MappyLand: Fast, Accurate Mapping for Console Games

Joseph C. Osborn[1], **Adam Summerville**[2], **Nathan Dailey**[1], **Soksamnang Lim**[1]
[1] Pomona College
[2] California State Polytechnic University, Pomona
joseph.osborn@pomona.edu, asummerv@cpp.edu

Abstract

We present *MappyLand*, a rewrite and enhancement of the earlier *Mappy* automatic game mapping system, which leverages instrumentation of a game console emulator to produce, from a sequence of game inputs, accurate annotations for action/adventure games including object detection and tracking, in-game camera movement, grid-based tile maps, and links between identified disparate spaces. The overhead of generating these annotations is on the order of one millisecond per observed frame.

We also show a higher latency (but still online) algorithm for merging together previous observations of distinct game maps for the purposes of agent localization across a long period of time. Specifically, our system can determine a consistent graph of game rooms from a set of strings of game rooms, capturing behaviors like backtracking and synthesizing observations from multiple play sessions.

Altogether, this fast, accurate approach to mapping yields new and useful knowledge representations and expedient ways to produce new datasets given just a game and some example play.

Introduction

Mappy (Osborn, Summerville, and Mateas 2017) was a set of techniques and prototype Python program for mapping video game levels via interaction with a game console emulator. Because of its performance issues, off-line algorithm, and its inability to recognize previously visited rooms, we have rewritten it in the Rust programming language and revised its core algorithms to obtain a roughly 1000x performance improvement. Our new real-time and online version is called *MappyLand*, as it now can address not just single rooms and their transitions, but form an accurate map when the same room is traversed repeatedly.

Model-free reinforcement learning has been a popular approach for game playing artificial intelligence over most of the past decade (Justesen et al. 2019). However, it is overly reductive to treat videogame understanding and play as simply a problem of picking the most rewarding move at a given moment. In games like *The Legend of Zelda* and *Metroid*, the player might revisit the same room dozens of times during a playthrough, with different items in their possession or having defeated certain bosses. In *Dragon Warrior* or *Final Fantasy*, the player can wander around the same small region of the world for hours defeating monsters and increasing their characters' statistics. Since these measures of progression are often not shown in a single screenshot of a game (or even in a few minutes of video) they would be indistinguishable to game playing algorithms that work from instantaneous vision alone. Approaches based on exploration and curiosity have begun to do well for simple adventure games like *Montezuma's Revenge* (Ecoffet et al. 2019), given some domain knowledge; but it is difficult to see how this would scale up to larger, longer adventure games with more backtracking.

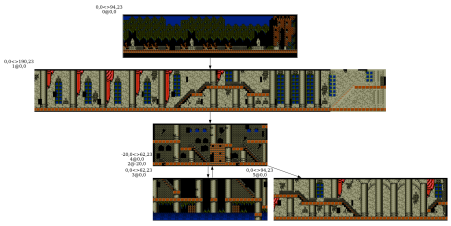

Figure 1: The first level of *Castlevania*, extracted automatically from observations of play.

Many games simulate the movement of characters in continuous or gridded spaces linked together via discrete jumps. Examples include the worlds and stages of *Castlevania* (Fig. 1) or *Super Mario Bros.*, the dungeon rooms of *The Legend of Zelda*, and the network of levels to select in *Bionic Commando* or *Super Mario Bros. 3*.

The underlying assumption that makes it hard to scale these approaches up to more complex games is that of seeing a game as merely a black-box transition system—an over-abstraction that discards important structuring information. Based on the game design theory of operational logics (Osborn, Wardrip-Fruin, and Mateas 2017), we consider action-adventure games as being comprised of *linked spaces* inhabited by *entities*; by focusing on the *Nintendo Entertainment System* (NES) (an easily emulated and well-understood hardware platform from the mid-1980s), we obtain a system which, given a game and a sequence of controller inputs, observes the movements of characters through spaces in an online fashion and efficiently builds a model of the

in-game camera movements and game world at large. This model accounts for spaces which are themselves dynamic (e.g., the destructible blocks in *Super Mario Bros.*) and non-Euclidean (exiting through a doorway and then re-entering it might take you to a new, third room).

Beyond game play, *MappyLand* promises to expedite the development of game-centric datasets like the Video Game Level Corpus (Summerville et al. 2016) and to mitigate reliance on synthetic data (Kim, Kim, and Osborn 2020). It also affords new opportunities for improving the accessibility of games for players with visual perception challenges.

