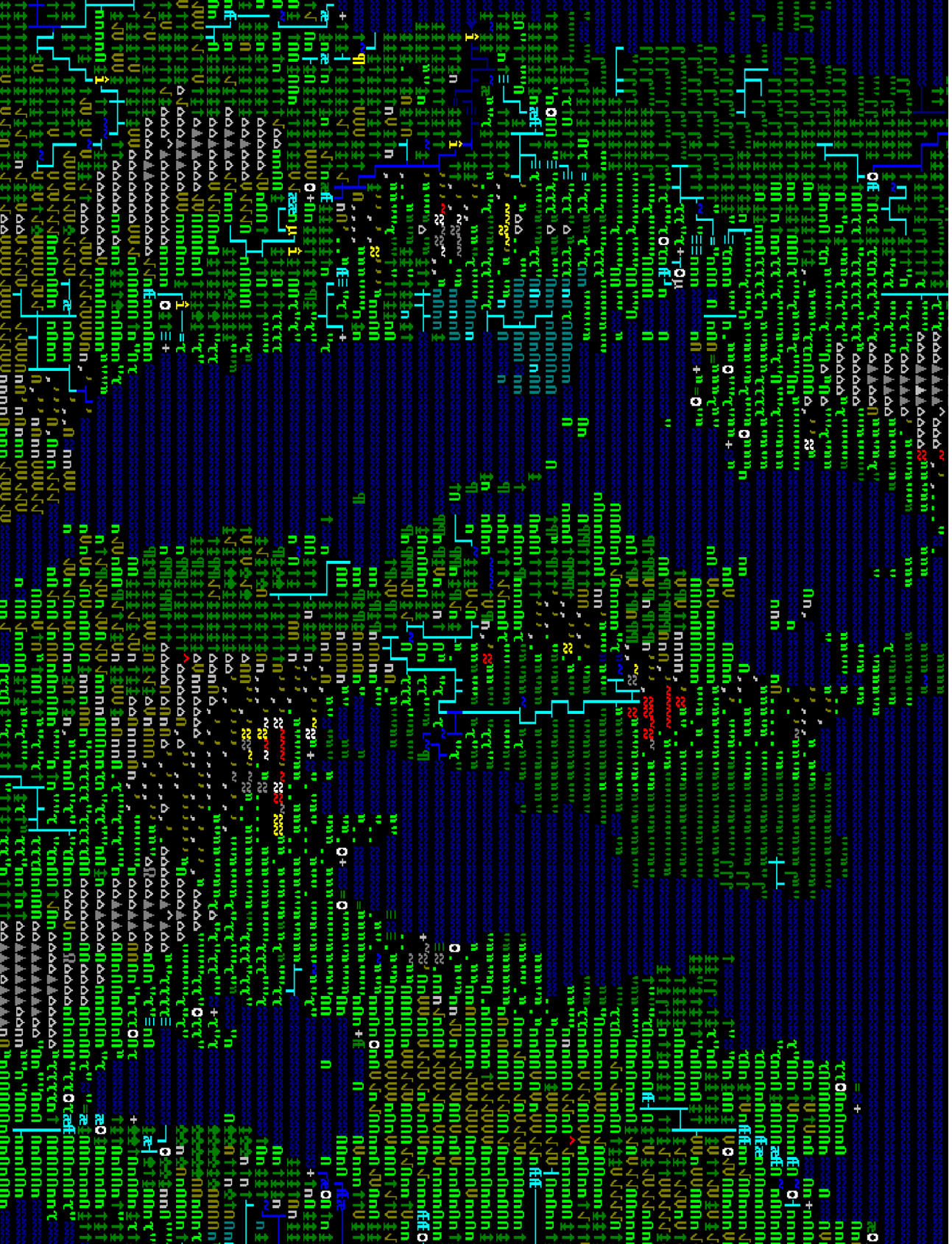


Hacklikes: Weird Interactions Between Things

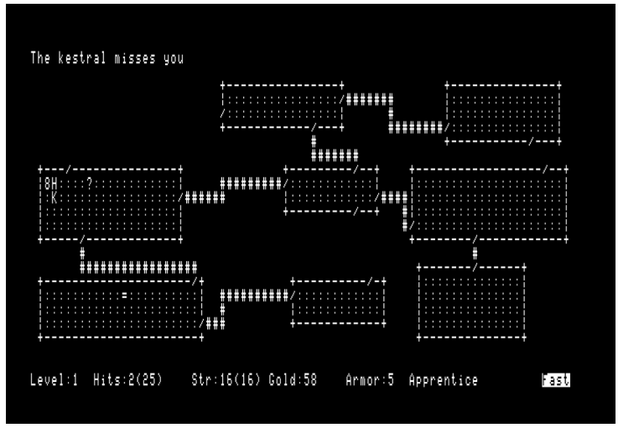
Jose Sanchez
Assistant Professor, University of Southern California School of Architecture
Director, Plethora Project

