Efficient Interactive Text Playback for Code Sharing and Tutorials

by

Eric Yi-Hsun Huang

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

B.SC. COMPUTER SCIENCE HONOURS

in

"THE DEPARTMENT OF FIVE" – John Braun

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Okanagan)

April 28th, 2017

© Eric Yi-Hsun Huang, 2017

Abstract

A recording of typed textual information is often stored as a video or GIF from a screen capture done using traditional video recording tools. These storage formats save frame-by-frame pixel data to describe the changes that occur in the video. This is often used to demonstrate the writing of software programs in tutorials, which is accompanied by narrative audio. However, these text-only videos inevitably have large file sizes due to the high resolution necessary in order to maintain legibility in small fonts. Additionally, these videos are immutable, and cannot be interacted with like a regular text document, making it impossible to copy and paste code. This project introduces *BITR: the Binary Interactive Text Recording*, which is a file specification designed for storing a recording of textual information written over time. BITR provides filesizes over 25,000x smaller than videos, interactivity, and equivalent functionality to video files. The BITR specification was also designed to be capable of being parsed as an online stream, allowing for the possibility of extremely bandwidth efficient text-only livestreams.

Table of Contents

List of Figures

Acknowledgments

My sincere gratitude to Dr. Ramon Lawrence for his continued support and mentorship throughout my undergraduate degree. It was his initiative and generosity in my first year that has shaped and defined the rest of my career.

Chapter 1

Introduction

In software development, learning an existing codebase of a product is one of the primary hurdles that a new developer must overcome. This codebase can be thousands of lines long and comprised of hundreds of files. However, these files were not written in the way they are stored – instead, code grows organically; in a non-linear fashion. Small pieces and modules are written which interact and exchange data, crossing file boundaries and creating relationships or dependencies that are implicitly expressed and invisible to anyone other than the original author. Without external guidance, these files appear static, flat, and linear to the new developer. The question is then posed, "how can we better display this organic relationship information in our code?"

One solution common in the computer science community is to utilize a video recording. This is often used in coding tutorials, where the tutor will type their code and narrate what they are doing. By showing the code incrementally as it is written, the organic information is preserved and becomes easy to follow. Unfortunately, a video has several drawbacks – the filesize of such a video is often extremely large, due to the high resolution needed in order to cleanly render the small point font used in most Integrated Development Environments (IDEs). These files often approach the order of several gigabytes for videos approximately 15 minutes in length, even with modern day advancements in video compression technology. When the video is optimized to reduce filesize, a loss of quality could potentially render the video unreadable and useless. A video is also ill-suited for textual data, as the ability to interact with the text – such as to copy and paste – is lost. If the textual data were directly stored instead, then many of these problems would be solved – textual data is significantly smaller in size compared to pixel data. It can also be rendered in perfect quality and interacted with in a natural, intuitive fashion.

This thesis presents *BITR – The Binary Interactive Text Recording*, a file specification and platform designed to store a replay of code as it is

written. By utilizing the approach of storing textual data, all of the benefits above are realized, while still supporting all of the core behaviour that one would expect from a video file – such as pausing, seeking, and controlling the playback speed. Experimental results show BITR recordings to have filesizes over 25,000x smaller than conventional video recordings.

Some prior works exist that also attempt to exploit the advantages of storing textual data. One work utilizes the DrScheme platform to record and analyze code replays in order to better measure student performance [KBÜ09], and another called JTutor [KSN04] builds upon the Eclipse platform to record code replays that can be used as example programs, which replace the static demo files that are typically provided as reference material in introductory programming courses. Both of these works are heavily coupled to their target platforms (DrScheme, Eclipse) and focus on the educational niche. Since BITR is platform and language agnostic, it provides an open interface that allows it to support any text editor or IDE that can write BITR encoded files. This fills the gap left by the platform dependent prior works, and allows BITR files to behave more akin to a video file.

Included in this work are three primary components – the BITR file specification, an example IntelliJ plugin to write BITR files, and an example reader program that plays back a BITR replay. BITR is not coupled to the example reader and writer programs presented, and could be easily extended to support playback on any platform, such as a web platform. BITR has the scaffolding in place to support a streaming architecture, but was left as possibility for future work.

