A Generative Framework of Generativity

Kate Compton, Michael Mateas
University of California Santa Cruz, Expressive Intelligence Studio
kcompton, michaelm@soe.ucsc.edu

Abstract

Several frameworks exist to describe how procedural content can be understood, or how it can be used in games. In this paper, we present a framework that considers generativity as a pipeline of successive data transformations, with each transformation either generating, transforming, or pruning away information. This framework has been iterated through repeated engagement and education interactions with the game development and generative art communities. In its most recent refinement, it has been physically instantiated into a deck of cards, which can be used to analyze existing examples of generativity or design new generative systems. This Generative Framework of Generativity aims to constructively define the design space of generative pipelines.

Introduction

What is generativity? To the procedural generation community, it often means the method of constructing content for use in a game. Game-creators seek to generate content to fulfill many conflicting constraints: content that is novel yet not game-breakingly novel, content that may either fill out the background of a game, or radically change the gameplay. There are established tools that are of use in this pursuit. Many in this community are expert practitioners with Perlin noise, L-Trees, grammars, tile-based placements, or genetic algorithms. The particular constraints of game content may be unique to games, but these generative methods, and their issues and a affordances, are common across many fields(Compton, Osborn, and Mateas 2013).

Generative music, computational creativity, parametric architecture, and many other fields all use these same methods. Additionally many successful generative systems are hybrids of many kinds of generative methods, chained together. Games like Spore (Hecker et al. 2008), No Man's Sky (McKendrick 2016), Dwarf Fortress (Harris), and SimCity (Wilmott, Quigley, and Moskowitz 2012) each have many intersecting pipelines of generativity, in which the output of one subsystem becomes the input of another. Some subsystems contain cellular automata (Dwarf Fortress and SimCity). Some use inverse kinematics (Spore and No Man's Sky), some simulate particles or agents in a field of forces (Spore terrain generation, SimCity transportation). Some games use Perlin noise to create terrain (Dwarf Fortress and No Man's Sky). In games and other interactive works, humans interactors may be part of this pipeline!

This paper describes a way of thinking about generativity as a pipeline of data transformations, constructed from methods that take one form of data, and return data that is annotated, transformed, compressed, or expanded. Although there are many frameworks of generativity (in games and elsewhere), this framework attempts to provide a system to design and describe generative pipelines in a way that captures how generativity is able to take input (random numbers, mouse input, and more) and transform it into something new and surprising.

We proposed a framework shifting the focus from *PCG as game-content* (and how it is used) to the *pipelines of generativity*, and how they apply across fields. This broader lens on generativity has allowed the first author to work in conversational agents, bot-making, interactive art, and generative textiles using the same tools developed for games [1]. We believe that formalizing this approach is a useful contribution, so that others, in games, art, generative text and other fields, may also use these methods. In the latest experiment, this framework has been developed into a deck of cards, which were recently tested as a tool to help students understand the design of generative systems (Compton, Melcer, and Mateas 2017).

Related Frameworks

There is no shortage of frameworks available for describing generative systems, especially for systems used to create game content.

The framework by Hendrikx, et al, (Hendrikx et al. 2013) classifies what can be generated by a generator: sound, textures, vegetation, buildings, behavior, volumetric fire-water-stone-and-clouds, outdoor maps, indoor maps, bodies of water, ecosystems, road networks, urban environments, entity behavior, puzzles, storyboards, the story, levels, system design, world design, leaderboards, and news. They also define five categories of tools: pseudo-random number generators, generative grammars, image filtering, spatial algorithms (tiling, fractals, Voronoi diagrams), modeling of com-

[1] `tracery.io`

plex systems (cellular automata), and artificial intelligence (genetic algorithms, ANNs, constraint satisfaction), and provide examples of pairings of method and content, such as L-systems for trees. This style of framework provides a well-structured way to catalog the PCG applications of past games, and offer recommendations for the best approaches, such as using Perlin noise to generate cloud textures.

However, the possibility space of games is ever-expanding, and many exciting game designs use content in novel ways (such as object design in Katamari Damacy, or the line-of-sight puzzles in the Witness). A Lindenmayer system, as a context-free grammar, has no concept of the outline of the tree that it generates, or its branches' locations relative to other objects, and so could not address the gameplay needs of a tree for Witness-like spatial puzzles. It is easy to think of many kinds of content not described by this ontology (social customs, pottery, poetry, language, memes, recipes, crochet patterns) which might be of great importance to some future game. Each one of them might be part of a gameplay, a story, or the background of a game. While this framework classifies *common* uses, it cannot predict or help design novel uses. Additionally, it is easy to create a description with specification gaps: as exemplified by the statement that "building floor plans can be procedurally generated using PRNG techniques". How to translate a number into a floor plan is a difficult puzzle, one which is not explained by this framework.

