Diminishing Reality

Andreas Hackl and Helmut Hlavacs(✉)

Entertainment Computing Research Group, University of Vienna, Vienna, Austria
a1308402@unet.univie.ac.at, helmut.hlavacs@univie.ac.at
http://entertain.univie.ac.at/~hlavacs/

Abstract. We explore ways of removing objects from live video feeds in augmented reality-like use cases, using a method for inpainting unwanted objects using previously captured visuals of the surrounding environment. In contrast to related previous work, this approach can completely retain and reproduce hidden objects in all their detail. We describe the approach and detail results from our evaluation.

1 Introduction

Augmented reality (AR), the "augmentation" of a view of the physical world, has found its way into normal peoples' everyday lives, with smartphones becoming ubiquitous devices in the developed world. For entertainment purposes it found widespread use in humorous overlays over peoples' faces, or even real-time alteration of their physical appearances [16], in social networking applications like Snapchat or SNOW, and in video games that use real environments as backdrops to render in-game graphics onto.

On a different front, technologies and applications for the removal of unwanted objects from still images and even video, generally described as "inpainting", have existed for some time, mainly for the improvement of prerecorded media. They allow removal of a person from a scenic photograph, the seamless stitching panoramas, or removal of strings from a video that should not be visible to viewers. These technologies have also found their way into software available to standard consumer software, most notably in Adobe Photoshop with its "content-aware fill" feature, based on the PatchMatch [1,7] algorithm.

By combining both of these technologies, objects could be made to vanish from a user's view in real-time. In doing so, reality would not be augmented with additional information, but real information would be reduced, resulting in *diminished reality* (DR). The goal of this work is to devise and implement an efficient and lightweight DR approach to this on the Android platform.

2 Related Work

While the subject of inpainting is well researched and continues to be subject of new research, there have not been many applications so far with the aim

E. Clua et al. (Eds.): ICEC 2018, LNCS 11112, pp. 28–39, 2018.
https://doi.org/10.1007/978-3-319-99426-0_3

of, or any functionality similar to, the concept of diminished or reduced reality proposed in this work.

A single application has been developed as part of very promising research into exactly this topic in 2010, with viable results, on even then-current hardware, named "Diminished Reality" [12]. The rendering pipeline proposed by Broll et al. is very similar to the one chosen by us, but differs in that it uses a patch-based approach, similar to PatchMatch, that operates entirely on a frame-by-frame basis, without considering any earlier input. The visual fidelity and accuracy of its inpainting is therefore largely dependent on the input image and surroundings of the object to be inpainted.

Research into the broader topic of general video inpainting by Newson et al. [14] utilises a sort of memory, while still being patch-based, by considering patches from multiple frames of a video to inpaint objects. It manages to produce very convincing results, but is not efficient enough for real-time use.

Other approaches to improve accuracy and fidelity in real-time inpainting, compared to the traditional patch-based ones, have originally been considered, but turned out to not be quite applicable to this task.

Neural networks in particular have in recent years become sophisticated enough, and consumer hardware powerful and cheap enough, to make the implementation and use of powerful networks for various image generation and alteration tasks viable.

An application of Generative Adversarial Networks (GAN) to the problem of texture synthesis and style transfer by Ulyanov et al. [17] provides an entirely feed-forward way of generating images, once a model has been trained. Because of its promises of very fast image generation, it has been tested in consideration of its viability as a base for real-time inpainting.

Using a relatively inexpensive GPU (Nvidia GTX 960 with 2 GB VRAM) generation of images was possible at an average of 20 frames per second with a resolution of 512×512 pixels, which makes it fast enough for real-time use. It can however only generate textures that were trained into a model beforehand. Although one model can be trained for multiple textures at once, without much increase in needed computation time or loss in image quality, is a very limiting factor. Training one such model on the same hardware limits the maximum input size to 256×256 pixels (although not necessarily the output size), due to the high VRAM requirements of the training process, and takes more than 20 min. In real-world environments, where any kind of texture could appear in a user's surroundings, this training process would therefore need to be on an as-needed basis, for which this time frame is still far too long. It also does unfortunately not provide any consistency between generated inputs, or any facility to blend generated textures into an existing background texture (cf. Fig. 1).

