overload 118

DECEMBER 2013 £3

Migrating from Visual SourceSafe to Git

The story of a version control system migration; the trials and triumphs

Object-Environment Collision Detection using Onion BSPs

We look at a new technique for collision detection

On the Other Side of the Barricade

Interviewing is an important skill which is hard to get right

How to Program Your Way Out of a Paper Bag

It's often claimed people can't program their way out of a paper bag. Is that *truly* the case?

A magazine of ACCU

ISSN: 1354-3172

OVERLOAD 118

December 2013

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Matthew Jones
m@badcrumble.net

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simonsebright@hotmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 119 should be submitted by 1st January 2014 and those for Overload 120 by 1st March 2014.

The ACCU

The ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members - by programmers, for programmers - and have been contributed free of charge.

Overload is a publication of the ACCU
For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

Not So Much a Program, More a Way of Life

In an attempt to continue and improve on the successful formula of previous editorial avoidance techniques, Frances Buontempo considers what a program really is.

After many attempts at answering vital questions about programming, in lieu of writing an editorial, the time has finally come to get straight to the point. So with no further distractions or mucking around, let us once more focus and turn our attention to a vital question which requires an answer before we can even consider what to write about for an editorial: what do we mean by a program?[1] I will attempt to avoid distractions, such as whether it should be spelled as 'program' or 'programme'. The internet says 'programme' is the old way of spelling 'program' when it comes to computer programs, so let us take that as read.[2]

Are programs dynamic?

Programs come in many guises. They can be dynamic, that is a script that is interpreted at run time. Alternatively, they can be static, that is requiring compilation. They can also be a half-way house, compiling to byte code, such as Java or C#. Please note, that static does not mean they never change. Has anyone written a program that never changed?[3] There is a slim chance a one line throwaway script might never change. I have heard rumours of such things, but never seen one. Whether a program is dynamic or static, if it has been in source control the log may indicate it is in truth always dynamic, or constantly changing. Code only tends to stagnate, or become truly static, when it's dead. Perhaps code is like a shark. 'I'm like a shark, I don't swim backwards' [Google], though perhaps many of us have done something along the lines of

```
git reset --hard HEAD…
```

Forwards or backwards is still a change and therefore dynamic. More importantly, if the code itself is not in version control that is a potentially life threatening decision, if the person who failed to use version control is in striking distance. More positively, as Heraclitus [Heraclitus] said,

> All entities move and nothing remains still.

sometimes taken as "You can't stand in the same river twice." Aside from Heisenbugs [Heisenbugs], wherein attempts to debug the code change how it behaves, a program tends to evolve and change over time, either meandering towards the requirements or trying to play catch-up with the requirements, or even trying to discover the real requirements. It could be argued that a program is its current code along with its history, including all the crime scenes to watch out for [Tornhill13] and the bug tracker history and perhaps the documentation. Sometimes there is no documentation, or no version control. This leaves us with just the code as it is now. If the code is compiled, there is no guarantee this matches what's in production. This is a frightening place to be. Sometimes, we have only the executable and no code. An environment including tests, useful documentation, version control and the code itself is a safer happier place to be. A program is much more than just something that can be run or executed.

Are programs executable?

Not all code is executable. Certainly some code will run on one machine and stubbornly refuse to even start on another machine, though some code is not designed to be run directly. Most of us will have used a variety of libraries, APIs, interface, protocols and similar at some point. Some of us may even have written libraries, hopefully managing the difficult task of making them easy to use correctly and hard to use incorrectly [97Things].This sage advice tells us we are really writing code for fellow programmers, not the customer or the machines. The executable is for the outside world. The code itself is another matter.

Recently on accu general, a controversial quote from Bjarne Stroustrup [Stroustrup] caused a brief discussion:

> I have yet to see a program that can be written better in C than in C++. I don't believe such a program could exist.

By better, Bjarne explains he means 'smaller, more efficient, or more maintainable' and goes on to say:

> Many then fall in love with their obscure and complex code, considering it a sign of expert knowledge. It is amazing what people can fail to learn when they are told that it is difficult and useless.

As stated at the outset, we might need a clear definition of program in order to evaluate this claim. One immediate response on the email discussion was "Program, sure. Library, not convinced."[4] Is this difference significant? In terms of the wiring between a library and what uses the library, some may suggest that a C interface to a library is easier to work with from another language than a C++ interface, so a program and a library do differ. Nonetheless, on paper, if you'll forgive the misnomer, both a program and a library are lines of code written in a specific language. At a high level, the main difference is simply that one can be run or executed while the other will be used by something that runs. Some

1. The title and strapline are from http://en.wikipedia.org/wiki/Not_So_Much_a_Programme,_More_a_Way_of_Life
2. US: program is the only spelling normally used. UK: programme is used in all cases except for computer code, in which case program is generally used. Older sources may use programme for computer code. http://en.wiktionary.org/wiki/program#Usage_notes
3. Recently seen on twitter: "never changed *since release* – several of the games I worked on, no patches, no sequels, code base thrown out, not repurposed." Thanks @codemonkey_uk
4. Tim Penhey

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a programmer for over 12 years professionally, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

sap[5] then side-lined the discussion, enquiring as to why such sporting terms as 'run' or violent terms such as 'execute' get used for programs. It seems the discussion has happened before, for example see the English StackExchange [SE]. It seems both words are rooted in the idea of carrying out instructions. The word 'program(me)' itself hails from Greek for a written public notice [program]. A variant of the meaning can be applied to people, wherein one is trained or perhaps brainwashed into behaving in a certain way. It is theoretically possible to train machines to solve some problems, though a long in depth discussion of genetic programming will have to wait for another day. The current 'Testing Times' series in our members' magazine *CVu* has been skirting around the subject [Polton].

Are programs observable behaviour?

Having touched on Heisenbugs earlier, we realise that the same code doesn't always do the same thing. Sometimes for apparently identical inputs we get non-deterministic outputs. This can be deliberate. For example, in order to approximate a difficult mathematical problem, a program may perform monte carlo simulation [Monte Carlo], producing various different values, driven by a mathematical model of the problem to be solved. The program itself will not be changed between runs, but a random number generator will provide different runs in order to cover the problem space and flush out an answer, by taking an average of the various outcomes. There are a variety of ways to make unchanged code behave in different ways. Many programs will use a configuration file which will point it at different data sources, or invoke different paths through the code allowing a user to flip switches or wave flags without the programmer having to change the code. Even without a configuration file, if a program relies on a data source which can change between runs, the observable behaviour will change even when the code does not. Furthermore, most people have war stories of misuse of a program, with users managing somehow to press an incredible sequence of buttons or provide other heretofore undreamed of inputs causing a spectacular crash. Even without malice or imagination, something very simple, such as attempting to run a program originally targeted at a 32-bit machine on a 64-bit machine can lead to surprises. A program therefore includes its code, its revision history, the bug history, any documentation, possible war stories from those who wrote it *and* the context in which it is run.

Given that the same code can do different things each time it is run, it is also possible to make different code behave in the same way. This might seem like a foolish waste of effort at first sight. Why would anyone rewrite a program in a different language in order to do the same thing? It's not unheard of. Perhaps a team of programmers who were obsessed with C++ template meta-programming have just left (or been sacked) and those who remain only know python, so needs must. More positively, *refactor* – an often overlooked part of the test-driven development cycle – deliberately changes the code to make it easier to maintain, more beautiful, with less repetition, and so on while making sure the observable behaviour stays the same. Is the program the same program after the re-write?

5. Yours truly

Are programs a way of life?

No matter what language has been chosen, or which sets of libraries code uses, or how it has changed through time, it seems a program is more than the sum of its parts. The history of the requirement changes, code refactorings, bug reports, various tests, options in configuration, choice of compiler and dependencies all tell a story. The machines a program will work on, the coding style adopted, the language choices and idioms and manner in which it is released all form part of a program's way of life. A program is far more than just the code you get from source control, assuming you are lucky enough to be in a place that uses source control. The context in which a program works, the manner in which it has been developed and the culture in which it was conceived all influence the code and in turn may influence those who work with the code. In a business, choices along the way were either knee-jerk reactions to crises or considered opinions to avoid crises. For your own personal project that may never leave the privacy of your own computer, there will still be a history and progression of changes and choices – which language, which compiler, is this a learning exercise or a Sudoku solver to stop those pesky kids troubling you? The only constant in a program, or indeed life, is change.

References

[97Things] Scott Meyers in '97 Things every programmer should know' Kevlin Henney, O'Reilly, 2010 http://programmer.97things.oreilly.com/wiki/index.php/Make_Interfaces_Easy_to_Use_Correctly_and_Hard_to_Use_Incorrectly

[Google] Take your pick from googling "I'm like a shark I don't swim backwards" it could be anyone – apparently sharks don't die if they stop swimming.

[Heraclitus] http://en.wikipedia.org/wiki/Heraclitus#Panta_rhei.2C_.22everything_flows.22

[Heisenbugs] http://en.wikipedia.org/wiki/Heisenbug

[Monte Carlo] http://mathworld.wolfram.com/MonteCarloMethod.html

[Polton] 'Testing Times', Richard Polton, *CVu* 25 v 4 and 5

[program] http://www.etymonline.com/index.php?term=program

[SE] http://english.stackexchange.com/questions/80974/where-does-the-phrase-run-code-or-run-software-come-from-why-run

[Stroustrup] http://electronicdesign.com/dev-tools/interview-bjarne-stroustrup-discusses-c

[Tornhill13] Adam Tornhill, 'Code as a Crime Scene' in *Overload* 117, October 2013

On the Other Side of the Barricade: Job Interviewer Do's and Don'ts

Interviewing is an important skill which is hard to get right. Sergey Ignatchenko gives advice to get you thinking.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Bunny, and do not necessarily coincide with the opinions of the translator or the *Overload* editor. Please also keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented providing an exact translation. In addition, both the translators and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Finally, all the years of hard work has paid off, and you've reached a team lead position (which should have been much easier if you'd followed all the advice contained in numerous 'No Bugs' articles ☺). One of the responsibilities which often comes with such a position is conducting technical interviews with job applicants who have passed initial resumé screening and are considered as candidates to join your team. As with many things out there, interviewing is easy to do wrong, and difficult to do right. In short, it is a skill, and an important one (especially if you want your team to perform). Of course, you've been on the other side of the interviewing table dozens of times, but on this side it is very different. So, what can be said about conducting job interviews?

Task definition

First of all, let's try to describe what we're trying to achieve. Of course, the task is 'to tell if the candidate is fit for the job', but it would be great to translate it into something more tangible. Do we really want him to know the API by heart or is just finding stuff she needs, within 30 seconds on the Internet enough? Do we want him to understand how the things are working, or is the 'black-box' approach is good enough? Do we want her to be a quick learner (because our project will take some time to grasp), or we want him to be able to work right away using technology XYZ?

Answers to many such questions depend a lot on the specifics of the project, and on the culture within the team. However, there are several things which interviewer needs to aim for in many cases regardless of specific project:

- we need to find out if the candidate is a 'thinking' developer or 'following-instructions' one (there are positions requiring both types, but placing one type into position which requires another type, is not a good idea)
- we would like to learn if the candidate has an ability to grasp things quickly (it is often very important, especially in agile development environments)
- knowledge of fine details of APIs (systems/protocols/...) by heart is normally not required, so this is what we shouldn't concentrate on; what is usually much more important is ability to find things a developer needs quickly.
- understanding of APIs (systems/protocols/...) is a very different beast, however, if you need somebody to start working quickly (without a learning curve). It does require an understanding of concepts which are relied on in the API/system/protocol, so this is one thing we need to work on during the interview.
  - To illustrate the difference between 'knowledge' and 'understanding', let's assume that you need to hire somebody to work on a Windows client. To do the job, usually the candidate doesn't need to be able to answer questions like "what is parameter #14 of the `CreateFont()` function" (it takes 15 seconds or so to find it in MSDN); however, an understanding of the concept of windows messages, threads where these messages are processed, and sometimes implicit priorities associated with some of these messages can be useful in quite a few projects.
  - and last but not least, we'd like to make sure that good candidate would still want to work in your team after the interview (remember, any interview works both ways – you're assessing the candidate, and she assesses your team based on your actions during the interview)

While this list is by no means exhaustive, I feel that it is a reasonably good starting point to prepare your own list of goals.

