Process Book

Basic Info

The project title:
Post-processing Image Statistics for Bidirectional Path Tracing

Git repository:
https://github.com/lediaev/vis2016

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

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.

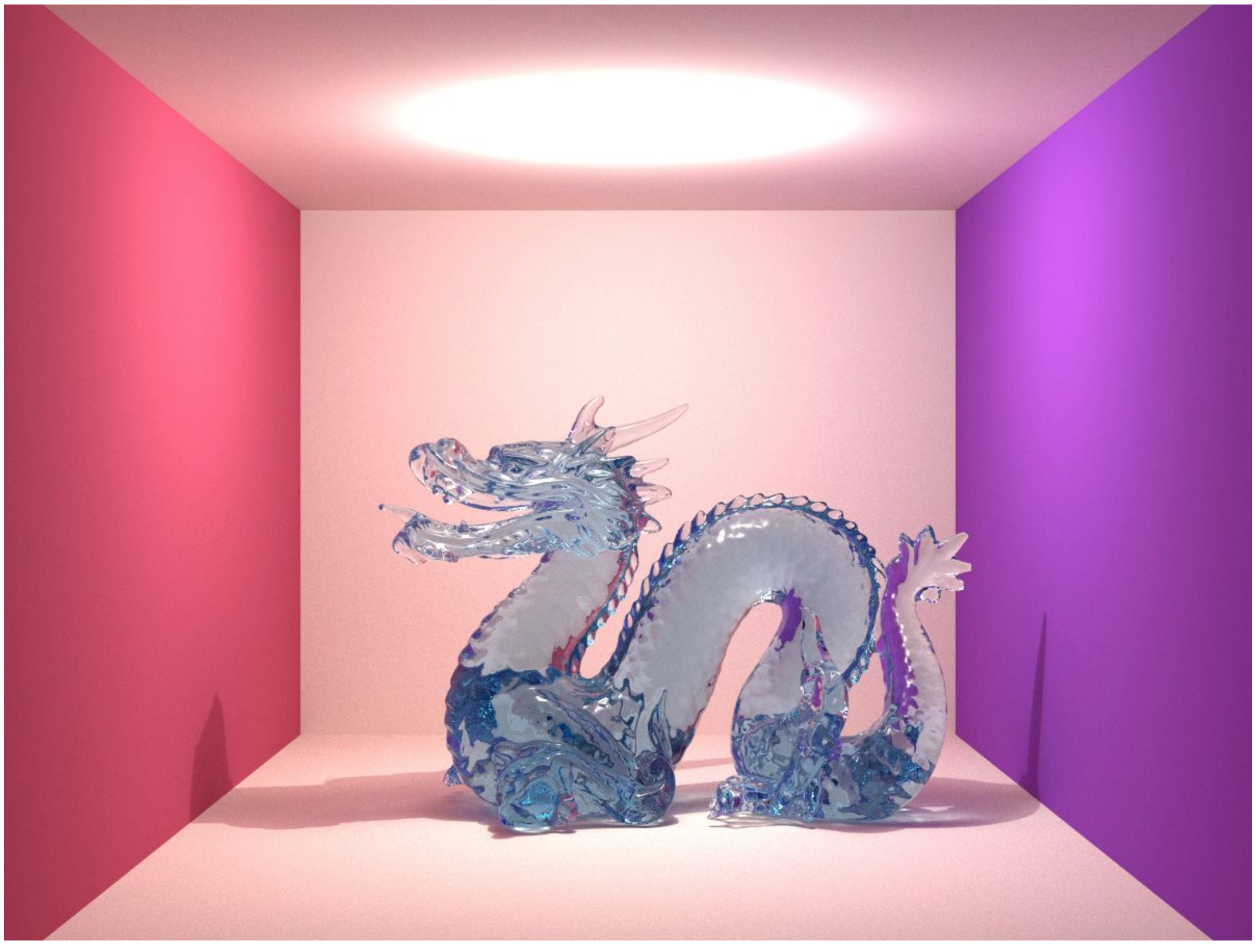


Figure 1: An example image rendered with path tracing.

