Dynamically-Generated Image Mosaics

by

J. An

Supervisor: Karan Singh

April 2010

Dynamically-Generated Image Mosaics

by

J. An

Supervisor: Karan Singh

April 2010

Abstract

The artistic process of decomposing an image into a collection of tiles is analysed. The process of replacing each tile with another image from a database is automated, such that the resulting mosaic of images resembles the original image from a distance. The concept of an image mosaic is not new, but few examples exist of wholly automated processes of constructing image mosaics where the image replacements are not corrected or otherwise tweaked.

A number of methods for the various components in such a mosaic generator are considered. Although the way of characterizing images and tiles is simplistic, a random method is proposed to mitigate the visual periodicity that becomes apparent in regular tiling across homogeneous regions of the original image.

Lastly, a web platform and framework is proposed to house this image generator.

Acknowledgements

It is a pleasure to give acknowledgement and thank those who made this thesis possible.

First and foremost, I would like the thank my thesis supervisor for proposing this topic and setting me in the direction to complete my research.

I would also like to thank my family, friends, and colleagues for their unending moral support as well as editorial support.

Lastly, I would like to thank the many open source communities, who's work made this thesis possible. Specifically, I would like to thank the Drupal community who continues to produce the enterprise-quality content management system, Drupal, on which this thesis is heavily based as a development framework.

Table of Contents

List of Figures

List of Equations

List of Tables

1. Introduction

The term *mosaic* first appeared in the English language in the 16th century, according to the Oxford English Dictionary [1], describing a "process of creating pictures or decorative patterns by cementing together small pieces of stone, glass, or other hard materials of various colours." Since then, the resulting work and any work analogous to the traditional mosaic could also be referred to as a mosaic, itself. Artists who create (often traditionally) mosaics sometimes call themselves mosaicists.

A mosaic is composed smaller pieces or *tiles*, traditionally made of something like stone, rock or coloured glass. Much like how brush strokes collectively compose a painting, these tiles compose a mosaic. When the artwork is viewed from up close, the individual smaller elements (brush strokes or tiles) are visible; yet when viewed at a distance the collection of strokes or tiles blend together to yield the overall picture.

Although [2] cites uses of the word photomosaic in the English language as early as 1920, the first images tessellated by other images created at least in part by software didn't emerge until the early 1990's according to [3] and [4].

1.1 Objective

This paper explores the use of software to generate *image mosaics* (referred to as just mosaics from now on) both on-the-fly and entirely automatically. The various components of a mosaic generator are described and the various implementations described. In this paper, mosaics are created by tessellating a collection of small images (or thumbnails) or *tiles* so that they suggest an overall image.

Even a brief Internet search across the topic of mosaics will retrieve commercially, personally, and even politically motivated purposed for creating mosaics. The primary purposes for the work described in this paper is both artistic and accessible in nature.

Tessellating tiles in a mosaic may eventually rival the artistic mosaics of current artists or it may provide such artists with a new tool or method with which art and creativity may be expressed. It also provides opportunities to juxtapose or contrast the various tiles from one another or from the overall image, such as for creating advertisements and political or popular messages.

Accessibility, especially to a non-technical user, is emphasized as users should not need to know how the mosaic generator works to use it. With the proliferation of image capture devices (e.g. digital cameras and DSLRs, webcams, camera-equip mobile devices, scanners) and the vast repositories of public or private images, there is interest in creating something new from these images.

This paper explores both the technical challenges of creating the generator and a software framework to implement this generator for general use.

2. Related Work

The earliest known examples of traditional mosaics dates back to the second half of the 2nd millenium BCE (1500 BCE–1000 BCE) according to [5]. Depending on the strict definition, modern mosaics using images as tiles began to appear in the first half of the 20th century. These mosaics and their manual methods of composition are not directly to this paper.

Algorithms and software to either assist in generating or wholly generate mosaics began to appear in the early 1990's. These early pioneering examples were constrained by the comparatively limited computing power and small image databases available at the time.

It appears that the topic of mosaic generation has not garnered much academic attention, but a number of commercial and personal projects are publicly available [6–11]. The concept and innovation around creating mosaics has also been fettered by a trademark on "Photomosaic" patent [12] on technology creating photomosaics both held in the US by [7] who has asserted his patent against others [13–14]. The patent was applied in 1997 and came into effect in 2000. As of February 16, 2010, a vast majority of the patent has been invalidated on the grounds of prior art [14].

A seminal paper [15] describes a process for creating mosaics and explores various algorithms in the various components of the process. It focuses on areas not covered in this paper: on digital half-toning; it also relies heavily on colour correcting the tiles or overlaying the original overall image translucently over the mosaic so that the mosaic is more suggestive of the overall image. It appears the work in this paper is constrained by the number of images to choose tiles from and may be constrained by the hardware of the time, limiting the number of tiles in the mosaic and the size of the mosaic. It also indicates that manual work in either preparing the images for tessellation or tweaking the resulting mosaic occurs.

[16] explores more complicated methods of creating mosaics, using a greedy approach to place larger tiles over homogeneous areas of the source image, and more descriptive ways to describe both the source image tiles and its replacements. To this end, the paper focuses on improving the quality of mosaic generation by easing the visual periodicity of regular tiles with uneven tile rectangles and by more accurately describing tiles. The experiments described in the paper were performed on an average computer with a 2.2 GHz CPU and 2 GB of RAM with the mosaics generated within a few minutes with several thousand tiles.

3. Problem Specification

Two processes are required for creating mosaics: one to build a database of images from which to choose tiles, and one to accept a source image and generate a mosaic from it using the database of images.

The second process of generating the mosaic has a few components. The source image is first broken into tiles. Second, the tiles need to be somehow described or characterized. These characterizations are thirdly compared against the characterizations of images in the database. A good or best matching image is selected. Finally, the selected image replaces the tile. By replacing all tiles with image matches, the mosaic is tessellated.

The first process in building the database is straight-forward. Images are fetched from a source and formatted into the database. Some preprocessing work can be done here in characterizing the images in advance and stored with the images. If the tile sizes are fixed or somehow known in advance, thumbnails of appropriate size can also be cached with the images so that the images need not be cropped or scaled when the mosaic in being generated.

4. Application Framework

The specific framework to implement the two processes that will compose the mosaic generator is crucial, as the selection will indicate the programming paradigm, general speed of execution, and ease of development as the various frameworks and toolkits in the various programming languages can differ greatly in what they offer and how they assist in the development of software.

The choice of programming language is also key as specific language is closely link to the programming paradigm or model and actual speed of execution (as opposed to time complexity, which is often implementation-agnostic).

The focus is on speed of code execution, availability of software development kits (SDKs) and libraries, platform independence, and ease of use.

4.1 Development Philosophy

The development process borrows from Agile software development or simply *Agile*. Agile in its modern definition evolved in the 1990's and now many specific Agile methods exist [19]. The development presented in this paper does not ascribe to any specific Agile methodologies; rather the process borrows some principles from Agile, namely:

- Software is maintained in a functioning manner;
- Modifications to the code leaves it in a consistent and functioning manner;
- Progress is done in a short-term iterative cycle;
- Simplicity, code reuse, and leveraging open source code bases are emphasized;
- Requirements and specifications are expected to evolve as the work progresses; and
- Goals of the work are expected to change as the work progresses.

The focus of the philosophy is this. The aim of the work is to produce a reasonably functional image mosaic generator as well as experiment with different ways to implement the various components. Despite developing specific requirements and specifications, understanding of the problem and the code will inevitably evolve over time and fulfilling the aim may mean changing the development of the software.

4.2 Programming Language Prevalence

The selection of the development programming language and framework needs to be done together as frameworks are built on their language foundations and the choice of a language is heavily weighted on the collection of mature SDKs available in that language.

[20] collects data on the relative popularity of programming languages across a number of sources. Its normalized comparison, across sources like Google Code, Ohloh, del.icio.us, and Craigslist, show the top five most popular languages are in the descending order from

most popular: Java, C, C++, PHP, Javascript, with Java being just over twice as popular as Javascript. On a separate metric, its normalized discussion comparison indicates the five most discussed languages are in descending order: C++, C, Java, Python, and PHP.

[21] maintains an index updated monthly measuring the relative frequency of web searches made with Turing complete programming language names, across a number of search engines. Its top five ranked languages are in descending order: C, Java, C++, PHP, (Visual) Basic.

The four languages that consistently rank high among these popularity indices are: C, C++, Java, and PHP.

4.3 Analysis of Programming Language and Framework

Java is heavily object-oriented and its code is compiled into an intermediate language called bytecode, which is a set of instructions that a Java Virtual Machine (JVM) can interpret and execute. Because JVMs are available on many hardware and software platforms, the same bytecode can be executed on many platforms interchangeably.

The Java Development Kit (JDK) is a large library covering various API. Image processing is handled by Java Advanced Imaging (JAI), an extension application programming interface (API).

C and C++ are both general purpose languages which are cross-platform only in that code can be compiled in multiple platforms; yet, once the code is compiled into a binary image stored as an executable, the binary can only execute on the specific operating system and computer architecture for which it was built. Porting the same code to a different platform is non-trivial, and can be a huge refactoring task depending on the code's purpose and the platforms to and from which the code is being ported.

The C and C++ Standard Library provides a collection of common operations. The C++ library supports templates, generic containers and common algorithms, as part of the C++ ISO standard. The Boost C++ Libraries is a collection of some 80 peer-reviewed libraries, one of which provides functionality around image processing.

PHP is a general-purpose scripting language designed initially for web development and the dynamic creation of web pages. As a server-side scripting language, PHP is embedded in HTML source documents and interpretted by a web server with a PHP processor extension. As a server-side language, the code is compiled on the fly, as the document is being requested by a client using a program like a web browser. Any client that can access the PHP embedded document on the server may run the script regardless of its platform or configuration. PHP is supported by the three major web servers [22]: Apache, Microsoft IIS, and nginx, who collectively serve 85% of the market. (Google's private web server accounts for an additional 6%.)

PHP's library is also extensive. The PHP Extension and Application Repository (PEAR) further extends PHP's functionality through contributed PEAR packages. The GD Graphics Library is well supported in PHP in dynamically manipulating images.

Drupal is a free and open source content management system written in PHP, that powers both small and enterprise-sized websites [24], such as whitehouse.gov. It's standard release or core is supplemented by a module extensible system and a vast collection of community contributed modules that cover functionality like image processing and caching, and record storage.

Comparing speed of code execution [23], C and C++ are very similar, whereas Java executes about two times slower. PHP executes about two orders of magnitude slower than C, C++, and Java.

PHP and Drupal 6 are chosen for this work despite its inherent speed drawbacks. When storing and retrieving data, a MySQL database will be used, which will help mitigate the performance issues when dealing with many images and their characterizations. PHP has a large standard library of functions and Drupal will provide a general web framework that will handle most operations right up to accessing images.

