DOOM 3 BFG
ENHANCEMENT PROPOSAL

BASEMENT GURUS

Isaac Chan
Daniel Elmer
Matthew Sherar
Yohanna Gadelrab
Dylan Liu
Kainoa Lloyd

Table of Contents

Abstract

Our proposed new feature for this game is a real time radar map positioned on the top right corner of the screen to show enemies positions in relation to the player. We proposed two implementation methods, one affecting the UI, renderer, and game logic subsystems and one that only affected the renderer and game logic. Through our SAAM analysis we decided that the latter was better suited for efficiency purposes as well as impacting less of the overall architecture. This method added a new subcomponent (*StateMap*) to Game Logic and the HUD subcomponent to the Renderer. The most notable design pattern used when changing our architecture is the observer design in the *StateMap*. We learned that a benefit of the Object Oriented Style architecture is that our testing of the new implementation is limited to the Game Logic and the Renderer to verify that the Radar constantly updates the correct position of the monsters. We use a sequence diagram to help describe a use case of the radar. A noteworthy point related to concurrency is that the radar uses a single thread. Overall there were a few limitations of our implementation method mainly being poor portability due to high coupling.

Introduction

This report discusses a new game feature that we have proposed to add to Doom 3 BFG. The new feature is a heads up display radar map that shows enemy positions in relation to the player on a small display at the top right corner of the screen at all times. The motivation for this is to allow players to more easily search out and destroy monsters. Through this report we will propose two different implementation methods and conduct a SAAM analysis on both methods to determine which one we should go through with. With the selected method we will discuss the impact on the game's architecture showing an updated sequence diagram of