The *Background* chapter discusses the technologies used in the implementation of BITR and the BITR platform. In the following chapter, the BITR file specification is described, as well as the algorithms and strategies used in the implementation of the core features. *Experimental Results* discusses in detail the filesize measurement made to compare BITR to a standard video recording. The final chapter concludes and draws attention to the applications of BITR and ideas for future work.

Chapter 2

Background

In *Design Decisions*, several of the core architecture choices in BITR are reviewed. The technologies that were chosen are described in more detail, in their respective chapters.

2.1 Design Decisions

The core design philosophy of BITR stems from the following core criteria.

Compactness and Speed of Usage

In order for BITR to be more useful compared to a video file, whatever strategy is used to record data must have minimal space usage as its primary priority. This storage format must also take into account the performance of loading and parsing the file, as any hiccups or delays in reading the BITR file would be unacceptable and jarring in playback.

Core Video Functionality Must be Supported

All of the core functionality one would expect from a video must be supported:

- Playback
- Pausing
- Seeking – the ability to jump around in the replay.
- Fast-forward – controlling the playback speed, as most people will not want to watch the code be written in real-time.

Remain open to streaming

Since BITR would have value as a streaming protocol as well, keeping the storage format open to streaming would be desirable. Specifically, the file cannot require an intermediate loading step where the entire file must be pre-processed, reorganized, or translated before playback. Additionally, the storage design must be robust enough that a stream that is interrupted or initialized mid-way through can be rebuilt in order to continue playback.

Allow future extension

The data encoding used must be remain open to extension or future change, to allow the embedding of any desired metadata into the replay. This can then be used to include annotations into the recording, which can be displayed by the BITR reader. It is important that any changes to the data encoding remain backwards compatible with older BITR files, as to not render any previous file useless after a change to the encoding is made. Many video files have a lifespan of several years, and BITR files must compete with this standard.

2.1.1 BITR Architecture

Several approaches to the implementation of BITR were considered. Borrowing from the idea of videos, one of the most natural storage methods for textual data would be to record the full text at each time interval, like how a video will record matrices of pixel data every time step. The advantage of such an approach is robustness – this storage format is very durable, since each time step is effectively independent and can stand alone. This would satisfy the streaming requirement, as the full text state can be found in a received step at any given point. However, such a storage method would be extremely costly in terms of space, and would contain a significant amount of redundant data. A compromise here would be to only snapshot the text state whenever a change is made to the text – this approach would save on space, but still contain redundant data that is repeated in each step.

Instead, the implemented strategy used in BITR is to only track what changes in the text – effectively, a "text delta". This completely eliminates any redundant information, and ensures that the BITR file is of minimal size. Then, the problem of streaming can instead be pushed to the platform – when a new stream connection is received, a full text state can be sent, like a keyframe. From there, the playback can be continued by applying changes

as they are received. As another benefit, the filesize of a BITR file is now bounded only by the total number of changes made through the recording, instead of by the length of the recording. This is especially well suited to the playback of programming; as there will often be large pauses, with small bursts of edits interspersed throughout.

Two additional extensions were considered for implementation. First was to group together rapid sequences of edits as a single change. This is known in BITR as *chunking*. The advantage here would be to save in overhead – as an example, instead of writing five rapid inputs of the letter 'a' as individual changes, it would be possible to chunk the changes together as if it were a single edit. Since each edit has a timestamp associated with it, in many cases the timestamp may be the most space expensive piece of data that is recorded. In a hypothetical case, this timestamp may be a 4 byte integer (it is not necessary to store an 8 byte long, as the timestamp may be relative to the start, instead of absolute like an epoch timestamp), which is much more expensive than the 1 byte character. However, the disadvantage of chunking would be a loss in granularity. If the recorded edits are played back as-is, the flow would be choppy and jarring, as chunks of text would materialize into existence. A possible solution here would be to pre-process the edit back into single character pieces, but then the timing data for each individual character would have to be either extrapolated, or be written alongside the edit in some fashion. Given these drawbacks, and the fact that the current implementation of BITR still eclipses video files by a significant margin, it was ultimately decided that chunking was not worth the additional implementation cost.

