Wave function collapse

Proceduralne generowanie map

Koncepcja proceduralnego generowania elementów gier komputerowych nie jest niczym nowym. Początkowo był to tylko stosunkowo tani sposób na zwiększenie re-grywalności (ang. replayability, chodzi o to, przez jaki czas gracz powraca do gry po jej ukończeniu), ale rozwiązanie to okazało się z czasem tak dużym źródłem inspiracji, że niektóre współczesne produkcje wręcz bazują na proceduralnym generowaniu świata. Poznamy dziś algorytm, który pomaga generować losowe obrazy, zachowujące jednak pewien zbiór ograniczeń.

ALE O CO CHODZI?

Jednym z bardzo wciągających tytułów bazujących na proceduralnie generowanym świecie jest platformowa, utrzymana w pikselowym stylu retro gra „Noita" fińskiego studia Nolla Games. Oprócz tego, że każdy piksel świata podlega fizycznej symulacji (ma przyporządkowany konkretny materiał, może spaść, spłonąć, zamarznąć, rozpuścić się itp.), mapa świata generowana jest na nowo przed każdą rozgrywką.

Twórcy gry podeszli do tematu dosyć rozsądnie, pozostawiając niektóre aspekty rozgrywki niezmienne: wszystkie biomy znajdują się zawsze na swoich miejscach, mają stały rozmiar, połączenia pomiędzy sobą oraz niektóre cechy (na przykład występujących w nich przeciwników). Sprawia to, że rozgrywka jest na tyle przewidywalna, by gracz mógł korzystać z doświadczenia nabytego podczas poprzednich prób. Z kolei wewnętrzny układ biomów jest za każdym razem inny – generowany na nowo przed każdą grą, dzięki czemu każda rozgrywka jest inna, więc gra szybko się nie nudzi.

Podobnie dzieje się w „Terrarii", grze zbliżonej tematyką do „Noity", wzbogaconej o elementy rzemieślnictwa i budowania świata. Również i tutaj świat generowany jest proceduralnie.

Bodaj najbardziej spektakularnym przykładem jest chyba gra „No Man's Sky", w której proceduralnie generowany jest nie tyle świat, co właściwie wszechświat: systemy planetarne wraz z planetami, ukształtowaniem ich terenu, fauną oraz florą.

Rysunek 1. Gra „Noita" fińskiego studia Nolla Games

NASZE DNI SĄ POLICZONE – O NIEPISANIU NOWYCH BIBLIOTEK PRZETWARZANIA CZASU | Z ARCHIWUM CVE

INDEX: 285358

www.programistamag.pl

Magazyn programistów i liderów zespołów IT

programista

2/2023 (107)

Cena 28,90 zł (w tym VAT 8%)

