpływa na to, że każdy z nich składa się z kilkunastu tysięcy trójkątów. Warto mieć to na uwadze i na scenie tworzyć ich nie więcej niż 100.

6.2. Porównanie z komercyjnymi rozwiązaniami

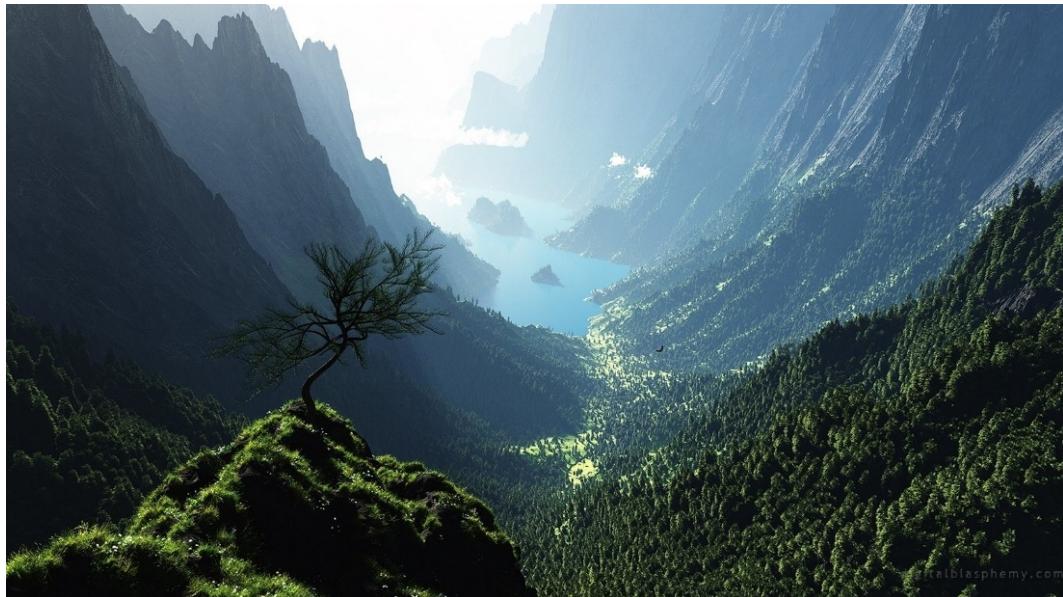
Generowanie i realistyczne renderowanie krajobrazów jest często potrzebne między innymi w grach lub przemyśle filmowym. Z tego powodu powstało wiele komercyjnych rozwiązań, które owy koncept dopracowały praktycznie do granic perfekcji.

Jednym z popularniejszych programów tego typu jest *Terragen* firmy *Planetside Software*. Na swojej stronie udostępniają wiele przykładów powstałych dzięki ich oprogramowaniu. Jednym z nich jest render stworzony przez Jeff'a Boser'a przedstawiony na rysunku 6.5.



Rysunek 6.5: Obraz stworzony przy pomocy programu *Terragen* przez Jeff'a Boser'a [7].

Inny program pozwalający osiągać bardzo realistyczne wyniki to *VUE* od firmy *e-on software*. Obraz Ryan'a Bliss'a 6.6 jest tego najlepszym przykładem.



Rysunek 6.6: Obraz stworzony przy pomocy programu *VUE* przez Ryan'a Bliss'a [8].

Program zaprezentowany w ramach tej pracy generuje zauważalnie słabsze obrazy. Ma problemy, które zostały już tam rozwiązane takie jak maksymalna liczba elementów na scenie, ich różnorodność i fotorealistyczny rendering. Warto jednak zauważyć, że rozwiązania wykorzystywane przez firmy komercyjne, bazują w dużej mierze na opisanych wcześniej zagadnieniach. Dobra ich znajomość poparta przykładową implementacją to dobry punkt startowy dla osób zainteresowanych tą tematyką.

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] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.
- [5] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016.
- [6] A. J. Preetham, Peter Shirley, and Brian Smits. A practical analytic model for daylight. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, page 91–100, USA, 1999. ACM Press/Addison-Wesley Publishing Co.
- [7] Jeff Boser Planetside Software. Terragen image gallery. <https://planetside.co.uk/terragen-image-gallery/>.
- [8] Ryan Bliss e-on software. Vue image gallery. https://info.e-onsoftware.com/gallery-environment#gallery_isotope.