COMPUTER
WORLD

INTRODUCTION

For a while now, I've been interested in the simulation of complex systems. Often, this is a means to some end: the backend of a game, an artificial society, a model of an organism or a swarm of insects. However, I've lately become interested in the idea of simulation for simulation's sake, as a 'performance' by a computer, or other simulating medium. At the moment, I am developing a collection of such performances that take place between a computer program and computer memory.

I find this relationship interesting because it's a very particular kind of embodiment: a computer program both is computer memory (in as much as it consists of memory instructions), and is contained within it, and interacts with it as an environment. This idea of spatialised computation is not unique to computers: it occurs in many entities that can be said to 'physically compute', such as ants and slime moulds, where an entity's form comprises both its environment, and its means of navigating the world.

This zine is an exploration of some early ideas for these performances, and some thoughts about their inspiration, context, and where they can be taken. It's also been an opportunity to write down a lot of things i think are interesting in one place.

1. STACK AND HEAP

Computer memory can be a hard thing to visualise, because it's both a total abstraction, and an actual, real, physical thing. Memory is the space computers use to store instructions that run each program, and it's structured in such a way that each program is able to run simultaneously, without confusion.

When a computer program is compiled, it becomes something called an ELF: an executable and linkable file. ELFs are a series of assembly instructions, that are stored in computer memory. The program itself runs by directing a single moving agent - a 'pointer' - around these instructions: different entries in memory cause the pointer to do different things, and move to different places. The instruction pointer is, also, simply an entry in computer memory: one in a fixed place, that gives the

address of the current progress through the file.

The memory used by the program is divided into different sections, including two that change while the program is running: the stack, and the heap.
The stack and heap sit at opposite ends of a reserved area of computer memory, kept empty in order to run the program. As the program runs and the space filled by each grows, the stack expands up, into higher numbered memory addresses, while the heap expands down.

The stack grows every time a function is called: the current state of the program is pushed onto the stack (so it can be retrieved later), and the program jumps to the new function. When that function finishes, the original function can progress, and the saved state is removed from the stack. To make the stack get larger and larger, the program keeps calling nested functions: that way, before one can finish, another has started, and more states are pushed onto the stack.

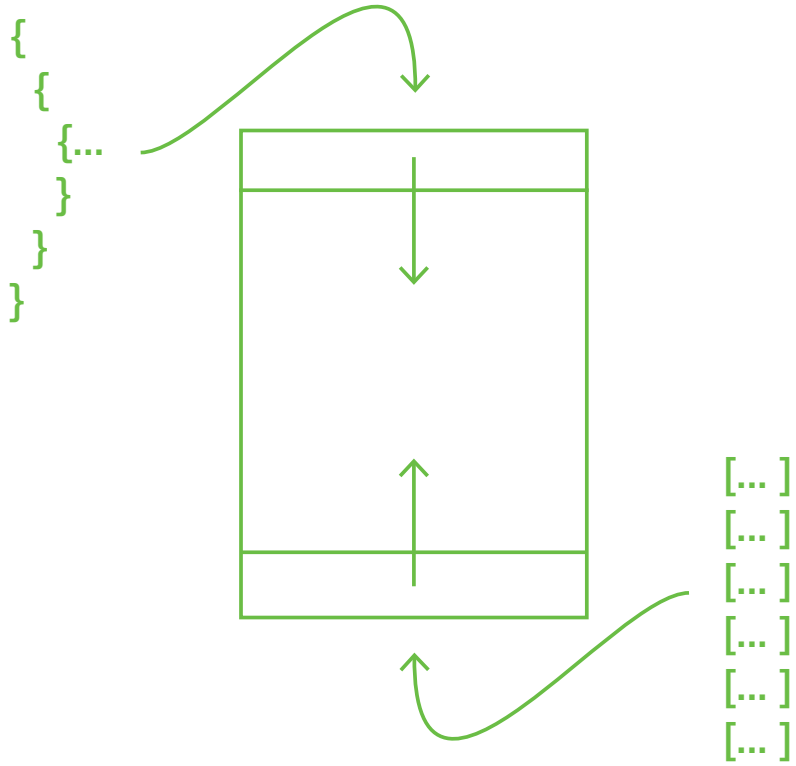
The heap can be used more directly, by allowing users to allocate parts of it to store variables. There is technically no limit on the size of variables that can be stored on the heap, and you can continue to add until the space runs out. To reduce the size of the heap, the user simply has to clear those memory entries again.

