Adventures in Data Compilation

Uncharted: Drake's Fortune

Dan Liebgold

Naughty Dog, Inc.
Santa Monica, CA

Game Developers Conference, 2008

Outline

Motivation

- Code is compiled, data is “built”
- What should be code, what should be data? Plenty, right?
  - Game logic, geometry, textures...
- What is not clearly either?
  - Particle definitions, animation states & blend trees, event & gameplay scripting/tuning, more...

Motivation

- Code is compiled, data is “built”
- What should be code, what should be data? Plenty, right?
  - Game logic, geometry, textures...
- What is not clearly either?
  - Particle definitions, animation states & blend trees, event & gameplay scripting/tuning, more...

Motivation

- Code is compiled, data is “built”
- What should be code, what should be data? Plenty, right?
  - Game logic, geometry, textures...
- What is not clearly either?
  - Particle definitions, animation states & blend trees, event & gameplay scripting/tuning, more...

Motivation

- Code is compiled, data is “built”
- What should be code, what should be data? Plenty, right?
  - Game logic, geometry, textures...
- What is not clearly either?
  - Particle definitions, animation states & blend trees, event & gameplay scripting/tuning, more...

Motivation

- Code is compiled, data is “built”
- What should be code, what should be data? Plenty, right?
  - Game logic, geometry, textures...
- What is not clearly either?
  - Particle definitions, animation states & blend trees, event & gameplay scripting/tuning, more...

The in between stuff

- We have a legacy of Lisp at Naughty Dog
  - Common Lisp, GOAL, GOOS, GOOL, scripting, animation tools – more than a dozen Lisps all told.
  - GOAL is the primary influence. We stopped using it, so we need something to replace some of its features.
- Lisp supports the code/data duality implicitly
- It also has features (like macros, symbol table) that open unanticipated opportunities
- We will build a solution in Lisp (well, Scheme actually)

The in between stuff

- We have a legacy of Lisp at Naughty Dog
  - Common Lisp, GOAL, GOOS, GOOL, scripting, animation tools – more than a dozen Lisps all told.
  - GOAL is the primary influence. We stopped using it, so we need something to replace some of its features.
- Lisp supports the code/data duality implicitly
- It also has features (like macros, symbol table) that open unanticipated opportunities
- We will build a solution in Lisp (well, Scheme actually)

The in between stuff

- We have a legacy of Lisp at Naughty Dog
    - Common Lisp, GOAL, GOOS, GOOL, scripting, animation tools – more than a dozen Lisps all told.
    - GOAL is the primary influence. We stopped using it, so we need something to replace some of its features.
- Lisp supports the code/data duality implicitly
- It also has features (like macros, symbol table) that open unanticipated opportunities
- We will build a solution in Lisp (well, Scheme actually)

The in between stuff

- We have a legacy of Lisp at Naughty Dog
  - Common Lisp, GOAL, GOOS, GOOL, scripting, animation tools – more than a dozen Lisps all told.
  - GOAL is the primary influence. We stopped using it, so we need something to replace some of its features.
- Lisp supports the code/data duality implicitly
- It also has features (like macros, symbol table) that open unanticipated opportunities
- We will build a solution in Lisp (well, Scheme actually)

The in between stuff

- We have a legacy of Lisp at Naughty Dog
  - Common Lisp, GOAL, GOOS, GOOL, scripting, animation tools – more than a dozen Lisps all told.
  - GOAL is the primary influence. We stopped using it, so we need something to replace some of its features.
- Lisp supports the code/data duality implicitly
- It also has features (like macros, symbol table) that open unanticipated opportunities
- We will build a solution in Lisp (well, Scheme actually)

The in between stuff

- We have a legacy of Lisp at Naughty Dog
    - Common Lisp, GOAL, GOOS, GOOL, scripting, animation tools – more than a dozen Lisps all told.
    - GOAL is the primary influence. We stopped using it, so we need something to replace some of its features.
- Lisp supports the code/data duality implicitly
- It also has features (like macros, symbol table) that open unanticipated opportunities
- We will build a solution in Lisp (well, Scheme actually)

Let’s define some types

- A DC type declaration:

```
(deftype vec4 (:align 16)
 ((x float)
  (y float)
  (z float)
  (w float :default 0)
  ))
```

- Automatically gets translated to a C++ declaration:

```
struct Vec4
{
  float m_x;
  float m_y;
  ...
};
```

Let's define some types

- A DC type declaration:

```
(deftype vec4 (:align 16)
 ((x float)
  (y float)
  (z float)
  (w float :default 0)
  ))
```

- Automatically gets translated to a C++ declaration:

```
struct Vec4
{
  float m_x;
  float m_y;
  ...
};
```

Types continued

We define some more 3D types

```
(deftype quaternion (:parent vec4)
 ())

(deftype point (:parent vec4)
 ((w float :default 1)
  ))

(deftype locator ()
 ((trans point :inline #t)
  (rot quaternion :inline #t)
  )
 )
```

Types continued

We define some more 3D types

```
(deftype quaternion (:parent vec4)
 ())

(deftype point (:parent vec4)
 ((w float :default 1)
  ))

(deftype locator ()
 ((trans point :inline #t)
  (rot quaternion :inline #t)
  )
 )
```

Types continued

We define some more 3D types

```
(deftype quaternion (:parent vec4)
 ())

(deftype point (:parent vec4)
 ((w float :default 1)
  ))

(deftype locator ()
 ((trans point :inline #t)
  (rot quaternion :inline #t)
  )
 )
```

Types continued

We define some more 3D types

```
(deftype quaternion (:parent vec4)
 ())

(deftype point (:parent vec4)
 ((w float :default 1)
  ))

struct Locator
{
  Point m_trans;
  Quaternion m_rot;
};
```

Define some instances

```
(define *y-axis* (new vec4 :x 0 :y 1 :z 0))
(define *origin* (new point :x 0 :y 0 :z 0))
```

This instance will be exported (available at runtime):

```
(define-export *player-start*
 (new locator
      :trans *origin*
      :rot (axis-angle->quaternion *y-axis* 45)
      ))
```

Define some instances

```
(define *y-axis* (new vec4 :x 0 :y 1 :z 0))
(define *origin* (new point :x 0 :y 0 :z 0))
```

This instance will be exported (available at runtime):

```
(define-export *player-start*
 (new locator
      :trans *origin*
      :rot (axis-angle->quaternion *y-axis* 45)
      ))
```

How we use these definitions in C++ code

In our runtime C++ code:

```
...
#include "dc-types.h"
...
const Locator * pLoc =
  DcLookupSymbol("*player-start*");
Point pos = pLoc->m_trans;
...
```

Build upon this basis

We build upon this basis to create many many things

- Particle definitions
- Animation states
- Gameplay scripts
- Scripted in-game cinematics
- Weapons tuning
- Sound and voice setup
- Overall game sequencing and control
- ...and more