Zeeshan Khalish CGA
180010339

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

The resource management strategy used for loading the assets has been Content pipeline from the XNA Framework. All the graphics have been saved in the content file using the monogame pipeline tool and all the sprites are in there own folder such as the enemy folder for the enemy sprites and player folder for the player sprite, and the path of these assets are saved in the necessary classes. (player class and enemy class for example.) the assets are then loaded using the path with the content manager and the will be saved until they are no longer needed. The code below is a snippet from the Player class that shows the different animations being loaded into the game, where content is a reference to the content manager.

```
1 reference
public void LoadContent()
{
    // Load animations
    idle = new Animation(Level.Content.Load<Texture2D>("Player/Idle"), 0.1f, true);
    celebrate = new Animation(Level.Content.Load<Texture2D>("Player/Celebrate"), 0.1f, false);
    jump = new Animation(Level.Content.Load<Texture2D>("Player/Jump"), 0.1f, false);
    run = new Animation(Level.Content.Load<Texture2D>("Player/Run"), 0.1f, true);
    die = new Animation(Level.Content.Load<Texture2D>("Player/Die"), 0.1f, false);

    //Load Sounds
    lose = Level.Content.Load<SoundEffect>("Audio/Lose");
    //damage = Level.Content.Load<SoundEffect>("Audio/Dmgtaken");
    healthPotion = Level.Content.Load<SoundEffect>("Audio/healthpotion");

    // Calculate bounds within texture size.
    int width = (int)(idle.FrameWidth * 0.4);
    int left = (idle.FrameWidth - width) / 2;
    int height = (int)(idle.FrameWidth * 0.8);
    int top = idle.FrameHeight - height;

    Bounds = new Rectangle(left, top, width, height);
}
```

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.

An event driven architecture has been used to separate input hardware from the responding code. This means that when an event is executed a listener class will receive it.

The Command_Manager class has a dictionary called m_KeyBindings, the key for the dictionary is Keys and the value is the Action delegate. The keys will be the input and when they are pressed the Action will be triggered. The keyDown method is an event handler method, the sender is the object that triggers the event, and a is an instance of keyboardEvents. The Action

action line references the delegate called Action that is tied to a.Key (the key that is pressed from the m_keyBindings dictionary)The if statement checks if the action is associated with the pressed key.

```
1 reference
public void KeyDown(object sender, Keyboard_Events a)
{
    Action action = m_KeyBindings[a.Keys];

    if (action != null)
    {
        action(Button_States.DOWN, new Vector2(1.0f));
    }
}
```

The input_Listener Class is responsible for receiving keyboard inputs and mouse inputs. A HashSet is made to store the currently pressed key and mouse button, the previous and current states for the keyboard and mouse are retrieved, then each event handler is defined. KeyboardButtons is a hashSet collection that stores the currently pressed keys it is initialized as empty.The method AddKButton(Keys Key) adds a value of Keys to the keyboardButtons the same process is also done for the mouse.

```
1 reference
public void AddKButton(Keys key)
{
    KeyboardButtons.Add(key);
}

1 reference
public void AddClick(MouseButton button)
{
    MouseButtons.Add(button);
}
```

The KeyBoard_events class stores a key and the current and previous keyboard state. Which is used by the command manager. The constructor initialises these values which makes them encapsulated so they can be accessed. The Mouse_Events class works the same.

```
class Mouse_Events : EventArgs
{
    public readonly MouseState CurrState;
    public readonly MouseState PrevState;
    public readonly MouseButton Button;

    1 reference
    public Mouse_Events(MouseButton button, MouseState currState, MouseState prevState)
    {
        CurrState = currState;
        PrevState = prevState;
        Button = button;
    }
}
```

```
3 references
public Keyboard_Events(Keys key, KeyboardState currKS, KeyboardState prevKS)
{
    CurrKS = currKS;
    PrevKS = prevKS;
    Keys = key;
}
```

