Settlers of Catan: Developing an Implementation of an Emerging Classic Board Game in Java

Vladimir Costescu
Professor David P. Walker
Princeton University
August 19th, 2014

"This paper represents my own work in accordance with University regulations" - Vladimir Costescu

Contents

Abstract

The goal of this project was to study the workings of the Settlers of Catan® board game, seemingly simple but in reality quite complex, and faithfully implement a computer program in the Java™ programming language through which the game can be played. Furthermore, in the process of creating this implementation, I intended to learn the basics of producing a complex piece of software on my own and from scratch, with planning and design constituting major challenges in this endeavor. In this paper, I will discuss the background of the game and related work, the rules of the game, the steps and reasoning behind my implementation, the challenges I encountered in the course of development, and features of the game I would like to implement in the future.

1. Motivation and Goals for the Project

As with all human undertakings, large and time-consuming projects begin with an underlying motivation, a drive that propels those who engage in such endeavors to start a project and also to continue working on it, despite inevitable hardship during its course, until some goals are met and/or a calendar deadline comes to pass. This senior thesis project is no different. Two years ago, I was first exposed to Settlers of Catan, a three to four person board game, when a number of my fellow eating club members invited me to join them in playing a strange and complicated-looking board game. Having nothing better to do at the time, I accepted their offer and started learning how to play the game. Not surprisingly, it took me some time to begin to grasp the complex rules of the game, and I lost quite a few games, coming in dead last consecutively and consistently. I even became angry at my inability to win, and my friends, tauntingly, told me to "please continue to rage" as they laughed amongst themselves. With some practice, I was able to win on sporadic occasions, and I learned to not take the game so seriously and enjoy the social interactions facilitated by Settlers of Catan.

Although those responsible for piquing interest in the game have since graduated, the knowledge that I gained during those spring days remained dormant in my mind for some time, until my memories of the game resurfaced this past fall and I decided to invest in a specimen of the game in order to play with my suitemates. During an ordinary gameplay session, an idea struck me: as a Computer Science major and Settlers of Catan aficionado, why not implement the game as a computer program for my senior thesis? I mulled over the idea briefly and decided that it would make for an interesting research opportunity, and thus I chose it as my topic of investigation for this

paper. What follows is a report of the multifaceted learning experience in which I engaged in my study of Settlers of Catan, including the sum of small successes encountered throughout and also the many moments of frustration that inevitably crept up when things did not go according to plan.

2. Background and Related Work

2.1. Klaus Teuber and Die Siedler von Catan

The story behind the genesis of the game is a curious and unexpected one, beginning with a frustrated dental technician who tinkered with board game ideas in his spare time in his basement workshop and culminating in a wildly successful commercial enterprise that has taken the world by storm and even prompted observers to posit that Settlers of Catan has the potential to displace the classic Parker Brothers game Monopoly in terms of household popularity [10]. While such predictions have not come to pass, the profits generated by Catan and its numerous offshoots and adaptations have certainly enabled Klaus Teuber, the game's creator, to abandon his career as a dental technician and focus on board game development as a full-time job under the banner of his company, Catan GmbH.

Born in 1952 to a father specializing in the dental technician profession, Teuber studied chemistry as an undergratuate and then pursued the same career track as his father, becoming an apprentice in his father's dental lab in 1978 [11]. Just a few years later, in 1981, Teuber created his first game, a predecessor to Barbarossa, and embarked upon what would become a fruitful journey into the world of board games [11]. Throughout his dental career, Teuber continued experimenting with game design, fatefully releasing Die Siedler von Catan in Germany in 1995 and then the translated version, Settlers of Catan, in the United States in 1996 [10]. Experiencing unexpected success with Catan, Teuber was able to afford quitting his job as dental lab managing director in 1999 and subsequently established his company, Catan GmbH, together with his son Guido in 2002 [11].

2.2. Impact of Catan

At the turn of the 21st century, Catan was still a niche product despite enjoying solid success in its home country of Germany after Teuber won the 1995 Spiel des Jahres, but by 2011 it had sold over 18 million copies worldwide [10, 14]. As Catan expanded beyond Germany, consumer demand grew so quickly that the resource needs for manufacturing the game exceeded available supply, and as a result Mayfair Games, Catan's publisher, had to reach out to American companies in order to acquire the necessary materials [10]. Now available for purchase at major retailers such as Amazon.com, Barnes & Noble, Target, and Walmart, Catan is considered mainstream enough to constitute serious competition for more entrenched titles such as Monopoly or Risk [4, 6, 10, 14].

