Grzegorz Michalak

Algorytmy proceduralnego generowania rzeczywistości na przykładzie dwuwymiarowej gry cRPG

Praca magisterska
wykonana pod kierunkiem
dr. Tomasza Gwizdałły

Uniwersytet Łódzki
Wydział Fizyki i Informatyki Stosowanej
Łódź, 2012

Spis treści

1. Wstęp

Generowanie proceduralne jest dziedziną szeroko wykorzystywaną w produkcji mediów cyfrowych. Algorytmy tworzenia zasobów stosuje się głównie w programach przeznaczonych do manipulacji obrazem, jak również grach wideo. W pracy opisany został sposób generowania wszystkich istotnych elementów gier komputerowych wraz z użytymi w tym celu algorytmami.

1.1 Proceduralność oraz losowość

Element nazywany jest wygenerowanym proceduralnie jeżeli został stworzony automatycznie przez algorytmy komputerowe. Jako prosty przykład można przyjąć funkcję rysującą prostokąt o wymiarach `20x10px` w lewym górnym rogu ekranu. Prostokąt niegenerowany proceduralnie będzie najczęściej grafiką wczytywaną z pliku (np. `draw_image("prostokat.bmp")`) o wymiarach `20x10px`, natomiast prostokąt generowany proceduralnie będzie funkcją rysującą cztery połączone ze sobą pod kątem prostym linie (np. `draw_rectangle(0,0,20,10)`).

Podejście proceduralne umożliwia swobodną modyfikację parametrów procedur. Korzystanie w takim przypadku z generatorów liczb losowych jest bardzo intuicyjne. Nawiązując do powyższego przykładu, nie jest problemem wygenerowanie prostokąta o losowej szerokości. Jeżeli przyjmiemy, że funkcja `random(x,y)` zwraca liczbę całkowitą z przedziału `<x,y>`, to można ją wykorzystać do wywołania funkcji w postaci np. `draw_rectangle(0,0,random(10,30),10)`.

Użytym w grze generatorem liczb pseudolosowych jest funkcja `rand()` należąca do standardowej biblioteki języka C++. Zwraca ona całkowitą liczbę z przedziału `<0,RAND_MAX>`. `RAND_MAX` jest stałą zdefiniowaną w `<cstdlib>`. Jej wartość domyślna zależna jest od implementacji lecz nie może być mniejsza niż `32767`.

Algorytm korzysta z ziarna losowości, które powinno być inicjalizowane określoną wartością poprzez funkcję `srand(int)`. Dla każdej różnej wartości ziarna, generator liczb pseudolosowych tworzy inne sekwencje liczb przy wywołaniu funkcji `rand()`. Dwie różne inicjalizacje `srand()` z podaną tą samą wartością ziarna losowości zapewniają uzyskanie tych samych sekwencji liczb przy kolejnych wywołaniach `rand()`. Dlatego każdy obiekt występujący w grze przechowuje swoje ziarno by móc zostać tak samo odwzorowanym graficznie przy ponownym pojawieniu się na ekranie.

1.2 Zastosowania generatorów proceduralnych

Najczęściej spotykanymi elementami wygenerowanymi proceduralnie są:

- **tekstury** – efekty wizualne, takie jak szum Perlina[1], są szeroko wykorzystywane w produkcji filmów z efektami specjalnymi (pierwsze zastosowanie w filmie Tron z 1982 roku) oraz w dziedzinie gier komputerowych,
- **siatki modeli 3D** – proceduralność pozwala na bardzo szybkie tworzenie rozległych terenów i obszarów miejskich[2],
- **dźwięk** – spotykany zarówno w syntezie mowy jak i komputerowej kompozycji melodycznej[3] (Stephen Hawking korzysta z systemu generowania mowy, który umożliwia mu komunikację),
- **fraktale[4]** – wizualizacja odpowiednich wzorów matematycznych pozwala uzyskać obrazy reprezentujące obiekty samo-podobne.

Wykorzystywanie generatorów w grach komputerowych jest zjawiskiem powszechnym. Zwykle są to zasoby takie jak:

- **struktura poziomów** - tworzenie labiryntów dla produkcji typu hack and slash (*Diablo II*[5] wydana w 2000 roku ma swoich aktywnych fanów po dziś dzień, natomiast *Minecraft*[6], z tworzonym pseudolosowo światem, zdobył ponad milion nabywców w przeciągu kilku miesięcy),
- **elementy fabuły** - głównie spotykane w grach typu cRPG (począwszy od *The Elder Scrolls II: Daggerfall*[7]),
- **animacja postaci** - system automatycznego tworzenia animacji stworów w *Spore*[8],
- **fauna i flora** - popularny system *SpeedTree* pozwala wygenerować losowe, niepowtarzające się drzewa (z aplikacji tej korzystają zarówno czołowi producenci gier komputerowych na świecie jak i przemysł filmowy, np. James Cameron w filmie *Avatar*),
- **rozmieszczenie przedmiotów** - *Borderlands* - losowe generowanie znajdywanych broni i przedmiotów[9],
- **liczba przeciwników** - innowacyjny moduł AI Director[10] na podstawie osiągnięć gracza automatycznie dopasowuje do niego poziom trudności gry (*Left 4 Dead*),
- **efekty pogodowe** - animacja fal na wodzie, śladów na śniegu, bądź spadających kropli deszczu również przebiega proceduralnie[11] (*The Elder Scrolls III: Morrowind* korzysta z systemu “Water Interaction” promowanego swego czasu przez NVIDIA).

Osobnym, lecz bardzo mało popularnym, gatunkiem gier komputerowych są produkcje generowane w całości proceduralnie. Sztandarowym dziełem owej dziedziny jest *.kkrieger*[12]

[1] http://www.noisemachine.com/talk1/
[2] http://www.procedural.com/, http://www.planetside.co.uk/
[3] http://www.animenewsnetwork.com/feature/2011-07-15/
[4] http://chaospro.de
[5] http://diablo2.diablowiki.net/Diablo_Levels
[6] http://notch.tumblr.com/post/3746989361/terrain-generation-part-1
[7] http://www.uesp.net/wiki/Daggerfall:Quest_hacking_guide
[8] http://www.chrishecker.com/Real-time_Motion_Retargeting_to_Highly_Varied_User-Created_Morphologies
[9] http://borderlands.wikia.com/wiki/Lootable_object
[10] http://left4dead.wikia.com/wiki/The_Director
[11] http://exoed.com/source-engine-daynight-cycle-dynamic-weather/1330/
[12] http://www.theprodukkt.com/kkrieger

- gra akcji FPP, podobna do *Quake III Arena*, zajmująca na dysku 96kB, gdyż jedynym zasobem jaki posiada, jest jej kod wykonywalny.

Zakres wykorzystywania elementów generowanych proceduralnie zmienia się wraz z rozwojem technologii informatycznych. Kiedy rozmiary pamięci komputerowej były bardzo ograniczone, generowanie proceduralne pozwalało zredukować wielkość aplikacji, głównie poprzez losowe tworzenie poziomów gry.

Wraz z rosnącą mocą obliczeniową komputerów, producenci gier ukierunkowali się na ręczne tworzenie zawartości. Wszelkie modele i tekstury są tworzone przez grafików, a muzyka i efekty nagrywane w studiach, umożliwiając pełną kontrolę nad produktem wynikowym. Drugą stroną takiego rozwiązania jest powtarzalność - za każdym razem kiedy uruchomimy grę otrzymujemy tę samą zawartość. Taki sposób produkcji gier kładzie zdecydowany nacisk na pracę grafików, natomiast coraz mniejszą wagę przykłada się do rozwiązań algorytmicznych, przez co wydawanych jest coraz więcej tytułów klasyfikowanych jako *visual-novel*[13], które dalece odstępują od klasycznej definicji gry komputerowej.

Obecnie generatory stosuje się w ograniczonym zakresie, jednak twórcy w zdecydowanej większości produkcji wprowadzają czynnik losowy aby urozmaicić rozgrywkę bądź dostosować ją do umiejętności graczy. Stąd też proceduralną zawartość można spotkać zarówno w produkcjach klasy AAA, jak i grach niezależnych.

Począwszy od rozdziału 3. analizowane są sposoby generacji kolejnych elementów gry. Nazwa rozdziału jest równocześnie nazwą komponentu jaki opisuje, natomiast podrozdziały służą uporządkowaniu informacji na temat algorytmicznego tworzenia danego elementu.

[13] Chris Klug, Josiah Lebowitz: Interactive Storytelling for Video Games: A Player-Centered Approach to Creating Memorable Characters and Stories. Burlington, MA: Focal Press. pp. 194–7. ISBN 0-240-81717-6.

2. Cel

Celem pracy jest analiza zalet i wad elementów generowanych proceduralnie w grach. Źródłem analizy jest w pełni funkcjonalna gra cRPG (*computer Role Playing Game* - Komputerowa gra fabularna korzystająca z tej samej mechaniki i terminologii co tradycyjne gry RPG), z zasobami wygenerowanymi w całości proceduralnie, utworzona specjalnie na potrzeby pracy.
Do tej pory nie stworzono komputerowej gry fabularnej, która w ten sposób zapewniała by jednocześnie grywalność oraz niepowtarzalność elementów.

2.1 Implementacja algorytmów

W celu stworzenia niezbędnych zasobów do gry, należy wykorzystać szereg algorytmów generujących, m.in.:

- lokacje:
  - świat (lądy, wody, strefy klimatyczne),
  - otwarte tereny (lasy, rzeki, skały),
  - miasta, w tym:
    - budynki,
    - drogi,
  - labirynty,
- postacie:
  - bohater gry,
  - postacie niezależne,
- przedmioty:
  - uzbrojenie,
  - przedmioty jednorazowego użytku,
  - elementy wystroju wnętrz (meble, dekoracje),
- zjawiska pogodowe (deszcz oraz śnieg),
- grafikę (wyżej wymienione elementy muszą posiadać swoją reprezentację graficzną na ekranie),
- nazwy (dla miast, bohaterów niezależnych, czy przedmiotów),
- przebieg fabuły (drzewa dialogowe dla różnych zadań).

2.2 Wybrane środowisko programistyczne

2.2.1 Język programowania

Wybranym językiem programowania użytym do implementacji pisanej gry jest C++. Programowanie obiektowe jest obecnie używanym standardem w pisaniu kodu gier komputerowych, a język ten doskonale wspiera takie podejście.

Kolejną zaletą języka C++ jest możliwość programowania niskopoziomowego z jawnym dostępem do pamięci. Umożliwia to optymalizację instrukcji generatorów oraz minimalizację rozmiaru uzyskanej aplikacji.

2.2.2 Kompilator

Obecnie istnieją dwa liczące się kompilatory języka C++ - GCC (GNU Compiler Collection) oraz Visual C++. Pierwszy z nich jest kompilatorem rozwijanym na zasadzie wolnego oprogramowania i powszechnie wykorzystywanym do tworzenia programów głównie w środowisku Linux. Visual C++ jest kompilatorem działającym w systemach rodziny Windows oraz domyślnie dostarczany w zintegrowanym środowisku programistycznym Visual Studio. Oba kompilatory są zgodne ze standardami języka C++ i zapewniają optymalny proces kompilacji programów.

Z uwagi na możliwości oferowane przez Microsoft Visual Studio 2010 Premium oraz możliwość korzystania z jego pełnej, legalnej wersji poprzez program akademicki DreamSpark Premium[14], wybranym kompilatorem został Visual C++.

2.2.3 Biblioteka graficzna

Wbrew pozorom nie istnieje wiele bibliotek graficznych wspomagających tworzenie gier z grafiką dwuwymiarową. Biblioteki takie jak SDL[15], czy SFML[16] oferują niskopoziomowy dostęp do funkcji multimedialnych, lecz przystosowanie ich do stworzenia gry zajęłoby nieotymalnie dużo czasu. Najlepszym wyjściem w tym przypadku okazuje się biblioteka Allegro[17] w wersji 5. Jest to platforma przystosowana do tworzenia gier 2D, a do jej głównych zalet można zaliczyć:

- akceleracje sprzętową przy użyciu karty graficznej,
- pełną kompatybilność na różnych platformach sprzętowych i systemach operacyjnych,
- możliwość niskopoziomowego dostępu do funkcji karty graficznej poprzez DirectX i OpenGL.

[14] www.microsoft.com/poland/edukacja/dreamsparkpremium_dla_studenta.aspx
[15] www.libsdl.org
[16] www.sfml-dev.org
[17] alleg.sourceforge.net

3. Mapa świata

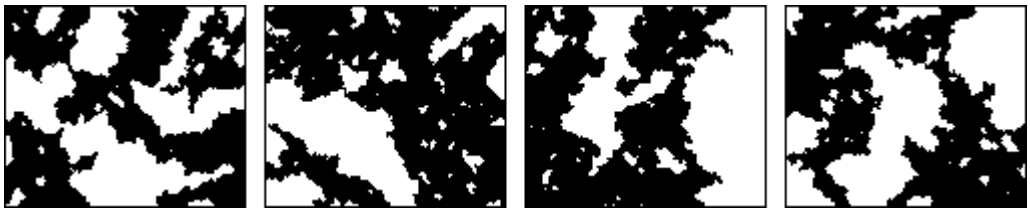
Świat w grze reprezentowany jest przez siatkę składającą się z równych rozmiarów kwadratów, w dalszej części pracy nazywanych również polami. Mapa świata znajdująca się w grze jest tablicą pól o wymiarach `120x100`. Aby jak najlepiej oddać warunki rzeczywiste, należy rozróżnić obszary lądowe od morskich, nadać im charakterystyczne cechy klimatyczne oraz florę.

3.1 Elewacja

Pierwszym etapem generowania świata gry jest stworzenie mapy wysokości terenu. Część wykorzystywanych algorytmów zwraca jednak wartości zero-jedynkowe, co pozwala wyłącznie na oddzielenie wód od lądów. W najprostszej wersji rezultatem takiego działania jest przypisanie dla każdego pola wartości `1` (ląd) lub `0` (woda). Czynność tę można wykonać na wiele sposobów, w pracy zostały przetestowane dwa z nich.

3.1.1 Automat komórkowy

Proces ten bazuje na koncepcji *damage spreading*[18] automatów komórkowych. Reguła użytego automatu polega na wybraniu zadanej ilości losowych kwadratów na siatce wypełnionej nieokreślonymi polami, a następnie przypisania im wartości `1`. Powstałe pola lądowe dodawane są na listę. Dla każdego elementu listy sprawdzane jest osiem pól sąsiadujących, jeżeli którekolwiek z nich nie ma ustalonej żadnej wartości, dokonuje się losowego przypisania zera lub jedynki z ustalonym wcześniej prawdopodobieństwem oraz dodanie tego pola na listę. Po sprawdzeniu wszystkich sąsiadów, obecny kwadrat usuwany jest z listy. Automat działa dopóki lista nie jest pusta.



Rysunek 1: Przykłady lądu wygenerowanego poprzez automat komórkowy. Kolor biały oznacza wodę.

[18] F.Bagnoli, R.Rechtman, S.Ruffo, Damage spreading and Lyapunov exponents in cellular automata, Phys.Lett.A, 172, 34, 1992

3.1.2 Szum Perlina

Algorytm generujący szum Perlina polega na połączeniu ze sobą interpolowanych tablic pseudolosowych szumów ze zmienną dokładnością. Rozwiązanie takie, w przeciwieństwie do automatu komórkowego, pozwala na nadanie polom wartości z wybranego przedziału, a nie tylko zero-jedynkowych. W praktyce funkcja generująca szum Perlina zwraca dwuwymiarowy obraz w odcieniach szarości. Dokonując prostej interpretacji można przyjąć, że jasność poszczególnych pól jest proporcjonalna do ich wirtualnej wysokości jako elementów świata. Jeżeli wybierzemy wartość odpowiadającą wysokości poziomu wody, łatwo można określić które pola znajdują się pod taflą, a które nad.

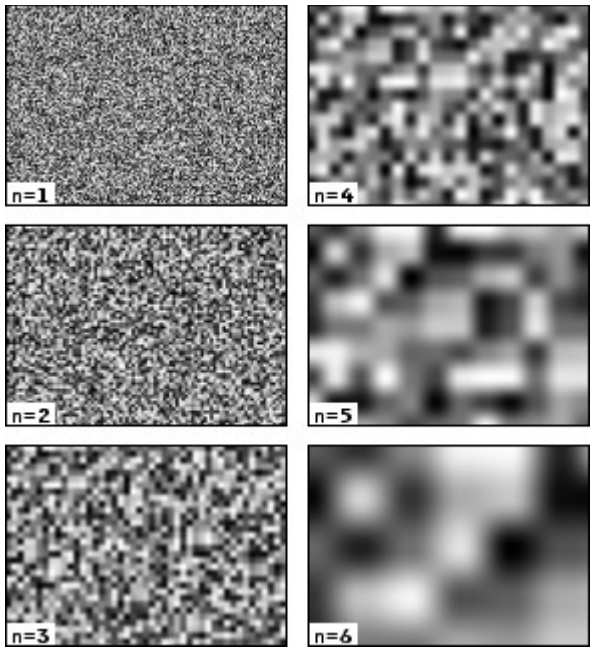
Algorytm działania szumu Perlina na przykładzie dwuwymiarowej bitmapy:

1. Korzystając z generatora liczb pseudolosowych, tworzymy podstawową mapę szumu o określonym rozmiarze (AxB). Powstały obraz składa się z pojedynczych, czarnych lub białych pikseli (Rysunek 2 dla n=1).