Because much of the input and output (I/O) and the graphical user interface (GUI) is provided out of the box with some configuration by GUI, the setup and tear down of the process is all handled without much coding.

Overall, the system implements a solution stack running Linux, Apache (web server), MySQL (database), and PHP, collectively known as the LAMP stack, plus Drupal 6. The entire stack sites on a publicly accessible virtual private server (VPS) provided by Slicehost. The VPS specifications are:

- Quad-core, 64-bit CPU running at 2.2 GHz (two Dual-Core AMD Opteron Processor 2214)
- RAID-10 disk storage
- Gigabit network backbone
- 256 MB of RAM

Slicehost sells its VPS based on how much RAM is dedicated to the VPS. The 256 MB VPS is quite constrained in terms of memory.

The work is being hosted currently on **mosaic.jamesan.ca**.

4.4 Drupal Structure

The specific details of the Drupal implementation can be found in Appendix A. The following is a discussion about the overall structure of this implementation of Drupal.

Drupal refers to each item of content (e.g. a web page, a news item, a blog post) as a node. Nodes are assigned a content type (e.g. page, forum post) upon creation. Here, Drupal is implemented with two content types: images and mosaics. An *image* node stores an image to be used by the mosaic generator as both a candidate to replace a tile and the source image for the generator, and its pre-computed characterization. A *mosaic* node stores the resulting image mosaic produced by the mosaic generator.

On the home page, the user is presented with two listings: one of image nodes and one of mosaic nodes. Clicking on a mosaic thumbnail brings up the mosaic node which just shows a larger version of the mosaic that is click-able to a pop-up showing the mosaic in 100% magnification. Clicking on an image thumbnail brings up a similar page with the addition of a link that commences the tessellation process to generate a mosaic with the image as the source.

The mosaic generator is written as a module of Drupal, leveraging the ways Drupal is designed to be extended using its hook system. Drupal allows modules to *hook* into its process in generating the requested web page at specific points in the execution. For example, when the list of menus and menu items is being built, Drupal will allow modules that implement the menu hook to add its own menus and menu items. When a form is being constructed, when a node is being viewed, loaded, or saved, Drupal provides an opportunity for modules to execute its own code. This is how Drupal's functionality is extended—for example, how additional fields can be defined and implemented for specific content types.

The duration required to import many images into the database or generate mosaics with many tiles will often far exceed the time allotted to a single PHP script execution, resulting in a timeout. To work around this, Drupal provides a Batch API that allows code execution over several page requests to avoid the PHP timeout and also display a progress indicator to the user after each request in the batch is processed via Ajax (or DOM scripting, as some have argued is more accurate a label). Across page requests, data is kept persistent by storing the serialized form in the database prior to the end of one page request and retrieving it in the next request. Because of the way Drupal stores the persistent variables under a single row in the database, large data structures can severely hamper the speed of processing done with the Batch API.

5. Process Component Design

The following is a breakdown of the various components of these two processes and various ideas and methods to fulfil their requirements.

5.1 Image Database

Earlier work used a small database of images: a couple hundred. It became clear the database needed to be vastly expanded to capture the variety of images and possible colouration.

There exists huge banks and caches of images on the Internet that can be used to populate the database. There are now many candidates such as Google Image Search, Wikimedia, Flickr, Photobucket, and deviantART. These have collections in the order of millions or billions (in the case of Google [17] and Flickr [18]) of images.

Flickr was chosen as it contains the most images and its API response can be returned in the PHP serialized format. The Flickr API implements, among other request formats, the Representational State Transfer (REST) architecture. The RESTful service is accessed by sending an HTTP GET request to the Flickr REST Endpoint URI with the parameters encoded in the query fragment of the URI. The relevant paramters are: the Flickr API method, flickr.photos.search; a comma-delimited list of tags; the number of photos to return in this call, up to 500; and the page number, if there are more photos than requested. For example, to request for 500 images with the tag, “black and white,” on page 5 (that is, images 2001 through to 2500), the REST Endpoint URI would be:

```
http://api.flickr.com/services/rest/
?api_key=5c0b0b7235116793a29e8bbd1f516f92
&method=flickr.photos.search
&tags=black+and+white
&format=php_serial
&per_page=500
&page=5
```

To populate the image database, the above method of searching for images on Flickr is used as follows.

The user is presented with a form asking for two inputs: a comma-delimited list of tags to search, and the number of images to retrieve per tag. Although Flickr can process all the tags together in a single request, each term is isolated to guarantee images will be retrieved across all tags.

An array of Flickr URI to images is first constructed. If more than 500 images per tag is indicated, multiple requests to Flickr will need to be made for each tag. In either case, the number of pages is first retrieved by making a request to Flickr with the tag and number of results per page, dumping all the results, and returning just the number of pages available.

In the case of more than 500 images, 500 images are requested with a random page number for each tag until less than 500 images are needed to fulfil the specified number of images to be retrieved. At this point, this devolves to the case where less than or exactly 500 images are needed. Each set of results returns an array of image information that is formatted into the Flickr URI that addresses the image. One last request is made to Flickr to complete the array of image URI.

When retrieving images and even between unrelated requests to Flickr, the algorithm should somehow avoid ever selecting the same page number. Tracking the page number is not realistic, especially since the number of images per page may not always be 500 and since the Flickr image database is constantly changing. A simple heuristic to avoid selecting the same page number is to just select a random page at every request. This heuristic becomes increasingly successful the more pages are available.

Tag	# of Pages	# of Images
landscape	6636	3.3 mil.
building	3840	1.9 mil.
portraits	1851	0.9 mil.
animals	4445	2.2 mil.
snow	10066	5.0 mil.
art	14116	7.1 mil.
poster	768	0.4 mil.

Table 1: Listing of Flickr tags used to populate image database.

Using some of the most popular tags [25] from Flickr shown in table 1, it's highly unlikely that the same page will be selected more than once, especially if the database is being populated by just some several thousand images. Calculating the risk using the most and least popular abovementioned tags and requesting 2000 images or four pages of 500 images, there is a 0.03% and 0.5% chance, respectively, in selecting a page more than once.

Once the array of image URI is constructed, a new batch is *open*, as Drupal calls the process of creating a new batch job, and immediately processed. The user is then led to a page with an progress indicating bar that updates progressively as the batch is processed.

With each image URI, the image file at the URI is first copied to the server. An entry in Drupal's file tracking table is made and lastly a node is created to associate the image to its *image* content type exposing it both to the GUI and to the methods used later to select all images. Using the hook system, the image characterization is calculated right before the image node is saved (the *presave* operation) and inserted into the node to be saved.

Currently, the database includes about 11 300 images, or about 1600 images for each of the seven search tags used as listed in the table above.

5.2 Characterizing Images

When replacing the source image tiles with scaled down images, the tiles and images needs to somehow be compared. Here, there is a trade off between performance and quality of selection. On the extreme side of quality, if a tile is compared to full size images, there is the greatest amount of information that can potentially lead to the *best* matching image, whatever best is defined, but the amount of work in this process is unrealistic. On the other extreme, both tile and image can be characterized by a single colour value, as if both were scaled down to a single pixel (px). Between these extremes, the resolution of the characterization can be increased and transition between the subset of colour values (or pixels) can also be approximated by various interpolation methods.

Characterizing images with its average colour is the simplest method. The colour values, red, green, and blue (RGB), are averaged over all pixels. Implementing an average function that loops over all pixels in PHP is incredibly inefficient as the pixel data is not directly exposed to the code. Images are stored as a resource, a special PHP variable that references an external resource. The I/O to access specific pixel information is not efficient. Instead, it's faster to resample the image into a one by one pixel space and then retrieve this pixel's RGB value. This effectively averages the image's colour data.

The image can instead by resampled to larger spaces (e.g. three by three or greater) to better approximate how the colour is distributed across the image. In this paper, the results use the average colour method as the results were unexpectedly good given a large enough database of images and small enough tile size.

The two methods described above to find the average colour are summarized as follows: one, the image is resampled to a 1 px by 1 px space and the colour of that pixel is returned; two, the colour data of each pixel of the image is retrieved and the average colour is manually calculated.

The time to calculate the average colour was measured using both methods and across a collection of images of varying file size and image size (number of pixels). Linear regression was applied to the data using the 2nd-order non-linear equation in the form:

$$z=a_1x+a_2x^2+a_3y+a_4y^2+a_5xy$$

Equation 1: The 2nd-order non-linear equation used to analyse the relationship between image and file size and processing time to characterize images.

where z is the processing time (in ms), x is the file size (in kilobytes or kB), and y is the image size (in megapixels or MP).

Eight images with the following attributes were benchmarked:

File Size (kB)	Image Dim (px)	Image Size (MP)	Time 1 (ms)	Time 2 (ms)
24.21	224 by 324	0.08	9.1 ± 0.8	520 ± 50
46.91	353 by 500	0.18	20 ± 3	520 ± 50
176.44	375 by 500	0.19	28 ± 3	580 ± 60
193.57	500 by 333	0.17	28 ± 7	490 ± 50
273.63	500 by 334	0.17	28 ± 2	500 ± 30
304.99	324 by 500	0.16	28 ± 3	500 ± 80

339.36	500 by 500	0.25	42 ± 3	740 ± 70
369.14	3920 by 4000	15.68	1500 ± 180	49000 ± 2000

Table 2: Benchmarking results measuring duration of the two characterization methods..

Using the PHP function *microtime()*, the processing time for the resample method (time 1) and manual method (time 2) was measured. For time 1, the averaged value is computed from 5000 independent measurements. For time 2, the value is computed from 50 independent measurements. By visual inspection, the manual computation method is about 1.5 orders of magnitude ($10^{1.5} \approx 31.6 \text{ times}$) slower than the resampling method and this is quite consistent across the range of file and image sizes.

Inspecting the time 1 and 2 data sets, there's a clear positive correlation between file and image size to the processing time. Yet, the processing time of the respective data sets depend differently to the file and image size. Time 2 depends more heavily on image size than time 1, as the manual method is slower in the 3rd case where the image size increases, but the file size decreases, while the resampling method is relatively unaffected.

