Jester Sight

Mashal Al-Nimer Miles Shamo Ansel Rosso Klakovich Quentin Karpman

University of Illinois at Chicago

Abstract

We propose and describe an implementation of a method of detecting Spelunky 2's internal timer for the True Crown in-game item via analysis of UI elements to extract game state information. This gamestate information is updated in real-time to remain in line with the game to provide useful information to the player. The overarching concept of extracting hidden or obscured information in a UI to present to the user separately is likely generalizable to other contexts.

1. Introduction

Spelunky 2 is a video game that was released in September of 2020. Its predecessor, Spelunky, was notorious for being extremely difficult to win; the sequel is significantly harder depending on the definition of "win" as both games feature multiple endings of varying challenge for a player to attempt to reach. Both are members of the roguelike genre: the player has one life to complete the game and losing entails restarting from the beginning with a new set of randomly generated levels. Because of both the game's difficulty and its roguelike nature, the game has inspired a community of players whom try to push themselves with difficult runs or playthroughs of the game. For example, they may attempt to beat the game as fast as possible, end the game with the highest score possible, or try to complete the game without killing enemies.

1.1. The True Crown

Some of these runs of the game involve an item called the True Crown (Figure 1) which can be acquired by a player through a rather convoluted series of steps. If the player successfully completes the requirements, an NPC (Non-Player Character) named Beg will give the player the item [5]. After obtaining the item, an internal timer runs in the background. Every twenty-two seconds after picking up the True Crown, the player receives twenty-two bombs—an extremely useful item in the game. However, at the same time, the player teleports some distance in the direction they are facing. Teleporting is not necessarily bad—it is often desirable to traverse the game's levels quickly—but if the player teleports into a wall, they die instantaneously. Because the twenty-two second timer is not displayed, it is hard to time the interval and easy to die while unexpectedly teleporting.

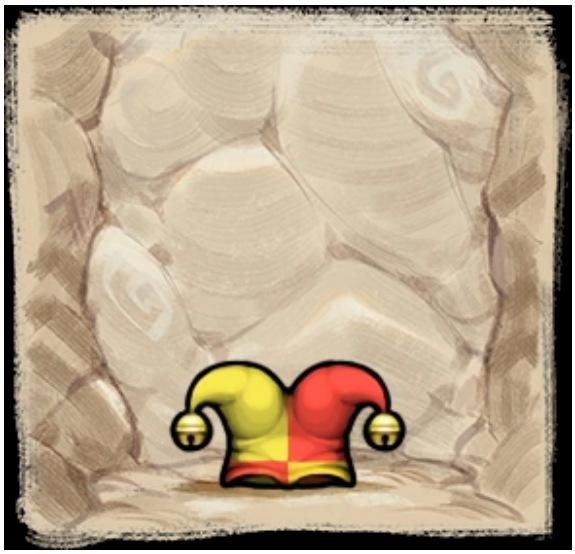

Figure 1. The True Crown item as depicted in Spelunky 2's item journal [5]

However, the True Crown can be extremely useful for high level play. For example, the current world record score run [8] (to attempt as high of a score as possible) uses the infinite bombs the item provides to uncover every piece of gold in each level. A timer was developed to help players use the item while avoiding accidental teleportation related deaths. This timer hooks into Spelunky 2's process and checks the game's memory to determine if the player has picked up the True Crown and, after doing so, exactly when their

next teleport will be [6]. The timer can then play warning sounds for the player at intervals before the teleport happens, allowing the player to prepare. Unfortunately, for any players attempting to submit their accomplishments to the wider community, any tools that modify or access the internal memory state of the game are banned [1]. Therefore, the timer cannot be used in runs submitted to the community, relegating its usage for practicing only. To circumvent this technicality, we have implemented our own True Crown timer, dubbed JesterSight, which also tracks the cycle of the True Crown item but does so solely through visual analysis of Spelunky 2's UI instead of reading the game's memory.