The second possible extension considered was *branching*. It was thought that it may be valuable to be able to visualize all of the different versions and branches the text went through. Effectively, every time a user deletes and types new text, they have diverged off of the previous path into a new branch. After further consideration, it was decided that branching was non-trivial to accomplish, and would be difficult to support in a streaming environment.

2.1.2 BITR Storage Format

The primary design choice here was to decide whether to store BITR data in a text based format, or a binary format. A text based format using some form of markup, like JSON or XML, would be human readable and

editable. However, there would be a large overhead in the cost to store and parse the data, due to the added syntax and structure of the markup. Additionally, such a format would require serialization before it could be put "on-the-wire" and used in a streaming context.

Instead, the decision was made to use a binary storage format. By trading human readability, we gain a significant space improvement. Most binary formats are often faster to parse as well, since it obviates the need for costly string processing. *Protocol Buffers* were ultimately decided upon as the storage technology powering BITR, which is described in further detail below.

2.1.3 Target Platform

Even though BITR is platform agnostic, it would not be useful unless replays could be recorded and played back. To this end, reference implementations for reader and writer programs were implemented.

The writer is a plugin for the IntelliJ IDE platform, and allows the recording of BITR files directly through the IDE. The plugin was tested on the Java flavour of the editor (IntelliJ IDEA), but the flexible nature of the platform means that the BITR writer can work with any editor from the same family, such as CLion, PyCharm, or RubyMine. The BITR writer allows the user to start and stop the recording procedure, and target filename can be specified at the time of recording.

The reader is written in Java, using the *JavaFX* library to drive the GUI. The UI design of the reader was made to resemble that of video players. All of the core functionality expected from a video player is supported. See Figure 2.1 for a screenshot of the BITR reader.

2.2 Google Protocol Buffers

Protocol Buffers (*Protobuf*) is a binary storage format for records designed by Google intended for fast parsing and ease of transfer over a network. It was originally created to unify the communication protocol used by the internal services that power Google's platform, but is now widely used in other applications due to its speed and robustness. Google claims that Protobuf parses "20 to 100 times faster than XML", and is "3 to 10 times smaller than XML" to store. Protobuf provides backwards compatibility

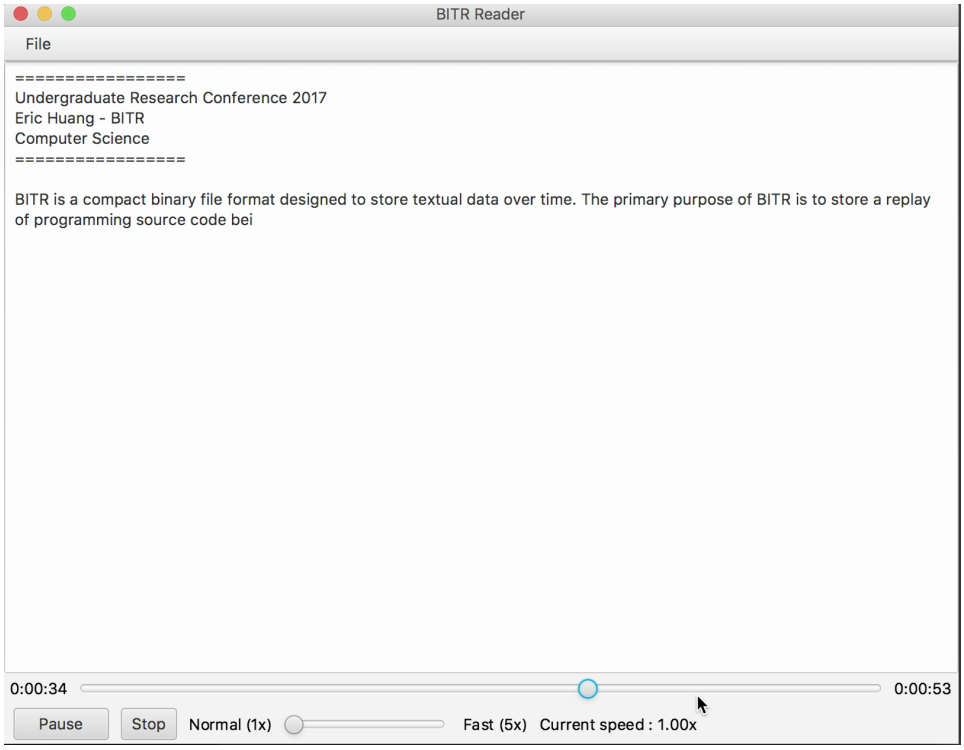

Figure 2.1: BITR reader displaying a recording.

