

Codename - Slash

The Architecture Report

Manibharathi Periasamymanjula

Computer Games Technology

180055781

Contents

Contents	1
Project Overview	2
Instruction Manual	3
Part I	4
Load appropriate assets for the game type using a resource management strategy	4
Control of the game character or first person view using keyboard, joystick, mouse or touch control. An event-driven architecture should be used to separate input hardware from the responding code	5
Collision detection or alternative hit detection using basic brute force techniques	6
Moving and animated game elements, demonstrating frame-rate independent game loop control.	7
Part II	8
Configurable game world with positions/attributes of game elements/opponents demonstrating a data-driven approach	8
Collision response based removal of game elements showing separation of collision detection and collision response code	10
Scoring system demonstrating use of event listeners	12
High-score table demonstrating use of serialization or an alternative approach to provide a game state load/save mechanism	14
Part III	16
Start-screen (containing intro and keyboard controls) and game over screen (with score and restart options) demonstrating use of state pattern and FSM with game loop	16
Power-ups demonstrating use of event-listeners and re-use of a base-class for game objects	18
NPC opponents demonstrating FSM control of game objects	19
Overall game-play and presentation, including use of additional level challenges as necessary	21
Part IV	23
An analysis of improvements to the speed of your algorithms that have been, or could be, achieved using your knowledge of hardware architecture	23

Project Overview

I have approached this project with the mindset of creating a prototype, that contains good architecture. Therefore the game itself is nowhere near finished, but has lots of functionality that can be expanded upon to realise the final vision. The game itself is a top down shooter battle arena. It is about fighting hordes of different types of enemies with various weapons (mainly guns, but will implement a sword in the future). There are currently 2 playable stages, but this can easily be increased by adding a new XML file called Stage{n}.xml with n stage number, no need for code change.

Design Patterns used:

- Game Loop
- Update
- Object Pool (GameObjects)
- Observer (Event System)
- Singleton (Manager classes)
- Subclass Sandbox (Weapon System)
- Flyweight (MapGen Tile Texture).

NOTE: I have attempted to complete all objectives of the coursework, ones which were not also there in my presentation, including separate thread for loading of stages.

NOTE: On beating both the levels, the game goes to GameOverState, which is the same as losing, this is not a bug.

Instruction Manual

The Menu system is self-explanatory. However, the **save-system** works by autosaving at the end of a stage, and if the game is closed and reopened, there will be a continue button to start from the same stage.

The **aim** of the game is to destroy all the monsters without dying. The table below shows key bindings for gameplay.

Key	Binding
WASD	Movement of the character
Mouse move	Move the cursor to aim
Mouse left click	Shoot the equipped weapon
Mouse scroll (up or down)	Swap between the weapons held

Part I

Load appropriate assets for the game type using a resource management strategy

There are 3 parts to my method of resource management for my game engine.

ContentManager for each GameState (Scene)

Firstly, since my scene management runs through a state pattern, which houses a state for each scene, e.g. GameplayState, MenuState etc. The obvious strategy for all basics assets needed for each state would be loaded through their **separate ContentManager**.

So, on the **Enter()** method of each state, it calls the **LoadContent()** method that loads the assets needed for the state. Then on the **Exit()** method it will call the **UnloadContent()** method that unloads the entire contentManager. This was mainly useful to load the basic, group of assets needed to build the such as a few UI elements and hero's assets.

Some Separate ContentManagers

The **EnemyDirector** and the **MapGen** classes both have their own ContentManager. This is because during the Gameplay state there will be times when either's content need to be **flushed**. For example, at the end of each stage the EnemyDirector will unload its content and only load in the necessary content (maybe 2 types of enemies' textures) for the next stage. This is useful so that the memory does **not** have to have all the types of enemy textures for each animation during **every stage**.

The MapGen's ContentManager is not used for unloading between stages on the current prototype since there is only one map, however due to the engine's flexibility when **later maps** are added for each stage, this feature would be **very easy** to add in.

Loading Objects/Pools and appropriate assets between stages

Between each stage, I have a system that will destroy the current enemy object pools and unload their assets, then load only the needed pools and assets for the following stage. E.g. Stage 4 - Max 5 Doge, and max 8 Skull enemies on screen at once, but Stage 5 - Max 10 Doge, max 3 Skull and max 7 Dark enemies on screen at once. Here, the **advantage** of loading in only those assets and having new object pools of the appropriate size, I found greatly benefited the smoothness of the gameplay **during a stage**.

However, the disadvantage to this is the unloading and loading of assets in between stages. This came at a **cost of a hiccup** in between each stage. I tackled this issue using a design goal, which was to give the player a brief moment or break before the next stage commenced.

