PROJ Interim Report - Revision Aid for Veterinary Students

Murray Crichton - s1232200
2016 UoE

Abstract:
This report details and discusses the topic of creating a veterinary revision aid smartphone app, a project proposed by Amy Livens, a student here at the University of Edinburgh. Livens provided the source material, as well as guidance on her vision for the final app. The goal was to convert a set of anatomical diagrams, contained in a bulky physical form with little hope for replication, to an easily distributable smartphone app, while maintaining the format of the diagram set, which was designed to ease revision of complex anatomy and physiology.

The Story So Far:

The goal of the project, proposed by veterinary student Amy Livens, was to convert a large set of handmade anatomical diagrams into an app, suitable for distribution to other students in order to aid revision. This diagram set, created by Livens, was provided and scanned in, totalling over one-hundred and thirty hand-drawn diagrams (this was in fact a subset of the diagrams Livens had created, only detailing the general anatomy of the dog). This diagram set (alongside the others Livens had created) aroused great interest in her fellow students, with requests to borrow the voluminous binder containing the set becoming commonplace. The set's unique feature, Livens' usage of layered acetate sheets to allow diagrams to be built up gradually, easing the revision process, was seemingly of great desire to the students.

The issue here was that demand far outweighed supply, as replicating such a set of diagrams would be extremely time-consuming, incredibly dull, and relatively expensive in terms of material cost. A solution, therefore, would be to find some method to quickly create a great many copies of the set, with minimal expense. Creating additional physical copies, likely by scanning in and printing out the diagrams would have still incurred a hefty material cost, the acetate sheets used not being particularly cheap. Another issue, the physical size and weight of the final set, would not be solved, either. An app, capable of running on a smartphone or tablet (specifically targeting the Android platform) was therefore proposed, hopefully circumventing the above issues, while providing something as close as possible to the original physical copy.

With the proposal given, a meeting was organized between Livens and I, to discuss further details and hand over the source material, this fabled folder of dog diagrams that had been in such heavy demand. The requirements seemed simple enough, and we parted ways, Livens returning to her hectic schedule in the veterinary school, and me returning to my flat to begin a marathon scanning session, throwing on a brainless action film to ease the boredom of the process.

I initially began to experiment with methods of app creation. The first, and most obvious option, was to simply use the official Android Studio[1], used to create native apps capable of running on a predefined set of Android versions. This, however, involved learning the seemingly bewilderingly complex process of creating a Java application capable of running on an Android device. Now, while I will admit that Java is an excellent language for certain situations, I strongly dislike the language, and while I can usually tolerate it for other projects, the added layer of complexity (and the generally lackluster documentation of biblical proportions) brought by this being and Android app had me quickly scurrying for another solution.

While searching for common alternatives, I came across, firstly, a method for implementing a "native" Android app using the built-in "WebView widget"[2] which, in essence, launches an instance of the smartphone's (or tablet's, presumably) basic browser, which would generally be either the Android Browser (no longer supported) or mobile Google Chrome. This instanced browser, running within the app itself, could then have JavaScript enabled, and be directed to a local web page, and would then operate as one might expect, allowing for all normal web-browsing capabilities. While investigating this method, I developed a basic web application, using HTML, CSS, and JavaScript, and found that, while having some rather obnoxious input delay (common to all smartphone browsers, as later became clear), the web application functioned well, and seemed to provide all I would wish for, certainly capable of fulfilling the task at hand. Having had far more experience developing web applications than Java applications (my zero prior experience creating smartphone apps notwithstanding) this seemed a convenient solution. While my experience using JavaScript was rather limited before embarking on this project, the language, I find, is far more enjoyable to use for basic tasks compared to Java, the stifling syntactical requirements of which I find deeply enraging.

As I continued development, I became more dissatisfied with this method, however. The native Android app wrapping the lightweight web app added a lot of bloat, and an unnecessary layer of technologies that needed wrestled into submission to produce the desired final product. I then came across what turned out to be a far cleaner solution - Adobe PhoneGap[]. Advertised as a free, open source, multi-platform method to allow web apps to be deployed as seemingly-native smartphone apps, it appeared to be the ideal solution. I ported over my existing implementation (butchering it slightly in the process) and with some surprisingly minor configuration tweaking had a working PhoneGap app, running happily on my aging smartphone.