In the framework of "Search-based Procedural Content Generation: A Taxonomy and Survey" by Togelius et al(**?**), the process after generation is inspected through three lenses: what kind of content is generated, how it is represented, and how the content can be evaluated. This framework intentionally does not address the methods used in creation, specifically how the representation of a piece of content (its genotype) can be translated into the finished artifact (the phenotype). Instead, their focus is on the content evaluation, heuristics (how desirable or apt for its purpose) the piece of finished content is. They, like Hendrikx et al, list several kinds of content (trees, terrain, racetracks), and they decline to add storytelling to their taxonomy, as it has not yet had a search-based generative implementation. While it necessarily catalogs only existing systems (and ones the authors are familiar with), in this case, unlike the Hendrikx classification, the steps to implement a search-based algorithm would be useful even for unknown applications.

In Gillian Smith's "Understanding procedural content generation: a design-centric analysis of the role of PCG in games" (Smith 2014), Smith divides procedural content generation into five categories:

- generation as optimization
- generation as constraint satisfaction
- generation with grammars
- generation as content selection
- generation as constructive process

The last category, constructive process, refers to generators that "build content in an ad hoc manner by piecing together customized building blocks", but uses Rogue level-generation (a series of sequential operations on a grid of empty-or-solid cells) as an example, so it is unclear if the building blocks refer to steps in an algorithm, or actual pieces of content, in which case it may be more closely related to a grammar. The framework further breaks the parts of a generator into experiential chunks, templates, components, and subcomponents, human-authored sections of varying sizes of granularity and emergence. Unlike the other frameworks, this one specifically addresses how the player affects the generator: no influence, setting parameters, selecting from generated content, or directly manipulating the artifact itself.

As we want to model not only games (and static game content) but also interactive generative art and generative art tools (as well as works like Panoramical[2] and the works of StrangeThink[3], which exist somewhere between them), it is worth checking in with our neighboring field of generative art. Unsurprisingly, there are as many frameworks for generative art as there are for PCG in games. "A framework for understanding generative art" by Dorin et al (Dorin et al. 2012) has many features that are well suited to our purpose here, including a focus on generativity, a way to represent methods, and a recognition of interactors and viewers/experiencers as part of the generative system.

They propose a framework that looks at each generative artwork as a construction of four types of components: entities, processes, environmental interaction, and sensory outcomes. Entities are persistent state-carrying units with individual or shared attributes, like cells in Conway's Game of Life or agents in a simulation. Entities are acted on by processes, actions with initialization conditions and termination conditions. Interactors, the creator, or other environmental forces can set conditions or parameters or entity attributes: these environmental interactions are "flows of information between the generative processes ... and their operating environment."Entities and processes are captured in static form ("snapshots", "endpoints" or "accretions") or viewed continuously "live". All entities and outcomes may be *visible* ("flat" system) or *translated*, a translation with may be structurally similar ("natural mapping") or arbitrary (for example, cellular automata mapped to musical notes, lights, or knitting).

Of all the previously cited frameworks, this framework takes clearest note of the interesting property of generative systems: they are a pipeline of information and processes. A generative system may transform numbers into content selection into 3D model into a rendered image, but there is also always the possibility to interrupt the system, or add interactivity, by inserting a human interactor or viewer.

A framework that represents generativity by a pipeline of methods can also represent generative systems *outside* the digital, and outside of the human-constructed artifact. The spiral of a gazelle's horn to spiral or the spots on a butterfly's wing can be described in terms of the chains of physical and chemical algorithms (Ball and Borley 1999). Likewise the transformations of digital generativity can be described

[2] `http://panoramic.al/`

[3] `https://strangethink.itch.io/`

in terms of biological processes, as by McCormack, et al (McCormack et al. 2004).

In our framework, though, we consider the specific properties of these transformations. What data types goes into the transformation? What types of data come out? This specificity and concreteness allows our framework to model constraints in a way that is closer to the implementation of a system. This framework enables us to examine the algorithmic details of generative system through the lens of data transforms.

A framework for generativity

The framework that we are presenting in this paper is intended to represent the compression, expansion, extrapolation, and generation of information that occurs in the running of a generative algorithm. To that end, we propose a lens on the components that make up generative systems that separates the ways that they construct content from the ways that that content is used or optimized.

