15

Runtime Compiled C++ for Rapid AI Development

Doug Binks, Matthew Jack, and Will Wilson

15.1 Introduction
15.2 Alternative Approaches
15.3 Runtime Compiled C++
15.4 Runtime Compiled C++ Implementation
15.5 Runtime Compilation in Practice—a Step-by-Step Example
15.6 Runtime Error Recovery
15.7 More Complex Code-State Preservation and Header File Handling
15.8 Code Optimizations for Performance-Critical Code
15.9 Use Case: A Behavior Tree and Blackboard Architecture
15.10 Crytek Case Study
15.11 Future Work
15.12 Conclusion

15.1 Introduction

Scripting languages have always been a foundation of rapid AI development but with the increasing demands of AI, their performance drawbacks are becoming ever more problematic. On the other hand, traditional C++ development approaches generally lead to lengthy compile and link times, which limit the amount of iteration and testing that programmers can undertake. Though development tools are progressing in this area, developers still need to run the build and load content to see the end result, and edit-and-continue style approaches do not work for all codebases or changes.

In this article we demonstrate how the same fast iteration times and error-handling can be achieved using pure C++ through a novel approach, which we call Runtime

Compiled C++ (RCC++). RCC++ allows developers to change code while the game is running, and have the code compiled and linked into the game rapidly with state preserved. Programmers can thus get feedback on their changes in seconds rather than minutes or more. The technique has been used in the development of AAA games at Crytek, and gives similar results to Hot Reload as seen in *Epic's Unreal Engine 4*. The RCC++ code is available as a permissively licensed open source project on GitHub and the accompanying disk [Jack, Binks, Rutkowski 11].

15.2 Alternative Approaches

A variety of alternative approaches can be used to achieve fast iteration times in games, each with their own pros and cons. Overall we believe that none offer the main benefits of Runtime Compiled C++, and many techniques can be enhanced by using RCC++ alongside them.

15.2.1 Scripting Languages

Scripting languages provide perhaps the most common solution to achieving fast iteration. A recent games engine survey showed Lua to be the foremost scripting language in game development [DeLoura 09], though UnrealScript [Epic 12a] likely represents a substantial proportion of the market, given the size of the Unreal Engine community. Despite the popularity of scripting languages, we feel that they have a number of issues when used for core features, namely integration overheads, performance, tools and debugging, and low level support such as vector operations and multithreading. In fact, many of the game engine implementations we have encountered do not permit runtime editing of game scripts, which reduces the benefit of using them considerably.

Games written in C++ need an interface to the scripting language in order to allow features to be implemented. A variety of techniques and methods are available to make this relatively simple, but a recompile and game reload will be forced when an interface change is required.

While script performance is improving, it is still well below that of compiled code such as C++ [Fulgham 12]. Indeed, Facebook moved their PHP script code to pre-compiled C++ using the open source cross compiler HipHop, which they developed to address performance issues. They found significant benefits in terms of reduced CPU overhead and thus expenditure and maintenance of systems [Zao 10]. Garbage collection can also be an issue, with optimized systems even spending several milliseconds per frame [Shaw 10]. On consoles, Data Execution Protection (DEP) prevents VMs from Just-In-Time (JIT) compiling, limiting an opportunity for performance optimizations offered by cutting-edge VMs running on PC.

Debugging a C++ application that enters a script VM can prove difficult to impossible, and many of the tools developers use on a day-to-day basis, such as performance and memory profilers, also may not have visibility of the internals of a VM in a way that can be traced back to the original source code.

C++ provides easy integration with native instruction sets via assembly language, as well as direct access to memory. This low level support permits developers to target enhanced, yet common, feature sets such as SIMD vector units, multithreading, and coprocessor units such as the Sony PS3's SPUs.

Havok Script (originally Kore Script) was developed out of the motivation of solving some of these issues [Havok 12], providing an optimized VM along with performance improvements, debugger support, and profiling tools.

15.2.2 Visual Studio Edit and Continue

Edit and Continue is a standard feature of Visual Studio for C++ introduced in Visual C++ 6.0, allowing small changes to code to be made while debugging through a technique known as "code patching" [Staheli 98]. However, there are some major limitations, the most critical of which are that you can't make changes to a data type that affect the layout of an object (such as data members of a class), and you can't make changes to optimized code [Microsoft 10].

15.2.3 Multiple Processes

Multiple processes can be used to separate code and state so as to speed compile and load times, with either shared memory, pipes, or more commonly TCP/IP used to communicate between them. This is the fundamental approach used by many console developers with the game running on a console and the editor on a PC, and as used by client/server style multiplayer games. Support for recompiling and reloading of one of the processes can be implemented reasonably simply if the majority of the state lies in the other process, and since there is a clean separation between processes, this can be a convenient approach to take. However, the communication overhead usually limits this to certain applications such as being able to modify the editor, while having the game and its game state still running while the editor process is recompiled and reloaded. Turnaround times can still be relatively long since the executable needs to be compiled, linked, and run.

