Halvor Bakken Smedås
Aksel Hjerpbakk
Nikolai Åkerholt
Jone Martin Skaara

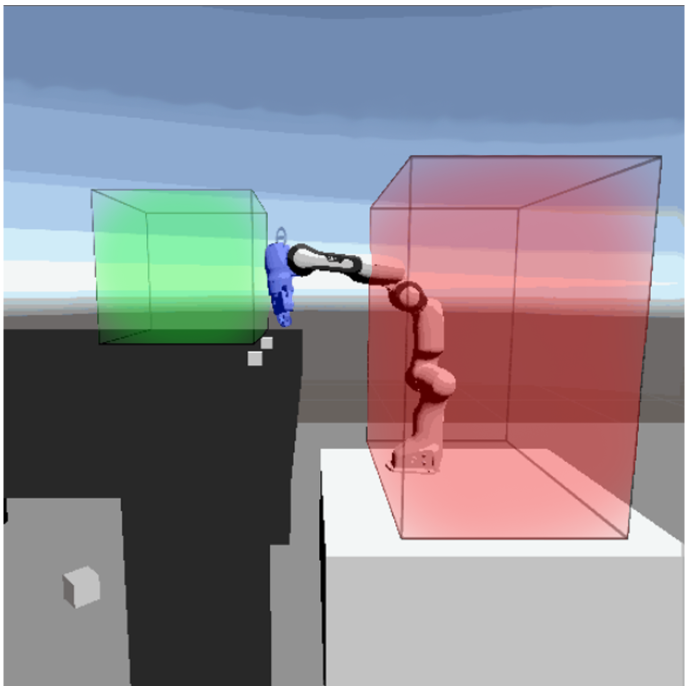
Neodroid Playground

Designing environments and tasks for learning robots in virtual reality

Bachelor's project in Programming [Games|Applications]
Supervisor: Mariusz Nowostawski
May 2019

Bachelor's project

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science



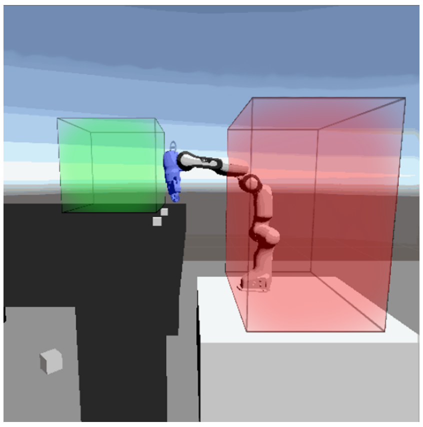
NTNU
Kunnskap for en bedre verden

SINTEF

Halvor Bakken Smedås
Aksel Hjerpbakk
Nikolai Åkerholt
Jone Martin Skaara

Neodroid Playground

Designing environments and tasks for learning robots in virtual reality



Bachelor's project in Programming [Games|Applications]
Supervisor: Mariusz Nowostawski
May 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

NTNU
Norwegian University of
Science and Technology

Neodroid Playground: Designing environments and tasks for learning robots in virtual reality

Author(s)

Halvor Bakken Smedås
Nikolai Åkerholt
Jone Martin Skaara
Aksel Hjerpbakk

Bachelor in Programming [Games|Applications]
20 ECTS
Department of Computer Science
Norwegian University of Science and Technology,

20.05.2019

Supervisor Mariusz Nowostawski

Sammendrag av Bacheloroppgaven

Tittel:	**Neodroid Playground: Design av miljøer og oppgaver for lærende roboter i virtuell virkelighet**
Dato:	20.05.2019
Deltakere:	Halvor Bakken Smedås Nikolai Åkerholt Jone Martin Skaara Aksel Hjerpbakk
Veiledere:	Mariusz Nowostawski
Oppdragsgiver:	SINTEF Ocean SeaLab
Kontaktperson:	John Reidar Mathiassen, John.Reidar.Mathiassen@sintef.no
Nøkkelord:	Bacheloroppgave, Kunstig Intelligens, AI, UX, Virtuell Virkelighet, VR, Unity, Programmering, Maskinlæring
Antall sider:	76
Antall vedlegg:	9
Tilgjengelighet:	Åpen

Sammendrag:

Denne bacheloroppgaven beskriver et Unity-prosjekt som forsøker å simplifisere utvikling av maskin-lærende robotter sine virtuelle miljøer og oppgaver. Prosjektet er todelt; en del baserer seg på å definere hvordan en bruker kan beskrive en oppgave for en lærende robot som tar i bruk Reverse Curriculum Learning. Den andre delen går ut på konstruksjon av et rammeverk for å definere moduler og grensesnitt for å beskrive oppgaver, slik at også eksterne utviklere skal være i stand til å bygge på kodebasen med minst mulig problemer i tilfeller hvor eksempelmodulene vi har bygd ikke strekker til. En av de mer utfordrende problemstillingene er implisitt konstruksjon av en evalueringsfunksjon for AI-agenten. Vi løste dette i systemet vårt ved å la brukeren beskrive tilstanden de ulike objektene skal være i for en måltilstand av miljøet. Dette gjør det mulig å beskrive en hvilken som helst oppgave i VR. Neodroid plattformen er ment for akademisk forskning, da prosjektet er open-source, men det er også et mål å løse problemstillinger i ulike industrier.

Automasjon innad i ulike industrier har vært et en problemstilling i flere tiår. Det er mange manuelle oppgaver kan bli automatisert hvis en tilstrekkelig løsning fantes. Dette vil føre til reduserte driftskostnader, og i tillegg redusere potensielt skadeomfang i farlige arbeidsmiljøer. Problemet er at det å automatisere arbeidsoppgaver ikke er enkelt. Et annet problem er at mange arbeidsoppgaver er for kompliserte og dynamiske til at de kan bli hardkodet.

En mulig løsning på dette problemet er å simulere robotens trening i et virtuelt miljø slik at det ikke er noen fare å trene den opp ved hjelp av maskin læring i stedet for hardkodet oppførsel. Vårt mål for Neodroid Playground er å generalisere hvordan arbeidsoppgaven beskrives ved hjelp av virtuell virkelighet.

Summary of Graduate Project

Title:	**Neodroid Playground: Designing environments and tasks for learning robots in virtual reality**
Date:	20.05.2019
Authors:	Halvor Bakken Smedås Nikolai Åkerholt Jone Martin Skaara Aksel Hjerpbakk
Supervisor:	Mariusz Nowostawski
Employer:	SINTEF Ocean SeaLab
Contact Person:	John Reidar Mathiassen, John.Reidar.Mathiassen@sintef.no
Keywords:	Thesis, Artificial Intelligence, AI, Virtual Reality, VR, Unity, Programming, Machine Learning, User Experience, UX
Pages:	76
Attachments:	9
Availability:	Open

Abstract:

This thesis describes a Unity project that aims to simplify the development of machine learning agents' environments and tasks. There are two parts of the task. It requires external developers to be able to extend it with relative ease. Users of the system need to be able to load up a scene and annotate a task with the use of VR. One of the more difficult issues to handle is implicit construction of an evaluation function for the AI. We solved this in our system by using conditions that describe what the desired goal state of the environment is. This makes it possible to describe any task through our interface in VR. The Neodroid platform is intended for use in academic research, as it is open source, but potentially also to solve issues in the industry.

Automation in the industry is something people have been striving towards for several decades. There are a lot of man-hours of manual labour that can be automated if there is a reasonable solutiton to them. This will save time and money, but additionally, it will help reduce damage potential for labour that is done in hazardous working environments. However, automating manual labour is not very straightforward, and the standard in the industry is that simple tasks that do not need any consideration of the environment can easily be done by a hard-coded robot. This becomes a problem when the tasks have complex issues that are too advanced for hard-coded robots to resolve.

A solution to these issues with automation is to simulate the robot in this environment and apply machine learning to it. That way, it will get years of experience through a simulated environment that is approximating the environment the task needs to be automated within. Our vision for the Neodroid Playground is to simplify and generalize a way to describe these tasks within virtual reality, such that a robot can learn how to do it in the simulated environment.

Acknowledgements

We would like to thank SINTEF Ocean SeaLab, John Reidar Mathiassen and Jonathan Sjølund Dyrstad for this amazing opportunity to contribute to their Neodroid Project, it has been a true joy working with you on this bleeding edge technology. We hope to see it and Neodroid Playground evolve and flourish in the future, and we are excited to help it along the way as we are going to keep working on it throughout the summer at SINTEF Ocean SeaLab, and further integrate it into the Neodroid Platform.

In addition, we thank Christian Heider Nielsen for taking the time to talk us through some of the more complicated concepts in artificial intelligence, and teach us about what the different modules of the Neodroid Platform does, and how they work.

We would like to thank our supervisor Mariusz Nowostawski for his counsel throughout the development of the Neodroid Playground, and through the process of writing this thesis.

Lastly, we would like to thank NTNU's Department of Computer Science, NTNU IDI, for their support; without them, the cooperation with SINTEF Ocean SeaLab would have been more difficult to arrange.

Contents

List of Figures

List of Tables

Listings

Glossary

Tools

Blender is an open source tool that offers 3D modelling, animation and simulation among other more in-depth features related to 3D media creation. 33

Discord is a communication platform that supports several text channels with user access control for individual channels [1]. 35, 69, 70, 73

Docker is a virtualization tool that does virtualization on a operating-system-level [2]. 36, 38, 65

Doxygen is a free software for generating/writing software reference documentation. The documentation is written within code, and is thus relatively easy to keep up to date. Doxygen can cross reference documentation and code, so that the reader of a document can easily refer to the actual code. 36–38

Franka Emika Panda A robot arm created by Franka Emika GmbH [3]. Widely used in the industry, and also used by SINTEF. A model of this robot arm is what we often used as the agent in the Playground. 50, 55, 56

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. 33, 36, 38, 66

GitLab is a DevOps-application including source code management and CI/CD operations. 33, 36–38, 65, 69, 73

Large File Storage A git extension that allows for handling of Binary Large Object data. xviii, 66

Microsoft Visual Studio is an Integrated Development Environment from Microsoft. As any IDE, it is used to draft and write code for software development while also giving extra functionality required to do so. xii, 33

NUnit is a unit testing framework for .NET code. Initially a ported version of JUnit a Java version of the framework. Version 3 is a complete rewrite rather than a port. 58

ReSharper is a static code quality analysis software. Can be used through its Command-Line Interface or with integration in Microsoft Visual Studio. 33, 37, 59

SAColliderBuilder A Unity package that build a collider for a mesh using only primitives. 48, 49

SteamVR A Unity package for working with Virtual Reality (VR). 1, 13, 22, 25, 26, 42, 48, 49

Toggl A time tracking tool for counting hours spent working. 33

Unity is a cross-platform real-time game engine developed by Unity Technologies. xii–xvi, 1, 3, 4, 7, 10, 15, 16, 21, 23, 26, 33, 36, 40, 43, 45–48, 52, 58, 61, 64–66, 71

Development Concepts & Terms

Microsoft Mixed Reality A category of virtual reality headsets that use inside-out tracking, allowing for portable VR with a short setup time [4]. 71

Object-Oriented Programming is a programming paradigm based on the concept of *objects*, which can contain data, in the form of fields, and code, in the form of procedures/methods. xviii, 46

Quality of Life Additional feature to *make life easier*. In software development this term is often used for features that improve user experience, or when used in the context of development, it might involve additional layers abstractions to simplify the interface to some code. xviii, 58

Refactoring To go over older work and improving/altering code structure without changing its behaviour. 22

Scrum An agile software development framework. 33, 69, 73

Test-Driven Development Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the software is improved to pass the new tests, only. This is opposed to software development that allows software to be added that is not proven to meet requirements [5]. xviii, 36

Unit Test is a software testing method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use [6]. 58

Code Concepts

Binary Large Object describes any large file that consist of binary data usually in the context of data management systems. xii, xviii, 66

Generic The concept of *type parameters* introduced to the .NET Framework, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code [7]. Sample: `public class SomeClass<T> where T : SomeOtherClass`. 16, 22, 26, 27, 40, 44

Pure Object A regular C# class. It does not derive from any Unity-classes. This often has implications on serialization as we need to tell it to serialize. Additionally, it will mean the object will not reside in the scene, but rather purely in memory, which often can be a desirable trait, as the former will cause some overhead (simply because they derive from larger classes and are maintained by Unity's different run time systems). 16, 71

Reflection Reflection provides objects (of type `Type`) that describe assemblies, modules and types. You can use reflection to dynamically create an instance of a type, bind the type to an existing object, or get the type from an existing object and invoke its methods or access its fields and properties. If you are using attributes in your code, reflection enables you to access them [8]. 24

Serialization The process of converting data to byte streams. Usually used for transference of data or intermediate object states. xiv, 16, 21, 45, 46

Singleton The singleton is a pattern for an object that *always* exist (if used), and is the only object of the specific type. The pattern ensure this by checking against a *static* field of its own type in itself and deleting itself if it finds that it is not the instance, if the field is not set when another object tries to communicate with it, the pattern instantiates an object to be *the instance*. 21, 41, 46

Surrogate is a feature designed to be used for type customization and substitution in situations where users want to change how a type is serialized [9]. 21

Thread A unit that execute commands/code. There can be multiple threads working in a single process, which allows the process to multitask. 67

Unsafe The unsafe keyword denotes an unsafe context, which is required for any operation involving pointers [10]. C# is a *managed* language, meaning it deals with pointers and references under the hood, so the programmer does not need to worry about making memory leaks, with the exception of code blocks marked as *unsafe*. 27

Unity Concepts

Component A Unity Component, typically implementing some specific behaviour for the GameObject. xiv–xvii, 16, 39–41

GameObject An object that has a presence in a scene, by extension it always holds a Transform. Typically it also holds some additional components defining its behaviour. xiv, xv, xvii, 11–13, 15, 39, 42, 45, 72

MonoBehaviour The base class of (nearly) all implemented Unity components. 21, 36, 45, 46, 71

Prefab A prefab is a GameObject stored aside as a reusable asset, complete with all its components, property values, and child GameObjects. The Prefab asset acts as a template from which you can create new Prefab instances in the scene. Allows for use of *the prototype pattern* on a GameObject level. 24

Rigidbody A Unity component that lets a GameObject have physics behaviour, such as applying forces on objects and gravity. 49

ScriptableObject The base class of classes that do not need to exist attached to a GameObject in a scene, but rather should exist on its own. This is typically used to store data both in run time and edit time. 24

Transform A special kind of a Unity component that exist on all GameObjects. It defines the position, rotation, and scale of the object. xiv, 40, 48, 49

AI Concepts & Terms

Entropy is a measure of uncertainty for a machine-learning agent. The higher entropy, the less it has "understood" of the environment, and takes more random actions.. 56

Frontier An abstract factor defining the training scope in Reverse Curriculum Generation. It increases incrementally as the agent learns how to complete its task in the environment, causing the agent to incrementally learn more and more of the full task. In a discrete environment, the frontier is directly comparable with the *expected number of actions to complete the task*. 2, 52, 54, 57

Reinforcement Learning is a machine learning concept based on how learning might occur in nature; where an agent might perform an action, and either cumulate a reward, or a penalty based on the outcome of the performed action. xv, 2, 54

Reverse Curriculum Generation is a state generation concept used in machine learning to train an agent in reverse. The agent will start in the goal state and expand out from it [11], gradually learning how to reach the goal state from a set of different start states. Each state is called a reverse curriculum point. xv, xviii, 2, 52, 54, 55, 57, 72

Sparse Reward is a rewarding mechanism used in machine learning, specifically in Reinforcement Learning. It provides few signals to the agent performing actions. Usually it only gives a signal when an agent has fully completed its goal, or when it has done something it was not allowed to. This is an approach of doing reinforcement learning that often leads to better results, as the agent will not be able to peak its performance by continuously striving for intermediary rewards rather than actually completing its task. This is can eality occur when using reinforcement learning, when sparse reward is not applied. 2, 3, 51, 53, 54

Neodroid Terms

Actor A component that works as a container for motors the AI can interact with. 3, 41

Agent A collection of machine learning algorithms that interfaces with Neo. xvi, 6, 10, 13, 14, 33, 51, 52, 55, 56

Droid The interface between Neo and a learning environment in Unity. xvi, 1, 10, 11, 13, 14, 18, 20, 51, 52, 57, 71

Neo A process that runs a given learning algorithm and interfaces through Transmission Control Protocol packets. xvi, 1–3, 6, 13, 14, 18, 33, 39, 51, 52, 57, 71

Neodroid Platform is the platform made by SINTEF including Neo, Droid and Agent. xvi, 1–3, 6, 7, 14, 36, 68

Neodroid Playground is the module we are developing to add a final abstraction layer between the Neodroid platform and the users of the system. 2–4, 7, 11, 13, 16, 18, 21, 22, 33, 52, 55, 71

Neodroid Project is the reseach project funded by the Research Council of Norway. It consists of the Neodroid Platform, the development of Artificial Intelligence that processes 3D images to be used as input, and testing of AI learning processes when used on real-life robots. 1, 6

Playground Terms

2D-Widget User Interface elements such as sliders and dropdown menus that are used to define input to conditions similarly to 3D-widgets. Although 2D-widgets are operated using a laser pointer in Virtual Reality, or potentially a mouse outside VR. xvii, 9, 12, 13, 23, 26, 29, 30, 43, 44

3D-Widget A GameObject in the scene that is VR interactable, and can be used to define a volume, a vector, numerical value or any other property that could be used as to define a property of a condition. In code they are named `WidgetVR` precisely for this reason. xvi, xvii, 9, 12, 13, 23, 26, 29–31, 45

Absolute State A purely internal state, i.e. an object-state which can be considered a global fact as it does not depend on external factors (side effects). Any condition with no relatives will operate on absolute states, either evaluate properties of themselves or their context SceneObject. 16, 39

Annotate To give explanation on how a task should be performed by describing it through User Interface. xvii, 1, 3, 4, 7, 13, 17, 21, 47, 68

Condition A constraint defining what states an object can be in, in order to fulfill a goal state or terminating state. Like being inside a volume or touching another object. All conditions can be evaluated to satisfied (`true`) or unsatisfied (`false`). xvi, xvii, 4, 6, 9, 10, 12–14, 16–18, 21, 25, 27–30, 36, 39–42, 44–47, 53, 55, 56, 58, 60, 62–64, 67, 68, 71, 73

Condition Data Container is a container for all required data within a condition. The containers are used for serializing and de-serializing data for loading and saving profiles. 21

Condition Editor A class responsible for the creation of, and maintenance of widgets used to interface a condition. 22, 23, 27

Condition Group A special type of condition that allow nesting of conditions by utilizing the composite pattern. The evaluate function propogates an evaluate-call. 17, 41, 42, 46, 53, 62, 67

Context Object The SceneObject that is held as a context in a condition. xvii, 16, 18, 30, 39, 63

Environment Engineer The person that builds a virtual environment by populating a scene with SceneObjects, and defines new conditions that is needed for a given tasks to be performed/trained in the environment. xvii, 8, 21, 41, 42, 45, 46, 48

Goal State is the desired state of an object or a scene. When the AI-agent has fulfilled a goal state it is given a reward. xvi, 2, 9, 17, 18, 28, 53, 54, 63, 68

Relative A component that a condition's fulfillment depends on. A good example of this is the *TouchCondition* (full code F.2) which uses the `Collider` of a SceneObject as relative, essentially constraining the condition to only evaluate touches between the context of the condition and the relative. xvi, 16, 31, 39, 40, 63

SceneObject A GameObject in the scene that has the attached *SceneObject*-component. It has a goal and terminating list of conditions and is evaluated as a part of the scene state to use as input for AI. xvi, xvii, 9, 11, 17, 18, 20, 21, 28–31, 39, 40, 42, 43, 45, 46, 51, 53, 61–63, 67, 68

Terminating State is when an AI-agent has done something it was not supposed to. This will cause the session to terminate and the agent will be reset and punished. xvi, 3, 9, 17, 18, 28, 53, 63, 67, 68

Trainer The person that interacts with an environment built by an environment engineer in order to define a task for the AI by annotating the conditions they want. 4, 7, 21, 26, 31, 41–43, 47, 73

Widget An abstraction for interactable elements used to annotate conditions. (see 3D-widget & 2D-widget). xvii, 9, 12, 13, 22, 24, 25, 27, 44, 73

Acronyms & Abbreviations

AI Artificial Intelligence. xvi, 1–3, 7, 33, 39, 41, 42, 48

API Application Programming Interface. 1, 4

BLOB Binary Large Object. xii, 66

CI Continuous Integration. xii, 33, 36–38, 59, 65

CLI Command-Line Interface. xii, 33, 37, 58, 59, 65

CPU Central Processing Unit. 67

DOTS Data-Oriented Technology Stack. 15

ECS Entity Component System. 15

FPS Frames per Second. 3, 13, 14, 60

GPU Graphical Processing Unit. 13

GUI Graphical User Interface. 6, 25, 58, 65

IDE Integrated Development Environment. xii, 33

LERP Linear Interpolation. 43, 49, 57

LFS Large File Storage. 66

OOP Object-Oriented Programming. 22

QOL Quality of Life. 58

RAM Random-Access Memory. 67

RCG Reverse Curriculum Generation. xv, 2, 3, 54, 55, 68, 71, 72

RCP Reverse Curriculum Point. 2, 54, 57

TCP Transmission Control Protocol. xvi, 1

TDD Test-Driven Development. 36, 58, 64

UI User Interface. xvi, 2, 4, 14, 23, 24, 26, 29, 30, 43, 45, 58, 63, 65, 71, 73

VR Virtual Reality. xiii, xvi, 1–4, 6, 7, 10, 13, 14, 26, 29–31, 33, 40, 41, 43, 47, 48, 50, 63, 65, 71

1 Introduction

1.1 Background

SINTEF Ocean SeaLab (hereby referred to as SINTEF) is actively building on something they call *Neodroid project* which is a collection of smaller projects. *Neodroid platform* is one of these, built in collaboration with Christian Heider Nielsen to create "a reality-ready robot brain in virtual reality" [12]. When we started this semester there were two main modules to this platform: Neo and Droid. Neo is a python software that uses Transmission Control Protocol (TCP) to communicate with other processes and uses this data to learn from the said process. Droid is a Unity package and an Application Programming Interface (API) for developers to communicate with Neo using standard Unity C# programming.

The ultimate goal for SINTEF with the Neodroid project is to have a framework that can be used by staff in industries in many different disciplines. This framework should be used to demonstrate how a given task should be performed. Then have a robot trained in this task until it is deemed capable of performing it with a high precision, to then have it deploy image data for training the physical robot.

Our goal is to add a final abstraction layer for the end user. This layer would become Neodroid Playground which allows the user to annotate an agent's environment with little to no prior knowledge about Unity or Artificial Intelligence (AI). Ultimately, enabling any layman to create an environment through VR for an agent to learn in.

1.1.1 Academic Background

The educational program has introduced us (the students) to the basics of concepts like AI and game programming. We have touched upon topics like machine learning and reinforcement learning. We also have become sufficiently skillful in game oriented code and code abstractions to simplify and solve complex issues. These are some of the relevant skillsets which have come in handy throughout the development of the Playground. There are also aspects of professionalism that we need to account for while working on more complex, open source systems, such as work ethics, coding conventions and documentation.

We have also had experiences with working on VR environments in the Unity engine already, and specifically C# and the SteamVR API. This has given us a good basis for issues we need to deal with, and how we should design and structure the system.

1.1.2 Subject Area

Virtual Reality

VR is a fairly new technology to work with, and good user input methods are constantly evolving. The standard for VR equipment is constantly being changed, and because of that, software made for VR also must adapt to these changes. One example is that some VR headsets motion controls use a touchpad (Vive) while others make use of an analog thumb-stick (Oculus).

User-Experience and User-Interface

User experience in VR is also quite a design challenge. User Interface (UI) in VR must be designed very unique compared to mouse and keyboard applications. The user must also be comfortable with the environment they are placed in.

Reinforcement Learning

Machine learning is a central theme in the Neodroid Playground as Neo is based on this. Reinforcement learning is a subclass of machine learning, and is based on either rewarding or punishing an agent for its actions. Even though AI is not going to be the main focus for our project, we are going to develop a system that works with the already implemented machine learning algorithms in the Neodroid platform, which means it requires consideration on our end to make AI training possible. There are several concepts that can be used to apply machine learning, and one of the desired techniques for use in Playground is sparse rewards. *Sparse Reward* is a principle of only giving a reward signal when the task is fully completed, otherwise, the agent will only receive zero or minus one as signals (if it did something it was not allowed to). Using sparse rewards simplifies the concept, but results in a high probability that the agent will never learn as it is unlikely to reach a goal state. One way to solve this is to use Reverse Curriculum Generation (RCG), which initializes the agent with a simple state at first, with very few steps required to reach the goal state (a Reverse Curriculum Point (RCP) with low difficulty). It will learn to recognize these states before expanding its frontier by initiating it with progressively harder RCPs to solve the problem from.

Simulate / Emulate Physics

In the Playground, the robot should learn to interact with a number of objects that will behave as realistically as possible. In some cases, it may be relevant to simulate their properties, such as soft-body physics to determine the behaviour of the object in order to make it as true to life as we can.

1.2 Project Scope

1.2.1 Limitations

The Neodroid platform is intended for use in academic research, specifically for the developer's own interests. Our task is to streamline the use of the platform with regards to the construction of a playground and its user interface. The goal is to make it easier to go from specific issue to implemented solution as quick as possible.

1.2.2 Task Description

Our task is to create what we call the Playground module. This includes focus on how annotating and demonstrating a task should be done, but also generalizing the overall Neodroid platform. This means we need to make it possible to easily describe a range of different tasks to an agent within this playground, and make it possible for the agent to learn that specific task if the environment is set up correctly. Having a developer set up an environment for a task is the first part of the intended flow. It is desirable to create a standard for different types of objects, and what components they need within the playground. The playground should then consist of a set of objects (such as the robot, and interactable objects).

After playground construction is done, a user should be able to demonstrate the task the agent is supposed to learn correctly and seamless without any necessary prior knowledge about Unity or AI. This will be an essential part of the task, as a good user interface is crucial for a good product. The user will be defining a task by annotating goal- and terminating states according to the desired behaviour of the agent. The Actor should be able to deal with simple single-goal environments, but should also be able to potentially take several important "intermediate" states. These states will then be used to generate the data set for learning by RCG. If the order of these states is important, that should also be considered by the agent.

When annotating objects and points in a demonstration, one should be able to emphasize what observations are important for the goal state. In some cases, one might need to consider the position of the object, and for other cases, it might be the rotation, or it could also be a combination of these observations. These constraints should be annotated and taken into account. The process of defining small precise targets should also be as intuitive as possible for the user, and optimally allow editing of several objects' properties at once.

For the machine learning, sparse rewards and RCG is used to achieve the desired result. In order for the agent to be able to learn how to optimize the result, one can also use energy minimization to avoid unnecessary steps. When the agent is crashing into objects it is not supposed to interact with, or doing unwanted actions it is given a penalty and terminated.

1.2.3 Restrictions

Documenting or refactoring existing code in the Neodroid platform is not part of the task. However, where applicable we are encouraged to do so.

It is not part of the task to expand Neo (the machine learning component of the system), even though if we would see it necessary we are free to expand on it.

Our task is not to train a real robot. We do not have access to a physical robot to try this, but the product owner should have access. The virtual playground will be only operating with perfect data for learning. The reason why we use perfect data is that the robot should train itself in the virtual world, the physical robot will then use image data generated from the training to apply machine learning based on those images.

1.2.4 Boundaries

The application we develop is made in the Unity game engine, as specified by SINTEF since their existing framework is made in this environment.

It is not a focus to extend the Neodroid platform's existing modules. We are however encouraged to improve it. The main focus is on creating a Unity application with a VR playground where the user can demonstrate a task. Playground can be used to train an AI to perform a demonstrated task. The Playground is planned to be integrated as a part of the Neodroid platform in the future.

Performance is important both for the Playground part of the application and for the simulation section. In VR, it is important that the frame rate is more than 90 to avoid VR sickness.

1.2.5 Target Audience

The target audience is a combination of non-technical people and developers using Unity. The layman should be able to pick up the headset and motion controllers and annotate valid conditions for an agent to follow.

For this person, it is not very relevant how we evaluate the agent-environment state, how condition logic is built or any other technical details. The user can communicate with our system through the VR-UI.

The Unity developer needs clean code to read, as well as simple and clear instructions on how a scene should be built. Like the layman, the developer needs their own interface, although this person will be more exposed to an API that allows them to expand the functionality without having to rewrite existing systems.

1.3 Project Goals

1.3.1 Business Goals

- Create a standard for components used in a Playground, such that a technician can build the needed playground with ease, let a demonstrating user define the conditions and virtual robots interact with it.
- Allow for simulation of a scene with realistic behaviour.

1.3.2 Impact Goals

- Generalize how robots are used in industry - if a robot can learn and do any number of different tasks, it will mean that you do not need to construct hard-coded specialized robots.
- Reduce damage potential for people and equipment in the context of automation in hazardous work environments - a robot can be taught in complete safety using the virtual training environment

1.4 Thesis Structure

The document consists of 9 chapters.

1. Introduction - Project overview and our motives for working on this bachelor thesis.
2. Specification - Chapter describing functional requirements and use-cases for the application.
3. Technical Design - Describing components and underlying architecture in the Playground system.
4. User Experience Design - Describing how we have approached user experience design, both by making user interfaces simpler, as well as dealing with input in a sensible way.
5. Development Process - Tools and processes used during the development of the software, and how they were used.
6. Implementation - Discussing how the functionality was implemented from a lower level perspective.
7. Testing and Quality Assurance - Discussion about code quality assurance and profiling in Unity.
8. Discussion - Discussing different aspects of the project and evaluate our own work.
9. Conclusion - Final thoughts on the project and results.

2 Specification

2.1 Functional Requirements

We are going to develop a module that is supposed to generalize the Neodroid platform, making it easier to implement a solution for teaching a robot any type of task that it should be able to do in the real world. This means our system must provide a natural and precise way of defining conditions for someone with little or no technical knowledge. The system must also be very flexible as it should be possible to teach a robot both simple and advanced tasks.

We must also have developers in mind, as they are supposed to set up an environment for an expert within a field to demonstrate how the task should be done correctly. The system must be easy to understand, and components needed for it to function correctly should be well defined. This is important to developers, so they can easily set up their own environment, and possibly expand on features or create their own where they feel it is necessary. As the Neodroid project is open-source, we also feel it is important that the code is easy to understand and well documented. This makes it simpler to understand and expand.

Integration with existing Neodroid modules must work as intended. Meaning that any task set up with any type of actor(s), should be able to learn through the learning process in Agent if the environment is set up correctly by the developer. Furthermore, saving and loading conditions into scenes must be possible, making it easier to go back and edit an environment's conditions. Loading an environment with pre-set conditions from the demonstrator or from earlier sessions is also very practical when applying machine learning, so one does not need to re-demonstrate every time.

2.1.1 User Stories

- The users can with ease start training with Neo when having a valid playground scene.
- The user can select any SceneObject in the current scene using a VR laser pointer. After selection, the user can view the objects conditions through a Graphical User Interface (GUI).
- Through the selection-GUI, the user modifies condition that is being displayed.
- Through the selection-GUI, the user adds a condition to goal and to terminating list.
- Through the selection-GUI, the user deletes a condition from goal and from terminating list.
- Through a menu system, the user can save and load conditions into the scene.

2.1.2 Use Cases

The ultimate goal of this technology is to allow a person who does not know anything about AI, VR, or Unity to train an AI to do any task in any environment. This would involve making some system that allows the trainer to both define the environment, and the tasks. As an intermediary step, the thought scenario is that we have a system that allows the end-user to specify the tasks for the AI, but leave the construction of the training environment up to someone with experience in the platforms used (i.e. Unity & Neodroid platform). This essentially means that the Playground is both a framework and a tool; as such, it made sense to model the use cases into two diagrams:

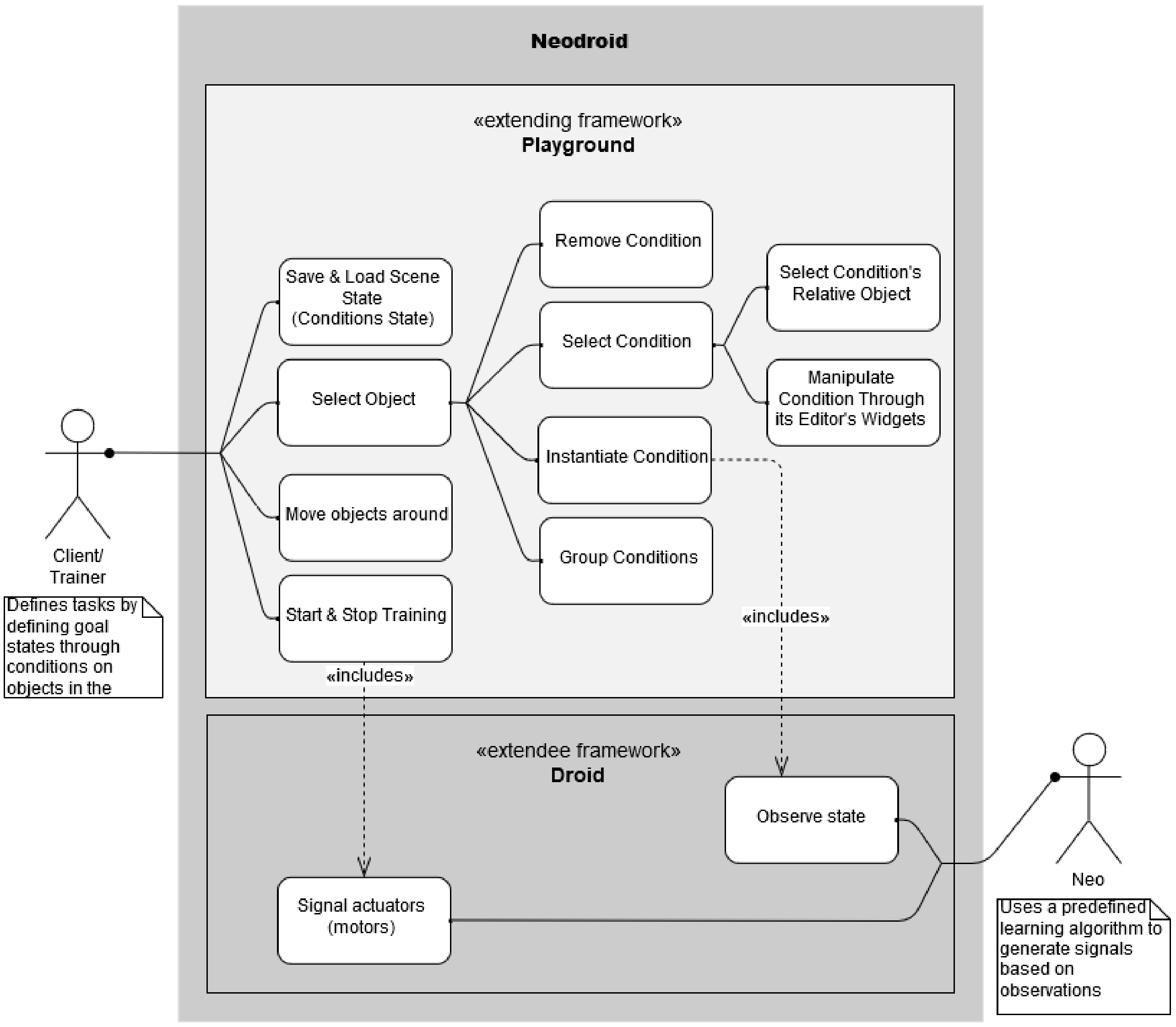

Neodroid
«extending framework»
Playground
Save & Load Scene State (Conditions State)
Select Object
Move objects around
Start & Stop Training
Remove Condition
Select Condition
Instantiate Condition
Group Conditions
Select Condition's Relative Object
Manipulate Condition Through its Editor's Widgets
«includes»
«includes»
Client/ Trainer
Defines tasks by defining goal states through conditions on objects in the
«extendee framework»
Droid
Observe state
Signal actuators (motors)
Neo
Uses a predefined learning algorithm to generate signals based on observations

Figure 1: **Run time Use Cases** - Use cases showing the capabilities of the Playground during run time (also called *annotation mode*), in which a trainer interacts with the different components already built as part of the Playground, or components deriving from them.

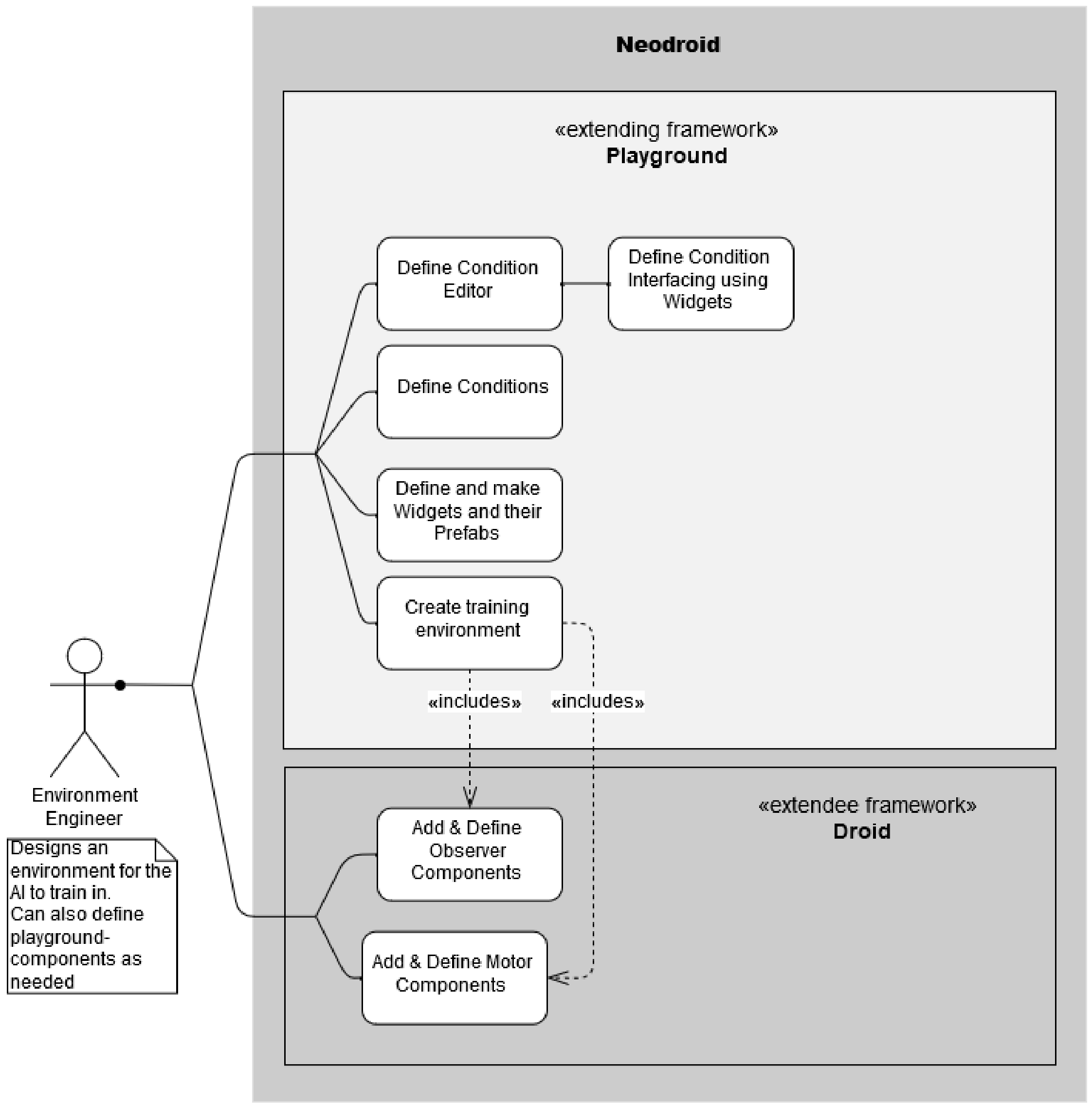

Neodroid
«extending framework»
Playground
Define Condition Editor
Define Condition Interfacing using Widgets
Define Conditions
Define and make Widgets and their Prefabs
Create training environment
«includes»
«includes»
«extendee framework»
Droid
Add & Define Observer Components
Add & Define Motor Components
Environment Engineer
Designs an environment for the AI to train in. Can also define playground-components as needed

Figure 2: **Edit time Use Cases** - Framework use cases show the use cases of the Playground as a framework, highlighting what a developer, or environment engineer will use it for.

2.1.3 High-Level Use Cases

Run Time

Use case:	Select object
Primary actor:	Client/Trainer
Goal:	Edit an object's conditions.
Alternative goals:	View an object's conditions.
Regular flow:	1. The actor points at SceneObject. 2. The actor selects the object by button press. 3. SceneObject's condition overview menu is displayed (see fig. 13).
Possible variations:	None

Use case:	Instantiate condition
Primary actor:	Client/Trainer
Goal:	Add a preferred state (goal state) to the selected object.
Alternative goals:	Add an undesirable state (terminating state) to the selected object.
Regular flow:	1. The actor chooses the desired condition type 2. The actor selects the condition's relative object 3. The actor configures condition through its editor's widgets 4. The actor saves condition setup
Possible variations:	The actor can choose to not select a relative object. This means the condition would default to be relative to the environment. The actor can also choose to not configure the condition values, which results in the condition saving its default values.

Use case:	Manipulate conditions through its editor's widgets
Primary actor:	Client/Trainer
Goal:	Configure intended condition values
Alternative goals:	None
Regular flow:	1. The actor has a SceneObject's condition selected. 2. The actor can change a condition's values by interacting with the widgets provided by the condition's editor: 3. The widgets consists of two different types: • 2D-widgets: Configuring values through a 2D-Tablet in VR (see section 4.2). • 3D-widgets: Configuring values through 3D-representations of condition values (see fig. 15).
Possible variations:	None

Use case:	Group conditions
Primary actor:	Client/Trainer
Goal:	Describe a condition with a group of several conditions to achieve the desired result.
Alternative goals:	Group conditions to describe the correct order of completion to fulfill a task.
Regular flow:	1. The actor opens the condition overview menu (see fig. 13). 2. The actor drags a condition from the conditions tab into another existing condition in goal or terminating list. 3. The actor can instantiate any of these grouping types: • AND • OR • XOR
Possible variations:	None

Use case:	Save and load scene state (conditions state)
Primary actor:	Client/Trainer
Goal:	Save current setup to run machine learning on it later.
Alternative goals:	Save the setup so one can load and edit it later.
Regular flow:	1. The actor opens VR saves-menu 2. The actor gives name to the current setup 3. The actor saves the current setup to disk locally
Possible variations:	The actor can also choose to manage the saves from Unity's editor, using an EditorWindow for Playground-saves. If the scene exits runtime without having the current setup saved to disk yet, a temporary save will be written to disk, so the actor does not lose unsaved changes.

Use case:	Start and stop training
Primary actor:	Client/Trainer
Goal:	Apply machine learning so the agent learns the task.
Alternative goals:	None
Regular flow:	1. The actor has an environment with set conditions active. 2. The actor starts a process in Neodroid-Agent for machine learning. 3. This process connects to the running Droid environment to receive and send signals.
Possible variations:	None

Edit Time

Use case:	Create training environment
Primary actor:	Environment Engineer
Goal:	Replicate the surroundings of the intended task's environment to the best extent possible. Later on, this will be given to the client/trainer, so they can annotate how the task in it should be performed. After annotation in this environment is done, the goal is to apply machine learning on the agent.
Alternative goals:	None
Regular flow:	1. The actor has a Unity project with the Playground and Droid module. 2. The actor creates a new scene. 3. The actor adds all required components for a Playground-environment. 4. The actor can now do several things to create his environment: • Add new SceneObjects • Add and define observer components • Add and define motor components
Possible variations:	If the actor does not have the Playground or Droid correctly setup, the actor will receive warnings and have to read up on documentation on how to resolve the issue.

Use case:	Add and define observer components
Primary actor:	Environment Engineer
Goal:	Ensure all correct observations needed is sent to the AI when applying machine learning.
Alternative goals:	None
Regular flow:	1. The actor selects a GameObject in the environment that needs to be observed. 2. The actor adds the observer components required on the specific object.
Possible variations:	The actor might not find an observer suitable for his needs and could decide to implement his own observers to fulfill this.

Use case:	Add and define motor components
Primary actor:	Environment Engineer
Goal:	Create an agent that can interact in the scene with output from the AI.
Alternative goals:	None
Regular flow:	1. The actor adds his desired agent GameObjects to the scene. 2. The actor adds motor components to this agent.
Possible variations:	The actor might not find a motor component suitable for their needs, and could decide to implement his own motors to fulfill this.

Use case:	Define conditions
Primary actor:	Environment Engineer
Goal:	Create a new condition type as there does not exist any that fulfills the actor's or client's needs.
Alternative goals:	None
Regular flow:	1. The actor creates a script that inherits from the base condition class or Condition<TRelative> 2. The actor creates a container with mirrored variables 3. The actor creates an editor for the new condition. 4. The actor creates the interface for the editor with new or existing widgets.
Possible variations:	The actor might not know the structure of the conditions and how they are implemented. This means that the actor must read up on existing documentation on conditions, and also on how to implement their own.

Use case:	Define condition editor
Primary actor:	Environment Engineer
Goal:	Create a way for the client/trainer to interact with a condition's values.
Alternative goals:	None.
Regular flow:	1. The actor creates a script that inherits from the base ConditionEditor class. 2. The actor creates widgets to be used by the editor for user interaction. These could be either 2D-widgets or 3D-widgets. 3. The actor implements the interface for the condition by implementing the widgets into the editor.
Possible variations:	The actor might already find existing widgets suitable for the specific editor and can just directly implement these into the editor.

Use case:	Define and make widgets and their prefabs
Primary actor:	Environment Engineer
Goal:	Create an interface for editing a condition's values by implementing these widgets into a condition's editor.
Alternative goals:	None
Regular flow:	1. The actor creates a script that inherits from the 2D-widget or 3D-widget class. 2. The actor creates GameObject prefabs for these widgets.
Possible variations:	None

2.2 Supplementary Requirements

2.2.1 System Requirements

Platform

Our Playground module is mainly about interacting with a scene in VR, so naturally, our application requires a VR headset. Preferably HTC Vive, as our platform is mainly developed for that. However, other headsets will supposedly also work, as long as it has controllers or some other input mechanism for interacting with the scene. We have been using the SteamVR module for implementing VR into our project, and it is aimed at working with multiple VR headsets [13]. Despite functionality probably working with other headsets, we can not guarantee it. We have decided to mainly target and develop for HTC Vive on Windows 10, and make sure functionality is working as intended for that platform. This is due to our time limit within the scope of the project.

For the machine learning part done in Neo, it is preferred to have a decent Graphical Processing Unit (GPU). This is also the case for our VR environment, even though the scene might not be that complex, it is important to have sufficient frames per second (FPS) to avoid VR sickness.