Zeeshan Khalish CGA
180010339

The Button_States class is an enum class that defines the buttons states. The bindings method in the Game_State class is responsible for binding keys to actions so A for walk Left and etc. Layers() works by if checking if the button state is up if so Layers is called. Left() checks if the button state is DOWN if so the player speed is set to -1 to indicate left movement

```
public void Layers(Button_States button, Vector2 amount)
{
    if (button == Button_States.UP)
    {
        Layers();
    }
}

1 reference
public void Left(Button_States button, Vector2 amount)
{
    if (button == Button_States.DOWN)
    {
        level.Player.Speed = -1;
    }
    else
    {
        level.Player.Speed = 0;
    }
}
```

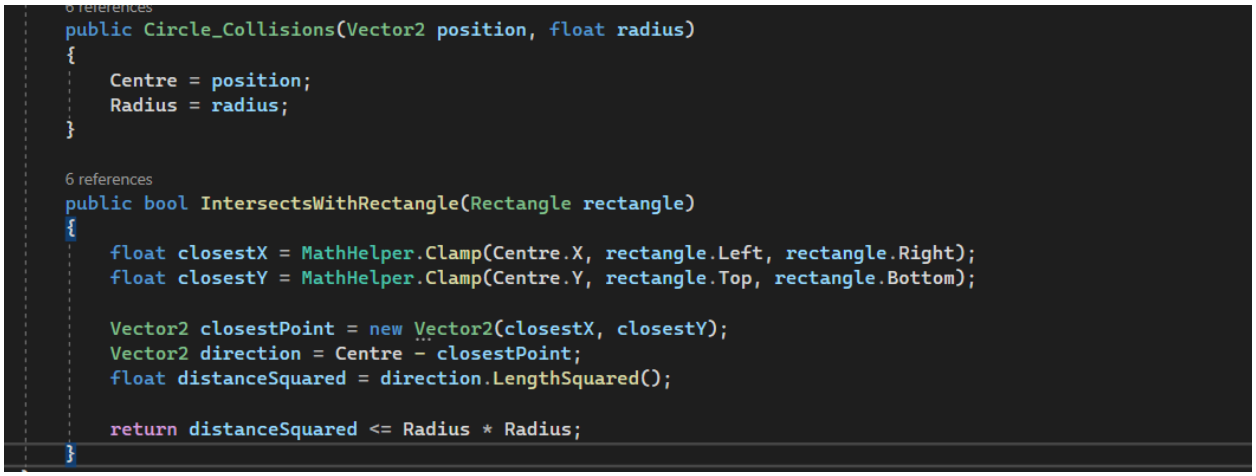
ZCollision detection or alternative hit detection using basic brute force techniques.

