DATA IS LIFE
You think games are this

GAME
In reality, games are this

GAME DATA

PRODUCTION / TRACKING

IMPORT PIPELINES

ENGINE

EDITORS

TOOLS

ON-LINE SERVICES

TESTING / VALIDATION

in other words, the game itself is a small part of the eco-system
Fundamental Truth #1

A Game engine is BOTH

- runtime engine
- ecosystem of tools and asset management systems that surround *content creation*

→ The primary function of any game engine is to allow content creators to shovel content into the engine as fast possible.
This Lecture

- Part 1: The Big Picture
  - give an appreciation for the multiplicity of problems surrounding data
  - Hard (in this context) because ....?

- Part 2: Practical Issues
  - Loading and saving datafiles
  - Schemas
  - Versioning

- Part 3: More Big Ideas
  - Data Inheritance
  - Bundling
  - Game Data vs. Game State
PART 1: THE BIG PICTURE
in other words, the game itself is a small part of the eco-system
Zooming In

GAME DATA

"PIPESINES"

ENGINE
EDITORS
TOOLS
TESTING / VALIDATION
ON-LINE SERVICES
PRODUCTION / TRACKING
The Pipeline

- What is it? Why does it exist?
  - One representation is good for editing
  - Another is good for runtime
  - Rarely the same
Typical Pipeline

- Source
- Intermediate
- Engine
- Optimized
- In-Memory
Typical Pipeline (Sputnik)

- .PSD
- .TIF
- .SDF
- PVRTC

**Source**

**Intermediate**

**Engine**

**Optimized**

**In-Memory**
Typical Pipeline (Sputnik)

- PSD
- TIF
- SDF
- PVRTC

Source

Intermediate

Engine

Optimized

In-Memory
Typical Pipeline (Sputnik)

- PSD
- TIF
- SDF
  
  SOURCE
  
  INTERMEDIATE
  
  ENGINE
  
  OPTIMIZED
  
  IN-MEMORY
  
  PVRT

  SOURCE
  
  OPTIMIZED
  
  IN-MEMORY
  
  AI

  DXT
Typical Pipeline (Sputnik)

- PSD
- SOURCE
- Engine
- Optimize
- In-memory
- .psd
- .tif
- .ai
- PVRTC
- Optimized
- IN-MEMORY
- DXT
- IN-MEMORY
Example: Bitmaps

- In general: the longer the pipeline, the longer the iteration time

- Great tech-artists and engineers can shorten this pipeline, and therefore make iteration easier
  - A lot of coding right here!
ASSIGNMENT #2!

PART 2: “PRACTICAL” ISSUES
Getting Practical

Consider “Engine” files

What do we want to be able to do with engine-format datafiles?

- Serialization
  - Load
  - Save
- Edit them
- Analytics
Saving and Loading Data

`s_player_config *config;

Ultimately, data needs to turn into in-memory structures
What formats do we load?

Standard ones: .png, .wav, .fbx, etc.

- These could be the engine-formats for standard types of content
  - e.g. Sputnik loads tiffs

- Most standard formats also have open-source libraries for i/o
  - tifflib, jpeglib, pnglib, etc.

Question
Why not just integrate as many as possible?
What formats do we load?

- TONS of non-standard ones:
  - level files
  - configuration files
  - player profiles
  - AI behavior params
  - physics parameters
  - etc.

- Need some way of storing arbitrary structures
  - XML
  - JSON
  - SDF (sputnik)
XML

- Pros
  - human readable
  - lots of 3\textsuperscript{rd}-party editors

- Cons
  - SUPER inefficient to save/load
  - SUPER inefficient to store
  - 3\textsuperscript{rd}-party editors are never good enough

Compare to binary format
  - blit it in and out
  - a bajillion times faster than parsing an xml file
So...

Why not just do this?

// save an array of vectors to a file
fwrite(file, &num, sizeof(int)); // write the count
fwrite(file, varray, sizeof(s_vector)*num); // write the data

// load from a file
int num = *(int*)fread(file, sizeof(int)); // read the count
varray= (s_vector *)fread(file, size(s_vector)*num); // read the data
Because of this

**Day 1:**

```c
struct s_vector {
    float x, y;
};
```

**Day 2:**

```c
struct s_vector {
    // added the "z" field
    float x, y, z;
};
```

Save 10,000 vector files

Load old files

AAAAARGH!!!
What happened?