Related Work

The idea of combining a set of static snapshots into a more coherent understanding of the world as a whole is common in the domains of photogrammetry (Ackermann 1984) and Structure from Motion (Ullman 1979). The field of robotics also addresses the problem of Simultaneous Localization And Mapping (SLAM) (Smith and Cheeseman 1986) which tries to solve the problem of accurately constructing a map of the world while also situating an agent's location in that map.

The problem of mapping videogame worlds is, of course, important to communities of game players; games are either mapped by hand (Leung 2017) (often by stitching together screenshots) or by game-specific, purpose-built tools (Hansen 2017). In the former case, the resulting maps are just images—grids of pixels with no particular semantics. In the latter case, richer maps can be obtained, but the process is tedious and hard to generalize as each game stores map data in ROM in its own particular way.

Obtaining high-quality game maps is important for the computational creativity community, especially in the area of *procedural content generation via machine learning* (PCGML). High-quality corpora like the Video Game Level Corpus (Summerville et al. 2016) are time-consuming to create, so some researchers have devised automatic techniques for particular games, making certain simplifying assumptions or assuming foreknowledge of the level components (the terrain and other graphics used in composing the level) (Guzdial and Riedl 2016). Our approach, in contrast, uses no specific knowledge about the game under consideration except the assumption that it is a game where a player controls one or more characters moving in a space.

The idea of taking advantage of a simulation (often games) to obtain high-quality data is well-known in the computer vision community (Richter et al. 2016; Krähenbühl 2018). We note two main differences with this work (besides the two-dimensional setting): first, we are interested in developing not only segmentations, but richer spatial data; and second, we assume slightly less transparency into the operations of the simulated system. Whereas previous work obtains 3D models, transforms, and shader programs for every object along with other API-level notions, we work mainly from pixels with two auxiliary data streams (the positions of certain objects and the movement of the in-game camera).

Some of the techniques used in *MappyLand*—in particular, parts of the in-game camera tracking algorithm—were pioneered in WideNES (Prilik 2020), an emulator meant to produce 16:9 widescreen versions of NES games and assemble maps of game worlds by stitching together screenshots; *MappyLand* constructs richer models and recognizes returns to previously seen locations. Several heuristics (including sprite tracking and room transition detection) are adapted from *Mappy* (Osborn, Summerville, and Mateas 2017). Our key improvements over *Mappy* are computational efficiency (our implementation runs in real time whereas Mappy was off-line only and roughly 1000 times slower), memory compactness, and a more complete knowledge representation that supports the identification of previously visited rooms, a key limitation of the original Mappy.

Mapping in particular is important in the speed-running community, and the idea of instrumenting a game to obtain maps has recently been explored in *wanderbar* (Mandelin 2021), for which an automatic mapping plugin specialized for *The Legend of Zelda 2: The Adventure of Link* was recently developed. Like these tools, *MappyLand* runs with a live emulator in the loop, but it does not rely on any game-specific knowledge.

MappyLand derives abstract game knowledge at the level of game rules and design features from general principles of game design and some instrumentation of the emulator platform. Of these two givens, the latter can be relaxed in the presence of robust detection of in-game camera movement and moving object detection. Notably, *MappyLand* is exclusively focused on the mapping problem and not the exploration problem: it relies on a given sequence of inputs to replay (or interactive play), and our whole emphasis so far has been on recognizing and interpreting observations, rather than on automated exploration of a game's configuration space.

Setting

In this paper, we describe algorithms and knowledge representations for mapping NES games. The inputs comprise a game (a ROM file) and a series of controller states; by making use of the open-source emulator core FCEUMM (CaH4e3 and FCEU Team 2020) with some modifications we can drive the game using these inputs or other, synthetic inputs and obtain data from the running code. As stated earlier, we have constrained our attention to action/adventure games, so e.g. *Punch-Out!* or *RBI Baseball* are outside of the scope of the present work. To apply our approach, the player must control one or more characters moving in a simulated 2D space.

The NES hardware, first released in 1983 in Japan as the Famicom, sported a 1.79MHz CPU, 2KB of RAM, and 2KB of video RAM (NESDev 2017). It would have been impossible for the CPU to both simulate games and render the 61,440 (256×240) on-screen color pixels—storing the pixel data alone as 8-bit RGB color would require 30 times the amount of available video RAM, and the number of pixels to be rendered during a second (at 60 frames per second) is roughly twice the number of instructions the processor could execute in a second. Like the Atari 2600 before it, the NES used specialized hardware (the Picture Processing Unit, or PPU) to convert from a compact representation of

Figure 2: Score displays can use different areas of VRAM.

game graphics to rendered pixels, one raster scanline at a time.

