5-2012

Virtual "University of Arkansas" Campus

Seth Williams
University of Arkansas, Fayetteville

VIRTUAL ‘UNIVERSITY OF ARKANSAS’ CAMPUS

VIRTUAL ‘UNIVERSITY OF ARKANSAS’ CAMPUS

A thesis submitted in partial
fulfillment of the requirements for the degree of
Bachelor of Science in Computer Science

By

Seth Williams

April 2012
University of Arkansas

ABSTRACT

The aim of the Virtual Campus project is to develop a way to automate building 3D virtual worlds using map and other data from the real world. A demonstration presented in this BS honors thesis uses automated tools to build the University of Arkansas campus from data supplied by the UA Center for Advanced Spatial Technology (CAST). At present, in virtual worlds like Second Life and Unity, terraforming of terrain is a manual process, and it can take days to weeks to build a landscape. But by using existing map data, we are now able to automate that process. Working with my Andrew Tackett and Jonathan Holt in our senior design project, we created a script (program) which will take height map data in `.raw` format and texture data in photographic format and which uses this data, along with functions built into the *Unity* API, to create a terrain with the proper heights and textures. A second script that I developed for this thesis allows a user to identify certain colored regions as buildings. The script then populates the regions defined as buildings with "sugar cube" buildings (white blocks), which the user can keep, remove, or replace at their discretion. The end result is a 3D landscape that avatars can visit. The design enables users to further populate the virtual campus by replacing the "sugar cube" buildings with their own more detailed 3D building models, created using outside programs. Using such an environment, students can go to a virtual class or meet together at a virtual union. At present, many virtual campuses exist in Second Life, but none that we know of accurately reflect the geography of their campus.

This thesis is approved for recommendation
to the Honors College

Thesis Director:

____
Craig Thompson

Thesis Committee:

____
David Frederick

____
Russell Deaton

ACKNOWLEDGEMENTS

I wish to thank all individuals who made this thesis possible. To my thesis advisor, Dr. Craig Thompson, thank you for providing me with an interesting and engaging idea to research, as well as your instruction and encouragement throughout this process. To Dr. David Fredrick, thank you for allowing me the use of the Game Design Lab and for showing a high level of enthusiasm for the completion of this project. I hope that you will find use for the programs that were created. To Andrew Tackett and Jonathon Holt, thank you for all of the hard work you put into this project to help create the Terrain Automation script. I thank Angie Payne and Snow Winters for providing me with the height map and texture for the development of these scripts. Finally, I thank my family, my friends, and my fiancée, Courtney May, for their love, support, and prayers while I worked on this project. Without the amazing support I received from this group of people, none of this would have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Problem

Game design and virtual world development have both become large industries in recent years. 3D virtual worlds like Second Life are multi-player social worlds where humans in the form of "avatars" meet with one another, use voice or chat to communicate, and can own virtual land, build virtual buildings and script objects. Millions of people have downloaded the free Second Life client (like a browser). Currently, surprisingly, the only way to create a 3D virtual model of a real-world region is to do so manually. This involves hours-to-weeks of elevating terrain "by hand," sculpting and placing roads and buildings, and filling terrain locations with vegetation. The result is either a fantasy world or else a surrogate for a real world area that is so inaccurate that we are unaware of anyone succeeding in developing an accurate 3D virtual world model of their surroundings (with the exception that accurate military 3D simulations do exist).

Real world 3D elevation and land use maps are available. Today's satellite imagery is very detailed, and has led to many developments in the field of global, interactive maps. It is difficult for us now to remember what life was like before Google Maps and other GPS navigational applications and machines took away the need to carry paper maps in order to travel to places that we had never been before. However, at present, there are no established methods to import such map data into a game engine or a virtual world to create an accurately terraformed terrain. In addition, there is no current method to use a script to parametrically populate an urban region with vegetation, roads, and buildings. If an automated means of building 3D worlds from map and land use data could be developed, it would expedite the process of building 3D games – but more than that, it would find application in so-called "serious games" in architecture, archaeology, military strategy, geology, and many other fields.

Even with the availability of 3D maps, 3D virtual worlds and gaming, there is a striking disconnect between these three different communities. Though it could be argued that the virtual worlds community is ultimately derivative from the game design community, we see that even the most detailed virtual worlds simply do not live up to the graphical standards now expected from today's PC-based games. These games are often on the cutting-edge of graphical quality, where virtual worlds like Second Life have fallen behind. Likewise, there seems to be a chasm between the mapping community, the game design and the virtual world communities. Video games have used real-world locations before: *Spider-Man* used developer-generated iterations of New York City; *Assassin's Creed* used historic representations of cities across Europe and the Middle East; and *Grand Theft Auto* use parodical versions of cities across America. None of these games, however, used true topographical data. To my knowledge, only one very recent game has used real topographical data to create its levels: *SSX*, a snowboarding game published by Electronic Arts used satellite data from NASA to create iconic mountainsides like Mount Everest and peaks from the Himalayas for their players to traverse [1]. But this is just one game out of thousands.

Part of this disconnect exists because the mapping community itself is a fractured entity. There is not a standard geographic information file type. The most commonly seen types are GeoTIFF, ArcGIS, and DEM (provided by the United States Geological Survey). Unfortunately, all of these file types are incompatible with the *Unity* game engine. Most, as it stands now, are not even compatible with the same programs that would be capable of reading another one of the file types. In a similar way, both the 3D virtual world and gaming communities are fragmented without common standard architectures or representations. Instead, most virtual worlds and games are built from the ground up.

1.2 Objective

The goal of this project is to begin to bring these three separate communities together by creating an automated way to aggregate the data collected by the mapping community with a 3D virtual world game engine used by game developers and virtual world enthusiasts. This project's main objective is to create a series of scripts for the *Unity* game engine that will automate terraforming and populating a region in a 3D virtual world with real world map data. The resulting 3D virtual worlds then can form the basis for 3D gaming environments for both recreational and serious gaming applications.

1.3 Organization of this Thesis

Chapter 2 covers background on the thesis. Chapter 3 describes the design, implementation, and results. Chapter 4 summarizes the thesis and describes future work.

2. BACKGROUND

This section describes how and why I became interested in automating the process of creating 3D virtual worlds.

At the University of Arkansas in the fall semester 2010, I took a course from Dr. David Fredrick in the Classics Department in which the class modeled portions of ancient Pompeii using a 3D game engine called *Unity*. This powerful game design tool is available in two forms: the free *Unity Basic* and *Unity Pro*. We used *Unity Pro* in the class, and I was stunned by the amount of realistic detail that could be put into user-created games, for instance, detailed mosiacs and wall paintings.

However, I was not skilled at using the array of terrain brushes, which enabled users to add textures, change elevations, and add details like trees or grass. These tools were not very easy to learn or to master. In part, this is because most game developers use third-party software like *Vue*, *Sketchup*, or *Maya* to create their terrains because of the ease of use of these programs' modeling features so *Unity* developers had little cause to create sophisticated tools because their user base was already familiar with this third-party software. Even so, the process of building objects and terrains was still a manual and very time consuming process that often did not match a real world location.

This led me to wonder how developers could more efficiently make games that drew from real-world locations at multiple levels of detail. Reflecting on games I had played and reviewing others led me to the conclusion that, while many games depict the most iconic portions of real world locations, they rarely, if ever, use actual map data available for the entire area. The results are games where large portions of a city rests on flat ground, where it is known that these areas are not flat at all in the real world. Though not bothersome enough to break the concentration of players whipping around the New York City skyline as Spider-Man, this lack of detail is disappointing especially given that it would not tax gamers' machine resources of today (or even of five or more years ago) if the settings were more realistic.

While I was taking Dr. Frederick's course, I was also taking Dr. Craig Thompson's Virtual Worlds course. He proposed a challenge problem of using *Open Simulator* to create a large-scale "Virtual Earth" or a "Virtual Mars" based on map data from the Center for Advanced Spatial Technologies, which would involve identifying and removing roadblocks on a path to this goal. I began to consider how to do the project in *Unity*, though my knowledge concerning that game engine was still limited at the time.

The *Unity* engine already supports a type of elevation data known as a grayscale map. Using this data type, *Unity* will terraform a region at the proper points to form a 3D terrain. However, *Unity* currently does not support the use of a `.DEM` file, a widely used format in real-world mapping community. Users would need to convert the `.DEM` files into grayscale maps by via a translator similar to the *SimpleDEMviewer* [2]. Researchers from Intel had explored this problem using *Open Simulator* in a virtual world grid known as *ScienceSim*. There, they created the terrain of Yellowstone National Park, complete with proper elevations and textured land use (see screenshots below) [3, 4]. This experiment did not include any flora or fauna, but did provide a glimpse the future of virtual worlds leading towards a 3D virtual earth!

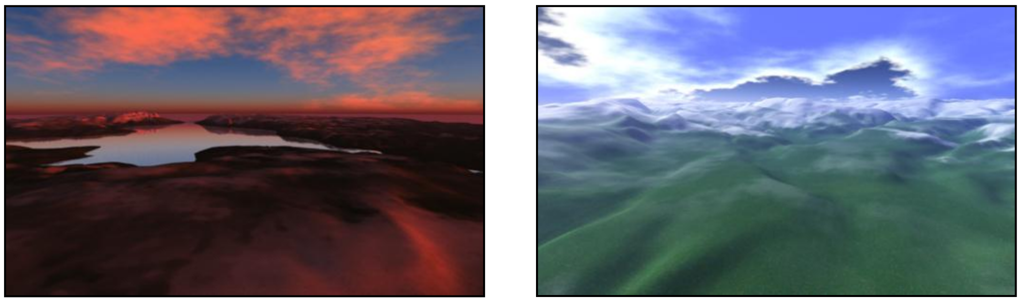


Figure 1: Intel Labs' Yellowstone Park Simulation

I chose to use the *Unity* game engine over *Open Simulator* because it has an online application programming interface (API) that I was familiar with. Also, while the functions that the API provides are not well-documented, there is a dedicated base of *Unity* users who can provide help since they have used several of the functions in the API to create their own games, some of which have been published by video game companies. The functions included with the *Unity* system provide their users the ability to script any object that can be created by the engine or brought into the engine from outside sources. For this project, the functions of interest were those that allow users to alter the terrain.

Previous work on *Unity* terrain alteration proved useful to the development of the scripts developed in this project. The *Terrain Toolkit*, designed by Sandor Moldan [5], enables a user to alter terrains in a variety of ways and to create new terrains randomly. The terrain creation tools use popular patterns, such as Voronoi diagrams or fractals, to create terrains with landmarks, like mountains. The *Terrain Toolkit* includes procedures for blending textures according to heights and the sharpness of angles present in the terrain. It also provides functions which simulate the effects of erosion on a landscape to make a region more realistic and "playable." For this project, the most useful function was a smoothing function. That function dynamically checked the height values at all points on the map and used an averaging algorithm which creates a new height map that is completely smooth. This was useful because the height maps that the *Unity* engine can use are all in `.raw` format. This means that, for each pixel in the height map, there is a given integer value, from 0 to 255 (for an 8-bit-per-pixel depth) which determines the height of that particular point. 0, or black, is the minimum height, while 255, or white, is the maximum height. Using these height maps in *Unity* with no smoothing results in a stepped landscape, which is far from realistic. While an artificially smoothed level is also not entirely realistic, it is more realistic than a stepped terrain and also easier for players in the world to traverse. The *Terrain Automation* script, developed by this project, uses Moldan's smoothing function to automatically smooth imported terrains.

