12-2018

Agent Based Terrain Generator: Cruthú

Lawrence L. O'Boyle
Grand Valley State University

Follow this and additional works at: https://scholarworks.gvsu.edu/theses

Part of the Computer Sciences Commons

Recommended Citation
https://scholarworks.gvsu.edu/theses/916

This Thesis is brought to you for free and open access by the Graduate Research and Creative Practice at ScholarWorks@GVSU. It has been accepted for inclusion in Masters' Theses by an authorized administrator of ScholarWorks@GVSU. For more information, please contact scholarworks@gvsu.edu.
Agent Based Terrain Generator: Cruthú

Lawrence Longfield O’Boyle

A Thesis Submitted to the Graduate Faculty of
GRAND VALLEY STATE UNIVERSITY

In
Partial Fulfillment of the Requirements
For the Degree of
Master of Science in Computer Information Systems

School of Computing and Information Systems

December 2018
Abstract

Terrain generation models are applied in different industries and fields of study. Current techniques assist in planning transportation networks, visualizing population migrations, conducting epidemiology research, and training self-driving cars and drones. Many current applications and research models generate complex realistic artifacts, e.g., trees, rivers, coastlines, populations, and even cities. Unfortunately, most of these techniques are described and implemented separately. Techniques do not work together to generate a complete holistic view. This thesis proposes a model, Cruthú (Gaelic for "creation"), that provides a novel platform allowing for complete world generation by integrating existing research and algorithms. The model is inspired by the Unix command-line pipeline where independent tools can be linked together to handle a complex workflow. Cruthú proposes a similar approach. This thesis uses the proposed model to create a framework application. To do this, we have investigated several industry standards to allow a generation "module" to present a robust, unified application program interface. Cruthú then uses dynamic loading and a pipelined architecture to generate a complete world. Cruthú is highly modular, allowing components to be swapped in, swapped out, or re-ordered to achieve desired results. A prototype model was built that implements the main features, and further work will allow for individual algorithms to be integrated in more complex interactions. In this prototype, common terrain generation techniques were used to show the ease of implementing algorithms.
# Contents

Thesis Approval Form .................................................. 2  
Contents ....................................................................... 4  
List of Figures ................................................................. 5  

## 1 Introduction .............................................................. 6  

## 2 Background ............................................................. 8  

### 2.1 Model ................................................................. 8  
### 2.2 Implementation ..................................................... 9  

## 3 Design ................................................................... 10  

### 3.1 Design Inspiration .................................................. 10  
### 3.2 Cruthú’s Modules .................................................... 11  
#### 3.2.1 Node ............................................................... 12  
#### 3.2.2 Tera ............................................................... 12  
#### 3.2.3 TeraGen .......................................................... 12  
#### 3.2.4 Forma ............................................................ 13  
#### 3.2.5 Indexer ............................................................ 14  
#### 3.2.6 Operations Sequencing ....................................... 15  
#### 3.2.7 Dynamic Runtime Loading ................................. 15  
#### 3.2.8 Configuration File ............................................ 15  

## 4 Implementation ........................................................ 16  

### 4.1 Dynamic Runtime Loading ........................................ 16  
### 4.2 Node ................................................................ 17  
### 4.3 Tera ................................................................ 17  
### 4.4 TeraGen .............................................................. 19  
### 4.5 Indexer ............................................................... 19  
### 4.6 Forma ................................................................. 19  

## 5 Discussion ............................................................... 21  

## 6 Future Work ............................................................ 22  

## 7 Conclusion ............................................................... 23  

References ................................................................... 25  
ScholarWorks Submission Agreement .............................. 26
List of Figures

1  Cruthú’s Work Flow Diagram ................................................. 11
2  Inner Landscape - Generated with Oceans, Mountains, Rivers, and Lakes ... 13
3  Random Sphere Projection - Height Transformation ............................... 14
4  FormaRiver.cpp ....................................................................... 18
5  Island and Mountain - River Close Up ............................................. 22
6  Island Generation with Cruthú ...................................................... 23
1 Introduction

Terrain generation models are diverse and complex systems applicable in many different industries and fields of study. Current techniques assist in planning transportation networks, visualizing population migrations, conducting epidemiology research, and training self-driving cars and drones. For example, transportation systems simulate highly complex interactions at the level of a city, where delays are felt among the population, to determine optimum routes. Models for population migration can be found in simulation for various scales, cities to countries, to help determine growth and decline of population centers. Epidemiology modeling utilizes simulations to gather population information for disease, modeling disease spreading with high accuracy. Research on training of self-driving cars has shown that using generated realistic models in conjunction with real training data can improve result fidelity [9]. Individually, these various fields of study are affected by terrain generation models, and many of them rely on other work in the field to do so.