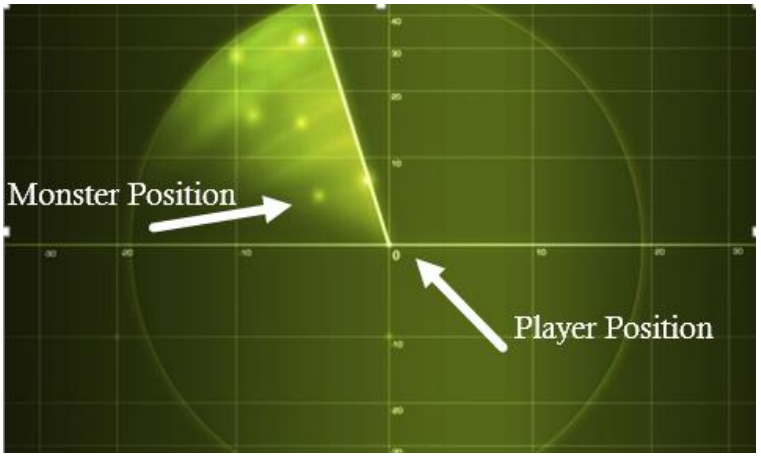


Figure 2 Description of the components of the map

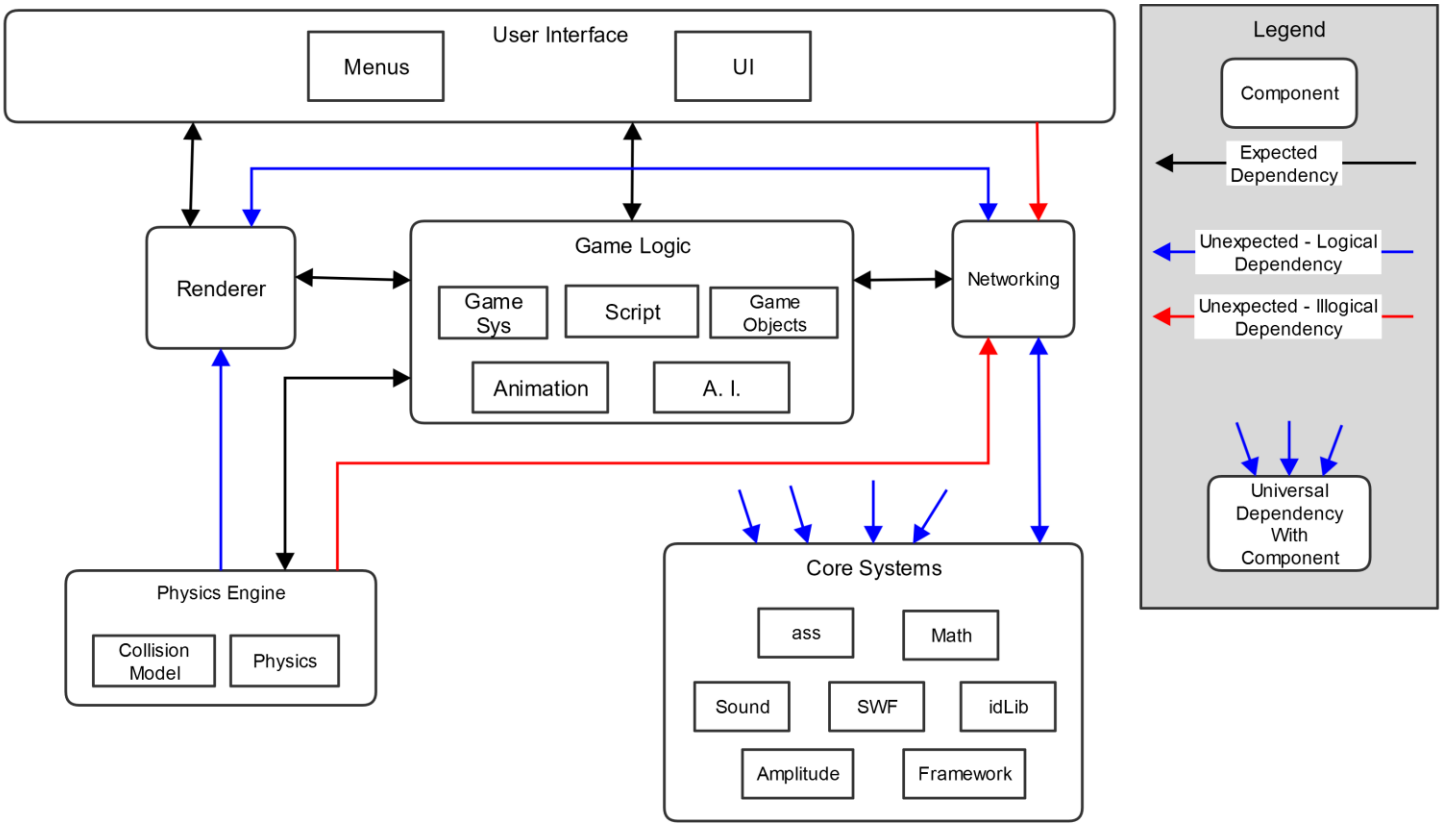


Figure 1 Concrete Architecture

a use case for the new feature. In discussion of the selected method we will elaborate on how it relates to the concurrency of the system and the limitations of implementing this feature. We will conclude our report highlighting lessons we have learned through creating this report and summarizing key details.

Figure 1 shows our previous Concrete Architecture. The key components along with their key functions that will be affected by the new feature are:

Game Logic: Handles all calculations the game needs. Such as, Math, A.I. and calculating co-ordinates of objects on the screen.

Renderer: Uses information obtained from other components to generate the required graphics that will be displayed on the screen.

User Interface: Displays graphics generated by the renderer on the screen.

Proposed Implementation

The feature to be added to the game is basic radar map that is overlaid on the players screen and constantly displayed. It does not show solid game objects such as walls, but displays the player's position relative to other enemies and players present in the game. In the mock up shown below there is a standard game screenshot with the game map shown in the upper right hand corner of the screen. This is constantly displayed and continually updating. The player is oriented as looking north and the other players and enemies are shown as the small dots in the radar.

Figure 3 A mockup of how will look like while playing the game

To implement the proposed feature we decided to add a new subsystem that is contained within the game logic subsystem. This will initialize and maintain an object that contains a 2d view of the world with the player positions

included in it. In order to render this object to the screen a small class was added to renderer that will check for updated versions of the map, and then hand it off to the Renderer Frontend to render to the screen. We choose this implementation due to its simplicity and small number of new dependencies introduced.