as a feature, where files written with an older structure remain readable to programs that expect a different layout of data. Additionally, Protobuf requires no serialization to send over a network, as it is already encoded in a binary format.

In light of these benefits, Protobuf was chosen as the foundation for BITR. Partial inspiration for this idea stems from initial research in how other applications store replays. The popular video game *Dota 2* records replays of every match that is played, and also uses Protobuf as its storage format. Replays for video games are a similar design problem that is faced in the implementation of BITR – a video game must track actions (mouse clicks, ability usages) that players make at a given time, just as BITR must track the keystrokes the user types.

Chapter 3

BITR: Binary Interactive Text Recording

3.1 BITR Storage Explained

In BITR, a *frame* refers to a record containing a single edit action. A *timeline* is a contiguous sequence of frames, comprising one whole BITR file.

Frame
Timestamp
Offset
Erase Flag
Payload

Figure 3.1: A BITR frame.

Timeline (BITR File)
Frame_0
Frame_1
...
Frame_n

Figure 3.2: A BITR file.

Below is the protobuf definition of a frame used in BITR:

```
message Frame {
    int32 tickstamp = 1;
    string payload = 2;
    bool is_erase = 3;
    int32 offset = 4;
}
```

The above syntax reads almost like that of a C `struct`. The only notable portion are the equivalences – this is protobuf way of specifying the *tag* of a field. A tag is used to quickly identify a field once it is encoded in binary as part of the reading and writing process of protobuf. This specification is then saved in a `.proto` file and compiled using the `protoc` compiler, which translates the protobuf syntax into ready-to-use code in the specified target language. In the case of BITR, the target language is Java. A class is then produced, containing all of the code necessary to interact with a frame in an object oriented fashion that is natural for Java. For example, the following is the snippet of code used to construct a frame:

```
FrameOuterClass.Frame frame = FrameOuterClass.Frame.newBuilder()
    .setTickstamp((int) timeDelta)
    .setPayload(is_erase ? oldFrag : newFrag)
    .setIsErase(is_erase)
    .setOffset(offset)
    .build();
```

BITR then utilizes a `CodedOutputStream` provided as part of the protobuf library in order to write the constructed frame to a file. As edit actions are recorded by the BITR reader, frames are continuously appended to construct the timeline. If a streaming context were desired, the `CodedOutputStream` would instead be pointed at a socket, instead of a file. Since protobuf writes all of the necessary length information alongside the message, no padding or separator is required in the BITR file. The file extension `.bitr` is used for all BITR recordings.

In a BITR frame, a *timestamp* is an integer that describes when the edit action occurred, in milliseconds. A timestamp of 0 represents the beginning of the recording. The *payload* contains the actual text that was added or deleted. In many cases, this is a single character – in the case of a copy and paste or other block operations, all of the characters that were added/deleted are included as a single action. The boolean flag *is_erase* signals whether or not the frame represents a deletion. An important design decision was made to have the *payload* contain the snippet of text that was removed in a delete action – this is crucial for the implementation of reverse seek. The *offset* is an integer describing the index in which an action occurred. An insertion is placed starting at the specified index, and a deletion removes all characters in the range starting at the offset (inclusive), and ends at `offset + len(payload)` (exclusive).

3.2 Example Timeline

If the user were to type the word “cat”, the following frames would be generated:

Frame 1	
Timestamp	3
Offset	0
Erase Flag	FALSE
Payload	“c”

Figure 3.3: A BITR frame for the letter “c”.

Frame 2	
Timestamp	5
Offset	1
Erase Flag	FALSE
Payload	“a”

Figure 3.4: A BITR frame for the letter “a”.

