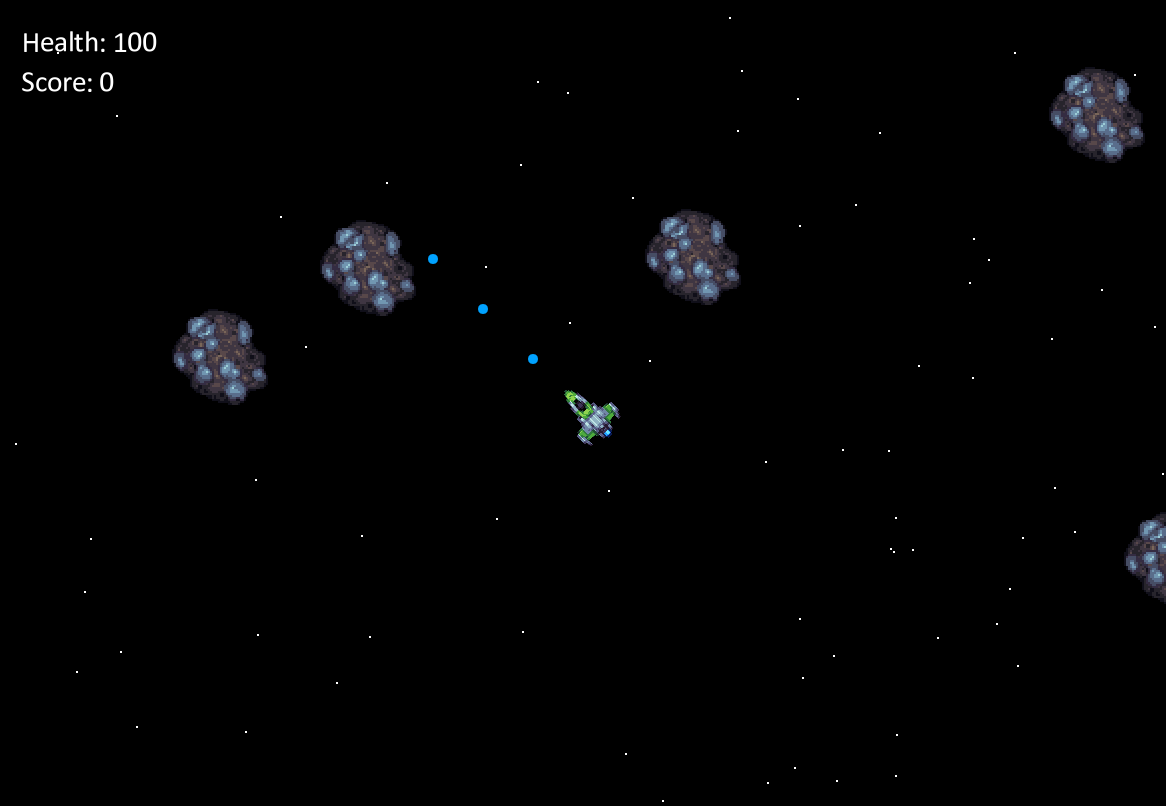
**Group Game**

The aim of this exercise is to work as a group to create a simple space shooter similar to asteroids:



The game will feature a ship controllable by the player. A turret on the ship rotates to follow the mouse and can shoot.

There will be asteroids randomly floating around which can be destroyed after a few shots, and which will destroy the player if too many collisions occur.

Each asteroid destroyed should give the player points.

Enemies occasionally spawn and chase the player.

The game should feature a menu and a game over screen, with the ability to restart the game from the game over screen.

The purpose of this project is to practice working in a group and to get used to working to a common interface: when multiple people are making different classes which all have to call each other’s functions, how do you know the name of a function to call if the class hasn’t been written yet? The answer is good planning.

The teacher will be acting as both the designer and producer for this project and will make any gameplay and creative decisions which you must then implement, as well as making sure everyone is working on the correct parts of the project.

**Planning out classes**

This document describes each class to aid in planning. It only lists public and protected member functions and variables since those are the ones that other people on the team need to know about. You must stick to the names and functions described so that you don’t break the project.

You may define your own private variables and functions for your class since those won’t be used by others so you can call them whatever you like.

You may also create additional classes that are used internally by your class, however you should check with the designer (the teacher) before adding new game features.

