Code Structures

# 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.

#### Member Functions

##### Public

Game() - Constructor. Creates the RenderWindow. Creates the first state in StateManager, the SplashScreenState.

~Game() - Deconstructor. Deletes the RenderWindow.

loop() - Loops through the game. While the window is open it does several things. It polls for events and passes the events to the StateManager to process. It calls the general StateManager processing function. It clears the RenderWindow, calls the StateManager draw function, and displays the RenderWindow.

#### Member Variables

##### Private

int FPS\_m - This is the maximum frame per second speed. It is initialized in Game() and the window is set to its value.

sf::RenderWindow \*window - The RenderWindow pointer is given a new RenderWindow in Game(). This is passed on to all GameObjects and States through StateManager. It is created with the dimensions from screen dimensions and is given the maximum framerate set by FPS\_m.

sf::Vector2i screenDimensions - Initializes the screen dimensions of the RenderWindow.

StateManager stateManager - Runs the StateManager. It is initialized with a new SplashScreenState. It is the interface to the actual states.

# StateManager

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.

#### Member Functions

##### Public

StateManager() - Constructor

~StateManager() - Destructor. Deletes all of the states held in the states vector.

void processEvents(sf::RenderWindow \*, sf::Event) - Calls the BaseState at the back of the states vector to process the event.

void process() - Calls the BaseState at the back of the states vector to do general processing.

void draw() - Calls the BaseState at the back of the states vector to draw.

void push\_State() - Adds a state to the back of the states vector.

void pop\_State() - Deletes a state off the back of the states vector.

#### Member Variables

##### Private

std::vector<BaseState\*> states - Holds the BaseStates of the StateManager. The one on the end of the list is the active one. Acts as a state stack.

# BaseState

The general state for the states.

#### Member Functions

##### Public

BaseState() - Constructor.

BaseState(sf::RenderWindow \*) - Constructor.

~BaseState() - Destructor. Deletes the GameData.

bool pause() - Returns true if pause\_m is true.

virtual void processEvents(sf::Event) - Specific event handling.

virtual void process() - Specific process handling.

virtual void draw() - Specific drawing handling.

bool switchTrue() - Returns true if stateSwitch is true.

std::string nextState() - Returns nextStateS.

GameData \* getGameDataPTR() - Returns the state’s gameData pointer value.

#### Member Variables

##### Protected

bool stateSwitch - Boolean flag for switching the state.

std::string nextStateS - A string for the next state.

bool pause\_m - Boolean flag for pausing the state.

InputManager inputManager - Holds the InputManager.

GameData \*gameData - Holds the GameData pointer.

sf::RenderWindow \*window - Holds the RenderWindow pointer.

## SplashScreenState

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

#### Inheritance

Inherits from BaseState

#### Member Functions

##### Public

SplashScreenState() - Constructor

SplashScreenState(sf::RenderWindow \*) - Constructor

~SplashScreenState() - Destructor

void processEvents(sf::Event) - Controls event handling for the SplashScreenState.

void draw() - Draw.

## 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.

#### Inheritance

Inherits from BaseState

#### Member Functions

##### Public

MenuState() - Constructor

MenuState(sf::RenderWindow\*) - Constructor

~MenuState() - Destructor

void processEvents(sf::Event) - Controls event handling for the MenuState.

## 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.

#### Inheritance

Inherits from BaseState

#### Member Functions

##### Public

GameState() - Constructor

GameState(sf::RenderWindow \*) - Constructor

GameState(sf::RenderWindow \*, bool) - Constructor with boolean variable for two players.

~GameState() - Destructor

void pause() - Pause the state. Currently unimplemented.

void processEvents(sf::Event) - Process game specific events for the GameState.

void process() - Process updating events for the GameState. Passes through all GameObjects in accordance to the hierarchy of reaction.

void draw() - Draws the GameState. Passes through all GameObjects.

##### Private

void initialize(bool) - Creates the initial GameObjects.

void processIndividual(unsigned int) - Passes through all GameObjects in the GameData list with the passed in index value.