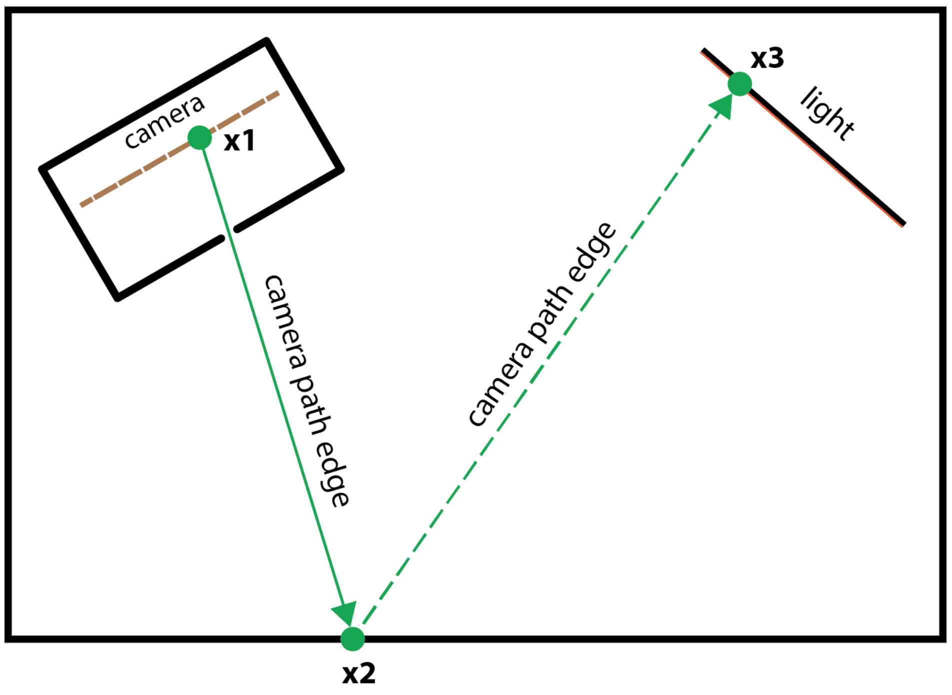


Figure 2: A diagram showing a complete path with two edges.
Although unidirectional path tracing will produce a converged image, it may take a very long time. Bidirectional path tracing attempts to reduce the amount of time needed to render a finished image. It does this by combining several different 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.

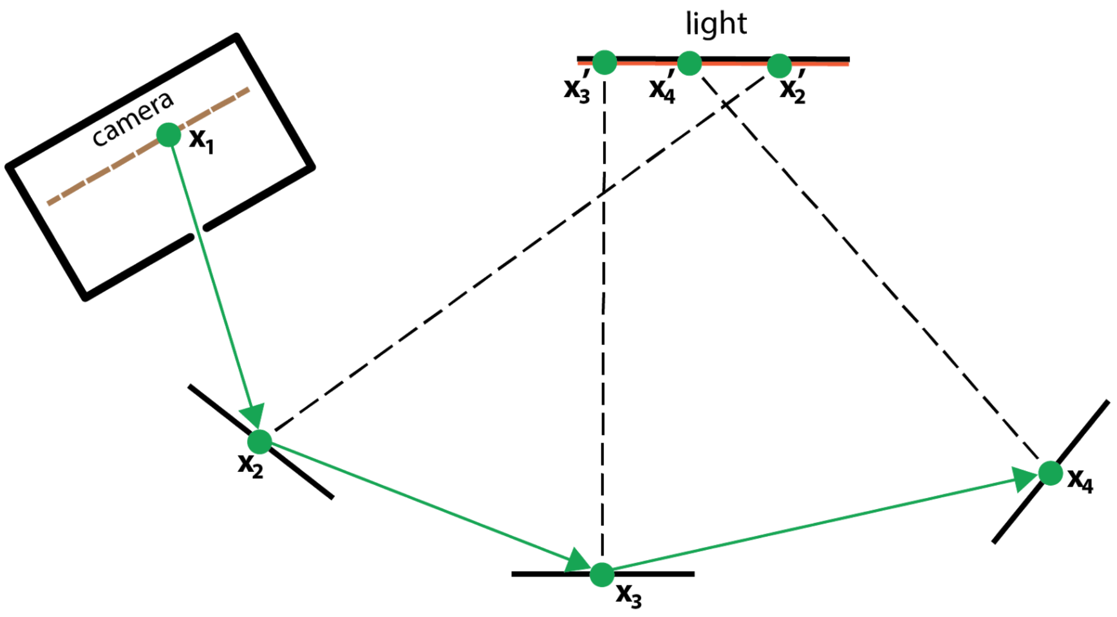


Figure 3: A diagram showing the process of each path vertex sampling a light.

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.

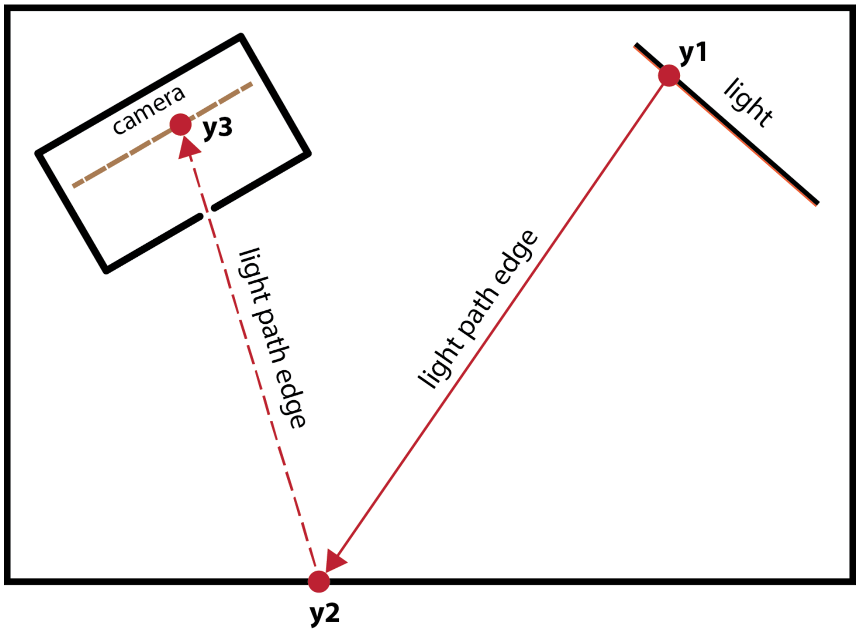


Figure 4: An example of a path generated by light tracing.