The game has a collision detection feature, the Rectangle_Extension class checks the intersection between two rectangles, the centre of each rectangle is found by adding the top left position to the divided width and heights, The non intersecting distance between the two centres is found , by subtracting the x coordinate and y coordinate of the centre, If the absolute value of dist_X is greater than or equal to min_Non_Intersect_Dist_X or the absolute of dist_Y is is greater than or equal than the min_Non_Intersect_Dist_Y return Vector2.Zero since the rectangles are not intersecting.
If dist_X is greater than 0 it shows that rectangle a is to the right of rectangle b, this means the intersection depth is calculated by min_Non_Intersect_Dist_X - dist_X.
If dist_X is less than 0 it shows that the rectangle a is to the left of rectangle b so the intersection depth is calculated by -min_Non_Intersect_Dist_X -dist_X,
Intersection_Depth_X and Intersection_Depth_Y are returned as a Vector2. This class has been mainly used for detecting collision between the player and the floors

```
3 references
public static class RectangleExtensions
{
    1 reference
    public static Vector2 Intersection_Depth(this Rectangle a, Rectangle b)
    {
        float Half_Width_A = a.Width / 2.0f;
        float Half_Height_A = a.Height / 2.0f;

        float Half_Width_B = b.Width / 2.0f;
        float Half_Height_B = b.Height / 2.0f;

        Vector2 Centre_Of_Rectangle_A = new Vector2(a.Left + Half_Width_A, a.Top + Half_Height_A);
        Vector2 Centre_Of_Rectangle_B = new Vector2(b.Left + Half_Width_B, b.Top + Half_Height_B);

        //Calculates current non-intersecting distances between centers.
        float dist_X = Centre_Of_Rectangle_A.X - Centre_Of_Rectangle_B.X;
        float dist_Y = Centre_Of_Rectangle_A.Y - Centre_Of_Rectangle_B.Y;

        //Calculates minimum non-intersecting distances between centers.
        float min_Non_Intersect_Dist_X = Half_Width_A + Half_Width_B;
        float min_Non_Intersect_Dist_Y = Half_Height_A + Half_Height_B;

        //If there is no intersection, program will return (0,0)
        if (Math.Abs(dist_X) >= min_Non_Intersect_Dist_X || Math.Abs(dist_Y) >= min_Non_Intersect_Dist_Y)
            return Vector2.Zero;

        //Calculates intersection depths
        float Intersection_Depth_X = dist_X > 0 ? min_Non_Intersect_Dist_X - dist_X : -min_Non_Intersect_Dist_X - dist_X;
        float Intersection_Depth_Y = dist_Y > 0 ? min_Non_Intersect_Dist_Y - dist_Y : -min_Non_Intersect_Dist_Y - dist_Y;

        return new Vector2(Intersection_Depth_X, Intersection_Depth_Y);
```

Zeeshan Khalish CGA
180010339

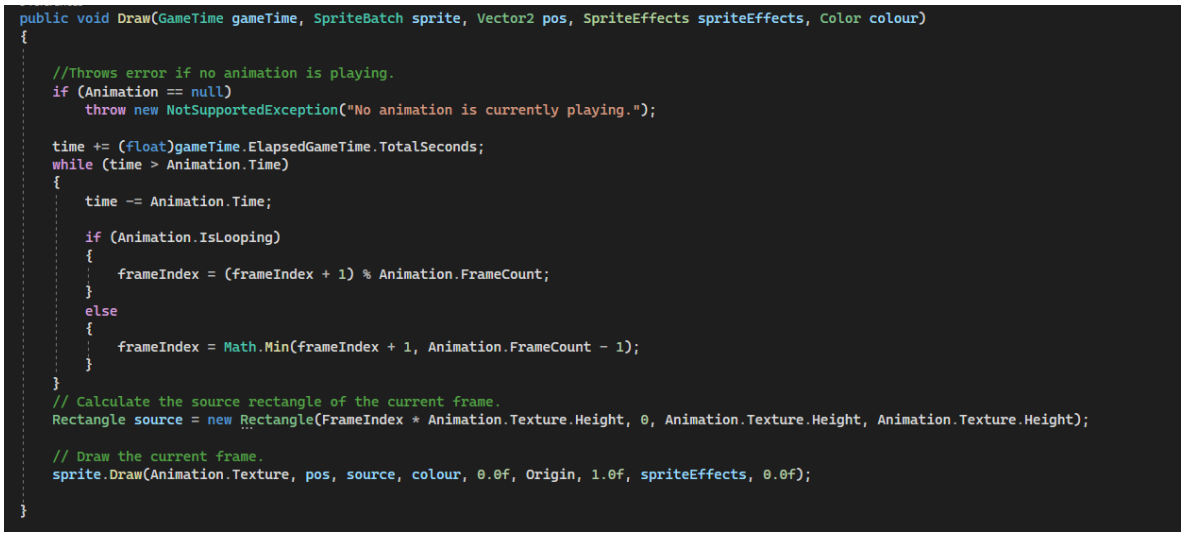
The circle collision class works by a constructor setting the centre to the position and the radius to the radius parameter. The method IntersectsWithRectangle works by checking the rectangles closest point to the circle's centre on the x and y axis. If distanceSquared is less than or equal to the square of the circle's radius (Radius * Radius) it means the circle and rectangle are colliding.This class is mainly used for when the player is colliding with an enemy or collectible



```
public Circle_Collisions(Vector2 position, float radius)
{
    Centre = position;
    Radius = radius;
}

6 references
public bool IntersectsWithRectangle(Rectangle rectangle)
{
    float closestX = MathHelper.Clamp(Centre.X, rectangle.Left, rectangle.Right);
    float closestY = MathHelper.Clamp(Centre.Y, rectangle.Top, rectangle.Bottom);

    Vector2 closestPoint = new Vector2(closestX, closestY);
    Vector2 direction = Centre - closestPoint;
    float distanceSquared = direction.LengthSquared();

    return distanceSquared <= Radius * Radius;
}
```