Another promising project based on Convolutional Neural Networks (CNN) by Yang et al. [20], with the explicit purpose of high-quality inpainting of images, similar to PatchMatch but with better results, has recently managed to reduce processing time to only half a minute per image [20]. That means it might soon be usable even for real-time purposes, but at this time it is not.

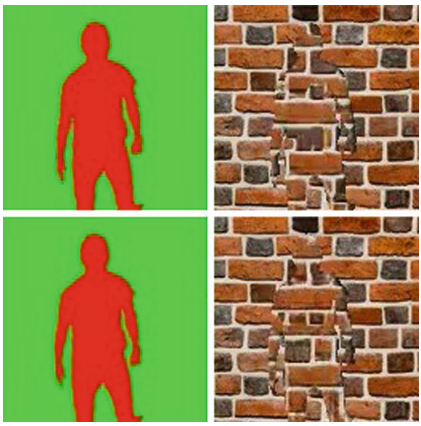


Fig. 1. Two frames, foreground (red) generated by [17] over a static image of the image the model was trained. (Color figure online)

Since this kind of method poses a great challenge in making it fit the purpose of real-time inpainting, in either some kind of extensive pre- or post-processing being necessary or being simply too slow overall, while still offering behaviour, strengths and drawbacks very similar to patch-based ones, it has been discarded in favour of the method described here.

3 Our Inpainting Approach

The image processing pipeline for inpainting relies on a simple in-memory structure that can be used to store the visual appearance of the environment around a point in space and later retrieve parts of it to fill in regions in images. It consists of five core parts:

1. **Input:** Capturing of frames from the camera. Every captured frame potentially needs to be put into a spacial relation to other frames and/or the positioning of the capturing device in space. For this purpose, at the time of frame capture, the orientation of the device is queried and stored alongside it, adding other metadata, such as a measurement of time (e.g. timestamp or frame number).
2. **Object detection:** Detection of objects and generation of regions inside of a frame that are to be inpainted. An object detection algorithm is applied to the newly captured frame, identifying one or more regions inside it that contain objects which the user might want to vanish. To make description of arbitrary regions easy, and because later stitching of frames requires it anyway, they are stored as monochrome bitmaps.
3. **Frame storage:** Input frames in which no objects of interest have been detected are candidates for use in inpainting to fill in objects and as such need to be stored in a *frame store* for later use. Frames are stored uncompressed

in a 2D array-like structure with each cell representing a few degrees of yaw and pitch, essentially subdividing the range of possible device orientations (disregarding roll) into a grid. Each cell can hold one frame at most at any given time, limiting the maximum amount of frames to be stored in a given range of orientation, therefore preventing the storage of too many too similar frames.

4. **Frame search:** For input frames in which objects have been detected that are to be inpainted, a frame that is potentially suitable to fill in the region(s) generated by the object detection step needs to be found. As comparison of actual frame content, of potentially many pairs of frames, is computationally too expensive to do for each input frame in real-time, a frame is searched inside the frame store by the most similar orientation to that of the input frame.
5. **Stitching:** If a frame suitable for inpainting has been found in the frame store, a single frame is stitched together using the newly captured frame, the frame found in the frame store, and an object mask. The stored frame needs to be aligned to the input frame as precisely as possible and then blended together according to the object mask.

Other than simple, direct storage of input frames into memory during runtime, there is no need for any lengthy preprocessing (e.g. pre-generating a sphere map) in order to work in any given environment, as frames are stored and accessed independently of each other.

4 Implementation

An application implementing the proposed method has been developed for Android using OpenCv4Android [2], which provides Java bindings for OpenCV's C++-based API. It is largely based on the desktop OpenCV Java API, with several Android-specific convenience features added.

OpenCV provides basic functions for image manipulation (e.g. translation, colour manipulation, conversion), as well as more advanced features typical for computer vision (e.g. feature extraction), both of which this project heavily relies on. Because it is a thin wrapper around natively compiled C++ code, it avoids many potential performance problems caused by the JVM and its garbage collection. It being a standard API available on many platforms also makes porting applications using it easy.

The architecture is kept largely modular, with the main roles in the program separated out into their own classes/interfaces.