# 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.

#### Member Functions

##### Public

GameData() - Constructor

~GameData() - Destructor

void kill() - Goes through the killList and deletes all corresponding data from the gameObjects list.

std::vector<std::vector<GameObject \*>> getAll() - Returns the gameObjects vector.

std::vector<GameObject \*> getList(unsigned int) - Returns the gameObjects subvector corresponding to the parameter index value.

void add(unsigned int, GameObject \*) - Adds a GameObject to the gameObjects, subvector determined by the parameter index value.

void addToKillList(unsigned int, GameObject \*) - Adds a GameObject to the killList, subvector is determined by the parameter index value.

bool exist(unsigned int) - Returns true if there are objects in the specified subvector in gameObjects.

void clear(unsigned int) - Adds the whole specified subvector to the killList.

##### Private

void remove(unsigned int, GameObject \*) - Removes the specified GameObject from the specified subvector.

void remove(unsigned int, unsigned int) - Removes the GameObject specified by the two indicies from the gameObjects list.

void remove(unsigned int) - Removes every GameObject in the specified list.

#### Member Variables

##### Private

std::vector<std::vector<GameObject \*>> gameObjects - Is a presorted type list that is used in keeping track of GameObjects. The list ordering is determined by the hierarchy of precedence. See flows and hierarchies for more.

std::vector<std::vector<GameObject \*>> killList - This is a vector of identical size as the gameObjects vector. It is filled with objects to be killed in their respective subvector as prespecified by the hierarchy of precedence.

# GameObject

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.

#### Member Functions

##### Public

GameObject() - Constructor. Initializes member variables to default values.

GameObject(std::string, sf::RenderWindow \*, GameData \*) - Constructor. Initializes member variables to default values with the exceptions of name, window, and gameData.

~GameObject() - Destructor.

virtual void update() - Performs GameObject updating. See flow of control and command for more.

virtual void levelPlay() - Performs GameObject updating for level play specific behavior. See flow of control and command for more.

virtual void levelTransition() - Performs GameObject updating for level transition specific behavior. See flow of control and command for more.

virtual void levelStart() - Primes the GameObject for level play. See flow of control and command for more.

virtual void levelEnd() - Primes the GameObject for level transition. See flow of control and command for more.

virtual void collision(GameObject \*) - Performs basic rectangle collision.

virtual void collideWith() - Determines what GameObjects it will test collision with.

virtual void collided(GameObject \*) - A reaction function that is called if there is collision.

void time() - The function for time limit testing.

void setTimeLimit(sf::Time) - Sets the time limit to the passed in value.

void startClock(sf::Time) - Sets the time limit to the passed in value and starts the clock.

void startClock() - Starts the clock.

sf::Time stopClock() - Stops the clock and returns time elapsed.

sf::Time getTimeElapsed() - Returns the time elapsed.

virtual void timeLimitPassed() - An overridable function that is called when the time limit has been passed. It is the GameObject’s reaction to that event.

void distance() - The function for distance testing.

void setPedometerLimit(float) - Sets the pedometer limit to the passed in value.

void startPedometer(float) - Sets the pedometer limit to the passed in value and starts the pedometer.

void startPedometer() - Starts the pedometer.

double stopPedometer() - Stops the pedometer and returns the distance elapsed.

float getDistanceElapsed() - Returns the distance elapsed.

virtual void distanceLimitPassed() - An overridable function that is called when the distance limit has been passed. It is the GameObject’s reaction to that event.

void moveLeft() - Moves the GameObject left. Used for keyboard input in Player and pathing in Monster.

void moveRight() - Moves the GameObject right. Used for keyboard input in Player and pathing in Monster.

void stopHorizontalVelocity() - Stops the GameObject from moving horizontally.

void stopVerticalVelocity() - Stops the GameObject from moving vertically.

void reverseDirectionHorizontal() - reverses the velocity horizontally.

