Incorporating Coherent Terrain Types into Story-Driven Procedural Maps

Elizabeth A. Matthews and Brian A. Malloy
School of Computing
Clemson University
Clemson, South Carolina 29634
{ematth2,malloy}@clemson.edu

Abstract

We present a system for procedurally generating a map for a story-driven game, where locations in the map are assigned algorithmically or by designer preference. In addition, we also generate terrain, together with climate to match the terrain, with smooth, coherent transitions between terrain exhibiting different weather. We summarize weather approximations using a tuple to represent conditions such as temperature and humidity. We then exploit our previous work in map construction by placing locations of interest in the story on the map and then build a terrain boundary map that determines the boundaries between ranges of tuple values that belong to specific terrain types. We complete our construction by combining the climate map with a terrain type lookup, producing a final map with cohesive terrains. We describe the implementation of our system and illustrate the construction with some procedurally generated maps, including the procedural generation of the Narshe/Figaro area from Final Fantasy VI.

Introduction

Some game companies have forsaken the story-driven genre in favor of multiplayer, first person shooter games such as *Call of Duty* and *Battlefield*, because of their potential for high revenue. In fact, *Call of Duty: Modern Warfare 3* was able to pass the one billion mark in revenue only 16 days after its release (Bedigian, 2012). However, the recent success of *The Elder Scrolls V: Skyrim*, whose revenues have exceeded all previous Elder Scroll games including Morrowind and Oblivion, has motivated some game companies, such as *Remedy* for example, to produce story-driven games (Sterling, 2012; Yin-Poole, 2012).

Gee has shown that learning occurs in playing all types of video games, where the player must learn the rules of the game, the names of characters, the game map and the geography of the world (Gee, 2005). However, the puzzle-driven nature of story-driven games offers two key aspects that enhance the learning experience: *domain knowledge* and *insight* (Vara & Osterweil, 2010). The domain knowledge required in story-driven games involves the information that the player needs to be able to solve a puzzle in the game, whereas insight is required to be able to apply the domain knowledge to solve the particular puzzle posed in the game.

However, the problem with the story-driven genre is its lack of replayability, so that once the player has learned to traverse the game world, acquire domain knowledge, and solve the puzzles in the game, there is little incentive to replay the game. Nevertheless, the story-driven genre offers the potential for large revenue, as illustrated by the success of Skyrim, as well as the potential for enhanced learning far beyond what may be acquired by playing a first person shooter style video game, where there are fewer puzzles to solve and game play is motivated by the adrenaline rush of shooting and explosions.

One approach to enhancing the replayability of the story-driven genre is to exploit *procedural generation*, which is the development of game media using algorithms. Procedural generation has been used successfully in story-driven games including the *Diablo* series, where the dungeons in the game were procedurally generated, so that a player was unlikely to encounter the same dungeon layout more than once. Some research has taken this concept further. For example, Dormans uses a context free grammar to arrange various terrain artifacts to support the story (Dormans, 2010). Similarly, Hartsook et al. have developed *game forge*, a system for procedurally generating a playable story-driven game (Hartsook, Zook, Das, & Riedl, 2011).

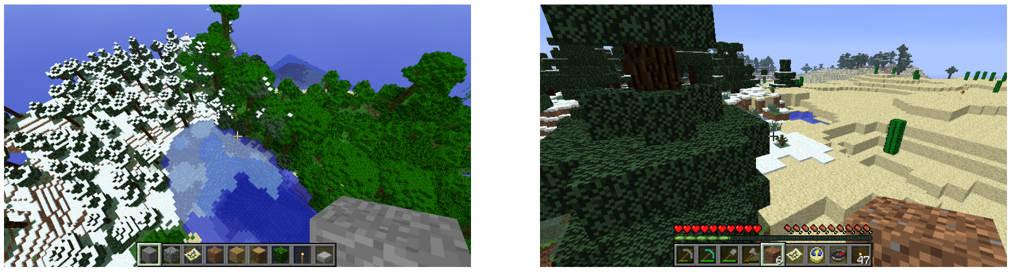

(a) Snowy Biome adjacent to a Jungle Biome (b) Snowy Biome adjacent to a Desert Biome

Figure 1. : Two terrain samples from the virally popular game Minecraft. Both samples exhibit starkly contrasting terrain placed adjacent to each other. Using the technique described in this paper, a game developer can establish terrain with smooth, coherent transitions between contrasting terrain types.

However, no previous research has been developed that permits the procedural generation of terrain with matching climate, where transitions from different terrain types, and different climate patterns, are guaranteed to occur in a smooth, coherent fashion. Even in the virally successful game *Minecraft*, a player might encounter a snowy Biome directly adjacent to a dessert Biome, with no coherent transition in between the two Biomes. For example, Figure 1 illustrates two terrain samples from Minecraft: the terrain in Figure 1a illustrates a snowy Biome directly adjacent to a jungle Biome, and Figure 1b illustrates a snowy Biome directly adjacent to a desert Biome.

In this paper, we present a system for procedurally generating a story-driven game world. We summarize weather approximations using a tuple to represent conditions such as temperature and humidity. We then exploit our previous work in map construction by placing locations of interest in the story on the map and then build a terrain boundary map that determines the boundaries between ranges of tuple values that belong to specific terrain types (Matthews & Malloy, 2011). Finally, we complete our construction by combining the climate map with a terrain type lookup, producing a final map with cohesive terrains and climate patterns. The technique that we present provides flexibility to the game designer to produce coherent terrain transitions, thereby precluding the possibility of radically different terrain types being placed directly adjacent to each other.