Applying the data from the resampling method to equation 1 for linear regression, the coefficients are:

$a_1 = 0.05$
$a_2 = 0.0$
$a_3 = 96.16$
$a_4 = -4.17$
$a_5 = 0.18$

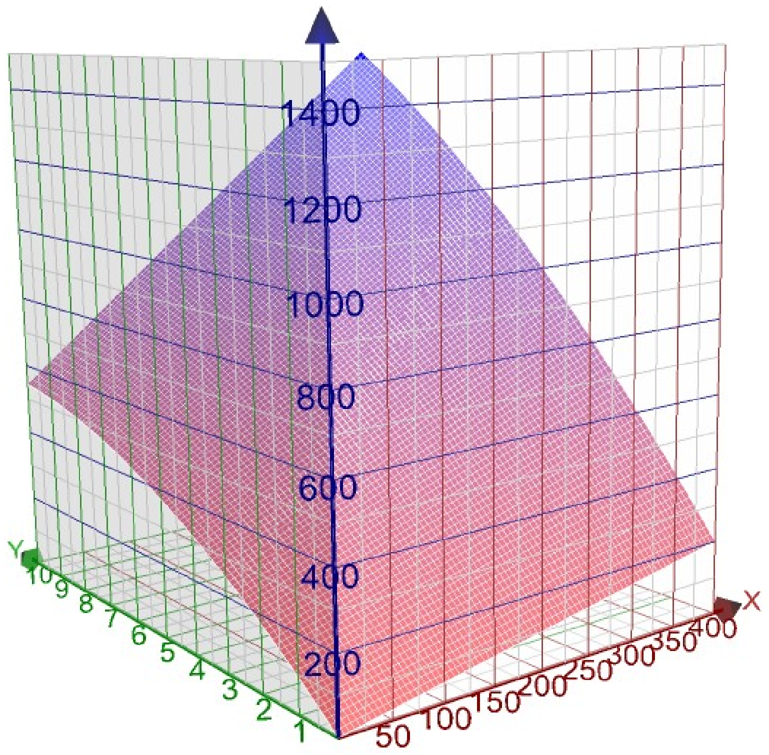


Figure 1: *Visualization of equation 1.*

5.3 Comparing Characterizations

There are a number of methods to compare the characterizations of the images with the tile. In this paper, using the average colour characterization, the problem becomes a vector search issue. The tile has a specific RGB value to match. The problem is to find the image who's average colour is the closest to the tile's average colour. In RGB space, a number of metrics can be used to measure the 3D distance between RGB values.

The Euclidean, Manhattan, and Chebyshev metric are three popular distance functions for vector space with the following definitions, given two vectors p and q, with standard coordinates p_i and q_i:

$$D_{Euclidean} = \sqrt{\sum_i (p_i - q_i)^2}$$
$$D_{Manhattan} = \sum_i |p_i - q_i|$$
$$D_{Chebyshev} = \max(|p_i - q_i|)$$

Equation 2: Popular metrics for Euclidean space.

The International Commission on Illumination (CIE, for its French name, *Commission internationale de l'éclairage*) defines the colour metric differently, in an attempt to accurately reflect perceived colour differences. Over the years, CIE has refined this colour metric called ΔE^*_{ab} or simply *Delta E*, as the human eye is more sensitive to certain colours than others. Delta E is a complicated function with numerous square roots, exponents, and divisions, all of which are expensive math functions.

It also uses a different colour model, lightness, chroma, hue (LCh) and [26] and [27] show that the conversation between RGB and LCh is also an expensive calculation, involving trigonometric and piecewise functions. Delta E was not chosen due to the expensive nature of the involved calculations, but the Euclidean metric was chosen as [28] shows the original Delta E calculation was a simple Euclidean distance in Lab colour space.

5.4 Searching for Images

When the source image is broken into tiles, each tile must be replaced with an appropriate image from the database. The speed of this process depends heavily on what algorithm is used to search through the image database. Searching for images can be abstracted as the nearest neighbour search or proximity search using the characterization of the image as the key. The characterization is considered an n-dimensional point in metric space. Given a set of such points representing all the images in the database, find the closest point in this set to the characterization of the specific tile.

As discussed previously, the characterization method used in this work is the average colour of the image represented as a single colour point (3D). Here, the proximity problem is finding the closest 3D point to the average colour of the tile given a set of 3D points representing the average colours of all the images in the database.

The early results of the searching for the closest match produced mosaics with highly periodic regions where the original source image had regions of homogeneous colour (e.g. A large piece of blue sky). To work around this visual problem of periodicity, the selected image would only need to be a *good* match to the tile.

Approximate nearest neighbour search algorithms were explored, where the algorithm returns a close match to the candidate without guarantee that the result is the absolute closest match. This loosening of the search criteria often yields faster searches. Yet [29] reported that the *Best Bin First* solution the approximate nearest neighbour search problem often yielded the best match, which would not solve the problem with repeating images in the mosaic.

Instead, the problem was changed to boundary search problem, where all neighbours within a boundary around the searching point is returned. Between these candidates, one image is randomly selected as the replacement. Empirically, the distance in RGB space of $O(tn)$ was found to be sufficiently small that only similar images were returned and sufficiently large that, in homogeneous regions of the source image, the visual periodicity is broken up.

The naive approach is a linear search for each tile. [30] shows that the linear search may be as fast as or even better than the more complicated search methods that partition space, especially as the dimensionality of the data increases. The time complexity of the linear search is: $O(tn)$ where t is the number of tiles and n is the number of images in the database.

Using the branch and bound methodology, another method is to partition the RGB colour space. A *kd*-tree is a binary space partitioning (BSP) tree that recursively bisects space with planes that are normal to one of the axes (in a cyclical order, i.e. bisect the space with a plane normal to the red axis first, then the green axis, then the blue axis). The time complexity of constructing such a tree is $O(n \log^2 n)$ and the time complexity of traversing such a tree is $O(tn^{2/3})$ [31].

Although using a BSP tree provides substantive performance gains and grows increasingly better as the image database increases in size, the large tree structure would hinder the Batch API process as it would need to be stored persistently in the database and unserialized on each page request. Because of this, the linear search was implemented.

5.5 Colour Correction Technique

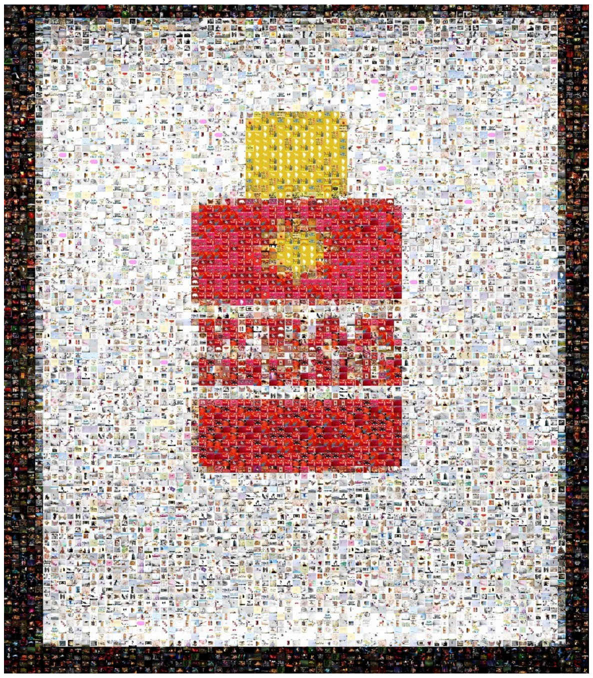
Many other mosaic generators modify the images used to replace tiles either by overlaying a translucent version of the source image over the mosaic or by biasing the colour of the image replacement to the average colour of the tile.

A colour correction technique was implemented in this generator to experiment with the performance of the technique and quality of the resulting mosaic. Essentially, the technique overlays a translucent fill of the average colour of the tile on top of the image replacement giving it a hue that more closely resembles the tile it is replacing.

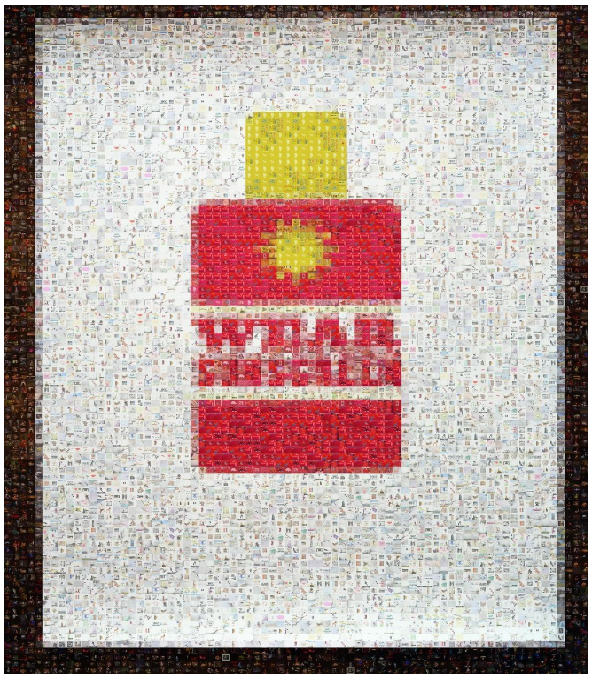
This additional process extends the duration of the tessellating process by approximately 15%.

Figure 2 shows two 7300-tile mosaics generated from the same source image, one without and one with colour correction. The aesthetic value of colour correction is arguable and likely depends on the subjective application or use of the mosaic. Hence, the appropriateness of using colour correction is beyond the scope of this work.

The remainder of the mosaics are largely not colour corrected as it can be considered a post-process addition. The generator deals strictly with the tessellation process.

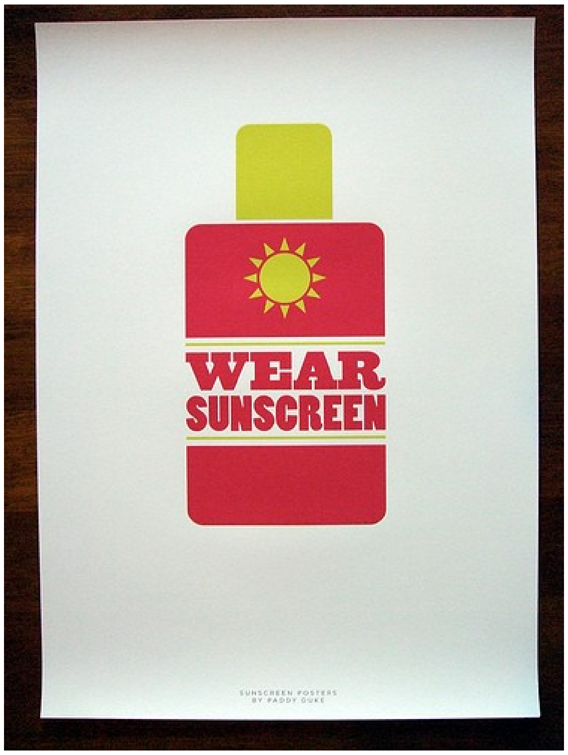


(a)



(b)

Figure 2: (a) is a mosaic without colour correction and (b) is one with colour correction based on the source image (c). Colour correction enhances the recognition of the original image and dulls the details within the individual image replacements.



(c)

6. Results

Results were generated throughout the process of refining the work, as described in the development philosophy. The evolution of results is discussed here along with the improvements made to the mosaic generator to both increase the quality of generated mosaics and also scale the generator to deal with a larger image database and larger mosaics.