Now let's proceed with some do's and don'ts for the job interview (keeping in mind our goals stated above).

Do #-1: Do have a resumé in front of you

It should really go without saying, but you do need a resumé in front of you during the interview. Not only you will need it (you'll see below why), but also not having it (as well asking candidate if he has a copy of his resumé) creates an impression of being unprofessional, which is one thing to avoid (remember that point about candidate assessing your team based on the interview?)

Do #0: Do prepare for your first interview

Conducting job interviews is not that easy and not as fun as it might look when you're sitting on the other side of the table. For your first interview to be meaningful, spend some time, read the candidate's resumé, think about things you would like to know, and about questions you're going to ask. After a few interviews, you'll be able to conduct job interviews without (much) preparation, but for the first one – do yourself (and the interviewee) a favour and come prepared.

Don't #1: Don't try to show you're better than the candidate

I'm important
I'm really important
Did I say I'm important?
I'm really important
~ 'No Bugs' mantra for those suffering from *an in*feriority complex

'No Bugs' Bunny Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 15+ years of industry experience, including architecture of a system which handles hundreds of millions of user transactions per day. He is currently holding the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

The main symptom of this 'boosting self-esteem via demeaning others' syndrome is asking questions nobody of a sane mind can possibly answer

After doing the very basic homework described above, the very first trap a new team lead can fall into when interviewing is to try asserting that he's better than the candidate he's interviewing. It is a Very Bad Thing™ for several reasons, with the main one being "if you're trying to show you're better, you're not doing your job of interviewing". However, I've personally seen this kind of behaviour many times, and it is so common that I simply must caution team leads against it. The main symptom of this 'boosting self-esteem via demeaning others' syndrome is asking questions nobody of a sane mind can possibly answer (unless she accidentally ran into it within last 2 weeks, as happened to the interviewer), or saying that the candidate is wrong not because of substance, but because of terminology (which is always easy when you're interviewing). If you ever find yourself asking such impossible questions or arguing about terminology, you need to sit back and ask yourself, what is the real reason you are doing it?

Those suffering from an inferiority complex (and trying to boost their self-esteem by showing they're better than at least somebody), may console themselves with the thought that the mere fact that it is you interviewing automatically makes you better. While this logic is fundamentally flawed, it is still much better than conducting a job interview with the sole intent to show that the candidate is not on par with (usually the intent is more like showing that the candidate is 'light years behind') the interviewer.

Don't #2: Don't ask questions about the 14th parameter

The second worst thing which can happen during the interview is when the interviewer starts to ask questions about subtle details which nobody really cares about and which can be found in man (MSDN, on the web, etc.) within 30 seconds. One extreme example of such a question is an aforementioned question about the 14th parameter in the `CreateFont()` function. In defense of this approach, I've heard arguments like "if he doesn't know this, why are we even talking?", and I'm sure that both these lines of argument and the questions are deadly wrong. Such questions (especially if accompanied with 'doesn't even know this' logic) often stem from the interviewer working for a long while within one very specific development project, when he needs to remember such things just because this project forces him to do it 15 times a day. However, when interviewing, one needs to remember that people come from different environments, and that those 30 seconds spent searching MSDN will not affect developer performance in any way. Another (and probably main) reason to ask such questions is that they're really really easy to ask (finding smarter questions requires much more thinking); however, this is one case when you get what you've paid for. While smarter questions are indeed much more difficult to find, they're also much more useful for our goals.

Do #1: Do ask questions leading to a dialogue

Remember that we're trying to find if the candidate is able to think, right? The interviewer is trying to determine how to distinguish smart candidates from not-so-smart ones (or, putting the same thing in a politically correct way, how to distinguish very smart candidates from simply smart ones)? Asking multiple-choice questions, and questions which should be answered in a single word, usually does not work well for this purpose; instead, it is much better to ask questions which start dialogue. For example, the question "what does `a+++b` mean?" usually is a pretty lame question (not to mention that it also falls under the 'questions about the 14th parameter' and probably under the 'showing that you're better than candidate' categories), because it is either the candidate knows the answer, or he does not. To compare, a question like: "Can you have a static virtual function?" is much better in this regard, as for any answer (especially for the correct one) you'll be able to ask "Could you explain why?" The answer to this 'why?' question will tell a lot more about the candidate than a dozen '`a+++b`' questions. In practice, this 'static virtual' question lies on the lower end of the spectrum of questions leading to dialogue, and if the candidate is good, one may proceed to more elaborated ones like, "What is the guarantee on insertion into `std::map<>`?" (once again, to be followed by "Could you explain why?").

It is important to remember that reasoning in answers to such questions is much more important than the answer itself. For example, when answering a question about the possibility of inline virtuals, the answer "There is no such thing" is technically wrong, but if the candidate gives a good explanation why he thinks so (such as "because virtual functions should be resolved via virtual table, which won't work for inlines"), it is much better than the simple answer "It is ok" without any evident understanding.

It is also worth noting that potential for the dialogue is heavily context-dependent. The same question about '`a+++b`', if asked when interviewing for a position in compiler development, can easily be a good one, leading to a healthy dialogue about lexical parsers in general and Lex in particular.

Do #2: Do ask questions about previous projects

If the candidate has substantial previous experience, it is usually a very good idea to ask her about previous projects. It can be either specific projects on the resumé which you feel are relevant to what you're doing (or projects that simply look interesting for any reason), or can be a very generic "Could you tell me about the project in your career which you like the most?" In fact, if the candidate is experienced (and assuming you have 15 minutes to spare, which you should), it is almost always a good idea to ask the latter question. You'll see immediately if the candidate is excited about what she's done in that favourite project, and you'll also see if she's excited about the right things. In my experience, there was a candidate rabbit who positioned himself as a senior developer, and who was really excited about the function he wrote to parse e-mail addresses (and it didn't recognize complicated stuff like RFC822 comments and domain literals); well, in my books it didn't really count as a "lifetime achievement to be proud of".

One thing to remember in this regard: if candidate speaks in detail about a project but doesn't mention his role in it, don't forget to ask him.

Do #3: Do ask questions about things on the resumé

If the candidate has mentioned something on the resumé, they should be able to talk about it. Treat the resumé as a carte-blanche to ask about anything the candidate has mentioned. Remember – if he's lied about one thing, he might have lied about the others. While not necessarily fatal, you (both as an interviewer and as a potential team lead) really need to know about it. So, if you see on the resumé something like "in-depth knowledge and 5 years of hands-on experience with boost::containers, such as vector, map, etc.", don't hesitate to ask "Could you remind me if `insert()` can invalidate vector iterators and why?"If this is not answered, just write it down and be much more careful with trusting any other claims in that resumé.

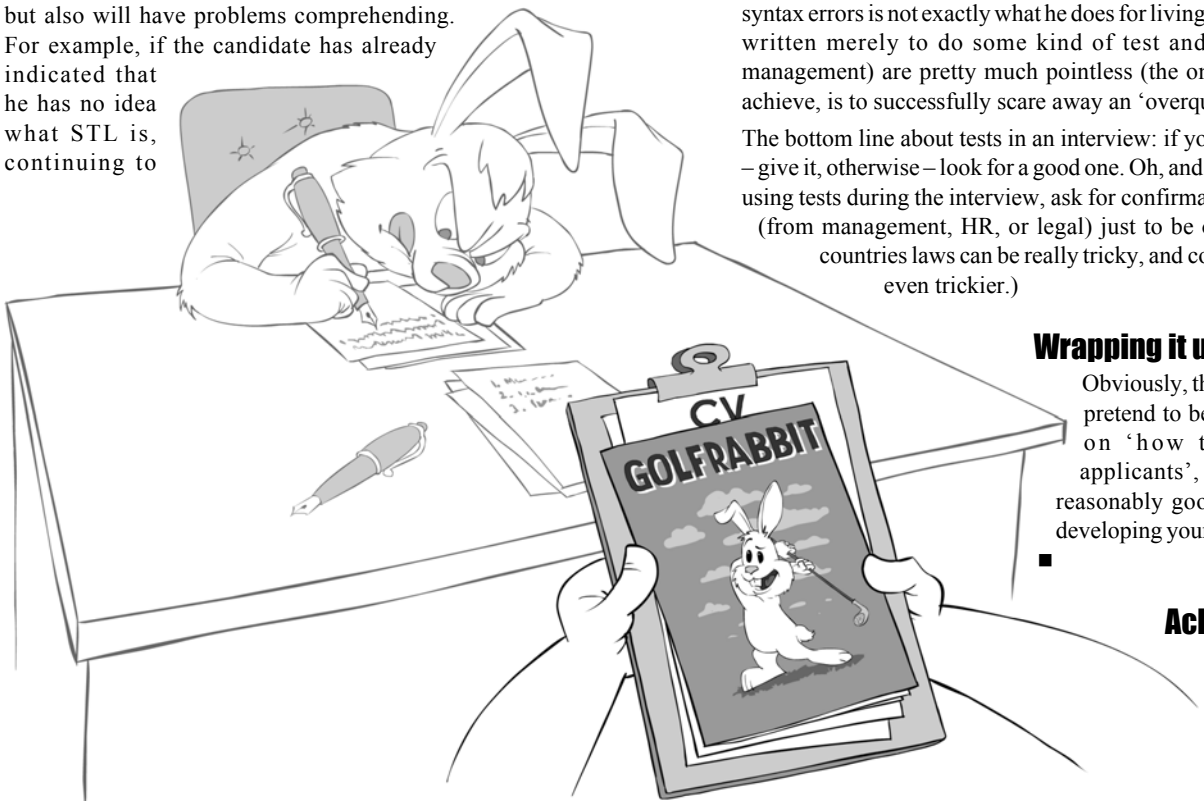
Do #4: Pay attention to candidate communication skills and style

While it is often not too easy for somebody coming from the development side, it is still really important to remember that the candidate is not just a machine to convert coffee into lines of code: she is also a person who you will need to work with, and communicate with. The key questions here are similar to the following: "Can I sit beside this person all day?" "Do I want to go for a drink with them?" However, to be on a safe side, it is better to double-check with your HR if you're allowed to take this kind of things into consideration (as anti-discrimination laws and company policies can be sometimes really ugly in this regard).

Don't #3: Don't ask questions which are too simple or too complicated

It usually happens that after several interviews. You compile a list of your favourite questions, and are following it when conducting subsequent interviews. It is perfectly fine, as long as you're making a sanity check on your standard list depending on the candidate. If you're asking a candidate for a position of architect something like "how much is 0x0A+0x0B?", you're risking that he'll ask himself a different question "Am I sure that I want to work with team lead who asks this stupid question?" This "don't #3" shouldn't be interpreted as "if you're interviewing for the position of architect, restrict your questions to 'how to play golf?'", but questions which do nothing but waste the time of both interviewer and candidate should be avoided.

On the other hand, one also should avoid asking questions which the candidate will not only be unable to answer, but also will have problems comprehending. For example, if the candidate has already indicated that he has no idea what STL is, continuing to questions about guarantees for STL containers is pretty much pointless.



Neither do nor don't: Tests

Good is better than bad
Happy is better than sad
My advice is just be nice
Good is better than bad
~ Pink Dinosaur from Garfield and Friends

On tests during the interviews, there are two common points of view: the first is that a test is the only thing that matters, the second one is that all tests during interviews are evil. As usual, I disagree with both of these ☺. As I see it, good tests are good, and bad tests are bad, but there is a BIG problem with how to write a good test.

