Computer Game Architecture: "Temple Platformer" coursework documentation.

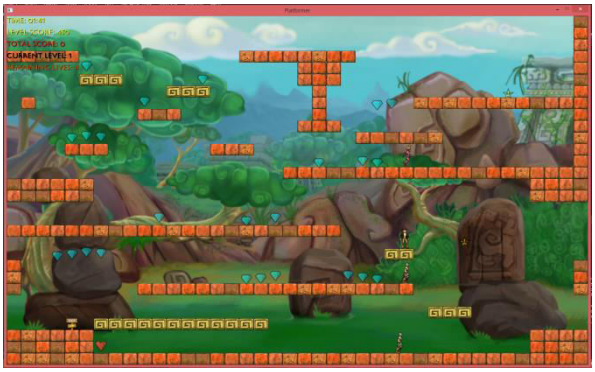
by Ziggy McLaren.

Contents

Controls and Instructions. 1
In Menu 2
In Game 2
Tasks Completed 3
Part 1 3
Load appropriate assets using a resource management strategy 3
Collision detection or alternative hit detection 5
Moving and animated game elements (frame-rate independent) 6
Part 2 7
Configurable game world (data-driven) 7
Collision detection/response (separation and removal objects) 9
Scoring System Demonstrating use of Event Listeners 11
High Score Table with Serialization 11
Part 3 12
Start/End screens with FSM and state pattern (score/re-start) 12
NPC opponent (FSM controlled) 14
Overall game-play and level challenges (e.g. timer/lives) 15
Part 4: Short essay section 16
Additional Notes 18
Bibliography 18

Controls and Instructions.

There are two controllable screens, the main menu and the gameplay screen respectively displayed below (here the gameplay screen is showing the first level).



In Menu

When navigating the main menu, the mouse cursor is visible. Moving the mouse cursor onto one of the three buttons will cause it to glow on and off. Clicking the mouse on the button will activate it. Options will do nothing at present, as it is not functional. Exit will close down the application, and play will launch a new game with the first level. To put in place the glowing functionality, and indeed for defining most of the *ClickableButton* class, some of the code in the following video was used (Oyyou).

It is also possible to navigate this menu without the mouse, using either a keyboard or a gamepad.

Actions	Keyboard Key	Gamepad Button
Select next button below	Down arrow	Down on D-pad/ Down on left joystick
Select next button above	Up arrow	Up on D-pad/Up on left joystick
Activate selected button	Enter/Space	Start/A
Close Down application	Esc	back

will override the mouse selection and then pressing enter or space will activate the selected button. Another possibility is using the gamepad I implemented in a similar way to the keyboard, where pressing up or down on the D-pad or the left joystick will override the mouse selection, then pressing start or A will activate the button.

In Game

Once the first level has launched, the player move the player character using either the keyboard or the gamepad.

Actions	Keyboard Key	Gamepad Button
Jump	Space	A
Continue (after death, time out or end of level)	Enter/Space	Start/A
Shoot (not properly implemented)	E	B
Move right	D/Right arrow	Right on D-pad/Right on left joystick/ Right trigger
Move left	A/Left arrow	Left on D-pad/Left on left joystick/ Left trigger
Close Down application	Esc	Back
Skip to next level (only to be used for testing purposes)	L	Not implemented

Previous level (only to be used for testing purposes)	K	Not implemented

Tasks Completed

Part 1

Load appropriate assets using a resource management strategy

To load the game assets, I have used a resource management strategy. I have implemented two content managers, one is the default content manager called *Content* in *PlatformerGame.cs* since it is a subclass of *Microsoft.Xna.Framework.Game*. The assets for the menus, fonts, game screens and overlays are loaded only once, with a call to *PlatformerGame*'s *LoadContent()* method, when launching the application.

