Wydział Matematyki i Informatyki Uniwersytetu Łódzkiego

Programowanie Gier Komputerowych

Przemysław Szymański

Październik 2010

Praca napisana pod przewodnictwem
dr Piotra Fulmańskiego

Spis Treści

1. Wstęp 5
2. Programowanie ogólne 8
2.1. Wzorce Projektowe 8
2.1.1. Singleton 8
2.1.2. Automatyczny singleton 11
2.1.3. Fabryka 13
2.1.4. Inteligentne Wskaźniki 15
2.2. Zarządzanie pamięcią 21
2.2.1. Manager obiektów o stałym rozmiarze 23
2.2.2. Model FreeList 28
2.2.3. Ramkowy manager pamięci 32
2.3. Zarządzanie zasobami 38
2.4. Wirtualny system plików 61
2.5. Optymalizacje językowe 64
2.5.1 Klasy 64
2.5.1. Operacje na liczbach całkowitych 68
2.5.2. Operacje na liczbach zmiennoprzecinkowych 71
3. Sztuczna inteligencja 74
3.1. Automat o skończonej liczbie stanów 74
3.2. Wyszukiwanie drogi 79
4. Programowanie grafiki 87
4.1. Sprzętowy potok graficzny 87
4.1.1. Programowalna jednostka wierzchołków 88
4.1.2. Programowalna jednostka fragmentów 89
4.2. Podział geometrii 91
4.3. Oświetlenie 95
4.4. Mapowanie nierówności 101
4.4.1. Bump Mapping 101
4.4.2. Parallax Mapping 106
4.5. Efekty przetwarzania obrazów (post-process) 110
4.5.1. Rozmycie 110
4.5.3. Efekt Bloom 116

4.5.4. Szeroki zakres dynamiczny (HDR – High Dynamic Range) 118
4.6. Renderowanie środowiska 124
4.6.1. Teren 124
4.6.2. Trawa 130
4.6.3. Woda 134
4.7. Animacja 144
4.7.1. Animacja za pomocą ujęć kluczowych 144
4.7.2. Animacja za pomocą kości 145
5. Podsumowanie 147
6. Słowniczek 151
7. Bibliografia 152

1. Wstęp

Gra komputerowa jest specyficznym przypadkiem programu komputerowego. Różnica między nimi polega na tym, że w przypadku programu, to użytkownik musi się do niego dostosować, natomiast w przypadku gry to gra musi dostosować się do użytkownika. Program komputerowy posiada zestaw opcji, których użytkownik może, często w ustalonej kolejności, używać, aby osiągnąć zamierzony efekt. Gra ustala z góry pewne reguły (mogą być zmieniane) w ramach, których użytkownik może działać. Samo „granie" polega na interakcji użytkownika z obiektami gry (poprzez odbijanie piłeczki, przestawienie figury szachowej, rozmowę z postacią kierowaną przez komputer, itp.), aby osiągnąć ustalony cel. Dzisiejsze gry oferują olbrzymie światy, po których gracz może się poruszać oraz tysiące obiektów, na które można oddziaływać. Taka złożoność powoduje, że ten typ programu stawia przed programistami olbrzymie wyzwanie. Komputery (PC, konsole, telefony komórkowe) mają ograniczoną ilość mocy obliczeniowej i pamięci. Wymusza to konieczność pójścia na kompromis pomiędzy jakością a szybkością. Ponieważ gry mają na celu dostarczenie użytkownikom rozrywki na najwyższym poziomie, oraz dochodów finansowych twórcom, programiści starają się uzyskać jak największą wydajność przy jednoczesnej maksymalizacji jakości. Tradycyjny program może pozwolić sobie na wolniejsze działanie podczas pewnych operacji, natomiast gra musi zawsze działać szybko. Zmusza to programistów do ciągłego optymalizowania kodu, algorytmów oraz obliczeń. Wiele firm posiada swoje zespoły naukowe, które opracowują technologie używane w kolejnych grach. Dzięki tym badaniom społeczność otrzymuje zestaw bardzo pomocnych narzędzi, lub nawet gotowych rozwiązań przyspieszających i ułatwiających proces tworzenia gier oraz innych programów komputerowych. Dzielenie się pomysłami jest bardzo ważne ponieważ w programowaniu gier należy posiadać szeroką wiedzę na różne tematy związane z informatyką: znajomość architektury sprzętu, kompilatora, języków programowania i bibliotek pomocniczych, języków skryptowych, sztucznej inteligencji, grafiki dwu i trójwymiarowej, matematyki i fizyki, dźwięku, sieci komputerowych. W obecnych czasach szybkość procesorów nie wzrasta w tak szybkim tempie jak kilka lat temu. Teraz moc obliczeniowa zwiększana jest za pomocą ilości dostępnych procesorów (rdzeni). Większość gier komputerowych rozkłada obliczenia na kilka wątków (fizyka, sztuczna inteligencja, itp.), dzięki czemu możliwe jest znaczne zwiększenie wydajności. Problem pojawia się w rozłożeniu tych obliczeń na wątki i ich synchronizacji. Programowanie wielowątkowe jest bardzo trudnym zagadnieniem i niewielu programistów jest w stanie poradzić sobie dobrze z tym problemem. Teraz, gdy mamy dostępne procesory nawet 6-rdzeniowe programy aż proszą się o wykorzystanie tej mocy[1]. Kolejnym problemem, z którym programiści gier muszą się zmierzyć to różne platformy sprzętowe. Przez ostatnie lata konsole firm takich jak: Microsoft, Sony czy Nintendo zyskały na popularności, na tyle dużej, że każda większa firma nie może tego zignorować i musi wydawać gry na jak największą liczbę platform. Pisanie kodu wieloplatformowego (ponieważ nie ma sensu

[1] Konsola Sony PlayStation 3 została wypuszczona na rynek w listopadzie 2006 roku i jest wyposażona w procesor Cell Broadband Engine, który posiada 6 rdzeni.

pisać tego samego programu kilka razy) jest kolejnym wyzwaniem stawianym przed programistami. Różnice znajdują się nie tylko w systemie operacyjnym ale i w samej architekturze. Prostym przykładem może być konsola Xbox, która obsługuje DirectX oraz konsola PlayStation, na której jest OpenGL. Jeżeli chcemy wydać grę na obie te konsole, wówczas musimy napisać kod, który potrafi wyświetlać grafikę przy użyciu obu bibliotek. Utworzenie podsystemów rozwiązujących takie problemy jest przeważnie bardzo trudne. Dodatkowo producenci konsol sami muszą przetestować daną grę i ją zatwierdzić. Aplikacja musi spełniać pewne wymagania (na przykład nigdy nie może działać wolniej niż ok. 30 klatek na sekundę. Dlatego gry na konsolach zawsze działają płynnie) inaczej nie zostanie wydana na tą platformę.

Powstaje pytanie: jak napisać grę komputerową? Otóż, nie ma jednego, dobrego przepisu na to. W każdej grze można spotkać inne podejście do tego samego problemu. Nie ma idealnych rozwiązań, zamiast tego są pewne kompromisy (przeważnie jakość-wydajność), na które trzeba pójść w danej sytuacji. Czy skorzystamy z algorytmu A czy B w danym zagadnieniu, w dużej mierze zależy od typu gry. Jeżeli robimy strzelarkę (FPS) dla wielu graczy, wówczas sztuczna inteligencja będzie ograniczona do minimum, natomiast w grze cRPG postacie komputerowe powinny się zachowywać bardzo naturalnie, natomiast element multiplayer może ograniczać się do minimum.

W swojej pracy przedstawiam kilka problemów, z którymi prędzej czy później zmierzy się każdy, kto chce tworzyć gry, oraz propozycje na ich rozwiązanie, które zostały już wykorzystane w niejednej profesjonalnej produkcji. Wybrałem zagadnienia, które stanowią fundamenty niemal każdej nowszej gry.

- Wzorce projektowe stanowią ułatwienie w programowaniu. Są przydatne w tworzeniu każdej większej aplikacji.
- Dobre zarządzanie pamięcią może skutecznie zwiększyć wydajność
- Zarządzanie zasobami jest mechanizmem, bez którego ciężko się obejść. Dzięki przemyślanemu Managerowi zasobów można łatwo manipulować teksturami, dźwiękami, siatkami geometrii czy skryptami nie martwiąc się przy każdym użyciu o sposoby tworzenia i niszczenia zasobów czy używania konkretnych struktur danych używanych do przechowywania obiektów.
- Automaty stanów przydatne są nie tylko w sztucznej inteligencji, ale mogą też sterować przebiegiem całej aplikacji. Stosując wskaźniki na funkcje można przyspieszyć ich działanie i uprościć cały zapis, jednocześnie rozkładając kod na drobniejsze fragmenty, które potem łatwiej edytować.
- Wyszukiwanie drogi jest chyba najczęstszym problemem w grach komputerowych. Popularny algorytm A*, choć już dość stary, cały czas jest używany i jednocześnie ulepszany. W grach trójwymiarowych użycie go nie jest trywialne, ponieważ świat nie składa się z mapy kwadratów lub sześcianów.
- O sukcesie gry może zadecydować jakość jej grafiki. Obecna technologia pozwala na uzyskanie kinowej jakości grafiki czasu rzeczywistego. Warto wykorzystać ten fakt i zaimplementować algorytmy wymyślane już w latach 60-tych ubiegłego wieku, które teraz mogą być stosowane w grach. Czy była by scena trójwymiarowa bez oświetlenia? Prosty model oświetlenia Phong'a pozwala na uzyskanie całkiem ciekawych efektów przy niewielkim spadku wydajności. Już od kilku lat karty graficzne pozwalają na tworzenie fotorealistycznych obrazów stosując technikę HDR, dzięki której scena wygląda jeszcze bardziej przekonująco.
- Dobrze wykonana woda, drzewa czy trawa może dodać bardzo dużo realizmu całej scenie. Efekty wizualne mogą wprowadzić gracza w zachwyt, choć wbrew pozorom składają się z kilku kwadratów i tekstury.

Zakładam, że czytelnik posiada podstawową wiedzę na temat programowania w języku C/C++, HLSL/Cg oraz grafiki komputerowej. Słowniczek pojęć znajduje się tuż przed bibliografią. Uważam, że tego typu praca nie może się obejść bez fragmentów kodu, aby lepiej zobrazować opisywane zagadnienie. Używam uproszczonej notacji węgierskiej w celu łatwiejszego czytania i zrozumienia. Przedstawiony w pracy kod został przetestowany w kompilatorze Visual Studio 9.0.

Przedrostek	Znaczenie
_ (podkreślenie)	Argument funkcji
m_	Pole klasy
a	Tablica
str	Łańcuch znaków, obiekt typu std::string
i	Liczba całkowita typu int
s	Liczba całkowita typu short int
b	Typ logiczny bool
ch	Typ jednobajtowy char
l	Liczba całkowita typu long int
t	Zmienna typu zdefiniowanego z klasie wzorcowej
e	Zmienna typu wyliczeniowego enum
f	Zmienna rzeczywista typu float
d	Zmienna rzeczywista typu double
p	wskaźnik
u	Liczba dodatnia typu unsigned
ui	unsigned int
us	unsigned short
ul	unsigned long
C	Klasa
S	Struktura
f2	Zmienna typu float2 (język HLSL/Cg)
f3	Zmienna typu float3 (język HLSL/Cg)
f4	Zmienna typu float4 (język HLSL/Cg)

2. Programowanie ogólne

2.1. Wzorce Projektowe

W definicji wzorzec projektowy jest to pewne uniwersalne narzędzie często pojawiających się problemów projektowych. W programowaniu gier takie problemy również się pojawiają. Ponieważ często pewna część kodu gry jest używana ponownie w innych projektach, musi być dobrze zorganizowany.

2.1.1. Singleton

Singleton to obiekt, który posiada tylko jedną instancję. W działaniu przypomina zmienną globalną, z tym wyjątkiem, że jest to obiekt i nie można utworzyć drugiego obiektu o tym samym typie. W programowaniu gier często korzysta się z singletonów wtedy, gdy chcemy opisać rzeczy, które nie występują w ilości większej niż jedna. Przykładem może być klawiatura, monitor lub manager textur.

W dalszej części tego podrozdziału skupię się na przykładzie managera textur (`CTextureManager`). Dla uproszczenia przedstawiony przeze mnie manager textur ma tylko kilka funkcji: ładowanie textury (`LoadTexture`), usuwanie textury (`DeleteTexture`) oraz pobieranie textury (`GetTexture`). Ten krótki opis mówi sam za siebie, dlaczego warto taką klasę zrobić singletonem. Więcej niż jedna instancja takiej klasy nie będzie potrzebna, a inne podsystemy z dowolnego miejsca mogą korzystać z managera textur. Oczywiście wiąże to ze sobą pewne implikacje. Istnieje bowiem duży problem podczas refactoringu kodu, kiedy używamy singletonów wewnątrz funkcji, których kod chcemy zmieniać. Dużo lepszym pomysłem jest przekazywanie przez referencję lub wskaźnik singletonu. Minusem tego rozwiązania jest to, że i tak w którymś momencie kodu, w jakiejś metodzie należy pobrać referencję lub wskaźnik do singletonu, a przekazywanie jej później jako parametr do funkcji lub metod też może być uciążliwe.

Na pierwszy rzut oka może wydać się oczywiste wykorzystanie statycznych metod w klasie CTextureManager.

```
class CTextureManager
{
      public:

            static int        LoadTexture(const string& textureFileName)
                              { /*...*/ }
            static CTexture*  GetTexture(const string& textureName)
                              { /*...*/ }
            static void       DeleteTexture(const string& textureName)
                              { /*...*/ }

            //...
};
```

Głównym problemem tego rozwiązania jest brak możliwości tworzenia składowych obiektu na żądanie (brak inicjalizacji i niszczenia). Drugim problemem jest to, że funkcje statyczne nie mogą być wirtualne czyli jeśli byśmy chcieli w jakikolwiek sposób zmodyfikować klasę, nie jest to możliwe.

Lepsze rozwiązanie.

```
class CTextureManager
{
      public:

            static CTextureManager& GetSingleton()
            {
                  static CTextureManager Singleton;
                  return Singleton;
            }

            int         LoadTexture(const string& textureFileName)
                        { /*...*/ }
            CTexture*   GetTexture(const string& textureName)
                        { /*...*/ }
            void        DeleteTexture(const string& textureName)
                        { /*...*/ }

            //...
};
```

W ten sposób otrzymaliśmy tworzenie singletonu tylko na żądanie. Singleton tworzy się przy pierwszym wywołaniu metody `GetSingleton()`. Do powyższego kodu należy dodać jeszcze prywatny konstruktor, prywatny konstruktor kopiujący, prywatny destruktor oraz prywatny operator przypisania. Kompilator C++ sam tworzy domyślne, publiczne, konstruktory, destruktor oraz operator przypisania. Skoro potrzeba nam tylko jednej instancji obiektu, należy własnoręcznie te metody dopisać, żeby zablokować kompilator przed ich utworzeniem. Utworzenie prywatnego konstruktora zapewni nas, że nie będzie możliwości utworzenia nowej kopii, natomiast nadpisanie domyślnego konstruktora kopiującego da nam pewność, że nikt nie utworzy kodu podobnego do poniższego.

```
CTextureManager TexMgr( CTextureManager::GetSingleton() );
```

Teraz, w związku z użyciem takiego kodu Microsoft Visual Studio wygeneruje następujący błąd:

error C2248: 'CTextureManager::CTextureManager' : cannot access private member declared in class 'CTextureManager'

Jawny destruktor zabezpiecza przed omyłkowym skasowaniem obiektu. Jeśli nie chcemy, żeby singleton był usuwany (operator delete), można przeciążyć operator delete dla singletonu[2].

Zastanówmy się teraz nad problemami związanymi w powyższym singletonem. Jeżeli często w kodzie używamy metody `GetSingleton()` zarówno w konstruktorach innych klas oraz w metodach można mieć duże problemy z odnalezieniem momentu tworzenia singletonu. Jeżeli singleton klasy CTextureManager korzysta z innych singletonów (np. CFileManager, który zarządza odczytywaniem plików z dysku) nie mamy pewności, czy pozostałe singletony zostały utworzone. Kompilator C++ wykonuje inicjowanie statyczne przed wykonaniem jakiejkolwiek instrukcji programu (zazwyczaj podczas ładowania programu[3]). Język C++ nie określa kolejności inicjalizacji obiektów statycznych, które znajdują się w różnych jednostkach translacji[4]. Kolejnym problemem jest usuwanie singletonów, które następuje dopiero po zamknięciu aplikacji. W programowaniu gier kontrola życia obiektów jest bardzo ważna. Jeden podsystem może się wyłączyć dopiero jak wszystkie inne podsystemy, z których on korzysta zostaną wyłączone. W tym rozwiązaniu nie jest również możliwe wyłączenie jednego podsystemu w trakcie działania gry. Często jest to bardzo ważne podczas testowania aplikacji. Oczywiście pewnym obejściem tych problemów jest zastosowanie dodatkowych metod tworzenia i usuwania.

```
CTextureManager& TexMgr = CTextureManager::GetSingleton();
TexMgr.Create();

//...

TexMgr.Shutdown();
```

Co prawda metoda `Shutdown()` nie usunie singletonu z pamięci, ale może wyczyścić i usunąć obiekty z których ten singleton korzysta.

Skoro tak ważna jest możliwość śledzenia singletonu zastanówmy się nad stworzeniem wskaźnika na niego.

```
class CTextureManager
{
      public:

            static CTextureManager* GetSingleton()
            {
                  if( !m_pSingleton )
                        m_pSingleton = new CTextureManager();

                  return m_pSingleton;
            }

            //...
```

[2] W ten sposób można zablokować możliwość usunięcia każdej klasy. Operator delete powinien być prywatny.
[3] Przeważnie inicjatory statyczne znajdują się w pliku wykonywalnym.
[4] Jednostka translacji to plik źródłowy C++ wraz ze wszystkimi plikami dołączonymi przez preprocesor. Jest to kod widziany przez kompilator podczas kompilacji.

```
	private:

		static CTextureManager* m_pSingleton;
};
```

W tym przypadku singleton tworzy się przy pierwszym użyciu. Dzięki wskaźnikowi możemy śledzić singleton w dowolnym miejscu programu. Nie będzie wycieków pamięci, jeśli klasa singletona zostanie zabezpieczona przed niechcianym usuwaniem. Jest to standardowa postać singletona stosowana w wielu językach. W programowaniu gier też zda egzamin jeśli metody `GetSingleton()` nie będziemy używać w każdej klatce animacji. O ile jedną instrukcję warunkową można pominąć, to w przypadku korzystania z większej ilości singletonów ilość `if`'ów również się zwiększy. Instrukcje warunkowe nie są darmowe, należy ich unikać jeśli jest taka możliwość. W znakomitej ilości przypadków związanych z używaniem singletonów wskaźnik na nie można pobrać w innych miejscach niż w metodzie renderującej. Singleton, z definicji, się nie zmienia, zatem wskaźnik na niego zawsze będzie ten sam, zatem nie ma potrzeby pobierania go w każdej klatce animacji.

2.1.2. Automatyczny singleton

Jedną z odmian singletonów są automatyczne singletony. Automatyczny singleton jest to klasa singletonu, która zawiera jedynie metody do tworzenia, usuwania i pobierania singletonu. Każda klasa, która ma być singletonem powinna dziedziczyć z klasy automatycznego singletonu stając się w ten sposób singletonem. Oto przykładowy kod automatycznego singletona.

```
template<class T>
class CSingleton
{
	public:

		CSingleton()
		{
			assert( !m_pSingleton );
			int iOffset = (int)(T*)1 - (int)(CSingleton<T>*)(T*)1;
			m_pSingleton = (T*)((int)this + iOffset);
		}

		~CSingleton()
		{
			assert( m_pSingleton );
			m_pSingleton = 0;
		}

		static T* GetSingletonPtr()
		{
			assert( m_pSingleton );
			return m_pSingleton;
		}
```

```
            static T& GetSingleton()
            {
                  assert( m_pSingleton );
                  return *m_pSingleton;
            }

      private:

            static T* m_pSingleton;
};
```

Dzięki takiemu rozwiązaniu można łatwo pozbyć się problemów jakie wcześniej zostały opisane oraz bardzo ułatwić sobie tworzenie nowych singletonów. Powyższa implementacja zapewnia, że singleton zostanie utworzony dokładnie jeden raz. Jeśli zostanie utworzony drugi raz, w innym miejscu programu, zostanie zgłoszony `assert`. Jeśli zostanie użyty singleton bez wcześniejszego utworzenia również zostanie zgłoszony `assert`. Ten singleton tworzyć można w dowolny sposób, dzieje się tak dlatego, że w konstruktorze klasy `CSingleton` obliczany jest względny adres pochodnej klasy zapisywany we wskaźniku singletonu. Klasa pochodna nie musi wywodzić się tylko z jednej klasy `CSingleton`. W takim przypadku istnieje szansa, że wskaźnik `this` klasy pochodnej może być inny od wskaźnika `this` klasy `CSingleton`. W związku z tym należy pobrać nieistniejący obiekt znajdujący się w pamięci pod adresem 0x1 oraz zrzutowanie go na oba typy i pobranie różnicy, która będzie odległością między `CSingleton<CTextureManager>` i jego pochodnym typem `CTextureManager`, którego można użyć do obliczenia wskaźnika singletonu.

Ten typ singletonów nie narzuca sposobu tworzenia singletonu. Jeśli korzystamy z własnego managera pamięci nie będzie problemu żeby się nim posłużyć do tworzenia singletonów. Jeśli stosowali byśmy singleton z tworzeniem w metodzie `GetSingleton()` za pomocą `new`, tak jak w poprzednim przykładzie, należało by zastosować jakiś typ allocator'a, żeby mieć możliwość własnoręcznego przydzielania i zwalniania pamięci. Należy zauważyć, że ten typ singletonu narzuca konieczność tworzenia go. Nie ma tutaj tworzenia przy pierwszym użyciu. Nie jest to wadą jeśli chcemy mieć możliwość tworzenia singletonów w konkretnej kolejności oraz usuwania ich w konkretnym miejscu aplikacji.

Przykład zastosowania automatycznego singletonu.

```
class CTextureManager : public CSingleton<CTextureManager>
{
      public:

            int         LoadTexture(const string& textureFileName)
                        { /*...*/ }
            CTexture*   GetTexture(const string& textureName)
                        { /*...*/ }
            void        DeleteTexture(const string& textureName)
                        { /*...*/ }

            //...
};
```

```
//...

CTextureManager* pTexMgr1 = new CTextureManager();
//...
CTextureManager* pTexMgr = CTextureManager::GetSingletonPtr();
//...
CTexture* pTexture = pTexMgr->LoadTexture( "texture1.dds" );
//...
delete pTexMgr;
```

2.1.3. Fabryka

Fabryka jest wzorcem nie tak często używanym w programowaniu gier jak w programowaniu silników oraz framework'ów. Fabryka jest to obiekt, który tworzy inne obiekty najczęściej wywodzące się z jednej klasy bazowej. Fabryka może mieć jedną lub wiele metod do tworzenia obiektów. Przykładem fabryki może być system do tworzenia graficznego interfejsu użytkownika (ang. GUI), gdzie jest klasa do tworzenia komponentów.

```
CButton Button = (CButton)GUI.CreateComponent("button", "Przycisk1");
CPanel Panel = (CPanel)GUI.CreateComponent("panel", "Panel1");
Panel.AddComponent(Button);
```

W powyższym pseudokodzie widzimy obiekt `GUI`, który jest obiektem klasy tworzącej komponenty interfejsu. Metoda `CreateComponent`, tworzy komponent o typie podanym w pierwszym argumencie, drugi argument to nazwa tego komponentu. Zwracany jest obiekt klasy bazowej (np. `CComponent`, z której wywodzi się `CButton` oraz `CPanel`). Klasa `CButton` odwzorowuje przycisk GUI natomiast `CPanel` jest to panel, na którym można umieszczać różne kontrolki. Metoda `AddComponent` przyjmuje w argumencie klasę bazową CComponent. Łatwo zauważyć, że takie rozwiązanie jest dużo prostsze w tworzeniu oraz w późniejszym używaniu. Alternatywą było by stworzenie metod o nazwach: `CreateButton`, `CreatePanel`, tylko, że miały by dokładnie te same typy argumentów i robiły by niemal to samo. W praktyce można stworzyć takie metody prywatne, które będą używane w metodzie `CreateComponent`. Można tak zrobić dla ułatwienia pracy przy tworzeniu GUI.

Dzięki mechanizmom polimorfizmu w prosty sposób można stworzyć obiekty pochodne klasy bazowej. Za pomocą prostych identyfikatorów (w pseudokodzie były to napisy) można rozróżniać typy obiektów. Oto przykładowa implementacja metody `CreateComponent`.

```
enum COMPONENT_TYPE
{
	UNKNOWN = 0,
```

```
	BUTTON,
	PANEL
};

CComponent* CreateComponent(  COMPONENT_TYPE eType,
                              const std::string& strName)
{
	CComponent* pComponent = 0;
	switch( eType )
	{
		case UNKNOWN:
			pComponent = new CComponent( strName );
		break;
		case BUTTON:
			pComponent = new CButton( strName );
		break;
		case PANEL:
			pComponent = new CPanel( strName );
		break;
	}

	return pComponent;
}
```

Funkcja wygląda bardzo prosto, za pomocą typu wyliczeniowego wybiera typ obiektu, który następnie tworzy[5]. Jeśli jako typ została podana wartość, której nie ma wśród wartości typu wyliczeniowego wtedy zwracane jest 0 (`NULL`). Przyjrzyjmy się teraz wadom tego rozwiązania. Po pierwsze zastosowanie rozgałęzienia[6] na podstawie identyfikatora. Trudno taką fabrykę rozszerzyć, brak skalowalności. Aby dodać kolejny komponent należy stworzyć nową klasę dziedziczącą z klasy `CComponent` i zaimplementować jej metody wirtualne (tego akurat nie da się uniknąć). Następnie należy dodać nowy typ wyliczeniowy i utworzyć obiekt w metodzie `CreateComponent`. Jest to niemożliwe do rozwiązania jeśli korzystamy z zewnętrznej biblioteki, którą chcieli byśmy rozszerzyć o własne komponenty a nie mamy do niej kodu źródłowego.

Przede wszystkim należy wyeliminować instrukcję `switch`. Dobrym pomysłem osiągnięcia tego celu są wskaźniki na funkcje. Sygnatura może wyglądać następująco.

```
CComponent* CreateConcreteComponent(const std::string& strName);
```

To rozwiązanie jest dobre szczególnie w przypadku, kiedy każdy obiekt będziemy tworzyć w oparciu o tą samą listę argumentów. W tym uproszczonym przypadku potrzeba nam jedynie nazwy. Fabryka będzie przechowywać wskaźniki na funkcje tworzące obiekty. Tutaj należy zastanowić się nad wyborem sposobu przechowywania tych wskaźników. Po pierwsze trzeba wybrać typ identyfikatorów. Jeśli mają to być liczby całkowite można zastosować `std::vector`. Zaletą tego pojemnika jest szybkość dostępu do konkretnego

[5] Każdy obiekt może być tworzony w inny sposób, to jest jedna z zalet fabryk, programista nie musi nic wiedzieć na temat tworzenia danego obiektu. Dla uproszczenia w przykładzie wszystkie obiekty tworzone są tak samo.

[6] W tym przypadku instrukcja `switch`.

pola oraz łatwość dodawania nowych. Zauważmy, że nie potrzebujemy pojemnika, który oferował by sprawne usuwanie, więc `std::vector` wydaje się być pod tym względem dobrym rozwiązaniem. Zauważmy też, że ten pojemnik jest dobry tylko jeśli chcemy mieć identyfikatory w spójnym przedziale co nie jest dobrym pomysłem jeśli fabrykę mogli by rozszerzać inni programiści. Prawdopodobieństwo powtórzenia się identyfikatora w tym przypadku jest dość duże. Lepszym pomysłem będzie zastosowanie kolekcji asocjacyjnej (np. `std::map`). Eliminujemy spójny przedział identyfikatorów oraz otrzymujemy możliwość posiadania dowolnego typu identyfikatorów[7].

2.1.4. Inteligentne Wskaźniki

W opisie singletonu oraz fabryki zostały użyte przykłady managera textur oraz managera GUI. Pierwszy z nich m.in. ładował textury z pliku do pamięci i zwracał na nie wskaźniki natomiast drugi tworzył różnego rodzaju kontrolki graficznego interfejsu i też zwracał na nie wskaźniki. W obu przypadkach rezerwowanie pamięci mogło się odbywać poza wiedzą użytkownika – programisty, usuwanie także mogło być wykonywane automatycznie, podczas niszczenia managera textur lub fabryki. Klasy reprezentujące teksturę lub komponent GUI mogły mieć przeciążone operatory delete, aby niepowołany programista nie mógł ich dowolnie usuwać. W związku z takim podejściem warto zastanowić się nad ulepszeniem tego rozwiązania. Większość języków wysokiego poziomu jak np. Java czy C# nie udostępniają programistom dostępu do wskaźników[8] . Zamiast tego operujemy referencjami, które są zarządzane przez garbage collector. W C++ można zbliżyć się nieco do tego rozwiązania używając inteligentnych wskaźników. Inteligentny wskaźnik jest to klasa, która naśladuje składniowo i niekiedy semantycznie zwykły wskaźnik. Klasa taka jest przeważnie parametryzowana typem wskaźnika, dlatego że implementacja klasy inteligentnego wskaźnika dla obiektów różnych typów nie różni się. Załóżmy, że mamy klasę `CSmartPointer<T>`, gdzie `T` jest typem zawieranego wskaźnika przez tą klasę. Klasa dodatkowo zawiera `operator->` oraz `operator*` dzięki czemu składnia będzie podobna do normalnego wskaźnika.

```
template<class T>
class CSmartPointer
{
      public:

            explicit CSmartPointer(T* _pPtr) : m_pPtr( _pPtr )
            { /*...*/ }
            ~CSmartPointer() {}

            CSmartPointer& operator=(const CSmartPointer& _pPtr)
            { /*...*/ }

            T& operator*() const
            {
                  /*...*/
                  return *m_pPtr;
            }
```

[7] Wystarczy, że identyfikator będzie miał przeciążony `operator==`, `operator<` oraz konstruktor kopiujący.
[8] W rzeczywistości C# daje możliwość operowania na wskaźnikach jeśli kod będzie oznaczony jako unsafe.

```
            T* operator->() const
            {
                  /*...*/
                  return m_pPtr;
            }

      private:

            T*    m_pPtr;
};
```

Załóżmy, że mamy klasę `CTest`, która ma metodę `Foo()`. Chcemy dynamicznie utworzyć obiekt tej klasy w pamięci i przechowywać do niego wskaźnik.

```
CSmartPointer< CTest > pTest( new CTest() );

pTest->Foo();
(*pTest).Foo();
```

Używanie inteligentnego wskaźnika nie różni się niczym od zwykłego, jedynie tworzenie go jest dość nietypowe.

Łatwo zauważyć, że inteligentne wskaźniki, w przeciwieństwie do normalnych, mają semantykę wartości[9]. Wskaźnik używany do przechodzenia po elementach tablicy również ma semantykę wartości. Ustawiamy wartość wskaźnika na początek i przechodzimy kolejno dalej aż do końca. Niestety inaczej jest dla wskaźników, które zawierają wartości z dynamicznie przydzielonej pamięci (`new`). Te wskaźniki nie mają semantyki wartości.

```
CTest* pTest = new CTest();
```

Jeśli teraz do `pTest` przypiszemy 0, stracimy wskaźnik na zarezerwowaną pamięć czyli utracimy możliwość zwolnienia jej co zaowocuje wyciekiem pamięci. Dodatkowo jeśli skopiujemy wartość `pTest` do innego wskaźnika wskazywany obiekt nie zostanie automatycznie skopiowany. Otrzymamy dwa wskaźniki, które wskazują na ten sam obiekt. Tutaj dodatkowo trzeba uważać żeby nie usunąć obu wskaźników, gdyż grozi to załamaniem się aplikacji. Inteligentne wskaźniki rozwiązują ten problem. Ponadto mogą zarządzać własnością wskaźników. Można zaimplementować inteligentny wskaźnik tak by wiedział kiedy własność się zmienia, natomiast destruktor może mieć różne metody zwalniania pamięci.

Zarządzanie własnością może odbywać się na wiele sposobów. Podczas kopiowania wskaźnik źródłowy może zostać wyzerowany (`NULL`), a wskaźnik docelowy zostaje jego

[9] Obiekt, który ma semantykę wartości to obiekt, który można kopiować oraz do którego można przypisywać. O takich obiektach można myśleć jak o wartościach typów zmiennych (int, float itp.), które można dowolnie tworzyć, kopiować i zmieniać.

właścicielem[10]. W programowaniu gier często używa się licznika referencji. Inteligentne wskaźniki mogą mieć zaimplementowaną tą metodę. Polega ona na zliczaniu ilości przypisań danego wskaźnika (ilość wskaźników wskazujących na dany obiekt). Operator delete odejmuje jedną referencję z licznika (zazwyczaj jest to liczba całkowita np. `int`), a gdy licznik dojdzie do zera obiekt zostaje usunięty z pamięci a wskaźnik zerowany. Licznik referencji musi być wspólny dla wszystkich inteligentnych wskaźników. W niektórych przypadkach staje się to wadą, gdyż rozmiar inteligentnego wskaźnika wzrasta dwukrotnie. Tego problemu można pozbyć się przechowując wskaźnik i licznik razem, niestety szybkość dostępu do wskaźnika spada, gdyż teraz znajduje się o jedno wyłuskanie dalej. Przeważnie wskaźnik tworzy się tylko raz, ale wyłuskuje wielokrotnie. Najwydajniej jest przechowywać licznik referencji bezpośrednio w obiekcie. Ta technika nazywana jest intruzyjnym zliczaniem referencji. Jak sama nazwa wskazuje to rozwiązanie też nie jest pozbawione wad. Każdy obiekt, który chcielibyśmy używać w formie inteligentnego wskaźnika powinien zawierać dodatkowe pole, w związku z tym należy pamiętać o tym projektując cały system klas.

Kolejną techniką, której można użyć są listy referencji. Ilość odwołań do obiektu nie jest istotna. Ważna jest tylko informacja, że nie ma żadnych wskaźników na obiekt i trzeba go wtedy usunąć. Wskaźniki tworzą listę dwukierunkową. Gdy tworzymy nowy inteligentny wskaźnik, tworzymy go na podstawie poprzedniego i dołączamy do listy. Destruktor usuwa wskaźnik z listy a obiekt jest usuwany, kiedy na liście nie ma już żadnego wskaźnika. Nie można zastosować listy jednokierunkowej, gdyż operacje na niej (szczególnie usuwanie) są wykonywane w czasie liniowym. Wektor również nie może być użyty, dlatego że wskaźniki nie tworzą spójnego obszaru.

Zliczanie referencji, choć jest dobrym pomysłem, nie jest pozbawione wad. Przede wszystkim można natknąć się na cykl referencji. Załóżmy, że mamy obiekt O1 oraz obiekt O2. O1 zawiera inteligentny wskaźnik na O2, a O2 zawiera inteligentny wskaźnik na O1. O1 i O2 wskazują na siebie nawzajem zatem śledzenie referencji nie ma możliwości wykryć takiego cyklu więc obiekty nie zostaną usunięte z pamięci.

Przyjrzyjmy się nieco bliżej upodobnieniu inteligentnego wskaźnika do normalnego. Inteligentne wskaźniki powinny dopuszczać taką samą składnię równości i nierówności jak wskaźniki zwykłe. Załóżmy, że mamy inteligentny wskaźnik `pPtr`, `pPtr2` i wskaźnik `p`.

```
/*1*/if( pPtr ) {}
/*2*/if( !pPtr ) {}
/*3*/if( pPtr == 0 ) {}
/*4*/if( pPtr == pPtr2 ) {}
/*5*/if( pPtr == p ) {}
```

1) Sprawdzenie, czy wskaźnik nie jest pusty. 2) Sprawdzenie, czy wskaźnik jest pusty. 3) Jawne porównanie z zerem. 4) Porównanie dwóch inteligentnych wskaźników. 5) Porównanie ze zwykłym wskaźnikiem.

Jeśli w implementacji inteligentnego wskaźnika dodamy konwersję do zwykłego wskaźnika (`operator T*()`) powyższe porównania skompilują się. Niestety robiąc tak

[10]Tak działa `std::auto_ptr` w standardzie C++

odblokowujemy operator `delete`. Lepszym rozwiązaniem było by zdefiniowanie operatora `bool`.

```
operator bool() const
{
	return m_pPtr != 0;
}
```

Niestety w tym przypadku dochodzą nowe błędy:

```
bool b = pPtr;
if( pPtr * 23 == 345 ) {}
```

Taka konwersja niestety to umożliwia. Inteligentny wskaźnik zachowuje się zupełnie jak typ `int`. Dodatkowo odpada porównanie z zerem[11]. Najlepszym rozwiązaniem, które we wszystkich przypadkach będzie poprawne jest stworzenie operatorów równości i nierówności dla wszystkich typów.

- Porównanie inteligentnego wskaźnika ze zwykłym wskaźnikiem
- Porównanie zwykłego wskaźnika z inteligentnym wskaźnikiem
- Sprawdzenie czy wskaźnik istnieje
- Porównanie inteligentnego wskaźnika typu T z inteligentnym wskaźnikiem drugiego U.
- Porównanie inteligentnego wskaźnika typu U z inteligentnym wskaźnikiem typu T.
- Operator eliminujący niejednoznaczność.

Przedostatnie dwa punkty są konieczne. Jeśli byśmy je pominęli otrzymamy niejednoznaczność w przypadku porównania inteligentnego wskaźnika klasy bazowej A ze zwykłym wskaźnikiem klasy B, która jest pochodną klasy A. Eliminacja niejednoznaczności daje nam możliwość poprawnego działania porównania dwóch inteligentnych wskaźników różnych typów. Każda konkretyzacja inteligentnego wskaźnika ma swój własny `operator==` przez co kompilator nie wie, który operator ma wybrać. Operator eliminujący niejednoznaczność w rzeczywistości porównuje zwykłe wskaźniki. Kompilator zgłosi błąd kompilacji jeśli porównanie nie będzie miało sensu. Tak wygląda implementacja powyższych punktów (poza sprawdzeniem czy wskaźnik istnieje):

```
template<class T>
class CSmartPointer
{
	public:
```

[11] Należy usunąć `operator T*()` ponieważ powoduje on błąd kompilacji. Kompilator nie wie, który operator ma wybrać przy porównywaniu z zerem.

```
        /*...*/

        bool operator!()
        {
              return m_pPtr == 0;
        }

        inline friend bool operator==(const CSmartPointer& l,
        const T* r)
        {
              return l.m_pPtr == r;
        }

        inline friend bool operator==(const T* l,
        const CSmartPointer& r)
        {
              return l == r.m_pPtr;
        }

        template<class U>
        inline friend bool operator==(const CSmartPointer& l,
        const U* r)
        {
              return l.m_pPtr == r;
        }

        template<class U>
        inline friend bool operator==(const U* l,
        const CSmartPointer& r)
        {
              return l == r.m_pPtr;
        }

        template<class U>
        bool operator==(const CSmartPointer<U>& r) const
        {
              return m_pPtr == r.m_pPtr;
        }

        //To samo dla operator!=

};
```

Sprawdzenie, czy wskaźnik istnieje (`if( pPtr )`) ogólnie nie jest możliwe jeśli nie zostanie zdefiniowany operator konwersji to zwykłego wskaźnika. W związku z tym pozostaje pisanie `if( pPtr != 0 )`, albo zablokowanie operatora delete. Można to zrobić definiując prywatną klasę, która ma przeciążony operator delete.

```
template<class T>
class CSmartPointer
{
      private:

            class CBoolTest
            {
                  void operator delete(void*) { /*...*/ }
            };
```

```
	public:

		operator CBoolTest*() const
		{
			if( !m_pPtr ) return 0;
			static CBoolTest Test;
			return &Test;
		}

	/*...*/
};
```

Teraz sprawdzenie `if( pPtr )` użyje operatora `CBoolTest*`, który zwróci NULL jeśli wskaźnik jest pusty. Operator delete jest zablokowany przez klasę `CBoolTest`. Jeśli ktoś wywoła usunięcie wskaźnika `pPtr` zostanie zgłoszony błąd kompilacji:

```
error C2248: 'CSmartPointer<T>::CBoolTest::operator delete' : cannot access
private member declared in class 'CSmartPointer<T>::CBoolTest'
```

Inteligentne wskaźniki są użytecznym narzędziem, jednak ich implementacja wbrew pozorom nie jest trywialna. Inteligentne wskaźniki rozszerzają funkcjonalność zwykłych wskaźników. Można dodatkowo zabezpieczyć się przed prawdopodobnymi błędami np. eliminacja pustych wskaźników. Należy wziąć pod uwagę także wielowątkowość, która w programowaniu gier występuje bardzo często.

2.2. Zarządzanie pamięcią

Żadna aplikacja komputerowa nie jest w stanie działać bez użycia pamięci RAM. Programiści mają do dyspozycji stos oraz stertę. Na stercie pojawiają się obiekty alokowane dynamicznie przez operator `new` lub `malloc/calloc/realloc`. Obiekty polimorficzne narzucają wręcz konieczność tworzenia ich na stercie. Niektóre mechanizmy takie jak Pointer To Implementation (pimpl)[12] także muszą zostać utworzone na stercie. Ważnym aspektem dynamicznego przydzielania pamięci jest jej zwalnianie, by zapobiec wyciekom. W dzisiejszych czasach gry komputerowe wymagają dużej ilości pamięci RAM. Przetwarzają bowiem wiele danych, alokują wiele obiektów zarówno związanych z rozgrywką jak i z grafiką bądź dźwiękiem (tekstury, modele, muzyka, itp.). Dane te mogą być w ciągłej rotacji, czyli może zdarzyć się tak, że jedne dane są usuwane z pamięci a w ich miejsce umieszczane są nowe. Za przykład posłużyć może model lasu. Gracz porusza się po olbrzymim terenie pokrytym drzewami, trawą, krzewami, kamieniami. Nie jest możliwe wczytanie całej lokacji do pamięci, gdyż musiało by jej być znacznie więcej niż przeciętny gracz posiada i taka lokacja wczytywała by się bardzo długo. Gry, które korzystają z takich poziomów ładują dane w locie (podczas rozgrywki, nie pojawia się żaden ekran ładowania) – w osobnych wątkach. Ładowane są te modele, które aktualnie gracz może zobaczyć, natomiast te, których już nie zobaczy są usuwane z pamięci. Jak widać system zarządzający zasobami jest ciągle w ruchu. Pamięć może być cały czas zwalniana i przydzielana dla nowych obiektów.

Język C/C++ oferuje programistom kilka sposobów alokacji pamięci:

- `new` – jest to standardowy operator przydzielania pamięci. W przypadku niepowodzenia rzucany jest wyjątek.
- `new(std::nothrow)` – działa podobnie do operatora `new` z tą różnicą, że nie rzuca wyjątku tylko ustawia wskaźnik na NULL. Jest to istotna sprawa w programowaniu gier, gdyż ze względów wydajnościowych wyłącza się obsługę RTTI.
- placement `new` – ten operator przydziela pamięć z puli podanej jako jego argument.
- `malloc` - jest to standardowa funkcja przydziału pamięci w języku C. Różni się od operatora `new` tym, że można go używać do tworzenia obiektów (operator `new` wywołuje konstruktor stworzonego obiektu, `malloc` nie).

Wyżej wymienione operatory mają swoje odmiany, które pomagają śledzić wycieki pamięci w trybie debug. Standardowa biblioteka CRT Visual Studio oferuje następujące usprawnienia:

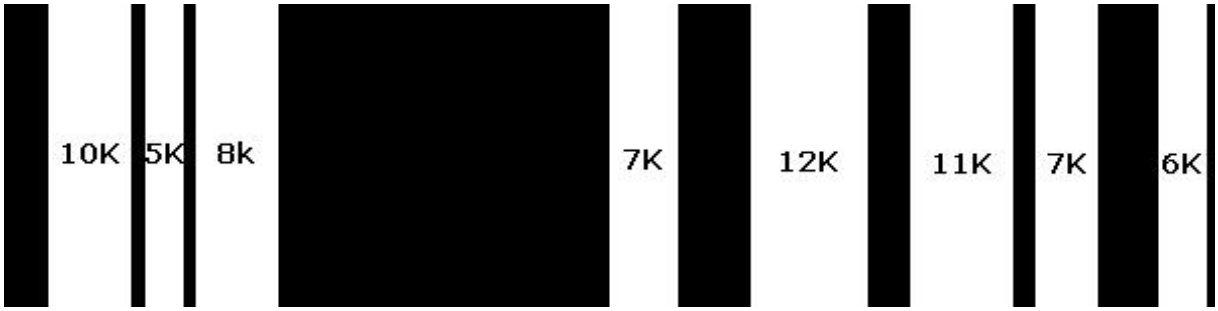
[12] Pimpl to mechanizm, który polega na utworzeniu struktury danych, która ma wszystkie pola jakie powinna mieć dana klasa. Ta klasa, w konstruktorze tworzy dynamicznie tą strukturę (pimpl), która stanowi jedyne pole klasy. Deklaracja struktury (pimpl) jest umieszczona w pliku cpp, także klasa w swojej deklaracji nie wie jakie pola ma struktura. Dzięki temu mechanizmowi kompilacja kodu odbywa się znacznie szybciej i dużo sprawniej wygląda zmiana pól klasy, które odbywają się jako zmiana pola struktury. Kompilowany jest wówczas jedynie plik z deklaracją struktury, czyli plik wykonywalny, a żadne inne pliki, które mogły by korzystać z owej klasy nie będą przekompilowywane.

- `new( _NORMAL_BLOCK, __FILE__, __LINE__ )` – ta wersja operatora new przyjmuje trzy parametry:
    1. Typ alokacji pamięci. Typy te są śledzone osobno w celu detekcji i raportowania wycieków. Dostępne typy to:
        a) **_NORMAL_BLOCK** – każde wywołanie malloc/calloc lub `new` tworzy Normal Block.
        b) **_CRT_BLOCK** – bloki pamięci alokowane wewnętrznie przez funkcje dynamicznie dołączanych bibliotek są oznaczane jako bloki CRT, dzięki temu mogą być trzymane osobno.
        c) **_CLIENT_BLOCK** – aplikacja może przechowywać specjalne śledzenie alokacji w celu debug'owania.
        d) **_FREE_BLOCK** – bloki, które zostały zwolnione i usunięte z listy.
        e) **_IGNORE_BLOCK** – pewne bloki można wyłączyć ze śledzenia wycieków na pewien czas.
    2. Plik, w którym pamięć została zaalokowana.
    3. Linia w pliku gdzie pamięć została zaalokowana

- `_malloc_dbg(_size, _NORMAL_BLOCK, __FILE__, __LINE__)` – wszystkie parametry poza `_size` są takie same. Pierwszy parametr określa ilość zarezerwowanego miejsca (w bajtach).

Wersja debug `malloc` musi używać wersji debug `free` (`_free_dbg(_ptr, _NORMAL_BLOCK)`). Operator `new` w trybie debug, poza pierwszymi trzema parametrami może też przyjąć dodatkowy parametr (tak jak normalny operator `new`). W przypadku wycieku pamięci kompilator wyrzuci na konsolę odpowiedni komunikat. W tym przypadku będzie to dokładna ścieżka do pliku, linia oraz ilość bajtów, które nie zostały zwolnione. Aby wycieki pamięci były monitorowane należy skorzystać z odpowiednich funkcji biblioteki CRT.

W programowaniu gier istnieje duży problem z częstym przydzielaniem i zwalnianiem pamięci. W praktyce alokacja i dealokacja pamięci nie powinna być stosowana w głównej pętli gry (dopuszczalne momenty są podczas ładowania kolejnych etapów gry). Jednakże, jak wyżej zostało nadmienione niektóre gry już w trakcie rozgrywki zwalniają jedne zasoby, by na ich miejsce załadować kolejne. Zatem powstaje problem, gdyż częste alokacje i dealokacje pamięci powodują fragmentację. Duża fragmentacja pamięci może doprowadzić do tego, że nie znajdzie się blok o wystarczająco dużym rozmiarze niż aktualnie jest potrzebny.



Rysunek 1. Sfragmentowana pamięć. Całkowita wolna pamięć to 66K. Niestety nie jest możliwe zarezerwowanie więcej niż 12K ponieważ nie ma ciągłego bloku pamięci większego niż 12K.

Spowoduje to znaczny spadek wydajności aplikacji ponieważ system operacyjny zacznie używać wirtualnej pamięci. Na niektórych platformach, takich jak konsole z niezbyt zaawansowanym systemem operacyjnym, może spowodować to załamanie się aplikacji. W programowaniu gier taka sytuacja jest niedopuszczalna. Aby temu zapobiec używa się managerów pamięci.

2.2.1. Manager obiektów o stałym rozmiarze

Niemal każda aplikacja korzysta z małych, kilkubajtowych obiektów. W grach komputerowych takim obiektem może być cząsteczka (particle). Efekty oparte na cząsteczkach korzystają z kilkuset lub kilku tysięcy małych obiektów. Domyślny przydzielacz pamięci C++ nie radzi sobie zbyt dobrze z zadaniem przydzielania pamięci dla niewielkich obiektów. Każdą przydzieloną pamięcią trzeba jakoś zarządzać dlatego zawsze przydzielana jest jakaś dodatkowa pamięć. Za pomocą operatora `new` będzie to od 4 do 32 dodatkowych bajtów. Jeśli zarezerwujemy większy blok narzut będzie niewielki (kilka procent), natomiast dla bloku ośmiobajtowego będzie to już od 50 do 400%. Jak widać dynamiczne tworzenie wielu małych obiektów jest zupełnie nieopłacalne pamięciowo jak i szybkościowo.

Najprostszym managerem pamięci może być tablica bajtów.

```
struct SStruct
{
	float f;
	int i;
	char c;
};

unsigned char memory[20];
unsigned char* pMem = memory;

SStruct* pStruct1 = (SStruct*)pMem;
pMem += sizeof(SStruct);
```

Ten przykład ilustruje jak w prosty sposób można przydzielać pamięć nie bojąc się o wycieki czy fragmentację. Oczywiście takie rozwiązanie ma wiele wad. Pamięć przydzielona w ten sposób nie może być zwolniona (`delete pStruct1` spowoduje załamanie się aplikacji). W przypadku usuwania obiektów nie wiemy, które bajty są wolne do użycia.

W programowaniu gier rzadko kiedy korzysta się z rozwiązań ogólnego przeznaczenia. Dużo lepiej jest tworzyć narzędzia odpowiednie dla konkretnego zastosowania. Dzięki takiemu podejściu można uzyskać dużo większą wydajność oraz uprościć zadanie. Załóżmy więc, że tworzymy manager pamięci dla konkretnych obiektów. Taki manager będzie konkretyzowany typem. W takim razie znamy już rozmiar danych. Możemy także

dynamicznie określić rozmiar puli pamięci jakiej będziemy używać. Skupmy się na początek na prostszym rozwiązaniu.

```
template<class _T_>
class TCMemoryManager
{
	enum
	{
		DATA_SIZE = sizeof(_T_)
	};

	typedef unsigned char* DataPtr;

	public:

		TCMemoryManager(unsigned int _uiCount)
		{
			m_uiCount = _uiCount;

			m_pMemory = new unsigned char[m_uiCount * DATA_SIZE];

			FreeAll();
		}

		~TCMemoryManager()
		{
			delete [] m_pMemory;
		}

		_T_* Allocate()
		{
			_T_* pPtr = 0;

			if( !m_sFree.empty() )
			{
				pPtr = (_T_*)m_sFree.top();
				m_sFree.pop();
			}

			return pPtr;
		}

		void Free(const _T_* _pPtr)
		{
			m_sFree.push( (DataPtr)_pPtr );
		}

		void FreeAll()
		{
			m_pCurrMem = m_pMemory;

			while( !m_sFree.empty() )
			{
				m_sFree.pop();
			}

			for(unsigned int i = m_uiCount; i --> 0;)
			{
```

```
                        m_sFree.push(m_pCurrMem);
                        m_pCurrMem += DATA_SIZE;
                    }
                }

        private:

                std::stack<DataPtr>     m_sFree;

                unsigned int        m_uiCount;
                unsigned char*      m_pMemory;
                unsigned char*      m_pCurrMem;
};
```

Powyższy kod prezentuje cały manager pamięci. Jest on oparty o stos wolnych bloków. Został tutaj użyty stos z STL, ale można zastosować własną, wydajniejszą wersję stosu. W konstruktorze określana zostaje maksymalna ilość obiektów jakie będą mogły być zaalokowane tym managerem i utworzony jest bufor pamięci. Następnie wywołana jest metoda `FreeAll`, która czyści stos, a następnie uzupełnia go nowymi wskaźnikami na dane (obiekty typu `_T_`). Metoda `Allocate` zwraca wskaźnik na obiekt, lub zerowy wskaźnik jeśli nie ma już wolnego miejsca. Wskaźnik na nowy obiekt jest to po prostu wskaźnik z wierzchu stosu. Jeśli stos nie jest pusty, zostanie zwrócony poprawny wskaźnik. Metoda `Free` usuwa wybrany wskaźnik dodając go na stos wolnych wskaźników. Ten manager usuwa podstawowy problem fragmentacji pamięci, szybkości alokacji i dealokacji (te obiekty są zawsze zaalokowane) oraz wycieków. Wycieki mogą powstać w samym managerze. Jeśli programista zapomni użyć metody `Free` może w końcu okazać się, że nie ma dostępnych miejsc w puli, chociaż takie będą, tylko nie będzie wiadomo pod jakimi adresami. Niestety bardzo trudne może okazać się przydzielenie pamięci dla tablicy obiektów ponieważ musiały by znajdywać się w spójnym obszarze pamięci, a tego ten manager nie gwarantuje. Jednym z możliwych sposobów było by stworzenie dynamicznie tablicy wskaźników na obiekty i każdemu jej elementowi przypisanie wskaźnika z managera pamięci.