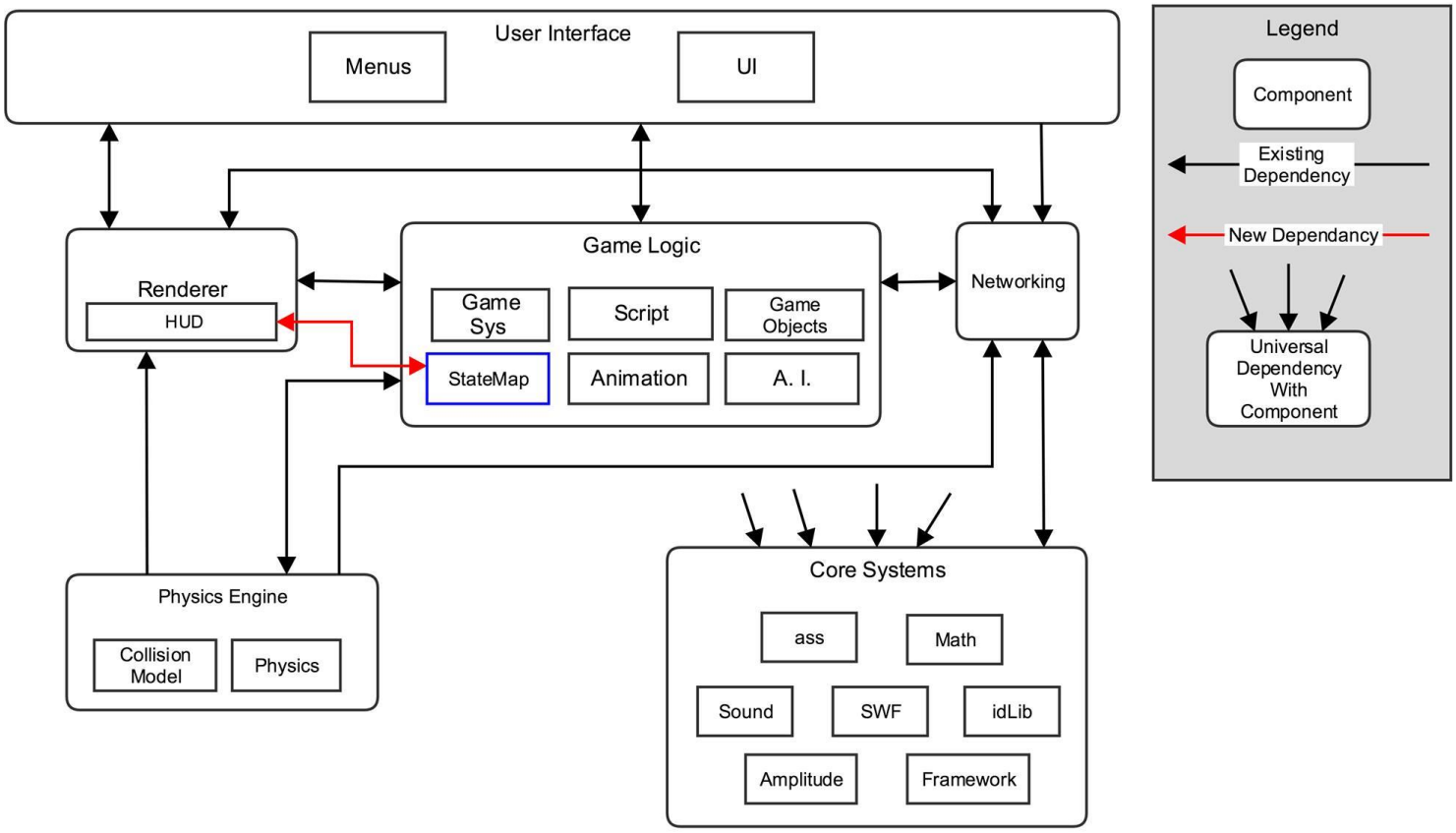


Figure 4 Proposed Implementation

Alternative Implementation

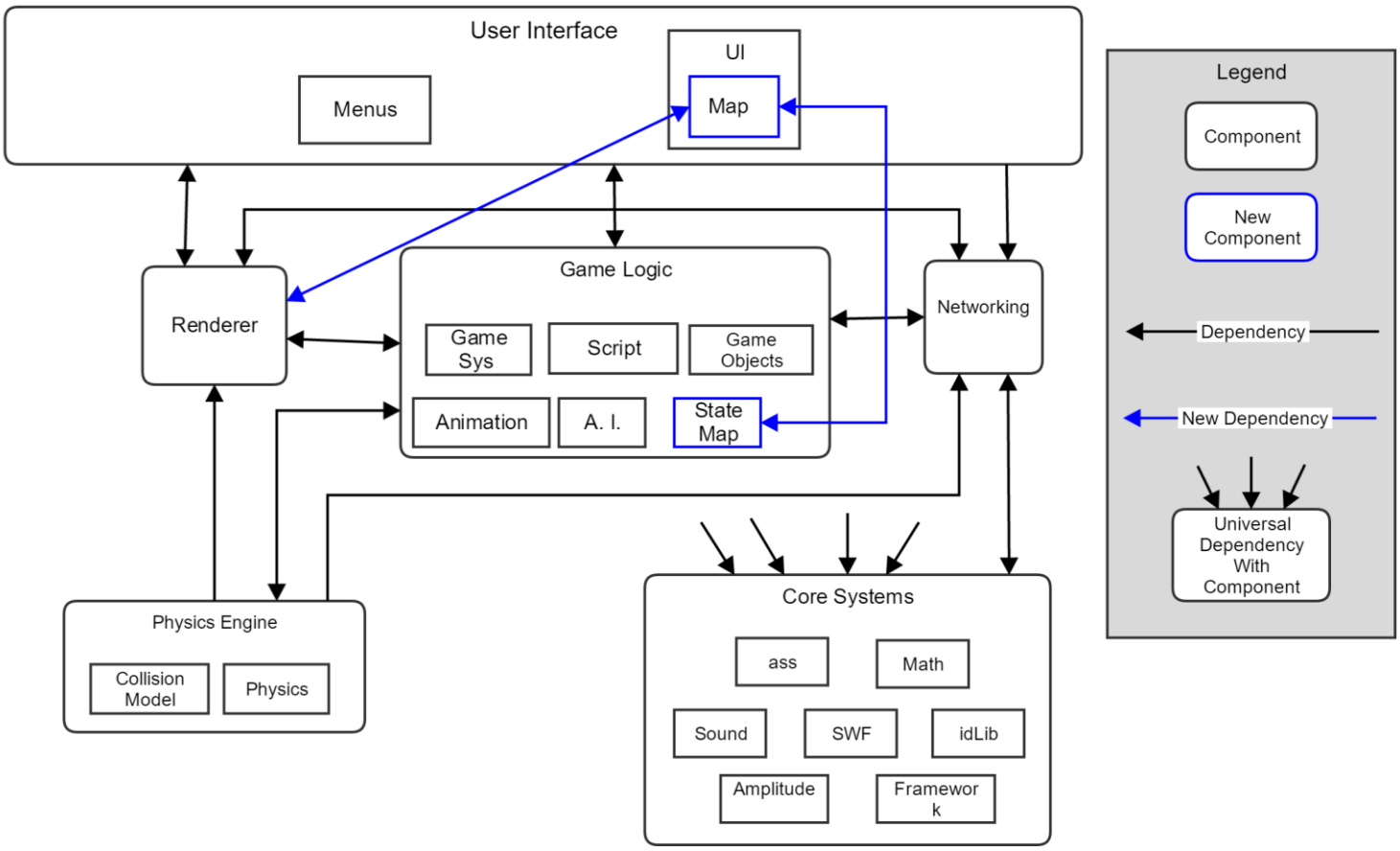


Figure 5 Alternative Implementation

Another implementation would include the **User Interface.** This works similarly to the Proposed Implementation, the **State Map** calculates the co-ordinates of Monsters on the Map but instead of sending them to **Renderer** to generate the updated Map, it sends them to a new subcomponent in **User Interface, Map** that in turn will send those co-ordinates to **Renderer** for rendering the map with the new monsters positions and display that rendered map when the **Renderer is done.**

SAAM Analysis

Stakeholders

- End user (or the player) whose interests involve the functionality, usability, and ease of understanding of the minimap element.
- Developer whose interests involve minimizing the number of affected subsystems and dependencies between subsystems (i.e. maximizing cohesion and minimizing coupling).

Candidate Implementations

- Game Logic highly coupled with Renderer for a simpler, but efficient geometric map.
- UI (minimap as a UI element) and Renderer centered with assistance from Game Logic.

Non-Functional Requirements (NFRs)

- Performance
- Integration
- Resource Constraints
- Portability

Non-Functional Requirement	SAAM Alternative Implementation	SAAM Proposed Implementation
Performance	Due to the constant communication between the Renderer, Game Logic and UI subsystems this puts a lot of more stress on the CPU due to the intensive amount of processing. Constant communication between these subsystems allows us to create a more detailed map. Unfortunately, creating a more detailed map does require more power therefore causing a loss in performance.	This particular architecture satisfies the performance requirement as it appears to minimize unnecessary coupling and maximizes cohesion since it complements the functionality of the existing subsystems. This is because the primary benefit of choosing this modified architecture is that it delegates the task of computing the states of the player, enemy, and environment to a subsystem that is already responsible for doing those kinds of tasks (Game Logic). It also complements the existing functionality of the Renderer subsystem, which is a largely visual subsystem that is responsible for rendering the graphics and visuals on screen.
