CS 371 Project 1:

Meshes

Figure 1: *An island from the in-development game The Witness (http://the-witness.net/news/?p=459) implemented as a heightfield. The Witness is by Jon Blow, author of the award-winning indie title Braid. In this project you'll learn how to work with 3D data files and the indexed triangle mesh structure. By the end you'll be able to model a scene like this.*

1 Introduction

1.1 Overview

Tip: Read the report question about irregular heightfields in Section 4.1 before you start programming. It requires some thought and shouldn't be left until just before the deadline.

The interior of an opaque object is never visible. Therefore we only need to model the **surface** in order to render it. The **indexed triangle mesh** is one of the most popular data structures for modeling 3D surfaces.

A 3D **heightfield**[1] is a mathematical function $y(x, z)$ from $\mathbf{R}^2$ to $\mathbf{R}$. In other words, one dimension is a function of the others. Heightfields frequently employed in graphics as models of terrain, non-breaking waves, and small bumps on surfaces. They can't model features like bridges or tunnels in terrain, however one can augment a heightfield with additional models in the context of a scene to represent those features. That is how most video games model their outdoor settings, as seen for example in the development screenshot of *The Witness* in Figure 1. A heightfield can be represented as an indexed triangle mesh by displacing the vertices of a regular grid in a single dimension.

Data-driven programming is a technique for pushing constants out of source

[1] A bit confusingly, this is quite reasonably called a *2D* heightfield as well; and it is also occasionally referred to as a 2.5D model!

code and into data files that are read at runtime. It has several advantages over so-called **hard-coded** constants that are embedded in code. Data files are interchange formats that allow the use of different tools for creating and processing data. They allow the same input to be used with many programs, and are a way of connecting programs to each other. By working with standard data file formats, we can increase the number of programs that our own can interact with and gain access to large repositories of information. You've already written your first data-driven program: most of the work in the Cubes lab last week was creating the data file, not writing the program for the 3D rendering. Some of you created the data file using *another* program...this is called **procedural modeling**, and is also an important technique. You'll note that the `preprocess` part of a scene file is also a little program inside the data. Nesting code in data in code in data, etc. is a key technique for advanced programming.

This week we'll all extend the Cubes program to use a mixture of data-driven and procedural modeling. We'll continue to use the G3D `.scn.any` file format for our scenes. This week we won't load the stock `cube.obj` file, however. Instead, we'll create our own 3D indexed triangle meshes and save them in the Geomview OFF file format [Geometry Techonologies 2010]. The OFF file format is relatively obscure. I chose it because it is similar to more widely used formats but is much easier to create and debug, especially since they are in plain ASCII text. In later projects you will work with some mainstream extensible binary formats that follow the same principles. As for most projects, you will use your solution code as the starting point for the next project, so take care to structure the program in a flexible manner and be sure to document your source clearly.

1.2 Schedule

Out:	Tuesday,	September 11	
Checkpoint:	Thursday,	September 13,	1:00 pm
Due:	Monday,	September 17,	12:00 pm

Tip: Project specifications are available on Tuesday afternoons–if you read them before class on Wednesday, then you can ask questions in lecture.

This is a moderately challenging, team programming project. You will implement the checkpoint independently and submit it **on paper at the beginning of lab**. You will then collaborate with assigned partners and submit a single solution to the full specification. I recommend completing the checkpoint a bit before the deadline so that you have time to organize and set up your code repository as a team before lab. That way we can use our time in lab more effectively.

Remember to track how much time you spend on this project outside class. You're required to include this in your final report.

As a reference, my solution required 230 statements and 300 comment lines including the reports (as reported by icompile), plus several data files. If your codebase looks like it is going to be more than 1.5× larger or smaller, come talk to me because you may be approaching the problem in a suboptimal way.

2 Rules/Honor Code

Collaborate freely, but don't show you solutions to anyone outside of your group.

