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 3.2 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 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.

Data files also enable people with different skill sets to work on scenes independently. In computer graphics, data files are often created by artists using 3D modeling, animation, and texturing tools, while the source code is written by programmers. When data is outside the program, you can modify and reload it while the program is running. This eliminates the time to compile and run your program from the debugging cycle, which is important for streamlining your workflow. Last week you modeled a scene very slowly by editing source code. Hopefully that experience convinced you that hard-coding scenes is an inefficient way to work! From now on, you'll create data files for constants like scene information and reload them at runtime when adjusting your program.

For this project you will convert your Cubes code to be data-driven, and then incorporate indexed triangle meshes into it. We'll use the G3D `.scn.any` file format for our scenes and the Geomview OFF file format [Geometry Techonologies 2010] for meshes. Both of these are relatively obscure. I chose them because they are similar to more widely used formats but are 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 Educational Goals

On this project you'll learn about:

1. **Principles**
   (a) Indexed triangle meshes
   (b) Heightfields
   (c) Texture mapping
2. **Practice**
   (a) Exporting from Subversion
   (b) Pair programming
   (c) Typesetting equations in LaTeX
   (d) Data-driven programming
   (e) Working with file-format specifications

1.3 Schedule

Out:	Tuesday,	September 14	
Checkpoint:	Thursday,	September 16,	1:00 pm
Due:	Monday,	September 20,	10:00 pm

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

This is a moderately challenging, pair-programming project. You will work with an assigned partner and submit a single solution. Note the that the checkpoint deadline is at the **beginning** of the scheduled lab period–you need to start this project on your own before that lab period.

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 on a bad path.

2 Rules/Honor Code

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 pair-programming project of the semester. You and your partner share a single Subversion repository. 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. I designed the tasks so that doing them in order guides you to the right solutions. You may 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.

Part of this week's project requires reverse engineering two file formats from sample files and source code. I encourage everyone to pool their efforts towards this and share what you learn on the mailing list. You can share any information related to this except for the specific data files and code that you are required to write. For example, you may share a sample OFF file for a pentagon, but not for a cube.

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 `ArticulatedModel::createHeightField`.

3 Specification

Implement the following by extending the Cubes project from last week:

Tip: G3D is installed at /usr/mac-cs-local/share/cs371/G3D/ and is also available at `http://g3d.sf.net`

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

2. Copy and integrate the `Scene` class and associated GUI from the G3D `starter`

code sample. The GUI includes the "reload" button and scene-select drop-down list.

3. Create a `data-files/cornell-box.scn.any` scene that re-creates your hard-coded Cornell Box scene from Project 0 using only data.

4. Manually create a data file `data-files/cube.off` that contains a 1 m^3 cube centered at the origin. Use only a triangle mesh and no quadrilateral polygons. You'll need to create a corresponding scene to test it.

5. Write a procedure that generates a polygonal approximation of a cylinder using only triangles and saves it to an OFF file. Parameterize the cylinder by the height, radius, and number of slices. Do not check generated the file into Subversion.

6. Write a procedure to generate a regular heightfield from a grayscale image and saves it to an OFF file.

   (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, and the horizontal axes scales.

   (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 and (1,1) the lower-right of the original image.

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

7. 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 cs-local that it needs to run to the `data-files` directory. Commit the source heightfield image that you used as well.

8. Create the documentation reports specified in Sections 3.1 and 3.2.

As always, you should remove unused code from your program. This includes last week's hard-coded Cornell Box scene (although you should do so by porting it into a data file). An exception is debugging and unit testing code that you may need again in the future, which is acceptable to retain.

3.1 Checkpoint (Thursday 1:00 pm)

Most projects, like this one, have checkpoint reports. This is something that must be on your mainpage *before* that week's scheduled lab. Checkpoints motivate you to get to the hard part before lab, so that we're together when you hit the crux of the lab. They also give me a chance to verify that you're on the right track and offer suggestions if not.

Checkpoints are largely pass/fail. Your work should demonstrate a good faith effort but need not be perfect to receive full credit at this stage.

