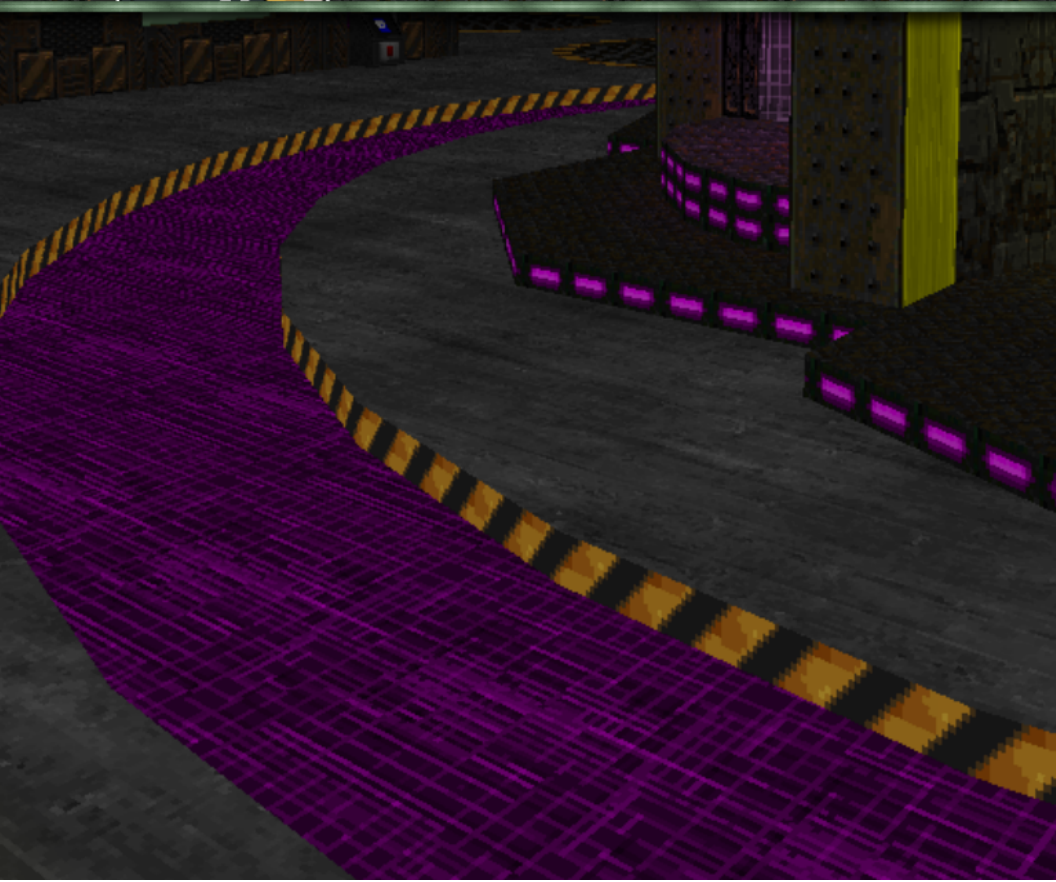


RABBIT'S ALL-COMERS MAPPING PROJECT



ROOMP



# Backstory

RAMP was the first Doom community project I ran, held during June and July 2021. The idea was to encourage beginners to create things and let them contribute to a collaborative project right away - and I was also interested in experimenting with some ideas I hadn't seen people try in Doom projects before. This is the story of how it was pieced together!

## *The Dawning of Doom*

The first time I encountered Doom, I must have been about eleven years old (and don't worry, you're not going to get my entire life story here). It was 1994, and the best computer in the house was my dad's 66MHz 486 that had a newly installed double speed CD-ROM drive.

My siblings and I shared a last-generation computer, and up until then our PC games had usually come from magazine coverdisks or whatever floppy disk sized ZIPs my dad had found in the shareware libraries he had access to at the university where he lectured. But now with the advent of CD-ROM, an unimaginable amount of content was available to us - my dad would vet games from CD collections on his computer and then copy them across to ours via a cumbersome cassette-based device called a TapeXchange<sup>1</sup> (also borrowed from the university).

On one occasion, I remember a group of us huddled around the computer looking at the menu of the latest CD we'd acquired - all four of the siblings and my uncle Brian, with my dad at the keyboard. On the list was a new game that was the talk of every computer magazine at that moment, and I just had to insist my dad tried it first.

I had grown up with Apogee Software's shareware as my primary source of computer entertainment, and we had recently discovered and got into playing Blake Stone. But when Doom started up for the first time, what I was seeing was on a level so far from any of those that it was unimaginable - the player's view bobbed as he walked, the rooms were all

---

<sup>1</sup> [https://www.atarimagazines.com/compute/issue153/98\\_Interpreter\\_TapeXcha.php](https://www.atarimagazines.com/compute/issue153/98_Interpreter_TapeXcha.php)

dynamically shaped instead of on a grid, you could even go up and down stairs! We watched as my dad carefully piloted Doomguy up to the armour and back, wondering aloud how to fire the pistol hovering in the middle of the screen. I suggested the Ctrl key because most of the Apogee games defaulted to that, and sure enough, he was able to destroy the stationary target of the barrel with no problem. Emboldened, he opened the door and took on the first enemies of the easiest game mode in the large computer room, then proceeded through to the next part of the base where he instantly fell into the unhealthy-looking green stuff.

I remember Brian leaning over his brother's shoulder and saying "Looks like you're nae daein' too guid, there, Robert" as his character floundered in the nukage, the face at the bottom of the screen gradually deteriorating as the character took damage from the various creatures around him as well as his comical inability to reorient himself on to the path. Finally, mercifully, he succumbed and the player's view dropped to the floor. And that was the end of my first sight of Doom.

My dad decided not to select this game as one to transfer across to our computer, thinking it far too violent for us - but a couple of years later we got hold of Doom 2 when a friend came round with its stack of five black disks. By now we had inherited the 486 as my dad had obtained a shiny new Pentium - at this point the fan was emitting a constant groaning noise and there was something disconnected in the power button that meant it wouldn't start up in cold weather and you had to blow a hair-dryer at it to get it running sometimes, but broadly speaking it still worked. And we played through Doom 2 mostly with IDDQD on and skipping around levels with whatever cheats we could find.

I had always loved making my own levels for games that allowed it, so learning of the existence of Doom editors was tremendously exciting - yet another CD yielded Geoff Allan's DoomEd 4.2<sup>2</sup>. In those days, though, being able to create working sectors was a bit hit or miss and you had to have a pretty good ability to perform calculations in 3D space in your head to work out what your map was going to look like, so I was only able to produce semi-functional abominations. And by this point, id's latest new

---

<sup>2</sup> [https://doomwiki.org/wiki/DoomEd\\_4.2](https://doomwiki.org/wiki/DoomEd_4.2)

amazing shareware game Quake had come out - so as we jerked through that as quickly as the wheezing and panting 486 could manage, we sort of forgot about Doom.

## *Send Lawyers to Hell*

Like most things in my life, I became the operator of a Doom Youtube channel mostly by accident. After rediscovering the Doom editing community going stronger than ever in 2016, I had joined a couple of community projects, put together my own GZDoom episode Vulkan Inc where I threw in every bit of custom content that interested me, and was recording occasional videos by request to help mappers see how a new player navigated their maps and took on the challenges in them.

Things really kicked off when the company I work for was outed in the media for furnishing prison camps on the southern border of the US. My team and I had assumed that the unexpectedly large order we were being asked to jam through the system was for the hurricane relief efforts that were going on at the time, and we had spent a lot of effort making sure that it got safely through - so finding out the true destination was a thorough kick in the face for all of us. Even though I wasn't aware of what we were doing, I felt very tainted to have had any part in the cruelty of the US administration.

To attempt to make up for it, I turned my Doom videos into a charity drive called Send Lawyers to Hell, where I accepted donations in exchange for map requests and forwarded money on to RAICES to help get legal assistance for people (particularly children) crossing the border from Mexico. I was pretty happy with how it went - donations from generous Doom community members totalled over \$2,000 by the time they dried up, and after the charity drive ended I kept going with the videos.

I was coming to realize that I really enjoyed playing WADs from newcomers to the Doom community - not just because they tended to be very manageable compared to the gradually inflating difficulty level that Doom veterans had ridden for the last few decades, but because of how much imagination and variety they had. On top of that, it felt really good to



let newcomers see their maps being played and to give them a boost in visibility. For this reason I had always loved the DUMP trilogy headed by TerminusEst13, and I had been disappointed for a while that there had never been a fourth one - its attitude was very inclusive, with the aim of helping people new to Doom to get over the hurdle of releasing something. I had vaguely wanted to try running a community project of my own for a while, and wanted to keep this spirit going.

Eventually, in May 2021 with my daughter Penny safely back in school after the worst of the Covid pandemic, I found myself with enough free time available to try running a large project myself. After making my preparations, I announced it on the Doomworld forums with a name satisfyingly metaphorical for helping people up - the Rabbit's All-comers Mapping Project, or RAMP.

# RAMP Central

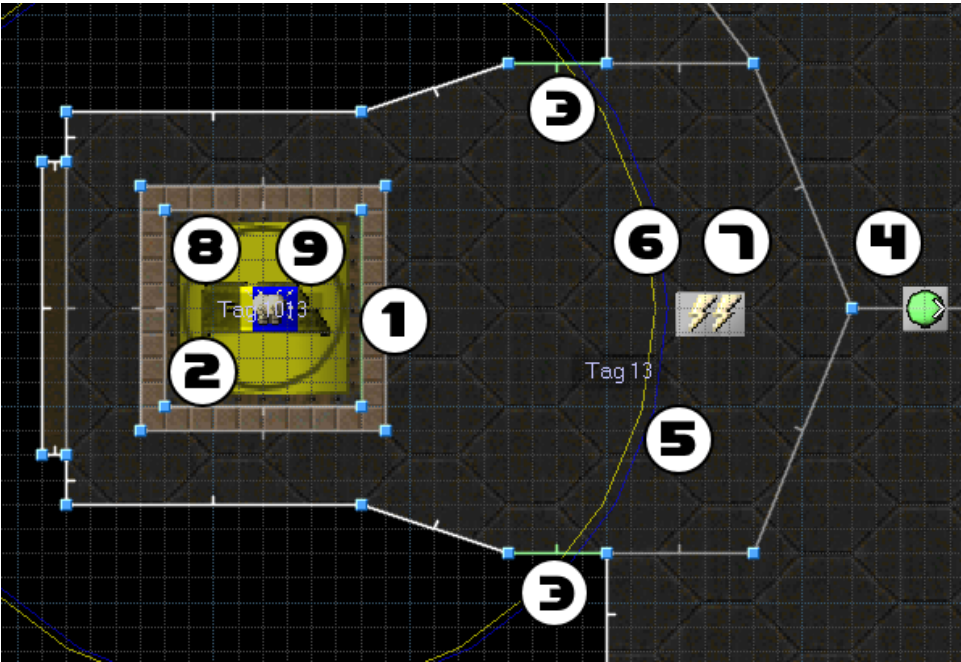
GZDoom packages have a couple of great advantages over vanilla-styled Doom WADs for community projects with unrelated maps. For one, in the original Doom style, the progression of levels is completely locked - you can have exactly 30 normal maps, after which the game ends, and 2 secret levels, the first of which is always accessed from MAP15. The second big plus is that you can arrange the project so that maps can be connected together in definable ways instead of having to be played strictly one after the other. Again taking a cue from DUMP, my plan was to make a single hub level from which any submitted maps could be played.

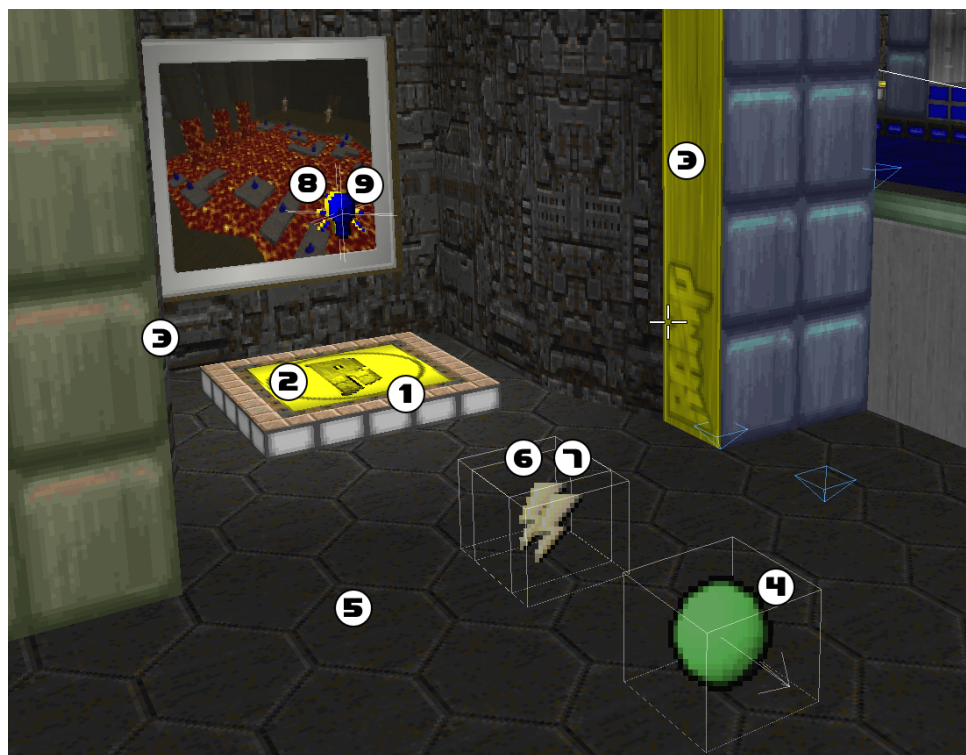


I had thought about arranging the maps into a game where a player would unlock harder levels by completing easier ones, but decided that I didn't want to do anything that would mean that any map was more accessible than any of the others - to fully use the hub format's advantages over the linear one, it was important to me that all of the maps should be open from the beginning. At this point I didn't know exactly what the game that tied all the levels together was going to be, but I went ahead with this basic template in mind hoping that it would emerge as I went on.

### Linking to Levels

I put together a template for a level entrance in Ultimate Doom Builder fairly quickly, and I was fortunate that I liked my first attempt enough not to ever feel compelled to go back and redo the basic shape or components of it at all later. Level entrances consist of a few components, which I gave tag numbers according to a scheme that made it possible to refer to each of them in a script just by knowing the map number.





Part	Description	Tag/parameter
1	The line that transports the player to the map. Calls the script "goToMap" when the player crosses it	(Parameter to script) Map number
2	The sector that represents the teleporter pad	Map number + 1000
3	The lines that border the level entrance alcove	Map number
4	The spot where the player should be placed on returning to the hub from this map	Map number
5	The sector surrounding the map entrance	Map number

6	An “Actor Enters Sector” Thing which calls the script “clueDisplay” when the player enters	(Parameter to script) Map number
7	An “Actor Leaves Sector” Thing which calls the script “clueClear” when the player leaves	(None)
8	A yellow dynamic light	Map number + 1200
9	A blue dynamic light	Map number + 1400

A monitor at the back of the alcove showing a screenshot of the level completes the arrangement, but this isn’t part of the tagging scheme - I just prepared and placed those textures manually.

With this numbering scheme, I could copy and paste the whole arrangement, then use Ultimate Doom Builder’s ability to set tag numbers in a selection relatively (by prefixing the tag field with ++ ) to create new map entrances without too much difficulty. I could possibly have reduced the number of elements I had to change manually with a bit of scripting, but it worked well enough for my purposes.