15.2.4 Data-Driven Approaches

Many game engines provide data-driven systems to enhance the flexibility of their engine, for example the *CryENGINE 3* uses XML files to specify AI behavior selection trees [Crytek 2012]. Since the underlying functionality can be written in C++, this methodology can overcome many of the issues with scripting languages, while providing designers not fluent in programming languages a safe and relatively easy to use environment for developing gameplay. While frameworks such as data-driven behavior trees can allow a good balance to be struck, this approach can descend into implementing a scripting language in XML, with all the associated problems.

15.2.5 Visual Scripting

Visual Scripting systems such as Epic Games' *Unreal Kismet* and Crytek's *CryENGINE* Flow-Graph provide a graphical user interface to game logic [Epic 12b, Crytek 12]. Functionality blocks are typically written in C++ with inputs, outputs, and data members that can be graphically edited and linked to other functional blocks, thus providing the potential for complex systems to be created. This is fundamentally a data-driven system with an enhanced user interface.

15.2.6 Library Reloading

Dynamic link libraries (DLLs, called "shared objects" on Unix variants) can be loaded during runtime, providing an obvious means to allow fast iteration by recompiling and

reloading the DLL. A key feature for game developers is that both the Xbox 360 and PS3 consoles support dynamically loaded libraries, as do iOS (for development purposes only—not in released code) and Android. Function and object pointers need to be patched up, and the interfaces to the DLL can't be changed. If the code for the DLL project is large, compile times can be significant limiting the iteration turnaround. Despite the attraction of this approach, in practice the difficulty of splitting up and maintaining a game with many small-sized DLLs, along with the infrastructure required for delay load linking and state preservation, means this approach is rarely used.

15.3 Runtime Compiled C++

Runtime Compiled C++ permits the alteration of compiled C++ code while your game is running. RCC++ uses DLLs, but rather than building an entire project, the runtime compiler only rebuilds and links the minimal source files required. By using loose coupling techniques, dependencies are kept to a minimum. The resulting dynamic library is loaded, and game state is saved and then restored with the new code now being used. Changes to C++ code are thus possible during a gameplay or editor session with a turnaround time of several seconds. Developers do not need to manage multiple project configurations, as the DLLs are constructed automatically on the fly as required. Indeed, the codebase can be built as a single executable. RCC++ can be seen in action in Figure 15.1.

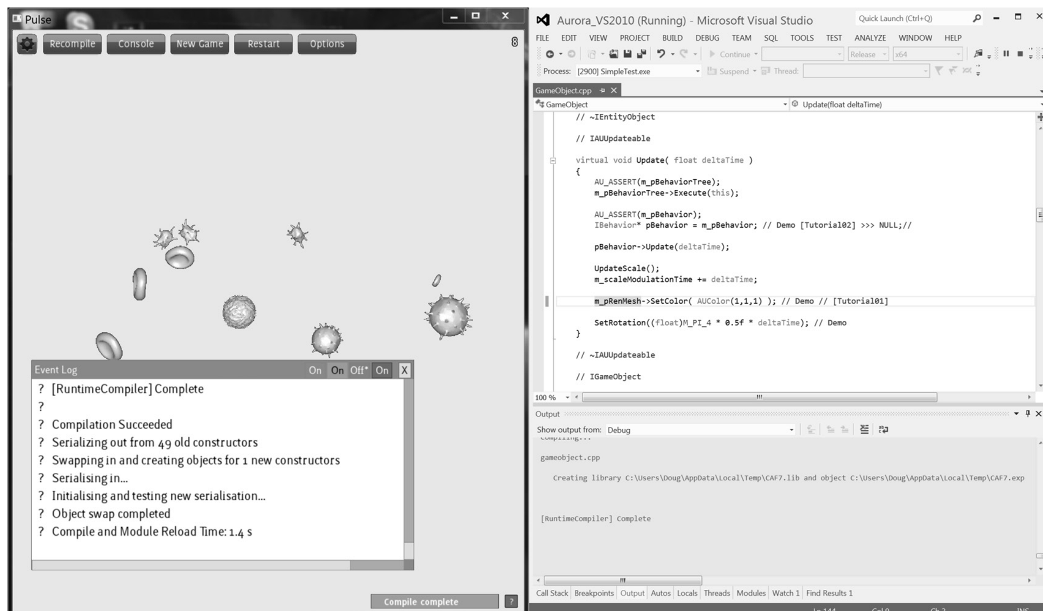

Figure 15.1

The demo game *Pulse* compiling files at runtime after a change, with Visual Studio on the right being used to make the changes. Compiler output goes to both demo event log and Visual Studio Output window to enable navigation to errors with double clicking. Time taken to compile and load changed code is 1.4s as shown by event log in this instance.