Some techniques used for terrain generation, rely on randomization rather than a holistic approach. A holistic approach would require the system to be compromised of the whole, not individual features being determined by fractals. Examples of holism would be the placing of forests in places they are likely to survive in, determined by waters levels, soil content, temperature, etc. Examples of non-holistic approaches include procedurally generated video games like Minecraft, RimWorld, and No Man’s Sky. These games use Perlin’s noise, diamond-square, and other algorithms that are computationally efficient and the resulting structures look relatively real. Not being holistic isn’t a limitation in this context. Whereas trying to model the complete interaction of the environment of a particular area requires a large computation space. This can lead to issues with interaction of different terrain modeling algorithms, as many are very focused into the niche of its area.

Many of the terrain generation algorithms are developed individually with a specific problem to be solved. The algorithm proposed by Kamal and Uddin [6] is a good example of being focused on one feature. They intentionally chose to make a singular mountain or crater at a specific area, with the decision being affected by height differences. Looking at another approach, Belhadj and Audibert [2] use both a way to model ridges and a way to model rivers.
Belhadj and Audibert [2] made a more complicated interaction of two different models to create the terrain they were looking for. Neither of them designed their algorithms to be used together, making ridgelines with peaks scattered throughout.

Trying to combine Kamal and Uddin’s [6] approach with Belhadj and Audibert’s [2] may be difficult. While designing the approaches, they do not consider the others’ requirements or approach. This can lead to undetermined results that negatively affect each other, or make using the two together to increase realism difficult. Furthermore, if a more complete river algorithm were to be used, interaction with the river algorithm used in Belhadj and Audibert [2] may be difficult or impossible.

This thesis proposes a model, Cruthú (Gaelic for "creation"), that provides a novel platform allowing for complete world generation by integrating existing research and algorithms. The model is inspired by the Unix command-line pipeline where independent tools can be linked together to handle a complex workflow. Cruthú proposes a similar approach. This thesis uses the proposed model to create a framework application. We have investigated several industry standards to allow a generation "module" to present a robust, unified application program interface. Cruthú then uses dynamic loading and a pipelined architecture to generate a complete world. Cruthú is highly modular, allowing components to be swapped in, swapped out, or re-ordered to achieve desired results. A prototype model was built implementing the main features, and further work will allow for individual algorithms to be integrated in more complex interactions.


2 Background

2.1 Model

Previous work on terrain generation has focused on a specific implementation with a parameter list to modify the generated terrain. These parameters limit the use of these algorithms and the terrain involved. This paper presents a solution that will allow the users far more control over the data and interactions of algorithms.

Kamal and Uddin [6] proposed an algorithm that creates realistic mountains with height, area of effect, and location of the primary peak. This algorithm creates realistic mountain peaks, but only takes into account height variations of the area specified. From a holistic approach, how does this interact on an area if it is a body of water, a forest is currently occupying some positions of the area, or the area is a valley?

Belhadj and Audibert [2] proposed an algorithm that utilized a ridge walk and river network walk. This was to create a more natural landscape rather than using a fractal of height transformations. The approach used a ridge walk to make height transformations and to set points where a river would start to erode. This was followed by a river erosion technique to place river areas and to modify height in the eroded areas. This approach is more holistic than Kamal and Uddin [6], as Belhadj and Audibert [2] used two features of a mountain range, and modeled them together. While this was a great interaction, to modify the ridge walk or the river walk, the user will need to remove and segment the algorithm. This can be difficult, as the user will need to understand the algorithm, then manipulate it to retain the intent of the original algorithm.

