Large Scale 3D Modeling in Real Time

Douglas Binder, Tommy Markley, and Margaret Wanek
Department of Math, Statistics, and Computer Science
St. Olaf College
1500 St. Olaf Avenue, Northfield MN 55057
binderd@stolaf.edu, markley@stolaf.edu, wanek@stolaf.edu

Abstract

This project involves creating a dynamic cross-platform framework to render an explorable, three dimensional model from massive amounts of image data. The dataset consisted of over 7000 2D images all translated in to 3D coordinates. Our specific research aimed to build this 3D data into a visualization of the entire interior of a building. Most of the framework is written in WebGL, a robust technology that allows hardware-accelerated 3D graphics in the browser. Although WebGL is still experimental technology, it allows us to read in the data as sets of vertices and texture coordinates, and render them rapidly. Because of the limitations of mobile devices and the size of the data set, we had to ensure that our use of memory was smart. This meant optimizing our object creation and data transfer by researching limitations with regard to render time and memory capacity on both tablets and computers.

1 Introduction

This document details the work done by our team during the January 2014 continuation of the St. Olaf College Computer Science department's ongoing work in 3D vision and modeling. We were designated the development of the web interface for the three dimensional model of the interior of Regents Hall. We started with 2D images, that were converted into tiles, or a three dimensional planar object that has texture mappings for multiple images. These tiles were represented by coordinates in 3D space, and texture coordinates for mapping part of the image onto the resulting polygon. This data was stored in custom file format, each file would include all vertex data for a polygon as well as the name of the image it was included in. To render this information, we used WebGL: a browser equivalent of OpenGL, a hardware based graphics language. WebGL is very experimental, and is sometimes not supported in browser. Because of its novelty, it has much fewer resources and documentation, and is missing some of the more complex additions to OpenGL.

2 Approach

2.1 Initial Hallway

2.1.1 Movement

Our project was part of a pipeline that started in photography and ended up with real time 3D modeling of the building's interior. Because we were the last stage in this pipeline, we had to start without real data, and began to write the interface that we wanted to use with a fake hallway. The hallway consisted of walls and a floor, a cube on one end and a pyramid on the other. Such an environment helped us create the controls, movement and interaction we wanted to provide the user. We implemented rotation and movement in 3D space by manipulating the X, Y, and Z coordinates of the camera position as well as the yaw and pitch, or the angles of rotation to the side and top. We redraw the scene with these parameters according to the browser's animation frame rate.

2.1.2 Collision Detection

To prevent the user from walking through walls and moving off the model, we implemented a form of collision detection. Thinking strategically, it wouldn't be practical to simply check every object's location and see if the camera's location was too close to it. While this would work for our hallway, an entire building model would contain thousands of tiles and checking each would be an impractical use of processing power. Because of this limitation, we decided to read in the coordinates of the current floor tile and test that the camera remained between these points.

2.1.3 Mini-Map Implementation

We wanted to create a map in the corner that showed the user's location on the floor plan. The creation of the mini-map was pretty easy with this hallway, rectangular as it was. A proportional SVG in the corner represented the floor plan, with a red dot showing the user's position. Once we figured out the conversion between SVG coordinates and WebGL coordinates, it was just a matter of moving the camera icon on the map according to the camera coordinates. Because collision detection was already handled, the red dot couldn't leave the SVG either. Dragging the icon along the mini-map also moves the camera, for quick movement from one side of the hallway to the other.

2.2 Real Data

Once we had our interface ready, we began to work with real data. Reading in a test file resulted in a mis-drawn tile, split up into several triangles all across the screen. In order to get the tile to render correctly, we had to tessellate it. Because of the lack of tessellation available in WebGL, the files were pre-processed, tessellated by another team using the OpenGLU tessellator. This resulted in fully formed tiles and more plausible tile data.