Playground depends on the Droid module to function correctly. Additionally, to run machine learning it is required to have a functional instance of the Agent module set up.

Recommended system to run the application:

- Minimum Nvidia GTX 970 GPU
- 3 GHz Processing power
- Minimum 8GB RAM
- HTC Vive-headset

2.2.2 Performance

The main performance measurements in our system are frame rate and scene evaluation speed. Frame rate is important in VR to avoid potential motion-sickness caused by low FPS [14]. The speed of condition evaluation is critical because it will affect FPS in both annotation mode and also machine learning mode. We do not want scene evaluation to affect the performance of either of these modes, therefore it is something we need to consider thoroughly. Secondary performance measurements for our system is saving and loading speed of scenes' conditions and how much disk space these saves use.

Requirements

- The application must be running above *90* FPS while in VR.
- The system must be able to handle up to *500* conditions without any noticeable performance hit.
- Loading *500* conditions should take less than *1* second on a SSD (solid state drive).

2.2.3 Usability

The intentions of the module lie in the hands of a developer's own interests, or for a client that wants a task to be automated. For this reason, it is important that our system is intuitive to use and understand. For a developer, it must be simple to understand our code and how to use it and expand it. Additionally, a client using the system might not have technical knowledge of how machine learning or VR works. This means it is important that our user interface is easy to understand and interact with. A client must be able to set up correct conditions on an environment with ease, with a few simple instructions on how the Playground works.

Requirements

- Setting up conditions must be intuitive, and have a UI with as little friction as possible.
- Conditions that are set up should be displayed if wanted, and they must be simple to understand.
- The code must be easy to understand from a developer's point of view, well documented and be expandable.

2.2.4 Neodroid Integration

Our system is not a stand-alone module, though parts of it will work without dependencies. It is supposed to be working with both the Neo and Droid modules in Neodroid platform. This has brought our attention to how we should interface with these existing modules, to make interaction fluid and functioning.

Requirements

- A universal method of evaluating Droid-environments with conditions regardless of the condition types must be operative. Each specific condition type must have its own working evaluation function.
- Starting a learning process in Agent and connecting it with the Playground should be possible.
- Proper documentation explaining what components are required for the system to correctly interact with the other Neodroid modules.

3 Technical Design

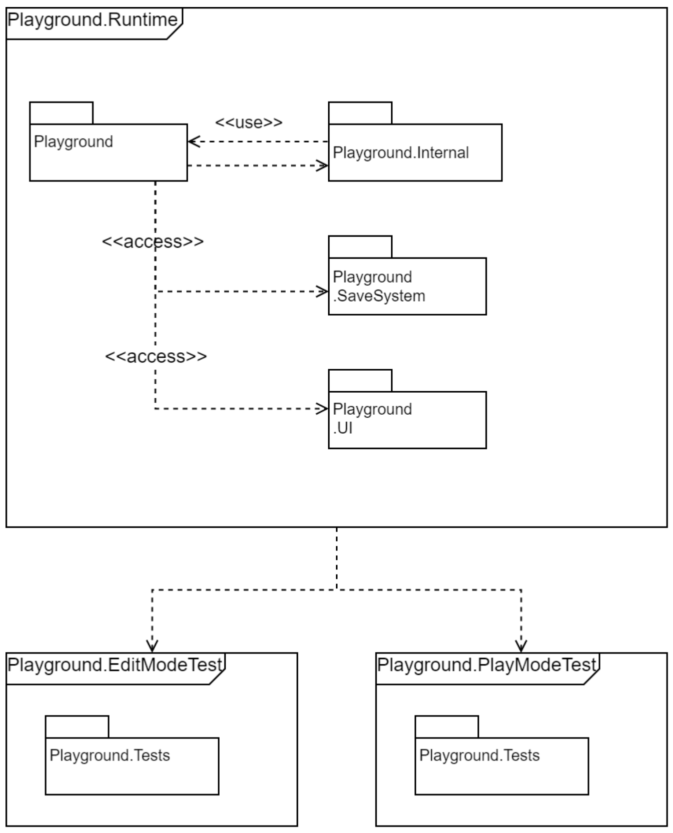


Figure 3: System overview (Package Diagram): There are 3 assemblies in our system with a total of 5 namespaces.

3.1 Unity's Paradigm

As we work in Unity, all work we do has to take Unity's paradigms into consideration when we design and implement features. Object-oriented code is one of these paradigms. Unity uses C# as its main language. C# is mostly object-oriented in nature. For example, C# does not allow functions to exist without a class. This will change in the coming versions of Unity where they are planning to properly deploy their burst compiler and Entity Component System (ECS) (data-oriented pattern [15]) through their Data-Oriented Technology Stack (DOTS). This will make Unity more data-oriented in nature.

Unity uses the component pattern to keep as many systems as possible independent of each other. Every GameObject can have any number of components that do different tasks. Adapting to the component paradigm in Unity delivers a lot of benefits in developing. For instance inspection of values in the inspector during run time and easy communication between different GameObjects.

Components work by providing functionality for other objects without having to use inheritance. The reason for using the component pattern is to avoid the *deadly diamond of death* problem [16] and duplicate data/logic [17].

3.2 Conditions

Conditions are one of the core elements that Playground consist of. In essence, they are components that wrap boolean expressions of any complexity describing a certain aspect of a scene's state. Initially, they were planned to be pure objects as they are generally more light-weight; hence, they're faster. We later redesigned this, as going with pure C# classes added a lot more complexity in regards to serialization and in-editor displaying of the objects, mainly due to the lack of support for serialization of generic classes in Unity (see sections 6.4 and 8.4).

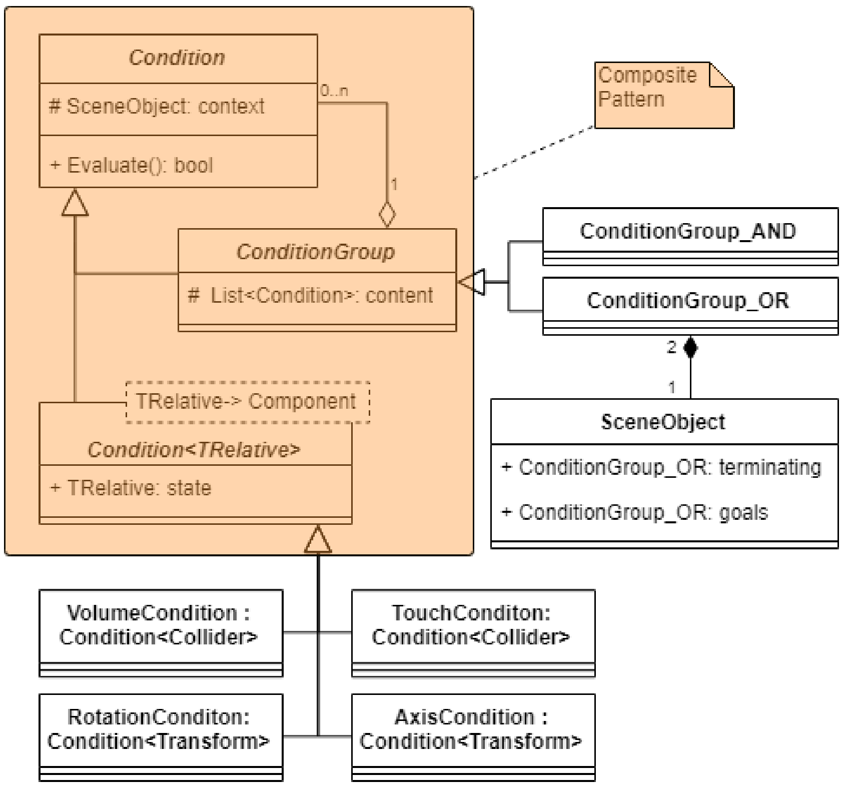

Condition
# SceneObject: context
+ Evaluate(): bool
0..n
1
Composite Pattern
ConditionGroup
# List<Condition>: content
ConditionGroup_AND
ConditionGroup_OR
2
1
TRelative-> Component
Condition<TRelative>
+ TRelative: state
SceneObject
+ ConditionGroup_OR: terminating
+ ConditionGroup_OR: goals
VolumeCondition : Condition<Collider>
TouchConditon: Condition<Collider>
RotationConditon: Condition<Transform>
AxisCondition : Condition<Transform>

Figure 4: The Fundamental Design of Conditions. Highlighted is the composite pattern.

3.2.1 Condition Relatives

We found that we would be able to describe fairly complex scene states with just two fundamental condition types: One describing absolute states, meaning it does not depend on external factors (i.e. no relative dependencies), but rather purely on its context's own internal properties. The other type describes the complete opposite: Conditions that *are* dependent on external factors. We have designed a special kind of condition for this, that uses C#-generics to defer what kind of component it depends on. We have come to call these components *relatives*.

3.2.2 Terminating Conditions and Goal Conditions

Conditions can be used to annotate tasks in two ways: A condition can be enlisted as a *goal condition*, where the condition embodies a requirement for the scene to be in a goal state, or it can be enlisted as a *terminating condition*, where the condition, if fulfilled puts the scene in a terminating state.

3.2.3 Group Conditions

As illustrated by the figure above (fig. 4), we have a special type of condition that can contain other conditions. This *condition group* embodies the structural pattern known as the *composite pattern*. It is useful in cases where a task is more complex than just one condition on a SceneObject and depends on states of inter-condition fulfillment. The thought behind these composite constructs is that they allow us to model highly complex logical evaluations of a scene's different conditions.

We have modeled three condition groups thus far, where only two are actually used in the current version of Neodroid Playground: the *AND*-, *OR*- and *XOR* groups, which incidentally are named as such because they closely resemble the equally named logic gates in electronic circuits in behaviour. Seeing as a condition group may contain more than two conditions, we have had to extend the behaviour of them beyond that of logic gates, as they typically only have two inputs, whereas we have range of $0 - n$ inputs in our groups' evaluation functions.

The *AND*-group requires *all* conditions to be satisfied, the *OR*-group requires *one or more* conditions to be satisfied, lastly the *XOR*-group requires *one and only one* condition to be satisfied. With this being the case, we might yet change the name of these in the future, to further emphasize their use.

3.3 SceneObject

The SceneObject was designed as one of the core component of Playground. It acts as the subject of conditions (i.e. the context of a condition), and as the container of them. Each SceneObject is observed by the Neo network through Droid-*Observers* that gather information about objects' states relevant for machine learning.

Below (figs. 5 and 6) we have described a simple scene to showcase how SceneObjects are used to construct a definition of goal states. There are five objects: 1 mug, 2 plates, and 2 tables, each of which hold their conditions grouped in a specific way to convey the requirements of the goal state. In this scenario the goal state is to have the mug placed on one of the plates, facing upwards, and also to have the plates facing up relative to the either of the two tables.

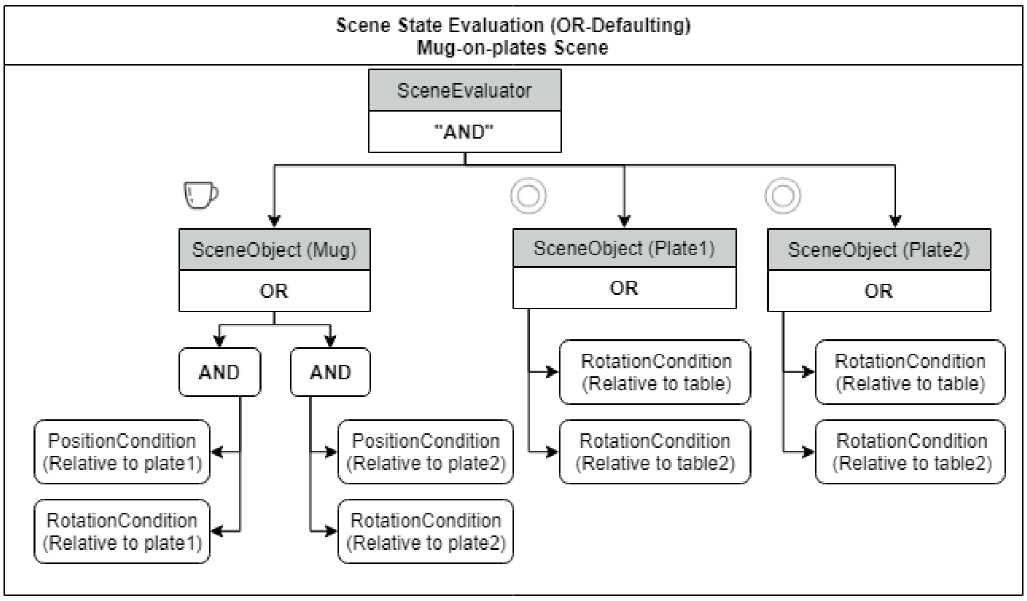

Scene State Evaluation (OR-Defaulting)
Mug-on-plates Scene
SceneEvaluator
"AND"
SceneObject (Mug)
OR
SceneObject (Plate1)
OR
SceneObject (Plate2)
OR
AND
AND
PositionCondition (Relative to plate1)
RotationCondition (Relative to plate1)
PositionCondition (Relative to plate2)
RotationCondition (Relative to plate2)
RotationCondition (Relative to table)
RotationCondition (Relative to table2)
RotationCondition (Relative to table)
RotationCondition (Relative to table2)

Figure 5: Condition Evaluation Logic tree grouping as a logic tree

One of the important decisions we had to make when designing the SceneObject was what the default behaviour of evaluation should be: We could either consider the SceneObject satisfied when *at least one* of its conditions were satisfied, or when *all* of its conditions were satisfied. It is a decision with fairly severe consequences and could result in as can be seen in the figure below (fig. 6), so we discussed this for quite a bit. Through testing both by setting up these "evaluation trees" for a few simple and a few more complex scenes we landed on the first of the two options: a SceneObject is in a terminating state if *at least one* of its terminating conditions are satisfied, and it's in a goal state if it's *not* in a terminating state *and at least one* of its goal conditions are satisfied

The diagrams (figs. 5 and 6) are incomplete models made just to show of the difference between the two defaulting modes we could have opted for. The table on which the different SceneObjects are placed on, for example, is a SceneObject left out of the tree, though that would also have a list of conditions on it

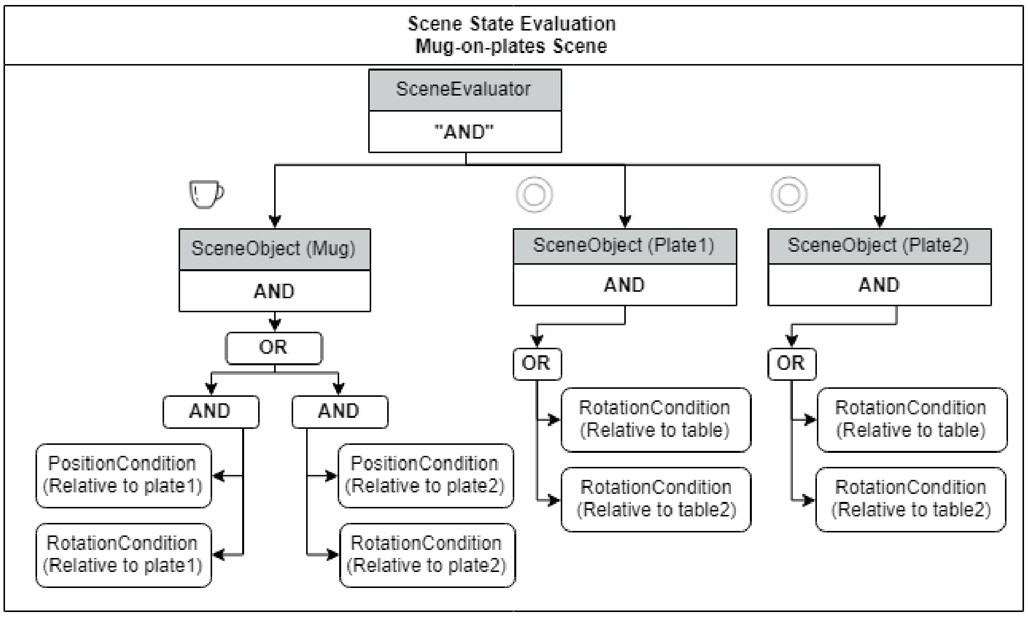

Scene State Evaluation
Mug-on-plates Scene
SceneEvaluator
"AND"
SceneObject (Mug)
AND
OR
AND
AND
PositionCondition (Relative to plate1)
RotationCondition (Relative to plate1)
PositionCondition (Relative to plate2)
RotationCondition (Relative to plate2)
SceneObject (Plate1)
AND
OR
RotationCondition (Relative to table)
RotationCondition (Relative to table2)
SceneObject (Plate2)
AND
OR
RotationCondition (Relative to table)
RotationCondition (Relative to table2)

Figure 6: Condition evaluation tree defaulting SceneObjects' conditions to be evaluted in an AND-manner (all or nothing).

Then again, there are cases where the optimal evaluation tree would be based around AND-defaulting. A thought scenario for this could be a simpler scene than the one above (figs. 5 and 6), where there's only one plate, and only one table.

3.4 Scene State Evaluation

Evaluation of the scene is rooted in the `SceneStateEvaluation` class. It will fetch all the SceneObjects in the scene and initiate the evaluation using the mediator pattern. The mediator pattern is about having an intermediate class that functions as an interface between classes [18]. So all Droid has to evaluate is the `SceneStateEvaluation` which will propagate the evaluation-call through the scene and its SceneObjects.

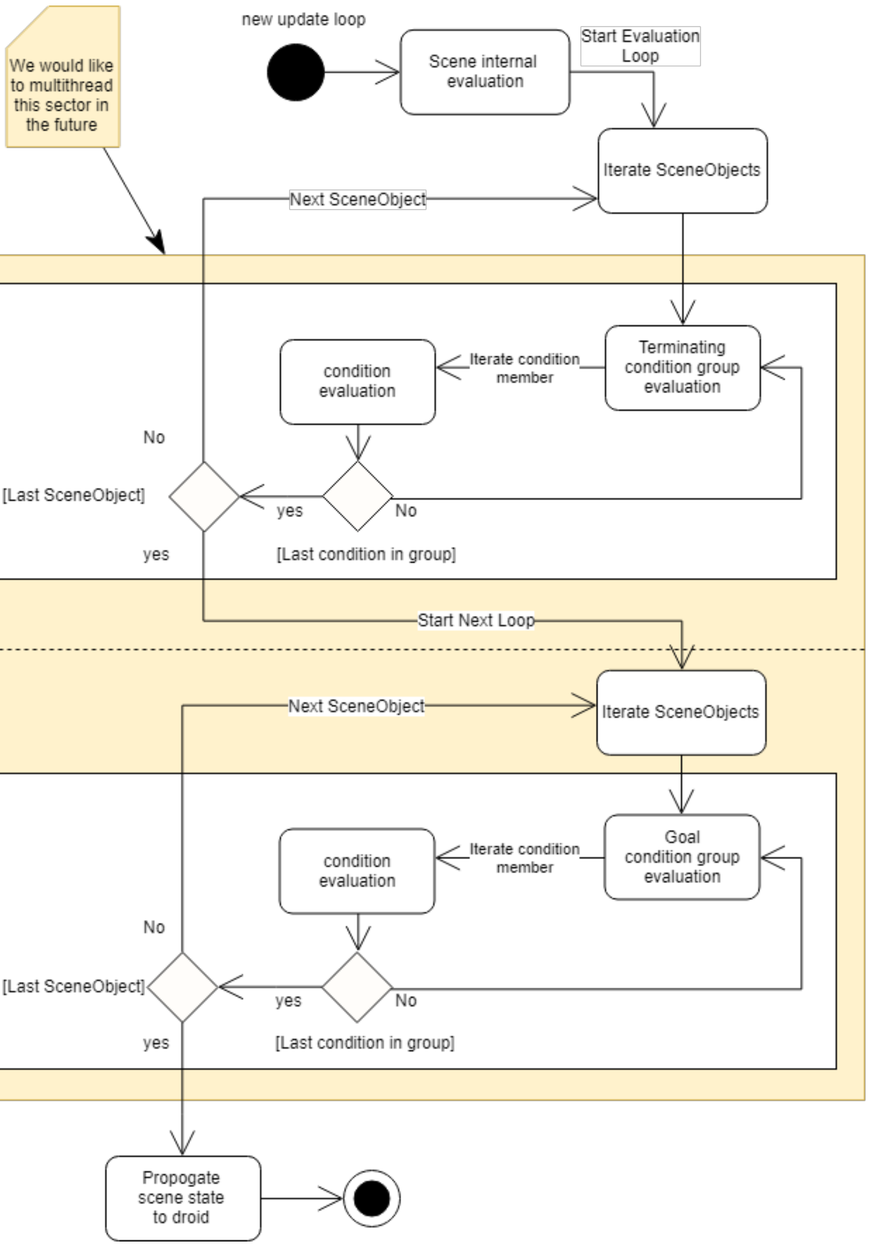


Figure 7: Evaluation of scene represented as an activity diagram.

3.5 Playground Manager

The Playground manager is, as the name suggests, an entity dealing with the business logic in the scene. It contains information about the state of the environment, and all its SceneObjects. It ensures that the scene is set up correctly by fetching the required objects for a Playground environment. During annotation of the SceneObjects, the Playground manager evaluates the scene and displays condition-fulfillment, allowing the trainer to rapidly test their conditions.

The specific design of the Playground manager was not decided early on, instead, it became progressively more clear what it needed to contain and what functionality it needed to have. Now that we know what it needs to be able to do, we have seen a potential way of restructuring it into several components and divide its responsibilities in a more modular way.

3.6 System Serialization and File I/O

When a developer/contributor creates a new condition, they also have to create a condition data container, and the functions required to ensure interchangeability (see fig. 8). This is only if they want to save these conditions after instantiating. The container has a defined method (`GetDataContainer` see section 6.4.1) that can be implemented to convert it into a condition and vice versa (see serialization). The goal of this design is that the generation of these containers could be automated and therefore the environment engineer would not have to think about serialization when implementing new conditions. We have a singleton class that uses containers to save and load to file.

A lot of conditions derive from MonoBehaviour, which means that the data manager's serializer needs surrogates to support these. An environment engineer can also add custom surrogates to the manager if their condition is using Unity classes that we do not surrogate natively.

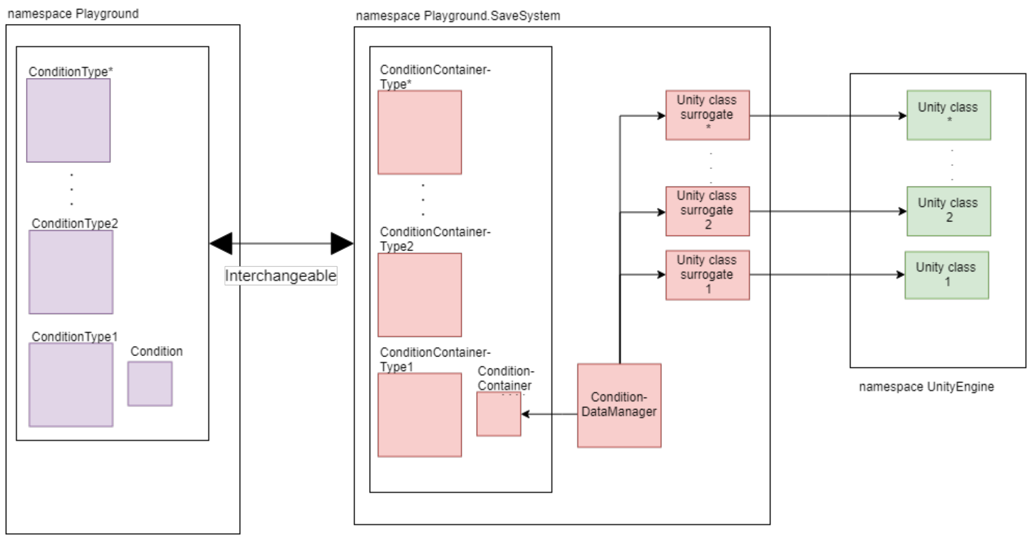


Figure 8: Serialization architecture

3.7 User Interface

3.7.1 Editor and Widget Creation

The underlying structure

of widgets and condition editors is fairly complex, and so we spent quite a while refining and refactoring the design of it throughout the development of Playground. The ultimate design of it is very much an object-oriented one, where the main components all derive from the same base class: `Widget`

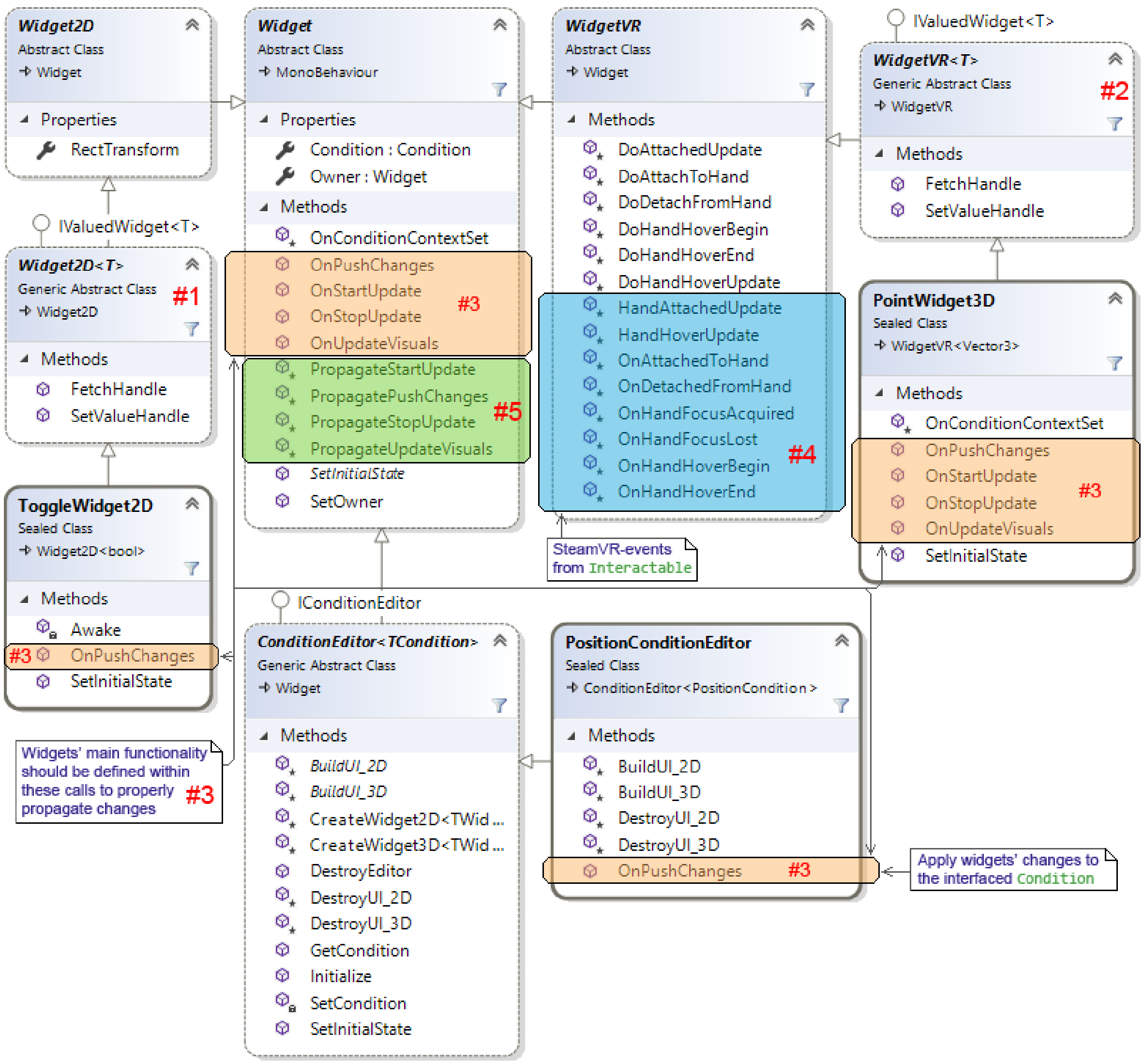


Figure 9: Function overview shown in a class diagram of the fundamental `Widget` classes. The red numbers (#n) are specific points of interest tagged later throughout this section.
In summary they are:

#1 & #2 Value-wrapping Widgets using C#generics.
#3 Widgets' main calls to update values.
#4 Calls to propagate the relevant events.
#5 SteamVR's `Interactable`-events.

When coming up with the concept of condition editors we were inspired by Unity's own way of dealing with (custom) editors, where you can define how everything is presented explicitly. An editor is in essence just an instruction set for what UI elements need to be present, and how they interface the thing being edited.

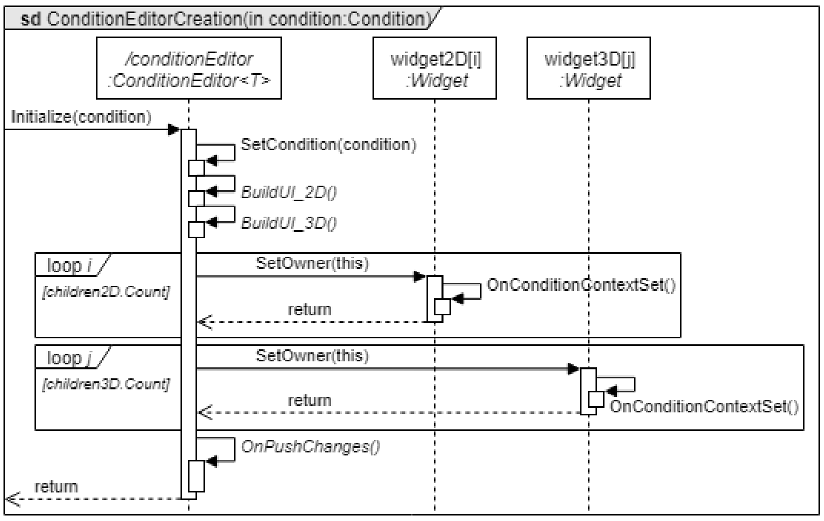
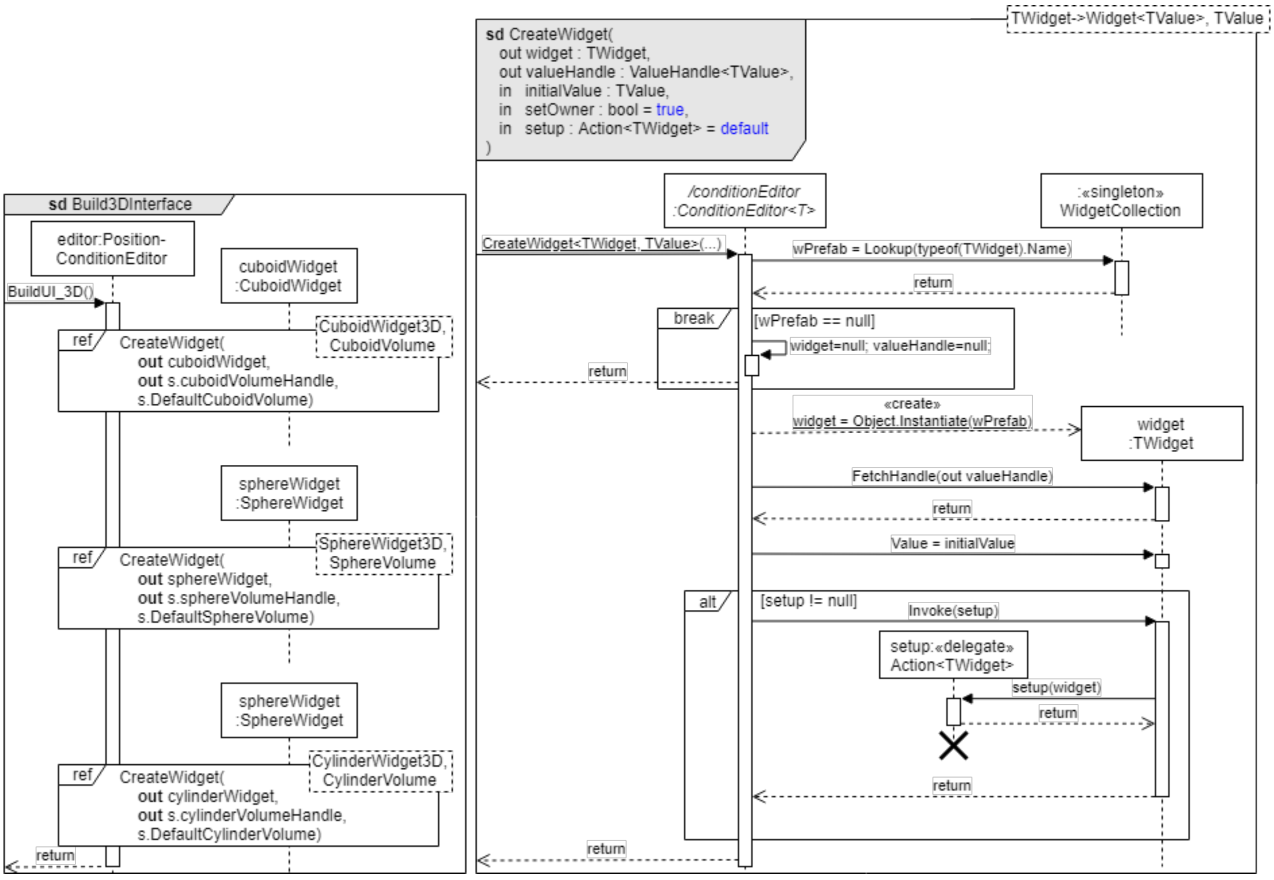


Figure 10: Creation of a condition editor, where `BuildUI_2D()`, and `BuildUI_3D()` are the overridable instruction sets for any condition editor.

Our system is designed to deal with two core types of such elements, namely 3D elements - *3D-widgets*, and 2D elements - *2D-widgets*, where the core difference is in how you interact with them. As illustrated in the sequence diagram fig. 11a, the creation of widgets are fairly straight forward. Given that both a widget type and prefab are defined, an editor will be able to request the needed widgets by its type.


sd Build3DInterface
editor:Position-ConditionEditor
BuildUI_3D()
cuboidWidget :CuboidWidget
CuboidWidget3D, CuboidVolume
ref
CreateWidget(out cuboidWidget, out s.cuboidVolumeHandle, s.DefaultCuboidVolume)
sphereWidget :SphereWidget
SphereWidget3D, SphereVolume
ref
CreateWidget(out sphereWidget, out s.sphereVolumeHandle, s.DefaultSphereVolume)
sphereWidget :SphereWidget
CylinderWidget3D, CylinderVolume
ref
CreateWidget(out cylinderWidget, out s.cylinderVolumeHandle, s.DefaultCylinderVolume)
return
sd CreateWidget(out widget : TWidget, out valueHandle : ValueHandle<TValue>, in initialValue : TValue, in setOwner : bool = true, in setup : Action<TWidget> = default)
TWidget->Widget<TValue>, TValue
/conditionEditor :ConditionEditor<T>
:«singleton» WidgetCollection
CreateWidget<TWidget, TValue>(...)
wPrefab = Lookup(typeof(TWidget).Name)
return
break
[wPrefab == null]
widget=null; valueHandle=null;
return
«create» widget = Object.Instantiate(wPrefab)
widget :TWidget
FetchHandle(out valueHandle)
return
Value = initialValue
alt
[setup != null]
Invoke(setup)
setup:«delegate» Action<TWidget>
setup(widget)
return
return
return

(a) For condition editors, the building of UI (both 3D and 2D) is done through a few calls to the inherited `CreateWidget2D/ CreateWidget3D` functions.

(b) A widget is created by looking for its type name in the *WidgetCollection* (a ScriptableObject acting as a dictionary for widget-prefabs). The found prefab is instantiated, its *ValueHandle* (see section 3.7.3) is obtained, and the rest of the widget's setup is invoked.

Figure 11: Construction of an editor's UI components - Widgets.

3.7.2 Widget Event Propagation

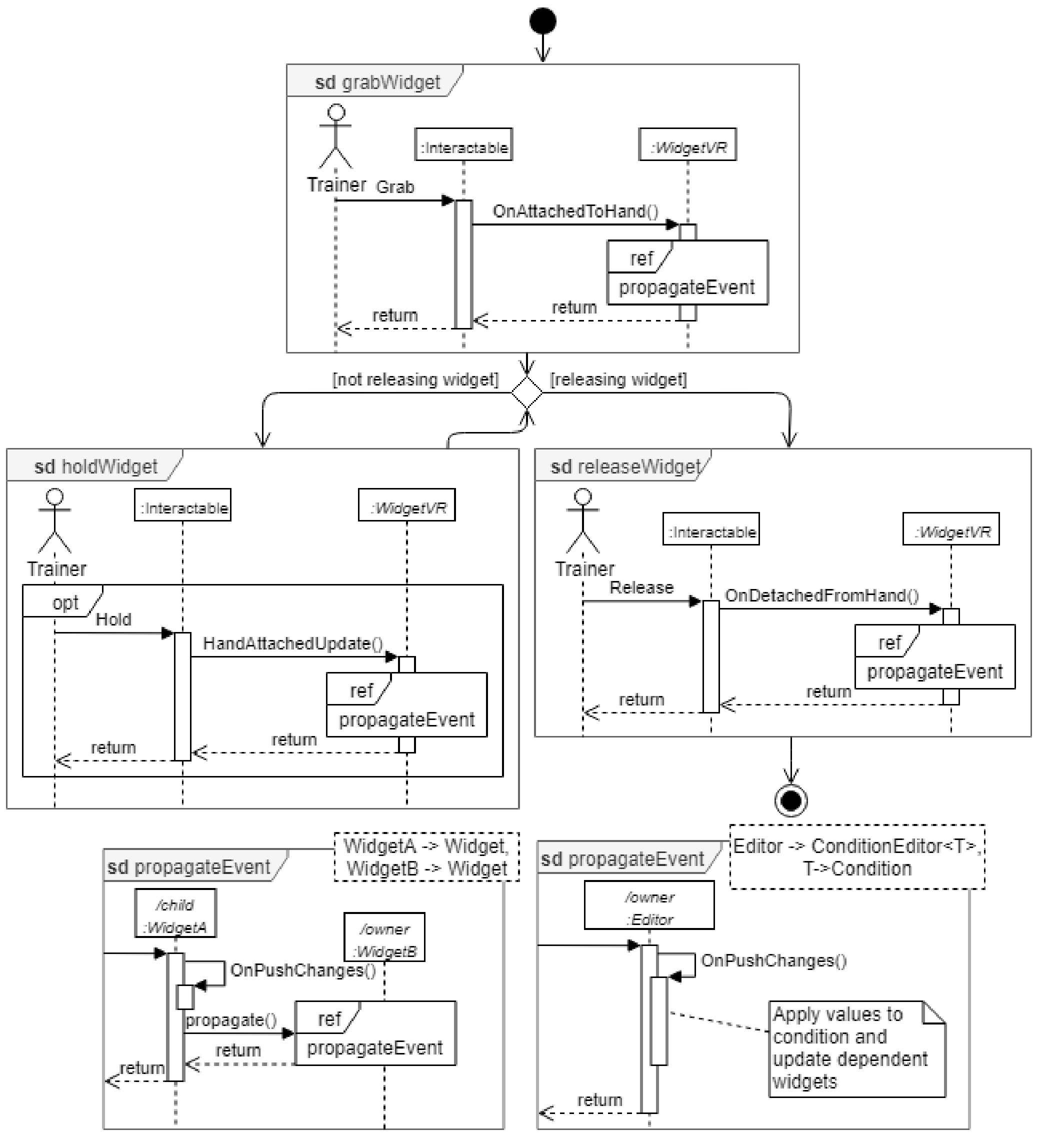


Figure 12: How 3D widgets receive events and propagate them to the editor in order to update the condition. A 3D widgets event propagation is first invoked (fig. 9 #5) by listening to SteamVR's interactable-events (fig. 9 #4)

Widgets are designed as a means to modify a condition's parameters, yet they cannot directly modify the internal values of the condition, as the conditions internal parameters might not be easily interfaceable - a condition may hold one set of parameters that are useful from an evaluation point of view, but completely useless from a GUI-point of view.

Designing a good interface for a condition will in many cases involve translating the parameters of a condition into a different, more user-friendly set of parameters that can more easily be understood and interfaced. An example of this is a *positional volume* (as seen in full code F.1 - PositionCondition), where the evaluation of whether something is inside the volume is most easily done through mathematical formulas, making this how they are defined internally. However, presenting these values to the trainer makes little sense, and does not utilize the third dimension VR grants us for use in UI.

We wanted to present the trainer with an *easy-to-use-* and *easy-to-understand* UI, meaning that the widgets needed to process some user input and convert it to the condition's actual value. Equally, the widget's *value* must be restorable from the condition's internal value in order to maintain persistence. We modeled this value-wrapping using C#-generics (fig. 9 #1 & #2).

3D-widgets work by defining some arbitrary data that should represent the condition. For example, you could draw a vector in world space to represent a max velocity for a velocity condition. This vector should then be able to be scaled by grabbing an edge of the vector. 2D-widgets work a little bit differently as they need to represent a value in a more conventional manner. The velocity data could be represented with the use of three seek-bars (x, y, z) as an example. The seek-bars can then be placed in our condition editor menu as a UI element.

We allow widgets to be children of other widgets as part of our design (i.e. the composite pattern); a pattern we utilize to build complex widgets. To do this we defined a set of overridable functions (fig. 9 #3), and made their execution propagate through the widgets' parents, ultimately ending up at the editor itself, which applies the values of the widgets to the appropriate properties of the condition (as seen in fig. 12).

For 3D-widgets the *triggers* for these functions are the events raised by the SteamVR Interactable upon grab/hold/release (with the VR-Controller) of an object. This is nice from a design point of view, as it means that all 3D-Widgets essentially have the same entry point of execution, making it easier for any external developer to understand how to use the widgets.

2D-widgets are a bit different, in that they don't inherently have a shared trigger for execution like the 3D-Widgets have. This is mainly due to the fact that we use Unity's own solution for 2D UI, where there's no shared interface for all UI elements, as there are several of these that do not need events such as *onPointerPress*, *onPointerHover*, *onPointerUnpress*, etc.

We still wanted a unified way of talking about the propagated events of widgets however, as that has the benefit of being easier to understand from an external point of view. The solution we ended on is to let 2D-widgets themselves hook up to the Unity-UI elements' events during run time as event listeners. This is a solution that works well as it is very explicit about how a 2D-widget's state is updated and propagated.

```
public class DropdownWidget2D : Widget2D<int>
{
    // UnityEngine.UI.Dropdown - the Dropdown Component by Unity
    public Dropdown dropdown;

    private void Awake()
    {
        //inject calls to our interface
        dropdown.onValueChanged.AddListener(_ => PropogatePushChanges());
            //call directly to the base in order to propogate the event
    }
    public override void SetInitialState() => dropdown.value = Value;
    public override void OnPushChanges() => Value = dropdown.value;
}
```

Listing: Dropdown Widget Hooking Itself Up To Unity's Dropdown Component]Dropdown Widget hooking itself up to Unity's Dropdown Component, making the *OnPushChanges*-call propagate up through its parents whenever the dropdown is changed (full code F.7).

3.7.3 Widget ValueHandles

One of the concerns we got while designing widgets was the fact that there might be some values of a condition that would make sense to be presented both in 2D and 3D, or a case where you might want to switch between editing modes; that is, editing the same value of a condition, but through different widgets. In such a case, it would be problematic if all widgets edit their own separate values before they are pushed up to the condition editor to be applied to the condition. In such an event, the editor would have to take preference of one widget over another when applying the widgets' values (because it essentially is trying to change the same condition value). This means that the other widget, in this case, could not actually modify anything, it could only receive updates indirectly from the other widget.

The wanted behaviour is that when moving a point widget (`PointWidget3D`) in VR, the changed values would be reflected in the condition menu (in the `PointWidget2D`).

Our solution to this problem is very much inspired by shared pointers from C++ where two classes might reference the same variable through a shared pointer. There's no such concept as pointers in non-unsafe C#-code. However, there is another concept core to C# we *can* use, which is the fact that objects of classes are copied by reference (meaning that the copied object will be pointing to the object that was copied, rather than being a new object on its own, essentially giving us the same behaviour as from a shared pointer in C++).

```
public class SomeClass {
    public int someInteger = 123;
}
public void SomeFunc() {
    SomeClass a = new SomeClass();
    SomeClass b = a;       // reads as 'b' is 'a', or 'b' points to 'a'
    b.someInteger = 321;
    print(b.someInteger); // prints '321'
    print(a.someInteger); // prints '321' - a & b refer to the same object
}
```

Having C#'s generics, we designed a class that could wrap anything into itself, allowing us to share it between multiple widget instances.

4 User Experience Design

4.1 Tablet Menu

The tablet menu is the user's main way of interacting with the application. The user gets a laser pointer in one hand, and the tablet menu in the other. If the user opens the menu in the left hand, the laser pointer is in the right hand, and vice versa. This way of opening the menu is intuitive for both left- and right-handed users. See how it works in this gif[1].

To mitigate tracking noise and shaky hands, the tablet menu follows the user's hand with a smooth motion that stops within a dead-zone. The dead-zone allows minute movements to be ignored, making it easier to hit buttons accurately with the laser pointer. The behaviour of the tablet menu is demonstrated in this gif[2].

4.1.1 Overview Menu

The overview menu is only available when a SceneObject is selected. It is structured into three lists:

- Goal list: Contains conditions or condition groups, when all these evaluate to true, the object is in a goal state.
- Terminating list: Contains conditions or condition groups, when at least one evaluates to true, the object is in a terminating state, even if all goal conditions also are true.
- A list of all possible conditions. These can be dragged into the goal list or terminating list in order to instantiate them and create a new condition.