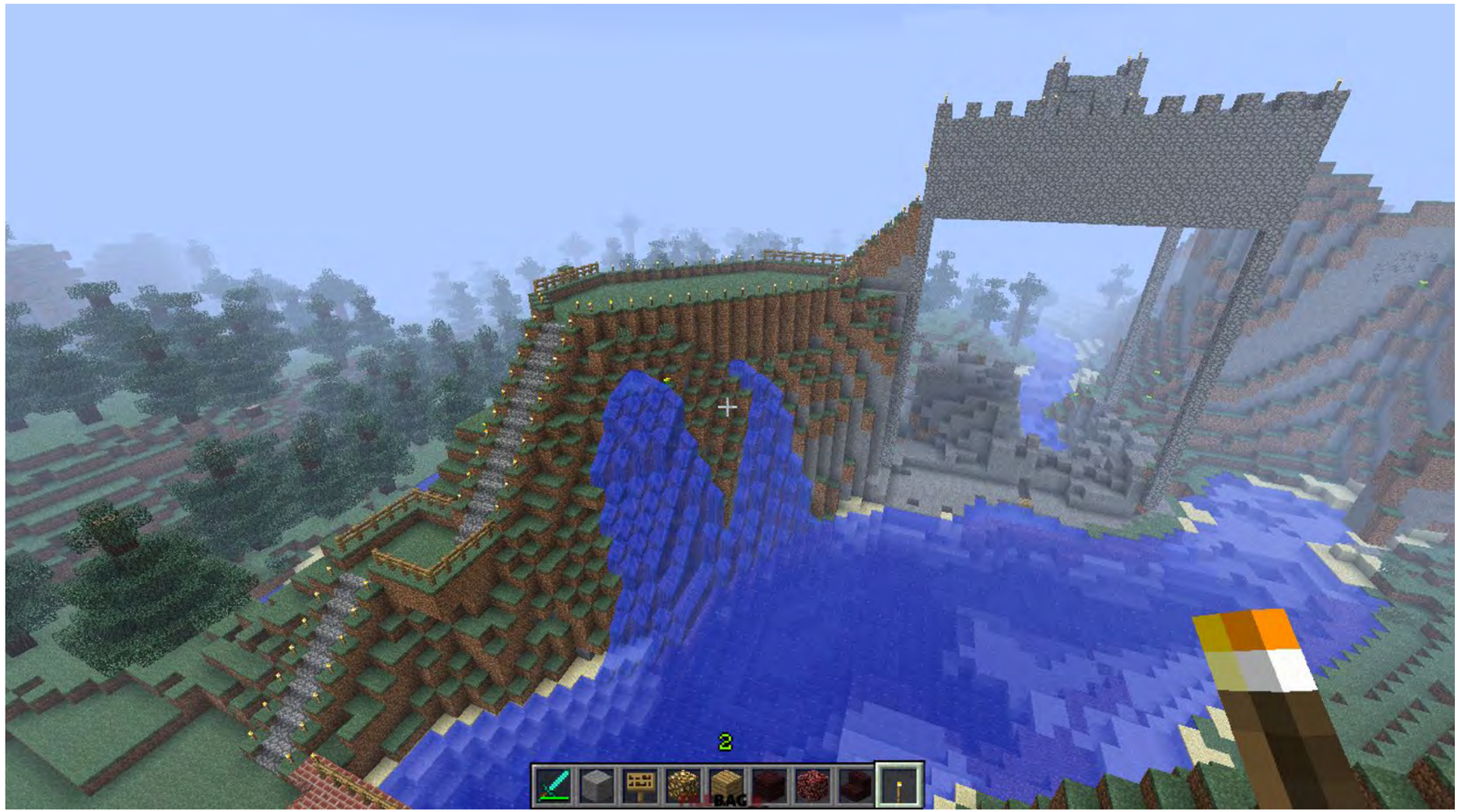
INTRODUCTION

While Roguelikes in general could be easily dismissed due to their usually rudimentary graphics composed by ASCII characters, most of such games contain an accumulated repository of algorithmic strategies to deal with procedural world generation and interactions between discreet units. The denomination 'Roguelike' is attributed to games that follow a series of principles first proposed by the game 'Rogue' 1980. In a Roguelike world, every 'tile' is an 'object' with properties or data. Each of these objects would be represented by the computer by a symbol allowing the player to differentiate a wall tile from a door, for example. The player possesses a limited visibility of the world and of the properties of objects, making exploration the primary objective of the world. Often, the only way of learning about the properties of an object is by trying to interact with it. The properties of the object will define if such interaction is resolved as a positive or negative effect to either of the units involved in it. Every interaction determines a 'turn,' allowing the simulation only to move forward once an action is performed. While the first games of the genre would focus on the interactions that would involve the player, later games have explored the idea that the relation between units is universal, player- or non-player-centric, allowing for the creation of vast algorithmic worlds that might never interact with a human player.

It is easier to understand Roguelikes by looking at the cellular automata (CA) algorithm by Stephen Wolfram.

Screenshot of Rogue – 1980





Screenshot of Minecraft

Such algorithm evaluates the 'state' of a cell or unit, and its immediate neighbors in a grid. The new state of the cell will be determined by rules that describe the state of its neighbors. Conceptually, the algorithm could be understood by a continuous interaction of data between neighbor units. Such interactions could be understood as 'transactions' from one algorithmic entity to another. Especially if we consider a CA written in an Object Oriented Programming (OOP) style, the transactions between 'objects' drive the simulation. The interactions can vary from flipping a Boolean state to altering of variables or killing the object altogether. A Roguelike game represents real properties of objects as several data fields within a tile allowing for more complex interactions and a larger inventory of possible tiles.

TIME VS. TRADE

