Scurry

Technical Design Document

Written by *Team Mischief*

Table of Contents

1.0 Revision History

Version	Date	Description
0.1	23/07	Initial document. Most headings were filled out with information relating to our game. Some specific things that we can't fill out at this point in development have been left blank intentionally.
0.2	29/07	Updated document. Added External Software section, Naming Conventions section, and Mechanics section.
1.0	5/08	Updated document. Added remaining sections. Added onto Mechanics List. Fixed some formatting mistakes.
1.1	3/12	Updated document. Refined formatting and replaced temporary diagrams. Removed unused mechanics "Player Movement (Swimming)" and "Player's Tail", replaced them with descriptions for Climbing and Grabbing.

2.0 Development Environment

2.1 Game Engine

To develop the project at its core, we'll be using the Unity Engine, more specifically it's **2021.3.31f1** version.

2.2 IDE

The Integrated Development Environment that'll be used to produce the project's programming related works will be Microsoft's Visual Studio 2022 application. It'll allow for more versatile development and integration with the Unity Game Engine being used, as well as providing an all-in-one place to manage all scripts without the need of manually crawling through all files in the filesystem being edited.

2.3 Coding Guidelines

To ensure that our code across this project is coherent and explanatory in its functionality just from a glance, we will be writing code following a set of rules. The guidelines are as follows:

- **Comment your code.** This is extremely important as it will eliminate any doubt in what a specific line or function does. Our comments should be similar in structure regardless of what they're for.
- **Provide tooltips.** This is in a similar vein to the point above; however, this is in relation to providing comments and instructions to designers. These comments, which are found when hovering the mouse over variables that are accessible in the Inspector in Unity, should instruct designers on their use clearly and avoid getting technical or needlessly informative.
- **Eliminate repetition.** If you notice that you're reusing a certain function or line of code, move it to its own function and call that instead. This gets rid of unnecessary clutter and neatens the overall system.

- **Avoid "quick and dirty" functions.** Don't create these kinds of functions without the intention of going back to clean them up later. It's bad practice to leave functions like this in the system and can lead to issues, like performance issues and confusion, especially the latter if not commented on properly. Efficiency is key.

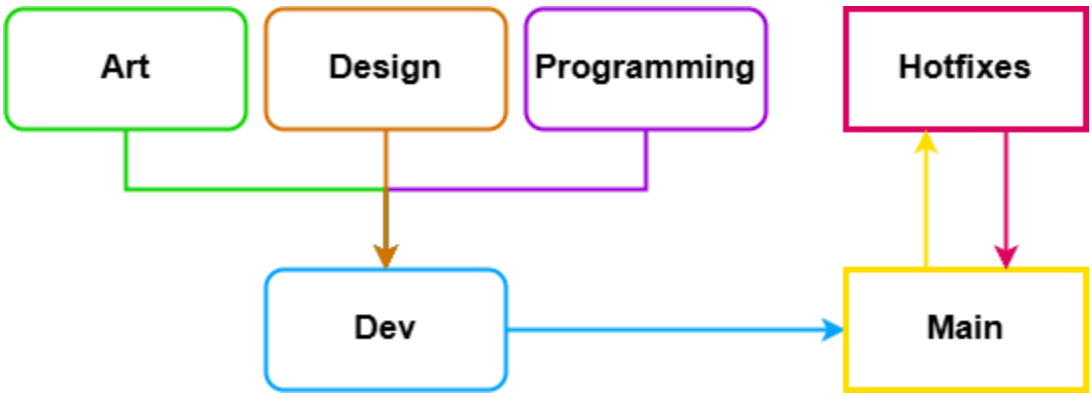
2.4 Naming Conventions

Generally, asset names will be following specific conventions to better convey what they are supposed to be for in the project. This will be done by adding a prefix and an optional suffix to the names of the files, eliminating any confusion on what they are. In terms of scripts created by programmers however, we will be following different conventions. We will be naming our scripts with two or more words, the first of which is the main subject matter while the ones following are more specific to the functionality of the script specifically. For example, a script focusing on axis control of the camera would be called "CameraAxis," or a script focusing on movement for the player would be called "PlayerMovement." Main scripts will be named using pascal case while private scripts will be named using camel case.