The *Terrain Toolkit* was also useful in helping my group figure out how to create editor scripts, which are scripts that function on objects in a *Unity* environment outside of `play` mode. However, the *Terrain Toolkit* only functioned as a random map generator. Even to date, there is no research I could identify in the area of creating true-to-life representations of real-world regions programmatically in *Unity*.

However, if realistic regions do come into being, the *Unity* community has done extensive research in the area of creating virtual world connectivity in *Unity*. ReactionGrid, the developers of *Open Simulator*, has made a free-to-use virtual world based on the *Unity* game engine called *Jibe*. The significance of this is that anybody can create their own regions inside this virtual world, or even make their own separate virtual world based on real-life locations, should they choose to do so. Virtual meeting spaces have been created in *Second Life* and in *Open Simulator*, but none of them are based on real-world locations. The ability to dynamically create the earth in a virtual format playable through a simple web browser would give people this ability in a way that they never had before.

Even if we do learn how to build virtual worlds on today's real world, we should not be confined to mapping out present locations. There is extensive data about the geography and layout of ancient cities and landmarks that would provide users a glimpse into the past and, perhaps, would provide a valuable teaching tool for professors of antiquities and archaeology. Dr. Fredrick is tapping into this potential by recreating ancient Pompeii, and by having his students recreate scenes from ancient literature as video game levels. Often, myths and stories were set in real locations, with certain fictional landmarks added to the region to give the stories a sense that they happened in close proximity to the people who heard them. Some of the geographical information about these regions has been lost over time. But, some has been preserved or recreated by the mapping community. I can think of no better way to preserve ancient (as well as present) geography, archaeology, history, and literature than to create interactive reserves of this information so that users may one day be able to engage in a 3D world that they may have only been able to read about previously.

A virtual real world could truly be at our fingertips. That is the motivation for this project.

3. DESIGN, IMPLEMENTATION, AND RESULTS

3.1 Adding Real-World Terrain and Ground Use Textures to a Virtual World

At the beginning of this project, the goal of automating the building of 3D virtual worlds from existing map data looked immediately attainable. I knew that map data existed in many formats with free access to the general public. I also knew that high resolution map data was available for a cost, like the USGS maps. The surprise came in the Virtual World course when I discovered that none of this map data played well with 3D virtual worlds, including the *Second Life*, *Open Simulator,* or *Unity* system.

For my Capstone project, I decided to work on this problem and recruited two other students, Jonathan Holt and Andrew Tackett. We met with two CAST employees, Snow Winters and Angie Payne, who provided us with a satellite image of the University of Arkansas campus and the surrounding area, as well as a height map of the area in `.raw` format. Payne worked with the *Unity* engine for a separate project and informed us that the Digital Elevation Maps (`.dem` file format) that we were originally planning to use were not compatible with the *Unity* system, and that only height maps in `.raw` format would work for our purposes. Winters and Payne suggested that we learn how to use *Vue* and *Sketchup* because both are popular tools for creating terrains for use in the *Unity* engine. We decided that this was outside of the scope of our project, and that we would only use `.raw` height maps because they were the only ones we could use to *automate* terrain generation in *Unity*.

The original way that we approached the Capstone project was an online search for any projects that had done something similar to what we were trying to accomplish. This led us to the aforementioned *Terrain Toolkit* as well as one other script which we found on the Unify

Wiki, which is a collection of knowledge and programs contributed to by *Unity* users. This script was called the *Terrain Importer* [6]. It required that users input a series of data entries into a `.txt` file and then place that file into the `Assets` folder of the *Unity* project in which they were working. However, it provided no graphical user interface (GUI) for the user, which meant that if the user did not correctly input some piece of data, they would have to go back to the text file and check each line to determine where in the file the mistake had been made. Additionally, the code itself was opaque. We analyzed the code for two weeks before determining that if we could not understand its functions, then it would be prohibitively difficult to add further functionality to that code. So, we abandoned that script.

In considering requirements and goals for our project, we determined that we wanted to ensure that the user should have a simple, straightforward user interface for specifying parameters they wanted to use to create the terrain. We also wanted the user to use a script in the editor (which runs at game development time) to reduce the overhead of creating terrain at the beginning of play mode.

This meant needing to learn how to create *Unity* editor scripts, which are scripts that act on the objects present in a game scene before the user launches into play mode. The *Terrain Toolkit* provided a way to learn how to script a user interface that would be visible from the inspector of the terrain object to which our script would be attached. The toolkit was coded in C#, a familiar language. The remaining task involved figuring out exactly what parameters the user should specify in order to create all aspects of a given terrain.

To gain this knowledge, we decided to first use a manual process which involved using *Unity*'s built-in functionality to create a terrain, applying a height map, and then adding a texture to the terrain.

The first step in that process is creating a blank terrain. To do this, a *Unity* user goes to the `Terrain` tab and selects `Create Terrain`, as shown in Figure 2 below.

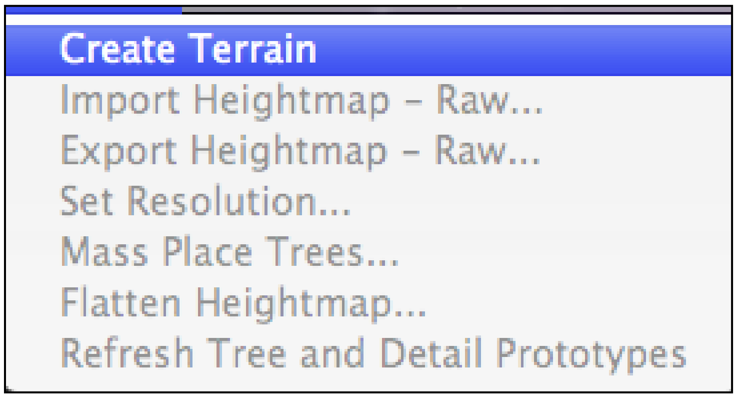

Create Terrain
Import Heightmap - Raw...
Export Heightmap - Raw...
Set Resolution...
Mass Place Trees...
Flatten Heightmap...
Refresh Tree and Detail Prototypes

Figure 2: Create Terrain Function

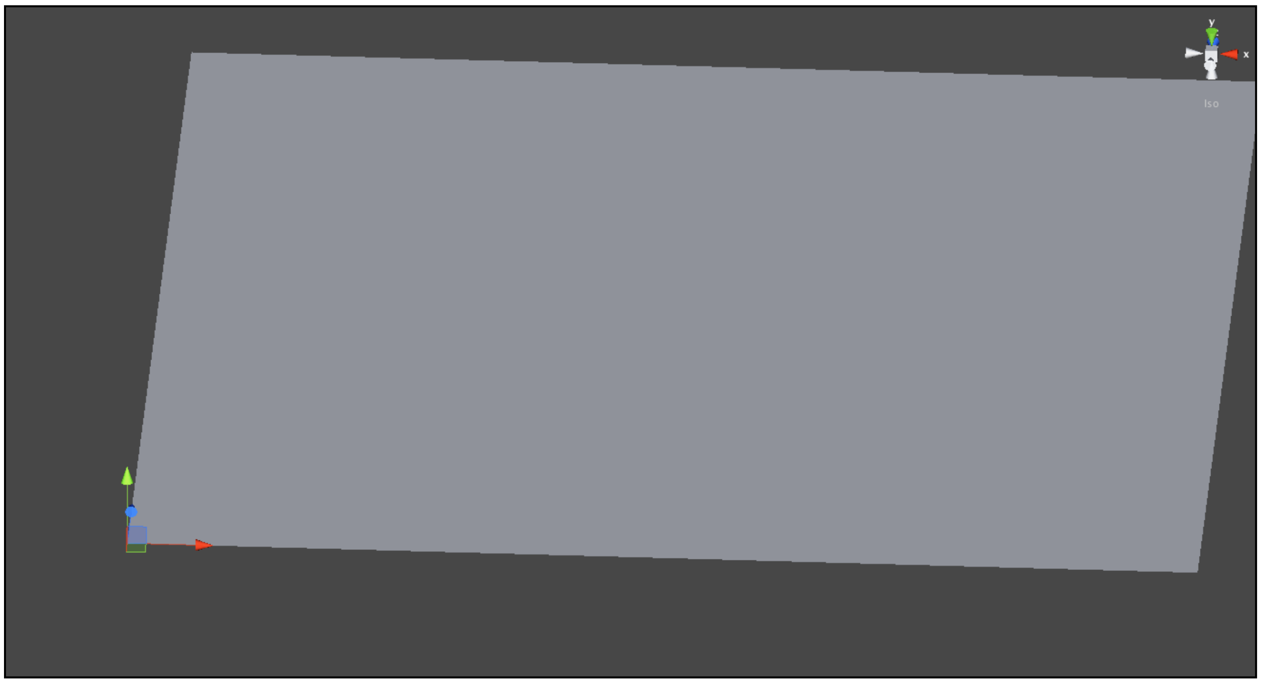
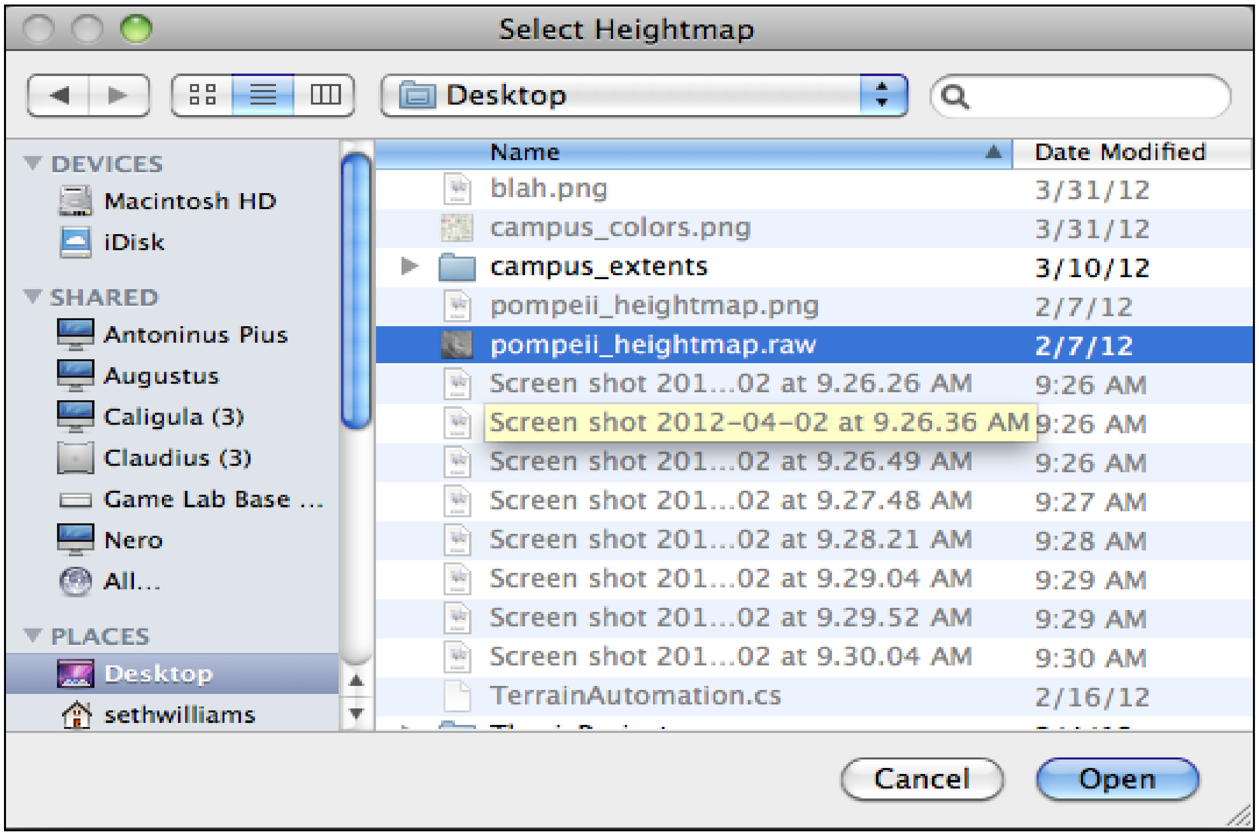


Figure 3: Blank Terrain in Scene View

This creates a blank terrain in the scene view as shown in Figure 3 above. The user then returns to the `Terrain` tab and selects `Import Heightmap - Raw`.... This takes the user to a file dialog box where they select a file in `.raw` format as shown in Figure 4 below.