Frame 3	
Timestamp	7
Offset	2
Erase Flag	FALSE
Payload	“t”

Figure 3.5: A BITR frame for the letter “t”.

When these frames are applied, the original word is then reconstructed.

3.3 BITR Writer

The BITR writer is an IntelliJ plugin written in Java. The writer utilizes an IntelliJ `DocumentListener`, which provides an event hook that is fired whenever the document being watched is edited. The event then provides a

`DocumentEvent` object, which contains context on the change. Most important for BITR are the *newFragment*, *oldFragment*, and *offset* fields provided by the object.

The *newFragment* contains any additional text that was added to the document, *oldFragment* contains any text that was deleted, and *offset* describes where the change had happened. The event object also provides a field called *oldTimeStamp*, but no documentation could be found on the field and its purpose could not be determined based on the values returned. BITR frames are written directly from the data received from the events, and the BITR file is finalized once the user stops the recording.

3.3.1 BITR Reader

The BITR reader is a Java program written using the JavaFX library. A screenshot of the program can be found in Figure 2.1. When a BITR file is loaded into the reader, each frame read from the file is placed into a Java *TreeMap*. The TreeMap acts as a key-value store associating the timestamp of the frame to the rest of the frame data. Internally, this in-memory TreeMap of frames is referred to as the BITR *timeline*. A *StringBuffer* is used to represent the current text state, which is modified by the application of frames. A StringBuffer is used due to the immutable nature of strings in Java, and allows fast in-place modification of the text.

Why TreeMap?

The fundamental problem the BITR reader must solve is to keep a timer that continually counts upwards (referred to as a *playHead*), and to apply frames as the timer exceeds the timestamp of any frame. This requirement implies that retrieval of frames based on the timestamp must be fast, which suggests that a map based data structure would be appropriate. The monotonically increasing nature of the play head implies that storing the frames in sorted order would be optimal, since we know that is the ultimate order that frames will be accessed. However, the popular *HashMap* in Java would only be appropriate if we knew the exact key to retrieve, which we often do not – consider the case where the next relevant frame that we would apply has a timestamp of 307 milliseconds. Since our timer cannot guarantee that it always increments in a consistent manner (due to other processes on the system that compete for resources and CPU time), we could potentially go

from a play head of 200 to 353 milliseconds, and thus effectively "skipping" the next frame that should be applied. Given these requirements, we essentially need a "fuzzy" approach to retrieving the next frame that is flexible enough to handle this inaccuracy.

The Java implementation of the *TreeMap* satisfies all of the requirements of BITR, and its capabilities are relied upon for the implementation of many core BITR algorithms. The TreeMap is backed using a Red-Black Tree, and provides the usual binary tree time and space guarantees, such as *O(log n)* insert and retrieval in the worst case. It provides the core interface expected from a *Map* such as *put(key, value)* and *get(key, value)*, and also provides additional functionality that leverage its tree structure, such as *floorEntry(key)*, which returns the closest key-value pair that is less than or equal to the given key. Given the fuzzy key matching described earlier, it is evident why the TreeMap is the preferred back end data structure for BITR.

3.3.2 Playback, Pausing and Stopping

The BITR reader contains a separate thread that handles the playback of a BITR file. By keeping the playback in its own thread, the play head can be accurately incremented in real time without worry that any blocking operation would throw off the timing. Pausing in BITR is accomplished by using a toggle-able flag that prevents the play head from being incremented, effectively halting playback. Stopping in BITR is implemented by resetting the play head to 0 and wiping the text state.

In each cycle of the playback thread, the play head is incremented and then a range slice operation (called a *subMap*) is performed on the timeline starting from the previous play head to the current play head. This returns all frames that occurred in the given time interval – which is important behaviour needed in order to catch the possibility that multiple frames need to be executed simultaneously. One can see this situation arising if a large delay impacts the play back – if it were 3,000 milliseconds since the last cycle, then it becomes extremely likely that several frames lie in the spanned time interval.

