Wybrane techniki programowania gier

Dariusz Maciejewski

Spis treści

Streszczenie

Poniższy esej ma na celu przedstawienia dwóch technik programistycznych wykorzystywanych w tworzeniu gier. Zastosowanie singletonów umożliwia efektywne i przejrzyste odwoływanie się do podsystemów gry. Wykorzystanie szablonu menadżera zasobów bazującego na uchwytach pozwala natomiast na scentralizowane, szybkie i bezpieczne zarządzanie licznymi danymi, z jakich korzystają gry.

Automatyczne singletony

Co to jest singleton?

Singleton jest to obiekt, który może być utworzony tylko w jednej instancji. W grze komputerowej singletonem będzie menadżer tekstur, dźwięków lub plików, może to być także obiekt przedstawiający aplikację, bądź też interfejs użytkownika.

Dlaczego warto ich używać?

Stosując odpowiednie nazewnictwo singletonów, mamy zapewniony czytelny dostęp do kluczowych elementów programu. Bez singletonów musielibyśmy przedzierać się przez wielopoziomową hierarchię obiektów w stylu: GetApp()->GetServices()->GetGui()->GetTextureMgr().

Jest to niewygodne dla programisty i mało wydajne z powodu kilku dereferencji. Do obiektu singleton odnosimy się bezpośrednio przez nazwę jego typu.

Singleton a globalny obiekt

Singletony zapewniają prosty i czytelny dostęp do obiektów które reprezentują. Tak prosty, jak obiekty globalne. Czemu więc nie poprzestać na obiektach globalnych? Kolejność ich tworzenia i niszczenia jest zależna od implementacji i trudno przewidzieć jej działanie przy przenoszeniu między różnymi systemami. Dzięki singletonom sami zdecydujemy kiedy obiekt ma być utworzony, a kiedy zniszczony, jednocześnie mając możliwość odwoływania się do nich jak to globalnego obiektu.

Rozwiązanie ”szkolne”

Istnieje kilka wzorców tworzenia singletonów, niektóre uwzględniają wielowątkowy dostęp do obiektu, inne są specyficzne dla kompilatora, lecz najprostsze rozwiązanie wygląda następująco:

```
class Singleton
{
  private:
    Singleton() {}
   ~Singleton() {}
  public:
    static Singleton &GetSingleton()
    {
      static Singleton s_Singleton;
      return s_Singleton;
    }
}

#define g_Singleton Singleton::GetSingleton()
```

Dzięki definicji g_Singleton mamy zapewniony prosty sposób odwoływania się do obiektu jak do globalnej zmiennej. Prywatny konstruktor i destruktor zabezpieczają nas przed przypadkowym stworzeniem więcej niż jednej instancji klasy Singleton.

Obiekt tworzony jest dopiero przy pierwszym użyciu, a nie przy starcie programu. Niestety nie mamy już kontroli nad niszczeniem obiektu, zostanie on usunięty dopiero podczas zamykania programu. Tworząc grę komputerową potrzebujemy większej kontroli nad cyklem życia wszystkich podsystemów, lecz przy tym rozwiązaniu nie jest to możliwe.

Rozwiązanie alternatywne

Jeśli zamiast zmiennej statycznej, wykorzystamy wskaźnik, możemy tworzyć i niszczyć instancję w dowolnym momencie:

```
class Singleton {
  private:
    static Singleton* ms_Singleton;
  public:
    Singleton() {
      assert( !ms_Singleton );
      ms_Singleton = this;
    }
   ~Singleton() {
      assert( ms_Singleton );
      ms_Singleton = 0;
    }
    static Singleton &GetSingleton() {
      assert( ms_Singleton );
      return *ms_Singleton;
    }
}
Singleton* Singleton::ms_Singleton = 0;

#define g_Singleton Singleton::GetSingleton()
```

W tej wersji musimy wyraźnie wskazać moment utworzenia i niszczenia obiektu, nim będziemy mogli się do niego odwołać.