The underlying concept of time within these simulations is discrete, as it is defined by the transactions themselves. There is not a continuous background time that allows the system to be played-out; on the contrary, only the interaction of one unit with another is what defines its temporality and (de)generative properties. This is easy to grasp by looking at the game of Tic-Tac-Toe; the game is constantly in a 'frozen' state. Nothing really happens until one of the players decides to make a move. Only that move can trigger the subsequent move, and then, slowly, the game plays itself out. Time is executed by the very objects interacting. This is why we could consider these simulations to be 'trade based systems' as opposed to 'time based.'

PROCEDURAL WORLDS

The universe of Roguelike games has developed into different directions, but for the most part, they are surrounded by a heavy repository of algorithms of world generation and data structures to handle procedurally generated worlds. One of the most iconic examples derived from some of the ideas living in the Roguelike community is Minecraft. Minecraft utilizes a large 3-dimensional voxel structure, meaning that there is a 3-dimensional grid of points, or tiles, each one representing a different object constituting a world. These objects can be collected and transformed by players using the right tools and combinations of materials. The simulation trades hi-resolution graphics for the vastness of a procedurally generated world with hundreds of different tiles for players to discover and build.

The video game Dwarf Fortress, by Tarn and Zach Adams, operates in a similar framework. Dwarf Fortress also uses a voxel structure, although it is constantly displayed in 2D. The player can navigate the different strata of the game at will. Each object defines a type of tile, like a 'soil' tile or a 'tree.' Each type in the game is represented with one icon in ASCII graphics. Additionally, each individual object has a series of properties, like resistance or humidity, making every tile potentially slightly different.

Dwarf Fortress offers the player the ability to influence and alter a world to its fitting. The key for Dwarf Fortress is the depth of the simulation engine. Each unit in the world is constituted by dozens of variables of data, qualities that define the physics and properties of every tile in the world. In the game, the interactions between units are not just simple rules like in the case of a CA, but rather a simulation between physical properties. The possible combinations of properties are vast, allowing for truly unexpected interactions. Mining a wall might end up in the discovery of a gold vent or in the flooding of your fortress due to the existence of an aquifer.