2. Implementation

In short, our implementation consists of three parts:

- Detecting when the player has the crown
- Watching the timer to determine when twenty-two seconds is going to elapse
- Alerting the player to the impending teleport

When a player has the True Crown an icon appears over the player's health in the top right of the screen. This icon is the True Crown's sprite and it's unique to the item, so we only need to detect its presence to detect if the player is holding the item. Once we detect that the player has the crown, we could start a twenty-two second timer to tell when the next teleport will happen. However, as it is twenty-two seconds bound to the in-game time, we cannot just set a timer and wait because the player might pause the game, which also pauses the in-game time. Because of this, we must read the in-game timer that is displayed in the top right corner of the screen. Tracking the in-game time value allows us to get an accurate reading of how long the time in the game has passed which is the important metric for our application. When we begin approaching the twenty-two seconds mark, we alert the player with an audible sound to warn them that they are about to teleport. Then, after the full twenty-two seconds have passed, we restart the counting of the timer and repeat. Throughout this process, we need to account for the player moving between levels; when the player changes level the system needs to recognize that and reset the timer accordingly.

2.1. Challenges

In Spelunky 2, the user is able to change the UI's scale and opacity within its settings. This requires our implementation to be adaptable to the user's settings, assuming they do not make it impossible to extract information, such as setting the UI's opacity to "invisible." We must also consider paused states and level transitions since it hides the UI. Additionally, some levels have a backdrop with similar colors and textures to the crown so the implementation needs to take that into consideration. Furthermore, Spelunky 2 has other crowns that the user can collect and these crowns are displayed similar to the True Crown as exemplified in figure 2. This requires an implementation that is able to distinguish between the crowns to ensure that the detector does not incorrectly flag a crown similar to the True Crown as the real True Crown. Furthermore, the implementation would need to be able to analyze or retrieve information from the in-game timer to prevent the system's teleport warning timer from falling out of sync with the game.

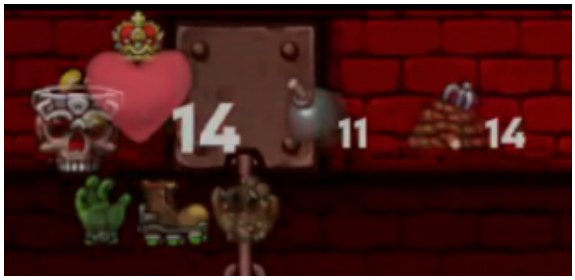

Figure 2. A similar crown (simply called "Crown") and other held items displayed around the player's health.

2.2. Detecting UI Elements

Initially, our approach involved detecting the crown and detecting if the UI is present. Detecting the crown would allow us to know if the user has the item or not, and detecting the presence of the UI allows us to know if the player is in the pause state or not. Thus, we both know when the user has the item and can remain informed of the overall game state.

2.2.1 Detecting UI Presence

To detect the presence or absence of the UI we initially chose to use the Hough Transform [7]. This would detect one of the icons that was ever-present in the in-game UI, such as the heart or bomb as seen in 4; these icons all disappear when the game is paused. If they are present that means the game is not in a paused state while if they are gone then the game is in a paused state. However, as we progressed, we decided the use of the Hough Transform was not necessary. We came to this conclusion after we recognized that the information we already have to collect collect about the crown,

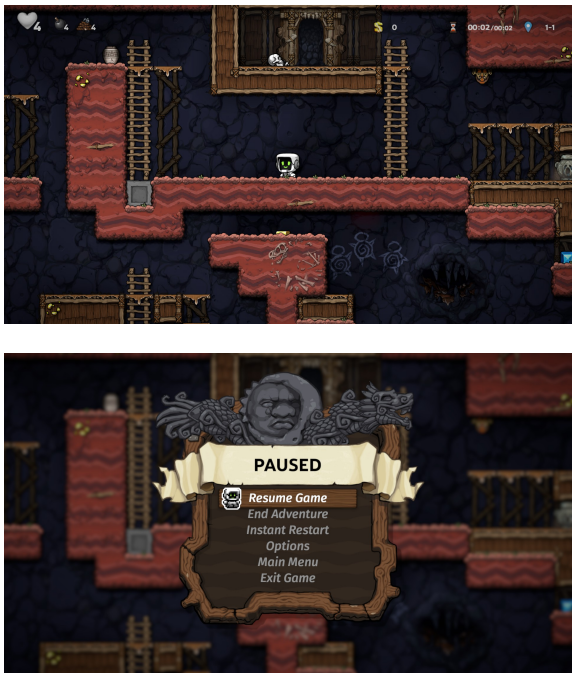

Figure 3. Spelunky 2 both unpaused (top) and paused (bottom).