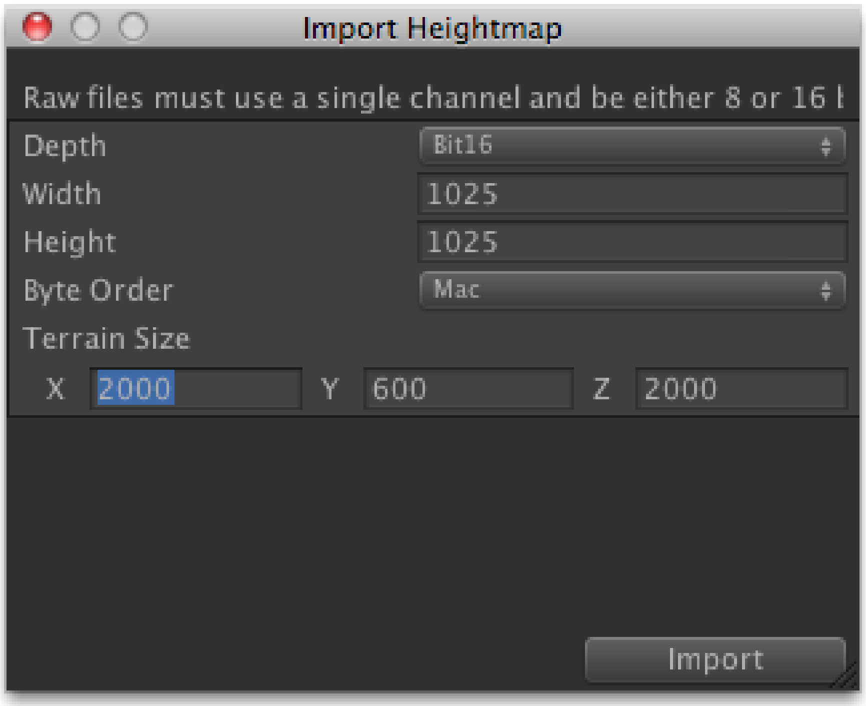
Name	Date Modified
blah.png	3/31/12
campus_colors.png	3/31/12
campus_extents	3/10/12
pompeii_heightmap.png	2/7/12
pompeii_heightmap.raw	2/7/12
Screen shot 201...02 at 9.26.26 AM	9:26 AM
Screen shot 2012-04-02 at 9.26.36 AM	9:26 AM
Screen shot 201...02 at 9.26.49 AM	9:26 AM
Screen shot 201...02 at 9.27.48 AM	9:27 AM
Screen shot 201...02 at 9.28.21 AM	9:28 AM
Screen shot 201...02 at 9.29.04 AM	9:29 AM
Screen shot 201...02 at 9.29.52 AM	9:29 AM
Screen shot 201...02 at 9.30.04 AM	9:30 AM
TerrainAutomation.cs	2/16/12

Figure 4: Selecting the Height Map

The height map that CAST provided is in the `campus_extents` folder in the figure. Next, the user inputs the dimensions of the height map. They input the byte order of the height map, which we discovered was "big endian" for a Mac machine and "little endian" for a Windows machine![1] The user also inputs the depth of the height map, which equates to the number of bits per pixel for the image of the height map. Finally, the user inputs the dimensions of the terrain that they wish to create. The x- and z-values for the terrain should match the dimensions of the height map to ensure that the height map and the terrain will match up

[1] Big endian means the first byte is the most significant; little endian means the last byte is the most significant.

perfectly. The y-value is the maximum height of the terrain in "world units." *Unity* suggests that each world unit is equivalent to one meter, which maps only granularly to the real world.


Import Heightmap

Raw files must use a single channel and be either 8 or 16

Depth: Bit16
Width: 1025
Height: 1025
Byte Order: Mac
Terrain Size
X 2000 Y 600 Z 2000

Figure 5: Manual Import of Height Map

The height map that we received from CAST had dimensions of 1025x1025, because *Unity* requires height maps to be in a "power of two plus one" format for each dimension in order to display properly in the system. Therefore, our terrain's width and length dimensions were 1025x1025. We estimated the maximum difference in height from the lowest point to the highest point of the region to be approximately 70 meters. The result after this step is a bumpy white terrain, as evidenced in the picture below.

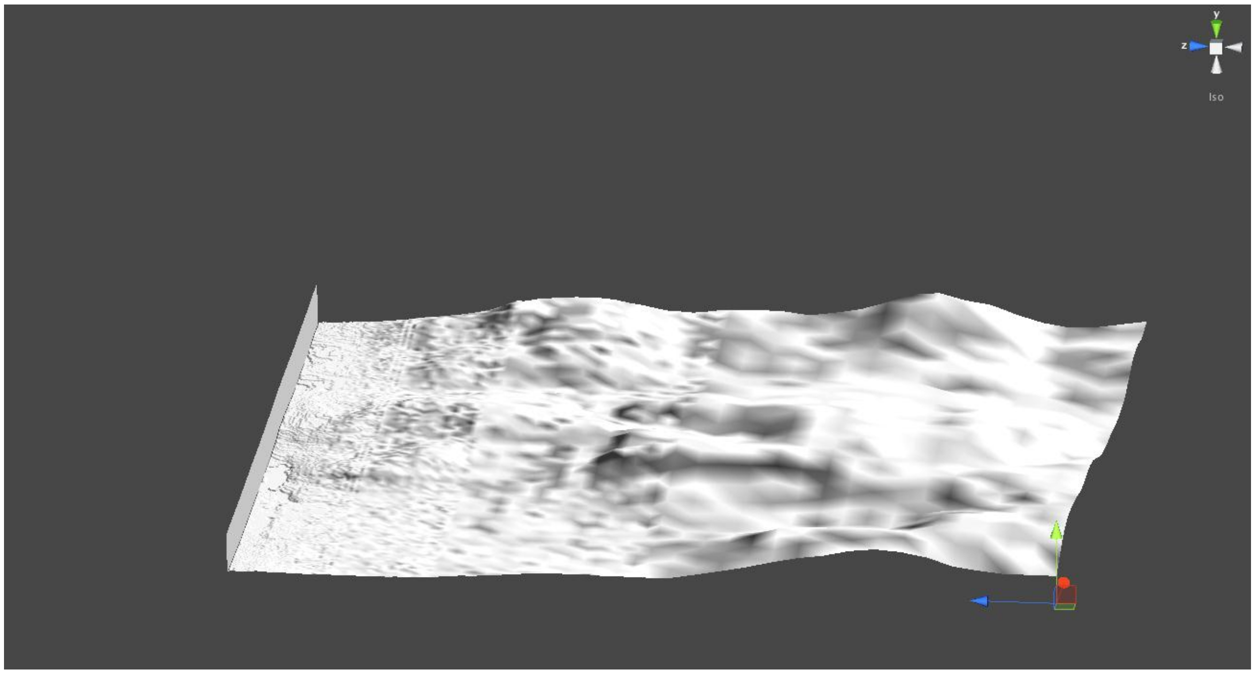
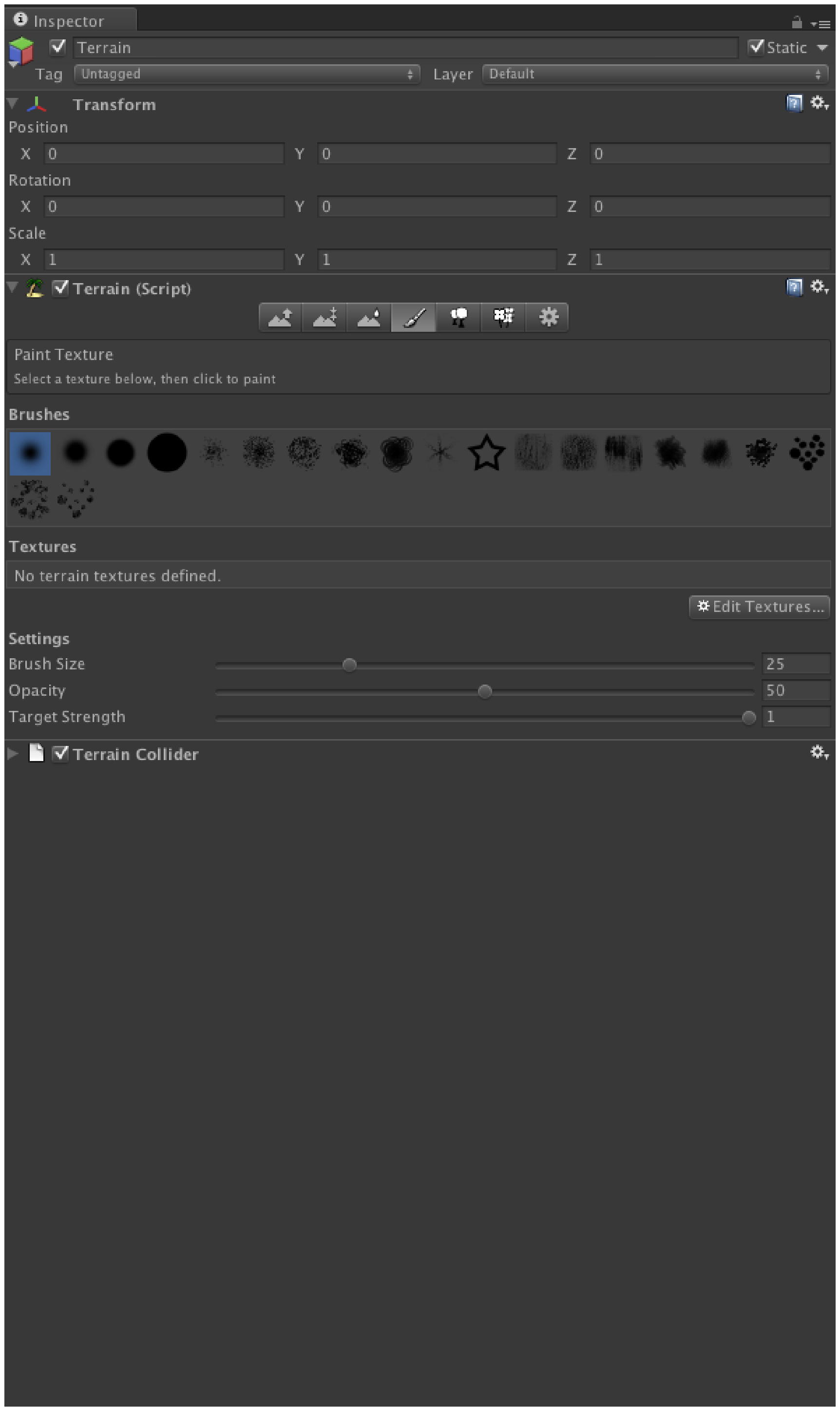


Figure 6: Terrain After Height Map Import

As shown in Figure 6, there was some artifact with the `.raw` file that we imported that affected the entire left edge of the terrain pictured. It read that edge of the height map as all white values, leading to a "wall" on the terrain map. We never discovered what caused this issue, though we suspect it occurred when we cropped the height map in Photoshop to fit the "power of two plus one" constraint.