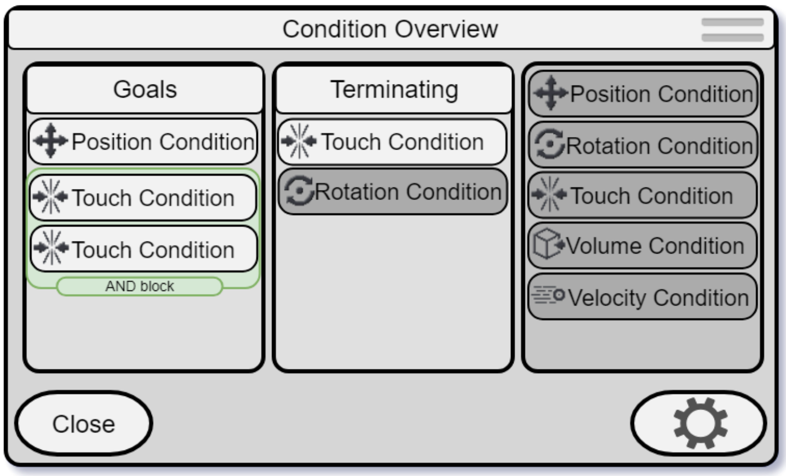


Figure 13: Condition overview menu mock-up

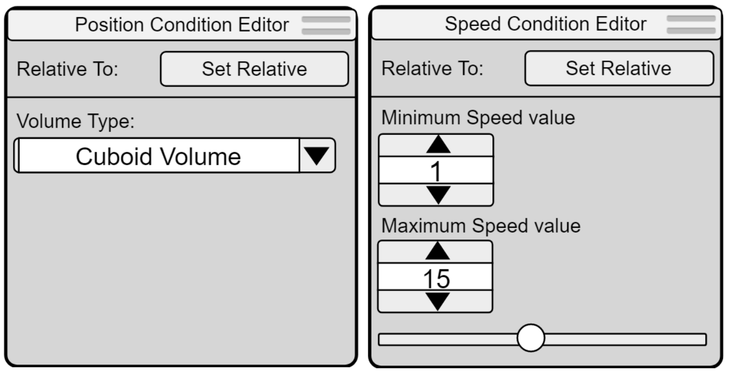
[1] A demonstration of how the tablet menu can open in either hands.
https://justworksltd.gitlab.io/playground-docs/media/Gifs/UI/Toggleable.gif
[2] A demonstration of how the tablet menu follows the player's hand in VR.
https://justworksltd.gitlab.io/playground-docs/media/Gifs/UI/UIFloat.gif

Simplifying UI

While designing the 2D UI for the application, we tried minimizing the number of clicks the user has to make. This is even more important in VR as laser pointer clicking is more tedious than with a mouse. Originally, we had a setup-queue menu that contained all the conditions that needed some setup process to work. We initially saw this as very useful, because we thought the process of setting up conditions would have a very unified flow. However, what the setup-queue actually did, was moving the setup process behind unnecessary clicks and forcing a specific way to instantiate conditions. For these reasons, we decided to discard the setup-queue from the UI design, and go for a more flexible solution where the user could go into any step of the setup process at any time.

4.1.2 Condition Editor Menu

The condition editor menu is only available when a condition is selected within either of the two state lists (fig. 13). The menu contains all the 2D-widgets related to that condition, which are used to change its properties. There will usually just be a few 2D-widgets in the editor menu, as we focus as much as possible on annotatable 3D-widgets. This is a decision we have made, as we want to reduce the number of clicks needed because it is harder and more time consuming to use than an equivalent solution as a 3D-widget.



(a) Position condition editor menu (b) Speed condition editor menu

Figure 14: Mockups of condition editor menus.

4.1.3 Status Menu

The status menu displays a summary of the scene, it displays:

- How many SceneObjects are in a goal state.
- How many SceneObjects are in a terminating state.
- How many SceneObjects there are in total.

The status menu has buttons to interact with the application and perform tasks like:

- Reloading the scene.
- Removing all conditions in the scene.
- Saving the state of the conditions to disk.
- Running setup for AI training.

4.2 2D Widgets

2D-widgets are UI elements used with the laser pointer within the condition editor menu to change the condition-properties. They are very similar to common UI-element abstractions in other frameworks, such as *Views* in Android Studio or *UIControls* in iOS. Examples of properties to change with 2D-widgets are: max speed for a `VelocityCondition` using a seek-bar, or volume type selection for a `PositionCondition` using a drop-down widget. Some of the 2D-widgets we see useful are:

Check-box widget	A button that is either checked or unchecked.
Radio-button widget	A group of buttons where only one can be selected at a time.
Seek-bar widget	A slider between a minimum and a maximum value. (See fig. 14b)
2D-direction Widget	A circle where the user can select a point within this circle.
Drop-down Widget	A menu that expands when clicked to reveal a list of items where one is selected. (See fig. 14a)
Spinner Widget	A text field with an up and a down button that increases or decreases a numerical value in the text field. (See fig. 14b)
Separator Widget	A simple line to separate the editor into parts. (See fig. 14b)
Label Widget	A simple text label. (See fig. 14)

4.3 3D Widgets

3D-widgets are the user's way to define a condition's properties in VR with motion-controllers. When setting up a condition that requires a volume, a 3D-widget will be used to define that volume, as doing this in 2D would be tedious in comparison. For instance, a `PositionCondition` needs to know where the SceneObject in question (the condition's context) is supposed to end up in order for it to be fulfilled. The user is able to change which volume widget they would like to use, move it around, scale it and rotate it in VR. The 3D-widgets are instantiated with a size matching the relative object of the condition it belongs to, meaning a grape will have a small 3D-widget to mark its volume, while a table will have a large one. Some of the 3D-widgets we see useful are:

Point Widget	A single point that can be moved in 3D space to define a position.
Vector Widget	A set of point widgets that define a vector in 3D space.
Scale Widget	A single handle that can be moved along an arbitrary 3D axis relative to its origin to define a numerical value (being the magnitude of the vector to the handle-point).
Cuboid Widget	A set of 6 scale widgets and a grab handle (for moving the entire widget around). Allowing it to be picked up, moved, rotated and scaled in 3-dimensional space to define a cuboid volume.
Sphere Widget	A point widget and a grab handle that can be picked up, moved, rotated and scaled along an arbitrary axis (radius) to define a volume.
Cylinder Widget	A set of 2 scale widgets, a point widget and a grab handle that can be picked up, moved, rotated and scaled along 2 axes to define a volume.

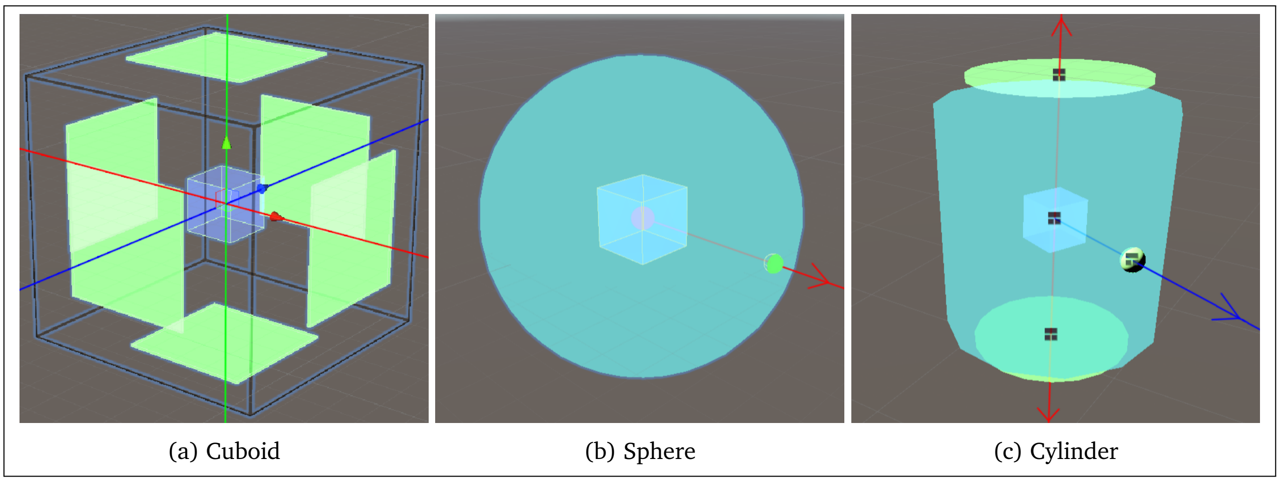
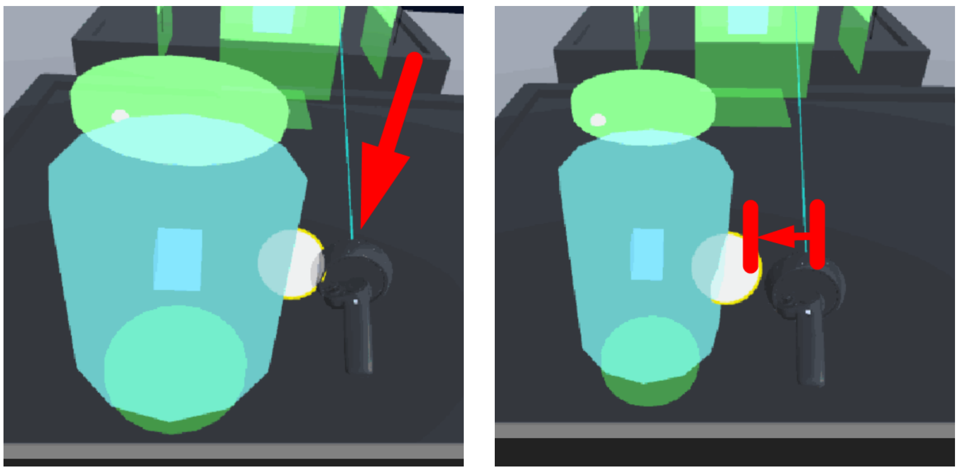

(a) Cuboid (b) Sphere (c) Cylinder

Figure 15: 3D-widgets defining a primitive volume to annotate a goal area or fatal area for a SceneObject.

3D-widgets are moved and rotated by grabbing its center handle (called a *grab handle*), see the blue cube in the center of the widgets (fig. 15). The widget can then be moved around like any interactable object in VR. Most 3D-widgets can also be scaled to better fit the desired area the trainer wishes to mark as a goal or terminating area. This is done by grabbing the radius handle (a point widget, the small white sphere. see fig. 16a) and moving it. This will change the radius of the widget, some of the widgets can also be scaled along an axis. These scale handles are the green handles on the widget (see fig. 16c). To better see how the widget is used, have a look at this short video[3]. It is worth noting that the widget in this short video was not relative to anything, so they were instantiated with quite large size.

[3]A demonstration of how widgets were scaled and moved in VR.
https://justworksltd.gitlab.io/playground-docs/media/Gifs/UI/2019-04-06_20-13-58.mp4


(a) Grab radius handle (b) Move radius handle and release

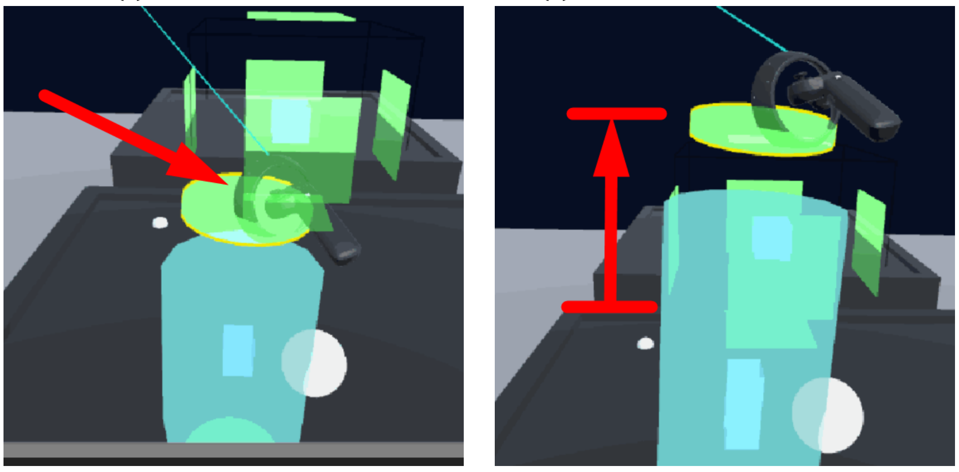

(c) Grab height scale handle (d) Move height handle and release

Figure 16: Scaling of a Cylinder Widget.

5 Development Process

5.1 Technology

5.1.1 Digital Tools

The following tools have been used actively throughout the development of Playground:

Microsoft Visual Studio	Integrated Development Environment (IDE).
Unity	Component-based 3D game engine [19].
Git	A free and open source version control system [20].
GitLab	Version control, Scrum-board for assignment of work tasks, and CI [21].
ReSharper	Command-Line Interface (CLI) for linting and inspecting code quality before pushing to remote [22].
Blender	3D modeling tool [23].
Toggl	Time tracking tool [24].

5.1.2 Programming Languages

C# is the main language we program in, this is because C# is the Unity standard, and because it is the language that gets the most support from the developers and users of Unity.

Python is also a candidate for certain parts of the project that involve interacting with the machine learning module (Agent) or the communication link (Neo) between the machine learning algorithms and Unity. The reason for this is that these modules are all written in Python, besides Python is recognized as one of the best languages to work in when it comes to AI.

ShaderLab & CG are also used to write shaders to visualize highlighted objects in VR.

Bash is used to implement the Continuous Integration (CI) parts of our development process.

5.2 Project Organization

5.2.1 Scrum

The development model we decided on for the project was Scrum. This is the model we were most familiar with, and is also commonly used in the industry. Since all the design and structure had not been decided when the development started, an agile method was the best option. With the project having a hard deadline in only a few months, dividing the total time into sprints of one week and setting milestones was a good plan for the project. The scrum board was also heavily in use, with labels for all main categories of tasks.

5.2.2 Work Boundaries

For the internal organization of the group we had compulsory scrum meetings once a week where we had discussions of work done in between meetings.

Group Policies

§1 Money and expenses

1. For expenses that exceed what is covered by the client and NTNU IDI, the expenditure(s) is shared evenly between the group members, unless otherwise agreed between the parties involved.

§2 Illness

1. If you become too ill for attendance during the project, you must communicate this to the other group members.

§3 Scrum

1. It is up to each member to report a lack of sprint goals if one completes all the sprint goals they were given at the beginning of the sprint period.
2. It is up to each member to report having too much work assigned them as early as possible, if it is obvious that it will not be completed, the other members should be informed.
3. If the scope becomes so large that it slows down the project's progress due to dependencies in certain modules, there should be a group meeting on how this should be solved, this should also be with the supervisor.

§4 Meeting

1. In case of any disagreements in regulations/polls, the group leader has two votes, so that we can always end up with a majority since our group consists of 4 members.

§5 Digital tools

Version control:

1. There are several branches:
   - **master** is the most stable and verified.
   - **dev** is the head of development. This is where general progress is pushed to. The dev branch is merged to master after verifying that the system is in a stable state.
   - **feature branches** are dedicated to developing one specific set of features that would create friction for other developers if pushed to dev.
2. All Git commit messages must follow a standard to clarify the changes that have been made. Each commit should only do one thing:
   1. Each commit message should start with a verb ("refactored", "removed", "fixed", etc) to specify what was done to the object of the message. Otherwise "added" is assumed. (e.g. "Refactored Script.cs" or "Widget.prefab").

2. Each commit that has a correlation to an issue must reference the issue with its number. (e.g. "Fixed #104: Collider getting stuck.")

Communication platform:

Communication channel is Discord using our server. Messages should be posted in the relevant channel.

§6 Documentation

1. Each class and function must be documented using C# XML Docs. [25]

5.2.3 Roles

- **Contracting entity:** SINTEF Ocean SeaLab
  (John Reidar Mathiassen, Jonathan Sjølund Dyrstad, Christian Heider Nielsen)
- **Project leader:** Halvor Bakken Smedås
- **Scrum master:** Halvor Bakken Smedås
- **Log writer:** Jone Martin Skaara
- **Thesis supervisor:** Mariusz Nowostawski

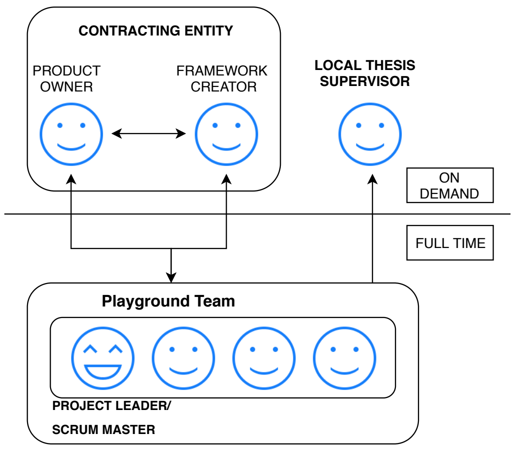


Figure 17: An overview of the involved parties in the development of Neodroid Playground.

5.2.4 Routines and Rules in the Group

- The agreed workload is 30 hours per week per group member (approximately 5 hours each weekday).
- We have a weekly sprint meeting on the first day of a sprint. In the meeting we review the previous sprint, verify issues in the scrum-board, and move new issues into the sprint backlog.
- After a meeting, we write a short summary of the topics that were discussed so we can look back on details, and reflect on decisions that we make.
- Whenever an important implementation decision is made, it is documented so we can reflect on them later and get a good overview of the progress of the project.

5.3 Test-Driven Development

Test-Driven Development (TDD) is something we wanted to apply to this project. The goal of test-driven is to ensure functionality in the program, but also create specification for it. We realized that using TDD would come at a cost of initial time use. One can argue that the time cost is repaid as bugs are detected early, and will be beneficial to the project over time, especially during the iterations of the architecture. We decided to use TDD for our condition components as they were very important for the project, and very testable by their boolean nature. This way we could ensure condition performance and still avoid TDD taking too much focus away from the project.

5.4 Continuous Integration

We were intrigued to try setting up CI for our project as we saw a potential gain in having a few different integration-processes running remotely while we kept working with the code. The idea initially was to get a container up and running with Unity to compile our project and run all the unit tests. This would be useful as we could set up an automatic Git-mechanism remotely, where we would be able to push to an "untested" branch, then leave the remote process to run all the unit tests. If successful, it would push the work to a "tested" branch. This proved to be problematic as the CI-integration on the local GitLab instance was not configured to have the needed Docker images to run Unity, nor were there any available Docker images for the latest version of Unity. This made building remotely impossible in the first place, as we (and the Neodroid platform as a whole) utilize bleeding edge technologies from the Unity feature stack.

Branching was another concept we did not find necessary in the early stages of the project. We did not meet any merge conflicts that gave us issues as we had individual Unity scenes and generally avoided working on the same scripts. When working on larger features such as serialization and converting conditions to MonoBehaviour we used branches to avoid breaking our master.

As the project grew in size and complexity the need for generated documentation grew with it. We ended up using GitLab to run CI processes on code from our *prod3* (NTNU Gjøvik's local GitLab) repository. We then used GitLab's deployed pipeline to generate documentation and host it using doxygen.

5.4.1 Git Hooks

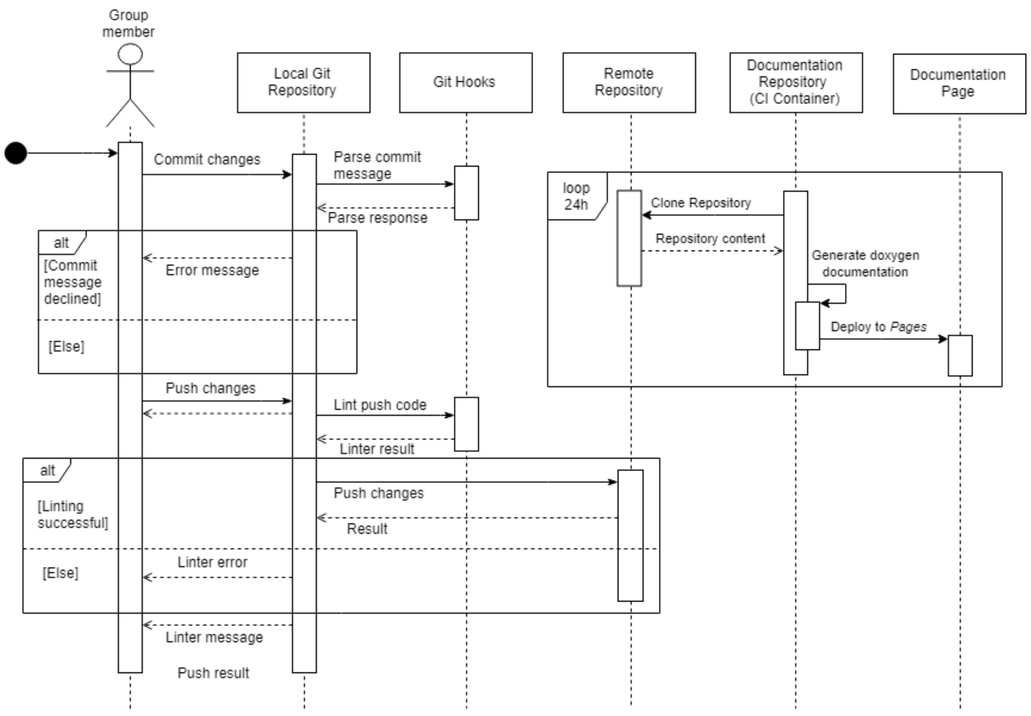


Figure 18: Sequence diagram describing how we applied git hooks and CI to our development process.

In order to enforce a specific style of code, we introduced *ReSharper CLI InspectCode* [22] as part of our commit- and push-processes by using Git hooks. With it, our code is linted pre-push and a log of all bad practices in the commited work is printed, enforcing us to correct them before actually pushing to the remote. Initially this was a bit of a hurdle to get over, as we needed to get a few settings set up correctly using *editorconfig* and *DotSettings*. Ironically enough, the documentation for doing so was a bit lacking, but when all was set up correctly, it turned out to be a very valuable step in the process to ensure good code quality and consistent coding style between the members of the group.

5.4.2 Doxygen Documentation

We revisited CI again when we realized that we could utilize it to automatically generate reference documentation by having it run doxygen on our project files. This has since been very useful by giving us an overview of the current state of documentation, and to see where we are lacking documentation. Documentation is, after all, very important in our project, as we are essentially building a framework for others to later utilize and extend upon.

Seeing as we already had committed to working on the local GitLab instance, we actually have an unusual approach to CI, where we have the two repositories: One with our code, and one just containing the CI-instructions (listing 5.1) which was hosted on the official GitLab instance, and was configured to have the pipeline running on a schedule, once every day.

We later realized that we could host media files on the documentation pages. This enabled us to reference and use these in our documentation on certain parts where we felt graphical and practical explanations were required.

```
image: alpine:latest

before_script:
- apk update
- apk add git
- apk add doxygen
- apk add ttf-freefont graphviz

pages:
  script:
  - git clone "http://prod3.imt.hig.no/justworks/playground.git"
  - cd playground
  - most_recent_branch=$(git for-each-ref --sort -committerdate
      refs/remotes | awk -F'/' 'NR==1{print $4}')
  - git checkout $most_recent_branch
  - cd ..
  - '(cat Doxyfile; echo "PROJECT_BRIEF=Generated of a WIP branch:
      $most_recent_branch") | doxygen -'
  - mv ./docs/html/* ./public/
  artifacts:
    paths:
    - public
  only:
  - master
```

Listing 5.1: GitLab CI configuration for documentation generation.

In our configuration file we opted for using the smallest *Linux* image we found, as we did not require many features from the running container, additionally there is a benefit of shorter setup-time with smaller images. We obtain Git and doxygen as those are the two essentials for obtaining the remote repository files and generating the documentation. We obtain *graphviz*, which is used by Doxygen to generate graphs for the documented code.

As discussed previously, we are using separate repositories for our CI and our code, as we could not set up CI on the local GitLab-instance. We found that this configuration was nice to use as it does not clutter up the folder structure of the project (i.e. the code repository), but rather kept them separate.

We clone the code repository in order to let Doxygen iterate over it, look for the branch most recently pushed to (for development reasons, we thought it most useful to see the latest state of documentation), to clarify this we also append a *project brief* to the *Doxyfile* making the header of the documentation page show which branch the documentation is generated from. The end result is that we have Doxygen generated documentation hosted on the official gitlab domain. Please see footnote [1] for an example of our documentation. Do note some of our classes are lacking in documentation and specifications on the hosted page at the time of this writing.

[1] Playground documentation of the condition editor
https://justworksltd.gitlab.io/playground-docs/class_playground_1_1_u_i_1_1_condition_editor.html

6 Implementation

6.1 Conditions

An AI learns a task within an *environment*. As such, it should hold restrictions and goals that can be evaluated and propagated to Neo in the form of positive and negative *signals* passed to a training algorithm. Conditions are abstractions of these goals and restrictions.

A condition always has a context, which is the SceneObject it is attached to. Most conditions also have a concept of a relative component which they will use differently depending on the nature of the condition. An example of a condition with a relative is the `PositionCondition` (see full code F.1), which has a `Transform`-component as its relative. This allows us to describe a position relative to another transform. Another example is the `TouchCondition`, which has a `Collider`-component as its relative. As such, the relative component's GameObject is the one that needs to be touched in order to satisfy the condition.

There might exist conditions that are not dependent on a relation with something else in the scene, but rather on some internal state of the condition's context itself. Such conditions could be called absolute state-conditions, as they do not depend on anything but their own state. Some theoretical examples of this would be `IsOnFireCondition`, `IsAliveCondition`, `IsOldCondition`, all of which we deliberately named just as you would with boolean variables and predicate functions, precisely because these conditions would either be wrappers of booleans or of simple boolean expressions.

6.1.1 The Problem

It can become intricate and difficult for a user to define complex logical structures without any prior experience with boolean logic. Conditions had to be simple in order to not confuse the user. Our goal was to project natural language onto goals and restrictions. Additionally, to combine the two so that the user only had to learn one system.

6.1.2 Evaluation

Each condition is self-contained in its state-evaluation, making conditions modular. Each condition type implements their own unique evaluation function. The evaluation is one of the two main differences between different types of conditions, the other being how (or if) it uses its relative. For example:

```
public override bool Evaluate()
{
    var evaluation = _volume.InsideVolume(Context.transform.position);
    return evaluation;
}
```

Listing 6.1: PositionCondition's evaluation of state (see full code F.1).

```
public override bool Evaluate()
{
    return _isColliding;
}
```

Listing 6.2: TouchCondition's evaluation of state (see full code F.2).

There are two types of evaluation functions: Active and passive. In the case of the `PositionCondition`, the evaluation-function (listing 6.1) is *active* in the sense that it continuously updates its own state. In contrast, the `TouchCondition`'s evaluation-function (see listing 6.2) is a *passive* one as its state is updated externally by others, while the evaluation-function only returns the currently stored state. In the case of `TouchCondition`, its state is updated by collision events in Unity.

The implementation of the evaluation-functions of conditions can deviate a lot from each other, although the pattern stays the same. This means that developers have familiarity when creating new conditions, but still have flexibility in what they can do with them.

To create your own condition you either make your condition a subclass of `Condition` or of `Condition<TRelative>`. You then define variables needed to define the condition. After this is done, the logic of it all is put into the overridden `Evaluate`-function.

In order to use the condition though, you will also need to define how to interact with it in VR, which is a separate issue entirely (discussed in section 6.3.3).

6.1.3 Condition Relatives

Some conditions are more complex than others and many will depend on something else in the scene. Therefore we wanted to allow conditions to have a relation with anything in the scene, and so we opted for using C#-generics in one of the condition base classes to defer this relation:

```
public abstract class Condition<TRelative> : Condition,
    Internal.IHasRelative where TRelative : Component
```

Listing 6.3: Foundation of conditions with a relative.

Doing this essentially generalizes the structure in a way that allows each condition to have a relative of an arbitrary component-type, while still leaving the implementation of evaluation up to the individual condition.

An example of one such condition is the `PositionCondition`. It holds a `Transform` as its `TRelative`, allowing it to access said transform in its evaluation function. The reason for this is that a position condition should be able to change based on the position of another SceneObject: If you want a cup to be placed on a table, then you want the condition to follow the table. You want it to be relative to the table, hence the name. It is up to each condition derived from `Condition<TRelative>` to define what the relation to their `TRelative` means by utilizing the relative's properties in the evaluation-function.

6.1.4 Grouping of Conditions

To achieve a verbose logical structure, you need groupings of conditions into *AND-groups*, *OR-groups*, *XOR-groups* and so on. condition groups themselves are conditions in our system. We have a base group class that derives from the base class `Condition`. The group class manages a list of conditions, which it uses to evaluate its own state. Let us use the *AND-group* as an example:

```
public override bool Evaluate()
{
// The base class 'ConditionGroup' implements IEnumerable<Condition>,
// allowing us to loop on 'this',i.e. the content of the underlying list
    foreach (Condition condition in this)
    {
    // if we hit an unsatisfied condition, the AND-group by definition
    // cannot be satisfied, do an early return for performance reasons.
        if (!condition.Evaluate()) return false;
    }
    return true;
}
```

Listing 6.4: AND-condition group evaluation function.

As seen in this example, just like in boolean logic, our *AND-group* ensures that there are no member conditions that evaluate to false before returning true.

6.2 Playground Manager

The Playground manager is a singleton that handles state in the scene, presence of important objects and some central tasks like reloading the scene.

6.2.1 The Problem

In a scene, there is no guaranty that all necessary components are present. The environment engineer is required to populate the scene with a handful of objects that perform critical tasks:

- VR player: the trainer's presence in the world.
- Tablet menu and its corresponding laser pointer: the trainer's means of interacting with conditions.
- Actor: the AI's means of interacting with the scene. Its reactions are performed by the actor.

If any of these are missing, then the scene will not work as intended. It can be hard for the environment engineer that builds the scene to manage these if there is no feedback from the application.

6.2.2 Central Tasks

Evaluate the Scene

The Playground manager keeps lists of different kinds of GameObjects, one of which is a list of all SceneObjects which it uses to update visual feedback regarding their state.

Setup

On start up the Playground manager will search for all key components. If some are missing, it will try to instantiate them. It can fail if it was not configured with prefab references to the object it is trying to instantiate. In that situation it will be unable to instantiate them, and will instead print errors to let the environment engineer know what is missing so they can add these references to the Playground manager, or add the prefabs manually to the scene.

Another situation where this becomes relevant is the reloading of scenes. The scene that is being loaded does not necessarily have all the relevant components, or it may have them already. Duplicates of unique components can occur, in that case, only one will remain, the others are destroyed.

Switching States

The application needed to switch between states, going between annotating a task and running AI training. The Playground manager switches between states by enabling and disabling objects in its lists.

6.3 The Player and UI

The player had no way of interacting with conditions or the application as a whole. We were limited to SteamVR's functionality like teleporting and picking up SceneObjects. In order to set up a scene that is ready for AI training, the player needs to be able to select SceneObjects, add a condition to it, specify which list it should belong to, group it with others in condition groups and modify its values, as well as the rest of the use cases in the use case diagram (see fig. 1).

6.3.1 Sub-Menus

When we initially developed our menu system, we wanted menus to be very flexible in the way they were created and interacted with. We came up with a system which we call Sub-Menus. Essentially sub-menus can have a hierarchy of child sub-menus that can be moved around and interacted with using SteamVR. Trainer can change and move sub-menus to other locations in the virtual environment to fit their workflow better. In practice, this means that the trainer can change the hierarchy of menus (see full code F.8). The current menus do not utilize the strengths of the sub-menus to the best extent possible, as we needed the base functionality of the menus in place before focusing on usability.

6.3.2 Floating UI and Laser Pointers

Tablet Menu

The user interface in VR is moved to be positioned above the trainer's hand using spherical LERP (SLERP) for rotation. The menu also has a dead-zone that causes minute movements to be ignored, making it easier to interact with 2D-widgets using the laser pointer.

```
if ((target.position - transform.position).magnitude > _posAlignmentDeadzone)
{
    Vector3 vec = target.position
                + target.TransformVector(_initialPosition)
                - transform.position;

    transform.position += (vec - vec.normalized * _posAlignmentDeadzone)
                        * _posAlignmentSpeed
                        * Time.deltaTime;
}

//Rotation, slerping if outside deadzone:
float t = Mathf.Abs(Quaternion.Dot(transform.rotation, target.rotation *
     _initialRotation));

Quaternion to = target.rotation * _initialRotation;
Quaternion from = transform.rotation;

Quaternion deltaTarget = Quaternion.RotateTowards(
                             to, from, _angAlignmentDeadzone
                         );

transform.rotation = Quaternion.Slerp(
                         from, deltaRotation, t * _angAlignmentSpeed
                     );
```

Listing 6.5: Menu alignment. The menu / laser pointer aligns with one motion controller each.

The player can also choose which hand they want the menu in by pressing the menu button on their preferred hand.

Laser Pointer and Ray casting Input Module

The laser pointer uses the same script as the tablet menu to follow the player's hand, although without dead-zones and with much higher speed values. It is using both physics ray casting and graphics ray casting. Physics ray casting is used to select SceneObjects, and graphics ray casting is used to interact with UI elements/2D-widgets on canvases like the tablet menu.

Our menu system is based upon Unity's own menu system. We need to emulate a mouse cursor on the canvas by translating the laser pointer's `RaycastHit` into a canvas-space position, and call events on the UI-elements under said cursor as the default mouse input in Unity does (see full code F.9). To know our position in the menu we need to do a graphics ray cast and that requires a camera [26]. The camera admittedly could create overhead, but it's disabled and is only used to perform a graphics ray cast. We also use a low field of view and a sum of 1 pixel on the camera so even if it did render most of the overhead would come from the extra draw-call.

6.3.3 Widget

Most widgets are generic classes in our system. A member of those is the `ValueHandle<T>` which has the same generic specifier as the widget, as this makes it easier to set up an editor with many widgets that operate on different condition properties (which most often is the case). If an editor needs two or more widgets to operate on the same values of a condition, they should do this by using these functions accessible through the widgets:

- `FetchHandle(out ValueHandle<T> handle)`
- `SetValueHandle(ValueHandle<T> handle)`

```
[ConditionEditor(typeof(SomeCondition))]
public sealed class SomeConditionEditor : ConditionEditor<SomeCondition>
{
    //3D Widgets
    SomeWidget3D         _someWidget;
    SomeOtherWidget3D    _someOtherWidget;
    ValueHandle<float>   _someFloatHandle;

    protected override void BuildUI_3D()
    {
        CreateWidget3D(
            widget:          out _someWidget,
            valueHandle:     out _someFloatHandle,
            initialValue:    Condition.theFloatTheConditionDependsOn
        );
        CreateWidget3D(
            widget:          out _someOtherWidget,
            valueHandle:     out var throwAwayHandle,
            initialValue:    Condition.theFloatTheConditionDependsOn
        );
        _someOtherWidget.SetValueHandle(_someFloatHandle);
    }

    public override void OnPushChanges()
    {
        Condition.theFloatTheConditionDependsOn = _someFloatHandle.Value;
    }
}
```

6.3.4 Condition Editor Menu Generation

The condition menu is generated by each condition. A condition implements an abstract function - `BuildUI_2D()`, which tells the editor to create all the necessary 2D-widgets.

```
ValueHandle<int> _dropdownHandle;
DropdownWidget2D _volumeModeDropdown;

\\ ... some other code ...

public override void BuildUI_2D()
{
    CreateWidget2D(
        widget:        out _volumeModeDropdown,
        valueHandle:   out _dropdownHandle,
        initialValue:  (int)Condition.volumeMode,
        setup:         w => w.SetOptions(Condition.volumeMode)
    );
}
```

Listing 6.6: An example of condition editor menu generation. This is the position condition, which only has a single drop-down 2D-widget in its editor, controlling the type of volume the position condition should use (see fig. 11).

This way of generating the editor panel makes it easy to create new conditions in respects to UI.

A downside is that the editor menu will simply put the UI elements downward, one after the other. More complex structures could be introduced by implementing a *horizontal grouping widget* (similar to horizontal layout groups in other UI-frameworks). This grouping widget would hold an arbitrary number of widgets, and divide the width of the editor among the widgets it contains. This would be very useful in the case of labeling a seek-bar, where you would want both to be on the same line with an uneven division of the editor's width. A solution like this would allow a more natural UI layout with better utilization of the space to fit more widgets on the screen than it could otherwise.

Like every GameObject in Unity, our menu system uses the prototype pattern for instantiating. We use a template menu (Unity prefab) to instantiate the menu in the scene on demand. The prototype pattern is an alternative to the factory pattern and offers a more lightweight solution to instantiating an object [27] (it also avoids complex polymorphic structures which can become an issue with the factory pattern).

The current 3D-widgets (see fig. 15) should suffice in most cases. However, the environment engineer can at any time create their own if need be. The documentation and existing code should be enough to learn how to develop custom widgets.

6.4 System Serialization and File I/O

Serialization is important in many aspects of software. We need it specifically to save SceneObjects to file. The project has some unique problems when it comes to serialization. Normally when you have a `UnityEngine.MonoBehaviour` you can let Unity serialize the class for you, and the object will be stored in the scene file. The problem is that SceneObjects are instantiated during run time and will therefore not be saved to the scene and will also be destroyed when exiting run time. `MonoBehaviour` objects are also not serializable with for example .NET serialization using the `ISerializeableSurrogate` [28] when using multi-inheritance. The reason being that surrogate selectors do not support inheritance in the sense of surrogating base classes (like `MonoBehaviour`) of a class as the serializer is unable to select the proper surrogate in generics.

6.4.1 Condition Container

It became clear we needed a custom surrogate type; the `ConditionContainer` filled this need. Since the container only holds the unique members of each condition and does not have a rooted inheritance of `MonoBehaviour`, we could serialize these containers and instantiate conditions when loading based on the container data. This solution was deemed a "necessary evil" to get serialization working. Each condition needs to implement a `GetDataContainer`-function which a later component can use to serialize the data.

```
public override ConditionContainer GetDataContainer( ConditionValue
    conditionValue , bool inNestedGroupParam )
{
    var container = base.GetDataContainer(conditionValue ,
        inNestedGroupParam);

    var pcc = container.CopyBasicValues<PositionConditionContainer>();
    pcc.volumeBase = _volume;
    pcc.volumeMode = volumeMode;

    return pcc;
}
```

Listing 6.7: PositionCondition container creation (see full code F.1).

We tried to make the implementation of new `GetDataContainer`-functions as easy as possible for the environment engineers. For this, it is useful to utilize the object oriented nature of `ConditionContainer` by calling the base class' implementation of the `GetDataContainer`-function, so that the base class fields are serialized. Remaining fields can often be serialized using functions like `CopyBasicValues<TContainer>()`.

To summarize: when creating a new condition, a `GetDataContainer`-function needs to be implemented. When implementing it, you should probably call `GetDataContainer` on the base class to get a basic container. After this is done, you can get all the base values set with `CopyBasicValues<TContainer>()` where `TContainer` is the container type you have created to mirror your new condition's serializable values. When this is done you can simply copy your custom values into your custom container. We hope to generate these containers in the future so that an environment engineer does not have to think about this.

6.4.2 Condition Data Manager

The `ConditionDataManager` is a singleton that handles saving and loading containers while maintaining the nested structure of the condition groups. As of now, it only supports byte data storage, but with the container system implemented, it should be possible to use other forms of data storage as well, such as pure text-*yaml*.

The idea is that the `ConditionDataManager` should be easy to modify, so that we could deprecate the container solution in the future, in favor of a better one.

To serialize a condition, it needs to create a custom container type which gets serialized by the data manager. When loading, the data manager needs to load custom containers and restore their original condition's data; like location in the scene and member variables.

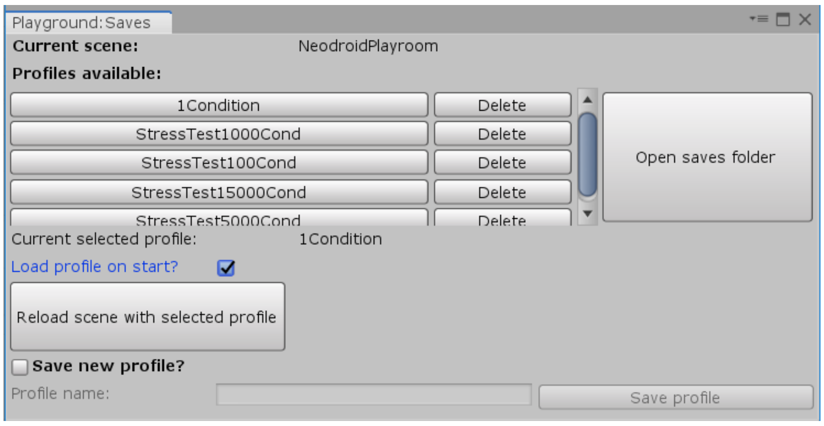
Just like `MonoBehaviour`, any other `UnityEngine`-namespace type does not support `System`-namespace serialization. However, since these types are members of our classes they need to be serialized in some way. The containers could contain extracted values of Unity-types (for example 3 floats instead of a `Vector3`), but a more elegant solution is implementing `ISerializeableSurrogate` for each Unity-type like `Vector3`. We have some of these types wrapped as such surrogates and they are used in the data manager, but an environment engineer can add new ones by implementing the interface for the wanted type and use a function called `AddSurrogate`.

The manager also utilizes a simple ID-system. When a Condition or SceneObject is created it requests a unique ID from the manager which will return said ID. This logic is in the most basic condition type and the environment engineer do not need to think about this. This ID is in turn used by the manager to identify its location in the scene and grouping hierarchy. This means that groups are stored as a list of IDs. Conditions that are part of nested groups become tagged as such.

When the manager saves, it saves the members before the group. When the manager loads, it stores all the nested conditions in their container form. When a group is loaded, the manager finds these unloaded containers and converts them to actual conditions in the group.

6.4.3 Saves Editor Window

We found it useful to have an editor window for interacting with the saved profiles without needing to go into VR. This tool makes it less tedious to load and save profiles, especially if the goal is to run machine learning on a previously annotated set of conditions. Potentially, one could create editors for instantiating conditions in Unity's editor instead of in VR, making this tool even more useful.



Playground: Saves

Current scene: NeodroidPlayroom

Profiles available:

1Condition	Delete
StressTest1000Cond	Delete
StressTest100Cond	Delete
StressTest15000Cond	Delete
StressTest5000Cond	Delete

Current selected profile: 1Condition

Load profile on start? ✓

☐ **Save new profile?**

Profile name:

Figure 19: How the saves window looks in Unity's editor.

The window made it easy to hot-swap and manage a scene's profiles. In VR, the trainer will, for the most part, just save the state into a new profile and occasionally load existing profiles. Managing the saves is a process that is simpler to do with mouse and keyboard, which led us to the decision of making this editor window.

6.5 Collision Detection and Physics Simulation

6.5.1 The Problem

Colliders

The colliders for mesh objects need to be easy to pick up and act as similar as possible to how they would behave in real life. Unity physics have some limitations when it comes to colliders and physics that operate on them. Concave colliders were not supported, so convex colliders are the only option for mesh colliders. In other words, mesh colliders cannot have holes.

This is problematic in our torus-on-pole-example, as shown in the animated gif.[1] The

[1]Problems with convex colliders in Unity.

objects would not behave realistically, and the AI would not be able to learn from this flawed data. There are workarounds like, dividing the object up into several colliders, however, this is very time consuming for the environment engineer who will be building the scene.

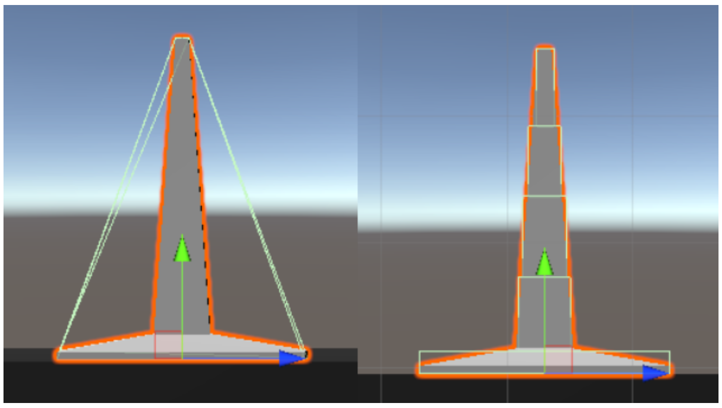


Figure 20: Convex collider vs manually built collider

Friction

The grabbing-physics is very important to the goal of the project, and a critical part of the implementation. Since the focus is to have the grabbing work as realistically as possible, using object parenting is out of the question as friction is effectively infinite when parenting two objects together. The other issue with parenting using Unity-physics is that moving something by setting its transform's position, is that friction is not calculated on the objects, as seen in the animated gif.[2] This was a huge problem because a part of the task was to create a robot hand that would pick up objects using a motion-controller in VR. Unfortunately, SteamVR's `Interactable` works by setting the held transform's position. Setting a transform's position directly also meant that the object that is held would pass through solid surfaces, which was also a major flaw with parenting that needed to be solved because of the requirement for realism.

6.5.2 SAColliderBuilder

Our problem was that colliders were too different from the mesh of the object. SAColliderBuilder solved this problem for us in most cases. SAColliderBuilder is a tool to generate a collection of primitive colliders to approximate a concave collider. The process of converting an object using a mesh collider, into an object using a SACollider is relatively painless depending on the complexity of the mesh. It allowed us to make a torus with a hole pretty fast, see it in the animated gif.[3] Though, at times it is needed to tweak the SACollider. In many cases, a convex collider will be better. However, for the objects that have a concave shape which the collider needs to reflect, SAColliderBuilder saves a lot of time.

https://justworksltd.gitlab.io/playground-docs/media/Gifs/Colliders/issueswithcolliders210219.gif

[2]Panda hand has no friction.
https://justworksltd.gitlab.io/playground-docs/media/Gifs/Panda%20hand/VRSlippery.gif

[3]Torus using a SACollider.
https://justworksltd.gitlab.io/playground-docs/media/Gifs/Colliders/TorusWithHole.gif

A limitation of the SAColliders is that it generates primitives that all have an origin somewhere on the surface of the mesh. Meaning that the SACollider will always have a larger boundary than the corresponding mesh collider, making it less accurate by nature. However, for our use, the accuracy is more than enough. If some edge case occurs where the collider is too inaccurate, it can be adjusted manually. This takes significantly less time than making the whole collider from scratch.

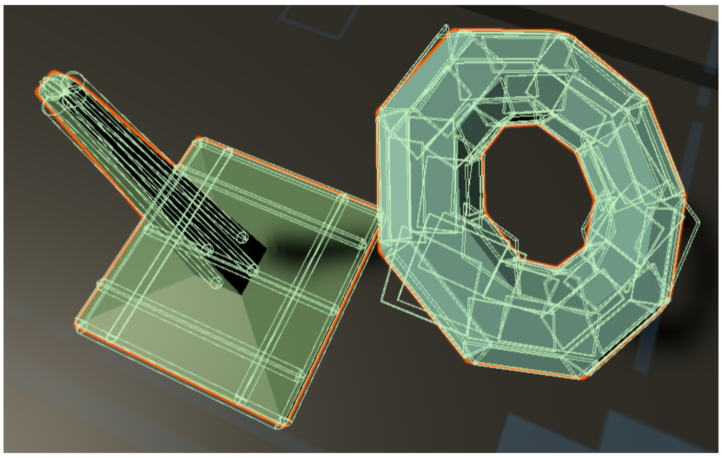


Figure 21: SACollider - A collection of primitive colliders generated along an object's surface. The pole uses capsule colliders, while the torus uses cube colliders.

6.5.3 Friction Solutions

Rigidbody's Move Function

An early solution to the friction problem was using a Rigidbody's move function (`someRigidbody.MovePosition(somePosition)`) to translate the hand as shown in the animated gif.[4]

Proxy Rigidbody

When moving something by setting the transform's position directly, friction is not applied. We solved this by creating a *target transform* that is moved by SteamVR(setting the transform's position directly), then the actual object that needs friction sets the rigidbody's velocity to move towards the *target transform* as demonstrated in this animated gif.[5]

Since it now uses velocity and angular velocity, we do spherical LERP on the rotation to make it more realistic, as a robot hand in real life does not suddenly change rotation. Other minor improvements this allowed, was to clamp both the acceleration and the max speed of the robot hand to make it act more like the real life servo powered robot hand. This approach also fixed the issue with the robot hand passing through solid surfaces, as the motion-controller and the actual hand was disjointed, meaning the controller could move through the wall while the hand collides with the wall and stops.