Integration	This implementation will be difficult to implement. It requires modification in both the Game Logic and UI subsystems adding a State Map and a Map component to each subsystem respectively. To create our radar map we would need to create a new window interface. To do this we would be inheriting classes from the UI class that is already in place in DOOM 3 BFG. Due to this, this implementation is heavily reliant on the UI subsystem and has high coupling with the UI classes.	The minimization of unnecessary coupling also allows for easy integration, since most of any modifications made will only affect two subsystems. The only two major subsystems that will be interacting with one another are Game Logic and Renderer. Of course, the UI subsystem will also have a role as the minimap itself is a UI element, but the interdependencies between UI and the other two subsystems are reduced.
Resource Constraints	Since this implementation puts more stress on the system it requires more processing power from the CPU. For this reason it would be more difficult for older machines to renderer images quickly and keep a consistent frame rate. For more modern machines the requirement should not be an issue since machines today are generally more powerful.	It is unclear whether or not this candidate architecture will comply with resource constraints, as the radar has to be updated in real time (i.e. frame by frame) in relatively high quality, which could put a lot of stress on the system. We will use geometric drawing to attempt to minimize the impact of this problem.
Portability	This implementation has a far superior chance for potential reuse due to being able to inherit from already existing U.I code.	The tight coupling between the components of Game Logic as well the need for new, exclusive interfaces makes it only good for the one use.