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

The player and enemies are animated, the animation class gets all the necessary items needed for animations such as the frames of the animation and if its looping. The Player_Animation class works by drawing the current frame animation, time is incremented by the elapsed time from GameTime to keep track of the total time that has passed. A loop is then used while the time is greater than the duration of a single frame. This allows advancing multiple frames in case the animation is shorter than the elapsed time. If the animation is looping the frameIndex is incremented and wrapped around to make sure it stays within the frame range. Afterwards a rectangle is made to define the areas of the texture that corresponds to the current frame. And the Play() method plays the animation.



```
public void Draw(GameTime gameTime, SpriteBatch sprite, Vector2 pos, SpriteEffects spriteEffects, Color colour)
{

    //Throws error if no animation is playing.
    if (Animation == null)
        throw new NotSupportedException("No animation is currently playing.");

    time += (float)gameTime.ElapsedGameTime.TotalSeconds;
    while (time > Animation.Time)
    {
        time -= Animation.Time;

        if (Animation.IsLooping)
        {
            frameIndex = (frameIndex + 1) % Animation.FrameCount;
        }
        else
        {
            frameIndex = Math.Min(frameIndex + 1, Animation.FrameCount - 1);
        }
    }
    // Calculate the source rectangle of the current frame.
    Rectangle source = new Rectangle(FrameIndex * Animation.Texture.Height, 0, Animation.Texture.Height, Animation.Texture.Height);

    // Draw the current frame.
    sprite.Draw(Animation.Texture, pos, source, colour, 0.0f, Origin, 1.0f, spriteEffects, 0.0f);

}
```

Zeeshan Khalish CGA
180010339

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

Data driven design was chosen to create the levels since it allowed for multiple levels to be made simply. A loader class was made to load text and XML files. The method ReadTextFileComplete works by reading the content of the input stream as a single string, a stream reader is used to read from the mFileStream and then StringBuilder is used to construct the complete text content, and any exceptions are caught and an error message is displayed and it is returned as a string. Each item of a level has its own specific letter so for example 'I' represents an invisibility potion, 'D' represents Diamond.

```
0 references
public string ReadTextFileComplete()
{
    StringBuilder result = new StringBuilder();
    try
    {

        using (StreamReader reader = new StreamReader(mFileStream))
        {

            result.Append(reader.ReadToEnd());
        }
    }
    catch (Exception e)
    {

        Console.WriteLine("ERROR: File could not be read!");
        Console.WriteLine("Exception Message: " + e.Message);
    }

    // Return the resulting string
    return result.ToString();
}
```