Not only does Catan compete with Monopoly for marketshare, it also competes for mindshare, offering an alternative to the strictly adversarial, capitalistic, and zero-sum world depicted in Monopoly [4, 14]. In Monopoly, there is little opportunity for players to cooperate in order to improve each other's fortunes, nor is there any incentive to do so. Instead, the goal of Monopoly is to drive other players to financial ruin, and the game does not end until one player succeeds in driving the others into bankruptcy, an outcome that can take hours to achieve and likely induces boredom in the players who know that they will lose [4]. Furthermore, although Monopoly's vicious and cutthroat nature unfortunately remains relevant today, more than 75 years after its creation, its teachings also serve to perpetuate the harmful behaviors that have led the world into catastrophic financial crisis, much in the same way as the solitary winner in the game leads all other participants to their economic downfall [4]. In such a toxic climate, Catan attempts to change the status quo through its simple and innovative gameplay.

In contrast to Monopoly and Risk, Settlers of Catan encourages a cooperative rather than confrontational play style, and in fact it is quite difficult, if not impossible, to win the game without engaging with other players in trade for resources [4, 14]. Due to the nature of the game, victory is seldom assured for any one player, and it is not uncommon for players in last place to make a miraculous comeback and win the game. This property of the game keeps players engaged and makes strategic planning more difficult, since each player's actions can provide a material advantage

to another player and change the tide of the game [4]. Furthermore, the interconnectedness of players' fortunes serves to discourage any excessively hostile actions by any one player, since the other players might then conspire to frustrate that player's progress in the game and even strip him or her of any chance of achieving victory. Finally, the conditions that align to create this deterrence mimic the real world quite well, making Catan a useful learning tool and a simplified model for how things actually work in practice, particularly in the domain of international relations [4].

2.3. Other Software Implementations of Catan

Settlers of Catan has been implemented as a software program on a number of platforms, including PC / Mac, iPad / iPhone, Android, Xbox 360, and even Amazon Kindle [12]. In addition to these official versions, there also exist numerous open source incarnations of the game, including SolitaireSettlers, Pioneers, Settlers3d, JSettlers2, and HTML5Settlers [1, 3, 5, 7, 9]. Given the nature of the project at hand, I did not peruse the source code of any of these projects, since I did not want to taint my implementation with ideas taken from sources other than the Settlers of Catan physical board game that I purchased from Amazon.com and the rule booklet included within. It is worth noting that the Settlers of Catan legal team may have, at some point in the past, been involved in the removal of some non-official implementations of the game from the Web, meaning that the list of implementations provided here may not be complete [8].

3. Catan Rules [13]

3.1. Game Components

Before a description of the Settlers of Catan game rules can be attempted, it is necessary to first enumerate and explain the components of the game. The most basic element of the game is the board, which is composed of nineteen hexagonal tiles surrounded by six sea frame pieces. Eighteen of the tiles depict one of five attainable resources in the game, Wool, Grain, Lumber, Brick, and Ore, while the remaining tile depicts a resource-void desert. The sea frame pieces each have one or two sea ports, which are essential to maritime trade, an important game activity, adding up to a total

of nine ports, four of which are of a generic type and the rest of which are of a particular resource type, granting a better trade ratio due to their specificity. Augmenting the board at various stages in the game are roads, settlements, and cities, which can be placed by players during their turn. Enabling the placement of these game pieces are ninety-five resource cards, nineteen of each of the five types, which are collected by players during the normal course of gameplay. The acquisition of resource cards by players is controlled by eighteen dice roll chits, which are distributed across the eighteen resource tiles on the board, and by a pair of standard six-sided dice that are rolled during a normal turn of gameplay. Resource distribution can be curtailed by the placement of the robber piece, which initially starts out on the desert tile but can be moved when certain events occur during gameplay. In addition to being used to purchase roads, settlements, and cities, resource cards may also be used to purchase one of twenty-five development cards, which are shuffled in a deck and placed face down at the beginning of the game. Each development card is imbued with a special property that can affect the course of gameplay, and a small subset can even contribute to a player's victory. Finally, two special cards, Longest Road and Largest Army, can be granted to the player who meets certain special criteria, as alluded to by the titles of the cards.

3.2. Game Setup

The initial setup stage of the game proceeds according to a well-defined workflow. The first step is putting the board pieces together as described in significant detail in the game manual, whether following the prescribed static layout for beginners or using some degree of randomization. In the most randomized setup procedure that is mentioned in the rulebook, the hexagonal tiles, dice roll chits, and port types are shuffled before being used to construct the board. The hexagonal tiles are placed without further restriction after being shuffled, although in the physical version of the game players may opt to avoid excessive resource clustering. The dice roll chits are almost fully random, with the only stipulation being that the numbers six and eight may not be on adjacent hexes, due to their high chances of rolling (14%) relative to the other possible numbers. Finally, the available port locations do not change in this setup, but the port types are distributed randomly across all the possible locations.