Jak wcześniej zostało wspomniane taki przydzielacz pamięci nie radzi sobie z klasami. Pamięć przydzielona w ten sposób nie wywoła domyślnego konstruktora obiektu ponieważ działanie tego managera jest zbliżone do działania funkcji `malloc`. Aby wywołać domyślny konstruktor należy użyć operatora `new`. Obecnie, aby przydzielić pamięć należy wykonać poniższe instrukcje:

```
TCMemoryManager<SStruct> MemoryManager(2);
//...
SStruct* pStruct1 = MemoryManager.Allocate();
//...
MemoryManager.Free( pStruct1 );
```

Gdzie SStruct to jakaś struktura danych. Jeśli zamiast struktury była by klasa, to już jej konstruktor nie zostanie wywołany co, w większości przypadków, spowoduje błędy

aplikacji (zazwyczaj w konstruktorach inicjuje się pola obiektu[13]). Najlepiej było by tworzyć obiekty i zwalniać je tak jak się to robi standardowo w C++:

```
CClass* pClass = new CClass();
//...
delete pClass;
```

Aby umożliwić taki mechanizm klasa powinna mieć przeciążony `operator new`. Ten operator może przyjąć dodatkowy argument jeśli odpowiednio go przeciążymy. Dodatkowy argument powinien być to manager pamięci. Dzięki temu w definicji operatora możemy użyć managera do alokacji obiektu.

```
CClass* pClass = new( pMemoryManager ) CClass();
```

W tej konstrukcji, nie jest używany domyślny `operator new`[14], tylko jego przeciążona wersja, która nie alokuje pamięci, ale korzysta z managera pamięci. To rozwiązanie nie jest zbyt eleganckie. Niesie za sobą konieczność alokowania pamięci w sposób niestandardowy co może prowadzić do pomyłek i używania `new` bez argumentu. Dodatkowo problem jest w użyciu operatora `delete`, ponieważ nie przyjmuje on dodatkowych parametrów, więc nie będzie wiadomo w jaki sposób ma zwolnić pamięć. Problem ten można obejść tworząc pole ze wskaźnikiem na manager pamięci, oraz metodę, która ustawia ten wskaźnik (np. w konstruktorze, lub w metodzie inicjalizacji), następnie w definicji operatora `delete` obiekt zwraca wskaźnik na manager pamięci, który za pomocą metody `Free` zwalnia obiekt. Niestety, jak widać, obiekt, który miałby być zarządzany za pomocą managera pamięci wymaga dopisania wielu dodatkowych metod. Można to nieco usprawnić tworząc specjalną klasę, która będzie miała te metody już zaimplementowane. Klasa, która będzie zarządzana powinna po prostu odziedziczyć z tej klasy z metodami. Dodatkowo, aby usprawnić używanie operatora `new` i obciąć do minimum ilość dodatkowych metod można manager pamięci zrobić singleton'em.

```
template<class _T_>
class TCMemoryManager : public TCSingleton<TCMemoryManager<_T_>>
{
	typedef unsigned char* DataPtr;

	public:

		class CNew
		{
			public:

				void* operator new(size_t _stSize)
				{
					return TCMemoryManager<_T_>::GetSingletonPtr()
					->Allocate();
```

[13] W kwestii wydajnościowej małe obiekty powinny mieć puste konstruktory.
[14] Domyślnego `operator new` używa się korzystając z globalnej przestrzeni nazw: `::operator new`

```
                        }

                        void operator delete(void* _pPtr)
                        {
                              TCMemoryManager<_T_>::GetSingletonPtr()
                              ->Free( (_T_*)_pPtr );
                        }

                        void* operator new(size_t _stSize,
                        TCMemoryManager<_T_>* _pMem)
                        {
                              return _pMem->Allocate();
                        }

            };

            //...
};

class CClass : public TCMemoryManager<CClass>::CNew
{
      //...
};
```

Użycie takiej klasy jest już bardzo proste. Jedyne o czym trzeba pamiętać to o utworzeniu singleton'u managera pamięci. Należy to zrobić oczywiście przed utworzeniem obiektu `CClass`. Należy również pamiętać by nie usuwać żadnych obiektów po usunięciu singleton'u.

Prostszym rozwiązaniem może być użycie operatora placement `new`. Wystarczy napisać jedynie metodę (w managerze pamięci) do tworzenia obiektów za pomocą tego operatora.

```
_T_* Alloc()
{
      _T_* pPtr = Allocate();
      return new( pPtr ) _T_();
}
```

Teraz wywołanie metody `Alloc` spowoduje również wywołanie domyślnego konstruktora. Istnieje coś takiego jak domyślny konstruktor typów atomowych. Domyślnie typ atomowy nie ma zdefiniowanej wartości, ale jego domyślny konstruktor ustawia początkową wartość na 0. Metoda `Alloc` działa również dla typów atomowych wywołując ich domyślny konstruktor. Niestety, korzystamy tutaj z globalnego operatora placement `new`. Taki operator już został przeciążony więc występuje niezgodność dopasowania argumentu. Wobec tego należy dodać kolejną wersję operatora placement `new` z argumentem typu `_T_*`. Jedyne co ten operator robi to po prostu zwraca wskaźnik podany jako drugi argument.

Kwestia różnych sposobów alokacji pamięci pozostaje do rozstrzygnięcia przez samego programistę. Powinien użyć takiej metody, która będzie najlepsza dla jego potrzeb. Przeciążone operatory (nie tylko `new`) powodują spadek wydajności. Należy mieć to na uwadze tworząc manager pamięci, który ma działać szybko.

Na koniec pozostało zwalnianie pamięci. Napisane zostało wyżej, że funkcja `malloc` nie wywołuje konstruktora, tak samo funkcja `free` nie wywołuje destruktora. Przedstawiony wyżej manager pamięci też nie wywołuje destruktora. Na szczęście nie stanowi to problemu w języku C++. Tutaj destruktor można wywołać „ręcznie". Do metody `Free` należy dopisać linijkę: `_pPtr->~_T_();` i destruktor już będzie wywoływany. Dla typów atomowych nie będzie to miało znaczenia, nie wystąpi błąd kompilacji i aplikacja się też nie załamie.

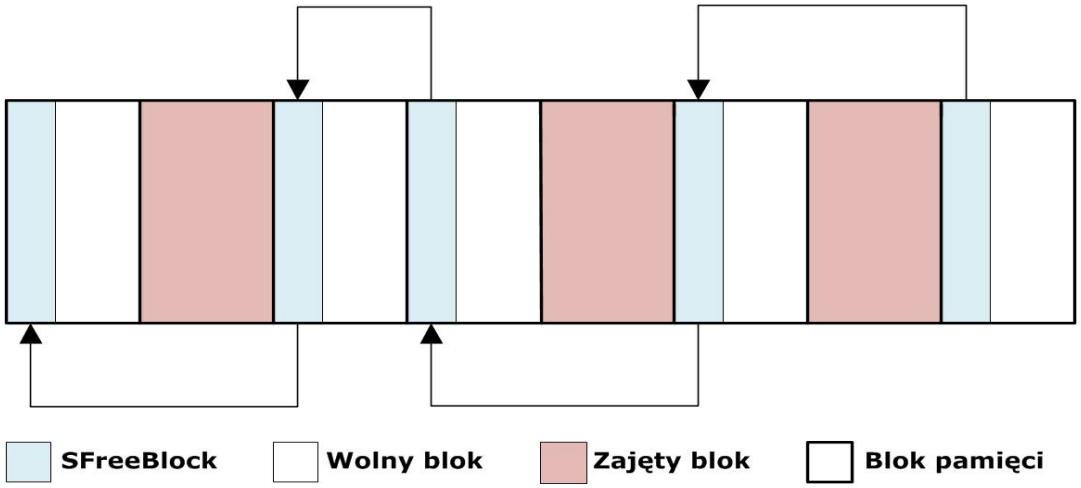
Ten manager działa dla każdych obiektów, zarówno dużych jak i małych. Poza stosem nie rezerwuje dodatkowej pamięci.

2.2.2. Model FreeList

Manager pamięci typu FreeList jest podobny do managera przedstawionego powyżej. Również jest konkretyzowany typem, więc może zarządzać pamięcią tylko dla jednego typu obiektów. Posiada dwie tablice. Jedna jest tablicą obiektów, a druga jest tablicą wskaźników na obiekty (jest to odpowiednik stosu wolnych wskaźników). Rozmiar puli pamięci również jest ustalany z góry (przy pewnych modyfikacjach może być dynamicznie rozszerzalny, ale wiąże się to z mniejszą wydajnością), więc wybór tablicy przechowującej wskaźniki jest znacznie lepszy, niż struktury alokujące pamięć dynamicznie (oczywiście stos jest tylko jednym z możliwych zastosowań dla pojemnika przechowującego wskaźniki. Można zbudować stos oparty o tablicę właśnie tak jak jest to robione w modelu FreeList).

```
TCFreeList(unsigned int _uiCount)
{
	assert( _uiCount > 0 );
	m_pMemory = new _T_[_uiCount];
	m_ppFreeList = new _T_*[_uiCount];

	FreeAll();
}
```

Korzystając z tej wersji puli pamięci mamy dodatkową korzyść w postaci zainicjalizowanych obiektów, w związku z tym, nie trzeba się już martwić wywołaniem domyślnego konstruktora. Lepiej jednak było by oszczędzić pamięć i nie wywoływać dodatkowego operatora `new` dla tablicy wolnych bloków. Wskaźniki na wolne bloki można przechowywać bezpośrednio w puli pamięci. Będzie to jednokierunkowa lista. Pula pamięci nie może już być typu `_T_`, ponieważ struktura tworząca listę będzie przechowywana w pierwszych czterech bajtach nieużywanego bloku. W Praktyce wygląda to tak jak na rysunku 2.



Rysunek 2. Model pamięci managera FreeList. Każdy wolny blok pamięci posiada wskaźnik na następny wolny blok.

A o to podstawowa implementacja:

```
template<typename _T_>
class TCFreeList
{
	private:

		struct SFreeBlock
		{
			SFreeBlock* pNext;
		};

		typedef	unsigned char	Data;
		typedef	Data*		DataPtr;

private:

		DataPtr		m_pMemory; //pula pamieci
		SFreeBlock*	m_pFirstFree; //pierwszy wolny blok
		unsigned int	m_uiCount; //ilosc blokow w puli

	//...
};
```

Struktura `SFreeBlock` jest to lista wolnych bloków pamięci. Dla uproszczenia użyte zostały zdefiniowane typy Data i `DataPtr`. Konstruktor klasy jest niemal identyczny jak w przypadku poprzedniego managera pamięci.

```
TCFreeList(const unsigned int _uiCount)
{
	assert( sizeof(_T_) >= sizeof(SFreeBlock) );
```

```
	m_uiCount = _uiCount;
	m_pMemory = new Data[_uiCount * sizeof(_T_)];

	FreeAll();
}
```

Podobnie jak poprzednio metoda `FreeAll` ponownie tworzy wolne bloki.

```
void FreeAll()
{
	DataPtr pCurrMem = m_pMemory;
	SFreeBlock* pLastFree = 0, *pCurrFree = 0;
	m_pFirstFree = 0;

	for(unsigned int i = m_uiCount; i --> 0;)
	{
		pCurrFree = (SFreeBlock*)pCurrMem;
		pCurrMem += sizeof(_T_);

		pCurrFree->pNext = pLastFree;
		pLastFree = pCurrFree;
	}

	m_pFirstFree = pLastFree;
}
```

Pula pamięci jest typu `unsigned char` ponieważ daje to możliwość adresowania danych na poziomie jednego bajta, aczkolwiek w tym managerze najmniejszy typ danych musi być rozmiaru `sizeof(SFreeBlock)` żeby można było odpowiednio zaadresować wskaźniki na wolne bloki. Wersja managera z osobną listą wolnych bloków pozwala na zarządzanie typami wielkości nawet jednego bajta.

Dla uproszczenia przykład zawiera najprostszą metodę alokacji. Nie wywołuje ona domyślnego konstruktora. Nie ma też przeciążonych operatorów `new`. Każda opcja alokacji pamięci jest przedstawiona w poprzednim managerze a w tym wygląda analogicznie.

```
_T_* Allocate()
{
	_T_* pPtr = 0;

	if( m_pFirstFree != 0 )
	{
		pPtr = (_T_*)m_pFirstFree;
		m_pFirstFree = m_pFirstFree->pNext;
	}
```

```
	return pPtr;
}
```

Metoda zwraca 0 (NULL) jeśli nie ma już wolnych bloków w liście. Dodatkowo można przechowywać licznik wolnych bloków w zmiennej prywatnej i zrobić dodatkowe porównanie, lub `assert`, dla większej pewności.

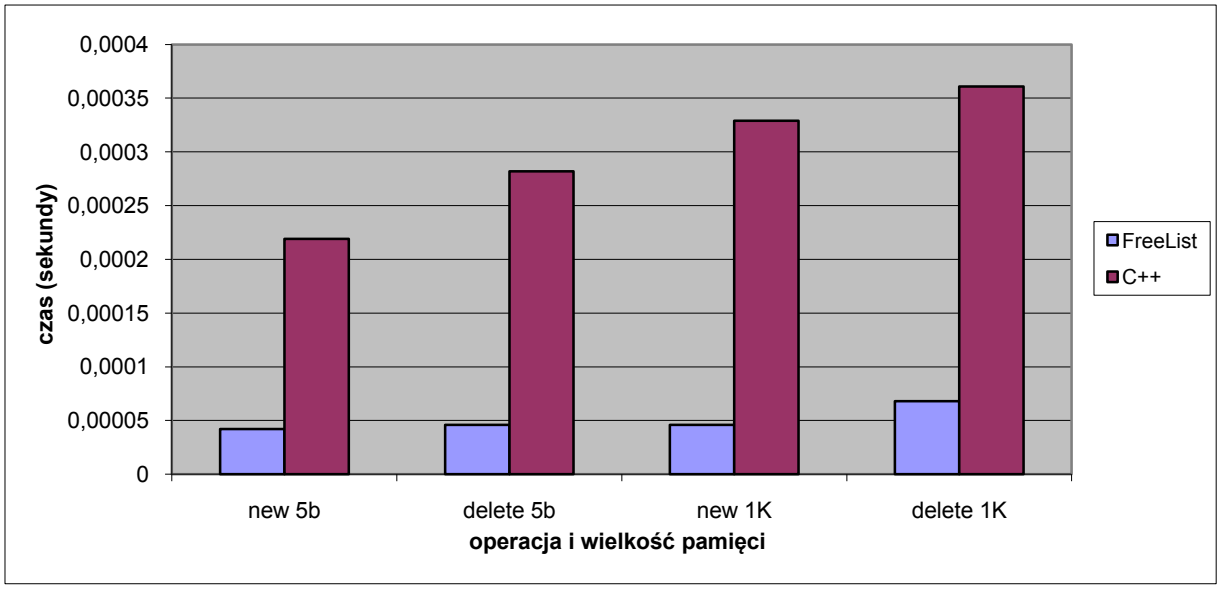
Usuwanie elementu polega na dodaniu bloku pod adresem wskaźnika przekazywanego jako parametr do listy wolnych bloków. Oczywiście wskaźnik nie może mieć wartości 0 oraz musi pochodzić w puli pamięci tego managera.

```
void Free(_T_* _pPtr)
{
	assert( _pPtr != 0 );
	assert((unsigned int)_pPtr >= (unsigned int)m_pMemory &&
		(unsigned int)_pPtr <= (unsigned int)m_pMemory + m_uiCount *
		sizeof(_T_) - sizeof(_T_)  );

	SFreeBlock* pBlock = (SFreeBlock*)_pPtr;
	pBlock->pNext = m_pFirstFree;
	m_pFirstFree = pBlock;
}
```

Pierwszy `assert` sprawdza czy wskaźnik nie jest zerowy. Drugi `assert` sprawdza czy wskaźnik znajduje się w puli managera pamięci. W tym celu wszystkie wskaźniki sprowadzane są do adresów pamięci, które im odpowiadają i obliczane jest odpowiednie przesunięcie. Dzięki temu taki kod zostanie wykryty podczas debug'u:

```
SStruct* p = new SStruct;
MemMgr.Free( p );
```

Zakładając, że `new`, w tym przypadku, jest globalnym operatorem, czyli standardowym z C++. Gdyby została używa wersja przeciążona tak jak w poprzednim managerze, wówczas wskaźnik byłby poprawny ponieważ został by stworzony przed manager pamięci. Niezależnie od wersji alokacji jaką przyjmiemy taki `assert` zwykle jest pomocy (w poprzednim managerze pamięci również można go zastosować). Ważną rzeczą jest pierwsza instrukcja. Otóż pobieramy blok pamięci spod adresu wskazywanego przez zwalniany obiekt. Rzutowane jest to na strukturę, która dodawana jest do listy wolnych bloków.



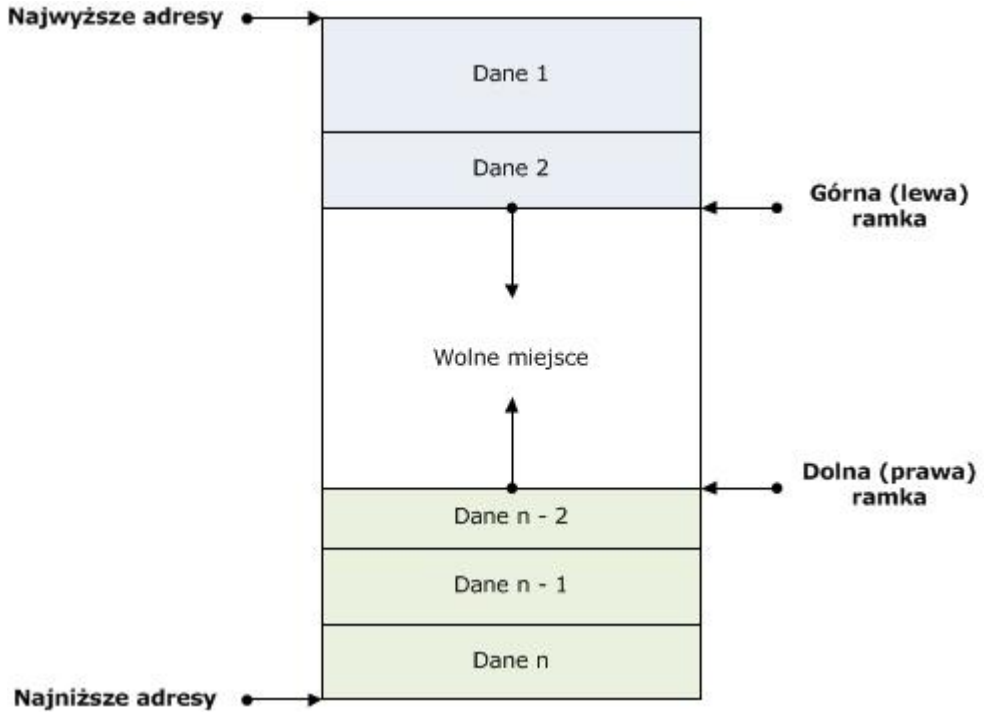
Rysunek 1 . Wykres przedstawiający różnicę w czasie alokacji i zwalniania pamięci metodą FreeList oraz operatora new i delete z C++.

Powyższy wykres prezentuje różnicę czasu alokacji i zwalniania pamięci za pomocą managera FreeList oraz C++. Do testu zostały użyte bloki pamięci o rozmiarze 5 bajtów oraz 1024 bajty. Oczywiście pula pamięci alokowana w managerze pamięci nie wlicza się do czasu alokacji, gdyż takie operacje zwykle są wykonywane podczas inicjalizacji, więc są pomijalne. Jak widać rozmiar bloku alokowanego przez manager nie wpływa na szybkość alokacji. Inaczej wygląda to przy użyciu C++, gdzie jest zasadnicza różnica czy alokujemy duży blok czy mały. Zwalnianie zwykle trwa dłużej co zostało potwierdzone na wykresie. Manager pamięci niemal w identycznym czasie alokuje i zwalnia bloki co stanowi kolejną zaletę. Ponad to alokacja i zwalnianie odbywa się w czasie stałym. Przyspieszenie FreeList jest oczywiste, chodź na szybszym procesorze C++ nie radzi sobie aż tak źle. Różnica pomiędzy managerem pamięci jest duża, ale patrząc na sam czas C++ zaalokował 1000 bloków o rozmiarze 1024 bajtów w niespełna 0.0003 sekundy.

2.2.3. Ramkowy manager pamięci

Manager pamięci oparty na ramkach działa zupełnie inaczej niż te przedstawione powyżej. Jedyne podobieństwo jest w puli pamięci, która również jest tablicą bajtów. Ramkowy manager pamięci może rezerwować bloki o różnej wielkości tak jak na rysunku 3.

W tym przykładzie pamięć zarezerwowana jest z dwóch stron ponieważ ten manager może być tak zaimplementowany. Jego działanie przypomina stos. Alokujemy pamięć od lewej strony lub od prawej (ewentualnie od góry lub od dołu) i otrzymujemy uchwyt na zaalokowaną pamięć. Istotą tego managera są ramki. Ramką nazwiemy uchwyt na miejsce w pamięci. Za pomocą ramek można łatwo zwalniać pamięć. Wystarczy pobrać ramkę w pewnym miejscu, do którego chcieli byśmy usunąć pamięć a następnie wywołać odpowiednią metodę podając ten uchwyt jako argument. Zostanie zwolniona cała pamięć do miejsca wskazywanego przez ramkę. Dla przykładu skorzystajmy z rysunku 3. Załóżmy, że pobraliśmy ramkę po alokacji Dane 1. Następnie zrobiliśmy alokację dla



Rysunek 3. Model pamięci ramkowego managera. Dwie sterty, jedna alokowana od góry, druga alokowana od dołu.

Dane 2. Teraz, w momencie usuwania pamięci cofamy się do miejsca, do pierwszego bajtu, zaraz za Dane 1. Jest to pierwszy wolny bajt pamięci, która może być ponownie przydzielona. Analogiczna sytuacja dzieje się dla dolnej sterty. Bardzo ważne jest, aby usuwać pamięć w odwrotnej kolejności w jakiej była przydzielana. Jest to wada tego managera pamięci.

Ramkowy manager pamięci świetnie można połączyć z managerem FreeList. Wystarczy dopisać konstruktor, który będzie przyjmował wskaźnik na miejsce w pamięci oraz rozmiar dostępnego miejsca. Ten wskaźnik będzie pochodził z puli ramkowego managera. Patrząc na rysunek 3, taką pulą mogą być Dane 1. Niestety, w wersji dynamicznie powiększanego managera FreeList nie będzie możliwa taka współpraca, gdyż ramkowy manager przydziela stałe bloki. W momencie, gdy Dane 1 są używane przez FreeList i chcieli byśmy zwiększyć mu dostępną pulę należało by usunąć blok Dane 1. Niestety nie można tego zrobić ze względu na blok Dane 2, który również został by usunięty. Trzeba by wtedy robić tymczasowy blok pamięci, który by pamiętał wszystkie bloki, które chcieli byśmy zachować i następnie przepisać je powrotem do puli ale na nowe miejsce. Taka operacja może być zbyt kosztowna czasowo i pamięciowo.

Ramkowy manager pamięci świetnie się nadaje to alokowania i zwalniania pamięci dla poszczególnych etapów gry. Przyjmijmy, że gracz przenosi się z budynku do budynku. Gdy gracz przechodzi do nowego budynku, w momencie etapu ładowania, pamięć zajmowana przez poprzedni budynek jest zwalniana a na jej miejsce jest przydzielana nowa, dla nowego budynku. Dodatkowo, można alokować z obu stron. Z jednej strony mogą być dane statyczne (które bardzo rzadko się realokują) z drugiej dynamiczne, lub z jednej strony może być grafika a z drugiej dźwięk. Dodatkowym atutem ramek jest to, że można bardzo szybko zwolnić pamięć całego poziomu gry. Nie trzeba nawet pamiętać zasobów należących do tego poziomu. Wystarczy, ze będziemy alokować pamięć dla tych zasobów od ramki w ciągłej przestrzeni.

Spójrzmy teraz na przykładową implementację:

```
class CFrameMemoryManager
{

	public:

		typedef unsigned char	Data;
		typedef Data*		DataPtr;

		struct SFrame
		{
			DataPtr pFrame;
			bool  bTop;
		};


	private:

		DataPtr			m_pMemory;
		int			m_iByteAlign;
		int			m_iMemSize;
		DataPtr			m_pBottom;
		DataPtr			m_pTop;
		DataPtr			m_pTopFrame;
		DataPtr			m_pBottomFrame;

};
```

Na początek zdefiniowane zostały typy danych. Następnie zdefiniowana jest struktura będąca ramką. Zawiera ona wskaźnik na miejsce w puli pamięci oraz wartość logiczną odpowiadającą za to czy pamięć była alokowana od dołu czy od góry. Pola klasy są to: pula pamięci managera, wyrównanie pamięci (w bajtach), rozmiar puli pamięci, wskaźniki na spód i na wierzch puli oraz ramki na wierzch i na spód. W konstruktorze klasy można stworzyć i zainicjalizować pulę:

```
CFrameMemoryManager(int _iMemSize, int _iAlign)
{
	m_iMemSize = Align( _iMemSize, _iAlign );

	m_pMemory = new Data[m_iMemSize];

	m_iByteAlign = _iAlign;

	m_pBottom = (DataPtr)Align( (int)m_pMemory, _iAlign );
	m_pTop = (DataPtr)Align( (int)m_pMemory + _iMemSize, _iAlign );

	m_pTopFrame = m_pTop;
	m_pBottomFrame = m_pBottom;
}
```

Najpierw określamy ilość pamięci jaką będziemy rezerwować w puli. Pamięć jest wyrównywana do wielokrotności wyrównania (`_iAlign`). W implementacji można zrezygnować z wyrównania pamięci, ale może się to wiązać ze spadkiem wydajności. Większość procesorów wymaga wyrównanych danych. Wielkość wyrównania równa jest rozmiarowi słowa maszynowego. Procesory 32-bitowe wyrównują do czterech bajtów, zatem każde wyrównanie ma wielokrotność czterech. Każdy lepszy procesor sam wyrównuje dane i programista nie musi się tym martwić. Na przykład struktura `SFrame` z przykładu mimo, że zawiera jedynie czterobajtowy wskaźnik (platforma 32-bitowa) oraz typ logiczny o rozmiarze jednego bajta to rozmiar struktury jest wyrównany do ośmiu bajtów. Niektóre procesory obsługują niewyrównane dane, ale wiąże się to ze spadkiem wydajności.

W kolejnych linijkach alokowany jest bufor pamięci oraz ustawiane są wskaźniki na podstawę i wierzch stosu (również z odpowiednim wyrównaniem). Metoda wykonująca wyrównanie zaokrągla pierwszy argument do wielokrotności drugiego. Czyli, jeśli wielkość pamięci będzie 11 a wyrównanie 2 to metoda zwróci wartość 12.

```
inline unsigned int Align(const int& _iAddress, const int& _iBytes)
{
return ((((unsigned int)_iAddress) + (_iBytes)-1) & (~((_iBytes)-1)));
}
```

W przykładzie zastosowane zostały dwie metody do alokacji: z góry i z dołu.

```
void* AllocateTop(int _iBytes)
{

	_iBytes = Align( _iBytes, m_iByteAlign );

	if( m_pBottomFrame + _iBytes > m_pTopFrame )
	{
		return 0;
	}

	m_pTopFrame -= _iBytes;
	return (void*)m_pTopFrame;
}
```

Metoda `AllocateBottom` jest analogiczna. Różnica jest w dwóch ostatnich linijkach kodu. Używana jest ramka dolna, której wskaźnik jest zwiększany a nie zmniejszany. Wszystkie bajty jakie rezerwujemy tymi metodami są wyrównywane. Jak widać warto dobrze dobrać rozmiar wyrównania. Instrukcja warunkowa sprawdza, czy ramki nie najdą na siebie po alokacji. Jeśli miało by to nastąpić zostanie zwrócona wartość 0. Sama alokacja polega na zwiększeniu (lub zmniejszeniu) pozycji wskaźnika i zwrócenia go.

```
void Free(SFrame _Frame)
```

```
{
	if( _Frame.bTop )
	{
		m_pTopFrame = _Frame.pFrame;
	}
	else
	{
		m_pBottomFrame = _Frame.pFrame;
	}
}
```

Usuwanie jest bardzo proste. Jeśli ramka została pobrana to można na jej podstawie przywrócić pamięć do miejsca, na które wskazuje. Pole `bTop` w ramce wskazuje na to, czy ramka wskazuje na dolną stertę czy na górną. Aktualny wskaźnik stosu jest ustawiany na wskaźnik z ramki.

```
SFrame GetFrame(bool _bTop)
{
	SFrame Frame;
	Frame.bTop = _bTop;

	if(_bTop)
	{
		Frame.pFrame = m_pTopFrame;
	}
	else
	{
		Frame.pFrame = m_pBottomFrame;
	}

	return Frame;
}
```

Na koniec metoda `GetFrame`, która pobiera aktualną ramkę. Argumentem tej metody jest informacja czy chcemy pobrać ramkę górnej czy dolnej sterty. Następnie ustawiany jest wskaźnik ramki na wskaźnik odpowiedniej sterty i ramka jest zwracana.

Ten prosty manager pamięci jest szybki w działaniu i daje możliwość alokowania bloków o dowolnym rozmiarze. Niestety jego działanie przypomina stos więc bardzo ważne jest, aby usuwać dane w odwrotnej kolejności do ich tworzenia. Jest to istotna cecha tego managera, która wymaga by używać go tylko w określonych przypadkach.

Przedstawione powyżej managery pamięci mogą być łatwo zaimplementowane na różnych platformach, gdyż działają na surowym bloku pamięci. Zapobiegają fragmentacji co znacznie zwiększa wydajność całej aplikacji, są przyjazne dla pamięci cache oraz bardzo szybko alokują i zwalniają bloki pamięci. Niestety nic nie jest za darmo. Jak łatwo zauważyć tych managerów nie można używać do wszystkiego. Ich przeznaczenie jest ściśle określone z góry i należy się nimi posługiwać rozważnie. W programowaniu gier

wszystko ma swoją cenę, ale w znakomitej większości przypadków opłaca się stosować rozwiązania dla konkretnych przypadków kosztem zmniejszenia ich ogólności.

2.3. Zarządzanie zasobami

Gry komputerowe korzystają z pokaźnej ilości zewnętrznych zasobów. Zasoby te mogą być dynamicznie zmieniane, wczytywane lub usuwane. Wiele elementów gry korzysta z nich więc warto by były one przechowywane w jakiejś bazie danych. W grach takimi bazami są managery zasobów. Są to klasy odpowiedzialne za odpowiednie wczytywanie, udostępnianie i usuwanie zasobów. Odpowiedzialność spoczywająca na takim managerze jest bardzo duża, gdyż potrzeba szybkiego dostępu do zasobu lub stworzenia, szybkiego usunięcia go oraz wydajnej iteracji po wszystkich zasobach. Dlatego pierwszym krokiem przy tworzeniu managera zasobów jest wybranie odpowiedniego pojemnika. Kolejnym problemem jest wybranie typu zasobu. Należy postawić sobie pytania typu: czy przechowujemy wskaźniki na obiekty? Czy przechowujemy inteligentne wskaźniki na obiekty? Czy przechowujemy uchwyty na obiekty? Czy tworzymy klasę (lub klasy) bazową dla wszystkich typów zasobów i przechowujemy te obiekty bazowe?

Korzystanie z samych wskaźników jest najłatwiejsze, ale też najniebezpieczniejsze. Niech jeden podsystem wyśle żądanie usunięcia zasobu do managera i już inne podsystemy korzystające z tego zasobu tracą ważność wskaźnika co może spowodować załamanie się aplikacji. Oczywiście taką sytuację można ominąć wielokrotnie testując grę w trybie debug. Podstawową rzeczą jest to, aby żaden podsystem nie korzystał z zasobów, które mogły być wcześniej zwolnione. Kod aplikacji należy planować w ten sposób, żeby do takich sytuacji nie dochodziło. Używanie wskaźników niesie jeszcze jeden problem. Nie wiadomo ile jest odnośników do wskaźnika na dany zasób. Dlatego dużo lepiej było by korzystać z inteligentnych wskaźników lub jakiegoś ich odpowiednika.

Mówiąc o zasobach mamy przede wszystkim na myśli pliki odczytywane z dysku. Niemniej jednak za zasób można również potraktować zwykłe obiekty, których będzie tworzyć się wiele (na przykład klasa typu: `CItem` – przedmiot lub `CCharacter` – postać). Manager zasobów może również takimi obiektami z powodzeniem zarządzać. W praktyce manager zasobów może zarządzać każdym typem obiektów i warto ich używać do tego celu.

Podstawą przy projektowaniu managera zasobów jest wybór pojemnika przechowującego je. Najprostszym wyborem może okazać się tablica. Niestety, tablica ma z góry wyznaczony rozmiar, więc kolejny wybór mógłby paść na rozszerzalną tablicę (np. `std::vector`). I to już może być dużo lepszy pomysł pomijając fakt, ze usunięcie zasobu wiąże się z ponowną alokacją pamięci dla takiej tablicy, co powoduje znaczny spadek wydajności. Jeśli potrzeba szybkiego usuwania to najlepiej było by użyć listy. Owszem, usunięcie elementu listy jest szybkie, chociaż niestety wymaga czasu liniowego na odszukanie elementu. Listę natomiast źle się iteruje[15]. W zależności od potrzeb należy wybrać odpowiedni pojemnik. Najczęściej najważniejsze będzie wybieranie zasobu, więc potrzebna jest taka struktura danych, która będzie miała możliwość szybkiego odszukania elementu. Z pomocą przychodzi słownik (na przykład `std::map`), który ma kilka ważnych zalet. Przede wszystkim można posiadać dowolny typ klucza.

[15] Ponieważ elementy listy są wskaźnikami dynamicznie tworzonymi w pamięci nie ma pewności, że zajmą spójny obszar pamięci, więc skoro są porozrzucane iteracja po nich jest bardzo nieefektywna dla pamięci cache.

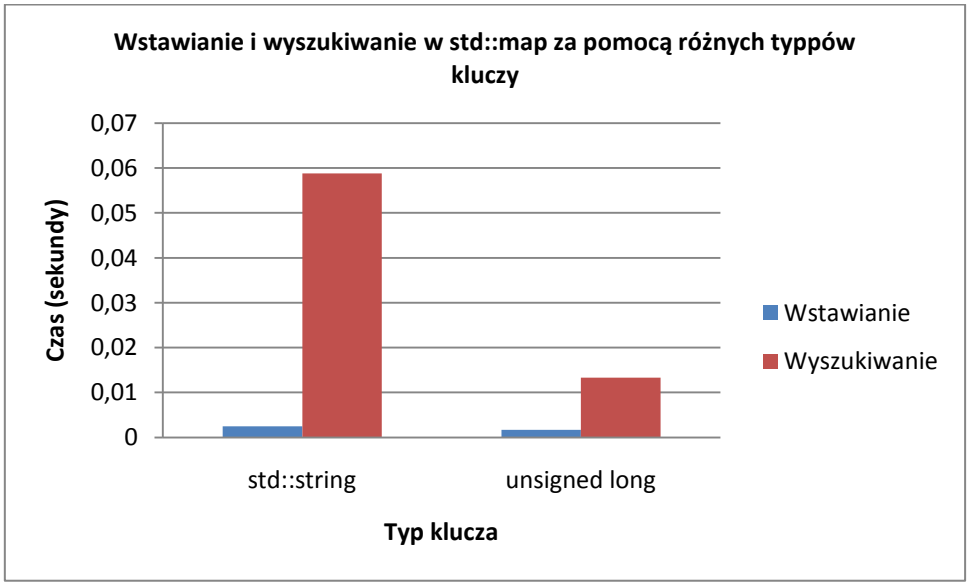
Wyszukiwanie w mapie jest szybkie (O(logn)) ponieważ jest to drzewo czerwono-czarne, więc uzyskujemy szybki dostęp do wybranego elementu. Ponieważ jest to drzewo usuwanie z niego jest również szybkie i nie wymaga czasu liniowego. Głównym problemem mapy jest słabe iterowanie po niej. W drzewie każdy węzeł jest w innym miejscu pamięci, a przeskakiwanie po wskaźnikach nie jest wydajne ze względu na pamięć cache. Sprytnym rozwiązaniem problemu mógłby być manager pamięci, który trzymał by obiekty. Taki manager niestety nie jest łatwy w implementacji ponieważ manager zasobów może mieć dane o różnych rozmiarach (np. tekstury), więc model FreeList nie będzie uniwersalny. Musi być to manager pamięci dla obiektów o różnych rozmiarach i dodatkowo musi mieć bardzo ważną (dla managera zasobów) funkcję iteracji po zarezerwowanych blokach pamięci. Dzięki tej iteracji będzie można swobodnie iterować z mniejszymi dziurami w cache. Przechodzenie po wszystkich zasobach jest bardzo ważne, szczególnie jeśli projektujemy silnik graficzny. Przykładem mogą być siatki geometrii reprezentujące trójwymiarowe modele. W każdej klatce animacji silnik musi przejść po wszystkich modelach i odpowiednio je wyświetlić, niektóre odrzucić, gdyż są niewidoczne, innym ustawić odpowiedni poziom szczegółowości (LOD) no i przede wszystkim posortować. W grafice komputerowej sortowanie geometrii jest bardzo ważne. Sortuje się przede wszystkim po materiale (shader, tekstura) z jakiego dana geometria korzysta, żeby maksymalnie zmniejszyć ilość wywołań do karty graficznej. Dla obiektów przezroczystych również należy zastosować sortowanie po odległości od kamery i odpowiednio je wyrenderować. Jak widać, możliwość przechodzenia po elementach pojemnika jest istotna. Na szczęście nie wszystkie zasoby wymagają iteracji w każdej klatce animacji. Tekstury, dźwięki lub skrypty są częściej wybierane z pojemnika i przypisywane do danego modelu.

Zdecydowanie najrzadziej usuwa się elementy z pojemnika w trakcie działania gry. Jeśli już się usuwa to całą porcję danych w momencie usuwania poziomu gry i ładowania kolejnego. W niektórych grach, gdzie poruszamy się po dużej, otwartej przestrzeni ładowanie nowych lokacji odbywa się w locie (w osobnym wątku). Tutaj szybkość usuwania i wkładania nowych elementów ma znaczenie. Najlepiej wtedy przygotować sobie więcej miejsca w pojemniku od razu, podczas inicjalizacji managera zasobów. Na ogół z góry wiadomo ile miejsca będzie potrzebne żeby łatwo dołożyć nowe zasoby. Dla przykładu, zawsze można zarezerwować 50 lub 100 elementów więcej niż aktualnie potrzeba. Nakład pamięciowy nie będzie wielki (patrząc na dzisiejsze standardy PC, w konsolach nie ma wiele pamięci i tam trzeba radzić sobie nieco inaczej), a szybkość dodawania nowych elementów może przyspieszyć cały proces.

Kolejnym argumentem przemawiającym za wybraniem mapy są klucze. Zasoby muszą być w pewien sposób identyfikowane. Oczywiście można się cały czas posługiwać wskaźnikami i traktować manager zasobów tak jak manager pamięci (metody typu: `LoadResource`, `FreeResource`), jednakże wtedy nie mamy dostępu do konkretnego zasobu. Załóżmy, że mamy model, który korzysta z tekstury o nazwie `tekstura01.bmp`. Model zostaje wczytany do pamięci i teraz należy wczytać odpowiednią teksturę dla niego. Oczywiście tekstura zostanie wczytana ponieważ znamy jej nazwę i zostanie nawet umieszczona w managerze zasobów tekstur. Błędem będzie wyszukanie tej tekstury w managerze tekstur i przypisanie jej do modelu. Taką operację powinno się wykonać od razu podczas ładowania modelu, żeby nie wyszukiwać. Więc teoretycznie nie potrzeba nam klucza dla tekstury ponieważ model, który z niej korzysta ma już przypisany wskaźnik na nią. Ale, jeżeli wczytamy kolejny model, który również korzysta z tekstura01.bmp będzie musiał wczytać tą teksturę ponownie do pamięci. Powoduje to

stały czas ładowania modeli oraz duży narzut pamięciowy ponieważ zasoby zostaną powielone. Jeżeli zastosujemy klucze (na przykład kluczem może być nazwa tekstury) wtedy podczas ładowania modelu zostanie wyszukana tekstura, z której korzysta, w managerze tekstur, i jeśli taka będzie, zostanie przypisany jej wskaźnik do modelu. Co więcej, bardzo dobrym pomysłem podczas ładowania danych jest zbieranie statystyk na podstawie częstotliwości ładowania konkretnych zasobów. Jeżeli często jest wczytywany model model01.model to powinno zostać to w pewien sposób odnotowane w managerze aby nie ładował ponownie modelu, a tylko szybko zwrócił kopię już załadowanego. To samo tyczy się wybierania. Mogą powstać sytuacje, gdzie kilka razy pod rząd wybierany jest ten sam zasób. Warto więc zrobić sobie kopię ostatnio wybieranego klucza i podczas wybierania porównać i w razie czego zwrócić bez ponownego wyszukiwania. W najlepszym wypadku mamy jedną instrukcję warunkową, w najgorszym jedną instrukcję warunkową, jedno przypisanie i jedno wyszukanie.

Skoro już wiadomo, że klucze identyfikujące zasoby są ważne należy wybrać typ klucza. Najprościej było by używać napisów. Niestety niesie to za sobą konsekwencje. Po pierwsze, jeśli użyjemy klasy `std::string` to otrzymamy spory narzut pamięciowy oraz szybkościowy (`std::string` dynamicznie alokuje tablicę znaków). Jak się okazuje w wielu projektach wąskim gardłem okazują się właśnie `std::string`'i. Najwydajniejsze jest używanie liczb całkowitych.



Rysunek 2.3.1. Wykres przedstawiający różnicę między wstawianiem i wyszukiwaniem w mapie, w której kluczem był napis oraz liczba całkowita. Czas mierzony był w sekundach.

Powyższy wykres przedstawia porównanie wstawiania i wyszukiwania elementów z mapy. Do testu została użyta mapa z kluczem typu `std::string` oraz mapa z kluczem typu `unsigned long`. Jak widać szybkość wstawiania elementów niewiele się różni, natomiast wyszukiwanie jest znacząco szybsze w przypadku liczb. W takim razie powstaje kolejny problem. Mianowicie za pomocą liczb ciężko odwzorować napisy. Jeśli byśmy chcieli mieć manager tekstur to, jak wcześniej zostało napisane, bardzo przydatne są klucze napisowe. Jak się okazuje za pomocą liczb również można reprezentować napisy. Każdy napis należy przepuścić przez funkcję hash'ującą (mieszającą), która zwraca liczbę całkowitą. Szansa, że dla różnych napisów hash się powtórzy jest znikoma, w praktyce

pomijalna, a ewentualne powtórzenia można wychwycić podczas testów. Oczywiście do mieszania potrzeba odpowiedniego algorytmu. Przede wszystkim musi być szybki, a po drugie musi gwarantować minimalną ilość powtórzeń. W teście z rysunku 1 został użyty algorytm CRC. Obie mapy korzystały z tych samych napisów jako kluczy, tylko, że do drugiej mapy, przed wstawieniem lub wyszukaniem, napis był hash'owany, żeby uzyskać klucz. Mimo użycia algorytmu mieszania liczby-klucze okazały się o wiele szybsze.

Najprościej wyglądający manager może mieć następującą postać.

```
class CResource
{
	//...
};

class CResourceManager : public CSingleton<CResourceManager>
{
	public:

		typedef unsigned long 			Handle;
		typedef std::map<Handle, CResource*>	ResourceMap;

		CResourceManager()
		{
		}

		~CResourceManager()
		{
			ReleaseResources();
		}

		void ReleaseResources();

		void ReleaseResource(CResource* _pRes);
		void ReleaseResource(const std::string& _strName);
		void ReleaseResource(const Handle& _ulKey);

		CResource*  CreateResource(const std::string& _strName);
		CResource*  CreateResource(const Handle& _ulKey);

		CResource*  GetResource(const std::string& _strName);
		CResource*  GetResource(const Handle& _ulKey);


	private:

		ResourceMap	m_mResources;
};
```

Manager zasobów jest świetnym przykładem singleton'u ponieważ zasoby mogą być używane w różnych podsystemach (różne miejsca kodu programu). Klasa `CResource` jest przykładową klasą zasobu. Definicja klasy może być dowolna, w zależności od potrzeb projektu. W przykładzie będą używane zwykłe wskaźniki na zasoby. Dobrze zaprojektowana klasa zasobu powinna zapobiegać nieprawidłowemu usunięciu obiektu

(przeciążenie operatora delete). Licznik referencji może być zaimplementowany na poziomie klasy `CResource`, lub można do tego użyć inteligentnego wskaźnika.

W powyższym przykładzie mapa przechowuje wskaźniki na zasoby. Przechowywanie wskaźników w pojemnikach jest bardzo ważne, szczególnie w przypadku `std::vector`. Jeśli chcieli byśmy przechowywać obiekty zamiast wskaźników jest duża szansa na spadek wydajności aplikacji. Dzieje się to z tego powodu, że tablica musi dla siebie mieć zarezerwowany ciągły obszar pamięci. Jeśli tablica przechowuje obiekty dużych rozmiarów potrzeba będzie dużego wolnego bloku pamięci, żeby to wszystko pomieścić. Ze względu na fragmentację pamięci może się okazać, że takiego bloku nie ma dostępnego, wówczas system operacyjny zacznie używać pamięci wirtualnej a to spowoduje znaczny spadek wydajności. Wskaźniki natomiast są porozrzucane po całej pamięci (tam gdzie znajdzie się odpowiednio dużo miejsca aby je pomieścić), więc tablica o elementach rozmiaru wskaźnika ma dużo większe szanse na uzyskanie dla siebie odpowiedniego bloku pamięci.

Metoda `ReleaseResources` zwalnia wszystkie zasoby z mapy. Zwalnianie zasobu również jest zależne typowo od potrzeb projektu. Najprostsza wersja to po prostu usunięcie zasobów z pamięci i wyczyszczenie mapy. Takie zadanie może właśnie przypaść przeciążonym metodom `ReleaseResource`. Wersja przyjmująca wskaźnik na zasób może okazać się najbardziej kosztowna ponieważ oprócz zwolnienia pamięci obiektu należy go usunąć z mapy, ale żeby tego dokonać należy go odnaleźć, czyli w najgorszym przypadku przejść po wszystkich elementach pojemnika. Dużo szybciej będzie działać metoda przyjmująca napis lub liczbę, która jest uchwytem zasobu. Jak już wiadomo wyszukanie liczby w mapie jest szybkie, więc szybko otrzymamy miejsce w mapie z zasobem. Dzięki iterator'owi można szybko usunąć zasób z mapy.

Następnie, każdy manager zasobów powinien posiadać metodę tworzącą zasób. Tworzenie zasobu najczęściej odbywa się przez dynamiczną alokację pamięci dla obiektu i umieszczenie wskaźnika w pojemniku. Ważną rzeczą jest, aby sprawdzić czy obiekt o podanym kluczu znajduje się już w kontenerze. W zależności od implementacji można go nadpisać (uprzednio usuwając obiekt z pamięci) co jednak nie jest najlepszym pomysłem, gdyż może to spowodować załamanie się aplikacji, jeśli jakiś podsystem korzysta z tego zasobu. Najlepiej jest zwrócić 0 oraz informację o błędzie. Podczas tworzenia zasobu błędów może być klika. Na przykład brak wolnej pamięci lub powielenie klucza. W tym managerze głównym typem klucza jest napis, który jest zamieniany na liczbę. Jednakże manager nie powinien z góry narzucać algorytmu hash'ującego, dlatego należy dać możliwość używania samych uchwytów (liczb).

Oczywiście żaden manager pamięci nie może obyć się bez metody zwracającej wybrany zasób. W przykładzie takie metody są dwie. W praktyce ten model managera stanowi po prostu przysłonięcie operacji na mapie.

Teoretycznie klasa zasobu może być dowolna. Niemniej jednak warto by posiadała pewne przydatne funkcje. Przede wszystkim należy ustalić tworzenie zasobu. Można zaprzyjaźnić klasę zasobu z klasą managera i manager będzie tworzył i uzupełniał wszystkie pola obiektu zasobu. Da to taką korzyść, że programista nie będzie mógł „ręcznie" stworzyć zasobu, a tylko przez manager, więc nie zostanie popełniona żadna pomyłka. Z drugiej strony projektując bardziej obiektowo kod warto przerzucić

odpowiedzialność tworzenia zasobu do klasy zasobu, a managerowi pozostawić zarządzanie zwykłymi wskaźnikami. Daje to zaletę odseparowania dwóch, w praktyce, odrębnych podsystemów. Tak naprawdę manager zasobów nie powinien wiedzieć nic o zasobach, którymi zarządza. Oczywiście wszystko zależy od projektu i można tworzyć managery zasobów dla każdego z osoba, ale warto zastanowić się nad jedną, wspólną klasą dla każdego zasobu. Wystarczy, że klasę `CResourceManager` zrobimy klasą szablonową i już można tworzyć managery dla każdego typu zasobów.

W grach komputerowych typów zasobów może być (i w praktyce jest) wiele. Warto się zastanowić nad architekturą systemu zasobów. Można zrobić jedną klasę managera dla wszystkich zasobów w projekcie. Każdy zasób dziedziczy z bazowej klasy zasobu. Dzięki temu mechanizmowi klasa może trzymać wskaźniki na każdy typ zasobu nie wiedząc jaki dokładnie jest to zasób. Pobierając (dodając lub usuwając) wybrany zasób rzutujemy go na odpowiedni typ (w zależności od operacji: typ bazowy lub typ docelowy). Żeby programista nie musiał wiedzieć o architekturze managerów zasobów warto wówczas napisać klasy przysłaniające pewne operacje głównego managera.

```
class IResource
{
};

class CTexture : public IResource
{
	//...
};

class CTextureManager : public CSingleton<CTextureManager>
{
	public:

		CTextureManager(CResourceManager* _pResMgr) :
		m_pResMgr( _pResMgr ) {}

		CTexture*   GetTexture(const std::string& _strName)
		{ return (CTexture*)m_pResMgr->GetResource( _strName ); }

	private:

		CResourceManager* m_pResMgr;
};
```

Ten bardzo prosty przykład z grubsza pokazuje jak łatwo można zwiększyć użyteczność podsystemów. Klasa `CResourceManager` przechowuje wskaźniki typu `IResource` (interface zasobu), w swoim konstruktorze tworzy singleton CTextureManager oraz innych managerów. Pojawia się tutaj pewien problem. Jeżeli wszystkie zasoby (można liczyć je w setkach) trzymamy w jednym pojemniku to czas wyszukiwania ulega wydłużeniu, oraz istnieje większa szansa powtórzenia się klucza (hash'a). Dodatkowo, bardzo trudno będzie usunąć zasoby danego typu. Manager nie wie, jakiego typu zasoby przechowuje. Co prawda klasa `IResource` może mieć identyfikator zasobu, który będzie dawał informacje o typie, ale nadal problem pozostaje w tym, że możemy chcieć usunąć

wszystkie zasoby danego typu, i aby to zrobić, należy przejść po całej mapie i sprawdzać typ każdego zasobu. Nie jest to wydajny sposób.