6.1 Initial Results

The initial generator implemented most of the designs and algorithms as described in the previous section, but the results were limited and discouraging.

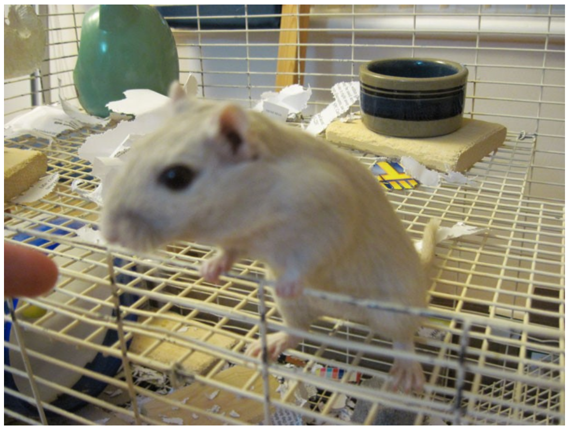


Figure 3: *The source image for the adjacent mosaics (figure 2 and 3).*

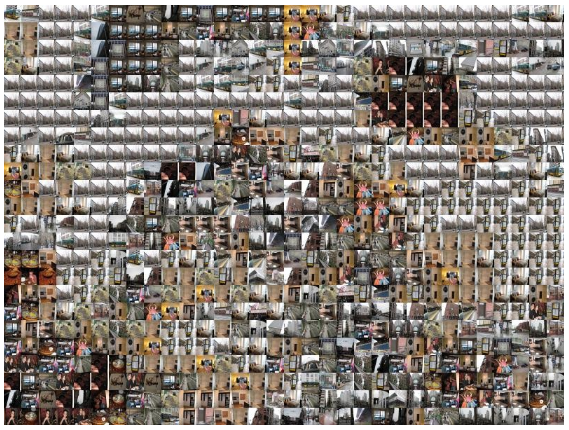


Figure 4: *One of the first mosaics.*

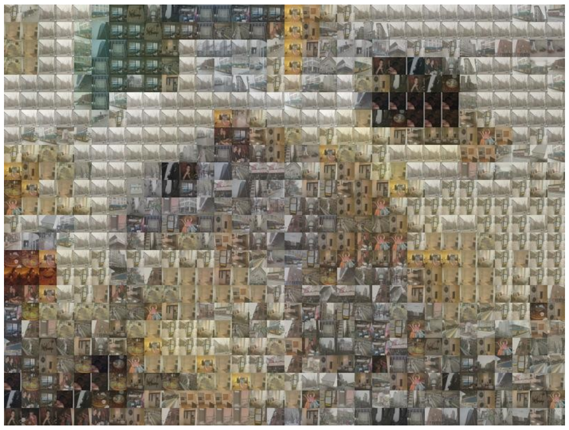


Figure 5: *The mosaic with colour corrected tiles.*

Figures 3-5 show one of the earliest results of the generator. Figure 4 is a mosaic that uses the original images as tiles, just as they are (i.e. without colour correction) and it's obvious that the mosaic does not resemble the source image very well. Among the problems is the noticeable periodicity of repeated images over the mosaic and the inappropriate match of many of the tiles.

Figures 4 and 5 contain 768 25 px by 25 px tiles creating an 800 px by 600 px mosaic.

These early results were greatly constrained by an excessively small image database of about 100 images from which to choose as tile replacements. Hardly enough images to appropriately construct a mosaic. Still, it's visually evident that the algorithm is basically functional, as the tiles do respond to the colour changes throughout the image. Figure 5 confirms this as the colour correction clearly shows that the average colour of each tile has been appropriately calculated (and the tile images corrected accordingly).

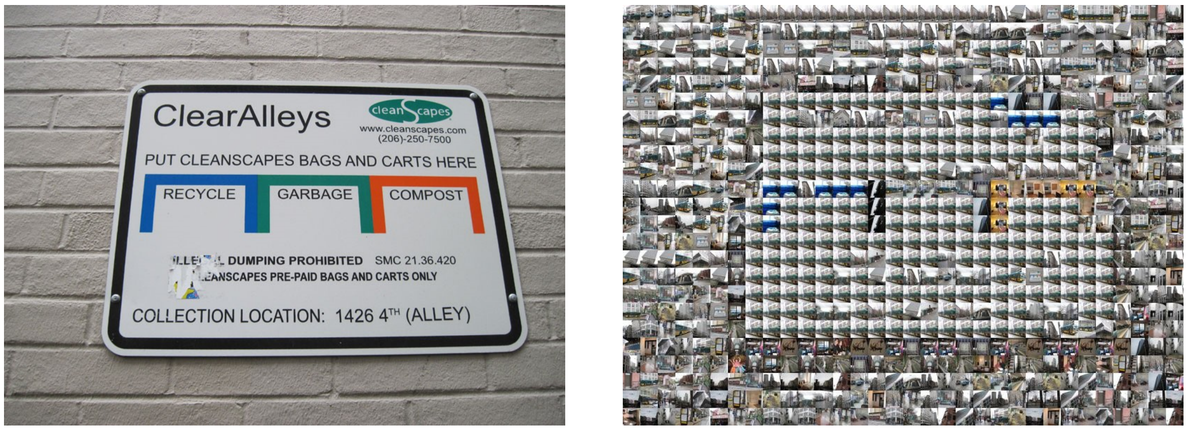

ClearAlleys
cleanScapes
www.cleanscapes.com
(206)-250-7500
PUT CLEANSCAPES BAGS AND CARTS HERE
RECYCLE
GARBAGE
COMPOST
DUMPING PROHIBITED SMC 21.36.420
CLEANSCAPES PRE-PAID BAGS AND CARTS ONLY
COLLECTION LOCATION: 1426 4TH (ALLEY)
(a)
(b)

Figure 6: *Another mosaic created by the early algorithm plagued with the same issues as discussed with figures 1-3.*

Figure 6 clearly shows the responsiveness of the basic algorithm to colour of the tiles, especially around the blue, green, and orange markings on the sign in the source image. Still, the mosaic suffers mainly due to relatively large tiles (i.e. the image should be composed of more images) and a relatively small image database.

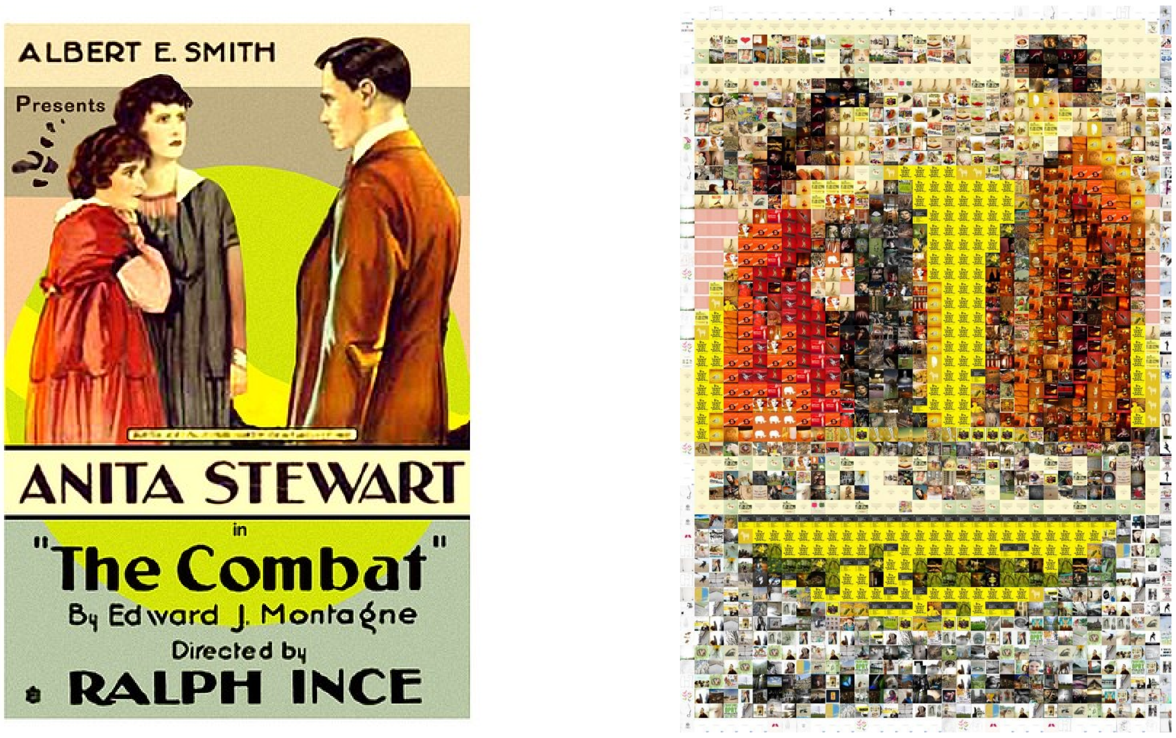


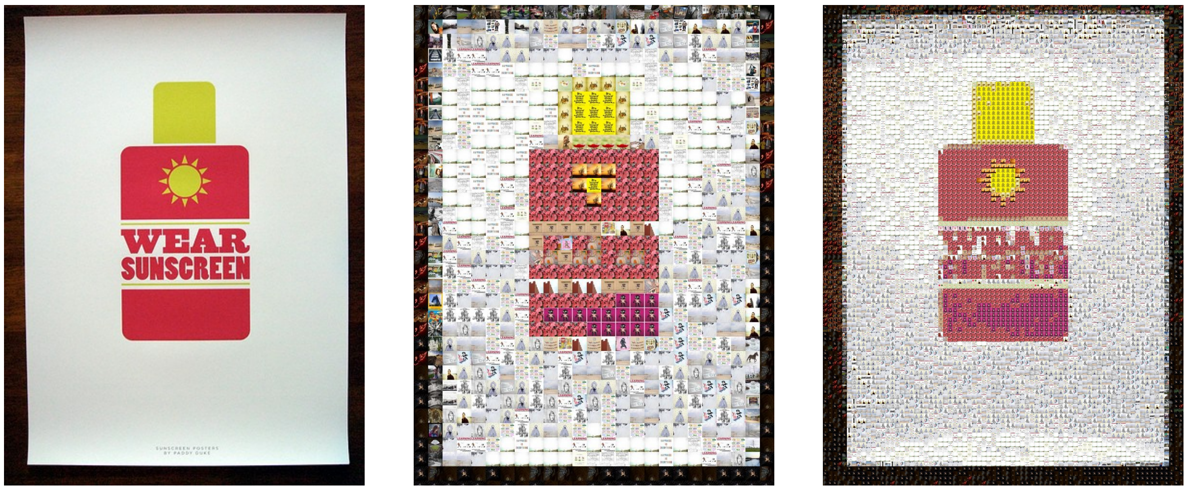
Figure 7: *The mosaic is generated with smaller tiles and an expanded image database.*

Figure 7 is generated with more tiles (1700) and using several hundred images in the database. The increased database of images significantly improves the selection of images during the tessellation process as is visually evident. Yet, this mosaic is still challenged with the repetition of images in areas where the source image is homogeneous in colour. Here, the algorithm to randomly select an image from a collection of appropriate matches to the tile has already been implemented. The yellow and pink regions are still highly repetitive as there is only one or two images that meet the criteria for replacement despite the expanded database.