For this checkpoint:

1. Typeset the following equation into your documentation using LaTeX embedded in Doxygen:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \left[\frac{f(x+h) - f(x-h)}{2h} \right]$$

2. Commit the files for the `Scene` class and connect up the GUI for it.

3. Commit a (mostly) correct `data-files/cornell-box.scn.any` data file to Subversion.

3.2 Report

1. Show pictures of:

   (a) Your data-driven Cornell Box scene

   (b) Your manually created cube

   (c) Your generated cylinder

   (d) A top-view of a flat heightfield, showing the tessellation

   (e) A visually interesting scene using your heightfield, cylinder, cube, and any other objects of your choice. You may turn off the wireframe outlines for this shot to improve the appearance.

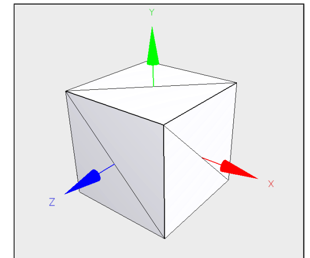

Figure 2: *A "hand crafted" cube.*

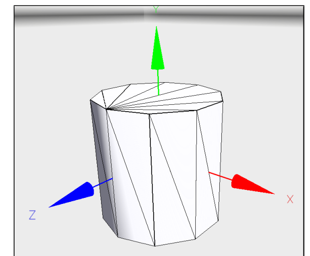

Figure 3: *A tessellated cylinder.*

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?

   (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 propose an algorithm for computing a more efficient tessellation for a 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:

[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.

(a) How many hours you spent **outside** of class on this project on **required** elements, i.e., the minimum needed to satisfy the specification.

(b) How many additional hours you spent outside of class on this project on **optional** elements, such as polishing your custom scene or extreme formatting of the report.

(c) 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?

(d) What did you learn on this project (very briefly)? Rate the educational value relative to the time invested from 1 (low) to 5 (high).

4 Evaluation Metrics

For team projects I almost always assign the same grade to both partners. That means that you are responsible for the entire project, not just the parts that you did. You're also responsible for the correctness and clarity of code that you exported from Subversion to start the project, even if you were not the original author. So make sure that you know what code is in your project and that you understand it all!

To evaluate your project, I will check your project out from Subversion as of the deadline time. I will then run `icompile --doc` to generate the final report and documentation. I will read sections of your source code, the report in the `index.html` page generated by Doxygen, and sections of your documentation as generated by Doxygen. I may run your program, but I will primarily investigate its functionality by the description that you provide in the report. Note under this scheme, that the artifacts from your creation of and experimentation with the program are more important than the executable program itself. For many projects you can receive a favorable evaluation even if your program does not compile or execute.

As described in the *Welcome to Computer Graphics* document, I will evaluate your project in several categories:

- Mathematical (algorithm, geometry, physics) correctness
- Adherence to the specification
- Program quality
- Report quality

Some questions I consider when evaluating the source code are: Is it possible for someone unfamiliar with it to find specific routines quickly? Is the code easy to understand? Does it make good tradeoffs between efficiency, clarity, and flexibility? Are data structures used effectively? Are the algorithms correct? Are the geometry and physics correct?

When evaluating the report, I consider: Do the experiments adequately explore the correctness, performance, robustness, and parameter space of the algorithm?

Are known bugs made clear, along with how you tried to solve them? Are appropriate sources cited for algorithms and code? Does the overview documentation guide a reader to the relevant source code documentation? Is the architecture of the program clear?

The report and code should both be as concise as possible without compromising clarity. Use the papers we've read as examples of how to describe experiments compactly.

5 Implementation Advice

5.1 Teamwork

Three ways to work with a partner are:

1. Pure pair programming: one person drives a computer while the other watches and makes suggestions.

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 partner is and what the project is. This week I ask that you mostly **work in a pure pair programming style**. The project is designed to have two people actively collaborating on each of the task, and to have you progress through the specification in order.

You may switch to the other modes for the heightfield scene creation task at the end. If you do so, 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 partner e-mail whenever you commit code.

