The Anatomy of a Design Document, Part 2: Documentation Guidelines for the Functional and Technical Specifications

by Tim Ryan

Did you ever look at one of those huge design documents that barely fit into a four-inch thick, three-ring binder? You assume that by its page count that it must be good. Well, having read some of those design*volumes* from cover to cover, I can tell you that *size does not matter*. They are often so full of ambiguous and vague fluff that it was difficult finding the pertinent information. So why does this happen? Because the authors didn't follow guidelines.

This article is part two of a two part series that provides guidelines that when followed will ensure that your design documents will be pertinent and to the point. Unlike the authors of those prodigious design volumes, I believe in breaking up the design document into the portions appropriate to the various steps in the development process – from concept and proposal to design and implementation. I covered the first two steps in part one of the article, providing guidelines for the game concept and game proposal. This part will provide guidelines for the two heaviest undertakings – the functional specification and technical specification, as well as some guidelines for the paper portion of level design.

Functional vs Technical Specifications

Traditionally in the game industry, there was only one spec. How technical it was depended on who wrote it. Any documentation the programmers wrote afterwards to really plan how they were going to implement it was informal and often remained on white-boards or notepads. Yet in order to ensure the project would proceed without hazard and on time and on budget, the documentation needed to be more technical. Such detailed technical specifications took time – time wasted if the goals and function of the product should change or fail to gain approval.

This problem was tackled as more and more seasoned programmers and managers of business software development moved into games. They brought with them new standards for documentation that helped ensure more accurate plans and less technical problems. They introduced a division in the design document between goals and method and between function and technique. They separated the design document into the functional specification and technical specification. This way, the clients, users or principal designers of the product could review the functional specification and approve the goals and functions of the proposed software – leaving the determination and documentation of the methods and technique up to the technical staff of programmers.

Therefore, the technical staff waited until the functional specification was approved and signed-off before starting on the technical specification. They worked from the functional specification alone, ignoring any design changes that occurred after sign-off unless the spec was updated and a new schedule agreed to. Thus the division saved time for the programmers and gave them more control of the schedule, while still ensuring they had a complete plan for the methods and technique for implementation.

Many companies still refer to the functional specification as the "design document" and yet also produce a technical specification. The term "functional" is a clearer term adapted

by businesses and these guidelines to clarify what is expected in the document. Here is a link to a formal definition:

http://webopedia.internet.com/Software/functional_specification.html
In short, *what goes into the game* and *what it does* is documented in the functional specification. This is often written from the perspective of the user. *How* it is implemented and *how* it performs the function is documented in the technical specification. This is often written from the system perspective. Both form important deliverable milestones in the design stage of the game development process.

Guidelines for the Functional Specification

The functional specification (or *spec* for short) outlines the features and functions of the product. The target audience is the team doing the work and those responsible for approving the game design. The functional spec is a culmination of the ideas, criticisms and discussions to this point. It fleshes out the skeleton of the vision as expressed in the game concept and game proposal. It is a springboard from which the technical specification and schedule is derived and the implementation begins.
It's important that it is all written from the user perspective. In other words, what is seen, experienced or interacted with should be the focus of the document. It's often very tempting (especially to programmers) to create something that's very system oriented. This often leads to distraction and hard-to-fathom documents. Readers are really just looking to this document to visualize what's in the game, not how it works.

The length can vary from ten pages to a few hundred, depending on the complexity of the game. You really should not aim for a page count. I've seen and written really GOOD design documents that were less than fifty pages and some that were much more. It is just important that each section under these guidelines be addressed. This will eliminate the vagaries and guesswork that comes with insufficient documentation and the apparent need to ramble-on that comes with aiming for a high page count.

The time involved in writing the functional spec is anywhere from a few days for say a puzzle game, a month for a shooter, to a few months for a complex game such as an RPG or strategy game. The amount of time spent may not be congruent to the resulting length. The discrepancy comes with deliberation time, especially if the game has any unique, unexplored qualities or if the game play is particularly deep. Of course, how efficient the principal designers are in making their decisions is of enormous impact as well, especially if everyone is particularly imaginative and passionate about the game.

For many, this functional specification is where the documentation begins. They skip the important research and review phase of the concept document and game proposal, which would otherwise help it anchor the vision and target market firmly in place. By skipping the first steps, they also put off the inevitable criticism from marketing, finance and technical staff, which leads to wasted efforts.

The game's lead designer usually produces the functional specification. It may be a compilation of other's work and hence a cooperative effort or it could simply be a matter of putting the vision of the producer on paper. Sometimes the producer will produce the document him/herself; which is ideal for assuring that the vision expressed is indeed what the producer desires. Like the game of telephone, sometimes the message gets altered when it goes from the lead visionary to the author. Whatever the process and whoever the author, it's important that the producer and lead designer totally agree with everything expressed in this document. They cannot be preaching one thing and documenting another or the documents will be ignored and serve no purpose.

I've never seen a design that didn't undergo some changes during implementation, but the process of communication has to be expressed through the specification, even if it requires updates or an addendum. Some changes need to be fast and furious due to time constraints so the documentation may be light. So, even if it's an electronic memorandum or notes on a piece of paper, be sure to distribute these and attach them to all future copies of the specification. If the vision of the game changes, however, it's best to start from the beginning with a new concept document and proposal. The clarity the updated documentation brings will save time in the long run.