Some of the game’s technical features:

* A state machine will keep track of the game state and will be able to change between a menu state, a game state, and a game over state.
* All movable and collidable objects (Player, Asteroids, Bullets, etc) will inherit from a common base class called GameObject.
* A collision manager will calculate all collisions so that the individual objects don’t need to.
* A bullet manager will use pooling to keep a pool of bullets created at the start and recycled so that new bullets don’t need to be created during the game.
* A GUI that will display the player’s health and score.
* A Texture Manager that will keep track of which textures have been loaded and recycle them as needed so that a new one doesn’t need to be loaded for every object.

# Tasks

Each person is assigned a number of classes to create. When everyone is finished, the classes will be combined together and should form a working project. So long as no one has changed the name of a public function, nothing should need to be changed to make all the classes work together.

The classes will be divided up as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Class** | **Description** | **Difficulty** |  |
| StateMachine | Handles the switching of states. | Easy | Steven |
| BaseState | Base class for all states. | Easy | Lena |
| MenuState | A state that draws and handles input for the main menu. | Medium | Thomas |
| GameState | The main game state. Runs the game. Creates and updates the level and player. | Easy | Daniel Stewart |
| GameOverState | A state that draws and handles input for the game over screen. | Medium | Lena |
| GameObject | A base object for all other objects. | Medium | Behnam |
| Actor | A base class for all objects that interact, have collision, and can die. Inherits from GameObject. | Medium | Daniel C |
| Level | Loads, updates, and draws all level objects. Inherits from GameObject | Medium | Connor Li |
| Camera | A singleton class that wraps the camera for easy access | Easy | Connor Young |
| Player | The player object. Inherits from Actor. | Medium | Chris |
| Turret | Turret on the player. Rotates to follow the mouse. Can shoot. Inherits from Actor. Attached to player through a matrix hierarchy. Has an ObjectPool for spawning bullets. | Medium | Chris |
| BulletManager | Stores a pool of bullets and functions as an ObjectPool | Medium | Seth |
| Bullet | One of the player’s bullets. Inherits from Actor. | Easy | Connor Young |
| Rock | An asteroid that floats around. Can be shot and destroyed by the player. Can harm the player if it hits them. Inherits from Actor. | Medium | Connor Li |
| Star | A star displayed in the background. Inherits from GameObject. | Easy | Daniel C |
| Enemy | Spawns randomly, chases the player and attempts to collide. Can be destroyed by the player. Inherits from Actor. | Medium | James |
| Health Pickup | Gives the player health if touched and is then destroyed. Inherits from Actor. | Medium | Grant |
| CollisionManager | Handles all collision checks. | Hard | Lachlan |
| Collider | Stores the collider for an Actor which is used by CollisionManager to calculate if two objects are colliding. | Medium | Behnam |
| TextureManager | Loads textures and keeps track of all pointers to textures. | Hard | Troy |
| GUI | Displays the player’s health and score. | Easy | Thomas |

# Project Dependencies

Work on some classes is dependent on others: for example, it is difficult for a programmer to write an Actor class before there is a GameObject class since Actor needs to inherit from GameObject. These work dependencies often need to be factored in when planning a project.

An easy way to do this is to create the header files with empty public members first and check these in to version control so others can start using them while you create the source files.

The dependencies are:

* GameObject
  + Actor
    - Player
    - Turret
    - Bullet
      * BulletManager
    - Rock
    - Enemy
    - HealthPickup
  + Star
  + Level
* Collider
  + CollisionManager
* TextureManager
* BaseState
  + StateMachine
  + MenuState
  + GameState
  + GameOverState
* GUI
* Camera

# Class: StateMachine

A finite state machine to control the overall game flow.

* Is created by Application2D.
* Creates MenuState, GameState and GameOverState.

This class should implement a finite state machine which creates three states to control the game flow. It starts with the MenuState as the active one, changes to GameState when a new game is started, and goes to GameOverState when the player loses.

The finite state machine is just a manager for these states, creating them, providing the function to change states and cleaning up afterwards. It doesn’t do any of the game logic itself, that is handled inside the individual states.

## **Public Functions:**

**Constructor:** Initialize everything. Create the three states.

**Destructor:** Clean up everything. Delete the three states.

**Update:** Call Update() on the active state.

* Params: float for detatime

**Draw:** Call Draw() on the active state.

* Params: Pointer to a Renderer2D

**ChangeState:** Change to the specified state: first call Exit on the current state to tell it that its time is over. Then call Enter on the new state and set the new state to be active so it gets updated next time the Update() function is called. This shouldn’t create states, they should all be created from the start, it just changes which one is active.