Adam Smith (Smith 2012) divides PCG techniques into two broad categories, additive and subtractive techniques, after the terms used in sculpture for building up a form, and carving away undesired parts. Our framework intends to separate these full systems into their component pieces: pipelines of many transformations that add or expand information, and some algorithms and methods that *consume* pipelines. Generative pipelines almost always contain many such transformations, as it is necessary to construct content (or a space of content) before subtracting or searching over the space. This framework is not meant as a litmus test to decide whether a particular pipeline is generative, nor whether any algorithm or substep is itself a generative transformation. "Generativity is often not identifiable in individual components but in the overall transformations through the pipeline. Each method in the pipeline may expand, augment, reduce, or transform the data from a previous step, in a way that in isolation does not feel generative. But through the construction of a pipeline of many of these methods, the pipeline itself becomes generative.

Each transformation has a set inputs and output (a property that allowed us to build a dominos-like set of cards to model these pipelines (Compton, Melcer, and Mateas 2017)). Often each category shares an input/output set (notably not "Geometric Transformations). Each category also defines the *suitability* of methods to different situations. Markov chains and grammars are often confused by novice text-generation creators, understandable since they both output generative text. But one requires a corpus of existing data as input, and one requires hand-authorship of rules, so they will be useful in diverse situations.

In this paper, we propose eight broad categories of transformations which can be use to build a generative pipeline: **content selection, tiles, grammars, parametric, geometric transformations, distributions, agent-based (particles, cellular automata, flow, and graph-based simulations), and machine-learned statistical models**. Each of these categories shares some useful properties, but may contain many separate algorithms and techniques. Additionally, we identify three broad **consumers** of finished pipelines: search-based techniques, constraint solvers, and human interactivity.

Categories of Transformations

Content Selection Though not often considered PCG in itself, content selection is commonly a component in these pipelines of generativity. If a 20-sided dice is rolled, that action selects from 20 possibilities, as does drawing from a deck of 20 cards. A content selection method has some logic of selection, whether random, or random with changing probabilities (like cards), weighted randomness, or more advanced non-randomness based logic, such as sorting content based on some heuristic. It may even represent a request for external data, such as asking for the last tweet sent to a source, or the most popular newspaper headline

Requires: some collection of content
May require: a heuristic for selecting content
Returns: some content

Tiles Perhaps the most "classic" form of generativity is tile-based generativity, in which available slots are filled with some selection of content. A historical example of this is a tarot spread, in which from 78 cards, 10 are drawn and placed in a Celtic cross pattern, and the choice of each card in each position may be interpreted into a personal meaning. Another historical example is the Musikalisches Wurfelspiel (Hedges 1978), a tile-based music generator from the 1700s. Similarly, each cell in Rogue is a slot which can be empty, or full of a wall, monster, or object. Some tiles have constraints: a Settlers of Catan board must use all the tiles, but ocean and harbor tiles may only be placed in specific locations. Open tiles are recursive: one tile may contain a template to fill with more tiles (i.e., equipping a socketable weapon in Diablo II) but this may be better modeled as a grammar.

Requires: A set of possible tiles, a template of slots or sockets
May require: selection logic (see "content selection" above), constraints (which tiles can fill which sockets, orientation or proximity constraints)
Returns:the selected tile (or none) for each socket

Grammars Grammars recursively replace non-terminal templates with either more non-terminals, or with terminal symbols. Often, grammars require a selection from multiple possible expansion rules. Tracery (Compton, Kybartas, and Mateas 2015) is an example of a system designed to handle text-generation with user-authored grammars,

Requires: A set of replacement rules (a grammar)
May require: non-context free rules that govern preferences or constraints (if so, some form of a constraint solver is required)
Returns:A tree of recursive choices.

Tracery returns not only a tree of recursive choices, but also a flattened depth-first traversal of the finished text of each node. Grammars themselves *do not return finished content*, they still require some additional methods to compress that tree into the final flattened content. A grammar may generate, SVG text, for example, but a browser is still required to interpret those parametric geometry instructions to draw an image on the screen.

Parametric A parametric generative method is one which converts some numerical values to content (even if the content is another numerical value). This conversion may be as simple as turning a value into the sine of that value, or turning two values into the Perlin (or Worley) noise value at that point. But many unusual and custom functions can be created. The "superformula" (Gielis 2003) is an equation that can describe a striking range of 3 and 2 dimensional shapes using 6 parameters, and was considered for use in No Man's Sky (McKendrick 2016).

Since Lindenmayer system are so common in game flora, any taxonomy cannot neglect to include them. Interestingly, an L-system is a hybrid: a computational grammar, with parametric values defining the parameters of each grammar replacement.

