Programowanie Gier Komputerowych

Wykład 10: System zapisu i odczytu stanu gry

mgr inż. Hubert Staniszewski

Po co grze system zapisu i odczytu?

- System save/load pozwala przerwać rozgrywkę i wrócić do niej bez utraty istotnego postępu.
- Dla gracza oznacza to wygodę, bezpieczeństwo i zaufanie do gry.
- Dla programisty jest to problem do zaprojektowania i rozwiązania: trzeba zdecydować, jaki stan gry jest ważny i jak go odtworzyć.
- Save/load nie dotyczy tylko pliku na dysku, ale całego modelu stanu gry.

Najważniejsze pytanie

Czy po wczytaniu gry świat wróci do stanu, który jest **spójny**, **przewidywalny** i **zgodny z oczekiwaniem gracza**?

Czym właściwie jest stan gry?

- **Stan gry** to zbiór informacji potrzebnych do kontynuowania rozgrywki od konkretnego momentu.
- Obejmuje nie tylko stan gracza, ale też np. stan świata czy ekwipunek.
- W praktyce zapisujemy nie „całą grę”, lecz tylko ten fragment stanu, który jest potrzebny do jej poprawnego odtworzenia.

Dane statyczne vs stan runtime w save/load

Dane statyczne

- definicje przedmiotów,
- konfiguracje przeciwników,
- opisy questów,
- data assets i tabele danych.

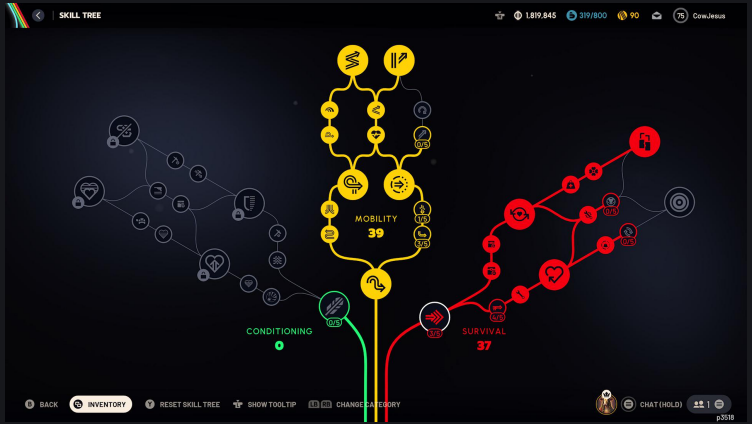
Stan runtime

- aktualne punkty życia,
- bieżące questy,
- pozycja gracza,
- zawartość ekwipunku,
- otwarte drzwi i aktywne przełączniki.

- Zwykle zapisujemy **stan runtime**, a dane statyczne odczytujemy ponownie z definicji gry.

Jakie elementy stanu gry trzeba zapisywać?

Zapisujemy to, co wpływa na dalszy przebieg rozgrywki i co można uznać za istotny postęp w takich elementach jak: stan gracza, stan świata, stan zadań, stan ekonomii.



Stan gracza

- Najczęściej zapisywane elementy stanu gracza to:
  - pozycja i orientacja,
  - aktualne HP, stamina, mana,
  - poziom doświadczenia,
  - wyposażone przedmioty,
  - aktywne buffy i debuffy.
- Warto rozróżniać to, co należy do **PlayerState**, od tego, co powinno być wyliczone ponownie po wczytaniu.

Stan świata

- Oprócz gracza trzeba zapisać także zmiany w świecie, które mają znaczenie dla dalszej gry.
- Przykładowe dane do zapisu to: zebrane przedmioty jednorazowe, otwarte drzwi i skrzynie czy odblokowane miejsca szybkiej podróży.

Stan questów i progresji

- Questy i progresja fabularna zwykle wymagają zapisu: aktywnych questów, etapów misji, flag fabularnych czy podjętych decyzji.
- To są dane, które wpływają na przyszłe zachowanie świata i innych systemów.
- Często nie zapisujemy całej treści questa, tylko jego stan i identyfikatory potrzebnych definicji.

