Technical Design Document v0.4

Tetrimination

Contents

[Intro](#_86xq8y7btnke)

[Summary](#_3dm92houafsc)

[Game Overview](#_8y1mt3mkn627)

[Algorithms](#_11a6z4jeyyqp)

[Flood Fill](#_40sxl5yay633)

[Evaluation](#_j2zo13aek15q)

[Development stack](#_yl9lh9esdy4s)

[Target Platform](#_hxbouueprai)

[Target Audience](#_ag29zbaz3cy0)

[Target Context](#_hxbouueprai)

[Technical Guidelines & Conventions](#_wh7oo89q1ixu)

[Folder structure](#_l85hno1j6f8i)

[Naming Conventions](#_rrvcfhictw9a)

[Materials](#_d835vvf7gb98)

[Models](#_j8u0rp2wuikz)

[Prefabs](#_i8xjvrnxwop3)

[Scenes](#_z4knpy2u8y7i)

[Scripts](#_rekjepeg1kiw)

[Textures](#_v3qz9r7fxkpt)

[Coding Conventions](#_ta777vr2uj)

[Language Guidelines](#_tsfstx4yishj)

[Naming Conventions](#_ecg5lyl526tl)

[Layout Conventions](#_sj3djb5rzpnx)

[Commenting Conventions](#_1ctiqb6y475n)

[Importing models](#_kg8ose3d9igw)

# Intro

This tech document serves the purpose of being a full compilation of all the conventions and technical guidelines we use while building our project; these will mainly be naming conventions and coding conventions. We will also explain all specific tools we use. Examples of these are algorithms and 3DS Max. We try to be as clear as possible by providing good and bad examples, so that no questions should arise. Examples may be hyperlinks or images, so we can use a wide variety of tools to convey all needed info.

# Summary

## Game Overview

The game is about 2 types of characters: climbers and one big overlord. At the start of a round one of the players is picked as the overlord. The overlord drops blocks down into a tetris like field. There is a delay between each drop, and the next block gets randomly assigned. Blocks are automatically aligned into a grid, as to keep the tetris feel in the game. Climbers run and jump around in the field, trying to survive the shower of blocks. The overlord is trying to kill all the climbers by crushing or imprisoning them with blocks, while the climbers are trying to reach the overlord by using the tools available (pushing a block around and placing a ladder to bridge a vertical gap). Each climber has a ladder, which needs to be picked up again to use another time.

To determine a winner, we use a score system.

# Algorithms

## Flood Fill

We use the flood fill algorithm to determine which positions in the grid are filled or cut off from the top row. This is useful, because we don’t really have a way to determine which positions in the grid are cut off while not being filled by a block.

Flood fill works by starting at a specific position in the grid, after which the positions of the neighbours of this position are put in a list that still needs to be visited. If a neighbour, however, contains a block, it is not put in that list. This process loops until there are no more blocks that have to be visited. All the blocks that were visited are stored in a list.

You can then take all the positions and subtract the found positions to get the positions which are filled with a block or cut off.

The [pseudocode can be found here](https://en.wikipedia.org/wiki/Flood_fill#Pseudocode):

# Evaluation

## Development stack

Engine: Unity   
Unity is used because it’s one of the best engines for Indie Game Development.  
It’s free, easy to use and there’s an asset store if we need specific models and mechanics for our prototype.

Visual Studio 2020: as IDE  
One of the most used and best free IDE’s out there. It has an entire community around it and the intellisense is great for scoping out bugs.

Programming Language: C#  
A great object oriented programming language.   
  
Source control: Github  
Github is used because we need to keep track of more than just our Unity project, we also have models we want to keep track of. That’s why we didn’t use the collaboration mode in Unity itself.

Modeling: 3DS Max

Most of our team members are pretty confident with 3DS Max, because that is the program we learned last year. So it is self-evident that

## Target Platform

From the description of the assignment, we are forced to build a project suitable to play with Xbox One controllers. For testing purposes, we also made it compatible with keyboard, but that is not the intended way to play.

