A Canon Hack Development Kit implementation of Time Domain Continuous Imaging

Katie Long, Henry Dietz, and Clark Demaree;
Department of Electrical and Computer Engineering, University of Kentucky; Lexington, KY

Abstract

Time Domain Continuous Imaging (TDCI) is a new model for photography that allows exposure timing to be freely manipulated after capture. This is done by creating, and operating on, a continuous waveform representation of how the value of each pixel changes over time. However, at this writing, there are no sensors that directly implement TDCI capture. The FourSee multi-camera prototype enables temporally-skewed exposures to be captured using the four component cameras and then later post-processed to create a TDCI representation, but the post-processing is awkward and requires upload of image data to a separate computer. In contrast, this paper reports on a method whereby a single, conventional, Canon PowerShot camera can be used as a stand-alone TDCI platform. The camera programming is enhanced by custom code which is loaded into the camera using the Canon Hack Development Kit (CHDK). Thus, using code that should be portable to most camera models supported by CHDK, an inexpensive Canon PowerShot camera is able to internally capture and manipulate TDCI streams in the new `.tik` *(Temporal Image Kontainer) file format.*

Introduction

In the Time Domain Continuous Imaging (TDCI) model[1], one might still press the shutter button to take a photo, but the camera does a lot more than capture of a single image in response to that button press. Ideally, the camera records a continuous waveform for how the value of each pixel varies over a much longer period of time than the originally indicated shutter speed. For example, requesting a 1/30s exposure might record waveforms for a period of 1-2s. Using a conventional image sensor, these per-pixel waveforms must be synthesized from what is essentially a high-framerate video sequence. The TDCI waveforms are then saved as a `.tik` file[2].

From a `.tik` file, it is conceptually simple to extract the originally-intended exposure as a still image. However, it is just as easy to extract an exposure representing any interval one might select after capture. **The value of a pixel is simply the estimated average value of that pixel's waveform during the selected interval**. This allows freely changing the virtual exposure interval, or even production of a video sequence at any desired framerate. As the virtual exposure interval is changed, images do not get darker or brighter, they report the average value for the pixel over that time interval... and the more samples are averaged, the closer to the pixel's "ideal" value the estimate will be. In fact, even if the pixel was not sampled at all in the virtual exposure interval selected, an average can still be computed by interpolating from temporally neighboring samples (recall that the waveforms are continuous functions). By further incorporating a noise model, it is even possible to obtain better pixel value estimates by averaging over longer temporal intervals during which the value was constant within the noise model – basically decreasing noise by intelligently "stacking"[3] over intervals where individual pixels were apparently changing only due to noise.

Figure 1. *Canon PowerShot Elph 115 IS – a CHDK TDCI platform*

The current work describes how both the TDCI capture and computational extraction of virtual exposures can be implemented entirely stand-alone inside a conventional Canon PowerShot camera, such as that shown in Figure 1, using the Canon Hack Development Kit (CHDK)[4].

CHDK and the TDCI camera platform

The Canon Hack Development Kit (CHDK) is a comprehensive software package that allows users to modify the behavior of their supported Canon camera in ways that would normally be impossible using the standard camera hardware and software. CHDK does not replace the Canon firmware, rather, it temporarily extends it by loading alongside the original firmware, where it intercepts various functions and can call firmware routines to implement any desired camera operations. For example, cameras previously incapable of behaving as an intervalometer need only have CHDK installed, as it includes a script that can cause capture of images at regular intervals by using the camera's internal clock to determine when to fire the shutter and then invoking the firmware routines to focus and fire the shutter.

Most of the CHDK software is written the C programming language and compiled using a GCC cross-compiler to target the ARM cores. This gives two benefits. First, modification of any of the CHDK-internal modules and functions is within grasp of a large number of developers and enthusiasts. Even basic low-level properties of the camera control, user interface, etc., can be al-

Figure 2. *Sample images captured using live-view-based CHDK TDCI as 720x240 UYVYYY, converted to 720x480 RGB*

tered. Second, although the ARM processors in most PowerShots are not particularly fast, carefully-written compiled C code can be efficient enough to perform non-trivial processing in the camera.

