2022

University of Idaho

Video Game Environment for Arm Assessment Through a Robotic Exoskeleton: Final Design Report

AUTHORS
MIGUEL VILLANUEVA, ALEX PEÑA, DAWSON HILL

UNIVERSITY OF IDAHO | College of Engineering

Table of Contents

Final Design Report

Letter of Transmittal

Dear Dr. Perry and Team,

In this report, we will address background and technical design aspects of our project, “Video Game Environment for Arm Assessment Through a Robotic Exoskeleton,” created for the University of Idaho Assistive Robotics Lab. We will go into detail about the aspects of our design choices and some of the various concepts we considered while building this project.

Thank you in advance for your considerations and attention to this project, and please let us know if you have any questions.

Sincerely,

Miguel Villanueva
Alex Peña
Dawson Hill

Executive Summary

Our team used the Unity game engine to design a modular and intuitive video game environment for arm assessment, which can be used with a robotic exoskeleton, the BLUE SABINO, which stands for Bi-Lateral Upper-limb Exoskeleton for Simultaneous Assessment of Biomechanics and Neuromuscular Output. Our client was Dr. Joel Perry, who represents the University of Idaho Assistive Robotics Lab. The listed requirements that were provided by our client are as follows:

- Consistent, repeatable tasks
- Configurable task settings
- Fully designed point-to-point reach task
- Clear communication of on-screen objects
- A virtual arm that can modularly use multiple joints

The project originated from the need that arose during the development of the BLUE SABINO for a task environment, as simple auditory cues and visual demonstrations were insufficient for providing clear and repeatable tasks for its users. Our team assessed that the Unity game engine would work well as a tool to create this environment.

Our features were implemented in a way that satisfy all these requirements while also providing the foundation and tools necessary to further develop the solution if the Assistive Robotics Lab deems it useful to their overall goals. On top of meeting the specific requirements, we developed additional features, such as an integrated task system that utilizes data objects in Unity; a way of reporting

statistics about the task after it was completed, something that can be built on later to include information about range of motion, control of force, etc.; and a real-time object positioning configurator embedded within the menu system. Using these additional features, we believe that further development on this application can result in an incredibly useful tool to pair with the BLUE SABINO exoskeleton.

Project Background, Planning, and Execution

Project Background

Project Background

Our client was Dr. Joel Perry, who is the lead principal investigator for the University of Idaho Assistive Robotics Lab, which is in the process of building a robotic exoskeleton called the BLUE SABINO (Bi-Lateral Upper-limb Exoskeleton for Simultaneous Assessment of Biomechanics and Neuromuscular Output). The exoskeleton is being developed in a series of phases, from two to thirty joints (fifteen per arm). The BLUE SABINO utilizes high precision motors and runs a simulation on a high-performance target computer called a Speedgoat to capture and measure arm and hand function in individuals that have neurological impairments, such as those who have suffered from a stroke. Together, these integrated systems record joint position information to report on aspects such as range of motion, control of motion, range of force, and control of force. The project began and is being developed with a research and educational focus. It has been determined by Dr. Perry and his team through their work that auditory cues and visual demonstrations are not sufficient for patients using the system. Thus, given these needs and challenges, the team has decided they need a virtual environment that can correctly display virtual objects within that environment. A few different solutions have been developed in the past, but there were issues with shadows in the scene, as well as displaying a proper sense of depth perception. Not only is there a need for consistent and repeatable tasks, but also an environment that has the proper lighting and shadows to accurately display the position of objects within that environment. This is where our team came into play. Our project was specifically focused on creating a virtual environment that can correctly display on-screen items and utilize a task system to help patients use the system. Our team made the determination that, given our requirements, it would be appropriate to develop this task environment using the popular game engine, Unity. The game engine has many built-in tools that will lend themselves useful in tackling these challenges. In addition to this, everyone on our team has varying degrees of familiarity with Unity, which aided in the development process. Creating this virtual task environment would benefit users in allowing for clear, consistent, and repeatable task scenarios, as well as a sense of realism while using the system.

