## SUNDAY, NOVEMBER 21, 2010

### [Building a Reflective Object System in C++](http://replicaisland.blogspot.de/2010/11/building-reflective-object-system-in-c.html)

Every game engine I've worked with for the last several years has had some sort of reflection system.  *Reflection*, and its cousin *introspection*, are features supported by some languages that allow runtime inspection of object structure and types. My most recent game, Replica Island, as a Java application makes use of Java's Class object, as well as useful type-related tests like *instanceof*.  Many other languages support some sort of reflection, especially those that are modern and VM-based (like C#).  Reflective objects can provide the answers to all sorts of interesting questions at runtime, like "is this anonymous pointer derived from class X?" or "does this object have a field named Y?"  It's also common to be able to allocate and construct objects by string in reflective systems.   
  
These traits make it very easy to serialize and deserialize reflective objects: entire object trees can be written to disk and then read back again without the serialization code having to know anything about the objects it is writing.  Reflection also makes it very easy to write tools; if the tools and the runtime have access to the same type data, the tools can output data in the same format that the runtime will access it in memory (see my previous post about loading object hierarchies).  Reflection systems can also make for some pretty powerful tools: for example, it's easy to build a Property Editor dialog that can edit any object in the code base, and automatically adds support for new classes as they are written over the course of development.  
  
But C++ doesn't support reflection.  There's RTTI, which gives a tiny little sample of the power that a real reflection system brings, but it's hardly worth the overhead on its own.  C++'s powerful template system is often used in places that reflection might be used in other languages (e.g. type independent operators), but that method is also restricted to a small subset of a fully reflective system.  If we want real reflection in C++, we'll need to build it ourselves.  
  
Before we dive into the code, let me take a moment to describe the goals of my reflective object system.  Other reflection systems might choose a different approach based on different needs (compare, for example, Protocol Buffers to the method I'm about to describe).  For me, the goals are:

1. Runtime type information of pointers.
2. Ability to iterate over the fields in an object.
3. Ability to query and set a field in an object.
4. Ability to allocate and construct an object by string, or based on other meta data.
5. No extra overhead for normal class use.

The Java Class object is a pretty good model to follow here: each class has a static object that describes the class in detail.  Using classes normally involves no extra overhead, but if we choose we can also load the Class descriptor and work with objects using reflection.  So my goal is to make something like the Java Class object in C++; my approach is to generate a class for each reflected class that holds metadata about the class it describes.  A "meta object."  
  
Once I start generating meta objects for individual classes, servicing the first goal of anonymous pointer typing isn't very hard.  As long as we know that the pointer comes from some base type that implements a virtual accessor for returning the meta object, we can pull a static class definition out and compare it to other classes, or to a registry of meta objects (and, if we're tricky, we can even do this when we can't guarantee that the pointer is derived from some known base).  So let's assume we have a way to generate meta information for each class, wrap it up in a class called a MetaObject, and stuff it as a static field into the class it describes.  
  
The next question is, what does this MetaObject class contain?  Well, per requirement #2, it must at least contain some information about fields.  In order to access fields within a class we'll need to know the offset of the field, its size (and, if it's an array, the size of each element), and probably the name of the field and the name of its type as strings.   
  
Now might be a good time to think about what a C++ object actually looks like in memory.  Say we have the following object:

class Foo : public MetaBase

{

   public:

      static const MetaObject sMeta;  // The object describing Foo

      virtual const MetaObject\* getMetaObject() { return &sMeta; }  // required by the abstract base

      void setup() { mBar = 10; mBaz = 267; }

   private:

      int mBar;

      int mBaz;

};

If we allocate an instance of Foo and then pop open our debugger to inspect the raw memory, it probably looks something like this (assuming MetaBase has no fields):

C0 A1 D0 F7     // Pointer to the virtual table; some random address

00 00 00 0A     // Value of mBar

00 00 01 0B     // Value of mBaz

I say "probably" because the actual format of a class in memory is basically undefined in C++; as long as it works the way the programmer expects it to, the compiler can do whatever it wants.  For example, the actual size of this object might very well be 16 bytes (rather than the twelve bytes shown above), with zero padding or junk in the last word; some architectures require such padding for alignment to byte boundaries in memory.  Or the vtable might be at the end of the object rather than the beginning (though, to be fair, all the compilers I've worked with have put it at the top).   
  