Unlike Edit and Continue, RCC++ permits changes to object layout including the addition of entirely new classes, and it works with optimized code as well as debug. Since we're dealing with compiled C++, we enjoy the full feature set this gives along with the ability to use intrinsic functions which access instruction set features like SIMD and atomics, along with OS level functionality such as multithreading. By using one language for development we can use all the tools we have at our disposal, from seamless debugging to performance profilers. Intriguingly, both fixing bugs and optimizing code can now be done on the fly.

We believe that this approach is well suited to replacing scripting as used by programmers for fast iteration, and that designer tools such as data-driven systems and visual scripting languages can be easily enhanced through the use of RCC++. Other developers appear to agree; for example, Epic has dropped UnrealScript entirely in *Unreal Engine 4* by switching to an approach similar in results to RCC++ called Hot Reload. Flexibility for designers is preserved through improvements to the Kismet Visual Scripting system [EPIC 12c].

15.4 Runtime Compiled C++ Implementation

The Runtime Compiled C++ implementation consists of two main components: the Runtime Compiler and the Runtime Object System. The Runtime Compiler handles file change notification and compilation. The Runtime Object System handles tracking of source files and their dependencies. It also provides the `IObject` base class interface and `IObjectFactorySystem` for creating objects and swapping them when new code is loaded. This separation is intended to allow developers to create their own dynamic reloading code while keeping the compiler functionality. Additionally, for console and mobile games, the Runtime Compiler can more easily be moved to a separate process running on the host system.

15.4.1 The Runtime Compiler

The Runtime Compiler provides a simple C++ interface to a compiler. Currently Visual Studio 8.0, 9.0, and 10.0 compilers are supported (Visual Studio 2005, 2008, and 2010) in both the free Express and full versions, and both x86 and x64 targets can be built.

A primary goal of the compiler is to provide rapid turnaround. Compiles are run from a command line process which is instantiated on initialization of the compiler and preserved between compiles. We found that setting up the environment variables for compiles took a substantial amount of time, so by setting these only once and keeping the process active we cut down the time of small compiles significantly. A simple compiler logging interface is provided that can be implemented and passed to the compiler to get compilation output. A handy trick on Windows is to output this using the `OutputDebugString()` function in addition to any other logging; with Visual Studio any errors or warnings can then be doubled clicked to navigate to the offending source.

The Runtime Compiler monitors files that have been registered with the system by the Runtime Object System, and when changes are detected it compiles these files and any dependencies into a DLL module, which is then loaded by the Runtime Object System.

15.4.2 The Runtime Object System

The Runtime Object System provides the functionality needed to have code changes detected, compiled with the correct dependencies, and for the resulting module to be loaded and objects switched over to using the new code. To make compilation as fast as possible, the minimal number of dependencies is compiled.

The runtime object system relies on the C++ virtual function feature to be able to swap objects by simply swapping pointers from the old object to the new object. If the functions were non-virtual, the function pointer would be embedded in any code that calls the function, so the old code would always be run. A simple runtime-modifiable class could be comprised of a header declaring the class as follows:

Listing 15.1. An example runtime object, deriving from IObject.

```
class MyRuntimeModifiableObject : public IObject
{
public:
    virtual void DoSomething();
};
```

In order to be able to make changes at runtime we need to expose this class via a factory style constructor. This is done by applying the macro `REGISTERCLASS(MyRuntimeModifiableObject)` in the.cpp file, which adds the path of the source file to a list that the Runtime Compiler monitors. When this source file is changed, we compile the source file along with the common Runtime Object System interface code into a DLL and then load the resulting module. Even if the project contains many source files only two need to be compiled and linked into the module in the minimum case. After module load we now have two or more classes called `MyRuntimeModifiableObject` along with their respective constructors, which is possible as each is in a different module (for example, one is in the executable and another in a DLL). The Runtime Object System can be used to create a new set of objects to replace old ones using the new constructor, and their pointers swapped so that code now references the new class.

We considered several techniques for achieving the runtime swapping of pointers. The two main candidates were a smart pointer, which added one further level of indirection through a pointer table, allowing object pointer swapping to occur at extremely low cost, and serialization of all objects with pointer replacement occurring using an object ID that offered low runtime overhead but a higher cost to swap pointers. In order to preserve object state when changing code we need to have serialization in place anyway, and the reduced runtime overhead seemed a win for serialization. Since the state only needs to be written and read from memory, this serialization can be extremely fast. For non runtime modifiable code, an event system permits developers to use the Object ID to swap object pointers when new code is loaded.

