Tile-based Method for Procedural Content Generation

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

David Maung

Graduate Program in Computer Science and Engineering

The Ohio State University

2016

Dissertation Committee:

Roger Crawfis, Advisor; Srinivasan Parthasarathy; Kannan Srinivasan; Ken Supowit

Abstract

Procedural content generation for video games (PCGG) is a growing field due to its benefits of reducing development costs and adding replayability. While there are many different approaches to PCGG, I developed a body of research around tile-based approaches. Tiles are versatile and can be used for materials, 2D game content, or 3D game content. They may be seamless such that a game player cannot perceive that game content was created with tiles. Tile-based approaches allow localized content and semantics while being able to generate infinite worlds. Using techniques such as aperiodic tiling and spatially varying tiling, we can guarantee these infinite worlds are rich playable experiences. My research into tile-based PCGG has led to results in four areas: 1) development of a tile-based framework for PCGG, 2) development of tile-based bandwidth limited noise, 3) development of a complete tile-based game, and 4) application of formal languages to generation and evaluation models in PCGG.

Vita

2009	B.S. Computer Science, San Diego State University, San Diego, CA
2010	Graduate Research Associate, Advanced Center for Computer Art and Design, The Ohio State University, Columbus, OH
2012-2013	Intern, Battelle Research Institute, Columbus, OH
2013-2014	Graduate Research Associate, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH
2015-2016	Graduate Teaching Associate, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH

Publications

Maung, D.; Crawfis, R. “Applying Formal Picture Languages to Procedural Content Generation”, CGAMES 2015. (IEEE TCIM Best Paper)

Maung, D.; Crawfis, R.; Gauthier, L.; Worthen-Chaudhari, L.; Lowes, L.; Borstad, A.; McPherson, R.; Grealy, J.; Adams, J. "Development of Recovery Rapids - A Game for Cost Effective Stroke Therapy", Foundations of Digital Games (FDG), 2014, 3-5 April

Maung, D.; Crawfis, Roger; Gauthier, Lynne V.; Worthen-Chaudhari, Lise; Lowes, Linda P.; Borstad, Alex; McPherson, Ryan J.; "Games for therapy: Defining a grammar and implementation for the recognition of therapeutic gestures", Foundations of Digital Games (FDG), 2013, 14-17 May

Maung, D.; Shi, Y.; Crawfis, R.; "Procedural Textures Using Tilings With Perlin Noise", Computer Games (CGAMES), 2012 17th International Conference on , 30 July-2 Aug 2012

Liang, J.; Fuhry, D.; **Maung, D**; Crawfis, R; Gauthier, L; Nandi, A; Parthasarathy, S. “Data Analytics Framework for A Game-based Rehabilitation System”, SDM Workshop on Data Mining, Vancouver, CA. April 2015.

Gauthier, L., Borstad, A., Lowes, L., Worthen-Chaudhari, L., Crawfis, R., Jaffe, J., & **Maung, D.** (2014). Delivery of Constraint-Induced Movement Therapy Through a Video Game: a Pilot Study in Stroke. Archives of Physical Medicine and Rehabilitation, 10(95), e8-e9.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

While tile-based games have existed for a very long time, these are often hand-crafted tiles and tilings. Recently, games have started using Procedural Content Generation (PCG) to build tilings using artist created tiles. This thesis develops a software framework and explores its usage in a new area we call Procedural Content Generation for Games (PCGG). We define PCGG as the algorithmical creation of game content with limited or indirect user input [71]. Content can be levels, maps, game rules, textures, stories, items, quests, music, weapons, vehicles, characters, etc. Games add to PCG rules and metrics to create tilings with a purpose.

Recently, much research has been done in the industry, including the Procedural Content Generation Workshop often hosted simultaneously with the Foundation for Digital Games conference [46]. Procedural content generation itself started independently of video games, most notably in SIGGRAPH which has many papers on algorithms for the procedural models of plants, terrains, clouds, and textures in general. Video games place requirements on PCG such as fair gameplay, fitting a game theme, avoiding boring repetition, etc. These requirements develop an interesting environment in which PCGG may be evaluated as to its quality or suitability for a specific game.

This thesis on tile-based PCGG focuses on four specific areas. First, we wanted a common software foundation for our research. We accomplished this by generating a

framework for tile-based PCGG. In Chapter 3, we present our framework along with a detailed case study involving a terrain generated for a racing game.

Second, we describe a method of generating tile-based bandwidth limited noise based on Perlin Noise and using Wang tiling. A fundamental tool in PCGG is bandwidth limited noise introduced by Ken Perlin in his paper An Image Synthesizer [1]. Perlin's version is a spatial function and can't be cut into tiles for use in Wang tiling. If you naively cut it along a tile edge to generate a tile set, it is not obvious how you could generate a tiling without having visible seams. Yinxuan Shi and I developed methods of generating bandwidth limited noise tiles such that we could generate seamless tilings and maintain aperiodicity. We discuss our results in Chapter 4.

Third, we develop a complete tile-based game. During the last three years, I have had the opportunity to participate on an interdisciplinary team to develop a serious game for the treatment of hemiparesis (one sided paralysis due to stroke or other brain injury). The problem given was to gamify Constraint Induced Movement (CI) therapy, the gold standard for rehabilitation of hemiparesis [28]. CI therapy requires participants to be actively involved for 30 hours during the course of a 2 week treatment. Development of 30 hours of entertaining game content by artists was not within our budget; we saw this as a perfect opportunity and platform for PCGG. Our team has developed a game with 2 distinct tile-based game environments uniquely generated each lap. Each lap lasts approximately 15 minutes and about 120 laps are presented to the participant during the course of treatment. This work is detailed in Chapters 5 and 6.

Fourth, we examine formal methods, 2D Regular Expressions (2RE) and Array Grammars (AG) for the generation of tilings. Without refinement the search space for tile-based PCGG is intractable and we look to these methods to help refine it.

As an example, consider generating a maze in a 20 x 20 grid where the edges of each square are either open or closed. The player's goal is to move from the center of the starting square to the center of the ending square, traversing only open edges. An example of a metric function might be the length of the solution path measured in number of squares. Search based approaches are able to find a reasonable solution; however they are not guaranteed to find an optimal solution. The problem is the search space can be huge and the number of optimal solutions small. A fellow graduate student, Paul Kim, generated ~23 million mazes of the kind mentioned for his research and measured the length of the solution path as a percentage of the number of cells in the maze [62]. We know an optimal solution is a labyrinth that encompass 100% of the squares, yet the highest percentage his search could obtain is 70.5%.

Another approach may be to try and enumerate all possible mazes. A perfect maze in a grid is a spanning tree. We know from the work of Wu [61] that the number of spanning trees in a grid is e^{ZN} where $Z = \frac{4}{\pi}(1.1662436)$. By this calculation, a 20 x 20 grid has $\sim e^{593}$ mazes. Using a modern computer, one might expect to enumerate 1 million mazes per second per processor. At that rate, it is infeasible to enumerate all mazes even with cloud based computing.

This example shows the importance of reducing the search space. 2RE and AG are designed to work with 2D arrays and a finite alphabet of elements such as tiles. They offer the potential to help in both reducing and enumerating the search space. 2RE and AG are discussed in Chapter 7.

Chapter 2. Related Work

PCGG can be divided by the scope of the content. On the broad end is PCGG which covers a large terrain or whole world. The middle level is PCGG for game or dungeon levels. At the smallest scope is PCG for individual game components like cities, plants, or buildings.

At the terrain level, a common technique for generating height maps or even fictitious worlds is the use of gradient noise such as Perlin noise [1]. Although Perlin did not give an example in his paper, it was readily apparent that gradient noise could be used to generate height map terrains. Another technique for terrain generation is fractal methods. In these methods, a surface polygon is subdivided into smaller polygons and the vertices of the smaller polygons are perturbed by a random amount. Gavin Miller describes the concept of fractal terrains in [47]. A third approach is physically based modeling of natural systems in an attempt at more realistic results. Musgrave introduces the concept of eroding a fractal terrain for more realism [48]. Many papers followed that provided more detailed hydraulic models, considered the type of ground being eroded, and implemented parallel algorithms for faster simulations.

While the above techniques add realism to terrain, they are not easily adaptable for use in video games, where designers want more control over the landscape generated for gameplay purposes. For example, a game designer may seek to embed sniper positions

or choke points into a terrain for an FPS game. In these cases other approaches which are not physically based are of useful.

Doran used an agent-based approach for the generation of landscapes [49]. He created 5 distinct types of agents: coastline agents, beach agents, smoothing agents, mountain agents, and river agents. To each of these agents, he added parameters which control the type of terrain generated.

Genetic algorithms are a promising approach to PCGG. In genetic algorithms, terrain (or game level) representations are called genotypes. A collection of randomly selected terrains is generated and a metric function representing the terrain quality is calculated. The population of terrains is culled with the poorest performing terrains being eliminated. New terrains are then created by "breeding" (or crossover) and mutation. In breeding, 2 terrains from the existing population are used to produce an offspring. Some of the genotype is taken from each "parent" and spliced together. In mutation, the genotype is randomly changed in some small way.

An example of this is the work of Frade. He introduced two metric functions, namely accessibility [50] and obstacle length [51]. Accessibility is a measure of the largest contiguous accessible area in a terrain. This is calculated by determining the slope at each point in a heightmap and labeling points above a threshold inaccessible and those below the threshold accessible. Accessibility is calculated as the largest contiguous accessible area relative to the heightmap size. His second metric function, obstacle edge length, is obtained from the accessibility map by summing all cells which are on the

boundary of an accessible area. Frade claims increasing this metric makes for more interesting terrains because it increases the complexity of accessible areas.

Another approach, interesting because of its focus on the designer instead of a metric function, is offered by Zhou [52]. He incorporated real world terrain patches and an artist sketch as input to synthesize a terrain. Figure 1 shows a terrain synthesized from a user sketch of the Half Life symbol.

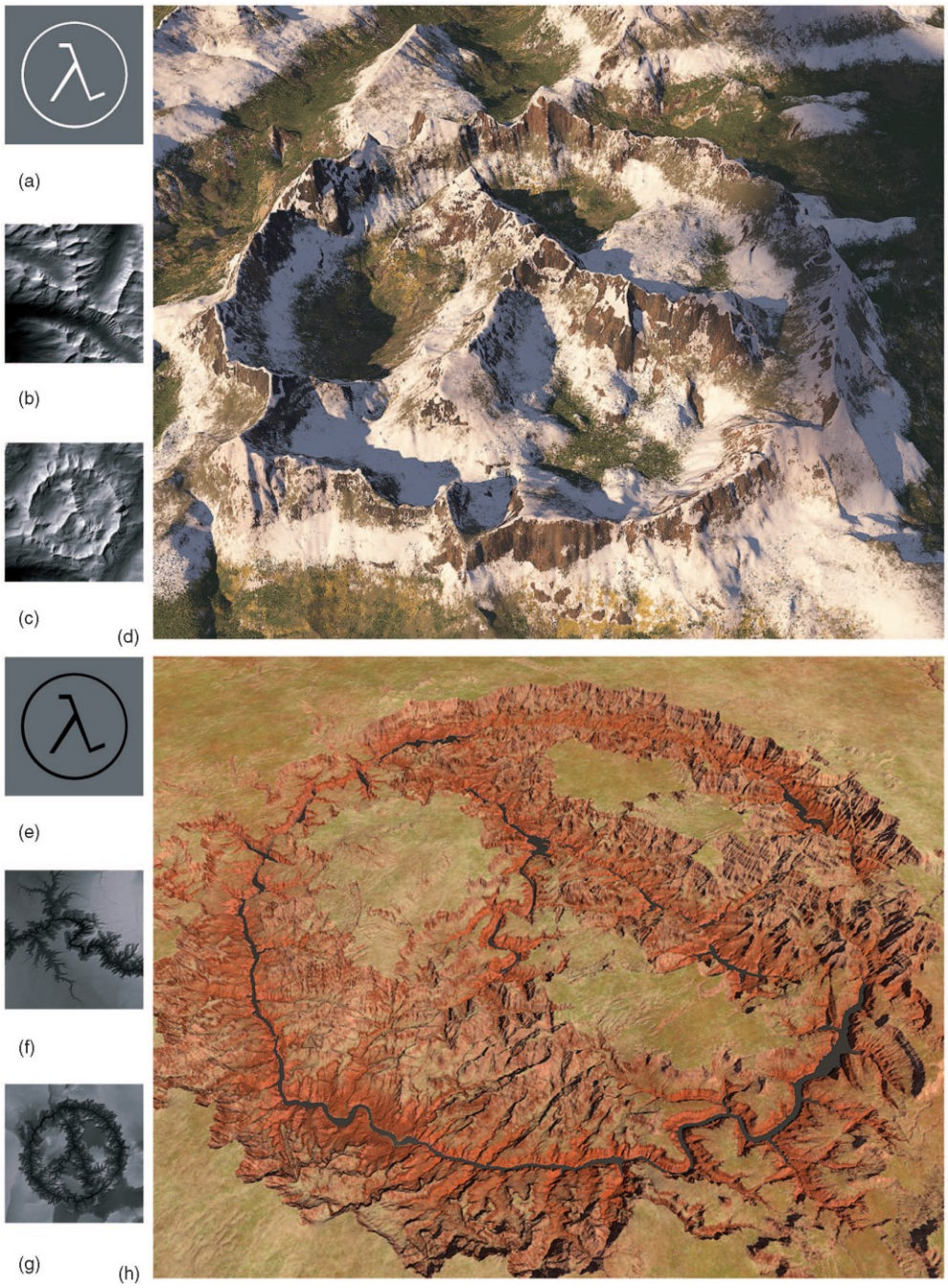


Figure 1. Zhou's method.

An interesting technique which is not based on a standard height map is Lengyel's voxel based terrain [53] . Lengyel represents terrain as a scalar function in 3D instead of a height map. He calls it a *voxel map*. Locations where the scalar function is positive are empty space and defined as outside the terrain. Locations where the terrain is negative are inside the terrain. The isosurface where the scalar function is zero is the surface of the terrain. A modified marching cubes algorithm [63] is used to triangulate the terrain surface for viewing. The advantage of this technique is that it can generate terrains with overhangs and caves, which is impossible with a standard height map.

Figure 2. Voxel-based terrain.

As opposed to terrain generation which focuses on large areas such as playable outdoor game maps and game worlds, level generation is concerned with smaller areas such as an individual game scenario. There is usually more focus on quality measures such as playability, game balance, drama, and being able to match a story or quest. Some specific examples of note follow.

Dormans divides the generation into two parts: missions and spaces [44]. First, he uses a grammar to procedurally generate a random mission from a set of rules. Then, he uses a shape grammar to adapt a game level to fit a mission.

An example of using genetic algorithms for platform games is given by Mourato [54]. Mourato uses the game Prince of Persia as his example platformer. The level is divided into cells (tiles) which are considered the genes. During mutation, multiple cells are randomly substituted. Crossover involves taking contiguous regions from each parent and splicing them together.

Johnson discusses using cellular automata to generate cave-like levels for adventure games [55]. The method begins with a randomly initialized 50 x 50 grid of binary values which indicate either rock or open space. The cellular automata are allowed to run on the grid for a fixed number of iterations. The result is a game level with cave-like constructions. Although simplistic, this technique is interesting because cellular automata can be shown to be similar to other formal methods such as 2D regular expressions. Figure 3 shows an example of generating a cave using cellular automata. The left indicates a random initial map. The right is the resulting cave level. Red and white indicate solid rock while other colors indicate contiguous open space.

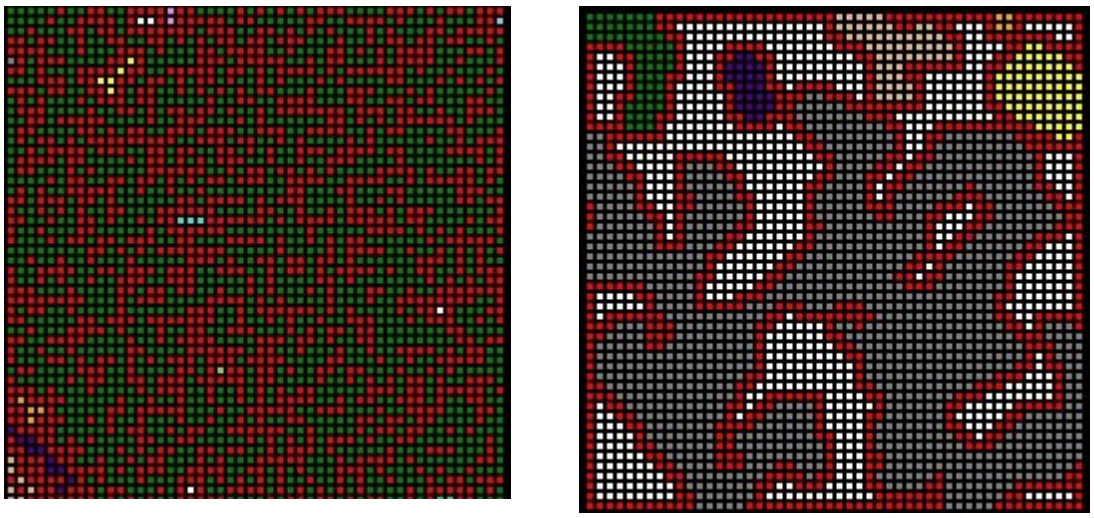


Figure 3. Caves generated using cellular automata.

Matt Boggus develops a body of research about the generation and illumination of cave levels for video games [70]. His cave modeling approach is based on creating caves which mimic 2 types of naturally occurring solution caves (caves formed by water): phreatic caves and vadose caves. He develops a mathematical model for the cross section of each type of cave and then sweeps along either an artist or procedurally generated control curve. An example phreatic passage generated with this technique is shown in Figure 4 and an example vadose passage is shown in Figure 5

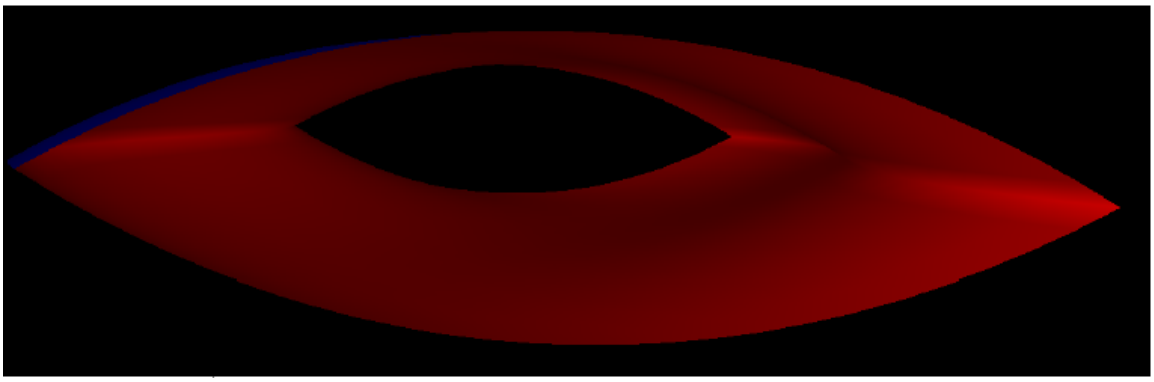


Figure 4. Model of a phreatic cave passage generated by Matt Boggus.

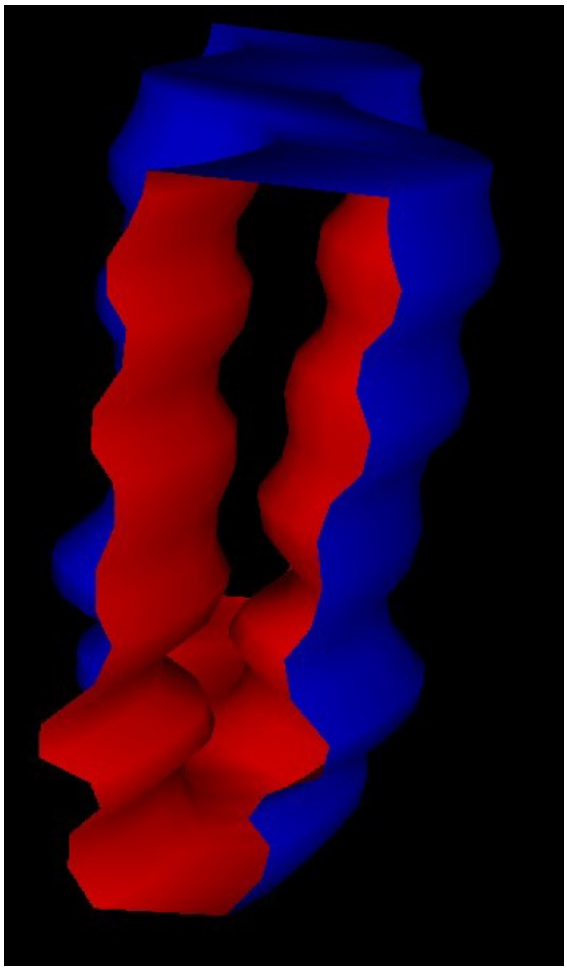


Figure 5. Vadose cave passage generated by Matt Boggus.

Hullett approaches the generation of first person shooter (FPS) levels by borrowing the engineering concept of design patterns [56]. He conjectures that all FPS games will have similarities such as choke points, sniper locations, and arenas; and that each of these can be well defined as to its in-game purpose, uses, and relationships. They can then be used as a toolbox for the level designer.

Another novel approach to content generation introduced by Smith is the concept of generating game levels based on rhythm of play [41]. In this work a generic platformer is built by first designing the rhythm of the game based on user actions such as jump, run, and shoot. The rhythm elements are then converted to game geometry which requires the player to achieve the target rhythm to complete the level.

Although this dissertation is specifically based on PCGG, there are a couple of noteworthy examples of general PCG which can be used by games. First, Hanan introduces using L-systems for the generation of plants [34]. L-systems are string rewriting grammars which, like fractals, tend to result in strings with a self-similar nature. These well represent the fractal nature of trees. See Figure 6.

Figure 6. Fractal weeds generated with L-systems.

Second, Silva uses shape grammars for the construction of buildings [45]; however, he hides this grammar in a visual editing tool which presents the grammar choices as a set of graph nodes with directed edges. Each node takes its input (a non-terminal) and performs some operation on it, resulting in an output. An edge in the graph indicates that this output can then be used as an input for another node.

Finally, I summarize the structure of the rest of this dissertation. In Chapter 3 I build a foundation of tile-based PCGG by discussing our work on a framework. Chapter 4 follows with a description of the work on aperiodic noise done by Yinxuan Shi and myself. In Chapters 5 and 6, I detail the development of Recovery Rapids, a game for the

rehabilitation of hemiparesis, done as an interdisciplinary project by a team of 7 individuals:

- Lynne Gauthier, PhD. Assistant Professor of Physical Medicine and Rehabilitation and Principal Investigator
- Roger Crawfis, PhD. Associate Professor of Computer Science
- Linda Lowes, PhD, PT. Nationwide Children's Hosiptal.
- Alex Borstad, PhD, PT, NCS. Assistant Professor of Research Physical Therapy
- Ryan McPherson. Lecturer Electrical and Computer Engineering.
- Lise Worthen-Chaudhari, PhD. Associate Director, Motion Analysis and Recovery Laboratory
- David Maung. Graduate Student, Computer Science

Finally, in Chapter 7, I describe my work on using regular expressions and AG as generators for PCG.

Chapter 3. A Tile-based Framework for Generating Procedural Terrains for Video Games

Introduction

To begin, we need to define a tile and a tiling a little more formally. By tiling, we mean a tessellation of space into regular polygons. A tile is a polygon in a tessellation. A tiling is regular if all the tiles are the same size and shape (i.e. congruent). A tiling is semi-regular if there is more than one set of polygons in a tiling where each member of a set is congruent to another member of the same set, but members of different sets are not congruent to each other. Tiles can vary from the simple single material tiles of Johnson's cave tiles [55] (rock or air), to very complex artist created or algorithmically created tiles (such as city block tiles used to create an in-game city). We are often interested in tiles for which we have more properties than a simple scalar value or a binary choice (such as rock or air). Thus tiles let us subdivide the problem of content generation of for a large space into the problem of content generation for a set of tiles along with a tiling algorithm. After introducing key concepts for tiles, tilings, and tile-based PCGG, we develop our framework. We also provide a sample application which demonstrates its intended implementation and use.

Tiles and Tilings

The first thing that comes to mind when you think about tiles and tiling is probably a 2D planar tiling such as a tiled floor or wall. We also consider tilings in 1D, 3D or more. A simple example of a 1D tiling is a string of dominoes where each domino half matches its neighboring domino.

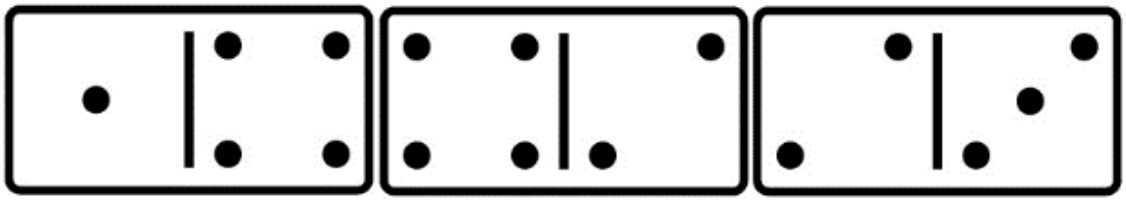


Figure 7. A 1D tiling of Dominos.

This is a tiling with a constraint for neighboring tiles. We will return to such constraints below in when we discuss Wang Tiling. 1D tilings can be useful for the creation of side scrolling games such as Super Mario Brothers.

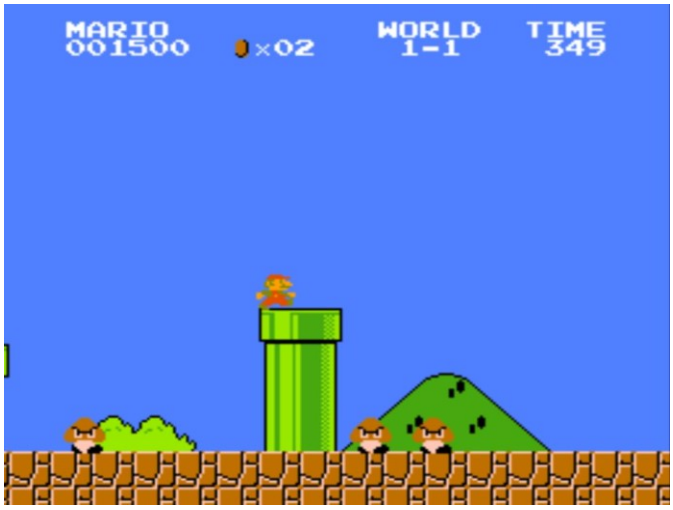


Figure 8. Super Mario Brothers[1]

Games may also consider a 3D tiling. An example game was created by students in a special course on Procedural Content Generation. In this game, 3D cube tiles were used where the edge (i.e. face) is either open or closed. A maze was embedded in the game space using this technique and the goal of the player was to solve the 3D maze. See Figure 9.

[1] Image by www.giantbomb.com.

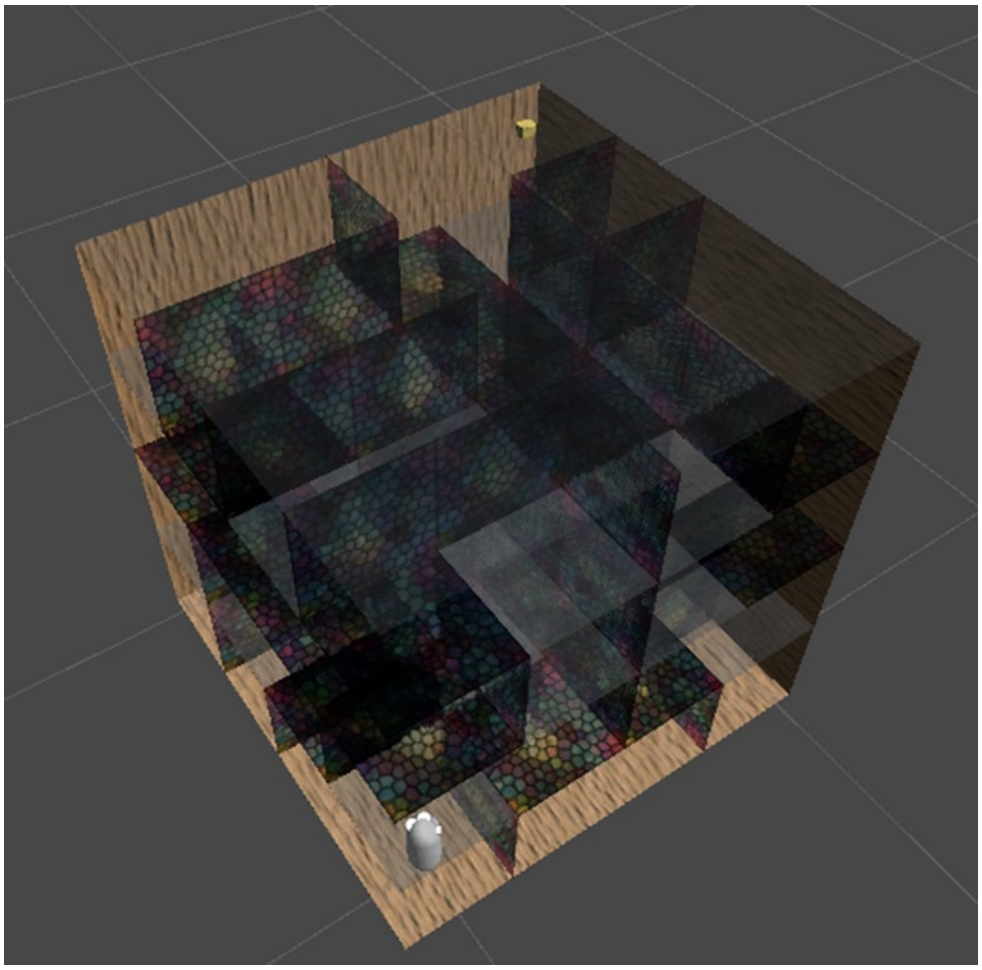


Figure 9. Example of a 3D maze build with tiles. The player is the grey capsule on the bottom and the goal is the gold cube on top.