Problem Definition

Our stated problem, as defined by our client, was that auditory directions and visual cues were insufficient in directing a user of the BLUE SABINO system to complete tasks consistently. Using physical objects with the BLUE SABINO was good for a realistic feel but did not meet the requirement of being repeatable and consistent. A graphical environment had been created in the past, but the team decided it did not effectively communicate proper depth perception nor accurate object

positioning due to lackluster shadowing and lighting within the scenes. The goal of our project was to create a software application that would allow for the configuration of consistent, repeatable tasks. We also had the goal of fine-tuning the light sources and shadows within the scene to properly convey depth and position in the three-dimensional environment. To execute on these goals, our team decided to use the Unity game engine. A more in-depth list of the goals of our project is listed below:

- Create intuitive configuration settings for tasks
- Use data-binding objects within the Unity game engine to implement a modular task system
- Utilize realistic light sources
- Fine-tune shadows to allow for proper depth perception
- Use object materials that reflect light in a subtle way
- Develop menus and user interface items with aesthetics in mind
- Create an “assessment result screen” for tasks within the application
- Integrate “repeat” functionality for tasks
- Modify an existing arm model asset for use during development
- Fully develop a point-to-point reach task to demonstrate application capabilities and potential
- Modularly integrate joints for demonstration of concepts

In summary, our overarching goal for this project was to create a software application that has all these features built in. We wanted to build a conceptual foundation that had these goals in mind to eventually turn back over to the Assistive Robotics Lab that would be suitable for further development. Thus, the project was implemented modularly to encourage the addition of future features. Our codebase is our final deliverable for the project and is available on a public GitHub repository.

Project Plan

In the initial stages of this project, we were able to narrow down specific goals and responsibilities for each team member. Because there were three of us, we broke the technical development of the application into three distinct parts and assigned them: Miguel would work on implementing an arm model that would mimic the BLUE SABINO, Alex would develop the task scenes and work with the lighting and shadows, and Dawson would build out a menu system to navigate through the application. Miguel was also elected by the team to represent the team when communicating with our client and external personnel. This helped streamline communication between our team and our partners.

In addition to these responsibilities, our team was under the impression that we would be also creating an interface that would allow communication between our software and the BLUE SABINO’s hardware. This turned out to be inaccurate, however our initial schedule and timeline included time slotted for developing that interface.

At the beginning of the project, we allotted ourselves time to research various Unity development concepts. We also gave ourselves time to set up the project infrastructure, such as the GitHub repository and all of our documentation. By early November, we planned to have a moveable arm within an environment of some kind, as well as keyboard controls for development purposes. Due to some technical challenges that we needed to research and overcome; the arm model went through a

series of iterations that progressively got us closer to our goal of making it mimic the BLUE SABINO. When we were informed by our client that creating the interface between our software and the BLUE SABINO hardware did not fall within the scope of our project, we shifted our focus to making sure the application was developed modularly. This was important, as we were now just building a piece of a larger project and wanted to ensure what we built could be paired with work from other teams..

We also had underestimated the amount of time needed to create fully developed tasks. We had an original goal of creating a repeatable task by the beginning of December 2021, and quickly found that it would take much longer to fully integrate our systems. By December, we had prototype menus and a basic scene with lighting. We also had some basic arms that were a work in progress. We pushed out our original timeline to 2022 and continued work. By January, we had a working version of the arm model and the beginnings of our task system. Work continued on the point-to-point reach task. By the end of February, our menu systems had a basic flow defined, as well as the foundation for our task system. Using the point-to-point reach task, we could demonstrate how the configuration settings would interact with the objects within the scene. Then as EXPO approached, our team transitioned to fully focusing on the Task System and finished the Point-to-Point reach task. A large amount of work went into creating new configuration menus which allowed for a live picture of the actual scene to be interacted with as different options where checked.

- *Discuss intended and actual Schedule (reference Gantt charts in Appendix) ?*

Concepts Considered

During our concept development, the team identified three key areas of this software which we decided were the main pieces of work. These three key features were: the virtual arms, the configuration of tasks, and the task environment.

Virtual Arms

Arm Model

The first consideration for the virtual arms was how we were going to create the actual model for the virtual arms in Unity. As we were starting from scratch, there was no pre-existing arm model for our team. To properly model the BLUE SABINO exoskeleton, the team needed an arm model which could individually control each of the nine joints. Then, to get this arm model into Unity, our team needed to either find an acceptable arm model on the internet and import it, or to design our own model from scratch using modeling software such as Blender. Both paths had merit to them.