The best test I've seen was a task to create a working program for a certain algorithm (the algorithm came with test vectors). The test took several hours, but the developer was allowed to use anything she needed (compiler, debugger, Internet, whatever; there was even an option "you can write it in your favourite programming language"). Some didn't like it, but it has indeed proved to provide an extremely good insight into the developer's style, attention to detail, code structuring, comments, etc. While it was still just an insight and was not 100% conclusive, it was extremely valuable in assessing candidates. Still, there was a major problem even with that test, and it is that the test has eventually leaked into developer community, and creating another meaningful test is really really difficult. And one more thing to remember about such long tests – they can cause substantial problems with candidates reluctant to spend that much of their time with little chance to get something out of it; in the case I'm talking about, it was somewhat mitigated by the test being a part of the second technical interview, with an understanding that company is "almost ready" to make an offer if the test is successful (and also restricting the test to the second interview was rather important to mitigate the leaks).

Another bunch of reasonably good tests was given to me by companies engaged in algorithm development. These usually involved non-trivial sorting, permutations, etc. It is not clear how useful these tests were for potential employees (my guess is that they were – to certain extent), but at least they were fun for me :-).

And the last category of tests was the stuff which shouldn't have been asked in the first place: things like "here is a program, find a bug here" - just to figure out that the task was to find a typo causing syntax error (hint for test writer: a developer is usually using compiler, so finding typos and syntax errors is not exactly what he does for living). Such tests (presumably written merely to do some kind of test and report about it to the management) are pretty much pointless (the only thing such a test can achieve, is to successfully scare away an 'overqualified' candidate).

The bottom line about tests in an interview: if you have a really good one – give it, otherwise – look for a good one. Oh, and a word of caution: before using tests during the interview, ask for confirmation that using tests is ok (from management, HR, or legal) just to be on a safe side. (In some countries laws can be really tricky, and company policies are often even trickier.)

Wrapping it up

Obviously, the present article does not pretend to be a comprehensive guide on 'how to interview IT job applicants', but I hope it may be a reasonably good starting point to start developing your own interview methods. ■

Acknowledgement

Cartoon by Sergey Gordeev from Gordeev Animation Graphics, Prague.

How to Program Your Way Out of a Paper Bag Using Genetic Algorithms

It is often claimed people cannot program their way out of a paper bag. Frances Buontempo bucks the trend using genetic algorithms.

It is frequently suggested that people cannot program their way out of a paper bag, and a quick Google for the state-of-the-art currently reveals nothing until FizzBuzz is added to the search terms. This adds weight to the claim, suggesting people implement FizzBuzz rather than program their way out of a paper bag. I would therefore like to demonstrate one approach to programming one's way out of a paper bag using genetic algorithms. Genetic algorithms, GA, are a form of evolutionary algorithm, drawing on ideas from natural evolution to find solutions to problems. John Holland brought GA to attention with his book, *Adaptation in Natural and Artificial Systems*. They can be used to solve a variety of problems, indeed Holland [Holland92] says

> Computer programs that 'evolve' in ways that resemble natural selection can solve complex problems even their creators do not fully understand

Problem statement

Since the requirements of our current problem are a little vague, this implementation will assume a rectangular, unmoving paper bag of any colour. A projectile will be fired from an imaginary cannon at the bottom centre of the bag at a velocity and angle of elevation to be decided. A projectile is considered 'out' of a paper bag when it has fired over the top of the bag thereby disallowing bursting through the sides or the bottom. If it hits the side or bottom of the bag it will stick there. These decisions are arbitrary and any different assumptions are equally valid and would lead to different solutions. It is possible to solve the problem without an imaginary cannon, but we will leave that for another article.

Clearly, the problem has a closed-form range of solutions for possible angles and required velocities, but this article will show how to discover the solutions programmatically.

Given a bag with bottom left corner at (k, 0), of width w, and height h, assuming the projectile is smaller than the bag, the cannon is a point of no size, and given the acceleration due to gravity, g, after time t the projectile will be at point (x, y) where

$$x = k + \tfrac{1}{2}w + vt\cos(\theta)$$

$$y = vt\sin(\theta) - \tfrac{1}{2}gt^2$$

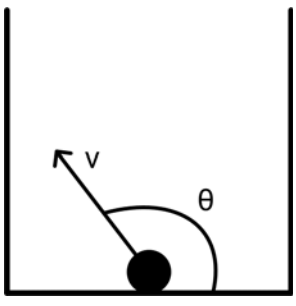


Figure 1

Here x is the horizontal displacement and y the vertical displacement. The projectile will just escape when $y \geq h$ and $x < k$ or $x > k + w$. Clearly as the projectile moves further left or right the height y can be below the height of the bag, h, even if the height was greater than h at the edges since the projectile will start to come down again at some point. g will be taken as 9.81 m/s^2. For simplicity, the code will assume k is zero.

Problem solution

A range of solutions can be found using genetic algorithms, along with other machine learning techniques. Initially, we form a generation of projectiles of some pre-specified size, say n, with random angles, θ, and velocities, v, and fire these from the cannon to see where they end up. We then form a new generation by randomly selecting two of the better parents from the last generation, performing crossover, wherein a child has a v 'gene' from one parent and a θ 'gene' from the other. 'Better' will need to be quantified, though at this stage intuitively a projectile that escapes the paper bag is better than one which does not. If two projectiles fail to escape, we must provide a criterion for choosing the better, though initially we decide they are both unacceptable to simplify things. Pairs of parents breed in this fashion until the next generation also has n items. The genes (angle and velocity) of some randomly selected children may then be mutated, wherein the values are changed slightly. This ensure the generations do not get stuck in the same place and explore more of the solution space. The generations breed for a pre-specified number of steps.

1. Create n ballistics as pairs of random (v, θ)
2. Fire the n ballistics and note which escape
3. For each epoch
   a. Make n new ballistics by:
      If no ballistics escape randomly, create a new pair
      Else randomly choose two parents that escaped, and
         Cross-over: take v from one and θ from the other to create a new ballistic
         Mutate: Possibly randomly change either number by a small amount
   b. Kill off the last generation and repeat with the new gene pool

Implementation in Python

Full details can be found in the code available on github [Github]. The main function, given in Listing 1, runs the pseudo code just presented. The `display_results` function uses MatPlotLib to create the graphs given in the results section.

The initial generation is simply formed by randomly populating a list of angle and velocity tuples as shown in listing 2. The launch function then

Frances Buontempo has a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining. She has been a professional programmer for over 12 years, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com

```
if __name__ == "__main__":
  epochs = 10
  items = 12
  height = 5
  width = 10

  generation = init_random_generation(items)

  generation0 = list(generation)
  results = launch(generation, height, width)
  results0 = list(results)
  for i in range(1, epochs):
    generation = crossover(generation, results,
       height, width)
    mutate(generation)
    results = launch(generation, height, width)

  display_results(generation0, results0,
     generation, results, height, width)
```

Listing 1

```
def init_random_generation(items):
  generation = []
  for i in range(items):
    theta = random.uniform(15, 180) * math.pi/180
    v = random.uniform(2, 20)
    generation.append((theta, v))
  return generation
```

Listing 2

loops from a time, t, of 0 to a hard-coded value of 20 in steps of 0.2, finding the values of x and y for each t from the equations modelling the path of the projectile given earlier. It uses linear interpolation between each time step to decide if the projectile has hit the side of the bag, and makes it stick if it does. For brevity, it is not listed here.

The crossover operation chooses two parents from any projectiles in the previous generation which escaped, as shown in listing 3. If none escaped, then a new random item is created. This may prove to be a foolish decision. This is called in a loop until the next generation is complete.

Results

Results presented here use the parameters epochs = 10, items = 5, height = 5 and width = 10. Initially one projectile escaped, and on the tenth epoch none escaped (see Figure 2). In this instance, few escape initially, which

```
def get_choices(generation, height, width,
    results):
  return [(generation[i][0], generation[i][1]) \
    for i in range(len(generation)) if
      escaped(height, width, results[i]) ]

def crossover(generation, results, height, width):
  choices = get_choices(generation, height,
    width, results)
  if len(choices) == 0:
    return init_random_generation(items)
  next_generation = []
  for i in range(0, len(generation)):
    mum = generation
      [random.randint(0, len(choices)-1)]
    dad = generation
      [random.randint(0, len(choices)-1)]
    t = (mum[0], dad[1])
    next_generation.append(t)
  return next_generation
```

Listing 3

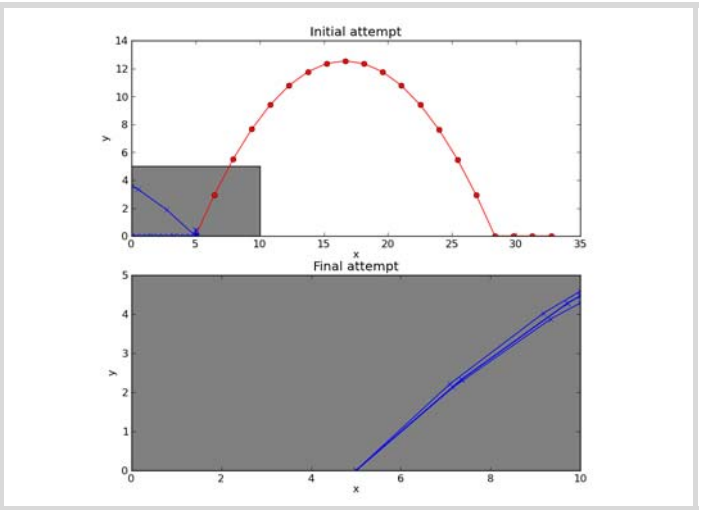


Figure 2

doesn't give enough chance for children forming the next generation to learn, since we have gone for an all or nothing strategy, just choosing projectiles that escape as suitable parents and resorting to new random ones if none escape.

Leaving all parameters the same, but increasing items to 25, better results are obtained. Initially six projectiles escaped, and on the tenth epoch eight escaped (see Figure 3). This improvement may come about because more initial projectiles happened to escape thereby providing more suitable parents. Initial investigations indicate using at least 12 items for the given bag size and number of epochs do well enough.

The crossover function will chose parents from projectiles that escaped. What should be done if no projectiles escaped? In version 1, a new random generation is created. This is wasteful since even when none escape some will still do better than others, for a suitable definition of better, so this approach will lose some of the information the algorithm has discovered. We require a scoring system, known as a fitness function, to decide which pairs are better than others.

Step back and think

We can directly find the height, y, when the x position is at the edge of the bag, rather than the approach taken in phase 1 where interpolation was used at the point just before and then just after the edge of the bag to decide if the height was great enough to escape. This will allow us to ascribe a score for each choice of angle and velocity without looping through each time-

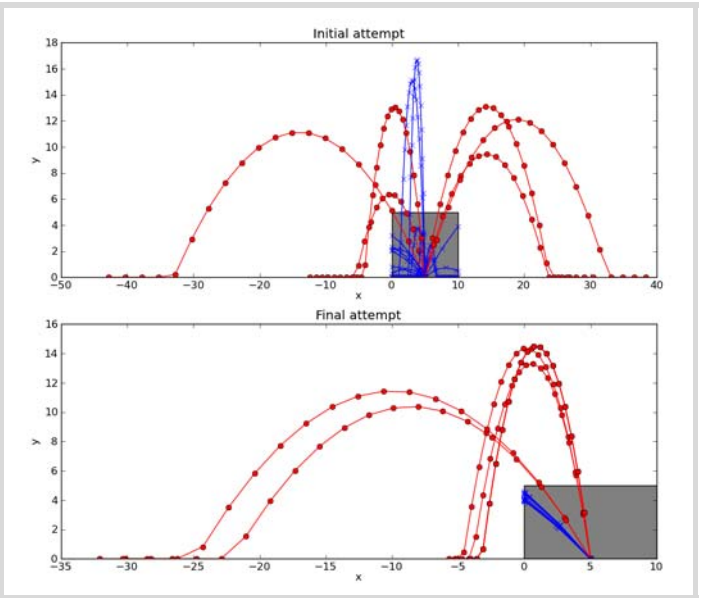


Figure 3

```
def cumulative_probabilities(results):
  cp = []
  total = 0
  for res in results:
    total += res[1]
    cp.append(total)
  return cp

def get_choices(generation, height,
   width, results):
  choices = cumulative_probabilities(results)
  return choices

def choose(choices):
  p = random.uniform(0,choices[-1])
  for i in range(len(choices)):
    if choices[i] >= p:
      return i
  return i

def crossover(generation, results, height, width):
  choices = get_choices(generation, height, width,
     results)
  next_generation = []
  for i in range(0, len(generation)):
    mum = generation[choose(choices)]
    dad = generation[choose(choices)]
    t = (mum[0], dad[1])
    next_generation.append(t)
  return next_generation
```

Listing 4

step and allow us to use the parabolic equation, rather than approximating with linear interpolation. Furthermore Figure 4 shows what can happen when the ballistic just sneaks over the bag. If we interpolate using a straight line between two positions at adjacent time points, the ballistic will appear to hit the edge of the bag, when it in fact would miss.