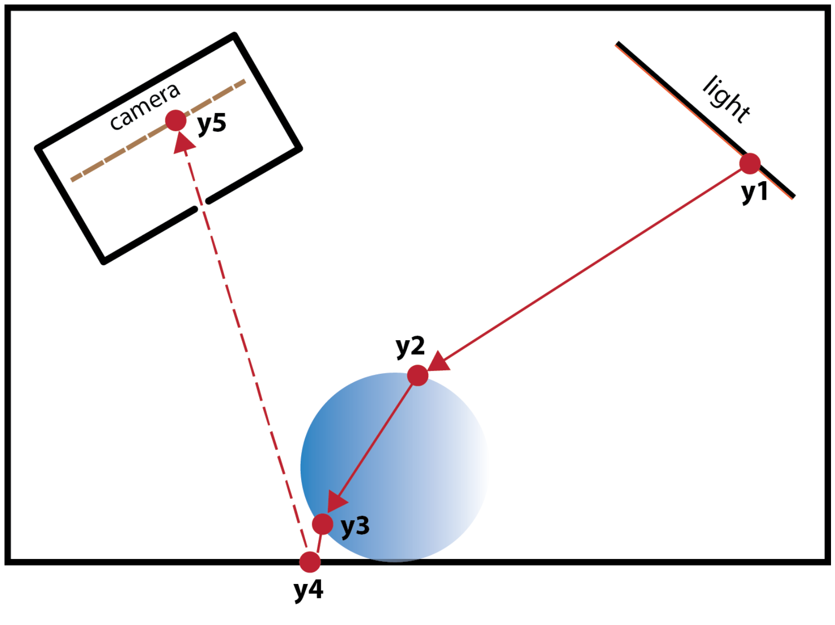


Figure 5: A light path showing a caustic beneath a glass sphere at vertex y4.

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.

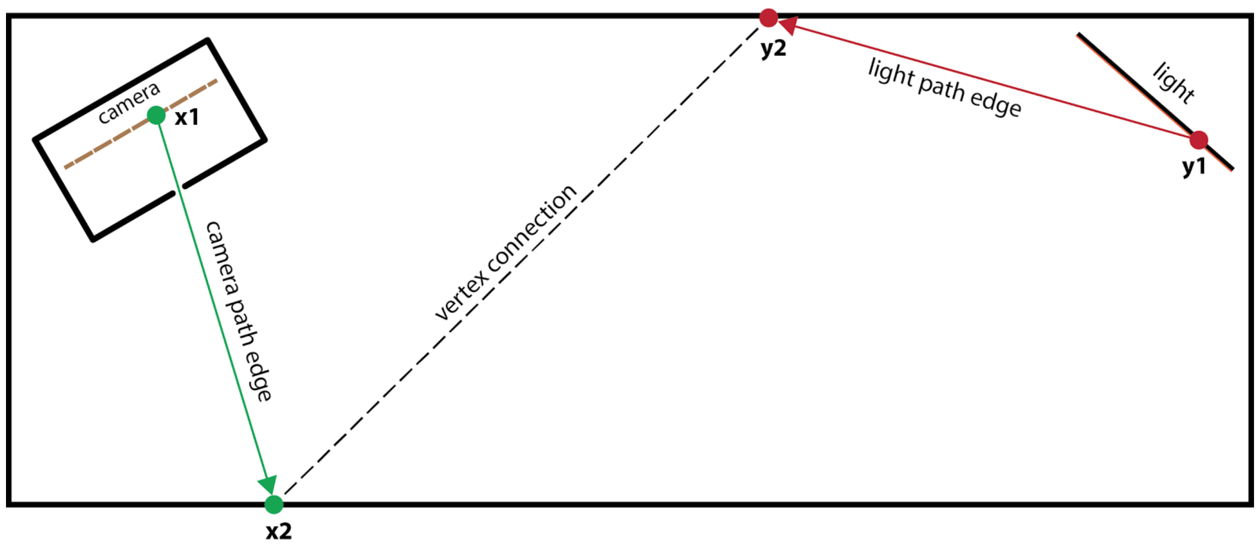


Figure 6: A bidirectional path with three edges, combining a camera path with a light path.

