The glEnd() of Zelda

Dr. Tom Murphy VII Ph.D.[*]

1 April 2016

Abstract

3D ZELDA

Keywords: small keys, boss keys, dungeon keys

1 Introduction

1986. Hyrule. The Legend of *frickin'* Zelda for the Nintendo *freakin'* Entertainment System. Need I say more? A *god damn* gold cartridge. Fortunes made just from melting the gold cartridge down to make gold teeth grilles, after carefully extracting and preserving the even more precious ROM inside. A die-cut hole in the box so that you could get a peek of the cartridge and presage that you were getting some solid gold. A die cut little window that you could palpate through the wrapping paper on Christmas Eve, presaging some epic thumb blisters in store for the coming weeks. Koji *freckin'* Kondo. Koji Kondo whipping up a nice 8-bit arrangement of Boléro as the theme music until realizing at the last minute that this music was copyrighted[1] and so instead composing its epic theme in *one day*??

A gold cartridge that contained ROMs and a little swallowable battery to keep the onboard SRAM powered up so that it could retain your epic save game. A battery designed to last 70 years. Nothing could cause you to lose your save game, even once you were half way through the Second Quest. Unless your little brother starts a completely new game and saves over your slot, earning him one of the most righteously deserved clobberings this side of Inigo Montoya. Saves right on top of your slot with a player called just `A` . Right over your slot, erasing it, and hasn't even picked up the *sword* yet. All you have to do is step on the black square in the first room and the guy gives you the sword and then dies. Jeez pooleaze. Saves right over yours with no sword even though there are TWO OTHER UNUSED SLOTS and you even wisely put your game in the third slot *exactly to avoid this kind of calamity.*

Figure 1: Technical diagram of mathematical equations.

[*]Copyright © 2016 the Regents of the Wikiplia Foundation. Appears in SIGBOVIK 2016 with the danger of going alone of the Association for Computational Heresy; *IEEEEEE!* press, Verlag–Verlag volume no. 0x2016. 255 Rupees

[1]Perhaps ironic, since Ravel's Boléro itself was composed as a result of Ravel getting cop-blocked.[1] And surely some Zelda knock-off since then has included a Muzak ersatz of the Zelda theme! What is the longest documented sequence of compositions due to Copyright restrictions?

Need I say more? Shall I say it? Apparently so. KIDS THESE DAYS. Kids these days and their three dimensions. You can certainly make a respectable case that the Zeldest of the series was *Legend of Zelda III: A Link To The Past*, or even *Legend of Game Boy Zelda: I forgot What It's Called*. But there seems to be an established consensus that *Legend of Zelda LXIV: The Ocarina of Time* is the true masterpiece; that while the NES Original indeed invented the genre, the technical limitations of the system handicapped it. You're talking to me about technical *frockin'* limitations? Like somehow appreciating the subtle wonder of the universe and its mechanics requires me to look at 3D tree people with tetrahedric clubs for hands. And they say this of the 64th game in the series? Please jouize.

A fine game, don't get me wrong. But was its theme song composed in one day? Did it come in a gold cartridge?

And the thing is: The original Zelda was built with enough forethought that it *can* be rendered in first-person 3D. So here I will make the game 3D so that it can be enjoyed by these kids and they can get off my lawn, tapdancing around playing their titular Ocarinas. I will make it possible to play the game in glorious HD, or even 4K resolution, and then some (Figure 1). And while we're at it, why not try to do it in a way that can play loads of NES games in three *fröckin'* dimensions? What's the worst that could happen?

2 Technical limitations

As presaged, the NES does have certain technical limitations, which I prefer to think of as constraints. Some basic details are important for understanding how this works.

The NES has a modest CPU that executes game logic. It's a little 8-bit baby puppy with access to 2 kilobytes of RAM. If you think about it, the NES's 256x240 pixel screen is already 61,440 pixels, which if it used the entire 2k of RAM to represent it, we'd have only $0.2\overline{6}$ bits of RAM per pixel, which makes no sense. At 1.7 MhZ, and a generous 1 instruction per pixel, we'd only be able to render 28 frames per second! This doesn't make sense, and this is because the CPU does not render the graphics on a NES.

