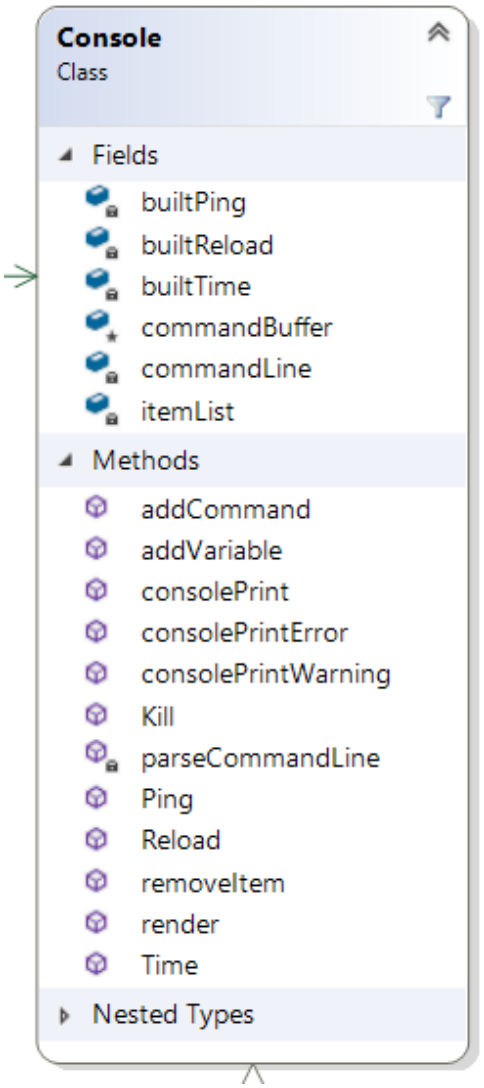
Developer Console



Background Research

Elevator: As part of the Sly Engine I am looking to implement a Developer Console that will give gameplay programmers the ability to run commands in real-time.

Inspiration: Primarily I have drawn inspiration from the Source Engine while exploring additional features and implementations of Nexuiz, GoldSrc (original source engine), and the grandfather Quake Console.

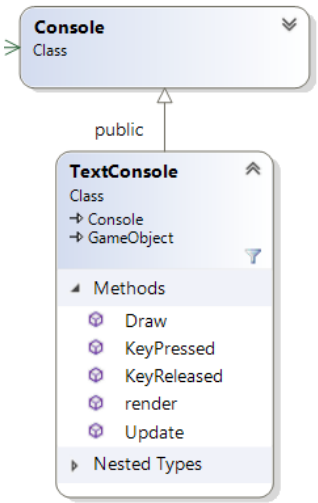
Goals of Sly Console

There are two features that I would like the console to have:

- Built-in Commands: The console will provide a handful of commands that the user can run. These commands are meant to control engine level features in real-time. See examples of functions here.
- User Commands: The system will provide a method in which the user can pass commands to the console. These commands are meant to tweak gameplay specific features in real-time.

Keyboard to Screen

Input:

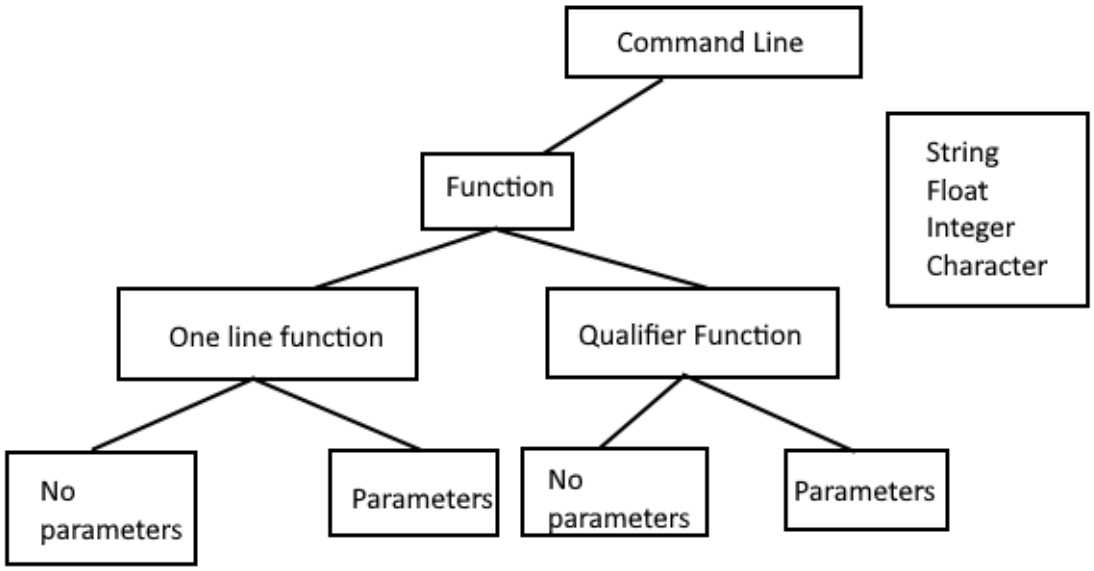


- Console Game Object: We need the console to read keyboard input every frame so that the user can type in commands. One way that I can implement this is to create a listener object that will scan the keyboard every frame. An alternative to creating a separate listener would be to have an engine managed game object. Said game object would have all of the keyboard keys (that we have fonts for) registered.

  Between these two options, I have chosen the latter implementation to focus my efforts on other components of the console. The keyboard input for the console is handled by a console game object that the console manages.
- Storage: As keys are pressed down and sent to the game object, the game object needs to handle the input. The goal here is to eventually have the text the user types in parsed by the console. Thus, each key the user types in is passed to the console to be stored in a buffer. This buffer can be as simple as a string or as complex as a homebrewed list of exactly 80 characters to prevent overflow. For simplicities sake, I chose appending each letter the user types into a string. When the user presses the backspace key, we want to pop letters off of the back of this string and when they press enter, we want the string to be parsed.

Parse:

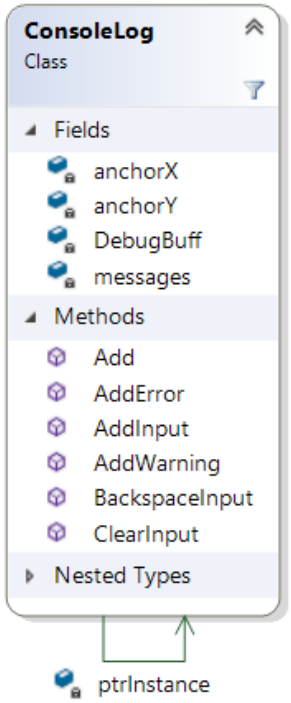
- Description: The text that the user types in is sent to the console through the game object listener to a string. This string is sent through the parser which identifies the first argument of the string. If the argument matches the name of a function/variable stored in the system, it will begin function execution or variable printing.
- Variable Types: The user may enter either a function or a variable into the console. When parsing the input, we break down the types into either functions or variables. See below for a rough overview of the parsing types and image "SendingCommands.png" located in this file's directory for more info on these types.



Render:

- Description: One all of the text has been parsed and a function/variable has been requested, the final step is to show any messages that need to be communicated to the user. The logger will always echo back the command the user typed in so that the user knows what was typed in and any additional text is displayed below the echo message. The question is, how do we display this text?

- Console Logger: We need a way to be able to display text on the screen. We could use an existing system to implement this called the ScreenLog tool, however this tool does not behave like a normal text console. Instead, we will use the ScreenLog tool as a base for a new Console Logging system. This console logging system utilizes sprite fonts to display text on screen. The user may request text to be displayed in this system through commands similar to "ConsoleLogger::print("x"). The console logger will be a singleton to avoid duplication of loggers and it will contain two separate buffers:

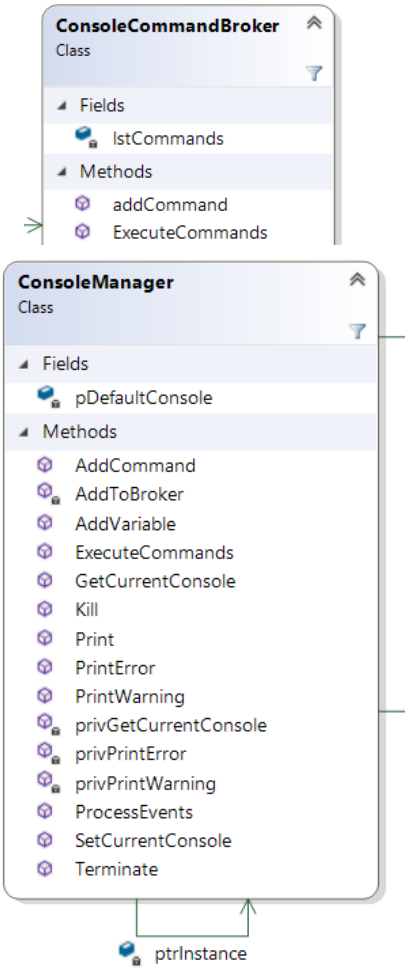


    - Command Buffer: The console buffer is essentially a screen log panel in itself. Messages are batch inputted to the console one at a time with each successive command being placed underneath the previous one. The command buffer will utilize a vector to store the commands (however, as I plan to have a fixed amount of messages, an array may be more appropriate). The buffer will be wiped each frame to allow for new messages to flow in. The user will be able to add text to the command buffer through the ConsoleManager. By passing a string/char array to the ConsoleManager, the data will be passed to the current console (likely the default one). This data is added to the command buffer and is repeatedly rendered until it is popped off the buffer for space. Since the buffer has a fixed size of messages, any time a new message is added that exceeds capacity, the oldest message is popped off (pop_back).
    - Input Buffer: The input buffer is to be treated a bit differently from the command buffer. Instead of having each successive message added in one after another, we only ever have one message on screen. In fact, this is less of a buffer and more of a single string that is edited and displayed in the same location on screen. The text for the input buffer is anchored to a specific location (likely underneath the maximum reach of the command buffer). As the Input buffer is being treated a bit differently, we do have to have separate pass through information to differ from the command buffer. Command buffer information is stored within the Console object and is not displayed up until it is parsed. Whereas, the input buffer is updated every time a registered key is hit. The pass through method is similar to the Command Buffer with the extra step of the data getting sent to a CommandLogger instead of just the internal console.
- Logging coloration: In addition to generic print commands in a single font, I would like the ability to print in multiple colors to indicate whether a message is an error message, a warning message, or a general message (red, yellow, and white).
    - Implementation: Instead of adjusting the sprite itself, I figured I could instead load in different fonts for each type of color I was looking to use. In the ScreenLogger we already created a sprite font (with associated font) for each message we want to print to screen. Instead of using the same font for each instance of a message, why not swap the fonts based a message type?

- Message Types: We must now add a bit more sophistication to the logging system as we are no longer simply dealing with plain text. My goal is to first implement the console logging system without any coloration and to eventually work in the idea of message types. When implementing message types, I will need to add a new data structure that contains not only a string but also an enumeration of {ERROR, WARNING, DEFAULT} to ensure that the correct font is used. The end user will not need to know the internal functionality of the message typing though, there will be three new functions in the ConsoleManager named Print(…), PrintError(…), PrintWarning(…).