3.3.3 Time Display

In the BITR reader, the current elapsed time and the total duration of the recording is displayed in the Reader GUI. The current time is directly derived from the current play head value by converting from milliseconds to hours, minutes, and seconds. The total duration of the recording is found by saving the timestamp of the last frame in the file when all of the frames are read off of disk. This timestamp represents the time at which the last edit is made, and thus is the total duration of the file. This timestamp is also converted from milliseconds to HH:MM:SS format for display.

3.3.4 Time Control (Fast-forward)

In the playback thread, the play head increment is determined by taking the difference between the current system millisecond time and the system time recorded in the last cycle. By multiplying this time delta by a desired speed factor, we can effectively simulate a faster or slower time passage. As an example, if the current system time is 300ms and the previous system time was 200ms, then a 2x speedup can be approximated by taking $time\_delta \cdot speed\_factor = 100ms \cdot 2.0 = 200ms$. A slider is provided in the reader GUI that allows the user to control how fast they desire the playback to be.

3.3.5 Seeking

Seeking is the most complicated algorithm to implement for BITR. Unlike a video, where seeking is as simple as reading the data stored at the new play head and displaying it, BITR must determine what the text state looks like at the new timestamp. Since all of the data in BITR is relative, this state must be derived by applying transformations from a known text state to arrive at the desired state.

Conceptually, seeking is effectively just rapid application of frames. The simplest seek model would be to wipe the text state and apply all of the frames spanned between the start of the BITR file and the new play head location. However, we can see that this operation has a time cost that is proportional to the new play head, and that a seek close to the end of the file could result in a lengthy delay. Instead, the implemented algorithm in the BITR reader utilizes the current state the text is in, and only applies the frames that span between the current play head and the new play head. In this model, the cost of a seek is proportional only to the size of the interval

spanned, which is significantly smaller in many practical cases.

However, a relative seek as described suffers from a "hair in the soup" when a seek into the past is performed. In order to "rewind" the current text state, any frames encountered must be applied in the opposite manner of how they are written – an insert becomes a delete, and a delete becomes an insert. This is why it was crucial to preserve the text data in a delete frame, since then any frame can be easily reversed by inverting its erase flag. Our seek algorithm then becomes the following: Find all of the frames spanned in the time interval defined by the new and old play head positions. Apply these frames normally if it's a forward seek, or invert them and apply in backwards order if it's a reverse seek.

In the BITR reader, this is accomplished with help from the TreeMap. Using the *subMap* method on the range defined starting from the smaller to the larger playhead (out of *oldPlayHead* and *newPlayHead*, as the new play head would be smaller than the old in the case of a reverse seek), we can obtain every frame in the expressed interval. If it's a forward seek, then every frame found is simply applied in rapid sequence. If it's a reverse seek, then the call to *subMap* would give us the frames in forward order – so the list of frames is reversed, and then each frame is inverted before it is applied. As a caveat, a reverse seek is more expensive than a forward seek, since it necessitates the reversal of the frame order before application.

Practically, the BITR reader shows no noticeable delay, even when seeking across large intervals in a heavily edited file. This was a surprising result, as I was initially concerned about the performance of the seek operation due to its non-absolute nature.

3.3.6 Applications

BITR has a wealth of potential applications, in particular anywhere where a traditional video is currently being used. Live streaming and code base learning are applications that has been mentioned many times, but an interesting educational application would be in plagiarism detection for coding assignments. Currently, assignments given in programming courses suffer from a common problem of students that copy and paste other solutions, and then changing superficial aesthetic details to avoid detection. If students were required to submit a BITR file of their assignment alongside their completed code, this information would be significantly harder to fake.

The provided recording could also serve as a helpful teaching tool, since being able to watch the process of a student write a project helps an instructor better pinpoint the weak points and errors that a student encounters.

Chapter 4

Experimental Results

As BITR's primary contribution is in its implementation, there is not a significant wealth of quantifiable data that could be gathered and examined. The possibility of running a quantitative trial to determine the effectiveness of BITR as a learning tool was considered early on in the project, but was ultimately decided against due to the necessary ethics clearance required in order to run such an experiment. The original plan for the trial would be to first draft a short codebase, such as the implementation of the Skiplist data structure – then, students who are in first year computer science would be chosen and divided into two groups. One group is given flat code files, and the other group is given BITR recordings. The students would then be given the same amount of time to familiarize themselves with the codebase, and then be asked to answer a series of comprehension questions, as well as to perform a sequence of simple code changes – such as fixing bugs that were intentionally included, or adding additional functionality. The goal here would be to attempt to quantitatively assess if there is any significant performance difference between the two groups, with the hypothesis being that the group presented with BITR files would perform stronger at the assigned tasks.

