

***Chief Executive Unpaid Intern Chief Technology Officer***

James Brunet Haydon Ruth

100972413 100859519

***Former President Lead Scientist***

Rafael Martinez Sanchez Zachary Fry

100990360 100847625

Unity Glossary

Running on Fumes is a 2.5D top-down space puzzle game. Developed almost entirely within Unity’s development environment, some terminology may not be familiar to you. A few buzzwords that our team will use during this introduction are:

* Scenes: Scenes are the base container in unity: they hold all the game objects. Scenes are used to create a menu or individual levels. Our project has five scenes:
  + The main menu scene, which triggers the loading of either the high scores view scene or the first level scene.
  + The Level1 scene, which contains the first level of the game.
  + The Level2 scene, which contains the second level of the game. It is only loaded upon successful completion of the first level.
  + The high scores submission scene, which prompts the user for input and submits their high score to our server.
  + The high scores view scene, which displays the top ten high scores to the user.
* GameObjects: This is the base class for all entities in a Unity scene
* Components: Components give functionality to GameObjects. An example component is the RigidBody component, which gives a GameObject physics simulation and allows other objects to apply force to it. Another component is the Mesh component, which gives a GameObject a graphics primitive.
* Scripts: Scripts are components that are created by us! They trigger game events, respond to input, and can modify component properties over time. Our scripts are written in C#.

Architecture

This project does not have a “pure” architectural style. Instead, it features a mix of three distinct architectural styles.

* Object-oriented
* Blackboard
* Client-server

Object-Oriented:

All of our unit behavior, entities, and scripts are designed with the object-oriented style. In a single level, data is shared between objects using method calls and inheritance. Because of this, we were able to break our problems into different parts: we implemented ship thruster physics completely independently of ship gravity interactions, for example. Because our game has complex and constantly changing data due to interactions between players and the environment, this architecture style is extremely important in the scope of a single level.

Blackboard:

All of our inter-level communication is modeled after the blackboard style. The shared problem to be solved is the final state of the player (score, victory/loss, other). Critical information about our game state is pushed to a shared data repository in SharerPreferences before loading the next scene. After switching scenes, this information is pulled into the current level

Client-Server:

On the highest level, our application can be broken into two layers: The Unity application (runs on the client device) and our Python/Django/SQLite code (runs on our server). This layering is very strict.

The client layer in our Unity application is only visible in two scenes: the high scores submission and high scores viewing scenes. Our implementation is fairly faithful to this architecture within this scope: Clients do not talk to each other, the server is capable of handling large numbers of clients, and data integrity is maintained on the server.

This architecture is necessary to properly handle our original requirements. One of our proposed functional requirements was “responsive leaderboard system that can be used to compare individual level scores against other players,” and in the opinion of our team the only responsible way to implement that was by using a client-server architecture.