Rozwiązaniem mogą być grupy zasobów. Zasoby mogą należeć do danej grupy (na przykład dźwięki) i chcąc usunąć wszystkie zasoby danego typu usuwamy po prostu całą grupę. Implementacji tego pomysłu może być klika. Można stworzyć mapę map (pierwsza mapa posiada klucz np. `std::string`, który jest nazwą grupy. Druga mapa to mapa zasobów danej grupy), lub mapę managerów zasobów. To drugie rozwiązanie jest bardziej czytelne i w praktyce prostsze w implementacji. W praktyce otrzymujemy managera managera zasobów. Grupy zasobów można dynamicznie tworzyć i usuwać. Tworząc grupę tworzymy nowy manager zasobów. Generalnie rzecz biorąc to rozwiązanie zdaje egzamin, jednak w pewnym momencie tworzenia gry programista dochodzi do momentu, kiedy chciał by za jednym zamachem usunąć nie tyle całą grupę, co konkretną część grupy zasobów. Załóżmy, że w grze mamy miasto. Gracz wchodzi do domu, wówczas ładują się zasoby wymagane do odwzorowania domu. Gdy gracz wychodzi, te zasoby powinny być usunięte. Usuwanie każdego zasobu z osoba prawdopodobnie spowoduje duży spadek wydajności (ze względu na dużą ilość operacji: znajdź zasób – usuń zasób), dlatego dużo lepiej dać możliwość tworzenia grup zasobów danego typu. To może zaprowadzić do stworzenia managera managera zasobów. Zastanówmy się czy wspólny manager zasobów jest naprawdę konieczny. Nie jest, ponieważ i tak będziemy korzystali z konkretnych managerów (tak jak w przykładzie `CTextureManager`) przysłaniających główny manager. Pisanie managera dla każdego typu zasobu może być żmudne, jednak szybko okazuje się, że typ zasobu nie ma wpływu na klasę, która nim zarządza. W związku z tym można napisać automatyczny manager zasobów, czyli taki (podobnie jak automatyczny singleton), z którego każda klasa będzie mogła odziedziczyć.

```
typedef unsigned long   ul32;
typedef unsigned int    u32;
typedef int             i32;

template<class _T_, class _MGR_CLASS_>
class TCResourceGroupManager;

template<class _T_, class _MGR_CLASS_>
class TCResourceGroup
{
      friend class TCResourceGroupManager<_T_, _MGR_CLASS_>;
      friend _MGR_CLASS_;

      public:

      typedef void (_MGR_CLASS_::*FuncPtr)
      (TCResourceGroup<_T_, _MGR_CLASS_>* _pGroup, _T_* _pResource);

      typedef _T_                   Resource;
      typedef _T_*                  ResourcePtr;
      typedef const ResourcePtr     ConstResPtr;
      typedef Resource&             ResourceRef;
      typedef const Resource&       ConstResRef;

      typedef map<ul32, ResourcePtr>           ResourceMap;
      typedef pair<ul32, ResourcePtr>          ResourcePair;
      typedef typename ResourceMap::iterator         ResourceIterator;
```

```
typedef typename ResourceMap::const_iterator   ResourceConstIterator;

typedef stack<ResourcePtr>    ResourceStack;

public:

      TCResourceGroup(_MGR_CLASS_* _pResGroupMgr, FuncPtr
      _OnResRemPtr, FuncPtr _OnResDesPtr, const ul32& _ulID,
      ResourceStack* _pResStack)
      {
            m_pResGroupMgr = _pResGroupMgr;

            m_OnResourceDestroyPtr = _OnResDesPtr;
            m_OnResourceRemovePtr = _OnResRemPtr;

            m_ulID = _ulID;

            m_uiResourceCount = 0;
            m_pLastUsedResource = 0;
            m_ulLastUsedID = 0;

            assert( _pResStack != 0 );
            m_psFreeResources = _pResStack;
      }

      virtual     ~TCResourceGroup()
      {
            _DestroyAllResources();
      }


      inline ul32 GetID() const
      {
            return m_ulID;
      }

      i32   AddResource(const ul32& _ulID, _T_* _pResource);

      inline void AddResource(ResourceIterator _Itr, ul32 _ulID, _T_*
      _pResource);

      int   RemoveResource(const ul32& _ulID);

      ResourcePtr SimpleRemoveResource(const ul32& _ulID);

      _T_*  GetResource(const ul32& _ulID);

      inline ResourceIterator GetResourceItr(const ul32& _ulID);

      inline bool ResourceExists(ResourceIterator _Itr, const ul32&
      _ulID);

      void  RemoveAllResources();

      inline u32  GetResourceCount() const
      { return m_uiResourceCount; }

      inline ResourceIterator GetBeginIterator()
      { return m_mResources.begin(); }

      inline ResourceIterator GetEndIterator()
```

```
        { return m_mResources.end(); }

protected:


        inline      void  _SetID(const ul32& _ulID)
        { m_ulID = _ulID; }

        void  _OnResourceRemove(_T_* _pResource)
        {
              CALL_MEMBER_FN_PTR(m_pResGroupMgr,
              m_OnResourceRemovePtr)(this, _pResource);
        }

        void  _OnResourceDestroy(_T_* _pResource)
        {
              CALL_MEMBER_FN_PTR(m_pResGroupMgr,
              m_OnResourceDestroyPtr)(this, _pResource);
        }


        i32 _BeginResourceCreate(const ul32& _ulID, _T_** _ppOutRes);

        void _EndResourceCreate(_T_* _pRes);

        inline ResourceMap& _GetResourceMap()
        { return m_mResources; }

        void  _DestroyAllResources();

        void _DestroyResource(_T_* _pResource);

        void _DestroyResource(const ul32& _ulID);

        inline ResourceStack& _GetFreeResourceStack()
        { return *m_psFreeResources; }

        inline void _AddFreeResource(_T_* _pRes)
        { m_psFreeResources->push(_pRes); }

        inline _T_* _GetFirstFreeResource() const
        { return m_psFreeResources->top(); }

        inline void _RemoveFreeResource()
        { m_psFreeResources->pop(); }

protected:

        ResourcePtr       m_pLastUsedResource;
        ResourceIterator  m_BeginCreateItr;
        ul32              m_ulBeginCreateID;
        bool              m_bCanEndCreate;
        ul32              m_ulLastUsedID;
        ResourceMap       m_mResources;
        ResourceStack*    m_psFreeResources;
        u32               m_uiResourceCount;

private:

        _MGR_CLASS_*      m_pResGroupMgr;
        FuncPtr           m_OnResourceRemovePtr;
```

```
        FuncPtr         m_OnResourceDestroyPtr;
        FuncPtr2        m_OnResourceCreatePtr;
        FuncPtr3        m_OnResourceCreateNewPtr;
        FuncPtr4        m_OnResourceCreateFromStackPtr;
        ul32            m_ulID;

};
```

Klasa `TCResourceGroup` jest managerem zasobów danego typu. Jest to klasa parametryzowana dwoma typami: typ zasobów oraz klasa docelowego managera zasobów (o tym później). Manager ten jest zaprzyjaźniony z głównym managerem zasobów oraz z docelowym managerem. Następnie zdefiniowane są typy danych używane w tej klasie. Wskaźniki na funkcje i konstruktor zostaną omówione później. Manager używa mapy jako pojemnika na zasoby oraz stosu. Przeznaczenie stosu zostanie omówione później.

Metoda `GetID` zwraca uchwyt tego managera. Uchwyt tworzony jest w `TCResourceGroupManager`. Metoda `AddResource` ma dwie przeciążone wersje. Pierwsza dodaje zasób (stworzony) do pojemnika używając konkretnego uchwytu (także uprzednio stworzonego). Druga wersja dodaje stworzony zasób do pojemnika używając uchwytu oraz konkretnego miejsca w mapie. Metoda `RemoveResource` usuwa zasób z managera. Metoda `SimpleRemoveResource` usuwa zasób tylko z pojemnika. Metoda `GetResource` zwraca zasób o podanym uchwycie. `GetResourceItr` zwraca miejsce w pojemniku zasobu o podanym uchwycie. `ResourceExists` zwraca prawdę jeśli zasób w podanym miejscu i o podanym uchwycie istnieje. `RemoveAllResources` usuwa wszystkie zasoby z managera. GetResourceCount zwraca ilość zasobów. `GetBeginIterator` zwraca pierwsze miejsce w pojemniku z zasobem. `GetEndIterator` zwraca ostatnie nieużywane miejsce (czyli bez zasobu). `_SetID` ustawia uchwyt grupy. `_OnResourceRemove`, `_OnResourceDestroy`, `_BeginResourceCreate` i `_EndResourceCreate` zostaną omówione później. `_GetResourceMap` zwraca pojemnik z zasobami. `_DestroyAllResources` to wewnętrzna metoda usuwania zasobów. `_DestroyResource` to wewnętrzna metoda usuwania zasobu.

A teraz szczegóły implementacji.

Konstruktor przyjmuje pewne parametry:

- Wskaźnik na docelową klasę managera zasobów. Usuwanie zasobów z managera dla każdego typu zasobów może odbywać się inaczej, dlatego używane są wskaźniki na funkcje. Te funkcje (metody) implementowane są w docelowym managerze. Jak łatwo zauważyć ten manager nie jest przystosowany do samodzielnego użytkowania. Należy stworzyć klasę managera zasobów (np. tekstur) i odziedziczyć z `TCResourceGroupManager` (który będzie zaraz omówiony).
- Wskaźnik na metodę usuwającą zasób. Usuwanie i niszczenie zasobu to dwie różne operacje. Niszczenie zasobu całkowicie go usuwa, natomiast usuwanie zasobu usuwa go tylko z pojemnika.
- Wskaźnik na metodę niszczącą zasób. Niszczenie zasobu może się odbywać na różne sposoby. Zależnie od implementacji, może to być usunięcie z pamięci (delete), usunięcie w managerze pamięci (`manager.free`), wywołanie konkretnej

metody, która usuwa zasób (`resource->release`) itp. Przedstawiony wyżej manager pamięci nic nie wie o zasobach. Nie wie jak je tworzyć i usuwać. To zadanie spoczywa na implementacji konkretnego managera.

- Uchwyt tej grupy zasobów. Główny manager zasobów tworzy grupy zasobów. Każda grupa to osobny manager, który jest przechowywany w pojemniku, więc posiada swój uchwyt. Uchwyt jest potrzebny ponieważ programista może pobrać i operować na konkretnej grupie.
- Ostatni parametr to wskaźnik na stos. Stos tworzony jest w głównym managerze. Ponieważ jest to manager konkretnego typu zasobów może być jeden stos. Stos przechowuje wolne zasoby. Domyślnie usunięcie zasobu wiąże się z usunięciem go z mapy i dodaniem wskaźnika do stosu wolnych zasobów. Jest to pewna optymalizacja, gdy nie jest używany manager pamięci, polegająca na wykorzystaniu wcześniej zarezerwowanej pamięci. Dzięki temu nie trzeba wywoływać ponownie operatora `new` dla nowego zasobu. Stos jest wspólny dla wszystkich grup zasobów.

```
i32	AddResource(const ul32& _ulID, _T_* _pResource)
{
	ResourceIterator Itr = m_mResources.lower_bound(_ulID);

	if(	Itr != m_mResources.end() &&
		!(m_mResources.key_comp()(_ulID, Itr->first)) )
	{
		return 0;
	}
	else
	{
		m_mResources.insert(Itr, ResourceMap::value_type(_ulID,
		_pResource));
		return 1;
	}

	return 0;
}
```

W tej metodzie (oraz w wielu innych) zastosowana jest pewna sztuczka wydajnościowa (w przykładzie wykorzystany jest pojemnik `std::map`). Na ogół w metodach tego typu wykorzystywane są metody `find` oraz insert. Metoda `find` sprawdza czy element o danym kluczu już się znajduje, jeśli się znajduje należy zwrócić jakiś błąd, natomiast jeśli się nie znajduje to go dodajemy. Takie rozwiązanie jest intuicyjne, ale niewydajne. Metoda insert również wykonuje wyszukiwanie, więc mamy dwa wyszukiwania w przypadku, gdy `find` nie odnajdzie elementu. Skoro metoda insert również wykonuje wyszukiwanie (ponieważ mapa nie zezwala na duplikowanie kluczy) więc wystarczy użyć tylko jej do wstawiania, a metodę `find` pominąć. Insert zwraca parę, której drugim elementem jest pole logiczne, które jest ustawiane na 1 jeśli element został wstawiony, lub 0 jeśli element o tym kluczu już istnieje. Lepszym rozwiązaniem jest użycie metody `lower_bound`, która oblicza miejsce w mapie, na którym powinien znaleźć się element o podanym kluczu, następnie instrukcja warunkowa sprawdza czy taki element już w mapie jest. Jeśli nie to jest dodawany na wcześniej obliczone miejsce, dzięki czemu nie ma dodatkowego wyszukiwania. W przypadku, gdy element już się znajduje w mapie może być on podmieniony (w zależności od implementacji).

```
inline void AddResource(ResourceIterator _Itr, ul32 _ulID, _T_* _pResource)
{
	m_mResources.insert(_Itr, ResourceMap::value_type(_ulID,_pResource));
}
```

Metoda wstawia element o podanym kluczu w konkretne miejsce w mapie (zazwyczaj wyliczone przez metodę `lower_bound`).

```
i32	RemoveResource(const ul32& _ulID)
{

	if(m_ulLastUsedID == _ulID)
	{
		m_uiResourceCount--;

		m_mResources.erase(m_ulLastUsedID);
		_OnResourceRemove(m_pLastUsedResource);
		m_pLastUsedResource = 0;
		m_ulLastUsedID = 0;

		return 1;
	}


	ResourceIterator Itr = GetResourceItr(_ulID);
	if( ResourceExists(Itr, _ulID) )
	{
		m_uiResourceCount--;

		_T_* pResource = Itr->second;
		m_mResources.erase(Itr);
		_OnResourceRemove(pResource);
		return 1;
	}

	return 0;
}
```

Metoda usuwająca element o podanym uchwycie. Jest to główna metoda usuwająca pojedynczy element w managerze. Pierwsze porównanie sprawdza czy używany jest ostatnio użyty element. Jeśli tak, licznik zasobów jest zmniejszany, element jest usuwany z mapy oraz wywoływany jest wskaźnik na metodę (zaimplementowany w docelowym managerze) z ostatnio użytym zasobem (optymalizacja zapobiegająca zbędnemu wyszukiwaniu). Jeżeli usuwany zasób nie był ostatnio użyty pobierany jest iterator (miejsce) zasobu i sprawdzane jest czy klucz jest poprawny (czyli, czy zasób istnieje). Jeśli istnieje to zmniejszany jest licznik ilości zasobów, zasób jest usuwany z pojemnika oraz wywoływany jest wskaźnik na metodę. W przypadku, gdy zasób nie istnieje zwracana jest wartość 0.

Metoda `SimpleRemoveResource` jest bardzo podobna do powyższej, ale nie wywołuje wskaźnika na metodę, czyli tylko usuwa zasób z pojemnika. Należy uważać na używanie

jej, aby nie utracić wskaźnika i nie spowodować wycieku pamięci. Metoda zwraca wskaźnik na zasób, dzięki czemu można go bezpiecznie usunąć z pamięci.

```
_T_*  GetResource(const ul32& _ulID)
{
	if(m_ulLastUsedID == _ulID)
	{
		return m_pLastUsedResource;
	}

	ResourceIterator Itr = m_mResources.lower_bound(_ulID);

	if(	Itr != m_mResources.end() &&
		!(m_mResources.key_comp()(_ulID, Itr->first)) )
	{
		m_pLastUsedResource = Itr->second;
		m_ulLastUsedID = _ulID;
		return Itr->second;
	}

	return 0;
}
```

Podstawowa metoda pobierania zasobu z managera. Jeżeli chcemy pobrać ostatnio użyty zasób zostanie on natychmiast zwrócony. Jeżeli jest to inny zasób zostają wykonane operacje podobne do tych przy wstawianiu. Sprawdzane jest czy element o podanym kluczu istnieje, jeśli tak to ustawiany jest ten zasób jako ostatnio użyty i jest on zwracany. Jeśli taki zasób nie istnieje zwracane jest 0.

```
inline ResourceIterator GetResourceItr(const ul32& _ulID)
{
	return m_mResources.lower_bound(_ulID);
}

inline bool ResourceExists(ResourceIterator _Itr, const ul32& _ulID)
{
	return (	_Itr != m_mResources.end() &&
			!(m_mResources.key_comp()(_ulID, _Itr->first)) );
}
```

Obiekt metody to zwykłe przysłonięcia operacji na mapach w celu zwiększenia czytelności. Jak widać metoda `ResourceExists` jest uzależniona od `GetResourceItr` (konkretnie od iteratora jaki ta metoda zwraca).

```
void  RemoveAllResources()
{
	ResourceIterator Itr = m_mResources.begin();
	for(Itr; Itr != m_mResources.end(); ++Itr)
	{
		_OnResourceRemove(Itr->second);
	}
```

```
	m_mResources.clear();
	m_uiResourceCount = 0;
	m_ulLastUsedID = 0;
	m_pLastUsedResource = 0;
}
```

Metoda usuwająca wszystkie zasoby z pojemnika. Najpierw następuje iteracja po całej mapie , i dla każdego elementu wywoływany jest wskaźnik na metodę (domyślnie usuwany element nie jest usuwany z pamięci a tylko czyszczony i dodawany do stosu wolnych zasobów).

```
void	_OnResourceRemove(_T_* _pResource)
{
	CALL_MEMBER_FN_PTR(m_pResGroupMgr, m_OnResourceRemovePtr)
	(this, _pResource);
}

void	_OnResourceDestroy(_T_* _pResource)
{
	CALL_MEMBER_FN_PTR(m_pResGroupMgr, m_OnResourceDestroyPtr)
	(this, _pResource);
}
```

Obie metody w praktyce wywołują wskaźniki na metody zaimplementowane w docelowym managerze. Wykorzystywane jest tutaj macro

---

```
#define CALL_MEMBER_FN_PTR(object,ptrToMember) (object)->*(ptrToMember))
```

Macro to wywołuje wskaźnik metody obiektu.

```
i32	_BeginResourceCreate(const ul32& _ulID, _T_** _ppOutRes)
{
	(*_ppOutRes) = 0;

	m_BeginCreateItr = m_mResources.lower_bound(_ulID);

	if(	m_BeginCreateItr != m_mResources.end() &&
		!(m_mResources.key_comp()(_ulID, m_BeginCreateItr->first)) )
	{
		m_bCanEndCreate = false;
		return 0;
	}

	m_ulBeginCreateID = _ulID;
	m_bCanEndCreate = true;

	if( !m_psFreeResources->empty() )
	{
```

```
            (*_ppOutRes) = m_psFreeResources->top();
            m_psFreeResources->pop();
      }

      return 1;
}
```

Ta metoda używana jest przed dodaniem zasobu. Najpierw sprawdzane jest czy element o podanym kluczu istnieje w pojemniku, jeśli istnieje zwracane jest 0 (informacja dla metody tworzącej zasób w docelowym managerze. Jeśli zasób istnieje może go zwrócić lub zwrócić błąd). Następnie sprawdzane jest, czy jest jakiś wolny wskaźnik (czyli wcześniej usunięty zasób), jeśli jest to zostaje on zwrócony jako drugi argument. Jeśli nie ma, zwracana jest wartość 0 w drugim argumencie (wówczas metoda tworząca zasób w docelowym managerze wie, czy ma tworzyć nowy obiekt w pamięci, czy może wykorzystać poprzednio utworzony. W przypadku korzystania z własnego managera pamięci można pominąć używanie stosu).

```
void _EndResourceCreate(_T_* _pRes)
{
      assert(m_bCanEndCreate == true);

      m_mResources.insert(m_BeginCreateItr,
      ResourceMap::value_type(m_ulBeginCreateID, _pRes));

      m_BeginCreateItr = m_mResources.end();
      m_ulBeginCreateID = 0;
      m_bCanEndCreate = false;
}
```

Ta metoda używana jest razem z metodą `_BeginResourceCreate`, która ustawia iterator aktualnie stworzonego zasobu oraz jego uchwyt. Używa tych wartości, aby wstawić je do pojemnika na odpowiednie miejsce. Nie jest sprawdzane, czy element taki już istnieje ponieważ sprawdza to poprzednia metoda. Obie metody są chronione, aby nie można ich było użyć w nieodpowiedni sposób.

```
void  _DestroyAllResources()
{
      while(!m_psFreeResources->empty())
      {
            _OnResourceDestroy(m_psFreeResources->top());
            m_psFreeResources->pop();
      }

      ResourceIterator Itr = m_mResources.begin();
      for(Itr; Itr != m_mResources.end(); ++Itr)
      {
            _OnResourceDestroy(Itr->second);
      }

      m_mResources.clear();
```

```
	m_uiResourceCount = 0;
	m_pLastUsedResource = 0;
	m_ulLastUsedID = 0;
}
```

Metoda niszcząca wszystkie zasoby. Na początek niszczone są wszystkie wolne zasoby (używany jest odpowiedni wskaźnik na metodę). Następnie niszczone są wszystkie aktualnie używane zasoby.

```
void _DestroyResource(_T_* _pResource)
{
	if(m_pLastUsedResource == _pResource)
	{
		m_pLastUsedResource = 0;
		m_ulLastUsedID = 0;
	}

	_OnResourceDestroy(_pResource);
}

void _DestroyResource(const ul32& _ulID)
{
	if(m_ulLastUsedID == _ulID)
	{
		m_ulLastUsedID = 0;
		_OnResourceDestroy(m_pLastUsedResource);
		m_pLastUsedResource = 0;
	}
}
```

Obie metody niszczą wybrany zasób na dwa sposoby, albo podając wskaźnik albo uchwyt.

Teraz przyjrzyjmy się bliżej głównej klasie managera zasobów.

```
template<class _T_, class _MGR_CLASS_ >
class TCResourceGroupManager :
public virtual TCResourceManager<TCResourceGroup<_T_, _MGR_CLASS_> >
{
	public:

		typedef _T_			Resource;
		typedef TCResourceGroup<_T_, _MGR_CLASS_>	ResourceGroup;
		typedef ResourceIterator			GroupIterator;

	protected:

		typedef typename ResourceGroup::ResourceStack
		BaseResourceStack;

	public:

		TCResourceGroupManager()
		{
```

```
        }

        virtual ~TCResourceGroupManager()
        {
        }

        inline i32 AddGroup(const ul32& _ulGroupID,
        ResourceGroup* _pResGroup)
        {
              return AddResource(_ulGroupID, _pResGroup);
        }

        i32 RemoveGroup(const ul32& _ulGroupID);

        ResourceGroup* GetGroup(const ul32& _ulGroupID);

        i32 RemoveResource(const ul32& _ulResourceID,
        const ul32& _ulGroupID);

        i32 RemoveResource(const ul32& _ulResourceID);

        _T_* GetResource(const ul32& _ulResourceID);

        _T_* GetResource(const ul32& _ulResourceID,
        const ul32& _ulGroupID);


        i32 MoveResource(const ul32& _ulResourceID, const ul32&
        _ulDstGroupID, const ul32& _ulSrcGroupID)


        i32 MoveResources(const ul32& _ulDstGroupID, const ul32&
        _ulSrcGroupID)

        void RemoveAllResources();

protected:

        inline void _AddFreeResource(_T_* _pRes)
        {
              m_sFreeBaseResources.push( _pRes );
        }

        inline void _AddFreeGroup(ResourceGroup* _pGroup)
        {
              m_sFreeResources.push( _pGroup );
        }

        GroupIterator _GetGroup(cul32& _ulGroupID);

        bool _GroupExists(const GroupIterator& _Itr,
        ul32& _ulGroupID);

        virtual void _OnGroupRemove(ResourceGroup* _pResGroup)
        {
              delete(_pResGroup);
        }

        virtual void _OnGroupDestroy(ResourceGroup* _pResGroup)
        {
              delete(_pResGroup);
```

```
		}

	private:

		void _OnResourceRemove(ResourceGroup* _pResGroup)
		{
			_OnGroupRemove(_pResGroup);
		}

		void _OnResourceDestroy(ResourceGroup* _pResGroup)
		{
			_OnGroupDestroy(_pResGroup);
		}

	protected:
		BaseResourceStack m_sFreeBaseResources;
};
```

Klasa `TCResourceGroupManager` dziedziczy z klasy `TCResourceManager`, która jest bardzo podobna do klasy `TCResourceGroup`, dlatego nie zostanie opisana. Większość metod to przysłonięcie metody klasy bazowej, zmieniona jest tylko nazwa.

Metoda `RemoveResource` ma dwie wersje. Pierwsza usuwa zasób o uchwycie podanym jako pierwszy parametr z grupy o uchwycie podanej jako drugi parametr. Druga wersja metody usuwa zasób o uchwycie podanym jako parametr. Metoda ta przeszukuje wszystkie grupy w poszukiwaniu danego uchwytu. Zasób zostanie usunięty z pierwszej grupy w jakiej został odnaleziony (to znaczy, że jeżeli w innych grupach są zasoby o takim samym uchwycie nie zostaną one usunięte. Oczywiście mogą być, zależnie od implementacji). Metody `GetResource` działają podobnie do `RemoveResource`, z tym, że oczywiście zamiast usuwać zwracają zasób. Metoda `MoveResource` przenosi zasób z jednej grupy do drugiej (dlatego ważne jest, żeby aplikację projektować tak, by uchwyty zasobów się nie powtarzały), natomiast `MoveResources` przenosi wszystkie zasoby z jednej grupy do drugiej. Metoda `RemoveallResources` usuwa wszystkie zasoby ze wszystkich grup. Metody `_OnGroupRemove` oraz `_OnGroupDestroy` służą do usuwania i niszczenia grup zasobów. Jak się niebawem okaże tworzenie grupy zasobów programista musi sam zaimplementować, dlatego istnieje możliwość nadpisania tych metod. `_OnResourceRemove` i `_OnResourceDestroy` to z kolei nadpisania metod z bazowego managera zasobów (z którego ten dziedziczy i właśnie z tych metod w praktyce korzysta by poprawnie usuwać grupy, które faktycznie są dla niego zasobami). Szczegóły implementacji są bardzo podobne do tych z poprzedniego managera, więc zostaną pominięte.

W skrócie rzecz ujmując klasa `TCResourceGroupManager` zarządza zasobami, którymi są klasy TCResourceGroup. Ponadto klasa `TCResourceGroupManager` dziedziczy z klasy `TCResourceManager`, która jest bardzo podobna do klasy `TCResourceGroup` (wyjątek stanowi brak wskaźników na funkcje oraz stos wolnych zasobów. Dodatkowo klasa bazowa posiada dwie wirtualne metody, które należy są odpowiednio nadpisane w managerze zasobów – usuwają i niszczą grupę zasobów).

Przyjrzyjmy się teraz implementacji przykładowego managera skryptów.

```
class CScriptManager :
public TCSingleton<CScriptManager>,
public TCResourceGroupManager<CScript, CScriptManager>
{

	public:


				CScriptManager();
				~CScriptManager();

		ResourceGroup*	CreateGroup(const std::string& _strName);

		CScript*	CreateScript(const std::string& _strName,
				const std::string& _strGroup);

	inline CScript*	GetScript(const std::string& _strName,
				const std::string& _strGroupName)
				{
					return GetResource(CHash::GetCRC(_strName),
						CHash::GetCRC( _strGroupName ) );
				}

	inline CScript*	GetScript(const std::string& _strName)
				{
					return GetResource(CHash::GetCRC(_strName));
				}

	private:


		ResourceGroup*	_GetOrCreateGroup(u32& _ulGroupID);

		void	_OnResourceRemove(ResourceGroup* _pGroup,
			CScript* _pScript);

		void	_OnResourceDestroy(ResourceGroup* _pGroup,
			CScript* _pScript);

};
```

Klasa dziedziczy z singleton'u oraz z bazowej klasy managera zasobów. Jedyne co tak naprawdę musi zrobić programista to napisać metody `_OnResourceRemove` i `_OnResourceDestroy`. Te dwie metody przyjmują dwa parametry (dlatego, że konstruktor managera grupy zasobów – `TCResourceGroup` – w konstruktorze przyjmuje dwa wskaźniki na funkcje o właśnie takich parametrach) wskaźnik na grupę zasobów oraz wskaźnik na zasób. Podczas usuwania lub niszczenia zasobu warto czasem wiedzieć do jakiej grupy należy.

Metoda pozostałe metody stanowią jedynie przysłonięcie metod bazowej klasy. Różnią się jedynie tym, że przyjmują bardziej czytelny dla programisty argument – napis i zamieniają go na liczbę – format kluczy managera zasobów). Klasa `CHash` dostarcza metodę, która zamienia napis na hash daną metodą.

Nową metodą jaką można dopisać jest metoda do tworzenia zasobu. Może być to metoda raczej niskopoziomowa, gdyż managery zasobów przede wszystkim ładują je z dysku a nie tworzą. Metoda `CreateScript` tworzy obiekt typu `CScript` o konkretnej nazwie i dodaje go do grupy o nazwie podanej w drugim argumencie. Metoda prywatna `_GetOrCreateGroup` jest to metoda pomocnicza przydatna podczas tworzenia zasobu. Metoda `CreateScript` może również stworzyć grupę dla danego zasobu, aby nie było konieczne wywoływanie osobnej metody do tworzenia grupy, dlatego korzysta z metody `_GetOrCreateGroup`, która tworzy lub zwraca grupę (jeśli już taka istnieje).

Przypatrzmy się bliżej implementacji niektórych metod:

```
void OnResourceRemove(ResourceGroup* _pGroup, CScript *_pScript)
{
	_pScript->_ClearResource();
	_AddFreeResource( _pScript );
}

void OnResourceDestroy(ResourceGroup* _pGroup, CScript* _pScript)
{
	delete _pScript;
}
```

Metoda `OnResourceRemove` wywoływana jest podczas usuwania zasobu (`RemoveResource`). Ta implementacja czyści skrypt jego wewnętrzną metodą oraz używa metody bazowego managera zasobów, aby dodać go do puli zasobów gotowych do ponownego użycia. Metoda `OnResourceDestroy` po prostu usuwa obiekt z pamięci. Należy pamiętać, że obie metody są wywoływane tylko (i tylko tam być wywoływane powinny) w bazowym managerze podczas usuwania lub niszczenia zasobu (w obu przypadkach są usuwane z pojemnika).

```
CScript* CreateScript(  const std::string& _strName,
                        const std::string& _strGroupName)
{
	ul32 ulScriptID = CHash::GetCRC( _strName );
	ul32 ulGroupID = CHash::GetCRC( _strGroupName );

	ResourceGroup* pGroup = _GetOrCreateGroup( ulGroupID );
	CScript* pScript = 0;
	if( pGroup->_BeginResourceCreate( ulScriptID, &pScript ) == 1 )
	{
		if( !pScript )
		{
			pScript = new CScript();
		}

		pScript->_SetHandle( ulScriptID );
		pScript->_SetName( _strName );
		pScript->Init();

		pGroup->_EndResourceCreate( pScript );
		return pScript;
	}
```

```
	return 0;
}
```

Ta metoda tworzy skrypt i dodaje go do pojemnika. Najpierw obliczane są hash'e. Następnie tworzona jest i/lub zwracana grupa, do której ten skrypt ma należeć. Następnie zasób jest przygotowywany to stworzenia – sprawdzane jest czy jest zasób gotowy do ponownego użytku. Jeśli jest (czyli `pScript != 0`) pola zasobu są po prostu ustawiane na nowe. Jeśli nie ma zasobu należy go stworzyć. W przykładzie tworzony jest zasób dynamicznie w pamięci (potrzebny jest wskaźnik) i na koniec dodawany jest do puli zasobów (`_EndResourceCreate`).

```
ResourceGroup* _GetOrCreateGroup(ul32& _ulGroupID)
{
	ResourceIterator Itr = GetResourceItr( _ulGroupID );
	if( this->ResourceExists( Itr, _ulGroupID ) )
	{
		return Itr->second;
	}

	ResourceGroup* pGroup = 0;
	if( this->_BeginResourceCreate( _ulGroupID, &pGroup ) == 1 )
	{
		if( !pGroup )
		{
			pGroup = new RESOURCE_GROUP_CONSTRUCTOR(
					CScriptManager,
					_ulGroupID );
		}

		this->_EndResourceCreate( pGroup );
		return pGroup;
	}

	return 0;
}
```

Ta metoda korzysta z wielu metod bazowego managera. Najpierw pobierany jest zasób (obiekt zarządzający grupą zasobów), jeśli istnieje, jest zwracany. Jeśli nie należy go stworzyć. W tym celu używana jest metoda taka sama jak w `CreateScript` tylko, że dla innego typu zasobów (dla grupy zasobów), jeśli nie ma wolnej grupy (w zależności od implementacji, domyślnie nigdy nie będzie wolnej grupy na stosie ponieważ usunięcie grupy jest równoznaczne z jej zniszczeniem. Można to łatwo nadpisać), należy nową stworzyć. Używane jest tutaj specjalne macro ponieważ konstruktor grupy zasobów posiada wiele argumentów.

```
#define RESOURCE_GROUP_CONSTRUCTOR( _className, _ulGroupID )\
ResourceGroup(	this, &_className::_OnResourceRemove,\
		&_className::_OnResourceDestroy,\
		(_ulGroupID), &m_sFreeBaseResources )
```

Pierwszym argumentem konstruktora grupy zasobów jest wskaźnik na obiekt managera docelowego (w tym przypadku `CScriptManager`) ponieważ jest on potrzebny do wywoływania wskaźników na funkcje (`_OnResourceRemove`, `_OnResourceDestroy`). Następnie przekazywane są te wskaźniki do konstruktora, który ustawia je jako pola prywatne. Jak widać ważne są nazwy tych wskaźników (nazwy metod), dlatego w docelowym managerze mają takie nazwy a nie inne (ze względu na użycia macra, które z góry narzuca te nazwy). Kolejne dwa, ostatnie, argumenty to uchwyt grupy (hash nazwy) oraz stos wolnych zasobów (współdzielony przez wszystkie grupy).

Jak widać, małym nakładem pracy można osiągnąć funkcjonalny manager zasobów. Klasy bazowe `TCResourceGroupManager` oraz `TCResourceManager` (odpowiednik `TCResourceGroup`) oprócz korzystania z uchwytów mogły by korzystać również z napisów (jako kluczy). W tym przykładzie nie zostało to zaimplementowane ponieważ w zależności od aplikacji każdy może używać innego typu klas napisów. Jak już wcześniej zostało napisane `std::string` nie jest zbyt wydajną klasą więc narzędzia nie powinny narzucać używania jej. Generalnie wszystkie metody bazowego managera powinny zostać przysłonięte wersjami z napisami w docelowym managerze.

Używanie napisów jako uchwytów niesie ze sobą pewien problem – w tej samej grupie nie może być dwóch zasobów o tej samej nazwie. Jeśli korzystamy z plików (np. tekstury) mogą one posiadać te same nazwy, ale w różnych katalogach. Jeżeli postanowimy, że każdy katalog na dysku stanowi osobną grupę zasobów w managerze to problemu nie będzie, jednakże jest to nienajlepsze rozwiązanie. Załóżmy, że mamy model posiadający kilka tekstur. Model taki posiada jedynie nazwy tych tekstur i nie wie nic o ich położeniu czy grupie do jakiej należy. Wobec tego, podając wiele grup zasobów wyszukanie odpowiedniej tekstury zajmie więcej czasu. Zarządzanie wieloma grupami też może nie być łatwe ponieważ czasami chcemy dostęp do konkretnego zasobu a możemy nie wiedzieć w jakiej grupie się znajduje. Mniejsza ilość grup (teoretycznie grupy zasobów powinny być tworzone w zależności od ich przeznaczenia, np. interface dla interfejsu użytkownika, geometry dla tekstur modeli itp.) jest łatwiejsza w późniejszym użytkowaniu, ale wymaga by w całym drzewie katalogów zasobów projektu nie znalazły się dwa pliki o tych samych nazwach.

Zarządzanie zasobami w programowaniu gier jest rzeczą ważną. Napisanie dobrego managera może okazać się nie takie łatwe jak na pierwszy rzut oka wygląda. Sama klasa zasobu może już przysporzyć problemów. W przedstawionym powyżej managerze celowo nie został przedstawiony żaden model klasy zasobu ponieważ ten manager w założeniu ma nic nie wiedzieć na temat obiektów, którymi zarządza (czyli ma jak najbardziej uprościć korzystanie z niego). Jak łatwo zauważyć korzystanie ze zwykłych wskaźników może być bardzo niewygodne. Po pierwsze pewne podsystemy mogą korzystać z zasobów, które inne podsystemy właśnie zwolniły. Klasa zasobu powinna w jakiś sposób zliczać referencje. Pomocne mogą się tutaj okazać liczniki referencji. Niestety, aby zwiększyć bezpieczeństwo kodu należy zwiększyć integrację zasobu z managerem. Manager powinien wtedy wiedzieć, że zasób posiada przynajmniej taką metodę jak `ReadyForRemove` (metoda, która zwraca 1 lub 0 w zależności czy są jakieś referencje na zasób czy nie).

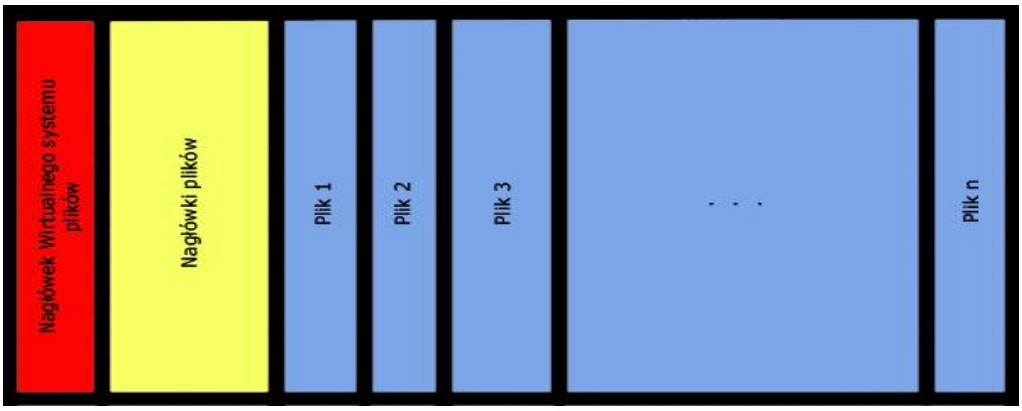
Taki manager może być wykorzystywany do zarządzania nie tylko plikami odczytywanymi z dysku, ale także obiektami tworzonymi w pamięci (np. klasa postaci). Nie wszystkie te obiekty muszą mieć swoje nazwy, dlatego warto wyposażyć się w system generujący unikalne identyfikatory. Najprościej jest używać licznika liczb całkowitych, dzięki temu w łatwy sposób można uzyskać miliardy unikalnych kluczy zanim wyczerpie się limit. W większości przypadków takie rozwiązanie wystarcza. Oczywiście gra komputerowa to program, który powinien móc działać bezbłędnie nawet przez kilka dni. W tak długim czasie może być tworzonych i usuwanych bardzo wiele zasobów, dlatego lista wolnych kluczy jest bardzo przydatna w tym przypadku (może być to stos taki jak w przypadku stosu wolnych zasobów). W praktyce, jeszcze dużo czasu minie zanim gry komputerowe będą mogły jednocześnie mieć załadowanych kilka miliardów tekstur lub innych zasobów tego samego typu, a nawet jeśli będzie to możliwe to zakres dostępnych liczb też będzie dużo większy. Można powiedzieć, że nigdy nie będzie możliwe załadowanie zasobów tego samego typu w ilości odpowiadającej maksymalnej wartości liczby całkowitej dostępnej na danym procesorze. Dlatego właśnie wyczerpaniem się tego typu identyfikatorów nie powinniśmy się martwić. Większym problemem może być to, ze programista tworząc własny zasób może również wpaść na tworzenie kluczy jako liczb, wówczas aplikacja może korzystać ze złych zasobów. Automatyczne klucze powinny mieć nieco bardziej wyrafinowaną formę. Można nadawać im jakiś unikalny prefiks np. „_moja_tekstura_01”.

Managery zasobów można łatwo połączyć z managerami pamięci. Zamiast używać stosu wolnych zasobów warto skorzystać z listy wolnych bloków pamięci w managerze pamięci. Ponadto manager zasobów wie ile jest dostępne wolnej pamięci dla zasobów i nie utworzy więcej niż oferuje dostępna pula. Łączenie ze sobą różnych mechanizmów to bardzo użyteczna rzecz, dlatego muszą być one dobrze zaprojektowane. Korzystanie z takich narzędzi znacznie przyspiesza proces tworzenia aplikacji.

2.4. Wirtualny system plików

Wąskim gardłem każdej aplikacji jest odczyt danych z dysku twardego. W dawniejszych czasach, gdy nie było jeszcze pojemnych dysków twardych, gry odczytywały dane z płyt CD. W dzisiejszych czasach wszystkie te dane są kopiowane na dysk twardy i nie ma już potrzebny odczytywania z płyt (co jest najwolniejszą operacją jeśli chodzi o odczyt). Dyski twarde nie są tak szybkie jak pamięć RAM. Dostęp do danych jest znacznie wydłużony a gry wymagają odczytu wielu ilości danych takich jak: tekstury, modele 3D, dźwięki, skrypty itp. Zasoby te są bardzo często odczytywane do pamięci RAM i zwalniane, kiedy należy odczytać następne (gra nie powinna zająć całej dostępnej pamięci RAM komputera). Niektóre gry odczytują cały poziom za jednym razem, inne odczytują poziom w trakcie gry. Za jednym razem odczytać można np. wnętrze domu, które jest niewielkie, nie ma dużej ilości obiektów, natomiast gdy gracz znajduje się na zewnątrz np. w dużym mieście, prawdopodobnie nie będzie możliwe odczytanie tego za jednym razem (nie będzie tyle dostępnej pamięci), dlatego należy odczytać tylko te modele, tekstury, dźwięki itp., które znajdują się w najbliższym widocznym otoczeniu gracza, a następne doczytywać w miarę jak gracz będzie się poruszał po przestrzeni. Jak łatwo teraz zauważyć szybkość odczytu danych z dysku jest bardzo ważna. Nie można pozwolić na sytuację, gdy co parę kroków jakie zrobi gracz będzie mu się pokazywał napis „Ładowanie” i gra będzie się zatrzymywała. Poniekąd niektóre gry tak właśnie robią, ale odczyt danych jest wtedy robiony w osobnym wątku, więc gra się nie zacina na czas ładowania. Odczytując pliki z dysku jesteśmy zdani na szybkość odczytu tego pliku przez system operacyjny (oczywiście szybkość samego dysku twardego też jest ważna, ale jest to rzecz pomijalna, gdyż należy przyjąć, że gracz ma najwolniejszy dysk), a dokładniej, przez szybkość systemu plików systemu operacyjnego.

Dobrym pomysłem na szybki odczyt danych z dysku jest stworzenie własnego systemu plików. Taki system nazywa się wirtualnym systemem plików (VFS – Virtual File System) i stanowi pomost między systemem plików system operacyjnego oraz aplikacji. Idea wirtualnego systemu plików polega na stworzeniu jednego pliku, którego zawartość stanowią inne pliki. Struktura takiego systemu plików może mieć postać jak na rysunku.



Rysunek 4. Schemat przykładowego wirtualnego systemu plików.

Pierwszym blokiem pliku jest jego nagłówek. W skład tego nagłówka może wchodzić jakaś suma kontrolna, wersja i inne potrzebne informacje. Blok o nazwie „nagłówki

plików" zawiera informacje o plikach takie jak: nazwa pliku, rozmiar, przesunięcie (w bajtach) względem początku pliku (lub innego wybranego punktu) i dowolne dodatkowe informacje. Kolejne bloki danych to już konkretne pliki. Znając pozycję pliku (przesunięcie) oraz jego rozmiar można łatwo odczytać ten plik za pomocą jednej funkcji czytającej (wiadomo gdzie zacząć czytać oraz ile bajtów przeczytać).

Taka struktura jest prosta a jednocześnie daje wiele korzyści. Po pierwsze, szybkość użycia funkcji `seek` na jednym pliku jest o wiele większa, niż `open` i `close` co sprawia, że dostęp do pliku będzie znacznie szybszy. Z nagłówków plików można zrobić tablicę asocjacyjną (mapę) dzięki czemu łatwo i szybko będzie wyszukać odpowiedni plik. Ten blok powinien zostać zaczytany cały do pamięci, dzięki czemu mamy wiedzę o wszystkich plikach w wirtualnym systemie plików[16]. Kolejną zaletą jest możliwość podglądu VFS. Dane w nagłówku plików zawierają wszystkie potrzebne informacje do stworzenia podglądu (podgląd będzie potrzebny tylko podczas tworzenia gry). Pliki można ułożyć tak, by za pomocą jednego odczytu odczytać wszystkie potrzebne pliki dla konkretnego poziomu w grze (np. plik1 do plik100 są to pliki dla poziomu pierwszego w grze. Za pomocą jednej funkcji odczytu można je wszystkie przeczytać a następnie rozdzielić i odpowiednio wykorzystać).

Kolejnym ważnym aspektem wirtualnego systemu plików jest kompresja. Każdy z plików w VFS może zostać skompresowany. Informacje o kompresji pliku mogą być zawarte w nagłówkach plików. Okazuje się, że szybsze jest wczytanie do pamięci pliku skompresowanego i dekompresja, niż odczytanie z dysku już zdekompresowanego pliku.

Przyjrzyjmy się małemu testowi odczytu danych nieskompresowanych oraz skompresowanych.

Metoda	Min. (MB/s)	Maks. (MB/s)	Średnia (MB/s)
ANSIC	47.847	48.828	48.527
W32	47.483	48.852	48.497
MMapped	45.830	48.828	48.190
Async I/O	48.239	48.852	48.700

Tabela 1. Pomiary odczytu danych z dysku twardego.

Metoda	Min. (MB/s)	Maks. (MB/s)	Średnia (MB/s)	Przyspieszenie %
ANSIC	59.067	69.348	67.558	+39%
W32	67.797	69.396	68.446	+41%
MMapped	52.247	53.562	52.980	+10%
Async I/O	68.634	69.832	69.242	+42%

Tabela 2. Pomiary odczytu skompresowanych danych z dysku twardego.

Testy zostały przeprowadzone w [58]. Do przeprowadzenia testów zostały użyte cztery metody odczytu danych z dysku twardego:

[16] Warto używać kilku VFS dla różnych typów danych ponieważ mapa z informacjami o plikach będzie mniejsza, więc wyszukanie odpowiedniego pliku będzie szybsze. Niektóre gry mają osobne VFS dla tekstur, modeli, dźwięków itp.

1. **ANSIC**: metoda odczytu danych za pomocą standardowej biblioteki języka C (fread).
2. **W32**: metoda odczytu danych za pomocą WinAPI (ReadFile).
3. **MMapped**: metoda odczytu danych za pomocą WinAPI (mechanizm używany w wirtualnej pamięci – Virtual Memory)
4. **Async I/O**: metoda odczytu danych za pomocą WinAPI (mechanizm odczytu danych w wątkach. W przykładzie zostało użytych osiem wątków)

W powyższym przykładzie odczytywane są dane wielkości 100Mb, które zostały skompresowane do 64Mb przy drugim teście. Łatwo zauważyć, jak duże przyspieszenie zostało uzyskane podczas odczytu pliku skompresowanego. Zauważmy, że odczyt mniejszego pliku jest szybszy. W momencie, gdy dysk odczytuje plik skompresowany, procesor jest w stanie idle (czyli nic nie robi), zatem można go wykorzystać do dekompresji poprzednio załadowanego pliku. Dzięki temu mamy dekompresję za darmo.

Zalety wirtualnego systemu plików nie tylko ograniczają się do szybkiego odczytu plików. W grach komputerowych bardzo często trafiają się błędy. Nawet w wersji release (wydaniowej). Dlatego niemal każda gra posiada łatki (patche) usuwające te błędy. Większość błędów w grze jest na poziomie kodu, więc patch polega na podmianie oryginalnego pliku wykonywalnego na nowy, w którym te błędy zostały naprawione. Inne gry (np. World of Warcraft) używają patch'y do rozszerzenia możliwości gry. Nie tyle naprawiają błędy, co modyfikują pewne aspekty gry. Do tego już potrzeba plików – zasobów takich jak tekstury, modele, dźwięki, skrypty itp. Wirtualny system plików świetnie nadaje się do tego typu patch'ów. Patch może składać się z odpowiedniego archiwum z plikami. Aplikacja odczytuje pliki z archiwum o najmłodszej dacie, jeśli plik zostanie znaleziony, wówczas jest odczytywany z danego archiwum. Jednakże datę użytkownik może sobie dowoli zmieniać i mogą wystąpić poważne błędy z grą. Lepszym pomysłem było by odczytywanie wersji archiwum, a konkretniej to wersji danych w archiwum. Takie dane mogą zostać zawarte w nagłówku wirtualnego systemu plików. Aplikacja na początku odczytuje nagłówki wszystkich VFS, następnie segreguje je według wersji danych (patchy). Gra powinna wiedzieć jakie patce ma już „zainstalowane" i pobierać dane zależnie od potrzeb. Warto zauważyć, że mogą być patch'e do patch'y. Dokładanie nowych archiwów do aplikacji podczas patch'owania ma tą zaletę, że łatwo można zrobić patch cofający do poprzedniej wersji. Wystarczy podmienić plik wykonywalny aplikacji, wtedy archiwa najnowszego patch'a nie będą brane pod uwagę. Jeśli nie chcemy aby gra miała możliwość cofania się w wersjach (czyli najczęstszy przypadek) wówczas, nowe dane mogą zostać dodane do poprzednich. Generalnie wirtualny system plików nie posiada możliwości edycji, gdyż pliki gry są tylko do odczytu. Aplikacja w żaden sposób nie powinna zmieniać tych plików.

Jednym z możliwości dodania plików do VFS jest umieszczenie nagłówka plików na końcu archiwum. Nagłówek wirtualnego systemu plików powinien wtedy zawierać informację o pozycji nagłówków plików i podczas odczytywania archiwum należy od razu skoczyć pod dany bajt i odczytać nagłówki plików. Jeśli chcemy dodać nowy plik, należy po prostu nadpisać nagłówki plików nowym plikiem i na końcu dopisać nagłówki plików wzbogacone o informacje o nowym pliku. Dzięki takiemu podejściu archiwum może się dowolnie rozszerzać.

2.5. Optymalizacje językowe

2.5.1 Klasy

Gry komputerowe to programy, które przede wszystkim stawiają na wydajność. Jest to najważniejsza i kluczowa cecha tego typu programów. Jeżeli gra nie będzie działać wystarczająco szybko (na tyle, aby uzyskać płynną animację obrazu, czyli ok. 25-30 klatek na sekundę), w praktyce, nie będzie dało się w nią grać. Znakomita większość gier pisana jest w języku C/C++. Programiści decydują się na ten język z powodu jego szybkości. Powstaje pytanie czy może być on szybszy od swojego pierwowzoru – języka C? C++ daje dużo większe możliwości niż C, ale jest to okupione zmniejszoną wydajnością. Jednakże znajomość podstaw C++ oraz kompilatora sprawia, że aplikacje napisane w tym języku mogą być niemal tak samo wydajne jak te napisane w C.

Nie należy optymalizować kodu przedwcześnie, gdyż przeważnie powoduje to spadek wydajności. Wąskich gardeł należy szukać przy użyciu profiler'ów. Dzięki profilowaniu kodu można łatwo zauważyć, w których momentach aplikacja potrzebuje najwięcej czasu na obliczenia i dopiero te miejsca należy optymalizować. Bardzo często sprowadza się to do zmiany lub ulepszenia jakiegoś algorytmu. W grafice trójwymiarowej najlepsze optymalizacje polegają na minimalizacji ilości wyświetlanej geometrii oraz efektów oświetlenia.

Drugą kwestią jest optymalizacja odpowiedniego kodu. Nie wszystkie fragmenty gry muszą działać z maksymalną wydajnością i tymi nie należy zajmować się w pierwszej kolejności. Przede wszystkim zależny nam na głównej pętli aplikacji i najbardziej zagnieżdżonych w niej pętlach.

Jeżeli optymalizacje „wyższo poziomowe" nie okażą się wystarczające należy zejść nieco niżej i spojrzeć na kod jaki napisaliśmy. Jednym z podstawowych pojęć C++ są obiekty. Jeżeli źle zaprojektujemy naszą aplikację, wówczas kompilator może tworzyć kod „za plecami". Szczególnie tyczy się to konstruktorów i destruktorów obiektów. Istnieje kilka podstawowych zasad, których warto się trzymać. Rozpatrzmy następującą funkcję:

```
void Function(int _iArgument)
{
      Object O;
      if( _iArgument < 0 )
            return;
}
```

Problem powyższej funkcji polega na, potencjalnie, niepotrzebnym utworzeniu obiektu. Jeżeli instrukcja warunkowa zostanie spełniona wówczas funkcja zakończy swoje działanie. Tworzymy i niszczymy niepotrzebnie obiekt. Jeżeli konstruktor i/lub destruktor obiektu alokuje/zwalnia pamięć, wówczas strata czasu może być znaczna. Należy również pamiętać, aby, jeśli to możliwe, nie tworzyć obiektów w pętlach. Najlepiej zrobić to wcześniej, a w każdej iteracji uaktualniać pola obiektu. Podobnie funkcje, które tworzą obiekty na stosie. Najlepiej funkcje takie zaprojektować w taki sposób, aby zamiast tworzenia, przyjmowały obiekt przez referencję.