WAVE FUNCTION COLLAPSE
PROCEDURALNE GENEROWANIE MAP

Nowy numer juz w empikach

ISSN 2084-9400

Źródło grafiki: Adobe Stock

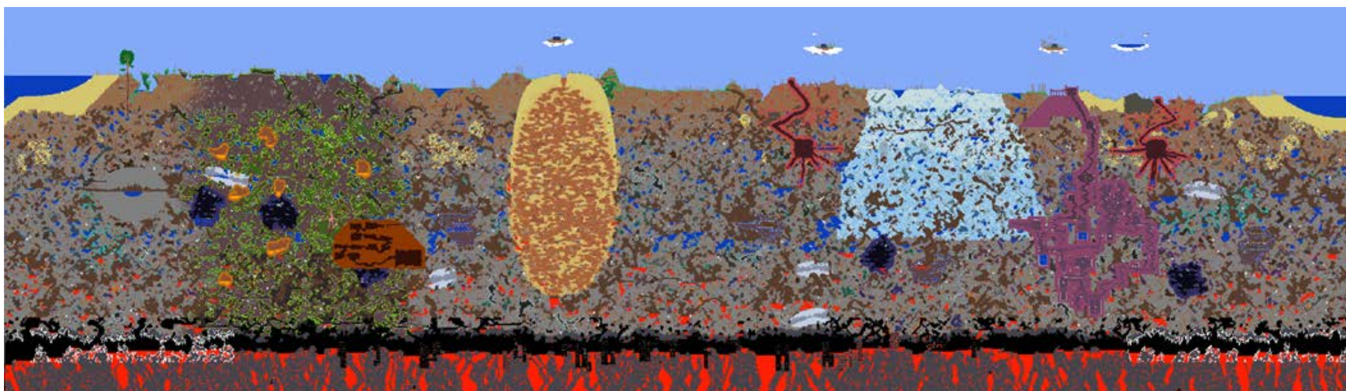
Testy jednostkowe w Go

Projektowanie interfejsu bibliotek .NET

Wzorce strukturalne i behawioralne

Enumeracje w PHP 8.1

Sprzętowy akcelerator krypto kontra procesor M33



Rysunek 2. „Terraria" – przykładowa mapa

Rysunek 3. „No Man's Sky" – proceduralnie wygenerowana planeta

Proceduralne generowanie zawartości nie należy do najłatwiejszych zadań. Wynika to z faktu, iż w wielu przypadkach wygenerowana zawartość musi podlegać pewnym ograniczeniom. Dla przykładu, w „Noita" konieczne jest, by wszystkie korytarze każdego biomu były w miarę możliwości ze sobą połączone. Gracz ma wprawdzie możliwość przebijać się przez miękkie i twarde struktury świata, ale konieczne jest do tego posiadanie odpowiednich narzędzi, które – szczególnie na początku gry – są dosyć trudno osiągalne.

Rozwiązania tego ciekawego problemu podjął się w 2016 roku Maxim Gumin, który zaprojektował algorytm o nazwie Wave function collapse. Wpadł on na pomysł, by mapy gier generować nie losowo, ale na bazie pewnego dostarczonego algorytmowi obrazu stanowiącego swego rodzaju ziarno. Obraz ten traktujemy poniekąd jako zbiór reguł, które muszą znaleźć się w wynikowej mapie.

Aby jednak dobrze zrozumieć cały algorytm, cofnijmy się najpierw w czasie o dwie dekady…

PLAN LEKCJI

Kiedy chodziłem do liceum, co drugą sobotę przyjeżdżał do nas programista, który prowadził kółko informatyczne. Zapytałem go kiedyś o to, w jaki sposób rozwiązać problem, nad którym myślałem od jakiegoś czasu, a chodziło o projektowanie planu lekcji. Nie podał mi wtedy odpowiedzi, ale okazuje się, że jednym z rozwiązań może być właśnie algorytm Wave function collapse.

Zadanie projektowania planu lekcji nie należy do najłatwiejszych, ponieważ jest zależne od dużej liczby czynników. Załóżmy na początku, że planujemy maksymalnie po 8 godzin dziennie, 5 dni w tygodniu. Plan musimy ułożyć dla czterech lat, w obrębie którego mamy kilka różnych klas, różniących się zwykle wymaganiami (klasy w liceum są profilowane), więc dla każdej z klas istnieją inne wymagania dotyczące liczby godzin przeznaczonych na konkretny przedmiot. Nauczyciel zwykle prowadzi zajęcia tylko dla jednej klasy naraz, więc musimy zadbać o to, by w planie w tym samym czasie nie znalazły się zajęcia dla różnych klas, w różnych miejscach. Jeśli zaś o miejscach mowa, to trzeba wziąć też pod uwagę, że niektóre przedmioty – jak na przykład informatyka lub (w lepiej wyposażonych szkołach) geografia, biologia czy chemia – muszą odbywać się w konkretnych salach. Jeżeli więc nawet mamy do dyspozycji dwóch nauczycieli informatyki, ale jedną salę, to wciąż w jednym momencie może odbywać się tylko jedna lekcja tego przedmiotu. Niektóre przedmioty mogą występować w grupach (z mojego liceum pamiętam podwójne lekcje języka polskiego, języków obcych oraz matematyki), zaś niektóre nie; dodatkowo preferowane jest również, aby niektóre przedmioty zamykały dzień (dla przykładu WF). A gdyby i tego było mało, to musimy również uwzględnić mniej oczywiste, ale równie ważne sytuacje, jak na przykład dostępność nauczycieli w konkretnych dniach i godzinach (nauczyciel może prowadzić w danym czasie kółko zainteresowań, pracować na część etatu albo na przykład być ograniczonym z przyczyn osobistych).

Zastanawiałem się, czy dla opisanego problemu istnieje rozwiązanie deterministyczne (oczywiście niebędące rozwiązaniem typu *brute force*). Okazuje się jednak, że problem układania planu lekcji jest NP-trudny, co oznacza, że znacznie lepiej poradzą tu sobie algorytmy heurystyczne, czyli takie, w których fragmenty albo główną część obliczeń wykonuje się losowo.

ROZWIĄZANIE

Zobaczmy, jak moglibyśmy podejść do rozwiązania opisanego wcześniej problemu.