## **Public Enum:**

StateMachine.h should define or include a file with defines an enum for each state:

* ESTATE\_MENU
* ESTATE\_GAME
* ESTATE\_GAMEOVER

# Class: BaseState

An abstract base class for all states.

* MenuState, GameState and GameOverState will all inherit from this class.
* This class is abstract: meaning it only contains pure virtual functions and doesn’t implement any code.

## **Public Functions:**

**Constructor:** Empty.

**Destructor:** Empty.

**Enter (pure virtual):** This will be called when the state is entered.

**Update (pure virtual):** This will be called each frame.

* Params: float for detatime, pointer to a StateMachine

**Draw (pure virtual):** This will draw all the graphics for the menu.

* Params: Pointer to a Renderer2D

**Exit (pure virtual):** This will be called when the state is exited.

# Class: MenuState

A state for the main menu.

* Inherits from BaseState

This class should implement a main menu featuring a game title and at least buttons to start the game and quit. Could also include a button to a credits screen or options.

## **Public Functions:**

**Constructor:**

**Destructor:** Delete all assets used in menu and clean up everything.

**Enter:** Called when the state starts.Load all the images needed for the menu. Initialize everything.

**Update:** Process input and when the player has clicked the start button, you can start the game using **pStateMachine->ChangeState(ESTATE\_GAME);**

* Params: float for detatime, pointer to a StateMachine

**Draw:** Draw all the graphics for the menu.

* Params: Pointer to a Renderer2D

**Exit:** Called when the state is exited. Delete all assets used in menu and clean up everything.

# Class: GameState

A state for the game.

* Inherits from BaseState

This class runs the game, it handles the creation, update, drawing and destruction of most major systems.

## **Public Functions:**

**Constructor:**

**Destructor:** Destroy the Level and GUI.

**Enter:** Load andInitialise the Level and GUI.

**Update:** Update the Level. Check if the player’s health is zero, if so change to the GameOverState.

pStateMachine->ChangeState(ESTATE\_GAMEOVER);

* Params: float for detatime, pointer to a StateMachine

**Draw:** Call Draw() on the Level. Call the Draw() function on the GUI.

* Params: Pointer to a Renderer2D

**Exit:** Destroy the Level and GUI.

# Class: GameOverState

A state for displaying a game over screen to the player when they die.

* Inherits from BaseState
* Should display the player’s score and optionally a table of high scores.
* Should have the ability to play again.

This class should implement a game over screen listing the players score. The appearance and controls are up to you but there should be a way to start a new game.

## **Public Functions:**

**Constructor:**

**Destructor:** Clean up everything.

**Enter:** Called when the state is entered. Initialize everything need for this screen.

**Update:** Called each frame to update input.

To start the game again, call **pStateMachine->ChangeState(ESTATE\_GAME);**

* Params: float for detatime, pointer to a StateMachine

**Draw:** Called each frame to draw everything in this state.

* Params: Pointer to a Renderer2D

**Exit:** Called when the state is exited. Clean up everything.

# Class: GameObject (Base class)

This class will act as a base class for all other level objects, including all Actors (player, rocks, bullets, power ups, and enemies), as well as the level and the decorative Star object. It will handle updating and drawing the object although these functions can be overridden by inheriting classes. It will also act as a scene graph node and track parent/child relationships.

## **Public Functions:**

**Constructor:** Needs to initialise all variables.

* Params:
  + Starting position. Should be stored in the global transform.

**Destructor:**

**Update (virtual):** Update this object’s position based on its velocity, transform the node’s local transform by its parent’s global transform.

If an object has m\_bWrapAndRespawn set to true and is too far from the camera (way off-screen) it should wrap around the level and appear the same distance on the other side of the camera, that way as the camera moves around it will never run out of objects in the level, they will teleport to keep up but never when on the screen. If the object is not visible when this happens, they can also be respawned at that time by setting m\_bVisible back to true. This way the game will never run out of objects in the level. It will also need to call update on all child objects

* Param: A float for deltatime.