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // Load fonts
    hudFont = Content.Load<SpriteFont>("Fonts/Hud");
    HighScoresFont = Content.Load<SpriteFont>("Fonts/HighScores");
    // Load overlay textures
    winOverlay = Content.Load<Texture2D>("Overlays/you_win");
    loseOverlay = Content.Load<Texture2D>("Overlays/you_lose");
    diedOverlay = Content.Load<Texture2D>("Overlays/you_died");
    //Load Menu and Game ending screens
    mainMenu = Content.Load<Texture2D>("Screens/mainMenu");
    gameOverScreen = Content.Load<Texture2D>("Screens/GameOver");
    winScreen = Content.Load<Texture2D>("Screens/Win");
    //Load textures for the main menu buttons
    startTex = Content.Load<Texture2D>("Buttons/Play");
    exitTex = Content.Load<Texture2D>("Buttons/exit");
    resumeTex = Content.Load<Texture2D>("Buttons/Resume");
    optionsTex = Content.Load<Texture2D>("Buttons/options");
    topScoresTex = Content.Load<Texture2D>("Buttons/topScores");


    //Known issue that you get exceptions if you use Media PLayer whil
    //See http://social.msdn.microsoft.com/Forums/en/windowsphone7seri
    //Which means its impossible to test this from VS.
    //So we have to catch the exception and throw it away
    try
    {
        MediaPlayer.IsRepeating = true;
        MediaPlayer.Play(Content.Load<Song>("Sounds/Music"));
    }
    catch { }

    //LoadNextLevel();
}
```

All the gameplay game objects, textures and sounds are loaded with a different content manager within *Level.cs*. All this content is disposed after each level, so space is freed and the next level is only loading the content it needs. This content manager is not only being called within one

LoadContent() method. It is being called by any methods within *Level.cs* that needs to load content, and within any other class that may need it, such as the player class, which has its *LoadContent()* method represented below.

```
public void LoadContent()
{
    // Load animated textures.
    idleAnimation = new Animation(Level.Content.Load<Texture2D>("Sprites/Player/Idle"), 0.1f, true);
    runAnimation = new Animation(Level.Content.Load<Texture2D>("Sprites/Player/Run"), 0.05f, true);
    jumpAnimation = new Animation(Level.Content.Load<Texture2D>("Sprites/Player/Jump"), 0.1f, false);
    celebrateAnimation = new Animation(Level.Content.Load<Texture2D>("Sprites/Player/Celebrate"), 0.1f, false);
    dieAnimation = new Animation(Level.Content.Load<Texture2D>("Sprites/Player/DieMod"), 0.1f, false);

    // Calculate bounds within texture size.
    int width = (int)(idleAnimation.FrameWidth * 0.4);
    int left = (idleAnimation.FrameWidth - width) / 2;
    int height = (int)(idleAnimation.FrameWidth * 0.8);
    int top = idleAnimation.FrameHeight - height;
    localBounds = new Rectangle(left, top, width, height);

    // Load sounds.
    killedSound = Level.Content.Load<SoundEffect>("Sounds/PlayerKilled");
    jumpSound = Level.Content.Load<SoundEffect>("Sounds/PlayerJump");
    fallSound = Level.Content.Load<SoundEffect>("Sounds/PlayerFall");
}
```

Control of the game character using event-driven architecture

To implement an event driven architecture for my controls, the guidance of the lab on event-driven design was followed. It wasn't straightforward though, as the original platformer code had to be broken apart to implement it, and because controls for a gamepad were also implemented which was not featured in the lab. To begin, an input listener class was implemented, which makes use of the standard XNA polling for input, and translates these inputs into events. How it does this, is that it accesses the KeyboardState, MouseState and GamePadState, which are all defined types within XNA, as shown in the following fragment of the input listener class.

```
private GamePadState PrevGamePadState { get; set; }
10 references
private GamePadState CurrentGamePadState { get; set; }
```

The input listener then needs to handle events using delegates. Custom events can be created by inheriting from XNA's EventArgs class. The classes KeyboardEventArgs, MouseEventArgs and GamePadEventArgs were implemented for this purpose. For illustration purposes, here is the GamePadEventArgs class, followed by the corresponding event handlers within the input listener class, the other two classes were implemented in a similar fashion.

```
public class GamePadEventArgs : EventArgs
{
    3 references
    public GamePadEventArgs(Buttons button, GamePadState currentGamePadState, GamePadState prevGamePadState)
    {
        CurrentState = currentGamePadState;
        PrevState = prevGamePadState;
        Button = button;
    }

    public readonly GamePadState CurrentState;
    public readonly GamePadState PrevState;
    public readonly Buttons Button;
}
```