[4]Panda hand uses Rigidbody move function to calculate friction.
https://justworksltd.gitlab.io/playground-docs/media/Gifs/Panda%20hand/VRGrabby.gif
[5]Panda hand moves by proxy Rigidbody.
https://justworksltd.gitlab.io/playground-docs/media/Gifs/Colliders/GoodAgent.gif

NewtonVR

Another solution to the friction problem was to use a package called NewtonVR[29]. It offers velocity based physics between motion-controllers in VR and `Interactable` objects. Unfortunately, we needed velocity based physics on an equippable object (the robot hand), and NewtonVR did not have an implementation for this. If we had decided to use NewtonVR for Neodroid Playground, we would have had to implement this extension ourselves.

We ultimately decided not to use NewtonVR. However, if we had used it, we could have gotten a few things like object collision sound as a bonus.

Logic Based Friction

Our last resort to solving our friction problem if all else had failed, would have been to emulate friction using logic. We would approach this by calculating some value describing how good your grip on an object was. For example by how much force are being used combined with how much surface area is supposed to be generating the "friction".

Using these two factors as a baseline, we would attach the object to the hand with a joint. This joint would break if the force acting upon it became too great. This force limit would be calculated using the surface area value, force applied, the physical material's properties of the object being picked up, and whatever other factors we could think of that would improve the results.

There are many programming challenges with this approach, such as finding the surface area. We could test if both fingers are colliding with the same object, but this is a boolean value, and not a float. This means that getting a really bad angle on the object would not matter, as it would still give perfect friction, which is not realistic behaviour.

Another issue would be how to find the force used in a reliable way. When grabbing the object, we could potentially store at which distance both the fingers collided. Then we could measure the difference in distance between where the fingers are, and how far they want to go. Essentially, measuring how far into the object they would have gotten if the object was not in the way of the fingers. Then using that distance as a value for how much force is being used.

The Solution We Used

Because using a proxy-rigidbody to get friction worked as well as it did, and considering how complicated logic based friction would have been, we decided not to attempt it. Implementing logic based friction would have taken up a lot of time as well, while probably still not giving ideal results. We decided on using the proxy-rigidbody solution instead, as it provided promising results, had better scalablilty, and was a more elegant solution in general.

6.5.4 Panda Hand Colliders

The robot grabbing hand on the Franka Emika Panda robot needed to behave realistically. It consists of three bodies, the main body of the hand, and one for each finger. The fingers are connected to the main body using the `ConfigurableJoint` class, and are opened and closed with a script.

The fingers would fold unnaturally when it hit a static object since no amount of spring/motor strength in the finger's joint could overpower the infinite inertia of a static object. It would fold on an axis it was supposed to be completely restricted in. This issue was solved using collision layers, which we have illustrated in fig. 22. The four colours of colliders highlight how the hand is split up in order to get the most realistic collision: blue, green, yellow and red. The green is on the default collision layer, the blue is on a second, while the yellow and red are on a third.

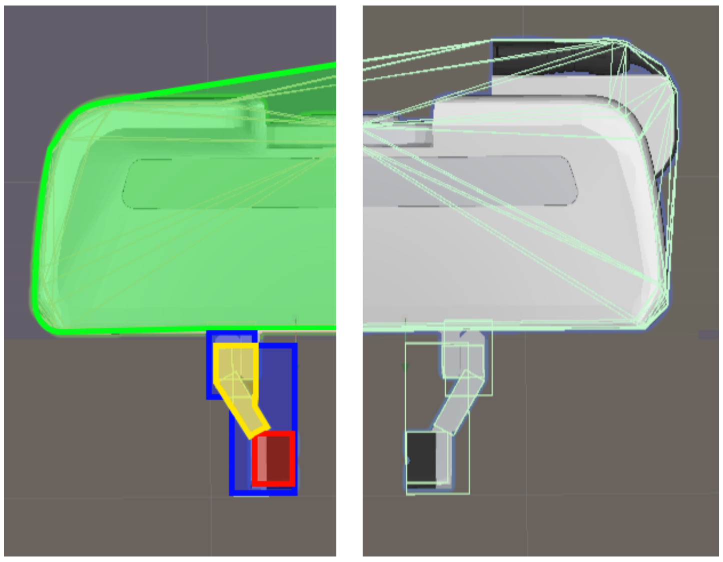


Figure 22: How the colliders of the Panda Hand are organized - Green collides with *everything*, blue collides with everything *except* SceneObjects, and is used as a "shield", and yellow & red collides with *only* SceneObjects

The green section collides with everything. The blue layer collides with everything except SceneObjects. It acts as a shield for the fingers against static objects, like a table or the ground. The yellow section of the collider only hits SceneObjects and acts like metal in respects to friction values. Lastly, the red section acts the same way as the yellow, however, it has friction properties that act more like rubber because it is the part of the hand that is supposed to be the easiest to grab objects with.

In order for the blue "shield" to protect the fingers, it needed to follow them as they slid back- and forwards. They could not be a part of the fingers themselves, as they would then be affected by static objects the same way as the yellow and red colliders. The blue "shield" is instead a part of the main body and is moved to the same location as the fingers using the same script that is moving the fingers' joints.

6.6 AI Training

The end goal of the system is to apply machine learning on an agent in a set environment with a specific task to solve. There are several issues to consider in regards to this. First of all, we need to have a working interface with Neo, meaning that we must create our system as required by Droid to have a working connection with Neo, when there is an Agent-process running. Furthermore, we must ensure that the evaluation of the scene makes sense and works to get the intended behaviour from the agent. Also, as we are using sparse rewards for the evaluation sent to Neo, we have to consider that the machine learning will struggle to even find a reward at all. Especially when there is a multi-dimensional action-space, which can cause the agent in our scene to get lost quite quickly.

To counteract the issues of sparse rewards, we need to use reverse curriculum generation to ensure that the agent first is given simple states that it can solve easily. That way it will learn those simple states and expand its frontier to learn how to complete the task.

6.6.1 Interfacing with Neo

To ensure a functional interface between Neo and the Playground we needed to look into how Droid establishes this connection between the learning-process and the environment within Unity. For a visualization of this, see fig. 23 below. We got an impression of how to set up a functional environment with Droid when we visited SINTEFfor a workshop. When we started developing the Playground and established working environments for machine learning, we found this prior experience to be really helpful. As it taught us a lot about how the Droid system works internally.

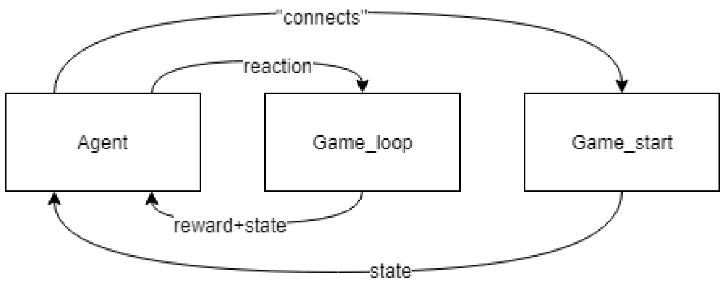


Figure 23: Displays the relation between the Agent process and the scene containing Droid components in Unity. Figure taken from the official Neodroid documentation [30].

In essence, we used a lot of the pre-existing components in Droid. Where we saw it necessary, we implemented our own versions of the Droid-components. At its core there are a couple of components required for Droid and Neo to work together correctly:

DroidEnvironment	This is a container for the agent's environment.
Evaluation	This component evaluates whether an agent should receive a reward or be punished depending on the state of the scene. Evaluation can be implemented in several different ways, depending on what the current task is.
Actor	This component acts as the agent in a scene, fundamentally it works as a container for the motors that the agent uses to interact with objects in the environment.
Motors	Components on an actor that can apply some sort of motion based on the output from Neo. These are components that make it possible for the actor to interact with the environment.
Observers	These are the components gathering information (observations) on objects in the environment, which is sent as input to Neo.

Taking these into account, we could for the most part use these components as they were. We found the pre-existing motors and observers sufficient for our needs. However we needed to implement a generalized `Evaluation` for any configuration of a scene. All the existing `Evaluation` classes in Droid were very specialized. Thus, we created the `SceneStateEvaluation` class that would take all conditions into consideration when evaluating the state of the environment (see fig. 7). By doing this, we made sure that no matter how many or what type of conditions existed in the environment, it would be able to evaluate it correctly for the Neodroid-Agent module.

6.6.2 Sparse Rewards and State Evaluation

For the agent to learn the task correctly, we use sparse rewards. In practice, this means that the agent will only be rewarded when successfully entering the intended goal state. Which means that the signals it could receive are:

1: The environment is now in its intended goal state and the agent receives a reward for this to emphasize that this is something it should work towards.

0: The environment is neither in a goal state nor a terminating state, meaning that the agent receives a neutral signal that it has not caused anything wrong, but not fulfilled the goal state yet either.

-1: The environment is in a fatal state, because the agent did something it was not allowed to. This yields a punishment for the agent to emphasize that this is something it should try to avoid.

These signals originate from the `SceneStateEvaluation` which is traverses the condition-logic-trees (as seen in fig. 5). The `SceneStateEvaluation` will go through each SceneObject in the environment and its conditions each object will evaluate whether they are fulfilled or not (see section 6.1.2). In other words, the `SceneStateEvaluation` will go through a tree-like structure consisting of one list of terminating conditions, and one list of goal conditions, as can be seen in listing 6.8 below, it will go through each SceneObject in the scene. First, it will check for any terminating conditions, if these are fulfilled it will return -1 and terminate the session. However, if no terminating conditions are fulfilled, it will go through the list of goal conditions and the condition groups within it. If any of the root conditions or condition groups are fulfilled, it will return 1 to reward the agent for its actions which resulted in a goal state.

```
public override float InternalEvaluate()
{
    foreach (var sceneObject in sceneObjects)
    {
        if (sceneObject.TerminatingConditions.ConditionsInGroup > 0 &&
             sceneObject.TerminatingConditions.Evaluate())
        {
            ParentEnvironment?.Terminate($"Entered terminating state on object: { sceneObject }");
            return -1; // Terminating condition were fulfilled, returning negative signal
        }
    }
    foreach (var sceneObject in sceneObjects) // Checking goal conditions after termination conditions
    {
        if (sceneObject.GoalConditions.ConditionsInGroup > 0 && sceneObject.GoalConditions.Evaluate())
        {
            ParentEnvironment?.Terminate($"Entered goal state on object: { sceneObject }");
            return 1; // Goal conditions were fulfilled, returning positive signal
        }
    }
    return 0; // No conditions were fulfilled, returning neutral signal
}
```

Listing 6.8: Scene state evaluation (see full code F.5).

6.6.3 Reverse Curriculum Generation

Using sparse rewards causes some issues when it comes to reinforcement learning, mainly due to the low rate of signals. When an agent does not receive a positive, nor a negative signal for its actions, it is difficult for the agent to learn how to solve the task. We are working with an agent that has a multi-dimensional action space, the likelihood that the agent will choose a correct set of actions to reach a goal state is very low. To solve these issues caused by sparse rewards, we use a concept called reverse curriculum generation. Fundamentally, this entails that the agent starts from simple states that are few actions away from a goal state, and when it learns to recognize correct action sets from these easier states, we expand its frontier by deviating further from the states it has learned to recognize.

As of now, we do not have generic RCG implemented as intended. The reasons for this are discussed in section 8.4. Our idea for how this would work can be broken down in a few quite simple steps:

1. Put the environment in a goal state.
2. Put the agent next to the object that is in a goal state.
3. The agent applies some random motions on its motors until the amount of actions required are just outside the agents frontier.
4. If the environment no longer is in a goal state and not in a terminating state either, we save this scene state as a RCP.
5. Repeat steps 1-4 until we have a big enough sample size to start machine learning.
6. Apply machine learning on the samples collected.
7. Once the agent's accuracy reaches a certain threshold from these RCPs we repeat steps 3-6 again. However, now we use the previously generated RCPs as the starting points, expanding its frontier.

This is simple in theory, but it gets quite advanced for environments with several complex condition configurations. Furthermore, there could also be that some conditions that depend on other conditions. With all of these factors to consider, an implementation of RCG in our system will have to be carefully planned. Even though this is not something we have implemented at this point, it is something we are looking to implement in the future (see section 8.5).

We do have a simple proof-of-concept. Although it is not as functional as intended, it does show how the concept works to some degree. This is further covered in the section 6.6.4 below.

6.6.4 Simple Prototype Environments

To confirm that our interface with Agent was working as intended, we created a few simple environments to be able to test this out. This way it was easier to see if the Playground was implemented correctly, and whether it had any flaws.

Simple Franka Scene

The first time we set up a prototype environment to test our system for connecting with Agent and for testing our conditions in practice, we created a simple scenario where a robot would try to move its hand into a specific volume (using `PositionCondition`). The scenario was very simple as we did not have any reverse curriculum generation yet. For this agent, we used an actor consisting of the Franka Emika Panda robot arm, created by Franka Emika GmbH [3], which is one of the robots SINTEF has been using.

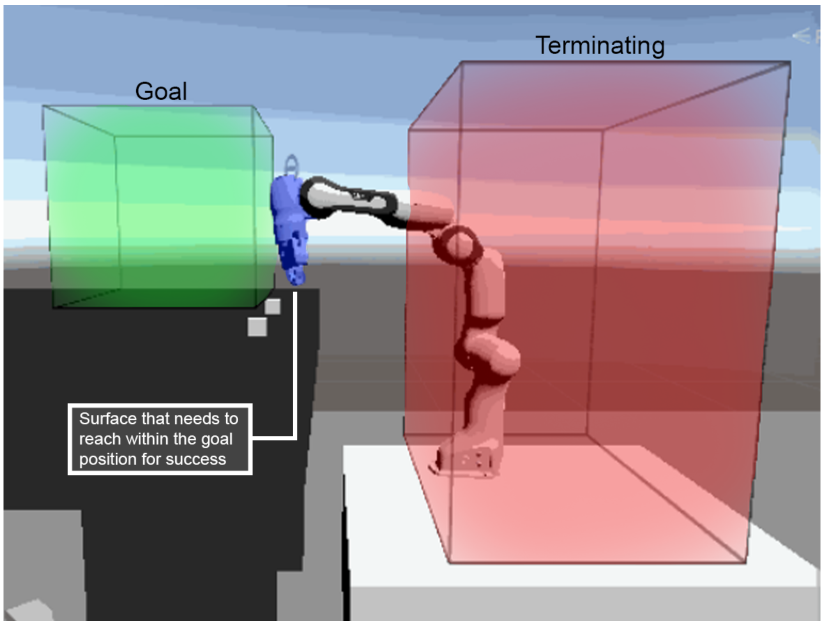
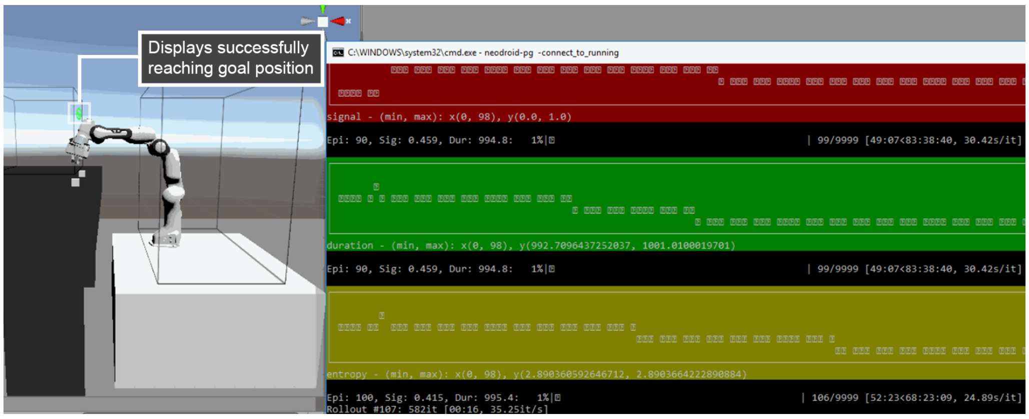

Terminating
Goal
Surface that needs to reach within the goal position for success

Figure 24: Scene with a very simple task for the Franka agent to solve. The task is to move the hand (marked with blue) into the goal volume, and stay outside the terminating volume

To mitigate the issues with not having RCG set up yet, we tried to create a really simple case where the actor would also be put very few actions away from the goal state. This did work to some degree. However, we did not find the agent to be very clever in that we did not see any significant improvement in its learning. It would appear to be relatively random whether the agent made it to the goal or not. This is not really unexpected, as the agent still needs to perform a multitude of actions. Even when we placed it very close to a goal state, the action-to-reward rate was too low to learn from. In other words, the action space and chance of getting off track is very likely, despite the task at hand being so simple. Have a look at this gif to see it in action.[6]

[6]Franka robot arm training example.
https://justworksltd.gitlab.io/playground-docs/media/Gifs/AI%20Training/TrainingStart.gif


Displays successfully reaching goal position
C:\WINDOWS\system32\cmd.exe - neodroid-pg -connect_to_running

```
signal - (min, max): x(0, 98), y(0.0, 1.0)
Epi: 90, Sig: 0.459, Dur: 994.8:   1%|                    | 99/9999 [49:07<83:38:40, 30.42s/it]
duration - (min, max): x(0, 98), y(992.7096437252037, 1001.0100019701)
Epi: 90, Sig: 0.459, Dur: 994.8:   1%|                    | 99/9999 [49:07<83:38:40, 30.42s/it]
entropy - (min, max): x(0, 98), y(2.890360592646712, 2.8903664222890884)
Epi: 100, Sig: 0.415, Dur: 995.4:   1%|                  | 106/9999 [52:23<68:23:09, 24.89s/it]
Rollout #107: 582it [00:16, 35.25it/s]
```

Figure 25: Franka Emika Panda achieving its goal condition. The graphs displayed are debug information and machine learning information from Agent. The red graph displays the signals the agent has received (rewards), the green displays the duration the episodes' lasted, and the yellow displays its entropy.

It is worth mentioning that it seemed to make some progress after many iterations, but because the rate of completion was very random, the progress seemed to be very slow. If this scenario had used RCG, the number of actions done in each episode would on average be significantly lower because the likelihood of completion would be a lot higher. Whenever the agent completes the task, it would be rewarded and go to the next episode. This would help emphasize what is important in the environment for the agent as well as speeding up the whole machine learning process drastically. We worked out a way of approximating reverse curriculum for another simple environment, which is covered in the next section.

Rolling Ball Scene

We made a simple scene to showcase our system working with Neo to train a ball. The scene consists of five cubes and one ball. Four of the cubes have a terminating `TouchCondition` on them, and the remaining cube has a goal `TouchCondition`. The ball is a Droid actor, meaning that with a proper Neo environment the ball takes actions and learns from the Playground scene's feedback. The goal is randomized to be either up, down, left or right.

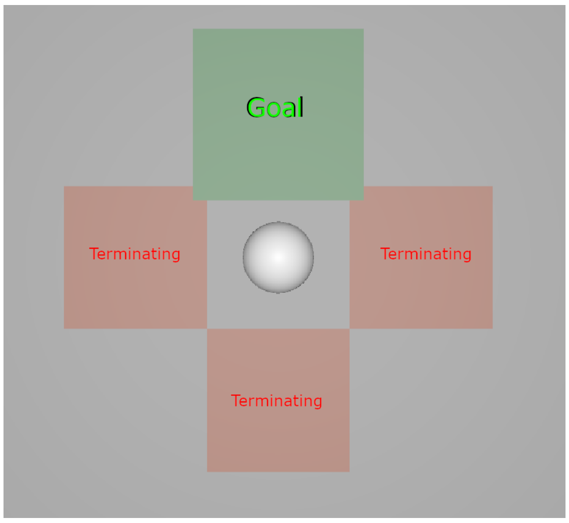


Figure 26: Simple ball-agent scene.

We also simulate RCPs in this scene by using Linear Interpolation (LERP) on the ball's spawn position based on the agent's accuracy. Right now this accuracy is independent of where the cube spawns. So the success rate is global and not relative to where the goal is (up, down, left or right). We would like to iterate this later. LERPing the ball's spawn position does to some degree work as an approximation of what reverse curriculum does, it is not as accurate. With working RCG, we would ensure that the agent would always expand its frontier every time its success rate had reached the desired threshold. Meaning that the agent would only have to apply one new set of actions on its motors to get to a state it could recognize. With LERPing the spawn position, we approximate the same thing, but there is a lot more room for errors. When LERPing; the agent will be less likely to fall into a state it could recognize from previous episodes. Even though it is not as precise as we would like, this scene does show the proof-of-concept to a certain degree.

It is worth mentioning that in a real scenario, one probably has a more complex environment than this, so the proof-of-concept using LERP, although seemingly working, would probably not scale to such a complex environment.

7 Testing and Quality Assurance

7.1 Pair Programming

Pair programming is a work procedure where two or more programmers work on the same screen to solve a problem. The persons with the keyboard is a *driver* while the others are *navigators* [31].

Pair programming was used when a problem was deemed complex in nature or very ambiguous in how one should solve it. Conditions were always very essential to our project and because of this, we pair programmed the conditions the first week. This also helped us share ideas about conventions. UI was also one of the bigger parts in the system and was designed and created through pair programming and -revision.

7.2 Test-Driven Development

We wanted to integrate TDD into our work process when developing the conditions as they are logical in nature and, in theory, easy to unit test. *Unity test runner*, an integration of Nunit (a framework for developing unit tests in C#) in Unity was our best contender [32]. The framework supplies functionality that you can come to expect from a unit test framework, like executing tests in any order through code (and GUI, and CLI), common setup and teardown functionality between the execution of tests, multiple assert functions, and so on.

7.2.1 Creating Good Tests

According to Microsoft [33], there are five main characteristics of a good test:

Fast Tests need to be fast as the number of tests can become very high. This is true, but not a key point in our project as tests are mainly used while developing conditions and not the whole system.

Isolated It is also important to stride for isolation of tests, so that they do not depend on mutable data to work properly.

Repeatable Repeatable tests are something we have been striving towards. Admittedly, some of our tests are not always repeatable as they are based on physics, and may change based on this (because physics in Unity is not deterministic). This could cause some tests to fail even when they should not.

Self-Checking Self-checking is something all tests become if you write unit tests in a proper framework. In our case, a test needs an "assert" to work and will be self-checking in nature.

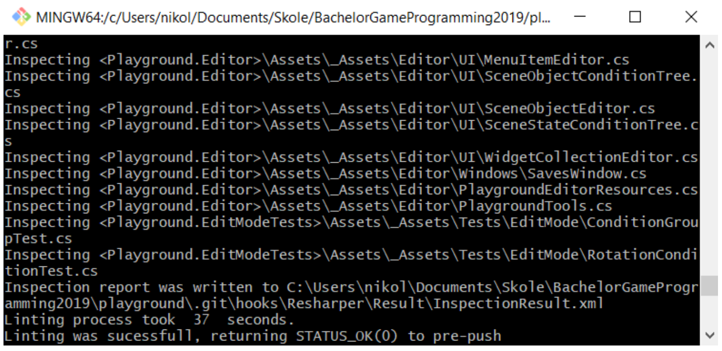
Timely Timely is another thing you get for free by using a proper test framework. Examples of Quality of Life (QOL) features from Unity Test Runner are the teardown function, setup function, and data source (see full code F.3).

We used the *AAA* (Arrange Act Assert) pattern when writing tests. Unit tests should start by arranging the scene, then act upon the built scene, and finally, assert that the result of the action is the expected outcome.

7.3 Git Hooks

We wanted to apply CI to our work process, but working on NTNU's deployment of GitLab [1] meant that we did not have all GitLab's functionality. There were some desired behaviour we wanted running upon a *push*, including: automatic linting of pushed code, execution of written tests, generation of documentation when pushing new code.

Our supervisor advised us to implement some git hooks to get this done instead (as we were unable to set up CI initially), which could do all of these things in theory although in a more inconvenient manner as it involves running the processes locally before a push.


MINGW64:/c/Users/nikol/Documents/Skole/BachelorGameProgramming2019/pl...

```
r.cs
Inspecting <Playground.Editor>\Assets\_Assets\Editor\UI\MenuItemEditor.cs
Inspecting <Playground.Editor>\Assets\_Assets\Editor\UI\SceneObjectConditionTree.
cs
Inspecting <Playground.Editor>\Assets\_Assets\Editor\UI\SceneObjectEditor.cs
Inspecting <Playground.Editor>\Assets\_Assets\Editor\UI\SceneStateConditionTree.c
s
Inspecting <Playground.Editor>\Assets\_Assets\Editor\UI\WidgetCollectionEditor.cs
Inspecting <Playground.Editor>\Assets\_Assets\Editor\Windows\SavesWindow.cs
Inspecting <Playground.Editor>\Assets\_Assets\Editor\PlaygroundEditorResources.cs
Inspecting <Playground.Editor>\Assets\_Assets\Editor\PlaygroundTools.cs
Inspecting <Playground.EditModeTests>\Assets\_Assets\Tests\EditMode\ConditionGrou
pTest.cs
Inspecting <Playground.EditModeTests>\Assets\_Assets\Tests\EditMode\RotationCondi
tionTest.cs
Inspection report was written to C:\Users\nikol\Documents\Skole\BachelorGameProgr
amming2019\playground\.git\hooks\Resharper\Result\InspectionResult.xml
Linting process took  37  seconds.
Linting was sucessfull, returning STATUS_OK(0) to pre-push
```

Figure 27: Output from ReSharper [22] linting on pre-push hook.

7.3.1 Result

We made a batch script that moves the hooks from the repository into the *.git*-folder. The hook itself is used to run ReSharper CLI to do basic linting on the commited code. It also checks the commit message to make sure it follows our conventions. The result is that if you are trying to commit code that does not follow the agreed-upon conventions or make invalid commit messages, it will cancel the push and provide feedback on why the push was cancelled.

[1]NTNU Gjøvik's deployment of GitLab - http://prod3.imt.hig.no/

7.4 Profiling in Unity

7.4.1 System and Environment

The laptop used for the profiling was an *MSI GL62 6QF-1418NE*

Storage		**RAM**		**CPU**	
Type	SSD	**Memory size**	8 GB	**Model**	Intel Core i5 6300HQ
Read	534MB/s	**Memory speed**	2133 MHz	**Cores**	4
Write	178MB/s	**Memory type**	SO-DIMM (DDR4)	**Clock speed**	2.3 GHz

GPU: Nvidia GeForce GTX 960M

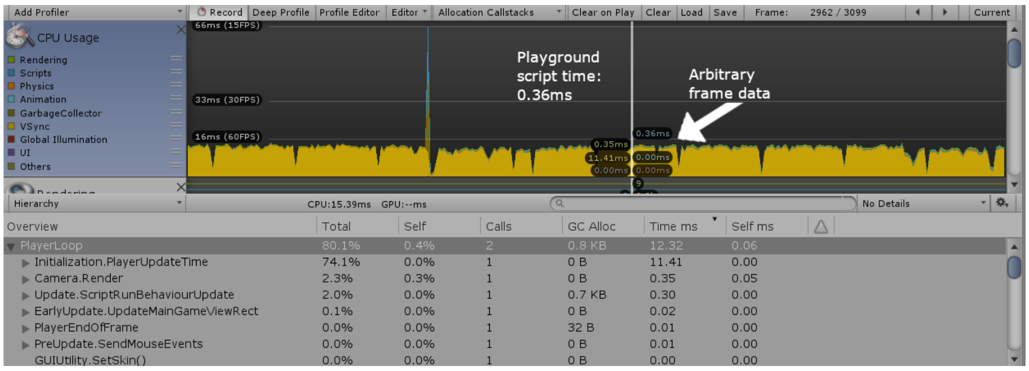
Vsync (Vertical Sync) was enabled with 60 FPS during the profiling session. We also conducted tests using only *PositionConditions* as they are the most used condition internally, has a decent amount of member data, and a relatively complex evaluation.

The specs used to conduct the profiling are below our recommended specs, but will work to showcase how scalable our system is.

7.4.2 Conditions

Condition evaluation is something that in theory might happen every frame. Because of this, we think performance is essential.

Evaluating 1000 Conditions



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	80.1%	0.4%	2	0.8 KB	12.32	0.06
Initialization.PlayerUpdateTime	74.1%	0.0%	1	0 B	11.41	0.00
Camera.Render	2.3%	0.3%	1	0 B	0.35	0.05
Update.ScriptRunBehaviourUpdate	2.0%	0.0%	1	0.7 KB	0.30	0.00
EarlyUpdate.UpdateMainGameViewRect	0.1%	0.0%	1	0 B	0.02	0.00
PlayerEndOfFrame	0.0%	0.0%	1	32 B	0.01	0.00
PreUpdate.SendMouseEvents	0.0%	0.0%	1	0 B	0.01	0.00
GUIUtility.SetSkin()	0.0%	0.0%	1	0 B	0.00	0.00

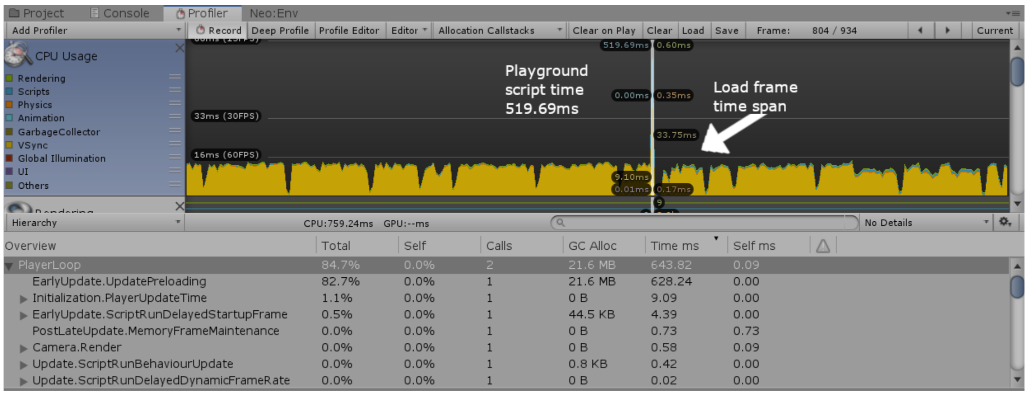
Figure 28: Profiling evaluation of 1000 conditions.

The evaluation time of 1000 false *PositionConditions* is between 0.25-0.4 ms. Using an average of 0.33ms and dividing with the number of conditions gives us an evaluation time of 0.00033ms or 330ns per condition. It's worth noting that if for example the first of the 1000 conditions were true, none of the others would be evaluated as the grouping is an *OR-group*.

7.4.3 Saving and Loading

Prior to our conducted tests, we tried reloading a Unity scene to see how long it would take to reload, which on average was 70 ms on our system. We also note that a high condition count will slow down and can potentially crash the inspector if a SceneObject with high condition count is selected in the Unity inspector. This is expected, as the inspector will execute evaluation and redraw for every condition (in order to visualize condition fulfillment). Hence the loading of a scene with such a high count of conditions should not be used for editor inspection, but rather for editing conditions on an existing profile or to run machine learning on the environment.

Loading 1000 Conditions

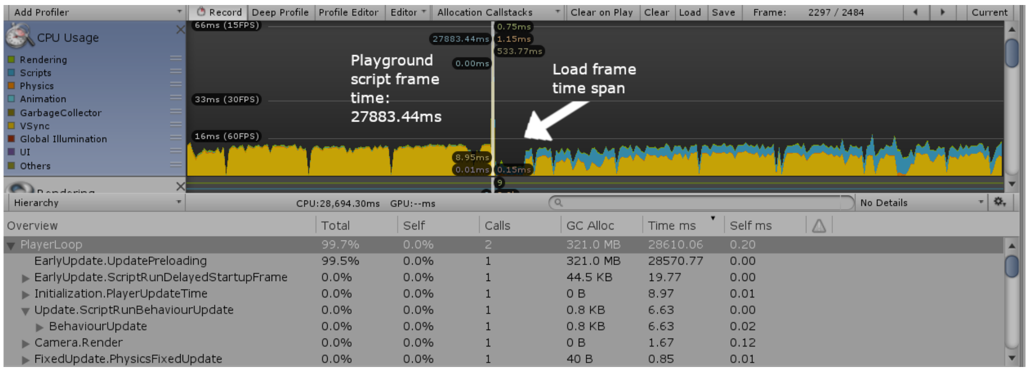


Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	84.7%	0.0%	2	21.6 MB	643.82	0.09
EarlyUpdate.UpdatePreloading	82.7%	0.0%	1	21.6 MB	628.24	0.00
Initialization.PlayerUpdateTime	1.1%	0.0%	1	0 B	9.09	0.00
EarlyUpdate.ScriptRunDelayedStartupFrame	0.5%	0.0%	1	44.5 KB	4.39	0.00
PostLateUpdate.MemoryFrameMaintenance	0.0%	0.0%	1	0 B	0.73	0.73
Camera.Render	0.0%	0.0%	1	0 B	0.58	0.09
Update.ScriptRunBehaviourUpdate	0.0%	0.0%	1	0.8 KB	0.42	0.00
Update.ScriptRunDelayedDynamicFrameRate	0.0%	0.0%	1	0 B	0.02	0.00

Figure 29: Profiling loading 1000 conditions.

The result of loading 1000 *PositionConditions* is about 519 ms load time (script time used on the load frame) - Approximately ∼0.519 ms per condition.

Loading 15000 Conditions



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	99.7%	0.0%	2	321.0 MB	28610.06	0.20
EarlyUpdate.UpdatePreloading	99.5%	0.0%	1	321.0 MB	28570.77	0.00
EarlyUpdate.ScriptRunDelayedStartupFrame	0.0%	0.0%	1	44.5 KB	19.77	0.00
Initialization.PlayerUpdateTime	0.0%	0.0%	1	0 B	8.97	0.01
Update.ScriptRunBehaviourUpdate	0.0%	0.0%	1	0.8 KB	6.63	0.00
BehaviourUpdate	0.0%	0.0%	1	0.8 KB	6.63	0.02
Camera.Render	0.0%	0.0%	1	0 B	1.67	0.12
FixedUpdate.PhysicsFixedUpdate	0.0%	0.0%	1	40 B	0.85	0.01

Figure 30: Profiling loading 15000 conditions.

The result of loading 15000 *PositionConditions* is about 27883 ms load time. The resulting load time per condition is ∼1.86 ms. That's about ∼360% increase in load time per condition compared to the 1000 condition result.

Results Discussion

We are not sure about the cause of the increased load time per condition. Our guess is that it has to do with caching of the groups' lists becoming segmented after a growing to a certain size. Naturally, a user will probably never get 15000 conditions in a scene unless they are doing some benchmarking of their own. It gives us some insight into the performance cost of conditions and load time that's why it's worth looking at.

We would like to improve load times in the future, but it would require a total rewrite of the save/load-system. There are some things we could change in the current system. For example: at the moment it loads a container of the condition and then converts it. It would likely be faster to collect all containers and convert them all at the end instead.

7.5 User Interface

When we developed the menus for conditions, we created a quick prototype early in development. This was due to the fact that we wanted to have the program flow as early as possible. That way, we could iterate through initial design decisions and work on improving parts of the system whilst maintaining an operating program flow The prototype gave us access to test out more functionality in development, but it had a few flaws: It was very hard-coded and had a specific sequence flow, which was not something we wanted for the end product. It also caused this menu to lack certain features, like observing what conditions a SceneObject already contained, or modifying the properties of those conditions.

Another issue with the prototype menu was that it required a lot of "clicks" from the user, creating a less fluid experience.[2]

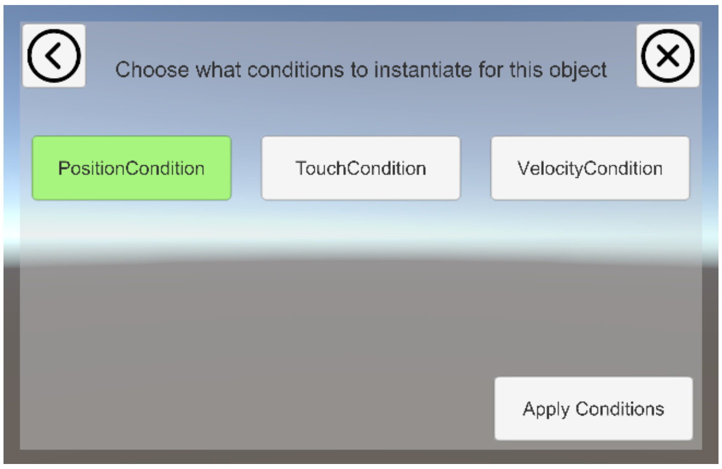


Figure 31: How one of the states of setup within the prototype menu looked like.

[2]Demonstration of prototype condition setup menu.
https://justworksltd.gitlab.io/playground-docs/media/Gifs/UI/earlymenuversion15022019.gif

7.5.1 Minimizing Number of "Clicks"

When we implemented the revised tablet menu for the Playground, we had the prototype menu's flaws in mind. We knew there were several things to improve upon, particularly creating a better user experience by reducing the number of "clicks" needed within the menu to achieve the same thing.

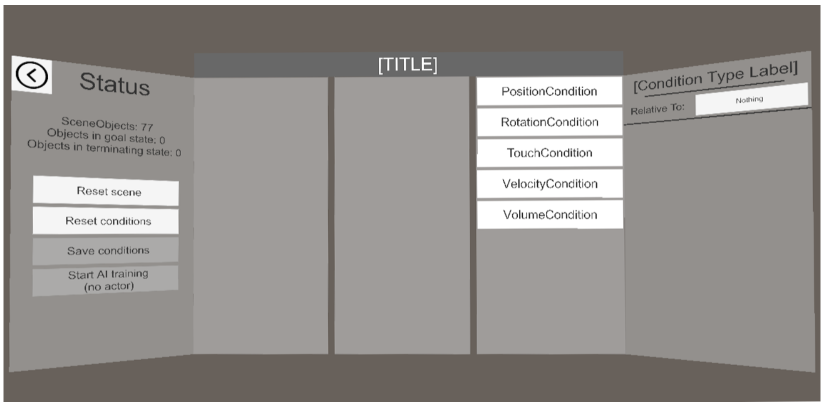


Figure 32: How the current condition overview menu looks like. The two empty columns represent the goal list and terminating list, as can be seen in fig. 13.

We immediately felt that this design was a lot better, even though it took some time to implement the way we wanted, as UI in VR can be quite challenging to do correctly. The new menu uses drag-and-drop interactions and some clicking, whilst the prototype menu only used clicking. So we can compare them by *number of interactions* instead of *number of "clicks"*:

To instantiate a new condition, the prototype menu required at least *8* interactions:

1. Select context SceneObject.
2. Confirm context selection.
3. Select relative SceneObject.
4. Confirm relative selection.
5. Select condition to add.
6. Press 'Apply Conditions'.
7. Edit condition's values.
8. Select state type (terminating state or goal state).

With the current menu we managed to reduce the number of actions required to *4*:

1. Select context SceneObject.
2. Drag-and-drop condition to add into terminating list or goal list.
3. Select relative SceneObject.
4. Edit the condition's values.

One factor that allowed fewer interactions was the fact that any action can be undone. This menu allows a user to edit or remove existing conditions, thus, there is not a lot of confirmations required from the user. Further, as all information is available in the menu's default view at all times, the whole interface is easier to manage for the user.

8 Discussion

8.1 Implementation Specific

8.1.1 Unit Testing

Testing Gains

We decided early on that conditions *should* have tests covering their use. Parallel to the creation of conditions, we made tests utilizing them. Most of the tests were made before the conditions as is most common to do in TDD. These tests have saved us some time as we discovered some condition-states previously not thought of.

We got what we wanted out of tests: We ensured working conditions throughout development, and the tests we have made can now serve as examples for future additions to the code base.

Refactoring

Our current tests are problematic at best. Some of them use physics-frames to work, meaning they are neither deterministic (see section 7.2.1) nor fast. This is planned to be phased out - Full code F.3 is a refactored example that runs in edit time (i.e. no physics-frames) instead. We plan to move *most* existing tests to edit time in the future. Some tests like those in `TouchConditionTest` (see full code F.4) will still require physics-frames to pass, to let Unity raise events.

8.1.2 Use of Namespaces

Conventions

One should not take namespace names for granted. There needs to be a certain clarity and transfer of information when reading a namespace name (just like general naming in code). Microsoft defines good naming of namespaces as follows:

`<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]` [34]

There is a good reason for this convention, especially the first part "<Company>". If a developer wants to use two systems called `Playground` by different companies, then they would be unable to. If both namespaces were used, they would get "symbol already defined" as the two namespaces collide.

Microsoft also has some other rules regarding namespaces:

- Use pascal casing when naming namespaces.
- Do not inlcude version number in the namespace name.
- Consider using plural names where approperiate.

Our Namespaces

We have tried to follow Microsoft's naming convention. We contravene one of the main rules. As of writing this: we do not work directly for SINTEF nor NTNU, and because of this we are not starting our namespaces with a company name, though this can easily be done later.

Our namespaces are as followes:

Playground - For components that are meant to be extended and used by external developers

Playground.Internal - For components that are designed to be "hidden" from external developers. This simply means that they ultimately should not need to be used directly, they are only used by internally by other systems.

Playground.SaveSystem - For the core save functionality. We chose a unique namespace for this module as it is one of the main modules we would like to rewrite and change in the future. Making a namespace makes it easier to do so. It is also one of the larger modules. Note that we do not follow the plural rule. As it is a single system namespace we deemed it appropriate.

Playground.UI - For all our UI related components. Just like the SaveSystem, the size of the UI system justifies its own namespace.

Playground.Tests - For unit tests and testing utilities.

8.1.3 Discarded CI Functionality

As mentioned, the GitLab CI pipeline did not work locally, and therefore we were unable to be deploy a Docker container that would run tests, do linting, and generate documentation remotely. All "CI" functionality was, for most of the development-time, implemented using git hooks.

Some group members used the first week of development to look into this. The Unity CLI was the first thing that wanted to have running remotely. There were no existing docker images of the Unity verion we used, and it was deemed too much work to make one from scratch to get it to build and run test, and was therefore discarded within the first days. We then tried to have a git-hook run the tests automatically locally upon pushing, but the Unity CLI did not allow this if you had a Unity process already running. Any attempt at making a workaround did not fix this issue and automatic testing was therefore discarded.

8.1.4 Graphical User Interface In Virtual Reality

One of the feature sets we had the most difficulty with was GUI. We were surprised by how little of Unity's GUI systems worked out of the box with VR. Subsequently, we spent a lot more time on it than we anticipated. We did not spend much time researching packages that could have helped us with this, which we should have in hindsight, as GUI is something most applications need. If we had found a suitable package to help us, we could have saved a lot of time.

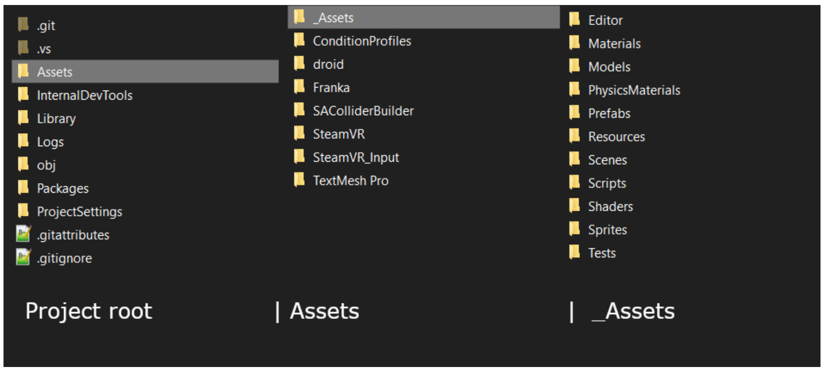
8.1.5 Using Git with Unity

Unity tends to generate a lot of Binary Large Object (BLOB) data by default, and Git happens to be pretty bad at handling BLOB data. Fortunately, Unity has an option to force yaml-serialization, meaning we will not have binary-file conflicts. However, it is still very difficult to properly merge if a conflict arise in such a yaml-file, as they *barely* human-readable. [35] To avoid this problem, we decided early on to work on separate files of this type (scenes, meta files, prefabs, asset files, etc.). Additionally, whenever pushing asstes, we ensured the corresponding meta file was included in the same commit. This strategy worked fine for us.

Git LFS

One of the shortcomings with Git is that it is slow at transferring large files. We enabled Git Large File Storage (LFS) to handle large files in our project. Git uses HTTPS to transfer the files separately, and so by using LFS, we are sparing Git these large file transfers. Using Git LFS in projects with large files improves pull-/pull times significantly.

Project File Structure



```
.git                  _Assets               Editor
.vs                   ConditionProfiles     Materials
Assets                droid                 Models
InternalDevTools      Franka                PhysicsMaterials
Library               SAColliderBuilder     Prefabs
Logs                  SteamVR               Resources
obj                   SteamVR_Input         Scenes
Packages              TextMesh Pro          Scripts
ProjectSettings                             Shaders
.gitattributes                              Sprites
.gitignore                                  Tests

Project root        | Assets              |  _Assets
```

Figure 33: Project File Structure

From previous experiences, we knew that working actively on files within Unity's default folder *Assets* can be tedious when trying to add files for commit in Git. It also tends to create messy navigation within Unity in the *Assets*-folder, as it tends to get filled up with a lot of other files (from external plugins and packages). For these reasons we used an _*Assets*.folder as our main folder, making Unity place it on the top in the project window, and allowing us to separate our project files from external files.

8.1.6 Optimizing Evaluation

It's is worth starting this section by mentioning that the performance of condition evaluation has not been a problem in our current tests (see section 7.4.2). We admit however that the current evaluation is not speed oriented. Although we would like to make evaluation faster, we do not want to trade in readability for speed when speed is not an issue. There are some things that we could do in theory to increase the speed while maintaining readability.

We are already optimizing evaluation speed by ending the evaluation if we find a SceneObject to be in a terminating state. By evaluating terminating conditions on SceneObjects before its goal conditions, we ensure that we end evaluation early if the environment is in a TerminatingState (see fig. 7). However, there are several other ways we see as potential candidates for improving performance when evaluating the scene state:

Multithreading is the first improvement we would like to introduce. The groupings of conditions allow us to create threads to evaluate larger groupings. In the case of SceneObjects, we could dynamically create threads when groups are deemed big enough. Since the group structure is branching, we could also create threads when there are big enough branches where the gained speed makes up for the overhead of creating a new thread. This would come at the cost of introducing complexity to the code and although complexity does not imply worse readability, it usually is a precursor to it.

The second improvement we could make is usage of *Data oriented programming*. It usually results in a great speed boost when handling big sets of data, as you move the data from the heap to the stack to allow the Central Processing Unit (CPU) to cache data more optimally as relevant data is in most cases sequential in memory. You achieve this by separating data from logic so that data can be arranged into arrays. The mindset of this paradigm originates from something called a *Von Neumann bottleneck*, which has become an increasing problem in software performance. This is when the CPU has to wait for the transfer of data between it and the Random-Access Memory (RAM) [36]. The problem is alleviated with the use of data-oriented design, as it drastically decreases the amount of cache-misses because of the sequential layout of the data. The paradigm also allows for easier multithreading as types are placed in arrays. An arbitrary amount of threads can then iterate an assigned array and process it. This can lead to more cache misses and would have to be tested before applied in production.

