Visualizing Game Mechanics with Automated Clustering

Jacopo Ameli[1,2] and **David Thue**[1,3]

[1]Department of Computer Science
Reykjavik University
Menntavegur 1, 101
Reykjavik, Iceland

[2]Department of Comp. Sci. and Engineering
University of Bologna
Mura Anteo Zamboni 7, 40126
Bologna BO, Italy

[3]School of Information Technology
Carleton University
1125 Colonel By Drive
Ottawa, ON, K1S 5B6, Canada

jacopo18@ru.is, david.thue@carleton.ca

Abstract

Visualizing game mechanics is an important way for game developers to refine, review, and improve their work, and to facilitate communication between designers and programmers. This challenge has seen many different manual and automatic approaches, but they have been either too abstract and distant from the game, too close to the actual code, or lacking in connections to the code. We propose a modular system that can semi-automatically build and display a graphical representation of a given game's mechanics, to stimulate reasoning about the dependencies between the game's variables. We present an initial implementation of this visualization tool, which can read a graph from a user-made file and partition it based on user parameters. We also discuss a pilot evaluation of the tool's capacities for creative support.

Introduction

The complexity of game development is increasing, as more studios use the latest technologies and target powerful machines (Koster 2018). Part of the increase in complexity often involves a game's mechanics, which in turn affects its design and implementation. To simplify the tasks of reasoning over and iterating on a game's mechanics, we present a creativity support tool that allows visualizations of game mechanics to be generated and explored.

Good practices for general software design include the use of visualization tools during initial phases, like UML diagrams (Arlow and Neustadt 2005) and other tools for fast prototyping, extracting requirements, or determining the scope of the project. Visualizing code is important for designing, testing, and improving a codebase (Park and Jensen 2009), which are common needs in game development studios, especially in teams that often have members joining or leaving. Visualizing and abstracting game logic is also helpful for supporting discussions, discovering meta-mechanics and hidden behaviours, or prototyping (Fullerton 2014). Game developers use many different and specialized tools for visualization, and some of these (e.g., pen and paper prototyping (Mignano 2016)) are created on a per-problem basis by highly experienced developers, with a specific objective or issue in mind; such tools often lack generality, which inhibits their reuse (Machinations (Dormans 2012; 2018) is an exception). Meanwhile, visual scripting tools like Blueprints in Unreal Engine 4 (Epic Games 2012) are often limited by the types of available modules or their complexity and performance characteristics. As a result, they are more commonly used for prototyping and as a partial replacement for traditional programming. While our contribution falls into the former category, we offer the innovation that our tool uses Artificial Intelligence techniques to help designers reason about a game's mechanics.

Before we can visualize game mechanics, we first need a representation that will work with any game. Thue and Bulitko (2018) used deterministic, Factored-state Markov Decision Processes to represent a game's internal logic. A simplified version of this representation is a directed graph, where the nodes are the game's variables and the edges are the direct dependencies between them that arise from the game's mechanics (Figure 1). We tested this representation with pen and paper prototypes during an initial participatory design session. We hypothesize that by visualizing such a graph of game mechanics and highlighting potentially interesting areas within it, we can help designers reason about possible improvements to their game. We use graph clustering to identify different areas of a game mechanics graph.

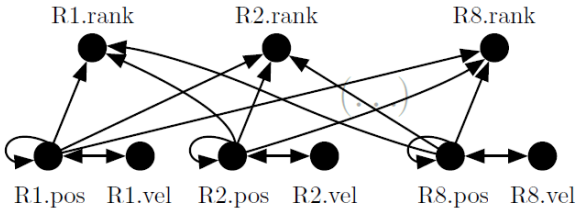

Figure 1: An abstract view of game logic by Thue and Bulitko (2018), used with permission. Vertices are game state variables from *Mario Kart* (Nintendo 2008). Edges show abstract game mechanics as dependencies between variables.