```
//GamePad event handlers
//key is down
public event EventHandler<GamePadEventArgs> OnButtonDown = delegate { };
//key was up and is now down
public event EventHandler<GamePadEventArgs> OnButtonPressed = delegate { };
//key was down and is now up
public event EventHandler<GamePadEventArgs> OnButtonUp = delegate { };
```

The input listener then has fire event methods to dictate how to respond to each of these events. Next a command manager was implemented to receive these events, this was done in a similar manner as in the lab, except with extra methods corresponding to the delegates within the input listener, and an extra method to add gamepad bindings. Then, to handle game actions, an extra dictionary to handle the gamepad buttons was added.

Within the PlatformerGame class, the method InitialiseBindings() called the addKeyboardBinding and addGamePadBinding methods from the command manager. Each binding associated a key (or button) with an action, that we also defined within the PlatformerGame class in their own region.

Implementing the actions so they behave differently depending on which screen is active and so they keep working smoothly when changing levels was quite tricky. For example the input for the jump action (shown below) is implemented to call the continue action instead of jumping if the level is complete. This caused accidental strange effects such as jumping at the start of a new level. To remedy this, a flag preventing the player from jumping when a new level has just begun was added. Other flags had to be added to avoid other unpleasant effects. The continue action only has an effect at certain points in gameplay such as at death, end of level or time out. This is why it can be called in all other cases. Other actions for movement in the main menu and in game, and a click action for mouse clicks when navigating the main menu were implemented. If desired, the up arrow on keyboard or D-pad can easily be set to jump as it is already implemented but commented out within the MoveUp game action.

```
public void Jump(eButtonState buttonState, Vector2 amount)
{
    if (fsm.GetCurrentState().Name == "Game")
    {
        if ((buttonState == eButtonState.DOWN || buttonState == eButtonState.PRESSED) && level.Player.IsAlive && !level.ReachedExit
        && (level.TimeRemaining.CompareTo(TimeSpan.FromMinutes(1.99)) < 0) && !(level.TimeRemaining == TimeSpan.Zero))
        {
            level.Player.IsJumping = true;
        }
        else
        {
            level.Player.IsJumping = false;
            Continue(buttonState, amount);
        }
    }
    else if (fsm.GetCurrentState().Name == "MainMenu")
    {
        Continue(buttonState, amount);
    }
}
```

Collision detection or alternative hit detection

All objects in the game that required collision detection were made to inherit from an abstract collidable class. This way, a method testing for collisions is implemented within each collidable as can be seen in the code fragment from the Player class below.

```
public override bool CollisionTest(Collidable obj)
{
    if (obj != null)
    {
        PowerUp powerUp = obj as PowerUp;
        if (powerUp != null)
        {
            return powerUp.BoundingCircle.Intersects(boundingRectangle);
        }
        else
            return boundingRectangle.Intersects(obj.boundingRectangle);

    }

    return false;
}
```

This method returns whether the player's bounding rectangle intersects with another bounding rectangle, or a bounding circle in the case of a power up, which class also inherits from collidable.

One exception is the collisions with the tiles in the levels, which are handled within a method HandleCollisions() within the necessary objects. Unfortunately an issue that sometimes arises because of this implentation is that the player may fall through or jump through certain tiles that should be impassable when the game is running in variable time.

The *CollisionManager* is implemented to maintain a list of all the entities that can collide, check collisions between them, and then resolve them. The CollisionManager also has a hashset to handle the collisions in a single update implemented in a similar fashion as the lab on collision detection.

The platformer that began as a template to this game was actually quite tricky and required modifying a lot of the platformer code to make it work. I did this successfully for all collidables except I left the collisions for the tiles to be handled by the original code.

I defined the abstract class *Collidable.* and made the classes *Player, Enemy* and *PowerUp all inherit from it.* Each of these classes has a bounding rectangle to test for collisions except the power up class which uses a bounding circle instead.

Moving and animated game elements (frame-rate independent)

In the original implementation of the platformer game, certain movements and collisions act differently when variable time is turned on, showing frame-rate dependence. In this implementation, the movements are factors of time and handle correctly. For example the velocity in the player class below is modified as a factor of time to ensure frame-rate independence.

