Dungeon Siege Technical Manual

Volume XXI (Si-Sq) Section 8 Subsection 14-A: Skrit

Scott Bilas, Gas Powered Games
scott@gaspowered.com
http://scottbilas.com/dungeon_siege

Abstract

Skrit is the name of a subsystem in Dungeon Siege. It comes from the word *Sanskrit* and has no particular meaning. The act of writing skrit code is sometimes known as "shoveling skrit". This document explains what skrit is and how it works.

Important: this is a "living" document and will be added to and updated over time. Major changes are logged to the table at the bottom so finding out what's new when updates are made should be a little easier.

Disclaimers

Here are some important notes regarding this doc:

- This doc is *highly incomplete*. I'm writing this in my spare time so it's going to be very slow getting updates in here, but I'll do my best. It is fairly technical, but I hope that doesn't scare anybody off. We do have a second documentation effort under way that will give higher level information about how to mod the game.

- Because this doc is incomplete, I'd really appreciate it if people would hold back emails about how this works or that works. There's a good chance that any questions about skrit will be answered in further updates to the doc. Now once we ship, of course, it's open season.

- This doc is only about skrit. How to make templates and all that is going to need to be a separate doc. I want to keep this one focused.

- Many of the examples here will probably get out of date by the time we ship. This doesn't invalidate the examples, but it does mean that you might not be able to find the same exact code if you extract all the skrits from the tank files (though there should be something very similar available).

- I have a feeling that some people are going to see the doc, see that a particular feature is not part of skrit, and then throw up their hands and say that this system doesn't do what they want. I ask those people to be patient, partly because some of those features that you can't live without are actually done using a different method, and partly because we'll be changing our focus once we ship to better support the mod community.

Thanks for your patience, and have fun. And if this stuff looks too complicated to you, well, all I can say is that once you get your hands on the full content of the game (when we ship) you'll have hundreds of skrit files, mostly well-commented, to use as examples to work from. That's how the scripters at GPG learned!

You can quite easily take an existing skrit, modify it a tiny bit, and see the results immediately, all while running the game. Don't be scared off thinking you'll never be able to write one of these from scratch.

.Scott@GPG

Conventions

In this doc some simple conventions are followed. Better get those out of the way first.

- Code, variable names, and object names, will always be written in *italics* if inline in the text. Blocks will be printed

  ```
  in this font
  ```

- Paths to resources in the file store (i.e. in all the currently mounted tanks) are shown as an ordinary DOS/unix style path. For example, this gets you a *farmboy* mesh:

  ```
  art/meshes/characters/good_a_heroes/farmboy/m_c_gah_fb_pos_a1.asp
  ```

  Note that in Dungeon Siege the forward or back slashes are considered the same. It's usually more convenient to use forward slashes, as they won't need to be "escaped" in skrit strings. Also note that the full path as shown above is rarely necessary due to the auto-expanding functionality of the Naming Key, which is plugged into pretty much everything (more on this later). When a full path is given in this doc, it's usually for reader convenience so the file is easy to find in tanks for reference.

  On occasion a file must be pulled from outside the general purpose file store, and in these cases a URL-style prefix for the protocol or driver name is required. For example when the engine queries all available maps, it iterates through directories underneath:

  ```
  maps_query://world/maps
  ```

  Note that forward slashes *are* required for the URL-style driver prefix.

- Paths to data in the *.gas* store are shown as colon-delimited strings. For example, this gets you the *farmboy* template in the content database:

  ```
  world:contentdb:templates:actors:good:farmboy
  ```

  The actual file that the farmboy template is stored in is called:

  ```
  world/contentdb/templates/actors/good/heroes.gas
  ```

  For efficiency, multiple fuel blocks are usually stored in the same file. For addressing purposes, everything within a single directory in the file store is grouped under the same parent fuel block. The actual name of the *.gas* file is not important, and is only useful for developers. The system only pays attention to what is inside the file(s).

  Fuel supports multiple databases, and uses a prefixed addressing scheme. For example, to get at the

imported multiplayer character installed for the host player, access this fuel address:

```
::import:0:0:member
```

Here, *import* is the name of the fuel database.

Definitions

As a system, skrit is based on a common and simple design that is also used by Java and UnrealScript – the buzzword-heavy pseudo-compiled, interpreted virtual stack machine! C++ code in the game engine interprets "p-code", which you can think of as an instruction set that executes on an imaginary CPU, which is internally called the "skrit virtual machine". You might wonder why we'd bother with a virtual CPU, and why we wouldn't just output machine code for the real CPU. Short answer: writing a back-end compiler for a real processor is a lot more work! Plus it's usually unnecessary, as scripting engines are typically not meant for high performance applications.

"Skrit" actually refers to many things, so first let's list them all here. This contains some internal details about how the system operates, all of which are safe to ignore if you don't care about such things. Actually, if you're just interested in doing some shoveling, take a look at the skrit that comes with the game (you will need a tank extractor to pull them out of course). A good way to learn skrit is to adapt existing GPG code, change a few things here and there, see what happens. Anyway, back to the definitions:

- *"A/The/His/Her skrit" etc.*

  Referring to skrit as a noun usually means you're talking about written scripting code, usually an individual skrit file. This code may be used in a variety of contexts, depending on the system. The major skrit-using systems in our game are: game object components, AI jobs/brains, animation chores, commands, formulas/rules, and skritbots. These will be covered in detail later in this document.

