Abstract

Explore 3 different tea cultures in a stylistic journey through dream-like tea worlds.

Train to Teasan  
Production Planning   
TECHNICAL DESIGN DOCUMENT

Team Name: Blind Elk

Contents

[About 2](#_Toc53676175)

[Change Log 2](#_Toc53676176)

[Team Members 2](#_Toc53676177)

[Development Environment 3](#_Toc53676178)

[Software Requirements 3](#_Toc53676179)

[Accounts 4](#_Toc53676180)

[Third Party Libraries 5](#_Toc53676181)

[Version Control 6](#_Toc53676182)

[Repository 6](#_Toc53676183)

[Contributors 6](#_Toc53676184)

[Git Workflow & Formatting: 6](#_Toc53676185)

[Target Platform 8](#_Toc53676186)

[Windows / PC 8](#_Toc53676187)

[PC Limitations 8](#_Toc53676188)

[Minimum PC Specs 8](#_Toc53676189)

[Release Build Instructions 8](#_Toc53676190)

[WebGL / Browser 8](#_Toc53676191)

[Browser Limitations 8](#_Toc53676192)

[Minimum Browser Specs 8](#_Toc53676193)

[Release Build Instructions 8](#_Toc53676194)

[Deliverables 9](#_Toc53676195)

[Controls 9](#_Toc53676196)

[Keyboard / Mouse 9](#_Toc53676197)

[Custom Game Systems 9](#_Toc53676198)

[Coding Standards 12](#_Toc53676199)

[Coding Standards Enforcement 12](#_Toc53676200)

[Technical Goals and Challenges 12](#_Toc53676201)

[Technical Goals: 12](#_Toc53676202)

[Technical Risks: 12](#_Toc53676203)

[Risk Avoidance 12](#_Toc53676204)

# About

Describe the project / Game (1 paragraph)

This document details the projects technical information, notably the version control formatting, game system logic, and technical goals and challenges, as well as the software used, deployment platform specs, control schemes, and coding standards. This document will be used throughout development as the specs change or the scope increases.

# Change Log

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Author | Date of change | Description |
| 0.0.0 | AIE | 22/09/2020 | Initial Template created |
| 0.1.0 | Thomas D | 09/10/2020 | Add development environment, player controls, and technical goals and challenges |
| 0.1.1 | Thomas D | 10/10/2020 | Expand upon Game Systems |
| 0.1.2 | Thomas D | 11/10/2020 | Finish Dialogue and Triggers & Interaction game systems |
| 0.1.3 | Thomas D | 12/10/2020 | Finish Scene Transition game system |
| 0.1.4 | Thomas D | 15/10/2020 | Finish Help Message game mechanic, and technical goals and challenges  Add Code Standards, Target Platform |
| 0.2.0 | Thomas D | 16/10/2020 | Finish Version Control Finalize document |
| 0.2.1 | Thomas D | 21/10/2020 | Update git formatting  Add assets |
| 0.2.1 | Thomas D | 22/11/2020 | Update help system game mechanic |

# Team Members

|  |  |
| --- | --- |
| Name | Role |
| Thomas | Programmer |
| Andrew | Designer |
| William | Designer |
| Chloe | Artist |
| Josh | Artist |

# Development Environment

This section outlines the required software and systems required for development of this project.

## Software Requirements

The below table outlines the software requirements for development of this project. Developers contributing to the project are required to use the approved software outlined below.

Any software that contributes to the direct development including planning and communication tools should be outlined below. A developer contributing to the project should have the below software available to them for use.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Software | Version | License | Used By | Used For |
| Unity 3D | 2019.3.6f1 | Education | Programmers, Designers, Artists (On Campus) | Development of Game |
| Unity 3D | 2019.3.6f1 | Free | Programmers, Designers, Artists  (At Home) | Development of Game |
| Visual Studio Enterprise | 2019 | Education | Programmers, Designers  (On Campus) | Writing code & debugging |
| Visual Studio Community | 2019 | Free | Programmers, Designers  (At Home) | Writing code & Debugging |
| Git | 2.19.1.windows.1 | Free | Programmers, Designers, Artists | Version Control |
| GitKraken | 6.1.4 | Education | Programmers, Designers, Artists | Version Control |
| HackNPlan | NA | Free | Programmers, Designers, Artists | Planning |
| Microsoft Teams | 1.0 | Free | Programmers, Designers, Artists | Communication |
| Maya | 2019 | Education | Artists | 3D Modeling |
| Z-Brush | 2020 FL | Education | Artists | 3D Sculpting |
| Substance Painter | 2019.2.3 | Education | Artists | Texturing & Baking |
| Adobe Photoshop | 21.0 | Education | Artists | Texturing |