Concepts and Terminology. Our visualization of game logic derives from Thue and Bulitko's (2018) representation of game mechanics as deterministic, Factored-state, Markov Decision Process (FMDP). A non-factored, deterministic MDP is defined by a set of possible states S, a set of possible actions A, and a transition function $\tau : S \times A \rightarrow S$ to transition between states, where in this case $\tau(s, a) \rightarrow s'$ means that the state s' will deterministically occur when an agent performs an action a in state s (Bellman 1957). In the factored-state version (Chakraborty and Stone 2011), each single state s is defined by a vector of state factors. For the purposes of this work, the agent is a player of a game, the state factors are variables whose values comprise the game's state, and the transition function represents the game's mechanics and thus the dependencies that exist between its variables. A graphical model of a game's mechanics, as in Figure 1, can be represented as a directed graph where each vertex represents a game variable and each edge represents a direct causal dependency between two variables. For terms related to graph theory, we use definitions based on work by Schaeffer (2007) and Menegola (2012). Given a *graph* $G = (V, E)$ where V is a set of vertices and $E \subseteq V \times V$ is a set of edges, an *edge* (v, w) defines a connection between a pair of vertices $v, w \in V$. A *path* from one vertex to another is a sequence of edges between the two vertices, and if a graph has paths between every pair of vertices in V, then that graph is said to be *connected*. For a given edge $(v, w) \in E$, w is also called a *neighbour* of v, and the set of all neighbours of a vertex is the *neighbourhood* of that vertex. Given that $m = |E|$ is the number of edges in G and $n = |V|$ is the number of vertices in G, the standard *density* of a full graph G is given by:

$$\delta(G) = \frac{m}{\binom{n}{2}}, \tag{1}$$

where $\binom{n}{2}$ is the total number of possible edges between the vertices. In a *weighted* graph, a *weight function* $\omega(v, w) \rightarrow [0, 1] \in \mathbb{R}$ assigns a real-valued *weight* to each edge.

Part of our approach involves separating a graph representation of a game's mechanics into a number of parts. This is known as a *k-way partitioning problem*, which we introduce here. Consider a positive integer k and a weighted graph $G = (V, E)$. Intuitively, part of the goal of the k-way partitioning problem is to divide G into k *parts* $\{P_0, P_1, ...P_{k-1}\}$, each of which is a subset of V, such that none of the parts overlap and all of the parts can be combined to recover the full set of vertices. The set of k parts is called a *partition*, P, of G. While each part $P_i \in P$ refers only to vertices in G, it can be useful to consider two kinds of edges in G: those that connect vertices within any single part (*internal* edges), and those that connect vertices from two different parts (*external* edges). An *induced subgraph* of a graph $G = (V, E)$ is a graph (P_i, E_{P_i}) where $P_i \subseteq V$ and $E_{P_i} \subseteq E$, such that E_{P_i} includes only the edges from E that have vertices in P_i: $\forall (v, w) \in E$, $v, w \in P_i \implies (v, w) \in E_{P_i}$. The induced subgraph of a part P_i contains only the edges in E that connect P_i's vertices. Given two parts P_i and P_j, each edge in E that connects two vertices across P_i and P_j is called a *cut* edge. Given a partition P of graph G, consider the set C of all cut edges across all pairs of parts P_i and P_j (where $i \neq j$) in P: $C = \{(v, w) \in E | v \in P_i, w \in P_j, i \neq j\}$. The full goal of the k-way partitioning problem is to find a partition P such that the *size* of C is minimized and P's parts have roughly equal numbers of vertices. When G is a weighted graph, the *size* of C is the total weight of the edges in C; if G is not weighted, the size of C is given by its cardinality. Given a partition P of a graph G, we say that a *cluster* is a subgraph that is induced by any part P_i in P. Each cluster should have certain properties: (i) it should be connected – there should be a path between every pair of cluster vertices that traverses only the cluster's internal edges; (ii) it should have more internal edges than external edges; (iii) its density should be higher than the density of the full graph (Equation 1). These properties would identify subgraphs that have highly interconnected vertices, possibly representing a subsystem within a system. We assume that graphs derived from implemented game systems are highly clusterable; this assumption is driven by the nature of human labor division, as well as the most widely used patterns for system design, like the top-down approach (Schank and Abelson 1977).

