Interview with Andrew Oliver – Making video games

Andrew Oliver, known with his twin brother Philip as 'The Oliver Twins', has been making video games since the mid 1980s. In 1986 15% of all games sold in the UK were written by Philip and Andrew. I talked to Andrew about his use of maths in programming.

Photographer: Vicky Page

Andrew and Philip Oliver

What sort of maths did you use in your programming?
Maths is extremely fundamental to programming and whilst I've had a lifetime of making video games, I couldn't have done that without having a solid foundation of maths understanding behind me. Whilst in later years I needed to program an algorithm to solve quaternion transformations using only integers, all programming is about logical problem solving and maths teaches logic. Most console games are 3D and there is a lot of 3D maths required to create these games.

Did you enjoy maths at school and could you see the link to programming then?
I'm very science minded and did relatively well in maths. I wouldn't say I enjoyed maths for the sake of maths, but I did choose Applied Maths and Pure Maths at A-level, when I was at school around 1985. I really understood Applied Maths and got an A grade. But, I just couldn't get my head around Pure Maths, and only got a C grade. Solving quadratic equations and the like, just didn't make any sense to me. It was all so theoretical with no obvious use for it in any practical application.

However, Applied Maths was totally different; I remember you'd have problems to solve, like working out the amount of thrust needed to launch a rocket to the moon. You then had to calculate the mass of the rocket with the fuel and so needed to recalculate with the added weight of the fuel. You then work out the angle you'd need to shoot the rocket to hit the moon's orbit, but then need to recalculate the new angle due to the gravitational pull that would now be affecting the angle. This was all maths I could picture. I could even draw diagrams for it and then work out the related equations and see why this maths has a very real application.

© Richard Watkins | Alamy Stock Photo

A vintage BBC Micro

When I did the maths for programming games, I would always write out my workings with full variable names, such as 'angle', 'thrust', 'velocity' and 'mass', rather than α, β, γ, m, c, or v. I would argue with the maths teacher that maths was about algebra and computers used algebra. In computer programming you should use names that represent their properties. I said Greek symbols had no place in modern maths and computer programming. On this point we agreed to disagree! I like to see maths and code as readable and understandable as possible.

Did you meet computers at school?
My brother and I received a Christmas present of a computer – a Dragon 32 that we quickly exchanged and upgraded to a BBC Micro (see photo). We started programming on the then new BBC Micro by Acorn Computers, which was renamed Acorn RISC Machines and later shortened to ARM – that's the Arm chips that are in every mobile phone!

We had a love of the newly invented arcade machine games such as *Pac-Man* and *Space Invaders*, so we attempted to try to recreate these early video games on our new computer, learning coding along the way. Our maths teacher allowed us to use the BBC Micro at school break times because he didn't know what to do with this new machine that had turned up. We said we did, so we ran a computer club to teach our friends how to program. We had our BBC Micro at home and would work out how to make the computer do a certain thing, write the code neatly and then show it at our next computer club. Not only did some of our friends also go on to become programmers, we sold some of these 'listings' to magazines like *Acorn User* to make some good pocket money.

Is there any particular maths you remember using in your programming?
We were in our sixth form at Trowbridge school, in Wiltshire, when we 'discovered' computers. We had a sixth form common room with a pool table. So, there were times when we went back to the common room and would play pool with our friends. At the same time, we would be learning about vectors and matrices and we were learning programming on our BBC Micro.

We met a Bath University lecturer, who wrote and published a computer chess game. He explained that the technologies and techniques were improving fast and it was only a matter of time before a computer would beat a chess master, which happened a few years later, when IBM's Deep Thought computer made history by beating chess grandmaster Bent Larsen, becoming the first computer to win a regular tournament game against a grandmaster.

Whilst we could appreciate that computer algorithms are perfect for analysing potential chess moves and working out how to beat a human, we wondered about making a pool game. Our aim was a game where you could play pool on a computer and maybe even play against a computer. It was obviously all about working out the angles of bounce and transfer of momentum from impacts. We fully understood and embraced vectors and matrix maths, but we just couldn't work out how to get all of this working in the then very limited 8-bit integer computer.