2.2.1 Dynamic Buffer Creation

WebGL creates objects from a vertex array. This array is read into a WebGL buffer, which is then loaded by name and drawn in conjunction with the array and buffer for the texture for the object. We approached this by storing maps of names to their images and textures. WebGL itself keeps track of the actual vector data, so we were able to have WebGL draw each tile by giving it the unique name of each tile and its texture.

2.2.2 Textures

Textures are mapped to objects by U,V coordinates which show the coordinates of the object on the image. The pixels between these points are turned into a texture to paint on the tile. To eliminate duplicate textures, we mapped each tile name to the image name that it needed. [2] Each image name was given a texture. This way, if three objects had the same image, we only needed to create one texture. Since our primary concern was speed and memory consumption, this helped to considerably reduce both.

2.3 Scaling

Once we had a framework for reading in any number of tiles, textures, and images, we thought about scaling. The actual building is huge, and the size of our dataset reflects that.. We approached the large data set with a volumetric query. The query would give the camera's current coordinates and fetch the appropriate tiles within a certain range. As the user moved, we'd be able to swap out any tiles that were no longer relevant and add ones that would soon be visible. We wanted the transitions to appear seamless, so a user could "walk" through the building without noticing. This would mean creating requests whenever the user moves, sending the coordinates to the server. [1] The server could query the database, finding the appropriate number of tiles to render, and sending them to the front end. Ideally, we could change this number as well as the texture resolution based on the internet speed and relevance of a tile. Users on slower internet or devices have fewer tiles constructed in front of them, making the load time much faster. In addition, tiles at the far end of the hallway would be textured with lower resolution images to speed up the data transfer for images. As the user approached these tiles, the texture could be changed to a higher resolution. In this way, we would have a comprehensive 3D map loaded in chunks. Dynamically swapped out, the user could seamlessly explore areas of the building and find their way to rooms or other landmarks.

3 Results

3.1 Interface

Our model allows the user to control their position by moving and looking. On a desktop or laptop computer, position is controlled with the arrow keys and the view is controlled with WASD. Position simply moves the camera forward, backward, or to the left and right without changing the view. View rotates the camera, spinning it in place allowing the user to look around. On the tablet, these movements are controlled with two joysticks similar to a gaming controller. This camera manipulation is performed by changing the XYZ coordinates and the yaw and pitch of the camera. With the exception of a few bugs, our interface runs smoothly. With a lower frame rate, the animation would probably be choppier. As is, the movement doesn't jump much and gives the impression of moving through the space in a smooth, clean way. On the tablet, our joysticks work really well. The user can both press and hold and drag on the SVGs, and it will accordingly move the camera and rotation, but they are unable to be used at the same time. This means that the user would be unable to turn left at the same time as they moved left.

3.2 Applied to Real Data

3.2.1 The Hallway

We completed a hallway that can be fully interacted with. The user can move and look at the fully textured space. The tessellation works correctly, so the polygons shown are close to their actual form. However, some irregularities exist in the shapes and in the model in general. Some tiles are missing and some have gaps between them. These flaws appear to be a produce of the data that we were given, although some could be a product of some minor mis-tessellation. Because the data was being worked on continuously, some tiles were corrupted and unable to be displayed. However, on the whole the hallway looks pretty good. The images below were rendered from one set of tiles. These tiles came from one image.