Anyway, assuming this is what we see in the debugger, it's not hard to pull out the information we want for our meta object.  The value of mBar is at offset 4 (the first four bytes are the address of the virtual table), and the value of mBaz is at offset 8.  We know that sizeof(int) == 4.  And sMeta, because it is static, doesn't actually appear in the class instance at all--it's stored off somewhere else in the data segment.  If we had this information about every field in every class, we'd easily be able to access and modify fields in objects without knowing the type of the object, which satisfies most of the goals above.  And since this data is stored outside the object itself, there shouldn't be any overhead to standard class use.  
  
Here's a abbreviated version of the object I use to describe individual fields in classes.  You can see the entire object [here](http://code.google.com/p/androidndkgame/source/browse/project/jni/game/core/MetaField.h).

class MetaField : public Object

{

public:

enum MetaType

{

TYPE\_value,

TYPE\_pointer,

};

MetaField(const MetaType type, const char\* pName,

const char\* pTypeName, int offset, size\_t fieldSize)

: mType(type),

mpName(pName),

mpTypeName(pTypeName),

mOffset(offset),

mFieldSize(fieldSize),

{};

const char\* getName() const;

const char\* getTypeName() const;

const int getOffset() const;

const size\_t getFieldSize() const;

const MetaType getStorageType() const;

virtual void\* get(const MetaBase\* pObject) const;

virtual void set(MetaBase\* pObject, const void\* pData) const;

private:

const MetaType mType;

const char\* mpName;

const char\* mpTypeName;

const int mOffset;

const size\_t mFieldSize;

};

A static array of MetaFields is defined for each class and wrapped up in a container MetaObject, which also provides factory methods and some other utility functions.  You can see that object [here](http://code.google.com/p/androidndkgame/source/browse/project/jni/game/core/MetaObject.h).  These two objects, MetaField and MetaObject, make up the core of my C++ reflection implementation.  
  
So we know what information that we need, and we have a class structure to describe it.  The hard part is finding a way to generate this information automatically in a compiler-independent way.  We could fill out MetaObjects by hand for every class, but that's error prone.  It might be possible to pull this information out of the debug symbols generated for a debug build, but symbol formats change across compilers and we don't want to compile a debug build for every release build.  We could probably contort C++ macro expansion to generate meta data, but in the interests of sanity let's not do that.  We could write a preprocessor that walks our header files and generates the necessary meta data, but that's actually a sort of annoying problem because of the idiosyncrasies of C++ syntax.   
  
The solution I chose is to use a separate format, an interface definition language, to generate metadata-laden C++ header files using a preprocessor tool.  The idea is that you write your class headers in the IDL, which converts them to C++ and generates the necessary metadata objects in the output as it goes.  I leverage compiler intrinsics like sizeof() and offsetof() to let the compiler provide the appropriate field information (meaning I don't care where the vtable is stored, or what padding might be inserted).  My IDL looks like this:

metaclass PhysicsComponent

{

base GameComponent

function void update(const float timeDelta, GameObject\* pParentObject) { public }

function virtual bool runsInPhase(const GameObjectSystem::GameObjectUpdatePhase phase) { public }

field mMass { type float, value 1.0f, private }

field mStaticFrictionCoeffecient { type float, value 0.5f, private }

field mDynamicFrictionCoeffecient { type float, value 0.1f, private }

// mBounciness = coeffecient of restitution. 1.0 = super bouncy, 0.0 = no bounce.

field mBounciness { type float, value 0.0f, private }

field mInertia { type float, value 0.1f, private }

}

.. and the output of the preprocessor tool looks like this:

class PhysicsComponent : public GameComponent

{

public:

void update(const float timeDelta, GameObject\* pParentObject);

virtual bool runsInPhase(const GameObjectSystem::GameObjectUpdatePhase phase);

private:

float mMass;

float mStaticFrictionCoeffecient;

float mDynamicFrictionCoeffecient;

// mBounciness = coeffecient of restitution. 1.0 = super bouncy, 0.0 = no bounce.

float mBounciness;

float mInertia;

public:

