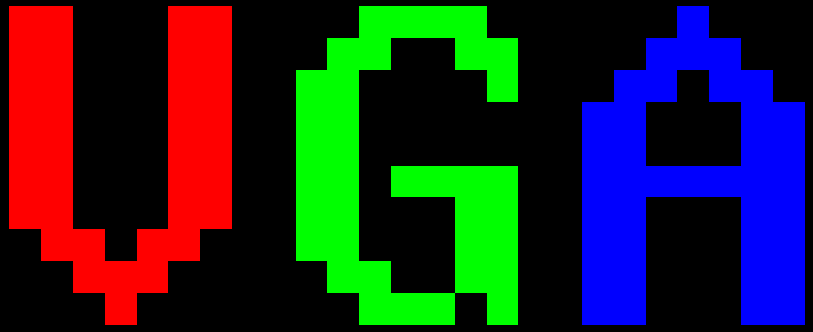


Herní assety a velké světy

Jan Matoušek

22. 10. 2026

Vymezení pojmu

A game asset is…

- **a building block of every video game, shaping what players see and hear.** – Michael Bierut
- **any resource used in video game development.** – multiple sources
- **a digital element used to create and enhance the gaming experience.** – My3dmeta

Lze najít spousty podobně nepoužitelných definic.

Vymezení pojmu

- Asset je každá část herních dat *kromě kódu hry, který nesouvisí přímo s logikou hry.*
- Tedy: Asset je každá část herních dat kromě kódu technického zázemí hry (enginu).
- Engine + assety = hra.
- *Assety jsou části herních dat vytvořené umělci.*
- Assety lze upravovat bez nutnosti mít u sebe hru nebo herní platformu.

Datová média

Kazeta (pro Commodore 64)	96 kB
3.5″ disketa	1.44 MB
CD	700 MB
DVD	4.5 - 8.9 GB
Digitální distribuce	∞

Velikosti assetů

1993	Doom	12 MB
1995	Rayman	50 MB
1998	Half-Life	400 MB
2005	GTA: San Andreas	4 GB
2011	Portal 2	8 GB
2017	theHunter: Call of the Wild	60 GB
2022	God of War: Ragnarok	175 GB
2024	Call of Duty Black Ops 6	220 GB

Používané formáty

- Obrázky
  - BMP, PCX, ICO, TGA, PNG, JPG, DDS…
- Zvuky
  - WAV, MP3, OGG, FLAC…
- Animace a videa
  - GIF, BIK, AVI, DivX, H.264, H.265…
- Hudba
  - MID, MOD, XM…
- 3D modely
  - OBJ, MDL, 3DS, FBX…
- Manuály
  - TXT, RTF, HLP, CHM, PDF…
- Levely, skripty, dialogy, konfigurace, scénáře, uložené hry, shadery, neuronové sítě…