Kolejnym aspektem optymalizacji obiektów są listy inicjalizacyjne. Często można zobaczyć taki kod:

```
class CPerson
{
	public:

		CPerson(const std::string& _strName)
		{
			m_strName = _strName;
		}

	private:

		std::string m_strName;
};
```

Pola klasy są inicjalizowane przed wywołaniem konstruktora. Zatem obiekt `m_strName` zaalokuje na stercie domyślny dla klasy std::string bufor znaków (kompilator najpierw wywoła domyślny konstruktor klasy `std::string`). Następnie zostanie wywołany konstruktor klasy `CPerson` i tam obiekt `m_strName` zostaje ustawiany (zostaje wywołany operator=). Może się okazać, że będzie musiał zwiększyć rozmiar bufora znaków, wówczas następuje zwolnienie i ponowna alokacja pamięci. Jeżeli przeniesiemy inicjalizację `m_strName` na listę inicjalizacyjną, wówczas zostanie wywołany niedomyślny konstruktor klasy `std::string`, więc alokacja nastąpi tylko raz. Jeżeli konstruktor klasy `CPerson` będzie pusty, kompilator może go odpowiednio zoptymalizować.

W przypadku małych obiektów takich jak `Vector` czy `Matrix` konstruktory powinny być puste. Oczywiście dobrze jest ustawiać początkowe wartości na 0 lub macierz jednostkową, żeby uniknąć potem błędów. Niemniej jednak, jeśli korzystamy w tych obiektów jako obiektów tymczasowych, przeważnie nie będzie potrzebna nam inicjalizacja w konstruktorze domyślnym. Taką operację lepiej przenieść do alternatywnego konstruktora i/lub utworzyć odpowiednią metodę do inicjalizacji. Klasę `CPerson` można zatem zoptymalizować do takiej postaci:

```
class CPerson
{
	public:

		CPerson()
		{
		}

		CPerson(const std::string _strName) : m_strName( _strName )
		{
		}

		void SetName(const std::string& _strName)
		{
			m_strName = _strName;
		}

	private:

		std::string m_strName;
};
```

Gdyby nie było alternatywnego konstruktora, wówczas mogła by się pojawić taka sytuacja:

```
CPerson Person;
Person.SetName( ”My long long long name” );
```

Taka sytuacja spowoduje, że wywoła się domyślny konstruktor `CPerson`, który wywoła domyślny konstruktor `std::string`. Następnie metoda `SetName` wymusza ponowną alokację pamięci w obiekcie `m_strName`.

Można do tego dodać jeszcze jeden fakt. Taniej jest korzystać z konstruktora kopiującego, niż tworzyć obiekt i korzystać z operator=. Najlepiej, zatem, jest tworzyć obiekty w ten sposób:

```
CPerson Person( Person2 );
```

niż

```
CPerson Person;
Person = Person2;
```

Programiści lubią korzystać z operatorów przeciążonych, ponieważ ułatwiają operację i sprawiają, że kod jest bardziej przejrzysty. Niestety nie ma nic za darmo. Operatory przeciążone, które zwracają obiekt przez wartość mogą być bardzo niewydajne, szczególnie, jeśli są często stosowane. Standardowy sposób na dodawanie dwóch wektorów w C++ jest następujący:

```
v = v1 + v2;
```

Taki zapis jest możliwy dzięki przeciążeniu operator+

```
Vector operator+(const Vector& _vec1, const Vector& _vec2);
```

W programowaniu gier bardzo często korzysta się z obiektów typu `Vector` czy `Matrix`, a szczególnie z operacji na nich. Dlatego funkcje przeznaczone do tego powinny być jak najbardziej wydajne. W przypadku operatorów przeciążonych, które zwracają obiekty przez wartość czas utworzenia tymczasowego obiektu i skopiowania go może okazać się zbyt duży. Mniej eleganckim, ale za to wydajniejszym sposobem jest napisanie metody `Vector::Add(const Vector& _vec1, const Vector& _vec2)` i w ten sposób dodawanie wektorów. Problem ten nie występuje w operator+= ponieważ modyfikuje on wartości pierwszego argumentu. Dlatego często wydajniejsze jest pisanie

```
v += v1;
```

niż

```
v = v + v1;
```

Bardzo często w programach pojawia się inkrementacja i dekrementacja. Jeżeli zapiszemy taki kod: i = j++ wówczas kompilator zrobi następujące czynności: stworzy tymczasową kopię obiektu j, zwiększy j (wywoła operator++ dla obiektu) i zwróci tymczasową kopię przypisując ją do i. Inkrementacja przedrostkowa nie posiada tego problemu. Obiekt tymczasowy nie jest tworzony. Ma to duże znaczenie dla typów zdefiniowanych przez programistę, dlatego zawsze kiedy to możliwe należy stosować inkrementację przedrostkową.

Jeżeli chcemy przejść po zbiorze (np. tablicy) i nie zależy nam na kolejności warto skorzystać z dekrementacji zamiast inkrementacji. Odejmowanie jest wykonywane szybciej na większości procesorów.

Ze względu na ograniczoną ilość pamięci należy starać się tworzyć jak najmniejsze pliki wykonywalne. Bardzo często może okazać się, że kod optymalizowany pod względem rozmiaru jest szybszy niż ten optymalizowany pod względem szybkości (mniejszy kod może być bardziej przyjazny dla szybkiej pamięci cache procesora). Niektóre kompilatory dodają informacje pomocne przy debugowaniu do pliku wykonywalnego, jeśli tak się dzieje należy wyłączyć tą opcję. Dodatkowy kod generowany jest także przez system obsługi wyjątków. Jeśli jest to możliwe należy to wyłączyć i zamiast rzucania wyjątków zwracać odpowiednie kody błędów z ewentualnym opisem (system logowania również się może sprawdzić).

Kompilator ma pełną swobodę w respektowaniu słowa kluczowego `inline`. Może to powodować wygenerowanie bardzo długich funkcji, tym bardziej, że kompilator może sam, bez wiedzy programisty, określać funkcje jako inline. Jest to dobry powód, aby tworzyć jak najprostsze domyślne konstruktory.

System RTTI (Runtime Type Information) generuje dużo informacji o obiektach. Dzięki RTTI możliwe jest użycie operatora `dynamic_cast` (który może być bardzo wolny w działaniu) oraz określenie typu obiektu. Warto rozważyć wyłączenie opcji RTTI. Spowoduje to uszczuplenie pliku wykonywalnego.

2.5.1. Operacje na liczbach całkowitych

Liczby całkowite mogą mieć różne rozmiary. Poniższa tabela przedstawia różne typy liczb całkowitych i ich minimalne oraz maksymalne rozmiary (w bitach).

Typ	Rozmiar	Minimalna wartość	Maksymalna wartość
char	8	-128	127
unsigned char	8	0	255
short int	16	-32768	32767
unsigned short int	16	0	65535
int	32	-2^31	2^31 – 1
unsigned int	32	0	2^32 – 1
__int64	64	-2^63	2^63 – 1
unsigned __int64	64	0	2^64 - 1

Niestety, deklaracja liczby całkowitej typu `int` o konkretnym rozmiarze różni się w zależności od platformy. Na przykład systemy 16-bitowe typ `short int` reprezentują jako `int`, a typ `int` jako long `int`. Typ `__int64` na 32-bitowym systemie Linux nazywa się `long long`, natomiast na 64-bitowym systemie Linux – `long int` (analogicznie dla wersji `unsigned __int64`). Niektóre kompilatory używają słów kluczowych dla typu `int` o danym rozmiarze, na przykład: `__int8`, `__int16`, `__int32`, `__int64`.

W większości przypadków operacje na liczbach całkowitych są bardzo szybkie. Nie zaleca się jednak używania większego rozmiaru liczby całkowitej niż rozmiar największego dostępnego rejestru (czyli używanie 64-bitowych zmiennych na 32-bitowym systemie), szczególnie dla operacji mnożenia i dzielenia.

Używając typu `int` kompilator zawsze wybierze najlepszy rozmiar dla zmiennej. Liczby całkowite o mniejszym zakresie takie jak char czy `short int` są nieco mniej wydajne, dlatego kompilator może zastąpić je liczbami o domyślnym rozmiarze (`int`) podczas wykonywania operacji (zostaną użyte wtedy tylko 8 lub 16 niższych bitów). W systemach 64-bitowych nie ma w zasadzie większej różnicy w wydajności między liczbami 32-bitowymi i 64-bitowymi jeżeli nie używamy dzielenia.

Często można spotkać się z kodem typu:

```
int a = b / 2;
```

Dzielenie jest najwolniejszą operacją arytmetyczną. Zatem dużo lepiej jest napisać:

```
int a = b * 0.5f;
```

Nie ma różnicy w inkrementacji przedrostkowej i przyrostkowej jeśli używamy zmiennej jako licznika pętli:

```
for(int i = 0; i < 10; i++) {}
```

jest tak samo szybkie jak:

```
for(int i = 0; i < 10; ++i) {}
```

Kompilator sam zoptymalizuje inkrementacje w takiej pętli ponieważ wie, że podczas zwiększania zmiennej i nie jest ona do niczego przypisywana. Jeżeli napiszemy kod:

```
x = tablica[++i];
```

okaże się bardziej wydajny niż:

```
x = tablica[i++];
```

Dzieje się tak z tego powodu, że obliczenie adresu elementu tablicy musi poczekać na obliczenie wartości i, co w rezultacie wydłuża czas otrzymania wartości x przynajmniej o dwa cykle. Kolejnym przykładem może być taki kod:

```
a = ++b;
```

W tym przypadku kompilator rozpozna, że wartości a i b są takie same po wykonaniu instrukcji, więc może je umieścić w tym samym rejestrze. Natomiast w przypadku:

```
a = b++;
```

Może się okazać, że wartości a i b są różne, więc zostaną umieszczone w osobnych rejestrach.

Typ `bool` jest reprezentowany jako 8-bitowa liczba całkowita, która może przyjąć wartość 0 lub 1. Operatory, które zwracają na wyjście typ bool mogą jedynie zwrócić 0 lub 1. Może to spowodować, że operacje na tym typie będą mniej wydajne niż się tego spodziewamy.

```
bool a, b, c, d;
c = a && b;
d = a || b;
```

Powyższego kodu kompilator może nie zoptymalizować i jedyne co wygeneruje to:

```
bool a, b, c, d;
if( a != 0 )
{
```

```
	if( b != 0 )
	{
		c = 1;
}
else
{
	goto CFALSE;
}
}
else
{
	CFALSE:
	c = 0;
}

if( a == 0 )
{
	if( b == 0 )
	{
		d = 0;
	}
	else
	{
		goto DTRUE;
	}
}
else
{
	DTRUE:
	d = 1;
}
```

Taki kod jest daleki od wydajnego, a ilość rozgałęzień może sięgać znacznie dalej. Jeżeli byśmy stosowali operacje logiczne dla wartości innych niż tylko 0 lub 1, wówczas kompilator nie musiał by się martwić o konkretne wartości. Bardziej optymalny kod zakłada, że wartości a i b będą ustawione na prawidłowe wartości logiczne lub będą pochodziły ze źródeł, które takie wartości dostarczają.

```
int a = 0, b =0, c, d;
c = a & b;
d = a | b;
```

Zamiast typu `int` można skorzystać z typu char, ponieważ ma ten sam rozmiar co `bool`. Dzięki skorzystaniu z typu całkowitoliczbowego możliwe jest użycie operatorów bitowych, które są bardzo szybkie (jeden cykl zegara).

2.5.2. Operacje na liczbach zmiennoprzecinkowych

Konwertowanie liczb zmiennoprzecinkowych na całkowite może być niewydajne w odniesieniu do niektórych procesorów oraz kompilatorów. Rzutowanie float na int może zabrać nawet kilkadziesiąt cykli procesora. Dzieje się tak ze względu na standard ANSI C, który wymusza odrzucenie części ułamkowej. Z drugiej strony, FPU zaokrągla domyślnie do najbliższej wartości całkowitej. Poniższe optymalizacje działają dla liczb zmiennoprzecinkowych standardu IEEE.

s	e	e	e	e	e	e	e	e	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m
31	30							23	22																			0

Tabela 3. Standard IEEE dla 32-bitowej liczby zmiennoprzecinkowej. Format posiada 1 bitu znaku, 8 bitów eksponenty i 23 bity mantysy. Liczba zmiennoprzecinkowa to s * 1,m * 2^(e-127)

Do optymalizacji będziemy potrzebować unii, dzięki której będziemy mieć możliwość zapisu jednocześnie do liczby całkowitej i zmiennoprzecinkowej

```
typedef union
{
      int i;
      float f;
} IntFloat;
```

Konwersja `float` do `int (342.345f do 342)`:

```
IntFloat n, bias;

bias.i = (23 + 127) << 23;
n.f = 342.345f;
n.f += bias.f;
n.i -= bias.i;
```

Dodanie 1*2^23 w wersji zmiennoprzecinkowej powoduje umieszczenie mantysy na 23 niższych bitach, jednocześnie ustawiając eksponentę na wartość 23 + 127. Odjęcie tej eksponenty w wersji całkowitej usuwa wyższe bity dzięki czemu wartość całkowita pozostaje na niższych bitach. Taka operacja może spowodować przyspieszenie z kilkudziesięciu do zaledwie kilku cykli procesora. Ta optymalizacja działa tylko dla liczb, które nie przekraczają zakresu 2^23 (przekroczenie zakresu dodawanej stałej progu).

Czasami chcemy sprawdzić jaki znak ma liczba. Załóżmy, że mamy liczbę f typu `float`. Normalnie sprawdzenie będzie wyglądało następująco:

```
if( f < 0.0f )
```

Ponieważ zarówno liczby zmiennoprzecinkowe jak i całkowite znak przechowują na 31 bicie, można to zrobić tak:

```
IntFloat n;
n.f = f;
if( n.i < 0 )
```

Okazuje się, że porównanie liczb całkowitych może być szybsze niż porównanie liczb zmiennoprzecinkowych z powodu szybszego przebiegu strumienia instrukcji liczb całkowitych.

Gdy chcemy uzyskać wartość bezwzględną musimy porównać liczbę z 0 i ewentualnie zmienić jej znak:

```
if( f < 0.0f ) f *= -1.0f;
```

W praktyce potrzebujemy jedynie zmiany znaku. Instrukcja warunkowa jest niepotrzebna i okazuje się, że można ją pominąć:

```
IntFloat n;
n.f = f;
n.i &= 0x7FFFFFFF;
f = n.f;
```

Jedyne co tutaj robimy to ustawiamy bit znaku na 0.

Przycinanie liczby do pewnego zakresu (najczęściej [0,1]) również może być bardzo przydatne. Standardowa metoda wykonuje jedno lub dwa porównania. Przycinanie liczby f do 0 gdy f < 0 może wyglądać następująco:

```
IntFloat n;
n.f = f;
int s = n.i >> 31;
s = ~s;
n.i &= s;
f = n.f;
```

Przesuwamy w prawo bity liczby f o 31 pozycji, czyli powielamy bit znaku na wszystkie pozostałe bity liczby i przypisujemy to tymczasowej zmiennej s, która następnie neguje wszystkie swoje bity. Negacja powoduje utworzenie maski z zerami, gdy f jest liczbą ujemną lub z jedynkami, gdy f jest dodatnie. Iloczyn logiczny powoduje wyzerowanie liczby f, lub może niczego nie zmienić. Jeżeli f jest ujemne otrzymujemy 0, jeśli jest dodatnie nic nie zostanie zmienione.

Przycięcie do 1, gdy f > 1, można zrobić w ten sposób, że najpierw odejmiemy 1, przytniemy do 0 i dodamy 1:

```
IntFloat n;
n.f = f – 1.0f;
int s = n.i >> 31;
```

```
n.i &= s;
f = n.f + 1.0f;
```

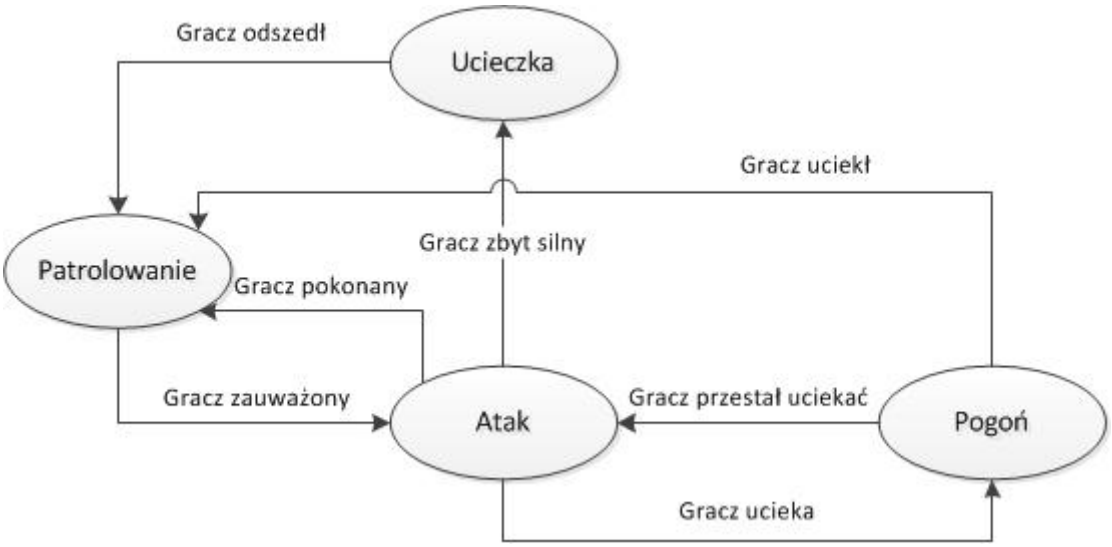
Optymalizacje to bardzo ważna rzecz w programowaniu gier. Przedstawiłem powyżej kilka sztuczek związanych z językiem C/C++, które mogą przyspieszyć działanie programów. Należy pamiętać jednak przede wszystkim, żeby szukać wąskich gardeł przy używając profilowania. Warto również patrzyć na kod assemblerowy wyprodukowany przez kompilator. Bardzo często jest on dobrze zoptymalizowany, a nasze, własnoręczne przyspieszanie obliczeń może wpłynąć negatywnie (na przykład funkcje inline mogą niespodziewanie utworzyć rozbudowany kod).

3. Sztuczna inteligencja

3.1. Automat o skończonej liczbie stanów

Automat o skończonej liczbie stanów to automat, którego liczba stanów jest skończona. Stanem nazwiemy określoną sytuację, w której znajduje się obiekt. Automat jest zawsze w dokładnie jednym stanie. Przykładem stanu może być lampka. Lampka może być włączona, czyli być w stanie „włączona", lub wyłączona, czyli być w stanie „wyłączona". Automat posiada wejście, które powoduje przejście do określonego stanu. Aktualny stan nazywany jest stanem wyjściowym. Weźmy na przykład drzwi. Drzwi mogą być w stanie: zamknięte, otwarte, zamknięte na klucz. Jeśli do automatu damy na wejście użycie klucza, a drzwi są zamknięte na klucz to nastąpi otwarcie zamka (jeśli są zamknięte ale nie na zamek to zostaną zamknięte na zamek). Następne wejście – użycie ręki, spowoduje otwarcie drzwi, lub zamknięcie, jeśli są otwarte. Ważne jest to, że jeżeli drzwi są otwarte, a na wejście automat dostanie użycie klucza to nic nie nastąpi.

W grach komputerowych automaty o skończonej liczbie stanów (FSM – Finite State Machine) mogą być użyte do wielu zastosowań. W sztucznej inteligencji można go użyć do symulacji zachowań postaci kierowanych przez komputer (NPC – Non Player Character). Diagram poniżej przedstawia automat o skończonej liczbie stanów.



Rysunek 5. Diagram przedstawia automat stanów dla komputerowego, wrogiego gracza.

Przy pomocy stanów oraz sygnałów wejściowych można określić macierz przejść między stanami.

Aktualny stan	Wejście	Wyjście
Patrolowanie	Gracz zauważony	Atak
Atak	Gracz ucieka	Pogoń
Atak	Gracz zbyt silny	Ucieczka
Atak	Gracz pokonany	Patrolowanie
Pogoń	Gracz przestał uciekać	Atak
Pogoń	Gracz uciekł	Patrolowanie
Ucieczka	Gracz odszedł	Patrolowanie

Tabela 4. Przykładowa macierz przejść między stanami

W zależności od aktualnego stanu i sygnału wejściowego zmienia się stan NPC, czyli jego zachowanie. Budując odpowiednio złożony mechanizm stanów można dobrze symulować zachowania sztucznej inteligencji na reakcje gracza i/lub środowiska.

Każdy agent (obiekt sztucznej inteligencji) może posiadać własny automat stanów. Jeżeli w aktualnym momencie gry mamy wielu agentów, należy, w każdej aktualizacji sztucznej inteligencji, sprawdzać ich stany. Dodatkowo, bardziej złożone systemy mogą stosować komunikację między agentami. Istnieją dwa podstawowe sposoby reakcji agentów na otoczenie:

- Przyglądanie się otoczeniu (odpytywanie)
- Oczekiwanie na bodźce (zdarzenia)

Dla dużej liczby agentów odpytywanie może być bardzo niewydajne. Jeżeli w grze mamy bombę, której wybuch wpływa na pewną ilość agentów, każdy z nich musi sprawdzić czy bomba wybuchła i czy trafiła go. Za pomocą zdarzeń, bomba sama wyśle komunikat do agentów o tym, że wybuchła, czy trafiła i jak mocno. Agent wtedy może odpowiednio zareagować. Komunikat może być obiektem przechowującym pola z informacją o zdarzeniu. Warto, żeby taki komunikat zawierał nadawcę, wówczas odbiorca będzie miał większe pole manewru. Dzięki systemowi komunikatów można łatwo zrobić dziennik, który można wykorzystać do wychwytywania błędów.

Zachowania agenta można zaprogramować przy pomocy automatu stanów, natomiast przejścia między stanami za pomocą komunikatów. Poniżej znajduje się przykład automatu sterowanego zdarzeniami:

```
BeginStateMachine

      OnEnter
      {
            SetState( STATE_PATROLOWANIE )
      }

      OnMessage( MSG_SMIERC )
      {
            UsunObiekt();
      }

      State( STATE_PATROLOWANIE )
      {
```

```
	OnEnter
	{
		UstawCelPatrolowania();
	}

	OnUpdate
	{
		if( GraczZauwazony() )
		{
			SetState( STATE_ATAK )
		}
	}
}

State( STATE_ATAK )
{

	OnEnter
	{
		UstawCelNaGracza();
	}

	OnUpdate
	{
		if( GraczPokonany() )
		{
			SetState( STATE_PATROLOWANIE )
		}
		else if( GraczZbytSilny() )
		{
			SetState( STATE_UCIECZKA )
		}
		else if( GraczUcieka() )
		{
			SetState( STATE_POGON )
		}
		else if( GraczWZasiegu() )
		{
			Atakuj();
		}
	}
}

State( STATE_UCIECZKA )
{
	OnEnter
	{
		UstalKierunekUcieczki();
	}

	OnUpdate
	{
		if( GraczGoni() )
```

```
                {
                        UciekajWUstalonymKierunku();
                }
                else if( GraczOdszedl() )
                {
                        SetState( STATE_PATROLOWANIE )
                }
        }
    }

    State( STATE_POGON )
    {
        OnEnter
        {
                UstawCelNaGracza();
        }

        OnUpdate
        {
                if( GraczUcieka() )
                {
                        GonGracza();
                }
                else if( GraczUciekl() )
                {
                        SetState( STATE_PATROLOWANIE )
                }
        }
    }

EndStateMachine
```

Powyższy automat stanów sterowany jest za pomocą zdarzeń. Komunikat przekazany do `OnMessage` może spowodować uruchomienie, zatrzymanie lub przejście automatu do dowolnego stanu. Powyższy pseudo kod można zrealizować w języku C/C++ za pomocą odpowiednio napisanych makr:

```
#define BeginStateMachine if( STATE_Global == _iState ) {
#define State(_S) if(0){return true;}} else if(_S == _iState){ if(0){
#define OnEnter return true;} else if(MSG_RESERVED_Enter == _pMsg->name) {
#define OnExit return true;} else if( MSG_RESERVED_Exit == _pMsg->name ) {
#define OnUpdate return true;} else if(MSG_RESERVED_Update == _pMsg->name){
#define OnMessage(_S) return true;} else if( _S == _pMsg->name ) {
#define SetState(_S) SetStateInGameObject( _pGameObj, (int)_S );
#define EndStateMachine return true;}} else { return false; } return false;
```

Następnie należy umieścić automat w poniższej funkcji:

```
Bool ProcessStateMachine(CGameObject* _pGameObj, int _iState, CMsgObject* _pMsg);
```

Za pomocą makr można stworzyć nowy język programowania. Ponadto kod automatu staje się czytelniejszy i łatwiej szukać błędów.

Automat stanów realizowany jest za pomocą instrukcji `if-then-else` lub `switch-case` (które w praktyce może być rozwijane do `if-then-else`), a aktualny stan przechowywany jest jako liczba całkowita. Takie rozwiązanie daje możliwość szybkiej implementacji oraz małe zapotrzebowanie na pamięć. Niestety instrukcje warunkowe, szczególnie w przypadku większych automatów, mogą stanowić zbyt duży narzut. Automat, który ma n stanów potrzebuje średnio n/2 porównań `if-then-else`, aby znaleźć odpowiednią funkcję. Można skrócić czas takiego wyszukania stosując bardziej obiektowe podejście. Każdy stan może być reprezentowany przez obiekt. Obiekt ten przechowuje wskaźniki na funkcje, które odpowiadają funkcjom stanu (`OnEnter`, `OnUpdate`, itd.). Automat śledzi aktualny stan posiadając wskaźnik na obiekt stanu. Wywołanie funkcji stanu odbywa się w obiekcie przy użyciu wskaźnika na funkcję.

Automat stanów może być również managerem stanów. Dzięki obiektowemu podejściu można, w prosty sposób, dodawać stany, nawet w trakcie działania aplikacji. W ten sposób można stworzyć sztuczną inteligencję, która sama się rozbudowuje. Automat stanów może przechowywać wskaźniki na stany w jakimś buforze. Metoda `AddState` dodaje do bufora nowy stan. Stany, w postaci obiektów, nie muszą nic o sobie wiedzieć prócz tego, jaki stan w danym momencie mają wywołać (inaczej: muszą znać identyfikatory innych stanów), ponieważ może wystąpić problem, że stan A chce przejść do stanu K, więc wywołuje metodę automatu `SetState(K)`. Jeśli stanu K nie ma w buforze stanów, wystąpi błąd. Oczywiście jedną instrukcją warunkową można zaradzić sytuacji, ale wówczas należy przemyśleć rozwiązanie zastępcze (być może jakiś domyślny stan), które będzie używane jeśli dany stan nie zostanie odnaleziony. Ponieważ chcemy maksymalnie zmniejszyć ilość instrukcji warunkowych, nie opłaca się stosowania mapy jako bufora stanów. Stany to zazwyczaj liczby całkowite, możemy zatem przyjąć, że mogą być również reprezentowane jako liczby naturalne. W takim razie, można użyć zwykłej tablicy. Jeśli ma być rozszerzalna, to można skorzystać z dynamicznej tablicy lub z tablicy o stałym rozmiarze i z góry ustaloną maksymalną liczbą stanów. Użycie tablicy powoduje pewną niedogodność. Nie można usunąć stanu. Z dynamicznie tworzoną tablicą jest to możliwe, ale wiąże się z realokacją pamięci. Dynamiczne dodawanie stanów w trakcie działania aplikacji może powodować pewne błędy, szczególnie jeśli stany będą dodawane z kilku źródeł (modyfikacje AI w skryptach przez modder'ów). Identyfikator stanu może się powielić, czyli nadpisać wskaźnik o danym identyfikatorze. Należy o tym pamiętać.

3.2. Wyszukiwanie drogi

W grach komputerowych bardzo często występują postacie kierowane przez komputer (sztuczną inteligencję). Postacie takie poruszają się po świecie i wchodzą w interakcje z graczem. Przemieszczenie komputerowego gracza z punktu A do punktu B w trójwymiarowym świecie nie jest rzeczą trywialną. Wyszukanie odpowiedniej drogi polega przede wszystkim na ominięciu przeszkód oraz ślepych uliczek. Dodatkowo, wybór drogi powinien być optymalny ze względu na ukształtowanie terenu (postać powinna iść drogą a nie przez bagno).

Algorytm A* jest to algorytm przeszukujący graf w poszukiwaniu najkrótszej drogi pomiędzy dwoma wierzchołkami. Graf może być inaczej nazywany jako przestrzeń stanów (w odniesieniu do A*). W wyszukiwaniu drogi stanem nazwiemy aktualne położenie agenta w świecie gry. Przyległe stany uzyskujemy przemieszczając agenta na przyległe położenie. Jeżeli świat gry podzielony jest na kratki, to stanem będzie kratka, w której aktualnie znajduje się agent, a przyległe stany to będą kratki sąsiadujące z aktualną. Algorytm A* sprawdza najbardziej obiecujące, nie odkryte położenia, które widzi. Jeżeli dane położenie zostało zbadane i jest ono zarówno celem, wówczas algorytm kończy działanie. Jeżeli aktualnie badane położenie nie jest celem, wówczas algorytm zapamiętuje wszystkie sąsiednie położenia, aby sprawdzić je później.

Algorytm A* śledzi dwie listy stanów: `Open` i `Closed`. Lista `Open` przechowuje stany, które należy przebadać, natomiast lista `Closed` zawiera stany już sprawdzone. Początkowo lista `Closed` jest pusta, a `Open` zawiera jeden stan – początkowe położenie (aktualne położenie aktora). W każdej iteracji algorytm wybiera najbardziej obiecujący stan z listy `Open` i sprawdza go. Jeżeli nie jest to stan docelowy, zapamiętuje sąsiednie stany. Jeżeli te stany są nowe, zostają zapamiętane w liście `Open`. Jeżeli są już na liście `Open`, algorytm uaktualnia informacje o nich, gdy można się dostać do nich szybciej. Jeżeli są na liście `Closed` należy je zignorować ponieważ zostały już przebadane. Jeżeli lista `Open` została opróżniona, a cel nie został osiągnięty wówczas oznacza to, że nie istnieje droga z początkowego położenia do docelowego. Najbardziej obiecujący stan to stan z najmniejszą średnią ścieżką, która do niego prowadzi. Każdy stan S pozwala określić:

- `CostFromStart(S)` - Koszt najtańszej ścieżki z położenia początkowego do tego położenia
- `CostToGoal(S)` – koszt pozostałej drogi do przebycia (heurystyczna ocena)
- `CostFromStart(S) + CostToGoal(S)` - łączną ocenę drogi
- `TotalCost(S)` – najniższa wartość z tej funkcji to łączna ocena drogi, dzięki której wiadomo, który następny stan wybrać.

Oto pseudokod algorytmu A*

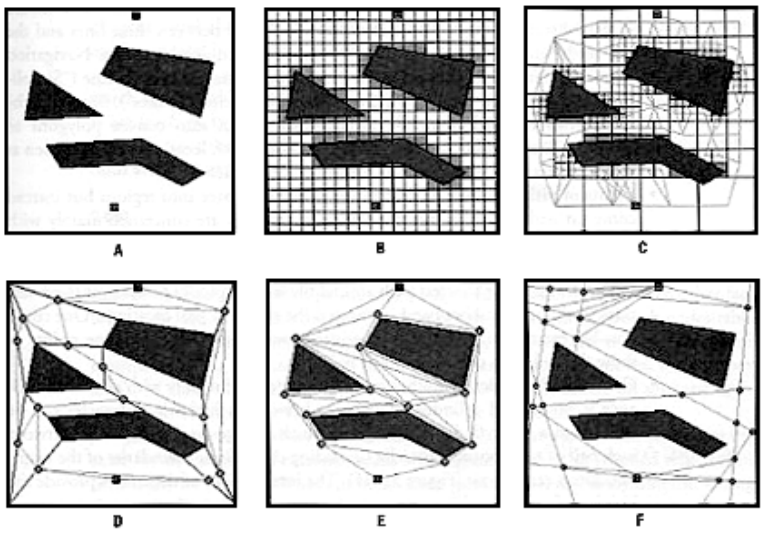
```
PriorityQueue Open;
List Closed;

AStarDoSearch(location _StartLocation, location _GoalLocation, agent Agent)
{
      Open.clear();
      Closed.clear();

      StartNode.location = _StartLocation;
```

```
StartNode.CostFromStart = 0;
StartNode.CostToGoal = PathCost(_StartLocation,_GoalLocation,Agent);
StartNode.parent = NULL;

Open.add( StartNode );

while( !Open.empty() )
{
  Node = Open.pop();

  if( IsGoalNode( Node ) )
  {
    ConstructPath( Node, _StartLocation );
    return true;
  }
  else
  {
    foreach(NewNode in Node)
    {
      NewCost = Node.CostFromStart +TraverseCost(Node,NewNode,Agent);

      if( (Open.contains(NewNode) || Closed.contains(NewNode)) &&
          ( NewNode.CostFromStart <= NewCost ) )
      {
        GotoNextElement();
      }
      else
      {
        NewNode.parent = Node;
        NewNode.CostFromStart = NewCost;
        NewNode.CostToGoal = PathCost(NewNode.location,_GoalLocation,
                                      Agent);

        NewNode.TotalCost = NewNode.CostFromStart+NewNode.CostToGoal;

        if( Closed.contains( NewNode ) )
        {
          Closed.erase( NewNode );
        }

        if( Open.contains( NewNode ) )
        {
          ModifyNode( Open, NewNode );
        }
        else
        {
          Open.add( NewNode );
        }

      }//else

    }//foreach
```

```
        }//else

        Closed.add( Node );

      }//while

      return false;

}
```

Algorytm przyjmuje trzy argumenty: położenie początkowe, położenie docelowe oraz agenta. Na początku listy `Open` i `Closed` są czyszczone oraz inicjalizowany jest początkowy węzeł i dodawany do listy `Open`. Lista `Open` przeszukiwana jest dopóki nie jest pusta (czyli przynajmniej jedna iteracja), czyli do znalezienia drogi lub napotkania błędu podczas szukania. Najpierw zdejmowany jest element z najmniejszą wartością `TotalCost` i jeśli jest to docelowy stan należy utworzyć ścieżkę i zakończyć algorytm. Jeśli nie jest to stan końcowy należy przejść przez wszystkie elementy z `Node`. Ustalany jest koszt przejścia z aktualnego węzła do nowego. Jeśli ten węzeł istnieje i nie jest lepszy (koszt) należy go zignorować i przejść do kolejnego węzła. W przeciwnym wypadku należy zapamiętać nowe wartości dla nowego węzła. Jeśli znajduje się w `Closed` to go usuń, jeśli jest w `Open` to go zmodyfikuj na tej liście. Jeśli go nie ma w `Open` to go dodaj. Na końcu umieszczamy aktualny węzeł w `Closed`, jeśli nie jest to węzeł docelowy i zwracamy `false` jeśli algorytm nie odnalazł ścieżki. Funkcja `PathCost` wyznacza oszacowany koszt przejścia z jednego położenia do drugiego biorąc pod uwagę typ agenta.

Skoro znamy już przybliżoną postać algorytmu można zastosować go w grze. Niestety w trójwymiarowym świecie nie jest to takie łatwe ponieważ mamy teoretycznie nieskończenie dużą liczbę położeń – stanów. Jednym ze sposobów na zminimalizowanie ilości stanów jest podział przestrzeni (siatki nawigacyjne). Rysunek 3.2.1. przedstawia kilka sposobów podziału.



Rysunek 3.2.6. Różne typy podziału przestrzeni. A) przestrzeń bez podziału B) Prostokątna siatka C) Drzewo czwórkowe D) Wielokąty wypukłe E) Punkty widoczności F) Uogólnione cylindry. Obrazek pobrany z [13].

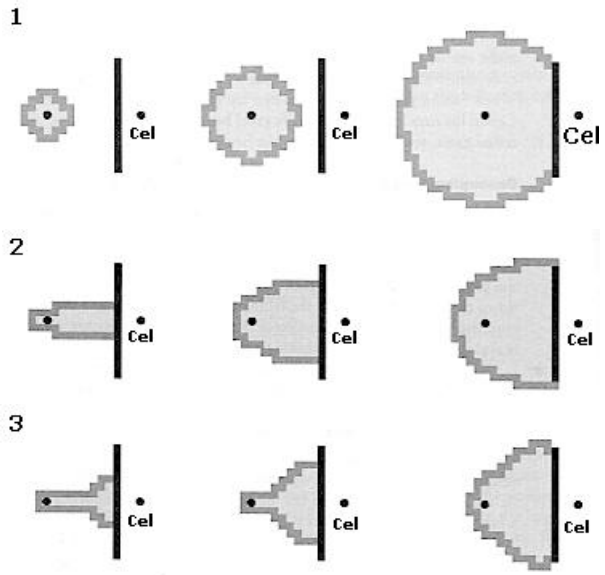
- **Prostokątna siatka** – przestrzeń podzielona jest na regularną siatkę kwadratów. Kwadraty, na których znajduje się geometria, której nie da się przekroczyć są zaciemnione. Stany te należy wykluczyć w procesie wyszukiwania drogi. Położeniem może być środek lub wierzchołek kratki.
- **Drzewo czwórkowe** – podobny sposób do poprzedniego, ale zawiera kwadraty różnych rozmiarów. Większe kwadraty zapewniają szybsze przeszukiwanie, ponieważ wiadomo, że nie ma w nich przeszkód.
- **Wielokąty wypukłe** – świat podzielony jest na wielokąty wypukle. Każdy wierzchołek łączony jest z najbliższym widocznym wierzchołkiem.
- **Punkty widoczności** – punkty widoczności nie dzielą przestrzeni, ale wyznaczają położenia. Każdy zbiór nieprzekraczalnej geometrii może być otoczony punktami widoczności. Najkrótsza droga między dwiema przeszkodami to odcinek między ich punktami widoczności.
- **Uogólnione cylindry** – przestrzeń między sąsiednimi przeszkodami można traktować jako cylinder. Pomiędzy parą sąsiednich przeszkód liczy się środkową oś. Przecięcie tych linii tworzy położenia wykorzystywane jako stany.

W zależności od wybranego typu siatki nawigacyjnej sąsiedzi stanu inaczej są określani. Dla siatki prostokątnej jest osiem sąsiadów (lub cztery, jeśli pomijamy sąsiadów na ukos) i łatwo jest ich określić: x+1, x-1, y+1, y-1, itd. Niestety w innych typach siatek określenie sąsiadów może już nie być takie łatwe, ponieważ w niektórych sąsiedzi to wszystkie widoczne węzły. Może być ich zbyt wiele co znacznie wydłuży czas wykonania algorytmu. Ponadto, sąsiedzi zależą również od terenu (dlatego w algorytmie do uzyskania kosztu drogi jednym z argumentów jest aktor) – samochód nie będzie jeździł po wodzie.

Funkcja, która liczy koszt między dwoma położeniami określa sposób na zminimalizowanie szukanej ścieżki. Może być to na przykład: odległość, czas dotarcia, zużycie paliwa, itp. Dodatkowo można dodać kary lub nagrody za przechodzenie po danym terenie. Wartości te powinny być zmienne w zależności od aktora. Jeśli jedziemy samochodem osobowym to powinien dostać więcej punktów za poruszanie się po drodze, natomiast dla piechura nie będzie miało większego znaczenia czy porusza się po drodze czy po łące.

Ocena kosztu dojścia do celu jest to uzupełnienie odległości od punktu początkowego. Ocena nie powinna być zawyżana jeśli chcemy znaleźć optymalną ścieżkę. Najlepiej jest pomnożyć rzeczywistą odległość na mapie przez minimalny koszt terenu na jednostkę odległości. Uzyskany wynik zawsze będzie nieoszacowany ponieważ droga nigdy nie może być krótsza niż bezpośrednia odległość między dwoma punktami. Bardzo ważna jest szybkość przeszukiwania, dlatego wartość CostToGoal ma znaczący wpływ na efektywność algorytmu. Jeśli udało by się określić heurystyczne informacje, w którym kierunku należy przeszukiwać stany, wówczas można znacząco skrócić czas wykonania algorytmu. Czasami chcemy przeszacować koszt do celu, ale może to prowadzić do otrzymania nieoptymalnej ścieżki kosztem szybkości algorytmu. Jeżeli byśmy znali rzeczywisty koszt dotarcia do celu, A* nie będzie przeszukiwał nieprawidłowych ścieżek, jednakże znów może to dać bardzo złą drogę do celu. Policzenie geometrycznej odległości między węzłem i celem daje pewne niedoszacowanie kosztu, dzięki czemu zawsze otrzymamy optymalną ścieżkę (ponieważ bezpośrednia odległość daje najmniejszy koszt). Lekkie przeszacowanie kosztu może pomóc w szybkim odnalezieniu w miarę optymalnej ścieżki. Heurystyczna część łącznego kosztu: koszt do węzła + heurystyczny koszt, może zniekształcić sposób pobierania węzłów z listy Open, jeżeli jest większa niż

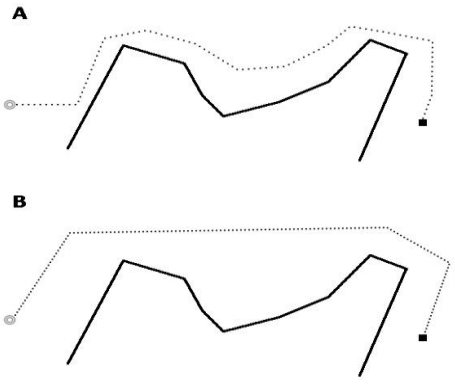
powinna. A* wybiera zawsze węzły z największym kosztem, dlatego będą promowane węzły bliższe celu. Jeżeli heurystyczny koszt ustawimy na 0 (czyli niedoszacowanie), wówczas będą przeszukiwane węzły dookoła startowego położenia. Zwiększenie heurystyki spowoduje bardziej intensywne przeszukiwanie węzłów w kierunku celu.



Rysunek 3.2.7. Wyszukiwanie drogi w zależności od kosztu heurystycznego. 1) Heurystyczny koszt wynoszący 0 2) Heurystyczny koszt obliczony przy pomocy euklidesowej odległości do celu 3) Przeszacowany heurystyczny koszt. Obrazek pobrany z [15]

Sztuczna inteligencja ma działać nie tylko dobrze i szybko, ale inteligentnie. Wyszukiwane ścieżki są zazwyczaj kanciaste i prowadzą najkrótszą drogą do celu. Nikt nie przechodzi obok budynku ocierając się o ściany tylko w pewnej odległości. Wyszukana droga powinna być poprawna estetycznie. Uzyskane ścieżki można poprawić na trzy sposoby: wygładzić, zrobić bardziej bezpośrednimi lub wyprostować.

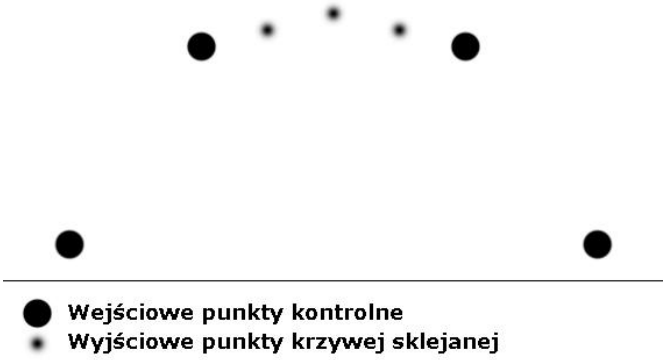
Ścieżki uzyskane algorytmem A* często wyglądają bardzo nienaturalnie ponieważ mają tendencję do tworzenia obrysu przeszkód. Aktor, który porusza się po takiej ścieżce robi to sztucznie i nie sprawia przekonującego wrażenia. Jednym ze sposobów na ulepszenie tych ścieżek jest ich lepsze ocenianie w algorytmie, natomiast drugi to prostowanie ich już po ich obliczeniu.



Rysunek 8.2.3. A) Standardowa ścieżka A* B) Wyprostowana ścieżka A*

Na rysunku 3.2.3. obie ścieżki przebywają taką samą odległość. Ponieważ obie ścieżki mają taki sam koszt, A* wybiera pierwszą, która się pojawi. Funkcja wyznaczająca koszt powinna promować prostą ścieżkę. Można to zrobić dodając karę za wybranie drogi, która zmienia kierunek w stosunku do poprzedniego. Wystarczy dawać kary jedynie w momencie, kiedy nowy krok nie podąża w stronę wyznaczoną przez poprzedni. Rozsądną karą może być połowa kosztu kroku w danym kierunku. Dla regularnych siatek nawet najmniejsza kara spowoduje wybranie tylko prostych ścieżek.

Algorytm A* ma jeszcze jedną złą cechę. Ścieżki mają bardzo ostre zakręty. Agent porusza się wtedy jak robot. Technika prostowania drogi nie rozwiązuje tego problemu, robią to natomiast krzywe sklejane Catmulla-Roma. Tworzą one ścieżkę przechodzącą przez wszystkie punkty kontrolne. Punktami kontrolnymi w tym przypadku mogą być tylko punkty wyznaczone algorytmem A* (w przeciwnym wypadku agent może wpaść na przeszkodę).



Rysunek 3.2.3. Krzywa sklejana uzyskiwana z punktów kontrolnych.

Algorytm Catmulla-Roma wymaga podania czterech punktów kontrolnych, a uzyskana krzywa zawarta jest między drugim i trzecim punktem. Aby uzyskać krzywą pomiędzy pierwszym i drugim punktem należy podać dwa razy pierwszy punkt, następnie drugi i trzeci. Jeśli chcemy uzyskać krzywą pomiędzy trzecim i czwartym punktem, należy podać drugi i trzeci oraz dwa razy czwarty punkt.

Wzór na krzywą sklejaną Catmulla-Roma wygląda następująco:

$$\begin{aligned} punktWyjściowy = {} & punkt1 \cdot (-0.5 \cdot u^3 + u^2 - 0.5 \cdot u) + \\ & punkt2 \cdot (1.5 \cdot u^3 + -2.5 \cdot u^2 + 1.0) + \\ & punkt3 \cdot (-1.5 \cdot u^3 + 2.0 \cdot u^2 + 0.5 \cdot u) + \\ & punkt4 \cdot (0.5 \cdot u^3 - 0.5 \cdot u^2) \end{aligned}$$

Krzywa ta przechodzi przez punkty kontrolne. Dla u = 0 otrzymamy punkt2, natomiast dla u = 1 otrzymamy punkt3. Ponieważ szybkość obliczeń sztucznej inteligencji jest bardzo ważna powyższy wzór można częściowo obliczyć dla pewnych stałych wartości u: 0.0, 0.25, 0.5, 0.75. Jedynymi niewiadomymi we wzorze pozostają punkty.

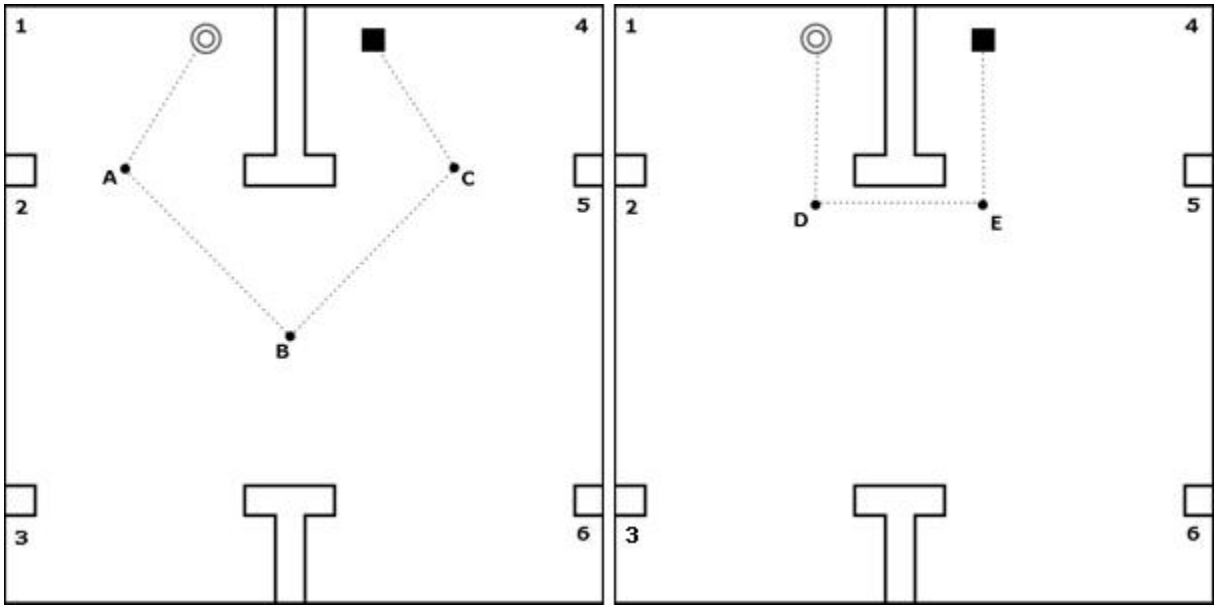
Mając już częściowo obliczone cztery wzory (dla każdej wartości u jeden wzór), można teraz przejść przez ścieżkę otrzymaną z algorytmu A* i utworzyć nową. Należy pamiętać,

aby na początku podać dwa razy pierwszy punkt i na końcu dwa razy ostatni. Ponadto, jeżeli A* zwrócił jedynie dwa punkty, należy pominąć powyższy wzór. Ścieżkę można znacznie zmniejszyć szukając punktów na linii prostej i eliminując je. Trzeba pamiętać, że używając wygładzania otrzymujemy cztery razy więcej punktów.

Stosując algorytm A* do wyszukania drogi w zamkniętych pomieszczeniach również można napotkać na problemy. Algorytm A* posiada bardzo ważną technikę: hierarchiczne wyszukanie ścieżki. Odbywa się to w dwóch krokach:

- Odnajdź drogę na poziomie globalnym
- Odnajdź drogę na poziomie lokalnym

Jeśli chcemy zastosować to do zamkniętego pomieszczenia, które jest podzielone na pokoje, algorytm A* można zastosować do wyszukania drogi pomiędzy dwoma pokojami. Następnie należy wyszukać drogę przez pokój z jednego punktu do drugiego, przeszukując tylko te pokoje przez które agent będzie przechodził. Końcowymi punktami każdego etapu przeszukiwania będą drzwi. Oznacza to, że agent będzie przechodził przez ich środek. Jeżeli drzwi są bardzo szerokie agent będzie się zachowywał bardzo nienaturalnie.



Rysunek 3.2.4. Z Lewej: standardowa ścieżka między dwoma pokojami. Z prawej: zoptymalizowana ścieżka.

Rozwiązaniem tego problemu będzie znajdowanie drogi do kolejnych drzwi zamiast do najbliższych. W momencie, kiedy agent przejdzie przez pierwsze drzwi należy porzucić wyszukaną ścieżkę i obliczyć ją na nowo. Algorytm ma następującą postać:

- Używając połączenia między pokojami znajdź najlepszą drogę z pokoju do pokoju przy użyciu algorytmu A*. W rezultacie otrzymamy ścieżkę przechodzącą przez kolejne pokoje.
- Znajdź drogę z położenia początkowego do położenia B.
- Niech agent przemieści się tylko do pokoju 2.
- Usuń pozostałą drogę i szukaj drogi do C.

- Niech agent przemieści się tylko do pokoju 5.
- Usuń pozostałą drogę i szukaj drogi do celu.
- Niech agent przemieści się do celu.

Hierarchiczne wyszukiwanie drogi to optymalizacja, która sprawdza się również dla dużych, otwartych przestrzeni. Tam, zamiast drzwi, można użyć kratek podziału przestrzeni. Należy najpierw wyszukać połączenia między kratką zawierającą startowe położenie oraz kratką zawierającą położenie końcowe. Następnie należy wyszukiwać drogi bezpośrednio w każdej kratce przez którą przechodzi agent. W ten sposób można również zminimalizować ilość przeszukiwanych stanów. Linie styku dwóch sąsiednich kratek można traktować jak drzwi.

Wyszukiwanie drogi jest bardzo często wykorzystywane w grach komputerowych. Niestety w grafice trójwymiarowej nie jest to zadanie trywialne. Sztuczna inteligencja musi posiadać dodatkowe informacje na temat otoczenia. Siatki nawigacyjne najczęściej tworzone są przez grafików w edytorze gry. W samej implementacji A* problemem może być efektywny wybór sąsiednich stanów.