The input delay problem, which I had falsely attributed to the clunky nature of my previous implementation, persisted, however. A bit of research, and a brief dive down the rabbit-hole of smartphone browser technology, revealed that this delay I was perceiving was, in fact, an intentional inclusion to allow pinch-zooming gestures to function. The idea here being that the delay (seemingly 300 msecs) would allow a touchscreen device to discern between simple taps, and tap-holds, used to zoom in and out on poorly-mobile-optimized web pages. This feature is present in the native Android Browser (at least the version present on my smartphone, although I believe it to be a universal trait) and therefore carried over to both of my implementations, PhoneGap apps using the same basic principle (effectively running an instance of the base browser) by default. This behaviour has been killed off in more modern versions of Google Chrome[3], but remains in older devices and Android versions, such as mine. Conveniently, this being a common problem, a solution has been created, the JavaScript library "FastClick."[4] Using this, my PhoneGap app, built almost purely as a web app, ran as smoothly as any native app I've used, a seemingly solid alternative.

With a reasonable basic UI nailed down and running smoothly on my smartphone, I turned my attention to generating the content that would fill the app. During The aforementioned scanning marathon I merely scanned the final layered version of each page of the folder, and not the individual layers. This now seemed far from adequate, and with a grim resolve (and no film to distract me) I set about re-scanning the folder, the full complement of one-hundred and thirty-three sheets, awkwardly and inconsistently taped together to further hinder my efforts. The original scanning was, thankfully perhaps, not a wasted effort, still useful for pulling together the layers in the correct position, given the somewhat unavoidable flaws created when attempting to scan so many pages in exactly the same orientation.

With the scanning work done, and the folder returned to its owner, I set about finding a way to convert the high-resolution scanned images of the diagrams (complete with fingerprints, stray hairs, and whatever other detritus the folder had collected throughout its storied career) into a more phone-ready format. Initially, I toyed with the idea of hand-tracing each diagram using a photo-editing tool, allowing smooth, clean lines to be created with good colour distinction. This would have been a valid solution, perhaps, were I on a Computer Arts degree, and had a couple of hundred hours to spare meticulously re-creating the diagram set more or less from scratch. Another issue here, was labelling the diagrams. The original diagrams[Appendix 1] were, of course, heavily labelled with important information and complex names for various parts of the dog anatomy. Having text be part of an image, with the exception of a few cases (logos being a prime example, especially in web design), would fall somewhere near the “Cardinal Sin” end of the Computer Science “what not to do” scale. This leads to a problem where labelling the diagrams should be done with a text overlay, but I would lack a convenient method to properly place labels in their correct location, for usage in the final app.

The first problem, converting and cleaning the original images, I solved by creating a separate Python application, using the image processing library “OpenCV.”[5] The aim of this application was to allow rapid processing (preferably automated) of large numbers of these scans, to prepare them for further usage in the most efficient manner possible. The combination of technologies here was chosen for a few reasons, first and foremost being my familiarity with Python over other languages - I tend to favour the language when possible for individual tasks, and have previously contributed to a relatively large group Python project. Using OpenCV here seemed a natural choice, it being a renowned Python library for image processing, with a sensible enough interface that I happen to have had some experience with in the past. As the name would suggest, the library is open-source; and perhaps less obviously (although not by much) is actually “written in optimized C/C++” with a set of Python (other language options being available) bindings provided when installing. As far as I can tell, this C/C++ basis for the library ensures the fastest possible execution, and in fact, the purpose of the library itself is to provide tools for real-time image processing, where speed is obviously key. While the application I would be writing would not necessarily be what one would associate with real-time image processing (using video feeds as input being the common case here) it would still require sufficiently fast image processing to ensure a smooth user experience. Despite this, the editor application turned out to be too sluggish and clunky to provide much of an enjoyable user experience, but this was not the primary goal (or much of a secondary goal, for that matter) and it works well enough for the task.