Problem Formulation

Given a graph where the vertices are variables of a game's state and the edges are the dependencies between them, we aim to highlight areas within that graph that might be *interesting* for the designer to consider. By "interesting areas", we mean those that might *not* match the game's actual subsystems or the designer's grouping of the mechanics, but could instead spark new considerations and reasoning about the game. We decided this after an initial participatory design session, where it became apparent that simply matching the code structure perfectly would yield no new information, and matching a designer's particular grouping would be highly unlikely. Ideally, designers should be encouraged to think causally about their game's mechanics and improve them through iteration. Since our solution's success depends on whether or not it supports the creative process of designers, we will report its Creativity Support Index (Carroll and Latulipe 2009), as collected from a group of game designers.

Related Work

Some game design tools offer assistance from Artificial Intelligence systems for generating game content, while also helping with parameter tuning and balancing (Perez-Liebana et al. 2018). To support analysis, other tools can automatically generate heat maps of movements, trajectories, and other data in multi-player or single-player levels while also suggesting improvements (Andrade et al. 2006). Such tools are usually employed to visualize data gathered during tests or live play, and can help discover meta-mechanics. They are usually introduced to visualize the effects of the design on the game's flow and enjoyment, and they do not aim to visualize the game's mechanics themselves. The game engine *Unreal Engine 4* (Epic Games Inc. 2012) offers a visual scripting tool called Blueprints (Epic Games 2012) that allows users to instantiate objects within the game world, in-

voke individual functions or general events, implement level triggers, AI behaviours, and more. Visual scripting tools such as Blueprints serve a different purpose than what we aim to achieve; they support implementing a game's design, while our work seeks to support general, creative reasoning about a game's design. Machinations (Dormans 2012; 2018) is a tool for visualizing and testing game mechanics by abstraction. It requires the user to manually build an interactive model of the logic of the game so that its gameplay can then be abstractly simulated in a graph-like user interface. It provides an approximation of board game prototyping and testing, and it is completely independent from a game's implementation. As with other independent tools and methods, it loses relevance as soon as the implementation changes significantly; manual synchronization is required. While our solution also requires some initial manual input, the creation of the visualization is semi-automatic; new graphs and partitions are computed automatically in response to user inputs.

Proposed Approach

We adopted the simplified version of Thue and Bulitko's representation of game mechanics as edges that connect individual variables (vertices) in a directed graph (Figure 1). We will refer to this representation as a *game graph*. In that representation each directed edge from variable v to w indicates that v's value is used (by the game's mechanics) to compute a new value for w at one or more times during gameplay. Each edge thus represents the direct, causal dependence of the target variable on the source variable, and tracing a variable's causal ancestors or descendants can reveal (indirect) dependencies between potentially distant variables in the graph. We assume that including *all* of a game's variables in a game graph would impede the graph's usefulness and readability for designers. To support representing dependencies between only *some* of a game's variables, we extended Thue and Bulitko's representation to include a weight on each edge of the graph. This weight represents the degree to which the edge's target variable depends on its source variable. A weight of 10 represents direct dependence and close definitions in the script (e.g., w's value is computed from v's in a single line of code and they are defined in the same component), while 1 represents highly indirect dependence and distant definitions. This extension allows edges in the game graph to represent indirect dependencies between vertices, which helps the graph remain connected when less-important variables are omitted.

Finding Clusters. Our approach is driven by the intuition that there exists a structure underlying a given game's graph whose analysis might lead to new insights regarding the game's design. Specifically, we suspect that different clusters of a game's graph can be conceptualized as *systems* (or subsystems) within the game's mechanics, and that presenting designers with different candidates for such systems might serve as a creative aid. Our solution uses automated clustering to generate different candidates for systems that might exist within the game's mechanics. Specifically, to solve the k-way partitioning problem, we: (i) apply a Multi-Level Kernighan-Lin algorithm to find a tentative partition (Hendrickson and Leland 1995; Kernighan and Lin 1970), (ii) create a second partition by separating the found parts into sets of vertices that belong to connected subgraphs (clusters; see Algorithm 1), and (iii), if the two partitions are not the same, merge parts in the second partition until a total of k parts are obtained (Algorithm 2), and return the result. If the clusters induced from the parts were not required to each be (internally) connected, they could include variables that were completely unrelated to one another. If the parts were unbalanced, some clusters could be too small or too big to be relevant or readable. This balance is guaranteed, to a degree, by the partitioning algorithm and the ordering of parts to be merged in Algorithm 2.

