A Web Based Tool for Labeling the 3D World

by

Aaron Scott Nech

Supervised by Steve Seitz and Richard Newcombe

A senior thesis submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Engineering

With Departmental Honors

Computer Science & Engineering

University of Washington

November 18, 2015

Presentation of work given on _____

Thesis and presentation approved by _____

Date _____

Abstract

Through a large increase in 3D sensing becoming available to consumers, we have been motivated to tackle the problem of a computer understanding 3D scenes as humans do. In particular, one central sub-problem is identifying which objects are present in a 3D scene. To solve this problem in a general setting, we propose using prediction algorithms learned over a large storage of labeled 3D scenes. To produce such a set of data, we require a novel 3D labeling tool that is easily expandable and freely accessible by researchers around the world.

In our work, we create this tool to provide the ability to add semantic labels to an otherwise static view of the virtual world. For example, after scanning an office room, the entirety of that room will be reconstructed into a 3D model. The model can then be annotated to specify precisely which parts of that 3D geometry correspond to objects such as *coffee cup* or *office chair*. With emerging rendering and computational power in web browsers, this tool targets the web platform utilizing a full 3D viewer to manipulate and label 3D scenes. We create easy to use labeling tool which allows us to outsource this 3D labeling to a broader community.

Table of Contents

1 Introduction and Discussion

Humans are very good at recognizing objects. In one study, Biederman indicates that humans can recognize over 30000 categories of objects [Biederman, 1987]. Historically, recognition has been hard for computers. Even in 2D, recognition tasks have only succeeded in a small number of categories until recent years (for example, digit recognition); computers simply lacked the datasets necessary to sustain accuracy over larger sets of object categories. This has improved greatly as tools and projects such as LabelMe [Russell et al., 2008] and ImageNet have increased the amount of ground-truth labeled datasets available for 2D recognition training. Projects such as ImageNet have curated large labeled datasets allowing researchers to benchmark various algorithms. This acts as a catalyst for computer recognition research. In recent years, convolution based neural network approaches utilizing large quantities of data have had success in various benchmark tests. As a whole, big data approaches have won over previous approaches to solving recognition tasks [Krizhevsky et al., 2012]. These approaches tend to give more robust and accurate models across a wider range of categories and problems.

Similarly, we believe that the 3D recognition problem currently suffers from a lack of labeled data, and that big data approaches can enable solving recognition problems in a very general setting. As more 3D sensing technology becomes available to consumers, we believe there will be a large amount of unlabeled 3D geometry. Annotating this data can help remove limitations on researchers to benchmark and test various 3D computer vision research. The ability for machines to understand the 3D world has very wide applications. Similar to how ImageNet and LabelMe moved 2D recognition forward, we expect a 3D labeling tool to enhance tasks fueled by 3D data. In the broad sense, a 3D annotated database can enable and catalyze research in the fields of robotics, transportation, visualization, accessibility, and communication.

For example, a better understanding of 3D geometries can enable precision navigation systems in robotics which has many rescue and transportation implications. Furthermore, as virtual reality technology progresses, the ability for programs to understand 3D geometry opens up many opportunities to enable powerful experiences for users.

1.1 Related Work

Crowd sourcing [Howe, 2008] allows large scale participation in tasks enabled by the Internet. It has been successful in a wide range of applications. For example, many fields have used Amazon's Mechanical Turk to crowd source user studies [Kittur et al., 2008]. Annotation tasks can also benefit from crowd sourcing. Large scale recruitment of users to annotate raw data has quickly been adopted in recent years for supervised machine learning tasks [Hsueh et al., 2009]. Although a general audience can have trouble with more complex tasks (such as our 3D labeling tool), the principles of crowd sourcing remain very valuable to annotate data. A possible work around is to curate a more specialized group of individuals to annotate geometry. Creating tools that are web based allow easy integration with web based crowd sourcing platforms such as Mechanical Turk.

An example of using web-based tools for data annotation is LabelMe [Russell et al., 2008], a prominent tool in the 2D labeling space. It follows a similar philosophy we adopted for using web technology to enable easy access and portability of the labeling tool. LabelMe does not, however, allow labeling of 3D geometry.

Another popular tool from UC Irvine is Vatic [Vondrick et al., 2011], a web based video annotation tool. This tool also targets a browser environment to allow widespread crowd sourcing of annotations for video media. It is also flexible in the sense that it can be used offline for a curated set of expert annotation participants.

