Process Document

1House1Fish – House Hunting

Version History

Version Number	**Author(s)**	**Notes**	**Review Date**
1.0	James Leung	First Release	17/04
1.1	Kai Siang Kao	Added Link to GitHub Repo	18/04
1.2	Isaac Thomas, Ethan Franks, Kai Siang Kao, Allen Pham, Paul Daniel Mihai	Added strengths and weaknesses	18/04
2.0	James Leung	First draft Trello Process Breakdown, formatting	18/04
3.0	Kai Siang Kao	Added Furniture Creation Process section	19/04
3.1	James Leung	Added Art Checklist	21/04
3.2	James Leung	Added link to Game Design Document	26/04
4.0	James Leung	Added GitHub Commit Process	02/05
5.0	Patrick Siassios	Added Bug Tracking Process	04/05
5.1	James Leung	Minor tweak to Github Commit Process	09/05
5.2	Kai Siang Kao	Updated Furniture Creation Process section	21/05
5.3	Kai Siang Kao	Moved Furniture Creation Process to another file	27/05
6.0	Kai Siang Kao	Remade Programming Conventions to be more specific	27/05
6.1	Kai Siang Kao	Added changes to the programming convention to improve clarity	30/05

About this document

This document outlines the processes the 1House1Fish team will follow during the development of House Hunting.

Important Links

Trello	https://trello.com/b/B2CNfJQD/house-hunting-prototypes
GitHub Repository	https://github.com/KS-103819380/HouseHunting (3D Prototype)
Google Drive	https://drive.google.com/drive/folders/1-_C_zVi0lfeIEEOWwJcYkiXA1wMBEldc
Bug Report Form	*https://forms.gle/TKPKPRGsxQHHcpUh8*
Game Design Document	*Design Document - 1H1F*

Table of Contents

Team

Design Team

Member	**Strengths**	**Weaknesses**
Isaac Thomas	Narrative Design, production, level design	Programming, art
Ethan Franks	Production, level/environmental design, basic art, post-processing, previous experience in developing for and exhibiting at PAX (Unreal Engine 4)	Programming, limited understanding of unity
Jacob Waltzer	Mechanic Design, Programming, Project Management	Art

Art Team

Member	**Strengths**	**Weaknesses**
Natasha Wong	2D character design, 2D environment design, 2D animation, basic 3D modelling skills	Programming, 3D animation, rigging,
Zhenbang Cao	2D characters, 2D environment, experience in UI design	Lack of building 3D models into Unity

Programming Team

Member	**Strengths**	**Weaknesses**
Nikita Golev	Design background, familiarity with C#, good with Git, quick learner	Lack of game development experience, new to Unity
Kai Siang Kao	Strong programming background	Relatively new to Unity
James Leung	Strong Unity/C# familiarity, systems design/development	Communication can sometimes be slow
Paul Mihai	Strong understanding of Unity and C#, as well as other commonly known programming languages.	Code can be very experimental, and UI is tricky to work with.
Allen Pham	Used to Unity/C#	Never made a deep 3D project
Patrick Siassios	Used to Unity/C#	Been a while since using unity

Software

	Software	**Notes**
Version Control (Unity Project)	GitHub	A GitHub repository will be used to share project files and act as version control.
Git Client	Git CLI, GitHub Desktop	GitHub Desktop and the Git CLI will be used as Git clients to make changes to the repository. Team members are encouraged to use whichever they are more comfortable with
Time Tracking	Toggl	Toggl will be used to keep track of time worked.
Task Tracking/ Allocation	Trello	Trello will be used for task management, including organising sprints and due dates.
File Sharing	Google Drive	A Google Drive will be used for sharing documents.
Communication	Discord	Discord will be used to facilitate communication between team members.
Game Engine	Unity – [VER]	Unity [VER] will be used as the game engine to develop the game.
Modelling Software	Blender	Blender will be used to create 3D assets for the game.
Bug Tracking	Google Forms	Google Forms will be used to keep track of any bugs in the game

Bug Tracking Process

Team members should be tracking and logging bugs when they are found. Bugs can be found or identified via testing, code review, or user feedback. Once a bug is identified, it needs to be logged. The bug should then be logged in the bug report form.

The form should be filled out with as much relevant information as possible. On submission of the form, a new Trello card will be created.

Any Team member willing to take on a bug can allocate themselves to the bug. See Bug Cards for additional info.