What sort of workarounds did you use because of the limitations of early computers?

An 8-bit computer could only hold numbers between 0 and 255. You could make a negative number, by having another number indicating this number represented a negative number, or two 'bytes' together could represent a larger number. Very quickly you make shortcuts like the number of degrees in a circle is 256 not 360. But the maths needed to create a realistic 'break' for a pool game was just beyond us.

We did however create the best selling *Advanced Pinball Simulator*, on the Amstrad CPC, Commodore 64 and ZX Spectrum 8-bit computers. It had pretty good realistic ball physics and was a great fun recreation of the real machine. Whilst at Codemasters, a Cambridge student then created *Professional Snooker Simulator*, which nailed the maths perfectly. Although, you couldn't play against the computer until a friend wrote a game called *Jimmy White's 'Whirlwind' Snooker,* which allowed you to play a decent game against the computer. Exceptionally clever, if you think about the amount of calculations needed to go through all the possible angles and speeds to determine a good shot that not only gets the ball into the pocket but also, on a miss, leaves the table relatively safe from the human player.

Did you ever have to learn some maths because you needed it for programming a game?

For several years my brother and I created many best selling games. But these were all 2D 8-bit games, which just moved chunks of memory around that looked like characters moving around the screen. Even our 16-bit games, on the Atari ST, Amiga, Mega Drive and Super Nintendo, were still 2D.

The Sony PlayStation games console changed everything. It was mandated by Sony that all games *must* be 3D, and suddenly that created a 'barrier to entry' for everyone. It was daunting but also exciting. Suddenly, we had to work out how to create 3D characters in 3D worlds. I literally went back to my old school maths books to brush up on matrix transformations and worked out code to perform these calculations fast and smoothly, using the new 32-bit PlayStation processor.

As it was using an integer processor, I still used the tricks like having 256 degrees in a circle, only I then broke each degree down into a further 256 parts of a degree. Real maths divides a degree into 60 minutes, as a fraction of a degree. So what I lost in accuracy in the degrees, I gained in the fractions! This meant that the code was extremely fast, so I could perform more calculations in the same time. I remember that a typical PlayStation game could have around 5000 polygons in a single scene to create a game that ran smoothly at 30 frames per second.

Did you ever implement a solution in a game only to discover the related maths later?

This happened many times. For example, in 3D games you want only to display polygons that are facing towards the viewer. Objects are created as shells, and the screen only needs to display those you can see. Polygons that had their ordered vertices in the clockwise direction could be seen but if they are anti-clockwise then you are looking at the wrong side and they can be discarded. I thought the maths for this would be quite complicated and then I saw a diagram in my book that described it perfectly, and it's actually rather simple and elegant. Suddenly, another thing from my school day maths proved extremely useful.

I wrote the PlayStation's *WarGames* loosely based on the MGM film starring Matthew Broderick of the same name, with the gameplay developed from a 2D battlefield game called *Command and Conquer*. I created a chequerboard landscape, with all the vertices at different heights so that it looked like a 3D undulating landscape and covered it with trees and buildings. I then created little tanks and military vehicles, each of around 50 polygons, that would navigate the environment with their own unique abilities.

… went back to my old school maths books to brush up on matrix transformations …

It was all programmed with integer maths and my own unique 256-degree matrix code, so movements and rotations often end up as fractions, in my case, from 0 to 255 degrees. But every now and then I would see a tank distort, flip or just plainly look bugged. I was 100% convinced my code must have a bug in it somewhere. For more than a week solid I tried to track down why multiplying fractions in matrices occasionally went wrong. Issues with rounding in matrix calculations had never occurred to me.