The NES also contains a custom Picture Processing Unit, or PPU, the 8-bit baby puppy's baby ppupy brother. This thing outputs 60 frames of NTSC video per second. In fact, since the PPU also drives a CPU interrupt for each scanline and vblank, and most games use this for timing,[2] in many ways the CPU is sub-

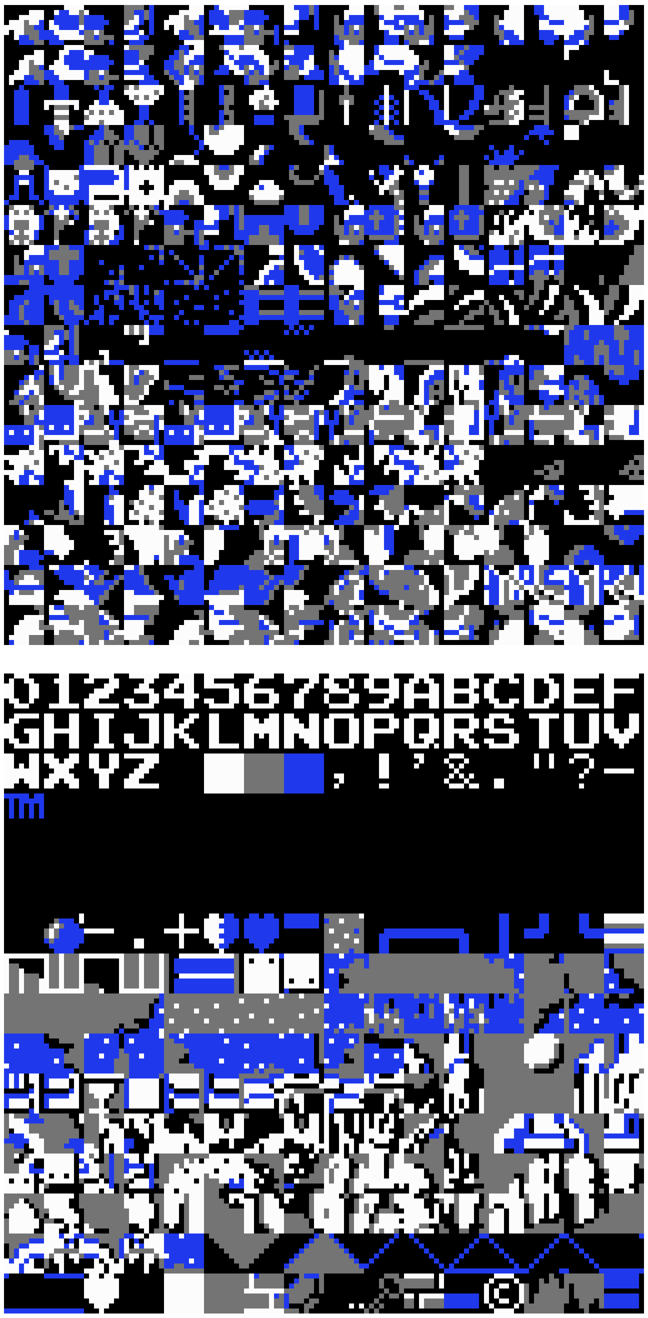

Figure 2: Pattern tables for the Zelda overworld. In Zelda, the top tiles are used for sprites, and the bottom are used for the background graphics. All tiles are 8x8 and use 4 color values (color 0 is transparent). These color values index into 4 sprite palettes and 4 background palettes, with some limitations.