- *Skrit Virtual Machine (VM)*

  The virtual stack machine is C++ code that interprets the p-code, and this is known as the skrit virtual machine, or mondoSkritVM ™©®. From within skrit code, the currently executing VM can be accessed as the object *vm*, which exports a few exciting debugging features such as "which line am I currently executing".

  VM's exist just long enough to execute some code, and then they go away immediately afterwards (though they're pool cached for performance). They are designed to execute a fragment of skrit code at a time, not partially, but completely – there are no "latent" functions as in Unreal or Gabriel Knight 3 here, although it's simple if not tedious to simulate this functionality using states, events, and *at* functions. Every code fragment must finish completely before control is given back to the system. A fragment is a skrit function, event handler, transition handler, or dynamic trigger, and may be called specifically or automatically.

  Performance of the skrit VM often comes up, but it's difficult to judge or assign numbers because of the wide variety in p-code instruction costs. Overall, the runtime environment takes up less than 1% of total CPU at any time in current Dungeon Siege code. This is possible not because skrit is

blinding fast (which it isn't), but because GPG skrits are coded for performance by using events and triggers, which lets highly optimized C++ code detect early-out situations where the engine can completely avoid calling into the skrit code in the first place. In summary, skrit is fast because it rarely gets called!

- *Skrit Compiler*

  The compiler generates p-code from skrit code. All compiling for skrit is done on the fly and behind the scenes. There is no standalone skrit compiler, nor is there a need for it currently. The developer generally never interacts with the compiler directly. Instead, the compiler is called by various systems in the game on demand, and it will put up dialogs as it finds errors in the code that it compiles. One neat feature is that if you run the game with *skrit_retry=true* on the command line, then after the errors are reported in a skrit that fails to compile, you're given the opportunity to alt-tab away, fix the skrit, alt-tab back, and retry the compile. This is useful when prototyping and typing too fast.

  Skrit's compiler creates debug symbols in non-retail builds to aid in disassembly and runtime error reporting. The language supports events, triggers, states, dynamic and static state transitions, local functions, locally and globally scoped variables, C-preprocessor-style conditional compilation, etc. More on all this later. For the curious, it was built using MKS Lex & Yacc for easier scanner and parser generation and maintenance. Also, many shortcuts in the grammar were added out of the author's laziness in typing.

- *Skrit Object*

  A skrit object is the basic unit of skrit operation. It is a self-contained unit that has all the data and code required to execute skrit code. A skrit object is composed of two parts – the object instance and its shared implementation.

  The shared implementation is a compiled and cached piece of skrit source code, including all the p-code, debug symbols, export tables, state transition tables, and caches, as well as any shared globals if they exist.

  The object instance contains state information, non-shared global variables and properties, timers, static triggers, caches, and other per-instance data. The shared implementation is broken out from the instance data to save memory and cache compiled data. Generally developers work only with the object, and never worry about the data-sharing going on behind the scenes.

- *Skrit Engine*

  The skrit engine is the system that owns all things skrit. It manages all currently open skrit objects, the compiler, and the VM cache. It handles persistence of this system as well. If you want to obtain a skrit or execute a skrit command, you must ask the skrit engine to do it. There are a few ways to do this, depending on need, which is covered later on.

  Here is the typical skrit usage scenario from the game's point of view. A unit of skrit source code (a skrit object) is requested by a system in the game for whatever purpose – perhaps the AI wants to construct a job to run, maybe have an actor drink a potion. The skrit engine checks to see if it

already has this object in memory (they are indexed by name). If not, it finds it and compiles it on the fly into p-code and generates a shared implementation. Then it builds a skrit object that shares this implementation, assigns it storage, and gives it back to the requesting system as a handle. This system may then send events to the skrit, give it an opportunity to update itself for polling or timers, or it may call functions directly on it. Finally, when the system is finished with this object, it releases the handle back to the engine, which will release the object if all references to it are closed. If this is the last object that uses the shared implementation, then the shared memory is deleted as well.

- *System Exports*

  Functions that may be called in skrit are not actually part of the language. Various systems in the game may export functions, types, and enumerations that are detected as game startup time. There are nearly a thousand of these. Skrit scans them all, figures out which ones can be called from the language, and tags them as skrit-capable. During skrit compilation, these are all available to be used from any skrit code.

  Ultimately, skrit is simply a command dispatcher – it hooks events up to function calls, and that's about it. This is actually a problem for documentation, because it's difficult to talk about skrit without talking about the game code, and yet the game code is completely separate, from skrit's point of view. Because skrit spends most of its time calling other functions, this part of the system is highly optimized. Internally, the skrit virtual stack is identical to the system stack so that function calls have minimal latency. Don't worry about the overhead of calling functions here, as it's barely detectable on a profiler. Focus on the cost of the actual functions that you call.

Design Goals

Before we go any further, let's examine some of the design goals of skrit. This is important to help understand why certain features are implemented a certain way, or perhaps were left out completely. Hopefully this will answer in advance many of the "why doesn't skrit have feature x" types of questions. It's also important to note that these design goals were meant for the prerelease development process. After the game is finished, the goals will change to accommodate the mod community.

The most important design goal of skrit is that it is not meant to be used as a general purpose language. As a result, it's missing many features of GP languages such as *for*-loops, *switch/case* statements, inheritance trees, structure definition, arrays, lists, memory allocation, etc. Many of these features, such as *for*-loops, were left out on purpose to discourage use of skrit as a GP language, and save development and test time for more relevant features, such as the state system. Skrit was deliberately designed to prevent our scripters from trying to build engine features in the language. Not only does this potentially destabilize the game, suck up cpu and memory, and increase maintenance cost, but it also potentially duplicates engineering efforts. Instead, when an advanced feature is required, an engineer will implement it so that all systems in the game can use it, and then skrit can call into that.