4. Programowanie grafiki

4.1. Sprzętowy potok graficzny

Potokiem nazwiemy ciąg operacji wykonywanych równolegle w określonej kolejności. Każdy etap otrzymuje dane wejściowe z poprzedniego etapu i swoje wyście przekazuje do następnego etapu.

Każdy wierzchołek posiada pozycję oraz zazwyczaj kolor, drugi kolor (rozbłysku), jeden lub więcej zestaw koordynat tekstury, wierzchołek normalny, binormalny oraz styczny. Wektor normalny jest używany do obliczania oświetlenia natomiast wektory binormalny oraz styczny (wraz z normalnym tworzą przestrzeń styczną) wykorzystywane są przy mapowaniu wypukłości.

Przekształcenia wierzchołków są pierwszym etapem w sprzętowym potoku graficznym. Transformacje wierzchołków są to sekwencje działań matematycznych przeprowadzanych na każdym z wierzchołków. Do tych operacji zaliczają się przekształcenia pozycji wierzchołka do pozycji na ekranie przy użyciu rasteryzera, generowanie współrzędnych tekstury dla operacji teksturowania oraz oświetlenie dla uzyskania koloru wierzchołka.

Przekształcone wierzchołki trafiają do kolejnego etapu, w którym odbywa się łączenie ich oraz rasteryzacja. Najpierw z otrzymanych wierzchołków tworzy się kształt geometryczny na podstawie podanej kolejności łączenia wierzchołków. W ten sposób uzyskiwane są tzw. prymitywy (trójkąty, linie, punkty). Prymitywy te mogą wymagać przycięcia do widocznego obszaru (view frustum) w przestrzeni 3D widzianego z kamery, lub określonego przez aplikację. Rasteryzer może także odrzucić poligony (wielokąty), które są odwrócone tyłem lub przodem do kamery. Jest to tzw. pomijanie zbędnych obiektów (culling).

Poligony, które przeszły etap odcinania i usuwania zostaną poddane rasteryzacji. Jest to proces ustalający zbiór pikseli pokrytych przez prymityw (piksele, które wypełniają prymityw). Każdy typ prymitywu (trójkąt, punkt, linia) jest rasteryzowany oddzielnie według zasad rasteryzacji przeznaczonych dla konkretnego typu prymitywu. Wynikiem rasteryzacji są zbiory pozycji pikseli oraz zbiory fragmentów. Nie ma zależności między ilością wierzchołków z jakich składa się prymityw a ilością fragmentów generowanych podczas rasteryzacji tego prymitywu.

Piksel jest to skrót od picture element (od słowa pixel). Piksel jest najmniejszym fragmentem obrazu. Piksel może mieć różne właściwości (na przykład może reprezentować kolor lub głębię). Fragment jest natomiast potencjalnym stanem piksela

wymaganym do jego aktualizacji. Przez fragment można rozumieć najmniejszy element z jakiego składa się prymityw po rasteryzacji. Wszystkie dane fragmentu (kolor, kolor rozbłysku, koordynaty tekstury, wektor normalny, binormalny i styczny) są to dane uzyskane z interpolacji pomiędzy przekształconymi wierzchołkami. Na przykład, jeśli wierzchołek A ma kolor 0.0 a wierzchołek B ma kolor 1.0 to fragment znajdujący się dokładnie na środku odcinka AB będzie miał kolor 0.5. Jeśli fragment przejdzie testy rasteryzacji zostanie zaktualizowany piksel w buforze ramki.

Jak już prymityw zostanie zrasteryzowany do zbioru fragmentów następuje etap interpolacji parametrów fragmentów. Wykonywane jest teksturowanie, operacje matematyczne oraz obliczanie koloru każdego fragmentu. Dodatkowo do ustalenia wynikowego koloru fragmentu może zostać obliczona nowa głębia lub nawet piksel w buforze ramki (frame buffer), który odpowiada temu fragmentowi może nie zostać poddany aktualizacji (z tego powodu dla każdego fragmentu wejściowego powstaje jeden lub zero fragmentów wynikowych).

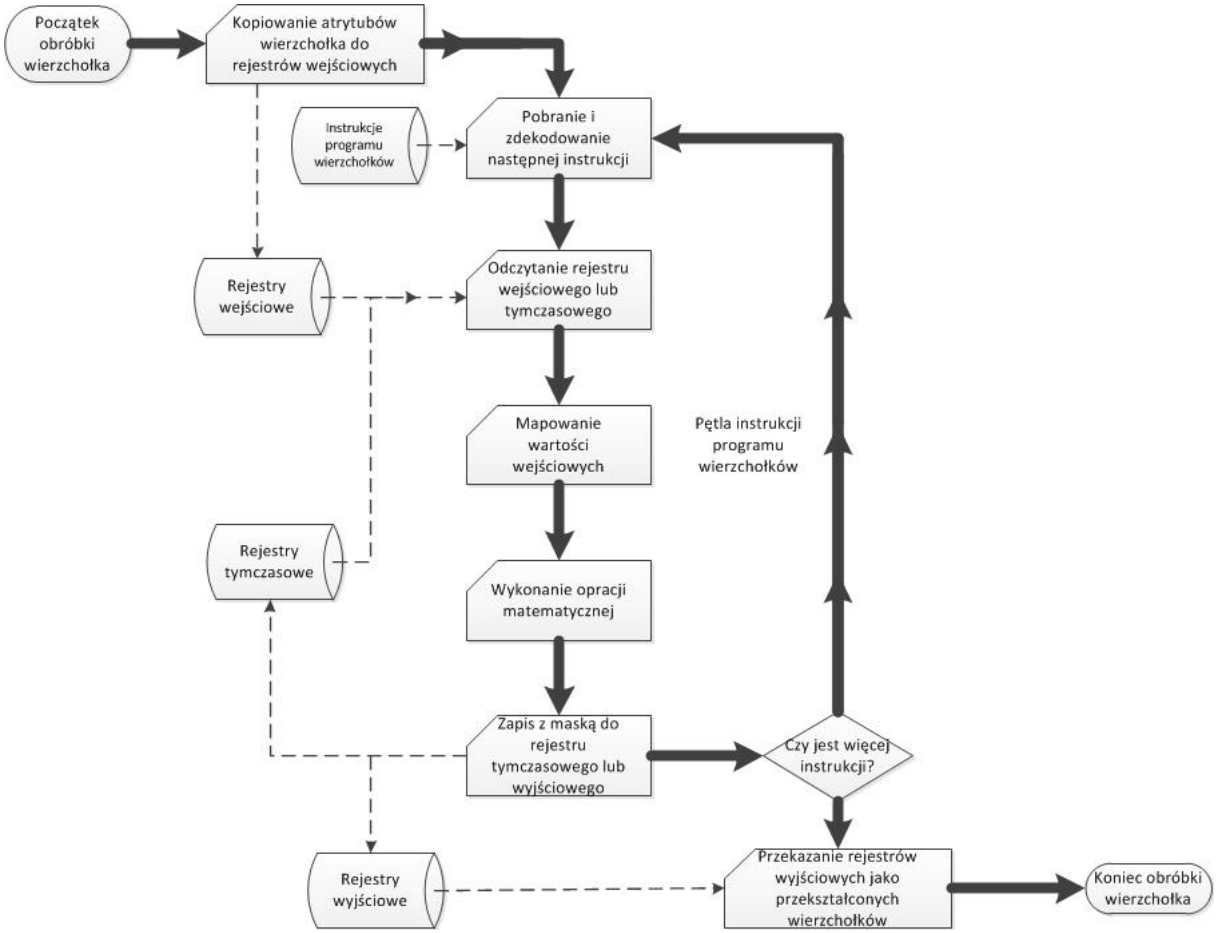
Operacje na rastrze są ostatnim etapem operacji na fragmentach. Wykonywane są zaraz przed aktualizacją bufora ramki. W tym etapie usuwane są niewidoczne powierzchnie (test głębokości), mogą być również wykonywane operacje przezroczystości (blending) oraz rzucanie cieni z użyciem bufora szablonu (stencil buffer). Operacje na rastrze wykonują serię testów dla każdego fragmentu, takie jak: scissor, alpha, stencil i depth. Testy te określają wynikowy kolor lub głębię fragmentu, pozycję piksela oraz dla każdego piksela wartość jego głębi i szablonu (depth i stencil values). Jeżeli jakikolwiek test się nie powiedzie fragment jest odrzucany i odpowiadający mu piksel nie zostanie zaktualizowany[17]. Jeśli test głębokości (depth test) się powiedzie wartość głębi piksela może zostać zastąpiona wartością głębi fragmentu. Po zakończeniu testów operacja mieszania (blending) łączy kolor fragmentu z kolorem piksela. Na końcu frame buffer (bufor ramki) zapisuje operację zastępując kolor piksela kolorem uzyskanym z operacji mieszania.

4.1.1. Programowalna jednostka wierzchołków

Model przepływu danych dla przetwarzania wierzchołków rozpoczyna się podczas ładowania każdego z atrybutów wierzchołka do jednostki wierzchołków (vertex procesor). Jednostka wierzchołków cyklicznie pobiera kolejne instrukcje i wykonuje aż do zakończenia programu wierzchołków. Instrukcje mają dostęp do różnych grup rejestrów, które zawierają wartości wektorów takie jak pozycja, wektor normalny lub kolor. Rejestry atrybutów wierzchołków są tylko do odczytu i zawierają informacje o wierzchołku dostarczone przez aplikację. Rejestry tymczasowe są do odczytu i do zapisu, można w nich przechowywać wyniki obliczeń. Rejestry wyjściowe służą tylko do zapisu. Program (aby był poprawny) musi zapisać do nich dane. Kiedy program wierzchołków kończy swoje działanie rejestry wyjściowe zawierają przetransformowany wierzchołek. Po określeniu trójkątów i rasteryzacji interpolowane wartości tych rejestrów są przekazywane do jednostki fragmentów.

[17] Aczkolwiek może nastąpić zapis do stencil buffer'a (bufora szablonu).

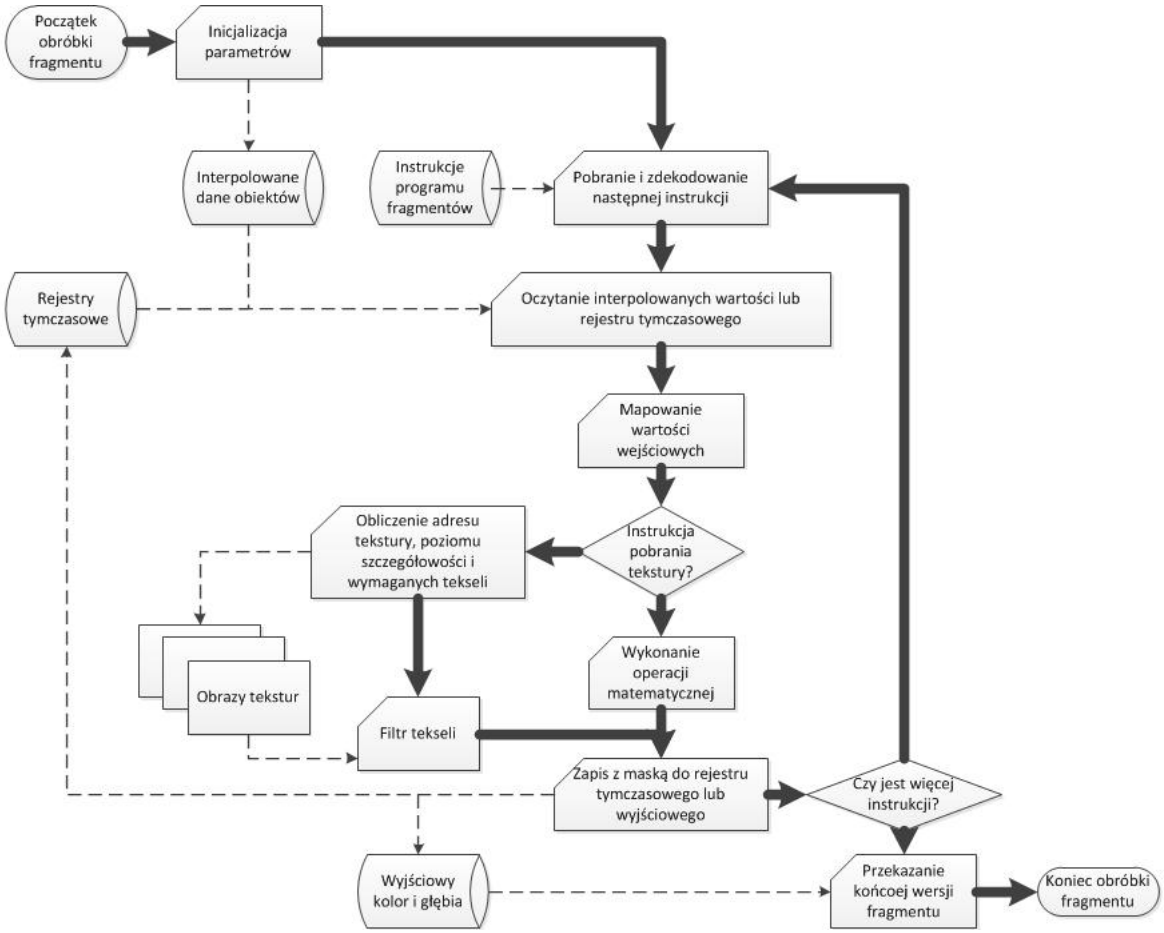
Większość działań na wierzchołkach używa określonej ilości operacji. Można wykonywać operacje matematyczne na wektorach dwu, trzy i cztero elementowych takie jak dodawanie, mnożenie, mnożenie z dodawaniem, iloczyn skalarny, minimum, maksimum, negacja, odejmowanie czy iloczyn wektorowy.



Rysunek 4.1.1. Diagram przepływu dla programowalnej jednostki wierzchołków.

4.1.2. Programowalna jednostka fragmentów

Programowalna jednostka fragmentów różni się od jednostki wierzchołków tym, że program fragmentów wykonuje się dla każdego fragmentu osobno, a więc tysiące razy więcej. Dostępne są również inne operacje. Wartością wyjściową jest wektor odpowiadający za kolor (opcjonalnie może być też zapisana głębia). Jednostka fragmentów dodatkowo operuje na teksturach. Ilość dostępnych tekstur jednorazowo jest uzależniona od sprzętu. Operacje na teksturach polegają na dostępie do obrazu na podstawie odpowiednich współrzędnych (przekazywane są one przez aplikację, ale można je dowoli zmieniać w programie wierzchołków i fragmentów). W jednostce fragmentów występują rejestry wejściowe, które są przeznaczone do odczytu. Wartości tych rejestrów zawierają interpolowane dane pobierane z jednostki wierzchołków.

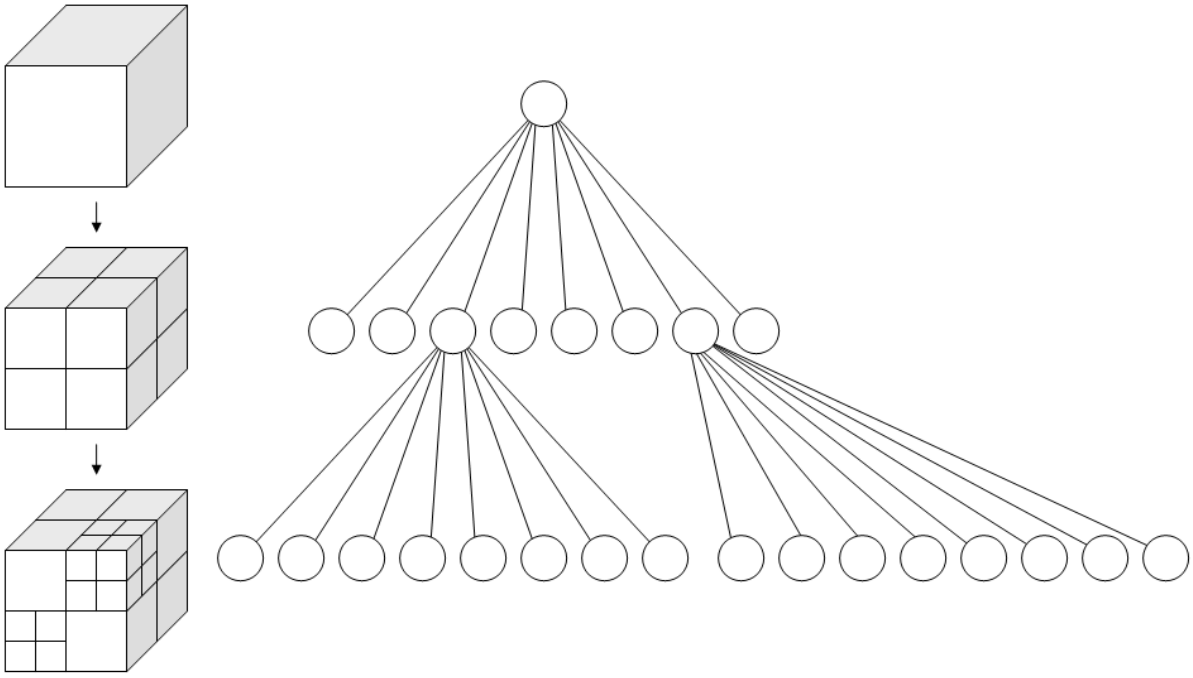


Rysunek 4.1.2. Diagram przepływu dla programowalnej jednostki fragmentów.

4.2. Podział geometrii

Gry komputerowe wykorzystują dużą ilość geometrii do reprezentowania świata. Jednocześnie renderowanych jest bardzo dużo modeli, a każdy z nich ma od kilkudziesięciu do kilkunastu tysięcy wierzchołków. Co prawda, dzisiejsze karty graficzne radzą sobie dobrze z renderowaniem dużej ilości geometrii i nie stanowi to dla nich wąskiego gardła. Niemniej jednak czym więcej, i czym bardziej złożone modele tym scena jest bardziej realistyczna, szczególnie w przypadku otwartych przestrzeni (lasy, miasta, itp.). Dlatego już od dawna (kiedy karty graficzne nie radziły sobie dobrze z renderowaniem większej ilości wierzchołków) zaczęto stosować algorytmy podziału geometrii. Dzięki takiemu podziałowi możliwe jest wyświetlanie tylko tej geometrii, która rzeczywiście jest widoczna. Jak się okazuje, nawet w dzisiejszych czasach daje to duży skok wydajności, a zaoszczędzone klatki animacji można przeznaczyć na inne rzeczy.

Jednym z pomysłów podziału jest drzewo ósemkowe. Drzewo ósemkowe dzieli scenę na sektory – sześciany. Każdy sześcian jest węzłem drzewa i posiada do ośmiu potomków. Korzeniem drzewa jest również sześcian, który otacza całą scenę.

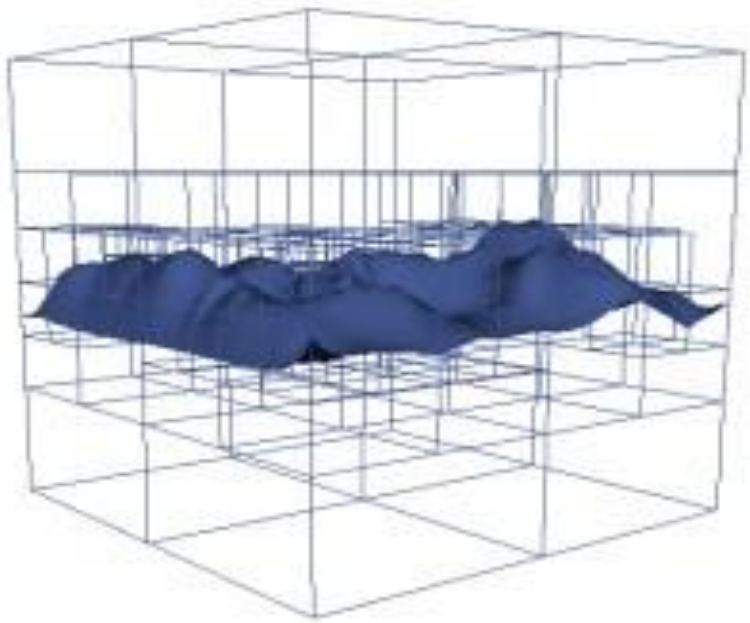


Rysunek 4.2.1. Konstrukcja drzewa ósemkowego. Każdy węzeł posiada do ośmiu potomków. Z lewej strony reprezentacja graficzna. Obrazek pobrany z [62].

Dzięki drzewom ósemkowym można szybko wyeliminować niewidoczną geometrię lub sprawdzać kolizje.

Każdy węzeł drzewa posiada osiem wierzchołków dzięki czemu ma swoją lokalizację w przestrzeni. Wierzchołek drzewa otacza całą scenę, więc posiada całą geometrię. Każdy węzeł drzewa posiada tylko część geometrii rodzica. Dla uproszczenia wielu obliczeń stosuje się Bounding Box'y (otaczający prostopadłościan) oraz Bounding Sphere'y

(otaczająca kula)[18] dla modeli trójwymiarowych. Wykorzystując samą kulę otaczającą można w prosty i szybki sposób umieścić model w odpowiednim węźle drzewa (oczywiście drzewo i scena są tworzone oddzielnie. Grafik tworzący scenę nie wie nic o drzewie. Geometria jest umieszczana w węzłach automatycznie podczas ładowania sceny w grze). Jak łatwo zauważyć drzewa ósemkowe nadają się przede wszystkim dla geometrii statycznej. Może zdarzyć się tak, że model będzie stał na granicy pomiędzy sąsiadującymi węzłami. Należy wtedy go umieścić w węźle będącym rodzicem obu sąsiadów. Taka sytuacja oczywiście nie jest dobra, ponieważ model znajdujący się w „wyższym" węźle będzie miał przeprowadzane o wiele więcej testów niż powinien. Niestety nie zawsze da się uniknąć takich sytuacji.



Rysunek 4.2.2. Wizualizacja drzewa ósemkowego. Obrazek pobrany z [63].

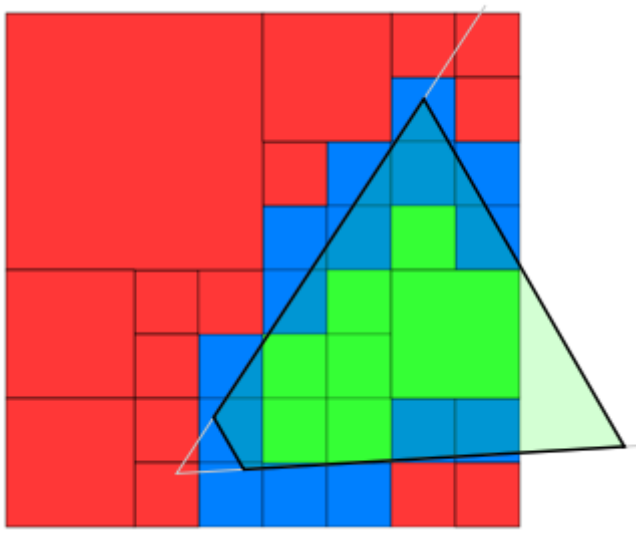
Ruchome modele w drzewie ósemkowym są problematyczne. Każdy węzeł drzewa ma listę geometrii, która się w nim znajduje. Jeśli chcemy przechowywać modele niestateczne w drzewie ósemkowym należy zastanowić się nad sposobem przechowywania modeli w węzłach. Jeżeli model przemieszcza się z jednego węzła do drugiego należy zapobiec sytuacji podwójnego renderowania tej geometrii. Sposobów może być klika. Można, na przykład użyć jakiegoś pojemnika, z którego będzie się usuwać model, gdy wyjdzie z węzła, a jeśli do niego wejdzie, będzie się go dodawać. Innym rozwiązaniem jest licznik klatki. Wówczas model może znajdować się w kilku węzłach, ale dzięki licznikowi zostanie wyrenderowany tylko raz.

Jeśli byśmy chcieli zaimplementować wykrywanie zderzeń drzewo ósemkowe dobrze się do tego nadaje ponieważ redukuje liczbę sprawdzeń. W tym celu każdy węzeł drzewa powinien posiadać dodatkowo listę sąsiadów (do sześciu sąsiadów, dla każdej ściany po jednym). Sąsiadem może być sześcian o boku równym lub większym. Żeby dwa boki

[18] Prostopadłościan otaczający jest to sześcian w którym zawiera się cały model. Dzięki temu wiadomo jaka jest wysokość i szerokość modelu. Jest to bardzo duże przybliżenie geometrii. Kula otaczająca spełnia tą samą funkcję. Otaczających figur (czasami stosuje się też walec) używa się np. do sprawdzania kolizji. Żeby nie sprawdzać od razu przecięć każdego trójkąta geometrii każdego modelu warto wykonać prosty test przecięcia z kulą a potem z prostopadłościanem i dopiero jeśli te testy przejdą należy sprawdzać przecięcia z konkretnymi trójkątami. Jak widać daje to duże przyspieszenie w procesie eliminacji zbędnych testów przecięcia.

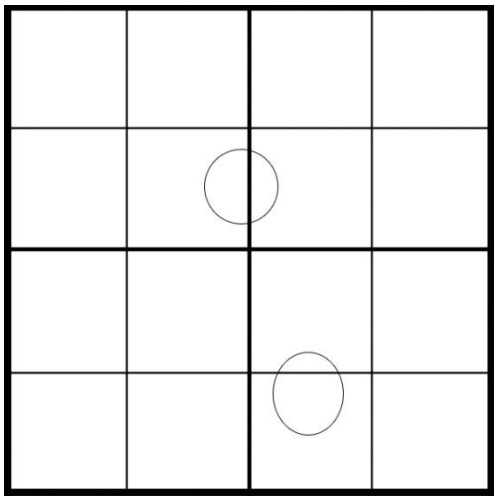
traktowane były jako sąsiedzi muszą spełniać pewne wymagania: normalne muszą wskazywać w przeciwne strony, wszystkie wierzchołki źródłowego boku znajdują się na lub wewnątrz docelowego boku oraz rozmiar źródłowego boku musi być mniejszy lub równy rozmiarowi docelowego boku.

Wszelkie operacje powinny być wykonywane tylko dla widocznych wielokątów. Test widoczności najlepiej jest zacząć od jak największego węzła. Jeżeli sześcian korzenia drzewa całkowicie pokrywa się z widocznym obszarem wówczas cała geometria powinna być rysowana. Jeśli tylko częściowo należy sprawdzać potomków i przerwać sprawdzanie jeśli jakiś węzeł nie jest widoczny.



Rysunek 4.2.3. Test widoczności w drzewie czwórkowym (lub ósemkowym widocznym od góry). Zielone obszary to węzły widoczne, niebieskie częściowo widoczne, czerwone nie widoczne. Obrazek pobrany z [62].

Jak zostało wcześniej wspomniane drzewa ósemkowe (lub ich dwuwymiarowe odpowiedniki – drzewa czwórkowe) mają jedną istotną wadę. Obiekt znajdujący się na przecięciu dwóch węzłów musi zostać włożony do węzła nadrzędnego.



Rysunek 4.2.3. Drzewo ósemkowe (lub czwórkowe) widziane z góry. Dwa obiekty przecinają węzły.

Rysunek powyżej obrazuje taką sytuację. Jeden obiekt zostanie dodany do korzenia drzewa, a drugi do (prawego, dolnego) potomka. Takie węzły zostały nazwane „lepkimi płaszczyznami", ponieważ zabierają obiekty, które nie powinny do nich należeć. Drzewa ósemkowe (lub czwórkowe) najlepiej sprawdzają się wtedy, gdy obiekt znajduje się w węźle najlepiej dopasowanym do siebie.

Jednym z możliwych rozwiązań powyższej sytuacji jest umieszczenie obiektu w węzłach, z którymi się przecina, ale jest to skuteczne jedynie w przypadku obiektów statycznych. Innym rozwiązaniem są swobodne drzewa ósemkowe. Tutaj sześciany otaczające węzły są zwiększane, tak, że nachodzą na siebie, ponieważ odległości między środkami węzłów pozostają bez zmian. Wzór na długość krawędzi sześcianu otaczającego w drzewie ósemkowym ma postać:

$$długość(głębia) = \frac{RozmiarŚwiata}{(2^{głębia})}$$

gdzie:

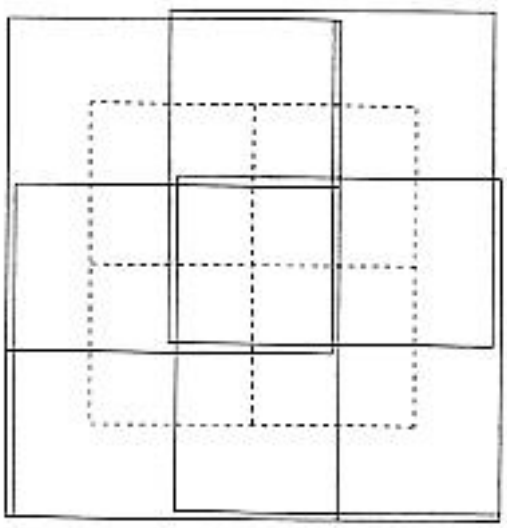
- długość to długość krawędzi sześcianu otaczającego
- głębia to poziom, na którym znajduje się węzeł (głębia dla korzenia wynosi 0)
- RozmiarŚwiata to długość krawędzi świata (zakładając, że świat jest kwadratowy)

Natomiast wzór na długość krawędzi sześcianu otaczającego w swobodnym drzewie ósemkowym ma postać:

$$długość(głębia) = k \cdot \frac{RozmiarŚwiata}{(2^{głębia})}$$

gdzie:

- k to pewna stała i k > 1



Rysunek 4.2.4. Widok z góry swobodnego drzewa ósemkowego. Obrazek pobrany z Game Programming Gems 1 – Loose Trees.

4.3. Oświetlenie

Z punktu widzenia realizmu generowanej sceny trójwymiarowej oświetlenie ma kluczowe znaczenie. Odpowiedni model oświetlenia pokazuje materiał, z jakiego oświetlany obiekt jest zbudowany.

Rysunek 4.3.9. Różne modele oświetlenia. Obrazek pobrany z [3].

API graficzne takie jak OpenGL i Direct3D mają wbudowany model oświetlenia o stałej funkcji. Jest to model parametryzowany kilkoma stanami. Niestety ten model sprawdza się tylko dla niewielkiej ilości materiałów. Dzięki programowalnym jednostkom wierzchołków i fragmentów programista jest w stanie stworzyć własny model oświetlenia dla obiektu. W modelu podstawowym wynikowy kolor piksela jest sumą koloru emisji, otoczenia, rozproszenia i rozbłysku.

Współczynnik emisji (emissive) jest niezależny od źródeł światła. Reprezentuje światło emitowane lub oddawane przez powierzchnię. Jest to kolor, jaki widzimy oglądając przedmiot, który nie jest w żaden sposób oświetlany. W prostszych modelach oświetlenia współczynnik emisji nie oświetla pobliskich obiektów (co w rzeczywistym świecie ma miejsce). Modele bardziej złożone wykorzystują emisję.

$$emissive = C_e$$

Równanie 1. Równanie na współczynnik emisji gdzie:
- C_e to kolor emisji materiału.

Współczynnik otoczenia (ambient) jest światłem rozproszonym. Nie pochodzi z konkretnego źródła, w związku z tym nie ma kierunku ani pozycji. Współczynnik ten zależy od tego jak materiał odbija światło oraz od koloru światła rzucanego na obiekt. Współczynnik otoczenia wyrażany jest jako iloczyn światła odbitego oraz światła globalnego (global ambient).

$$ambient = C_a \cdot C_{ga}$$

Równanie 2. Równanie na współczynnik otoczenia gdzie:
- C_a to kolor odbicia materiału.
- C_{ga} to kolor oświetlenia ogólnego.

Współczynnik rozproszenia (diffuse) jest to światło odbite we wszystkich kierunkach w jednakowym stopniu pochodzące ze światła posiadającego położenie oraz kierunek. Współczynnik rozproszenia symuluje powierzchnie o mikroskopijnej chropowatości. Natężenie światła odbitego jest proporcjonalne do kąta padania światła. Dla każdego punktu powierzchni wartość rozproszenia jest taka sama.

$$diffuse = C_d \cdot C_l \cdot \max\left(dot(V_n, V_l), 0 \right)$$

Równanie 3. Równanie opisujące współczynnik rozproszenia gdzie:

- $\boldsymbol{C_d}$ to kolor rozproszenia materiału
- $\boldsymbol{C_l}$ to kolor światła, które pada na powierzchnię
- $\boldsymbol{max}$ to funkcja maksimum
- $\boldsymbol{dot}$ to funkcja iloczynu skalarnego
- $\boldsymbol{V_n}$ to znormalizowany wektor normalny do powierzchni
- $\boldsymbol{V_l}$ to znormalizowany wektor skierowany w stronę źródła światła

Im bliżej źródło światła będzie wektora normalnego tym natężenie światła odbitego będzie większe. Dzieje się tak ze względu na iloczyn skalarny wektora normalnego i wektora skierowanego w stronę źródła światłą. Im mniejszy kąt między tymi wektorami tym większa ilość odbitego światła. Jeśli wektor światła jest zwrócony w tym samym kierunku co normalna wtedy iloczyn skalarny da ujemną wartość. Aby tak się nie stało należy zastosować funkcję maksimum. Dla takiego punktu nie powstanie kolor rozproszenia.

Współczynnik rozbłysku (specular) symuluje idealne odbicie od powierzchni (lustrzane). Za pomocą rozbłysku symuluje się gładkie, lśniące powierzchnie (np. wypolerowany metal). W przeciwieństwie do poprzednich współczynników ten zależy od kąta widzenia kamery. Jeśli kamera jest po przeciwnej stronie niż padające światło rozbłysk nie będzie widoczny (kamera musi być w zasięgu padających promieni światła aby rozbłysk był widoczny). Dochodzi tutaj dodatkowy czynnik taki jak poziom połyskliwości. Czym większy tym powierzchnia bardziej połyskliwa czyli punkt rozbłysku mniejszy (tak jak na rysunku 1 w oświetleniu Phong'a).

$$specular = C_s \cdot C_l \cdot F \cdot \left(\max(dot(V_n, V_h), 0 \right)^S$$

Równanie 4. Równanie symulacji rozbłysku gdzie:

- $\boldsymbol{C_s}$ to kolor rozbłysku materiału
- $\boldsymbol{C_l}$ to kolor światła
- $\boldsymbol{V_n}$ to znormalizowana normalna powierzchni
- $\boldsymbol{V_h}$ to znormalizowany wektor (wektor połówkowy) w połowie między wektorem zwróconym w stronę kamery (V_v) a wektorem zwróconym w stronę źródła światła (V_l)
- $\boldsymbol{F}$ to liczba, która jest równa 1 jeśli iloczyn skalarny normalnej i wektora zwróconego w stronę źródła światła jest większy od 0, lub w przeciwnym przypadku wynosi 0.
- $\boldsymbol{S}$ to współczynnik rozbłysku.

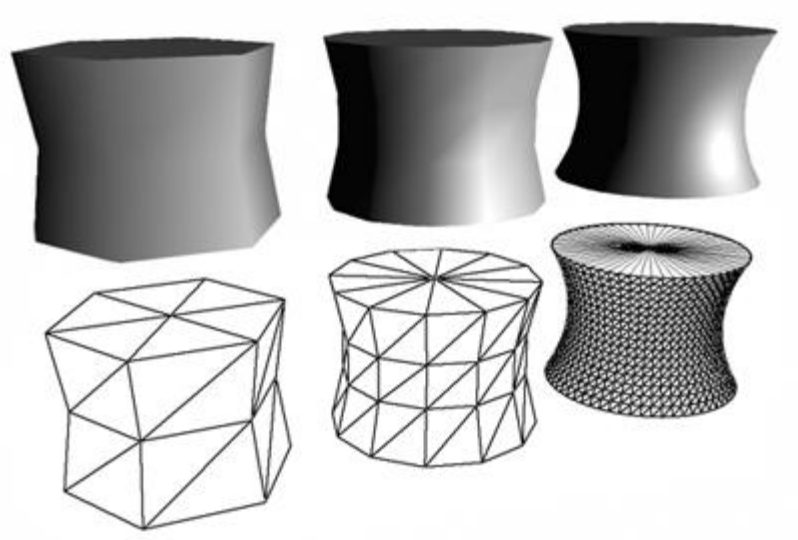
Jeśli kąt między wektorem zwróconym w stronę kamery V_v a V_h jest niewielki wówczas pojawia się rozbłysk. Gdy wektory V_n i V_h zaczynają się rozchodzić obliczony wykładnik z ich iloczynu zapewnia szybki zanik rozbłysku. Współczynnik rozbłysku może mieć wartość

zerową, gdy iloczyn skalarny między V_l i V_n jest ujemny. Dzięki temu jeśli normalna jest odwrócona tyłem do źródła światła nie otrzymamy rozbłysku.

Powyższy model oświetlenia należy do najprostszych. Nie uwzględnia nawet zaniku światła. Spójrzmy teraz na program wierzchołków implementujący ten model.

```
void Lighting(      float4 inPosition           : POSITION,
                    float3 inNormal             : NORMAL,
                    out float4 outPosition      : POSITION,
                    out float4 outColor         : COLOR,
                    uniform float4x4 modelViewProj,
                    uniform float3 Cga,
                    uniform float3 Cl,
                    uniform float3 vLightPosition,
                    uniform float3 vEyePosition,
                    uniform float3 Ce,
                    uniform float3 Ca,
                    uniform float3 Cd,
                    uniform float3 Cs,
                    uniform float S)
{
      outPosition = mul( modelViewProj, inPosition );

      float3 vPos = inPosition.xyz;
      float3 Vn = inNormal;

      float3 emissive = Ce;
      float3 ambient = Ca * Cga;

      float3 Vl = normalize( vLightPosition - vPos );
      float Cdiff = max( dot( Vn, Vl ), 0 );
      float3 diffuse = Cd * Cl * Cdiff;

      float3 Vv = normalize( vEyePosition - vPos );
      float3 Vh = normalize( Vl + Vv );
      float Cspec = pow( max( dot( Vn, Vh ), 0 ), S );

      if( Cdiff <= 0 )
            CSpec = 0;

      float3 specular = Cs * Cl * Cspec;

      outColor.rgb = emissive + ambient + diffuse + specular;
      outColor.a = 1;
}
```

Powyższy program prezentuje oświetlenie w przestrzeni obiektu. Po odpowiednich przekształceniach można to zrobić w przestrzeni kamery co jest bardziej użyteczne w przypadku kilku źródeł światła.



Rysunek 4.3.10. Oświetlenie dla wierzchołków. Czym gęstsza siatka tym efekt oświetlenia jest lepszy. Obrazek pobrany z [3].

Na obrazku przedstawione jest oświetlenie oparte na wierzchołkach. Trzy modele, każdy z różną ilością wierzchołków, pokazują jakość oświetlenia. Czym większy poziom szczegółowości (ilości wierzchołków) tym oświetlenie jest lepsze.

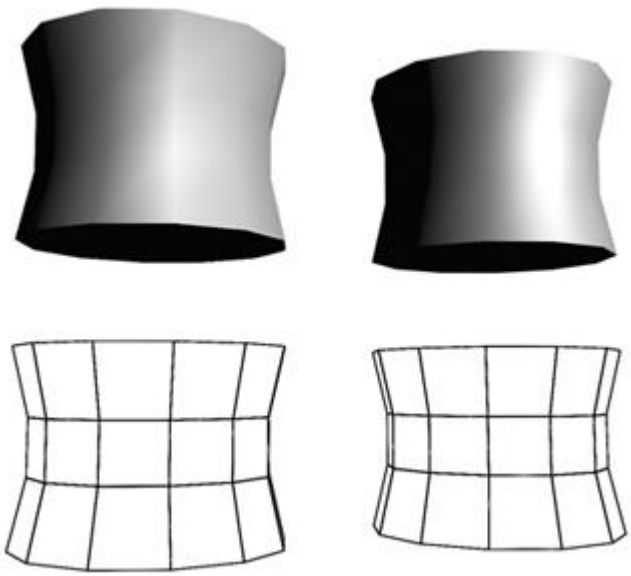
Pierwsza linijka kodu to zapis wynikowej pozycji wierzchołka. W programie wierzchołków zapisanie tej pozycji jest konieczne do prawidłowego działania programu. Następnie ustawiane są tymczasowe zmienne przechowujące wartości pozycji (wejściowej) i normalnej. Kolejne dwie linijki kodu to zastosowanie przedstawionych powyżej równań dla emisji i otoczenia. Później obliczany jest wektor światła. Jest to znormalizowany wektor (wektor musi być znormalizowany aby wskazywał kierunek potrzebny do obliczeń) o początku w pozycji wierzchołka a końcu w pozycji źródła światła (czyli jest to wektor skierowany w stronę źródła światła). Kolejna linijka przedstawia obliczenia współczynnika rozproszenia. Wbudowana funkcja `dot` (iloczyn skalarny) zwraca cosinus kąta między dwoma wektorami. W tym przypadku jest to wektor normalny i wektor skierowany w stronę źródła światła (obliczony powyżej). Zastosowana jest funkcja `max` ponieważ powierzchnia ustawiona tyłem do światła otrzyma oświetlenie ujemne, dlatego że wektor normalny będzie odwrócony od źródła światła. Zatem, aby uniknąć błędów z ujemną wartością iloczynu skalarnego, dzięki funkcji `max`, otrzymujemy wartość zerową dla rozproszenia. Wynikowy rezultat współczynnika rozproszenia otrzymujemy z równania mnożąc wcześniej otrzymany wynik z kolorem rozproszenia i kolorem światła. Następnie obliczany jest współczynnik rozbłysku. Wektor światła został policzony już wcześniej, wektor widoku (kamery) jest obliczany w analogiczny sposób. Wektor połowy kąta teoretycznie można policzyć jako znormalizowany wektor sumy połowy wektora światła ($0.5 \cdot V_l$) oraz połowy wektora widoku ($0.5 \cdot V_v$) . Takie rozwiązanie nie będzie wydajne, a normalizacja zniesie mnożenia (obcięcie wektorów do połowy). Okazuje się, że to działanie zapisać w prostszy sposób jako normalizację wektora będącego sumą wektora światła i wektora widoku. Podobnie jak w przypadku rozproszenia teraz też należy wykonać maksimum z iloczynu skalarnego wektora normalnego oraz wektora połowy kąta (obliczony linijkę wyżej). Dochodzi tutaj funkcja `pow`, która podnosi pierwszy parametr do potęgi zawartej w drugim parametrze. Wykładnikiem jest wartość rozbłysku. Ważną sprawą jest, by nie było rozbłysku, gdy wartość rozproszenia wynosi 0. Jeśli nie ma rozproszenia rozbłysk należy ustawić również na 0. Na koniec zapisujemy wartość

rozbłysku jako iloczyn koloru rozbłysku, koloru światła i wielkości rozbłysku. Wynikowy kolor wierzchołka zapisujemy jako sumę wcześniej wyliczonych współczynników.

Ten sam program można zastosować do obliczeń dla każdego fragmentu z osobna. Oto treść programu wierzchołków i fragmentów:

```
void VertexProgram( float4 inPosition          : POSITION,
                    float3 inNormal            : NORMAL,
                    out float4 outPosition     : POSITION,
                    out float3 outPos          : TEXCOORD0,
                    out float3 outNormal       : TEXCOORD1,
                    uniform float4x4 modelViewProj)
{
      outPosition = mul( modelViewProj, inPosition );
      outPos = inPosition.xyz;
      outNormal = inNormal;
}

void FragmentProgram(float4 inPosition         : TEXCOORD0,
                     float3 inNormal           : TEXCOORD1,
                     out float4 outColor       : COLOR,
                     uniform float3 Cga,
                     uniform float3 Cl,
                     uniform float3 vLightPosition,
                     uniform float3 vEyePosition,
                     uniform float3 Ce,
                     uniform float3 Ca,
                     uniform float3 Cd,
                     uniform float3 Cs,
                     uniform float S)
{

      float3 vPos = inPosition.xyz;
      float3 Vn = inNormal;

      float3 emissive = Ce;
      float3 ambient = Ca * Cga;

      float3 Vl = normalize( vLightPosition - vPos );
      float Cdiff = max( dot( Vn, Vl ), 0 );
      float3 diffuse = Cd * Cl * Cdiff;

      float3 Vv = normalize( vEyePosition - vPos );
      float3 Vh = normalize( Vl + Vv );
      float Cspec = pow( max( dot( Vn, Vh ), 0 ), S );

      if( Cdiff <= 0 )
            CSpec = 0;

      float3 specular = Cs * Cl * Cspec;

      outColor.rgb = emissive + ambient + diffuse + specular;
      outColor.a = 1;
}
```

Jak widać kod programu nie zmienił się za wiele. Różnica polega na tym, że teraz główny kod programu wykonywany jest w jednostce fragmentów. Oto rezultat:



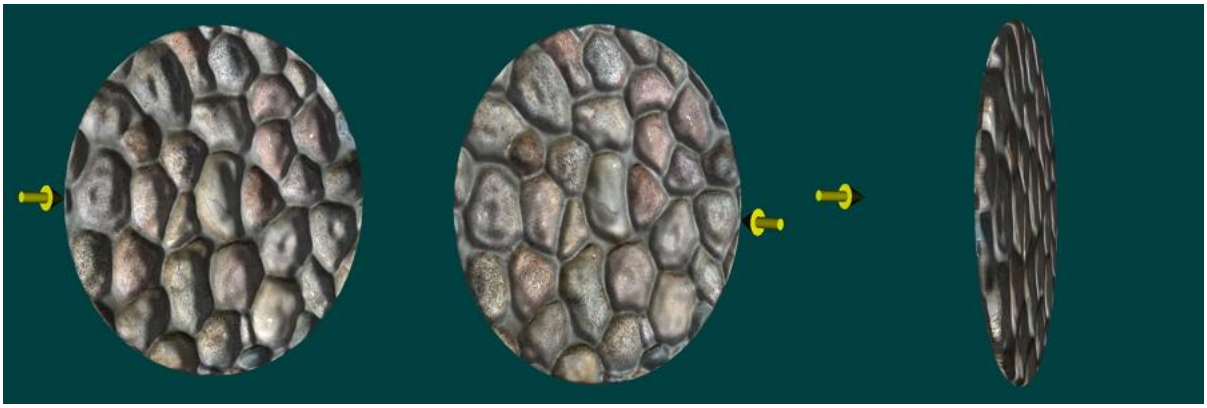
Rysunek 4.3.3. Z Lewej strony: model z oświetleniem dla wierzchołków, z prawej: ten sam model z oświetleniem dla pixeli. Obrazek pobrany z [3].

Mimo, że modele mają niski poziom szczegółowości oświetlenie jest bardzo dobre. Brane są interpolowane wartości normalnej i pozycji wierzchołka, następnie dla każdego fragmentu dane te są wykorzystywane do obliczenia oświetlenia. Dzięki temu można mieć niewielką ilość wierzchołków a jakość oświetlenia nadal będzie zadowalająca. Jest to bardzo ważna rzecz w programowaniu gier, gdzie ilość geometrii ma duże znaczenie.

4.4. Mapowanie nierówności

4.4.1. Bump Mapping

W grach komputerowych bardzo często symulowane są powierzchnie z dużą ilością wypukłości lub chropowatości. Dobrym przykładem może być ceglany lub kamienny mur czy kostka brukowa. Do symulacji takich powierzchni idealnie było by wymodelować, za pomocą programu graficznego, każdy kamień w murze. Niestety taki model, mimo idealnego wyglądu, miałby zbyt wiele wierzchołków by można było go umieścić w interaktywnej aplikacji (oczywiście w przypadku większej ilości takich powierzchni). Mapowanie nierówności jest sposobem zwiększenia szczegółowości powierzchni bez zwiększania ilości jej wierzchołków. Dzięki temu kamienny mur może być prostokątem złożonym z czterech wierzchołków jednocześnie posiadający niemal dowolną ilość kamieni, gdzie każdy kamień może mieć niemal dowolną chropowatość. Taki mur, dzięki odpowiedniemu oświetleniu będzie wyglądał jakby był naprawdę zrobiony z wielu kamieni, choć tak naprawdę (jeśli widz przyjży mu się z odpowiednio małego kąta) jest płaską powierzchnią.



Rysunek 4.4.11. Model okręgu z zastosowaniem mapowania normalnych. Te sam model widziany (lewa) z oświetleniem z lewej strony, (środek) z prawej strony oraz (prawa) widziany z boku. Przykład pobrany został z DirectX SDK (ParallaxOcclusionMapping).

Technika ta jest stosowana obecnie przy symulacji znakomitej większości powierzchni (nawet ludzkiej skóry). Można zastosować tutaj standardowy model oświetlenia opisany w poprzednim rozdziale. Różnica polega na zastosowaniu dodatkowej tekstury zwanej mapą normalnych. Tekstura ta, jak jej nazwa wskazuje, przechowuje normalne powierzchni w danym punkcie. Normalne te zapisane są jako kolor w formacie RGB lub RGBA (kanał alpha zwykle używany jest do przechowywania dodatkowych informacji przeważnie nie związanych bezpośrednio z normal mappingiem). W poprzednim rozdziale napisane jest, że normalna jest to wektor jednostkowy prostopadły do powierzchni. Innymi słowy, normalna wskazuje kierunek, w którym zwrócona jest powierzchnia. W standardowym modelu oświetlenia każdy wierzchołek przechowuje swoją normalną, natomiast jednostka fragmentów, do obliczenia oświetlenia, korzysta z interpolowanej

normalnej. Dzięki tym interpolowanym wartościom każdy fragment może mieć inny poziom oświetlenia. Niestety interpolowane normalne, dla danego polygonu, zawsze będą wskazywały podobny kierunek, więc nawet jeśli nałożymy na powierzchnię teksturę kamieni będą one płaskie. Każdy kamień z tej tekstury należało by potraktować jako osobny model. Mapa normalnych tworzona jest z odpowiadającej jej tekstury. Każdemu tekselowi oryginalnej tekstury (często nazywanej mapą koloru) odpowiada teksel mapy normalnych. Do symulacji wypukłości nie jest używana już interpolowana normalna tylko normalna zawarta w mapie normalnych. Jak wiadomo format RGB ma zakres wartości całkowitych [0, 255] natomiast wektor normalny jest z zakresu [-1, 1] w ciele liczb rzeczywistych. W procesorach graficznych kolor jest w zakresie [0, 1] w ciele liczb rzeczywistych. Wartości koloru odczytywane z tekstury są zmieniane z zakresu [0, 255] na [0, 1]. Tworząc mapę normalnych każdy wektor musimy zmapować z wartości [-1, 1] na [0, 1]. Można to zrobić stosując proste równanie:

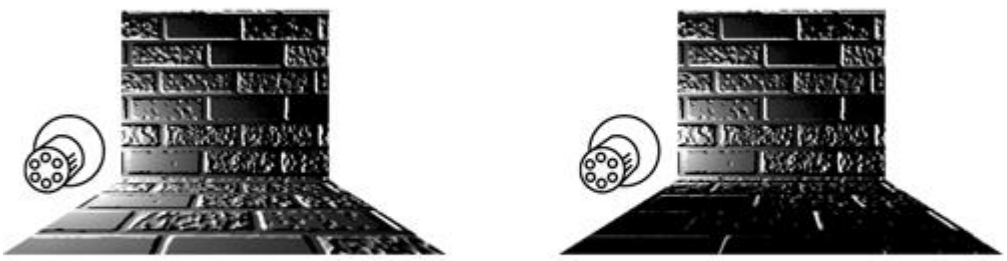
$$kolor = 0.5 \cdot normalna + 0.5$$

W programie musimy rozszerzyć z powrotem ten wektor do wektora z zakresu [-1, 1]. Oto równanie:

$$normalna = 2 \cdot (kolor - 0.5)$$

Dzięki temu można łatwo zakodować składowe wektora normalnego w składowych koloru w formacie RGB (x,y,z wektora odpowiadają składowym r,g,b koloru).

Do tej pory omawiany był model płaskiej ściany – prostokąta, który ma jedną normalną jednorodną (taką samą w każdym wierzchołku) (0,0,1). Dla takiego modelu można używać normalnych zawartych w mapie normalnych ponieważ współrzędne tekstury są powiązane z położeniem wierzchołków (dla dowolnego punktu prostokąta współrzędna tekstury różni się o dodatni współczynnik skali). Niestety, jeśli normalne nie są jednorodne a geometria jest bardziej złożona (czyli w znakomitej większości przypadków) oświetlenie może wydawać się poprawne, ale po bliższym przyjrzeniu się można odnaleźć błędy.



Rysunek 4.4.12. Rysunek przedstawia oświetlenie ściany oraz podłogi. Z prawej strony widać poprawne oświetlenie, z lewej błędne. Błędne oświetlenie wynika z różnych układów współrzędnych dla normalnych z mapy normalnych oraz wektora połowy kąta i światła. Obrazek pobrany z [3].