timer and level number could all be used to also determine if the game was paused, as they all are not present, as exemplified in figure 3. As we are already required to detect the crown, timer and level number, we can deduce the information directly; if we can detect any of the three then we know the game is being played.

2.2.2 Detecting the True Crown

In order to detect the presence of the True Crown icon, we compute the distance transform of a small region around the crown and then template match within the region. This allows us to get the position of the crown if it is in the image based off a predetermined template, shown in figure 4. While template matching can be implemented with a form of gradient descent, the image region we actually need to search is extremely small as we always know where the crown is within the larger image. Computing patches of our template that are large enough to get an accurate gradient estimate would be close to computing every possible patch. As such, to template match, we simply compute the value of the template at every possible window in the image and return the position if it is below a set threshold. Because the True Crown icon itself is always the same, we can always provide the same template. While scales other than the largest will not match exactly, the shape

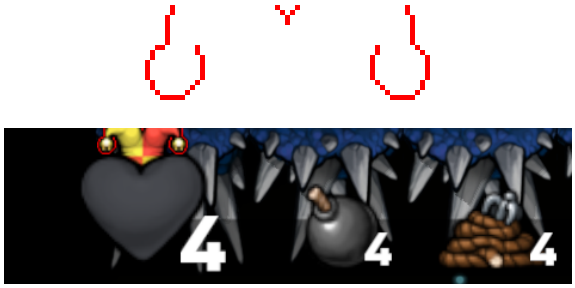

Figure 4. Our template for the True Crown (top) and the template overlaid on the True Crown in the UI (bottom) with the largest UI scaling. We modeled our template off this scale as the top of the crown is cut off in it.

is distinctive enough to ensure that the crown is still found by our search.

As detecting the True Crown reliably is crucial in our program, this system needed to be fine-tuned to ensure we would pick up the crown accurately in as many situations as possible. Unfortunately, Spelunky 2 can look very different from level to level, adding further complexity to the challenges mentioned in 2.1, as the True Crown had to be detected regardless of the level, enemies, or anything else in the background with respect to the UI. To combat this and verify our implementation, we collected over 100 screenshots of gameplay using the Overluky [2] mod to easily put the character onto random levels in random positions. Half were taken with the True Crown item and half without, all with differing UI scaling and opacity settings. A subset of these tests is displayed in 5. Using the testing setup we created, we fine-tuned the threshold at which we consider the template to be "found" via our template matching algorithm, as well as tweaks to the template made to increase accuracy at all UI scales.

Assuming our template match finds a global minima under the threshold set, we use the positional data returned to further confirm that the local region is the correct colors, yellow just to the left of the True Crown's center and red just to its right. Such a color check is prone to falsely flagging orange backgrounds such as the explosion of a bomb item. Hence, it is not reliable enough to be the only check used. However, we have found the template match approach will occasionally flag random images that are from outside the game. For example, we noticed that Windows 10's Recycle Bin icon would get flagged as the True Crown icon simply because the text underneath happened to make edge patterns that put the template under the threshold. As something similar could potentially in happen inside the game, we implemented the color check as a secondary measure to filter out any similar false pos-

