|  |
| --- |
| University of Canberra |
| Assignment 2 - Technical Design Document |
| Advanced Game Programming - 9746 |

|  |
| --- |
| Unit Convener: Dr Reza Ryan |
| Mitchell Phillips – u3202396  4-21-2020 |

# Table of Contents

[Table of Contents 2](#_Toc40110930)

[Table of Figures and Tables 3](#_Toc40110931)

[Game Analysis 4](#_Toc40110932)

[Prototype Game Requirements 4](#_Toc40110933)

[Use Case Diagrams 8](#_Toc40110934)

[Game Design 9](#_Toc40110935)

[Architectural Design 9](#_Toc40110936)

[Design Document 9](#_Toc40110937)

[Data Structure Design 9](#_Toc40110938)

[Entity-Relationship Diagrams 9](#_Toc40110939)

[Algorithm Design (Pseudocode) 10](#_Toc40110940)

[Game Implementation 11](#_Toc40110941)

[Code with Comments 11](#_Toc40110942)

[Testing and Verification 22](#_Toc40110943)

[Test Plan 22](#_Toc40110944)

[Error Report 22](#_Toc40110945)

[Improvement Report 24](#_Toc40110946)

[Bibliography 26](#_Toc40110947)

# Table of Figures and Tables

[Figure 1: Priority Backlog List 7](#_Toc40110909)

[Figure 2: Player Use Case Diagram 8](#_Toc40110910)

[Figure 2: Inheritance System 9](#_Toc40110911)

[Figure 4: SpawnObject.cs 11](#_Toc40110912)

[Figure 5: SpawnRooms.cs 11](#_Toc40110913)

[Figure 6: RoomType.cs 12](#_Toc40110914)

[Figure 7: FinishAndGoToNextLevel.cs 12](#_Toc40110915)

[Figure 8: LevelGeneration.cs – Part 1 13](#_Toc40110916)

[Figure 9: LevelGeneration.cs – Part 2 14](#_Toc40110917)

[Figure 10: LevelGeneration.cs – Part 3 15](#_Toc40110918)

[Figure 11: LevelGeneration.cs – Part 4 16](#_Toc40110919)

[Figure 12: LevelGeneration.cs – Part 5 17](#_Toc40110920)

[Figure 13: LevelGeneration.cs – Part 6 18](#_Toc40110921)

[Figure 14: Entity.cs 18](#_Toc40110922)

[Figure 15: Unit.cs 19](#_Toc40110923)

[Figure 16: Player.cs – Part 1 20](#_Toc40110924)

[Figure 17: Player.cs – Part 2 21](#_Toc40110925)

[Figure 18: Table of Windows Computers used in testing 22](#_Toc40110926)

[Figure 19: Table of Test Functionalities 23](#_Toc40110927)

[Figure 20: Bug-List Table 23](#_Toc40110928)

[Figure 21: Improvement Report 25](#_Toc40110929)