CHDK also provides two interpreted scripting languages: BASIC and Lua[5]. These easy to use languages provide a rich interface for writing simple scripts – such as the intervalometer mentioned above – and do not need to be compiled-into CHDK. A BASIC or Lua script can be changed and run immediately; there is even provision for editing scripts inside a camera running CHDK. These scripting languages also provide an easy way to pass parameters into the C code inside CHDK itself. Thus, they provide easily user-customizable interfaces to any functions compiled-into CHDK.

Perhaps most importantly, CHDK is maintained by an active community that continues to improve and support CHDK on every camera that ever has been supported. Currently, over 130 camera models and firmware versions can run the latest CHDK build. This means that CHDK provides an unusually stable and portable environment. Well-written CHDK C code, as well as Lua or BASIC scripts, can be executed on most of those 130 camera models without any changes.

Although any of those many supported models could be an excellent candidate for TDCI capture, the work reported in this paper originally centered on using the very modestly priced Canon PowerShot Elph 115 IS and currently is using the Canon PowerShot Elph 160. In fact, our Elph 115 cameras cost just $80 each new from a respectable vendor. The Elph 160 costs about $100 directly from Canon as a factory refurb.

Sensor

Although there are many PowerShot models, most models available new at any given time use the same sensor. The Elph 160 is typical of the current generation, using a 20MP CCD sensor. The Elph 115 is typical of the previous generation, using a 16MP CCD sensor. It is somewhat surprising to find a CCD rather than CMOS sensor in a modern camera, especially one supporting live view. Canon actually does use CMOS sensors in their ”HS” (high speed) PowerShot models, but the ones using CCD sensors have higher resolution and arguably produce better image quality. In earlier work[6], it was found that these CCD sensors are also remarkably ”ISO-less” – delivering about the same image quality digitally boosting an underexposed base-ISO exposure as using analog gain to implement a higher ISO capture. This is also a highly desirable trait for implementing TDCI because it means that there is no disadvantage to using low-ISO exposures that can record a larger dynamic range without reaching saturation and losing image data.

Of course, the catch with using CCD sensors is that they are not capable of as fast readout as the ”HS” CMOS sensors can deliver. These CCDs are not intended to handle high speed capture, and in video modes, they are not capable of more than 30 FPS (frames per second) at a modest resolution. However, using the live-view feed produces acceptable image quality (as shown in Figure 2) while providing three key benefits:

- Intercepting the live-view feed allows the code to be easily portable between different PowerShot models
- Under the right conditions, the live-view feed can sustain a fairly high framerate – with disturbingly large jitter. Fortu-

nately, it can be fairly precisely known when acquisition of each sample completed.

- The live-view data bypasses the still-capture frame buffer, so that raw image buffer becomes available as working space for buffering TDCI data. The frame buffer is actually most of the main memory of the camera; the Elph 115 and Elph 160 both use 12 bits per pixel in their raw image buffer – so their raw buffers respectively provide 24MB and 30MB of working space.

The primary disadvantage in using the live-view frames is that they do not encode the full raw data. Each group of four pixels is represented by six bytes, which still averages-out to 12 bits/pixel, but the live-view image encodes color in the YUV space that JPEGs use rather than as linear gamma samples with color determined by position within the RGB Bayer color filter array. The Y channel values are 8-bit unsigned values, one per pixel. However, there are single 8-bit signed values for the U and V channels; thus, a block of four pixels can have a different luminance for each pixel, but all four pixels share the same color. To recover the RGB (Red, Green, and Blue) color channel values, CHDK uses the following formulas:

```
clip(n) = ((n<0)?0:((n>255)?255:n))
R = clip(((Y<<12)+(V*5743)+2048)>>12)
G = clip(((Y<<12)+(U*1411)+(V*2925)+2048)>>12)
B = clip(((Y<<12)+(U*7258)+2048)>>12)
```

Decoding is further complicated by the fact that the bytes for each six-pixel block are in the sequence UYVYYY. It would have been preferable to intercept the live-view feed before it was converted from its raw 12-bit form, but that is not an interface that CHDK currently exposes. Even if that interface was exposed, extracting pixel values to operate upon would require awkward shifting and masking to assemble pixel values – the 12-bit raw format packs data into bytes using a different endian than the main processor, presumably because the JPEG compression hardware hardware prefers that byte order.