6.2 *Algorithmic Improvements*

As mentioned in the above discussion regarding the early results, the primary challenges that emerged early in the work was the limited image database and large tile dimensions relative to that of the mosaic.

Colour correction was briefly implemented to confirm the functionality of the algorithm (i.e. that it was correctly assessing the colour values of the tiles and images), but ultimately removed to adhere to using unaltered images in the generation of mosaics.



(a) (b) (c)

Figure 8: *A pair of mosaics demonstrating the effects of reducing the tile size.*

In figure 8, the source image (a) is used to generate two mosaics: one of 825 tiles (b) contrasted with one of 7500 tiles (c). By visual inspection, it's clear that the reduction in tile size improves the accuracy of the mosaic.

At this point in the work, there is a distinction between the tile dimension of the source image and that of the resulting mosaic. The dimensions of the images (a), (b), and (c) are, respectively: 375 px by 500 px, 750 px by 990 px, and 1500 px by 2000 px. In create (b) from (a), the source image was broken up into 15 px by 15 px input tiles and the resulting mosaic was composed of 30 px by 30 px output tiles. Similarly, in creating (c), the input tile dimension is 5 px by 5 px and the output tile dimension is 20 px by 20 px. Theoretically, resulting mosaic would be perfectly accurate if the input tile dimension is 1 px by 1 px; hence, each pixel is represented by an image.

The challenge is to balance the visual closeness of the mosaic to the source image while maintaining input tile dimensions comparable to the source image dimensions.

It is reasonable to expect that the average colour method is not ideal; it does not capture the colour dynamics within the image and works best in colour-homogeneous image regions. Even though it represents homogeneous tiles well from the source image, there is no way to characterize the dynamics within the images in the database. Therefore, images replacing the tile may be similar homogeneous, which is a good replacement, but they may also be highly dynamic and poorly represent the tile in question.

6.3 Implementation Improvements

Although the abstract algorithms did not change very much throughout the work, the implementation was vastly revised between the first results the final results below, mainly with respect to caching, memory usage, and how the Batch API was implemented to minimize the data structures that persisted between each page request.

In generating mosaics, the general heuristic is to avoid loading data as long as possible and persisting any expensive data (i.e. image resources) for as long as possible once loaded. For example, in handling the image database, the characterization and URL of all images are loaded at the start of the tessellation process, but the actual images are not loaded into memory. When an image has been selected to replace a tile, it is loaded then, if it has not already been loaded. The image resource is retained through the tessellation process, such that future selections of this image need not reload the image resource. The approach increases memory consumption gradually and risks depleting the available memory if there is no handler to catch the exception preemptively or as it occurs to free up some memory occupied by image resources. The generator has not yet approached this upper bound yet.

6.4 Final Results

The final iteration of the generator had a database of over 11 000 images imported from Flickr using the following seven popular tags: landscape, building, portraits, animals, snow, art, and poster.

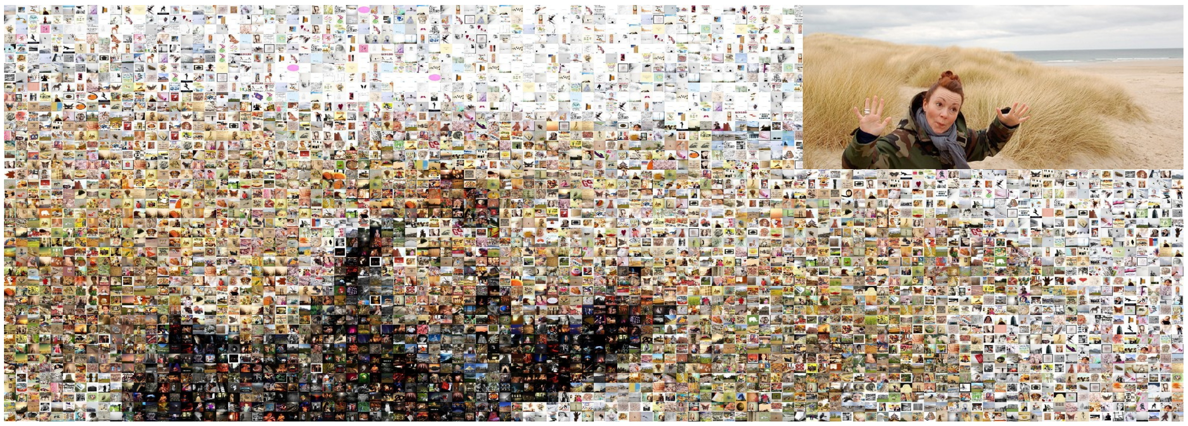


Figure 9: *A 4300-tile mosaic.*

© Chris Fabbri

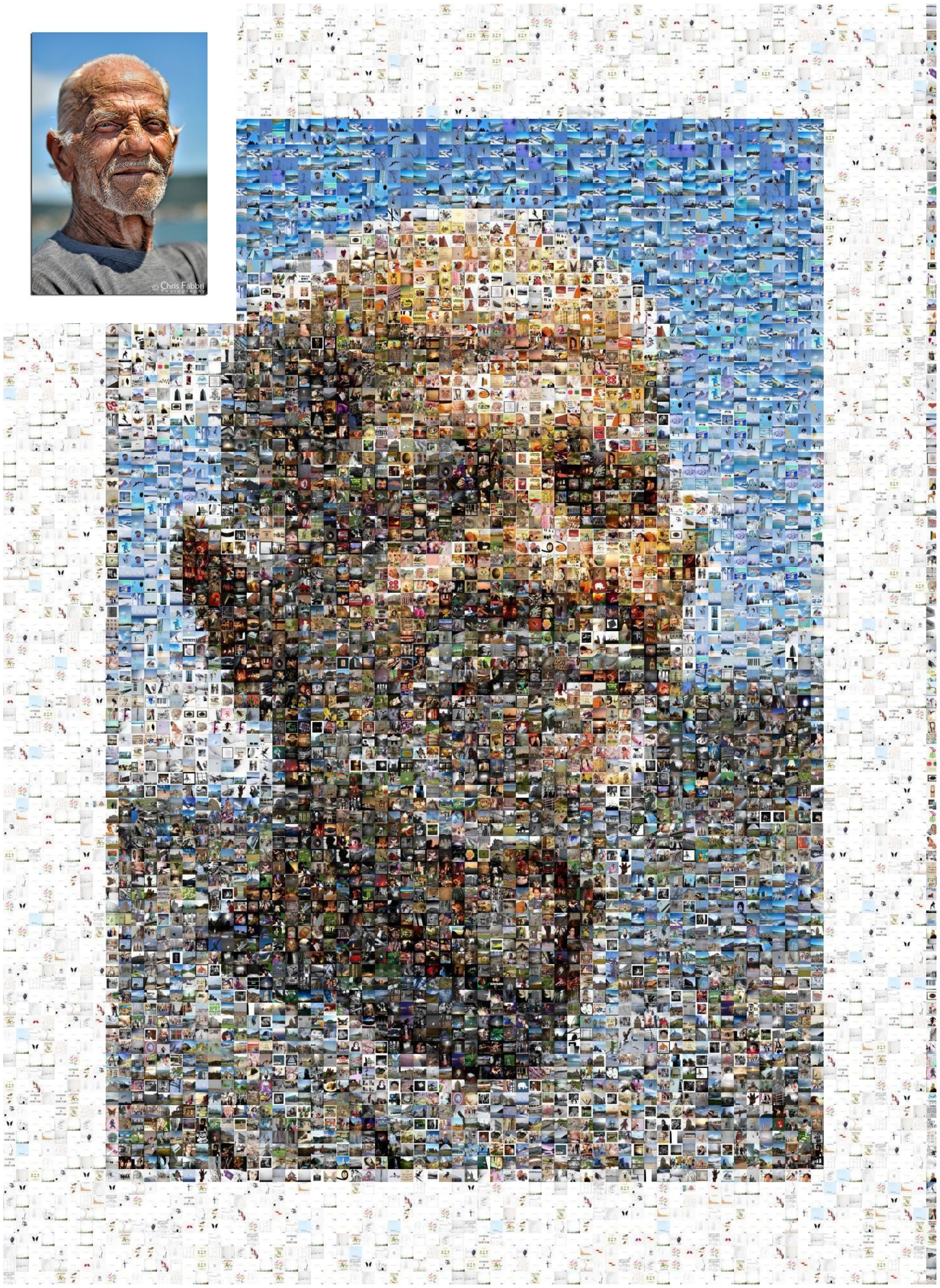


Figure 10: *A 7300-tile mosaic.*

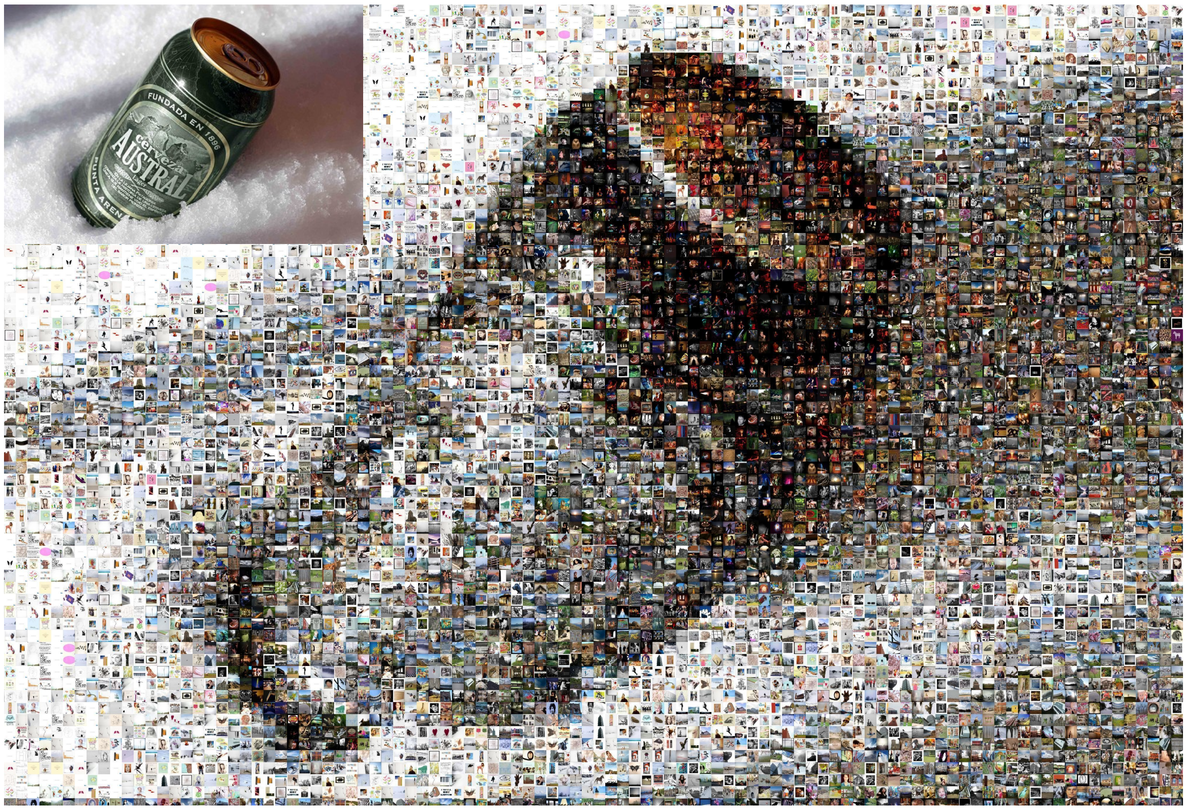


Figure 11: *A 6700-tile mosaic.*

7. Work Extensions

7.1 Different Characterization Methods

It is clear the average colour method does not capture colour dynamics within an image region. Interpolation methods and other more complicated characterizations attempt to approximate these dynamics. Using these methods greatly increases the computation of the image characterizations and complicates the way these characterizations can be compared.