## 

## Target Audience

This game is suitable for all, but our main target is people with fast reactions and spatial vision. It can be hard sometimes to find out what the best next move is and where blocks are falling.

## Target Context

This game is perfect to play with friends on the couch or with family. It is a party game after all.

# Technical Guidelines & Conventions

## Folder structure

|  |  |
| --- | --- |
| **Assets** | Default Root Folder of Unity. Must not contain any files, only folders. |
| Assets/**Animations** | Folder containing all the animations and animators. |
| Assets**/Audio** | Folder containing the audio clips |
| /Audio**/Music** | Subfolder with all the background music |
| /Audio**/SoundEffects** | Subfolder with all the sound effects |
| Assets**/Editor** | Miscellaneous Folder that contains a.o. ScriptableObjects, Color Preset Libraries, README files. Can contain subfolders if the pool of assets becomes too large. |
| /Editor**/ScriptableObjects** | Optional folder that contains all ScriptableObjects. |
| /Editor**/ColorPresetLibraries** | Optional folder that contains all Color Preset Libraries. |
| /Editor**/Readme** | Optional folder that contains all README files. |
| /Editor**/…\*** | Any other optional folders. |
| Assets/**Fonts** | Contains fonts used in the UI |
| Assets**/Input** | Folder that stores any asset linked to Unitys Input System. Note: self-made scripts that interact with the Input System must be placed in the ‘Scripts’ folder. |
| Assets**/Materials** | Folder that contains all materials used in the game. |
| Assets**/Models** | Contains imported models in .fbx format. Can have optional subfolders. |
| /Models**/[PlayerName]** | Contains the different models of the Player character. Optional folder. |
| /Models**/Environment** | Contains different environment pieces. Optional folder. |
| /Models**/…\*** | Any other optional folders. |
| Assets/**Polybrush Data** | Auto-generated folder. |
| Assets**/Prefabs** | Folder for all prefabs used in the game. At least one subfolder must be present. A distinction between regular prefabs and persistent[[1]](#footnote-0) ones is preferred. |
| Assets**/Presets** | Auto-generated folder by Unity containing Audio and Texture Importers.  WARNING: Do not modify this folder nor its content unless you know what you are doing. |
| /Presets**/Defaults** | Auto-generated folder by Unity containing the default Audio Importer, Texture Importer and Directional Light. |
| Assets**/Scenes** | Folder containing all scenes, both those used for prototyping (that will not be built) and those in the final build. |
| Assets**/Scripts** | Folder for all scripts. Subfolders are named after their respective assemblies. |
| /Scripts**/GameSystem** | Folder named after the assembly ‘GameSystem’ inside. Logically, scripts inside must have a namespace starting with ‘GameSystem’. |
| /GameSystem**/Characters** | Folder with scripts related to controlling characters. Logically, scripts inside must have a namespace starting with ‘GameSystem.Characters’. |
| /GameSystem**/Management** | Folder with scripts related to controlling the game loop. Logically, scripts inside must have a namespace starting with ‘GameSystem.Management’. |
| /GameSystem**/…\*** | Other (optional) folders containing specialized scripts. Logically, scripts inside must have a namespace starting with ‘GameSystem. …’ \*. |
| /Scripts**/Utils** | Folder named after the assembly ‘Utils’ inside. Scripts inside are a.o. the SingletonBehaviour. Logically, scripts inside must have a namespace starting with ‘Utils’. |
| /Scripts**/…\*** | Other folders named after the assembly inside. Logically, scripts inside must have a namespace starting with ‘...’ \*. |
| Assets**/Settings** | Auto-generated folder by Unity URP.  WARNING: Do not modify this folder nor its content unless you know what you are doing. |
| Assets/**Shaders** | Contains shaders. |
| Assets**/TextMesh Pro** | Auto-generated folder by the TextMeshPro package.  WARNING: Do not modify this folder nor its content unless you know what you are doing. |
| /TextMesh Pro**/…\*** | All sorts of assets are stored in auto-generated folders. |
| Assets**/Textures** | The folder that stores all sorts of textures, from regular images to sprites and render textures. |
| /Textures**/RenderTextures** | A subfolder containing Render Textures. |
| /Textures**/UI** | This folder holds all sprites and images used for the UI. |