Using virtual functions introduces one level of indirection, so developers of performance-critical areas of code should look at the Section 15.8 on Code Optimizations for

Listing 15.2. Accessing functionality via a system table.

```
//Interface to Runtime Object System
struct IRuntimeObjectSystem
{
public:
    ...
    virtual void CompileAll(bool bForceRecompile) = 0;
    ...
}
//System table to access engine functionality
struct SystemTable
{
    IRuntimeObjectSystem * pRuntimeObjectSystem;
    //other interface pointers can be added here
};
//Example usuage in a GUI button event:
class OnClickCompile : public IGUIEventListener
{
public:
    virtual void OnEvent(int event_id, const IGUIEvent& event_info)
    {
        SystemTable* pSystemTable = PerModuleInterface::GetInstance()->
GetSystemTable();
        pSystemTable->pRuntimeObjectSystem->CompileAll(true);
    }
};
```

suggestions. Note that the use of virtual functions does not imply that the developer needs to implement a complex polymorphic class system with deep inheritance trees.

In order for runtime code to access features of nonruntime modifiable code, we provide a mechanism for passing a System Table object pointer to runtime modifiable code. This System Table can then be used to store pointers to interfaces of the game engine's subsystems, such as rendering, audio, physics, etc. In Listing 15.2, we make accessible the Runtime Object System subsystem itself to allow the user to force a full recompile. This example is taken from the Pulse demo in the code sample included with this book (called SimpleTest in the code base).

The overall architecture of the Runtime Object System is best illustrated by looking at a full example of its application, which we'll do in the next section.

15.5 Runtime Compilation in Practice—a Step-by-Step Example

In this example we'll look at implementing a simple Win32 console application which runs a main loop every second, calling an update function on a runtime object. In the following listings we've removed some of the implementation so as to focus on the main elements. Once again see the included source code for full details, in the ConsoleExample project.

The main entry point of the project simply constructs a `ConsoleGame` object, calls its `Init()` function, and then calls the `MainLoop()` function in a loop until this returns

Listing 15.3. The `ConsoleGame` class declaration in ConsoleGame.h.

```
class ConsoleGame : public IObjectFactoryListener
{
public:
    ConsoleGame();
    virtual ~ConsoleGame();
    bool Init();
    bool MainLoop();
    virtual void OnConstructorsAdded();
private:
    //Runtime Systems
    ICompilerLogger*      m_pCompilerLogger;
    IRuntimeObjectSystem*  m_pRuntimeObjectSystem;
    //Runtime object
    IUpdateable*  m_pUpdateable;
    ObjectId      m_ObjectId;
};
```

Listing 15.4. ConsoleGame initialization. The string "`RuntimeObject01`" refers to the class name of the object we wish to construct.

```
bool ConsoleGame::Init()
{
    //Initialize the RuntimeObjectSystem
    m_pRuntimeObjectSystem = new RuntimeObjectSystem;
    m_pCompilerLogger = new StudioLogSystem();
    m_pRuntimeObjectSystem->Initialize(m_pCompilerLogger, 0);
m_pRuntimeObjectSystem->GetObjectFactorySystem()->AddListener(this);
    //construct first object
    IObjectConstructor* pCtor =
m_pRuntimeObjectSystem->GetObjectFactorySystem()->
GetConstructor("RuntimeObject01");
    if(pCtor)
    {
        IObject* pObj = pCtor->Construct();
        pObj->GetInterface(&m_pUpdateable);
        if(0 == m_pUpdateable)
        {
            delete pObj;
            return false;
        }
        m_ObjectId = pObj->GetObjectId();
    }
    return true;
}
```

false. The `ConsoleGame` class, which derives from `IObjectFactoryListener`, receives an event call when a new constructor has been added after a code change, allowing us to swap the runtime object pointers using an ID lookup. The data member `m_pUpdateable` is where we store the native pointer, while `m_ObjectId` stores the id of the runtime object.

ConsoleGame.cpp contains the fundamental aspects required for a RCC++ implementation. The initialization requires creating a logging system (which simply writes the log output to `stdout`), and the Runtime Object System, followed by adding the `ConsoleGame` as a listener for events and then creating our runtime object—in this case, a class called simply `RuntimeObject01`.

When new code is loaded, the `IObjectFactoryListener::OnConstructorsAdded()` method is called, and we use this to swap our object pointer to the new code as in the listing below.