The next step is determining the order in which players will take their turns, and there are a few potential mechanisms for doing so. The simplest is for the oldest player to go first and then the remaining players to follow in descending order by age. Another option is for all players to roll the dice, where the player with the highest roll goes first and the remaining players follow in descending order by dice roll, and any dice roll ties are resolved by one or more rerolls. A third option is to alter either the age or dice roll mechanism such that the second, third, and potentially fourth players are ordered according to their seating location, either clockwise or counterclockwise. No matter which mechanism is chosen, the generated turn order remains the same throughout the game.

In the final step in the game setup process, each player is given the opportunity to place two settlements and two roads on the board. First, players take turns placing one settlement/road pair, such that settlements are placed on hex tile vertices, which are called "intersections", and roads are placed upon the edges connecting these vertices, with the stipulation that roads placed on the board must be connected to a settlement owned by the same player. Once each player has placed one settlement/road pair, the turn order reverses and the last player is the first to place the second pair, with the restrictions that the second settlement must not be an immediate neighbor of the first and that the second road must be attached to the second settlement. After each player has placed two settlements and roads, everyone collects one resource card of the same type as each hex tile adjacent to their second settlement. Finally, the turn order reverts to normal and the game proper can begin.

3.3. Victory Conditions

Victory in the three to four player version of Catan is achieved by the player who is the first to earn ten Victory Points, which are rewarded by the game when certain actions are completed. The simplest way to earn a Victory Point (VP) is to build a settlement, and as such all players start with two "free" points since each starting settlement counts for one point. Another way for players to gain VPs is to upgrade an existing settlement to a city, which grants an additional VP per city built. A third source of VPs is the development card deck, which primarily provides special gameplay advantages to players but also contains five cards that immediately grant one VP each to the player who acquires them. Respectively, these special cards are titled Chapel, Library, Market, Palace,

and University. Finally, there exist two special cards that can be granted to players who satisfy their conditions. The first of these is Longest Road, which is valued at two VPs and is given to the player who has the longest connected road of length at least equal to five. Equivalent in VP value to Longest Road, the Largest Army card is given to the player who has played the largest number of Knight development cards, with a minimum of three played Knights necessary to acquire the card in the first place. Both of these cards can change ownership throughout the game, an event that happens whenever another player becomes the superlative road builder or army commander, respectively.

3.4. Typical Turn Workflow

3.4.1. Rolling the Dice The first step in any player's turn is to roll the two standard six-sided dice included with the game in order to determine resource production. For each hex tile whose dice roll chit displays the rolled number, all players who own a settlement or city at one of the tile's vertices receive one or two resources of the hex's type, respectively. If there are not enough resource cards to be distributed according to this rule, then no player receives any resources of the type or types lacking sufficient cards in the resource decks. A special rule, henceforth referred to as the robber's rule, applies to rolls where the dice total seven in any configuration. The robber's rule dictates that the player who rolled the dice must move the robber object from its current position to a different tile of that player's choosing. If the robber is moved to a tile where other players own settlements or cities, the current player chooses one of these players and "steals" one random resource card from that player's hand. For the duration of the robber's residence on the new tile, no resources are allocated for that tile when the dice are rolled. One other aspect of the robber's rule is that all players who possess more than seven resource cards must choose half of them to discard and return to the resource decks, where fractional numbers of cards are rounded down to the nearest integer. It is important to note that players cannot perform any other actions during their turn before rolling the dice, for example in an attempt to spend resource cards for fear of rolling a seven.

3.4.2. Engaging in Trade After resources are allocated, a player may engage in either maritime trade or trade with other players, where "maritime trade" refers to the ability to swap multiple

resource cards of the same type for one card of a different type without interacting with other players. The default maritime trade ratio is four to one, but this can be augmented by the ownership of settlements or cities on an intersection adjacent to a port along the edges of the Catan island. Specifically, generic ports grant a ratio of three to one for any kind of resource, while resource ports grant a two-to-one ratio only for a particular kind of resource, for example enabling a trade of two Lumber cards for any other resource card. For player-to-player resource trades, any combination of resource cards can be traded provided that both players agree to the trade, with the caveat that only the player whose turn it is may initiate player-to-player trades.

3.4.3. Building Roads, Settlements, Cities, and Development Cards The primary activity undertaken during a player's turn is that of building objects, in accordance with the resource build costs printed on the four identical informational cards that are given to each player at the start of the game. Roads cost one Lumber card and one Brick card and may only be built adjacent to existing roads and/or existing settlements/cities owned by the same player. Furthermore, each hex tile edge permits the construction of only one road at a time, and roads may not be destroyed by any means during the course of the game. Finally, each player is allowed to build a maximum of fifteen roads, a restriction that grants paramount importance to careful road placement, particularly given the fact that roads cannot be reclaimed once placed.