```
#region Apply Physics, jumping and shooting
/// <summary>
/// Updates the player's velocity and position based on input, gravity, etc.
/// </summary>
2 references
public void ApplyPhysics(GameTime gameTime)
{
    float elapsed = (float)gameTime.ElapsedGameTime.TotalSeconds;

    Vector2 previousPosition = Position;

    // Base velocity is a combination of horizontal movement control and
    // acceleration downward due to gravity.
    velocity.X += movement * MoveAcceleration * elapsed;
    velocity.Y = MathHelper.Clamp(velocity.Y + GravityAcceleration * elapsed, -MaxFallSpeed, MaxFallSpeed);
    velocity.Y = DoJump(velocity.Y, gameTime);
    Shoot(gameTime);
```

Unfortunately there are still issues with the original implementation of the collisions with tiles and even the physics engine within Player which would need some time to fix.

The animated game objects in the game are the player, the enemies and the power ups. The power up animation is the same for all power ups but could easily be overridden in the inheriting classes such as invincibility and life. These power ups inherit from the *PowerUp* class (this used to be the gem class).

In other animations, It was planned to have the player be able to throw a shuriken, which could easily be adapted later to have some enemies shoot projectiles. This is not fully functional at present, but the code structure is there to enable this. There is a shuriken class, inheriting from the collidable class and a shoot method as well as bindings associated with it in the command manager (E on keyboard).

Part 2

Configurable game world (data-driven)

The data for each level is contained within a text file. The data for certain game object variables such as colour and speed are contained within an XML file. Modifying the levels is easy, as all is needed is changing one of the characters in the text files associated with the levels. Taking this further than the original platformer, I added a few more methods in *Level.cs* to be able to add tiles for the extra power ups which are invincibility and extra lives. To add these in the level, one only needs to change a character in the associated text file based on this following fragment of the level class.

```
        // Blank space
        case '.':
            return new Tile(null, TileCollision.Passable);

        // Exit
        case 'X':
            return LoadExitTile(x, y);

        // Gem
        case 'G':
            return LoadGemTile(x, y);

        //Invicibility pickup
        case 'I':
            return LoadInvicibilityTile(x, y);

        //Life pickup
        case 'L':
            return LoadLifeTile(x, y);

        // Floating platform
        case '-':
            return LoadTile("Platform", TileCollision.Platform);
```

5 new levels were added and the original backgrounds were copied and modified to suit those new levels. Modifying the variables within the XML file is easy, all is needed is to modify the values in the following XML fragment. This is to enable easy access to enemy speed, power up colours and the point values of these game objects.

```
<EnemyInfo>
  <PointValue> 100</PointValue>
  <Speed>100</Speed>
  <FastSpeed> 250</FastSpeed>
  <SlowSpeed>35</SlowSpeed>
  <InsaneSpeed>750</InsaneSpeed>
</EnemyInfo>

<PowerUpInfo>
  <PointValue>0</PointValue>
  <BounceHeight>0.18</BounceHeight>
  <BounceRate>3.0</BounceRate>
  <Color>
    <R>0</R>
    <G>127</G>
    <B>128</B>
    <A>127</A>
  </Color>
</PowerUpInfo>

<InvincibilityInfo>
  <PointValue> 0</PointValue>
  <Timer> 7.0</Timer>
  <Color>
    <R>150</R>
    <G>127</G>
    <B>0</B>
    <A>127</A>
  </Color>
</InvincibilityInfo>
```

Collision detection/response (separation and removal objects)

The OnCollision method within the player class shown below does most of the hard work as it can collide with any of the various power ups or enemies.

```
public override void OnCollision(Collidable obj)
{
    Enemy enemy = obj as Enemy;
    if (enemy != null && !enemy.dead)
    {
        if (isInvincible)
        {
            level.Score += GameInfo.Instance.EnemyInfo.PointValue;
            enemy.dead = true;
            enemy.OnKilled();
        }
        else
            this.OnKilled(enemy);
    }
    PowerUp powerUp = obj as PowerUp;
    if (powerUp != null)
    {
        if (powerUp.removed == false)
        {
            Gem gem = powerUp as Gem;
            Invincibility inv = powerUp as Invincibility;
            Life life = powerUp as Life;
            if (inv != null)
            {
                if (isInvincible)
                    invincibilityTimer += GameInfo.Instance.InvincibilityInfo.Timer;
                isInvincible = true;
            }
            else if (gem!=null)
            {
                    level.Score += GameInfo.Instance.GemInfo.PointValue;
            }
            else if (life!=null)
            {
                PlatformerGame.numberOfLives++;
            }
            powerUp.OnCollected(this);
        }
        powerUp.removed = true;
    }
}
```

Whenever there is a collision, an event is fired and the appropriate OnCollision methods are executed. In this case most of the handling is done from the player class.

As this game is quite simple, this implementation is sufficient. Another, perhaps better implementation would be leaving the details for each collidable the player can collide with within their own OnCollision() method. As in general a collidable object should handle the details of what happens to them when they collide.

Enemies can collide freely but to avoid groups of enemies on the same platform grouping together permanently after fleeing or chasing a player, their OnCollision method registers an obstruction if the other enemy is in front of it. This means once the enemies walk in a different direction, the one in front will not be obstructed and they will space out.

