

Process Book

Basic Info

The project title:

Post-processing Image Statistics for Bidirectional Path Tracing

Git repository:

<https://github.com/WBPPISBPT/wbppisbpt.github.io>

Laura Lediaev

lediaev@sci.utah.edu

U0863084

Ouermi Timbwaoga Aime Judicael

touermi@cs.utah.edu

U1078389

Hirad Sabaghian

hirad.sabaghian@gmail.com

U0930231

Post-processing Image Statistics for Bidirectional Path Tracing

Table of Content :

1.	Introduction
2.	Motivation
3.	Overview and Goals
4.	Data
4.1.	Data Processing
4.2.	Data transmission bottleneck
4.3.	Data Assimilation
5.	Design Evolution
5.1.	Initial Design Iteration
5.2.	Final Design
5.3.	User Guide
6.	Implementation
6.1.	Iteration 1
6.2.	Iteration 2
6.3.	Iteration 3 (Final product)
7.	Evaluation

1. Introduction

Bidirectional path tracing is a physically-based rendering technique. In many cases it is capable of producing a rendered image that is practically indistinguishable from a photograph. The rendering program attempts to simulate the physical properties of all elements of a scene, including lights, geometric objects, materials, and cameras. Path tracing is a Monte-Carlo integration technique that generates paths using probability distributions.

Regular (unidirectional) path tracing starts by choosing a random position on the film plane and then randomly choosing a position on the camera aperture. These two choices will create a ray that shoots from the camera into the scene. Every time the ray intersects a surface, the material properties of the surface are used to choose a new direction to travel in. The ray continues to bounce around the scene until one of three conditions is met. The ray may hit a light creating a complete path, at which point the ray's path is terminated. Alternatively, the ray may have bounced so many times that we choose to terminate it. Lastly, the scene might be open and the ray may have gone off into outer space (a non-hit). Every surface interaction, including the camera, is called a vertex. A complete path starts at the camera and ends at a light, and every vertex contains information needed to calculate the final color contribution for that path.

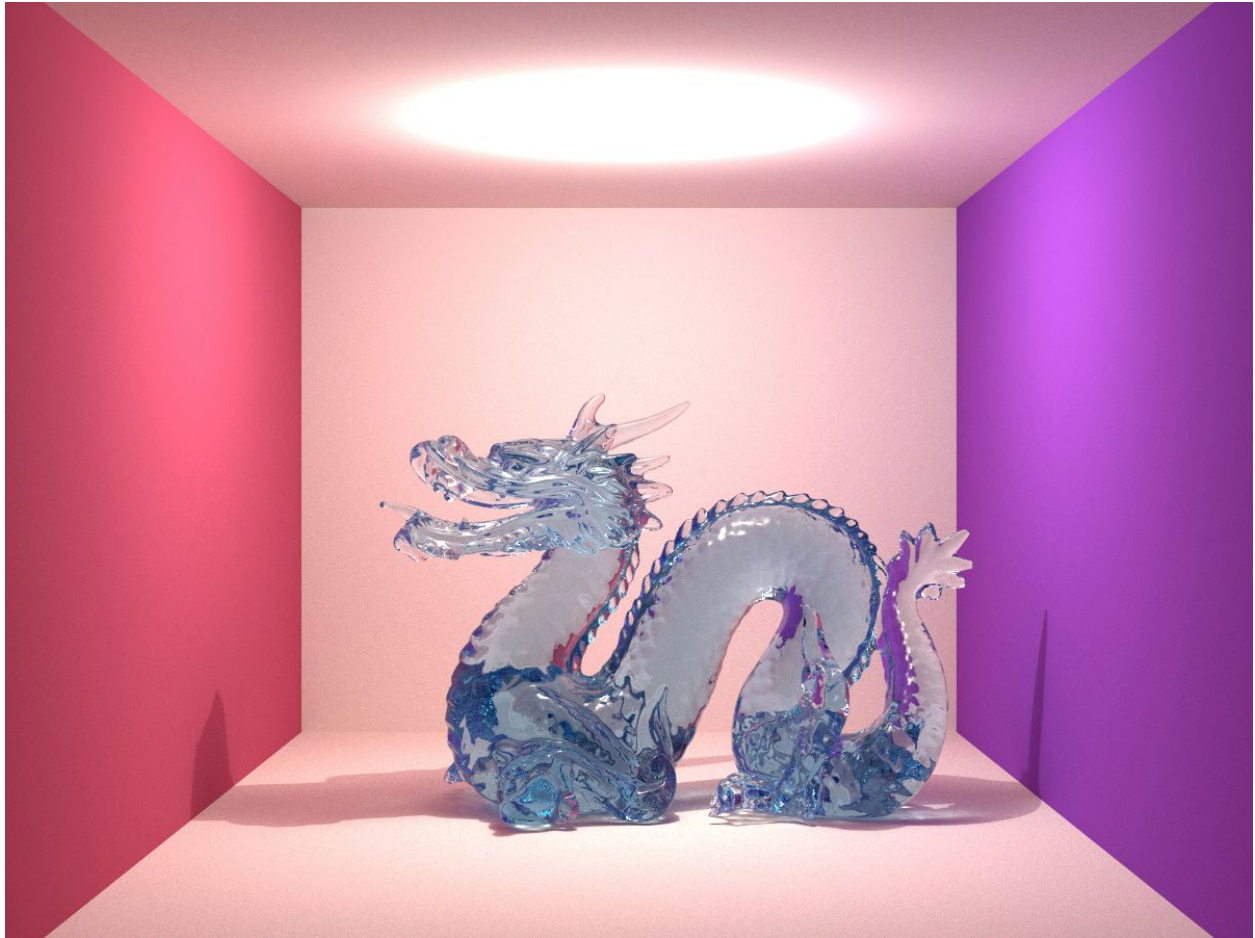


Figure 1: An example image rendered with path tracing.

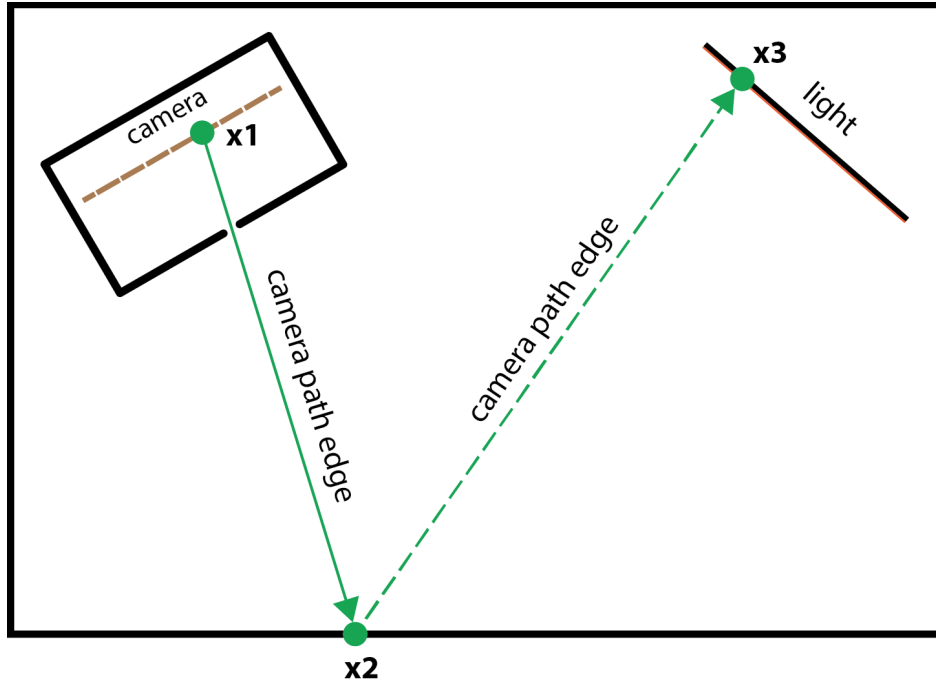


Figure 2: A diagram showing a complete path with two edges.

Although unidirectional path tracing will produce a converged image, it may take a very long time. Bidirectional path tracing attempts to reduce the amount of time needed to render a finished image. It does this by combining several different rendering techniques and averaging them in a clever way in order to reduce noise. The general term for this type of averaging is multiple importance sampling (MIS). The only necessary condition for using MIS is that we know the probability for generating a path for each type of technique. Every time we generate a path with a specific technique, we then calculate the theoretical probabilities for creating the same path using the other possible techniques. Once we know all the probabilities, we weight the current path contribution by the relative probability of our path compared to the other theoretical methods. If the probability is relatively small, then the weight will be small. This is useful because low-probability paths often cause problems when rendering an image. These samples can cause bright spots (fireflies). When using MIS we hope that at least one of the methods can generate a path with a high probability, thus eliminating fireflies.

Figure 3 shows explicit path tracing. Implicit, or classic, path tracing allow a camera ray to bounce around a scene until it hits a light by accident. At that point the total path contribution is added to the pixel average. In contrast, with explicit path tracing we generate the camera paths in the same way, but now we explicitly sample a light from every vertex on the path. A complete path starts at the camera and ends at a light. By sampling a light at every vertex we generate a sequence of paths (each with a different number of edges) which increases the probability of generating a good sample, and this

reduces noise. Sampling does not guarantee a complete path because an object may be between the vertex and the light sample (the light is occluded). Usually, explicit path tracing produces complete paths with much higher probability than implicit path tracing, but in some situations implicit path tracing is actually better than explicit path tracing. Given the nature of the scene that we want to render, sometimes we know which type of path tracing will work best, but sometimes different parts of the scene will work better with a particular method.