During this break, the player would be able to move around, but the loading would be done in the background. This could be done using a separate **background Thread**. I invoked the **OnNextStage()** method through this background thread, while allowing the normal gameplay **Update()** to execute on the main thread. This allowed for smooth loading of assets and object pools between each stage.

```

    stageLoaderThread = new Thread(BackgroundLoadStage);
    stageLoaderThread.Start();
}

// Unload previous assets and objects and
// Load assets and objects needed for next stage
// note: should be called on separate thread
private void BackgroundLoadStage()
{
    waitingToBegin = true;

    // Save Game
    gameManager.SaveGame();

    // Thread.Sleep(2000);
    poolManager.ClearPoolsForNextStage();

    // Load all relevant assets
    enemyDirector.OnNewStage(gameManager.CurrentStageData);
    // Create relevant object pools
    poolManager.CreateStageSpecificPools(gameManager.CurrentStageData);

    waitingToBegin = false;
}

```

I also had a bool, to render some text to say “Creating Stage {0}” just for a visual cue, which is currently displayed for split second only due to small stages, but as the game is expanded with large hordes of many types of enemies, it will be longer.

Control of the game character or first person view using keyboard, joystick, mouse or touch control. An event-driven architecture should be used to separate input hardware from the responding code

The input handling system consists of CommandManager and InputListener classes which **provide an interface** to bind a Keyboard, Mouse or Scroll input to a method. Then when the CommandManager **Update()** is called, each assigned inputs are checked for input and an appropriate **event** is fired to invoke the attached method.

Integrating this system into the state system meant, for each state, I have a `CommandManager` object and an `InitialiseKeyBindings()`. On `Enter()` of each state, the `InitialiseKeyBindings()` will be called in order to attach the appropriate key inputs to each given method.

```
protected override void InitialiseKeyBindings()
{
    if(commandManager != null)
    {
        commandManager.AddKeyboardBinding(Keys.W, gameManager.Hero.MoveUp);
        commandManager.AddKeyboardBinding(Keys.D, gameManager.Hero.MoveRight);
        commandManager.AddKeyboardBinding(Keys.A, gameManager.Hero.MoveLeft);
        commandManager.AddKeyboardBinding(Keys.S, gameManager.Hero.MoveDown);
        commandManager.AddKeyboardBinding(Keys.Space, gameManager.Hero.Dash);

        commandManager.AddMouseButtonBinding(MouseButton.LEFT, gameManager.Hero.ShootWeapon);
        commandManager.AddMouseButtonBinding(MouseButton.RIGHT, gameManager.Hero.ShootWeapon);
        commandManager.AddScrollBinding(Scroll.DOWN, gameManager.Hero.PreviousWeapon);
        commandManager.AddScrollBinding(Scroll.UP, gameManager.Hero.NextWeapon);
    }
}
```

Through this decoupling of input and response using events, the associated methods can do the response wherever it needs to using the given button state. The example below shows one hero command of moving right, only when the given button is pressed.

```
#region Hero Commands

public void MoveRight(eButtonState buttonState, Vector2 amount)
{
    if (buttonState == eButtonState.PRESSED)
    {
        movement.X = 1.0f;
    }
}

public void MoveLeft(eButtonState buttonState, Vector2 amount)
{
}
```

Collision detection or alternative hit detection using basic brute force techniques

I will expand on the entire Collision Management system in the Part 2 of the report speaking about `ColliderTypes` and response, however for each of the **GameObjects** on the screen it will implement an **ICollidable Interface**, which allows it to be part of the collision system. Each `ICollidable` will have a **boundingRect** that will act as its collider.

```
public bool CollisionTest(ICollidable other)
{
    if (other != null)
    {
        return BoundingRect.Intersects(other.BoundingRect);
    }
    return false;
}
```

Since each `ICollidable` will have a `boundingRect`, this will be used to check if it intersects with the other `ICollidable` it is testing against.

Furthermore, since the Collision tests are **separate**, if in the future there is a need for circular colliders or even 3D collision detection, it can be **easily implemented** using a circle/sphere property for an ICollidable and the collision test can be added accordingly.

For the **player boundaries** on the static map I have taken a simpler approach than for **dynamic objects**. Although, there is a staticEnvironment ColliderType present, which can be implemented for different types of maps, with different static collider regions, for this prototype I am simply clamping the hero's movement across each border if the Mapcollider enum is set to BattleArena.

```
// Boundary colliders
if(MapGen.Instance.CurrentMapColliderType == MapCollider.BattleArena)
{
    position.X = MathHelper.Clamp(position.X, (BoundingRect.Width / 2) + 32, (Game1.SCREENWIDTH - BoundingRect.Width / 2) - 32);
    position.Y = MathHelper.Clamp(position.Y, (BoundingRect.Height / 2) + 64, (Game1.SCREENHEIGHT - BoundingRect.Height / 2) - 64);
}
```

Moving and animated game elements, demonstrating frame-rate independent game loop control.