Výběr formátu

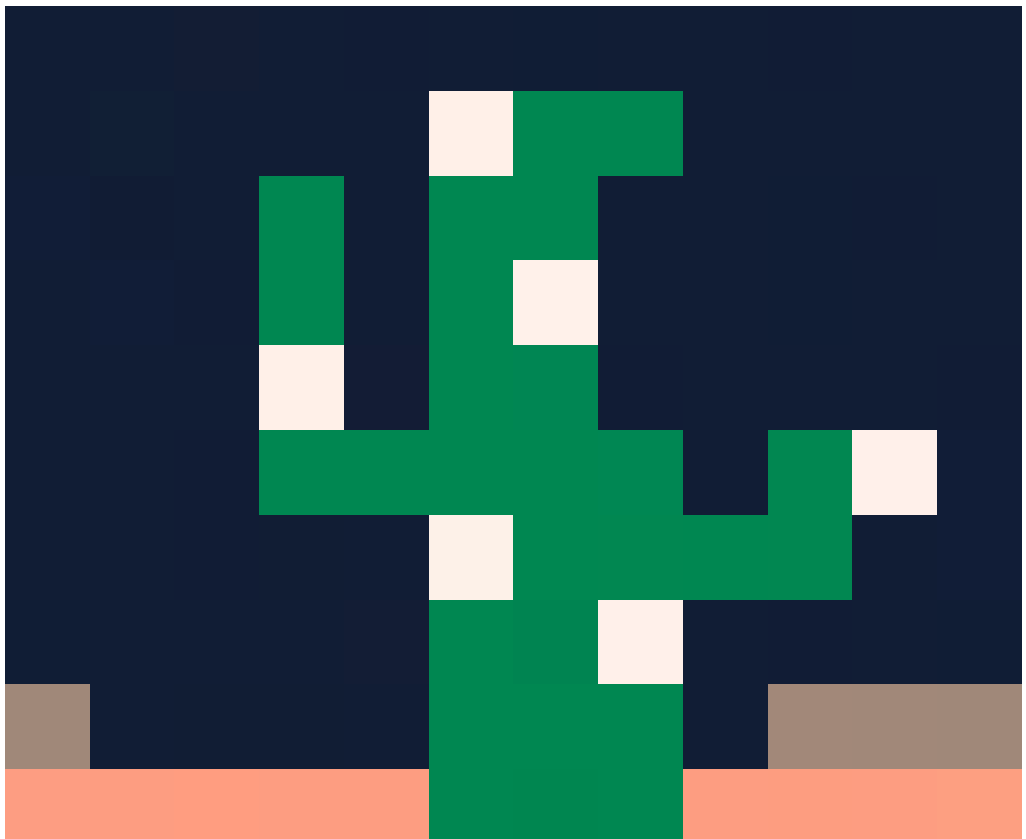
- Výběr vhodných formátů záleží na potřebách autorů hry a na cílové platformě
- Dilemata:
  - Existující formát versus vlastní
  - Proprietární versus svobodný
  - Binární versus textový
  - Druh komprese
- U audiovizuálních assetů obvykle volíme standardní formáty (existují na ně nástroje).
- U assetů specifických pro hru (např. prostředí, dialogy, uložené hry apod.) obvykle volíme vlastní formát.

Binární a textové formáty

- Binární formáty jsou blíže reprezentaci dat v paměti
  - Snadno se čtou programově
  - První volba při omezené paměti a výpočetních prostředcích
  - Nelze jednoduše vytvářet ručně, vyžadují speciální nástroje
- Textové formáty jsou vhodné pro člověka
  - Vhodné zejména pro věci, které nezpracovává specializovaný hardware
  - Předpokládá se úprava obyčejným textovým editorem
  - Zabírají více místa na disku

Příklad binárního formátu – bitmapa, 8 bitů na pixel

```
0C 0A
FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF 7D 18 18 FF FF FF FF
FF FF FF 18 FF 18 18 FF FF FF FF FF
FF FF FF 18 FF 18 7D FF FF FF FF FF
FF FF FF 7D FF 18 18 FF FF FF FF FF
FF FF FF 18 18 18 18 18 FF 18 7D FF
FF FF FF FF FF 7D 18 18 18 18 FF FF
FF FF FF FF FF 18 18 7D FF FF FF FF
0D FF FF FF FF 18 18 18 FF 0D 0D 0D
33 33 33 33 33 18 18 18 33 33 33 33
```



Příklad binárního formátu – level ve hře Doom

Soubor se skládá z bloků zvaných *lumps*, každý *lump* je pole jednoduchých struktur s určeným významem.

hlavička	název levelu a příp. další info
`THINGS`	entity v levelu
`LINEDEFS`	zdi
`SIDEDEFS`	stěny
`VERTEXES`	souřadnice vrcholů zdí
`SEGS`	segmenty zdí (vytvořené dělením do BSP)
`SSECTORS`	části sektorů (vytvořené dělením do BSP)
`NODES`	vrcholy BSP
`SECTORS`	sektory
`REJECT`	maska viditelnosti mezi sektory
`BLOCKMAP`	mřížka pro detekci kolizí entit

Příklad textového formátu – GLSL shader

```
#version 330

layout (location = 0) in vec3 vertPos;
uniform mat4 viewMatrix, projMatrix;
out vec3 color;

void main(){
  color = (vertPos + 10.0) / 20.0 ;
  gl_Position =  projMatrix * viewMatrix * vec4(vertPos, 1.0);
}
```