Skrit is meant to be a duct tape language. It is designed to tie fast and powerful C++ engine systems together quickly and easily. Skrit has custom grammar intended to make it easier to naturally express solutions to the problems of a real-time event-driven game. If you've ever tried to create a state machine in C++ without using a bunch of inflexible, ugly, hard to read macros that only an engineer armed with a compiler can do anything with, you'll know what I mean. Because it's duct tape, skrit is useless by itself. It

can't do anything interesting without calling engine features, and so understanding how the engine works is vital to understanding how skrit may be used to create interactive content. Documentation for other systems in the game is beyond the scope of this document, however.

Another important design goal of skrit is that it should be very difficult to take the game down with skrit. If you try to do it, you certainly can, but during normal development a bad skrit should almost never cause a crash of the game. The engine is generally able to report the error, shut down the offending skrit, and continue. Note that this only applies to the development builds of the game. In retail, to save memory and CPU we have removed the majority of error checking and handling routines, and so it assumes nearly perfect content in order to run.

One thing to notice about skrit is that, like all other Dungeon Siege systems, it has strayed from its original design to handle the needs of our rapidly and ever changing project. Some features may have gotten stale from not being used in a while. Other features may be tuned to a particular need we had at one time and look kind of gnarly. To support the mod community, some features will be added over time. Once native interfaces are in, any remaining missing features can be filled in via external DLL's built by other languages such as C++ or Delphi.

Storage

Skrit code can live pretty much anywhere, but it's typically found in three types of places.

1. It can be a *.skrit* file that sits in the resource store (usually the logic tank file). This file does not need to have the *.skrit* extension, but it's recommended, as this is the convention. It's also the default when the extension is left off of skrit file queries.

   Skrits are found by other systems in the game via the resource store by using their fully qualified path name, unless the system using them knows they must be in a particular place. For example, the content database knows that all of its component skrits are located under *world/contentdb/components*, so the prefix there is automatic, but any external systems that need to reference those must use the full path prefix. This is inconvenient in general, so the naming key is supported (using the *art/namingkey.nnk* configuration file). For example, the engine will expand *k_job_c_attack_utils* into *world/ai/jobs/common/k_job_c_attack_utils.skrit*. Note that the extension can and should be dropped when referring to skrits from code.

2. Skrit code can live as a complete skrit embedded as a value in a fuel block (in a *.gas* file). For example, the *world/contentdb/pcontent.gas* file, which is used to configure the parameterized content system, contains (among other things) a *[formulas]* block that is a complete skrit. This type of skrit is generally only used by special systems, as the engine doesn't have any direct support for running Skrit from fuel blocks – it's more efficient for them to be separate files.

3. It can live in a *.gas* file as a partial skrit embedded pretty much anywhere. Skrits can be loaded as "commands" rather than as actual objects (more on this later) in which case they can be a simple expression. An example of this is the formula used to calculate the duration that an effect may last, which is stored as the *effect_duration* field of the *[magic]* component in a template or object instance. The skrit command gets wrapped in a simple function, compiled, executed, then discarded once the result (if any) is obtained (though caching is in place to keep this peppy).

Taxonomy

There are a few major classifications of skrits used in the game. The skrit engine doesn't see any difference among these, but different systems use the engine in different ways, and it's worth enumerating all of them. Here they are:

1. *Skrit component*

   A skrit component is a general purpose chunk of logic that can be attached to a game object, and must live in *world/contentdb/components* or a subdirectory. This is the most common type of skrit in Dungeon Siege, and is responsible for most of the non-AI interactive content in the game. Usually a skrit component is added to the system by modifying an object template to include the new skrit component, like this:

```
[t:template,n:camera_stomp]
{
	[camera_stomp]
	{
		duration = 3.0;
		magnitude_y = 0.2;
	}
}
```

   In this example, we've created a template called *camera_stomp* that has a single component in it, also called *camera_stomp*, which refers to the skrit file *world/contentdb/components/fx/camera_stomp.skrit* from the resource store. At game startup time the content database scans for all available templates, and then it compiles the skrit components that are used by them. Creating a skrit component and adding it to the game is easy: stick it somewhere in the *components* directory or a subdirectory and add a template that uses it.

   In the game we would instantiate this template ("clone a Go" in DS lingo) to cause the camera to temporarily shake when a large monster stomps on the ground (among other things). Here we've also customized the fields for the component to cause a 3 second duration but low magnitude stomp. The "fields" are actually properties in the skrit – the purpose of the template is to configure reasonable default values that can be tuned later at runtime if it's useful.

   Skrit components receive a standard set of events, such as *OnGoUpdate*, *OnGoDraw*, and *OnGoHandleMessage*. These events are broadcast to every component in a game object (Go) and can be caught by event handlers or triggers in skrit. Dungeon Siege runs on world messages.

2. *AI brain*

   The AI controlling a monster is run through the *GoMind* component, which may have a queue of brains that $$$

3. *AI job*

   $$$

4. *Animation skrit*

The AI handles high-level tasks such as path finding and sending chore requests to the *[body]* component for animation. The next level down is the animation system that performs the chores that are requested by AI. Animation skrits in the *art/animations/skrits* directory are responsible for these. These skrits must be hooked up in the *[chore_dictionary]* block of the *[body]* component of a template. Inside the chore dictionary is a set of chores that a particular Go may execute, and each block (such as *[chore_die]*) has a set of parameters the determine how to play each. Of interest to this doc is the *skrit* value inside each chore block, which says the name of the skrit to direct the chore, in addition to any parameters it may require.