// AUTO-GENERATED CODE

static void initialize(PhysicsComponent\* pObject);

static PhysicsComponent\* factory(void\* pAddress = 0);

static void\* factoryRaw(void\* pAddress, bool initializeObject);

static PhysicsComponent\* arrayFactory(int elementCount);

static const MetaObject\* getClassMetaObject();

virtual const MetaObject\* getMetaObject() const;

static bool registerMetaData();

static PhysicsComponent\* dynamicCast(MetaBase\* pObject);

};

You can see that the IDL pretty much just spits C++ out exactly as it was written, but in the process it also records enough information to generate the functions at the bottom of the class.  The most interesting of those is getClassMetaObject(), which is a static method that defines the meta data itself:

inline const MetaObject\* PhysicsComponent::getClassMetaObject()

{

static MetaField field\_mMass(MetaField::TYPE\_value, "mMass", "float",

offsetof(PhysicsComponent, mMass), sizeof(float));

static MetaField field\_mStaticFrictionCoeffecient(MetaField::TYPE\_value,

"mStaticFrictionCoeffecient", "float",

offsetof(PhysicsComponent, mStaticFrictionCoeffecient), sizeof(float));

static MetaField field\_mDynamicFrictionCoeffecient(MetaField::TYPE\_value,

"mDynamicFrictionCoeffecient", "float",

offsetof(PhysicsComponent, mDynamicFrictionCoeffecient), sizeof(float));

static MetaField field\_mBounciness(MetaField::TYPE\_value, "mBounciness", "float",

offsetof(PhysicsComponent, mBounciness), sizeof(float));

static MetaField field\_mInertia(MetaField::TYPE\_value, "mInertia", "float",

offsetof(PhysicsComponent, mInertia), sizeof(float));

static const MetaField\* fields[] =

{

&field\_mMass,

&field\_mStaticFrictionCoeffecient,

&field\_mDynamicFrictionCoeffecient,

&field\_mBounciness,

&field\_mInertia,

};

static MetaObject meta(

"PhysicsComponent",

MetaObject::generateTypeIDFromString("PhysicsComponent"),

MetaObject::generateTypeIDFromString("GameComponent"),

sizeof(PhysicsComponent),

static\_cast(sizeof(fields) / sizeof(MetaField\*)),

fields,

GameComponent::getClassMetaObject(),

&PhysicsComponent::factoryRaw);

return &meta;

}

\*note that, in more recent versions, I've replaced offsetof() with a macro that does the right thing for compilers that do not support that intrinsic.  Offsetof() isn't really kosher in C++, but for my purposes it works fine.  If you want to learn all about it, and why it's rough for "non-POD types," try [Stack Overflow](http://stackoverflow.com/questions/1129894/why-cant-you-use-offsetof-on-non-pod-strucutures-in-c).  
  
With this data, I now have a pretty complete reflection system in C++.  I can iterate over fields in a class, look them up by string, get and set their values given an anonymous pointer.  I can compare object types without knowing the type itself (I can implement my own dynamic\_cast by walking up the MetaObject parent hierarchy and comparing MetaObject pointers until I find a match or reach the root).  It's very easy to construct objects from a file, or serialize a whole object tree.  I can, for example, make a memory manager that can output an entire dump of the heap, annotated with type and field information for every block.  And I can compile all of my objects into a DLL and load them into a tool and have full type information outside of the game engine environment.  Sweet!  
  
There are, however, many caveats.  This implementation doesn't even attempt to support templates, multiple inheritance, or enums.  If we serialize and deserialize using only this data, some standard programming practices start to get screwed: what happens when we can construct objects without invoking the constructor?  How do we deal with invasive smart pointers or other data that weakly links to objects outside of the immediate pointer tree?  How do we mix objects that have this meta data and objects that do not?  How do we deal with complex types like std::vector?  If object structure is compiler dependent, how can we serialize class contents in a way that is safe across architectures?  These are all solvable problems, but the solutions are all pretty complicated.  They often involve dusty corners of the C++ language that I rarely visit, like placement new.  If you get into this stuff, [Stanley Lippman](http://www.amazon.com/Inside-Object-Model-Stanley-Lippman/dp/0201834545) is your new best friend.  
  