Commonly, we discuss tiles which topologically form a 2D planar tiling whether or not the tiles themselves are 3D or 2D. An example shown in Figure 10 is a city maze for our game, Survive! This game embeds a 3D maze into a 2D tiling. Players travel vertically in the maze by climbing stairs. Dead ends on the first level occur when the path is blocked by a building. Dead ends on the second level occur when the path is either

blocked by a building or a gap that is too large to jump. The level shown is composed of a 6 x 6 tiling of city scape where each tile is composed of a central single story building surrounded by either buildings or open pathways. The detailed use case we develop below to demonstrate our framework is an example of a 2D topological tiling that builds a 3D terrain.

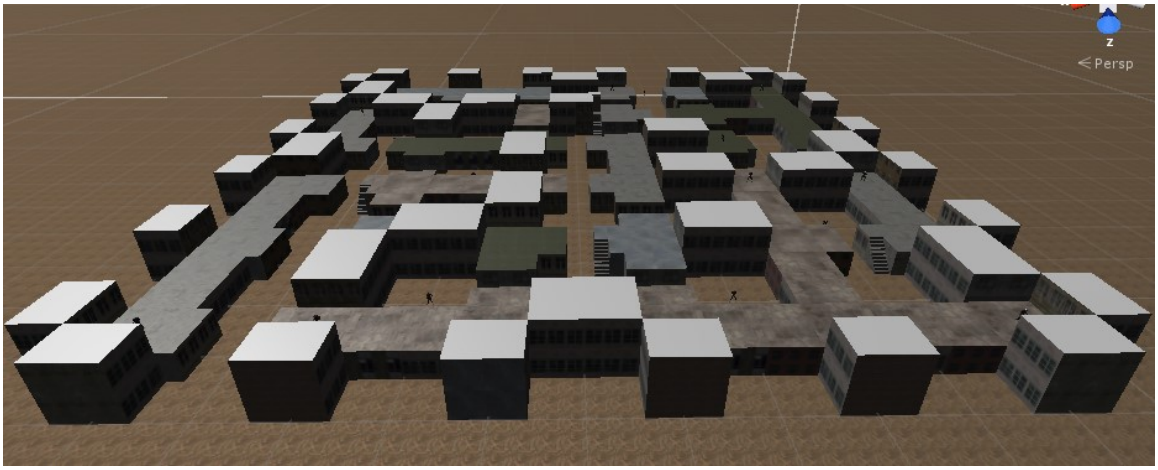


Figure 10. Survive! game constructed using a 6 x 6 tiling.

There are 3 types of regular tilings of the plane with each tile being topologically equivalent to a square, a triangle, or a hexagon as shown in Figure 11. Figure 12 shows a semi-regular tiling of octagons and squares. Most often we use tiles which are topologically equivalent to a square. Our framework design, however, should be flexible enough to support all types of regular tilings as well as semi-regular tilings (with 2 or more tile types).

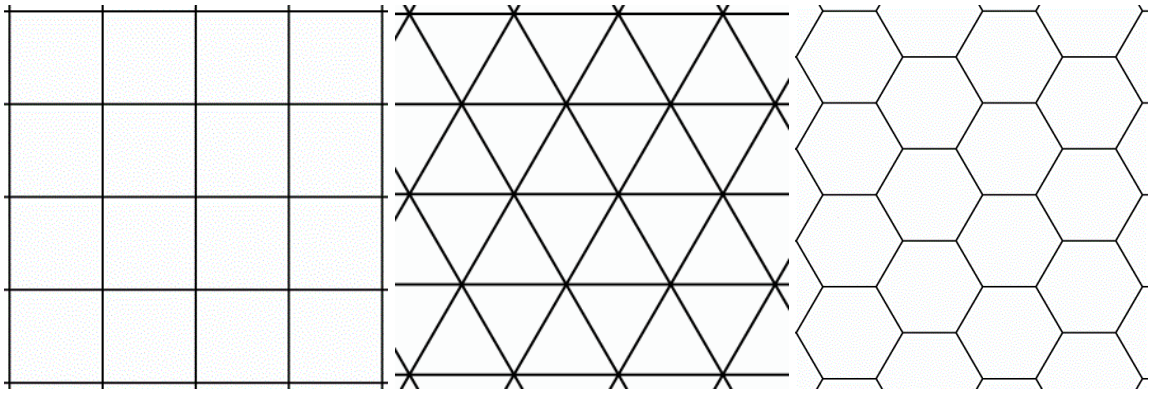


Figure 11. The 3 regular tilings of a 2D plane.

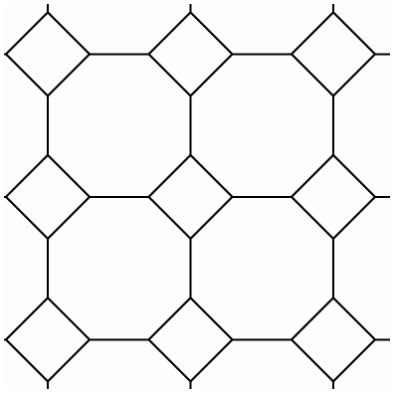


Figure 12. A semi-regular tiling using squares and octagons.

In some games, tiles may be placed randomly into a tiling with no constraints on tile placement. An example is the generation of terrain maps for role playing games such as those created by the Random Hex Terrain II terrain generator [64]. Figure 4 shows an example of a terrain generated with this tool.

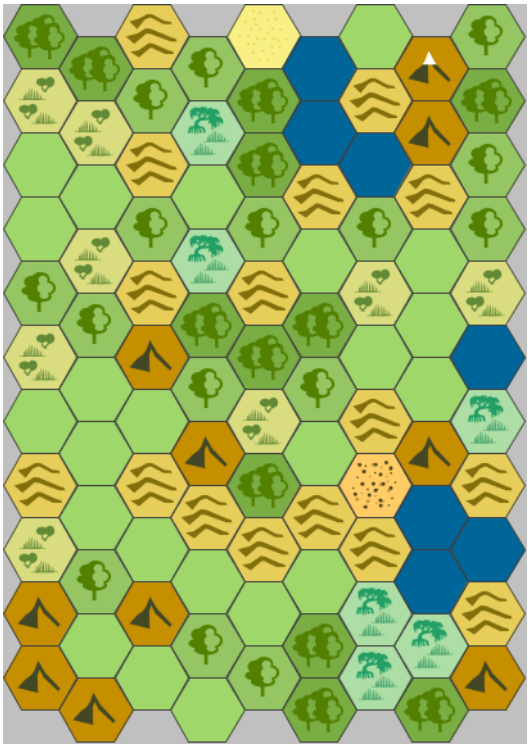


Figure 13. Randomly generated terrain for a role playing game.

More often we deal with constraints based on adjacent tiles. For example, we may use a simple set of race track tiles (shown in Figure 14) to lay out a race track with the constraint that the road must align with adjacent tiles. Figure 15 shows a race track generated in this manner. Without this constraint a tiling of race track tiles would not make sense.

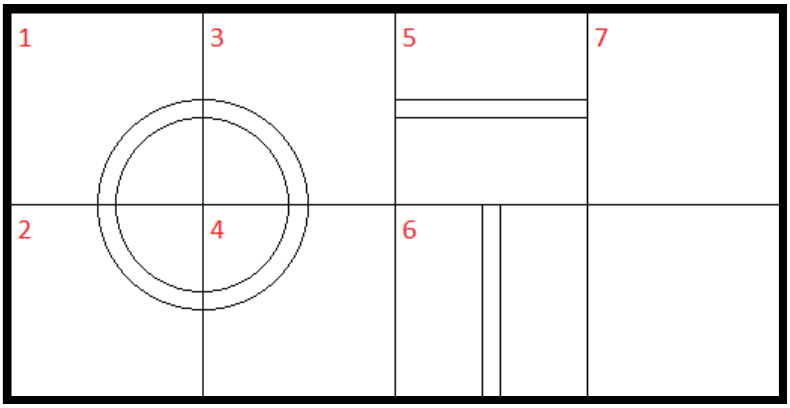


Figure 14. Race track tile set. Tiles are numbered in red.

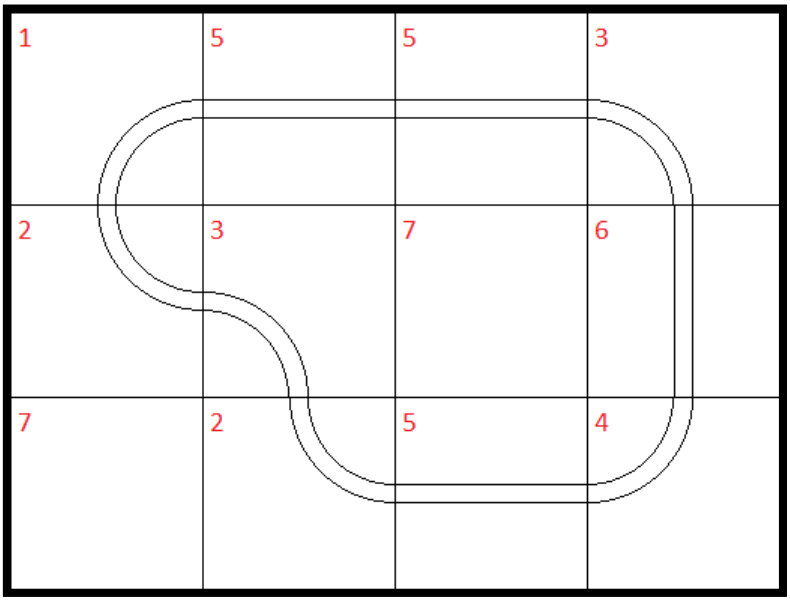


Figure 15. A race track tiling. Tiles are numbered in red.

Our adjacency constraint requires that the road align **across the adjacent edge**. The path of the road between the tile edges is not relevant for a valid tiling. To demonstrate this,

we construct another tile set which maintains this adjacency, shown in Figure 16. The race track generated using the same tiling as above is shown in Figure 17.

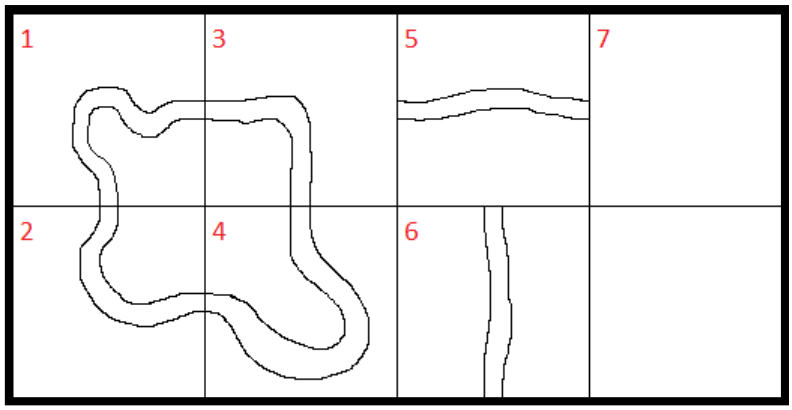


Figure 16. A second set of race track tiles.

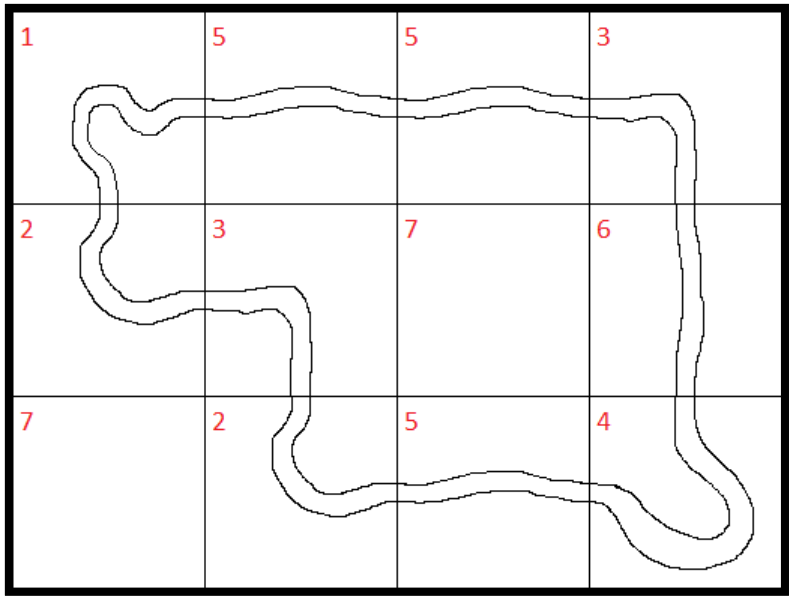


Figure 17. Race track with same tiling as Figure 15

In fact, we can use both tile sets, making a binary choice for which tile set we use each time we place a tile in the tiling. The track covers 10 tiles, so with the 2 sets, there are 2^{10} or 1024 valid race tracks we can make from this single tiling and the 2 tile sets above.

We introduce Wang tiles to formalize matching tile edges. Wang tiles are named in honor of Hao Wang who conjectured in 1961 that any set of tiles that can produce a valid tiling of the plane must also be able to produce a periodic tiling of the plane [72]. Wang originally developed a set of tiles with colored edges. A Wang tile set is shown in Figure 18 [69]. A valid Wang tiling is a tiling in which adjacent edges have the same color. A Wang tiling is shown in Figure 19 [69].

Figure 18. A set of 16 Wang tiles

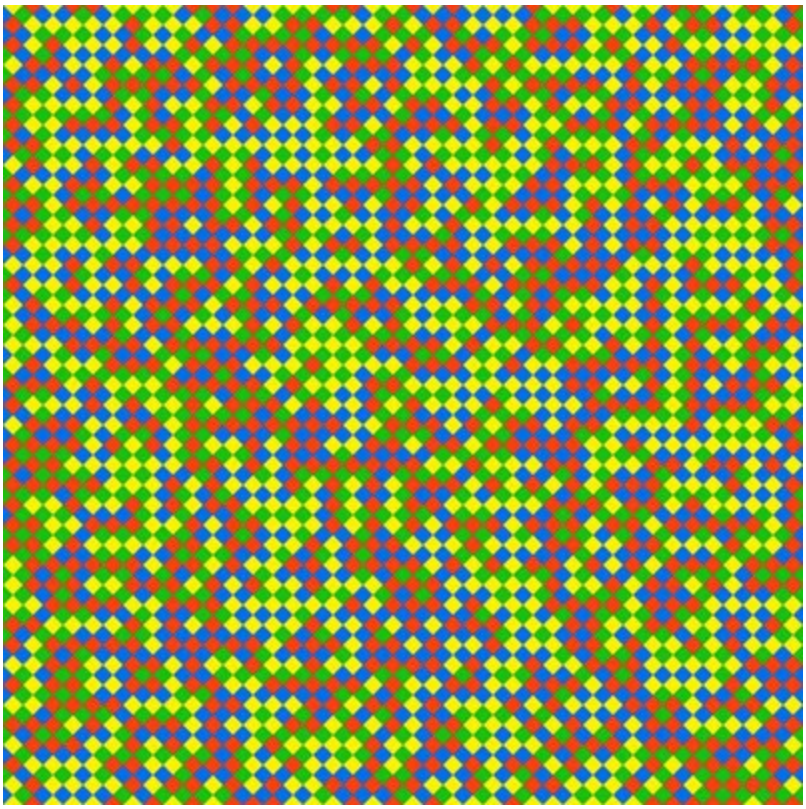


Figure 19. A 40 x 40 Wang tiling.

We extend this concept for our work. Herein, a Wang tile is a tile where each edge is defined by an N-vector of properties. These properties need not be homogenous. For example, one property might be a binary value indicating whether or not the road crosses the center of the edge (as in the above race track tiles) and another may be a vector of integers specifying the profile of the height map along the edge. A specific setting of all the properties of a tile edge is called the edge color. The number of possible edge colors is equal to the number of possible valid ways all the properties of an edge can be set.

To generate a Wang Tiling, we introduce an oracle *O(P,Q)* which takes 2 adjacent edge colors *P* and *Q* and returns a Boolean indicating whether the adjacency is valid. A valid Wang tiling is a tiling in which all adjacent edges are valid under an oracle *O*. For rectangular Wang tilings, we can have a horizontal oracle O_H and a vertical oracle O_V. This dissertation will focus on rectangular Wang tilings. We leave hexagonal and triangular Wang tilings to future work.

The most common oracle tests if *P* and *Q* are equal; however, there are some applications for other types of oracles. For example, suppose we have 2 edge colors black and white and we create a tile set with only two tiles: one all black tile and one all white tile. Valid Wang tilings under an oracle *O(P,Q)* that returns true if *P*<>*Q* and false if *P*=*Q* would all be checker board patterns like that in Figure 20.

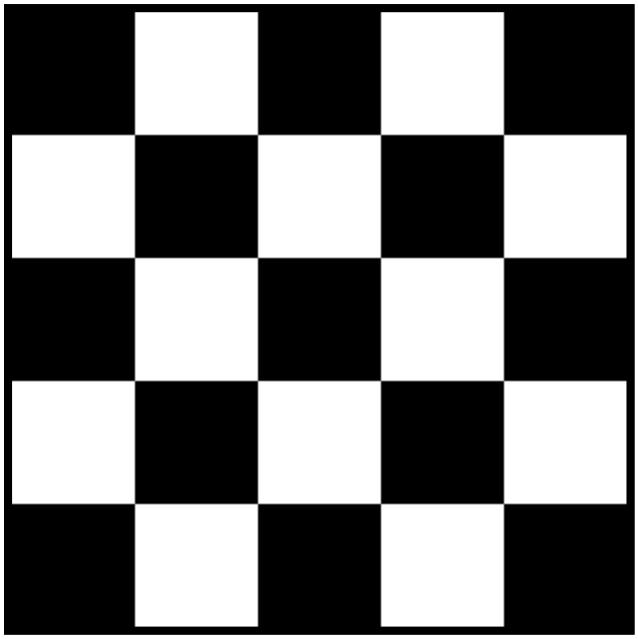


Figure 20. A valid Wang tiling under an oracle that returns true for different colors.

If we perform tiling beginning in the upper left hand corner and proceed in a top to bottom, left to right fashion (as one reads a book), we need only check at most 2 edges to maintain a valid Wang tiling. This means that if there are n colors in our tile set, we need n^2 tiles (one for each left edge color and top edge color combination) to make sure we can generate a valid Wang tiling. If we wish to be able to choose a tile by coin flip every time we lay down a new tile (thus enabling non-periodic tiling), we need $2n^2$ tiles. However, not all applications tile top down, left to right. If we wish to be able to create a valid tiling regardless what order we place tiles in, we need n^4 tiles. Similarly, we need $2n^4$ tiles to maintain a choice of tiles under any tile placement order.

In practice, we constrain the number of edge colors to maintain a reasonably sized tile set. For example, if the edge color is a vector of integers which specifies the heightmap profile along an edge, we constrain this vector to a set of a few possible choices which we define as valid.

Throughout this dissertation, beginning with the race track example above, we show how to use Wang tiling in procedural content generation. In the race track above, we show how to tile such that we have a race track that is continuous. In the more elaborate race track use case for our framework we also show how to create terrain that smoothly transitions from one tile to another and tree placement such that no 2 trees are placed next to each other. In Chapter 4 we show how to use Wang tiling to create seamless textures. Chapters 6 and 7 demonstrates generating mazes with Wang tiles.

Advantages of tile-based terrain generation.

Tile-based PCGG has several nice features. First, it conveniently subdivides the content creation process. This means it is easy to define local features. For example, tiles 1 and 4 of Figure 16 introduce sharp turns to our race track. In a mixed terrain that incorporates both artist generated content and procedurally generated content, Wang tiles provide a convenient method of separating the two. We can define either a single tile or a large area where artist content can be included. Another advantage of tiles is that they can be generated at design time or load time and used to create infinite play spaces at run time without requiring lengthy content generation. We use this feature in Recovery Rapids described in Chapter 5.

In the next section, we develop a framework to support tile-based PCGG software development. We first discuss the goals of our framework. We then develop a typical use case to help illustrate our framework in action. Finally we describe each part of our framework along with its implementation in a use case.

Tile-based framework

Requirements for a tile-based framework

Our goal is to develop a tile-based framework in support of PCGG for a diverse variety of games. We cannot anticipate all the different tile-based PCGG that will be developed; however, we would like our framework to be flexible and support as many use cases as we can. Our framework should support 1D, 2D, and 3D tile-based PCGG. It should support regular tilings and semi-regular tilings. It should support the inclusion of artist generated content along with procedural content. Often we desire to transition between

tile sets in a tiling, so it should support multi-tile sets. We also desire to support compound tilings or compositions of tilings or tiles.

Tile creation is not the only thing the framework must support. It must also support tile use during run time. Runtime support involves the following:

- Support for player/AI navigation
- Tile geometry and textures for the GPU.
- Tile models for the physics engine.
- Surface height (basic height map queries).
- Tile semantics such as intended game use.
- Terrain ecology for the placement of vegetation.

We provide a component model for the implementation of these queries.

Introducing a use case

For our use case, we provide a fully developed race track scene for a racing game. This scene supports the tile-based race track generation research being done by an undergraduate student, Jacob Haynes. In this scene, we wish to embed generated race tracks in a mountainous terrain with trees and water features. This use case gives an actual example of PCGG that assists in ongoing research and provides several real world issues that we encounter and need to solve for tile-based PCGG. Our race-track example contains water features which are specified by a designer using a sketch. This shows one valid method of designer control over terrain generation. There are many different approaches we could take to accomplish this. Here, we separate the creation of mountains, valleys, trees, water, and race track into different tilings and then combine

them into a final product. We will demonstrate that this resolves some of the combinatorial issues with having a large number of edge colors in a Wang Tiling.

Our use case will be developed in C# using the Unity3D game development API in a Microsoft Windows 10 environment. As such, our tiling(s) will need to provide several components to the game and game engine. These include textures and meshes for shaders and physics engines, and height maps and terrain ecology for tree placement. These four outputs will suffice to demonstrate our component model. A full-blown game may require several more outputs from a tiling.

Tiles

The fundamental object in our framework is the tile itself. Tiles in a game may be 1D, 2D, or 3D. In 1D, a tile has 2 edges. The number of edges or faces in 2D and 3D is dependent upon the application. We represent the tile with an interface *ITile*. *ITile* has one property, *EdgeCount*, an integer that represents the number of edges of the tile. The *ITile* interface is shown in Figure 21.

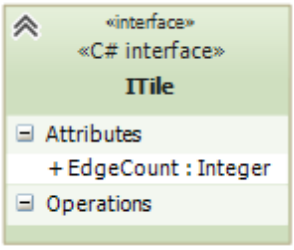


Figure 21. ITile interface

Tiles are queried for various game specific tile properties during design time and run time. Some design time examples are terrain height and terrain ecology at a given point. Some run time examples are terrain texture and terrain mesh for the GPU. In Recovery Rapids, tiles are queried for a tile semantic, "Therapy Type," which indicates the intended physical therapy to be accomplished while the player traverses the tile. We provide a component based model to deliver game specific tile properties. We include the *IComponentProvider* interface (shown in Figure 22). This interface has one generic method *T GetComponent<T> ()*. Tile developers may implement the *IComponentProvider* interface for tiles which return properties other than edge count. We provide a base class implementation of the *IComponentProvider* interface called *ComponentProviderBase* (also shown in Figure 22). This base class implementation conveniently provides a simple dictionary with add, remove, and retrieval methods. Framework users may choose to extend *ComponentProviderBase* or implement *IComponentProvider* themselves.

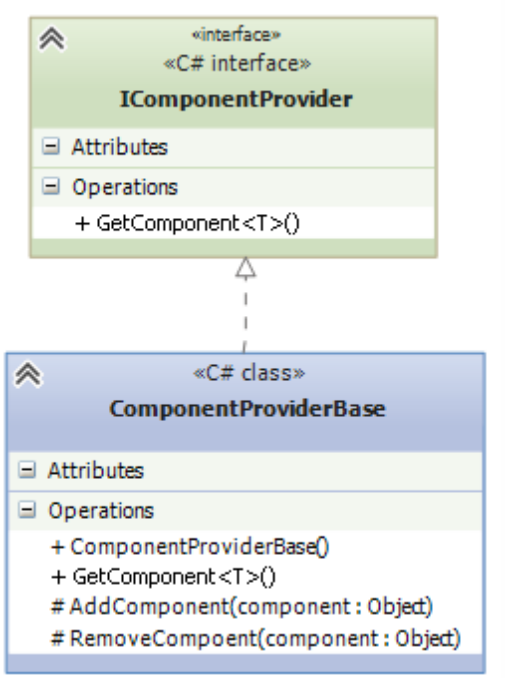


Figure 22. IComponentProvider interface and ComponentProviderBase class

Our framework supports 2D rectangular Wang tiles through the *IWang2DTile* interface. Figure 23 shows the *IWang2DTile* interface. This interface is a generic interface with two type parameters *N* and *C* for tile edge names and edge colors respectively. This allows game developers to use more descriptive names for edges such as north, south, east, and west. It also allows our abstraction of edge color into an N-vector. For example, one may define the edge color type to support an array of gradients of size *M* for gradient noise tiles as in:

```
public enum Direction { North, East, South, West };
public class HeightmapTile : IWang2DTile<Direction, int[M]>
{...}
```

Gradient noise tiles are described in Chapter 4. The interface provides one method, *C GetEdgeColor(N edgeName)*, which returns the color for a specified tile edge.

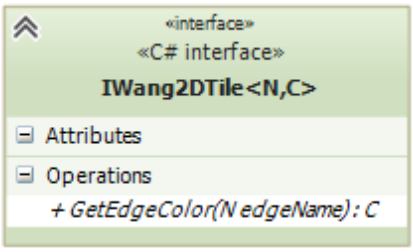


Figure 23. IWang2DTile interface

Race Track Tiles

In our race track use case, we desire to have a race track that is level, traversing a landscape which could be depicted as plains with slight rolling variation in terrain height. We also wish to include randomly distributed mountains. We would like a distribution of trees such that there are areas of dense trees mixed with glades in a naturally appearing distribution. Finally, we would like to include water features whose locations are specified by user sketch. Wang tiles are capable of handling all of these features in a seamless manner. A naïve approach might be to create a single tile set where each tile's edge color is a combination of all of the above: a heightmap profile consisting of mountains and valleys, a tree distribution, the track location, and the inclusion of water. A pre-generated tile of this format might be depicted as shown in Figure 24.

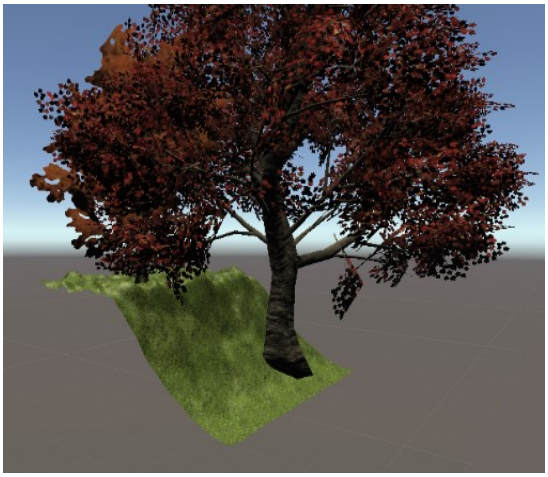


Figure 24. Example of a fully specified terrain tile with height map, terrain geometry, trees and texturing.

The problem with this approach lies in the number of tiles which need to be generated to achieve a large variety of terrains. Let us just consider several properties of each edge of each tile in such a tile set. Supposing we have 3 different distributions of trees for each edge, a binary choice of whether the track crosses the edge or not, 3 different mountainous profiles for each edge, and 4 different water profiles for each edge (water on top with land on bottom, water on bottom with land on top, all water, all land), we would need $3 \times 2 \times 3 \times 4 = 72$ edge colors for our tile set to include each combination. There are 4 edges on a square Wang tile, so we would need 72^4 or ~27 million tiles to have a complete tile set.

Instead we handle this issue with a composition of tiles. Our final tiling may be composed of composite tiles which identify all these properties, but during the generation

phase, we treat each property separately. To support our terrain example, we will build the following types of tiles:

- Noise tiles for mountains with 3 edge colors
- Noise tiles for plains with 3 edge colors
- Noise tiles for masking with 3 edge colors
- Race track tiles with 2 edge colors
- Poisson disc tiles with 3 edge colors
- Water tiles with 4 edge colors

(Poisson disc sampling is a method of generating a random distribution of points where no two points are closer together than some specified λ [73].) Even generating complete Wang tile sets for all these tilings equates to $4 \times 3^4 + 4^4 + 2^4 = 596$ tiles. Each of these tile sets is described as follows:

Bandwidth limited noise tiles are constructed using the method described in Chapter 4. In this use case, we generate 3 types of noise tiles: 1) noise tiles for generating mountainous terrain, 2) noise tiles for generating plains, and 3) noise tiles for generating masks. Each of these tile types will have different frequency characteristics. We create one class called *NoiseTile* for these three tilings. Its signature is as follows:

```
public class NoiseTile : IWang2DTile, IHeightmap,
    IComponentProvider
{
     …
```

```
}
```

We introduce a new component type *IHeightmap* for our application. This interface exposes a single method, *float GetHeight(float x, float z),* which specifies the height at the given location on the tile. We then implement the *GetComponent* method of the *IComponentProvider* interface to return a reference to the tile as an *IHeightmap* if one is requested as shown here:

```
public T GetComponent<T>() where T : class
{
    return this as T;
}
```

Race track tiles will be used to provide two types of information to the game. They will provide artist generated race track models (meshes and textures) in the form of game objects to the Unity3D game engine. They will also provide the path of the race track in the form of a query. The game may ask if the track covers a given *(x, z)* coordinate. We provide two new component provider types to provide this information. The first, *IGameObjectProvider*, has a single method *GameObject GetGameObject()* that returns a Unity3D game object. The second, *IPathQuery*, provides a single method bool *IsWithinPath(float x, float z)* which indicates whether the given *(x, z)* location is covered by the race track. Similar to the *NoiseTile* class, the *GetComponent* method is

implemented to return a reference to the class if an *IGameObjectProvider* or an *IPathQuery* is requested (and null otherwise).

```
public T GetComponent<T>() where T : class
{
    return this as T;
}
```

For these simple examples it is sufficient to implement the provider interface in the class itself; however, a cleaner implementation, especially for a larger number of component types, would be to have extended the *ComponentProviderBase* and used its dictionary implementation instead.