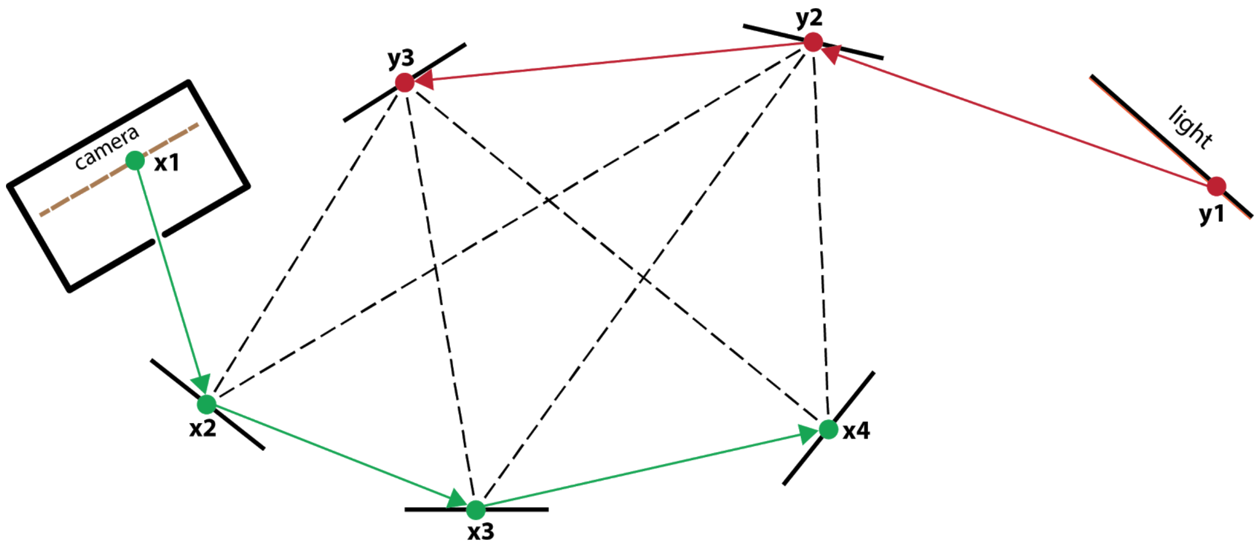


Figure 7: In bidirectional path tracing a collection of new complete paths is created by connecting different vertices from a camera path and a light path.

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

The main purpose of this visualization project is to provide an overview of the rendering process and make it easy to explore how the image was created and possibly identify problematic pixel samples. This allows the artist/developer to understand and determine the components of the scene that result in low-probability paths that can lead to noise in the final image. 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*8 = *16* bytes total per sample
● **Path**:	5 Total	5*8 = *40* bytes total per path
● **Vertex**:	5 Total	5*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*

(#ofSamples * 16) + (#ofSamples * #ofPaths * 40) +
(#ofSamples * #ofPaths * #ofVertices * 40)
=
#ofSamples * (16 + **#ofPaths** * (40 + (**#ofVertices** * 40))
#ofSamples * (16 + **#ofPaths** * 40 * (1 + **#ofVertices**)

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 bytes 241.6 kilobytes

100 samples per pixel, **10** paths per sample, **10** vertices per path
100 * (*16* + **10** * *40* * (1 + **10**)) = 441600 bytes 441.6 kilobytes

200 samples per pixel, **10** paths per sample, **10** vertices per path
200* (*16* + **10** * *40* * (1 + **10**)) = 883200 bytes 883.2 kilobytes

500 samples per pixel, **10** paths per sample, **10** vertices per path
500* (*16* + **10** * *40* * (1 + **10**)) = 2208000 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 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

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.

