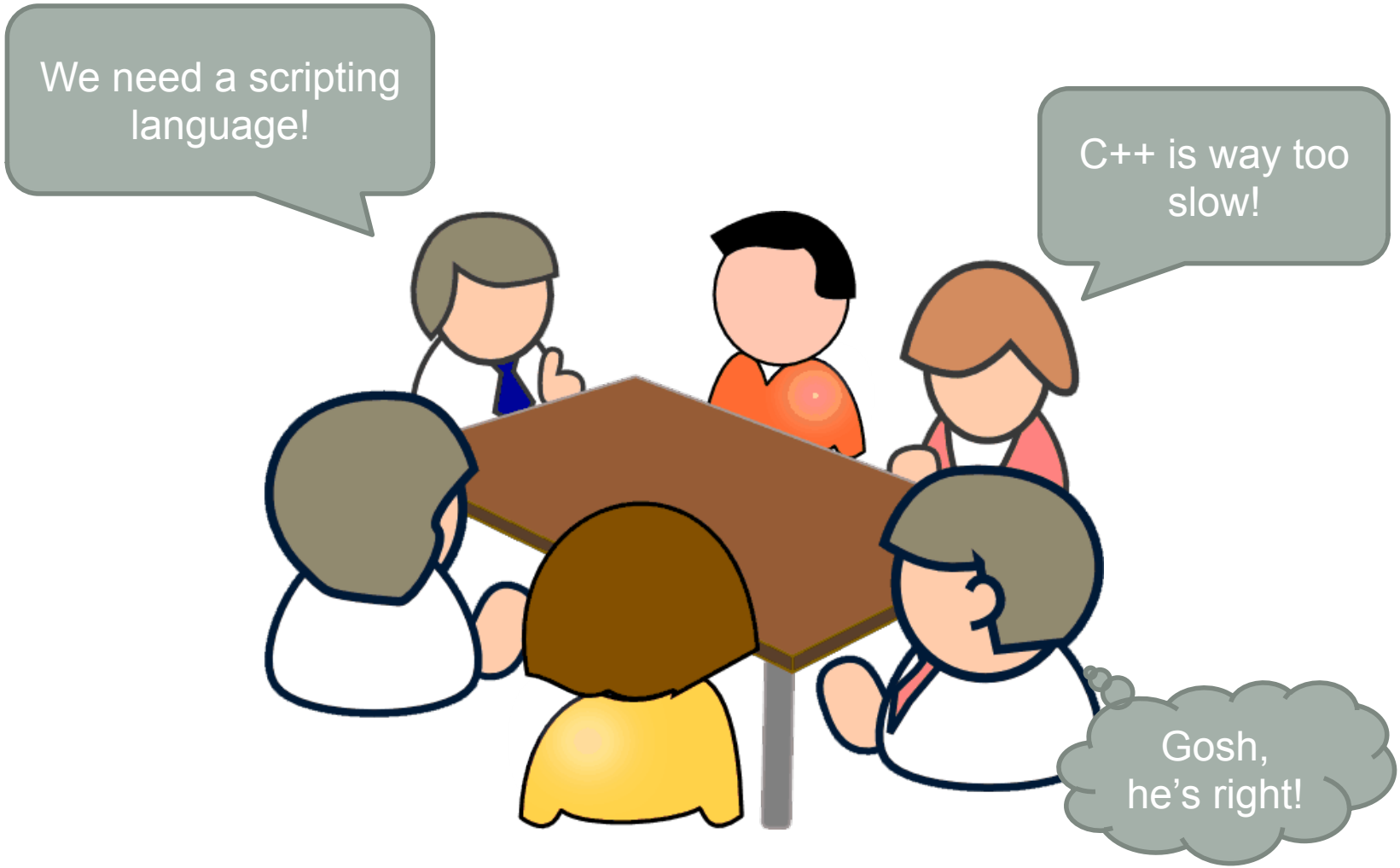
USING D ALONGSIDE A TYPICAL GAME ENGINE

With Manu of Brisbane

D IS AWESOME… BUT CAN IT BE USED IN A COMMERCIAL GAME?

Let me tell you a story...

“How did we come to this?”, I hear you ask...