The arrow-shaped sector surrounding the map entrance contains two sector action Things: one to call a script to display an onscreen message when the player enters it, and the other to clear that message when they leave. They’re called clueDisplay and clueClear because I copied the code from one of my earlier maps called Hell’s Library, where they were used to display puzzle clues when you approached a book or a sign on the wall and so on. Only the Player Enters Sector thing needed to provide the level’s number as a parameter - the Player Leaves Sector thing could just clear the message without needing to know its contents.

The dynamic lights, border walls and teleporter pad floor and ceiling are used to indicate whether a map has been completed or not. I chose to copy the colour scheme from Zelda: Breath of the Wild because I was playing it at the time and found it effective - by changing textures and flats and activating certain lights, maps that haven’t been completed have

eye-catching bright yellow surroundings, and maps the player has visited are in a more sedate blue.

It's also possible for a map entrance to be assigned a map number that doesn't exist, which causes the map entrance to be visibly turned off - but in the finished project I made sure there were exactly as many map entrances as there were maps, so this is never seen.

When the player crosses the line in front of the visible teleporter pad, a script called goToMap is called with the map number as a parameter. This does a few things:

- Sets the global variable "mapNumVisited" to the map number that the player is going to
- Fades the screen to black
- Teleports to the map pointed to by the map number.

The rest of the actions related to entering a map are performed after the player returns to this level once the map they've chosen is complete.

In addition to the Things I placed in the map to represent entrances, I was inclined to add several more over the course of the project, which I could do dynamically with a similar numbering scheme:

Part	Description	Tag/parameter
10	A map marker (only visible on the automap) that displays the level's current status	Map number + 1600
11	A floating Thing that also displays the level's current status	Map number + 1600 (again)
12	A Thing to emit circular teleport effects	Map number + 1800
13	A Thing to represent the map's length	Map number + 2000
14	A Thing to represent the map's difficulty	Map number + 2200



(It's worth mentioning in passing that this is not the original numbering scheme - I had to hastily alter my first one in mid-June in which I had spaced things in increments of 100 instead of 200, not expecting that the project would reach anywhere near one hundred maps.)

## *To hub or not to hub*

In a slightly confusing detail, the hub map is not actually a hub, as far as GZDoom is concerned. It's possible to set up maps in GZDoom in what are called clusters, which follow a true "hub" format like in the Hexen games - the player can freely move back and forth between levels in a cluster, and their state is saved between visits. The DUMP collections (and my own only previous hub-based GZDoom project, Keeper 3) used this layout, taking advantage of the way that the main map's state would be saved between visits but physically preventing the player from re-entering levels by blocking off the level entrances once they had been visited.

I decided to do things differently here - I wanted the player to be able to play maps as many times as they wanted, in anticipation that I would reward them by completing a level in certain ways such as clearing it of monsters. This meant that I couldn't set the maps up in a cluster, because I needed the levels to reset on each visit. I would therefore have to restore the state of the hub map manually when the player entered it, instead of being able to rely on GZDoom to remember its state for me.

As big a deal as it sounds, this didn't really feel like a huge undertaking. GZDoom provides 64 global variables, which can be arrays - and usefully, they even act a bit more like associative arrays than non-global ACS arrays do, with no need to specify a fixed length. I used three at first, with some more joining the lineup over time: an int called mapNumVisited as mentioned above, then an array called levelsDone and another variable called monstersRemaining.

There is an ENTER script on the hub level called comeFromMap that handles restoring the hub's state:

- If the player has just come back from a map, alter the global levelsDone array as appropriate to indicate the level has been visited, then award them some bonuses based on the difficulty and length of the level and whether they exited with no monsters remaining (a global exit script counts this and puts it in the monstersRemaining global just before a player leaves a level)
- Loop through the possible map numbers to decide how each level entrance should be displayed:
  - If a map doesn't exist with this number, disable the entrance's lights and set its textures/flats to look deactivated (using the numbering scheme above to find the relevant lines and objects - if we want to affect the entrance of map 24, we want to alter the floor/ceiling of the sector tagged 1024, deactivate the dynamic lights with tags 1224 and 1424, etc)
  - If this map number's entry in the levelsDone array is greater than 0, the player has completed this map once - remove the yellow light, change its textures to blue and spawn a blue map marker and teleporter effect
    - If the levelsDone entry is 1, the player has completed the level - display a blue token in the teleport area
    - If it's 2, they've also exited the map with no monsters remaining - display a star instead
  - If the entry in levelsDone is still 0, then the map hasn't been visited yet. Remove the blue light, leave the textures as the default yellow, spawn a yellow map marker and teleport effect, and spawn some hovering items that show the difficulty and length of the map.
- Update the progress bar visible on the screen to reflect the number of levels completed and mastered ("mastered" being the term I chose to mean "completed the level with 0 monsters remaining").

Like most things in this project, this script grew in complexity a bit as I added more elements to the game, but this was more or less where the

hub was as the project began. Now I needed a way to put some levels into it.

# RAMPART - The uploader/compiler

## *Setting up*

Bill Gates is often quoted as saying “I choose a lazy person to do a hard job, because a lazy person will find an easy way to do it.” And by that measure I’m an incredibly lazy programmer - I try not to do anything myself if I can possibly get a computer to do the work for me. This means I’ll probably be the first to be eaten when the machines finally rise up against us.

Before I started putting together a project of my own, I knew that I wanted to have some sort of framework in place for compiling it all together without me having to be directly involved in that process. A few years ago I’d seen DUMP 3 drive TerminusEst13 half insane with its vast quantity of maps and assets submitted (and I suspect this is a large part of the reason there was never a DUMP 4). In fact, every forum thread where someone had to keep track of any project with multiple contributors looked like a dreadful tangle, with the project leader hunting for people’s submissions, hastily-made adjustments and new versions, finding if they’d updated a link deep in a topic thread or deleted it and posted a new one, and on top of that, having the Sisyphean task of putting together endless snapshot versions and asking everyone to check that all the changes had been incorporated since the last one.

While I’m online a frankly unhealthy percentage of my life, I knew I didn’t want to go it alone with the task of keeping up with people’s submissions, and that I would require some automated support. I had considered just using Git, but that would involve the giant leap of getting people to install and learn how to use it. As useful as it is as a collaborative tool, Git is absolutely terrible for non-programmers - and let’s be honest, despite having used it for about a decade now I’m only absolutely certain about three commands: “pull”, “push”, and “delete everything and start again”.

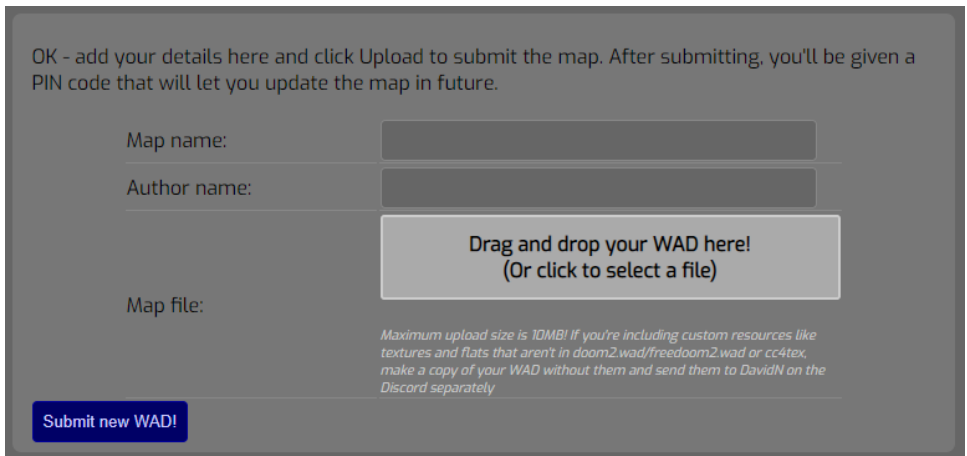
And so I started on the system I would eventually christen RAMPART, a rather tortured acronym for RAMP Aggregator for Rapid Tessellation. The

idea was like continuous integration for Doom WADs - I was going to accept submissions via a site instead of having to keep track of them through forum threads and emails, and would write something that would zip them up in a PK3 file on demand, bundling them with my own material that supported the game built around the maps.

As usual, things turned out a little more complicated than I expected. Then they grew from there and became a lot more complicated instead.

## The first steps

Despite all the frameworks and tools that exist in 2021, I still like to write websites in pretty much bare PHP. It was what I learned to make dynamic sites on a couple of decades ago, and while it has its insanities as laid out by Doom aficionado Eevee in their well-known Fractal of Bad Design article<sup>3</sup> I still can't hammer out the basics faster in anything else. A menu bar, some CSS, some pages and backend scripts - who needs more than that?



OK - add your details here and click Upload to submit the map. After submitting, you'll be given a PIN code that will let you update the map in future.

Map name:

Author name:

Map file: 

Drag and drop your WAD here!  
(Or click to select a file)

Maximum upload size is 10MB! If you're including custom resources like textures and flats that aren't in doom2.wad/freedoom2.wad or cc4tex, make a copy of your WAD without them and send them to DavidN on the Discord separately

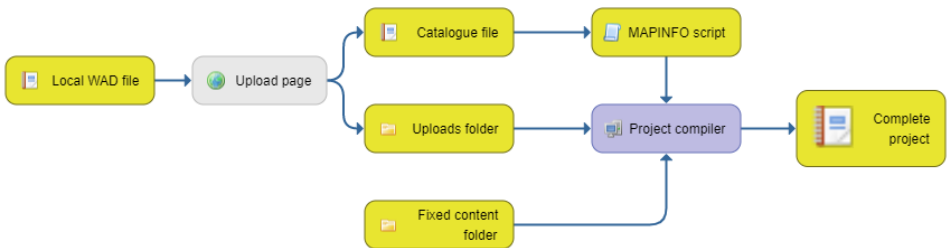
Allowing people to upload things was no problem, but I made a few false starts before deciding exactly how I would implement the method for people to get the files back down again. I had first thought of involving a cronjob or even something like Jenkins to sweep an uploads folder

---

<sup>3</sup> <https://eev.ee/blog/2012/04/09/php-a-fractal-of-bad-design/>

periodically and incorporate them into the project, but eventually decided to keep things as simple as I could and not involve any dependencies. I even went with using a file-based “database” system instead of headaching through PHP’s infamously awful SQL communications. The workflow went like this:

1. When someone uploads a WAD, check for the next available map number.
2. Store the uploaded WAD in an uploads folder under the name MAPXX.WAD.
3. Enter the provided map name and author, together with the assigned map number, into a JSON-encoded catalogue file which acts as a super-lightweight database.
4. When a download of the project is requested, create a ZIP with the contents of my “fixed content” folder plus the contents of the upload folder, incorporate a MAPINFO script to define all the levels that had been uploaded, and serve it as RAMP-SNAPSHOT.pk3.



Because I knew creating a ZIP wouldn’t be an instant process, I made the download script check if the catalogue file had been updated since the last PK3 generation, and had it just serve that same PK3 back again if it hadn’t - therefore saving a lot of waiting around for downloads if a snapshot had already been generated with the files that were currently in the uploads folder.

I was also aware that people were almost certain to want to update their maps after submission, and I didn’t want them to have to go through the trouble of setting up some sort of account to do it. So when someone






















uploaded a map for the first time, the response message would include a PIN that they could write down (or, in real-world use, ask me to retrieve it for them because they'd forgotten). This could later be entered into the upload form to call up the details they had entered before and provide an updated WAD or amended information for the map.

At first, I provided a field for people to enter their own PIN during a map's first upload. I chose the term "PIN" over "password" to discourage people from reusing a bank password or anything that was meant to actually be secure - while I know about hashing and salting and the various other ways to cook a password, I didn't want the remotest chance of a mistake by me accidentally handing out people's secure information. Eventually I decided not to accept user input at all for this, and just to hand out PINs to the contributors from a predetermined list to make it clear this wasn't meant to be a secure system.

## GZDoom and WADs

According to Tom Hall, the Doom data file format got its name when, having decided to use the word "lump" for a piece of game data, John Carmack swivelled his chair around to him and asked "Hey, what's a word for a lump full of lumps?". "A wad?" was the unsure answer, and so WAD files were born, later backronymed to "Where's All the Data?". These WADs are like lists of files - each lump has a name of up to eight letters and some associated data. Unlike files, the order of lumps in a WAD matters, and they're interpreted based on where they are in the list.

A Doom-format map, for example, isn't a single entry but is made up of eleven lumps. The first is a lump with no data behind it called a "marker", which has a name matching the internal name of the level slot (such as E1M1 or MAP01). This is followed by ten further lumps that describe various aspects of the map such as the locations of vertices and objects.

Entries		
Name	Size	Type
 PLAYPAL	10.50kb	Palette
 COLORMAP	8.50kb	Colormap
 ENDOOM	3.91kb	ANSI Text
 DEMO1	16.84kb	Demo
 DEMO2	7.83kb	Demo
 DEMO3	17.48kb	Demo
 MAP01	0	Map Marker
 THINGS	690	Map Things
 LINEDEFS	5.06kb	Map Lines
 SIDEDEFS	15.50kb	Map Sides
 VERTEXES	1.50kb	Map Vertices
 SEGS	7.04kb	Map Segments
 SSECTORS	776	Map Subsectors
 NODES	5.28kb	Map Nodes
 SECTORS	1.50kb	Map Sectors
 REJECT	436	Map Reject Table
 BLOCKMAP	6.27kb	Map Blockmap
 MAP02	0	Map Marker
 THINGS	1.69kb	Map Things

In addition to WADs, GZDoom supports the newer PK3 format for Doom mods. A PK3 is really just a ZIP file with a sheet over it - it's much easier to work with quickly than vanilla WADs are, as files are interpreted based on where they are in a folder structure instead of whether they're between a certain set of markers or not. Textures and flats go into textures/ and flats/ folders and will be interpreted even without converting them to Doom's specific image formats. The maps/ folder is the only place where WADs come into the equation - for each WAD file in this folder, GZDoom will include the map described by the WAD, using its filename (and not the name of the map marker within the WAD) as the internal name for the map.

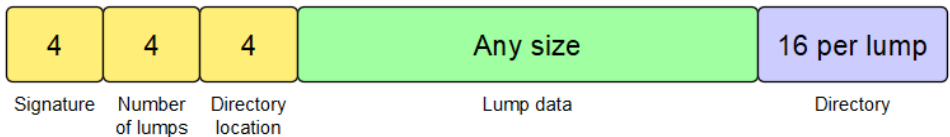
So, knowing about this arrangement, I could create compiled PK3 projects very easily without even manipulating the uploaded WADs - when a download was requested, I could copy the files from my fixed

content folder into a temporary work folder, copy the uploaded WADs into a maps/ folder within that using the filenames to arrange them into their level slots, then ZIP the whole thing to produce a PK3 that was ready to run in GZDoom. I implemented this, threw in some WADs I happened to have lying around, and I was immensely pleased that it worked instantly.

Except... for some reason it didn't work for every level. About half the WADs that I had included just refused to be recognized, with GZDoom giving a "No map MAP13" error when I tried to jump to them. Eventually, by scouring through the WADs in SLADE, I saw the pattern - the levels that worked were in WADs that either just contained the map lumps and nothing else, or that happened to have placed their music and other assets *after* the map data in the list of lumps. Maps in WADs that had any data before the initial map marker were just being discarded. It looked like I was going to have to explore the WAD format more deeply after all.