You are encouraged to talk to other students and share strategies and programming techniques. You should not look at any other group's code for this project. You may look at and even use anyone's code from *last week's* project, with their permission.

This is the first team programming project of the semester. You and your partners share a single Subversion repository. For this project, you should work together on the major pieces of the project. For example, do not have one person implement the cylinder while the partner implements the heightfield. That is because I designed the tasks for this lab so that doing them in order guides you to the right solutions. You should, however, divide the work of writing helper procedures, debugging, or modeling different parts of the final scene. It is up to you to ensure that each of you has an approximately equal workload and gains experience at equivalent tasks. Remember that your grade depends partly on the correctness and clarity of code written by your partners. I will choose a single C++ method to evaluate as representative of your entire project's code quality. So, you should review and critique your partners' code carefully.

I encourage you to share strategies and information about the file formats with people outside the group. You can share any information related to this except for the specific data files and code that you are required to submit. For example, you may share a sample OFF file for a pentagon, but not for a cube. You should also help one another with C++ and Doxygen syntax and with tips for debugging.

During this project, you are not permitted to directly invoke the following classes and methods or look at their source code: `G3D::Welder`, `G3D::MeshBuilder`, `G3D::MeshAlg`, and `G3D::ArticulatedModel::loadHeightfield`. You are not permitted to leverage ArticulatedModel's ability to directly load an image as a heightfield when specified in a scene file. All source code in G3D, the textbooks, or the Internet that is not prohibited is legal to use, but you should cite Internet resources that you use to avoid plagiarism.

3 Individual Checkpoint

1. Manually create a data file `data-files/cube.off` that contains a 1 m^3 axis-aligned cube **centered at the origin**. Use only an indexed triangle mesh (i.e., no quadrilaterals). You'll need to create a corresponding scene if you want to test it using your program from last week. Print out the data file to hand in.

2. Write pseudocode for a program that generates a cylinder in OFF format. The radius, height, and number of edges on the side should all be variables. Don't worry about the exact interface for writing to files now. Print this out or hand write it.

This checkpoint is worth 50% of your grade on the project.

4 Team Specification

Work in the following groups:

- alpha: Daniel F, Cody, Greg, Qiao
- beta: Scott P-S, Lily, Lucky, Daniel E
- delta: Nico, Tucker, James W, April
- gamma: Daniel S, Dylan, Jonathan, James R
- zeta: Scott, Donny, Michael
- omega: Owen, Josh, Alex

to implement the following by extending the Cubes project from last week:

1. Maintain a blog-like journal in Doxygen that always describes the latest state of your project. This is the story of your project; the report is just the polished conclusion.

   (a) Maintain your journal text in `journal.dox` in the project root directory. I recommend just copying the `journal.dox` and `mainpage.dox` files from the G3D starter project when you begin this week.

   (b) Store all data files (which will be almost exclusively images, I expect) referenced by the journal in the `journal` subdirectory. Note that the journal will automatically link to any classes or methods that you describe.

   (c) The newest entry should always be at the top and the oldest at the bottom.

   (d) Describe bugs that you encounter and how you are diagnosing / did diagnose and fix them.

Tip: G3D is installed at /usr/mac-cs-local/share/cs371/G3D/ and is also available at `http://g3d.sf.net`. It includes sample code...and the samples/starter project has many of these features in it already.

   (e) Make liberal use of images inserted with the \thumbnail command. You will probably find that when it is time to finish your report you can just select some images that you've already made. Remember to put short captions on the images or descriptions below them.

2. Add a checkbox to the GUI for toggling display of the wireframe.

3. Add a reload button to the GUI for reloading the scene. The GUI includes the "reload" button and scene-select drop-down list.

4. Implement the cylinder-generating function in C++. Use one of `G3D::TextOutput`, `fprintf`, or `std::cout` to write to the file; I prefer the first option.

5. Write a procedure that generates a regular heightfield from a grayscale image and saves it to an OFF file. Use Photoshop to draw the height field and `G3D::Image` to load it in your program.