Algorithm 1: Finds a partition with connected parts.

```
Data: Graph G, Partition P
Result: A partition with connected parts, P'
X ← ∅   // initialize set of explored vertices
P' ← ∅  // initialize output partition
foreach part P_i ∈ P do
    foreach vertex v ∈ P_i do
        initialize empty queue of vertices Q
        P'_i ← ∅                    // initialize new part
        if v ∉ X then
            X ← X ∪ {v}    // mark as explored
            P'_i ← P'_i ∪ {v}  // add to new part
            N ← {w ∈ P_i : w ∈ Neighborhood(v)}
            foreach w ∈ N do
                enqueue w in Q
            end
        else
            continue
        end
        while Q is not empty do
            q ← dequeue from Q
            X ← X ∪ {q}
            P'_i ← P'_i ∪ {q}
            N ← {w ∈ P_i : w ∈
              Neighborhood(q), w ∉ X}
            foreach w ∈ N do
                enqueue w in Q
            end
        end
        P' ← P' ∪ {P'_i}
    end
end
return P'
```

Estimates of Quality. To support generating a variety of different partitions, our solution estimates a measure of *quality* for each partition. Since each cluster should have higher density than the full graph and more internal edges than external edges, we considered using external density (Schaeffer 2007) as a possible measure of quality, but the equations that we found in the literature lacked any parameters that could be used to prioritize different types of re-

Algorithm 2: Merges partition parts to up to k parts.

Data: Partition P, desired number of parts k, Graph G
Result: A partition with k parts, P.
while $|P| > k$ **do**
  sort P's parts from smallest to biggest
  found $\leftarrow$ False
  foreach part $P_i \in P$ **do**
    foreach part $P_j \in P : P_j \neq P_i$ **do**
      if P_i is connected to P_j in G **then**
        $P \leftarrow P \backslash \{P_i, P_j\}$ // remove parts
        $P_{\text{temp}} \leftarrow P_i \cup P_j$ // merge parts
        $P \leftarrow P \cup \{P_{\text{temp}}\}$ // add merged part
        found $\leftarrow$ True
        break
      end
    end
    if *found* **then**
      break
    end
  end
end
return P

sults. To address this problem, we developed an alternative estimator of external density that contains such a parameter. The standard estimator of external density (Equation 2) considers edge cuts only in terms of their global cardinality:

$$\delta_{ext}(G|P_0..P_{k-1}) = \frac{|(v,w) \mid v \in P_i, w \in P_j, i \neq j|}{n^2 - n - \sum_{l=0}^{k-1} |P_l|^2 - |P_l|} \quad (2)$$

Instead, our estimator (Equations 3 and 4) considers edge cuts in terms of cardinalities that are local to each cluster, and uses the parameter x to define a family of external density estimators:

$$\delta_{ext}(x, G|P_0...P_{k-1}) = \frac{\sum_{a=0}^{k-2} \sum_{b=a+1}^{k-1} (\delta_{par}(P_a, P_b))^x}{\binom{k}{2}}, \quad (3)$$

where:

$$\delta_{par}(P_a, P_b) = \frac{|(v,w) \mid v \in P_a, w \in P_b|}{|P_a||P_b|}. \quad (4)$$

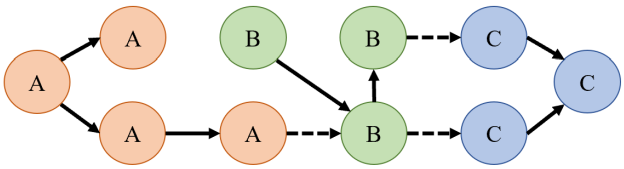

Figure 2: A toy graph for demonstrating different quality estimators. The letters and colours represent vertices that belong to different clusters, the solid arrows represent internal edges, and the dashed arrows represent external (cut) edges.