## Exploring Lumps

I hadn't looked much at the internals of WADs before starting RAMP, but the file format is well documented around the community including Doomwiki<sup>4</sup> - so even though I hadn't done a lot of work with files on the byte level before, they turned out fairly easy to parse. A WAD file is divided into three sections:



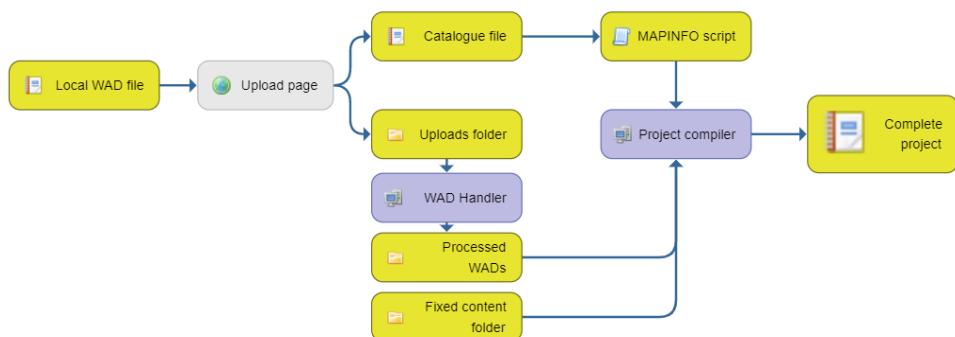
1. A header that describes what the WAD is and how many lumps are going to be in it
  - a. 4 bytes for a signature - this is either the string IWAD or PWAD, to identify that this either an internal or patch WAD
  - b. 4 bytes for the number of lumps in the WAD.
  - c. 4 bytes for the byte number where the directory starts - i.e. the first byte after the main section

---

<sup>4</sup> <https://doomwiki.org/wiki/WAD>

2. The lump data: appropriately, this is a large wad of continuous data with no separation between data for the individual lumps
3. The directory, which describes how to parse the lumps back out of the WAD. For each lump, it stores 16 bytes:
  - a. 4 bytes for the location in the lump data where this particular lump's data starts
  - b. 4 bytes for the size of the data for this lump
  - c. 8 bytes for the name of the lump (8 characters long, padded with null bytes if the actual name is shorter than 8 characters)

So, in what was retrospectively a sign that the project was beginning to snowball, I added a WAD handler to the equation. Now, instead of being copied directly into the project during the PK3 compilation step, each uploaded WAD file would be run through this code. The handler would interpret each file, go through the lumps and discard anything that wasn't part of a map, then save the filtered WAD into the project ready for compiling into the ZIP. The original files would remain intact in the uploads folder, so they could be reprocessed at any time without having to ask the contributor to re-upload their file.



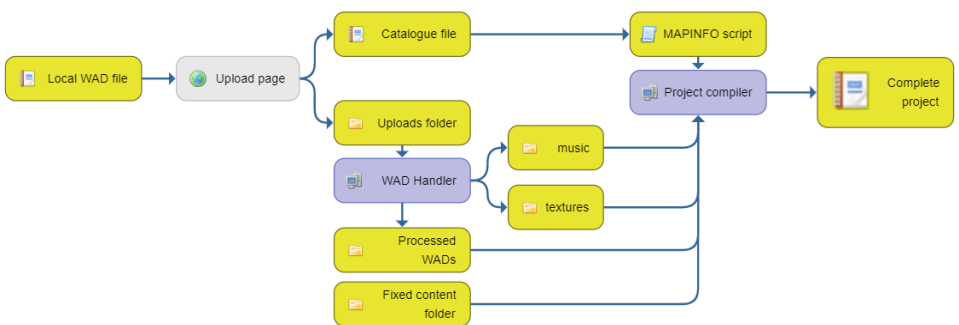
I ran my random test subject files through the new process, and the PK3 was produced with the doctored WAD files. Now, even the maps that originally had extra data that interfered with the reading of the map were recognized and ran without problems. Everything was working.

## Allowing Limited Customization

It was now a bit restrictive, though. GZDoom takes the already fairly customizable Doom and amplifies it massively with a ton of ways to extend and fiddle with it, and my upload process only allowed map data through. During the phase where I'd experimented by uploading a load of WADs I happened to have lying around, I had hoped that GZDoom would be able to recognize and include music and graphics lumps inside the map WADs as well, but as it turned out, it was only able to read the map data in them.

This wasn't a huge issue, though, because now I had a way of opening a WAD up and poking around the insides, it would be simple to give the PK3 updater the ability to recognize a couple more kinds of lumps. So I added two extra cases:

1. Assume the first MIDI file in the WAD is the map's music (later this was expanded to allow MP3, OGG and even SNES SPC files, which I wasn't previously aware GZDoom could play). When it's encountered, copy its data out into the PK3 music folder under the name MUSXX where XX is the map number
2. On finding a graphic called RSKY1, use this as the map's sky texture - similar to the above, copy it into the PK3 textures folder with the name MSKYXX.



After getting the WADs together, the PK3 updater would then write a MAPINFO file (GZDoom's way of describing game and map properties) which pointed to the appropriate music and sky for each map if those files

had been parsed out of the uploaded WAD, and the default Doom 2 RSKY1 and D\_RUNNIN sky and music if they hadn't.

Upload a WAD file containing a single map in any format (Doom, Hexen, UDMF). You can amend a submission at any time by using the PIN given to you on first upload.

As well as a map, you can include these resources:

- The first MIDI/OGG/MP3 file in the WAD will be used as your map's music, no matter what its name is.
- For skies, you can do one of two things:
  - Include a lump called RSKY1 and it'll automatically be used as the sky

I decided that at this point that this was about all that the uploader needed - it would work for most people's cases and I could work with people to include the occasional thing that would need manual intervention. As this project was primarily aimed at people who were taking their first steps into DOOM modding, I expected that not many people would want to include custom content.

## *Extending RAMPART*

Everyone wanted to include custom content. I should really have anticipated this, because when I got into GZDoom in 2016 and discovered its vast potential for adding and adjusting things, I went absolutely mad throwing in everything that interested me and replaced about half the game by the time Vulkan Inc was finished. In addition to the custom content, people were a bit more adept at using MAPINFO than I had thought, and the instructions on the upload page that I thought had been straightforward actually went against people's expectations on how they should submit their maps.

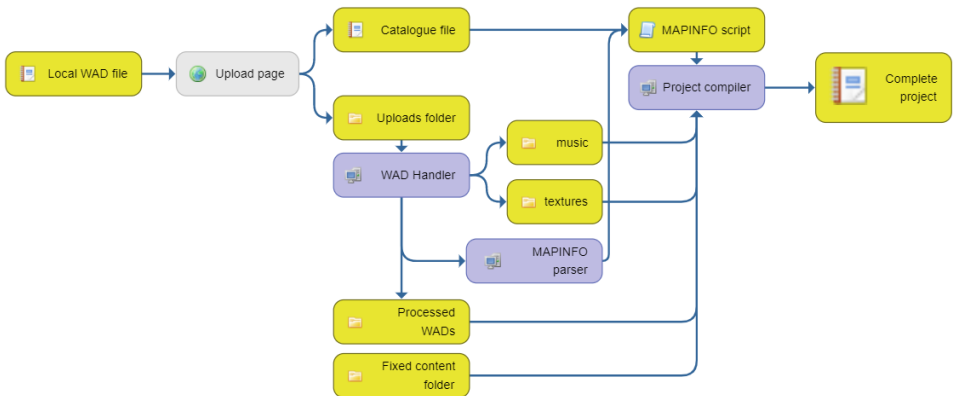
### Custom metadata

One of the most obvious things I'd forgotten to include was an option to allow jumping and crouching in a map. It wasn't possible to just set this at a project level, because many people would design their levels in a way that required jumping to progress, while many others would assume the player wasn't allowed to jump, enabling massive skips if jumping were to be enabled.



So I went back and added a toggle for jumping and crouching to the upload form - now, the catalog file included a flag for whether jumping would be allowed on the map, and the compile step would recognize that flag and put the appropriate flags into the MAPINFO for that map. While I was at it, I added a second flag so that people could indicate that their map was a work in progress - ready for people to try out and give feedback, but perhaps not fully completable. Unlike the jump property, this one had no actual effect on the MAPINFO - it was just an indicator for the site.

As more maps were submitted, though, there were people who included MAPINFO lumps in their own WAD, which my compiler would ignore as I hadn't told it to do anything with them. The NoJump and AllowJump flags weren't a problem, because there was already a toggle for that in the upload form - but there were all sorts of things that I hadn't considered would be incredibly important for maps, such as flagging that killing a Cyberdemon or Spider Mastermind should do something in the level, or that the level should behave like Dead Simple with its two hardcoded tags.



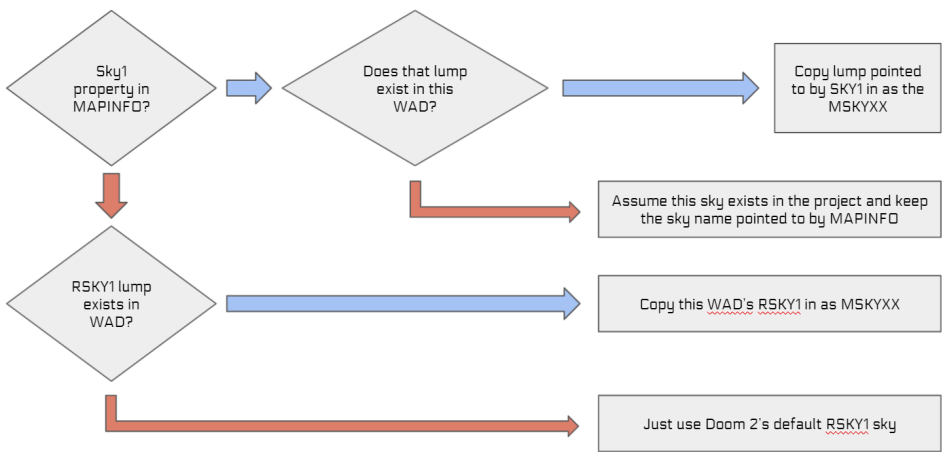
This meant adding another component to the compiling step, a MAPINFO handler. On finding a MAPINFO lump in an uploaded WAD, the WAD handler would call this and it would use the very crude parser I'd written to loop through its contents and pay attention to some of the properties depending on an allowlist. Then, when it reached the step for writing the

MAPINFO for the map into the completed project, it would include the properties that it had derived from the submitted MAPINFO along with its own details.

## Catering to everyone

Skies and music presented further problems. I think complete beginners appreciated my special cases for how to include them in their maps, but people familiar with MAPINFO skipped straight past reading any of the instructions. Because why would they have needed them? They already knew how to include music and sky graphics and how to tie them into the map. I had now inadvertently set up a challenge for myself in allowing both methods to work.

The problem, as in many things with computers, was in correctly working out what the submitter intended to happen, no matter what combination of things they uploaded. If they had a lump called RSKY1, they had probably followed the instructions on the upload page and wanted that as the sky. If there was a MAPINFO with a sky1 property that pointed to a different lump name, then they probably wanted the lump they'd uploaded by that name to be the sky. But it was also possible for them to want to use a lump in the base project or Doom 2 resources as the sky - so the third possibility is that they would provide a sky1 property that pointed to a lump name that didn't exist in the uploaded WAD. In these cases, I just had the compiler assume that a lump by that name would somehow exist in the project when it was run.



Despite all of this automation I was building, new unexpected cases kept on coming - maps which needed more than one music included because they switched in the middle, or two skies which were used in different places. These felt specialist enough to just deal with manually by uploading the resources people wanted to use myself.

In addition to all of that, throughout the project I made sure that new textures, sprites and sound effects had to be manually copied to the PK3 fixed content folder instead of included automatically. This was a deliberate decision so that I could review them before inclusion, as I felt these were the easiest way for people to include inappropriate content in the project. It also meant that I could keep track of anything that was attempting to override a default lump - there was an incident midway through the project where all red hell pillars sprouted into giant trees because of someone's choice of sprite names.

## Raising limits

The sheer pace of submissions was beyond anything I'd expected, with an entire megawad's worth of maps being submitted within a week of me opening the site in what was intended to be a slow ramping-up period. And in addition to continuously providing new ways to upload things, as the project grew it ran into limits that I hadn't expected it to hit. I had to raise the PHP file upload and POST data limits (upload\_max\_filesize and

post\_max\_size in php.ini) from the initially adequate-seeming 2MB several times, eventually hitting 15MB because it's possible for files in UDMF format to hit very large figures just from having a lot of vertex and line data.

Another mystery arose early on when the project rolled over from MAP19 to MAP20, and the map wasn't getting added to the project when it was regenerated and updated. I had assumed that there was something subtle wrong in my code that caused maps above 20 to break, but the real situation was one of those bugs where you really need to put on your detective hat with the silly ear flaps:

- On requesting a download, the project would start the handle\_pk3\_update script, recognize that it needed to rebuild the project and start work on it.
- During this script's running time (usually during the phase when it had prepared everything and was making a ZIP of the compiled project), it would run over the PHP script timeout and terminate.
- Because of how PHP handles writing ZIPs by creating a randomly-named temporary target file before copying it to the real one, this meant the temporary file was left behind when the script terminated and the compiled project from the last successful run remained unchanged.
- Therefore, people were getting the "Generating a PK3..." message and eventually getting a download, but the download was the last one that was made before MAP20 was added, and not the one that had just apparently been generated.

This was circumvented fairly easily by looking at PHP's vast array of levers and pulleys again and finding how to specify a custom timeout value for a script - I set it to ten minutes to be safe, but the real amount of time to compile the final project was more like five minutes in the end. This issue also needed some more investigation early on because it turns out that PHP's handling of ZIP files is unusually slow - what I ended up doing was not touching the ZIP process in PHP at all, but having it just prepare the files to be included in a folder then call a Python script to perform the actual zipping process (then I realized there wasn't much point to the Python script either and just had it call Unix's own command

line zip utility instead). For reasons not entirely clear to me, this is much faster than having PHP write the ZIP itself.

In a similar vein, the download page briefly stopped working when the size of the compiled PK3 drifted over 64MB - the naive way that I had sent the file to the client's browser (loading the entire file then sending that) caused PHP to run out of memory, and I had to quickly learn the proper way of doing it by using the `readfile` function (which, in the best of PHP traditions, is not used to read a file but instead writes the contents of a file to the output buffer).

Once the project got ridiculously large, I also found out that GZDoom doesn't automatically assign `LevelNum` properties to maps with lump names above MAP99 (it only does so for lump names in the format MAPXX, and maps over 100 would be MAPXXX instead). `LevelNums` are needed to tell the game where to teleport the player when you use the "teleport to new level" linedef action, so the hub teleporters stopped working for maps which were assigned slots above MAP99. This was a simple fix to make the MAPINFO writer just add the property according to the map slot number, but it was yet another example of having to spend some energy on a problem that arose because of the unexpected size of the project!

## *Downloading RAMPART*

After RAMP ended, I extended RAMPART a little to handle a greater variety of types of Doom projects, and it's free for anyone to use - all that's needed is a web server with PHP. The source code is available at <https://github.com/davidxn/rampart>.

# Rapidly Annihilating My Predictions

At the beginning of the project I had made sixteen map slots available, a number which had turned out to be comically inadequate very quickly. Therefore a lot of my role throughout the project was in bolting on new rooms in the hub map to accommodate new transporters.

The DUMP projects had gone with fairly organized, regular and symmetrical hub maps, but RAMP's was meant to be more chaotic from the start, looking like one of Doom's own vague techbases with all sorts of components and environments. As a result, after twenty or so maps had been submitted it began to get very hard to navigate to the map you intended to play. As the hub expanded I tried to give each area its own sort of identity, colour scheme and landmark to aid navigation, but it didn't really solve the problem of not being able to track down where a particular map was if you didn't know.