Komprese

- Potřeba dostat / přenést co nejvíce dat na omezená datová média / spojení
- Bezeztrátová komprese – rozbalená data jsou vždy identická s původními
  - Run-length encoding (RLE)
  - Dictionary encoding (ie. LZ77)
  - Huffman encoding
  - Deflate (LZ77 + Huffman)
- Ztrátová komprese – informace je upravena (zjednodušena), aby šla lépe komprimovat
  - Discrete wavelet transform
  - JPEG
  - DXT1 (a další)
  - MP3
- Výběr vhodného algoritmu záleží na povaze dat

Komprese: Run-length encoding

Sekvence stejných hodnot jsou zakódovány jako počet hodnot v sekvenci a hodnota.

Příklad:

a	b	b	b	b	b	a	a	a	a

↓

1	a	5	b	4	a

Komprese: Dictionary encoding

Opakující se sekvence jsou zapsány do slovníku a nahrazeny odkazem do slovníku.

Příklad:

a	b	c	a	b	c	d	e	d	e

↓

1	a,b,c
2	d,e

1	1	2	2

Komprese: Huffman encoding

Nejčastější údaje (bajty, sekvence) jsou nahrazeny nejkratšími sekvencemi bitů, méně časté mají delší sekvence bitů.

Příklad:

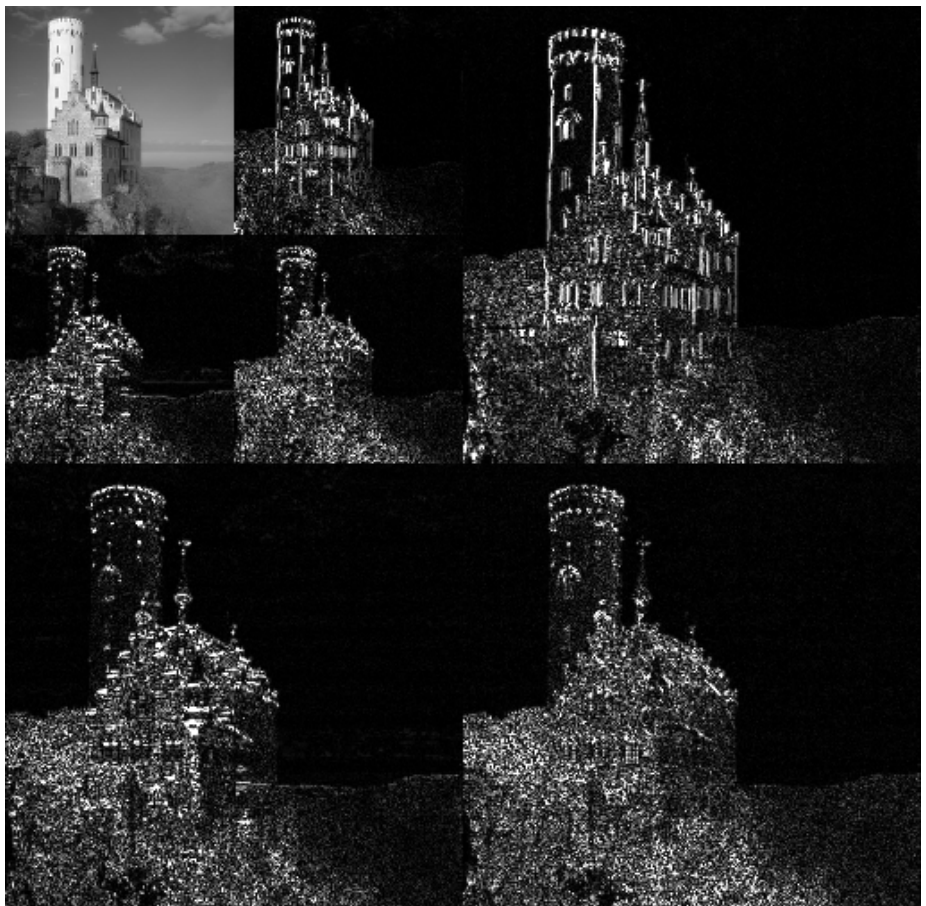
a	a	c	e	b	a	d	a	a	e

↓

a	5×	0
e	2×	10
c	1×	110
b	1×	1110
d	1×	1111

0	0	110	10	1110	0	1111	0	0	10

Komprese: JPEG



Komprese: Neuronové sítě

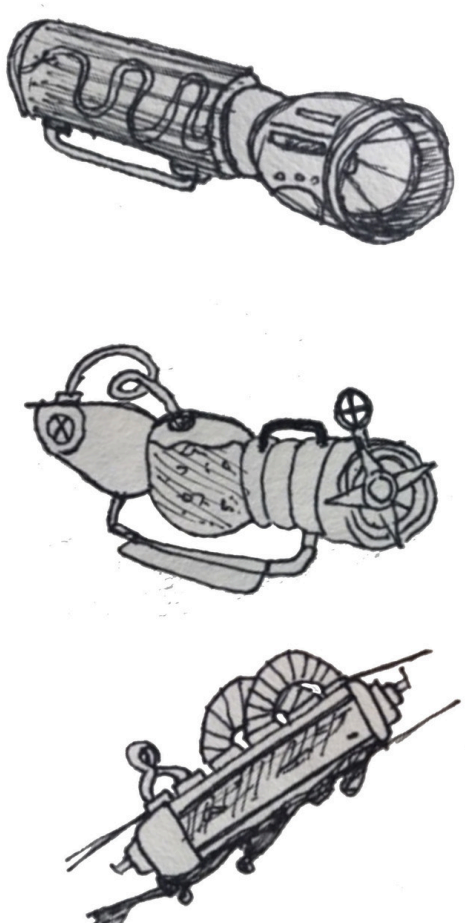
- Natrénujeme neuronovou síť na určité vzory v datech.
- Příklad použití: *Dune: Awakening*, ztrátová komprese výškové mapy terénu.



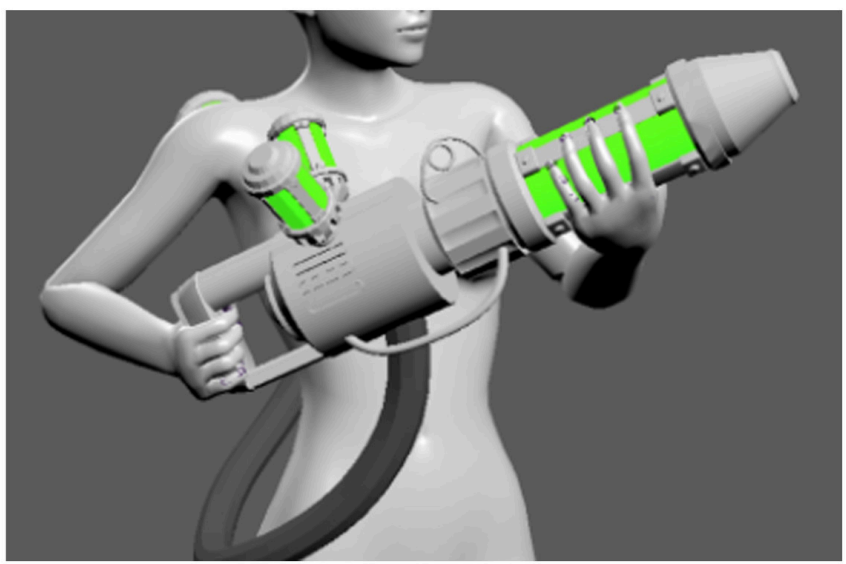
A. Martini, *Advanced Heightmap Compression Using Deep Learning in Dune: Awakening* (2023)