STACK AND HEAP PERFORMANCE

A program that grows the stack and heap until they meet in the middle.

The stack grows by repeatedly calling functions. as the stack meets the heap, the memory management unit will cause the program to crash

The heap grows by allocating memory. when the heap touches the stack, the program will crash.



2. ENVIRONMENT

Earlier this year, I got the chance to spend a few days using an industrial knitting machine called a Shima Seiki. The machine itself was exciting for obvious reasons, but what really left an impression on me was the software you use to program it. This software was super-convoluted, the kind of nice/horrific industrial software with thousands of features, but that at its core did not seem to have changed since the 1970s.

The main compelling part of this software was a kind of spatial programming language that both described real stitches in the thing you were about to knit (a knit pixel, a purl pixel, a skip pixel etc) and also, depending on its position in the ‘workspace’ effectively programmed the machine to perform particular operations on a row or set of stitches.

The primary information of the software was ‘stitches’ so it made sense to just have all the code be ‘stitches’ too — even if it made no sense at all to anyone using it. This seemed like an efficient way to do the code, probably a hangover from when memory usage was much more of a constraint than it is now.

One of the things that’s interesting about certain kinds of co-operative organism — like ants, termites, slime moulds — is that their environment becomes part of their way of thinking. When a brain consists of more than one individual, part of the ‘thinking’ is situated in their surrounds. Slime moulds leave chemical trails in places they’ve already searched, while termites direct one another by leaving complex pheromonal markers during decision making processes.

This concept exists for humans too; the pencil and paper as an extension of intelligence. There’s a classic thought experiment where someone is given a long division sum to perform, and a pencil and paper. When they work it out, they are asked: where does the maths happen? The pencil or the brain, or both?

We don’t tend to think of computer programs as being physically manifest: there is, obviously, a physical computer, but the relation of the program to that space is like a consciousness, or a spirit: totally abstracted from the hardware with which it interacts. Of course, we also know this is not true: at some point, the computer works by moving memory around, and moving memory around is really just moving electrons around. But -- electrons are very small, and we don’t like to think of them taking up space.

ENVIRONMENT PERFORMANCE

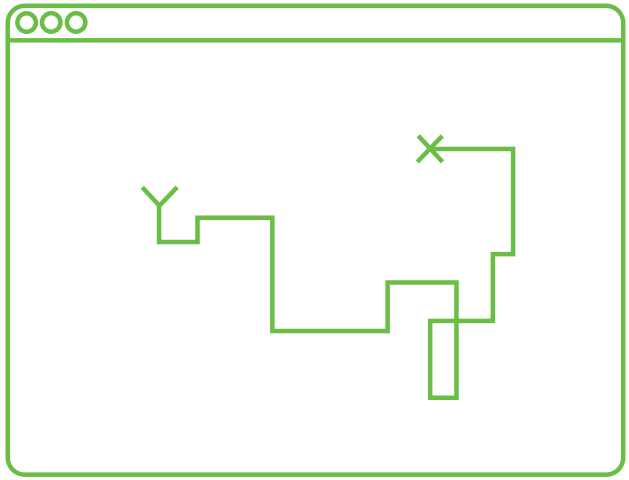
A simulation that moves itself though memory