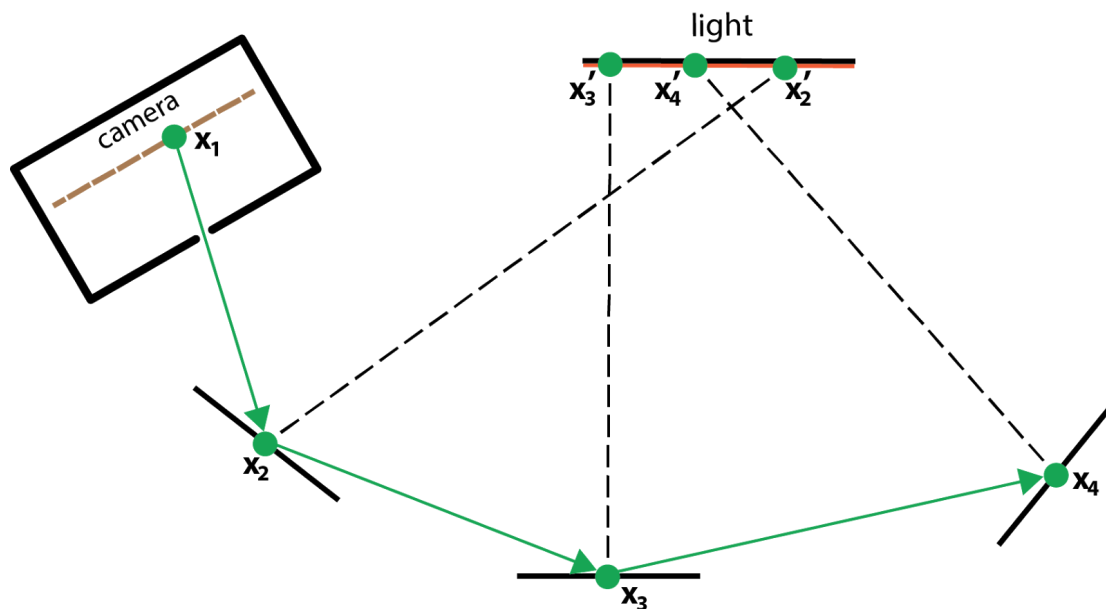


Figure 3: A diagram showing the process of each path vertex sampling a light.

Light tracing, as shown in figure 4, is a different rendering method but is still similar to path tracing. The main difference is that paths begin at light sources, bounce around the scene just like path tracing, and then each vertex on the light path is connected to the camera. Light tracing offers a third method for rendering, but does a very poor job of rendering specular parts of the image. Specular refers to materials such as mirror or glass. Light tracing does do a good job of rendering caustics, which is light that is concentrated by specular objects onto a diffuse surface. Light collecting on the ground below a glass sphere is an example of a caustic, and is shown diagrammatically in figure 5.

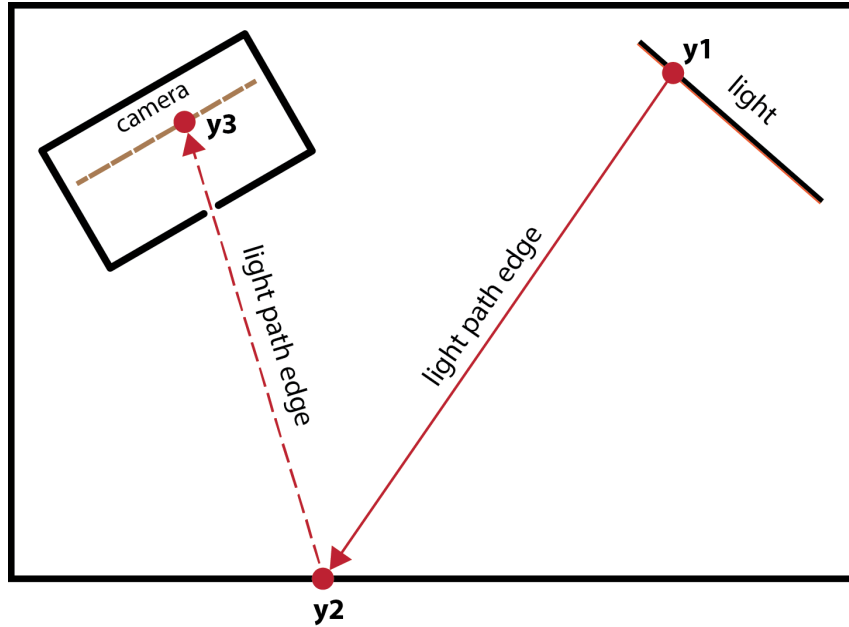


Figure 4: An example of a path generated by light tracing.

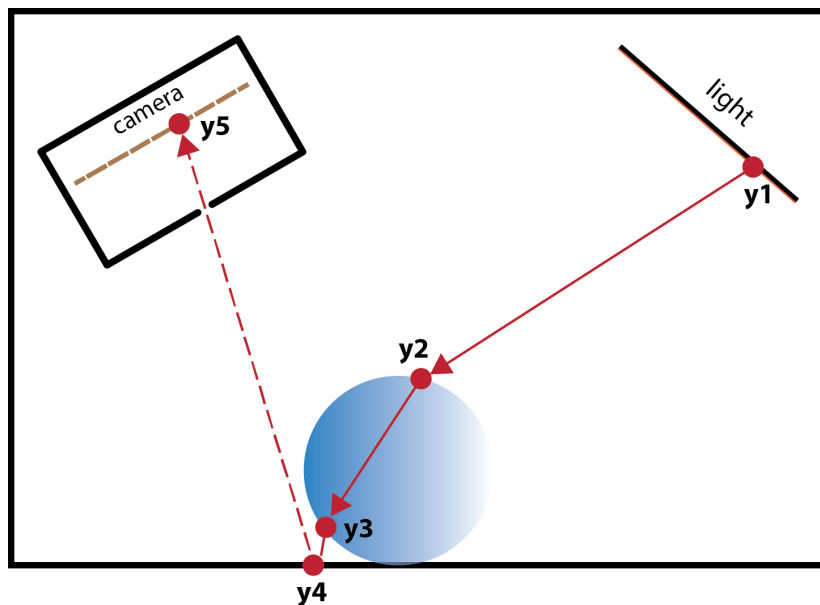


Figure 5: A light path showing a caustic beneath a glass sphere at vertex y_4 .

Bidirectional path tracing combines path tracing and light tracing. Figure 6 shows how this is done for a path with three edges. A vertex from the camera path is connected deterministically to a vertex on the light path, creating a new complete path. The probability for this new path is the product of the probability for generating the camera path and the probability for generating the light path. Things get more complicated when

we have longer paths. Figure 7 shows when we can create several new paths depending on which two vertices we choose to create a connecting edge.

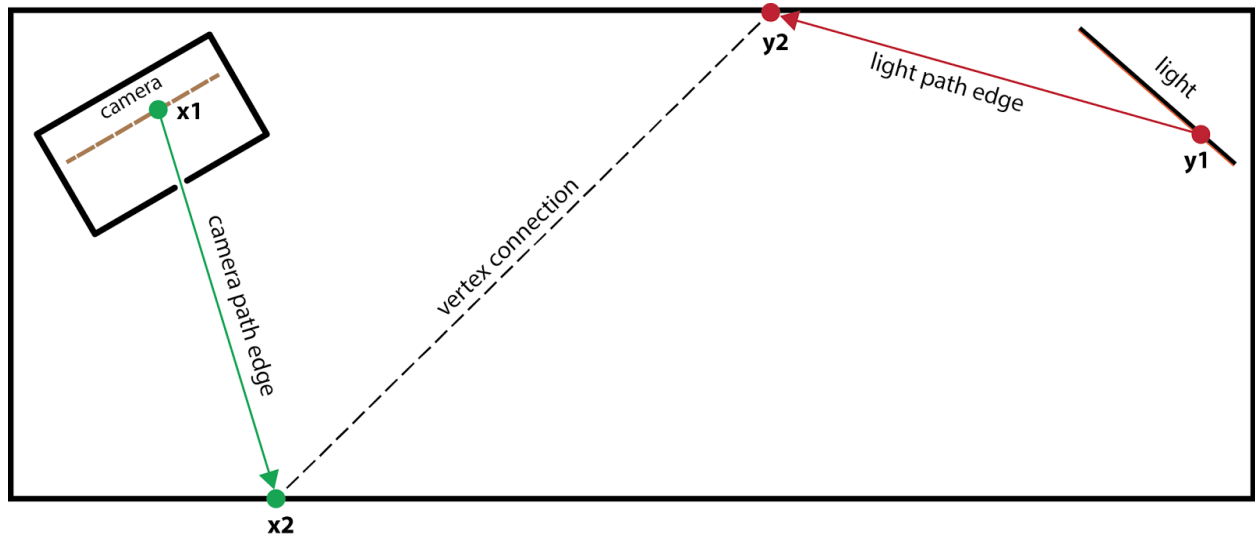


Figure 6: A bidirectional path with three edges, combining a camera path with a light path.