The main difference from the first approach is a change to the parent selection. The height where the projectile crosses the bag is used as the fitness score of the gene. This means projectiles which get higher are more likely to be chosen as parents, increasing the chance of an item escaping, and using information from items which do not escape, allowing the next generation to learn from the mistakes of its parents. The new approach is shown in Listing 4.

Further results

The results in this section are for 12 items, keeping the bag width and height and number of epochs at the values stated in attempt 1. The first run is a complete success, as shown in Figure 5. With just one initial escape, all of the final generation have been programmed out of the paper bag, in stark contrast with the first approach.

The second run, Figure 6, demonstrates that even if no projectiles escape initially the algorithm still finds successfully genes. It can use parameters

Linear interpolation of a curve may lead to incorrect decisions about whether a projectile has gone over the top of the paper bag or broken through the side

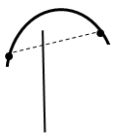


Figure 4

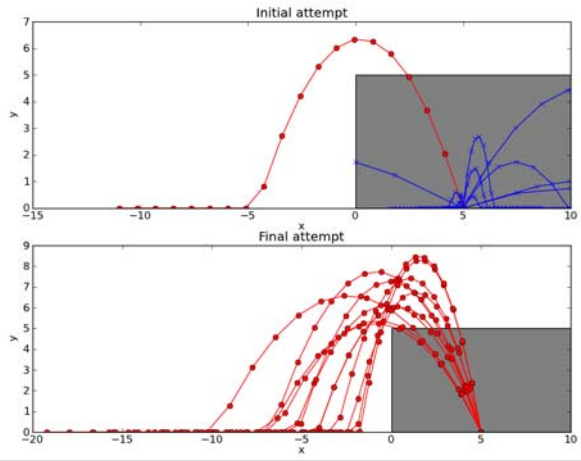


Figure 5

from the better projectiles, i.e. those which got higher up the bag if they hit the edge of the bag, allowing them to escape in future generations.

Conclusion

Using the idea of programming one's way out of a paper bag can provide a variety of ways to learn a language and a new algorithm. This article has demonstrated how to use genetic algorithms to escape from a paper bag. The results presented could do with more rigorous analysis, for example to demonstrate an improvement between the techniques, the results of several runs should be compared. Ultimately, discovering a range of solutions for possible angles and required velocities or finding parameters which allow all the projectiles to escape from the paper bag would fully solve the problem. We could also consider what happens on other planets, specifically varying the value of gravity. The purpose of this article has been to demonstrate one over-engineered approach to programming one's way out of a paper bag. There are, of course, many possible simpler approaches to the problem, but it is hoped this will serve as a gentle introduction to genetic algorithms and will inspire others to try to programme their way out of a paper bag. ■

References

[Github] https://github.com/doctorlove/paperbag/tree/master/ga

[Holland92] *Scientific American*, Vol. 267 (1992), pp. 66–72

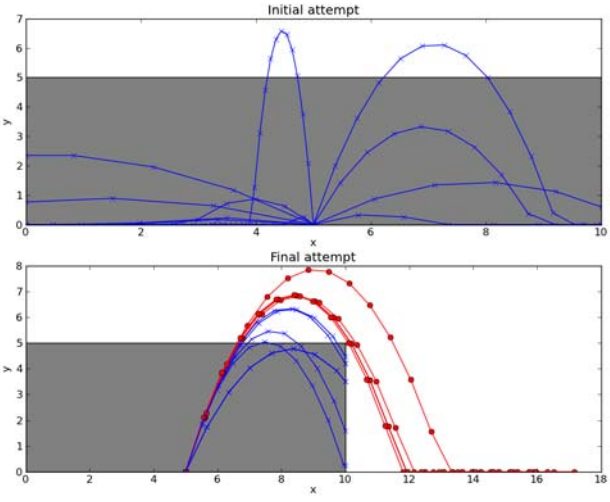


Figure 6

Object-Environment Collision Detection using Onion BSPs

Previously we considered 3D navigation. Stuart Golodetz demonstrates how to detect collisions using onion binary space partitioning.

In my last article [Golodetz13], I described how to automatically generate navigation meshes to support the navigation of agents around 3D environments (e.g. game worlds), as implemented in my homemade *hesperus* engine [hesperus]. However, there is far more to such navigation than simply mesh generation: it remains to be shown how to determine where (if anywhere) an agent can be found on the mesh and how to make best use of the mesh when allowing both user-controlled and AI agents to move around the environment. Agent movement must necessarily interact with an implementation's physics system, since the navigation mesh only covers the walkable surfaces of the world and there is a need to ensure that agents are simulated correctly even when they are not on the mesh. In particular, any implementation needs to ensure that agents do not collide with either the world or each other, and that the effects of forces such as gravity are properly applied to them when not on the mesh. For that reason, before tackling the agent movement problem itself, it is important to take a step back and look at how the physics system in *hesperus* works.

As a first step, I want to focus this article on a way of detecting collisions between objects (including agents) and their environment, via the construction of a special binary space partitioning (BSP) representation of the world that I call an *onion BSP* (for reasons that will be explained). Onion BSPs are a simple extension of BSP trees for multiple configuration spaces, based on the ideas of van Waveren for *Quake III Arena* in [VanWaveren01]. The collisions (also known as contacts) that we detect can be fed to the rest of the physics system for later resolution. Future articles will focus on how to detect object-object collisions using a technique called Minkowski Portal Refinement [Snethen08], and how to combine the techniques into a rudimentary physics system, before we return to the original problem of agent movement. Readers who are interested in a more general look at games physics engine development are advised to take a look at the excellent (and aptly-named) book by Millington on the topic [Millington07].

The organisation of this article is as follows: (a) I briefly revisit the ideas behind binary space partitioning; (b) I describe how to construct onion BSPs; (c) I describe how to perform (swept) collision detection between objects and onion BSPs using an algorithm for finding the first point at which a half-ray crosses a wall in the world; and (d) I discuss the limitations of this approach and briefly compare it to a related approach that achieves the same effect by moving the planes of a normal BSP at runtime.

Binary space partitioning

Binary space partitioning is a technique for representing n-dimensional space as a binary tree (known as a BSP tree) by recursively dividing it into two using *hyperplanes* (the n-dimensional generalisation of planes). It was originally introduced by Fuchs et. al. [Fuchs80] in 1980, and saw widespread use in first-person games of the *Quake* era (e.g. see [Abrash97]) as a way of representing 3D polygonal game worlds, most notably because it provided a way of rendering a world's polygons in either back-to-front or front-to-back order [Fuchs80, Gordon91] without the need for a z-buffer on the graphics card (z-buffers once used to be quite costly). As graphics cards have matured, commercial games have moved away from binary space partitioning as a rendering approach because traversing a BSP tree is relatively slow in comparison to simply throwing large numbers of triangles at the graphics card and letting the z-buffer handle the rendering order, but BSP trees remain interesting as a basis for collision detection and constructive solid geometry techniques [Ericson05, Lysenko08].

An example BSP tree is shown in Figure 1. Each node of the tree represents a convex subspace of the world being partitioned; moreover, the leaves of the tree represent a *partition* of the entire space, i.e. they are mutually disjoint and their union is equal to the space. Each branch node has an associated split plane (a line with facing in 2D) that divides the subspace represented by the node in two. Each leaf node contains the polygons (line segments in 2D) that fall within the subspace it represents, and carries a flag that indicates whether the subspace represented by the leaf is empty (i.e. navigable by an agent, denoted as $\perp$) or solid (non-navigable, denoted as $\top$). The BSP tree as a whole can be used to decide whether or not any given point in the world lies in empty or solid space in O(h) time, where h is the height of the tree, by the simple means of classifying the point against the split planes in the tree, starting from the root, and recursing down the relevant side of the tree at each stage until hitting a leaf.

Constructing a BSP tree for a polygonal world is also done recursively, starting from the set of all the polygons in the world. At each recursive step, one of the current set of polygons whose plane has not been used further up the tree is chosen as the *split polygon*, and its plane is used to split the other polygons into two sets, one of polygons that are in front of the plane and one of those that are behind it. (If no suitable split polygon can be found, then we create a leaf of the tree to contain the current set of polygons and return.) If a polygon straddles the plane, it is split, with its two halves being placed in the appropriate sets. If a polygon lies on the plane, it is put into either the front or back set based on the orientation of its normal with respect to the plane. The two sets of polygons are then processed recursively to construct the subtrees of the current node. Finally, a branch node is constructed from the split plane and the two subtrees.

An extremely detailed description of BSP construction, together with diagrams that clarify precisely how the process works, can be found in [Golodetz06]; readers may additionally wish to take a look back at a previous article I wrote for *Overload* [Golodetz08].

Onion BSPs

As mentioned in the previous section, standard BSP trees can be used to determine whether individual points are in empty or solid space; moreover, this extends to line segments – there is a relatively straightforward BSP

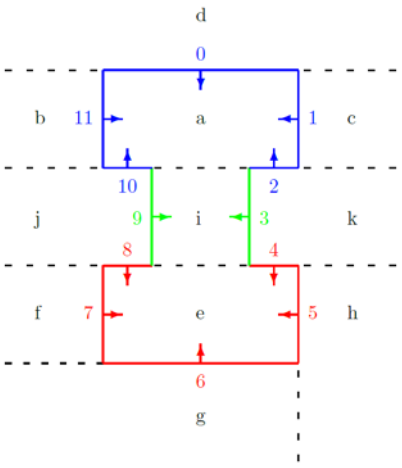
Stuart Golodetz obtained his DPhil in Computer Science in 2011, working on 3D image segmentation and feature identification. Following two years in credit risk management, logic programming and software analytics, he is working on object detection and tracking for an assisted vision project. His areas of interest include image processing, computer games development and the intricacies of different programming languages, especially C++.

algorithm that will allow us to find the first transition point at which a line segment crosses from empty to solid space (e.g. see [Arvo88, Jansen86, Sung92]). This can form the basis for a simple collision detection scheme for *point-based* agents – at each frame, we can test the line segment representing an agent's proposed movement for that frame against the tree, taking the first transition point as the point of collision if the agent tries to walk into a wall.

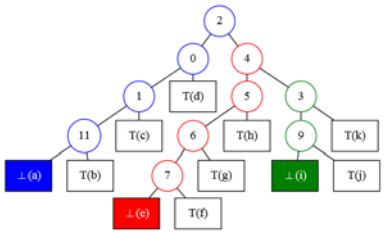
Unfortunately, however, most agents in 3D games are not point-based, so we need a way to handle objects with extent. The way I describe here is due to van Waveren [VanWaveren01] and uses the notion of configuration spaces I described in [Golodetz13]. An alternative, similar approach, that works by modifying a normal BSP at runtime, is mentioned in the 'Discussion' section. Both of the approaches described are based on the same principle – that performing collision detection between an object with extent and the world is equivalent to performing collision detection between a point at the centre of the object and a copy of the world that has been suitably expanded in accordance with the size of the object.

A BSP example for a simple 2D world with two rooms, connected by a corridor: (a) shows a top-down view of the world, where the arrows represent the facings of the world polygons and the dashed lines represent the split planes chosen when constructing the BSP in (b); (b) shows the BSP tree that is constructed for the world based on the chosen split planes; (c) shows what a 3D version of the world looks like in hesperus, with portals (doorways) rendered as translucent polygons to illustrate the boundaries between the empty leaves (a, e and i) of the BSP. (Note that the 3D version actually has additional floor and ceiling polygons, but we ignore that here for the purposes of explanation.)

