Game Developers
Conference

Data-Driven Programming Made Easy

Eric Malafeew

Harmonix Music Systems

(emalafeew@harmonixmusic.com)

SAN FRANCISCO CA
MAR 7-11
GDC ›05

Data-driven programming

- Data drives logic
- Parameterization and scripting
- Benefits
  - Faster change - test cycle
  - Open program to designers
  - More reusable C++
  - Enables dynamic programming

Our system

- Evolved over 5 years
- LISP inspiration
- Data format + API + script engine
- Small (70K) but widely used

Python experience

- Just wanted token → int
- Got token → command string → interpreter → function → pyint → int
- Wrapper hell
- 3 Mb tough on 32 Mb console

Talk topics

- Working with data
- Scripting support
- Advanced integration
- Wrap up

Topic 1: Working with data

Data format

- Basic element: arrays
- Array nodes can be any type
  - Subarrays, ints, floats, strings, symbols, more
- Load from file or create

Example: Data file

```
(menu
   (“bacon and eggs”
      (price 12.75)
      (calories 120)
   )
   (“fruit and cereal”
      (price 11.75)
      (calories 12)
   )
)
```

Anti-example: XML file

```
<menu>
  <item>bacon and eggs
    <price>12.75</price>
    <calories>120</calories>
  </item>
  <item>fruit and cereal
    <price>11.75</price>
    <calories>12</calories>
  </item>
</menu>
```

Anti-example: Raw file

- No structure or annotation

```
“bacon and eggs”
12.75
120
“fruit and cereal”
11.75
12
```

Memory representation

- DataArray
  - DataNode*
    - 4-byte value (union of int, float, pointers)
    - 4-byte type
  - Size
  - File, Line
  - Reference count

Packed into 12 bytes

- **Low overhead**
- **Serializable**

Basic API

```
// Parse using Flex
DataArray* menu = DataReadFile(“menu.dta”);

// Price
menu->FindArray(“eggs”)->FindFloat(“price”);

// Error message on wrong type
menu->FindArray(“eggs”)->FindInt(“price”);

// Says ’12.75’ not int (menu.dta, line 3)
```

Nodes are smart pointers

- Of reference counted types, like arrays and heap strings
- That’s all to memory management

Create your own arrays

```
// (price 12.75)
DataArray* price = new DataArray(2);
price->Node(0) = “price”;
price->Node(1) = 12.75;

// (eggs (price 12.75))
DataArray* arr = new DataArray(2);
arr->Node(0) = “eggs”;
arr->Node(1) = price; // adds ref count

price->Release();
```

Don’t actually use strings

- Mostly use “symbols”
- Unique permanent string
- Saves memory for multiple instances
- Fast pointer comparisons
- Still need heap strings

Array searches

```
menu->FindArray("eggs")->FindFloat("price");
```

- Of subarrays with symbol tag
- Linear with pointer comparisons
- For more speed, sort subarrays for binary search

Also in data files

- Comments
- Macros
- #include
- #merge
- #ifdef

Macros

- Persistent outside of file
- Multiply reference arrays

```
[TRUE 1] // (happy TRUE) -> (happy 1)
[RED 1 0 0] // (color RED) -> (color 1 0 0)

[HANDLER (hit {fall_down})]
(object1 HANDLER)
(object2 HANDLER)
```

Merging data files

- For each subarray, look for match
  - Insert if not found
  - Recurse if found

```
(a (b 1)) // original
(a (b 2) (c 2)) // merge
(a (b 1) (c 2)) // result
```

Cache files for fast loading

- Avoid text parsing
- Load then serialize into binary file
- Requires special handling of macros and #ifdef

Program configuration

- Load config file at startup
- Globally accessible
- Encrypt on caching (or you may be mailed your game cheats)

A default config file

```
(renderer
   (show_timers TRUE)
   (screen_size 640 480)
   (clear_color 0.3 0.3 0)
)

(mem
   (heap (size 30000000) (name bob))
   (enable_tracking TRUE)
)
```

Override in app config file

```
(renderer
   (show_timers FALSE)
)

(mem
   (heap (name fred))
)

#merge default.dta
```