void reverseDirectionVertical() - reverses the velocity vertically.

void jump() - Lets the GameObject jump. Changes the verticalAcceleration variable to be a predetermined number less than zero.

void gravity() - Implements a piecewise function for vertical acceleration. Uses the verticalAcceleration variable to determine where the GameObject is on the function.

int getVerticalAcceleration() - Returns the verticalAcceleration value for where the GameObject is in its jump.

void changePositionVertical(float) - Used in level teleportation from the bottom to the top and vice versa. Changes the vertical position by adding the passed in value to the current y position.

void setPosition(float, float) - Sets the position of the rectangle to be the passed in values in the form of (x, y).

int getDirection() - Returns the direction the GameObject is facing. 1 for right, -1 for left.

virtual void levelPathing(std::vector<int>, std:vector<int>) - Performs the interpretation functions for the GameObject after having been tested on the level bitmap. The passed in values are the bitmap collision values.

bool offTop() - Returns true if the GameObject is off the top of the RenderWindow.

bool offBottom() - Returns true if the GameObject is off the bottom of the RenderWindow.

virtual void render() - Rudimentary drawing. The rectangle is drawn onto the screen by the RenderWindow.

virtual void death() - Pushes itself to the GameData killList. Life is set to false.

void setName(std::string) - Sets the name to be the passed in value.

std::string getName() - Returns the name.

void setVelocity(sf::Vector2f) - Sets the velocity to be the passed in value.

void setVelocity(float, float) - Sets the velocity to be the passed in value in the form of (x, y).

void velocityToNextGridLine(bool) - Moves the GameObject to the closest gridline determined by velocity. Horizontal movement is done when a true is passed in. Vertical movement is done when a false is passed in.

sf::Vector2f getVelocity() - Returns the velocity.

void setTexture(sf::Texture) - Sets the texture of the rectangle to be the one passed in. Calls setTexture()

void setTexture(std::string) - Loads a new texture from a passed in filename. Sets it to the rectangle texture. Calls setTexture()

void setTexture() - The texture is applied to the rectangle and the rectangle’s size is set to be the texture’s size multiplied by the SCREEN\_MULTIPLIER constant.

sf::Texture getTexture() - Returns the texture.

sf::RectangleShape getRectangle() - Returns the rectangle.

void setAnimation(std::string) - Sets the animation to a string type.

bool isTransitioningLevels() - Returns the transition flag’s value.

bool isFriendly() - Returns the friendly variable’s value.

void setRenderWindow(sf::RenderWindow \*) - Sets the RenderWindow to a the passed in RenderWindow.

void setGameDataPTR(GameData \*) - Sets the GameData pointer to the passed in GameData pointer.

void flipTexture() - Flips the texture according to the direction variable.

void setLives(unsigned int lives) - sets the livesRemaining variable to the passed in value.

int getLivesRemaining() - returns the value of the livesRemaining variable.

#### Member Variables

##### Protected

const int SCREEN\_MULTIPLIER = 3 - Used in translating motion and size to the relative screen size.

const int BITMAP\_CONVERTER = SCREEN\_MULTIPLIER \* 8 - Used in converting actual positional coordinates to the preloaded bitmap.

std::string name - Stores the name of the GameObject.

sf::RenderWindow \*window - Stores the GameObject’s RenderWindow.

GameData \*gameData - Stores the GameData.

bool friendly - A flag for if the GameObject is on the Player’s side. Default is true.

bool transition - A flag for if the GameObject is in transition. Used in determining whether levelPlay() or levelTransition() should be called during update().

sf::Vector2f velocity - The velocity of the GameObject.

sf::Clock clock - The clock variable is used for the time based events. Keeps track of time from starting.

sf::Time timeElapsed - Stores the amount of time elapsed.

sf::Time timeLimit - Stores the time limit.

bool trackingTime - A boolean flag for determining if the GameObject should test for time.

float pedometer - The pedometer variable is used for distance based events. Keeps track of distance from starting.