Poisson tiles contain a Poisson disc sampling of points such that a valid tiling of Poisson tiles will also be a Poisson disc sampling. As above, we create an interface for the data this type of tile will provide. In this case, *IPointListProvider*, provides an enumerable list of points through its single method *IEnumerable<Tuple<float,float>> GetPoints()*. As above, we implement *IComponentProvider* to return only an *IPointListProvider* or *null*.

Finally, Water tiles support a spatial query which returns whether the surface at a given *(x, z)* coordinate is covered by water. We create the interface *IWaterMapProvider* with the single method *bool IsWater(float x, float z)* for this purpose and implement *IComponentProvider* to return only *IWaterMapProvider* or *null*.

Tile Sets

A tile set is a collection of tiles. Applications may access tiles directly by keeping a reference to them or they may access tiles indirectly by keeping a reference to a tile set and an index. Indexing into tile sets provides more convenient storage with a smaller memory footprint for large tilings. Tile sets may also be used to retrieve tiles that meet certain criteria. The most relevant example is a Wang tile set in which we often wish to query for tiles by edge color.

We provide the interface *ITileSet* to represent a collection of tiles. This interface has one property and one method. The property *int NumberOfTiles* returns the number of tiles in the set. The method *ITile GetTile(int index)* returns the indexed tile. The interface is shown in Figure 25.

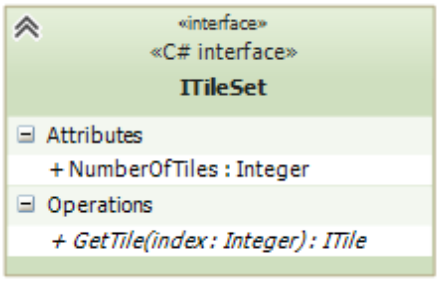


Figure 25. ITileSet interface

For Wang tile sets, we provide the interface *IWang2DTileSet*. Like *IWang2DTile*, this is a generic interface with two type parameters, *N* for edge names and *C* for edge color data. The interface has one property and one method. The property *IEnumerable<C>*

EdgeColors provides a list of edge colors used in the tile set. The method *IEnumerable<ITile> MatchingTiles(IEnumerable<Tuple<N,C>> edgeColorConstraints)* provides an enumerable set of tiles which match the specified constraints. This will help in creating tilings, which I discuss later.

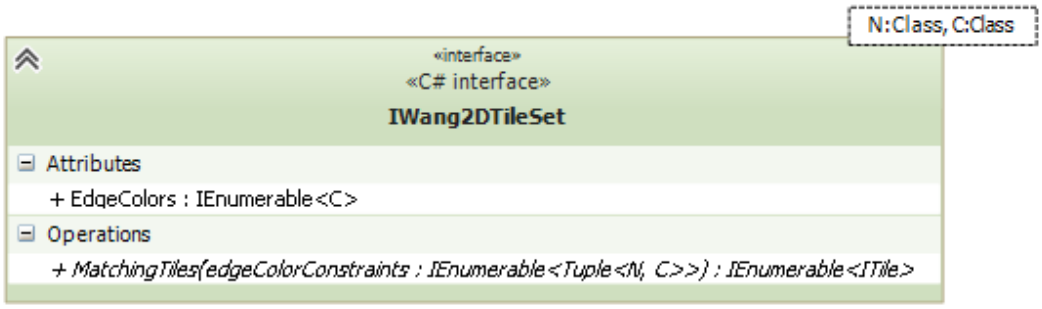


Figure 26. IWang2DTileSet interface

Finally, we provide an interface to support tile set construction, *ITileSetBuilder,* with one method *ITileSet Build()*. The user should allow the user to specify tile set parameters through either builder properties or the builder constructor and then call *Build* to generate a tile set.

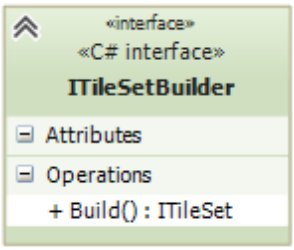


Figure 27. ITileSetBuilder interface

Race Track Tile Sets

We continue the race track use case with a collection of tile sets for each of the tile types above. For noise tiles, we create the *NoiseTileSet* which implements the *IWang2DTileSet* interface as shown below.

```
public class NoiseTileSet :
    OhioState.Tiling.IWang2DTileSet<int, int>
{
…
}
```

Note that the type parameters edge name and edge color are both represented by integers. We create a *NoiseTileSetBuilder* which implements *ITileSetBuilder*. For the constructor, we specify 4 parameters:

1. Number of edge colors
2. Size of the gradient array (see Chapter 4)

3. Number of bands of noise (see chapter 4)
4. Number of tiles per edge color combination

The race track tile set is a set of Wang tiles with 2 edge colors. The first edge color indicates no race track crosses an edge. The second edge color indicates the race track crosses the edge in the middle. Every track tile (except a blank tile with no track which we call the null tile) must have the race track cross exactly 2 edges. The 6 track tiles are shown in Figure 28. Note that the grey background is only present to show tile extent. In our Unity3D implementation, the tile only consists of the road geometry.

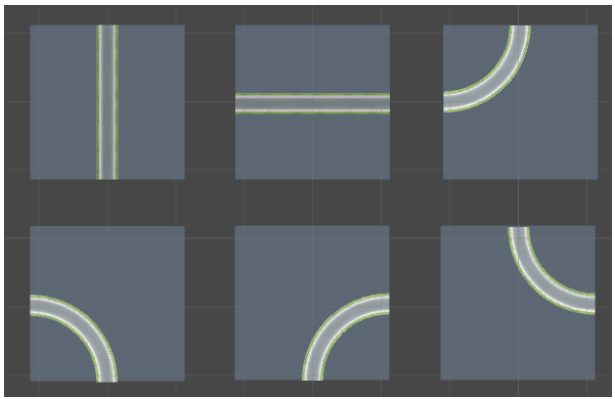


Figure 28. The 6 race track tiles.

We create the class *RaceTrackTileSet* which implements *IWang2DTileSet.* We create a *RaceTrackTileSetBuilder* which has a constructor with no parameters. The builder loads the 6 tiles from disk (and adds the null tile).

To create Poisson tiles, we divide a tiling into 3 regions as shown in Figure 29. These regions are the corners shown in white, the edge regions colored blue, orange and green,

and the tile centers colored grey. Each horizontal edge color is divided into a top half and a bottom half. Each vertical edge color is divided into a left half and a right half. To create an edge color, first we create a Poisson disc sampling in the rectangular region which encompasses the edge of 2 adjacent tiles using standard dart throwing. We create 3 edge colors for the horizontal edges and 3 edge colors for the vertical edges. These edge colors are then divided into 2 regions (top/bottom and left/right respectively). We then create a Poisson disc sampling for the middle of each tile by first setting the edge colors and then using standard dart throwing in the middle of the tile. We leave the corners empty.

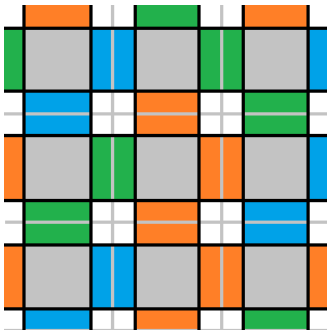


Figure 29. 3x3 Wang tiling emphasizing edge color regions.

We create the class *PoissonPointTileSet* which implements the interface *IWang2DTileSet.* We create a *PoissonPointTileSetBuilder* whose constructor has two parameters: the Poisson disc diameter and the number of edge colors.

We chose designer drawn water tiles. We define the edges of the tiles such that there are only 4 possible colors. For the left/right edge, the colors are:

- Water is on the top half of the edge and land is on the bottom half
- Water is on the bottom half of the edge and land is on the top.
- The edge is all water.
- The edge is all land.

For the top/bottom edge, the colors are:

- Water is on the left of the edge and land on the right.
- Water is on the right of the edge and land is on the left.
- The edge is all water.
- The edge is all land.

We avoid any tile which would require more than one contiguous shore line. Each tile is represented by a 256 x 256 monochrome bitmap where white specifies land and black specifies water. The complete tile set is shown in Figure 30.

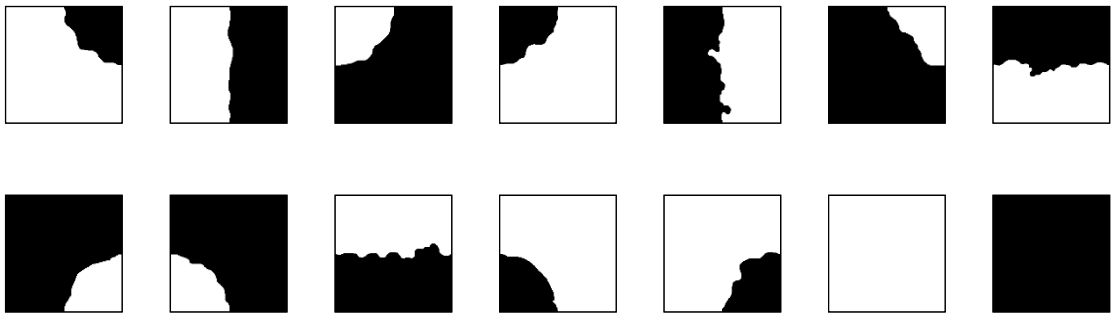


Figure 30. Water tile set.

We create the class *WaterTileSet* which implements *IWang2DTileSet.* We also create a *WaterTileSetBuilder* whose constructor is parameterless. The builder loads the tile set from disk.

Tilings

A tiling is a final product of the tile-based PCGG process. A tiling may have a representation in Euclidean space. An application may query for a tile at a given location or all tiles in a frustum (for viewing). Since a tile is a graph, it makes sense for applications to make adjacency queries. This adjacency may be Euclidean or it may represent some other game mechanic which traverses from one tile to another. (For example, a tiling may encode the movement of a game piece as a knight moves on a chess board.)

We provide the *ITiling* interface to represent tilings in our framework. This interface has only one property which is *int NumberOfTiles.* The *ITiling* interface is shown in Figure 31.

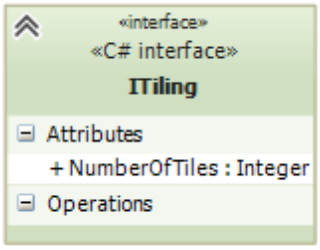


Figure 31. ITiling interface

We represent rectangular 2D tilings with the interface *ITiling2D.* We assume the tiling is laid out on the *XZ* plane. This interface has two properties. The property *int NumTilesX* specifies the *x* dimension of the tiling and the property *int NumTilesZ* specifies the *z* dimension of the tiling. We provide a struct *TileIndex2D* to specify an index into a tiling and a method *ITile GetTile(TileIndex2D index)* to query for a tile at a specific location in the tiling. The *ITiling2D* interface is shown in Figure 32.

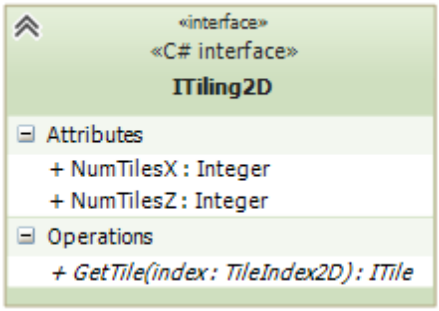


Figure 32. ITiling2D interface.

2D tilings are mapped to Euclidean space through the *ITileMapping* interface. This interface specifies the origin of the tiling and the size of each tile through properties. The *ITileMapping* interface is shown in Figure 33.

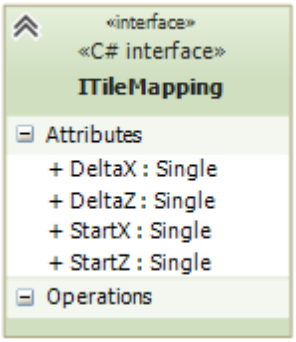


Figure 33. ITileMapping interface.

As with tile sets, we abstract out building from behavior using the *ITilingBuilder* interface. This interface has one method *ITiling Build()* which is called to produce a tiling. As with tile set construction, all tiling parameters should be passed to the concrete builder class through the constructor or properties. The user then calls *Build* to produce a tiling. The *ITilingBuilder* interface is shown in Figure 34.

Figure 34. ITilingBuilder interface.

The interface *IIndexedTiling2D* is intended for tilings in which the tiles are referenced by an index into the tile set. It provides an indexed version of *GetTile* which is *int*

GetTileIndex(TileIndex2D index). This interface returns the integer index of the tile at a given location. There are also methods to return the tile set and retrieve a tile reference by index. The interface is shown in Figure 35.

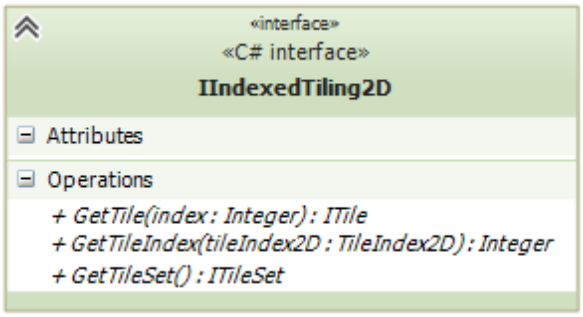


Figure 35. IIndexedTiling2D interface.

Finally we conclude our framework with a wrapper interface *IMappedIndexedTiling2D* which provides support for a 2D Euclidean tiling that stores its tiles by index. We have found this to be a common use case in our game development. The UML Diagram for the complete framework is shown in Figure 36.

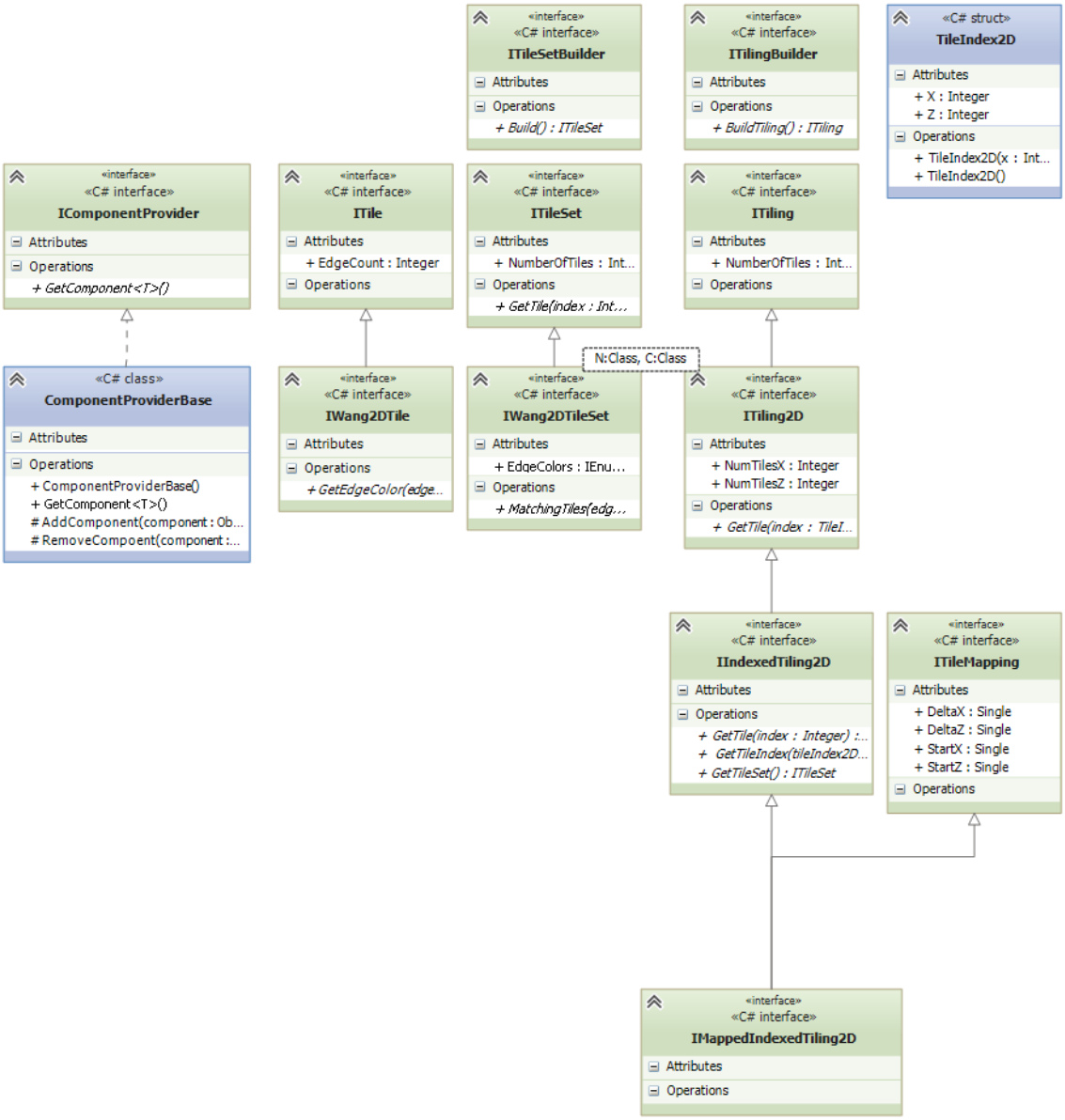


Figure 36. UML Class Diagram of the Tiling Framework

Race Track Tiling

We continue our race track example. First we develop individual tilings in support of final race track generation. We then composite those tilings into final results. We store the results in a new composite tiling which is used at run time.

We develop a class *RaceTrackTiling*, which implements the *IMappedIndexedTiling2D* interface. Race track tilings will be taken from Jacob Hayne's research which enumerates all possible race track *N x M* tilings constructed from a 2 color Wang tile set of race track pieces shown in Figure 14.

We create a *RaceTrackTilingBuilder* class that implements *ITilingBuilder*. Its constructor takes a single parameter which is the number of the race track we wish to choose. We embed this track into an 8 x 8 tiling as shown below.

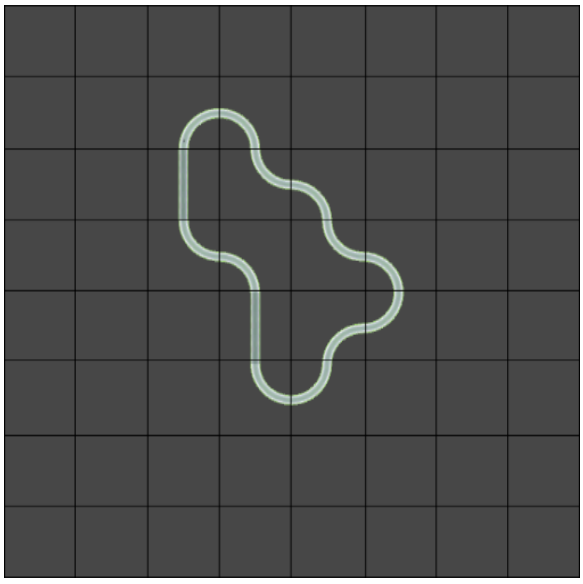


Figure 37. Race track tiling (8 x 8).

Then, we begin generating a height map for the terrain. As discussed earlier, we developed noise tiles for mountainous terrain. To generate tilings of noise tiles, we create the *NoiseTiling* class which implements the *IIndexedTiling2D* interface. We begin our height map generation process by generating a seamless tiling of noise using a *NoiseTileSet* to be interpreted as a height map for mountain features. Figure 38 shows a typical mountainous terrain generated with this technique using a single noise tile set.

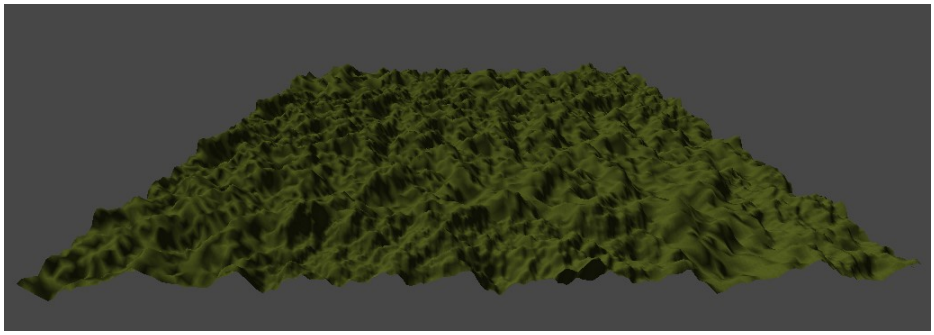


Figure 38. 4 x 4 Tiling of mountain tiles.

We use a different *NoiseTileSet* to generate a valley terrain *NoiseTiling*. We use lower frequency noise and fewer bands to simulate low rolling plains. Figure 39 shows a typical valley terrain.

We use a third *NoiseTileSet* created with a single band of low frequency noise to create a *NoiseTiling* for a mask. We threshold the noise to determine where mountains and valleys will be placed.

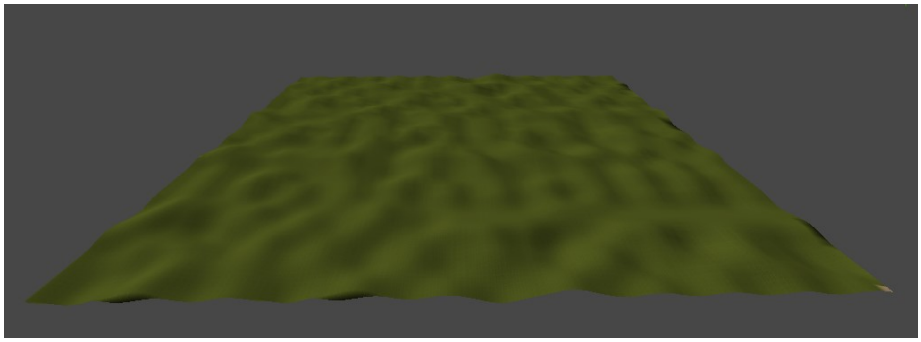


Figure 39. 4 x 4 Tiling of plains terrain.

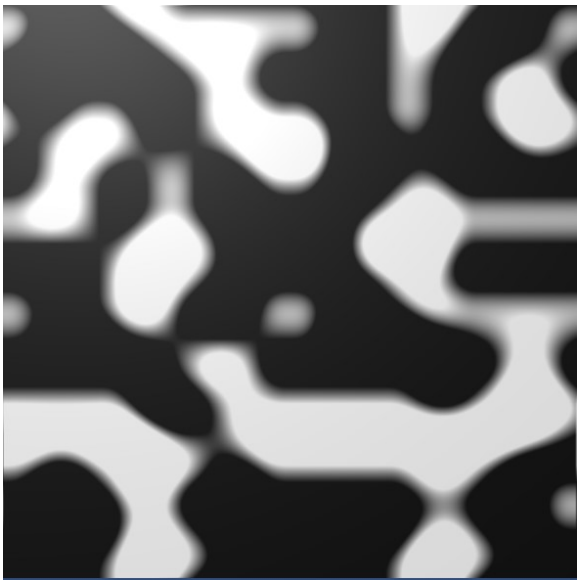


Figure 40. Mountain / Valley mask (2 x 2 tiling).

A mask tiling is shown in Figure 40. Black indicates valleys while white indicates mountains. Grey indicates a region where height will be linearly interpolated between mountain and valley to create smooth transitions. We composite the race track into the mask so that we will not have mountains intersecting the track. One possible mountain

valley mask with a track embedded is shown in Figure 41. We reduced the amplitude of the mountains to avoid having the track cut through them too often. The resulting height map is shown in Figure 42.

Figure 41. Mountain / Valley mask with track embedded.

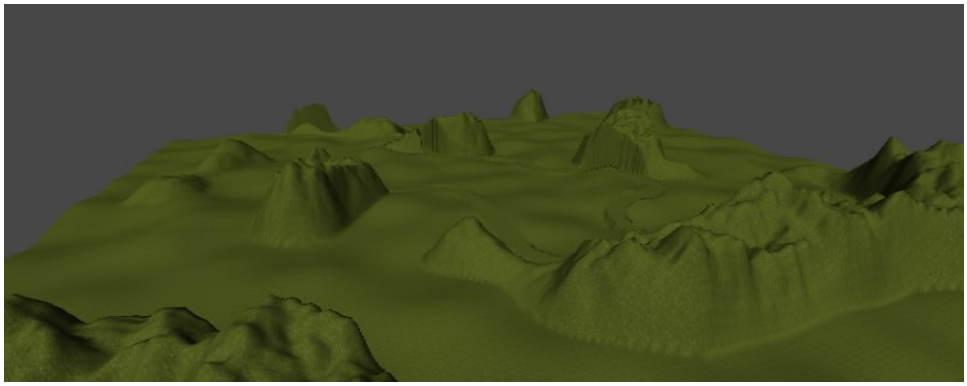


Figure 42. Mountain and valley tiling.

We use the hand-drawn water tiles described above to create a water tiling. We create the *WaterTiling* class which implements the *IMappedIndexedTiling2D* interface to hold

this tiling. A *WaterTilingBuilder* uses a designer sketch to specify the tiling. The designer sketch is overlaid over an $N\ x\ N$ tiling and edges are categorized into 4 possible states which match the edge colors of our water tile set. A designer sketch is shown in the figure below.

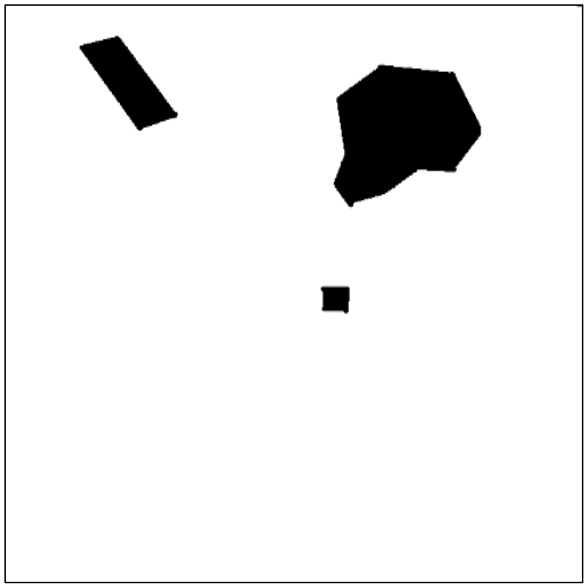


Figure 43. Water sketch method.

The water tiling is used with our already existing race track tiling and our terrain mask to embed water features into the terrain such that there is no water under the track and water is not under a mountain. An example final heightmap with water is shown in Figure 44.

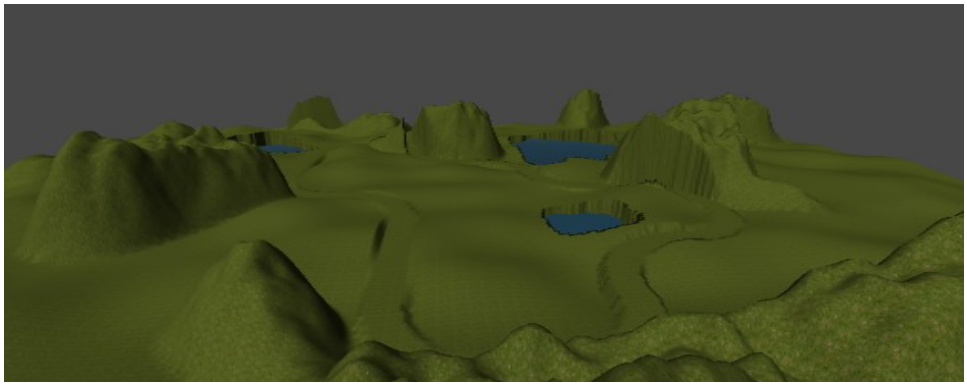


Figure 44. Water added to height map.

Lastly, we place trees according to a Poisson disc sampling. We create the *PoissonTiling* which is a simple Wang tiling of Poisson tiles. To add glades to a forested landscape, we again use a single frequency Perlin *NoiseTiling* as a mask. Finally, we check the *TrackTiling* and the *WaterTiling* to make sure no trees are placed on the track or in the water. A final race track is shown in Figure 45.

Figure 45. Final race track.

Results

We have used our tiling framework in the development of Recovery Rapids (Chapters 5 and 6), in Unity3d terrain examples, and in Survive!, a maze game. It has been used successfully by 13 students in a procedural content generation special topics class spring 2015. It has also been presented as a design concept to the autumn 2015 and spring 2016 game design capstone students. So far, we have found this framework flexible enough to enable a diverse set of tile-based PCGG. It is currently being used for ongoing projects and we continue to solicit feedback to improve its design and usefulness.

Chapter 4. Procedural Textures Using Tilings with Perlin Noise

Procedural content generation is used to create increasingly realistic and interesting content for games and film, while CPU and GPU performance continue to increase and reduce the overall cost of such techniques. Tilings can be used to reduce the memory and development cost of textures used in video games and film. Here we study the use of procedural textures and tiling together. Specifically we examine tiles of noise. One difficulty encountered when generating procedural textures is to avoid repetition. Repetition in textures reduces believability. This can be avoided using aperiodic tiling. We present a method of constructing a tile set which can produce an aperiodic tiling of Perlin Noise.