The next major step for a user wishing to create a realistic looking terrain is to add the texture on to the terrain. This is done by highlighting the terrain in the `Scene Hierarchy`, which opens up the terrain's `Inspector` view. All of the *Terrain Inspector*'s tools are visible in the screenshot shown in Figure 7 below. There are a multitude of brushes, from circular shapes to obscurely used star shapes, which can be used to paint the terrain with textures, or to elevate or compress the terrain heights down to user input levels. Users can even add vegetation to the terrain in the form of grass and trees using these brushes. From this screen, the user must switch to the textures tab in the terrain editor.



Inspector

Terrain ✓ Static

Tag Untagged Layer Default

Transform

Position

X 0 Y 0 Z 0

Rotation

X 0 Y 0 Z 0

Scale

X 1 Y 1 Z 1

✓ Terrain (Script)

Paint Texture

Select a texture below, then click to paint

Brushes

Textures

No terrain textures defined.

Edit Textures...

Settings

Brush Size 25

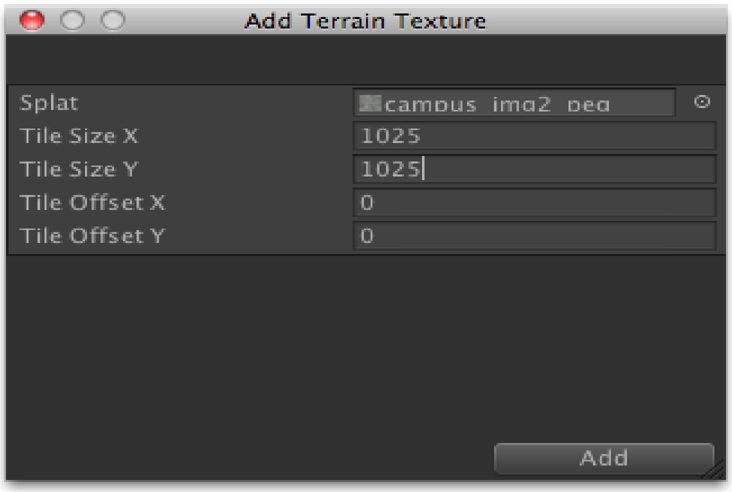
Opacity 50

Target Strength 1

✓ Terrain Collider

Figure 7: Terrain Object's Inspector

Clicking on the `Edit Textures` button prompts the user to add a texture to begin the process. Users then select the texture they wish to use through the following menu.



Splat	campus_img2_peg
Tile Size X	1025
Tile Size Y	1025
Tile Offset X	0
Tile Offset Y	0

Figure 8: Manual Texture Addition

When the user selects the texture, also known as a *splat map*, they also input the tile size of the texture. If the tile size is less than the size of the terrain that it is being applied to, *Unity* will copy the texture to fill the terrain's surface. After inputting this data, the texture is applied to the terrain automatically.

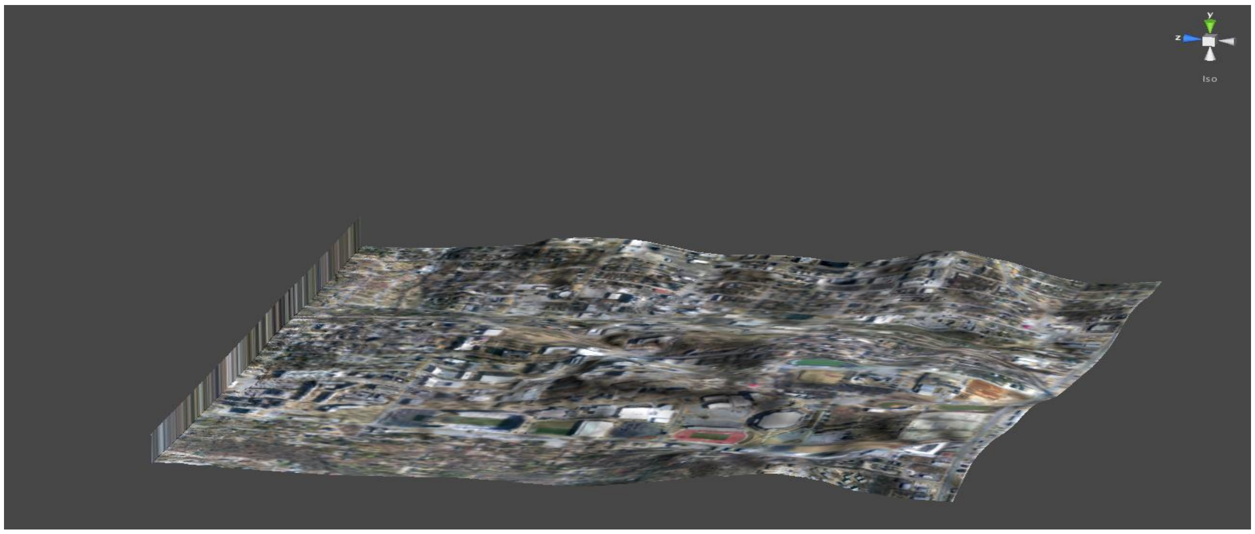


Figure 9: Terrain After Adding Texture

The eagle's eye view originally provided by the scene view does not provide an accurate depiction of just how bumpy the terrain still is. So the next step for the user is to create a directional light and a character controller. After doing this, they will be able to click the `play` button for the scene, enter into play mode, and be able to move around in the world that they have created. However, it is not a "pretty" place because the landscape is stepped, nor is this stepped landscape easy to traverse.

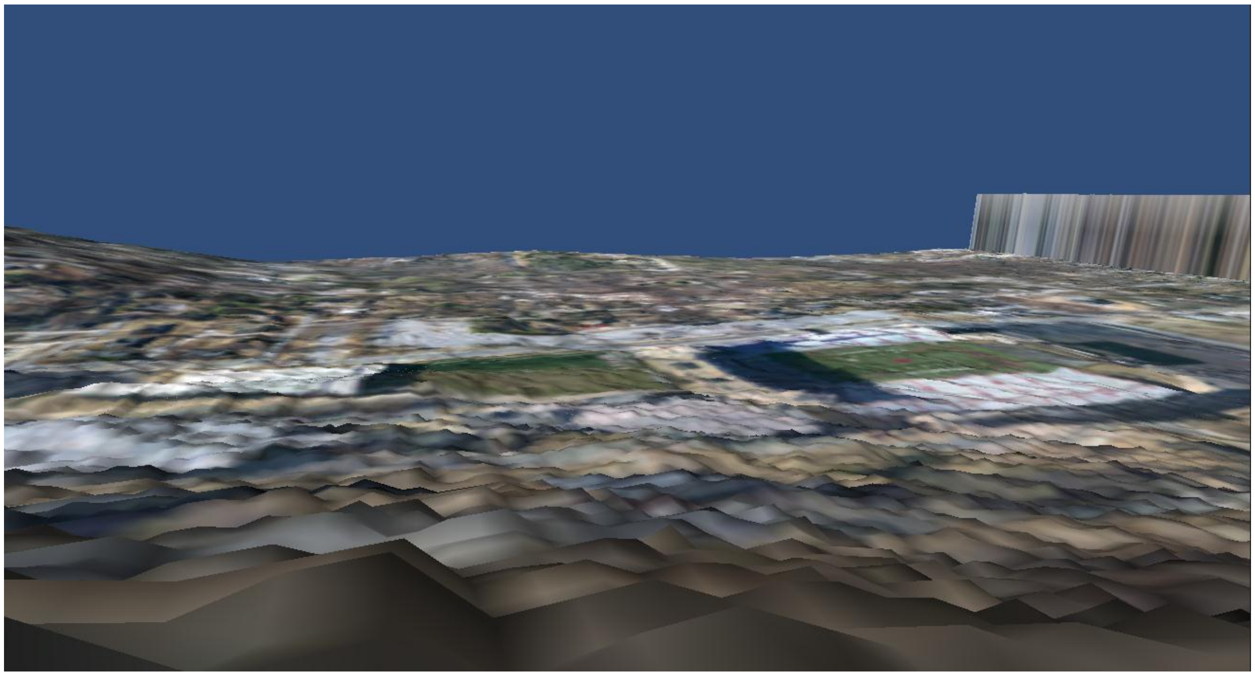
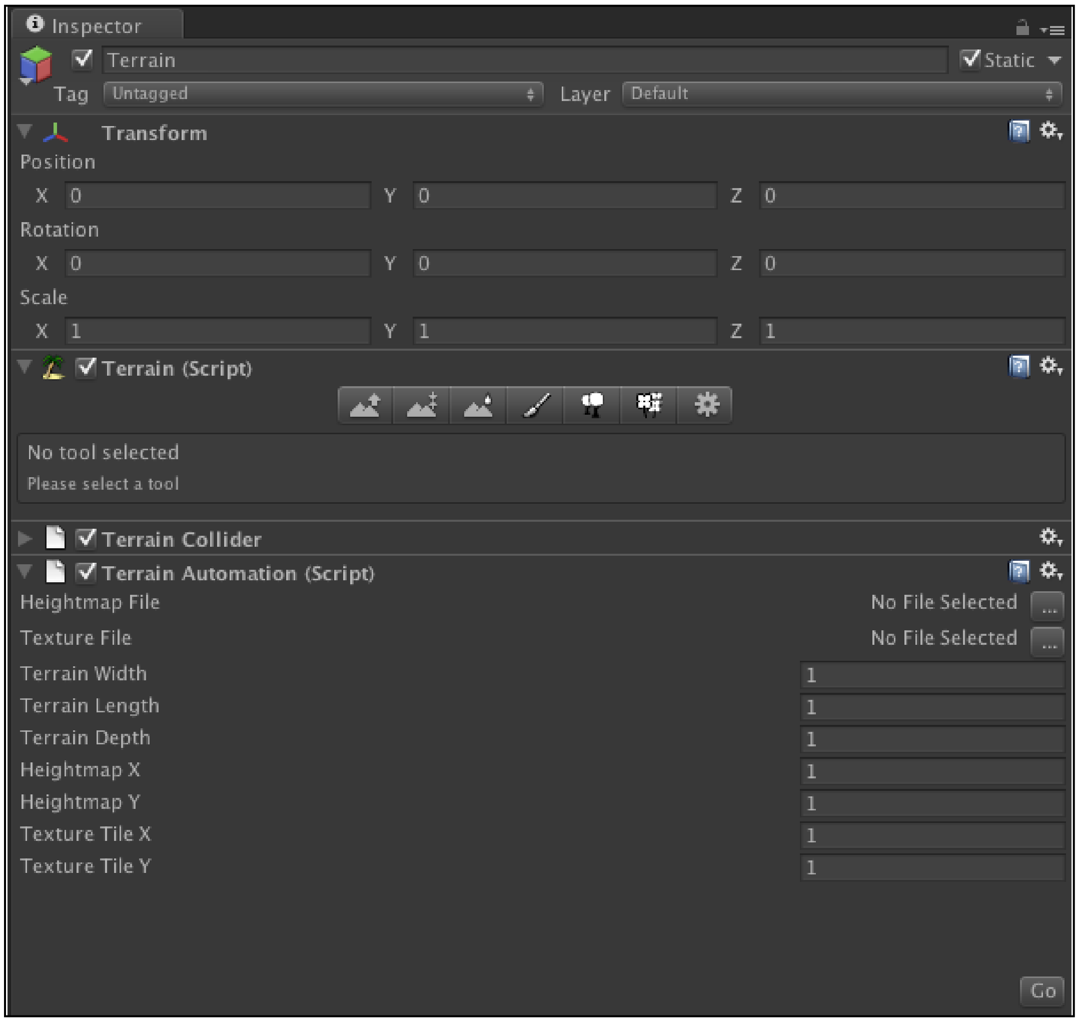


Figure 10: Terrain After Manual Texture Addition

The user can, after exiting play mode, attach the *Terrain Toolkit* to the terrain object in the scene view and run the smoothing algorithm provided therein to create a smoother landscape.