# Game Analysis

## Prototype Game Requirements

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ID | User Story (as a player of this game) | Tasks | Time Estimation (Hours) | Priority  1 = Very High  2 = High  3 = Medium  4 = Medium to Low  5 = Low |
| 1 | I want to be able to enjoy unique and interesting levels. | Create 1-5 tiles that will become the base environment for the game.  One tile will be empty for later use. | .16  10 Minutes | 1 |
| Create Zones with openings in the:  Left-Right  Left-Right-Top  Left-Right-Bottom  Top-Bottom  Top-Bottom-Left  Top-Bottom-Right  Left-Right-Top-Bottom  By placing empty game objects in 10x10 grid. | .5  30 Minutes | 1 |
| Attach an object spawning script to the empty game objects.  This will randomize which non-empty tile will spawn at these locations. | .16  10 Minutes | 1 |
| Create an Empty game object at positioned at 0,0,0 that will become the level generator. Rename it to LevelGenerator | .1  5 Minutes | 2 |
| Create 4 empty game objects,  Rename to Pose 1-4,  Place them equally apart. | .1  5 Minutes | 2 |
| On the LevelGenerator object, create a Script that finds one of 4 starting positions randomly.  Continue with creating an array of game objects that are to be filled with our zones.  Next add variables, for level generation including limits, a stop generation bool and a layer mask.  Finalise by writing the methods of controlling the generation. | 4 Hours | 1 |
| 2 | I expect that the level will be filled out even if the pathway is completed. | In Unity add 12 more Pose game objects named Pose 5-16 located correctly.  On the Pose objects create a script that will check if there is a room on top of it.  If there is, do nothing.  Otherwise spawn a random room, then destroy this game object. | 2 Hours | 2 |
| 3 | I expect that after the level is filled that If a room spawns on an outside area of the level that if it is open to open space. | Create a border around the level area that will restrict the player to the level. | .16  10 Minutes | 3 |
| 4 | I expect that I will be able to play as a character. | Create an empty game object,  Rename to Player,  Add an empty game object as a child rename to sprite,  Add a sprite renderer component to the Sprite game object. | .16  10 Minutes | 3 |
| 5 | I expect that the character that I will play will be able to traverse through the level using control for the platform of the game. | Create a character movement script that will use Unity’s Horizontal and Vertical Input system (the Arrow keys or WASD)  I will also add the space as a jump so that up and down can be used if I choose to have vines or ladders in the game. | 2 Hours | 2 |
| Create an empty game object,  Rename it ground check and move it to be a child of the player.  Move its position to be at the bottom of the player’s sprite.  This will be used to check if we are standing on the ground, and if we are, we have the option to jump, this makes it so that when we jump it will not just keep jumping through the roof.  Modify the character movement script to check if the ground check is touching the ground. | 1 Hour | 2 |
| Add components to the player Game objects.  RigidBody2D,  2 Colliders, a box for the top half of the character, a circle for the legs and feet, this allows for smoothness should I add diagonal tiles.  Modify the character movement script to use these components.  Freeze the rotation on the rigidbody2D component.  Create a Physics Material, make sure the material has no friction or bounciness, this will stop the player from sticking to walls mid-air. Add this to the collider components. | 1 Hour | 2 |
| 6 | I expect that the game I am playing has a way to track what state the game is in.  Initialization, Game, End Good, End Bad. | Create an empty game object.  Rename to GameManager  Create a script called GameManager.  This script will keep track of the game state, spawning the player, ending the level, and any scoring for the game.  Spawn the player in the first zone created. | 4 Hours | 1 |
| 7 |  |  |  |  |

Figure 1: Priority Backlog List

# Use Case Diagram for the PlayerUse Case Diagrams

Figure 2: Player Use Case Diagram

# Game Design

* Architectural Design
  + Component Design
  + Interface Design
* Data Structure Design
* Algorithm Design (Pseudocode)

## A close up of a piece of paper Description automatically generatedArchitectural Design

Figure 2: Inheritance System

## Design Document

**Entity:**

* Int, String

**Unit:**

* Float, Bool, RigidBody2D

**Player:**

* Float, Transform, LayerMask, Bool

**SpawnObject:**

* GameObject Array, Random

**SpawnRooms:**

* LayerMask, LevelGeneration, Collider2D, Random

**RoomType:**

* Int, GameObject

**FinishAndGoToNextLevel:**

* String, GameObject, Collider2D, SceneObject

**LevelGeneration:**

* Transform Array, GameObject Array, Int, Float, Bool, LayerMask, List<Transform>, GameObject, CinemachineVirtualCamera, Random

## Data Structure Design

## Entity-Relationship Diagrams

## Algorithm Design (Pseudocode)

# Game Implementation

## Code with Comments

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class SpawnObject : MonoBehaviour

{

// fill this array of objects in the Unity Inspector

public GameObject[] objects;

void Start()

{

// choose a random object from the array, spawn it, then make it a child of the object that has this script on it

int rand = Random.Range(0, objects.Length);

GameObject instance = (GameObject)Instantiate(objects[rand], transform.position, Quaternion.identity);

instance.transform.parent = transform;