Processor and memory

Like most recent Canon cameras, both the Elph 115 and Elph 160 use ARM946 main processors. The ARMv5TE architecture supports the basic ARM instruction set and both Thumb and DSP extensions. According to CHDK's built-in benchmarks, the CPU executes about 83 MIPS (million instructions per second), which is a sufficiently low number to make code efficiency a major issue.

The memory system bandwidth is about 59 Mb/s for writes and 21 Mb/s for reads. Those somewhat disappointing numbers can be roughly doubled by using the small cache that the system provides. Using cache, the bandwidth improves to about 109 Mb/s for writes and 59 Mb/s for reads, but with only 8 KB of data cache, cache pollution is a potentially serious problem. The way around this is that caching can be explicitly controlled at the level of individual references: each memory address is double-mapped so that the function `ADR_TO_UNCACHED()` converts a cacheable address into one that will bypass the cache. Bypass is typically used to keep block I/O operations from polluting the cache, but can be more generally used to avoid any type of single-access address reference from accidentally bumping more useful items from the cache.

Unfortunately, aside from the raw still frame buffer, there isn't much main memory available to hold data. On the Elph 160 it's only about 3 MB. With just a small amount of main memory available, it is significant that the Flash memory card can be used as both file storage space and a scratch space. Bandwidth of Flash memories is highly dependent on the particular card being used. For this work, we have been using Toshiba FlashAir cards, which provide bidirectional 802.11 wireless file transfer as well as 16 GB of Class 10 SDHC storage. Measured read and write bandwidth varies between about 6 and 9 Mb/s – slow, but not really as slow compared to main memory speeds as one might have expected. Of course, Flash memory wears out quickly with many writes (around 10K writes/cell), but we have not yet encountered wear errors on any of our FlashAir cards.

Overall, using these PowerShots with CHDK provides a very flexible programmable camera platform, but obtaining good performance for TDCI capture and rendering requires careful performance tuning.

Considerations and limitations

A fundamental problem in TDCI capture is the potentially huge size of the streams produced. A TDCI stream is constructed from the pixel data extracted from a long sequence of frames. The obvious approach would involve buffering the sequence in memory – but there is barely enough memory to buffer a single frame at full resolution! Writing full-resolution images to the SD card as fast as possible yields a framerate of about 1 FPS. In fact, the simple act of filling the still raw frame buffer with the digitized sensor data takes about 1 second – it isn't just one aspect, but many, that limit the framerate. Fortunately, using the live-view frame stream can work much faster, but there is still the issue of where to put all that data in real time.

In a 2016 Electronic Imaging paper[7], we showed that it is possible to compress a TDCI stream very effectively in the temporal dimension – if you can do so guided by an accurate model of the noise in the images. However, the 83 MIPS processor limits how sophisticated compression can be.

Thus, the primary technical issue is how to balance resolution, capture framerate, storage use, and processing overhead to achieve the maximum possible TDCI quality. There is also the more philosophical question of how to best control the process – what should the user interface be and which CHDK mechanisms should be used to implement the control?

CHDK has a C-coded module that detects movement or other changes in the scene by measuring how the values of blocks of pixels change over time. Although it works at a greatly reduced resolution sampling the live-view feed, it is famously fast enough to trigger to catch lightning strikes. It does that by directly using the live-view feed. By modifying the motion detector module, `motion_detector.c`, to perform conversion to a TDCI representation of the frames over a time interval instead of triggering the shutter and shooting an image, a stream of TDCI data can be captured and processed at potentially faster than video rates.