Related Work

In the SIGGRAPH ’85, both Ken Perlin and Darwyn Peachey introduced the concept of noise and its use in computer graphics [1][58]. Both describe a process of using smooth functions to generate a texture; however, Perlin introduced the concept of a piece-wise smooth function based on a matrix of random gradients on an integer lattice. The function was guaranteed to be continuous in the first derivative. Later, Perlin [2] improved his noise algorithm by changing the interpolation function to be continuous in the 2^{nd} derivative as well as providing a speedup (on a Pentium 3 processor) of about 10%. He also provided a reference implementation which is ubiquitous today. Since this

introduction, other noise algorithms have been considered. Cook and DeRose, working at Pixar Animation Studios, noted 2 issues with Perlin noise [8]. First, because it is not completely band limited, it can cause aliasing when viewed at a distance. Because aliasing is usually more unacceptable than loss of detail, bands are attenuated aggressively. This causes loss of detail in the distance and textures seem to pop onto objects as you zoom in. This can be seen in Figure 46 on the left where the terrain fades to grey as you look further into the distance.

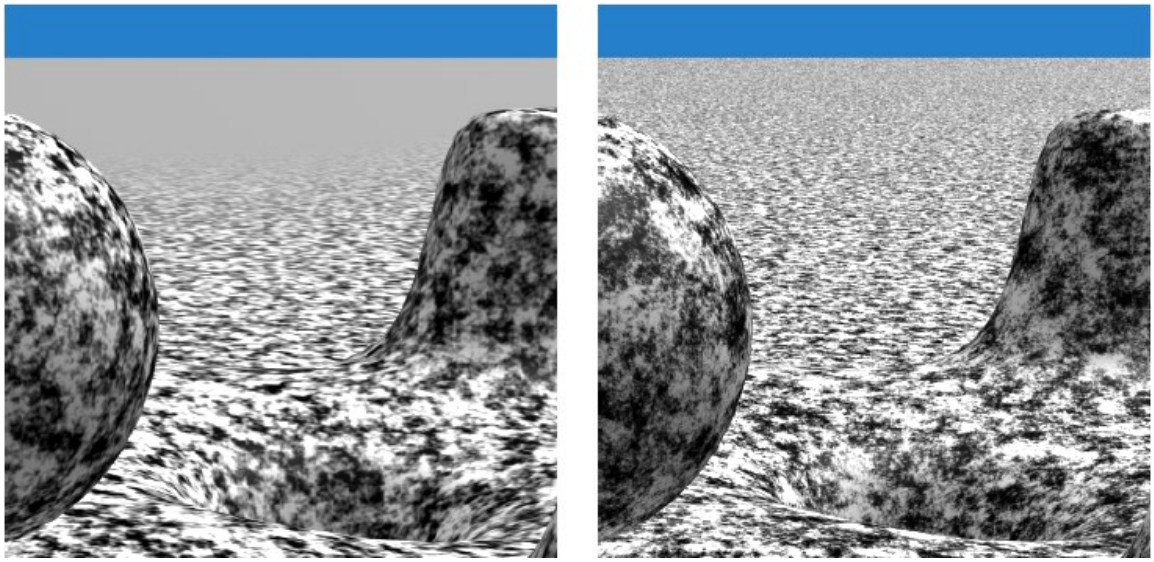


Figure 46. Perlin noise (left) and Wavelet noise (right).

Second, the standard technique of texturing the surface of 3D objects by sampling into a 3D Perlin Noise is even less band limited. Cook and DeRose address these issues by using wavelets to generate noise. Wavelet noise is more narrow-band than Perlin noise and can be more easily attenuated to avoid aliasing. For 3D texturing they use a

projection of the integral of wavelet noise along the perpendicular. They show that this projected noise is also band-limited. (For our purposes of 2D textures, Perlin noise is adequate and still widely used. Future work would involve adapting our technique to Wavelet noise.) Goldberg shows another approach to solving the balance between level of detail and aliasing [9]. He generates oriented noise by decomposing white noise using Fourier analysis and then dividing the noise into oriented bands. Noise textures are then generated from the inverse Fourier transform of the oriented noise bands. These textures are better suited for anisotropic filtering. Lagae proposed generating anisotropic noise based on a Gabor kernel [11]. This allows better control of both the frequency and orientation of the noise. They also provided an algorithm for calculating surface noise based on a point *p* on the surface and the surface normal. Similar to Goldberg, this surface noise is also band-limited. A good survey of procedural noise functions can be found in [6].

The concept of aperiodic tiling has been an interesting mathematical problem for years. Stam was the first to apply the mathematical concept of aperiodically tiling an infinite plane with a finite set of tiles to graphics [3]. His tiles were precomputed homogeneous textures. These tiles had labeled edges, analogous to Wang tiles, which were tiled such that edges with the same label were adjacent to each other. The tiles were constructed such that the tiling would be seamless if this condition held. In [7], Cohen further popularized the use of Wang Tiles in computer graphics. He showed that a simple stochastic algorithm could be used to generate aperiodic tilings. He also showed that this technique could be used with both 2D Poisson disc samplings and 3D geometry.

To support tile-based PCGG, we introduce a fast, low memory footprint GPU implementation of aperiodic tiling of Perlin Noise. Secondly, we introduce a technique of providing tileable turbulence on the GPU. Finally, we show how to implement varying frequency noise with tiles.

Noise Tiles

Current state of the art with regard to seamless tiling still uses periodic tiling [4]. Consider a noise function $F : \mathbb{R}^2 \rightarrow \mathbb{R}$. A seamless tile can be generated by a function $G : \mathbb{R}^2 \rightarrow \mathbb{R}$ defined as follows:

```
G(x, y) = (
      F(x, y) * (w - x) * (h - y) +
      F(x - w, y) * (x) * (h - y) +
      F(x - w, y - h) * (x) * (y) +
      F(x, y - h) * (w - x) * (y)
      ) / (w*h)
```

Here *w* is the width of the resulting tile, and *h* is the height. As *x* goes from 0 to *w* and *y* goes from 0 to *h*, this method samples noise from 4 rectangular regions of $\mathbb{R}^2$. Note that at each of the corners, there is no blending, and at the middle of the image, all 4 samples are equally blended. A tile generated like this is seamless along the edges; however, it still generates noticeable artifacts when tiled due to the blending in the center of the tile. Figure 47 shows a plane tiled with a seamless tile of Perlin Noise. This figure shows a discernible

repeating pattern even though the tile is seamless. This pattern is accentuated because of the blurring in the middle of the tiles due to averaging properties of the equation above.

To improve on this we wish to construct an aperiodic tiling without blurring. This involves the generation of a tile set containing multiple tiles of noise. To construct such a tile set, we must first consider the properties of the tiles themselves and then consider the properties of the noise with which we wish to fill the tiles.

Cohen describes how a set of Wang Tiles can be used to tile the plane. The edges of the tile are considered colored, where color is abstracted to mean a combination of color, pattern, and appearance along the edge. In order to seamlessly tile a plane, the pattern on the right edge of one tile must match the pattern on the left edge of the tile immediately to its right. Similarly, the pattern on the bottom edge of a tile must match the pattern on the top edge of the tile immediately below.

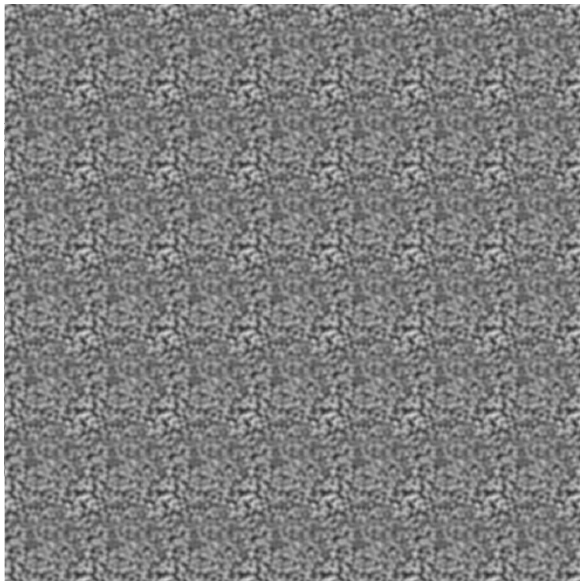


Figure 47. A seamless tiling of noise generated using Equation 1.

Our technique takes advantage of the Perlin Noise algorithm to generate seamless boundaries procedurally. Perlin Noise is generated by using a set of random gradient values specified on the integer lattice. Each gradient value is chosen using a pseudo random algorithm.

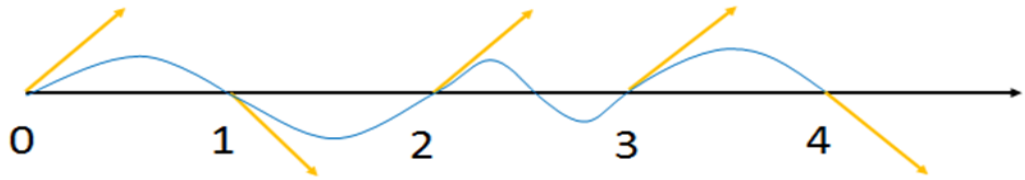


Figure 48. Perlin noise in one dimension.

Perlin Noise can be explained as shown in Figure 48. A random gradient (shown by orange arrows) is selected for each integer point. The noise value for any point between two integers is calculated as a bi-quintic interpolation of the nearest two gradients. This means instead of linear interpolation using t as fractional measure of the distance between one gradient and the next, Perlin interpolates using u calculated from t as follows:

$$u = 6t^5 - 15t^4 + 10t^3$$

This function is chosen to guarantee that the first and second derivative of the noise function is *0* when $t = 0$ or $t = 1$ (i.e. at any integer point on the number line). Thus the noise function is C^2 continuous even though it is calculated using a piecewise function.

For any given point(x, y) in $\mathbb{R}^2$, Band limited noise is achieved using a bi-quintic interpolation between the gradients at the four surrounding integer lattice points. The range of influence of a given gradient is thus a window ± 1 unit in all coordinate directions.

Now let us consider two 2D tiles constructed of Perlin Noise. These two tiles can be seamlessly tiled horizontally by constructing the gradients along the left edge of one tile to match the gradients along the right edge of another as shown in Figure 49. The top shows two tiles individually with matching gradients along the adjacent edge. When they are joined together (bottom), the tiling will be seamless. Note that this involves making sure an $N\ x\ 1$ vector of gradients on the left edge of one tile matches the $N\ x\ 1$ vector of gradients on the right edge of the other. We know that the value of the noise at the gradient location will be the same. Furthermore, because of the quintic interpolation used by Perlin, the first and second derivative of the noise function is 0 at the integer lattice points. Because the noise function is C^2 continuous, the transitions across tile boundaries will not be apparent to the eye. This guarantees a seamless tiling. Similarly these tiles can also be seamlessly tiled vertically by constructing the gradients such that those on the top of one tile match the gradients on the bottom of another. Since the $N\ x\ 1$ column (or row) is arbitrary and does not affect the rest of the tile, the width K of the Wang Tile needed for Perlin Noise is $K = 1$.

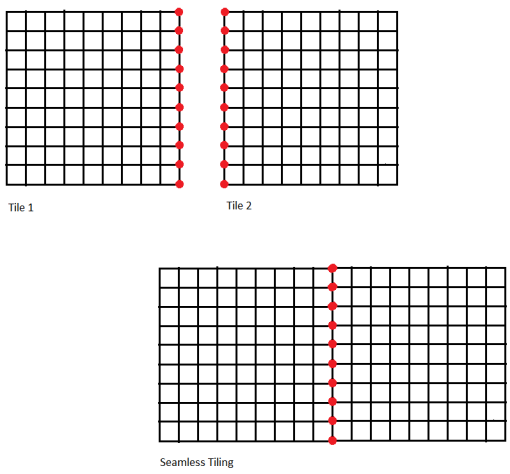


Figure 49. Creating a seamless tiling with adjacent integer lattices.

Constructing Noise Tiles

Using the above, we construct a set of Wang Tiles to aperiodically tile a plane. In order to do this, let us consider tiles of size *m x m.* There are $N = m + 1$ integer lattice points along the edge of the tile. Each tile needs an $N \; x \; N$ integer lattice to generate noise. We define a color C as an $N \; x \; 1$ vector of gradients. In preparation for creating our integer lattice for each tile, we construct a set of n colors, $\{C_0, C_1, \ldots, C_{n-1}\}$. Using these colors, we construct a Wang tile by constructing its integer lattice as shown in Figure 50. Each byte at coordinate (x, y) represents an index to a gradient. Each edge is an edge color vector C_n. All the corners use the same value. Note that left, right, top, and bottom of this tile are all different colors chosen from our set of *n* precomputed edge color vectors.

73	52	231	127	2	18	38	144	37	210	34	233	21	63	183	73
151	109	174	183	253	94	84	151	220	224	18	38	22	138	203	29
227	15	149	60	139	237	67	215	32	120	102	24	238	128	164	206
72	116	27	62	157	56	18	112	33	170	86	90	89	125	250	195
52	19	208	58	27	62	109	127	97	127	60	44	228	226	126	219
120	136	164	128	9	160	39	233	237	91	187	84	120	177	111	241
184	202	69	53	174	107	248	173	122	128	210	156	213	3	124	28
238	18	219	57	157	121	191	173	244	106	183	10	86	9	255	250
214	199	95	18	180	95	14	30	26	21	234	198	127	98	101	148
15	100	110	204	109	68	32	158	129	66	72	191	41	118	149	138
17	66	207	245	30	127	162	194	211	226	68	92	226	44	148	4
46	43	121	234	145	39	26	110	66	27	2	77	253	249	60	16
58	190	110	74	78	140	48	88	252	245	145	101	218	54	217	48
10	115	181	247	182	110	92	179	6	150	231	123	232	2	112	15
207	60	42	51	95	88	122	38	187	121	242	231	185	193	160	193
73	18	134	176	9	8	198	28	1	98	141	194	188	143	252	73

Figure 50. Integer lattice for a single Wang tile.

We usually choose the tiles such that N is a power of 2. The corner is shared among 2 edges. Furthermore each corner is adjacent to 3 other tiles in a tiling. Again since the radius of influence of a single gradient value is ± 1, a gradient at the corner affects the noise generated across 4 tiles. If 4 tiles meet at a corner, but choose a different gradient for that spatial location, the noise function would not be continuous in the region around the corners. We resolve this by using a single common gradient for all corners. Our testing

shows that if $N > 4$ the choice of a single corner color does not develop discernible patterns in the noise. When $N \leq 4$, we have also used the corner tiling method shown in [10] to add variability to the corners. In this case, an edge color is a unique combination of the corner colors on its endpoints, meaning that if we have p corner colors, we need $n = p^2$ edge colors.

Turbulence Tiles

So far, we have discussed a single frequency Perlin Noise; however, Perlin Noise is often used in multiple frequencies or bands. Perlin first described this in [1] as 1/f noise which he called turbulence. Turbulence can be defined as:

$$Turbulence(p) = \sum_{i=0}^{n} \frac{Noise(2^i p)}{2^i}$$

Above, we create tiles of size $N \times N$ where $N = m + 1$ to support noise queries across a tile whose size is $m \times m$. This would not support a query for $Noise(2^i p)$ where $i > 0$ as the location $2^i p$ would be off of the tile and we would have no gradients for that location. It may be possible to subsample from a single integer lattice of size $Q \times Q$ where $Q = 2^n m + 1$, but our edge would only be consistent for the highest frequency band of noise. A complex method would be needed to make sure we maintained edge colors correctly. We instead generate n tilings of noise to implement turbulence. If we think of each band of noise independently, each band samples from an integer lattice of random gradients. It is not important that they sample from the **same** integer lattice at different frequencies. The

bandwidth limited nature of Perlin noise is maintained regardless of the random gradients chosen. To encapsulate this concept, we modify the standard turbulence equation above to be:

$$Turbulence(point) = \sum_{i=1}^{n} \frac{Noise_{\lambda_i}(point)}{2^i}$$

Where λ_i represents a new set of gradients for each band of noise. To support the highest frequency band of noise needed for turbulence, we construct tiles using an integer lattice of size $Q \; x \; Q$ where $Q = 2^n m + 1$. To generate noise of half the frequency we construct tiles using an integer lattice of $\frac{Q}{2} \; x \; \frac{Q}{2}$. We continue this for each band of noise until we reach $m \; x \; m$ tiles. We then create a tiling for each band of noise. From the turbulence equation, we construct Turbulence as a weighted sum of i bands.

The tile set (and the tiling) for each band of noise can be identical in terms of edge colors (and adjacent tiles). The only thing that changes is the integer lattice used for the noise. We find it more convenient to have a single tile set where each tile has several integer lattices – one for each band of noise – instead of several similar tile sets. A structure which supports this is mipmaps. A mipmap is a special structure for holding a set of textures where each texture's dimensions are twice the size of the last. It is implemented in both DirectX and OpenGL and can be stored directly on the GPU. If each band used the algorithm above to tile seamlessly, we have $n \; x \; i$ edge colors. We construct the mipmaps as follows. We generate a set of gradients to use as corner colors with one gradient for

each band. We define edge colors for turbulence to be a set of vectors of sizes $N\ x\ 1, \frac{N}{2}\ x\ 1, \frac{N}{4}\ x\ 1$, ..., where we use one vector for each mipmap level. The number of bands available for turbulence is determined by the number of mipmap levels available and the maximum tile size. We use a minimum tile size of 4 x 4. Therefore, if we have 128 x 128 pixel tiles we can have 6 mipmap levels or bands.

Varying Frequencies

If a shader requests noise of frequency *f* at point p where f is not a factor of 2, we interpolate between mipmap levels. A simple linear interpolation between mipmap levels does not work because the frequency increases exponentially as the mipmap level increases. Instead we use $log2(\ freq\ )$ to determine the interpolation between two mipmap levels. Figure 55a and Figure 55b show two examples of varying frequency tilings. In Figure 55a, the frequency varies linearly in the horizontal direction. Figure 55b uses a noise tiling to determine the frequency requested at a given spatial location.

Implementation

Noise Tiles

To pass a tiling of noise to the GPU as a single texture would require a large amount of memory. Consider a 128 x 128 tiling of Perlin Noise. Suppose each tile is represented by a 16 x 16 array of bytes with one byte for each gradient. We would have to pass a 2048 x 2048 texture to the shader to implement this tiling, using 4Mb. Instead, we pass the gradients for the tiles as a texture array to a GPU shader. We also pass a texture which includes the tiling. In general, this is called indirect or indexed tiling. In the above example, for a set of Wang tiles with 2 edge colors and one tile per edge color

permutation ($2^4 = 16$ tiles), we would need one 16x16x16 texture array and one 128 x 128 texture for the tiling, using a total of 20Kb, a 200x reduction in memory usage. The gradients for each tile are generated using the following algorithm:

1) Choose a random corner gradient G.
2) Construct an array for edge colors C[M, N] where M is the number of edge colors and N is the size of the gradient tile.

   a. `For i = 0 to M -1, C[i,0] = G,`
      `C[i, N-1] = G`

   b. Set the rest of the elements of the array to random gradients (or random indices into an array of predetermined gradients).

3) Each tile in the tile set has a top, bottom, left, and right edge color. Set the edges of the T[N, N] array of gradients for each tile.

```
For i = 0 to N - 1
     T[i, 0] = C[top, i]
     T[i, N-1] = C[bottom, i]
     T[0, i] = C[left, i]
     T[N-1, i] = C[right, i]
Next
```

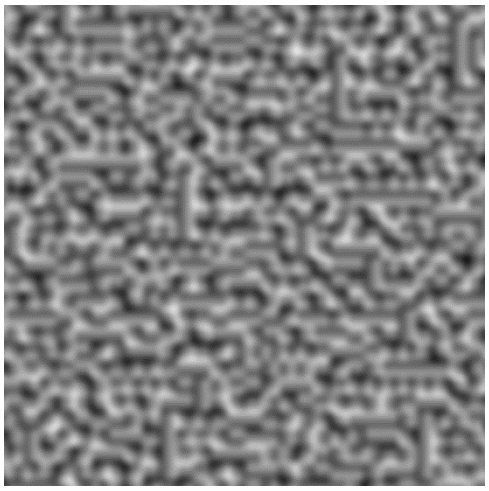
4) Fill the middle of the tile with random gradients.

```
For i = 1 to N - 1
     For j = 1 to N - 1
          T[i,j] = Random gradient.
     Next
Next
```

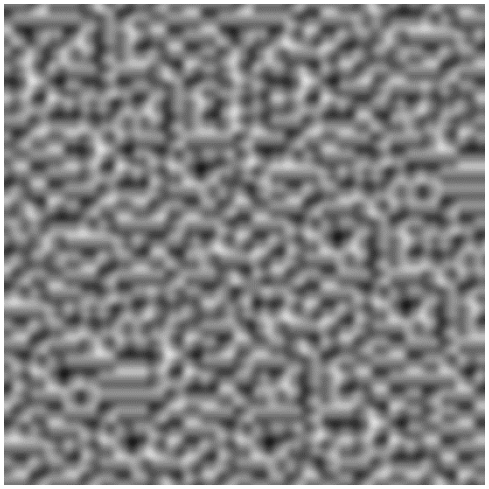
These gradient values are pre-computed and stored in a texture array with one $N \; x \; N$ texture for each tile. The final procedural texturing is computed on the GPU. Figure 51 shows a plane tiled with Wang tiles constructed in this manner on the bottom. The top is standard Perlin noise shown for comparison. We show the frequency characteristics of Perlin Noise in Figure 53. Original Perlin noise and Wang tiled Perlin noise exhibit similar frequency characteristics.

Turbulence Tiles

We implement turbulence on the GPU similarly passing the gradients as a texture; however, for turbulence, each frequency is stored as a separate mipmap level as described above. Figure 52 shows a comparison between turbulence generated from tiles and the Perlin implementation. Figure 54 shows the 2D FFT of Perlin turbulence and Wang tiled turbulence. Our turbulence (on the right) shows a more directionally independent nature to the signal, though it is also less band limited in nature.



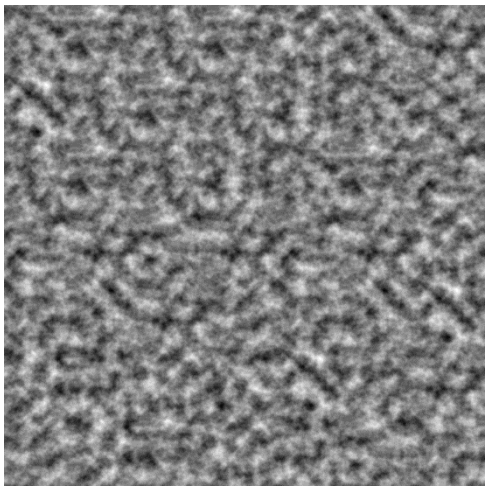
a.



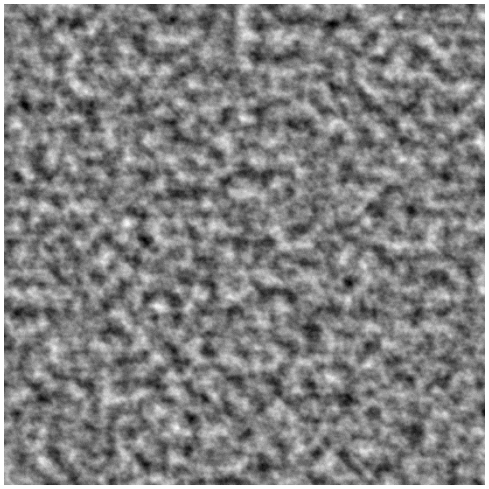
b.

Figure 51. Original vs. Tiled Perlin.

The Original Perlin is above and the Tiled Perlin is below.



a.



b.

Figure 52. Original vs. Tiled Turbulence.

The Original Turbulence is above and the Tiled Turbulence is below.

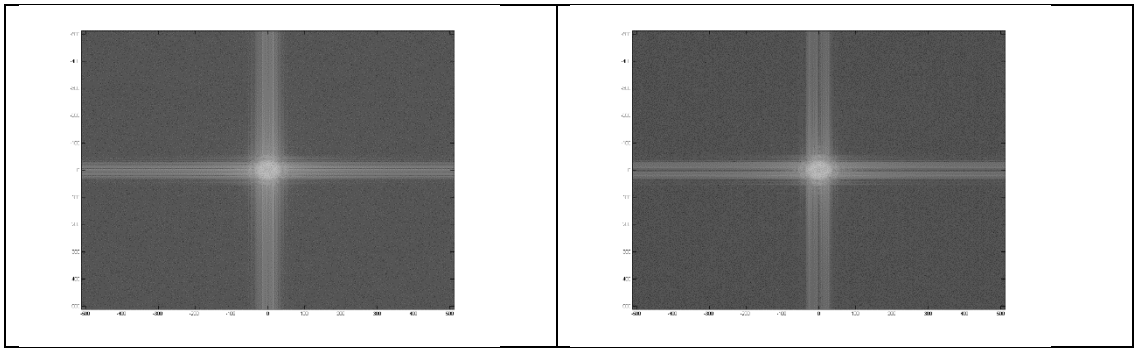


Figure 53. 2D FFT of Perlin noise (left) and Wang tiled Perlin (right)

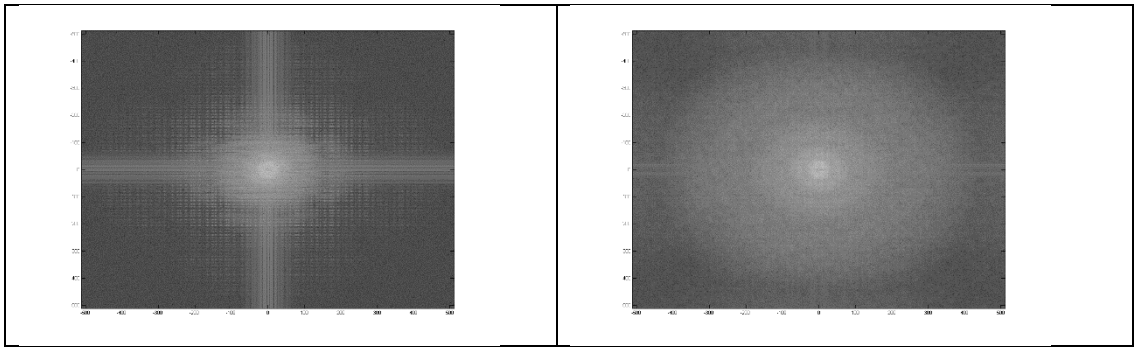


Figure 54. 2D FFT of Perlin turbulence (left) and Wang tiled turbulence (right)

Varying Frequency Tiles

We implement the varying frequencies described in section VI by generating a set of gradients as with turbulence. We then perform 2 lookups to determine the noise at the closest 2 frequencies to the requested frequency and perform a log interpolation between the noise values. We provide two sample tilings of varying frequency in Figure 55. First we show a linear ramp function as the frequency increases linearly in the x direction across the tiling in Figure 55a. Second, we show an interesting pattern where the requested frequency itself is generated using noise in Figure 55b.

Marble

Marble is also implemented on the GPU. To generate marble, we subtract different layers which are composed of turbulence themselves. We use a single tiling to generate turbulence; however because the gradients for each tile are fixed, the turbulence for a specific frequency for each tile is unique. The tiling of turbulence is generated as described above. We use an offset (Δx, Δy) into the tiling for each layer of marble, subtracting the tiling from itself, shifted by the offset to generate a marble pattern. Figure 55d shows marble generated with this method.

Wood

Our wood pattern is a composition of tilings of noise. Each tiling specifies a property of the wood. For all three layers, we sample the noise at higher frequency in X to create wood bands. In the first layer, shown in Figure 56a, we use the equation

$$output = frac(\, noise(p) * banding \,)$$

Where *frac* returns the fractional part of a real value and *banding* is the banding density of the wood. This creates bands similar to growth rings in a cut plane of wood. The second and third layers (Figure 56b and Figure 56c) are different frequency samplings of Perlin noise (sampled at much higher frequency in *x* than in *y*) to simulate marks left by cutting and sanding. Figure 55c shows the final tiled wood.

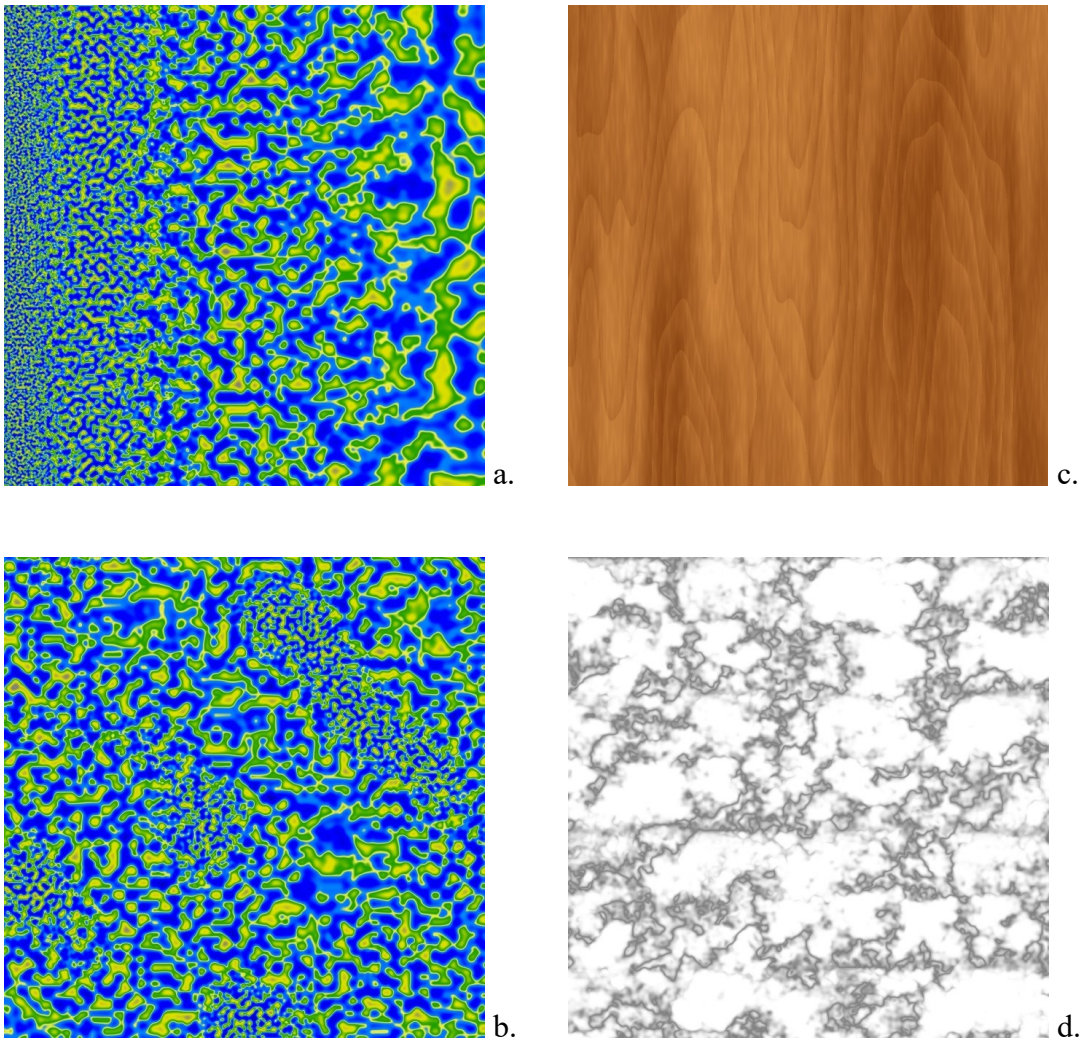


Figure 55. Seamless 8 x 8 tilings of wood, marble and varying frequency.