   (a) Treat white as high and black as low in the input image.

   (b) Parameterize the procedure on the input image filename, the vertical axis scale, the scale of the horizontal axes, and the scale of the texture coordinates.

   (c) Put the vertex corresponding to the (0,0) pixel at (0,0,0) in world space.

   (d) Have the image x-axis increase along the world-space $+x$-axis and the image y-axis increase along the world space $+z$-axis.

   (e) Assign texture coordinates to the vertices. Make (0,0) correspond to the upper-left of the original image.

Tip: To debug heightfield texture mapping, create a scene that assigns the original height image as the material.

6. Create a visually-compelling scene containing a texture-mapped heightfield and any other elements that you would like. Commit the scene file and anything not in `mac-cs-local` that it needs to run to the `data-files` directory. Commit the source heightfield image that you used as well. Your program should not generate the height field every time that it is run.

7. Create the documentation reports specified in Sections 3 and 4.1.

As always, you should remove unused code and data from your program. This includes last week's cube scenes and images. An exception is debugging and unit testing code that you may need again in the future, which is acceptable to retain.

Note that the group part of making the actual program is only worth 50% of your grade on the project. Don't worry if your program isn't perfect. Also, while it is motivating to produce the final scene, balance the satisfaction that you receive from that against your other responsibilities and the fact that it contributes a very small amount to your total grade. It is great when students "hit it out of the park" on the creative portion of the assignment, but I don't expect or require you to do that.

4.1 Report

1. Show pictures of:

   (a) A manually created cube

   (b) A generated cylinder

   (c) A top-view of a flat heightfield, showing the tessellation in wireframe

   (d) A visually-compelling scene using your heightfield, and any other objects of your choice. You may turn off the wireframe outlines for this shot to improve the appearance, and should choose appropriate lighting and skybox values.

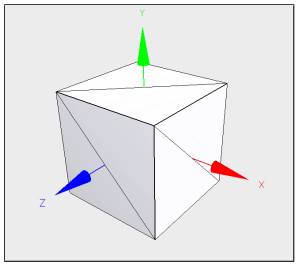

Figure 2: *A "hand crafted" cube.*

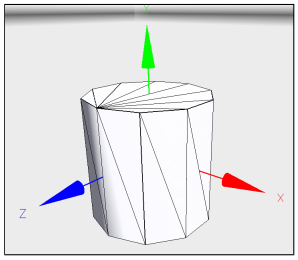

Figure 3: *A tessellated cylinder.*

2. **Questions.**

   (a) How can you simulate a planar reflection, such as the island's reflection in the ocean from Figure 1, using only non-reflective models?

   (b) Why are triangle meshes a popular modeling primitive compared to, say, quadrilaterals, which would be more space efficient?

   (c) The regular tessellation we used for heightfields is inefficient. It will allocate the same number of triangles for flat portions of the heightfield that need few triangles as for very hilly ones that need many to accurately represent the shape. **Briefly** sketch an alternative algorithm for computing a more efficient triangular tessellation for a given heightfield. Do not actually implement your algorithm. You may invent one on your own or use external resources to discover existing algorithms. If you describe an existing algorithm you must cite a reliable source[2]. In either case, explain the algorithm enough that someone *could* figure out the details and implement it from your description.

3. **Feedback.** Your feedback is important to me for tuning the upcoming projects and lectures. Please report:

   (a) How many hours did each of you spend on this project on **required** elements , i.e., the minimum needed to satisfy the specification? Include scheduled lab and time spent on the checkpoint. Give either one number per student or a team average.

   (b) How many additional hours did each of you spent outside of class on this project on **optional** elements, such as polishing your custom scene or extreme formatting of the report? Again, give either one number per student or a team average.

   (c) I intended this to be a moderately challenging project. Rate the difficulty of this project for this point in a 300-level course as: too easy, easy, moderate, challenging, or too hard. What made it so?