[2]Fun fact: NTSC ("Never Twice the Same Color"), the video

servient to the PPU, kind of like the PPU is the *true* pure heart of the NES. In the context of this paper, you could say that the CPU is the Ocarina of Time and the PPU is the Legend of Zelda. It's just an analogy. The PPU is very complicated and has tetrahedra for hands, like clubs. The CPU communicates with the PPU not by drawing pixels, but by giving a high-level description of the screen (by writing it into PPU memory). The chief insight of this work is that the PPU representation can be interpreted and rendered in 3D instead of 2D. This allows us to use the exact original game logic and only change the way it's viewed. Sometimes this even allows us to see aspects of the game that are actually present but were not visible on NES hardware. Often it makes the game more difficult by making things invisible (enemies behind the player or behind walls) and inducing motion sickness.

2.0.1 Background

The PPU has two major drawing facilities: Background and Sprites. Both are made of "tiles", which are 8x8 graphics that are usually[3] stored in ROM. There are two tile sets, each with 256 tiles. Figure 2 shows the tile sets for the starting location in the Zelda overworld.

The NES background is described by the "nametable", which is an array of 32x30 bytes giving the tile numbers to fill the screen with. It's two rows shy of 256 probably for some reason having to do with the aspect ratio of TVs or number of actual NTSC scanlines, but also to make room for 64 bytes of attribute data. There are just two bits per four tiles. These two bits select which of the four palettes to use for each 2x2 group of tiles on the screen. It's an interesting hobby to look for the tricks that artists use to work around the constraint on the number of background colors.

Because of this array of tiles, NES games have to be built of high-level, block-like structures. In Zelda, this structure is particularly (but not unusually) regular. Since each 2x2 tile group (16x16 pixels) shares the same palette, the screens are built around 16x16 pixel grid-aligned blocks.

standard used in the Americas, has a ∼60 Hz frame rate. But PAL ("Pointless and Lousy"), used in Europe and Asia, has a ∼50 Hz refresh rate. This means that many games run 16% slower in Europe than the US, so if you grew up playing there, you were playing on *Easy* mode. ;-)

[3]Some games trick the NES hardware by remapping tile memory (e.g. in response to a memory read), even during a scanline. I do not allow for such chicanery in this work.

2.0.2 Sprites

Sprites, named in honor of the forest pixies whose homes were razed in order to make space inside the PPU for them, are more complicated.

There are 64 sprites, which are all turned on or off together. Each sprite is described by four bytes: Its x and y coordinates (these can be placed at arbitrary integer pixel locations), the index of the tile to draw, and an attribute byte:

	bit #	bit deets
MSB	7	Vertical flip
	6	Horizontal flip
	5	something weird
	4	I don't know
	3	unused? who can say
	2	bit 2; could be anything
LSB	1,0	Palette index

Additionally, there are some global sprite settings. The only one that is important for this presentation is the "tall sprites" flag. If it's set, then sprites are 8x16 (drawn of two consecutive tiles) instead of 8x8. Sprites are drawn in a particular order. Barring some bizarre mischief, all of the sprites are drawn; you can't turn them off individually. Instead, you position the ones you don't want off the screen or make them be completely transparent. In order to make things more challenging, if there are exactly 8 sprites on a particular scan line, some really weird stuff happens.

Sprite Zero has some special properties. I don't want to tell you about the properties, I just wanted to make a soda joke.

3 Three to D—u and v[4]

The game runs in an emulator, a version of FCEUX[2] that I have hacked up almost beyond recognition. In normal operation, it

- Emulates a complete frame of the NES. This includes the CPU and PPU operation for all 240 scanlines. The execution of a frame depends on the current controller input (which doesn't change as it runs);
- Extracts scene geometry and textures. This consists of two pieces: Boxes and Sprites;

[4]Spoiler alert because I think this joke may be too abstruse. "Free to Be You and Me" was a semi-famous kids record (here meaning actual vinyl) by Marlo Thomas. D is "dimensions." u and v are common names of the axes for texture coordinates in computer graphics.

- Extracts the player location and orientation;
- Moves texture data to the GPU;
- Transforms the scene geometry into a textured model, sets the camera based on the player location, and renders.

I'll describe the interesting parts of these.