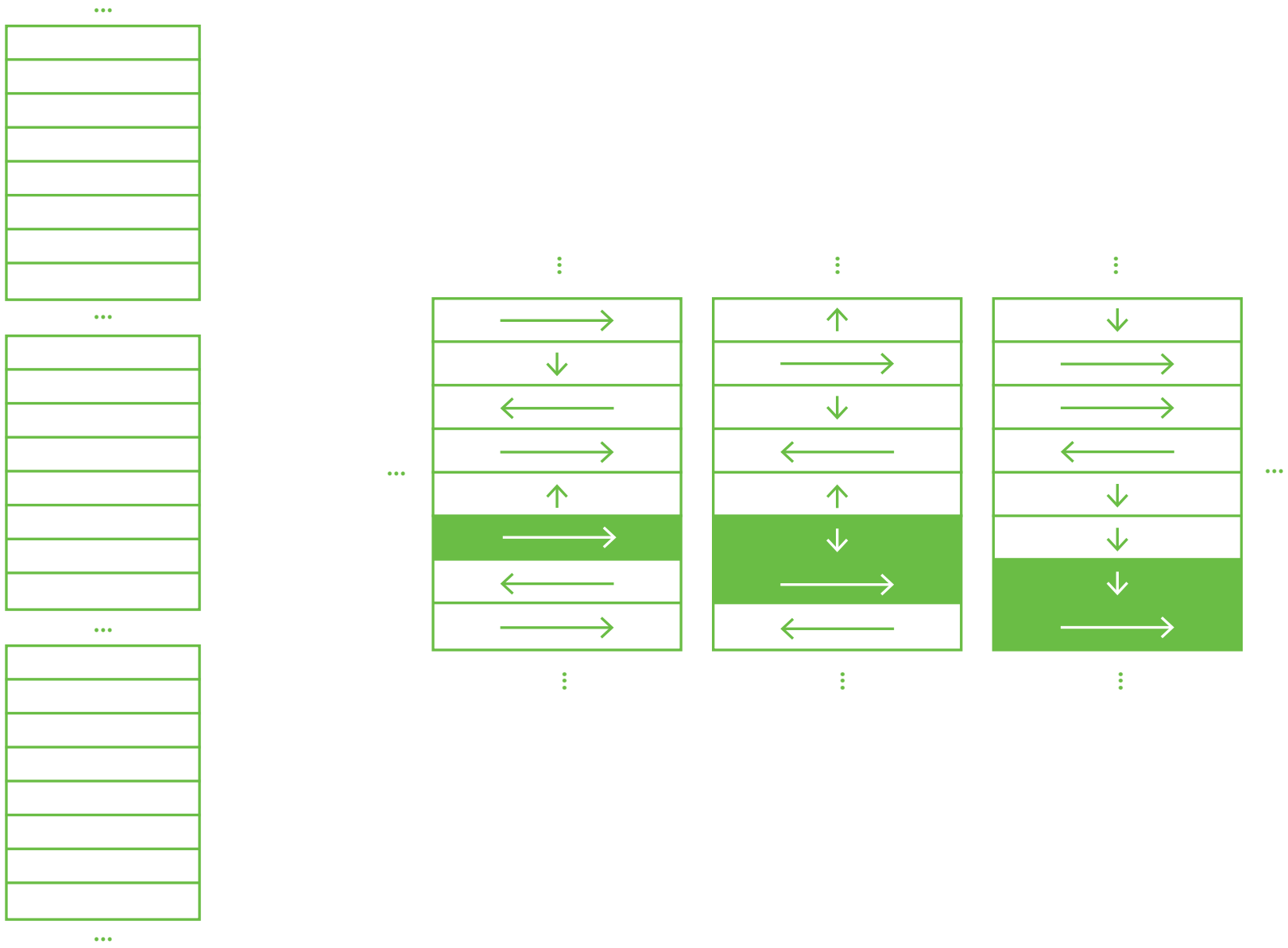
Memory is arranged as a matrix

Initialise memory with a bunch of mov instructions. as the program gets to each, it will travel in the direction pointed by that instruction, while overwriting its previous instruction with a different one (pseudorandom, or in the order 1, 2, 3, 4)

The entire process is, at all times, in danger of being eaten by the rest of the program

The performance ends when the program crashes, and the path is shown.





3 CONVERSATION

Throughout trying to experiment with these ideas, I've run into a lot of problems trying to get my (Mac) computer to do what I want. Looking at memory while a process is running isn't as straightforward as I had thought, with the main problems occurring if you accidentally try to access memory that doesn't belong to the current process. This is complicated by the fact that the organisation of memory has not been linear for a long time, and is instead woven from many interleaved processes running simultaneously. This interleaving is rationalised using a system referred to as 'Virtual Memory' where the computer program runs on an abstract, ordered model that is then translated into the actual bag of spaghetti that comprises a modern kernel.