To conclude, we chose the proposed implementation due to performance taking the highest priority as far as non-functional requirements go, but also due to the lesser impact on the currently existing architecture.

Impact/Effects on Current Architecture

Concrete Architecture - Impacted Areas

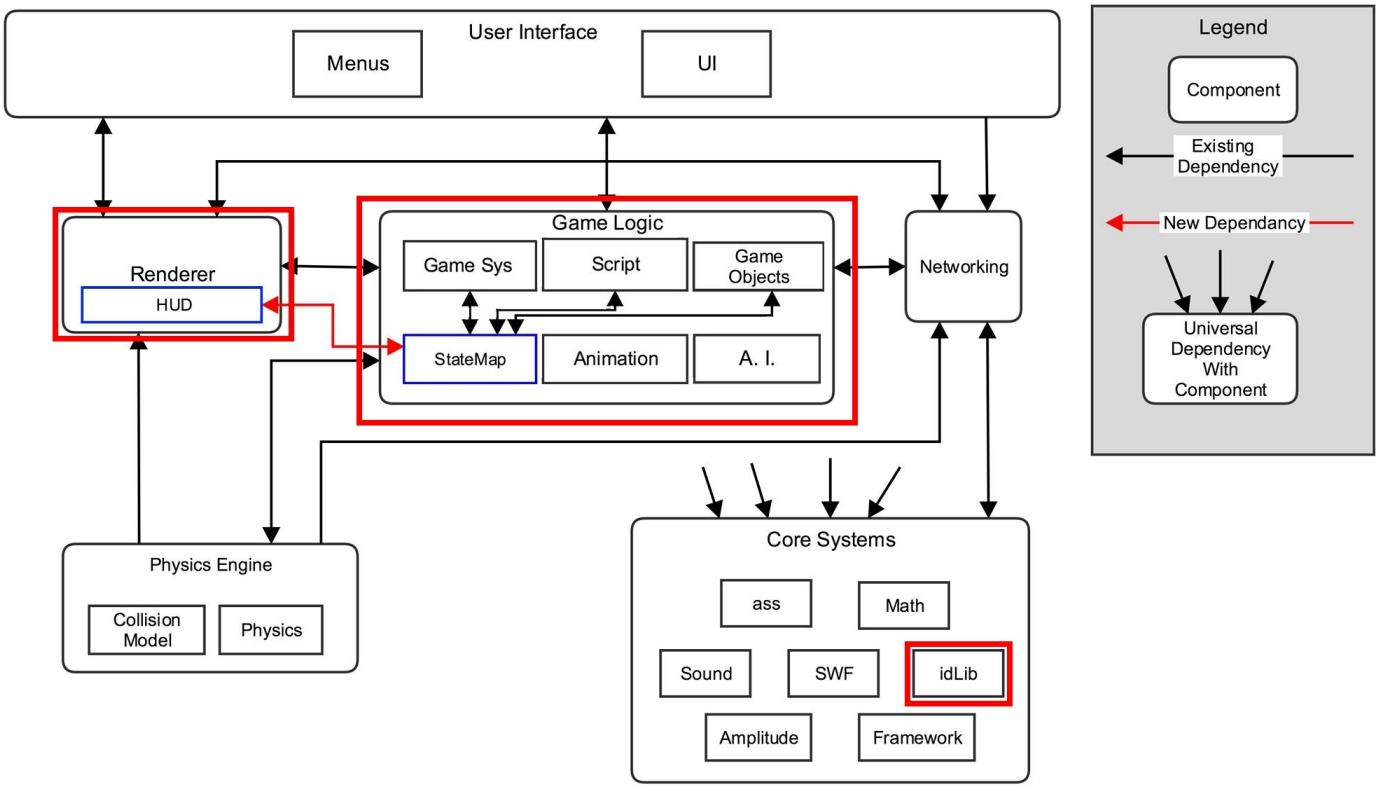


Figure 4 Impacted Areas of Concrete Architecture

Impact on Subsystems

The impact of the radar on the architecture largely comes from the new component, *StateMap*, on the rest of the existing Game Logic sub-components. We are going to build the radar entirely through new classes and interfaces while drawing upon the pre-existing code, thus minimizing the impact on previously existing features and components.

The *StateMap* will need to be highly coupled with the other subcomponents of Game Logic in order to observe and obtain the necessary data. It will be dependent on the game scripts, the scripting engine, and the Game System scripts in particular, which will relay the appropriate information to *StateMap*, which can compose the abstract radar data, with some references to Core Systems for assistance with the math. This way, we can connect the updating of the radar directly to the game state if we build this into the appropriate thread without changing the previously existing classes much.