For example, here is the chore dictionary for the spinning waterwheel near the first farmhouse at the start of the game:

```
[chore_dictionary]
{
	[chore_default]
	{
		skrit = rotatey?rpm=8;
	}
}
```

This defines a chore named *chore_default*, which uses *art/animations/skrits/rotatey.skrit* to drive it, configuring the *rpm* property of the skrit with a value of 8.

Animation skrits drive the lowest-level components of the animation system. They perform tasks such as configuring the blenders and converting embedded effect keys into world messages. They can also do procedural animation to manipulate bones directory (as in the *rotatey* skrit above).

5. Formula skrit

Formula skrits are used all over Dungeon Siege in random places as needed – for example, some are used for calculating damage or experience, and others are used for tuning the enchantments based on skill levels. A formula is simply a skrit fragment evaluated as an expression, where the result of the expression is returned to the calling code. So of course anything that can normally go into an expression can go into a formula skrit, such as function calls and simple math. An example of this type of formula is found in the experience calculation for casting the spell *healing_hands* at *world:contentdb:templates:interactive:spell:healing_hands:magic*:

```
cast_experience = [[(#src_mana > ((#maxlife - #life)*((2*(#magic+1) +
8)/(19+((#magic+1)*6))))) ? 0.12*(19+((#magic+1)*6)) :
0.12*(#src_mana*((19+((#magic+1)*6))/(2*(#magic+1) + 8))) ]];
```

There are a few special places that the code looks for entire skrit files stored in *.gas* for formula calculations. These formulas are stored in *.gas* files rather than C++ code so it's much easier for content developers to tweak them. Some examples of these are the parameterized content formulas in *world:contentdb:formulas*, and the skrit-callable arbitrary formulas (from the *ContentDb* object) stored at *world:global:formula:general_formulas*.

6. *Skrit command*

   Skrit commands are chunks of skrit code that get automatically wrapped up by the compiler into functions and executed. The simplest version of a skrit command is what happens on the Dungeon Siege console if you type something like this:

```
/Report.GenericF( "Hi there\n" )
```

   The console detects "this is a skrit command" and rips off the leading slash, then passes it along to the skrit engine to be wrapped, compiled, and executed. It eventually ends up looking like this:

```
X$
{
	Report.GenericF( "Hi there\n" )
	;
}
```

   Then the engine compiles the first function it finds (*X$*, here). If a skrit command is prefixed with #, then the code is compiled and executed as-is.

   The *Skrit.Command(string command)* function will execute arbitrary skrit code directly. This is an easy way to actually generate skrit code from within skrit to run. It can also be used to enable simple skrit code to be typed in from SE for storage in a component as a property, and then executed as a command by another skrit.

7. *Skritbot*

   Skritbots are used only for development purposes and will be disabled for the retail build. They are meant to permit automation of testing and profiling. Skritbots usually go under the *auto* directory and may be run from the command line in *skritbot=auto/mp?players=3* (start the multiplayer automation skrit, host a game, and wait for 2 other players to join), or added via skrit/console through the *SkritBotMgr* object.

   In order to help with automation (knowing when someone presses a button, for example), Skritbots automatically hook into and receive events for UI window and interface messages sent, UI actions initiated, and world messages sent. They also receive updates and draw messages. Skritbots may call UI functions to cause buttons to be pressed, etc.

8. *Game type skrit*

   In multiplayer, a set of "game types" are available that determine the rules for playing the game that session. For example, there are some global game types like "PvP Teams" and "Cooperative No Teams". Game types can be defined per-map in *world:maps:<map>:info:gametypes*, but if they do not exist, then the global types stored at *world:global:gametypes* are used instead.

   Inside each game type is a value named *skrit* that may contain a set of functions to implement the game type specific behavior for rules checking and other administrative details. Currently supported are the functions *check_team_for_victory$(int)* and *handle_world_message$(WorldMessage)*. These functions are not required to exist, and if they do, they are evaluated on the server only.

The victory check is called once each sim per team and passes in team ID's. Its purpose is to check to see if the conditions for victory have been met and set each condition that has been met accordingly (such as team-has-grail and team-spanked-monkey). The victory code will independently check all available conditions and if it detects a win, it will move the game state to the victory/defeat screen.

The world message handler is called on every single world message that is sent out. This permits it to perform a variety of responses to game events. For example, it could handle the *WE_KILLED* message and tell everyone that "Player got Snu-Snu'd", or it could watch for a *WE_DAMAGED* message, check the health of the damaged game object, and if it's low, send out messages to fellow team members saying "Red Warrior Needs Food Badly!". Or maybe it could detect a *WE_DROPPED* message, check what was dropped, and then if it's the remote nuke detonator, then set off the nuke!

Because these functions are called frequently, care should be taken not to eat up too much CPU.

Keywords and Operators

Finally, what everybody seems to be interested in – the keywords and operators of the language itself. You may be asking yourself "well if it doesn't have *for*-loops or arrays, what exactly *can* it do?" Lots of stuff, and almost all of them fun! Let's start with a list of keywords and lexer features first. Because most of these are the same as C, if they are unfamiliar to you, you can refer to an online tutorial for that language.