(a)



(b)



(c)

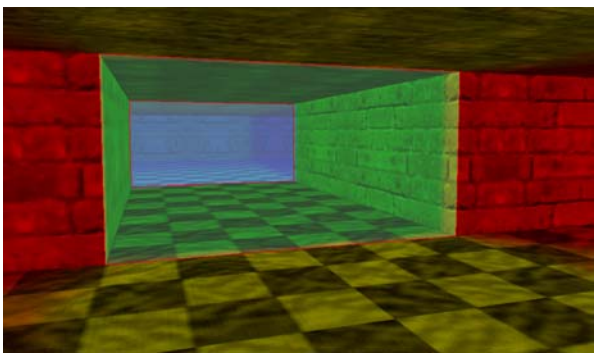
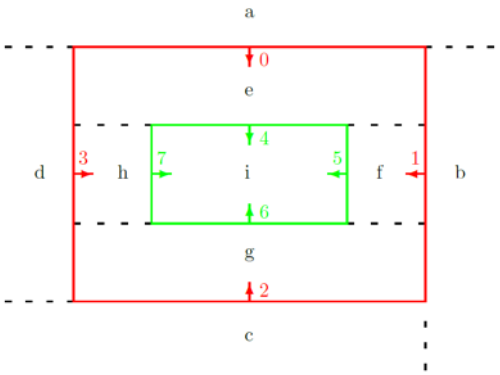


Figure 1

An example 2D world (a) and one possible onion BSP for it (b). The solid rectangles denote two separate configuration spaces (an outer one and an inner one). Their polygons (shown as numbered, oriented line segments) are compiled into the same onion BSP as shown. Individual leaves (labelled with letters) can be empty (⊥) in one space and solid (T) in another, e.g. leaf e is empty in the outer space but solid in the inner one.

(a)



(b)

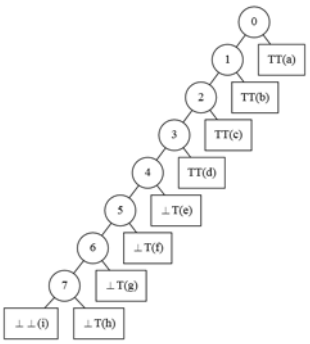


Figure 2

The van Waveren approach is an offline method designed for brush-based 3D environments (that is, environments built up by combining simple, convex polyhedra). Agents are represented by axis-aligned bounding boxes (AABBs); each class of agent may have multiple AABBs for different poses (e.g. standing or crouching). At level compilation time, the brushes of the environment are expanded for each AABB and the faces of the expanded brushes are unioned together to form an expanded world for that AABB. Each of these expanded worlds can be compiled into a BSP tree, allowing us to perform collision detection for an object represented by the corresponding AABB. However, maintaining multiple BSP trees is inconvenient because then objects that are in the same physical location but have different sizes cannot be resolved to a leaf in any particular tree – we would much prefer to have a single tree that represents all of the information available.

We can achieve this by constructing a different type of BSP tree that I call an *onion BSP*[1]. Onion BSPs are a generalisation of standard BSPs in which we replace the empty/solid flag in each leaf node with a vector of flags indicating whether the leaf is empty/solid in each configuration space associated with an AABB. Figure 2 shows two configuration spaces generated for an example world (the original, unexpanded world is not shown) and an onion BSP that might be generated for it.

1. For the interested reader, the name 'onion BSP' comes from the idea that the expanded worlds look rather like the layers of an onion when superimposed in an image. This analogy is not strictly accurate, because the various different AABBs will not, in general, nest inside each other, but the name is nevertheless both convenient and suggestive.

```
function build_tree
:   (polys: Vector<Polygon>) -> OnionBSPTree

var nodes: Vector<Node>;
var ancestors: Vector<Plane>;
var polyIndices: Vector<PolyIndex> := {(i,true) |
0 <= i < |polys|};
build_subtree(ref polys, polyIndices, ref nodes,
ref ancestors);
return make_onion_bsp(nodes);

class PolyIndex
  var index: int;
  var splitCandidate: boolean;
```

Listing 1

The compilation process

Onion BSP compilation is in principle much the same as BSP compilation (see the 'Binary space partitioning' section), but slightly trickier because we have to test the solidity of each leaf in each configuration space rather than getting it for free as part of the compilation process. An explanation of this testing process is deferred to the next section, but it also has an impact on the main part of the compilation. In particular, the test involves checking an arbitrary point in the leaf for solidity in each configuration space (the solidity of any point in the leaf is guaranteed to be the same as that of the entire leaf), so we will need (a) a way of determining an arbitrary point in a leaf, and (b) a way of testing a point for solidity in a configuration space. As will be seen, determining an arbitrary point in a leaf will involve knowing the set of split planes on the path from the root of the tree to the leaf, so these should be maintained during compilation.

The resulting main compilation process is shown in Listings 1 and 2. The key thing to note is the way in which a set of split planes is maintained in order to facilitate solidity testing – we add the current split plane to the set before each recursive call to `build_subtree` and remove it again afterwards, so that whenever we reach a leaf it will contain precisely those split planes on the path from the root of the tree to the leaf. Note that orientation is important, so the current split plane must be reversed when recursing into the right-hand subtree.

Determining leaf solidity

A *solidity descriptor* for a leaf in an onion BSP is a vector of flags indicating whether the leaf is empty or solid in each configuration space for which we compiled the BSP. It is common for a leaf to be empty in one configuration space and solid in another – for example, a leaf might be empty in the configuration space corresponding to the crouch pose of an agent, but solid in the configuration space corresponding to the standing pose, indicating that the agent can traverse the leaf whilst crouching but not whilst standing (e.g. think of a low tunnel). To determine a leaf's solidity descriptor, we find an arbitrary point in the leaf and check its solidity in each configuration space in turn; the resulting empty/solid results are combined to form the full solidity descriptor. To test points' solidity in a configuration space, we build a normal BSP tree (called a *map tree* in the code) for the space at the start of the compilation process and later classify any relevant points with regard to it. The top-level process to determine leaf solidity is shown in Listing 3.

Finding an arbitrary leaf point

To find an arbitrary point in a leaf, recall that each leaf represents a convex subspace of the world. Our first intuition might be to create an explicit representation of the leaf as a convex polyhedron and then compute the average of the midpoints of the polyhedron's faces as our point. This does in fact work perfectly for fully-bounded leaves (see Figure 3(a)), but unfortunately fails for unbounded ones (see Figure 3(b)). Fortunately, in practice, there is an easy solution to this problem: we can simply stipulate that the world we are representing is bounded by an inward-facing box, thereby ensuring that every leaf is bounded (see Figure 3(c)). This is clearly

```
function build_subtree
:  (polys: ref Vector<Polygon>;
    polyIndices: Vector<PolyIndex>;
    nodes: ref Vector<Node>;
    ancestors: ref Vector<Plane>) -> Node

var splitter: Plane :=
  choose_splitter(polyIndices);

// If there were no suitable split candidates,
// this is a leaf.
if splitter = null then
  var solidity: DynamicBitset :=
    determine_leaf_solidity(ancestors);
  var indicesOnly: Vector<int> := {i | (i,_) in
    polyIndices};
  nodes.push_back(Leaf(|nodes|, solidity,
    indicesOnly));
  return nodes.back();

var backPolys, frontPolys: Vector<PolyIndex>;
for each pi@(index, splitCandidate) in
   polyIndices
  var poly: Poly := polys[index];
  switch classify_against_plane(poly, splitter)
    case CP_BACK:
      backPolys.push_back(pi);
      break;
    case CP_COPLANAR:
      if splitter.norm().dot(poly.norm()) > 0 then
        frontPolys.push_back((index, false));
      else
        backPolys.push_back((index, false));
      break;
    case CP_FRONT:
      frontPolys.push_back(pi);
      break;
    case CP_STRADDLE:
      (back, front) := split_poly(poly, splitter);
      polys[index] := back;
      polys.push_back(front);
      backPolys.push_back(pi);
      frontPolys.push_back((|polys| - 1,
        splitCandidate));
      break;

ancestors.push_back(splitter);
var left: Node := build_subtree(frontPolys,
   nodes, ancestors);
ancestors.pop_back();

ancestors.push_back(splitter.flipped());
var right: Node := build_subtree(backPolys,
   nodes, ancestors);
ancestors.pop_back();

var subRoot: Node := Branch(|nodes|, splitter,
   left, right);
nodes.push_back(subRoot);
return subRoot;
```

Listing 2

a reasonable assumption in the context of representing a 3D world, since it would not be meaningful for such a world to be infinite. (The interested reader may wish to take a look at [Seidel91], where a similar approach is taken to deal with unboundedness in a related linear programming problem.)

The algorithm itself is shown in Listing 4. It is called with the set of ancestor planes leading down to the given leaf in the tree (recall that these are maintained as part of the top-level compilation process). These are

```
function determine_leaf_solidity
:   (ancestors: Vector<Plane>) -> DynamicBitset

// Assumed available from elsewhere:
// * mapTrees: Vector<BSPTree>

// Find an arbitrary point within the leaf with the
// specified ancestor planes.
var p: Vec3 := arbitrary_leaf_point(ancestors);

// Classify the point against each map tree to
// determine the solidity descriptor for the leaf.
var solidity: DynamicBitset(|mapTrees|);
for each mt_i in mapTrees
  var leaf: BSPLeaf := mt_i.find_leaf(p);
  solidity[i] := leaf.is_solid();

return solidity;
```

Listing 3

augmented with the planes of the inward-facing box that we are assuming bounds the world. We then construct an extremely large polygon on each of the planes in turn, and clip it to the other planes (see [hesperus] for the implementation details). The set of polygons that results forms a convex polyhedron representing the (bounded) leaf. As previously stated, we finally compute the midpoint of each face of the polyhedron and average them to produce an arbitrary point that is guaranteed to be inside the leaf.

Finding an arbitrary point in a leaf of a simple 2D world with a single room (drawn as a square). In (a), we can successfully build a convex polyhedron for the bounded leaf representing the room itself and then average the midpoints of the polyhedron's faces (the dots on the square) to find a suitable point (the dot in the centre). In (b), the same procedure fails for the unbounded leaf behind the room's topmost wall. In (c), we rectify the problem by adding bounding planes around the world as a whole. This ensures that all of the leaves are bounded, allowing the method to work.

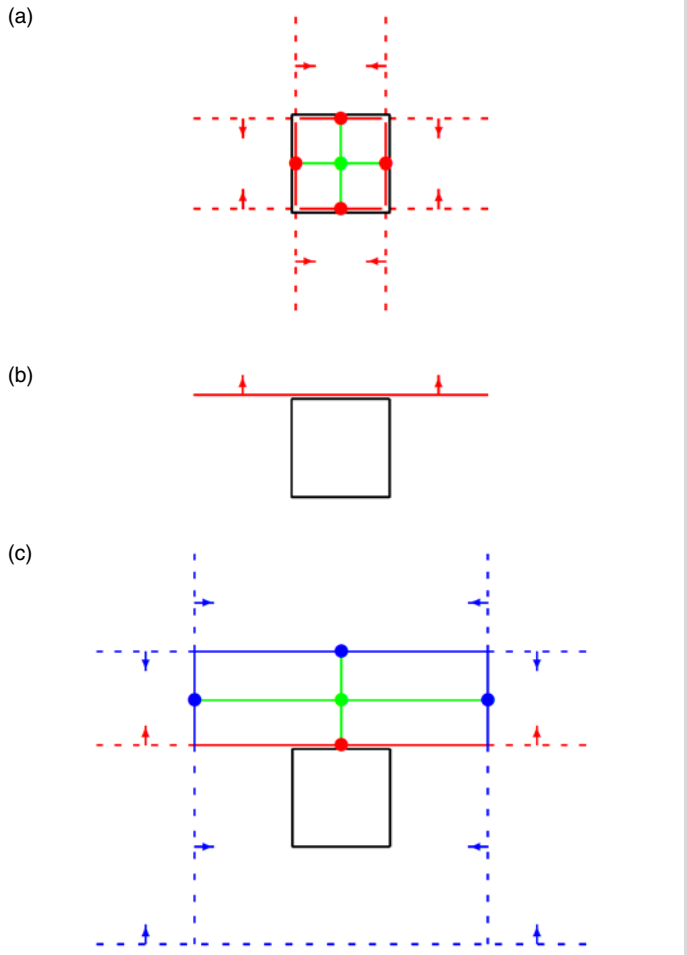


Figure 3

Collision detection

Recall that our goal is to detect collisions between moving objects of various sizes and a stationary world. The desired output of our collision detection approach is a set of collisions (or contacts), each of which is specified by a *collision point* (a first point at which the moving object touches the world), a *collision normal* (the normal of the surface that is hit by the moving object) and a *collision time* (a number in the range [0,1] indicating at what point during the movement the collision occurs).