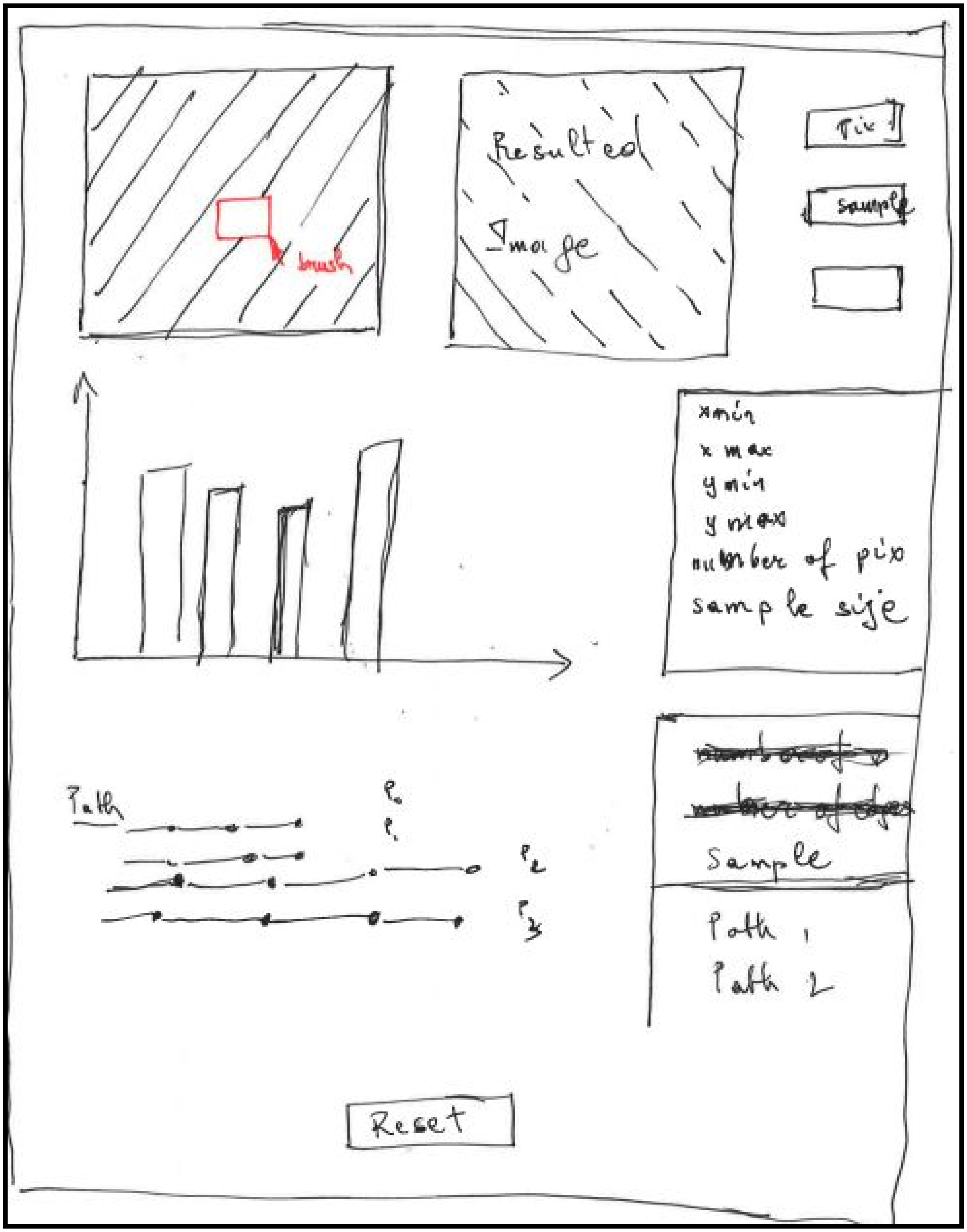


Figure 8: Sketch 1.