The whole world is procedurally generated once you start the game, simulating through hundreds of years of geological and cultural history—forking rivers through the land as seasons change, and the development of roads as new algorithmic civilizations conquer or decay over the landscapes of the game. After 250 years of game history (5 min. simulation) you are ready to start playing. In this initial stage, the player is just a spectator of the physics of the simulation. Geology and culture are developing and decaying to set the stage for every new version of the game. The algorithm displaying the procedural world generation process reminds us that the player is a small influence in such a world, and might not make much of a difference, dwarfing the ambitions to conquer and colonize the land.

Your objective as a player is to embark a small caravan of dwarves and build a fortress as you wait for more fellow dwarves to arrive. You operate seven dwarves that can perform different tasks, like chopping trees, digging caves, building chambers, or farming food, allowing the player to perform a series of tasks simultaneously and develop a design strategy.

The fortress needs to accommodate a series of needs; it needs to provide not only shelter for the dwarves, but also storage for the food or resources you obtain. It should host workshops and facilities to produce new items. Every little detail needs to be considered, as any little detail could trigger unexpected consequences.

Dwarf Fortress could be understood as a poli-performative design challenge; your concern of walking distances between areas is contrasted with the minimum amount of space that a dwarf needs to be happy and not develop bad thoughts. The proximity to water and drainage systems needs to be in consideration of the dry conditions that some food needs in order not to rot. Every

decision in the fortress is a negotiation between systems that, for the most part, operate parallel to each other.

As the Adams brothers declare it, at the beginning of every game, losing is fun, and ultimately, your fortress will collapse usually because of something you did not anticipate.

Playing and losing (the only way to finish a game of Dwarf Fortress) is a statement of the blindness between networks, an anti-holistic system critic to contemporary design strategies, even if the designers themselves have not presented it in such a way. There is no possible optimization in Dwarf Fortress, as agencies within the world might be utterly contradictory.

'Dwarves' are just a fun skin for an algorithmic agent that develops design tasks. All tasks that an agent can produce is the permutation or arrangement of matter in the virtual world. A robust economy of permutations allows for objects not to disappear for some arbitrary rule but rather simulate an encounter with another object and determine an outcome for such an interaction.

The robustness of the object-to-object interaction of the Dwarf Fortress architecture allows for more combinatorics than players could play over thousands of years, making the game a perpetual Pandora's box. Due to the fact that it is also fully procedural, each game is different and might force the player to adopt a different strategy to endure an inevitable demise.

What emerges out of this challenge are behavioral design patterns. A player will find a way to organize storage or arrange rooms for efficiency and circulation, allowing for creativity to avoid topographical randomness adopting it into features of the fortress. Dwarf Fortress has been so extremely rigorous over more than 10 years of development that players have been able to forge a community that keeps the game alive even as the creators still suggest that they need another 10 years to finish it.

Dwarf Fortress has become a sandbox for algorithmic interactions, not in the form of code, but in the form of 'design patterns.'

Screenshot of Dwarf Fortress, world generation

THE SANDBOX

There is a denomination for these speculative combinatorial games: the 'Sandbox.' While most sandbox games are described by the ability of the player to create or modify a virtual world, it is important that the nature of the creation is one of combinations. Very much as with Legos, creations emerge from putting together units or parts that already exist. It is in this context that new objects emerge of aggregates of discrete units. It is in digital sandbox games like Dwarf Fortress where the power of combinatorics doesn't only remain in the aggregation of physical objects, but is further expanded in the realm of algorithmic behaviors. For this, we need to consider that NPCs (non-playable characters) are also objects or building blocks in the simulation. Forces and attributes manifest their own agency in the world, pushing matter and decisions. A simulation with hundreds of computational objects interacting, each one of them with their own agenda, trading and altering the structure of the design output, is certainly far from the design strategies which we architects use to design nowadays.