When it comes to the rendering process we decided to limit the impact to just the connection between Game Logic and the Renderer to lighten the impact with our chosen implementation. However, it still requires a new interface to be added to the Renderer in order to properly interpret *StateMap's* abstract view of the radar in order to draw it properly. To make sure that abstract view is as minimal as possible, we would just build the majority of the radar's drawing process to the renderer's interface, only requiring the location information of enemies and character to complete the process. We would use geometric drawing for a less intensive process on the renderer instead of rendering windows like our alternative proposition. To do this, the new interface can draw from the large library of geometric drawing functions in the Core Systems. No changes will need to be made to access those libraries, lightening the impact of the radar further.

Design Patterns

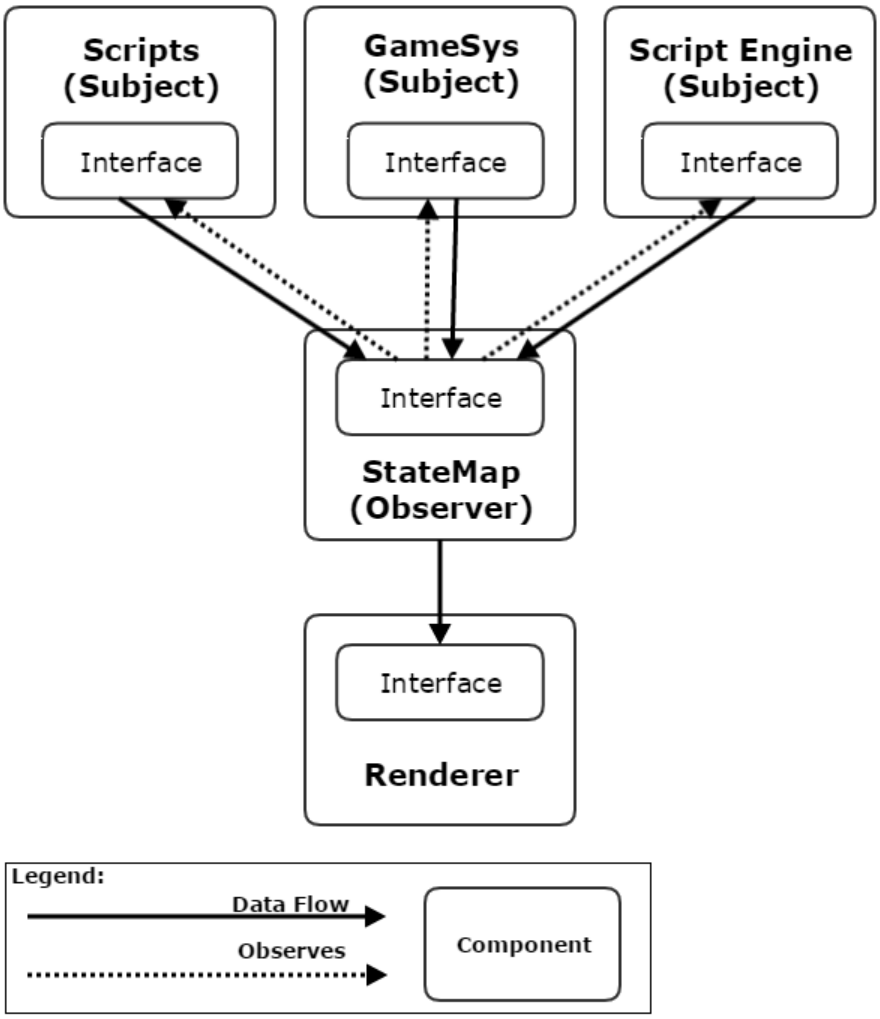


Figure 5 Proposed Observer Structure. Note that there are other subjects aside from the three shown, and the Core Systems dependencies are not shown.

We intend to take advantage of the benefits of an observer design when changing our architecture to accommodate the radar feature; it is required that *StateMap* receives updates whenever the relevant position data in Game Logic is altered. Our observer is *StateMap*, which will accommodate both the concrete objects for managing the received data as well as updating the interface to take in that data. Our concrete subjects are the previously existing classes in Game Logic, and we will need to build observer interfaces into all the relevant sub-components on which *StateMap* will interact. They will need to be aware of the observer *StateMap,* and send the notifications to *StateMap* when the relevant information to the locations of the player and enemies changes. This way, we can connect the updating of the radar directly to the updating of Game Logic if we build this into the appropriate thread without changing the previously existing classes much.