**Draw (virtual):** Draw the object if m\_bVisible is true. Use the global matrix to determine the objects transform.

* Params: A pointer to a Renderer2D

**OnCollision (virtual):** This function will be called by the CollisionManager on both objects involved in a collision. The base version may not need to do anything, collision response could be left to each inherited class.

* Params: Will need to take in a pointer to the other object involved in a collision, also as much information about the collision as there is available (this will depend on type of collision detection used), might include impact point, impact speed, collision normal, etc.

**Getters/Setters:** Will need Get and Set functions for most variables in this class, including things like SetPosition(), SetRotation(), SetColour(), SetParent(), SetHealth(), etc.

## **Variables:**

These are just some of the variables this class will need:

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| m\_v2Velocity | Vector2 | Stores the object’s current velocity. |
| m\_fDrag | float | How quickly object should slow down once acceleration stops. Represents wind resistance, friction, etc. |
| m\_pTexture | Texture\* | A pointer to a texture to be drawn by the object. |
| m\_m3LocalMatrix | Matrix3 | Stores the object’s local transform relative to its parent. |
| m\_m3GlobalMatrix | Matrix3 | Stores the object’s transform relative to the world’s origin. |
| m\_pParent | GameObject\* | Pointer to a parent object. |
| m\_Children | List<GameObject\*> | List of pointers to child objects. Could be an std::list or your own linked list class. |
| m\_bVisible | bool | Is the object visible/in existence. Set to false to stop updating and rendering the object without needing to delete the object. |
| m\_bWrapAndRespawn | bool | When the object leaves the screen, should it wrap around and get placed on the other side so it enters the screen again? |

# Class: Actor (Base Class)

This class inherits from GameObject and serves as the base class for all objects that move, have collision, and can be destroyed. It will store the object’s health and react to collision between objects.

## **Public Functions:**

**Constructor:** Needs to initialise all variables.

* Params: Starting position. Should be stored in the global transform.

**Destructor:**

**Update (virtual):** Call the base classes Update, and then update the collider object to recalculate bounding boxes and similar collision data

* Param: A float for deltatime.

**OnCollision (virtual):** This function will be called by the CollisionManager on both objects involved in a collision. The base version may not need to do anything, collision response could be left to each inherited class.

* Params: Will need to take in a pointer to the other object involved in a collision, also as much information about the collision as there is available (this will depend on type of collision detection used), might include impact point, impact speed, collision normal, etc.

## **Variables:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| m\_pCollider | Collider | Instance of a class containing data about how the objects should collide. |
| m\_nHealth | int | Current health. |
| m\_nMaxHealth | int | Maximum health. |

# Class: Level

This class will be created by the GameState and will inherit from GameObject. It will act as the root node in the scene graph with other GameObjects as its children.

* The level should create all level objects.
* The level should store all the objects it creates in its list of children.
* The level should delete all the objects it creates when its destructor is called.
* m\_bWrapAndRespawn in the base class should be set to false.

Use whatever method you choose to lay out the level, you could place objects randomly or use some kind of procedural level generation (research them), or even read it in from a file created with some kind of level editor. The method you use is up to you. Make sure objects that are created don’t overlap though!

## **Public Functions:**

**Constructor:** Create all objects and add them as children of the level.

**Destructor:** Delete all objects.

# Class: Camera

A singleton class that simply stores the camera’s position for easy access by both the Player and Application2D.

* Is created by Application2D.
* Is a singleton.

The camera needs to follow the player. In the 2D version of the bootstrap, the camera is just two floats, an X and a Y position inside the Renderer2D so it can only be set using a pointer to the Renderer2D. The Camera needs to be set with the Player’s position but each class only has access to the renderer in their Draw function, to make matters worse, the position can only be changed before the renderers Begin() function is called.

All of this makes it very difficult to set the camera’s position.

So this class aims to solve that: it will simply store two floats, an X and a Y and provide functions to get and set them. During the Player’s Update() function, it will access this class and set those values. During the Application2D’s next Draw() function, it will read the two values and use them to set the camera before it starts drawing everything. This class really just facilitates communication between the two.

## **Public Functions:**

**GetInstance:** Static function that returns the internal static instance of the singleton class.

* Returns a pointer to this class.

**SetPosition:** Set’s the internal position to where the Camera should be.

* Params: A Vector2 storing the X and Y position.

**SetPosition:** An overloaded version of the SetPosition function so that floats can be passed in.

* Params: Two floats, one for X, one for Y.