Emulation. Well, this is not that interesting. I give the emulator its controller input and it does its thing, modifying the RAM and PPU and CPU state. I have been working on improvements to emulator technology but these are independent of this paper and better described elsewhere.

Extracting boxes. Yeah! Okay! Boxes are the three-dimensional version of squares, and are what give the game its 3Dness. I read the nametable in chunks of 2x2 tiles, which is the size of one overworld tree (and the same size as Link). Recall that Zelda uses this grid size, probably because color attributes can only be set at this scale. These become $2x2x2$ cubes. The x and y coordinates are obvious (pixel positions divided by 16, since each 2x2 tile block is 16x16 pixels), but what should z be? Zelda and other games tend to use tiles quite consistently to represent walls, obstacles, floor, empty space, and so on. This makes sense because a tree graphic, for example, suggests something to the player. So we simply build a file, `zelda.tiles`, which maps tiles (by index) to one of `wall`, `floor`, `rut`, or `unknown`. Floor blocks are placed at $z = 0$, which will be under the player's feet (we typically only see the top face). Walls are placed at $z = 2$, so that their bottoms are coplanar with the floor. Rut blocks are placed at $z = -0.5$; this is used for blocks that are part of the "floor" but that can't be walked on, like water. Because each box is actually made of 4 tiles, we just use the maximum z value for the constituent tiles (they should normally be the same). It would make sense to develop heuristics for when games use tile-level resolution, even to generate primitives other than cubes.

This approach is a bit unsatisfying because it requires us to have specific knowledge of the game as well as do some manual labor. However, there really aren't that many tiles, and it's kind of fun. (In fact, you really only need to pick out a handful of floor tiles before the game is totally playable.) There is another problem, which is that many games switch tile sets, which may change whether tile `0x84` ("desert dots" in the overworld) is floor or wall. What happens if you go into a dungeon in Zelda, for example? I didn't even try that yet but I bet it's nuts! This can easily be fixed by identifying distinct tile sets (for example by using the source ROM address, or a hash of their content) and mapping them all.

However, there's some hope that this can be done in a much more automatic way. I describe an idea in Section 4.3.

Box textures. The game is pretty understandable with just TRON-cubes, but it is better to texture them. We put the same texture on each face of a box; that texture is just the 16x16 pixel graphic consisting of the 4 tiles that made it, in the appropriate palette. It does *not* work to generate a bunch of tile-sized boxes (with the individual tile texture on all sides); these will only look right on the top face. Indeed, it is possible to get weird texture mismatches between these 2x2x2 boxes for basically the same reason. It usually looks pretty good, though.

Conceivably, every box on the screen could have a different texture, and conceivably these all change every frame. It is inefficient to move 960 distinct textures to video memory every frame, so instead we actually just move the rendered background (256x240) itself and record with each box its texture coordinates within. This is *much* faster, though it can create artifacts at the edges of boxes as an adjacent pixel peeks into the texture. I checked out Ocarina of Time and it is *loaded* with texture seam issues, so I figure this is how they like it.

Extracting sprites. Sprites are the most difficult. Aside from just being more fiddly to implement, it is tricky to recover an important high-level structure. Although we know the location of each individual sprite and the two tiles (Zelda uses "tall' 8x16 sprites; see Section 2.0.2) that make it up, most game objects (like enemies) are in fact drawn with *two* sprites. These sprites could be in any of the 64 sprite slots (it's not at all obvious how Zelda allocates these, or that it would be similar for other games). Adding to the complexity, sprites like Link's sword or the pulsing balls of plasma that frogmen shoot at you from the water *just for being an elf* are just a single sprite, and bosses can be like 6 or more sprites (Figure 3). If we promote sprites to 3D without knowing which ones are supposed to be part of the same object, then the enemies get cut up along their sprite seems and it looks super wrong.