If you feel certain materials, physic materials, scripts, etc. belong together**\*\*** you are allowed to make a new folder *with a descriptive name* to put them in as long as they are named and placed logically. A new folder called “CharacterMaterials” for example is allowed to be made in the “Materials” folder but shouldn’t be found in the “Scripts” folder.

***\**** *A descriptive name is also required here. The ellipsis is used as a placeholder for the name.*

***\*\**** *Only assets of the same type are allowed to be combined in a folder; don’t combine multiple types of assets.*

## Naming Conventions

### **Materials**

Material names must be prefixed by “MAT\_”. Then, the name should start with a Capital letter and compound names are PascalCased.

Examples:



If there are multiple materials that are similar to each other, the name must clarify what their difference is, e.g. if material 1 has texture ‘Bricks’ and a white base color and material 2 has texture ‘Bricks’ and a blue base color, the following naming is used:

|  |  |  |
| --- | --- | --- |
| Material 1 | MAT\_BricksWhite |  |
| Material 2 | MAT\_BricksBlue |  |

**Parent Folder: “Assets/Materials”**

### **Models**

Models do not use any prefixes. Their name must be something logical though. Compound Names preferably in PascalCase.

Wrong:



Correct:



Models are what we refer to as the pure 3d model, once a model has modifications applied to it in Unity, it counts as a prefab.

**Parent Folder: “Assets/Models”**

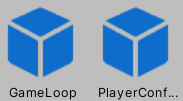
### **Prefabs**

Prefabs do not use any prefixes. Their name must be something logical though. Compound Names preferably in PascalCase.

Wrong:



Correct:



It’s also important that prefabs are more than just models. They have to have something like a collider, script or other feature that makes them more than a model.  
**Parent Folder: “Assets/Prefabs”**

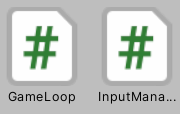
### **Scenes**

Scenes do not use any prefixes. Their name must be something logical though. Compound Names preferably in PascalCase.

**Parent Folder: “Assets/Scenes”**

### **Scripts**

Scripts do not use any prefix whatsoever, as this would require the classes to use this in their name, which is not what we want.



**Parent Folder: “Assets/Scripts”**

### **Textures**

Textures do not use any prefixes. Their name must be something logical though. Compound Names preferably in PascalCase.

**Parent Folder: “Assets/Textures”**

## Coding Conventions

When coding in a team, it happens often that developers follow different conventions when naming their Data Types. Besides, each coding language has its own set of rules. The conventions that are described below are based on the C# Coding Conventions found in the [Microsoft Documentation](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions). Be aware that some rules may differ.

Some general rules beforehand:

* Use meaningful names;
* Avoid using (uncommon) abbreviations;
* Do not use type indicators in identifier names (I only do this in the section ‘Naming Conventions’ as a quick way to name my identifiers);
* Avoid using System Data Types (Int32, String, Boolean etc.) over the Predefined Data Types (int, string, bool etc.);
* Always declare variables locally and as close as possible to their use;
* Prefer to pass variables through as method parameters, unless impossible;
* Always declare properties and methods as private as much as possible to ensure data hiding;

### Language Guidelines

As we use Unity as our Game Engine, the choice for C# as our programming language was quite simple.

Since C# 3.0 a developer has the option to rely on the compiler to use the correct data type by (ab)using the ‘var’ keyword. This is called implicit typing.

However, please use implicit typing as little as possible. Even when the type of the variable is obvious from the right side of the assignment.