The goal of processing large numbers of images as quickly as possible was achieved to a reasonable extent, however. The application I created to do this was similar, in some ways, to other image-editing programs, but with only the specific subset of features I would require implemented. This (rather greatly, given the amount of source material to process) streamlined the editing of the images, but did not bear a huge advantage over using a traditional image-editing application to do the same job. The key feature, here, then was in being to group images into titled “pages” (analogous to a set of sleeved acetate sheets with a paper backing in the folder), layering individual diagram “slides” (analogous to a single acetate or paper sheet) with their relevant annotations and associated information to create a finished page. This entire package, the virtual folder of diagrams, could then be exported with the push of a button, and was then used to provide the content to fill the smartphone application.

To process the images and prepare them for exporting, various tools (the aforementioned subset of standard image-editing tools, although perhaps this is misleading - these tools were more specialized, built only for the use case at hand) were created using the OpenCV library. Additionally, a set of preprocessing steps was implemented to automatically (or as close to automatic as possible) remove the background from the image, leave only transparent space and the coloured lines of the diagram.

The preprocessing stage followed a few relatively simple steps. Firstly, a Gaussian blur is applied to any image loaded. The resultant image is then converted to grayscale, and thresholded (using a “binary inverse” threshold - any pixel with value less than the threshold converted to white, otherwise converted to black) to produce a binary mask. This mask was then recombined with the original image using a binary AND, and an alpha channel added to produce a final image, hopefully having all of the diagram visible, with the background removed. Initially, I had used a static threshold value to achieve this effect, but this was insufficient for some images, scanned from acetate sheets, with very light colouring in some parts. Adjusting the static value to accommodate these images meant having parts of, or all of, the background in some of the scanned paper sheets fail to be removed (due to, I believe, the off-white/beige colour of the paper used). The solution here was to add a dynamic threshold value, with an adjustable slider, set to a reasonable default value. While not automatic, this worked in the majority of cases, and in any edge cases where manual tweaking was required I had a simple method to do so.

Of the tools created, the first I implemented was a cropping tool, which could be used to remove excess whitespace from the border of the scanned image, or pick a specific diagram from any scans with multiple diagrams drawn on one sheet. The user interface for this was simply selecting the crop tool, and then clicking on two points on the image, to create a bounding box to crop to. As I had previously decided on a horizontal resolution for images to be used in the mobile app (allowing the vertical resolution to be dynamic, with the smartphone app capable of vertical scrolling as needed) the editor would initially scale the images loaded to a set width. Upon cropping an image, I would then need to enlarge the image sub-section to re-fit this pre-defined width (see Figure X below).