Asset management

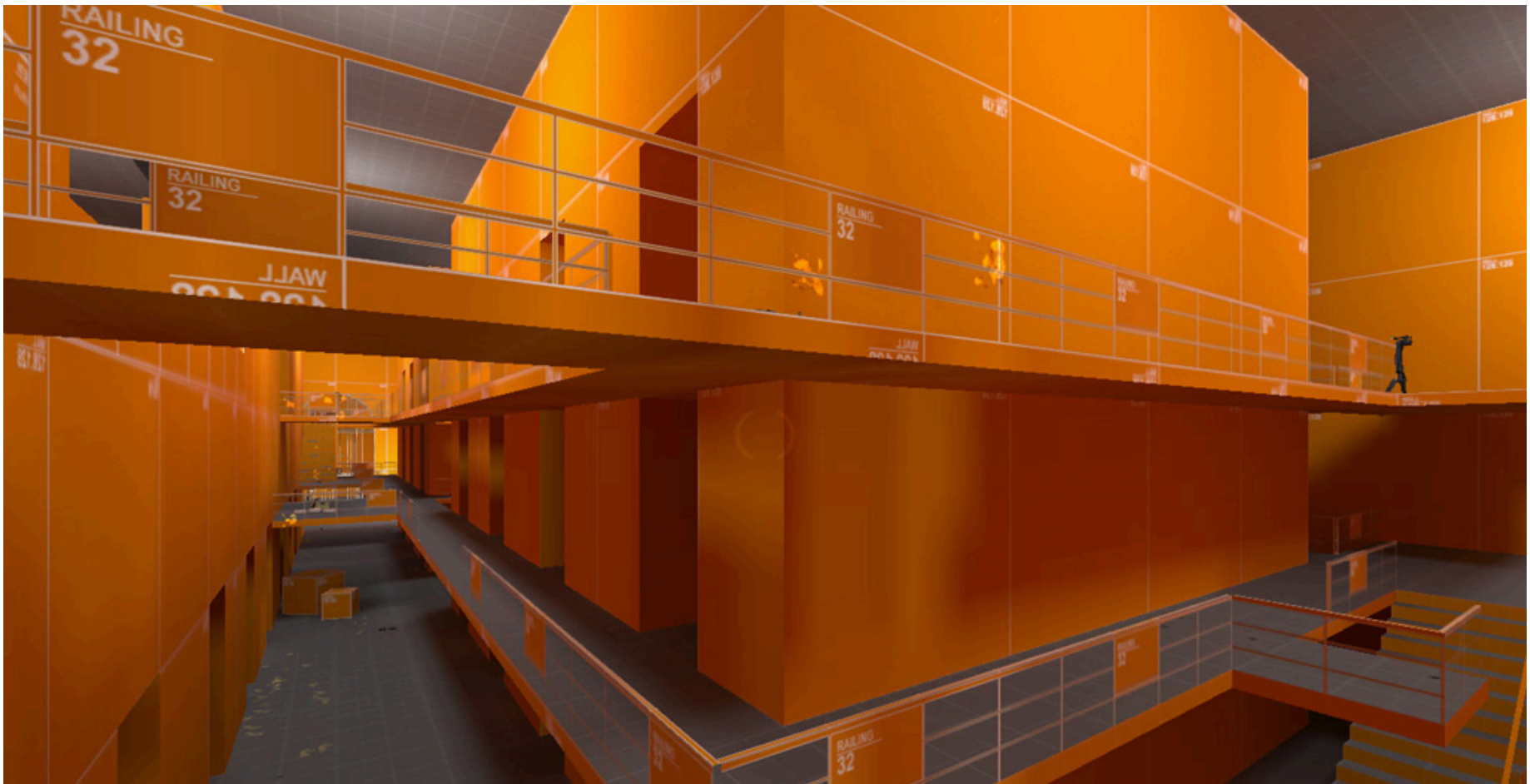
Vytváření assetů (ukázka, autor: Jenna Harding)



Weapon schets +3D model in Maya



Vytváření assetů (ukázka, Half-Life 2, mise Nova Prospekt, blok 4)



Vytváření assetů (ukázka, Half-Life 2, mise Nova Prospekt, blok 4)



Shromažďování assetů

- Verzovací systém, cloud, lokální file server…
- Ideální organizace:
  - Adresáře podle typů assetů (modely zvlášť, textury zvlášť atd.)
  - Další členění podle kapitoly příběhu hry, prostředí apod.