float distanceElapsed - Stores the distance elapsed.

float distanceLimit - Stores the distance limit.

bool trackingDistance - A boolean flag for determining if the GameObject should test for distance.

int verticalAcceleration - A variable that places itself on the gravity piecewise function.

int direction - A variable that stores the direction the GameObject is facing.

sf::RectangleShape rectangle - Stores the GameObject’s rectangle for drawing.

sf::Texture texture - Stores the GameObject’s texture that is on the rectangle for drawing.

std::vector<Animation> animations - Stores the animations.

bool life - A boolean flag for if the GameObject is alive or dead.

int livesRemaining - holds the number of lives a GameObject has. Default is one.

## Level

The Level class is for the level.

#### Inheritance

Inherits from GameObject.

#### Member Functions

##### Public

Level() - Constructor

Level(std::string, sf::RenderWindow \*, GameData \*) - Constructor

~Level() - Destructor

void collision(GameObject \*) - Specifically designed to perform collision with itself. It has a specialized form of collision with a bitmap.

void update() - Calls the appropriate levelPlay or levelTransition function.

void timeLimitPassed() - Starts the transitioning the levels.

void distanceLimitPassed() - Whether the level is off the top of the RenderWindow, death or levelStart is called.

void levelTransition() - Moves the rectangle and tests the pedometer.

void levelStart() - Prime the players and generate monsters.

void levelPlay() - If the time is not being tracked, check for enemies. Otherwise, check for time.

void levelEnd() - Prime the level for transitioning.

void death() - Kills the level. Pushes it to GameData’s killList and its respective sublist.

##### Private

void bitmapMaker() - Loads the bitmaps for monster spawns and bitmap into the monsterSpawns and bitmap arrays.

void enemyCheck() - If there are no monsters in the GameData monsters subarray, start the clock for starting level transition.

#### Member Variables

##### Private

const int BITMAP\_WIDTH = 32 - Preset number of data values. Found by width / 8.

const int BITMAP\_HEIGHT = 26 - Preset number of data values. Found by width / 8.

int bitmap[32][26] - Holds the bitmap for collision checking.

int monsterSpawns[32][26] - Holds the bitmap for monster spawning.

## Player

The Player class is for the player.

#### Inheritance

Inherits from GameObject.

#### Member Functions

##### Public

Player() - Constructor

Player(std::string, sf::RenderWindow \*, GameData \*) - Constructor.

~Player() - Destructor

void update() - Calls the appropriate levelPlay or levelTransition function.

void collideWith() - Collides with the level, pickups, projectiles, and monsters.

void collided(GameObject \*) - Determines Player reaction to collision.

void pickedUp(GameObject \*) - Used for updating player from a pickup collison.

void levelEnd() - Prime the player for level transition.

void levelStart() - Prime the player for level play.

void levelPlay() - Appropriate gameplay functions are called.

void levelTransition() - Moves the player to the starting corner.

void death() - Kills the player. Pushes it to GameData’s killList and its respective sublist.

void timeLimitPassed() - The Skel\_Monsta is spawned.

## 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.

#### Inheritance

Inherits from GameObject.

#### Member Functions

##### Public

Monster() - Constructor. Sets instance variables to default values.

Monster(sf::RenderWindow \*, GameData \*) - Constructor. Sets instance variables to default values with the exception of window and gameData. They are set to their passed in counterparts.

~Monster() - Destructor.

void collideWith() - Determines the monster will collide with the level and projectiles.

void collided(GameObject \*) - The monster’s reaction to collision.

void captured(GameObject \*) - Primes the monster to appear to be in a bubble.

void death() - Kills the monster. Pushes it to GameData’s killList and its respective sublist.

void levelStart() - Primes the monster for level play.

void levelPlay() - The level play update sequence.

void distanceLimitPassed() - Determines that the monster is in proper placement for pathing.

#### Member Variables

##### Protected

bool noLevelCollision - A boolean flag that prevents level collision while the monsters move down to their starting location when true.

bool contained - A boolean flag that means the monster is contained a bubble when true.