2. Dokonujemy przypisania n=2.

3. Tworzymy nową mapę korzystając z wycinka podstawowej o n^2 mniejszym rozmiarze (A/n x B/n).

4. Skalujemy nowo powstałą mapę do rozmiarów bitmapy początkowej, stosując interpolację funkcją cosinus. Uzyskany obraz posiada czarne i białe piksele w odstępach równych n interpolowanych skalą szarości.

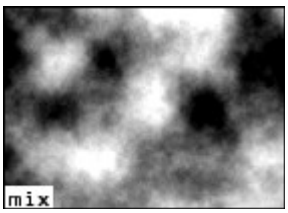


Rysunek 2: Bitmapy powstałe dla różnych iteracji szumu Perlina.

5. Powtarzamy punkty 3 i 4 za każdym razem inkrementując wartość n, aż do uzyskania pożądanej dokładności.

6. Łączymy uzyskane bitmapy poprzez sumowanie wartości pikseli pomnożonych przez wagę przypisaną do każdego obrazu. Dla najwyższego n, waga bitmapy wynosi w=0.5, natomiast dla pozostałych obrazów jest to w(n)=w(n+1)/2. W przedstawionym na obrazie przykładzie będą to wartości:

```
n=6  w=0.5
n=5  w=0.25
n=4  w=0.125
...
```

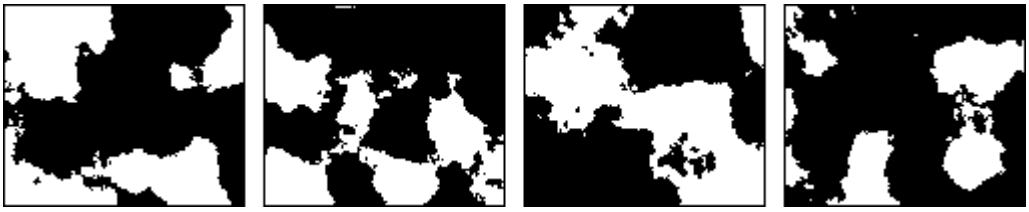


Rysunek 3: Bitmapa powstała ze scalenia map cząstkowych

Ze względu na specyfikę generowania i łączenia ze sobą obrazów, takie rozwiązanie nazywa się ułamkowymi ruchami Browna[19].

[19] J. Beran: Statistics for Long-Memory Processes, Chapman & Hall, 1994. ISBN 0-412-04901-5

Interpolacja w wygenerowanym szumie sprawia, że uzyskany obraz można przyrównać do rzeczywistości - czym dalej w głąb lądu, tym więcej obszarów górskich.



Rysunek 4: Przykłady lądu uzyskanego przez zdigitalizowany szum Perlina

Do generowania świata w projekcie użyty został szum Perlina. Głównym czynnikiem świadczącym o przewadze tego rozwiązania nad automatem komórkowym jest możliwość równoczesnego liczenia elewacji, co znacznie ułatwia późniejsze generowanie biomów. Czynnikiem kolejnym okazuje się zbliżona do rzeczywistej reprezentacja wybrzeża, która jest bardziej smukła i płynna niż otrzymana poprzez automat komórkowy.

3.2 Biomy

Kolejnym etapem generowania świata jest podzielenie go na biomy, czyli strefy różniące się klimatem, fauną oraz florą. Skala wielkości świata jest przyjęta tutaj bardzo umownie. Gdyby biomy występowały tak, jak w naturze, cała gra toczyłaby się w jednej strefie klimatycznej. Rozwiązanie takie jest niedopuszczalne, ze względu na występującą wtedy niską różnorodność odwiedzanych lokacji.

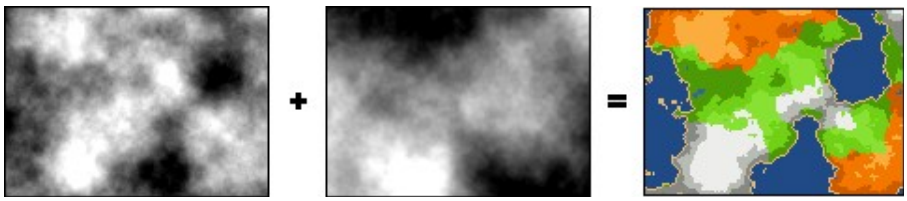
Biom dla każdego pola jest wyliczany jako funkcja dwóch parametrów: wysokości nad poziomem morza oraz temperatury. Dzięki temu, że jako generator terenu wybrany został szum Perlina, wysokość pola można łatwo uzyskać odczytując i digitalizując jego jasność. Temperatura wyliczana jest w identyczny do wysokości sposób. Korzystając z tej samej metody generowany jest obraz rozkładu ciepła.

Dla każdego pola będącego lądem przypisujemy wartość wysokości i temperatury, mieszczącą się w zakresie `1-3`. Taka ilość możliwości pozwala na zadeklarowanie dziewięciu różnych biomów. Poniższa tabela jest pewnym uproszczeniem używanego w klimatologii diagramu Whittakera[20].

▲ Wysokość ▲			
	Pustynia lodowa	Lasy twardolistne	Step
	Tundra	Lasy liściaste	Pustynia
	Tajga	Lasy równikowe	Sawanna
	► Temperatura ►		

Tabela 1: Przypisanie biomu w zależności od wysokości i temperatury.

[20] Robert H. Whittaker: Communities and Ecosystems, Macmillan, 1975. ISBN 0-02-427390-2



Rysunek 5: Przykład połączenia dwóch map z szumem Perlina. Obraz lewy przedstawia wysokość nad poziomem morza, środkowy temperaturę, a prawy wynikową mapę.

Oczywiście w naturze taki sposób jest nie do przyjęcia, ponieważ rzeźba terenu danego regionu jest zawsze ściśle zależna od obecnego biomu. Wysoce nieoptymalna byłaby jednak implementacja procesów formowania powierzchni, bazująca na klimacie, pogodzie i warunkach atmosferycznych. Z tego też względu zastosowano rozwiązanie pozwalające na niezależne generowanie ukształtowania terenu i stref klimatycznych, gdzie za powiązania tych dziedzin odpowiada podana wcześniej tabela.

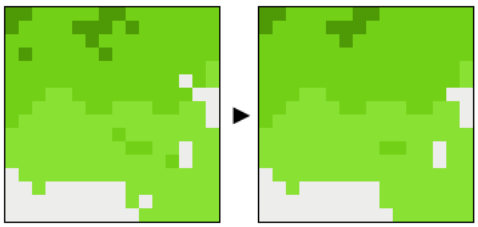
3.2.1 Korekta biomów

Mapa wygenerowana przez szum Perlina wymaga kilku poprawy jakościowych. Konieczna jest również analiza topografii uzyskanego terenu na potrzeby wiarygodnego odwzorowania świata.

- **Rozróżnienie oceanów i jezior**
  Ogólnie przyjętym założeniem jest, że każdy obszar wodny graniczący z krawędzią mapy (końcem obrazu) traktowany jest jako ocean. Pozostałe akweny wodne są otoczone przez ląd, więc przypisywana jest im wartość jeziora.

- **Wyszukiwanie wybrzeży**
  Każde pole będące lądem bezpośrednio graniczącym z oceanem zapisywane jest jako wybrzeże.

- **Usuwanie pojedynczych biomów**
  Po przypisaniu biomów, na mapie świata pojawiają się pojedyncze miejsca posiadające przypisany inny biom, niż sąsiadujące (w terminologii otoczenia von Neumana) z nimi pola. W rozumieniu szumu Perlina, są to ekstrema lokalne funkcji na ograniczonym obszarze (najwyżej `3x3` pola). W celu estetycznego wyrównania terenu, biomy w tych miejscach zastępowane są losowymi biomami z pól sąsiednich.



Rysunek 6: Redukcja pojedyńczych biomów.

Po wygenerowaniu stref klimatycznych i przypisaniu każdemu polu odpowiedniej wartości biomu, algorytm przechodzi do tworzenia terenu. Każdy kwadrat mapy ma przypisaną jedną wartość, charakteryzującą to, co znajduje się na danym gruncie.

3.3 Ukształtowanie terenu

3.3.1 Rzeki

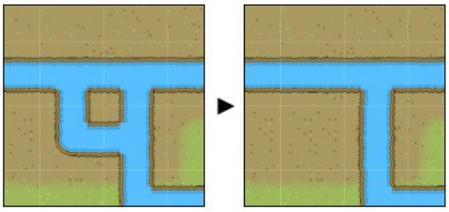
Rzeki zajmują około 0,1% całkowitej liczby pól na mapie świata. Każda rzeka ma swoje źródło na polu z biomem o najwyższej wysokości, czyli są to; step, las twardolistny lub pustynia lodowa. Dla każdej rzeki, losowo wyznaczany jest kierunek jej ujścia, następnie w myśl reguły automatu komórkowego, pole sąsiadujące ze źródłem jest oznaczane jako rzeka. To, które z pół sąsiednich zostanie wybrane jako nowe pole rzeki, wyznaczane jest na podstawie jej kierunku oraz losowego prawdopodobieństwa. Taki schemat zostaje powtarzany dopóki ostatnio aktualizowane pole jest oceanem, jeziorem, graniczy z krawędzią mapy lub inną, aktualnie istniejącą rzeką (powstaje wtedy dopływ).

Przykład:
Jeżeli kierunek rzeki zostanie wylosowany jako 30° w skali 360. stopniowej, gdzie 0° odpowiada kierunkowi wschodniemu, a 90° północnemu, to przy obecnym źródle w polu `(X,Y)` prawdopodobieństwa przypisania terenu rzeki dla pól sąsiednich przedstawia się następująco:

```
P(X+1, Y) = 2/3
P(X-1, Y) =  0
P(X, Y-1) = 1/3
P(X, Y+1) = 0
```

Należy pamiętać, że pole `(0,0)` mapy świata znajduje się w jej lewym-górnym rogu.
Jeżeli uzyskana wartość losowa wskazuje na pole `(X+1, Y)` to staje się ono polem aktualnym `(X,Y)`. Powyższe wartości prawdopodobieństwa pozostają niezmienne do końca ewolucji biegu rzeki.

Ponieważ przedstawiony algorytm może produkować zapętlenia (klastry pól `2x2` będących rzekami), stosuje się naiwne rozwiązanie redukujące takie pętle poprzez usuwanie rzek z pól posiadających najmniej takich samych sąsiadów.



Rysunek 7: Korekcja przepływu rzeki.

- **Rzeki w rozgrywce**
  Gracz nie może wejść na pole będące rzeką. Jeżeli powstanie tam droga, jest ona transformowana w most, już osiągalny dla bohatera.

- **Rysowanie rzek**
  W celu optymalnie szybkiego oraz estetycznego wyświetlania graficznej reprezentacji rzek, w programie zastosowano szablon do ich produkcji.

Mając do dyspozycji rysowanie okręgów oraz losowo zniekształconych brzegów rzeki, renderowanie każdego pola rozpoczyna się od narysowania koła w jego centrum, a następnie, w zależności od pól sąsiadujących, połączeń między nimi.

3.3.2 Góry

Prawdopodobieństwo przypisania terenu górzystego dla danego pola przedstawia wygenerowana, poprzez ponowne zastosowanie szumu Perlina, bitmapa w skali odcieni szarości. W tym przypadku, zważywszy na aspekt rozgrywkowy, szum posiada większą ziarnistość, dzięki czemu mapa jest zrównoważona pod względem występowania gór. Jasność piksela z uzyskanej mapy określa prawdopodobieństwo przypisania terenu górzystego do tego pola.

Rysunek 8: Reprezentacja graficzna gór.

- **Góry w rozgrywce**
  Pole zajęte przez skały jest nieosiągalne dla gracza, nie można również przez nie przeprowadzić drogi łączącej miasta.

- **Rysowanie gór**
  Wzgórze powstaje poprzez łączenie losowo wybranych punktów tworzących łamaną otwartą zwyczajną, która następnie jest zamykana poprzez połączenie z punktami podstawy. Uzyskana figura jest wypełniana kolorem zależnym od biomu na jakim się znajduje.

3.3.3 Lasy

Lasy na mapie świata prezentowane są jako pojedyncze drzewa. Prawdopodobieństwo umiejscowienia ich na danym polu zależy, analogicznie jak w przypadku gór, od wygenerowanej bitmapy oraz przypisanego biomu. Skutkuje to mapą, na której, dla przykładu, zalesienie w strefie lasów liściastych jest o wiele większe niż w strefie pustynnej.

- **Lasy w rozgrywce**
  Gracz może swobodnie przemieszczać się po polach oznaczonych jako lasy, jednakże prawdopodobieństwo zaatakowania bohatera przez losowych przeciwników jest wtedy znacząco wyższe.

- **Rysowanie lasów**
  Na każdy biom występujący w grze, przypada jeden rodzaj wyświetlanego drzewa. Są to:
  - akacja (sawanna),
  - kaktus (pustynia),
  - krzew stepowy (step),
  - bananowiec (lasy tropikalne),
  - dąb (lasy liściaste),
  - pistacja (lasy twardolistne),
  - świerk (tajga),
  - brzoza karłowata (tundra),
  - sosna karłowata (pustynia polarna).

Rysunek 9: Reprezentacje graficzne drzew.

W rysowaniu drzew wykorzystano wiele różnych technik, ponieważ każdy gatunek posiada zupełnie odmienną charakterystykę. Liście najczęściej są przedstawione jako

elipsy, natomiast iglaki generuje się przez nakładanie na siebie malejących trójkątów. Do wytworzenia niektórych specyficznych kształtów użyto krzywych Bézier'a[21].

3.4 Infrastruktura

Na powstałą, wypełnioną terenem, mapę nakłada się elementy infrastruktury. Należą do nich miasta, drogi, mosty oraz porty. Ilość miast losowana jest z przedziału <10,20>. Każdy element infrastruktury zajmuje powierzchnię jednego pola.

Pierwszym krokiem generowania infrastruktury jest losowe wybranie dwóch pól na których umieszczone zostaną miasta, oraz połączenie ich ze sobą. Miasto może powstać na każdym terenie z wyjątkiem oceanów, wybrzeży, rzek oraz gór. W celu wyznaczenia drogi pomiędzy dwoma początkowo utworzonymi miastami stosuje się heurystyczny algorytm A*[22]. Jest to powszechnie stosowany w grach komputerowych sposób wyznaczania trasy między dwoma punktami. A* jest najszybszym z algorytmów wyszukiwania najkrótszej drogi w grafie, dający przy tym najbliższe optymalnym wyniki.

Schemat działania algorytmu A*:

Wymagane struktury danych:
- zbiór pól przeszukiwania posiadających dodatkowo parametry:
  - F – ocena ścieżki będąca sumą parametrów G i H,
  - G – koszt ruchu z punktu startowego do danego pola,
  - H – przewidywany koszt ruchu z danego pola do pola końcowego,
  - współrzędne pola rodzica,
- lista otwarta,
- lista zamknięta.

1. Umieść pole początkowe na liście otwartej.
2. Znajdź na liście otwartej element posiadający najniższą wartość F. Od tej pory jest on aktywnym polem.
3. Dodaj aktywne pole do listy zamkniętej
   - 3a. *Pierwszy warunek stopu*: Pole docelowe zostało dodane do listy zamkniętej.
4. Dla każdego pola sąsiadującego z aktywnym:
   - 4a. Jeżeli jest nieosiągalne lub znajduje się na liście zamkniętej, zignoruj je.
   - 4b. Jeżeli nie znajduje się na liście otwartej:
     - dodaj je do niej,
     - uczyń wierzchołek aktywny rodzicem sprawdzanego pola,
     - oblicz dla niego wartości parametrów:
       - G przyjmuje wartość rodzica powiększoną o jeden,
       - H jest euklidesową odległością do punktu końcowego,
       - F jest sumą wartości G i H.
   - 4c. Jeżeli znajduje się na liście otwartej, sprawdź czy jego wartość G jest większa od zinkrementowanej wartości G pola aktywnego, jeśli tak:

[21] James D Foley, Andries van Dam, Steven K Freiner, John F Hughes, Richard L Phillips: Wprowadzenie do grafiki komputerowej. Jan Zabrodzki (tłumaczenie). Warszawa: Wydawnictwa Naukowo-Techniczne, 1995. ISBN 83-204-1840-2.

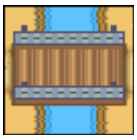
[22] P.E.Hart, N.J.Nilsson, B.Raphael: A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Transactions on Systems Science and Cybernetics SSC4 4 (2): 100–107, 1968

- uczyń wierzchołek aktywny rodzicem sprawdzanego pola,
- przelicz wartości parametrów `G` i `F`.