Doran and Parberry [3] proposed a very similar approach of an agent and user defined interaction. Doran and Parberry set a series of properties to meet: novelty, structure, interest, speed, and controllability. Using the properties, Doran and Parberry implemented a software agent approach with many settings for controlling them. They define the software agent as an agent that perceives its environment through sensors and acts on it through effectors [3]. The agents can be configured, but the certain agents always run before others. This act closely couples the interactions of the agents to within the run order.
2.2 Implementation

To maintain a system that can properly handle the complexity of Cruthú, two problems will be addressed. One problem was the module flexibility required for the various features, this was solved using dynamic runtime loading for the modules. Sabanl and Yason [10] showed that implementing and understanding the compilation of C++ for loading would be a difficult task. Beazeley, Ward, and Cooke [1] also examined the intricacies of the shared libraries and a common library to utilize for such functionality. Finally, Hjálmtösson and Gray [5] showed use cases that fit Cruthú’s needs of class inheritance in modules and dynamic runtime loading. The other problem was an understanding of the complexity and API design in these operations. Hjálmtösson and Gray [5] also note the importance of dynamic classes to allow continuous change and the difficulty that arises. Gregory [4] showed a complex interactions required of a game engine, and the need to separate and modulize the engine. Cruthú may not be as complex as a game engine, it can take the ideas presented in this similar approach of an underlying system that is acted upon. Finally, Reddy [8] explains the needs for an API, which allows Cruthú to maintain a solid groundwork for handle complexity.
3 Design

3.1 Design Inspiration

As mentioned above Doran and Paberry[3], open up the control and interaction to the user and designers, allowing for better manipulation of the outcome. Cruthú follows a basic interaction model, and opens up its API to provide a way to import shared objects that others create. The design of Cruthú was inspired by the Linux pipe ‘|’ and Apache HTTP Server’s module system. These approaches allow for a very diverse interaction with systems and a complex workflow. The Linux pipe, allows for all output on STDOUT to be redirected to the STDIN of the next application in the pipeline. For example:

```
MONITOR="$(xrandr | grep '\sconnected' | cut -d' ' -f1)" polybar -r main-top &
```

The bash command above is an example of the usefulness of the Linux pipe function. It allows for xrandr to get information, pass it through to grep, and cut to assign a environment variable MONITOR before the command polybar could be executed. This allows for polybar to have information it would not have when executed with out the bash variable assignment.

While the Linux pipe was important, Apache HTTP Server played an important role for a more diverse framework. The module systems implemented in Apache HTTP Server allow the server to load only those modules that are actually needed. These include encoding algorithms, interpreters like PHP, caching mechanisms, etc.

These two features inspired Cruthú’s design that allows functionality to be encoded as standalone modules. Examples of this would be implementing a parametrically controlled mountain [6] over a basic noise function such as Perlins noise. This allows user specialists, in the respective fields, to contribute and rely on others defined functionality, instead of trying to implement an area of knowledge they are not specialists in. To do this, Cruthú exposes an API and a comprehensive configuration file implementation.
3.2 Cruthú’s Modules

Cruthú’s model has seven key features: Node, Tera, TeraGen, Formas, Indexers, Operation Sequencing, Dynamic Runtime Loading, and a Configuration File. *Node* is the base data type of individual terrain point data. *Tera* is a container for *Node* that maintains a list of all *Nodes*. *TeraGen* initializes *Tera*, setting the shape and size, and exports the data. *Formas* are what act upon *Nodes* to modify the location data. *Indexers* query the *Tera* object and set a list of *Nodes* that match the query. Operation sequencing determines in which order Indexers and Formas interact. Dynamic runtime loading allows the loading of *Tera*, *TeraGen*, *Indexers*, and *Formas* that a user specifies. The configuration file allows users to control parameters of Cruthú and its modules. These all interact into a pipeline as shown in Figure 1.
3.2.1 Node

Cruthú provides a data type Node to represent and contain the terrain information for one particular point. Node is the base data type of Cruthú. Node maintains a list of its connected Nodes. Node is a data construct that only has knowledge of the connections to other nodes. Node is not aware of the topology of the world. Topology is created by TeraGen (described below). This allows a Node to know what Nodes it is connected to. The Node data type maintains a minimal list of operations as this is the main part of operation timings.

3.2.2 Tera

Tera maintains the list of Nodes in the model, as well as special information for indexed Nodes. The data container requires the ability to be accessed with ease and allows for multiple objects to access it at once. To make access easier, the model will pass references of individual Nodes to any callers. This reduces its footprint and allows it to be operated from multiple objects. Tera will be operated on from TeraGen, Indexers, and Formas. These operations can be at the same time.