In the next section we define terms and provide background information. In the *Related Work* Section we describe the research that relates to our work, and we then describe the problem that we address in this paper. We describe our methodology in the section titled *Coherent Terrain Map Construction*, which contains the technique that we use to build a coherent terrain map. We describe our implementation of the technique and provide some results of some game maps that we have produced, including the procedural generation of the Narshe/Figaro area from Final Fantasy VI. Finally, in the last section we draw conclusions.

Background

In this section, we define the terms and concepts that we use in our research on map generation. We begin with a description of *procedural generation*, followed by a review of a procedural generation technique referred to as *Perlin Noise*. Finally, we describe the use of *maps* in games and how they are, typically, interwoven with a *level* in the game.

Procedural Generation

Procedural generation is the process of using an algorithm to generate audio or video content for games, movies or other digital media. Using a random number, generated from a *seed* or start number, and a set of descriptive parameters, procedural generation techniques can create unique media tailored to a set of pre-defined criteria. The basic structure and variance of a particular generated content is determined by the procedure used to create the content. An important axiom that applies to virtually all procedural generation techniques is that a given seed will always produce the same content and that a variation of the seed will produce variations in content, and sometimes different seeds can produce a wide range of variations in content. Procedural generation has massive applicability to video games because it can be used to produce unique and non-repeated textures for objects in the game, such as buildings or trees; or, it can be used to build a set of unique maps or dungeon layouts.

Perlin Noise

Perlin Noise is a procedural generation technique, developed by Ken Perlin, to generate a texture, sometimes referred to as a *procedural texture* (Perlin, 2002). Procedural textures generated using Perlin noise are used by visual effects artists to increase the realism in a computer generated scene, due to the pseudo-random appearance of the texture. Perlin noise is frequently used to create effects such as fire, smoke, or clouds. In spite of the random appearance of procedural textures, they retain the reproducible quality of all procedurally generated media.

Levels and Maps

In video game terminology, the concept of a map is frequently interwoven with the notion of a level. A level can assume several different meanings, depending on the particular game being played. For example, a *level* may refer to the degree of skill or ability that a player currently possesses in the game; in this sense, the term level-up refers to an increase of skill to the next " level." A second meaning to the term *level* refers to the total space available to a player in their attempt to complete a specific objective in the game (Wikipedia, 2012). A historical example of a level can be found in early role-playing games where it referred to a level in a dungeon; players typically begin at the bottom level and proceed through increasingly numbered levels, usually also increasing in difficulty, until they reached freedom at the top level (Wikipedia, 2012). Alternatively, they might begin at the top level and proceed to lower levels until they arrived at a "treasure" at the bottom level.

A map in a dungeon-based game would typically show the current level of the dungeon, or, the current dungeon in the case of multiple dungeons. An interesting and important example of procedurally generating a map for a dungeon can be found in the popular game series Diablo.

A map is a visual description of a level, with important aspects of the map, such as the current player position, objective(s), treasures, rivers, streams, mountains, and other artifacts, clearly marked. There can be more than one map associated with a particular game; for example, a game might have a world map, a level map, and even a map of a cave that the player is currently exploring. In our research, we typically are referring to the *world* map, or a map that encompasses the entire world that a player would explore.

Minecraft is a popular game with high procedurally generated content. In Minecraft, a *Chunk* of the world is procedurally generated as the player explores the world. As the player continues to explore, the world is generated on demand as the player moves through, and continues to explore, the world. When talking about maps in Minecraft, one must be careful because there is a in-game item called a Map, which is useable in game by players as a visual representation of a small nearby area of the world as a whole. When we use the term *map*, we mean the entire world, rather than an item in the game. Because Minecraft's world is technically infinite, the term map as defined for our project does not transfer well.

In Minecraft, the physical world is generated in chunks, sections of land that can be generated on the fly when the player first reaches it, but are stored in a save file after an initial visit so that any modifications the player does to the terrain are permanent. Different types of terrain found in Minecraft are called *Biomes*, which are sections of land with similar characterisitcs, such as a rainforest or tundra. The physical height levels match up between Biomes and chunks due to the inherent procedural algorithm used by Minecraft.

Minecraft distinguishes Biomes with *temperature* and *rainfall*, as compared to our notion about temperature and humidity. Our climates are based on temperature and humidity, which we feel is more closely aligned with actual weather patterns, but not necessarily more complicated to store. Minecraft Biomes are haphazard in placement and are not required to blend in with surrounding Biomes, as seen in Figure 1. By contrast, the Biomes, or terrain, that we wish to build for story driven games are required to be coherent with neighboring climate patterns, since an abrupt change in terrain would throw off the immersion of the player.

Related Work

Procedural generation is used extensively in video games, both in the studio for pre-production media generation and, less frequently, for dynamically generated media during gameplay (Andrew Doull, 2011). *Canabalt* is a recent popular, 2D, online, platform game where all actions are automated except for the players ability to jump using a single key press. The extended replay value of Canabalt is due, in part, to the fact that the player must jump from one roof to the next and the height, length and distance between each roof is determined algorithmically. Procedurally generated media that can be developed dynamically can not only extend the replay value of the game, but permit wide variations in constraints that can be placed on the generation of a game world, including facility for guaranteeing coherence in the placement of game objects, including user-specified constraints (Matthews & Malloy, 2011).

In addition to Canabalt, an application of procedural generation that is more relevant to our approach is the generation of dungeon layouts in the *Diablo* series, a fantasy-themed, action role-playing game (Blizzard Entertainment, 1996). Most of the dungeons and many terrains in Diablo I and II are procedurally generated. However, map pieces in the generated dungeons are occasionally placed so that the generated level might not always maintain fidelity with the game objective or story line. For example, a goal of a particular level might be to travel from an entrance to a staircase, but the entrance and the staircase might be placed adjacent to each other, rendering the objective trivially achievable. We have described an approach that permits the user to place constraints on the location of objects in the world, obviating the lack of story coherence found in the Diablo approach (Matthews & Malloy, 2011). In this paper, we extend this previously developed technique to include coherent procedural generation of climate.