Besides related annotation tools, recent advances in 3D reconstruction techniques such as KinectFusion [Izadi et al., 2011], have given us the ability to produce accurate 3D geometry from RGBD video. Reconstruction techniques are already being applied to produce accurate indoor geometry [Henry et al., 2012]. As RGBD sensors become cheaper and more ubiquitous we expect this trend to continue and produce a large amount of 3D data analogous to the high volume of 2D images produced after RGB cameras became ubiquitous.

1.2 Key Goals In a Labeling Tool

In our labeling tool, there are three major design goals in mind. In particular, the tool must be easy to use by both developers and labelers, supported on a common accessible platform (allowing easy integration with crowd sourcing platforms), and provide useful data to computer vision researchers. Ease of use for a developer and labeler are two very different concepts. For developers, the tool provides a flexible Model-View-Controller (MVC) architecture that can be easily expanded, and implementation generality (making no little server assumptions). Thus the tool can be easily integrated into many projects.

For client side integration, we created an easy to use JavaScript Application Programming Interface (API), which exposes many tool settings, both fine and coarse in granularity. This allow developers to adapt the tool to their particular research needs.

To provide useful data to researchers, the tool both imports and exports the PLY model format which is widely used in academia. The tool applies an efficient scheme for PLY models such that they can store label information on a per-vertex basis with minimum space cost. The label models can then become labeled input for training algorithms and other research applications.

2 Tool Usage By Labelers

In this section we begin by introducing the interface the user is presented with, and then proceed to discuss the process by which a labeler can label a simple lounge. Along the way we will discuss each particular labeling tool, and their individual design considerations from a user experience prospective.

2.1 Labeler Start and Tutorial

When a user lands on a page for the first time, the user will be immediately greeted by the first stage in the tutorial (if one has been set up by the developer). The labeling system will also fetch the unlabeled model. Once the model is loaded it will be displayed in the center of the user's screen and be ready for interaction via the various labeling tools. In our example, the user will be presented with the view of a unlabeled lounge model produced by a KinectFusion based 3D reconstruction algorithm obtained from a source video of a lounge.

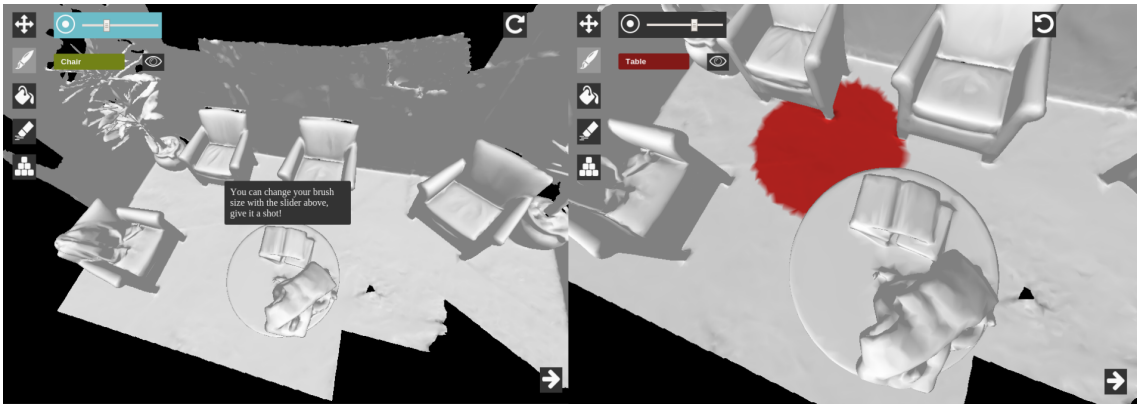

Figure 1: Tutorial system and pressure brush. **Left:** The tutorial system displays a scripted dialog system to the user, and highlights various UI components. **Right:** The pressure brush uses an algorithmic approach to avoid painting up hard angles common in rooms and other artificial geometry.

2.2 Navigation

Once the tool has loaded, and tutorial has been complete (if applicable), the user has a variety of tools at their disposal. The navigation tool is used for moving around the unlabeled lounge. First a user can navigate to a suitable view where they can easily click on all lounge floors and walls by pressing the left mouse button and dragging to rotate the view (using standard quaternion trackball behavior). Once rotated, the user can press the right mouse button and drag to pan the center of the view in three dimensions with respect to their current rotation to further navigate. Once positioned, the user can utilize the scroll wheel to position the zoom level of the camera such that the walls and floors of interest are comfortably in place.