Parametric methods are, in fact, often less of a category of generative method, and more like an interface which they may implement. In "Navigating the Constructive Space", below, we discuss some properties of certain parametric methods which make them particularly expressive for generativity and interaction. In "Example systems", several simple parametric generators are dissected to reveal that a parametric generator may be implemented with an L-system or a particle system.

Requires: An array of floats, usually normalized

Returns: Widely variable: L-systems create graphs in space, but other parametric generators may create curves (superellipses), 3D geometry, or waveforms or vectorfields.

Geometric transformations Unlike parametric methods which take numbers, some functions take other forms of geometric information and process it into a new kind of geometric information.

One familiar form of this is a Voronoi diagram. The Voronoi algorithm, given an array of points, calculates the region graph (a set of edges and nodes) that represents the region of space closest to each point. A Delaunay triangulation also takes a set of points, but returns a triangulation of those points. Many other speciality triangulation algorithms exist for various needs and dimensionalities.

Some methods take 3-dimensional points and turn them into triangle meshes, in a generative and emergent way, like Metaballs. Likewise, a region of arbitrary voxel data can be turned into a triangle mesh with voxel polygonization algorithms, a key step in No Man's Sky's generation of renderable terrain (McKendrick 2016).

There are even some simple, often overlooked generative methods by which curves are turned into more complex curves, or geometry. Offsetting a Bezier curve is one such method. This seems too simple to be worth noting, but being able to remap geometry along a curve is the foundation of many digital painting tools. Painting tools work as either deformed geometry or shapes along a curve, or distributions of images or geometry along a curve (Smith 1995). This technique is visible in Adobe Illustrator brush strokes, but it is also used in the physically impossible generative 3D paintbrushes of Google's Tilt Brush VR program. These techniques require some existing geometry, such as lines, curves, graphs, or 3D meshes, and return a different or extrapolated geometry.

These methods not easily categorized by their input and output data. Each one often takes a spatial or relational object (such as vectors, a 3D model, or a graph) and returns some other spatial or relational object.

Distributions Distributions, in this case, refer not just to the probability of content being selected (i.e. a uniform or normal distribution), but how it is placed spatially. Placing trees over a landscape requires not only deciding which species and which size of tree to select, but also where to place trees of that size and species in relation to the landscape and in relation to each other. Similarly, distributing notes in music is not spatial, but still requires the system to select notes from the current key and place them relative to other notes. A sufficiently sparse tile-based system (very few tiles to place, many possible sockets) can be treated like a distribution. Consider a roguelike world of 10x10 tiles. This may be considered as a grammar (each tile is a socket that can be filled). But if the space is expanded to 1000x1000, most tiles may be empty, and by the time the space is as large as Minecraft, it may be more practical to treat the space as a continuous space that can receive distributions of objects, rather than as several billion sockets.

Many different heuristics can be used to tune a distribution, such as suitability, but may also include aesthetic logic such as:

- **greebling**: a term originating in Star Wars production history (De la Mar), meaning that the uneven distribution of object over a surface makes the surface look "real"
- **footing:** the tendency for the intersection between two naturally-occurring objects to be noticeably different than non-intersection spaces, such as grass growing higher against a fence post, moss accumulating the the cracks of boulders, trash getting caught against a park bench, or sand pooling where a river bank meets the river.
- **barnacling:** a term-of-art on Spore used to describe the aesthetic value that a monumental object is often best framed by smaller simpler versions of itself, or by some contrasting object.

These aesthetic choices may suggest that distribution occurs only in landscapes, but is useful as well in generative art, music, decoration or fabric design.

Requires: Content to be distributed, rules to govern selection and distribution,

Returns: Location (and possibly other annotations) of the new positions for the content

Particles, cellular automata, and graph-based simulations Cellular automata are grid-based cellular systems where each cell represents an agent with rules for updating. The origin of the concept is in Conway's Game of Life (**?**), but it is also used in Dwarf Fortress, Minecraft, and in the pollution and water models of SimCity (Wilmott, Quigley, and Moskowitz 2012). It is also the basis of a platform for running emergent simulations to make arguments by Nicky Case[4].

[4] `http://ncase.me/simulating`

When agents are liberated from the grid and can move freely through space, they become particles. A particle has physical properties like velocity and acceleration, and may also have other properties that define how they respond to forces. Almost every particle system has some forces being applied to the particles, such as gravity or drag, but often those forces represent non-physical forces, such as flock cohesion and avoidance in the Boids algorithm (Reynolds 1987). Applying complex forces to particles can generate interesting patterns that respond to external geometry. Particles were used to texture the planets and creatures in Spore: as particles moved across the geometry, they acted as paintbrushes, painting into the texture map, using distribution of stamps or deformable brush geometry extruded along their paths (Compton et al. 2007).