<table>
<thead>
<tr>
<th>X</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>Y</td>
<td>3</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
</tr>
<tr>
<td>Y</td>
<td>5</td>
</tr>
<tr>
<td>X</td>
<td>6</td>
</tr>
<tr>
<td>Y</td>
<td>7</td>
</tr>
<tr>
<td>X</td>
<td>8</td>
</tr>
<tr>
<td>Y</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>3</td>
</tr>
<tr>
<td>Y</td>
<td>4</td>
</tr>
<tr>
<td>Z</td>
<td>5</td>
</tr>
<tr>
<td>X</td>
<td>6</td>
</tr>
<tr>
<td>Y</td>
<td>7</td>
</tr>
<tr>
<td>Z</td>
<td>8</td>
</tr>
<tr>
<td>X</td>
<td>9</td>
</tr>
</tbody>
</table>

What we probably wanted

<table>
<thead>
<tr>
<th>X</th>
<th>o</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>Y</td>
<td>3</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>4</td>
</tr>
<tr>
<td>Y</td>
<td>5</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>6</td>
</tr>
<tr>
<td>Y</td>
<td>7</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
</tr>
<tr>
<td>X</td>
<td>8</td>
</tr>
<tr>
<td>Y</td>
<td>9</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
</tr>
</tbody>
</table>
The Versioning Problem

This (code changing definition of saved structures while files containing those structures are still “out there”) is the rule not the exception.
2 Methods for Save/Load

- Explicit Handlers
- Reflection-based
Explicit Save/Load Handlers

class s_vector :
    public s_base, public serializable
{
    s_vector();
    float32 x, y;

    float normalize();

    bool read_in(istream *in) { // do stuff }
    bool write_out(ostream *out) { // do stuff }
}

Pros:
    - very explicit

Cons:
    - Separate code for every single structure/class type that needs to be serialized. Versioning needs to be handled in every class.
Reflection-Based Methods

Schemas

- Code-readable class/structure description

```
“s_vector”, type “struct”

Field: “x”, type “float”
Field: “y”, type “float”
Field: “normalize”, type “method”
```

(Note! Non-standard terminology)
Reflection-Based Methods

- Answer questions like
  - how big is structure A?
  - what is the name and type of the 3rd field of A?
  - what is the offset of the 3rd field into A?

- Allows you to write generic tools code for ANY structure
  - e.g., data editors, script interfaces

- Languages like Java, C# have this for free
Making Schemas: Some Options

- In addition to a struct/class definition, make some meta-data representation of the schema by hand (could be a datafile, a macro structure, or a piece of code).

- Define struct/class in neutral 3rd-party language, generate both C++ and runtime schema structure from it.
  - Unreal Engine

- Parse the header files in a pre-compile step
  - Sputnik
Example from Sputnik

From math.h

SPUTNIK_STRUCT(s_bounding_box2d)
{
    SPUTNIK_HEADER(s_bounding_box2d);

    VAR(rw) float32 minx;
    VAR(rw) float32 miny;
    VAR(rw) float32 maxx;
    VAR(rw) float32 maxy;

    METHOD bool valid() const;
    METHOD void clear();
    METHOD bool contains_point(float32 x, float32 y) const;
};
Example from Sputnik

From math.h

SPUTNIK_STRUCT(s_bounding_box2d)
{
    SPUTNIK_HEADER(s_bounding_box2d);

    VAR(rw) float32 minx;
    VAR(rw) float32 miny;
    VAR(rw) float32 maxx;
    VAR(rw) float32 maxy;