But even with those caveats in mind, the power of reflection is absolutely worth the price of admission.  It's the first chunk of code I write or port whenever I start a new project in C++.  It was the first bits of my old game engine that I got running on Android, and is now the core of the engine I am building on that platform.  Reflection is not a simple bit of infrastructure to get up and running, but once you have it it's really hard to go back.

POSTED BY [CHRIS](http://www.blogger.com/profile/16079744366139636224)AT [7:21 AM](http://replicaisland.blogspot.de/2010/11/building-reflective-object-system-in-c.html) [5 COMMENTS:](http://replicaisland.blogspot.de/2010/11/building-reflective-object-system-in-c.html#comment-form)

LABELS: [CODE DESIGN](http://replicaisland.blogspot.de/search/label/code%20design)

## SUNDAY, NOVEMBER 7, 2010

### [Leveraging Java and C++ for Hybrid Games](http://replicaisland.blogspot.de/2010/11/leveraging-java-and-c-for-hybrid-games.html)

I've been thinking a lot lately about how best to use the resources that Android provides for game development.  A lot of the game developers I know (and I know a lot!) are quick to treat any new platform as a dumb host to their game engines.  Usually developers have a bunch of code, or even entire games, that are written to be aggressively cross-platform, so all they need is a way to compile the source, attach it to input events, and draw to screen.  Any platform that can provide those basics can host their tech, so when evaluating a new platform to support, these developers only look at the most basic level of functionality.  
  
This is certainly true on Android as well.  Lots of developers look at the NDK and see a C++ environment that they can run their code in and decide that supporting the platform only requires gluing their existing code to the hooks that Android exposes.  And that's true--if your only goal is to port an existing game from one platform to another, only the minimal set of common functionality is necessary to get something up and running.  
  
But since I am in a position to write games exclusively for Android, I've been thinking about how to leverage parts of the platform that most game developers ignore: the OS and Java runtime itself.  There's a lot of functionality there, and maybe there are ways that I could leverage it to make better games.  
  
One project I've been working on recently is a little game framework using the NDK.  My friend Gregg and I ported Google's open source browser-based 3D framework, O3D, to Android a while back, and I've been using that to get some dudes running around on the screen.  O3D has a big Javascript component which we've ignored; the rest of it is a C++-based, shader-centric rendering backend.  Gregg did the heavy lifting of getting the thing to run on OpenGL ES 2.0 and I've been hacking in bits and pieces of old game engines on top.  The result is that we have a pretty complete rendering engine running on Android without a whole lot of effort.  
  
It's a lot of code considering that it doesn't really do anything yet--almost 500k lines of C/C++.  But it wasn't hard to port because in the end, Android is really just another Linux OS with hooks into things like OpenGL ES 2.0.  So for this work, we basically did what lots of other game developers do: we ported the code using as little Android-specific stuff as possible and got something up pretty fast.  
  
I've been slowly adding game code to this project, and as of this writing I have an early prototype of a shooting game up and running: you can run a little test character around and shoot placeholder art zombies with dual thumbsticks.  It's not a game, yet, but it's enough to prove out the code.

|  |
| --- |
| <http://replicaisland.net/img/blog/zombies.png> |
| Not a game, yet.  Place holder art courtesy of [3drt.com](http://3drt.com/). |

If this thing ever gets off the ground, it'll be a game written almost entirely in C++, with just a few hooks back to Java for input, sound, and application life cycle events.  Just like most games that are built to be cross platform, or brought over from other platforms.

But I think there's an opportunity to use Android's unique hybrid application structure to do things that might be difficult or impossible on other platforms.  There are areas where I can get a lot of value out of Java while leaving the game engine and performance-critical code all in C++.

For example, I've hooked up a web server to this game.  It's a very, very simple web server; I found some code on the web that implemented a basic HTTP server in Java, copied and pasted it, and then hacked it up until it did what I needed.  It runs in a separate thread within the main game process, and allows us to connect to the device from a desktop browser while the game is running.  Here's a graphic to illustrate the structure of the code.

|  |
| --- |
| <http://replicaisland.net/img/blog/o3dlayout.png> |
| The high-level structure of this engine.  Red bits are Android Framework, blue are separate threads, and green is native code. |

I'm sure you're reading this and are thinking, *why the heck would you want to run a web server inside a game?!*  Well, sir, I'll tell you.  With the web server in place, I've opened the door to real-time game editing.  This web server doesn't serve static pages, it reads and writes data directly to and from the native engine.  I can, for example, pipe O3D's scene graph up to the web server and let the user browse its structure from their browser.  I can do that with my game objects too (thanks to the meta system I referenced in the last post, which lets me query the structure of a given object by string).  And perhaps most useful, I implemented a simple interface for editing shader code on the fly; I can write vertex and fragment shaders right in the browser, click a button, and immediately see the rendering change in the running game.  
  