4.1 Input

The camera provides the image as both 4-channel RGBA (red, green, blue and alpha channels) and 1-channel monochrome data. A sensor manager class provides a combined 3-component orientation vector, computed from the "gravity" (virtual device based on accelerometer and gyroscope) and "magnetic field"

(uncalibrated magnetometer data) sensors. Data provided by the magnetometer, by nature of the hardware built into smartphones, reflects changes in orientation nearly instantly but is very noisy, sometimes jumping rapidly between up to 10° above/below the expected value. To get more stable readings, for every input value, the average of the last 100 values (including the new one) is calculated using a moving window approach.

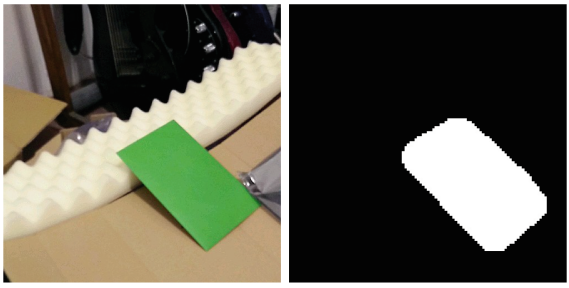
4.2 Object Detection and Mask Generation

Video frames are passed to a mask generator class that creates a monochrome mask for detected objects, which are masked white. All other pixels are masked black.

There are two approaches for object detection implemented. Colour keying computes a mask by converting the RGB colour image to HSV (hue, saturation and value) and generates a greyscale image, setting all pixels in it to white if their corresponding pixels' HSV values are within a given range. HSV is used because RGB makes it difficult to define a range of similar looking colours.

Because this is prone to create very holey masks and include undesirable single pixels, the resulting mask is blurred using a simple box blur with a kernel size of 10 pixels and then all pixels above a value of 50 (in the range of 0–255) are set to pure white, all below to pure black (cf. Fig. 2). This produces more coherent, hole-less areas and discards single stray pixels.

For performance reasons, input is (optionally) downscaled before processing and the result upscaled afterwards.

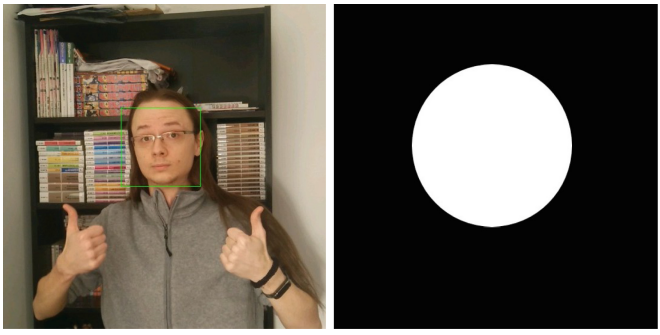


(a) Input frame (b) Generated mask

Fig. 2. Generating a mask using colour keying, with downscaling. (Color figure online)

The second detector is a face detector and uses OpenCV's `CascadeClassifier` and its `detectMultiScale` method, which uses Haar feature [18] or LBP-based (local binary pattern [13]) cascade classifiers to detect objects in a greyscale image. LBP cascade descriptors were chosen because the detection process is computationally less intensive than the Haar feature-based one, even if slightly more inaccurate.

For each detected object, the function returns a rectangle describing its position inside the image and its size. Because these rectangles often do not completely cover the object, instead of simply filling them with pure white to create a mask, a circle is drawn over each one, with the centre being that of the rectangle and the radius the average of its sides' lengths. This results in circles with approximately double the diameter of the rectangles (cf. Fig. 3).



(a) `detectMultiScale` output (b) Generated mask

Fig. 3. Generating a mask from a rectangle describing a face detected using an LBP cascade.

4.3 Frame Storage and Search

The `FrameStore` utilises a basic fixed-size 2D array of `VideoFrame`s, with each element representing a certain amount of degrees of yaw and pitch. Unoccupied elements are initialised to `null` to conserve memory.

To store a frame input to the frame store via its `replace` method, the X and Y indices in the array are computed using the frame's orientation. The new frame and the frame previously stored there, if there is one, are passed to the `shouldReplace` method of an `IReplacementPolicy` provided to the frame store, to determine if the old frame should really be replaced (hence the method name `replace`). A copy of the new frame is then stored in the array.