Można zauważyć, że oświetlenie nie jest poprawne względem położenia kamery oraz położenia źródła światła. Problem tkwi w tym, że wektor połowy kąta oraz wektor światła w przestrzeni obiektu nie znajdują się w tym samym układzie współrzędnych co wektory pobrane z mapy normalnych. Rozwiązaniem problemu jest przekształcenie wszystkich

wektorów równania oświetlenia do jednego układu współrzędnych. Od razu można pomyśleć o przestrzeni obiektu, ale to rozwiązanie posiada wiele ograniczeń. Podczas tworzenia mapy normalnych należy znać geometrię obiektu, co implikuje fakt, że nie można by używać jednej mapy normalnych dla wielu obiektów, które mogły by używać tej samej tekstury. Co więcej, dla obiektów animowanych, mapa normalnych musiała by być za każdym razem tworzona od nowa, zależnie od stanu animacji. Lepszym rozwiązaniem jest sprowadzenie wektorów do przestrzeni tekstury. W tym przypadku wystarczy, że wektor źródła światła oraz wektor połowy kąta przeniesiemy do przestrzeni tekstury (mapy normalnych).

Załóżmy, że mamy ceglaną ścianę oraz dokładnie taki sam model ale symulujący ceglaną podłogę. Dla ściany normalna wynosi (0,0,1), natomiast dla podłogi (0,1,0). Mapa normalnych dla przyjmuje wektor góry (0,0,1). Układ współrzędnych podłogi oraz mapy normalnych nie są identyczne zatem należy sprowadzić je do jednego wspólnego. Aby to zrobić potrzebujemy odpowiedniej macierzy obrotu:

$$[0 \quad 0 \quad 1] = [0 \quad 1 \quad 0]\begin{bmatrix}1 & 0 & 0\\ 0 & 0 & 1\\ 0 & -1 & 0\end{bmatrix}$$

W ten sposób można przenieść wektor góry z przestrzeni obiektu do wektora góry z mapy normalnych.

Załóżmy, że mamy macierz pozwalającą na przekształcenie wektorów z przestrzeni obiektu do przestrzeni tekstury. Dzięki tej macierzy można przenieść wektor połowy kąta oraz źródła światła do przestrzeni mapy normalnych. Jeśli V_l jest wektorem źródła światła to ten wektor w przestrzeni tekstury (V_l') obliczamy następująco

$$V_l' = \begin{bmatrix}V_{l_x}' & V_{l_y}' & V_{l_z}'\end{bmatrix} = \begin{bmatrix}V_{l_x} & -V_{l_z} & V_{l_y}\end{bmatrix} = \begin{bmatrix}V_{l_x} & V_{l_y} & V_{l_z}\end{bmatrix}\begin{bmatrix}1 & 0 & 0\\ 0 & 0 & 1\\ 0 & -1 & 0\end{bmatrix}$$

Dla wektora połowy kąta należy wykonać analogiczną operację.

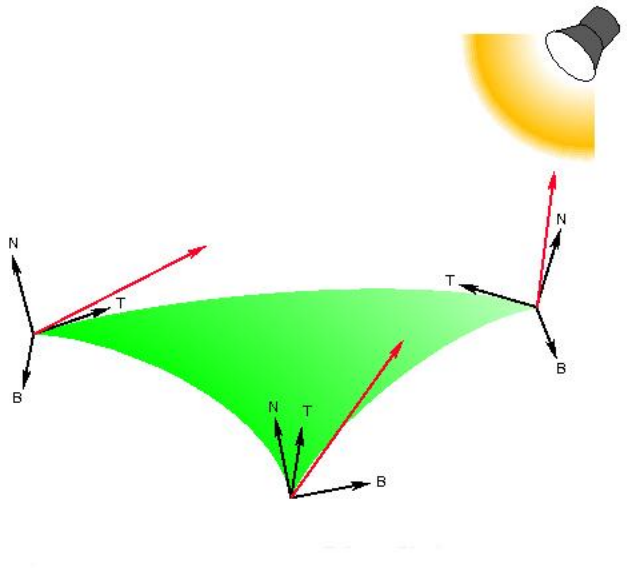
Aby przenieść wektory w przestrzeni obiektu do przestrzeni tekstury potrzebujemy macierzy 3x3, której kolumny złożone są z trzech wektorów: stycznego (V_t), dwunormalnego (V_b) i normalnego (V_n). Macierz taka ma następującą postać:

$$\begin{bmatrix}V_{t_x} & V_{b_x} & V_{n_x}\\ V_{t_y} & V_{b_y} & V_{n_y}\\ V_{t_z} & V_{b_z} & V_{n_z}\end{bmatrix}$$

Znając dowolne dwa wektory można obliczyć trzeci stosując iloczyn wektorowy:

$$V_n = V_b \times V_t$$
$$V_b = V_n \times V_t$$
$$V_t = V_n \times V_b$$

Łatwo zauważyć, że te trzy wektory są do siebie prostopadłe. Wektory z przestrzeni stycznej mogą być wyznaczone przez programy graficzne (np. 3D Studio Max) i wyeksportowane do pliku z modelem. Aplikacja może przekazać do programu wierzchołków wszystkie trzy wektory, więc nie trzeba stosować iloczynu wektorowego[19].



Rysunek 13. Rysunek przedstawia przestrzeń tekstury dla powierzchni. Rysunek pobrany z [86].

Na powyższym rysunku widać pewną powierzchnię oświetlaną jednym światłem. Czerwone (najdłuższe) strzałki są to wektory kierunku źródła światła. Wektor T to wektor styczny (tangent), B to wektor dwunormalny (binormal), a N to wektor normalny (normal) do powierzchni w danym punkcie.

Spójrzmy na przykładowy program wierzchołków i fragmentów.

```
struct VS_INPUT
{
      float4      position                : POSITION;
      float2      texCoord                : TEXCOORD0;
      float3      vNormal                 : NORMAL;
      float3      vBinormal               : BINORMAL;
      float3      vTangent                : TANGENT;
};

struct VS_OUTPUT
{
    float4 position          : POSITION;
    float2 texCoord          : TEXCOORD0;
    float3 vLightTS          : TEXCOORD1;
    float3 vViewTS           : TEXCOORD2;
    float3 vHalfAngle         : TEXCOORD3;
};
```

---

[19] Należy się zastanowić tutaj nad kwestią wydajności. Czym więcej danych aplikacja przesyła do shader'ów tym wolniej działa, dlatego czasem jedna operacja iloczynu wektorowego może być szybsza niż przekazanie wektora do shader'a.

Są to dwie struktury pomocnicze, przekazywane do programu wierzchołków (VS_INPUT) oraz do programu fragmentów (VS_OUTPUT). Program wierzchołków przyjmuje informacje na temat wierzchołka o jego pozycji, współrzędnych tekstury, normalnej, dwunormalnej oraz stycznej. Program fragmentów przyjmuje na wejście to co program wierzchołków podaje na wyjście czyli: pozycję wierzchołka, współrzędne tekstury, wektor światła w przestrzeni stycznej, wektor widoku w przestrzeni stycznej i wektor połowy kąta.

```
VS_OUTPUT VertexShaderMain(   VS_INPUT In,
                              uniform float3 vLightPos,
                              uniform float3 vEyePos,
                              uniform float4x4 mtxWVP)
{
      VS_OUTPUT Out;

      Out.position = mul( mtxWVP, In.position);
      Out.texCoord = In.texCoord;


      float3 vLightDir = normalize( vLightPos.xyz - In.position.xyz );
      float3 vEyeDir = normalize( vEyePos.xyz - In.position.xyz );
      Out.vHalfAngle = normalize( vLightDir + vEyeDir );

      float3x3 mtxRotation = float3x3(    In.vTangent,
                                          In.vBinormal,
                                          In.vNormal );

      Out.vLightTS = normalize( mul( mtxRotation, vLightDir ) );
      Out.vViewTS = normalize( mul( mtxRotation, vEyeDir ) );

      return Out;
}
```

Program wierzchołków prócz struktury opisanej wyżej przyjmuje trzy dodatkowe argumenty: pozycja światła, pozycja kamery (lub oka) oraz macierz model-widok-projekcja (world-view-projection). Najpierw obliczana jest pozycja wierzchołka oraz zapisywane są współrzędne tekstury jako dane wyjściowe. Następnie obliczany jest wektor światła, wektor kamery (oka) oraz wektor połowy kąta. Z wektora stycznego, dwunormalnego i normalnego tworzona jest macierz obrotu. Następnie obliczane są nowe wektory światła i widoku w przestrzeni stycznej.

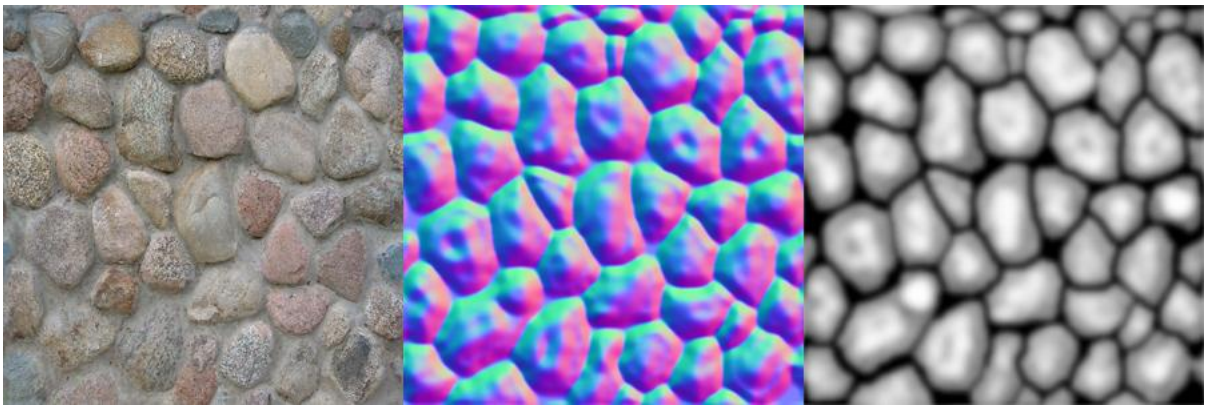
```
float4 PixelShaderMain( VS_OUTPUT In,
                        uniform sampler2D texBase     : TEXUNIT0,
                        uniform sampler2D texNormal   : TEXUNIT1,
                        uniform float4    cLightAmbient,
                        uniform float4    cLightDiffuse,
                        uniform float4    cLightSpecular): COLOR0
{
      float4 emissive = 0.1;
      float fSpecularExponent = 128.0f;

      float3 vNormal = normalize((tex2D(texNormal, In.texCoord) - 0.5) *2);
```

```
    float3 vL = normalize( float3(In.vLightTS.x,
                                  -In.vLightTS.y,
                                  In.vLightTS.z) );

    float fDiffuseLight = saturate( dot( vL, vNormal ) );
    float3 diffuse = cLightDiffuse * fDiffuseLight;

    float3 vV = normalize( In.vViewTS );
    float3 vH = normalize( In.vHalfAngle );

    float4 cSpecular = 0;

    float3 vReflectionTS = normalize( 2 * dot( In.vViewTS, vNormal ) *
    vNormal - In.vViewTS );

    float fRdotL = saturate( dot( vReflectionTS, vL ));
    cSpecular = saturate(pow(fRdotL, fSpecularExponent))*cLightSpecular;

    float4 color = float4(emissive +cLightAmbient +diffuse +cSpecular,1);

    return color * tex2D(texBase, In.texCoord);

}
```

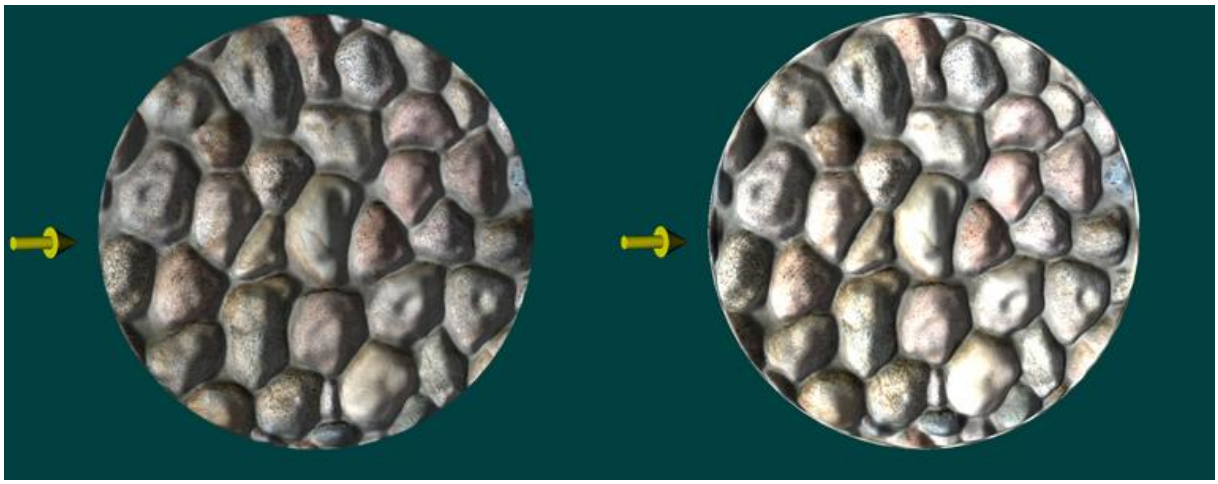
Powyższy program fragmentów przyjmuje dodatkowo bazową teksturę (mapę koloru), mapę normalnych, kolor światła otoczenia, rozproszenia oraz rozbłysku. Dla uproszczenia kolor światła emitowanego oraz poziom rozbłysku jest ustawiany na sztywno. W kolejnej linijce pobierana jest normalna z mapy normalnych (od razu jest rozszerzana do zakresu [-1,1]). Następnie obliczany jest wektor światła. Jest to wektor światła w przestrzeni stycznej ale z odwróconą współrzędną y. Kolejne dwie linijki obliczają światło rozproszenia. Wektory `vV` i `vH` są to odpowiednio wektor widoku i wektor połowy kąta. Obliczane są z wektora widoku z przestrzeni stycznej oraz wektora połowy kąta. Następnie obliczany jest rozbłysk. W dwóch ostatnich linijkach obliczany jest finalny kolor światła oraz ten kolor mieszany jest z bazową teksturą.

4.4.2. Parallax Mapping

Lepszą techniką mapowania wypukłości jest technika parallax mapping. Dzięki niej można uzyskać jeszcze doskonalszy efekt wypukłości. Technika ta bazuje na bump mapping'u, potrzeba tutaj również przestrzeni stycznej, wiele obliczeń jest takich samych. Dochodzi natomiast mapa wysokości. Jest to tekstura, która podobnie jak mapa normalnych przechowuje pewne dane dotyczące geometrii na płaszczyźnie. Mapa wysokości (height map) przechowuje informacje jak bardzo wypukła jest powierzchnia w danym punkcie. Czym jaśniejszy kolor, tym punkt jest bardziej wypukły. Wartości te można przechowywać jako składowa alfa mapy normalnych.

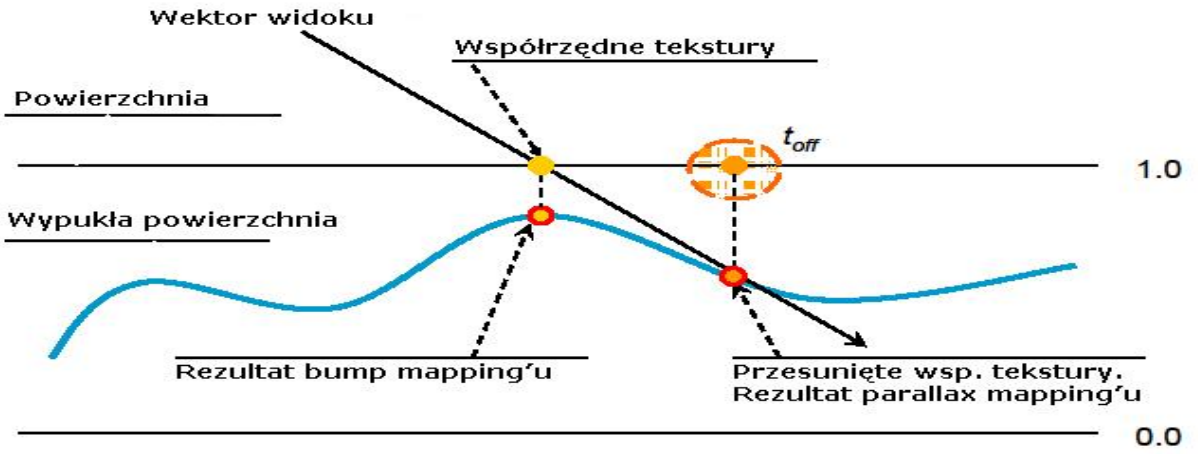


Rysunek 4.4.14. Od lewej: mapa koloru, mapa normalnych, mapa wysokości.



Rysunek 4.4.5. Porównanie technik mapowania wypukłości. Z lewej bump mapping, z prawej parallax mapping. Przykład pobrany został z DirectX SDK (ParallaxOcclusionMapping).

Kolejną innowacją jest przesuwanie współrzędnych tekstury zależnie od punktu widzenia kamery. W technice bump mapping'u widać było dokładnie ten teksel, na który wskazywały aktualne współrzędne fragmentu. W parallax mapping dzieje się nieco inaczej. Rysunek 4.4.6 ilustruje to.



Rysunek 4.4.15. Rysunek przedstawia schemat działania parallax mappingu. Zalerznie od wektora widoku wsp. tekstury są przesuwane w rezultacie czego pokazywany jest inny texel na powierzchni. Rysunek pobrany z [32].

Patrząc na powierzchnię, normalnie, zobaczymy punkt żółty (z obrazka), czyli texel odpowiadający współrzędnym tekstury. Może on być odpowiednio oświetlony dzięki odpowiadającemu mu tekselowi mapy normalnych (punkt żółty w czerwonej obwódce, pod punktem żółtym). Tak naprawdę patrząc z tej perspektywy powinniśmy zobaczyć zupełnie inny punkt (punkt pomarańczowy w czerwonej obwódce), dlatego należy przesunąć odpowiednio współrzędne tekstury (punkt t_{off}). Do obliczenia przesuniętych współrzędnych tekstury potrzeba trzech komponentów: oryginalne współrzędne tekstury, wektor widoku w przestrzeni tekstury oraz wartość wysokości z pola wysokości.

Niech P będzie punktem przesunięcia współrzędnych tekstury, V to znormalizowany wektor widoku w przestrzeni tekstury, h to wartość pola wysokości w punkcie T_0, który jest oryginalnymi współrzędnymi tekstury. Wysokość jest skalowana przez współczynnik s i przesuwana o wartość b, aby przekształcić wartość z zakresu [0, 1] do wartości pasujących do renderowanego świata[20]. Współczynnik skali najlepiej jest dobrać odpowiednio do symulowanej powierzchni. Dla przykładu można podać mur o wielkości 2x2 metra i grubości 0.2 metra. Odpowiednia skala dla takiego muru to będzie:

$$s = \frac{0.2}{2}$$
$$s = 0.1$$

Wartość przesunięcia b najlepiej jest obliczyć następująco:

$$b = s \cdot -0.5$$

Okazuje się, że wartość średnia s i b nadaje się najlepiej dla większości tekstur. Przeskalowana i przesunięta wartość wysokości ma następujący wzór:

$$h_{sb} = h \cdot s + b$$

Przesunięcie jest obliczane poprzez poprowadzenie wektora równoległego do powierzchni (polygonu) z punktu na powierzchni dokładnie nad punktem P do wektora widoku. Ten nowy wektor jest przesunięciem i może być dodany do T_0 aby otrzymać nowe współrzędne tekstury T_n.

$$T_n = T_0 + (h_{sb} \cdot \frac{V_{\{x,y\}}}{V_{\{z\}}})$$

Nowych współrzędnych tekstury używamy aby pobrać odpowiedni texel z mapy koloru oraz normalną z mapy normalnych.

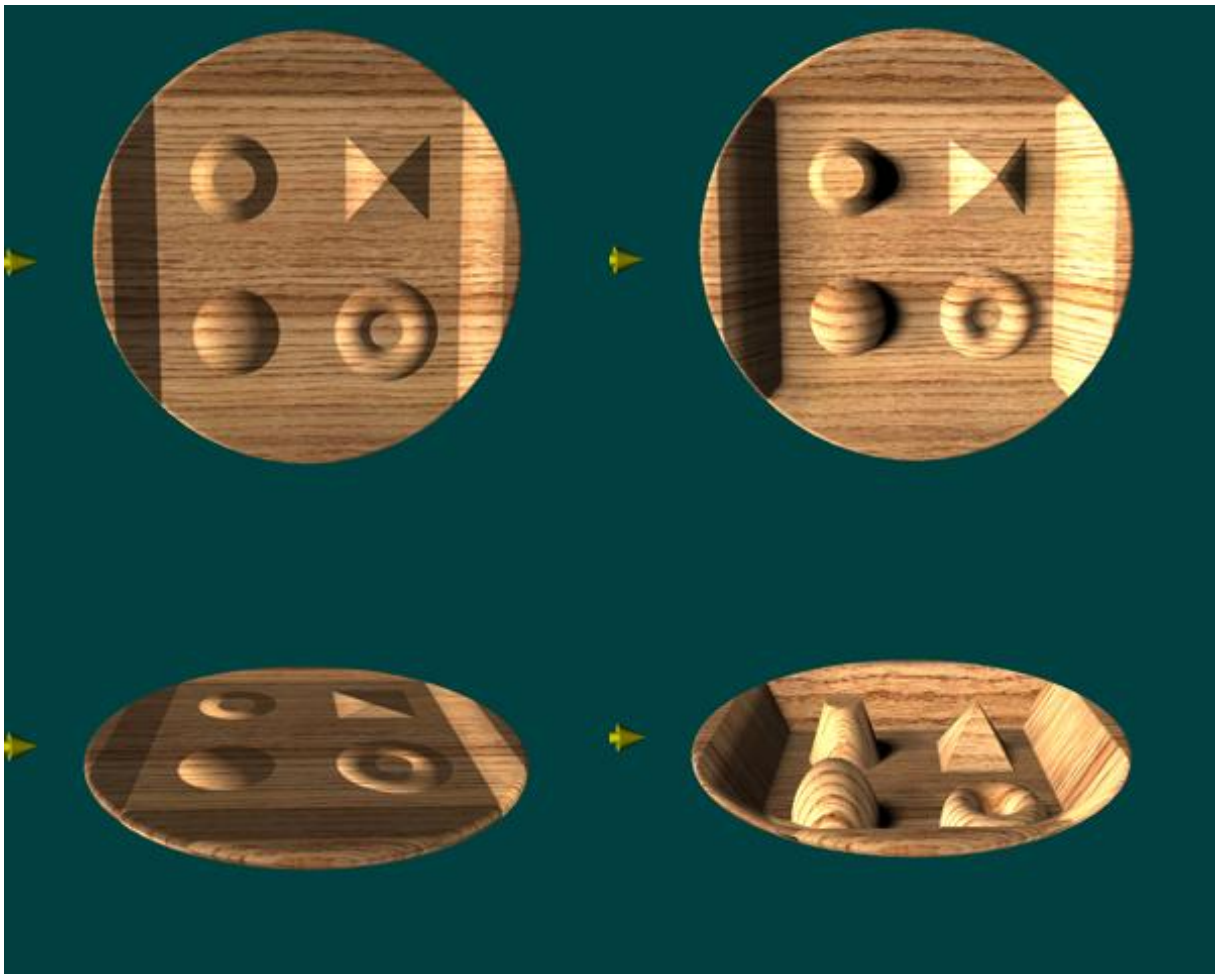
Kiedy kamera jest ustawiona pod ostrym kątem do powierzchni przesunięcia współrzędnych tekstury zmniejszają się. Oznacza to, że T_n zbliża się do T_0 czyli wartości przesunięcia zbliżają się do nieskończoności. Stanowi to problem, który jest widoczny jako piksele, które nie przypominają oryginalnej tekstury. Rozwiązaniem tego problemu

[20] Chodzi tu o wartość wysokości dla np. kamieni (rysunek 4). Wysokości (wypukłości) kamieni z zakresu [0,1] mogą być zbyt małe lub zbyt duże aby je realistycznie odwzorować.

może być ustawienie maksymalnej wartości przesunięcia współrzędnych na wartość h_{sb}. Może być wybrana dowolna inna stała wartość, ale użycie h_{sb} skraca długość shadera o dwie instrukcje. Nowe równanie dla przesunięcia współrzędnych tekstury wygląda następująco:

$$T_n = T_0 + (h_{sb} * V_{\{x,y\}})$$

W tym przypadku wektor widoku nie jest „normalizowany”, używamy tylko jego współrzędnych x i y.



Rysunek 4.4.7. Lewa kolumna: bump mapping widziany od góry i pod kątem. Prawa kolumna: parallax mapping widziany od góry i pod kątem. Przykład pobrany został z DirectX SDK (ParallaxOcclusionMapping).

4.5. Efekty przetwarzania obrazów (post-process)

Niemal każda dzisiejsza gra komputerowa wykorzystuje efekty specjalne w celu polepszenia wyświetlanej grafiki. Wiele świetnych efektów specjalnych można uzyskać przetwarzając wygenerowany obraz, czyli już po wyrenderowaniu sceny. Cała sztuczka polega na wyrenderowaniu sceny do obrazu, a nie od razu na ekran. Następnie ten obraz (powinien być w rozdzielczości ekranu) nakładany jest na ekran. W praktyce, wszystko co widzimy w grze to jest po prostu płaski obraz, który równie dobrze może być widzialny z boku, pod pewnym kątem. W ten sposób może być generowany, na przykład, interfejs użytkownika systemu operacyjnego.

Rysunek 4.5.16. Beryl, graficzny manager okien dla środowiska graficznego X Window System.

4.5.1. Rozmycie

Jednym z pierwszych efektów post-process w grach komputerowych był efekt rozmycia. Nie jest to do końca efekt realistyczny, ale w grach nadaje baśniową atmosferę, oraz wygładza nieco ostre krawędzie geometrii.

Rysunek 4.5.17. Z lewej: scena z wyłączonym rozmyciem. W prawej: ta sama scena z włączonym dużym rozmyciem.

Rysunek powyżej prezentuje efekt rozmycia obrazu techniką Gauss'a, która zostanie opisana w tym podrozdziale.

Aby wyrenderować scenę do tekstury (czyli obrazu) należy dynamicznie stworzyć teksturę o rozdzielczości ekranu i ustawić rysowanie nie do ekranu, a do tej tekstury. Następnie należy stworzyć prostokąt (z nałożoną wcześniej wygenerowaną teksturą) i rozciągnąć go na cały ekran. Bardzo dobrze użyć do tego programu wierzchołków oraz fragmentów. Jeśli mamy już wyświetlony prostokąt z nałożoną teksturą sceny możemy przystąpić do obróbki tego obrazu. Wbrew pozorom, nie tylko program fragmentów będzie do tego potrzebny. Do niektórych efektów (takich jak rozmycie) bardzo przydatny okazuje się program wierzchołków. Co prawda, nie manipuluje wierzchołkami (jedynie ustawia je w odpowiedniej pozycji, tak, aby prostokąt był rozciągnięty na cały ekran), ale ustawia pewne zmienne potrzebne przy obróbce pikseli.

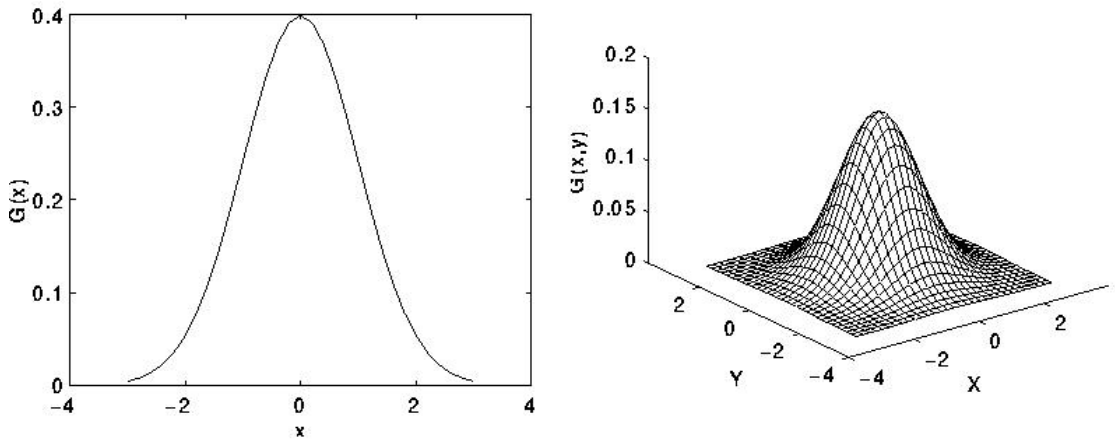
Rozmycie Gaussa jest efektem użycia funkcji Gaussa na obrazie w celu obliczenia transformacji dla każdego piksela. W jednym wymiarze funkcja Gaussa ma postać:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

natomiast w dwóch wymiarach:

$$G(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

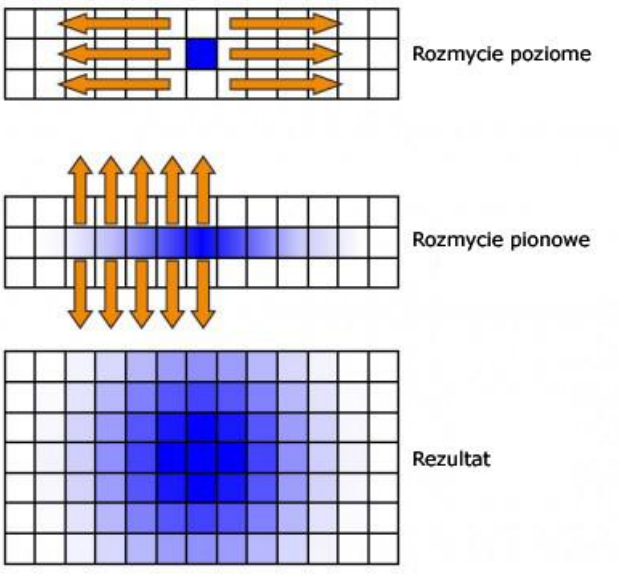
gdzie x jest odległością od początku poziomej osi, y jest odległością od początku osi pionowej, σ jest standardowym odchyleniem, które mówi o tym, jak bardzo funkcja jest rozciągnięta. Funkcja dwuwymiarowa daje w efekcie powierzchnię, której kontury są współśrodkowymi okręgami. Wartości obliczone funkcją Gauss'a tworzą macierz, której używamy do obliczenia wartości każdego piksela. Każdą nową wartością piksela jest średnia ważona jego sąsiadów. Aktualnie obliczany piksel otrzymuje najwyższą wartość, a jego sąsiedzi coraz mniejsze, w miarę zwiększania się ich odległości.



Rysunek 4.5.3. Z lewej: rozkład Gauss'a funkcji jednowymiarowej o środku w punkcie 0 oraz σ= 1. W prawej strony: rozkład Gauss'a funkcji dwuwymiarowej o środku w punkcie (0,0) oraz σ = 1.

Teoretycznie funkcja Gauss'a nigdy nie da wartości 0, co oznacza, że należy włączyć cały obraz do obliczeń dla każdego piksela. W praktyce, gdy obliczamy wartości funkcji Gauss'a piksele w odległości większej niż 3σ mogą być ignorowane, ponieważ ich wartości zbliżają się do 0. Oznacza to, że wystarczy używać macierzy o wymiarach $\lceil 6\sigma \rceil \times \lceil 6\sigma \rceil$, aby zapewnić efekt wystarczająco podobny do tego, jaki byśmy uzyskali stosując rozkład Gauss'a dla każdego piksela.

Charakterystyczną cechą rozkładu Gauss'a, jest to, że funkcja dwuwymiarowa może być zastąpiona przez dwie niezależne funkcje jednowymiarowe. Wynika to z symetrycznej, względem okręgu, natury funkcji Gauss'a, które są liniowo separowalne. Zamiast używać dwuwymiarowej macierzy, można skorzystać z dwóch jednowymiarowych. Pierwszą macierz stosujemy do rozmycia poziomego, a drugą do rozmycia pionowego.



Rysunek 4.5.18. Rozmycie Gauss'a przy użyciu dwóch funkcji jednowymiarowych. Obraz rozmywany jest najpierw poziomo, następnie, rozmyty poziomo obraz rozmywany jest pionowo. Obrazek jest własnością ATI.

Przykładowe wartości dla $\sigma = 1$ to:

$$G(0) = 0.3989422804$$
$$G(1) = G(-1) = 0.2419707245$$
$$G(2) = G(-2) = 0.0539909665$$
$$G(3) = G(-3) = 0.0044318484$$

Gdzie:

- G to jednowymiarowa funkcja Gauss'a.

Czym więcej wykonamy próbek, tym lepszą otrzymamy precyzję. Zazwyczaj używa się 13 lub 15 próbek. Większa ilość może znacznie spowolnić aplikację, a mniejsza da gorsze efekty. W punkcie 0 funkcja daje największą wartość i jest to wartość dla aktualnie obliczanego piksela ponieważ jest to wartość środkowa.

Funkcja Gauss'a daje wartości symetryczne, dlatego algorytm może policzyć tylko połowę wag (w praktyce wartości otrzymywane z funkcji Gauss'a stanowią wagi dla pikseli). Aby wykorzystać sąsiednie piksele należy użyć macierzy przesunięcia. Macierz ta ma dokładnie tyle samo elementów co macierz wag. Macierz przesunięć może składać się z liczb całkowitych z przedziału $[-k, k]\ gdzie\ k \in Z\ i\ k = \frac{n}{2}$, gdzie n to liczba elementów macierzy wag (w tym przypadku chodzi nam o macierze jednowymiarowe – tablice). Macierz przesunięć dla macierzy 13 wag ma postać:

$$M_p = [-6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6]$$

Dzięki elementom tej macierzy wiadomo, które piksele (tj. jak bardzo oddalone od głównego, czyli tego na pozycji 7 w macierzy) należy brać pod uwagę podczas rozmywania aktualnego piksela.
Program fragmentów realizujący pierwszy przebieg (rozmycie poziome) powyższego algorytmu może mieć następującą postać:

```
float4 GlowPixelShader( float2 f2TexCoord : TEXCOORD0,
                        uniform float2 sampleOffsets[13],
                        uniform float4 sampleWeights[13],
                        uniform sampler2D texRT : register(s0) ) : COLOR
{
      float4 finalColor = float4(0.0f, 0.0f, 0.0f, 1.0f);
      float2 finalTexCoord;

      for( int i = 0; i < 13; i++ )
      {
        finalTexCoord = f2TexCoord + sampleOffsets[i];
        finalColor += sampleWeights[i] * tex2D( texRT, finalTexCoord );
      }

      return finalColor;
}
```

Gdzie:

- `f2TexCoord` to współrzędne rozmywanego teksela.
- `sampleOffsets` to jednowymiarowa macierz (tablica) typu float2, zawierająca przesunięcia współrzędnych tekstury.
- `finalTexCoord` to końcowe (już obliczone) współrzędne tekszla z danej iteracji
- `sampleWeights` to jednowymiarowa macierz (tablica) z wartościami wag – wartości funkcji Gauss'a.
- `texRT` to textura do której została wyrenderowana scena (właśnie te teksele rozmywamy)
- `finalColor` to końcowy kolor fragmentu

Powyższy kod realizuje algorytm rozmycia Gauss'a na podstawie wcześniej obliczonych wartości wag dla pikseli za pomocą funkcji Gauss'a. Pętla przechodzi po wszystkich tekselach z przedziału [-6σ, 6σ], gdzie σ = 1. W pierwszej linijce pętli ustawiane są współrzędne tekstury przesuwane o odpowiedni współczynnik przesunięcia. Potrzeba nam tego przesunięcia aby pobrać odpowiedni teksel obrazu. Zaczynamy pobieranie od

najdalej wysuniętego teksela z lewej strony (pozycja –6). Teraz już wiadomo dlaczego wartości przesunięcia również są symetryczne względem 0. f2TexCoord są to współrzędne teksela dla fragmentu spod indeksu 0 (czyli środkowego, tego, dla którego aktualnie program pikseli liczy kolor), czyli jeśli chcemy pobrać kolor teksela najdalej wysuniętego w lewą stronę, do jego współrzędnych należy dodać odpowiednie przesunięcie. Druga linijka pętli pobiera kolor teksela spod nowo obliczonych współrzędnych (czyli, dla pierwszego przebiegu, najbardziej wysuniętego w lewą stronę) i mnoży ten kolor (każdą składową koloru) przez odpowiednią wartość funkcji Gauss'a (czyli, dla pierwszego przebiegu, wartość najmniejszą, również najbardziej oddaloną w lewą stronę od środka). Ten iloczyn dodawany jest do bazowego koloru, który na wstępnie ma wartość $\{0,0,0\}$. Zmienna finalColor jest sumą wszystkich kolorów tekseli z przedziału [-6σ, 6σ] pomnożonych przez odpowiednie wartości funkcji Gauss'a. Ta zmienna zawiera kolor rozmytego teksela obrazu.

Powyższy fragment kodu można wykorzystać zarówno do rozmycia w poziomie, jak i do rozmycia w pionie. Cała sztuczka tkwi w tablicy sampleOffsets, która zawiera przesunięcia tylko w jednym wymiarze (w poziomie lub w pionie). Dla rozmycia poziomego ma postać:

$$sampleOffsets = [\{-6,0\},\{-5,0\},\{-4,0\},\dots,\{4,0\},\{5,0\},\{6,0\}]$$

Natomiast dla pionowego:

$$sampleOffsets = [\{0,-6\},\{0,-5\},\{0,-4\},\dots,\{0,4\},\{0,5\},\{0,6\}]$$

Gdzie tablicę sampleOffsets tworzą pary $\{x, y\}$. Dla rozmycia poziomego używa się tylko współczynnika x, więc tablica sampleOffsets musi mieć wartości y = 0 dla każdej pary. Wówczas otrzymujemy proste równanie dla i-tej iteracji:

$$finalTexCoord = texCoord + sampleOffsets[i] = \{u,v\} + \{x_i,0\} = \{u + x_i, v\}$$

Gdzie:

- finalTexCoord to końcowe współrzędne tekszla
- texCoord to bazowe współrzędne tekstury dla danego fragmentu (czyli te otrzymywane wraz z informacjami o wierzchołku)
- sampleOffsets to tablica przesunięć

Dodatkowo, w programie fragmentów należy dodać jeszcze jeden współczynnik do obliczenia przesunięcia współrzędnych tekstury (czyli do pobrania odpowiedniego teksela). W powyższym programie tablica sampleOffsets już zawiera ten współczynnik, a jest nim rozmiar piksela ekranu. Wzór na ten rozmiar jest następujący:

$$pixelSize = \frac{1}{\min(screenWidth, screenHeight)}$$

Gdzie:

- screenWidth to szerokość ekranu (wyrażona w pikselach)
- screenHeight to wysokość ekranu (wyrażona w pikselach)
- min to funkcja minimum

W przypadku, gdy rozmywamy obraz n x n pikseli (tak jak w programach typu Photoshop) wówczas ten współczynnik nie jest potrzebny, ponieważ w standardowym obrazie współrzędne pikseli są na pozycjach całkowitych. W grafice komputerowej, a dokładniej w teksturach, współrzędne tekseli (czyli pikseli tekstury) są z przedziału [0, 1]. Zatem nie jest możliwe indeksowanie tekseli za pomocą liczb całkowitych. Tym bardziej, że tekstura może być dowolnie rozciągnięta na powierzchni (np. rozszerzona lub zmniejszona). W przypadku efektów post-process, gdzie mamy do czynienia z efektami pełnoekranowymi, tekstury te są zazwyczaj rozmiarów ekranu. Wobec tego potrzeba nam pewnego współczynnika – pixelSize, który będzie modyfikował przesunięcie podczas próbkowania (pobierania) konkretnego teksela.

W programie fragmentów należy dodać tą stałą do przesunięcia:

```
finalTexCoord = f2TexCoord + sampleOffsets[i] * pixelSize;
```

`pixelSize` jest to stała przekazywana do programu fragmentów. Można pozbyć się tego przekazywania oraz dodatkowego mnożenia umieszczając ją już w tablicy `sampleOffsets`. Podczas wyznaczania przesunięć należy od razu zmodyfikować je o `pixelSize`. Wówczas tablica `sampleOffsets` będzie miała następującą postać (wersja horyzontalna):

$$sampleOffsets = [\{-6 * \text{pixelSize}, 0\}, \{-5 * \text{pixelSize}, 0\}, \dots, \{6 * \text{pixelSize}, 0\}]$$

Kiedy już rozmyjemy obraz poziomo, należy go również wyrenderować do tekstury. Następnie na tej nowej teksturze należy wykonać rozmycie pionowe (tak jak na rysunku 4.5.4.). Jak widać do uzyskania pożądanego efektu należy wykonać dwa przejścia. Niestety w programie fragmentów nie da się dobrze wykonać pełnoekranowego rozmycia w jednym przebiegu, ponieważ rozmycie pionowe musi zostać wykonane na już rozmytych poziomo tekselach.

4.5.3. Efekt Bloom

Efekt bloom jest to efekt rozmywającego się światła wpadającego do ciemnego pomieszczenia, lub efekt światła przechodzącego przez liście.
W prawdziwym świecie soczewki nie mogą się idealnie skupić. Nawet idealna soczewka splecie obraz w krążek dyfrakcyjny[21]. W normalnych okolicznościach to zjawisko nie jest widoczne, ale bardzo jasne, intensywne światło powoduje, że obraz jasnego światła zdaje się wychodzić poza naturalne granice. Gdy światło jest w obrębie tego samego zakresu, efekt powodowany plamką Airy'ego nie jest widoczny, ale w miejscach gdzie bardzo jasne światło sąsiaduje z ciemnymi miejscami krążek dyfrakcyjny jest widoczny i może wyjść daleko poza zasięg jasnej części obrazu.

Za przykład może posłużyć zdjęcie robione w domu. Obiekty widziane z okna będą ok. 70-80 razy jaśniejsze niż te widziane w domu. Jeśli naświetlenie jest ustawione na obiekty znajdujące się w pokoju, okna będą wystarczająco jasne

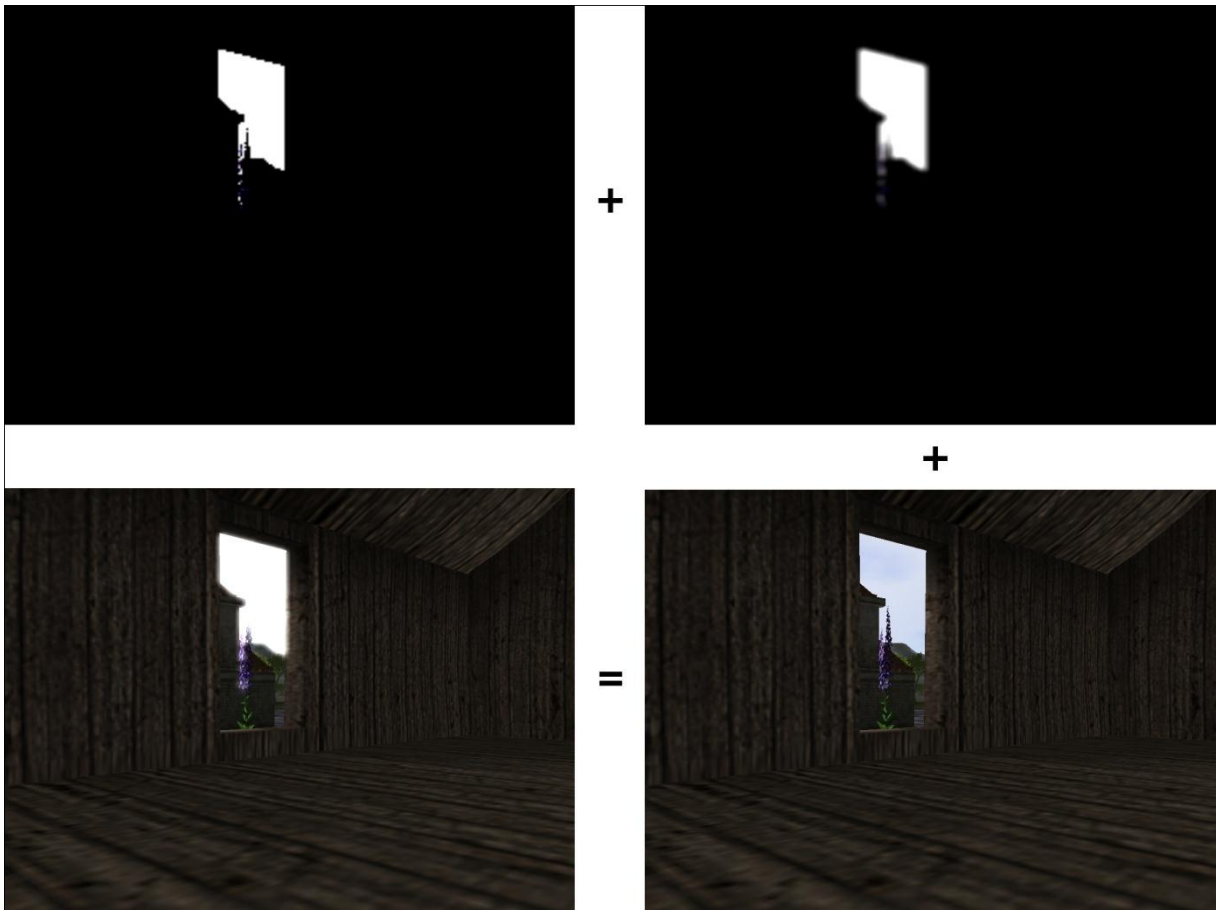
W efekcie bloom można łatwo przybliżyć efekt krążka dyfrakcyjnego, rozmywając, za pomocą funkcji Gauss'a tyko jasne części obrazu. Aby je uzyskać należy wyodrębnić tylko jasne części obrazu (bright-pass). Algorytmów może być wiele, w zależności jaki efekt chcemy uzyskać. Aby uzyskać naturalny efekt, należy użyć filtru, który podzieli kolory obrazu na czarne i białe[22], bez żadnych pośrednich. Dzięki temu rozmazane zostaną tylko te najjaśniejsze części obrazu. Jeżeli natomiast byśmy chcieli stworzyć bardziej baśniowy nastrój wtedy potrzeba będzie rozmyć cały obraz, więc należy zostawić więcej kolorów.

Prosty filtr bright-pass może mieć taką postać:

```
float4 BrightPassPixelShader( float2 texCoord: TEXCOORD0,
                              uniform sampler RT: register(s0) ) : COLOR
{
      float4 tex = tex2D(RT, texCoord);

      return tex - 0.5f;
}
```

Ten krótki program fragmentów dla każdego teksela obrazu wykona prosty filtr jasności kolorów. Każdy kolor, którego składowa jest większa od 0.5 pozostanie, natomiast reszta tekseli będzie czarna. Na takim biało-czarnym obrazie można wykonać rozmycie Gauss'a, dzięki temu otrzymamy krążki dyfrakcyjne tylko w najjaśniejszych miejscach.

[21] Plamka Airy'ego – w wyniku zagięcia światła na ekranie powstaje zakłócająca plamka. W świecie makroskopowym obrazem punktu powinien być punkt, ale w mechanice kwantowej obrazem punktu otrzymamy obraz dyfrakcyjny czyli jasną plamkę otoczoną na przemian ciemnymi i jasnymi pierścieniami.
[22] Jak wiadomo, czerń nie jest kolorem (jest brakiem koloru). Używam tego błędnego sformułowania w celu ułatwienia czytania.



Rysunek 4.5.19. Trzy kroki algorytmu efektu bloom. Lewy górny róg: filtr bright-pass. Prawy górny róg: rozmycie filtru bright-pass – efekt krążka dyfrakcyjnego. Prawy dolny róg: normalny obraz sceny. Lewy dolny róg: połączenie dwóch poprzednich obrazów w jeden.

Efekt bloom, jak widać na powyższym obrazku, robi się w trzech przejściach. Mając rozmyte jasne części obrazu oraz oryginalny obraz sceny można połączyć je w jeden, tworząc tym samym efekt bloom. Algorytmów połączenia tych obrazów też jest kilka. Można użyć interpolacji między nimi, można użyć wag dla każdego obrazu z osobna i następnie je zsumować. Na rysunku powyżej wykorzystano wagi:

```
float4 BloomBlendPixelShader(float2 texCoord: TEXCOORD0,
                             uniform sampler originalImg: register(s0),
                             uniform sampler blurImg: register(s1)) : COLOR
{
      float4 sharp = tex2D(originalImg, texCoord);
      float4 blur  = tex2D(blurImg, texCoord);

      return blur * 0.65 + sharp * 1;
}
```

`originalImg` to oryginalny obraz sceny, natomiast `blurImg` to rozmyty obraz filtru bright-pass. W programie obie tekstury są próbkowane w tych samych współrzędnych następnie kolory tekseli są sumowane z odpowiednimi wagami. Obrazowi z rozmyciem (blur) została przypisana mniejsza waga ponieważ jest zbyt jasny i zbyt rozmyty, co w rezultacie daje gorsze efekty. Oryginalny obraz pozostaje bez zmian w tym przypadku.

Wagi należy odpowiednio dobrać w zależności od filtru bright-pass. Przedstawiony powyżej filtr bright-pass niestety nie jest do końca dobry w tej postaci ponieważ zbyt ciemny efekt.

```
float4 BrightPassPixelShader( float2 texCoord: TEXCOORD0,
                              uniform sampler RT: register(s0) ) : COLOR
{
      float4 tex = tex2D(RT, texCoord);

      float fFactor = 0.5f;
      float4 f4Color = tex - fFactor;
      f4Color = dot(f4Color, float4( fFactor, fFactor, fFactor, fFactor ));

      return f4Color * 1;
}
```

W tym filtrze również odejmujemy pewną stałą od koloru, ale dodatkowo obliczamy iloczyn skalarny. Iloczyn skalarny z kolorem szarym da również kolor szary. Ponieważ wcześniej odjęliśmy 0.5 od koloru bazowego teksela to w iloczynie skalarnym mamy już do czynienia z obrazem mocno zaciemnionym. W rezultacie otrzymujemy szaro-czarny obraz. 1, która stoi przy f4Color to pewna stała, która może być użyta do rozjaśnienia.

Na koniec należy wspomnieć jeszcze o jednej istotnej rzeczy. Tekstury, do których będziemy renderować obraz po użyciu filtru bright-pass powinny być zmniejszone do ok. ¼ rozmiaru ekranu, i dopiero na tak zmniejszonym obrazie należy wykonać rozmycie. Dzięki temu zabiegowi rozmycie wykona się dużo szybciej, a po rozciągnięciu tekstury do rozmiaru ekranu otrzymamy lepszy efekt rozmycia. Trzeba uważać, by tekstura nie była zbyt mała ponieważ wtedy otrzymamy efekt migotania rozmytych, jasnych miejsc podczas poruszania kamerą. Dzieje się tak dlatego, że zbyt małe obrazy mają dużo mniej pikseli które rozmywamy. Gdy poruszamy kamerą duże, jasne, rozmyte piksele (po rozciągnięciu) zmieniają swoje położenie. Łącząc ten obraz z oryginalnym obrazem sceny otrzymamy migotanie jasnych miejsc sceny podczas poruszania kamerą.

4.5.4. Szeroki zakres dynamiczny (HDR – High Dynamic Range)

HDR jest to obraz o zakresie jasności porównywalnym z zakresem jasności widzianym przez człowieka. Do ludzkiego oka wpada promieniowanie elektromagnetyczne, które uaktywnia bodziec widzenia. Fale o długości $7{,}8 \cdot 10^{-7}$ - $4 \cdot 10^{-7}$ metra są widzialne przez oko. Światło charakteryzuje się dwoma współczynnikami: częstotliwość (zakres widzianych barw) i intensywność (energia docierająca do oka). To, jaki kolor oko widzi uzależnione jest od długości fal w wiązce światła. Częstotliwość jest odwrotnością długości fali. Intensywność określa jak bardzo przedmioty są jasne lub ciemne (wynika to

z intensywności fal emitowanych przez te przedmioty). Jasność jest odpowiednikiem intensywności. Jasność wyrażana jest w cd/m^2 (kandela na metr kwadratowy. Kandela to jednostka światłości źródła światła). Ludzkie oko reaguje na zakres jasności od $10^{-5}cd/m^2$ do $10^9 cd/m^2$.

Urządzenia nie są w stanie zarejestrować ani wygenerować tak szerokiego zakresu. Wynika to z faktu, że urządzenia cyfrowe generują lub rejestrują obraz o niskim zasięgu (LDR – Low Dynamic Range), ponieważ wykorzystują zbyt małą pulę bitów na jasność i kolor. Standardowo wykorzystywane są 24 bity na kolor w reprezentacji RGB. Daje to zakres jasności od $1\ cd/m^2$ do 80 cd/m^2.