- Control statements: *if*, *else*, *while*, *break*, *continue*, *forever*, *return*, *sitnspin*, *abort*, { } ;.

  These should be pretty familiar to any programmer. New is the *forever* keyword which is functionally identical to *while(1)*, the *sitnspin* keyword, which does absolutely nothing, and the *abort* keyword, which causes the skrit to immediately cease and desist operations. Note that the { } braces are required for any code block. And as in 'C', it's not necessary to add a *return* at the bottom of a function with no return value.

  [Ok, so no *for*-loops, but we grudgingly have *while* statements. That feature was for testing only, but somebody ended up needing it for real and so it stuck.]

- Type declarations (built-ins) and modifiers: *bool*, *int*, *float*, *string*, *void*, *shared*, *property*, *hidden*, *doc*.

  The types should be familiar – generally the *void* type can be left out, but it is included as an option for C++ programmers who can't break a habit. The rest are for setting up properties in a skrit that may be tuned from Siege Editor or other code means (such as URL-style skrit startup). Note that *bool* is considered an integral type and will get promoted accordingly.

- Constants and special variables: *true*, *false*, *null*, *__LINE__*, *__NAME__*, *this*, *vm*, *owner*.

  The first three constants are simple. The next two are meant for debugging, where you can do something useful with the current skrit filename and line number, perhaps in an error report. The last three are special variables that give you access to skrit internals, or the owner of the currently executing skrit (in Dungeon Siege, the owner is usually a component or *aspect*).

- Event, trigger, or state machine related: *event*, *trigger*, *at*, *msec*, *frames*.

  All of these deal with declaring an event handler or trigger. Event handlers and triggers are bound at compile time, and are a very fast callback mechanism, approaching C++ in efficiency. This is the primary method that the game uses to communicate with skrit logic. We also have the ability to define *at* functions, which are simple time-based triggers meant to help emulate latent functions.

- State machine and transition related: *state*, *poll*, *transition*, *and*, *startup*, *setstate*, *if*, ->.

  These let you define states, traits for them, transitions among the states, and triggers (static and dynamic) that define the conditions for state transitions.

- Preprocessor-style directives: *#include*, *#option*, *#only() [[ ]]*.

  The *#include* feature is just like C, in that it pulls one file into another, and both are compiled as if they were one unit. As skrit does not support external linkage, this is useful for having common utility functions used in multiple skrits. The *#option* feature is meant for tuning compiler options, and *#only* is sort of like *#if* and *#endif*, allowing you to conditionally compile chunks of code. This is most often used to remove development-only code from retail builds during compile to reduce memory usage and prevent errors on functions that are not implemented in retail.

- C and C++ style comments: /* *comment* */ or // *comment*.

  This is pretty exciting stuff, but please try to contain yourself. Note that nested comments are not supported. Instead, use *#only(0) [[ ]]* instead to "comment out" large blocks of code that may have embedded comments.

- Decimal or hexadecimal numbers: *1234*, *0x600DF00D*.

  Notice the lack of octal support. Too many times have people written 000123 as a number, padding the front with zeros, and ended up with weird resulting number (in this case 83), never knowing why. So no octal! Integers in skrit are 32-bit signed numbers, though it can accept unsigned numbers (as with 32-bit hex constants), which get converted to signed integers internally.

- Floating point numbers: *123.456*, *-.789*, *3.4e-10*.

  These are 32-bit floats internally, though skrit automatically converts to and from double when necessary (as with vararg functions). This is exactly the same as C.

- String literals with embedded C-style escapes: *"jooky\tis\ngood\x12"*.

  This is exactly the same as C in three ways. First, all of the standard C escapes are supported here: \a, \b, \v, \t, \n, \f, \r, \", \\, \x7F. Note that octal is not supported here either! Second, the C rule that a string may be "continued" on the next line by end it with a trailing backslash works. And third, strings are automatically concatenated when adjacent, such as this: *"jooky\t" "is" "\ngood\x12"*.

- Four character code (FOURCC) literals: *'Mule'*.

  This is the same as C as well. Integers in skrit are 32 bits wide, which can be represented as four characters, each eight bits wide. Up to four characters can be put between single quotes to create an integer. This is generally done for performance reasons, because integers are very fast to pass around and compare (especially relative to the cost of using actual strings), and many human-readable constants can be represented in only four characters. The animation system uses these quite a bit for event codes.

- Postfixed user symbol names: *MySymbol$*.

  Every user-defined symbol in skrit is always postfixed with the $ character. The $ doesn't mean anything other than "this is user-defined". This makes it very easy to extend the language – I can easily add new keywords and system functions, etc., without breaking any existing skrits. The $ postfix was inspired by C-64 BASIC, though in that specific case $ meant "string".

- Standard comparison operators: `< <= > >= == != <>`.

  All of these are exactly the same as 'C', with the extra fun that they all work on all types including strings. All comparisons with strings are case-sensitive. Also note that the last two operators *!=* and <> are functionally identical.

- Case-insensitive comparison operator: ~=.

  This operator does an approximate test for equality and works on either strings or floats. Strings will do a case-insensitive test, and floats will do an "approximately equal" test using an epsilon value (approx 0.00001).

- Boolean logic operators: `||` *&&*.

  These operators work the same as in 'C', where `||` returns *true* if either side of the expression is true, and *&&* returns true only if both sides of the expression are true. These operators only work on integral types.