The first practice helps you distinguish whether you or your partner broke something–if it was your partner, you know to look at the revision control history or code that he or she wrote rather than the code you added. The second helps avoid breaking the build on your partner. 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 partner knows when new updates are available, and what will happen if he or she does update.

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.

When pair programming, it is important that both partners share responsibility for the actions on the computer even though only one is actively typing. The non-typing partner should be thinking one step ahead so that you never lose momentum. That partner should also be checking what is typed for bugs as it goes in, and can use a second computer 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 a partner, especially a new partner, 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 assignment, schedule a few pair programming sessions. I recommend pre-scheduling five hours of time outside of class during the week. 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 partner. 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, it doesn't need comments because it is obvious what the interface is and how the implementation works.

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 you will not lose points for that.

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/1-Meshes/meshes-<group>
```

Where you should replace `<group>` with the name of your group, which you should have received Monday night by e-mail.

Most of G3D's GUI routines have you pass a pointer (yes, the dangerous thing I told you to be careful with last week!) to the 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;
```

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.

The `.scn.any` file format has two extensions because it has two layers of structure. `G3D::Any` is a class that simplifies writing simple, structured data to disk and

reading it back. By convention, files containing `G3D::Any` data end in `.any` so that text editors can detect them and so that you can set up appropriate file associations.

The `Scene` class that you imported from the `starter` project happens to use a particular structure of `G3D::Any` for its data files. You can change it it you'd like. The `Scene` class has minimal documentation. That means that you need to figure out the file format from reading the implementation code and from looking at the `crates.scn.any` file from the `starter` project. Fortunately, it is fairly friendly, and most of it is designed to mimic G3D classes that go by similar names.

The `crates.scn.any` file contains some animation data that we won't need yet, and has more complicated materials than we require. Position objects using data like

```
object = model(CFrame::fromXYZYPRDegrees(x,y,z,yaw,pitch,roll))
```

and set materials with data like

```
materialOverride = Material::Specification {
    lambertian = Color3(1, 0, 0)
}
```

or

```
materialOverride = Color3(1, 0, 0)
```

Both of these code snippets should look familiar, since they closely match the hard-coded constants you set in the previous project.

5.5 Printing `std::string`

`printf` 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 pictures in the data file and in your notebook–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

Tip: Things get really confusing in the two `for` loops if you name your Image1 variable "height." Consider "elevation."

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.

G3D contains several classes for loading grayscale images, including `G3D::Image1`, `G3D::Image1uint8G3D::GImage`, and `G3D::Texture`. I recommend `G3D::Image1` for creating the heightfield. Think about why are there so many image classes in the library, and why I recommend this one for this application.

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,

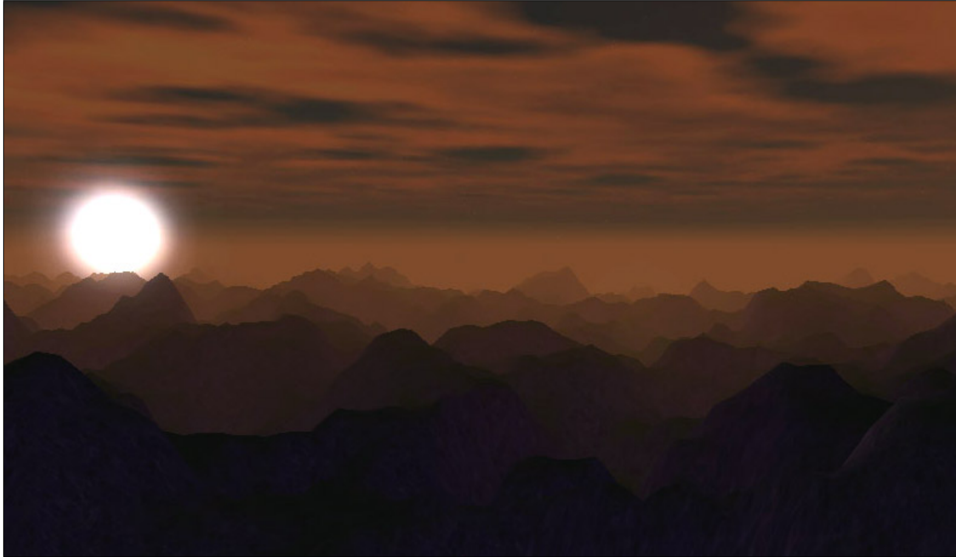

Figure 4: *Heightfield at sunset. You can simulate the atmospheric perspective by painting a gradient across the heightfield's texture map in Photoshop.*

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 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.