2.5 Source Control procedures

Source Control for the project will be utilizing the "Git" system. This paired with "Sourcetree" as the front end application for Source Control means the project can have a basis on multiple branches, as a result it can be contributed to by everyone in the project to get to those final steps of development a lot easier, as well as providing a way for the project to be hosted on the cloud for easier storage access, although the game is backed up at the end of each week on local storage in case GitHub Source Control does in fact, go horribly wrong.

For the time being, Source Control for GitHub is done with a few set branches all acting as stand-ins for certain development stages of the game and types of commits being pushed to and from the origin (cloud), these branches can be represented with a diagram, as shown below.



Having each branch set up means we can restrict access for those who are not directly maintaining Source Control in its entirety and making sure it doesn't break apart restrictions to have them **only** commit, push, and pull in their respective branches. As for the programmers, they have full control over how commits and pushes are distributed across the Source Control solution, as they are responsible for making sure that development goes correctly.

2.6 Memory limits per system

Using C#, Unity has a built-in garbage collector that manages any data that we won't be using anymore, so while this means we're able to not worry about having to clean up garbage data after we're finished using it, we will still be doing things specifically to ensure that we are minimizing our memory usage on a given system. One of the ways we will be doing this is by utilizing Arrays for data storage rather than Lists unless necessary, seeing as the latter has functionality that will go unused.

Unity by default already limits the amount of memory it can use, but our goal is to minimize our usage of memory as much as we can to ensure our game runs smoothly without relying on Unity alone. If we were to rely on Unity to manage our memory for us, we would surely run into issues later down the road.

2.3 Third Party Libraries

"Third Party Libraries" (or in this case, Unity Packages) are being utilized with this project to make development much easier within the Unity Engine, as Packages extend Unity's functionality much like headers to a C/C++ program would in a Custom Engine.

The Unity Packages being utilized for the project are as follows:

- HDRP; a custom rendering pipeline that allows for more control and versatility over Unity's lighting system.
- ProBuilder; a powerful tool that allows artists and designers to edit 3D objects much easier within Unity directly instead of requiring another tool just to get the same results.
- InputSystem; an overhaul of Unity's pre-existing Input System that's built into Unity by default, this newer Input System allows for more control over how binds are registered and recognized internally, as well as providing more control over how said inputs are hooked into your games' programming works.
- Cinemachine; a heavily customizable Camera System that's being used in place of the regular Camera System Unity provides.

2.4 Other Software

To produce the game's core assets, other third-party software is required to be able to both conceptualize and create said assets to be imported within Unity directly after creation. The following software is utilized to produce the end product's art and design elements:

Software Applications

- Adobe Creative Cloud: Used for its wide range of Software for Artists to coble assets together, mostly used for Adobe Photoshop
- Autodesk Maya: Utilized for its Animation and Modelling capabilities.
- Davinci Resolve: Used to create video promotions for the game later in its development, mostly in the form of trailers.
- Audacity: Utilized to manage, modify, add, and remove audio sections to create better edited audio for the game.
- Git: The back end for GitHub Source Control functionality.
- Sourcetree: The front end for GitHub Source Control functionality.

Web Applications

- Microsoft OneDrive: A cloud service by Microsoft that has integration between Microsoft Teams and Microsoft Office which are also being used for this project.
- Microsoft Teams: Communication based software that allows the team working on this project to communicate everything necessary.
- Microsoft Office: A productive suite of software that's being utilized for applications such as Microsoft Word and Microsoft Excel to write documents and collaborate between members with said documents.

- HacknPlan: A Project Manager website used to control the project's timeframes, track how long people have been doing a specific task for (in hours) and finally what tasks need to be done by which specific people all organized under roles and sections during development.