[example]

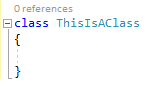
Do not rely on the variable name to specify the type of the variable. It might not be correct.

[example]

### Naming Conventions

* Class

Use PascalCasing for naming classes.



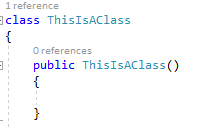
* Constant

Don’t use SCREAM\_CASE for constants. This distracts too much from the rest of the code. Instead, use PascalCasing when naming constants.



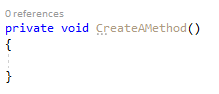
* Constructor

Use PascalCasing. The name of a constructor is always the same name as the class name.

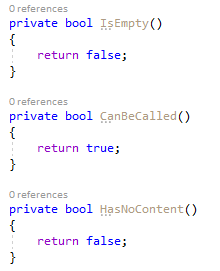


* Method

Use PascalCasing to name methods. A method name must contain a verb and describe what the method does.



If the method return type is a Boolean, start its name with ‘Is’, ‘Can’ or ‘Has’.



* Local (variable)

Local variables follow camelCase naming.



* Method Argument

Method arguments are essentially local variables, so they are camelCased. Same rules apply.



* Field

The name of a field starts with an underscore (\_) where after it follows the camelCase naming.



* Property

Properties are PascalCased.



* Delegate

A delegate is a surrogate of a method, so use PascalCasing. Same rules apply.



* Enum

Write enum names PascalCased.



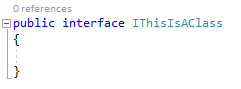
* Generic Type

Generic Type names always start with the letter ‘T’ followed by the name of an identifier[[2]](#footnote-1). PascalCased.



* Interface

Interface names always start with the letter ‘I’ (capital ‘i’) followed by the name of an identifier2. PascalCased.



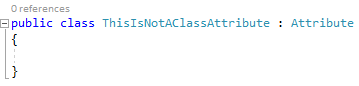
* Event

An event name uses PascalCasing and should either be a verb that tells what it does, or append -ing/-ed when it is triggered after an action occurred.



* Attribute

Attribute names are also PascalCased and end with the ‘Attribute’ suffix.



### Layout Conventions

* Every class must be wrapped inside a namespace (corresponding to the folder structure in Unity);
* List all fields at the top of the class; serialized fields are separated (with a blank line) from ordinary fields and placed higher. The attribute should be on the same line as the field;
* All serialized fields must have the ToolTipAttribute with a description and a default value. The Attribute is placed right above the field;
* Add headers in between serialized fields to create overview in the Editor.
* Public fields must have the HideInInspectorAttribute when inheriting from MonoBehaviour if the field should not be manipulated in the Editor;
* Non-serialized fields are ordered as follows:  
  ‘Components’ section -> alphabetically on Data type -> alphabetically on name; ‘Structs’ section -> ‘complexity’ of the struct -> alphabetically on name;
* Wrap all fields in a region named “Fields” when there are more than 10;
* Properties are placed underneath the fields and separated from them with a blank line;
* Wrap all properties in a region named “Properties” when there are more than 10.
* Methods from the Unity lifecycle (Awake(), OnEnable(), Start(), Update() ...) are separated from the other methods and placed underneath properties, all separated by a blank line. Wrap those methods in a region named “Unity Lifecycle” and order them logically;
* Private methods are placed underneath protected and public methods;
* Input callback methods are placed at the bottom of the class and wrapped in the “Input” region;
* Methods and properties should be alphabetically ordered inside their specific group (see previous points). They are ordered on return type, then on name;
* Prefer using lambda expressions for get and set accessors of properties;
* For readability purposes, always put opening and closing curly braces on separate lines and align them vertically.
* If-statements with an execution body of one line must be enclosed by curly braces, unless the line has the return keyword, then it’s optional;
* Every line of code should be readable without having to scroll horizontally. If a line gets too long, split it up right before the following ‘stops’: , (comma/stop) . (dot/full stop) && (AND operator) || (OR operator)