5. *Drugi warunek stopu*: Lista otwarta nie zawiera żadnych elementów. Nie znaleziono drogi.
6. Wróć do punktu `2`.

Jeżeli nie znaleziono drogi pomiędzy miastami, losowana jest następna para, a wyszukiwanie drogi powtarzane. Mając dwa połączone miasta początkowe, algorytm losuje kolejne i łączy je z istniejąca infrastrukturą.

- **Mosty**
  Jeśli okaże się, że droga prowadzi przez rzekę, na polu tym zostanie umieszczony most.



Rysunek 10: Reprezentacja graficzna mostu.

- **Porty**
  W przypadku, kiedy nowo wylosowane miasto nie łączy się z żadną drogą ani innym miastem, przyjmuje się, że znajduje się ono na innej wyspie. W takim wypadku powtarza się algorytm wyszukiwania drogi z możliwością poprowadzenia trasy przez ocean. Jeżeli takie połączenie znaleziono, to każde pole drogi położone na oceanie i graniczące z drogą lądową traktuje się jako port. W przeciwnym wypadku losowane jest nowe położenie miasta.

Rysunek 11: Reprezentacja graficzna portu.

- **Miasta w rozgrywce**
  Każde takie pole symbolizuje obszar który może zostać odwiedzony przez gracza. Wchodząc do miasta bohater przenosi się z mapy świata do wygenerowanej lokacji, gdzie może odwiedzać poszczególne domy i rozmawiać z mieszkańcami.

- **Rysowanie miast**
  Zależnie od temperatury biomu na obszarze którego znajduje się miasto, istnieją trzy typy miast:
  - na terenie pustynnym - domy zbudowane głównie z kamienia, częściowo lepianki i szałasy,
  - na terenie umiarkowanym - budynki z cegieł, osady ogrodzone murem,
  - na terenie chłodnym - chaty z drewna, ocieplane słomą.

  Do rysowania domów nie używa się żadnych specjalistycznych algorytmów generujących. Elementy architektoniczne tworzy się poprzez losowe dobieranie wymiarów budynków, umiejscowienia drzwi, czy bram.

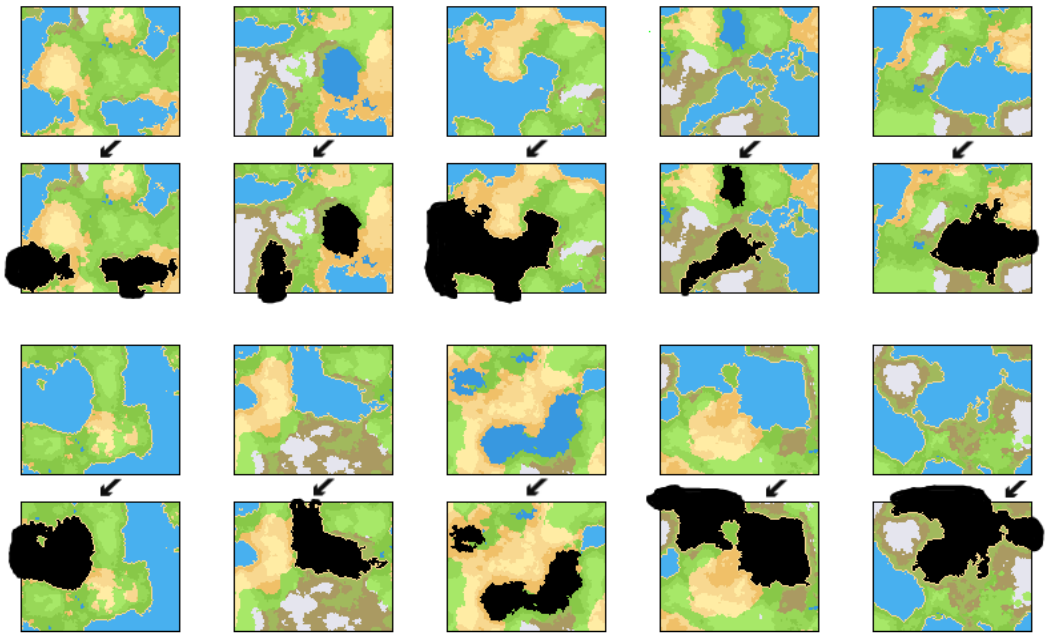
Rysunek 12: Reprezentacja graficzna miast.

3.5 Analiza kształtów

W celu sprawdzenia wiarygodności wyników algorytmu szumu Perlina oraz tego jak z geograficznego punktu widzenia ma się on do świata realnego, dokonano analizy porównawczej obu uniwersów.

Głównym czynnikiem do zestawienia w przypadku generowania mapy świata jest kształt lądów i wód. Najlepszym sposobem ich porównania jest analiza geometryczna z zakresu cyfrowego przetwarzania obrazów. Ponieważ obliczanie współczynników kształtu ma sens tylko dla jednolitych, zamkniętych obszarów, najlepiej w tym przypadku dokonać analizy jezior wygenerowanych przez aplikację oraz tych istniejących w rzeczywistości.

Na potrzeby testu wygenerowano dziesięć kolejnych losowych map, a następnie wyodrębniono z nich kształty zbiorników wodnych. Jeżeli nie były one zamknięte i kończyły się poza granicami obrazu, dokonywano ich ekstrapolacji. Uzupełnianie wykonywano ręcznie by przy jednoczesnym zachowaniu wiarygodnego kształtu dodać zbiornikom jak najmniej objętości.



Rysunek 13: Wyodrębnienie obszarów wodnych z przykładowych map.

Dla celów projektu wybrane zostało pięć wskaźników kształtów[23] obiektów:

- **Zawartość**
  Popularny wskaźnik niezależny od liniowych transformacji - skali i rotacji, opisywany wzorem:

$$R_Z = \frac{L^2}{4\pi S}$$

  Gdzie
  ***L* -** obwód,
  ***S* -** pole powierzchni.

- **Centryczność**
  Jest to stosunek długości maksymalnej cięciwy A, obiektu, do maksymalnej długości cięciwy B prostopadłej do A (cięciwy na rysunkach oznaczono kolorem zielonym).

$$R_C = \frac{A}{B}$$

  Gdzie
  ***A* -** długość cięciwy dłuższej,
  ***B* -** długość cięciwy krótszej.

- **Smukłość**
  Jest stosunkiem długości boków prostokąta granicznego opisanego na obiekcie (kolor czerwony na rysunkach) wyrażana wzorem:

$$R_S = \frac{a}{b}$$

  Gdzie
  ***a* -** długość dłuższego boku prostokąta grnaicznego,
  ***b* -** długość krótszego boku prostokąta granicznego.

- **Prostokątność**
  Jako stosunek pola powierzchni obiektu do pola powierzchni prostokąta granicznego opisanego na tym obiekcie.

$$R_P = \frac{S}{ab}$$

  Gdzie
  ***S* -** pole powierzchni,
  ***a* -** długość dłuższego boku prostokąta granicznego,
  ***b* -** długość krótszego boku prostokąta granicznego

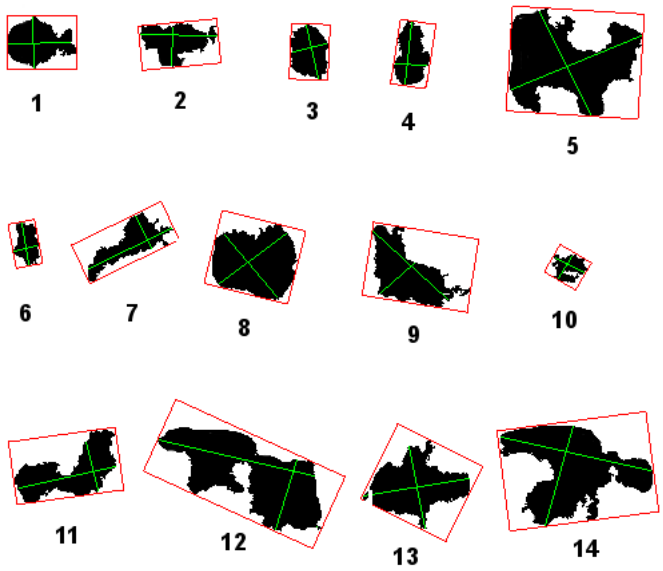
- **Współczynnik Malinowskiej**
  Jest jednym z najprostszych współczynników kształtu stosowanych w cyfrowej analizie obrazów i wyraża się wzorem:

$$R_M = \frac{L}{2\sqrt{\pi S}} - 1$$

  Gdzie
  ***L* -** obwód,
  ***S* -** pole powierzchni.

[23] Według wykładu M. Kujawińska: Cyfrowe przetwarzanie obrazów. Politechnika Warszawska, Wydział Mechatroniki, Instytut Mikromechaniki i Fotoniki

Poniższa tabela prezentuje otrzymane wyniki dla kształtów wygenerowanych przez szum Perlina.



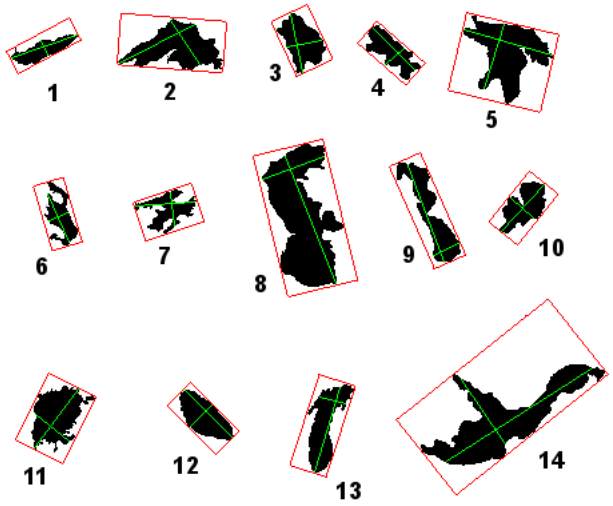
Rysunek 14: Wygenerowane obszary wodne poddane analizie.

Numer figury	Zawartość	Centryczność	Smukłość	Prostokątność	Współczynnik Malinowskiej
1	3,28	1,26	1,29	0,57	0,81
2	4,26	1,97	1,74	0,55	1,06
3	2,12	1,50	1,41	0,75	0,46
4	3,23	1,92	1,89	0,62	0,80
5	5,15	1,22	1,27	0,60	1,27
6	2,21	1,78	1,57	0,63	0,49
7	6,15	2,46	2,34	0,45	1,48
8	2,41	1,02	1,12	0,71	0,55
9	4,92	2,20	1,43	0,46	1,22
10	5,33	1,18	1,08	0,52	1,31
11	4,84	2,05	1,71	0,49	1,20
12	5,35	2,14	2,33	0,56	1,31
13	4,43	1,29	1,13	0,54	1,11
14	6,73	1,49	1,47	0,50	1,59

Tabela 2: Współczynniki kształtu dla wygenerowanych obszarów.

W celu porównania kształtów z figurami realnie występującymi w przyrodzie, do zestawienia wybrano największe jeziora na Ziemi. Obrazy satelitarne zostały przeskalowane, aby względnie pasować do wymiarów kształtów z poprzedniego testu. Następnie wykonano dla nich identyczne pomiary.

Poniższa tabela prezentuje otrzymane wyniki dla kształtów największych jezior na kuli ziemskiej.



Rysunek 15: Realne zbiorniki wodne poddane analizie.

Numer figury	Zawartość	Centryczność	Smukłość	Prostokątność	Współczynnik Malinowskiej
1	3,80	4,08	2,84	0,48	0,95
2	6,59	1,56	2,00	0,38	1,57
3	2,56	1,81	1,57	0,63	0,60
4	3,86	2,32	2,09	0,63	0,97
5	5,19	1,37	1,18	0,45	1,28
6	7,13	2,41	2,00	0,41	1,67
7	9,45	1,84	1,53	0,46	2,07
8	5,03	2,04	2,02	0,60	1,24
9	5,55	3,50	3,23	0,51	1,36
10	3,33	2,04	1,75	0,56	0,82
11	4,92	1,75	1,43	0,56	1,22
12	2,20	2,22	2,00	0,65	0,48
13	4,46	3,14	2,52	0,51	1,11
14	8,04	2,17	1,99	0,31	1,84

Tabela 3: Współczynniki kształtu dla realnych zbiorników wodnych.

Wyniki obu testów oscylują w podobnych granicach. Aby dokonać bezpośredniego porównania otrzymanych wartości obliczona została średnia arytmetyczna oraz odchylenie standardowe z wyników każdego współczynnika kształtu.

Cecha	Średnie wartości elementów wygenerowanych	Średnie wartości elementów rzeczywistych
Zawartość	4,32 ± 1,46	5,15 ± 2,07
Centryczność	1,68 ± 0,46	2,30 ± 0,77
Smukłość	1,56 ± 0,41	2,01 ± 0,55
Prostokątność	0,57 ± 0,09	0,51 ± 0,10
Wsp. Malinowskiej	1,05 ± 0,37	1,23 ± 0,45

Tabela 4: Porównanie średnich wyników kolejnych współczynników kształtu dla obu testów.

Największą różnicą odznacza się wskaźnik centryczności z czego wywnioskować można, że elementy generowane komputerowo nie posiadają tak dużego zróżnicowania kształtów jakie występuje w naturze.

Zbliżone wyniki prostokątności oraz smukłości świadczą o dobrym doborze parametrów szumu Perlina oraz ogólnym okołoeliptycznym charakterze jezior w naturze.

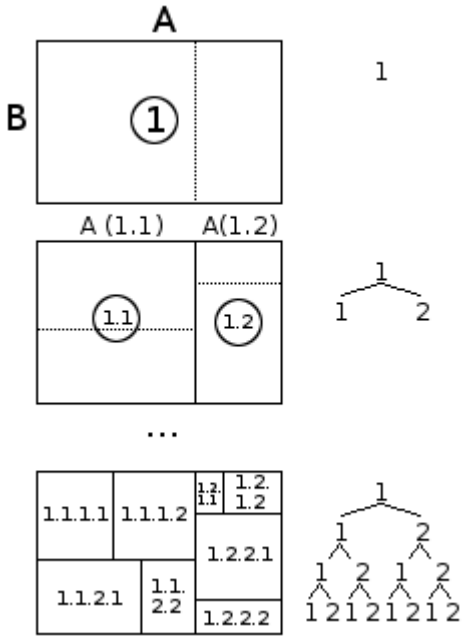
Szczególną uwagę powinny zwrócić podobne średnie współczynnika Malinowskiej, które wskazują stopień skomplikowania obwodu. Rzeczywiste zbiorniki wodne posiadają nieregularne rodzaje nabrzeża, lecz, jak widać, algorytm generowania szumu jest w stanie wytworzyć kształty o zbliżonym poziomie zróżnicowania obwodu.

4. Miasto

Ze względu na przyjętą konstrukcję map gry (które składają się z kwadratowych pól, a w przypadku miast są ich tablicą o wymiarach `60x60`), reprezentacja poszczególnych budynków w mieście opiera się na planach prostokątów. Najbardziej adekwatnym dla takiego modelu schematem rozmieszczenia ulic jest architektura nowojorska, tj. miasta w którym plany poszczególnych dzielnic składają się z siatki różnej wielkości prostokątów. Przykładem rodzimej metropolii o takiej budowie jest Łódź. Plac Wolności pełnił rolę punktu centralnego, a otaczające go prostopadłe ulice Zachodnia, Północna, Wschodnia i Południowa (dziś ul. Rewolucji 1905 oraz ul. Próchnika) wyznaczały podstawowy szablon dla budowy następnych dróg.

4.1 Plan dróg

Pierwszym etapem generowania miasta jest stworzenie siatki dróg rozdzielającej poszczególne parcele. Do takiego zadania wygodne jest użycie algorytmu BSP[24]. Danymi wejściowymi algorytmu są wymiary prostokąta do podziału (w tym wypadku będzie to całkowita mapa miasta) oraz maksymalna ilość możliwych iteracji. Wynikiem działania algorytmu korzystającego z drzewa binarnego jest zbiór prostokątów zawierających się w prostokącie wejściowym (które są reprezentantami uzyskanych parceli).



Rysunek 16: Graficzna reprezentacja kolejnych kroków rozwoju drzewa binarnego.

Algorytm działania binarnego partycjonowania przestrzeni:

`1`. Do drzewa BSP dodaj prostokąt wejściowy - stanie się on korzeniem struktury i obecnie aktywnym węzłem. Ustaw zmienną opisującą ilość dokonanych iteracji `i=0`.

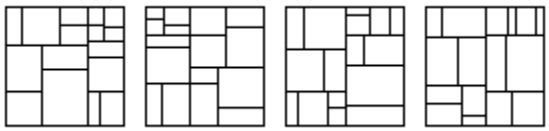
`2`. Jeżeli `i` jest równe pożądanej ilości iteracji, przejdź do punktu `4`. W przeciwnym wypadku dla każdego bezpotomnego węzła drzewa wykonaj punkt `3`. i zwiększ wartość zmiennej `i` o jeden, a następnie powtórz obecny krok.