2.3 Using the Smart Fill Tool to Label Walls and Floors

The smart fill tool is used to label flat surfaces similarly to how a 2D "paint bucket" tool fills in bordered 2D image boundaries. The goal of this tool was to provide a way to quickly hide walls and floors in a room. The user can click the smart fill tool, and then select the appropriate label (floor for example), from the label selection picker located in the top left. Once selected, the user can click space on each wall and floor in the lounge geometry to quickly mark the floors and walls with the appropriate label. The user can then hide these layers to focus marking the rest of the geometry. Since the walls and floors are hidden, it becomes much easier to target the vertices belonging to objects in the room.

2.4 Erasing Labels and the Undo Redo System

In the event that a user makes a mistake while labeling, we created a robust undo and redo system to allow users to recover in these situations. An undo button will appear after an action that is undo-able has taken place. The button can be clicked, or alternatively the user can press the control (or command) key and Z in combination to undo. A redo button will appear after a change is undone, this button can be clicked or the user can press the control key and Y again in combination to perform the redo.

In the event that a undo change batch is too large (each entire paint stroke is batched as one

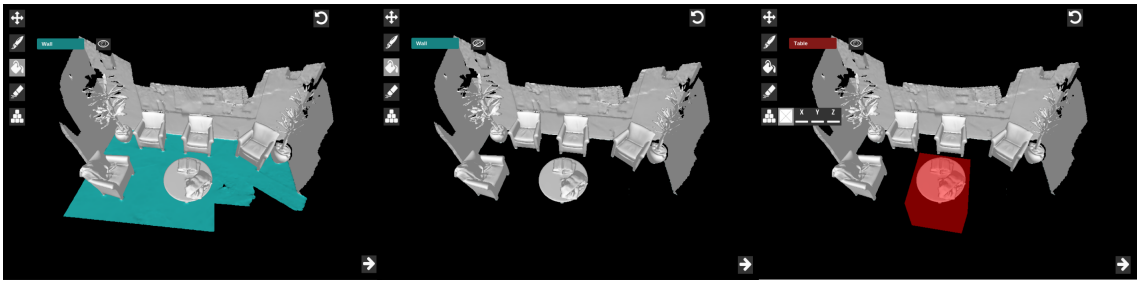

Figure 2: Typical process to start labeling a room. **Left:** First one can use the fill tool to label large flat surfaces such as the floor. **Center:** Second, the labeler can hide the floor using the hide label button in the top left. **Right:** The labeler can then use tools such as the box selector to select remaining objects in the room such as the table.

undo change for example), the user is offered an erase tool to select areas of the model which should be unlabeled.

2.5 The Pressure Brush Tool

The goal of the pressure brush tool is to provide a way to automatically "color in the lines." In 2D for example, if a user was provided a coloring book that is blank, it would be nice if the labeling tool (a crayon for example) hugged the lines to easily fill in sections. The pressure brush tool attempts to mimic this effect in 3D. Since artificial geometry (primarily what we aim to label) has many 90 degree angles that separate distinct surfaces, using these angles as the 3D coloring lines is useful.

When the user applies the pressure brush, it expands around angles gradually such that longer times spent applying the brush will eventually over come these 90 degree angles. The result is the brush very naturally fills in distinct surfaces while being resistant to flooding into distinctly different areas.

2.6 Shape Segmentation Tools

The goal of the set of shape segmentation tools is to provide a easy way to enclose a set of geometry which naturally corresponds to geometry primitives. For example, a chair can naturally fit inside a box. For the room use case, the shape segmentation tools are most useful after walls have been removed. For example, the user can first remove the walls as described in the **Using the Smart Fill Tool to Label Walls and Floors** section, and then once removed use the shape segmentation tools to enclose the objects in the room and quickly label all of the vertices belonging to them. As of now, we provide box and plane segmentation tools. In the future we can expand this to more complex geometry if the use case arises.

3 Tool Usage and Expansion By Developers

In this section we begin by introducing how a developer can obtain the labeling tool, and integrate it with an existing web application codebase. We then explore how a developer launches a simple instance of the tool application through browser JavaScript. After exploring a simple example, we will proceed to explain how a developer can set up a unlabeled model source, and a labeled model destination through a remote web server. Finally, we explore the various customization options

the developer has access to to fit the tool to their particular research needs, and give examples of such setups.

3.1 Project Packaging and Build System

This section describes how the project is distributed and built on various platforms as a bundle of raw non-compiled source files, and how the streaming build system, gulp, is configured to assemble the minified source code for deployment.