To understand the difference between the standard estimator and ours, consider the the graph shown in Figure 2. Using Equation 2, the standard external density δ_{ext} would be $\frac{1}{18}$. This value gives a reasonable indication of the graph's overall density, as it takes into consideration the edge cuts, the total number of possible edges, and the internal edges of the clusters. Calculating the external density with our estimator, however, can give us more information. First, we use Equation 4 to calculate the partial external density between each pair of clusters. For clusters A and B, Equation 4 yields $\frac{1}{9}$, while the partial external density is $\frac{2}{9}$ between B and C and 0 between A and C (there is no cut edge between them). Once the partial densities have been calculated, we already have information on which clusters might affect the overall external density, and by how much. Finally, we compute the overall external density using Equation 3 which for $x = 2$ would equal 0.02, for $x = 1$ would equal $0.\overline{1}$, and for $x = \frac{1}{2}$ would equal 0.27. As a result, different values of x favour different kinds of partitions – particularly, with different amounts or sizes of clusters. As an example, Figure 3 shows the same graph partitioned using $x = 2$ and $x = 1$.

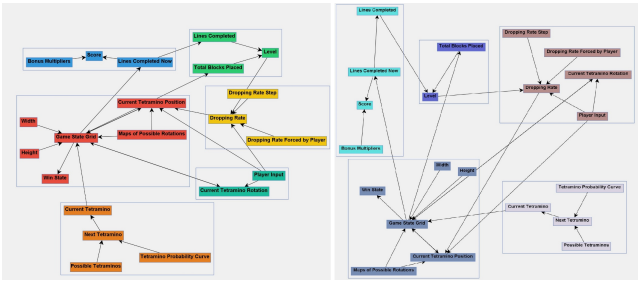

Figure 3: Graph partitioned with $x = 2$ (L) and $x = 1$ (R).

In addition to allowing our solution to generate a variety of different clusters, our quality estimator offers a way to compute that is more modular than the standard approach, and this offers an opportunity to save some computation. Specifically, when any change to the graph leaves some of the partition's parts unchanged, only some of the partial density values need to be recomputed.

Search. In our context, the choice of k in the k-way partitioning problem can be somewhat arbitrary – some designers might not know (nor care to specify) how many clusters the algorithm should find. To support this scenario, our solution can search automatically through a space of differently-sized partitions and return one based on its estimated quality.

Architecture and Implementation

Figure 4 shows the system architecture of our prototype, all of which has been implemented except for the code analyzer. The tool's systems include a *loader* for reading game variables and their dependencies as inputs into to the system, a *graph database* for storing the loaded data, a *partitioner*, *estimator*, and *searcher* for automatically finding clusters within a given graph, a *user interface* to facilitate users exploring and modifying the visualized graphs and their clus-

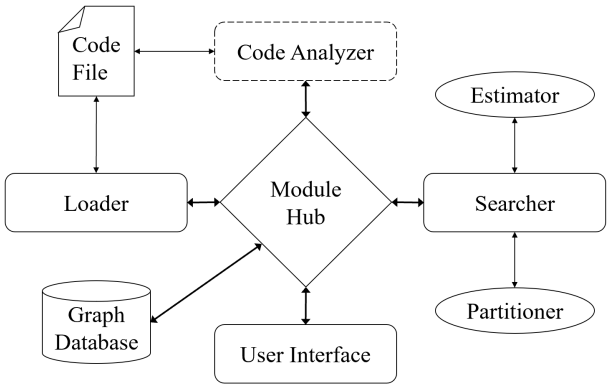

Figure 4: An overview of our solution's architecture. Rounded rectangles show modules that interact directly with the Module Hub. The dashed line shows a module with no current implementation. The cylinder shows storage. Ellipses show sub-modules used by the searcher.

ters, and a *module hub* to handle communication and primary control flow across the other modules. During typical execution, the loader keeps track of changes in the game's code by reading the latest data from a given code file. The module hub uses data from the loader to update the internal graph representation in the database, and invokes the searcher to find a way to cluster the graph. Finally, the system shows the partitioned graph through the user interface (UI; Figure 5) and handles communication between the UI and the other modules.