[24] B. Naylor: Constructing Good Partitioning Trees, Graphics Interface (annual Canadian CG conference) May, 1993

3a. Ustal kierunek podziału prostokąta na podstawie jego wymiarów. Jeżeli bok A jest dłuższy od boku B, czworokąt dzielony będzie linią pionową, w przeciwnym razie linią poziomą.
3b. Wylosuj miejsce przecięcia prostokąta w aktywnym węźle, zachowując odpowiedni margines.

3c. Jeżeli określony margines nie pozwala na przecięcie czworokąta, nie rób nic, w przeciwnym razie dodaj dwa powstałe prostokąty do drzewa BSP jako węzły potomne do aktywnego.

4. Utwórz tablicę prostokątów składającą się z wymiarów przechowywanych tylko w liściach drzewa binarnego. Każde pole mapy znajdujące się na obwodzie dowolnego z uzyskanych prostokątów potraktuj jako drogę.



Rysunek 17: Przykłady wygenerowanych siatek dróg w mieście przy wykorzystaniu sześciu iteracji algorytmu.

W powyższy sposób, poza globalną siecią dróg, algorytm zwraca współrzędne każdej działki na której następnie można stworzyć określony rodzaj zabudowy.

Wielkością uzyskanych działek można sterować poprzez zmianę ilości iteracji algorytmu BSP. Dla miast generowanyh w grze, algorytm wykonuje cztery iteracje, co daje wiarygodne rozmiary parceli na mapie o rozmiarze 60x60 pól.

4.2 Przygotowanie listy budynków

Każde miasto posiada zestaw budynków które obowiązkowo muszą się w nim znajdować. Bazując na konwencji przyjmowanej w grach typu cRPG, są to:

- **Tawerna**
  Tawerna jest miejscem w którym bohater może nabyć od oberżysty różne rodzaje trunków. Jest to również miejsce gromadzące przyjezdnych z innych miast.

- **Kuźnia**
  W kuźni bohater może przede wszystkim wyposażyć się w oręż służący do walki. W każdym obiekcie tego typu znajduje się co najmniej jeden kowal, który udostępnia towary na sprzedaż.

- **Zbrojownia**
  Zbrojownia jest bliskim odpowiednikiem kuźni. Jedyną różnicą jest możliwość nabycia zbroi i elementów ochronnych zamiast oręża do walki.

- **Zielarnia**
  Chata zielarki to miejsce w którym bohater wyposaży się w mikstury leczące oraz poprawiające statystyki.

- **Świątynia**
  W każdej świątyni gracz może otrzymać od kapłana błogosławieństwa, które będą zwiększać statystyki bohatera przez określony czas.

- **Rynek**
  Rynek jest głównym miejscem spotkań mieszkańców miasta. Można na nim kupić różne przedmioty niedostępne w konwencjonalnych sklepach. Plac umożliwia również dostęp do studni, którymi schodzi się do miejskich katakumb.

Budynkiem opcjonalnym, zależnym od uzyskanej siatki dróg, jest **pałac królewski** oraz **cmentarz**. Dodatkowymi zabudowami mogącymi pojawić się w mieście są prywatne domy mieszkalne.

4.3 Gospodarowanie przestrzenią

Kolejnym etapem generowania miasta jest dopasowywanie zabudowy z listy budynków, do działek uzyskanych w etapie tworzenia siatki dróg. Każdy budynek posiada wymagane minimalne rozmiary parceli. Przykładowo; pałac królewski można wybudować na działce o rozmiarze minimum `27x17` pól, natomiast kuźnię już na działce `9x9` pól.

Rozważane zagadnienie można zakwalifikować jako dyskretny problem plecakowy[25]. Budynki jednak nie posiadają swojej wagi ważności i obowiązkowo muszą znajdować się na mapie, co dyskwalifikuje popularne realizacje algorytmów rozwiązujących ten problem.

Zabudowa jest umiejscawiana na losowych działkach o odpowiednim rozmiarze. Jeżeli na mapie istnieje przynajmniej jedna parcela o wymiarach minimum `27x17` pól (lub analogicznie `17x27`) to umieszczany jest w niej pałac królewski. Następnie według kolejności listy budynków, są one tworzone na kolejnych losowych działkach. Jeśli pozostały jakieś niezagospodarowane parcele to stawiane są tam zwykłe budynki mieszkalne. Działki nie spełniające minimalnych wymagań rozmiarowych są wypełniane losowymi drzewami.

4.4 Generowanie budynków

Każda zabudowa posiada swój algorytm generowania. Niektóre plany są wspólne dla budynków różnego rodzaju, natomiast różnią się obiektami umieszczanymi wewnątrz.

Elementem wykorzystywanym w większości planów generowania zabudowy jest algorytm dzielący ją na mniejsze części. Ponownie używane jest tutaj drzewo binarne, dla którego prostokątem wejściowym jest dany budynek, a zbiór wyjściowy to jego pokoje.

[25] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest: Wprowadzenie do algorytmów. Warszawa: Wydawnictwa Naukowo-Techniczne, 2003. ISBN 83-204-2800-9.

- **Pałac królewski**
  Minimalny rozmiar działki: `27x17` pól

  Pałac budowany jest w kształcie obróconej litery C. Głównym pomieszczeniem jest komnata królewska w której znajduje się między innymi tron. Hol łączy ze sobą dwa skrzydła pałacu podzielone losowo na pokoje.

Rysunek 18: Pałac królewski.

- **Świątynia**
  Minimalny rozmiar działki: `12x12` pól

  Każda świątynia oparta jest na tym samym schemacie; droga do niej prowadząca połączona jest z przedsionkiem, a ten z nawą główną. Wewnątrz znajduje się ołtarz, a ściany będące za nim układają się w kształt trójkąta. W zależności od rozmiaru działki ustalana jest ilość ławek wypełniająca główną komnatę.

Rysunek 19: Świątynia.

- **Tawerna**
  Minimalny rozmiar działki: `13x15` pól

  Z drogą połączony jest węższy hol recepcyjny będący przedsionkiem do pokojów gościnnych oraz kuchni, powstałych przez podzielenie jednego szerokiego budynku.

Rysunek 20: Tawerna.

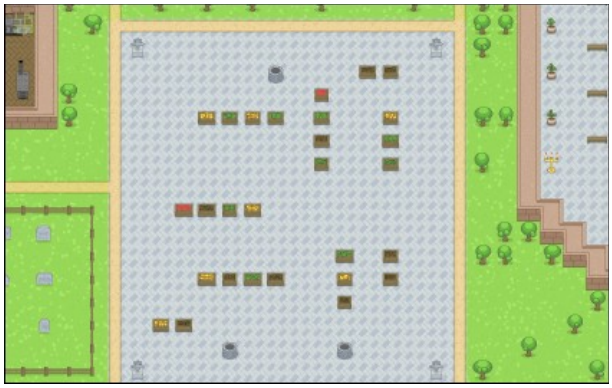
- **Sklep**
  Minimalny rozmiar działki: `9x9` pól

  Z przedstawionego algorytmu korzysta zarówno kuźnia, zbrojownia jak i zielarnia. Tworzony jest prostokątny budynek o wymiarach działki, w dalszej kolejności zostaje on podzielony algorytmem drzewa binarnego na mniejsze pokoje. Zaraz obok wejścia do budynków tego typu, dodawana jest tabliczka z symbolem prowadzonej działalności.

Rysunek 21: Sklep.

- **Rynek**
  Minimalny rozmiar działki: `5x5` pól

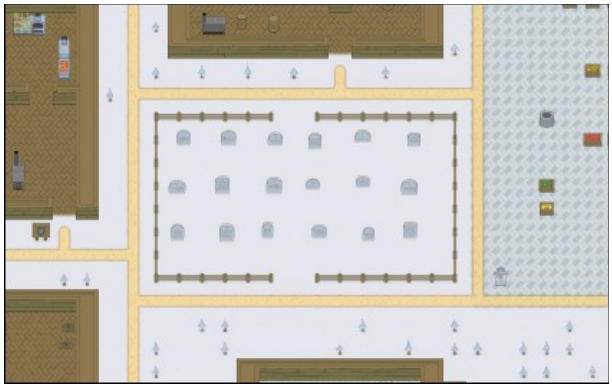
  Plac główny nie jest w rozumieniu *stricte* budynkiem, lecz zbiorem poustawianych losowo kramów kupieckich i studni publicznego użytku. Na każdym rogu działki rynkowej stoi posąg tarczy i miecza zwiększający estetykę okolicy.



Rysunek 22: Rynek.

- **Cmentarz**
  Minimalny rozmiar działki: `5x5` pól

  Miejsce pochówku mieszczan składa się z nagrobków ustawionych w odległościach dwóch pól na planie siatki. Cmentarz dodatkowo otoczony jest płotem.



Rysunek 23: Cmentarz

- **Dom mieszkalny**
  Minimalny rozmiar działki: `9x9` pól
  Zastosowano tutaj algorytm generowania sklepu, jednak bez umieszczania tablicy przy wejściu.

4.5 Aranżacje pokojów

Każdy pokój wygenerowany w budynku ma swój typ. Od typu pokoju zależą znajdujące się w nim obiekty, a te z kolei podzielone są na trzy rodzaje. Możliwe rodzaje aranżacji pokojów pokazuje tabela.

Rodzaj pokoju	Obiekt zajmujący dwa pola	Obiekt znajdujący się przy ścianie	Obiekt stojący na środku
Sypialnia	Łóżko	Garderoba	Stół
Kuchnia	Kuchenka	Beczka	Stół
Salon	Regał	Garderoba	Stół
Magazyn	Skrzynia	Skrzynka	Beczka
Kuźnia	Piec	Kowadło	Wiadro
Zielarnia	Suszarka ziół	Kocioł	brak

Tabela 5: Rodzaje obiektów znajdujących się w określonych pokojach.

Obiekt zajmujący dwa pola znajduje się zawsze w rogu pokoju.

W każdym pomieszczeniu obowiązkowo występuje jeden obiekt podwójny. W zależności od wymiarów danego pokoju może być generowana ich większa ilość.

Ilość obiektów stojących przy ścianie jest wprost proporcjonalna do powierzchni pomieszczenia. Przedmioty umieszczane są tylko w takich miejscach, które gwarantują przepustowość, tj. nie zastawiają przejścia do innego pokoju.

Jeżeli na środku pomieszczenia pozostało wystarczająco dużo miejsca, można tam umieścić dodatkowy obiekt. Jeśli będzie to stół, dookoła niego pojawią się losowo rozstawione krzesła.

W zależności od generatora budynków poszczególne pokoje mają przypisywane swoje typy. Jeżeli generowany jest dom mieszkalny to pierwszeństwo typu pokoju ma sypialnia, następnie kuchnia i salon. Odpowiednio podczas generowania zbrojowni, pierwszeństwo należy do pokojów typu kuźni, następnie magazynu czy kuchni.

5. Labirynt

Mimo iż struktury labiryntowe w naturze objawiają się głównie w korytarzach wytworzonych przez mrówki czy korniki, są one powszechnie wykorzystywane w zagadnieniu generacji map dla gier. Wszelkiego rodzaju podziemia, katakumby, a głównie lochy w produkcjach typu *roguelike* (nakierowanych na eksplorację ciągów pokoi i korytarzy) korzystają z różnorakich algorytmów generowania labiryntów.

5.1 Zastosowanie algorytmu

W miastach utworzonych w grze znajdują się place rynkowe. Na tych osiągających większe rozmiary, porozmieszczane są studnie którymi gracz może przenieść się do podmiejskich katakumb. Aby wygenerować znajdujący się tam labirynt, użyty został algorytm typu *Hunt&Kill*[26], który gwarantuje optymalny stosunek czasu generowania do jakości otrzymanego wyniku.

Uzyskany labirynt będzie labiryntem doskonałym. Oznacza to, że istnieje jedna i tylko jedna droga łącząca dwa dowolne jego punkty. Co za tym idzie, strukturę taką można wyrazić dowolnym grafem nie posiadającym cykli, czyli drzewem rozpinającym.

5.2 Hunt&Kill

Jak sama nazwa wskazuje, algorytm *Hunt&Kill* pracuje w dwóch fazach. Faza polowania *"hunt"* określa pole od którego rozpocznie się następna faza - *"kill"*. Aby opisać działanie tej metody należy wpierw zapoznać się z algorytmem błądzenia losowego.

> ***Algorytm błądzenia losowego***
>
> Poniższy przepis jest częścią algorytmu właściwego *Hunt&Kill* i odpowiada on fazie *"kill"*. Parametrem startowym procedury jest pole aktywne należące do mapy labiryntu.
>
> `1.` Jeżeli pole aktywne nie posiada nieodwiedzonych sąsiadów, kończymy działanie algorytmu.

[26] Jamis Buck: "Algorithms" is Not a Four-Letter Word. New Orleans, LA: RubyConf 2011

2. Spośród pół sąsiadujących z polem aktywnym, wybieramy losowe, nieodwiedzone pole.

3. Wybrane pole czynimy odwiedzonym i aktywnym, dodajemy krawędź łączącą obecne pole z poprzednim.

4. Wracamy do punktu 1.

Algorytm Hunt&Kill

1. Wybieramy losowe pole należące do docelowej mapy labiryntu i czynimy je aktywnym.

2. Dla aktywnego pola uruchamiamy algorytm błądzenia losowego.

3. Dokonujemy skanowania całej mapy labiryntu zaczynając od lewego górnego rogu, przeszukując kolejne wiersze w celu odnalezienia dowolnego nieodwiedzonego pola sąsiadującego z dowolnym polem odwiedzonym.

4. Jeżeli taka para pól została znaleziona, tworzymy krawędź grafu łączącą je. Pole nieodwiedzone czynimy odwiedzonym i aktywnym oraz przechodzimy do punktu 2.

5. W przeciwnym wypadku kończymy działanie algorytmu.

Wykorzystanie algorytmu Hunt&Kill dla generowania labiryntów	
Zalety	**Wady**
Względnie (w stosunku do innych algorytmów) krótki czas obliczeń. Niska złożoność obliczeniowa i pamięciowa. Satysfakcjonujące wyniki działania dla potrzeb gier komputerowych.	Algorytm ma tendencję do tworzenia długich, pojedynczych korytarzy. W skrajnych przypadkach technika błądzenia losowego może być wysoce nieoptymalna.

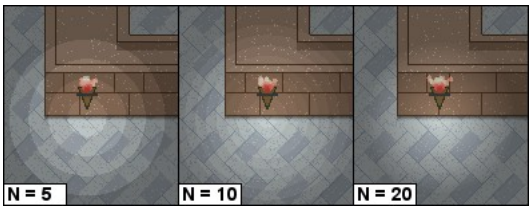
Tabela 6: Wady i zalety algorytmu Hunt&Kill.

5.3 Mapa oświetlenia

Oświetlenie będące na mapie świata lub miasta jest jednolite. W dużym uproszczeniu, światło słoneczne pada pionowo na ziemię zapewniając równomierne oświetlenie obiektów. Nie można powiedzieć tego samego o katakumbach znajdujących się pod miastem, gdzie mrok rozpraszają pochodnie wiszące na ścianach. W celu prostej symulacji takiego oświetlenia wykonana została mapa świetlna.

Mapa oświetlenia jest maską nakładaną na reprezentację graficzną aktualnej mapy. Wartość maski określana jest dla każdego piksela liczbą rzeczywistą należąca do przedziału <0,1>, w którym 0 odpowiada oryginalnemu kolorowi piksela mapy,

a `1` pełnej czerni. W przypadku podziemi miejskich maska początkowo posiada całkowite zaciemnienie w wysokości `0.5`. Następnie, w miejscach w których znajdują się pochodnie określane są kołowe obszary z mniejszymi wartościami zaciemnienia. Dla każdej pochodni tworzone jest `N` kół o różnym stopniu jasności i promieniu. Jasność danego koła jest odwrotnie proporcjonalna do kwadratu jego promienia, co ma pokrycie z rzeczywistym modelem oświetlenia punktowego. Po uzyskaniu kół o różnych parametrach następuje nałożenie ich na obraz poczynając od tego o największym promieniu. Pozwala to na uzyskanie efektu gradientu bardzo niskim kosztem obliczeniowym.



Rysunek 24: Symulacja gradientu światła pochodni dla różnych ilości rysowanych kół.

Dodatkowym efektem czysto kosmetycznym jest animacja blasku pochodni. W rzeczywistości płomień pulsuje, dając różne stopnie natężenia światła w przeciągu czasu. Wartość natężenia światła pochodni zmieniana jest wraz z upływem czasu zgodnie z regułą jednowymiarowych ruchów Browna.

6. Postacie

6.1 Statystyki postaci

We wszystkich wygenerowanych miastach znajdują się mieszkańcy. Każdy obywatel ma swój indywidualny zestaw cech opisujących zarówno jego wygląd zewnętrzny jak i nastrój. Wraz z tworzeniem budynków, na mapie umieszczane są punkty startowe dla poszczególnych postaci. Dla przykładu: w każdej świątyni obok ołtarza swój punkt startowy ma kapłan.