  IMPORTANT: *unlike* 'C', neither of these operators "short-circuits". This is one of the most common errors for people writing skrit who have a C/C++ background. Both sides of the expression are *always* evaluated. So this type of code, while normally safe in 'C', will probably cause an access violation in skrit:

```
if ( (something != null) && something->IsPretty() )
{
      ...
}
```

  So be careful! Note that this code is safe:

```
if ( something != null )
{
      if ( something->IsPretty() )
```

```
        {
            ...
        }
}
```

[Making these operators short-circuit is high on the list of feature requests for the next game.]

- Math related operators: + - * / % ( ) **.

  Here we have (left to right) operators to add, subtract/negate, multiply, divide, modulo, group, and raise to a power. The order of operations is the same as 'C', except for the new ** (power) operator which lives between multiply/divide/modulo and 'not'. Note that the % and ** operators only work on integer and floating point types.

  Note that the + operator is supported for strings, as in BASIC. It causes the result string to be concatenated, so if *A* is *"hi"* and *B* is *"there"*, then *A* + " " + *B* returns *"hi there"*.

- Standard assignment operators: = += -= *= /= %= **=.

  These work as in 'C', plus the additional **= operator for raising a variable to a power and assigning the result back to the variable. And of course += works on strings as well.

  IMPORTANT: *unlike* 'C', each of these operators can only be used as a statement, and not as an expression. So if you are accustomed to doing this in 'C':

```
if ( (rc += func()) > 10 )
{
    ...
}
```

  Do this instead in skrit:

```
rc += func();
if ( rc > 10 )
{
    ...
}
```

  This is to prevent people from accidentally assigning values where they mean to compare them (as in the classic = vs. == mistake made in 'C' from inside an *if* conditional expression).

- Ternary assignment operators: *? :*.

  This goofy operator is similar to 'C' but is stricter, and internally operates as an *if* statement. Both expressions on either side of the *:* operator must be compatible (i.e. of the same type or can be promoted to the same type). Note that because this operates as an *if* statement, this operator does actually short-circuit, and will only evaluate either the second or third expressions depending on the condition of the first expression.

  Many people don't like this operator for a variety of reasons, but it's really useful for this sort of thing:

```
Report.GenericF( "Guess what, I %s an actor!",
      owner.Go.HasActor ? "am" : "am not" );
```

- Bitwise operators: | ^ &.

  These operators – *or*, *exclusive-or*, *and* - work the same as in 'C', and only operate on integral types.

- Bitwise assignment operators: |= ^= &=.

  These operators – *or-assign*, *exclusive-or-assign*, *and-assign* - work the same as in 'C', and only operate on integral types. As with the other assignment operators, they only work in statement form, and not as expressions.

Language Features

$$$ much more coming soon here

Opcodes

As mentioned previously, skrit is a p-code interpreted language. At runtime, the compiler just-in-time compiles script code into p-code, and then caches this. When the skrit objects are required to execute, the p-code instructions are interpreted by the skrit virtual machine. This section lists all available skrit opcodes.

This is pretty low-level stuff, but the motivation here is to better understand errors in the code. In skrit it's easy to nest calls and pass multiple parameters. Knowing what the opcodes mean will help diagnose problems – e.g. it will be easier to figure out which parameter is bad, which function failed, etc. If Something Bad happens while a skrit is executing, and it's a development build (non-retail), then an error box will pop up with a mixed-mode partial disassembly around the area in the skrit that the error occurred.

The most often cause of an error box in skrit is calling a function with bad parameters. For example, if an engine function operates on actors, and a skrit passes a pointer to a Go that does not have an actor component, then the engine will access violate. Skrit then catches it, handles the condition, and puts up an error box that points to the opcode in skrit (probably a *CALLMEMBER* instruction). This can be used by developers to fix their skrit code.

Many opcodes have one or more trailing letters which signify the type of variable they operate on, and they are:

B	= boolean
C	= const char*
O	= offset into string table (constant)
F	= float
H	= managed class handle
I	= integer
P	= general pointer
S	= string - actually a string handle

Finally, opcodes that are marked with *binary* operate by popping the top two parameters off the stack, operating on them, then pushing the result. Ops marked with *unary* operate directly on the top of the stack.

General

SITNSPIN	do-nothing
ABORT	abort skrit with fatal error

Flow Control

RETURN0	return from current function (void return)
RETURN	pop the top element off the stack and store in *eax*, then return

Note that on the *CALL* variants, if it's a *vararg* function being called, the number of parameters will appear on top of the stack. If it's a member function call, then *this* as a handle or pointer will appear beneath all the parameters.

CALL	call the given function with the vm's current stack as the parameter block
CALLSINGLETON	…called on a singleton type
CALLMEMBER	…member function call on a class
CALLSINGLETONBASE	…called on a base class of a singleton (upcast required)
CALLMEMBERBASE	…member function call on a base class (upcast on *this* required)
CALLSKRIT	call the given function with the vm's current stack as the parameter block
BRANCH	add offset to current instruction pointer (IP)
BRANCHZERO	pop integer off stack, if zero then add offset to IP

Arithmetic

ADD(I/F/S/C)[(S/C)]	*(binary)* x + y (note the S/C forms are for string concatenation)
SUBTRACT(I/F)	*(binary)* x - y
MULTIPLY(I/F)	*(binary)* x * y
DIVIDE(I/F)	*(binary)* x / y
MODULO(I/F)	*(binary)* x % y
POWER(I/F)	*(binary)* x ** y
NEGATE(I/F)	*(unary)* -x

Logic

BANDI	*(binary)* x & y
BXORI	*(binary)* x ^ y
BORI	*(binary)* x | y
BNOTI	*(unary)* ~x
ANDIB	*(binary)* x && y
ORIB	*(binary)* x || y
NOTIB	*(binary)* !x

Comparisons

These each push a 0 or 1 result after doing the comparison.