Reloading on-the-fly

- Reload portion of database
- Notify dependent C++
- Must group parameters by user

In-memory param editing

- Interface to cycle and change params
- Provide extra info for editing
- How to save

```
(box
   (width 2 (range 0 10) (step 1) (desc “Box width”))
   (height 3 (range .1 5) (step .1) (desc “Box height”))
)
```

Topic 2: Scripting support

When you want “code” in data

- Indicated when data is too fragile or limited
- Total control of program flow
- Wordier than data
- Combine with data, don’t subsume it with scripting

Script inside data

```
(object
   (height 10)
   (collide
      {play “bonk.wav”}
      {game add_score 10}
   )
)
```

Data inside script

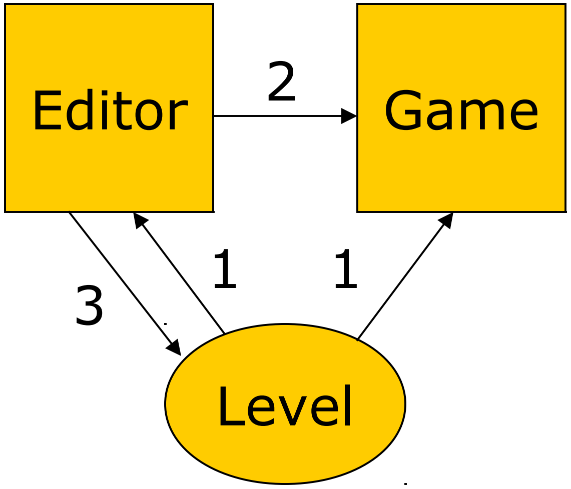
```
{setup_player (name eric) (height 6) (weight 170)}

(launchpad
   (run_over ($obj)
      {$obj enter_flight (force 10) (auto_align TRUE)}
   )
)
```

- **This data you can hard-code**

Uses for scripting

- Event handling in our UI and world systems
- Custom tool plugins
- Command console
- Remote level editing
- C++ messaging system

Remote level editing



1. Load level
2. Preview changes using serialized script protocol
3. Save level

Commands look like

```
{<func> <args>} or {<object> <method> <args>}

{if {game_over} {print “winner”}}

{renderer set_clear_color 1 1 0}

{game add_points {banana worth}}
```

- **DataArray but different type**

Executing func command

```
array->Command(1)->Execute();
```

- Lookup C++ “handler” registered with <func> name
- Call it with actual command array
- Arguments are evaluated inside handler

C++ func handler

```
DataNode Add(DataArray* cmd)
{
   // {+ a b}
   return cmd->Int(1) + cmd->Int(2);
}

DataRegisterFunc(“+”, Add);
```

- **Returns a node**

Implicit arg evaluation

- Node accessors, by default, automatically execute commands and provide return value
- Unless accessed as command
- Trades LISP complexity for a little danger

More on scripting

- “Language” is just built-in funcs
  - Avoid stupid names
- Optimization: bind handler to <func> node on first execute
- Document your script hooks
- Not compiled

Script variables

- Globally named data nodes
- Pointed to by variable nodes
- Automatically evaluate on access, like commands, by dereferencing pointer

Access from C++ or script

```
DataVariable(“game_time”) = 500;

{print “game time is” $game_time}

{set $game_time 100}

int time = DataVariable(“game_time”).Int();
```

Dynamic scoping

- Push variables onto a stack
- Use them locally
- Pop stack and restore values
- Trades LISP lexical scoping for simplicity

Local variables

```
{do ($a)
   {set $a {some_func}}
   {other_func $a}
}
```

- “do” func implements dynamic scoping for arbitrary body commands

Executing object commands

```
{<object> <method> <args>}
```

- **Look up object by name**

```
DataObject* object = NamespaceFind(<object>);
```

- **Call virtual Handle with command**

```
object->Handle(cmd);
```

DataObject

```
class DataObject
{
   const char* name;
   virtual DataNode Handle(DataArray*) = 0;
};
```

- Name stored in a namespace
- Can have NULL name, bind directly to nodes and vars