**GetPosition:** Get the current position stored in this class

* Returns a Vector2.

# Class: Player

This class will be the object controllable by the player.

* Inherits from Actor
* Handles player input and movement.
* m\_bWrapAndRespawn in the base class should be set to false.

## **Public Functions:**

**Constructor:** Needs to initialise the object. Requests a texture from TextureManager.

**Destructor:** Clean up. Don’t delete the texture, it belongs to the TextureManager.

**Update (overload):** Use the Bootstrap’s Input class to check for input and move the player. You should use the GameObject’s functions to set and get velocity, position, and facing as needed.

Make sure to call Actor’s base update function so that the internal matrices still get updated.

Call the Camera’s SetPosition function so that it follows the player.

Finally tell the GUI what the player’s health currently is so it can be displayed on the screen.

**OnCollision (overload):** What happens to the player when they collide with an asteroid? An enemy? A power up? This function will be called by CollisionManager whenever any type of collision occurs. One of the parameters will be a pointer to the other object that collided. Asteroids and enemies should damage the player, powerups should restore health.

# Class: Turret

This class will represent a turret mounted on the player’s ship. It will be a child of the Player, transformed using a matrix hierarchy.

* Inherits from Actor
* Child of Player.
* Rotates to point at the mouse.
* Has a BulletManager.
* Shoots a bullet when left button is pressed by calling function on the BulletManager.
* m\_bWrapAndRespawn in the base class should be set to false.

## **Public Functions:**

**Constructor:** Needs to initialise the object. Requests a texture from TextureManager.

**Destructor:** Clean up. Don’t delete the texture, it belongs to the TextureManager.

**Update:** Use the Bootstrap’s Input class to get the mouse position and rotate the turret to point at the mouse. You should use the GameObject’s functions to set the rotation as needed.

Make sure to call Actor’s base update function so that the internal matrices still get updated.

# Class: BulletManager

This class will be created by the Turret and will store a pool of bullets for the player(s) to fire.

* Creates a pool of bullets.
* All bullets should be created at the start.
* Bullets should be recycled after being used.

## **Public Functions:**

**Constructor:** Needs to initialise the object.

* Create all the bullets and store them in an array or list.

**Destructor:** Clean up. Destroy all the bullets.

**ShootBullet:** This function will be called by the Turret when a new bullet is to be fired. The BulletManager should look through its array of bullets and find one that is not currently active. Once one has been found, call that bullet’s Shoot() function to fire it.

* Param: Will need to accept a start position for where the bullet should spawn, and a direction indicating which way the bullet should go. These are just passed on to the Bullet’s Shoot function.

# Class: Bullet

A bullet that the player can shoot to destroy asteroids

* Inherits from Actor.
* Bullets should have a limited lifetime so they don’t keep going forever. They should vanish either when they leave the screen or after a set period of time, whichever you prefer.
* m\_bWrapAndRespawn in the base class should be set to false.

## **Public Functions:**

**Constructor:** Needs to initialise the object.

* Request a texture from the TextureManager.

**Destructor:** Clean up. Don’t delete the Texture, it belongs to the TextureManager.

**Shoot:** The shoot function is called when this bullet is being fired. It should set the bullet’s initial position and velocity and set it to be visible so it gets rendered.

* Param: Will need to accept a start position for where the bullet should spawn, and a direction indicating which way the bullet should go.

**OnCollision (overload):** What happens when the bullet collides with something? It should probably disappear. Can do this just by setting the baseclass’s m\_bVisible to false.

# Class: Rock

This class will represent a single Asteroid in the game.

* Inherits from Actor
* Handles the rocks movement and collision.
* m\_bWrapAndRespawn in the base class should be set to true.

## **Public Functions:**

**Constructor:** Needs to initialise the object.

* Request a texture from the TextureManager.
* Use the BaseClass’s SetVelocity function to give the rock a random velocity.
* Set the base class’ health to some value to represent how hard the rock is to destroy.

**Destructor:** Clean up. Don’t delete the Texture, it belongs to the TextureManager.

**OnCollision (overload):** What happens to the asteroid when it collides? It should take damage and it should also bounce off other large objects.

This is also where you’d detect that a bullet hit the asteroid and it should take damage.If the rock gets destroyed by the player, use the GUI class’ AddScore() function to give the player points.

When destroying the asteroid, don’t actually delete it. Just set the baseclass’s m\_bVisible to false so it is no longer drawn.