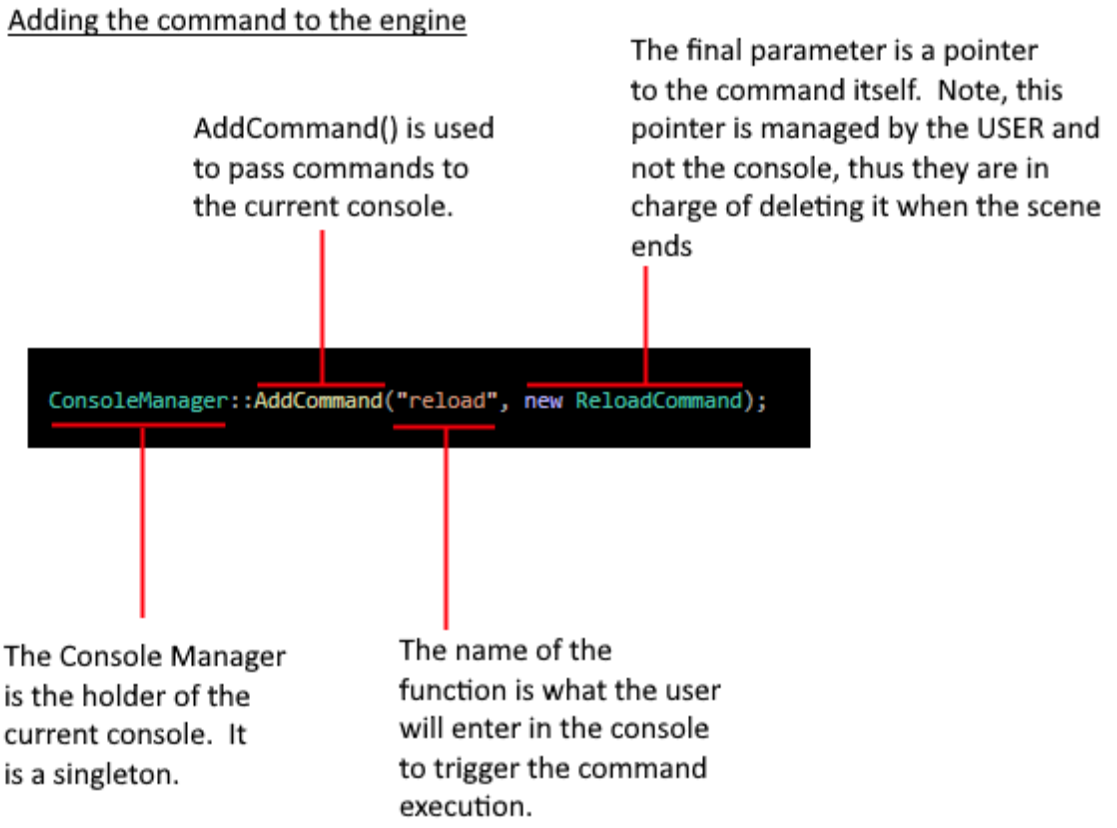
Command System

- Broker: I will need to create a new broker to specifically deal with console commands. The broker will manage a list of commands to execute at the top of the frame (Sly::Update). The interesting component of this is previously for our brokers we would instantly discard any commands after execution. In this instance however, I am choosing to not throw the commands away as the user may want to repeatedly execute the same command multiple times. The commands are not owned by the broker, they are owned by the console.
- Manager: The console manager is in charge of delegating access from the user to the current console loaded into the system. It handles built-in derived command overloading as well as the ability to add commands to the current console. It also functions as a pass through to the console logger to print text within console.