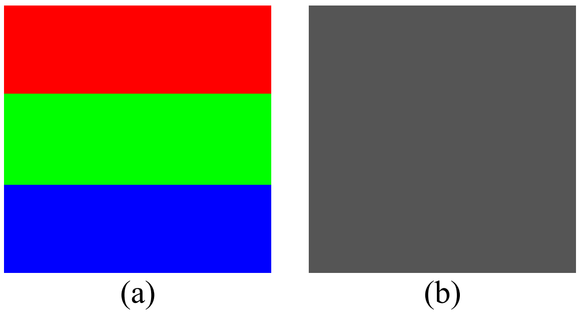


Figure 12: A*verage colour characterization does not capture colour dynamics within the image region.*

A worst case example of the colour dynamic problem is as illustrated in figure 12 and described in the following. The tile in the source image is a solid colour, 33% grey with the RGB coordinates, (0.33, 0.33, 0.33) where white is (1.0, 1.0, 1.0) and black is (0.0, 0.0, 0.0). An image that is one-third red (1.0, 0.0, 0.0), one-third green (0.0, 1.0, 0.0), and one-third blue (0.0, 0.0, 1.0) would be characterized by an average colour of exactly 33% grey.

This characterization algorithm would assume a perfect match between this colourful image and the grey tile. Using the average colour characterization method, the input tile dimension must be kept small so that these errors remain relatively minor and do not obstruct the overall look of the mosaic.

7.2 Implement the Standard Colour Difference Metric

As complicated as the CIELAB colour difference equation may be, it more appropriately compares colours and quantitatively characterizes their difference, taking into account the varying perception of different colours by the human eye. It also works in the LCh colour space which separates a colour's lightness, chroma, and hue. Parameterizing colours in this space allows the similarity metric to be more easily modified. For example, hue similarity may be more important than similarity in lightness or chroma (i.e. red images are preferred for red tiles even if they are darker or lighter than the tile).

By operating in the LCh colour space, the colour dimensions are more tangible related to the perception of colour. It will be difficult to implement this expensive calculation while suppressing the length of time required to generate mosaics.

7.3 Vastly Increased Image Database

[16] implemented a mosaic generator that could draw from millions of photos. Increasing the image database by several orders of magnitude would greatly improve the quality of generated mosaics.

8. Conclusion

In this work, the process of generating image mosaics was decomposed. A number of design possibilities were proposed for each component of the generator and some were benchmarked to inform the decisions made in implementing the various components. A multi-step method for constructing image mosaics was proposed without using colour correction or any other means of mosaic post-processing.

Source images were imported from the Internet and a database in the order of ten thousand images was created. The generator was optimized in the implementation level to generate mosaics in the order of several thousand tiles in about ten minutes. Platform limitations became apparent when handling huge mosaics of tens of megapixels with tens of thousands of tiles. The largest mosaic created by the generator was a 100-thousand-tile mosaic that was first broken into two pieces for the generator and later merged manually.

The selection of images was fairly basic using average colours to characterize both tiles and images. This worked fairly well in the case of small tiles. Yet the methods failed to capture colour dynamics within image regions and this became apparent as the tile size was increased relative to the mosaic size.

Finally, this paper outlines a web platform and framework from which the generator can function, leveraging a number of open source and free technologies.

9. References

[1] Oxford University Press. (2010, April 08). "mosaic, n. and adj.1" [Online]. Available: http://dictionary.oed.com/cgi/entry/00316181

[2] Oxford University Press. (2010, April 08). "photomosaic, n." [Online]. Available: http://dictionary.oed.com/cgi/entry/30004709

[3] J. Francis. (2010, April 08). History of Photo Mosaics. [Online]. Available: http://www.digitalartform.com/archives/2004/12/history_of_phot.html

[4] Wikipedia. (2010, April 08). Photographic mosaic. [Online]. Available: http://en.wikipedia.org/wiki/Photographic_mosaic

[5] Iran Chamber Society. (2010, April 08). . [Online]. Available: http://www.iranchamber.com/art/articles/tile_history1.php

[6] P. Wilkins. (2010, April 08). Pixelize. [Online]. Available: http://lashwhip.com/pixelize.html

[7] R. Silvers. (2010, April 08). Photomosaic. [Online]. Available: http://photomosaic.com/

[8] R. Feinson. (2010, April 08). Roy Feinson. [Online]. Available: http://www.royfeinson.com/

[9] A. Olejnik. (2010, April 08). Mosaic Creator. [Online]. http://www.aolej.com/mosaic

[10] M. Probst. (2010, April 08). Metapixel. [Online]. http://www.complang.tuwien.ac.at/schani/metapixel/

[11] M. Tesser and G. Fornasar. (2010, April 08). imosaic. [Online]. http://www.imosaic.net/

[12] R. Silvers. "Digital composition of a mosaic image," U.S. Patent 6 137 498, October 24, 2000.

[13] P. Kirchgessner. (2010, April 08). The Gimp plug-ins. [Online]. http://www.kirchgessner.net/photo-mosaic.html

[14] Public Patent Foundation. (2010, April 08). Silvers Photomosaic Patent. [Online]. http://www.pubpat.org/silvers.htm

[15] A. Finkelstein and M. Range, "Image Mosaics," Comp. Sci. Dept., Princeton Univ., Princeton, NJ, Tech. Rep. TR-574-98, 1998.

[16] D. Pavi´c, U. Ceumern and L. Kobbelt. "GIzMOs: Genuine Image Mosaics with Adaptive Tiling," Computer Graphics Forum, vol. 28, no. 8, pp. 2244–2254, 2009. doi: 10.1111/j.1467-8659.2009.01437.x

[17] Google Press Center. (2010, April 08). Images. [Online]. http://images.google.com/intl/en/press/descriptions.html#images

[18] H. Champ. (2010, April 08). 4,000,000,000. [Online]. http://blog.flickr.net/en/2009/10/12/4000000000/

[19] Wikipedia. (2010, April 08). Agile software development. [Online]. http://en.wikipedia.org/wiki/Agile_software_development

[20] DedaSys. (2010, April 08). Programming Language Popularity. [Online]. http://langpop.com/

[21] TIOBE Software BV. (2010, April 08). TIOBE Programming Community Index for April 2010. [Online]. http://www.tiobe.com/index.php/content/paperinfo/tpci/

[22] Netcraft. (2010, April 08). April 2010 Web Server Survey. [Online]. http://news.netcraft.com/archives/web_server_survey.html

[23] The Computer Language Benchmarks Game. (2010, April 08). Fastest programming language. [Online]. http://shootout.alioth.debian.org/fastest-programming-language.php?calc=chart&gcc=on&gpp=on&java=on&php=on

[24] US&V Designs. (2010, April 08). Who is Using Drupal?. [Online]. http://websites.usandv.com/who-is-using-drupal

[25] Flickr. (2010, April 08). Popular Tags on Flickr. [Online]. http://www.flickr.com/photos/tags/

[26] Wikipedia. (2010, April 08). Lab color space - CIE 1976 (L*, a*, b*) color space (CIELAB) - RGB and CMYK conversions. [Online]. http://en.wikipedia.org/wiki/Lab_color_space#RGB_and_CMYK_conversions

[27] Wikipedia. (2010, April 08). Colorfulness - Chroma in CIE 1976 L*a*b* and L*u*v* color spaces. [Online]. http://en.wikipedia.org/wiki/Colorfulness#Chroma_in_CIE_1976_L.2Aa.2Ab.2A_and_L.2Au.2Av.2A_color_spaces

[28] Wikipedia. (2010, April 08). Color difference - Delta E - CIE76. [Online]. http://en.wikipedia.org/wiki/Color_difference#CIE76

[29] J. Beis, D. G. Lowe, "Shape indexing using approximate nearest-neighbour search in high-dimensional spaces," Conf. Comp. Vision and Pattern Recognition, Puerto Rico, 1997.

[30] R. Weber, H. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," Proc 24th VLDB Conf., New York, NY, 1998.