To fix this, we perform *sprite fusion.* We say that two sprites are part of the same object if they share

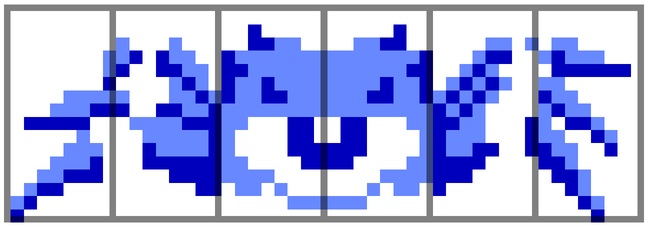

Figure 3: Link's spider nemesis, made of six 8x16 sprites. The sprites have adjacent edges, so they end up being fused into one merged sprite.

a complete edge (transitively). Since this allows the construction of very tall (64×16) or wide (64×8) sprites, we have to make sure that our textures are large enough to contain large sprites. Rather than try to pack them into a texture at runtime (computationally expensive since they can also be weird shapes), we just pre-allocate one texture of the maximum size for each sprite slot (16MB video ram; nothing). Sprite fusion, computation of the sprite size, drawing the sprites, extracting the geometry, and moving the textures to the video card are all intertwined. The simplified algorithm is

- Create 64 "pre-sprites", which are baby sprites. These know their coordinates and have the index of a "parent" pre-sprite (initialized to -1).
- While filling these in, consult a hash-map to see if there are already pre-sprites exactly aligned in each of the cardinal directions. If so, use a union-find–like algorithm[3] to efficiently update its parent index; each fused sprite has one canonical representative whose parent index is -1. Add the pre-sprite to the hash-map.
- Another pass pushes the min/max bounds to the canonical representative, so that we know how big the fused sprite is.
- Another pass draws all sprites into the texture data for their canonical representative, at the appropriate location. This pass must happen in the same order as on the NES hardware, so that overlay effects are correct.
- Finally, create proper sprites to return for each of the fused sprites. Copy the used region of the texture data for these to the video card.

Two types of sprite rendering are supported: in-plane and billboard. In-plane just draws the sprites on the floor or some other xy plane of your choice. (This does not actually need sprite fusion, but it is not harmful.) Billboard sprites are drawn so that they face the player, like in Wolfenstein 3D. These are positioned at their xy centroid and positioned in z so that the bottom of the sprite is touching the floor.

Extracting location. Most NES games use very simple layouts for their game facts (i.e., global variables at fixed memory locations), since the processor is simple and memory light.[4] In Zelda, Link's x coordinate is at `0x70` and y at `0x84`. Since this is his top-left corner, we add 8 to each to get to the middle of the 16x16 pixel boy.

We also have to face in the camera in the same direction as Link, to make the perspective first-person. Link's orientation is at memory location `0x98`.

Link moves smoothly (with pixel resolution) around the screen, but turns abruptly. This is sickening to watch. I apply some smoothing to Link's angle, by treating the location from memory as a "target" angle and twisting towards it (along the shortest angular path) at a limited speed.

Knowing what memory locations hold the player's position and orientation also involves some manual work. You could imagine heuristics for trying to determine this, by seeing which bytes tend to go up when pressing right on the controller, and so on. However, it's easy to find these values manually or to even look them up for popular games.

Rendering. Finally, we can render the scene. I used "classic" OpenGL interface, even though it is terrible, for the sake of making good on the excellent pun that is this paper's title. The final result looks like Figures 4 and 5.

Rendering with OpenGL is quite straightforward given the geometry extracted from the previous sections. I try to preserve the pixel aesthetic by using `GL_TEXTURE_MAG_FILTER` = `GL_NEAREST`, but at this point we can use whatever approach to 3D graphics most titillates the youth. Full-screen multisample anti-aliasing? Sure. Screen Space Ambient Occlusion? I hear that this creates physically realistic shadows on the cheap, sure. Stencil buffer shadows? Volumetric fog? God rays? Depth-of-field? Lens flares? You got it.

3.1 Quirks

There are some problems or quirks with the current approach.

