SD2C/GApp Resources

This section is dedicated to pertinent information and resources that intersect with the SD2C and The GApp Lab Fellowship operations, workflow, and other miscellaneous details

- Setting up Parking @ 102 Tower Instructions
- GApp Fellow/Intern Hiring Process
- APK SideLoading onto the Quest 2 & 3
- Unity Coding Standards
- U of U Affiliates

Setting up Parking @ 102 Tower Instructions

Steps to Obtain a Parking Permit in the 102 Tower building

1.) Fill out the Parking Agreement PDF file (

https://uofu.box.com/s/x66srgnvb80dzdt127rzd9xv9nlr60qi)

- Decide what parking fee you want to pay based off of how much you plan to be in office. You can see their pricing plan here.
- Your Key Card number are the 6 digits right after the + sign on your University ID badge. To see an example, click here.
- If you need to see an example of how the parking agreement should be filled out, click here.

2.) Send it to Mike Leonhardt, SP+ - mleonhardt@spplus.com and mention that you would like to setup parking. He will respond with an Account number.

3.) Goto Parking.com and click on Monthly Accounts.

4.) Enter your account number and then follow the instructions to register and setup an account.

5.) Once you've setup an account, you can sign in and goto the main page where you can click on "Manage Payments" and add a credit card for billing.

6.) Once you've entered everything into the system, it should take 24 hours before you're all ready to go!

If you run into any problems you can reach out to Stacey Earle or Thomas Jennings with questions.

Notes on Using your Keycard:

You'll use your University of Utah ID badge (also your keycard) to swipe the scanner at the parking entrance. When you tap it, it shows you the amount of hours you have left for parking each month. This means that if you have to leave and come back on any given day, it will not subtract a day from your plan but rather track how many hours you have used for that day.

GApp Fellow/Intern Hiring Process

Hiring Process

When hiring a new cohort of students for the GApp, it's important to look at the TBP allocation due date. This past year, the soft deadline (for the DoG - it may differ in other departments) was August 1st while the hard deadline was August 16th. The TBP allocation due date is basically when you want to have selected your GApp fellows meaning it is the deadline of the application. This means you want to create and circulate the Fellowship application at least 2 weeks before the TBP allocation due date. Also a side note on waiting until the hard deadline could cause issues with allocating TBP to international students because of all the additional bureaucratic paperwork that they must endure.

By the beginning of July, you should check with whatever department you want to hire students from and see when their TBP allocation date is. Then once you've created your application, distributed it and accepted Fellows into the program, you will want to have them immediately fill out THIS (DoG example, each department will have their own which you will need to get) and send it to the department's accountant.The linked example lists the chartfield and salary we currently have but you'll want to verify those numbers every time. Once you have done that then you will want to have the students fill out the following New Hire Form and send it to PHS's accountant, along with the chartfield you'll be using, their salary and the funding letter which in our case will be the offer letter. If it's an international student in question, they most likely will not have an SSN so you will have to direct them to this document and have them follow its steps to obtain an SSN while keeping in mind it could take up to 10 days to get before they can actually start work.

Applying for an SSN is a complicated process for international students; be sure to check out the pages from ISSS when applying:

https://isss.utah.edu/current-students/students-f1-and-j1/f-1-students/employment/on-campus-employment/index.php

One of the things that the Social Security office will have international students do when applying for an SSN is confirm their status which they can do through SEVIS. They will need to login and confirm their status here: https://egov.ice.gov/sevis/. If they haven't registered they will need to do

so first. They can also contact ISSS to check their SEVIS status; if there is something wrong with a student's SEVIS, they need to get it fixed ASAP because it could jeopardize their F-1 visa and cause problems. Once their status is confirmed, they will need to have me (you) fill out the the following form for them to submit to ISSS:

https://drive.google.com/file/d/1RhtY5YSIECj78VlxY8q-5dJG9I-aZl9F/view?usp=drive_link

Employment offers are contingent upon completion of a background check and drug screen. So in addition to everything listed above, students will need to visit:

https://www.hr.utah.edu/forms/lib/Certiphi_Background_Check_Info.pdf for more information on this process. They will receive an invitation e-mail from Certiphi Screening, the firm that will perform their background check for the University of Utah. The email will come from applicationstation@certiphi.com. It is important that students read this letter in its entirety. It is also extremely important for them to review the Social Security number they provide. If entered incorrectly, the applicant will need to complete another background check. The additional check may cause a delay of the hire date and will result in additional charges to the department.