# Class: Star

A tiny star to draw in the background to add some decorative detail to the level.

* Inherits from GameObject
* m\_bWrapAndRespawn in the base class should be set to true.

## **Public Functions:**

**Constructor:** Needs to initialise the object.

* Request a texture from the TextureManager.

**Destructor:** Nothing needed. Don’t delete the Texture, it belongs to the TextureManager.

# Class: Enemy

This class will represent a single enemy in the game.

* Inherits from Actor
* The enemy should remain still until it gets within a certain radius of the player then it should pursue and attempt to collide. The radius should be larger than the screen so the player never sees them sitting still, but not so large that all the enemies in the level come at once.
* m\_bWrapAndRespawn in the base class should be set to true.

## **Public Functions:**

**Constructor:** Needs to initialise the object.

* Request a texture from the TextureManager.
* Set the base class’ health to some value to represent how hard the enemy is to destroy.

**Destructor:** Clean up. Don’t delete the Texture, it belongs to the TextureManager.

**Update (overload):** When the enemy gets within a radius of the player, use a Pursue steering behaviour to try and collide with the player but should steer to avoid rocks.

Make sure to call Actor’s base update function so that the internal matrices still get updated.

**OnCollision (overload):** If the enemy collides with the player, the enemy is destroyed and the player takes damage.

The enemy should steer to avoid rocks.

If the player shoots the enemy, the enemy loses life and when its life reaches zero it is destroyed and the player should gain points using the GUI class’s AddScore() function.

When destroying the enemy, don’t actually delete it. Just set the baseclass’s m\_bVisible to false so it is no longer drawn.

# Class: Health Pickup

This class will represent a single health pickup in the game.

* Inherits from Actor.
* Floats around slowly and if touched by the player gives health.
* m\_bWrapAndRespawn in the base class should be set to true.

## **Public Functions:**

**Constructor:** Needs to initialise the object.

* Request a texture from the TextureManager.
* Use the BaseClass’s SetVelocity function to give the health pickup a random small velocity so it slowly floats around.

**Destructor:** Clean up. Don’t delete the Texture, it belongs to the TextureManager.

**OnCollision (overload):** If the player collides with the health pickup, the pickup is destroyed and the player gains health.

If the health pickup collides with rocks or enemies, they should bounce off without damaging either.

If the player shoots the health pickup it should be destroyed.

When destroying the pickup, don’t actually delete it. Just set the baseclass’s m\_bVisible to false so it is no longer drawn.

# Class: CollisionManager

A class used to calculate collisions between all objects.

All Actors which could collide must be added to the CollisionManager where they should be stored in a list/array/container of some kind. Collision can then be checked between all objects every frame. Actors which have no collision don’t need to be added to the CollisionManager.

* Could be a Singleton since there will only be one and it will need to be accessed everywhere.
* Should keep a list of all Actors.

## **Public Functions:**

**Constructor:** Initialise whatever storage is going to be used to store all the Actors.

**Destructor:** Delete the storage. Don’t delete the Actors it contains, you don’t own them. Remember that only the class that creates the object should delete it.

**AddObject:** Add a passed in Actor to the internal list.

* Params: Pointer to an Actor

**RemoveObject:** Search through the internal list to find the specified GameObject and remove it from the list.

* Params: Pointer to the Actor to be removed.

**Update:** Loop through all objects and test collision between them. If collision is detected, call the OnCollision function on both objects, they then decide what to do about it, that’s not the CollisionManager’s responsibility. The type of collision detection you use is up to you, some are harder than others:

* Easy but crude: AABB checks.
* Hard but accurate: Separated Axis Theorem.
* Very Hard but efficient: GJK.

Collision rules:

* Objects that are not visible should never collide (m\_bVisible in GameObject).
* Objects should never collide with themselves.
* Objects should never collide with objects on the ECOLLISIONLAYER\_NONE layer.
* ECOLLISIONLAYER\_PLAYER shouldn’t collide with ECOLLISIONLAYER\_BULLET.
* ECOLLISIONLAYER\_ENEMY shouldn’t collide with ECOLLISIONLAYER\_ROCK.

All Actors have a Collider which contains any information about the objects bounds, shape, etc that might be needed for collision.

# Class: Collider

A Collider stores the bounds of an object, for use by the CollisionManager. The bounding box will need to be updated every frame by the Actor so that it stays up to date as the object moves and rotates.