A recently hired maths graduate explained to me that, many decades ago, people started using quaternions rather than matrices to avoid some of these issues. He explained these to me and so I re-coded it all into the 16 and 32-bit integer code and magically my vehicles realistically trundled over the 3D landscape. I had just discovered not all maths is perfect, which surprised me. But the length of code that quaternions creates equally disappointed me. So, I then had to decide what things could use the simpler matrix maths and which needed quaternions.

I am however extremely proud that our PlayStation games were rock solid 3D, when many PlayStation games, like *Tomb Raider*, had very wobbly 3D graphics. I knew my maths was better than theirs! In later years, I came across geometric algebra which calculated 3D far quicker and more elegantly than quaternions. But by this time all graphics processing chips had built-in quaternions as the way to do 3D maths. It felt like the VHS versus Betamax argument, everyone knowing that Betamax was digital and a better format, but VHS became the common, slightly inferior standard.

Can you tell us a bit more about the maths of *WarGames*?

Another trick that I created for *WarGames* that I'm very proud of is the extremely fast and accurate AI route finding system. In the game you control a set of a dozen vehicles. As you advance through the levels the initial 'load out' is different, so you have varying starting numbers of tanks, jeeps, boats, helicopters and then some infantry. For your own side you switch to each vehicle and give it a command, such as a direction to travel or a final destination point. Whilst all your units advance the computer has the same setup and is on the attack.

Remember, I said the 3D landscape was an undulating chequer board. It was exactly like a large chessboard and each unit was calculating several 'squares' ahead. In exactly the same way that chess works, each vehicle has its own unique movement attributes. If it was on 'look ahead 1' then it would just move to the 'square' adjacent, which wouldn't look great, although it would slowly drive to it. But, just like in chess, you could calculate several turns ahead and before long you would be aiming each unit at a destination several 'squares away' and it would be making its way there on what looked like an arbitrary angle. However, extra layers were added on, for example rivers were impassable by tanks and jeeps, which favoured bridges; boats had to get as close as they could but had to stay in water and helicopters could just go anywhere.

Then the algorithm would keep recalculating, so units heading for a destination would occasionally turn around as they find that other destinations were now scoring higher. Lastly, these calculations can be very slow, just as a chess computer could take a few seconds to calculate a good move because the complexities and exponential nature of look-ahead code create vast numbers of calculations. So, I further optimised things, so that I was using all 'spare processing' to calculate AI route finding.

As a video game, it's important to run at a smooth 30 frames a second. But this often means in games that there are times where the computer is just waiting for the next frame, because not much is happening. Rather than just wait, I created the AI route finding system to calculate better routes, until the time was up. So, suddenly this very maths-intensive system effectively had no overhead.

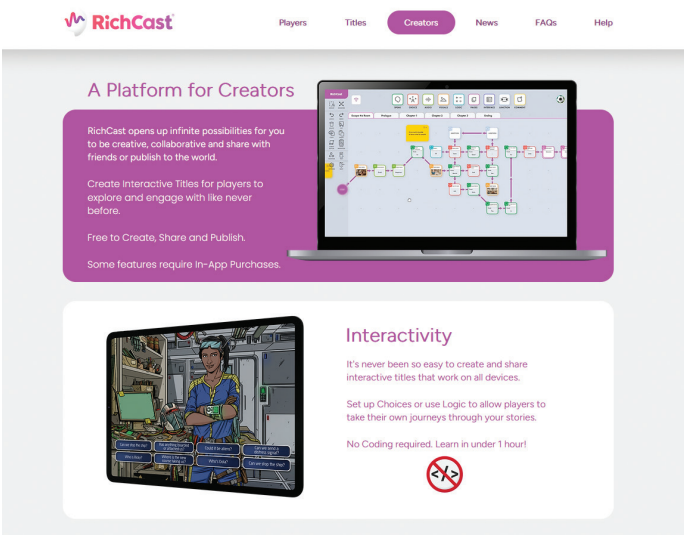
… using all 'spare processing' to calculate AI route finding …

But it did have a funny side effect. If it was a fairly quiet battlefield, you could see the enemy tanks behaving very cleverly, getting into good positions, navigating bridges well and so on. However, once you get a busy battlefield and lots of missiles flying and explosions occurring the computer has no processing time left for AI route finding and you can see the computer's units stopping and struggling to get into good positions. This actually makes them almost more human. When it's quiet they can think, but create lots of explosions and they lose the ability to think straight.