### Banebou

This class represents the Banebou monster. Full implementation has yet to occur.

#### Inheritance

Inherits from GameObject and from Monster.

#### Member Functions

##### Public

Banebou() - Constructor. Initializes the name to be “Banebou”. Initializes the RenderWindow and GameData objects as nullptr.

Banebou(sf::RenderWindow \*, GameData \*) - Constructor. Initializes the name to be “Banebou”. Initializes the RenderWindow and GameData objects with the passed values.

~Banebou() - Destructor.

### Drunk

This class represents the Drunk monster. Full implementation has yet to occur.

#### Inheritance

Inherits from GameObject and from Monster.

#### Member Functions

##### Public

Drunk() - Constructor. Initializes the name to be “Drunk”. Initializes the RenderWindow and GameData objects as nullptr.

Drunk(sf::RenderWindow \*, GameData \*) - Constructor. Initializes the name to be “Drunk”. Initializes the RenderWindow and GameData objects with the passed values.

~Drunk() - Destructor.

### Hidegons

This class represents the Hidegons monster. Full implementation has yet to occur.

#### Inheritance

Inherits from GameObject and from Monster.

#### Member Functions

##### Public

Hidegons() - Constructor. Initializes the name to be “Hidegons”. Initializes the RenderWindow and GameData objects as nullptr.

Hidegons(sf::RenderWindow \*, GameData \*) - Constructor. Initializes the name to be “Hidegons”. Initializes the RenderWindow and GameData objects with the passed values.

~Hidegons() - Destructor.

### Invader

This class represents the Invader monster. Full implementation has yet to occur.

#### Inheritance

Inherits from GameObject and from Monster.

#### Member Functions

##### Public

Invader() - Constructor. Initializes the name to be “Invader”. Initializes the RenderWindow and GameData objects as nullptr.

Invader(sf::RenderWindow \*, GameData \*) - Constructor. Initializes the name to be “Invader”. Initializes the RenderWindow and GameData objects with the passed values.

~Invader() - Destructor.

### Mighta

This class represents the Mighta monster. Full implementation has yet to occur.

#### Inheritance

Inherits from GameObject and from Monster.

#### Member Functions

##### Public

Mighta() - Constructor. Initializes the name to be “Mighta”. Initializes the RenderWindow and GameData objects as nullptr.

Mighta(sf::RenderWindow \*, GameData \*) - Constructor. Initializes the name to be “Mighta”. Initializes the RenderWindow and GameData objects with the passed values.

~Mighta() - Destructor.

### Monsta

This class represents the Monsta monster. Full implementation has yet to occur.

#### Inheritance

Inherits from GameObject and from Monster.

#### Member Functions

##### Public

Monsta() - Constructor. Initializes the name to be “Monsta”. Initializes the RenderWindow and GameData objects as nullptr.

Monsta(sf::RenderWindow \*, GameData \*) - Constructor. Initializes the name to be “Monsta”. Initializes the RenderWindow and GameData objects with the passed values.

~Monsta() - Destructor.

### Pulpul

This class represents the Pulpul monster. Full implementation has yet to occur.

#### Inheritance

Inherits from GameObject and from Monster.

#### Member Functions

##### Public

Pulpul() - Constructor. Initializes the name to be “Pulpul”. Initializes the RenderWindow and GameData objects as nullptr.

Pulpul(sf::RenderWindow \*, GameData \*) - Constructor. Initializes the name to be “Pulpul”. Initializes the RenderWindow and GameData objects with the passed values.

~DrPulpulunk() - Destructor.

### Skel\_Monsta

This class represents the Skel\_Monsta monster. Full implementation has yet to occur.

#### Inheritance

Inherits from GameObject and from Monster.

#### Member Functions

##### Public

Skel\_Monsta() - Constructor. Initializes the name to be “Skel\_Monsta”. Initializes the RenderWindow and GameData objects as nullptr.