Integrace assetů

- *Asset pipeline* zajišťuje vložení assetů do hry
  - Obvykle sada skriptů, případně funkcionalita zvoleného enginu
  - Možné konverze formátů, zpracování a kompilace textových souborů apod.
  - Přenos do hry se zachováním vazeb mezi soubory (např. model a textury)
  - Reakce na změny v assetech
- Hladký chod je důležitý
  - Úspora času vývojářů, rychlá iterace assetů
  - Nefunkční integrace znemožňuje vývoj hry

Verzování assetů

- Verzovací systémy programátorů jsou vhodné primárně na textové soubory
  - Git, Mercurial, SVN…
- Na velké binární soubory se příliš nehodí.
- Často se v dnešní době využívají *Perforce* nebo *PlasticSCM*.

Násobné použití assetů

- Úspora místa i práce umělce
- Nutnost dobré koordinace využití assetů
  - Budoucí změna assetu může způsobit, že někde jeho použití nebude vhodné
- Štítkování – styl, autor, zda je hotový atd.

Serializace / deserializace

Motivace

- Řada assetů má využití pouze v daném engine/hře a používá vlastní formát.
- Musíme být schopní načítat data z assetu do struktur v paměti a naopak.
- Tomuto procesu se říká serializace / deserializace.

```
struct Something {
    int x, y;
    std::string name;
    void serialize(Stream& data) const;
    void deserialize(Stream& data);
}
```

Metody serialize a deserialize budou mít podobný vnitřek.

(De)serializace

```
struct Sth {
    int x, y;
    std::string name;
    template<typename X>
    friend void serialization<Sth>(Stream& data, Sth& var) {
        serialization(data, var.x);
        serialization(data, var.y);
    }
}
```

Různé datové formáty a verzování

- Podpora různých datových formátů je typicky vedlejší.
- Častější je, že se změní definice sruktury.
- Chceme zachovat kompatibilitu se staršími soubory.
- Do souboru typicky ukládáme interní verzi formátu.
- Pokud při čtení narazíme na starší formát:
  - Máme větve v serializaci podle verze

```
friend void ds<Sth>(Stream& data, Sth& var, int ver) {
    serialization(data, var.x, ver);
    serialization(data, var.y, ver);
    if (ver >= 5) { ... }
}
```

Snižuje čitelnost.

Různé datové formáty a verzování

- Pokud při čtení narazíme na starší formát:
  - Máme větve v serializaci podle verze
  - Máme pro každou verzi napsaný jiný deserializér
  - Máme funkce / nástroje pro převod verze

(De)serializace polymorfních objektů

- U polymorfních objektů si potřebujeme ukládat typ
- Podle načteného typu vytvoříme a deserializujeme správný objekt
- Hodí se využití vzoru Factory Function.

```
std::map<char type, std::function<Obj*()> fcs = {
    'T' => []() { return new ObjT(); }
};
```

Virtuální souborové systémy

Motivace

- Assetů může být velké množství (stovky, tisíce)
- Velké množství souborů se hůře kopíruje na disk
- Problemtická správa aktualizací a modifikací
  - Které soubory je třeba při instalaci přepsat?
  - Které soubory je třeba při odinstalaci modifikace vrátit do původního stavu?
- Soubory mohou být na různých místech souborového systému, podle platformy
  - Zejména uložené hry a jiné soubory vytvářené hraním
- Můžeme chtít ztížit úpravu herních souborů (šifrování, kontrola integrity)

Řešení

- Jednotné rozhraní pro čtení/zápis souborů
  - např. `vfs_read()`, `vfs_write()`, `vfs_fopen()`, `vfs_fclose()` apod.
- Převod virtuálních cest na skutečné cesty podle platformy
  - např. `save:file1` na `%APPDATA%\ThePublisher\TheGame\saves\file1`
- Sdružení souborů do archivů
  - např. soubor `hero_sprite.bmp` bude uvnitř `data.zip`
- Vrstvený přístup do archivů – hledám nejnovější soubor toho jména
  - např. `res:hero.fbx` se převede na `mod.zip/hero.fbx` místo `base.zip/hero.fbx`
- Pokud mám vlastní formáty archivů, musím vytvořit nástroje na jejich sestavení
- Během vývoje může být načítání assetů z balíčků vypnuté