ISEQUAL(I/F/B/S/C)	*(binary)* x == y
ISNOTEQUAL(I/F/B/S/C)	*(binary)* x != y
ISAPPROXEQUAL(F/S/C)	*(binary)* x ~= y
ISGREATER(I/F/B/S/C)	*(binary)* x > y
ISLESS(I/F/B/S/C)	*(binary)* x < y
ISGREATEREQUAL(I/F/B/S/C)	*(binary)* x >= y
ISLESSEQUAL(I/F/B/S/C)	*(binary)* x <= y

Type conversions

INTTOBOO	*(unary)* convert int to bool (anything nonzero is *true*)
INTTOFLT	*(unary)* convert int to float
FLTTOINT	*(unary)* convert float to int
FLTTODBL	*(unary)* convert float to double (generally used for vararg func calls)
OFFTOPCC	*(unary)* convert an integer offset to a *const char** (string table lookup)
STRTOPCC	*(unary)* convert a string object pointer to a *const char** (*c_str()* call)
CSTRTOSTR	*(unary)* convert a *const char** to internal string object
HDLTOPTR	*(unary)* dereference a managed handle and convert to an object pointer
BASEADJUST	*(unary)* upcast a derived class pointer to a base class by offsetting
MASK(2/3)	*(unary)* mask the high bytes, necessary for x86 'C' compatibility

Stack operations

PUSH(1/2/4)	push a constant given by the data onto the expression stack
PUSHTHIS	push *this* (skrit object pointer) onto the expression stack
PUSHVM	push currently executing skrit VM pointer onto the expression stack
PUSHOWNER	push *owner* of skrit onto the expression stack
PUSHPRM	push next param from incoming args and advance the pointers
SKIPPRM	skip to the next param in the incoming args
SWAP	swap the top two elements on the stack
POP	pop the top x elements off a stack and throw them away
POP1	pop the top element off the expression stack and throw it away

These opcodes load something out of one of the skrit stores and push it onto the expression stack.

LOADVAR	load a builtin variable from the general store
LOADSTR	load a string pointer from the string store
LOADCSTR	load a *const char** from the string store (*LOADSTR* + *c_str()*)
LOADHDL	load a handle from the handle store

These opcodes pop the top of the expression stack and store it in one of the skrit stores.

STOREVAR	store a builtin variable in the general store
STORESTR	store a string pointer in the string store
STORECSTR	store a string pointer in the string store (special)
STOREHDL	store a handle in the handle store and add a reference to it

Storage operations

ALLOCVAR	allocate space for a new variable in the general store
ALLOCSTR	allocate a new string in the string store
ALLOCHDL	allocate space for a new handle in the handle store
FREEVAR	free the last x elements from the general store
FREESTR	free the last x strings from the string store
FREEHDL	free and dec the ref counts of the last x handles in the handle store

State operations

SETSTATE	set the state index to the new value (set the next pending state)

Pitfalls

$$$ Start with: Watch out for thinking you're in C and declaring a local complex variable (e.g. a *vector*).

Events

There is a standard set of events sent by the system to skrit objects, which have the opportunity to respond to them via handlers and transition triggers.

- General

OnConstruct	- called right after skrit object construction
OnDestroy	- called right before skrit object destruction
OnLoad	- called right after loading a skrit object from a saved game
OnConstructShared	- same as *OnConstruct* except for the shared implementation
OnDestroyShared	- same as *OnDestroy* except for the shared implementation
OnLoadShared	- same as *OnLoad* except for the shared implementation
OnEnterState	- called right after a state change, upon entering a state
OnExitState	- called right before a state change, upon leaving a state
OnTransitionPoll	- called when dynamic transitions are about to be polled
OnTimer	- called when a skrit timer expires

- NeMa

OnUpdate	- called each time an animation state machine is updated
OnStartChore	- called when a new chore is started on an aspect

- Skrit components

OnGoCommitCreation	- called when a Go is committing creation (second thread! careful!)
OnGoCommitImport	- called after finalizing an imported character in a multiplayer game
OnGoShutdown	- called when a Go is shutting down
OnGoPreload	- called after committing creation (preload other Go's etc. here)
OnGoHandleMessage	- called when a Go receives a world message directly
OnGoHandleCcMessage	- called when a Go is cc'd on a world message due to watching
OnGoUpdate	- called each sim, don't do anything *too* expensive here
OnGoUpdateSpecial	- called each sim for player characters only (special use only!)
OnGoDraw	- called each sim to permit extra drawing
OnGoResetModifiers	- called when a Go's modified values should be reset to natural values
OnGoRecalcModifiers	- called when a Go's modified values should be recalculated
OnGoLinkParent	- called on a new child right after it gets a new parent
OnGoLinkChild	- called on a new parent right after it gets a new child
OnGoUnlinkParent	- called on a child right after it loses its parent
OnGoUnlinkChild	- called on a parent right after it loses its child
OnGoDrawDebugHud	- *(dev-only)* called each sim when visible to draw debug info
OnGoDump	- *(dev-only)* lets a component participate in a Go dump