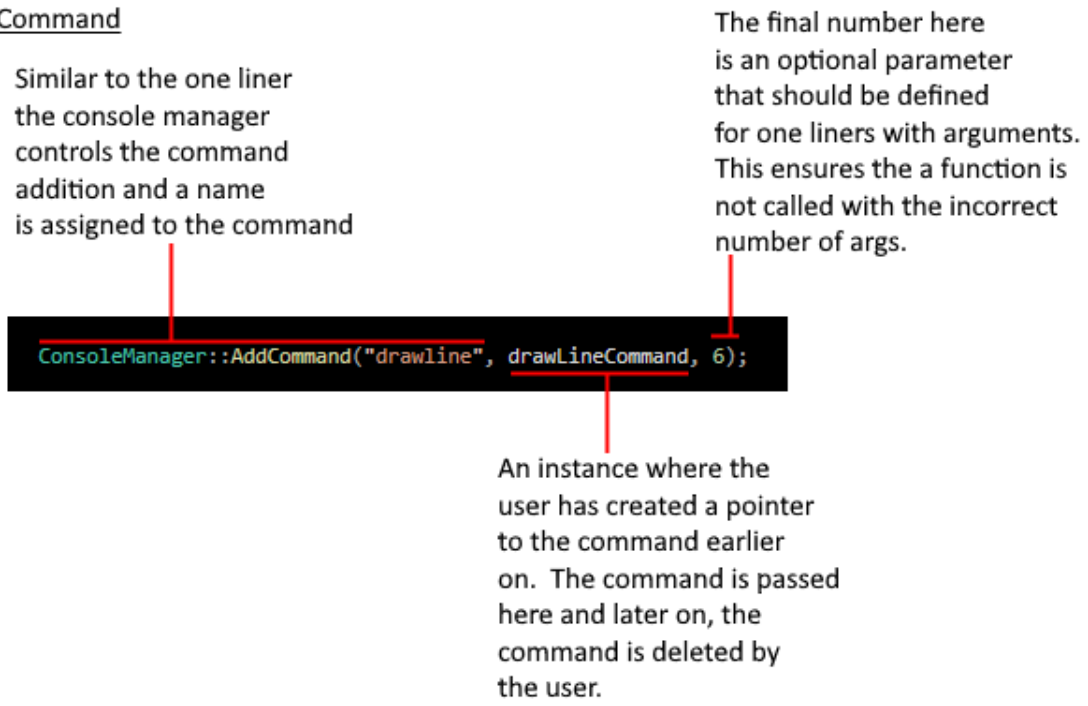
The console manager is a singleton as there should only be one manager no matter how many consoles are created. The console manager allows the user to setup a new console with the SetCurrentConsole command. This is useful if the user wants to disable built-in functions within a release context and to limit the amount of available commands to the player. The manager is also in charge of communicating with Sly who calls for the execution of commands. Once this call is made, the console manager asks the command broker to execute all existing commands within its list.

- Built-in Automatic: These types of commands are the closest to my previous discussion of internal commands I would like to have. These commands control engine level features and have internal calls to locations such as the asset managers, time manager, scene manager, etc. These commands are fairly simple to implement. They will need to be created at construction of the console. I'm utilizing my prototypes existing parser system to recognize the name of passed commands. Thus, if the user wants to run the PingCommand() (which sends an alive message to the console log), they can simply type in "ping" into the console. The ping text is sent through the parser and is matched against items of type function. If there's a match, the associated command is sent to the broker for execution as the top of the frame. Each of these commands *should* have a factory. They must have a factory each as internal commands may not share the same internal data.
- User Built Commands: These commands fall into four categories. "One liners," "One liners with arguments," "Commands with qualifiers" and "Qualifier with arguments." Examples of all of these types are found in "SendingCommands.png" (located in this directory, diagram too large to include).
    - One liner: One liners are used when a command requires no associated object or parameters to be called. This can be used to reference stored console variables such as health and enemies, however it can also be used to reference materials within context. For example, a command in which the user types in "Win" or "Lose" would be able to call the scene manger's current scene to notify it of a win condition. The command itself does not need to know what the current scene is to run.



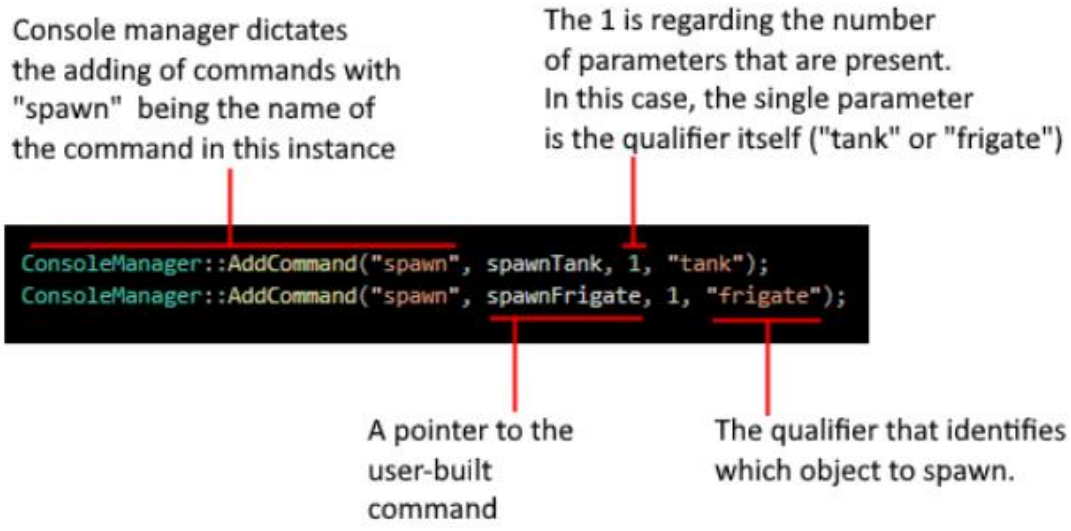
- One line with parameters: A one line that allows parameters is a bit different in terms of setup from its base counterpart. The difference being that the command must have an initializer method. This method is used to pass command line arguments back to the user-built command upon parsing the input.