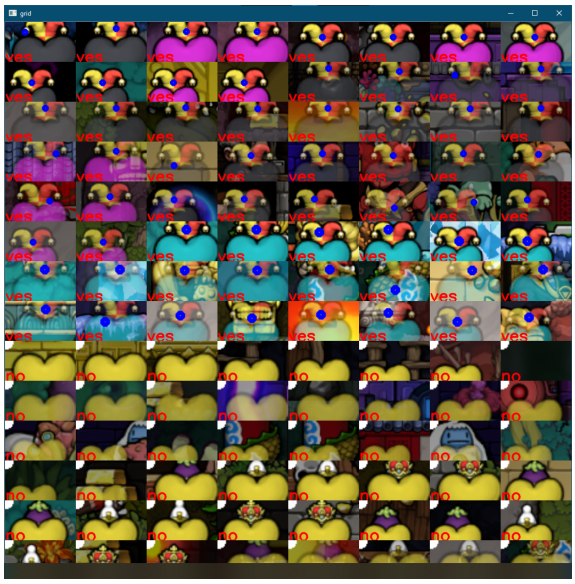

Figure 5. A sample run of our test set. The program finds errors by comparing against given labels for each image but the output of all trials are shown to give an overview of the system. Yes is written for each detected crown, alongside a circle marking its position, while No is written otherwise.

itives. Combined with the template match to quickly categorize most frames, we can accurately distinguish between the player having the True Crown item and not having it.

2.3. Text Detection

To detect text present in the UI, we chose to use the open-source Tesseract program and its accompanying Python bindings [3]. Tesseract is an optical character recognition solution with multiple methods of detecting text available as well as numerous options to tune its algorithms to specific inputs. Tesseract can even be further trained on specific input sets to fine-tune results, though we did not take advantage of that feature for this implementation. Of the many options available to tune (besides direct training), the ability to set character whitelists was important for this project to ensure we could only detect digits and specific characters. Both the timer and the level number have different, restricted character sets so being able to restrict the output significantly improves the relevance of recognized text. While Tesseract is a standalone binary, the Python bindings allow us to seamlessly pass images into the binary with specified command-line arguments and then receive the parsed text as an output directly. These bindings facilitated development and helped achieve real-time processing. Using Tesseract, we are able to parse the game's timer and level number

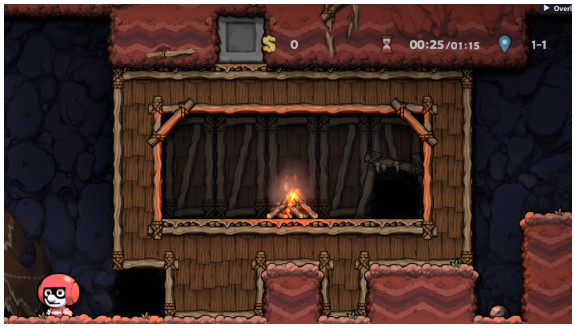

Figure 6. The upper right corner of the screen during gameplay. The players current score, the in-game time (configurable to display level time or both level and total time), and the current level are displayed alongside corresponding icons.

from the top right of the screen, as showcased in figure 6. The former allows us to keep track of the current in-game time while the latter provides information on the stage the player is currently on. Combined, the two provide enough information to allow us to calculate the state of the True Crown timer. If the player picked up the True Crown in the current level, they will teleport on twenty-two second intervals starting from the moment they picked up the item. If the player picked up the True Crown on an earlier level, the crown consistently activates every twenty-two seconds from the start of the level.

While tesseract itself is well suited to arbitrary character recognition, we had significant difficulty in ensuring Tesseract would reliably detect the text on screen. To provide input that Tesseract could better handle, we pre-processed the corresponding regions of the image using OpenCV before feeding them into the program. The processing required is dependent on the (hopefully major) version of Tesseract used, as different versions of the program have different requirements. To make matters more complicated, Tesseract's documentation is still a work in progress; various portions are missing entirely or refer exclusively to old versions with little explanation as to the differences between them. We settled on Tesseract v5.0.0 and used that version exclusively in our program. This ensured our results were consistent, if at times confusing based on the documentation.

