Persistence

Patterns for object creation

```
class Thing
{
public:

    static Thing* create(const char* aFile);
    ~Thing();

private:

    Thing();
    bool init(const char* aFile);
}
```

Patterns for object creation

```
Thing* Thing::create(const char* aFile)
{
    Thing* t(new Thing);

    if(t && !t->init(aFile))
    {
        delete t;
        t = NULL;
    }

    return t;
}
```

Patterns for object creation

- Object existence coupled to successful asset load

Patterns for object creation

- Object existence coupled to successful asset load
- Good:
  - object can't exist if not properly initialized with valid asset!

Patterns for object creation

- Object existence coupled to successful asset load
- Good:
  - object can't exist if not properly initialized with valid asset!
- Bad:
  - object can't exist if not properly initialized with valid asset!

Patterns for object creation

- Object existence coupled to successful asset load
- Good:
  - object can't exist if not properly initialized with valid asset!
- Bad:
  - object can't exist if not properly initialized with valid asset!
  - makes system data centric
    - i.e. shifts focus from code/functionality to sources of data

Data sources

- When creating games, the need to edit data is everywhere!
- What kinds of sources can we use?
  - Text files
    - Human readable
    - Many good tools available for view / edit
    - Can be hard to parse / error prone
  - Binary files
    - Not human readable
    - Requires tools to view and edit
    - More computer friendly / optimized
  - Databases
    - Standard format
    - Requires tools to view and edit
- Primary issue is often ease of edit

How is data authored?

- Games often use a mix of standard formats and custom formats
- Off the shelf tools
  - Photoshop
    - .psd for originals
    - .tga, .dds for export
  - Maya
    - .mb for originals
    - plugins / custom export?
  - Text editors
    - all other stuff, i.e. custom text formats

“We don't want to waste time building tools, and besides; Guis are hard!”

Tool selection

- Some existing tools are very good for games
  - Photoshop, Maya
  - Let us export exactly the formats we want
- Text editors are also good
  - Let us edit text
- But is text what our games want?
  - Often we do major transformations at load time
  - Text files are bad at enforcing constraints
  - Lots of error checking code

Juice / JuiceMaker

- Massive Entertainment “Juice” data language
  - One size fits all?
- JuiceMaker – one tool to rule them all
  - Extensible with custom editors
- Ice exports
  - Optimized, read-only data format for game
- Asset tracking
  - Juice gave us a way to recursively track all assets
  - Game depends on directory.ice, lists all other assets
- But still bad...
  - Still lots of load time validation and transformation of data
  - Very fragile, code is transparently dependent on .fruit headers
    - Breaks at run-time, not compile time
  - We didn't keep .fruit in version control...

A better way

- What if it was easy to build even more custom tools?
  - Fewer data errors, i.e. not possible to enter bad data
  - Easier to manage game-related constraints
    - i.e. can only select certain values for a certain field
- IMGUI to the rescue
  - Gui's are not hard!
  - Can make several custom Controllers for specific editing needs
  - Build these Controllers/editors into the game application
    - MUCH faster edit-play-repeat cycle
- But how do we persist changes?

Custom tool persistence

- When data is authored in the game...
  - The in-memory format is the ONLY thing we care about
  - We really don't care how things are persisted...
  - But they must be persisted...

- What if data was just magically persistent:
  - i.e. values of variables remain across executions
  - Implies some kind of robust instance identification scheme

???

Instance identification

- Borrow from relational theory
  - Table = Class
  - Row = Instance
- In a C++ program
  - objects have the same class every execution
  - objects DO NOT have the same address every execution
  - but, they are logically the same objects as they were last time
- It turns out we can use:
  - Class name as string
  - Unsigned int as instance id
    - This can be automatically generated

Persistent baseclass

- Use a template baseclass which gives us:
  - Static class name as string
  - Static linked list of all instances
    - This allows the baseclass constructor to assign keys

```
template <class X>
class Persistent
{
public:

    const unsigned int myKey;

protected:

    Persistent();    //use class wide list to make sure keys are unique

private:

    static X* ourFirstInstance;
    static const char* ourClassName;
    X* myNextInstance;
};
```

Usage with static allocation

- Statically allocated instances work fine
  - Key enumeration in order of allocation
  - Out of our control, but is the same every time
  - Will mess up if you move objects around

```
class Map : public Persistent<Map>
{
};

//if you swap these declarations, objects will “change instance”
Map theMaps[64];
Map theOtherMaps[16];
```

Usage dynamic allocation

- Think in relational terms
  - Class = Table, Instance = Instance
  - Exploit class wide instance list

```
template <class X>
class Persistent
{
public:

    static X* first();   //returns first instance
    static X* find(const unsigned int aKey);
    X* next();           //returns next instance

    const unsigned int myKey;

protected:

    Persistent();    //use class wide list to make sure keys are unique

private:

    static X* ourFirstInstance;
    static const char* ourClassName;
    X* myNextInstance;
```

Usage dynamic allocation

- Allows for some nifty stuff

```
class Thing : public Persistent<Thing>
{
};

//application code
new Thing;    //implicit key
new Thing(49);    //explicit key

Thing* t = Thing::first();
while(t)
{
    t->method();
    t = t->next();
}

t = Thing::find(49);
```

How to actually persist

- Memory map member variables
- Store to “persistence context”

```
class Thing : public Persistent<Thing>
{
public:

     int myValue;
     String myString;
};

Thing::Thing()
:myValue(0)
,myString(“default”)
{
     //general purpose mapping of any memory
     map(“myValue”, &myValue, sizeof(myValue));

     //string version
     map(“myString”, myString);

     //pull maps from context
     loadMaps(context());
}

Thing::~Thing()
{
     //push maps to context
     saveMaps(context());
}
```

How to actually persist

- Baseclass exposes memory mapping functions
  - Usually only need generic version (void*) and string version
- Persistence context implementations traverse memory maps
  - default values / pull-model allows for transparent load failure
  - In general, data is unique like this:
    - Class name (string)
    - Instance key (unsigned int)
    - Member name (string)
    - Member size (unsigned int)
    - Member data (void*)
  - Various implementations:
    - One file per instance (49.Thing)
    - One file per class (Thing.persistent)
    - One file per context (Model.sfc)
    - Map classes to actual relational database
    - ?

Summary

- Clients can ignore details of persistence
- Basically this works like a language extension
  - i.e. “this class is persistent, specifically these members”
- Application behaves like variable state is “as you left it”
- Game production gains
  - Levels can be built incrementally
  - Data can be tweaked more easily and often (due to in-app editing)
- Beware!
  - Schema evolution
    - You must migrate data via code, or build specific tools
  - Use version control on context files, like any other asset
  - You still need to track asset dependencies
    - With one file per context, there are generally less files total