}

}

Figure 4: SpawnObject.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class SpawnRooms : MonoBehaviour

{

public LayerMask whatIsRoom;

public LevelGeneration levelGeneration;

void Update()

{

// check if there is a room at this location, then if there isn't

Collider2D roomDetection = Physics2D.OverlapCircle(transform.position, 1, whatIsRoom);

if(roomDetection == null && levelGeneration.stopGeneration == true)

{

// Spawn a random room and destroy the object this script is on

int rand = Random.Range(0, levelGeneration.rooms.Length);

Instantiate(levelGeneration.rooms[rand], transform.position, Quaternion.identity);

Destroy(gameObject);

}

}

}

Figure 5: SpawnRooms.cs

Figure 6: RoomType.cs

using UnityEngine;

using UnityEngine.SceneManagement;

public class FinishAndGoToNextLevel : MonoBehaviour

{

// name the scene in the Unity inscpector

[SerializeField] private string levelName;

[SerializeField] private GameObject finishParticle;

[SerializeField] private SpriteRenderer renderer;

private bool finished = false;

void OnTriggerEnter2D(Collider2D other)

{

// Check if the player touched this object, if they did do something, otherwise do nothing

if (other.CompareTag("Player") && finished == false)

{

// Invoke the next level after 2 seconds and instantiate particle effect

Invoke("NextLevel", 2f);

// Make the item disappear

renderer.color = new Color(0, 0, 0, 0);

// Make sure the player cannot touch it again

finished = true;

Instantiate(finishParticle, transform.position, Quaternion.identity);

}

else

return;

}

// go to the scene named in the Unity inspector

void NextLevel() {

SceneManager.LoadScene(levelName);

}

}

using UnityEngine;

public class RoomType : MonoBehaviour

{

// type of room set in the Unity inspector

public int type;

// if I need to destroy a room, call this method

public void RoomDestruction()

{

Destroy(gameObject);

}

}

Figure 7: FinishAndGoToNextLevel.cs

Figure 8: LevelGeneration.cs – Part 1

using System.Collections.Generic;

using UnityEngine;

using Cinemachine;

/// <summary>

/// Author: Mitchell Phillips

/// Description: This level generation code is designed to follow the game Spelunky's level generation (G, 2016)

/// Reference Link: https://www.youtube.com/watch?v=Uqk5Zf0tw3o

/// </summary>

public class LevelGeneration : MonoBehaviour

{

#region Variables

[Header("Level Generation:")]

[SerializeField] private Transform[] startingPositions;

public GameObject[] rooms; // index 0 = LR, index 1 = LRB, index 2 = LRT, index 3 = LRTB

private int direction;

[SerializeField] private float moveAmount;

private float timeBetweenRoom;

[SerializeField] private float startTimeBetweenRoom = 0.25f;

[SerializeField] private float minX;

[SerializeField] private float maxX;

[SerializeField] private float minY;

public bool stopGeneration;

[SerializeField] private LayerMask room;

private int downCounter;

[SerializeField] private List<Transform> pathIndex = new List<Transform>(); // Use this to find the beginning and end of the path

[Header("Player Object:")]

[SerializeField] private GameObject player;

private bool playerSpawned = false;

[Header("Level End:")]

[SerializeField] private GameObject levelEndDiamond;

[Header("Camera Options:")]

[SerializeField] private CinemachineVirtualCamera vcam;

#endregion

Figure 9: LevelGeneration.cs – Part 2

#region Start & Update Functions

void Start()

{

// chose a random starting position for this object, spawn a room of type [0] at this location

int randomStartingPositions = Random.Range(0, startingPositions.Length);

transform.position = startingPositions[randomStartingPositions].position;

GameObject spawnedRoom = Instantiate(rooms[0], transform.position, Quaternion.identity) as GameObject;

// Add the spawned room to the pathIndex List

pathIndex.Add(spawnedRoom.transform);

spawnedRoom.name = "Path: " + pathIndex.Count;