Na początku konstruujemy tablicę indeksowaną dniami, godzinami i klasami, zawierającą sloty, w których umieszczać będziemy konkretne przedmioty. Do każdego slotu podwieszamy na początku listę wszystkich możliwych przedmiotów – uwzględniając ewentualne warunki początkowe, o ile takie istnieją.

Teraz wybieramy jeden ze slotów i spośród wszystkich możliwych przedmiotów losujemy jeden, a następnie umieszczamy go w slocie, usuwając wszystkie alternatywy.

Zauważmy, że ustalenie konkretnego przedmiotu w slocie pociągnie za sobą konsekwencje w postaci ograniczenia zbioru możliwych przedmiotów w innych slotach. Skoro zajęcia prowadzi konkretny nauczyciel, to ze slotów indeksowanych tym samym dniem i godziną musimy usunąć wszystkie przedmioty przez niego prowadzone. Podobnie dzieje się też z przedmiotami, które prowadzone są w wybranej przez nas sali. Idąc dalej, jeżeli przedmiot nie może występować godzina po godzinie, musimy go usunąć ze slotu następującego po tym, który właśnie wybraliśmy.

Zauważmy, że zmiany w dostępności przedmiotów dotyczą nie tylko tych, na które bezpośrednio wpłynął nasz ostatni wybór. Rozważmy sytuację, w której umieściliśmy w slocie lekcje języka angielskiego. W moim liceum prowadzone były one dla uczniów z wielu klas – wynikało to z faktu, iż uczniowie mogli wybrać jeden z kilku języków podstawowych i dodatkowych. W takiej sytuacji wybór języka obcego musi pociągnąć za sobą ograniczenie możliwości wszystkich pozostałych klas w tej godzinie do tego samego przedmiotu. Z innego wymagania może wynikać, iż lekcje języków obcych muszą występować parami, godzina po godzinie. W takim przypadku zmianie ulegnie zbiór przedmiotów nie tylko slotu następującego bezpośrednio po wybranym przez nas, ale również slotu po nim następującego w tej i wszystkich innych klasach tego rocznika. To jeszcze nie wszystko; jeżeli z powodu naszego wyboru zajęliśmy jakąś współdzieloną salę, to zmodyfikować musimy również sloty innych roczników w tych dwóch godzinach, usuwając z nich przedmioty, które w tej sali muszą być prowadzone.

Skoro wstawienie do slotu języka obcego powoduje tak wiele zależności, skąd wiadomo, że było to w ogóle możliwe? Odpowiedź na to pytanie jest prosta: jeżeli wśród przedmiotów możliwych do rozlosowania w wybranym przez nas slocie znajdował się język obcy, oznacza to, iż do tej pory nie pojawiła się żadna decyzja zatwierdzenia przedmiotu w slocie, której konsekwencją byłoby usunięcie języka obcego z tej listy.

Gdy skończymy aplikować do slotów wszystkie konsekwencje zatwierdzonego przez nas przedmiotu, możemy przystąpić do pracy nad kolejnymi slotami w planie.

Tylko którymi?

Teoretycznie moglibyśmy znów wylosować slot i próbować w ten sposób zatwierdzać kolejne przedmioty. Twórca algorytmu Wave function collapse Collapse zauważył jednak pewną ciekawą prawidłowość: ludzie, którzy rozwiązują tego typu złożone problemy manualnie, mają tendencję do skupiania się na obszarach, w których decyzje są najłatwiejsze do podjęcia (czyli lista możliwości jest najkrótsza). Tak realizowany algorytm okazuje się mieć większą szansę powodzenia niż gdybyśmy robili to zupełnie losowo.

Co jednak stanie się, gdy podczas aplikowania konsekwencji zatwierdzenia przedmiotu w slocie okaże się, że lista przedmiotów możliwych do umieszczenia w jednym z nich zmalała do zera?

Mamy wtedy trzy opcje.

» Kontynuować działanie algorytmu (jest to opcja specyficzna dla układania planu lekcji, gdzie możemy pozwolić sobie na niewykorzystane sloty – „okienka").
» Wykonać backtracking, czyli cofnąć się o jedną lub więcej decyzji, i spróbować umieścić w slotach inne przedmioty w nadziei, że problem się wtedy nie pojawi.
» Wyrzucić częściowe rezultaty do kosza i rozpocząć cały proces od nowa.

Jeśli to możliwe, możemy też oczywiście zluzować nieco niektóre ograniczenia, aby algorytm mógł układać plan bardziej elastycznie.