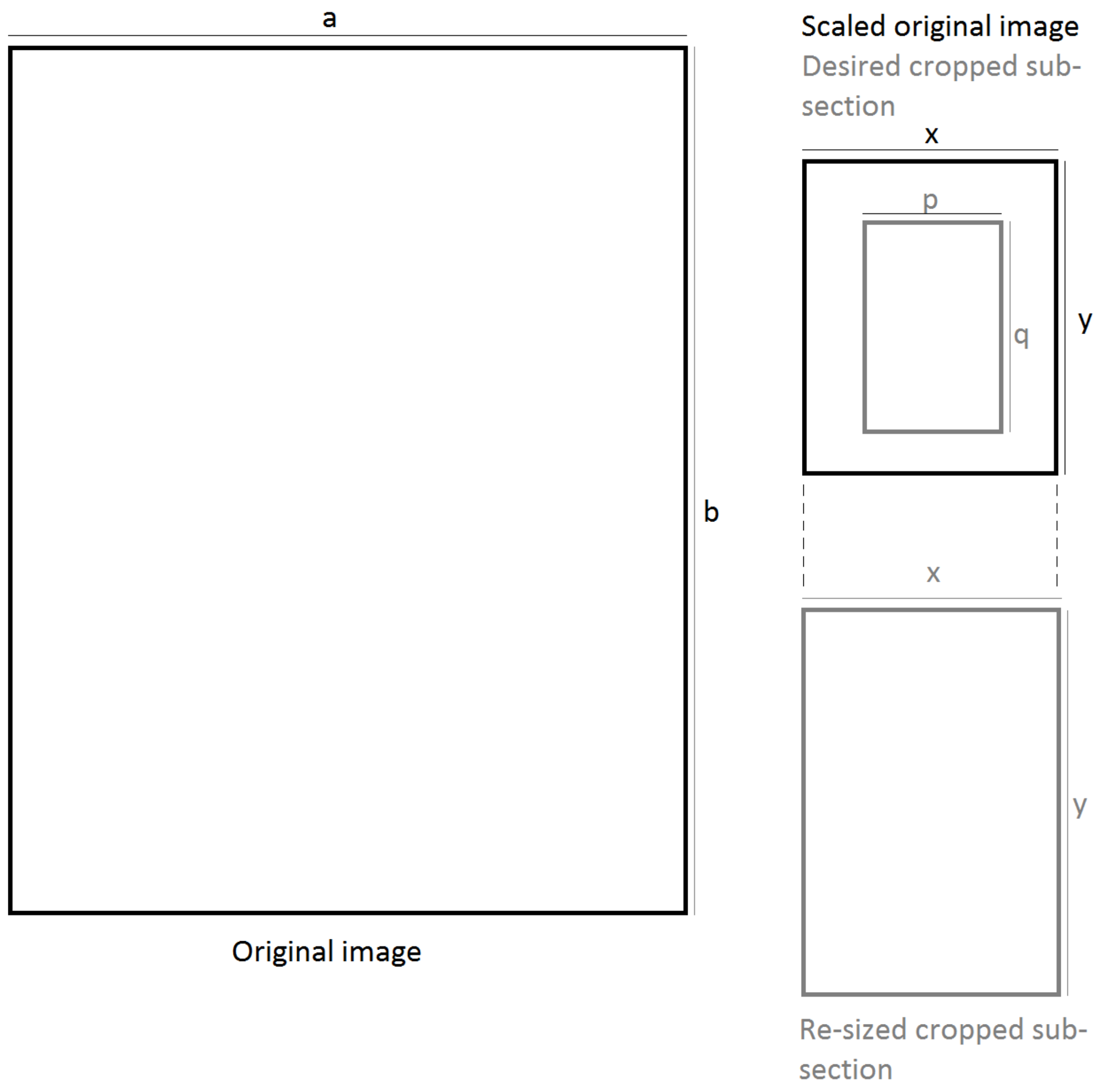

Figure X: Cropping while maintaining a set width

Unfortunately, doing these steps in the simplest manner (scaling the original image down to the correct width, then enlarging a subsection to this width when cropping) resulted in a sharp deterioration of image quality on each crop. Given, then, the size of the original images compared to the final resulting crop, the solution here was relatively simple: maintain a copy of the original image in memory, and upon cropping use the coordinates for the image subsection to reference against the original image. This section of the original image (which would still be the highest possible quality, not having been downscaled) could then be copied and scaled to the appropriate width. This added some complexity if multiple consecutive crops took place, as an offset would need to be calculated each time to ensure the correct section of the original image was used. Additionally, the crop could be reset by double-selecting the crop tool. This, in combination with the technique of copying sections from a high-resolution original image when cropping inwards allowed the crop tool to be used as a makeshift zoom tool, convenient for doing high-detail work.

Next, an eraser tool was needed, to cut out sections of the diagram containing only text, or remove any persistent artefacts created by fingerprints or other flaws in the source material after the preprocessing stage. This was implemented as a simple system using a complex-shape drawing function available in the OpenCV library. The user interface here allowed an unlimited (at least, in the human sense) number of points to be selected, each one adding a vertex to an area that would be erased. The area would then be filled with transparency, the background for the image, erasing that section. The image being modified here was the original image, the copy saved in memory to be used in case of cropping. Upon an erase taking place, the original image would therefore need to be re-cropped, keeping the current crop coordinates, after being modified.

The overlap of the tools here added complexity to both, and care had to be taken to ensure interactions between the two functioned as expected. While implementing the two tools, I attempted to implement an undo-redo system which would work as one would traditionally expect. This turned out to be quite challenging, for numerous reasons. Initially, the basic logic and functionality of the system required some head-scratching to properly nail down and implement in a satisfactory manner, given the aforementioned interactions between tools, and attempting to minimize work done (at this stage, the application was already performing slowly, the burden of processing high-resolution scans each time a tool was used strangling the hopes of a completely real-time interface).

