# Computer games testing using Artificial Intelligence

# Requirements and High-level architecture

## 1.1 Entities and their roles

The Ideas discussed so far involve two main entities:

• A ***GameAgent*** entity that plays the game in various environments/levels/setups, with the purpose of testing different things.

The GameAgent should be capable of registering mappings of **(Actions -> ExpectedBehavior)** in a structured way. We have implemented a first proposal version (details in the next subsections), but the format of this mapping is to be determined depending on the use-cases.

• A ***GameStateChecker*** that validates the expected behavior of the GameAgent actions.

## 1.2 Technical Requirements:

A set of examples, grouped by category, of what we could possibly test in game are given below. Note that the images are obtained so far are coming from **a small game prototype using Unreal Engine 4 , developed and work in progress**. more details in the later subsection.

***A. UI Testing:***

⁃ I shoot someone, did the score increase on the HUD?

⁃ I ended the game, did menu X appeared?

⁃ Is the ammo displayed on the screen in sync with the value in the game memory? E.g from our demo, where we check if the ammo displayed on the HUD at a given 2D bounding box is the same with the one expected and stored in the code.

A picture containing building, sitting, table, small

Description automatically generated

⁃ I changed the weapon; do I have the cross? E.g. of basic cross detection in our demo using simple feature matching methods:

A picture containing building, sitting, person, light

Description automatically generated

***B. Animation testing***

⁃ The agent stays in place and watches an AI with walk animation. Is he moving into the right direction over a sequence of N frames? E.g. of pose detection in our demo, based on OpenPose (a state of the art methods using deep NN):

(Original image)

A picture containing building, person, riding, large

Description automatically generated

(Human like detection and skeleton matching using OpenPose method):

A person standing in front of a building

Description automatically generated

Thus, we can **infer the skeleton in either 2d or 3d from images then test automatically the animation’s correctness over a sequence of frames.**

***C. Rendering***

⁃ the agent is being shot, low health, etc. We expect to see some post-processed effects on the screen. Are they visible? WIP for demo.

⁃ I am using binoculars, is camera centered correctly?

***D. Physics***

⁃ The GameAgent pushes an object over a sequence of frames. Does the collision system responds correctly / as expected?

***E. Gameplay***

⁃ GameAgent re-spawned on a map, does it have a clear view, not in front of a wall or something, enemies free?

⁃ GameAgent pressed the button to enter in a car? Does the visuals on the screen looks like he is after N frames?

***F. Sounds***

- Did a specific sound played at in the last N frames?

All these things can be implemented using 2D/3D Segmentation methods and/or with a combination of objects detection such as Yolo / Faster-RCNN models.

## 1.3 Detailed discussion about entities and their communication

***GameAgent –*** not a topic of interest in the first phase of collaboration as we spoke in the meeting. The reason is that your environment already supports scripted agents that check very specific situations. However, part of our individual research, and maybe a topic of interest for you later is to implement these kind of testing agents with no manual tuning. Overall, the current idea is to use:

* Reinforcement Learning methods
* Inputs: camera images, and/or internal game state values, observations.
* Rewards: specific to the type of test. E.g. do you want to test collision? Then train that particular agent rewarding collisions more than other actions.

At runtime, the GameAgent can send to the ***GameStateChecker*** entity the pairs of (**Action -> Expected Behavior)** that it wants to test on the particular frame or on the last N sequence of frames. As shown on the diagram below, on each frame of the game, the GameAgent creates a corpus of tests to do on a frame. Thus, on each frame the framework builds pairs of:

* ***Action***is represented by a dictionary of context variables that the game needs to pass to the other side to do the test properly, i.e. one or more screenshot representing a sequence, recorded game sounds, positions of various items on the screen, etc.
* ***Expected Behavior*** is represented by a dictionary describing the effect expected to happen and must contain all the data necessarily to quantify results. For instance, it could contain a numeric value describing how much ammo should be visible as text in the screen

Look in the implementation example inside the test main function Code/GameStateChecker/LogicLayer.py or on main\_flask\_client.py and main\_flask\_client.py to see how we do the communication and text context and expected behavior for two of the proposed actions: the ammo sync text check and the weapon cross presence check.