Wykorzystanie szablonów

Powyższe rozwiązanie spełnia już wszystkie nasze wymagania. Niestety dla każdego typu mającego być singletonem, musimy powielać ten sam kod. Dzięki zastosowaniu szablonu i pewnej prostej sztuczki, możemy szybko definiować nowe typy singletonów:

```
template <typename T> class Singleton {
  private:
    static T* ms_Singleton;
  public:
    Singleton() {
      assert( !ms_Singleton );
      int offset = (int)(T*)1
                  -(int)(Singleton<T>*)(T*)1;
      ms_Singleton = (T*)((int)this + offset);
    }
   ~Singleton() {
      assert( ms_Singleton );
      ms_Singleton = 0;
    }
    static Singleton &GetSingleton() {
      assert( ms_Singleton );
      return *ms_Singleton;
    }
};
template <typename T> T* Singleton<T>::ms_Singleton = 0;
```

Aby dowolna klasa została singletonem wystarczy ją publicznie odziedziczyć z typu Singleton<NazwaKlasyPotomnej>. Należy pamiętać, by przed użyciem klasy, w dowolny sposób utworzyć jej instancję (zmienna globalna, statyczna lokalna, operator new)!

Konstruktor klasy Singleton oblicza przesunięcie między wskaźnikiem na siebie (this), a wskaźnikiem na klasę potomną (this wołane w klasie potomnej) i przypisuje odpowiednią wartość do prywatnej zmiennej ms_Singleton. Wskaźniki te mogą się różnić, jeśli klasa potomna dziedziczy z więcej niż jednej klasy.

Przykład użycia klasy:

```
class TextMgr : public Singleton <TextMgr> {
  public:
    Texture* GetTexture( const char* name );
    /* ... */
};

#define g_TextMgr TextMgr::GetSingleton()

void Example( void )
{
  Texture* stone = TextMgr::GetSingleton().GetTexture(„stone”);
  Texture* wood  = g_TextMgr.GetTexture(„wood”);
}

int main()
{
  new TextMgr();
  /* ... */
  return 0;
}
```

Zarządzanie zasobami w grach

Zasoby w grach

Gry komputerowe operują na wielu typach danych. Potrzebują szybkiego i wygodnego dostępu do tekstur, dźwięków, modeli, czcionek oraz innych zasobów. Różne elementy gry mogą potrzebować tego samego zasobu, np. kilka modeli może być pokrytych tą samą teksturą. Nie ma sensu, by każdy z nich tworzył kopię danych, gdyż było by to powolne i zużywało by zbyt dużo pamięci. W zapanowaniu nad danymi może nam pomóc menadżer zasobów, który będzie automatycznie wczytywał potrzebne dane i zwalniał je, gdy już nie będą wykorzystywane.

Wady wykorzystywania wskaźników

Odwoływaniem się do zasobów przechowywanych przez menadżera z wykorzystaniem wskaźników jest proste i samo nasuwa się na myśl. Jest to dogodna, lecz jednak niebezpieczna metoda. Jeśli któryś z podsystemów wyśle rządanie usunięcia danych, wskaźniki stosowane w innych

podsystemach staną się nieważne, a my dowiemy się o tym dopiero gdy program się załamie. Nie mamy możliwości zliczania ilości referencji do zasobu, gdyż można utworzyć wiele kopii wskaźnika, bez powiadamiania o tym menadżera.

Stosując wskaźniki, nie możemy zmieniać wewnętrznej organizacji danych. Jeśli dane zostaną przeniesione w inny obszar pamięci, to wszystkie wskaźniki stracą swą ważność. Problem ten najlepiej widać przy próbie zapisu stanu gry. Wskaźnika nie można zapisać, gdyż podczas ponownego wczytywania gry, rozkład danych w pamięci będzie zupełnie inny.

Zastosowanie uchwytów

Zamiast szukania metod, na zabezpieczenie wskaźników przed problemami opisanymi w poprzednim podrozdziale, możemy wprowadzić dodatkową warstwę abstrakcji i korzystać z uchwytów.

Uchwyty to sprawdzona technika programistyczna, wykorzystywana od dawna w interfejsach programistycznych. Przykładem uchwytów są identyfikatory plików zwracane podczas otwierania pliku i przekazywane do funkcji na nim operujących. Jeśli przekażemy do funkcji błędny bądź zamknięty uchwyt pliku, menadżer to wykryje i zwróci kod błędu, bez zawieszania programu.

Aby uchwyty były wydajne, powinny mieścić się w jednym rejestrze procesora. Ponadto muszą zapewniać prosty sposób kontroli poprawności. Uchwyty możemy zapisywać na dysk, wystarczy tylko, że podczas odczytu, odtworzymy dane w tej samej kolejności, w jakiej były podczas zapisu.

