Level, Room, and Sprite Scopes

In descending order, Level -> Room -> Sprite(s). that is, levels contain rooms, and rooms contain sprites. Proper storage and management of all the data we need is critical to effective flow between these three scopes, so we will need to think carefully and plan accordingly, but seeing as this is a huge bottleneck on our project, we will need to act quickly.

To that end, I have come up with various design requirements, possible problems, and several suggested design features to handle the first 2 things.

First, some basic requirements

1. Global: the global scope just means that at some (arbitrary) scope, this thing will have to be handled
   1. Collecting keyboard/mouse/controller input and then sending hat to the player.
   2. Storing, accessing, and reading extraneous modules, test documents, and images as assets for the game
   3. Entities need to be able to create other entities (such as projectiles) and then get them to the proper scope.
2. Levels:
   1. Must be able to store multiple rooms and preserve the state of said rooms in between visits by the player
   2. Should be able to store all of the raw input assets necessary for building (and then populating) the rooms within.
   3. Should be able to store the seed data (data from which new rooms can be {procedurally} generated) and then pass that data to the room constructor to build a new room
   4. Must be able to handle transitions between rooms, which will include transferring data up and down across all three scopes.
   5. Must be able to transfer data up and down across all three scopes. (Specifically, this is probably where we will store most of the assets for building rooms and sprites, so it will have to be able to send those assets down to where they are needed.)
3. Rooms:
   1. Needs a constructor that can take in all the requisite data and then build a room (ideally in layers), store the image(s), populate that room with some number of entities, and then pass data to/from those entities to control their interactions
      1. Current implementations use ‘mapcode’ which is a list of strings that is passed in and then parsed, char by char. Each char is used as the lookup key in a dictionary of images/surfaces. The looked-up image/surface is then blitted onto the correct surface at a specific, procedurally-generated location.
      2. Mapcode may not be the best idea, as I started with the idea that by combining strings of letters would result in an easy to read format for storing not-yet-visited rooms. But upon trying to implement it, I ended up having to use so many workarounds that I may have rendered mapcode obsolete.
      3. For example: assume that you’re creating a rectangular 6x7 room consisting of plain floor tiles. The mapcode format for such a room would be: [‘ffffff’, ‘ffffff’, ‘ffffff’, ‘ffffff’, ‘ffffff’, ‘ffffff’, ‘ffffff’, ‘ffffff’] (note that the walls are assumed based on the implicit bounds created by the length of the first string and the overall list). But we could just as easily pass in a tuple ( 6 , 7 ) to do the same thing only considerably faster.
      4. To create irregularly shaped rooms using mapcode, we would pass in the above code, only w/ a special sequence of characters inserted at some point specifying that tere will be a block at this position that will be this big. For example: [‘fff<block 3x6, f>ff’, ‘ffffff’, ‘ffffff’, ‘ffffff’, ‘ffffff’, ‘ffffff’, ‘ffffff’, ‘ffffff’]. This is the same room as before, only a 3x6 solid wall/block has been inserted along the top right wall, creating an “L” shaped room. The special char sequence would be ‘<block 3x6, f>’. Parsing it would look like: parser reads ‘<’ which tells it that a special char sequence is coming next. It reads ‘block’ telling it that to spawn/create a ‘block’ type entity at current location (which is determined by its location in the (string, list) = (3, 0) ) of size 3x6. The ‘,’ (comma) tells it that this special instruction is done, so move on to the next special instruction (if there is one). In this case, there is not next instruction, so it reads in the ‘f’, which tells it to blit a floor tile at current location (under the block). Finally it reads the ‘>’ telling it the special char sequence is over.
      5. The above method sounded rational at the time, but I realized a few things:
         1. Assuming every room is a rectangle means wall positions can be assumed, and so do not need to be in mapcode.
         2. Rectangular rooms can just as easily be created with a simple tuple (x, y)
         3. Adding special block entities means that walls and irregularly shaped rooms can be easily generated/dictated by placing a spawn sequence inside the mapcode, OR by including a second (x, y) specifying the coordinates at which to spawn the entities.
         4. The majority of our tiles will be either floors or walls, so if we assume the two (which we can do by making all rooms initially rectangular) then everything else can be dictated as separate parameters (ie: pass in a tuple for size, then maybe a list containing all the entities and locations to spawn them)
         5. Keeping mapcode in its current form means that we will be working with long strands of ‘f’s repeatedly broken up by sentenes of char seqs. One of the key features of mapcode was human readability, which is severely compromised by this.
   2. Needs some level of semi-permanent/long-term storage w/I level scope
      1. This will require some cooperation with the guy who does level-scope
      2. I think just storing the already-visited levels in a List[] (as member data of the level) will suffice, but double-check this
   3. Needs some way to organize, store, and manage multiple entities and effectively check for interactions between them
      1. Best way to do this is probably thru the use of “sprite-groups” which are pre-defined collections of sprites that can easily checked for interactions, updated, and drawn
      2. Most interactions between rooms and sprites will be spawning new ones, tracking old ones, and managing current ones. I will take pains to ensure that this can be handled thru “.draw()” and “.update()” methods (both of which can be called periodically on sprite-groups
   4. Needs to effectively manage boundaries for sprites (that is, restrict entities/sprites from wandering outside of the map/current room, into ‘walls’, or anywhere else they shouldn’t be.
      1. This will require either regularly extracting or constantly tracking and updating the positions and sizes of all sprites within the room, and I’m unsure of how best to handle that. Either way, it will require some cooperation between the sprite guy (me) and the room guy.
   5. Cannot accidently create new entities if a room is re-entered (that is, if the player enters a room set to spawn 5 fish, leaves the room, and returns, the room should store and re-spawn the original 5 fish, not a new 5 fish, and not an additional 5 fish.)
4. Sprites: I (Logan/Skullking) can/will handle this. Its kinda complex, so I won’t go too into detail, as I’m the one who will be dealing with most of this (but I would be more than happy to explain literally everything I know about this, should one of you desire to know).

\*\*\*\*Notes:

* Rectangles are very useful for collision detection, positioning, drawing, and a bunch of other things. Can be used in almost all situations
* Subsurfaces are smaller sections of other surfaces. I think that we should transition most if not all of our graphics towards a surface asset sheet broken up into smaller subsurfaces model.