3.0 Game Overview

2.1 Technical Goals

The technical goals for the project are simplified due to the decision to develop the game within the Unity Engine, meaning reaching goals such as getting 3D Graphics to properly work or getting 60FPS isn't exactly all that trivial, what will matter to a bigger extent will be the following three dot points.

- Optimization – C# Is a powerful language used as the "Scripting Language" stand in for Unity, as the core Unity Engine itself runs with C++ Code, it is however, necessary to make sure this C# is as optimized as possible due to its slower nature in comparison to other languages such as regular C or C++. Repetition code should be made public and reused in multiple types of classes and namespaces to free up more space in code design and to not have all methods tangled close together. This and relying on caching techniques to make sure that the game isn't constantly running updates and time-consuming checks / methods aren't causing the game to take a performance hit with everything else needing to be done all at the same time as well.

- Artificial Intelligence (Scripted) – Seeing as all the code is being done within C#, it's necessary to not only ensure Optimization is key, however it is also necessary to ensure said Optimization doesn't get in the way of the Scripted Artificial Intelligence. While these Scripted AIs will be fairly simple in design, they still need to do a variety of activities and "jobs" every so often when they're loaded in a given scene, to have them move around and be able to do said activities / jobs assigned to them, they need to have clean cut code that'll not only show to the reader what the code behind them is doing, however they'll also likely need to be programmed with an Object Oriented Programming design pattern so as to have all Scripted AIs behave similar enough, however have their own

distinctive patterns to make sure they're all unique in some way and are doing different tasks instead of the same one in the same place.

2.2 Game Objects and Logic

The game will include several logical elements that the player will interact with in some way.

- Player Character – The player-controlled asset that they use to move throughout the levels. Can run, jump, and utilise their tail for traversal.
- Floors and Walls – Level geometry made up of static walls and floors.
- Interactable Elements – Can vary depending on what level the player is in and what the circumstances are, but they all require the player to interact with them in some way. This could be something like a wire that needs to be chewed through to remove an obstacle, for example.

2.3 Game Flow

The basic Game Flow at its core will be a linear survival, puzzle based game that starts with a basic Title Screen Menu that allows the player to either Continue where they left off from the last time they played the game, or to Start their adventure from the beginning, both options will be available for those who've at least played once and saved, however the "Continue" option will be greyed out for those who haven't got any save data available on disk.

The game will include upwards of three save files that can be loaded, overwritten, and modified in the Title Screen itself. When the player is playing the game, their game is saved after reaching specific checkpoints placed around each Scene (Level) for the game, that holds true *unless the player chooses to turn off Auto-Saving in the Options Menu!* You cannot choose to toggle the Auto-Saving feature during gameplay, ***the player must select it exclusively from the Title Screen to avoid gameplay issues.***

After starting the game, the player is then tasked to follow a set of Levels in a linear fashion until the end goal is met, where the game ends and they're given the option to pick Levels from a Level Select if they choose to replay the game.

4.0 Mechanics

4.1 Player Movement (Ground)

The player's basic movement is running and jumping. They can travel along a single horizontal axis by pressing and holding left or right. When jumping, the player will always ascend the same height in a smooth arc. Once they reach the peak height of their jump, they will begin descending.

4.2 Player Movement (Climbing)

When the player clings onto a pipe, they can move up and down at a slower speed than on ground. Their movement options are somewhat restricted while climbing as they are no longer able to scurry or climb. Depending on how the pipes are set up, the player will be able to freely jump to and from pipes at any point during the climb.

4.3 Grabbing (Pulling/Pushing)

While on the ground, the player can go to either side of a cube-shaped box and hold the interact key, which will grab the box. While the interaction key is held, the player can pull or push the box to solve platforming or obstacle puzzles.

4.4 Camera Angling and Movement

The camera moves along with the player, always keeping them in view when the player is in control. The camera will have smoothing on it so that its movements and rotations are not sudden. When coming across a big puzzle, the camera will zoom out to encompass the whole puzzle before returning to focus on the player, allowing them to survey the problem before attempting to solve it. The camera will occasionally alter its angle slightly to focus on elements off screen and will always return to the same angle when it no longer needs to focus on the external element. The camera will never rotate to face a different axis.