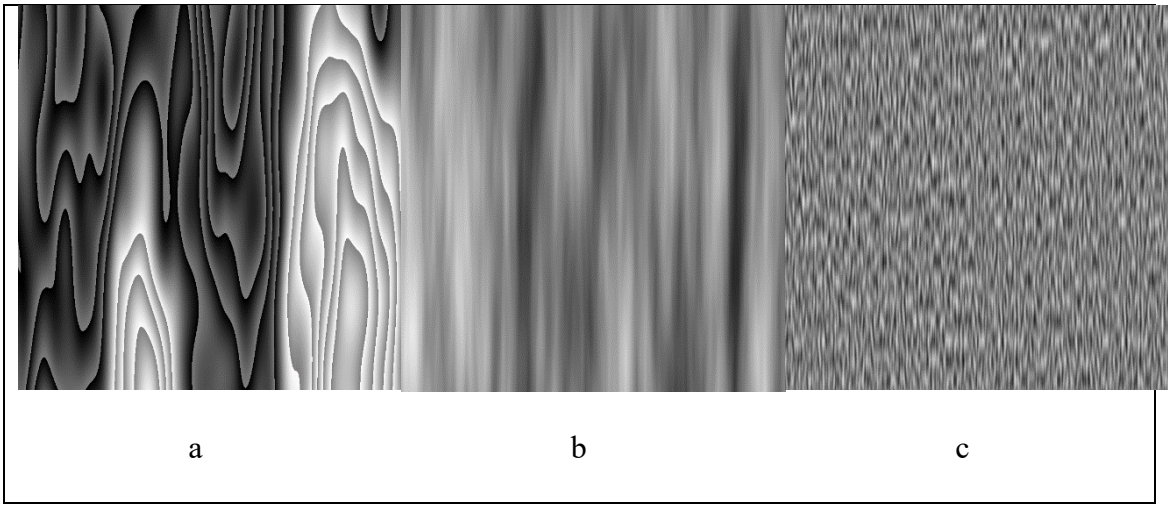


Figure 56. Construction of wood tiling.

Chapter 5. Development of Recovery Rapids – A Game for Cost Effective Stroke Therapy

The majority of United States citizens with diminished hand and arm function have inadequate access to long-term rehabilitation care. Stroke, the leading cause of hemiparesis, affects arm-hand function in about 350,000 people per year [21][24] in the United States. Hemiparesis can additionally result from other neurological events including traumatic brain injury, multiple sclerosis, cerebral palsy, spinal cord injury, and surgical resection. If clinical practice guidelines [25] were followed, the majority of individuals would be expected to receive at least some outpatient rehabilitation [26], yet only 30.7% received this care. Amongst those who do receive outpatient rehabilitation services, most do not receive evidence-based treatments [27]. One particularly striking example of the dissociation between the scientific rehabilitation literature and clinical practice is the limited clinical dissemination of constraint-induced movement therapy (CI Therapy). CI therapy has been shown to provide improvements in paretic arm function and frequency of use [27], and to promote structural and functional brain plasticity [15]. Over the past 20 years since it was first described in the literature, more empirical support for its efficacy has been generated than for any other upper extremity intervention. Although it can be successfully implemented in up to 75% of stroke patients who exhibit residual hand/arm dysfunction, fewer than 1% of stroke survivors have access to this

treatment. CI therapy is unavailable to most stroke survivors due to its cost, travel/scheduling demands, and dearth of trained providers.

To overcome these accessibility challenges, we assembled an interdisciplinary team from The Ohio State University and Nationwide Children's Hospital consisting of researchers from computer science, engineering, rehabilitation, and therapy. This team had the following members:

- Lynne Gauthier, PhD. Assistant Professor of Physical Medicine and Rehabilitation and Principal Investigator
- Roger Crawfis, PhD. Associate Professor of Computer Science
- Linda Lowes, PhD, PT. Nationwide Children's Hospital.
- Alex Borstad, PhD, PT, NCS. Assistant Professor of Research Physical Therapy
- Ryan McPherson. Lecturer Electrical and Computer Engineering.
- Lise Worthen-Chaudhari, PhD. Associate Director, Motion Analysis and Recovery Laboratory
- David Maung. Graduate Student, Computer Science

We developed a game using the Microsoft Kinect that is designed to provide CI Therapy in the home setting. To the best of our knowledge, this is the first attempt to create a stand-alone home-based rehabilitation program for upper extremity hemiparesis **based on an empirically-validated treatment**. Although a few avatar-based virtual reality games and programs [16] have been designed for rehabilitation, they have not promoted critical motor learning elements, such as intensity of practice and carry-over of therapy gains to daily activities, that are being implemented here and have typically been of low

complexity (prompting the person to reach out and touch virtual objects in repetitive patterns). Furthermore, other robotic technologies offered to facilitate delivery of motor rehabilitation have been criticized for not incorporating critical motor learning elements or delivering empirically-validated treatments [13]. Existing upper extremity stroke rehabilitation technologies may supplement therapy effectively, but are thus unlikely to be successful stand-alone interventions that can be implemented in a home setting. In keeping with the delivery schedule for CI therapy employed in the EXCITE trial [29], game content has been designed to support three hours of gameplay per day for ten consecutive weekdays. This 30 hour per individual play time necessitated the use of PCGG as the team didn't have the resources to generate enough hand-made content for a 30 hour game.

This chapter discusses how the team incorporated the requirements of CI Therapy and feedback from the target audience into game design and implementation.

Requirements for Gamified CI Therapy

The overarching requirements of this game design are founded in the empirical basis of CI therapy. Critical elements of this protocol are massed motor practice and the "transfer package" - a number of behavioral techniques that *facilitate transfer of therapeutic gains to everyday activities*. It is through this combination of activity-based medicine and behavioral techniques that CI Therapy enables the patient to overcome a conditioned suppression of movement (e.g., learned nonuse) characteristic of chronic hemiparesis. [30][31] Transfer package techniques include a behavioral contract, daily monitoring of

use of the more affected arm for everyday activities [via daily administration of the Motor Activity Log (MAL)] and guided problem-solving to overcome perceived barriers to using the extremity.

Effective video-game delivery of CI therapy required the following six game design considerations: 1) support play time of three hours daily for 10 consecutive weekdays, 2) be sufficiently motivating to encourage the user to perform repetitive challenging therapeutic movements, 3) incorporate elements of the "transfer package" to encourage the user to increase use of the more affected upper extremity for daily activities. These include automated administration of the Motor Activity Log (MAL) with problem-solving via branching logic from user responses and encouraging the user to wear a restraint mitt on his/her less affected arm during daily life outside of the game. 4) adjust the difficulty and type of motor practice based on an individual's therapy needs, 5) adjust difficulty as a user's motor function improves (shaping/ leveling up), and 6) aesthetically appeal to the target audience. Our target audience consists primarily of older adults over the age of 50 with little video game experience who have had a stroke and children with hemiplegia. These 2 dichotomous populations made choosing an engaging environment challenging. Surveys of potential adult consumers revealed heterogeneous desired aesthetics, yet the majority of stakeholders desired aquatic elements (e.g., lake, ocean, stream, underwater). Acceptance by pediatric healthcare necessitated that violence in the game is avoided. Every aspect of the game reflects these requirements. Recent advances in game interface devices has led to the development of commercially available and affordable motion capture devices such as the Microsoft Kinect camera/sensor, shown in

Figure 58. Using the Microsoft Kinect, our team developed a game that delivers the critical components of CI therapy, based on the aforementioned considerations, which is described in detail in the Game Mechanics and MAL sections below.

Assessment

One unique element to our design and development process is that success is defined by clinical outcomes. Design assessment therefore incorporates validated outcome measures from the field of rehabilitation medicine. These assessments are administered both before (pre-test) and after (post-test) a prescribed duration of game play (30 hours over 10 weekdays). They quantify the ability of the game to enhance rehabilitation compliance and improve motor function.

Outcomes measured include the following: time the game was played (logged within the game), total number of gestures performed for each type of gesture (logged within the game), time the constraint mitt was used (measured by the instrumented mitt and logged through the game), and standardized tests of movement (e.g., Wolf Motor Function test).

Target Audience

In the video game arena our target audience of people with chronic hemiparesis is unique for several reasons. First, they are often from an older demographic that is not accustomed to video games. Second, they often have other cognitive difficulties resulting from neurological injury. Third, nearly 75% of people who experience stroke are over 65 years of age. Finally, they are highly motivated to recover function in their hand or arm. This equates to the following design considerations: 1) this audience may be less able to

tolerate in-game errors, poor instructions, and difficult user interfaces, 2) they will be more motivated to continue playing a game that they perceive as beneficial to their motor function, with less regard for aesthetics.

Platform

Our team sought a low cost, commercially available platform with a modern natural user interface device that facilitates registration of skeletal joints and movement kinematics. We selected the Microsoft Kinect for Windows. Our current systems are quad-core Intel i5-3450 processors running at 3.1 GHz with 8 GB RAM and NVidia GeForce GT 640 graphics cards

A camera-based controller was chosen to ensure that the participants were being compliant with the therapy. The use of the Kinect was a key decision in designing the game. Using an off-the-shelf skeleton tracking system freed up substantial developer time.

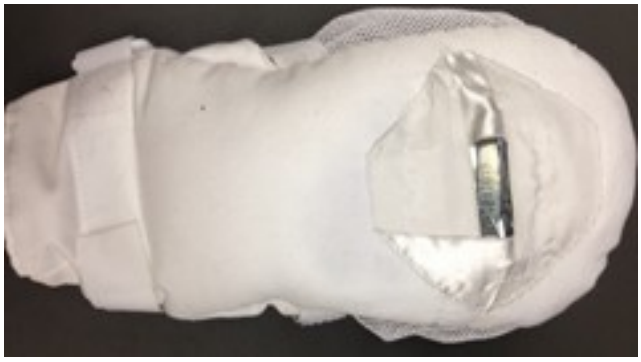


Figure 57. Mitt with timer.

Restraint Mitt

One component of CI therapy is a thick padded restraint mitt that, by constraining the less affected arm, encourages use of the more affected side for daily activities. Constraint is prescribed for 90% of waking hours during the 10 treatment days. See Figure 57. In clinic-based CI Therapy, mitt time would be logged and reported to the therapist daily. The team desired an objective method of quantifying mitt use to verify compliance with this portion of the therapy and therefore designed an instrumented mitt. A standard Posey® mitt was enhanced with a digital timer, electronic switch, and LCD screen to display wear time. At the beginning of each game session, the player is asked to input the mitt time and is reinforced with bonus points based on the time entered.

Kinect One

Important components of the rehabilitation program include training supination and distal motor function (i.e., hand). The original Microsoft Kinect was unable to accurately identify some of the skeletal motions associated with reaching and grasping – specifically Kinect could not recognize wrist supination/pronation nor individual finger flexion/extension motions. Our team was able to participate in the Microsoft Early Access program for the Kinect One. This device has both enhanced resolution over the original Kinect and an additional 6 skeletal joints including a thumb and hand tip.

Figure 58. Microsoft Kinect One[2].

Game Mechanics

One effective method of approaching game design is to examine a game from the perspective of its mechanics, their interactions, and their effect on the game state [22]. We considered all three of these elements to produce this gamified version of CI therapy. With regard to game state, in many games the most important state, or the ultimate goal, is the win state. Our desire to have a less stressful, more relaxing play style, and to accommodate repetitive play guided us toward a game in which the user completes laps on a game circuit for a score. The ultimate goal is therefore to compete against one's own previous scores, thus providing feedback on improvement. Score is based on successfully performing in-game activities (each of which requires exercise of a

[2] Image Evan Amos / Wikimedia Commons / Public Domain

therapeutic gesture) and lap completion time. In this way, the score is reflective of the amount and quality of therapy performed. A time-based bonus encourages achieving more successful gestures per unit time.

Because this gamified therapy focuses only on movement of the upper body, the game was designed to be performed seated, both for safety and accessibility (for individuals with limited mobility). Qualitative data from stakeholder interviews revealed the importance of a game aesthetic that incorporated water. For these reasons, we chose the protagonist to be an individual paddling downriver in a kayak. We balanced the need for variety, which is important to maintain engagement, with the need for familiarity, which was important for patients with post-stroke cognitive deficits, by sequencing through two scenes in a continuous loop. The game environment consists of an outdoor river valley in a natural setting, followed by an underground maze with a semi-industrial theme as shown in Figure 59 and Figure 60. Each continuous lap takes about 15 minutes to navigate through. To add additional diversity, each lap the maze and river valley are procedurally generated and the outdoor ecology changes by retexturing the plants and landscape.

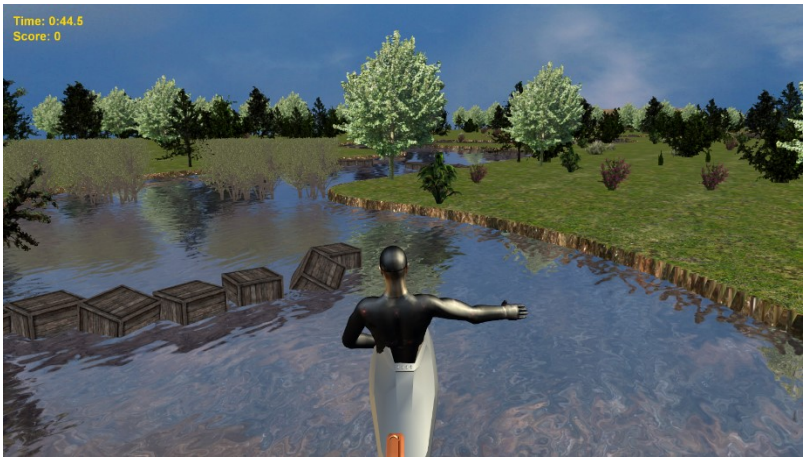


Figure 59. Outdoor gameplay.

Many game mechanics were tailored to therapy goals such as range of motion per joint, interjoint coordination, and smoothness of movement performance. The movements used to achieve these goals were permutations of reaching and grasping. Mechanics were further reduced to target motion at the skeletal joints involved in reaching and grasping, such as the shoulder, elbow, wrist, fingers and thumb.

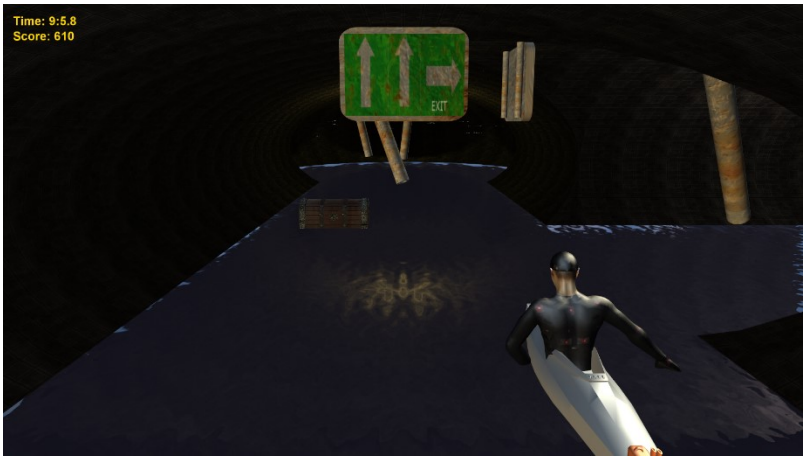


Figure 60. Underground gameplay.

Our team developed a gesture library which allows rehabilitation researchers to encode gestures recognized by the Kinect sensor and tie these gestures to game mechanics [17]. Gestures coded to encourage reaching of the paretic arm included the following: rowing (from forward of the torso to behind the torso), reaching across the midline of the body, and reaching laterally to the paretic side. These gestures were attached to the rowing, move right, and move left mechanics, respectively. One particularly difficult reaching motion for persons with stroke is to reach above the head. A gesture requiring overhead motion was attached to the Bat Repelling game mechanic described below. Here are many of the mechanics that were designed specifically to promote desired therapy movements:

- Rowing. The player must complete a row gesture to make the boat travel down-river
- Avoid Barriers. Barriers are placed along the river path, blocking 1 or 2 lanes at a time. These barriers require the player to perform the move-right or move-left game mechanic to successfully navigate the river. If the player runs into a barrier, the boat stops and a small negative value is added to the score.
- Rapids. In rapids, the boat automatically moves downriver at higher speed than in rowing sections, allowing the user to focus on the gestures for moving the kayak from side to side, while making it more challenging to avoid barriers.

  Fishing. The fishing section is indicated by jumping fish and the avatar's use of a fishing net. The player gestures to scoop the net through the water to attempt to

catch fish. Each successful scoop yields a 50% probability of catching a fish. See Figure 61.

Figure 61. Fishing. Not all fish are equally rewarding!

- Parachutes. Parachutes yielding survival supplies fall from the sky over the river. The user must perform a gesture to catch the parachute and receive a reward – currently extending the arm forward and performing supination. See Figure 62.

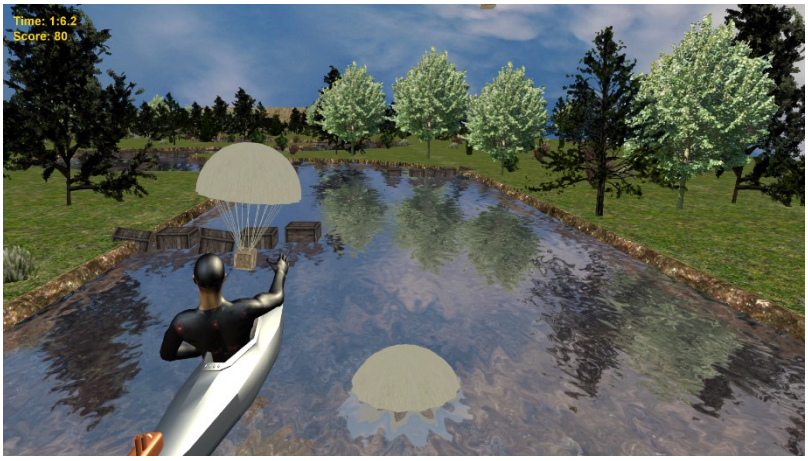


Figure 62. Attempting to catch a parachute.

- Bottles. The river is littered with plastic (2 liter) bottles as shown in Figure 63. The player must perform a gesture to retrieve the bottle from the water and receive a reward.

Figure 63. Pulling bottles from the river.

- Maze navigation. The lanes in the maze control the direction of the boat as indicated by signs on the ceiling of the tunnel. The user must perform the left and right gestures to position the kayak in the desired lane to navigate the maze.
- Pipes. Dangerous pipes hang down from the ceiling inside the maze as shown in Figure 64. The player must perform a gesture to avoid the pipes or use the left/right gestures to navigate to a lane that does not contain a pipe.

- Chests. Treasure chests are found within the maze. See Figure 64. The user must navigate to the correct lane to run into a chest to receive its reward. Chests encourage exploration of the maze for increased score at the expense of time.

Figure 64. Swatting bats to move forward.

- Bats. Bats are located throughout the river and underground maze. If the player runs into bats, the bats harass the player and stop forward motion. The player must perform a gesture to shoo away the bats and continue. See Figure 64.
- Picking fruits. Picking fruits involves both a targeted reaching and grasp/release. While traveling downriver, bushes indicate the location of fruit-picking. When the player reaches the bushes, the game changes to a scene which provides a hand cursor and a set of fruits on background bushes. See Figure 65. The player must move the cursor over each fruit target and perform a gesture (closing and opening

the hand) to pick and release as many fruits as possible within a 2 minute time frame.

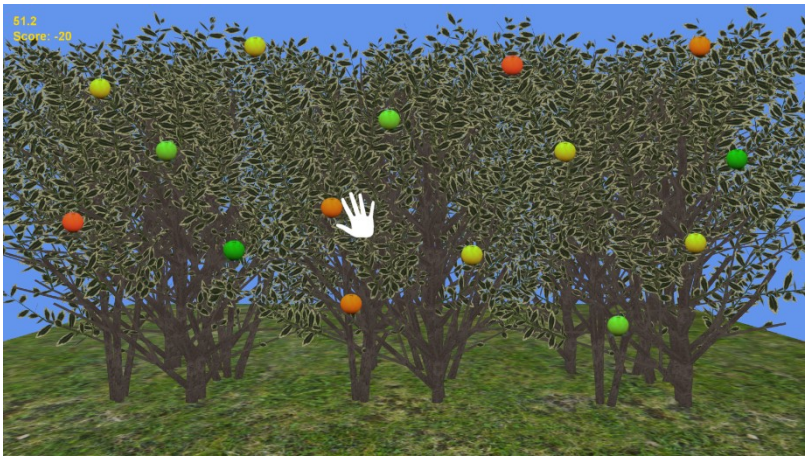


Figure 65. Fruit picking mini-game.

Because the goal of the game is to increase mobility and function of the arm and hand, a change in performance during the course of game play is expected and desired. Therapeutic goals were promoted over time by leveling up the kinematic-based game mechanic triggers to require increasingly more challenging movements to be performed. For instance, range of motion required to trigger a game mechanic might be increased over time as might duration of hold at the reach/grasp end point.

Transfer Package

The transfer package is a number of behavioral techniques that facilitate transfer of therapeutic gains to everyday activities. Those portions of the transfer package which are implemented in game are the daily monitoring of the use of the affected arm in everyday activities through the MAL and problem solving. The MAL implementation in-

game consists of a series of 26 questions asked during the course of 6 hours of gameplay. At the end of one lap of the content, a number of MAL questions are administered such that the total number of questions asked per day is 13 or greater. Each question asks the individual to rate his/her ability performing a daily task such as turning on/off a light switch on a 5 point scale. This can be seen in Figure 66. Video queues are given to help establish a common rating scale. If the individual had the opportunity to perform such a task, but did not, problem solving is performed via branching logic from user responses. In this way, a user selects a technique that may assist him/her in accomplishing a task in the future. MAL results are logged and performance gains can be measured over the course of treatment.

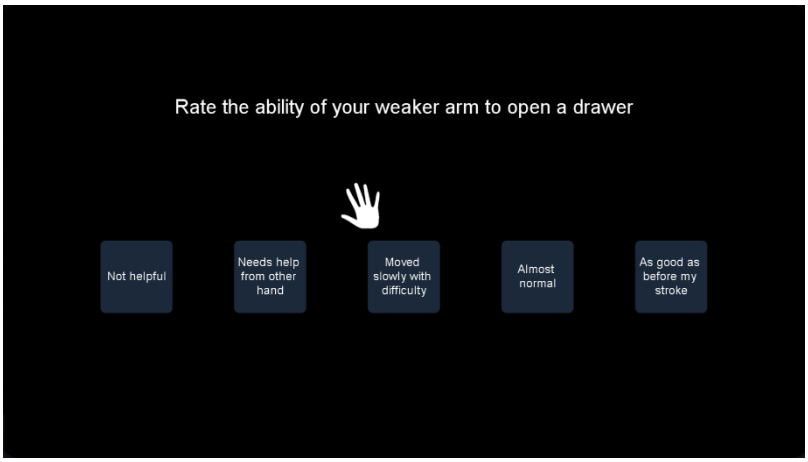


Figure 66. Motor Activity Log.

Acknowledgements

In addition to the authors, a number of students have contributed to the Recovery Rapids game during the course of their education at The Ohio State University. These include Garrett Davis, David Hazlett, Jacob Grealy, Josh Adams, Dan Carlozzi, Ben Bricker,

Kala Phillips, Yinxuan Shi, Juan Roman, Jana Jaffe, Dan Bibyk, Wesley Stover, Kyle Donovan, Tristan Reichardt, Evan DeLaubenfels, and Rengao Zhou. This work was supported by the Patient-Centered Outcomes Research Institute (PCORI) grant and UL1TR001070 from the National Center for Advancing Translational Sciences. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Center for Advancing Translational Sciences or the National Institutes of Health.

Chapter 6. Implementation of Recovery Rapids

Developing artistic content for 30 hours of gameplay was not feasible given limited development resources. Procedural content therefore provided a feasible alternative. Procedural content is used in creation of the outdoor river valley scene; the indoor maze scene; the placement of trees, barriers, pipes, and chests, and the creation of interesting boulders.

Procedural River

Our team produced two methods of generating a river. In the first case, the path of the river was modeled as a periodic function and specified based on its Fourier coefficients. This produced a believable river path through the valley, but it was not easy to control the direction of the river at the beginning and end when joining the outdoor river with the underground waterway. In the second method, the river was built from tiles, each containing a segment of river. When constructing seamless tilings to create a believable terrain, it is necessary to make sure adjoining edges match. Seamless tilings with textures was described in Chapter 4, and our method for 2D height maps is analogous, with the addition of criteria for the river. Adjoining tiles were constructed such that the river crossed the boundary smoothly and the first and last tiles were designed such that the river meets the underground pathway. Tiles allowed us to customize therapy to the individual user by allowing the user/therapist to select the frequency with which various

game mechanics occur (each associated with particular motor movements). Therapists specify the number of river tiles and the therapy to be performed on each tile using an XML configuration file.

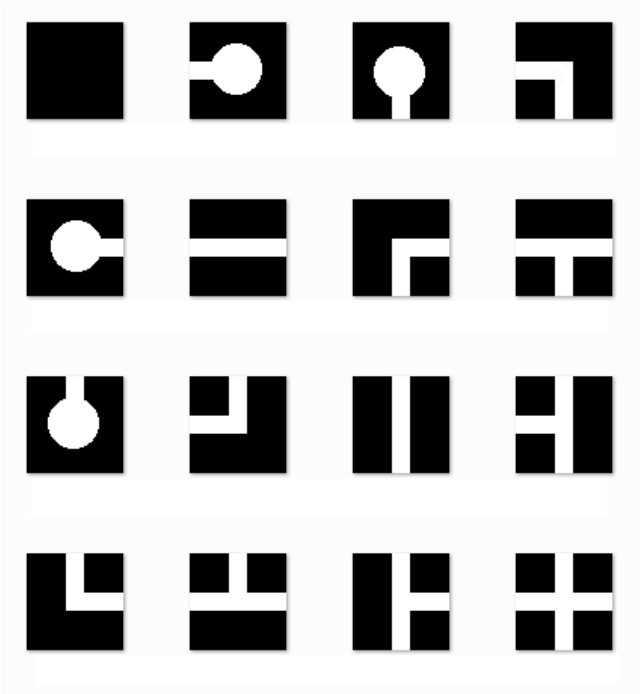


Figure 67. Set of 16 maze tiles.

Procedural Maze

We construct our maze using a set of Wang tiles. This particular set of Wang tiles recurs frequently in our research, yet with a different image. (Compare Figure 67 and Figure 75.) For this Wang tile set, we choose 2 valid edge colors. The first indicates that a player traversable path crosses the edge and the second indicates that no path crosses the edge. We stipulate that all paths on a tile intersect in the center. For this tile set, we need $2^4 = 16$ tiles to cover every possible permutation of edge coloring of a tile. One set of 16

tiles is shown in Figure 67. To generate a maze, we create an *N x N* indexed tiling where each element of the array is an index into the tile set as follows. The maze is initially seeded with a set of random integer indices in each cell by choosing a random state for each cell edge as open or closed. (This technique maintains a valid Wang tiling under the rule that adjacent edge colors match.) All boundary edges of the maze are initialized as closed. The entrance to the maze is then placed in the upper left-hand corner and the exit is placed at the bottom right-hand corner. The maze at this stage may be disconnected. We generate a list of all disconnected components. To connect the maze, we recursively grow a random component by choosing a closed edge adjacent to the component and changing it to open. If this edge was adjacent to two components, opening it will join the components. This algorithm continues until all components are joined. This algorithm is in fact a version of Prim's algorithm seeded with a random forest [65]. Maze difficulty was tuned based on play testing, and a maze size of 5 x 5 was found to provide sufficient challenge and interest with desired lap time.

Tree Placement

Trees are placed using a Poisson disc sampling with a given radius *r*. We use a quick Poisson disc sampling algorithm which takes a given number of trees *n* and a radius *r* and a maximum fail value of *m*. If a new point is not able to be found in *m* iterations, *r* is modified to be *r*/2. Trees are further constrained so as not to fall within the river path.

Barrier Placement

The river is divided into segments of contiguous tiles where all tiles provide the same therapy. We call each segment a river section and often refer to it by the name of the

major game mechanic the section provides such as “fishing section” or “parachute section”. As the boat travels down river, the user may travel in one of three lanes: center, left of center, and right of center. Barriers are placed throughout the river path, requiring the use of the move left and move right game mechanics, which are attached to therapeutic gestures, to avoid the obstacles.

Barriers are placed procedurally along the river path traversing from the start of the canyon to the end of the canyon. Barriers block one or more lanes of travel down river. The set of possible barriers and which lanes they block is shown in Table 1. The placement of a new barrier is based on the previous barrier as shown in Table 2. The choices in Table 2 are designed such that at least one lane that was previously available is no longer available. For example, the player would have to use the right lane to pass a barrier of type 2. The next barrier will be of type 4 or 5. Both of these barriers close the right lane and require the player to move left. Thus, barrier placement is designed to require movement to the left and right while navigating the river.

