

ALL FORMATS
116 PAGES MONTHLY

Wireframe

LIFTING THE LID ON VIDEO GAMES

NEW DOOM

The rise of the nineties-style retro FPS

FLASH GAMES

The fight to preserve browser game history

DO IT YOURSELF

Code a block-pushing puzzle game

Issue 42 £6
wfmag.cc



BIOHAZARD

SUSPENSE, CO-OP SHOOTING, AND BODY HORROR IN GTFO

Tackling industry racism • Ghost of Tsushima • Wholesome video games



Squeezing WarGames onto the Sony PlayStation

How the Oliver Twins managed to fit WarGames: Defcon 1 onto Sony's debut console



AUTHORS

PHILIP AND ANDREW OLIVER

The Oliver Twins have been making games since the early eighties, and can now be found at their new consultancy firm, Game Dragons. gamedragons.com

As Sony entered the console market with the PlayStation in 1995, Hollywood film studios were beginning to take video games more seriously. Rather than license their properties to game companies, as they had been doing up to this point, many of them started their own interactive divisions to produce and publish titles themselves.

Robb Alvey was a former producer at Virgin Interactive and had just moved to the film studio MGM, and saw the potential of using the WarGames film license for a real-time strategy game. He was good friends

with our agent Jacqui Lyons and was sent the pitch for a game we'd been working on at our company, Interactive Studios. Called *Conquest*, it was a space-based RTS in the vein of the hit *Command & Conquer*. Alvey asked if we could make a proposal for a ground-based strategy game based on WarGames. We loved the movie – it was definitely a film for computer geeks of the eighties – and so we jumped at the chance.

Released in 1983 and directed by John Badham, *WarGames*

follows a young hacker who unwittingly accesses War Operation Plan Response (WOPR), a United States military supercomputer. We took the film's back story as a basis, but moved it slightly into the future. In our game, WOPR has taken control of factories and created its own arsenal, resulting in a war scenario akin to the original *Terminator* movie. MGM liked the premise, and saw there was even a possibility of a movie sequel based on the idea. They talked to Badham, who liked it and said that he'd rework the script and wanted to be credited on the game. Badham is credited on the game, but we never received that reworked script.

The first version of the game we pitched was for PC, with tanks, jeeps, mechs, and troops all based on a full grid-based rolling terrain that players could zoom, scroll, and rotate around with mouse control. Once this got underway, MGM asked if we could produce a PlayStation version. Clearly this lacked a mouse and would be played on CRT TVs with much lower resolution and rendering capabilities than a PC, and so adapting the concept to a relatively new console would prove something of a challenge.

It took a lot of design work and experimentation to get the controls of multiple units feeling natural with a PlayStation controller. While we played with the idea of group selection around a cursor, it felt wrong to try and emulate

✓ *WarGames: Defcon 1* emerged for the PlayStation and Windows in 1998.





◀ All of *WarGames'* graphics had to fit in the PlayStation's 1MB of VRAM.

the PC's functionality on a console. Instead, we arrived at the principle that you controlled each unit directly until you switched to another unit. AI would then take over and follow the last simple command you gave:

“Sony had mandated that all games on the PlayStation were 3D”

Attack, Retreat, Defend, or Follow. So typically, at the start of a game, you'd set a few units on attack, a couple on defend, and then jump into the driving seat of the lead attack tank. This gave us the right balance of console playability and strategic gameplay.

In the next section, Andrew Oliver explains how he got to grips with making a 3D game for the PlayStation, and cracking *WarGames'* tricky AI.

GETTING TECHNICAL

The PlayStation was programmed in C. SN Systems, a small development business comprising just two people, created the PlayStation's programming environment using the GNU open-source C compiler for Windows 3.1.

When first programming the PlayStation, I was very suspicious of the code that C was producing under the hood. I'd lived a life of having to optimise everything, and suddenly I was writing in a high-level language – and while C is powerful and easy to read, I wasn't sure it was producing the most efficient code. I wrote small tests and disassembled the code to see what it had produced, and created a document for the other programmers of dos and don'ts for writing C code.