Settlements cost one of each distinct resource, with the exception of Ore, and may only be placed on hex tile vertices that do not already house a building of any type. Aside from the initial two settlements placed in the game setup stage, settlements must be built adjacent to an existing road owned by the same player. Furthermore, settlements must be separated by at least two hex tile edges, whether or not they are owned by the same player. Like roads, settlements cannot be destroyed, but players can replace their own settlements with cities provided that they have sufficient resources. Each player may build up to five settlements, but they can be reclaimed by city upgrades. Finally, players receive one Victory Point for the construction of each settlement, as described in the Victory Conditions section above.

Cities cost two Wheat cards and three Ore cards and may only be placed on hex tile vertices with

settlements owned by the same player. Like roads and settlements, cities cannot be destroyed by any means. Each player may build up to four cities on the board, and unlike settlements there is no way to reclaim cities once they are placed. Cities are considerably more valuable than settlements in terms of resource production capabilities, producing twice as many resources as settlements as described in the Rolling the Dice section above. They are also the main vehicle for the pursuit of victory, granting two VPs each, and in fact it is virtually impossible to win the game without the strategic deployment of cities.

Development cards each cost one Wool, one Wheat, and one Ore, and they can be kept until their owner deems it appropriate to enter them into play. However, it is important to note that only one development card may be played during a player's turn, and furthermore a card cannot be played during the same turn in which it is acquired. The most common type of development card is the Knight, which constitutes a majority of fourteen out of twenty-five total development cards. When played, the Knight card allows a player to move the robber and steal a resource card from a player, as described above in the discussion of the robber's rule, but unlike the event where a seven is rolled, no players are required to discard half of their resource cards if they possess more than seven cards. Furthermore, Knights contribute to players' eligibility to receive the Largest Army card, with three Knight cards being the minimum necessary to acquire the card in the first place. It is important to reiterate that the Largest Army card only transfers ownership once another player's collection of played Knight cards exceeds the size of the next largest player-owned army.

The next most common development card type is that which immediately grants one Victory Point to the player who acquires it. There are five distinct cards in this category, as enumerated above in the Game Components section: Chapel, Library, Market, Palace, and University. Unlike other mechanisms for acquiring VPs, ownership of these cards is not made public to other players until their owner chooses to do so, which generally occurs when a player has enough VPs to win the game. Because of this property of VP cards, they are highly coveted and can be used strategically by clever players. In fact, the existence of VP cards is an important motivation for players to spend the resources necessary to acquire development cards, because they can be used to take other players by

surprise and surge ahead to a leadership position.

The final category of development cards that can be built during a player's turn is that of progress cards. Conferring a specific gameplay advantage to the player who deploys them, there are two of each of three different types of progress cards available in the game. The Road Building card allows players to build two roads according to ordinary road-building procedures, at no additional cost beyond that which was required to acquire the card. This card cannot, however, grant players additional roads beyond the physical per-player limit of fifteen roads. The Monopoly card allows its owner to name one specific resource type and collect all cards of that type in the possession of all other players. Finally, the Year of Plenty card allows its owner to collect any two resource cards from the remaining pool of cards in the resource decks.

4. Implementation Details [2]

In the following sections, I will take the reader on a journey through the workflow of my Catan implementation, from the dynamic generation of the game board to the functioning of the game as it is played. The goal of the discussion is to show the reader how the program actually works rather than to dissect its components in a sterile clean-room. As such, the progression of the discussion will follow the rough outline of the program's operation, introducing concepts and data structures when the necessity arises and explaining important design decisions along the way. Additional discussion of design challenges will take place in the Challenges Encountered section of this paper.

4.1. Insertion Point

As for all computer programs, execution begins with the `main` method, which in the case of the Catan implementation currently being discussed is in the `Game` class. The program is executed by typing `java Game NUM_PLAYERS BOARD_RADIUS BOARD_DIMENSIONS` at the command line, where `NUM_PLAYERS` is the number of players desired, `BOARD_RADIUS` indirectly describes the logical size of the board desired (note: a more detailed explanation of board radius can be found in the next section), and `BOARD_DIMENSIONS` lets the `BoardDraw` class know how large the rendered board should be in pixels (the graphical output is square, so only one parameter

is needed). If no values are entered or the values entered cannot be parsed as integers, the program lets the user know that the default values (`NUM_PLAYERS = 3`, `BOARD_RADIUS = 3`, and `BOARD_DIMENSIONS = 700`) will be used. Whether the user's input is taken into consideration or the default values are used, the program proceeds to create a new `Game` object, which sets into motion the constructor of that class. The constructor calls the `startGame` method, whose first line calls the `init` method of `Rules`, the main subject of the next section.