The PPU's video memory stores a grid of 8×8 pixel *tiles* and a vector of (usually) 8×8 pixel *sprites*. While tiles are arranged on a grid, sprites may be rendered anywhere on the screen. To achieve the effect of a scrolling camera (or a split screen), the PPU can offset the drawing of the tiles by a number of vertical or horizontal pixels. Since the 2KB tile grid is too small to hold a complete game level, generally a game with large rooms will update tiles in memory in columns (for horizontal scrolling) or rows (for vertical scrolling) just as the new column or row is about to become visible.

The PPU runs at triple the clock rate of the CPU, and the CPU communicates with the PPU by reading from and writing to particular memory addresses. Based on the approach used by WideNES (Prilik 2020), we have added instrumentation to the emulator to catch reads and writes involving these addresses to observe updates to the hardware scrolling registers of the PPU. This is important not only to know how and whether the in-game camera has moved, but also to help distinguish the playfield from menus or other static parts of the screen. For example, the score display at the bottom of the screen in *Super Mario Bros. 3* (Fig. 2) is positioned in a separate part of video memory from the playfield, and scrolling is reset on the scanline just above it; this leads to some visual artifacts but prevents the user interface elements from scrolling out of frame as the player moves.

While we currently read memory locations to learn object positions (as of the end of the frame), and we monitor memory writes to certain addresses to determine changes in scrolling state (during the frame), we could relax these two constraints given robust vision-based models. In so doing, we could expand our scope beyond just NES games to include other 2D games; although the representations used in the present work do assume a tile-based world with only foreground and background layers that do not scroll independently.

Our system can map NES games with around 1ms per frame overhead on average. In a test with 34,514 frames of input (about 575.2 seconds of gameplay), the overhead due to our system was approximately 1.5ms per frame (emulation on its own took about 0.2ms per frame). Most of this 1.5ms overhead is actually from executing additional emulation steps, as explained in Sec. . Because NES games all use the same hardware platform and thus we evaluate them through the same emulator, we do not expect large variations in performance or overhead for different NES games.

Algorithms

MappyLand relies on a variety of heuristics and algorithms to generate maps from multiple observations. We'll explore these techniques in increasing order of abstraction, from low-level graphical features up to high level concepts like recognizing portions of previously visited rooms. Briefly, starting from the emulated system's memory state and framebuffer (visual output), instrumented detection of changes to hardware scrolling registers (NES-specific), and control inputs, we begin by determining the game's current playfield, scrolling state, sprite locations, and whether the player currently has control. Next, the screen data is interpreted as a grid of tiles and registered (with the scrolling information) onto the current room. Whenever the room changes (perhaps because the screen has scrolled substantially while the player has not had control), the current room is finalized and a new room is initialized. Meanwhile, sprites are tracked from frame to frame (to go from instantaneous locations to long-term records of object identity and movement) and previously visited rooms are merged together into a graph of rooms.

MappyLand operates in a loop, where first input is read from an input source (a physical controller, recorded inputs, or some other source), a step of emulation is performed, and the emulator's state (memory, framebuffer, etc) is analyzed. This is an essentially online process, and the real-time performance means that new applications are possible that were not feasible in the original *Mappy* (e.g., automated support for accessibility or inclusion in a game playing agent).

Examples in this paper come from the authors' play on a representative sample of games. The derived maps are accurate up to human inspection. Because there is no suitable reference data set for game levels, an automated quantitative evaluation is beyond the scope of this paper.

Source code is available for *MappyLand*[1] and the modified NES emulator core [2].

Scroll Registration

The fundamental algorithm on which our system relies splits the screen into viewports and determines the local coordinate system of each viewport. We call this *scrolling detection* or *scroll registration*, as it relates one frame's pixels to those of the next. As mentioned earlier, we hook writes to and reads from certain memory addresses in the emulator (writes to 0x2005 and 0x2006 and reads from 0x2002); these events either manipulate the hardware scrolling registers or reset the latch used to determine whether writes modify the hardware scrolling x or y offset. Whenever such a read or write occurs, we record the scanline on which it happens and the new values of the x and y scrolling offsets.