GitHub Commit Process

Each time a new feature is to be added, a new branch should be created for that feature. Team members may also have their own branch for testing if they wish. When the team member is happy with their changes, they can create a pull request in the repository. When a pull request is created, a message will automatically be posted in the needs-testing-review Discord channel, notifying other team members.

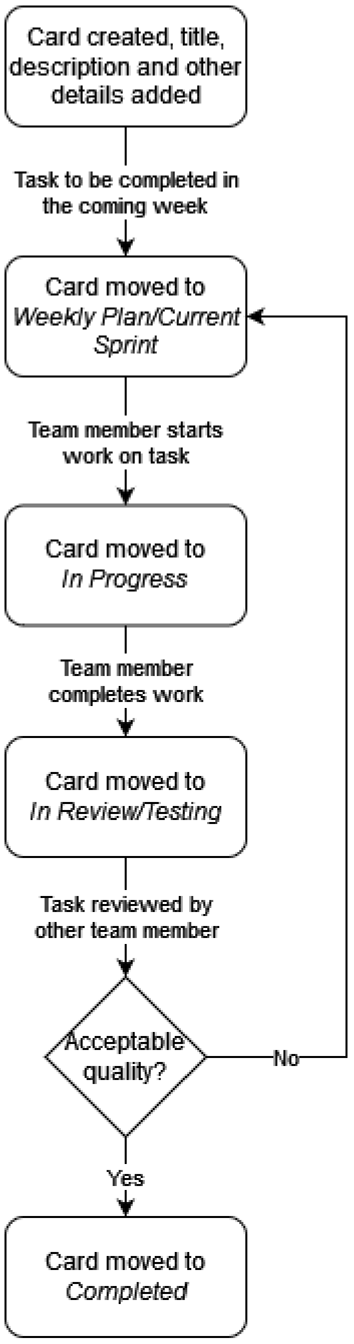
At any point, another member of the team may review the pull request. If the pull request is of appropriate quality, the team member can accept the pull request, leave a comment on if any future considerations need to be made and merge it into the main branch. They should then react to the posted message in Discord with a ✅, indicating a review has been completed. If the pull request is not of appropriate quality, the team member should leave a comment on the pull request as to what is wrong. Once the original team member has fixed the issues, they should re-request a review. The posted message in the Discord should be reacted to with a ❌ until the feature is properly implemented.

Trello Task Tracking and Allocation Process

Task/Card Creation

The team will create cards for tasks that need to be completed. These should be labelled by discipline (Art, Design or Programming), have a clear title and a more detailed description of what the task entails and the result that should be obtained upon completion of the task. Diagrams should be attached if necessary. Tasks should be created in their relevant backlog list.

If the task is reliant on another task being completed first, a comment should be written in the Activity section of the card saying the task is blocked, with a link to the prerequisite task.


Card created, title, description and other details added
Task to be completed in the coming week
Card moved to *Weekly Plan/Current Sprint*
Team member starts work on task
Card moved to *In Progress*
Team member completes work
Card moved to *In Review/Testing*
Task reviewed by other team member
Acceptable quality?
No
Yes
Card moved to *Completed*

Task Allocation

Tasks are allocated during weekly meetings. The member assigned to the task should add themselves to the card. If the task is to be completed in the coming week, the card should be moved to the Weekly Plan/Current Sprint list. When a member begins work on a task, they should move the task to the In Progress list. Only tasks being actively worked on should be in the In Progress list.

Task Testing

For a task to be moved from the *In Review/Testing* list to the *Completed list*, a team member who didn't work on the task must check that the changes made are up to scratch. The team member should post a comment saying they are reviewing the task and then add themselves to the card.

After checking the work done, if the task has been satisfactorily completed, the member should post that the task was completed in the Activity section of the card and move it to the *Completed* list. If the task was not completed satisfactorily, the member should instead post what changes need to be made, tagging the member who did the work, and move the card back to the *In Progress* list.

Bug Cards

Bug cards will automatically be added to the board in the *Bugs* list though the Bug Report Form. Bug cards are not part of the *Weekly Plan/Current Sprint* and any available team member can choose to allocate themselves to fixing the bug, at which point the card should be moved to the *In Progress* list and treated as usual.

If a bug is a duplicate of another bug, a post should be made in the *Activity* section and it should be linked to the original and archived.

A team member can propose a bug should not get fixed in the *Activity* section of the card. If a consensus is reached that the bug will not be fixed, it should be labelled as a *Will Not Fix* bug and moved to the *Completed* list.