As with many computer issues, I

ended up in a long, deep rabbit-hole trying to find what the problem was, and in doing so went deeper and deeper into the low-level processes that try and keep OSX running smoothly (my attempts to look at illegal memory here being considered 'non-smooth'). Eventually, via the LLDB (explain) and the signal library, I came to the Mach exception handling system.

Mach (a misspelling of 'MUCK' by an Italian in the 1990s, standing for Multi User Communication Kernel) is a system that sits around the OSX kernel, and is used, among other things, to handle exceptions. Exceptions are faults in a piece of code that do something illegal: trying to look at memory that doesn't belong to you is one, so is ??? and ???. These things are illegal for good reason: trying to look at other processes is a good way, especially in cloud computing, to see a lot of things you're not allowed to. That's because the cloud is of course, just somebody else's computer, and it's the same computer being used by a lot of different people. Mach is also the formal way by which different processes communicate information between one another -- and I think the eventual way to get this code to work.

This experience of trying to wrestle with my computer reminded me a lot of a computer game that I've always been interested in, called Core Wars. Written in 1984, Core Wars simulates a computer memory as one, large empty arena, in which programs written in a fictional assembly language called 'RedCode' might be run. The idea of the game is that two players write competing programs and launch them into this memory space, where each program proceeds to overwrite the other,

with the goal being to fill as much memory as possible. The real tactic, it turns out, is to make code that rewrites itself, adapting to its environment.

A central theme of cybernetics is that of the feedback loop: a relationship between an agent and an environment, or a system and another system, that regulates and controls. These systems are everywhere, and, in many ways, a computer is a fantastic model of a closed loop system, where the internal operation is effectively independent of the outside environment, bar a few controlled inputs and outputs. The concept of feedback was something I was thinking about a lot at the start of this project -- a relationship between an agent or a process, and the environment it operates in, how that environment regulates and changes it.

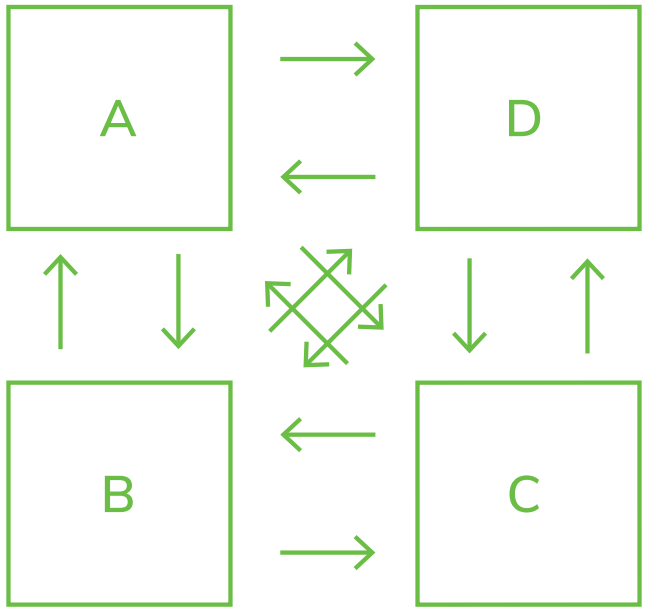
What is the feedback that exists between a computational process and its environment? Of course, that's a tough question; the environment is quite a slippery construct. But, if the environment is not just a static chunk of memory, and instead a system of other, competing processes, how does a single process relate to the others?

CONVERSATION PERFORMANCE

This performance borrows from a canonical piece in the history of cybernetics: Ashby's homeostat. The homeostat is a self-regulating system composed of four individual, but inter-linked bodies. Information can transfer between them perturbing one causes the others to correct.

Four computer programs run at once. each one regulates the state of the others, to make them use constant memory.

Processes must be shared between each program to keep them the same size at all times.



4. GRAPHICS

One of the other things I wanted to experiment with was the idea of graphical representation both as a way of understanding/ communicating what was happening with computer memory, but also as a fairly direct interface to memory itself. As quite a memory-hungry process, computer graphics code requires the writer to have a fairly full understanding of what transformations it takes to render computer graphics from first principles. The computer graphics ‘pipeline’ consists of several stages for rendering a simulated, imagined physical object into a flat, shaded representation of that on a computer screen. The real trick is, of course, that as one moves around an object, a new image (‘frame’) is calculated every time, thus giving the appearance that the player is moving in 3 dimensions.