While cellular automata are confined to a grid, and particles can move freely through space, some agents are confined to paths on a directed graph, or even simulated as continuous flow through that graph. The difference between flow and agents in a directed graph is ambiguous: SimCity's transport system implemented both with the same system (Wilmott, Quigley, and Moskowitz 2012). Loopy, by Nicky Case, is a platform for designing emergent simulations based on direct graph flows [5]. Similarly a simulation of emergent behavior in game systems can be modeled as a directed graph, as in Dorman's Machinations (Dormans 2011). Sometimes multiple simulations in each of these styles will interact with each other. In SimCity, the distribution of new houses along a graph of streets is driven by the values of cellular automata maps, each of which has its own rules for how to update (groundwater pollution diffuses downhill). Those houses create Sims, which route themselves as particles or along the streets, as the houses siphon energy off an energy graph, and write into the pollution map (Wilmott, Quigley, and Moskowitz 2012).

Requires: A directed graph with update rules, or forces and distribution(particles), or cell update rules (cellular automata)

Returns: Particle trails, or the new states of the graph or grid-cells

Machine-learned statistical models So far, all the rules discussed for various systems (grammars, graph simulations) have been hand-authored. But there are systems which generate rules, and those, too, can be part of the generative pipeline. Markov chains and grammars can create similar-looking output, but a Markov chain (at least, a machine-learned model) requires an input of content to learn its statistical rules. Beyond Markov chains, the PCG community is beginning to experiment with using deep learning to generate content (Summerville et al. 2017). For example, Deep Forger [6] is a bot that has trained on classical artworks until it has generated a flexible statistical model of what colors go where in a Picasso. When given an image from a user on Twitter, the model can be used to iteratively optimize that image towards its ideal of Picasso-ness. Since the use of deep learning in PCG is relatively new, the community is still experimenting with where it will fit in the workflow. However, with this framework, there is a clear input, and a clear output: surprisingly, "machine learning" is actually *two* components, which we can use separately.

Model construction:

Requires: Lots of content:

Returns: A model representing a set of statistical patterns in that content

Transformation via model:

Requires: A trained model. Deep-learning activation-optimization may also require a starting piece of content to optimize, as Markov models may require a starting sequence.

Returns: For a Markov model: a statistically-viable walk through a Markov chain. For a neural network: an activation-optimized version of the source content

Consumers of generative pipelines

The previous section defined broad categories consisting of many transformations that can be used to construct a generative pipeline. Such a pipeline will take some number of inputs, and transform them into a generative and emergent space of outputs. But in what context are these inputs generated, and in what context are the outputs used? This section describes several contexts in which a generative pipeline is *a component*. For any particular generative pipeline, it could be used in many different contexts, creating a different system each time, so this section is titled **Consumers of generative pipelines** to reflect that role.

Search-based techniques There are many approaches to navigating a space that has locality and continuity. By adding another dimension to the vector, a heuristic value, we can model that each vector in the space can have has a quality dimension, computed somehow from the artifact generated from that vector. A hill-climbing algorithm can take a given vector (representing some artifact), and sample the space nearby, moving from a lower quality artifact to a higher quality one, repeating the process until it has achieved a local maximum. Starting from many scattered starting spots can increase the chance of finding more maxima (not just the local one). Other approaches such as modeling inertia (particle swarm optimization) or interpolations between two points ("crossover" in genetic algorithms) can improve speed or performance in some situations.

Even in systems that are not mathematically continuous or local, these techniques can still work, just not as smoothly or well. In Flowers ([7]), the flowers have a 26-dimensional parametric "genotype", which generates L-system flowers in a mostly continuous function, but the number of leaves and petals are integers. Because of the mostly-continuous nature, a random walk throught the possibility space creates a smoothly morphing animation. But it is not fully continuous, from the integer leaves, so when the flowers animate from one form to another, there is some popping as leaves and petals appear. In systems that must be modeled as trees of choices, such as the classic genetic programming example of building an expression tree to model a function, there

[5] http://ncase.me/loopy

[6] https://deepforger.com/

[7] http://www.galaxykate.com/apps/flowers/

is no possibility to create a continuous and local space. And yet, with a large enough tree, the effect of choices at the leaves are smaller, and it begins to behave *more* like a continuous and local system. Many systems are continuous at some sections and discontinuous at others. For most practical generative systems, a designer will end up with a mix, but the more continuous and local it is, the better many forms of search (and animation) will work.