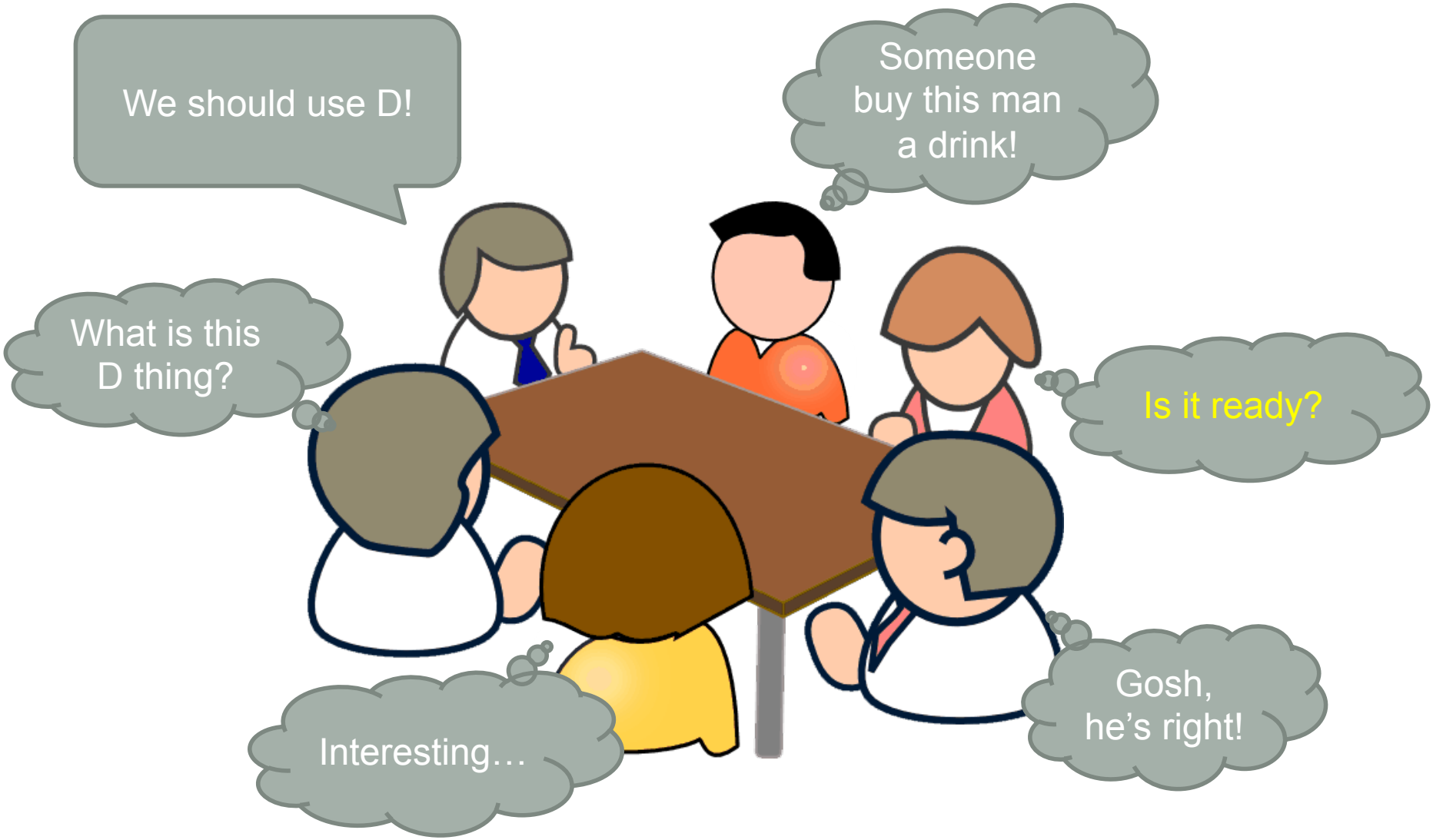


“How did we come to this?”, I hear you ask...

Considerations

	Light weight	Modern	Performance
Lua	★★★★★	★★★	👎
C#	👎	★★★★★	★★★
C++	★★★★	👎	★★★★★
D	★★★☆	★★★★★	★★★★★

So we took it to the team…



Is it ready?