This is the process that we wished to automate with our *Terrain Automation* script. We wanted to give the user one location, easily accessible through a terrain object's `Inspector`, to input all of the data necessary to create the terrain of their choosing. To recap, for this action, a user would need the ability to alter the terrain object's dimensions, choose the height map they wanted to use for the terrain, input the height map's dimensions, choose the texture they wanted to "paint" the terrain with, and choose the tile size of that texture.

We used examples given by the *Terrain Toolkit* to create our GUI, which consists of several labels and text fields where users could input the data pertinent to the terrain. This script originally contained three buttons: two were used to open *Unity*-supported file dialog boxes (and the button labels were appended with "…" to indicate the need for a user to select files), and one was used to alter the terrain data of the terrain to which the script was attached and apply the changes of the terrain data in the scene view. The result is pictured in Figure 11below.



Terrain ✓ Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

✓ Terrain (Script)
No tool selected
Please select a tool

✓ Terrain Collider

✓ Terrain Automation (Script)

Heightmap File	No File Selected ...
Texture File	No File Selected ...
Terrain Width	1
Terrain Length	1
Terrain Depth	1
Heightmap X	1
Heightmap Y	1
Texture Tile X	1
Texture Tile Y	1

Go

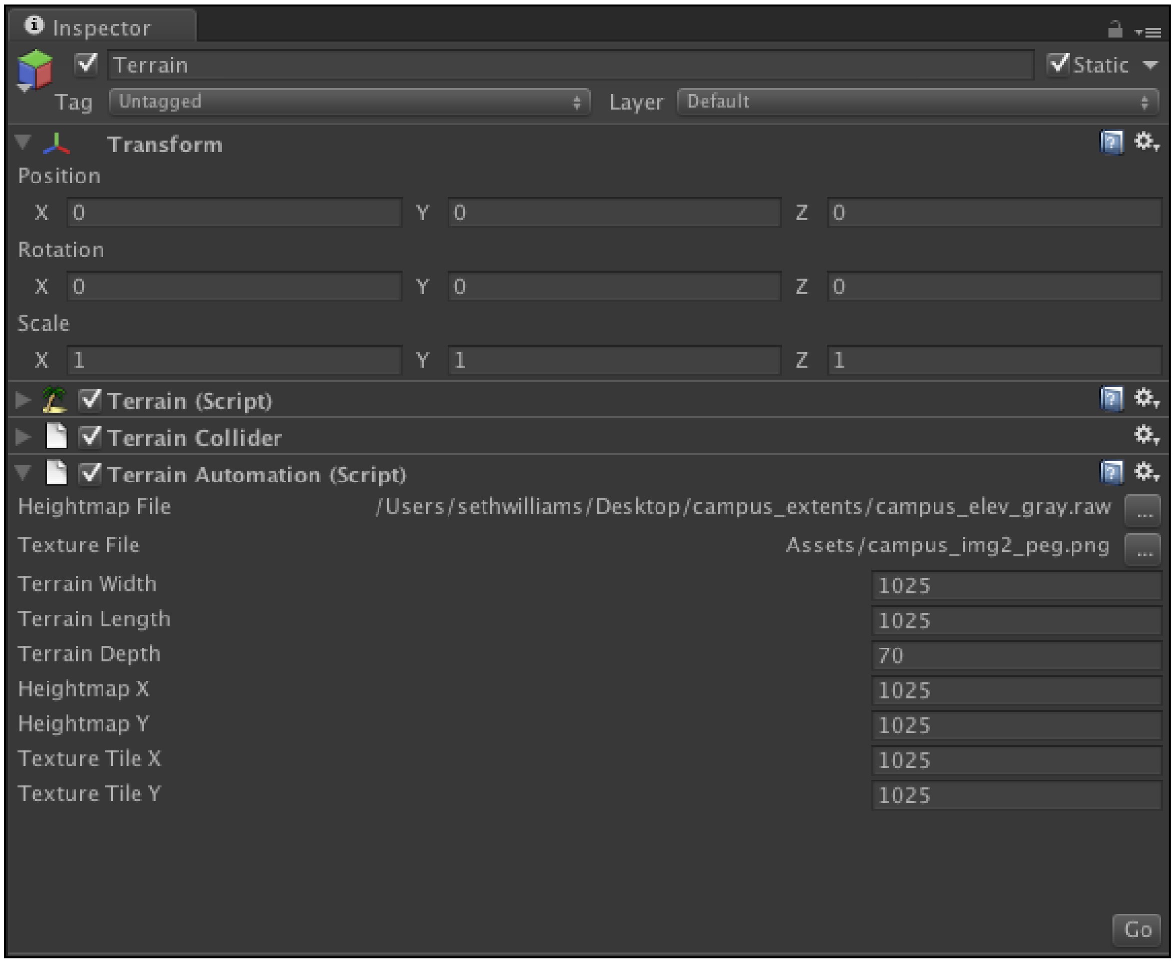
Figure 11: Terrain Inspector with Terrain Automation Script

The button used for selecting the height map brings up a file dialog box which restricts users to selecting only `.raw` formatted files, just like *Unity*'s built-in `Import Heightmap – RAW` menu option. The button used for selecting the texture allows a user to select any file type they desire, simply because of the sheer number of file types that *Unity* recognizes for images. If the user selects a non-image file type, humorously, the texture of the terrain becomes a white background with a large red question mark in the middle of the blank field. Whenever a user chooses a file, the label to the left of the corresponding button changes to show the user the complete file path of the file they have chosen. This serves as a warning to the user, in case they have selected the wrong file and wish to change it before running the program.

Each of the text fields, which allow users to input all of the terrain, height map, and texture dimensions, have a default value of 1, and require users to input integer values in order to continue. If they do not, they receive an error message to input an integer value and the text field is reset to 1.

When the user hits the `Go` button, the program changes the three-dimensional vector for the terrain's size to the three dimensional vector containing the user-input width, length, and height of the terrain, accessible from the use of the writable value `TerrainData.Size`. It also accesses the `Terrain.heightmapWidth` and `Terrain.heightmapHeight` and sets those values equal to the corresponding user inputs. The script contains a 1-entry array for the terrain data's `splatPrototypes` array. We discovered that we were required to restrict the array to one entry (else the program throws a null reference exception and fails to update the terrain). In C#, arrays must be created with fixed lengths. But, for the purpose of this project, we never wanted the user to be able to leave a portion of that array empty due to the aforementioned null reference issue. So, for our script, the user *must* select a texture in order for the rest of the script to run. The inputs they make for `Texture Tile X` and `Texture Tile Y` are then applied to the `TerrainData.splatPrototypes[0].tileSize` two-dimensional vector.

The user is further restricted when selecting their texture. In order to use a given asset if the project is run as standalone, console, or web player build, it is required that the texture file that the user chooses reside in the `Assets` folder of the project that the user is working in. We discovered that this was necessary because we were originally using a different kind of file transfer class supported by *Unity*, called WWW. This class allows the user to select any file from either their own computer (with the phrase "`file://`" prepended to the file path given by the user) or over the Internet (with the full Internet path beginning with "`http://`" given by the user) to use for the texture of the terrain. Under this protocol, the terrain receives the correct texture while in the scene view. However, whenever the user launches into play mode, the *Unity* system assumes that the texture is a part of the project that is being launched. Because it is not, the user is left with the terrain, height map intact and smoothed as expected, but with no texture. So, we instead used the function call, `AssetDatabase.ImportAsset`, with the texture stored inside the `Assets` folder of the project, to ensure that when the user entered play mode, the texture would be there. The file path that is passed to this function call is the same file path that is displayed to the left of the button which opens the file dialog box to select the texture itself.



Position: X 0, Y 0, Z 0
Rotation: X 0, Y 0, Z 0
Scale: X 1, Y 1, Z 1

Heightmap File	/Users/sethwilliams/Desktop/campus_extents/campus_elev_gray.raw
Texture File	Assets/campus_img2_peg.png
Terrain Width	1025
Terrain Length	1025
Terrain Depth	70
Heightmap X	1025
Heightmap Y	1025
Texture Tile X	1025
Texture Tile Y	1025

Figure 12: Input for Terrain Automation Demonstration

As mentioned previously, the height map we chose to use for this project was 1025x1025, and the maximum height of our terrain was estimated to be 70 meters. So, after setting all of the values accordingly as show in Figure 12 above, the user achieves the result pictured in Figure 13 below.

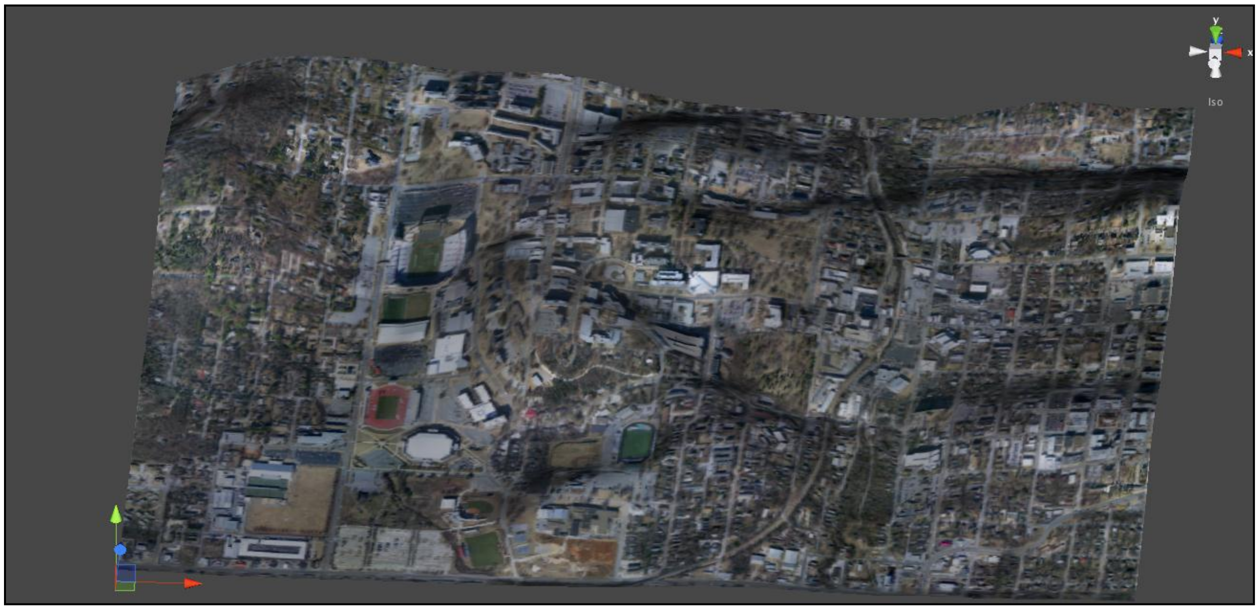


Figure 13: Terrain After Using Terrain Automation Script

The terrain generated by this function does not have a light source or a first person controller yet. These are elements of the world that the user must still create, so the process to create a playable world is not yet completely automated, but the steps for adding these elements to the world is simple. To begin, the user opens *Unity*'s `GameObject` tab, and selects `Create Other`. This brings up a variety of options for the user to select, but the light options are the ones that are pertinent to this step.