The replacement policy implemented compares the two frames' `frameNumber`s (a value incremented every frame) and indicates to replace an old frame, only if their difference is large enough. This prevents frames from being copied and discarded many times per second, for a similar device orientation, which can lead to large memory consumption and even crashing of the application, depending on available memory and garbage collection frequency of the JVM. To search for the nearest frame to a given orientation, the `getNearest` method again calculates the corresponding array indices, and then searches from there outwards until a frame is found.

4.4 Stitching

If a suitable frame has been found in the frame store, the input frame and stored frame need to be *stitched* together, to fill regions marked in the generated mask and produce a single, coherent image.

Since orientation is recorded alongside every captured frame, and changes in orientation of the input device are reflected in captured frames as mainly translation and rotation, using these data an approximate translation of the `fill` frame to match the `base` frame can be computed.

A simple translation matrix is computed, with X and Y components' pixel values approximated using the difference in yaw and pitch angles of both frames and the number of pixels per degree of field of view of the camera. The exact formula being (for the X component, FOV being the field of view in degrees along the X axis):

$$\frac{frame\_size}{FOV} \cdot (fill\_yaw - base\_yaw) \cdot (1 + \frac{|fill\_yaw - base\_yaw|}{FOV})$$

As this is only a simple translation based on angles and percentages, it does not correct for difference in roll (image rotation) or any distortion, like the distortion caused by the camera lens. It is further hindered from reliably overlaying the frames accurately by the inaccuracy and lag in recorded orientation data and the possibility of the capturing device having moved, other than simple orientation change.

Translating the `fill` frame in this manner before further processing has still proven beneficial, as it is not a very computationally expensive operation and in most cases provides a better overlap of both frames (cf. Fig. 4).

(a) Before (b) After

Fig. 4. Overlap of `base` and `fill` before and after orientation-based translation.

Transformation by Features. After the `fill` frame has been translated once, key features from both it and the `base` frame are extracted from their greyscale variants using OpenCV's `FeatureDetector` and `DescriptorExtractor` classes and their implementation of ORB (Oriented FAST and rotated BRIEF) [8]. Although OpenCV does implement other algorithms for feature detection, ORB

proved to be the most efficient while still producing good features. Features of both frames are then correlated using OpenCV's `DescriptorMatcher` class and its "Bruteforce Hamming" algorithm, which is a brute-force search based on the Hamming distance of extracted feature descriptors.

Found matches are then filtered by the distance (in pixels) between the features matched of each match, discarding all matches with distances greater than two times the smallest matched distance (cf. Fig. 5). This serves to exclude disproportionately far matches, which are likely to be wrong. Including only relatively very short distances proved to often produce very accurate overlap in the final image, at the expense of sometimes not producing a match at all.

Filtering this way has purely empirically turned out to work well; other factors or for example basing the calculation on the mean distance also leads to usable results.

A translation matrix is then computed using OpenCV's `estimateRigid Transform` function, the filtered matches and the `fill` frame translated by it.

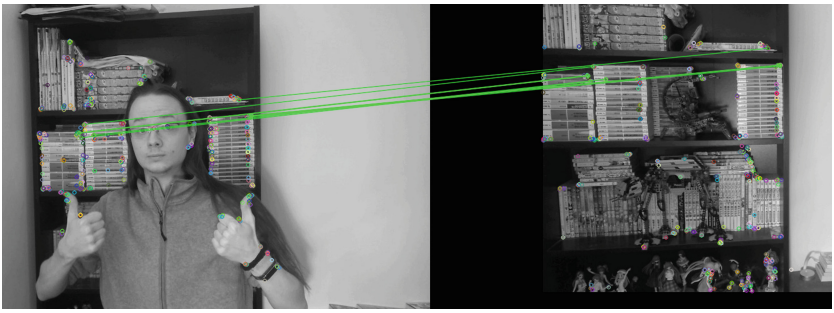


Fig. 5. Extracted ORB features in both frames and selected matches.

Final Blending. To blend the now aligned frames together into one complete frame, the `fill` frame is multiplied by the `base` frame's mask and the `base` frame by the inverse of its own mask, leaving holes in the `base` frame and only the content to fill them in the `fill` frame, which are then added together (cf. Fig. 6, an app screen shot is shown in Fig. 9).

5 Evaluation

Although conceptionally relatively simple, our DR application manages to produce visually coherent and realistic looking results. Through the use of a simple visual memory, structures and objects entirely hidden behind an inpainted area of a frame can be successfully retained with all their details. This amount of detail is impossible to produce using inpainting methods that operate exclusively on the content of one frame (cf. Fig. 7).