This is a fair concern…

D has little experience in the commercial space.

Hedge our bets:

- Initial plan was to use C-in-a-DLL
- Build a framework that works with either language
- If D doesn't work out, fall back to C/C++

AND THUS IT BEGINS…

Requirements

- Windows/Visual Studio workflow
  - Visual-D! 
- Target x64/Win64
- Symbolic Debugging
-  No DMD! (only supports Win32)

Options?

- LDC?
  - No Win64 exceptions or debug info
- GDC?
  - Working Win64 build!
  - 2 enthusiastic developers
  - Good for prototype…
- We really need DMD

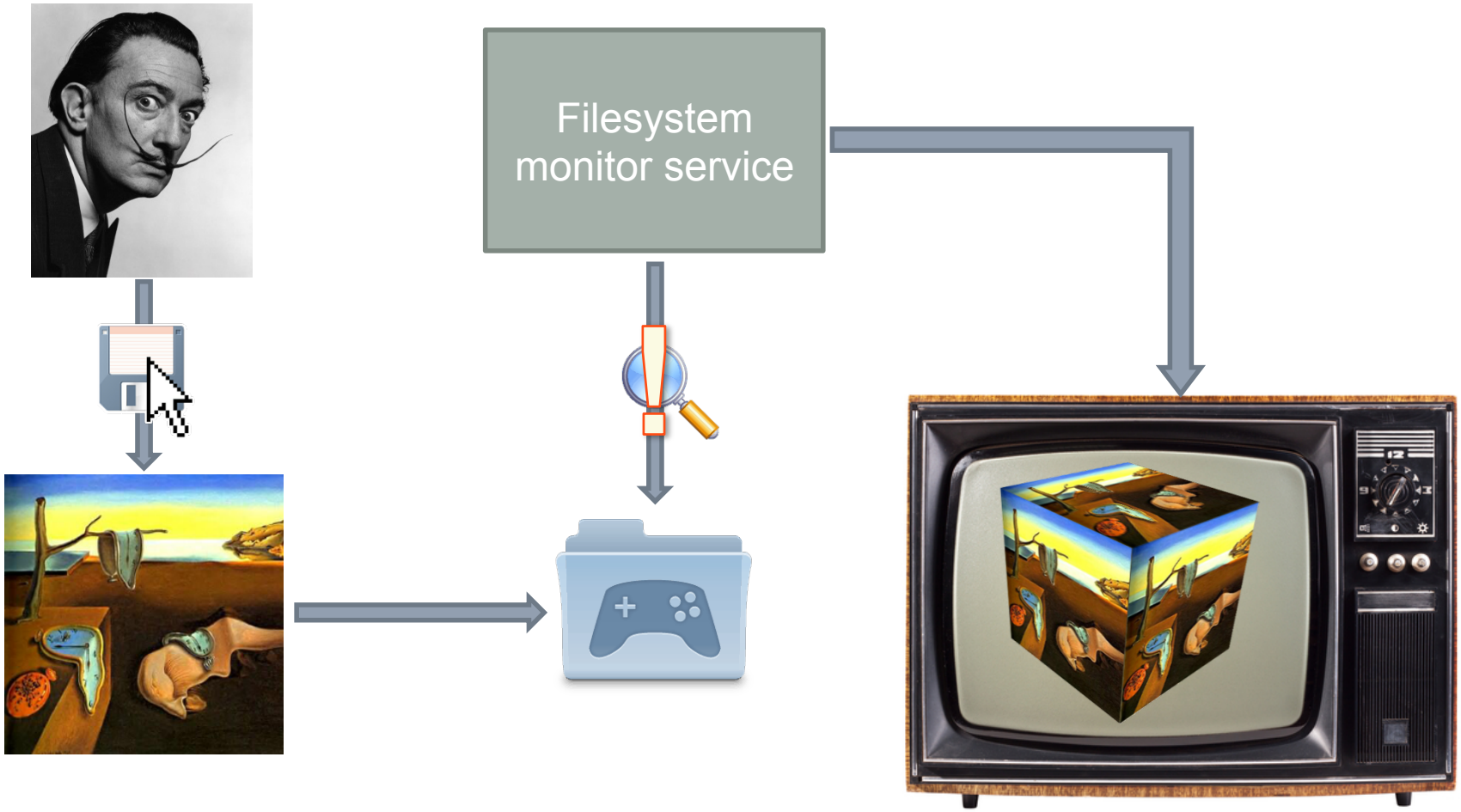
SO WE HAVE A COMPILER

What do we do with it?