[2] Textbooks and scientific publications are considered reliable sources for computer science knowledge. Encyclopedias and blogs are great for discovering ideas...for which you should then track down a reliable source.

(d) I listed my learning objectives for this project on the web page. Briefly describe what you actually learned in one paragraph or a bulleted list; please highlight anything that doesn't match my list.

(e) Rate the educational value of this project relative to the time invested on required elements from 1 (low) to 5 (high).

5 Implementation Advice

Everything in this section is optional–you don't have to follow my advice, or even read it.

5.1 Teamwork

Three ways to program with partners are:

1. Pure pair programming: one person drives a computer while the others watch carefully and makes suggestions or dictate code.
2. Side-by-side programming: each person sits at a separate computer at the same time. Frequent commits and updates and constantly running communication allow them to synchronize their efforts.
3. Asynchronous programming: the partners divide tasks and work at different times, relying on revision control to merge their work. In this model, one typically ensures that the project always compiles and runs before committing to avoid "breaking the build" and slowing down the partner.

You'll need to experiment with these throughout the semester to find which works best for you. It will probably vary depending on who your partners are and what the project is.

Review the Subversion best practices in the Tools handout from the first lab. In particular, always:

1. update and test the build before you start working,
2. ensure that the build is functioning correctly before you commit, and
3. send your partners e-mail whenever you commit code.

The first practice helps you distinguish whether you or your partners broke something–if it was your partners, you know to look at the revision control history or code that they wrote rather than the code you added. The second practice helps avoid breaking the build on your partners. You may need to temporarily disable code that is not compiling if you are out of time but can't fix a problem. The third ensures that your partners know when new updates are available, and what will happen if they choose to update.

When pair programming, it is important that all partners share responsibility for the actions on the computer even though only one is actively typing. The non-typing partners should be thinking one step ahead so that you never lose momentum. Those partners should also be checking what is typed for bugs as it goes in,

Tip: Put effort into creating an effective working relationship. Beyond the current project, your reputation with your peers and your class participation grade are influenced by how you interact with your partner.

and can use other computers to reference documentation just before it is needed. It is a good idea to take turns typing, swapping after crossing milestones (e.g., whenever you check in to Subversion). I find that it takes *more* effort to be the non-typing partner, because more of the thinking onus is on you. You should use the same practices while debugging.

Working with partners, especially new partners, slows down your programming rate. That's because you spend more time thinking and communicating. However, when you get the workflow under control your net development time should decrease. That's because the extra time invested in programming saves significant debugging time, since you tend to generate more correct programs. As soon as you learn your partner assignments, schedule a few pair programming sessions in which you will plan to complete the lab. I recommend pre-scheduling five hours of time outside of class. You might not need that much time. But it is easier to cancel a meeting than it is to schedule one at the last minute.

Now that you're moving code between projects and between partners, correct and useful documentation is very important. Remember; you're not programming for the computer, and you're not programming for me–you're programming for your future self and partners. They'll be a lot more critical than I am, although perhaps not so vocal.

The weakest form of documentation is a comment. Strong interfaces, clear method names and helper methods, clear variable names, useful auxiliary classes, and consistent use of design patterns are much better forms of documentation. **I view comments as the thing you add to poorly written code to help understand it**. If the code is good enough, then it doesn't need comments because it is obvious what the interfaces are and how the implementations work.

Coding conventions become important when bringing together code from many sources. These conventions include the use of capitalization and whitespace. It doesn't matter what conventions you use so long as they are consistent across the codebase. Since we'll be switching partners frequently, it helps if everyone follows the same coding conventions. So I recommend that you follow the G3D coding conventions, which are largely the official Java conventions adapted to C++ syntax. You have the G3D source code available to you, which abundantly demonstrates the coding conventions. If your code is self-inconsistent, you will lose clarity points on the evaluation. If it is consistent but different from the G3D conventions then you will not lose points.