Yannakakis has done extensive work in procedural generation, particularly focusing on a player's emotional experience, or how much "fun" was had (Yannakakis & Togelius, 2011). Replayability is the focus for our work, which is implied to increase the positive experience upon a player's successive play throughs of a particular game. Yannakakis focuses on reactive game content (Shaker, Yannakakis, & Togelius, 2012), with various models of a player's emotional levels influencing a game's level set up. Most relevent is his work on level generation (Shaker et al., 2011), where the physical layout of a platformer's level design is the desired result of the procedural generation. However, these techniques focus on action-like platformers, similar to Canabalt, where a level's layout needs to be coherent in the sense that it must be playable, but not necessarily fit a complex storyline, which is required for a RPG map generation. Additionally, a platformer has no concept of world-based ideas, such as climate or terrain.

Dormans describes an approach for procedurally generating levels in adventure games (Dormans, 2010). He summarizes a level as consisting of *missions* and *spaces*, where missions are the tasks that the player must perform to complete the level, and space is the geometric layout of the level. Dormans represents a mission as a directed graph indicating which tasks are made available by completing the previous task, and spaces are represented as a graph whose nodes are rooms and the edges are connections between the rooms. A level is generated in two steps, where missions and spaces are generated separately using different context free grammars and different generative algorithms for each. However, the approach does not permit a game designer the flexibility of easily specifying constraints on level generation and there is no facility for handling climate and coherent terrain generation.

Another approach to the procedural generation of levels using grammars is described by Smith et al. (Smith et al., 2011). The focus of their work is 2D platform games and they describe *Launchpad*, a two-tiered, grammar-based approach where the first tier is a rhythm generator, and the second tier creates the geometry of the level based on the rhythm generated in the first tier. The result is a collection of *rhythm groups* that can be combined to form a "level." Their approach creates complicated play fields with obstacles and collectible items consistent with the rhythm intended for the game, independent of the generated geometry. However, their approach does not provide for terrain or weather in the resulting geometry and they make no attempt to maintain coherency across rhythm groups that form the generated level.

Hartsook et al. describe techniques for automatically generating a fully playable *computer* *role-playing game* based on a story (Hartsook et al., 2011). The story can be written either by a human, or created by a computational story generator using information about the play style and preferences of the player. They describe a system, *game forge*, which can generate the story for a role-playing game, and map the story to a space using two metaphors: *islands* and *bridges*. Islands are areas where critical plot points occur, and bridges are areas between islands where non-plot specific game play can occur.

In their approach, a game *world model* is provided by the designer and *game forge* generates the game world through the use of a genetic algorithm (Hartsook et al., 2011). A fitness function is used to constrain the *world model* to more "natural" worlds. An environment transition graph is used to determine adjacency of environment types. For example, there may be a high probability of a cave being placed adjacent to a mountain, but there may be a low probability of a forest being placed next to a mountain. In our previous work, we describe techniques that provide a user interface and a spring-based physics engine to facilitate world generation by the designer, which provides more flexibility in map generation and makes it easier for the designer to adjust locations in the procedurally generated map (Matthews & Malloy, 2011). Also, the approach described by Hartsook, et al., does not include a facility for generating climate in a coherent fashion, as we do in this current article.

The Problem that we Address

The original goal of our work was to develop a technique for generating maps for story-driven video games that might enable a game developer to build a map with cities and towns strategically placed so that they align with constraints specified by the developer, and consistent with the story driving the game (Matthews & Malloy, 2011). We were successful in developing such a technique and we summarized our approach in reference (Matthews & Malloy, 2011). Our previous approach included constraints that permitted the game designer to address issues about cities, towns and other objects in the environment, such as number, distance, direction, navigation and placement; the technique also coherently filled the area around the city or town with appropriate terrain to encourage or discourage navigation, whichever might be appropriate within the context of the story.

However, our previous approach did not include a facility for procedural generation of climate considerations, or to guarantee that these procedurally generated climate patterns were consistent with the story and the terrain within which they might occur. We address this shortcoming in this paper and, in this section, we provide an overview of our approach to procedurally generate a coherent map for story-driven video games. In the next sub-section we describe our map generation system, and in *Overview of Coherent Terrain Map Generation*, we describe a problem in our previous approach that will be addressed in the final sub-section.

Problems in Our Previous Map Generation System

The approach to procedural map generation that we describe in reference (Matthews & Malloy, 2011), began with the assumption that the terrain must be fully integrated into the story that drives the game. Moreover, the progression of the story is likely to be intimately tied to progression through the map within which the story occurs. Thus, we developed an abstraction of the story that consists of the *Locations of Interest*, LOI, in the story and the restrictions between these LOI. A Location of Interest is a particular place where something important happens in the game's story. A typical LOI might be the town where the player begins, a cave where the player must confront an

```
1. Read or generate a list of LOI
   and their corresponding two-tuples;
2. Make an empty grid for the world;
3. Populate the grid with LOI
4. Interpolate
```

Figure 2. : Algorithm outline to build a Climate Map

enemy, or a hidden forest where the player might find an optional hidden castle. Using restrictions between these LOI, we described a system that permitted the developer to specify or approximate where the LOI may be placed, how far apart they might be, and whether or not there is a traversible path between successive LOI.