- Skritbot

OnBotHandleMessage	- called when a world message is sent
OnBotHandleUiAction	- called when a UI action is executed
OnBotHandleUiWindowMessage	- called when a UI window message is sent
OnBotHandleUiInterfaceMessage	- called when a UI interface message is sent
OnBotUpdate	- called each sim
OnBotSysUpdate	- called each sim (ignores paused state of game)
OnBotDraw	- called each sim to permit drawing at correct time

- AI job

OnWorldMessage	- called when a Go's mind receives a direct world message
OnCcWorldMessage	- called when a Go's mind is cc'd on a world message
OnJobInit	- called on job construction (set up the job here)
OnJobInitPointers	- called on load game (init cached pointers here)

Debugging Support

Like most scripting languages, skrit does not have a debugger. However, it does come with an array of debugging support features. The majority of these are only available from non-retail builds of the game. To save memory and CPU, these are compiled out of the retail build of the game. Also note that some of these features have nothing to do with skrit, but they are useful debugging tools that can be used in conjunction with skrit development.

- *The Development Console*

Dungeon Siege has a development console that serves four purposes. First, it serves as a log of all messages generated by the system. Second, it contains real-time statistics about the game and various context-sensitive states. Third, it allows entry of console commands, where custom code is written to support each console command. And fourth, it allows direct entry of skrit. That's what this section is about.

If you bring up the dev console and type a forward slash, any remaining text on the line will be wrapped up in a skrit function and executed as a command by the engine. A simple skrit command you can type in to delete the focused game object would look like this:

```
/GoDb.SMarkForDeletion( GoDb.FocusGo )
```

Note that a terminating semicolon is optional. The console supports copy/paste, so a trick that we use at GPG is to keep a file open in a text editor that contains micro-skrits like these for various purposes. Then you can copy/paste them directly into the console for execution. The console automatically packs separate lines together so large chunks of skrit code can be executed easily. This in itself is also a convenient testing tool – if you are working on a skrit function somewhere and want to test a portion of it, you can just copy-paste that section into the console for execution.

- *The Help System*

  A set of functions exist as part of the skrit set of exports that are meant to help out in querying the system's functions, types, and enumerations by printing out generated documentation to the console (via the *generic* report stream). These are all contained within the *Help* class, and contain functions such as these:

  All – do a full dump of all system exports. This used to be worth doing back when we had a few functions, but these days the log is too large to be useful unless you route it to a file.

  Classes, Enums, Globals – list all available classes, enumerations, or global functions in the system. This is just a list of names and docs if available.

  Class, Enum, Global – each of these functions will query an individual instance of a class, enumeration, or global, and print out as much information on it as possible. Classes will list all their methods (with parameters), and variables, enumerations will list all their constants (if available), and global will list all parameters the function takes.

  As these are all callable from skrit, they are also callable from the console. This will list all available help on the *GoDb* class, for example, if typed into the dev console:

  ```
  /Help.Class( "GoDb" )
  ```

  Note that the *Help* class can be used to get help on itself.

- *Report Streams*

  A "report stream" is just a named character output stream that may be written to from code. Streams are like pipes in DOS or Unix in that they can be routed to one or multiple targets. The

default target of a stream is the console log and debugger output (if a system debugger is running). Additionally you can hook a stream up to output to a file via the console *report* command. Streams support a variety of features such as indentation, variable-argument (printf-style) output, enabling/disabling, and table reporting.

Skrit can use report streams to do classic brute-force "printf debugging", which just means that you sprinkle diagnostics and messages in your skrit code to print out variables and other info at useful times to help track down problems in the script. For example, if you think that an object might be getting deleted just before you set its position, you could put a few lines like this in the script in various places to try to track down when it gets deleted:

```
Report.GenericF("Go 0x%08X %s exist!\n",
      m_Goid$, m_Goid$.IsValid ? "does" : "DOES NOT" );
```

This simple line of code will print out whether or not a particular game object exists by testing its *Goid* (game object identifier) for validity.

Additional report streams exist for performing assertions, bringing up error boxes, fatals, and message boxes, logging warnings, and causing debugger breakpoints. Get help on the *Report* class for the full set of these.

- *Exception Handling*

  If a skrit does something bad, such as dividing by zero or dereferencing a *null* pointer, the condition will be caught and you will be notified via a dialog box. The dialog contains lots of information about the skrit that caused the problem, including a mixed-mode partial disassembly of the skrit source code and its opcodes, with an arrow pointing to the offending line. The dialog can be ignored, execution will continue, and the skrit function will simply have its execution aborted. Of course, the next time this skrit is called the same thing will probably happen again. The exception that occurred, as with all errors, warnings, and fatals, will be logged to a file with a timestamp for tracking purposes, to aid reporting the bug to the content developer.

  In retail mode, nothing happens – the exception is caught, ignored (no error logged to a file either), and gameplay continues without the user noticing a thing.

- *Dynamic Content Reloading*

  Many things in Dungeon Siege can be changed and reloaded dynamically while the game is running. Most types of skrits are included $$$

Revision History

9/13/2001	Scott Bilas	Initial draft (containing many holes, quite drafty)
9/22/2001	Scott Bilas	Added more random stuff, still drafty
9/25/2001	Scott Bilas	Added all-important disclaimers until draft is finished
1/1/2002	Scott Bilas	Made a bunch of random changes Added "Conventions" section Renamed and filled out "Taxonomy" section with all known types

Renamed and finished “Keywords and Operators” section
Filled out “Opcodes” section
Filled out “Events” section
Made New Year’s resolution to write more docs faster