W momencie generowania miast przechowywane są tylko rodzaje wykonywanych zawodów, przypisanie cech indywidualnych następuje w następnej fazie jaką jest tworzenie postaci.

6.1.1 Zawody

Zawód	Punkty startowe
Żołnierz	pałac, rynek
Kowal broni	kuźnia, rynek
Kowal zbroi	zbrojownia, rynek
Oberżysta	tawerna, rynek
Zielarka	zielarnia, rynek
Król	pałac
Kapłan	świątynia
Kupiec	rynek
Kucharz	kuchnia, rynek
Tragarz	magazyn, rynek
Bezrobotny	dowolne

Tabela 7: Punkty startowe postaci w zależności od zawodu.

W zależności od zawodu, do postaci dobierane są konkretne charakterystyki. Przegląd wszystkich możliwych cech znajduje się w następnej tabeli.

6.1.2 Cechy indywidualne

Cecha	Dostępne opcje	
Płeć	mężczyzna kobieta	
Imię	(patrz: *rozdział 7.2 Imiona postaci*)	
Rasa	biała czarna nordycka	
Wiek	dziecko młody dorosły sędziwy	
Wysokość	liczba całkowita z przedziału 100-200	
Waga	liczba całkowita z przedziału 40-120	
Zawód	(patrz: *rozdział 6.1.1 Zawody*)	
Nastrój	spokojny wesoły smutny zły zaskoczony rozbawiony	
Fryzura	mężczyzna	łysy grzybek krótkie irokez
	kobieta	krótkie kitki długie kok
Kolor włosów	blond brązowy czarny rudy siwy	

Tabela 8: Możliwe wartości cech indywidualnych postaci.

6.1.3 Modyfikacje cech

W celu zwiększenia realizmu przy tworzeniu postaci w grze, należy każdej z nich przypisać odpowiednie reguły generowania. Złożenie reguł powstałych z różnych cech zewnętrznych, zapewnia możliwie wierne odwzorowanie realnie istniejących powiązań (np. występowanie czarnoskórych osób na terenach z najcieplejszym klimatem).

W utworzonym projekcie edycje cech podzielone są na trzy grupy.
Modyfikacje ze względu na:

- biom na którym znajduje się miasto,
- zawód wykonywany przez postać,
- wiek postaci.

Poniższe tabele prezentują kolejno transformacje konkretnych cech w zależności od innych cech początkowych.

Biom	Rasa	Kolor włosów
Sawanna Pustynia Step	czarna	czarny brązowy siwy
Lasy równikowe Lasy liściaste Lasy twardolistne	biała	bez modyfikacji
Tajga Tundra Pustynia lodowa	nordycka	blond rudy siwy

Tabela 9: Modyfikacje cech postaci ze względu na biom macierzysty.

Zawód	Minimalny wzrost	Maksymalny wzrost	Minimalna waga	Maksymalna waga	Płeć
Żołnierz	180		100		M
Kowal			80		M
Oberżysta			100		M
Zielarka	140		70	90	K
Król			80		M
Kapłan		150	80		M
Kucharz			100		
Tragarz					M

Tabela 10: Modyfikacje cech postaci ze względu na zawód.

Wiek	Zmiana cechy	Nowe wartości
Dziecko	maksymalny wzrost	120
	maksymalna waga	50
	fryzura męska	grzybek krótkie
	kolor włosów	blond brązowy czarny rudy

Sędziwy	maksymalny wzrost	140
	minimalna waga	70
	maksymalna waga	90
	kolor włosów	siwy
	fryzura męska	łysy grzybek krótkie
	fryzura damska	długie kok

Tabela 11: Modyfikacje cech postaci ze względu na wiek.

Dodatkowa reguła zakłada, że każde dziecko jest bezrobotne.

Powyższe zbiory reguł łączą się poprzez sumę, natomiast każda cecha powtarzająca się w różnych zbiorach jest dodawana poprzez iloczyn mnogościowy.

Przykład:
Jeżeli przyjmiemy, że do zbioru `A` należą reguły powstałe przez obecny biom (np. sawannę);

```
A = {
      rasa:              [czarna]
      kolor_włosów:      [brązowy, czarny, siwy]
    }
```

natomiast do zbioru B przyjmiemy cechy wieku (np. dziecka);

```
B = {
      maksymalny_wzrost: [120]
      maksymalna_waga:   [50]
      fryzura_męska:     [grzybek, krótkie]
      kolor_włosów:      [blond, brązowy, czarny, rudy]
    }
```

to wynikowym zbiorem modyfikacji reguł, które należy zaaplikować do generatora będzie:

```
A + B = {
      rasa:              [czarna]
      maksymalny_wzrost: [120]
      maksymalna_waga:   [50]
      fryzura_męska:     [grzybek, krótkie]
      kolor_włosów:      [brązowy, czarny]
    }
```

6.2 Animacja szkieletowa

Aby wprowadzić postacie w ruch, niezbędne było zastosowanie systemu animacyjnego. Najbardziej odpowiednią techniką, która jednocześnie współpracuje z proceduralnością jest model szkieletowy postaci.

Animacja szkieletowa składa się z zestawu dwuwymiarowych grafik reprezentujących kończyny oraz połączonych z odpowiednią strukturą kości. Poruszenie kością powoduje transformację skojarzonej z nią bitmapy.

Animacja szkieletowa	
Zalety	**Wady**
Bardzo niskie zużycie pamięci komputera. Każda postać posiada swój model szkieletu, a jej akcje zapisane są jako kolejne transformacje poszczególnych kości.	Wzmożony czas obliczeniowy. W każdej klatce animacji wyznaczana musi być transformacji kości, która następnie jest przekładana na transformację połączonej bitmapy.
Dowolność interakcji z otoczeniem. Szkielet może reagować na bodźce zewnętrzne i w ten sposób zmieniać swoją strukturę.	Widoczność łączeń między kośćmi. Postacie animowane szkieletowo mogą wyglądać jak poskładane z wielu drobnych obiektów.
Możliwość użycia jednej zdefiniowanej animacji dla wielu różnych obiektów o tej samej strukturze.	

Tabela 12: Wady i zalety animacji szkieletowej.

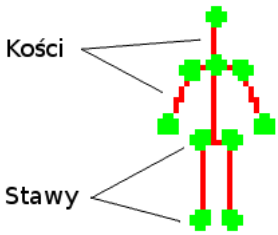
6.2.1 Budowa modelu szkieletowego

Na model animacji szkieletowej składają się:

1. Skóra, czyli zestaw dwuwymiarowych bitmap odpowiadających reprezentacji graficznej poszczególnych kończyn postaci.
2. Szkielet zbudowany z hierarchii
   - kości,
   - stawów.

Następujące reguły określają podstawowe założenia tego typu animacji.

- Skóra połączona jest ze szkieletem.
- Każda bitmapa przypisana jest do jednej z kości.
- Każda kość połączona jest z dwoma stawami.
- Każdy staw jest połączony z przynajmniej jedną kością.
- Hierarchię kości można opisać w strukturze drzewa.



Rysunek 25: Przykład podstawowego modelu szkieletowego człowieka.

Dla każdej postaci wygenerowano indywidualny szkielet na podstawie wzorca przedstawionego powyżej. Różnice między szkieletami dotyczą długości kości, uzależnionych od wygenerowanej wysokości postaci, oraz przechowywanych wartości szerokości dla każdej kończyny.

Grafiki połączone ze szkieletem generowane są na podstawie cech postaci. Oto kilka przykładów uzyskanych po etapie skórowania.

Rysunek 26: Zestaw losowo wygenerowanych postaci.

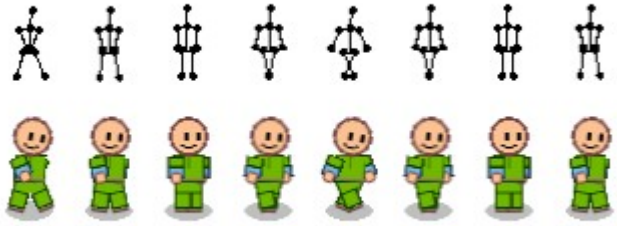
6.2.2 Animacja

Do uzyskania płynnej animacji niezbędne jest zdefiniowanie poszczególnych klatek kluczowych szkieletu, a następnie implementacja algorytmu interpolującego te stany.

Metodą zastosowaną w łączeniu ruchów między kośćmi jest kinematyka prosta[27]. Zakłada ona, że pozycja poszczególnych części modelu w danym czasie jest liczona według pozycji i orientacji samego modelu, uwzględniając wszelkie stawy znajdujące się po drodze. Przykładowo, jeżeli obrócone zostanie ramię postaci, to wraz z nim poruszy się tak samo przedramię, nadgarstek jak i palce, tak by zachować swoją względną orientację w stosunku do ramienia.

Przeciwieństwem kinematyki prostej jest kinematyka odwrotna, gdzie pozycja modelu jest obliczana na podstawie transformacji poszczególnych jego części. W przypadku kinematyki odwrotnej to poruszenie dłonią spowoduje ruch nadgarstka oraz przedramienia.

[27] J. M. McCarthy: Introduction to Theoretical Kinematics. MIT Press, Cambridge, MA, 1990



Rysunek 27: Cykl animacyjny dla poruszania się w prawo.

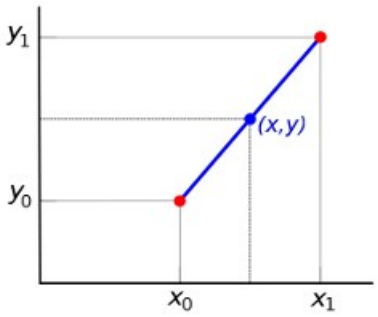
Klatka kluczowa składa się z tablicy zawierającej kąt obrotu oraz długość każdej z kości. Do wykonania animacji poruszania się postaci w prawo wystarczy użyć ośmiu zapętlonych klatek.

Kolejnym etapem tworzenia animacji jest interpolacja klatek kluczowych. Gdyby akcja odgrywana była tylko poprzez czasowe przechodzenie z jednego stanu do następnego, animacja nie byłaby płynna. Interpolacja pozwala na obliczenie stanu szkieletu pomiędzy klatkami kluczowymi pobierając z nich wartości kątów obrotu kości oraz ich długości.

Na potrzeby projektu w zupełności wystarcza interpolacja liniowa. Stan obecnych parametrów kości można wyrazić wzorem:

$$y = y_0 + (x - x_0)\frac{y_1 - y_0}{x_1 - x_0}$$

gdzie
x₀ - czas wystąpienia pierwszej klatki kluczowej
x₁ - czas wystąpienia drugiej klatki kluczowej
y₀ - wartość parametru pierwszej klatki kluczowej
y₁ - wartość parametru drugiej klatki kluczowej
x - obecny czas
y - wartość parametru w obecnej chwili



Rysunek 28: Wykres przykładowej interpolacji liniowej.

Parametrem w tym przypadku może być kąt obrotu kości bądź jej długość.

Interpolacja odgrywa znaczącą rolę w czynieniu animacji bardziej wiarygodną. Posiadając mechanizm kalkulujący transformacje szkieletu między dwiema dowolnymi klatkami kluczowymi, możliwe jest płynne przechodzenie różnych cykli animacji (np. z chodzenia do biegu) bez stosowania dodatkowych stanów.

Manipulacja czasem wystąpienia kolejnych klatek pozwala na dowolne zwalnianie lub przyśpieszanie animacji szkieletu bez żadnych strat jakościowych. Jedynym ograniczeniem jest tylko dokładność liczb zmiennoprzecinkowych opisujących kąt obrotu kości oraz jej długość.

7. Nazwy

7.1 Technika generowania

Każda istniejąca w grze postać lub lokacja posiada swoją nazwę. W celu ich wygenerowania zastosowano algorytm posługujący się własnościami ciągu Markowa. Sposób ten oparty jest w całości na zbiorze danych wejściowych i pozwala uzyskać wyniki będące danymi we wstępnie określonym stopniu podobnymi do tych wchodzących.

7.1.1 Łańcuch Markowa

Procesem Markowa[28] nazwać można każdy ciąg zdarzeń w którym prawdopodobieństwo wystąpienia kolejnego zdarzenia zależy tylko i wyłącznie od *n* zdarzeń bezpośrednio je poprzedzających.

W odniesieniu do generowania wyrazów, korzystając z określonego słownika, własność Markowa pozwala uzyskać słowa brzmiące bardzo podobnie do podanych. Taka metoda ma również swoje minusy, jak np. duże zapotrzebowanie pamięciowe.

Wykorzystanie własności Markowa dla generowania imion	
Zalety	**Wady**
Pełna zależność od danych wejściowych. Różne zbiory wejściowe gwarantują różne wyniki.	Pełna zależność od danych wejściowych. Są one niezbędne do działania algorytmu.
Produkty działania algorytmu są wiarygodne, a uzyskane wyrazy brzmią bardzo podobnie do wejściowych.	Dla zbyt małych zbiorów wejściowych uzyskane wyniki będą często identyczne jak wprowadzone słowa.
Bardzo niska złożoność obliczeniowa oraz czas wykonywania.	Wysokie zużycie pamięci. Przetworzone dane wejściowe muszą permanentnie znajdować się w pamięci operacyjnej.

Tabela 13: Wady i zalety łańcucha Markowa.

[28] Maria Podgórska i in.: Łańcuchy Markowa w teorii i zastosowaniach. Warszawa: Szkoła Główna Handlowa, Oficyna Wydawnicza, 2002.

Algorytm działania łańcucha Markowa dla generowania słów:

1. Określ ilość liter będących bazą do wybierania kolejnych znaków i zapisz ją do zmiennej n.

2. Pobierz kolejny wyraz ze zbioru wejściowego oraz zapisz go jako wyraz aktywny.

3. Dla każdych n kolejno występujących liter w wyrazie aktywnym dokonaj nowego przypisania w tablicy asocjacyjnej:

[podciąg wyrazu aktywnego o długości **n** liter] => [znak występujący zaraz za tym ciągiem]

4. Jeżeli aktywny wyraz nie jest ostatnim wyrazem zbioru wejściowego, wróć do punktu 2.

5. Dokonaj losowego wyboru podciągu znajdującego się w tablicy asocjacyjnej oraz zapisz go jako aktywny podciąg.

6. Z tablicy asocjacyjnej losowo wybierz znak należący do zbioru skojarzonego z aktywnym podciągiem i przyłącz ten znak do jego końca, a następnie usuń pierwszą literę podciągu.

7. Jeżeli wybrany znak jest końcem wyrazu (np. \0 w języku C++) albo wyraz osiągnął pożądaną maksymalną długość - przerwij działanie algorytmu. W przeciwnym wypadku wróć do punktu 6.

Przykład:

Dany jest zbiór wejściowy:
banan, jabłko, kabanos

Uzyskane dane dla konkretnych wartości bazy liter:

n = 1			
banan	jabłko	kabanos	wynikowa tablica
[b] => [a] [a] => [n] [n] => [a] [a] => [n] [n] => [\0]	[j] => [a] [a] => [b] [b] => [ł] [ł] => [k] [k] => [o] [o] => [\0]	[k] => [a] [a] => [b] [b] => [a] [a] => [n] [n] => [o] [o] => [s] [s] => [\0]	[b] => [a, ł, a] [a] => [n, n, b, b, n] [n] => [a, \0, o] [j] => [a] [ł] => [k] [o] => [\0, s] [k] => [o, a] [s] => [\0]
Przykładowe rezultaty		nano babananos łk abłkos	

Tabela 14: Efekty działania łańcucha Markowa dla n=1.

n = 2			
banan	jabłko	kabanos	wynikowa tablica
[ba] => [n] [an] => [a] [na] => [n] [an] => [\0]	[ja] => [b] [ab] => [ł] [bł] => [k] [łk] => [o] [ko] => [\0]	[ka] => [b] [ab] => [a] [ba] => [n] [an] => [o] [no] => [s] [os] => [\0]	[ba] => [n, n] [an] => [a, \0, o] [na] => [n] [ja] => [b] [ab] => [ł, a] [bł] => [k] [łk] => [o] [ko] => [\0] [ka] => [b] [no] => [s] [os] => [\0]
Przykładowe rezultaty:		bananos łko jabanan nananos	

Tabela 15: Efekty działania łańcucha Markowa dla n=2.

n = 3			
banan	jabłko	kabanos	wynikowa tablica
[ban] => [a] [ana] => [n] [nan] => [\0]	[jab] => [ł] [abł] => [k] [błk] => [o] [łko] => [\0]	[kab] => [a] [aba] => [n] [ban] => [o] [ano] => [s] [nos] => [\0]	[ban] => [a, o] [ana] => [n] [nan] => [\0] [jab] => [ł] [abł] => [k] [błk] => [o] [łko] => [\0] [kab] => [a] [aba] => [n] [ano] => [s] [nos] => [\0]
Przykładowe rezultaty:		banan łko abanos jabłko	

Tabela 16: Efekty działania łańcucha Markowa dla n=3.