Beyond this, I began to experience another, more worrisome issue - after performing a short series of operations, the application would crash, reporting a memory limit exceeded error. As I investigated the error, the source began to emerge: the high resolution scans, stored in memory, were being stored as a simple array of pixel colour values (as one might expect). While these scans, stored on disk as simple PNG files, were rarely more than a megabyte in size, the unpacked four-channel colour versions I was maintaining in memory were arrays of almost nine million entries in size, a not-insignificant amount of data. With some rough arithmetic, I believe these worked out to be around thirty-six megabytes each. While this may not seem too significant (indeed, the application was crashing when approaching one gigabyte of memory used, according to Windows' Task Manager), in creating the undo-redo system I had implemented the storage of operation history for the erase tool in the simplest manner possible - saving a copy of the original image prior to the erase operation, and saving a copy of the image after the erase operation. These copies were, as described above, based on the original high-resolution scan kept in memory, and so each was roughly thirty-six megabytes in size. A single erase operation, then, required seventy-two megabytes of memory, as well as an overhead necessary to perform the erase operation. This quickly added up as I attempted to modify an image, especially if multiple smaller regions required erasing, and appeared to be the source of the memory errors.

As a band-aid solution to the problem, I disabled the undo-redo system, awaiting future re-implementation with a better solution, and ploughed on in development. I did, however, take some time to attempt to optimize the rendering system, and the erase tool, to minimize the memory overhead required in both. I am still not entirely satisfied with either system, but feel that the current implementation works well enough, and using more time to further improve these aspects would likely be unproductive, and is certainly unnecessary.

A tool to manipulate how the images stacked on top of one another was sorely needed, to compensate for offsets generated by my original scanning. The user interface here allowed a layer to be put into a mode where it could be dragged around, allowing it to be easily overlaid onto the layer(s) below. This was simple enough to implement, a dynamic offset of each layer (other than the base layer, it being the immovable starting point) allowed the layers to be positioned as needed. Care had to be taken to ensure proper interaction with the cropping tool, and that when a cropped image was moved it loaded any region that would otherwise been “out of bounds” of the cropped zone (see Figure Y below).