3.1.1 Node Package Manager

Node JS, an open source JavaScript runtime for non-browser based JavaScript employs a package manager called Node Package Manager (NPM). This is a convenient way to access a large repository of JavaScript based projects and source code and easily declare dependencies in your project. We utilize NPM for all our dependencies and libraries. To declare dependencies, you create a `packages.json` file. Once created you can declare which libraries you would like to include in your project.

From a developers prospective using our project, you simply have to run `npm install` in the base directory of the labeling project. Once run, NPM will fetch the list of dependencies and download the appropriate files such that our build system can find them.

3.1.2 Gulp Project Compilation

We use a streaming JavaScript build system called Gulp in our project compilation. This allows us to provide multiple steps in the build process, and declare the process as a set of streaming operations. For example, a TypeScript file can be streamed through multiple transformations before being compiled by the TypeScript compiler, and ultimately combined and compressed as a last step.

From a developers prospective using our project, nothing special has to be done to install Gulp. This is because Gulp also happens to be included in our NPM dependency list, so if NPM dependencies are already fetched as described in the previous section, then the developer can proceed to build. To build the project simply run the script `npm run-script make` in the base directory of the labeling project.

Once the project is compiled, there will be a directory `bin/client/static` which is a fully independent top level example of the labeler use. The `bin/client/static/js` folder contains all the compiled JavaScript you need to use the labeling tool in your own client side projects. In particular there is a `main.js` file which is the entire project compiled into one file. This is what you include on pages using the labeler. There is also a folder `bin/client/js/worker` which contains two separate JavaScript files which are launched as workers.

3.2 Instance Based Construction

This section describes the top-level API exposed to the developer and how the labeling tool is constructed on a per-instance basis. Various construction options are also discussed.

3.2.1 Top Level Exposure

When the label tool file is included (`bin/client/js/main.js`) it will execute a very small piece of bootstrap code which injects `LabelerApp` into the global name space (`window`). This allows the developer to write their own script to access the labeling application constructor, and minimizes name space pollution, since the entire application is enclosed in a JavaScript function closure.

3.2.2 Construction and Options

Once the main script file is included, you will have access to the `LabelerApp` object. You can construct an instance of this object like so:

```
var app = new LabelerApp(document.body, {
    MODEL_SOURCE : "data/lounge.ply",
    MODEL_SAVE : "foo/"
});
```

The anatomy of this call is as follows: it creates an instance of `LabelerApp` which is the top level object controlling the entire labeling application. The parameters are a target DOM element and an options JSON object. The application will fill the target DOM element on the page with the canvas application. In this example we are filling the document body element with the application which creates a full screen application instance. The options JSON object contains various options such as the endpoints for server integration as discussed in the next section. For a full list of options look at `src/client/Settings.ts` in the non-compiled source directory.

3.3 Server Integration

A server communicates to the application in three possible ways. First, it usually necessary to have the labeling application served via HTTP for the user's browser to fetch it. This is optional however, as the developer can run the application directly on the client machine without an internet connection. This would require the developer to keep all the JavaScript, CSS, HTML, and other client side files directly on the client machine.

Secondly, the application will make GET requests to fetch the next model from a model loading endpoint. It is the server's job to keep state of the client making a request to this endpoint. This can be done through browser cookies. The request requires the response to be an unlabeled PLY binary file. The developer can write a simple server which responds with a single model, or create a more complicated server. For example, the server could keep a simple counter for the client's request number, and use this as an index into a list of possible PLY models, effectively cycling through the list.

Finally, the application will make POST requests to save labeled models to a model saving endpoint. It, again, is the server's job to implement any state logic between these requests (via cookies for example). A simple example server can take the raw PLY binary in the POST request and write it to storage with a time stamp and client identifier. This allows later utilizing the collection of labeled PLY models in other post-processing steps.

4 Implementation Details

In this section we discuss the various implementation details of the labeling tool. We begin by exploring the Model-View-Controller (MVC) pattern that is fundamental to the application maintainability and functionality. We proceed to discuss the various new web browser technologies that we utilize including WebGL and WebWorkers. We then explore the rendering engine, ThreeJS, and our interaction with it including Vertex Shaders, Fragmentation Shaders, PLY model rendering, and the various rotation and translation systems that are the core of our input system.

We then discuss the application input system, and application modes. Following input discussion, we describe our implementation of the User Interface (UI) system, and the various Document Object Model (DOM) interactions that are part of it. At that point we progress into application controls, and label tool implementation including the creation of a normal-sensitive expansion algorithm, which is the basis for both the smart brush tool and surface flood filling tool.