The choice to develop our own set of virtual arms using modeling software was the first option. Creating our own set of virtual arms would have allowed the highest level of customization and specialization for this software. In this way, this path would allow for the team to setup exactly the 9 joints needed. Having complete control would also allow the data read in from the BLUE SABINO to be applied to the model easier, then if the arm was pre-selected. The second option was to research and look for online resources so that we could take the arm model another person had created and rigged, and then adjust it for our needs.

Arm Movement System

The other development our team had to consider was the system we would use to move the arms within our environments. Given, the ultimate goal of the software is to interface with the BLUE SABINO and move the arm that way. However, for our purposes of creating the environment and testing, we needed a set of controls to use as developers. To control the arms, our team devised two possible types of controls. One control scheme our team considered was to mimic each individual joint movement/rotation with keyboard. Essentially each joint would be mapped to certain keys to move and/or rotate. Alternatively, another option we considered was controlling each arm through a single point end effector. The end effector would be located on the hand of the arm, and the user would control this one end effector through the keyboard. Think of this as working like how the controls for a drone function.

Configuration

Passing Data Between Scenes

Another key feature which we spent time researching and considering different paths for was the configuration of tasks. Many of the requirements which this project needed to fulfill relied upon some configuration interface as a part of any given task. Since this feature was an integral part of the system, out team spent a good chunk of time researching the most efficient ways to preserve data between scenes. As we searched for the best solution, we also aimed to write this system In a way such that it is highly modular, and highly expandable for future work by the team working on the BLUE SABINO. There were three main paths which we considered for the passing of data between scenes in Unity: singleton, file i/o, ScriptableObjects.

Using a singleton to pass data between scenes was the first idea our team considered. It is extremely easy to do so and takes little setup. The singleton pattern specifies that only one instance of the class can exist at any given time, and this instance can be referred to as the “manager” which has global access to its public members and methods. In Unity, there is a special attribute you can apply to objects in a scene called “DontDestroyOnLoad”. Any object marked with this will not be destroyed on changes between scenes, and in combination with using the singleton pattern, this would allow an object to function as a manager/container to exist between multiple scenes.

Another path considered was saving the configuration data directly to a file in some format (e.g., json). Directly using file storage would be messy, as it creates extra files not directly related to Unity to worry about. However, saving to and reading from files allows for data to be configured in such a way that every object has the same access. No references are needed within Unity, and it removes dependencies that may exist between scenes. Reading in from a file directly also opens up the possibility to edit/create configurations from outside of the software we created, and then import the configurations directly as those type of files.

The last path looked into was using specific containers for data and functions within Unity, called ScriptableObjects. ScriptableObjects have qualities which allow them to have pieces of each of the two paths. Since they are a Unity-defined object, they fit into Unity's system extremely well, and are easy to create from the GUI in Unity. Essentially, you can define a default script that sets up the fields and functions for a ScriptableObject, and then a person can create new objects using the Asset menu in Unity. Each instance that you create is also stored on the file system, and therefore is always accessible to any script that cares about it. And instead of simply just data contained inside these objects, they also can have functions inside them and other Unity-related functionality as well. ScriptableObjects also seem to be looked upon very well by those who are well-versed in Unity programming. A lot of inspiration came from two videos of speakers at the Unity Unite Conference in 2016 and 2017 from the videos "Unite 2016 - Overthrowing the MonoBehaviour Tyranny in a Glorious Scriptable Object Revolution" and "Unite Austin 2017 - Game Architecture with Scriptable Objects". Links to these videos will be in the appendix.

Task Environment

Task Running / Main Environment

The last key feature our team identified was the actual environment in which the task would run. This constitutes setting up various different things depending on the configuration beforehand, running the main loop of the task, and stopping the task once some threshold has been reached. Again, during concept development our team identified three main paths which we could go down to achieve this. These paths were: using gameObject.find(), dragging objects into the inspector, and an event system w/ ScriptableObjects.

In Unity, you can search for other objects within a scene from a current object, by calling the gameObject.find() function, and searching for a specific object or tag of an object which you want to find. On successful finding, the function will return a reference to the targeted gameObject. This allows for a search for any object in the scene and removes the necessity to drag-in references with the inspector for each object we want to manipulate. However, another path we identified was doing just that. Dragging each object we wish to modify into the inspector, and then accessing it within the script.