[31] Wikipedia. (2010, April 08). kd-tree - Complexity. [Online]. http://en.wikipedia.org/wiki/Kd_tree#Complexity

Appendix A. Mosaic Generator Code

The following is the Drupal code for the mosaic generator module.

mosaic.info

```
; $Id$
name = Image Mosaic Generator
description = Allows users to generate image mosaics from a gallery of images.
core = 6.x
package = Thesis
```

mosaic.settings.inc

```
<?php
// $Id$

/**
 * @file
 * Defines constants for mosaic generator.
 */

// Content types
define('MOSAIC_TYPE', 'mosaic');
define('IMAGE_TYPE', 'image');

// Field names
define('IMAGE_FIELD', 'field_image');
define('CHAR_FIELD', 'field_char');

// Tiling constants
define('INPUT_TILE_SIZE', 5);
define('OUTPUT_TILE_SIZE', 40);
define('TILE_COLOR_DISTANCE', 2000);
define('COLOUR_CORRECTION', 50);

// Other constants
define('SEARCH_MAX_IMAGES', 500);
define('IMPORT_LIMIT', 10);
define('TESSELATE_LIMIT', 100);
define('QUERY_LIMIT', 20000);
```

mosaic.module

```
<?php
// $Id$

/**
 * @file
 * Allows users to generate image mosaics from a gallery of images.
 */

require_once(drupal_get_path('module', 'mosaic') . '/mosaic.settings.inc');

/**
 * Implementation of hook_perm().
 */
function mosaic_perm() {
  return array('create image mosaics');
}

/**
 * Implementation of hook_menu().
 */
function mosaic_menu() {
  $items = array();
  $items['mosaic_generate/%node'] = array(
    'title' => 'Generate a tile composite',
    'page callback' => 'mosaic_generate',
    'page arguments' => array(1),
    'access arguments' => array('create image mosaics'),
    'type' => MENU_CALLBACK,
    'file' => 'mosaic.generate.inc',
  );
  $items['admin/mosaic'] = array(
    'title' => 'Mosaic',
    'description' => 'Configure mosaic generator.',
    'position' => 'left',
    'weight' => -99,
    'page callback' => 'system_admin_menu_block_page',
    'access arguments' => array('administer site configuration'),
    'file' => 'system.admin.inc',
    'file path' => drupal_get_path('module', 'system'),
  );
  $items['admin/mosaic/import'] = array(
    'title' => 'Import images',
    'description' => 'Create image library from external source.',
    'page callback' => 'drupal_get_form',
    'page arguments' => array('mosaic_import_images'),
    'access arguments' => array('administer site configuration'),
    'type' => MENU_NORMAL_ITEM,
    'file' => 'mosaic.admin.inc',
  );
  $items['admin/mosaic/settings'] = array(
    'title' => 'Mosaic settings',
    'description' => 'Change mosaic generator settings.',
    'page callback' => 'drupal_get_form',
    'page arguments' => array('mosaic_admin_settings'),
    'access arguments' => array('administer site configuration'),
    'type' => MENU_NORMAL_ITEM,
    'file' => 'mosaic.admin.inc',
  );
  return $items;
}

/**
 * Implementation of hook_nodeapi().
 */
function mosaic_nodeapi(&$node, $op, $a3 = NULL, $a4 = NULL) {
  if(strcmp($node->type, IMAGE_TYPE) == 0) {
    switch ($op) {
      case 'view':
        if($a4 === true) {
          drupal_add_css(drupal_get_path('module', 'mosaic') . '/mosaic-char.css');

          $node->content['mosaic_link'] = array(
            '#value' => l(t('Create a mosaic from this image'), 'mosaic_generate/'
. $node->nid, array('attributes' => array('class' => 'mosaic-link'))),
            '#weight' => -10,
          );
        }
        break;
      case 'presave':
        if (!is_null($node->field_image[0]['fid'])){
          $color = _mosaic_characterize_image(imageapi_image_open($node-
>field_image[0]['filepath']));
          unset($color['alpha']);
          $node->field_char[0]['value'] = serialize($color);
        }
        break;
    }
  }
}
```

```php
/**
 * Helper function to mosaic_nodeapi(). Resamples image to a 1 px by 1 px space
 * and returns the average colour.
 */
function _mosaic_characterize_image($image) {
  imageapi_image_resize($image, 1, 1);
  return imagecolorsforindex($image->resource, imagecolorat($image->resource, 0,
0));
}
```

mosaic.admin.inc

```php
<?php
// $Id$

/**
 * @file
 * Allows users to generate image mosaics from a gallery of images.
 */

require_once(drupal_get_path('module', 'mosaic') . '/mosaic.settings.inc');

/**
 * Define form to import and create image library.
 */
function mosaic_import_images() {
  $form = array();
  $form['tags'] = array(
    '#type' => 'textfield',
    '#title' => t('Search terms'),
    '#default_value' => 'landscape, building, portraits, animals, snow, art, poster
',
    '#size' => 60,
    '#maxlength' => 64,
    '#description' => t('These terms will be used to search for images.'),
  );
  $form['num'] = array(
    '#type' => 'textfield',
    '#title' => t('Number of images per term'),
    '#default_value' => SEARCH_MAX_IMAGES,
    '#size' => 60,
    '#maxlength' => 64,
  );
  $form['submit'] = array(
    '#type' => 'submit',
    '#value' => t('Import'),
  );
  return $form;
}

/**
 * Submission hook to process form submitted from mosaic_import_images().
 *
 * Initiates batch process to import images.
 */
function mosaic_import_images_submit($form, &$form_state) {
  $num = intval($form_state['values']['num']);
  $tags = explode(', ', $form_state['values']['tags']);

  $urls = array();
  if ($num >= SEARCH_MAX_IMAGES) {
    foreach ($tags as $tag) {
      $tag_count[_mosaic_image_search_pages($tag)] = $tag;
    }
    for (; $num >= SEARCH_MAX_IMAGES; $num -= SEARCH_MAX_IMAGES){
      foreach ($tag_count as $pages => $tag) {
        $urls = array_merge($urls, _mosaic_image_search($tag, mt_rand(1, $pages)));
      }
      $page++;
    }
  }
  if ($num > 0) {
    foreach ($tags as $tag) {
      $pages = _mosaic_image_search_pages($tag, $num);
      $urls = array_merge($urls, _mosaic_image_search($tag, mt_rand(1, $pages), $nu
m));
    }
  }

  $batch = array(
    'operations' => array(array('_mosaic_import_images_batch', array($urls))),
    'finished' => '_mosaic_import_images_batch_finished',
    'title' => t('Importing images'),
    'init_message' => t('The image import process is beginning.'),
    'progress_message' => t('Importation in progress...'),
    'error_message' => t('The importation process encountered an error.'),
    'file' => drupal_get_path('module', 'mosaic') . '/mosaic.admin.inc',
  );
  batch_set($batch);
  // batch_process() not needed here because this is a form submit handler;
  // the form API will detect the batch and call batch_process() automatically.
}

/**
 * Define form to adjust the settings.
 */
function mosaic_admin_settings() {
  $form = array();
  return $form;
}

/**
 * Displays completion message at the end of the importation process started by
 * mosaic_import_images_batch().
 */
function _mosaic_import_images_batch_finished($success, $results, $operations) {
  if ($success) {
    $message = format_plural($results['images'], 'One image imported.', '@count ima
ges imported.');
  } else {
    $error_operation = reset($operations);
    $message = t('An error occurred while processing %error_operation with argument
s: @arguments', array('%error_operation' => $error_operation[0], '@arguments' => pr
int_r($error_operation[1], TRUE)));
    watchdog('mymodule', 'An error occurred while processing %error_operation with
arguments: @arguments', array('%error_operation' => $error_operation[0], '@argument
s' => print_r($error_operation[1], TRUE)), WATCHDOG_ERROR);
  }
  drupal_set_message($message);
  drupal_goto();
}

/**
 * Imports IMPORT_LIMIT images as part of the batch process started by
 * mosaic_import_images_batch().
 */
function _mosaic_import_images_batch($urls = array(), &$context) {
  if (!isset($context['sandbox']['progress'])) {
    $context['sandbox']['progress'] = 0;
    $context['sandbox']['max'] = count($urls);
    $context['message'] = t('Now importing image %progress of %max', array('%progre
ss' => $context['sandbox']['progress'], '%max' => $context['sandbox']['max']));
    $context['finished'] = 0;
  } else {
    $limit = IMPORT_LIMIT;
    $counter = 0;
    while ($counter != $limit && $context['sandbox']['progress'] != $contex
t['sandbox']['max']) {
      _mosaic_import_images($urls[$context['sandbox']['progress']]);

      $counter++;
      $context['sandbox']['progress']++;
      $context['message'] = t('Now importing image %progress of %max', array('%prog
ress' => $context['sandbox']['progress'], '%max' => $context['sandbox']['max']));
      $context['results']['images'] = $context['sandbox']['progress'];
    }

    if ($context['sandbox']['progress'] != $context['sandbox']['max']) {
      $context['finished'] = $context['sandbox']['progress'] / $context['sandbox']
['max'];
    }
  }
}

/**
 * Imports a single image to Drupal given a source URI.
 */
function _mosaic_import_images($src) {
  global $user;
  $uid = $user->uid;

  $node = new stdClass();
  $node->type = 'image';
  $node->uid = $uid;
  $node->status = 1;

  $dst = file_directory_path() . '/' . md5($src) . '.jpg';
  $result = copy($src, $dst);
  if($result === false) {
    return false;
  }

  // Create a Drupal file record for the new image.
  $file = new stdClass();
  $file->filename = basename($dst);
  $file->filepath = $dst;
  $file->filemime = file_get_mimetype($dst);
  $file->filesize = filesize($dst);
  $file->uid = $uid;
  $file->status = FILE_STATUS_PERMANENT;
  $file->timestamp = time();
  drupal_write_record('files', $file);
  $file->fid = db_result(db_query("SELECT fid FROM {files} WHERE filepath = '%s'",
$file->filepath));

  // Insert image into node.
  $node->field_image = array(array(
      'fid' => $file->fid,
      'list' => 1,
      'uid' => $uid,
      'filename' => $file->filename,
      'filepath' => $file->filepath,
      'filemime' => $file->filemime,
      'filesize' => $file->filesize,
      'status' => $file->status,
      'timestamp' => $file->timestamp
  ));
  $node->field_char = array(array('value' => ''));
  node_save($node);

  return true;
}
```

```php
/**
 * Returns the number of pages of photos in Flickr for a specific search tag.
 */
function _mosaic_image_search_pages($tag, $per_page = SEARCH_MAX_IMAGES){
  $results = _mosaic_image_search_flickr($tag, 1, $per_page);
  return $results['photos']['pages'];
}

/**
 * Returns an array of URIs to Flickr images given a search tag.
 */
function _mosaic_image_search($tag, $page = 1, $per_page = SEARCH_MAX_IMAGES){
  $results = _mosaic_image_search_flickr($tag, $page, $per_page);

  // Ensure request was successful.
  if ($results['stat'] !== 'ok'){
    return false;
  }

  $photos = $results['photos']['photo'];
  $urls = array();
  foreach($photos as $photo){
    $farm   = $photo['farm'];
    $server = $photo['server'];
    $id     = $photo['id'];
    $secret = $photo['secret'];
    $urls[] = "http://farm$farm.static.flickr.com/$server/${id}_$secret.jpg";
  }
  return $urls;
}

/**
 * Returns an array of Flickr results given a search tag.
 */
function _mosaic_image_search_flickr($tag, $page = 1, $per_page = SEARCH_MAX_IMAGES
){
  $params = array(
    'api_key'   => '5c0b0b7235116793a29e8bbd1f516f92',
    'method'    => 'flickr.photos.search',
    'tags'      => $tag,
    'format'    => 'php_serial',
    'per_page'  => $per_page,
    'page'      => $page
  );
  $encoded_params = array();
  foreach ($params as $k => $v){
    $encoded_params[] = urlencode($k).'='.urlencode($v);
  }

  // Call the API and decode the response.
  $url = "http://api.flickr.com/services/rest/?".implode('&', $encoded_params);
  $response = file_get_contents($url);
  $results = unserialize($response);

  return $results;
}
```