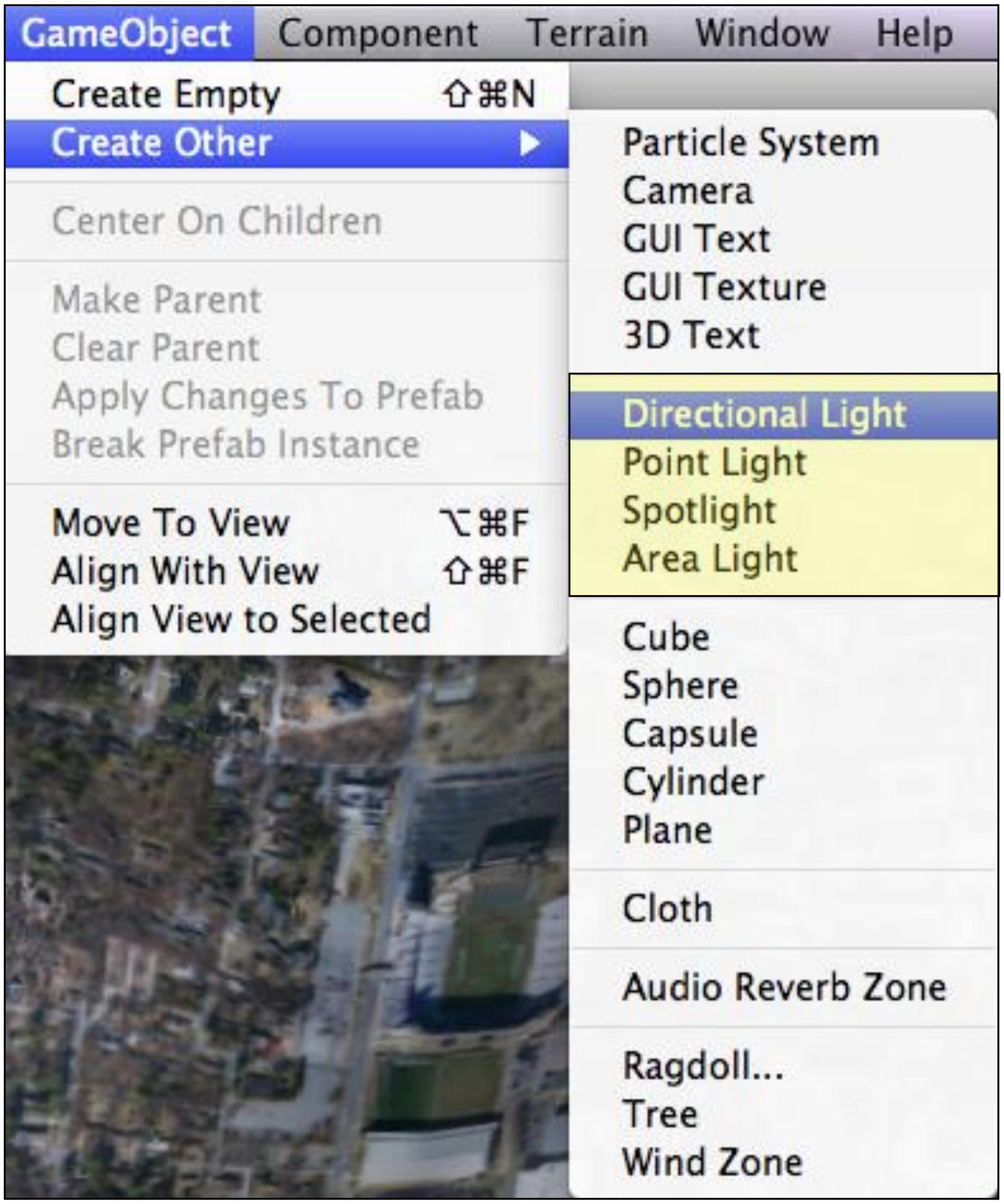


Figure 14: Adding a Directional Light

`Directional Light` is the most commonly used light for lighting terrains, as it emulates the effects of the sun on a terrain's light and shadow nicely. After selecting, the user can rotate it to whatever angle they please, so as to more accurately reflect a specific time of day.

For example, keeping the light pointed at a perpendicular angle to the terrain would simulate noon.

After creating a light source, the user can add a character controller that will enable them to move around the 3-D environment. To do this, the user can open the `Assets` folder of their *Unity* Project, if they imported the `Unity Standard Assets Package` when creating the project. Inside the `Assets` folder, there is another folder labeled `Character Controllers`. From here, the user can select a first- or third-person controller. Since we didn't create an avatar, we chose to use the `first person controller` for our scene. Now, when we enter play mode, we can move around the scene freely and see the difference between the terrain that was not smoothed and the terrain we created using the *Terrain Automation* script we developed, which is automatically smoothed.

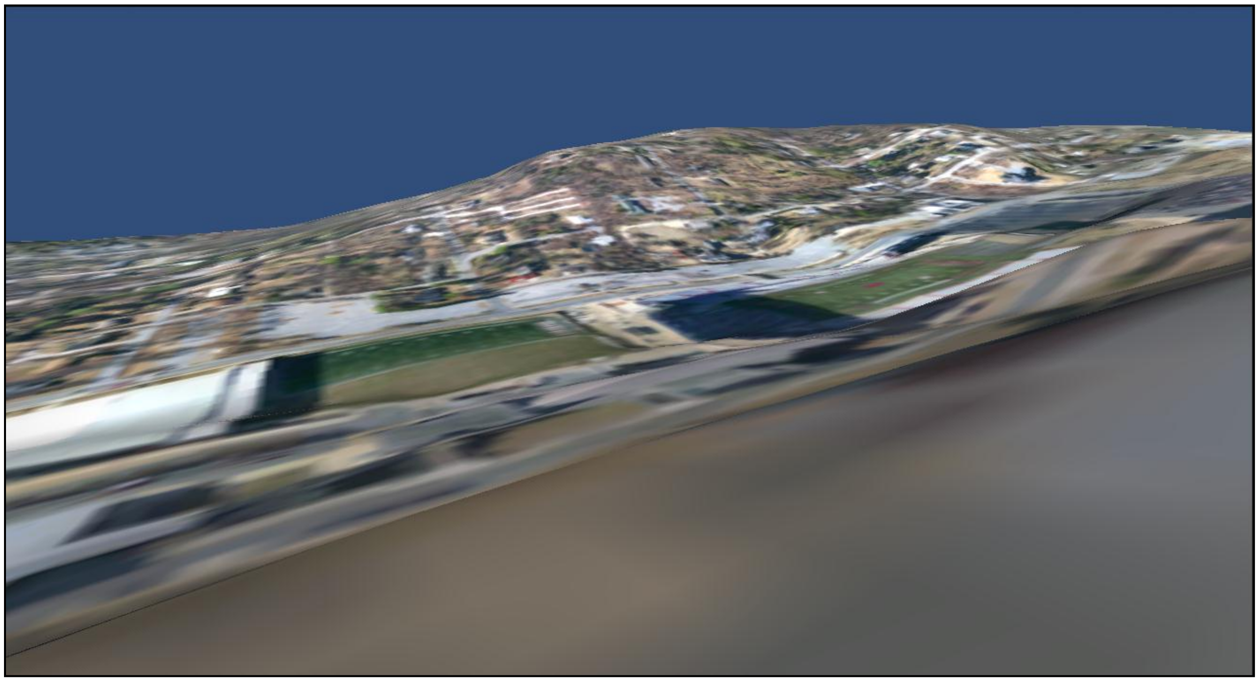


Figure 15: First Person View of Automated Terrain

The difference is extraordinary. The terrain is no longer “stepped” and the only troublesome locations for a user to traverse are where there are any extremely steep slopes. This, still does not perfectly reflect the real world terrain, but is more realistic and gives the user the ability to move around. From our experience, we believe the user will not notice minor discrepancies between the real world’s terrain and the virtual terrain.

3.2 Adding Buildings

With the landmass created, the next step was to enable the user to add 3D building models.

The already textured surface provided information on land use from an aerial view but, like most satellite data, was not high resolution enough to be believable for an avatar walking over the region.   Farther portions of the terrain look very good, but the portions of the terrain which were close to the first person controller are blurry, indistinct, and distorted.

Two approaches to adding buildings were tried: an automatic approach based on recognizing buildings in texture maps did not succeed so a manual approach to placing buildings was added instead. Both approaches are described.

In the first approach, a stylized map of the University of Arkansas was selected (see Figure 19) because it was far simpler than the earlier land use textures and clearly showed buildings, pathways, and green areas of campus.

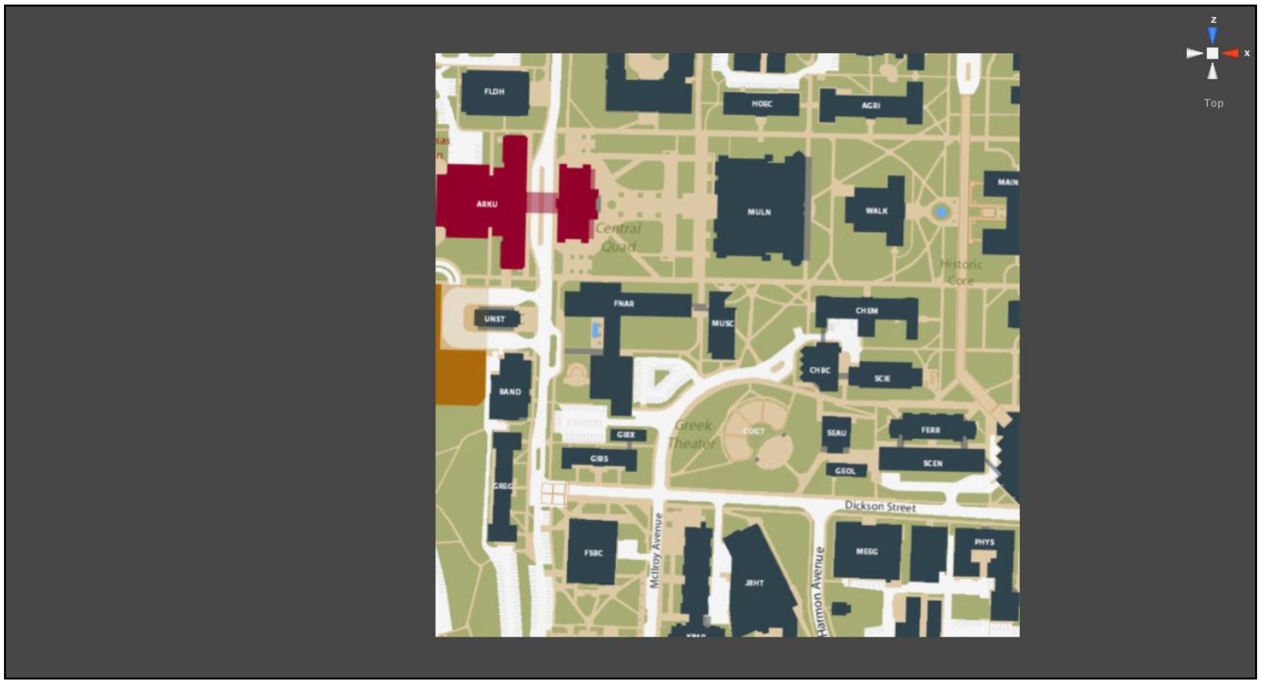
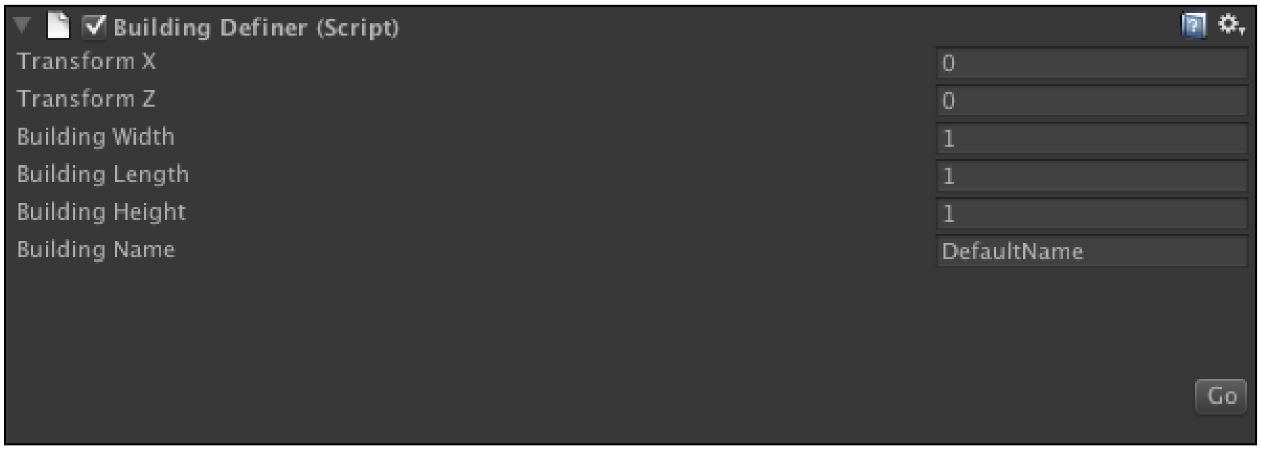