Despite documentation suggesting that newer versions of Tesseract work best with white text on black backgrounds [4], we found the opposite to be true. We struggled to detect black text on a white background and instead fed in white text on a black background. Tesseract also prefers a specific dpi (dots per inch). While our screenshots are not print, our screenshots

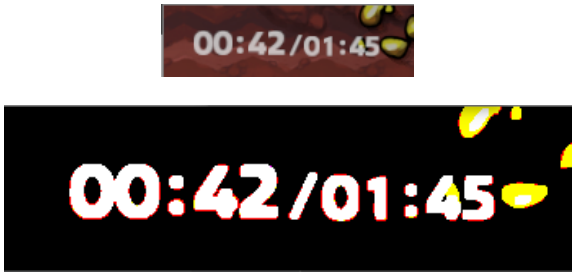

Figure 7. The in-game timer before (top) and after (bottom) our processing pipeline

are very low resolution (50-150 pixels across) which is equivalent to a low dpi; we scaled up our images to compensate. In full, our preprocessing prior to handing the image off to Tesseract is:

1. Scale the screenshot by a factor of 2
2. Invert the image
3. Increase the contrast of the image
4. Erode the image
5. Perform an inverse binary threshold

The final step specifically performs an inverse binary threshold to convert the image to white text on a black background, instead of white text as we originally planned for. The output of this process on the timer is shown in figure 7. As the thresholding is done per channel, non-white objects that are high enough contrast (such as the gold nuggets in the wall) can persist but remain differently colored. In our tests, this has seemed beneficial, as it helps distinguish the letters from any background artifacts that make it through our preprocessing.

For the level number, the same pipeline seemed to be insufficient; the timer was not detected accurately enough. We suspect this is due to the small number of characters—tesseract is tuned for working with blocks of text instead of just a pair of numbers such as 1-1. The level number seemed to alternate between being detected and not detected seemingly arbitrarily, due to minor pixel fluctuations in the processed image. To combat this, we added blurring at multiple stages in the pipeline to ensure a more reliable picture of the number. The result of this is shown in figure 8 Doing so has resulted in workable level detection; this in particular is a future area of improvement which will likely require digging further into the specifics of Tesseract's implementation to make more progress.

Figure 8. The level number after our pre-processing. The level number as additionally blurred and converted to grayscale to aid with recognition

```
while True:
    # if we don't have the crown, check if we do and continue
    if not has_crown:
        has_crown = check_crown(200)
        if has_crown:
            # check levels
            level = check_level()
            if level == "same":
                print("Unpaused, resuming timer")
                # timer setup already
            elif level == "new":
                print("Next level, resetting timer")
                prev_time = -1
                acquire_time = -1
            else:
                # couldn't determine
                print("Couldn't tell if new level, assuming same level")
    # if we do have the crown, start checking for time
    if has_crown:
        cur_time = check_timer()
        # if we can't get the time, set has_crown to false so we re-check
        if cur_time < 0:
            has_crown = False
            continue
        # if we have no previous time get it
        if prev_time == -1:
            prev_time = cur_time

        # Countdown and fire alarms as needed
        # via tracked state
```

Figure 9. A portion of our implementation's main loop

2.4. Final Implementation

In our final program, we detected the presence and absence of the True Crown item as well as the current level time and current level using the methods described prior. A portion of our main function is shown in figure 9. In this figure we see the majority of

our main loop's logic. The core consideration throughout was to avoid running multiple repeated checks in a single iteration to keep the program as performant as possible. Thus, for most iterations, we run only one of check_crown, check_timer, and check_level. Lines three through seventeen handle detecting if the player is holding the True Crown item. If, on this iteration, we find it, we then check the level to determine whether or not we need to reset our internal state. Lines nineteen and onward deal with the actual timing. We grab the level time, converted to seconds and keep track of the previous acquisition time. If the time parsing fails, we assume the game is paused and set has_crown to false. This allows us to use our True Crown detection system to also detect when the game is unpaused; if the time simply failed it will immediately flag the item as held and check the time again. We chose to use the distance transform for this purpose as it is the most reliable of our three detection mechanisms. Assuming we do have the time, we compare it to our stored set of times to fire alarms prior to the True Crown's teleport and alert the user if we are nearing twenty-two seconds since the last teleport, resetting after twenty-two seconds have elapsed.