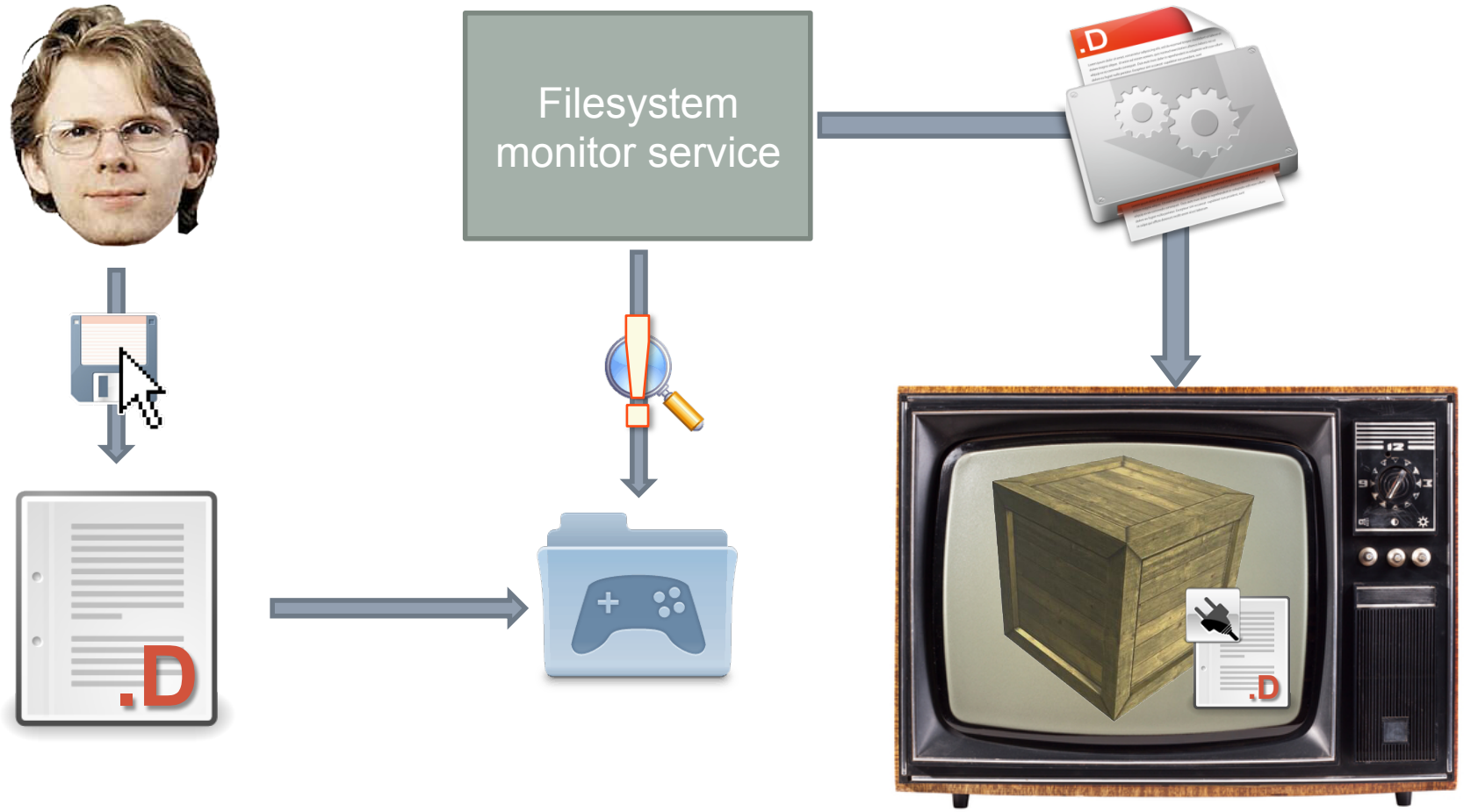
Goals

- Rapid iteration
- Dynamic linkage
- Interact with the engine
- Offer new functionality to the engine
- Retain objects state across update cycle