Adding the Command



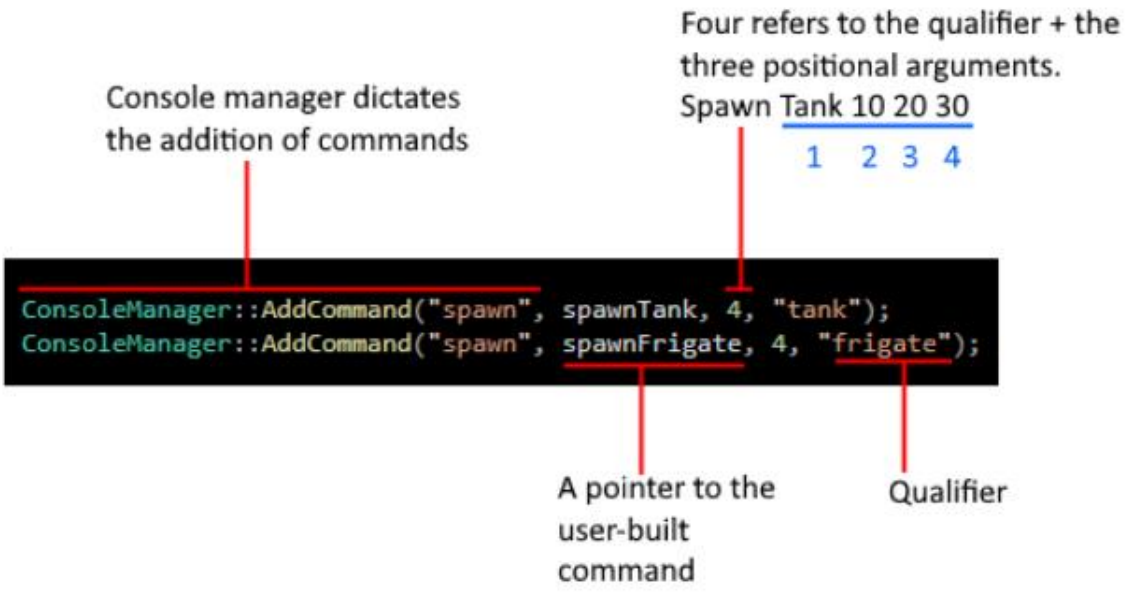
- Qualifier: Commands with qualifiers operate similarly to one liners with one subtle difference. One liners should have only one command associated with them. For example, if the user wanted to draw a line they would need a command DrawLine. This command's only purpose is to draw lines and it cannot draw other objects such as boxes. One approach to this is having a one liner command titled "draw" which has a switch on all possible arguments the user could pass in (terrible), another way is through qualifiers. Qualifiers associate a context with a command. While the input is being parsed a qualifier serves as a way of identifying a subcategory of commands such as "draw **box**" or "kill **frigate**."

Adding the Command



- Qualifier with Argument: Commands with qualifiers are the last type of command that the console supports. The one liner is the fastest command to run (in relation to the command console, not on user implementation), whereas the qualifier with arguments is the most expensive command type. The parser identifies the command, finds the associated qualifier, then passes the argument list to the command where the user must handle the output. The qualifier is meant for specialty functions where a default qualifier is not enough. Useful for testing, parameters can be used to modify the initializer of the command.