Skel\_Monsta(sf::RenderWindow \*, GameData \*) - Constructor. Initializes the name to be “Skel\_Monsta”. Initializes the RenderWindow and GameData objects with the passed values.

~Skel\_Monsta() - Destructor.

### SuperDrunk

This class represents the SuperDrunk monster. Full implementation has yet to occur.

#### Inheritance

Inherits from GameObject and from Monster.

#### Member Functions

##### Public

SuperDrunk() - Constructor. Initializes the name to be “SuperDrunk”. Initializes the RenderWindow and GameData objects as nullptr.

SuperDrunk(sf::RenderWindow \*, GameData \*) - Constructor. Initializes the name to be “SuperDrunk”. Initializes the RenderWindow and GameData objects with the passed values.

~SuperDrunk() - Destructor.

### Zenchan

This class represents the Zenchan monster. Full implementation has yet to occur.

#### Inheritance

Inherits from GameObject and from Monster.

#### Member Functions

##### Public

Zenchan() - Constructor. Initializes the name to be “Zenchan”. Initializes the RenderWindow and GameData objects as nullptr.

Zenchan(sf::RenderWindow \*, GameData \*) - Constructor. Initializes the RenderWindow and GameData objects with the passed values.

~Zenchan() - Destructor.

## Pickup

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

#### Inheritance

Inherits from GameObject.

#### Member Functions

##### Public

Pickup() - Constructor. Initializes default values.

Pickup(sf::RenderWindow \*, GameData \*) - Constructor. Initializes default values with the exception of window and gameData. They are initialized to their passed in counterparts.

~Pickup() - Destructor.

void collideWith() - Determines that the pickup collides with the level.

void collided(GameObject \*) - The pickup’s response to collision.

void death() - Kills the pickup. Pushes it to GameData’s killList and its respective sublist.

## Projectile

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

#### Inheritance

Inherits from GameObject.

#### Member Functions

##### Public

Projectile() - Constructor. Initializes default values.

Projectile(sf::RenderWindow \*, GameData \*) - Constructor. Initializes default values with the exception of window and gameData. They are initialized to their passed in counterparts.

~Projectile() - Destructor.

void collideWith() - Determines that the projectile collides with the level.

void collided(GameObject \*) - The projectile’s response to collision.

void death() - Kills the projectile. Pushes it to GameData’s killList and its respective sublist.

### Bubble

#### Inheritance

Inherits from Projectile.

#### Member Functions

##### Public

Bubble(sf::RenderWindow \*, GameData \*, Player \*) -

~Bubble() -

void levelPlay() -

void distanceLimitPassed() -

void collided(GameObject \*) -

void pop() - pops the bubble.

std::vector<bool> levelPathing(std::vector<int>, std::vector<int>) - Performs pathing functions.

#### Member Variables

##### Public

GameObject \*monsterContained - holds the monster being contained.

## PointText

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

#### Inheritance

Inherits from GameObject.

#### Member Functions

##### Public

PointText() - Constructor.

PointText(sf::RenderWindow \*, GameData \*, GameObject \*, unsigned int, bool) - Constructor. Sets the window and gameData to their passed in counterparts. The GameObject pointer, unsigned int, and bool determine the type, size, and placement of the PointText.

~PointText() - Destructor.

void death() - Kills the PointText. Pushes it to GameData’s killList and its respective sublist.

# Animation

The animation class takes in a texture pointer and the dimensions of the image then it moves a uvRect across the screen on a set timer in order to create an animation. The uvRect is used to apply the dimensions for a texture then it is set to a normal rectangle.

#### Member Functions

##### Public

Animation(sf::Texture\* texture, sf::Vector2u imageCount, float switchTime, int row, bool faceRight = true) - Constructor

~Animation() - Destructor

void update(float deltaTime) - updates the uvRect on the spritesheet based on a timer

#### Member Variables

##### Public

sf::IntRect uvRect

sf::Texture\* texture

std::string ID

##### Private

sf::Vector2u imageCount

sf::Vector2u currentImage

int row

float totalTime

bool faceRight

float switchTime

# InputManager

InputManager performs interfacing functions for easy event handling.