5.0 Graphics

The Graphics for the game will be rendered under Unity's optional "HDRP" (High-Definition Render Pipeline) Rendering Pipeline, as it will utilize a mixture of gritty and stylized Graphics together, this will result in post processing effects likely being utilized to further the game's visuals, including blurring, depth of field and lastly Unity based Shaders.

These will be in the form of Shader Graphs which will be implemented by Artists and double checked by the Programmers to ensure these effects not only do not cause the project to nose-dive in performance, however, also to create the intended effect instead of it looking "off" from its original intention.

If Shader Graphs are not suitable enough to create dynamic shaders, HLSL and or GLSL Shaders can be utilized to effectively create programmed shaders that'll be a stand in for Shader Graph automatically generated code.

The game will also implement Level of Detail techniques to ensure the game isn't rendering everything at the highest quality despite being further back in the scene and potentially not being fully visible to begin with, this will ensure that the game doesn't need to draw as many polygons and, in turn, vertices on the Graphics Processor Unit (GPU), allowing for more processing room for visuals and effects via Unity Shaders.

6.0 Artificial Intelligence

Artificial Intelligence will be mainly scripted as it does not need to adapt heavily based around the environments and scenery being implemented within the game itself, however, it is possible a State Machine will be implemented to be able to dictate what states said Artificial Intelligence is currently in, as well as how it is managing it's set "processes", or in a designer / visual sense; "Activities / Jobs".

This AI will be simple, as it doesn't need to do all too much from the generic Object-Oriented Programming design being implemented to have all AI have a "base level behaviour", it *will* however, support custom coding callbacks to be able to dictate any differences in behaviour between each AI without needing to compile copy and pasted Scripts with almost identical code minus the smaller alterations. This will likely be done by having each "Custom Code" sections of the differencing AI to be in their own files and actively call Static, AI based functionality that'll dictate the AI's pathfinding, State Machine, and more "Logical" processes.

To further demonstrate how this AI will be programmed, below are some diagrams / flowcharts to properly visualize how simplistic in design this AI will be for the project at hand. This'll showcase how the AI will dictate its current state, their pathfinding and of course, their decision making between states. These will also demonstrate the Code-Design pattern being utilized to pull all the AI based Scripting together to form a coherent and competent Artificial Intelligence system.

7.0 Physics

Physics for the project is kept to a minimum, as they aren't necessarily part of any core functionality for the project, however, Physics is still utilized thanks to Unity Engine's base 3D Physics Framework being utilized.

The Physics in question is mainly done automatically via Unity, as objects that AI can potentially carry with them, as well as the player themselves with them carrying items too, can be dropped on the ground and interacted with via traditional Physics means.

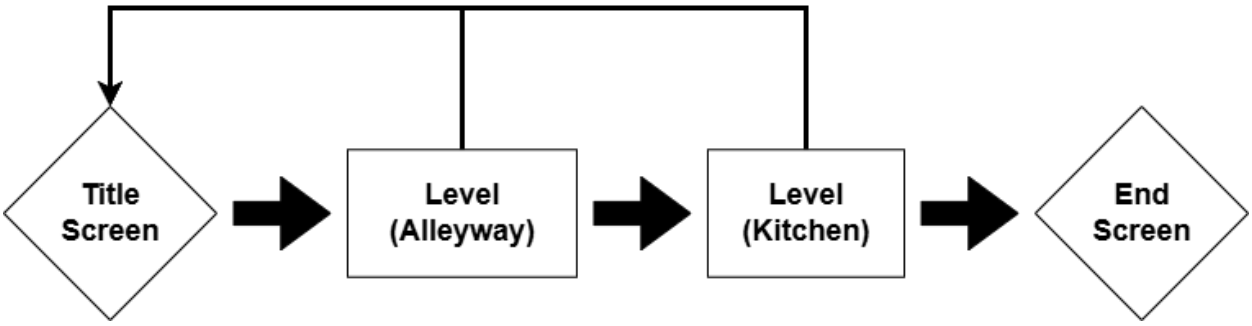
These are done through generic Mesh / Box Colliders, sometimes these colliders have their "Physics" disabled to act as a trigger for the player or select Ais to interact with. Rigidbody Components will also be a re-occurring Physics based tool for the project, as they'll be tied to any objects that are intended to behave under Unity's 3D Physics.