Are there times when the real maths just doesn't produce a good result in a game?

The games industry has grown up by creating tricks to make games that work well, are responsive and look great. However, there is a *real* industry that makes true to life simulations. For example, commercial flight simulators or vehicle driving systems.

The people who write these systems tend to be academics who use true maths to try and get a completely true simulation. But, if you write everything to true maths you will find things run very slowly, even on very fast computers.



Build your own game with Richcast.com

So, we have all got used to see amazing games like *Gran Turismo* on PlayStation or *Forza Horizon* on Xbox easily outperforming commercial driving simulators that were running on far faster computers than these home consoles. These programmers are so wedded to the idea that everything is in its truest form that their coding uses no clever tricks that create the same answers in a fraction of the processing time.

What is your advice to a budding programmer in relation to maths skills?

Modern society runs on computers, but computers are only as good as the instructions given to them by the programmers that write their code. There's so much talk of AI Systems and things like cars that will drive themselves and robot companions. But whilst companies would love you to think this technology is fairly close, it's proving to be a much harder challenge.

The world will always need, at least for the next few decades, programmers to make these systems. But whilst programmers start to use higher level languages and libraries, not needing to code things like quaternions anymore, learning maths still gives you the basics of logical thinking and helps you understand the underlying fundamentals.

Fighting realistic looking enemies in *Call of Duty* is just lots of rules for each specific case. Similarly, self-driving cars aren't magic, they are computers following a lot of set rules. Although, the problem with driving is the huge number of unpredictable 'edge cases' which need great care because a crash could be a real crash and not just a game freezing that needs a reboot!

So, four decades on from making all those ZX Spectrum games, are you still making games?

We have created literally hundreds of games over the decades, mostly through a company called Blitz Games; such as games of TV franchises like *SpongeBob SquarePants* and games of movies for all studios, including several for DreamWorks and Disney.

I hear you used your lockdown time to write a game on the Nintendo Switch, how did that come about?

We had recently invested in a company called FUZE that wanted to write a new BASIC language for running on a Nintendo Switch. So, just like on the BBC Micro when students and hobbyists could easily write games and share them, this company had the mission to make this possible for the Nintendo Switch.

We were impressed with their system and the fact that they had been granted the publishing rights on the Nintendo Switch. We said what it really needed rather than just lots of small demos was a big fun game that people would enjoy and could then dive in and look at the code. We said we would do it if we had the time, but we were just too busy with other things.

Then came lockdown and all our speaking engagements at colleges, universities and conferences got cancelled, so we decided to recreate a 'classic maze game' from years ago. Our very own '*Fast Food Dizzy*', from the late 1980s, that was a homage to *Pac-Man*. So, anyone wanting to learn how to program and share their creations can do so with FUZE for Nintendo Switch.

… create tools to allow anyone to make these new interactive stories and games …

Have you been involved with any other recent games?

We have decided to turn our hands to a new technology and a new opportunity. Mobile phones are amazing, they're in everybody's possession and can do so much more than apps currently allow. So, you can talk to your phone to set an alarm, play music or ask it a search question. But we've decided to create a system that allows you to talk to characters in games so that they understand and respond. We decided to create tools to allow anyone to make these new interactive stories and games. So anyone can decide which new characters you can talk to as part of their stories. It requires no programming, but lots of creative imagination and a good logical mind. Think of a Sherlock Holmes story where you solve the mystery by interrogating the suspects, or an escape room where you have to solve the puzzles by instructing the AI people in the room. Not only can anyone create these kinds of entertainment using our new RichCast system, they can also publish them straight onto their mobile phones and share them with friends, or even the whole world! The possibilities are endless for this new form of entertainment.

Richard Lissaman
University of Warwick