```
public override void OnCollision(Collidable obj)
{
    Enemy enemy = obj as Enemy;
    if (enemy != null)
    {
        if ((direction == FaceDirection.Right && enemy.Position.X > Position.X) || (direction == FaceDirection.Left && enemy.Position.X < Position.X))
        {
            obstructed = true;
        }
        else
            obstructed = false;
    }
}
```

As mentioned in Part1 "Collision detection or Alternative hit detection", the collision detection and response for the tiles in the level are dealt with separately. This causes some issues in variable time but otherwise works properly.

Scoring System Demonstrating use of Event Listeners

There isn't a scoring specific event listener in this game, instead the collision manager handles this and the scoring is dealt with within the OnCollision method of the player class (shown above within the collision response section). Every second remaining on the level timer at the end of a level also gives a certain amount of points (5 by default), this is handled separately. Each gem collected or enemy killed also gives points and those values are set within the XML file and then called inside the OnCollision method. At the end of each level, the score is returned to 0 and the total score is incremented accordingly.

High Score Table with Serialization

A list of ints was created to store the high scores. On the main menu, a texture with its own font displays the high scores in the appropriate order. The way this is done is within a DrawHighScores method (shown below) called every time the PlatformerGame Draw method is called if the main menu is active. (Note. There are three high scores by default that are being added within the PlatformerGame class constructor)

```
private void DrawHighScores()
{
    if (HighScores.Count > 0)
    {
        //This orders the highscores so that the top score is at the top on the main menu.
        HighScores.Sort();
        HighScores.Reverse();
        for (int i = 0; i < HighScores.Count; ++i)
        {
            if (i < 5)
            {
                DrawShadowedString(HighScoresFont, HighScores[i].ToString(), topScoresPosition + new Vector2(150.0f,100.0f*(i+1)), Color.Red);
            }

        }
    }
}
```

This method sorts the high scores in ascending order then reverses them so that they are in descending order instead. The high score table on the main menu is limited to only the top 5 high scores, for space reasons. Unfortunately, there was not enough time to implement serialisation for the high scores, but this would be handled fairly similarly to the level loading. The high scores could

first be written to a text file and then read from that text file whenever needed so that one may retain them even after closing the application.

Part 3

Start/End screens with FSM and state pattern (score/re-start)

Menu and game screens using FSM. options isn't functional and I had the ambition to add pause and resume to gameplay but did not have time. There is also a game over screen and a victory screen, shown here, which don't have any functionality except that they disappear after a certain amount of time (set to 6 seconds at present).

ClickableButton was based off cButton class in this video, the rest of the code wasn't used as the different menu screens were handled with the FSM class whereas this youtuber used an enum instead (Oyyou).

The transitions in the FSM are handle within a method called InitialiseFSM shown below which is called only once in the PlatformerGame Initialise method.

```
public void InitialiseFSM()
{
    fsm = new FSM(this);

    // Create the states
    GameState game = new GameState();
    OptionsState options = new OptionsState();
    MainMenuState mainMenu = new MainMenuState();
    GameOverState gameOverState = new GameOverState();
    WinState win = new WinState();

    // Create the transitions between the states
    mainMenu.AddTransition(new Transition(game, () => (clickableButtons[0].isClicked == true)));
    mainMenu.AddTransition(new Transition(game, () => (clickableButtons[0].isActivated == true)));
    game.AddTransition(new Transition(win, () => (endOfGame==true)));
    game.AddTransition(new Transition(gameOverState, () => (gameOver == true)));
    gameOverState.AddTransition(new Transition(mainMenu, () => (gameOverTimer <= TimeSpan.Zero)));
    win.AddTransition(new Transition(mainMenu, () => (gameOverTimer <= TimeSpan.Zero)));


    // Add the created states to the FSM
    fsm.AddState(mainMenu);
    fsm.AddState(game);
    fsm.AddState(options);
    fsm.AddState(gameOverState);