Lastly, the "PlayerController" Component being utilized for the Player Character does have Physics tied to it thanks to it naturally supporting Physics based properties within the Unity Engine. The Velocity, among other properties for the Player are however, programmed manually with Scripts in their "FixedUpdate()" method loop to have complete freedom with Player Control.

8.0 Interactions

Name	Description	Level Appearance
Wire/Cable	The player will need to chew through wires/cables that power obstacles. By cutting off their power, the player can pass them.	Alleyway
Takeaway Box	The player can find this box sitting atop a shelf that they need to knock off. Once done, they can use it to hide themselves from detection.	Alleyway
Grabbable Boxes	The player can grab onto these boxes from either side, allowing them to push or pull them. They are used for platforming and puzzle solving.	Alleyway & Kitchen
Pipes	The player can climb these vertical surfaces to reach higher areas. If the player sits idle on them for too long, they will automatically start sliding downwards.	Alleyway & Kitchen
Mop	The player can utilize it as a means of launching themselves a distance. The strength of the launch depends on how accurate the player's timing is.	Kitchen

9.0 Game Flow



9.1 'Mission' / 'Level' structure

Levels are structured as Point A to Point B, where they will start at one point in the level and need to reach the "end" of it. When starting a level, only the needed level is loaded. Once the player transitions from one level to another, it will be loaded, and the previous level will be unloaded. Any time a level needs to be loaded; the screen will turn black to mask any stuttering that may occur during this period. The only data that is saved between levels is the player's progression, as in what level they're currently up to. The player can pause and return to the Title Screen at any time.

9.2 Objectives

The player's goal is to reach the end of each level, which is done by running, jumping, and avoiding obstacles along the way. The player's progress within a level is evaluated by checkpoints throughout that become active as the player passes or touches them. If they die, they will be returned to their latest checkpoint.

10.0 Levels

Levels within Scurry are mainly linearly structured levels that have you, the player, exploring and traversing through a dingy and urban city. The Levels in question do have Level Specific behaviours to them to *some* level of degree, however these are already managed through Unity's "Scene" System, which, in this case, is being used to split the Title Screen and Levels from each other, as well as a few other areas for development reasons.

11.0 Interface

11.1 Menu

The menus will be entirely comprised of 2D assets. Menus will be interactable through both clicking with a mouse or navigated using a controller. The menus will all be clear and easy to navigate.

11.2 Camera

Being a 2D platformer in a 3D space, the camera will be utilising a perspective view, though the Field of View will be set low enough to allow us to mess with the perspective and pull off visual tricks that wouldn't be possible otherwise. The camera follows the player closely, maintaining its necessity to keep the player on-screen. When reaching points of interest, such as a crow or a particular puzzle, the camera will shift focus away from the player and encompass the point instead until the player has moved out of its view. The camera is very versatile, allowing for cinematic moments and the masking of perspective-cheating techniques.

11.3 Controls

Scurry will be controlled using both keyboard and mouse, Xbox controllers, and PlayStation controllers.

Action	Controls	Details
Movement	- Keyboard (Normal): A/D Keys - Keyboard (Climbing): W/S Keys - Xbox: Left Joystick - PlayStation: Left Joystick	
Scurry	- Keyboard: Left Shift - Xbox: Left Trigger - PlayStation: Left Trigger	- Cannot be performed while climbing.
Jump	- Keyboard: Spacebar - Xbox: A Button - PlayStation: Cross Button	- Player will always jump the same height, regardless of how long the button is held for.
Interact/Grab	- Keyboard: E - Xbox: X Button - PlayStation: Square Button	- Grabbing requires the interact key/button be held.
Push/Pull	- (Same as Movement).	- Only while grabbing.
Pausing	- Keyboard: Escape Key - Xbox: Start Button - PlayStation: Start Button	- Pressing the button again while paused unpauses the game.