Unlike the game concept and proposal, the functional specification is not a *selling* document. It merely breaks down and elaborates on the vision in very clear terms understandable by every reader. It can be a little boring or dry as the necessary details are filled in. I'd recommend putting in summary level paragraphs at the start of every section, so that readers can get the gist without skipping any sections or losing any confidence in the thoroughness of the specification. Why do I recommend this? Well, the question should really be "Why do some people never find the time to thoroughly read the project specifications?" While your company's managers and team members might not fail in this regard, there's always a third party, like the publisher or contractor.
On the other hand, this document cannot be technically explicit, as its readers are mostly non-programmers. If you find yourself getting technical, stop. That's what the subsequent technical specification is for. Besides, getting technical with a bunch of non-technical readers can make their eyes glaze over or open up a can-of-worms. You don't want to give them an invitation to stick their nose into something they don't necessarily understand and really shouldn't care about. Likewise, you or the other authors may be non-technical, and you shouldn't be dictating to the programmers how they accomplish what you want them to do. Let them determine that when they write the technical specification. This document is purely for the communication and approval of *what* goes into the product as opposed to *how*to accomplish it. Limit descriptions of how something should be accomplished to those areas that are you believe are really important that it work a certain way. For example, you would not indicate what variables to use and how to use them to simulate a law of physics; however, you might want to indicate the factors involved in the physics equation. Similarly, telling a programmer how to define his data structures and objects is a bad idea, but proposing the interface for data entry and the delineation of data is certainly within the confines of *function*.
The functional specification can be broken down into a few major sections:

- Game Mechanics
- User Interface
- Art and Video
- Sound and Music
- Story (if applicable)
- Level Requirements

Game Mechanics
The game mechanics describe the game play in detailed terms, starting with the vision of the core game play, followed by the *game flow*, which traces the player activity in a typical game. The rest is all the infinite details.
Core Game Play: In a few paragraphs describe the essence of the game. These few words are the seeds from which the design should grow. Planted in the fertile soil of a known market, they should establish roots that anchor the vision firmly in place and help ensure a successful game. This is similar to the description section in the game concept, except that it's non-narrative, and usually expressed clearest in bullet points, though this could vary depending on the type of game.

Game Flow: Trace the typical flow of game play with a detailed description of player activity, paying close attention to the progression of challenge and entertainment. If the core game play is the root of a tree, the game flow is the trunk and the branches. All

activity should actualize and extend from the core game play. Be specific about what the player does, though try to use terms like "shoot", "command", "select" and "move" rather than "click", "press" and "drag". This keeps the description distinct from how the actual GUI will work, which is likely to change. Refer readers to specific pages in the User Interface section when you first mention a GUI element such as a screen or window or command bar.

Characters / Units (if applicable): These are the actors in the game controlled by the players or the AI. This should include a brief description and any applicable statistics. Statistics should be on a rating scale i.e. A to Z or Low to High, so that it's clear where units stand in relation to each other in broad terms. It's a waste of time plugging in the actual numbers until the programmers have written the technical specification and created an environment for you to experiment with the numbers. Special talents or abilities beyond the statistics should be listed and briefly described, but if they are complex, they should be expanded upon in the game play Elements section.

Game Play Elements: This is a functional description of all elements that the player (or characters/units) can engage, acquire or otherwise interactive with. These are such things as weapons, buildings, switches, elevators, traps, items, spells, power-ups, and special talents. Write a paragraph at the start of each category describing how these elements are introduced and interacted with.

Game Physics and Statistics: Break out how the physics of the game should function, i.e. movement, collision, combat etc., separating each into subsections. Describe the look and feel and how they might vary based on statistics assignable to the characters, units and game play elements. Indicate the statistics required to make them work. Get feedback from the programmers as you write this, as how the game handles the physics and the quantity of the statistics will severely impact performance issues.

This can get a little dry, but avoid getting too technical. Avoid using actual numbers or programming terms. These will come later in the technical specification, written by the programmers who will want to do things their way (usually the right way). Just tell them what you want to accomplish. For example: "The units should slow down when going up hill and speed up when going down, unless they are a hover or flying vehicle. How much they are affected should be a factor of their climbing and acceleration statistic as well as the angle of the incline." You would not tell the programmers what math to use to adjust the speed. Assuming you are not a programmer yourself, they're just better at that than you.

Artificial Intelligence (if applicable): Describe the desired behavior and accessibility of the AI in the game. This includes movement (path finding), reactions and triggers, target selection and other combat decisions such as range and positioning, and interaction with game play elements. Describe the avenue through which the AI should be controlled by the level designers, i.e. using .INI files, #include files of game stats or C-code, proprietary AI scripts, etc.

Multiplayer (if applicable): Indicate the methods of multi-player play (i.e. head-to-head, cooperative vs. AI, teams, every man for himself, hotseat) and how many players it will support on the various networking methods. Describe how multi-player differs from solo-play in game flow, characters/units, game play elements and AI.