5.2 Getting Started

Start by exporting the previous week's code to new Subversion project. See the Tools handout from last week or refer to the Subversion manual [Collins-Sussman et al. 2008] for information about how to do this. Remember that you can use *anybody's* code from the previous week with their permission, so if you aren't happy with your own project as a starting point, just ask around.

The Subversion command to check out your project this week is:

```
svn co svn://graphics-svn.cs.williams.edu/371/1-Meshes/meshes-<group>
```

Where you should replace `<group>` with the name of your group, which is at the

end of this handout. Note that you will use your own username and password with Subversion, but this week that username does not appear in the project name.

Most of G3D's GUI routines have you pass a raw pointer (yes, the dangerous thing I told you to be careful with last week!) to the GUI control so that it can synchronize with a value in your program. For a checkbox, you naturally want a pointer to a `bool` variable. `G3D::GApp` automatically creates a GUI window called `debugWindow` for you, and a pointer to a `G3D::GuiPane` called `debugPane` as another `G3D::GApp` member variable. Look at the methods on `G3D::GuiPane` in the documentation to see how to create a checkbox.

5.3 Writing Files

There are many ways of writing text files in C++ using G3D. These include the << operator, `G3D::Any::saveG3D::Any`, `G3D::TextOutput`, and `fprintf`. In general, `fprintf` is the worst way of doing this because it provides few features and is easy to make mistakes with. However, you're writing really simple files this week and you already know how its cousin, `printf`, works, so there are certain advantages to `fprintf` right now. Here's an example of writing to a file:

```
// "wt" means "write text"
// fopen is a C function, so you have to pass C-strings to it
FILE* f = fopen("myfile.txt", "wt");

// if f is NULL, we don't have permission to open
// the file or the path is bad
debugAssert(f != NULL);

std::string s = "Hi there!";

fprintf(f, "A constant string\n");
fprintf(f, "Some numbers: %d %d\n", 1, 2);
fprintf(f, "A variable string: %s\n\n", s.c_str());

// Close the file
fclose(f);
f = NULL;
```

I prefer to use the `G3D::TextOutput` class because it provides more sophisticated format control later in the course.

5.4 File Formats

The Geomview OFF file format is documented in the Geomview Manual [Geometry Techonologies 2010] under "input formats". The file format specification is ambiguous and a little bit confusing. That's typical–learning to work with such specifications is part of the point of this project. Your cube and cylinder files do not need texture coordinates, normals, or homogenous vertices. None of your files need vertex colors or face colors.

5.5 Printing `std::string`

It is handy to use `printf` and `G3D::debugPrintf` for debugging. Unfortunately, `G3D::debugPrintf` is a C function, and `std::string` is a C++ class, so they aren't directly compatible. To convert a `std::string` to a C string, use `std::string::c_str()`, e.g.,

```
std::string h = "Hello";
debugPrintf("%s\n", h.c_str());
```

5.6 Making 3D Models

Use large comments with "ASCII art" diagrams to help explain your indexing schemes in source code and in manually-created data files. Remember that you can also embed images in the generated documentation and report.

Program in small, modular pieces. That leads to reuseable code that is easier to debug and maintain. When generating a data file, I find that this advice is best applied by separating the process of generating the data structure in memory from the process of serializing that data structure to disk. Although this week's data structure and file format are similar, this approach is especially helpful when the data structure that you need to build the model is different from the one that you use to encode the completed model.

5.6.1 Cube

When working on the cube, add one face at a time to simplify debugging. Make lots of diagrams in the data file and in your notebook (or draw on your cube)–this stuff is hard to visualize entirely in your head.

5.6.2 Cylinder

When working on the cylinder, generate the top disk first and debug that procedure. Then generate the bottom disk. Finally, fill in the sides. Note that a cylinder with four slices is a cube, so the process of writing this procedure is one of generalizing from data you encoded by hand. There are several ways to tessellate a cylinder, so yours might not look exactly like the one shown in Figure 7.