(a) `base` frame (b) `fill` frame (c) Final blended image

Fig. 6. Blending of `base` and `fill` frames into the final complete image.

(a) Content aware fill (b) Our DR app

Fig. 7. Comparison of inpainted frames using Adobe Photoshop's PatchMatch-based content-aware fill and output of our DR application.

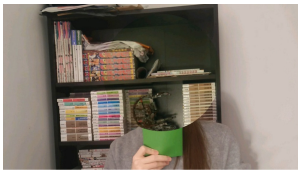


Fig. 8. Half of an envelope in front of the detected face being wrongly inpainted.

5.1 Foreground and Background

Because the only concept of separation between fore- and background is that of inpainted objects being "in front of" the stored frames' content, other things moving in the camera's view cannot really be distinguished or separately reacted to. This leads to moving backgrounds leaving out of date data in the frame store and (partly) view-obstructing things being wrongly inpainted (cf. Fig. 8). Largely static surroundings produce the best results for this reason.

A more sophisticated object detection algorithm may solve at least the foreground problem (e.g. the colour key method does not suffer from this problem), the background problem however is somewhat inherent to the use of a frame store and may not be easily solvable.

5.2 Blending

In addition to non-static surroundings possibly leaving wrong data in the frame store, even mostly static environments can change slightly over time, through lighting changes, moving shadows or the often not software-controllable adjustments made by Android and camera hardware in smartphones to parameters like exposure and white balance.

The naive blending approach of only masking areas in both frames does not correct for these differences and can lead to visible seams, sometimes even very obvious ones (cf. Fig. 10).

Fig. 9. Split view containing (from top left to bottom right): camera input, `fill` frame, (inpainted) output frame, frame store array

Fig. 10. Obvious seam after the exposure auto-adjusted.

Correcting for colour shift and blurring the edges between the two blended frames could drastically reduce these effects. Less uniform changes, like shadows cast by an inpainted object, may not be as easily corrected for.

If objects are detected in a frame, but no usable frame is found in the frame store, the frame is shown to the user unaltered. Instead of doing so, in a kind of best-effort way, another inpainting method that is independent of the frame store could be used to hide the objects. Using such a fallback method may then be less accurate, but would hide objects in these circumstances, which is better than not doing so at all.

5.3 Runtime Performance

During runtime, various timings are taken of different parts of the program to provide an overview over their performance. Running the application on different devices and comparing them reveals a relatively clear bottleneck in need of improvement (cf. Table 1).

Stitching is responsible for the majority of the processing time of one frame, taking around 150 ms even on a relatively recent high-performance mobile processor, limiting the overall frame rate to below 6 frames per second. While not extremely slow and especially in hand-held situations still very usable and good enough to prove the concept, for users to better enjoy using the application, the highest possible frame-rate should be pursued.

Since the application runs all image manipulation sequentially in a single thread, performance could be dramatically improved by parallelising these operations and possibly offloading them to the GPU. Although OpenCV implements

Table 1. Average performance on three different Android devices at a resolution of 1280×720 pixels.

Device/CPU	Stitching	Colour mask	Face detection	Frame search
Galaxy Nexus Arm Cortex A-9 (1.2 GHz)	433.7 ms	9.8 ms	65.4 ms	0.8 ms
Google Nexus 7 (2013) Krait 300 (1.51 GHz)	253.2 ms	4.7 ms	32.7 ms	1.5 ms
Oneplus One Krait 400 (2.5 GHz)	147.5 ms	1.6 ms	22.3 ms	0.7 ms

acceleration using OpenCL for most of the functionality used by our DR application [4], which would enable exactly this kind of parallelisation, OpenCL is not officially supported on Android. Some devices nonetheless support it [3,5] and tests have in the past shown very good performance using it [15], but getting OpenCL support for OpenCV working on Android turned out to be non-trivial and it was therefore not used in our DR application.

Reimplementing key parts of the application using RenderScript would likely yield similar performance improvements, but be equally non-trivial and specific to the Android platform.