User Interface

The interface changes so very often that it almost seems pointless to document it; however, it's got to start somewhere. It's structured here to minimize the impact of

changes. It's starts with a flowchart of the screen and window navigation, then breaks down the functional requirements of all the screens and windows. That done, the GUI artist is free to do what he or she feels is right as long as it meets the requirements. To get him or her started you should provide mock-ups. This often is to the designer's benefit to think everything through. Then follow up with a description of all the GUI objects that need to be programmed to make all the screens work.

Flowchart: This charts the navigation through the various screens and windows. Use VISIO or similar flowcharting tool to connect labeled and numbered boxes together, representing screens, windows, menus, etc. On the corner of each sheet, put a numbered list of all the items for easy referencing and ease of defining tasks for the programmers.

Functional Requirements: This functional breakdown of every screen, window and menu lists the user actions and the desired results and may include diagrams and mock-ups. While the specific interaction (buttons, hotspots, clicks, drags and resulting animations) can be listed, it's often best to keep this separate from the list of functional requirements as these can evolve during implementation. Of course if it's just easier to think in terms of clicking a button or it's really important that something work a certain way, then by all means get specific about the method of interaction.

Mockups: Create a mock-up for all the screens, windows and menus. This may end up getting ignored, but it's a good starting point for the artists if they have no idea what else they may want to do. Don't waste your time creating anything really pretty. Just create simple line drawings with text labels. Color can be very distracting if it's bad, but if it's important, go ahead. Some drawing programs have templates that make creating mock-ups very quick and easy.

GUI Objects: These are the basic building blocks used to create all the screens, windows and menus. This should not include the items seen in the main view portal, as these are covered in the art list in the next section. The GUI objects are primarily listed here for the programmers to know what pieces they'll need to code and have for putting together the screens. You should explain in detail how each is interacted with and how they behave. It may seem a bit obvious and not worth documenting, but it really helps when drafting together the technical spec and schedule to know exactly everything the game will need.

For some games, this can be a very quick list to put together – buttons, icons, pointers, sliders, HUD displays etc. But it's much more complicated in games where the interface is at all different. However, keep in mind that the methods of interaction are not all that different. A button is still a button, even if it's clicking on a gorgon's head instead of a gray rectangle.

Art and Video

This should be the definitive list for all the art and video in the game. We all know how things creep up, though, so add a couple of placeholder references for art to be named later, like mission specific art and art for marketing materials, demos, web pages, manual and packaging.

Overall Goals: This is where you should spell out the motifs, characteristics, style, mood, colors etc. that make up the goals for the art. Gather consensus with the lead artists and art director and make sure they see eye to eye with the project's director or

producer. Doing so now will save a lot of time later if they end up redoing everything because the goals were never clearly defined.

2D Art & Animation: This is really just a huge list that can be thrown into the art schedule. It can also include descriptions if needed. Some art isn't self-explanatory, and other may involve specific needs from a design standpoint. Be sure to explain it all. Break your art down into sections. The lead artist may have some particular way he or she would like you to do that. I'll list the typical section and their contents. Read them all to be sure you don't forget anything.

- **GUI:** Screens, windows, pointers, markers, icons, buttons, menus, shell etc.
- **Marketing and Packaging Art:** You might as well list it here and the schedule, because they'll ask for it. This includes web page art, sell sheet design, demo splash screens, magazine adds, press art, the box and manual.
- **Terrain:** Environment art like tiles, textures, terrain objects, backgrounds
- **Game Play Elements:** Player and enemy animations (sprites or models), game play structures and interactive objects, weapons, power-ups, etc. Don't forget damage states.
- **Special Effects:** Salvo, explosions, sparks, footprints, blood spots, debris, wreckage

3D Art & Animation: This serves the same purpose and has the same requirement of the 2D Art list above. The difference may be in how the work may be divided. Art teams like to divide 3D art task lists into models, textures, animations and special effects, as they usually divide the tasks this way to maximize talent and skill and maintain consistency.

Cinematics: These are the 2D or 3D scenes often shown as an intro, between missions, and at the end of the game. These should be scripted like a film script as separate documents. This, however, is production work. For the purposes of the functional spec, just list them here with the general purpose, content and target length. If any video is involved, list it in the following subsection.

Video: Unless you are doing an FMV (full motion video) game, this subsection is pretty light. If you have any video in your GUI for say pilot messages, break it down here. All video tasks will require scripting, but that is production work. List the general purpose, expected length, and general content like number of actors and set design, even if it ends up being blue-screened into a 3D rendered background.

Sound and Music

Overall Goals: Stress the aesthetic and technical goals for the sound and music. Describe the themes or moods you want. Name existing games or films as examples to aspire to. Issue technical edicts and editing objectives, such as sampling rates, disk space, music formats, and transition methods.

Sound FX: List all the sound FX required in the game and where they will be used. Include the intended filenames, but be sure to consult with the sound programmer and sound technician (or composer) on the file naming convention. This makes it easier for people to find the sound FX and fold them into the game.