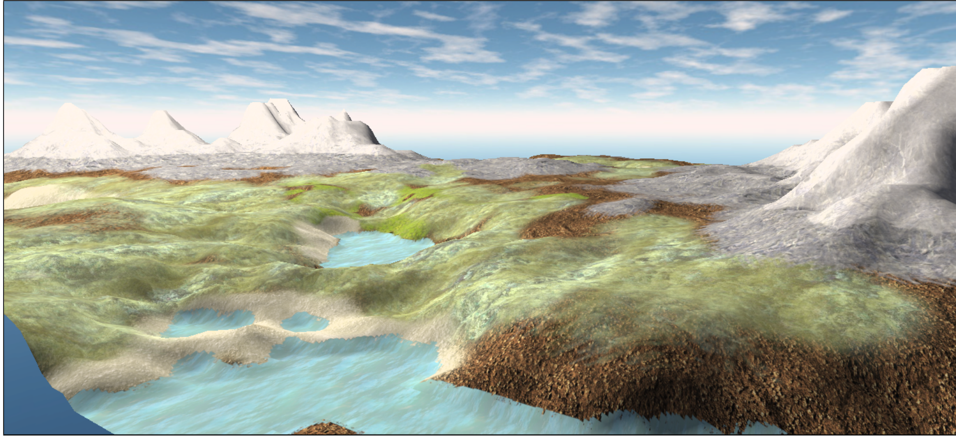

Figure 5: *It takes about an hour of texturing work in Photoshop to paint a terrain like this.*

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.

It took me about an hour to create the terrain in Figure 5. I first generated a 128×128 grayscale Cloud field with a Photoshop filter. I ran the filter a few times until I liked the pattern it made. I 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.

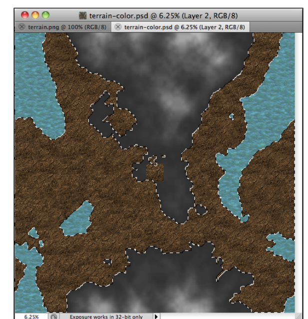

Figure 6: *2D terrain texture map in progress in Photoshop.*

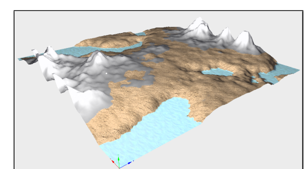

Figure 7: *In-progress texture map viewed in 3D.*

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 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 high quality 3D art, but it is pretty enough that we can tell this technique would yield reasonable terrain in the hands of a talented texture artist.

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 final 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

Index

.scn.any, 2, 9

ASCII art, 10

bool, 9

checkbox, 3
checkpoint, 4
coding conventions, 8
comment, 11
Cornell Box, 4, 5
cube, 5
cylinder, 4, 5, 11

data-driven programming, 1
debugging, 8
debugPane, 9
Doxygen, 5

fprintf, 9

G3D::Any, 9
G3D::GApp, 9
G3D::GuiPane, 9
G3D::Image1, 11
G3D::TextOutput, 9
gamma, 14
GUI, 5, 9

hard-coded, 2
heightfield, 1, 4, 5

indexed triangle mesh, 1

LaTeX, 2, 5

megatexture, 13

OFF, 2, 4, 9

pair programming, 7
preprocessor, 11

report, 5

Scene, 5, 9
scene, 4
starter, 4, 9
Subversion export, 8
surface, 1

teamwork, 7
texture, 12
texture coordinate, 12
texture mapping, 12

wireframe, 3