Příklad použití – Godot engine

```
var PlayerNode:Node

var save_path := "user://player_data.ini"

func save() -> void:
  var config_file := ConfigFile.new()
  config_file.set_value("Player", "health", PlayerNode.health)
  config_file.save(save_path)

func load() -> void:
  var config_file := ConfigFile.new()
  var error := config_file.load(save_path)

  PlayerNode.health = config_file.get_value("Player", "health", 1)
```

Příklady archivů

- Doom: WAD
- Duke Nukem 3D: GRP
- Quake III Arena: ZIP (s koncovkou PK3)
- Godot Engine: PCK

Správa zdrojů

Správce zdrojů

- Služba herního enginu, jejíž zodpovědností je:
  - Načítat assety z disku do paměti dle potřeby a připravovat je k použití
  - Uvolňovat nepotřebné assety z paměti (a ze specializovaného hardware)
  - Řešit případné závislosti mezi assety
  - Deduplikovat stejné assety
  - Snažit se neplýtvat prostředky
  - Přitom co nejméně zdržovat hru

Načítání/uvolňování závislostí a deduplikace assetů

- Každý resource má unikátní identifikátor (např. cesta a název souboru, jejich hash…)
- Grafy závislostí (BFS, DFS atd.)
- Počítání referencí, garbage collection

Načítání bez zdržování hry

- Prokládané načítání (vždy část dat během snímku)
- Asynchronní načítání
  - `async`/`await`, korutiny…
- Načítání v jiném vlákně
  - Nemá smysl mít více než jedno vlákno pracující s diskem
  - Dekomprese a příprava dat však může být ve více vláknech
  - Komunikace se specializovaným HW bývá jednovláknová
- Načítání s předstihem
  - Podle polohy a směru pohybu hráče
  - Nebo na předem určených místech

Příklad asynchronního načítání – Godot engine

```
const ENEMY_SCENE_PATH : String = "Enemy.tscn"

# Loading screen button code
func _ready():
  # Start request
  ResourceLoader.load_threaded_request(ENEMY_SCENE_PATH)
  self.pressed.connect(_on_button_pressed)

func _on_button_pressed(): # Button was pressed.
  # Obtain the resource now that we need it.
  var enemy_scene = ResourceLoader.load_threaded_get(ENEMY_SCENE_PATH)
  # Instantiate the enemy scene and add it to the current scene.
  var enemy = enemy_scene.instantiate()
  add_child(enemy)
```

Flyweight

Kontext

```
struct TerrainField {
    double walkingSpeed;
    double painPerSecond;
    bool requiresSwimming;
    Texture tileTexture;
    // ...
};

struct Map {
    TerrainField data[4096][4096];
};

// sizeof(TerrainField) == 100 B
// map: 4096x4096x100 ... 1.6 GB
```

```
struct TerrainFieldType {
    double walkingSpeed;
    double painPerSecond;
    bool requiresSwimming;
    Texture tileTexture;
    //...
};

struct Map {
    TerrainFieldType grass, water, rock, ...;
    TerrainFieldType* data[4096][4096];
};

// map: 4096x4096x8 ... 128 MB
```

Poznámky

- Dereference pointeru navíc versus úspora paměti.
- Lze kombinovat pointer na typ entity se stavy entity.
- Lze kombinovat více typů pomocí více pointerů:
  - zde např. typ povrchu, typ budovy na povrchu, druh podloží.
- Lze převést do virtuálních metod, ale opatrně:
  - pozor na dynamic dispatch v horké smyčce!
- Lze seskupit části scény podle shodných pointerů na typ a zpracovávat scénu v pořadí podle typů (vhodné zejména při vykreslování)
  - přirozeně vede na instancing na GPU
  - lze ušetřit častý dynamic dispatch

Dělení prostoru

Motivace

Problém: Mám svět obrovský, že jeho velikost paměť přesahuje.