3. Future Improvements

With the application's core functional, the most immediate improvement would be a better method of communicating warnings to the end user. Currently, the output of the application is written to the console. While this could be useful, having a second monitor to display this on is an uncommon case, not to mention that it is hard to track another window while playing a game. As such, the ideal way to convey this information to the player would be via audio alerts; instead of printing a warning to the console we'd play a sound through their speakers. This would not be hard to implement, but due to prioritizing the core functionality, we were not able to implement it during this project.

In a similar vein, a full UI would also be a major improvement, simply to avoid interacting with the program via solely a terminal or integrated development environment. This would be significantly harder to implement, but would make the program viable as an actual application for the Spelunky 2 community to use. Such a user interface could be extremely minimal, at the most extreme simply just a window with the option to start and stop the system, but in creating one it would be easy to add further options for the user or querying information, such as the current UI setting. Of course, anything like this would also have to have the associated logic implemented internally but the effort may be well worth the improved experience, either on the UI side or in the program's actual performance.

3.1. Detection Improvements

Improving detection and, thus, reliability of our program is centered around text. Our crown detection system is more consistent than the Tesseract system at present, so the largest improvements will be made by improving the reliability of detecting text in the UI. Unfortunately, as discussed in 2.3 doing so will likely require a deep dive into the inner workings of Tesseract; we have reached the limit of what our immediate understanding will allow. While Tesseract is a bit of a black box, we do control the command line inputs and any preprocessing done. The command line inputs have already been tuned to the point where we suspect there is little else we can tweak with respect to them, but our preprocessing could improve.

The main question to answer to improve our own preprocessing pipeline is what preprocessing Tesseract itself does. We have treated the entire program as a black box, but it is possible we are either repeating steps of Tesseract's preprocessing in our own pipeline, which we could then remove, or we are making some steps in Tesseract's pipeline less effective and would need to modify our pipeline. Confirming either of these require a dive into the specific pipeline Tesseract uses. In a similar vein, we could also attempt to retrain Tesseract to better detect the specific font and character set that Spelunky 2 uses. This should increase the accuracy of the program but would require significantly greater understanding of Tesseract's internal model than we currently have. Furthermore, collecting the necessary quantity of data would be a complex task in itself, likely requiring an automated mod for the game which could teleport the user to random locations and take the needed screenshots with random items and configurations. Nevertheless, despite the challenges, further refinement of the detection mechanisms for in-game time and the current level would likely greatly improve the performance of our implementation.

4. Conclusion

Our approach for this implementation—capturing game-state information indirectly through analysis of the screen—is general enough to be applied to other contexts. Many other games or even programs in general have hidden or semi-hidden information. Similar techniques could be used to extract that information and provide it to the player in an alternative form. Such general systems could be used to provide "extra" information to a user or even to accommodate disabilities by presenting displayed information in an alter-

native form. The alternative, recompiling the program to provide alternative methods, is often not possible or not allowed, especially in the case of games which are rarely re-compilable.

Our implementation is not a perfect, user-ready program. There are significant improvements that should be made in the future before it would be ready for use by the Spelunky 2 community. Regardless, the core program is fully functional. With further testing, refinement, and a proper interface for the end user, JesterSight would be a helpful tool for any player looking to push the limits of the game and their own skill, either as a practice tool or, if the community permits, as an aid during record breaking attempts.

References

[1] MossRanking rules! 2

[2] Overlunky. original-date: 2020-11-09T02:35:54Z. 3

[3] Tesseract OCR. original-date: 2014-08-12T18:04:59Z. 4

[4] Tesseract user manual. 4

[5] The true crown. 1

[6] crash. JesterCap. original-date: 2020-11-19T03:14:34Z. 2

[7] Paul V. C. Hough. Method and means for recognizing complex patterns. 2

[8] twiggle. Spelunky 2 - high score $13,235,350 (world record). 1