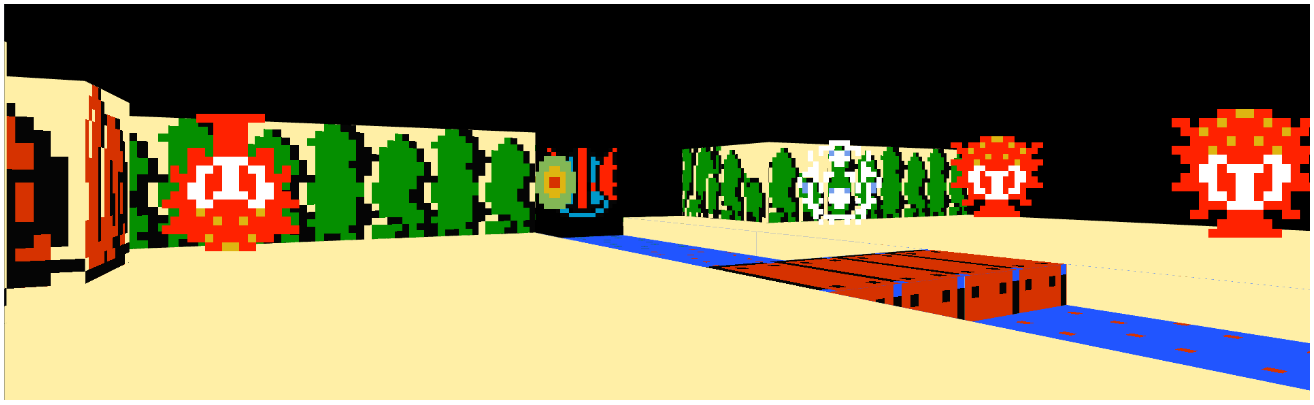

Figure 4: A representative view of the Legend of Zelda in glorious 3D.

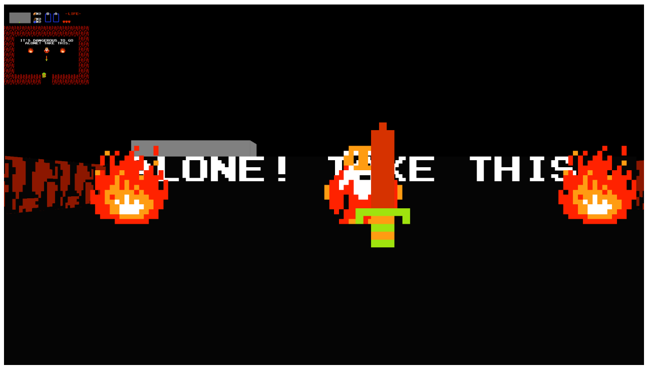

Figure 5: The first cave, where this man gives you a sword and then dies, rendered in 3D. Letters are treated as walls so you can see 'em upright, but then of course the second line is in front of the first. The 2D NES screen is superimposed on the top left for comparison.

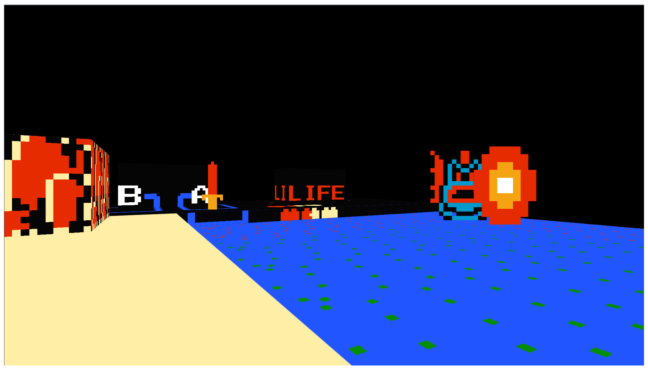

Figure 6: The game menu visible to the North because the ocean does not get in its way. Did you know that if you were able to see beyond the cluttering trees and mountains at the horizon of Earth, you'd see your own status menu in real life?