Prostą metodą implementacji uchwytu jest zastosowanie dwóch pól bitowych oraz liczby całkowitej bez znaku. Liczba całkowita będzie unikalnym identyfikatorem i służy do porównywania uchwytów. Po rozbiciu identyfikatora na dwa pola bitowe otrzymamy indeks i magiczną liczbę. Indeks służy do szybkiego pobrania danych z bazy menadżera uchwytu. Magiczna liczba umożliwia sprawdzenie poprawności uchwytu. Podczas dereferencji menadżer sprawdza czy zgadza się ona z liczbą przechowywaną w bazie.

```
class Handle {
  union {
    struct {
      unsigned m_Index : 16; // indeks tablicy
      unsigned m_Magic : 16; // magiczna liczba
    };
    unsigned int m_Handle;
  };
public:
  Handle () : m_Handle( 0 ) {}
  void Init( unsigned int index );
  /* ... */
};
```

Do pól uchwytu powinien mieć dostęp tylko menadżer. Dla reszty programu uchwyt jest czarną skrzynką.

Do operacji na klasie Handle wystarczą operatory porównywania oraz metoda inicjująca uchwyt. Podczas inicjowania, magiczna liczba ma przypisywaną kolejną wartość, przechowywaną w prywatnej statycznej zmiennej metody Init(). Magiczna liczba o wartości 0 jest zarezerwowana dla pustego uchwytu.

Zarządca uchwytów

Klasa zarządzająca danymi i ich uchwytami składa się z trzech wektorów: tablicy danych, tablicy magicznych liczb oraz listy wolnych miejsc. Tablica danych przechowuje obiekty typu DATA, podanego podczas deklarowania zmiennej korzystającej z szablonu HandleMgr. Wektor magicznych liczb jest wykorzystywany do sprawdzania poprawności uchwytów – przed pobraniem danych menadżer sprawdza, czy odpowiednia magiczna liczba jest równa tej z uchwytu. W tablicy wolnych miejsc przechowywane są indeksy wolnych miejsc w wektorach danych i magicznych liczb.

```
template <typename DATA> class HandleMgr {
  private:
    typedef std::vector <DATA>     DataVec;
    typedef std::vector <unsigned> MagicVec;
    typedef std::vector <unsigned> FreeVec;
    DataVec  m_Data;
    MagicVec m_Magic;
    FreeVec  m_FreeSlots;
  public:
    DATA         *Acquire( Handle& handle );
    void          Release( Handle handle );
    DATA         *Dereference( Handle handle );
};
```

Metody Acquire() i Release() służą do uzyskiwania i zwalniania uchwytów, zaś metoda Dereference() zwraca dane wskazywane przez uchwyt.

Przykład zastosowania

Klasa HandleMgr jest bardzo ogólna. Dla konkretnego zastosowania, należy ją opakować menadżerem konkretnego typu, na przykład menadżerem tekstur:

```
class TextMgr : public Singleton<TextMgr>  {
  private:
    class Texture { /* ... */ };
    HandleMgr<Texture> m_Textures;
    // name to handle std::map for searching
    NameIndex          m_NameIndex;

  public:
    Handle GetTexture    ( const char* name );
    void   DeleteTexture ( Handle htex );
```

```
    // Some operations on texture
    int     GetWidth      ( Handle htex ) const;
    int     GetHeight     ( Handle htex ) const;
    void    BindTexture   ( Handle htex ) const;
};
```

Menadżer definiuje typ danych, jakimi chcemy zarządzać (Texture) i udostępnia na zewnątrz interfejs umożliwiający pobieranie uchwytu na konkretny zasób (metoda GetTexture()). Uchwyt jest następnie przekazywany do metod operujących na zasobie, np. związujących teksturę w OpenGL, lub usuwających niepotrzebny zasób.

Zasób jest wczytywany tylko jeśli nikt jeszcze z niego nie korzysta. Jeśli już wcześniej został wczytany, menadżer znajdzie go w tablicy mapującej nazwę na uchwyt i zwraca już istniejący obiekt. W metodach GetTexture() i DeleteTexture() można zaimplementować zliczanie referencji i usuwać obiekt dopiero po zwolnieniu wszystkich referencji.

Literatura

„Perełki programowania gier – tom 1” pod redakcją Marka DeLoura – Helion 2002