Another potential optimization of condition evaluation as the system is now, is sorting `SceneStateEvaluation`'s list of SceneObjects based on how many conditions they have in their terminating condition groups. Larger groupings within a SceneObject's terminating list are evaluated first, as they are more probable to terminate the session early (in theory).

After there are no more SceneObjects with terminating conditions, the evaluation could sort on condition count and stop the evaluation loop when it reached the first SceneObject without a single condition. Since SceneObjects without conditions always evaluate to true, there is no need to change stack frames lots of times when evaluating potentially thousands of empty SceneObjects.

8.2 Project Planning

Our Gantt chart (see fig. 35 in appendix G) describes some of our initial milestones. As the module we were developing did not have any hard requirements on what it should be able to do or how it should be implemented, therefore these milestones had to be taken with a grain of salt. This is because the scope could easily change over the course of the project due to its agile nature. So when talking about milestones, one has to know that they were desired features to have implemented at certain times of the project, but they were not necessarily needed to be implemented within the time frame we had initially set.

8.2.1 Actual Milestones

Looking at a few of the milestones, we see that we did not really fulfill the set dates for completion. Take the *GameLoop*-milestone as an example - this was something we wanted to have early due to experiences from earlier projects; as it can end up being a lot of work to sew and couple things together if we do not consider the game loop early. We did think about the program loop early. The loop consists of several parts. One of the first parts we had was being able to annotate conditions in a scene, where they are contained in SceneObjects with some standards being defined there to some degree. We ended up having a loop with the Neodroid platform about one month later than the intended milestone date, around the middle of March. Although there are still some missing key points in this, like RCG, there are some hacked solutions in the system as a proof-of-concept. This is something we will look into in the future, as we will also be working with SINTEF this summer.

We moved away from referring to the milestones, and moved over to user-stories instead over the course of the project. We saw this as more suiting for the development. It allowed us to fill them into our scrum-board and move them over to the *completed*-list when they were implemented and verified.

User-Stories Completed

- Point to a SceneObject and add a condition (end of February).
- Verify that conditions are setup correctly by manually putting objects in their supposed goal states and terminating states (beginning of March).
- Start learning phase with robot arm after task annotation (middle of March).
- The user can save and load conditions into the scene through a editor window (beginning of April).

8.3 Work Reflection

8.3.1 Scrum

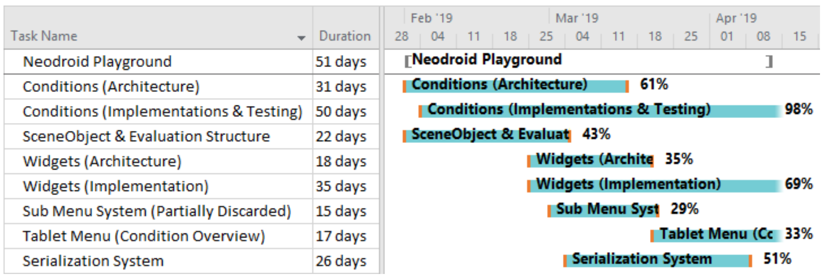
How we used Scrum

We used Scrum as our development model of choice. The issue board has been a fantastic tool to help us organize our workload. We have used GitLab's issue tracker, which we have used extensively. Issues are made for every task we come up with, and its issue number is referred to in commit messages that are relevant. Throughout the development we have not been punctual in relation to the activities and routines related to Scrum.

Deviation from Scrum

Scrum uses *sprints* to divide the work period into manageable chunks, with routine meetings to discuss the previous sprint, verify functionality, estimate the coming tasks and problems, and assign work to the developers. In the start of the project, we were more punctual. We had sprint meetings every Monday, and the first part of the day was dedicated to the sprint meeting. As the deadline moved closer, we stopped having the sprint meeting and continued working instead. We chose to prioritize productivity over blindly following structure. This is a risk we took, as the lack of management had the risk of hurting our productivity. Being less strict about meetings worked out for us because of proper communication about issues and other details through channels like the repository and Discord. Every day we worked in the same room, making it easy to quickly ask each other questions, ask for help or discuss issues on the board.

8.3.2 State of Completion



Task Name	Duration
Neodroid Playground	51 days
Conditions (Architecture)	31 days
Conditions (Implementations & Testing)	50 days
SceneObject & Evaluation Structure	22 days
Widgets (Architecture)	18 days
Widgets (Implementation)	35 days
Sub Menu System (Partially Discarded)	15 days
Tablet Menu (Condition Overview)	17 days
Serialization System	26 days

Figure 34: Time delegation percentages of major project topics in the implementation phase.

The current state of the project is that a lot of the architecture is there, and the main components are working as intended. The features we decided to focus on the most, got to a state where they are feature complete and without major known bugs. However, as fig. 34 suggests, three of our core features were still worked on at the last day of development. These being *condition implementations and testing* (like `PositionCondition`), *Widgets* (functional 3D- and 2D-widgets) and the *tablet menu*. This is because these features are not fully implemented at the time of writing. There are two main reasons for these components being incomplete: The first reason is that we have focused more on the framework and underlying architecture rather than front-end parts of the application.

The second reason is that the task itself was very open-ended in what way we wanted to approach it, which led to us creating a too big of a scope to manage within the time limit of the project. Though, it is worth noting that despite the large scope, we feel we managed to keep our focus on the most important aspects for the final product, and still delivered something that works because we focused on *minimum viable product*.

At different times in the development, we learned to be able to take a break from certain modules we had started on. This was the case for less vital tasks, or if a task would take too long to compared to what benefit it would provide. One example of a system like this is the *SubMenu* system as shown in fig. 34. These were features we wanted, to make a better experience for the end user, but we realized during development that the *tablet menu* was more important, and had to be prioritized. Even though the *SubMenu* is partially deprecated, parts of it were used to create the current *tablet menu*, so it was not like all the work was discarded. We also intend to expand on the features we did not get to fully implement when we are going to work at SINTEF as explained in section 8.5.

8.3.3 Group Work Reflection

Work Commitment

Throughout the project, all group members have been hard working. There has been very little absence of members in our group, and full day leaves have always had a good reason and been communicated.

Punctuality related to meeting times were a little lacking. This was not a big problem for us since all group members still were productive. It did not matter if a member showed up on time, as long the member showed up eventually during the meeting, and they got the agreed amount of work done. This flexibility meant that some members worked the night before a meeting, and were a bit late to the meeting. While others did not work much from home but worked the agreed time during the meeting. We believe this flexibility has had a positive impact on productivity as members could work when they felt like it without feeling that they were forced to work more than others. If we had been stricter, members might have chosen not to work when they would be most productive, and wait for the meeting where they would be less productive.

Attitude Towards Structure

Although we assigned specific roles with responsibilities, we have not felt the need to enforce that rigid structure. The roles in our project like the *log writer* (Jone Martin Skaara), were assigned because we needed to ensure those responsibilities were fulfilled. However other members have also been handling this responsibility, resulting in an even distribution of work. Each member was encouraged to write reflection notes about the features they had been working on. We learned more from each other as we got to read each others' reflections on the problems we encountered.

Another of the roles we assigned was *project leader* (Halvor Bakken Smedås). The main function of the group leader was to make decisions when there were disputes in the group. In the case where a vote resulted in a tie, the group leader would have two votes, effectively being the tie-breaker. We never had to use this as we primarily managed to discuss our options until the group met an agreement.

Communication

As mentioned we used Discord for communication. We had one channel for bachelor related discussion, one for off-topic discussion and a final for communication with the supervisor. This really lowered the bar for what was an acceptable post. You could write about, or link to anything in the off-topic channel, even if it was just something insignificant like a funny tweet, or a nice shader. All but the supervisor-channel was internal and is not visible to the supervisor, meaning we did not have to write formally. We feel this created a better group dynamic, as not all group members had worked together before. This lets us share interests and get to know each other better, which overall led to better work morale.

Portable VR for Testing

When developing for VR, a VR headset is required to test certain aspects of the functionality, some of the functionality can be tested without a VR setup, for example, serialization while other functionality like UI requires it.

During the implementation phase of the project, we had access to a MR headset. We set this up in whatever room we were working in that day. This meant that any functionality members had changed or expanded on from home could be tested in group work the next day. A positive side effect of this was that the functionality would be subject to pair review, opening for suggesting improvements and uncovering problems that would not have been discovered if it had been tested only by the developer who made it. It also served as a demonstration to the rest of the group that kept everyone up to date on the status of the different UI features.

Emulating VR Input

Some things related to UI needed a VR headset to be properly tested. When one was not available to a group member who was working on UI, they would be unable to test the functionality they were making. We solved this by emulating the behaviour of a VR headset and were able to test the UI using this system. An example of this is the drag-and-drop of conditions within the condition overview menu. for an animated gif demonstrating this, see [1].

8.4 Alternative Choices

In retrospect, there are some things we probably would have done differently if we approached this task again. First of all, when we had our workshop in Trondheim at SINTEF, we made a simple game with Droid, to apply machine learning on it with Neo. However, the game we made was not really suitable for RCG, and as a result of that, we did not get a chance to properly understand RCG with Neo and Droid. This has led to us not having worked out RCG in Playground. It is however something we will be looking at in the future (see section 8.5).

Secondly, we initially created conditions as pure objects. The reason for this was that we did not think we needed all the overhead functionality you get from using Unity's MonoBehaviours. However as the project went on, we ended up migrating over to MonoBehaviours after all, this gave us free Unity serialization for inspecting data in the editor, but it made it hard to do any .NET-serialization (or use other libraries that serialize). We did this migration due to serialization problems with more complex pure C# classes in Unity. As the system required some restructuring when it got the point were we needed serialization. We acknowledge that we should have gone for MonoBehaviours

[1] Demonstration of VR laser pointer emulation.
https://justworksltd.gitlab.io/playground-docs/media/Gifs/UI/laserpointer_emulate.gif

in the first place, while also thinking more about the design of the different systems in regards to serialization earlier on.

When the Playground Manager was drafted we did not think about the pattern and ended up using the *manager pattern*. We talked about Unity's paradigm (see section 3.1) previously. The *component pattern* works great for decoupling large classes into more modular and maintainable code. What we realize is that Playground Manager's tasks are too ambiguous, therefore the component pattern would have been the better alternative as we could decouple the logic into smaller components while still having a manager GameObject that contained said components. In theory, the components would make the code more maintainable and decoupled for further potential restructuring.

The *private pattern* could be used in our project. The private pattern is when you have a dedicated private data class for your main class [37]. As the serialization at the moment uses a container class of each condition, it would be easy to apply this pattern to existing conditions. It would also simplify the logic of serialization as the classes would be interchangeable without any custom logic. Potentially, it could break Unity-serialization and we therefore decided not to use it. Admittedly, the benefit of the private pattern is just speculation on our end, as we have not tested it because it struck us as an opportunity too late in the project.

8.5 Future Work

There are a few missing key points in Neodroid Playground that we would like to have fully implemented, like reverse curriculum generation. We have also focused a lot on making the `PositionCondition` full code F.1 work as well as possible, as we see it as the most essential condition. For future development, we wish to expand our set of conditions and look more into how other conditions should be implemented. This will make the system more viable for a wider range of tasks that would need to be described within Playground. We have been offered to work at SINTEF over the summer to continue the work we have started with this bachelor thesis, and we are looking forward to expanding Playground and its features when we are there. Also by co-operating with Christian Heider Nielsen we will hopefully be able to set up RCG to work with Playground. When we have that, there will be more room for testing out the system and expanding existing functionality.

9 Conclusion

The final product is something we definitely are satisfied with. Most of the underlying structure and architecture of the framework is complete, and the parts using the framework is in working order. On the other hand, we are lacking some essential UI functionality, but everything we want the user to do is possible through scripts. In other words: the framework is mostly complete. We are not done with all the functionality in the run time part of the solution.

The Scrum model turned out to be useful for our group, but our deviations from it helped us confine the model more to our needs. The sprints aspect of Scrum was used to *divide and conquer* our problems. Scrum meetings helped us become more efficient at communicating and teamwork, but became seemingly redundant later as communication efficiency increased through channels like Discord and the GitLab issue board. We changed the meetings to be more "on demand" rather than on a regular basis.

The scope of the project was never limited to a certain set of features. Instead, we and SINTEF came to an understanding of what the system was supposed to be. We definitely did not think we would be able to implement everything we discussed, and we did not get as far as we initially thought.

We have learned to take a step back every once in a while to look at other options instead of focusing on whatever problems are present at that time. We also experienced that keeping a document with reflections and implementation-dilemmas helped other group members get an overview of the project in the sense of why features were implemented the way they were, and what they tried first that failed. Developing and documenting a publicly available product has also taught us about what mindset we need to be in to properly get the point across - we needed to be more explicit and write in a coherent way so that any other developer would understand how to use our system.

In the end, we do have a solution that is only partially implemented, due to the reasons described in section 8.3.2. We defined a way to interface the abstract concept of *task descriptions* in a simple, yet powerful way by means of widgets and condition. We have made a framework that can be used to define new ways to describe tasks by making new widgets and conditions.

There are still important functionality missing from Neodroid Playground, such as persistence in condition properties (after changing them through widgets and opening the editor anew), interactions enabling the trainer to group conditions (which currently only is possible through code), missing widgets for existing conditions (meaning some conditions are not manipulatable, making them only useful from code), and removal of individual conditions.

To summarize, the project and report-writing in its entirety has been very educational, and exciting to work with as this was a new experience for us. It has been a nice conclusion to our bachelor studies here at NTNU Gjøvik.

Bibliography

[1] Discord. 2019. Discord - free voice and text chat for gamers. (Accessed on 10/05/2019). URL: https://discordapp.com.

[2] Wikipedia contributors. 2019. Docker (software) — Wikipedia, the free encyclopedia. (Accessed on 07/05/2019). URL: https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=895045866.

[3] GmbH, F. E. 2018. Franka Emika. (Accessed on 05/05/2019). URL: https://www.franka.de/.

[4] Microsoft. 2019. Windows mixed reality | ar mixed with vr gaming, travel & streaming in windows 10. (Accessed on 07/05/2019). URL: https://www.microsoft.com/en-us/windows/windows-mixed-reality.

[5] Wikipedia contributors. 2019. Test-driven development — Wikipedia, the free encyclopedia. [Accessed on 15/05/2019). URL: https://en.wikipedia.org/w/index.php?title=Test-driven_development&oldid=895358559.

[6] Huizinga, D. & Kolawa, A. 2007. Automated defect prevention; best practices in software management. *Scitech Book News*, 31(4). URL: http://search.proquest.com/docview/200122040/.

[7] Microsoft. *Generics - C# Programming Guide*, 07 2015. (Accessed on 11/04/2019). URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/.

[8] Microsoft. 07 2015. Reflection (C#). (Accessed on 07/05/2019). URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/reflection.

[9] Microsoft. 03 2017. Data Contract Surrogates. (Accessed on 15/05/2019). URL: https://docs.microsoft.com/en-us/dotnet/framework/wcf/extending/data-contract-surrogates.

[10] Microsoft. 07 2015. Unsafe keyword - (C# reference). (Accessed on 08/05/2019). URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/unsafe.

[11] Florensa, C., Held, D., Wulfmeier, M., & Abbeel, P. 2017. Reverse curriculum generation for reinforcement learning. *CoRR*, abs/1707.05300. URL: http://arxiv.org/abs/1707.05300, arXiv:1707.05300.

[12] Heider, C. 2018. Neodroid Platform. URL: https://github.com/sintefneodroid.

[13] Valve. 2019. SteamVR. (Accessed on 05/05/2019). URL: https://store.steampowered.com/app/250820/SteamVR.

[14] Palmer, C. 2018. *Virtual Reality Blueprints create compelling VR experiences for mobile and desktop*. Packt Publishing, S.l.], 1 edition. URL: http://portal.igpublish.com/iglibrary/search/PACKT0000015.html.

[15] Wikipedia contributors. 2019. Entity component system — Wikipedia, the free encyclopedia. (Accessed on 30/04/2019). URL: https://en.wikipedia.org/w/index.php?title=Entity_component_system&oldid=883535139.

[16] Wikipedia contributors. 2019. Multiple inheritance — Wikipedia, the free encyclopedia. (Accessed on 06/05/2019). URL: https://en.wikipedia.org/w/index.php?title=Multiple_inheritance&oldid=895053371#The_diamond_problem.

[17] Bob Nystrom. 2014. Component. (Accessed on 06/05/2019). URL: http://gameprogrammingpatterns.com/component.html#tying-back-together.

[18] Gamma, E. 1995. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass., 37th printing. edition.

[19] Technologies, U. 2019. Unity 3d. URL: https://unity.com/.

[20] Git. 2019. Git–fast-version-control. (Accessed on 06/05/2019). URL: https://git-scm.com/.

[21] 2019. Gitlab. (Accessed on 05/05/2019). URL: https://about.gitlab.com.

[22] JetBrains. 2013. Resharper cli. URL: https://www.jetbrains.com/resharper/features/command-line.html.

[23] 2019. Blender - 3d modelling tool. (Accessed on 05/05/2019). URL: https://www.blender.org.

[24] 2019. Toggl - Time tracking tool. (Accessed on 05/05/2019). URL: https://toggl.com.

[25] Microsoft. *C# XML Documentation Guidelines*, 07 2015. URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/xmldoc/xml-documentation-comments.

[26] Unity. 2019. Discord - free voice and text chat for gamers. (Accessed on 10/05/2019). URL: https://docs.unity3d.com/ScriptReference/UI.GraphicRaycaster.Raycast.html.

[27] Wikipedia contributors. 2019. Prototype pattern — Wikipedia, the free encyclopedia. (Accessed on 30/04/2019). URL: https://en.wikipedia.org/w/index.php?title=Prototype_pattern&oldid=877422408.

[28] Microsoft. *C# ISerializationSurrogate Documentation*, 07 2015. URL: https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.iserializationsurrogate?view=netframework-4.8.

[29] Labs, T. T. 2018. NewtonVR. URL: http://www.newtonvr.com.

[30] Heider, C. *Neodroid Architecture*. Sintef Ocean, 2017. (Accessed on 05/05/2019). URL: http://documentation.neodroid.ml/architecture.html.

[31] Wikipedia contributors. 2019. Pair programming — Wikipedia, the free encyclopedia. (Accessed on 05/05/2019). URL: https://en.wikipedia.org/w/index.php?title=Pair_programming&oldid=892120921.

[32] Unity developers. 2018. Unity Test Runner. (Accessed on 01/05/2019). URL: https://docs.unity3d.com/Manual/testing-editortestsrunner.html.

[33] John Reese. 2018. Unit testing best practices with .NET Core and .NET Standard. (Accessed on 01/05/2019). URL: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices#characteristics-of-a-good-unit-test.

[34] Microsoft. 2008. Names of namespaces. (Accessed on 03/05/2019). URL: https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/names-of-namespaces.

[35] Rick Reilly. 2017. How to git with unity. (Accessed on 05/05/2019). URL: https://thoughtbot.com/blog/how-to-git-with-unity.

[36] Wikipedia contributors. 2019. Von neumann architecture — Wikipedia, the free encyclopedia. (Accessed on 06/05/2019). URL: https://en.wikipedia.org/w/index.php?title=Von_Neumann_architecture&oldid=895354628#Von_Neumann_bottleneck.

[37] Bob Nystrom. 2014. Component. (Accessed on 09/05/2019). URL: https://sourcemaking.com/design_patterns/private_class_data.

A Appendix

B Project Agreement

NTNU

Vår dato   Vår referanse

Norges teknisk-naturvitenskapelige universitet

Prosjektavtale

mellom NTNU Fakultet for informasjonsteknologi og elektroteknikk (IE) på Gjøvik (utdanningsinstitusjon), og

____

____ (oppdragsgiver), og

____

____

____ (student(er))

Avtalen angir avtalepartenes plikter vedrørende gjennomføring av prosjektet og rettigheter til anvendelse av de resultater som prosjektet frembringer:

1. Studenten(e) skal gjennomføre prosjektet i perioden fra ____________ til______________ .

   Studentene skal i denne perioden følge en oppsatt fremdriftsplan der NTNU IE på Gjøvik yter veiledning. Oppdragsgiver yter avtalt prosjektbistand til fastsatte tider. Oppdragsgiver stiller til rådighet kunnskap og materiale som er nødvendig for å få gjennomført prosjektet. Det forutsettes at de gitte problemstillinger det arbeides med er aktuelle og på et nivå tilpasset studentenes faglige kunnskaper. Oppdragsgiver plikter på forespørsel fra NTNU å gi en vurdering av prosjektet vederlagsfritt.

2. Kostnadene ved gjennomføringen av prosjektet dekkes på følgende måte:
   - Oppdragsgiver dekker selv gjennomføring av prosjektet når det gjelder f.eks. materiell, telefon/fax, reiser og nødvendig overnatting på steder langt fra NTNU på Gjøvik. Studentene dekker utgifter for ferdigstillelse av prosjektmateriell.
   - Eiendomsretten til eventuell prototyp tilfaller den som har betalt komponenter og materiell mv. som er brukt til prototypen. Dersom det er nødvendig med større og/eller spesielle investeringer for å få gjennomført prosjektet, må det gjøres en egen avtale mellom partene om eventuell kostnadsfordeling og eiendomsrett.

3. NTNU IE på Gjøvik står ikke som garantist for at det oppdragsgiver har bestilt fungerer etter hensikten, ei heller at prosjektet blir fullført. Prosjektet må anses som en eksamensrelatert oppgave som blir bedømt av intern og ekstern sensor. Likevel er det en forpliktelse for utøverne av prosjektet å fullføre dette til avtalte spesifikasjoner, funksjonsnivå og tider.

4. Alle bacheloroppgaver som ikke er klausulert og hvor forfatteren(e) har gitt sitt samtykke til publisering, kan gjøres tilgjengelig via NTNUs institusjonelle arkiv hvis de har skriftlig karakter A, B eller C.

Tilgjengeliggjøring i det åpne arkivet forutsetter avtale om delvis overdragelse av opphavsrett, se «avtale om publisering» (jfr Lov om opphavsrett). Oppdragsgiver og veileder godtar slik offentliggjøring når de signerer denne prosjektavtalen, og må evt. gi skriftlig melding til studenter og instituttleder/fagenhetsleder om de i løpet av prosjektet endrer syn på slik offentliggjøring.

Den totale besvarelsen med tegninger, modeller og apparatur så vel som programlisting, kildekode mv. som inngår som del av eller vedlegg til besvarelsen, kan vederlagsfritt benyttes til undervisnings- og forskningsformål. Besvarelsen, eller vedlegg til den, må ikke nyttes av NTNU til andre formål, og ikke overlates til utenforstående uten etter avtale med de øvrige parter i denne avtalen. Dette gjelder også firmaer hvor ansatte ved NTNU og/eller studenter har interesser.

5. Besvarelsens spesifikasjoner og resultat kan anvendes i oppdragsgivers egen virksomhet. Gjør studenten(e) i sin besvarelse, eller under arbeidet med den, en patentbar oppfinnelse, gjelder i forholdet mellom oppdragsgiver og student(er) bestemmelsene i Lov om retten til oppfinnelser av 17. april 1970, §§ 4-10.

6. Ut over den offentliggjøring som er nevnt i punkt 4 har studenten(e) ikke rett til å publisere sin besvarelse, det være seg helt eller delvis eller som del i annet arbeide, uten samtykke fra oppdragsgiver. Tilsvarende samtykke må foreligge i forholdet mellom student(er) og faglærer/veileder for det materialet som faglærer/veileder stiller til disposisjon.

7. Studenten(e) leverer oppgavebesvarelsen med vedlegg (pdf) i NTNUs elektroniske eksamenssystem. I tillegg leveres ett eksemplar til oppdragsgiver.

8. Denne avtalen utferdiges med ett eksemplar til hver av partene. På vegne av NTNU, IE er det instituttleder/faggruppeleder som godkjenner avtalen.

9. I det enkelte tilfelle kan det inngås egen avtale mellom oppdragsgiver, student(er) og NTNU som regulerer nærmere forhold vedrørende bl.a. eiendomsrett, videre bruk, konfidensialitet, kostnadsdekning og økonomisk utnyttelse av resultatene. Dersom oppdragsgiver og student(er) ønsker en videre eller ny avtale med oppdragsgiver, skjer dette uten NTNU som partner.

10. Når NTNU også opptrer som oppdragsgiver, trer NTNU inn i kontrakten både som utdanningsinstitusjon og som oppdragsgiver.

11. Eventuell uenighet vedrørende forståelse av denne avtale løses ved forhandlinger avtalepartene imellom. Dersom det ikke oppnås enighet, er partene enige om at tvisten løses av voldgift, etter bestemmelsene i tvistemålsloven av 13.8.1915 nr. 6, kapittel 32.

12. Deltakende personer ved prosjektgjennomføringen:

NTNUs veileder (navn): ____

Oppdragsgivers kontaktperson (navn): ____

Student(er) (signatur): ____ dato ____________

____ dato ____________

____ dato ____________

____ dato ____________

Oppdragsgiver (signatur):____ dato ____________

Signert avtale leveres digitalt i Blackboard, rom for bacheloroppgaven.
Godkjennes digitalt av instituttleder/faggruppeleder.

Om papirversjon med signatur er ønskelig, må papirversjon leveres til instituttet i tillegg.
Plass for evt sign:

Instituttleder/faggruppeleder (signatur): ______________________________________ dato ____________

C Meeting Logs

The following is our meeting logs (norwegian).

Referat onsdag.09.01.19   kl 14:00 - 15:10
Ikke oppmøtt: INGEN
Sted: A153

- Lagde liste over nærmeste gjøremål.

  "Det viktigste å gå igjennom, e jo.... nei ka va det eg sa??"
  - *Aksel Hjerpbakk, 2k19*

Referat onsdag.10.01.19   kl 14:24 - Seint
Ikke oppmøtt: INGEN (Alle seint ute)
Sted: S410

- Den store referansedagen!

Oppgavereferanser:
https://www.forskningsradet.no/prosjektbanken/#/project/NFR/262900
https://ieeexplore.ieee.org/document/8324578
https://www.youtube.com/watch?v=ox_uJd6yHzo
https://github.com/sintefneodroid

Benchmarking av physics engines:
https://homes.cs.washington.edu/~todorov/papers/ErezICRA15.pdf

MuJoCo (Physics engine) Unity plugin - http://www.mujoco.org/book/unity.html

Expected learning outcome:
https://www.ntnu.edu/studies/courses/BPROG39#tab=omEmnet

"jeeezes!"
- *Aksel Hjerpbakk, 2k19*

"Min feil"
- *Halvor Smedås, 2k19*

Referat fredag.11.01.19   kl 12:45 -

Ikke oppmøtt: INGEN

Sted: S410

Paths etterlatt av kontrolleren under demonstrasjonen som kan markeres som prioritet eller viskes vekk.

Hovedmålet er å lage en unity applikasjon som kan brukes til å bygge en lekegrind. Lekegrind kan bestå av:

- Paths.
- Death sone der AI-en straffes og resettes.
- Meny til å spawne objekter.
- Velge et objekt som target, slik at resten blir ignorert.

Docs:

https://neodroid.ml

"Ingenting ska funk i dag"

- *Aksel Hjerpbakk, 2k19*

"Gad daaaaemn Skype for 'Business'"

- *Jone Skaara, 2k19*

Referat mandag.14.01.19   kl 09:45 - 10:25

Ikke oppmøtt: Nikolai

Sted: Mariusz' kontor

"Weeeeeell..."

- *Mariusz, 2k18*

Referat mandag.14.01.19   kl 12:00 - 14:00

Ikke oppmøtt: INGEN

Sted: A255

- Satt opp budsjett for tur til SINTEF
- Sendt mail for forespørsel om støtte fra NTNU og SINTEF

"Prisen for hotell er grønn fordi det ikke er utgifter men inngifter"

- *Nikolai, 2k19*

Referat onsdag.16.01.19   kl 12:15 - 16:15

Ikke oppmøtt: INGEN
Sted: S-bygg 2. Etasje

- Bestilt togbillett opp til Trondheim
- Bestilt hotell 22. Jan til 25. Jan
- Sendt kontaktinformasjon for NTNU til Sintef
- Startet på prosjektplan

Referat torsdag.17.01.19   kl 14:30 - XX:XX
Ikke oppmøtt: Nikolai (gyldig)
Sted: S-bygg 4. Etasje
- Det er et poeng å definere en spesifikk måte å ta input på toughpad-en på vive kontrolleren på slik at det ikke blir flere standarder innad i prosjektet.
- Vi bør vurdere hvordan input fra toughpad-en bør gjøres med tanke på at mange kontrollere kun har analogstikke, og å bruke den som en erstattning for touchpad kan bli *clunky* hvis input systemet blir designet for dårlig.
- **Mulighet**: legge in MoVR-funksjonalitet for å ta opp demonstrasjonene for instant replay. Dette kan så brukes for å se demonstrasjonen spilt igjen i VR, men og i virkeligheten ved bruk av den fysiske roboten.

Referat mandag.21.01.19   kl 9:40 - 13:00
Ikke oppmøtt: Ingen
Sted: Mariusz kontor
- Det blir bestilt 2 nye vive kontrollere og 2 pucker/ sensorer.
- Det er lurt å ta notater under oppholdet i Trondheim.
- Nevne angående hosting av repo
- Prosjektplanen trenger ikke å være engelsk, men er lurt siden den blir brukt i bacheloroppgaven.
- Hvem er sekretær? JONE!
- Microsoft room building tool https://www.maquette.ms/

"Historyen skal holdes..."

- *Nikolai, 2k19*

"Med unntak av når man ikke vil at den ska det."

- *Halvor, 2k19*

Referat tirsdag.22.01.19   kl 14:45 - 21:30

"Det er dobbelt så varmt i Trondheim enn i Gjøvik" (-14C Gjøvik, -7C Trondheim :) :))

- *Halvor, 2k19*

Referat fredag.25.01.19   kl 09:15 - 11:00

- Definere lekegrind
    - Definere lekegrind
    - Ui vs ux:
    - ui = utvidelse av informasjon i for a hovedsaklig GUI.
    - Ux = GUI og funksjonalitet som forbedrer brukeropplevelsen.
        - Konfigurérbar med object selection and action of poses/states - also choose how many **(UX & object component system thingy (OCST))**
        - Mål: Fra problem-statement til implementert solution så fort som mulig (Må ikke nødvendigvis være i VR) (-)
        - Kan være enklere å annotere direkte hva som er viktig mens man demonstrerer en task i VR. Istedenfor å generere paths og gå over etterpå. **(UI)**
        - Hvordan definere hva som er mål for objektet, vi forstår colliders, men ikke "bonden". Hvordan definere eksplisitt/implisitt hvilke constraints som gjelder for objektet når det skal nå mål. **(UX)**
        - Lage et system der det skal være enkelt å generere reward states **(UX & neo?)**
        - Annotere meny (si hva som er viktig for staten) **(UI)**
        - Plassere objekter i lekegrinden (editor, ev. VR om det skulle vise seg å være bedre) **(UX & OCST)**

- Men lage en form for meny for hva slags scripts som skal plasseres på et objekt som man vil utføre læring på i VR (f.eks om man har 3d-printet en fisk og hva slags colliders som skal plasseres på den og de forskjellige scripts, ha en generell convention (readme for hvordan det skal gjøres?)) **(UX)**
- Hvilke ledd er viktig, f.eks fisk, må den ligge rett i boksen eller kan den være krølla, etc? **(UI & Neo?)**
- Definere goal states **(UX)**
- Definere rekkefølge på flere objekter (f.eks 5 fisk i en eske legges på mest effektiv måte) **(UI & Neo?)**
- Demonstrere intermediate/required states for en task **(Neo & Droid & ux)**
    - Sparse rewards, men flere states som gir rewards
    - Eller ulike agenter koblet opp i rekkefølge?
    - F.eks. ta av korken på noe før man gjør neste task
- Bruke path / timesteps før rewards states for sampling og reverse curriculum **(Neo?)**
- Gjøre denne sekvensen på fisk, plukke opp og putte i boks **(Testing & example)**
- Negativ reward for å applye for mye force på objektet man jobber på, f.eks velte en stable med fisk **(Neo & ui)**
- Annotere steps kan være for flere goal states (delsekvenser) eller intermediate steps for sampling **(Droid)**
- Energy minimization for å unngå unødvendige actions **(NeoDroid)**
- Kan bruke sparse reward på goal state, men relativt til hvor mye energi agenten har brukt f.eks **(NeoDroid)**
- Objekter forhåpentligvis simulert så virkelig som mulig til virkeligheten, f.eks. fisk. dette for datasettet som skal brukes for "kamera" for opplæring i VR. **(Unity fysikk)**
- Begynne med Fisk og Boks
- Kan se på rigging og fysikk av fisk (få til å plukke opp fisk med fysikk) **(Unity fysikk)**

    - Sette opp kamera for å generere datasett når agenten klarer en task **(?)**
    - Age parameter for fisk? (forandre mesh og collidere følger med) fisk vokser forskjellig i forskjellig alder **(Unity fysikk & UX)**
    - Terminal state i fysikk på om det er noe som virker harmful **(Unity fysikk & UX)**
-

Referat mandag 28.01.19 kl 18:00-21:10
Ikke oppmøtt: Ingen
Gjorde ferdig prosjekt plan.
Definert kodestandard.

"Sees i morgen med mindre vi ikke gjør det"
Nikolai og Aksel, 2k19

Referat onsdag.30.01.19 kl 13:10 - 13:XX
Ikke oppmøtt: Nikolai (gyldig)
Sted: Mariusz kontor

- Gikk gjennom prosjekt plan, litt.
- Submodules are messy, having to copy files in some cases. Easier to just have a production branch. Submodules are better suited for libraries.
- Branches: master(duh), development, feature branches and hotpatch branches. Remember its possible to have local branches that you don't push.
- CI is a good thing that looks good to have in the project, profesjonalizm.

- Will probably use gitlabs scrum-board.
- "Even if you don't use something, write down that you researched it. You might use it in the thesis" ~ Mariusz
- Gitlab pipeline with linter to inforce code standard?
- Include gantt diagram, simply because the consor might want to see one. Even if it doesn't help us. It can be high level and vague. Have some milestones, like deadline.
- When writing the thesis, it's important to write about all the time spent doing tasks that are not visible in the end product, like fixing bugs, time spent developing features that ended up discarded. "the black time".
- Create a discord channel with a "with mariusz channel" and a "without mariusz channel".
- Check out SOLID. A methodology for organizing code. "A class should be a separate entity that only does one thing."
- In the thesis we should cover both architecture(vague) and design(more spesific), some of it we will do beforehand to help us plan.

Referat torsdag 31.01.19 kl 15:15-18:00

Ikke oppmøtt: Nikolai (gyldig)
- Begynte på design av annoterings system.

"For min del så går det find for meg"
Jone, 2k19

Referat fredag 01.02.19 kl 11:15-16:15

Ikke oppmøtt: Nikolai (gyldig), HALVOR E SEIN (surprise surprise!)
- Lagd repo.
- Startet å fylle inn backlog.
- Bestemt at det ikke var verdt det med CI (se Reflection notes for mer info).
- Sett på Git hooks, mulig erstatning for CI som da bare tester lokalt når man committer / pusher.

"Vi har jo alltid gode sitater."
Aksel, 2k19

Referat mandag 04.02.19 kl 12:00-14:00
Ikke oppmøtt: Ingen

- Første sprint start
- Diskutert angående conditions, sceneobject, evaluator
  https://drive.google.com/open?id=1da0E1GOWrypjoKqxYCX5KndagUwSCySu
- Namespaces: Playground og PlaygroundInternal
- SteamVR eller Unity.XR?

Ukens sprint backlog:

-

"TODO"
Aksel, 2k19

Referat tirsdag 05.02.19 kl 9:30-10:05
Ikke oppmøtt: Ingen
- Can use docker to host a pipeline on the server that builds unity using a script.
- Look at dash for documentation lookup.

"The first sprint always goes to hell."
Mariusz, 2k19

Referat tirsdag 05.02.19 kl 10:20-12:05
Ikke oppmøtt: Ingen
- Vi alle objekter som har conditions, også har observable. Men ting som har obeservable har ikke nødvendigvis condition. Eks bord(goal) eller nono-zone(obstable).
- Røff rekefølge features skal implementeres i:

Plukke opp ting i VR
VR UI for å sette opp conditions på et Scene object (også observer + eval)
Serialize og lagre keypoints
Lagre state av scene object og deres conditions med relationer.

Multiple goals chained.

Referat fredag 08.02.19 kl 10:00-xx:xx
Ikke oppmøtt: Ingen

Gått gjennom Conditions, og SceneObject.
Videre gått gjennom PandaHand og physics problemer med friksjon ved holding av et objekt.
Git hooks kort diskutert.
Verifyet tasks fra sprint backlog til done.

Referat mandag 25.02.2019
Ikke oppmøtt: Ingen

- Mer pair programming nå og fremover for å få alle på samme side angående Widgets, og få kommet litt igang med det
- NewtonVR ikke så mye forskjell på å holde objekter
- NewtonVR kan være et bedre kollisjonssystem likevel
- Evt. "hotswappe" to panda-hands? (En med rb på childs og en uten)
- MeshColliders er ganske shit, burde specifye for importing av models at det antagelig bør gåes over (evt. se mer på dette og finne en god standard og gjøre det på)
- Se på XR prosjektet til Christian og hvordan colliders gjøres der
- https://assetstore.unity.com/packages/tools/sacolliderbuilder-15058
- Observers kan puttes på Conditions
- Conditions kan potensielt kopieres om det skal annoteres på f.eks x antall fisker med samme type condition

"When you make prefab included mesh collider (Convex Hull) collider will lose."
Guy about SAColliderBuilder on UnityAssetStore, 2k19

"Det e jo potensielt en fisk for det."
Halvor, 2k19

Referat tirsdag 26.02.2019 kl 10:30-12:00
Ikke oppmøtt: Ingen

- Gått gjennom widgets mer, diskutert litt arkitektur og hvordan det skal funke i praksis

  "INTUATIVITET!"
  Jone, 2k19

Referat onsdag 27.02.2019 kl 12:15-18:00
Ikke oppmøtt: Ingen

- Pair-programmet mer av CuboidWidget
- Diskutert litt mer design rundt CuboidWidget

Referat torsdag 28.02.2019 kl 10:15-17:00
Ikke oppmøtt: Ingen

- Laget Activity diagram UML for annotering av objekter i scene

Referat fredag 01.03.2019 kl 10:15-13:30
Ikke oppmøtt: Ingen

- SubMenu

Referat mandag 04.03.2019 kl 12:15-14:00
Ikke oppmøtt: Aksel (DAMA PÅ BESØK)

Fant et potensielt problem med conditions:
Conditions på sceneobject som er avhengig av andre sceneobjects conditions. I.e. relative er en condition (sort of).

Komplekse dynamiske conditions som er dependent på states av andre sceneobjects - fish

Referat torsdag 14.03.2019 kl 12:15-14:00
Ikke oppmøtt: Ingen

Referat fredag 15.03.2019 kl 11:30-16:15
Ikke oppmøtt: Ingen

- Møte med Mariusz for første gang på noen uker
- Fått oppdatert han litt mer om hvordan status er
- Skriv på thesis underveis
- Les opp på liknende systemer som vi kan trekke noe ut av og skriv om på bacheloren
    - AI-systemer
    - Pathfinding?
    - Logic-based goal/terminating states
- Sett en hardcap rundt 4 uker før innlevering at det ikke skal implementeres noe nytt (med mindre det er noe veldig spesifikt og som man absolutt bør få gjort)
- Fokuser derfra kun på selve rapporten og eventuelt fikse/tweake systemet
- Rapporten er det som teller absolutt mest

- Reverse curriculum generation er hard

- Mange konsepter er vanskelige å definere og er relative
- Hvordan definerer man om en condition er en viss % andel ferdig?

Referat mandag 18.03.2019 11:00-13:00
Ikke oppmøtt: Halvor

- Venter egentlig på svar fra Christian om hvordan vi skal implementere reverse curriculum
- Tenker å lage en gameloop som bare instantiater trening på tross av å ikke ha reverse curriculum punkter så langt, så vi i det minste får en loop som vi kan expande og utbedre
- Treningen vil antagelig ikke gi mening og vil være helt random så agenten vil ikke nødvendigvis lære noe, men vi vil da kunne få opp en gameloop som vi kan jobbe ut ifra
- Dette åpner også ganske mange dører for hva som kan gjøres videre

Referat onsdag 20.03.2019 11:15-16:00
Ikke oppmøtt: Ingen
- Midway evaluation meeting
- Skrevet reflection notes rundt dette
- Fokus videre:
    - Expande ConditionEditors/Widgets
    - Få inn tablet/UI for conditions
    - Reverse curriculum kan hackes inn på ulike måter for en somewhat fungerende gameloop og et proof-of-concept og minimum-viable-product

Referat torsdag 21.03.2019 kl 14:15-18:00
Ikke oppmøtt: Ingen
- Jobbet med SubMenus

Referat mandag 25.03.2019 kl 12:00 - 14:00

Ikke oppmøtt: Ingen

<u>Møte med Christian:</u>

- API reverse-curriculum
    - Hans eksempler?
    - Interface mot neo? I praksis?
    - Når skal det instantieres?
    - Hvordan? Vi har ikke motorer?
- Memory leak neo? Tar mye ram?
    - Saving av "brain"?

- På Unity-siden klasse "Configurable"
    - Sette en verdi til noe og vil være persistent gjennom reset-sessions
    - F.eks sette en configurable til et målpunkt ved å sette actor sin transform der
- Setter Flag som sier at du ikke kan terminerer
    - Motorer kjører så random og gir noen randome states som kan være reverse curriculum
- Unobservables
- Python kaller et antall states han vil ha
- Droid -> states -> neo
- Neo -> reaction -> droid
- Ulike groups av states S1, S2, S3..
- Umulige states som at agenten ikke holder objektet den skal plassere et sted vil komme litt av seg selv, da den vil foretrekke de states der den finner ut at her klarer jeg oppgaven min f.eks. 90% av gangene
- Man kan bruke en brukers demonstrering for å generere tilfeldige states fra de punktene og på sånn vis på en måte ha en % fullført state
- State generering skal funke implisitt, neo setter flag om ikke terminate og ber om nye states utfra states som er suksessfulle

Referat mandag 03.05.2019 kl 11:00 - 14:00
Ikke oppmøtt: Ingen

<u>Møte med Mariusz:</u>

- Levere oppgaven på Inspera innen 20.05, kan kun leveres EN gang
- Move Boundaries to be a part of Project Scope (subsection)
- Academic background part of background
- Project Goals next to project scope
- Target Audience before project scope and goals
- Target audience also part of project scope (subsection)
- Design class diagram of the whole system from some level of abstraction where it is possible that fits within one page?
    - Class diagram of specific modules could fit more well in between text to discuss around the system architecture
    - Use (...) to show there is interface with the rest if needed, and just explain that this part is not important for what we are looking at right here
    - Showing niche parts of the system with graphical representations for modules / subsystems (this can describe it better), don't need to map directly to the classes
    - For concepts / abstract modular design don't need to necessarily follow strict UML. Can create our own diagrams and explain and describe them.
- Should explain how the Unity entity system works and how their constraints work and later on could say how it affects our system
- Todo's in UX / Implementation for design to Technical design
- Ask someone if they understand how a widget works after explanation, see if they understand how it works? If they do, the explanation and videos are well explained. Put bubbles and annotate things.
- You can put links to gifs in the thesis
- More figures for UI/UX components part for design, also for 2DWidgets even though we dont have so many

- Add bash for Programming languages in Chapter 5
- Change refs (to figures/tables) to using cref (cleverref), and make sure it displays if it is a figure, table, diagram, etc.
    - We can change format of cref on how it displays the ref (capital letters, not capital letters, etc, makes it easier when we want to change the formatting, instead of doing it manually)
- We can pull a local copy of the latex, and edit it there (if we want to search up certain words and parts to change it) and push up to the repository
- Don't have text in pictures too small or too big compared to the text in the thesis
- Remove cells that are empty in high-level use-case (like where something says: "none")
- Code snippets text should be smaller in the thesis
- Use figures where it can help explain something for the reader that is not self-explanatory, or just in general create a better picture of the concept for the reader
- If something within the picture is not self-explanatory, mark things and use "bubbles" and explanations

D Project Plan

The following is our internal planning document

Neodroid playground - project plan

1. Goals and Boundaries

1.1. Background

The client for this project is SINTEF. SINTEF's goal is to develop and apply technology for a better society. Among the projects that SINTEF has been running is Neodroid, an open-source, multi-component project, funded by *Forskningsrådet* researching machine learning in virtual environments, and the potential ways of transferring the learnt/learning AI into rea environments.

"The idea of Neodroid is to create a reality-ready robot brain in virtual reality. We specifically focus on creating a robot brain capable of humanoid visual-motor ability. Visual-motor ability is the integration between visual perception and motor skills. More specifically, it is the ability to perform constructive tasks integrating both visual perception and motor skills. The motivation behind Neodroid is to enable robots to assist humans in performing such tasks."[1]

The Neodroid platform as it is today has a number of functionalities that can be used. "Neo" is an interface for communicating between the developer environment: the Unity 3D game engine, and the Machine Learning implementations (the "Agent" module) in a python backend. The "Droid" module consists of a number of components built to organize "observations" and "motor actuators" and to communicate seamlessly with Neo, which allows developers to construct virtual machine learning environments.

The project is still heavily in development, but SINTEF has started looking outwards at hFPSow Neodroid should be utilized for general purpose robot teaching/learning. - This will be the basis for our project.

1.2. Project Goals

The ultimate goal for SINTEF with the Neodroid project, is to have made a framework that can be used by staff in industries in many different disciplines to demonstrate how a given task should be performed, and then have a robot trained in this task during the following days / hours.
Our goal can be seen as a step towards this, as we limit the issue to the demonstration of a few generic tasks/problems/goals for the robot to learn to deal with initially (and when we've gotten

[1]Neodroid Documentation - Project Idea (02.11.2018) Link (25.01.19)

that working we may go on to more complex, specific tasks/problems/goals). As this will be a first attempt at generalizing how the framework can be used, a large portion of our contribution will also be to develop a standard for the robot interactables.

1.2.1 Business goals

- Create a standard for components used in a Neodroid virtual playground, such that a technician can build the needed playground with ease, and let a demonstrating user and virtual robots interact with it.
- Create an user interface to demonstrate a task/tasks to a virtual robot, in order to show the demonstrating user that the robot is working within the correct constraints of the task
- Create virtual objects that mimic real objects as closely as possible in behavior and appearance. One of the focuses will be to model dead fish as one of these interactable objects, as this is one of SINTEF's own uses for this software.
- The robot and the playground should be possible to recreate in reality and physically test.