The variables in the Collider can be public since this class is just a container.

## **Public Functions:**

**Constructor:** Takes in two Vector2s to set the initial min and max of the bounding box.

* Params: Two Vector2s for min and max offset.

**Destructor:**

**UpdateBounds:** Takes in the GameObject’s matrix3 which it then uses to update the bounding box. This function will be called by Actor’s Update().

* Params: Matrix3 for the GameObject’s transformation.

**SetLayer:** Sets which layer the object is on.

* Params: The layer as an ELayer.

## **Public Variables:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| m\_v2MinOffset | Vector2 | The initial offset of the minimum corner of the bounding box, does not include object’s position or rotation. Should be set in Constructor.  For example if an object is 64x64 pixels, this would be set to -32, -32 regardless of objects current transform. |
| m\_v2MaxOffset | Vector2 | The initial offset of the maximum corner of the bounding box, does not include object’s position or rotation. Should be set in Constructor.  For example if an object is 64x64 pixels, this would be set to 32, 32 regardless of objects current transform. |
| m\_v2Min | Vector2 | The current minimum corner of the bounding box, including the objects transformation. Should be calculated every frame by UpdateBounds() by multiplying the objects transformation with m\_v2MinOffset. |
| m\_v2Max | Vector2 | The current maximum corner of the bounding box, including the objects transformation. Should be calculated every frame by UpdateBounds() by multiplying the objects transformation with m\_v2MaxOffset. |
| m\_eLayer | ELayer | The collision layer for this object. Used to identify the object to the CollisionManager |

## **Collision Layers:**

m\_eLayer is an enum which stores which collision layer the object is on.

The layers should be defined in the Collider.h as:

enum ELayer

{

ECOLLISIONLAYER\_NONE,

ECOLLISIONLAYER\_PLAYER,

ECOLLISIONLAYER\_BULLET,

ECOLLISIONLAYER\_ROCK,

ECOLLISIONLAYER\_ENEMY,

ECOLLISIONLAYER\_HEALTH

};

# Class: TextureManager

A singleton class used to load textures. This class exists to improve loading speed and efficiency. The Bootstrap normally loads each texture individually, which means that if you want 100 identical rocks, you need to load the same rock texture 100 times.

Not only is this a huge waste of memory, it’s also very slow.

Instead, the other classes should use this TextureManager to load the texture: The TextureManager should keep a list of pointers to all the textures that have already been loaded. When another class attempts to load a texture, the TextureManager should check if the desired texture already exists, if so just return the pointer we already have, if it doesn’t then load the new texture and add it to the list.

This ensures that each texture only exists once but is used by many objects.

The TextureManager should be handling all the calls to the “new” and “delete” keywords for textures since only it knows if the Texture is in use.

* Is a Singleton.
* Should keep a list of previously loaded Textures and some way to tell which is which… an STL map would be ideal for this purpose, but you can use any method you like.
* TextureManager owns all the textures in the game and no other system should delete them.

## **Public Functions:**

**Constructor:** Create some kind of list/array/storage to store all the textures

**Destructor:** Delete all the textures

**LoadTexture:** Takes in the path and filename of the texture to load.

Should check if that texture already exists in a list, if not, load it and add it to the list. If it does already exist then just return the pointer to that texture that was loaded previously so that the same texture can never be loaded more than once.

* Param: filename of the texture to be loaded
* **Note**: An STL map would be ideal for storing these textures. The map stores two values joined together into a “pair” and one can be used to get the other. In this case the two values could be the filename of the Texture and the pointer to the Texture. That way we can ask the map to search for a texture based on the filename.

# Class: GUI

A singleton class used to draw the in-game UI. The appearance of the UI are up to you but should include graphics, not just text.

* Is a Singleton to make access easier.
* Should display the following things:
  + Player’s health
  + Player’s score
  + Player’s lives

## **Public Functions:**

**Constructor:** Needs to initialise the object.

**Destructor:** Clean up.

**Draw:** Draw the UI.

**SetHealth:** Will be called to tell the UI how much health to display.

**AddScore:** Will be called to tell the UI to increase the player’s score. The amount passed in should be added to a value that has previously been stored.

**GetScore:** Used by the Game Over screen to get and display the player’s final score.

**SetLives:** Set how many lives the player has left.