    // Set the starting state of the FSM
    fsm.Initialise("MainMenu");
}
```

The game over timer mentioned above is used whether the game is over due to failure or victory. The FSM class is implemented similarly to the one in the relevant lab, and only has minor changes. The MainMenuState, GameState, GameOverState and WinState all inherit from a State class. The MainMenuState makes the mouse cursor visible on entry and invisible on exit. The GameState initialises a new game on entry and adds the total score to the high scores on exit. The GameOverState and WinState are mostly empty classes apart from defining their name. (Note. to test if the win state screen works correctly one can use L repeatedly to skip to the last level and perhaps also spawn the player next to the end of the level if that level is too hard or long to complete)

Power-ups (with event-listeners and base-class use)

The power ups in this game all inherit from a power up class which itself inherits from the collidable class. The power up class sets up most of their functionality such as their bouncing pattern and used to be the gem class in the platformer game used as a template, the code inside it was not modified much. The child classes Gem, Invincibility and Life modify the point values, required textures or particular effects. The gems give 30 points on collection, the invincibility power up makes the player invincible for 7 seconds, during which they are able to kill the enemies and acquire more points that way. The life power up looks like a heart and gives the player an extra life.

Power ups are dealt with the same event-listener as the score and the collisions, the collision manager. Similarly to the score, the effects of the power ups such as giving an extra life happen within the player's OnCollision method, which was featured above. The actual life class only contains the collected sound and the texture of the life power up as can be seen in its LoadContent method below.

```
4 references
public override void LoadContent()
{
    texture = Level.Content.Load<Texture2D>("Sprites/PowerUps/Life");
    origin = new Vector2(texture.Width / 2.0f, texture.Height / 2.0f);
    collectedSound = Level.Content.Load<SoundEffect>("Sounds/OhYeah");
}
```

NPC opponent (FSM controlled)

The enemies, like the menu screens, are controlled with a FSM, which is contained in the enemy class. They have the FleeState, IdleState and ChaseState classes implemented, which all inherit from the State class. When in idle state, the enemies wander back and forth along their platform, waiting a short time before turning around each time. The enemies have a circle called sensor, which detects if the player is within a certain range and goes in their direction, or runs away if the player has picked up the invincibility power up. Every update, the sense method, the think method and then the move method are called. The sense method uses the sensor to check if it intersects with the player's bounding rectangle. The think method updates the fsm appropriately and thus may change the enemy state. The move method then moves the enemy depending on what state they are in. The update method in the enemy class is shown here.

```
public virtual void Update(GameTime gameTime)
{
    Sense();
    Think(gameTime);
    Move(gameTime);

    sensor.Center = Position;
}
```

The enemy class's Initialisation method below is called once per enemy and adds all the necessary transitions to its fsm.