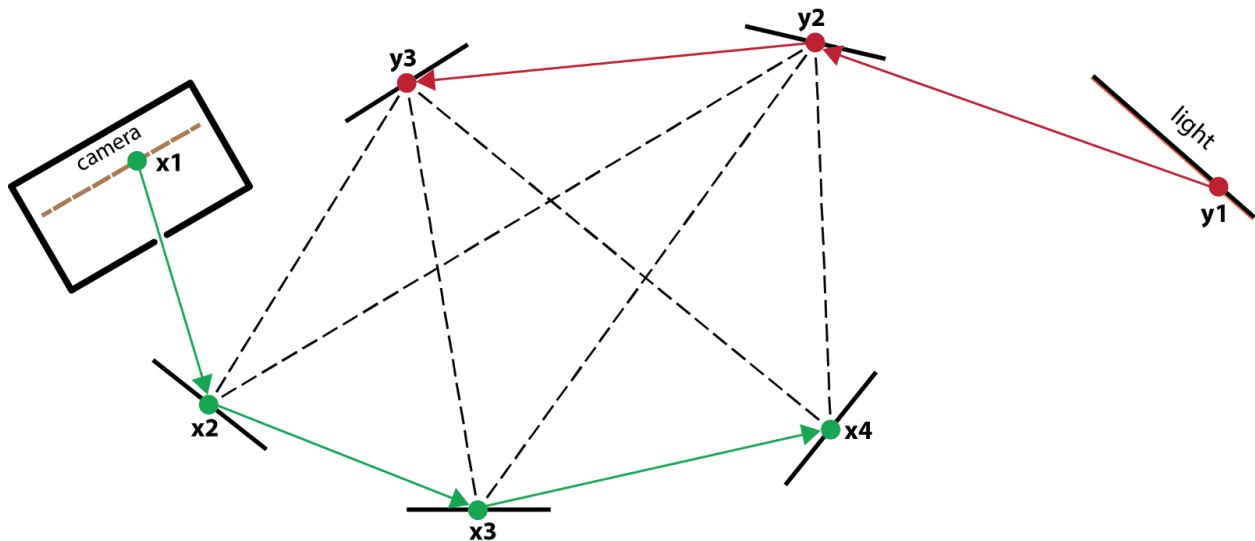


Figure 7: In bidirectional path tracing a collection of new complete paths is created by connecting different vertices from a camera path and a light path.

2. Motivation

Laura (Team member) initial interest in this project was to help me debug my rendering program. Her interest has evolved to include post-processing and compositing. Having all information available to recreate an image allows a great deal of possibility in processing the image to either improve how it looks or to help understand how different types of image components contribute to the final image. A variety of sources for probabilities are involved in generating pixel samples. This is somewhat complicated, in a combinatorial sense. Probabilities for explicitly created paths need to be compared to the theoretical probabilities from all other methods that can create the same path. This generates a sort of matrix of probability comparisons. It would be nice to be able to visualize these comparisons in order to confirm that the program is working correctly. Another motivation is being able to identify problematic pixel samples, and explore what makes that sample problematic. Lastly, we can reconstruct the final image after possibly eliminating certain problematic paths or/and sample, thus creating a highly converged image that is free of noise and fireflies.

3. Overview and Goals

c Such low-probability paths can result in outlier pixels, informally referred to as fireflies, that do not fit in terms of color with their neighboring pixels due to intense brightness. Through visualizing path statistics and probabilities, one could easily trace the source of the error and proceed with modifying the image to eliminate the outlier pixel samples, or conversely changing the scene to make rendering easier.

There is one question that is very important to answer, which samples cause a pixel to be a great deal brighter than it should be. These “fireflies” are a common nuisance in rendering, and can come from many difference sources. These bad samples are not incorrect, strictly speaking, but would require a very high number of samples in order to average to the expected (correct) color value. That is because these are usually very low probability samples. We possibly need an unreasonably large number of samples to get a substantial set of these rare samples. In order to produce a nice image in a reasonable amount of time, it is often more pragmatic to simply remove these samples from the image.

The other questions involve comparing probabilities of generated paths with the calculated probabilities of theoretical methods of producing the same path. This “correspondence” is important, as all methods should agree with other when calculating probabilities. This ensures the rendered image is not too bright or too dim. Being able to confirm correspondence helps verify that a rendering technique was implemented correctly.

4. Data

The actual data is generated as a matter of course during the rendering process. Normally this data is generated iteratively, processed to get a final color value, and then discarded in order to minimize memory usage. Since there will be a large amount of data, We have decided to store everything in a database. The database may need to be optimized for queries, since fetching data may cause a bottleneck. My rendering program is written in C++ and can create and write to an SQLite database file. All information necessary to recreate the final rendered image will be saved to the database. The data has not yet been created, but Laura is currently designing the database structure and we will consult with a couple people who have experience in database design.

4.1. Data Processing

We will not need to change or clean up the data after it has been saved to the database since we are generating the data specifically for this project. The SQLite database file will reside on the server, and I will need to create an interface to conduct queries on the database. Our preference is to use Node.js on the server to execute database queries. Another possibility is to use PHP. For the most part, we will need one main function to execute SQL query strings, since We do not need to worry about security issues. For testing purposes, we can run our project on the server itself (e.g. my laptop), which will reduce data transit times.

Data will be saved to the database in a top-down approach. The highest level is the final image. The image is made of pixels. Each pixel contains samples. Each sample contains a weighted sum of paths. Each path contains a list of vertices. The vertices specify most of the specific data, such as probabilities, physical 3D coordinates, color throughput, hit object information, material information, etc. This is the exact same data

that was used to calculate the final pixel values, and thus it can be used or manipulated in order to regenerate the image, and to explore sub-images (partial images using only certain types of paths).

4.2. Data transmission bottleneck

We are dealing with a large amount of data in this project thus we have to carefully think about the amount of data that is sent from server to client and vice versa.

** Assume the conventional 1 byte per character

** The transmitted JSON will consist of only characters, meaning regardless of type, be it int, float, or char, everything is transmitted as characters. (1.5 is 3 chars and 24 bytes even though computationally it is 4 bytes)

From our final design we can derive the following attribute information for each entity:

- **Sample:** 1 UID [STRING] 1 Num[FLOAT] 2 Total
- **Path:** 1 UID [STRING] 4 Num[FLOAT] 5 Total
- **Vertex:** 1 UID [STRING] 3 Text [STRING] 1 Num[FLOAT] 5 Total

Since everything is transmitted in characters, let's be naive and assume that an attribute, be it UID or a numerical value is on average 8 characters long.

- **Sample:** 2 Total $2 \times 8 = 16$ bytes total per sample
- **Path:** 5 Total $5 \times 8 = 40$ bytes total per path
- **Vertex:** 5 Total $5 \times 8 = 40$ bytes total per vertex

Now let's understand the hierarchy of our data and try to derive a formula from the above assumptions for calculating the size of data transmitted for a single pixel.

Samples include Paths include Vertices

$$\begin{aligned}
 & (\text{\#ofSamples} * 16) + (\text{\#ofSamples} * \text{\#ofPaths} * 40) + \\
 & (\text{\#ofSamples} * \text{\#ofPaths} * \text{\#ofVertices} * 40) \\
 & = \\
 & \text{\#ofSamples} * (16 + \text{\#ofPaths} * (40 + (\text{\#ofVertices} * 40))) \\
 & \text{\#ofSamples} * (16 + \text{\#ofPaths} * 40 * (1 + \text{\#ofVertices}))
 \end{aligned}$$

Example Data Size for each Pixel:

100 samples per pixel, **10** paths per sample, **5** vertices per path

$100 * (16 + 10 * 40 * (1 + 5)) = 241600 \text{ bytes}$ 241.6 kilobytes

100 samples per pixel, **10** paths per sample, **10** vertices per path

$100 * (16 + 10 * 40 * (1 + 10)) = 441600 \text{ bytes}$ 441.6 kilobytes

200 samples per pixel, **10** paths per sample, **10** vertices per path

$200 * (16 + 10 * 40 * (1 + 10)) = 883200 \text{ bytes}$ 883.2 kilobytes

500 samples per pixel, **10** paths per sample, **10** vertices per path

$500 * (16 + 10 * 40 * (1 + 10)) = 2208000 \text{ bytes}$ 2208 kilobytes

We have to be conscious of our data and the renders must have a limit on sample size, paths and bounces, so that the data transmission doesn't take forever and remains manageable. If the client and server are on a local machine, this is not an issue, but it definitely poses challenges when we are transmitting between remote machines.

For example:

1000 samples per pixel, **12** paths per sample, **12** vertices per path

$1000 * (16 + 12 * 40 * (1 + 12)) = 6256 \text{ KB}$ 6.256 MB A 5M & 40S MP3!

It can quickly get out of hand, and parsing a 6MB JSON file is a challenge on its own, let alone trying to do computations too!!! Even if we assume a max length of 4 chars instead of 8 it is still 3MB! And let's keep in mind, this is data only for a single pixel

4.3. Data Assimilation

One challenge we face is the data assimilation. In this project we have to generate the data ourselves. Thus we have to build bidirectional path tracer to collect the data we need for the visualization. In order to anticipate the difficulties we assign the had one of solly focus on collecting the data.

- Single pixel data by the end of Version I
- One image data by the end of Version II

- Multiple Image data for Version III

The objective is to collect the necessary data for a single pixel by the the milestone submission deadline. We were able to collect the data but it was not connected to our Visualization according to milestone.

At second stage of the data assimilation, we were suppose the have the data of one image connected to our web based visualization. We were not able to meet this deadline. This called for reevaluation of data assimilation.

We decided that we need to have a second teammate help with data collection so we can have some functioning prototype at the earliest possible.

5. Design Evolution

5.1. Initial Design Iteration

We went through multiple design iterations to find the best iteration to identify an optimal User Interface that will adequately encapsulates the project goals described above. In this project we are aiming to provide the user with a tool where they can visually analyze the rendering method for a specific region of a given picture. First the user has to select a region of a given picture. Once the region has been selected we query the data corresponding to that region.

To visualize the data of the selected region we first show the outliers of the region and then allow the user to choose a pixel to further explore the behavior of a specific outlier. After the user has selected the pixel of interest, we visually encode the sampling information of that pixel so the user knows what kind of paths and probabilities went into creating that pixel. The sketches below attempt to provide an optimal design that not only captures the features of this tool but gives the user an easy and intuitive interface to interact with.