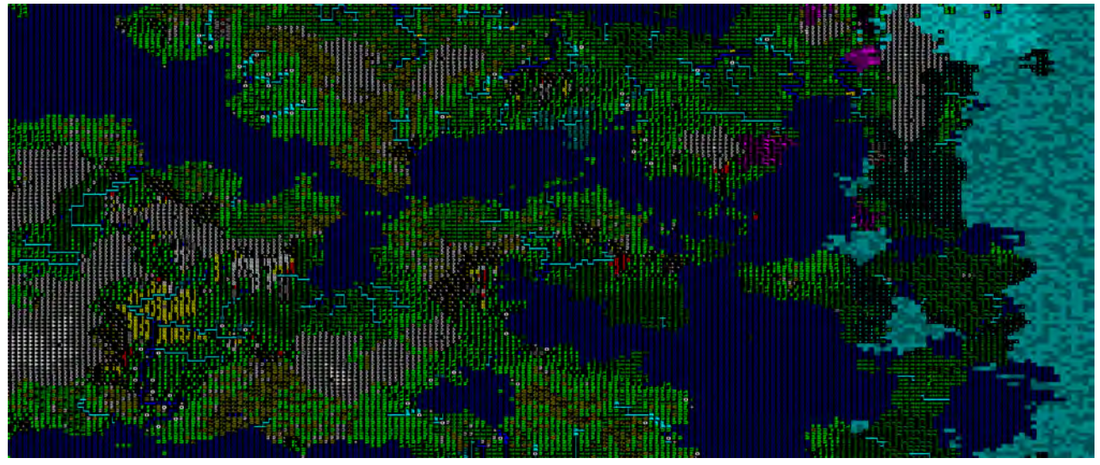
FLAT ONTOLOGY

When we analyze these games as 'trade-based systems,' we focus our attention to the objects, the actors that constitute the game. These objects are often just data, an algorithm that allows you to store a series of qualities in a "bucket" or unit. But objects could be much more than that; in a Roguelike framework, the units could be intelligent algorithms that interact both with the player and the world itself, influencing the player's decision-making or having agendas of their own, in complete isolation from the players' perspective. Is perhaps the realization that the player itself is just another object what allows us to talk of a 'flat ontology' within the simulation? Every unit stands on the same footing at the moment of the trade. Different agendas get played out simultaneously, often with unexpected consequences.

The philosopher Timothy Morton describes the relation between entities as 'strange strangers.' For Morton, the always-incomplete information available at the moment of an interaction between objects suggests that the outcome is always unexpected.

> *The strange stranger, conversely, is something or someone whose existence we cannot anticipate. Even when strange strangers show up, even if they lived with us for a thousand years, we might never know them fully—and we would never know whether we had exhausted our getting-to-know process. We wouldn't know what we did not know about them—these aspects would be unknown unknowns. (Morton 2010)*

It is this infinite unknown that allows for the inexhaustion of game-like simulations such as Dwarf Fortress. There is always a side of the object that 'escapes' the interaction, a side that withdraws.



Screenshot of Dwarf Fortress, resulting world

HACKLIKES

Within the genre of Roguelike games, we can find a subbranch with the denomination of 'Hacklike,' following the game 'Hack' from 1985. There are several conditions to be met to be considered a Hacklike, but I would argue that the primordial one is 'Weird Interactions.'

Unlike Roguelikes, which usually contain simple rule-based interactions between entities, Hacklikes expand the complexity of properties of objects, allowing for unforeseen outcomes.

Here, the term 'weird' is used in relation to the literary tradition initiated by H.P. Lovecraft. The weird has become a literary style that draws upon those unknown forces that exceed human understanding. In the case of Lovecraft, this was a tool for horror, but the weird could be understood in a broader sense as the exploration of the withdraw space between objects.

> *Lovecraft stresses that the 'true weird tale' is characterized by 'unexplainable dread of outer, unknown forces.' (Mieville 2008)*

Hacklikes place the player in a testing ground. Every gameplay will inevitably explore new areas of the simulation, by confronting the player with qualities that he might have never interacted with before. The simulation is non-deterministic not because interactions might yield random results, but rather because a player will never be able to anticipate the outcome of every encounter.

For Graham Harman, Lovecraft's Weird Realism offers a non-reducible model, one in which the simple awareness of that which we are not exposed to forfends the forecast of a transaction.