Firstly, in order to test for frame-rate independence, you will be able to change the IsFixedTimeStep bool to false, on **line 33** in **Game1**. The thing that will need to be frame-rate independent is any sort of time based events, i.e. animations and object movements. All are handled with reference to **deltatime** or the time it takes for each frame to pass. This deltatime I obtained through Gametime.elapsedtime.seconds.

Animation of sprites

For the animation system, there is a class **Animation** that takes a sprite sheet texture and splits it into a list of separate frames. The **Animator** class, when told to draw each frame for a given amount of frame time, it will have a **time** variable that gets added by deltaTime on each **update()**. So the correct amount of time is passed before the next frame is drawn.

Movement and timers

Firstly, any type of movement of GameObjects is done per frame with a multiplication of deltaTime and a given speed. The example below shows an **enemy's chase state**, where the enemy moves towards the position of the player by moving its position in the given **direction**, at a given **speed**, relative to **deltaTime**.


```
public override void Execute(Enemy owner, float deltaTime)
{
    // Get direction to hero
    Vector2 dir = EnemyDirector.Instance.DirectionToHeroNormalised(owner.Position);

    // Move position in that direction by movespeed
    owner.Position += owner.MoveSpeed * dir * deltaTime;
}
```

For some other time based events, I created a **Timer class** that is updated with deltaTime. This again means it will be frame-rate independent.

Part II

Configurable game world with positions/attributes of game elements/opponents demonstrating a data-driven approach

There are 2 parts to where I have used this data driven approach to configuring the world.

Map Generation

The **MapGen** class can be used to load in tiles as maps and paint them on the screen.

- 1) **GetMapData()** loads in a **CSV** file of the map data as a **2D string array**, by first importing as a jagged array and then converting to a 2D. Each **value** of this array will be the **name** or key of each tile in that position. It adds this array to a **Dictionary of Maps** by name.
- 2) **LoadMapTextures()** can then be called to load each unique texture needed for each tile on the map. Based on the name/key it loads in the texture with the same name and stores it on a string to texture2d Dictionary. This is **separate** to loading the map values as it will take significantly longer to load textures than an array of strings, and can be called based on engine's resource loading structure.
- 3) **DrawMap()** is called to draw the currentMapToDraw, from position (0,0) evenly with the size of the tile given before.

```

public void DrawMap(SpriteBatch spriteBatch)
{
    // Position of tile's rect with tile's size
    Rectangle tileRect = new Rectangle(0, 32, tileSize, tileSize);
    // Origin to draw sprite, always 0,0
    Vector2 origin = new Vector2(0, 0);
    // Get the current map to draw
    Map map = mapDictionary[currentMapToDraw];

    for (int y = 0; y < mapGridSizeY; y++)
    {
        for (int x = 0; x < mapGridSizeX; x++)
        {
            // Get appropriate tile's texture
            Texture2D tex = map.textureDictionary[map.TileMapValues[y, x]];

            // Draw texture in appropriate position
            spriteBatch.Draw(tex, tileRect, null, Color.White, 0, origin, SpriteEffects.None, 0);
            tileRect.X += tileSize;

            // Reset X point
            tileRect.X = 0;
            //
            tileRect.Y += tileSize;
        }
    }
}

```

Stage Data (Enemies, Spawn-points, pickups)

Each stage of the game session has a lot of **different stats** that must be designed according to the designer of the game. Applying a **data driven** approach to this allows **game designers** to carefully design and test each level by simply editing XML data files of each Stage.

```

// Stage Data object that can
public class StageData
{
    public int stageNumber;

    //
    public int enemiesToFight;
    public int intervalBetweenSpawn;
    public int spawnPointCount;

    // Number of enemies of each type that can be present at 1 time
    public int maxDogeCount;
    public int maxBaldCount;
    public int maxSkullCount;
    public int maxDarkCount;

    // Probability of spawning each type of enemy, must add up to 1
    public float probDogeSpwan;
    public float probBaldSpwan;
    public float probSkullSpwan;
    public float probDarkSpwan;

    // Probability of pickup drop, on enemy death
    public float probHealthDrop;
    public float probAmmoDrop;
}

```

The below is the view of the **XML file** the designers can edit. By deserializing this into the StageData the level can be created through the **EnemyDirector** using that StageData object.

```

1  <?xml version="1.0" encoding="utf-8" ?>
2  <StageData>
3      <stageNumber>1</stageNumber>
4      <enemiesToFight>5</enemiesToFight>
5      <intervalBetweenSpawn>3</intervalBetweenSpawn>
6      <spawnPointCount>4</spawnPointCount>
7      <maxDogeCount>5</maxDogeCount>
8      <maxBaldCount>0</maxBaldCount>
9      <maxSkullCount>5</maxSkullCount>
10     <maxDarkCount>0</maxDarkCount>
11     <probDogeSpwan>0.5</probDogeSpwan>
12     <probBaldSpwan>0</probBaldSpwan>
13     <probSkullSpwan>0.5</probSkullSpwan>
14     <probDarkSpwan>0</probDarkSpwan>
15     <probHealthDrop>0</probHealthDrop>
16     <probAmmoDrop>0</probAmmoDrop>
17 </StageData>

```

Collision response based removal of game elements showing separation of collision detection and collision response code

The **CollisionManager** singleton oversees the all collisions as part of the engine. I have used a **Singleton Pattern** here and in a few other manager classes since the game engine will only have 1 ever object of this type present in a given session.