Spróbujmy teraz uogólnić cały algorytm w kilku krokach.

WAVE FUNCTION COLLAPSE

Na początku musimy jasno określić:

» Jak wyglądają komórki, w których chcemy rozlokować dane.
» Jaki jest zbiór danych, które chcemy rozlokować.
» Jakie są reguły lokowania danych.

Kiedy zakończymy ten etap, możemy przystąpić do rozwiązywania problemu:

» Każdą komórkę inicjujemy listą zawierającą możliwe do wprowadzenia dane – na początku będzie to zbiór wszystkich dostępnych danych.
» Jeżeli istnieją jakieś warunki początkowe, aplikujemy je w tym momencie.
» Następnie powtarzamy następujące kroki – aż do wyczerpania komórek:
  » Wybieramy komórkę, która ma najniższą niezerową entropię (co przekłada się na najmniejszy zbiór dostępnych do wyboru opcji). Na samym początku może to być dowolna, losowo dobrana komórka.
  » Spośród wszystkich opcji znajdujących się na jej liście losujemy jedną pozycję. Umieszczamy ją w komórce i czyścimy listę dostępnych opcji (entropia spada wtedy do zera).
  » Przeglądamy wszystkie komórki, na które mogliśmy wpłynąć podjętą przez nas decyzją i aplikujemy jej konsekwencje poprzez ograniczenie ich list możliwych do wprowadzenia danych.
  » Aplikację konsekwencji powtarzamy rekurencyjnie, czyli przeglądamy komórki, na które wpłynęła zmiana możliwych przedmiotów w tych komórkach, na które wpłynęła z kolei nasza decyzja – i tak dalej, aż nie pozostaną nam już komórki do zaktualizowania.
  » Jeżeli w procesie aktualizacji list dostępnych do wprowadzenia elementów wyczyścimy którąś listę do zera, oznacza to, że w tamto miejsce nie da się wprowadzić żadnego elementu. Mamy sprzeczność – podejmujemy decyzję, co zrobić dalej (zwykle uruchamiamy proces od nowa).

Jeżeli w wyniku powtarzania kolejnych kroków uda nam się podjąć decyzję dla każdej komórki, algorytm kończy się sukcesem.

SKĄD NAZWA?

Z początku podchodziłem do opisanego powyżej algorytmu jak pies do jeża, bo nazwa kojarzyła mi się z jakąś ciężkokalibrową matema-

tyką. Tymczasem nie tylko ze świecą szukać w nim kolapsu funkcji falowej, funkcji falowych, ale nawet funkcji jako takich. Nic dziwnego, bo terminologia pochodzi „tylko" z fizyki kwantowej. Jednak spokojnie: znajomość tej dziedziny nauki zupełnie nie jest tu potrzebna; autor zaczerpnął z niej tylko inspirację do całego algorytmu.

Listy możliwych wartości wszystkich komórek zawierają na początku wiele elementów – czego analogią jest superpozycja, czyli sytuacja, w której funkcja falowa znajduje się w wielu stanach jednocześnie. Przyporządkowanie komórce konkretnej wartości jest z kolei podobne do sytuacji, w której dokonujemy obserwacji, ustalając konkretną postać funkcji – jest to właśnie wspomniany w nazwie algorytmu kolaps funkcji falowej. To właśnie z tego powodu w wielu implementacjach algorytmu Wave function collapse funkcja lub metoda wybierająca konkretną opcję dla komórki nazywana jest `Observe`.

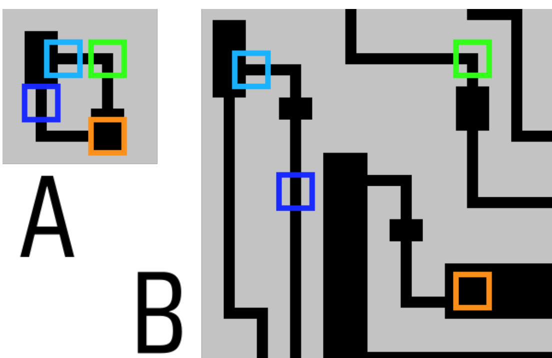
Wiemy już, jak ogólnie działa algorytm. Aby go jednak zaaplikować do generowania obrazów, czeka nas jeszcze trochę pracy.