1. Rapid iteration

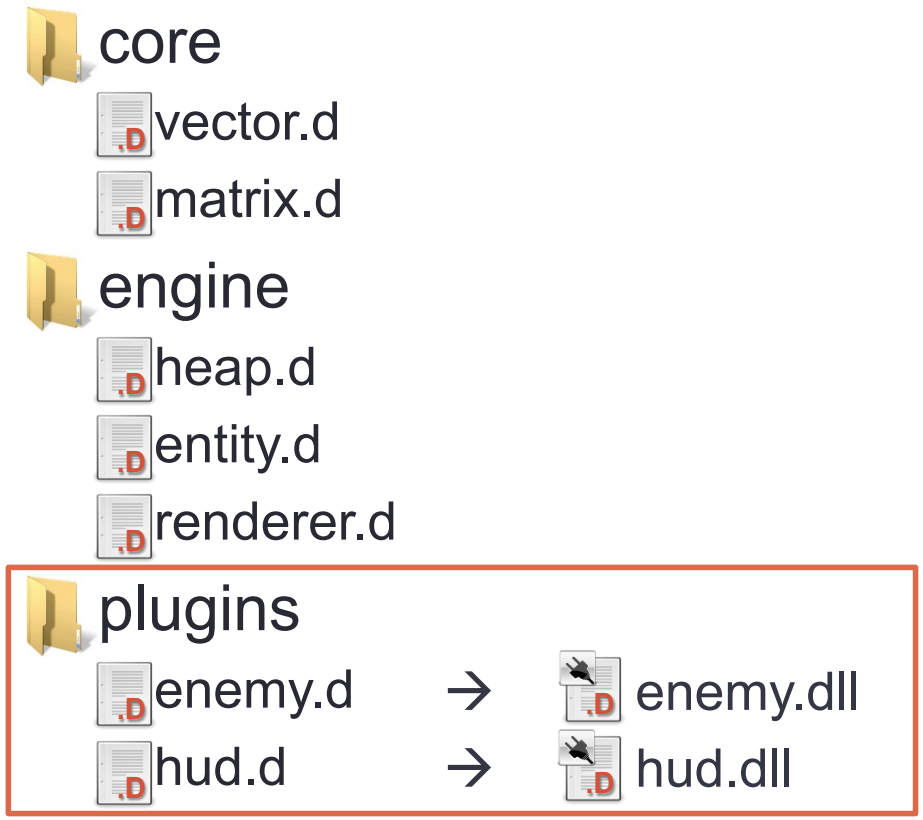


1. Rapid iteration



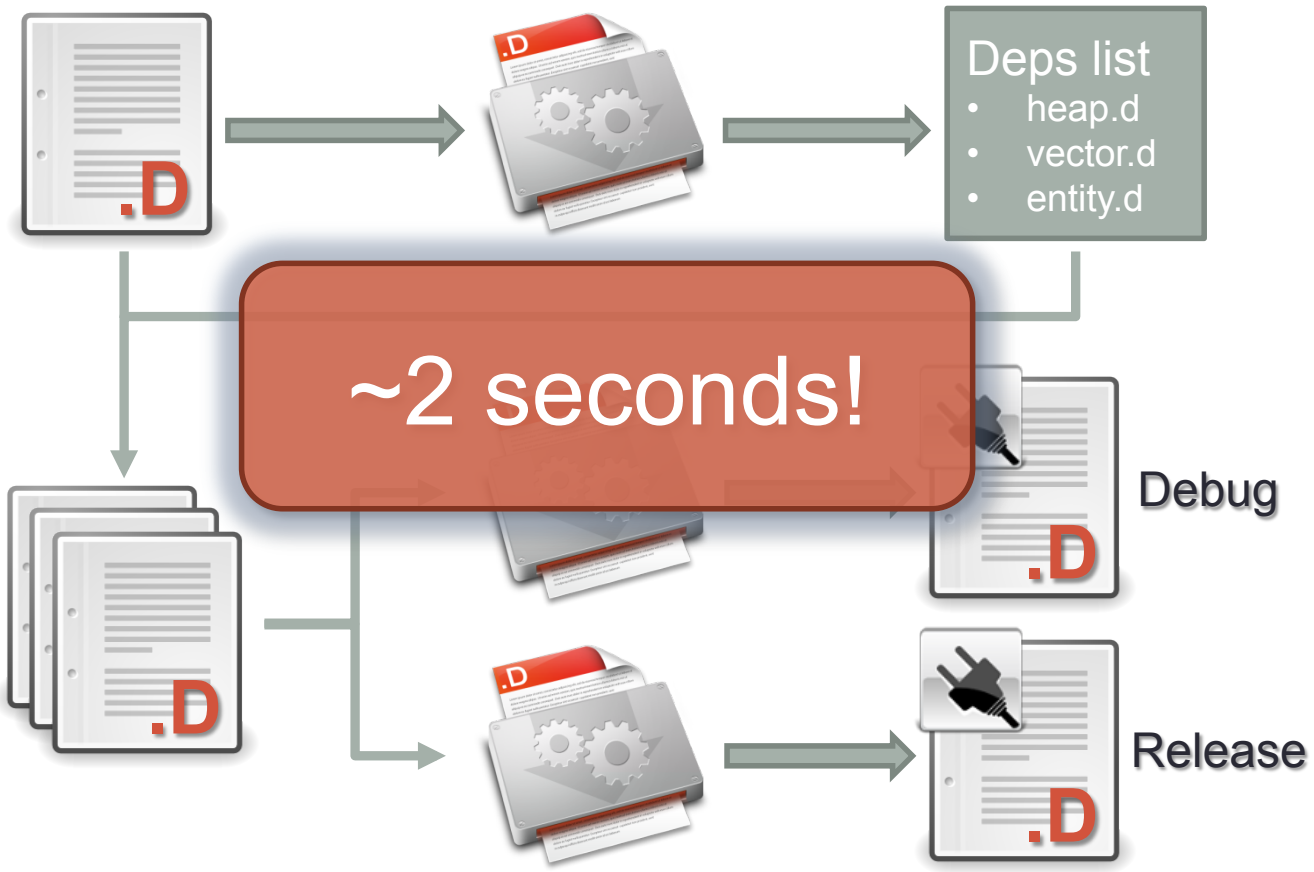
1. Rapid iteration

Lots of code... How do we build?



1. Rapid iteration

Compiling performed in 2 passes



2. Dynamic Linkage

We now have ‘plugin’ DLL’s, we need to load them…

We wrote a fairly simple PluginManager, which:

- Scans for any plugins and loads them on startup
- When it receives an update signal:
  - Save the state of all object instances created by the plugin
  - Unload the DLL
  - Reload the rebuilt DLL
  - Recreate object instances from the saved state

The implementation is not particularly interesting.

…and we’ll talk about the state management later.

3. Interact with the engine

So we have a system that will manage our plugins.

To be useful, they need access to the engine…

Which it turns out is not so simple!

D needs access to C++:

- Structs
- Static functions
- Classes

We'll look at each of these...

3. Interact with the engine

Structs

- Mirror definition in D
- No way to assert that C++ and D definitions remain in synch…
  - I'm open to suggestion!

3. Interact with the engine

Functions

- D supports C/C++ ABI
- C++ registry of functions to share with D
- D stubs which link on module load

- Not so hard right?