Some sort of a guide was needed, and the problem kept getting more evident as new maps continued to flood in. Somebody in the project's Discord server generously provided an annotated version of the map like you get in shopping centres, with the maps listed alphabetically and tied to their map numbers. It was very useful to have, but wouldn't fit very well in the game without a lot of thinking about how to display and interact with it.

What I needed was a simple interface - a way for the player to see a sorted list of all the maps so that they could find the one they were searching for, then to somehow indicate to them where the selected map was. I vaguely knew that GZDoom supported the Strife dialogue system, which was a way of setting up a branching tree of conversations, and this seemed to be a promising way to let the player flip through a menu without having to code the entire system myself.

## *The Guide*

I don't think it's unfair to call the dialogue system a bit awkward. Like many things in the GZDoom world, it comes in multiple generations - first



there was the binary format that was made up for the original Strife game, then it was refined into a more malleable text-based format called Universal Strife Dialogue Format or USDF<sup>5</sup>, and then built upon further by GZDoom with its own extensions and termed ZSDF<sup>6</sup>. And even though ZSDF includes some of the familiar GZDoom concepts like interacting with inventory items and so on, what you can do with it is much more limited than what you might expect if you're coming from ACS.

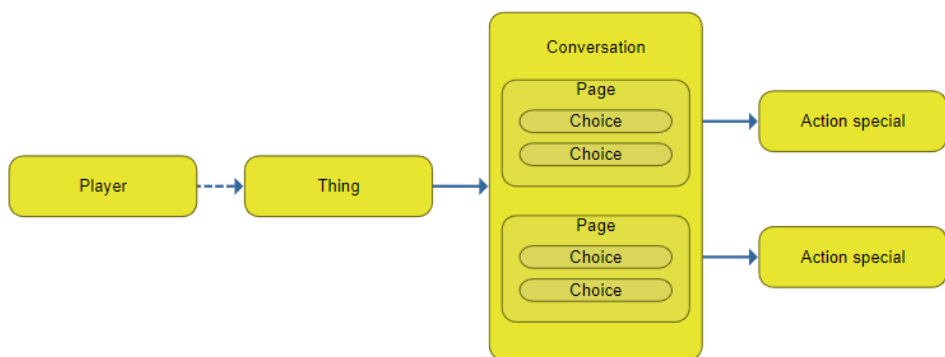
Here's what I was able to work out:

- DIALOGUE lumps are tied to maps, and are not global to the WAD (which didn't matter, because I only needed this to run in one level).
- Conversations are assigned to Thing types (or actor classes, in GZDoom-speak).
- When a conversation is initiated, the player has to talk to a Thing. It doesn't matter how the conversation is started - even if it's through a script that's run by a linedef, the action performed is "start a conversation between the player and this tagged Thing". The conversation that's started is the one assigned to that Thing's type.
- Conversations consist of a list of Pages, which the player can navigate between according to the Choices that are made available on each page. To some extent, you can include conditions that decide whether a Choice is visible or not.
- A Choice can optionally perform an action special (such as action special 80 to run a script) when selected. If no next page is specified, it will also end the conversation.

---

<sup>5</sup> [https://zdoom.org/wiki/Universal\\_Strife\\_Dialog\\_Format](https://zdoom.org/wiki/Universal_Strife_Dialog_Format)

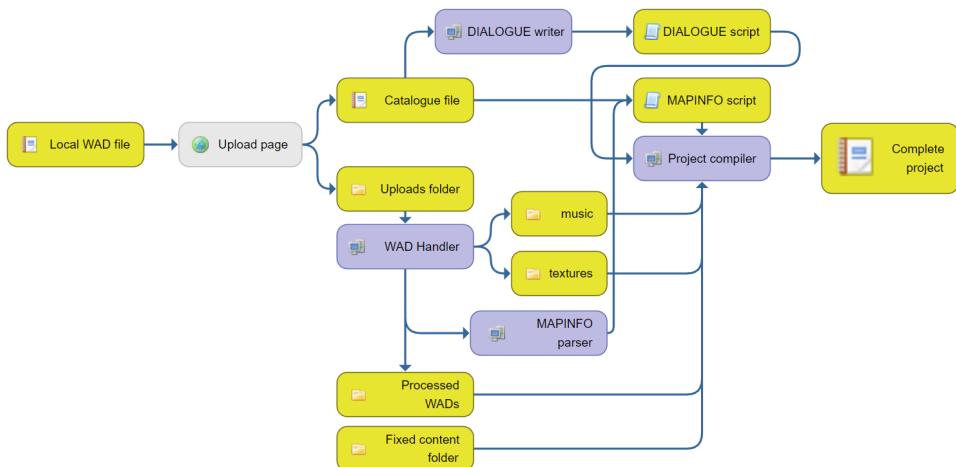
<sup>6</sup> [https://zdoom.org/wiki/ZDoom\\_Strife\\_Dialog\\_Format](https://zdoom.org/wiki/ZDoom_Strife_Dialog_Format)



So the framework was just about visible now - I had to generate a list of Choices that represented each map, which would call a script with the map number as a parameter when selected. From there, I could hand control over to ACS and do what I needed to do to guide the player to their selected map.

However, as I mentioned, the Dialogue format is limited, particularly if you're accustomed to GZDoom's niceties. The biggest letdown was that there was no automatic paging - if I threw a list of all 60 or so maps at the time into a dialogue page, it would just give up on displaying the choices (I found 10 to be the practical limit per page before things started getting strange). In addition, you can't specify any sort of page identifiers - they're assigned numerical IDs depending on their position in the file, and unlike everything else, are indexed from 1. So I would have to keep track of a changing list of maps across multiple pages, and because new maps might be inserted anywhere in the alphabetical list, it would never be certain which page number a map would appear on. Naturally, I didn't want to have to update the list of maps manually every time someone submitted a new one, either. So this would require - what else? - another script which would output the dialogue script for me!

The guide-writing script started life as one of the admin tools on the RAMP site that ran every time I felt the guide needed updating, but before too long I'd made it part of the compilation process. Its purpose is to produce a dialogue script readable by GZDoom that can be added to the hub map data while preparing the complete download.



The script reads the project catalogue file to get a list of all current map numbers and their names, sorts the list according to the map name (ignoring case) and then organizes them into a list of groups of a configurable size. These groups would become the list of Choices on each page.

Then it's just a matter of writing out the script:

- Write the conversation information (just the name of the actor class that will run this conversation)
- For each page:
  - Write the name and dialogue to display, and the text of the option which will end the conversation (each page of the conversation has this added automatically, you can't have a page where the player is forced to make a choice and not quit the conversation). In our case these are all the same no matter the page, but they still have to be repeated on each one.
  - Write a "Previous page" choice that leads to the page before this one (or just a "Top of list" choice that leads back to here if this is the first page)
  - For each map in this page group:
    - If the name of the map is too long for display here, truncate it

- Write a choice with the text of the map, which will cause a script to be run with the map number as an argument when selected
- Write a “Next page” choice that leads to the page after this one (or if this is the last page, just “End of list” which leads back here)
- Add 1 to our current page number and loop until all our map groups have been handled.

Initially, I made the map choices move the dialogue to a final page announcing that a marker had been added to the map, but in the end I found this got in the way a bit and opted just to get the script that started the conversation to log a message to the screen after a map choice was made instead.

This is the start of the conversation script followed by the first page. The TOP OF LIST choice just leads back to this same page 1 - on subsequent pages this is replaced with an option that allows the player to return to the previous page, and the TOP choice exists here so that the position of the first map on the page remains consistently at position 2 throughout all pages. The choices with the map names perform action 80 which means “call a script”, and have parameters that tell them to call script 1 on the current map with an argument matching the map number for the requested map. The “next page” choice, comfortably obviously, advances to the page after this one (if this is the last page, the guide writer replaces this with an “END OF LIST” choice instead).

```
conversation {
  actor = "RAMPO";
  page { // page 1
    name = "RAMPO";
    dialog = "Hello! I'm RAMPO (which stands for RAMP Assistant for
Map Pointing-Out). Which map can I help you navigate to today?";
    goodbye = "Close (ESC)";
    choice { text = "-- TOP OF LIST --"; nextpage = 1; }
    choice { text = " 12 Gauge Catharsis"; special = 80; arg0 = 1;
arg1 = 0; arg2 = 106; }
    choice { text = "  A Baleful Heart"; special = 80; arg0 = 1; arg1
= 0; arg2 = 88; }
    choice { text = "  Abandoned"; special = 80; arg0 = 1; arg1 = 0;
```

```
arg2 = 66; }  
    choice { text = "  Ad Viridis Andron"; special = 80; arg0 = 1;  
arg1 = 0; arg2 = 89; }  
    choice { text = "  Airborne"; special = 80; arg0 = 1; arg1 = 0;  
arg2 = 93; }  
    choice { text = "  Ambition's Cost"; special = 80; arg0 = 1; arg1  
= 0; arg2 = 135; }  
    choice { text = "  And They All Fall Down"; special = 80; arg0 =  
1; arg1 = 0; arg2 = 59; }  
    choice { text = "  Arterial Space"; special = 80; arg0 = 1; arg1 =  
0; arg2 = 90; }  
    choice { text = ">> NEXT >>"; nextpage = 2; }
```

When I decided to make this an automatic part of the compilation process, a bit more trickery was needed. Previously, the hub map had been part of the fixed content folder which was just copied entirely into the PK3 folder without changes, but messing with its data would require an extra step. Therefore the hub WAD wasn't really "fixed content" any more, but I chose to cheat just a little and keep it as part of the copying process then mess with the copied file before finalizing the zip. It felt better than having an entire separate workflow for handling the WAD for the hub.

So another step was added to the compilation process. After performing the copy of fixed content, the code would open up the copied hub WAD and copy all the lumps into a new WAD file. After encountering the BEHAVIOR lump (which represents the compiled script code), it would insert the generated DIALOGUE, then save the doctored WAD over the original one. Later on I added further dialogue to the hub, and so this became appending the text of the generated dialogue on to the existing one, rather than inserting a completely new lump.

After absolutely ruining the WAD a couple of times because I forgot to update the reported size of the lump when I altered it and sent the pointers to all the lumps in the WAD directory off into neverland, my meddling worked very well. I created an out of bounds Thing to hold the conversation, constructed a little computer terminal with my ZZT player avatar from my Stumbling video series to give it some character, and wrote an ACS script to create a marker on the map depending on the map number that had come into it (in another mildly annoying DIALOGUE

limitation, you can't specify a script name - it has to be numbered instead, but this wasn't a huge deal).



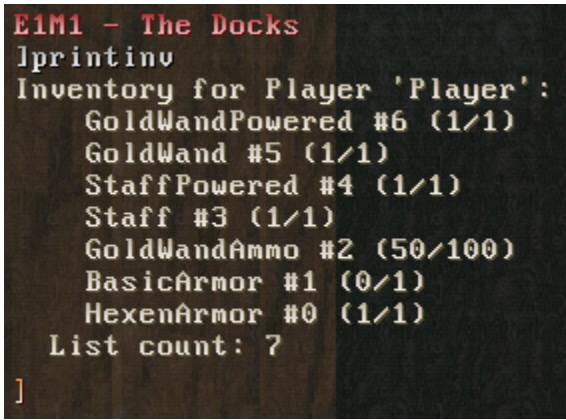
## Displaying progress

There was one last detail that nagged at me, though - finding the location of a specific map in the hub if you knew its name was something that contributors and testers for the project would be interested in, but if you were a player exploring the hub, you would probably be using this to look for an interesting-sounding level that you hadn't yet visited. To make it useful for that, there needed to be some indication of whether a level had been finished previously, or completely cleared of monsters.

Here, I ran into the fairly major obstacle that the DIALOGUE system can't change the text or appearance of choices at all at runtime - what you put in the script is exactly what you get. However, it does have a couple of ways to attach conditions to whether choices should appear - a limited system that can look at the player's inventory and hide options if they're carrying (or not carrying) specified objects.

The player's inventory in ZDoom can be a bit of a strange place if you're not used to its way of thinking. In the original Doom-engine games that have an inventory, such as Heretic, it's a place where you can select and

use special items that you've picked up. The concepts of carried weapons, ammunition and keys are separate - only special usable items go into this inventory. ZDoom combines all of these ideas together, representing everything that's attached to the player as some kind of Inventory item - this could mean things in what you'd traditionally think of as a selectable inventory, items that the player is "holding" in the context of the game like their keys and weapons, or even abstract things that are more like properties that are attached to the player.

A screenshot of a ZDoom console window showing the player's inventory at the start of the level 'E1M1 - The Docks'. The text is as follows:

```
E1M1 - The Docks
lprintinv
Inventory for Player 'Player':
  GoldWandPowered #6 (1/1)
  GoldWand #5 (1/1)
  StaffPowered #4 (1/1)
  Staff #3 (1/1)
  GoldWandAmmo #2 (50/100)
  BasicArmor #1 (0/1)
  HexenArmor #0 (1/1)
List count: 7
]
```

This is the player's inventory on starting Heretic within ZDoom - the player has four weapons (two each to represent the Tome of Power-enhanced and normal varieties of the wand and staff), their ammunition, and an object that's used internally to represent what kind of armour the player is wearing. HexenArmor is obviously only used by default in Hexen, where armour value is represented by some brain-melting algorithm that involves four different armour items<sup>7</sup> - but the ZDoom player is always carrying it because it's needed to represent some variables.

The point of explaining all this is that the player can be given "inventory" items that just represent values or flags that you want to associate with the player. I call these "StatInventory" items, and they're a habit that sort of infected me during my exploration and documentation of Brutal Doom a few years before this<sup>8</sup> - in the course of this project I realized there was a

---

<sup>7</sup> <https://zdoom.org/wiki/Classes:HexenArmor>

<sup>8</sup> <https://forum.zdoom.org/viewtopic.php?t=62203> BDLite

better way to handle variables on the player, but the idea is to define a generic inventory item with a high maximum that does nothing except sit in the inventory, and then specific ones that extend it without modifying anything.

```
Actor StatInventory : Inventory
{
    inventory.maxamount 999999
    inventory.interhubamount 999999
}

// There's got to be a better way to do this, but here we go anyway
Actor RampLevelComplete10 : StatInventory {}
Actor RampLevelComplete11 : StatInventory {}
Actor RampLevelComplete12 : StatInventory {}
Actor RampLevelComplete13 : StatInventory {}
Actor RampLevelComplete14 : StatInventory {}
...
Actor RampLevelMastered10 : StatInventory {}
Actor RampLevelMastered11 : StatInventory {}
Actor RampLevelMastered12 : StatInventory {}
Actor RampLevelMastered13 : StatInventory {}
Actor RampLevelMastered14 : StatInventory {}
...
```

No doubt you can already see that this approach is rather inelegant - there's no such thing as an array in the player's inventory, so you have to create a huge list of numbered classes to sort of simulate elements in an array instead. There were two hundred of each of these in the project's DECORATE file, significantly bloating its size (but not affecting the game performance - just defining classes has little effect on this).

The point of these two big lists of classes was to have a representation of the levels that the player had visited within their inventory. In the script that was run when the hub map started when it had to loop through each level number to decide how to display the portals, I added a new action - if the player had visited and completed a map, give them the RampLevelComplete StatInventory item with the number matching that map number. If the player had also cleared out the monsters, give them the RampLevelMastered item as well.