After running a frame of emulation, we iterate through the recorded scroll register updates and record spans which maintain the same x and y scrolling offsets. Each span is treated as a candidate playfield. Taking the largest such span as the main playfield, we further refine it by splitting off any upper or lower portion which has a thick (24px) solid

[1] https://github.com/faim-lab/mappy

[2] https://github.com/faim-lab/fceumm-libretro-hooks

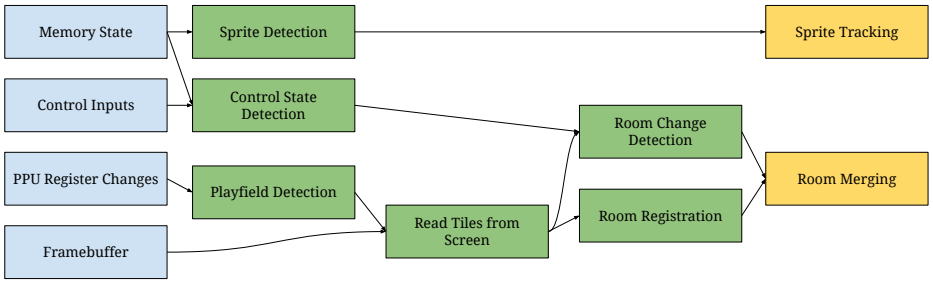

Figure 3: The *MappyLand* pipeline. Blue represents the inputs to *MappyLand*, Green the intermediate steps, and Yellow the output.

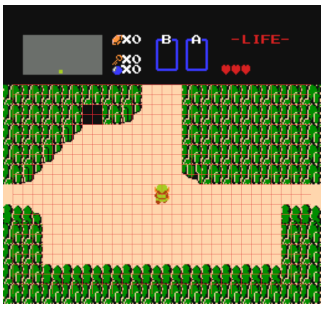

Figure 4: *Zelda's* menu is not part of the playfield (red grid).

color border. For example, in *The Legend of Zelda* we do not want to treat the menu portion of the screen as part of the playfield; while it is usually contiguous with the room the player is in, during transitions between rooms it is kept stationary while the room scrolls out of view (see Fig. 4).

Scroll registration is linear in the number of scroll changes (a small constant) and is dominated by the time taken to scan splits for solid color border backgrounds (which is constant given the thickness parameter). The individual operations are simple, so scroll registration has a negligible cost (less than 0.1% of the total overhead of the mapping system). This is a key algorithmic improvement over *Mappy*, which had to resort to image registration against a rendering of the PPU's nametables, which itself was an error-prone process that would not work correctly for games that switched CHR banks while rendering a frame (e.g., *Super Mario Bros. 3*.

Tile Mapping

Knowing which pixels make up the playfield and what their *grid alignment* is, we can aggregate the pixels into a grid of 8×8 tiles. This is done by iterating through 8×8 pixel blocks of the image, with the origin set to the top-left corner of the first whole grid square, and building a *screenful* of tiles where each distinct tile is represented by a unique identifier. This is more compact than using pixels and allows for quick comparisons of the current screen against the previous frame's. Since we know how much the camera has moved since the last frame and which tiles make up the current and previous screens, we can register the new screen on the existing room to update and extend our map.

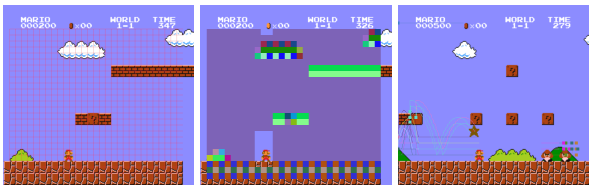

Figure 5: The playfield (left) of *Super Mario Bros.*, the corresponding known tiles (center), and live sprites (right).

Importantly, we cannot observe every pixel of the tilemap—some tiles are obscured by overlapping sprites. We use our oracle for sprite positions to mark these partially obscured tiles as *indeterminate*; indeterminate tiles are not used to extend or update maps (although each room is initialized assuming it is full of indeterminate tiles). See Fig. 5 for a visual explanation of the mapping process, where each tile is visualized based on its ID.

The map itself is not stored as a screenful of tiles, but as a screenful of *tile changes*: observations that we have gone from a tile with ID a to a tile with ID b (for example, from *brick* to *sky*). Like tiles, we remember every distinct observed tile transition, and phrase maps as grids of IDs of particular known transitions. This leads to an asymptotic reduction in memory usage compared to the original *Mappy*, which tracked every instantaneous state of every tile every frame.

This indirection is important to account for levels with dynamic elements—in e.g. *Super Mario Bros.*, if Mario attempts to break a brick while small, the brick tile will disappear and be replaced by a sprite stand-in that plays an animation before returning the tile to its place. If we only stored instantaneous tiles instead of tile changes, we might be forced to conclude that any sky tile could arbitrarily transform into a brick tile; instead, we observe that we can only transition from sky to brick if we had previously transitioned from brick to sky. This strikes an effective balance between storing all the recorded states of a tile (which makes a compact representation impossible) and preserving too little information to disambiguate complex tile transitions.