5.6.3 Heightfield

Photoshop's Render Clouds and Difference Clouds filters generate terrain-like heightfields. You can also find heightfields online. You don't have to make a terrain–heightfields can represent car bodies, buildings, a person sleeping under a sheet, and lots of other shapes with some creativity. Remember that in your scene you can rotate the heightfield, so the "up" direction can change.

Tip: Things get really confusing in the two `for` loops if you name your `Image` variable "height." Consider "elevation."

When writing your heightfield, watch out for integer division:

```
int a = 10;
int b = 7;
float c = a / b;         // c == 0.0f
float d = float(a) / b; // d == 0.7f
```

You're going to face a number of places where you need to decide between using `width` and `(width - 1)` because a grid has one fewer cell than line in each

dimension. Try drawing out a 2×2 heightfield grid and working out the indexing and texture coordinates by hand on that to get these right.

5.7 Disabling Code

There are three ways of disabling debugging code in your program: block comments, run-time branches, and compile-time branches. Block comments are convenient for quickly disabling code, e.g.,

```
/*
debugPrintf("%d %d %d\n", v.x, v.y, v.z);
*/
```

There's no way of easily enabling the code in block comments distributed throughout a program and they don't nest properly, so they are inferior to other methods and your final submission should not include them.

Run-time branches allow you to toggle debugging code at run time, e.g.,

```
if (m_debugVertexPositions) {
    debugPrintf("%d %d %d\n", v.x, v.y, v.z);
}
```

You can easily map the conditionals of run-time branches to GUI controls. Run-time branches ensure that the code is compiled, even if it is not ever run. That means that the disabled code is less likely to become out of date over time. They do incur a small overhead, so sometimes they are not appropriate for an inner loop. However, since our goal is not a shipping product but a reliable code base for experiments, performance considerations should be secondary to program maintenance and ease of use at this point.

Preprocessor commands for compile-time branches look like this:

```
#if DEBUG_VERTEX_POSITIONS
    debugPrintf("%d %d %d\n", v.x, v.y, v.z);
#endif
```

They require recompilation of your program to toggle, however they ensure that there is no run-time cost for disabled debugging code. They also allow you to disable code in syntactically incomplete ways, which is occasionally very useful, if confusing.

5.8 Texture Mapping

Instead of assigning a single "color" to a surface, we can model color variations by stretching an image across a surface. This process is called **texture mapping** and in this context the image is a **texture**. In English, "texture" usually means the feel and bumpiness of a surface, which is called a "normal map" or "bump map" in computer graphics.

We need some way of specifying the mapping between points in the image and locations on the surface of the object. A common way to do this is to tie specific locations in the image to vertices. The in-between areas of surface are then filled by linearly stretching the in-between areas of texture across them. A **texture coordinate** is a 2D point, typically with each element on [0, 1], that specifies a location

Figure 4: *Heightfield at sunset. You can simulate the atmospheric perspective by painting a gradient across the heightfield's texture map in Photoshop.*

in the image. By assigning a texture coordinate to each vertex we establish the mapping. By convention, texture coordinates are often referred to by the variables (u, v) to distinguish them from points in screen space.

If you have a really big area to cover (like an island...) and a fairly unstructured texture map (like grass or sand), it is more efficient to tile the texture over the area than it is to make unique grass for every bit of the island. To do this, simply assign texture coordinates that are larger than 1. By convention, this means "tile the texture". If your texture coordinates range from (0,0) to (100,100), then the texture will tile in a 100×100 grid.

Sometimes we need two texture coordinates at the same location on the surface. For example, we might want to texture map a cube so that the left edge of one side aligns with the right edge of its neighbor for a tiling texture. To do this, one creates *two* vertices with the same (x, y, z) but different (u, v). Fortunately, you don't have to do that for the heightfield in this project and you don't need texture coordinates for your other models.

5.9 Texturing

If you're inspired by the artistic side of this project, here are some tricks for making terrain. This is for your information only–you won't receive extra points if you go to this length on the heightfield part of the project, and it certainly isn't required.