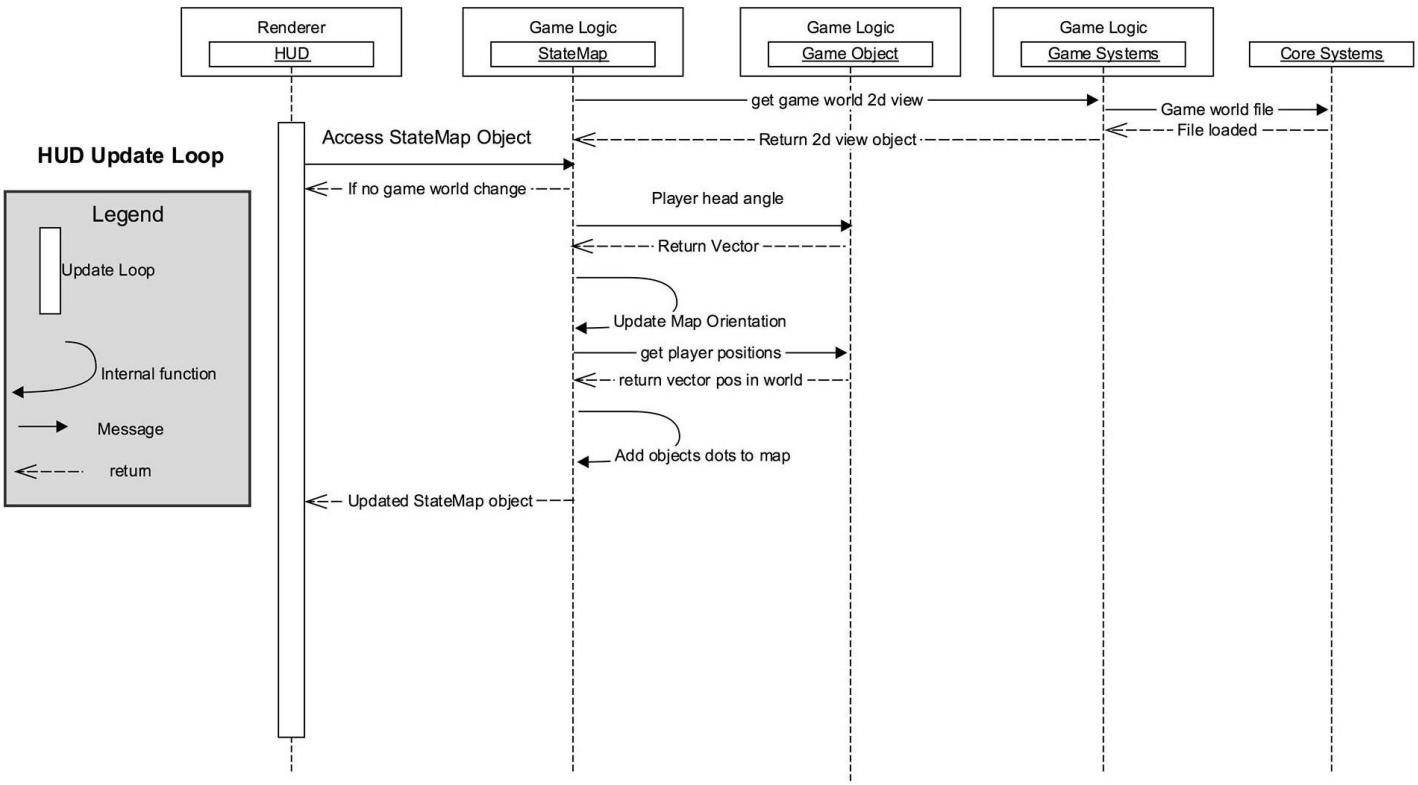
Testing

This implementation only affects two subsystems, the Game Logic and Renderer, therefore we need to test these two subsystems. To do this we plan to test the Game Logic subsystem first to make sure state maps are generated correctly. The Game Logic subsystem is required to be tested first since the Renderer subsystem depends on the information given by the state map. Once this has been thoroughly tested we can test the Renderer to make sure that it is refreshing constantly and consistently.

For the Game Logic subsystem we need to make sure that enemies are in the correct position when displayed on the HUD map. This subsystem will require two types of tests. The first to check if one enemy appears in the correct position in the state map and the other to check whether multiple enemies are appearing in the correct positions. This test requires a calculation of the distance between the player and the enemy. This information is then translated to the state map to be displayed on the HUD map which will be generated by the Renderer.

For the Renderer subsystem we need to make sure that the map is constantly refreshing. The map should update every frame to give the player the most up to date information about enemy location. The map should not be a second or a few seconds behind since this will provide incorrect information to the player. For this reason the map needs to be in sync with the Game Logic subsystem. To test we need to generate an enemy or enemies, calculate the distance the player is from the enemy (or enemies) and then check that their location corresponds to the correct location on the map.