This obviously isn't a full runtime editor, but with just a little bit of effort it's already proved to be pretty powerful.  The whole thing is exceedingly simple: my copy-pasted web browser calls down into the native code via JNI and just passes a string payload, which a few hundred lines of runtime code process and then return to the server and thus to the browser.  I'll extend this interface as necessary to other aspects of the game; building a way to do very fast iteration for things like game play physics and shaders is the way to turn a mediocre game into a good one.  
  
Despite the simplicity of the web server system, I'm not sure it would have been as successful on other platforms.  C++ is great for rendering a 3D shader-based game, but it's actually a bit arduous to use for building a web server.  Java, on the other hand, is a great language to write a web server in--it's actually designed with that kind of application in mind.  Android hybrid apps let you leverage both native code and Java simultaneously, which can lead to some pretty neat combinations.  I think that, if this particular game engine ever becomes a full-fledged game, this kind of language diversity will make it a lot of fun to build.  
  
**Update**: Oh, internet, you fickle beast.  Every potentially disputable line of text must be disputed!  
  
OK, to be clear: of course it's not very difficult to write a web server in C or C++.  I did not mean to offend your sensitive language fanboyism by suggesting that maybe, perhaps, possibly, some languages are more predisposed to certain types of work than others.  Though I could write a GLES 2.0 game entirely in Java, I would not choose to do so: that language is not the best fit for that problem.  So yes, you may of course write a web server in C++, or in C, or in assembler or any other language.  And it's not that hard.  But in Java, it's so, so easy.  Heck, I even implemented a memory file cache just for the heck of it.  The code generates JSON on the fly based on results coming back from the engine runtime.  Sure, you could do this in C.  Be my guest.  Me, I'm looking for the simplest possible solution to each of my problems, so I can spend most of my time on the part that counts: making the game fun.  
  
I also did not mean to suggest that I am the first to think of piping game data through a web server.  I just thought it was a neat and easy method for this project specifically on Android.  So there.

POSTED BY [CHRIS](http://www.blogger.com/profile/16079744366139636224)AT [12:27 AM](http://replicaisland.blogspot.de/2010/11/leveraging-java-and-c-for-hybrid-games.html) [12 COMMENTS:](http://replicaisland.blogspot.de/2010/11/leveraging-java-and-c-for-hybrid-games.html#comment-form)

LABELS: [CODE DESIGN](http://replicaisland.blogspot.de/search/label/code%20design), [TARGETING ANDROID](http://replicaisland.blogspot.de/search/label/targeting%20Android)

## WEDNESDAY, NOVEMBER 3, 2010

### [Game Object Construction Rabbit Hole](http://replicaisland.blogspot.de/2010/11/game-object-construction-rabbit-hole.html)

Today I want to write a little bit about a boring, yet utterly fundamental part of game engine design: spawning game objects.

|  |
| --- |
| [http://3.bp.blogspot.com/_EIf-QLT1Gfk/TNFlvoVDKpI/AAAAAAAAAHw/7wDecM6iOhg/s320/Garrys_Mod_Updates_September.jpg](http://3.bp.blogspot.com/_EIf-QLT1Gfk/TNFlvoVDKpI/AAAAAAAAAHw/7wDecM6iOhg/s1600/Garrys_Mod_Updates_September.jpg) |
| Garry's Mod makes it look easy. |

Say you have a level that contains a player object and an enemy object.  However these objects are represented in memory, you have to allocate them somehow.  And probably register them with some sort of update loop, and maybe load some other data (graphics, sound?) associated with those objects.  There's a little bit of bootstrap to just get the game into a state where the simulation can start.  
  
So, how do you do it?  How do you get that player and enemy up and running?  This is one of those problems that can be as complex as you choose to make it.  
  