4.2. The `Rules` Class

The purpose of the `Rules` class is to encapsulate as many of the rules of Settlers of Catan as possible. It contains a number of constants that describe immutable (but customizable) features of the game, such as the fixed set of 19 hex tiles that are used to create a board of standard size, the 18 dice roll chits that are to be placed on each tile except for the desert, and the cost of building roads, settlements, cities, and development cards. All of the remaining data structures in this class, including the logical structure of the game board, are generated dynamically from a single seed, the `radius` value entered by the user as described above. As its name suggests, this user-provided parameter measures the distance in hexes from the center of the board to the edge of the ocean surrounding it. A more useful definition of the radius, however, is the number of hex "rings" that make up the board. The center hex comprises the first ring, and the remaining rings grow outward in a concentric fashion. The default value for the radius is 3, as mentioned above, which produces an ordinary Catan game board. Since the radius can take values higher than 3, it is possible to play Catan with larger boards of arbitrary size. The manner in which the board is generated, a task split between the `Rules` and `Board` classes, is the subject of the next section.

4.3. Dynamic Board Generation

4.3.1. Numbering Conventions When playing Settlers of Catan using a physical board, it is easy to refer to a desired hex or intersection simply by pointing to it with a finger. However, in order to interact with a computer-based representation of a Catan board, a unique identifier for each hex and intersection is required. The simplest and most obvious option is to use integer IDs for this task, but

the immediate and inevitable issue that arises is the manner in which these IDs are assigned. It turns out that, for both hexes and intersections, the ring paradigm mentioned above makes it possible to perform this assignment in a logical and scalable way. For hexes, the numbering begins at 0, with the solitary hex in ring 0, and proceeds outward to the right and then in a counterclockwise direction. Thus, the hex immediately to the right of hex 0 is assigned ID 1, and then the hex up and to the left from hex 1 is assigned ID 2. The three rings of a standard board that result from this numbering scheme contain the following ID ranges, in ascending order: 0, 1-6, and 7-18. For intersections, a similar assignment mechanism can be followed, with each ring beginning at the bottommost intersection of the rightmost hex in the corresponding hex ring. The three rings that result contain the following ID ranges, in ascending order: 0-5, 6-23, and 24-53. This numbering paradigm lends itself to relatively straightforward implementation in code, as explained in the following section.

4.3.2. Rings of Catan The first step in the board generation process is the initialization of two data structures, `hRings` and `iRings`, which store the IDs for hexes and intersections, respectively. Each is configured as a two-dimensional `ArrayList`, where rows index the rings and columns index the items within each ring. For hexes, the size of each ring `i` (except for ring 0) is determined by `i * HexShape.NUM_SIDES`, where `HexShape.NUM_SIDES = 6`. For intersections, the size of each ring `i` is determined by `((2 * i) + 1) * HexShape.NUM_SIDES`. Each entry in `hRings` and `iRings` stores the actual ID of a hex or intersection, so the net result of using these two-dimensional arrays is to allow IDs to be accessed using the row and column indices. This is immensely useful because, as the next two sections illustrate, there exist exploitable relationships between the column indices of adjacent rings.

4.3.3. Hex Graph The next step is to generate the hex graph, which is called `hGraph` and is stored as an adjacency list in a **`boolean`** two-dimensional array, where **`true`** indicates that two hexes (indexed by their integer IDs, as described above) are adjacent to each other. Having this data structure available and initialized is of crucial importance when attempting to randomize the dice roll chits, because the rules of Catan, enumerated in great detail above, stipulate that dice roll chits 6 and 8 cannot be adjacent to each other. In fact, in the absence of a data structure that stores

information about which hexes are adjacent to each other, this rule cannot be enforced at all.

The generation of `hGraph` involves the use of `hRings` and two loops that iterate through the rows and columns of that data structure, such that all hexes are traversed in the same order in which their IDs were initially assigned, as described above. For each hex, the algorithm first looks at the next hex in the same ring, and then at the adjacent hexes in the next ring. Looking ahead within the same ring is a straightforward operation in that `j`, the column index, is simply incremented by 1 at each step, with the caveat that modulus arithmetic must be used so that the last hex in a ring can "wrap around" to the first hex in that ring. However, determining which hexes in the next ring (if it exists, i.e. the current ring is not the last one) are to be examined and marked as adjacent to the current hex is a bit trickier and involves the introduction of a couple of simple concepts.

First, it is necessary to categorize hexes as either corners or not corners. A "corner" is defined in the following way: a hex is a corner if and only if its column index (its index inside the ring it is in) can be evenly divided by its row index (the index of the ring it is in). Visually speaking, the centers of all corner hexes lie on one of three axes that are perpendicular to the six sides of the center hex and pass through its center. Any hex that does not fit this criteria is not a corner. This distinction is important because corners are adjacent to three hexes in the next ring, while non-corners are only adjacent to two hexes in the next ring.