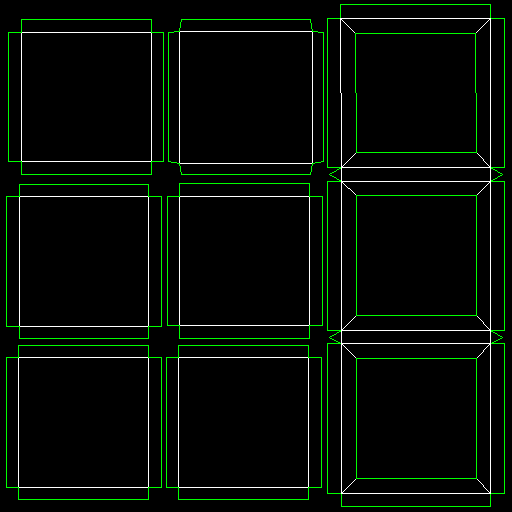
### Commenting Conventions

* Place each comment on a separate line, and above the concerned code block.
* Insert a single space after the comment delimiter (//) before starting the comment text.
* Do not create formatted blocks of asterisks around comments.

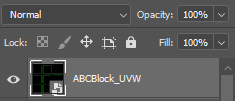
# Importing models

These are the steps we follow to implement a model into our game:

0. **Make a sketch / model sheet:** this step is not always needed but is recommended for models that are somewhat complex, like a character. It will also give you a better starting point to model from.

1. **Make the model:** make sure that it fits in the style we’re going for. Therefore, be sure to export your plain model to Unity several times to test it (see: *5. Export your model as FBX*) . If it doesn’t fit with already existing models then it’s important you take your time to tweak it.
2. **Unwrap and Render UVW:** [Unwrapping](https://www.youtube.com/watch?v=SKrnXWeTNuk) is done with the built-in UV Editor in 3DS Max.When you’re done unwrapping, export your UVW’s in PNG (recommended) or JPG format in one of the following sizes, depending on the details and the size of the object: 128x128, 256x256 , 512x512, 1024x1024.

You’ll get an image like this:

1. **Make the texture:** To texture our models we use Photoshop. Use the UVW map as a guide. Drag it into a photoshop file of the same size. If your UVW image is a JPG, chances are you’re gonna have to change the blendmode from normal to difference:



**→**

To get a real-time preview of your texture, drag and drop the texture (in PSD format) on your prop. Save your PSD files in the folder called “PSD-Files”, feel free to make folders in this folder to keep it clean.

1. **Render the texture:** Once you’re happy with your texture drop it in the “Textures” folder as an exported PNG, *make sure your texture name ends with \_tex*. Don’t forget to make the UVW map invisible (hide or delete the layer in PS) and to swap out the PSD texture on your model with the PNG.
2. **Export your model as FBX:** Center your model by putting it on X:0 , Y:0, Z:0 in 3ds max. After that select it and make sure it’s the only model selected. Then click on “File” in the top left corner , click on “Export" and finally click on “Export Selected...”. Give it a [descriptive name](#_j8u0rp2wuikz).
3. **Getting everything in the right folder:**  
   Your Model(.fbx) goes in the “Models” folder.  
   Your Texture(.png) goes in the “Textures” folder.  
   Make a Material in the “Materials” folder and do this for each of your textures.  
   Put your model in the scene to be able to make it a prefab.  
   Combine your model, with the wanted scripts, components and materials and put it in the “Prefabs” folder.  
   Delete the model you placed in the scene **(!)**.

1. Persistent prefabs contain a script that prevents them from being destroyed on scene load. These will mostly be Singletons and linked to the management of the game loop. [↑](#footnote-ref-0)
2. In programming languages, identifiers are used for identification purposes. Or in other words, identifiers are the user-defined name of the program components. In C#, an identifier can be a class name, method name, variable name or label. Source: GeeksForGeeks.org - C# | Identifiers [↑](#footnote-ref-1)