Don't forget about all the areas that sound FX may be used. You don't want to overlook anything and throw off the schedule. Go through all the game elements and your art lists to see if there should be some sound associated with them. Here are some to consider:

- **GUI:** Button clicks, window opening, command acknowledgments
- **Special Effects:** Weapons fire, explosions, radar beeping
- **Units/Characters:** Voice recordings, radio chatter, stomping, collisions
- **Game Play Elements:** Pick-up jingle, alerts, ambient sounds
- **Terrain (Environment):** Birds, jungle sounds, crickets, creaks
- **Motion:** Wind, footfalls, creaking floors, wading, puddle stepping

Music: List all the music required in the game and where it will be used. Describe the mood and other subtleties. Music will often reuse the same themes and melodies. Mention where these themes should be reused. Consult the composer on this.

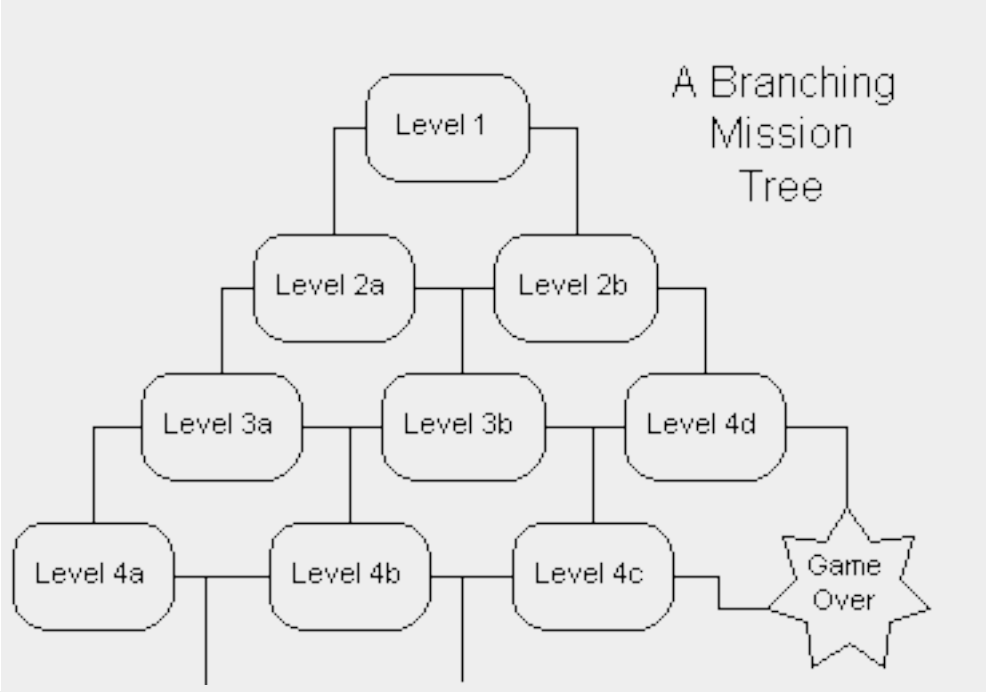
- **Event Jingles:** Success/failure/death/victory/discovery etc.
- **Shell Screen:** Mood setting for title screens, credits, end game
- **Level Theme:** Level specific music (designers choose the theme)
- **Situations:** Sets the mood for situations (lurking danger, combat, discovery)
- **Cinematic Soundtracks**

Story (if applicable)

Write the synopsis of the story told by the game. Include the back-story and detailed character descriptions if it helps. Indicate the game text and dialogue requirements so they can be added to the schedule. Some game designs focus so much on this that they overlook everything else that should be in the spec. Telling a story is not the focus of most games. Of course, if you are doing an adventure game, it is extremely important. Expand and organize this section as is necessary to tell the story.

Level Requirements

Level Diagram: Whether this is a linear campaign, a branching mission tree, or a world-hopping free-for-all, this diagram should be the backbone upon which all the levels are built. A diagram isn't necessary if the structure is so simple that a list would suffice. The following is an example of a typical success/fail branching mission tree. Of course this will vary greatly for each game. The important thing is that it just presents a road map for the level designers and for the readers.



Asset Revelation Schedule: This should be a table or spreadsheet of what level the game's assets are to be revealed to the player for the first time. There should be a row for each level and a column for each general type of asset. Assets include power-ups,

weapons, enemy types, tricks, traps, objective types, challenges, buildings and all the other game play elements. The asset revelation schedule ensures that assets, the things that keep the players looking forward to the next level, are properly spaced and not over or under used.

If it's important to the game that certain assets stop being used, then the schedule might be better drawn as a Gannt chart with lines indicating the availability of assets. This gives the level designers a guide to what assets they have to work with so they don't ruin their level or anyone else's.

Level Design Seeds: These are the seeds for the detailed paper designs to follow. Detailed paper designs at this point are less legitimate and unlikely to survive intact. Designs created after the designers have had time to experiment with the tools and develop the first playable level are much more likely to succeed. It's best to just plant the seeds for each level with a description of the goals and game play and where it ties into the story (if applicable). A thumbnail sketch is optional, but very helpful if the designer already has a clear idea of what he or she wants. Be sure to list any specific requirements for the level, such as terrain, objectives, the revelation of new assets, and target difficulty level.