As mentioned previously, objects that implement **ICollidable** are all objects that will be checked for collisions. The CollisionManager has every ICollidable in a list and **loops through each other**, lets each ICollidable do its own collision test against the other.

The beauty of this system is in the fact that each ICollidable has a **ColliderType** enum value and a list of **ColliderTypes** that it is allowed to interact with. For example, below shows that the **Bullet** gameobject can interact, enemy, and staticEnvironment. This means we **do not** check for collisions between bullets, which is huge performance boost.

```

public List<ColliderType> interactionTypes;
public List<ColliderType> InteractionTypes { get { if (interactionTypes == null)
    { List<ColliderType> i = new List<ColliderType>();
      i.Add(ColliderType.enemy);
      i.Add(ColliderType.staticEnvironment);
      return i;
    } return interactionTypes; } }

```

During the loop to check collisions, the above rule is checked simply by asking if the other `ICollidable`'s type is contained within this one's `InteractableTypes` list. If the collision test returns true, then each of collision pair is added into a hashset of of **Collision** objects.

```
// Make sure we're not checking an object with itself
// Plus if it is one of the collider types it can interact with
if (!collidable1.Equals(collidable2) && collidable1.InteractableTypes.Contains(collidable2.ColliderType))
{
    // If the two objects are colliding then add them to the set
    if (collidable1.CollisionTest(collidable2))
    {
        collisionOccuranceList.Add(new Collision(collidable1, collidable2));
    }
}
```

After getting a hashset full of collisions that have occurred this frame, the resolution is done by called each collisions' **Resolve()** that will call the first `ICollidable`'s **OnCollision()** method that will apply the unique resolution to that collision.

Below shows an example of the **Hero** object resolving its collision. The hero can pretty much collide every `ColliderType` except **HeroAttacks**, therefore there is some logic to its resolution.

- 1) If not a **TriggerRegion**, then the hero's position should be pushed in the opposite direction to the collision occurrence, by using an intersected rectangle to calculate the distances.
- 2) If an **enemy** and the hero is not currently invulnerable, by casting the enemy as the **IDamageDealer** interface which will contain the `DealDamageValue`, the player can take said amount of damage.

```
public void OnCollision(ICollidable other)
{
    if (other.ColliderType != ColliderType.triggerRegions)
    {
        // Get rectangle of the intersection/collision depth
        Rectangle r = Rectangle.Intersect(BoundingRect, other.BoundingRect);

        // Move the collider in the opposite direction by that amount
        position += new Vector2(r.Width, r.Height);
    }
    if (other.ColliderType == ColliderType.enemy && !invulnerabilityTimer.Running)
    {
        TakeDamage((other as IDamageDealer).DealDamageValue);
    }
}
```

Scoring system demonstrating use of event listeners

For my game of battle, the scoring system is very simple, the player gains score for every enemy kill, with each enemy type contributing a different value.

The parent **Enemy Class** has a virtual method called **TakeDamage()**, that has 2 **Action<Vector2>** events, **OnDamage** and **OnDeath**, that send the current position of the enemy.

```
public virtual void TakeDamage(int damagePoints)
{
    CurrentHealth -= damagePoints;
    if (CurrentHealth > 0)
    {
        OnDamage?.Invoke(Position);
    }
    else
    {
        IsActive = false;
        OnDeath?.Invoke(this, Position);
    }
}
```

The **PoolManager** that is in charge of objects and objectPool management, **subscribes** each instance of enemy's events to the relevant **SpawnEnemyHitEffect()** and **OnEnemyDeath()** methods shown below. Therefore, when any enemy takes damage or is dead, the subscribed methods will be **invoked** if they exist (the **"?.Invoke"** notation is used to do this above).

```
//
private void SpawnDoge(IArgs args)
{
    Doge doge = dogePool.SpawnFromPool(args);
    enemiesAlive.Add(doge);
    doge.OnDamage += SpawnEnemyHitEffect;
    doge.OnDeath += OnEnemyDeath;
    OnAddCollider?.Invoke(doge);
}
```

Although, there is not **DeathEffect** object to play an animation yet, the system allows for the addition of this in the **PoolManager's** method. The method invokes another event that sends the enemy object that is destroyed, which is listened to by the **EnemyDirector**.

```
// Calls ondeath action, with the enemy object and calls the death effect method at given position
private void OnEnemyDeath(Enemy enemy, Vector2 position)
{
    OnDeath?.Invoke(enemy);
    SpawnDeathEffect(position);
}
```