mosaic.generate.inc

```php
<?php
// $Id

/*
 * Generate a tile composite from a given node object.
 */
function mosaic_generate($node) {
  // Check node is of content type that can be used as tile composite source.
  if (strcmp($node->type, IMAGE_TYPE) != 0) {
    drupal_not_found();
    return;
  }

  // Create tile composite image.
  $image_path = $node->field_image[0]['filepath'];
  list($image, $tiles) = _mosaic_generate_ops($image_path);

  // Save source's nid.
  variable_set('mosaic_generate_nid', $node->nid);

  $batch = array(
    'operations' => array(array('_mosaic_generate_batch', array($image, $tiles))),
    'finished' => '_mosaic_generate_batch_finished',
    'title' => t('Tesselating mosaic tiles'),
    'init_message' => t('The tesselation process is beginning.'),
    'progress_message' => t('Tesselation in progress...'),
    'error_message' => t('The tesselation process encountered an error.'),
    'file' => drupal_get_path('module', 'mosaic') . '/mosaic.generate.inc',
  );
  batch_set($batch);
  batch_process('');
}

function _mosaic_generate_batch_finished($success, $results, $operations) {
  if ($success) {
    // Save new mosaic image.
    global $user;
    $uid = $user->uid;

    // Create a Drupal file record for the mosaic image.
    $file_drupal_path = $results['image']->source;
    $mime = 'image/jpeg';

    $file = new stdClass();
    $file->filename = basename($file_drupal_path);
    $file->filepath = $file_drupal_path;
    $file->filemime = $mime;
    $file->filesize = filesize($file_drupal_path);

    $file->uid = $uid;
    $file->status = FILE_STATUS_PERMANENT;
    $file->timestamp = time();
    drupal_write_record('files', $file);
    $file->fid = db_result(db_query("SELECT fid FROM {files} WHERE filepath = '%s'
", $file->filepath));

    // Save the mosaic image as a new node.
    $node_new = new stdClass();
    $node_new->type = MOSAIC_TYPE;
    $node_new->uid = $uid;
    $node_new->status = 1;
    $node_new->field_image = array(array(
        'fid' => $file->fid,
        'title' => basename($file->filename),
        'filename' => $file->filename,
        'filepath' => $file->filepath,
        'filesize' => $file->filesize,
        'mimetype' => $mime,
        'description' => basename($file->filename),
        'list' => 1,
    ));
    $node_new->field_source = array(array('nid' => variable_ge
t('mosaic_generate_nid', 0)));
    node_save($node_new);

    // Clear source's nid.
    variable_del('mosaic_generate_nid');

    // Set success message.
    $message = format_plural($results['count'], 'One tile tesselated.', '@count til
es tesselated.');
    $seconds = intval($results['time']) / 1000;
    $min = str_pad(intval($seconds / 60), 2, '0', STR_PAD_LEFT);
    $sec = str_pad($seconds % 60, 2, '0', STR_PAD_LEFT);
    $timer = t('Tesselation took @min:@sec.', array('@min' => $min, '@sec' => $sec)
);
    drupal_set_message($message);
    drupal_set_message($timer);
    drupal_set_message($results['time']);

    // Go to the new tile composite node.
    drupal_goto('node/' . $node_new->nid);
  } else {
    $error_operation = reset($operations);
    $message = t('An error occurred while processing %error_operation with argument
s: @arguments', array('%error_operation' => $error_operation[0], '@arguments' => pr
int_r($error_operation[1], TRUE)));
    watchdog('mymodule', 'An error occurred while processing %error_operation with
arguments: @arguments', array('%error_operation' => $error_operation[0], '@argument
s' => print_r($error_operation[1], TRUE)), WATCHDOG_ERROR);
    drupal_set_message($message);
  }
}

function _mosaic_generate_batch($image, $tiles, &$context) {
  if (!isset($context['sandbox']['progress'])) {
    $results = array();
    $image = imageapi_image_open($image->source);
    $results['num_width'] = intval($image->info['width'] / INPUT_TILE_SIZE);
    $results['num_height'] = intval($image->info['height'] / INPUT_TILE_SIZE);
    $results['width_counter'] = 0;
    $results['input_width'] = 0;
    $results['input_height'] = 0;
    $results['output_width'] = 0;
    $results['output_height'] = 0;
    $results['image'] = clone $image;
    $results['image']->resource = imagecreatetruecolor($results['num_width'] * OUTP
UT_TILE_SIZE, $results['num_height'] * OUTPUT_TILE_SIZE);
    $results['image']->source = file_directory_path() . '/' . md5(mt_rand()) . '.jp
g';
    imageapi_image_close($results['image']);
    $results['time'] = 0;

    $context['sandbox']['progress'] = 0;
    $context['sandbox']['max'] = $results['num_width'] * $results['num_height'];
    $context['message'] = t('Now tesselating tile %progress of %max', array('%progr
ess' => $context['sandbox']['progress'], '%max' => $context['sandbox']['max']));
    $context['results'] = $results;
    $context['finished'] = 0;
  } else {
    $results = $context['results'];
    $image = imageapi_image_open($image->source);
    $results['image'] = imageapi_image_open($results['image']->source);
    $limit = TESSELATE_LIMIT;
    $limit_counter = 0;
    $tile_avg_color_res = imageapi_gd_create_tmp($image, 1, 1);
    $tile_composition_res = $results['image']->resource;
    timer_start('mosaic_generate');
    while ($limit_counter != $limit && $context['sandbox']['progress'] != $contex
t['sandbox']['max']) {
      _mosaic_generate($image, $tiles, $results, $tile_avg_color_res, $tile_composi
tion_res);

      $limit_counter++;
      $results['width_counter']++;

      if($results['width_counter'] == $results['num_width']) {
        $results['width_counter'] = 0;
        $results['input_width'] = 0;
        $results['output_width'] = 0;

        $results['input_height'] += INPUT_TILE_SIZE;
        $results['output_height'] += OUTPUT_TILE_SIZE;
      } else {
        $results['input_width'] += INPUT_TILE_SIZE;
        $results['output_width'] += OUTPUT_TILE_SIZE;
      }

      $context['sandbox']['progress']++;
      $context['message'] = t('Now tesselating tile %progress of %max', array('%pro
gress' => $context['sandbox']['progress'], '%max' => $context['sandbox']['max']));
      $results['count'] = $context['sandbox']['progress'];
    }
    $timer = timer_stop('mosaic_generate');
    $results['time'] += $timer['time'];
    $results['image']->resource = $tile_composition_res;
    imageapi_image_close($results['image']);
    $context['results'] = $results;

    if ($context['sandbox']['progress'] != $context['sandbox']['max']) {
      $context['finished'] = $context['sandbox']['progress'] / $context['sandbox']
['max'];
    }
  }
}

function _mosaic_generate($image, $tiles, $results, $tile_avg_color_res, &$tile_com
position_res) {
  $num_width = $results['num_width'];
  $num_height = $results['num_height'];

  $input_width = $results['input_width'];
  $input_height = $results['input_height'];
  $output_width = $results['output_width'];
  $output_height = $results['output_height'];

  imagecopyresampled($tile_avg_color_res, $image->resource, 0, 0, $input_width, $in
put_height, 1, 1, INPUT_TILE_SIZE, INPUT_TILE_SIZE);
  $color = imagecolorsforindex($tile_avg_color_res, imagecolorat($tile_avg_color_re
s, 0, 0));
  $tile_key = _mosaic_generate_select_tile($color, $tiles);

  $tile_image = $tiles[$tile_key]['image'];
  if(is_null($tile_image)) {
    $tiles[$tile_key]['image'] = imageapi_image_open($tiles[$tile_key]['path']);
    imageapi_image_resize($tiles[$tile_key]['image'], OUTPUT_TILE_SIZE, OUTPUT_TILE
_SIZE);
    $tile_image = $tiles[$tile_key]['image'];
  }
  $tile_info = $tile_image->info;

  $tilecopy = imagecreatetruecolor(OUTPUT_TILE_SIZE, OUTPUT_TILE_SIZE);
  imagecopy($tilecopy, $tile_image->resource, 0, 0, 0, 0, OUTPUT_TILE_SIZE, OUTPUT_
TILE_SIZE);

  //$masked = imagecreatetruecolor(OUTPUT_TILE_SIZE, OUTPUT_TILE_SIZE);
  //imagefill($masked, 0, 0, imagecolorallocate($masked, $color['red'], $color['gre
en'], $color['blue']));
  //imagecopymerge($tilecopy, $masked, 0, 0, 0, 0, OUTPUT_TILE_SIZE, OUTPUT_TILE_SI
ZE, COLOUR_CORRECTION);

  imagecopy($tile_composition_res, $tilecopy, $output_width, $output_height, 0, 0,
OUTPUT_TILE_SIZE, OUTPUT_TILE_SIZE);

  return $tile_composition_res;
}

/**
 * Execute tiling algorithm based on source image.
 */
function _mosaic_generate_ops($image_path) {
  // Load collection of tiles.
  $tiles = array();
  $nodes = db_query("SELECT nid, vid, type FROM {node} WHERE type = '%s' LIMIT %d",
'image', QUERY_LIMIT);
  $node = db_fetch_object($nodes);
  while ($node = db_fetch_object($nodes)) {
    content_load($node);
    $tile = unserialize($node->field_char[0]['value']);
    unset($tile['alpha']);
    $tile['path'] = $node->field_image[0]['filepath'];
    $tiles[] = $tile;
  }

  usort($tile, '_mosaic_generate_ops_sort_tiles');

  // Calculate image variables.
  $image = imageapi_image_open($image_path);
  $num_width = (int) $image->info['width'] / INPUT_TILE_SIZE;
  $num_height = (int) $image->info['height'] / INPUT_TILE_SIZE;

  return array($image, $tiles);
}

function _mosaic_generate_ops_sort_tiles($tile1, $tile2) {
  return (($tile1['red'] == $tile2['red']) ? 0 : ($tile1['red'] < $tile2['red']) ?
-1 : 1);
}

/**
 * Select tile with closest average color.
 */
function _mosaic_generate_select_tile($color, $tiles) {
  $tiles_close = array();
  $tile_closest = NULL;
  $tile_closest_distance = PHP_INT_MAX;
  foreach ($tiles as $key => $tile) {
    $tile_distance = pow($tile['red'] - $color['red'], 2) +
                     pow($tile['green'] - $color['green'], 2) +
                     pow($tile['blue'] - $color['blue'], 2);
    if ($tile_distance < $tile_closest_distance) {
      $tile_closest = $key;
      $tile_closest_distance = $tile_distance;
    }
    if ($tile_distance < TILE_COLOR_DISTANCE) {
      $tiles_close[] = $key;
    }
  }
  if (empty($tiles_close)) {
    return $tile_closest;
  } else {
    return $tiles_close[array_rand($tiles_close)];
  }
}
```