In the Level class the constructor uses the loader to save the lines of the file in a list, this is achieved by the method LoadTIles() it works by reading each line of the file and adds it to the lines list then it checks to makes sure each line is the same if not an exception is thrown. Tiles are equal to the width and number of lines read. A for loop is used to go through each tile position using a nested for loop it gets the character representing the tile type from the lines list at the current position. The LoadTile method is used to create a tile object . The second to last if statement checks if the player has a starting point

```
1 reference
private void LoadTiles(Stream fileStream)
{
    // Loads the level and checks whether all lines are the same length.
    int width;
    List<string> lines = new List<string>();
    using (StreamReader reader = new StreamReader(fileStream))
    {
        string line = reader.ReadLine();
        width = line.Length;
        while (line != null)
        {
            lines.Add(line);
            if (line.Length != width)
            {
                throw new Exception($"The length of line {lines.Count} is different from all preceding lines.");
            }
            line = reader.ReadLine();
        }
    }

    // Allocate the tile grid.
    tiles = new Tiles[width, lines.Count];

    // Loops through every tile position,
    for (int y = 0; y < Height; ++y)
    {
        for (int x = 0; x < Width; ++x)
        {
            // to load each tile.
            char tileType = lines[y][x];
            tiles[x, y] = LoadTile(tileType, x, y);
        }
    }

    // Confirms whether the game has starting and ending point.
    //If not, the game will throw the error that will be easily identifiable.
    if (Player == null)
        throw new NotSupportedException("A level must have a starting point.");
    if (exit == InvalidPosition)
        throw new NotSupportedException("A level must have an exit point.");
}
```

Zeeshan Khalish CGA
180010339

Removal of game elements based on collision response, showing separation of collision detection and collision response code.

The way of managing the collisions response in the game-engine classes is the following:

-Collidable: This class provide the collidable objects with a method called OnCollision, that the inherited classes will have to implement with their own response to the collision.

-Collision: The Collision class has a Resolve method that calls to the OnCollision method of the collidable that has being part of a collision.

-CollisionManager: The collisions manager has a ResolveCollisions method that calls to the Resolve method of all the collisions saved on its hash map.

The collision works by using the Circle_Collison class the class has already been explained before. Below is an example of a collison working. For every diamond in the game an if statement is used where if the circle bounding the diamond intersects with the player bounded rectangle the diamond is removed the collision is checked with the method IntersectsWithRectangle() the method belongs to the Circle_Collison class. BoundingCircle creates the circle around the diamond and so does BoundingRect. A similar design approach is used for player colliding with power ups and enemies.

```
1 reference
private void UpdateDiamonds(GameTime dt)
{
    for (int i = 0; i < diamonds.Count; ++i)
    {
        Diamonds diam = diamonds[i];
        diam.Update(dt);

        if (diam.BoundingCircle.IntersectsWithRectangle(Player.BoundingRect))
        {
            diamonds.RemoveAt(i--);
            CollectDiamond(diam, Player);
        }
    }
}
```

Zeeshan Khalish CGA
180010339

Scoring system demonstrating use of event listeners

The scoring in the game works by the player collecting diamonds and coins being collected by the player and then more points are awarded depending on how much time is remaining from the game. The number of seconds that have passed since the last frame update is calculated and then multiplied by 100, so a scale can be used for point calculations.

```
public void Update(GameTime dt, KeyboardState ks, DisplayOrientation orientation)
{
    if (!Player.isAlive || TimeLeft == TimeSpan.Zero)
        Player.Physics(dt);
    else if (LevelCompleted)
    {
        //Collect points for the seconds left on the timer.
        int seconds = (int)Math.Round(dt.ElapsedGameTime.TotalSeconds * 100.0f);
        seconds = Math.Min(seconds, (int)Math.Ceiling(TimeLeft.TotalSeconds));
        timeleft -= TimeSpan.FromSeconds(seconds);
        score += seconds * timerpoints;
    }
```

If the player collides with a diamond they are rewarded points after the diamond has disappeared the code below shows the method used to increment the players score, the add points belongs to the diamonds class and it is being called in the level class. The method CollectDIamonds will only be called if the player has collided with the Diamond bounding circle. A similar principle is used for the coins as well.

```
1 reference
private void CollectDiamond(Diamonds diam, Player pl)
{
    score += Diamonds.AddPoints;
    diam.Collected(pl);
}
```