Analizując powyższe przykłady można wyciągnąć pewne wnioski:

- czym większa wartość zmiennej `n`, tym większy stopień podobieństwa wygenerowanego wyrazu w stosunku do zbioru wejściowego,
- dla małych wartości `n` otrzymujemy wyrazy przypominające konkatenacje losowych liter.

Dwoma parametrami sterującymi algorytmem są ilość liter będących bazą do losowania kolejnych oraz maksymalna długość otrzymanego słowa. Dodatkowo, aby nie otrzymywać

wyrazów zbyt krótkich, można ograniczyć zbiór startowych podciągów tylko do tych, które znajdują się na początkach wyrazów (w podanym przykładzie dla `n=3` otrzymane wyrazy mogłyby zaczynać się tylko od `ban`, `jab` lub `kab`).

7.2 Imiona postaci

Dla każdej z trzech obecnych w grze ras wykorzystano inne dane słownikowe. Dodatkowo bazy danych wejściowych zostały podzielone na męskie i żeńskie, co w ogólności daje 6 różnych słowników. Przyjętymi parametrami dla algorytmu generującego są 2 - jako ilość bazowych liter służących do losowania kolejnych oraz 7 - będąca maksymalną długością uzyskanego wyrazu.

7.2.1 Imiona nordyckie

Bazą do utworzenia zbioru imion nordyckich były imiona mitologicznych bogów skandynawskich. Popularne nazwy jak Thor czy Gunnar dodatkowo zostały połączone z klasycznymi imionami nadawanymi wikingom. Przykładowe wykorzystane słowa to:

```
Alf, Bjorn, Bodvar, Einar, Fridgeir, Glum, Grim, Hakon, Harald,
Knut, Mord, Olaf, Sigurd, Stein, Thord, Thorstein, Valgard
```

Inaczej sprawa wygląda w przypadku imion żeńskich. Oryginalne skandynawskie słowa określające kobiety brzmią bardzo męsko i trudno je od takowych rozróżnić. Sposobem na wybranie bardziej kobiecych imion z jednoczesnym zachowaniem klimatu mroźnej północy jest zastosowanie słownika będącego zbiorem najpopularniejszych i teraźniejszych imion norweskich kobiet, jak np.:

```
Thea, Nora, Victoria, Vilde, Frida, Mia, Sunniva, Hanna, Eline,
Andrine, Tuva, Pernille, Ingeborg, Linn, Malene, Rikke
```

7.2.2 Imiona europejskie

O ile dla przeciętnego europejczyka imiona skandynawskie będą brzmiały podobnie, to trudno ujednolicić słowa europejskie. Istnieją bowiem znaczące różnice między wyrazami dla przykładu niemieckimi, francuskimi, a hiszpańskimi. Aby wybrać neutralny i wiarygodny zbiór danych posłużono się imionami bohaterów literatury fantastycznej.
Przykładowe imiona męskie i żeńskie:

```
Agnar, Bewul, Cardon, Darkkon. Enidin, Folmard, Gryn, Hezak,
Ilgenar, Jex, Kirder, Lackus, Mythik, Nydale, Ospar, Pildoor
```

```
Adorra, Brana, Celoa, Fayne, Helenia, Kassina, Lyna, Mirayam,
Orwyne, Qara, Rivatha, Sennetta, Tressa, Vessere, Zoura
```

7.2.3 Imiona afrykańskie

Dla czarnoskórych postaci w grze zdecydowano się na dobór imion o brzmieniu afrykańskim. Są w to głównej mierze najbardziej popularne imiona nadawane obecnie obywatelom państw takich jak Kongo, Czad lub Nigeria.

Przykłady imion kolejno męskich i żeńskich:

`Abebi, Aziza, Chipo, Falala, Iman, Jendayi, Kafi, Lateefah, Maizah, Malika, Nakeisha, Ramla, Safara, Shasa, Waseme, Zalika`

`Abimbola, Babajide, Chidi, Chima, Emeka, Faraji, Isingoma, Kojo, Masamba, Nkosana, Nnamdi, Paki, Simba, Wekesa, Zuberi`

7.3 Nazwy miast

Nazwy miast są o wiele bardziej skomplikowane niż imiona. W ogólności można podzielić je na:

- związane z położeniem geograficznym (np. South Lake City),
- związane z konkretnymi osobami (np. Washington).

Generowanie nazw wieloczłonowych, w których poszczególne ich części są istniejącymi przymiotnikami, nie jest zgodne z ideą tworzenia wyrazów poprzez ciąg Markowa. Dla celów projektu nazwy miast ograniczone są do jednego, maksymalnie dziesięcioliterowego wyrazu.

Do bazy słownikowej zostały wybrane pomniejsze miejscowości Wielkiej Brytanii. Posiadają one szlachetnie brzmiące pre- i postfiksy, które nadają charakterystycznego patosu brzmieniu nazw wygenerowanych. Przykładami danych wejściowych są:

`Sheffield, Bradford, Newham, Derby, Norwich, Poole, Gloucester`

Kilka wynikowych nazw miejscowości:

`Brawle, Shestmidge, Southon, Watfortham, Yorough, Exet`

7.4 Tytuł gry

Ze względu na specyfikę tytułów samych w sobie, wykorzystanie ciągu Markowa dla generowania nazwy gry nie jest najlepszym rozwiązaniem. Powstałe w ten sposób tytuły nie brzmiałyby wiarygodnie, a baza przykładowych nazw gier musiałaby być nieoptymalnie duża. Wybrane rozwiązanie dla generowania tytułu to odpowiednie łączenie ze sobą zdefiniowanych wcześniej słów (danymi w zbiorach wejściowych są odpowiednio posortowane wyrazy należące do tytułów realnie istniejących gier cRPG), tak, by rezultat wyglądał podobnie do:

`<słowo1> of <słowo2>: <słowo3> <słowo4>`

Użyte zbiory wejściowe:

- Przymiotniki (np. *Final, Sacred, Broken*) - **A1**
- Rzeczowniki wraz z liczbą mnogą oraz biernikiem (*ang*. possesive) (np. *Dragon* | *Dragons* | *Dragon's, Hero* | *Heroes* | *Hero's*) - **A2**
- Rzeczowniki w dopełniaczu (z przedrostkiem "*of*") (np. *of Fate, of Heaven, of War*) - **B1**
- Rzeczowniki podrzędne (pojawiające się najczęściej jako drugie słowo w tytule) wraz z liczbą mnogą (np. *Quest* | *Quests, Fantasy* | *Fantasies*) - **B2**

Algorytm, kierując się zbiorem reguł, dopasowuje wylosowane z konkretnych grup wyrazy w celu uzyskania maksymalnie wiarygodnego tytułu. Poniższa lista przedstawia niektóre z reguł:

- Jeśli pierwszym słowem jest wyraz z grupy **B2**, preferuj po nim wyraz z grupy **B1.**
- Nie stosuj biernika w drugiej części tytułu, jeżeli występowała ona w pierwszej.
- Jeśli pierwsze słowo jest w liczbie mnogiej, następne może należeć tylko do grupy **B1.**
- Nie stosuj liczby mnogiej, jeśli pierwsze słowo jest biernikiem (np. zabezpieczaj przed konstrukcjami typu "*Hero's Days*")
- Odrzuć nieelegancko wyglądające kombinacje;
  - jeśli występują obok siebie dwa słowa, których mianownik liczby pojedynczej jest ten sam,
  - jeśli pierwsza litera drugiego słowa jest taka sama jak ostatnia pierwszego, oraz ta litera nie jest samogłoską.
- Stosuj szansę na dodanie przedimka *The* lub *A/An.*
- Stosuj szansę na dodanie rzymskiego numerału do tytułu.

Wygenerowane przykładowe tytuły:

```
Dragon's Dusk: The Hero Blade
The Sanctums of Life VI: Dark Hero
Spells of Avalon: Ogre's Spell
Valkyrie Swords: Dragon Paths
Nightmare Devil IV: A Dragon's Quest
Sonatas of War: Ages of Arcadia
```

8. Efekty specjalne i czcionka

8.1 Zastosowanie w praktyce

Obiekty występujące w grze oraz posiadające stały kształt, łatwo reprezentować poprzez rysowanie prostych obiektów składających się z podstawowych brył geometrycznych. Ciecze będące we względnym spoczynku (np. reprezentacja rzeki płynącej przez kontynent) można odzwierciedlić płaszczyzną odpowiadającą tafli wody. Dynamiczne generowanie staje się bardziej skomplikowane jeżeli pojawia się potrzeba odwzorowania dymu, ognia, lub chociażby deszczu. Powszechnie stosowaną metodą uzyskiwania takich efektów są systemy cząsteczkowe[29].

Czcionka jest jednym z najrzadziej generowanych proceduralnie elementów w grach. Nawet najstarsze gry, opierające się wyłącznie na znakach wyświetlanych w konsoli poleceń, korzystały z czcionek systemowych. Założenie projektu o tym, że wszystkie elementy generowane są proceduralnie wymaga jednak algorytmicznego utworzenia fontów.

8.2 Płomień

Opisane w rozdziale 5. labirynty tworzone pod miastem, oświetlone są płomieniami pochodni umieszczonymi na ścianach. Aby najlepiej odzwierciedlić taki rodzaj ognia, należy przyjąć kilka założeń:

- źródłem płomienia jest jeden punkt (szczyt pochodni),
- ogień wędruje od swojego źródła w górę, lecz dopuszczalny stopień odchylenia od osi pionowej to około $45°$,
- źródłowy kolor płomienia to czerwony, następnie przechodzi on w żółty, aż do pełnego zaniknięcia.

Stosując te reguły w odniesieniu do systemu cząsteczkowego, tworzymy strukturę która w punkcie źródłowym generuje cząsteczki. Następnie wyrzuca je z pewną startową prędkością w kierunku górnym z możliwością odchylenia o kąt należący do przedziału $<-45°, 45°>$. Kolor cząsteczki zmienia się wraz z upływem czasu, od czerwonego do żółtego, równocześnie zwiększając swoją przeźroczystość. Pojedyncze punkty

[29] William T. Reeves: Particle Systems - A Technique for Modeling a Class of Fuzzy Objects. Computer Graphics 17:3 pp. 359-376, 1983 (SIGGRAPH 83)

reprezentowane są na ekranie poprzez koła o kolorze oraz pozycji zależnych od właściwości cząsteczki. Znaczącym parametrem jest również czas życia elementu, różnicuje on m.in. wysokość płomienia.

8.3 Zjawiska pogodowe

W odróżnieniu do płomienia pochodzącego ze źródła punktowego, opad atmosferyczny zazwyczaj posiada równomierny rozkład na danym obszarze. Zastosowanie systemu cząsteczkowego jest dobrym wyjściem również w tym przypadku, gdyż pozwala na swobodne przypisanie pojedynczych kropli lub płatków do generowanych cząsteczek.

Założenia:

- opad jest względnie równomierny na całym obszarze,
- krople i płatki przemieszczają się w kierunku dolnym z możliwym odchyleniem o $45°$, musi być ono jednakowe dla wszystkich cząstek w danej jednostce czasu,
- reprezentacją kropli deszczu jest przezroczysta linia, lecz w momencie zetknięcia z ziemią przedstawiona jest jako okrąg symbolizujący rozprysk,
- reprezentacją płatku śniegu jest biały okrąg zmniejszający swój promień wraz z upływem czasu,
- opad nie może być obecny wewnątrz budynków.

Rodzaj i częstotliwość występowania opadu uzależnione są od biomu w którym znajduje się miasto. Jedynym miejscem gdzie pojawiają się opady śniegu jest pustynia lodowa, prawdopodobieństwo opadów deszczu przedstawione jest w poniższej tabeli.

Biom	Prawdopodobieństwo opadu deszczu
Sawanna	50,00%
Las tropikalny	80,00%
Tajga	20,00%
Pustynia	0,00%
Las liściasty	50,00%
Tundra	20,00%
Step	20,00%
Las twardolistny	50,00%
Pustynia polarna	0,00%

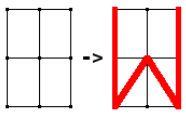
Tabela 17: Prawdopodobieństwo opadu deszczu w mieście w zależności od biomu.

8.4 Czcionka

Napisy są nieodłącznym elementem każdej gry *cRPG*. Dzięki nim możliwa jest intuicyjna nawigacja po menu gry, dialogi z napotkanymi postaciami, opisy zachodzących zdarzeń fabularnych czy chociażby wyświetlanie imion i nazw przedmiotów. Gra składająca się z elementów wyłącznie generowanych musi zapewniać również tworzenie swojej czcionki.

Kroje ogólnie dostępnych fontów są wysoce zróżnicowane, część z nich sprzyja metodzie generowania algorytmicznego, są to głównie czcionki typu *courier* w których każdy znak posiada taką sama szerokość. Zdecydowanie prościej jest również generować kroje składające się tylko z linii prostych, tak zwane fonty typu *square*.

Każdy znak generowanej czcionki da się utworzyć poprzez ustalenie połączeń miedzy punktami tworzącymi kratownicę o wymiarach `3x3`.



Rysunek 29: Szablon dla rysowania znaku.

Rysunek 30: Wygenerowana proceduralnie czcionka.

8.4.1 Parametryzacja

Generowana czcionka pozwala na pełną modyfikację jej parametrów. Dzięki temu możliwe jest ustalanie wzorców odpowiednich dla różnych zastosowań. Na zestaw cech opisujących konkretny wzorzec składa się:

- wysokość znaku,
- szerokość znaku,
- kolor,
- grubość linii,
- kolor obramowania znaku,
- grubość obramowania.

Rysunek 31: Przykład uzyskanych fontów.

9. Zagadnienia fabularne

9.1 Przedmioty

Każda gra cRPG dysponuje szerokim wachlarzem przedmiotów spotykanych w grze. Głównie są to bronie wykorzystywane przez bohatera do walki z przeciwnikami, mikstury modyfikujące statystyki oraz przedmioty fabularne.

W opisywanym projekcie największy nacisk położono na przygodowy aspekt gry, oznacza to, że ma umożliwiać ona graczowi swobodną eksplorację wygenerowanego świata. Wiąże się to z dużą gamą dostępnych przedmiotów, niekoniecznie użytecznych, lecz będących odbiciem świata realnego.

9.1.1 Inwentarz

Każdy przedmiot występujący w grze zalicza się do jednej z możliwych kategorii, takich jak:

- **Ubrania**
  Odzienie znajdujące się w ekwipunku postaci wpływa na jej reprezentację graficzną. Gra oferuje możliwość przebierania sterowanej postaci w celu wywołania konkretnych zachowań u postaci niezależnych.

  Przykład: Przebranie się za kapłana zwiększa zaufanie rozmówcy.
  Sposób nabycia: Kupcy lub otwarte szafy w domach.

- **Broń**
  W grze występuje kilka rodzajów broni. Nie służą one graczowi do walki lecz znajdują inne zastosowanie.

  Przykład: Użycie buzdyganu powoduje zniszczenie kłódki uniemożliwiającej dostęp do zamkniętego kufra.
  Sposób nabycia: Kowal broni i skrzynie.

- **Jedzenie**
  Pożywienie zaspokaja rosnący wraz z upływem czasu głód bohatera.

Przykład: Zjedzenie sernika obniży poziom głodu o 2 punkty w dobowej skali 24. punktowej.
Sposób nabycia: Kupcy, kucharze, beczki oraz drzewa.

- **Picie**
  Napoje alkoholowe zwiększają poziom nietrzeźwości bohatera, co skutkuje efektem rozmycia ekranu. Alkohol wykorzystywany jest w grze do pozyskiwania informacji.

  Przykład: Wypicie kilku kufli piwa razem z oberżystą skutkuje zdradzeniem przez niego informacji o ukrytym przejściu do piwniczki.
  Sposób nabycia: Oberżyści.

- **Biżuteria**
  Ozdoby są dobrem ogólnym i nie mają żadnych właściwości poza wartością pieniężną oraz możliwością wręczania jej jako prezent.

  Przykład: Sprzedanie znalezionej bransolety kupcowi daje nam złoto, a ofiarowanie jej wybranej kobiecie pozyskuje jej względy.
  Sposób nabycia: Kupcy i kufry skarbów.

- **Przedmioty fabularne**
  Posiadanie rzeczy specjalnych, będących unikatowymi w grze, jest warunkiem koniecznym do ukończenia powierzonych graczowi zadań.

  Przykład: "Napotkany sędziwy pan zagubił swoją fajkę." Zobowiązaniem gracza jest jej odnalezienie.
  Sposób nabycia: W pełni zależne od aktualnego zadania.

- **Pozostałe**
  Przedmioty codziennego użytku nie mają żadnego konkretnego zastosowania w rozgrywce, mogą jednak służyć jako środki do osiągnięcia dalszych celów.

  Przykład: Posiadanie pustej butelki w ekwipunku umożliwia zebranie magicznej rosy, będącej przedmiotem fabularnym.
  Sposób nabycia: Kupcy oraz szafki.