And now that the player's inventory was set up, I could alter the way the conversation script was written. For each map, I needed three separate choices:

- Choice 1, the map name alone. Visible if the player does not have the RampLevelCompleteXXX item appropriate to this level in their inventory.
- Choice 2, the map name prepended with an X (to represent it being crossed off the list). Visible if the player has a RampLevelCompleteXXX item but does not have the corresponding RampLevelMasteredXXX.
- Choice 3, the map name prepended with a \* (which appears as a diamond in Doom's default small font). Visible if the player has a RampLevelMasteredXXX item.

Setting up three choices and all these conditions per level would usually be a very dull task - and as you can imagine, it made the script very large - but with the guide writer, it became very easy to write a loop and make this part of the process. Therefore, this is what the start of page 1 of the conversation script looked like, with the three different options for a single level - only one of which will be displayed depending on the player having the RampLevelCompleteXXX or RampLevelMasteredXXX items in their inventory.

```
conversation {
  actor = "RAMPO";
  page { // page 1
    name = "RAMPO";
    dialog = "Hello! I'm RAMPO (which stands for RAMP Assistant for
Map Pointing-Out). Which map can I help you navigate to today?";
    goodbye = "Close (ESC)";
    choice { text = "-- TOP OF LIST --"; nextpage = 1; }
    choice { text = " 12 Gauge Catharsis"; special = 80; arg0 = 1;
arg1 = 0; arg2 = 106; exclude { Item = "RampLevelComplete106"; Amount
= 1; } }
    choice { text = "X 12 Gauge Catharsis"; special = 80; arg0 = 1;
arg1 = 0; arg2 = 106; require { Item = "RampLevelComplete106"; Amount
= 1; } exclude { Item = "RampLevelMastered106"; Amount = 1; } }
```

```
choice { text = "* 12 Gauge Catharsis"; special = 80; arg0 = 1;
arg1 = 0; arg2 = 106; require { Item = "RampLevelMastered106"; Amount
= 1; } }
```

At least, that's what it looked like until I wrote all of that description of how I did it above during the time when RAMP was in public testing. In this concluding paragraph I was going to say how absolutely overwhelming it looked to do this in ZScript just so that I could justify to myself that this solution, while clumsy, was actually a clever workaround for the limitations of ACS and DECORATE. Then I noticed that ZSDF allowed the possibility of using a custom ZScript class to lay out the data in the dialogue tree, and when I looked at the code of the ConversationMenu class that is used by default, I could just about understand what it was doing and how to adjust it.

So now I was in the awkward situation of having realized how I might make a more satisfying, less workaroundish version of the guide - while definitely not straightforward, it was at least within the realms of human possibility. So on a whim, I went off on my first proper exploration of ZScript.

## Displaying progress, but this time using ZScript

DECORATE and ACS are languages with all sorts of madness in them - one of my favourite details about ACS is that all of its variable types are smoke and mirrors, and everything is an integer underneath. A decimal number? It's an integer with the top 16 bytes representing the whole numbers and the bottom 16 representing how many 65536ths in the decimal part. A boolean? It's just an integer, you can assign the number 7 to it if you like. Surely a string can't be an int? But it is - when you create a string, ACS actually bundles the contents of the string away in a strings table, and gives back an integer that points to that entry in the table - every function that accepts a string really accepts an integer and silently looks up the entry in the table to get the real contents of the string. It's a mess that's had bits attached to it with elastic bands and toothpaste, and I am absolutely fascinated by it.

But what makes the ZDoom scripting languages so nice to work with is the extensive wiki and the history of questions and answers that have built up on the forum - if you don't know how to do something, asking Google will very likely land you on one of these places or the other with details of functions, parameters and examples. ZScript, being relatively new, just can't compete with this - there are guides to converting a DECORATE class to ZScript, but it's hard to grasp exactly what the further possibilities are, and the details on the wiki seem to assume that you already know about the internals of the ZDoom engine and the best place to extend its functionality to achieve what you want.

However, looking at the ConversationMenu class<sup>9</sup>, I felt I had a fairly self-contained testing ground for the first time, as well as a grasp on what I wanted to change. My goal was to alter the code of the dialogue options so that I could influence how they were displayed, and to work out how to transmit level statuses from my existing ACS code into a place that ZScript could use.

### Altering the conversation display

My first experiments were very encouraging - I could specify in the conversation script that it should use my own RampGuideMenu to display itself, and then (in a concept that will be familiar to DECORATE users as well) extended the ConversationMenu class so that I had a place to build on.

The methods in ConversationMenu are rather long and unwieldy, performing many actions each which makes them hard to demonstrate with (and awkward to extend because you have to copy and paste the whole thing) - however, just one call in DrawReplies is responsible for how the text of a choice is displayed:

```
screen.DrawText(  
    displayFont,  
    Font.CR_GREEN,  
    sx / fontScale,  
    sy / fontScale,  
    mResponseLines[i],
```

---

<sup>9</sup> In <https://github.com/coelckers/gzdoom>

```

DTA_KeepRatio,
True,
DTA_VirtualWidth,
displayWidth,
DTA_VirtualHeight,
displayHeight
);

```

There are a ton of parameters here, but I've bolded the ones we care about - the colour to display the message in, and the actual text of the message. By copying and altering the DrawReplies method in which this happens, I was soon able to colour each choice according to which character it began with, matching the colours I'd selected for the new and visited transporters in the hub.

```

string nonMapChars = "<->";
string textToDisplay = mResponseLines[i];
string startChar = textToDisplay.Left(1);
int isMap = (nonMapChars.IndexOf(startChar) != -1);
int optionColour = Font.CR_GREY;
if (isMap) {
    if (startChar == " ") {
        optionColour = Font.CR_GOLD;
    }
    else if (startChar == "X") {
        optionColour = Font.CR_LIGHTBLUE;
    }
    else if (startChar == "*") {
        optionColour = Font.CR_WHITE;
    }
    textToDisplay = textToDisplay.Mid(2);
}

screen.DrawText(
    displayFont,
    optionColour,
    sx / fontScale,
    sy / fontScale,
    textToDisplay,
    DTA_KeepRatio,
    True,
    DTA_VirtualWidth,
    displayWidth,
    DTA_VirtualHeight,
    displayHeight
);

```

All of this logic is made of pretty standard stuff that you'd find in something like C#, and it isn't exclusive forbidden ZScript knowledge. However, I was flummoxed for a while because the line "`<->.IndexOf(startChar)`" (used for seeing if the start char matched one of the characters that would indicate it's a menu option) wouldn't work for some sort of deep engine reasons - fortunately, the helpful ZDoom Discord members were on hand to advise me to instantiate, assign and then use the string on separate lines.

This code adds an extra bit of decision-making before just displaying the text of the choice. It sets the default option colour to grey and then looks at the first character of the text to see if it's one of "<", "--" or ">" (the "<< PREVIOUS <<", "-- TOP/BOTTOM OF LIST --" and ">> NEXT >>" options. If it isn't, then it must be a map - so it looks at the first character of the text again and changes the colour to display depending on which of the three states the character represents the map is in. Having done that, it takes off the first two characters of the text, because now that we have the colour scheme we don't need the text to display with the prepended character any more. Finally, the DrawText call is made with the optionColour and textToDisplay that we've worked out from this process.

This was good, but it still wasn't quite in the spirit of ZScript because all I'd done was implement an alternative way to display text depending on its first character, with my three-reply backend still holding the system up. What I really needed was a way to read the state of each level from here - the data that was currently stored in the levelsDone global array in ACS.

### A global ZScript data store

ZScript requires you to think in a different way from ACS or DECORATE. It's more similar to non-ZDoom coding than either of those older languages are, but paradoxically I had got so used to the way that the odd ZDoom-exclusive languages think that it was difficult to re-learn the concepts of object oriented code and apply it to a ZDoom environment.

In DECORATE, you're working within an object called an Actor, and you set up a path for it to go from state to state. In ACS, you're writing out lists of things to do and setting up when to trigger them. ZScript does both of

these things - you create a class, define the things that it can do, then tell it to do things. Unlike DECORATE actors, ZScript classes don't necessarily have a physical presence in the level, making them immediately more abstract to think about compared to Doom's comfortingly physical way of doing things.

EventHandlers and Thinkers are two of the classes that ZScript allows access to, and both of them sit in the background and wait for things to happen. An EventHandler is similar to a script declaration in ACS - it can be set up so that it gets notified when certain events occur in the game such as a level being entered or the player dying, but its strength over ACS is that it can also get down to much smaller things like detecting when a line or actor is damaged, and being able to identify much more about the nature of the event - which line ID? How was it damaged, who damaged it, and by how much? And a Thinker is something that performs tasks - each dynamic light has a thinker behind it that controls its effects and position, and the game creates a thinker whenever a sector needs to move so it can control modifying the heights of the floor or ceiling.

And so, to make something to store data, I needed to make a new kind of Thinker. It would just have to exist in the background, hold a list of map numbers and their appropriate statuses - 0 for unvisited, 1 for completed and 2 for cleared of monsters - and allow the RampGuideMenu to retrieve things from this list as needed. (The list couldn't exist on the RampGuideMenu itself, because that object is only created when Doom's conversation system needs it - I needed something that was always present.)

```
class RampDataLibrary : Thinker
{
    //Level status index corresponds to map number
    int levelStatuses[200];

    //Quick, make this a STAT_STATIC thinker when we initialize
    RampDataLibrary Init(void)
    {
        ChangeStatNum(STAT_STATIC);
        return self;
    }
}
```

This is the outline of a basic Thinker that doesn't do anything yet. It provides an array for level statuses with 200 elements, and an initialization method. (If you're familiar with object-oriented code, it's important to point out that this is not a constructor as I had assumed - it's just a method that happens to be called Init, which you have to call manually when you instantiate the class.) This Init method only changes the StatNum of the thinker, which determines the kind of thinker that it is - a STAT\_STATIC thinker persists across levels, making it able to act like the global data store that I needed.

### Getting data from ACS to ZScript

With a place to store data in the ZScript environment, the next step was to find a way to put data into it. If I had been doing this from scratch without my existing ACS base, the way to do this would have been to set up an EventHandler that reacted to the hub map being loaded (the equivalent of the ENTER script) and to create and populate the object from there - but as I already had most of my logic in ACS, I could take a shortcut.

To communicate between ACS and ZScript, ZDoom provides the ScriptCall function. This allows you to pass parameters to any static function in a ZDoom class, just as if you were calling another ACS script - a static function is a function that's written as part of a class but that you call directly without the class being instantiated.

Three static functions were enough to read and write the levelStatuses array - two to call directly from ACS and one more to act as support:

```
static void WriteStatus(Actor activator, int position, int status)
{
    RampDataLibrary.GetOrCreateInstance().levelStatuses[position] = status;
}

static int ReadStatus(int position)
{
    return RampDataLibrary.GetOrCreateInstance().levelStatuses[position];
}
```

```
static RampDataLibrary GetOrCreateInstance(void)
{
    ThinkerIterator it= ThinkerIterator.Create("RampDataLibrary", STAT_STATIC);
    let p = RampDataLibrary(it.Next());
    if (p) return p;
    return new("RampDataLibrary").Init();
}
```

The read and write functions first call `GetOrCreateInstance`. As its name suggests, this creates a `RampDataLibrary` if it doesn't exist in the list of the game's thinkers already (note the explicit call to the `Init` method here, which will set it up as a static thinker). Whether it had to create a new one or just get the existing one, it then passes the `RampDataLibrary` back to the function that called it. This kind of pattern is called a singleton, and is commonly used when you need exactly one instance of a class to exist within an environment - every call to these methods should retrieve the same `RampDataLibrary`.

From there, the `ReadStatus` and `WriteStatus` functions can read from or write to the `levelStatuses` array on the `RampDataLibrary` object - and by using `ScriptCall` in my ACS to pass data from the ACS `levelsDone` array to `ZScript` each time the hub was started, I had a source of level statuses that would be reachable when the `RampGuideMenu` was called up.

```
script "debug_levelsdone" (void) {
    for (int i = 10; i < 199; i++) {
        log(d:i, s:": ", d:ScriptCall("RampDataLibrary", "ReadStatus", i));
    }
}
```

## Reading the level status data

I had only paid attention to the `DrawReplies` method of the conversation menu class before, in which the things that are passed to the function for it to display are just strings stored in an array called `mResponseLines`. To do some processing on the choices before they were lined up for display, I had to look at the `FormatReplies` function which is called before `DrawReplies`.



FormatReplies is responsible for:

- Deciding whether or not each choice should be displayed (based on the conditions regarding the player's inventory laid out in the DIALOGUE script)
- Deciding whether to display the cost of a dialogue choice (for things like Strife's shops, where you need a certain amount of gold to buy things)
- Splitting choices with long dialogue into multiple lines for display
- Adding the default "goodbye" text (or the one specified by the script) to the end of the choices to exit the conversation
- Calculating the Y position on the screen at which to start writing dialogue, based on the number of total lines.

To make this more specific to my needs, I added a replyMapStatuses array to the class in which I could keep track of how to display each line, so that the DrawReplies function could refer to that in parallel with the mResponseLines array (ideally I would have made DrawReplies accept a list of objects instead of just strings so I could include this data on each line, but I didn't want to overcomplicate things in a language I was unfamiliar with). FormatReplies then became:

- For each specified choice, first assume it isn't a map (write 3 to this line's replyMapStatuses entry).
- If the choice's second argument is above 0, it represents a map, because those choices are all set up to call script number 1 with the map number. Refer to the RampDataLibrary's levelStatuses array to decide whether the map status is 0 (not visited), 1 (completed), 2 (cleared out), then put that in replyMapStatuses.
- Don't split lines, truncate map names when they're too long instead
- Don't add any goodbye text

With this logic in place, DrawReplies could be made a lot simpler again - the working out of whether the level has been completed or mastered has already been done, so all DrawReplies needs to do is check the corresponding entry in replyMapStatuses every time a line is drawn and display it in the appropriate colour.

```

int textColours[4] = {Font.CR_GOLD, Font.CR_LIGHTBLUE, Font.CR_WHITE,
Font.CR_GRAY};
screen.DrawText(
    displayFont,
    textColours[replyMapStatuses[i]],
    sx / fontScale,
    sy / fontScale,
    mResponseLines[i],
    DTA_KeepRatio,
    True,
    DTA_VirtualWidth,
    displayWidth,
    DTA_VirtualHeight,
    displayHeight
);

```

## Scope

		Code scope		
		data	play	ui
Calling context	data	full	readonly	none
	play	full	full	none
	ui	full	readonly	full

And that would have been it... except scope was the last thing that confounded me. ZScript has a set of rules around what can be called by what else, and from what I can understand these rules exist in order to make sure that multiplayer games and demo playbacks stay in sync. In my case, Thinkers are designated as in the “play” scope (meaning they run during gameplay) and the conversation menu exists in the “ui” scope (it’s part of the user interface, or menus). And one of the rules is that UI-scoped classes can’t even think about creating any object in the play scope - therefore, if you attempt to run the RampDataLibrary’s GetOrCreateInstance method from the RampGuideMenu, you’ll get the following error message:

```
Can't call play function GetOrCreateInstance from ui context
```

I was unsure how to get around this obstacle for a while because this concept wasn't something I had ever had to deal with in my coding life, but eventually hit on the idea of providing an alternative method for the RampGuideMenu to call to retrieve the library:

```
static clearscope RampDataLibrary GetInstance(void)
{
    ThinkerIterator it = ThinkerIterator.Create("RampDataLibrary",
STAT_STATIC);
    let p = RampDataLibrary(it.Next());
    if (p) return p;
    return NULL;
}
```

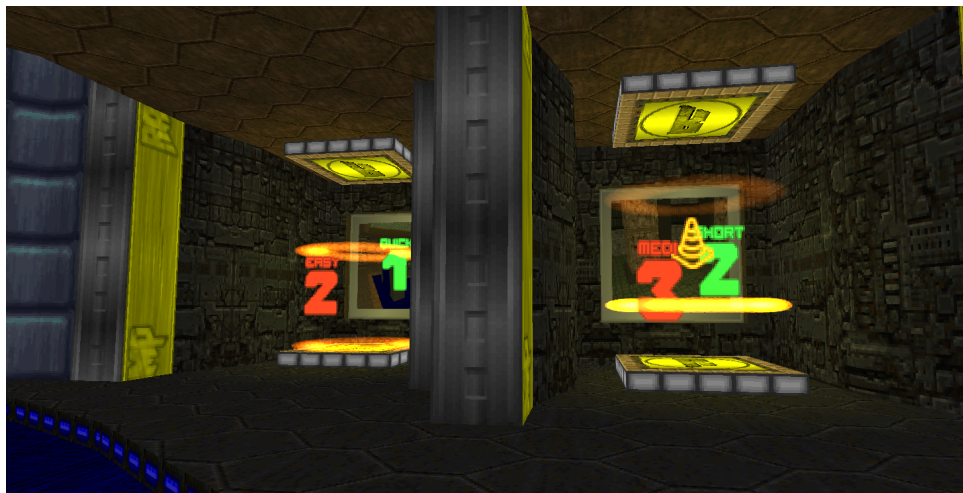
The “clearscope” keyword in the method signature, as far as I can tell from the documentation, designates this method as being allowed to be called from either the UI or menu scopes as long as it only reads. If the GetOrCreateInstance method hadn't been called already then the guide menu would just get a NULL back - but this doesn't matter, because the guide menu is never called before we've had the chance to set the instance up.