Well, I mean, you could write a function that looks like this:

void startUpGame()

{

   spawnPlayer();

   spawnEnemy();

   // done!

}

Sounds good until you get to level 2, which has two enemies.  You could make a spawning function for every single level, I guess.  It would work, but it wouldn't scale, especially if you have lots of objects to spawn.  I think the average Replica Island level has several hundred objects to spawn, so writing one of these for each level would suck hard.  
  
Long, long ago I wrote a game called *Fysko's Playhouse* for Mac OS 6 computers.  If that fact alone doesn't date me, this will: it starred the mascot of a local BBS of the same name.  Anyway, back then I had no idea what I was doing, and so when faced with this problem of how to spawn game objects for different levels in a way that doesn't suck, I hit the code with a hammer and moved on.  My solution back then was to write a level editor (in [HyperCard](http://en.wikipedia.org/wiki/HyperCard)!!) that would output start up functions like the one above.  I could draw a level in the editor, hit a button, and out would pop a bunch of Pascal code which I could then copy and paste into the game.  Great!  
  
Well, not really.  That kind of solution works only for very simple games and only when the programmer is the level designer.  And even then it's kind of crappy.  Move a guy 2 pixels to the left and you have to recompile your code.  
  
Some years later I wrote a Bomberman clone called [Bakudanjin](http://www.dreamdawn.com/bakudanjin/sshots.php).  This time I was a little smarter.  Instead of hard coding my level information, I made a map file describing every level.  The map file was basically just a bunch of indexes with XY locations.  To start the level, I load the file and then walk through each index.  The index maps to a table of function pointers (or, since this was also Pascal, a big-ass switch statement) that call the appropriate spawn functions.  Actually, Replica Island [works this way too](http://code.google.com/p/replicaisland/source/browse/trunk/src/com/replica/replicaisland/GameObjectFactory.java).  
  
And if you just clicked on that link, you can see why this method isn't so hot either: Replica Island only has about 50 different object types and that code is still 6500 lines long.  And a lot of it is copy and paste, because many objects turn out to be structurally similar.  And god forbid you accidentally get your map format indexes out of sync with your runtime indexes; arbitrary enums needing to correctly index into separate arrays is a recipe for bugs.  
  
Still, this is a quick and easy method, and it worked fine for Bakudanjin and Replica Island.  All that code gets thrown out and rewritten when I start a new game, though.  
  
The problem here is that all this bootstrap code is basically code describing data.  I draw a line between "code" and "data" as follows: code is literally programming that instructs the game how to operate.  Data is non-code that is input to the code system.  You feed data into your game and out comes a playable simulation.  Things like enemy placement on a map are subject to lots of iteration and change; the code to actually move an enemy to that location is probably pretty stable and static.  Therefore, the placement information is data and shouldn't live in code, while the runtime for consuming that data is code but can be generic and reused across multiple levels and games.  
  
So in order to write better code and to enable faster iteration and reusability, it's probably a good idea to move more of the information for spawning guys into data.  Moving spawn locations into a file wasn't a bad first step, but we can go further.  
  
What does spawnPlayer() do, anyway?  It probably allocates a bunch of objects, ties them together with pointers, and sets a bunch of parameters on those objects.  Hmm, sounds like something we could represent with just some smarter data.  
  
How about this: we'll make all objects in the world basically the same, make a single spawnObject() function, and then expose a bunch of parameters which we can use to customize the objects and differentiate them.  If we can do that, all we need to do is serialize all the parameters and pass them to spawnObject().  Health = 5, Speed = 10, Sprite = "AngryGorilla.png", etc.

GameObject\* spawnObject(ObjectParams\* params)

{

   GameObject\* object = new GameObject;

   object->setLife(params->life);

   object->setSprite(params->sprite);

   if (params->isThePlayer)

   {

      object->setRespondToControllerInput(true);

   }

   // ...

   return object;

}

OK, that works, but now we have a new problem: it's actually pretty hard to make all our objects exactly the same.  Take the code that reacts to controller input, for example.  That belongs on a player object but on nothing else; with this type of system every single angry gorilla is going to carry that code around, probably turned off with a flag that we serialized.  Or consider what happens when an object is hit.  The player probably wants to get damaged, go into a short invincibility mode, and then go back to normal.  The enemies probably want to get damaged and not be invincible.  If the player dies there's probably some code to cause the Game Over screen to pop up.  Not so for an enemy.  
  