Let us consider genetic algorithms (GA) as model consumers of generative pipelines. Each GA requires three pipelines to function:

1. A pipeline that transforms a genotype to a phenotype
2. A pipeline that transforms a phenotype to an evaluation
3. A pipeline that transforms the evaluated set of genotypes to the next generation of genotypes

Each one of these may be simple or complex. The Flower app's second and third pipeline consist only of displaying the flowers to the screen, waiting for the user to select one, then creating several slightly varied versions of its DNA array for the next generation. But examining these as pipelines show that each have input and output data, and that any of the transformations about *could* be used to construct the pipeline. Likewise, a pipeline that already existed could be used by a GA, *or* could be used by other forms of search.

Constraint solvers For certain classes of well-defined problems, there exist computational solvers. Such solvers take in problem specifications in their particular format, and compute (in reasonable time) solutions to those problems. Two examples of this pattern are **IK-solvers** and **Answer-set programming**.

IK solvers are common in games for any kind of procedural animation, but perhaps reached the zenith of expressiveness in the development of Spore, where they were the final part of a complex chain of generativity, through which animator-authored animations were de-coupled from the sample creature they were animated on, to become generalized sets of "goals", which became IK-specification input for creature skeletons with arbitrarily many-jointed, twisted, and branching morphologies: "the resulting pose goals preserve the overall motion and stylistic details of the authored animation. Stylized locomotion is synthesized for the player-created leg morphology and layered onto the goals. These goals are fed into an inverse kinematics (IK) solver tuned to handle conflicting objectives while attaining natural solution poses." (Hecker et al. 2008)

A less-commonly-used solver is Answer Set Programming (ASP), a recent descendent of Prolog-style logic programming. In ASP, sets of constraints and possible options are grounded into a boolean satisfiability problem, which is then used to generate sets of final values for which all the constraints are satisfied. This method allows an author to describe a generative world as a *space of possible choices that can be made*, while also describing the rules for how different choices impact each other (for example, a tile in a Roguelike dungeon cannot contain both a wall and an object). ASP has been used in a small number of indie games (Smith 2012), as it can handle complex gameplay and design constraints, without having to code them implicitly in the generative pipeline.

Interactivity: users and players performing search As a data flow framework, it is important to consider that any part of the pipeline should be modular, if possible. Practically, this often means that a technique like search can be defined either as a computation, or as a user task. A user will probably navigate a possibility space very different from a computational optimization. Often they will dynamically switch goals, or their goals will shift from directed goals (towards a piece of content), or to meta-goals like novelty, discoverability, control and mastery. Unlike a computer, a user finds it pleasurable to **map** a possibility space, to get a model of it that fits in their mind. Talton et al recognized this in the Stanford Dryad project (Talton et al. 2009) and created socially-created landmarks which mapped commonly developed tree forms. Landmarks are useful for navigation purposes and spatial mapping, in physical spaces and abstract possibility spaces as well.

The "10,000 bowls of oatmeal"(Compton 2016) problem is common in generated content. In a smooth, multidimensional possibility space, there are many mathematically unique pieces of content, but the content is not perceptually unique to the user. So additional techniques can build structures *within* the possibility space. One example of this is in Petalz(Risi et al. 2016), where computational clustering was used to group the flower-generation space into distinct regions, each of which became collectible. More broadly, we have seen the applicability of the navigation techniques in Kevin Lynch's classic Image of the City (a book about humans navigating urban space) (Lynch 1960) become equally useful in the virtual non-physical possibility space. Content which is *characterful* is memorable as a landmark, while content that is differentiable, recognizable as either of a class, or *not* of a class creates boundaries and zones in an otherwise undifferentiated possiblity space.

Because search is a task, rewarding the player or user for performing search is also part of the pipeline of generativity. Mapping the possibility space and feeling mastery is an autotelic reward, but systems may provide additional rewards, like the collectible tracking of Petalz. Some systems also encourage, or at least afford, social rewards for search. Spore let users name and own (socially if not legally) the creations they made, and players delighted in becoming notable "creators". Often, users pull the content they created in the system into their "real" world, whether posting it to Twitter, or manufacturing it into real objects through print-on-demand services. A numinous possibility space of equally-interesting (and thus disinteresting) artifacts can become a single cherished possession with which the user or player shares a social history.