4.1 Model-View-Controller (MVC)

Model-View-Controller (MVC) is a object oriented design pattern which is central to the labeling tool implementation. The application data including labeled and unlabeled models, tutorial data, undo and redo systems, label information, and segmentation geometries are represented and implemented as models in the application. Controllers act as central action routers, and contain relationships with models. Controllers in the application include our central client controller, and a user interface controller. Views contain any display information and necessary state that is relayed directly to the user interface views in the application. This includes pop up windows for the tutorial system, the UI system, ThreeJS mesh and stage rendering components, and the DOM interaction.

We employ a simple action and event system which utilizes controllers that define application behavior. This allows the flexibility to integrate various application functionality in a decentralized way such that each component only listens for actions, and does not strongly interact with other major application components. In general, views and controllers fire events while controllers listen for them.

4.2 Navigation and Control

During navigation mode we allow navigation in the labeling application such that the user can label all surfaces of an unlabeled model. The implementation of navigation through the 3D environment is a traditional "trackball" implementation with rotation and panning. The rotation implementation uses mouse drags across the XY camera projection plane to turn a virtual sphere "trackball" through unit quaternion transforms. Panning is implemented by changing the center of the virtual trackball. Using quaternion transforms as opposed to Euler angles allows us to avoid Gimbel lock.

4.3 Tutorial System

Since the tool is largely more complex than a simple 2D labeling application, we made the decision to implement a tutorial system that can be scripted to guide users through using the application, or even through labeling an example. The tutorial system has two major subcomponents: A communication system and a tutorial scripting system.

4.3.1 Communication System

To communicate the current stage of the tutorial, we enable two mediums. First, we allow highlighting any set of UI components in the tool. This flexible model allows us to name any number of DOM UI component by ID, and will subsequently blink that component during that stage of the tutorial. Secondly, we enable a simple pop up window system which creates a DOM element with specified text. This element is styled such that is looks like a dialog box over the labeling tool.

4.3.2 Tutorial Scripting System

As touched on by the **Tool Usage and Expansion By Developers** section, we created a system for scripting tutorials. This allows developers to create additional tutorials and enable them through construction options in their label tool instances. This system allows specification of tutorial *stages*, which are steps in the tutorial. During a stage, developers can specify which tools are highlighted, and which actions will trigger the advance of the tutorial forward.

When a tutorial is run (usually on tool start up), we create a simple object which tracks the current location in the tutorial. Additionally, we use HTML5 LocalStorage to track whether or not a tutorial has been run. This allows flexibility in not showing the same tutorial twice.

4.4 Labeling Tools

For the labeling project we created a variety of tools to label 3D models. For each tool we create a new button on the left side which when selected changes the application input mode. This mode directs the input actions to their appropriate controller code that each tool implements. All labeling tools operate with some variation of control based on ray tracing the 2D mouse location to the geometry. Labels themselves are stored as simple JSON objects which describe their label and indexing information for model save functionality. Each vertex then stores a pointer to this JSON object while the labeling tool is running.

4.4.1 The Navigation Tool

The implementation of the navigation tool is simply an toggle switch of the full navigation and control systems discussed in the **Navigation and Control** implementation section. Additionally, we enable a quick "toggle" to navigation mode by holding the control key from any other input mode. When the user initiates this event we simply fire the event as if the navigation button was clicked.

4.4.2 The Pressure Brush Tool

The pressure brush tool behavior is based on the geometry of the model. As an entrance point, we enable a paint brush input mode when the tool UI element is clicked. The pressure brush employs surface normals and a dependency on time to create a pressure effect which favors similar surfaces. During application start up and model load, one data structure we build over the loaded model geometry is a graph of connected vertices. When a paint stroke is applied we obtain a set of candidate vertices that are within our brush stroke radius. To do this, we begin a breadth first search (BFS) of the graph of connected vertices from the paint source vertex limited by the current brush size. This gives us a upper limit on the current paint brush size, and a set of vertices which

we can apply the painting algorithm to.

We then begin filling in vertices starting with the source with a label. This process starts with assigning a cost to each vertex such that the cost is proportional to the dot product between the source vertex normal and the normal of the candidate vertex:

$C(V_c) = N(V_s) \cdot N(V_c)$

Since vertex normals are vectors, this heuristic has the property that

$C(V_c) \sim 0$

When the source and candidate vertex normals are near right angles, and

$C(V_c) \sim 1$

When the source and candidate vertex normals are close to the same direction.