FootNote:

We had difficulty obtaining SSN for international students this year; one way to bypass this is to get them a temporary working number from the tax service at the U (https://fbs.admin.utah.edu/tax-services/contact_tax/)

They will require the following in order to issue a TWID:

1. A receipt for the individual's SSN application from the Social Security Administration.
2. Copies of the individual's U.S. visa and International Student & Scholar Services issued work authorization letter.
3. A written acknowledgement by the department that a copy of the employee's social security card will be provided to Payroll Accounting when it becomes available.

APK SideLoading onto the Quest 2 & 3

The most complex part of sideloading APK files onto either the Quest 2 or 3 headsets is all in the setup. Here are the following steps you will need to do so:

1. Create a meta developer account. This part is two-fold; first you create a regular meta account and in order for it to become a developer account, the email you used to create the account will need to be added as a developer to the project/organization of which you're a part on developer.meta.com.
2. You will need to download the Meta Quest Developer Hub app and sign in with the same address you created.
3. You then will need to signin to your Quest device with your meta account; you simply goto profiles > add profile and then you'll be given a code which you'll use to log in. Note: If the headset in question is one that may have been checked out and has an Admin account on it, you will not be able to sideload through your profile but only the admins.
4. You'll need to either join the same wifi that your computer is on or you can setup a Hotspot on your computer which you can then connect to with your headset
5. In the headset, goto Settings > System > Software update.
6. Open the Meta Quest Developer Hub and select Device Manager. In the top right corner, select "Set Up New Device", click next and select your device model (Quest 1, 2, 3 etc..)
7. Then you'll need to select the serial number associated with your headset - if you are in a lab and there are other headsets around, you will see more than one option.
8. Click next and make sure Developer Mode is enabled and make sure you allow access inside the headset. Connect your headset to your computer; make sure you have a proper Quest 2 Computer cable - a regular USB C to USB A cable (like the one you use for your phone) will not work. When you do this, a message should pop up in your headset to allowing access - if it doesn't, reboot the headset and hubs and try again.

At this point you should have successfully connected your headset to your computer and it should be ready to sideload software onto. You do this by:

1. Select the APK file you want to install and drag it over to the drag/drop panel that should now appear because your headset is connected or click the "Add Build" button to search for the APK manually.
2. The cast button allows you to cast what you're seeing in the headset and can be handy
3. If you have additional content that needs to be installed and is not part of the APK, you will need to copy and paste it in File Explorer. First you'll need to go inside your headset and find the notification that is asking you to allow usb access from your computer to your headset; once you do that, you'll see the Quest device appear in File Explorer and you'll

be ready to copy files over

FootNote: we have to do this for the SDoH project but in order for certain folders to appear (com.DefaultCompany.sodh) I first had to run the APK as a shell with no content...

Unity Coding Standards

Introduction

Establishing a coding standard is essential to the long-term health of a project. While there are certain
general rules that are almost universally good to follow, other rules can be a matter of personal
preference.

However, **the most important thing is that all of these rules are followed**, once established.

Over time, the codebase will develop a sort of "second language" that will allow current engineers to
read and understand new code easily, and new engineers to ramp-up and familiarize themselves with
the codebase quickly.

It may seem like some rules put up some annoying hoops for you to jump through just to get a simple
feature working. That's good. These hoops may seem arduous, but they will force you to write good code.

Good code is:

- Easily readable and understandable, especially to other engineers who have not written it
- Flexible, and easy to modify or add to
- Sufficiently optimized for the needs of the project
- Written to expose appropriate properties for non-developers to tinker with in a way that makes
  sense (in Unity)

Good code is NOT:

- Written with a motivation to have as few lines and characters as possible
- Over-optimized to the point where the code becomes over-engineered and difficult to parse for
  non-authors
- Covered in comments explaining all aspects of it in detail. Reading the code should be
  self-explanatory (if written properly) and comments should only be used to explain parts of code
  that necessitate the use of more advanced/confusing practices.