The original object is represented as a set of ‘vertices’ in memory, each describing triangles that compose the object’s surface. These are stored in what’s known as a ‘Vertex Buffer Object’, and it’s this object that is used as the basis for all of the transformations required to make a 3D-rendered output.

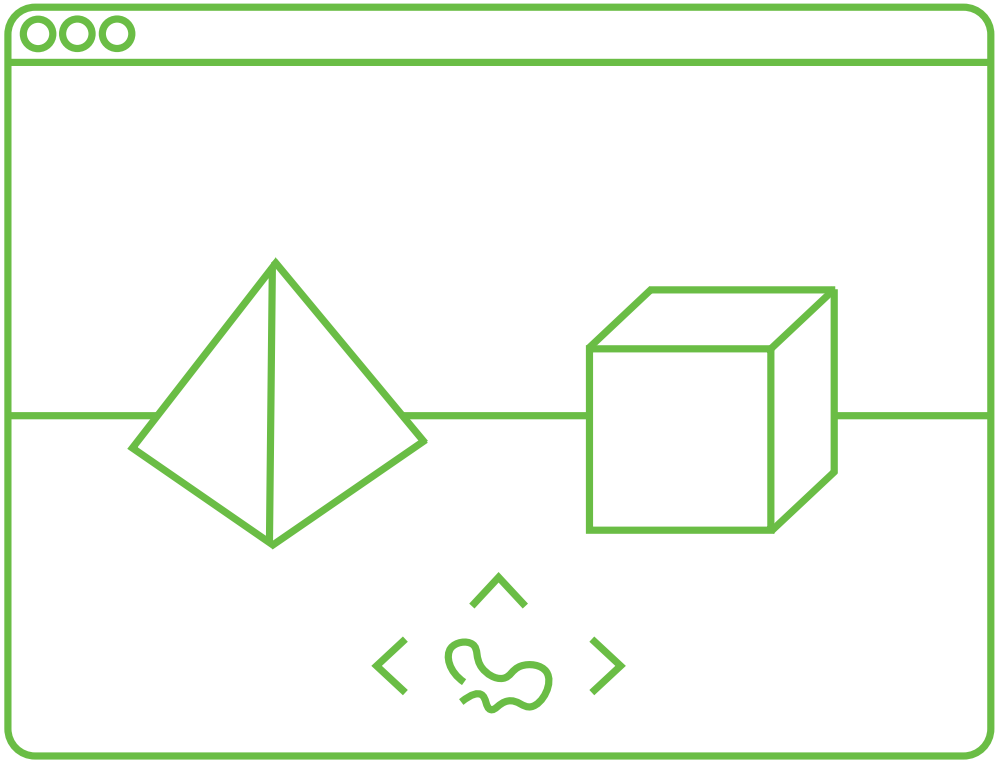
Much like the physical object (unless it gets deformed) these vertices remain constant, and all that changes is their representation. Other buffers are used to add detail: such as lighting and shading, the second action lending its name to the small computer programs that are called by the graphics pipeline -- ‘shaders’.

GRAPHICS PERFORMANCE

The game begins with a number of objects scattered around a flat plane. The player controls a small sprite.

As they navigate the plane, objects appear and disappear according to the amount of memory being used by the simulation. The game is won when all of the objects disappear.

The player begins to understand the embodiment of the computer program: as they walk their experience is restricted or expanded according to the constraints of memory.



I developed this work from the germ of an idea for which I would make a small piece, into a larger, more chaotic and sprawling assembly of ideas and no piece during my residency at Signal Culture. I'm really grateful to have had a space to think through and around these ideas, and have some new ones: it feels like such a luxury.

In particular, the things that I read here that inspired this work were: Daniel Tempkins' New Languages, and Iannis Xenakis' Formalised Music.

I also really enjoyed Paolo Soleri's Arcology, but mostly for the comfort that there was someone stranger than me in the room.

agnes cameron 2019