We exploited a physics engine to translate these parameters into a physical shape: the LOI are physical bodies in the engine, and the restrictions between the LOI are represented as spring-like connections between the physical representation of the LOI. This physical representation permitted us to build a space within which we might generate the coordinates of the LOI so that they do not violate the restrictions placed by the game designer (Matthews & Malloy, 2011). This previous system used a randomized floodfill for both the landmass generation and the terrain determination. However, the randomized floodfill approach that we utilized might generate terrains with illogical or incoherent terrain transitions, such as a desert next to snowy tundra. This article summarizes our solution to this problem by describing an approach to generating coherent transitions between different terrain.

Overview of Coherent Terrain Map Generation

To incorporate coherency in our procedural generation of maps, we need an abstraction that might capture the important considerations about weather, and that might permit us to build a map with smooth transitions between regions with different weather conditions. Our view is that climate will likely influence terrain, and that climate can be represented by a set of tuples that capture the important features of the weather. In our current approach, we capture weather with a two-tuple consisting of temperature and humidity. Using the LOI as points in a map that influence the climate, transitions between LOI can be inferred from these locations using interpolation. Then, the calculated, interpolated values can be used to determine the terrain type.

Coherent Terrain Map Construction

Climate in a story-driven game world should be *coherent* with the theme of the game and the world. While climate itself is a complex system to estimate, for video games the subtleties of actual weather patterns are not usually essential to the advancement of the plot. Inspired by Minecraft *Biomes*, we represent climate as a two-tuple of values that capture temperature and humidity; however, our approach is extensible to tuples of values that can be sufficiently large enough to capture other facets of weather and climate.

Using a two-tuple to summarize weather approximations, we begin our construction of a coherent terrain map by building a cohesive climate map based on the important locations in the story. Then, in the *Terrain Boundary Map* Section, we present our approach to the construction of

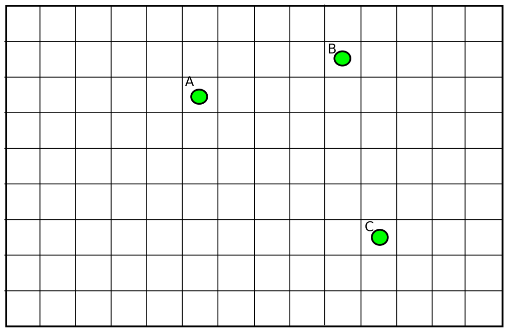


Figure 3. : This figures illustrates LOI in an interpolation grid.

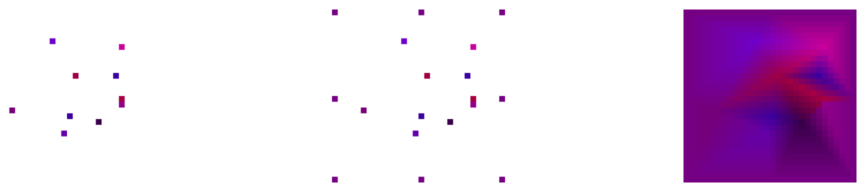
a Terrain Boundary Map that is used to determine the boundaries between ranges of tuple values that belong to a specific terrain type. Finally, in the *Final Mapping* Section, we complete our construction by combining the climate map with a terrain type lookup, producing a final map with cohesive terrains.

Climate Map Construction

Our approach to the construction of a climate map is summarized in Figure 2, which begins by generating or reading a set of Locations of Interest in the map, LOI, together with a corresponding climate attribute tuple for each location. The Locations of Interest are specialized for each story, for example, a desert palace, or an ice cave. The climate of the former would likely have high temperature and low humidity, while the latter would likely have low temperature and medium humidity. These locations might be included in the game map in several ways; for example, they may be created in a completely random fashion, or they may be directly related to the plot in a story driven game, which forms the subject of our research.

After reading or generating the LOI, we then build an empty grid, step 2 in Figure 2, and populate the grid with the LOI, step 3; Figure 4a illustrates a grid populated with LOI. To visualize the climates in Figure 7, *red* and *blue* are used for *temperature* and *humidity* respectively, resulting in *purple* for a high temperature and humidity rating or *black* for low in both values. Another visual example is seen in Figure 3, with three LOI: A, B, and C. Each one of these LOI would have a climate attached to it so that it would influence the passive area of the map.

Due to the nature of the interpolation, and to provide values for *all* areas of the map, edge locations are added, as shown in Figure 4b. Edge locations are managed in the same fashion as the



(a) Climate (b) Climate with edges (c) Climate Filled

Figure 4. : This figure illustrates our procedure for creating climates. Blue values are high humidity, red are high temperature, black is low in both, and purple are high in both values.

```
1. Populate list of Points of Terrain
2. Make empty grid in range 0...1
3. Put POT in grid
4. Use nearest neighbor algorithm to
   establish terrain boundaries
```

Figure 5. : Algorithm outline to build Terrain Boundary Map

LOI but are, in fact, place-holder locations that provide a climate for the edges of the map. The strict average of climates for all the LOI on the map is used for these edges. This allows for particular maps to have higher or lower average temperatures, as well as looping the map. Looping is when the player travels off the edge of a map and is transported to the opposite side. Looping is usually used in World Maps.

Once the LOI's influence is added to the grid, we interpolate values between the LOI for the passive map area, which is step 4 in Figure 2. This interpolation maintains map coherence by ensuring that no two neighboring tiles vary by too large a value in temperature or humidity, unless the developer explicitly places two greatly varying Locations of Interest within a small distance of each other; but these instances will be intentional rather than inconsistencies in the procedurally generated map. The interpolation also ensures that a world's climate will progress between two locations logically, instead of suddenly getting hotter when walking towards a colder climate town. A linear interpolation is used for speed, but other interpolation methods can be used to fill in the passive area of the map. Figure 4c illustrates our placement of linearly interpolated tiles to produce a climate filled terrain. Since climate typically follows irregular patterns, we introduce noise, as seen in the *Results* Section, into the climate map for a more realistic pattern (Perlin, 2002).