Calling objects

```
{bob grow 10}

{$focus_character set_speed 5}

{{nearest_object $bomb_position} suffer_damage}

{iterate_materials $mat
   {$mat set_alpha 0.5}
}
```

Virtual Handle implementation

- Map <method> to C++ methods using macro language, like MFC

```
BEGIN_HANDLERS(Person)
   HANDLE(grow, OnGrow)
   HANDLE_EXPR(height, mHeight)
   HANDLE_SUPERCLASS(Parent)
   HANDLE_CHECK
END_HANDLERS
```

C++ object handler

```
DataNode Person::OnGrow(DataArray* cmd)
{
   // {object grow 10}
   mHeight += cmd->Float(2);
   TellMom();
   return mHeight;
}
```

- Or wrap existing C++ method

Topic 3: Advanced integration

Script-side funcs

- So scripts can call scripts

```
{func add1 ($a)
  {+ $a 1}
}

{add1 4} // 5
```

- Makes DataFuncObj “add1”

DataFuncObj

```
class DataFuncObj
{
   DataArray* mFunc;
   virtual DataNode Handle(DataArray*);
}
```

- **Handle assigns arguments with dynamic scoping, executes body and returns last expression**

Script object handlers

```
(dude
   (hit ($force)
         {play “bonk.wav”}
         {if {> $force 10} {$this fall_down}}
   )
   (miss {print “whoosh.wav”})
)
```

- **Associate with C++ object by DataClass**

DataClass

```
class DataClass: public DataObject
{
   DataArray* mHandlers;
   DataArray* mParams;
   virtual DataNode Handle(DataArray*);
}
```

- Handle finds <method> then executes like script func
- Assign $this *after* arg evaluation

Share handlers with macros

```
[OBJECT
   (miss {play “whoosh.wav”})
   (local_hit ($p) {game add_points $p})
]

(banana OBJECT
   (hit {$this local_hit 10})
)
(berry OBJECT
   (hit {$this local_hit 20)})
)
```

More on DataClass

- Supports instance parameters

```
dude->Set(“strength”, 10);
{$this get strength}
```

- Script-side classes possible

```
{class Person <handlers>}
{new Person Bob}
```

Calling handlers from C++

- Then can call C++ or script handlers from either C++ or script
- Use Message class to make command array

```
object->Handle(Message(“hit”, 20)); // {“” hit 20}
```

Specializing Message

- When designing Message before handlers

```
class HitMsg: public Message
{
   HitMsg(int points): Message(“hit”, points) {}
   int Points() { return mCmd->Int(2); }
}

object->Handle(HitMsg(20));
```

Specialized C++ handling

```
HANDLE_MSG(HitMsg)

DataNode Object::OnMsg(const HitMsg& m)
{
   return TheGame->AddPoints(m.Points());
}
```

- Look Ma, no DataArrays!
- Use for all C++ messaging

Specialized script handling

- Match with specialized macros
- Can then change specialization without breaking handlers

```
[HIT hit ($points)]

(object
   (HIT {game add_points $points})
)
```

Balancing C++ and script

- Use script handlers
  - For flexibility and prototyping
  - To avoid C++ dependencies
  - Reduce C++ subclasses
- Use C++ handlers
  - Special arg handling
  - Performance, maintainance

Topic: Wrap up

Script tasks

- **Commands that execute over time**

```
{scheduler delay_task 100 {print “100 ticks later”}}

{scheduler interp_task $frame 0 100 {use $frame}}

{scheduler thread_task
   {walk_to A}
   {wait {near A}}
   {walk_to B}
}
```

More on tasks

- Must preserve variables used in script from construction time
- Done now explicitly, investigating LISP closures

```
{scheduler delay 100 (preserve $msg)
   {print $msg}
}
```

Script debugging

- Dump script call stack on ASSERT

```
Error: Something’s not right
Script calls:
   arena.dta, line 45
   game.dta, line 20
```

- Print statements!
- Interactive debugger next

Conclusion

- Hope this helped to design and use your data system
- Slides available after GDC at

  http://www.harmonixmusic.com/gdc.htm

- Questions?