# Systems

There are several systems in Bubble Bobble. These are divided up into Game, States, GameObjects, and GameData. There are additionally the supporting classes of Animation and InputManager, but they are not significantly important to game function.

## Game

Game runs the game. It is created in main(), and subsequently creates and manages the RenderWindow, and StateManager. The loop function in Game is called in main(), and the game begins. In this loop function is where events are polled. In it general game processing occurs and window clearing and displaying happens as well. The reaction to the events is handled by the StateManager and higher specific game processing is handled by it as well.

## States

The StateManager controls the states and uses polymorphism to control what state is used. It pushes and pops back states to a vector and chooses the last one to operate. Each state is held in a BaseState pointer. Its derivatives include SplashScreenState, MenuState, and GameState. Individual processing is done in each state.

### SplashScreenState

The SplashScreenState is what it says. It loads and shows a blank screen. Bypassable by pressing spacebar.

### MenuState

The MenuState is where a menu can be implemented. If there is no menu, switching to the GameState can be done by pressing escape.

### GameState

The GameState runs the game. It creates a new GameData pointer object, the first level, and the players. In processing events, only keyboard input is taken in via a parameter from Game and passed through StateManager, and once the input has been determined, the GameObjects handle reactions. There is one exception. In the case of an escape key release, the program returns to the MenuState. In general processing, subvectors in GameData are passed through according to the hierarchies of precedence and reaction. Each GameObject has its update function called and further specificity is handled in there. For rendering, every GameObject is passed through and its render function is called. GameObjects are moved by calling GameData’s kill function.

This leads to the next main system:

## GameObjects

Its general form is called GameObject, but it has many derivatives. GameObject is not an abstract class. It is made with basic functionality. Most of its virtual functions are not overridden it each child class, but several are based on behavior. There are a whole host of functions to make the GameObject class easy to use, but first there are general requirements: each GameObject requires a RenderWindow pointer and a GameData pointer. This is so that window tests and GameData object addition and subtraction can occur. There are several types of functions: updating, event initiation, event reaction, and interfacing. The first is composed of the update function. It is further broken down into levelPlay() and levelTransition(). levelPlay is for updating during playing the level and levelTransition() is for updating during transitioning levels. Swapping between these two functions is controlled when levelStart() and levelEnd() are called. Each performs the necessary changes for preparing the respective phase and is called once per level switch.

The next update functions are time() and distance(). Time controls updating for time based events and Distance controls for pedometer based events. These are initiated events. There are functions for setting limits, getting current elapsed units, stopping testing, and functions for when the limits have been passed. The only one that can be overridden out of that set is limit passing functions. This is so because different GameObjects react differently to limit passing and have different uses for the timers and pedometers.

This transitions into the next portion: collision. All three in this set are virtual. collision() performs the actual testing. collideWith() determines what GameObjects collide with itself. collided() is the reaction to a collision. When a collision occurs, both GameObjects’ collided() function is called, so it is imperative to not double check. This leads into the individual GameObjects derivatives and the hierarchies of precedence and reaction.

Additionally, there are position and velocity functions.

Lastly, there are getters and setters.

### Level

The Level class is for the level.

### Player

The Player class is for the player.

### Monster

The Monster class has many derivative classes. It was determined that polymorphism was the fastest method of distinguishing between monsters and their respective pathing and updating can be changed. Full implementation of this class has not been determined.

### Pickup

Pickups are currently for objects spawned from Monster deaths. Full implementation of this class has not been determined.

### Projectile

The Projectile class is used for bubbles and enemy spawned projectiles. Full implementation of this class has not been determined.

### PointText

PointText is for the floating point value effect of collecting pickups or multiple enemy kill streaks. It disappears after a distance limit is passed.

## GameData

The GameData class keeps track of all of the GameObjects. They are placed a vector with a fixed number of subvectors for each GameObject in order determined by the hierarchy of precedence. One GameData object is shared between the state and all GameObjects in that state. The state can then run through the GameObjects in order of the hierarchy of reaction, and the GameObjects can have access to adding new states and killing themselves. However, actual deletion of the GameObjects only occurs in the GameState. Otherwise nested deletion errors could occur. An example of this is when one GameObject calls on another GameObject which deletes the previous GameObject. When the program moves back out of scope, it moves into a GameObject which no longer exists. This is bad. That is why it is not allowed. GameData also provides an interface for interacting with GameObjects. Whether GameObjects of in a certain index exist - which in reality tests for if there are any GameObjects of a certain type since each subvector is of a certain type, presorted -- adding objects to a killList, adding objects to the data list, and getting GameObjects can all be done in GameData. It is designed to be as easy to use as possible. However, when in a GameObject, use addToKillList(), not remove(), or clear() to delete an object, otherwise the nested deletion error could occur.