Listing 15.5. Swapping runtime object pointers after new code is loaded.

```
void ConsoleGame::OnConstructorsAdded()
{
    if(m_pUpdateable)
    {
        IObject* pObj =
m_pRuntimeObjectSystem->GetObjectFactorySystem()->GetObject(m_ObjectId);
        pObj->GetInterface(&m_pUpdateable);
    }
}
```

The main loop first checks for compilation completion, and loads a new module if a compile has just completed. We then update the file change notification system and our runtime object through the `IUpdateable` interface.

If we want to change the class declaration at runtime, a separate virtual interface is needed for nonruntime modifiable code. This interface shouldn't be modified at runtime, though header tracking allows the interfaces between different runtime source files to be changed. In practice, the more code that is moved to being runtime compiled, the less restrictive this becomes. Our console example has a simple interface as in Listing 15.6; this defines the `Update()` function and also derives from `IObject`.

Listing 15.6. The update function is declared as an abstract interface.

```
struct IUpdateable : public IObject
{
    virtual void Update(float deltaTime) = 0;
};
```

Listing 15.7. The runtime object. This code can be modified while the console example is running, with saving the file out causing compilation and reloading of the code.

```
class RuntimeObject01 : public TInterface<IID_IUPDATEABLE,IUpdateable>
{
public:
    virtual void Update(float deltaTime)
    {
        std::cout << "Runtime Object 01 update called!\n";
    }
};
REGISTERCLASS(RuntimeObject01);
```

The runtime object code is correspondingly simple. The class, defined in Listing 15.7, derives from a template which implements the `GetInterface()` member function, which in this case allows us to get hold of a pointer to an `IUpdateable*` using the interface ID, `IID_IUPDATEABLE`.

Putting this together, and running the sample, we're able to change the `Update()` function and save out, seeing our changes in action in the console. Listing 15.8 shows example output from a session where we've added "NEW!" to the output.

15.6 Runtime Error Recovery

We felt that for RCC++ to be really practical, it had to have a form of crash protection. After all, it would be a frustrating experience if you could avoid quitting, recompiling, reloading only until your first simple mistake, when a null pointer forces you to close it all down and start over. We looked at two main approaches to achieving this: using a separate process, and structured exception handling.

Google Chrome uses the process approach for each of its tabs, allowing one to crash without affecting any of the others. This is a very robust approach, but unless your engine's architecture has this in mind, it may result in a very large number of interprocess function calls, which would have severe performance impact. We wanted an approach that would be easy to drop into existing projects.

Structured exception handling (SEH) is a Win32 API feature that allows handling of runtime errors such as access violations. It behaves much like standard exceptions but is in fact quite separate; there are various reasons why standard exceptions are not used on games consoles, but these don't affect SEH. When a runtime error such as a null pointer dereference occurs, the OS checks a stack of possible handlers registered by the application to see how to proceed. The crash dialogs you see in Windows are, in fact, the default handler; when Visual Studio's debugger is attached, that adds another.

Using SEH it is quite easy to catch an error and carry straight on. In our case, the key place for this is around the update calls on our game objects. When an update fails, we disable it until the code has been runtime-recompiled, then we try again. During this process the rest of the application—rendering, GUI, logging—all keep running.

But actually, you don't want to handle a crash silently—you'd really like to find out what caused it first! You could add code to produce a stack trace. However, debuggers like

Listing 15.8. The console example compiling files at runtime after a change. Note the added 'NEW!' to the output of the update function.

```
Main Loop - press q to quit. Updates every second.
Runtime Object 01 update called!

Main Loop - press q to quit. Updates every second.
Runtime Object 01 update called!

Main Loop - press q to quit. Updates every second.
FileChangeNotifier triggered recompile of files:
Compiling...
Created intermediate folder "Runtime"
cl /nologo /O2 /LD /Zi /MP /Fo"Runtime\\" /D WIN32 /EHa /FeC:\Temp\
BFCB.tmp "e:\aurora\examples\consoleexample\runtimeobject01.cpp"
"e:\aurora\runtimeobjectsystem\objectinterfacepermodulesource.cpp"
echo _COMPLETION_TOKEN_
Runtime Object 01 update called!
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

E:\Aurora\Examples\ConsoleExample>
Setting environment for using Microsoft Visual Studio 2010 x64 tools.

runtimeobject01.cpp
objectinterfacepermodulesource.cpp
    Creating library C:\Temp\BFCB.lib and object C:\Temp\BFCB.exp

[RuntimeCompiler] Complete
'ConsoleExample.exe' (Win32): Loaded 'C:\Temp\BFCB.tmp'. Symbols loaded.
Compilation Succeeded
Serializing out from 1 old constructors
Swapping in and creating objects for 1 new constructors
Serialising in...
Initialising and testing new serialisation...
Object swap completed

Main Loop - press q to quit. Updates every second.
NEW! Runtime Object 01 update called!

Main Loop - press q to quit. Updates every second.
NEW! Runtime Object 01 update called!
```

that in Visual Studio already provide a really ideal interface when crashes occur, allowing you to inspect state easily. So really, we would like to crash, but then continue.