3.2.3 TeraGen

TeraGen has two purposes, generation of the base Tera object and exporting the Tera object. The generation of the base Tera object can be done by importing a saved Tera object from a previous run of Cruthú or by creating one with Nodes already made and connected. TeraGen has these two operations, as this functionality will have knowledge the generated Tera object shape and size the best. The object shape that TeraGen creates within Tera with Node can be anything from a two-dimensional grid, Figure 2, to a four-dimensional object, Figure 3. Allowing the export function to be managed by TeraGen lets Cruthú’s users define its outputs wherever is needed, not forcing data into one format. This export control will allow Cruthú to interact with other applications through its data.
3.2.4 Forma

Forma actors modify the Tera data. Forma actors have two modes of operation on Tera, step and modify. The step mode performs a single operation on the Tera object. The modify mode will operate on Tera until the Forma has decided it is done. These two modes exists for finer control of the modification of the data and for optimization when a Forma must run by itself. Step mode gives per-operation control allowing many other Formas to operate in similar step, forcing all Formas to operate at the time. While modify allows the individual Formas to operate out of sync of other Formas. Muliple Forma can operate at one time, but they can
declare a requirement or blacklist of other *Forma*, *Indexers*, and *TeraGen*. The declaration of a requirement allow the *Forma* to specify modules that it needs for modification, a module reacquiring certain aspects of another module before it can run. The declaration of a blacklist allow the *Forma* to specify modules that it is incompatible with from running.

### 3.2.5 Indexer

The *Indexer* lets the users make a query against the *Node* data in *Tera*. The query will create an indexed list of *Nodes* on *Tera*. This list of indexed nodes can be used for improved control of which terrain the *Forma* may call upon. The *Indexer* is limited to one per *Operation* in the *Operation Sequence*. The *Indexer* can require certain *Forma* or *Tera* dependencies are met
before it queries the the Tera object. Indexer can also require certain Forma or Tera blacklists restrictions are met before it queries the the Tera object.

3.2.6 Operations Sequencing

Operation sequencing determines the sequence of operations that will be executed and how they will be executed on the Tera object. An operation maintains an Indexer, Formas, and the configurations for them. An operation will execute an Indexer, then call the list of Forma. Operation sequencing will create the list of operations that will be passed to the execution process. This process allows for a complex interaction of Formas, TeraGen, and Indexers.

3.2.7 Dynamic Runtime Loading

Dynamic runtime loading places library or binary into memory. Cruthú uses this method to allow users to implement their own version of Forma, Tera, TeraGen, and Indexers. The goal of Cruthú is to create a framework that will give users low-level control of the terrain generation while abstracting complexities away. Cruthú publishes the API for the relevant objects for user to extend.

3.2.8 Configuration File

Since the interactions of Cruthú’s features can be complex, and are likely require the user to tweak settings that will affect the execution of the model, a comprehensive configuration file is provided. Settings to determine individual dependencies, blacklist, random seeds, CPU threads, and shared objects are all located in this file.
4 Implementation

For the implementation of Cruthú, several factors were considered: computation time, installation ease, cross-platform, and memory usage. Computation time requires that operations on basic data be quick and efficient.

Installation ease meant that the users should be easily able to configure and install this on any system. Cross-platform requirements directed library usage, language usage, and installation means. Memory usage was vital, when maintaining a list of millions or billions of data points the smallest improvement can save gigabytes of memory.

The decisions to match these requirements of the implementation lead to the usage of C++ and CMake build system. C++ is cross-platform, and library were chosen to make this transition easier. For the most part, the implementation minimized the usage of libraries outside of the C++ 17 standard. CMake is also fully supported on all most operating systems, making project specifications and requirements easier for the end user to meet.

Before going over the individual implementations of modules Tera, TeraGen, Indexer, and Forma, some features are shared among them. Each of these modules have logging functionality using the library spdlog, which is a light weight and efficient logging system. Logging was added for debugging and general information logging. These modules also share a seed, to allow repeatable generation for the modules to use if using any type of pseudo generators.

4.1 Dynamic Runtime Loading

Prior to dynamic runtime loading being implemented, the user would have to compile their module with the source code of the application of Cruthú. This caused compilation complexity, and the need to recompile the entire application with any other module being loaded every time a module was added, removed, updated, or when the application was updated. Dynamic runtime loading allows the flexibility of loading multiple modules at once. This also allows the user to load their module, without the need of worrying about the intricacies of the framework implementation. The implementation has two parts to it, the actual dynamic loader and the
special instance functions allocator and deleter. The dynamic loader has three functions: open library, close library, and get instance. Open library takes the file path to the shared object and opens it to be used. Close library closes the shared object and decrements the shared object counter on the library. Get instance calls the allocator and deleter functions on the modules, returning an instance of the class as a shared pointer.