W technice HDR do reprezentowania obrazu należy przyjąć znacznie szerszy zakres. Kolor wyraża się za pomocą liczb zmiennoprzecinkowych. Dzięki precyzji liczb rzeczywistych możliwe jest zapisanie obrazu z jasnością i nasyceniem w jakości podobnej do tej, w jakiej widzi człowiek.

Zakres dynamiki jasności obrazu to stosunek najjaśniejszego punktu do najciemniejszego. Monitor nie wyświetli idealnie czarnego punktu ponieważ zawsze ten punkt będzie świecił (co można zaobserwować wyłączając światło w pokoju). Badając zakres dynamiki jasności nie należy brać pod uwagę kolorów o wartości 0, z tego powodu, że w naturalnym świecie nie ma idealnie czarnych przedmiotów.

Obrazy HDR nie mogą być wyświetlane na normalnym monitorze, dlatego należy zastosować na nich filtr tonacji koloru, który zmieni je na obrazy LDR (czyli zmieni ich kolory do przestrzeni kolorów urządzenia), tak, aby przybliżyć obraz do jak najbardziej rzeczywistego. Jak już wcześniej zostało wspomniane do filtru tonacji koloru należy użyć zakresu dynamiki obrazu. W tym celu trzeba wyznaczyć średnią jasność (luminance) obrazu. Zazwyczaj używa się logarytmicznej średniej

$$\bar{L}_w(x,y) = \exp\left(\frac{1}{N}\sum\left(log\left(\delta + L_w(x,y)\right)\right)\right)$$

Gdzie:

- exp to funkcja wykładnicza e^x
- log to funkcja logarytmiczna
- N jest to ilość pikseli obrazu
- δ to pewna, mała, stała potrzebna, aby nie liczyć logarytmu z pikseli o wartości 0.
- $L_w(x,y)$ jest to jasność obrazu w punkcie (x,y)

Poziom jasności oblicza się w czterech krokach. W obrazie sceny w formacie zmiennoprzecinkowym (który będę nazywał po prostu obrazem sceny) należy znaleźć średnią jasność dla 3x3 tekseli (9 tekseli otaczających aktualnie przetwarzany) i rezultat zapisać w teksturze o rozmiarach 64x64 w formacie zmiennoprzecinkowym. Następnie na poprzednio wygenerowanej teksturze 64x64 należy wykonać tą samą operację i rezultat zapisać w teksturze 16x16. Ten krok należy powtórzyć dla tekstur 4x4 oraz 1x1. Dla lepszego rezultatu algorytm można rozpocząć od tekstury 128x128, czyli otrzymujemy aż pięć kroków (dla tekstury 128x128, do policzenia średniej można skorzystać tylko z 4 sąsiednich tekseli).

Rysunek 4.5.20. Oryginalny obraz sceny i pięć rezultatów obliczania średniej jasności.

Kolejnym krokiem algorytmu jest wyznaczenie klucza sceny. Klucz ten mówi jak bardzo scena jest naświetlona, niedoświetlona lub prześwietlona. Stosując dany klucz można zmapować wszystkie piksele do, wyznaczonej wcześniej, średniej jasności sceny

$$L(x,y) = \frac{a}{\bar{L}_w(x,y)} L_w(x,y)$$

Klucz oraz powyższe równanie potrzebne jest podczas tonowania kolorów, by sprowadzić obraz do formatu kolorów monitora. L(x,y) to kolor teksela w punkcie (x,y). Dla typowych obrazów klucz a ma wartości z przedziału [0.09, 0.75], w zależności od poziomu naświetlenia (exposition) jaki chcemy uzyskać. Problem z wyborem stałego klucza jest taki, że nawet bardzo jasne lub bardzo ciemne sceny będą miały ten sam poziom naświetlenia. Dlatego lepiej jest generować klucz automatycznie, korzystając ze wzoru:

$$a = \max\left(0, 1.5 - \frac{1.5}{\bar{L}_w(x,y) \cdot 0.1 + 1}\right) + 0.1$$

Podobnie jak w przypadku efektu bloom, w technice HDR także należy zastosować filtr bright-pass, czyli oddzielenie jasnych kolorów od ciemnych. Filtru bright-pass używamy na już zmniejszonej teksturze. Wynik filtru bright-pass powinien zostać zapisany do tekstury w formacie 8-bitów na kanał koloru (najczęściej RGB). Pierwszym krokiem tego filtru jest zmapowanie koloru HDR to LDR:

$$L_s(x,y) = a \cdot \frac{L_w(x,y)}{\bar{L}_w(x,y)}$$

Gdzie:

- $L_s(x,y)$ to kolor teksela w punkcie (x,y)

Następnie należy ustalić próg, który będzie odpowiadał za poziom jasności kolorów, które zostaną odseparowane:

$$L_b(x,y) = \max(L_s(x,y) - t, 0)$$

Gdzie:

- $L_b(x,y)$ to kolor teksela w punkcie (x,y)
- $\max$ to funkcja maksimum
- t to próg jasności koloru

Aby kolor nie miał wartości ujemnych należy przyciąć go do minimalnej wartości 0. Czym większa będzie wartość t tym mniejsza będzie tolerancja dla jasnych kolorów (przykładowo dla t=0.9 pozostaną same kolory białe, lub bliskie bieli, ewentualnie kolory o maksymalnym nasyceniu pewnych składowych).

Końcowym krokiem tego filtru jest przycięcie koloru do zakresu [0,1], ponieważ jasne światła nadal mogą mieć zbyt wysokie wartości.

$$B(x,y) = \frac{L_b(x,y)}{o + L_b(x,y)}$$

Gdzie:

- $B(x,y)$ to kolor teksela w punkcie (x,y)
- o to przesunięcie, które kontroluje oddzielenie jasnych obiektów i bardzo intensywnych świateł.

Gdy filtr bright-pass zostanie ukończony należy jego rezultat poddać rozmyciu, tak jak to miało miejsce w efekcie bloom.

Ostatni krok algorytmu polega na sprowadzeniu sceny z formatu HDR do LDR. Należy do tego użyć filtru tonowania kolorów. W ostatnim przebiegu potrzeba współczynnika średniej jasności (który otrzymujemy z ostatniej tekstury 1x1), tekstury z efektem bloom oraz tekstury HDR ze sceną (czyli tekstury w formacie zmiennoprzecinkowym, do której została wyrenderowana scena). Na obrazie HDR należy zastosować filtr tonowania kolorów. Jakiego filtru należy użyć zależne jest od efektu jaki chcemy uzyskać.

Mapowanie liniowe to jeden z najprostszych filtrów tonowania kolorów:

$$color(x,y) = a \cdot \frac{L_w(x,y)}{\bar{L}_w(x,y)}$$

Ten filtr niestety nie jest zbyt dobry. Ten filtr daje duży kontrast ponieważ używa kolorów bardzo bliskich średniej jasności sceny, skalowalnej przez klucz a.

Tonowanie Reinhard'a używa nieliniowego mapowania do uzyskania dynamicznego zasięgu obrazu. Najpierw należy przeskalować obraz HDR kluczem i logarytmiczną średnią jasności sceny do obrazu LDR używając równania z mapowania liniowego.

$$color(x,y) = \frac{L_s(x,y)}{1 + L_s(x,y)}$$

Ta funkcja skaluje duże wartości jasności przez 1/L, a niskie przez wartości zbliżone do 1, więc wszystkie wartości znajdą się w przedziale [0,1]. Wadą tego rozwiązania jest to, że jasne kolory nigdy nie będą idealnie białe, będą zawsze w odcieniach szarości.

Zmodyfikowane tonowanie Reinhard'a to ulepszona wersja poprzedniego tonowania. Pierwszy krok jest identyczny jak poprzednio – należy przeskalować obraz HDR do LDR. Następnie należy użyć rozszerzonej wersji równania Reinharda:

$$color(x,y) = \frac{L_s(x,y) \cdot \left(1 + \frac{L_s(x,y)}{L_{white}^2}\right)}{1 + L_s(x,y)}$$

To równanie pozwala określić które wartości jasności powinny osiągnąć wartość 1. L_{white} jest to stała, która daje dobre rezultaty, gdy jest ustawiona na 2.5, gdyż daje wystarczająco duży kontrast i kolory mogą pozostać odpowiednio intensywne.

Dostosowujące się mapowanie logarytmiczne jest kolejnym tonowaniem. Charakteryzuje się tym, że zawsze daje wartości z przedziału [0,1].

$$color(x,y) = \frac{a}{\log_{10}(\bar{L}_w(x,y) + 1)} \cdot \frac{\log(L_w(x,y) + 1)}{\log\left(2 + \left(\left(\frac{L_w(x,y)}{\bar{L}_w(x,y)}\right)^{\frac{\log b}{\log 0{,}5}}\right) \cdot 8\right)}$$

Ta funkcja interpoluje wartości między dużym kontrastem a dużą kompresją. Wartość b kontroluje bieg interpolacji i powinna być z przedziału (0.5, 1).

Który z filtrów tonowania kolorów należy wybrać zależne jest od potrzeb oraz wydajności. Mapowanie logarytmiczne jest w stanie dać dobre efekty, ale koszt tego filtru może okazać się zbyt duży. Mapowanie liniowe jest najtańszą obliczeniowo metodą, ale może dawać zbyt duży kontrast. Tonowanie Reinhard'a jest nieco bardziej wydajne od mapowania logarytmicznego a daje bardzo podobny rezultat. Każdy z tych filtrów używa klucza i do każdego z nich klucz może być inaczej dobrany.

W skrócie algorytm oświetlenia HDR ma następującą postać:

- Wyrenderuj scenę do tekstury w formacie zmiennoprzecinkowym o rozmiarze ekranu.
- Zmniejsz tą teksturę i wyszukaj średniej wartości jasności obrazu.
- Powtarzaj poprzedni krok dla tekstur o wymiarach 16x16, 4x4 i 1x1
- Odseparuj jasne części sceny od ciemnych
- Rozmyj tylko jasne części sceny
- Zastosuj tonowanie kolorów dla obrazu sceny i połącz z efektem bloom (dwa poprzednie kroki).

Tak może wyglądać program wierzchołków liczący prostą średnią jasności:

```
float4 AverageLuminence128x128(float2 uv : TEXCOORD0,
                               uniform float2 texelSize,
                               uniform sampler2D inRTT : register(s0)
                              ) : COLOR
{

      float4 accum = float4(0.0f, 0.0f, 0.0f, 0.0f);

      float2 texOffset[4] = {
            -0.5, -0.5,
            -0.5,  0.5,
             0.5, -0.5,
             0.5, 0.5 };

      for( int i = 0; i < 4; i++ )
      {
            accum += tex2D(inRTT, uv + texelSize * texOffset[i]);
      }

      float lum = dot(accum, LUMINENCE_FACTOR);
      lum *= 0.25;
      return lum;
}
```

Tego programu można użyć to policzenia średniej dla pierwszego kroku. W tym przypadku jest to tekstura 128x128, więc bierzemy tylko 4 sample.

4.6. Renderowanie środowiska

Wiele gier komputerowych rozgrywa się w otwartych scenach (outdoor). Takie sceny wymagają renderowania terenu, drzew, krzewów, trawy, rzek i jezior. Realizm wygenerowanej sceny może przesądzić o sukcesie gry. Wiele gier skupia się na maksymalnym wykorzystaniu sprzętu do renderwoania wyśmienitej grafiki (bardzo często kosztem fabuły, najczęściej są to gry akcji), a wielu graczy ceni sobie dobrą grafikę w grach. Czym bardziej szczegółowy krajobraz uda się stworzyć, tym gra stanie się atrakcyjniejsza.

4.6.1. Teren

Najważniejszą częścią scen typu outdoor jest teren. Bez terenu nie było by świata, nie było by po czym się poruszać. Dodając fizykę zawiśli byśmy w próżni, lub z grawitacją – ciągle byśmy spadali. Teren stanowi naturalne podłoże, po którym gracz będzie się poruszał. Ponieważ teren reprezentuje cały dostępny świat (drogi, góry, równiny, itp.) w praktyce jest dość duży. Oczywiście, jak wszystko w grafice komputerowej, teren również jest zbiorem wierzchołków – geometrią. Czym ciaśniej są wierzchołki ułożone tym lepszy jest uzyskany efekt (wzniesienia są bardziej płynne, brak ostrych krawędzi. Teren jest mniej „kwadratowy").

Teren można generować na wiele sposobów. Najpopularniejsze z nich to proceduralne generowanie, generowanie z mapy wysokości, lub po prostu stworzenie terenu w jakimś edytorze (gry, lub w programie do grafiki 3D).

Proceduralne generowanie terenu

Istnieje kilka sposobów aby wygenerować teren algorytmem. Jeden z nich polega na przemieszczaniu środkowego punktu płaszczyzny.

Algorytm przemieszczania środkowego punktu

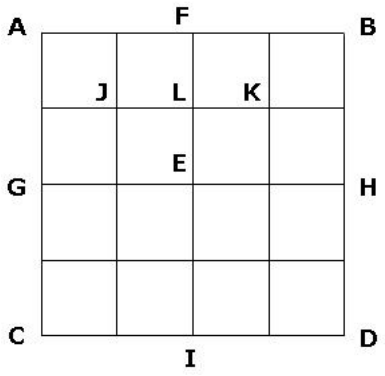
W jednym wymiarze algorytm ma następującą postać. Weźmy dowolny odcinek AB, następnie zaznaczmy na nim punkt C będący jego środkiem. Punkt ten przesuwamy losowo w przedziale $[-\frac{w}{2}, \frac{w}{2}]$, gdzie w to pewna stała wartość (dobrym przykładem w jest długość odcinka). Następnie w każdym z odcinków AC oraz CB wykonujemy te same operacje, z tym, że w każdej kolejnej iteracji wartość w jest mnożona przez 2^{-r}, gdzie r to pewna stała informująca o chropowatości. Gdy r ma wartość 1 wówczas w jest dzielone przez 2 w każdej iteracji i fragmenty terenu (ponieważ każdy odcinek też jest dzielony na pół) będą idealnie podobne do siebie (mniejsze i większe).

W dwóch wymiarach algorytm jest podobny. Różnica polega na policzeniu pięciu środków dla każdego kwadratu. Załóżmy, że mamy kwadrat ABCD. Jego środek liczmy jako uśrednioną wartość sumy jego boków:

$$E = \frac{A + B + C + D}{4} + random(-\frac{w}{2}, \frac{w}{2})$$

Gdzie E to środek kwadratu.

W tym momencie mamy już dodatkowe cztery kwadraty (dochodzą nowe wierzchołki: F,G,H,I): AFEG, FBHE, GEIC, EHDI. Teraz należy zmniejszyć wartość w, $w = w \cdot 2^{-r}$ i powtórzyć operacje dla nowych kwadratów.

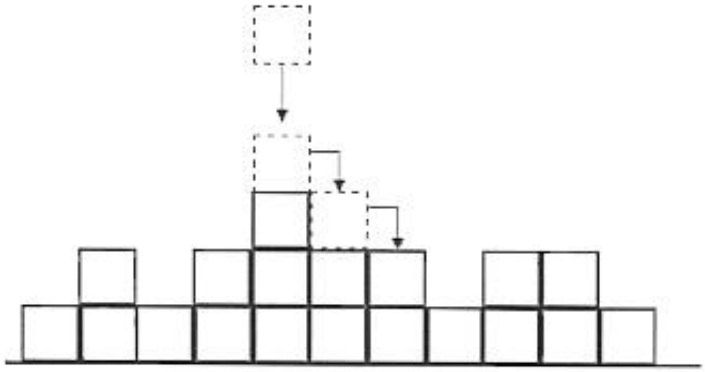


Rysunek 4.6.21. Dwuwymiarowa kratka geometrii. Drugi etap algorytmu przemieszczenia.

Algorytm osadzania cząsteczek

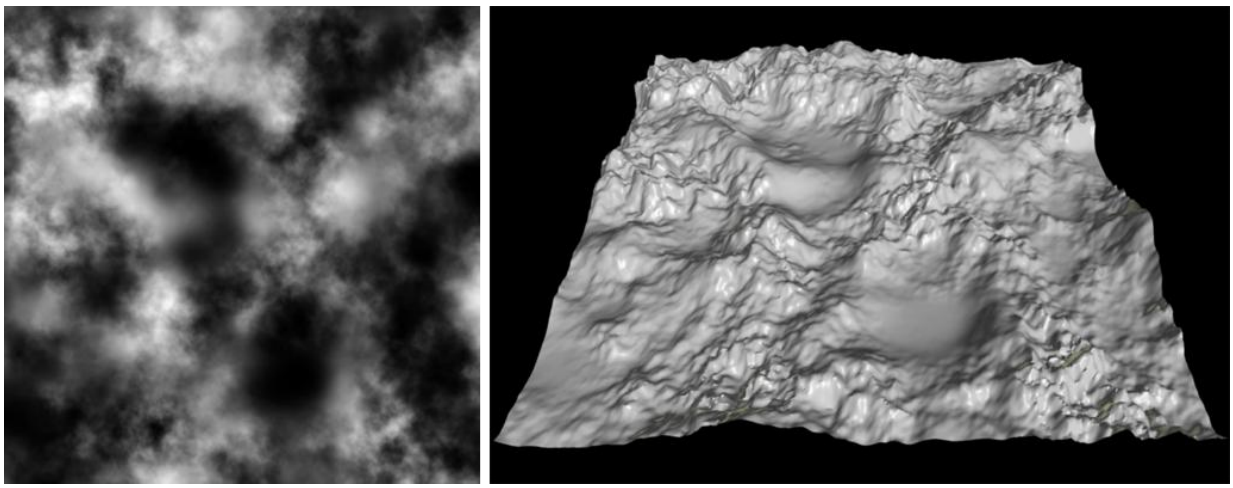
Algorytm ten wziął swój początek z molekularnej wiązki epitaksji (Molecular Beam Epitaxy) i jest to proces osadzania się atomów na podłożu. Aby utworzyć proceduralnie teren, należy spuszczać cząsteczki na płaską powierzchnię. Cząsteczka musi się znaleźć w stanie stabilnym, czyli takim, w którym sąsiednie cząsteczki nie są na niższym poziomie. Załóżmy, że powierzchnia ma poziom 0. Opuszczając na nią pierwszą (c1) cząsteczkę znajdzie się na poziomie 0. Jeśli teraz opuścimy na nią drugą (c2) cząsteczkę znajdzie się również na poziomie 0, ponieważ c1 nie ma sąsiadów oraz c1 znajduje się na najniższym poziomie. W takim razie, c2 zostanie ulokowana obok c1. Jeżeli teraz opuścimy trzecią cząsteczkę (c3), na c1, również zostanie ulokowana na poziomie 0, obok c1, ale po drugiej stronie. Opuszczając czwartą cząsteczkę (c4) na c1, znajdzie się ona już na poziomie 1, ponieważ wszystkie sąsiednie cząsteczki c1 znajdują się na poziomie 0, więc c4 nie może niżej zostać umieszczona niż na poziomie 1 (rozpatrując przypadek jednowymiarowy, w przypadku dwuwymiarowym należy rozpatrzyć więcej sąsiadów).

Aby teren był bardziej realistyczny należy zmieniać miejsce zrzutu cząsteczek. Jeśli zrzucanie będzie tylko z jednego miejsca otrzymamy jedno duże wzgórze.



Rysunek 4.6.22. Algorytm osadzania cząstek. Rzut z boku. Obrazek pobrany z [21].

Powyższe dwa algorytmy dobrze się nadają do tworzenia map wysokości (heightmap) terenu. Mapą wysokości dla terenu nazwiemy obraz w skali szarości, gdzie jaśniejsze kolory oznaczają wyższe punkty w terenie (wartość koloru {0,0,0} oznacza najniższy punkt, {255,255,255} – najwyższy).



Rysunek 4.6.23. Z lewej mapa wysokości. Z prawej teren wygenerowany za pomocą mapy wysokości. Obrazek pobrany z [87].

Jak już wcześniej zostało wspomniane teren może być bardzo duży, więc do jego utworzenia potrzeba będzie bardzo dużej ilości wierzchołków. Obecne karty graficzne potrafią wyrenderować nawet do kilkuset milionów trójkątów. Tworząc scenę, nie należy kierować się jedynie terenem, przecież na nim znajduje się wiele innych obiektów, które również muszą zostać narysowane. Jeśli chcemy mieć teren dobrej jakości musimy na niego poświęcić dużo wierzchołków. Warto zwrócić uwagę na to, że w przypadku dużych odległości rzadko kiedy widoczny będzie cały teren, a już na pewno nigdy nie będzie widoczny z bliska każdy jego element. Kamera znajduje się w pewnym punkcie i widzi tylko pewny obszar, więc tylko widoczny obszar można wyświetlać z dużą dokładnością, a obszar, który jest dalej od kamery może mieć już mniejszą dokładność. Dzięki temu, dalsze miejsca będą przypominały wyglądem oryginalny teren, ale z tej odległości i tak będą dobrze wyglądać. W prawdziwym świecie jest podobnie. Z daleka nie widać wszystkich szczegółów tylko zarysy, a w miarę zbliżania się, wyłania się dokładniejszy obraz. Podobnie można zrobić w przypadku wirtualnego terenu. Ta technika, nazywa się

poziomem szczegółowości (Level Of Detail – LOD) i jest stosowana do każdego typu modeli 3D, nie tylko terenu.

Istnieją różne algorytmy wyznaczania poziomu szczegółowości. Najprostszy może polegać na podzieleniu terenu na kratki. Na przykład, podzielmy teren na dziewięć kratek, oraz niech teren ma wielkość 1500x1500. Każda kratka będzie miała rozmiar 500x500, oraz, dla dobrego poziomu szczegółowości, niech każda kratka ma również 500x500 wierzchołków (wierzchołki są w odległości o 1 od siebie). Przy takim poziomie otrzymujemy 2250000 wierzchołków dla samego terenu. Jeśli chcielibyśmy to wszystko wyświetlić za jednym razem ilość klatek mogła by być zbyt mała (w zależności od karty graficznej). W takim razie, stwórzmy odpowiednie wersje kratek dla każdej z dziewięciu wcześniej utworzonych. Będą to ich odpowiedniki, ale o mniejszej ilości wierzchołków. Ten poziom kratek będzie zawierał o połowę mniej wierzchołków czyli 250x250. Stwórzmy dodatkowo kolejny poziom, który będzie miał o połowę mniej wierzchołków niż poprzedni, czyli 125x125. W przypadku jeżeli kamera znajdzie się w środkowej kratce, zostanie wyświetlona kartka z maksymalną szczegółowością (czyli 500x500), a wszystkie pozostałe z poziomem drugim (czyli 250x250). Jeżeli kamera znajdzie się w kratce brzegowej, wszystkie kratki sąsiadujące (w tym przypadku dwie brzegowe i środkowa) zostaną wyświetlone z drugim poziomem szczegółowości (250x250), a dwie kratki po przeciwnej stronie z poziomem trzecim (125x125). W tym przypadku mamy już:

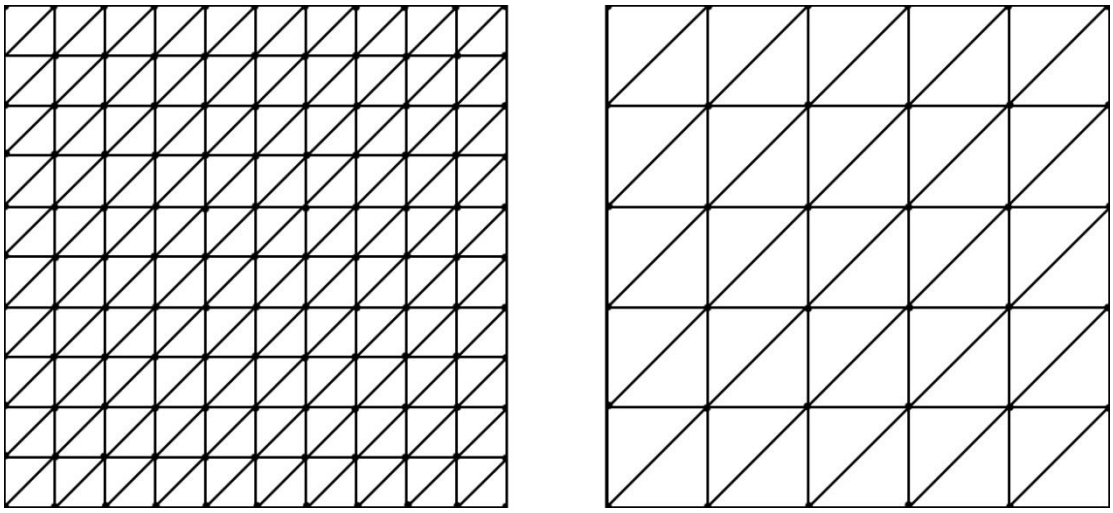
$$500 \cdot 500 + 3 \cdot (250 * 250) + 2 \cdot (125 \cdot 125) = 250000 + 187500 + 31250 = 468750$$

Otrzymujemy prawie pięć razy mniej wierzchołków wyświetlając cały teren, a poziom szczegółowości nie będzie zauważalny dla odległych kratek. Oczywiście można dobrać poziomy w inny sposób. Dzielenie przez 2 każdego kolejnego poziomu może nie być najlepszym pomysłem. W podanym przykładzie czwarty poziom było by ciężko uzyskać w ten sposób. Dlatego dla tej metody najlepiej brać rozmiar terenu o wielokrotności 2. Ten algorytm, choć bardzo prosty, ma wiele wad. Po pierwsze stwarza duży narzut pamięciowy. Należy trzymać wszystkie kratki (w tym przykładzie 27) w pamięci. Po drugie na łączeniach siatek z różnym poziomem szczegółowości mogą powstać dziury (niedopasowanie wierzchołków, na dwa trójkąty wyższego poziomu przypada jeden trójkąt niższego). Pierwszy problem można w pewien sposób rozwiązać trzymając bufory indeksów na wierzchołki dla każdego poziomu szczegółowości. W przypadku zmiany poziomu szczegółowości kratki należy zaktualizować bufor indeksów i wysłać ponownie do karty graficznej (indeksy te mówią karcie, które wierzchołki ma brać aby stworzyć trójkąty).

Zazębianie kratek

Ten algorytm jest podobny do poprzedniego, ale eliminuje jego wady. Mamy teren podzielony na kratki. Każda kratka posiada taką samą ilość wierzchołków. Do tego mamy n (wartość n będzie podana później) buforów indeksów dla wierzchołków. Ponieważ każda kratka to osobny model, więc ma k wierzchołków, dlatego do każdej kratki można zastosować każdy z buforów indeksów. Jest to ważna informacja, ponieważ redukuje się ilość potrzebnych buforów do minimum (każda kratka ma tą samą ilość wierzchołków – k,

wierzchołki są indeksowane w przedziale [0, k - 1], więc każdy bufor indeksów może być przypisany każdej kratce bez obawy, że wyjdziemy poza ostatni wierzchołek). W tym algorytmie stosujemy optymalizację z poprzedniego. Nie przechowujemy kratek o różnej ilości wierzchołków, tylko bufory indeksów o różnych rozmiarach.

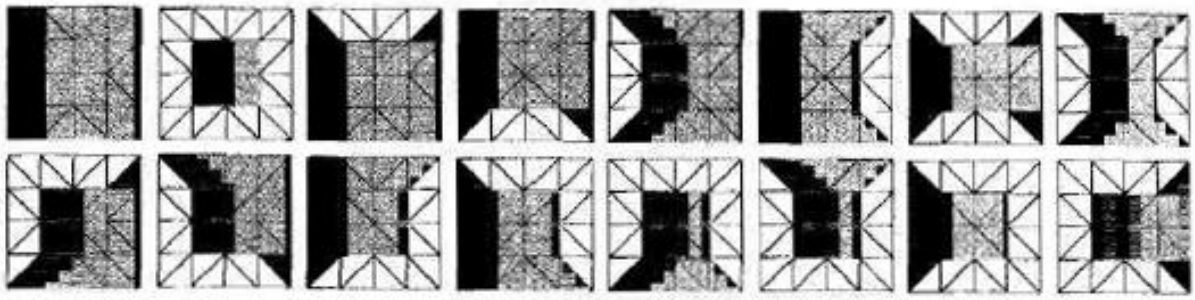


Rysunek 4.6.24. Ta sama siatka z różnymi poziomami szczegółowości.

Rysunek powyżej demonstruje wykorzystanie buforu indeksów do stworzenia różnych poziomów szczegółowości. Siatka z prawej strony wykorzystuje o połowę mniej wierzchołków (czyli bufor indeksów zawiera o dwa razy mniej indeksów na wierzchołki. Indeksowane są parzyste wierzchołki).

Mając odpowiednie bufory indeksów na wierzchołki dla każdego poziomu szczegółowości można odpowiednio wyświetlać kratki. Aby uniknąć szczelin między kratkami o różnych poziomach szczegółowości należy użyć odpowiednich łączeń między nimi. Łączenie to w rzeczywistości odpowiedni bufor indeksów na wierzchołki. Istnieje szesnaście podstawowych wersji kratek w zależności od ilości łączeń:

- bez łączeń, z łączeniami ze wszystkich stron
- z łączeniami z góry
- z łączeniami z dołu
- z łączeniami z lewej strony
- z łączeniami z prawej strony
- z łączeniami z góry i z dołu
- z łączeniami z lewej i z prawej strony
- z łączeniami z góry i z lewej strony
- z łączeniami z dołu i z lewej strony
- z łączeniami z dołu i z prawej strony
- z łączeniami z góry i z prawej strony
- z łączeniami z góry, lewej i prawej strony
- z łączeniami z dołu, lewej i prawej strony
- z łączeniami z góry, dołu i prawej strony
- z łączeniami z góry, dołu i lewej strony



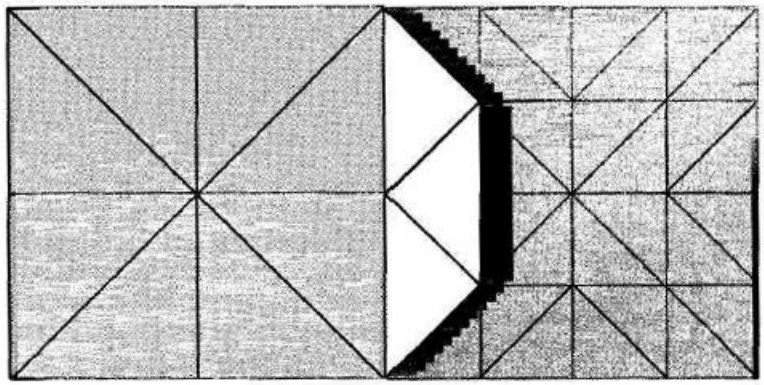
Rysunek 4.6.25. Szesnaście wersji kratek. Obrazek pobrany z [25].

Każda z tych kratek ma odpowiednio wycięte wierzchołki (czyli odpowiednio ustawione bufory indeksów), tak aby, w zależności od typu kratki, w to wycięcie zmieściło się odpowiednie łączenie. Łączenie zabiera jedną kolumnę i/lub jeden wiersz kratki (krawędzie). Ilość łączeń jest odpowiednia dla każdej kratki (czyli też 16). Każda kratka ma łączenia tylko w dół, czyli do niższego poziomu szczegółowości. W skrócie rzecz ujmując teren składa się z dwóch typów siatek: kratek z odpowiednim poziomem szczegółowości z odpowiednio wyciętymi miejscami na łączenia, oraz z łączeń, które są „pomostem" między dwoma sąsiednimi poziomami szczegółowości. Alternatywnie można od razu utworzyć szesnaście typów kratek (dla każdego poziomu szczegółowości) od razu z odpowiednimi łączeniami. Algorytm sprawdzał by poziom szczegółowości dla danej kratki oraz dla jej sąsiadów. Następnie dla danej kratki wybierał by bufor indeksów z odpowiednim poziomem szczegółowości oraz łączeniami. Na przykład, jeśli mamy teren 9x9 kratek i znajdujemy się w lewej dolnej kratce – K_00 (indeksując od lewego, dolnego rogu), to algorytm powinien wybrać kratkę K_00 o najwyższym poziomie szczegółowości oraz z łączeniami góra-prawo. Kratka górna – K_01, nad K_00, kratka z prawej strony – K_10 kratki K_00, oraz kratka K_11, która styka się z prawym górnym rogiem K_00 będą miały mniejszy poziom szczegółowości. Dla kratki K_01, powinna zostać wybrana kratka z łączeniem górnym (ponieważ tylko kratka sąsiadująca z górną krawędzią K_01 ma niższy poziom szczegółowości), dla K_10 kratka z łączeniem z prawej strony, a dla K_11 kratka z łączeniem z góry i z prawej strony. Układ tych kratek przypominać macierz. Niech wartość 1 to najwyższy poziom szczegółowości.

$$\begin{bmatrix} 3 & 3 & 4 \\ 2 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

Poziomy szczegółowości kratek można wybierać na różne sposoby. Przedstawiona powyżej metoda oblicza odległość kratki od kamery i na tej podstawie dobiera poziom szczegółowości dla każdej kratki. Oczywiście poziom ten dobierany jest również w oparciu o kratki sąsiednie. Algorytm powinien sprawdzać i ustawiać LOD dla każdej kratki poczynając od tej, w której znajduje się kamera. Dzięki temu algorytm sprawdzi każdego sąsiada każdej kratki i ustawi mu odpowiedni poziom szczegółowości. Sprawdzanie odległości od kamery może być robione za pomocą odległości kratek od kamery. Nie jest to do końca dobra metoda, ponieważ kamera może znajdować się na przecięciu kilku kratek. Algorytm będzie działał najlepiej jeśli rozpocznie sprawdzanie od kratki, w której znajduje się kamera, następnie sprawdzi odległość z kamery do sąsiednich kratek. Jeśli odległość będzie wystarczająco mała wówczas kratki te zostaną narysowane z takim samym poziomem szczegółowości. Jeśli algorytm obliczy już LOD dla każdej z

sąsiadujących kratek do kratki, w której znajduje się kamera wiadomo, że każda pozostałe kratki będzą mieć już mniejszy poziom szczegółowości. Wszystkie kratki sąsiadujące z sąsiadami kratki A (niech A oznacza kratkę, w której znajduje się kamera) mają o jeden poziom szczegółowości mniej (każdy z sąsiadów kratki A może mieć poziom szczegółowości równy poziomowi kratki A, lub o jeden mniejszy).



Rysunek 4.6.26. Kratki z różnymi poziomami szczegółowości oraz łączenie między nimi. [26].

4.6.2. Trawa

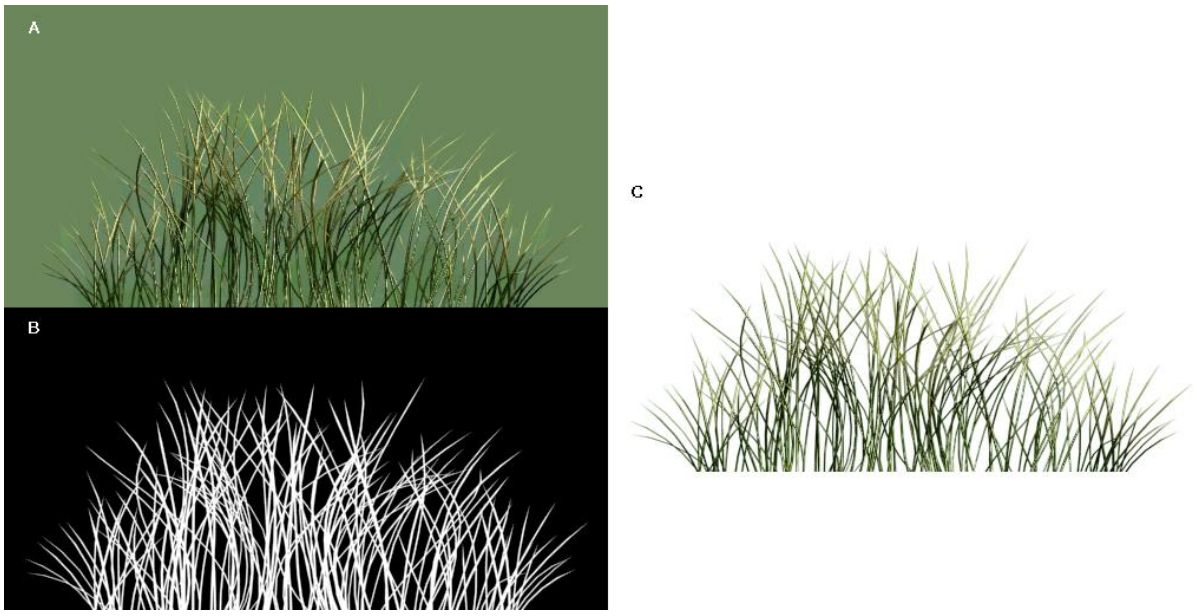
Aby wygenerowany teren nie był zupełnie pusty warto umieścić na nim trochę drzew i trawy. Obecne karty graficzne pozwalają na wygenerowanie olbrzymiej łąki, tak jak na obrazku poniżej.

Rysunek 4.6.27. Obrazek pochodzi z gry The Elder Scrolls: Oblivion

Najprostszym sposobem na wygenerowanie sceny z obrazka powyżej było by umieszczenie każdego źdźbła trawy osobno. Oczywiście najprostsze rozwiązanie będzie w tym przypadku najgorszym. Dla tak dużej ilości trawy ilość potrzebnych wierzchołków była by zbyt duża, aby można było je wyrenderować naraz. Ilość potrzebnych wierzchołków jest najważniejszym problemem podczas tworzenia scen typu otutdoor. Sceny te wymagają wielkiej ilości geometrii, by uzyskany został jak największy realizm. Zatem, podobnie jak w przypadku terenu, należy zminimalizować ilość potrzebnych wierzchołków. Nasze założenia są następujące:

- należy wygenerować jak najwięcej źdźbeł trawy używając jak najmniejszej ilości trójkątów
- źdźbła trawy powinny być widoczne tak samo niezależnie od kąta patrzenia.

Aby zrealizować punkt pierwszy, najpierw należy przygotować odpowiednią teksturę dla trawy. Kilka źdźbeł trawy należy umieścić na jednej teksturze. Tło tej tekstury powinno być przezroczyste, tak aby tylko źdźbła były widoczne. Dlatego należy użyć tekstur w formacie 32-bitowym z kanałem alfa.

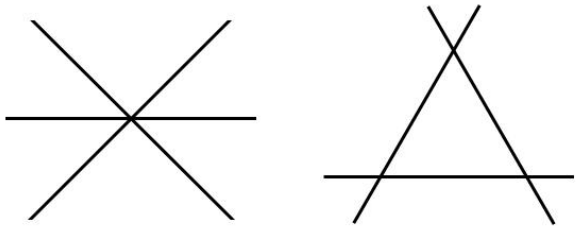


Rysunek 4.6.28. A) Tekstura trawy. Kanały RGB. B) Kanał alfa tej samej tekstury. C) Zastosowanie przezroczystości w renderowaniu.

Jak już łatwo się domyśleć trawa będzie składała się z prostokątów pokrytych odpowiednią teksturą. Dzięki temu za pomocą dwóch trójkątów (czyli jednego prostokąta) można przedstawić wiele źdźbeł trawy.

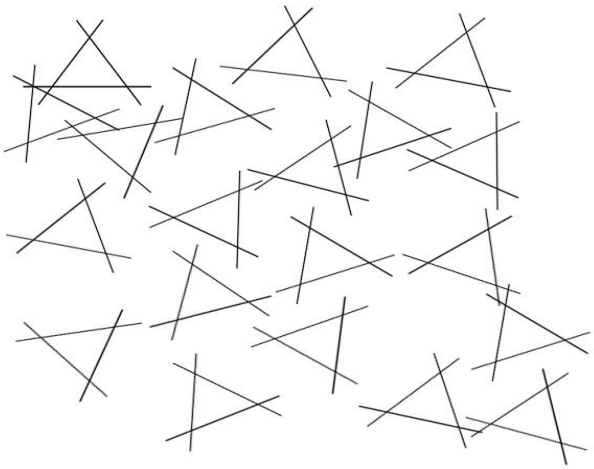
Teraz należy się zastanowić jak te prostokąty rysować – przedstawiać na scenie. Jeżeli byśmy je ułożyli w rzędy, jeden obok drugiego, mogli byśmy uzyskać efekt realizmu tylko z pewnych kątów widzenia. Niestety bardzo szybko wyszły by na jaw rzędy prostokątów, oraz trawa wyglądała by na płaską. Takiego wyświetlania trawy można użyć jedynie do rysowania w bardzo odległych i niedostępnych dla kamery miejscach. Innym

rozwiązaniem było by umieszczanie pojedynczych prostokątów w losowych miejscach. Pozbycie się rzędów na rzecz pojedynczych, losowo umiejscowionych prostokątów zapewni nieco więcej realizmu, ale nadal pozostanie efekt płaskiej trawy – łatwej do zauważenia, płaskiej geometrii trawy. Rozwiązaniem może okazać się zastosowanie billboardów[23]. Dzięki temu nigdy nie zauważymy, że trawa zrobiona jest z płaskiej powierzchni[24]. Problemem z wykorzystaniem billboard'ów jest ich natura. Łatwo zauważyć, szczególnie przy tak dużej ich ilości, że odwracają się zawsze w stronę kamery. Jeśli zrezygnujemy z billboard'ów pozostaje nam jeszcze jeden wybór – obiekty trawy. Obiektem takim będzie zbiór oteksturowanych prostokątów, który będzie reprezentował kępkę trawy. Minimalna ilość prostokątów składających się na obiekt to trzy, i jest to optymalna i najczęściej spotykana ilość. Dobre efekty dają dwa kształty obiektów: gwiazdka oraz trójkąt (patrząc na nie z góry).



Rysunek 4.6.29. Dwa typy obiektów trawy. Z lewej: gwiazdka, z prawej trójkąt.

Jeżeli weźmiemy wystarczająco dużą ilość takich obiektów, ułożymy je odpowiednio blisko siebie, obrócimy każdy o losowy kąt względem swojego środka (dla większego realizmu – mniejsza powtarzalność), użyjemy testu alfa aby uzyskać przezroczystość tła tekstury otrzymamy realistycznie wyglądającą trawę.



Rysunek 4.6.30. Losowo ułożone i obrócone obiekty trawy. Widok z góry.

Na tym etapie trawa już wygląda dobrze, ale do pełnego realizmu brakuje jej animacji. Lekki powiew wiatru może sprawić, że trawa zacznie falować. W grafice komputerowej animować trawę można na kilka sposobów. Generalnie chodzi o użycie funkcji

[23] Billboard to, w grafice komputerowej, płaski obiekt (przeważnie prostokąt), który zawsze jest zwrócony przodem w stronę kamery.
[24] Wyjątkiem jest widok na trawę z góry. Wtedy zawsze będzie można zobaczyć płaski obiekt. Uwaga ta tyczy się całego podrozdziału dotyczącego tworzenia trawy.

trygonometrycznych: sinus i cosinus. Należy wziąć pod uwagę pozycję wierzchołków, środka obiektu lub całego zbioru obiektów, czas, siłę i kierunek wiatru. Oczywiście należy animować tylko górną część trawy ponieważ dolna jest na stałe przyczepiona do podłoża. Łatwo jest odróżnić wierzchołki górne od dolnych. Wystarczy użyć współrzędnych tekstury. Wierzchołki górne powinny mieć taką samą wartość współrzędnej t (czyli tej w osi pionowej) równą 0 lub bliskiej 0. Animację powinno się wykonać w programie wierzchołków, aby zaoszczędzić cenne cykle CPU.

Animacja zbioru obiektów

Trawę należy podzielić na zbiory obiektów (kępek trawy) i każdy z nich animować w ten sam sposób. Każdy górny wierzchołek każdego obiektu jest przesuwany w tym samym kierunku o tą samą wartość. Używając odpowiednich algorytmów można bardzo dobrze symulować wiatr. Do programu wierzchołków wystarczy przesłać wcześniej obliczony wektor kierunku oraz wektor przesunięcia. Minusem tej metody jest duża ilość wywołań rysowania. Ze względów optymalizacyjnych najlepiej jest wszystkie obiekty umieścić w jednym buforze wierzchołków (tzw. batching) i za pomocą jednego wywołania rysowania wyrenderować wszystko na raz. W tej metodzie dzielimy całą trawę na mniejsze fragmenty i animujemy osobno wszystkie obiekty z danego fragmentu. Żeby animacja wyglądała dobrze należy utworzyć wiele zbiorów obiektów, co spowoduje wiele wywołań rysowania.

Animacja każdego wierzchołka z osobna

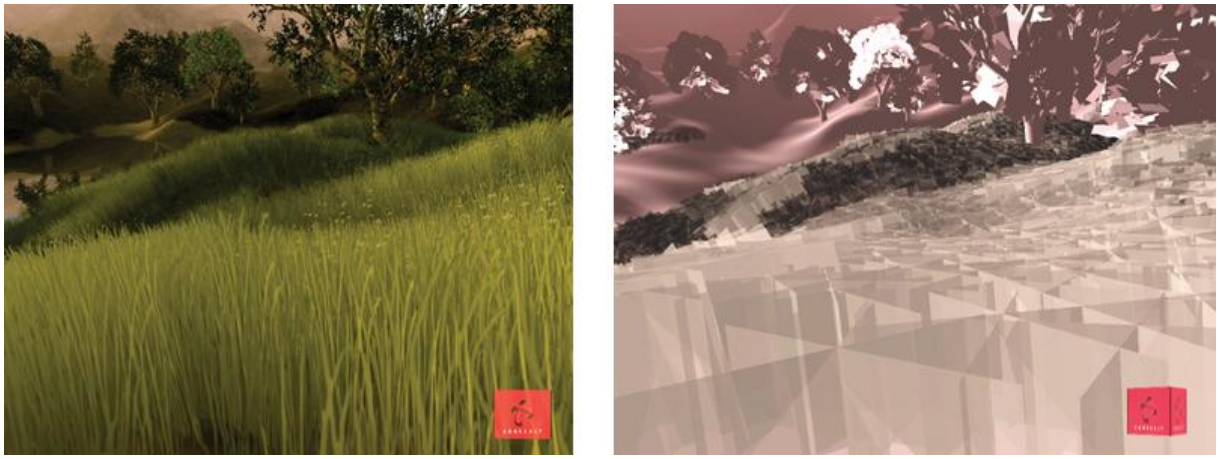
W tej technice animacji program wierzchołków zajmie się animacją każdego wierzchołka z osobna. Dzięki temu nie będzie trzeba tworzyć wielu małych grup obiektów i animować ich z osobna, tak jak to było w przypadku poprzedniego algorytmu. Aby animacja była realistyczna wektory przesunięć dla każdego wierzchołka nie mogą być tej samej długości (dwa sąsiednie wierzchołki mogą się przesunąć o inny wektor). Algorytm ten może spowodować pewne nienaturalne efekty. Wierzchołki wszystkich pobliskich obiektów mogą być przesuwane bardzo podobnie, co spowoduje brak pewnego chaosu i nieprzewidywalności. Zapobiec temu można używając jakiejś pseudo-losowej funkcji.

Animacja każdego obiektu trawy

Ten algorytm jest kombinacją poprzednich. Nie animujemy każdego wierzchołka z osobna, ale cały obiekt, bazując na jego pozycji. Dzięki temu można zasymulować chaos, ponieważ każdy obiekt trawy może być odchylany w innym kierunku (nawet sąsiednie). Każdy wierzchołek musi znać pozycję obiektu, do którego należy (zatem informacja ta powinna zostać przekazana do programu wierzchołków jako jedna ze składowych formatu wierzchołka). W tym algorytmie wszystkie górne wierzchołki danego obiektu są przesuwane w tym samym kierunku i o tą samą wartość. Otrzymujemy minimalną ilość wywołań rysowania oraz bardziej naturalną animację całej trawy.

Animacja w programie wierzchołków ma pewną wadę. Ze względu na długość programu wierzchołków, algorytm animacji powinien być krótki (czym krótszy program tym

szybszy), więc symulacja będzie mniej złożona, czyli mniej naturalna. Natomiast zyskujemy duże przyspieszenie w porównaniu z użyciem CPU.



Rysunek 4.6.31. Z lewej: scena z wyrenderowaną trawą. Z prawej: ta sama scena z wyłączonym teksturowaniem. Obrazki pobrane z [44].

4.6.3. Woda

Woda jest jednym z najtrudniejszych, do wygenerowania, elementów środowiska. Bertrand Guillot (A reappraisal of what we have learnt during three decades of computer simulations on water) przeanalizował 46 modeli matematycznych opisujących wodę i okazało się, że żaden z nich nie odzwierciedla w pełni rzeczywistości. Większość z tych modeli jest zbyt droga obliczeniowo, aby zastosować je w grach komputerowych, które na rendering wody mogą poświęcić tylko pewien procent czasu obliczeniowego. Jednakże woda jest jednym z tych elementów grafiki, która sprawia, że scena może wyglądać naprawdę realistycznie.

Na rendering wody składają się dwa procesy:

- Efekty optyczne
- Symulacja fal oraz interakcja ze światem

Efekty optyczne

Podczas generowania wody należy skupić się na dwóch ważnych efektach, bez których nie uda się stworzyć dobrze wyglądającej wody:

- Załamanie światła
- Odbicie światła

Prędkość światła jest różna, w zależności od gęstości ośrodka w jakim się znajduje (czym większa gęstość tym mniejsza jest szybkość światła). Powoduje to efekt załamania

promienia (kierunku), który przechodzi przez dwa ośrodki o różnych gęstościach. Aby poprawnie obliczyć załamanie światła należy skorzystać z prawa Snella: promień padający, promień załamany oraz wektor normalny powierzchni leżą w jednej płaszczyźnie, a kąty między nimi spełniają zależność:

$$\frac{\sin\theta_i}{\sin\theta_t} = \frac{n_2}{n_1} <=> n_1 sin\theta_i = n_2 sin\theta_t$$

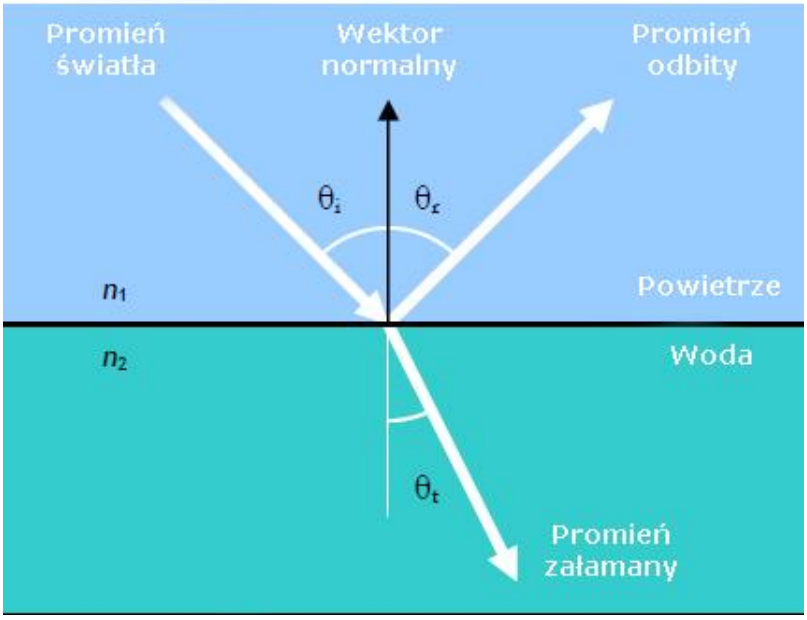
Gdzie:

- θ_i to kąt między wektorem światła i normalną powierzchni
- θ_t to kąt między normalną powierzchni oraz załamanym wektorem światła
- n_1 i n_2 to wartości załamania dla ośrodków

Poniższa tabelka prezentuje współczynniki załamania dla typowych materiałów.

Materiał	Współczynnik załamania
Próżnia	1,0
Powietrze	1,0003
Lód	1,31
Woda	1,333
Szkło	1,5
Plastik	1,5
Diament	2,417

Tabela 4.6.5. Współczynniki załamania dla niektórych materiałów. Dla szkła i plastiku 1,5 to tylko przybliżona wartość, ponieważ mogą być różne rodzaje tych materiałów.



Rysunek 4.6.32. Załamanie promienia na granicy ośrodków oraz odbicie promienia.