Sony had provided lots of demos and libraries. These were useful, but when it came to the heart of the graphics engine and formats, I looked at

theirs and then spent time optimising. I replaced all of Sony's libraries except sound and CD

handling. I wrote the main PlayStation game engine for *WarGames*, while Ian Bird wrote the gameplay and missions. John Whigham

and Richard Hackett wrote the PC version and helped massively with my understanding of 3D.

DEVELOPING A 3D ENGINE

I'd been writing code for over ten years by the mid-nineties, but all that skill was built up in assembly language, and largely for 2D games. Sony had mandated that all games on the PlayStation were 3D, since this was the ➤

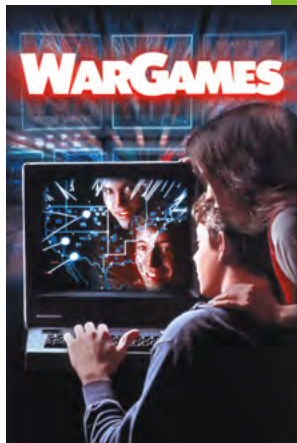
▼ *WarGames: Defcon 1* contained a total of 16 story-based missions.





▲ Lead artist Steve Thompson working away on *WarGames'* low-poly units.

▼ The 1983 film that led to *WarGames: Defcon 1*.



► Figure 1: A diagram showing how to calculate front-facing polygons.



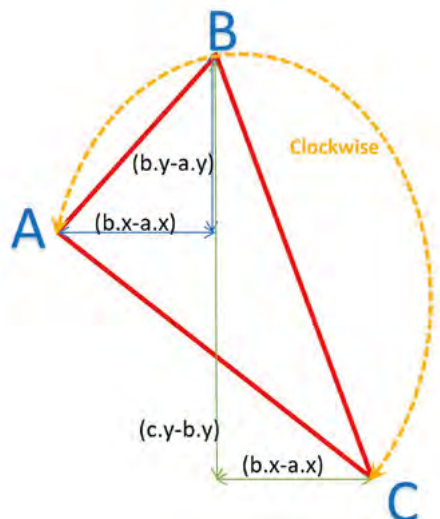
console's unique selling point. Programming the 3D was trickier than I thought, as I relied on my A-level maths knowledge of matrices and vectors to program it.

The PlayStation was based on 32-bit integer maths. I was used to dealing with integers, but I'd never had to do 3D transformations of three-dimensional coordinates, so this presented a challenge. I effectively used the top 16-bit as my integer and the lower 16-bit as the fraction, and this worked well. In fact, I was proud of how smoothly the system projected into 3D space, especially when compared to a game like *Tomb Raider*, which had very wobbly 3D graphics.

3D MODELS

Inevitably, lots of 3D models had to be created for our game. John Whigham wrote a simple 3D modeller we called JOBE (John's Object Editor). Having a bespoke editor made our artists produce more efficient models – which was important, as we'd estimated that we had a budget of around 3000 polygons to create each frame. This might sound like a lot, but once you start creating 3D environments, you get through the budget very quickly.

Using my old maths books, I looked up the rotation, translation, and scaling of three-dimensional points and got the results I expected. I was soon able to spin a basic 3D tank around on the spot – but once the game was running, the tank would occasionally collapse in on itself. I was convinced it was a bug in my code, but it wasn't: I'd come across a



► Figure 2: Sony's tool showed how many times each pixel was overdrawn. The lighter the pixel, the greater the overdraw.