Sequence Diagram



The sequence diagram above depicts the update loop of the renderer that works in correspondence with the update loop contained in the StateMap folder. When the game level is initialized the the StateMap retrieves a 2d view of the map from the Game Systems component, so StateMap knows the boundaries and map size. Once the

game is active, the renderer loop is running continually drawing a new view of the game world. Since the renderer needs to be updated very often (perhaps at 60Hz) whereas the StateMap needs to be updated much less frequently as player and enemy positions do not change as quickly. Therefore when the renderer updates a new screen and checks for a new map to display, most of the time the map will be unchanged and there is nothing new for the StateMap object to return. When the StateMap object does update it needs to orient the map in the direction that the player is facing, so it first gets the vector coordinates of the players head angle from the Game Objects subsystem. Once the map is correctly oriented the the StateMap needs to get all updated player positions and all enemy positions and add them as dots to the new 2d map object StateMap maintains. This new map is then returned to the renderer the next time that the renderer object accesses.

This sequence diagram demonstrates the limited modifications to renderer that are required to support the addition of the map displayed on screen in the game. It only needs functions to check for new map objects, but can then renderer them exactly as it renderers all other game objects. In addition as the StateMap only needs to be updated perhaps 2 or 3 times per second, these additional checks by the renderer should have minimal effects on performance. Within the StateMap object, once the map is instantiated most of the computation is simple vector calculations which isn't computationally expensive.

Concurrency

Since the map will be designed as a radar, only the monsters positions relevant to the player will need to be updated. Therefore, only one thread will be needed to update the map. That includes calculating new monsters positions, generating the new map based on those calculations. Displaying the map on the screen is already handled by the Main process of the game since that part won't be affected by the new feature.

Using one thread also will help prevent taking too much resources thus keeping the gameplay as smooth as possible, which is a huge priority on an action-heavy game like Doom 3.

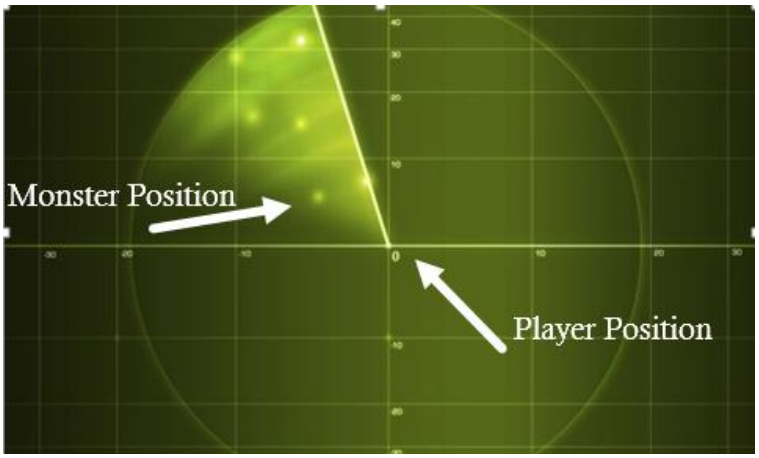


Figure 6 Since the map is designed as radar, the monsters positions will get updated every 1-2 seconds

Potential Risks

Our analysis of the risks of implementing this feature concluded there were no major concerns that might affect the gameplay.

For the **Security** aspect of the game, the new feature won't affect the **Network** component of the game, thus guaranteeing no player will find a security hole that will allow them to cheat. In terms of **Data Integrity** the new subcomponent implemented inside **Game Logic** will not affect other subcomponents.

With respect to **Performance**, the map will only use one thread to update itself, and it will get updated every few seconds per design thus preventing clogging of the system resources.

Limitations

There were a few limitations of our chosen implementation, one being that it is not very portable. It requires a high amount of coupling with the Doom 3 BFG's script files and the required interface in the renderer. It also loses the benefits from using already written U.I. code which could help with flexibility and reuse in the system for other purposes. In addition, the map itself will need to be less detailed if we want to keep it as efficient as possible.

Lessons Learned

Through this report we learned some valuable skills and insight on proposing new features to implement to a game system. Firstly was how to conduct a SAAM analysis which was a completely new concept to our group. In addition, we found that knowledge of the concrete architecture made it much easier to conduct a SAAM analysis and see what components would be impacted. Finally, the object-oriented style of the architecture facilitates easier changes to the system by minimizing the number of affected components.

Conclusions

To summarize our report, the new feature we proposed to add to Doom 3 BFG was a radar type map. After conducting a SAAM analysis of two different methods of implementation we decided to use the simpler implementation for the sake of efficiency, while sacrificing potential code reuse and aesthetic quality. The implementation method would add a new subcomponent to the Game Logic (State Map) and added an interface to the renderer that can interpret State Map's data as well as get the proper mathematical data from the Core System's Geometric Libraries. Our planned testing includes verifying correct generation of state map and making sure map is consistently updating. We have learned valuable insight on how to conduct a SAAM analysis and the process of proposing a new feature for a game.