Finally, the last path we considered was the most complicated one as well, which is using an EventSystem with ScriptableObjects. In Unity, using Events is a type of implementation of the Observer pattern. We can create events in Unity using ScriptableObjects, and then have scripts which are "listeners". For any action which needs to take place inside the task environment, let us say a sphere changing color, we can create an event for this. Then, any objects which care about this, can add this event to their "listeners", and will get called when this event is sent out into the scene. Any object "listening" for this event can then do something in response. This system is highly modular and allows for minimal decencies to exist between different objects. Again, this system is considered a good practice by those involved with developing in Unity, as highlighted by Ryan Hipple in the talk "Unite Austin 2017 - Game Architecture with Scriptable Objects" (link in Appendix).

Concept Selection

Virtual Arms

Arm Model

For the virtual arm model, out team decided to import a model found on the Unity Asset Store, as opposed to designing our own model. One of team members was actually excited to attempt to create an arm model and perform the rigging of the arm in blender by themselves. However, as time progressed and our research continued, we realized how giant of a task that creating our own arm model would be. More time was spent to finding an arm model that was suitable for the task environment, as well as had the rigging structure to allow modular movement of each joint. Eventually, our team settled on a robotic arm model which had a base humanoid rigging.

Arm Movement System

The concept for the arm movement system we selected involved both of the proposed paths. Firstly, the controls to mimic the individual joint movements of each arm were important to validate the customer's requirements, as well as for testing our task environment. We needed to have exact control of each joint, so that we could mimic the data coming in from the exoskeleton. But, these controls alone were a little tedious to operate when merely showing off the task environment. Therefore, the "drone" controls we also mentioned were implemented in addition to the induvial joint controls. With these controls, one could simply move the whole arm with presses on the keyboard, and the rest of the joints and the arm would stay connected/animated correctly through the pre-applied inverse kinematics on the arm.

Configuration

Passing Data Between Scenes

After much research, our team decided to choose the path of using ScriptableObjects to pass data between scenes. While using the Singleton pattern would have been an easy solution and allowed the team to begin working on quicker, this was not the best approach. In terms of encapsulation, putting all of the configuration data for different tasks inside one Singleton object would get extremely messy quickly. And creating multiple singletons for each task would also defeat the purpose of using singletons, as the name implies. Using ScriptableObjects we can configure different settings for each different task and have them be accessible globally as long as we know what the ScriptableObject is called. This allows us to write to the ScriptableObject in our task configuration stage, and then read from it during the task environment phase.

Task Environment

Task Running / Main Environment

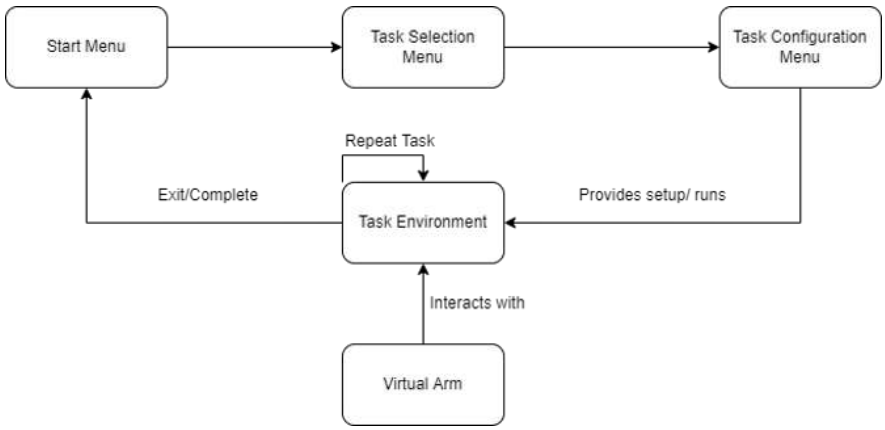
For the task environment, out team also selected to work with the event system and scriptable objects as was mentioned previously. The option of using gameObject.find() is the easiest solution to make calls to different objects from inside the main script of the task, however it is not very efficient, and it can lead to NULL reference errors. The other option was selecting

each object which we want to modify from the main task script and placing a reference to it in the inspector. While this would cause less possible NULL exceptions, the issue becomes for every new object we wish to have been affected by the task, the script has to be updated. Likewise, dragging different objects into the inspector for the script takes away from the modularity as well, as each of these objects must exist to run. However, with the event system, we can simply reference the ScriptableObject events we need to call during a task. Then at certain times we can trigger these events, and any objects which care about this event, will already be "listening" for it. This is highly modular, as we can simply drag in an object which should respond to the event into the scene, and then trigger the corresponding event. Doing this would allow us to test the desired responses for that event, without needing to drag in the entire task to test it.