6 Conclusion

This project explored not only a way of utilising AR technology that has not been subject of much research so far, but also an approach to it that has not yet been described. The proposed approach to removing objects from the live view of a camera has in practice turned out to work quite well within its inherent limitations and even produce more accurate images than prior approaches. Performance of the implementation is not on-par with them yet, but the potential for optimisation is still great, depending mainly on the support of OpenCL on the chosen platform. On other, less restricted platforms these improvements may even be implemented easily.

Although users are restricted to orientation and otherwise only very limited movement of the device, since users of smartphones and similar devices are often very stationary (i.e. sat at a desk), this may not be a big problem in many cases. It is not unthinkable that an application like this could in the future be used, if AR technology has matured enough to be widely used in work environments, to filter out unwanted visual distractions, much like noise cancelling headphones are used to filter out unwanted distracting sounds.

Overall this was an interesting foray into a niche topic, which maybe in the future, as AR technology improves and new use-cases thereof emerge, could even become a basis for solving real needs.

References

1. Adobe research. https://research.adobe.com/project/content-aware-fill/. Accessed 10 Feb 2018
2. Android - OpenCV library. https://opencv.org/platforms/android/. Accessed 13 Feb 2018
3. Android devices with OpenCL support. https://docs.google.com/spreadsheets/d/1Mpzfl2NmLUVSAjIph77-FOsJeuyD9Xjha89r5iHw1hI/edit. Accessed 14 Feb 2018
4. OpenCL - OpenCV library. https://docs.opencv.org/2.4/modules/ocl/doc/object_detection.html. Accessed 28 Feb 2018
5. OpenCL overview. https://www.khronos.org/opencl/resources. Accessed 14 Feb 2018
6. opencv/data/lbpcascades at master. https://github.com/opencv/opencv/tree/master/data/lbpcascades. Accessed 20 Feb 2018
7. Barnes, C., Shechtman, E., Finkelstein, A., Goldman, D.B.: PatchMatch: a randomized correspondence algorithm for structural image editing. TOG **28**(3), 1 (2009)
8. Bartolini, I., Patella, M.: WINDSURF: the best way to SURF. Multimed. Syst. **24**, 459–476 (2017)
9. Bruno Patrão, S.P., Menezes, P.: How to deal with motion sickness in virtual reality. Sci. Technol. Interact. (2015)
10. Daly, S.: Google translate App. Nurs. Stand. **28**(29), 33 (2014). Accessed 10 Feb 2018
11. Guihot, H.: RenderScript. In: Guihot, H. (ed.) Pro Android Apps Performance Optimization, pp. 231–263. Apress, New York (2012). https://doi.org/10.1007/978-1-4302-4000-6_9. Accessed 13 Feb 2018
12. Herling, J., Broll, W.: Advanced self-contained object removal for realizing real-time diminished reality in unconstrained environments. In: 2010 IEEE International Symposium on Mixed and Augmented Reality, pp. 207–212. IEEE, October 2010
13. Liao, S., Zhu, X., Lei, Z., Zhang, L., Li, S.Z.: Learning multi-scale block local binary patterns for face recognition. In: Lee, S.-W., Li, S.Z. (eds.) ICB 2007. LNCS, vol. 4642, pp. 828–837. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74549-5_87
14. Newson, A., Almansa, A., Fradet, M., Gousseau, Y., Pérez, P.: Video inpainting of complex scenes. SIAM J. Imaging Sci. **7**(4), 1993–2019 (2014)
15. Ross, J.A., Richie, D.A., Park, S.J., Shires, D.R., Pollock, L.L.: A case study of OpenCL on an android mobile GPU. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6. IEEE, September 2014
16. Shaburova, E.: Method for real time video processing for changing proportions of an object in the video (2014)
17. Ulyanov, D., Lebedev, V., Vedaldi, A., Lempitsky, V.S.: Texture networks: feed-forward synthesis of textures and stylized images. CoRR abs/1603.03417 (2016)
18. Viola, P., Jones, M.: Rapid object detection using a boosted cascade of simple features
19. Ware, C., Balakrishnan, R.: Reaching for objects in VR displays: lag and frame rate. ACM Trans. Comput.-Hum. Interact. **1**(4), 331–356 (1994)
20. Yang, C., Lu, X., Lin, Z., Shechtman, E., Wang, O., Li, H.: High-resolution image inpainting using multi-scale neural patch synthesis. In: 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE, July 2017