> *No reality can be immediately translated into representations of any sort. Reality itself is weird because reality itself is incommensurable with any attempt to represent or measure it. (Harman 2012)*

Dwarf Fortress uses the 'inevitable death' mechanic to expose the flaws of a holistic systemic approach, one in which units could know and understand each other. Every game has one inevitable conclusion, the ultimate destruction of your attempt for control. This will often happen because of some unforeseen event—something in the blind spot of the player, something that wasn't programmed or planned.

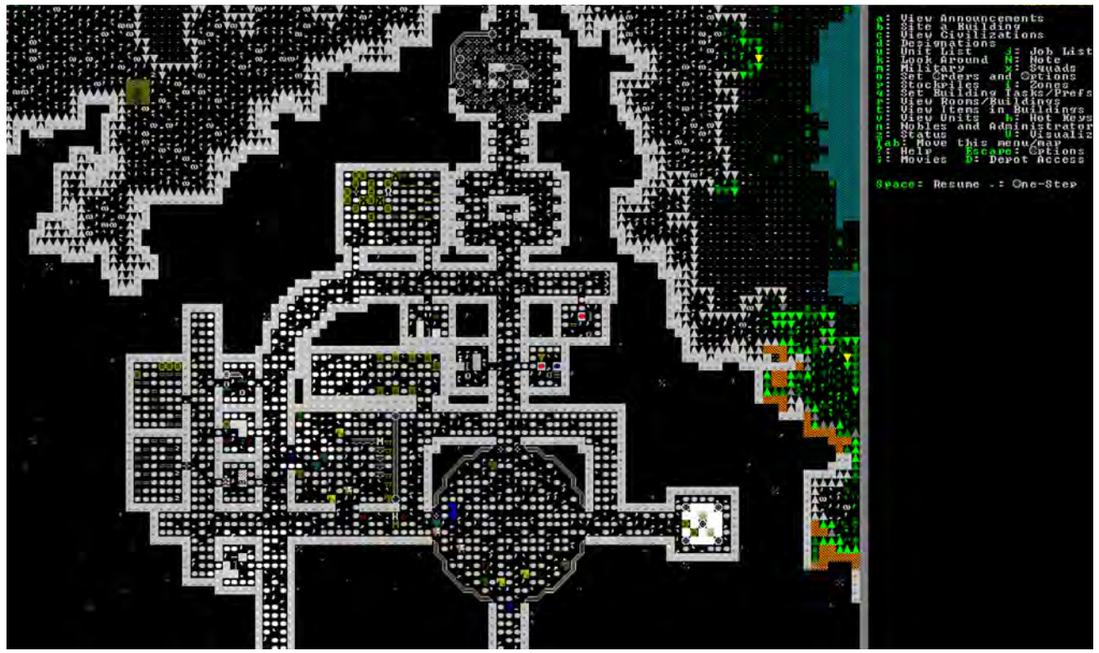
Dwarf Fortress teaches us that even robust systems of control are incomplete and destructible.

NOT EVERYTHING IS A SYSTEM

It is considering these points that it becomes difficult to talk of a 'system,' as the emphasis in the simulation is awarded to the blindness between units rather than their relations. Ian Bogost suggests the mechanism of the 'List' to define non-holistic aggregates with no common interest. Lists offer the view of an inconsistent collective, one that does not necessarily create a coherent whole.

> *Lists offer an antidote to the obsession with the Deleuzian becoming, a preference for continuity and smoothness instead of sequentially and fitfulness. (...) By contrast, alien phenomenology assumes the opposite: incompatibility. The off-pitch sound of lists to the literary ear only emphasizes their real purpose: disjunction instead of flow. Lists remind us that no matter how fluidly a system may operate, its members nevertheless remain utterly isolated, mutual aliens. (Bogost 2012)*

The emphasis on the isolation between units provides for a framework in which opposed ideas could co-exist



Screenshot of Dwarf Fortress, player-generated fortress

without attempting to override each other or merge into a style: co-existence over convergence.