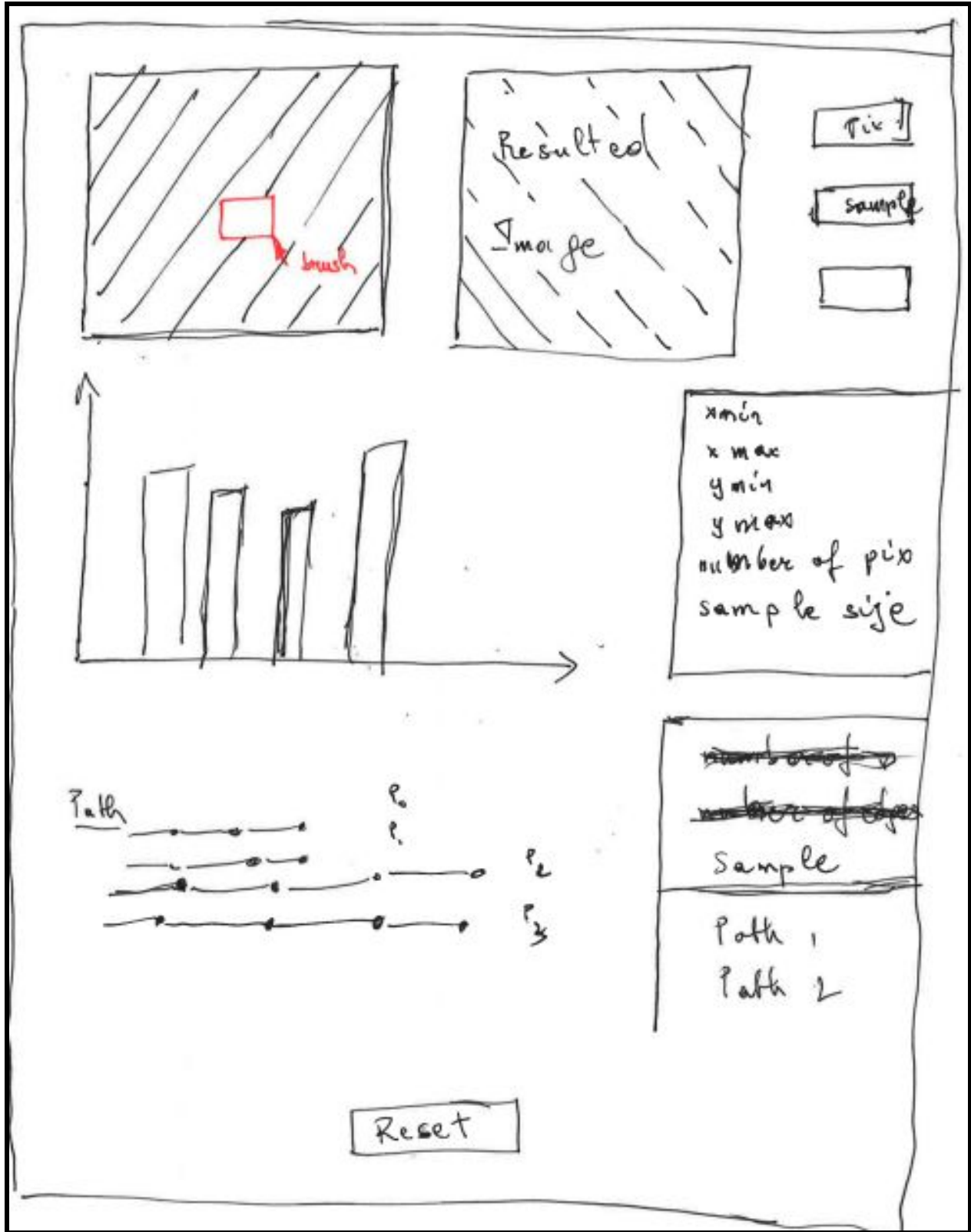


Figure 8: Sketch 1.

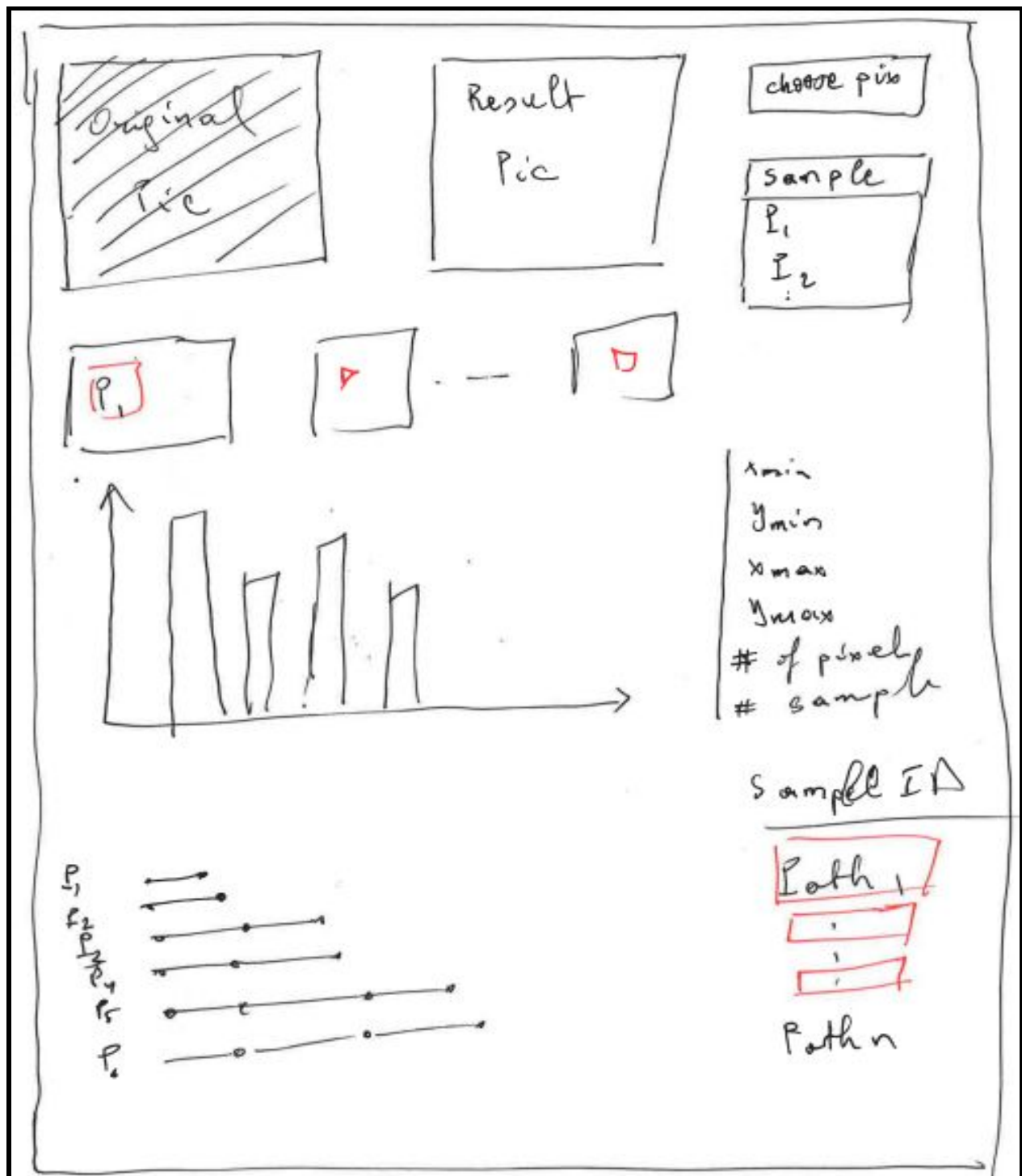


Figure 9: Sketch 2.