Examples of Using the Framework for Analysis and Design

With this set, it becomes possible to analyze a few example systems as generative pipelines. These systems were chosen for their simplicity and clarity (in the first examples of

Stipplegen and the works by the author) or to show that the framework holds for comparing complex games-industry PCG systems, in the case of No Mans Sky. In addition, each one has a different context (industrial design, genetic algorithm, software art toy, or game) in which the pipeline is consumed as part of a larger context.

StippleGen

StippleGen (Evil Mad Scientist Labs 2012) is an example of a generative pipeline for non-game content. However, this content still had constraints: it needed to take an image, and return a series of dots that could be drawn with a single CNC-controlled marker (on an egg), and still be recognizable as the original image.

The image is first converted into a greyscale image. The greyscale image is used to control the distribution and size of many dots (more, larger dots for dark areas). This distribution is not as aesthetic as desired for a final product 2. Create a voronoi diagram of the distributed points. Simulate easing (particle-based force simulation) to redistribute the points. Use the new point locations to guide a CNC-controlled marker to draw on an egg.

Several small works by the author

The Icemaker [8], a generative art toy, uses parametric values to set the strength of forces controlling the movement of a particle system (using steering-style forces analogous to flocking). Geometry is extruded along the length of the particles' paths, then duplicated and reflected around 6 axes to create snowflake symmetry. A random distribution sets the starting positions and thickness of the tubes.

The Flower generator [9] uses a custom parametric generator, controlling color, shape, and a Lindenmayer system to determine branching. Multiple flowers are created, and the user clicks on the best one. The selected ""favorite" flower spawns several closely-related and similar-looking children(thanks to a mostly local and continuous possibility space), and the user can iteratively evolve their flower. Users can also set it to continuously evolve autonomously, randomly walking the possibility space, as an automated procedure replaces human intention (without changing the pipeline of data flow).

Idle Hands [10] is an interactive art installation. It uses a very straightforward generative pipeline to create emergent, highly reactive animations from very little coding effort. The Leapmotion uses machine vision to detect hands, and returns a set of points representing the joints of each finger. A small cloud of particles is simulated in the background (for visual interest when the hands are still). Each frame, the screen positions of the joints and the particles (about 100 total sites) compute a Voronoi pattern. The regions of the Voronoi pattern are triangulated, and the triangles are colored based on the Perlin noise value of their centroid (this reduces flickering when triangles appear or disappear from the diagram), and the resulting triangles are drawn to the screen.

[8] https://galaxykate.itch.io/ice

[9] https://galaxykate.com/apps/flowers

[10] https://galaxykate.com/apps/idlehands

No Man's Sky

In No Man's Sky, (McKendrick 2016) ground is modeled as a thin voxel layer around a sphere. Various generative methods create terrain interest, such as using thresholded Worley and Perlin noise to add and remove voxels, which creates canyons, caves, tunnels and ""Perlin worms". Turbulence is applied, a rough physical simulation of particle movement in a forcefield. The voxels are polygonized. Various parametric objects are generated and distributed on the surface of the landscape. Like many shipped systems, this is a collage of many different layered techniques, in the words of the engineer, Innes McKendrick, "Nothing by itself is really interesting enough, the key is having this toolkit of loads different techniques that you can use in different areas until you get something that looks like the goal you're seeking."

Examining No Man's Sky as a pipeline allows us to compare it to Spore's terrain generation. Though both had similar goals and outputs, Spore used particle systems which controlled textures stamping directly into a heightmap, a completely different process than No Man Sky's noise, voxels, and triangulation.

Conclusions and Future work

This framework describes to a very fine granularity of what a generative system is and how it works. No data can magically appear: if a Markov chain requires an array of content, and all one has is an array of parametric values, then some method will be needed to convert the data before that pipeline is valid. Conversely, most methods that require compatible inputs and output (say, an array of values) can, with some flexibility, be made to work together, even if hooking them together is not a meaningful action.

Unlike many frameworks, our framework is constrained enough to make a physical construction set, with "sockets" that determine whether inputs or outputs are compatible. We recently instantiated this as a deck of cards (Compton, Melcer, and Mateas 2017)[11], with 171 cards representing various inputs, outputs, and transformative generative methods, with 17 types of data (values, voxels, vectors, text, and more) and 3 kinds of input-control (human, sensor, and content) determining compatibility. While this framework strives for general applicability, Generominos encodes specific granular algorithms. Generominos can be used to diagram existing generative systems (games and artworks alike), and, excitingly, can be used to quickly design fantastical new systems. The instantiation of this framework as Generominos will allow the framework to be readily used and tested by practitioners and investigators of generative systems across the many contributing fields and disciplines. Will this framework be flexible enough, and informative enough to be useful to practitioners and investigators of generative systems, across our many fields?