GENEROWANIE OBRAZÓW

Kompletny kod całego algorytmu, napisany przez jego autora, implementujący dwie główne funkcjonalności (o tym później) dostępny jest na licencji MIT w serwisie GitHub (link na końcu artykułu). Jeżeli ktoś chciałby jednak go obejrzeć, muszę uprzedzić (z całym szacunkiem do autora), że… powiedzmy delikatnie, nie pasuje za bardzo do standardów współczesnego C#. Mówimy tu o jednoliterowych zmiennych, trzykrotnie zagnieżdżonych tablicach, bardzo długich metodach, korzystaniu ze wskaźników w blokach unsafe i tego typu fajerwerkach. Dlatego też – do celów edukacyjnych – napisałem własną implementację algorytmu, która jest wprawdzie znacznie mniej wydajna od oryginału, ale za to o niebo łatwiejsza do zrozumienia (link również na końcu artykułu).

Naszym celem jest wygenerowanie obrazu, który jest „podobny" do dostarczonego algorytmowi obrazu-ziarna (ang. *seed*). Aby jednak móc zaimplementować algorytm, wyrażenie „podobny" musimy oczywiście trochę doprecyzować.

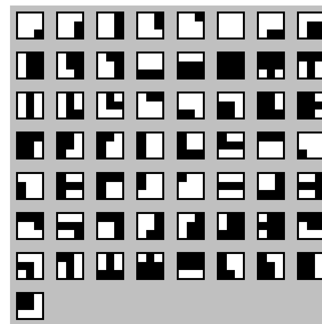
Zdefiniujmy więc kwadratowe okno o stałym rozmiarze, na przykład dwóch na dwa lub trzech na trzy piksele. Powiemy, że obraz B jest podobny do obrazu A, jeżeli po przyłożeniu okna do dowolnego spójnego zbioru pikseli na obrazie B odnajdziemy gdzieś dokładnie taki sam zbiór na obrazie A (Rysunek 4 – przykładowe, pasujące fragmenty zaznaczono kolorowymi ramkami).



Rysunek 4. Obraz B podobny do obrazu A dla okna 3x3 piksele

Jeżeli chcemy otrzymać ciekawsze wyniki (jak również zmniejszyć prawdopodobieństwo sprzeczności), możemy nieco rozluźnić regułę podobieństwa i powiedzieć, że obrazy są podobne, gdy dany zbiór z obrazu B odnajdziemy na obrazie A również w postaci obróconej o 90, 180 lub 270 stopni, albo też odbity lustrzanie względem poziomej lub pionowej osi symetrii.

Skoro naszym warunkiem podobieństwa jest skorzystanie z kafelków wyciętych oknem z obrazka-ziarna, jasnym staje się, że to właśnie te kafelki będą stanowić zbiór danych, który będziemy rozmieszczać w slotach. W przypadku obrazu-ziarna zaprezentowanego na Rysunku 4 mamy do czynienia z bitmapą o rozmiarze 14x14 pikseli, co daje (14-2)*(14-2)=144 obrazki. W praktyce jednak wiele z nich będzie się powielało, a takie same wyniki można scalić, więc liczba ta skurczy się do zaledwie 57 kafelków (Rysunek 5).



Rysunek 5. Wszystkie unikalne kafelki 3x3 obecne na obrazie-ziarnie z Rysunku 4

Listing 1. Metody biorące udział w ekstrakcji kafelków z obrazu-ziarna

```
private static List<Sample> GenerateBaselineSamples(
    Color[,] pixels)
{
    ConsoleHelper.Info("Generating baseline samples");

    int maxX = Constants.SOURCE_WRAPPED_AROUND ?
        pixels.GetWidth() :
        pixels.GetWidth() - Constants.SAMPLE_SIZE + 1;
    int maxY = Constants.SOURCE_WRAPPED_AROUND ?
        pixels.GetHeight() :
        pixels.GetHeight() - Constants.SAMPLE_SIZE + 1;

    List<Sample> result = new List<Sample>();

    for (int x = 0; x < maxX; x++)
        for (int y = 0; y < maxY; y++)
        {
            List<Sample> currentSamples = new()
            {
                new Sample(pixels.ExtractSample(x,
                    y,
                    Constants.SAMPLE_SIZE))
            };

            var currentData = currentSamples[i].Data;

            if (Constants.MIRROR)
            {
                int currentSampleCount =
                    currentSamples.Count;
                for (int i = 0; i < currentSampleCount; i++)
                {
                    currentSamples.Add(new Sample(
                        currentData.MirrorX()));
                    currentSamples.Add(new Sample(
                        currentData.MirrorY()));
                }
            }

            if (Constants.ROTATE)
            {
                int currentSampleCount = currentSamples.Count;

                for (int i = 0; i < currentSampleCount; i++)
```