1.2.2 Impact Goal

- Generalize how robots are used in industry - if a robot can learn and do any number of different tasks, it will mean that you do not need to construct hard-coded specialized robots.
- Reduce damage potential for people and equipment in the context of automation in hazardous work environments - a robot can be taught in complete safety using the virtual training environment

1.2.3 Learning Objectives

- Apply machine learning for a practical purpose in a hybrid reality (real and virtual world).
- Expand our knowledge of UX / UI in a VR environment in an application context.
- Use of good coding standard in a publicly available product.

"The vision behind Neodroid is to teach robots to accomplish tasks that combine vision and motor skills, by training the robots' brains in VR" - Norway's Research Council's page on Neodroid.

1.3. Boundaries

The application we will develop will be made in the Unity 3D game engine, as specified by SINTEF since their existing framework is made in this environment.

The Neodroid platform is the system developed by SINTEF which we will make a Unity application for. It's not a main focus to extend the Neodroid platform's functionality. However where applicable, we are encouraged to improve it. The main focus will be on creating a Unity application with a VR playground where the user can demonstrate a task. The playground will be capable of generating a huge dataset, which in turn will be used to train an AI to perform the demonstrated task. The optimal outcome would be to have this playground integrated as a part of the Neodroid platform.

Performance is important both for the playground part of the application and for the simulation section. In VR, it is important that the framerate is more than 90. If it goes below, the VR headset enters a safety mode where it reduces framerate to 45. This reduced frame rate will make the user nauseous (VR sick / motion sick).

For the simulation part of the application, performance is also important. All the objects we make in Unity that mimic reality must run fast so that the simulation can go as fast as possible. Training the AI faster as more instances can run in parallel.

2. Scope

2.1. Subject area

Virtual reality

Virtual reality (VR) is a fairly new space to work in, and good user input methods are constantly evolving. The standard for VR headsets is constantly being changed, and because of that software made for VR must also adapt to these variations in the standard, in comparison to other markets such as mobile. One example is that some of the VR headsets motion controls have a touchpad (Vive) while others have an analog thumb (Oculus).

User-experience (UX) and user-interface (UI)

User experience in VR is an additional design challenge. User interfaces in VR must be designed very differently than any other mouse and keyboard applications. The user must also be comfortable with the environment in which they are placed in. Fear of heights or claustrophobia must be taken into account when designing the environment.

Reinforcement learning

Machine learning is a central theme in the playground as Neo is based on this. Reinforcement learning is a subclass of this and is based on either rewarding or punishing an agent for its actions. There are several principles and concepts that can be used to apply machine learning, but the desired technique for our task is to use "sparse reward" where you either give zero or 1 point when the agent has successfully completed the task after a given time. This simplifies the concept, but results in a high probability that the agent will not find the right solution. One of the solutions to that problem is to use "reverse curriculum" that initializes the agent with simple environments at first with very little steps required to get to the goal state, and to further make it more and more difficult for the agent to solve the problem over several iterations.

Simulate / emulate physics

In the playground, the robot should learn to interact with a number of objects that will behave as realistically as possible. In some cases, it may be relevant to simulate their properties, such as soft-body physics to determine the behavior of the object in order to make it as realistic as possible from the AI's perspective. In other cases, using logic may be more effective and accurate to determine the wanted behavior instead.

In the context of the task, a big part of the challenge is to create an intuitive user interface that allows users to easily demonstrate tasks by defining target states. In addition, we should be able to define when a demonstrated goal condition is satisfied, whether it is implicitly defined from the demonstration, or whether it must be explicitly defined, and if so, how it can be defined explicitly.

2.2. Limitations

The Neodroid platform is intended for use in academic research, specifically for the developer's own interests. Our task will be to streamline the use of the platform with regards to the construction of a playground, and user interface. The goal is to make it easier to go from a certain issue, to the implemented solution as quickly as possible.

2.3. Task description

The task is to create Neodroid playground in the Unity engine that to some extent should be able to generalize how objects in the playground are to be instantiated to teach a robot to solve a specific problem. However, thew construction does not have to be done in VR. Although it is a nice bonus. The flow in this goes from the construction of the playground and its components either by using the Unity's Editor, or constructing it through the VR environment. It is desirable to create a standard for different types of objects, and the components they need to function optimally within the playground. The playground should then consist of a set of units (such as the robot, objects to pick up, storage boxes).

Furthermore, after a playground has been constructed, the user should be able to easily demonstrate a task that the agent should be able to learn. In practice, this will be done by the user themselves annotating the objects and positions that are important and relevant for the task to be performed correctly by the agent. This will either be resolved with just a goal state, or could also potentially take several important “intermediate” states, which will be used to generate the data set for the reverse curriculum. It may also be possible to define multiple goal states for the agent, depending on the task. If the order of these states is important, that should also be considered by the agent.

When annotating objects and points in a demonstration, one should be able to emphasize which observations are important for the goalstate .For example, rather the end position of the target object is important, or rather the rotation also has to be within a threshold. Then these constraints should be annotated and taken into account. The process of defining small precise targets should also be as intuitive as possible for the user, and optimally allow editing of several object's properties at once.

When it comes to the machine learning part, sparse rewards and reverse curriculum should be used to achieve the desired result. In order for the agent to be able to learn how to optimize the result, one can also use some sort of energy minimization to avoid unnecessary steps. While the agent is crashing into objects it is not supposed to interact with, or doing unwanted actions, it should be given a penalty and terminated.

2.4. Restrictions

Documenting or refactoring existing code in the Neodroid platform is not a main focus of the task. However, where applicable we are encouraged to do so.

It is not part of the task to expand Neo (the machine-learning component of the system), even though if we see it necessary we are free to expand on it.

Our task will not be to train a real robot. - We do not have access to a physical robot to try this, but for the client it will be easy to do this, because the Neodroid platform have components that can easily be inserted into our environment to further educate the physical robot. The task within the virtual playground will be only operating with perfect data for learning.

3. Project Organization

3.1. Development model

The development model for the Neodroid playground application is SCRUM, we will have a sprint period of one week. Weekly sprint meetings will be on Mondays where we write a short report on last week's sprint, and discuss which tasks have be moved back to the backlog.

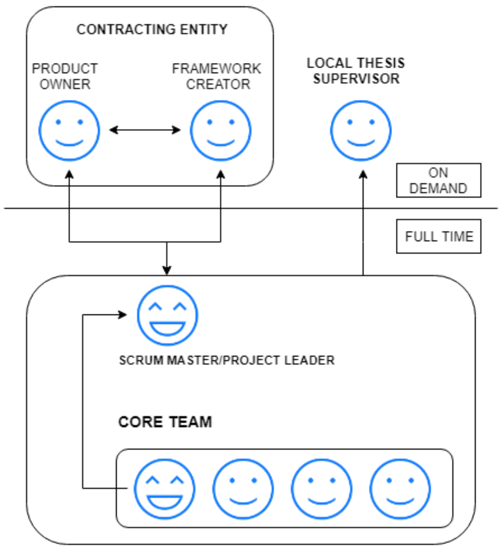
We then discuss which tasks are most important to do this sprint, and distribute them to all members and register them in our GitLab scrum-board. We write a brief summary of what decisions were made in this meeting so it can be referred to later.

Tuesdays are going to be our dedicated day for consultant meeting with our supervisor. Every other week we will also include our Sintef contacts in the consultant meeting in order to brief them on our progress and ask any questions about what direction the project should take.

3.2. Responsibilities and roles

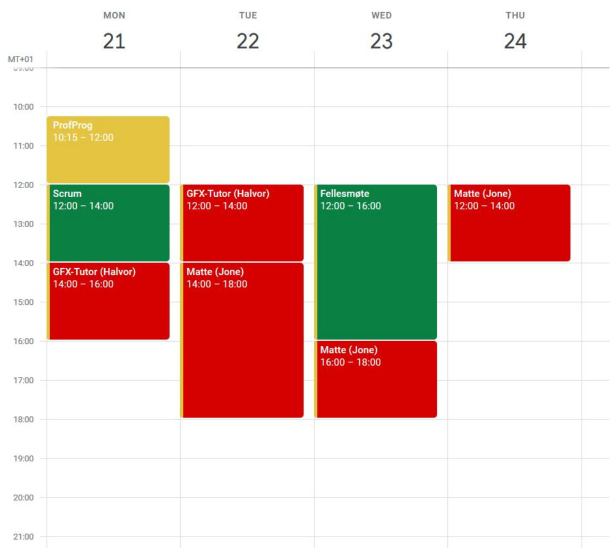
Client - SINTEF Ocean
Platform developer / designer - Christian N. Heider
Product owner - John Reidar Mathiassen and Jonatan S. Dyrstad.
Project leader - Halvor B. Smedås.
Supervisor - Mariusz Nowostawski.
SCRUM roles:

- **SCRUM master** is Halvor B. Smedås.
- **Log writer** for the project is Jone Skaara who will have the main responsibility writing reports and note important decisions and gather information in an organized manner.



3.3. Routines and rules in the group

- Agreed workload - 30 hours / week (at least 5 hours each weekday).
- Time schedule for group work
- Write weekly sprint report after each sprint is completed.
- Write report after each meeting. Both after meetings with supervisor and product owner, and after internal group meetings.
- Documentation and work on logs relevant to parts of the bachelor's degree are worked on by each group member where it fits along the way in the project.



Fixed schedule for each week

3.4 Technology

3.3.1 Digital tools

- Microsoft Visual Studio Community, and - Enterprise Edition - Integrated Development Environment.
- Unity 3D - Component-based 3D game engine.
- Git - Version control (Local running GitLab server (prod3.imt.hig.no))
- GitLab scrum-board for assignment of work tasks.
- Blender - 3D modeling
- Toggl - Time tracking tool.

3.3.2 Programming languages

- **C#** will be the main language we program in, this is because C# is the Unity standard, and because it is the language that gets the most support from the developers and users of Unity
- **Python** is also a current candidate for certain parts of the project that involve interacting with the machine learning module (Agent) or the communication link (Neo) between the machine learning algorithms and Unity. - The reason for this is that these modules are all written in Python, so something else would be much more laborious, besides Python is recognized as one of the best languages to work on when it comes to artificial intelligence.
- **C++** Can be relevant in cases where we need high performance when a lot of data is processed.

- **HLSL (Compute shader)** can also be relevant in the same case as above, depending on if the problem can be parallelized
- **ShaderLab**, **CG**, **HLSL**, **GLSL** are also good candidates for languages we can end up using as it is not unlikely we need shaders for visualizations in VR.

3.5. Group policies

§ 1 Money and Expenses

1.1 - For expenses that exceed what is covered by the client and NTNU IDI, the expenditure (-e) is shared evenly between the group members, unless otherwise agreed between the parties involved.

§ 2 - Illness

1.1 - If you become too ill for attendance during the project, you must communicate this to the other group members.

1.2 - If the scope becomes so large that it slows down the project's progress due to dependencies in modules in the project, there should be a group meeting on how this should be solved, this should also be with the supervisor.

§ 3 - Scrum

3.1 It is up to each member to report a lack of sprint goals if one completes all the sprint goals they were given at the beginning of the sprint period.

3.2 - It is up to each member to report having too much work assigned them as early as possible, if it is obvious that it will not be completed, the other member should be informed.

§ 4 - Meeting

4.1 - In case of any disagreements in regulations / polls, the group leader two has votes, so that we can always end up with a majority.

§ 5 - Digital tools

5.1 - All Git commits must follow a standard to clarify the changes that have been made. This standard can be customized for three different cases:

1. Hotfix - for line changes and similar little things (these should generally be avoided): "HOTFIX - added missing semicolon".

2. File change - For changes that have a strong connection to one or a few files, file names must be listed before a summary of the file change. "ChangedFile.cs - added new function to satisfy issue # 37."
3. Larger change - usually occurs after refactoring code or the like. “SPAWN SYSTEM - major rewrite to a more data-driven approach on things”

5.2 - Communication channel is Discord.

§ 6 - Documentation

6.1 Each class and function must be documented using. C # xml Docs[2]

4. Organizing of Quality Assurance

4.1. Coding standards

We will follow microsoft’s coding standard for C#.[3]

Exception

LINQ syntax, where we will also use extension methods directly instead of using full query syntax.

4.2. Configuration management

The software project should be (to the best of our ability) kept in a state where it builds and executes. It should always be possible to retrieve the last commit from the project, build and run it. In other words: pushing to git should only be done when the code has proven to be runnable.

SCRUM increments, and the backlog will be used to identify what has changed, and can therefore easily be used to write the change log for each increment. Also, the sprint period is so short that commit messages during the sprint can be used to more explicitly state changes.

In this project major features will have their own unique branch. Branches should be removed after said feature becomes integrated with the main or production branch. We will also validate along the way by continuous integration (CI) which will check that the modules compiles after they have been pushed to the remote.

[2] XML Documentation C# - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/xmldoc/xml-documentation-comments

[3] Dot Net - C# coding conventions - https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions

Risk analysis

Feature creep
Probability: Low
Consequences: Medium
Description: When the task is very open ended, there is a danger that we may be exposed to feature creep.
Mitigation: Strict framework for SCRUM tasks and no feature list expansion without completing the tasks already defined.

Performance Issues (VR)
Probability: Low
Consequences: High
Description: When using a more advanced physics engines and expanding code, there is a danger of performance issues. Something that will have major consequences in VR. Fortunately, the need for a VR user in the system is minimal and therefore will not require much overhead in the terms of processing.
Mitigation: Good code standard and pair programming if needed. Use of Unity's profiler to find slow parts of the code.

Problems with implementing annotation of transform targets in VR
Probability: Low
Consequences: high
Description: Annotation of transform targets through VR should be relatively trivial to implement but may lead to problems in the project if implementation takes longer than expected
Mitigation: Good design and planning of annotation system.

Problems with implementing annotation of general component targets in VR
Probability: medium
Consequences: medium
Description: On the same line as transform, the goal is to annotate the state of other components. The idea is that one should see similarities in the target status of components without being directly defined by the user, but interpreted through VR.
Mitigation: Design and planning of annotation system. If this devours too much time, we will abandon interpretation in VR and rather be explicit in the device menus and / or config file system.

Problems with implementing annotation of relationship targets in VR
Probability: high
Consequences: medium

Description: Goals should also be relevant to several objects that have a relationship. This can create great complexity and prove to be a big task to cover during a bachelor's degree
Mitigation: Design and planning of annotation system. If this devours too much time, we will abandon interpretation in VR and rather be explicit in the device menus and / or config file system.

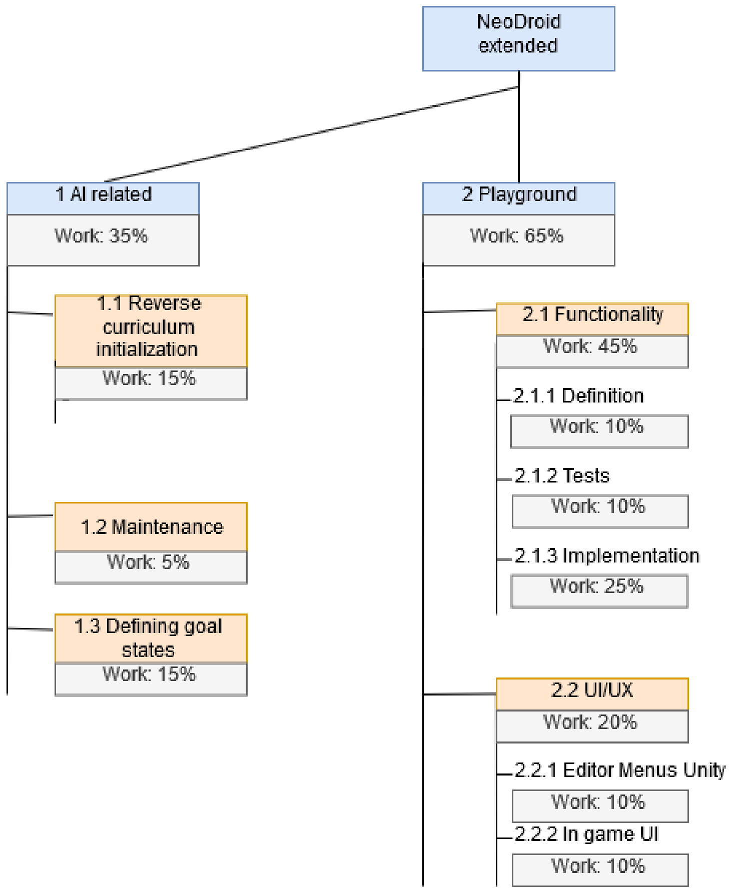
Unity version mismatch
Probability: Medium
Consequences: Low
Description: Updating to Unity introduces new issues or instabilities that need to be resolved.
Mitigation: Version control, change unity version.

Major changes in neo or droid modules cause conflict with our code
Probability: Medium
Consequences: Medium
Description: While developing we might have to change things in either neo or droid that will be conflicting with the external changes done to the Neodroid platform. Changes with neo or droid might also cause conflicts with our internal code.
Mitigation: Communication with the ones responsible for the framework.

Soft body physics is too heavy to drive or is too difficult to integrate
Probability: High
Consequences: Low
Description: If soft-body physics is too slow or too difficult to integrate into the project, it cannot be used in the project.
Mitigation: If it does not work, the branch for this feature is discarded.

5. Plan for Implementation

5.1 Work Breakdown Structure



5.2 Milestones and decision points

- **“Getting Started”** - Make a simple project to familiarize ourselves with the Neodroid platform

- **Game loop** - Before we start working on features, we want some base functionality in our Unity project.

- **Standard for Neodroid Playground Interactables defined** - A standard for further developments, probably in the shape of interfaces and super classes.

- **“Cube”** - Have a simple goal state described/annotated. A box with *transformational* constraints.

- **Reverse Curriculum Starting Point** - Able to pick out meaningful reverse curriculum data sets for the agent.

- **Midway Pivot point** - Midway through the project we will analyze the progress in comparison with plan in order to pick up on discrepancies between reality and plan, such that we can refurbish the plan and thus appropriately scale our project.

- **Simple Joint-based Interactables** - e.g. fish with bones that deform the model.

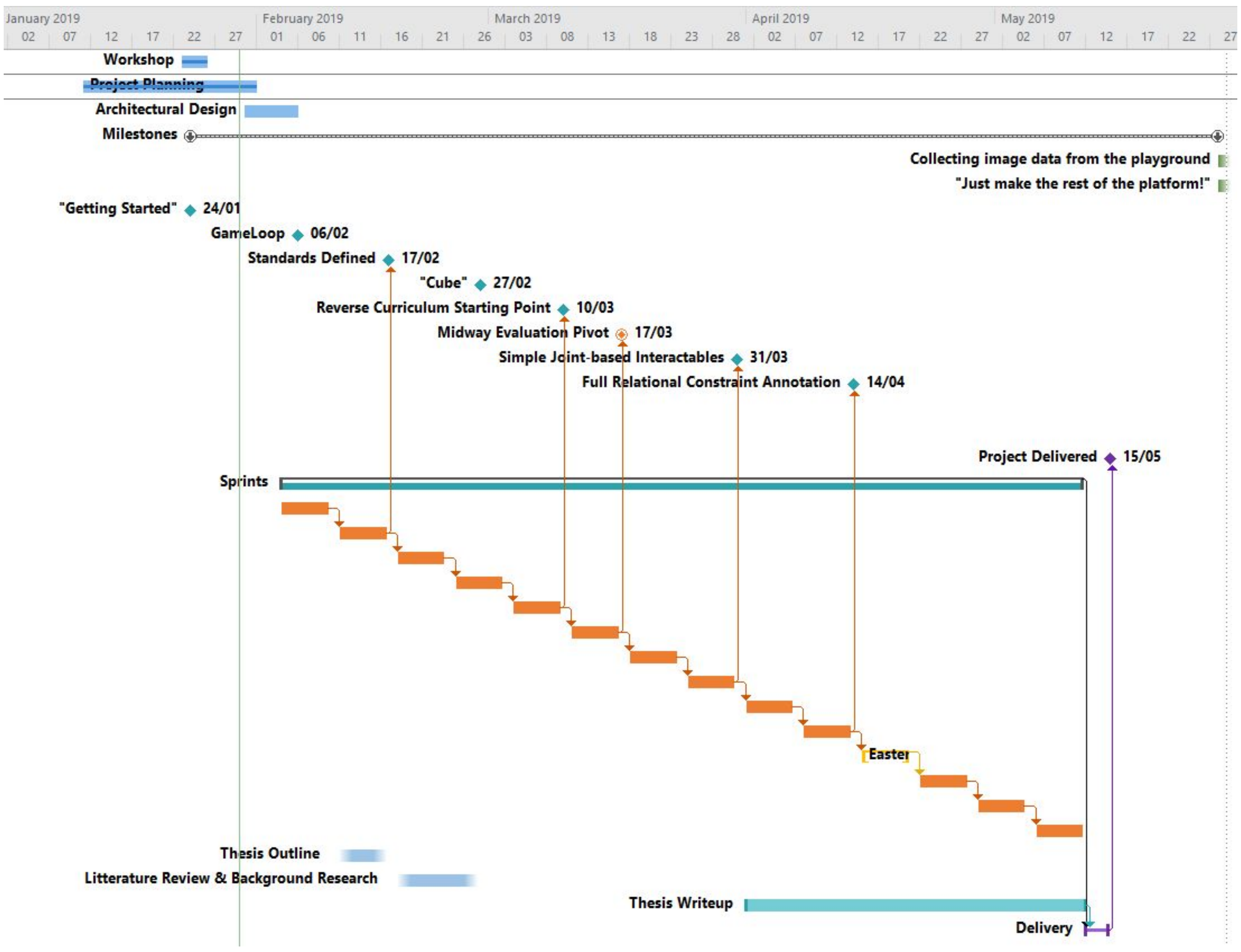
- **Full relational annotation support for VR -** Be able to select target state attributes for each object in the playground after a demonstration without leaving VR.

- **Project Delivered** - Delivery of the thesis

- **Collecting image data from the playground** - Past delivery (probably): Gather a large collection of pictures from the playground to be used for teaching the physical robot.

- **“Just make the rest of the platform” -** Past delivery: further developments.

5.3 Gantt Chart



E Reflection notes

These are notes we took under development discussing different aspects of features we were working on at the time.

31/1/2019:
The problem of annotation of constraints
What is the constraint relative to?

Must different parts of a constraint be relevant to different objects?
Example: A fish must be put in a box (positional constraint relative to the box's position), but also it should be stacked in a certain way with the other fish in the box. This means it has a secondary constraint that is only relevant and relative to the other fish in the box, and here one must consider both position and rotation relative to several fish.

How to easily and intuitively specify this in VR?

- *Interpreting user gestures and convert to logical constraints:*
    - *Hard to do*
    - *Error prone*
- *GUI*
    - *Causes user friction*

How does the demonstrating user define the tree structure of the goals?
goal is reached if (fish rotation && fish touch table) || *fish inside volume*
this logical structure must be defined by the demonstrating user after the demonstration, and must be intuitive enough that farmer frank can figure it out. One way this could be made intuitive is having a list of all conditions in the scene. All conditions in this list all has OR relations to each other by default. Dragging one condition and dropping it onto a second puts both in a new AND-condition-containter(similar to dragging a smartphone app icon onto another to create a folder).

Symmetry is basically just a set of two rotation constraints with an OR relation.
The fish must lay on its XY symmetry plane = the fish must have Z rotation 0 OR Z rotation 180.

1/2/2019:
SceneObjects, evaluator structure:
We decided to have the evaluator in a simulation instance update all its SceneObjects, the SceneObjects will update its conditions. When a SceneObjects goal state or termination state occurs, it is returned to the evaluator which gives signal and resets the instance. This a a good approach because the evaluator can choose which of the Scene objects needs to be evaluated, instead of having Unity call their update() every frame.

CI
We have decided to discard CI for now as it will result in too much time consumption. We will use git hooks to do some basic unit test and probably check that coding conventions has been met.

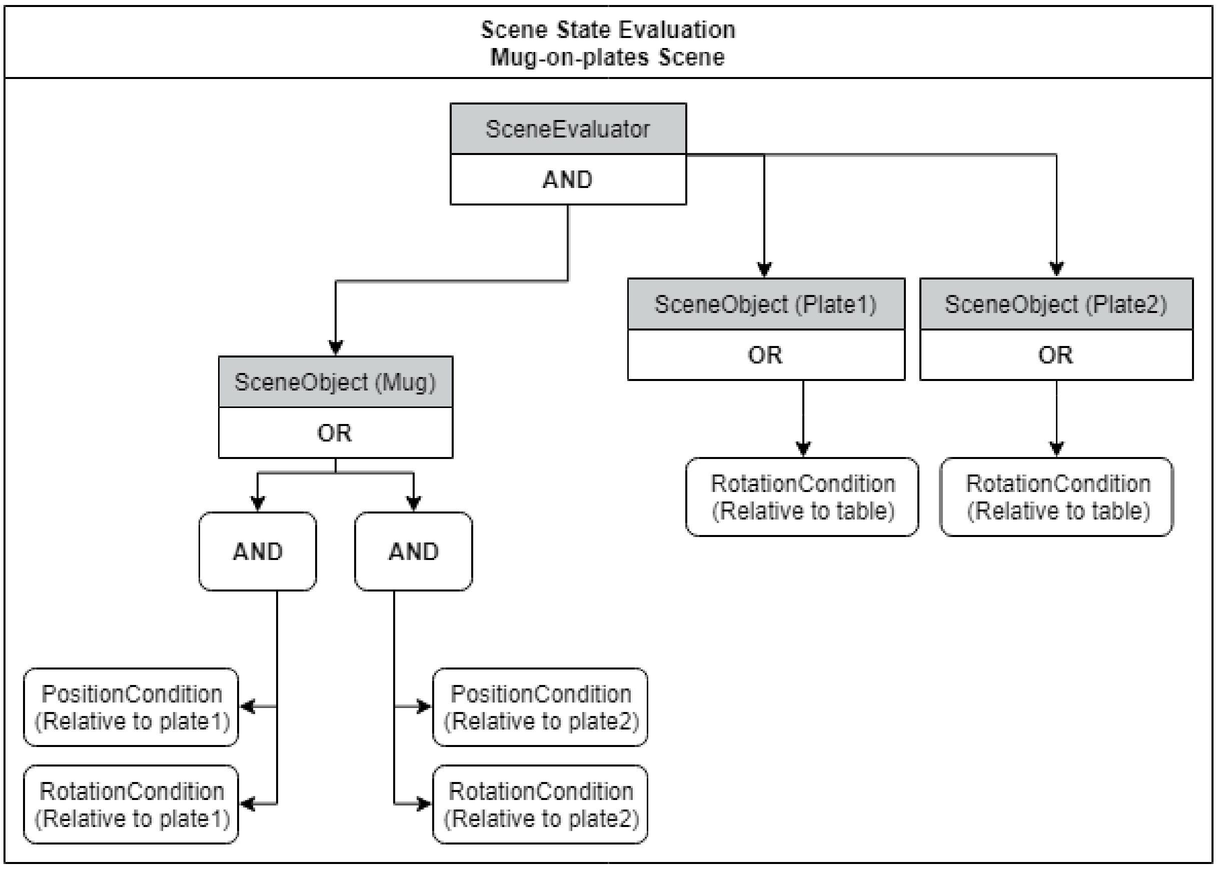
5/2/2019
Git hooks
https://git-scm.com/docs/githooks
https://stackoverflow.com/a/10929511

6/2/2019
No build on hooks
Reasoning: build on hooks would create a lot of friction on the push process. And most times you have already built the project manually. In contrast with having a pipeline CI where the build could fail after being pushed and then having the pipeline revert automatically, the hooks seems too cumbersome to make with little reward.

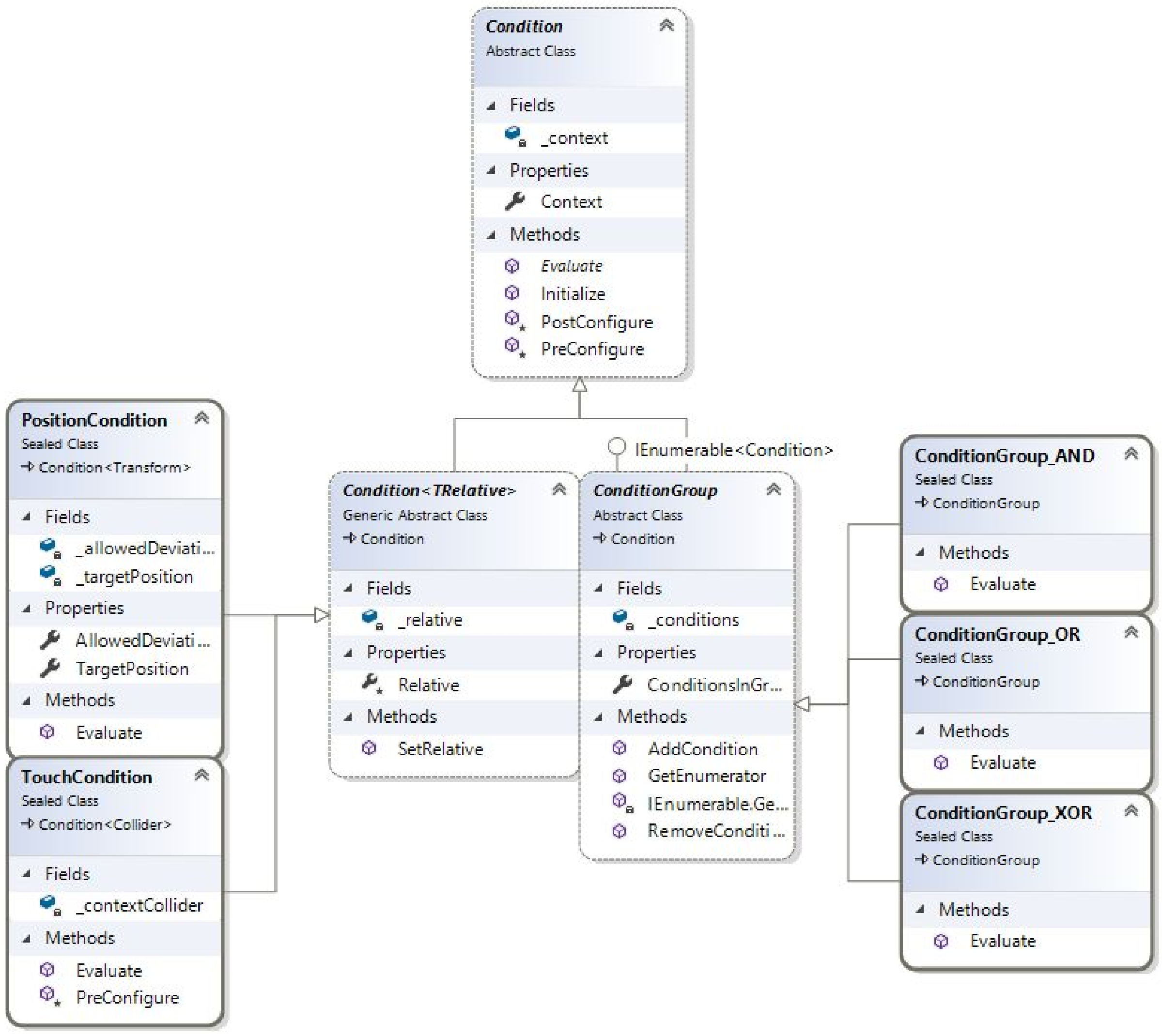
Resharper CLI lacks documentation

13/2/2019
Condition Based States
We've come to design our scene states through the use of *conditions*.
Conditions are encapsulated properties of the scene's objects. It's base interface is designed to be flexible towards all object components, allowing us to easily create modular state definitions. An example of this is: you want a coffee mug to stand on a plate. - There are several properties to consider as part of such a state: the mug's position relative to plate, the plate's orientation, the mugs orientation relative to the plate, the size of both (arguably this remains a constant, and so it might not matter too much).
All of these properties can be mapped as conditions in our system. As a condition is placed on an object, we consider the object to be the *context* of the condition. There's also the matter of relativeness on some conditions: "the mug's position relative to the plate". We've created the condition base interface with this in mind, and so we allow a condition to have a generic *TRelative* (i.e. a Type that is somehow relative to the context in some way for the given condition), with the only constraint on TRelative being that it's a *Component* type. This allows for greater flexibility within the condition, as we can create evaluate relations between the context and the relative objects' component (of type TRelative).
Going back to the mug-on-a-plate example; we can set up a Condition for position, with the TRelative being Transform (`class SomePositionCondition : Condition<Transform>`), such that we can do an evaluation of the relative position within our `Condition.Evaluate()`-function.

Having this functionality we can already loop over all conditions in the scene and in that way evaluate whether or not the scene is in a goal state. This would assume however that we either needed to consider all conditions essential for the goal state, or simply have consider it a goal state when one of the conditions were met. The first of the two would probably make sense in some demonstrations of goal state. Especially in cases where there's only the one goal state. But in cases where there's more than one goal state, our condition system would be able to map these goal states.
For the sake of giving examples; let's say we could've placed the mug of either of two plates. It might be possible to make a whole new type of condition, but that's not ideal as we want to cover the bases with some more generic ones for the end user.
No, instead we introduced *GroupConditions* to our system. Group conditions is an implementation of the composite pattern in our condition system, where a group condition is a collection of *n* conditions. We've defined a few concrete group conditions, all based on logic gates such as AND and OR, where the `Condition.Evaluate()`-function evaluates the conditions within with respect to the logic gates' definitions. Using these group conditions the scene state ends up being a tree structure made up of conditions.


Scene State Evaluation
Mug-on-plates Scene
SceneEvaluator
AND
SceneObject (Plate1)
OR
SceneObject (Plate2)
OR
SceneObject (Mug)
OR
RotationCondition (Relative to table)
RotationCondition (Relative to table)
AND
AND
PositionCondition (Relative to plate1)
RotationCondition (Relative to plate1)
PositionCondition (Relative to plate2)
RotationCondition (Relative to plate2)

And, as seen above, we can now manage to set up multiple goal conditions. The scene evaluator behaves just like an AND-grouping condition does - i.e. loops through all conditions.
SceneObjects however may still change...



15/02/19

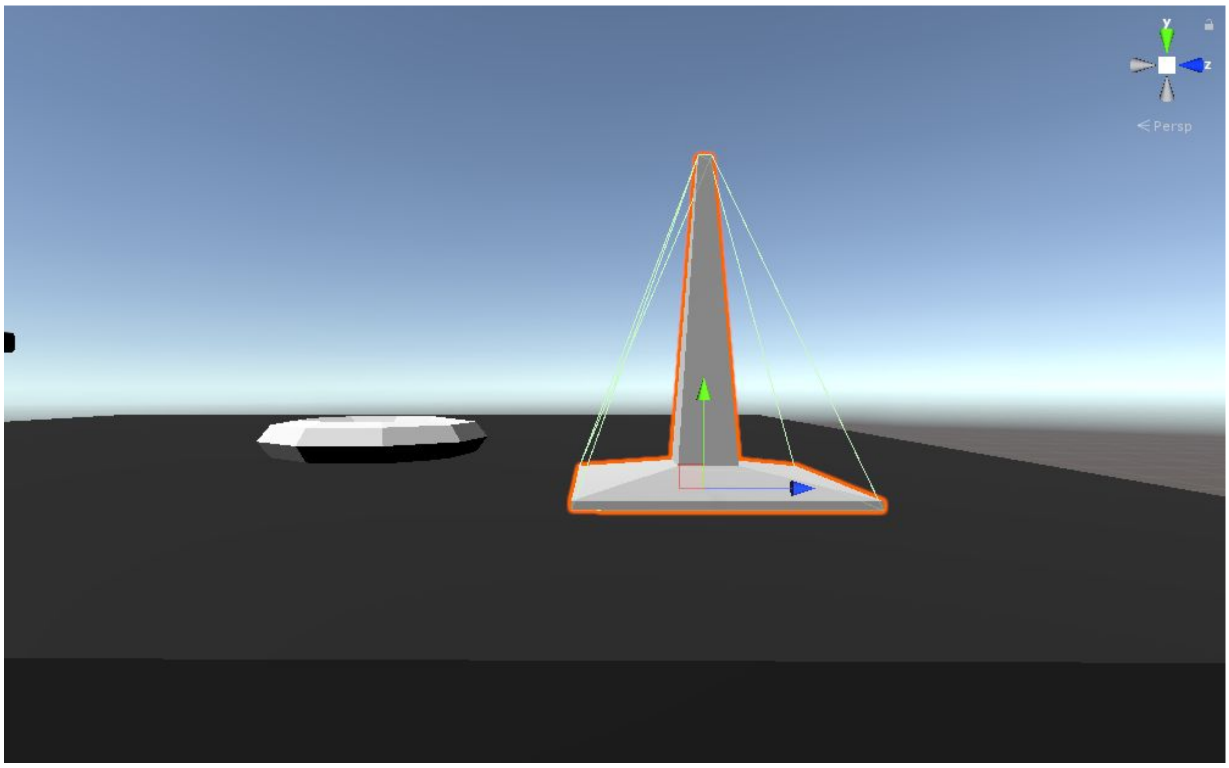
Test-driven development for conditions to avoid issues later on in training as condition are essential to work for training.

24/02/2019

Complex models and problems with MeshColliders

Our whole system is supposedly built to represent the reality to the extent it is possible. Whether it is about collisions of objects, general physics or just how the world looks. This is important for several parts of the system to give a wanted result. First of all you have the programmer/designer that is supposed to make environments that make sense, and hopefully this process should not be too tedious, it should be relatively quick and easy to setup an environment to teach a robot a new task in the Neodroid Playground. Secondly, there is the person or expert within the certain field that is supposed to demonstrate the task the robot should learn. For this person one needs the environment to match the reality as much as possible, if not it would cause flaws when that person is demonstrating the certain task. Either it would cause the demonstrator to handle the task in an unnatural manner, which would not feel good for the demonstrator, and possibly cause issues when it comes to demonstrating the task correctly when considering what the wanted result is. And lastly, when this task is supposed to be learnt by a real physical robot with the usage of VR cameras, the camera data must pretty much match the camera data a real physical robot would put out. This last part is probably the most vulnerable part. As if the VR robot could seemingless carry an object despite there being thin air between the robot's arm and the object it is carrying, how would the robot know that wouldn't work in the real world? The problem is, it wouldn't. So these are issues that have to be carefully thought about and things we have to discuss and design carefully when considering this whole task.

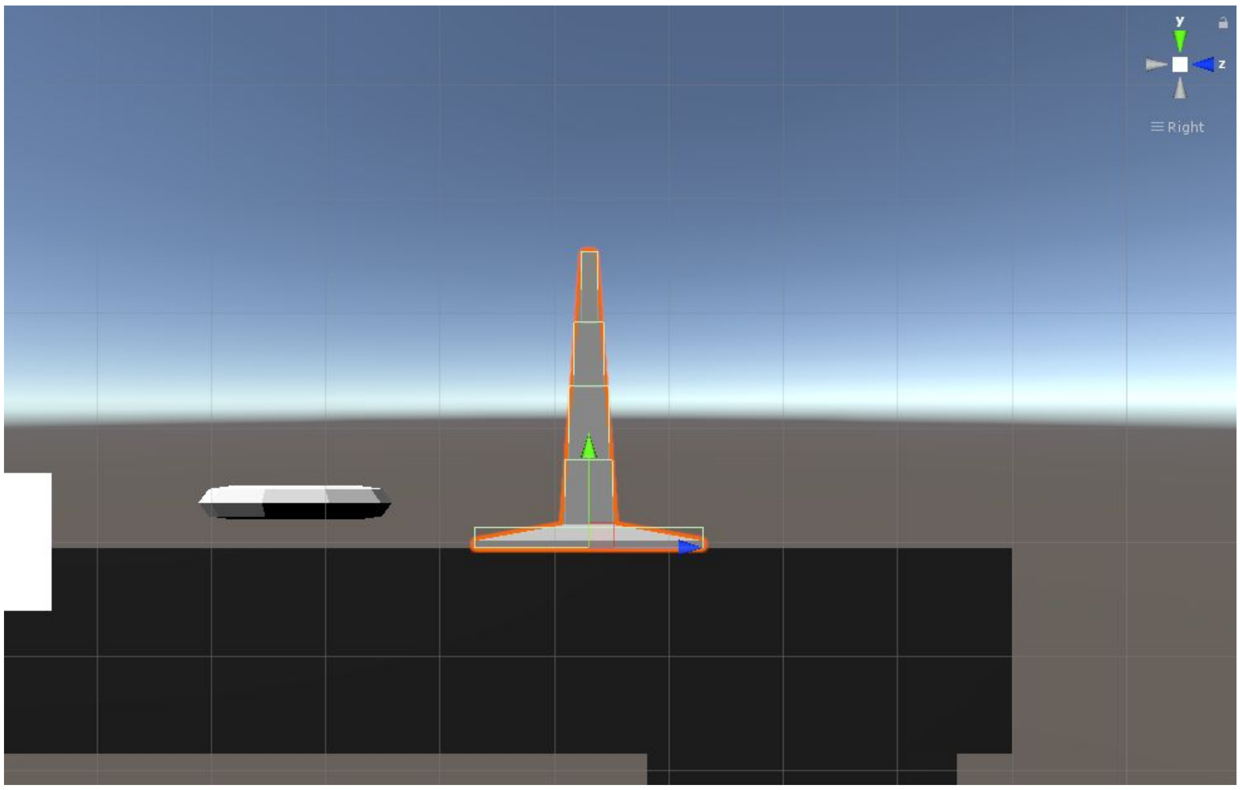
There are several issues we have already met when it comes to this, one of which is Unity's built in MeshColliders for more complex objects than their primitives. As these MeshColliders are not necessarily accurate at all in terms of how an object looks and how it's collision detection works. This means that by using MeshColliders as is, it would at times detect collisions when it is not supposed to, or other times not detect collisions when it is supposed to, the first more than the latter though. This is ofcourse not a problem when working with simple object primitives, like Unity's built-in objects like spheres, cubes, etc. All of these primitives have matching colliders that work really well, and they match the reality as good as possible for a virtual environment. But for complex models, this is not necessarily the case with using MeshColliders, it is just an approximation of the models vertices and the game-engine makes an assumption of how the model is built up and how it should interact with the environment. As we are working in Unity, for a lot of games it doesn't matter too much if the collider doesn't exactly match up with the model, players may acknowledge it or they may not, but it usually isn't gamebreaking. However, the system we are working on is supposed to match how the object would act in reality, and this can cause some real issues. Let's look at a fairly simple example:



Here we have a pole, made in Blender and imported into Unity, with a torus next to it that is supposed to be placed on this pole. A pretty simple object, and also a pretty standard Use-Case one would consider for this system. Where a technician makes these models for someone that wants a task to be simplified with a robot being able to do it, and the flow of it would possibly be something like this. The technician will import these models and most likely put a MeshCollider on this object. However, as you can see, the MeshCollider is not as accurate at all as one would like it to be. (The collider is visualized by the green lines on the object)

Basically for this case, the task that is supposed to be demonstrated, would not be possible to demonstrate correctly, nor would the AI learn properly by the images generated after this task would be done by the AI. As a matter of fact the wanted goal would probably not even be possible to get to in the virtual environment. Yet, this is a pretty simple object, but it shows pretty good how flawed the MeshColliders can be, and why it is something that needs to be acknowledged when designing and considering this system as a whole.

So, we know that MeshColliders is an issue, but what are the alternatives?
One thing we do know that can be a way to go about the issue, is to try and construct more accurate collision detection by building up the model with a whole lot of primitive colliders, to try and match the model to the extent it is possible. This would be by using BoxColliders, SphereColliders, and CapsuleColliders. Now, let's look at how that could be done for this object:



This is how the object’s colliders could look like if one built it up with using BoxColliders. As you quickly can see, it certainly represents the object more accurately, but yet not really as accurate as one would like it to be. It is far from what you need to get the object to behave truly realistically, but it is something that at least could work and the task possibly could be taught somewhat correctly to the AI. However, it is also worth noting that it is a really tedious process to set up these colliders on a object, yet given just a simple object like this, it can be really hard and tedious process, because you want it to be as accurate as possible. So one can easily imagine how much harder it gets if you have more complex objects than this simple example. Arguably could an object with more detail have a better looking MeshCollider from Unity, however that may not be the case, and MeshColliders are also more performance-heavy.

So as one can see, there are a lot of issues that needs to be handled and acknowledge already just for simple problems like this one with the “Torus on a pole”-issue. And of course that is because the simple fact that it needs to be as close to reality as possible, this is not just colliders for a game, but colliders that will affect whether a physical robot will be able to correctly learn a task or not.

We are happy to be able to see these cases pretty early in the development process, so they can be considered, and we can have internal discussions around these issues early, and possibly come up to some good solutions to these issues. A lot of these issues that come up with the system, has been brought up by our own Use-Cases and what we want the system to be like. We put up some user-stories also to describe parts of our system, and with these user-stories we can create these test-scenes pretty early on so we can see what issues they present.

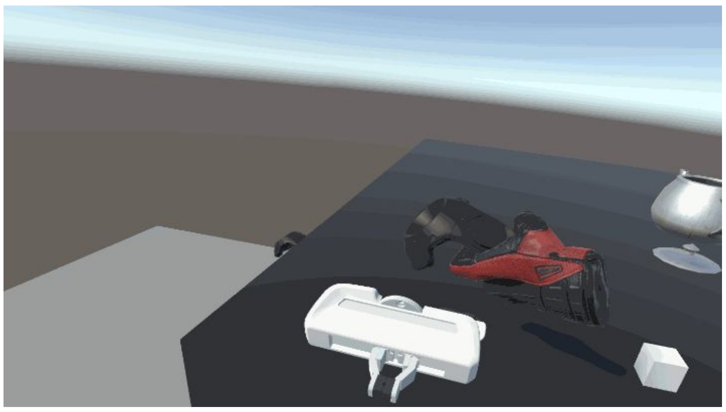
2/28/2019
Minimum viable product: We define minimum viable product as being able to customize scene to define conditions and relations of objects such as the scene can reset and reward/punish agent without human interaction after the annotation phase is done.

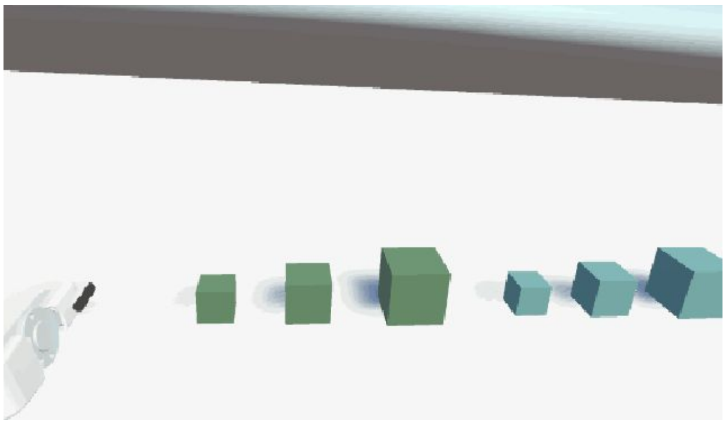
- User enters created scene
- User applies conditions, relative and logical relation between conditions in vr
- Save the conditions to disk and edit them later
- User initiate learning phase
- Conditions become persistent between scene loading
- Scene can reset for learning in a proper manner (meaning agent is capable of learning correct action from start state)

2/28/2019

Grabbing with physics:

The grabbing physics is very core to the goal of the project, and a fatal part of the implementation. Since the focus is to have the grabbing work as realistic as possible, using object parenting is out of the question. There are several ways to go around this problem.