Extramural geometry. In Zelda, some of the tiles on screen are used to draw a menu at the top, which is not part of the gameplay area. This still gets interpreted as part of the geometry of the world, of course (Figure 6). It would be a simple matter to crop the region of interest (especially if done manually), and it would even be natural to overlay it on a heads-up display. But I like it over there. Why shouldn't menus be 3D as well?

Positive self-image. We place the camera at Link's position and orientation, so that the player "becomes" Link. However, Link is still represented by a sprite in the game, and we do not treat it specially. This means that the fused Link billboard is always in the scene geometry right at the location of the camera. The upshot is that whenever the camera is facing south, an enormous Link graphic fills the entire screen, reminding you that he has always been following a few pixels behind you, haunting you (Figure 7). This sprite could be rejected by seeing that it is coincident with the player's location (which we extract) or by its particular content. We could also set the "near plane" so that nearby sprites are not visible. However, the everpresent looming spectre of a ghost mimicking our every move is a good reminder of our mortality, so I kept it.

Surprising sprite fusion. Sprite fusion works well for merging sprites that are part of the same logical object in Zelda. (It may not work as well in other games where e.g. a monster's legs are moved independently from its body.) However, it occasionally mistakenly

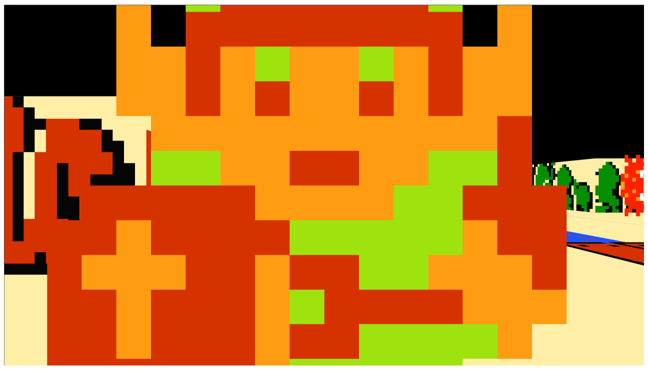

Figure 7: Look who's behind you! WE ARE FRIENDS FOREVER `:-) :-)`

fuses two sprites that just incidentally share an edge (Figure 8). This usually persists for just a tiny moment since the sprites are moving. But who's to say that as those monsters high-five, they aren't forming a momentary totem? Only Shigeru *froakin'* Miyamoto can say that isn't what he intended.

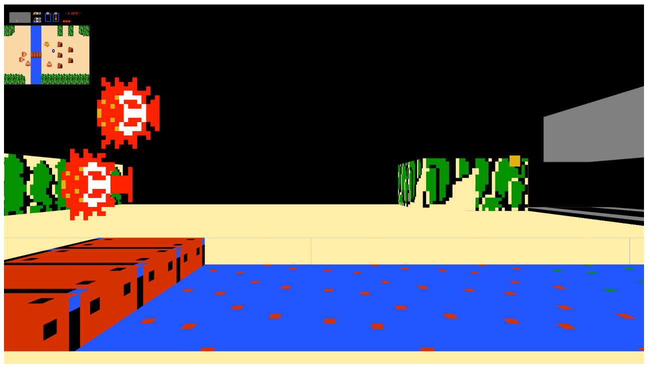

Figure 8: Two Octoroks that were incorrectly fused because they have constituent sprites that have adjacent edges (the fused sprite shape is not rectangular). The formation is a Hexadecirok.

Texture orientation. The textures of both backgrounds and sprites often have strange and inconsistent perspective ideas in video games. For example, the Octorok points its nose in the direction it's moving, because it is always viewed "from above". However, when Link is facing south, you see his front (not the top of his head facing down), and when facing west you see his side. This doesn't make sense, and makes it impossible to apply any consistent rule for mapping sprites into world orientation. This mostly just produces humorous results, like when Link throws his sword and it moves away from him in first person view: It may be pointing towards the sky, towards the floor, or to either side, but never in the direction it's thrown. Even if we were to render the sword "in plane", its handle should always be oriented towards the camera (which would not be the case). No single policy will work for all cases, but perhaps heuristics can look for the use of the horizontal and vertical flip bits to figure out what sprites are rotatable, and then rotate them relative to the camera.