References

Ball, P., and Borley, N. R. 1999. *The self-made tapestry: pattern formation in nature*, volume 198. Oxford University Press Oxford.

[11] www.galaxykate.com/generominos

Compton, K.; Grieve, J.; Goldman, E.; Quigley, O.; Stratton, C.; Todd, E.; and Willmott, A. 2007. Creating spherical worlds. In *SIGGRAPH Sketches*, 82.

Compton, K.; Kybartas, B. A.; and Mateas, M. 2015. Tracery: An author-focused generative text tool. In *ICIDS*, 154–161.

Compton, K.; Melcer, E.; and Mateas, M. 2017. Generominos: Ideation cards for interactive generativity. *Experimental AI in Games Workshop, AIIDE*.

Compton, K.; Osborn, J. C.; and Mateas, M. 2013. Generative methods. In *The Fourth Procedural Content Generation in Games workshop, PCG*, volume 1.

Compton, K. 2016. So you want to build a generator. `http://www.galaxykate.com/buildagenerator-kcompton.pdf` (Accessed: 2017-07-01).

De la Mar, N. Bad design and the greeble. `http://adage.com/article/on-design/bad-design-greeble/138613/` (Accessed: 2017-07-01).

Dorin, A.; McCabe, J.; McCormack, J.; Monro, G.; and Whitelaw, M. 2012. A framework for understanding generative art. *Digital Creativity* 23(3-4):239–259.

Dormans, J. 2011. Simulating mechanics to study emergence in games. *Artificial Intelligence in the Game Design Process* 2(6.2):5–2.

Evil Mad Scientist Labs. 2012. Stipplegen2. `hhttp://www.evilmadscientist.com/2012/stipplegen2/` (accessed 2016-5-30).

Gielis, J. 2003. A generic geometric transformation that unifies a wide range of natural and abstract shapes. *American journal of botany* 90(3):333–338.

Harris, J. Interview: The making of dwarf fortress. `http://www.gamasutra.com/view/feature/131954/interview_the_making_of_dwarf_.php`. Accessed: 2017-07-01.

Hecker, C.; Raabe, B.; Enslow, R. W.; DeWeese, J.; Maynard, J.; and van Prooijen, K. 2008. Real-time motion retargeting to highly varied user-created morphologies. *ACM Transactions on Graphics (TOG)* 27(3):27.

Hedges, S. A. 1978. Dice music in the eighteenth century. *Music & Letters* 59(2):180–187.

Hendrikx, M.; Meijer, S.; Van Der Velden, J.; and Iosup, A. 2013. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9(1):1.

Lynch, K. 1960. *The image of the city*, volume 11. MIT press.

McCormack, J.; Dorin, A.; Innocent, T.; et al. 2004. Generative design: a paradigm for design research. *Proceedings of Futureground, Design Research Society, Melbourne*.

McKendrick, I. 2016. Building a galaxy: Procedural generation in no man's sky. NUCL.AI, `https://archives.nucl.ai/recording/building-a-galaxy-procedural-generation-in-no-mans-sky/`.

Reynolds, C. W. 1987. Flocks, herds and schools: A distributed behavioral model. *ACM SIGGRAPH computer graphics* 21(4):25–34.

Risi, S.; Lehman, J.; D'Ambrosio, D. B.; Hall, R.; and Stanley, K. O. 2016. Petalz: Search-based procedural content generation for the casual gamer. *IEEE Transactions on Computational Intelligence and AI in Games* 8(3):244–255.

Smith, A. R. 1995. Varieties of digital painting.

Smith, A. M. 2012. Mechanizing exploratory game design.

Smith, G. 2014. Understanding procedural content generation: a design-centric analysis of the role of pcg in games. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, 917–926. ACM.

Summerville, A.; Snodgrass, S.; Guzdial, M.; Holmgrd, C.; Hoover, A. K.; Isaksen, A.; Nealen, A.; and Togelius, J. 2017. Procedural content generation via machine learning (pcgml). arXiv preprint arXiv:1702.00539.

Talton, J. O.; Gibson, D.; Yang, L.; Hanrahan, P.; and Koltun, V. 2009. Exploratory modeling with collaborative design spaces. *ACM Transactions on Graphics-TOG* 28(5):167.

Wilmott, A.; Quigley, O.; and Moskowitz, D. 2012. Glassbox: A new simulation architecture. `hhttp://www.andrewwillmott.com/talks/inside-glassbox`(accessed May 26, 2017).