In fact, this is exactly what we do; we first allow the crash to proceed as normal so we can use Visual Studio to debug it, after which the user can hit the "continue" button in the IDE. Usually the "continue" option is quite useless but in our case we catch the crash on the second attempt—and proceed with execution. We get the best of both.

As an aid to productivity, we include a runtime assert macro that is compatible with the runtime exception handling process.

15.6.1 Handling Errors on Loading New Code

When new code is loaded (see the `ObjectFactorySystem::AddConstructors` function in the prototype), we need to construct the new objects, serialize state into them,

and initialize them (if required). Since the process is reliant on working serialization, we also test serializing out from the newly constructed objects. Structured exceptions (crashes) could occur during this process, so we use the runtime exception filter to handle them, and revert to running the old code and state. This gives the developer an opportunity to fix the problem and reload the new code without restarting the application.

Should an error be caught, the new objects that were constructed are deleted. This introduces a new failure path in the destructor, so this also uses runtime exception handling and simply leaks the objects when an error occurs.

Another issue which can commonly occur is that the new code pollutes the data during this process in such a way as to not crash, but to render the application instance unusable. A trivial solution, which has not yet been implemented in our codebase, would be to maintain an undo list of code and data to allow developers to revert back to a previously working state and make further changes from there.

Further stability could be added by unit testing new code prior to loading, and/or creating a save point on disk. If developers intend to use the Runtime Compiled C++ approach in an editing tool we would recommend adding these features, but for a programming productivity tool we believe these steps would be counterproductive due to the extra delay in turnaround.

15.7 More Complex Code-State Preservation and Header File Handling

When swapping new code for old, it is desirable to ensure that the state of the old object is propagated to the new one. The Runtime Object System includes a simple serialization system designed for the needs of runtime modification of code.

The `SERIALIZE` macro can be used for any type that implements the `operator =` method, which also permits types to be changed to compatible types, such as `float` to `double` or `int` etc. Non runtime pointers can be serialized by their value since the address doesn't change, but runtime pointers are serialized through their `ObjectId`.

Listing 15.9. An example serialization function.

```
virtual void Serialize(ISimpleSerializer *pSerializer)
{
    IBaseClass::Serialize(pSerializer);
    //any type which implements operator = including pointers
    SERIALIZE(m_SomeType);
    //serialize a runtime object pointer
    SERIALIZEIOBJPTR(m_pIObjectPointer);
    if (pSerializer->IsLoading()) {
        //do something only when loading values
    }
    //serialize something whose name has changed
    pSerializer->Serialize("m_Color", m_Colour);
}
```

To save developer time, the same serialization function is called on both load and save of state. For situations where different behavior is needed, the `IsLoading()` function can be used.

Values are serialized by name, so if the name of the object changes the developer should use the `Serialize` function directly rather than the macro, and replace with the macro once it's been loaded.

A simple header file dependency tracking method has been implemented. To use this, add the `RUNTIME_MODIFIABLE_INCLUDE` to the header file (defined in `RuntimeInclude.h`), and when this file changes any code which includes it and has a runtime modifiable class will be recompiled. This macro expands to a recursive template specialization that, when such headers are included from a.cpp file, uses the _COUNTER_ macro to assign successive numbers to the filenames of each. This enables the system to iterate through all of the include files for a given class.

15.8 Code Optimizations for Performance-Critical Code

We use virtual functions to give runtime redirection of function calls when new code is loaded. This adds a small performance penalty to the function call. For many AI and scripting style use cases, this is still equal or better performance than alternatives such as running a script VM or using a data-driven approach. However, some simple approaches can reduce or eliminate this penalty.

For developers who only use the virtual functions for runtime compiling of code, it is possible to write a macro for final release builds which declares the appropriate functions to be nonvirtual, so long as polymorphic behavior is not required. This introduces a potentially substantial difference between builds, so a strong test regime is required.