## Accounts

The below table outlines any accounts that may be needed for the development of the project. An account is usually identified by 2 areas:

* **Individual**: Each developer of the project may need an individual user account, for various software or services. This includes software like Trello, HackNPlan or Git.
* **Organization**: A project or organization account is often developed for the software to integrate with other services, this includes things like Advertising / AdSense / git organizations / repos, Facebook developer account etc. An organization account is usually managed by 1 or more team members. Ownership of the account should be able to change between members.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Account/Service | License | Used By | Used For | Owner |
| Github | Free | Programmer, Art, Design | Contributing to projects hosted on github | NA |
| HackNPlan | Free | Programmer, Art, Design | Planning | NA |
|  |  |  |  |  |
|  |  |  |  |  |

## Third Party Libraries

Unity/Unreal comes with a default collection of plugins, tools and assets. Its plausible, and often encouraged to pull in additional assets, tools, plugins or scripts etc. developed by a 3rd party. Any additional library or assets developed by the third party should be listed below.

|  |  |  |
| --- | --- | --- |
| Asset/Library/Package name | License | Used For |
| xNode | MIT | Creating dialogue |
| ProGrid | Unity Companion Package License | Accurately placing props |
| ProBuilder | Unity Companion Package License | Greyboxing levels |
| [POLYGON - Samurai Pack](https://syntystore.com/products/polygon-samurai-pack) | Synty Store standard EULA | Prototyping levels |
| [POLYGON - Western Pack](https://syntystore.com/collections/frontpage/products/polygon-western-pack) | Synty Store standard EULA | Prototyping levels |
| [UltimateOutline](https://github.com/Shrimpey/UltimateOutline) | MIT | Outlining interactable objects |

# Version Control

## Repository

https://github.com/AIESydneyStudents/BlindWurclsElk

## Contributors

* TheTastyGravy
* Lekonia
* wbeaven
* chloe-cookie
* joshmorrow1185

## Git Workflow & Formatting:

The feature branch workflow will be used with the project, with a branch being created for each task and merged into a development branch after the task is complete, with the branch being deleted. The reason for using this workflow is to keep the repo organized, and reduce the frequency of merge conflicts, as merges are only done at the end of a task instead of after each commit.

Commit messages will be written in present tense (add vs added), and should name the primary things being committed e.g. create props, add room to level, etc. The description will not conform to a format, but should detail what the commit has done, what it's for, how it works, and why it's been done (if applicable).

A branches name will contain the following information:

* **Category**: The category of the task being worked on.
* **Scope:** Refers to the area of the project being worked on. Generally, only one branch should exist for a scope at a given time. As development progresses, the scopes will change, but some scopes that might be used are:
  + Dialogue
  + Level
  + Menu
  + Script
  + Character
* **TaskID:** ID of the task being worked on.
* **TaskName:** The name of the task being worked on, using dashes as spaces.

**Format:**

|  |
| --- |
| Category(Scope)\_TaskID\_TaskName |

**Examples:**

|  |
| --- |
| Programming(Dialogue)\_#1302\_Add-implementation-of-node-graph |
| Bug(Menu)\_#1395\_Fix-resume-button-not-working |
| Design(Level)\_#1129\_Create-first-level |
| Art(Character)\_#1234\_Create-player-model |
| Sound(Script)\_#4321\_Add-voiceover-to-dialogue |

# Target Platform

This project will be deployed to the following platforms:

* Windows / PC
* WebGL / Browser

## Windows / PC

### PC Limitations

PC has full access to all of Unity’s capabilities, so there are no limitations beyond the user's specs.

### Minimum PC Specs

Taken from Unity's 2019.3 system requirements:

* OS: Windows 7/Windows 10
* CPU: x86, x64 architecture with SSE2 instruction set support
* GPU: DX10, DX11, DX12 capable
* Appropriate drivers

### Release Build Instructions

In Unity, select ‘PC, Mac & Linux Standalone’ under platform, and choose ‘Windows’ under ‘Target Platform’. After ensuring all necessary scenes are included and are in the correct ordering, click ‘Build’ and select a folder to build to.