Second, the board needs to be divided into sextants, much in the same way as the two-dimensional Cartesian plane is divided into quadrants. As the name suggests, a "sextant" is one of six equal-sized slices into which a regular hexagonal Catan board (i.e. a board that has symmetry along the three corner axes) can be divided. Due to the nature of the ring indexing paradigm, it is easy to determine which sextant a given hex falls into simply by looking at its row and column indices, `i` and `j`, respectively: `sextant = j / i`. Note that, since `i` and `j` are both integers, integer division is performed and the remainder is ignored (although, as above, the existence of a remainder determines whether or not a hex is a corner). Also, observe that `sextant` can take values in the range 0-5.

Having introduced the corner and sextant concepts, it is now possible to explain the manner in which the hex graph algorithm determines, for a given hex, to which hexes in the next ring that

hex is adjacent. A typical hex located at row index `i` and column index `j` in `hRings` is adjacent to the two hexes in the next ring (row index `i + 1`) whose column indices are `j + sextant` and `j + 1 + sextant`. The role of the `sextant` value here is to account for the fact that each ring contains 6 more hexes than the previous ring, distributed evenly across the 6 sextants. Thus, sextant 1 has to account for the extra hex contributed by sextant 0, sextant 2 has to account for the extra hexes contributed by sextants 0 and 1, and so on. If the current hex is not a corner, the story ends here. However, if it is a corner, then there is an extra hex in the next ring that must be considered, which is indexed by `j - 1 + sextant`. An extra wrinkle is that it is necessary to account for wrap-around by using modulus arithmetic, since `j - 1 + sextant` can become negative if `j` and `sextant` are both 0.

4.3.4. Intersection Graph Following the initialization of the hex graph, the next step is to generate the intersection graph, which is important because the edges between intersections constitute the only valid locations for the placement of roads during a game of Catan. Similar to the hex graph, the intersection graph is titled `iGraph` and is stored as an adjacency list in a **`boolean`** two-dimensional array, where **`true`** indicates that two intersections (indexed by their integer IDs, as described above) are adjacent to each other. Note that, although hexes and intersections are inextricably linked on a physical board by virtue of the board being flat, in the implementation presently being discussed hexes and intersections are logically separate entities (that said, the next section describes the initialization of the logical bridges between hexes and intersections).

The generation of `iGraph`, like that of `hGraph`, involves the use of two loops to traverse `iRings` in order of ascending intersection ID. However, a key difference is that the `j` index of the inner loop begins at 1 and ends at 0, using modulus arithmetic to wrap around to the beginning of the current ring when the end is reached. The reason for this will be explained in more detail shortly, but the basic point is that starting at 1 rather than 0 eliminates the need to have a special case for the first ring. For each intersection traversed, the algorithm first looks at the next intersection in the same ring, and then at the solitary adjacent intersection in the next ring, if one exists. Looking ahead within the same ring is a straightforward operation in that `j` is simply incremented by 1 at

each step, but it takes a bit more work to determine when a link to the next ring needs to be created.

A basic observation is that every intersection in a ring contains 3 links to other intersections, 2 of which are in the same ring (pointing forward and backward) and one of which points to either the previous ring or the next ring. Since the algorithm only moves forward, it is only necessary to consider one of the links within the same ring, as described above. However, the third link alternates between pointing forward and backward, a fact that requires the use of a **`boolean`** called `hasNextLink`, which flips back and forth between **`true`** and **`false`**. The other challenge posed by the third link is determining the column index of the intersection in the next ring. Since the algorithm starts at `j = 1`, the very first intersection examined for any ring always has a link to the next ring, specifically to column index 0. Thus, we initialize `hasNextLink = true` and `ringIndexDiff = -1`. Recall from the Rings of Catan section above that each ring has 12 more intersections than the previous ring, a difference which can be distributed across the 6 corners of each ring such that each corner contributes 2 extra intersections. This can be verified by visually comparing corner and non-corner hexes: for a corner hex, 2 of its 6 vertices are in one intersection ring and the remaining 4 in the other, whereas for a non-corner hex the split is 3 and 3. Now consider the fact that for each hex, corner or non-corner, 2 of the 6 sides connect two intersection rings. Since the algorithm only moves forward in a ring, only one of these sides is relevant, so we can count hexes by the number of intersections in one ring that connect to the next ring. For any given hex ring `i`, the number of non-corner hexes in a sextant is equal to `i - 1`. Because extra intersections appear only at corner hexes, `ringIndexDiff` only gets incremented by 2 at a corner hex after one of the 2 corner hex intersections in the first ring is traversed. Thus, we can set a counter variable `waitTillNextInc = i`, since the corner hex gets counted in addition to the non-corner hexes. For each sextant, this variable is initialized and then decremented at every other intersection, since as established above only every other intersection has a link to the next ring. Using `hasNextLink`, `ringIndexDiff`, `waitTillNextInc`, and the logic described above, it is thus possible to determine when a link to the next ring needs to be created and also the value of the column index of the intersection in the next ring to be linked to, which is simply

`j + ringIndexDiff`.