    METHOD bool valid() const;
    METHOD void clear();
    METHOD bool contains_point(float32 x, float32 y) const;
};

note, from schema_macros.h:
#define SPUTNIK_STRUCT(name) struct name
#define VAR(permissions) 
#define METHOD
Example from Sputnik

```cpp
s_bounding_box2d::x_s_bounding_box2d_schema_handle= schema_new("s_bounding_box2d", "");
if (s_bounding_box2d::x_s_bounding_box2d_schema_handle.valid())
{
    s_schema *schema= s_bounding_box2d::x_s_bounding_box2d_schema_handle.get();
    schema->add_primitive_field("minx", _primitive_type_float32, _read_write, 1);
    schema->add_primitive_field("miny", _primitive_type_float32, _read_write, 1);
    schema->add_primitive_field("maxx", _primitive_type_float32, _read_write, 1);
    schema->add_primitive_field("maxy", _primitive_type_float32, _read_write, 1);
    schema->add_method_field("valid", g_null_schema_handle, s_bounding_box2d_valid_return_schema_handle, s_bounding_box2d_valid_callback);
    schema->add_method_field("clear", g_null_schema_handle, g_null_schema_handle, s_bounding_box2d_clear_callback);
    schema->add_method_field("contains_point", s_bounding_box2d_contains_point_args_schema_handle, s_bounding_box2d_contains_point_return_schema_handle, s_bounding_box2d_contains_point_callback);
}
```

Precompile step

- .h
- python parser
- .inl
- #include in
- math.cpp
- .cpp
Schemas are Useful for All Kinds of Things

- Generic editors
- Scripting interfaces
- Dependency Analysis
- Storage format agnosticism
The $280M question

How does this help us solve the versioning problem?
SDF Format

vector_file.sdf

Schema: “s_vector”:
  float x
  float y

chunk header
type: s_vector
count: 10000

raw vector data

Current s_vector schema:

```c
struct s_vector {
    float x;
    float y;
};
```

Definitions match?
Blast it in!
SDF Format

vector_file.sdf

Schema: "s_vector":
  float x
  float y

chunk header
  type: s_vector
  count: 10000

raw vector data

Current s_vector schema:

struct s_vector
{
  float x;
  float y;
  float z;
};

Definitions match?
Blast it in!

Definitions don’t match?
Perform versioning!
When does versioning happen?

- Overnight when everyone’s at home in bed?
  - For some particularly complicated version jobs, it needs to be done this way.

- At load time
  - when we load a file, the loader makes sure that the data is in a format that it can use
  - Note: does this mean OLD-format code can read NEW-format data?
  - In principle, yes. Dangerous though.
2 forms of Data Versioning

1) Simple
   - field re-ordering
     - match by name and type
   - fields removed
   - fields added
     - where do we get the new data from?
       - default value based on field type (e.g. float → 0.0f)
       - exemplar
2 forms of Data Versioning

2) Complex

- write explicit code handlers to take us from one version to another

- In Sputnik, comes in 2 flavors:
  - “minor” (v1.2 to v1.3)
  - “major” (v1 to v2)

- Can do absolutely anything.
  - restructuring internals of the class
  - separating pieces of data into new datafile
  - etc.
Versioning c_model

v0

```cpp
SPUTNIK_CLASS(c_model_v0)
{
public:
    SPUTNIK_HEADER(c_model_v0);
    SPUTNIK_VERSION_MINOR(1, c_model_0_0_to_0_1);

    void initialize() {}
    void erase() {}

    VAR(r) s_import_data m_import_data;

    STORAGE(rw, s_part) m_parts;
    STORAGE(rw, c_geometry) m_geometry;
    STORAGE(rw, c_geometry) m_physics_geometry;
    STORAGE(rw, s_flash_geometry) m_flash_geometry;
    STORAGE(rw, s_marker) m_markers;

    VAR(rw) c_skeleton m_skeleton;
}
```
Versioning c_model

v0

```cpp
SPUTNIK_CLASS(c_model_v0)
{
    public:
        SPUTNIK_HEADER(c_model_v0);
        SPUTNIK_VERSION_MINOR(1, c_model_0_0_to_0_1);

        void initialize() {}
        void erase() {}

        VAR(r) s_import_data m_import_data;

        STORAGE(rw, s_part) m_parts;
        STORAGE(rw, c_geometry) m_geometry;
        STORAGE(rw, c_geometry) m_physics;
        STORAGE(rw, s_flash_geometry) m_flash_geometry;
        STORAGE(rw, s_marker) m_markers;

        VAR(rw) c_skeleton m_skeleton;
}
```