// Chose a new random direction

direction = Random.Range(1, 6);

}

private void Update()

{

// only spawn a room when time is 0 and stopGeneration is set to false, otherwise move on

if(timeBetweenRoom <= 0 && stopGeneration == false)

{

Move();

timeBetweenRoom = startTimeBetweenRoom;

}

else

{

timeBetweenRoom -= Time.deltaTime;

}

// when the level has finished generating

if(stopGeneration == true && playerSpawned == false)

{

// spawn the player in the first PATH room spawned

player = Instantiate(player, pathIndex[0].position, Quaternion.identity);

playerSpawned = true;

// spawn the level end in the last PATH room spawned

levelEndDiamond = Instantiate(levelEndDiamond, new Vector2(pathIndex[pathIndex.Count - 1].position.x, pathIndex[pathIndex.Count - 1].position.y + 1), Quaternion.Euler(0,0,45));

// make the camera follow the instantiated player

vcam.Follow = player.transform;

}

}

#endregion

Figure 10: LevelGeneration.cs – Part 3

#region Level Generation

private void Move()

{

if(direction == 1 || direction == 2) // Move RIGHT

{

if (transform.position.x < maxX)

{

downCounter = 0;

// Move this object to the new location

Vector2 newPos = new Vector2(transform.position.x + moveAmount, transform.position.y);

transform.position = newPos;

// Spawn room with left and right openings

int rand = Random.Range(0, rooms.Length);

GameObject spawnedRoom = Instantiate(rooms[rand], transform.position, Quaternion.identity) as GameObject;

// Add the spawned room to the pathIndex List

pathIndex.Add(spawnedRoom.transform);

spawnedRoom.name = "Path: " + pathIndex.Count;

// Make sure that generation cannot go back on itself

direction = Random.Range(1, 6);

if(direction == 3)

{

direction = 2;

}

else if(direction == 4)

{

direction = 5;

}

}

else

{

direction = 5;

}

}

Figure 11: LevelGeneration.cs – Part 4

else if(direction == 3 || direction == 4) // Move LEFT

{

if (transform.position.x > minX)

{

downCounter = 0;

// Move this object to the new location

Vector2 newPos = new Vector2(transform.position.x - moveAmount, transform.position.y);

transform.position = newPos;

// Spawn room with left and right openings

int rand = Random.Range(0, rooms.Length);

GameObject spawnedRoom = Instantiate(rooms[rand], transform.position, Quaternion.identity) as GameObject;

// Add the spawned room to the pathIndex List

pathIndex.Add(spawnedRoom.transform);

spawnedRoom.name = "Path: " + pathIndex.Count;

// Make sure that generation cannot go back on itself

direction = Random.Range(3, 6);

}

else

{

direction = 5;

}

}

Figure 12: LevelGeneration.cs – Part 5

else if(direction == 5) // Move DOWN

{

downCounter++;

if (transform.position.y > minY)

{

// Check that the room has a lower opening before moving down, if it doesn't destroy it and spawn one that does

Collider2D roomDetection = Physics2D.OverlapCircle(transform.position, 1, room);

if(roomDetection.GetComponent<RoomType>().type != 1 && roomDetection.GetComponent<RoomType>().type != 3)

{

// Make sure that when going down a second plus time, that the room has openings at the top and bottom

// otherwise destroy it and try again

if(downCounter >= 2)

{

pathIndex.Remove(pathIndex[pathIndex.Count - 1]);

roomDetection.GetComponent<RoomType>().RoomDestruction();

GameObject spawnedRooom = Instantiate(rooms[3], transform.position, Quaternion.identity) as GameObject;

// Add the spawned room to the pathIndex List

pathIndex.Add(spawnedRooom.transform);

spawnedRooom.name = "Path: " + pathIndex.Count;

}

else

{

pathIndex.Remove(pathIndex[pathIndex.Count - 1]);

roomDetection.GetComponent<RoomType>().RoomDestruction();

int randBottomRoom = Random.Range(1, 4);

if (randBottomRoom == 2)