The tool is meant to be used mainly during development, once a playable prototype can be reasoned over and refined. Some modules contain implementations of the algorithms and equations we introduced in the Proposed Approach section. The partitioner finds clusters (connected induced subgraphs) within a full graph, and it implements a Multi-Level Kernighan-Lin algorithm that is originally from a neural net simulator (Kernighan and Lin 1970; Nengo 2018). We added a method to separate the initial tentative parts into parts belonging to connected subgraphs or clusters (Algorithm 1). If those parts are not the same as the initial parts, it tries to get a connected result (Algorithm 2). The estimator calculates the overall quality of a partition using our adapted estimate of external density (Equations 3 and 4). Our implementation allows the user to choose between four different estimators, with $x = 2, 1, \frac{1}{2}$ and $\frac{1}{3}$, respectively. As shown in Figure 3, using different values of x during the search process can result in different numbers of clusters being found. We now present the remaining modules.

Code Analyzer. The code analyzer is meant to analyze the game's codebase and automatically construct a graph representation. This means that it should be able to identify and track the graph's vertices (game variables) and connect them according to their direct dependencies. Furthermore, it should also estimate how closely the two variables are defined as a script closeness weight to the edge. For example, `xPos = xPos + xSpeed;` would be abstracted as a vertex `xPos` with a self-dependency, a vertex `xSpeed`, and an edge going from the latter to the former. The edge weight would be high, as it is a direct dependency. The implementation of the code analyzer remains as future work – creating it will take substantial work, and we wished to first perform an initial assessment of the other parts of the tool. Nonetheless, we confirmed the possibility of creating the analyzer via early discussions with researchers of causality in Computer Science. The code graph files are currently created by the user via their knowledge of the game's code.

Loader. The loader converts a given code file into an internal representation that is used by the rest of the tool. Its current implementation can load data from DOT (.dot) files and tab-separated value (.tsv) files. DOT is an open file format that facilitates describing graphs. We supported .tsv files because they allowed for easier testing and evaluation, as they can be created easily from spreadsheets. The first column and the first row of the sheet contain the variable/vertex names. A non-zero entry in a cell indicates an edge from the row variable to the column variable, and the value in the cell is the edge weight. Empty and zero-valued cells represent non-edges in the graph.

Database. The database stores the internal representation of the graph. In the current implementation, it also stores a *script closeness factor* and computes changes to the graph based on it. The factor specifies how closely the edge weights in the graph should follow the analyzer's estimated script closeness. On a scale of 1 to 10, a script closeness factor of 1 would bring all current edge weights down to a minimum of 1, while a factor of 10 would bring them up to their original values, as given by the following equation:

$$\omega(A, B) = 1 + \tfrac{1}{9}(\textit{scriptFactor} - 1) * (\textit{originalWeight} - 1).$$

Searcher. The searcher module finds the best quality graph partition by exploring a space of possible partitions using a partitioner and an estimator, but it is independent from the partitioner's algorithm and the estimator's quality calculation. In the prototype, the searcher explores different numbers of partitions (k), linearly, increasing k by 1 iteratively and checking the quality of the current test partition using a user-selected estimator; iteration stops when the quality starts decreasing.

User Interface. The UI provides the user with a readable graph and tools to modify a partition to their liking (Figure 5). For the user interface of the prototype, we used the mxGraph library (Jgraph Ltd. 2018), which includes many tools and data types to easily implement a graph visualizer/editor. The user can change different parameters for partitioning, including: (i) the script closeness factor, or how close the graph edge weights should be to the analyzer's (or the user's) estimate of the dependence between two variables; (ii) which estimator will be used to calculate the quality of the graph – this changes the exponent x in the modified external density equation; and (iii) how many parts should be in a partition – by default it is automatic, but it can also be manually specified. The users can also specify a timeout for