Table 1. Lanes blocked for each barrier type.

Barrier Type	Lane is Blocked?		
	Left	**Center**	**Right**
1	Y	N	N
2	Y	Y	N
3	N	Y	N
4	N	Y	Y
5	N	N	Y
6	Y	N	Y

Table 2. Allowable next barrier type.

Barrier Type	**Choices allowed for next barrier**
1	2, 3, or 4
2	4 or 5
3	1,2, 4, 5, or 6
4	1 or 2
5	2,3 or 4
6	2,3, or 4

Pipes and Chests

Pipes and chests are placed in the maze on a per tile basis. Note that in Figure 67 a straight path exists from the center of each tile to each open edge. This gives us an algorithm for choosing pipe and chest placement along the tile path with equal probability for any given location. Given *r* pipes and *s* chests per tile, a location is chosen for each along the valid paths in the current tile as follows. A random open pathway on the tile is chosen (N, S, E, or W) and a unit distance from the center of the tile to the edge. The pipe or chest is then placed the selected distance from the center of the tile along the selected path.

Players navigate the maze by choice of lane when they approach an intersection. If there is an open pathway to the left and the player is in the left lane, the player will turn left. Similarly, if there is an open pathway to the right and the player is in the right lane, the player will turn right.

Due to therapist and patient feedback, we avoid placing chests in intersections at the center of the tile, as this created a conflict between the lane in which the player wants to travel to make the appropriate turn in the intersection and the lane required to retrieve the chest.

Procedural Boulders

Believable rock formations are formed by applying functional operations on a regular unit sphere mesh. Each operation perturbs the vertices of the mesh. These operations are random displacement, subdivide, extrude to sphere, random slice, and smooth. A rock is defined as a set of operations $O = \{o_1, o_2, \ldots, o_n\}$, where o_i indicates a specific operation

from the above list. Several rocks are constructed at load time using a fixed random seed and used throughout the game.

Logging

The following items are logged during play:

- Session start time.
- Skeletal data.
- Each gesture state completed.
- Each completed gesture.
- Mitt total time.
- Answers to MAL questions.

Chapter 7: Applying Formal Picture Languages to Procedural Content Generation

In this chapter, we focus on the use of tiling in procedural content generation for games. We examine two new approaches to specifying procedural content: 2D Regular Expressions (2RE) and Array Grammars (AG). Formal languages, specifically grammars, have been used in PCGG. L-systems have been used in procedural modeling for everything from plants to cities [34][43]. Smith, et al. use a grammar to convert game rhythm based cues into game geometry in the development of platformer levels [41]. Figure 68 shows the grammar used. Here player states shown in bold on the left-hand side of each rule (which are non-terminals) are used to determine game constructs on the right hand side of each rule (terminals) which will be generated in a game level.

Moving → Sloped | flat_platform
Sloped → Steep | Gradual
Steep → steep_slope_up | steep_slope_down
Gradual → gradual_slope_up | gradual_slope_down
Jumping → flat_gap
| (gap | no_gap) (jump_up | Down | spring | fall)
| enemy_kill
| enemy_avoid
Down → jump_down_short | jump_down_medium | jump_down_long
Waiting-Moving → stomper
Waiting-Moving-Waiting → moving_platform_vert
| moving_platform_horiz

Figure 68. A rhythm based grammar

Dormans discussed a two-step process of developing the game mission and spatial representation separately. He used a shape grammar to embed the mission into a mission space [44]. Later, he used graph grammars to modify missions [60]. Figure 69 shows an example of rules in his grammar. Each circle or square represents a room in a game. Each edge is a path from one room to another. The following abbreviations are used in the figure: T = task, K = key, L = lock. He did not consider his graph grammars a formal method though, and he did not use them in a spatial sense as we use AG here.

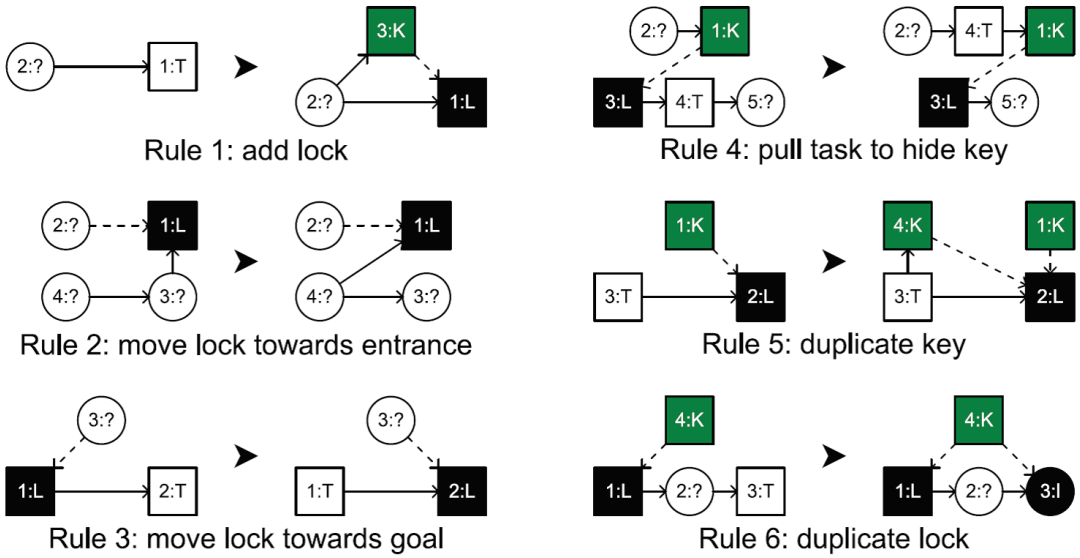


Figure 69. Dormans' grammar for changing mission spaces

2D Regular Expressions and Array Grammars

Scientists in the area of vision and picture recognition have been interested in formal languages describing pictures. In this area, pictures are modeled as distinct elements or pixels ordered into a 2D array. The elements are considered to be members of a finite alphabet Σ. A language L over a 2D array of elements (usually infinite in theory) is a set of pictures which is a subset of all possible pictures Σ^{**}, where ** indicates an extension of the idea of Kleene * into 2 dimensions. For example, a** would be the language of all pictures containing only the letter a.

Procedural content generation using tilings in 2D can be thought of as selecting a subset which meet a certain criteria from the set of all possible tilings. In any reasonable sized tiling of $n \times m$, the number of possible tilings is intractable. Given a tiling of $n \times m$ with an alphabet of p possible tiles, the number of possible tilings is bounded by p^{mn}. For example, if we wanted to generate a 16 x 16 tiling with only 4 possible tiles, there would be $4^{(16x16)}$, or 2^{512} possible tilings we could generate. This is assuming all tiles are able to be used interchangeably. Often, we have rules which constrain our set of valid tilings (e.g. adjacency rules such as Wang tiling or rules which restrict tile choice at a given location.). In these cases a random approach to tiling generation would be inefficient.

In order to make generation feasible, we would like to generate only tilings which have a reasonable probability of being within our subset of desirable tilings. If we think about tilings as pictures, we can consider the desired subset as a language L using an alphabet Σ of p possible tiles. Then the desired subset is a formal language over Σ^{**}. Formal

language theory applies. This chapter is an exploration into the application of formal theory over 2D pictures for procedural content generation for games.

In [40], Togelius and Shaker describe a standard model for PCG called the Search Based Approach. This approach has three main features: 1) the search algorithm, 2) the content representation, and 3) the evaluation function. Many generators do not have a good search procedure, possibly leading to many missed configurations and many duplicates.

Formal language theory has generators which could be thought of as search algorithms. It also has acceptors which could be used as evaluation functions in a search based approach. Acceptors include regular expressions, logic formulas, automata, and tiling systems. Generators are generally thought to be various types of grammars (2RLG [33] and array grammars [39]). Acceptors naturally fit the role of content evaluation. Togelius shows in general that grammars have been accepted as a method of content creation.

Herein we explore the space possible with generators. We theorize about their computational complexity and we examine the types of content creation these generators are useful for.

We use two types of generators. For the first, we choose 2RE to define our language [33]. We show that 2RE, with a few novel programming techniques, can be used effectively as generators as well as acceptors. Even with simplistic operations on atoms (or single tiles) we can generate some useful patterns. We also show that 2RE can be used to enumerate all possible members of a given language over tilings.

For the second type of generator, we choose array grammars [39]. AG were also used by Bollen in the generation of 'Dules (See chapter 5 of [40]). We explore the difference in content that can be generated with 2RE and AG. We give three further examples of generated content: 1) perfect mazes using a set of 16 tiles, 2) possible river paths for Recovery Rapids (see Chapter 6) using a set of 6 tiles, and 3) a simplistic outdoor setting using a set of 27 tiles from a 7-color seamless Wang tiling.

Formal Language Theory

Formal language theory shows that 2RE and AG generate a large set of languages, implying that most procedural content using tilings could be expressed with one or the other. Giammarresi and Restivo discuss several methods of defining languages L over Σ and provide unifying theories describing the set of languages which can be defined [33]. One important set of languages is the recognizable languages called REC. These languages can be represented by 2RE. We choose this representation and the set of languages that it can generate as the first representation to explore. We describe our implementation and results with 2RE in Section 3.

In [39], Rosenfeld discusses AG and their associated languages. The important theorem from this work is:

> Theorem 5.1. The languages generated by array grammars are the same as the languages accepted by Turing array acceptors.

A Turing array acceptor is the 2D equivalent of a Turing machine on a single tape. In other words, a Turing array acceptor is a Turing machine that operates on a 2D tape. It can move up, down, left, and right.

While theory is interested in infinite arrays or pictures without bound, we are often interested in generating pictures in which at least one dimension is bounded. We reason that for fixed size pictures, or pictures with one dimension (either height or width) constrained, AG is as powerful as any other model of computation. We do this by first showing that we can represent any picture of the above type on a 1D tape. We then show that for any Turing machine M_1 which operates on these 1D tapes and accepts a language L representing a set of pictures P, we can create a Turing array acceptor M_2 which accepts P. Thus, Turing array acceptors can accept the same set of languages as Turing machines and by Theorem 5.1, AG can generate any language a Turing machine can accept.

To show this, we consider pictures with constrained width first. Then we consider pictures with constrained height.

If width is constrained to n columns:

- Create a mapping H between the set of all 2D pictures of width n, called Ω^{n*}, and their representation on one dimensional tapes, denoted Σ^n, where the rows of each picture p are laid out sequentially on t. This mapping is bijective.

- For any Turing machine M_1 which runs on the tapes Σ^n and accepts a language L, create a Turing array acceptor M_2 which runs on Ω^{n*}. It runs exactly like M_1 with the exception of the following changes:
    - M_2 has additional states to keep track of which column of a picture p it is on.
    - If M_1 would move left and we are on the first column of p, M_2 moves up one row and to column n instead.
    - If M_1 would move right and we are on column n of p, M_2 moves down one row and to the first column instead.

M_2 will accept P such that $H(P) = L$. If height is constrained to m instead, we create our mapping such that the columns of P are laid out on sequentially on t instead. M_2 then keeps track of the current row of p instead of the current column.■

In all but the case of tiling the infinite plane, since Turing array acceptors accept the same set of languages as Turing machines, by Theorem 5.1, AG's generate the same set of languages accepted by Turing machines. The question remains how hard it is for a designer to define such content using AG. We describe our implementation and results with AG in the section "Array Grammars".

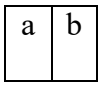
2D Regular Expressions

Definition of 2D Regular Expressions

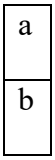
A 2D regular expression is constructed from a set of atoms (where each atom is a character of the alphabet Σ) and the following set of operators:[3]

∪	Union
∩	Intersection
|	Horizontal concatenation
-	Vertical concatenation
*|	Horizontal concatenation (0 or more times)
*-	Vertical concatenation (0 or more times)
()	Parenthesis indicate precedence of operations

In our case, each atom indicates a tile chosen from a finite tile set. The regular expression a|b indicates



While a-b indicates



[3] In [2], horizontal and vertical concatenation were represented by circumscribed horizontal and vertical bars respectively, however, these symbols are not readily available in common Unicode fonts.

The regular expression ((a|b)-(b|a))*|*- indicates all possible *N x N* checkerboard tilings where N is even, beginning with tile a in the upper left corner, such as that shown in Figure 70. Here we use a = white and b = black to generate checker boards such as those in

a	b	a	b	a	b
b	a	b	a	b	a
a	b	a	b	a	b
b	a	b	a	b	a
a	b	a	b	a	b
b	a	b	a	b	a

Figure 70. Checker board tiling with labeled tiles.

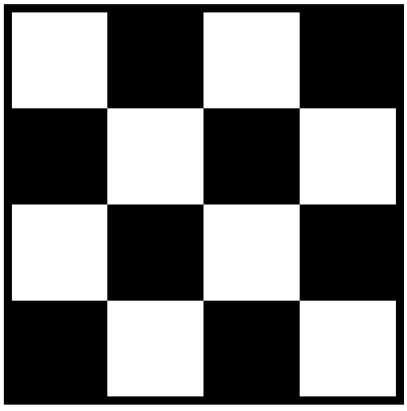


Figure 71. Checker board tiling with white and black tiles.

We introduce labels as a shorthand for our designers, noting that labels add no expressive power to 2RE and act only as a convenience. Atoms (or tiles) will be indicated with lower case letters or numbers, and labels will begin with an upper case letter and consist of only letters and numbers. In the next section, we describe how we use 2RE as generators.

2D Regular Expressions as Generators

In vision, 2RE were used to identify pictures; however, in PCGG we are interested in generation of tilings. There are two ways in which we can generate tilings. First, in a search-based approach, we can generate random samples and evaluate their suitability using a fitness function. Second, if the sample size is small enough, it is desirable to enumerate all possible tilings and find the best one. Given a 2RE (formally which describes a language L of tilings which is a subset of Σ**), we can either generate a random sample from L or enumerate all tilings. First we show how to generate; then we show how to enumerate.

Generating a Random Tiling

First we fix the dimensions to the user desired tiling size ($m \times n$) before generation. Given any regular expression R, parse R into an expression tree. Beginning with the root node, Let A be the left hand side and B be the right hand side. Recursively do the following based on the node operator:

$A \cup B$ Either generate a random tiling from A or generate a random tiling from B.

$A \cap B$ Either generate a random tiling from A and check if B accepts it or generate a random tiling from B and check if A accepts it.

A|B Generate a random tiling from A and generate a random tiling from B. Concatenate them horizontally. (Note that if the two tilings are not the same height, the resulting concatenation would not be a rectangular tiling. We avoid this result by constraining the tilings generated to be the same height).

A-B Generate a random element from A and generate a random tiling from B. Concatenate them vertically. (Note that if the two tilings are not the same width, the resulting concatenation would not be a rectangular tiling. We avoid this result by constraining the tilings generated to be the same width.)

A*| Generate between 0 and MAX tilings from A and concatenate them horizontally.

A*- Generate between 0 and MAX tilings from A and concatenate them vertically.

This process continues until you reach the leaf nodes of the expression tree in which the regular expressions are atoms. In this case, the operation is trivial.

Enumeration

As with generation, we fix the dimensions to the user desired tiling size ($m \times n$) before enumeration. Given any regular expression R, parse R into an expression tree. Beginning with the root node, Let A be the left hand side and B be the right hand side. Recursively do the following based on the node operator:

A∪B	Enumerate all elements in A. Then enumerate all elements in B.
A∩B	Either enumerate elements in A returning only those accepted by B, or enumerate elements in B returning only those accepted by A.
A|B	For each element a_i in A, enumerate all elements b_j in B, returning $(a_i\|b_j)$.
A-B	For each element a_i in A, enumerate all elements b_j in B, returning $(a_i\text{-}b_j)$.
A*|	Find all permutations of horizontal concatenations of elements of A that meet the width criteria of the picture and enumerate.
A*-	Find all permutations of vertical concatenations of A that meet the height criteria of the picture and enumerate.

In the next section, we show example regular expressions and generated tilings.

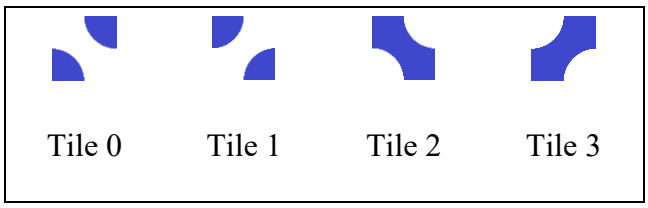


Figure 72. Truchet tile set.

Truchet Tiles

Using the above tiles from **Error! Reference source not found.**, the following regular expression describes a language that is a subset of Truchet tilings {named in honor of Father Sebastien Truchet (1657-1729)} [59]. Two random examples generated using the method described above are shown in Figure 73.

A=0∪2
B=1∪3
(A-B)|(B-A)*-*|

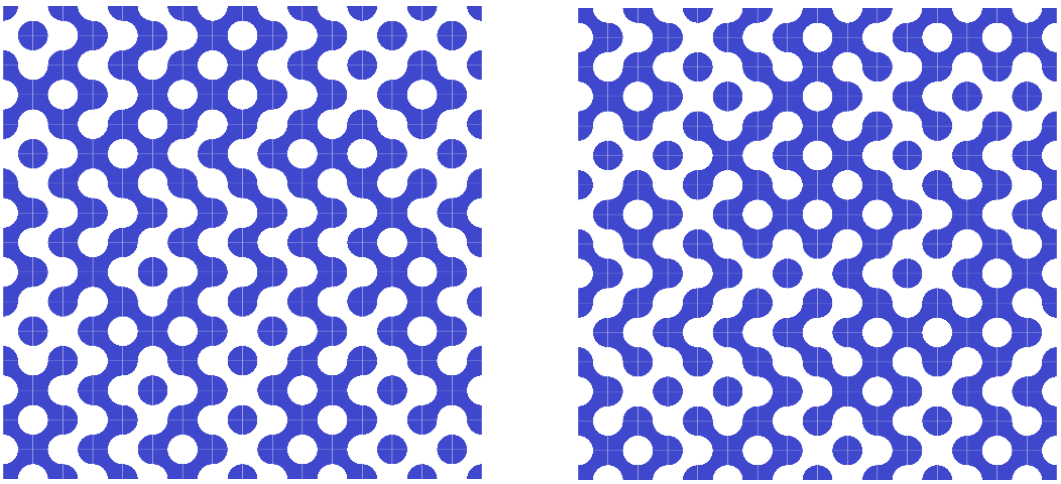


Figure 73. Two examples of Truchet tiling.

Jittered Grid

For natural distributions of plants, a Poisson disc sampling is often used. A quick way to approximate a Poisson disc sampling is with a jittered grid. A jittered grid of plants can be quickly generated using only 2 tiles: an empty tile and a tile with a plant. To generate

a jittered grid we place 1 plant tile in each 3 x 3 tile square. Without jittering, this plant will be placed in the lower left corner. We randomly jitter the plant one tile to the right or one tile up. Below shows the grammar for this jittered grid. Even though we only use 4 possible points for plant placement, a reasonable effect can be reached.

Regular expression for a jittered grid:

```
A=0|0|0                 // Row with no plant
B=1|0|0∪(0|1|0)         // Row with plant in first or
                        // second tile.
C=A-B-A∪(A-A-B)         // Concatenation of rows
C*|*-                   // Kleene * operations
```

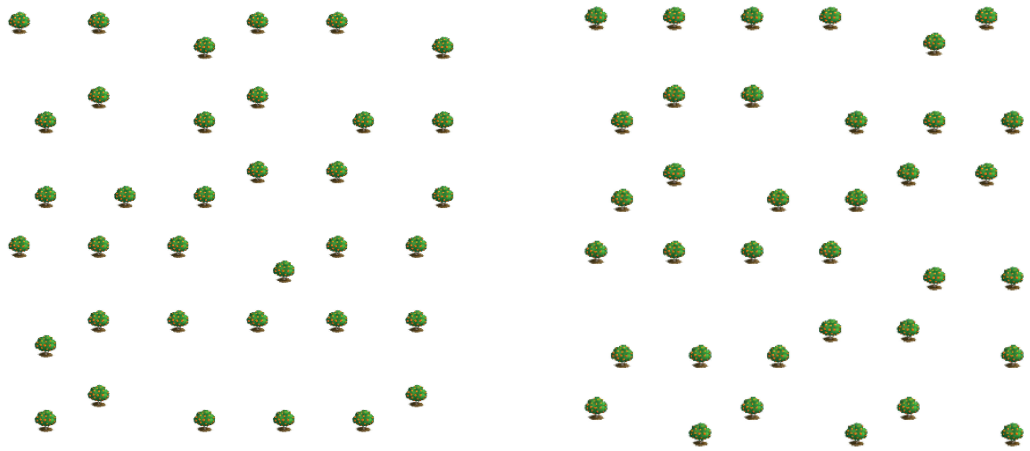


Figure 74. Two simulated Poisson disc samplings using a jittered grid.

Standard Tiling Patterns

One tile set we commonly use consists of 16 tiles. Each tile edge is either open or closed, making 2^4 possible combinations. The set is shown in Figure 75. Note, we prefer the tile with all edges closed as a solid black rectangle; however, it could also be unfilled as well.

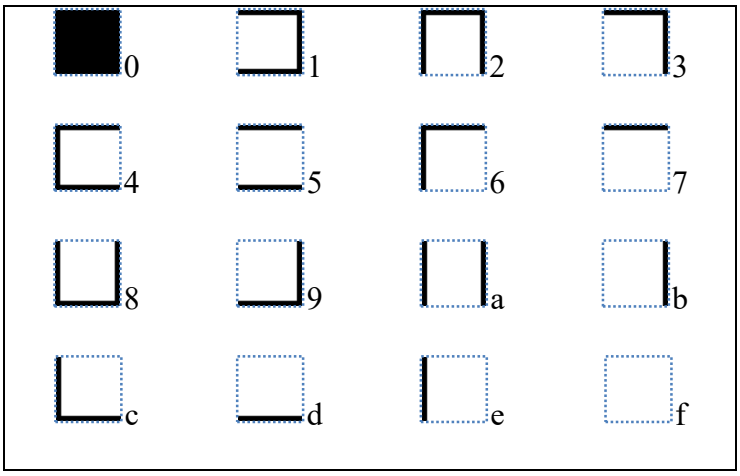


Figure 75. Standard tile set

We can use these tiles to show standard tiling patterns. For example, we can represent a horizontal brick with tiles 4 and 1. We can represent a vertical brick with tiles 2 and 8. Figure 76 shows two tilings for horizontal and vertical brick.

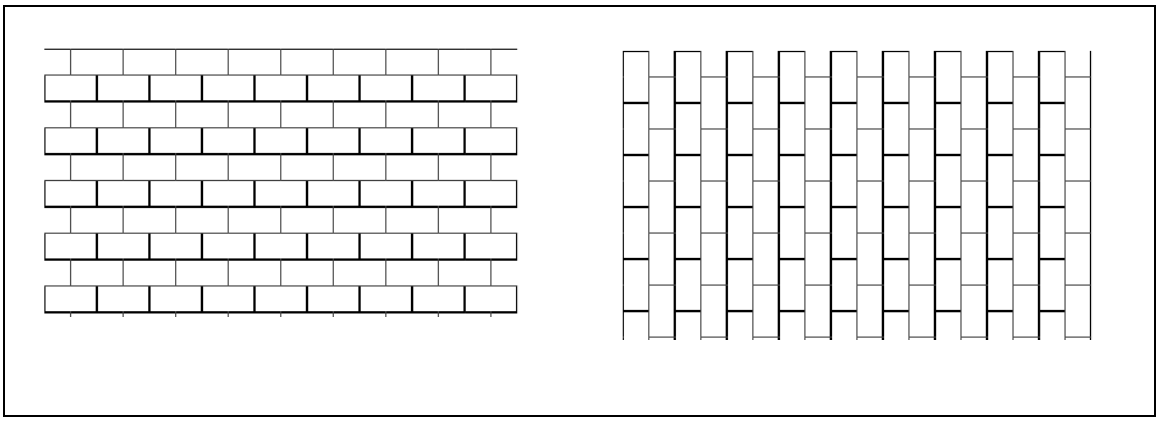


Figure 76. Tilings of horizontal and vertical brick.

Each of these tilings repeats a single pattern which is a 2 x 2 tiling. This is the same as the chess board with tile substitutions. Their 2D regular expressions are shown here.

Horizontal brick tiling	**Vertical brick tiling**
Q = 1|4	S = 2|8
R = 4|1	T = 8|2
(Q-R)*|*-	(S-T)*|*-

Here we present two more complex tiling patterns: a French pattern and a Herringbone pattern in Figure 77.

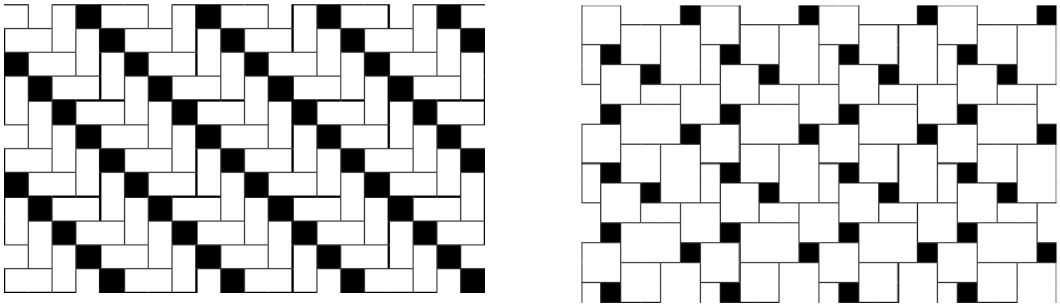


Figure 77. Herringbone tiling (left) and French tiling (right).

To generate these patterns with 2RE, we analyze the tilings to find the smallest period of repetition in each one. With Herringbone, if we count from the left along the top row, we see that the pattern repeats every 5 tiles. We get the same period of repetition if we count downward. We see that the Herringbone pattern can be constructed by a concatenating a tiling of 5 x 5. Similarly, with the French pattern, we see a 6 x 6 tiling. We write the 2RE for the meta-tiles and then use the Kleene* operations to extend the tiling. These are

shown below. In the 2RE for Herringbone, F is a 5 x 5 tiling. Similarly, in the 2RE for a French tiling, G is a 6 x 6 tiling.

Herringbone Pattern:	**French tiling**
A=1|8|2|0|4	A=6|3|c|d|9|0
B=4|1|8|2|0	B=c|9|6|3|6|3
C=0|4|1|8|2	C=2|0|c|9|e|b
D=2|0|4|1|8	D=8|6|3|0|c|9
E=8|2|0|4|1	E=3|c|9|4|1|6
F=A-B-C-D-E	F=9|0|6|7|3|c
F*|*-{*	G=A-B-C-D-E-F
	G*|*-

Array Grammars

Herein we explore the power of AG and their context sensitive nature. We show three examples of generation using AG that cannot be accomplished with 2RE.

Definition of Array Grammars

Array grammars are defined as a quintuple $G = (V, V_T, \Pi, s, \#)$ where:[4]

V is the vocabulary (a finite non-empty set of symbols)

$V_T \subset V$ is the terminal vocabulary ($V_T \neq \emptyset$).

$S \in (V - V_T)$ is the start symbol

[4] We are not interested in growing or shrinking pictures, so we ignore some detailed conditions on Ξ to prevent unconnected components. All of our pictures start out as a rectangular array of start symbols surrounded by #s. See [39] for more details.

is a special border symbol

Π, the rule set, is a non-empty set of pairs of rectangular arrays (Λ,Ξ) of elements of V where:

(a) Λ and Ξ are geometrically identical and

(b) Λ does not consist entirely of #s

An AG operates on a tiling by searching for applicable rules. The whole tiling is searched for any occurrence of the array Λ for each rule in Π. If more than one rule in Π matches or a rule matches more than one location, a location and rule is chosen out of all the matching locations and rules randomly. A rule is applied by replacing the occurrence of Λ in the tiling with Ξ. This is shown in detail in our first array grammar below.

We use AG to operate on tilings, which are rectangular arrays of elements of V. When a grammar completes, every element of the array is a member of V_T. Each element of V_T is mapped to a tile to generate a tiling. We always start with a rectangular array of start symbols (S), surrounded by a single row/column of border symbols (#). The size of the array of start symbols specifies the size of the desired tiling. Border symbols are never replaced and are removed once the grammar completes.

We use the following syntax to describe our rules.

array Λ := array Ξ

We specify arrays by listing the terminals and non-terminals in order from left to right. Commas are used to separate rows of the array. Upper case letters are used for non-terminals, and numbers and lower case letters for terminals. A simple grammar with 5 rules which generates the checkerboard pattern is shown below.

1. ##,#S:=##,#a
2. aS:=ab
3. bS:=ba
4. a,S:=a,b
5. b,S:=b,a

Figure 78 shows successive application of these rules on a 2x2 tiling with a border and an initial interior of all S's. The rules 1, 4, 2, and 5 were applied sequentially. Note that a different sequence of rules could have been applied, but the result would be the same. Rule 1 is the only valid initial rule.

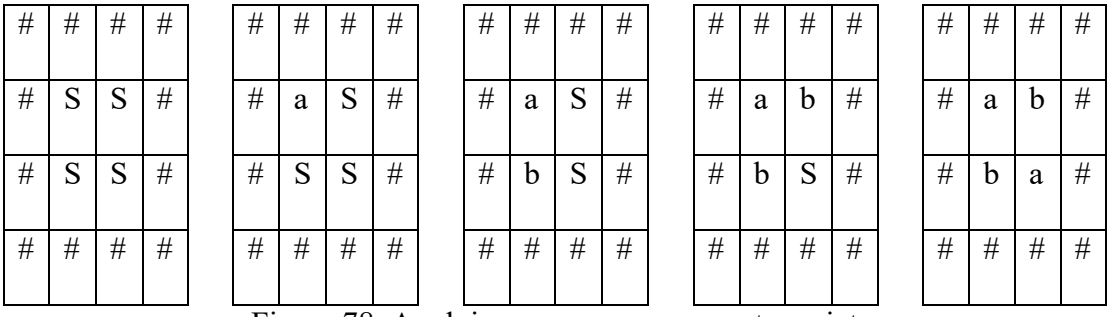


Figure 78. Applying an array grammar to a picture.