9.1.2 Interakcja z przedmiotami

Każda postać występująca w grze ma swój własny, wygenerowany inwentarz. Podczas rozgrywki, gracz ma możliwość wpływania na swój ekwipunek poprzez menu wywoływane wciśnięciem klawisza inwentarza. Akcje możliwe do wykonania na przedmiotach to:

- **Wyrzucenie**
  Powoduje trwałe usunięcie przedmiotu z ekwipunku. Po wybraniu tej opcji gracz nie jest w stanie przywrócić wyrzuconego przedmiotu.

- **Założenie ubrania**
  Bez względu na płeć i zawód, gracz może zakładać dowolne ubrania. Nie ma ograniczenia co do ilości noszonego na sobie odzienia.

- **Zdjęcie ubrania**
  Każde noszone przez bohatera ubranie może być w dowolnym momencie zdjęte, a on sam być rozebranym do naga.

- **Kupno**
  Postacie posiadające możliwość sprzedaży przedmiotów mają dodatkowy ekwipunek będący ich zapleczem. Każdy przedmiot posiada swoją określoną wartość w wewnętrznej walucie gry - złocie.

- **Sprzedaż**
  Handlarze znajdujący się na rynku oferują możliwość kupowania przedmiotów od gracza za cenę stanowiącą połowę wartości przedmiotu. Gracz może sprzedać dowolne przedmioty nie będące przedmiotami fabularnymi.

- **Kradzież**
  Opcja kradzieży pozwala graczowi na wgląd w inwentarz wybranej postaci. Każdy przedmiot będący w jej ekwipunku, poza założonymi ubraniami, jest podatny na kradzież. Gracz może spróbować przywłaszczyć sobie wybrany przedmiot, następuje wtedy test na udaną kradzież, będący wylosowaniem wartości "udana" bądź "nieudana". Jeśli kradzież powiodła się, wybrany przedmiot staje się własnością bohatera, w przeciwnym przypadku rabowana postać alarmuje o kradzieży. Jeśli bohater został przyłapany na gorącym uczynku, okradana postać nie będzie chciała mieć z nim żadnych kontaktów, co uniemożliwi rozmowę z nią lub handel (w przypadku kupców).

9.2 Fabuła

Na fabułę gry składają się zadania przydzielane do wykonania graczowi przez postacie niezależne - *NPC* (*ang. non-player character*, to każda postać występująca w grze, której gracz nie może kontrolować w sposób bezpośredni. Termin NPC w zawężeniu do tej pracy określa postacie w grze, które mają możliwość przydzielenia graczowi zadań do wykonania). Historie przekazywane przez postacie niezależne nawiązują do świata w którym porusza się gracz oraz tworzą stosowną atmosferę gry. *NPC* losowani są dla każdego nowego miasta, a rodzaj zadania którym dysponują określany jest na podstawie ich zawodu.

Rdzeniem każdego zadania, które ma zostać wykonane przez bohatera, jest drzewo dialogowe. Składa się ono z sentencji wypowiadanych przez *NPC* oraz odpowiedzi udzielanych przez gracza. Każda wypowiedź postaci niezależnej zakończona jest co najmniej dwiema opcjami dialogowymi z których gracz, aby kontynuować, musi wybrać dokładnie jedną. Wybranie konkretnej opcji rozmowy rozwija skojarzoną z nim gałąź drzewa dialogowego. Liśćmi takiej struktury są flagi decydujące o postępie zadania.

Drzewo dialogowe niekoniecznie musi być drzewem w konwencji struktury algorytmicznej. Możliwość istnienia pętli w której wybrana opcja dialogowa rozwija gałąź znajdującą się na niższym poziomie drzewa uogólnia taką reprezentacje do struktury grafu.

9.2.1 Implementacja zadań

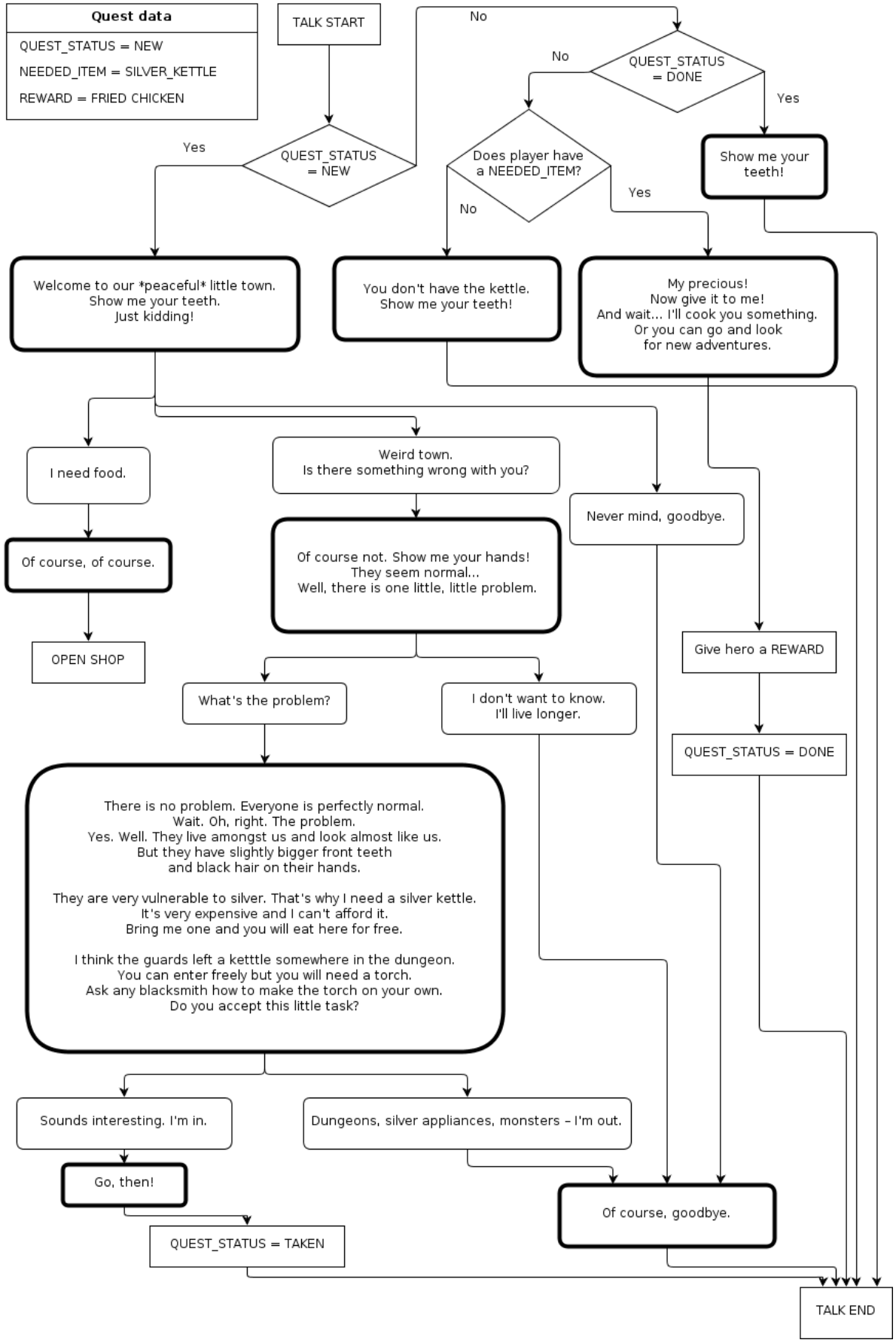
Zadanie do wykonania (*ang*. quest) składa się z drzewa dialogowego oraz danych dodatkowych typu lista przedmiotów do zdobycia lub lista postaci z którymi należy przeprowadzić rozmowę. Takie nieszczegółowe podejście wymusza zastosowanie programowania generycznego, w którym kod programu musi być pisany w taki sposób, aby mógł dostosować się do konkretnej implementacji. Głównym nurtem w dziedzinie implementacji zadań w kod programu jest wczytywanie ich jako skrypty z oddzielnych plików. Jednym z założeń projektu jest pełne uwolnienie aplikacji od zasobów zewnętrznych. Rozwiązaniem które spełnia taki warunek oraz wspomaga programowanie generyczne, jest polimorfizm. Stąd też zadania w grze są zaimplementowane w osobnych klasach dziedziczących po typie abstrakcyjnym.

9.2.2 Przykładowe zadanie

W prezentowanym przykładzie gracz dostaje od kucharza zadanie, aby znaleźć srebrny czajnik. Jeżeli bohater wróci z pożądanym przedmiotem w swoim ekwipunku - zostanie wynagrodzony. Poniższy diagram został wykonany na potrzeby projektu w darmowej aplikacji Diagramly[30]. Prezentuje on szkielet przykładowego zadania, na który składają się:

- kwestie wypowiadane przez:
  - NPC,
  - bohatera,
- zmienne zadaniowe (np. poszukiwany przedmiot),
- warunki zmiany statusu zadania.

[30] http://www.diagram.ly/



Rysunek 32: Jedno z zaimplementowanych drzew dialogowych.

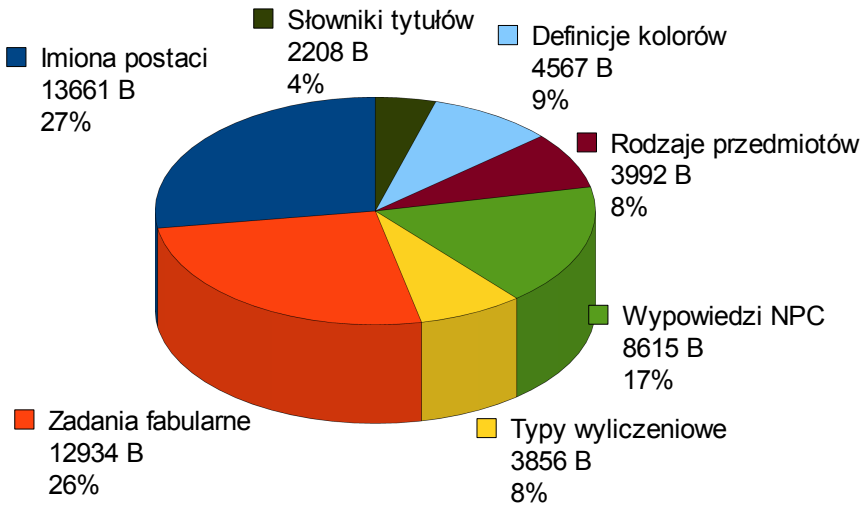
10. Optymalizacja aplikacji

10.1 Rozmiar aplikacji

Rozmiar gotowej aplikacji analizowany jest najszczególniej z uwagi na to, że zastosowanie generatorów proceduralnych drastycznie ogranicza ilość pamięci trwałej wymaganej przez program.

10.1.1 Dane wewnętrzne

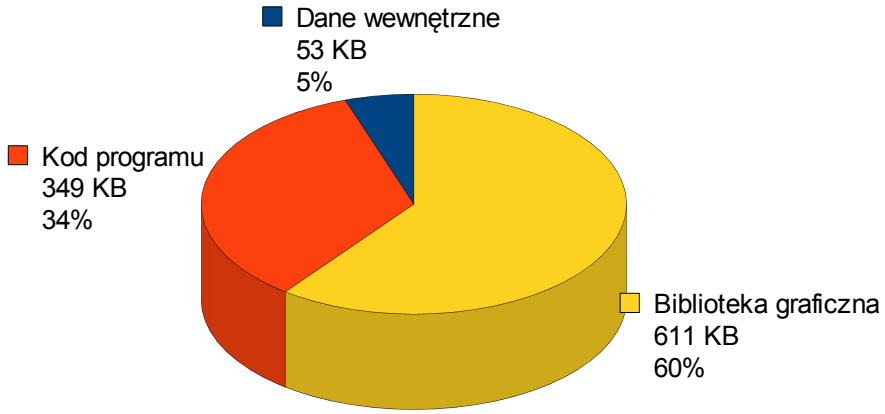
Głównym założeniem tworzonej gry jest całkowity brak zasobów wczytywanych z zewnętrznych źródeł - plików oraz danych pobieranych z internetu. Ogranicza to rozmiar powstałej aplikacji do jej kodu wykonywalnego. Oczywistym jest, że pomimo proceduralnego charakteru programu, zawiera on dane wewnętrzne (np. lista przykładowych imion postaci). Wpływają one znacząco na rozmiar wynikowy pliku wykonywalnego, gdyż nie są przekształcane na instrukcje maszynowe zajmujące optymalnie małą ilość pamięci.



Rysunek 33: Procentowy udział poszczególnych danych w zbiorze danych wewnętrznych.

10.1.2 Komponenty pliku wynikowego

Poza instrukcjami oraz danymi wewnętrznymi na rozmiar gry ma wpływ również załączona biblioteka Allegro5 obsługująca funkcje graficzne oraz systemowe (np. tworzenie okna). Poniższy wykres prezentuje procentową zawartość poszczególnych komponentów w pliku wykonywalnym.



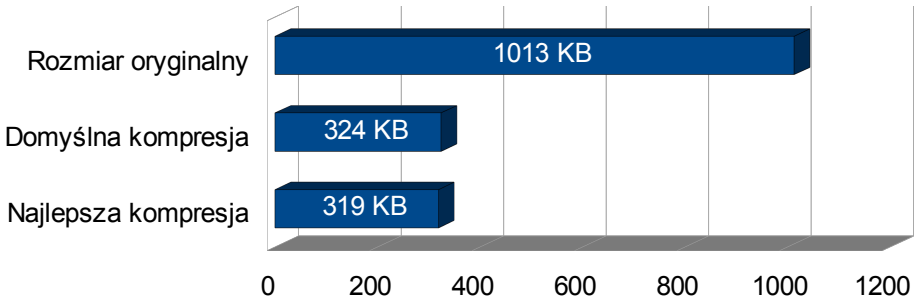
Rysunek 34: Procentowy udział poszczególnych komponentów w rozmiarze pliku wynikowego.

Pomiary dokonane zostały po kompilacji w Microsoft Visual Studio 2010 Premium (MSVC) w systemie Windows 7 Professional z zastosowanymi domyślnymi opcjami kompilacji w trybie *Release* (Synteza kompilatora MSVC2010 wraz z biblioteką Allegro5 oferuje dwa domyślne tryby kompilacji - *Debug* i *Release*. Tryb *Debug* umożliwia lepsze raportowanie błędów aplikacji kosztem gorszej wydajności, natomiast *Release*, jak sama nazwa wskazuje, jest opcją służącą wydaniu końcowej wersji aplikacji działającej maksymalnie wydajnie). MSVC posiada szereg dodatkowych możliwości automatycznej optymalizacji kodu z ukierunkowaniem na wielkość pliku wynikowego lub prędkość działania aplikacji. Najlepsza konfiguracja pozwoliła zminimalizować rozmiar kodu wynikowego z 1156 KB do 1013 KB.

10.1.3 Zewnętrzna kompresja pliku

Programy komputerowe służące minimalizowaniu rozmiarów plików *exe* nazywa się potocznie packer'ami. Rolą packer'a jest kompresja instrukcji oraz danych zawartych w pliku wykonywalnym w sposób podobny do kompresji *zip* czy *rar*. W przypadku tworzonej aplikacji użyty został Ultimate Packer for eXecutables (UPX). Rozmiar pliku wykonywalnego poddanego kompresji przez UPX zmniejszył się ponad trzykrotnie.

UPX korzysta z algorytmu kompresującego UCL. Został on zaprojektowany tak, by zaimplementowany dekompresor zajmował zaledwie kilkaset bajtów. UCL nie wymaga również alokowania żadnej dodatkowej pamięci dla dekompresji więc zużycie pamięci dla pliku spakowanego poprzez UPX nie zwiększa się.

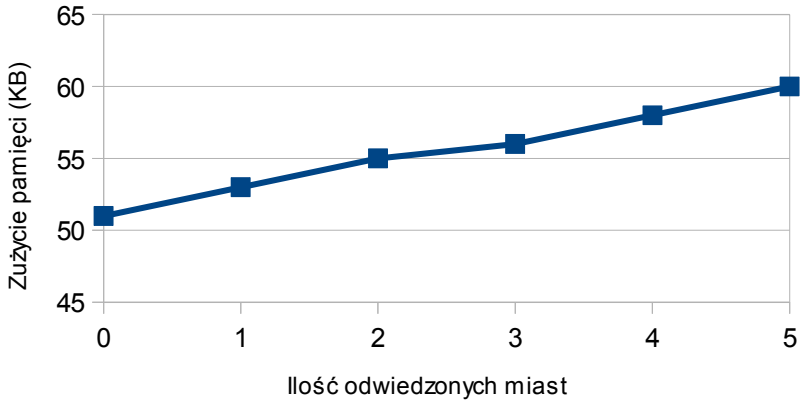


Rysunek 35: Rozmiar aplikacji wynikowej dla różnych poziomów kompresji w UPX.