We also follow the property that an vertex can only be labeled if it is connected by a labeled vertex in the graph, we initialize the source vertex to be labeled. As the user holds down the paint stroke in place, we increase a value $H(V_c)$ that is proportional to the time the user has spent holding the paint stroke down. We then multiply this value times the $C(V_c)$ for each vertex, such that when this value passes a constant threshold T we label that candidate vertex if it is connected by another labeled vertex:

$H(V_c)C(V_c) \geq T$

Using this algorithm, the brush fills in surrounding flat surfaces with ease, but avoids painting up sharp edges, and tends to label vertices within a desired area.

4.4.3 The Smart Fill Tool

The smart fill too employs the same algorithm discussed in the **The Pressure Brush Tool** section, with one key difference: the brush size is unlimited, and our threshold value T is altered such that flat surfaces are encouraged more. The result is that walls and other similar flat structures can be flooded similar to a flood fill paint bucket tool found in many 2D image processing applications.

4.4.4 The Erase Tool

The erase tool removes the label from vertices in a defined brush size area. Logically, the brush simply unsets the label from the vertices found in a brushed area. The tool follows a similar algorithm as discussed in the **The Pressure Brush Tool** section. Since erasing felt like more of a bulk operation, we opted for erasing the labels of the entire candidate area enclosed by the initial BFS of the graph of connected vertices on the model. Therefore the tool is not restricted to favoring flat surfaces.

4.4.5 Shape Segmentation Tools

An observation we made is that many shapes in our unlabeled data sets can be contained in simple geometric primitives such as planes and boxes. Therefore, we developed a set of shape segmentation tools which allow you to do exactly that: label a set of vertices contained in geometric primitives. These primitives can be dragged and placed throughout the scene, when the user is satisfied with their selection, they can double click to finalize the labeling. We developed two primitives: planes and boxes. We included planes which align on any one of the X, Y, and Z axises, and label all vertices on the opposing side of the geometry plane with respect to the camera location. This allows easy labeling of large areas of flat surfaces. We also include box segmentation. This simply produces an axis aligned box geometry which can be dragged to contain a set of vertices. Once selected, all vertices within the geometry will be labeled with the current selected label.

4.5 Undo and Redo System

The undo and redo systems are accomplished with a stack-based approach. Controller actions that can be undone are intercepted on the fly and added to the undo stack. A simple construction is used to determine the inverse of an action that can be undone. The construction returns another action which will revert the application to a state just before the action being undone was applied.

The system also allows batching. We simply enter a batch change surround grouped sub-routine tasks, and when the logic returns from the sub routine we exit the batch change which will then add the batch change to the stack. Batch changes are reverted by finding the inverse change for each action in the batch as described above. Both batch and normal changes are encapsulated into a separate object which can be applied similar to the raw actions they encapsulate.

View logic is simply controlled by the state of the undo and redo stack. If either the undo or redo stack contain changes we display the associated button. In the case that an action is taken when there are changes in the redo stack, we simply empty the redo stack to preserve state integrity in the application.

4.6 Layer Label System

Similar to many 2D processing applications, our labeling tools employed the notion of layers. Layers cannot be created or destroyed and are directly tied with a one-to-one relationship to the set of labels that the user can label vertices with. For example, if a “chair” label is available, there will also exist a “chair” layer which contains all vertices that have been assigned the label “chair.”

4.6.1 Hiding and Showing Layers

Layers can be shown and hidden via the hide and show toggle button located next to the label selector tool. Showing and hiding labels is done via Vertex and Fragmentation Shaders in OpenGL. We assign a boolean vertex attribute which dictates whether or not that particular vertex is shown. The shader utilizes these attributes to alter the appearance of particular vertices on the screen. We then alter these attributes in batch to coincide with the state of layer visibility. For example, when a layer is hidden, we traverse the set of vertices that have that layer’s label and apply a hidden attribute to them.