How is this motion detection routine accessed? The module is invoked by a Lua script calling the built-in function `md_detect_motion()`. With such a script loaded and enabled (see Figure 3), simply pressing the shutter button initiates the script and hence causes TDCI capture. Of course, the bulk of the TDCI capture operation occurs inside the compiled

Figure 3. *CHDK display with Lua script for TIK Capture loaded*

`motion_detector.c` module; the Lua interpreter is far too slow to directly process the pixel data in real time.

Some arguments can be made for using the video functionality on the camera and just averaging the values over several frames. Certainly, this would be much easier to implement as it is after capture and would involve little to no modification of the camera software itself. However, the motion detection module is, as mentioned earlier, capable of providing reasonably high frame rates in a form that is much more amenable to customized in-camera processing than the standard video encoding. Perhaps the biggest disadvantage to using a video mode for TDCI capture is that it does not seem to be feasible to implement the computational extraction of virtual exposures from a video using the camera's ARM processor – and CHDK has not yet provided a way to use Canon's built-in video decoding logic.

Data compression

By directly processing the image data as it is captured, the size of the frames can be dramatically reduced, thus lowering the resource requirements in both writing delay and storage capacity. As suggested in the 2016 paper[7], this can be achieved entirely by compressing in the time domain – using a single "pixel value change record" to represent that pixel value over a period of potentially many frames. If the value at a particular location changes, it's recorded. If it remains the same, then there is no need to record anything.

Of course, that begs the question: how do we know the value has changed? The answer is that we will use a precomputed noise model to determine if two temporally-consecutive values for that pixel are the same. They might not be identical, yet still be the same within noise, in which case the compression can still combine their values. For example, a pixel with value 6 and then 8, where both 6 and 8 are judged to differ only by noise, could be recorded without emitting a new value-change record (perhaps updating the 6 value to 7 to reduce noise).

Thus, for example, one might take a several second capture of a scene where only a few items change (e.g., leaves on a tree moving in the wind). With the data compression, instead of several seconds of frames at the full 720x240 resolution of the PowerShot Elph 115 or Elph 160 live-view feed, one full resolution initial frame is captured and new value-change records are created only for the pixels changing value more than the noise model predicts. A value-change record is actually very simple: it records the spatio-temporal distance from the previous change record and the new value. In other words, these `.tik` change records use a variation on run-length encoding.

The CHDK TIK implementation

The `motion_detector.c` module is an excellent starting point for extending CHDK to record TDCI streams. It works on the 720x240 live-view frame sequence to detect when pixel values have changed, and is able to return a Lua array of the relevant pixel values. Initially, we tried implementing TDCI capture using a Lua script to process pixel data provided by the motion detect function – but this proved impractical for many reasons. Most significantly, the motion detection module limits the number of cells that the user can choose. A typical resolution might be just 12 by 12 cells, and the absolute maximum allowed is just 1024 cells. However, it was too slow to be useful for TDCI even at such low resolutions. Examining the C code for the motion detector, it quickly becomes apparent that there are a variety of speed issues, and the resolution limit was set to ensure reasonably fast operation. However, much of the speed problem is not from the resolution, but from inefficiencies involving handling of the many options provided by the motion detector interface. For example, one can select to detect motion based on various colors: not only the Y, U, and V directly provided in the live-view frames, but also R, G, or B computed using the (relatively expensive) formulas given earlier in this article.

Given these issues, it was clearly necessary to write custom C code for TDCI capture. However, because motion detection is conceptually so similar to TDCI, we found that it was appropriate to implement TDCI as an extension of the motion detect module – which already had interfaces to the appropriate internal functions and data structures. In particular, the motion detector module already included a scheduling mechanism that handles repeatedly invoking the motion detection logic (e.g., when a new live-view frame becomes available). Thus, our TDCI capture is implemented inside the motion detector logic, but essentially disables the other logic when TDCI capture is enabled.