And with that last detail, everything worked - on entering the hub level, the RampDataLibrary would be instantiated and filled, and the guide conversation menu was able to read the level statuses from the library and display lines as appropriate. Later on, I would realize that having got this far it made no sense to keep taking up an ACS global variable, and would move all the interaction with the list of level statuses to my read/write CallScript calls to ZScript.

At this point, having achieved my first success with ZDoom's new (several years old by now) scripting language, I wondered about redoing the rest of the hub's code with ZScript as well. Then, realizing that I'd spent two days on something that was pleasing to develop but almost undetectable by the player, I opted to just be satisfied with this amount of ZScripting instead - so you'll be getting a description of the DECORATE and ACS flavour of the hub's garden minigame once you get to that chapter.

## Map descriptions

One of the other things I realized I would need as the project grew was a quick way to get across the content of a level from the hub - not just by showing a screenshot but with a brief visual summary of what it was like to play. Each transporter in the hub level shows an icon depicting the theme of the level along with some numbers giving a rating out of five for length and difficulty (these scores combined determine how many points a player gets for completing the level).



This data is provided to the game through what I would humbly call a truly world-class abuse of a GZDoom feature. I put this together before my ZScript adventure above, and in the world of ACS and DECORATE there's no such thing as a lump that you can just treat as a store of arbitrary data... but the LANGUAGE lump acts a bit like one.

As you would guess from the name, the GZDoom LANGUAGE lumps are intended to aid internationalization - instead of storing strings that will be visible to the player directly in your code, you store them in a file that has variations of them for different languages. You then refer to those strings by keywords - for example, instead of a locked door displaying the locked yellow door message directly, it looks up the keyword PD\_YELLOWWK and displays whatever value it points to. If you're playing in English, this is the

familiar “You need a yellow key to open this door” message from the original DOS version, but if you have the language set to Italian, it’ll look up the same keyword in the Italian language lump and will display “Ti serve una chiave gialla per aprire questa porta”. If no specific language has a keyword for a string it will fall back to the default language (the language spoken in america that approximates English).

So if you set up some keywords in only american English and define that as the default, they’ll be the same no matter what language GZDoom is being played in and can be treated as a store of key-value pairs.

```
[enu default]
MAP10LENGTH = "XX"; MAP10DIFFICULTY = "XXX"; MAP10THEME = "City";
MAP11LENGTH = "X"; MAP11DIFFICULTY = "XXX"; MAP11THEME = "Temple";
MAP12LENGTH = "XXXXX"; MAP12DIFFICULTY = "XXXX"; MAP12THEME = "Tech";
MAP13LENGTH = "X"; MAP13DIFFICULTY = "XX"; MAP13THEME = "Tech";
```

In a usual LANGUAGE lump each key and value would be on a separate line, but happily I found out it also allows you to arrange them like this which makes it a bit neater for my purposes. This file was generated from a Google Sheet<sup>10</sup> that I kept during the project to record my testing notes and other bits of data that I would need, and I just copy and pasted it into the fixed content folder every so often (it would have been possible to generate it by hooking into the Google Sheet API at build time but I didn’t want to spend the time to get it set up).

Having got that LANGUAGE lump into the PK3, it was possible to use it together with ACS’s rather unorthodox method of nailing strings together and wrangle the arrangement into behaving a bit like an associative array. This code is called in a loop when the hub is loaded, to set up the appearance of the level portals:

```
int mapspeciallength = StrLen(StrParam(1:StrParam(s:"MAP",
d:levelnumber, s:"LENGTH")));
int mapspecialdifficulty = StrLen(StrParam(1:StrParam(s:"MAP",
d:levelnumber, s:"DIFFICULTY")));
```

<sup>10</sup> <https://docs.google.com/spreadsheets/d/1MoLDIWT2txqAdFv00vawNeXk663W-YyrDiP7eZdjli0>

```

int lengthTid = levelnumber + OFFSET_PORTAL_LENGTH;
SpawnSpotForced(StrParam(s:"RampLengthHover", d:mapspeciallength),
levelnumber + OFFSET_YELLOW_PORTAL_LIGHT, lengthTid, 0);
SetActorPosition(lengthTid, GetActorX(lengthTid) + 24.0,
GetActorY(lengthTid) + 24.0, GetActorZ(lengthTid), false);

int skillTid = levelnumber + OFFSET_PORTAL_SKILL;
SpawnSpotForced(StrParam(s:"RampSkillHover", d:mapspecialdifficulty),
levelnumber + OFFSET_YELLOW_PORTAL_LIGHT, skillTid, 0);
SetActorPosition(skillTid, GetActorX(skillTid) - 24.0,
GetActorY(skillTid) - 24.0, GetActorZ(skillTid), false);

```

If there was ever a piece of ACS code that cries out for explanation it's this one, so let's step through it starting with the nest of parentheses that form the expression on the top line:

```

StrLen(StrParam(l:StrParam(s:"MAP", d:levelnumber, s:"LENGTH"))));

```

Working from the inside out, the first function called is the rightmost occurrence of StrParam - an enigmatically-named function that allows you to concatenate strings, numbers and various other things together into a single string. The character on the left of the colon describes how to interpret each term - s stands for string, and d probably stands for decimal (I'm not certain, but personally that's how I remember that d = a number). What StrParam is doing here is taking the literal string "MAP", the level number it's examining, and then the string "LENGTH" - therefore, if we're looking at map 10, the result would be MAP10LENGTH.

This now looks a lot like the key that was set up in the LANGUAGE lump! So we pass the result of that into another StrParam function, this time with the prefix l. This tells StrParam not to interpret this as a literal string, but to go off and search for a language key with this name. It finds it, and the value of MAP10LENGTH is "XX", so this is the result of that StrParam call.

Why "XX" and not the number 2? Well, this is another duct-taped workaround to this method - StrParam allows you to convert numbers to

strings, but once you have a string in ACS it's very difficult to change it back into a number. Ways to do it exist, but there's another way that isn't immediately obvious to get a number out of a string - to just look at its length.

Therefore, the strings in the LANGUAGE lump aren't actual numbers - they're strings of letter Xs, with the number of Xs representing the number we want. The last thing done on the line is a StrLen call on our result, and in this example this means we finally end up with the number 2 as the length rating for the map. (For display purposes, getting the string "2" back would have been OK because we don't use it as a number, but further down the file it has to be a number so that it can be used in points calculations.)

With that done, the same hideous procedure happens for the difficulty, then the objects that show the numbers are spawned (again, using StrParam with a string and number to construct the whole class name - in map 10's case it has length 2 and difficulty 3, so we want RampLengthHover2 and RampSkillHover3). They're offset by 24 map units on the X and Y coordinates so that no matter which cardinal direction the mouth of the portal happens to be facing, both digits will be visible to a player from the outside.

I have the feeling that I should conclude this section by apologizing to anyone who's ever contributed code to GZDoom - now that I explain this I can see how nonsensical and impractical it is, especially now that I've graduated to exploring ZScript. I can promise you that the details behind the next part of RAMP's implementation are far less mad, even if the concept isn't.

# Ultraviolet Gardening



I am as surprised as anybody else that RAMP contains a garden-building minigame. In truth, when I started the project I hadn't really thought about the game that would grow around the individual maps - when I imagined the project being sixteen maps long it didn't seem to need one. My inspiration DUMP had made the player choose and complete a certain number of maps from the selection before unlocking a boss level, but I wanted to do something a little different and attempt to encourage people to play all of the vast collection.

The most effective way that I'm aware of to keep a player engaged is to offer them the promise of continuous reward. It's something that the Civilization games are particularly good at (and it's what keeps me playing the dozens of Etrian Odyssey games as well) - you're always just one turn away from something exciting happening, like gaining access to a new technology, or finishing a Wonder, or finally capturing the last of Montezuma's cities. With the format I'd chosen, I couldn't let the rewards have an effect on the gameplay like this, but I could at least appeal to the player's sense of aesthetics.

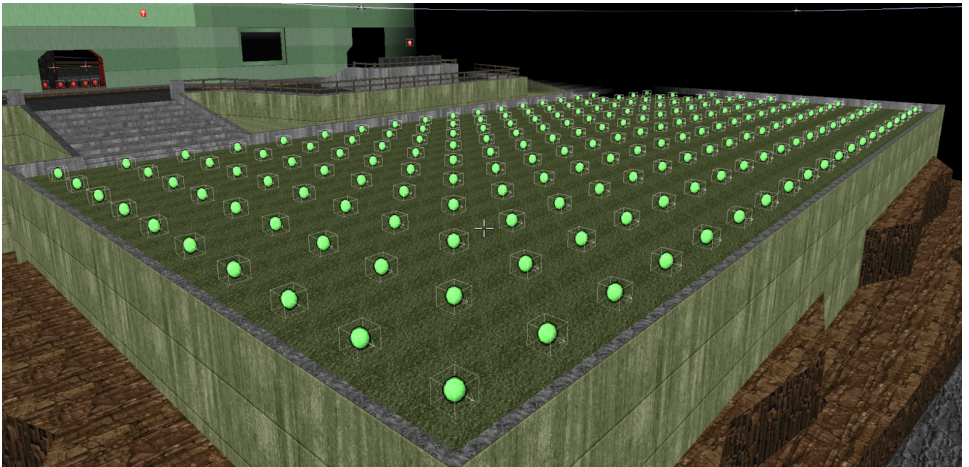
In Doom Eternal, the Tony Stark-style battlestar-castle-command centre changes gradually throughout the game - blank spaces are filled with vinyl



records, action figures and other assorted memorabilia that the Doomslayer collects throughout the levels. In this spirit, I thought of setting up a trophy garden that the player could visit that would gradually change and populate throughout the game.

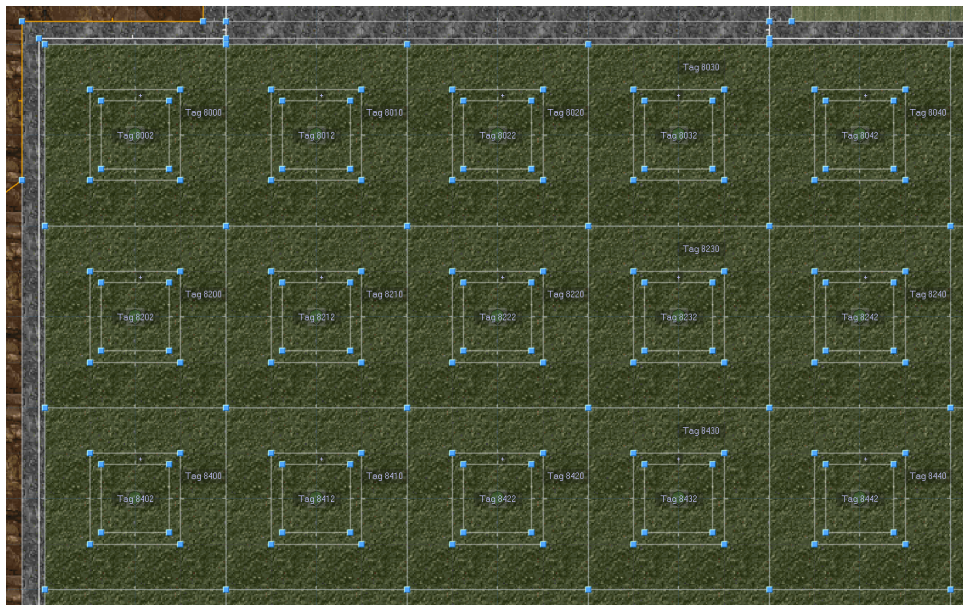
In snapshots of RAMP from about halfway through the project, an outdoor area appeared in the south of the hub map with five plinths. On top of these, I made various dolls from EndHack's collection on Realm667 appear depending on how many levels the player had completed. It worked as a proof of concept, but it definitely wasn't very exciting - and with other parts of the hub always demanding my attention with the increasing number of maps submitted, I didn't know how to make the arrangement more attention-grabbing. And therefore, something possessed me to code an editor that would let the player lay out their rewards themselves.

## *Garden layout*



The garden area is a 20 by 10 grid of 128-unit squares comprising three concentric sectors and a MapSpot. The MapSpots are numbered starting from tag 8000, with the tag number increasing by 10 towards the east and by 200 towards the south - an awkward numbering scheme, but I knew I could abstract it away with a function so I didn't need to worry about it while coding. The outer sector for each square is tagged the same as the

MapSpot, with the two inner sectors' tags being 1 above and 2 above this number.



The idea was that the player would be able to customize each of these squares to give it a floor style like a path or water, and for the middle two sectors to form a plinth if the player happened to place a trophy on the square - although very quickly I realized it would be better to treat plinths and trophies as separate ideas so that I could do more with plinths besides put trophies on them. The data behind the garden is kept in three more global arrays - gardenObjects for the things on the squares, gardenPlinths for how the middle two sectors should appear to form a plinth, and gardenTerrain for the ground type (of all three sectors per square if the square is blank, or just the outer one if a plinth is present).

Like everything in this project, the concept evolved a bit as I implemented it, but mercifully I did everything I wanted on my first try on the physical layout of the garden tiles and they stayed the same throughout.

## Editing the garden



Writing an editor in ACS sounds like a ridiculous task, and to be fair, it is - but the concepts are the same as scripting anything else. It's possible to take the player out of their own first-person view by using Camera objects, which are often used to sweep across the landscape in titlemaps, or in levels to let the player see what's going on in another part of the map - and after that, everything else is just about moving objects around and manipulating sectors and things, the same as any other script.

The player gets access to the garden minigame through the computer terminal they encounter when they visit the south side of the map - just like the guide system, it activates a conversation with a tree of options that result in a script being called with certain parameters when the player eventually selects an item to place. This particular script is called gardenBuild (except because we can't call a named script from a conversation, we call script number 2 which just forwards its parameters directly on to gardenBuild).