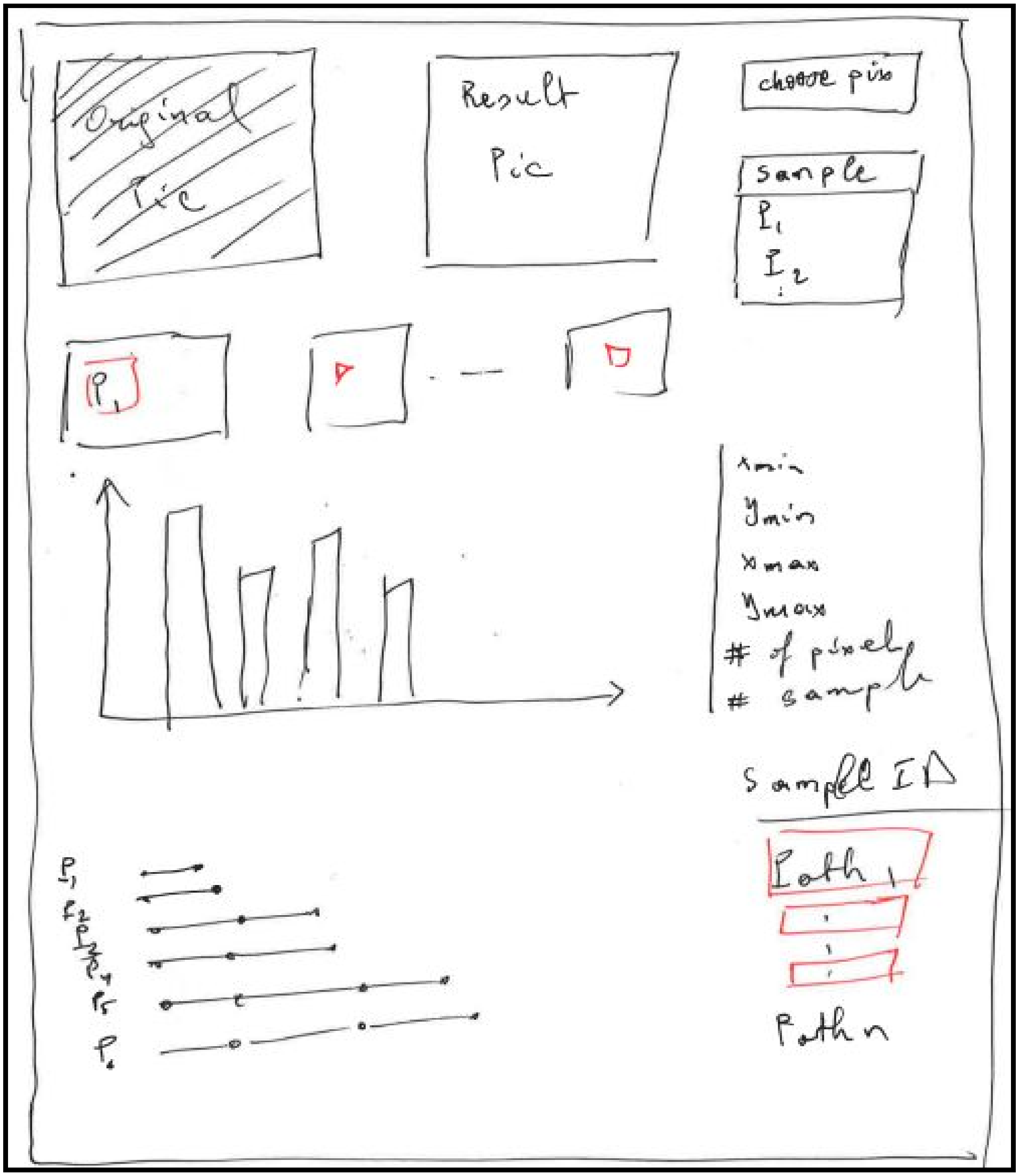


Figure 9: Sketch 2.

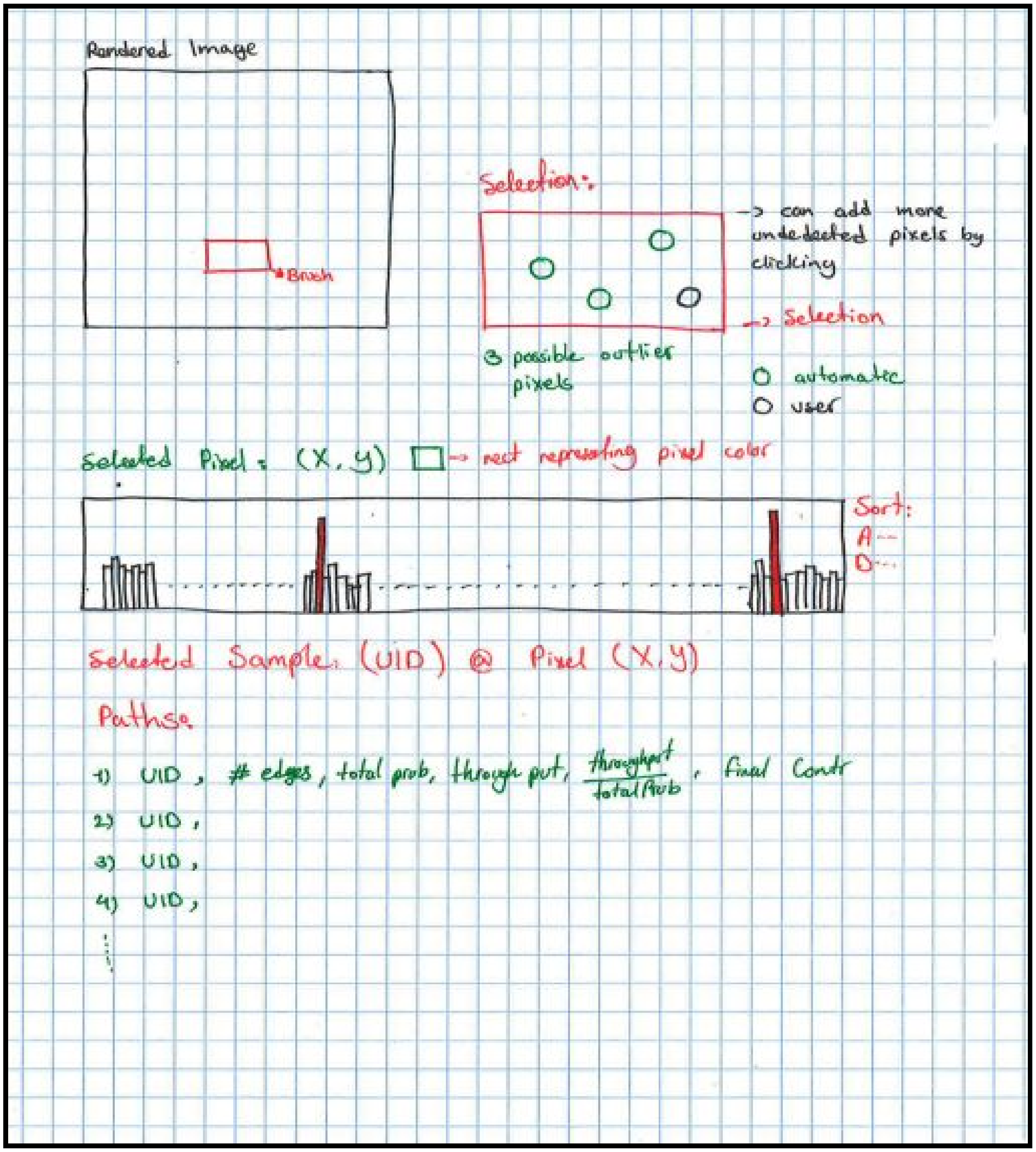

Rendered Image
Brush
Selection:
-> can add more undetected pixels by clicking
-> Selection
3 possible outlier pixels
O automatic
O user
Selected Pixel = (X, Y) □ -> rect representing pixel color
Sort:
A--
D--
Selected Sample: (UID) @ Pixel (X, Y)
Paths:
1) UID, # edges, total prob, through put, throughput / total Prob, final Contr
2) UID,
3) UID,
4) UID,

Figure 10 : Sketch 3.