- Vykreslit / simulovat celý svět naráz nedává smysl.
- Rozdělíme svět na části, různými způsoby.

Problém: Mám v kobce milion démonů, a všichni se chtějí vzájemně mlátit.

- Předpokládáme, že každý démon chce mlátit nejbližší démony.
- Řešením je rozdělit si démony do přihrádek, typicky podle polohy.

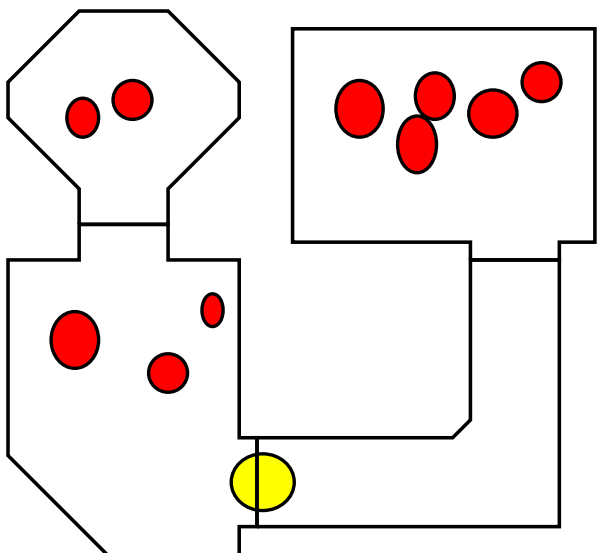
Dělení světa v designu hry – levely



Half Life, 1998 – Teleport z jednoho levelu do druhého.

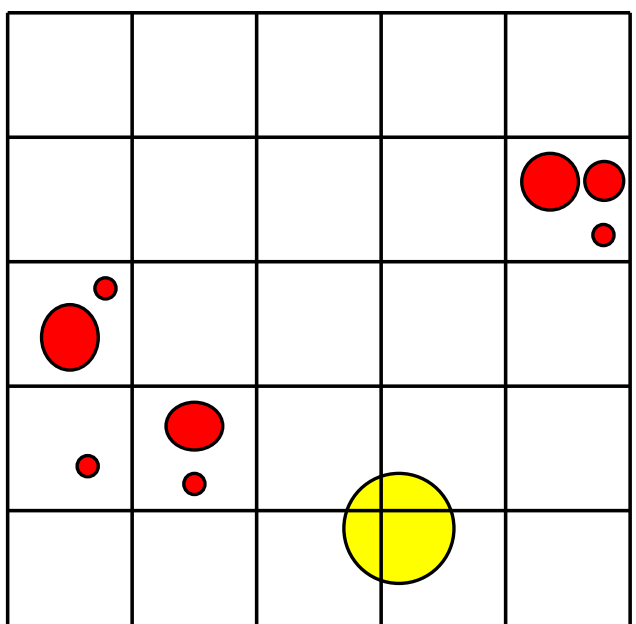
Dělení světa v designu hry – sektory

- Svět je sestaven ze sektorů (místnosti, úseky tratě)



Pravidelná mřížka

- Dělení pomocí pravidelné mřížky (čtvercová, krychlová, hex…)
- Rychlé sestavení i aktualizace, ale nevhodné pro různé škály detailů.



Spatial hashing

- Některé prvky mřížky bývají plnější než jiné.
- Mohli bychom zavést hashování podle souřadnic prvku mřížky.
- Spoléháme na kolize, musíme s nimi umět počítat.

```
struct Cell { std::vector<Entity> entities; }
std::unordered_map<std::pair<int,int>, Cell> map;
```

Nutná manipulace s hash mapou při změně polohy.

Hierarchická dělení (stromy)

- Pravidelné stromy - quadtree (2D), octree (3D)
- Binary space partitioning tree (BSP)
  - V každém vrcholu dělí prostor podle vhodné řezné roviny
  - Pomalejší sestavení i aktualizace, obvykle jsou předpočítané