We could make this all the same code and control it with flags, but it's going to become ugly fast.  Maybe if we can refactor our game such that all objects are functionally similar we can ease the pain, but that will make it hard to say, add in little bits of player-specific hacks to improve the feel of game play later.  A better system would be able to generate objects that only contain the code that they need.  If we use the [aggregate object model](http://replicaisland.blogspot.com/2009/09/aggregate-objects-via-components.html) that I often recommend, we could think of this as only inserting relevant components.  In a more traditional object model, we could think about instantiating the object at a particular level of derivation from the base to provide only the necessary set of functionality.  Either way, we're talking about sticking code together with pointers and setting parameters.  Hmm, sounds like data again.  
  
One method I've seen but never tried myself is to define a hard-coded list of object types ("player", "angry gorilla", "banana bomb", etc), and then provide a unique struct of static spawn data related to each type.  For example, the player object would be allocated to contain code for movement based on the controller, and it would read at spawn time a PlayerObjectData struct.

GameObject\* spawnObject(ObjectType type, void\* params)

{

  GameObject\* object = NULL;

  switch (type)

  {

    case TYPE\_PLAYER:

        object = new PlayerObject();

        object->parseParams(static\_cast<PlayerObjectData\*>(params));

        break;

    // ...

  }

  return object;

}

The attractive thing about this method is that you get a chance to control object creation on a per-type basis, but you can also serialize different data for each type, thus avoiding the one-size-fits-all problem.  That should let you move almost all information about this object into data, and just leave this one spawn function in code.  
  
But let's go a step further.  Say we don't want to have to enumerate every object type in code.  If objects are really just collections of data and code, why can't we move the entire structure of a game object into data?  
  
In a language that supports reflection, this shouldn't be too hard.  We can encode an entire object hierarchy, say as XML, and then use it to allocate classes, patch pointers, and set fields.  In a language like Java, we can look up class names by string and instantiate them, and poke into fields to finish constructing objects.  We can imagine some XML that looks like this:

<object name="gorilla" type="com.gorillaboom.gameobject">

  <field name = "mLife">10</field>

  <field name = "mSprite">gorillagraphic</field>

</object>

<object name="gorillagraphic" type="com.gorillaboom.imagefile">AngryGorilla.png</object>

If we write an XML parser to build object trees for us based on this, our entire spawn code can become a single call to that parser.  The parsing code is complicated but generic, it can be reused across objects and levels and games.  And we still have the ability to customize each object because the hierarchy can contain custom classes or be structured differently per type.

GameObject player = (GameObject)constructFromXML("player.xml");

Even cooler, this method isn't even specific to game objects.  We could use it to serialize all sorts of data, even the contents of our main loop.  The code is still code, but once we put the structure in data we have a crazy amount of control.  
  
But, getting back to just the game object application of this idea, there are two problems.  First, this approach requires us to walk the XML object tree every time we want to spawn an object.  We could try to walk the XML once and then use the resulting object tree as an entity "archetype," but that way leads to a particular brand of hell known as "shallow copy vs deep copy." When trying to copy the tree to create a new game object instance, how do you know which pointers should be recursively copied and which should be copied by value?  People have lost years of their life to that problem.  
  
A less practical but ultimately simpler solution is just to re-parse the XML for every instantiation.  Which brings us to the second problem: reading XML and stuff is slow.  I mean, really slow.  Compared to the simple code functions we started with, it's glacial.  And allocation-heavy.  Not something we really want to do during the runtime of a game.  
  
I know what you're going to say.  *Hey, dumbass, just use a binary format instead of XML.  Problem solved.*  And that's true, to an extent.  Excuse me for a moment while I go out on a limb to the extreme of this line of thought.  
  
If you're going to go to a binary format, why not just store the entire object tree in its native memory format on disk?  Build the object hierarchy offline, write it as binary data to a file, then load it at runtime and just patch pointers.  Boom, instant object tree.  In C++ you can actually do this by building your own reflection system, making liberal use of placement new, and then walking the objects stored in the file to patch pointers.   
  