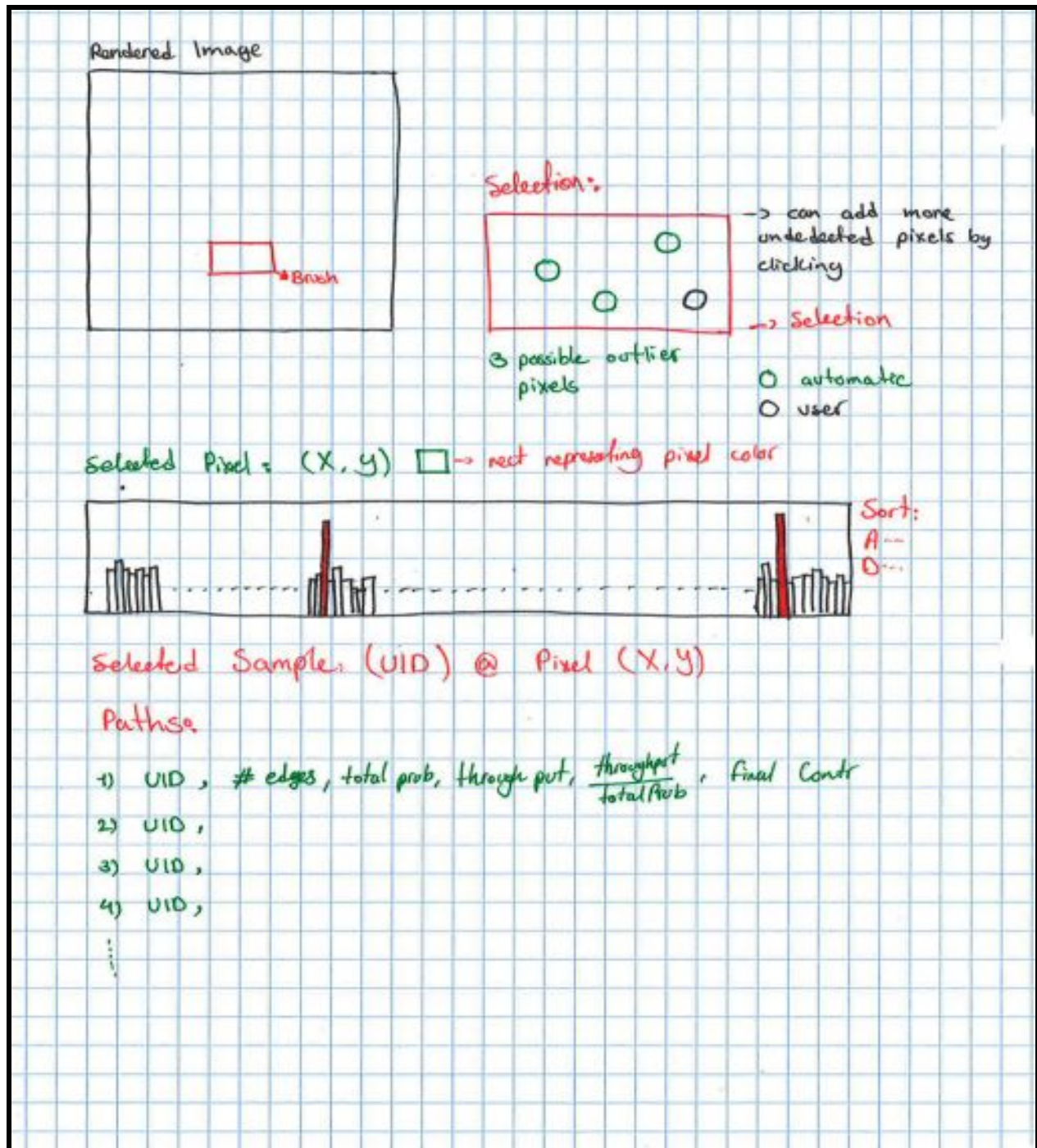


Figure 10 : Sketch 3.

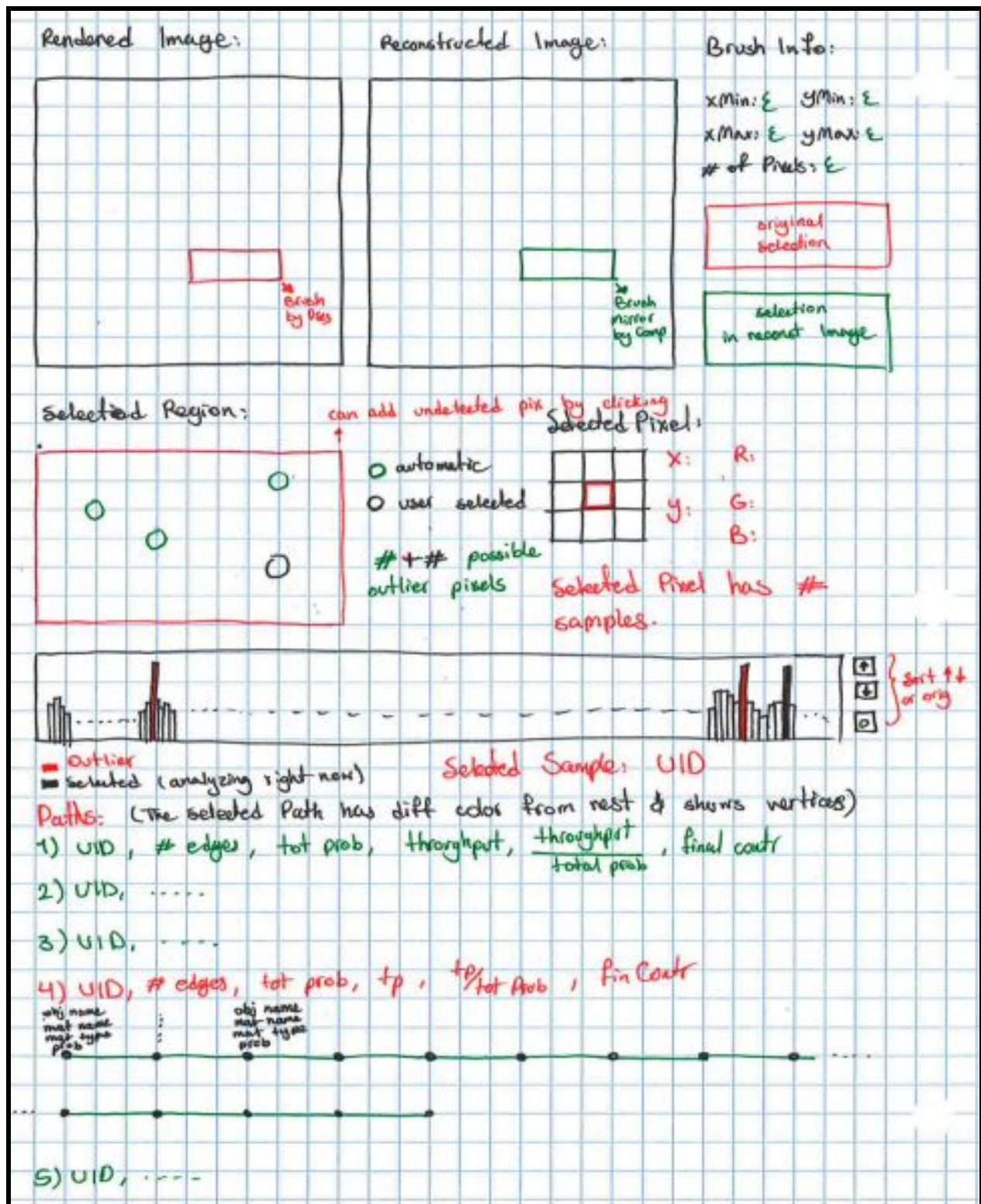


Figure 11 : Sketch 4.

Figure 11 (the 4th) builds on the ideas presented in figures 8, 9, and 10.

- 1) Putting both pictures (Original and Process) next to each other allows the user to easily compare the process picture with the original one.
- 2) The top right corner shows some information about the brushing which allows the user to know the area they selected with the brush, and the number of pixels selected.
- 3) The second row in our final design shows a zoomed version of the brushed area and the different outlier pixels. This allow the user to clearly see which pixels are the outliers so they can easily select one to explore further.
- 4) On the right side of the middle part, we show the number of outliers and some information about the pixels selected by the user. The information is comprised of the RGB color values and the sample size. The color encoding of the pixel selected is different than the others to clearly distinguish it.
- 5) Below the middle part we have a histogram which shows the same information as above and clearly distinguishes the outliers with a red bar. Furthermore, we distinguish the selected outlier using a different color. To the right of the histogram we have different buttons that allow the user to sort the histogram.
- 6) The bottom part of our design has a table which contains the different paths that make up our sample. The table allows to clearly show the different components of each path. In this bottom part we incorporated a line view on the different vertices and edges that constitute a path. This line view can be seen by selecting a specific row.

5.2. Final Design

- **Can be implemented towards the end**
- **Best to implement at the end**

1. UI → Original Image View

- ◆ Get an image •
- ◆ Brush Selection

2. UI → Firefly Detect View

- ◆ Get brush region
- ◆ Detection

- Automatic
- User-defined
- ◆ Pixel Mapping
 - $UI[x, y] \rightarrow Image[x, y] \rightarrow Pixel$
- ◆ `circle.onClick()`
 - Request samples from server for $[x, y]$
 - Request all paths from server fro $[x, y]$

3. UI → Samples View

- ◆ Sort Function
 - Ascending
 - Descending
 - Original
- ◆ `bar.onClick()`
 - Populate Paths View
 - Sample Elimination ○
 - Reconstruct image and display

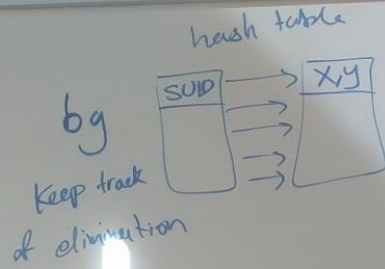
4. UI → Paths View

- ◆ Tabular implementation similar to HW4 (Countries)
- ◆ Sortable columns
- ◆ `path.onClick()`
 - Populate Vertices View
- ◆ Reconstruct image usings paths of specific size ○
 - Reconstruct image and display

5. UI → Vertices View

- ◆ Tabular implementation similar to HW4 (Games)
- ◆ Consists of vertices connected by edges
- ◆ `vertex.onHover()`
 - Display vertex information

17!



of optional

* 0) get img

1) UI → Orig Im
get an brush sel

2) UI → fire fly

get select
automatic
manual
fn(UI)

3) data retrieval

amples

Orig
is easy

click()
elimination
this

all
ter based on person

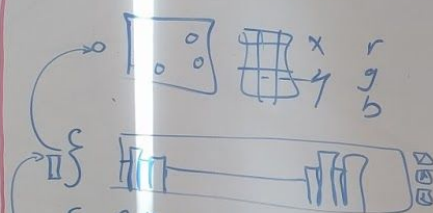
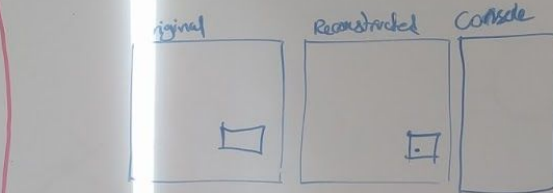
click()

ts

it + put

()

of paths w/
length



Paths:

1) UID - thypet - tot prob - $\frac{tp}{fp}$ - fi contrib

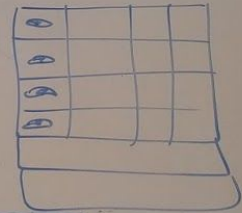
2)

3) obj num
mat num
type
prob



eye	w	prob	length
elim	o	o	o	o	o	o

deleted Sample "UID" w/ int "-" from Pixel (x,y)



5.3. User Guide

After meeting from with TA we decided that we wanted to have a user guide and a screen cast. However as we progressed in the project. We realized that we were behind in schedule. Thus we revisited the our goals and decided that the user guide was not necessary since we are doing a screencast and that will go back to if time allows.