Tile observations of a room could grow arbitrarily in x, y, and time, so we store rooms internally as a vector of 32×30 tile-transition screenfuls and their locations in world space.

Very wide and very tall rooms only use the minimum necessary RAM to store their map data, and RAM usage does not depend on how long the player spends in the room.

Tile mapping takes amortized constant time, only incurring a linear memory copy when the number of observed tiles or tile transitions exceeds the allocated space. The time taken to read the screen pixels from memory dominates the actual mapping; combined, both account for under a hundredth of the total overhead due to the mapping system.

Sprite Tracking

The NES PPU stores a vector of sprite data; at the end of each frame we read in this vector to know where sprites are on the screen. The role of sprite positions in mapping was explained in the previous section, but there are some quirks of the hardware and game software that we must address.

First, since most game characters are larger than 8×8 pixels, game developers build complex characters out of many simpler pieces. This is evident in the rightmost panel of Fig. 5, where we see several trace lines extending leftwards from the player character; some of these make up Mario's body, while others occupy the blank space above Mario—during certain animations or when Mario changes size, these too might be part of the visible character.

Second, there are certain hardware limits that come into play: on a given scanline, the NES can only render 8 sprites (any subsequent sprites overlapping that scanline are not drawn); moreover, the hardware can only store 64 total sprites in its memory. Game developers therefore do things like reversing the vector of sprites on alternating frames to give the appearance of more than eight sprites on a scanline (thanks to the slow recovery time of CRT phosphors). Since there are a limited number of sprite *slots* available, the same slot will in general be used for multiple distinct entities.

Altogether, this means that we need some trick to decide, for each hardware sprite, whether it is a new entity or a continuation of an existing one (possibly with a new visual appearance, e.g. due to an animation cycle). This is in essence a minimum-weight bipartite matching problem, where on one side we have the known live entities and a special *new* node and on the other we have the new sprite observations. To solve it, we employ a greedy matching algorithm based on our strong prior of temporal and spatial coherence (sprites rarely move more than a few pixels per frame). This minimizes the number of candidate matches.

The cost of a match is related to the distance between the entity's last known position and the sprite's position; the parameters of the sprite (e.g. whether it is flipped horizontally or vertically during rendering); and the pattern and palette IDs used to render it. We create new entity *tracks* when the penalty of assuming a new entity has been created is cheaper than forcing a poor match, and delete old tracks when they have not had a match for several frames in a row. Otherwise, we extend tracks to which we can assign new observations.

Since the number of sprites is bounded, the cost of this step is also bounded; it grows quadratically with the number of active sprites, but in practice this only happens if many sprites occupy the same region of the screen. Tracking accounts for around a hundredth of the system's overhead.

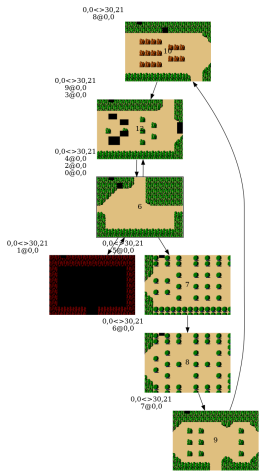

Figure 6: The first few rooms of *The Legend of Zelda* form a strongly connected graph.

Control Checking

By far, the most expensive aspect of our system is determining whether the player is presently in control of the game. A human player develops an intuition for this based on experience with the game and consistency between their given inputs and the game's response; in our case, we exploit the determinism of the emulator. To know if the player has control at a given frame, we take a memory snapshot of the emulated system state (its RAM and registers), then run the emulator forwards to pursue two alternative futures for $T = 17$ frames each: one where the player holds up and right, periodically pressing one button on the controller; and one where the player holds down and left, periodically pressing the other button. If these two lead to worlds that are the same in terms of the playfield layout and the sprites on the screen, then we assume that the player's inputs don't matter at the moment and that the player therefore doesn't have control. This branching of history is possible because of the memory snapshot saving and loading features and because we have an emulator in the loop. After determining control, we reload the original state and continue with mapping.

Because control checking requires that we emulate a large number of frames for every one observed frame, this determination is made every $\mathrm{skip} = 7$ frames. The computational cost is linear in the number of emulated frames, and when averaged across all frames it accounts for all but a few percent of the system's overhead.