4.2 Node

In the model, Tera maintains a list of data points, this was reflected in the creation of a Node data type. Each Node maintains a list of user-defined data, e.g. water percentage, height, terrain type, etc. The Node also has a list of neighbors that it is connected to for locality movement, this is a bi-directional relationship creating a link list like connection. This data type also maintains a universally unique identifier (UUID), from the Boost library, for identification purposes. The data modification function are created to make them asynchronous ready using Resource Acquisition Is Initialization (RAII) methodology. Node is the base data used in many operations, the functionality was purposely minimized to reduce complexity and computation issues. The example code, from Figure 4, shows several examples of common calls or uses of Node. This code example shows terrain being modified and neighboring Nodes being used for a walk.

4.3 Tera

The Tera data container has many requirements. These requirements include efficient manipulation of Nodes and parallelization safety. Tera contains the master list of Nodes and current list of indexed Nodes. The master list of Nodes is used for three things: quick lookups if the node is known, keeping the nodes from the shared pointer destruction, and the search list that Indexer utilizes. This model allows for Tera to be passed around freely for operations on itself. On line 25 in Figure 4 shows the basic use of Tera, grabbing an indexed node to start its walk.
```cpp
#include <cruthu/formarivers/FrameRivers.hpp>
#include <cruthu/ITera.hpp>
#include <cruthu/Node.hpp>
#include <random>
#include <memory>
#include <unordered_map>
#include <vector>
#include <string>
#include <utility>

cruthu::FrameRivers::FrameRivers() {
    this->mSeed = std::random_device{}();
    this->mStepsTaken = 0;
}

void cruthu::FrameRivers::Step(std::shared_ptr<cruthu::ITera> tera) {
    if(tera->IndexedNodes.size() < 1) {
        this->mLogger->warn("No mountains indexed");
        return;
    }
    if(this->mNode.get() == nullptr) {
        // If we are in here, that means this would be the 'first' step taken
        this->mNode = tera->GetIndexedNode();
        this->mVisited.insert(this->mNode->toString());
    }
    if(this->mNode->GetTerrain() == cruthu::Terrain::Type::WATER) {
        return;
    }
    this->mNode->SetTerrain(cruthu::Terrain::Type::RIVER);
    auto neighbors = this->mNode->GetNeighbors(); at(0);
    for(auto neighbor : this->mNode->GetNeighbors()) {
        auto it = this->mVisited.find(neighbor->toString());
        if(it == this->mVisited.end()) {
            continue;
        }
        if(neighbor->GetHeight() < lPoint->GetHeight()) {
            lPoint = neighbor;
        }
    }
    this->mNode = lPoint;
    this->mVisited.insert(this->mNode->toString());
    this->mStepsTaken++;
}

void cruthu::FrameRivers::Modify(std::shared_ptr<cruthu::ITera> tera) {
    if(tera->IndexedNodes.size() < 1) {
        this->mLogger->warn("No mountains indexed");
        return;
    }
    for(auto step = 0; step < 1000; ++step) {
        this->Step(tera);
    }
}

void cruthu::FrameRivers::SetSink(std::shared_ptr<spdlog::sinks::sink> sink, spdlog::level::level_enum level) {
    this->mLogger = std::make_shared<spdlog::logger("FrameRivers", sink));
    this->mLogger->set_level(level);
    this->mLogger->trace("Logger Initialized");
}
```

Figure 4: FormaRiver.cpp
4.4 TeraGen

TeraGen handles three functions, import initialization, empty initialization, and exporting. Importing or initialization is the first step in interacting with Tera. TeraGen creates the linked connection of the Nodes in Tera. It does this by importing the data (import function) or initializing an empty set (initialization function) for the later interactions to work on. The last function TeraGen handles is the export function. This decision stems from the knowledge that TeraGen has of the underlying data connections.

In the prototype application, TeraGen was used in basic ways to show functionality. The initialization functionality creates a basic two-dimensional grid map that has the same height and width. This export function uses ImageMagick++ to create a basic PNG output of the map.

4.5 Indexer

The first implementation indicated a need have an efficient means of querying Tera. These queries are used to create an indexed list based on the criteria specified. The Indexer module dictates what criteria is matched and how it is searched for. The indexer has a single function that takes a Tera object to used for its search and to set result list in Tera.