The below is the method that is subscribed to the **PoolManager's OnDeath** event, which increases the **killCount** and decreases the number of that type of enemy present on the screen. This logic is for handling **Stages** and spawning enemies. There is again one more event that is invoked, which sends only the score that should be added.


```
// On Enemy death, increases score and removes from appropriate counts
private void OnEnemyDeath(Enemy enemy)
{
    // Increment score
    ScoreIncrement?.Invoke(enemy.KillScore);

    // Increase killCount
    killCount++;

    //
    // Decrease enemy type count present
    if (enemy is Doge)
    {
        if (currentDogeCount > 0)
        {
            currentDogeCount--;
        }
    }
}
```

This is listened to by the GameManager, which updates the CurrentScore value.

```
private void IncrementsScore(int increment)
{
    CurrentScore += increment;
}
```

Usually, it is bad practice to have a chain of events and ideal to have many methods subscribed to a single OnDeath Event. However, with this case it is not possible as:

- 1) Only the **PoolManager** knows about each individual enemy object that is alive and needs to listen to.
- 2) The **EnemyDirector** controls the Enemy spawn logic, so needs the object but the **GameManager** only needs an int of the score, which means it is unnecessary to subscribe to the same event EnemyDirector subscribed to.

High-score table demonstrating use of serialization or an alternative approach to provide a game state load/save mechanism

I have used serialization for both a Highscores table and Save/Load mechanism. Both XML files that are serialized to are kept in the `\Codename - Slash\Codename - Slash\bin\Windows\x86\Debug` folder. Finally, for both systems, I am using the `ToXmlFile()` method to convert a given object to an xml file of a given name.

```
/// <summary>
/// Serializes an object to an XML file.
/// </summary>
public static void ToXmlFile(Object obj, string filePath)
{
    var xs = new XmlSerializer(obj.GetType());
    var ns = new XmlSerializerNamespaces();
    var ws = new XmlWriterSettings { Indent = true, NewLineOnAttributes = NewLineOnAttributes, OmitXmlDeclaration = true };
    ns.Add("", "");

    using (XmlWriter writer = XmlWriter.Create(filePath, ws))
    {
        xs.Serialize(writer, obj);
    }
}
```

Highscores

I have referred to highscores as **Awards** through the entirety of my code. I have a **AwardsData** object that stores a list of ints which represent the scores, which will be serialized.

```
public class AwardsData
{
    public List<int> scores;

    public AwardsData()
    {
        scores = new List<int>();
    }

    public AwardsData(List<int> scores)
    {
        this.scores = scores;
    }
}
```

There are 2 helper methods in **GameManager**, that deal with Awards. One of which, is given a new score to add to the already existing list and sort by **descending order using LINQ**, and then write those values to the AwardsFile, shown below. This method is called on entering the **GameOverState**.

```
// Adds new score and updates the AwardsFile
public void UpdateAwardsFileWithNewScore(int newScore)
{
    // Add new score and sort descending
    AwardsData.scores.Add(newScore);
    AwardsData.scores = AwardsData.scores.OrderByDescending(c => c).ToList();
    // Write to xml
    Loader.ToXmlFile(AwardsData, "AwardsFile.xml");
}
```

The second method, does the reverse by getting all the values from the XML file and keeps it in the AwardsData property within the GameManager, for later use. This is called when the game begins in the **MainMenuState**, to be displayed when the player goes to the **AwardsState**.

```
// Loads awards file into awardsData variable
public void LoadAwardsFile()
{
    if (File.Exists("AwardsFile.xml"))
    {
        // Grab values and sort descending
        AwardsData a = new AwardsData();
        Loader.ReadXML("AwardsFile.xml", ref a);
        a.scores = a.scores.OrderByDescending(c => c).ToList();
        AwardsData = a;
        return;
    }
    AwardsData = new AwardsData();
}
```

Save/Load

The Save/Load mechanism works in a very similar fashion except it is used at different times in the game. At the **end of each stage**, the system will **auto save** the progress, with (StageNumber, Score, and ammo remaining of each weapon). The **SaveGame()** method in **GameManager** handles this.

```
try {
    // Load save data from xml file
    SaveData s = new SaveData();
    Loader.ReadXML("SaveFile.xml", ref s);
    CurrentSaveData = s;

    // Create new hero instance
    Hero = new Hero();

    // Set up session with save data values
    CurrentScore = CurrentSaveData.currentScore;
    ChangeStage(CurrentSaveData.stageNumber - 1); // Also load stage data
    for (int i = 0; i < Hero.WeaponHandler.WeaponsList.Count; i++)
    {
        Hero.WeaponHandler.WeaponsList[i].CurrentAmmoCarry = CurrentSaveData.weaponDataList[i].currentAmmoCarry;
        Hero.WeaponHandler.WeaponsList[i].CurrentMagHold = CurrentSaveData.weaponDataList[i].currentMagHold;
    }
}
catch (FileNotFoundException e)
```

This means if the game is closed by accident during a game session (can be tested by alt+f4 during latter stage), on reopening game there will be an extra **Continue** button. This is done, by simply checking the existence of the **SaveFile** as the game starts up (done on the **MainMenuState**).