Often we wish to generate several similar rules. To keep grammars simple, we introduce regular expressions into our grammar notation. A set of tiles is indicated with square

brackets, e.g. the set {1, 3, 5} is represented as [135]. A range of tiles is indicated with square brackets and a dash, e.g. the set {1, 2, 3} is represented as [1-3].

The wildcard '?' stands for any symbol in *V* on the left-hand side of a rule and means "replace with self" on the right-hand side of a rule. Similar to the labels we introduced for 2RE, these shortcuts do not add expressiveness to our grammars and only serve as a shorthand for the designer to avoid writing many rules in place of one. An example is the following rule.

1S, ?1 := 12, ?1

This rule will replace the symbol S when there is a 1 on its left and a 1 below it. Without this shorthand, we would need several rules as follows:

1S, 11 := 12, 11

1S, 21 := 12, 21

1S, 31 := 12, 31

…

River Path

The first problem we approached is to see if we could use a grammar to define the river path in Recovery Rapids, a game for the rehabilitation of hemiparesis (see Chapter 5). Our design goal was to develop a river canyon in which the player navigates a kayak downriver while performing several game challenges. We desired a river which meandered across a terrain from west to east with some variation in its north/south travel.

The goal is to provide a pleasing and varying setting in which the player performs therapeutic gestures to trigger in-game mechanics.

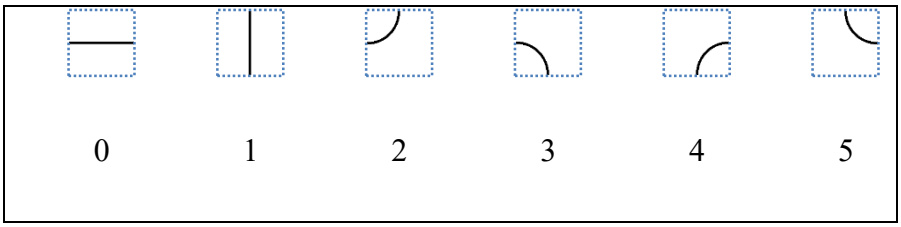


Figure 79. River Tiles

River segments are represented as a set of 6 tiles which indicate horizontal travel, vertical travel, and turning left and right. See Figure 79. The original design used a set of 9 super tiles composed of 3 rows and 2 columns of river segment tiles. The large tiles each progress the river 2 tile units east as shown in Figure 80

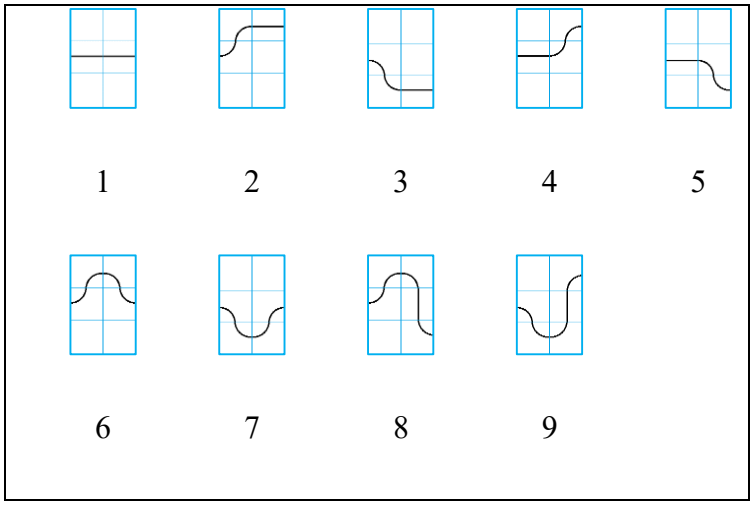


Figure 80. Super Tiles

In practice, the rule replacement algorithm is a search for all occurrences of the left-hand side of the rule (an $n \times m$ array) in a picture. In developing our grammars, we optimized for the size of each rule to optimize this search. The above grammar could easily be implemented with 9 rules using 3x3 arrays; however, we chose to use dominoes (1x2 and 2x1 arrays) instead. The rules for the first 3 super tiles are shown here.

```
1. [045]S:=?A   // Super Tile 1
2. AS:=00
3. [045]S:=?C   // Super Tile 2
4. S,C:=B,2
5. BS:=40
6. [045]S:=?E   // Super Tile 3
7. E,S:=3,D
8. DS:=50
   …
```

Note the use of non-terminals to provide intermediary steps to building each super tile. The grammar also contains rules to select the river starting point and to remove any remaining non-terminals in the picture. See Appendix A for the full listing of 43 rules. Example river paths are shown in Figure 81.

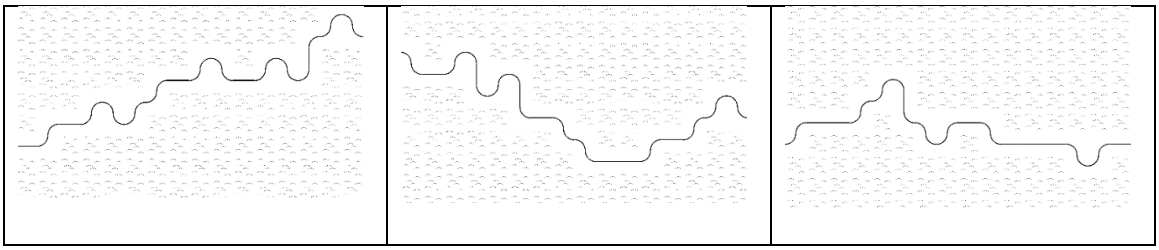


Figure 81. Some example river paths generated with a grammar.

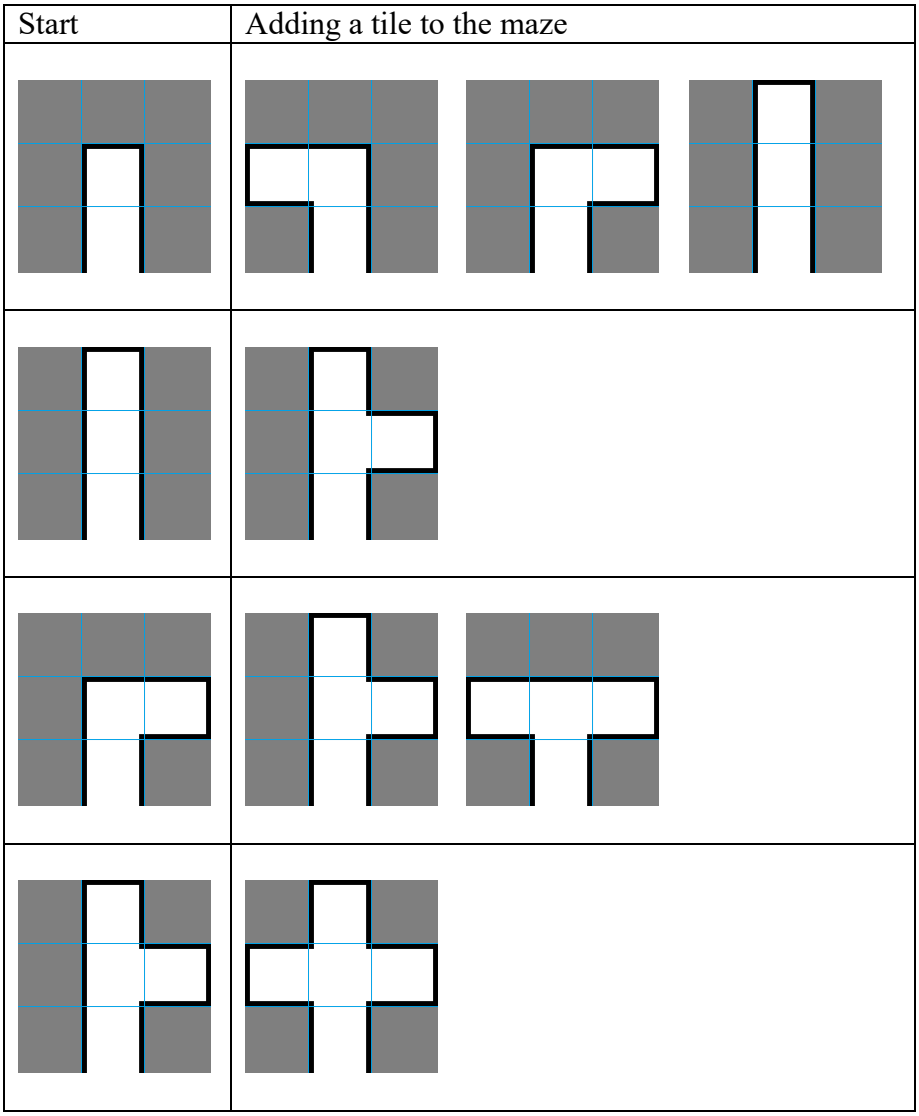


Figure 82. Growing a maze.

Genrating a Perfect Maze

A perfect maze using a 2D regular tiling is an embedding of a spanning tree in a 2D Cartesian grid. This implies no loops nor self-intersections, and every tile in the tiling is used. We choose to put the entrance in the upper left corner of the tiling. We choose to place the exit in the bottom right corner. Entrance and exit placement is an interesting sub-problem on its own; however, the focus here was on testing the feasibility of using a grammar, so we did not address entrance and exit locations.

We consider this problem as a logical extension of growing a river. In this case we wish to grow a tree. This is an implementation of Prim's algorithm implemented using an AG (compared to our implementation of Prim's in C# for Recovery Rapids shown in Chapter 6).

Figure 82 shows all the possible ways we may add one tile to an existing tree. Starting with the tiling on the left in each row, the right column shows all possible ways to add a tile (to the center tile). Every other possibility is a rotation of one of these 4 situations. Note that row 1 adds to the tree without creating a new branch. Rows 2, 3, and 4 create a new branch in the tree.

Using the tile set from Figure 75, here is a partial listing of the grammar for a growing maze showing some of the growing and branching rules as well as the start and end rules. The full listing of 32 rules can be found on the author's website.

```
1. ##,#S:=??,?1 // Begin maze
2. S,1:=2,9          // Grow upward
3. 1,S:=3,8          // Grow downward
```

```
4. 1S:=51       // Grow right
5. 3S:=71       // Branch right
6. S,3:=2,b     // Branch forward (up)
7. S,5:=2,d          // Branch up
   …
8. 1#,##:=5?,?? // Punch exit in bottom left corner
9. 8#,##:=c?,??
10.  9#,##:=d?,??
```

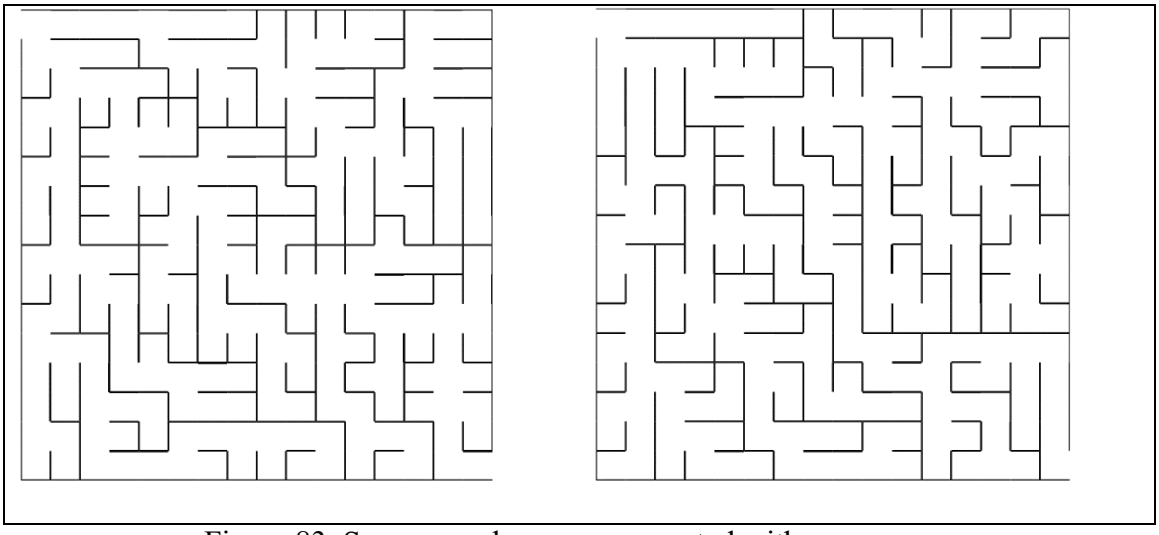


Figure 83. Some sample mazes generated with a grammar.

This grammar can easily be modified to change the characteristics of the resultant maze. For example, we may wish to emphasize rules which add to the existing tree without creating a new branches (as shown in Figure 82, row 1). Since a rule is randomly selected when more than one apply to the tiling, we can duplicate a rule so there is more

chance it is selected. After emphasizing such rules, the resulting maze is shown in Figure 84. This maze has a longer solution path (55 tiles) than those shown in Figure 83 (31 and 34 tiles for the left and right mazes respectively).

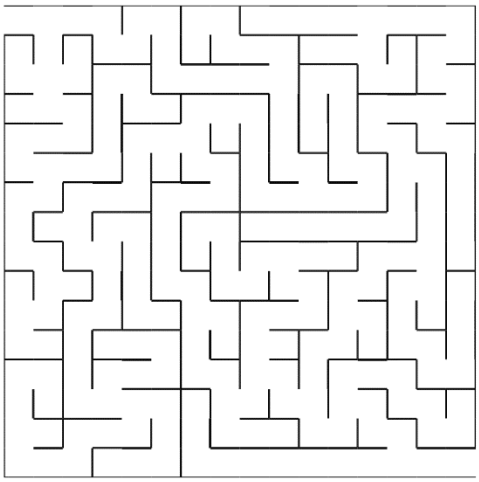


Figure 84. Generated maze with longer solution path.

Wang Tiling for Terrain

So far, we have shown several examples of Wang tiling in PCGG. We would like to know if our tools, 2RE and AG support Wang tiling in general. So far in our experience, we have not been able to directly implement a simple Wang tiling oracle such as edge matching with 2RE. See *Evaluation* below. It intuitively seems as though Array Grammars will work better for us because grammar rules are based on a local context of neighboring tiles. Often we don't want a complete set of Wang tiles, but a specially defined subset of Wang tiles fits our purpose such as in the next example. We generate a 3-ecosystem terrain of water, grass, and desert. Edge colors are created for tiles which

transition from water to grass and from grass to desert across the edge. This yields a total of 7 colors. For a complete Wang tile set with 7 colors we would have to create 7^4 or 2401 tiles. Instead we choose a tile set of just 27 tiles as shown here:

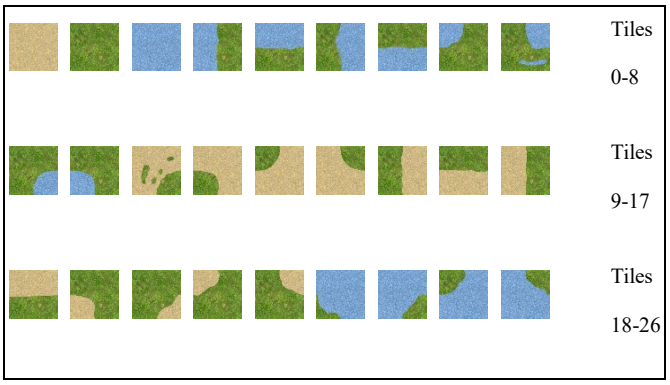


Figure 85. Wang tile set for generating a seamless terrain of water to grass to mountain.

If we start the tile generation in the upper left-hand corner and proceed top-down and left to right, for each new tile we lay down, we need to make sure the top and left colors match the adjacent tiles. With an incomplete tile set, however, there is the additional difficulty of making sure we don't place a tile that would generate an impossible match in the future. For example, with the above tile set, we create a 4x4 picture tiling from top down and left to right. At some point we could arrive at the state in Figure 86.

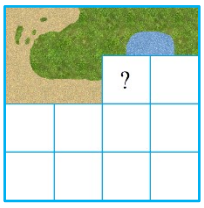


Figure 86. Stuck while tiling from top down and left to right.

There is no tile in our tile set that contains all 3 ecosystems; hence no tile will fit at the next location. We resolve this by using larger grammar rules to look ahead. By using 2 x 3 rules, we can examine the tile above and to the right of the tile currently being placed and avoid placing tiles that generate this conflict. The rules examine the tiles in the red rectangle shown in Figure 87. Any rules which would place a tile where the "?" symbol is and result in a situation where we could not continue tiling was discarded from the array grammar.

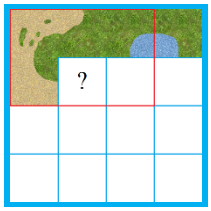


Figure 87. Using a 2 x 3 rule to tile the terrain.

Listing 3 shows some of the rules from terrain generation. Tiles are numbered 0-9, then a-q. The full listing, containing the 138 rules, can be found in Appendix A. Two example terrains generated from this grammar are shown in Figure 88. Green represents vegetation, blue represents water, and brown represents an arid landscape. Our array grammar maintained that water would be surrounded by vegetation as well as the standard Wang Tiling rules.

```
1. // Rule to place a tile at the start position.
   ##,#S:=##,#[0123456789abcdefghi]
2. // For top row, match tiles with sand on left edge
   ##,[0cdf]S:=??,?[0beh]
3. // For top row, match tiles with grass on left edge
   ##,[137ahjl]S:=??,?[1589fkm]
4. // Starting a new row.  Match tiles with sand on the
   // bottom (followed by tile with sand on the bottom).
   #[0deg][0deg],#S?:=???,?[0bci]?
5. // Starting a new row.  Match tiles with sand on the
   // bottom (followed by a tile with sand and grass on
   // the bottom)
   #[0deg][bhj],#S?:=???,?[0bci]?
```

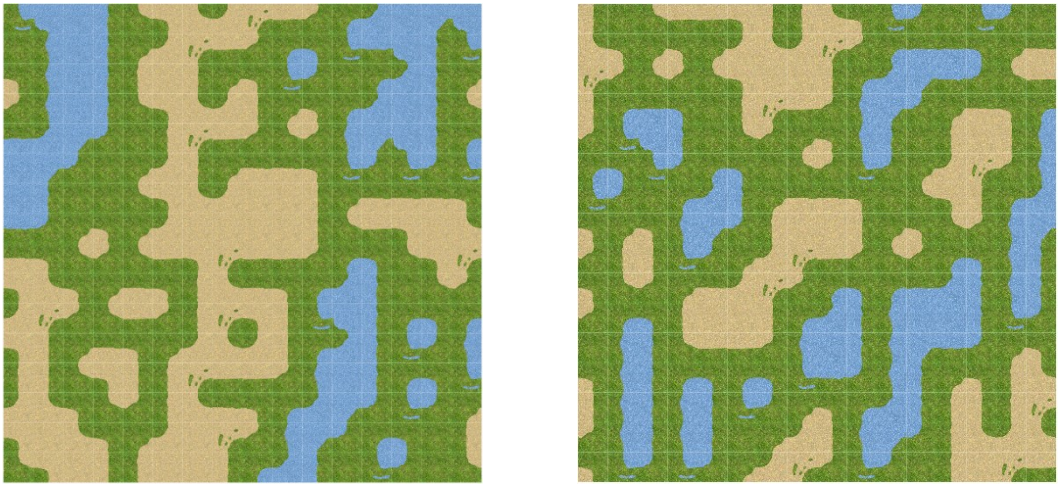


Figure 88. Terrains created with an array grammar.

Again, we can modify the grammar to tune the results of the PCGG. For example, we could make sure that the border of the tiling is not adjacent to land or grass. Modifying the grammar to do this produces island terrains such as those in Figure 89. This generally involved removing rules such as those which placed land or grass near the left edge. The new terrain generation AG is shown in Appendix A, Listing 3.

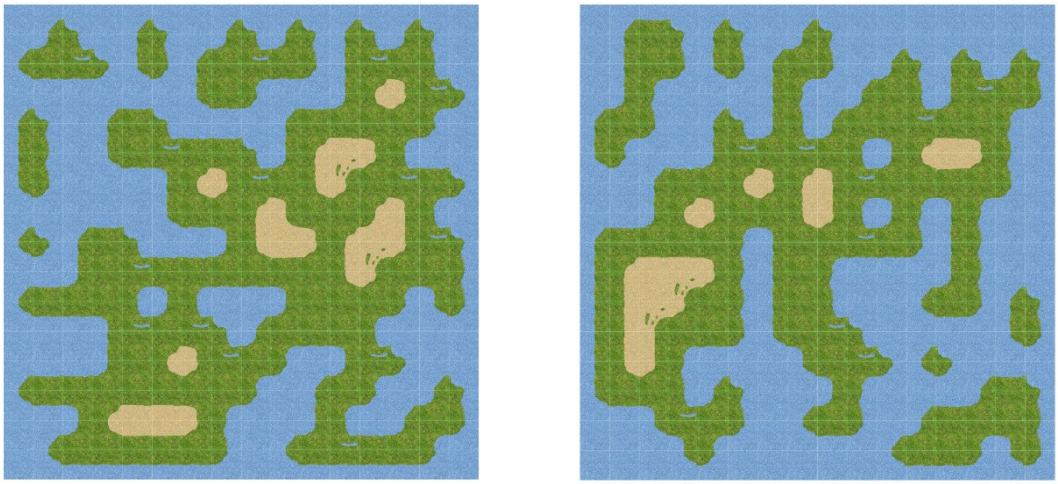


Figure 89. Island terrains generated with an array grammar.

Evaluation

2D Regular Expressions

2D Regular expressions are able to generate some regular patterns and do well for random distributions such as a distribution of trees in a forest or potholes in a road. Combined, these abilities work well for regular content or content with small variety. For

example, we can generate patterned brick walls with random imperfections such as chips and cracks or a paved road with occasional weeds and potholes.

The enumeration ability of 2RE is a powerful advantage over standard search-based techniques. With enumeration we can guarantee we find the optimal solution.

Unfortunately, there are also some limitations of 2RE.

We attempted to specify a seamless tiling using our standard set of 16 tiles above in 2RE.

As a first attempt, we define the possible matches in a 1 x 2 tiling:

2D Regular Expression for Wang Tiling (1 x 2)

A=0∪2∪4∪6∪8∪a∪c∪e (All tiles closed on the left)
B =1∪3∪5∪7∪9∪b∪d∪f (All tiles open on the left)
C =0∪1∪2∪3∪8∪9∪a∪b (All tiles closed on the right)
D= 4∪5∪6∪7∪c∪d∪e∪f (All tiles open on the right)
(C|A) ∪(D|B)

We immediately run into trouble when we wish to extend this to a 1x3 tiling. The resulting tiling has either a tile from set A or set B on the right. But A∪B is all tiles, meaning we have tiles with both edge colors (closed and open) on the right! We cannot simply concatenate to our existing expression. We can handle the situation with intersection by using the above labels and adding the following to our expression:

S=A∪B
((C|A|S) ∪(D|B|S))∩((S|C|A) ∪(S|D|B))

Expanding on this relationship, we generate the following expression for a 3x3 Wang tiling:

S=0∪1∪2∪3∪4∪5∪6∪7∪8∪9∪a∪b∪c∪d∪e∪f
A=1∪3∪5∪7∪9∪b∪d∪f
B=0∪2∪4∪6∪8∪a∪c∪e
C=0∪1∪2∪3∪8∪9∪a∪b
D=4∪5∪6∪7∪c∪d∪e∪f
E=0∪1∪4∪5∪8∪9∪c∪d

F=2∪3∪6∪7∪a∪b∪e∪f
G=0∪1∪2∪3∪4∪5∪6∪7
H=8∪9∪a∪b∪c∪d∪e∪f
L1=((C|B)∪(D|A)|S)∩(S|((C|B) ∪ (D|A)))
T1=L1-L1-L1
L2=((E-G)∪(F-H)-S)∩(S-((E-G)∪(F-H)))
T2=L2|L2|L2
T1∩T2

There are two problems with this approach. First, we can see as the size of the tiling grows, the regular expression becomes so large as to be unwieldy for a human designer. The second is the intersection operator. Using our method of generate from the left-hand side of the intersection and test against the right-hand side (or vice versa), with a 3x3 we have a reasonable chance to find a member of the intersection, but this quickly becomes infeasible. For a 3x3, there are 12 (dependent pairs of) internal edges and 12 independent external edges for a total of 2^{24} possible valid Wang Tilings using our standard 2 color tile set. In the expression labeled T1 (or T2) above, we have 6 internal edges and 24 external edges for 2^{30} possible tilings. We have a roughly 1 in 64 chance of generating a valid tiling. With a 10 x 10 tiling, this would quickly degenerate to 1 in 2^{80}. It seems 2RE is incapable of specifying the set of valid tilings given a Wang tile set with any reasonable sized expression. We showed that this can be done with AG both in the maze example and in the terrain example.

Array Grammars

With AG, addition or subtraction of rules make significant changes to the types of tilings produced. We showed with mazes that we can change the AG to change the characteristics of the resultant maze, such as the increased length of solution path in

Figure 84. In our terrain example, we created islands by only allowing water to touch the border of the image. With a little practice it is not difficult to see how to add or remove rules to tune the results. AG also have the benefit of being able to easily deal with local context. Unlike 2RE, it is not clear that AG can easily be used for enumeration of tilings.

Chapter 8. Conclusion and Future Work

Procedural Content Generation for Games (PCGG) is becoming more prevalent in the development of games and impact of research in this area should only increase. Our hope is that this body of work will promote and facilitate the development of tile-based PCGG. With our framework we provide a starting point for the software development process. Beginning with our software framework and maintaining a good design should promote software reuse amongst and between teams as more projects are undertaken.

With bandwidth limited noise tiles, we provide a fundamental tool for the generation of tile-based PCGG. Perlin noise has been a workhorse of the graphics industry. One of our first tasks was to rework Perlin noise to be able to generate tiles which could be used in seamless tilings. We have shown their application in the generation of textures such as wood and marble, in height maps, as turbulence, and as varying frequency noise.

We promote the use of 2D Regular Expressions (2RE) and Array Grammars (AG) in PCGG. Each has its merits which it brings to the game designer's toolbox. 2RE proved useful, especially for generating patterns in tilings. These include patterns with variability such as the jittered grid, or Truchet tilings with interchangeable tiles. We were unable to implement an oracle for a Wang tiling efficiently and 2RE did not prove useful for content in which tiling depends on local context. AG's, due to their context sensitive nature, are more capable and intuitive than regular expressions. We have shown that in

theory, AG's are as capable as any other known computational model when it comes to 2D pictures which are bounded in at least one dimension. We have only begun to explore the capabilities of what AG have to offer. Future research asking AG to accomplish more complex tasks such as generating balanced levels for turn based strategy games could be done.

Finally, Recovery Rapids served as a platform for tile-based PCGG research. Our limited resources as well as the long therapy treatment time necessitated the use of PCGG. A tile-based strategy allowed us to provide therapy in a proscribed fashion. To prevent boredom, proscribed therapy can be achieved with a different tiling each time to provide variety. It also allows us to individualize the therapy by changing the proportion of each therapeutic activity to meet the individual player's needs.

Not only is Recovery Rapids a platform for PCGG research, but also it fills a deficiency in current outpatient treatment of hemiparesis. Future work involves an on-going 3 year efficacy study and efforts to improve the personalization of the therapy through data analytics and feedback. Dr. Gauthier, Dr. Crawfis and I are working to commercialize and distribute this software through a public benefit corporation.

References

[1] Perlin, K., "An Image Synthesizer," in *Proceedings of the 12th annual conference on computer graphics and interactive techniques (SIGGRAPH '85)*, New York, NY, USA, 1985.

[2] Perlin, K., "Improving Noise," *ACM Trans. Graph.,* vol. 21, no. 3, pp. 681-682, July 2002.

[3] Stam, J., "Aperiodic Texture Mapping," European Research Consortium for Informatics and Mathematics, 1997.

[4] Zucker, M., "Perlin Noise Math FAQ," 2001. [Online]. Available: http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html. [Accessed 22 05 2012].

[5] Perlin, K. and Hoffert, E., "Hypertexture," in *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 1989.

[6] Lagae, A., Lefebvre, S., Cook, R., DeRose, T., Drettakis, G., Ebert, D., Lewis, J., Perlin, K., and Zwicker, M., "A Survey of Procedural Noise Functions," *Computer Graphics Forum,* vol. 29, no. 8, pp. 2579-2600, 2010.

[7] Cohen, M., Shade, J., Hiller, S. and Oliver, D., "Wang Tiles for image and texture generation," New York, NY, USA, 2003.

[8] Cook, R. and DeRose, T., "Wavelet Noise," in *ACM SIGGRAPH 2005 Papers*, New York, NY, USA, 2005.

[9] A. Goldberg, M. Zwicker and F. Durand, "Anisotropic Noise," in *ACM SIGGRAPH 2008 papers*, New York, NY, USA, 2008.

[10] A. Lagae and P. Dutré, "An alternative for Wang tiles: colored edges versus colored corners," *ACM Trans. Graph.,* vol. 25, no. 4, pp. 1442-1459, October 2006.

[11] A. Lagae, S. Lefebvre, G. Drettakis and P. Dutré, "Procedural noise using sparse Gabor convolution," *ACM Trans. Graph.,* vol. 28, no. 3, pp. 54:1-54:10, July 2009.

[12] Morris DM, Uswatte G, Crago JE, Cook EW, 3rd, Taub E. The reliability of the wolf motor function test for assessing upper extremity function after stroke. Arch Phys Med Rehabil. 2001;82:750-5.

[13] Brewer B, McDowell SK, Worthen-Chaudhari L. Post stroke Upper Extremity Rehabilitation: A Review of Robotic Systems and Clinical Results. Top Stroke Rehabil; 14(6):22-44.

[14] Wolf SL, Catlin PA, Ellis M, Archer AL, Morgan B, Piacentino A. Assessing Wolf motor function test as outcome measure for research in patients after stroke. Stroke. 2001;32:1635-9.