3. Interact with the engine

In practise - Uh oh, code!

experiment.cpp

```
void engineFunc(int x, float y)
{
  // awesome functionality!
}

EXPORT_FUNCTION(engineFunc) // you don't wanna know!
```

experiment.d

```
__gshared extern (C) void function(int x, float y) engineFunc;

mixin RegisterModule;
```

```
shared static this()
{
  registerImports(
    (EngineImport imports[]) {
      engineFunc = findFunction(imports, "engineFunc");
    }
  );
}
```

3. Interact with the engine

Surely we can do better than that…

```
void engineFunc(int x, float y);

mixin RegisterModule;
```

```
shared static this()
{
  registerImports(
    (EngineImport imports[]) {
      engineFuncPtr = findFunction(imports, "engineFunc");
    }
  );
}

private __gshared extern (C) void function(int, float) engineFuncPtr;
void engineFunc(int x, float y);
{
  engineFuncPtr(x, y);
}
```

- Now we can prototype and declare in the same module…
- Enhance the mixin to generate a stub

3. Interact with the engine

Supporting overloads - on the C++ side

```
void engineFunc(int x, int y)
{
  // awesome functionality!
}

void engineFunc(int x, float y)
{
  // even more awesome functionality!
}

EXPORT_FUNCTION(engineFunc, void, int, int) // you RELLY don't wanna see this now!
EXPORT_FUNCTION(engineFunc, void, int, float)
```

```
#RET_TYPE "(" #__VA_ARGS__ ")" == "void(int, float)"
```

- Supply the argument information to disambiguate
- Use stringification to generate a function signature string

3. Interact with the engine

Supporting overloads - on the D side

```
void engineFunc(int x, int y);
void engineFunc(int x, float y);

mixin RegisterModule;
```

```
shared static this() {
  registerImports(
    (EngineImport imports[]) {
      engineFunc_int_int   = findFunction(imports, "engineFunc", "void(int,int)");
      engineFunc_int_float = findFunction(imports, "engineFunc", "void(int,float)");
    }
  );
}

private __gshared extern (C) void function(int, int) engineFunc_int_int;
void engineFunc(int x, int y) {
  engineFunc_int_int(x, y);
}

private __gshared extern (C) void function(int, float) engineFunc_int_float;
void engineFunc(int x, float y) {
  engineFunc_int_float(x, y);
}
```

- Generate a similar string from the D type info
- Mangle the function pointer names

3. Interact with the engine

Attributes!

```
private @Import void engineFunc(int *things, size_t numThings);

void engineFunc(int[] things)
{
  engineFunc(things.ptr, things.length);
}

mixin RegisterModule;
```

- Provide nice D API's with trivial wrappers
- RegisterModule recognises functions marked `@Import`

- Awesome! Maybe we're done here?

3. Interact with the engine

Classes

D does not interact with C++ classes very well.

- Can’t extern to C++ methods
- Virtuals are tricky

3. Interact with the engine

Static methods

```
class EngineClass
{
  void method(int x);
}

EXPORT_METHOD(EngineClass, engineFunc, void, int, float)
```

```
struct EngineClass
{
  @Import void method(int x);

  mixin RegisterClass;
}

mixin Registe
```

```
private extern (C) __gshared void function(EngineClass* _this, int x) method_int;
void method(int x)
{
  method_int(&this, x); // explicitly call with 'this'
}
```

- Export a C++ member function pointer
- Declare just like other functions, but we have some new magic…
- Abuse our knowledge of the ABI. Not portable!

3. Interact with the engine

And finally, virtuals…

```
class EngineClass
{
  virtual void virtualMethod();
}
```

```
extern (C++) interface IEngineClass
{
  void virtualMethod();
}

struct EngineClass
{
  @property IEngineClass _vtable() { return cast(IEngineClass)&this; }
  alias _vtable this;
}
```

- We don't need to do anything in C++!
- Use an '`extern(C++) interface`' to mirror the vtable
- Use 'alias this' to incorporate it into a struct

4. Making use of plugins

The game needs to make use of this somehow!

- Same features in reverse
- But no existing code, we can make restrictions
  - Static functions
  - Opaque classes
  - Use interfaces

Let’s look at these in practise… (more code, sorry!)

4. Making use of plugins

Static functions

```
@Export extern (C++) void dFunc(int x)
{
  // do something amazing!
}

mixin RegisterModule; // RegisterModule handles @(Export) too!
```

```
void somewhere()
{
  DFunc<int> dFunc = PluginManager::findFunction("dFunc");

  if(dFunc)
    dFunc(42);
}
```

- Find functions by name
- DFunc implements a smart pointer to handle module reload
- C++ can’t assert the signature matches
  - Perhaps a template solution is possible?