Common Mistakes

Here are some common mistakes to look out for:

- **Insufficient details:** The descriptions need to be specific enough to convey intent and function. Avoid using vague terms unless you follow up with specifics.
- **Patronizing material:** You wouldn't give a chef a recipe that told him how to make a marinara sauce, so you don't tell artists how to manage their 256 color palette or programmers how to define a particular data structure. Just list the facts important to the vision. Not only does it waste their time (and annoy them), but it wastes the writers' time. Such details are more appropriate for the technical specification anyway, which is written by the programmers.
- **Ambiguous or contradictory material:** Watch for this. It clouds the vision, creates misunderstandings, and invalidates the functional specification.
- **The Design Document from Hell:** Nothing stupid, nothing ambiguous, nothing lacking – it just is too damn much. Try to keep a mental total of how long the design is going to take to implement when fleshing out the specification. Cut extraneous, non-essential features and save them for the sequel; or be prepared to argue the merits of keeping the features and extending the ship date.
- **Getting too personal with the design:** You are not your work. Your personal boundaries should not include the design. As I have stressed throughout this document, game design is a collaborative process. While you want people to take ownership and responsibility for their work, the functional specification should have joint ownership. This keeps people from feeling isolated and more a part of the process, and it makes the documents feel less like marching orders and more like a plan. The team members are also much more likely to read something that they helped put together. Criticism is then aimed at the design not the documentors who put is all together; thus making the team more comfortable and productive in offering their criticism.
- **Wandering vision:** This may happen as you write the functional spec. Even with a good concept document and proposal championing the vision, there's still some room for interpretation. Creative folks have a wandering imagination and may be influenced strongly by whatever game they may be playing at the moment.

Guidelines for the Technical Specification

While the functional specification explains *what* is going into the product, the technical specification explains *how*. The technical specification (or *tech spec*) is a working blueprint for the game. It turns conjecture into reality by forcing the programmers to think through how the game will be implemented, by reducing implementation/integration headaches, and by delineating the program areas and tasks for the schedule.
Many companies will skip this step, as it is time consuming and seemingly benefits only the programmers. However, time spent working on a tech spec is less than the time lost from pitfalls that come with not writing one. The primary author is the lead programmer or technical director, though it is often more timely and useful if the programmers responsible for implementing the various program areas be responsible for documenting them. In its compiled form, it should present a plan that any programmer can understand and work from.

The target audience is the lead programmer on the project and the technical director of the company. Therefore it will generally be written from the system perspective as opposed to the user perspective. It will be boring and Greek to the producer and any other non-technical readers. By asking for one, the producer is just making sure the technical staff thinks everything through, even if he or she doesn't understand it. To the lead programmer, it's a way of organizing his or her thoughts and creating an accurate picture of the work involved. The process of writing it will flag any of the uncertainties on the programming side and any of the holes, ambiguities or absurdities in the functional spec.

Many good technical specifications vary from the form described here. This form mirrors the functional specification to ensure that all areas of the functional specification are covered. Sometimes it's easier for a team writing this spec to organize it differently, if only because they are splitting the work differently or because of the organization of the underlying system. If they do, I'd recommend going through every line of the functional specification and do a correlation with a highlighter to make sure nothing has been overlooked. An overlooked detail can lead to undesirable results in the product, project and team dynamic.

These guidelines will not tell you how to implement your game. It's assumed that you are a technically competent, experienced game programmer. An inexperienced or untrained game programmer should not attempt this task. These guidelines are the result of what I've come to expect in a good technical specification, though I certainly couldn't tell you how to program your game. These guidelines force you to define the most common elements one finds in all games. Some may not be applicable, but each should be considered carefully. It may spark a question you haven't asked yourself yet; which is sort of the whole point of writing this spec.

Game Mechanics

This is certainly the bulk of the document. Right away you'll see that any attempt to match up specific subsections with the game mechanics section of the functional spec is totally ludicrous. The perspective must be from the system out as opposed to the designers' or users' perspective. This starts with the hardware platform and the operating system, the use of externally provided code objects (DLLs, EXEs, drivers), and the delineation of internally generated code objects (if any). Then it breaks down the specific mechanics of game code stemming from the control loop.

Platform and OS: Indicate the hardware platform and the operating system and the versions supported. For PC/Mac games, mention the minimum system requirements and the target machine. If distributed on something other than a CD like a cartridge, indicate the target ROM.

External Code: Describe the source and purpose of all the code used but not developed by the project team. This includes OS code and preprocessing tools of the various game platforms, drivers and code libraries like DirectX, any acquired 3D API, or any other off-the-shelf solution.

Code Objects: Break down the purpose and scope of the various code objects coded, compiled and built into the EXE. If any out-of-process or in-process code libraries (DLLs) are used, break them down as well, but be sure to explain the use of object instancing and their persistence (like Direct Draw objects).

Control Loop: Every game has one. Be specific about how control is transferred from the start-up code to the shell and down into the main game code. Spell out the names of the functions in the core loop and what they will do, like the collision, movement and rendering routines. Explain the use of multi-threading, drivers, DLLs and memory management. Of course further details on the likes of multi-threading and memory management will be covered in the areas that they will be used most, like the rendering or sprite engine, sound code and AI.

This subsection summarizes the system and underlying framework that supports the core game play and game flow described in the functional specification.