Simulations such like Dwarf Fortress succeed in creating enough tension and autonomy between building agencies to allow us to think that such simulation is not biased, attempting to prove a specific goal that was already clear from the beginning, but rather a design tool for the exploration and learning of new strategies—a space where the unexpected could be terrifying but also a generator of novelty. Dwarf Fortress pictures a designer *entangled* in the simulation rather than a spectator.

OBJECT ORIENTED DESIGN

When Graham Harman took the name Object Oriented Philosophy for the denomination of his philosophical agenda, he borrowed the name from computer programming culture without pointing to a direct relation to OOP. It has been Ian Bogost who has come to develop a comparison, detecting that although similarities between the two approaches do exist (Bogost 2009), there are also several inconsistencies. Regardless, object technology and the programming paradigms developed by Alan Kay do provide a powerful operational framework for design, one in which many of the Object Oriented Ontology (OOO) ideas could be explored.

As Bogost explains:

> *Software must exhibit four properties to be considered Object Oriented: abstraction, encapsulation, polymorphism and inheritance. Abstraction is the programmatic representation of an object, disassociated from any specific instance; only modified or instantiated version of an object model or class actually exist. Encapsulation means that the content of the software is hidden from other parts of the system. Polymorphism means that different derived instances of a class can be different behaviors. And Inheritance means that the class itself can be used to create other classes, which adopt or inherit the parent's classes' structure, attributes and behavior. (Bogost 2008)*

Some of these object oriented attributes like abstraction and encapsulation have a direct implication on the way we conceptualize and design, while polymorphism and inheritance have direct implication on the design and community culture, and the propagation of knowledge. The format of OOP is rigorous, as it attempts to allow communication from diverse different authors. It effectively becomes a language for operational ideas that could be read and understood by humans and executed by computers. Many designers, including myself, have started developing object libraries: lists of computational objects living in a pre-design space. These libraries, often having author denominations, constitute an ever-growing toolset for computational design that allows us to trace how design outputs hybridize concepts from different sources.

It is with object libraries that the design discipline has been able to start building a heterotopian and operational landscape of ideas, one in which the building blocks take the form of code and could be combined in infinite number of ways—computational design objects that are described in a language for potential interaction.

NetHack 1987 by Mike Stephenson – a 'Hacklike'

It is precisely this state of pre-interaction of objects where the lists or libraries of algorithmic ideas take their stronger form. The design objective of such libraries describes a pre-actual medium that allows for an intense propagation of knowledge. The inevitable game of combinatorics that will place some of these objects together will always remain a fraction of the list they came from. To this regard, Luciana Parisi describes algorithms to a new actuality for architecture and design.

> *Algorithmic architecture is not a whole constituted by parts, but rather shows that parts are irreducible inconsistencies divorced from the totality that can be built through them; it works not against but rather with the chaotic parts of information that are comprised neither within mathematical axioms nor within the law of physics. (Parisi 2013)*

In this context Open Source is not only a political stance in relation to the democracy of knowledge but the very medium in which computational objects defined their vector of propagation.

It is perhaps in times where the discipline is threatened with convergence that it becomes relevant to acknowledge the limits of the medium in which we operate. The incompleteness living at the core of computational strategies is a lesson that games like Dwarf Fortress strongly point out. It is perhaps necessary to allow for computation to open more speculative attitudes towards combinatorics of logic and desire without the necessity of complex formal output, but rather a search and exploration of the weird, the unexplored territories of contemporary design.

REFERENCES

Bogost, Ian. 2012. *Alien Phenomenology*. Minneapolis: University of Minnesota Press.

—. 2009. *Ian Bogost – Videogame Theory, Criticism, Design*. 16 July. Accessed September 22, 2013. http://www.bogost.com/blog/objectoriented_p.shtml.

—. 2008. *Unit Operations: An Approach to Videogame Criticism*. Cambridge: MIT Press.

Harman, Graham. 2012. *Weird Realism: Lovecraft and Philosophy*. Winchester: Zero Books.

Mieville, China. 2008. "M.R. James and the Quantum Vampire." In *Collapse IV*, by Robin Mckay. Falmouth: Urbanomic.

Morton, Timothy. 2010. *The Ecological Thought*. Cambridge: Harvard University Press.

Parisi, Luciana. 2013. *Contagious Architecture: Computation, Aesthetics and Space*. Cambridge: The MIT Press.