The live-view data feed appears to provide new frame data at a rate that varies significantly, causing some jitter. Perhaps the variation is due to lighting conditions, but it also could be due to real-time scheduling constraints being processed by the operating system inside these Canon PowerShots. The unit time for scheduling is 1/1000s, so finer time measurements are unreliable. However, the time since power up, in 1/1000s units, can be read by calling `get_tick_count()`. Thus, it is possible to track when events happened to nearly 1/1000s intervals – the sampling of frames may jitter, but at least the time at which a sample is processed can be known with good accuracy and precision. Normally, TIK accounts for time in nanoseconds, so CHDK TIK times in 1/1000s units are generally output with six additional zeros in `.tik` files.

TDCI stream capture

Despite the complexity of the concept and the process of converting the stream to TIK images, the camera implementation is relatively concise. It must be in order to maximize real-time performance. In essence, the process of creating a file representing a TDCI stream can be expressed in three phases: initial frame capture, pixel updates, and a stream write-out.

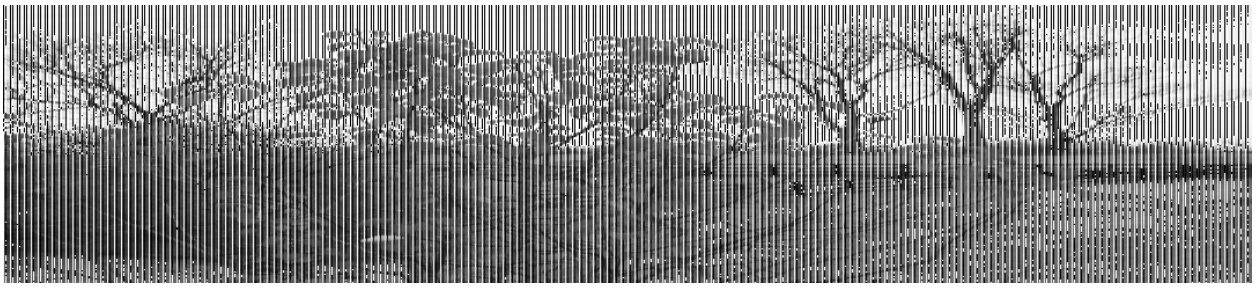

Figure 4. *Rendering of CHDK TDCI TIK file using 720x240 UYVYYY encoding as a 1080x240 monochrome P5 PGM*

Figure 5. *Sample single frame captured using live-view-based CHDK TDCI and virtual exposure combining approx. 62 frames*

Initial frame capture

The first frame's uncompressed UYVYYY data is written immediately after the following TIK header in the buffer:

```
P5
# TIK V 20161130 UYVYYY
# TIK X 180
# TIK Y 240
# TIK F 1000000
# TIK G 2200000
1080 240
255
```

Although the TDCI data in the file is not, this header is fully compatible with the NETPBM[8] PGM (Portable Gray Map) file format and so is the initial frame content. This allows a graphics display or editing program that does not understand TDCI files to still show some representation of the image data contained. The 1080x240 resolution is really describing a 180x240 array of six-byte UYVYYY pixel data, but allows a sensible monochrome rendering. As seen in Figure 4 (which is the P5 rendering of the same TIK TDCI file used to generate the images in Figure 5), the Y channels are essentially a monochrome image, so it is only the vertical lines from the U and V fields that disrupt the image display with a vertical bar pattern. We find these crude monochrome "preview" images most useful for file managers, which otherwise could not render any representation of the TIK file contents.

The `TIK` structured comments describe the TDCI contents of the file. Most importantly, the first structured comment identifies which file format this uses. `F` specifies the default framerate, which is really the change timestamp increment, as being 1/1000s (i.e., 1000000ns). `G` specifies that the file data is encoded with an approximate gamma of 2.2, although how, or even if, gamma correction is performed is unspecified. In-camera, our software currently does not correct the gamma, nor do the CHDK formulas for conversion to RGB.

Pixel change records

Once the initial image is stored in the raw buffer, the whole point of TDCI is to only record pixel value changes that exceed the predicted noise level – thus obtaining significant compression in the time domain. This compression has the happy side-effects of increasing framerate by reducing the delay in writing to memory and increasing the length of the TDCI stream that can be buffered in the raw buffer memory space.