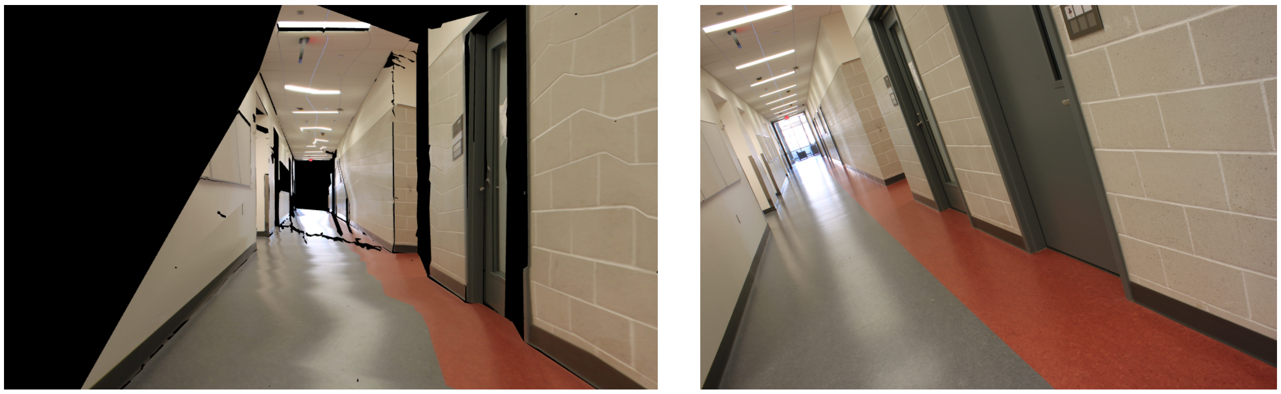

Figure 1: Our 3D Model on the left, and the photo on the right. The photo is skewed, but this is corrected in the model. The black spots on the model are either not covered in the photo, or faulty tiles.

If more data was ready, we would be able to extend the hallway, and complete the walkthrough. The closer the user gets to the other side of the hallway, the more obvious the mistakes are. The data is less accurate because of the lack of feature points and inability to distinguish tiles so far down. This could be corrected by loading a second set of tiles further down the hallway. This image would have been taken closer to the tiles, and consequently produce better models.

3.2.1 Textures

The textures on the tiles are loaded on initialization. A request is sent to the server, which sends back these tiles and the name of the image file associated with them. From here, we create a WebGL texture. The texture is created and mapped to the image name, so that further requests for this image won't recreate a texture already present. The hallway above uses one image for 376 tiles. Some tiles show some warping of the texture, as though they have additional vertices in the middle. We speculate that this is a product of the tessellation, as it breaks up each tile into renderable triangles. This creates vertices in the middle of an object, but it allows WebGL to create the tile without error.

3.2.1 Data Transfer

The size of our data set means that we will not be able to render every possible tile. We tested the limits of our texture capabilities by creating textures for each tile, as opposed to using the same image texture for all of them. This showed that a laptop (Dell XPS 14z) can handle over 300 textures pretty smoothly. The Nexus 7 tablets that we used were unable to handle this many. They were able to handle 94 textures, but this produced a loading time of over 30 seconds. Without sacrificing much loading performance, the tablet was able to load 10 textures. This is significantly different than the ability of the laptop, but would be sufficient. On load, we envision a sort of bubble of images loading around the user. This would load tiles all around them, and on movement these could be swapped out pretty quickly. With an initial load of 10 textures, it would still be imperceptible to replace them one at a time as the user moved. Tile loading could also be performed asynchronously, so the page could constantly load tiles as needed. On a computer, performance is much better and would allow for a big initial load. Given this increased capability, we would use a different initial load circumference based on the type of device used. This obviously varies on device, but a general loading handicap on a mobile device should be assumed.

3.3 Overall Framework

Because of the limitations of data, we weren't able to test our framework with more than a few hundred tiles. However, this was still a sufficiently large data set as it shows how our framework deals with loading in a lot of data, dynamically creating buffers, and rendering it. The tiles are read in by a python script. Currently, we don't have the volumetric query in place so there's no way to find only relevant tiles and images. The data we use currently was picked due to the completeness of its tiles. The script reads in all available files. As more data becomes available, this will need to be modified to query a database for nearby tiles. On the browser side, our code is comprehensive. It reads in all the tiles that come in, allocates new buffers for them, and sets them in a map. The draw function goes through the map and draws each object and its similarly mapped texture data. This code handles any number of tiles and any number of textures. Also implemented is the mini-map. The camera icon moves with the users movements, although this has a few bugs when the user rotates their view. Collision detection is not implemented, as our floor buffer is not very clean, especially due to the tessellation. We anticipate making this a feature on the mini-map, such that if the icon hits a line it has hit a wall and shouldn't move any further. Overall, we implemented an extendable framework that reads in tiles and textures dynamically and can be fully interacted with although it lacks the completeness of the model we sought to make.