This script accepts two parameters, both of which are integers (not that it has much choice in ACS): a mode, and a tiletype. The mode lets the script know if we're placing a floor, a Thing decoration, a plinth/box or a

trophy. The tiletype denotes the specific thing that we want to place. The objects and tile types available for the player to use are stored in an unwieldy ACS database consisting of multiple long arrays to describe the names of the items, the textures and heights that should be associated with them, and so on - I had another Google spreadsheet to keep track of these and to write them out into a neat list of lines that I could just copy and paste into the map's script.

```
27  //// DATABASE - AUTO GENERATED
28
29  int GARDEN_INNER_PLINTH_HEIGHTS[13] = {0, 30, 0, 28, 30, 28, 16, 46, 46, -4, 8, 8, 8};
30  int GARDEN_OUTER_PLINTH_HEIGHTS[13] = {0, 28, 8, 28, 32, 32, 24, 48, 48, 0, 16, 16, 16};
31  str GARDEN_OUTER_PLINTH_TEXES[13] = {" ", "WOODF1", "FLAT20BR", "GROUND23", "DEM5_5", "DEM4_5", "FLAT20BR", "R",
32  str GARDEN_INNER_PLINTH_TEXES[13] = {" ", "WOOD_N07", "GROUND26", "FLAT20BR", "DEMGOTH", "DEM6_6", "FWATER1",
33  str GARDEN_OUTER_PLINTH_WALLS[13] = {" ", "WOOD_N12", "FLAT20BR", "GROUND23", "DEM5_5", "MARBBLK2", "FLAT20BR
34  str GARDEN_INNER_PLINTH_WALLS[13] = {" ", "WOOD_N12", "FLAT20BR", "GROUND23", "DEM5_5", "MARBBLK2", "FLAT20BR
35  str GARDEN_PLINTH_NAMES[13] = {"(Nothing)", "Wooden Plinth", "Small Planter", "Stone Plinth", "Marble Plint
36  int GARDEN_PLINTH_COUNT = 13;
37
38  int GARDEN_FLOOR_HEIGHTS[16] = {0, -2, 2, 1, -2, 3, 3, 3, 3, -2, -1, 3, 3, 0, 0, 0};
39  str GARDEN_FLOORS[16] = {"OSOILA02", "SLIME01", "FLAT14G", "GROUND12", "FWATER1", "GROUND23", "OMRBL44", "OMR
40  str GARDEN_FLOOR_NAMES[16] = {"Soil", "Mud", "Grass", "Path", "Water", "Stone", "Grey Marble", "Red Marble", "C
41  int GARDEN_FLOOR_COUNT = 16;
42
43  str GARDEN_DECORATION_NAMES[12] = {"(Nothing)", "Rose Bush", "Tree", "Lamp", "Argemone", "Blue Pea", "Conical
44  str GARDEN_DECORATION_CLASSES[12] = {"", "RoseBush", "Gardentree1", "TechLamp", "Argemone", "BluePea", "Conic
45  int GARDEN_DECORATION_COUNT = 12;
46
47  int GARDEN_PLINTH_POINTS[13] = {0, 0, 118, 169, 728, 795, 558, 965, 897, 84, 660, 423, 1066};
48  int GARDEN_DECORATION_POINTS[12] = {0, 321, 592, 186, 457, 389, 152, 626, 33, 101, 931, 491};
49  int GARDEN_FLOOR_POINTS[16] = {0, 0, 50, 135, 220, 355, 694, 1033, 999, 863, 67, 762, 829, 524, 287, 16
50
51  int TROPHY_THRESHOLDS[34] = {10, 100, 200, 200, 2, 5, 50, 100, 9, 10, 200, 300, 400, 500, 10, 25, 50, 1
52  int TROPHY_TYPES[34] = {1, 2, 3, 2, 3, 3, 1, 1, 4, 4, 1, 1, 1, 1, 3, 3, 3, 4, 4, 4, 2, 2, 2, 2, 2, 2, 2
53  str TROPHY_NAMES[34] = {"Ramping Up", "Bronze Doomguy", "Gold Doomguy", "Silver Doomguy", "Chocolate Minigu
54  str TROPHY_REASONS[34] = {"taking your first steps into the RAMP levels", "completing half of the RAMP 1
55  int TROPHY_CLASSES_IDS[34] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
56  int TROPHY_COUNT = 34;
57
```

After looking up the name of the thing the player wants to place for the sake of the display at the top of the screen, the code will make sure that the objects that represent the arrow cursor and the four corners of the highlighted tile are created and set them back to where the player left them the last time the editor was opened. It will then freeze the actual player in place and enter an infinite loop where it's checking for inputs for the garden controls.

## Setting up the editor

ACS provides the function `GetPlayerInput` with two options to interpret the general input from the player - it can be called with `INPUT_BUTTONS`, which will return a bitmask representing the buttons that the player currently has held down, and `INPUT_OLDBUTTONS`, which represents the buttons that the player had held down on the previous frame. By

comparing the two, it's possible - if rather long-winded - to detect whether a player has just hit an input (a keydown), is currently holding an input, or has just released an input (a keyup).

```
if (buttons & BT_FORWARD && !(oldButtons & BT_FORWARD)) {  
    if (gardenCursorTile < 8200) continue;  
    targetTile = gardenCursorTile - 200;  
    SetActorPosition(  
        GARDEN_TID_CURSOR,  
        GetActorX(GARDEN_TID_CURSOR),  
        GetActorY(GARDEN_TID_CURSOR) + 128.0,  
        GetSectorFloorZ(targetTile+2, 0, 0) + 64.0,  
        False  
    );  
    gardenCursorTile = targetTile;  
}
```

For the case of the player moving forward (towards the north), the code checks if the player was not holding the Forward input on the previous frame but is now. If the tag of the current location of the cursor (kept track of in `gardenCursorTile`) is below 8200, then it must be on the top line already, so it ignores the input. Otherwise, it will use the `SetActorPosition` function to move the cursor 128 units to the north, and 64 units above the floor level of the centre sector of the tile it's moving to (the central square's tag is 2 greater than the tag of the square's corresponding `MapSpot`). After this piece of code, the objects representing the corners of the target tile get updated in a similar way.

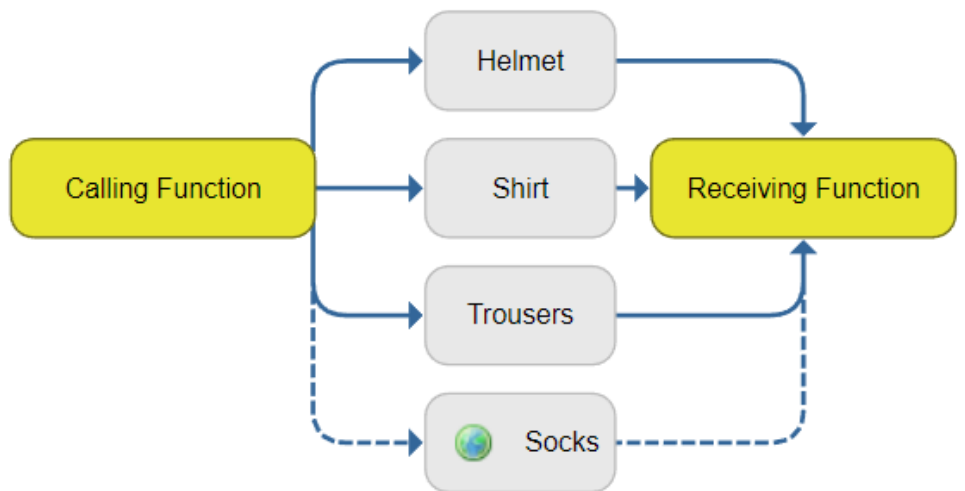
## Editing a tile

So moving about is taken care of - now we need to be able to manipulate the square when the player presses Attack or Jump to place our selected object. The exact actions depend on the mode that we're in, but the aim is to pass the required sector number, along with the ground type, plinth type and object it should be given to a function called `gardenSetTileBySector`.

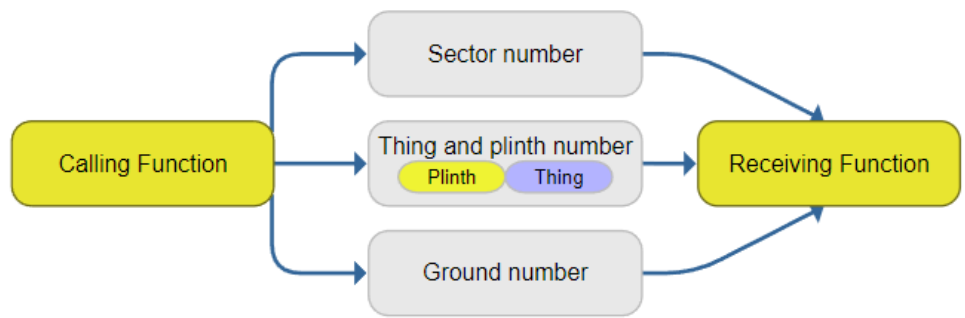
Except we run into a bit of a problem here, because one of ACS's little oddities is that it doesn't allow more than three parameters to be passed



to a function. Sometimes I've got around this by storing extra "parameters" in top-level variables instead just before the function is called, and having the accepting function read them:



However, on this occasion, I chose to use a small amount of bitwise arithmetic because I was never going to need a value more than 127 on any of the parameters. Therefore, the parameters became the sector number, a variable composed of [the plinth number bitshifted to the left by 8] plus [the thing number], and then the ground type number on its own.



After setting the appropriate values in the gardenObjects, Terrain and Plinth global arrays, the gardenSetTileBySector function now calls gardenRedrawTile with the number of this tile. (I came up with the idea of "tile numbers" to make referring to them easier, but I was really inconsistent about whether functions accepted the sector number to work

on, or the tile number - the designated tile numbers ascend in 1s from left to right, top to bottom, so the tile number of a sector is the sector tag minus 8000 then divided by 10.)

The `gardenRedrawTile` function will set the tile's three sectors back to height 0, remove the Thing tagged with [this sector's number + 1], which is the decorative Thing currently spawned in that sector. Then it will interpret how to set up the tile for display from the three global arrays:

- Set all three sector floors to the required ground texture.
- Handle the Thing that should be placed in this sector:
  - If the number we were given for the object is below 20, this is a junk piece that appears as part of the litter in the garden at the start of the game - spawn the Thing type associated with this ID at the sector's MapSpot, then give it a random offset from the centre.
  - If the object number is between 50 and 100, this is a trophy - construct the appropriate class name by appending the ID to the string "RampTrophy" and then spawn it at the MapSpot.
  - If between 100 and 128, this is a decorative class that isn't a trophy - look up the class that should be spawned according to this ID and spawn it with no offset.
- Handle the plinth for this sector - if plinth number is above 0, then look up the appropriate heights and textures for the inner two sectors from our set of arrays, then change the floors and line textures as appropriate.

There was one hangup that I had while writing this section - it turns out that sectors don't really take kindly to being told to change their heights twice in the same tick. So instead of directly using a floor movement action to tell the full tile to reset to 0 and then for the sectors to go to their appropriate height again (and then again if those sectors were part of a plinth), I kept an array of target heights for each sector and then had yet another function, `gardenSetFloor`, set all the floors appropriately at the end. Even this had its annoyances, because even though there seem to be half a million different functions to move a floor in GZDoom, just being able to instantly set a floor to a specified height is not one of them - so I

used the best approximation that happened to work, working out the amount that the floor had to move from its current position and then using `Generic_Floor` to do it.

## *Loading and saving*

Because the hub level is not a true hub, I had to do the same thing here that I did with the level teleporter statuses, and restore the state of the garden manually when the player was finished with a level. But this turned out to be the most straightforward thing in the entire project, because I already had a function to redraw a tile by using the data from a set of global arrays - all that the game needed to do was loop through the 200 possible tiles when the hub was entered, and call `gardenRedrawTile` on each of them.

Additionally, when the game is started, the garden is full of random rubbish - this is done by looking to see if the player has no previously set map number that they came back from, and then picking random terrains and objects from a limited selection.

## *Unlocking rewards*

The player doesn't have access to all the garden items from the start of the game. For trophies, this is an obvious requirement, as they're awarded for certain milestones, but the size of RAMP was so enormous that only including those wouldn't have resulted in the sense of continuous reward that I was going for. Therefore, I needed to start the player out with only some basics, and to allow new decorations and terrains to be placed as the game progressed.

The data for this, at least, is very simple. There are three arrays in the ACS database that hold the number of points the player needs to have collected to use each plinth, decoration or terrain:

```
int GARDEN_PLINTH_POINTS[13] = {0, 0, 118, 169, 728, 795, 558, 965,
897, 84, 660, 423, 1066};
```



```
int GARDEN_DECORATION_POINTS[12] = {0, 321, 592, 186, 457, 389, 152,
626, 33, 101, 931, 491};
int GARDEN_FLOOR_POINTS[16] = {0, 0, 50, 135, 220, 355, 694, 1033,
999, 863, 67, 762, 829, 524, 287, 16};
```

To act on this data, I added yet another section to the increasingly bloated `comeFromMap` function called at the start of the hub level. For each of the reward types, it will look to see if the player's score exceeds each item's requirement, and if it does, it will give the player an inventory item that flags them being allowed to use that item - it will also print a message to the screen if that new item hasn't already been announced (kept track of in yet another global array). The same idea is used for opening the garden at all - a minimum score of 10 is hard-coded in, and the game will direct the player to the garden the first time they come back to the hub with a score exceeding this value.

The awarded inventory items then come into play when the player is in the conversation with the garden computer - the options are made visible depending on which inventory items are present. Due to the limited number of options available on a conversation page I had to subdivide each of the three categories into two finer ones, with ground types being separated into terrains/walkways, plinths being boxes/planters and things being decorations/greenery - but eventually I fit them all.

It would have been possible to do what I did for the guide computer here as well, make a custom conversation class that just looked at the player's score directly and avoided any use of the inventory items at all - but I'd reached project fatigue a long time ago by this point and the use of inventory items for flags like this was nowhere near as flagrant an abuse of the system as the triple-optioned guide menu was.

```
page { // page 3
    name = "Terrains";
    dialog = "Choose the terrain you want to place.";
    goodbye = "Close (ESC)";
    choice {text = "<< BACK"; NextPage = 2; }
    choice {text = "Soil";
```

```

        require { Item = "RampGardenTerrain0"; }
        special = 80;
        arg0 = 2;
        arg3 = 0;
    }
    choice {text = "Mud";
        require { Item = "RampGardenTerrain1"; }
        special = 80;
        arg0 = 2;
        arg3 = 1;
    }
    choice {text = "Grass";
        require { Item = "RampGardenTerrain2"; }
        special = 80;
        arg0 = 2;
        arg3 = 2;
    }
}

```

## Awarding trophies

The point of all that groundwork was to allow the player to receive and place trophies. The code treats these in the same category as the other decorative items like pillars and hedges, but the way they're awarded and placed is slightly different from the other objects.

On entering the hub level, after the unlocking of the regular items has been done, the `announceNextTrophy` function is called. This will further call another function `findNextTrophyID` to find the first trophy that's available for the player to place.

Trophies come in four different types:

Type	Description	Trophies
1	Given for crossing score thresholds, like the decorative items	9
2	Given for completing a certain number of levels	12
3	Given for mastering a certain number of levels	6

	(completing them with zero monsters remaining)	
4	Given for completing certain sets of maps, or completing any map on Nightmare difficulty	9

Trophy types and their thresholds (the number of points, maps, etc needed to unlock them) are also stored in the ACS database. The findNextTrophyID function loops through all the trophies, checking to see if the threshold of any trophy has been met according to its type. If it finds one, it will return its trophy ID number, letting the announceNextTrophy function then display its name and details. Multiple trophies are never announced at a time - only the first one whose object is not currently present in the garden and which meets the requirements.