This cost is necessary because we do not want to record map updates when the player has no control, and we need to know when control is lost and regained to determine whether the player has moved from one room to another (and to determine which characters, if any, the player controls).

Room Change Detection

Videogame spaces are not real, and they are often non-Euclidean; doorways serve as portals, and spaces can even loop back on themselves. Moreover, moving between rooms can change the game's state (for example, the monsters in

a room might be reset if the player leaves and re-enters a room). For these reasons, any attempt at mapping must recognize that game worlds are made of networks of linked rooms—and we need a means to determine whether we have moved or are moving from one room to another.

Just as cinematography has certain conventions for explaining that a scene change has taken place (the establishing shot, the wipe or fade), so do videogames. On the NES, the two main sorts of scene transition are what we call *instantaneous*—a fade-out and fade-in—or *scrolling*, where the player observes (but does not control!) a smooth sliding transition from the character being at, say, the right edge of one room to the same character being at the left edge of a new room. This is distinct from regular scrolling since this transition takes control away from the player for a time while the camera pans to the new room. Games like *The Legend of Zelda* use both types of transition (Fig. 6): the dark room is entered and exited through a cave mouth via instantaneous transitions, while the other rooms use scrolling transitions.

In fact, the temporary loss of control (lasting more than our empirical parameter $R = 45$ frames) is a hallmark of game room transitions that distinguishes them from regular scrolling movement. During the period between losing and regaining control, one of two things must happen for our system to view it as a room transition: either the new screen's tiles are substantially different from the previous screen's (a Hamming distance of more than $D = 400$ tiles, out of a possible $32 \times 30 = 960$), or more than a screen's worth of scrolling has happened since control was lost.

While control is lost, no mapping takes place; if control is regained and no transition has happened, mapping resumes; and if control is regained and a transition has concluded, the old room is finalized (its local coordinate system's origin is set to $(0, 0)$) and a new room is created. A transition from the old room to the new room is also recorded. Finally, the old room is merged into any previously seen rooms if possible.

The overhead of this step is negligible and bounded by a small constant (the calculation of the Hamming distance between two screenfuls). Room registration is inexpensive even for large rooms because of the data model we use—for any registration, no more than four screenfuls of the room data are read or written, putting a hard bound on the number of memory accesses needed to update the room. While sometimes reallocation of the vector of room screenfuls (or rooms) leads to a linear memory copy, this amortizes to a constant cost and doesn't effect the analysis.

Room Merging

Since every transition might result in a new room, we assume that even if our first glimpse of a room is similar to a previously seen room, it might in fact be a different room we have never seen before. On the other hand, if we enter what seems like a novel room, it could be a previously-unseen part of a room we've been in before. We divide the task of *remembering* places the system has seen before into two phases: matching and merging. Instead of storing just a vector of seen rooms, we also maintain a partial order of *metarooms*. In this ordering relation, a metaroom has an edge to another metaroom if it could be part of that room. Metarooms are defined by an unique ID and a vector of *registrations*—room and coordinate pairs.

After finalizing a room, *matching* computes the optimal matching score and offset (if any) to register the room in question against each metaroom. Since rooms contain dynamic tiles that can be destroyed or changed, matching must take into account not only the currently observed tiles, but whether the tiles of each room could in principle transition into the presently observed tiles of another. Moreover, since one room might have valid registrations in several metarooms, we need to maintain those relationships in case later information disambiguates the situation.

The *match penalty* of a room *R* at some position against a metaroom is the sum of the minimum *change cost* of each tile in *R* against the corresponding tile (if any) of each room already in the metaroom. This is defined as 0.0 if the observed tile change is the same or if this tile is absent in either *R* or the currently known metaroom; as 0.1 if the target tile of the change is the same; or as 0.25 if the tile changes point to each other (one's *from* is the other's *to* or vice versa).

Whenever a tile in *R* has a corresponding tile in any room of the metaroom, we increment a counter. If the number of comparisons exceeds half the size in tiles of the smaller of *R* and the metaroom, and if the net cost is less than some threshold (determined empirically, presently 16.0), then this merge position is valid; the best valid merge is committed to our partial order.

To motivate this complexity, consider Fig. 7, which maps World 1-1 of *Super Mario Bros.* This level features a pipe through which the player can reach a secret coin room and then re-emerge several screens to the right—skipping nearly the whole stage! Since the player cannot scroll the camera left, it is impossible to observe the complete stage in a single play. On one trip the player might take the pipe, and on another trip the player might not; only then could they see that the main part of the stage is one continuous room! Conversely, if the player first played the stage straight through and then reset and took the pipe, they would recognize on emerging from the pipe that they had re-entered the same level and not gone to some different third room. Metarooms give us a mechanism to account for visiting parts of rooms in different orders, remembering the individual visits but still aggregating them into a map.