Game Object Data: Read carefully over the functional spec at all the character/unit descriptions and game play elements. Then list and formulate all the data structures and their identifiers that are required to support the described attributes, functions and behaviors. To a certain extent, these will not be complete until the game physics and statistics and AI subsections are completely thought through and documented. Add statistics for user interface or any other area of the game that have unit or game play object specific data (i.e. icons, HUD displays, animation or special effect references, etc.).

If using object oriented programming methods, show the class inheritance tree and each class' interface properties and functions. Describe the use of collections. Identify any variables that could possibly be made into global variables to increase performance, such as any objects variables that may be referenced multiple times during critical game routines such as collision, movement or rendering. Again, I'm not telling you how to program your game. I'm just trying to get you thinking about common technical issues, specifically in regard to optimizing data structures for neatness, versatility or speed.

Data Flow: Explain how data is stored, loaded, transferred, processed, saved and restored. While references should be made to data entry or processing tools, separate functional and technical specifications should be made for any complex or user intensive tools.

Game Physics and Statistics: This is the nitty gritty – movement, collision, combat – and probably the most fun to document and implement. However, it can also be the code that gets altered more than any other part of the program. Designers like to change things. It's often only after they can play it for a while before they can really decide what is right. For this reason, you should plan to implement things as modular and flexible as

possible. Put all the factors that control behavior into data files read at run-time, so the designers can change and balance things at their leisure without involving coding changes and new builds. The specification should clearly identify the modularity and divisions between code and the data that controls it.

Define each function or procedure. Describe its purpose. Define what statistics control its behavior (constants, variables etc.) and how they can be modified. Include the function prototype listing all the parameters. If using function pointers and function overloading, specify where the different versions of the function will be used. For example, you may have multiple functions that handle movement for the various unit types – one for land movement, one for air, one for water, etc. Briefly describe how the function will work. For complex functions, use pseudo code to specify exactly how you will code it. This is especially important for CPU intensive functions that do a lot of number crunching or are just called very often. Think about how they can be optimized to increase performance. Perhaps bit-shifting or macros could speed things up.

Artificial Intelligence: This often grows to a major section unto itself and is then scaled back when the schedule dictates the necessity to keep it simple. This shows a growing enthusiasm for complex AI, but a lack of time and resources to make AI anything more than *simulated* intelligence or scripted behaviors. Be mindful of this when you design the AI scheme. Try to accomplish the behaviors and decision making described in the functional specification without adding a huge layer of unnoticed and therefore unappreciated realism to the process. The basic rule of production applies here. If something that costs less and takes less time to build does the job, then don't spend more time and money creating something else.
Of course, there are exceptions that should be mentioned. Sometimes something might take longer to build, but it saves the designers a lot of time working on their levels. Also, creating something more flexible or powerful may make it a valuable asset to the company for other projects or just make it more capable of handling design changes should they occur. Discuss these with your producer and director of development before making a decision.

Be sure to include the methods of manipulating the AI as dictated by the functional spec, i.e. whether it's data driven or embedded into compiled code, and whether it's a scripted language or a fixed set of variables or a combination of both.

AI should include path finding, target selection, tests and events to attach reactionary behaviors to, and other decisions made by characters, units or intelligent game elements involving game situations and unit statistics.

DO NOT include the actual scripts or data driving the AI. That's production work. Merely be specific enough to explain how the decisions and behaviors will be derived. Break down the statistics used to control the behavior.

Multiplayer: It's extremely important that the implementation plan is reviewed from a multiplayer perspective. This subsection should break down all the multiplayer considerations in game mechanics and all the multiplayer specific requirements specified in the functional spec.

Multiplayer over multiple PCs (as opposed to console sharing or hotseat) has a lot of unique requirements that should be addressed. What connection methods and protocols are supported? Is it client-server or peer-to-peer? What are the packet sizes and how often are they sent? What is the structure of the packet? How are missed packets and

latency issues handled? What messages are broadcast and what are sent to specific hosts? How many different messages are there and when are they used?

User Interface

Look and feel is one area of the design that undergoes the most changes during development. Therefore, its necessary that the programming for the GUI be as flexible as possible, separating game purpose from GUI function, so that changes that occur to the user interaction methods will not affect other areas of the game or require significant reprogramming. Create a variety of GUI objects (controls) using inheritance to maintain a consistent code interface to the events and the values. This way a slider bar can be exchanged with a text box or radial buttons with little or no changes to the calling functions. Assume that any of the GUI objects can be exchanged at any point in the project.

To this end, your documentation should be flexible and generic. While it should break down the GUI into the screens, windows and menus, it should not go any further into the specific interaction. Instead, document how the various GUI objects will work, wherever they are used.

Make references to functions in the game mechanics documented in the previous section, but anything that's interface related should go here. Explanation of the drawing and clipping routines of the graphics engine should be left for the Art and Video section, but certainly they should be referenced here in terms of view ports and HUD attachments and anything the player can interact with.

Document the names for any of the global variables, constants, macros, function names or interface properties, so that other programmers can refer to the documentation without having to dig through code. This also avoids replication and inconsistency and increases clarity.