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

An XML file has been used to serialise the players score. The ScoreManager class works by using a method called LoadFile() which works by loading a file using streamReader and returning a scoreManager Object, the file is read and creates an instance of XMLSerializer, ranking is a variable with the content deserialised. SaveFile() writes to a file and serialise it. AddScore it adds a new score to the rankings list it takes a socreFile object and its to the rankings list. The image below shows the class ScoreManager working.

```
1 reference
public static ScoreManager LoadFile()
{
    if (!File.Exists(FileName))
        return new ScoreManager();

    using (StreamReader read = new StreamReader(new FileStream(FileName, FileMode.Open)))
    {
        XmlSerializer serializer = new XmlSerializer(typeof(List<ScoresFile>));

        var ranking = (List<ScoresFile>)serializer.Deserialize(read);

        return new ScoreManager(ranking);
    }
}

1 reference
public static void SaveFile(ScoreManager scoreM)
{
    using (StreamWriter write = new StreamWriter(new FileStream(FileName, FileMode.Create)))
    {
        XmlSerializer serializer = new XmlSerializer(typeof(List<ScoresFile>));

        serializer.Serialize(write, scoreM.Rankings);
    }
}
```

Then the method LoadFile() and SaveFile() are then called in the Game class where the Ranking.XML file is loaded and the players score is added to the file and then saved
The updateScores() method is called when the player finishes the game. The scoresFile class gets and sets the players name and score.

```
0 references
protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);
    Font = Content.Load<SpriteFont>("Font/gameFont");

    //Loads Main Menu
    currState = new MainMenu(this, Content, graphics.GraphicsDevice);

    rankingManager = ScoreManager.LoadFile();

}

1 reference
public void UpdateScores(int scores)
{
    rankingManager.AddScore(new ScoresFile()
    {
        Value = scores,
        playerN = "Zeeshan",
    });

    ScoreManager.SaveFile(rankingManager);
}
```

The image below shows a snippet of the Ranking XML file

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfScoresFile xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <ScoresFile>
    <Value>8000</Value>
    <playerN>Zeeshan</playerN>
  </ScoresFile>
  <ScoresFile>
    <Value>1040</Value>
    <playerN>Zeeshan</playerN>
  </ScoresFile>
```

Zeeshan Khalish CGA
180010339

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

The game uses a FSM to control which state the game is in and how it flows between other states. The State_Manager is an abstract class the method Update() updates the state, the method Draw() draws the content of the state, and the Unload() unloads tasks when transitioning between states. Each method is abstract so every class that inherits from them will have their own implementation. The different states in the game are Controls_State, HighScore_State, Game_State, Won_State, Lost_State.

```
public abstract class State_Manager
{
    protected ContentManager content;
    protected GraphicsDevice graphics;

    protected Game1 game;

    6 references
    public State_Manager(Game1 g, ContentManager cm, GraphicsDevice gd)
    {
        game = g;

        content = cm;

        graphics = gd;
    }

    7 references
    public abstract void Update(GameTime dt);

    7 references
    public abstract void Draw(GameTime dt, SpriteBatch spriteB);

    6 references
    public abstract void Unload(GameTime dt);
}
```

For example the Controls_State inherits from the class State_Manager class the constructor takes an instance of Game1, contentManager and graphics device as the paratets which are passed to the constructor of the State_Manager Class. The method Update() overrides the base Update() it checks whether the Escape key is being pressed if it is the current game state is is changed to the main menu state. The unload method() works similarly as well by overriding the base method but it instead unloads any resources that were loaded during the states lifetime so the textures and any other content have been unloaded. The Draw() method draws the the loaded texture on the screen.

```
class Controls_State : State_Manager
{
    Texture2D texture;

    1 reference
    public Controls_State(Game1 g, ContentManager cm, GraphicsDevice gd) : base(g, cm, gd)
    {
        texture = content.Load<Texture2D>("Background\\ControlsBG");
    }

    2 references
    public override void Update(GameTime dt)
    {
        if (Keyboard.GetState().IsKeyDown(Keys.Escape))
            game.ChangeCurrentState(new MainMenu(game, content, graphics));
    }

    1 reference
    public override void Unload(GameTime dt)
    {
        content.Unload();
    }