Both the **OnNewGame()** and **OnContinueGame()** are called before the transition to **GameplayState** from **MainMenuState**, the first deletes the old save and creates a new **SaveData** object, however the second loads the xml file into a **SaveData** object to be used.

Part III

Start-screen (containing intro and keyboard controls) and game over screen (with score and restart options) demonstrating use of state pattern and FSM with game loop

For the **GameStates**, the scene management, I have used a traditional state pattern in the style of the Gang of Four. The pattern works using a **dynamic dispatch** to update the current state and return value to transition between states.

- 1) There is an abstract **GameState** that contains common abstract and virtual methods, **InitialiseState()**, **Enter()**, **InitialiseKeyBindings()**, **LoadContent()**, **UnloadContent()**, **Update()** and

Draw(). There is also a **CommandManager**, reference to an **IServiceProvider** and a **ContentManager**, that can be used for each state.

```
protected CommandManager commandManager;
protected IServiceProvider services;
protected ContentManager stateContentManager; // Content manager for each state

public virtual void InitialiseState(Game1 game)
{
    // Create new content manager for each state
    stateContentManager = new ContentManager(game.Services, "Content");
}

// Enter method to be called on entry into particular state
public virtual void Enter(Game1 game)
{
    // Load each state's content
    LoadContent();
    // Create new commandManager
    commandManager = new CommandManager();
    // Initialise the keybindings
    InitialiseKeyBindings();
}
```

- 2) There is then **concrete classes** that **inherit** from **GameState**, for each state in the game, i.e. **GameplayState** or **ProtocolState**. These methods override the **GameState**'s methods to implement their own behaviour.

```
public class ProtocolState : GameState
{
    private SpriteFont hudFont;

    private MenuUI menuUI; // State's menuUI
    private GameManager gameManager; // Reference to Gamemanager Singleton
```

- 3) Each state is created and stored in the **GameState** class as a static property since there will only be one of each state needed, during the game.

```
public static MainMenuState MenuState { get; } = new MainMenuState();
public static AwardsState AwardsState { get; } = new AwardsState();
public static ProtocolState ProtocolState { get; } = new ProtocolState();
public static GameplayState GameplayState { get; } = new GameplayState();
public static GameOverState GameOverState { get; } = new GameOverState();
// public static NextStageState NextStageState { get; } = new NextStageState();
```

- 4) The states are initialised on the **Game1 Initialize()** method, then the first state is set as the current state and its **Enter()** method is called.
- 5) The current state is then Updated by using a **dynamic dispatch**, that will return **null** if there is not transition or return the state it will transition to. Based on that the appropriate **Exit()** and **Enter()** methods are called.

```
// Scene/Game State handling area
GameState s = state.Update(this, (float) gameTime.ElapsedGameTime.TotalSeconds);
if (s != null)
{
    state.Exit(this); // Call previous state's exit method
    state = s;
    state.Enter(this); // Call new state's enter method
}
```

6) Finally, the **Draw()** method is called for the current state from the **Game1**'s draw method.

Power-ups demonstrating use of event-listeners and re-use of a base-class for game objects

I have not implemented Powerups yet for the prototype, however I have used event listeners in various parts of the code in order to **decouple** different areas. I have also used **inheritance** in various parts of code including: Enemies, Furthermore, I have used a **GameObject** class as a base class for all **dynamic** objects to inherit from that have common functionality.

A few examples of Event listener usage:

- 1) **PoolManager**, that deals with creation and handling of objects and object pools, has a few events that are listened to by the CollisionManager to add, Remove or Remove colliders of type.
- 2) **Weapon handling**, subscribing and unsubscribing event listeners for shoot and reload based on which weapon object is currently equipped.
- 3) **GameplayUI**'s weapon swap and health value is updated through events it listens for.

When it comes to the implementation of the GameObject base class, it is an abstract class that cannot be instantiated. It implements IPoolable so that all gameobjects can be pooled and will have the correct implementation for being able to be pooled.

```
// GameObject base class for all dynamic in game objects
public abstract class GameObject : IPoolable
{
    protected float liveTime;

    public bool IsActive { get; protected set; }

    public abstract void OnPoolInstantiation();

    public abstract void OnSpawnFromPool(IArgs args);

    public abstract void Update(float deltaTime);

    public abstract void Draw(float deltaTime, SpriteBatch spriteBatch);
}
```

Bullet and Enemy classes inherit from this currently. Furthermore, there is an `IsActive` bool, to specify whether the PoolManager should dispose of it or not. It works sort of like the Garbage Collector when there is no reference to a piece of dynamic memory, it removes it from being alive.