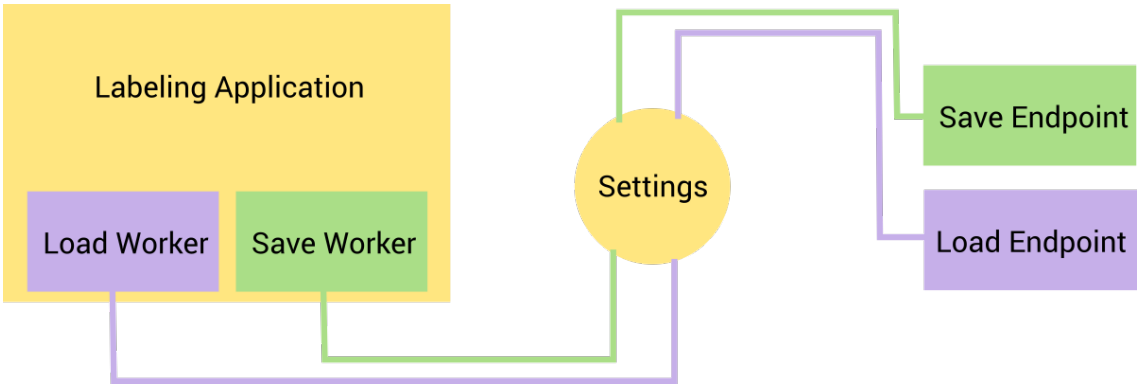

Figure 3: Data flow and processing of the labeling application. Unlabeled model binaries are fetched from the server endpoint specified in the constructor settings via HTTP GET request. Once downloaded to the client, raw model binaries are parsed client side into appropriate data structures via a web worker. After models are finished being labeled and are to be saved, a web worker takes application data structures and outputs a labeled model binary. This binary is sent to the server endpoint specified in the constructor via HTTP POST request.

4.7 Server Connection

The application was designed to operate almost entirely client side to make it portable and easy to use for developers. We make connections with the server for two key operations, model loading and model saving.

4.7.1 Model Loading

Model loading simply makes a HTTP GET request to the specified model loading URL. We intentionally made this operation simple as it allows complex loading schemes to be implemented by a server. For example, you can keep client state via HTTP Cookies on the server application and for each subsequent model loading request send an arbitrary list of models to be labeled. You can therefore hook this labeling application up to a private server which contains unlabeled models, and continuously serve them to labeling clients. The response of the GET request must be a valid PLY file. If no valid PLY file is specified, a client error will occur.

The PLY file is loaded and immediately dispatched to the load web worker which runs in a separate thread to parse the raw binary buffer into various data structures used by the application. The final data structured are likewise passed back as a message to the main thread. Loading the model in this way allows the main UI thread to be uninterrupted during application load time.

4.7.2 Model Saving

Model saving is likewise simple and open ended for implementation. When a model is saved the application will make a HTTP POST request to the developer specified model saving URL. The payload of this request is a binary blob of the saved PLY file.

To form this binary, we first launch a instance of the save web worker which runs in a parallel thread. The web worker receives the various data structures representing the labeled model in the application. The web worker then constructs a final binary buffer representing the saved label model. The final binary buffer is a PLY file that contains an extra label list appended to the end

in valid PLY binary format. Each vertex in the saved PLY file contains an extra byte which acts as an index into the list of labels contained at the end of the PLY file. This representation minimizes the extra space required by avoiding storing the entire label at each vertex.

4.8 Browser Technologies

We used multiple cutting edge browser technologies to enable a hardware accelerated smooth user experience using our labeling tool. These browser technologies are widely supported in all major latest browsers and platforms, which allows us to take advantage of the web platform for wide distribution and adoption.

4.8.1 ThreeJS, WebGL, and Shaders

WebGL is a relatively new browser standard that enables hardware accelerated 3D graphics in the web browser via OpenGL interfaces. We utilize WebGL as our primary rendering platform to allow users to interact with the labeling tool. The current loaded model is shaded using Fragment Shaders that are loaded as external files. The Fragment Shaders shade pixels with label colors and simple dot product luminosity via a projection from the camera viewing angle. We also utilize the fragment shader to hide pixels which are labeled with a currently hidden label. ThreeJS gives multiple thin layers of abstractions for easy manipulation of the scene. ThreeJS also has the benefit of providing access to the underlying WebGL and buffer representations of model geometry such that we can build our own efficient data structures over the unlabeled models. Using ThreeJS allows significantly more effort to be spent on application logic instead of the underlying rendering engine.

4.8.2 Web Workers

One down side of modern JavaScript in performance critical applications is that it is single threaded, and follows a event based concurrency model. When execution stacks are short lived, this model behaves well; however, because user interface updates also share this central thread, the user experience can suffer when execution stacks are long lived. Web Workers provide a simple message based protocol for browser multi-threading. The process to create one is to specify a target JavaScript file and communicate through a limited set of data types including buffers and strings. Once a Web Worker is launched, it acts as an independently run JavaScript application in the browser.