    2 references
    public override void Draw(GameTime dt, SpriteBatch spriteB)
    {
        spriteB.Begin();

        spriteB.Draw(texture, new Rectangle(0, 0, 800, 480), Color.White);

        spriteB.End();
    }
}
```

The class Game1 has a method called ChangeCurrentState() the method is responsible for transitioning states. The classes HighScore_State, Game_State, Won_State, Lost_State all work in a similar way.

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

The game has three power ups: a health boost, extra time, and invisibility. Each one has their own class; the health_Boost class draws the potion and makes it float up and down. The player class increases the health in a method called DrankHealthBoost() this method will be called in the level class when the players bounding rect intersects with Health potion Bounding Circle. The other two potions function in the same way, where time potion will add 20 extra seconds to the game time if they collide with the potion and the invisibility potion will change the players sprite opacity to make him look invisible and he will be immune to damage for 10 seconds.

```
1 reference
private void UpdateHealthPotions(GameTime dt)
{

    if (RandPotion == 0 && player.HasDrinkHealthPotion == true)
    {
        player.DrankHealthBoostRand(2);
        player.HasDrinkHealthPotion = false;
    }
    for (int i = 0; i < HealthPotions.Count; ++i)
    {
        Health_Boost health = HealthPotions[i];
        health.Update(dt);

        if (health.BoundingCircle.IntersectsWithRectangle(Player.BoundingRect))
        {
            HealthPotions.RemoveAt(i--);
            player.HasDrinkHealthPotion = true;

            if(player.HasDrinkHealthPotion == true)
            {
                CollectHealthPot(health, Player);
                player.HasDrinkHealthPotion = false;

            }
        }
    }
}
```

NPC opponents demonstrating FSM control of game objects.

There are two state classes an idle state and a chase state which both inherit from the abstract State class the class has a list of transitions that are associated with the state the add method adds a transition to the list.

```
public abstract class State
{
    4 references
    public abstract void Enter(object owner);
    3 references
    public abstract void Exit(object owner);
    3 references
    public abstract void Execute(object owner, GameTime dt);

    3 references
    public string Name
    {
        get;
        set;
    }

    private List<Transition> TransitionsList = new List<Transition>();
    1 reference
    public List<Transition> Transitions
    {
        get { return TransitionsList; }
    }

    0 references
    public void AddTransition(Transition a)
    {
        TransitionsList.Add(a);
    }

}
```

The chase state class has a method called enter which is called when the state is entered, the owner parameter represents the object that owns the state. Inside the method owner is set to Enemey_1 to access the class. The execute method called during the execution of the method,in the method owner is set to Enemy_1 class again. Exit method is called when state is exited.

```
class Chase_state : State
{
    0 references
    public Chase_state()
    {
        Name = "Chase";
    }

    3 references
    public override void Enter(object owner)
    {
        Enemy_1 enemy_1 = owner as Enemy_1;
    }

    2 references
    public override void Execute(object owner, GameTime dt)
    {
        Enemy_1 enemy_1 = owner as Enemy_1;
        if (enemy_1 == null) return;
    }

    2 references
    public override void Exit(object owner)
    {
        Enemy_1 enemy_1 = owner as Enemy_1;
    }
}
```

The IdleState class has a method called enter, Execute and Exit which act in the same way, but since this is for the enemy being idle it works by checking if the current time is greater than or equal to DirChange if it is the time is set to 0 seconds and IdleAction method is called from the Enemy_1 class if the condition has not been met Current_Timer will be elapsed.

The FSM class has a method called Initialise_State() where the state is searched for if the state has been found the Enter method is called so the state can be entered. The update method updates the state if the state is null it is returned instantly.

The enemy class has a method called State_Update() it works by If the player and enemy position on the X-Axis is outside of the range the state is set to idle. If the player is inside the range but not inside the stop distance the AI state is set to chase. If the player is within the

range and within the stop distance but not in the retreat distance the state is set to attack. If the player is within the range and within the stop distance and within the retreat distance the state is changed to retreat. If the health is less than or equal to 0 the enemy dies.