With all of this being said, here are the rules that have been established for Unity projects as of January
2024 (projects started before this date may not fully adhere to them).

Code Formatting

Indentation

Rule: Code should be indented with 4 spaces, not tabs. If you're not a fan of mashing the spacebar

(understandably), you can configure Visual Studio to automatically replace any tabs you have with the
designated number of spaces instead.

Motivation: This is mostly personal preference, however in my experience using spaces tends to make
files and diffs easier to read on Git.

Example:

Good Coding Example: Indentation

Bad Coding Example: Indentation

Spaces

Rule: Each distinct component of a line of code (variable name, expression character, keyword, etc.)
should be separated by a space.

Motivation: Spreading out elements of code with a single space can make it easier for engineers to
parse code without it feeling claustrophobic (this is admittedly a personal preference).

Example:

Good Coding Example: Spacing Bad Coding Example: Spacing

Brackets

Rule: All brackets should be on a newline, and should not be put on the same line as a class name, conditional statement, etc.

Motivation: Placing brackets on newlines makes it easier for developers parse code that is particularly
nest-y. Compartmentalizing code based on its indentation is easier to do when the brackets are essentially used as "markers" for this separation.

Example:

Good Coding Example: Brackets

Bad Coding Example: Brackets

Loops and If Statements

Rule 1: All loops and if statements, even if they execute a single line of code, should be enclosed in
brackets.

Motivation: There is no functional purpose for writing single-line loops or if statements without brackets other than to save a trivial amount of characters and lines. Instead, it makes it more difficult to
compartmentalize code while reading it, and makes it easier to accidentally break if another engineer
tries to modify it (and forgets to add brackets that they assumed would be there).

Example:

Good Coding Example: Loops and Ifs Statements

Bad Coding Example: Else If Statements
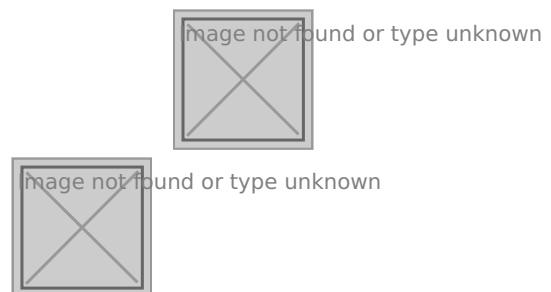
Else If Statements

Rule: Else If and Else statements should be started on a newline, and should not be defined on the same line of a closing bracket.

Motivation: Formatting statements like this makes it easier for engineers to parse and mentally compartmentalize code, which is worth sacrificing a trivial number of extra lines.

Example:

Good Coding Example: Else If Statements

Bad Coding Example: Else If Statements
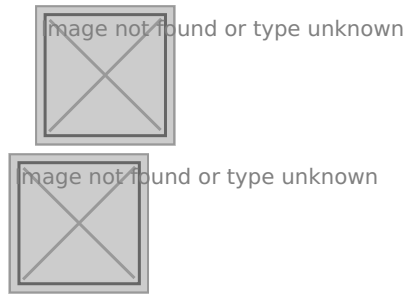
**Compound Conditional Statements**

Rule: If a loop or if statement evaluates more than one condition at once, enclose those conditions with
parentheses.

Motivation: Mostly personal preference. However, enclosing each condition in a compound statement
makes it easier for engineers to determine which values are involved with which statement at a glance,
especially for more complicated compound statements.

Example:

Good Coding Example: Compound Conditional Statements

Bad Coding Example: Compound Conditional Statements
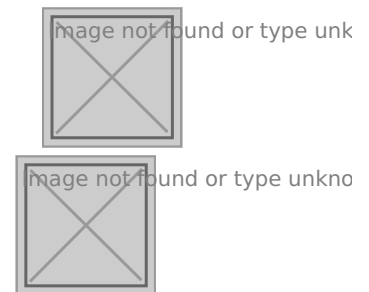
Private Values

Rule: By default, C# designates all variables and functions as "private" unless designated otherwise by a
keyword such as "public" or "protected." However, we will be explicitly designating our values using the
"private" keyword whenever it is applicable.