System Architecture

During conceptual development, our team designed a high-level flow for our software, which we stayed true to throughout the further stages of the project. When first launching the software, the user will be brought to a main menu screen. Currently, this screen just serves as a staging point for the application, and also gives the user a way to exit the game. The user can then navigate into the task selection menu. All current tasks our team has implemented are available from this menu, as well as any future tasks developed would be selectable from here as well. Once the user selects a task, then, the configuration screen for that task will be switched to.

At the configuration screen, the user can then interact with the multiple various configuration options for that task. For example, the Point-to-Point reach task has a handful of options, from moving the positions, adding a third position to move to, and changing the number of repetitions required for the task. Finally, once the configuration is satisfactory, the user can then begin the task at hand. This transitions the user to the task environment, where the virutal arms appear and the visual cues begin telling the user what to do. The user then interacts with the environment in whatever way is cued, until the number of task repeats reaches 0. At this point, an assessment result screen will pop up with data such as the elapsed time, as well as the number of repeats taken. The user can then return to the main menu and select more tasks or exit the application.



Throughout the implementation of these different pieces of the software, our team identified some major architectural points which should be expanded on in this section. The rest of this section will be dedicated to diving a little deeper into each of these important components, and how they all tie together to provide the system we showcased at EXPO. The most important components of this system are scene switching, task configuration, and the task runner.

Scene Switching

Unity is built on a system where different levels or menus are typically treated as their own encapsulated pieces of data, referred to as "scenes". This makes it slightly difficult to transition between these scenes. To remedy this, usually a manager-type object is used which persists between scenes and allows a player to make a call to it and tell it what scene to change to. Our system has a similar system in place, which uses the Singleton pattern, to create only one instance of this manger-type object throughout all scenes. Internally, this is referred to as the "Task System", and different objects in the scene (like the menu buttons) can make calls to this system to switch to a different scene (e.g., from a menu scene to a task configuration scene).

Recall that earlier in our design for the "task runner", it is mentioned that our team wanted to avoid the reliance on the singleton pattern for passing data between scenes. However, in the context of scene switching, utilizing a singleton allows for easily accessible methods to switch a scene on demand. And the use of the singleton pattern is not evil per se, from our research (reference the videos in the appendix), the singleton pattern can be overused and relied on too much. But for the single use case of switching scenes and handling these types of actions, the singleton pattern performs extremely well.

Task Configuration

The task configuration is an extremely important component of our system as well. As mentioned earlier, our choice for this design was to use ScriptableObjects to pass data between scenes. For example, when the user reaches the "task configuration" screen for any given task, they can interact directly with different configuration settings. However, this screen is in a separate scene than the actual task itself. This brings up the issue of how one preserves the data from the configuration screen scene, into the task environment scene. This is where the ScriptableObjects come into play.

In Unity, a script is defined which gives different classes for each type of task, and a boilerplate ScriptableObject structure is defined as well. When creating the configuration screen, a developer can create a new task configuration through adding a new class. Then, the ScriptableObject which contains all of this data is stored on the filesystem through Unity, and each scene can have a direct reference to it before runtime. At the configuration screen, when the user

is modifying the values, these are stored into the ScriptableObject. Then, when the task environment loads, the first action taken is to load the values form the ScriptableObject into the task to set itself up. This configuration provides the foundation to allow for consistent and repeatable tasks through the use of data written to and read from ScriptableObjects.

Task Runner

The main task environment is the scene in which the actual running of the task takes place. As mentioned in the Concept Selection section, the "task runner" is the name of the script which runs inside the main task environment for each invidudal task. This script has multiple purposes. Firstly, the script defines a reference to the ScriptableObject used for the configuration, which is tied to that specific task, and looks at the current data within. This is the "setup" phase, and any values which have changed within the ScriptableObject from the configuration screen will still hold the same value. This allows different objects within the scene to be set up accordingly.

After the setup is complete, the task runner enters the "run" phase. On this script there are also references to each event which needs to be emitted in order for the task to run. For example, in the default configuration of the Point-to-Point task, the even which gets emitted first is the "go to home" event. The script simply tells this event to emit itself, and that is all. From there, the following sequence of events are handled on different objects. The "home" object for example contains an "event listener", which holds responses for when certain events are emitted. When the "go to home" event is emitted, the "home" object responds to this event by glowing and changing its material to transparent. This indicates that the user should move to the "home" position.