Trophies of type 4 don't have specific thresholds to just check against, and these reach into ZScript again - they were intended to require special unique circumstances, but the unfortunate reality is there is only one special one. The HasDoneTrophyLevels ZScript function simply looks to see if all of a selection of levels have been marked as completed or mastered, and sends back an acknowledgment if they have - the only trophy to do otherwise is the cacodemon-shaped bush, which is awarded for completing a level on Nightmare difficulty via a quick hack specifically signalling this in comeFromMap.



The garden editor uses this same function to decide whether to allow the player to place a trophy. The “Place Trophies” option is always available in the garden conversation, and when it’s selected it will enter the garden editor in a fourth mode specifically for placing trophies.

This fourth mode uses the same `findNextTrophyID` function as the start of the level to find the first trophy available and not currently placed in the garden. If it doesn’t find one, it will just quit the editor with an apology - if it does, it will place a representation of the trophy on the cursor and will wait for the player to place it like a standard decoration.

The code to place a trophy works slightly differently as well - on putting a trophy down, an effect will be spawned at the trophy’s location, there will be a slight delay to allow that effect to play, and the editor will then immediately return control to the player so that a second trophy of the same type can’t be placed. On exiting the garden editor, `announceNextTrophy` is called again. This will show if another trophy is available in addition to the one the player just placed - it also makes sure that the player gets the chance to recreate trophies if they happen to be deleted.

Despite the amount of work detailed here, I was able to write the majority of the garden minigame in a weekend and then just plug in more trophies and other rewards as I went on, and I really enjoy that it gives RAMP a unique attraction not seen in other vast level collections. I’m not sure whether it actually served to keep anyone interested beyond where they would have reached with just the maps, but I’ve been pleased to see a few screenshots of people’s cultivated gardens online.

# Regret Analysis: Mortem (Post-)

Not regrets, exactly, but every project has some things that the maker looks back on and realizes that they should have done differently in hindsight. When a sequel to RAMP comes around (and I will make sure there is one!), these are some of the things I'd like to improve or do differently.

## *The sheer size of it*

I'm fairly sure that as of 2021, RAMP is the largest single-WAD community project the Doom community has ever put together. It's a hexawad, if such a word exists, being over six times the size of a standard megawad. And I'm not trying to say this in a boasting way - in fact, its ridiculous size caused many problems and I'd be very surprised (and very congratulatory) if anyone plays it all the way through without getting overwhelmed, exhausted or dying of old age.

The unexpected acceleration of the project meant that the design of the hub map RAMP Central is rather inconsistent. I made a basic techbase hub before the project started with some fairly detailed, complex rooms that housed sixteen level teleporters, and I thought I would be able to expand this environment and add a couple more level portals at a time as the project grew. However, as we all know by now, the pace just kept on getting faster - adapting desperately to keep up, my style of adding a couple of them at a time gave way to designing larger rooms that could house five or six portals, and eventually to the much larger rooms around the edges of the map where I was adding ten or so at a time.

It's hard to say how exactly this could have been avoided without making the hub map a lot more boring - I could have made more regular and predictable rooms with ten or so portals each to keep things more organized, but I think it would just feel too stiff and at that point you might as well just give the player a menu. As it is, I love the hub's chaotic nature, with a tangle of vaguely technical looking distinct environments to explore like a miniature Metroid map. Very near the end of the project I added area names that pop up when you move from one to the other like

Energy Routing or the Central Atrium, which gave them a bit more personality and (I would like to think) aided the player's navigation through them. On the other hand, having a random distribution of maps makes it hard to keep track of them - some people in the Discord chatroom suggested organizing maps by difficulty, or by theme with different sections of the hub to match, but in the end I decided that it would be more effort than it was worth to rearrange the entire map list.

The need to design the areas of the hub so rapidly was one issue, but a closely related one was the repetitive mechanics of copy and pasting the level portal objects to make new ones (and really having to concentrate so that I didn't mismatch the tags of any lines, sectors or things). As listed in the chapter about designing the hub, for each new portal created I needed to update:

- The tags of the blue and yellow dynamic lights above the portal pad
- The tag of the map spot to place the player on coming out of the level
- The parameter of the Actor Enters Sector thing that calls the script to display the level name and details
- The tags of the lines bordering the transporter alcove
- The parameter to the goToMap function called by crossing the line on to the teleporter pad
- The tags of the sectors representing the teleporter pad and the alcove around it
- The screenshot for the back of the alcove

It wasn't quite as overwhelming as this list makes it sound because I could select multiple objects at a time and update them in bulk - often copy and pasting multiple complete portals and shifting their tags all at once to turn portals 52, 53 and 54 into new portals 72, 73 and 74, for example. But fewer objects to update and fewer potential things to get wrong would have been very welcome. Thinking about it just now, I can think of a way to at least get the number of Things to place down to one, with a single mapspot that would create the rest of them with the appropriate tags at the appropriate places (but can you set parameters on

created objects? I haven't yet looked.) Perhaps with some more ZScript trickery it could be made even more automatic.

## *Easy uploading is a double-edged shotgun*

Without the RAMPART uploader system, it would have been impossible to have run this project solo - the number of submissions and updates was utterly ridiculous, usually averaging four or five new maps per day but with a huge rush of 17 and then 20 maps in the final two days. I can only imagine (with a significant amount of dread) the mess that would have resulted if I were taking map submissions the traditional way and having to sort everyone's newest versions out.

However, I think that the presence of the uploader is also the reason that the project got as bloated as it did. By being able to upload something and then see it included in the project instantly, you're not dealing with a project leader directly and you don't have to think about the time and effort it's going to take to accommodate your contributions. Some of this is on me because I emphasized how easy and automatic the submission process was, while deliberately keeping some things as a manual process (so that I could review new textures, sprites and so on), vastly underestimating the amount of custom content that would be submitted.

At the beginning of the project, a few people asked me if it was all right for them to submit more than one level - and I always replied "You can submit as many as you like, within reason" because I didn't want anyone to feel like they couldn't try more than one idea out. It turns out I should have specified what "within reason" actually meant numerically, because there are a noticeably lopsided number of maps in the hub per author. In the chatroom, some contributors were getting into a strange sort of arms race, or a competition that I had never intended for the badge of honour of submitting the most maps - never a hostile one, but there was definitely an odd and unexpected attitude in a couple of people who seemed to feel the need to submit as many maps as was humanly possible.

Candidly, this struck me as rather self-centred - even though I was expanding the hub so nobody would miss out on slots, by submitting

many maps you're still taking up a greater percentage of the hub for yourself and burying the maps of people who have just submitted one or two. As a side effect, it took more of my time away from helping other people, with my constant need to concentrate on expanding the hub to accommodate more maps. Often these same people would contact me asking why I hadn't made any video updates, on days where I'd had to spend all my time making sure their large list of levels were working and accessible.

So while giving people freedom to design and contribute is something very important to this project that I don't regret, I should possibly have thought more about how to set people's boundaries more reasonably and protect both other people's contributions and my own sanity. Next time around, I don't think a three-map limit would be unreasonable. Or perhaps the central map could be a hub for episodes instead of individual levels.

## *Moderation and filtering*

The A in RAMP stands for "All-comers" and that part of the project is very important to me. The project is aimed at everyone, from beginners taking their first steps to mappers who have been around a while and want to try new ideas. With this in mind, it's very difficult to justify imposing any sort of minimum quality threshold - if a beginner submits something that's totally flat with default STARTAN textures, one BFG and a pile of a hundred imps (which would still be better than my own first attempts at maps) then they should be welcomed to start there and be coached as to how to improve it!

But some submitted maps tested the completely open nature of the project - some people asked if they could submit their previously constructed 2,000-monster epics, and I tried to steer them away from that by asking them to consider the players of the complete WAD and how they'd react to a map that would grind their progress to a halt. I think that guidance was mostly successful, but there are definitely still maps in there that really slow the pace down.



The first real challenge to my philosophy was uploaded about halfway through the project, a clear joke map with chaotic geometry, gigantic untextured walls and a little over six thousand monsters. It was utterly unplayable, but the uploader of it insisted that it was exactly as he wanted it and wouldn't listen to anyone who attempted to explain how unenjoyable it was. As the leader of the project it was my responsibility to decide what was allowed in, but I really wasn't sure how to reconcile the idea of "everyone who's starting out is welcome to submit" with "I don't like this map so I won't include it", and I'll have to think about this for any similar community projects in the future. In this case, it was fortunate in a way that the author of this map made some very inappropriate comments in the project's Discord server and was atrociously rude to one of the other mappers, so I was handed a much more concrete reason to remove him without agonizing over philosophy.

That brings me to another kind of filtering that falls to a project leader, which is moderation of the community that you build. For people like me who are anxious about confrontation, this is a difficult role - you need to be level-headed and fair, and no matter how well you do it you'll still have to put yourself into positions where not everybody will like you.

However, you've got to remember that as a moderator, your job is to moderate - and how you do that has a huge impact on the atmosphere of the community you create. There's another excellent article by Eevee<sup>11</sup> that talks about setting boundaries in online communities, and how that differs from actual written rules - if you try to be too objective, you'll inadvertently encourage troublemakers to find inventive ways to skirt just under the line, and you'll be hesitant to remove anyone because they're not technically breaking the rules of the community. Unfortunately, sometimes you have to remove someone who just creates a bad environment - and one of the great secrets of moderation (to an anxious person like me) is that you have no obligation to justify yourself to them! You have to remember that no matter how much obnoxious people like to scream and shout about their rights, you have the right (and the responsibility) to remove them so that other people can enjoy themselves.

---

<sup>11</sup> <https://eev.ee/blog/2016/07/22/on-a-technicality/>

I hope that this part doesn't sound too condescending in likening members of an internet community to small children, but I think that a couple of techniques that I picked up through my daughter Penny's toddlerhood really helped me deal with the people who weren't creating a good atmosphere. The first part is that very young children can't detect things like other people's emotions or tone of voice - a handicap that the Internet recreates even as we do our best to try to get tone across in text. If someone is being rude and demanding, you need to outright say it instead of quietly tolerating it, because otherwise there's no feedback on their behaviour. In the best case, they might not even have realized that they were coming across that way. Some other people might get defensive, but even if they do, they will often remember the message and adjust their behaviour a little.

The other technique that I found applicable was in making problems and consequences clear. I didn't want any actions that I took to be a surprise - the first time that there was an issue, I would summon my courage and speak up, asking to please keep these things off the Discord (whether it was people bringing in arguments from elsewhere, or antagonizing others, or anything) - in the public chat to make clear that I was moderating, and in private with a direct message asking them not to continue. I tended to phrase it as "I don't like doing this, but if that happens again I'm going to have to remove you from the server".

After someone had been warned, the next time I saw them making others uncomfortable I would kick them out with a short note explaining why. Then I ignored any further attempts to communicate with me - one of them sent me a flood of friend requests to get my attention before I just blocked him outright. It's true what I said above - I really didn't like doing this, I would have preferred to coach them into being pleasant members of the community and I really thought I was getting somewhere at one point - but sometimes it just isn't worth it. The general opinion of the rest of the community was that I had been too patient with them, so I might have been too optimistic as it was.

## *Trophies and Awards*

As much as I enjoyed putting the trophy garden together, I don't think I quite gave it the content that it deserved - towards the end of the project I was running around making sure that all the maps were completable and no new bugs had emerged since I last played them, and I didn't have a lot of time to think about what the player should be rewarded for. I would have loved to have specific little achievements in some levels, but that would have required me to go into the maps after they were completed, fiddling with the author's intent - nevertheless, having just one special circumstance among all the trophies just naturally being awarded if you choose to fully complete the game seems a bit plain.

Relatedly, adding the assumption that all monsters on a level could reasonably be defeated was a massive mistake - never change the rules of a community project after you've already got a significant number of submissions. In most levels it was perfectly possible to get 100% monsters, but some of them just didn't have that in their design and needed their star awarded automatically so that the player could get a full set of 200. Then that opened debate about whether the star should be automatic only if clearing the level was categorically impossible, or if it was merely impractical, and what the limits to that were (there are no weapons on this map but you could technically punch those last two Cyberdemons until they explode...)

So for next time, I think my lesson here is to make sure you know what your requirements are at the start of the project and to stick with those, not allowing feature creep to set in. But of course, that's easier said than done.

## *Deadlines*

My aim with RAMP was to foster an environment where people could feel free to be creative without feeling too much pressure. Initially I was vague about the deadline, saying I would close for new entries around the start of July. Eventually I solidified this to the 6th of July so that the US mappers could use the long weekend around that time.

But there was still some confusion about what that actually meant - in my attempt to be relaxed about it, I'd said that I didn't plan to be absolutely strict about the date, and while maps should basically be finished by then, that it was still OK to make updates after that. Some people interpreted this to mean that maps should be done but adjustments were still allowable, some went to the other extreme and thought that as long as you had a placeholder slot to signal your participation you could make major updates at any time.

This meant the end of the project dragged out much longer than I had intended - it was two weeks past the intended finish date before I poked the last of the mappers into declaring their maps out of WIP state, and by the end of that period I was feeling guilty about the amount of leeway I'd given to some people to make updates when other people had got their maps in by the finish date. However, I wouldn't have felt it fair to cut people off suddenly when I hadn't initially given a cutoff date, so I feel the best way to avoid this is simply to be clearer about the deadlines next time around.

# Conclusion

You might not believe it from reading the amount of suffering and heartache above, but I loved hosting RAMP and I'll certainly bring it back as a yearly event for as long as people aren't sick of it! I'm grateful for the experience not just in my experiments with continuous integration for Doom, but for accidentally becoming a ZScript convert and learning lessons about how to handle an online community.

That community was made up of Doom mappers of all ages, tastes and levels of experience, and the variety of approaches that people took to maps was incredible - from vanilla-styled techbases and hellish caves and castles to OTEX behemoths and strange voids, from running and gunning either hell fodder or the custom Ultra Cyberdemon to a Crystal Maze-style challenge map and a bizarre rhythm game. The goal was simply to try new ideas - making a map for the first time, experimenting with a new map format, or anything that the author wanted - and to have finished with such a showcase is truly amazing. I was pleasantly surprised that RAMP got a special mention in the 2021 Cacowards<sup>12</sup> exactly because of this breadth and sense of community.

If you're thinking about running a community project of your own, I hope that this book has been helpful in showing some pitfalls and successes. I also want to recommend the excellent thread started by MFG38 on the Doomworld forums on how to host a community project<sup>13</sup> - multiple people have contributed their experiences and it has some vital details on how to set people's expectations and define limits so that your own project gets to completion instead of being forgotten about.

RAMPART is also available for use by anyone who wants it<sup>14</sup> - since RAMP I've set up three instances for other people to run their projects on, and features have been added to allow projects of types other than a hub and spoke model - it now offers more control over the layout of levels

---

<sup>12</sup> <https://www.doomworld.com/cacowards/2021/special/>

<sup>13</sup> <https://www.doomworld.com/forum/topic/121977>

<sup>14</sup> <https://github.com/davidxn/rampart>

through MAPINFO properties, and setting up specific level slots for people to upload into instead of just accepting new levels and putting them on the end. It also supports writing the project out to a WAD file, so it could theoretically even be used for vanilla projects (though I admit I haven't yet tested this). I would really like to see it used more widely because I still see so many community projects struggling to be organized exclusively through a forum thread!

Thanks to everyone who contributed to this unexpectedly massive project, and especially to those who spent time giving feedback, advice and guidance to others. It was wonderful to see people's efforts, and I'm very happy that some of these maps have already been turned into the basis for episodes or larger projects because their authors didn't want to stop. As long as the influx of new mappers continues, Doom will never die!



- DavidN