Terrain Boundary Map Construction

In order to translate this resultant climate map into a final game map, we build a terrain boundary map to determine the boundary for each particular terrain type. The steps used in the construction of a terrain boundary map are summarized in Figure 5. An obvious choice of tuple values for a terrain boundary map might entail the use of high temperature and low humidity to represent desert climate, and high temperature and high humidity to represent jungle climate, as seen in real world examples. However, to afford complete freedom to the developer in generating the world, the climates and their humidity/temperature boundaries can specified in an input file, or they be generated in an arbitrary manner.

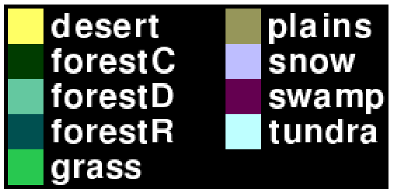
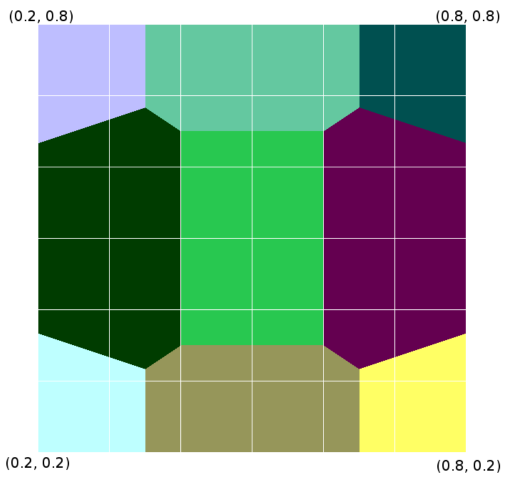
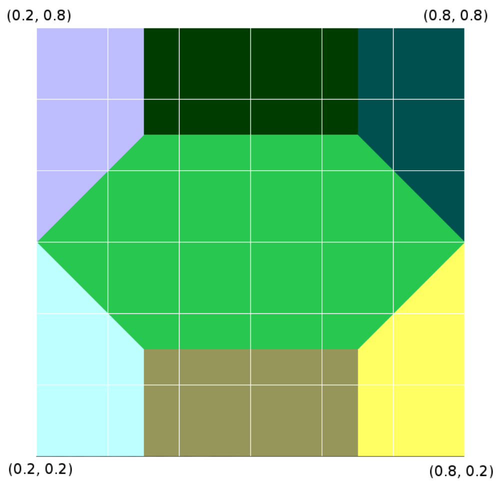


Figure 6. : This key shows what each color represents in the Terrain Boundary Map.



(a) Our default Terrain Boundary Map.



(b) An alternate Terrain Boundary Map.

Figure 7. : This figures illustrates complete Terrain Boundary Maps. Temperature is on the X axis and humidity on the Y axis.

The types of terrain we used can be seen in Figure 6. Most of the names are stand-alone, but the three different types of forest are: Coniferous Forest (forestC), Rain Forest (forestR), and Deciduous Forest (forestD). However decided, the *Points of Terrain* are populated as in step 1. The terrain boundary map is made in a two dimensional range from [0.0, 1.0) to represent the minimum and maximum values of humidity and temperature, step 2.

We reserve values less than 0.2 and greater than 0.8 for “extreme” climate situations, or a buffer zone, and avoid their direct use in map generation. The developer can then set locations within the range and give that specific coordinate value a particular climate name, and color for use in visualization. The developer is free to assign as many terrain types as desired, but two or less terrain types might produce a rather lackluster environment. Step 3 is then employed to place these POTs into the grid. After the desired terrain types are assigned, we use a nearest-neighbor algorithm to determine the range of humidity and temperature ratings that “belong” to that type of terrain, step 4. Optional noise can be applied at this step as well, but we disabled it for clarity of terrain boundary mapping; which is illustrated in Figure 7a. This is an example of the default map we generated, but another equally viable boundary map is seen in Figure 7b, which we created as a proof of concept for our results in the *Results* Section.

Final Mapping

This mapping of tuples ascribing humidity and temperature coordinates to a particular terrain is then used in the actual map to assign appropriate tiles in the game. The algorithm can be seen in Figure 8. For each of the tiles in the initial climate map, described in the *Climate Map Construction* Section, step 1, obtain the tuple values of (temperature, humidity), step 2. This tuple value is then used as a lookup value, step 3, into the Terrain Boundary Map, as described in the *Terrain Boundary Map Construction* Section. The terrain found at the specific coordinates is selected by this lookup index of (temperature, humidity), step 4. This terrain is then used to fill the final map’s

```
1. For all tiles in climate map:
2.  find (humidity,temperature) values
3.  Use (h,t) as index into terrain
    boundary map
4.  Obtain terrain type tt
5.  use tt in final world map at same (x, y)
    coordinate as the corresponding tile
    in climate map
```

Figure 8. : Algorithm outline to combine climate and terrain maps into a final world map

corresponding tile, step 5. This process is also illustrated in Figure 9.