10.2 Zużycie pamięci

Największy wpływ na zużycie pamięci operacyjnej w aplikacji ma wygenerowana grafika. Reprezentacja graficzna każdego pola gry jest unikalna, stąd też nie jest możliwe wielokrotne powielanie gotowych obrazów, tak jak w większości tego typu produkcji. W celu ograniczenia ilości przechowywanej grafiki w pamięci operacyjnej do optymalnego minimum, wszelkie elementy znajdujące się poza ekranem i widocznością gracza są usuwane. Obraz, który znajduje się na ekranie, składa się z siatki obrazów wygenerowanych dla poszczególnych pól. Podczas przemieszczania się bohatera po mapie, pola pojawiające się na ekranie rysowane są na miejsce pól z niej schodzących, a reszta przesuwana jest o odpowiedni wektor.

Drugorzędnymi, jednakże kluczowymi danymi rezydującymi w pamięci, są dane logiczne. Wśród nich znajdują się struktury przechowujące układ pól na mapie, aktualne pozycje postaci w grze, czy zawartość kufrów w katakumbach. Rozmiar danych logicznych ma charakter przyrostowy. Oznacza to, że raz wygenerowana struktura zostaje w pamięci do końca działania aplikacji. Przykładowo, gdy gracz wejdzie do miasta, a następnie je opuści, dane o tej miejscowości będą ciągle rezydowały w pamięci. Takie rozwiązanie eliminuje problem przywracania wartości domyślnych lokacji za każdym razem gdy zostanie ona odwiedzona przez gracza.



Rysunek 36: Zużycie pamięci operacyjnej przez aplikację po odwiedzeniu kolejnych lokacji.

10.3 Zużycie czasu procesora

Jak ujęto w poprzednim paragrafie - wszelkie elementy gry generowane są tylko raz. Proces algorytmicznego tworzenia komponentów jest najbardziej czasochłonną czynnością aplikacji, dlatego należało ograniczyć ją do minimum. Procedury wykonywane jednorazowo nie powinny być uwzględniane w pomiarach czasu procesora w przeciwieństwie do funkcji wywoływanych w czasie rzeczywistym. Najbardziej obciążającymi procesor procedurami są instrukcje rysujące. Głównie z uwagi na to, że muszą w szybkim tempie obsługiwać zmiany na dużych tablicach danych (w tym wypadku - bitmapach).

Wąskim gardłem aplikacji okazuje się zawartość pamięci cache instrukcji. Wiadomym jest już, że największe obciążenie procesora to rysowanie kształtów, czyli nic innego jak wywołania funkcji graficznych. Skutkuje to odciążeniem aplikacji z przetwarzania danych. Minimalizacja jej rozmiaru optymalizuje również rozmiar wywoływanych instrukcji, z czego można wywnioskować, że czym mniejszy rozmiar aplikacji na dysku, tym będzie ona szybciej działać.

Również same instrukcje rysujące muszą być optymalizowane podczas wywoływania. Biblioteka graficzna oferuje systemowe rysowanie kształtów prymitywnych, tj. najprostszych m.in. figur geometrycznych, których procedury rysujące są domyślnie zaimplementowane w bibliotece graficznej. W przypadku Allegro5 do prymitywów zalicza się; prostą, trójkąt, prostokąt oraz elipsę O ile dodawanie do bitmapy prymitywów, następuje w sposób błyskawiczny (ponieważ wykorzystywana biblioteka graficzna współpracuje bezpośrednio z układem GPU i pamięcią wideo), o tyle funkcje modyfikujące bądź sprawdzające kolory poszczególnych pikseli bitmapy, wymagają zastosowania specyficznego podejścia. Biblioteka Allegro5 umożliwia tzw. blokowanie obrazu. Procedura ta daje szybki i bezpośredni dostęp do tablicy kolorów reprezentującej bitmapę. Dla zablokowanego obrazu, pobieranie oraz ustawianie wartości kolorów poszczególnych pikseli jest optymalnie szybkie, natomiast procedury rysujące prymitywy zajmują o wiele więcej czasu. W aplikacji, do graficznej reprezentacji obiektów użyte są zarówno figury prymitywne jak i rysowane pojedyncze piksele, co wymusza ciągłe blokowanie i zwalnianie bitmap. Z punktu widzenia obciążenia procesora właśnie te funkcje zajmują najwięcej czasu.

Oczywiście w aplikacji znajdują się również elementy czysto logiczne, które mają niemały wpływ na prędkość działania gry. Najistotniejsze w tym wypadku są obliczenia funkcji trygonometrycznych; każdy logiczny szkielet postaci składa się z odpowiednio poobracanych kości, natomiast każdy obrót wymaga wywołania funkcji cosinus i sinus dla obliczenia wypadowych wierzchołków. Uogólniając - każda dodatkowa animowana postać na ekranie zwiększa zużycie czasu procesora w odczuwalnym dla aplikacji stopniu.

10.4 Wieloplatformowość

Jedną z głównych zalet biblioteki Allegro5 jest jej zgodność z trzema wiodącymi platformami - Windows, Linux oraz iOS. Dzięki temu kod napisany przy użyciu tego systemu jest natychmiastowo gotowy do kompilacji pod każdą z wymienionych platform. Aplikacja tworzona jest domyślnie na system Windows, lecz w przyszłości planowane jest wydanie wersji działającej w systemie Linux. Preferowaną dystrybucją jest tutaj Ubuntu, lecz gra zostanie uruchomiona na każdym standardowym jądrze systemu Linux z obsługą

środowiska graficznego Gnome lub KDE. Kompilacja na platformę Apple iOS nie jest przewidziana ze względu na odpłatność środowiska programistycznego.

Należy pamiętać, że sama wieloplatformowość biblioteki wspomagającej aplikację nie wystarczy do pełnej przenośności kodu. Aby aplikację dało się skompilować w innym środowisku programistycznym musi być ona niezależna od środowiska. Domyślne ustawienia projektu w MSVC zakładają wykorzystywanie pakietu redystrybucyjnego C++. Jest to dodatkowy komponent systemu, który niekoniecznie musi znajdować się w każdej jego instalacji. Często spotykane są również dyrektywy preprocesora optymalizujące aplikację lub w pewien sposób przyśpieszające pracę programisty. Opisywana aplikacja pozbawiona jest tego typu zależności i kodowana w ścisłej zbieżności ze standardami języka C++. Jedynym wyjątkiem jest stosowana dyrektywa `#pragma once`, jednakże jest ona obecnie wspierana przez wszystkie wiodące środowiska programistyczne.

10.5 Optymalizacja kodu programu

Aby uczynić kod maksymalnie czytelnym oraz minimalnie uwikłanym w procesie jego wytwarzania użyto szeregu standardów oraz notacji, na przykład:

- Kod programu jest w znacznej większości obiektowy, jednakże specyfikacja języka C++ wymaga nazwania go hybrydowym.
- Nazwy zmiennych, funkcji, struktur oraz komentarze kodu pisane są w języku angielskim.
- Wszelkie zmienne, nie będące automatycznymi, posiadają logiczne nazwy umożliwiające łatwą identyfikację ich zastosowania.
- Bardziej skomplikowane algorytmy lub procedury zostały otoczone komentarzami blokowymi.
- Zachowane są standardy notacji kodu:
  - Jeżeli jakakolwiek nazwa składa się z więcej niż jednego wyrazu, drugi i każdy następny wyraz rozpoczynany jest wielką literą i następującymi małymi (np. `TileDrawer`, `currentHeroEffect`, `aIsPassable`).
  - Nazwy klas oraz metod pisane są początkową wielką literą (np. `class Character`, `int GetCharacterHeight()`).
  - Nazwy pól klasy rozpoczynają się podkreślnikiem i małą literą (np. `float _gold`, `bool _isMoving`).
  - Argumenty funkcji zaczynają się przedrostkiem `"a"` i wielką literą (np. `string aName`, `float aHeroSpeedX`).
  - Nazwy typów wyliczeniowych rozpoczynają się wielką literą i przedrostkiem "`E`" (np. `EItemType`, `EBiome`), wartości enumeratorów pisane są wyłącznie wielkimi literami z wyrazami rozdzielanymi podkreślnikiem oraz pierwszym wyrazem będącym nazwą typu wyliczeniowego (np. `ITEM_TYPE_FOOD`, `BIOME_SAVANNA`).
  - Pozostałe zmienne nazywane są od małej litery (np. `int i`, `bool isRaining`).
- Podczas pracy nad grą, kod źródłowy był regularnie synchronizowany z repozytorium SVN. Umożliwiło to pełną kontrolę nad zmianami w kodzie,

możliwość swobodnej edycji projektu na różnych komputerach, i, co najważniejsze, ciągłe wykonywanie kopii bezpieczeństwa kodu.

- Wszelkie pliki znajdujące się w projekcie (37 nagłówków *h* oraz 40 plików źródłowych *cpp*) mieszczą się fizycznie w jednym katalogu projektu. Pozwala to na bezproblemowe i intuicyjne załączanie plików poprzez dyrektywę `#include`. Jednakże, aby zachować porządek w funkcjach wykonywanych przez odpowiednie klasy, pliki pogrupowane są logicznymi filtrami wewnątrz środowiska MSVC.

11. Wnioski

Wnioski wyciągnięte z projektu najlepiej przedstawić w formie analizy porównawczej elementów, które warto tworzyć proceduralnie w grach komputerowych oraz tych, których generowanie mija się z celem. Zestawienie to utworzone jest na podstawie algorytmów zastosowanych w projekcie, efektów ich pracy oraz nakładu roboczogodzin programisty w celu ich implementacji. Niestety ze względu na założenie projektu o tym, że generowany ma być każdy element rozgrywki, wybrane zostały tylko czołowe algorytmy dla każdego aspektu gry.

Elementy gry, które warto generować proceduralnie:

- **Mapa świata**
  Szum Perlina pozwala na wiarygodne odwzorowanie wysokości terenu i jest powszechnie używany dla generowania map w różnego rodzaju grach. Korzystnym zabiegiem jest nawet jednorazowe użycie algorytmu generowania mapy, która będzie następnie modyfikowana ręcznie przez deweloperów gry. Produkcje opierające się na elementach strategicznych bardzo często posiadają w zasobach dużą ilość map. Generowanie takich plansz oszczędza producentom czas oraz gwarantuje niepowtarzalność rozgrywki.

- **Labirynty**
  Ciągi podziemnych korytarzy stosowane są w grach komputerowych od samego początku ich istnienia. Nawet w produkcjach najwyższej klasy wydawanych obecnie istnieje zaimplementowany tzw. system generowania lochów (np. *Diablo III, Legend of Grimrock*). To właśnie możliwość penetrowania losowo tworzonych jaskiń oraz katakumb zagwarantowała takiej grze jak *Diablo II* rzesze fanów nawet 12 lat po jej premierze. Sam algorytm generujący jest również stosunkowo łatwy w implementacji i parametryzacji.

- **Postacie niezależne**
  Ten element warto generować tylko wtedy, gdy w grze występuje bardzo dużo postaci lub gdy istoty tworzone są dynamicznie w zależności od sytuacji. W wielu grach (jak chociażby popularnej *Assassin's Creed*) pojawia się zjawisko "ataku klonów", w którym gracz odnosi wrażenie, że nieustannie napotyka te same postacie. Proceduralne generowanie chociażby elementów ubioru postaci pozawala wyeliminować ten problem.

- **Animacja szkieletowa**
  Taki rodzaj animacji opłaca stosować się, kiedy gracz ma bezpośredni wpływ na animacje

odgrywane w grze (np. wybuchająca beczka odrzuca bohatera, a jego kończyny realistycznie reagują w trakcie kolizji ze ścianą - jest to tzw. system *ragdoll*[31]). Obecnie animacja szkieletowa jest prosta w implementacji i wspierana sprzętowo przez GPU.

- **Imiona**
  Spośród dziedziny generowania nazw tylko tworzenie imion daje zadowalające rezultaty. Ciąg Markowa jest tutaj jednym z proponowanych rozwiązań, lecz nie należy ograniczać się jedynie do niego. Generowanie imion dobrze współgra z generowaniem postaci oraz dodaje wiarygodności w tworzonym świecie gry. Unikalna osoba z unikalnym imieniem jest bardziej realna niż kolejna kopia tego samego modelu postaci o imieniu *Mieszkaniec*.

- **Efekty specjalne**
  Uwzględnienie tego komponentu jest w odniesieniu do dzisiejszego rynku gier formalnością. W starszych grach wideo, efekty specjalne tworzono różnymi wyszukanymi metodami, tylko po to, by możliwie ograniczyć zużycie czasu procesora i pamięci. Rozwój techniki umożliwił realistyczne modelowanie efektów cząsteczkowych poprzez swobodne tworzenie tysięcy kopii obiektów i zarządzanie nimi w niemalże każdym obiegu głównej pętli gry.

Elementy gry, których nie warto generować proceduralnie:

- **Mapy miast**
  Generowanie miast ma sens tylko wtedy gdy jest to głównym zagadnieniem aplikacji (np. *SimCity* lub *Pixel City*). W każdym innym przypadku możliwości rozłożenia topograficznego sieci dróg, budynków czy obiektów użyteczności publicznej są tak zróżnicowane, że bardzo trudno uzyskać wiarygodny wygląd miasta. Historycznie miasta powstawały z małych osad, które następnie rozwijane były w coraz większe metropolie. Dobry algorytm generujący miasta musiałby symulować ten proces.

- **Czcionka**
  Ilość dostępnych krojów czcionek, również na licencjach bezpłatnych, w zupełności wystarczy dla lwiej części projektów. Generowanie czcionki wymaga algorytmicznego podejścia dla każdego znaku, a utworzenie fontu wyglądającego np. jak pismo odręczne, jest wyjątkowo czasochłonne i nieopłacalne.

- **Fabuła**
  Komputerowe wytworzenie wiarygodnej i epickiej historii to zagadnienie z dziedziny sztucznych sieci neuronowych i na dzień dzisiejszy jest tematem dość abstrakcyjnym w odniesieniu do gier komputerowych. Jeżeli produkcja ma zainteresować gracza linią fabularną to zdecydowanie musi być ona nakreślona przez człowieka. Algorytm komputerowy może służyć w tym wypadku pomocą i wygenerować proste zmienne dla danej historii, np. nazwa szukanego przedmiotu, lub pokrewieństwo osoby zlecającej zadanie w stosunku do bohatera.

- **Grafika gry**
  W całym projekcie to generowanie i dynamiczne rysowanie grafiki do gry zajęło najwięcej czasu. Wytworzenie obrazów i modeli w programie graficznym trwa z reguły nie dłużej niżby to zajęło poprzez pisanie procedur rysujących, a możliwość kontroli grafiki

[31] Richard Wyckoff: Postmortem - DreamWorks Interactive's Trespasser. Game Developer, May 14, 1999

zdecydowanie zwiększa jej jakość wynikową na korzyść wytwarzania ręcznego. Dynamiczne generowanie grafiki ma jednak sens w kilku przypadkach.

- Jeśli istnieje potrzeba narysowania reprezentacji graficznej fraktali,
- wymagane jest by rysowane obiekty były unikalne (np. drzewa),
- lub należy wygenerować wiele prostych kształtów o ściśle określonych parametrach i pozycjach (np. szachownica).

Kolejną analizą którą warto dokonać w tym rozdziale jest bilans zysków i strat dla wykorzystania elementów generowanych proceduralnie w grze. Właściwości te są prawdziwe dla każdego generowanego komponentu.

Zalety elementów generowanych proceduralnie:

- Brak danych zewnętrznych, nie istnieje konieczność pisania dodatkowych importerów oraz parserów.
- Wielokrotnie mniejszy rozmiar elementu - zamiast danych podane są tylko parametry wejściowe algorytmu.
- Możliwość losowego doboru parametrów.
- Uniwersalność algorytmów generujących - ten sam zestaw instrukcji może wytworzyć imię bohatera jak i nazwę miasta.

Wady elementów generowanych proceduralnie:

- Wysokie zużycie czasu procesora podczas generowania elementów - wczytanie ich jako danych zewnętrznych z reguły zajmuje o wiele mniej czasu.
- Brak bezpośredniej kontroli nad zawartością gry - wiele zasobów łatwiej i szybciej można stworzyć w specjalistycznych programach zewnętrznych, produkując je samodzielnie posiada się bezpośrednią kontrolę nad jakością wytworzonego zasobu.
- Konieczność skrupulatnego doboru parametrów - nieprawidłowo dobrane parametry i brak reguł sprawdzających wygenerowane losowo elementy może prowadzić do mało wiarygodnych wyników generatora (np. w dużym podziemnym labiryncie dwie jedyne drabinki prowadzące do wyjścia znajdują się metr od siebie).
- Wąskie gardło aplikacji dla cache instrukcji - cache danych jest nienalezycie wykorzystywany i aplikacja nie działa optymalnie w sposobie równoważenia przydziału czasu procesora.

Komponenty generowane proceduralnie są bardzo dobrym sposobem na zapewnienie grom długowieczności poprzez losowe generowanie plansz i przedmiotów oraz urozmaicanie elementów rozgrywki. Jednocześnie produkcje nie powinny opierać się całkowicie na proceduralności, chyba, że takie jest główne założenie projektu. Należy pamiętać, że możliwość algorytmicznego tworzenia zawartości jest potężnym narzędziem, jednakże zawsze najważniejsza była, jest i będzie ogólna mechanika gry, do której to sposób tworzenia zasobów należy dostosować.