Listing 15.10. Optimization for final release builds by removing virtual from performance-critical functions. Note that we don't declare an abstract interface, so the developer must take care to ensure they only use virtual or RUNTIME_VIRTUAL functions in nonruntime code.

```
#ifdef RUNTIME_COMPILED
#define RUNTIME_VIRTUAL virtual
#else
#define RUNTIME_VIRTUAL
#endif
class SomeClass : public Tinterface<IID_ISOMECLASS,IObject>
{
public:
    virtual void SomeVirtualFunction();
    RUNTIME_VIRTUAL void OnlyVirtualForRuntimeCompile();
private:
    //members which are not virtual or RUNTIME_VIRTUAL
};
```

A simpler alternative is to base the high performance areas of the code around aggregated calls, similar to the approaches taken in data oriented programming. For example, if we consider the following function performing operations on a game object:

Listing 15.11. Simple virtual function example declaration and definition.

```
virtual void Execute(IGameObject* pObject)
{
    //perform actions on one game object
}
```

If we have many calls to execute in our game the virtual call overhead may become significant. So we could replace this by aggregating all calls as follows:

Listing 15.12. Redesigned function to reduce virtual function call overhead and improve cache coherency by processing multiple objects per call.

```
virtual void Execute(IGameObject* pObjects, size_t numObjects)
{
    //perform actions on many game objects
}
```

Here, we've defined this based on a new interface that requires the developer to pass in an array of game objects to process, and so have reduced the cost of the virtual function call to `1/numObjects` times the original cost. Additionally there are potential benefits from cache coherency, which could give even further performance benefits.

15.9 Use Case: A Behavior Tree and Blackboard Architecture

The ability to change code freely at runtime doesn't just speed up iteration on existing workflow. It can also allow us to approach problems in an entirely different manner. Some of the first techniques we reevaluated this way were behavior trees and blackboards.

Bastions of modern game AI, behavior trees provide a framework for breaking down complex AI into modular pieces with structured reactive decision making, and have been described widely, including in this very book. Blackboards are a simple approach to sharing data between aspects of an AI while remaining loosely coupled. Often the two are used together, with conditionals within the behavior tree reading primarily from the blackboard, while that data is updated regularly from sensory systems and written to by the operations of the behaviors themselves.

15.9.1 A "Soft-Coded" Behavior Tree

Here, we take as an example a "first-generation" behavior tree—essentially a decision tree with behavior states as its leaves, a very common design in modern games including *Crysis 2*.

Implementations have included the use of snippets of Lua to form flexible conditions at each node, or in *Crysis 2* a simple virtual machine executing a tree specified by XML [Martins 11]. However, in essence, the structure is a simple tree of if-else statements—the rest of the behavior tree architecture deriving from the requirement for rapid iteration, as a set of hardcoded C++ if-else clauses would bring AI development to a near halt.

We can now reevaluate that assumption. Under RCC++ such decision code can be thought of as "soft"-coded—we can change it at will while the game is running. So, assuming some nicely formatted code, let's consider its properties: being raw C++, it is obviously extremely fast, it will handle errors without crashing, it has easy access to all the state in our AI system, it can share sub-trees as function calls, it can use all our C++ math and utility routines, we can make use of our existing debugger, we can apply our existing profiler if we need to, you can see useful diffs from source control … the list goes on. In a few lines of code you've implemented a behavior tree—in fact, an unusually fast and powerful behavior tree.

The main aspect for improvement is that of the interface for designers. Simple parameters may, of course, be exposed through XML or a GUI. In this author's experience more structural changes are best kept the responsibility of AI programmers, who often find a simple text specification better to work with than a graphical one. However, such if-else code may be very easily generated from a graphical representation should one be required, with the simple act of replacing the source file triggering runtime-compilation.

15.9.2 A Blackboard

Blackboards may be naturally represented in languages such as Lua or Python as a table of key-value pairs, comprising strings and dynamically typed values. While similar constructions are possible in C++, they represent a possible performance bottleneck. In an analogous approach to our behavior tree, we can represent our blackboards under RCC++ as simple structs—each key-value pair becoming a named member variable of appropriate type.

Reading and writing from the blackboard becomes as simple as passing its pointer to our sensory systems and to our RCC++ behavior tree. The main difficulty we must resolve is that of dependencies, as all the code that uses it must include the struct definition as a header, and must be recompiled should it change.

The header-tracking feature of RCC++ makes this simple, with a change to the include file triggering recompilation of all the dependent code. We advise using a hierarchical form of blackboards with, for example, a particular agent's full blackboard inheriting from one shared blackboard across all agents, and one shared by its species, and with separate access to any blackboard used solely by the current behavior. This approach helps ensure that only relevant AI code needs to be recompiled when the blackboard definition is changed, keeping recompiling as efficient as possible.

15.10 Crytek Case Study

Following a presentation at the Paris Game/AI conference in 2011, Crytek implemented a similar system internally which they call SoftCode. The system has been used on several game projects currently active in Crytek and has been used for systems as diverse as render

effects, UI, and animation. It is most heavily employed currently on the AI system in use for *Ryse* for authoring behaviors and managing behavior selection.