4 Future Work

With our current framework in place, we have high expectations for future work. Some of these ideas we hope to complete while still attending St. Olaf College, while the rest we leave open for others to pursue.

4.1 Handling the Large Data Set

Since this interface might eventually need to handle up to 7000 images to display the entire model of the building, there needs to be an efficient way of providing only the most relevant data to the user. This is where a volumetric query (referenced in Section 2.3) would come into place. Given the user's location in the model, we would serve the client the highest resolution textures for the tiles that are closest, and then decrease the resolution incrementally as the tiles get farther and farther away. For our model, we only used 1 MP images in the textures. Since the original images were each 15.1 MP, there is a lot more data at our disposal. Based on the user's internet speed and device capabilities, we could compute the total amount of data that could be used on the client side, and scale the amount of data sent accordingly. Optimally, we would also swap out tiles that become irrelevant (those that the user couldn't see from their position because of occlusions, being on a different floor, or simply being too far away.) and render the rest for the user to see in the highest resolution possible. This would lead to a much more complete and detailed model.

4.2 Collision Detection

Ideally, we would have named tiles for all of the images in the data set. Assuming that we had this information, we could use the coordinates of the floor tile for collision detection and creation of the mini map. To implement this, we'd ensure that the user's X,Y,Z position in the model was unable to cross the borders of the floor buffer. To display the mini map, we would render polygons in the SVG based on the borders of the floor tile. Another desirable feature would be to allow the user to click on the red dot in the mini map and drag it around to very quickly move through the model.

Furthermore, this idea needs to be extended so that the user can walk up and down stairs in Regents. Once the user walks down a certain portion of the stairs, we would need to swap floor buffers and make sure we change the mini map to represent the correct floor. This is certainly a special case, but also one that needs to be considered.

4.3 Interior Navigation

What we hope our project contributes toward is the development of more interactive interior models with the long term goal of being able to get navigation directions and have the ability to query a specific location, room, or landmark inside of any building, anywhere in the world. A lot of work has been done with exterior navigation (such as with Google Maps), but we truly believe that interior navigation is something that users will want (and expect) very soon. We are optimistic that eventually we will be able to tell our interface to give us directions to a specific room inside of the building from our device's GPS location. We also envision a method of displaying what it would look like if you walked through the building to a location so that the user knows exactly what to expect.

5 Conclusions and Summary

Our research created a reusable framework that holds potential for future expansion. It can be built upon to increase the scalability, tile swapping, and tile precision. While we referenced the use of WebGL, a similar project could be implemented in any software, whether tied to the browser or not. We hope our project serves as a proof of concept to the ability of 3D modeling to enhance reality, particularly for mobile devices. Our world is becoming more technical by the day, and as it does, the demand for augmented reality increases. Products like Google Glass, and even Google Maps, illustrate the rapid incorporation of technology into our day to day lives. By allowing users to explore a virtual space, get directions to a room, or read the posters on the wall, this sort of model can expand the resources that we have at our disposal because of its intuitive, visual way of representing information. While any data that shows such accurate detail about the interior of a building could be abused, the potential for technological advancement is great and should not be ignored.

References

[1] Congote, John, Alvaro Segura, Luis Kabongo, Aitor Moreno, Jorge Posada, and Oscar Ruiz. "Interactive Visualization of Volumetric Data with WebGL in Real-time."*Web3D* ('11): 137-46.

[2]Schwartz, C., R. Ruiters, M. Weinmann, and R. Klein. "WebGL-based Streaming and Presentation for Bidirectional Texture Functions." *VAST* (11): 113-20. Print.