Adding the Command



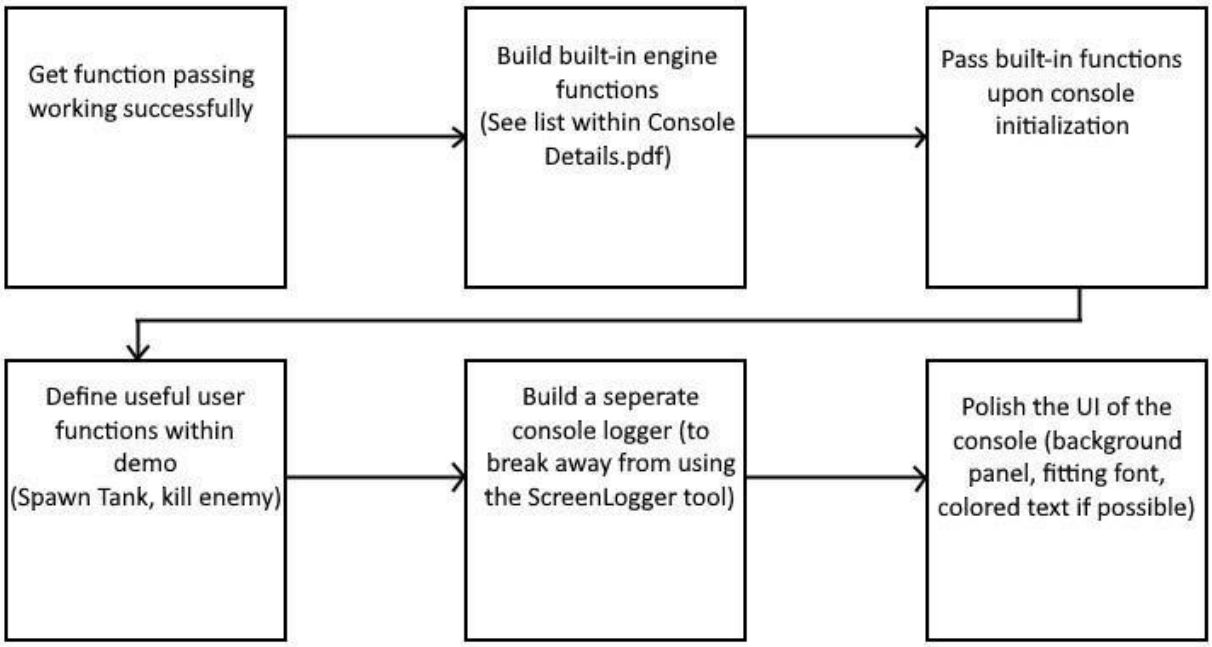
Built-In Command List:

Engine Commands (AKA Built-in Commands)	Description
NoClip / God Cam Startup	Provide a way to access the God Cam without a specific key combo. The idea here is it pulls the tools we're using out of the game and into the console. Gameplay – tools are separated.
Restart / Reload Scene	The ability to reload the current scene or to restart the game without fully closing the game out and starting from fresh. While restarting may be trivial for our demos, I can see the ability to reload a scene as useful if the game takes a while to get to the scene and the gameplay programmer isn't looking to reorder the scenes. This should be fairly straightforward to implement. The current scene is deleted in the scene manager and a new instance is created. If we wanted to restart the entire game (two separate commands) we could delete the current scene and load the default/starter scene.