There is a fundamental ambiguity in room merging, exhibited neatly in the early stages of *Metroid*. Observe that in the partial map of Fig. 8, the corridor in the middle is entered from the top room, exited through the bottom room, and then entered again from the bottom and exited again to the bottom-right. In fact the middle room is *two* distinct rooms, one of which is one screen high and the other several screens high. These two different rooms are merged since they are substantially similar, and while in this visualization it seems that the room exits nondeterministically to one or another room, the underlying data structures have not forgotten that the two rooms are indeed distinct. In the future, we might augment this system to leverage the room exit graph to further disambiguate rooms that seem otherwise identical.

All this flexibility comes at a cost: although merging is infrequent, it is slow enough that it ought to be done in a sepa-

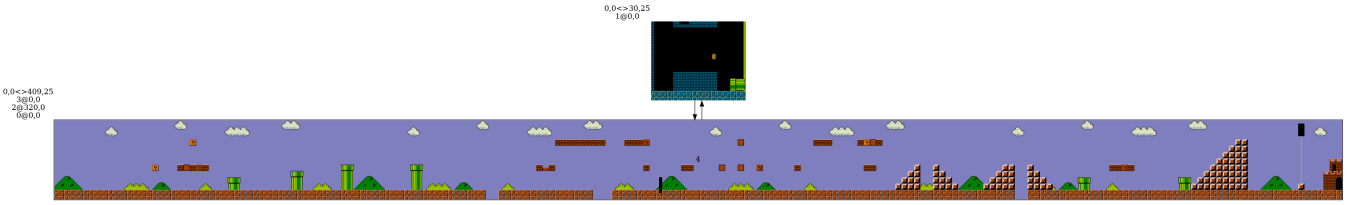

Figure 7: *Super Mario Bros.* World 1-1 cannot be fully observed in a single playthrough.

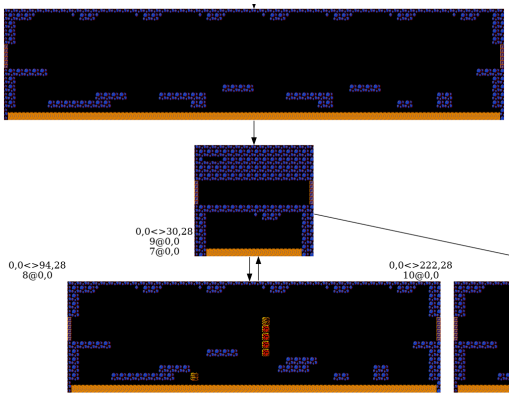

Figure 8: *Metroid* shows an interesting ambiguity in game mapping: the middle corridor is really two distinct rooms.

rate thread or possibly off-line. The template matching algorithm is linear in the number of rooms and quadratic in their dimensions, but is easily parallelizable and offers opportunities for early exit (effectively bounding the average case performance to a constant). In a longer experiment in *The Legend of Zelda* (roughly 10 minutes of play emulated and mapped in 52.3 seconds), the time spent calculating merges among 52 rooms (inducing 73 metarooms), averaged across all emulation frames, was only a few percent of the time spent computing control checks—although it only needed to happen 56 times, as opposed to 4,931 control checks. The net time spent computing merges (3.6s) was about half of the net time spent determining control (7.7s).

Conclusion and Future Work

We have presented an efficient automatic mapping system which derives accurate maps from gameplay. We have already used this system to produce training data for a convolutional neural network model classifying camera movements in games; the kinds of maps *MappyLand* creates have also been employed in previous research, so this should facilitate the production of high-quality game level datasets.

The key limitation of our system from a knowledge representation standpoint is that it mainly applies to 2D spaces with a single physical *layer*—while effects like bridges over walkable floors are possible to represent (a *bridge* tile might imply walkability whether above or below a character), we cannot yet address cases where parts of the level scroll independently of each other. While we could represent a room as a sort of stack of room layers (each of which with its own coordinate transformation), properly segmenting the room layers from pixels alone is difficult. Some game hardware (notably the Super Nintendo Entertainment System) has built-in support for such layers and in principle we could interrogate the hardware for help there, but in general the problem requires a robust computer vision model.

Immediate future work includes coalescing hardware sprites into game entities and determining which of these are controlled by the player. This will involve looking for hardware sprites which tend to accelerate in the same directions as nearby neighbors and noticing which entities tend to accelerate according to the player's directional input. This *avatar detection* could reduce the cost of control checking.