Figure 16: Stylized University of Arkansas Map

The intention was for a user to provide a similarly simple color-coded map to the system, which could then be presented to the users in RGBA format. The user would tell the system what RGBA values would correspond with given textures (e.g., green pixels would represent lanscaping, tan textures might represent concrete pathways). However, even the very basic campus map selected for this project contained well over 6,000 different RGBA values, despite the fact that the map appeared to have under ten colors. Most pixels at the edge of regions, when viewed up close, were mixtures of two or more different colors. 6,000 color values, while far too large for this project's purpose, is really only a small percentage of the 16,777,216 possible RGBA values. Still, although I could have manually smoothed the colors for the particular map selected, this was not a workable approach for a simple script that is meant to expedite the arduous process of building creation. An image smoothing process would have been needed to simplify the map, followed by a region detection process.

So, instead, a second method was provided to add buildings into the 3D regions. Instead of being automated, this method was manual and involved deloping a `Building Definer` script. The code for the GUI was copied from the `Terrain Automation` script.


Building Definer (Script)

Transform X	0
Transform Z	0
Building Width	1
Building Length	1
Building Height	1
Building Name	DefaultName

Figure 17: Default Building Definer Values

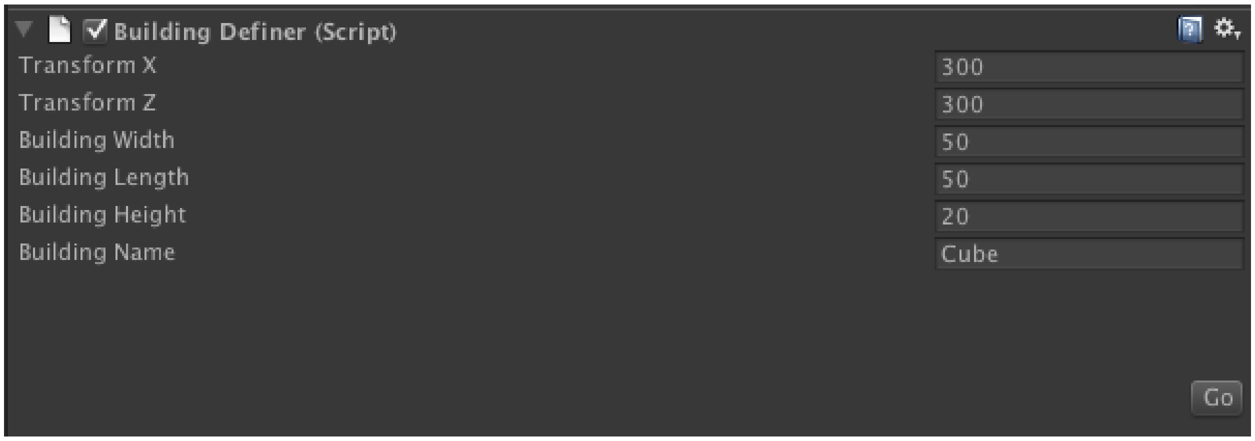
In this new script, there is no button to open a file dialog box. The user will be responsible for making sure that they have a default object in the `Resources` folder inside of the `Assets` folder of the *Unity* project. Any actual building models that they have will also go into this folder. The code then calls the `Resources.Load` function, looking for a file in this folder with the user-input `Building Name`. If the values present at default are unchanged when a user presses the `Go` button, then a cube of size 1x1x1 (in *Unity* world units) is placed in the world. The user is prompted for two values to determine where the building model or cube will go. The `Transform X` and `Transform Z` values determine the pixel on the terrain where the center of mass of the object will reside. Then, the code finds the y-value, or height of the terrain, at that point, and raises the object to that point.

The user is also asked for the building's width, length, and height. However, this is somewhat of a misnomer. Due to a limitation in *Unity*, it is not possible to physically change the size of an object in *Unity* without creating a complicated structure of meshes of vertices, and moving those vertices. Instead, what the user is really inputting is the scale by which the chosen object will be resized in each dimension. This is explained in the documentation of the code, and will also be included in the `README` for the program.

An original aim for building model placement was to use CAST-generated building models of UA campus buildings, which were created by utilizing LIDAR laser scans, which would have included both the exteriors and interiors of the buildings. However, there were some problems with using these models. First, the models were not in a file type which could be used by *Unity*. They were in the *Initial Graphics Exchange Specification* (IGES) file format. This file type did not export as `.obj` or `.fbx` files, which are easily usable by *Unity*. Second, they were very high resolution models, each containing hundreds of thousands of polygons, which *Unity* could not handle in play mode. Finally, these building models would not have included any texturing, so they would have been the same plain gray color that a new terrain is when it is created in the system, though they would have been the correct shape.

Due to this lack of useful models, another idea was to create buildings as "sugar cubes." The user is responsible for either creating textures for the "sugar cubes" and forsaking the interiors of buildings, or will have to build their own models in some modeling software like *Cinema4D, Blender, Maya, Vue*, or *Sketchup*. This process was beyond the automation capabilities implicit in *Unity*.

The user is also required to estimate the x-y location at which they want to place the center of their building, and the approximate scaling factors for their building. I experimented with this process with acceptable results. For instance, to place a cube at the spot where Bud Walton Arena is on our texturemap, I created a 1x1x1 cube, and placed it inside my `Resources` folder, naming it `Cube`. I input the following data to specify the X-Y location:



Building Definer (Script)

Transform X	300
Transform Z	300
Building Width	50
Building Length	50
Building Height	20
Building Name	Cube

Figure 18: Input Data for Building Definer Demonstration

When I clicked the `Go` button, the script found the file named `Cube` in my `Resources` folder and placed the cube into the virtual world as shown in Figure 18 below. Bud Walton Arena, like most buildings, is not perfectly square, so there some parts of the aerial view stick out of the sides but the sugar cube provides dimensionality and a first approximation to real buildings.

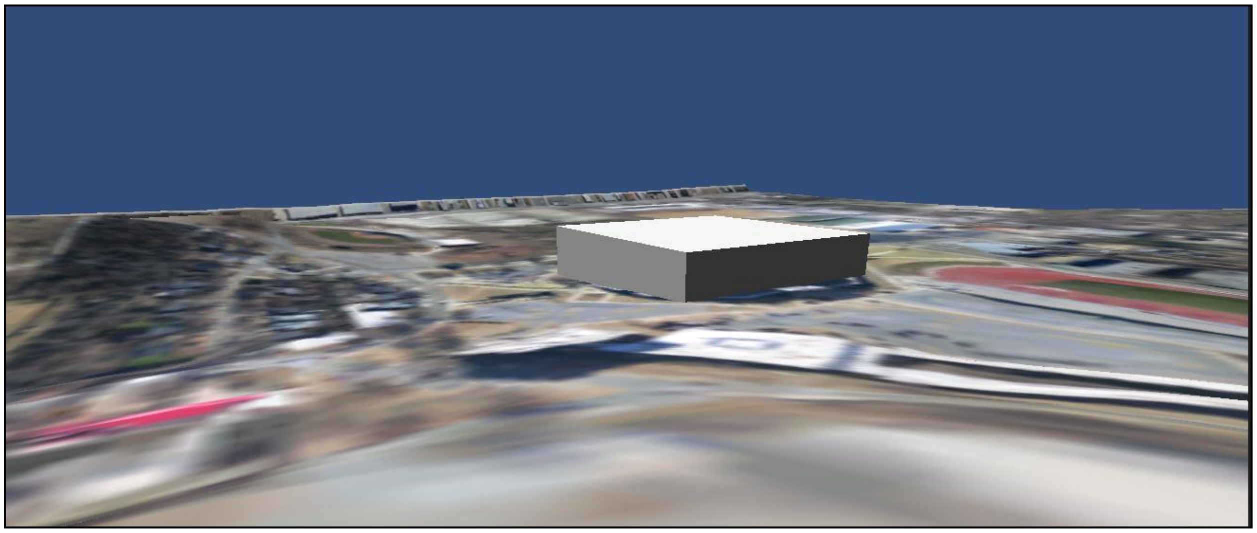


Figure 19: First Person View of "Bud Walton Arena" Cube

To demonstrate the process works, several sugar cube buildings were added to one area of campus. This took 20 minutes to add 10 buildings, indicating that it is not hard to add buildings. The user will have to rotate their building models or "sugar cubes" if they are not oriented properly, but this is easily done. Initially, the user may have difficulty determining the location and scale of the buildings as defined by the *Building Editor*, but this task becomes easier with practice. The result of adding the sugar cube buildings is shown in Figure 20 below.