Wygląd wody ściśle zależy od kąta patrzenia. Gdy spojrzymy dokładnie w dół możemy zobaczyć co jest pod powierzchnią, ale gdy spojrzymy pod dużym kątem zobaczymy tylko, niemal, lustrzane odbicie. Dzieje się tak, dlatego, że gdy światło osiąga granicę dwóch ośrodków, część światła jest odbijana a część przenika do drugiego ośrodka. Jest to efekt Fresnela. Augustin-Jean Fresnel stworzył równanie, które wyznacza ile światła jest odbijane, a ile załamywane.

$$R = \left(\frac{\sin(\theta_i - \theta_t)}{\sin(\theta_i + \theta_t)}\right)^2 + \left(\frac{\tan(\theta_i - \theta_t)}{\tan(\theta_i + \theta_t)}\right)^2$$

Prawdopodobieństwo, że foton przeszedł przez granicę ośrodków wynosi:

$$T = 1 - R$$

Niestety równanie Fresnela może nie być zbyt wydajne, dlatego w aplikacjach czasu rzeczywistego lepiej jest skorzystać z aproksymacji:

$$R = \max\big(1, min\big(1, bias + scale \cdot (1 + dot(I, N)^{power})\big)\big)$$

Gdzie:

- R to współczynnik odbicia (reflectionCoefficient)
- $\max$ to funkcja maksimum
- min to funkcja minimum
- $bias$ to przesunięcie
- $scale$ to skala
- dot to iloczyn skalarny
- I to wektor widoku
- N to wektor normalny powierzchni
- $power$ to potęga efektu

Przedstawiona funkcja posiada następujące właściwości:

- Gdy I oraz N są niemal równoległe iloczyn skalarny daje wartość 0 lub bliską 0, czyli światło w większości jest załamywane.
- Gdy I oraz N stają się coraz bardziej prostopadłe, wówczas współczynnik odbicia bardzo szybko dąży do 1 (przez potęgowanie).
- Gdy I oraz N znacznie się różnią, niemal całe światło jest odbijane, a pozostała część jest załamywana.

We wzorze, współczynnik odbicia ograniczany jest do zakresu [0,1], ponieważ w końcowym wzorze należy zmieszać kolor odbicia i załamania:

$$C_{final} = R \cdot C_{reflected} + (1 - R) \cdot C_{reflected}$$

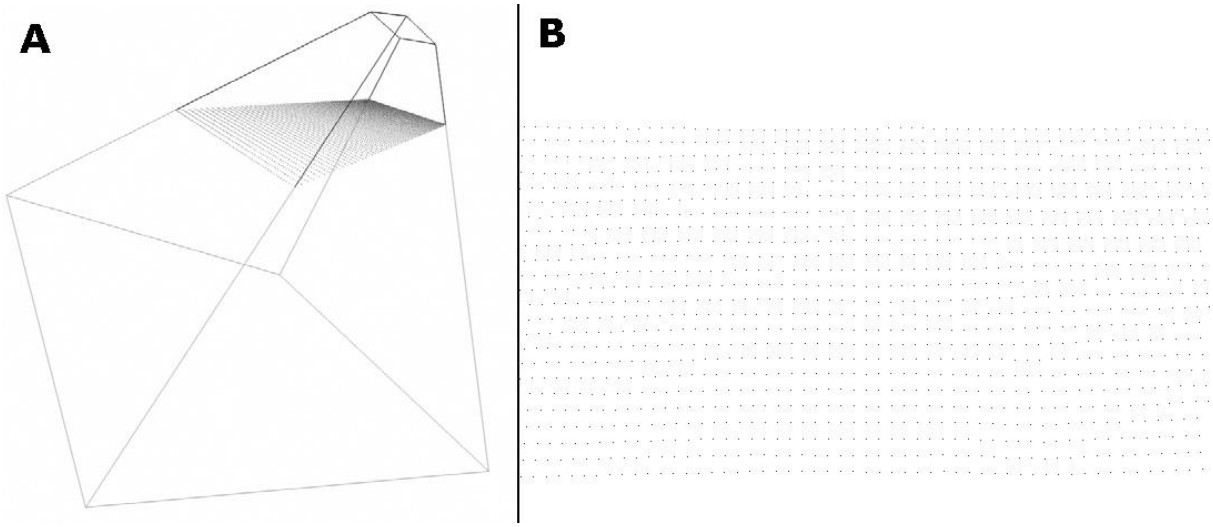
Gdzie:

- C_{final} to kolor końcowy
- R to współczynnik odbicia
- $C_{reflected}$ to kolor odbicia

Odbicie i załamanie to nie jedyne efekty optyczne związane z wodą. Wraz z głębokością wody barwy światła zanikają. Dla każdej składowej widma światła szybkość zaniku jest inna, ponieważ mają różne długości fali. W aplikacjach czasu rzeczywistego można przyjąć liniowy zanik światła wraz z głębokością. Światło przechodzące przez ośrodek załamuje się. Załamanie światła jest zależne także od jego barwy. Światło czerwone załamuje się bardziej niż światło niebieskie. Wynika z tego, że dla światła białego,

wszystkie jego składowe załamują się pod innym kątem. Ten efekt nazywa się rozszczepieniem chromatycznym i można go zaobserwować jako tęczę. W grafice komputerowej można przyjąć rozszczepienie tylko dla trzech kolorów: czerwonego, zielonego i niebieskiego. Jest to wystarczające przybliżenie, które daje dobre efekty. Aby odpowiednio zasymulować rozszczepienie chromatyczne musimy znać współczynniki załamania dla poszczególnych składowych. Zamiast obliczać współczynnik załamania dla jednego promienia światła należy teraz obliczyć dla trzech.

Skoro wiadomo już jak symulować optykę wody należy teraz zająć się jej geometrią. W skrócie, woda może być prostokątem złożonym z czterech wierzchołków. Oczywiście będzie wtedy bardzo nierealistycznie wyglądać. Jeżeli jednak nałożymy na powierzchnię teksturę oraz zastosujemy mapowanie normalnych z animacją współrzędnych tekstury możemy w prosty sposób uzyskać niewielkie fale. Symulując rzeki lub jeziora może dać to wystarczająco dobre efekty. W przypadku oceanu, gdzie są znacznie większe fale, trzeba odpowiednio animować geometrię wody. Do tego potrzebna jest dużo gęstsza siatka niż cztery wierzchołki. Niestety, tak gęsta siatka nie jest dobra, by pozwolić sobie na większy zbiornik wodny. Wyobraźmy sobie, że chcemy stworzyć wyspę otoczoną oceanem. Taka ilość wody wymaga bardzo dużej ilości wierzchołków do przetwarzania. W praktyce, taka woda nie różni się od terenu, więc można zastosować poziomy szczegółowości dla jej dalszych fragmentów. Jednakże nadal pozostaje problem przetwarzania, i tak, zbyt dużej ilości geometrii. Pomocna może się tutaj przydać technika Projected Grid'ów (rzutowanych kratek). Technika ta wyświetla prostokątną siatkę tylko w przestrzeni kamery. W praktyce, woda znajduje się tylko tam gdzie patrzy kamera. Jeśli się obróci woda (czyli prostokąt geometrii) obróci się razem z kamerą, oraz przemieści jeśli kamera się przemieści. Dzięki temu można symulować nieskończenie wielki ocean. Oczywiście minusem tej techniki jest to, że wszędzie jest woda. Aby nie było jej widać należy stworzyć odpowiednio wysoki teren (którego powierzchnia przykryje wodę). Kolejnym minusem tej techniki, wywodzącym się z poprzedniego, jest to, ze nawet jeśli tylko mała część wody jest widoczna (ponieważ akurat kamera, w większości, patrzy na teren) nadal jest renderowana cały Projected Grid.



Rysunek 4.6.33. A) Projected Grid widziany z boku. B) Ten sam Projected Grid widziany z kamery. Obrazki pobrane z [39].

Obojętnie jaką technikę wyświetlania geometrii wody wybierzemy należy zmierzyć się z dość trudnym problemem jakim jest realistyczna animacja fal.

Szybkie transformacje Fourier'a (FFT)

Fale morskie można symulować za pomocą FFT (Szybkie Transformacje Fouriera – Fast Fourier Transform). Szybkie tranformacje Fouriera jest to algorytm liczenia dyskretnej transformaty Fouriera oraz transformaty odwrotnej do niej. Transformata Fouriera ma postać:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}$$

Gdzie:

- $k = 0, \dots, N-1$
- $X_0, \dots, X_{n-1}$ są liczbami zespolonymi

Obliczenie sum za pomocą tego wzoru zajmuje $O(N^2)$ operacji. Szybkie transformacje Fouriera potrzebują jedynie $O(N \log_2 N)$ operacji. Algorytm FFT został użyty do symulacji wody w filmach takich jak Titanic czy Waterworld.

Przyjmijmy następujące notacje:

- k ilość fal
- $\boldsymbol{k}$ wektor fali
- T okres fali
- λ długość fali
- h wysokość wody
- $\boldsymbol{x}$ pozycja symulowanego punktu
- t czas
- g stała grawitacji
- P_h widmo Philipsa
- ξ wartość losowa Gauss'a ze średnią 0 i odchyleniem 1
- L największa możliwa fala powstała pod wpływem szybkości wiatru
- ω częstotliwość kątowa
- w kierunek wiatru

Chcemy symulować powierzchnię wody jako pole wysokości (heightfield) $h(\boldsymbol{x}, t)$, które jest sumą sinusoid z amplitudami zależnymi od czasu. Wzór na wysokość wody w punkcie **x** i czasie t ma postać:

$$h(\boldsymbol{x}, t) = \sum_{\boldsymbol{k}} \tilde{h}(\boldsymbol{k}, t) e^{i\boldsymbol{k}\boldsymbol{x}}$$

Gdzie:

- $\boldsymbol{k}$ jest wektorem $\left(k_x, k_y\right), k_x = \frac{2\pi n}{L_x}, k_y = \frac{2\pi m}{L_y}\ i - \frac{N}{2} \leq n < \frac{N}{2}, -\frac{M}{2} \leq m < \frac{M}{2}$

Ponieważ chcemy stworzyć powtarzającą się kratkę geometrii (reprezentującą powierzchnię wody), należy stworzyć zakres częstotliwości i pobrać jej odwrotną

transformatę Fourier'a co, z definicji, tworzy odpowiednie, przestrzenne, okresowe sygnały (pole wysokości).

Należy zdefiniować zbiór złożonych zakresów amplitud Fourier'a i odpowiadające im wartości faz dla pola wysokości fal w czasie 0.

$$\tilde{h}_0(\boldsymbol{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\boldsymbol{k})}$$

Widmo Philips'a jest często używanym modelem używanym do symulacji fal powodowanych przez wiatr.

$$P_h(k) = A\frac{e^{-\frac{1}{(kL)^2}}}{k^4}\left|\hat{\boldsymbol{k}} \cdot w\right|^2$$

Gdzie:

- $L = \frac{V^2}{g}$ Jest największą możliwą falą powstałą przez stałą szybkość wiatru V
- A Jest stałą, która wpływa na wysokość fal
- $\hat{\boldsymbol{k}} \cdot w$ to iloczyn skalarny

Eksponenta w liczniku może być użyta do zmniejszenia rozchodzenia się fal. $\left|\hat{\boldsymbol{k}} \cdot w\right|^2$ zapobiega temu, by fale poruszały prostopadle do kierunku wiatru.

Korzystając z obliczonych amplitud w czasie 0 można przystąpić do liczenia fal w czasie t.

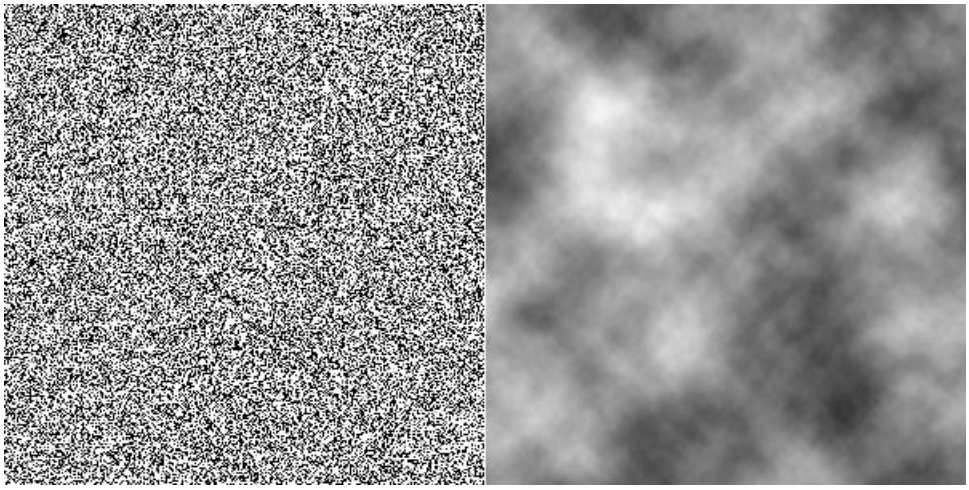
$$\tilde{h}(\boldsymbol{k}, t) = \tilde{h}_0(\boldsymbol{k})e^{i\omega(k)t} + \tilde{h}_0^*(-\boldsymbol{k})e^{-i\omega(k)t}$$

Gdzie:

- $\tilde{h}^*(\boldsymbol{k}, t) = \tilde{h}(-\boldsymbol{k}, t)$

Szum Perlin'a (Perlin Noise)

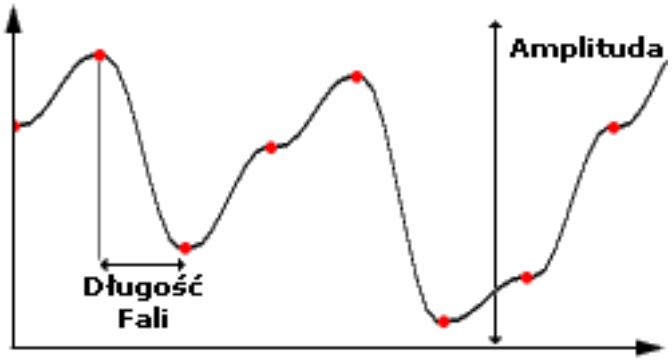
Fale oceaniczne można generować za pomocą szumu Perlin'a. Szum ten, to dwuwymiarowa tekstura wartości losowych, wygenerowana za pomocą algorytmu Ken'a Perlin'a.



Rysunek 4.6.34. Obrazy szumu. Lewa: standardowy szum. Prawa: szum Perlin'a

Funkcja szumu bazuje na generatorze liczb losowych. Generator przyjmuje pewną liczbę (ziarno) i na jej podstawie generuje ciąg liczb. Ciąg ten zawsze będzie taki sam jeśli do generatora podamy tą samą liczbę.

Podobnie jak w algorytmie FFT, szum Perlin'a charakteryzuje się amplitudą i długością fali. W wersji jednowymiarowej wykres szumu może mieć taką postać:



Rysunek 4.6.35. Wykres jednowymiarowej, interpolowanej funkcji szumu. Czerwone punkty to wartości funkcji. Obrazek pobrany z [82].

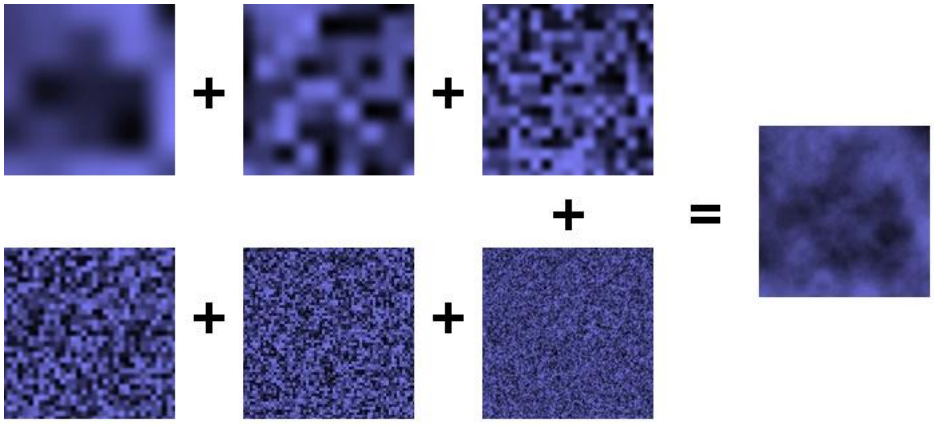
Jeżeli weźmiemy wystarczającą ilość funkcji szumu Perlin'a, z różnymi amplitudami i częstotliwościami, możemy je dodać do siebie otrzymując szum Perlin'a. Rysunek 16 prezentuje wersję algorytmu w dwóch wymiarach. Do wygenerowania wartości dla każdej funkcji zazwyczaj używa się amplitudy dwa razy mniejszej i częstotliwości dwa razy większej (poczynając od największej amplitudy i najmniejszej częstotliwości dla pierwszej funkcji). Ostatnia funkcja tworzy „najdrobniejszy" szum. Dla każdej funkcji częstotliwość i amplitudę można dobierać na różne sposoby, na przykład:

$$częstotliwość = 2^i, i \in [0, n)$$
$$amplituda = c^i, c \in (0,1], i \in [0, n)$$

Gdzie:

- c to pewna liczba rzeczywista
- i to i-ta funkcja, rozpoczynając od 0.

Przykładowe wartości dla częstotliwości z tego wzoru to: 1, 2, 4, 8, 16, 32. Dla amplitudy równej ¼ to: 1, 1/4, 1/16, 1/64, 1/256, 1/1024. Czym mniejsza amplituda tym funkcja ma łagodniejszy wykres.



Rysunek 4.6.36. Ilustracja obrazuje sześć funkcji szumu, które po zsumowaniu dają szum Perlin'a. Obrazki pobrane z [82].

Każdą funkcję jaką dodajemy nazwiemy oktawą ponieważ jej częstotliwość jest dwukrotnie większa od częstotliwości funkcji poprzedniej. Teoretycznie można dodawać nieskończenie wiele oktaw, ale w przypadku obrazu, zbyt duża ich ilość może nie dać oczekiwanego efektu. Powodem tego jest ilość pikseli, których może zabraknąć przy zbyt dużej częstotliwości (np. dwie fale na jeden piksel).

Sama funkcja szumu to po prostu zbiór losowych liczb z przedziału [-1,1]. Generator liczb losowych powinien na wyjście dawać liczbę zależną od liczby jaką dostaje na wejściu. Dla każdej oktawy funkcja powinna dawać nieco inny zbiór liczb. Gdy posiadamy już funkcję szumu należy jej wartości nieco wygładzić. Stosujemy do tego wybraną interpolację. W aplikacjach czasu rzeczywistego najlepiej wykorzystać interpolację liniową bo, chociaż daje niezbyt dobry rezultat, to jest szybka. Dodatkowo można jeszcze wygładzić wartości szumu, by dały mniej losowy efekt (zanim zastosujemy interpolację). Wygładzanie wartości funkcji szumu polega na uśrednieniu poprzedniej, następnej i aktualnej wartości. Można to zrobić stosując następujący wzór:

$$SmoothNoise(x) = \frac{Noise(x)}{2} + \frac{Noise(x-1)}{4} + \frac{Noise(x+1)}{4}$$

Gdzie:

- $Noise(x)$ jest funkcją szumu, czyli generuje losową liczbę na podstawie wartości

Dla szumu dwuwymiarowego funkcja będzie miała następującą postać:

$$SmoothNoise(x,y) = A(x,y) + B(x,y) + C(x,y)$$

Gdzie:

$$A(x,y) = \frac{Noise(x-1,y-1) + Noise(x+1,y-1) + Noise(x-1,y+1) + Noise(x+1,y+1)}{16}$$

$$B(x,y) = \frac{Noise(x-1,y) + Noise(x+1,y) + Noise(x,y-1) + Noise(x,y+1)}{8}$$

$$C(x,y) = \frac{Noise(x,y)}{4}$$

W rezultacie jednowymiarowy algorytm szumu Perlin'a ma następującą postać:

$$Perlin(x) = \sum_{n-1}^{i=0} Interpolate_i(x * frequency) * amplitude$$

Gdzie:

- n to ilość wszystkich oktaw
- $frequency$ to częstotliwość dana wzorem 2^i
- $amplitude$ to amplituda dana wzorem p^i
- p to stała z przedziału (0,1]
- $Interpolate_i$ to funkcja interpolująca.

Indeks przy funkcji interpolującej oznacza, że dla każdego przebiegu pętli korzystamy z innej funkcji. Każda oktawa ma swoją funkcję interpolującą ponieważ ma inną funkcję szumu. Tak jak wcześniej zostało napisane, dla każdej oktawy należy mieć inny szum. Generator liczb losowych każdej oktawy powinien zwracać nieco inne liczby dla tego samego wejścia. Inaczej mówiąc, każda oktawa posiada własny generator, ale nie może być to ten sam. W praktyce, oktawy różnią się jedynie funkcją szumu.

Funkcja Interpolate może mieć taką postać:

$$Interpolate(x) = LinearInterpolate(SmoothNoise(x), SmoothNoise(x+1), c_x)$$

Gdzie:

- $LinearInterpolate$ to funkcja liniowej interpolacji
- c_x to współczynnik przesunięcia między pierwszą a drugą wartością funkcji interpolującej z przedziału [0,1]. Można przyjąć, że ten współczynnik ma wartość równą części ułamkowej argumentu x.

Wersja dwuwymiarowa szumu Perlina jest niemal identyczna, różnica leży w funkcji Interpolate:

$$Interpolate(x, y) = LinearInterpolate(A(x, y), B(x, y), c_y)$$

Gdzie:

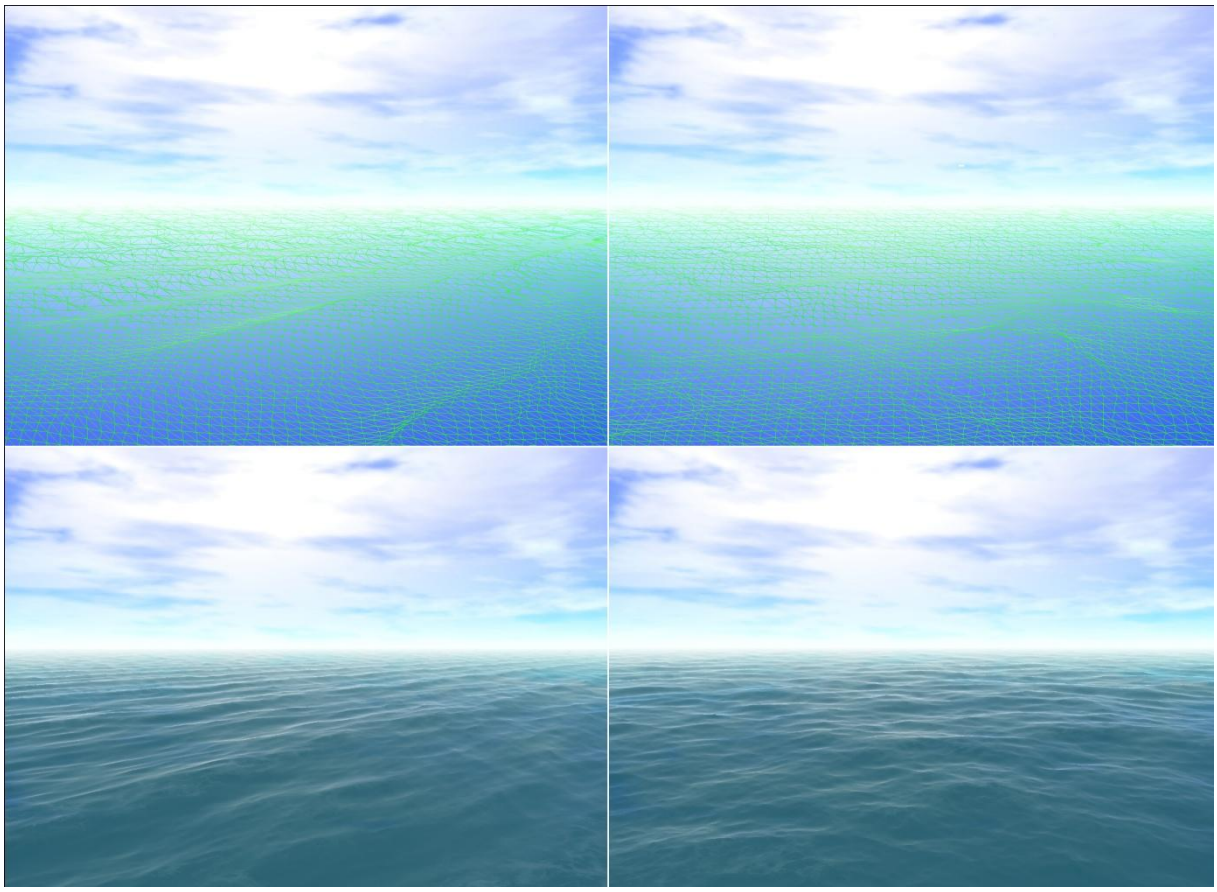
$$A(x, y) = LinearInterpolate(SmoothNoise(x, y), SmoothNoise(x + 1, y), c_x)$$

$$B(x, y) = LinearInterpolate(SmoothNoise(x, y + 1), SmoothNoise(x + 1, y + 1), c_x)$$

Sama dwuwymiarowa funkcja szumu ma postać:

$$Perlin(x, y) = \sum_{n-1}^{i=0} Interpolate_i(x * frequency, y * frequency) * amplitude$$

Szumu Perlin'a używa się nie tylko do generowania fal. Bardzo często stosuje się go do tworzenia proceduralnych tekstur (na przykład marmuru), terenu (mapa wysokości stworzona za pomocą tego szumu daje dobrze wyglądający teren), lub chmur.



Rysunek 4.6.37. Fale generowane algorytmem FFT (lewa kolumna) oraz szumem Perlin'a (prawa kolumna). W przykładzie został wykorzystany Projected Grid.

4.7. Animacja

Niemal żadna gra komputerowa nie może obejść się bez animacji. To właśnie animacje nadają realizmu generowanej scenie. W grach komputerowych zazwyczaj wcielamy się w jakąś postać. Czasem akcja jest widziana z oczu bochatera a czasem zza jego pleców. Animacje postaci stanowią nieodzowny element wielu gier komputerowych. Poziom realizmu animacji automatycznie wpływa na realizm całej gry.

4.7.1. Animacja za pomocą ujęć kluczowych

Podstawową techniką animacji są ujęcia kluczowe (key-frame). Obecnie już bardzo rzadko, lub wcale nie używane jednak warto się przyjrzeć tej technice ze względu na podstawy jakie wnosi do animacji.

Technika ujęć kluczowych polega na utworzeniu zbioru ujęć stanów animacji. Ujęcia te przedstawiają tylko najważniejsze pozycje obiektu. Gdybyśmy chcieli odtworzyć taką animację wyglądała by ona jako sztywne przeskoki z pozycji do pozycji. Dobrą metodą sprawienia by animacja była płynna jest interpolacja. Dzięki niej można utworzyć płynne przejścia między klatkami kluczowymi.

Najprostszą metodą interpolacji jest interpolacja liniowa. Między dwoma sąsiednimi ujęciami u0 i u1 tworzymy linię, na której określamy punkt.

$$u(t) = u0 + t(u1 - u0)$$

gdzie:

- t to czas między ujęciami u0 i u1

Jeśli znamy czas nowego położenia, ilość wszystkich ujęć oraz łączny czas trwania animacji można obliczyć punkt między dwoma sąsiednimi ujęciami kluczowymi. Dzięki tej metodzie łatwo można uzyskać dowolne ujęcie animacji, dzięki czemu uzyskujemy płynność animacji w czasie. Można określić procentowo położenie pośredniego ujęcia. Na przykład 0 to ujęcie u0 a 1 to ujęcie u1 (gdzie 1 to 100%). Znając procentowe położenie ujęcia pośredniego (między u0 i u1) można obliczyć pozycję wierzchołków dla tego ujęcia.

Niestety interpolacja liniowa ma pewną wadę. Niektóre siatki, podczas interpolacji animacji, mogą się zdeformować. Dlatego warto skorzystać z bardziej zaawansowanej metody interpolacji, na przykład krzywe sklejane Hermite'a (dla animacji jest to lepsza metoda niż np. b-spline ponieważ daje większą kontrolę nad przebiegiem animacji). Interpolacja ta bierze dwa sąsiednie ujęcia kluczowe po każdej stronie pożądanego położenia. Metoda wykorzystująca tą technikę sprawdza, które ujęcia kluczowe znajdują się po każdej stronie żądanego czasu animacji. Dzięki tym nowym ujęciom można poprawić obliczenia nowego położenia.

Aby obliczyć nowe położenie należy użyć czterech ujęć kluczowych.

$$u(t) = (2t^3 - 3t^2 + 1)u_0 + (t^3 - 2t^2 + t)m_0 + (t^3 - t^2)m1 + (-2t^3 + 3t^2)u_1$$

$$m_i = \left(1 - \frac{a}{2}\right)((u_i - u_i - 1) + (u_i + 1 - u_i))$$

Dzięki pierwszemu i czwartemu ujęciu obliczamy tangens m_i dla ujęć: pierwsze i drugie oraz trzecie i czwarte.

4.7.2. Animacja za pomocą kości

W animacji szkieletowej (skeletal animation) do animowania siatki (wierzchołków) modelu używa się szkieletu. Model taki podzielony jest na dwie części: skóra (skin) czyli siatka modelu oraz szkielet (skeleton) czyli hierarchiczna struktura drzewiasta złączeń – stawów (joints) łączących kości (bones), które tworzą kinematyczny model postaci (lub innej, animowanej geometrii). Każda kość ma wpływ na jeden lub więcej wierzchołków. W najprostszej wersji każdy wierzchołek ma przypisaną tylko jedną kość. Dla przykładu można wziąć rękę, która składa się z kości ramienia i przedramienia. Poruszając odpowiednią kością poruszamy odpowiednią częścią ręki. Kości mogą mieć na siebie wpływ. Jeśli poruszymy kością ramienia to poruszy się także przedramię, ale jeśli poruszymy kością przedramienia ramię nie musi się poruszyć.

Jeżeli na każdy wierzchołek będzie miała wpływ tylko jedna kość wówczas mogą powstać dziury w geometrii.

Rysunek 5.7.38. Z lewej: model ręki. Lewy górny róg: zgięta ręka, widać lukę w geometrii. Prawy górny róg: usunięte punkty. Lewy dolny róg: wypełnienie szczelin. Prawy dolny róg: poprawnie zgięta (animowana) ręka. Obrazek pobrany z [88].

Aby zapobiec powstawianiu szczelin korzysta się z techniki zszywania (stitching) lub odkształceń siatki (skinning). W technice zszywania każdy wierzchołek może być przekształcany przez inną macierz – kość, do której ten wierzchołek jest przypisany. W praktyce tworzone są wielokąty, które zszywają ze sobą kilka kości wypełniając powstałe szczeliny. W standardowej metodzie animacji szkieletowej kość posiada informację o geometrii, którą animuje, a macierz tej kości przekształca te wierzchołki. W technice

zszywania każdy wierzchołek zawiera informację, do której kości należy. Aby animować postać należy przejść przez wszystkie wierzchołki i przekształcić je przy pomocy macierzy obrotu oraz macierzy animacji odpowiedniej kości. Niestety technika zszywania posiada pewną wadę – w przypadku dużego obrotu łączenia geometria może wyglądać bardzo nienaturalnie. Czym większa szczelina powstaje na zgięciach tym gorzej wygląda efekt końcowy.

Technika skinning'u zapobiega temu problemowi. Tutaj każdy wierzchołek może być modyfikowany przez kilka kości (macierzy). W przypadku ludzkiego ciała ta technika sprawdza się bardzo dobrze. Skóra na łokciu przy zgiętej ręce nie jest modyfikowana tylko przez jedną kość, ale przez kość ramienia oraz przedramienia. Wobec tego każdy wierzchołek musi zawierać listę kości jakie na niego wpływają, ale żadna kość nie wpływa w ten sam sposób, dlatego należy wprowadzić dodatkowo wagi. Waga mówi o tym z jaką siłą dana kość wpływa na wierzchołek. Najprościej jest przyjąć liniowy wpływ wag, czyli suma wszystkich wag wynosi 1. Z tego wynika, że dla n kości trzeba przechowywać n-1 wag ponieważ:

$$WagaOstatniejKosci = 1 - (waga_1 + waga_2 + \ldots + waga_{n-1})$$

Aby animować poprawnie wierzchołek należy przekształcić go przez wszystkie macierze jakie na niego wpływają oraz wagi. Robi się to korzystając z następującego równania:

$$wierzchołek = (wierzchołek * macierz_0 * waga_0) + (wierzchołek * macierz_1 * waga_1) + \ldots + (wierzchołek * macierz_N * waga_N)$$

Jest to liniowa interpolacja między przekształcanymi wierzchołkami. Problem tej techniki polega na tym, że jest ona droga obliczeniowo.

Interpolacja liniowa, podobnie jak w przypadku animacji za pomocą ujęć kluczowych, może nie dać idealnych rezultatów. Ponadto operacje na macierzach są drogie obliczeniowo. Wobec tego warto zastosować interpolację krzywymi sklejanymi przy zastosowaniu kwaternionów i dopiero na sam koniec z otrzymanych wyników stworzyć odpowiednią macierz.

5. Podsumowanie

Gry komputerowe, od przynajmniej dwudziestu lat, wrastają w naszą kulturę. Technologia stale idzie do przodu w zastraszającym tempie, a kolejne pokolenia wychowują się przy komputerach. Można powiedzieć, że gra komputerowa jest to interaktywny film. Cechą każdego filmu jest opowiedzenie pewnej historii. W grach mamy to samo, tylko że widz może w tym aktywnie uczestniczyć. Implikuje to stwierdzenie, że gry są sztuką, ponieważ mogą zawierać bogaty tekst, muzykę a nawet poruszać poważne problemy. Branża gier, która coraz bardziej się rozwija, przynosi większe dochody niż filmy. Na całym świecie każdego dnia przybywa coraz więcej nowych osób lubiących interaktywną, elektroniczną rozrywkę. Obecnie coraz większą popularnością cieszą się gry korzystające z internetu (MMO), gdzie jednocześnie łączą się ze sobą tysiące ludzi budując żywy świat. Niektóre gry nie wymagają już nawet kupna i instalacji, można w nie grać przez przeglądarkę (QuakeLive). Powstają pomysły gier zupełnie przez przeglądarkę (OnLive). Nie trzeba mieć nawet odpowiednio dobrego komputera, żeby cieszyć się grafiką najwyższej jakości. Wszystkie obliczenia wykonywane są na serwerze, który wysyła do przeglądarki użytkownika obrazy – kolejne klatki gry. Problemem jak na razie jest szybkość internetu. Ponieważ każda klatka, czyli obraz w najwyższej rozdzielczości, nawet po kompresji może być zbyt duży. Do płynnej animacji gry potrzeba przesłać przynajmniej 25 obrazów w ciągu sekundy. Tego typu projekty zakończą problem piractwa w kontekście gier. Użytkownicy komputerów PC nie będą zmuszani do ciągłego zmieniania sprzętu, a konsole prawdopodobnie zostaną odłożone.

Jeżeli chodzi o sam proces tworzenia gier, to staje się on coraz bardziej złożony. Gracze stają się coraz bardziej wymagający. Zależy im na świetnej fabule oraz efektach wizualnych. Nowe gry wyznaczają kierunek i ustanawiają poziom, któremu musi sprostać każda kolejna produkcja, jeśli chce odnieść sukces. Zespoły programistów stale się rozszerzają. Powstaje coraz więcej pomysłów, rozwiązań i technologii, które trzeba opanować, żeby nadążyć za ciągle rozwijającym się rynkiem. Ogrom wiedzy może przerosnąć nawet najbardziej doświadczonych.

Mam nadzieję, że moja praca choć trochę naświetla czym jest programowanie gier. Do tej dziedziny zalicza się niemal wszystko co ma związek z komputerami, a każdy z tych elementów musi być opanowany do perfekcji przez osobę, która się tym zajmuje. Rozdziały, które napisałem zawierają elementarną wiedzę na podstawowe tematy związane z programowaniem gier. Każdy wie, jak bardzo wzorce projektowe są przydatne w procesie tworzenia zaawansowanych aplikacji. Te które opisałem przejawiają się bardzo często. Singleton'y powinny być używane z rozwagą. Najlepiej przekazywać takie obiekty jako parametr, dzięki czemu łatwiej będzie potem robić refactoring, chociaż jest to trochę sprzeczne z naturą używania tego wzorca. Fabryka przydatna jest szczególnie w przypadku tworzenia implementacji interfejsu. Można też użyć tego wzorca do tworzenia poszczególnych jednostek w grze. Jest to pewne przysłonięcie operacji, które powinny być zawsze wykonywane do utworzenia obiektów danego typu. Inteligentne wskaźniki mogą bardzo ułatwić zarządzanie zasobami poprzez zliczanie referencji. Zapobiegają też ewentualnym wyciekom pamięci i stanowią pewną abstrakcję dla wskaźników, jednocześnie zbliżając ich wygląd do tych znanych z Java czy C#.

Metody zarządzania pamięcią powinny stanowić fundament każdego większego projektu pisanego w języku C/C++. Ze względu na fragmentację pamięci nie należy do końca polegać na samym systemie operacyjnym. Tworząc jeden, ciągły blok o stałym, lub zmiennym rozmiarze (co może wpłynąć na wydajność w pewnych okolicznościach), zapobiegamy fragmentacji oraz otrzymujemy dużo szybszy mechanizm przydzielania i zwalniania pamięci. W wielu aplikacjach wąskim gardłem okazuje się biblioteka STL. Między innymi dlatego, że obiekty tej biblioteki dynamicznie alokują pamięć już w konstruktorze. Należy pamiętać też o tym, że standard STL nie mówi nam nic o zużyciu pamięci. Dlatego dobrym pomysłem jest napisanie własnych pojemników, wydajnych dla zastosowań, których aktualnie potrzebujemy.

Wzorce projektowe i manager'y pamięci są dobrą podstawą do stworzenia manager'ów zasobów. Manager zasobów może mieć postać fabryki obiektów (w praktyce korzystać z tego wzorca), nie tylko przechowywać pliki odczytywane z dysku. Dobre zarządzanie zasobami jest jednym z najważniejszych elementów w programowaniu gier, ponieważ każdy z tych programów w dużej mierze korzysta z tekstur, dźwięków, skryptów, geometrii, tekstów. Dlatego należy solidnie przyłożyć się podczas tworzenia tego podsystemu. Nie tylko powinien działać szybko, ale też posiadać wiele możliwości manipulacji zasobami.

Nic tak nie zniechęca jak powolna gra. Jeżeli gracz ma czekać kilka minut na załadowanie poziomu, po pewnym czasie może stracić nerwy i odinstalować aplikację. Szybkość ładowania zasobów jest bardzo ważna. Manager zasobów powinien współdziałać z podsystemem odczytującym dane z dysku. Ten podsystem z kolei powinien mieć możliwość odczytu plików z systemowego systemu plików, oraz z wirtualnego systemu plików gry. Jeżeli skompresujemy pliki, wówczas możemy otrzymać znaczny wzrost wydajności podczas ładowania zasobów do gry. Sam odczyt z wirtualnego systemu plików może odbywać się kompleksowo, jeżeli dane umieścimy w ciągłym bloku (na przykład dane poziomu: 1. gry znajdują się od 12 do 134 bajtu), który odczytamy jedną operacją. Sens w używaniu VFS pojawia się dopiero pod koniec projektu, gdy nie potrzeba już ingerować w zasoby.

W sztucznej inteligencji bardzo ważne jest by była jak najbardziej przekonująca. Gracze komputerowi powinni podejmować rozsądne decyzje, jak najbardziej zbliżone do tych jakie podjął by człowiek. Automat stanów to prosty system symulujący pewien zbiór zachowań i przejść między nimi. Bazując na pewnych regułach w prosty sposób można stworzyć mechanizm zachowań w konkretnych sytuacjach. Automaty stanów można połączyć z wyszukiwaniem drogi między dwoma punktami. Do tego zadania od wielu lat używany jest algorytm A*. Stosując odpowiednie optymalizacje dla wyszukanych ścieżek można uzyskać naturalnie wyglądającą drogę łączącą dwa punkty (jeżeli między nimi istnieje przejście).

Gry komputerowe wiele ze swojego uroku zawdzięczają grafice trójwymiarowej. Obecne gry wyświetlają nawet do kilku setek tysięcy polygon'ów na raz. Aby było to możliwe należy odpowiednio podzielić przestrzeń i usuwać niewidoczną geometrię. Dzięki drzewom ósemkowym można skutecznie zminimalizować ilość obiektów gry, które na siebie współoddziaływają (na przykład detekcja zderzeń). Stosowanie poziomów szczegółowości dla geometrii jest kolejnym dobrym przykładem optymalizacji w grafice. Modele widziane z większej odległości nie muszą posiadać wielu polygon'ów, gdyż gracz tego i tak nie zauważy. Podobna zasada tyczy się oświetlenia. Z bliska można zastosować

algorytm bump mapping'u, lub parallax mapping'u, ale już z nieco dalszej odległości może być to model Phong'a. Okazuje się, że właśnie stosowanie LOD dla materiałów (szczególnie oświetlenie) jest skuteczniejszą metodą optymalizacji niż LOD dla geometrii. Poniższe obrazki przedstawiają możliwości kart graficznych sprzed dwóch lat.

Rysunek 5.39. Z lewej: prawdziwy świat. Z prawej: możliwości silnika CryEngine2 (2008r).

Rysunek 5.40. Z lewej: prawdziwy świat. Z prawej: możliwości silnika CryEngine2 (2008r).

Obrazki powyżej przedstawiają możliwości współczesnej technologii z dziedziny grafiki trójwymiarowej i pokazują jak bardzo, już dzisiaj, możemy zbliżyć się do realizmu, o który przecież twórcom gier chodzi.

6. Słowniczek

cRPG – (Computer Role-Playing Game) Jest to typ gry, w którym rozwinięta jest fabuła, a bohater gry w trakcie przygody może rozwijać swoje umiejętności. Przykłady: Baldur's Gate, Dragon Age: Origins, Fallout

FPP – (First Person Perspective) Typ gry komputerowej, w której kierujemy postacią z perspektywy jej oczu. Przykłady: Crysis, Bioshock

FPS – (First Person Shooter) Typ gry komputerowej, w którym obserwujemy rozgrywkę z oczu bohatera. Są to gry zręcznościowe, polegające na strzelaniu z różnego rodzaju broni palnej. Przykłady: Quake 3, Unreal Tournament.

FPS – (Frames Per Second) W grafice komputerowej termin ten oznacza ilość wyświetlanych klatek w ciągu jednej sekundy.

Particle – cząsteczka. W grafice komputerowej używane są efekty oparte na cząsteczkach (na przykład ogień). Najczęściej, do symulacji cząsteczki używany jest trójwymiarowy punkt o konkretnej wielkości.

Piksel – (pixel – picture element) Najmniejszy element obrazu. Zazwyczaj składa się z trzech składowych koloru: czerwonego, zielonego i niebieskiego (w zależności od formatu obrazu).

Polygon – W grafice trójwymiarowej jest to najmniejszy wielokąt z jakiego składa się siatka geometrii. Jest to przeważnie trójkąt.

RGB – Format koloru składający się z trzech składowych: czerwony (R), zielony (G), niebieski (B). Każda ze składowych zapisana jest jako liczba całkowita o rozmiarze jednego bajtu (czyli o zakresie [0,255]).

Siatka geometrii – To w grafice trójwymiarowej zbiór polygon'ów połączonych ze sobą krawędziami reprezentujący pewien kształt (np. kula, człowiek, samochód).

Shader – program wykonywany na procesorze GPU przeznaczony do obróbki wierzchołków (vertex), pikseli (fragment/pixel), trójkątów (geometry).

Tekstura – (texture) Obraz używany w grafice komputerowej do nakładania na siatki geometrii w celu oddania materiału z jakiego jest zrobiona (np. ceglana ściana). Obrazy te przechowywane są na dysku lub innym nośniku danych (systemie plików) w formatach takich jak: dds, png, jpg, bmp, tga i inne.

Teksel – (texel – texture element) Najmniejszy element tekstury. W zależności od formatu tekstury może składać się z koloru o jednej lub kilku składowych w formacie całkowitym (jednobajtowym) lub zmiennoprzecinkowym (dwu lub czterobajtowym).

7. Bibliografia

1. Stanley B. Lippman i Josee Lajoie. *Podstawy Języka C++*. Wydawnictwa Naukowo-Techniczne, Warszawa 2001.
2. Charlez Petzold. *Programowanie Windows*. Wydawnictwo RM, Warszawa 1999.
3. Randima Fernando, Mark J. Kilgard. *The CG Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Pearson Education Inc. Addison Wesley Professional. 2003.
4. Andrei Alexandrescu. *Nowoczesne projektowanie w C++*. Wydawnictwo Naukowo-Techniczne. Warszawa. 2005.
5. Scott Bilas. *An Automatic Singleton Utility*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
6. James Boer. *Using the STL in Game Programming*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
7. Scott Bilas. *A Generic Handle-Based Resource Manager*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
8. James Boer. *Resource And Memory Management*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
9. Steven Ranck. *Frame-Based Memory Allocation*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
10. John Olsen. *Interpolation Methods*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
11. Steve Rabin. *Designing a General Robust AI Engine*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
12. Eris Dybsand. *A Finite-State Machine Class*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
13. Bryan Stout. *The Basics of A* for Path Planning*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
14. Steve Rabin. *A* Aesthetic Optimizations*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
15. Steve Rabin. *A* Speed Optimizations*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
16. Greg Snook. *Simplified 3D Movement and Pathfinding Using Navigation Meshes*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
17. Dan Ginsburg. *Octree Construction*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
18. Herbert Marselas. *Interpolated 3D Keyframe Animation*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
19. Torgeir Hagland. A Fast and Simple Skinning Technique. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
20. Ryan Woodland. *Filling the Gaps – Advanced Animation Using Stitching and Skinning*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
21. Jason Shankel. *Fractal Terrain Generation – Midpoint Displacement*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.
22. Jason Shankel. *Fractal Terrain Generation – Particle Deposition*. Game Programming Gems 1. CHARLES RIVER MEDIA, INC. 2000.

23. Andrew Kirmse. *Optimization for C++ Games*. Game Programming Gems 2. CHARLES RIVER MEDIA, INC. 2001.
24. Yossarian King. *Floating-Point Tricks: Improving Performance with IEEE Floating Point*. Game Programming Gems 2. CHARLES RIVER MEDIA, INC. 2001.
25. Greg Snook. *Simplified Terrain Using Interlocking Tiles*. Game Programming Gems 2. CHARLES RIVER MEDIA, INC. 2001.
26. Charles Farris. *Function Pointer-Based, Embedded Finite-State Machines*. Game Programming Gems 3. CHARLES RIVER MEDIA, INC. 2002.
27. David L. Koening. *Faster File Loading with Access-Based File Reordering*. Game Programming Gems 6. CHARLES RIVER MEDIA, INC. 2006.
28. Paul Glinker. *Fight Memory Fragmentation with Templated FreeLists*. Game Programming Gems 4. CHARLES RIVER MEDIA, INC. 2003.
29. Anger Fog. *Optimizing software in C++ An optimization guide for Windows, Linux and Mac platforms.* Copenhagen University College of Engineering. 2010.
30. Terry Welsh. *Parallax Mapping with Offset Limiting: A PerPixel Approximation of Uneven Surfaces*. Infiscape Corporation. Revision 0.3. 2004.
31. Natalya Tatarchuk. *Practical Dynamic Parallax Occlusion Mapping*. 3D Application Research Group ATI Research, Inc. SIGGRAPH 2005.
32. Natalya Tatarchuk. *Practical Parallax Occlusion Mapping For Highly Detailed Surface Rendering*. 3D Application Research Group ATI Research, Inc. GDC 2006.
33. Matayas Pemecz. *Iterative Parallax Mapping with Slope Information*. Department of Control Engineering and Information Technology. Budapest University of Technology.
34. Greg Ward. *High Dynamic Range Imaging.* Exponent – Failure Analysis Assoc. Menlo Park, California.
35. Bjorn Gustaffson, Linus Hilding. *High Dynamic Range Rendering in Real-Time.* 2007.
36. Nolan Goodnight, Rui Wang, Cliff Woolley and Greg Humphreys. *Interactive Time-Dependent Tone Mapping Using Programmable Graphics Hardware.* Department of Computer Science , University of Virginia. 2003.
37. Grzegorz Krawczyk, Karol Myszkowski, Hans-Peter Seidel. *Perceptual Effects in Real-time Tone Mapping.* MPI Informatik, Saarbrucken, Germany.
38. Christian Luksch. *Realtime HDR Rendering.* Institute of Computer Graphics and Algorithms, TU Vienna. 2007.
39. Claes Johanson. *Real-time water rendering. Introducing the projected grid concept.* Lound University. 2004.
40. Jason L. Mitchell. *Real-Time Synthesis and Rendering of Ocean Water.* ATI Research Technical Report. 2005.
41. Vladimir Belyaev. *Real-time simulation of water surface.* Applied Math. Department, St. Petersburg State Polytechnical University, St. Petersburg , Russia. 2003.
42. Rene Truelsen. *Real-time Shallow Water Simulation and Environment Mapping Clouds.* Department of Computer Science, University of Copenhagen. 2007.
43. Lesse Staff Jensen, Robert Golias. *Deep-Water Animation and Rendering.*
44. Kurt Pelzer. *Rendering Countless Blades of Waving Grass.* GPU Gems. 2004.
45. Cem Cebenoyan. *Graphics Pipeline Performance.* GPU Gems. 2004.
46. David Whatley. *Toward Photorealism in Virtual Botany.* GPU Gems 2. 2005.
47. Tiago Sousa. *Generis Refraction Simulation.* GPU Gems 2. 2005.
48. Simon Green. *Implementing Improved Perlin Noise*. GPU Gems 2. 2005.
49. http://gamedev.net
50. http://gamedev.pl
51. http://gamasutra.com
52. http://stackoverflow.com
53. http://flipcode.com/archives/
54. *http://groups.google.pl/group/pl.comp.lang.c/topics?pli=1*

55. http://msdn.microsoft.com
56. http://www.custard.org/~andrew/optimize.php (2010-07-02)
57. http://knol.google.com/k/shader-fx-parallax-mapping# (2010-08-20)
58. http://www.gamasutra.com/view/feature/1565/fast_file_loading_pt_1.php?page=2 (2010-06-23)
59. http://wiki.gamedev.net/index.php/D3DBook:High-Dynamic_Range_Rendering (2010-08-15)
60. http://www.gamasutra.com/view/feature/3942/data_alignment_part_1.php (2010-09-10)
61. http://wiki.polycount.com/Parallax%20Map (2010-07-05)
62. http://en.wikipedia.org/wiki/Octree (2010-07-01)
63. http://oktan.fm.interia.pl/optimum.htm (2010-07-01)
64. *http://gamasutra.com/features/20061003/kylmamaa_01.shtml* (2010-08-10)
65. *http://en.wikipedia.org/wiki/Gaussian_blur* (2010-08-10)
66. *http://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm* (2010-08-10)
67. *http://www.gamerendering.com/2008/10/11/gaussian-blur-filter-shader/* (2010-08-10)
68. *http://theinstructionlimit.com/?p=14* (2010-08-10)
69. *http://en.wikipedia.org/wiki/Full_screen_bloom* (2010-08-12)
70. *http://en.wikipedia.org/wiki/Convolution* (2010-08-12)
71. *http://en.wikipedia.org/wiki/Airy_disc* (2010-08-12)
72. *http://en.wikipedia.org/wiki/Tone_mapping* (2010-08-12)
73. *http://pl.wikipedia.org/wiki/Promieniowanie_elektromagnetyczne* (2010-08-14)
74. *http://pl.wikipedia.org/wiki/Kandela* (2010-08-14)
75. *http://www.fotohdr.pl/index.php/menu-fotohdr/menu-photohdr-image.html* (2010-08-14)
76. *http://www.fotohdr.pl/index.php/menu-fotohdr/menu-photohdr-dynamic-range.html* (2010-08-14)
77. *http://pl.wikipedia.org/wiki/Obraz_HDR* (2010-08-14)
78. *http://en.wikipedia.org/wiki/High_dynamic_range_imaging* (2010-08-14)
79. *http://en.wikipedia.org/wiki/High_dynamic_range_rendering* (2010-08-14)
80. *http://pl.wikipedia.org/wiki/Szybka_transformacja_Fouriera* (2010-08-22)
81. *http://fatcat.ftj.agh.edu.pl/~chwiej/mn/fft_2110.pdf* (2010-08-22)
82. *http://freespace.virgin.net/hugo.elias/models/m_perlin.htm* (2010-08-22)
83. *http://dunnbypaul.net/perlin/* (2010-08-22)
84. http://www.programmersheaven.com/2/perlin (2010-08-22)
85. http://www.iquilezles.org/www/articles/morenoise/morenoise.htm (2010-08-22)
86. http://www.opengl.org/resources/code/samples/advanced/advanced97/notes/node73.html (2010-06-30)
87. http://en.wikipedia.org/wiki/Heightmap (2010-08-06)
88. Andreas Vasilakis, Ioannis Fudos. *SKELETON-BASED RIGID SKINNING FOR CHARACTER ANIMATION.* Department of Computer Science, University of Ioannina, Ioannina, Grece. 2009.