The first way we tried was to rely on unity's friction that applies forces to objects were both surfaces have rigidbodies and colliders. The problem is that SteamVR uses object parenting to grab objects, causing there to not be any friction on the object. So we couldn't use it to directly demonstrate a task.

We tried to to around the problem by separating the hand into a rigidbody and a point. What the SteamVR hand grabs is the point, then the hand moves using its rigidbody to move the its position. By moving the hand by proxy this way, we are able to apply unity physics to the hand in order to pick up an object.

There are libraries that provides improved physics for VR controllers. The library we looked at is NewtonVR(the green cubes in the gif). The immediate problem is that they parent the objects they grab. There could be ways to go around this problem by trying something like the move by proxy method we used for the unity friction problem.

A very interesting aspect of NewtonVR is that it stops objects from moving inside each other. Unlike Unity's physics. Which could prove very useful to achieve the realistic behavior we are aiming for.

1/3/2019
UI system and the decision to prototype

UI is hard to design and you need to think about certain concepts, like: reduced friction, understandability/self explanatory and pleasant to view. After some design and discussion on UI we have landed on making prototype for GUI to complete the game loop. The prototype will also be used to iterate on our design decisions and after the game loop is completed we will move on to create our GUI module/system.

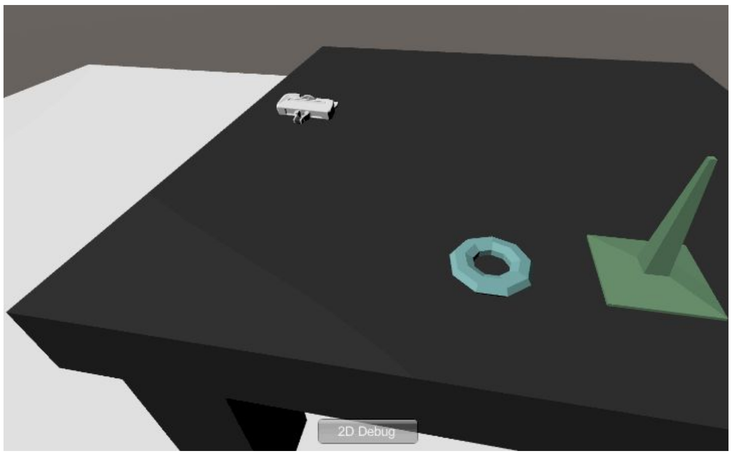
11/3/2019
SACollider Builder

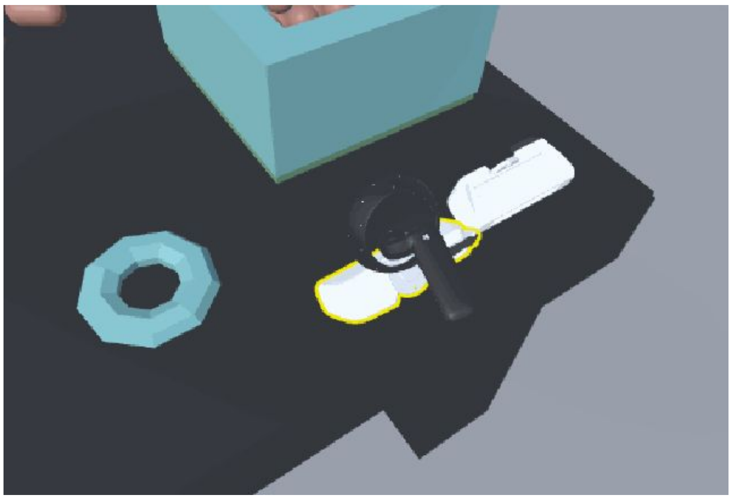
The colliders for mesh objects need to be easy to pick up, and act as similar as possible to how they would behave in real life.

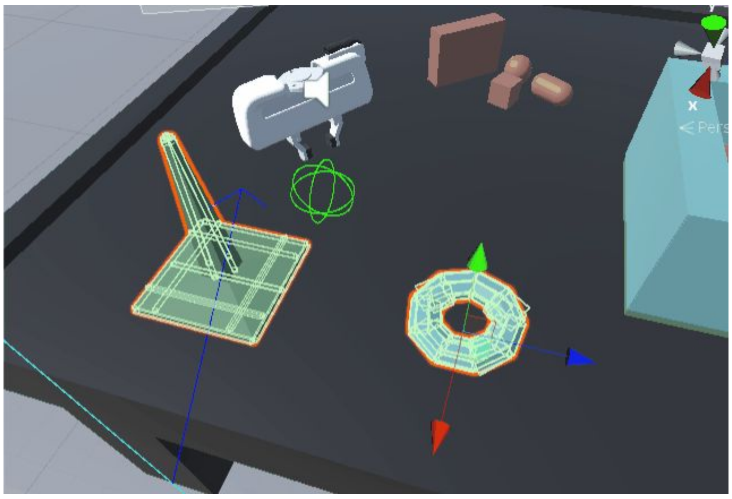
The main challenge is that Unity's mesh colliders are convex, and often act strangely when used with rigidbodies. Additionally, since they are convex that can not have holes. This is problematic in out torus on pole example.

We checked out a unity package called SAColliderBuilder that generates a set of primitive colliders the use of a convex mesh collider. The process of converting an object using a mesh collider, into an object using a SACollider is relatively painless depending on the complexity of the mesh. Though, some time is needed to tweak the SACollider.

A limitation of the SAColliders is that it generates primitives that all have origin somewhere on the surface of the mesh. Meaning that the SACollider will always have a larger boundery than the corresponding mesh collider. It is less accurate by nature. However for our use, the accuracy is more than enough.







11/03/19

Serialization of conditions and the mechanics of C# serialization

Saving data is a problem that has a lot of possible solutions. We decided to go for serialization as it provides speed to saving and loading scenes. The problems with serialization is that unity has its own serialization system that does not work with .net serialization. Most of unity's basic types like Vector3 and Quaternion does not support .net serializations.

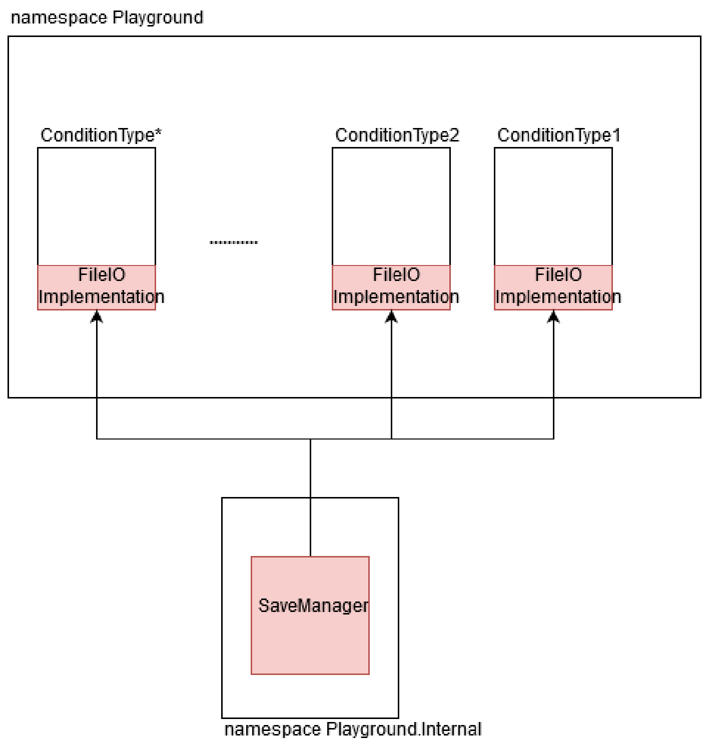
There are some alternatives to work around this. Number one is ISerializationSurrogate which allows you to create a dummy class that functions as a container to all essential data for class state

Example implementation:

```
// property extraction source: https://stackoverflow.com/a/4144817
namespace Playground
{
        [System.Serializable]
        public sealed class PositionCondition : Condition<Transform>
        {
                [SerializeField] private Vector3 _min;
                public Vector3 Position
                {
                        get { return _min; }
                        set { _min = value; }
                }

                [SerializeField] private Vector3 _size;
                public Vector3 Size
                {
                        get { return _size; }
                        set { _size = value.Abs(); }
                }
                public Vector3 Min
                {
                        get => _min;
                        set => _min = value;
                }
                public Vector3 Max
                {
                        get => _min + _size;
                }

                public override bool Evaluate()
                {
                        var evaluation = _volume.InsideVolume(Context.transform.position);
```

```
            return evaluation;
        }
    }

    /// <summary>
    /// Used to serialize and deserialize PositionCondition
    /// </summary>
    public sealed class PosCondSerializationSurrogate : ISerializationSurrogate
    {
        public void GetObjectData(object obj, SerializationInfo info, StreamingContext
    context)
        {
            var positionCondition = (PositionCondition)obj;

            info.AddValue("TargetPosition.x", positionCondition.TargetPosition.x);
            info.AddValue("TargetPosition.y", positionCondition.TargetPosition.y);
            info.AddValue("TargetPosition.z", positionCondition.TargetPosition.z);

            info.AddValue("AllowedDeviation.x",
        positionCondition.AllowedDeviation.x);
            info.AddValue("AllowedDeviation.y",
        positionCondition.AllowedDeviation.y);
            info.AddValue("AllowedDeviation.z",
        positionCondition.AllowedDeviation.z);
        }

        public object SetObjectData(object obj, SerializationInfo info, StreamingContext
    context, ISurrogateSelector selector)
        {
            var positionCondition = (PositionCondition)obj;

            Vector3 targetPosition;
            targetPosition.x = (float)info.GetDouble("TargetPosition.x");
            targetPosition.y = (float)info.GetDouble("TargetPosition.y");
            targetPosition.z = (float)info.GetDouble("TargetPosition.z");
            positionCondition.TargetPosition = targetPosition;

            Vector3 allowedDeviation;
            allowedDeviation.x = (float)info.GetDouble("AllowedDeviation.x");
            allowedDeviation.y = (float)info.GetDouble("AllowedDeviation.y");
            allowedDeviation.z = (float)info.GetDouble("AllowedDeviation.z");
            positionCondition.AllowedDeviation = allowedDeviation;
```

```
                        return positionCondition;
                }
        }
}
```



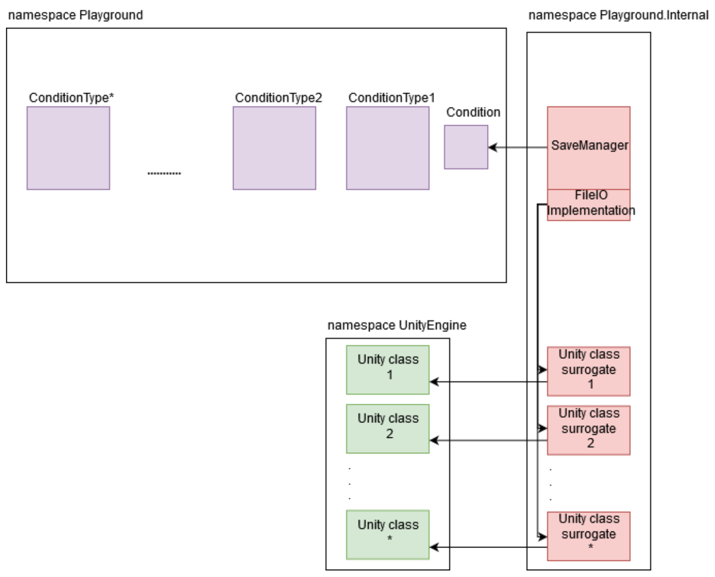
By doing this, all you need then is a black box manager class to do the serialization

This is actually a really simple solution and would work great as it shouldn't be much work and pretty simple for a contributor/dev to implement (in theory). There are two main problems with this solution:

Problem 1: You need to implement the same solutions multiple times as you create a new condition

Problem 2: Can become harder if you introduce more complex types to your condition. An example of this would be an interface that would implement a volume. Then you don't really know the basic types contained in the interface. A cube volume would maybe have center and the biggest x, y, z and the smallest x,y,z. A sphere would have center and radius. This means you need to implement even more for this condition. This would add friction on development in the future.

So the alternative to this becomes making the [System.Serializable] attribute work on any condition you would make, and add the ability to fix it if it doesn't. This means implementing serialization for unity types and any other non-serializable that you would want to use. This way we can blackbox file I/O so that the developers and users don't really need to touch it. Our goal is also then to be open about how you can make other types we have missed serializable, so that in the event where your custom condition fails to serialize you can view the error message and from there easily expand the serialization.



15/03/19
Generating reverse curriculum points. (RCP)
Training the AI with sparse rewards requires reverse curriculum to work effectively on task that are not very tiny. Generating the RCPs that are a certain percent complete is a difficult problem.

The core problem is the measurement of how close a condition is to being complete. Since whether a condition is complete or not, is too vague to use as a measurement, each condition will have to implement its own way of figuring out how close it is to being completed. It's a challenge to define, because each condition would be weighted more than others(like being in the right position is more important than the rotation). Some conditions are binary, like the touching condition, on a scale of 1-100 how touching are you?

The problems complicate when taking into consideration the logical structure conditions can have. For example: there are three conditions in a scene, A, B and C. A and B are in an AND block. Getting C to 100% will complete the scene, since a root goal condition is completed. So the highest root group of the conditions is the one that count for the complete state of the scene.
AND[A 100%, B 100%], C 0% =   scene 100%. A and B's and block is 100%.
AND[A 30%,   B 30%], C 80% = scene 80%. We take C's 80 since it is highest.
AND[A 80%,   B 20%], C 20% = scene ??%

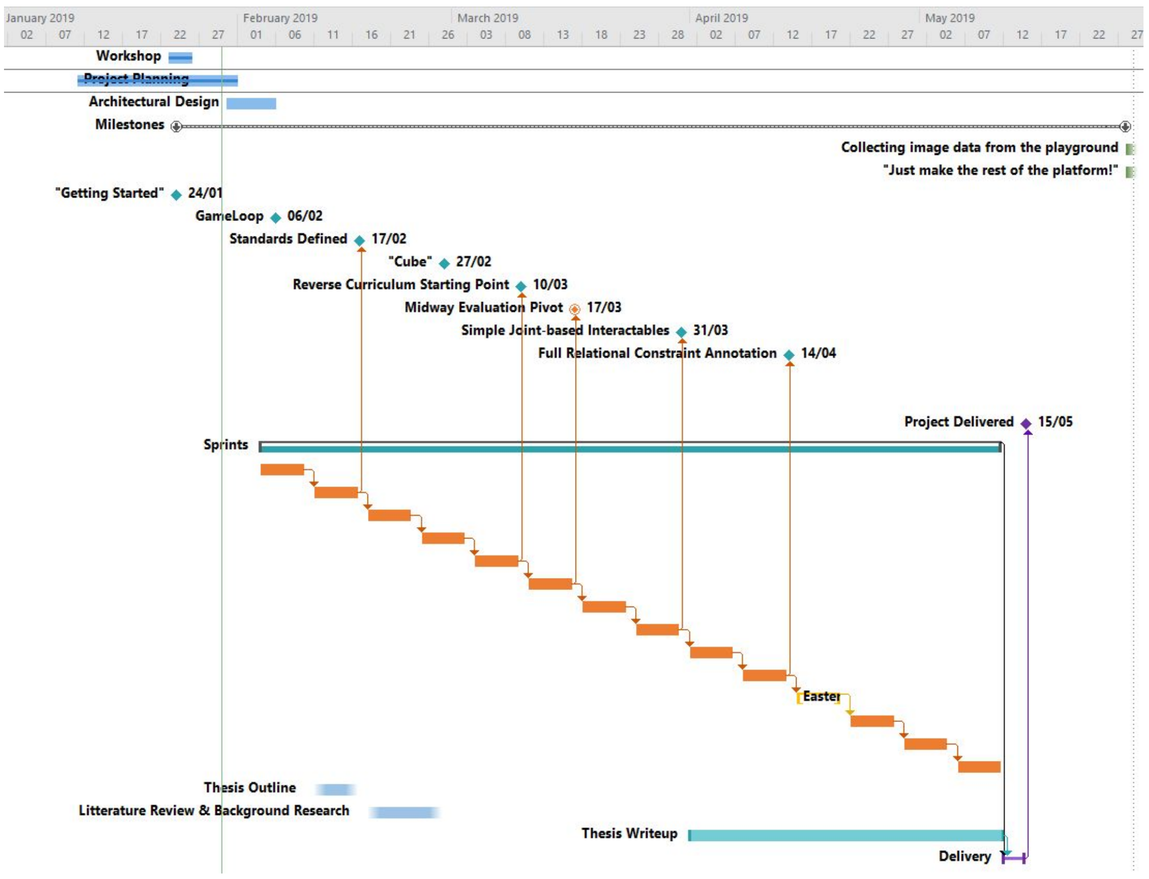
18/03/19
Switching to monobehaviour
Initially the conditions was implemented to be regular C# classes because they are lightweight and fast. This turned out to be a bad decision for several reasons.
Firstly, it meant we would have to write our own way of displaying conditions for debugging purposes, we need to be able to verify that conditions work as they should.
Secondly, serialization of the conditions and their relative game objects turned out to be hard to do using .net serialization as the conditions need to store references to other scene objects. It could always be worked around by implementing our own object ID system, that way we could reference the ID is the saved file. Though that would be a fair bit of work, and would still leave us we the other issue of displaying the conditions. Our solutions is to go back to using monobehaviour, and refactor the existing systems that deals with the c# class conditions.

19/03/19
Major changes and git branches
How do we deal with this? TODO: - discuss
From the start of the project and up until this point, we have been certain that our application has been stable. Now the conditions are changing from being C# object to being monobehaviours. With this large refactoring for the system we are not as sure whether the application is completely stable. To ensure that the head of the master branch is stable we have introduced a new branch dev, that we will push our changes to instead. The dev branch will be merged with master periodically when we have verified that the current dev head is stable.

20/03/19

Midway evaluation

Our original plan/timeline



What has gone well?

Our condition system has had a lot of focus. This had led to parts of the entries in the next header, but it has also lead to a solid backbone for the project. The resulting system is very simple in its functionality, although the grouping and relations between conditions allow for verbose definition of agent constraints and goals.

Group has a common goal. As a result of a lot of group meetings and discussion we all have common understanding of what we are trying to achieve, our structural design and how we want to reach our goal.

Improved work distribution compared to previous group work. We see clear improvements in splitting tasks compared to previous projects, although we don't necessarily feel like we have mastered this field. We feel that the streamlining of our issues and issue board is something to praise ourselves for.

What have gone wrong?

Friction in setting up CI
At the start of the project when the development environment was being set up, we wanted to have gitlabs continuous integration build an image of the application to ensure that is builds, run tests, and have it do linting on the new code. We were unable to set it up due to our gitlab server not being enabled for it, and the technician responsible for the system was unable to fix the problem.

We moved on to using git hooks instead. Writing a script that would use Unity's command line functionality to try to build the application locally before it was pushed instead. We ran into new problems where the Unity wouldn't start if an instance was already running locally, which there always would be as a developer would push changes while still working on the project. This problem made this solution obsolete as it wouldn't save us as much time.

Spending so much time on trying to have the project build when pushing changes we gave up on it, we settled on using ReSharper CLI to do linting on the all the code locally before it was pushed. We set it up to follow our coding standard and inforce referring to issue numbers in commit messages. Having ReSharper setup has led to a cleaner repository and has led up to being more mindful of writing good commit messages.

Overscope
We would have liked to be feature complete by now and begun code revision and redesign. This is sadly not the case and we are still in a crunch to reach an acceptable gameloop. In retrospect we should have been more reflected about the workload of implementing each system component. Although failure to estimate did not come as a surprise.

"Following Milestones", or rather: "not following milestones".
Our milestones were initially stated as they are as an indicator of where we would like to be and what we would've liked to have done in a seemingly sensible timeline; however, throughout the project development it slowly became evident that our milestones were both unrealistic and unhelpful, at least in some ways.
We started of stating that a "gameloop" was above all else in importance as it would make it easier to work with our project as an application always capable of running from start to finish. When we started working however, it was obvious that we would need to do a lot more design planning than we had initially set aside time for, and so we pushed back the milestone for later.

During this process of designing and planning, we started thinking about user stories, and allocated some space for some of these in our “Scrum”-board. During the next few weeks we were more focused on getting into the content we had just planned and designed, and so we unintentionally left the milestones untouched and forgot about them. When we later came back to realize we no longer were following the plan in terms of the milestones, and to some degree we weren’t expecting to either, but we had completely ignored using them at all.
In other words: the milestones were there in the beginning to indicate for ourselves where we wanted to go and how we, in big steps planned to get there.
However, seeing as we had mapped out a few user stories and set them up as issues in our issuetracker board, we had, without really planning for it, created points of importance in our project that could be ticked off as being completed. Internally we’ve found these to be more powerful as milestones as they directly map to what the users needs are for our application, and so we’ve managed to stay on the right course even without the milestones.

Summarizing wrongdoings
We generally were too slow in finding alternatives whenever encountering slow-downs or full on stops due to technological issues or other inter development related issues

What could we have done differently?

We spent more time discussing and designing the interfaces and structure of the conditions and menu system than we thought we would. The gantt diagram from the start of the project says we would have a game loop working after one or two weeks of development. Looking back with the knowledge of the project we have now, that did not make sense. If we started the project again, we would have estimated milestones a lot more accurately.

The conditions are the main components in the playground. They were initially normal C# classes. Having them as regular classes instead of monobehaviours. Because we made this decision, time was spent implementing functionality that already existed in monobehaviours. Although regular classes are faster, the flexibility and convenience of monobehaviour classes outweigh the extra speed, and we ended up changing the components to be MonoBehaviour classes instead. This was a tedious process and caused some issues in the project, and also some time spent on saving and serializing these as C# classes was work that just ended up being discarded. Though there has been time consumed on something that got discarded it has still been a learning process, and something we have learnt well from even though it caused some significant changes in the project.

Defined branch usage better: Currently our project is kind of messy on this part, as there’s no rule for when and why we should use different branches. Having some more clearly defined rules for this could be helpful to avoid conflicts, keeping the repository cleaner and generally give a better overview of what is being worked on.

How has the workflow been, and has the choice of development model (scrum) made sense for the project so far?

Using an agile development model such as scrum has made sense for the most part of the project, though we have been a bit flexible with our backlog and sprint backlogs during development. This has worked out pretty well as the project and design has come along the way during the project and not been strictly set from the beginning. Although, this has lead to some inconvenience the workflow. This has to some degree been a new experience for us, especially because we have had to inform and get on the same page as the 3rd parties at SINTEF. Therefore at certain parts of the project it has been some weeks where development went slower than intended, as a consequence of things not being mapped out entirely from the beginning.

However this is to be expected of a research project of this nature, as it hasn't been done before, at certain points in the project there has been needs for planning and designing modules of the application, and this has slowed down the development to some degree. This is due to the art of the project, as it not having hard requirements to exactly how it is supposed to be implemented and as it is supposed to be a publicly available product, there has been a larger need for us to design things properly than in previous projects.

For these reasons there has always been needs for planning and designing things over, and this has lead to changes internally on how the application layout will look like. This has also prompted changes in the backlog considering what tasks that should be focused on at different times of the project. All of this has been pretty natural while working with an agile development model like scrum, and changes have always been more expected and easier to deal with. Though we probably have had sprints where we have felt it has been hard to implement what we supposedly were implementing at that certain time, this has been due to the need of internal discussions of how things should be designed to get everyone on the same page. In that sense one could say that going for scrum has not been the best decision, but we have throughout the project so far been very flexible with the sprints. That have worked out well for us based on the internal changes throughout the project, and made design discussions and changes easier to handle.

20/03/19
Value Bindings in Widgets
A Central issue we started thinking about before implementing 2D UI-components (2D Widgets) were the relation of data between widgets.
Let's say we have two or more widgets acting on the same piece of data, how do we deal with synchronization between the widgets to make sure the UI is consistent with the actual underlying data the UI is controlling? - Our solution: Value Bindings.

Value Bindings are a concept utilizing C#'s standard where classes are passed by reference: Seeing as Widgets can control the values of objects of both structures and classes, the ideal thing would be to let all widgets that interface the same underlying data actually modify on the same object. In C++ you could do this through a shared pointer to the object in order to share the data between the widget instances.
Enter class references: we can achieve the same thing using C#'s copy-by-reference policy on classes. Where data controlling widgets before looked like this:

```
public abstract class Widget<T> : Widget { public T value; }
public abstract class Widget<T> : Widget { public ValueBinding<T> valueBinding; }
```

Where the ValueBinding is a simple generic class wrapping the underlying value.

```
public class ValueBinding<T>
{
    public T Value { get; set; }
    public static implicit operator T( ValueBinding<T> binding ) => binding.Value;
}
```

This gives the code using widgets the power to build bindings for said widgets in such a way that they all talk about the same underlying data, meaning that other widgets that manipulate the same data don't need to sync with the one being manipulated, but instead will always have the correct internal representation of the data being manipulated.
The external representation, i.e. the visuals of the UI will still need to be updated however, but this is no problem, as each of the widgets pushes calls to their parent whenever they manipulate the data. This in turn means that the widget can manipulate the internal data, push a message to its parent (which by the way is a recursive call, as any parent will keep calling their own parent till it's null), then the parent can force a "redraw" on all its widget children to update their external representation.

21/03/2019
The possibility of multithreading evaluation of conditions in the future
We realize that evaluation of the scene condition state can become heavy in more complex scenes, and therefore we have looked into multithreading as a possible mitigation. We see value in implementing multithreading for this, but we also understand that introducing more complexity at this stage would not be feasible at this point. Therefore we looked into possible solutions for future work.

.NET 4 multithreading introduces the concept of "task" through their library called TPL. The goal of TPL "is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications"
This is a nice alternative when implementing multithreading as it could speed up the process and result in safer code

Unity job system would be great for our project. The problem with this at the time of writing is that the job system is still in its preview stage and are therefore it might be risky to use. We would also have to introduce other preview packages like the BURST compiler. The job system would however give us a even safer development environment.

26/03/2019

Simplifying UI

While designing the in-world 2D UI for the application, we tried minimizing amount of clicks the user has to make. This is even more important in VR as laser pointer clicking is more annoying than with a mouse. Originally we had a setup-queue-menu that contained all the conditions that needed some setup process to work. All the menu did in reality is moving the setup process behind unnecessary clicks, so we decided to not have the setup-queue-menu.

F Referenced Code, Full Code

F.1 PositionCondition

```
using UnityEngine;
namespace Playground
{
    [System.Serializable]
    public sealed class PositionCondition : Condition<Transform>
    {
        public enum VolumeMode
        {
            Cuboid,
            Sphere,
            Cylinder
        }
        public VolumeMode volumeMode;
        [SerializeField]
        public VolumeBase _volume;

        public override bool Evaluate()
        {
            var evaluation = _volume.InsideVolume(Context.transform.position);
            return evaluation;
        }

        public override ConditionContainer GetDataContainer( ConditionValue conditionValue, bool
             inNestedGroupParam )
        {
            var container = base.GetDataContainer(conditionValue, inNestedGroupParam);

            var pcc = container.CopyBasicValues<PositionConditionContainer>();
            pcc.volumeBase = _volume;
            pcc.volumeMode = volumeMode;

            return pcc;
        }
    }
}
```

F.2 TouchCondition

```
using UnityEngine;
using droid.Runtime.Utilities.Sensors;
using droid.Runtime.Utilities.Misc;


namespace Playground
{
    [System.Serializable]
    public sealed class TouchCondition : Condition<Collider>
    {
        #region context_components
        private Collider _contextCollider; //NOTE: what about multi-collider objects - should be fixed by
             using sensors
        #endregion

        #region parameters
        #endregion

        private bool _isColliding;

        protected override void PreConfigure()
        {
            _contextCollider = Context.GetComponent<Collider>();
            NeodroidUtilities.RegisterCollisionTriggerCallbacksOnChildren<ChildCollider3DSensor, Collider,
                 Collision>(_contextCollider, Context.transform, on_collision_enter_child: SetColliding,
                 on_collision_exit_child: UnSetColliding);
        }

        void SetColliding(GameObject self, Collision collision)
        {
            if (collision.gameObject == Relative.gameObject)
                _isColliding = true;
        }
        void UnSetColliding(GameObject self, Collision collision)
        {
            if (collision.gameObject == Relative.gameObject)
                _isColliding = false;
        }

        public override bool Evaluate()
        {
            return _isColliding;
        }


        public override ConditionContainer GetDataContainer(ConditionValue conditionValue, bool
             inNestedGroupParam)
        {
            var container = base.GetDataContainer(conditionValue, inNestedGroupParam);
            var tcc = container.CopyBasicValues<TouchConditionContainer>();

            return tcc;
        }
    }
}
```

F.3 RotationConditionTest

```
using NUnit.Framework;
using System.Collections;
using UnityEngine;

// NOTE: this was written to test alternative testing methods and syntactic approaches
namespace Playground.Tests
{
    [TestFixture]
    public class RotationConditionTest
    {
        static TestData testData;
        private RotationCondition ParamToRotationCondition(float ad1, float ad2, float ad3, float tr1,
             float tr2, float tr3)
        {


            var condition = testData.contextObject.gameObject.AddComponent<RotationCondition>();
            condition.AllowedEulerDeviation = new Vector3(ad1, ad2, ad3);
            condition.TargetEulerRoation    = new Vector3(tr1, tr2, tr3);

            return condition;
        }

        // NOTE: params would break nuint it seems
        [Test]
        [TestCaseSource(typeof(RotationConditionData), "TestCases")]
        public bool Rotation_Tests(float ad1, float ad2, float ad3, float tr1, float tr2, float tr3)
        {

            testData = TestUtility.SetupDefaultScene();

            var condition = ParamToRotationCondition(ad1, ad2, ad3, tr1, tr2, tr3);

            // We need this to make linter happy even though we could have Assume for this...
            if (condition == null)
            {
                Debug.LogWarning("condition was null");
                return false;
            }
            if (testData.contextObject == null)
            {
                Debug.LogWarning("contextObject was null");
                return false;
            }
            if (testData.relativeObject == null)
            {
                Debug.LogWarning("relativeObject was null");
                return false;
            }

            testData.contextObject.transform.rotation = Quaternion.identity;
            //InitCondition(condition);

            condition.Initialize(testData.contextObject);
            condition.SetRelative(testData.relativeObject.GetComponent<Transform>());

            testData.contextObject.AddGoalCondition(condition);
            return TestUtility.EvaluateConditions(testData.contextObject);


        }
    }

    public class RotationConditionData
    {
        public static IEnumerable TestCases
        {
            get
            {
                yield return new TestCaseData(10f, 10f, 10f, 0f, 0f, 0f).Returns(true);
                yield return new TestCaseData(0f, 0f, 0f, 0f, 0f, 0f).Returns(true);
            }
        }
    }
}
```

F.4 TouchConditionTest

```
using System.Collections;
using UnityEngine;
using UnityEngine.TestTools;
using NUnit.Framework;

namespace Playground.Tests
{
    /// <summary>
    /// Class for testing touch condition
    /// </summary>
    public class TouchConditionTest : ConditionsTest
    {
        [SetUp]
        public override void Setup()
        {
            testCount = 2;

            base.Setup();

            TouchCondition condition = testData.contextObject.gameObject.AddComponent<TouchCondition>();

            condition.Initialize(testData.contextObject);
            condition.SetRelative(testData.relativeObject.GetComponent<Collider>());

            testData.contextObject.GoalConditions.AddCondition(condition);
        }

        [UnityTest]
        public IEnumerator NoContact_WithRelative_ResultsToFalse()
        {
            testData.contextObject.transform.position = new Vector3(0, 200, 0);
            yield return new WaitForFixedUpdate();//wait for physics

            Assert.IsFalse(TestUtility.EvaluateConditions(testData.contextObject));
        }

        [UnityTest]
        public IEnumerator Touching_Relative_ResultsToTrue()
        {
            testData.contextObject.transform.position = new Vector3(0, 0.9f, 0);
            for (int i = 0; i < 30; i++)
                yield return new WaitForFixedUpdate();

            Assert.IsTrue(TestUtility.EvaluateConditions(testData.contextObject));
        }

        // teardown in base
    }
}
```

F.5 SceneStateEvaluation

```
using droid.Runtime.Prototyping.Evaluation;
using System.Collections.Generic;

namespace Playground.Internal
{
    public class SceneStateEvaluation : ObjectiveFunction
    {
        protected const int DEFAULT_NR_OF_SAMPLES = 1000;

        public List<SceneObject> sceneObjects = new List<SceneObject>();

        protected override void PostSetup()
        {
            sceneObjects.Clear();
            foreach (SceneObject obj in FindObjectsOfType<SceneObject>())
            {
                sceneObjects.Add(obj);
            }
        }

        public override float InternalEvaluate()
        {
            foreach (var sceneObject in sceneObjects)
            {
                if (sceneObject.TerminatingConditions.ConditionsInGroup > 0 &&
                     sceneObject.TerminatingConditions.Evaluate())
                {
                    ParentEnvironment?.Terminate($"Entered terminating state on object: { sceneObject }");
                    return -1; // Terminating condition were fulfilled, returning negative signal
                }
            }

            // Checking goal conditions after termination conditions
            foreach (var sceneObject in sceneObjects)
            {
                if (sceneObject.GoalConditions.ConditionsInGroup > 0 &&
                     sceneObject.GoalConditions.Evaluate())
                {
                    ParentEnvironment?.Terminate($"Entered goal state on object: { sceneObject }");
                    return 1; // Goal condition were fulfilled, returning positive signal
                }
            }
            return 0; // No conditions were fulfilled, returning neutral signal
        }

        public override void InternalReset()
        {
            // Could possibly do some stuff for Reverse Curriculum Generation here
        }

        public void GenerateState( float percent, uint samples, params SceneObject[] objects )
        {
            // construct a randomized list of the given objects; dubbed "the frontier stack"
            // get number of sceneobjects with conditions
            // get total number of conditions across the entire scene
            // get nr of sceneobject's conditions to satisfy. -> and number of sceneobjects whole state to
               satisfy.
            // loop through the sceneobjects and conditions in question, try to generate samples that are
               valid states.
            // NOTE: conditions can implement their own "Shake" function in order to get the correct
               percentage for a given sceneobject
            // TODO: MORE

            objects.Shuffle(); //use the same shuffled configuration for all the preceding samples.

            //int nrOfSceneObjects = objects.Length;
            //int nrOfConditions   = 0;
            ////get the shallow count of conditions
            //foreach (var obj in objects)
            //{
            //    nrOfConditions += (obj.GoalConditions.ConditionsInGroup > 0) ? 1:0;
            //    nrOfConditions += (obj.TerminatingConditions.ConditionsInGroup > 0) ? 1 : 0;
            //}
        }

        public void GenerateState( float percent, uint samples = DEFAULT_NR_OF_SAMPLES)
        {
            GenerateState(percent, samples, sceneObjects.ToArray());
        }
    }
}
```

F.6 Widget Base Classes

F.6.1 Widget

```
using UnityEngine;
using Valve.VR.InteractionSystem;

namespace Playground.UI
{
    [HelpURL("https://justworksltd.gitlab.io/playground-docs/class_playground_1_1_u_i_1_1_widget.html")]
    public abstract class Widget : MonoBehaviour
    {
#if UNITY_EDITOR
        protected virtual void Reset() { }
#endif
        [SerializeField] protected internal Widget _owner;
        public Widget Owner => _owner;
        public Condition Condition { get; protected set; }

        public void SetOwner( Widget owner )
        {
            if (owner == this)
            {
                Debug.LogError($"{this} tried to set owner to self!");
                return;
            }
            _owner = owner;
            Condition = owner.Condition;
            OnConditionContextSet();
            PropagatePushChanges(); //ensure the Widget in its finalized initial state is pushed up  TODO:
                This causes pro
        }

        /// <summary>
        /// Override to do component specific setup based on condition context.
        /// This is usually where you would ensure persistency by reconstructing
        /// the widget settings based on the value handle of the widget
        /// </summary>
        protected virtual void OnConditionContextSet() { }

        /// <summary>Sets the widget to its initial state.</summary>
        public abstract void SetInitialState();
        protected void PropagateStartUpdate()
        {
            OnStartUpdate();
            _owner?.PropagateStartUpdate();
        }
        protected void PropagateUpdateVisuals()
        {
            OnUpdateVisuals();
            _owner?.PropagateUpdateVisuals();
        }
        protected void PropagateStopUpdate()
        {
            OnStopUpdate();
            _owner?.PropagateStopUpdate();
        }
        protected void PropagatePushChanges()
        {
            OnPushChanges();
            _owner?.PropagatePushChanges();
        }

        /// <summary>
        /// Called upon when grabbed/attached
        /// </summary>
        public virtual void OnStartUpdate() { }

        /// <summary>
        /// Updates done internally
        /// </summary>
        public virtual void OnUpdateVisuals() { }

        /// <summary>
        /// Called upon when released/un-attached
        /// </summary>
        public virtual void OnStopUpdate() { }

        /// <summary>
        /// update external state so that the rest of the UI-system reccognize the changes made.
        /// </summary>
        public virtual void OnPushChanges() { }
    }
}
```

F.6.2 Widget2D

```
using UnityEngine;
using Valve.VR.InteractionSystem;

namespace Playground.UI
{
    public abstract class Widget2D : Widget
    {
        public RectTransform RectTransform => transform as RectTransform;
    }
}
```

F.6.3 Widget2D<T>

```
using UnityEngine;
using Valve.VR.InteractionSystem;

namespace Playground.UI
{
    public abstract class Widget2D<T> : Widget2D, IValuedWidget<T>
    {
        private ValueHandle<T> valueHandle = new ValueHandle<T>();

        public void FetchHandle( out ValueHandle<T> handle ) => handle = valueHandle;
        public void SetValueHandle( ValueHandle<T> handle ) => valueHandle = handle;
        public T Value
        {
            get => valueHandle.Value;
            set => valueHandle.Value = value;
        }
    }
}
```

F.6.4 WidgetVR

```
using UnityEngine;
using Valve.VR.InteractionSystem;

namespace Playground.UI
{
    [RequireComponent(typeof(Interactable))]
    public abstract class WidgetVR : Widget
    {
#if UNITY_EDITOR
        protected override void Reset() => _interactable = GetComponent<Interactable>();
#endif
        [SerializeField] protected Interactable _interactable;
        protected bool attached;
        public bool Attached => attached;

        [EnumFlags]
        [Tooltip("The flags used to attach this object to the hand.")]
        public Hand.AttachmentFlags attachmentFlags = Hand.AttachmentFlags.ParentToHand |
            Hand.AttachmentFlags.DetachFromOtherHand;


        //========================================================
        //-------------------------STEAM VR-------------------------
        //========================================================

        //-------------------------------------------------
        // Attach To Hand
        //-------------------------------------------------
        /// <summary>
        /// Default implementation of hand attachment for widgets
        /// </summary>
        /// <param name="hand">the hand to attach to</param>
        protected virtual void DoAttachToHand( Hand hand )
        {
            attached = true;
            hand.HoverLock(null);
        }
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens when the hand tries to make this attach</para>
        /// </summary>
        /// <param name="hand">the attaching hand</param>
        protected void OnAttachedToHand( Hand hand )
        {
            DoAttachToHand(hand);
            PropagateStartUpdate();
        }

        //-------------------------------------------------
        // Hand Hover Begin
        //-------------------------------------------------
        /// <summary>
        /// Default implementation of hand beggining hovering for widgets
        /// </summary>
        /// <param name="hand">the hand hovering over this</param>
        protected virtual void DoHandHoverBegin( Hand hand )
        {
            if (!attached)
            {
                GrabTypes bestGrabType = hand.GetBestGrabbingType();
                if (bestGrabType != GrabTypes.None)
                {
                    hand.AttachObject(gameObject, bestGrabType, attachmentFlags);
                }
            }
        }
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens when a hand starts hovering over this</para>
        /// </summary>
        /// <param name="hand">the hovering hand</param>
        protected void OnHandHoverBegin( Hand hand ) => DoHandHoverBegin(hand);

        //-------------------------------------------------
        // Hand Hover End
        //-------------------------------------------------
        /// <summary>
        /// Default implementation of hand stopping hovering for widgets
        /// </summary>
        /// <param name="hand">the hand hovering over this</param>
        protected virtual void DoHandHoverEnd( Hand hand ) => hand.HideGrabHint();
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens when a hand stops hovering over this</para>
        /// </summary>
        /// <param name="hand">the hovering hand</param>
        protected void OnHandHoverEnd( Hand hand ) => DoHandHoverEnd(hand);

        //-------------------------------------------------
        // Hand Hover Update
        //-------------------------------------------------
        /// <summary>
        /// Default implementation of updates while hovering over widgets
        /// </summary>
        /// <param name="hand">the hand hovering over this</param>
        protected virtual void DoHandHoverUpdate( Hand hand )
        {
            GrabTypes startingGrabType = hand.GetGrabStarting();
            if (startingGrabType != GrabTypes.None)
            {
                hand.AttachObject(gameObject, startingGrabType, attachmentFlags);
                hand.HideGrabHint();
            }
        }
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens during updates while hovering over this</para>
        /// </summary>
        /// <param name="hand">the hovering hand</param>
        protected void HandHoverUpdate( Hand hand ) => DoHandHoverUpdate(hand);


        //-------------------------------------------------
        // Detach From Hand
        //-------------------------------------------------
        /// <summary>
        /// Default implementation of detaching widgets from hand
        /// </summary>
        /// <param name="hand">the detaching hand</param>
        protected virtual void DoDetachFromHand( Hand hand )
        {
            attached = false;
            PropagateStopUpdate();
            PropagatePushChanges();
            hand.HoverUnlock(null);
        }
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens when this is detached</para>
        /// </summary>
        /// <param name="hand">the detaching hand</param>
        protected void OnDetachedFromHand( Hand hand ) => DoDetachFromHand(hand);


        //-------------------------------------------------
        // Attached Update
        //-------------------------------------------------
        /// <summary>
        /// Default implementation of attached updates for widgets
        /// </summary>
        /// <param name="hand">the attached hand</param>
        protected virtual void DoAttachedUpdate( Hand hand )
        {
            PropagateUpdateVisuals();
            if (hand.IsGrabEnding(gameObject))
            {
                hand.DetachObject(gameObject);
            }
        }
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens in the update loop when this is attached</para>
        /// </summary>
        /// <param name="hand">the attached hand</param>
        protected void HandAttachedUpdate( Hand hand ) => DoAttachedUpdate(hand);

        //-------------------------------------------------
        // Hand Focus
        //-------------------------------------------------
        protected void OnHandFocusAcquired( Hand hand ) => gameObject.SetActive(true);
        protected void OnHandFocusLost( Hand hand ) => gameObject.SetActive(false);
    }
}
```

F.6.5 WidgetVR<T>

```
using UnityEngine;
using Valve.VR.InteractionSystem;

namespace Playground.UI
{
    public abstract class WidgetVR<T> : WidgetVR, IValuedWidget<T>
    {
        private ValueHandle<T> valueHandle = new ValueHandle<T>();
        public void FetchHandle( out ValueHandle<T> handle ) => handle = valueHandle;
        public void SetValueHandle( ValueHandle<T> handle ) => valueHandle = handle;
        public T Value
        {
            get => valueHandle.Value;
            set => valueHandle.Value = value;
        }
    }
}
```

F.7 DropdownWidget2D

```
using UnityEngine;
using UnityEngine.UI;

```

```
namespace Playground.UI
{
    [RequireComponent(typeof(Dropdown)), AddComponentMenu("Playground/Widget 2D/Dropdown")]
    public class DropdownWidget2D : Widget2D<int>
    {
#if UNITY_EDITOR
        protected override void Reset()
        {
            base.Reset();
            dropdown = GetComponent<Dropdown>();
        }
#endif
        // UnityEngine.UI.Dropdown - the Dropdown Component by Unity
        public Dropdown dropdown;

        private void Awake()
        {
            //inject calls to our interface
            dropdown.onValueChanged.AddListener(_ => PropogatePushChanges()); //call directly to the base
                in order to propogate the event
        }
        public override void SetInitialState() => dropdown.value = Value;
        protected override void OnConditionContextSet() => dropdown.value = Value;
        public void SetOptions( in System.Enum enumValue ) =>
             SetOptions(System.Enum.GetNames(enumValue.GetType()));

        public void SetOptions(in string[] options)
        {
            dropdown.ClearOptions();
            dropdown.AddOptions(new System.Collections.Generic.List<string>(options));
        }
        public void SetOptions(in System.Collections.Generic.List<string> options)
        {
            dropdown.ClearOptions();
            dropdown.AddOptions(options);
        }

        public ref readonly T GetValue<T>( in T[] options ) => ref options[Value];

        public override void OnPushChanges() => Value = dropdown.value;
    }
}
```

F.8 SubMenu

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

using Valve.VR.InteractionSystem;
namespace Playground.UI {

    [System.Serializable]
    public struct LayoutData
    {
        /// <summary>
        /// A reference to the child beeing layed out
        /// </summary>
        public SubMenu menuRef;
        /// <summary>
        /// The position of this child when placed inside the parent submenu
        /// </summary>
        public Vector3 internalPosition;
        public Vector2 internalSizeDelta;
        //TODO: Add internalRotation?
    }


    [RequireComponent(typeof(Interactable))]
    [System.Serializable]
    public class SubMenu : MonoBehaviour
    {
#pragma warning disable 0649
        [SerializeField] private string _name;
        public string Name
        {
            get => _name;
            set => _name = value;
        }

        [SerializeField] private SubMenu _owner;
        private SubMenu _root;

        [SerializeField] private LayoutData[] _childLayouts;
        [SerializeField] private Widget[] _widgets; //NOTE: may not need to be serialized
        [SerializeField] private Transform _grabbable; //i.e. the transform of the collider
#pragma warning restore 0649

        public SubMenu Root
        {
            get
            {
                if (!_root)
                {
                    for (SubMenu parent = _owner; parent; parent = parent._owner) {
                        _root = parent;
                    }
                }
                Debug.Assert(_root != null, $"{name}: Root was null when trying to access it!");
                return _root;
            }
        }

        [Range(50, 1000), Tooltip("Drag Threshold in pixelspace: how far you need to drag the element in
             order to detach it")]
        [SerializeField]
        private int _detachThreshold = 100;

        [SerializeField] RectTransform container;

        bool attachedToHand;

        [EnumFlags, Tooltip("The flags used to attach this object to the hand.")]
        public Hand.AttachmentFlags attachmentFlags = Hand.AttachmentFlags.ParentToHand |
             Hand.AttachmentFlags.DetachFromOtherHand;

        Vector3 startGrabPosition;

        RectTransform RectTransform => (RectTransform)transform;