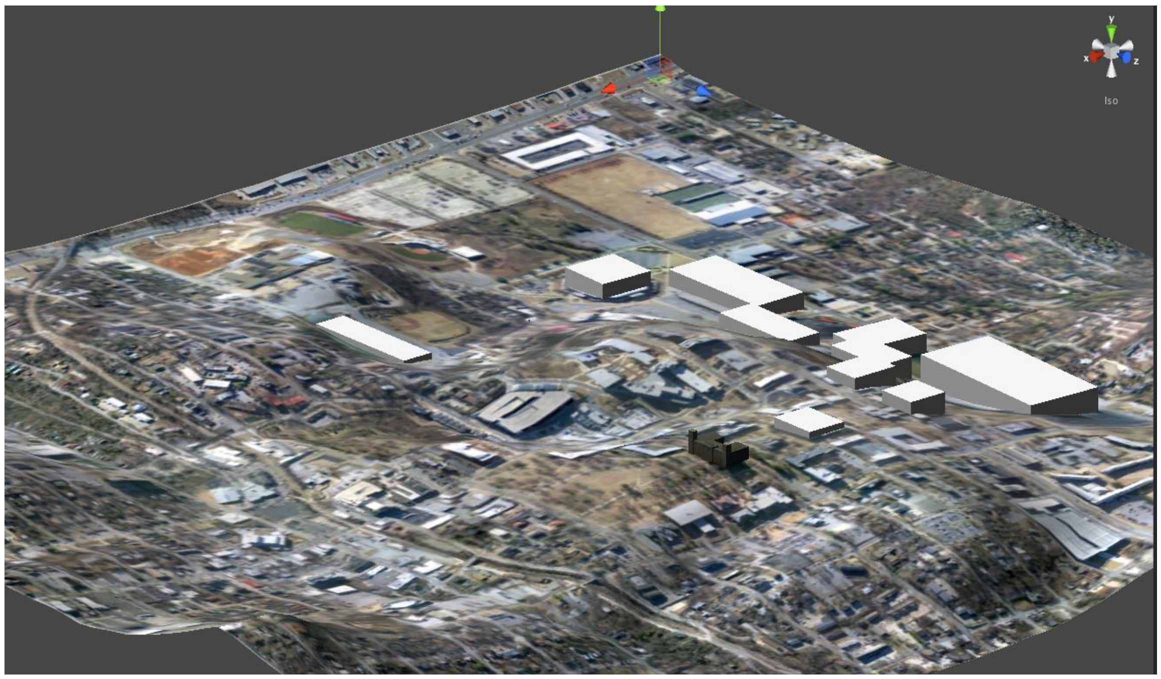


Figure 20: View of UA Campus after Adding Several "Sugar Cube" Buildings and One Custom Building (Old Main)

The result of adding Old Main is shown in Figure 21 below. This building model was taken from Google Warehouse and is available to anyone who wishes to use it. It is important to note that the textures for the model must be imported into *Unity* before the model is imported. They need to be in a folder named `Textures`, which should be located in the same folder as the model. If these two conditions are not met, then the model, when it is brought into the scene, will be bright pink.

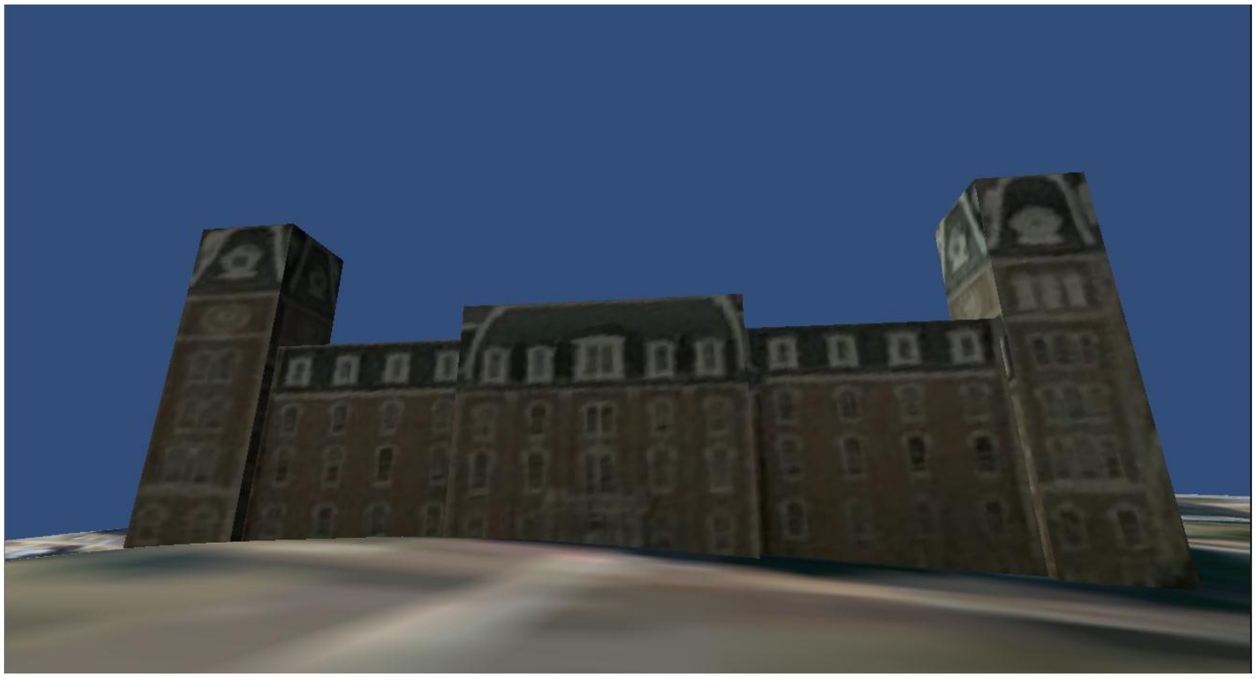


Figure 21: First Person View of Old Main

While the result is not aesthetically exciting and the current effort means the user still will need to put in considerable work to build realistic buildings, the current approach provides a reasonably realistic elevation model, textures that define land use and that can be used as a guide for placing buildings, roads, paths, and landscaping, and a first approximation of 3D buildings which can be refined manually by a user.

4. CONCLUSIONS

4.1 Summary

This thesis builds on my senior capstone project. In the capstone project, the problem faced involved automating the process of using real world elevation data of an area (in this case, the University of Arkansas campus) and transforming it to a format for importing into the *Unity* game engine so that it could terraform a region to automate the building of land areas in a 3D virtual world. Also, aerial data from Google maps was overlaid to “texture” or paint land use onto the 3D terrain. In this BS thesis, an additional capability was added to place “sugar cube” buildings into the map to provide a basic skyline. It would still be a manual process for virtual world designers to either texture the sugar cube buildings or replace the sugar cubes with more realistic 3D buildings (as demonstrated). Thus, our capstone project coupled with this project together automates some of the steps that are needed to automate the building of 3D virtual world regions that represents real world places.

4.2 Future Work

A long term goal of the research would be to scale virtual worlds to represent an augmented reality Earth and automate the process of building and maintaining the model always up to date. The project reported herein represents some first steps but there are other steps that will be needed.

- Currently, the Google map texture overlay or more stylized graphic campus maps provide information a human can use to judge land use – roads, paths, buildings, and vegetation. A human can use this information to manually place objects at these locations. A better solution would be to use image understanding to identify some or all of these land use features; then place objects at those locations. This would require figuring out the puzzle of dynamically texturing color-coded maps. This might involve identifying regions of nearly the same color to groups point together, then create meshes of the regions to build objects.

- In these color-coded maps, buildings are often clearly defined. Currently the elevation map we received from CAST does not yield flat areas where buildings should be, either at ground level or building height level. Other height maps may account for building locations. However, for the purposes of the examples provided in this project, it would be useful if an algorithm were developed to flatten areas where user-defined building definitions were to reside. This would add another dimension of realism to the world, by diminishing the effect of the generated terrain on the buildings in the world.
- One way to create more realistic buildings semi-authomatically would be to develop a script in *Maya*, one of the few modeling applications that have extended support for scripting, that could take images of building floor plans or blueprints, along with other pertinent information like heights of individual floors, and create 3D building models, to be exported in either `.obj` or `.fbx` formats. These building models would then be placed into the Resources folder of a *Unity* project by the user, who could then place the buildings into their scenes. An alternative would be to use LIDAR data and a LIDAR2model mapping (currently non-existant) to create buildings. The UA Facilities organization has LIDAR data for many of the buildings on campus but no way to map the data to a 3D model.
- After providing the world with buildings, it would also be useful to populate the areas covered by vegetation including trees. Unfortunately, tree locations are not typically kept

in color-coded maps, or in any other sets of information about given locations. Automating this step with models and locations of actual trees might prove challenging for years to come but it seems possible in the near term to add a function to the `Terrain Automation` script, along with corresponding text fields, that would enable a user to define a irregularly shaped region (as opposed to restricting them to circles or odd shapes given by the vegetation brush tools provided by the *Unity* system), and input the density of randomly placed trees in the region. It would be necessary for users to place individual trees for the creation of landscaped areas. But, the placement of trees for more rural or wooded areas may be significantly less important, and could be automated to aid in the creation of 3D virtual worlds covering these regions.

- It would also be interesting to add avatars representing their humans into 3D virtual worlds representing real places automatically. Since many humans carry smart phones and their cell phones can monitor their location, these avatars could be placed approximately onto a terrain and their movements could be monitored and updated.
- Finally, it would be interesting if man-made objects could be automatically placed into scenes. This could eventually be done as follows: first assume that future manufacturers provided 3D models using a standard CAD format and that these models were available in repositories like Google 3D warehouse. Then assume that all future retail objects had RFID tags and smart phones were extended with tag readers. Then, as a person walks around (and their locations are already tracked), their smart phone (augmented with an RFID reader) registers the approximate location of tagged objects that could then be automatically placed near or at their real world locations. Crowdsourcing this capability

with many humans could result in models that remained fairly up to date even when object locations change.

A second direction still needed for building 3D virtual world terrain data from map data is to build a web service that attaches to *Unity* to enable a user to select map data from a mapping archive. This would make it very easy for any *Unity* user to create realisticly contoured virtual spaces corresponding to real world spaces on Earth (or Mars).

A third direction (being pursued in a separate project by Andrew Tackett) would be to enable an avatar to walk in any direction and to demand load regions as they approached those regions. This would virtualize virtual worlds. This cabability is not available yet in 3D virtual worlds even though it would be beneficial – at any given time, no avatars are visiting many *Second Life*, *Unity*, or *Open Sim* regions but the regions are still "hydrated" and accessible on their servers, leading to server farms with thousands of servers even though many of their regions are idle.

A final direction for future research is the creation of games and activities using virtual world platforms. Dr. Fredrick in the UA Classics Department is beginning development of a game set in the Greek Theatre on the UA campus. The team involved will be modeling the monument, scripting the game functionality, and providing textures for "sugar cubes" to add an element of realism to each level of the game. Additionally, it will use the scripts that have been created through my project to create and texture the terrain, and to provide functionality to place the surrounding buildings in their proper locations. This game is an exciting application of my project, and it is my hope that it can continue to be used by the *Unity* community as a model for creating additional games and real-world simulations of industrial purposes of land usage, like building planning.

4.3 Potential Impact

The ultimate goal of this ongoing project is to allow users to interact with the real world via an augmented reality medium. I believe that we, as an educational community, are closer to that intended goal as a result of this project. I also believe this project points the way for how the GIS mapping community data can be useful in automating the creation of 3D virtual worlds that can then be used for gaming including serious games, and that we will see this fusion of communities in the near future.

REFERENCES

[1] T. Munroe, “SSX Developer Blog -- Owning the Planet,” posted: June 16, 2011 [Online]. Available: http://www.ea.com/ssx/blog/ssx-developer-blog-owning-the-planet.

[2] “Real Satellite Image to High Definition Terrain in Unity3d,” Unity Forums, posted: May 19, 2010 [Online]. Available: http://forum.Unity3d.com/threads/40870-Real-satelite-image-to-High-defenition-terrain-in-Unity3d?p=262122#post262122.

[3] “How Virtual Yellowstone National Park in ScienceSim Was Made,” Fashion Research Institute, posted: January 16, 2010. [Online]. Available: http://shenlei.com/2010/01/26/how-virtual-yellowstone-national-park-in-sciencesim-was-made.

[4] “A Glimpse of the Future of OpenSim: Virtual Yellowstone National Park in ScienceSim,” Fashion Research Institute, posted: January 24, 2010 [Online]. Available: http://shenlei.com/2010/01/24/a-glimpse-of-the-future-virtual-yellowstone-national-park-in-sciencesim.

[5] Terrain Toolkit, Unity 3D, 2012 [Online]. Available: http://Unity3d.com/support/resources/Unity-extensions/terrain-toolkit.html.

[6] D. Deeds, “Jibe/Unity School Quick Start Guide -- First Edition,” Scribd.com, February 2012 [Online]. Available: http://www.scribd.com/doc/81798024/Jibe-Unity-School-Quick-Start-Guide.