Unfortunately, the live-view frame data's UYVYYY encoding convolves samples for four pixels and is quite awkward to process with a noise model. Although each pixel has its own Y value, the U and V values are shared by each group of four pixels – so changes tend to be highly correlated within a four-pixel block. Processing an error model seems simple enough; just compute the absolute value of the difference between old and new byte values and compare that to an allowable error threshold. Treating the data one byte at a time is slow, but treating the bytes four-at-a-time (in 32-bit words) is complicated by the fact that the U and V fields are signed while Y values are unsigned.

Our initial CHDK TDCI TIK implementation was version 20160629 UYVYYY. After a single 0 byte, it used an variable number of bytes to specify a spatio-temporal "run length" between six-byte UYVYYY tuples in the same X,Y position that had significant value changes. A run could span multiple complete frames, adding 64800 per frame skipped. However, all the byte-level processing made this approach too slow.

Thinking in terms of the "KISS" principle, our second version, 20160712 UYVYYY, made no attempt to compress. In fact, it even included a P5 header (including a timestamp) for each frame copied into the buffer. This required copying more data, but avoided the slow and awkward byte-level operations – both for encoding data and for comparisons against the noise model. It obtained notably higher framerates than the first version, sometimes apparently exceeding 50 FPS.

Our current version is 20161130 UYVYYY. It uses very heavily optimized C code to process a 32-bit word at a time under all circumstances – in fact, the textual header is even optimized to be a multiple of four bytes long so that all 32-bit word accesses are aligned. The compression is still based on spatio-temporal run length encoding, but a run is encoded by a single 32-bit word with 1 in the least-significant bit position and a 31-bit run length in the higher bits. A block of four bytes containing at least some changed pixel data is copied intact except that the least-significant bit is cleared to 0. The loss of one bit of precision for a Y channel value is generally not significant (noise is far more), but this marking means the data size per frame cannot be increased by the compression algorithm. The comparison logic for the noise model is a branchless SWAR (SIMD Within A Register) algorithm that tests all four fields in a 32-bit word at once; the algorithm works for both signed and unsigned fields without needing to distinguish them. Typical compression within a frame reduces the pixel data to between 20% and 30% of its original size; any size gap between frames encodes as a single word.

File saving

Once the specified capture interval has ended, or the raw buffer has been detected as filled, the raw buffer contents are written to a file on the SD card in the camera. The file name is remembered so that it can be reloaded to extract virtual exposures as P6 PPM files – with all processing done in camera.

Extraction of virtual exposures

Before version 20161130 UYVYYY, we did not provide any mechanism for rendering virtual exposures in-camera. Although most of the stand-alone `tik` tools attempt to intelligently interpolate between temporally-adjacent pixel samples, that process seems a little too complex for the modest computational resources available within a CHDK PowerShot. Thus, the new virtual exposure rendering is kept very simple – and it too is invoked from the motion detector interface in Lua. An exposure interval is described as a start time, in 1/1000s units, from the beginning of the TDCI stream and an exposure integration time (shutter speed). Note that the requested exposure interval need not correspond to frame times in any particular way.

The last TDCI TIK file is loaded into the raw buffer, and the initial frame UYVYYY data is copied into a reference buffer. As each 32-bit word is read in the buffer, it is either interpreted as a span (run length) or a block of four data values. The data values are used to update the reference buffer. The span is used to determine the X,Y coordinates of the next datum. However, if the last block in a frame is written or the span passes the end of the reference buffer, then the reference buffer contents are added to the value buffer. The value buffer contains a 32-bit value for each datum, and carefully treats U and V values as signed; it acts to sum all the frame contributions.

Once all the frames in the interval have been processed, the value buffer entries are divided by the number of 1/1000s intervals over which they summed. These average values are then used to produce RGB data for a PPM (Portable Pixel Map) file by the formulas given earlier. The file data is constructed in the raw buffer and then written – examples are shown in Figures 5 and 6.

Figure 5 shows the first frame of a TDCI sequence and then a virtually exposed frame spanning a longer period. Note that the background isn't blurred, but the tree leaves fluttered in the wind, causing them to blur in the second image. Figure 6 is a more representative example of how we expect TDCI to be used. Virtual images were rendered for several time intervals, allowing a user to find/create a frame in which the train is centered. Then the same was done to obtain a pleasing look in a longer (motion blurred) exposure.