4 Future work

This work is literally in progress, and due to Draconian SIGBOVIK deadlines, this paper is figuratively already out of date by the time you read it. You can check out the project's home page

`http://tom7.org/zelda`

and maybe download it and play it, and probably watch a video demonstration. There may be some new features. Depends what I get to, you know?

4.1 Control

Currently, the game is viewed first person but played with native controls; this means that pressing "up" always moves link to the North, turning the camera if necessary. This is super weird. However, it is pretty easy to remap first person controls into native controls, as long as there is a way of implementing that intent (for example, it is not possible to "strafe" in Zelda). We can change the camera angle arbitrary, implementing "mouselook" or virtual reality support. Since the z axis is otherwise unused, we could allow the player to jump, though this would do nothing in the game itself.

4.2 Other games

This program was built in such a way that it should take minimal effort make other NES games 3D, perhaps with humorous results. There are some issues that need to be dealt with. A common type of game is the side-scrolling platformer, like Super Mario Bros.. At least two important issues are not yet handled: Mario has a different coordinate system than Zelda (it is "side view" rather than "top down"), so we need to support that rendering style and control/camera scheme. No big deal. Second, this game scrolls the screen rather than using single-screen layouts; this uses the PPU in a different and somewhat more complex way. Many games will use advanced PPU tricks that cannot be interpreted in a generic high-level way.

4.3 A Link to the cast

One dissatisfying problem with the current approach is that it requires some manual effort to map different tile types to geometry. One could imagine various heuristics (even a image classifier) to provide defaults or to remove the manual input entirely. An exciting angle is to use the game's own understanding of these tiles. The main thing that differentiates a "wall" from a "floor" is whether the player can walk over it. Clearly somewhere in the game's code it "knows" what is floor and what is wall. We could extract that directly, but this would require a different manual step. Alternatively, we could determine it experimentally. A simple heuristic would keep track of what tiles the player has walked on. But we are not limited to what the player actually does; since we are emulating, we can also emulate counterfactual scenarios. Imagine that as we draw a first person view, we cast lines out from the camera towards the geometry, as in a ray tracer. Except that instead of tracing light rays until an intersection with scene geometry, we trace a hypothetical player path by emulating the player moving in that direction, until Link can move no more. This only requires understanding the bytes that correspond to the player's location (which we already need) and how the controller works.[5] Testing contact with sprites might give us useful heuristics about how they should be rendered as well.

There are many ways we can imagine this going wrong, but probably spectacularly so. Interestingly, this would allow us to render parts of the game that are not even visible in the PPU yet, since emulating the player's path ahead far enough would cause the game to switch to future scenes.

5 Conclusion

In this paper I made Zelda three-dimensional so that it can be enjoyed by the kids. I am fully aware that these "kids" are actually also full-grown adults who themselves resent a slightly younger generation who inexplicably think of Legend of Zelda Twilight Princess HD as the canonical episode; these people should get to work at enhancing Ocarina of Time to add Amiibo support. While you're at it, get rid of the really long section where you have to be a dog. I didn't like that part.

[5]Most rays are diagonal. We can implement a north-northwest diagonal by repeatedly moving two pixels north, then one west. We could also set the player's position directly to a putative destination and then move in cardinal directions to test solidness.

References

[1] Wikipedia, "Boléro — Wikipedia, the free encyclopedia," 2016, http://en.wikipedia.org/wiki/Boléro.

[2] adelikat et al., "FCEUX, the all in one NES/Famicom emulator," http://fceux.com/.

[3] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, no. 2, pp. 215–225, Apr. 1975. [Online]. Available: http://doi.acm.org/10.1145/321879.321884

[4] T. Murphy, VII, "The first level of super mario bros. is easy with lexicographic orderings and time travel. after that it gets a little tricky," *SIGBOVIK*, pp. 112–133, April 2013.