## WebGL / Browser

### Browser Limitations

WebGL has limitations on audio, shaders, locking the cursor, and memory, and no support for threading. The limitations on audio shouldn’t affect this project, and shaders are not certain to be included, so these are largely non-issues. Locking the cursor can only be done after a user-initiated event, requiring it to be handled differently on browser, but this is not a major issue. Threading is unlikely to have any impact on this project. Memory limitations are the biggest factor in building to WebGL, with a reduced size being the main issue. While there shouldn’t be any problems, if there are, lower quality models and textures could be used for the WebGL build.

### Minimum Browser Specs

Taken from Unity's 2019.3 system requirements:

* System spec
* Browser: - WebGL 1.0 or 2.0 capable  
   - HTML 5 standards compliant  
   - 64-bit  
   - WASM capable

### Release Build Instructions

In Unity, select ‘WebGL’ under platform. After ensuring all necessary scenes are included and are in the correct ordering, click ‘Build’ and select a folder to build to.

## Deliverables

A Build of the project should be generated every week and placed in the following location:

|  |
| --- |
| path to network drive folder / <BuildID> / <platform> / \* |

**Build ID**:  
As versions will likely be undefined, using the date is most appropriate. The formatting will be DD\_MM\_YY.

# Controls

## Keyboard / Mouse

Player controls consist of WASD to move, mouse to look, and click on interactable objects to interact.  
Menu navigation will simply consist of UI buttons.

# Custom Game Systems

* **Interaction & Triggers:**  
  Triggers will be used throughout the game as a way of making things happen in the world and will be the most utilized mechanic in the game.

**Triggers** are a broad class of things that can cause things to happen when the player does something, such as enter a trigger collider, collect an item, go through dialogue, etc. These triggers can do just about anything, but a primary use for them is to invoke a UnityEvent with listeners set in the inspector, providing a quick and easy way to set up method calls.

**Interaction** is a unique implementation of a trigger, where a given object becomes highlighted when the player is both in range of it and looking at it, allowing the player to click on it and cause the trigger. The interaction system has two parts: the interactable trigger on the object, and the player interaction script.  
The interactable trigger is an abstract class that serves two purposes. The first is highlighting the object, which is done using a material set in the inspector, in addition to the material the object already has. A public function taking a bool is used to highlight or un-highlight the object, with it changing the material on the object. This function is called by the player interaction script when the object can be interacted with, and again when it can't. The other purpose is an interface for the player interaction script by setting the objects tag and having the abstract function Use().  
The player interaction script is used by the player to both highlight and interact with interactable objects. Its only field is the interaction range, being how far away the player can be from the object and interact with it. The script uses a ray cast from the camera in the direction its facing, with a max distance of the interaction range, checking that the object hit has the interactable tag, to determine if the player is looking at an interactable object, along with a reference to an interactable trigger script. If the ray hits an interactable object and the reference is set to null, it references it and highlights the object. In the other case where the ray doesn’t hit an object and the reference isn't null, it unhighlights the object and removes the reference. When the player clicks and the reference is not null, the trigger is used.

Triggers could be implemented as an abstract base class that others override the functionality of or use composition with separate scripts providing utility to a standard trigger script. The former is simpler to use, but if a trigger needs to be able to do multiple things, it may become messy. The latter will require more setup to use but is much more flexible in implementation. Which of these is used will be dependent on the requirements of the system as development continues.

Triggers can be used for many different things, with some requiring different implementations, but most of them are simply changing the code run when triggered, so it's not necessary to go over them all here. One notable exception is sitting at booths in the train carriage, as this is something of a sub-system. When transitioning between scenes, the player should remain sitting at the booth they are in. To achieve this, the player will exist in a separate scene to the rest of the world, allowing them to persist between scenes during transitions, and have their current state go unchanged, i.e. if the player is sitting before a transition, they will be sitting after too. As long at the train carriage is in the same location in both worlds, this will have no issues.

* **Dialogue:**  
  The dialogue system has two parts: the node graph and the dialogue manager.

**Node graphs** are created using the xNode package, which allows for the easy creation of graphs with custom nodes. A scene graph will need to be used to allow nodes to reference objects in the scene, but consequently they can't be stored in prefabs. There are two major types of nodes: dialogue and triggers, both of which derive from BaseNode.

BaseNode is an abstract node class and contains an input and output port, and an abstract method Activate(). The input and output ports are not used to pass data between nodes in this case, but rather as a method of connecting nodes together in a sequential way. A node can have multiple output connections, but only one input. This allows a dialogue node to have triggers come off it, while preventing possible issues with graph flow. The method Activate() is called by the graph when the node being used, but what it does is determined by node.