In our work, we integrate Web Workers by creating a set of worker TypeScript classes which are independently compiled to separate minimized JavaScript files to be launched by the browser Web Worker API. During development of this tool, we found a large slow down during PLY model loading and saving since parsing and building the application data structures for complex 3D models proved to be a long operation. This led to the experience of a locked up browser while the application initialized. To fix this problem, we opted to move all heavy data processing to HTML5 Web Workers.

During application load, a Web Worker is initialized to receive the downloaded raw model buffer and parse it into the set of data structures our application understands. The result is that users can interact with the page and see an animated loading screen, as opposed to their browser locking up, and in some cases, crashing.

5 Future Work

For future work, we would like to test the tool with people of varying backgrounds to find the average time spent for labeling scenes. In this way we can benchmark the effectiveness and simplicity of the labeling tool. We would also like to run tests on the effectiveness of the annotations and the noisiness of the labeled data with respect to the input model. In this way we can see how well the tool performs with varying levels of quality in the input model.

With the explosion in 3D sensing capabilities, we expect a large amount of 3D data to become available. A large body of work with this data is the ability to index it and utilize it for 3D detection and recognition. There are many places that we can extend the labeling tool as well. One such example is the ability to annotate objects with 3D characteristics, such as the ability to pivot or rotate. This meta information can then, too, be utilized to to infer information about new scenes.

Another body of future work is curating of workers with a higher expertise level than those found in the general audience (e.g. from general Mechanical Turk users). Labeling 3D geometry in an accurate and useful fashion is not trivial. To alleviate this issue, labeling can also be semi-automated to assist those marking areas of interest on the unlabeled model. For example, if we are able to gather hints about the orientation of the model from the user, we could infer which surfaces are floors and walls and automatically hide them. With larger amounts of data we can employ machine learning techniques to partially label or point out areas of interest as well. The goal here is to reduce the amount of work required of a participant labeling geometry. In this way we can reduce the amount of tools and options available to annotators. We envision such an operating mode for the regular computer user. This mode would require more detection and software assistance, and may require more assumptions about the unlabeled model geometry.

6 Conclusion

Our work is a small part of a large effort in a new frontier of incredible 3D computer vision technology. We have created a tool to provide the ability to add semantic labels to an otherwise static view of the virtual world. With emerging rendering and computational power in web browsers, this tool targets the web platform utilizing a full 3D viewer to manipulate and label 3D scenes. This tool and others like it are a stepping stone towards indexing and labeling large quantities of 3D geometry which is increasingly becoming available. We believe work such as this will enable 3D scenes that can be automatically labeled, allowing computers to understand the world in more complex and useful ways.

7 Acknowledgments

I thank Steve Seitz and Richard Newcombe for providing guidance during this project's development, Ryan Drapeau for discussions and company while writing this paper.

8 References

[Biederman, 1987] Biederman, I. (1987). Recognition-by-components: a theory of human image understanding. *Psychological review*, 94(2):115.

[Henry et al., 2012] Henry, P., Krainin, M., Herbst, E., Ren, X., and Fox, D. (2012). Rgb-d mapping: Using kinect-style depth cameras for dense 3d modeling of indoor environments. *The International Journal of Robotics Research*, 31(5):647–663.

[Howe, 2008] Howe, J. (2008). *Crowdsourcing: How the power of the crowd is driving the future of business.* Random House.

[Hsueh et al., 2009] Hsueh, P.-Y., Melville, P., and Sindhwani, V. (2009). Data quality from crowdsourcing: a study of annotation selection criteria. In *Proceedings of the NAACL HLT 2009 workshop on active learning for natural language processing*, pages 27–35. Association for Computational Linguistics.

[Izadi et al., 2011] Izadi, S., Kim, D., Hilliges, O., Molyneaux, D., Newcombe, R., Kohli, P., Shotton, J., Hodges, S., Freeman, D., Davison, A., et al. (2011). Kinectfusion: real-time 3d reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 559–568. ACM.

[Kittur et al., 2008] Kittur, A., Chi, E. H., and Suh, B. (2008). Crowdsourcing user studies with mechanical turk. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 453–456. ACM.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[Russell et al., 2008] Russell, B. C., Torralba, A., Murphy, K. P., and Freeman, W. T. (2008). Labelme: a database and web-based tool for image annotation. *International journal of computer vision*, 77(1):157–173.

[Vondrick et al., 2011] Vondrick, C., Patterson, D., and Ramanan, D. (2011). Efficiently scaling up crowdsourced video annotation. *International Journal of Computer Vision*, pages 1–21. 10.1007/s11263-012-0564-1.