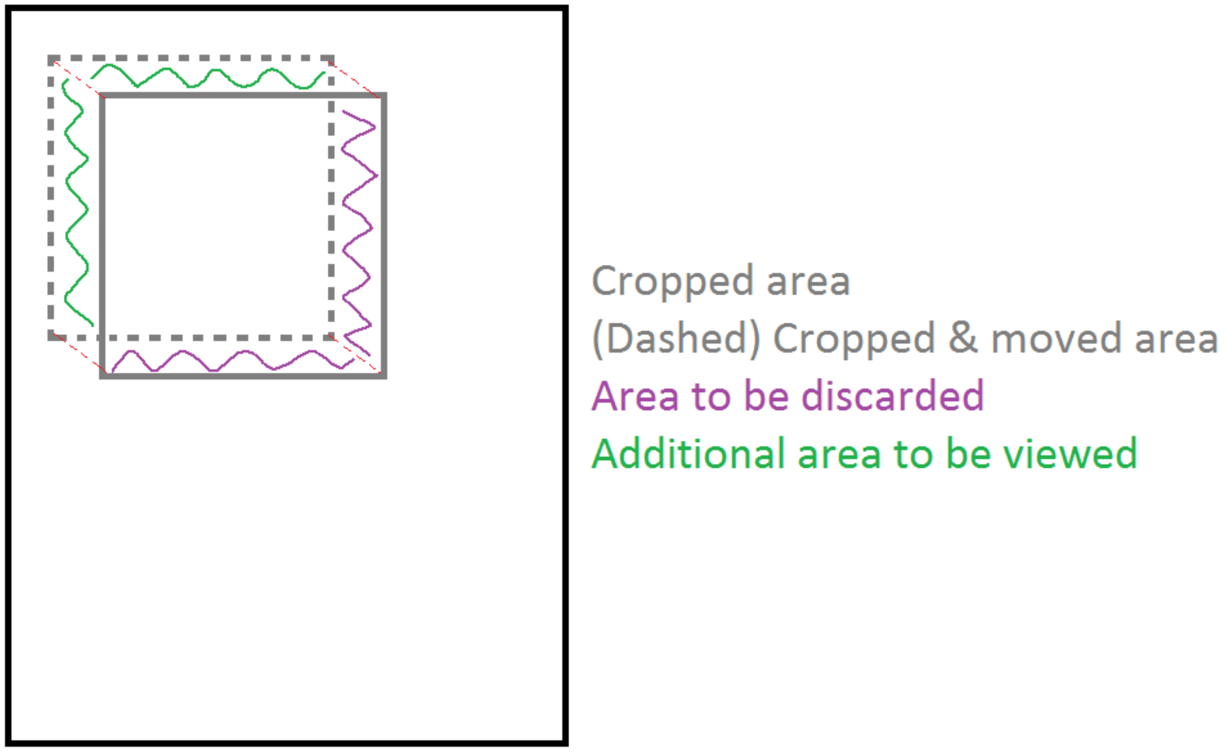

Figure Y: Interaction between crop and layer move tools

Additionally, an option to switch between viewing the entire image stack and just the currently selected image was added, to make editing diagrams with a high number of layers easier. This had the additional benefit of speeding up rendering when only working on a single slide, saving the processing time that would have been spent on the other slides.

With the ability to modify and prepare images for a single page more or less functional (missing undo-redo system aside) I switched to working on implementing a system for annotating the slides, to create a solution capable of adding text labels and other information.

While previously experimenting with the web application side of the project, I had modified some earlier experimental JavaScript code used to test the labelling functions, to instead generate a new, temporary label wherever I clicked on the image. This could have solved the problem of labelling the diagram, showing exactly what the label would look like if added as a permanent annotation, if I had implemented some way of saving these temporary labels.

I opted, however, to have the labelling be part of the Python application, to have a simpler, unified system with only one pass required. In retrospect, this might have been a mistake; implementing a system to perfectly recreate the web application rendering of a label in Python and OpenCV would have been extremely time consuming, and likely would not produce an accurate result without a ridiculous amount of tweaking and fine-tuning to emulate browser rendering using the primitive rendering tools available in the library. Instead, I used simple input fields and coordinate points to provide the label's text and location, respectively. While this basic solution was generally sufficient for purpose, being unable to view the resultant diagram and image combination that would be generated in the web application created situations in which labels I had placed could overlap, be positioned inaccurately, or simply not fit on the image. Due to the density of labelling on some of the more complex diagrams, I had to implement an additional feature (within the web application) after the fact, to allow a label to be flipped horizontally, to cram as much information as possible onto a slide. This required me to wade through the (soon to become monstrous, as I added more data) JSON file containing the location of the labels and add the relevant information manually, far from ideal. Instead, perhaps, I could create a second web application, sharing the rendering code used in the final web application, and implement a method to allow labels to be added and saved. Perhaps, even, this feature could be consolidated into the final web application (to be used as the smartphone application) to allow users of the smartphone app to add their own labels. This, however, would require a better storage solution for the smartphone app data, something I have been unfortunately unable to find a working solution for.

With the labelling taken care of, I could now create a full page of slides, labelled (somewhat poorly, in situations) and annotated. To convert this to a format the smartphone application could interpret, I implemented an exporting tool, which dumped the images to disk, organized in a specific directory structure, with a JSON file containing the related textual information. This would later be expanded to export multiple pages, using the same general principal.

With the ability to create, edit, and export a single "page" in the application (with any number of stacked "slides"), I began work on implementing the next level of functionality, having multiple pages. It was at this point that the memory issue I had run afoul of earlier, and promptly ignored, reared its foul head once again. With none of the pages having a particularly high number of slides (and therefore images required; the maximum being four or five) the impact of the relatively low memory ceiling had not been felt. Having the ability to add multiple pages, however, quickly brought it back to being the primary roadblock in development. Finally, it seemed, I was going to have to find a better solution to storing the images than simply leaving them in memory.

Conveniently, the array storage used in OpenCV is based off another popular Python library, “NumPy.” This library includes various methods of dumping array data to disk (simply exporting the arrays as images, and re-loading them, using OpenCV itself was not a valid solution, as confusingly the method OpenCV uses to load images discards the alpha channel of the image, of utmost importance to the task at hand - additionally, quality would likely be lost in compressing the images to common formats) so it was a relatively simple matter (some strange bugs aside) to just dump all the unneeded arrays to disk when switching between pages.