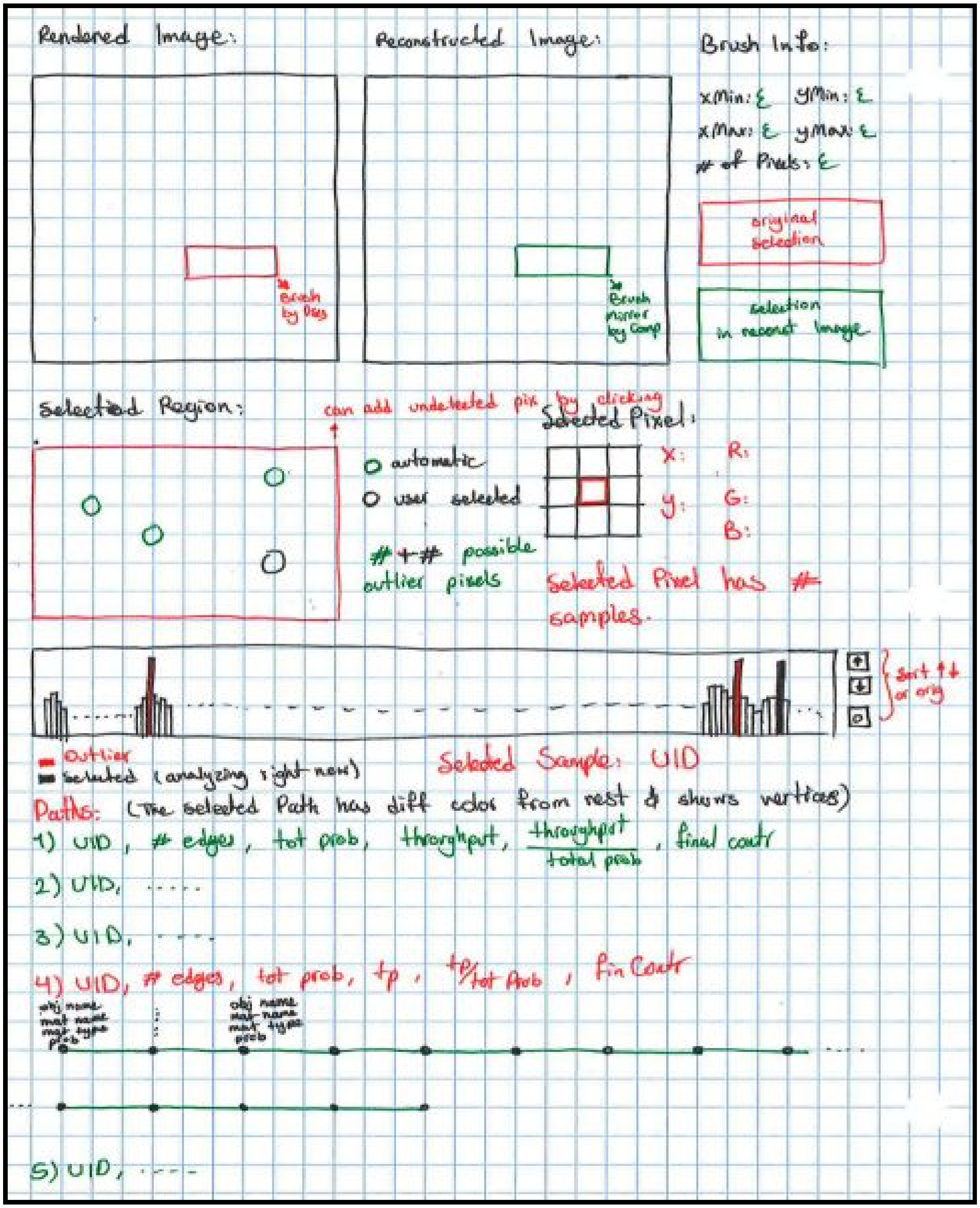

Rendered Image:
Reconstructed Image:
Brush Info:
xMin: ε yMin: ε
xMax: ε yMax: ε
# of Pixels: ε
original selection
selection in reconst image
Brush by User
Brush mirror by Comp
Selected Region:
can add undetected pix by clicking
Selected Pixel:
automatic
user selected
#+# possible outlier pixels
X:
y:
R:
G:
B:
Selected Pixel has # samples.
sort ↑↓ or orig
outlier
selected (analyzing right now)
Selected Sample: UID
Paths: (The selected Path has diff color from rest & shows vertices)
1) UID, # edges, tot prob, throughput, throughput / total prob, final contr
2) UID, ...
3) UID, ...
4) UID, # edges, tot prob, tp, tp/tot Prob, fin Contr
obj name
mat name
mat type
prob
obj name
mat name
mat type
prob
5) UID, ...