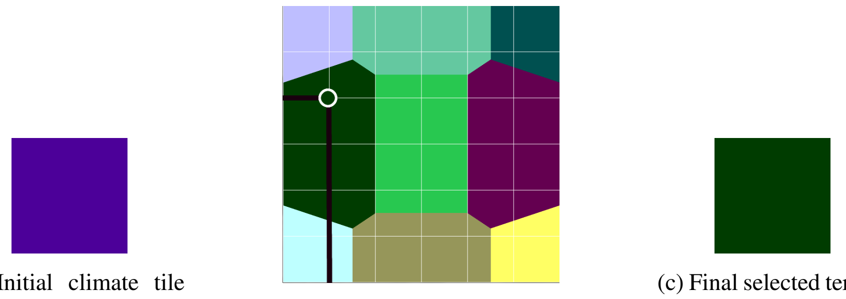
Results

In this section, we describe the results of our prototype implementation, written in Python and Pygame (Rossum, 2001; Shinners, 2011), of a procedurally generated map that incorporates smooth weather transitions between different locations of interest within the map. We include screen captures of some interesting maps that we have generated including a map that, using our approach, is similar to a map generated in Final Fantasy IV, V, or VI.

The results we obtained allows us to create a consistent and coherent world based on locations of interest influencing the surrounding humidity and temperature to dictate a specific terrain type. The locations of interest can be user-defined and hardcoded, random, or procedurally generated as well (Matthews & Malloy, 2011). Because of the interpolation we are guaranteed a gradual blend between two locations' climates.

These terrain types can then be applied to any sort of physical terrain layout, two dimensional or three dimensional. The selections of terrain types are also user-adaptable for worlds with unique terrains.

Figure 10 shows a randomized result for the system. The difference between Figure 10a and Figure 10b is the addition of noise to allow for a more organic terrain boundaries, as seen in the



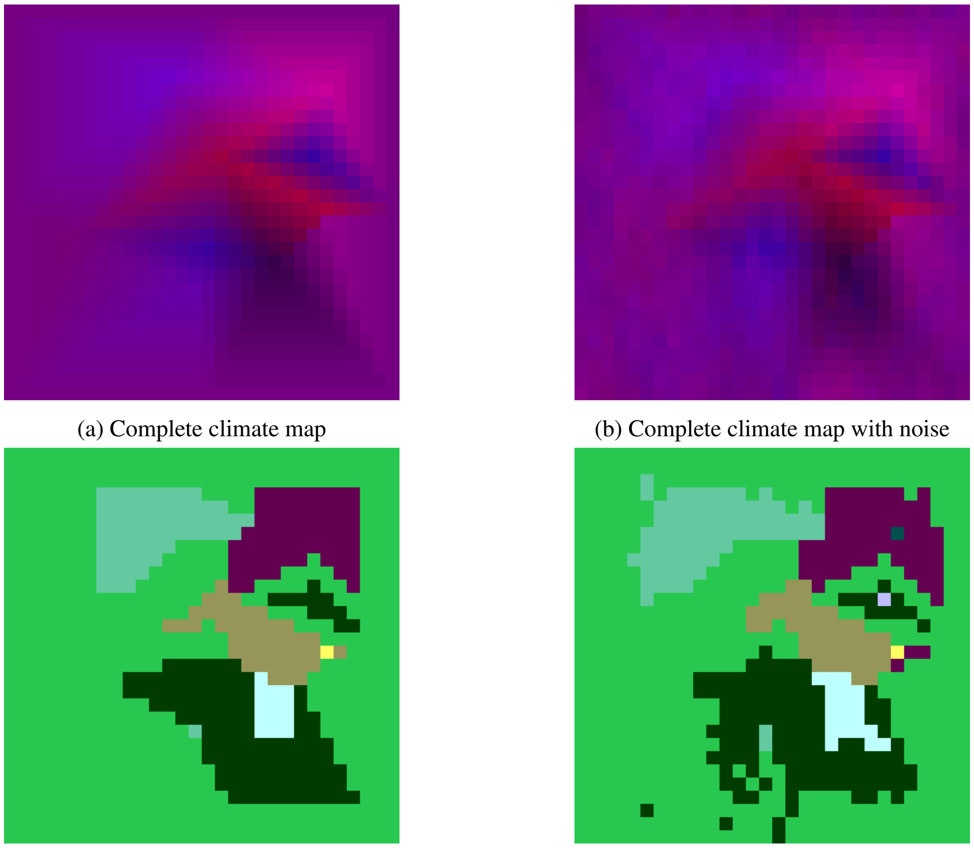
(a) Initial climate tile with t=0.3 and h=0.6 from the climate map.

(b) Looking up 0.3,0.6 in the Terrain Boundary Map

(c) Final selected terrain tile to be set in the final map.

Figure 9. : An example for a single tile creation for the final map.

differences between Figure 10c, which would be the result without noise, and Figure 10d, which is created from the noisy climate map.


(a) Complete climate map

(b) Complete climate map with noise

(c) Final result map

(d) Final result map with noise

Figure 10. : This figure illustrates the transition from temperature and humidity to a terrain type.

Figure 11 displays the capability of this system to re-create existing maps, particularly the Narshe/Figaro area from Final Fantasy VI as seen in Figure 11a. The climates available for this game aren't as numerous as the list we populated for the testing of the system, so another climate boundary map was generated, Figure 7b. To do this test, an estimation of the location and climate were hard-coded into the system. Using the actual map from FFVI, provided in Figure 11a, the Locations of Interest are as follows: Narshe, seen in the northern most part of the map, was estimated to be at coordinate location *(200, 13)*. The climate of this particular town is snowy, cold and wet, so a climate temperature/humidity value of *(250, 700)* was assigned. Figaro Castle, seen near the middle, is a castle located in a vast desert. A coordinate location of *(95, 240)* with the climate value *(800, 200)* was estimated. Cave of Figaro is the lower left location and is a cave located in a grasslands area, so location *(140, 333)* and climate *(500, 500)*. Two more subtle LOI are also included in this area: directly below Narshe is a section of plains and forest. Because this is where the player begins their first world map walk, these two seemingly arbitrary bits of terrain are actually LOI in the fact that their purpose is for the player to experience different monsters appearing in different terrain; a feature of Final Fantasy VI's game mechanics. So, when creating the LOI to re-create the same experience for the player, *five* LOI were created instead of the obvious three. The final two LOI were created at *(180, 150)* and *(300, 105)* with climate values of *(500, 750)* and *(500, 250)* respectively. Once these LOI were populated, the system proceeded and the resultant map created by our system can be seen in Figure 11b, as well as an overlay in Figure 11c to show the validity of the result.