Knowing how to use Photoshop is a useful skill as a programmer even if you aren't making pictures. The filters and commands are handy for adjusting large amounts of data that happens to be encoded in an image format...like our heightfield. I now describe two ways that you might approach this project.

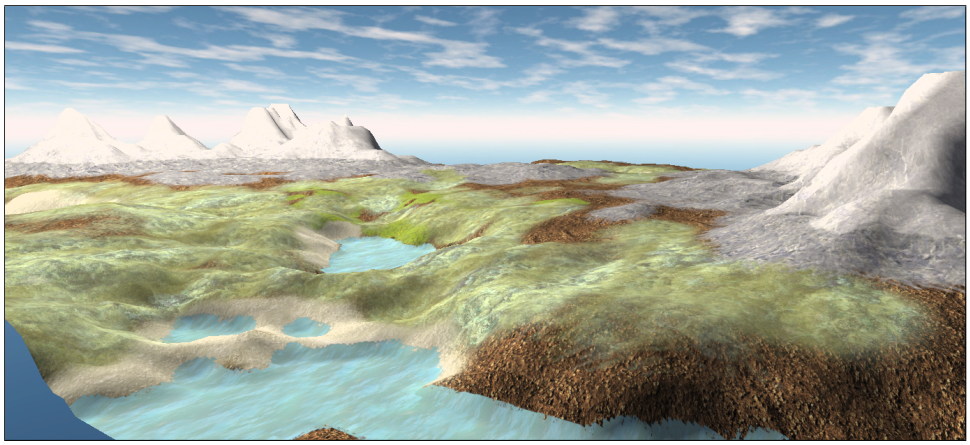

Figure 5: *It takes about an hour of texturing work in Photoshop to paint a terrain like this.*

5.9.1 The Multiple Heightfield Method

This method is well-suited to collaboration but can be tricky to get individual features in the terrain to line up well with one another. It also doesn't handle transitions between land types particularly well.

The basic idea is to create a different height field for each type of terrain and then let them stick through each other to reveal the appropriate one at the appropriate location. This allows you to create several height fields independently and texture them with small, repeating tiles. It also lets you create truly transmissive water by setting the `transmission` member of the `UniversalMaterial::Specification` in your scene file.

The follow section describes in more detail how to create a height image. You can follow those instructions and then adjust the result to yield a series of height fields that you then texture appropriately.

5.9.2 The Megatexture Method

This method gives you easy control over the interaction between different terrain types, but requires making some huge data files and is hard for multiple people to collaborate on.

It took me about an hour to create the terrain in Figure 5. I first generated a 128×128 grayscale noise field with a Photoshop filter called "Render Clouds". I ran the filter a few times until I liked the pattern it made. I then used the Levels command to stretch the histogram to fill the entire value range.

I used the Curves command to draw a profile of the terrain. I flattened the first 10% of the value range to create the large "water" regions, made the next 50% of the value range slope up linearly, and then steeply curved the mountainous areas and rounded their top.

I saved the grayscale to produce the heightfield and then resized it to 8192×8192 to make the color texture. Using a single texture for the entire world is the basic idea behind John Carmack's **megatexture** [Dornan 2006], which you might have heard about in the popular gaming press. It is easy to do this for a heightfield that