```
public virtual void Initialise()
{
    fsm = new FSM(this);
    // Create the states
    IdleState idle = new IdleState();
    FleeState flee = new FleeState();
    ChaseState chase = new ChaseState();
    // Create the transitions between the states
    idle.AddTransition(new Transition(flee, () => (playerSeen && level.Player.IsInvincible)));
    idle.AddTransition(new Transition(chase, () => (playerSeen && !level.Player.IsInvincible)));
    flee.AddTransition(new Transition(idle, () => !playerSeen));
    flee.AddTransition(new Transition(chase, () => !level.Player.IsInvincible));
    chase.AddTransition(new Transition(idle, () => !playerSeen));
    chase.AddTransition(new Transition(flee, () => level.Player.IsInvincible));
    // Add the created states to the FSM
    fsm.AddState(idle);
    fsm.AddState(flee);
    fsm.AddState(chase);
    // Set the starting state of the FSM
    fsm.Initialise("Idle");
    //initialise the sensor
    sensor = new Circle(Position, sensorLength);
    dead = false;
}
```

There was an attempt to improve the enemies behaviour by implementing a similar physics system as in the player class so that one could use the same actions as the player uses for movement or to make them jump. This was abandoned but can be seen in the advanced enemy class, and it is possible to add advanced enemies to the levels, they behave like the standard enemies but they can move off platforms. The jumping mechanic was unfortunately not successfully implemented.

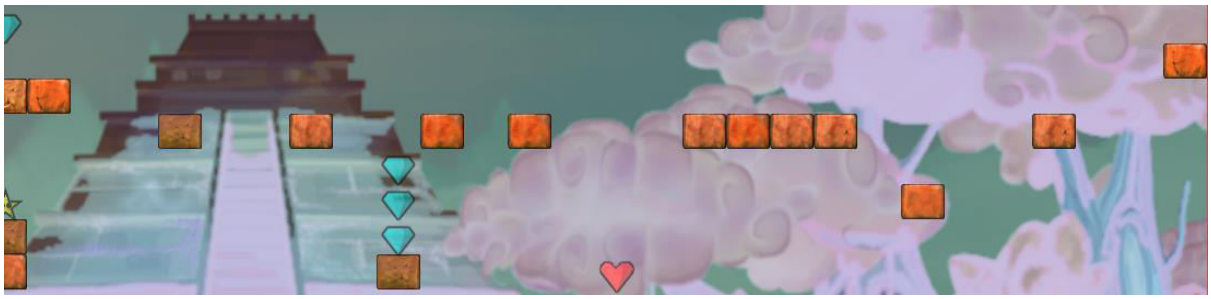
Overall game-play and level challenges (e.g. timer/lives)

The game contains 8 levels and the player begins the game with 5 lives. The objective of the player is to get to the end of each level within the time limit, whilst acquiring the most points they can by collecting gems or killing enemies and without losing any lives if possible. The enemies patrol around the levels and they will chase the player if they get within a certain radius of them. If the player loses all their lives, the game ends and goes to a game over screen. The high score is still added to the high score table but doesn't contain the last played level's score. Every level has a time limit, displayed in the HUD, as shown here.

The difficulty scales and forces to player to figure out the best way to get past an enemy or a tricky platforming area. For example in level 4, the timing of the jumps to avoid this enemy above the

player shown below is critical.

If the player is fast enough to jump out of their boxed area, they have to time their jumps because of the obstructing tiles coming down from above. If the player is too slow to leave their boxed area, the enemy may be blocking the way up. To avoid the player being stuck, a single tile was placed (circled in red). The player can thus attract the enemy to the left and then run right, jump up twice to find the safety of the platform and once the enemy is below them, they can run left to get past it. Certain levels like the final level have quite tricky platforming sections such as the one shown here.



Here the player must jump to the right onto these single tiles. If they miss they risk falling to their death or at the very least land on a previous platform below which will cause them to fall behind on time, since they have to climb back up to this section again. The life power up placement here is a sadistic temptation, since it is not really necessary to collect it and there is no way to get it without having to climb back up round again. But for the experts looking for the best score possible, the extra life will give an extra end game bonus of 200 points (by default, but can be modified within info.xml)