Game Shell: List all the screens that make up the *game shell* - all the screens and windows other than the main play screens. These are derived from the flowchart in the functional specification, but may include some additional screens that the lead designer may have overlooked or brushed over (like installation or setup screens). Each item listed should be its own subsection with a description of its purpose, its scope (i.e. before or after level specific data is loaded), the pertinent values it will be accessing and setting, and what functions it will call.
Main Play Screen(s): These are the one or more screens in which the core of the game is played. Though many people think from the GUI perspective down to the complexities of what's under the hood, this should be written from the low-level mechanics perspective (the engine and rotors) out to the GUI (the hood and the dash). This keeps it consistent even if the outward appearance of the GUI should change.

Art and Video

While this section in the functional spec pretty much just listed the art and video, the technical spec has to explain how the art and video will be stored, loaded, processed and displayed in the game. This includes the animation system, whether it's 2D or 3D, and the video decompression and streaming system. Of course some of these might be off the shelf solutions, especially the video code. But all the interfacing should be mentioned here.

Graphics Engine: Whether you are using sprites, voxels or 3D-polygon rendering or a combination, break down their functions in very specific detail. While it's only 2 sentences of description here, it will likely prove to be a very meaty piece of the spec. Describe areas like view ports, clipping, special effects, and the connection to the collision and movement functions described in the game mechanics.

Artist Instructions: Break out the details important to the artists, like resolutions, bit depth, palettes, file formats, compression, configuration file definitions and any other data the artists need to define to fold in the art. Consider what tools can be created to streamline the art pipeline, and indicate their specifications here or create separate specifications for the more complex or user intensive tools.

Sound and Music

Describe how sound will be loaded and played. Be specific about the use of mixing, DMA, multiple channels, 3D sound, and methods of determining priority. If using third party drivers, describe their interface and purpose. Be sure to address all of the requirements specified in the functional spec.

Sound Engineering Instructions: Break out the details important to the sound engineers and composers, like sample rates, the use of multiple channels, 3D sound definitions, sample length etc. If using MIDI, indicate the version to use and the number and type of instruments that can be used and possibly stored. Indicate the data path and file requirements including any specific configuration files that need to be created. Consider what tools can be created to streamline the sound pipeline, and indicate their specifications here or create separate specifications for the more complex or user intensive tools.

Level Specific Code

Based on the level design seeds in the functional specification, describe how code specific to those levels will be implemented and how it will accomplish the desired effect. Also describe how any other level specific code can be interfaced to the game code should the need arise to add more. In general, you should try to make any of the level specific code as generic and as flexible as possible so that it may be freely used to accommodate similar needs for other levels or new ideas.

Common Mistakes

Here are some common mistakes to look out for:

- **Hand waving:** It's very tempting to just list the functions and not fill in all the details that force you to really plan how you are going to implement them. Sometimes they are just glossed over, but really the hand waving should end with the functional spec. This spec is supposed to force the programmers to really think everything through ahead of time. How else are they going to estimate the task time correctly?
- While it can be very effective to assign portions of the technical spec to the individual programmers responsible for implementing it, it's not always in the best interest of the game or indeed the programmer to do so without some supervision. An entry-level programmer should get some guidance, and all the programmers should discuss and critique their documentation before it gets all

folded together. Some companies have regular code reviews where programmers critique each other's work. That should start even sooner during the design phase.

Guidelines for Paper Level Designs

The designers should do paper versions of level designs before they begin creating the levels in the editor. Ideally, the designers will be familiar with the design palette, the level editor and game engine capabilities before they get started. Paper level designs are creating during the implementation phase, though they are based off of level design *seeds* expressed in the functional specification. These *seeds* are the core idea for the level and/or the basic requirements that may indicate what new assets are being introduced or what to limit the design to. It's best not to do all of the paper designs at once, either, as the designers usually learn a lot while implementing each new level.
For some reason, producers often expect the cart to come before the horse, so before serious level design begins, push for a playable, prototype level to be created first. It's often a milestone unto itself that ensures that the tools and game mechanics are working well enough to develop levels. It should also serve as a guide to what can be accomplished with the editor and engine and epitomize the vision for level design.

Following the first playable mission, level design can start in earnest. Yet even here, documentation plays an important role in saving time and ensuring quality through meticulous planning and the critical process. The process of level design that works:

Step 1: Thumbnail & Discussion

The level designer conceives of a level layout that meets the requirements laid out in the functional specification and asset revelation schedule. He or she then produces a thumbnail sketch and discusses the concept with the lead designer. The thumbnail could be on a white board or a note pad. It is a visual aid in the discussion. It does not need to convey the entire idea or all the details for the level, as these often evolve during the discussion or get tossed out altogether.

The benefit of doing a thumbnail sketch and discussion rather than forcing a designer to first think everything through and document it is that it saves time. A senior or lead designer can in a matter of minutes determine whether a proposed level design has merit and give valuable advice that can drastically alter the design. A fully detailed and documented paper version can take days or even a week to put together. Depending on the skill of the designer, a designer might get sent back to the drawing board many times. This is especially true near the beginning of the project, when the designer is still learning what the lead designer wants, and near the end of the project, when original, compelling level concepts are harder to come by.