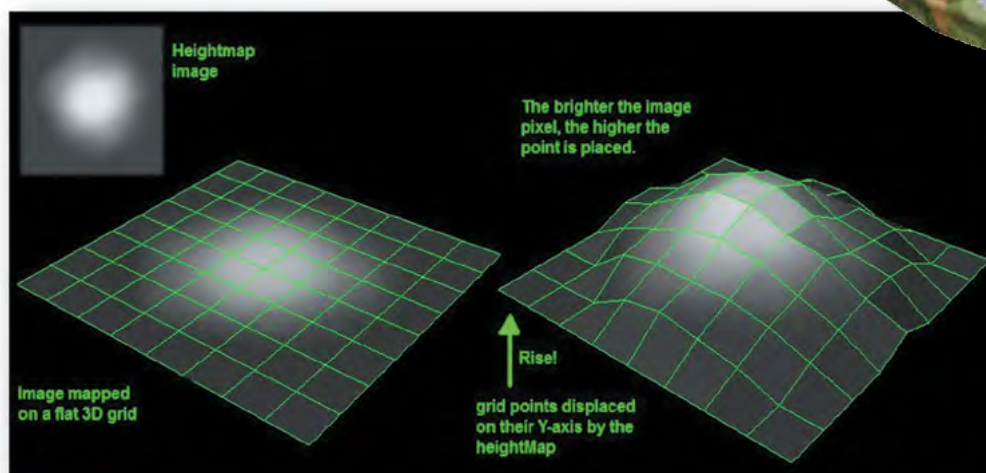
maths anomaly called gimbal lock. Eventually, I learned that I needed to use an entirely new algorithm system called quaternions. That single bug probably wasted around a week of my life!

When drawing a 3D object, I assigned a radius to each. The code would quickly check that the centre of the object plus its radius was within the boundaries of the screen. Any object that had no polygons on screen was discarded immediately.

Another useful piece of optimisation was back-face culling. Most game objects are hollow models and you only need to see the polygons that are facing you; I knew I'd only need to display half the polygons, but I wasn't aware of how it was done. I was also worried about the extra overhead the checks needed. I asked around and discovered it's actually unbelievably simple. Every polygon is a triangle, with nodes A, B, C, each with X, Y, Z coordinates. The calculation to see which way the polygons face is done once they've been translated into the 2D screen space. If the 2D coordinates of a triangle projected on the screen are clockwise, then they're visible; if they're anti-clockwise, they can be discarded. The code then works through translating polygons to a 2D projection while adding them to the origin of the object and checking if it's on the screen. If it is, then it checks to see if the polygon is front-facing before adding it to the draw list (see Figure 1).

RENDERING MODE

Modern games tend to use deferred rendering, but the PlayStation used a more rudimentary forward rendering system. This meant preparing a Z-sorted depth list, then drawing the furthest polygon from the camera first, and then drawing



◀ **Figure 3:** *WarGames: Defcon 1* used a simple height map system to create varied terrain.

the polygons nearer the camera on top of it, in order. The GPU would draw from the Z-sorted depth list, building the screen from the back to the front. To optimise the sorting, I pre-sorted each object within itself, as these polygons were, by their nature, all next to each other within a small Z-distance.

It's a simple process, but it can cause massive overdraw. For example, I'd print a large blue polygon across the whole screen to represent the sky, followed by the far landscape which would cover much of the sky, then coming forward I'd print more landscapes, buildings, and objects, and finally the screen overlays (HUD). What occurs is 'overdraw', where many pixels of the 640×480 pixel screen have been overwritten multiple times, which causes the game's frame rate to drop. This is often due to the GPU failing to draw all the polygons before the next game cycle is ready.

Sony produced a useful analysing tool which could snapshot the screen at the end of the frame and show how many pixels were overdrawn. The lighter the colour of the pixel, the more it's been written to (see **Figure 2**). We'd see what things had been completely obscured and come up with techniques to avoid this. For example, my landscape routine would store the lowest screen coordinate of its further edge, so that on the next draw, the sky would only print from the top to a few pixels below this point. This saved the GPU an expensive full screen redraw.

"We estimated that we had a budget of 3000 polygons to create each frame"

HEIGHT MAP LANDSCAPE

The landscape was created using a simple height map system (see **Figure 3**). The landscape was divided into tiles, a little like a chess-board, which had an 8-bit indexed 'tile' type. I gave every point a height, keeping this to 8-bit as this variation was ample. If the landscape looked too flat, I'd just multiply all points by a constant. Technically, these points were the corners of the tiles (their nodes); therefore there would be 65×65 nodes.

This size was awkward to store and index, and it's this kind of thing that often causes bugs, so I just kept to 64×64 while the far edges (the 65th position) used the 64th's height. This was generally the sea, which was flat anyway.