#### Member Functions

##### Public

InputManager() - Constructor.

~InputManager() - Destructor.

void update(sf::Event) - event\_m is set to the passed in value.

bool keyPressed(sf::Keyboard::Key) - Returns true if the key is pressed.

bool keyPressed(std::vector< sf::Keyboard::Key >&) - Returns true if all keys are pressed.

bool keyReleased(sf::Keyboard::Key) - Returns true if the key is released.

bool keyReleased(std::vector< sf::Keyboard::Key >&) - Returns true if all keys are released.

bool keyDown(sf::Keyboard::Key) - Returns true if the key is down.

bool keyDown(std::vector< sf::Keyboard::Key >&) - Returns true if all the keys are down.

#### Member Variables

##### Private

sf::Event event\_m - Holds the current event.

##### 

Flows and Hierarchies

# Hierarchies of precedence and reaction

Each GameObject affects the rest, but not in all of the same order. Careful updating assures that each GameObject is in the proper state when the others test with it. There are currently six GameObject derivatives: Level, Player, Monster, Pickup, Projectile, and PointText. Here are the order and reasons for the order.

## Hierarchy of precedence

It is the order of Level, Player, Monster, Pickup, Projectile, and PointText. The order of precedence is found by looking at what has power over what. Level is the active level, so when it transitions to the next, Monster, Pickup, Projectile, and PointText objects are automatically deleted. Player’s levelEnd() function is called. Level also starts new levels by making new Monsters and calling Player’s levelStart() function. Player is the next in order. It affects Monster, Pickup, and Projectile. Monster only affects Projectile. Pickup affects PointText. PointText does not affect anything. Pickup is before Projectile because of personal preference. The hierarchy of precedence is used for ordering of GameData subvectors.

## Hierarchy of reaction

The hierarchy of reaction is the processing order of the game. It is semi-related to the hierarchy of precedence. The order is Level, PointText, Pickup, Projectile, Monster, Player. It is found by what events have to occur for the next GameObject to be able to accurately test the GameObjects in the hierarchy of precedence. Level has overarching control over every other GameObject, so it is first in the hierarchy of reaction. The level has to exist and be playable for any other objects to work properly. Second is PointText. PointText does not affect anything, nor does anything need it to update. It could be at the end, but by personal preference, it is second. Third is Pickup. Pickups need to exist for the Player to be able to pick them up. Fourth is Projectile. Projectiles can either be friend or foe. Friendly ones can affect both Monsters and Players. Foes can only affect Players. This distinction is made in Monster’s collideWith(). The premise is that the projectile has to be update for Monsters or Players to react to them accurately. Fifth is Monster. Monsters have to be updated for Players to collide with their accurate position. Last is the Player. It needs everything else to be updated for accurate play. That is the hierarchy of reaction and determines the updating order.

# Hierarchy of Monster Order

This is based on order of appearance and is the following:

1. Zenchan
2. Mighta
3. Monsta
4. Pulpul
5. Banebou
6. Hidegons
7. Drunk
8. Invader
9. SuperDrunk

This hierarchy is used for monster spawns. An image is read and corresponding rgb values are attributed to each monster for spawning.

1. Zenchan RGB: (1, 1, 1)
2. Mighta RGB: (2, 2, 2)
3. Monsta RGB: (3, 3, 3)
4. Pulpul RGB: (4, 4, 4)
5. Banebou RGB: (5, 5, 5)
6. Hidegons RGB: (6, 6, 6)
7. Drunk RGB: (7, 7, 7)
8. Invader RGB: (8, 8, 8)
9. SuperDrunk RGB: (9, 9, 9)

This is used in the Bitmap Maker and LevelStart().

# Flow of Control and Command

The following is the diagram of game flow and command.