Opportunities for the present work include connecting it with previous research in learning game design information from observations of play, notably CHARDA which derives hybrid automata models of game characters and infers causal relationships between game events and character behaviors (Summerville, Osborn, and Mateas 2017). Using our system as a feature extraction step for a reinforcement learning algorithm is also a natural move, as would be leveraging its representations for automated exploration in the vein of Go-Explore (Ecoffet et al. 2019).

Generalizing to other game consoles with similar constraints (i.e., a single background layer beneath sprites) should also be straightforward. Examples include the Nintendo Game Boy, Sega Master System, and MSX platforms among others. On the other hand, the Super Nintendo or Sega Genesis would require solutions to the problems of multi-planar rooms and parallax scrolling backgrounds.

Ethical Considerations

An important application of this work is to improve the accessibility of older games: our approach reveals the positions of characters on the screen and distinctions between different sorts of terrain tiles. In exciting recent work, Aytemiz *et al.* suggest the use of AI techniques for making games more inclusive (Aytemiz et al. 2020), and we feel that our work could be a means to that end. Mapping is also important for problems of preservation and archiving, especially for games which are difficult to emulate correctly.

By the same token, automatic mapping could pose some copyright concerns since it facilitates the extraction of game assets including images and maps from games. We argue that the potential benefits to accessibility outweigh these issues, since there are plenty of ways to extract such assets already and sufficient mechanisms for enforcing copyright employed by game publishers.

References

Ackermann, F. 1984. Digital image correlation: performance and potential application in photogrammetry. *The Photogrammetric Record* 11(64): 429–439.

Aytemiz, B.; Shu, X.; Hu, E.; and Smith, A. 2020. Your Buddy, the Grandmaster: Repurposing the Game-Playing AI Surplus for Inclusivity. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

CaH4e3; and FCEU Team. 2020. Nintendo - NES/Famicom (FCEUMM). URL https://docs.libretro.com/library/fceumm/.

Ecoffet, A.; Huizinga, J.; Lehman, J.; Stanley, K. O.; and Clune, J. 2019. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995* .

Guzdial, M.; and Riedl, M. 2016. Game level generation from gameplay videos. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Hansen, K. 2017. Metroid Level Data Explained. URL http://www.metroid-database.com/m1/lvldata.php.

Justesen, N.; Bontrager, P.; Togelius, J.; and Risi, S. 2019. Deep learning for video game playing. *IEEE Transactions on Games* 12(1): 1–20.

Kim, C.; Kim, J.; and Osborn, J. C. 2020. Synthesizing Retro Game Screenshot Datasets for Sprite Detection. In *2020 Experimental AI in Games Workshop*.

Krähenbühl, P. 2018. Free supervision from video games. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2955–2964.

Leung, J. 2017. The Videogame Atlas. URL https://vgmaps.com/Atlas/.

Mandelin, C. 2021. Wanderbar SNES v1.0 + Wanderbar NES v1.0 + Wanderbar GB/GBC/GBA v.10. URL {http://tomato.fobby.net/wanderbar/}.

NESDev. 2017. NESDev Wiki: NES Reference Guide. URL http://wiki.nesdev.com/w/index.php/NES_reference_guide.

Osborn, J.; Summerville, A.; and Mateas, M. 2017. Automatic mapping of NES games with Mappy. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, 1–9.

Osborn, J. C.; Wardrip-Fruin, N.; and Mateas, M. 2017. Refining Operational Logics. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*.

Prilik, D. 2020. WideNES. URL https://prilik.com/ANESE/wideNES.html.

Richter, S. R.; Vineet, V.; Roth, S.; and Koltun, V. 2016. Playing for data: Ground truth from computer games. In *European conference on computer vision*, 102–118. Springer.

Smith, R. C.; and Cheeseman, P. 1986. On the representation and estimation of spatial uncertainty. *The international journal of Robotics Research* 5(4): 56–68.

Summerville, A.; Osborn, J.; and Mateas, M. 2017. Charda: causal hybrid automata recovery via dynamic analysis. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 2800–2806.

Summerville, A.; Snodgrass, S.; Mateas, M.; and Ontanón, S. 2016. The VGLC: The video game level corpus. In *Proceedings of the 7th Workshop on Procedural Content Generation*.

Ullman, S. 1979. The interpretation of structure from motion. *Proceedings of the Royal Society of London. Series B. Biological Sciences* 203(1153): 405–426.