Similarly, when the user reaches the home position, another event is fired, which is the "home reached" event. Here, both the home object and the start object are listening for this event. As the home object needs to change its material to completely transparent and turn off its light, and the start object needs to turn on its light, indicating that the user should move there next. This same type of logic is applied to all events and actions which occur during the running of the task. While our implementation most likely has issues, the concepts it was derived off of were very practical and modular. I highly encourage anyone attempting to pick up this system and continue work to watch the two videos linked in the appendix, as the system was inspired heavily from those talks. For reference, these talks are: "*Unite 2016 - Overthrowing the MonoBehaviour Tyranny in a Glorious Scriptable Object Revolution*" and "*Unite Austin 2017 - Game Architecture with Scriptable Objects*".

Future Work

Recommendations

Miscellaneous

The first recommendations our team highlighted for continued use of this software is to add some smaller unrelated additions. Firstly, integrating a pause menu into the task environment, and giving the assessor a way to stop the task if the patient cannot continue.

Currently, the task runs until it is complete, and does not assess the possibility of someone not completing the task. This was done for time's sake, but a further implementation would most definitely need this implemented. Similarly, another smaller feature would be to implement a settings menu of some type at the main menu. A generic settings menu would contain options to change the quality of the game settings, audio levels, and other such details. This addition is not strictly necessary, but if this software were to become a main use of people, then it would be an excellent feature to have. Implementing these miscellaneous improvements would be relatively quick and would most likely take only less than 8 hours or so to complete.

Integration with BLUE SABINO

The first of the major recommendations which our team has prepared is to integrate the BLUE SABINO outputs into our software. While our team had originally hoped to implement the additional software which would connect the BLUE SABINO to our environment, we realized quickly that this was most likely beyond our scope. Especially with the development of the BLUE SABINO simulation which another grad student was working on, our team is confident that work done in relation to the simulation can be combined with the task environment seamlessly. The virtual arms which our team use for demonstration purposes interact with the task environment solely through an invisible object that is pasted onto the hand of the arm. As soon as related teams have movements captured or simulated from the BLUE SABINO, and moving an arm within Unity, the invisible object can be placed on that new arm, and the task environment will still work the same. This connection between the two systems would be a huge leap for the project.

Additional Tasks

Another major improvement which should be made to the system is the addition of additional tasks. Originally, while our team understood that only a Point-to-Point reach task was absolutely required for the environment, we wanted to implement many different tasks. Due to time constraints, we clearly did not implement these other tasks. Some of these tasks with different variations could aim to evaluate range-of-force, control-of-force, range-of-motion, etc. Specially some interesting tasks we wished to look into father were the box n blocks test and the ARAT (Action Research Arm Test). Implementing each additional task is the most time-consuming feature which we are recommending. The system we built the Point-to-Point reach task off of can be observed and replicated for other tasks (as the design aimed for), but still requires ample time to get working correctly. In addition, creating or finding the assets for any additional task will also take at least 2-3 hours, if not more. Overall, implementing a completely new task into the system would conservatively take ~10 hours. That includes time spent researching, understanding the system in-place, and developing the new task.

In-depth Assessment Results

The final recommendation our team identified was adding in-depth assessment results when a task completes. Currently, due to the limited amount of information available solely within the environment, the assessment results screen only shows minimal information. This information is the time it took to complete the task, as well as the number of repeats which the

user has chosen to run through the task. While this current data is simple, adding more in-depth results is a clear path for expansion. As mentioned before, since other grad students were working on simulations for the BLUE SABINO, and capturing those outputs in Unity, this allows for the exoskeleton data to be analyzed in Unity as well. Our team has the vision that as this development continues, the data analyzed within Unity from the exoskeleton can then be assessed and displayed on this assessment result screen as well.

Appendices

- Unite 2016 - Overthrowing the MonoBehaviour Tyranny in a Glorious Scriptable Object Revolution
    - https://www.youtube.com/watch?v=6vmRwLYWNRo&t=10s

- Unite Austin 2017 - Game Architecture with Scriptable Objects
    - https://www.youtube.com/watch?v=raQ3iHhE_Kk&t=2152s