The importance of the recreation shown in Figure 11b is to show that, given only the LOI to influence a map and changing nothing else, the same layout of the world section where the LOI were obtained can be procedurally generated. This is vital because we intend to use this system for recreating a similar, but unique, area to facilitate the replayability of a story-driven game. When recreating this area, using the same LOI but with different coordinates, our proof of concept implementation demonstrates that the player can enjoy a similar experience as the original game's map with coherent terrain. Using new coordinates for the Locations of Interest, two new maps, seen in Figure 12a and 12b, are created. The map in Figure 12a differs only slightly from the original layout of the LOI taken from Final Fantasy VI, while Figure 12b shows that a completely unique layout still generates a coherent map. The work that we present here will be exploited to enhance our earlier research so that the procedural maps generated with our previous system will now be marked by a high degree of coherence in terrain (Matthews & Malloy, 2011).

Concluding Remarks

We have described a system for procedurally generating maps for story-driven games, where locations in the map are assigned algorithmically or by designer preference. We also described a technique for generating terrain, together with climate to match the terrain, with smooth, coherent transitions between terrain exhibiting different weather. We have also described our implementation, written in Python and Pygame, that illustrates our construction with some procedurally generated maps, including the procedural generation of the Narshe/Figaro area from Final Fantasy VI.

References

Andrew Doull. (2011). `http://pcg.wikidot.com/category-pcg-games`.

Bedigian, L. (2012). Call of duty's $1 billion milestone: Monumental success or avatar-sized hype. `http://www.benzinga.com/trading-ideas/long-ideas/11/12/2198163/call-of-dutys-1-billion-milestone-monumental-success-or-avatar`.

Blizzard Entertainment. (1996, November). `http://diablo2.diablowiki.net/Diablo\_Levels`.

Dormans, J. (2010). Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 workshop on procedural content generation in games* (pp. 1:1–1:8). New York, NY, USA: ACM.

Gee, J. P. (2005). Learning by design: good video games as learning machines. *E-Learning*, *2*(1).

Hartsook, K., Zook, A., Das, S., & Riedl, M. O. (2011). Toward supporting stories with procedurally generated game worlds. In S.-B. Cho, S. M. Lucas, & P. Hingston (Eds.), *Cig* (p. 297-304). IEEE.

Matthews, E. A., & Malloy, B. A. (2011, July). Procedural generation of story-driven maps. In *Proceedings of computer games: Ai, animation, multimedia, education and serious games* (pp. 13–15). Louisville, USA.

Perlin, K. (2002). Improving noise. *ACM Trans. Graph.*, 681-682.

Rossum, G. van. (2001). *Python library reference*. Python Software Foundation.

Shaker, N., Togelius, J., Yannakakis, G. N., Weber, B. G., Shimizu, T., Hashiyama, T., et al. (2011). The 2010 mario ai championship: Level generation track. *IEEE Trans. Comput. Intellig. and AI in Games*, *3*(4), 332-347.

Shaker, N., Yannakakis, G. N., & Togelius, J. (2012). Towards player-driven procedural content generation. In *Proceedings of the 9th conference on computing frontiers* (pp. 237–240). New York, NY, USA: ACM.

Shinners, P. (2011). *Pygame.* `http://pygame.org/`.

Smith, G., Whitehead, J., Mateas, M., Treanor, M., March, J., & Cha, M. (2011). Launchpad: A rhythm-based level generator for 2-d platformers. *IEEE Trans. Comput. Intellig. and AI in Games*, *3*(1), 1-16.

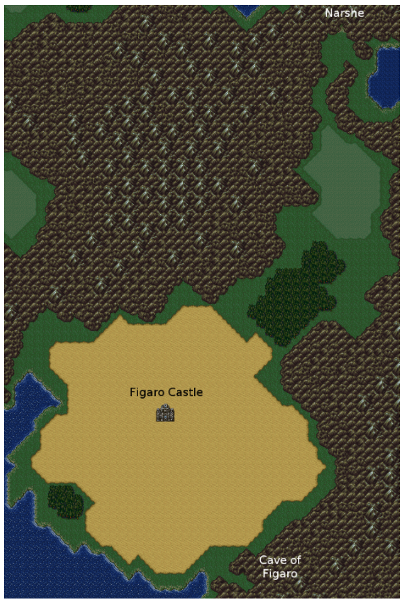
Sterling, J. (2012). *Skyrims success inspires remedy to make story-driven games.* Available from `http://www.gamefront.com/skyrims-success-inspires-remedy-to-make-story-driven-games/`

Vara, C. F., & Osterweil, S. (2010). Adventure games design: Insight and sense-making. In *Meaningful play 2010.*

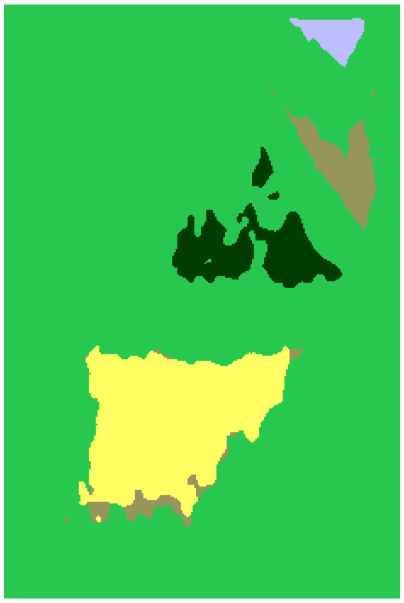
Wikipedia. (2012). *Level (video gaming).* Available from `http://en.wikipedia.org/wiki/Level\_(video\_gaming)`

Yannakakis, G. N., & Togelius, J. (2011). Experience-driven procedural content generation. *IEEE Transactions on Affective Computing*, *2*, 147-161.