Stan ekonomii i ekwipunku

- System save/load powinien uwzględniać także ekonomię gry czyli: walutę, stan ewkipunku i wyposażone przedmioty.
- Warto oddzielić definicję przedmiotu od konkretnej instancji.

Czego nie zapisywać bezpośrednio?

- Nie każdy element runtime powinien trafiać do zapisu wprost.
- Typowe przykłady danych, które lepiej wyliczyć ponownie:
  - cache i dane pochodne,
  - wskaźniki i referencje do obiektów runtime,
  - chwilowe stany debugowe,
  - dane, które można odtworzyć z definicji i prostszych pól.
- Im mniej zbędnych danych zapisujemy, tym mniejsze ryzyko niespójności po wczytaniu.

Typowy błąd

Zapisywanie stanu, który nie jest źródłem prawdy, tylko tymczasowym skutkiem działania systemu.

Właściciel danych w save/load

- Save/load powoduje problem właściciela danych: trzeba jasno wiedzieć, kto jest właścicielem której informacji.
- Jeżeli to same punkty życia istnieją w trzech miejscach, to po wczytaniu łatwo o konflikt.
- Najlepiej zapisywać dane u ich naturalnego właściciela:
  - HP w `HealthComponent`,
  - questy w `QuestManager`,
  - ekonomię w `EconomyState`.
- Potem system save/load zbiera te dane do wspólnego formatu zapisu.

SaveGame jako snapshot stanu

- W praktyce save file jest zwykle snapshotem stanu gry w danym momencie.
- Taki snapshot powinien być: spójny, jawny oraz możliwie prosty do odczytania przez system ładowania.
- Snapshot nie musi kopiować wszystkiego, ale musi zawierać to, co konieczne do poprawnego odtworzenia świata.

Przykład struktury SaveData

```
struct SaveData {
    PlayerSaveData player;
    WorldSaveData world;
    QuestSaveData quests;
    EconomySaveData economy;
    MetaProgressSaveData meta;
    int version;
};
```

- Taka struktura zbiera stan wielu systemów do jednego formatu zapisu.
- Ważne jest, aby każdy podsystem miał własny, czytelny fragment danych.

Identyfikacja obiektów w zapisie

- Żeby zapisać stan świata, trzeba rozróżniać konkretne obiekty.
- Najczęściej stosuje się identyfikacje obiektów przez instancjonowanie:
  - stabilne identyfikatory obiektów,
  - ID sceny + ID obiektu,
  - identyfikatory definicji oraz osobno identyfikatory instancji.
- Bez tego system nie będzie wiedział, które drzwi otworzyć, który przedmiot usunąć ani którego NPC oznaczyć jako pokonanego.

Referencje między obiektami

- W runtime obiekty często wskazują na siebie przez wskaźniki lub referencje.
- W zapisie zwykle nie zapisujemy samych wskaźników, tylko:
  - identyfikatory obiektów,
  - zależności logiczne,
  - klucze umożliwiające ponowne połączenie po wczytaniu.
- To oznacza, że po loadzie część relacji trzeba **zrekonstruować**, a nie tylko skopiować z pamięci.

Czym jest serializacja?

- **Serializacja** to zamiana danych z pamięci programu na format, który da się zapisać lub przesłać.
- **Deserializacja** to odtworzenie tych danych z zapisu do struktury użytecznej dla gry.
- W kontekście zapisu i odczytu serializacja musi być:
  - poprawna i przewidywalna,
  - odporna na zmiany wersji gry,
  - możliwa do debugowania.

Formaty zapisu stanu gry

Najczęstsze formaty

- binarny,
- JSON,
- XML,
- własny format tekstowy,
- format silnika lub biblioteki.

Kryteria wyboru

- szybkość,
- czytelność,
- rozmiar pliku,
- łatwość debugowania,
- kompatybilność między wersjami.