{

randBottomRoom = 1;

}

GameObject spawnedRooom = Instantiate(rooms[randBottomRoom], transform.position, Quaternion.identity) as GameObject;

// Add the spawned room to the pathIndex List

pathIndex.Add(spawnedRooom.transform);

spawnedRooom.name = "Path: " + pathIndex.Count;

}

}

Figure 13: LevelGeneration.cs – Part 6

using UnityEngine;

public class Entity : MonoBehaviour

{

// Keep track of each entity by giving them a unique id number and name

[Header("ID:")]

[SerializeField] private int id;

[SerializeField] private string entityName;

}

// Move this object to the new location

Vector2 newPos = new Vector2(transform.position.x, transform.position.y - moveAmount);

transform.position = newPos;

// Spawn Room with top opening.

int rand = Random.Range(2, 4);

GameObject spawnedRoom = Instantiate(rooms[rand], transform.position, Quaternion.identity) as GameObject;

// Add the spawned room to the pathIndex List

pathIndex.Add(spawnedRoom.transform);

spawnedRoom.name = "Path: " + pathIndex.Count;

direction = Random.Range(1, 6);

}

else

{

// Stop level generation

stopGeneration = true;

}

}

}

#endregion

}

Figure 14: Entity.cs

Figure 15: Unit.cs

using UnityEngine;

public class Unit : Entity

{

// Add unit statistics to entity

[Header("Unit Statistics:")]

[SerializeField] protected float maxHealth;

[SerializeField] protected float currentHealth;

[SerializeField] protected float moveSpeed;

[SerializeField] protected float jumpForce;

[SerializeField] protected bool isGrounded;

// Collect components that will not change

[Header("Unit Components:")]

[SerializeField] protected Rigidbody2D rb;

// Initialise components and set currentHealth to equal maxHealth before the start method in inherited members

void Awake()

{

rb = GetComponent<Rigidbody2D>();

currentHealth = maxHealth;

}

}

Figure 16: Player.cs – Part 1

using UnityEngine;

public class Player : Unit

{

[Header("Movement Settings:")]

[SerializeField] private float groundCheckRadius;

private Transform groundCheck;

[SerializeField] private LayerMask whatIsGround;

private float jumpTimeCounter;

[SerializeField] private float jumpTime;

[SerializeField] private float fallMultiplier = 2.5f;

[SerializeField] private float lowJumpMultiplier = 2f;

private float moveInput;

private bool isJumping;

void Start() // Before First Frame

{

// Find the ground check game object attached to this object

groundCheck = transform.Find("GroundCheck").transform;

}

void Update()

{

// Get player horizontal movement

moveInput = Input.GetAxisRaw("Horizontal");

// Set the direction the character is facing depending on the value of horizontal movement being more or less than 0

// more than = right

// less than = left

if(moveInput > 0)

{

transform.eulerAngles = new Vector3(0, 0, 0);

}

else if(moveInput < 0)

{

transform.eulerAngles = new Vector3(0, 180, 0);

}

// Set isGrounded to true, if the overlap circle finds another collider with the layermask of Ground

isGrounded = Physics2D.OverlapCircle(groundCheck.position, groundCheckRadius, whatIsGround);

// JUMP when the player presses the jump key/button

if(isGrounded == true && Input.GetButtonDown("Jump"))

{

isJumping = true;

jumpTimeCounter = jumpTime; // reset jumpTimeCounter

rb.velocity = Vector2.up \* jumpForce;

}

Figure 17: Player.cs – Part 2

// while JUMP is being held, the longer, the higher the player will go, to an extent

if (Input.GetButton("Jump") && isJumping == true)

{

if (jumpTimeCounter > 0)

{

rb.velocity = Vector2.up \* jumpForce;

jumpTimeCounter -= Time.deltaTime;

}

else

{

isJumping = false;

}

}

// if the player releases the JUMP button/key early

if (Input.GetButtonUp("Jump"))

{

isJumping = false;

}

}

void FixedUpdate()