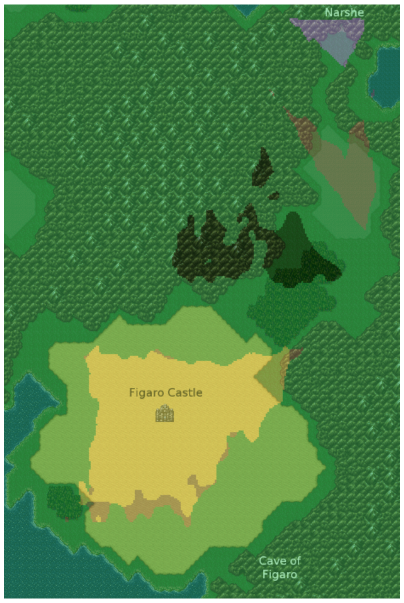
Yin-Poole, W. (2012). *Skyrim success motivates remedy to stick with story-heavy games.* Available from `http://www.eurogamer.net/articles/2012-02-24-skyrim-success-motivates-remedy-to-stick-with-story-heavy-games`



(a) Source map from Final Fantasy VI

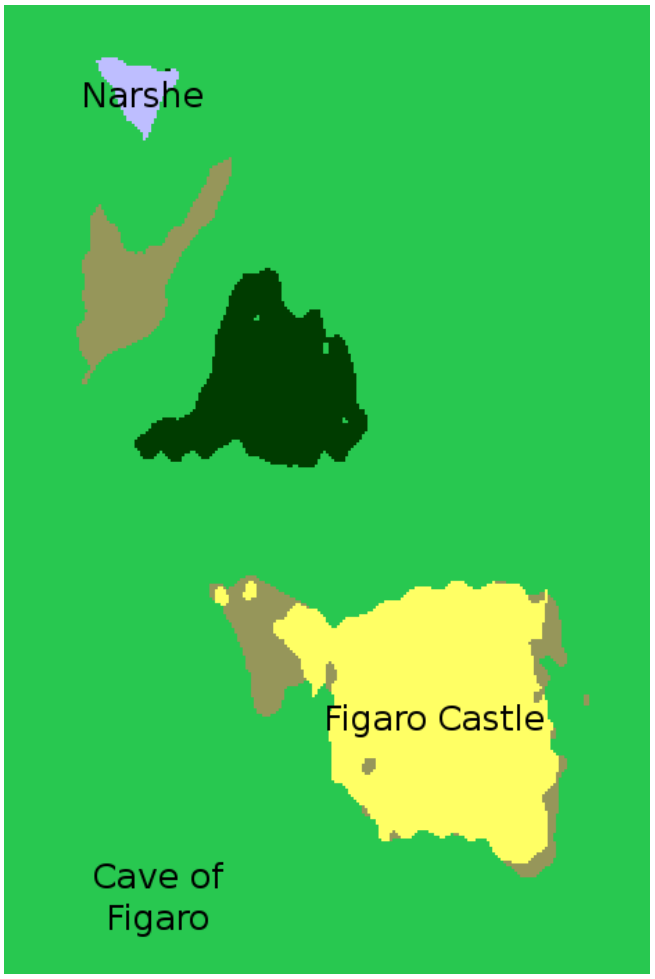


(b) Coherent result from direct estimated coordinates.

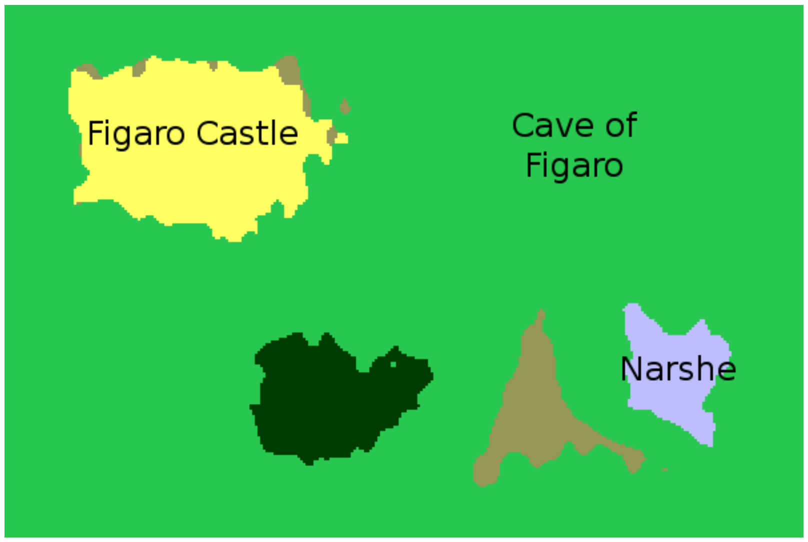


(c) Overlay

Figure 11. : This figure illustrates the capability of recreating existing layouts.



(a) LOI from Final Fantasy VI in a similar arrangement.



(b) LOI from Final Fantasy VI in a different arrangement.

Figure 12. : Alternate applications of the LOI from Final Fantasy VI.