12.0 File Formats

18. 11.1 External Formats

As it stands, the project will require multiple file formats to make sure development goes according to plan and is compatible with the Unity Engine, as such, the following External File Formats will be utilized.

- JSON; Used to save user settings in the user's directory under the Unity Player's "Company Name" Folder as "UserSettings.json"!
- CS; The format used for the C# scripts used in our project.
- FBX; 3D models, all the meshes used in our project will be using this format.
- MAT; Material files, used in tandem with texture files.
- OGG; Audio files, compressed to 44100 Hz. Sound effects will consist of one channel, while ambience and potential music will consist of two.
- PNG; Texture files, a maximum of 2048 x 2048.

These file formats being utilized means they can be directly implemented straight into Unity and the Engine will recognize said file formats, making it easier to manage development on a larger scale.

11.2 Internal Formats

Internal File Formats being utilized for the project stem from its Data Handling processes, the game will save to the User's "Persistent Data Directory" which'll usually be in Windows' "%appdata%" folder, or similar variants for other Operating Systems, such as MacOS or Linux.

The game will use a proprietary file format that we're using within Scripts called ".dat," this file format is typically used in industry-level fields as visual, more presentable ways to showcase savable data being utilized for the game they're assigned to. In this case, Scurry will be using said ".dat" internal file formats to represent game progress across three save files in a proprietary Unity binary format to avoid players from easily tampering with the game's save data. If the game detects a save is corrupted or needs to be replaced, the game will not load that data and will choose to bring up a message informing the player that particular save file is now corrupted, most likely due to said user overwriting the file when they've been specifically not to.

13.0 Audio

The audio being utilized for the game will be devised into multiple “Audio Mixers” within a base “Audio Mixer Group” for the sake of the player being able to customize their Audio settings, most notably their “Sound”, “Music” and “Master” Audio Volumes, as any “Sound Source” Component can specify if it’s being filtered through an “Audio Source” or not, this is useful for being able to script said Audio much easier, while giving everyone else on the project an easier time with customizing where specific Sound Files will be internally Audio wise.

14.0 Scripts

As per the Unity Engine's "Scripting Language" model and design it uses to incorporate developer made Scripts into its game logic, Unity Engine utilizes a Programming Language entitled, C#. This Programming Language is used in a "Script" based context, allowing developers to write ".cs" files that will then be applied to "Game Object" Classes which will then run logic as expected. Scripts can also be designed to run under other multiple contexts, which are provided below.

- Data File: Structs, Classes and Enums can be defined as exclusive "Data File" specifications which then can be used publicly by any Script without needing to reconstruct the same Script again.
- Scriptable Object: Unity has a custom "Scriptable Object" type of C# Class that can be utilized to create multiple instances of Data that can be then used by Scripts in multiple, more convenient, and optimized ways.
- Game Object: As described earlier, Unity provides the option for developers to add Scripts onto Game Objects that will then run the code as needed, if that Game Object is "Active," then the Script attached to the Object is ran as expected.

C#'s Specifications can be viewed here on Microsoft's learn.microsoft.com domain:

< C# Specifications >

More Information on how Unity provides support for C# Scripting in the Unity Engine can be found here:

< "Scripting in Unity for experienced programmers" >

15.0 Technical Risks

- Relying on the rat's tail for some functionality will be a bit complex to do and has the potential to break if not properly tested. It might also be a bit too technical for the player to properly utilize if it feels janky.
- Improperly managing resources could lead to performance issues.

16.0 References

- Game Design Document
- Artist Bible
- GitHub Repository
- HacknPlan