{

// Do left and right movement

rb.velocity = new Vector2(moveInput \* moveSpeed, rb.velocity.y);

// if we are falling, apply new physics to character to get a better, less floaty jump

if (rb.velocity.y < 0)

{

rb.velocity += Vector2.up \* Physics2D.gravity.y \* (fallMultiplier - 1) \* Time.fixedDeltaTime;

}

else if(rb.velocity.y > 0 && !Input.GetButton("Jump"))

{

rb.velocity += Vector2.up \* Physics2D.gravity.y \* (lowJumpMultiplier - 1) \* Time.fixedDeltaTime;

}

}

}

# Testing and Verification

## Test Plan

* Functionality Testing
  + Checking the game itself for general problems in gameplay and game mechanics, user interface and game asset integrity.
* Compatibility Testing
  + This relates to ensuring that your game plays with minimal problems on the platform you are targeting.
* Soak Testing
  + This is running the game for prolonged periods of time in various modes of operation – e.g. idling, paused or at the title screen. Soaking can detect memory leaks that manifest only over time.
* Alpha Testing
  + Typically run in-house for code testing and general running of the game. This is made prior to any Beta testing.
* Regression Testing
  + Conducted once a bug is fixed and checks to determine if the bug is still occurring.

## Error Report

* Device Testing

**Windows Computer(s) tested:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Date  dd/mm/yy | OS Version | Device Model | Screen Resolution | Processor | Processor Architecture | RAM  GB | GPU | Result |
| 05/05/20 | Windows 10 Home - 1909 | Home Built Gaming PC | 1920x1080 | Intel i7-6700k  4.0 – 4.2 GHz | Skylake | 16 | NVIDIA GeForce GTX 1080  10 Gbps | Passed | |

Figure 18: Table of Windows Computers used in testing

* Test Results
  + List of Functionalities Tested

|  |  |
| --- | --- |
| Functionality | Result |
| Installation of the game | |
|  |  |
| User Interface | |
| Check all graphic elements, text, and animations display in high resolution | Passed |
| Check the ability to quit the game from the “main” scene | Passed |
| Gameplay | |
| Check that procedural generation works as intended on the first load | Passed |
| Check that player movement works as intended | Passed |
| Check that the camera follows the player as intended | Passed |
| Check that player collision works as intended with all tiles generated | Passed |
| Check that the goal that the player needs to get to, works as intended | Passed |
| Check that when the player goes to the next level that the procedural generation works as intended | Passed |
| Post Processing | |
| Check that post processing is working as intended | Passed |
| Check that all particle effects are working as intended | Passed |

Figure 19: Table of Test Functionalities

* List of Bugs

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bug ID | Summary | Priority | Environment | Description |
| 1 |  |  | Windows 10 Home |  |
|  |  |  |  |  |

Figure 20: Bug-List Table

## Improvement Report

|  |  |
| --- | --- |
| Game Area | Strengths, Weaknesses, and Suggested Improvements |
| Performance on Windows Devices | **Strengths:**  1.  2.  3.  **Weaknesses:**  1.  2.  3.  **Improvements:**  1.  2.  3. |
| User Interface (GUI/UI) | **Strengths:**  1.  2.  3.  **Weaknesses:**  1.  2.  3.  **Improvements:**  1.  2.  3. |
| User Experience | **Strengths:**  1.  2.  3.  **Weaknesses:**  1.  2.  3.  **Improvements:**  1.  2.  3. |
| Game Mechanics | **Strengths:**  1.  2.  3.  **Weaknesses:**  1.  2.  3.  **Improvements:**  1.  2.  3. |
| Gameplay | **Strengths:**  1.  2.  3.  **Weaknesses:**  1.  2.  3.  **Improvements:**  1.  2.  3. |

Figure 21: Improvement Report

# Bibliography

* G. (2016, April 12). How (and Why) Spelunky Makes its Own Levels. Retrieved May 05, 2020, from <https://www.youtube.com/watch?v=Uqk5Zf0tw3o>