File Naming Conventions

Builds

`{Preprod,Alpha,Beta,Gold}_v<VersionID>_<DateID><BuildID>`		
`{Preprod,Alpha,Beta,Gold}`	Build stage	`Preprod, Alpha, Beta, Gold`
`<VersionID>`	Build version	`0.2, 0.7, 1.0`
`<DateID>`	Build date (YYMMDD)	`230513, 230922`
`<BuildID>`	Sub-build ID for if same day incremental builds	`a, b, c, d`

e.g.: `Alpha_v0.7_230430a`, `Beta_v0.2_230620b`

Textures

`Tex[_<Scene>]_<AssetName>[_<Descriptor>].<Extension>`		
`[<Scene>]`	Scene name if applicable	`Kitchen`
`<AssetName>`	Asset name	`Wood, Stone`
`[<Descriptor>]`	Optional description	`Tiling`

e.g.: `Tex_Wood_Tiling.png`, `Tex_World_Foliage.png`

Static Sprites

`Spr[_<Scene>]_<AssetName>[_<Descriptor>].<Extension>`		
`[<Scene>]`	Scene name if applicable	`Kitchen`
`<AssetName>`	Asset name	`Cauldron, Bottle`
`[<Descriptor>]`	Optional description	`Small, Round`

e.g.: `Spr_Kitchen_Bottle_Round.png`, `Spr_ProgressBar_Filled.png`

Spritesheets

`SpS_<Object>_<Action>[_NumericalIndicator].<Extension>`		
`<Object>`	Name of the object	`GenericRig, DeathRig`
`<Action>`	Action of the animation	`Idle, Walk`
`[NumericalIndicator]`	Optional additional indicator	`01, 02, 03`

e.g.: `SpS_Rat_Idle_01.png`, `SpS_Rat_Walk.png`, `SpS_Reticle_Select.png`

Materials (Unity)

`Mat_[Testing_][<Scene>_]<AssetName>[_<Descriptor>]`		
`[Testing]`	Optional name for testing materials	
`[<Scene>]`	Scene name if applicable	`Kitchen`
`<AssetName>`	Asset name	`Tree`, `Rock`
`[<Descriptor>]`	Optional description	`Small`, `Smooth`

e.g.: `Mat_Testing_Highlight`, `Mat_Kitchen_Cabinet_Left`

Models

`<AssetName>[_<Scene>][_<Descriptor>][_<NumericalIndicator>].<Extension>`		
`<AssetName>`	Asset name	`Tree`, `Rock`
`[<Scene>]`	Scene name	`Kitchen`
`[<Descriptor>]`	Optional description	`Small`, `Smooth`
`[<NumericalIndicator>]`	Optional additional indicator	`01`, `02`, `03`

e.g.: `Furniture_Bed.fbx`

NOTE: Mesh should also be named appropriately from within Blender - Naming convention is `[Descriptor_]<Mesh_name>`, e.g. `Furniture_Bed`

Audio

`{DGT,NDT}_{BGM,VOI,SFX}_<AssetName>[_<NumericalIndicator>].<Extension>`		
`{DGT,NDT}`	Diegetic or Nondiegetic	`DGT`, `NDT`
`{BGM,VOI,SFX}`	Type of audio	`BGM`, `VOI`, `SFX`
`<AssetName>`	Asset name	`FireWeapon`, `BackingTrack`
`[<NumericalIndicator>]`	Optional additional indicator	`01`, `02`, `03`

e.g.: `NDT_BGM_BackingTrack_02.wav`

Animation Clips (Unity)

`Ani_<Object>_<Action>[_NumericalIndicator]`		
`<Object>`	Name of the object	`Rat`, `Reticle`
`<Action>`	Action of the animation	`Idle`, `Walk`
`[<NumericalIndicator>]`	Optional additional indicator	`01`, `02`, `03`

e.g.: `Ani_Rat_Idle_01.png`, `Ani_Rat_Walk.png`, `Ani_Reticle_Select.png`

State Machines

`StM_<Object>`		
`<Object>`	Name of the object	`Rat`, `Reticle`

e.g.: `StM_Rat`

Asset formats

3D models	.fbx
Textures	.png – multiple of 4 (e.g. 256x256px), sized appropriately
Sprites	.png – multiple of 4 (e.g. 256x256px), sized appropriately
Sprites (UI)	.png – sized to fill a 1920x1080px canvas, real size
Sprites (UI – Sliced)	.png – multiple of 4 (e.g. 256x256px), sized appropriately
Audio	.wav