NPC opponents demonstrating FSM control of game objects

The Enemy logic works using an Finite State Machine, defined in the **NPCStateMachine** class. It has a list of **NPCStates** which all contain an **Enter()** **Exit()** and **Execute()** method. All unique States can derive from this abstract parent state and override each of the methods with its own functionality, like the **ChargeState** shown below.

```
public class ChargeState : NPCState
{
    float currentTimer;

    private Vector2 currentAttackPos;
```

This is similar to the state pattern I used for the scene management earlier, however the difference is with the state changing. **Transitions** objects are used to store a **func<bool>** to have a method that checks the condition to transition and a **NextState** to store the state object it should transition to.

```
// Transition class that stores the state to go to, with a Func condition to check
public class Transition
{
    public readonly NPCState NextState;
    public readonly Func<bool> Condition;

    public Transition(NPCState nextState, Func<bool> condition)
    {
        NextState = nextState;
        Condition = condition;
    }
}
```

For the game engine, I will be defining states that are **not specific** to a particular type of enemy, so that each of these states can be **reused** with different data by many types of enemies. For example, the **ChaseState** and the **IdleState** are used by **Skull** and **Doge**. However, the **moveSpeed** may be different for each owner, thus separating the **behaviour** from the **data**.

When it comes to using the FSM with each of the enemies, an enemy object will have its own **NPCStateMachine**, an instance of each different type of **State** that will be used, and each transition between the states. Note here, the difficulty of defining each transition between states for a larger set of states would become tedious or confusing, so state-transition visual editor would be useful.

```
stateMachine = new NPCStateMachine(this);

// Create the states
IdleState idle = new IdleState("idle");
ChaseState chase = new ChaseState("chase");
ShortRangeAttackState shortRangeAttack = new ShortRangeAttackState("shortRangeAttack", 2.0f, 1.5f, true);

// Create the transitions between the states
idle.AddTransition(new Transition(chase, () => (Math.Pow(distanceToBeginChase, 2) >= EnemyDirector.Instance
chase.AddTransition(new Transition(shortRangeAttack, () => (Math.Pow(distanceToBeginAttack, 2) >= EnemyDir
chase.AddTransition(new Transition(idle, () => (Math.Pow(distanceToBeginChase, 2) < EnemyDirector.Instance
shortRangeAttack.AddTransition(new Transition(chase, () => (Math.Pow(distanceToBeginAttack, 2) < EnemyDire

// Add the created states to the FSM
stateMachine.AddState(idle);
stateMachine.AddState(chase);
stateMachine.AddState(shortRangeAttack);
```

In the game there are 2 NPCs defined with a state machine called **Doge** and **Skull**, however there will also be **Dark** and **Bald** npcs for the complete game. The above example, the FSM definition for **Doge** gameobject. The behaviour is simple, after spawning, the npc will chase the hero when in given radius, on reaching a closer radius it will initiate a short ranged attack.

The short range attack itself is done through **interpolating** the gameobject's position between 2 places using **bell curve** function. This gives a smooth back and forth motion for each attack.

```

if(currentAttackCounter <= 1.0f)
{
    // Increase value by speed
    currentAttackCounter += attackSpeed * deltaTime;
    // Get lerp value based on function, so that the value peaks at 1 mid way and returns to 0 [to create a forward back effect]
    float positionLerpValue = (float) (- Math.Pow(currentAttackCounter, 2) + currentAttackCounter) * 4;
    // Assign appropriate position
    owner.Position = Vector2.Lerp(initialPosition, currentAttackPos, positionLerpValue);
} else
{

```

Overall game-play and presentation, including use of additional level challenges as necessary

Weapon Handling

The technical aspect of the weapon handling system and making it expandable for many different types of weapons took a good portion of time and gives the game fun gameplay.

- 1) **WeaponHandler** class handles all logic for **swapping** between weapons, **rotating** and **positioning** given weapon, **drawing** the appropriate texture, and acting as an **intermediary object** to invoke **shoot()** or **reload()** methods to a particular weapon.
- 2) There is a **Weapon** parent class. Here I have tried to use a **Subclass Sandbox** pattern, that is done through pushing most of the derived classes' functionality to the parent class. This is great for implementing many different by similar functionality, i.e. weapons to be used.

Character Invulnerability

After taking a hit from an enemy, the hero has a short invulnerability period in which he will not take damage to hits. This is done using a timer and a bool updated after each hit.