AI AND ROUTE FINDING

An important part of *WarGames* was commanding units: jeeps, armoured personnel carriers, tanks, boats, and helicopters. Infantry units could also assist, but these were controlled by the computer rather than the player. Each unit type had its own strengths, weaknesses, and navigated the landscape differently. The game had to handle direct control and AI control of each.

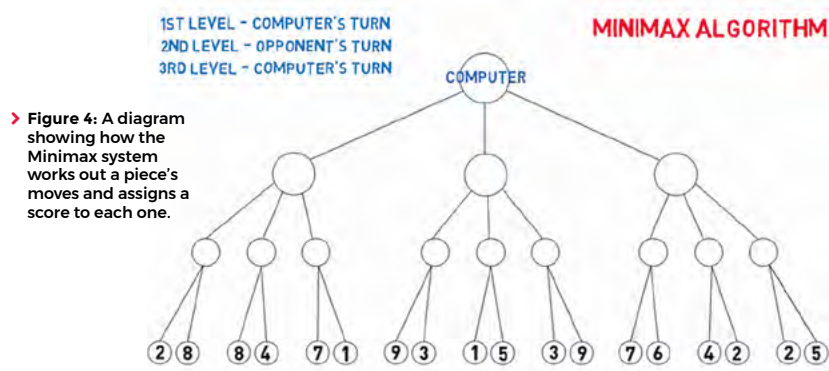
When playing *WarGames*, players would start with a set of up to eight different units. Players would switch between each of the units and when in control, could 'drive' around with the controller. Players also had the ability to set a unit's AI between four basic states: Attack, Retreat, Defend, or Follow. As players left ▶

▼ **Andrew programming *WarGames* back in 1997, when he still had most of his hair!**



SHADOWS

Shadows are important in games, since they bed objects into the world. However, with all the units possible on the screen, the complicated systems employed today were out of the question in *WarGames*. I therefore created a single polygon texture that could be rotated and printed in subtractive transparent mode underneath each unit, which darkened the area. It was simple, cheap on the GPU and CPU, and highly effective. Then we came across the issue of having WOPR units that are big mechanical walkers, so we created four frames of textures for these – effectively small GIFs.



► **Figure 4:** A diagram showing how the Minimax system works out a piece's moves and assigns a score to each one.



▼ Do not underestimate the power of PlayStation. Or the Oliver Twins.



direct control of a unit, it would follow this basic command. This was done in a system similar to Chess AI, which always fascinated me.

The system is called Minimax (see **Figure 4**). It takes a single piece, looks at the rules that govern it (a bishop can only move diagonally, for example), tries every place it's allowed to go, and gives it a score. If it's left in a position of being captured, it's a low score, and if it's a position of capturing an opponent's piece then it's a higher score, depending on the value of the piece it can take. The algorithm also needs to run through every available piece to see if other pieces would score higher. If it were to execute the move based on the highest scoring position, then it would appear to make a sensible move. This assumes it's looking ahead one place. Using this system plays a decent game of chess against an amateur.

The system can also be adapted to look multiple moves ahead and consider the opponent's potential moves. The code's written

in a recursive manner, where it not only moves every piece and assigns a score, but also moves every opponent's piece between each move and calculates those scores too. The size of the scoring table and calculations go up exponentially, but the algorithm is able to hold its own against even the smartest chess player.

I decided to use this system to control units in *WarGames*, since it's relatively simple but creates a smart adversary. Like chess pieces, each unit has its movement rules and values for scoring. Jeeps move faster than tanks, but can't go up steep inclines and have less armour, so can take less damage before they explode.

Each unit would locate a grid space and work out possible moves, decide which scored best, and then set its sights on moving to that position. It would move in the most sensible direction to get to its eventual destination. As it crossed boundaries between grid spaces, it would set its location to that grid space. This blocked the space for other units coming into the same space and used far fewer calculations than adding a separate system for collisions between units.

Unlike chess, however, units in *WarGames* used ranged weapons and needed to get into a good position where they could fire on the enemy. So while the first system was used for navigation, a second system checked whether the unit could see an enemy unit to shoot at, and this added to its AI score.