SoftCode builds upon the approach of RCC++ in several interesting directions. The first is that it provides the compilation and change tracking functionality of the runtime compiler as a Visual Studio 2010 add-in, which works for 32-bit and 64-bit Windows builds as well as Xbox 360. Additionally, it generalizes the concept of the runtime object system, allowing developers to expose their own system through a type library. Thus, rather than having to derive from a single `IObject` base class, each SoftCoded type can derive from their own base interface. Like RCC++, SoftCoding has an error-handling mechanism using structured exceptions, but its lambda functions allow call state capture so that individual methods can be retried after failure. To simplify development, SoftCode types use a `SOFT()` macro to expose class member variables rather than through an explicit Serialize function, and globals can be exposed through an `SC_API` macro.

15.11 Future Work

With Mac and even Linux becoming more popular targets for games, we feel support for these platforms is a natural next step for the project. Beyond this, moving to a client/server model for the Runtime Object System and Runtime Compiler would permit further targets such as Consoles and mobile devices to be targeted.

An interesting further step would be to support a simpler interface to programming than C++ with an intermediate compiler that generates C++ code, using RCC++ to permit runtime usage. A basic graphical interface to construct object data members and function declarations with function bodies editable in a variant of C++, which limits operator arithmetic and what functions can be called, may be sufficient for many technically oriented designers. Alternatively, full blown graphical scripting methods like Unreal's Kismet or Crytek's Flow-Graph could output C++ rather than data, allowing the compiler to make optimizations where possible. A hybrid approach permitting existing building blocks to be connected together, and new ones constructed in code on the fly, seems a good solution for allowing designers freedom of expression in a safe and highly productive environment.

Meanwhile, we will continue to evolve the current codebase in our search for faster iteration times, easier development, and the potential for maximum performance. Check the author's blog at RuntimeCompiledCPlusPlus.blogspot.com for updates.

15.12 Conclusion

Our initial goals with Runtime Compiled C++ were to demonstrate that compiled code could replace scripting for iterative and interactive development of game logic and behavior, particularly in performance-critical areas of code such as AI. We believe we have not only succeeded in this goal, but have also developed a technique (and permissively licensed open source code base), which permits developers to go further and develop substantial proportions of their game code this way.

Changing behavior, fixing bugs, adding new functionality, and even optimizing are now all possible without needing to restart the process, with turnaround times on the order of a few seconds.

With special thanks to Adam Rutkowski

References

[Crytek 12] Crytek. “CryENGINE 3 AI System.” http://mycryengine.com/index.php?conid = 48, 2012.

[DeLoura 09] M. DeLoura. “The Engine Survey: General Results.” http://www.satori.org/2009/03/the-engine-survey-general-results/, 2009.

[Epic 12a] Epic Games. “UnrealScript.” http://www.unrealengine.com/features/unrealscript/, 2012.

[Epic 12b] Epic Games. “Unreal Kismet.” http://www.unrealengine.com/features/kismet/, 2012.

[Epic 12c] Epic Games. “Unreal Engine 4.” http://www.unrealengine.com/unreal_engine_4, 2012.

[Fulgham 12] B. Fulgham. “The Computer Language Benchmark Game” http://shootout.alioth.debian.org/, 2012.

[Havok 12] Havok. “Havok Script.” http://www.havok.com/products/script, 2012.

[Jack, Binks, Rutkowski 11] M. Jack, D. Binks, and A. Rutkowski. “Runtime Compiled C++” https://github.com/RuntimeCompiledCPlusPlus/RuntimeCompiledCPlusPlus.

[Martins 11] M. Martins. Paris Shooter Symposium 2011. Available online (https://aigamedev.com/store/recordings/paris-shooter-symposium-2011-content-access.html).

[Microsoft 10] Microsoft. “Supported Code Changes.” http://msdn.microsoft.com/en-us/library/0dbey757%28v = VS.100%29.aspx, 2010.

[Shaw 10] J. Shaw. “Lua and Fable.” Presentation, Games Developer Conference (GDC), 2010. Available online (http://www.gdcvault.com/play/1012427/Lua-Scripting-in-Game).

[Staheli 98] D. G. Staheli. “Enhanced Debugging with Edit and Continue in Microsoft Visual C++ 6.0.” http://msdn.microsoft.com/en-us/library/aa260823%28v = vs.60%29.aspx, 1998.

[Zao 10] H. Zao. “HipHop for PHP: Move Fast.” https://developers.facebook.com/blog/post/2010/02/02/hiphop-for-php—move-fast/, 2010.