The one necessary postprocessing step is rescaling to correct the 1:2 pixel aspect ratio. The 720x240 images are kept at that resolution throughout the in-camera processing, but should be treated as being 720x480 using a system with square pixels. This doubling of the Y dimension should be done using interpolation, not just doubling of each line, but doing this in-camera would result in a longer processing time and a larger PPM file.

Noise Model

Underestimating noise leads to a greater number of pixel value change records being produced, less temporal smoothing, and a poorer signal-to-noise ratio for virtual exposures. Overestimating noise is worse; it tends to miss some temporal events. Although a noise model is used for TDCI encoding, we do not yet have code implemented in-camera to create a customized noise model. The fixed noise model used conservatively underestimates noise at base ISO, so it grossly underestimates the noise in high-ISO TDCI captures like the one used for Figure 6. Despite that, TDCI normally reduces noise, and it still did. Only the 1/1000s @0s virtual exposure in Figure 6 had no data from other frames available, and thus it has the most visible noise.

Conclusion

CCD-based Canon PowerShot cameras are certainly not intended to support high-framerate imaging and extensive reprogramability. Further, the images obtained from their live-view stream are not of the highest quality. There is also the minimal pool of computational resources with which to implement the processing. However, using CHDK, these cameras do make a surprisingly effective testbed for TDCI TIK[2] imaging.

The capture of TDCI TIK files directly in-camera clearly works at an acceptable level, crippled only by modest framerate and resolution. With the latest version, it is also possible to render virtual exposures in camera... but, at this writing, there is not yet a GUI to show you what you are getting. There is also currently no provision for computing a better noise model in-camera, but that also is planned. Our intent is to make all of this freely available from our website, `http://aggregate.org/DIT/TIK/`.

Acknowledgments

This work is supported in part under NSF Award #1422811, *CSR: Small: Computational Support for Time Domain Continuous Imaging.*