v2

```cpp
SPUTNIK_CLASS(c_model_v0)
{
    public:
        SPUTNIK_HEADER(c_model_v0);
        SPUTNIK_VERSION_MINOR(1, c_model_0_0_to_0_1);

        void initialize() {}
        void erase() {}

        VAR(r) s_import_data m_import_data;

        STORAGE(rw, s_part) m_parts;
        STORAGE(rw, c_geometry) m_geometry;
        STORAGE(rw, c_geometry) m_physics;
        STORAGE(rw, s_flash_geometry) m_flash_geometry;
        STORAGE(rw, s_marker) m_markers;
        STORAGE(rw, s_line2d) m_lines;
        STORAGE(rw, s_shadow_properties) m_shadow_properties;
        STORAGE(rw, s_bitmap_properties) m_bitmap_properties;

        HANDLE_EXTERNAL(rw, c_skeleton, NONE, true) m_skeleton;
    
    public:
        SPUTNIK_HEADER(c_model_v0);
        SPUTNIK_VERSION_MAJOR(2, c_model_v1, c_model_1_0_to_2_0);
        POSTPROCESS(model_postprocess);

        void initialize();
        void erase();

        STORAGE(rw, s_part) m_parts;
        STORAGE(rw, c_geometry) m_geometry;
        STORAGE(rw, c_geometry) m_physics;
        STORAGE(rw, s_flash_geometry) m_flash_geometry;
        STORAGE(rw, s_marker) m_markers;
        STORAGE(rw, s_line2d) m_lines;
        STORAGE(rw, s_shadow_properties) m_shadow_properties;
        STORAGE(rw, s_bitmap_properties) m_bitmap_properties;
```
PART 3: MORE BIG IDEAS
3 More Big Ideas (tm)

- Data Inheritance
- Bundling
- Data vs. Gamestate
Data Inheritance

Problem!

- It takes 1000s of parameters to define
  - NPCs
  - Weapons
  - Vehicles
  - etc.

- Similar objects share 99% of the parameters
  - Say I have 200 potions
  - Change “friction coefficient” from 0.2 to 0.3 on all of them?
Halo 2 Data Inheritance

- **Generic Character**
  - Grunt
    - Grunt Major
    - Kamikaze
  - Elite
    - Elite Major
      - Jetpack Elite
      - Elite Ultra
Unreal Engine: Archetypes

**Generic Character**
- Mass = 50
- Max Speed = 10
- Hitpoints = 100

**Grunt**
- Max Speed = 5
- Hitpoints = 50

**Elite**
- Max Speed = 120

**Elite Major**
- Hitpoints = 150
Unreal Engine: Archetypes

**Generic Character**
- mass = 50
- max_speed = 10
- hitpoints = 100

**Grunt**
- max_speed = 5
- hitpoints = 50

**Elite**
- max_speed = 120

**Elite Major**
- mass = 50
- max_speed = 120
- hitpoints = 150
Bundling

Problem!

- A single asset is made up of a LOT of individual files.

OR

- I want to create a whole bunch of objects in my level, but don’t want to create a separate file (and name them all) for each one.
Solution

- Allow files to be bundled together
  - “sniper_rifle” and “sniper_rifle_damage”
  - “grunt” and “grunt_perception”
  - etc.

- Extend the filesystem

Why not bundle entire levels?
In fact, sometimes we do.
Game Data vs. Game State

```c
struct s_character {
    float mass;
    float max_speed;
    s_vector velocity;
    s_vector position;
    array<s_vector> vertices;
    array<int> indices;
}
```

**Burnt to Disk**

**Saved in Game Save**
Game Data vs. Game State

```c
struct s_character_def {
    float mass;
    float max_speed;
    s_vector position;
    array<s_vector> vertices;
    array<int> indices;
}
```

```c
struct s_character {
    s_character_def *definition;
    s_vector velocity;
    s_vector position;
}
```
Zooming In

GAME DATA

ENGINE
EDITORS
TOOLS
TESTING / VALIDATION
ON-LINE SERVICES
PRODUCTION / TRACKING
Game Data vs. Game State

Move to an engine as follows:

1. In-memory data
2. Game state

The game state is saved as a saved game.
Thanks!