After much debugging, I could see from the calculations taking place that the AI was working, but discovered a problem: in chess, the recursive nature of the algorithm means



THE PREDATOR TANK

Inspired by the 1987 Arnold Schwarzenegger film of the same name, the Predator tank had a cloaking ability that rendered it almost – but not quite – invisible. The effect was created with a small piece of code in the texturing mapping routine. When it was printing the Predator tank as it tried to get the VRAM coordinates for the texture of the tank, I'd pass the screen coordinates it was going to print to. Ordinarily, that would just print exactly what was already on the screen. But I'd add a small offset of up to a few pixels based on the depth – applying this meant you could see the 3D shape of the tank, and it looked like it was slightly glassy as it just distorted the background.

► The stealthy Predator tank, inspired by a certain eighties Arnold Schwarzenegger flick.



^ The team behind *WarGames: Defcon 1* - Interactive Studios, later known as Blitz Games.

moves can take several seconds to calculate. *WarGames* was running real time at a constant 30fps, meaning that a complex recursive route-finding system would cause the game to stall when it did the calculations. I had a solution, or I thought I had a solution, whereby each frame the game would only calculate the AI for one unit. So if there were a maximum of 32 units in play, then every 32 frames, a unit would calculate its best destination. Even the fastest of units would take more than 32 frames to get to a previously desired destination. So this spread the calculations out to something more manageable for the processor.


In the heat of a battle, however, the AI was still slowing down considerably. I therefore took a novel approach: when games are locked to specific frame rates, you do all the processing for that frame, and then you wait for the next screen refresh. It was quite common on fairly empty screens, with only a couple of units, that the game could run every frame (60fps), but it would still wait for the second frame refresh (30fps). So I wrote a system where, instead of just waiting for the next screen refresh, it would start calculating the AI movement positions for the next units, until the screen refresh time arrived and then it would bail out and restart after the next frame had been processed.

Effectively, this processor-hungry routine now appeared to take no time off the processing. The game suddenly ran very smoothly - in fact, most of the time the game ran at 60fps, except when there were a lot of explosions, which were much slower for the GPU to process.

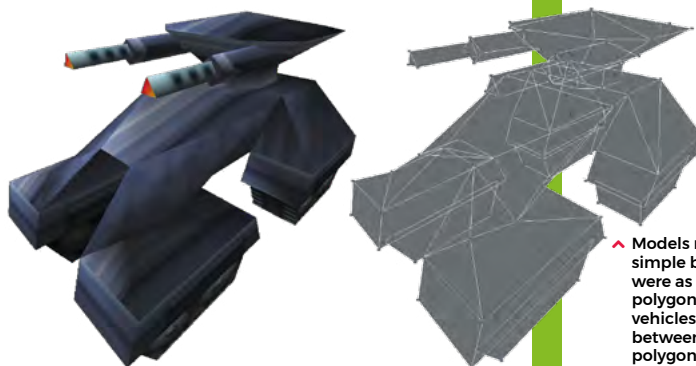
"In the heat of battle, the AI was still slowing down"

While I was pleased by how well the system worked, it had a downside. If there were a lot of units on the screen, there was little processing time left at the end of the main loop, meaning the AI didn't get much time to calculate the units' actions. As the system struggled to have enough processing time, it decreased the number of tiles it looked ahead to, down from the maximum of

five, to help mitigate this problem. However, that meant its destination was closer, and if the 'action' was prolonged, the units would reach their destinations before a new destination was calculated. It would get there and just stop. This only happened in prolonged battles with screens full of action, so most people wouldn't notice it. I'd like to think that the AI was behaving more human-like: that is, in a prolonged, high octave, battle scenario, its ability to think straight becomes more restricted!

You can now play in a browser right here:
wfmag.cc/wargames. 

✓ Music and sound effects were produced by composer Tommy Tallarico (*Earthworm Jim*, *MDK*).



^ Models ranged from simple buildings that were as low as ten polygons, to tanks and vehicles that ranged between 50 to 100 polygons each.