IDamageable and IDamageDealer Interfaces

In order to provide correct encapsulation and invoke functionality through a particular use case such as dealing damage. I have created 2 interfaces which can be implemented by any class.

```

public interface IDamageDealer
{
    int DealDamageValue { get; set; }
}

```

Firstly, the **IDamageDealer** gives a property that is the damage value. In this way, when doing **collision response** from an **ICollidable**, I will be able to access the **DealDamageValue** without having to access the actual object, but through an **interface**.


```
public void OnCollision(ICollidable other)
{
    // Take damage if collision with hero attack
    if (other.ColliderType == ColliderType.heroAttack)
    {
        TakeDamage((other as IDamageDealer).DealDamageValue);
    }
}
```

Further, I have set up the IDamageable, for any object that can take damage, with 2 **overloaded** methods. If the damage that it takes is in a particular direction, then the **special effect** can be drawn on the screen towards a certain way, if not it will simply take damage.

```
public interface IDamageable
{
    void TakeDamage(int damagePoints);
    void TakeDamage(int damagePoints, Vector2 direction);
}
```

Enemy Spawn system

The EnemyDirector houses a spawn system that is done through a series of probabilities that are passed from the XML document StageData.

- 1) Spawns the enemies based on the interval time and the the probability of the next enemy to spawn.
- 2) Creates and places portals of enemy spawn points in a random location around the map. These spawn points are then chosen randomly for the next point of enemy spawn.

Inhouse UI System

I built my own UI system that consists of **UIElement** and **Button** classes. **MenuUI** class uses both of those classes to build a complete system for interactivity that can be used on any GameState in the game.

```
// MenuUI class, derived from UI
// used for any menu UI system
public class MenuUI : UI
{
    public List<UIElement> UIElements { get; set; } // List of UIElements on the Menu
    public List<Button> Buttons { get; set; } // List of Buttons on the Menu
    private Button buttonOnHover; // Current button mouse is hovering over

    // Creates new lists and calls UI constructor with given contentManager
    public MenuUI(ContentManager content) : base(content)
    {
        UIElements = new List<UIElement>();
        Buttons = new List<Button>();
    }
}
```

The **interactivity** of the buttons are done by passing a **method** to be executed on select and a GameState to transition to if any. The SelectMethod is null by **default**, but can be given by the creator.

```
// Constructor to initialise and assign values, with SelectMethod action set to null by default
public Button(Texture2D texture, Rectangle destRect, GameState stateToReturn, Action SelectMethod = null)
{
    this.texture = texture;
    this.destRect = destRect;
    onHover = false;
    onPressDown = false;
    onSelect = false;
    this.stateToReturn = stateToReturn;
    this.SelectMethod = SelectMethod;
}
```


Part IV

An analysis of improvements to the speed of your algorithms that have been, or could be, achieved using your knowledge of hardware architecture

There is a lot of scope of improvement for many parts of the code to run more efficiently without worsening gameplay.

Object Pooling

I have already used this technique in order to reduce the time it takes to **create and destroy** objects, and the time it takes to **loop** through objects. If all the objects are created at **one time**, there is very little **fragmentation** between them which makes easier to find their locations when looping through a list of them. Further, since the game will eventually have **large groups** of enemies being destroyed and created very quickly on screen, it is much more efficient to reuse the objects. I have create enemy **object pools** based on the **max** number of each enemy type that can be present at one time.



I have not implemented any **particles effects** for damage or death effects. However, this technique will be extremely useful with this **use case**. Explosion particle effects that need to be created and destroyed very quickly will benefit from this even more than enemies due the sheer number of them. And if they are very **light objects** with little data, having a pool and keeping them for reuse will be of no issue. In the same way, it will be useful to push the common data outside the **enemy** objects to **reduce** their size.

Spatial Locality

When looping through lists of more than 1 dimension, it is very useful to loop through the correct order of dimensions to minimize amount of resources needed to be **cached** to access a specific areas. For example, when looping through a 2D array, it is better to loop through row by row since trying to do column by column means the cpu must cache the row to get a particular value along anyway, which is a waste. I have used this when painting my tiles on **MapGen**.

Task Scheduling

Certain tasks that are checked every frame need not be checked that many times. For example, checking the distance from the hero to move each enemy, does not need to be check 60 times a second (given fixed time rate), but can be done every 10 frames. It will show **minimal** if not any difference in gameplay, but allows other tasks to be executed with the **spare time**.

This can be applied for the state transition checks for GameStates and NPCStates. More frequent for NPCStates but not needed every frame. Having this done for every NPC on screen, would greatly improve performance, when needing to have many present at one time. This can also be done for **collision checks** and **responses** so that it is done every other frame.

Multithreading

Finally, with almost every PC having a **multi-core processor**, it is best to use this by splitting up tasks to be executed in different threads **concurrently**. This will ensure that tasks which are independent of each other are done in **parallel** without having to wait for another to finish. However, if two threads are accessing the same resource, this has to be designed properly, so that not more than one thread is accessing a resources at the same time.