Dialogue nodes have three fields: text, audio, and duration. The text is displayed on the screen, audio is optional and is played to the player, and duration is a float for how long the dialogue should be active (if there is audio, its duration is added to duration).

Trigger nodes only have a UnityEvent field, which is invoked in Activate(). This node allows things to be triggered at certain points in dialogue, such as changing a character's animation, or giving the player control back.

The graph effectively abstracts the individual nodes for use by the dialogue manager. First, a function is called to begin the graph, which finds the root node and activates it. When activating a dialogue node, the graph will always pass its string and audio clip to the dialogue manager (singleton), which will display the text and play the audio. Control is then shifted to the current dialogue node, which will wait until its duration is over to return control to the graph. All connected nodes are activated, with the dialogue being passed to the dialogue manager again. If there are no connected dialogue nodes to continue the graph with, the dialogue manager is instead told to end the dialogue.

**The dialogue manager** is a singleton script that ‘runs’ dialogue graphs and displays its text and plays its audio. A trigger would first set the scripts current graph, before calling a method to begin dialogue. This method makes the necessary UI elements visible, then starts running the graph. The graph can call two methods on the dialogue manager; one takes a string and audio clip, and is used to update the current dialogue, simply updating the displayed text and playing audio. The second method is called when the dialogue is finished and hides the UI elements.

* **Scene Transitions:**  
  This system is used to transition from the train carriage scene to one of the ‘dream world’ scenes, and back again.

To hide the loading, the screen will be covered using a three-phase animation with a start and end, and a looping intermediate. The animation controller has a state for each animation, with the first state going straight to the intermediate state after the animation is finished. The intermediate state transitions to the ending state when a bool parameter is set to true (done when the new scene is finished loading), which finishes the animation.

The transition script will be in a separate scene along with the player, so it doesn’t get destroyed during the transition. The player can have its position and rotation set during the transition to position them appropriately in the new scene, as well as have their controller changed to make them return to the train sitting in a booth.  
The transition script will have a public function taking an Enum for which animation to use, a bool for which controller to use, a nullable Vector3 for the position to set the player to (defaults to null), and a nullable Vector3 for the rotation. After starting the animation and disabling the players controller, it will start a private coroutine after a delay equal to the starting animations duration, passing a ref to the controller to enable and 2 Vector3s for the new player position and rotation.

The coroutine will load the new scene and unload the old one using async operations and set the players location. After the new scene is loaded, the animation graphs bool parameter is set to true and the passed controller script is enabled.

* **Help System:**  
  As a method of guiding the player, pathfinding agents with particle effects and trails will be created on the player’s position and path toward a set target, being the next collectable item. The agent itself is very simple, using a navmesh agent component, and a script that sets its target, and destroying the object once it gets close enough.

The help manager is a singleton script holding a public static reference to itself, in addition to a float for time until the help agents start being created, a Vector3 for the target position, and a prefab of the agent itself. Progression triggers can interact with the script through a public function, used to set the timer and target. In Update, the timer is decremented until it reaches 0, when it begins instantiating agents on the player and setting their targets. They are instantiated with a delay between them, done using a second timer.

Progression triggers are a component for triggers that set the help manager’s target and how long to wait before beginning to send out agents. Using a public function, it passes both variables set in the inspector, then disables itself to prevent it from being activated again.

# Coding Standards

Unity has its own [coding standard](https://wiki.unity3d.com/index.php/Csharp_Coding_Guidelines) that will work for this project.

## Coding Standards Enforcement

When opening a script through unity, Visual Studio will use unity coding standards and provide messages when code doesn’t use it. This will suffice if everyone pays attention to Visual Studios messages and warnings.

# Technical Goals and Challenges

## Technical Goals:

* Maintaining 60 FPS on min spec for all platforms.
* Creating a dialogue system that is easy to use and robust.
* Smoothly transitioning from the train carriage to dream land without loading screens.
* Using triggers in the world, with interactions, and in dialogue, to trigger events and animation.

## Technical Risks:

* Shaders and partials are areas I have not explored.
* The potential use of large scenes may lead to performance issues on low-end specs.
* Playing animations during scene transitions may be problematic.

## Risk Avoidance

* The current project scope is achievable within the set time frame, so there is no need to cut certain features/mechanics.
* While using shaders would improve the game, they are not necessary, and can be left out.
* While large scenes with many models should not become an issue, lower quality models could be implemented for lower graphics settings.
* If animations during scene transitions don't work correctly, it could be possible to play a video of the animation instead, or simply fade in and out if that fails.