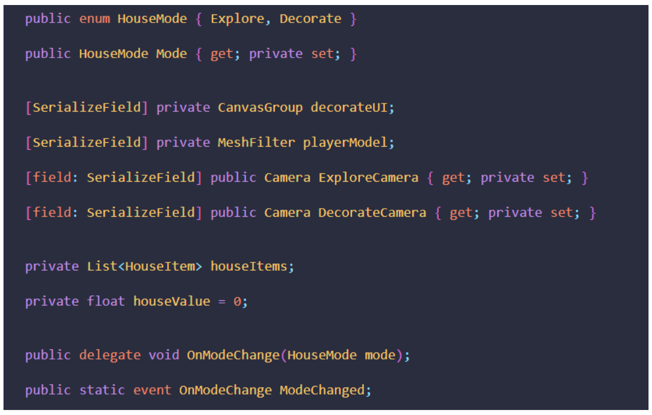
Modelling Export Checklist

- ☐ Is the pivot in the correct place?
- ☐ Is the scale of the model (1, 1, 1)?
- ☐ Is the model in the centre of the world?
- ☐ Is the mesh named correctly in Blender?
- ☐ Is the .fbx file named correctly?
- ☐ Are the file export settings facing the correct direction for Unity? (Z forward, Y up)

Programming Conventions

Classes

- Pascal case for class names
- Arrangement of members/methods (1 means at the top), there should be an empty line between each section:
    1. Enums
    2. `[SerializeField]` private members (`field: SerializeField`) should also be grouped under this, even if it's public or a property. For example:

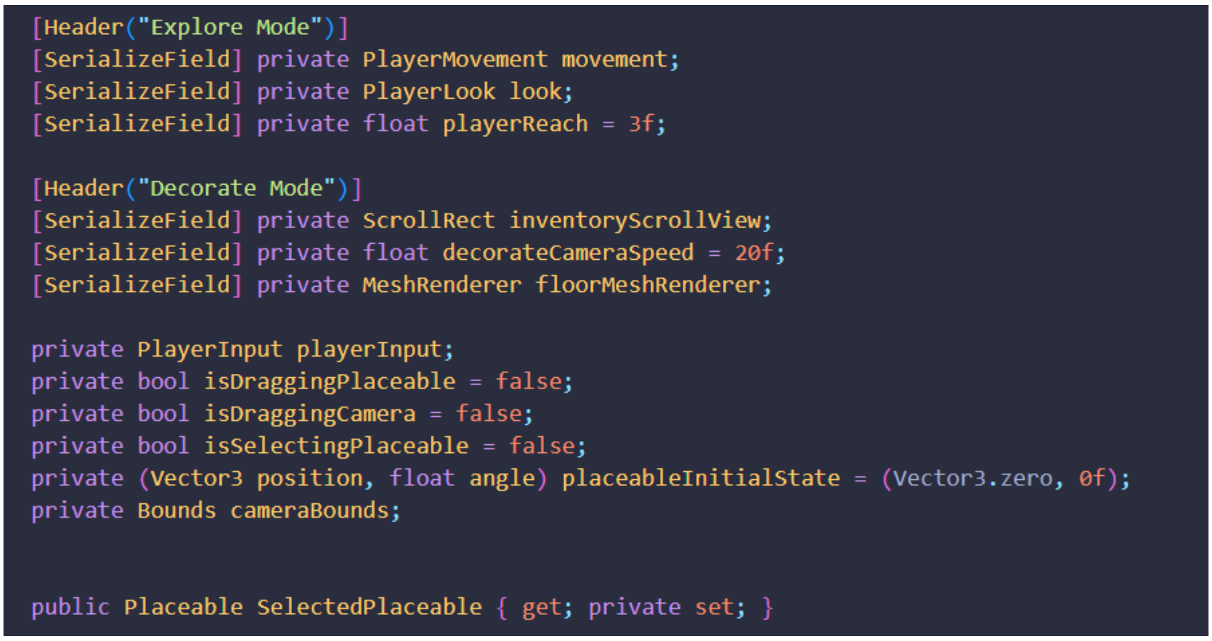


```
public enum HouseMode { Explore, Decorate }

public HouseMode Mode { get; private set; }


[SerializeField] private CanvasGroup decorateUI;

[SerializeField] private MeshFilter playerModel;

[field: SerializeField] public Camera ExploreCamera { get; private set; }

[field: SerializeField] public Camera DecorateCamera { get; private set; }


private List<HouseItem> houseItems;

private float houseValue = 0;


public delegate void OnModeChange(HouseMode mode);

public static event OnModeChange ModeChanged;
```