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

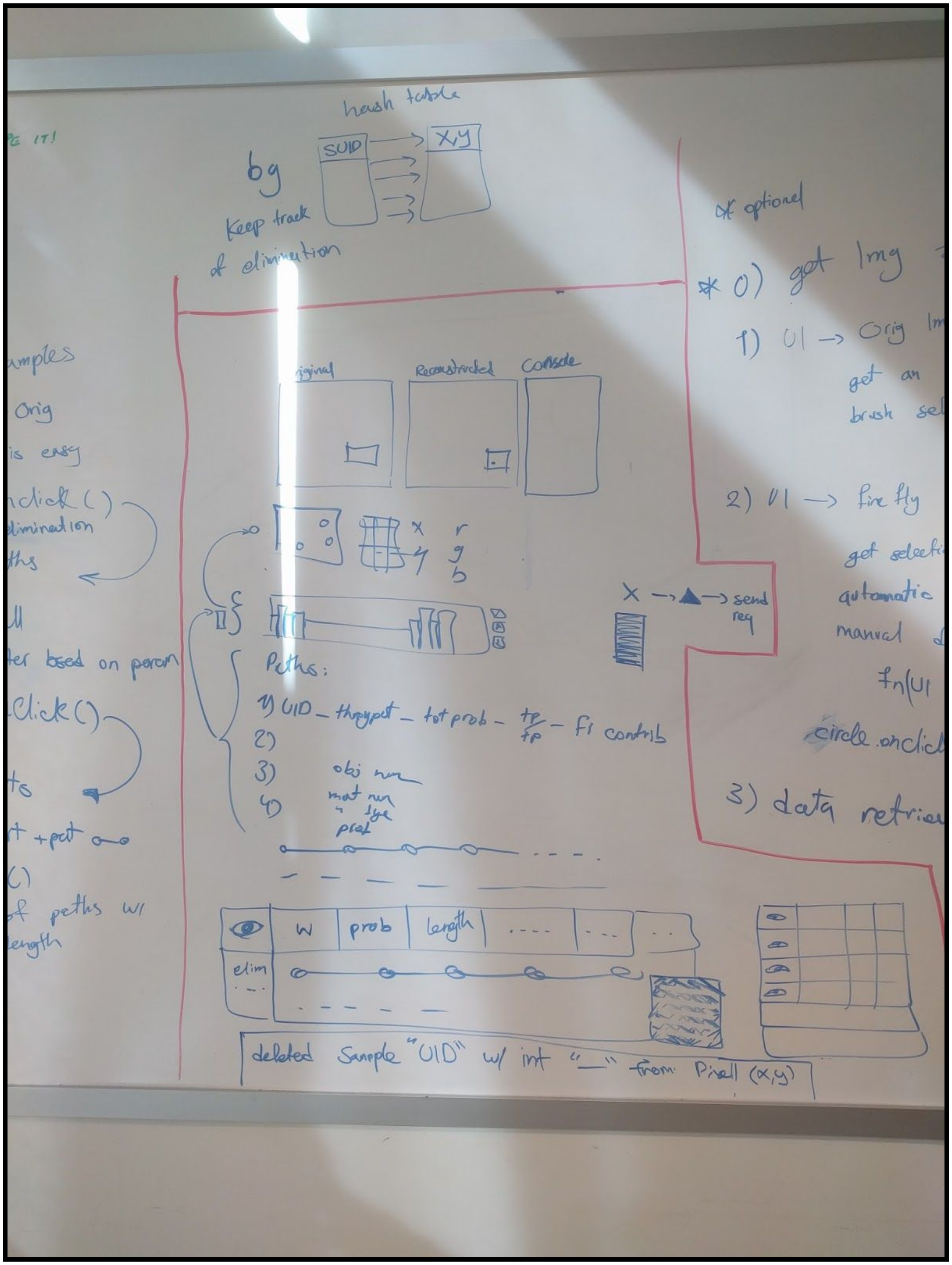
2. **UI → Firefly Detect View**
   - Get brush region
   - Detection

        - Automatic
        - User-defined
    - Pixel Mapping
        - UI[x, y] → Image[x, y] → 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


PE IT!
hash table
SUID
X,Y
bg
Keep track
of elimination
amples
Orig
is easy
nclick ()
elimination
ths
ll
ter based on param
Click ()
ts
rt +pat
()
of paths w/
length
original
Reconstructed
Console
x
y
r
g
b
II
Paths:
1) UID _ thruput _ tot prob - tp/fp - f1 contrib
2)
3) obj num
4) mat num
" dye
prob
X → ▲ → send req
W
prob
length
elim
deleted Sample "UID" w/ int "_" from Pixel (x,y)
* optional
* 0) get Img
1) UI → Orig Im
get an
brush sel
2) UI → fire fly
get selecti
automatic
manual
fn(UI
circle.onclick
3) data retriev