Having constructed an onion BSP for the world, it is now possible to perform collision detection against it for objects with a specific AABB, using a variant of the 'find first transition' algorithm mentioned in the 'Onion BSPs' section (see Listing 5). The key difference from the version for normal BSPs is that the leaf solidity test at the top of the **`fft_sub`** function is performed for a specific configuration space (e.g. one corresponding to an agent's crouching pose); in all other respects the two are essentially the same.

The algorithm is initially called on the movement ray (which is just a line segment) of an agent for the current frame and the root node of the onion BSP, and proceeds recursively, ultimately producing a 'transition' to indicate its result (transitions are either (a) **`RAY_E`**, indicating that the entire movement ray is in empty space, (b) **`RAY_S`**, indicating that the entire movement ray is in solid space, or (c) a triple (**`RAY_E2S`** or **`RAY_S2E`**, **`point`**, **`splitter`**), indicating that the movement ray first transitions from empty to solid, or solid to empty, space at the specified point on the specified split plane). At each recursive step, the relevant segment of the movement ray (initially, all of it) is classified against the split plane of the current node, and appropriate action is taken based on the result. If the movement ray segment is entirely on one side of the plane, we recurse down that side of the tree. If the movement ray segment is on the plane (the coplanar case), we pass it down both sides of the tree and subsequently combine the results. If the movement ray segment straddles the plane, we split it and pass the half of it near the start of the segment down the corresponding side of the tree. If this yields a non-trivial transition, we return it; otherwise, we pass the other half down the far side of the tree, and subsequently derive the result as shown in Listing 5. When we eventually reach a leaf, we return a transition based on the solidity of the leaf in the configuration space in which we are interested (this can be specified as an additional parameter to the algorithm, or provided by some other means). A few of the recursive cases are illustrated in Figure 4.

As mentioned, the ultimate result of the 'find first transition' algorithm is either a trivial transition (the movement ray is entirely in empty or solid space) or a non-trivial one; in the latter case, the point and the normal of the split plane found can be used directly as the *collision point* and *collision normal* for a detected collision. The *collision time* can be calculated using simple ratios as follows. Denote the source and destination endpoints of the movement ray as $\vec{\mathbf{s}}$ and $\vec{\mathbf{d}}$ respectively, and the collision point as $\vec{\mathbf{c}}$. Then the collision time is given by:

$$t = \frac{\left|\vec{\mathbf{c}} - \vec{\mathbf{s}}\right|^2}{\left|\vec{\mathbf{d}} - \vec{\mathbf{s}}\right|^2}$$

The case of a trivial transition that lies entirely in solid space needs special handling to ensure robustness. In practical terms, this can very occasionally happen due to rounding errors when the source endpoint of the movement ray is on an empty/solid boundary. A simple way of dealing with the issue is to repeat the find first transition call with a source endpoint that is moved back from the boundary by a small amount:

$$\vec{\mathbf{s}}' = \vec{\mathbf{s}} - \varepsilon(\vec{\mathbf{d}} - \vec{\mathbf{s}})$$

The collisions we generate are fed into the physics system for later resolution. I will explain how this works in a future article.

```
function arbitrary_leaf_point
:  (ancestors: Vector<Plane>) -> Vec3

// Step 1: Make an inward-facing convex polyhedron
// around the leaf.

// Make an array of possible bounding planes: these
// are the ancestor planes themselves, plus the
// planes that bound the 3D world. The planes are
// specified as ax + by + cz - d = 0.
const HALFWORLDBOUND: double := 100000;
var planes: Vector<Plane> := ancestors;
planes.push_back(Plane((1,0,0),
  -HALFWORLDBOUND));
planes.push_back(Plane((-1,0,0),
  -HALFWORLDBOUND));
planes.push_back(Plane((0,1,0),
  -HALFWORLDBOUND));
planes.push_back(Plane((0,-1,0),
  -HALFWORLDBOUND));
planes.push_back(Plane((0,0,1),
  -HALFWORLDBOUND));
planes.push_back(Plane((0,0,-1),
  -HALFWORLDBOUND));

var faces: Vector<Poly>;
for each p_i: Plane in planes
  // Build a large initial face on each plane.
  var face: Poly := make_large_poly(p_i);

  // Clip it to the other planes.
  var discard: bool := false;
  for each p_j: Plane in planes
    if j = i then continue;
    switch classify_against_plane(face, p_j)
      case CP_BACK:
        // Face entirely out of leaf.
        discard = true;
        break;
      case CP_COPLANAR:
        // Shouldn't happen: ancestors are unique.
        throw "Unexpected duplicate plane";
      case CP_FRONT:
        // Face entirely in leaf.
        continue;
      case CP_STRADDLE:
        // Part of face in leaf, part not.
        (_,front) := split_poly(face, p_j);
        face := front;
        break;

    if discard then break; // early out

  // Add surviving faces to the array.
  if not discard then faces.push_back(face);

// Step 2: Compute the average of the polyhedron
// face midpoints.
var denom: int := 0;
var p: Vec3(0,0,0);
for each face: Poly in faces
  for each v in face.vertices()
    p := p + v;
    denom := denom + 1;

assert denom != 0;
p := p / denom;
return p;
```

Listing 4

```
function fft_sub: (src: Vec3; dest: Vec3; node:
Node) -> Transition

// Assumed available throughout:
// * cSpace: int [the configuration space index]

var leaf: Leaf := node.as_leaf();
if leaf != null then
  return leaf.is_solid(cSpace) ? RAY_S : RAY_E;
var br: Branch := node.as_branch();
var left, right: Node := br.left(), br.right();
var splitter: Plane := br.splitter();
var cpSrc, cpDest: PlaneClassifier;
switch classify_against_plane
(src, dest, splitter, ref cpSrc, ref cpDest)
  case CP_BACK:
    return fft_sub(src, dest, right);
  case CP_COPLANAR:
    var trLeft: Transition := fft_sub(src, dest,
      left);
    var trRight: Transition := fft_sub(src, dest,
      right);
    if trLeft.class = trRight.class then
      switch trLeft.class
        case RAY_E|RAY_S:
          return trLeft;
        default:
          var dLeft: double :=
            |src - trLeft.loc|^2;
          var dRight: double :=
            |src - trRight.loc|^2;
          return dLeft < dRight ? trLeft
            : trRight;
    else if trLeft.class = RAY_E2S|RAY_S2E then
      return trLeft;
    else if trRight.class = RAY_E2S|RAY_S2E then
      return trRight;
    else return RAY_E;
  case CP_FRONT:
    return fft_sub(src, dest, left);
  default: // case CP_STRADDLE
    var mid: Vec3 := intersect(src, dest,
      splitter);
    (near,far) := cpSrc = CP_FRONT ? (left,right)
      : (right,left);
    var trNear: Transition := fft_sub(src, mid,
      near);
    if trNear.loc != null then return trNear;
    var trFar: Transition := fft_sub(mid, dest,
      far);
    switch trFar.class
      case RAY_E:
        return trNear.class = RAY_E ? RAY_E :
          Transition(RAY_S2E, mid, splitter);
      case RAY_S:
        return trNear.class = RAY_S ? RAY_S :
          Transition(RAY_E2S, mid, splitter);
      case RAY_E2S:
        return trNear.class = RAY_E ? trFar :
          Transition(RAY_S2E, mid, splitter);
      default: // case RAY_S2E
        return trNear.class = RAY_S ? trFar :
          Transition(RAY_E2S, mid, splitter);
```

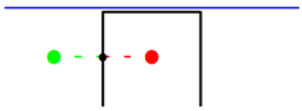
Listing 5

Discussion

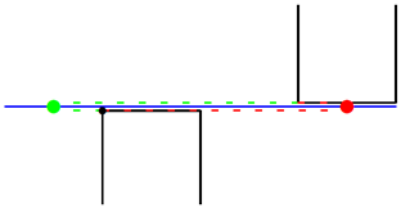
The approach that I have described thus far is a (comparatively) simple and effective way of detecting collisions between moving objects and a stationary world, but unsurprisingly it does have some limitations. One

A few of the recursive cases for the 'find first transition' algorithm, illustrated on a movement ray from an agent's current position in empty space (shown in green/light grey) to its attempted position in solid space (shown in red/dark grey).

(a) The movement ray is entirely on one side of the plane (the long horizontal line): recurse down that side.



(b) The movement ray lies in the plane itself: recurse down both sides and combine the results. In this case, both sides have a transition, so we take the nearer one.



(c) The movement ray straddles the plane: split it and recurse down the near side first. In this case, the near half is entirely in empty space, so we then recurse down the far side to find the transition point.

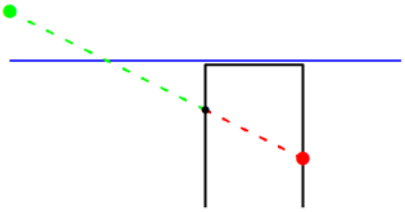


Figure 4

potential drawback is that it is designed to work for a small number of object sizes that are known at level compilation time: this was not a problem for games such as *Quake III*, but it makes the technique less suitable for games that want to contain a wide variety of differently-shaped characters, since compiling large numbers of different configuration spaces into an onion BSP would severely bloat the tree and lead to slow level compilation times and poor performance. Another drawback is that it only works for objects that do not rotate: games that want to support more realistic physical simulation have to use more complicated approaches (e.g. see [Ericson05, Millington07]).

A related approach that eliminates the first of these limitations, whilst still providing acceptable performance, was presented by Melax in [Melax00]. The details can be found in that article, but the essence of the approach is to replace the 'find first transition' algorithm with a variant that dynamically moves the planes of a normal BSP for the world during ray testing so as to simulate the configuration spaces for different types of object. This avoids the need to know the sizes of the objects up-front, at the cost of making ray testing somewhat more costly. The approach was used successfully in the BioWare game MDK2.

Conclusions

In this article, I have described a simple and effective technique (due to van Waveren) for detecting collisions between moving objects and their surrounding 3D environment. While there are important limitations to this technique in the context of realistic physical simulation (most notably the fact that it only works for non-rotating objects), it has proved useful in a games context because it can detect collisions for translating objects quickly and accurately. The alternative approach (due to Melax) mentioned in the 'Discussion' section builds upon this technique by allowing large numbers of differently-shaped characters to be handled without bloating the tree.

It remains to be shown how to detect inter-object collisions and how to build a working physics system, which I hope to address in future articles. We can then return to our original problem of agent movement, using the physics system and the environment's navigation mesh in tandem. ■

Acknowledgements

I would particularly like to thank the editorial team for the effort that has gone into typesetting this article for publication. Many thanks also to the rest of the Overload team for reviewing this article and suggesting ways in which to improve it.

References

[Abrash97] *Michael Abrash's Graphics Programming Black Book*. Michael Abrash. Coriolis Group Books, special edition, 1997.

[Arvo88] Linear-Time Voxel Walking for Octrees. Jim Arvo. *Ray Tracing News*, 1(12), March 1988.