The 8 levels have been tested to ensure there is no way to be completely stuck with no chance of success, and to ensure there are harder to reach areas that contain extra rewards such as a life power up. The default top score of 20000 points is reasonably easy to beat on a good playthrough even with a few deaths.

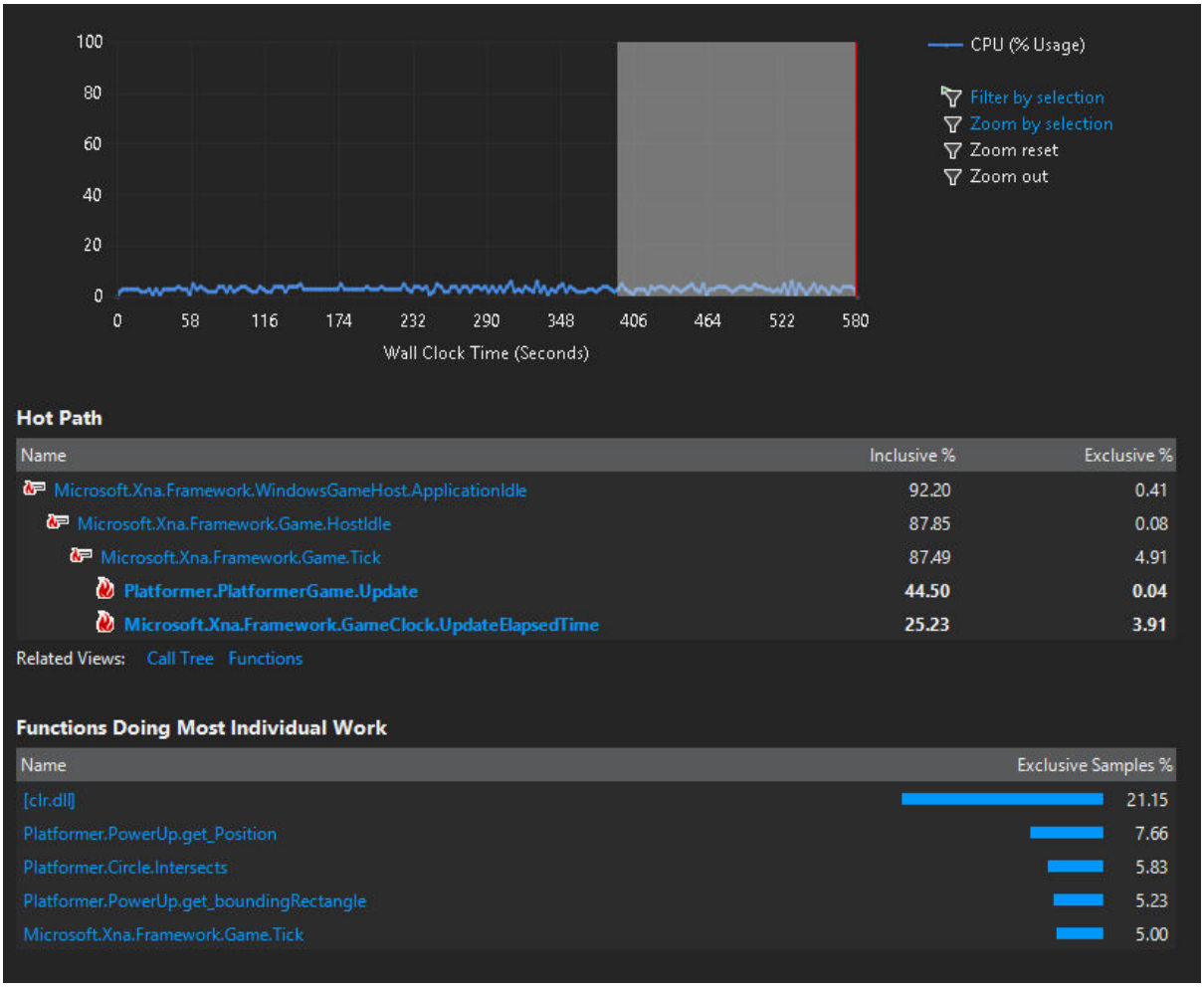
Part 4: Short essay section

A discussion of the use of profiling software to improve the performance of the game engine.

Games are real-time systems, so achieving and maintaining a high frame rate is important. But how do we know which parts to optimise? One solution is to use a profiler. This is a tool that measures the execution time of our code and can tell in which functions most of the processing time is spent.

There are different types of profilers: Call-graph based profilers, which give graphical representations of the CPU usage and the time spent within each function; statistical profilers, which run alongside the game and don't slow it down but give less precise results and instrumenting profilers which are the slowest but most detailed type of profiler.

Visual studio has a few built in profilers. Running the instrumenting profiler was very slow and made the game unplayable, for this reason it should only be used after using a less detailed profiler to be able to get more detail once certain bottleneck areas have been spotted. The built in call-graph based profiler was then run, and the game was played through from start to finish to get a proper assessment of all parts of the game. This gives us a good picture of the most used functions within the program. It's results are shown here.



It can be seen that the most called functions (after clr.dll which handles the running of the game) are related to the PowerUp class. This makes sense, as in each level it is the most frequent type of game object and its position is always moving due to the "bouncing" effect. Also, the collision detection is a common efficiency bottleneck in many games. This is usually due to measuring the length between two positions to compare directly and thus computing a square root. This can be avoided by

comparing the squares instead which is much more efficient. In this game, the collisions are computed using XNA's *Intersects* method. It can be assumed it is fairly efficient, but it may still be computing square roots and thus this part of the code could be improved.

If we were to optimise our code, the focus should be on optimising those functions, since they are used the most. It is clear that a profiler is the most practical tool to pin point these efficiency bottlenecks, and saves a lot of time when optimising. It avoids wasting times optimising functions that don't have much effect on the total efficiency. Here the optimisation is only done for curiosity's sake, but if this game were to target a certain platform, then efficiency would be critical and the more detailed profilers such as the instrumenting profiler may become essential to the optimisation process.

Additional Notes

All assets for this game were provided by the platformer template code or by my own making. The backgrounds were copied and modified to use in later levels. The main menu and end of game screens were made by me as well as the life and invincibility power ups, although the shading was based on the original gems' shading from the template. The "OhYeah" sound effect when picking a life was also recorded by myself.

Bibliography

Oyyou. (n.d.). *Youtube: XNA Tutorial 28 - Main Menu System.* Retrieved 04 2015, from Youtube: https://www.youtube.com/watch?v=54L_w0PiRa8