Having explained how the algorithm functions, it is now possible to return to the minor detail of beginning the algorithm at column index 1 instead of 0 and explain this design choice more fully. The reason for this design decision is that the link between column indices 0 and 15 of rings 0 and 1, respectively, would be difficult to capture without resorting to the use of special cases, either having a special case for the entire first ring or just for that troublesome link. By starting at column index 1, it is possible to initialize `ringIndexDiff = -1` as above and allow the algorithm to increment this variable until it reaches 15 once column index 0 in ring 0 is finally traversed.

4.3.5. Hex to Intersection Mappings As mentioned above, hexes and intersections are logically separate entities in the Catan implementation presently being discussed, but it is now necessary to create logical bridges between them, referred to henceforth as "mappings". These mappings occur in two directions: from hexes to intersections and from intersections to hexes, titled `hIMapping` and `iHMapping`, respectively. Both of these data structures are two-dimensional `ArrayLists` of integers, storing only intersection or hex IDs. `hIMapping` is set up such that the row indices refer to hex IDs and the columns store the 6 intersection IDs associated with each hex (the ordering and indices of the columns is irrelevant). `iHMapping` is set up in a similar manner, such that the row indices refer to intersection IDs and the columns store the 3 hex IDs associated with each intersection (again, the ordering and indices of the columns is irrelevant). `hIMapping` is needed for the scenario where the robber is moved to a new hex and the game has to know which players are eligible to have a resource card stolen by the player who moved the robber, by proxy of the player-owned settlements or cities that border that hex. `iHMapping` is needed to initialize the `Intersection` data structure (discussed in more detail later), so that when the time comes for players to collect resources after the dice are rolled, each `Intersection` owned by a player can know which hexes it borders and thus whether its player owner should be given one or more resource cards, depending on whether a settlement or city was built there.

The algorithm used to initialize these two mappings is reasonably straightforward, but there are several moving parts that require some explanation. Two loops are used to iterate through `hRings`,

such that `j` indexes the current hex ID in `hRings`, as in the Hex Graph section above. However, for each hex, intersections are also iterated in parallel, and in fact the row index `i` serves the dual function of indexing rings for both `hRings` and `iRings`, since the number of rings for each is equal to `radius`. For each hex traversed, both the current intersection ring (at index `i` in `iRings`) and the previous ring (at index `i - 1`) are visited, with `curIIndex` and `prevIIndex` being used to keep track of intersection column indices for the current and previous rings, respectively. The concept of corner hexes is again used here to determine what the split of each hex's intersections is with regard to the current and previous intersection rings. Recall from the previous section that the split for corner hexes is 4 and 2 for the current and previous intersection rings and 3 and 3 for non-corner hexes. This information is used, for each hex, to traverse the intersections in the current intersection row, add an appropriate entry to each mapping (an intersection entry for `hIMapping` and a hex entry for `iHMapping`), and then do the same for the previous intersection row. Once all hexes are traversed, the algorithm finishes and both mappings are fully initialized.

4.3.6. Resources The official Catan rules specify exactly how many resource tiles of each type (Wool, Grain, Lumber, Brick, Ore, and also Desert) are to be represented when putting together the game board, and for the physical game these are fixed by the printed hex tiles provided in the box. However, since the implementation presently being discussed allows for the use of board sizes larger than that produced using the standard radius parameter of 3, it is necessary to implement a method that decides how many additional tiles of each type are to be added to the board once the fixed 19 are exhausted. Because this decision has to be made outside of the official rules, which do not provide any information about tile type distribution beyond the 19 given tiles, I used my discretion to choose a scalable and fair method for generating additional tiles.

Before the simple decision algorithm is explained, however, a brief discussion of the basic `Resource` data type is required. The `Resource` class is essentially a wrapper for integer constants used to denote each different resource type, from `DESERT = -1` to `ORE = 4`. The ordering of these integer values reflects the relative worth of each resource type, with `DESERT` being the least valuable since no resource cards of that type can be collected (and even if such cards

existed, the standard build costs would still not include Desert). Other elements of `Resource` that are worth noting are the `Color` and `String` constants, for use by `BoardDraw` and for naming the resources, respectively. There also exists a **`static`** method called `getResourceType` (which overloads the `getResourceType` method that takes no arguments) for finding the integer value that corresponds to the given `String` parameter `sResource` containing the name of a resource. This method can be seen as the counterpart (in reverse) to the overridden `toString` method that gives the `String` name of a `Resource` instance. It is worth noting that `Resource` is used both for initializing the `Hex` data type (discussed in the Hexes section) and also as a resource card data type for initializing the resource deck. This dual use of `Hex` is the reason why `DESERT` is set to -1 rather than 0. Because the resource deck does not contain any Desert cards (as mentioned above) and is crucial to the functioning of the game (building anything costs resources), the latter use case is prioritized, so that `NUM_TYPES` can correspond to 1 greater than the last resource value (`ORE`) in order to allow these integer constants to be appropriated as array indices for the `ResourceBundle` data type.