At the moment below is the current state of the of our user guide and consider it to be lower priority

User Guide

This document is a user guided design to facilitate the navigation of our post-processing image statistics for bidirectional path tracing tool. It gives a the user example on how to to use the different feature found in this tool.

1. Load image

The first step in using this web base tool is to load an image of interest to analyse. This tool does not allow the user to load their personal image. User must choose from the given set of images available

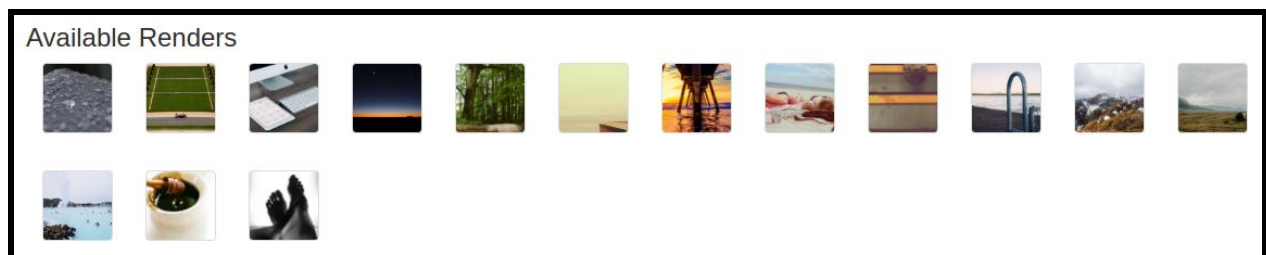


Figure 1: set of available images

The user choose and image by clicking on one of the image as show above. Once the user has selected the image. The image will be displayed in the selected render window.

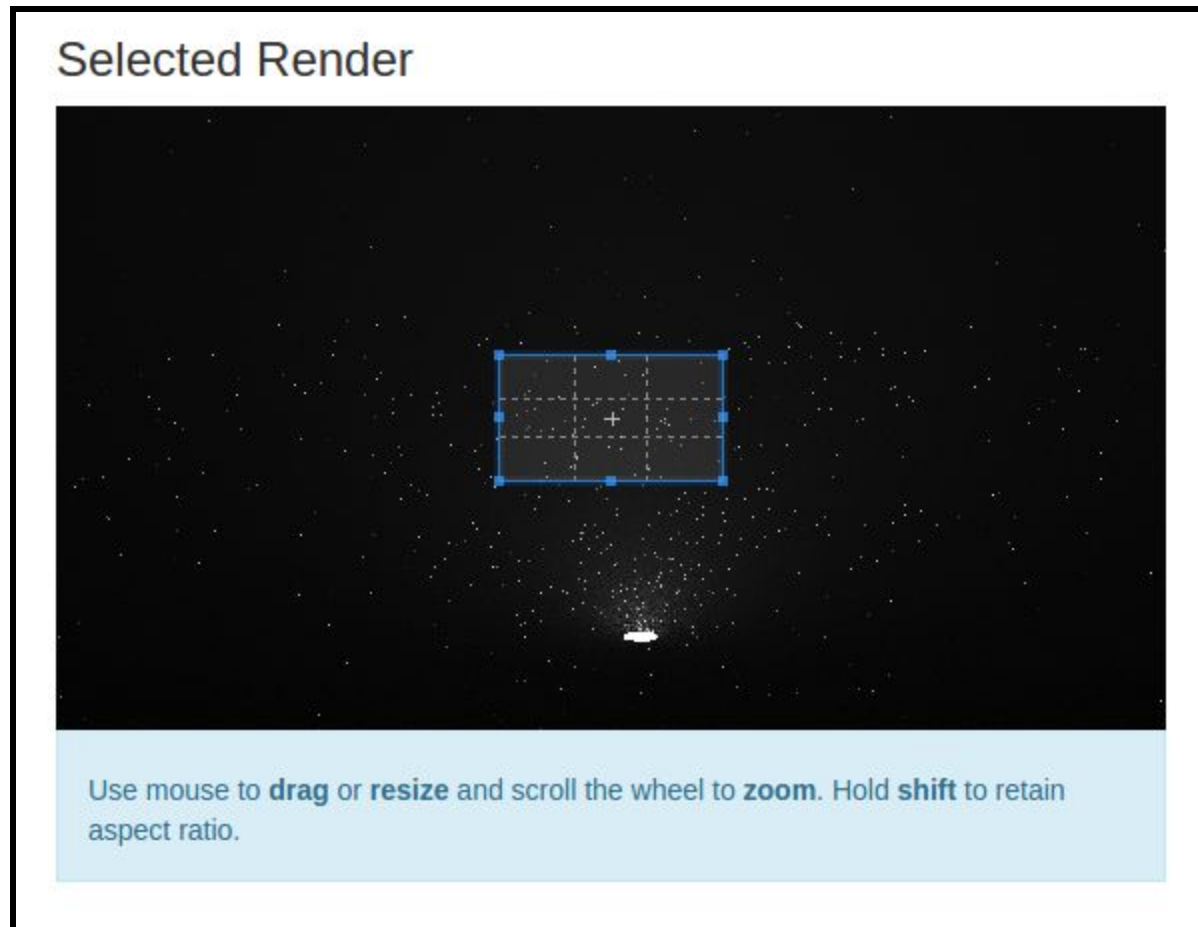
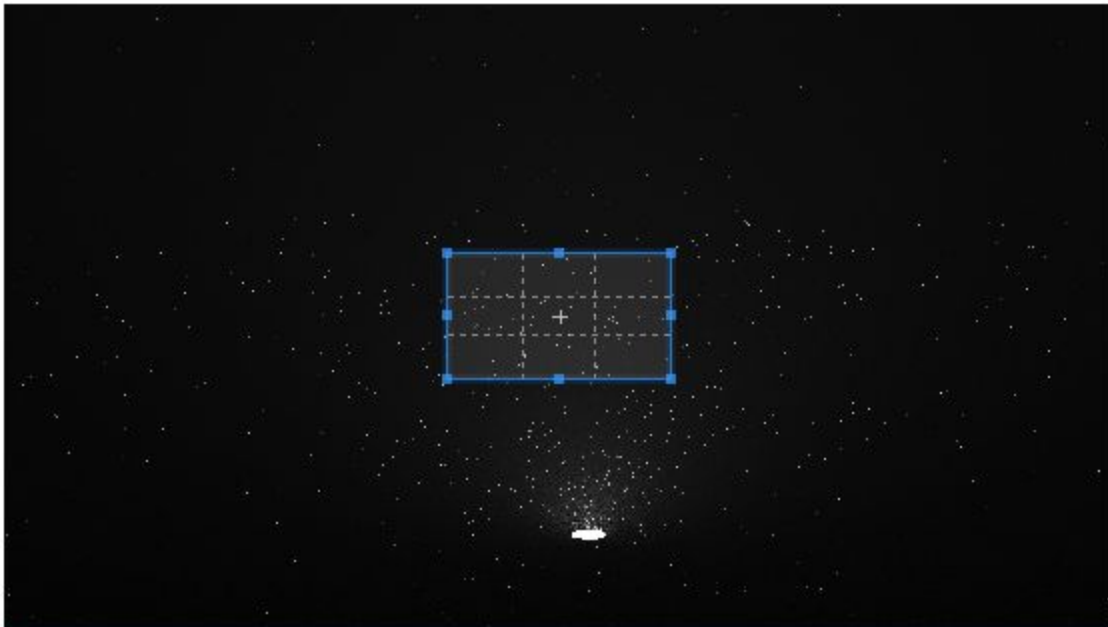


Figure 02: Selected render, contain the actual size of the image selected by the user.

2. Selecting region of Interest

The user can use the mouse to resize the blue box shown in the selected render in order to define the region that he or she wants to analyse.

Selected Render



Use mouse to **drag** or **resize** and scroll the wheel to **zoom**. Hold **shift** to retain aspect ratio.

Figure 03: Selected render showing mouse drag.

As shown in figure 03 how you can use the mouse to resize the region of interest. By putting the mouse at the corner it appearance will change to be similar. Then you can hold it and drag. This will cause the rectangle to shrink or enlarge.