        Vector3 OwnerSpace( Vector3 worldPosition ) {
            return _owner?.transform.InverseTransformPoint(worldPosition) ?? worldPosition;
        }

#if UNITY_EDITOR
        [ContextMenu("Store current Child Layout")]
        private void SaveLayout()
        {
            //TODO: Remove LayoutGroup-related things
            void RecurseTransform(ref List<SubMenu> children, Transform innerTransform)
            {
                for (int i = 0; i < innerTransform.childCount; i++)
                {
                    var child = innerTransform.GetChild(i);
                    var childMenu = innerTransform.GetChild(i).GetComponent<SubMenu>();
                    if (childMenu)
                        children.Add(childMenu);
                    else
                    {
                        var layoutGroup = child.GetComponent<LayoutGroup>();
                        if(layoutGroup)
                            RecurseTransform(ref children, layoutGroup.transform);
                    }
                }
            }
            List<SubMenu> childList = new List<SubMenu>();
            RecurseTransform(ref childList, transform);


            _childLayouts = new LayoutData[childList.Count];
            for (int i = 0; i < _childLayouts.Length; i++)
            {
                childList[i]._owner = this;

                _childLayouts[i].menuRef = childList[i];
                _childLayouts[i].internalPosition = _childLayouts[i].menuRef.transform.localPosition;
                _childLayouts[i].internalSizeDelta = _childLayouts[i].menuRef.RectTransform.sizeDelta;
                _childLayouts[i].menuRef.SaveLayout();
            }
            UnityEditor.EditorUtility.SetDirty(this);
        }

        [ContextMenu("Test Revert")]
        public void TestRevert()
        {
            RevertLayoutToDefault();
        }

        private void Reset()
        {
            if (!gameObject.GetComponentInParent<SubMenu>() && _root != null)
            {
                Debug.LogError("invalid menu structure: parent does not contain a submenu");
            }
        }
#endif
        /// <summary>
        /// Handles layout position data
        /// </summary>
        /// <param name="placableWidget">widget to be place</param>
        public void InsertAndLayoutWidget(Widget2D placableWidget)
        {
            placableWidget.transform.SetParent(transform, true);
            placableWidget.transform.localScale = Vector3.one;
            //TODO: MORE
        }

        private void Start()
        {
            _grabbable.Find("MenuLabel").GetComponent<Text>().text = _name;
        }

        protected void RevertLayoutToDefault()
        {
            foreach(var childLayout in _childLayouts)
            {
                childLayout.menuRef.RevertLayoutToDefault();

                if (childLayout.menuRef._owner == this)
                    continue;

                childLayout.menuRef._owner = this;
                var oldContainer = childLayout.menuRef.transform.parent;
                childLayout.menuRef.transform.parent = transform;
                Destroy(oldContainer.gameObject);
                childLayout.menuRef.transform.localPosition = childLayout.internalPosition;
                childLayout.menuRef.RectTransform.sizeDelta = childLayout.internalSizeDelta;
                childLayout.menuRef.transform.localRotation = Quaternion.identity;
            }
        }

        protected void DetachFromOwner()
        {
            //do some stuff, then:
            container = (RectTransform)(new GameObject($"{name}_container", typeof(Canvas),
                 typeof(CanvasScaler)).transform);

            container.transform.localPosition = _owner.transform.localPosition; //TODO: move the container
                 about to avoid floating point errors
            container.transform.localRotation = _owner.transform.localRotation;
            container.transform.localScale = Root.container.localScale;


            /*
             Container holds information about how to recover layout of detached parts
             Dictionary might be used for this

            OnDetachMenu should copy the submenu, reset the old one's position and disable it.
             */

            var ownerCanvasScaler = Root.container.GetComponent<CanvasScaler>();
            var canvasScaler = container.GetComponent<CanvasScaler>();

            Vector3 preUnparentPosition = transform.position;
            _owner = null;
            //container.sizeDelta = RectTransform.sizeDelta; Look at making the sizes nicer
            transform.parent = container;

            canvasScaler.dynamicPixelsPerUnit = ownerCanvasScaler.dynamicPixelsPerUnit;
            canvasScaler.referencePixelsPerUnit = ownerCanvasScaler.referencePixelsPerUnit;


            OnDetachFromOwner();
        }

        public virtual void OnDetachFromOwner()
        {

        }

        protected void SnapBack()
        {
            //snap back to the grabbing position relative to the attachment point of this menu
            transform.localPosition = startGrabPosition - _grabbable.transform.localPosition;
        }

        public virtual void OnSnapBack()
        {

        }

        #region STEAM_VR
        //-------------------------------------------------
        // Attach To Hand
        //-------------------------------------------------
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens when the hand tries to make this attach</para>
        /// </summary>
        /// <param name="hand">the attaching hand</param>
        protected void OnAttachedToHand( Hand hand )
        {
            startGrabPosition = OwnerSpace(_grabbable.position);
            if (_owner)
            {
                Debug.DrawLine(_owner.transform.TransformPoint(startGrabPosition) -
                    _owner.transform.forward, _owner.transform.TransformPoint(startGrabPosition) +
                    _owner.transform.forward, Color.red, 20.0f);
            }
            attachedToHand = true;
            hand.HoverLock(null);
        }

        //-------------------------------------------------
        // Hand Hover Begin
        //-------------------------------------------------
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens when a hand starts hovering over this</para>
        /// </summary>
        /// <param name="hand">the hovering hand</param>
        protected void OnHandHoverBegin( Hand hand )
        {
            if (!attachedToHand)
            {
                GrabTypes bestGrabType = hand.GetBestGrabbingType();
                if (bestGrabType != GrabTypes.None)
                {
                    hand.AttachObject(gameObject, bestGrabType, attachmentFlags);
                }
            }
        }

        //-------------------------------------------------
        // Hand Hover End
        //-------------------------------------------------
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens when a hand stops hovering over this</para>
        /// </summary>
        /// <param name="hand">the hovering hand</param>
        protected void OnHandHoverEnd( Hand hand ) => hand.HideGrabHint();

        //-------------------------------------------------
        // Hand Hover Update
        //-------------------------------------------------
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens during updates while hovering over this</para>
        /// </summary>
        /// <param name="hand">the hovering hand</param>
        protected void HandHoverUpdate( Hand hand )
        {
            GrabTypes startingGrabType = hand.GetGrabStarting();
            if (startingGrabType != GrabTypes.None)
            {
                hand.AttachObject(gameObject, startingGrabType, attachmentFlags);
                hand.HideGrabHint();
            }
        }


        //-------------------------------------------------
        // Detach From Hand
        //-------------------------------------------------
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens when this is detached</para>
        /// </summary>
        /// <param name="hand">the detaching hand</param>
        protected void OnDetachedFromHand( Hand hand )
        {
            if (_owner)
            {
                Debug.Log(Vector3.Distance(OwnerSpace(_grabbable.position), startGrabPosition));

                if (Vector3.Distance(OwnerSpace(_grabbable.position), startGrabPosition) >
                     _detachThreshold)
                    DetachFromOwner();
                else
                    SnapBack();
            }


            attachedToHand = false;
            hand.HoverUnlock(null);
        }

        //-------------------------------------------------
        // Attached Update
        //-------------------------------------------------
        /// <summary>
        /// Valve's interface for interactables
        /// <para>What happens in the update loop when this is attached</para>
        /// </summary>
        /// <param name="hand">the attached hand</param>
        protected void HandAttachedUpdate( Hand hand )
        {
            if (hand.IsGrabEnding(gameObject))
            {
                hand.DetachObject(gameObject);
            }
        }

        //-------------------------------------------------
        // Hand Focus
        //-------------------------------------------------
        protected void OnHandFocusAcquired( Hand hand ) => gameObject.SetActive(true);
        protected void OnHandFocusLost( Hand hand ) => gameObject.SetActive(false);
        #endregion
    }
}
```

F.9 RaycastingInputModule

```
using UnityEngine;
using UnityEngine.EventSystems;
using Valve.VR.InteractionSystem;
using Valve.VR;
using System;


namespace Playground.Internal
{
    //https://bitbucket.org/Unity-Technologies/ui/src/0651862509331da4e85f519de88c99d0529493a5/UnityEngine.UI/EventSystem
    //https://github.com/wacki/Unity-VRInputModule/blob/master/Assets/VRInputModule/Scripts/LaserPointerInputModule.cs

    [RequireComponent(typeof(Camera))]
    public class RaycastingInputModule : BaseInputModule
    {
        public Hand hand;

        private float m_PrevActionTime;
        private Vector2 m_LastMoveVector;
        //private int m_ConsecutiveMoveCount;

        private Vector2 _lastPointerPosition;
        private Vector2 _pointerPosition;


#if UNITY_EDITOR
        public bool debug;
        public bool fallbackMock => !GameManager.Instance.usingVR;
        private Transform _mockHandTransform => GameManager.Instance.player;//=> _hand.transform;
        private Transform handTransform => fallbackMock ? _mockHandTransform : hand?.transform ??
             _mockHandTransform;
# else
        private Transform handTransform => _hand.transform;
#endif

        public SteamVR_Input_Sources InputSource => hand?.handType ?? SteamVR_Input_Sources.Any;
#pragma warning disable 0649
        [SerializeField] private LayerMask interactionLayer;
        [SerializeField] private SteamVR_Action_Boolean _clickAction;
#pragma warning restore 0649
        //[SerializeField] private SteamVR_Action_Vector2 _scrollAction;
#if UNITY_EDITOR
        [Space]
#endif
        [SerializeField] private float _laserReach = 100f;
        private Camera _eventProcessingCamera;

        [NonSerialized] private GameObject currentEnter;
        [NonSerialized] private GameObject currentPressed;
        [NonSerialized] private GameObject currentDragging;

        RaycastHit[] sceneObjectHits = new RaycastHit[1];


        protected override void Awake()
        {
            base.Awake();
            _eventProcessingCamera = GetComponent<Camera>();
            // We don't really care about using the camera for rendering,
            // so don't do graphics related things
            _eventProcessingCamera.orthographic = true;
            _eventProcessingCamera.orthographicSize = 0.01f;
            _eventProcessingCamera.clearFlags = CameraClearFlags.Nothing;
            _eventProcessingCamera.enabled = false;
            _eventProcessingCamera.fieldOfView = 5;
            _eventProcessingCamera.nearClipPlane = 0.01f;
            _eventProcessingCamera.farClipPlane = _laserReach;
            //_eventProccessingCamera.cullingMask = CULL NON UI?

            inputOverride = GetComponent<SteamVRInput>();
        }

        protected override void Start()
        {
            base.Start();
            GameManager.Instance.Canvas.worldCamera = _eventProcessingCamera;
        }

        protected bool SendUpdateEventToSelectedObject()
        {
            if (eventSystem.currentSelectedGameObject == null)
                return false;

            var data = GetBaseEventData();
            ExecuteEvents.Execute(eventSystem.currentSelectedGameObject, data,
                 ExecuteEvents.updateSelectedHandler);
            return data.used;
        }

        public override void Process()
        {
            bool usedEvent = SendUpdateEventToSelectedObject();
            ProcessVRPointerEvents();
        }

        protected void ProcessVRPointerEvents()
        {
            UpdateEventProcessingCamera();
            var eventData = GetVRPointerEventData();
            ProcessMove(eventData);
            ProcessClick(eventData);
        }

        private void UpdateEventProcessingCamera()
        {
            _eventProcessingCamera.transform.SetPositionAndRotation(
                handTransform.position,
                handTransform.rotation
            );
        }

        private VRPointerEventData GetVRPointerEventData()
        {
            VRPointerEventData eventData = new VRPointerEventData(eventSystem){ inputSource = InputSource
                 };
            //TODO: cache
            eventData.Reset();
            eventData.position = _eventProcessingCamera.pixelRect.center;

            eventData.pointerPress = currentPressed;
            eventData.pointerDrag = currentDragging;
            eventData.pointerEnter = currentEnter;

            eventSystem.RaycastAll(eventData, m_RaycastResultCache); // fill the eventdata by casting a ray
            eventData.pointerCurrentRaycast = FindFirstRaycast(m_RaycastResultCache);
            //NOTE: it might be worth to do this every frame BEFORE the raycast instead,
            //      in case we want the other results for some logic
            m_RaycastResultCache.Clear();

            _pointerPosition = eventData.pointerCurrentRaycast.worldPosition;
            eventData.delta = _pointerPosition - _lastPointerPosition;
            _lastPointerPosition = _pointerPosition;

            //Debug.Log($"INIT DELTA: {eventData.delta}");

            return eventData;
        }


        public void ClearSelected()
        {
            if (eventSystem.currentSelectedGameObject) {
                eventSystem.SetSelectedGameObject(null);
            }
        }

        private static bool ShouldStartDrag( Vector2 pressPos, Vector2 currentPos, float threshold, bool
             useDragThreshold )
        {
            if (!useDragThreshold)
                return true;
            return (pressPos - currentPos).sqrMagnitude >= threshold * threshold;
        }

        private void ProcessMove( VRPointerEventData e )
        {
            var targetGO = e.pointerCurrentRaycast.gameObject;
            //TODO: Message the sender (controller) of the event
            //Pass OnEnter/OnExit calls down to the UI (for hover effects?)
            // IPointerEnterHandler and IPointerExitHandler
            if (ExecuteEvents.CanHandleEvent<IPointerEnterHandler>(targetGO))
            {
                if (currentEnter != targetGO)
                {
                    //if (currentEnter != null)
                    {
                        currentEnter = null;
                        //ExecuteEvents.Execute(currentEnter, e, ExecuteEvents.pointerExitHandler);
                    }

                    if (targetGO != null)
                    {
                        currentEnter = targetGO;
                        //ExecuteEvents.Execute(currentEnter, e, ExecuteEvents.pointerEnterHandler);
                    }
                }
            }
            HandlePointerExitAndEnter(e, targetGO);
            currentEnter = e.pointerEnter;
        }

        private void ProcessClick( VRPointerEventData e )
        {
            var currentOverGo = e.pointerCurrentRaycast.gameObject;
#if UNITY_EDITOR
            if ((fallbackMock && Input.GetMouseButtonDown(0)) || (!fallbackMock && _clickAction.GetStateDown(InputSource)))
#else
            if (_clickAction.GetStateDown(InputSource))
#endif
            {
                e.pressPosition = e.position;
                e.pointerPressRaycast = e.pointerCurrentRaycast;
                e.pointerPress = null;

                //if hitting a physics-interactable
                if(Physics.RaycastNonAlloc(
                    transform.position,
                    transform.forward,
                    sceneObjectHits,
                    _laserReach,
                    interactionLayer.value,
                    QueryTriggerInteraction.Ignore) > 0)
                {
                    var sceneObjectGO = sceneObjectHits[0].transform.gameObject;
                    HandlePointerExitAndEnter(e, sceneObjectGO);
                    currentEnter = e.pointerEnter;
                    if (ExecuteEvents.CanHandleEvent<IPointerClickHandler>(sceneObjectGO))
                    {
#if PLAYGROUND_DEBUG
                        Debug.Log($"POINTER CLICK on {sceneObjectGO}");
#endif
                        ExecuteEvents.Execute(sceneObjectGO, e, ExecuteEvents.pointerClickHandler);
                    }
                }
                else if (e.pointerCurrentRaycast.gameObject != null)
                {
                    GameObject newPressed = ExecuteEvents.ExecuteHierarchy(currentOverGo, e, ExecuteEvents.pointerDownHandler); //Execute a pointerDownEvent on the target
                    //TODO: Message the sender (controller) of the event
                    if (newPressed == null) //the target was not a handler of IPointerDownEvent, though it might still be one for IClickHandler
                    {
                        newPressed = ExecuteEvents.ExecuteHierarchy(currentOverGo, e, ExecuteEvents.pointerClickHandler);
#if PLAYGROUND_DEBUG
                        if (newPressed != null) Debug.Log($"POINTER CLICK on {newPressed}");
#endif
                    }
                    else
                    {
#if PLAYGROUND_DEBUG
                        Debug.Log($"POINTER DOWN on {newPressed}");
                        if(ExecuteEvents.Execute(newPressed, e, ExecuteEvents.pointerClickHandler))
                        {
                            Debug.Log($"POINTER CLICK on {newPressed}");
                        }
#else
                        ExecuteEvents.Execute(newPressed, e, ExecuteEvents.pointerClickHandler);
#endif
                        //TODO: Message the sender (controller) of the event
                    }
                    if (newPressed != null)
                    {
                        e.pointerPress = newPressed;
                        currentPressed = newPressed;

                        e.pointerDrag = newPressed;
                        currentDragging = newPressed;

                        ClearSelected();

                        if (ExecuteEvents.CanHandleEvent<ISelectHandler>(newPressed))
```

```
                                {
#if PLAYGROUND_DEBUG
                                    Debug.Log($"SELECT on {newPressed}");
#endif
                                    eventSystem.SetSelectedGameObject(newPressed);
                                }
#if PLAYGROUND_DEBUG
                                if(ExecuteEvents.Execute(newPressed, e, ExecuteEvents.initializePotentialDrag))
                                {
                                    Debug.Log($"INIT POTENTIAL DRAG on {newPressed}");
                                }
#else
                                ExecuteEvents.Execute(newPressed, e, ExecuteEvents.initializePotentialDrag);
#endif
                            }
                        }
                    }
                    ProcessDrag(e);

#if UNITY_EDITOR
                    if ((fallbackMock && Input.GetMouseButtonUp(0)) || (!fallbackMock &&
                        _clickAction.GetStateUp(InputSource)))
#else
                    if (_clickAction.GetStateUp(InputSource))
#endif
                    {

                        if (e.pointerPress)
                        {
#if PLAYGROUND_DEBUG
                            if(ExecuteEvents.Execute(e.pointerPress, e, ExecuteEvents.pointerUpHandler))
                            {
                                Debug.Log($"POINTER UP on {e.pointerPress}");
                            }
#else
                            ExecuteEvents.Execute(e.pointerPress, e, ExecuteEvents.pointerUpHandler);
#endif
                            var pointerUpHandler =
                                 ExecuteEvents.GetEventHandler<IPointerClickHandler>(currentOverGo);
                            if (e.dragging && e.pointerDrag != null && pointerUpHandler != e.pointerPress &&
                                 e.eligibleForClick)
                            {
#if PLAYGROUND_DEBUG
                                if(ExecuteEvents.ExecuteHierarchy(currentOverGo, e, ExecuteEvents.dropHandler) !=
                                     null)
                                {
                                    Debug.Log($"DROP on {currentOverGo}");
                                }
#else
                                ExecuteEvents.ExecuteHierarchy(currentOverGo, e, ExecuteEvents.dropHandler);
#endif
                            }

                            e.eligibleForClick = false;
                            e.pointerPress = null;
                            currentPressed = null;
                            e.rawPointerPress = null;

                            if (e.pointerDrag != null && e.dragging)
                            {
#if PLAYGROUND_DEBUG
                                if(ExecuteEvents.Execute(e.pointerDrag, e, ExecuteEvents.endDragHandler))
                                {
                                    Debug.Log($"END DRAG on {e.pointerDrag}");
                                }
#else
                                ExecuteEvents.Execute(e.pointerDrag, e, ExecuteEvents.endDragHandler);
#endif
                            }

                            e.dragging = false;
                            e.pointerDrag = null;
                            currentDragging = null;

                            // redo pointer enter / exit to refresh state
                            // so that if we moused over somethign that ignored it before
                            // due to having pressed on something else
                            // it now gets it.
                            if (currentOverGo != e.pointerEnter)
                            {
```

```
                        HandlePointerExitAndEnter(e, null);
                        HandlePointerExitAndEnter(e, currentOverGo);
                    }

                }
            }
        }
        private void ProcessDrag( VRPointerEventData pointerEvent )
        {

            bool moving = pointerEvent.IsPointerMoving();
            //Debug.Log($"Pointer has {((pointerEvent.pointerDrag != null) ? "a" : "no ")} dragging target");
            //Debug.Log($"Pointer should {(ShouldStartDrag(pointerEvent.pressPosition, pointerEvent.position, eventSystem.pixelDragThreshold, pointerEvent.useDragThreshold) ? "" : "not ")} drag!");

            if (moving && pointerEvent.pointerDrag != null
                && !pointerEvent.dragging
                && ShouldStartDrag(pointerEvent.pressPosition, pointerEvent.position, eventSystem.pixelDragThreshold, pointerEvent.useDragThreshold))
            {

                pointerEvent.dragging = ExecuteEvents.Execute(pointerEvent.pointerDrag, pointerEvent, ExecuteEvents.beginDragHandler);
#if PLAYGROUND_DEBUG
                if(pointerEvent.dragging) Debug.Log($"BEGIN DRAG on {pointerEvent.pointerDrag}");
#endif
            }

            // Drag notification
            if (pointerEvent.dragging && moving && pointerEvent.pointerDrag != null)
            {
                // Before doing drag we should cancel any pointer down state
                // And clear selection!
                if (pointerEvent.pointerPress != pointerEvent.pointerDrag)
                {

#if PLAYGROUND_DEBUG
                    if(ExecuteEvents.Execute(pointerEvent.pointerPress, pointerEvent, ExecuteEvents.pointerUpHandler))
                    {
                        Debug.Log($"POINTER UP on {pointerEvent.pointerPress}");
                    }

#else
                    ExecuteEvents.Execute(pointerEvent.pointerPress, pointerEvent, ExecuteEvents.pointerUpHandler);
#endif
                    pointerEvent.eligibleForClick = false;
                    pointerEvent.pointerPress = null;
                    currentPressed = null;
                    pointerEvent.rawPointerPress = null;
                }
#if PLAYGROUND_DEBUG
                if(ExecuteEvents.Execute(pointerEvent.pointerDrag, pointerEvent, ExecuteEvents.dragHandler))
                {
                    Debug.Log($"POINTER DRAG on {pointerEvent.pointerDrag}");
                }
#else
                ExecuteEvents.Execute(pointerEvent.pointerDrag, pointerEvent, ExecuteEvents.dragHandler);
#endif
            }
        }
    }
}
```

G Gantt Diagram

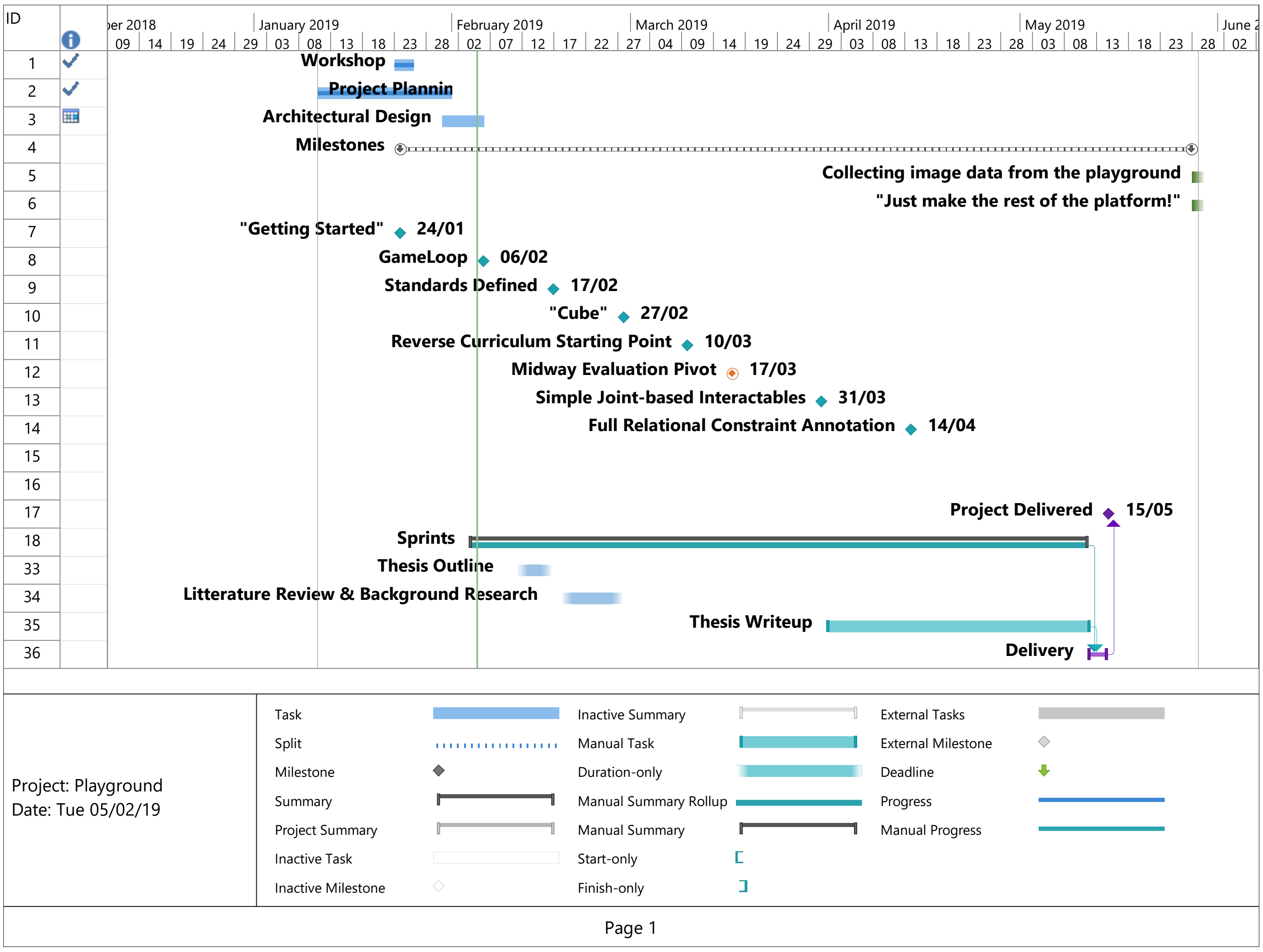


Figure 35: Gantt diagram

H Original Task Description

The following is the original project task description

SINTEF

NTNU
Kunnskap for en bedre verden

Oppdragsgiver

Oppdragsgiver:	SINTEF Ocean AS
Kontaktperson:	John Reidar Mathiassen
Adresse:	Brattørkaia 17C, 7010 Trondheim
Telefon:	+47 934 53 696
Epost:	john.reidar.mathiassen@sintef.no

Neodroid – En virtuell lekegrind for lærende roboter

SINTEF Ocean AS har et prosjekt finansiert av Norges Forskningsråd som heter Neodroid [1]. Målet med prosjektet er å bruker VR som et grensesnitt for å lære roboter til å gjøre handlinger som krever visuell input i form av 3D bilder, på en måte som gjør at roboten får til å gjøre handlingene i den virkelige verden.

Hittil har prosjektet demonstrert læring i VR [2] på enkle oppgaver (plukking av fisk) på en måte som fungerer i den virkelige verden [3]. Det er også utviklet et rammeverk [4] som kobler Unity spillmotoren til deep learning software. Med dette rammeverket har vi begynt å eksperimentere med læring basert på noe som heter 'automated reverse curriculum learning'. Det ser lovende ut på enkle oppgaver, men her trengs det mer forskning og eksperimentering. Derfor ønsker vi flere bacheloroppgaver som sammen lager en virtuell lekegrind for lærende roboter.

Oppgaven

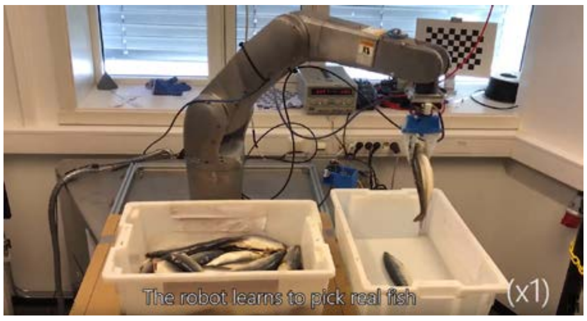
Utvikle en virtuell lekegrind for lærende roboter, og implementere læringsalgoritmer som gjør at roboten kan lære av mennesker som demonstrerer oppgaven til roboten i lekegrinden. Oppgaven kan deles inn i flere deloppgaver eller funksjonalitet som ønskes:

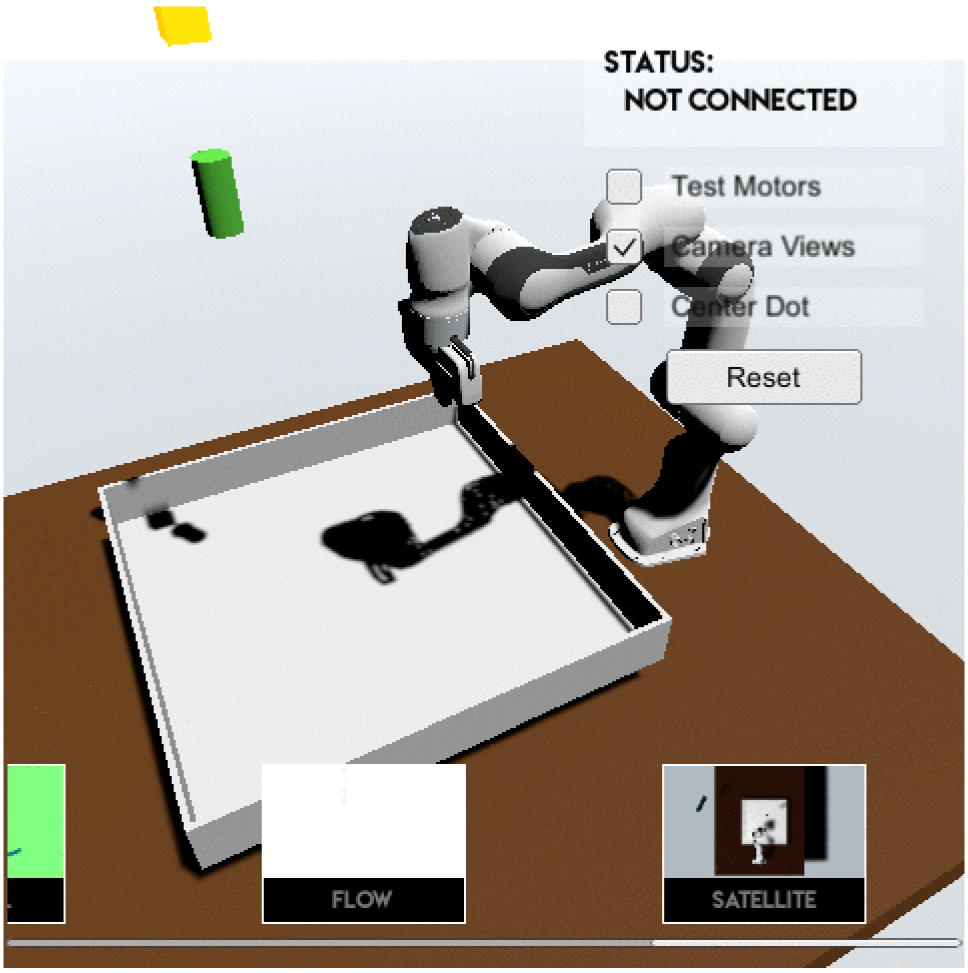
- Konstruksjon av en virtuell lekegrind med flere typer rigide, myke og bøyelige objekter.
- Et VR UX som gjør det mulig for et menneske å demonstrere en oppgave – hvor en enkel oppgave er f.eks. å plukke opp en sild. En mer komplisert oppgave vil deretter være å legge den i en boks, legge lokk på boksen og legge boksen i en eske. Begynn med enkle oppgaver. Ta opp demonstrasjonene slik at det fungerer som et treningssett.
- En læringsalgoritme med 'sparse reward' som lærer å gjøre som mennesket har demonstrert, gitt perfekt informasjon om objektene i lekegrinden. Dvs. at det ikke eksisterer noen virtuelle kamera, kun informasjon om posisjon, orientering og tilstanden til objektene. Fordelen med dette er at det er invariant til hvilken kamera hardware som velges, og det jobber i et tilstandsrom med lavere dimensjonalitet (noe som er en fordel når man jobber med reinforcement learning.

Den virtuelle lekegrinden og læringsalgoritmene kan benytte en modell av en ekte robot, og det bør være mulig å konstruere en slik lekegrind og dets objekter i den virkelige verden, slik at vi kan teste om det roboten har lært i den virtuelle lekegrinden kan overføres til den virkelige verden.

Dersom de ønsker det, vil studentene kunne jobbe tett sammen med en forsker og en PhD student hos SINTEF Ocean AS, samt en student som utviklet Neodroid rammeverket for Unity. Med et slikt samarbeid vil vi sørge for at det som gjøres i Bachelor-oppgavene kan overføres til en virkelig robot.

Bilder

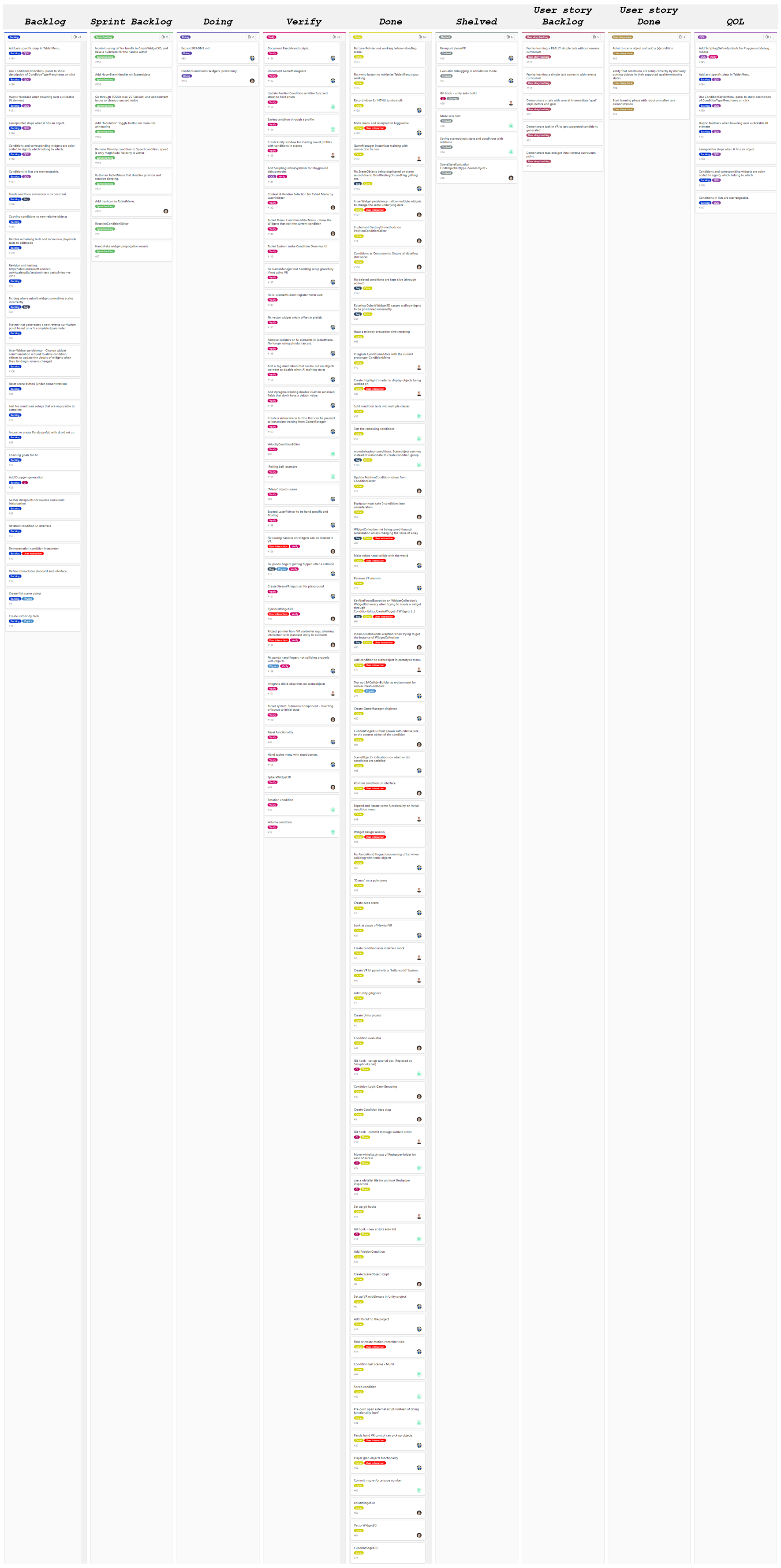




Referanser

1. https://www.forskningsradet.no/prosjektbanken/#/project/NFR/262900
2. https://ieeexplore.ieee.org/document/8324578
3. https://www.youtube.com/watch?v=ox_uJd6yHzo
4. https://github.com/sintefneodroid

I Scrumboard



Backlog

Backlog 24

- Add axis specific slerp in TabletMenu. — Backlog, QOL — #134
- Use ConditionEditorMenu-panel to show description of ConditionTypeMenuItems on click — Backlog, QOL — #139
- Haptic feedback when hovering over a clickable UI element — Backlog, QOL — #131
- Laserpointer stops when it hits an object. — Backlog, QOL — #130
- Conditions and corresponding widgets are color coded to signify which belong to which. — Backlog, QOL — #118
- Conditions in lists are rearrangeable. — Backlog, QOL — #117
- Touch condition evaluation is inconsistent — Backlog, Bug — #126
- Copying conditions to new relative objects — Backlog — #115
- Resolve remaining tests and move non playmode tests to editmode — Backlog — #109
- Revision unit testing: https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2017 — Backlog — #93
- Fix bug where cuboid widget sometimes scales incorrectly — Backlog, Bug — #86
- System that genereates a new reverse curriculum point based on a % completed parameter — Backlog — #85
- Inter-Widget percistency - Change widget communication around to allow condition editors to update the visuals of widgets when their binding's value is changed — Backlog — #108
- Reset scene button (under demonstration) — Backlog — #81
- Test for conditions setups that are impossible to complete — Backlog — #78
- Import or create Panda prefab with droid set up — Backlog — #37
- Chaining goals for AI — Backlog — #35
- Add Doxygen generation — Backlog, CI — #28
- Gather datapoints for reverse curriculum initialization — Backlog — #33
- Rotation condition UI interface — Backlog — #55
- Demonstration condition interpreter — Backlog, User interaction — #12
- Define interactable standard and interface — Backlog — #10
- Create fish scene object — Backlog, Physics — #4
- Create soft-body blob — Backlog, Physics — #11

Sprint Backlog

Sprint backlog 9

- lookinto using ref for handle in CreateWidgetXD, and have a nullcheck for the handle within — Sprint backlog — #158
- Add HoverEventHandler on Sceneobject — Sprint backlog — #156
- Go through TODOs (see VS TaskList) and add relevant issues or cleanup unused todos — Sprint backlog — #151
- Add "hideActor" toggle button on menu for annotating — Sprint backlog — #148
- Rename Velocity condition to Speed condition, speed is only magnitude. Velocity is vector — Sprint backlog — #144
- Button in TabletMenu that disables position and rotation slerping. — Sprint backlog — #137
- Add trashcan to TabletMenu. — Sprint backlog — #136
- RotationConditionEditor — Sprint backlog — #95
- Handshake widget propogation events — Sprint backlog — #67

Doing

Doing 2

- Expand README.md — Doing — #83
- PositionCondition's Widgets' persistency — Doing — #153

Verify

Verify 33

- Document PandaHand scripts. — Verify — #154
- Document GameManager.cs — Verify — #132
- Update PositionCondition serialize func and struct to hold enum — Verify — #152
- Saving conditon through a profile — Verify — #124
- Create Unity window for loading saved profiles with conditions in scenes — Verify — #125
- Add ScriptingDefineSymbols for Playground debug modes — QOL, Verify — #145
- Context & Relative Selection for Tablet Menu by LaserPointer — Verify — #142
- Tablet Menu: ConditionEditorMenu - Show the Widgets that edit the current condition — Verify — #140
- Tablet System: make Condition Overview UI — Verify — #113
- Fix GameManager not handling setup gracefully if not using VR — Verify — #127
- Fix UI elements don't register hover exit. — Verify — #143
- Fix vector widget origin offset in prefab. — Verify — #141
- Remove colliders on UI elements in TabletMenu. No longer using physics raycast. — Verify — #146
- Add a Tag Annotation that can be put on objects we want to disable when AI training starts — Verify — #150
- Add #pragma warning disable 0649 on serialized fields that don't have a default value. — Verify — #149
- Create a virtual menu button that can be pressed to instantiate training from GameManager — Verify — #103
- VelocityConditionEditor — Verify — #96
- 'Rolling ball' example — Verify — #119
- "Many" objects scene — Verify — #82
- Expand LaserPointer to be hand specific and floating. — Verify — #138
- Fix scaling handles on widgets can be rotated in VR. — User interaction, Verify — #129
- Fix panda fingers getting flipped after a collision — Bug, Physics, Verify — #76
- Create SteamVR input set for playground — Verify — #121
- CylinderWidget3D — User interaction, Verify — #98
- Project pointer from VR controller rays, allowing interaction with standard Unity UI elements — User interaction, Verify — #123
- Fix panda hand fingers not colliding properly with objects. — Physics, Verify — #116
- Integrate droid observers on sceneobjects — Verify — #101
- Tablet system: Submenu Component - reverting of layout to initial state — Verify — #110
- Reset functionality — Verify — #40
- Hand tablet menu with reset button. — Verify — #106
- SphereWidget3D — Verify — #97
- Rotation condition — Verify — #39
- Volume condition — Verify — #38

Done

Done 63

- Fix LaserPointer not working before reloading scene. — Done — #133
- Fix menu button to minimize TabletMenu stops working. — Done — #135
- Record video for NTNU to show off. — Done — #128
- Make menu and laserpointer toggleable — Done, User interaction — #120
- GameManager instantiate training with connection to neo — Done — #102
- Fix SceneObjects being duplicated on scene reload due to DontDestroyOnLoadFlag getting set. — Bug, Done — #114
- Inter-Widget persistency - allow multiple widgets to change the same underlying data — Done, User interaction — #107
- Implement DestroyUI-methods on PositionConditionEditor — Done — #74
- Conditions as Components. Ensure all dataflow still works — Done — #104
- Fix deleted conditions are kept alive (through editor?) — Bug, Done — #122
- Rotating CuboidWidget3D causes scalingwidgets to be positioned incorrectly — Bug, Done — #92
- Have a midway evaluation pivot meeting — Done — #99
- Integrate ConditionEditors with the current prototype-ConditionMenu — Done — #70
- Create 'highlight' shader to display objects being worked on — Done, User interaction — #46
- Split condition tests into multiple classes — Done — #57
- Test the remaining conditions — Done — #58
- monobehaviour-conditions: Sceneobject use new instead of instantiate to create condition group — Bug, Done — #105
- Update PositionCondition-values from ConditionEditor — Done — #77
- Evaluator must take 0 conditions into consideration — Done — #66
- WidgetCollection not being saved through serialization unless changing the value of a key — Bug, Done, User interaction — #89
- Make robot hand collide with the world — Done, User interaction — #65
- Remove VR utensils. — Done — #72
- KeyNotFoundException on WidgetCollection's WidgetDictionary when trying to create a widget through ConditionEditor.CreateWidget<TWidget>(...) — Bug, Done, User interaction — #91
- IndexOutOfBoundsException when trying to get the instance of WidgetCollection — Bug, Done, User interaction — #90
- Add condition to sceneobject in prototype menu — Done, User interaction — #73
- Test out SAColliderBuilder as replacement for convex mesh-colliders — Done, Physics — #75
- Create GameManager singleton — Done — #60
- CuboidWidget3D must spawn with relative size to the context object of the condition — Done — #84
- SceneObject's indications on whether it's conditions are satisfied — Done — #88
- Position condition UI interface — Done, User interaction — #54
- Expand and iterate some functionality on initial condition menu — Done — #49
- Widget design session — Done, User interaction — #68
- Fix PandaHand fingers becomming offset when colliding with static objects — Done — #63
- "Donut" on a pole scene — Done — #56
- Create cube scene — Done — #3
- Look at usage of NewtonVR — Done — #31
- Create condition user interface mock — Done — #2
- Create VR UI panel with a "hello world" button. — Done — #41
- Add Unity gitignore — Done — #7
- Create Unity project — Done — #1
- Condition evaluator — Done — #43
- Git hook - set up tutorial doc (Replaced by Setuphooks.bat) — CI, Done — #16
- Condition Logic Gate-Grouping — Done — #47
- Create Condition base class — Done — #9
- Git hook - commit message validate script — CI, Done — #17
- Move whitelist.txt out of Resharper-folder for ease of access — CI, Done — #24
- use a whitelist file for git hook Resharper inspection — CI, Done — #23
- Set up git hooks — Done — #15
- Git hook - new scripts auto lint — CI, Done — #19
- Add PositionCondition — Done — #25
- Create SceneObject script — Done — #8
- Set up VR middleware in Unity project — Done — #6
- Add 'Droid' to the project — Done — #29
- Find or create motion controller claw — Done, User interaction — #14
- Condition test scenes - NUnit — Done — #44
- Speed condition — Done — #42
- Pre-push open external scripts instead of doing functionality itself — Done — #48
- Panda hand VR control can pick up objects — Done, User interaction — #30
- Player grab objects functionality — Done, User interaction — #32
- Commit msg enforce issue number — Done — #62
- PointWidget3D — Done — #60
- VectorWidget3D — Done — #64
- CuboidWidget3D — Done — #71

Shelved

Shelved 6

- Reimport steamVR — Shelved — #45
- Evaluator debugging in annotation mode — Shelved — #87
- Git hook - unity auto build — CI, Shelved — #20
- Make save test — Shelved — #59
- Saving sceneobjects state and conditions with relations — Shelved — #34
- SceneStateEvaluation FindObjectsOfType<SceneObject> — Shelved — #79

User story Backlog

User story backlog 5

- Franka learning a REALLY simple task without reverse curriculum — User story backlog — #112
- Franka learning a simple task correctly with reverse curriculum — User story backlog — #111
- Demonstrate a task with several intermediate 'goal' steps before end goal — User story backlog — #61
- Demonstrate task in VR to get suggested conditions generated. — User story backlog — #51
- Demonstrate task and get inital reverse curriculum point — User story backlog — #52

User story Done

User story done 3

- Point to scene object and add a (x)condition — User story done — #50
- Verify that conditions are setup correctly by manually putting objects in their supposed goal/terminating states — User story done — #94
- Start learning phase with robot arm after task demonstration — User story done — #53

QOL

QOL 7

- Add ScriptingDefineSymbols for Playground debug modes — QOL, Verify — #145
- Add axis specific slerp in TabletMenu. — Backlog, QOL — #134
- Use ConditionEditorMenu-panel to show description of ConditionTypeMenuItems on click — Backlog, QOL — #139
- Haptic feedback when hovering over a clickable UI element — Backlog, QOL — #131
- Laserpointer stops when it hits an object. — Backlog, QOL — #130
- Conditions and corresponding widgets are color coded to signify which belong to which. — Backlog, QOL — #118
- Conditions in lists are rearrangeable. — Backlog, QOL — #117

Figure 36: Scrum Board.

Smedås, Hjerpbakk, Åkerholt, Skaara

Neodroid Playground

NTNU
Kunnskap for en bedre verden

SINTEF