Step 2: Detailed Paper Version

With an approved thumbnail and level concept, the level designer can work on a detailed paper version of the level design. The layout (or map) of the level should be much more detailed than the sketch and should be drawn to scale. This is best done on a large sheet of graph paper using colored pencils. Information about objectives, behaviors, buildings, enemies, events, locations etc. should either appear on the map or on a separate document with reference points on the map. Any mission specific art or code should also be listed. This amount of detail can take a few days or as long as a week to draw and document, but it saves a lot of time that would otherwise be spent "searching" or "redesigning" in the actual editor.

When completed, the lead designer, producer and any other principal decision-makers should subject the paper design to an approval process. They may approve it, throw in some changes, or kill the level right then and there.

It's also important that someone technical, preferably a senior programmer, review the paper design from a technical standpoint. This gives the programmers a heads up on what the level designers are going to attempt to do with the tools and graphics engine. They might add some features to the tools or make some code adjustments to make the level possible or just easier to implement. They may also vote to eliminate or alter any level designs that may break the game or are similarly unfeasible.

It's often very tempting to skip this step and jump right into the editor as it's often faster to just build a prototype of the level than to write up the paper version. This is especially tempting in tight schedule situations. Yet, it's these tight schedules that make documentation that much more important, because it means there's even more reason to get it right the first time and reduce the number of surprises and time to redo the work. The benefit of a detailed paper version is that it forces a designer to think everything through and express the fun and challenges before he or she implements it. It also ensures that the details that may involve more tasks for programmers, artists and sound technicians get documented and scheduled for completion before the designer begins working on the level.

This article is focused on documentation, but for completeness to this section and to this process, here are the remaining steps to level design as I see them:

Step 3: Creating the Core of the Level

The designers should establish the core game play of the level using broad strokes. They should get it to the point that it gives them the fun and challenge they envisioned in the paper design. The designer should then get feedback from the lead designer and producer, who will determine whether the level has merit or not. It may indeed prove impossible to accomplish what the paper design suggested, or it may prove to not be as fun as was expected. This is simply a review point in the level design that saves the designer time should drastic changes need to be made or the level dropped entirely.

Step 4: Filling in the Finer Details

Once the core game play of the level is established, everything else should just make it better. These are all the things that establish the setting, flesh out the level, and liven up the fun by providing more options, solutions, or surprises.

Often new art or code assets may seem appropriate, so be sure the designers find out they can get them before putting placeholders in. Then update the paper design and task lists.

Step 5: Play Test

Have the designers play their levels and get as much feedback as possible. Again, documentation plays a role here. Be sure they keep track of all their bugs, feedback and tasks with at least a notebook and pencil. It's very easy to lose track of issues at times (not to mention sheets of paper), so a centralized database with level specific issues and feedback is ideal.

For further guidelines for level design, I encourage you to read my articles on level design that present a background to level design and some rules to design by. The first part "Level Design Theory" can be foundhere. The second part "Rules to Design By and Parting Advice" is available here.

Documentation Milestones and the Development Schedule

Below is a list of the typical milestones in a schedule and where the documents described in this series serve as deliverable items. Following each milestone would be a review of that milestone, which would require approval to go on to the next. As the production schedule isn't due until after the bulk of the documentation, then there shouldn't be an impact on the schedule if time needs to be spent going back to the drawing board and creating specifications that everyone can agree with. It's a recipe for disaster to race into production with iffy design documents just because of the urgency to meet an arbitrary ship date. In the end, it usually doesn't save you any time, and in fact often leads to wasted efforts and significant delays.

Conceptual Phase

- Document: Game Concept
- Document: Game Proposal

Design Phase

- Document: Functional Specification
- Document: Technical Specification
- Documents: Tool Specifications (if applicable)

Production Phase (sometimes called Implementation Phase)

- Production Schedule
- Technology and Art Demo
- First Playable Level
- Documents: Paper Level Designs (not always a deliverable)
- Alpha - Functionally Complete

Testing Phase (Quality Assurance)

- Beta - First Potential Code Release
- Gold Master - Code Release

There could be more milestones, as it's often necessary for publishers to have some means of determining and ensuring that progress is being made. Sometimes there are arbitrary monthly milestones for particular art, code, and design. The ones suggested here are the most common and have the greatest significance.

Dealing with Change

It can be a beautiful thing to witness a project run smoothly following these guidelines, but what typically happens is some change to the vision due to inspiration or market trends. It's very easy to slip into design-on-the-fly mode to try to adapt to the new vision without impacting the schedule. Of course it inevitably does anyway, because design-on-the-fly has its dangers. In these cases, the only way to make sure this doesn't happen is to be adamant about the guidelines and the procedures they promote. This is easier if it's spelled out in the contract. Don't make any changes to the game without it going through the documentation. A change to the functional specification potentially invalidates the technical specification and subsequent schedule. It's certainly grounds for

reassessing the schedule. It should be very clear to the principle designers who may want these changes that when they sign-off on the functional specification and deliver it, they do so with no expectations on being able to make changes. Then, if they really want the change, the impact can be lessened with a period of updating the documentation and a reassessment of the schedule. The threat of being stuck with what you got or being late is certainly compunction enough to put as much forethought as possible into the design and produce the best possible documentation. The guidelines presented in this series of articles should help you do just that.

For more details on how to deal with change, I encourage you to read my article "Controlling Chaos in the Development Process" published here.

This concludes part 2 of a 2 part series of articles on the anatomy of a design document.