The hex tile generation algorithm begins by adding the standard 19 hex tiles, which are stored in `DEFAULT_HEXES` as an array of frequencies, to `hexTiles`. A slight detail to take note of in the loop that implements this operation is that the indices in `DEFAULT_HEXES` are shifted by one, since `DESERT = -1` and thus cannot be used unmodified as an array index. If the user-provided radius parameter is equal to the default value of 3, then the story ends here. For larger boards, the algorithm continues by iterating through `hRings` and adding equal amounts, determined by `curRingSize / Resource.NUM_TYPES`, of each non-Desert resource type to `hexTiles`. If the current ring happens to be evenly divisible by 5 (the value of `Resource.NUM_TYPES`), then the algorithm continues on to the next ring. Otherwise, a **`switch`** block decides what resource types to assign to the remaining hexes that are unaccounted for. I chose to implement this decision such that extra Desert tiles are added for all nonzero remainders except 3, in which case all non-Desert resources except Brick and Ore are added. The reasoning behind this is twofold: first, a huge Catan island would look odd with only one Desert tile on it, and second, the relative abundance of

Wool, Grain, and Lumber on the standard Catan board is higher than that of Brick and Ore, so it makes sense to maintain this property for larger boards when possible.

4.3.7. Dice Roll Chits

4.3.8. The `Board` Data Type

4.3.9. Hexes

4.3.10. Ports

4.3.11. Intersections

4.3.12. Roads

4.4. Rendering the Board with StdDraw

4.4.1. `HexShape` Data Type

4.4.2. Hex Coordinates

4.4.3. Intersection Coordinates

4.4.4. Port Coordinates

4.4.5. Drawing

4.5. (More sections to come)

5. Challenges Encountered

6. Future Features

7. Conclusion

References

[1] R. Clobus *et al.*, "Pioneers - The Settlers of Catan," http://sourceforge.net/projects/pio/.

[2] V. Costescu, "SettlersThesis2014," https://github.com/lbds137/settlers-thesis-2014.

[3] S. De Toni, "Solitaire Settlers of Catan ComputerGame," http://sourceforge.net/projects/solitairecatan/.

[4] B. Eskin, "Like Monopoly in the Depression, Settlers of Catan is the board game of our time," http://www.washingtonpost.com/wp-dyn/content/article/2010/11/24/AR2010112404140.html, Nov. 2010.

[5] J. Fugate, "Settlers3D," http://sourceforge.net/projects/settlers3d/.

[6] K. Law, "Settlers of Catan: Monopoly Killer?" http://mentalfloss.com/article/26416/settlers-catan-monopoly-killer, Nov. 2010.

[7] S. Maddock, "HTML5 Settlers of Catan," https://github.com/samuelmaddock/html5-settlers-of-catan.

[8] M. Masnick, "How Lawyers For Settlers Of Catan Abuse IP Law To Take Down Perfectly Legal Competitors," http://www.techdirt.com/blog/wireless/articles/20110211/21200213066/how-lawyers-settlers-catan-abuse-ip-law-to-take-down-perfectly-legal-competitors.shtml, Feb. 2011.

[9] J. Monin, "JSettlers2 - Java Settlers of Catan," http://sourceforge.net/projects/jsettlers2/.

[10] A. Raphel, "The Man Who Built Catan," http://www.newyorker.com/online/blogs/currency/2014/02/klaus-teuber-the-settlers-of-catan.html, Feb. 2014.

[11] K. Teuber, "The Official Website for Settlers of Catan - About Us," http://www.catan.com/about-us/klaus-teuber.

[12] K. Teuber, "The Official Website for Settlers of Catan - Electronic Games," http://www.catan.com/electronic-games.

[13] K. Teuber, "The Settlers of Catan Game Rules & Almanac," http://www.catan.com/files/downloads/soc_rv_rules_091907.pdf, Sep. 2007.

[14] J. Wingrove, "Settlers of Catan: A low-tech island retreat for a plugged-in generation," http://www.theglobeandmail.com/technology/settlers-of-catan-a-low-tech-island-retreat-for-a-plugged-in-generation/article583018/, Jun. 2011.