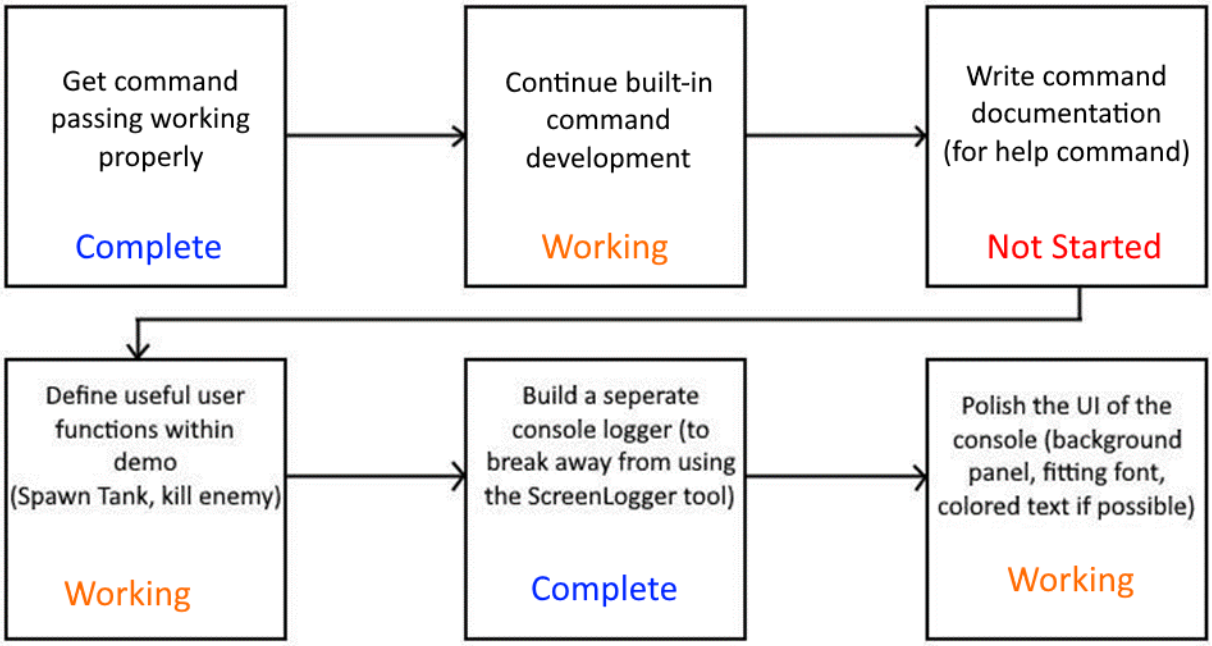
Reset Camera	Reset the Scene Manager's default camera back to its starting position. I can see this having an unintentional effect on the gameplay as the gameplay programmer may want to simply reset the orientation of the camera which the console wouldn't be able to do (too game centric). This merely resets the default camera to its original orientation and sets it back to the current camera.
Time	Get the current running time of the engine from the Time Manager. This will display the time that the game has been running since startup. Simple getter and printing the result to console.
Ping	Get a pong response with the time it takes to execute the ping command. Printing the time it takes to execute the command may be trickier than it sounds as this would require either a timer or a call first to get the time, store it, run the pong command, get the time again, and to print out the difference. At the very least, a hello world for the console.
FOV [x]	Change the current camera's FOV to the specified integer (delete the existing camera and recreate it with the FOV change).
Near [x] / Far [x]	Change the near or far clipping plane to x (delete and recreate the new camera).
Models, Images, Shaders, SpriteFonts, Textures	Print out a list of all stored models, images, etc. stored by name. The usefulness of this is limited, however a gameplay programmer may want to see if an asset they were trying to use was actually loaded into memory or not. By querying the stored assets they could confirm whether or not it is stored in the engine.
Alarms	Using the alarmable manager, display a list of alarms that are set to trigger on the timeline.
VisualizeAll	Place a Bounding volume (as defined by the collidable) around each one of the collidable objects.

Proposed Timeline

Previously:



Now:



Week by Week Timeline

Task	Estimated Week	Explanation of Task
Allow User to pass commands to the console	Week 4	Build up the systems within the console to allow the user to pass commands into the console. This is merely the receiving and management of user commands in the console after they are passed in. This system has been completed via prototyping.
Build a new screen logging tool	Week 5	Create a tool that the user can **easily** use to output data to the console. This tool will need to handle print messages in Default, Warning, and Error colors.
Build Built-in Commands	Week 6-8	Built-in Commands and User commands are both taking place during these two weeks. Many commands I initially think are Built-in commands turn out to be user commands only and vise-versa. I want these two weeks to be development of either a user or built-in command as they can flip flop. Either way, I intend to make several commands of each by the end of week 8.
Build User Commands	Week 6-8	See above
Polish UI	Week 9-10	UI polish includes making the panel of the console (background) a slight gray color along with adjusting the font and coloration to fit the new scheme. Ideally, I'd like the output to look similar to the Gold Source console.