Motivation: This is once again a matter of personal preference. However, using the "private" keyword
often results in code "lining up better" (particularly with lists of variables) and the color-coding that is

applied to most development environments can make these values easier to find, all of which results in
"easier to read" code.

Example:

Good Coding Example: Private Values

Bad Coding Example: Private Values

Serialized Variables

Rule: When exposing a variable in Unity using the [SerializeField] Attribute, put it on a newline above
the desired variable.

Motivation: This can make lists of variables easier to parse, as it keeps all of the variable declarations at
the same indentation. It also makes it easier to see which variables are exposed in the Unity Inspector at
a glance.

Example:

Good Coding Example: Serialized Variables

Bad Coding Example: Serialized Variables

Code/Naming Conventions

Loops and If Statements

Rule: Loops and conditional statements should be defined as explicitly as possible using two-sided
expressions such as ==, >=, and <=. Avoid using != when possible, as well as evaluating a Boolean or
other values by simply negating it with a !.

Motivation: Standardizing the way loops and conditional statements are defined lowers the risk of other
engineers misinterpreting the condition. It also makes code easier to follow, because the purpose of the
condition is explicitly described, and the data types of the variables involved are easy to identify.

Example:

Good Code/Naming Convention Example: Loops and If Statements

Bad Code/Naming Convention Example: Loops and If Statements

Bad Code/Naming Convention Example: Loops and If Statements
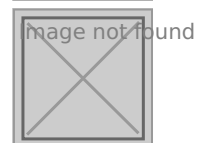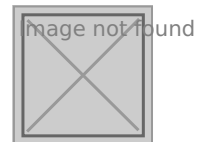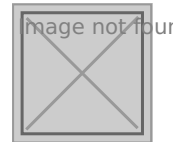
Variables

Ideally, if an engineer is looking over code written by someone else, they should be able to immediately
comprehend what each variable represents, as well as its scope within the context of the code they see
it in. With that in mind, we should strive to adhere to the following conventions when declaring and
referencing variables.

Naming
Variable names should describe as accurately as possible the value it contains, regardless of how long it
is. Ideally, an engineer will be able to understand what the variable is used for by simply reading the
name, and not having to search for references of it to see how it's used in context.

Avoid abbreviating any words in a variable name, especially when it is not clear what word the
abbreviation is meant to replace. The only exception to this rule is for variables that are declared solely
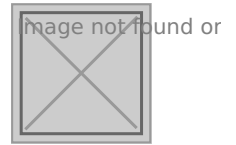to be incrementors in a for loop (ex: **for (int i = 0; i < 10; i++)**)

Code/Naming Convention Examples: Variables

Public Class Variables

Variables declared as public with a class-wide scope should be written in camelCase.

When referencing these variables within their declared class, they should be prefaced using the "this."
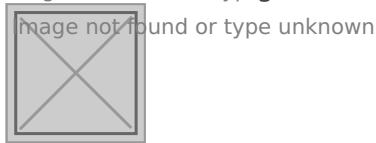identifier to indicate their class-wide scope.

Good Code/Naming Convention Example: Public Class Variables

Protected Class Variables

Variables declared as protected with a class-wide scope should be written in camelCase, then ended with
an underscore.

When referencing these variables, they should be prefaced using the "this." identifier, to indicate their
class-wide scope.

Good Code/Naming Convention Example: Protected Class Variables

Private Class Variables

Variables declared as private with a class-wide scope should be prefaced by an underscore, then written
in camelCase.

When referencing these variables, they should be prefaced using the "this." identifier, to indicate their
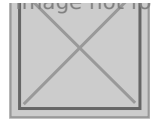class-wide scope.

Good Code/Naming Convention Example: Private Class Variables

Function Parameters

Function parameters should be prefaced by an underscore, then written in camelCase.