![A screenshot of a cell phone

Description automatically generated]()

# Prototype foundation implementation

A couple of functionalities were implemented to have a starting point for development in a team and to prove that some of the methods proposed are really working, at least at a prototype level.

2.1 GameStateChecker **(**current code in the folder with the same name**)** is implemented now as an external service, decoupled from the game. It will be implemented in a combination between Python and C++ code. The plan is to re-use state of the art methods, open source-code as much as possible to speed-up development. When we’ll need custom models or optimizations we jump and solve each individually depending on priorities. It has a layered architecture as shown below.

![A screenshot of a cell phone

Description automatically generated]()

* The ***Foundation layer*** aggregates several models, some generic, other customized for the game itself (e.g. why would we need a custom per game object? because some games have customized effects when the character is almost dying. Trying to detect this in a generic way would fail. The method/ algorithm is the same, but the data the model is trained with is different).
* ***The Logic layer*** - handles services specific to a game leveraging on the layer below it.
* ***The GameStateCheckerCommunication layer –*** is responsible for communication with the game side (i.e. GameAgent entity).

2.2 Communication between GameAgent and GameStateChecker can be handled with different options. Currently, GameStateChecker is deployed on **a Flask/FlaskRest**. It allows then RestAPI to be used as communication/persistence model. This would work great because in Python there are JSON and picklers that allow us to serialize/deserialize data with minimal effort. Also, C++ has ready to use libraries for doing this or can leverage on calling Python code.

The GameAgent side component can be implemented in two ways, each with different ***tradeoffs***.

A picture containing flower

Description automatically generated

1. The C++ code of GameAgent is calling Python code, as show in the Code/TestApp example. In that sample code, we use Python to send test requests using a Flask client app.

Advantage: very quick to implement.

Disadvantage: might be not trivial to build Python source code for Consoles…but maybe your engines already embed Python…

Note: this method doesn’t use Python interpreter at runtime, it just uses Python C++ libraries. It will not be slow, the cost will be just argument parsing and a few more function calls !

1. The C++ code of GameAgent cand send messages through sockets to an intermediate component GameAgentAdapter that handles the communication between consoles and server PC which indeed uses method A. The role of the Adapter instance then is to receive messages from a socket endpoint and convert them to Flask post messages.

The Adapter could be also implemented using higher level libraries such as libcurl or MicrosoftRestAPI, but there is always a problem from my experience with building these on consoles. So sockets seems to be the most common way to do OS/hardware abstraction on this component.

# Demo code.

## 3.1 Unreal 4 Engine integration

Initially, I put the Code given inside an Unreal 4 demo: <https://docs.unrealengine.com/en-US/Resources/SampleGames/ShooterGame/index.html?utm_source=launcher&utm_medium=ue&utm_campaign=uelearn>

We then figured out two things:

1. The test services and their implementation should be game-independent as much as possible. When not possible, we need to create abstract interfaces such that each game can leverage its on specific on top of our framework.
2. We can’t share any of the Unreal demo because they are too big ☺.

We can solve both using Unreal architecture by providing what they call a “Plugin”, and move the demo code there. Other well-known engines are calling this “Extension”, or “Package”. Probably yours has similar concepts. This is currently work in progress. After vacations, you’ll can put your hands on that code too if you want.

## 3.2 How to test the snippets code

What would you need to install:

You can test locally the snippets of code provided by installing:

* Pybind11, Python 3.7.

Preferably download the PDBs for your version and let VS Studio know about them such that you can debug crashes etc.

* Tesseract for windows (and put the path the executable on PATH).
* Install the following packages for python using pip: PyTesseract, opencv2, opencv2-contrib, json, json pickler, flask, requests.

How to test the Python code only:

* Start easy with LayerUtils.py file by running its main functions, if everything works go to the next level.
* Start the main\_flask\_server.py first then main\_flask\_client. If everything works, it means that Python codes correctly at the communication level.
* Start the main\_flask\_server.py and as client this time, run the TestApp.