The primary experimental result for BITR conducted was to determine how much more compact a BITR recording is compared against a video file of the same data. Since BITR primarily targets the ecosystem of coding tutorials, a screen recording was done of the *Sublime Text 3* text editor, using a full screen recording compressed with the H.264 (MP4) video codec. No audio information was written to the recording to provide a fair comparison. The environment and selected video codec were chosen to be in line with what a typical YouTube coding tutorial video would use. While it is possible to further optimize the filesize of the video using post-processing techniques, these strategies must be applied in a case-by-case basis that is uniquely tailored to the specific video at hand – something that is infeasible for the regular programmer to perform on every video they upload. In the video, the following text snippet was typed (partially abridged for clarity in

representation):

Hello

This is a video of a text file being written. This should be quite a large file. It is full of lots of text. Weeeeeeeee

Such text many length

wow

so word

The resulting video was 33 seconds in length, and has a pixel resolution of 1920x1008. The resulting filesize was 49,707,974 bytes (47.4MB).

The equivalent piece of text recorded using BITR also resulted in a 33 second recording, and has perfect textual resolution. The resulting filesize was 1,844 bytes (≈2KB), resulting in a filesize reduction factor of 26,956.60x – a difference of over 4 orders of magnitude. This clearly demonstrates the value of BITR over conventional video.

As further analysis, from the design of BITR we can see that the filesize is determined by the number of edits made in the recording. We could instead ask the question, what would a BITR recording that was 47MB in size look like? This analysis follows below:

1. The piece of sample text contains roughly 162 characters. Since no chunking is performed, this translates directly to 162 edit actions.
2. From the filesize of the BITR recording, we can determine the approximate byte-cost-per-action for BITR: $\frac{1,844\ bytes}{162\ edits} \approx 12$ bytes per action.
3. Thus, 47 megabytes worth of BITR would contain $\frac{49,707,974\ bytes}{12\ bytes/action} \approx 4,142,331$ edit actions.
4. The video recording had 162 edit actions over the course of 33 seconds. This results in $\frac{33\ sec}{162\ actions} \approx 0.2$ seconds per action made.
5. Therefore, 4,142,331 edit actions at the same pace would result in a BITR recording that is $4,142,331\ actions \cdot 0.2\ secs/action \approx 828,466$ seconds in length, which is 9 days, 14 hours, 7 minutes, and 46 seconds worth of video – or 29 full working days.

Chapter 5

Conclusion and Future Work

The BITR implementation presented in this work serves as a strong foundation that has clear, significant benefits over traditional video recordings. The application of BITR in the education field could be significant in revolutionizing the way that programming is presented, both in lectures and assignments.

There is much unexplored ground for BITR and many exciting possibilities ahead. Most significantly, the ability for the BITR writer to support multiple files being written would be a significant boon to any non-trivial software project. Utilizing the open-ended design of the BITR file specification, it would be interesting to explore recording more data into the BITR file, such as the current edit cursor position or mouse cursor position. Annotation support could also be realized. Additionally, the aforementioned trial could be run to gather better quantitative data on the benefit of BITR, and the streaming infrastructure could be finalized and implemented.

Bibliography

[KBÜ09] M Fatih Köksal, RE Başar, and S Üsküdarlı. Screen-replay: A session recording and analysis tool for drscheme. In *Proceedings of the Scheme and Functional Programming Workshop, Technical Report, California Polytechnic State University, CPSLO-CSC-09*, volume 3, pages 103–110. Citeseer, 2009. → pages 2

[KSN04] Chris Kojouharov, Aleksey Solodovnik, and Gleb Naumovich. Jtutor: An eclipse plug-in suite for creation and replay of code-based tutorials. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, eclipse '04, pages 27–31, New York, NY, USA, 2004. ACM. → pages 2