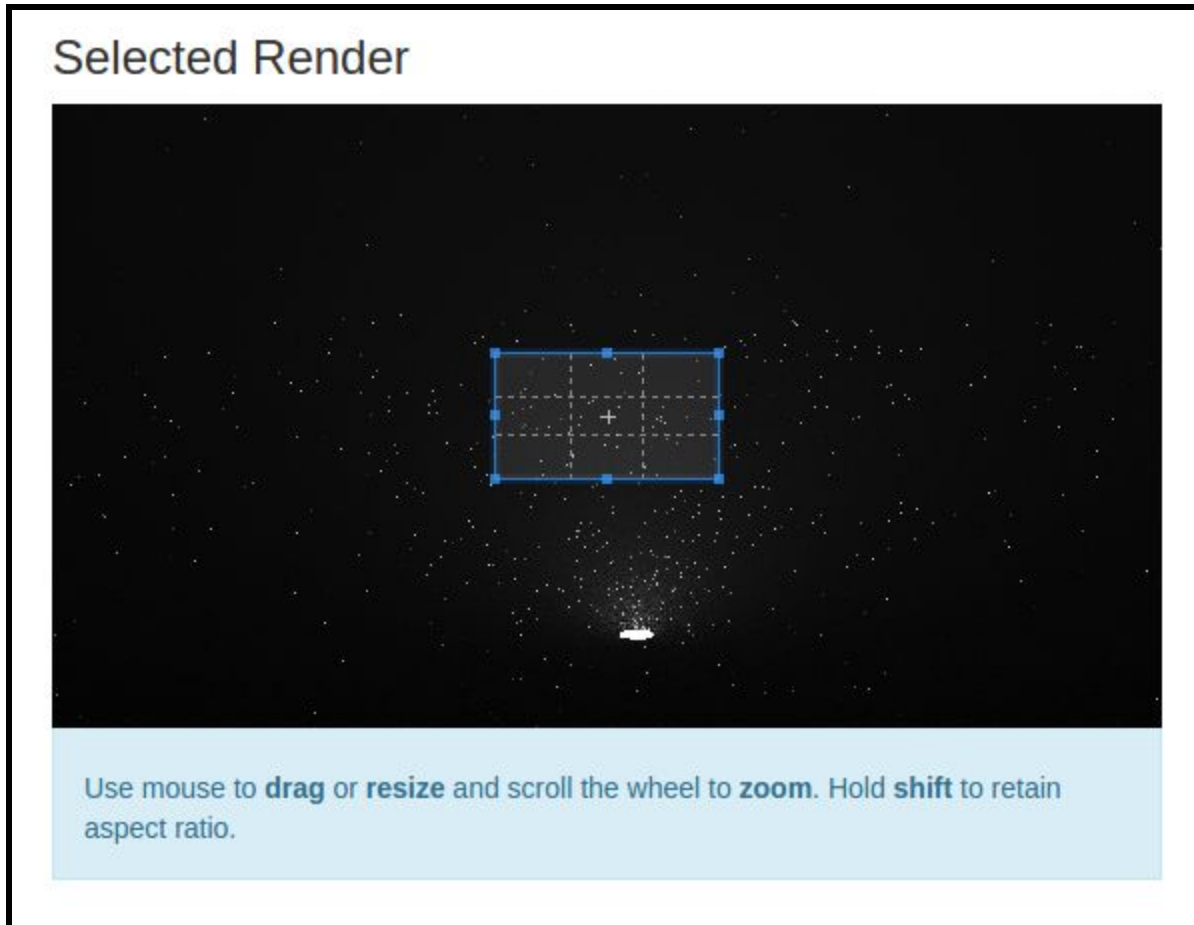


Figure 05: selected render showing how to move the entire square to a different location.

In order to move the rectangle defining the select region to a different part of the image loaded, you can click on the rectangle and hold. Now you can drag it to any location of interest in the image. While you dragging the rectangle you will notice that the selected region window is changing.

3. Freezing selection

Once you have selected the region of interest you must freeze it in order to propagate the data down and start analysing the different samples.

Freeze Selection

You must freeze your selection in order to select outliers.

Left	Right	Width	Height
512px	288px	256px	144px

Figure 06: Window showing information about region we are about select.

Unfreeze Selection

92 outlier pixels were automatically detected. Adjust the preciseness below.

5.1

Set

Most Precise

Least Precise

Left	Right	Width	Height
512px	288px	256px	144px

Figure 07: Window showing information about region selected

To freeze the selected region you must click of the “Freeze Selection” Button as shown in the figure 06. Once we click on the “Freeze Selection” the with change to be similar to the window shown in figure 07.

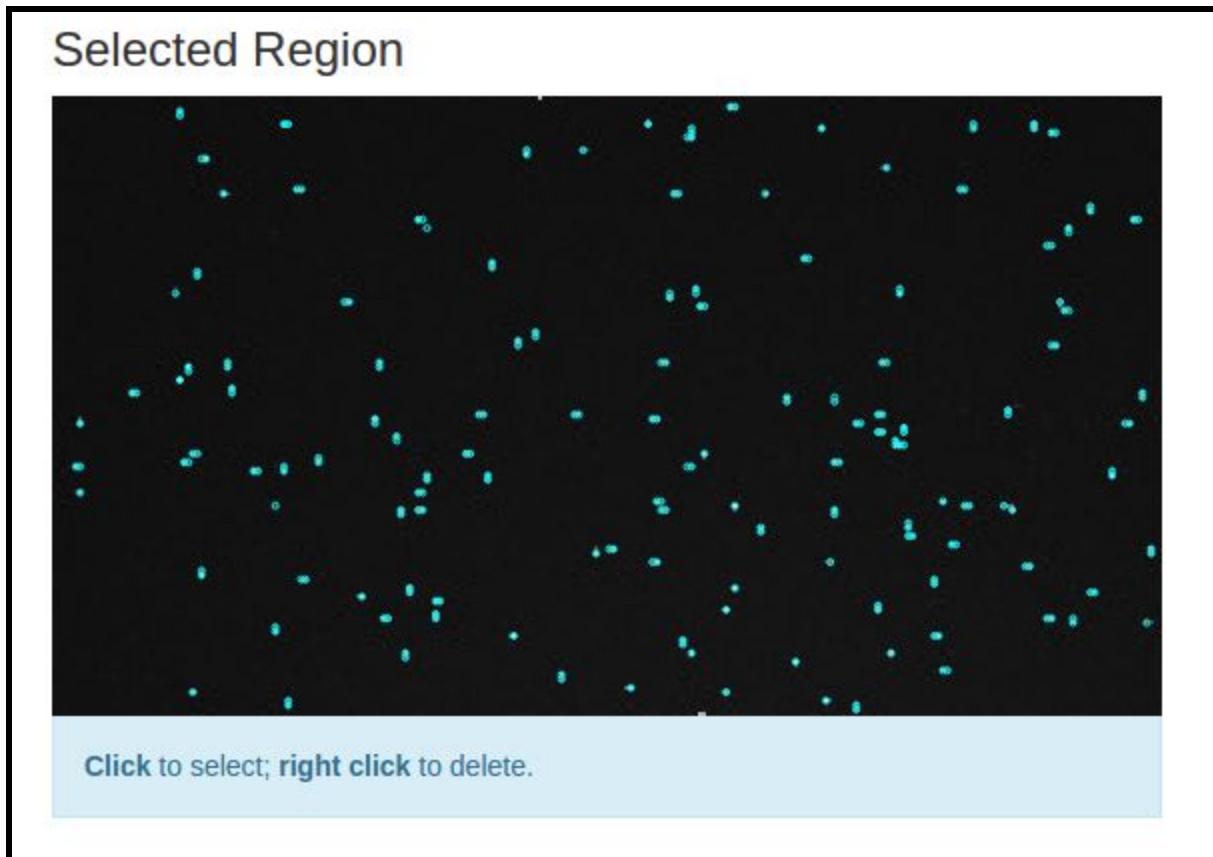


Figure 07: Automatic identification of fireflies.

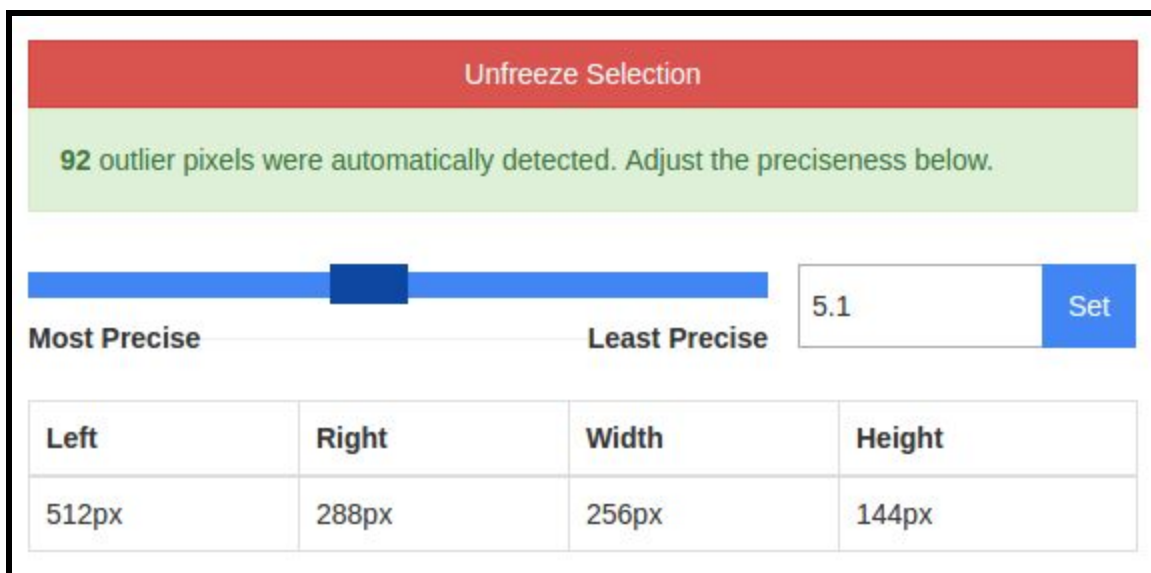


Figure 9: fireflies identification algorithm precision.

Once we freeze the selection few things happens. First, different “fireflies” are automatically identified and circled as shown in figure 08. Second, a horizontal scroll bar appears below the button. The scroll bar can be used to adjust the precision of the fireflies identification algorithm as shown in figure 09.

4. Download reconstructed image

Once you are done exploring and editing the loaded image you can download the result image by clicking “Download Reconstructed Image.”



6. Implementation

6.1. Iteration I

Iteration I or Version I of our application has two component to it. A design phase and a data collection phase. Both phases can be executed in parallel to accelerate the process.

Design phase consisted of implementing the basic visualization interface.

- The list of images from which the user can choose.
- Display of selected image with brush allowing the user to select a region to analyze.
- Display of selected region.