[15] Gauthier LV, Taub E, Perkins C, Ortmann M, Mark VW, Uswatte G. Remodeling the brain: plastic structural brain changes produced by different motor therapies after stroke. Stroke. 2008; 39:1520-5

[16] Jun-Da Huang. 2011. Kinerehab: a Kinect-based system for physical rehabilitation: a pilot study for young adults with motor disabilities. In The proceedings of the 13th international ACM SIGACCESS conference on Computers and accessibility (ASSETS '11). ACM, New York, NY, USA, 319-320. DOI=10.1145/2049536.2049627 http://doi.acm.org/10.1145/2049536.2049627

[17] Maung D, Crawfis R, Gauthier L, Worthen-Chaudhari L, Lowes L, Borstad A, McPherson R. Games for Therapy: Defining a Grammar and Implementation for the Recognition of Therapeutic Gestures. In the proceedings of the 8th International Conference on the Foundations of Digital Games. 2013. Chania, Greece. http://www.fdg2013.org/program/papers.html.

[18] Fua K, Gupta S, Pautler D, Farber I. Designing serious games for elders. In the proceedings of the 8th International Conference on the Foundations of Digital Games. 2013. Chania, Greece. http://www.fdg2013.org/program/papers.html.

[19] Kelly-Hayes M, Beiser A, Kase CS, Scaramucci A, D'Agostino RB, Wolf PA. The influence of gender and age on disability following ischemic stroke: the Framingham study. J Stroke Cerebrovasc Dis. 2003; 12:119-126.

[20] Maung D, Shi Y, Crawfis R. Procedural Textures Using Tilings with Perlin Noise. Computer Games (CGAMES), 2012 17th International Conference on, 30 July-2 Aug 2012.

[21] Centers for Disease Control and Prevention (CDC). Prevalence of disabilities and associated health conditions among adults--United States, 1999 MMWR Morb Mortal Wkly Rep. 2001;50:120-125.

[22] Adams E, Dormans J. 2012. Game Mechanics: Advanced Game Design. New Riders Publishing, Thousand Oaks, CA, USA.

[23] Taub E, Miller NE, Novack TA, Cook EW,3rd, Fleming WC, Nepomuceno CS, Connell JS, Crago JE. Technique to improve chronic motor deficit after stroke. Arch Phys Med Rehabil. 1993; 74:347-54.

[24] Lloyd-Jones D, Adams R, Carnethon M, De Simone G, Ferguson TB, Flegal K, et al. Heart disease and stroke statistics--2009 update: a report from the American Heart Association Statistics Committee and Stroke Statistics Subcommittee Circulation. 2009;119:480-486.

[25] Heart and Stroke Foundation of Ontario Centre of Excellence in Stroke Recovery. Stroke rehabilitation consensus panel report; 2000. 2000.

[26] Duncan PW, Zorowitz R, Bates B, Choi JY, Glasberg JJ, Graham GD, et al. Management of Adult Stroke Rehabilitation Care: a clinical practice guideline. Stroke. 2005;36:e100-43

[27] Lang CE, Macdonald JR, Reisman DS, Boyd L, Jacobson Kimberley T, Schindler-Ivens SM, et al. Observation of amounts of movement practice provided during stroke rehabilitation. Arch Phys Med Rehabil. 2009;90:1692-1698.

[28] Taub E, Uswatte G, Pidikiti R. Constraint-Induced Movement Therapy: a new family of techniques with broad application to physical rehabilitation-a clinical review. 1999;36:237-251.

[29] Wolf SL, Winstein CJ, Miller JP, Taub E, Uswatte G, Morris D, et al. Effect of constraint-induced movement therapy on upper extremity function 3 to 9 months after stroke: the EXCITE randomized clinical trial. JAMA. 2006;296:2095-2104.

[30] Taub E, Uswatte G, Mark VW, Morris DM. The learned nonuse phenomenon: implications for rehabilitation. Eura Medicophys. 2006;42:241-56.

[31] Taub E, Uswatte G, Mark VW, Morris DM, Barman J, and Bowman MH, et al. Method for enhancing real-world use of a more affected arm in chronic stroke: transfer package of constraint-induced movement therapy. Stroke. 2013;44:1383-1388.

[32] Cohen, M. F., Shade, J., Hiller, S., & Deussen, O. (2003). Wang tiles for image and texture generation (Vol. 22, No. 3, pp. 287-294). ACM.

[33] Giammarresi, D., and Restivo, A. (1997). Two-dimensional languages. In Handbook of formal languages (pp. 215-267). Springer Berlin Heidelberg.

[34] Hanan, J., & Prusinkiewicz, P. (1993). Parametric L-systems and their application to the modelling and visualization of plants. University of Regina.

[35] Maung, D., Shi, Y., & Crawfis, R. (2012, July). Procedural textures using tilings with Perlin Noise. In Computer Games (CGAMES), 2012 17th International Conference on (pp. 60-65). IEEE.

[36] Maung, D.; Crawfis, R.; Gauthier, L.; Worthen-Chaudhari, L.; Lowes, L.; Borstad, A.; McPherson, R.; Grealy, J.; Adams, J. "Development of Recovery Rapids - A Game for Cost Effective Stroke Therapy", Foundations of Digital Games (FDG), 2014, 3-5 April

[37] Minecraft. (2009) https://minecraft.net.

[38] Rogue (1983) http://www.mobygames.com/game/rogue.

[39] Rosenfeld, A. (2014). Picture languages: formal models for picture recognition. Academic Press.

[40] Shaker, N., Togelius, J., and Nelson, M. (2014). Procedural Content Generation in Games. http://www.pcgbook.com.

[41] Smith, G., Treanor, M., Whitehead, J., & Mateas, M. (2009, April). Rhythm-based level generation for 2D platformers. In Proceedings of the 4th International Conference on Foundations of Digital Games (pp. 175-182). ACM.

[42] Sokoban. (1982) http://www.sokoban.jp/

[43] Yoav I. H. Parish and Pascal Müller. 2001. Procedural modeling of cities. In Proceedings of the 28th annual conference on Computer graphics and interactive techniques (SIGGRAPH '01). ACM, New York, NY, USA, 301-308. DOI=10.1145/383259.383292 http://doi.acm.org/10.1145/383259.383292

[44] 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* (PCGames '10). ACM, New York, NY, USA, ,

Article 1 , 8 pages. DOI=10.1145/1814256.1814257 http://doi.acm.org/10.1145/1814256.1814257.

[45] Silva, P., Muller, P., Bidarra, R., and Coelho, A. (2013). Node-based shape grammar representation and editing. In Workshop Proceedings of the 8th International Conference on the Foundations of Digital Games. http://www.fdg2013.org/program/workshops/papers.html.

[46] Barnes, T., Bogost, I, Proceedings of the 9th International Conference on the Foundations of Digital Games. (Ft. Lauderdale, Florida, 2014), http://www.fdg2014.org/proceedings.html

[47] Gavin S P Miller. 1986. The definition and rendering of terrain maps. *SIGGRAPH Comput. Graph.* 20, 4 (August 1986), 39-48. DOI=10.1145/15886.15890 http://doi.acm.org/10.1145/15886.15890.

[48] F. K. Musgrave, C. E. Kolb, and R. S. Mace. 1989. The synthesis and rendering of eroded fractal terrains. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '89). ACM, New York, NY, USA, 41-50. DOI=10.1145/74333.74337 http://doi.acm.org/10.1145/74333.74337

[49] Doran, J., & Parberry, I. (2010). Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games*, *2*(2), 111-119.

[50] Frade, M., de Vega, F. F., & Cotta, C. (2010). Evolution of artificial terrains for video games based on accessibility. In *Applications of evolutionary computation* (pp. 90-99). Springer Berlin Heidelberg.

[51] Frade, M., de Vega, F. F., & Cotta, C. (2010, July). Evolution of artificial terrains for video games based on obstacles edge length. In *Evolutionary Computation (CEC), 2010 IEEE Congress on* (pp. 1-8). IEEE.

[52] Zhou, H., Sun, J., Turk, G., & Rehg, J. M. (2007). Terrain synthesis from digital elevation models. *Visualization and Computer Graphics, IEEE Transactions on*, *13*(4), 834-848.

[53] Lengyel, E. Voxel-based terrain for real-time virtual simulations University of California, Davis, ProQuest, UMI Dissertations Publishing, 2010. 3404919.

[54] Fausto Mourato, Manuel Próspero dos Santos, and Fernando Birra. 2011. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology* (ACE '11), ACM, New York, NY, USA, , Article 8 , 8 pages. DOI=10.1145/2071423.2071433 http://doi.acm.org/10.1145/2071423.2071433

[55] Lawrence Johnson, Georgios N. Yannakakis, and Julian Togelius. 2010. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games* (PCGames '10). ACM, New York, NY, USA, , Article 10, 4 pages. DOI=10.1145/1814256.1814266 http://doi.acm.org/10.1145/1814256.1814266

[56] Kenneth Hullett and Jim Whitehead. 2010. Design patterns in FPS levels. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games* (FDG '10). ACM, New York, NY, USA, 78-85. DOI=10.1145/1822348.1822359 http://doi.acm.org/10.1145/1822348.1822359

[57] "Fractal weeds". Licensed under Public Domain via Wikimedia Commons - http://commons.wikimedia.org/wiki/File:Fractal_weeds.jpg#/media/File:Fractal_weeds.jpg

[58] Darwyn R. Peachey. 1985. Solid texturing of complex surfaces. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (SIGGRAPH '85). ACM, New York, NY, USA, 279-286. DOI=10.1145/325334.325246 http://doi.acm.org/10.1145/325334.325246

[59] C S Smith, The tiling patterns of Sebastien Truchet and the topology of structural hierarchy, Leonardo, Vol.20, pp.373-385, 1987.

[60] Dormans J. Level design as model transformation: A strategy for automated content generation. In Proceedings of the Procedural Content Generation Workshop 2011, Bordeaux, France, 2011.

[61] Wu, F. Y. "Number of spanning trees on a lattice." Journal of Physics A: Mathematical and General 10.6 (1977): L113.

[62] P. H. Kim and R. Crawfis, "The quest for the perfect perfect-maze," *Computer Games: AI, Animation, Mobile, Multimedia, Educational and Serious Games (CGAMES), 2015*, Louisville, KY, 2015, pp. 65-72. doi: 10.1109/CGames.2015.7272964

[63] Lorensen, W. E.; Cline, Harvey E. (1987). "Marching cubes: A high resolution 3d surface construction algorithm". ACM Computer Graphics 21 (4): 163–169. doi:10.1145/37402.37422"

[64] Random Hex Terrain II." Web log post. *Random Diversions*. N.p., 18 June 2012. Web. 26 Mar. 2016. <http://rnd-diversions.blogspot.com/2012/06/random-hex-terrain-ii.html>.

[65] Prim, Robert Clay. "Shortest connection networks and some generalizations." Bell system technical journal 36.6 (1957): 1389-1401.

[66] Gauthier, L., Borstad, A., Lowes, L., Worthen-Chaudhari, L., Crawfis, R., Jaffe, J., & Maung, D. (2014). Delivery of Constraint-Induced Movement Therapy Through a Video Game: a Pilot Study in Stroke. Archives of Physical Medicine and Rehabilitation, 10(95), e8-e9.

[67] Cohen J. A power primer. *Psychological bulletin.* 1992;112(1):155

[68] Van der Lee JH, Beckerman H, Knol DL, De Vet HCW, Bouter LM. Clinimetric properties of the motor activity log for the assessment of arm use in hemiparetic patients. *Stroke.* 2004;35(6):1410.

[69] Miltiadou, Milto. "Wang Tiles." Web log post. *Art & M@thS*. N.p., 1 July 2012. Web. 28 Mar. 2016.

[70] Boggus, Matt. *Modeling, Evaluation, Editing, and Illumination of Three Dimensional Mazes and Caves for Computer Games*. Thesis. The Ohio State University, 2012. N.p.: n.p., n.d. Print.

[71] Togelius, J., Kastbjerg, E., Schedl, D., Yannakakis, G.N.: What is procedural content generation?: Mario on the borderline. In: Proceedings of the 2nd Workshop on Procedural Content Generation in Games (2011)

[72] Grünbaum, B., Shepard, G. C. 1987. Tilings and Patterns. W.H. Freeman and Company. ISSN 0716711931.

[73] Cook, Robert L. "Stochastic sampling in computer graphics." ACM Transactions on Graphics (TOG) 5.1 (1986): 51-72.

Appendix A. Listings

Listing 1: Array Grammar for maze generation

```
1.      **,*S:=??,?1
2.      1S:=51
3.      S,1:=2,9
4.      1,S:=3,8
5.      2S:=61
6.      S2:=43
7.      S,2:=2,a
8.      3S:=71
9.      S,3:=2,b
10.     S4:=45
11.     S,4:=2,c
12.     4,S:=6,8
13.     S,5:=2,d
14.     5,S:=7,8
15.     S6:=47
16.     S,6:=2,e
17.     S,7:=2,f
18.     8S:=c1
19.     S8:=49
20.     8,S:=a,8
21.     9S:=d1
22.     9,S:=b,8
23.     aS:=e1
24.     Sa:=4b
25.     bS:=f1
26.     Sc:=4d
27.     c,S:=e,8
28.     d,S:=f,8
29.     Se:=4f
30.     1*,**:=5?,??
31.     8*,**:=c?,??
32.     9*,**:=d?,??
```

Listing 2. Array Grammar for Terrain Generation

```
1.      **,*S:=**,*[0123456789abcdefghi]
2.      ***,[0cdf]SS:=???,?[0beh]?
3.      ***,[137ahjl]SS:=???,?[1589fkm]?
4.      ***,[25np]SS:=???,?[23oq]?
5.      ***,[48o]SS:=???,?[47n]?
6.      ***,[69q]SS:=???,?[6ap]?
7.      ***,[bim]SS:=???,?[cil]?
8.      ***,[egk]SS:=???,?[dgj]?
9.      ***,[0cdf]S*:=???,?[0beh]?
10.     ***,[137ahjl]S*:=???,?[1589fkm]?
11.     ***,[25np]S*:=???,?[23oq]?
12.     ***,[48o]S*:=???,?[47n]?
13.     ***,[69q]S*:=???,?[6ap]?
14.     ***,[bim]S*:=???,?[cil]?
15.     ***,[egk]S*:=???,?[dgj]?
16.     *[0deg][0deg],*S?,*SS:=???,?[0bci]?,???
17.     *[0deg][bhj],*S?,*SS:=???,?[0bci]?,???
18.     *[1478ilm][1478ilm],*S?,*SS:=???,?[169agjk]?,???
19.     *[1478ilm][59n],*S?,*SS:=???,?[169aj]?,???
20.     *[1478ilm][cfk],*S?,*SS:=???,?[1agjk]?,???
21.     *[26pq][26pq],*S?,*SS:=???,?[24no]?,???
22.     *[26pq][3ao],*S?,*SS:=???,?[24no]?,???
23.     *[3ao][1478ilm],*S?,*SS:=???,?[37q]?,???
24.     *[3ao][59n],*S?,*SS:=???,?[37q]?,???
25.     *[3ao][cfk],*S?,*SS:=???,?[37]?,???
26.     *[59n][26pq],*S?,*SS:=???,?[58p]?,???
27.     *[59n][3ao],*S?,*SS:=???,?[58p]?,???
28.     *[bhj][1478ilm],*S?,*SS:=???,?[ehl]?,???
29.     *[bhj][59n],*S?,*SS:=???,?[hl]?,???
30.     *[bhj][cfk],*S?,*SS:=???,?[ehl]?,???
31.     *[cfk][0deg],*S?,*SS:=???,?[dfm]?,???
32.     *[cfk][bhj],*S?,*SS:=???,?[dfm]?,???
33.     *[0deg][0deg],*S?,***:=???,?[0bci]?,???
34.     *[0deg][bhj],*S?,***:=???,?[0bci]?,???
35.     *[1478ilm][1478ilm],*S?,***:=???,?[169agjk]?,???
36.     *[1478ilm][59n],*S?,***:=???,?[169aj]?,???
37.     *[1478ilm][cfk],*S?,***:=???,?[1agjk]?,???
38.     *[26pq][26pq],*S?,***:=???,?[24no]?,???
39.     *[26pq][3ao],*S?,***:=???,?[24no]?,???
40.     *[3ao][1478ilm],*S?,***:=???,?[37q]?,???
41.     *[3ao][59n],*S?,***:=???,?[37q]?,???
```

```
42.     *[3ao][cfk],*S?,***:=???,?[37]?,???
43.     *[59n][26pq],*S?,***:=???,?[58p]?,???
44.     *[59n][3ao],*S?,***:=???,?[58p]?,???
45.     *[bhj][1478ilm],*S?,***:=???,?[ehl]?,???
46.     *[bhj][59n],*S?,***:=???,?[hl]?,???
47.     *[bhj][cfk],*S?,***:=???,?[ehl]?,???
48.     *[cfk][0deg],*S?,***:=???,?[dfm]?,???
49.     *[cfk][bhj],*S?,***:=???,?[dfm]?,???
50.     ?[0deg][0deg],[0cdf]S?,?SS:=???,?[0b]?,???
51.     ?[0deg][bhj],[0cdf]S?,?SS:=???,?[0b]?,???
52.     ?[bhj][1478ilm],[0cdf]S?,?SS:=???,?[eh]?,???
53.     ?[bhj][59n],[0cdf]S?,?SS:=???,?[h]?,???
54.     ?[bhj][cfk],[0cdf]S?,?SS:=???,?[eh]?,???
55.     ?[1478ilm][1478ilm],[137ahjl]S?,?SS:=???,?[19k]?,
   ???
56.     ?[1478ilm][59n],[137ahjl]S?,?SS:=???,?[19]?,???
57.     ?[1478ilm][cfk],[137ahjl]S?,?SS:=???,?[1k]?,???
58.     ?[59n][26pq],[137ahjl]S?,?SS:=???,?[58]?,???
59.     ?[59n][3ao],[137ahjl]S?,?SS:=???,?[58]?,???
60.     ?[cfk][0deg],[137ahjl]S?,?SS:=???,?[fm]?,???
61.     ?[cfk][bhj],[137ahjl]S?,?SS:=???,?[fm]?,???
62.     ?[26pq][26pq],[25np]S?,?SS:=???,?[2o]?,???
63.     ?[26pq][3ao],[25np]S?,?SS:=???,?[2o]?,???
64.     ?[3ao][1478ilm],[25np]S?,?SS:=???,?[3q]?,???
65.     ?[3ao][59n],[25np]S?,?SS:=???,?[3q]?,???
66.     ?[3ao][cfk],[25np]S?,?SS:=???,?[3]?,???
67.     ?[26pq][26pq],[48o]S?,?SS:=???,?[4n]?,???
68.     ?[26pq][3ao],[48o]S?,?SS:=???,?[4n]?,???
69.     ?[3ao][1478ilm],[48o]S?,?SS:=???,?[7]?,???
70.     ?[3ao][59n],[48o]S?,?SS:=???,?[7]?,???
71.     ?[3ao][cfk],[48o]S?,?SS:=???,?[7]?,???
72.     ?[1478ilm][1478ilm],[69q]S?,?SS:=???,?[6a]?,???
73.     ?[1478ilm][59n],[69q]S?,?SS:=???,?[6a]?,???
74.     ?[1478ilm][cfk],[69q]S?,?SS:=???,?[a]?,???
75.     ?[59n][26pq],[69q]S?,?SS:=???,?[p]?,???
76.     ?[59n][3ao],[69q]S?,?SS:=???,?[p]?,???
77.     ?[0deg][0deg],[bim]S?,?SS:=???,?[ci]?,???
78.     ?[0deg][bhj],[bim]S?,?SS:=???,?[ci]?,???
79.     ?[bhj][1478ilm],[bim]S?,?SS:=???,?[l]?,???
80.     ?[bhj][59n],[bim]S?,?SS:=???,?[l]?,???
81.     ?[bhj][cfk],[bim]S?,?SS:=???,?[l]?,???
82.     ?[1478ilm][1478ilm],[egk]S?,?SS:=???,?[gj]?,???
83.     ?[1478ilm][59n],[egk]S?,?SS:=???,?[j]?,???
84.     ?[1478ilm][cfk],[egk]S?,?SS:=???,?[gj]?,???
85.     ?[cfk][0deg],[egk]S?,?SS:=???,?[d]?,???
```

```
86.     ?[cfk][bhj],[egk]S?,?SS:=???,?[d]?,???
87.     ?[0deg][0deg],[0cdf]S?,***:=???,?[0b]?,???
88.     ?[0deg][bhj],[0cdf]S?,***:=???,?[0b]?,???
89.     ?[bhj][1478ilm],[0cdf]S?,***:=???,?[eh]?,???
90.     ?[bhj][59n],[0cdf]S?,***:=???,?[h]?,???
91.     ?[bhj][cfk],[0cdf]S?,***:=???,?[eh]?,???
92.     ?[1478ilm][1478ilm],[137ahjl]S?,***:=???,?[19k]?,
   ???
93.     ?[1478ilm][59n],[137ahjl]S?,***:=???,?[19]?,???
94.     ?[1478ilm][cfk],[137ahjl]S?,***:=???,?[1k]?,???
95.     ?[59n][26pq],[137ahjl]S?,***:=???,?[58]?,???
96.     ?[59n][3ao],[137ahjl]S?,***:=???,?[58]?,???
97.     ?[cfk][0deg],[137ahjl]S?,***:=???,?[fm]?,???
98.     ?[cfk][bhj],[137ahjl]S?,***:=???,?[fm]?,???
99.     ?[26pq][26pq],[25np]S?,***:=???,?[2o]?,???
100.    ?[26pq][3ao],[25np]S?,***:=???,?[2o]?,???
101.    ?[3ao][1478ilm],[25np]S?,***:=???,?[3q]?,???
102.    ?[3ao][59n],[25np]S?,***:=???,?[3q]?,???
103.    ?[3ao][cfk],[25np]S?,***:=???,?[3]?,???
104.    ?[26pq][26pq],[48o]S?,***:=???,?[4n]?,???
105.    ?[26pq][3ao],[48o]S?,***:=???,?[4n]?,???
106.    ?[3ao][1478ilm],[48o]S?,***:=???,?[7]?,???
107.    ?[3ao][59n],[48o]S?,***:=???,?[7]?,???
108.    ?[3ao][cfk],[48o]S?,***:=???,?[7]?,???
109.    ?[1478ilm][1478ilm],[69q]S?,***:=???,?[6a]?,???
110.    ?[1478ilm][59n],[69q]S?,***:=???,?[6a]?,???
111.    ?[1478ilm][cfk],[69q]S?,***:=???,?[a]?,???
112.    ?[59n][26pq],[69q]S?,***:=???,?[p]?,???
113.    ?[59n][3ao],[69q]S?,***:=???,?[p]?,???
114.    ?[0deg][0deg],[bim]S?,***:=???,?[ci]?,???
115.    ?[0deg][bhj],[bim]S?,***:=???,?[ci]?,???
116.    ?[bhj][1478ilm],[bim]S?,***:=???,?[l]?,???
117.    ?[bhj][59n],[bim]S?,***:=???,?[l]?,???
118.    ?[bhj][cfk],[bim]S?,***:=???,?[l]?,???
119.    ?[1478ilm][1478ilm],[egk]S?,***:=???,?[gj]?,???
120.    ?[1478ilm][59n],[egk]S?,***:=???,?[j]?,???
121.    ?[1478ilm][cfk],[egk]S?,***:=???,?[gj]?,???
122.    ?[cfk][0deg],[egk]S?,***:=???,?[d]?,???
123.    ?[cfk][bhj],[egk]S?,***:=???,?[d]?,???
124.    ?[0deg]*,[0cdf]S*:=???,?[0b]?
125.    ?[bhj]*,[0cdf]S*:=???,?[eh]?
126.    ?[1478ilm]*,[137ahjl]S*:=???,?[19k]?
127.    ?[59n]*,[137ahjl]S*:=???,?[58]?
128.    ?[cfk]*,[137ahjl]S*:=???,?[fm]?
129.    ?[26pq]*,[25np]S*:=???,?[2o]?
```

```
130.    ?[3ao]*,[25np]S*:=???,?[3q]?
131.    ?[26pq]*,[48o]S*:=???,?[4n]?
132.    ?[3ao]*,[48o]S*:=???,?[7]?
133.    ?[1478ilm]*,[69q]S*:=???,?[6a]?
134.    ?[59n]*,[69q]S*:=???,?[p]?
135.    ?[0deg]*,[bim]S*:=???,?[ci]?
136.    ?[bhj]*,[bim]S*:=???,?[l]?
137.    ?[1478ilm]*,[egk]S*:=???,?[gj]?
138.    ?[cfk]*,[egk]S*:=???,?[d]?
```

Listing 3: Array Grammar for Terrain Generation (surrounded by water)

```
1.      **,*S:=**,*[2o]
2.      ***,[2n]SS:=???,?[2o]?
3.      ***,[4o]SS:=???,?[4n]?
4.      ***,[2n]S*:=???,?[2]?
5.      ***,[4o]S*:=???,?[n]?
6.      *[26pq][26pq],*S?,*SS:=???,?[2o]?,???
7.      *[26pq][3ao],*S?,*SS:=???,?[2o]?,???
8.      *[3ao][1478ilm],*S?,*SS:=???,?[3q]?,???
9.      *[3ao][59n],*S?,*SS:=???,?[3q]?,???
10.     *[3ao][cfk],*S?,*SS:=???,?[3]?,???
11.     *[26pq][26pq],*S?,***:=???,?[2]?,???
12.     *[26pq][3ao],*S?,***:=???,?[2]?,???
13.     *[3ao][1478ilm],*S?,***:=???,?[q]?,???
14.     *[3ao][59n],*S?,***:=???,?[q]?,???
15.     ?[0deg][0deg],[0cdf]S?,?SS:=???,?[0b]?,???
16.     ?[0deg][bhj],[0cdf]S?,?SS:=???,?[0b]?,???
17.     ?[bhj][1478ilm],[0cdf]S?,?SS:=???,?[eh]?,???
18.     ?[bhj][59n],[0cdf]S?,?SS:=???,?[h]?,???
19.     ?[bhj][cfk],[0cdf]S?,?SS:=???,?[eh]?,???
20.     ?[1478ilm][1478ilm],[137ahjl]S?,?SS:=???,?[19k]?,
   ???
21.     ?[1478ilm][59n],[137ahjl]S?,?SS:=???,?[19]?,???
22.     ?[1478ilm][cfk],[137ahjl]S?,?SS:=???,?[1k]?,???
23.     ?[59n][26pq],[137ahjl]S?,?SS:=???,?[58]?,???
24.     ?[59n][3ao],[137ahjl]S?,?SS:=???,?[58]?,???
25.     ?[cfk][0deg],[137ahjl]S?,?SS:=???,?[fm]?,???
26.     ?[cfk][bhj],[137ahjl]S?,?SS:=???,?[fm]?,???
27.     ?[26pq][26pq],[25np]S?,?SS:=???,?[2o]?,???
28.     ?[26pq][3ao],[25np]S?,?SS:=???,?[2o]?,???
29.     ?[3ao][1478ilm],[25np]S?,?SS:=???,?[3q]?,???
30.     ?[3ao][59n],[25np]S?,?SS:=???,?[3q]?,???
31.     ?[3ao][cfk],[25np]S?,?SS:=???,?[3]?,???
```

```
32.     ?[26pq][26pq],[48o]S?,?SS:=???,?[4n]?,???
33.     ?[26pq][3ao],[48o]S?,?SS:=???,?[4n]?,???
34.     ?[3ao][1478ilm],[48o]S?,?SS:=???,?[7]?,???
35.     ?[3ao][59n],[48o]S?,?SS:=???,?[7]?,???
36.     ?[3ao][cfk],[48o]S?,?SS:=???,?[7]?,???
37.     ?[1478ilm][1478ilm],[69q]S?,?SS:=???,?[6a]?,???
38.     ?[1478ilm][59n],[69q]S?,?SS:=???,?[6a]?,???
39.     ?[1478ilm][cfk],[69q]S?,?SS:=???,?[a]?,???
40.     ?[59n][26pq],[69q]S?,?SS:=???,?[p]?,???
41.     ?[59n][3ao],[69q]S?,?SS:=???,?[p]?,???
42.     ?[0deg][0deg],[bim]S?,?SS:=???,?[ci]?,???
43.     ?[0deg][bhj],[bim]S?,?SS:=???,?[ci]?,???
44.     ?[bhj][1478ilm],[bim]S?,?SS:=???,?[l]?,???
45.     ?[bhj][59n],[bim]S?,?SS:=???,?[l]?,???
46.     ?[bhj][cfk],[bim]S?,?SS:=???,?[l]?,???
47.     ?[1478ilm][1478ilm],[egk]S?,?SS:=???,?[gj]?,???
48.     ?[1478ilm][59n],[egk]S?,?SS:=???,?[j]?,???
49.     ?[1478ilm][cfk],[egk]S?,?SS:=???,?[gj]?,???
50.     ?[cfk][0deg],[egk]S?,?SS:=???,?[d]?,???
51.     ?[cfk][bhj],[egk]S?,?SS:=???,?[d]?,???
52.     ?[26pq][26pq],[25np]S?,***:=???,?[2]?,???
53.     ?[26pq][3ao],[25np]S?,***:=???,?[2]?,???
54.     ?[3ao][1478ilm],[25np]S?,***:=???,?[q]?,???
55.     ?[3ao][59n],[25np]S?,***:=???,?[q]?,???
56.     ?[1478ilm][1478ilm],[69q]S?,***:=???,?[6]?,???
57.     ?[1478ilm][59n],[69q]S?,***:=???,?[6]?,???
58.     ?[59n][26pq],[69q]S?,***:=???,?[p]?,???
59.     ?[59n][3ao],[69q]S?,***:=???,?[p]?,???
60.     ?[59n]*,[137ahjl]S*:=???,?[5]?
61.     ?[26pq]*,[25np]S*:=???,?[2]?
62.     ?[26pq]*,[48o]S*:=???,?[n]?
63.     ?[59n]*,[69q]S*:=???,?[p]?
```