I've actually written this system before.  It becomes horrifically complicated, but it does work.  A couple of years ago I wrote an engine that could load a binary blob of C++ objects as a contiguous block, walk the objects therein to patch vtables and other pointers, and then simply start using the objects as regular fully constructed C++ objects.  You lose constructers (objects were constructed before they were written to disk) and destructors (lifetime of the object is the lifetime of the blob; individual objects within the blob can't be freed), and god help you if you try to manage those objects with reference counts or smart pointers, but the method does work and it's pretty fast.  To make a new instance of the object tree, you can just memcpy the whole block and repatch pointers.  Cool.  
  
It starts to break down when you need to target multiple platforms, or build your data files on an architecture that does not match the runtime architecture.  These problems are also solvable, but probably not in-place; you'll need to read the objects out of the file and then allocate them at runtime to ensure padding and vtable placement is correct.  And if you do that you're back to a lot of runtime allocation and object parsing.  The system is still complicated but some value is lost.  
  
So for a new code base that I'm working on, I'm experimenting with a slightly different approach.  I still want to use the "load code structure from data" approach, but I don't want it to be slow or architecture dependent (or complicated, if I can avoid it).  And I need to be able to spawn objects with this system dynamically at runtime.  So instead of constructing objects directly, I'm constructing factory objects that can generate new instances of my object hierarchy on the fly.  
  
The method is to read in an object hierarchy as XML.  Instead of building the tree right there, I build a bunch of "instruction" objects--basically the minimal list of commands required to recreate the object tree described in XML.  "Create an object of type GameObject," "Set field mLife of object index 5 to 10," "Point field mSprite of object index 22 to object index 25."  Each of these small "patch" objects gets initialized with a single command (representing a delta from the default state of the object upon construction), and the whole list of patches is stored in a factory object I call a "builder."  Reading the XML is still slow, but I only need to do it once before the game starts; at runtime, when I want to spawn a new object tree, I simply execute the appropriate builder.  The runtime speed is similar to what we had way back at the top of this lengthy post: just a bunch of object allocations and field initializations.  Should be pretty fast.

Builder\* enemyBuilder = makeBuilderFromXML("angrygorilla.xml");

GameObject\* enemy1 = enemyBuilder->build();

GameObject\* enemy2 = enemyBuilder->build();

GameObject\* enemy3 = enemyBuilder->build();

One nifty aspect of this approach is that I can easily extend the XML format to do more complicated things by building more complex patch object types.  The basic set of patches (int, float, vector, pointer, string, etc) are entirely generic, as is the builder system itself.  But I can add to that engine- or game-specific patches if necessary.  I've already added a patch that knows about this particular engine's asset system, and can use it to schedule other files for loading, thus allowing for references to external assets which may be loaded in the future (and patched at that time appropriately).  A different game might have an entirely different asset system, in which case I can chuck the one patch written for this game and write a new one against that engine; the system should scale without losing its general purpose core.   
  
The actual XML parser and builder system is very simple--only a couple hundred lines of code.  But I should mention that it only works in C++ because my game engine is backed by a (fairly involved) reflection system.  Using an Interface Definition Language, I can create classes that are laden with meta data, much like Java's Class object.  Using that data I can poke into anonymous classes at runtime and set fields, which is how the patches in the builder actually work.  I think this approach could be done without reflection, but it would basically resemble the hard-coded types with unique static data structs method that I mentioned above.  I'll talk more about the reflection system in a future post.  
  
To bring this giant document to a close, I just want to note that the methods I've described here are hardly an exhaustive list.  These are the various approaches that I've tried to spawn objects in games; there are many others and probably a lot of really good ideas that I've never considered.  But when making a game engine, the question of how objects get spawned--and what a game object actually *is*, is a pretty huge piece.  Though sort of mundane, it's probably worthy of a lot of thought.

POSTED BY [CHRIS](http://www.blogger.com/profile/16079744366139636224)AT [6:56 AM](http://replicaisland.blogspot.de/2010/11/game-object-construction-rabbit-hole.html) [8 COMMENTS:](http://replicaisland.blogspot.de/2010/11/game-object-construction-rabbit-hole.html#comment-form)

LABELS: [CODE DESIGN](http://replicaisland.blogspot.de/search/label/code%20design)