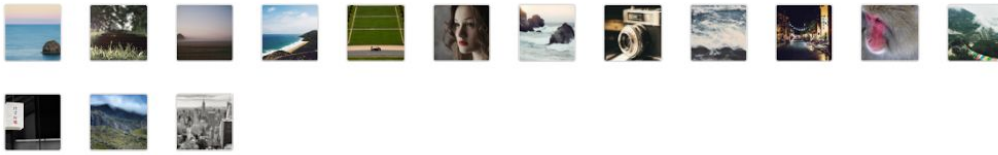
Data Collection phase consisted of Collecting the data for a single pixel so we can incorporate it in Version II and have a good understanding of the data in order to adapt our tool better manage the data.

The deadline for Version I according to our milestone was November 13. We were able to meet the design goals for this version. However, we were behind on the data assimilation process.

The image below shows the state of our version I iteration. Not only were we able to meet the deadline for design, but we were also able to implement a little more. As you can see we had the window to manipulate the selected region data partially implemented. The implementation of this section uses Bootstrap and D3.

We were able to collect the data but we had some road blocks. We had some miscommunication on data format. What is the actual json file format needed to the display. The server in order to enable the communication between our application and the data was not set up. This delayed us significantly.

Available Renders



Render



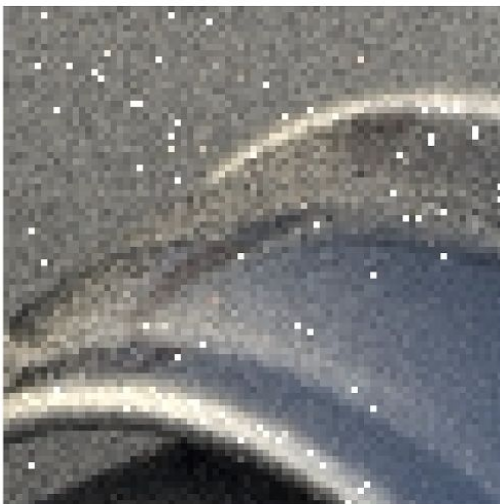
The cropbox is limited to 80x80 pixels which should be sufficient for most analyses. When you are satisfied with your selection, you can proceed with detection.

Zoom

Drag

Resize

Analyzer Region



Select Pixel

Analyze Selection

Remove Selection

Begin Detection

This step is **necessary**. Once the detection process is initiated, your cropbox will be analyzed for fireflies. Furthermore you can manually select fireflies.

Crop Box Info

Left	Top	Width	Height
500px	283px	80px	80px

Download the cropped region

Version I:

After Version one we met the TA VINITHA YASKI and as for feedback and recommendation. The few things that came out that meeting are :

- Adhere to your plan or schedule
- Integrate actual data to your application
- Include some sort of user guide or/and video

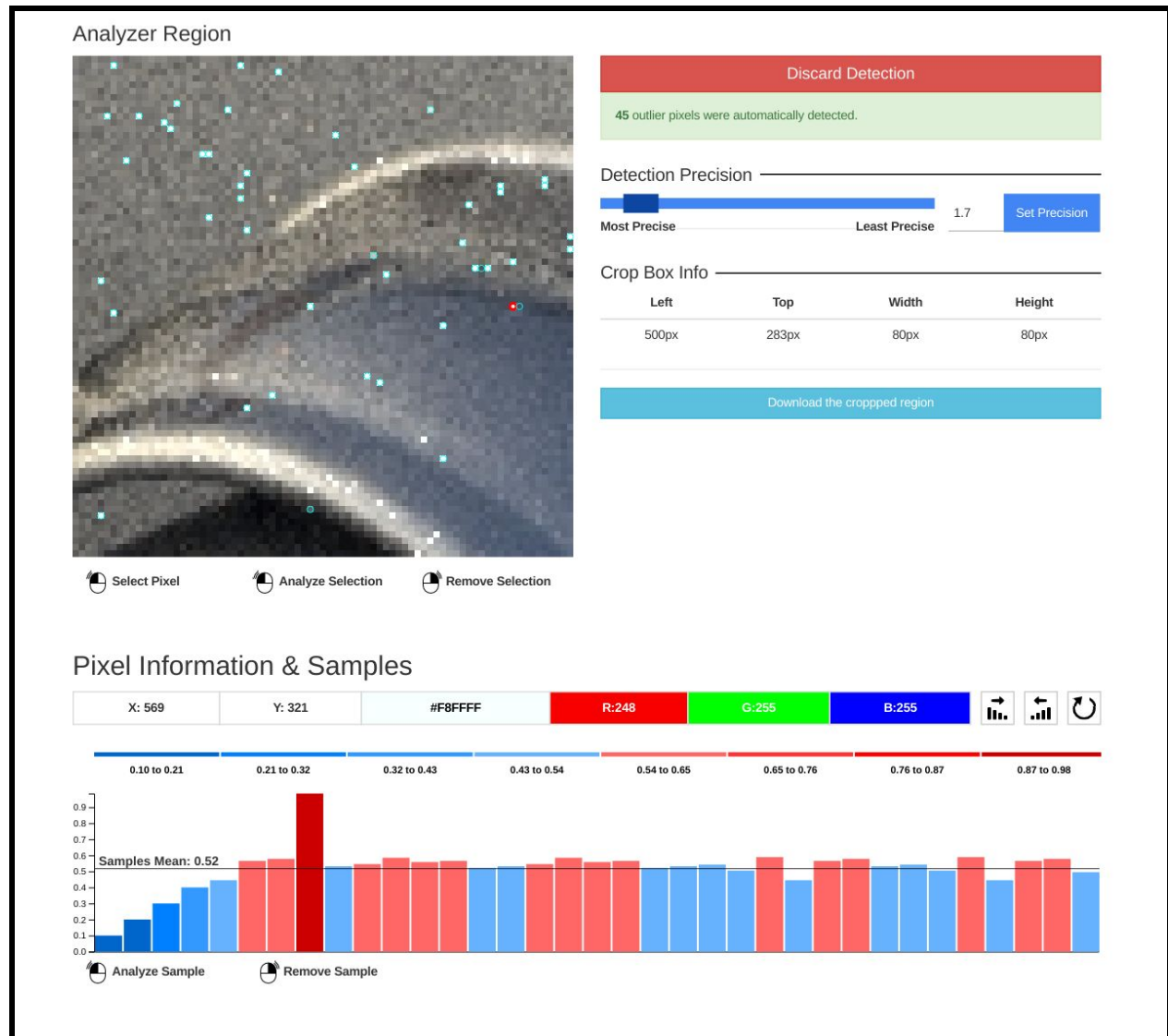
6.2. Iteration II

In this Second iteration as in the first one focuses on two phases: Data collection and visualization design. Ideally the data collection should have been completed at this stage.

In the design phase we aimed to implement :

- Detection toolbox
The detection toolbox allows the automatically detect outliers among the pixels in the region selected. The sensitivity of the algorithm can be adjusted using the horizontal scroll bar. Once selected the outliers will circles as shown in the picture bellow. The tool box shows different information about the region selected.
- Bar chart visualization of selected pixel's samples
The User can click on pixel to analyze the pixel samples. Once a pixel is clicked it will be circle in red and the associated samples to that pixel will be displayed below. This is where we start using the data associated to the picture.

In the data assimilation phase we aimed for collecting the data of one actual data of image and be able to request the data through the server. They were more roadblocks here. The data was collected but the integration and the formating of of the data with application was not ready.



Version II

We again met with TA VINITHA YASKI for more feedback and recommendations. She suggested that :

- Integrate the actual data
- Move histogram legend

The picture above include has this problem already fixed. Previously we had the histogram below the bar chart and it was confusing.

- Include smooth transitions
- Mention maximum size of selected region

6.3. Iteration III

This is our final Version. In this last part we aim to incorporate the feedback we received, display the paths associated with a selected sample, and reconstruct an image after the we remove a specific sample.

This iteration consumed the majority of our time . We implemented the image reconstruction two the user can choose specific pixel sand fix them. The used the example in our homework to display the different paths that make up a sample. In addition we opted for displaying selected path as we could have thousand or even millions of paths. We also use a very small size picture to have reasonable bound on the data that is handle by the browser.

Our final version incorporates the features below:

- User can remove sample from histogram
- User can display paths of specific sample. User can see R, G, B of values of the selected paths and the elements that make up the path.
- User can download the the processed picture

6.4. Tools

In the implement we used various tools. For the design and the front end of the application we used D3, Bootstrap, JQuery3.1.1 and Cropper.js. First, We use Bootstrap to make our application compatible on mobile and desktop browser. Second, D3 is used to dynamically manipulate the different windows displayed on the application's user interface. Third, JQuery is serves as tools to make queries on our data and retrieve information we need. Fourth, Cropper is used as tool to cropped the region of interest of the chosen image.

7. EVALUATION

The Goal of this project was to build a web base a tool to analyse visualize bidirectional path tracer data. This aim enable easy identification of problematic samples and remove them. We were able to build such tool with a given set of images. Multiple learning outcomes arises from this project.

7.1. Data Visualization

In our case the conceptual design and visualization seemed very simple to us that we did not anticipate any difficulties as far as display meaningful data to the user. We were only concerned about the data size that will be transmitted as the only problem. However as we progressed in the project it was a challenge to identify what paths to show the user. This is a problem because with path tracing you can have thousands even millions of paths. Thus we cannot show all of them to the user. In order to address this we decided to process the data and select relevant paths to display of the browser. In order to display all the paths and samples and different visualization method must used.

7.2. Team Work

Time and time again we hear about how important it is to have coherent team that works base on the goals with a clear milestone. Clearly outlining the goals and specifying the different tasks were very useful for progress. However, we fell behind on data collection which affected our testing and various aspect of the project. For instance how much data the browser can handle, how we can improve our implementation to effectively solve the problems at hand.