Porównanie formatów zapisu

Format	Zaleta	Wada	Typowe użycie
Binarny	szybki i mały	słabo czytelny	finalny save gry
JSON	łatwy do debugowania	większy rozmiar	narzędzia, prototypy
XML	czytelna struktura	dużo narzutu	starsze pipeline'y
Własny	pełna kontrola	wysoki koszt utrzymania	specyficzne potrzeby

Przykład serializacji prostych danych

```
1 struct PlayerSaveData {
2     int hp;
3     int stamina;
4     int gold;
5     float posX;
6     float posY;
7     float posZ;
8 };
```

- To jest prosty przypadek: zapisujemy liczby i pozycję.
- Taki model jest łatwy do serializacji, bo nie zawiera skomplikowanych zależności między obiektami.

Przykład serializacji zagnieżdżonych struktur

```cpp
struct ItemInstanceSaveData {
    std::string itemId;
    int upgradeLevel;
    int durability;
};

struct InventorySaveData {
    std::vector<ItemInstanceSaveData> items;
    int equippedSlot;
};
```

Gdy dane stają się zagnieżdżone, ważniejsza staje się spójność modelu niż sam format pliku.

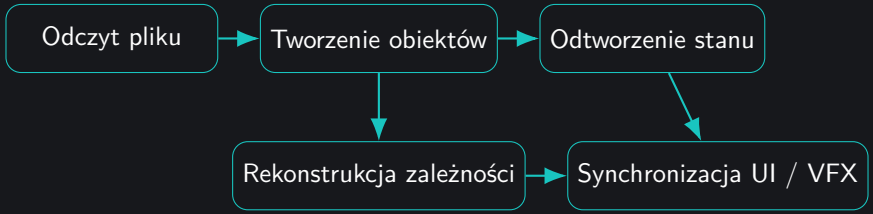
Problem wskaźników, referencji i assetów

- Wskaźniki i referencje działają tylko wewnątrz aktualnego procesu gry.
- Po restarcie programu nie mają sensu, więc nie można ich zapisać „wprost”.
- Podobnie assety i definicje danych zwykle nie są kopiowane do zapisu, tylko odwołujemy się do nich przez:
  - identyfikatory,
  - ścieżki,
  - klucze zasobów.
- To jest jeden z głównych powodów, dla których save/load wymaga osobnej architektury, a nie tylko „zrzutu pamięci do pliku”.

Jak przebiega ładowanie gry?

- Odczyt nie polega na prostym „wczytaj i graj”.
- Typowy proces wygląda tak:
  - odczyt pliku,
  - walidacja wersji i integralności,
  - utworzenie lub załadowanie potrzebnych obiektów,
  - odtworzenie danych runtime,
  - ponowne połączenie zależności,
  - synchronizacja UI i warstwy prezentacji.
- Kolejność tych etapów jest krytyczna.

Kolejność odtwarzania systemów ma znaczenie



- Jeżeli UI odtworzy się za wcześnie, pokaże niepełny stan. Jeżeli zależności odtworzą się za późno, systemy mogą pracować na niepoprawnych referencjach.

Tworzenie obiektów vs odtwarzanie danych

- W wielu grach ładowanie odbywa się w dwóch etapach:
  - najpierw tworzymy potrzebne obiekty runtime,
  - dopiero potem wypełniamy je danymi z sejwu.
- Dzięki temu można poprawnie rekonstruować relacje między obiektami.
- Ten podział jest szczególnie ważny przy:
  - NPC i przeciwnikach,
  - obiektach świata,
  - questach,
  - systemach zależnych od ID i referencji.

Load flow: przykład techniczny

```
void LoadGame(const SaveData& data) {
    SpawnWorldObjects(data.world);
    RestorePlayerState(data.player);
    RestoreQuestState(data.quests);
    RestoreEconomyState(data.economy);
    RebuildReferences(data.world, data.quests);
    RefreshUI();
}
```

- Najpierw pojawiają się obiekty, potem wracają ich dane, a na końcu odbudowujemy zależności i warstwę prezentacji.