    3. Private members
    4. Protected members
    5. Public members
    6. Properties

7. Default constructor
8. Parameterized constructors
9. Unity lifecycle methods following the order of execution, e.g.., `Awake()` comes before `Start()`. Details can be found here.
10. Unity specific methods. e.g., `OnCollisionEnter()`.
11. Private methods
12. Protected methods
13. Public methods
14. Coroutines

- `[Header]` should not override the order. Just group them the same way, add a new line between different headers if needed. For example:



```
[Header("Explore Mode")]
[SerializeField] private PlayerMovement movement;
[SerializeField] private PlayerLook look;
[SerializeField] private float playerReach = 3f;

[Header("Decorate Mode")]
[SerializeField] private ScrollRect inventoryScrollView;
[SerializeField] private float decorateCameraSpeed = 20f;
[SerializeField] private MeshRenderer floorMeshRenderer;

private PlayerInput playerInput;
private bool isDraggingPlaceable = false;
private bool isDraggingCamera = false;
private bool isSelectingPlaceable = false;
private (Vector3 position, float angle) placeableInitialState = (Vector3.zero, 0f);
private Bounds cameraBounds;


public Placeable SelectedPlaceable { get; private set; }
```

Variables/members

- Camel case for variable names
- All class members should be private unless it is a simple data container (consider using `struct` instead) or a scriptable object.
- If a member needs to be exposed to the inspector, use `SerializeField`
- If a member needs to be exposed to other classes, use property with the appropriate getter and setters.
- The `SerializeField` decorator should be placed on the same line as the member name.
- If there are other decorators, place them on top of the member (and on the same line if there are multiple).
- If a member needs to be exposed to the inspector, and the name could not convey its purpose clearly, use the `[Tooltip]` decorator to display comments in the inspector field.

Properties

- Use pascal case for properties

- If it only has a getter and is short, use arrow functions instead. E.g., `public int SomeProperty => return member;`
- If the getter and setter are short, make them the same line.

Methods

- Pascal case for methods
- State the access modifiers explicitly. E.g., instead of `void Start()`, use `private void Start()` instead.
- If there are methods overloading, the method with fewer parameters should be listed first.
- Leave an empty line between methods.
- Should be private unless it needs to be exposed.

Code Blocks

- If there are curly braces, the opening brace always goes to a new line.
- If the boolean check for an `if` statement is too long, break it down to a new line (preferably at a logical operator)
- Guard clauses should be on the same line as the condition.
- Remove the braces for one-line `if/else` statements, put the statement in the same line if it's short.
- Leave an empty line between blocks, such as `if-else`, `loops`, `switch`, and etc.

Spacing

- 4-space tabs
- Space between operators. E.g., `int x = y + z` instead of `int x=y+z`

Naming Conventions

- Interface must start with I. E.g., `IInteractable`
- Scriptable objects must end with SO. E.g., `FurnitureSO`

Comments

- Use comments if necessary
- Place comments on top of the line rather than inline.
- Add a space after double slashes. E.g. `// This is a comment`, rather than `//This is a comment`
- Delete the comment that unity generated for `Start()` and `Update()`.
- Use XML comments for complex functions that need long comments

MonoBehaviors

- Use `SerializeField` rather than `FindObjectsOfType<T>()` (or its equivalent) if the game object is known at compile time.
- If it is a singleton, inherit from the `Singleton<T>` base class instead.

Script Folder Organisation

- Do not put any script in the root Scripts folder.
- Organise scripts using the following rules (Rule at the top takes precedence):
    1. If a script is a manager (or contains the word manager), put it in the Managers folder.
    2. If a script is an SO, put it in the Scriptable Objects folder.
    3. If it's an interface, put it in the Interfaces folder.
    4. If nothing above applies, organise them according to their functionalities. If a script is responsible for multiple functionalities, then organise it according to the order of its name.
        i. For example, `WeaponUIManager` goes to the Managers folder (because it contains the word manager). `WeaponUI` goes to the Weapons folder rather than the UI folder (because the word Weapon comes before the word UI).

Other

- Use scene index to load scenes instead of scene names.
- Remove unused `using` statements
- Use new input system rather than the old one (no `Input.GetKeyDown()`)
- When constructing a object, use `new Type()` instead of `new()`