4. Making use of plugins

Classes - on the D side

```d
@Export extern (C++) interface IFeature
{
  void doSomething();
}

@Export class Feature : IFeature
{
  extern (C++) void doSomething()
  {
    // do something…
  }
}

mixin RegisterModule;
```

```d
shared static this()
{
  exportInterface("IFeature", (Object o) => cast(IFeature)o);

  exportClass(
    /+ name:    +/ "Feature",
    /+ create:  +/ ()          => new Feature,
    /+ destroy: +/ (Object o) => delete cast(Feature)o
  );
}
```

- Interface registers cast function
- Class registers create/destroy functions

4. Making use of plugins

Classes - on the C++ side

```
class IFeature
{
  virtual void doSomething() = 0;
}

void somewhere()
{
  DClass *pClass = PluginManager::newClass("Feature");

  IFeature *pFeature = pClass->queryInterface("IFeature");
  if(pFeature)
    pFeature->doSomething();

  pClass->deleteClass();
}
```

- Declare a mirror of the exported interface
- Create new class instance by name
- Query the opaque DClass object for interfaces
- If the class implements that interface, we can use it

4. Making use of plugins

We have everything we need!

We can create new entities that exist in the game world…
But what happens when a live class definition is changed?

Hint: It crashes spectacularly!

Which leads to our final goal…

5. Retain object state

Runtime code iteration is the principle goal…

What if we modify a data structure?

```
class Dude
{
  vector position;
  float health;
}
```

```
class Dude
{
  vector position;
  vector velocity; // add a new variable…
  float health;
}
```

Existing objects are incompatible with rebuilt code.

We can use serialisation to migrate the data...

5. Retain object state

Approach:

- PluginManager keeps registry of all instances
- RegisterModule mixin produces serialisation functions for structs/classes

Reload process:

- All instances are serialised to text
- Destroy all instances
- Unload/reload plugin
- Recreate instances from text
  - New members take on default values
  - If variable changes meaning, we may still crash!
    - But this is rare

Afterthoughts & improvements

We have a system that's working well.

But there are a few rough edges…

- Every module requires `mixin RegisterModule`
- Classes require additional `mixin RegisterClass`
  - Be nice if attributes had a method of introducing code…
- Can't assert that structures or virtuals remain in synch
- D -> C++ function sharing can't assert the signature
  - These are really deficiencies in C++
  - Can C++ templates help us?

WE HAVE A SYSTEM

So, what cool things does it offer?

Stuff programmers love…

We have D as an extension language!

But what makes it cool, and worth all that effort?

1. Coders love ranges, and foreach
   - Seriously, slices are awesome!
   - foreach should not be under-estimated

2. Event based programming
   - Game devs often have C# experience
   - Proper delegates facilitate nice event frameworks
   - C++ FastDelegate is compatible with D! (I'll bet this isn't a coincidence…)

Stuff programmers love…

3. Vector maths
   - Standardised SIMD!
   - opDispatch can be used for shader-style swizzling
     - Game devs often have HLSL/GLSL experience

```
vector cross(vector v1, vector v2)
{
  return v1.yzx*v2.zxy - v1.zxy*v2.yzx;
}
```

   - Perhaps experiment with DSLs in future?
   - Theoretically, `pure` should offer some nice wins…

Stuff programmers love…

4. Attributes are awesome!
   - Really help to simplify code
   - Great to see what class members can do at a glance

Some attributes we use:

- `@SaveGame`
  - Control variables that are written to save data
- `@Profile`
  - Variables will be tracked and charted on realtime graphs
- `@Tweakable`
  - Variables will be added to a runtime ‘tweakable’ menu
- `@Editor("Enemy colour", Colour.Red, Type.ColourPicker)`
  - Variable is exposed to the editor, where property grids are automatically populated

Final thoughts

Video games industry still stuck with C++!

- Native code is a requirement.
- High-risk industry, allergic to change.
- Aggressive schedules; C++ wastes time & sanity.

Industry desperate for salvation.

Using this approach, if D proves successful, we can ween ourselves towards D in the future…

…and may we all live happily ever after.

THAT'S IT

Questions?