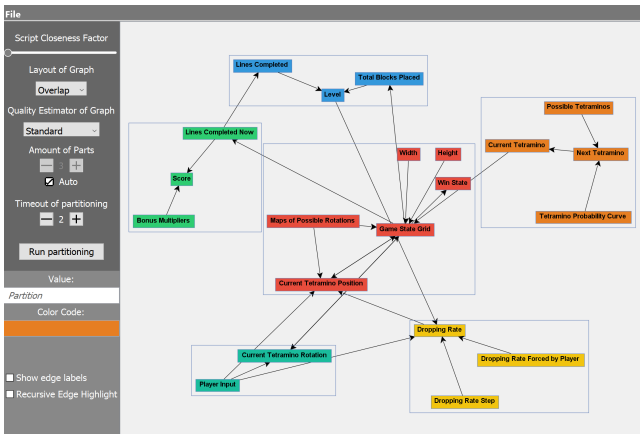

Figure 5: An initial view of our software's user interface. A sample game graph for the game *Tetris* is shown (Bullet Proof Software, Nintendo 1989).

the algorithm, choose which automatic layout to use, change the value and colour of single vertices or their parent group, show any edge weight, and recursively highlight in red the incoming edges for the selected vertex.

Evaluation

To better understand our tool's capacity to support creative reasoning, we conducted an evaluation with 36 undergraduate students who were presently enrolled in a course on designing and developing games. They had already been divided into 12 teams of 3 to 4 students, and each team already had a working prototype of a game that they were developing for the course. The games were very diverse, including competitive multiplayer platformers and point-and-click adventure games. We began our study by introducing the tool and giving tutorials on how to generate the required input file and how to understand and use the tool. After filling a spreadsheet with variables from their game (∼30 minutes) and the dependencies between them (∼45 minutes), each team then downloaded it as a tab-separated value file and loaded it into the tool. The tool then immediately visualized the game graph via the tool's user interface. We provided each team with some example questions for the teams to ask themselves (e.g., "Why are these variables grouped together?" or "Can we add other parameters to influence this mechanic?"), as prompts for using the tool for the remaining time. At the end of this 2-hour process, we asked each participant to complete a survey instrument for individually collecting the Creativity Support Index (CSI) (Carroll and Latulipe 2009). The CSI is designed to evaluate the "amount of helpfulness" of a creative tool, and it is determined through a survey. The result of this instrument is a value from 1 to 100, where 1 is the worst score and 100 the best, which can be averaged with the results of multiple participants to give an indication of the usefulness of the software as a creative tool. The final CSI Score average that our tool obtained was 53.07, with a standard deviation of 12.07. Under an assumption of normality, this gives us a 95% confidence that the tool's true average score is between 28.93 and 77.21.

Discussion, Limitations, and Future Work

While some teams found it helpful - for instance, one team identified a new lighting subsystem to centralize - the evaluation shows promising but inconclusive data about the usefulness of our prototype as a creative tool; the high variance makes it unclear where the true average CSI score would lie. The high variation among the types of games that were analyzed might contribute to the variance, as might the general design competence of our participants, who were largely inexperienced designers. We also did not control for any benefits coming from the initial spreadsheet-filling process, but we suspect that any new ideas that arose while using the tool were likely to have been influenced by using it.

Several changes could be made to the tool to improve its accessibility and usefulness. On a representational level, the game graph loses some information regarding execution order, conditions, and values. We hope that this simplification makes the tool more intuitive, but this should be investigated further to find whether a better trade-off exists. The current manual input method for the game graph is cumbersome, and it hinges on the user being skillful at interpreting the game's code. Implementing a code analyzer might help in this regard, although correctly identifying which variables should or should not be included in a graph might be a difficult problem. The analyzer could collect runtime information, like frequency of access to a variable and probability of choosing a specific branch of code, in order to support higher-level considerations. It might also use data types, annotations and save files to better filter the variables.

The current prototype uses a widely implemented partitioning algorithm with additional checks to ensure that the results are connected clusters, but this solution is computationally expensive. It would be interesting to compare the performance of different potential algorithms. A subset of the algorithms could also guarantee connected results with low external density, to further speed up the process.