Currently, the implementations of Indexer in the application has two different examples illustrating how different Indexers can be built. The first example gets Nodes in Tera at random with a pseudo random generator, indexing a percentage of the Nodes in Tera. The second implementation queries Tera for any terrain type mountain. It does this by searching all nodes for a terrain type MOUNTAIN.

4.6 Forma

Modification of the Tera object is done by Formas. These Forma are the only way to modify the Nodes in Tera. This design creates a simple way for multiple Forma to run at once.

Three sample Forma were created to illustrate common terrain generation functionality. The first one uses Perlin noise [7] to set basic height variations and bioming of water, grassland, forest, and mountains. This module is called FormaPerlin, and shows the ease of creating
*Forma* with functionality specified outside of the the interface. *FormaPerlin* does not utilize the indexing features, as it needs to cover the entire *Tera* object.

The second module is called *FormaMountains*, this implementation requires an *Indexer* that indexes mountains. *FormaMountain* first finds groupings of mountains from the indexed list. After the groupings are found, *FormaMountains* uses a pseudo random generator to give a modifier effect on the height set from *FormaPerlin*. Then a diamond-square algorithm is used on the group, to creating realistic heights from the random noise modifiers.

The third module is *FormaRiver*, which generates a basic river path from a mountain to a water location. *FormaRiver* is used as an example as seen in Figure 4. This path may create lakes, end a river in the middle of land, or end at a water location. The module requires mountains to be indexed, *IndexerMountain*, and starts off in a mountain *Node*. Afterwards, it takes the lowest neighboring *Node* and continues walking this path until it either hits a water *Node* or it has stopped being called from the step function.
5 Discussion

A prototype Cruthú framework was implemented from the model. Using the framework, we built sample Formas to actually build a world. This application created a two-dimensional grid with TeraGen to be stored within a Tera object. The operation sequence was determined: Random Index → Perlin Noise Forma → Mountain Indexer → Diamond Square Forma → River Walk Forma. The random Indexer randomly chooses one-thousandth of the node size to initialize the index list on Tera. Perlin noise was applied, setting height distributions and terrain types to the entire Tera object. Once the initial heights and terrain types were set, the mountain Indexer set the Tera index list to hold all Nodes that were a mountain type. The diamond square Forma used the indexed list in Tera to determine groupings of mountains. With the groupings found, a diamond square algorithm with a random generator is applied to add some height variation as seen in Figure 5. Then the river walk Forma walked starting from a mountain node and traveled to the lowest height of its neighbors. Each of the rivers is created from a different river walk Forma. This created a fully generated terrain shown in Figure 6.

During the implementation, there was a particular interaction between FormaPerlin, FormaMountains, and FormaRivers. After FormaPerlin and FormaMountains went through Tera and created height distributions, FormaRiver walked over the terrain and made very straight lines when not in mountains. Figures 2, 5, and 6 show the same straight lines. This was due to Perlin noise having regular gradients in its algorithm. Where FormaMountains manipulates and distorts the heights creating irregular patterns in the mountain terrain creating more interesting movement, the rest of the terrain has not had these operations done. The regular height gradients in forest and grasslands created rivers that were very straight and uninteresting. While FormaRiver could implement a varying movement, this was not the focus of this river walk, it was supposed to follow lowest connected neighbors.

Despite the interesting artifacts generated by our simple implementation, the sample world provides evidence that Cruthú’s model can be used to dynamically generate artificial worlds.
6 Future Work

Cruthú is currently missing several features, due to time, that would be useful. These features include GPU utilization to speed up certain calculations, multi-node support for computation speed-ups, memory caching for optimizing usage of the machine memory, and a formal language in the configuration file for easier determination of operation sequences.
7 Conclusion

The prototype implementation of Cruthú showed promising results. The pipelined approach to building a world works. Cruthú lets researchers experiment with algorithms without having to deal with the underlying data structure (Node, Tera, and TeraGen) or indexing. It further allows users to combine existing and emerging terrain generation algorithms to produce complete worlds. While implementing these generation algorithms, researchers can verify that they interact well with other algorithms. The ability for a user to add customized data to the underlying data structure (Node) allows holistic worlds to be created that capture much
more than just heights. During the runs, the ease of editing the configuration files allows for easy manipulation of what modules are loaded and the configuration of the individual model. Overall, the model and implementation of Cruthú shows good results.
References