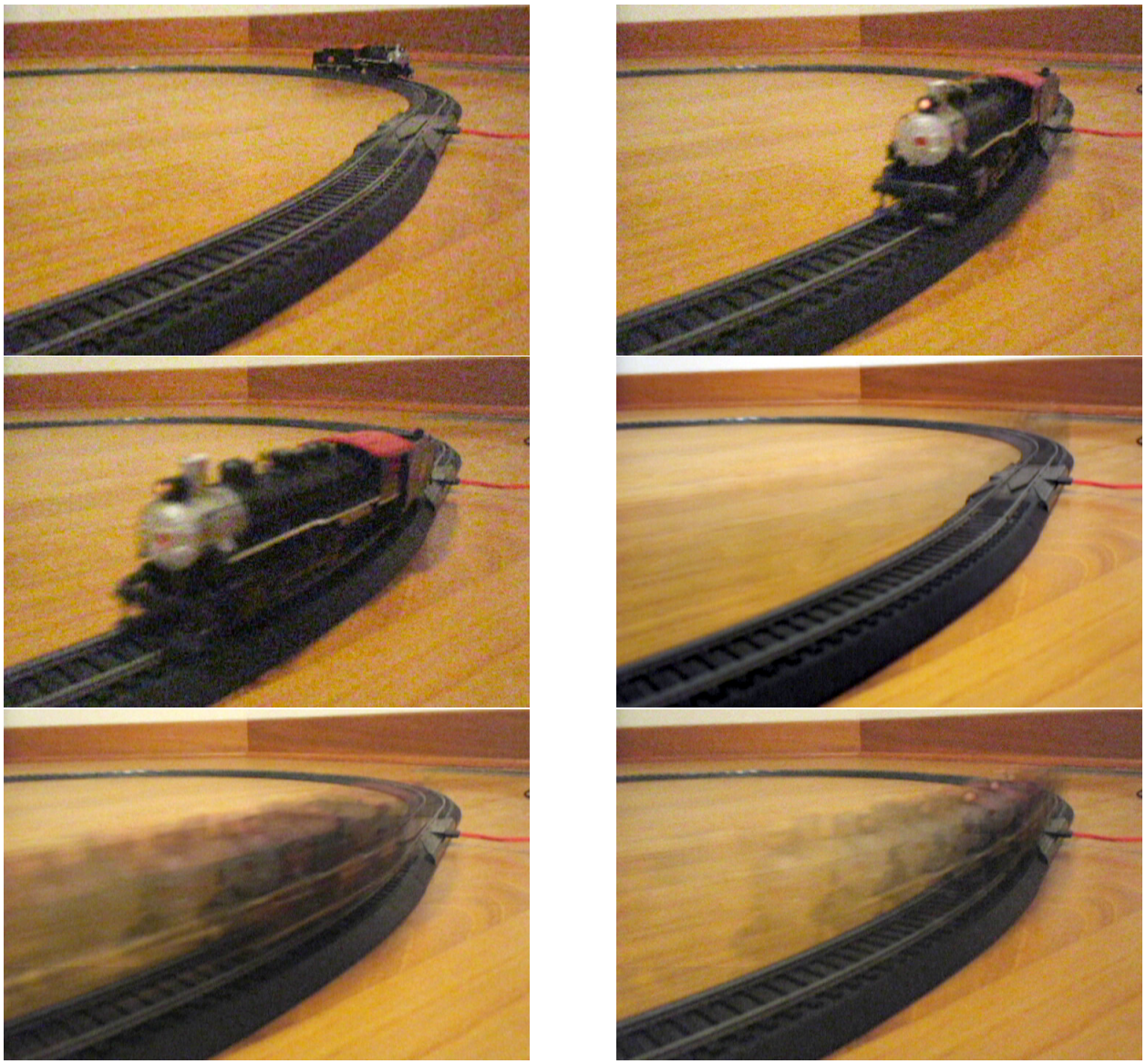

Figure 6. *Virtual exposures of an HO-scale train: 1/1000s @0s, 1/1000s @2.5s, 1/1000s @2.8s, 5s @0s, 1s @2.5s, 1s @ 2s*

References

[1] Henry Gordon Dietz, Frameless, time domain continuous image capture, *Proc. SPIE 9022, Image Sensors and Imaging Systems 2014, 902207 (March 4, 2014); doi:10.1117/12.2040016.* (2014).

[2] Henry Dietz, Paul Eberhart, John Fike, Katie Long, Clark Demaree, and Jong Wu, TIK: a time domain continuous imaging testbed using conventional still images and video, to appear in *Electronic Imaging, Digital Photography and Mobile Imaging.* (2017).

[3] Richard L. White1, David J. Helfand2, Robert H. Becker, Eilat Glikman, and Wim de Vries, Signals from the Noise: Image Stacking for Quasars in the FIRST Survey, *The Astrophysical Journal, Volume 654, Number 1; http://stacks.iop.org/0004-637X/654/i=1/a=99* (2007).

[4] Canon Hack Development Kit (CHDK), *http://chdk.wikia.com/wiki/CHDK* (accessed November 26, 2016).

[5] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho, Reference manual of the programming language Lua, (*Monografias em Cincia da Computaao 3. (1994).*

[6] Henry Gordon Dietz and Paul Selegue Eberhart, ISO-less?, Proc. SPIE 9404, Digital Photography XI, 94040L (February 27, 2015); doi:10.1117/12.2080168. *(2015).*

[7] Henry Gordon Dietz, Zachary Snyder, John Fike, and Pablo Quevedo, Scene appearance change as framerate approaches infinity, Electronic Imaging, Digital Photography and Mobile Imaging XII, pp. 1-7 (February 14, 2016); *(2016).*

[8] Jef Poskanzer, NETPBM: Extended portable bitmap toolkit, (1993).

Author Biography

Katie Long is pursuing a BS in Computer Engineering from the University of Kentucky (graduating 2018). Clark Demaree is also an undergraduate student, and Henry Dietz is the faculty member supervising this research.