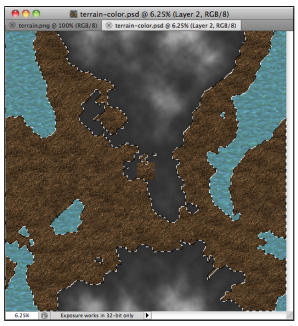

Figure 6: *2D terrain texture map in progress in Photoshop.*

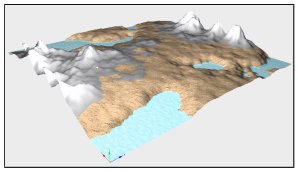

Figure 7: *In-progress texture map viewed in 3D.*

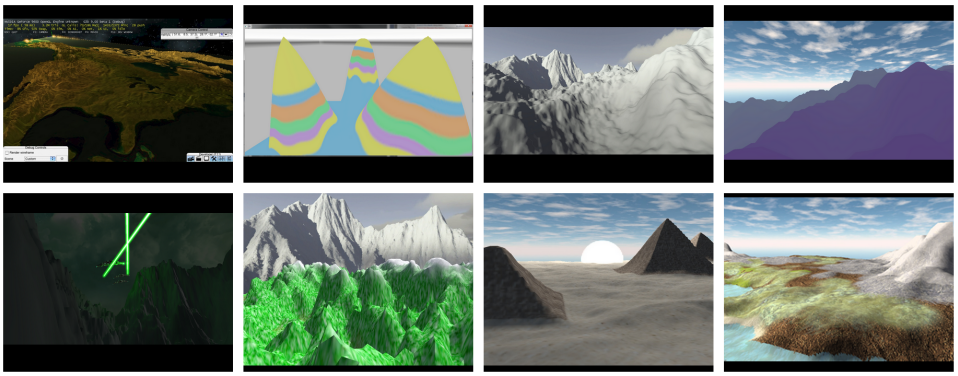

Figure 8: *Gallery from 2010*

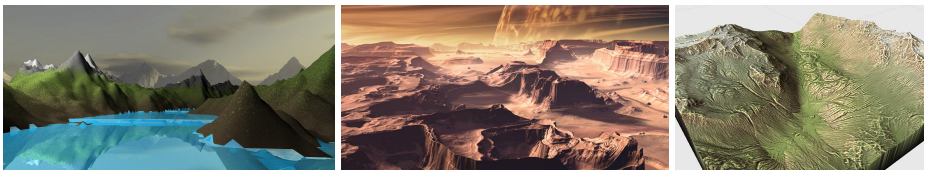

Figure 9: *Other motivating height field renderings*

fits entirely in memory. If the heightfield were bigger we'd need some kind of page table to manage multiple textures, and if we were working with an arbitrary mesh we'd need a more sophisticated texture parameterization.

I used the magic wand tool to select all of the water. I used the clone tool to paint a tiny swatch of water texture I found on Google Images as a semi-unique texture over all water areas. I then repeated this for other elevations with different textures, periodically checking how it looked in 3D.

Finally, I gamma adjusted the entire texture so that it appeared darker in Photoshop. I could have instead altered the `Material::Specification` in the corresponding Scene file to use a gamma factor on load.

From the G3D data documentation I selected a natural sky image for use as the skybox, and placed an appropriate light source to act as the sun. The result is clearly "programmer art"–it is ugly and amateur compared to a professional artist's work, but it is pretty enough that we can tell this technique would yield reasonable terrain in the hands of a talented texture artist and given enough time you could imagine refining it for a later project.

One of the problems with my terrain texture is that it only varies based on elevation. If you look at real terrain, you'll notice that vegetation and rockiness strongly correlate with slope, not just elevation. Making flatter parts have more vegetation and steeper parts more rocky would improve the overall appearance. You can probably imagine use tricks in Photoshop to obtain the derivative of the heightfield, or even writing a program to automatically texture a terrain. The latter would make a nice midterm project!

References

COLLINS-SUSSMAN, B., FITZPATRICK, B. W., AND PILATO, C. M. 2008. Subversion complete reference. In *Version Control with Subversion*. O'Reilly, ch. 9. http://svnbook.red-bean.com/en/1.5/svn.ref.html. 8

DORNAN, C. 2006. Q&A with John Carmack. *Gamer Within* (May). http://www.team5150.com/~andrew/carmack/johnc_interview_2006_MegaTexture_QandA.html. 13

GEOMETRY TECHONOLOGIES, 2010. Geomview manual. http://www.geomview.org/docs/html/index.html. 2, 9