[Ericson05] *Real-Time Collision Detection*. Christer Ericson. Morgan Kaufmann, 2005.

[Fuchs80] On Visible Surface Generation by A Priori Tree Structures. Henry Fuchs, Zvi M Kedem and Bruce F Naylor. *Computer Graphics*, 14(3):124-133, 1980.

[Golodetz06] *A 3D Map Editor*. Stuart Golodetz. Undergraduate thesis, Oxford University Computing Laboratory, May 2006.

[Golodetz08] Divide and Conquer: Partition Trees and Their Uses. Stuart Golodetz. *Overload*, 86:24-28, August 2008.

[Golodetz13] Automatic Navigation Mesh Generation in Configuration Space. Stuart Golodetz. *Overload*, 117:22–27, October 2013.

[Gordon91] Front-to-Back Display of BSP Trees. Dan Gordon and Shuhong Chen. *IEEE Computer Graphics and Applications*, 11(5):79-85, 1991.

[hesperus] The hesperus 3D game engine. Stuart Golodetz. Source code available online at: https://github.com/sgolodetz/hesperus2.

[Jansen86] Data structures for ray tracing. Frederik W Jansen. In Laurens R A Kessener, Frans J Peters and Marloes L P van Lierop, editors, *Data Structures for Raster Graphics*, pages 57–73. Springer-Verlag Berlin Heidelberg, 1986.

[Lysenko08] Improved Binary Space Partition Merging. Mikola Lysenko, Roshan D'Souza and Ching-Kuan Shene. *Computer-Aided Design*, 40(12):1113-1120, 2008.

[Melax00] Dynamic Plane Shifting BSP Traversal. Stan Melax. *Graphics Interface*, 2000:213-220, 2000.

[Millington07] *Game Physics Engine Development*. Ian Millington. Morgan Kaufmann, 2007.

[Seidel91] Small-Dimensional Linear Programming and Convex Hulls Made Easy. Raimund Seidel. *Discrete & Computational Geometry*, 6(1):423-434, 1991.

[Snethen08] XenoCollide: Complex Collision Made Simple. Gary Snethen. In Scott Jacobs, editor, *Game Programming Gems 7*, pages 165–178. Charles River Media, 2008.

[Sung92] Ray Tracing with the BSP Tree. Kelvin Sung and Peter Shirley. In David Kirk, editor, *Graphics Gems III*, pages 271–274. Morgan Kaufmann, 1992.

[VanWaveren01] *The Quake III Arena Bot*. Jean Paul van Waveren. Master's thesis, Delft University of Technology, 2001.

Migrating from Visual SourceSafe to Git

Migrating from one version control system to another is a big change. Chris Oldwood records the trials and triumphs of migrating from VSS to git.

Back in the late 1990s I found myself starting to acquire a considerable volume of code from various personal projects that I had been working on in my spare time. Having been brought up to use a Version Control System (VCS) to manage my source code assets at work it seemed eminently sensible to do the same thing at home. Unsure about exactly which product to choose I naturally fell into using the one that came bundled with the compiler – Visual SourceSafe (VSS). Given that it was also the VCS product I had been using at my client back then I got training and experience as a nice by-product of work.

Fast forward 15 years and I still find myself using the same VSS repo I set up all those years ago! Despite a considerable number of excellent free and open source alternatives springing up in the intervening years that were far superior, I found myself clinging on. This wasn't because of any killer feature, but mostly because I didn't want to lose easy access to my ever growing archive of version history when switching to another product. Mainstream support ended for SourceSafe last year so I don't even have that to prop me up any longer either.

This article then is my journey on how I finally shook off the shackles of SourceSafe whilst maintaining as much history as possible. It also goes into some of the questions I asked myself about source code structure of shared components in the face of online repositories like GitHub. Eventually I'll get to the mechanics of how to migrate source code and some of the problems I encountered along the way.

Why Git?

My lengthy procrastination has had the pleasant side-effect of letting the VCS ecosystem settle down quite a lot. I really didn't fancy going through all this again every time The Next Big Thing came along. The two clear choices for me were Subversion and Git, with the former being what I had been working with for the last few years, while the latter was just too big to ignore. I was also aware of the Git/Subversion integration so felt that there was a clear migration path in sight either way after finally putting VSS to rest.

If you're wondering whether I looked at any paid-for products, then, no. Remember that this was my own personal codebase and so I'm happy to live with whatever support I can get off the Internet these days. I might have shelled out for a migration tool if the freely available options didn't come up to scratch – they didn't with VSS2SVN, but fortunately they did with VSS2Git.

My workflow

Despite the Distributed Version Control System (DVCS) being a relatively modern invention compared to their Centralised (CVCS) counterparts, I've been (ab)using SourceSafe over the last decade in such a way that I've got a clunky similar model anyway.

Essentially I have a trusty old Network Attached Storage (NAS) box that contains the 'master' copy of my VSS repo and then on my desktop and laptop(s) I hold a copy. This gives me the full access to the history, which I want when developing, along with a repo to store my 'work in progress'. The downside of course is that I have to do a 'formal' check-in to the master some time later and re-sync the repo (think ROBOCOPY) to bring in changes made via other routes. Like I said, it's clunky.

Aside from the day-to-day development chores I also publish the source code along with the binaries on my web site to allow anyone foolish enough to tinker themselves. This has the nice added bonus of being an off-site backup. OK, so the .zip files don't have the full history but it was better than nothing until the arrival of cheap, cloud based backup services.

This neatly brings me to the final piece of my migration puzzle – the introduction of an online source code repository. Given that I'd already discovered Git was the best choice for migrating my history, another brief investigation also led me to GitHub as a suitable online provider. Once again the fact that all my source code is published (even the dross) means the public-only nature of the free repos happily suits my needs.

That said I still feel uneasy about relying solely on a service like GitHub as the canonical source for my code and so I still have bare Git repos on my NAS box to act as the central point for collaboration across my own devices. Bare repos only store revisions so you cannot use them for merging or development, which is essentially what GitHub provides anyway.

This leads to a workflow where the laptop(s) and desktop are the clones where I do my main development. I then pull and push to the repos on my NAS to synchronise (should the need arise) with a final push from the NAS to GitHub to publish (backup) my work in progress. When I want to formally release a new version I can build, package and publish to my web site as before. I'll also include the source code as a snapshot like always just to make it easier for someone else to pick it up.

As you've probably guessed there isn't exactly a lot of 'collaboration' going on, which is a major selling point for tools like Git and GitHub, but that's not the point really; for me it's as much about creating and dealing with a legacy codebase as a learning tool.

The monolithic repo

The biggest decision I had to make when deciding to move from VSS to another product was whether to keep using a single monolithic repo. My codebase contains around 50,000 SLOC of C, C++, SQL and PowerShell code distributed across 50 or so separate 'components'; these consist of scripts, libraries and applications. This way of working (one repo) has always suited me, but with a desire to publish via a more collaborative service too, I felt it might be a good time to see if it would be better to split it up into smaller components; perhaps with a separate repo for each one. Another eventual driver was the difference in the way VSS, Git and

Chris Oldwood is a freelance developer who started out as a bedroom coder in the 80s, writing assembler on 8-bit micros. These days it's C++ and C# on Windows in big plush corporate offices. He is the commentator for the Godmanchester Gala Day Duck Race and can be contacted via gort@cix.co.uk or @chrisoldwood

The worst case seemed to be migrating the whole thing once and then trimming it down manually to just the tree of interest

Subversion handle labels – in short, they don't; at least not in the same way that VSS does (on a per-file basis).

For projects with no dependencies the decision was easy as everything would be self-contained. For my C++ codebase however, where I have a number of different static libraries (Lib-1, Lib-2, etc. below) that are reused across many different applications (App-1, App-2, etc. below) the choice was harder. If I created one repo per application (option 1) I'd have to share copies of the libraries and then find a way to synchronise changes across them all. Doing this without just manually copy-and-pasting the changes is probably doable in Git but would likely require the kind of Git fu that is well beyond my current n00b skillz.

```
├──App-1 (repo)
│   ├──Lib-1
│   │   └──Lib-1 Tests
│   ├──Lib-2
│   │   └──Lib-2 Tests
│   └──App-1 Tests
└──App-2 (repo)
    ├──Lib-1
    │   └──Lib-1 Tests
    ├──Lib-2
    │   └──Lib-2 Tests
    └──App-2 Tests
```

The opposite choice (option 2) was to have one repo per shared component (i.e. static library) and then another separate repo for each of the application-level components (i.e. executable, COM server, etc.). My existing folder structure naturally lent itself to this approach because each component was already self-contained:

```
├──App-1 (repo)
│   └──App-1 Tests
├──App-2 (repo)
│   └──App-2 Tests
└──Lib
    ├──Lib-1 (repo)
    │   └──Lib-1 Tests
    └──Lib-2 (repo)
        └──Lib-2 Tests
```

Then there was the half-way house (option 3) which would see separate repos for each application-level component, but just a single repo for all the shared static libraries.

```
├──App-1 (repo)
│   └──App-1 Tests
├──App-2 (repo)
│   └──App-2 Tests
└──Lib (repo)
    ├──Lib-1
    │   └──Lib-1 Tests
    └──Lib-2
        └──Lib-2 Tests
```

The one exception to all this was my build scripts because they lived in the root of the tree. They aren't essential for building any single application as they're more for command-line batch-building of the entire estate. I decided they could live with the libraries or go in another separate repo.

So many choices! In the end I opted to split out the applications and all the shared libraries into separate components and repos (i.e. option 2 above). My rationale was that a 3rd party should only need to clone the bare minimum to get the application to build – the waters should not be muddied by any of my really old nasty legacy library code that is of no importance to the application in question. I also elected to create a separate repo for my build scripts as they are optional too.

From a Git migration perspective I was somewhat concerned about how much work this was going to be. The worst case seemed to be migrating the whole thing once and then trimming it down manually to just the tree of interest, along with moving all the files up into the root of the repo. Some VCS repos are immutable by default and Git is no exception, but it did appear that some advanced Git magic [Github-a] might make it possible to physically remove the unwanted sub-trees afterwards to conceal the initial mess.

Once again lady luck shined on me and whilst I was investigating how other developers have solved the problem of shared components in project-focused repos I stumbled upon a recent addition to the Git toolset – git subtree split [Pardus12]. As we shall see later this command can rebase (in the file-system sense) a sub-folder right back up to the root of a repo.

The investigation also lead me to the concept of 'submodules' [Git] in Git, which are akin to 'svn:externals' in Subversion, and it raised an important difference in the way that I, as owner and fundamentally the main developer of this entire codebase, work differently to how another collaborator would. Essentially I am interested in developing the libraries with one eye on the changes that affect all dependents, whereas a collaborator is likely to be only concerned in the effects on the application that directly interests them. Naturally I decided to optimise for the common case (me as the sole developer) and keep the libraries parallel to the applications rather than move them 'under' and then treat them as sub-modules. I can of course change my mind later as the hard work has already been done.

Project migration

Finally, after all that philosophical waffle we come to the mechanics of creating one or more Git repositories from a SourceSafe one.

Git Configuration

The first step is to read up on how Git handles line-endings as this will likely affect your code. GitHub have a splendid document that covers the practical aspects of this issue [Github-b]. If you think it's all rather convoluted you can read up on how the different settings evolved over time to cater for the different scenarios as they cropped up [Adaptive12]. Suffice to say that after opening a shell with the Git toolkit on your PATH you'll want to run this:

```
C:\> git config --global core.autocrlf true
```

The only real check-in data VSS2Git can go on are the commit timestamps and the check-in comments, if present.