Conclusion

This paper proposed a semi-automated approach to generating a visual representation of game mechanics with a game graph and automatically finding clusters therein. We also proposed a new alternative measure of external edge density for partitions, which offers more flexibility than more standard measures. We created a prototype capable of partitioning a given, weighted game graph into parts according to different quality parameters, and then displaying the result in an interactive application. We tested the prototype and evaluated it using the Creativity Support Index survey. The results were promising but inconclusive, and the high variation in game types and design experience of the participants may have contributed to the high variance in the results. While a code analyzer would be useful for a future implementation, the tool already provides a new, more general approach to visualizing game mechanics, and it may yet help designers with reasoning creatively about their games.

Acknowledgements. We thank Tigran Tonoyan for his help with the custom external density formulas, and the Erasmus+ funding programme for their financial support.

References

Andrade, G.; Ramalho, G.; Gomes, A. S.; and Corruble, V. 2006. Dynamic game balancing: an evaluation of user satisfaction. In *Proceedings of the 2nd AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2006)*, 3–8. Marina del Rey, California: AAAI Press.

Arlow, J., and Neustadt, I. 2005. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*. Pearson Education. Google-Books-ID: Fme5TXzP0VgC.

Bellman, R. 1957. A markovian decision process. *Indiana University Mathematics Journal* 6:679–684.

Carroll, E. A., and Latulipe, C. 2009. The Creativity Support Index. In *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '09, 4009–4014. New York, NY, USA: ACM.

Chakraborty, D., and Stone, P. 2011. Structure learning in ergodic factored MDPs without knowledge of the transition function's in-degree. In Getoor, L., and Scheffer, T., eds., *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 737–744. New York, NY: ACM.

Dormans, J. 2012. *Engineering emergence: applied theory for game design*. Ph.D. Dissertation, University of Amsterdam.

Dormans, J. 2018. machinations.io - the game design tool.

Epic Games Inc. 2012. *Unreal Engine 4*. Game Engine.

Epic Games. 2012. Blueprints Visual Scripting.

Fullerton, T. 2014. *Game design workshop: a playcentric approach to creating innovative games*. AK Peters/CRC Press.

Hendrickson, B., and Leland, R. 1995. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95. New York, NY, USA: ACM.

Jgraph Ltd. 2018. mxGraph Library. original-date: 2012-05-21T20:19:37Z.

Kernighan, B. W., and Lin, S. 1970. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal* 49(2):291–307.

Koster, R. 2018. The cost of games. *VentureBeat. https://venturebeat.com/2018/01/23/the-cost-of-games/*.

Bullet Proof Software, Nintendo. 1989. Tetris. Game [Gameboy]. Nintendo.

Nintendo. 2008. Mario Kart Wii. www.mariokart.com/wii/.

Menegola, B. 2012. A Study of the k-way Graph Partitioning Problem. Master's thesis, Federal University of Rio Grande do Sul.

Mignano, M. 2016. Use Paper Prototyping to design your games. *GamaSutra, www.gamasutra.com/blogs/MarcoMignano/20160725/277766/Use_Paper_Prototyping_to_design_your_games.php*.

Nengo. 2018. Nengo Neural Net Simulator. https://github.com/nengo/nengo-1.4.

Park, Y., and Jensen, C. 2009. Beyond pretty pictures: Examining the benefits of code visualization for Open Source newcomers. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 3–10.

Perez-Liebana, D.; Liu, J.; Khalifa, A.; Gaina, R. D.; Togelius, J.; and Lucas, S. M. 2018. General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms. *arXiv:1802.10363 [cs]*. arXiv: 1802.10363.

Schaeffer, S. E. 2007. Graph clustering. *Computer Science Review* 1(1):27–64.

Schank, R. C., and Abelson, R. P. 1977. *Scripts, Plans, Goals, and Understanding: An Inquiry Into Human Knowledge Structures*. Lawrence Erlbaum Associates. Google-Books-ID: YZ99AAAAMAAJ.

Thue, D., and Bulitko, V. 2018. Toward a Unified Understanding of Experience Management. In *Proceedings of the 14th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'18)*, 130–136. AAAI Press.