Błędy przy odczycie stanu gry

- Typowe problemy podczas ładowania to:
  - brakujące obiekty,
  - niezgodne wersje danych,
  - referencje do czegoś, co już nie istnieje,
  - częściowo odtworzony stan świata,
  - duplikowanie lub znikanie instancji.
- W praktyce błąd save/load często nie objawia się od razu, lecz dopiero kilka minut później jako niespójność systemów.

Wersjonowanie save'ów

- Gra rozwija się w czasie, więc format sejwu też się zmienia.
- Dlatego save file powinien mieć numer wersji lub metadane formatu.
- Dzięki temu system ładowania może:
  - rozpoznać stary format,
  - wykonać migrację danych,
  - odrzucić niekompatybilny plik w kontrolowany sposób.
- Bez wersjonowania nawet mała zmiana w strukturze danych może zepsuć wszystkie stare sejwy.

Przykład wersjonowania danych

```
struct SaveHeader {
    int version;
    std::string buildId;
};

if (header.version < CURRENT_SAVE_VERSION) {
    RunMigration(header.version, data);
}
```

- To prosty mechanizm, ale bardzo ważny: ładowanie starych danych powinno być świadome, a nie przypadkowe.

Uszkodzone dane i częściowy zapis

- Save file może zostać uszkodzony przez błąd aplikacji, brak miejsca na dysku lub przerwanie zapisu.
- Dlatego warto stosować:
  - zapis do pliku tymczasowego,
  - dopiero potem podmianę właściwego pliku,
  - sumy kontrolne lub walidację,
  - czytelne komunikaty o błędzie ładowania.
- To nie tylko problem techniczny, ale także UX i zaufania gracza do gry.

Auto-save, checkpoint i manual save

Auto-save / checkpoint

- wygoda,
- bezpieczeństwo postępu,
- mniejsza kontrola gracza.

Manual save

- większa kontrola,
- większa odpowiedzialność po stronie gracza,
- ryzyko save scummingu.

- Wybór modelu zapisu wpływa nie tylko na implementację, ale też na projekt rozgrywki.

Walidacja, bezpieczeństwo i oszustwa gracza

- W grach single-player manipulacja sejwem jest zwykle problemem projektowym, nie bezpieczeństwa.
- W grach sieciowych lub współdzielonych dane zapisu wymagają większej kontroli.
- Typowe zabezpieczenia to:
  - walidacja zakresów danych,
  - sprawdzanie spójności,
  - rozdzielenie danych lokalnych i serwerowych,
  - podpisy lub sumy kontrolne.
- Najważniejsze jest jednak to, aby system ładowania nie ufał bezwarunkowo każdej wartości z pliku.

Dobry i zły przykład architektury zapisu/odczytu

Zły przykład

- każdy system zapisuje dane po swojemu,
- brak wspólnego formatu `SaveData`,
- brak wersjonowania,
- bezpośredni zapis wskaźników i stanów tymczasowych.

Dobry przykład

- jeden jawny model save data,
- każdy system eksportuje własny fragment stanu,
- load ma kontrolowaną kolejność,
- referencje odbudowuje się przez ID i migracje wersji.

Typowe błędy architektoniczne

- mieszanie danych statycznych i runtime,
- zapis stanu pochodnego zamiast źródła prawdy,
- brak stabilnych identyfikatorów obiektów,
- brak wersjonowania save'ów,
- odtwarzanie systemów w złej kolejności,
- rozproszona logika zapisu bez jednego modelu danych.

Wspólny skutek

System save/load działa tylko w prostym przypadku, ale psuje się podczas rozwoju projektu, migracji danych i bardziej złożonych stanów świata.

Podsumowanie

- Save/load to problem architektury stanu gry, a nie tylko zapis pliku.
- Kluczowe zagadnienia to:
  - runtime state vs static data,
  - właściciel danych,
  - serializacja,
  - identyfikatory i referencje,
  - kolejność odtwarzania systemów,
  - wersjonowanie.
- Najważniejsza zasada: zapis musi odtwarzać spójny model stanu gry, a nie przypadkowy zestaw zmiennych.