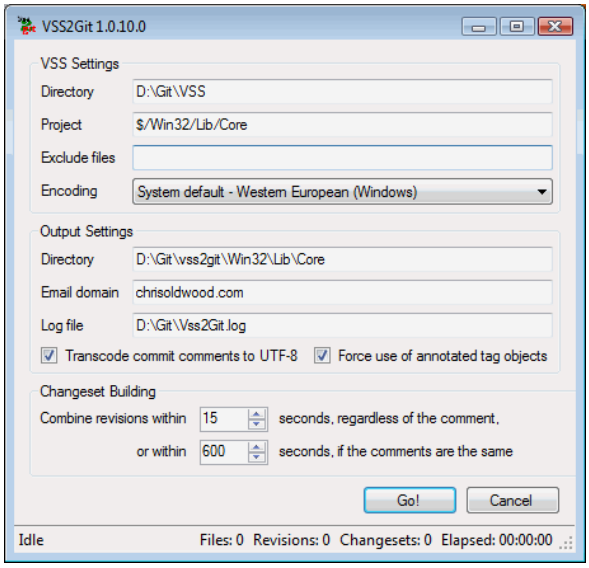


Figure 1

Next up create a folder under which you're going to create your migrated Git repository(s):

```
C:\> mkdir D:\Git\vss2git
```

VSS2Git

The tool I chose to use for the migration was VSS2Git [Google]. This is a GUI-based application that relies on another Git installation being installed, and being accessible via the PATH. The GitHub for Windows package [Github-c] contains both a UI and an easy way to launch a shell (CMD.EXE or PowerShell) pre-configured so that you don't have to have Git permanently on your PATH. With the planets suitably aligned you can run VSS2Git like so:

```
C:\> "c:\Program Files\Vss2Git\Vss2Git.exe"
```

Before starting I sync'ed my local VSS repo to match the one on my NAS box to ensure I was working with the latest stuff; this also gave me a separate copy which I could mess around with and see what happens when I migrate sub-folders instead of the root, e.g. my build scripts (see Figure 1).

As an example when I migrated my Core C++ library, which lives under $/Win32/Lib/Core, I used the following (non-default) settings:

```
VSS Directory: D:\VSS\Git
VSS Project: $/Win32/Lib/Core

Git Directory: D:\Git\vss2git\Win32\Lib\Core
Email domain: chrisoldwood.com
```

These should be fairly obvious. What might be less obvious are the two settings at the bottom (Changeset Building) that control how to turn a bunch of non-atomic commits into the more modern atomic ones. The only real check-in data VSS2Git can go on are the commit timestamps and the check-in comments, if present.

Personally I'm pretty fastidious about providing check-in comments and so I dialled down the setting that works off empty comments to a mere 15 seconds as I know any commit without a check-in comment is rare and almost certainly unimportant. In contrast I upped the other setting that matches comments to 600 seconds as I always tried to reuse the same check-in comment where possible; the only reason for a delay would have been getting temporarily distracted.

With everything configured it's time to hit 'Go!' and wait for VSS2Git to weave its magic. Depending on the repo size this could take some time as it has to build and replay the commits into the Git repo. This means that you can't point it to a bare Git repo because as explained earlier they have no working copy, but it will create a non-bare one for you at the path you specify.

Normalising paths

If you're migrating your entire VSS repo to a single Git repo then you're almost there, but if like me you want to split it up you have more work to do.

The VSS2Git tool converts files and folders exactly as they are in the source repo. This means that the file $/Win32/Lib/Core/ReadMe.txt will appear in the Git repo under the same relative path (\Win32\Lib\Core\ReadMe.txt) even if you only intend to migrate the $/Win32/Lib/Core project. What I wanted was for all the paths to be shifted up so that \Win32\Lib\Core is now the root of the Git repo, e.g. ReadMe.txt moves from \Win32\Lib\Core\ReadMe.txt to just \ReadMe.txt, and for all the superfluous parent folders to disappear.

As mentioned earlier Git has a separate command for this – subtree split – although it appears to be a very recent addition at the time of writing. When I was originally investigating, some Git builds supported it (msysgit) and others didn't (GitHub for Windows). Both of these now do, but it's another thing to catch out the unwary.

The way I used this command was to create a new branch in the VSS2Git generated repo based on master (a.k.a trunk), but with the paths munged to strip off the leading $/Folder/Folder parents:

```
D:\Repo> git subtree split
--prefix=Win32/Lib/Core -b split
```

Creating a master repo

When describing my current workflow earlier I suggested my point of collaboration was still going to be a central repo on my NAS box. Whilst the point of a DVCS like Git is that I can use it in a peer-2-peer like manner, it also gave me an opportunity to create a bare repo and see what the differences were in practice. The folder structure on my NAS box reflected the way I was going to clone them for use during development.

For the Core library I created the bare repo like so:

When cloning a Git repo the source is normally referred to by the name 'origin'

```
C:\> git init //ziggy/Git/Win32/Lib/Core --bare
```

…and then pushed the cleaned-up branch from the VSS2Git generated repo to it:

```
C:\> git push //ziggy/Git/Win32/Lib/Core
split:master -tags
```

This has the added effect of moving the changes from the 'split' branch back onto the 'master' branch where they belong going forward.

Linking the master to GitHub

Once again this extra step is somewhat contrived because I'm using GitHub more as a backup than a point of collaboration. After creating a new repo on my GitHub account, called 'Core', I can then link it to the bare repo on my NAS box and push the migrated source like this:

```
C:\> pushd \\Ziggy\Git\Win32\Lib\Core
Z:\> git remote add github https://github.com/
chrisoldwood/Core.git
Z:\> git push -u github master --tags
```

When cloning a Git repo the source is normally referred to by the name 'origin'. I chose to name it 'github' instead to signify its status as just another clone rather than being the canonical source.

Line endings, again!

If you read the article(s) listed at the beginning about line endings you'll have noticed there is now a per-repo setting for handling the line endings issue in a way that allows developers to configure their machine how they like it whilst letting the repo decide what its own policy should be.

Of course there is a chicken-and-egg problem here because you need a repo to add the `.gitattributes` file to, but trying to do this earlier means you can run into other problems because you've tainted the repo you want to push your migrated revisions to. This is not a major problem when you're used to Git, but it's a head-scratcher when you're new to it. Consequently I left this step to be the first new commit following the migration.

At a minimum you want to add a `.gitattributes` file in the root of the repo that has this at the top:

```
# Set default behaviour, in case users don't have
  core.autocrlf set.
* text=auto
```

If you read up on what goes into this file you'll probably come across the `.gitignore` file too and so I ended up committing them as a pair. If you're using the GitHub for Windows UI you'll find this is a key part of the Git repo configuration. Here is a tiny example of the kinds of files you normally avoid checking into any VCS repository and therefore get included in your `.gitignore` file:

```
# Build artefacts
*.dll
*.exe
*.lib
*.obj
```

Stripping VSS bindings

If your projects are based around Visual Studio then you'll probably have enabled the VSS integration. The GUI friendly way to untie your solutions from VSS is to use the 'File | Source Control | Change Source Control…' option and unbind the projects. You'll still be left with some detritus in your repo and in a couple of cases I had to prise the bindings manually from Visual Studio to stop it complaining every time I opened the solution. There is a blog post from Brian Carroll [Carroll04] that describes how to manually strip the bindings, but it basically boils down to the following steps:

- Delete any `*.vspscc` and `*.vssscc` files
- Remove the `GlobalSection(SourceCodeControl) = preSolution` fragment from the `.sln` file
- Remove any `Scc* = SAK` lines from the `.vcproj` file preamble

Automation

If, like me, you're going to split out many libraries and projects this all becomes very tedious if done manually and so a little dash of automation goes a long way. By sticking to one simple convention – the local Git repo folder names and GitHub repo name match – you can apply the Swiss-Army knife of DOS programming (the `FOR` loop) and run the same commands across every repo.

In my case I had a whole bunch of repos generated by VSS2Git at the same level in the file-system, except for the Lib folder, which itself contained another bunch of repos:

```
├──App-1
├──App-2
├──Lib
└──App-N
```

The following one-liner (ahem) will navigate into a repo folder, do the path normalisation, create the bare master repo, push the migrated changes, attach it to GitHub and push the changes to that too. And do it for every folder that appears to be a Git repo (i.e. has a hidden .git subfolder at its root) – see Listing 1.

The most laborious part of the process now just becomes driving VSS2Git. There is a request open to suggest automating it via the command line [VSS2Git-a] but I don't know if there has been any progress on it at the time of writing.

Impedance mismatches

Whilst Git is superior to SourceSafe in most of the ways that truly matter there are a couple of features of SourceSafe I had used that do not translate well to Git. The problems I've run into largely stem from the use of a single repo to store multiple projects.

In VSS you can attach a label to any file at any version, whereas in Git labels are called tags and they are attached to commits

Labels

In VSS you can attach a label to any file at any version, whereas in Git labels are called tags and they are attached to commits. Because my VSS repo contains multiple independent applications I have multiple 'v1.0' labels on different projects that are unrelated to one another. In contrast, Git tags are global and therefore you don't appear to be able to mirror the concept. This became another driver for me to split my monolithic repo up as I often use the labels to see what's changed between releases.

If I had chosen to maintain a monolithic repo under Git I could always have adopted a new convention whereby I prefix the version number with the application name, e.g. 'ProductX v1.0' and 'ProductY v1.0'. Of course the shared libraries ended up adopting this scheme anyway, even under VSS, as you can't label two versions of the same file with the same physical label!

The second problem with the way Git handles tags is more tooling related, but is a result of the indirect way that tags are associated with commits – you can't easily see what tags are attached to what versions of a file. Hence if you have a bug in one version of a file it's not easy to see what other releases might also be affected. Hopefully the tooling will eventually catch up in this area.

Shared files

Git is efficient in the way that it stores content because it only ever stores one physical copy for any file with the same content. However, that's an implementation detail and two files cannot be linked in different places in the repo so that a change to one automatically affects the other. SourceSafe on the other hand does support this feature; in fact this is exactly how moving files in SourceSafe is implemented – share and then delete.

```
for /d %d in (*) do @(
  if exist "%d\.git" (
    pushd %d
     git subtree split --prefix=Win32/%d -b split
     git init //ziggy/Git/Win32/%d --bare
     git push //ziggy/Git/Win32/%d split:master
      --tags
     pushd \\ziggy\Git\Win32\%d
      git remote add github
       https://github.com/chrisoldwood/%d.git
      git push -u github master --tags
     popd
    popd
  )
)
```

Listing 1

Whilst in some scenarios this feature can prove dangerous it also has its uses, such as sharing scripts and common non-code assets across applications.

Dropped history

When migrating projects from SourceSafe that had later been moved to another folder (e.g. a 'deprecated' folder), VSS2Git only imported any file revisions that existed at the point the project had been moved, or after. I'm not sure if this is a limitation of VSS2Git or a side-effect of the way files and folders are moved in SourceSafe. I have raised an issue on VSS2Git [VSS2Git-b] to see if the problem could be investigated further.

Summary

This article was about making the transition from the propriety world of Microsoft's Visual SourceSafe version control product to the modern, open-source world of Git. Along the way I discussed some of the more fundamental questions about how I wanted to re-organise my repository of projects, both to maintain as much history as possible, such as the labels, but also to allow easier collaboration in the future using an online repository like GitHub.

With the rationale explained I then showed how I migrated and split up my monolithic VSS repo with the help of VSS2Git and some command-line driving of the Git toolset. Finally I described the major features in VSS that couldn't be directly mapped onto Git and the effects they had on the migration. ■

Acknowledgments

Once again the ever watchful eyes of Frances Buontempo and Roger Orr have saved my blushes and added that much needed extra polish to the article. Thanks.

References

[Adaptive12] http://adaptivepatchwork.com/2012/03/01/mind-the-end-of-your-line/

[Carroll04] http://weblogs.asp.net/bkcarroll/archive/2004/03/08/86059.aspx

[Git] http://git-scm.com/book/en/Git-Tools-Submodules

[Github-a] https://help.github.com/articles/remove-sensitive-data

[Github-b] https://help.github.com/articles/dealing-with-line-endings

[Github-c] http://windows.github.com/

[Google] https://code.google.com/p/vss2git/

[Pardus12] http://log.pardus.de/2012/08/modular-git-with-git-subtree.html

[VSS2Git-a] https://code.google.com/p/vss2git/issues/detail?id=2

[VSS2Git-b] https://code.google.com/p/vss2git/issues/detail?id=8