This partially solved the problem (and perhaps should have been a complete solution) with the program functioning almost as desired, but still occasionally ran into the same memory error. I could not pin down the exact cause of this, but it appeared as though whatever garbage collection methods are employed in Python would, on occasion, not actually free up the memory upon the arrays being deleted. I would speculate some part of the C/C++ to Python bindings being at fault here, but, as mentioned, could not locate the exact source of the issue. Thankfully, due to the nature of the user interface wrapping the image-editing code, this would not cause a full crash of the program (as somewhat falsely suggested earlier in this report), but just cause whatever operation was currently being attempted to fail, and an ugly error message to be produced. Leaving the program idle for a period (where my suspicion of garbage collection being at fault arose) seemed to alleviate the issue, and work could be continued.

I would very much like to know the root cause of the problem, as presumably there must be some way of solving it, but am at somewhat of a loss on how to proceed in this regard. From my rough analysis using, again (perhaps shamefully), Windows’ Task Manager to measure overall memory usage, the memory ceiling was very inconsistent, and at all times less than the standard allowed amount of memory for a 32-bit process on a Windows machine, further confusing the situation. I will attempt to dig deeper into the issue, as inconsistent and irreproducible as it is, with a better set of tools; hopefully a better examination of memory usage within Python will reveal some new information.

What Remains to be Done:
With the smartphone app and editor tools largely functional, the next step is to evaluate the app against the project specification. This will involve distributing the app (hopefully simple enough using Google Play, a conveniently standardized method which will be familiar to anyone using an android phone) to a group of willing veterinary students, and collect feedback based on a trial period by means of a short survey. This survey will need to be created, and a final build of the app uploaded to the Play store, as well as organizing a group of testers.

Besides evaluating the app, I aim to make some key changes to apps implementation, and generally polish the user interface on top of this. The biggest change here would be to create a proper system to store the revision material on the smartphone, to hopefully have the ability to add or remove material without having to re-build the entire app. As for polishing the appearance and feel of the app, some tweaks to the design, a first-time-use tutorial, and perhaps a night-reading mode ought to be enough to finalize that aspect of the project.

On the content-creation side, the image editor program could also use some work, although this is not critical to the project as a whole. Ideally I would wish to get to the bottom of the memory error issues, which would then allow me to implement features such as proper saving and loading of half-completed revision sets (currently, one has to simple export any progress made before quitting the editor, and then manually stitch any further material onto this base package after exporting it separately).

In addition to adding features, the source code for the project (mainly the smartphone application, but the image editor to some extent) is in serious need of some janitorial work, much of it written at the start of development and since been left largely unchanged.

Timeline for Completion:

- Now-start of Week 4: finalize app, organize test group, produce survey
- Week 4-start of Week 7: work on final report
- Week 7-End: collect feedback, finalize report

Section-level Skeleton of Final Report:

- Title Page
- Abstract
- 1. Introduction and Synopsis
  - 1.1 Introduction to the project
  - 1.2 Overview of published literature and concepts
- 2. Discussion of Work Undertaken
  - 2.1 Implementation details of smartphone application
  - 2.2 Implementation details of image editor application
- 3. Feedback and Evaluation
  - 3.1 Overview of feedback
  - 3.2 Detailed view of feedback
  - 3.3 Evaluation of goals against feedback
- 4. Conclusion
  - 4.1 Overview of final outcome of project
  - 4.2 Overview of unsolved problems and areas for further development
- Bibliography
- Appendix

Bibliography and References:
[1] http://developer.android.com/tools/studio/index.html
[2] http://developer.android.com/reference/android/webkit/WebView.html
[3] https://developers.google.com/web/updates/2013/12/300ms-tap-delay-gone-away?hl=en
[4] https://github.com/ftlabs/fastclick
[5] http://opencv.org/

Appendix:

[1]: Original diagram

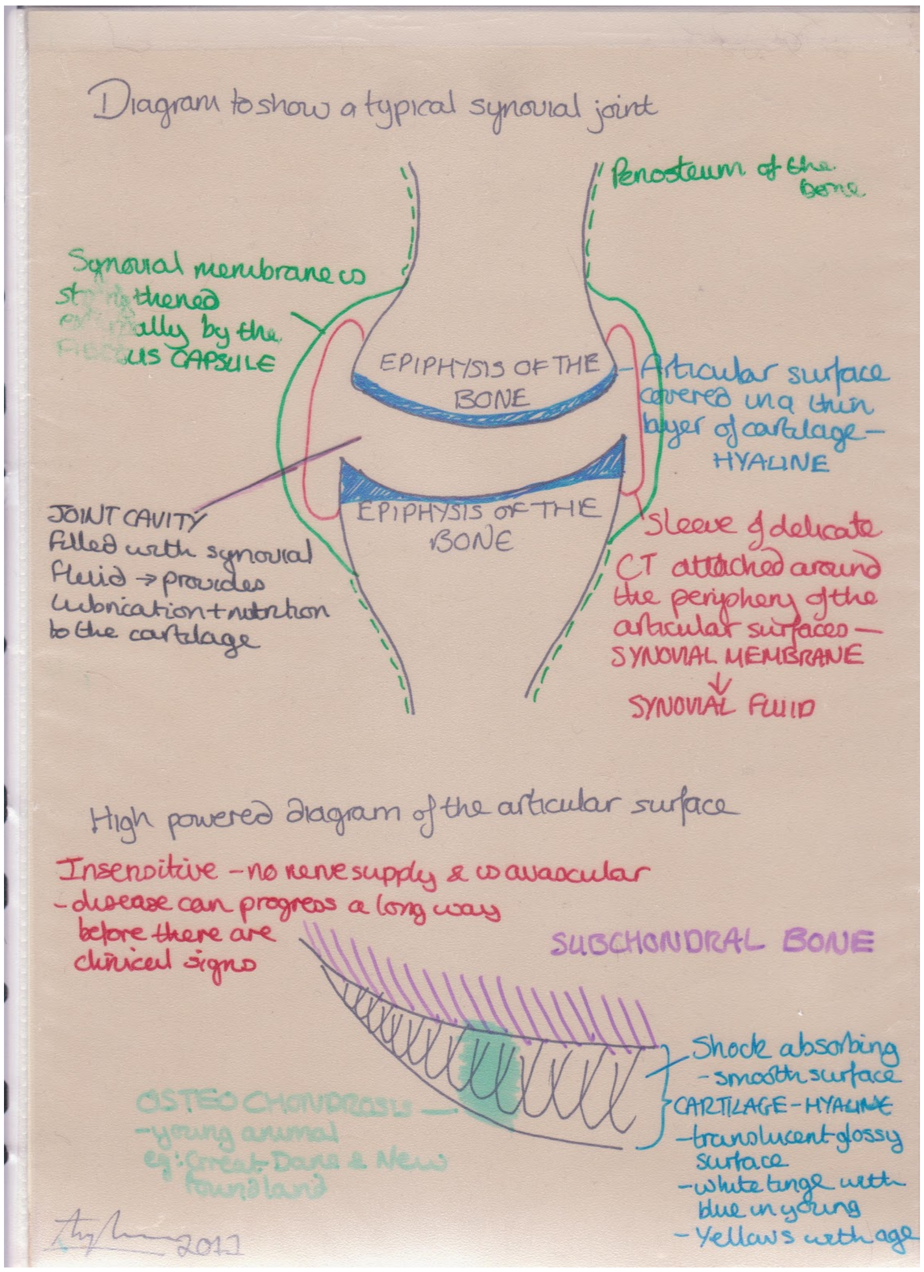
Diagram to show a typical synovial joint
Periosteum of the bone
Synovial membrane is strengthened externally by the FIBROUS CAPSULE
EPIPHYSIS OF THE BONE
Articular surface covered in a thin layer of cartilage - HYALINE
EPIPHYSIS OF THE BONE
JOINT CAVITY filled with synovial fluid → provides lubrication + nutrition to the cartilage
Sleeve of delicate CT attached around the periphery of the articular surfaces - SYNOVIAL MEMBRANE
SYNOVIAL FLUID
High powered diagram of the articular surface
Insensitive - no nerve supply & is avascular
- disease can progress a long way before there are clinical signs
SUBCHONDRAL BONE
Shock absorbing
- smooth surface
CARTILAGE - HYALINE
- translucent glossy surface
- white tinge with blue in young
- Yellows with age
OSTEOCHONDROSIS
- young animal
eg: Great Dane & Newfoundland
2011