Good Code/Naming Convention Example: Function Parameters

Function Variables

Local variables declared within a function should be written in camelCase.

Good Code/Naming Convention Example: Function Variables
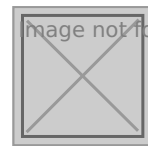

Static Variables

Variables declared as static should have each word in its name capitalized.

When referencing static variables, always preface the variable name with the name of the owning class
(even when you reference the static variable within the owning class).

Good Code/Naming Convention Example: Static Variables

Bad Code/Naming Convention Example: Static Variables

Constant Variables

Constant variables should be written in ALL CAPS, with underscores separating each word in the name.

When referencing constant variables, follow the same pattern as private, public, and static variables with
regards to prefacing identifiers.

Good Code/Naming Convention Example: Constant Variables

Functions

Function names should describe its purpose as accurately as possible. Ideally, an engineer should be

able to know what a function is supposed to do within a block of logic without having to look at its definition. With this in mind, functions should be declared with the following conventions:

- Each word in a function name should be capitalized.
- Avoid using abbreviations in a function name. Long descriptive names are better than short,
  abbreviated names.
- Whenever a function is referenced within the scope it was declared in, it should be prefaced
  with the "this." identifier.
- Whenever possible, function names should start with a verb that describes what the function
  "does" (Ex: Get, Set, Access, Calculate, etc.)
  - If you find it difficult to name a function with a single verb, check to see if the function
    can actually be split into multiple smaller functions.

Classes

Class names should describe its purpose as accurately as possible. Ideally, an engineer should be able to
understand the purpose of a class and have a relatively good idea of the functionality it contains by simply reading its name. With this in mind, classes should be declared using the following conventions:

- Each word in a class name should be capitalized.
- Avoid using abbreviations in a class name. Long descriptive names are better than short, abbreviated names.

Misc. Suggestions

The following are more subjective suggestions rather than hard-and-fast rules. However, you will likely
find that following them will result in code that is cleaner and easier to read and understand at a glance.

Compartmentalize Complex Logic in a Function

If you are writing an algorithm (or part of an algorithm) that is several lines of code long, consider putting all of the specific logic into its own (properly named) function. Even if the function is only ever
called once, it will be extremely helpful for other engineers (or yourself, in the future), as they will not be
unnecessarily forced to parse and understand how a complex process works if they don't have to.

Instead, they can read a descriptive function name, trust that it does what it says, and move onto other

parts of the code that they actually intend to investigate or modify.

Make Use of the [HideInInspector] Attribute

If a public variable is not meant to be set or modified in the Unity inspector, denote it with the
[HideInInspector] attribute before its declaration. This will not only make components in the Unity
editor far less cluttered, but it will also make it explicitly clear to other team members tinkering
with the
project which values are "allowed" to be experimented with without causing any technical issues.

Don't Be Afraid to Copy

When in doubt about how parts of code should be formatted or named, it's best to emulate what
you
already see in the codebase. In the end, it's better to have a consistent coding style across the
project
than it is to have a "correct" style (a futile endeavor to pursue). Alternatively, you can simply ask
another engineer on the team for advice on naming/formatting.

U of U Affiliates

Sometimes it is necessary to create a Unid for a non-University employee or student. The most typical use case we have seen is when we have taken on non-paid interns - usually in High School - who want some more exposure to programming, game design and developing software relating to medical, therapeutic or educational utility. Though there might be use cases where some system we're designing internal to the U requires a Unid login and testers who are not an employee or student will need a Unid to participate. Whatever the reason for requiring a Unid to someone outside to the U, we initiate the process by filling out the following form:

https://www.hr.utah.edu/forms/affiliate.php

You will need the affiliates info such as full name, home address, phone number, email and their SSN. When choosing Affiliate type, you will most likely need to select: 10060 U Affiliate and most likely need to select: 01720 PHS - Health Sys Inno & Rsrch as the authorizing department.

For more information on University of Utah Affiliates, click the link below:

https://regulations.utah.edu/human-resources/procedure/p5-207a.php#a.IV